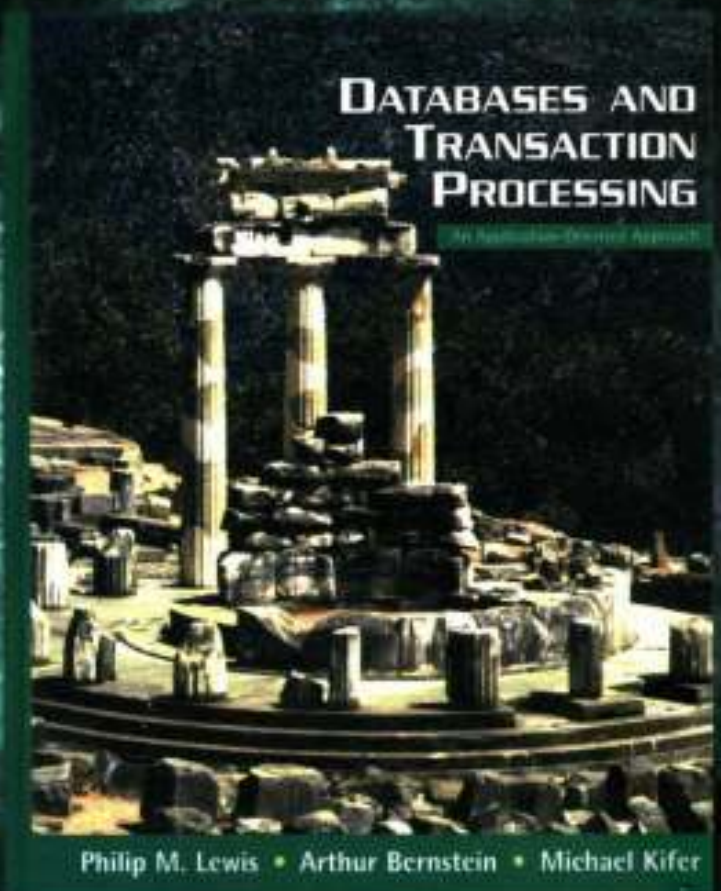
计 算 机 科 学 丛 书

数据库与事务处理

(美) Philip M. Lewis Arthur Bernstein Michael Kifer 著 施伯乐 周向东 方锦城 等译



Databases and Transaction Processing

An Application-Oriented Approach

机械工业出版社
China Machine Press

"This is a great book; This is the book I wish I had written."

——Jim Gray，著名数据库专家，1998年图灵奖获得者

本书系统介绍数据库和事务处理应用的基本概念和实现方法，重点关注如何构建数据库应用。书中始终贯穿关系数据库和关系查询语言的基础理论，为读者熟练掌握这些原理打下坚实的基础。

为了说明数据库和事务处理的概念，作者给出了一个贯穿全书的案例研究。全书围绕如何实现这个案例介绍相关的技术和相应的软件工程概念。

除了介绍关系数据库、SQL和事务的ACID性质之外，本书还深入介绍了以下有关数据库和事务处理的一些前沿论题：

- 嵌入式SQL、SQL/PSM、ODBC、JDBC和SQLJ
- 对象和面向对象数据库，包括SQL:1999、ODMG以及CORBA
- XML和Web上的文档处理
- 触发器和动态数据库
- OLAP和数据挖掘
- 分布式数据库
- TP监控器以及TP监控器如何实现事务的ACID性质
- 不同隔离级别上的并发控制
- 安全性和电子商务

作者简介

Philip M. Lewis Stony Brook的纽约州立大学计算机科学系的教授，于1954年和1956年在麻省理工学院分别获得硕士学位和博士学位，1956～1959年在麻省理工学院电子工程系担任助教，1959～1987年在通用电气公司工作，1987年进入Stony Brook的纽约州立大学任教。



译者简介

施伯乐 现任复旦大学首席教授，上海（国际）数据库研究中心主任，中国计算机学会数据库专业委员会副主任，上海市计算机学会理事长。他有多项研究成果获奖，并结合科研撰写了多部著作和近百篇论文。



www.PearsonEd.com

ISBN 7-111-15718-4

9 787111 157182

HZ BOOKS 华章图书

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

北京市西城区百万庄南街1号 100037

读者服务热线：(010)68995259，68995264

读者服务信箱：hzedu@hzbook.com

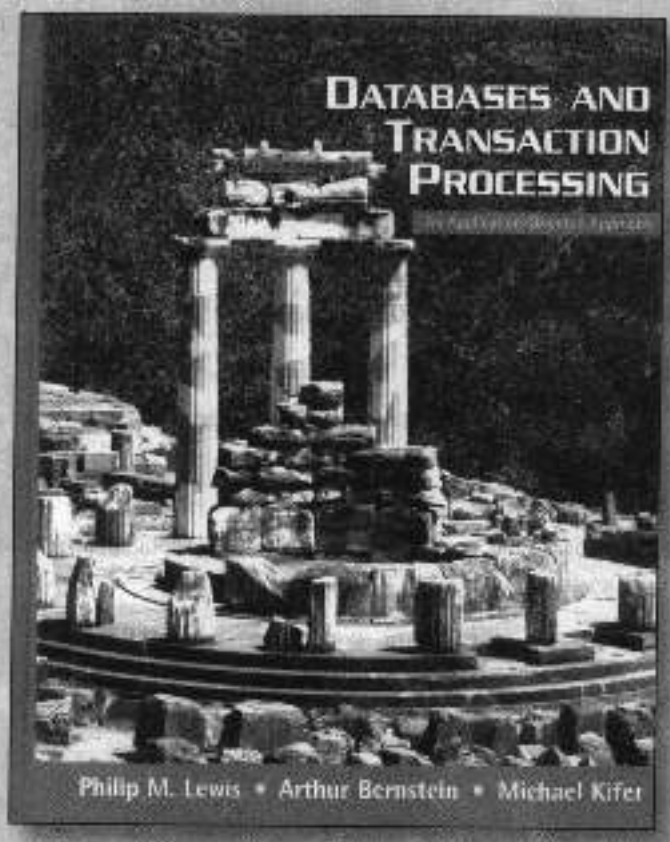
ISBN 7-111-15718-4/TP・4103

定价：85.00 元

计 算 机 科 学 丛 书

数据库与事务处理

(美) Philip M. Lewis Arthur Bernstein Michael Kifer 著 施伯乐 周向东 方锦城 等译



Databases and Transaction Processing

An Application-Oriented Approach

机械工业出版社
China Machine Press

本书对数据库和事务处理应用的设计和实现过程进行了全面、详细的介绍，主要内容涉及数据库和事务处理的基本知识、数据库管理、数据库和事务处理的前沿主题等。本书的重点在于如何设计、实现数据库与事务处理应用，而不是实现数据库系统本身，强调了事务处理在数据库系统中的地位，同时保留了经典关系数据库理论的体系框架。本书篇幅宏大，讲述透彻，适合作为高等院校计算机及相关专业数据库及事务处理课程的教材或参考书，从事数据库管理和开发的技术人员也可以从本书中了解到所需的知识。

Simplified Chinese edition copyright © 2005 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *Databases and Transaction Processing: An Application-Oriented Approach* (ISBN: 0-201-70872-8) by Philip M. Lewis, Arthur Bernstein and Michael Kifer, copyright © 2002.

All rights reserved.

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

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。

版权所有，侵权必究。

本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2003-1998

图书在版编目（CIP）数据

数据库与事务处理 /（美）刘易斯（Lewis, P. M.）等著；施伯乐等译. -北京：机械工业出版社，2005.5

（计算机科学丛书）

书名原文：Databases and Transaction Processing: An Application-Oriented Approach

ISBN 7-111-15718-4

Ⅰ.数… Ⅱ.①刘…②施… Ⅲ.数据库系统 Ⅳ.TP311.13

中国版本图书馆CIP数据核字（2004）第127843号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）

责任编辑：朱　劼

北京中兴印刷有限公司印刷 · 新华书店北京发行所发行

2005年5月第1版第1次印刷

787mm × 1092mm 1/16 · 47印张

印数：0 001-4 000册

定价：85.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall，Addison-Wesley，McGraw-Hill，Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum，Stroustrup，Kernighan，Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及庋藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业的教学度身订造的。其中许多教材均已为M. I. T.，Stanford，U.C. Berkeley，C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下:

电子邮件：hzedu@hzbook.com
联系电话：（010）68995264
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

专家指导委员会

（按姓氏笔画顺序）

尤晋元	王　珊	冯博琴	史忠植	史美林
石教英	吕　建	孙玉芳	吴世忠	吴时霖
张立昂	李伟琴	李师贤	李建中	杨冬青
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
周立柱	周克定	周傲英	孟小峰	岳丽华
范　明	郑国梁	施伯乐	钟玉琢	唐世渭
袁崇义	高传善	梅　宏	程　旭	程时端
谢希仁	裘宗燕	戴　葵		

译 者 序

从关系数据库理论之发轫到商用数据库管理系统的大规模应用，如今，“数据库”已经成为广大读者所熟悉的名词，各种数据库教材不胜枚举。本书把抽象的数据库理论与具体的应用案例结合起来，从数据库应用系统设计的角度，对数据库与事务处理的基本概念与理论进行系统的阐述，作者们对素材的选取和把握，给人留下了深刻印象。

本书的重点在于如何设计、实现数据库与事务处理应用，而不是实现数据库管理系统本身，强调了事务处理在数据库系统中的核心地位。同时，保留了经典关系数据库理论的体系框架。根据数据库与事务处理技术新的发展，本书对Web数据管理、分布式数据库、数据挖掘等新型数据库技术进行了系统解析。书中各章都附有大量的习题与参考文献，便于读者研习。如作者们指出的，本书既可以作为低年级本科生的数据库基础课程教材，也适宜作为研究生的高级数据库与事务处理教材。

本书的第1章～第5章由陈金海教授翻译，第6章～第10章、第12章～第15章由许建军博士翻译，第11章、第20章～第23章由严和平博士翻译，第16章～第19章以及附录由周向东博士翻译，第24章～第27章由方锦城副教授翻译。施伯乐教授、周向东博士对全书的翻译进行了统稿与审校。

本书篇幅宏大，内容丰富，立意新颖，不仅覆盖了数据库、事务处理理论与应用的方方面面，对与数据库相关的软件工程和操作系统的知识也多有涉及。由于译者水平有限，难免有翻译不妥与错误之处，敬请广大读者、同仁批评指正。

译 者

前　言

在当今的信息社会中，数据库和事务处理系统扮演着重要的角色。事实上，我们每天与之交互的任何大型系统，其核心都有一个数据库。这些系统可能是帮助我们处理日常生活中琐碎事务的系统（比如，超级市场付款系统），也可能是与我们的生活息息相关的十分重要的系统（比如，航空交通控制系统）。在今后的几十年中，我们会更加依赖于这些系统，依赖于它们的准确性和高效性。

我们认为，为了设计、建造、维护和管理这些高复杂性的系统，每一个计算机科学家和信息系统专家都应该熟悉这些系统的基本理论概念和工程概念。

本书可以作为下述课程的教材：

- 本科生或研究生的数据库入门课程。
- 本科生或研究生的事务处理课程，该课程为已经学过数据库入门课程的学生开设。
- 高年级本科生或低年级研究生的数据库课程，该课程为已经学过数据库入门课程的学生开设。

只要课程同时涵盖数据库和事务处理的内容，教师就可以选择与两个主题均相关的材料。

本书更关注怎样构建应用程序而不是构建数据库管理系统本身。我们相信大部分学生未来要实现应用程序，只有很少一部分学生会去构建数据库管理系统。因为事务处理提供应用程序访问数据库的机制，所以我们将把数据库的知识放在事务处理中讲解，以此来体现我们的教学重点。此外，本书包含丰富的材料来描述事务用来访问数据库的语言和API，比如嵌入式SQL、ODBC和JDBC。

本书既包括一些传统的主题（关系数据库、SQL和事务的ACID性质），也讨论比较前沿的主题，比如对象和对象-关系数据库，XML和因特网文档处理以及与因特网商务相关的事务问题等。

尽管本书包含很多数据库和事务处理应用实例，但我们主要关心这些主题下的概念而不是某些商务系统和应用程序的细节。因而，在本书的数据库知识部分，我们着重介绍与关系和对象数据模型相关的概念，而非商业数据库管理系统的概念。即使SQL被废弃，这些概念仍是数据库处理的基础。（回想学习COBOL的那代程序员，他们若学习其他语言极为困难。）类似地，在本书的事务处理部分，我们着重介绍ACID性质的概念和实现它们的有关技术问题，而不是某些商业数据库管理系统或TP监控器（TP monitor）。

为加强学生对技术的理解，我们加入一个事务处理应用的案例研究（学生注册系统），该案例将贯穿本书始终。尽管本书中的学生注册系统并不是非常有趣，但它的特点是：所有的学生都作为用户与这样的系统交互过。更重要的是，它是一个内容很丰富的应用，所以我们可以使用它说明很多有关数据库设计、查询处理和事务处理的问题。

本书的独特之处在于，它介绍了实现事务处理应用所需的软件工程概念（使用学生注册系统作为例子）。由于很多信息系统的失败源于项目管理不善和应用不适当的软件工程过程，

所以我们觉得这些主题应该成为教学的重点。我们对软件工程问题的讨论很简短，学生可以选择关于该主题的课程深入学习。我们相信，当学生领会后，他们会更能理解和应用学习材料。因为本课程不是软件工程课程，在课上我们不会对此作全面讲解，而是让学生自己去阅读并且要求他们在课程项目中去实践软件工程。我们将在学生注册系统中探讨相关问题，同时阐明数据库和事务处理的要点。

概述

本书中的材料可供三个学期使用。本书的前半部分可用于数据库课程。对于完成了数据库课程的学生，本书的后半部分着重讲述事务处理和数据库高级主题。在我所就职的学校，我们提供本科（入门的）和研究生（高级的）两种数据库课程，同时也提供本科和研究生两个版本的事务处理课程。

本书分成五个部分，这样教师可以更方便地组织教学材料。我们还有一张“各章之间的关系表”可以使定制课程更加容易。

第一部分：绪论

第1章～第3章包括入门性的材料，适用于初级数据库课程。第1章提供概括性的介绍。第2章简要地说明SQL和事务处理的ACID性质。将这些基础材料放在书的开始部分，就可以免除后面在安排所讨论主题顺序方面的一些束缚。

第3章讨论学生注册系统和与其实现有关的软件工程概念。我们将详细地讨论需求和规格说明文档，以及用来设计图形用户界面的应用软件生成器的使用。在我所就职的学校所开设的数据库入门课程中，我们不在课堂上讲述这些内容，而是要求学生自己去阅读这些材料。在学完这一章后我们开始课程项目，首先要求学生书写规格说明文档。

第二部分：数据库管理

第4章～第15章是初级数据库课程的核心部分。所涵盖的主题有：

- 关系的概念和SQL的DDL特性，包括自动约束检查。
- E-R（实体-联系）模型和模式设计，包括将E-R图转换到关系模式的方法（以及它们的局限性）。
- 关系代数、关系演算和SQL的DML特性，特别要关注通过关系代数和关系演算的语义表达复杂SQL查询。
- 函数依赖和规范化，包括把关系模式分解为3NF、BCNF和4NF的算法。
- 触发器和动态数据库，包括SQL:1999中的触发器。
- 在传统编程语言中嵌入SQL语句，包括嵌入式SQL、动态SQL、ODBC、JDBC和SQLJ。还将讨论最近为存储过程而标准化的语言SQL/PSM。
- 数据和索引的物理组织，包括B^+树、ISAM和散列索引。
- 查询过程和优化，包括选择和联结的算法，以及估算查询计划的代价的方法。

应用到学生注册系统的软件工程问题将贯穿于这些章中。在5.7节，我们讲述系统的数据库设计，包括E-R图和关系模式。第12章讲述设计文档、测试计划文档和完成系统所需的项目

计划。在12.6节，我们介绍详细的设计，并举一个Java/JDBC程序片段的例子，该程序实现系统的某一个事务。

第15章概述随后的几章中关于事务处理的部分内容。如果课程时间允许，它可以用来丰富数据库课程。

第三部分：数据库的高级主题

第16章～第19章包含高级数据库课程的部分主题。高级数据库课程包括第三部分的所有章和第27章的内容。据我们的经验，由于时间紧张，初级数据库课程很难包括第7、8、9、10、11和14章，所以在高级数据库课程中也可以加入这些章。第三部分中的主题有：

- 对象和对象-关系数据库，包括概念模式、ODMG数据库、SQL:1999对象-关系扩展和CORBA。
- Web文档处理的数据库问题，包括对XML Schema、XPath、XSLT和XQuery的详细讨论。
- 分布式数据库，包括异构和同构系统、多重数据库、分段、半联结、全局查询优化、查询设计和分布式数据库设计。
- 联机分析处理和数据挖掘，包括星型模型、CUBE和ROLLUP操作符、联合和分类。

第四部分：事务处理

第20章～第27章和第9章以及第10章的部分内容包括事务处理一个学期课程所需的材料。这些章中的很多实例都涉及学生注册系统的设计，我们在第3章、第12章和5.7节设计该系统。我们要求学生阅读这些材料。

第20章详细讲述事务的ACID性质。第21章、第22章描述多种不同的事务模型和在一个分布式的异构的C/S（客户/服务器模型）环境中事务处理系统的体系结构。其中的主题有：

- 事务模型，包括存储点、链事务、事务队列、嵌套和多级事务、分布式事务、多重数据库系统和工作流系统。
- 事务处理系统的体系结构，包括集中式和分布式数据库的客户/服务器组织方式、双层和三层体系结构、TP监控器和事务管理器。事务远程过程调用和点对点通信，以及它们在事务处理系统的组织中的使用。
- 事务体系结构的实现和因特网事务处理应用程序的模型。

第23章～第26章描述ACID性质中的原子性、隔离性和持久性如何在集中式和分布式系统中实现。其主题包括：

- 抽象数据库的并发控制，包括严格的两段锁、乐观的并发控制、基于时间戳的并发控制、对象数据库的并发控制和为实现不同的事务模型而制定的加锁协议。
- 关系数据库的并发控制，包括不同隔离级别下的加锁协议、在每个隔离级别下正确和不正确的调度实例、粒度加锁、索引加锁和多版本并发控制，其中包括SNAPSHOT隔离级别。
- 日志和恢复，包括提前写日志、转储和检测点。
- 分布式事务，包括两阶段提交协议、全局可串行化、全局死锁以及管理冗余数据的同步和异步算法。

第27章讲述安全和因特网商务。该章涉及下列主题:

- 对称和非对称加密、数字签名、盲签名和证书。
- 用于认证和密钥分发的Kerberos协议。
- 因特网协议，包括用于认证和会话加密的SSL协议、用于安全交易的SET协议、电子现金协议以及保证货物原子性、已验证交付和货币原子性的协议。

第20章～第27章的目标是:

- 使学生明白事务处理系统的体系结构，这样他们可以更好地评估系统提供商提供的功能。
- 描述实现事务ACID性质的代价，该代价可通过系统资源和性能衡量。
- 描述可以减少代价的多种技术。比如粒度锁、索引、反规范化和表分段。
- 描述即便隔离不全时应用程序仍然可以正确执行的情形。例如，在隔离级别比SERIALIZABLE稍弱时事务仍然可以正确执行。

本书附录A中包括的某些系统问题对于理解本书的部分内容很重要。其中包括模块化系统和封装的基本原理、客户／服务器体系结构基础、多路程序设计和线程以及进程间通信基础。如果学生在先前的课程中还没学到上述内容的话，教师可以选择其中的一些内容进行介绍。

章与章之间的关系

为帮助教师根据课程的需要选择本书内容，我们用星号标记一些可跳过（但不影响该章整体结构）的小节。尽管可跳过的小节可能有时被后面的材料所引用，但这些引用是可以被忽略的。此外，练习题会根据其难度级别用一个或两个星号标记。

根据课程目的，有多种使用本书的方式。为指导教师安排课程，表1列出可以包含在5门不同的课程中的章，这5门不同的课程适合不同的学生，强调了不同的教学重点。在这张表里，“是”表明该章中的所有内容都应该包含在课程中。“部分”意味着教师可以只选择其中一部分进行讲授。“阅读”表明该章可以布置为学生的阅读作业。

第1列标记数据库入门课程所需的章。在该课程里，可能只包括第8章关于规范化的部分内容，或许只包括介绍性的章节。类似地，第10章只涵盖关于SQL嵌入宿主语言的不同方法的部分内容——或许只需包含适用于课程项目的一种方法。

第2、3列描述两门紧凑的数据库入门课程。第2列的课程在数据库应用方面来展开课程材料，而第3列更面向理论。第10章在面向应用的材料中提供更加深入的规范化理论、查询语言基础和查询优化。尽管我们把这些材料描述成面向理论的，但是它也同样是面向系统的，因为它包括数据库管理系统设计的问题。在我所就职的学校，我们选择这两列当作本科生课程（如何选择这两种课程可根据教师的兴趣决定）。

第4列描述一门高级数据库课程。在课程刚开始的时候，教师可以给学生复习或补充他认为必须的入门知识。这些知识可以在第7、8、9、10和14章中找到。接下来的课程讲述高级数据库主题和一些在电子商务中关于事务处理的知识。在我所就职的学校，这门课程用于研究生教学，而研究生在本科阶段已经学过数据库基础课程。

第5列描述一门事务处理课程。该课程也假设学生已经学过数据库入门课程。在我所就职的学校，研究生和本科生都开设了这门课程。事务处理的知识需要有相关资料的补充。而这

些资料，比如第9、10章中的一些知识，可能没有被包括进数据库基础课程。

表1　5门课程中所包含的章

章	课程				
	数据库/入门	数据库/应用	数据库/理论	数据库/高级	事务处理
1	是	是	是		是
2	是	是	是		
3	阅读	阅读			
4	是	是	是		
5	是	是	是		
6	是	是	是		
7		部分	是	部分	
8	部分	部分	是	部分	
9		是	是	是	是
10	部分	是		部分	是
11	是	是	是		
12	阅读	阅读			
13	部分	是	是		
14		部分	是	部分	
15	是	是	是		
16				是	
17				是	
18				是	
19				是	
20					是
21					是
22					是
23					是
24					是
25					是
26					是
27				是	是

为进一步调整课程，下面的各章之间关系图可能有所帮助（见图1）。该图指出两种依赖关系。实线说明某一章依赖于另外一章中的除可选章节外的绝大部分知识。而虚线说明依赖较弱，也就是说只依赖于某章中的一小部分概念，这些概念在课堂上可以很快地带过。第27章的依赖比较特殊，它可以安排在事务处理课程的末尾，因为它依赖于第21、22和25章，也可以安排在数据库课程的末尾，因为它依赖于第15章。

补充材料

除本书之外，下面的补充材料有助于教师的教学工作：

- 在线的所有章的PPT演示文档。
- 在线的书中所有图的PPT幻灯片。
- 在线解决方案指南，包括练习题的答案。

• 我们认为读者可能感兴趣的额外的参考资料、注意事项、勘误表、家庭作业、测验等等。

要获得上述补充材料，请访问本书的网址 www.aw.com/cssupport。只有教师（通过联络Addison-Wesley销售代表）能得到解决方案指南和PPT。（读者可登录华章网站下载习题答案和PPT。）

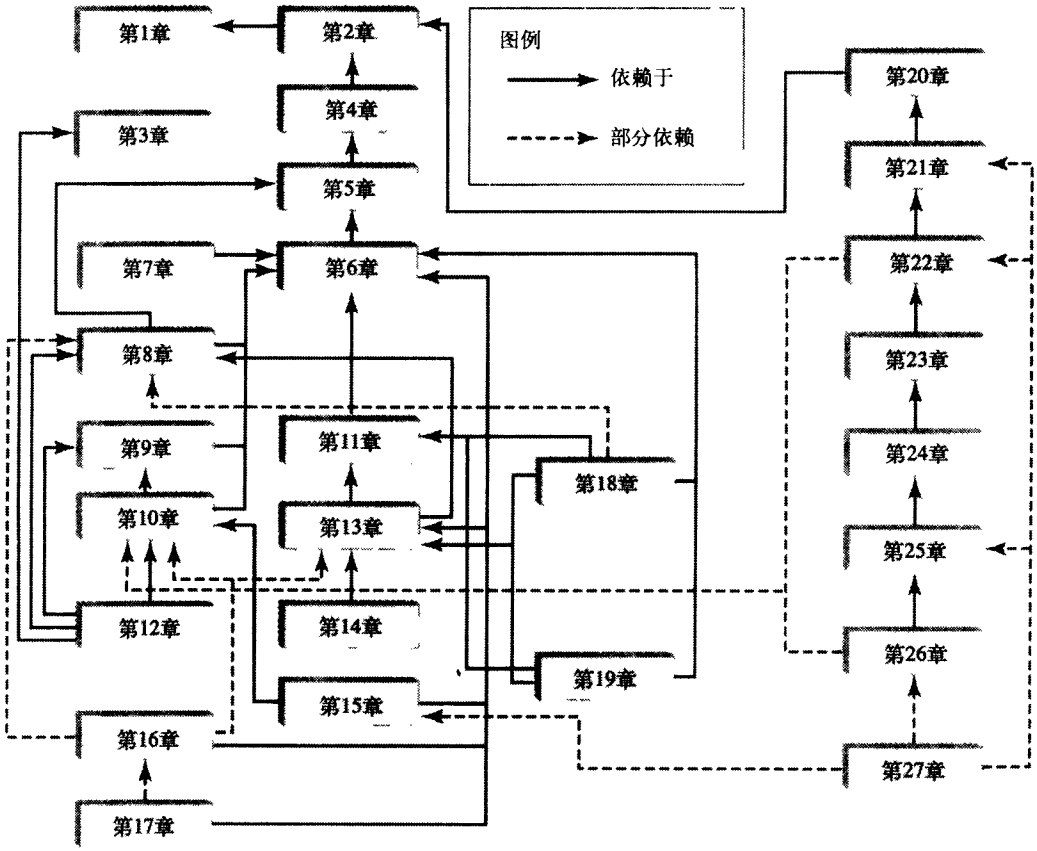


图1 各章之间的关系图

致谢

我们真诚地感谢下列的审校人员，他们的意见和建议极大地改进了本书的质量。

Suad Alagic（Wichita大学）

Catriel Beeri（Hebrew大学）

Rick Cattel（Sun公司）

Jan Chomicki（SUNY Buffalo）

Henry A. Etlinger（罗切斯特理工学院）

Leonidas Fegaras（德克萨斯大学阿灵顿分校）

Alan Fekete（悉尼大学）

Johannes Gehrke（康内尔大学）

Jiawei Han（Simon Fraser大学）

Peter Honeyman（密歇根大学）
Vijay Kumar（Missouri大学）
Jonathan Lazar（Towson大学）
Dennis McLeod（南加利福尼亚大学）
Rokia Missaoui（Quebec大学）
Clifford Neuman（南加利福尼亚大学）
Fabian Pascal（顾问）
Sudha Ram（亚利桑那大学）
Krithi Ramamritham（UMass Amherst and IIT Bombay）
Andreas Reuter（International University in Germany, Bruchsal）
Arijit Sengupta（佐治亚州立大学）
Munindar P. Singh（北卡罗莱纳州立大学）
Greg Speegle（贝勒大学）
Junping Sun（Nova Southeastern大学）
Joe Trubisz（顾问）
Vassilis J. Tsotras（加利福尼亚大学里弗赛德分校）
Emilia E. Villarreal（加利福尼亚州立工业大学）
Gottfried Vossen（Muenster大学）

Don Chamberlin、Daniela Florescu、Jim Gray、Pankaj Gupta、Rob Kelly和C. Mohan很热心地为我们提供额外信息和解答我们的问题，在此向他们表示感谢。

David S. Warren和Radu Grosu在教学中采用了本书的beta版本，并为本书提出了有用的意见和建议。Joe Trubicz不仅在本书手稿完成时进行了审校，并且对许多章的初始版本提供了关键性的意见。

Ziyang Duan、Shiyong Lu、Guizhen Yang和Yan Zhang等学生帮助了我们核对本书中的错误。

十分感谢Stony Brook计算机科学系全体教师，特别是Kathy Germana在工作期间给予我们的帮助。

特别感谢Addison-Wesley的编辑Maite Suarez-Rivas在初期组织内容与方法以及在编写本书的整个过程中所起的重要作用。我们也同时感谢Addison-Wesley的其他员工，Katherine Harutunian、Pat Mahtani、Diane Freed、Regina Hagen、Paul Anagnostopoulos和Jacqui Scarlott出色地完成了本书编辑和发行工作。

最后，我们要感谢我们的妻子Rhoda、Edie和Lora在编写这本书时给予我们很多的支持和鼓励。

目　录

出版者的话
专家指导委员会
译者序
前言

第一部分　绪　论

第1章　数据库和事务概述 ……2
1.1　什么是数据库和事务 ……2
1.2　现代数据库和事务处理系统的特点 ……4
1.3　实现和支持数据库与事务处理系统的主要成员 ……6
1.4　决策支持系统——OLAP和OLTP ……7
1.5　练习 ……8
第2章　进阶 ……10
2.1　案例研究：学生注册系统 ……10
2.2　关系数据库概述 ……10
2.3　怎样使程序成为事务 ……14
2.4　参考书目 ……18
2.5　练习 ……18
第3章　案例研究：开发学生注册系统 ……20
3.1　软件工程方法学 ……20
3.2　需求文档 ……21
3.3　需求分析——新问题 ……26
3.4　应用程序生成器 ……27
3.5　图形用户界面和对象 ……27
3.6　事件和过程 ……30
3.7　访问数据库和执行事务 ……32
3.8　详细说明学生注册系统 ……33
3.9　规格说明文档 ……34
3.10　参考书目 ……35
3.11　练习 ……35

第二部分　数据库管理

第4章　关系数据模型 ……38
4.1　什么是数据模型 ……38
4.2　关系模型 ……40
4.2.1　基本概念 ……41
4.2.2　完整性约束 ……43
4.3　SQL——数据定义子语言 ……48
4.3.1　指定关系类型 ……49
4.3.2　系统目录 ……49
4.3.3　键约束 ……50
4.3.4　处理空缺信息 ……50
4.3.5　语义约束 ……51
4.3.6　用户自定义域 ……53
4.3.7　外键约束 ……54
4.3.8　反应性约束 ……56
4.3.9　数据库视图 ……58
4.3.10　修改已有的定义 ……59
4.3.11　SQL-模式 ……60
4.3.12　访问控制 ……60
4.4　参考书目 ……62
4.5　练习 ……62
第5章　数据库设计I：实体-联系模型 ……64
5.1　E-R方法的概念建模 ……64
5.2　实体和实体类型 ……65
5.3　联系和联系类型 ……67
5.4　E-R方法的高级特性 ……71
5.4.1　实体类型层次结构 ……71
5.4.2　参与约束 ……74
5.5　一个经纪公司的例子 ……76
5.6　E-R方法的局限性 ……79
5.7　案例研究：学生注册系统的设计 ……82
5.8　参考书目 ……86

5.9 练习 ……86
第6章 查询语言Ⅰ：关系代数和SQL ……88
6.1 关系代数：在SQL的覆盖之下 ……88
6.1.1 基本运算符 ……89
6.1.2 导出运算符 ……96
6.2 SQL的查询子语言 ……101
6.2.1 简单的SQL查询 ……101
6.2.2 集合运算 ……106
6.2.3 嵌套查询 ……108
6.2.4 数据的聚合 ……112
6.2.5 简单查询计算算法 ……117
6.2.6 再论SQL中的视图 ……118
6.2.7 空值的窘境 ……122
6.3 在SQL中修改关系实例 ……123
6.4 参考书目 ……127
6.5 练习 ……127
第7章 查询语言Ⅱ：关系演算和可视化查询语言 ……131
7.1 元组关系演算 ……131
7.2 通过元组关系演算理解SQL ……138
7.3 域关系演算和可视化查询语言 ……140
7.4 可视化查询语言：QBE和PC数据库 …143
7.5 关系代数和关系演算之间的联系 ……148
7.6 SQL：1999中的递归查询 ……150
7.7 参考书目 ……155
7.8 练习 ……155
第8章 数据库设计Ⅱ：关系规范化理论 ……157
8.1 冗余所带来的问题 ……157
8.2 分解 ……158
8.3 函数依赖 ……160
8.4 函数依赖的性质 ……161
8.5 范式 ……165
8.6 分解的性质 ……167
8.6.1 无损分解与有损分解 ……168
8.6.2 依赖保持分解 ……170
8.7 分解BCNF的一个算法 ……173
8.8 3NF模式的合成 ……175
8.8.1 最小覆盖 ……175
8.8.2 通过模式合成的3NF分解 ……177
8.8.3 通过3NF合成的BCNF分解 ……178
8.9 第四范式 ……180
8.10 高级4NF设计* ……183
8.10.1 MVD和它们的性质 ……183
8.10.2 设计4NF的困难性 ……184
8.10.3 如何进行4NF分解 ……187
8.11 范式分解的总结 ……188
8.12 案例研究：学生注册系统的模式精化 ……188
8.13 性能调整问题：是否进行分解 ……190
8.14 参考书目 ……191
8.15 练习 ……192
第9章 触发器和动态数据库 ……195
9.1 触发器处理的语义 ……195
9.2 SQL:1999中的触发器 ……197
9.3 避免链式反应 ……202
9.4 参考书目 ……203
9.5 练习 ……203
第10章 真实世界中的SQL ……205
10.1 在应用程序中执行SQL语句 ……205
10.2 嵌入式SQL ……206
10.2.1 状态处理 ……208
10.2.2 会话、连接和事务 ……209
10.2.3 执行事务 ……210
10.2.4 游标 ……212
10.2.5 服务器存储过程 ……216
10.3 再论完整性约束 ……218
10.4 动态SQL ……219
10.4.1 动态SQL的语句预备 ……220
10.4.2 预备语句和描述符区 ……222
10.4.3 游标 ……224
10.4.4 服务器端的存储过程 ……224
10.5 JDBC和SQLJ ……225
10.5.1 JDBC的基本概念 ……225
10.5.2 预处理语句 ……227
10.5.3 结果集和游标 ……227
10.5.4 获取结果集的信息 ……229
10.5.5 状态处理 ……230

10.5.6 执行事务 ……230
10.5.7 服务器端的存储过程 ……231
10.5.8 示例 ……231
10.5.9 SQLJ: Java的语句级接口 ……231
10.6 ODBC* ……234
10.6.1 预处理语句 ……235
10.6.2 游标 ……236
10.6.3 状态处理 ……238
10.6.4 执行事务 ……238
10.6.5 服务器端的存储过程 ……238
10.6.6 示例 ……239
10.7 比较 ……240
10.8 参考书目 ……240
10.9 练习 ……241
第11章 数据的物理组织和索引 ……243
11.1 磁盘组织 ……243
11.2 堆文件 ……247
11.3 排序文件 ……249
11.4 索引 ……251
11.4.1 聚簇索引与非聚簇索引 ……254
11.4.2 稀疏索引和稠密索引 ……256
11.4.3 包含多个属性的查找键 ……257
11.5 多级索引 ……259
11.5.1 索引顺序访问 ……261
11.5.2 B^+树 ……263
11.6 散列索引 ……269
11.6.1 静态散列 ……269
11.6.2 动态散列算法 ……271
11.7 特殊用途的索引 ……277
11.7.1 位图索引 ……277
11.7.2 联结索引 ……278
11.8 调整问题: 为一个应用选择索引 ……278
11.9 参考书目 ……279
11.10 练习 ……279
第12章 案例研究: 实现学生注册系统 …282
12.1 设计文档 ……282
12.1.1 文档结构 ……283
12.1.2 设计评审 ……284
12.2 测试计划 ……285
12.3 项目计划 ……287
12.4 编程 ……289
12.5 渐进式开发 ……290
12.6 学生注册系统的设计和编程 ……291
12.6.1 完成数据库设计: 完整性约束 …291
12.6.2 设计注册事务 ……293
12.6.3 部分注册事务程序 ……295
12.7 参考书目 ……297
12.8 练习 ……297
第13章 查询处理基础 ……298
13.1 外部排序 ……298
13.2 计算投影、并和差 ……301
13.3 计算选择 ……303
13.3.1 具有简单条件的选择 ……303
13.3.2 存取路径 ……304
13.3.3 具有复杂条件的选择 ……306
13.4 计算联结 ……307
13.4.1 用嵌套循环来计算联结 ……307
13.4.2 排序-合并联结 ……309
13.4.3 散列联结 ……310
13.5 多关系联结 ……311
13.6 计算聚合函数 ……313
13.7 调优问题: 对物理数据库设计的影响 ……313
13.8 参考书目 ……314
13.9 练习 ……314
第14章 查询优化概述 ……316
14.1 查询处理概述 ……316
14.2 基于代数等价的启发式优化 ……317
14.3 估计查询执行计划的代价 ……320
14.4 估计输出的大小 ……326
14.5 选择计划 ……327
14.6 调整问题: 对查询设计的影响 ……330
14.7 参考书目 ……332
14.8 练习 ……333
第15章 事务处理概述 ……336
15.1 隔离性 ……336
15.1.1 可串行化 ……337
15.1.2 两段锁 ……338

15.1.3 死锁 ……340
15.1.4 关系数据库中的锁 ……341
15.1.5 隔离级别 ……342
15.1.6 锁粒度和意向锁 ……344
15.1.7 用意向锁的可串行化封锁策略 …345
15.1.8 总结 ……346
15.2 原子性和持久性 ……346
15.2.1 先写日志 ……347
15.2.2 从大规模存储器失效中恢复 ……348
15.3 实现分布式事务 ……349
15.3.1 原子性和持久性——两阶段提交协议 ……350
15.3.2 全局可串行性和死锁 ……351
15.3.3 复制 ……352
15.3.4 总结 ……353
15.4 参考书目 ……353
15.5 练习 ……354

第三部分 数据库的高级主题

第16章 对象数据库 ……356
16.1 关系数据模型的缺点 ……356
16.2 发展历史 ……361
16.3 概念上的对象数据模型 ……363
16.3.1 对象和值 ……363
16.3.2 类 ……364
16.3.3 类型 ……365
16.3.4 对象-关系数据库 ……367
16.4 ODMG标准 ……367
16.4.1 ODL: ODMG对象定义语言 ……370
16.4.2 OQL: ODMG 对象查询语言 ……374
16.4.3 ODMG中的事务 ……377
16.4.4 ODMG中的对象操纵 ……378
16.4.5 语言绑定 ……378
16.5 SQL: 1999中的对象 ……382
16.5.1 行类型 ……382
16.5.2 用户定义类型 ……383
16.5.3 对象 ……384
16.5.4 查询用户定义类型 ……385
16.5.5 更新用户定义类型 ……385
16.5.6 引用类型 ……387
16.5.7 集合类型 ……389
16.6 公共对象请求代理体系结构 ……389
16.6.1 CORBA基础 ……390
16.6.2 CORBA和数据库 ……394
16.7 小结 ……398
16.8 参考书目 ……398
16.9 练习 ……399
第17章 XML 和 Web数据 ……401
17.1 半结构化数据 ……401
17.2 XML概述 ……403
17.2.1 XML元素和数据库对象 ……406
17.2.2 XML属性 ……407
17.2.3 命名空间 ……409
17.2.4 文档类型定义 ……412
17.2.5 DTD作为数据定义语言的不足 …414
17.3 XML Schema ……415
17.3.1 XML Schema和命名空间 ……416
17.3.2 简单类型 ……418
17.3.3 复杂类型 ……422
17.3.4 一个完整的Schema文档 ……428
17.3.5 完整性约束 ……431
17.4 XML查询语言 ……436
17.4.1 XPath: 一种轻量的XML查询语言 ……436
17.4.2 XSLT: XML的一种转换语言 ……442
17.4.3 XQuery: XML的一个功能完善的查询语言 ……450
17.4.4 小结 ……464
17.5 参考书目 ……464
17.6 练习 ……465
第18章 分布式数据库 ……469
18.1 应用设计者对数据库的观点 ……470
18.2 在不同数据库中分布数据 ……472
18.2.1 分段 ……472
18.2.2 更新和分段 ……474
18.2.3 复制 ……475
18.3 查询策略 ……476
18.3.1 全局查询优化 ……476

18.3.2 多数据库系统的策略 ……481
18.3.3 调整问题：分布式环境下的数据库设计和查询计划 ……481
18.4 参考书目 ……482
18.5 练习 ……482
第19章 OLAP和数据挖掘 ……484
19.1 OLAP和数据仓库 ……484
19.2 OLAP应用的多维模型 ……485
19.3 聚合 ……488
19.3.1 下钻、上卷、切片和切块 ……488
19.3.2 CUBE操作符 ……490
19.4 ROLAP和MOLAP ……494
19.5 实现中的一些问题 ……495
19.6 数据挖掘 ……495
19.7 数据仓库的数据载入 ……499
19.8 参考书目 ……500
19.9 练习 ……500

第四部分 事务处理

第20章 事务的ACID性质 ……504
20.1 一致性 ……504
20.2 原子性 ……506
20.3 持久性 ……507
20.4 隔离性 ……508
20.5 事务的ACID性质 ……510
20.6 参考书目 ……512
20.7 练习 ……512
第21章 事务模型 ……513
21.1 平坦事务 ……513
21.2 提供事务的结构 ……514
21.2.1 存储点 ……514
21.2.2 分布式事务 ……515
21.2.3 嵌套事务 ……518
21.2.4 多级事务 ……520
21.3 把应用分解成多个事务 ……523
21.3.1 链式事务 ……523
21.3.2 用可恢复队列调度事务 ……526
21.3.3 扩展事务 ……529
21.3.4 工作流和工作流管理系统 ……531
21.4 参考书目 ……534
21.5 练习 ……535
第22章 事务处理系统的体系结构 ……537
22.1 集中式系统中的事务处理 ……537
22.1.1 单用户系统的组织 ……537
22.1.2 集中式多用户系统的组织 ……538
22.2 分布式系统上的事务处理 ……539
22.2.1 分布式系统的组织 ……540
22.2.2 会话和上下文信息 ……544
22.2.3 队列事务处理 ……545
22.3 异构系统和TP监控器 ……546
22.3.1 事务管理器 ……547
22.3.2 TP监控器 ……548
22.4 TP监控器：通信和全局原子性 ……550
22.4.1 远程过程调用 ……551
22.4.2 对等通信 ……556
22.4.3 事务中异常情况的处理 ……558
22.5 因特网上的事务处理 ……560
22.5.1 一般的体系结构 ……561
22.5.2 因特网上事务系统的组织 ……563
22.6 参考书目 ……564
22.7 练习 ……564
第23章 隔离性的实现 ……566
23.1 调度和等价调度 ……567
23.1.1 串行化 ……570
23.1.2 冲突等价与观察等价 ……571
23.1.3 串行图 ……572
23.2 可恢复性、级联异常中止和严格性 ……574
23.3 并发控制的模型 ……576
23.4 立即更新的悲观并发控制策略 ……577
23.4.1 避免冲突 ……577
23.4.2 死锁 ……579
23.5 立即更新的悲观并发控制的设计 ……580
23.5.1 锁集和等待集的实现 ……580
23.5.2 两段锁 ……581
23.5.3 锁的粒度 ……582
23.6 对象和语义交换* ……583
23.7 结构化事务模型中的隔离 ……587

23.7.1 存储点 ……587
23.7.2 链式事务 ……588
23.7.3 可恢复队列 ……588
23.7.4 嵌套事务 ……589
23.7.5 多级事务* ……589
23.8 其他的并发控制 ……592
23.8.1 时间戳顺序的并发控制 ……593
23.8.2 乐观的并发控制 ……594
23.9 参考书目 ……597
23.10 练习 ……597
第24章 关系数据库中的隔离性 ……601
24.1 加锁 ……601
24.1.1 幻影 ……602
24.1.2 谓词加锁 ……603
24.2 加锁与SQL隔离级别 ……605
24.2.1 更新丢失、游标稳定性和更新锁…609
24.2.2 案例研究：正确性和非可串行级调度——学生注册系统 ……612
24.2.3 可串行化、SERIALIZABLE和正态的 ……617
24.3 粒度加锁：概念锁和索引锁 ……618
24.3.1 索引锁：无幻影的粒度加锁 ……619
24.3.2 对象数据库里的粒度加锁* ……624
24.4 系统性能的改进 ……625
24.5 多版本并发控制 ……626
24.5.1 只读型的多版本并发控制 ……627
24.5.2 读取一致性的多版本并发控制 …628
24.5.3 SNAPSHOT隔离级别 ……628
24.6 参考书目 ……633
24.7 练习 ……633
第25章 原子性和持久性 ……637
25.1 崩溃、异常中止和介质故障 ……637
25.2 直接型更新系统和先写型日志 ……638
25.2.1 性能和先写型登录 ……641
25.2.2 检测点和恢复 ……644
25.2.3 逻辑型登录和物理逻辑型登录* …648
25.3 延迟更新系统的恢复 ……649
25.4 介质故障的恢复 ……650
25.5 参考书目 ……653
25.6 练习 ……653
第26章 分布式事务的实现 ……655
26.1 ACID特性的实现 ……655
26.2 原子终止 ……656
26.2.1 两阶段提交协议 ……657
26.2.2 两阶段提交协议中故障的处理 …661
26.2.3 格式和协议：X/Open标准 ……664
26.2.4 对等原子提交协议 ……664
26.3 协调的传递 ……665
26.3.1 线性提交协议 ……665
26.3.2 无准备状态的两阶段提交协议 …666
26.4 分布式死锁 ……666
26.5 全局可串行化 ……667
26.6 不能保证全局原子性的场合 ……668
26.7 复制数据库 ……670
26.7.1 同步更新复制系统 ……672
26.7.2 异步更新复制系统 ……674
26.8 现实世界里的分布式事务 ……677
26.9 参考书目 ……677
26.10 练习 ……678
第27章 安全性与因特网商务 ……681
27.1 认证、授权与加密 ……681
27.2 加密 ……681
27.3 数字签名 ……684
27.4 密钥发布与认证 ……686
27.4.1 Kerberos协议：票据……687
27.4.2 临时串 ……690
27.5 授权 ……690
27.6 已认证的远程过程调用 ……692
27.7 因特网商务 ……693
27.7.1 SSL协议：证书……693
27.7.2 SET协议：对偶签名 ……695
27.7.3 货物原子性、托管与已认证交付…698
27.7.4 电子现金：盲签名 ……700
27.8 参考书目 ……705
27.9 练习 ……705

附 录

附录A 关于系统的问题 ……708
附录B 参考文献 ……716

第一部分

绪　论

本书的介绍性部分由三章组成。

在第1章中，我们会介绍本书包含的大致内容，来激发读者对数据库与事务处理的兴趣。

第2章将介绍许多在数据库和事务处理领域的基本的技术性概念，包括SQL语言和事务的ACID性质。我们会在本书的后面详细讲解这些概念。

第3章开始讨论一个案例研究：学生注册系统，该案例将贯穿于本书始终。我们还会讨论许多涉及实现该系统的软件工程概念，并给出该系统的需求文档。

第1章 数据库和事务概述

1.1 什么是数据库和事务

在度假期间，你在东京一家百货公司的收款台前面，把信用卡递给店员，然后焦急地等待着你的购买被批准。在你等待的短短几秒钟内，购买信息已经环游世界，发往一个或多个银行和票据交换所，然后访问和更新数据库，直到系统批准了你的购买为止。每天会有超过两千万的信用卡交易在超过一千万的商家那里通过两万多家银行进行处理。这些交易涉及数以亿计的美金，而期间发生的交易会当作一条记录存储在网络数据库中。这些数据库的正确性、安全性和有效性以及访问数据库事务的正确性和性能特征对整个信用卡业务至关重要。

1. 什么是数据库

准确地说，**数据库**（database）是与某一企业相关的数据项的集合。例如，银行的储户的账户信息。数据库可以存储在Rolodex的卡片里或者文件柜的纸张里，我们尤其对按位或字节存储在计算机里的数据库感兴趣。这样的一个数据库可以**集中**（centralized）于一台电脑或者**分布**（distributed）在多台电脑里，后者可能在地理位置上完全分开。

越来越多的企业依赖于这种数据库。企业内没有纸介质的记录，只有当前情况的最新记录（比如，每个银行顾客的账户的余额）存储在数据库中。很多企业把他们的数据库视为最重要的资产。

例如，一家生产飞机的公司的数据库内含有工程设计、生产过程和十年前生产飞机的部件供应商的信息记录，以及在飞机的使用期限中每次检验的记录。如果在将来的某次检验中发现某个飞机的喷气发动机的涡轮叶片发生故障，公司可以从数据库中确定提供那种特殊引擎的转包商，然后该转包商可以从他的数据库中确定涡轮叶片生产的日期、生产叶片的机器和人员，制造叶片的原材料来源以及生产叶片时质量保证测试的结果。用这种办法，公司可以确定故障的原因，提高未来的飞机生产的质量。有一个详细的历史数据库，又可以从数据库中搜索到关于某架十年前生产的飞机上的一个特定的喷气发动机的一个特定的涡轮叶片的结构的信息，这使得拥有这种数据库的飞机生产商比那些没有数据库的飞机生产商具有更大的战略优势。

在某些情况下，数据库是企业的主要资产。例如，在一家信用卡公司中，当你申请信用卡的时候，信用卡公司就会向他们的数据库咨询。在另一些情况下，数据库信息的准确性对人类生活至关重要。例如，东京机场的航空交通管理系统所用的数据库。

为了能方便地进行访问，数据库往往封装在**数据库管理系统**（Database Management System, DBMS）中。数据库管理系统支持一种高级语言，应用程序员使用这种语言来访问数据库。结构化查询语言（Structured Query Language, SQL）是其中使用得最广泛的语言，本书将介绍这种语言。SQL的特点在于它是一种自然描述语言：程序员只需说出要做什么，由

数据库管理系统来解决怎样高效完成的问题。数据库管理系统解释每条SQL语句并且按其描述执行动作。程序员不必知道数据库的存储细节，不必明确地描述执行访问的算法，也不必关心管理数据库的其他许多方面。

2. 什么是事务

数据库通常存储描述企业当前状态的信息。例如，一家银行的数据库存储每个储户当前的账户余额。当现实世界中某一事件的发生改变了企业的状态，存储在数据库中的信息必须要做相应的改变。在联机数据库管理系统中，这些改变由一种称为**事务**（transaction）的程序实时地完成，当现实世界的事件发生的时候就会执行事务。例如，当一个顾客在银行存款（现实世界的一个事件）时，存款事务就会被执行。每个事务必须始终保持数据库状态与真实世界中企业之间关系的正确性。除了改变数据库的状态，事务自身可能在现实世界中也会发出一些事件。例如，ATM机的取款事务发出供应现金的事件。建立电话连接的事务需要在电话公司的基础设施中获得资源（长途链接的带宽）的分配。

你在东京度假时执行的信用卡核准是事务的一个例子。当你预定航班的时候，航班预定数据库中的事务会被执行。当你通过机场检查护照的时候，移民服务数据库中的事务会被执行。你在旅馆登记的时候，旅馆房间预定数据库的事务就会被执行。甚至当你从旅馆房间打电话到家里报平安的时候，旅馆记账数据库中的事务就被执行并建立电话连接。

其他事务在执行时可能经常涉及到ATM系统、超级市场扫描系统以及大学注册和记账系统。这些事务越来越多地必须访问分布式数据库，即分布在不同地理位置由不同数据库管理系统管理的多重数据库。东京旅馆里的电话事务就是其中一个例子。

3. 什么是事务处理系统

管理事务和控制事务访问DBMS的系统称为TP监控器（TP monitor）。一个**事务处理系统**（Transaction Processing System, TPS）通常由一个TP监控器、一个或多个数据库管理系统和一组包含多个事务的应用程序组成（见图1-1）。数据库是事务处理系统的核心，因为它比任何一个事务的生命周期都要长。越来越多的企业依赖于这些为他们的业务而设置的系统。例如，有人可能会说信用卡事务处理系统就是信用卡业务。

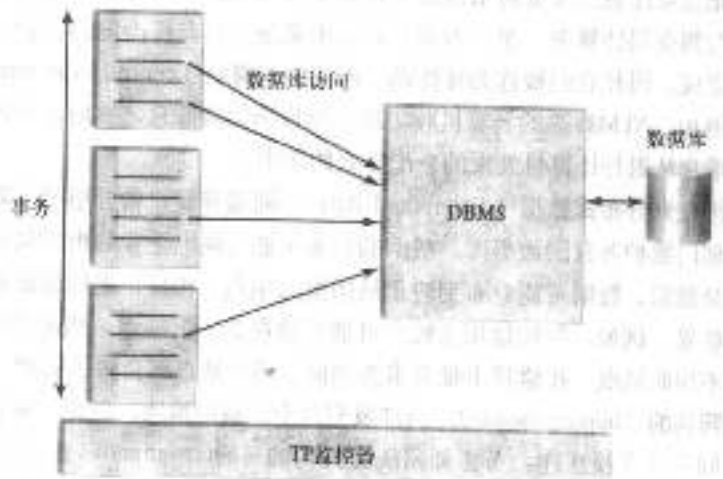


图1-1 事务处理系统的结构

1.2 现代数据库和事务处理系统的特点

现代计算机和通信技术的发展使数据库和事务处理系统的体系结构、设计和使用突飞猛进。下面是新系统与老系统之间的比较。

- 大部分新系统的数据库基于**关系模型**（relational model）。在关系模型中，数据存储在表中，并且可以利用（相对）简单的查询语言（例如SQL）访问数据。使用SQL的程序员不必知道复杂的数据物理布局。相反，老的系统使用复杂数据库模型，称为**网状模型**（network model）。在网状模型里，数据记录通过复杂的指针结构链接，而查询语言通过指针结构浏览数据，这使得查询的设计、编程和测试都十分复杂。
- 很多新系统中的数据库可以存储大型多媒体对象，例如图片和视频片断。如此产生了新一代的应用，比如基于因特网的目录和数字百科全书。老数据库只能存储文字和数字类型的数据，只限于少数种类的应用。
- 大多数新系统是**联机的**（online），然而一些老的系统经常成批执行数据库的文件备份。例如，当你在一台ATM机（联机地）上执行取款操作的时候，在拿到钱之前取款事务将（实时地）扣除你账户中的资金。相反，在老的系统中，取款操作需要在出纳员窗口使用取款单来执行，并且银行在下午2点关门，以便银行数据库在晚上（脱机地）更新数据。
- 大部分新系统允许多个事务**并发访问**（concurrent access）数据库。结果，事务会实时地交叉执行。另一方面，老的系统只允许**顺序访问**（sequential access）数据库，即一个事务只能在前一个事务完成以后才能执行。当事务顺序执行的时候，如果有大量的事务试图访问数据库，事务从提交到完成之间的平均时间延时会急剧地增加。每分钟执行的事务的数量同样也会下降。对于联机系统，这种延时会变得无法忍受，而并发事务处理可以给用户提供更快速的响应时间。
- 很多新系统涉及**分布式计算**（distributed computation）。随着计算机硬件价格的下降，对于一家企业来说，把电脑设备分布在多个场地已经变得比较经济。随着通信价格的下降，将这些站点互连也已经变得很经济。特别是现在用户所用的终端已经相当智能并且能积极地参与到全局计算中。另一方面，在老的系统中只有极少量的计算（如数据格式化）在终端完成，因此它们被称为哑终端，而**中央计算机**（central computer）集中了全部的智能。例如，ATM终端的计算机收集取款操作所需的信息，然后才与银行的计算机交互，最后响应从银行计算机发来的分配现金的请求。
- 很多新系统涉及到**分布式数据**（distributed data）。随着存储设备价格的下降，企业已经可以在不同部门维护各自的数据库。数据的分布可能反映出企业的组织结构，如果企业在地理上是分散的，数据可能分布到应用最频繁的站点。相反，老的系统把所有的数据存储在中心位置。例如，你的信用卡账户可能存储在某一个站点，然而商场的账户存储在另外一个不同的站点。在信用卡批准事务期间，两个站点都会被访问到。
- 老的系统是**同构的**（homogeneous），只涉及到单个厂商的模块。因此，兼容性不会有问题，模块之间可以直接互连。很多新系统是**异构的**（heterogeneous），涉及到不同厂商生产的硬件和软件模块。系统变得“开放”，随之而来的竞争导致厂商生产出更好的产

品。结果，行业接口标准的发展是目前的主要问题。

- 在老的系统中，事务主要由业务人员在办公场所执行。因特网把事务处理带到了家中，这样导致事务处理系统的数量以及被执行的事务的数量呈爆炸性增长。在商业和我们日常生活中，这种新功能将产生深远意义。

上述特性充分提升了数据库和事务处理系统的功能，在许多场合下这将给企业带来重要的新的商业机会。反过来，新功能预示着运行这些系统需要大量的额外需求。

- **高有效性**（availability） 因为系统是联机的，所以当企业在营业的时候，它必须每时每刻都可以使用。在一些企业中，这意味着系统必须一直可以使用。例如，航班预定系统可能要跨很多时区接受从售票处传来的航班预定请求，所以系统永不关闭。相反，脱机银行系统下的事务只在银行营业期间执行。联机系统中的故障会导致营业中断，如果航班预定系统的电脑发生了故障，乘客就不能预定机票。企业容忍故障的能力依赖于企业自身的性质。显然，飞行控制系统比航班预定系统更难忍受系统故障。高有效性系统通常与硬件和软件的配置有关。
- **高可靠性**（reliability） 系统必须准确地反映所有事务的结果。这意味着不仅必须正确地编写事务，而且必须保证不能由于事务并发执行或者事务执行时模块之间的通信而引入错误。而且，大型分布式事务处理系统包括成千上万个硬件和软件模块，所有模块都能正确地运作的可能性不大。除了发生灾难性的故障，系统必须不能丢失任何已完成的事务的结果。例如，银行系统的数据库必须能精确地反映所有存款操作和取款操作完成后的效果，不能因为随后的系统崩溃而丢失任何事务的执行结果。
- **高吞吐量**（high throughput） 因为企业拥有很多必须使用事务处理系统的顾客，所以系统必须有能力每秒钟执行大量的事务。例如，信用卡核准系统在最繁忙的时段可能每秒钟执行上千个事务。正如我们将会看到的，这种需求意味着事务不能顺序执行，而必须并发执行。这样会使系统设计变得很复杂。
- **低响应时间**（low response time） 因为顾客要等待系统的响应，所以系统必须能快速地响应。应用不同，响应需求也会不同。尽管你可能愿意忍受15秒钟时间等待ATM机吐出现金，但是你却期望电话在不多于1秒或2秒的时间内完成连接。此外，在一些应用中，如果在固定的一段时间内没有得到响应，事务就不能正确执行。例如，在工厂自动化系统中，可能要求事务在传送带上的某一元件通过一个特殊的位置之前启动一个设备。这种类型的应用被称为具有**硬实时**（hard real-time）约束。
- **长生命周期**（long lifetime） 事务处理系统很复杂，不能被轻易地替换。它们必须能把单独的硬件或软件模块用新版本（执行得更好或有更多的功能）替换，而周围的系统不必做大量的改动。
- **安全性**（security） 许多事务处理系统包含个人的私人信息（例如，他们所购的商品、他们的信用卡号码、他们看过的视频以及他们的健康记录和财务记录）。因为这些系统可以在许多地方（可能通过因特网）被许多人访问，所以安全变得至关重要。用户必须要经过认证（他们是否就是他们所声称的人），用户只能执行他们被授权可以执行的那些事务（只有银行出纳员可以执行一个产生保付支票的事务），数据库内的信息必须不能被攻击者破坏或读取，并且用户与系统之间传输的信息必须不能被篡改或被窃听。

本书关心的是数据库和事务处理系统的技术方面的知识，特别关注应用程序的设计和实现的内容，包括应用程序数据库的组织，但是不关心算法和实现DBMS与事务处理系统模块的底层数据结构。然而，我们必须非常熟悉这些底层系统，这样我们才能在应用程序中巧妙地使用它们。

1.3 实现和支持数据库与事务处理系统的主要成员

事务处理系统和它所关联的数据库可以是一个非常复杂的由硬件和软件组成的组合物，很多不同类型的人以不同的角色与之进行交互。通过检查这些角色就可以很好地理解什么是事务处理系统。首先考虑事务处理系统的设计和实现所涉及到的人员。

系统分析员　系统分析员与应用系统的顾客合作，规划出系统的正式需求和规格说明文档。他必须既能理解企业的业务规则，又能理解数据库和事务处理系统技术的底层实现，这样的应用系统才既能满足顾客的需要，又能高效地执行。然后，其他角色根据由系统分析员给出的规格说明来详细地设计出数据库格式和访问数据库的事务。

数据库设计师　数据库设计师需指定适合应用系统的数据库结构。数据库存储着描述真实世界应用的现状的信息。数据库结构必须支持事务所要求的访问，而且允许访问及时地执行。

应用程序员　应用程序员实现图形用户界面和系统中的事务。他必须保证事务能维持真实世界应用的状态和数据库状态之间的一致。应用程序员与数据库设计师共同保证管理企业工作的规定能强制执行。例如，在学生注册系统中，注册一门课程的学生人数不能超过分配给这门课程的教室的座位个数。

项目经理　项目经理负责成功地完成实现项目。他需要准备时间进度表和预算，分配人员，以及监督日常的项目运作。项目管理的难度极大。据斯坦迪什集团（The Standish Group，一个信息技术的咨询集团）对超过8000个IT项目的大范围的调查，结果表明只有16%的项目在时间和预算上能成功地完成。其失败的首要原因就是拙劣的项目管理[⊖]。

与运作的事务处理系统交互（而不是构建）的人员如下所示。

用户　用户通过与图形化用户界面交互，执行单独的事务。用户界面必须能适合未来用户的需求。例如，ATM机的用户界面简单，一般人在没有经过训练或指导的情况下通过屏幕上显示的信息就可以执行存款和取款操作。相反，航空公司的职员或旅行社代理人需要经过高级训练才能使用航班预定系统的界面。但无论是上述哪种情况，系统的复杂性都被屏蔽了。

数据库管理员　数据库管理员负责在系统运行时对数据库提供支持。他需要关注的是为数据库分配存储空间，监控和优化数据库性能，监控和控制数据库安全。另外，数据库管理员可能需要修改数据库结构以适应企业内的变化或处理性能瓶颈。

系统管理员　系统管理员负责在系统运行时对系统提供支持。他（她）必须了解以下情况：

- *系统体系结构*　在任何时候，哪些硬件和软件模块连接到系统，以及它们怎样互连？
- *配置管理*　每台机器中各个软件模块的版本是多少？
- *系统状况*　系统是否运行良好？哪些系统和通信链接是可运作的或拥挤的，需要做哪些

⊖ 对于大型公司，成功率降为9%。对于延时完工或超过预算的项目，平均完工时间是计划时间的222%，平均花费是预算的189%。令人惊讶的是，31%的项目在完工之前就被取消。关于这个称为“Chaos”的调查的信息，可参考[Standish 2000]。

工作改善拥挤的状况？系统当前执行情况怎样？

本书主要关注应用层面。所以，我们特别关注系统分析员、应用程序员和数据库设计师这些角色。然而，工作在应用层的人员要想充分利用底层系统的性能，也最好能了解其他的角色。

1.4 决策支持系统——OLAP和OLTP

数据库并不只在事务处理应用领域扮演关键性角色，它还在**决策支持**（decision support）领域扮演重要角色。事务处理使用一个数据库来维护一个映射现实世界状态的精确的模型，而决策支持使用数据库内的信息来指导管理决策。为了阐明这两个领域之间的区别，我们来讨论全国性连锁超级市场中的角色。

1. 事务处理

连锁店的每个超级市场都维护一个数据库来存储它所销售的所有商品的价格和当前库存信息。超市内的结账柜台使用数据库（以条形码扫描器）作为其事务处理系统的一部分。该系统的一个事务可能是“购买三块Campbell香皂和一盒乐之饼干；计算价格，打印收据，更新现金抽屉内的余额，在商店库存中删去这些商品。”顾客希望该事务能在几秒钟内完成。

事务处理系统的主要目标是维护数据库与真实世界状况一致。真实世界中的事件被建模为事务。在该例中，事件是顾客购物，真实世界的状况是商店的库存和现金抽屉内的现金总数。

2. 决策支持

超级市场连锁店的经理可能希望分析每个分店的数据库数据，来帮助他们为整个连锁店做出决策。因为企业期望把他们数据库中的数据转化为对他们制定长期战略目标有用的信息，所以这种决策支持应用变得越来越重要。

决策支持的应用需要查询一个或多个数据库，之后可能要对查询返回的信息做一些数学分析。决策支持应用有时被称为**联机分析处理**（Online Analytic Processing, OLAP），与此形成对比的是我们先前讨论过的**联机事务处理**（Online Transaction Processing, OLTP）应用。

在一些决策支持的应用中，查询很简单，可以作为OLTP应用所使用的本地数据库中的事务来实现。例如，“打印第27分店过去六个月内每周产品销售的报告。”

然而，在许多应用中查询十分复杂，在本地数据库下不能有效地执行。它们可能执行时间过长（因为数据库已经为OLTP事务优化），并导致本地事务（例如，结账事务）执行得很慢。因此，超级市场连锁店要为这样的复杂OLAP查询维护一个单独的数据库。数据库存储所有分店过去十年的销售和库存的历史信息。这些信息从分店数据库在不同的时间内抽取出来，每天更新一次。这样的数据库称为**数据仓库**（data warehouse）。

连锁店经理可以输入复杂的关于数据仓库数据的查询。例如，“在过去五年的冬季月份内，位于东北城市的超级商场的顾客买饼干同时又买香皂的比例是多少？”（可能这两个商品就放置在相近的货架上。）

数据仓库可以存储1TB（10^{12}字节）的数据，所以需要特殊的硬件来维护那些数据。OLAP查询可能很难明确地表达，因而需要使用比OLTP查询功能更强大的查询语言概念。OLAP查询通常没有严格的执行时间的限制，可能执行一次查询需要几个小时。数据仓库可以

通过结构化来加快查询的执行速度。数据库只需要定期更新，因为这种类型的查询没必要精确到分钟级别——即使数据库没有达到100%的准确，查询也可以得到满意的回答。

3. 数据挖掘

经理可能也对结构化不太高的查询感兴趣。例如，“顾客是否喜欢同时购买这些商品？”这样的查询称为**数据挖掘**（data mining）。与OLAP相比，OLAP的查询需要带有特殊信息，而数据挖掘可以被看作是知识发现——尝试从数据库数据中抽取新知识。

数据挖掘查询更难于明确地表达，可能要用到人工智能领域中高深的数学技术。一个查询的执行可能需要多个小时，可能还需要与经理交互来获得额外的信息或者重新构造部分的查询。

数据挖掘有一个被广泛流传的却有可能是虚构的成功故事。便利连锁商店使用上述的查询（“顾客是否喜欢同时购买……”）发现了一个意外的关联：在傍晚，买尿布的男顾客通常会同时购买啤酒——大概这些顾客是晚上在家里要照顾婴儿的父亲。

我们在第19章将进一步讨论OLAP和数据挖掘。

1.5 练习

1.1 举出三个你在过去的一个月内与之交互过的事务处理系统。在每个系统中你执行了哪些事务，现实世界中的哪个事件触发了事务的执行？你的事务导致数据库发生了哪些变化？

1.2 除了书中所提到的，举出三类的严重依赖于事务处理系统的企业。写出每个企业的事务处理系统的简单描述以及企业怎样依赖于这些系统。

1.3 举出超级市场使用联机扫描结账系统后可以获得的三个额外好处。例如，他们可以知道哪些商品被一同购买。

1.4 有些超级市场给用户积分卡，当结账的时候扫描积分卡，并且自动给用户优惠折扣。

a. 举出超级市场从该系统中可获得的三个好处（不包括节省处理优惠券的开销）。例如，他们可以知道你买了什么商品，然后把相关广告寄给你。

b. 写出三个OLAP查询用于处理这些数据。

1.5 解释为什么服装厂经常访问主要百货公司联机结账系统的数据库。

1.6 跨国公司的信息系统通常要保证一天二十四小时一周七天都可用。为什么？

1.7 对下列每个应用需求，给出三个实例：

a. 高可靠性系统

b. 高吞吐量系统

c. 低响应时间系统

1.8 解释下述现象：

a. 为什么你用于保存支票账户的支票簿是一个数据库？

b. 你的支票簿数据库与银行中关于你的支票账户的部分数据库是怎样关联的？

1.9 解释为什么工厂里的联机原料跟踪系统比白天把信息保存在文件中，晚上再把信息输入计算机的脱机系统更精确。（也就是解释脱机系统中的数据为什么在每天早晨更精确而不是一整天都很精确）例如，任何输入错误一旦出现，就可以被检测到并且立即纠正。

1.10 你所在的学校计划一个新的自动学生注册系统。作为可能的用户，你被邀请与系统分析师会面，进行系统的需求分析。

a. 你会推荐哪些对学生有用的事务？包括一些查询事务，学生可以从系统中询问一些信息。

b. 从这些事务中选择一个，并且概述一下用户与事务交互的界面设计。

c. 除了学生还有哪些类型的用户？你认为他们需要哪些事务？

1.11 a. 在上一个习题描述的学生注册系统中，系统管理员可能使用的OLAP查询有哪些？请给出三个例子。

b. 给出一个数据挖掘查询的例子。

1.12 如果你在因特网网站填写一个详细的多页的注册表格，很多网站会提供赠品。为什么？

第2章 进 阶

2.1 案例研究：学生注册系统

你所在的大学希望实现一个学生注册系统，以便学生可以在他们的家庭电脑上注册课程。要求你建造该系统的原型作为本书的项目。注册人员已经准备了如下的系统**目标陈述**（statement of objectives）。

学生注册系统的目标是允许学生和相关的教师完成如下操作：

1) 认证他们为系统的用户。

2) 注册（下一学期的）课程。

3) 获得某一学生情况的报告。

4) 维护关于学生和课程的信息。

5) 学生完成课程后，输入该学生最终的成绩。

上述的简单描述可以作为系统实现项目的出发点，但是它还不够明确和详细，所以不能够作为项目设计和编码阶段的基础。我们将贯穿全书介绍如何开发学生注册场景，并且用它来阐明数据库和事务处理中各种不同的概念。

下一步将结合注册人员、教师和学生来扩展上述的简单描述，使之成为正式的系统**需求文档**（requirements document）。（第3章将讨论需求文档。）然而，在完成这一工作之前，我们将深入介绍数据库和事务处理的一些基本概念。

2.2 关系数据库概述

大多数事务处理系统把数据库放在中心位置。在每一时刻，数据库必须保存有现实世界中由事务处理系统模型化的企业的精确描述，通常是唯一的精确描述。例如，在学生注册系统中，数据库是关于每门课程被哪些学生注册的信息的唯一来源。

我们特别关注使用**关系模型**（relational model）[Codd 1970, 1990]的数据库，其数据存储在**表**（table）中。例如，学生注册系统包含STUDENT表，见图2-1。表拥有一组**行**（row）。在图2-1中，每行包含一名学生的信息。表的每个**列**（column）只描述学生的某一方面的信息。在例子中，列为Id、Name、Address和Status。每列有一个关联类型，称为该列的**域**（domain），每行在该列的值都来自这个域。例如，Id的域是整型，而Name的域是字符串型。

这个数据库模型被称为“关系”模型，是因为它基于关系的数学概念。**数学关系**（mathematical relation）获得的是这样一个概念：不同集合的元素彼此相关。例如，人名（Name）集合里的一个元素John Doe与地址（Address）集合里的一个元素123 Main St.相关，也与Id集合里的一个元素111111111相关。关系是**元组**（tuple）的集合。在STUDENT表的例子

中，我们可以定义一个包含元组（111111111，John Doe, 123 Main St., Freshman）的称为STUDENT的关系。STUDENT关系假定包含描述每个学生信息的元组。

StudId	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

图2-1 STUDENT表。每行描述一个学生

关系可以看作谓词。**谓词**（predicate）是一个或真或假的声明性语句，其真假依赖于其参数的值。例如，谓词“在X天Detroit地区下雨了”的真假依赖于为X参数所选择的值。当把关系看作谓词时，谓词的参数对应于元组里的元素，并且只有当元组正好在关系里时，谓词被定义为真。这样，我们可以定义STUDENT谓词并且说STUDENT（111111111， John Doe, 123 Main St., Freshman）为真。谓词为真意味着Id为111111111的人的名字不会是Bill Smith，因此包含值111111111和Bill Smith的元组不在STUDENT关系中。

表与关系的对应性现在已经清楚了：关系的元组对应于表的行，表的列名也是关系的**属性**（attribute）名。这样，STUDENT表的行可以看作枚举所有满足STUDENT关系（也就是，StudId、Name、Address和Status）的四元组（有四个属性的元组）的集合。

在现实应用中，表可能很大，大学的一张STUDENT表可能超过一万五千行，并且每行可能含有比目前显示的更多的关于每一个学生的信息。除了STUDENT表外，大学的学生注册系统的整个数据库可能包含许多其他的表，每个表又含有许多行，以便存储学生注册的其他方面的信息。因此，大部分的应用数据库包含大量的信息，通常保存在大容量存储设备中。

在大多数应用中，数据库由数据库管理系统控制，而数据库管理系统由销售商提供。当一个应用程序想对数据库执行一个操作时，它需要向DBMS发出一个请求。一个典型的操作可能是从一张或多张表中抽取一些信息，修改一些行，或者增加、删除一些行。

除了数据库中的表可以由数学关系建模外，对表的这些操作也可以建模为相应关系上的数学运算。这样，某个特定的一元操作可能把表T作为一个参数，并且产生一个含有表T的行的子集的结果表。某个特定的二元操作可能把两个表作为参数，并且产生一个含有两个参数表的行的并集的结果表。关于数据库的一个复杂查询可能等同于一个涉及在许多表上进行的多个关系操作的表达式。

通过这样的数学表达，关系操作可以被精确地定义，并且可以证明它们的数学性质（例如交换性和结合性）。正如第14章我们将会看到的，这种数学表达具有很重要的实践意义。商业DBMS包含一个**查询优化器**（query optimizer）模块，该模块可以把查询转换为关系操作表达式，然后使用它们的数学性质来简化这些表达式，这样就可以优化查询的执行。

应用程序希望用DBMS支持的语言来描述DBMS代表它进行的数据访问。我们特别关注SQL，SQL是最通用的数据库语言，为访问关系数据库提供了便利，并得到几乎所有的商业

DBMS的支持。

操纵数据的SQL语句的基础结构十分简单并且易于理解。每条语句使用一个或多个表作为参数并且产生一个表作为结果。例如，为了找到Id为987654321的学生姓名，我们可以使用这样的语句

```
SELECT   Name
FROM     STUDENT
WHERE    Id = '987654321'
```
(2.1)

更准确地说，这个语句要求DBMS从FROM子句指定的表（也就是STUDENT表）中抽取所有满足WHERE子句的条件（也就是Id列的值为987654321）的行，然后从每行中删除SELECT子句中没有出现的列（也就是只保留Name列）。产生的行放置在由语句产生的结果表中。在这个例子中，因为Id唯一，最多只有一行可以满足条件，所以该语句的结果是一个只有一列并且最多只有一行的表。由此可知，FROM子句指定使用哪些表作为输入，WHERE子句指定结果表中行必须满足的条件，SELECT子句指定行的哪些列作为结果表的输出结果。

由上述例子产生的结果表只包含一列并且最多只有一行。下面看一个稍微复杂一些的例子：

```
SELECT   Id, Name
FROM     STUDENT
WHERE    Status = 'senior'
```
(2.2)

返回的结果表（见图2-2）包含两列和多个行：所有大学四年级（senior）学生的Id和姓名。如果希望产生一张包含STUDENT表的所有列但只描述大学四年级学生的表，我们可以使用语句

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

星号（*）是一个简单的速记符，表示列出STUDENT表的所有列。

在一些情况下，用户不关心输出的结果表而关心结果表中的信息。下面的语句就是一个例子：

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

这个语句返回结果表中行的个数（也就是，大学四年级学生的人数）。COUNT称为**聚合**（aggregate）函数，因为它产生结果表中所有行的个数的函数值。需要注意的是，当使用聚合函数时，SELECT语句产生的是单个值而不是一张表。

Id	Name
987654321	Bart Simpson
023456789	Homer Simpson

图2-2　SQL SELECT语句（2.2）返回的数据库表

WHERE子句是SELECT语句中最有趣的部分。它包含一个一般条件来对FROM子句中指定的表的每一行进行判定。行的列值被代入条件中得到一个值为真或假的表达式。如果条件判定为真，该行将被保留下来提供给SELECT子句进行处理，然后存储到结果表中。因此，WHERE子句起到过滤器的作用。

条件可能远比我们迄今为止所见到的更加复杂。条件可以是许多条件项的布尔（Boolean）

组合。例如，如果希望结果是关于Id在一定范围内的大学四年级学生的信息的表，可以用子句

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

也可以使用OR和NOT。此外，SQL语言提供了大量的谓词用于表达特殊的关系。例如，IN谓词测试集合成员。

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

其他的聚合函数和谓词以及更加复杂的WHERE子句将在第6章讨论。

结果表可以包含从若干个基本表中抽取的信息。这样，如果我们有一个TRANSCRIPT表，该表有列StudId，CrsCode，Semester和Grade，则语句

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

产生的结果表，每行包含一个大学四年级学生（senior）的名字、她选的某门课程以及她所获得的成绩。首先注意，结果表中的属性值来自于不同的基本表：Name来自于STUDENT表；CrsCode和Grade来自于TRANSCRIPT表。其次，语句保证TRANSCRIPT表中表示某个学生的行与STUDENT表中对应的行相关。这由WHERE子句的第一个合取项来保证，上例中WHERE语句的条件使得两个表的行的Id值匹配。例如，如果TRANSCRIPT有一行（987654321，CS305，F1995，C），它只能匹配STUDENT表中的Bart Simpson的行，最终在结果表中产生一行（Bart Simpson，CS305，C）。

SQL的一个非常重要的特点就是程序员不必详细地指定DBMS实现某个查询所用的算法。例如，表的定义中经常包含一些辅助性的数据结构，称为索引，它的作用是不必对表进行漫长的扫描就可以定位特定的行。这样，STUDENT表的一个Id列的索引可能是一个包含（Id，pointer）对的列表，其中pointer指向表中对应Id的行。如果建立这样的索引，DBMS将自动使用这些索引来查找满足查询（2.1）的行。如果表包含Status列的索引，则DBMS将使用这个索引来查找满足查询（2.2）的行。如果Status列的索引不存在，则DBMS将自动使用一些其他方法来满足（2.2）。例如，它可能检查表中的每一行来找到所有Status列的值为Senior的行。程序员不必指定使用哪个方法，只需指出结果表必须满足的条件即可。

除了选择使用适当的索引，查询优化器还使用关系运算的性质进一步提升查询处理的效率（这些也不用程序员干涉）。然而，程序员应该对DBMS用于满足查询的策略有一些了解，这样他们才能设计出符合应用程序需要的，能够有效执行的数据库表、索引和SQL语句。

下列三个SQL语句用于修改表的内容。语句

```
UPDATE   STUDENT
SET      Status = 'sophomore'
WHERE    Id = '111111111'
```

更新STUDENT表，使John Doe成为一个大学二年级学生（sophomore）。语句

```
INSERT
INTO     STUDENT (Id, Name, Address, Status)
VALUES   ('999999999', 'Winston Churchill', '10 Downing St',
         'senior')
```

在STUDENT表中插入一个描述Winston Churchill学生的新行。语句

```
DELETE
FROM    STUDENT
WHERE   Id = '111111111'
```

从STUDENT表中删除记录学生John Doe信息的行。这些操作的执行细节也不必程序员加以指定。

STUDENT表本身可以用下面SQL语句执行创建。

```
CREATE TABLE  STUDENT(
Id            INTEGER,
Name          CHAR(20),
Address       CHAR(50),                                    (2.3)
Status        CHAR(10),
PRIMARY       KEY(Id) )
```

在该语句中，我们声明了每个列的名称和存储在列中的数据的域（类型）。我们也声明了Id列作为表的**主键**（primary key）。主键意味着表的每行在该列的值唯一，并且DBMS也会（极有可能）自动地为该列创建一个索引。DBMS会强制执行唯一性约束，不允许任何INSERT或UPDATE语句产生一个Id值已经存在的新行。这是**完整性约束**（integrity constraint）也称为一致性约束（consistency constraint）的一个例子。完整性约束是一个基于应用的对数据库项的值的约束。下一节将详细讨论完整性约束。

我们已经为每个语句类型给出简单的例子，从而突出SQL基本思想在概念上的简单性，但是我们应该知道整个语言还有很多微妙的地方。每种语句类型都有大量的选项，从而进行非常复杂的查询和更新。基于这个原因，掌握SQL需要付出巨大的努力。第4章将继续讨论关系数据库和SQL。

2.3 怎样使程序成为事务

在许多应用中，数据库用于建立真实世界中的一些企业状态的模型。在这样的应用中，事务是一种为了维持企业状态与数据库状态一致的与数据库交互的程序。特别是，事务为了响应真实世界中影响企业状态的事件而执行对数据库的更新。例如，银行中的存款事务。事件可能是顾客给出纳员现金和存单，事务为响应存储事件而更新数据库中顾客的账户信息。

然而，事务不只是一种普通的程序。它们必须满足一些要求，特别是它们执行的方式超过了对非事务程序的通常要求。

一致性（Consistency）　事务访问和更新数据库时必须遵守所有的数据库完整性约束。每个现实企业都有一定的企业制度，这些制度限制了企业的一些可能状态。例如，注册一门课程的学生数不能超过分配给这门课的教室的座位数。如果存在这样的规则，就可以限制数据库的一些可能状态。

可以把限制声明为完整性约束。对应于上述规则的完整性约束断言：数据库中关于课程的注册学生的项的值不能超过关于教室容量的项的值。这样，当注册事务完成后，数据库必须满足这个完整性约束（假设事务开始的时候也满足约束）。

尽管我们还没有设计学生注册系统的数据库，但是我们可以给出一些要存储的数据的设想并且假定一些额外的完整性约束。

- IC0　数据库存储每个学生的Id。这些Id必须唯一。
- IC1　数据库包含每学期必修课程的列表和每个学生完整课程的列表。没有选满必修课

程的学生不能注册相关课程。

- IC2 数据库包含每门课程最多允许注册的学生数和每门课程已经注册的学生数。每门课程已经注册的学生数不能大于这门课程最多允许注册的学生数。
- IC3 从数据库中确定某门课程已经注册的学生数有两种可能方式：这个数值作为总数存储在课程信息中，它也可以从描述学生信息的记录中计算注册该课程的学生记录的总数而获得。这两种确定方式必须产生相同的结果。

除了维护完整性约束外，每个事务必须更新数据库使得新数据库状态反映被建模的现实企业的状态。如果John Doe注册CS305课程，但是注册事务却记录Mary Smith为班级里的新学生，完整性约束虽然满足了，但是新状态不正确。因此，一致性具有两个方面的含义。

> **一致性** 事务设计者可以假设当事务开始执行的时候，数据库状态满足所有完整性约束。设计者有责任保证当执行完成后数据库状态再一次满足完整性约束，并且新状态反映事务规格说明中所描述的变化。

SQL为事务设计者在维护一致性方面提供一些支持。当数据库设计完后，数据库设计者可以指定某些类型的完整性约束并且把它们包含在数据库表的格式声明语句中。SQL语句(2.3)的主键约束就是一个这样的例子。此后，当每个事务执行的时候，DBMS自动检查是否违反任何指定的约束，并且阻止任何违反约束的事务的完成。

原子性（Atomicity） 除了事务设计者要负责一致性外，TP监控器还必须为事务的执行方式提供某种的保证。其中一个这样的条件就是原子性。

> **原子性** 系统必须保证事务要么完全执行完要么没有执行，如果事务没有执行完，则不能产生任何效果（就像它根本没有执行一样）。

在学生注册系统中，一个学生要么已经注册了一门课程，要么他没有注册。部分注册没有意义而且可能导致数据库状态不一致。例如，正如约束IC3所指出的，当学生注册时，数据库中两项信息必须同时更新。如果注册事务只执行到一半，此时一个更新已经完成但是在进行第二个更新之前系统崩溃，结果数据库就会不一致。

当一个事务成功完成，我们说它已经**提交**（commit）。如果事务没有成功完成，我们称它已经**异常中止**（abort），并且系统有责任保证不管事务对数据库做了哪些部分改动，这些改动都必须被撤销，或称为**回退**（roll back）。事务的原子执行意味着每个事务要么提交要么异常中止。

需要注意的是，普通程序没有必要具有原子性。例如，当程序正在更新文件时，系统崩溃了，那么当系统再恢复之后，文件处于部分被更新的状态。

持久性（Durability） 事务处理系统的第二个要求是它不能丢失信息。

> **持久性** 系统必须保证一旦事务提交，事务执行后的效果应永久保持在数据库中，即使计算机或数据库的存储媒质发生故障甚至崩溃，也不能丢失执行结果。

例如，如果你成功注册了一个课程，你一定希望系统即使崩溃还能记住你注册过的课程。

注意，普通程序也没有必要具有持久性。例如，如果一个程序成功地更新完文件后，存储媒质发生故障，文件存储的可能不是更新过的数据。

隔离性（Isolation）　在讨论一致性的时候，我们关注单个事务的执行效果。接下来我们检查多个事务的执行效果。如果一个事务集内的事务在另外一个事务执行完后才开始执行，我们称这个事务集是**顺序**（serially）地执行的。顺序执行的好处是，如果所有的事务是一致的并且数据库初始化时就处于一致性状态下，则顺序执行将保持一致性。当事务集合中的第一个事务开始执行的时候，数据库处于一致性状态下，因为事务是一致的，所以该事务执行完后数据库将仍然是一致的。因为第二个事务开始执行的时候数据库是一致的，所以事务仍将正确执行。如此不断往复。

顺序执行对那些没有高性能需求的应用来说已经足够。然而，许多应用有严格的响应时间和吞吐量要求，唯一能够满足这种需求的是并发执行事务。现代计算机系统有能力同时执行多个事务，我们把这种类型的执行方式称为并发。对服务于多个用户的事务处理系统来说，并发执行是一种恰当的方式。在这种情况下，在任何给定的时刻都有许多正在执行的却只部分完成的事务。

在并发执行中，不同事务的数据库操作高效及时地交替执行。如图2-3所示，事务T_1交替地使用它的局部变量计算并向数据库系统发送请求，在数据库和它的局部变量之间传输数据。请求按序列$op_{1,1}$，$op_{1,2}$产生。我们把这种序列称为一个**事务调度**（transaction schedule）。T_2以类似的方式执行计算。因为两个事务不同时执行，所以操作到达数据库的顺序（称为**调度**（schedule））是两个事务调度的任意排序。图中的调度是$op_{1,1}$，$op_{2,1}$，$op_{2,2}$，$op_{1,2}$。

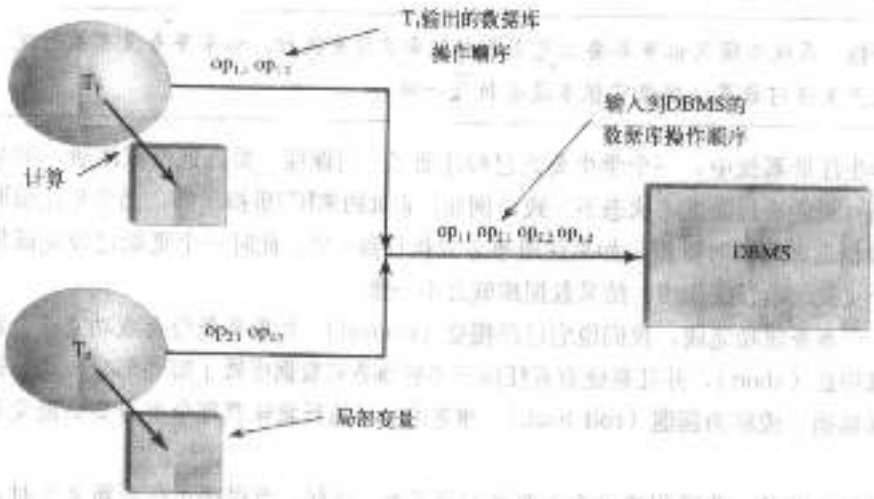


图2-3　在并发调度中两个事务的数据库操作可以交错输出

（注意，$op_{1,1}$第一个到达DBMS，然后是$op_{2,1}$等等）

当事务并发执行的时候，每个事务的一致性不能充分保证数据库正确反映企业的状态。例如，假设T_1和T_2是注册事务的两个实例，由两个想注册同一个课程的学生发起。图2-4显示了这两个事务的一个可能调度，时间从左到右。符号r(cur_reg: n)表示事务读取数据库对象cur_reg的值并返回值n，其中cur_reg记录当前的注册人数（符号w(cur_reg: n)的意义类似）。

图中只显示对cur_reg的访问[⊖]。

T_1: r(cur_reg: 29)	w(cur_reg: 30)
T_2: r(cur_reg: 29)	w(cur_reg: 30)

图2-4 没有相互隔离的两个注册事务的一个调度

假定最多可注册学生的人数是30并且当前的注册人数是29。第一步，两个事务都读取当前注册人数并存储在局部变量中，然后两个事务都认为课程还没有达到最多人数。第二步，每个事务都对自己保存的当前注册人数的局部变量增值；因此，它们都计算得到30。在写操作过程中，他们都把相同的值30写入cur_reg。

两个事务都成功地完成，但是当前注册人数错误地记录为30，而事实上注册人数是31（尽管允许的最大值是30）。我们通常称这种例子为**更新丢失**（lost update），因为其中一个增量丢失了。结果数据库不能反映现实的状态，并且违反了完整性约束IC2。相反，如果事务顺序执行，在T_2允许执行之前T_1已经执行完毕，T_2会发现课程注册人数已达最大并且不允许学生注册。

正如这个例子所演示的，我们必须对并发执行指定一些约束来保证维护数据库的一致性以及企业状态和数据库状态之间的一致性。下面就是这样的一个完全充分的约束条件。

> **隔离性** 即使事务并发执行，调度的整体效果必须等同于在某一次序下事务顺序执行的效果。

这种需求的确切意义将在第15、23和24章中阐明。然而，显而易见的是，如果事务是一致的，并且如果并发调度的整体效果等同于某一顺序调度的效果，那么该并发调度将保持一致性。满足这一条件的并发调度称为**可串行化**（serializable）。

隔离性通常要求事务给某个数据库项加锁才能实现。如果这些锁必须保持一个较长的时间段，则其他的事务可能必须等待，直到持有锁的事务执行完毕为止，这样既增加了响应时间又减少了吞吐量。对于某些应用来说这是无法接受的。为了能适应这些应用，大多数商业DBMS提供隔离性等级的执行选项，这不同于可串行化也不同于串行执行。我们将在第10、15和24章讨论这些选项。

和原子性、持久性一样，普通程序没必要具备隔离性。例如，如果更新多个文件的程序并发执行，更新可能交替执行，其结果可能与顺序执行的结果完全不同。这个结果可能完全不能接受。

ACID性质 区别事务与普通程序的特征通常缩写为ACID性质[Haerder and Reuter 1983]。

- 原子性（Atomic） 事务要么被完全执行，要么根本没有执行。
- 一致性（Consistent） 事务维护数据库的一致性。
- 隔离性（Isolated） 事务集合的并发执行与某个顺序执行的效果一样。
- 持久性（Durable） 事务提交后的效果永久记录在数据库中。

当一个事务处理系统支持ACID性质的时候，数据库就能维持一个一致的、当前最新的现

⊖ 在关系数据库中，r和w代表SELECT和UPDATE语句。

实世界的模型，并且事务一直能给用户提供正确的和最新的响应。

2.4 参考书目

[Codd1970,1990]提出了数据库关系模型。SQL-92标准[SQL1992]解释了SQL语言。[Haerder and Reuter 1983]提出了术语ACID，但是ACID的个别部分在更早期的论文中有所提及，例如，[Gray et al. 1976]和[Eswaran et al. 1976]。

2.5 练习

2.1 给定关系MARRIED、关系BROTHER和关系SIBLING。关系MARRIED由形为（a，b）的元组组成，其中a是丈夫，b是妻子；关系BROTHER由形为（c，d）的元组组成，其中c是d的兄弟；关系SIBLING由形为（e，f）的元组组成，其中e和f互为兄弟姐妹。请描述你怎样定义关系BROTHERINLAW，其元组的形式为（x，y），x是y的姊妹的丈夫。

2.2 设计下面的两张表（除了图2-1之外），它们可能在学生注册系统中使用。注意，同样的学生Id可能在每张表的多行中出现。

a. 一张表实现关系COURSESREGISTEREDFOR，该关系联系学生Id和他已经注册过的课程的数量；

b. 一张表实现关系COURSESTAKEN，该关系联系学生Id、他已经完成的课程和每门课的成绩。

指出各个表对应的谓词。

2.3 写出下述的SQL语句：

a. 返回STUDENT表中所有大学四年级学生（senior）的Id。

b. 从STUDENT表中删除所有大学四年级的学生。

c. 把STUDENT中的所有大学三年级学生（junior）晋级为四年级。

2.4 写出创建TRANSCRIPT表的SQL语句。

2.5 使用TRANSCRIPT表，写出下述的SQL语句：

a. 撤销Id=123456789的学生对2001年秋季CS305课程的注册。

b. 将Id=123456789的学生的2000年秋季CS305课程的成绩改为A。

c. 返回2000年秋季注册CS305课程的所有学生的Id。

2.6 写出SQL语句，返回在2000年秋季CS305课程成绩A的所有学生的姓名（不是学生Id）。

2.7 对于一个银行应用来说，下面的语句是不是检查账户数据的完整性约束？请说明你的理由。

a. 账户的balance列的值大于或等于$0。

b. 账户的balance列的值大于上周这个时候的值。

c. 账户的balance列的值是$128.32。

d. 账户的balance列的值是一个带两个小数位的小数。

e. 账户的social_security_number列非空，并且包含一个九位数。

f. 账户的check_credit_in_use列的值小于或等于total_approved_check_credit列的值。（这些列的意义很明显，就不再加以解释了。）

2.8 除本书给出的例子外，再给学生注册系统的数据库列出五个完整性约束。

2.9 给出一个学生注册系统的例子，它的数据库满足完整性约束IC0～IC3，但是它的状态没有反映现实。

2.10 给出航班预定系统的数据库的五个（可能的）完整性约束。

2.11 航班预定系统中的预定事务用于预定航班，在飞机上预留一个座位，打印一张机票，然后从信用卡账户中扣钱。假设预定数据库的某个完整性约束是：每个航班的预定人数不能超过飞机的座位

数。请说明该系统的事务可能违反下列性质的例子:

a. 原子性

b. 一致性

c. 隔离性

d. 持久性

2.12 请描述下面事件的哪些方面符合或违反事务的原子性和持久性。

a. 利用投币式公用电话打出一个电话。(考虑电话线路忙,没有回音,以及电话号码错误的情况。什么时候事务提交?)

b. 一个结婚典礼。(假设牧师没有出现。假设新郎拒绝说“我愿意。”什么时候事务提交?)

c. 购买一座房子。(假设在签订购买协议之后,业主没能申请到贷款。假设业主最后取消协议。假设两年之后业主没有能力支付贷款,银行将房子收回。)

d. 一场垒球比赛。(假设是雨天。)

2.13 每个学生修完课程后都会得到该课程的成绩。假设系统除了存储成绩外,还保存每个学生积累的学分(GPA)。请描述与这些信息有关的一个完整性约束。如果记录新成绩的事务不是原子性的,请描述该事务将怎样违反约束。

2.14 在上一个习题中,如果两个事务同时记录某一个学生的(不同课程的)成绩,请解释将如何发生更新丢失。

第3章 案例研究：开发学生注册系统

3.1 软件工程方法学

学生注册系统的实现需要按计划进行。由教师、学生和注册人员代表组成的团队已经开过几次会，并将第2章中所给出的非正式的目标陈述修正为正式的需求文档（在3.2节中再介绍）。现在可以开始注册系统项目的工作了。

事务处理系统的实现是一项非常费力的工程。项目必须按时完成，不能超出预算；并且在运行的时候，系统必须要和需求相吻合，要既可靠又有效。项目的文档和编码必须在很长一段时间内可以维护并且可以扩展。最重要的是，系统必须满足用户的需求。

在许多成功与失败的软件项目的基础上，许多过程和方法已经演化为大家公认为的执行这种项目的“好的工程实践”。许多书和课程都是关于软件工程的。这里，我们描述一种方法并将之应用到学生注册系统的开发中去。

项目一般开始于在第2章给出的非正式的目标陈述。对系统的顾客和用户来说，下一步可能在系统分析员的协助下，将这些目标扩展为正式的**需求文档**（requirements document）。需求文档详细描述系统应该做什么，而不是如何去做。在许多环境里，需求文档是对实现者的请求建议书（RFP），它描述顾客希望实现者建立的系统。

规格说明文档（specification document） 实现团队详细分析需求文档，然后写出规格说明文档。规格说明文档是需求文档的扩展版本，更详细地描述系统应该做什么。在许多情况下，规格说明文档是一个合同蓝本，精确地描述实现团队应该去建立的系统。规格说明文档的描述非常精确，以至于可以同时编写和发布用户手册和规格说明文档。下面的例子展示需求文档和规格说明文档之间在细节方面的差异。

- 在需求文档中，列出用户的交互需求，同时给出每个交互打算做什么。在规格说明文档中，指定与每个交互相关的表单。比如，每当某个按钮按下时，以及某个菜单项被访问时，会发生什么情况。
- 需求文档中列出系统必须提供的信息。规格说明文档则包含所有信息项的域。

注意，我们在需求文档和规格说明文档中，讨论的是交互而不是事务。交互是一种模糊的术语，它表明用户想去执行的功能（例如，注册课程、评学分）。在需求分析阶段，我们还不知道会有多少事务将用于实现交互——那是设计的一部分。

当顾客签署规格说明文档后，就可以开始项目的设计。规格说明文档描述系统应该做什么，而设计主要描述如何实现系统的功能。我们会在第12章讨论设计问题。涉及数据库设计的内容在第5章和第8章讨论。在5.7节我们会给出完整的学生注册系统的数据库设计，在12.6节给出注册事务的完整设计和部分代码。

在产生需求文档和规格说明文档上花费如此多的时间和精力的一个原因是：有经验表明

建立一个使顾客满意的系统相当难。一般来说，系统需求非常复杂，顾客描述出的需求很难达到编程所需要的精确度，或者顾客遗漏了重要的细节（例如，在大量的学生已经注册某一课程后，如果该课程将被取消，应该怎么办），又或者指定一些特征，而在系统实现这些特征后，用户又不满意。在项目开始的时候就应该与顾客协调草拟并精化规格说明文档，如果说明文档不符合用户的需求，而不得不在项目将近结束的时候改动系统，将使得项目实现成本高昂且低效。

学生注册系统项目的下一步 在3.2节中，我们将给出顾客提供的学生注册系统的需求文档。注意：一些交互可能只有教师会执行（例如，评学分），而其他的一些交互则是教师和学生都会执行的（例如，打印成绩单）。区别就在于**授权**（authorization）的角色——只有经过授权的人才能执行某些活动。授权应当和**认证**（authentication）区别开来，认证使用密码来确认用户。

同时要注意，在数据库中有两类信息。有些信息是必需的（例如，每个学生的Id号和密码），而有些信息是可选的（这些信息可能无法获得，或者在输入数据的时候还不知道）。例如，某门课程将会分配一间教室，但可能在将课程输入数据库的时候还不知道该教室的房间号。

项目的下一阶段是分析需求文档，产生正式的规格说明文档。3.4节～3.7节将会设计图形用户界面（规格说明文档的必要部分）。我们在3.8节将继续讨论规格说明文档。

3.2 需求文档

I 介绍

学生注册系统的目标是使学生和相关教师完成以下的操作：

A. 对系统的用户身份进行认证。

B. 注册下一学期的课程。

C. 获得某一学生情况的报告。

D. 维护学生和课程信息。

E. 输入学生已经完成的课程的最终成绩。

在文档中，术语“参与”指一个学生正在学习的课程，而“注册”指学生下学期可以选的课程。

II 相关文档

A. 学生注册系统的目标陈述（包括日期和版本号）。

B. 大学本科生公告栏（包括日期）。

III 系统中包括的信息

A. 系统应该[⊖]包括允许使用该系统的每位教师和学生的姓名、Id、密码和身份。身份指

⊖ 需求要进行编号，以便在以后的文档（比如用于测试系统是否满足需求的测试计划）中引用。而且，使用诸如“将会”和“必须”之类的词是必要的。“应当”或者“能”之类的词并不意味着该文档是一个有效力的文档，应当避免使用这些词，除非需求是可选的。例如，在早期的记录需求文档中（即使需求已经被编号）仍然使用“Thou shall not kill”，而不是“Thou should not kill”。

该用户是学生还是教师。密码用来认证用户并且决定该用户以什么身份登录，而Id号是唯一的。在初始化时假设至少有一个教师已经是有效用户。

B. 系统将会包括每位学生的学科记录。

1. 学生修完的每门课程，学生学习某门课程的学期，学生所取得的成绩（成绩可能是A、B、C、D、E、F、I）。
2. 本学期学生参与的每门课程。
3. 学生为下学期所选的每门课程。

C. 系统将会包括所提供的课程的信息。对于每门课程，系统将会包括如下信息：

1. 课程名、课程号（必须唯一）、开设课程的系、所需教材、课时。
2. 课程是春季开设还是秋季开设，或者都开设。
3. 预备课程（每门课程有可能有一些必须先行学习的课程）。
4. 允许参与的最多人数，即选课的最多人数（如果该课程本学期不开设，则不需要指定）；已经注册的学生人数（如果下学期不开设该课程，可以不指定）。
5. 如果一门课程本学期开课，给出开课的日期和时间；如果该课程下学期开课，要给出将开课的日期和时间。可能的取值应该从一些固定的日期中选取（如MWF10）。
6. 教授该课程（不管是本学期还是下学期开）的教师Id（如果指定的学期不开课，则可以不指定；但是必须在开课前指定好）。
7. 为该课程（不管是本学期还是下学期开）分配的教室。（如果指定的学期不开课，则可以不指定；但是必须在开课前指定好）。

所有信息必须和本科生公告栏内所公布的信息一致。

D. 系统应该包括已开设的所有课程的记录，该记录包含课程的开设学期和教师Id。

E. 系统应该包括教室编号和相应的座位数的列表。教室编号是一个唯一的三位整数。

F. 系统应该包括当前学期的标志（如：F1997，S1996）。

Ⅳ 完整性约束

在第Ⅲ部分描述的数据库将会满足下面的完整性约束：

A. Id唯一。

B. 假如在3.2节中的项Ⅲ B2（Ⅲ B3）中，列出已参与（或者注册）某一课程的学生，则该课程必须在项Ⅲ C2中说明在本学期（下学期）开设。

C. 在3.2节的项Ⅲ C4中，参与或者注册某一课程的学生人数不能大于允许的最大参与人数。

D. 在3.2节的项Ⅲ B2（Ⅲ B3）中提及的参与或者注册该课程的学生人数必须和项Ⅲ C4中当前参与或者注册的人数相同。

E. 一名教师不能在同一学期同一时间被安排两门课程。

F. 两门课程不能在给定的学期的同一时间安排在同一个教室开设。

G. 如果一名学生已经参与某一课程，则该学生必须已经完成该课程规定的预备课程，而且成绩不能低于C。

H. 一名学生不能注册在同一时间教授的两门课程。

I. 一个学生不能在特定的学期里注册超过20个学分的课程。

J. 分配给某一课程的教室的座位数必须大于等于该门课程允许的最大登记人数。

K. 一旦成绩（A、B、C、D或F）已经被给定，则它不能被改变到I[⊖]。

V 与系统的交互

与系统的交互被安排为会话。当一个用户执行认证交互时，开启一个会话；当用户执行结束会话交互时，会话结束。在一个会话中，用户执行若干交互。

A. *认证* 无论什么时候开启一个会话，用户必须被认证。认证的目的是验证用户，并确定他是一名学生还是教师，会话中随后的交互取决于该用户的身份。在交互中要输入Id和密码。

B. *注册* 这个交互只能由学生执行。目的是让学生注册下学期要修的课程。输入是课程代号，输出包括下面的内容之一：

1. 如果该学生注册一门课程成功，输出课程号。
2. 学生不能注册的原因。

若发生下列情况（包含在输出中），则注册失败：

1. 学生没有按照要求在该课程的预备课程中获得C以上的成绩，或者现在还没有完成预备课程。
2. 该课程已注册的学生数超过最大注册数。
3. 交互的发起者不是一个学生。
4. 该学生同时注册了另外一门课程。
5. 该同学正在参与该课程或者该同学已经修过该课程，并获得C以上的成绩。
6. 该课程下学期不开设。
7. 该学生已经注册过该课程。
8. 如果该门课程注册成功，该学生注册的课程的学分数将超过20。

C. *注销课程* 交互只能由学生执行。目的是让该学生注销已经在早些时候注册的下学期将开设的某门课程。输入是课程号，如果该学生没有注册过该课程，则注销课程将失败。

D. *取得成绩记录* 交互的目的是产生一个报告，该报告描述一个学生每个学期完成的课程的成绩。该交互使用学生的Id，如果该学生正在执行交互，则该学生不再需要输入Id，因为他只能看到自己的成绩报告，Id已经作为认证的一部分进行了验证。如果一个教师执行该交互并且输入一个无效的学号，则交互失败。报告将包括以下内容：

1. 当前的学期。
2. 学生名字和Id。
3. 按照学期分类的带有授课教师和成绩的课程列表。
4. 学期的GPA和每个学期该学生已经完成的学分数。
5. 到目前为止该学生完成的所有课程的学分数和累积的GPA。

⊖ 这是一个动态完整性约束的例子，它限制对于数据库状态可以进行的修改。而静态完整性约束限制数据库可允许的状态。我们在4.2.2中讨论动态完整性约束。

E. 取得注册课程　交互的目的是产生一个报告，该报告列出某个学生下学期注册的课程。交互时输入学生的Id。如果该学生正在执行交互，则该学生不需要输入Id，因为他只能看到自己的报告，Id已经作为认证的一部分进行了验证。如果一个教师执行该交互并且输入一个无效的Id，则交互失败。报告将包括以下内容:

1. 学生姓名和Id。
2. 课程号和学时。
3. 每门课程的时间安排。
4. 分配的教室（如果可以得到）。
5. 教师（如果可以得到）。

F. 取得参与的课程　交互的目的是产生一个报告，该报告列出某个学生本学期参与的课程。交互时输入学生的Id。如果该学生正在执行交互，则该学生不需要输入Id，因为他只能看到自己的报告，Id已经作为认证的一部分进行了验证。如果一个教师执行该交互并且输入一个无效的Id，则交互失败。报告将包括以下内容:

1. 学生姓名和Id。
2. 课程号和学时。
3. 每门课程的时间安排。
4. 分配的教室。
5. 教师。

G. 学生成绩　交互的目的是给出或者更改一个学生已完成的课程的成绩。交互只可以由授课教师来执行。输入为Id、课程号、学期和成绩。

1. 如果发生下列情况，则交互失败:
   a. 交互不是由本学期指定的授课教师发出。
   b. 学生的Id无效。
   c. 该学生当前没有参与课程，而且以前也没有修过该课程。
   d. 交互可以将成绩（A、B、C、D）改为I。
2. 如果交互成功:
   a. 学生不再作为参与该课程的人出现，只作为已经顺利完成该课程的学生出现。
   b. 如果该课程是学生已经注册的下学期的某门课程的预备课程，而且此学生的成绩低于C，那么会强行让该学生注销掉该门课程。

H. 学生、教师信息　交互的目的是增加、删除或者编辑在项III的A中指定的项。如果增加项，则必须包括名字、Id号、教师/学生身份和密码。如果删除或者编辑项，则必须提供Id号，以及要修改的任何字段。如果交互的执行者不是教师[⊖]，则交互失败。

I. 课程信息　交互的目的是显示或者编辑描述已经存在的课程（项ⅢC）的信息；或者输入新的课程信息。交互输入课程号，交互可以改变课程的任何信息，但是不能删除课程。学生只能浏览课程信息，不能输入或编辑课程信息。如果编辑的课程信息违反任何完整性约束，则交互失败。

⊖ 在真实系统中，数据库管理员会使用一个特殊的事务集来控制这些信息。在这个项目里，为简化起见，我们假设，数据库在初始化时就至少带有一个教师的姓名，Id和密码。

J. *学期结束* 交互的目的是结束该学期。该交互只能由教师执行。交互会产生下列的影响：

1. 在项Ⅲ的F中指定的当前学期的标志将会向前进1。
2. 对每个学生来说，成绩I将被分配给该学生已经在参与但还没有给出成绩的课程。
3. 数据库中将会把以前标记为学生已经注册的课程修改为已经参与的课程。
4. 对列在项Ⅲ的C4中的每门课程，参与的学生数将会被设为和已经注册的学生数相等，而注册的学生数将被设为0。

当某个下学期要开设的课程尚未被指定教师或是教室时，该交互失败。

K. *得到课程的学生花名册* 交互的目的是产生当前参与或者已注册某一门课程的学生的Id和姓名的列表。交互只能由教师完成。交互输入课程号以及一个指示要列出的是参与还是注册学生列表的标识。如果申请参与某门课程的学生列表，但本学期没有开设该课程或者申请注册某门课程的学生列表，但下学期不开设该课程，则交互失败。

L. *教室* 交互的目的是显示教室（项Ⅲ E）的座位数，或者输入新教室号和座位数。交互时输入教室号和座位数（如果是新教室）或者仅仅是教室号（如果需要该教室的座位数）。交互只可以由教师进行。

M. *OLAP查询* 交互的目的是允许用户从屏幕输入任意的查询语句。交互仅能由教师（假定他了解数据库的模式）执行。查询使用简单的SELECT语句的形式，查询结果以第一行列出属性名，其他行列出结果的格式显示在屏幕上。如果输入的不是SELECT语句或者SELECT语句输入不正确，则交互失败。

N. *结束会话* 交互的目的是结束会话。接下来的与该系统的任何交互都会需要重新认证。

Ⅵ 系统问题

A. 系统会作为客户/服务器（client/server）系统来实现。客户端计算机将是执行应用程序的个人电脑。

B. 应用程序将会使用一个应用生成器来实现。

C. 用户界面将是图形化的，很容易使用，学生和教师基本上不需要培训即可使用。

D. 数据库可以是任何SQL数据库。该数据库在服务器上执行，并且提供事务接口（换句话说，它可以执行提交和异常中止的操作）。

Ⅶ 交付的产品列表

A. 详细描述发生在每个交互中的事件序列的规格说明文档，其中包括：

1. 使用的表单和控件。
2. 每个表单使用的控件的效果，包括作为每个可能的动作的结果来显示的任何新表单。
3. 系统检查的错误和输出的错误信息。
4. 完整性约束。

B. 详细描述的设计文档。

1. 描述系统的实体-联系（entity-relationship）图。
2. 所有数据库元素的声明，包括表、域和断言。

3. 每个交互分解成的事务和过程。

4. 每个事务和过程的行为。

C. 测试计划，它描述系统如何被测试，包括如何测试每个需求和规格说明。

D. 一个完整系统的展示（包括运行测试计划中提到的测试）。

E. 系统的完整的文档化的代码。

F. 分别为学生和教师设计的用户手册。

G. 描述已交付的系统的规格说明文档、设计文档和测试计划的第二个版本。

3.3 需求分析——新问题

经验表明，不管怎样小心地编写需求文档，当实现小组为准备规格说明文档而分析需求时，总有一大堆新问题会涌现出来。我们会发现部分需求文档不一致或不完整，有关某些没有预见的情况下的系统行为问题等就会出现。实现团队通常向顾客说明这些问题，顾客会给出一个书面解决文档。解决过的问题成为新版本的需求文档的一部分，也成为初始版本的规格说明文档的一部分。这表明要精确指定系统的期望行为是非常困难的。

当将上一节给出的需求文档交给本地的实现团队进行分析的时候，就会指出大量的问题。接下来，我们列出其中一些问题，以及一些解决方案。可能你们本地的实现团队也会发现其他问题。

问题1　在课程信息交互时，如果为一门课程添加预备课程，但是发生循环，该怎么办？例如，课程A是课程B的预备课程，B是课程C的预备课程，而C是A的预备课程。换句话说，课程A是它自己的预备课程。

解决方案　必须在数据库中加入新的完整性约束来处理这种情况，即预备课程必须不存在循环。任何实现课程信息交互的事务都要检查这个条件。如果存在循环，则不能添加预备课程，并且将相关信息显示给用户。（一般情况下，循环检查并不简单。然而，我们可以要求一门课程的预备课程的编号小于它自己的编号。这样的要求可以避免产生循环。）

问题2　在课程信息交互时，如果为一门课程添加预备课程，但是有一个学生已经注册该课程却还没修过预备课程，该怎么办？

解决方案　不为下一个学期的课程添加新的预备课程。

问题3　在课程信息交互时，如果把一门课程允许的最大学生人数减少到某一个值，但是这个值小于现在已经注册的学生数，该怎么办？

解决方案　重新分配教室在现实的学校中很常见，所以必须允许进行这种操作。然而，必须撤销适当数量的学生对课程的注册，使得已注册学生数减少到新的最大值。撤销学生注册的顺序应该与注册时的顺序相反。所有已撤销注册的学生会收到一份书面通知。

问题4　在课程信息交互时，如果要修改一门课程的日期或时间，该怎么办？

解决方案　不对下一个学期课程的日期或时间进行修改。

问题5　在课程信息交互时，如果要取消一门课程，该怎么办？

解决方案　可以取消下一个学期的课程。对于在下一学期中已经注册该门课程的学生，可以撤销注册，学生会收到一份书面通知。

问题6 在学生/教师信息交互时，如果修改学生的Id号，该怎么办？

解决方案 一个Id号（相对于姓名或密码）永久代表一个人，所以企图改变Id号没有意义，除非如果原本输入的Id号是错误的。所以，只有学生在系统中没有其他相关信息，才允许改变该学生的Id号。

问题7 某些交互，比如取得成绩记录、取得注册课程和取得参与的课程，会产生一些描述在某一时刻的数据库状态的报告。这些报告是否包括产生日期和产生时间的信息？

解决方案 是的，所有这样的报告必须包括日期和时间信息。

问题8 项III D和III F的信息能否同时包含年份和学期信息？

解决方案 是的，并且在学期结束交互时需要适当地更新年份。这个信息会在开始的时候初始化。

问题9 系统中用几位数存储年份信息？

解决方案 四位。

3.4 应用程序生成器

学生注册系统需要复杂的用户界面，规格说明文档中必须详细说明用户界面的内容。用户界面必须包含表单、命令按钮、菜单、文本框等等。描述和编写这样的界面看起来是很艰难的任务。幸运的是，有很多应用程序生成器可以减少需要掌握的大量的编程知识和工作量。在本节，我们讨论这些应用程序生成器的基本概念。

应用程序生成器通常包含下面的组件。

- 图形用户界面（GUI）设计器 程序员可是使用它来设计GUI。其中包括用户交互的元素：表单、命令按钮、菜单等等。
- 编程语言 可用于编写应用程序，访问数据库和执行事务。
- 集成程序开发环境 包括一个程序编辑器和其他帮助程序员编写程序和事务以及链接到GUI的工具。
- 允许生成的程序连接和访问一个或多个DBMS的一种机制。DBMS可以位于本地计算机，也可以位于网络的其他计算机中。

一些数据库的应用生成器也包含从某些图形化或表格符号自动生成SQL语句的功能。然而，我们只关心GUI怎样产生，以及它们如何与应用程序交互。我们不讨论任何某个应用程序生成器，但是使用与许多商业系统相似的符号。我们应区别应用程序设计者和应用程序的用户。（在本节，我们使用术语*设计者*特指程序员，他们使用应用程序生成器来设计用户界面。）

3.5 图形用户界面和对象

事务处理系统的GUI包括一组**表单**（form）。例如，图3-1中所示的表单，可用于认证学生注册系统的用户。系统用一个**对象**（object）来代表每个表单。（我们将在附录A中简要地讨论对象。）例如，对应于某个表单的对象可能有一个方法Show，该方法使该表单显示在屏幕上。每个表单包含一组**控件**（control），比如

- 在屏幕上显示文本的**标签**（label）。例如，图中的标题。
- 用户可以输入信息的**文本框**（textbox），这些信息可作为应用程序的输入，应用程序也可以在文本框中写入信息作为输出。例如，在图3-1中，用户Id和密码可以通过文本框输入。
- 用户可以用鼠标点击**命令按钮**（command button），使系统完成一些操作。例如，在图3-1中，当“OK”按钮被按下时，系统读取文本框中的信息来认证用户。
- **菜单**、**单选框**（checkbox）、**复选框**（optionbox），用户可以通过鼠标点击从中选择。

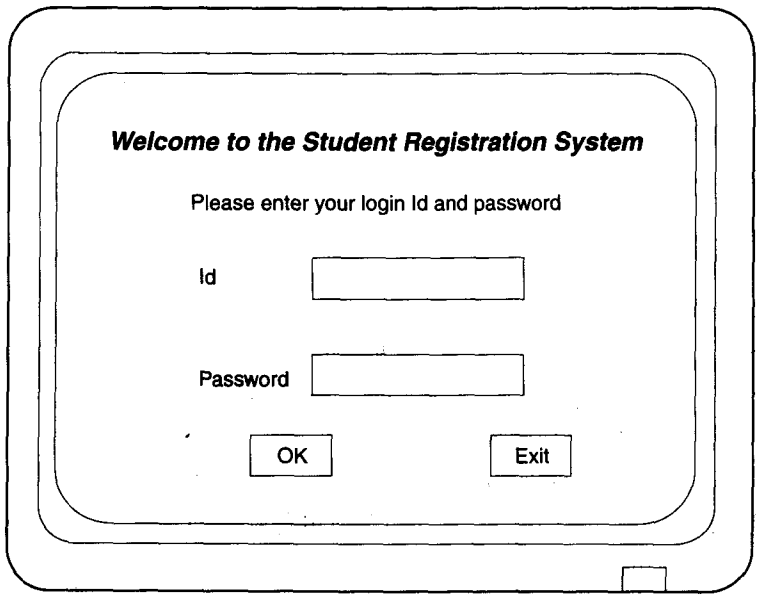


图3-1　学生注册系统的初始表单

控件在系统中也用对象表示。例如，一个菜单可能有一个方法display_menu，点击菜单时，就执行该方法，从而显示下拉菜单。

为设计用户界面，设计者进入GUI设计器，通过鼠标点击告诉设计器在屏幕上创建一个空白表单。然后，设计者可以从一个菜单中选择一组控件，通过鼠标点击把它们放置到表单中，接着使用一般的拖拽操作重新整理控件。GUI设计器可能显示一张文本框图，设计者可以拷贝图然后把图拖到表单中适当的位置。

这些控件和表单用对象表示，而对象在屏幕上有对应的可视化表示。这样，我们把这些对象称作可视化对象（visual object），我们可以把将可视化对象视为由下列两个数据结构包装而成。

1) 表示对象语义的数据结构。对于文本框对象来说，这种数据结构可能包括两个字符串：Name和Text。Name的值是文本框的名称（每个可视化对象必须有一个唯一的名称，应用程序使用它来指定对象），Text的值是文本框中存储的文本信息（也是文本框在屏幕上显示的信息）。

2) 包含对象的可视化表示信息的数据结构。一个命令按钮对象可能有一个标签，该标签就是打印在命令按钮上的文本。另外可能还包含位置、颜色、字体、字体大小等等的信息，

它们存储在名为Location、Color、Font、Typesize等变量中。

作为一个整体，两个数据结构中的信息被认为是对象的性质（property）或属性（attribute）。在设计过程中，应用程序设计者可以使用GUI设计器显示任何对象的属性，可以像图3-2中的表那样显示文本框对象的一些属性。

当对象开始被创建的时候，可以给它的一些属性（比如文本框的名称(Name)）赋予默认值（例如，Textbox1）。要改变名称，设计者只需在属性表的“Name”项中键入想要的名称即可。设计者可以以相似的方式通过修改属性来定制每个表单中的每个对象。我们将会看到，属性也可以在应用程序运行的时候改变。

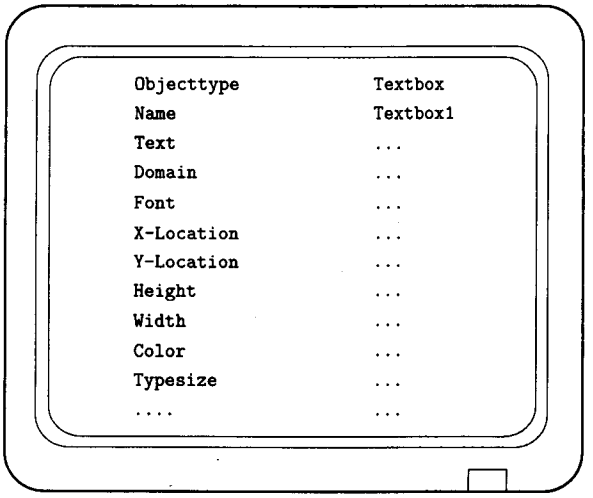


图3-2 文本框对象的一些属性

文本框对象的某个属性可能是信息的**域**（domain）——在运行的时候可以允许键入文本框的值。例如，一个文本框存储一个社会保障号码，它的域是一个九位的字符串数值；另一个文本框存储用户的性别，它的域是字符M或F。当用户键入信息的时候，系统自动检查键入的值是否在域所允许的范围内，否则会产生一个错误信息。

一个可视化对象的数据结构除包含它的属性之外，还会封装一组方法，这些方法使用数据结构内的信息在屏幕上画出对象的图形化表示。我们可以称之为**制图引擎**（drawing engine）。方法Show就是一个例子。制图引擎的方法是在对象显示的时候自动调用的。

制图引擎的一个重要任务是保持图形表示与对象的数据结构表示的一致性。例如，如果设计者使用图3-2中的表把color属性从“red”改为“green”，制图引擎的一个方法就被（自动）调用，把屏幕上对象的颜色从红色改为绿色。相似地，如果设计者使用鼠标把表单中的文本框从一个位置拖到另一个位置，制图引擎方法不但移动对象，还改变对象的数据结构表示中Location属性的值。

这样，设计者可以轻松地创建和定制某个应用的一组表单。其中一个原因是操纵对象（包括它们的图形化表示）的方法相当简单。应用程序生成器已经提供这些方法，并且封装在对象模块中。设计者只需改变对象的属性即可。

大多数应用程序生成器提供一个大型的可视化对象类库。设计者可以在表单中对其进行

各种各样的组合，但只需做少量的编程工作。此外，大多数的应用程序生成器允许设计者定义这些类的子类或编辑它们的代码来扩展它们的属性，这样，应用程序生成器能在增加复杂度的情况下提供更多的方便。

不使用应用程序生成器来创建GUI

很多语言（例如Java）包含一个可视化对象库（以及合适的制图引擎）。在没有应用程序生成器的情况下，我们可以利用这些对象库通过传统的编程技术来创建GUI。例如，使用Java抽象窗口工具包（Java Abstract Window Toolkit，AWT），一个程序可以创建一个按钮对象的新实例，明确地设置它的属性，然后把它放在一个窗口中——所有操作都通过库的方法调用来实现。因为库中的图形化对象包含合适的制图引擎，所以设计者不必关心绘制对象的真实细节。使用这样的一个库，设计者可以很轻松地编写创建GUI的程序，这比使用应用程序生成器更加简单。

为使GUI响应用户的输入，程序使用语言的事件控制机制。当应用程序生成器生成GUI时，也使用这种机制。这就是下一节的主题。

3.6 事件和过程

尽管只有少数的程序需要在屏幕上画控件，但是设计者必须编写过程来实现某一应用的功能。这些由应用程序生成器提供的语言所写的过程组成了应用程序。一个应用程序的过程可以完成如下工作:

- 在应用程序开始的时候显示一个表单。
- 收集和验证由表单输入的信息。
- 改变表单中一个对象的一些属性（例如，在屏幕上对象的外观）。
- 执行与应用相关的计算，并且初始化有关事务去访问数据库（包括执行SQL语句）。
- 在有关表单中显示从事务输出的信息。
- 产生有关打印报告。
- 控制与现实应用交互的设备。

其中许多功能的实现涉及到表单和控件的方法调用。

应用程序过程是**事件驱动**（event driven）的。一个**事件**（event）是一个经常（但不总是）由用户在运行时发起的动作。每个控件都有一组相关的响应事件。例如，当用户使用鼠标点击一个命令按钮的时候，按钮就会对点击事件有响应。而一个标签控件（比如图3-1的标题）对点击事件没有响应。设计者可以使一个应用程序过程与某个对象的每个事件关联。应用程序生成器（或者为应用程序过程使用的语言的实现）提供一种机制：在运行时，它能识别事件发生的时间和与事件相关联的对象，然后调用相关的过程。我们称之为事件**触发**（tigger）过程调用。用这种方法，设计者可以使一个过程与事件“图3-1所示表单的‘OK’按钮被点击”关联。这样，在运行时按钮一旦被点击，过程就会自动被调用。

事件有名称，例如点击事件的名称为“Click”。通过过程的名称可以将过程与事件关联起来，一旦事件发生，过程就被执行。这样，应用程序生成器可能要求在“OK”按钮被点击时调用的应用程序过程名为okbutton.Click，其中okbutton是按钮的唯一的（内部）名称（应该与它打印在屏幕上的外部标签“OK”区分清楚）——也就是它的Name属性的值。

设计者必须实现应用程序过程。这样才能执行事件所要求的过程。在图3-1的表单中，okbutton.Click过程可能完成以下功能：

- 使用文本框中输入的Id和密码来执行认证事务。
- 显示新表单，而表单的显示依赖于认证事务执行成功还是失败。

设计者也可以使应用程序过程与某个不是由用户直接发起的事件关联。例如，当应用程序启动某一段时间后，或当系统发生错误的时候，可能发生某个事件。

下一个问题是应用程序过程如何访问表单中的信息或改变对象的属性。正如我们看到的，许多对象的属性可以由应用程序生成器提供并且封装在对象模块中的方法操纵。例如，在设计的时候，设计者可以在命令按钮的属性表中修改颜色项的值，来改变命令按钮对象的color属性。这一修改导致调用（由应用程序生成器提供的）对象的某个方法，该方法修改对象的颜色属性，然后调用相关的制图引擎方法在屏幕上重新绘制命令按钮。

这些方法也可以由应用程序过程调用，所以这些方法可以在运行时执行，来改变表单或控件的外观或通过文本框输入/输出信息。图3-3显示例程的组织方式。例如，一个应用程序过程在运行时调用过程form1.Show，从而显示一个名为form1的表单。

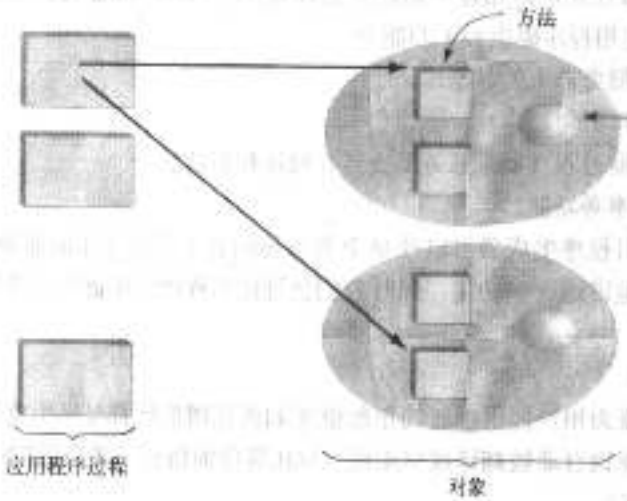


图3-3 应用程序过程与对象之间的关系

另一个例子，假设图3-1中的顶部的文本框的（内部）名称是IDbox，文本框的某个属性是Text（即文本框中显示的字符串）。应用程序过程可以使用标记IDbox.Text访问该字符串。这样，IDbox.Text看上去像是过程中的一个变量，过程可以读取该变量的值或者给它赋一个新值。如果应用程序过程想用IDbox中的文本给另外一个变量st赋值，它可以使用如下语句

```
st = IDbox.Text
```

如果它还想在名为outbox的文本框中输入文本作为输出的话，它可以使用如下语句：

```
outbox.Text = 'This is the text.'
```

在这两个实例中，赋值操作的执行调用对象的一个相关方法。在第一个例子中，方法的作用是把IDbox的数据结构中Text变量存储的值（Text属性的值）赋给（程序）变量st。（在这个例子中，对象方法没有影响对象的外观。）在第二个例子中，赋值方法把指定的字符串赋给

outbox文本框的数据结构中的Text变量，然后调用相关的制图引擎方法改变outbox的外观，从而在outbox文本框中显示出指定的字符串。

在一般情况下，过程可以使用变量

```
object_name.property_name
```

访问任何对象的任何属性，并且可以使用语句

```
object_name.method_name ( . . . arguments . . . )
```

执行对象的方法。

正如图3-3所示，应用程序由一组响应指定事件的应用程序过程组成。一旦应用程序过程被调用，它就可以使用与所显示的对象相关联的属性和方法来访问和修改屏幕上的信息。注意，应用程序过程不封装在对象模块中（像上面提到的方法）。它们访问对象就像它们访问其他的全局声明的数据结构一样。

3.7 访问数据库和执行事务

商业应用程序生成器在提供应用程序访问数据库和执行事务的功能方面差异很大。应用程序生成器至少应该为应用程序提供如下的能力:

- 连接到本地或远程服务器上的数据库。
- 在数据库上执行SQL语句。
- 在服务器内执行存储过程（如果服务器支持存储过程的话）。
- 利用服务器提供的事务功能。

越来越多复杂的应用程序生成器可以使单个事务访问若干个位于不同服务器的数据库。我们会在第10章更详细地讲述这些功能，到时我们会讨论怎样把SQL语句和其他数据库调用包含进应用程序。

1. *自动生成SQL语句*

一些应用程序生成器为用户提供创建简单数据库和使用图形化符号来生成简单查询的功能。用这种方法创建的查询自动被翻译成SQL或与SQL等价的语句。我们将在7.3节讨论这些可视化查询语言的概念框架。

2. *数据表单*

一些应用程序生成器提供高度的抽象，我们可以称之为**数据表单**（data form）有时称为**数据页**（data page）。对设计者或用户来说，数据表单好像直接连接到数据库。当表单一开始显示的时候，表单中的文本框显示相关数据库的值。无论何时用户改变某个文本框中的值，数据库立即响应并更新数据。

使用事件编程的概念，可以很容易地实现数据表单。表单的显示是一个事件，响应该事件的应用程序包含一个从数据库读取信息的SELECT语句并且把这些信息显示在表单的文本框中。应用程序生成器允许用户使用图形化符号来指定SELECT语句。

类似的，当用户改变某个文本框中的值，然后把光标移出那个文本框之后，一个change-of-focus事件就会发生，该事件会执行一个更新数据库的语句。该语句可由图形化符号指定。设计者不必了解事件编程的细节就可以创建一个数据表单，如果图形化符号足够简单的话，

甚至可以不必了解SQL编程的细节。

3. 面向对象应用程序语言

如果应用程序生成器提供的编程语言是面向对象的，那么与应用相关的实体可以表示为对象。例如，在学生注册系统中，程序员可以实现一个称为course的对象类，该类有一个方法register用于把学生加入某门课程。因为关于课程注册的信息保存在数据库中，所以register方法必须读取并更新数据库。

对象（比如course或student）有些时候被称为**企业对象**（enterprise object），因为它们对应于企业所关心的基本实体。对数据库的访问可以封装在这些对象里。数据库本身可以是关系的或面向对象的（见第16章），但是只有企业对象的方法才需要知道模式。

使用企业对象的应用程序过程响应现实的事件，当它们调用可视化对象的方法与用户通过GUI交互的时候，应用过程强制执行企业规则。例如，给学生注册课程的应用程序过程可能首先调用对象student的方法（比如，确定学生是否已经修完预备课程），然后调用对象course的方法register。

使用企业对象，应用程序过程可以用于实现企业需求，而不必知道任何数据库设计的知识。例如，设计注册过程的程序员不必知道数据库模式。

3.8 详细说明学生注册系统

既然我们已经知道创建一个GUI是多么的容易，那么我们就可以回去准备学生注册系统的规格说明文档。应用程序生成器的一个优点是，在项目的需求分析阶段，当表单和控件被详细说明后，设计者可以快速地实现界面的简单原型并显示给顾客。尽管这些原型不包含实现交互的应用程序过程，但是它们可以让顾客感知一下系统的外观。然后顾客可以提出一些修改和改进意见，使得界面对用户来说更实用和直观。

规格说明文档包含对系统做什么的完整描述，这是从最终用户的角度来说明的——它是需求文档的扩展版本。对于一个事务处理系统，规格说明文档应该包括以下内容：

- 企业的完整性约束。
- 为每个表单绘制一张图片，包括表单中的每个控件。
- 当一个控件被使用后，所发生的事件的描述，其中包括：
  - 执行哪个应用程序过程。
  - 表单中发生什么改变，或显示哪个新表单。
  - 什么情况下会发生怎样的错误，以及怎样应对错误。
- 每个交互的描述，其中包括：
  - 用户输入的信息，以及哪些事件导致与用户交互。
  - 交互行为的书面描述（例如，学生注册课程）。
  - 导致交互成功或失败的条件的列表，以及在每种条件下将会发生什么事情。

规格说明文档也可以包括其他关于项目计划的信息（比如，时间计划表、里程碑、可交付的产品列表、成本信息等等）、关于系统问题的信息（比如，运行系统所必须的软件和硬件）

以及任何时间或内存的约束。学生注册系统的规格说明文档的目录中包含下面几部分。

Ⅰ概述

Ⅱ相关文档

Ⅲ表单和用户界面

Ⅳ项目计划

A. 里程碑

B. 可交付的产品列表

注意需求文档与规格说明文档的内容之间的关系。

在下面的小节中，我们介绍规格说明文档中第Ⅲ部分的初始内容。我们将在5.7节给出学生注册系统的数据库设计，以及在12.6节给出注册事务的代码设计与部分代码内容。

3.9 规格说明文档

规格说明文档的第Ⅲ部分（表单和用户界面）包含与用户交互的详细描述。它的初始部分的描述如下所示。

Ⅲ. 表单和用户界面

A. 当进入学生注册系统，显示欢迎表单Form1（见图3-1）。在Form1中

1. 填写Id和Password文本框。
2. 点击“OK”命令按钮，执行认证交互。
   a. 如果认证交互失败，显示Form2（认证错误表单[⊖]）。在Form2中，点击“OK”命令按钮返回Form1。
   b. 如果认证交互成功，而且用户是个学生，显示学生选项窗口Form3（见规格说明B）。
   c. 如果认证交互成功，而且用户是个教师，显示教师选项窗口Form4（见规格说明C）。
3. 点击“Exit”按钮，显示确认退出表单Form5，在Form5中：
   a. 点击“Yes”按钮，退出学生注册系统。
   b. 点击“No”按钮，返回到Form1。

B. 如果认证交互成功，并且所认证用户是学生，则显示学生选项窗口Form3：

1. 如果选择课程描述菜单选项，将执行Get_Course_Names交互，显示课程名称表单Form6，Form6包括：
   a. 课程选项框被选中。
   b. 点击“OK”按钮，将运行Get_Course_Description交互，显示课程描述表单Form7。
   c. 点击“Cancel”按钮，返回Form3。

规格说明文档的第三部分的剩余内容与此类似，在此忽略。

⊖ 我们省略其他表单的图示；它们应该包含在实际的规格说明中。

3.10 参考书目

有许多优秀的介绍软件工程的书籍，例如，[Summerville 1996]、[Pressman 1997]、[Schach 1990]。[Blaha and Premerlani 1998]是介绍数据库和事务处理系统的软件工程相关内容的一本好书。

3.11 练习

3.1 编写一个关于简单的计算器的需求文档。

3.2 根据学生注册系统的需求文档，一个会话可能包括许多交互。后来，在设计阶段，我们会将每个交互分解为多个事务。ACID属性应用到所有的事务，但是涉及多个事务的会话可能不是孤立的或者原子性的。例如，几个会话的事务可能是相互交叠的。解释一下为什么会话可以不是独立的或者原子性的。为什么会话不是一个长的事务？

3.3 假设学生注册系统里的数据库满足需求文档第Ⅳ部分规定的所有完整性约束，那么数据库就肯定正确吗？解释原因。

3.4 学生注册系统的需求文档并不涉及安全问题。编写一个关于安全问题的问题，可以将其包括在一个更现实的需求文档中。

3.5 在3.3节问题2的解决方案中提到，新的预备课程不适合下学期开设的课程。注册事务怎样知道该预备课程是一门新的课程？

3.6 假设扩展学生注册系统使之包括毕业生的清除。描述一些必须存储在数据库中的额外的信息以及完整性约束。

3.7 使用本地应用程序生成器完成图3-1所示的窗体。

3.8 列出本地应用程序生成器所生成的所有的命令按钮对象属性。

3.9 列出本地的应用程序生成器所生成的所有的命令按钮对象的（用户可见的）方法。

3.10 列出本地的应用程序生成器所生成的所有的控制对象。

3.11 列出本地的应用程序生成器所提供的能触发过程调用的所有事件。

3.12 描述本地的应用程序生成器所提供的和数据库交互的工具。

3.13 使用本地的应用程序生成器实现tic-tac-toe游戏。在该游戏中，用户与系统对垒（但是用户不会赢）。

3.14 编写一个关于简单的计算器的规格说明文档。

3.15 使用本地的应用程序生成器实现一个简单计算器。

3.16 编写一个关于如何控制微波炉的规格说明文档。

第二部分

数据库管理

现在我们准备开始学习数据库系统。

第4章讨论在现代数据库管理系统（DBMS）中如何说明数据项以及在事务中如何使用它们。换言之，我们将学习一些关于数据模型和数据定义语言的知识。

第6、7和9章将讨论通过数据操纵和查询语言（特别是SQL）完成数据库管理系统中的事务访问和数据修改。

第10章介绍如何在C或Java这样的宿主语言中使用SQL语句。

第5章和第8章中，我们将探讨设计数据结构的技术，数据库的设计怎样影响只读事务（即查询）和更新事务之间的基本权衡。

第11、13和14章讨论数据库中使用的数据组织和查询处理技术。它们对调整数据库性能举足轻重。在5.7节和第12章，我们将运用这些方法设计学生注册系统。

第15章对事务处理作简要的介绍。事务处理是数据库最重要的应用领域之一。

第4章 关系数据模型

本章介绍关系数据模型。首先，我们定义关系数据模型中一些主要的抽象概念，进而在SQL的具体语法中来说明它们。特别地，本章包含SQL的数据定义子集，该子集可以定义数据库中的数据结构、约束和授权机制。

4.1 什么是数据模型

1. 数据独立

尽管最终所有的数据都以字节的形式记录在磁盘上，但作为程序员应该知道，若直接使用这些抽象级别很低的数据则会很乏味。没人对如何分配扇区、磁道和柱面来存储信息感兴趣，大多数程序员倾向用文件这样对应用程序来说更合理的抽象形式来存储数据。

在学习文件结构的课程中，读者可能已熟悉在文件中存储数据的多种方法。**顺序**（sequential）文件最适合按照数据存储的顺序来访问它们的应用程序。如果访问记录的顺序是随机的，则**直接访问**（direct access）或**随机访问**（random access）形式的文件将最适合。可为文件作**索引**（index），它是一种辅助数据结构，可使应用程序基于**搜索键**（search key）来检索记录。在第11章中我们会详细讨论各种索引类型。文件的记录可以是定长或不定长的。

文件中数据如何存储的细节属于数据建模的**物理层**（physical level）。在数据库领域，该层也称为**物理模式**（physical schema），它是描述文件和索引结构的语法。

早期数据密集型的应用程序直接工作在物理模式，而不像现代数据库管理系统那样提供高级的数据抽象。早期的应用程序这样做是有原因的。其一，商业数据库系统很少且很昂贵。其二，对慢速的计算机而言，直接使用文件系统可提供不错的性能。其三，从今天的标准看，大多数早期的应用程序都很原始，在程序和文件系统之间再构建一层抽象似乎不大合理。

该方法的最大缺点是物理层文件格式的改变会带来昂贵的软件维护费用。“千年问题”即是很好的例子。在20世纪的60和70年代，程序中普遍采用2位数字的硬编码来表示日期中的年份。其出发点在于这些程序会在15～20年内都被替换掉，因此用4位数字（或使用DATE这样的数据类型）会浪费磁盘空间。结果造成每个用到日期的例程都用2位数字来查找年份。因此，若改变日期格式则需要查找和修正程序中的所有代码。Y2K问题在上个世纪90年代使业界花费数十亿美元来更新软件。

如果那些程序中使用DATE这样的数据抽象，那么所有的问题都可以被避免。即使年份在数据库中还是以2位数字进行物理存储，但应用程序仍然可以将年份看作是4位数字。在千年虫问题上，设计者只需要把数据库中年份的物理表示改成4位数字即可，具体做法是先写一个简单的程序把数据库中的每个年份字段都加上1900，然后相应地，将实现函数都用DATE数据类型来访问新的物理表示。这样一来，现有程序都无需再作修改，因为它们仍然可以使用同样的DATE数据抽象。

在需要修改基本的数据结构（即使不频繁）的情况下，数据密集型应用程序的设计如果基于单纯的文件系统则会存在问题。即使是细微的改变，例如在文件中增加或删除一个字段也意味着用到该文件的所有应用程序都必须手工更新，重编译和重新测试。再细小的改变（如合并两个字段或将一个字段一分为二）都可能非常显著地影响现存的应用程序。适应这样的改变将带来繁重的工作甚至出错。此外，需要在原始文件中将数据转换为新的形式，这需要合适的工具来完成，否则代价将极为昂贵。

而且，对于需要频繁查询和使新查询快速实现的应用程序的开发，文件系统提供的数据抽象级别过低。对于这些应用程序，**概念层**（conceptual level）的数据建模是合适的。

概念模式隐藏物理数据表示的细节，取而代之的是用高级概念来描述数据，它们更接近于人们看待数据的方式。例如，**概念模式**（conceptual schema，在概念层描述数据的语法）能够像下面这样描述学生的信息：

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

尽管该模式看上去类似于文件记录的方式，但很重要的一点在于该模式描述的信息片断可能在物理上存储的形式与模式描述的方式不同。实际上，信息片断可能存储在别的文件中（甚至在别的计算机上）。

物理层和概念层上的模式分离产生了简单但有效的**物理数据独立**(physical data independence)思想。应用程序不再和文件系统直接打交道，而只看到概念模式。数据库管理系统自动地在概念模式和物理模式间映射数据。如果物理表示改变，所要做的只是改变这两层之间的映射。所有的应用程序由于仅和概念模式打交道，因此在新的物理数据结构下仍然正常工作。

在对数据抽象的过程中，概念模式还不是最终的结果。第三层的抽象称为**外部模式**(external schema，也称为**用户**（user）或**视图**（view）抽象层)。外部模式用于定制概念模式来满足各种用户的需要，另外它在数据库安全方面也起到重要作用（我们将在后文讨论）。

外部模式看上去像概念模式，现代数据库管理系统基本上以相同的方式定义这两种模式。然而，尽管每个数据库只有一个简单的概念模式，但它可能有多个外部模式（即在概念模式上的视图），通常每类用户一个。比如，为生成学生的账单信息，学校财务室需要知道每个学生的GPA和他的总学分，但对于课程名和成绩则不关心。即使GPA和学分在数据库中没有显式地存储，财务室仍可以获得具有这些项的视图，这些项就像普通字段（当访问字段时，这些普通字段的值在运行时计算出来）那样显示出来，而那些和账单信息无关的字段和关系都将被忽略。类似地，学生的指导老师不必知道任何账单信息，因为从他们看待注册系统的角度，这些信息也可以忽略。

以上观点带来**概念数据独立**（conceptual data independence）的原则：适合特定用户群的应用程序可以使用适合该用户群的外部模式。外部模式和概念模式之间的映射由数据库管理系统负责，因此应用程序从概念模式的改变以及物理模式的改变中都隔离出来。总体结构如图4-1所示。

2. 数据模型

一个**数据模型**（data model）包含工具和用于描述以下内容的语言。

1）概念模式和外部模式。模式指定数据库中数据存储的结构。模式用**数据定义语言**(Data Definition Language，DDL）描述。

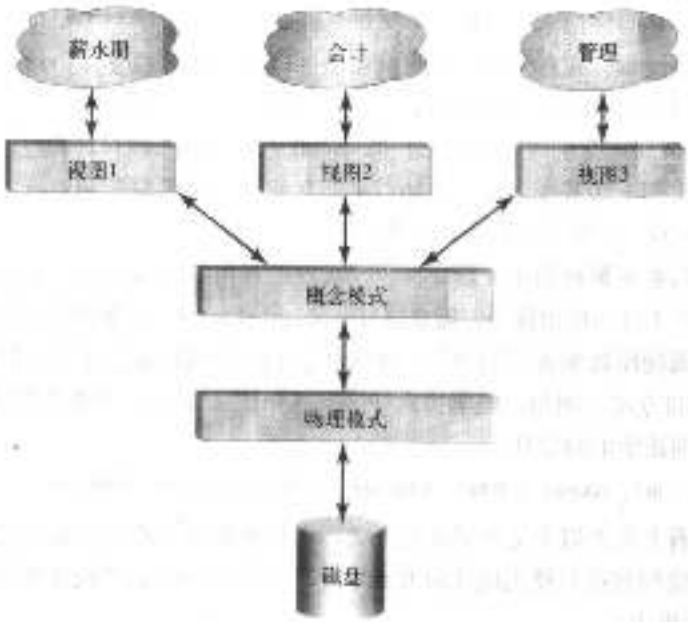


图4-1　数据独立的层次

2）约束。约束指定数据库中数据项必须满足的条件。约束规范语言一般是DDL的子语言。

3）数据上的操作。数据库项上的操作用**数据操纵语言**（Data Manipulation Language，DML）来描述。DML通常是任何数据模型中最重要也最受关注的部分，因为这些操作最终为用户产生高级的数据抽象。

此外，所有商业系统会提供某种**存储定义语言**（Storage Definition Language，SDL），它允许数据库设计者改变物理模式（虽然大多数系统保留最后的定义权）。SDL通常紧密地与DDL集成。若数据库管理员引入新的SDL语句，则物理模式可能会改变，但这种改变不会影响应用程序的语义，原因在于物理数据独立性将应用程序与物理改变屏蔽开来。因此，尽管应用程序的性能会受影响，但是它产生的结果不会受到影响。

在后面的几节中，我们描述商业数据库管理系统中所有数据模型的基础，即*关系模型*和这些数据库管理系统所讲的语言：*结构化查询语言*（SQL）。注意，SQL不只仅仅是查询语言，它是由DML、DDL和SDL组合而成的。

4.2　关系模型

关系数据模型（relational data model）由E.F.Codd在1970年提出，它在当时是数据库领域中的重大突破。事实上，在20世纪70和80年代，数据库研究和开发很大程度上是由Codd的开创性工作所左右的[Codd 1970,1990]。甚至在今天，尽管大多数的商业数据库管理系统已经开始加入面向对象的特性，但它们的基础仍然是关系模型。

关系模型主要的吸引力在于它构建在简单且自然的数学结构——*关系*（或表）上。关系包含一组功能强大且高级的操作，数据操纵语言植根于数学逻辑上，坚固的数学基础意味着关系表达式（即查询）可以被分析，因此，任何关系表达式能够被等价地转化为另外一种形

式，在这种形式下，表达式可以在查询优化过程中更加有效地执行。这样，程序员不必研究每个数据库内部的本质细节，也不必知道查询评估系统是如何工作的。他们只需要以自然且简单的方式构建某个查询，然后再发给查询优化器让它找出执行效率更高的等价查询。

然而，查询优化器也有局限性，对于某些复杂的查询会导致性能上的退化。因此程序员和数据库设计者都应该理解他们所用的语法，掌握相应的知识，这样程序员写出的查询会更容易由数据库管理系统优化，而数据库设计者通过增加合适的索引以及使用其他的设计技术可提高重要查询的性能。

4.2.1 基本概念

关系模型的中心是**关系**（relation）。一个关系是**模式**（schema）和该模式**实例**（instance）的组合。

1. 关系实例

关系实例只是由命名的若干列和行组成的表格。在不会产生混淆的情况下，我们就用“关系”指代关系实例。关系中的行称为**元组**（tuple），它们就像文件中的记录一样，但与文件记录的不同之处是，所有的元组的列数相同（该数目称为关系的元数），并且一个关系实际中不存在两个相同的元组。换句话说，关系实例是唯一元组的集合。关系实例中元组的数目称为**基数**（cardinality）。

图4-2是STUDENT关系的一个实例。在关系模型中，关系中的列一般都应该被命名，它们也被称为**属性**（attribute）。而且，由于关系是元组的**集合**，所以元组的次序是无关紧要的。类似地，由于列已经被命名，所以它们在表中的次序也是不重要的。图4-2和图4-3是同一个关系。

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

图4-2 STUDENT关系的实例

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

图4-3 列和元组顺序不同的STUDENT关系实例

请读者注意，术语“元组”、“属性”和“关系”一般用在关系数据库理论中；而“行”、

“列”和“表”一般用在SQL中。尽管如此，这些术语可以互相替换使用。

一个关系的任何行中的属性值的取值范围是一个集合，称之为该属性的**域**（domain），比如，STUDENT表中的Address属性，其取值域是所有字符串的集合。域中这些值要满足一个重要的需求，即**数据原子性**（data atomicity）[⊖]。数据原子性不是指这些值不能再分解。正如上面的例子中所示，值可以是字符串，它们是可以再分的。数据原子性的含义是关系模型不会采用任何手段进一步观察这些值的内部结构，对关系操作符来说好像它们是不可分的。

该原子性限制对关系模型来说有时是一种缺陷。大多数的商业系统会放宽这个限制，有些系统干脆去除该限制，于是对象－关系（object-relational）这样的新数据模型应运而生。我们将在第16章再介绍对象－关系模型这个内容。

2. 关系模式

一个**关系模式**（relation schema）包括如下的部分：

1) 关系**名**。关系名在数据库中必须唯一。

2) 关系中的属性的名字以及相关联的域名。**属性名**即是赋予关系实例中列的名字。关系中所有的列都必须被命名，且同一关系中的列不能重名。**域名**（domain name）是为定义良好的值集所赋予的名字。在编程语言中，域名通常称为类型。例如整型（INTEGER）、实型（REAL）和字符串（STRING）。

3) 完整性约束（Integrity Constraint，IC）。它是施加在该关系模式实例上的限制（也就是说，限制哪些元组可以出现在一个关系实例中）。只有当一个实例满足模式的所有完整性约束时，它才是**合法**（legal）的。

为说明这些概念，我们回顾上面提到过的模式：

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

该模式确定STUDENT关系必须有4个属性：Id，Name，Address和Status，它们对应的域是INTEGER和STRING[⊜]。如该例所示，同一个模式中不同的属性名字都不一样，但它们可以有相同的域。

域指明STUDENT关系的字段Id的所有值必须属于整型域，而其他列的值都属于字符串。我们很自然地假设整型域包含所有的整数，而字符串域包含所有的字符串。然而，模式中可以有用户自定义的域，比如SSN或STATUS域，它们包含适合属性的精确值。比如，可以定义STATUS域只包含“freshman”、“sophomore”等，定义SSN域包含所有的9位正数（但也仅仅包含9位的正数）。以上讨论的是施加于关系模式的所谓的类型约束。

若S是关系模式，s是一个关系实例，则**类型约束**（type constraint）要求s必须满足下面两个条件：

- 列命名　s中的每个列必须对应于S中的属性（反之亦然），并且列名必须和对应的属性名字相同。
- 域约束　对每个在S中的属性-域对<attr: DOM>，s中的列attr的值必须属于域DOM。

⊖ 这里的数据原子性和事务原子性无关，切勿混淆它们。

⊜ Id属性可以用字符串来表示，这对某些大学可能是更合适的；但是整数表示更节省空间，选择使用它仅仅是为了进行说明。

下面我们会看到，类型约束只是和关系模式相关的几个约束之一。模式的实例必须满足所有这些约束。

3. 关系数据库

关系数据库（relational database）是关系的有限集。因为关系由两部分组成，所以关系数据库也是由两部分组成，即关系模式的集合（和另外的实体，这些实体我们只做简述）以及对应的关系实例的集合。关系模式的集合称为**数据库模式**（database schema），对应的关系实例的集合称为**数据库实例**（database instance）。在不产生混淆的情况下，一般也用术语“数据库”指代数据库实例。图4-4是学生注册系统的数据库模式的片断。图4-5是对应该关系模式的数据库实例的例子。我们看到每个关系都满足对应模式所指定的类型约束。

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

图4-4 学生注册数据库模式的片断

4.2.2 完整性约束

我们在2.2节曾讨论过完整性约束在应用程序中起的作用。现在结合数据库模式再来考察它们。**完整性约束**（IC）是关于数据库中所有合法实例的语句。也就是说，这些关系实例必须满足所有和数据库模式相关的完整性约束。前面我们已经研究过类型约束和域约束，后面我们会看到其他类型的约束。

有些完整性约束是基于企业的业务规则的。比如，“所有雇员的收入不应该超过老板”就是这样一个约束。这些约束一般在应用程序的需求文档中列出。其他的约束（比如类型和域约束）则基于模式设计，由数据库设计者指定。

因为完整性约束属于数据库模式的一部分，它们通常在一开始的模式设计中就被指定。当然也可以在数据库已创建好且导入数据后再增加或删除一个完整性约束。一旦在模式中指定约束，数据库管理系统就应当保证在事务执行过程中不被违反这些约束。

一个完整性约束可以只限于**关系内**（intra-relational），即它只涉及到一个关系；它也可以是**关系间**（inter-relational）的，即它涉及到多个关系。类型约束就是关系内IC的例子。再看另外一个例子，有一个约束，它规定STUDENT表的实例所有行的Id属性值必须唯一，该约束称为**键约束**（key constraint，键约束将在后面的内容中介绍）。断言每位授课的教授的Id属性的值必须是出现在PROFESSOR表中的某一行的ID属性中的值就是一个关系间的约束，这个约束称为**外键约束**（foreign key constraint）（将在后面讨论）。而“员工收入不能超过老板”[⊖]的规则既可以是关系内约束也可以是关系间约束，这取决于薪水信息和公司管理结构信息是否存储在相同的关系中，该约束属于语义约束的类型，该约束也将在本节后面部分中描述。

⊖ 到目前为止，我们不必担心特殊情况，比如对公司总裁应用此约束，而总裁没有老板。

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 & save*
	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

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

图4-5 数据库实例片断

现在介绍的约束都是**静态的完整性约束**（Static IC）。**动态的完整性约束**（dynamic IC）的不同点在于：它们不是限制关系实例在数据库中的合法性，而是限制它们的变化。这种约束对表示企业的业务规则特别有用。例如，规定每个事务中薪水的上涨或下降比例不可超过5%。再例如，规定一个人的婚姻状况不能从单身变到离婚。银行可能会有这样的业务规则，若发生透支，则根据信用贷款的最高限额，通过现金转账，在下个营业日前必须补交所欠金额。

但是，主流的数据操纵语言（如SQL）和商业的数据库管理系统并没有为动态约束的自动执行提供很多支持。因此，若一个事务更新数据库，则应用程序设计者必须提供代码执行该事务中的约束。由于没有简便的办法来验证事务是否真的遵守那些规则，数据库完整性只好依赖于实现事务团队的设计、编码和质量保证的能力。

满足静态完整性约束则容易的多。与动态完整性约束相比，它们既容易指定也容易执行。本节我们讨论了最普遍的静态完整性约束。之后的几节将介绍如何在SQL中描述静态完整性约束。

1. 键约束

读者已经看到键约束的一个例子，即STUDENT表实例中的Id属性值必须唯一。对于更复杂的例子，可考虑TRANSCRIPT这个关系。一般来说，可以假定对任一个学期的任一门课，学生都会有一个最终的成绩，我们进而可指定｛StudId，CrsCode，Semester｝作为一个键。该规格说明保证对属性StudId、CrsCode和Semester的任意给定值，表中至多只有一条成绩记录对应这些值。若该元组真的存在，那么它指定某个学生在某个学期的某门课的唯一成绩。

有直观的认识后，再来看看键约束更加精确的定义。和关系模式S关联的**键约束**key($\overline{K}$)由S中属性的子集 $\overline{K}$ 组成，并有如下的性质：若S的实例s不包含这样的两个不同的元组，即元组中属于 $\overline{K}$ 的每个属性其值都对应相同，则s满足key($\overline{K}$)。

模式S的每个键约束key($\overline{K}$)都假定有一个确定的*最小值性质*：key($\overline{K}$)的子集不能再是S的键约束。因此，若A和B是属性，则不允许指定$\{A\}$和$\{A, B\}$都是同一关系中的键。此外，若$\{A, B\}$是键，则对于属性A和B，最多只有一个元组允许有$\{a, b\}$这样的一对值。然而，不同的元组对于属性A可能会有相同的值，但在B中不会有相同的值；反之亦然。

这样，在TRANSCRIPT关系中倘若某个学生在不同的学期中都修某门课程，则很可能会有不同的元组记录该学生该门课的成绩。同样地，也不限制某个学生在同一个学期中学习几门不同的课。

关于键这个概念有以下三点需要注意：

1) 若key($\overline{K}$)是模式S的键约束，$\overline{L}$ 是S中属性的集合且S中包含 $\overline{K}$，那么S的合法实例不应该包含这样的不同元组，它们中属于 $\overline{L}$ 的每个属性都对应相同。实际上，对于 $\overline{L}$ 中的每个属性，若元组t和元组s都有同样的值，则对于 $\overline{K}$ 中的每个属性它们也肯定有同样的值。但由于key($\overline{K}$)是键约束，所以t和s必定是同一个元组。

由以上观点可以产生下面的概念：S中包含键的属性集称为S的**超键**（superkey）。超键可能恰好是键，但也可以有不是键的超键。比如，在TRANSCRIPT这个例子中，{StudId，CrsCode，Semester}既是超键也是键。{StudId，CrsCode，Semester，Grade} 是超键，但不是键。换句话说，超键就像一个没有最小性条件的键。这样，我们能得出若属性集 $\overline{K}$ 是超键且它没有真子集也是超键，则该属性集是键。

2) 每个关系都有键（因而也有超键）。实际上，若模式S的合法实例中若有两个元组的每个属性值都相同，则由于关系是集合而集合不允许有相同的元素，所以这对元组只能是相同的，那么S的所有属性构成的集合必定是超键。另一方面，若S的所有属性构成的集合不是最小的超键，则一定会有超键是S的真子集。若该超键还不是最小的超键，那么应该还有更小的超键。但一个关系中属性的数目是有限的，我们最后一定会达到这个最小的超键，由键的定义可知，它一定是键。

3) 模式可以有不同的键。例如，在COURSE关系中，CrsCode可以是一个键。但由于同一个系不会开设同名的不同课程，我们还可以用{DeptId, CrsName}作为该关系的键。

若一个关系有多个键，则它们都称为**候选键**（candidate key）。然而，通常会指定其中一个键作为**主键**（primary key）。在应用程序中，主键可以有也可以没有任何特定的语义（通常选取相同候选键中的第一个作为主键）。然而，只要给定主键值，商业的数据库管理系统要使用主键来优化存储结构从而加快数据访问，所以主键的选择会影响到物理模式。

图4-6给出学生注册数据库模式以及包含的键约束。注意COURSE关系有两个键。

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

图4-6　学生注册数据库的键约束片断

2. 参照完整性

在关系数据库中，一个关系中的元组会常常引用同一个关系或其他关系中的元组。比如，图4-5中TEACHING关系的第一个元组的属性ProfID值是009406321，它指到PROFESSOR表中的教授Jacob Taylor这个元组，该元组Id字段的值就是009406321。同样地，TEACHING表的第一个元组的值MGT123指到COURSE表中包含的Market Analysis课程的元组。

在许多情况下，如果被引用的元组不存在则数据完整性会遭到破坏。比如，若COURSE关系中没有一个元组描述MGT123，则TEACHING关系中拥有元组<009406321,MGT123,F1994>就毫无意义。否则，Jacob Taylor教哪一门课呢？同样地，若TEACHING关系中没有Id为009406321的元组，则也无法确定哪位教授讲授Market Analysis这门课。

被引用的元组必须存在（这是数据的语义所需求的），这种需求称为**参照完整性**（referential integrity）。参照完整性的一个重要类型是所谓的**外键约束**[⊖]。

假设S和T是关系模式，$\overline{F}$是S中的属性列表，key($\overline{K}$)是T的键约束。并假设$\overline{F}$和$\overline{K}$的属

⊖ 请记住参照完整性的概念比外键约束的概念更为通用。很快我们会介绍包含依赖，它是另一种参照完整性。

性之间存在一对一的对应关系（当然这些属性名字不必相同）。当且仅当对于每个元组$s \in s$，存在t中的元组$r \in t$，它在$\overline{K}$中属性上的值和s在$\overline{F}$中对应属性的值相同，则关系实例s和t（分别对应模式S和T）满足**外键约束**“$S(\overline{F})$引用$T(\overline{F})$”，且F是**外键**（foreign key）。

图4-7解释了外键约束的概念。在图中，表T_1的属性D被声明为外键，它引用表T_2中的属性E。E必须是T_2的候选键或主键。但注意，两个表中引用和被引用的属性（D和E）其名字不必相同。还应注意，尽管T_1中的每行都正好引用表T_2的某行，但不是T_2中所有的行都必须被引用，并且T_1中可以有2行或更多的行引用T_2的同一行。

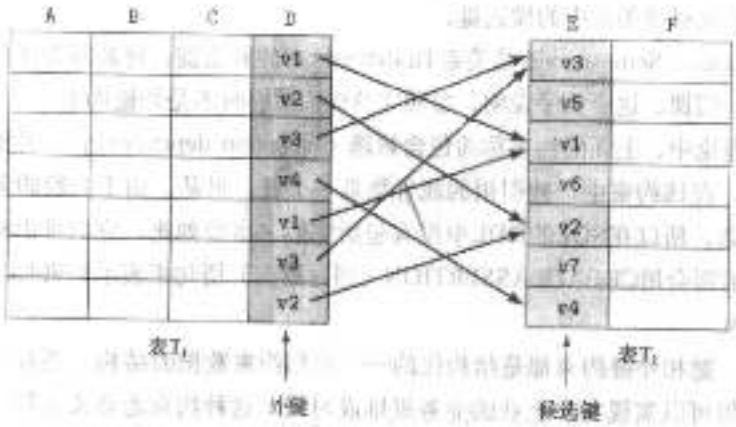


图4-7 表T_1中的属性D是外键，它引用表T_2中的属性E，而E是候选键

在下一节，我们会介绍如何在SQL中定义外键约束，并讨论这种约束的精确语义（比起上面的定义，它的自由度稍大一些）。

在外键约束中，引用和被引用的关系可以是相同的，比如，对于模式：

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

该模式的键是{Id}。主管也是雇员，故有约束“EMPLOYEE（MngrId）引用EMPLOYEE（Id）”。该约束意味着EMPLOYEE关系中的每个元组，比如<……,998877665>，属性MngrId的值998877665必须出现在这个关系的某个元组的Id字段中。

对应图4-6的外键约束如图4-8所示。这里有三点值得注意：

```
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)
```

图4-8 学生注册系统的部分外部键约束

1）用于交叉引用关系的属性不必有相同的名字。比如，TRANSCRIPT表的属性StudId引用STUDENT表的属性Id，而不是StudId。上面所说的EMPLOYEE关系中外键的情况也是一样的。

2）外键可能由多个属性组成，如图4-8中最后一个约束所示。浅显地说，该约束规定若某

学生在某个学期修某门课，那么在该学期必须有教授教这门课。

3) 这一点是微妙的。不同于人们普遍的想法，教授不会在没有学生的班级里授课——至少在某些大学里是这样。换句话说，

```
TEACHING(CrsCode,Semester)
        references TRANSCRIPT(CrsCode,Semester)
```
(4.1)

也应该是数据库中的一个约束，即关系TEACHING中的每一元组的课程必须被至少一个学生所选中。因此(4.1)也是一个参照完整性约束，但它不是外键。因为根据外键的定义，被引用的属性集合必须是被引用关系中的候选键。

集合<CrsCode, Semester>不是关系TRANSCRIPT中的候选键，这是因为在任何学期可以有多个学生选同一门课。这个例子说明，参照完整性约束有时不是外键约束。

在数据库理论中，上面的约束称为**包含依赖**（inclusion dependency）。外键约束是一种特殊的包含依赖，在该约束中，被引用的属性集是一个键。但是，由于一般的包含依赖在自动执行时比较复杂，所以在SQL的DDL中没有包括它们。尽管如此，它们可由SQL的断言机制来表达。本章后面会用CREATE ASSERTION（创建断言）语句来表示约束4.1。

3. 语义约束

类型、域、键和外键约束都是结构化的——它们约束数据的结构。还有一些约束和结构没什么关系，但可以实现某个企业的业务规则或习惯。这种约束是语义上的，这是因为它们源自被数据库建模的特定应用领域[⊖]。

学生注册系统中许多的语义约束在第3章已经讨论过。比如，注册某门课程的学生人数不能超过该门课的教室的座位数，学生注册一门课必须首先要完成该课程的所有预备课程，等等。后面会说明，SQL可以为指定各种语义约束提供支持。

4.3 SQL——数据定义子语言

读者已经熟悉关系模型的基本概念（如表和约束），本节将介绍如何利用SQL的数据定义子语言指定“现实世界”中的这些概念。我们的讨论基于SQL-92标准，但要注意，大多数的数据库厂商没有完全支持该标准。SQL的用户手册长达几百页（实际的标准有几千页），我们将只讨论该语言最重要的部分。若读者计划承担重要的SQL项目，应当阅读厂商提供的参考手册。

当然，SQL-92是过去的标准，我们还应该了解一下新的标准。为解释事务处理中的一些新概念，我们也会谈及SQL:1999。本书的其他部分也会介绍SQL:1999标准。9.2节将描述触发器，16.5节将介绍对象-关系数据库。一些厂商在他们最新版本的数据库管理系统中正逐渐支持SQL:1999。

模式用CREATE TABLE语句说明。该语句语法丰富，我们将逐步说明这些语法。它的几乎最短的语法可以指定类型约束：关系名，带有相关域的属性名；当然该语句还可以指定主键、候选键、外键约束，甚至某个语义约束。

⊖ 术语“语义约束”可能有点误导性，有人会说键和外键也应该是语义约束，因为它们也反映某些业务规则。

4.3.1 指定关系类型

STUDENT关系的类型可用如下语句定义[⊖]:

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

读者应能很容易地把该模式和前面的例子联系起来，上面我们用黑体显示保留字，使得它们的显示更为突出。

4.3.2 系统目录

当数据库管理系统将DML语句翻译成可执行的程序时，它必须用一些信息来描述数据库的结构。这些信息保存在**系统目录**（system catalog）中。上面CREATE TABLE命令的主要目的是定义一个模式，此外它还把一行描述创建表的信息插入到系统目录中。目录是带有模式的特殊关系的集合。图4-9中的COLUMNS表是系统目录的一部分，该表的每一行包含数据库表的某个列的信息，这样所有数据库中所有的列都以这样的方式来描述。例如，表COURSES的四个列由COLUMNS的四行来描述。

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)
	...	...	...	...

图4-9 目录关系

由于COLUMNS也是一个表，它的信息也需要记录下来。但我们不需要单独创建表来记录这些信息，而是可以方便地在COLUMNS表自身中来描述它自己！这样，COLUMNS表的前面几行描述该关系的模式。例如，COLUMNS第一行说明它的第一个列其属性名是AttrName，域是CHAR(255)。

那么所有这些引用复杂性会造成步步为营吗？读者无需担心这个问题，数据库管理系统的系统目录模式是由厂商设计的，当数据库管理员初始化一个新的数据库时，目录实例会自

⊖ 为免混淆，关系名、属性和类型用不同的字体。

动生成。

4.3.3 键约束

SQL用两种不同的语句来指定主键和候选键：PRIMARY KEY和UNIQUE。例如，COURSE表的模式如下所示：

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

注意，我们把属性DeptId的域从DEPTS改为CHAR(4)，把属性CrsCode的域从COURSE改为CHAR(6)。不要担心这个问题，后面我们会解释如何使用这些不正式的域名。临时改变的原因在于CHAR和INTEGER是内建的域类型而DEPTS和COURSE是用户自定义的。后面会解释在SQL中如何完成这个操作。

4.3.4 处理空缺信息

在我们上面定义的关系中，它包含的每个元组依次都有已知的值，如在元组<111111111,Doe John,123 Main St.,freshman>中，每个属性如Id，Name，Address等都有已知的值。但在现实中，某个属性的值可能是未知的。例如当学生Doe John刚注册时，他的住址可能是不知道的。学校会要求他尽可能提供地址信息，但是如果他没有这么做也不应当把他排除出数据库之外，而是应该先存储NULL这个占位符到地址属性上，等他确定地址后，再填入地址信息。类似地，当学生注册某门课程时，即使Grade属性没有特定的值（事实上，在学期结束以前不会有成绩），但相应的元组也能输入TRANSCRIPT关系中。

NULL占位符通常称之为**null值**，也许这会误导读者，因为NULL不是一个值，这里它只意味着这个地方缺乏“正常”值。在数据库理论和实践中，NULL被看作是特殊的值：它可以作为任何属性域的一员，但又不同于任何域中的其他值。实际上，在第6章我们会发现NULL甚至不等于它自己!

在上面的例子中，由于缺乏信息而出现null值。在有些情况下，设计中也会带来null值。比如属性MaidenName（婚前名字）只适合女性而不适合男性。数据库设计者可能会用模式EMPLOYEE（Id: INT, Name: STRING, MaidenName: STRING）来描述企业员工。若这种模式用于男性雇员的话，那么这些元组的MaidenName属性不会也不应该有任何值，只能用null来表示。

正如后面将要介绍的那样，null值经常带来其他的问题，特别在查询处理中。由于这样或那样的原因，有时会要求在某些特殊的地方不允许出现null值，比如，主键中就不允许出现null值。否则，我们怎么解释STUDENT关系中<NULL,Doe John,123 Main St.,freshman>这个元组？如果John Doe和他的一个也叫John Doe的朋友共居一屋，而他也是该校大一的新生，那么如何区分他们两个人呢？

大多数的数据库管理系统会拒绝可能使主键的某个属性出现null值的更新。除此之外，还有些地方不适合使用null值。比如，尽管可以不指明地址信息，但不输入学生的名字肯定会带来问题。

总之，数据库设计者可以用如下方法处理空值问题，就是在那些对数据库的语义完整性至关重要的属性中不允许出现null值。比如，可像如下的例子那样消除Name字段的null值:

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

该例中，null值不能出现在主键Id（不必显示地指明）和Name属性（我们必须显示地说明）中。这里还有个特性需要注意，用户可以为某个属性指明默认（Default）值。当插入一条新元组且没有给该属性赋值时，这个缺省值会自动赋给该属性。

4.3.5 语义约束

语义约束用CHECK子句说明，基本语法形式是:

CHECK (条件表达式)

条件表达式可以是任何出现在SQL语句的WHERE子句中的谓词或谓词的布尔组合。如果条件表达式的值是false，则说明违反完整性约束。

CHECK子句不是单独使用的。它要么附属在CREATE TABLE语句中，要么附属在CREATE ASSERTION语句中。对于前者，它通常是在关系上实现关系内（intra-relational）约束；对于后者，它通常实现关系间（inter-relational）约束。

下面的例子说明CHECK子句如何限制属性的取值范围。

```
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 )  )
```

(4.2)

限制属性的取值范围不仅仅只像上面那样使用CHECK约束。使用下面所示的关系模式，我们可以表达“经理必须比下属的薪水更高”这个约束:

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

CREATE TABLE语句中的CHECK子句语义上要求相应关系上的每个元组都必须满足相应CREATE TABLE语句中的CHECK子句的条件表达式。

该语义的重要推论是空关系（没有任何元组的关系）总是满足所有的CHECK约束，这是

由于此时没有元组可供检查。这种情况会导致某些不可预料的结果。考虑下面在句法上都正确的模式定义：

```
CREATE TABLE  EMPLOYEE  (
    Id              INTEGER,
    Name            CHAR(20),
    Salary          INTEGER,
    DepartmentId    CHAR(4),                                    (4.3)
    MngrId          INTEGER,
    CHECK ( 0 < (SELECT COUNT(*) FROM EMPLOYEE) ),
    CHECK ( (SELECT COUNT(*) FROM MANAGER)
                < (SELECT COUNT(*)FROM EMPLOYEE) ) )
```

两个CHECK子句都涉及到计算关系中元组的数目的SELECT。因此，第一个CHECK语句大致表示EMPLOYEE关系不能是空关系。然而该约束尽管看起来没有问题，但对于检查非空的关系它不会达到目标。实际上，我们介绍过，是关系EMPLOYEE的每个元组都应该满足该条件，而不是关系本身要满足该条件。因此，如果关系是空的，它也满足每个约束，即使这个CHECK想要限制关系不能为空!

(4.3)中的第二个CHECK子句说明使用了关系间的约束。先假定公司有一张表MANAGER，其元组存储公司每个经理的信息，而该约束表明雇员必须比经理多，当然要在表EMPLOYEE不为空的条件下。

除这些小问题之外，由于在表的定义中捆入另外的表，所以第二个约束看上去不是很直观。为克服这个问题，SQL-92提供使用CHECK子句的另外一个语句：CREATE ASSERTION。同表一样，断言也是数据库模式的组件，它合并CHECK子句以对称的关系给两张表施加约束。这样，上面的两个约束重新描述如下（这次它们是正确的）：

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

和表定义中出现的CHECK条件不同，整个数据库的内容都必须满足这些CREATE ASSERTION语句的CHECK条件，而不仅仅只是主表的单条元组满足条件。这样，当且仅当EMPLOYEE关系中的元组数目大于0，数据库满足上面第一个断言。类似的，只要MANAGER关系的元组数目比EMPLOYEE关系的元组数目少，则第二个断言也满足。

再来看一个断言的例子。首先假设有关雇员和经理的薪水信息存储在不同的关系中。关于谁赚得多的规则如下面的断言所示，该规则字面上意思是说不会有这样的雇员：他有老板但老板的薪水却很低。我们假设MANAGER关系有属性Id和Salary。

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

现在的一个有趣的问题是，如果在指定THOUSHALTNOTFIREEVERYONE这个约束时，EMPLOYEE关系是空的会怎么样？另外，在指定THOUSHALTNOTOUTEARNYOURBOSS约束时，关

系中已经有一个员工的收入超过老板，又会怎么样？问题答案由SQL-92标准来回答。SQL-92规定当定义一个新的约束时，若已有的数据库不满足它，则该约束不会被接受。此时数据库设计者不得不找出约束被违反的原因，然后或改正约束或调整数据库[⊖]。

最后再来看一个稍微复杂的例子[⊖]，它演示如何使用断言来说明非外键约束的包含依赖。更明确点，我们就用(4.1)中的包含依赖，其SQL-92的断言语句如下：

```
CREATE ASSERTION  CoursesShallNotBeEmpty
    CHECK (NOT EXISTS (
        SELECT * FROM Teaching
        WHERE NOT EXISTS (                                          (4.4)
            SELECT * FROM Transcript
            WHERE Teaching.CrsCode = Transcript.CrsCode
                AND Teaching.Semester = Transcript.Semester)))
```

这个CHECK约束表明，TEACHING关系中不出现这样的元组（外层NOT EXISTS语句），在TRANSCRIPT关系中不存在与之对应的课程（内层NOT EXISTS语句）。TEACHING关系中的元组和TRANSCRIPT关系中的元组在它们的CrsCode和Semester这两个属性值相等时就引用相同的课程，这是由最内层的WHERE语句完成的。

不同的断言需要不同的维护代价（数据库管理系统检查断言是否满足所需的时间）。一般来说，关系内约束的代价比关系间约束的代价低。在所有关系间的约束中，基于键的约束比那些不是基于键的约束更容易执行。因此，外键约束的代价要比一般的包含依赖的代价（如4.4）低。

由数据库管理系统完成完整性约束的自动检查是SQL的强大特性之一。它不仅可以保护数据库免受不可信用户或粗心的程序员的错误造成的危害，而且能简化对数据库的访问。比如，主键约束保证一个表中最多只能有一个元组包含特定的主键值。如果数据库管理系统不为该约束提供自动检查，那么应用程序若要插入一个新的元组或更新已有的某个元组的键属性，那也不得不从表头开始扫描一遍来保证没有违反主键约束。

4.3.6 用户自定义域

前面我们看到，在创建表时可由CHECK子句限定属性的取值范围。SQL-92还提供另外一种手段来执行这种约束，即允许用户定义合适的值范围，并赋予它们域名，然后就可以在表定义中使用它们。例如，我们可以创建GRADES域，并取代CHECK约束用在TRANSCRIPT关系的定义中。

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

它和(4.2)中的约束的唯一不同在于后者直接施加约束于STUDENT关系的Grade属性上，而这里用VALUE这个保留字而不是属性名（不能用属性的原因是域不隶属于任何具体的表）。现在我们就可以添加

```
Grade Grades
```

⊖ 但是，许多DBMS厂商没有遵守该规定，当存在的数据和新指定的约束冲突时，它们不会给出警告。

⊖ 这里牵涉到嵌套和关联的子查询。若读者不理解(4.4)，请在读完第6章后再回到此处。

到STUDENT的定义中去。使用用户自定义域的总体效果是一样的，但我们无须重复定义就能在许多表中使用这个预定义的域名。而且，若以后我们修改域的定义，则这个改变会自动传播到所有使用该域的表中。域同表或断言一样，都是数据库模式的一个组件。

请注意，和断言一起，我们可使用复杂的查询定义复杂的域：

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

这里，域UPPERDIVISIONSTUDENT由Status是senior或junior的所有学生的Id组成。而且，最后的子句从域中排除NULL值。注意观察，为验证施加到该域的约束是否满足，执行针对该数据库的一个查询。由于这样的查询代价可能是很大的，所以不是每个厂商都支持这种“虚拟”的域。

4.3.7 外键约束

SQL用简单且自然的方式来指定外键。下面的语句让ProfId和CrsCode成为外键，分别引用PROFESSOR关系和COURSE关系：

```
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)  )
```

若引用和被引用的属性名字相同，那么被引用的属性可以省略。上例中的CrsCode就是这种情形。若被引用的属性名字不同于引用属性，那么两个属性的名字都应该指明。上面第二个FOREIGN KEY子句的PROFESSOR(Id)部分显示了这种情况。

应当注意的是，尽管SQL标准没有要求被引用的属性必须是主键（当然也可以是任何候选键），一些数据库厂商却会强加主键这个限制。

在上面的例子中，无论何时TEACHING关系中拥有一个课程代码，则对应该代码的课程记录一定存在于COURSE关系中。类似的，TEACHING关系中教授的Id也必须引用PROFESSOR关系中已有的元组。一旦指定这些约束，数据库管理系统会保证自动执行这些约束。因此，如果某个过程删除PROFESSOR关系中的一个元组，数据库管理系统将自动检查以保证TEACHING关系中没有相应的元组。

考虑我们讨论过的空值现象，可列出下列的问题：若某个元组其外键中的某个属性值是NULL，那会怎么样？是不是我们也应该让被引用的关系中相应的元组中其键属性值也为NULL？这不是一个好的想法，特别是当被引用的键是主键时，这种想法的缺点更为明显。因此，SQL放宽外键约束的要求，让外键可有null值。在这种情况下，被引用的关系中就不需要有对应的元组。

外键约束引发许多有趣的问题。考虑(4.3)中定义的表EMPLOYEE，假设我们还有一个表描述部门的信息：

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

现在，我们回头看EMPLOYEE表中的DepartmentId属性，很明显，该属性应当描绘一个合法的部门Id（也就是说，部门的Id要存储于DEPARTMENT关系中）。换句话说，在EMPLOYEE关系的创建语句中会有如下的语句:

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

现在问题在于，EMPLOYEE和DEPARTMENT表总有一个会首先定义。若EMPLOYEE先定义，由于它引用尚未定义的表DEPARTMENT，所以我们不能在CREATE TABLE EMPLOYEE语句中拥有上面的外键约束。若关系DEPARTMENT先定义，在语句CREATE TABLE DEPARTMENT中试图处理外键约束时数据库管理系统将发出错误警告，这是因为该外键约束引用还未创建的表EMPLOYEE。这里我们碰到了鸡生蛋还是蛋生鸡的问题。

解决问题的办法是推迟在第一个表中引入外键约束。也就是说，如果先执行CREATE TABLE EMPLOYEE，则不应该包含FOREIGN KEY子句。在CREATE TABLE DEPARTMENT语句处理完之后，再使用ALTER TABLE 指令加入我们需要的约束。后面的小节中将详细描述该指令。这里只给出最终的结果:

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

在解决了这个循环引用问题后，我们来给数据库装入数据，此时读者将碰到另一件奇怪的事情。假定我们把数据<000000007, James Bond, 7000000, B007, 000000000>放入到EMPLOYEE的第一个元组上，而此时由于DEPARTMENT表还是空的，所以规定B007应当引用DEPARTMENT表的一个合法元组的外键约束被违反。

一个解决办法是开始时将关系EMPLOYEE的所有元组其DepartmentId属性都先赋给NULL值，然后当给DEPARTMENT关系填充完合适的数据后，我们再扫描EMPLOYEE表并用合法的DepartmentId值替换NULL值。但是这种方法很笨拙，且容易出错。更好的解决办法是使用事务，并推迟检查完整性约束。

在第2章中我们曾指出，事务会产生数据库的某个中间状态，这可能会造成不一致性，即暂时地违反完整性约束。当事务提交后，约束必须被保留。为适应暂时性的约束违反，SQL允许程序员指定某个完整性约束的模式是IMMEDIATE（立即的）或是DEFERRED（推迟的）。在IMMEDIATE模式下，每当执行一个SQL语句改变数据库时，就要立即执行完整性检查；在DEFERRED的模式下，只在事务被提交时才做完整性检查。为解决循环引用这个问题，我们可以利用以下方法:

1) 在表EMPLOYEE和DEPARTMENT中将外键约束初始模式声明为DEFERRED。

2) 让写入数据到表的这些更新操作在同一个事务中。

3) 保证当所有更新完成时外键约束得到满足，否则事务结束时会异常中止。

有关在SQL中如何定义事务以及它和约束的相互作用会在第10章中详细介绍。

4.3.8 反应性约束

一般来说，当约束被违反时，相应的事务会异常中止。然而在某些情况下应该采取补救措施。外键约束就是这样的例子。

假设元组<007007007, MGT123, F1994>要插入到关系TEACHING中，由于表PROFESSOR中没有一条记录其Id是007007007，所以该插入操作将违反外键约束（它要求TEACHING中的ProfId字段的所有非空值都应当引用到现有的教授）。在这种情况下，SQL的语义很简单：插入操作被拒绝。

对于因删除一条被引用元组而违反约束的情况，SQL通常提供更多的处理手段。考虑表TEACHING中的元组t = <009406321, MGT123, F1994>，根据图4-5得知，t引用PROFESSOR关系中的Taylor教授以及COURSE关系中的Market Analysis课程。假设由于教学任务过于繁重，Taylor教授没能发表足够的论文被辞退只好离开学校。此时t会怎么样？一个解决办法是临时将t中的ProfId值设为NULL直到找到一位新讲师为止。另一个办法是设法把Taylor教授的元组从表PROFESSOR中删除，但删除失败（所以不删除），失败可能缘于不允许教授在学期中途离开。最后，若Taylor教授是唯一有能力讲授该门课的老师，我们可从课程表中把MGT123删除。这样通过删除引用元组t，违反参照完整性的问题得到解决。

这些可能性可以使用反应性约束（reactive constraint）来描述。它是一种静态约束，需要再加上当某事件发生时该做什么的规定。比如上面的第一种方法的约束要求无论何时删除一个PROFESSOR元组，则TEACHING表中引用到它的所有元组的ProfId字段必须设为NULL。第二种解决方法的约束断言若引用元组存在，则被引用的元组不能被删除。第三种解决办法断言当被引用的元组被删除时，所有引用到它的元组都应该被删除。

响应这些事件要用到触发器（trigger），触发器语句的形式如下：

```
WHENEVER event DO action
```

SQL-92只支持在外键约束中加入非常简单的触发器。实际上，SQL-92甚至不提到触发器这个名字。做法是在FOREIGN KEY子句中增加ON DELETE和ON UPDATE选项，这些选项指明若删除或更新被引用的元组时引用元组该怎么做。为解释这个问题，我们再看一下TEACHING表的定义：

```
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 )
```

上面我们指定四个触发器，分别在PROFESSOR元组被删除、被修改，当COURSE元组被删除、被修改时，触发（fired，即执行）。子句ON DELETE NO ACTION意味着若PROFESSOR表的某个教授被TEACHING中某个元组所引用，则任何试图删除该教授的操作都会被立即拒绝。当没有

指明ON DELETE或ON UPDATE子句，则NO ACTION是缺省的操作。子句ON UPDATE CASCASE意味着如果PROFESSOR表中的某个元组的Id字段值被修改，那么该修改必须被传播到TEACHING关系中引用它的所有元组中（也就是说，引用元组要存储这个新的Id值）。这样，教课的教授信息被正确记录下来。(类似的，ON DELETE CASCADE子句则要求引用元组都应该被删除)。ON DELETE SET NULL告诉数据库管理系统，当一个COURSE元组被删除且TEACHING表中有元组引用它，那么引用属性CrsCode必须被设置为NULL值。另外，数据库设计者可以指定SET DEFAULT来取代SET NULL，这意味着如果定义CrsCode有DEFAULT选项(比如前面STUDENT关系中的Status属性)，那么当被引用的元组被删除时，CrsCode属性要被设置成它的默认值；如果CrsCode没有DEFAULT选项，则它被设置为NULL值（当然，NULL值也是DEFAULT选项的默认值)。

DELETE或UPDATE触发器和NO ACTION、CASCADE或SET NULL/DEFAULT选项的任意组合都可在外键的触发器中使用。即使这样，它也没有强大到满足数据库应用程序带来的各种各样约束的要求。在(4.1)中我们曾见过一个非外键约束的参照完整性约束，它就没有办法用DELETE或UPDATE触发器实现。更重要的是，外键触发器甚至不能解决一些普通的需求(比如在事务中阻止薪水的变化幅度超过5%)。

为处理这些问题，主要的数据库厂商都在他们的产品中提供触发器机制，作为产品竞争的筹码。有趣的是，在成为标准之前，SQL中的确是提供相关的功能强大的触发器的，但只到SQL:1999才加入触发器从而形成比较完整的标准。下面我们只简要地描述一般的触发器机制，更多的细节留到第9章再进行介绍。

触发器的基本思想是很简单的：只要发生一个特定事件，就执行某些特定的动作。考虑下面在SQL:1999中定义的一个简单的触发器。只要TRANSCRIPT表中的元组CrsCode或Semester属性值发生改变，该触发器就会触发。当触发器触发，且记录的该课程的成绩不是NULL，则出现一个异常。否则，若该成绩是NULL，我们就把这个改变看作是学生退出这门课，因此触发器什么都不做，而这个改变也被保留。触发器的创建语句如下：

```
CREATE TRIGGER  CRSCHANGETRIGGER
    AFTER UPDATE OF  CrsCode, Semester  ON TRANSCRIPT
    WHEN   ( Grade IS NOT NULL )
        abortit('666', 'Grade must be NULL when registering')
```

除了WHEN子句以外，该定义是很容易理解的。WHEN子句起保护的作用，也就是说，为让触发器执行，必须满足这个前提条件。如果该前提条件为真，WHEN后面的语句会被执行。在上面的例子中，语句调用一个用户自定义的过程abortit( )，并以错误码666和消息文本作为参数。在abortit( )过程中，当检查到这个错误码时，它可能就中止事务。

一般而言，在定义触发器时还应该说明更多的细节。比如，动作执行应该发生在触发更新施加于数据库之前还是之后？在事件发生之后这个动作是要立即执行还是过一段时间再执行？触发器动作能触发另外的动作吗？而且，为在WHEN子句中指定保护，我们可能需要引用被修改的元组的新值和旧值（比如，检查薪水的修改幅度是否超过5%)。这些问题将在第9章中讨论，该章将会说明更多的触发器的例子。我们还会特别说明当面临更新时如何使用一般的触发器来维护包含依赖（类似于在维护外键约束时使用的ON DELETE和ON UPDATE)。

4.3.9 数据库视图

在4.1节中，我们曾讨论过数据库的三层抽象：物理层、概念层和外部层。上面我们已介绍过在SQL中如何定义概念层。本节我们将讨论SQL的外部层（或视图）。物理层将在第11章中讨论。

在SQL中，外部模式用CREATE VIEW语句定义。在许多方面，视图很像一个普通的表：用户可以查询它，修改它，控制对它的访问。然而，视图有几个重要的方面是不同于表的。首先，视图的行来自数据库的表（和其他视图）。因此，视图是将存放在别处的信息进行的再包装。其次，视图的内容不是物理上存储于数据库中的，如何从数据库表中构造视图的定义存储于系统目录中。很快我们就会看到，视图的定义是CREATE TABLE语句和第2章中介绍过的SELECT语句的混合体。由于这些原因，视图有时也称为**虚拟表**（virtual table）。

为解释这个问题，我们考虑下面的视图，该视图用来说明哪些教授教哪些学生（“教授教一个学生”的含义是该学生在某学期选修该教授开的课）。

```
CREATE VIEW    PROFSTUD (Prof, Stud)  AS
SELECT TEACHING.ProfId, TRANSCRIPT.StudId
FROM TRANSCRIPT, TEACHING
WHERE TRANSCRIPT.CrsCode = TEACHING.CrsCode
        AND TRANSCRIPT.Semester = TEACHING.Semester
```
(4.5)

上面第一行定义视图的名字和它的属性，随后的语句就是用来定义视图的内容的SQL查询。和图4-5的数据库实例有关的视图内容如图4-10所示。为方便读者理解该视图中的元组都来自哪里，每个元组都用justification（理由）加以注释（一条针对<*P*, *S*>的理由是课程代码和学期对组成的，表示在该学期学生*S*选修教授*P*开设这门课）。

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

图4-10　由SQL语句(4.5)定义的视图内容

PROFSTUD是外部模式的一部分，它有助于大学和它的毕业生保持联系。通过课程建立学生和教授之间的联系可能是应用中十分重要且很常见的操作。因此，以视图的形式一次性地把这种联系确定下来，就不必在每个应用程序中都要定义该联系。一旦以视图的形式定义好这种联系，所有的应用程序都能引用该视图，就好像它是一张普通的表。视图的行在它被访

问的时候构建。

第6章中，我们会详述视图机制，解释如何用视图将复杂查询的构造模块化。授权机制是视图的另一个重要内容。在本章后面，我们将说明为授权访问存储在视图中的信息，可像普通表一样对待视图。

4.3.10 修改已有的定义

尽管从理论上说，创建好特定应用的数据库模式后不应该再修改它，但在实践中，模式有时的确是会变化的，比如增加一些字段或删除一些已有的字段，创建新的约束或域，而旧的约束或域可能无效（也许是因为业务规则发生改变）。当然，我们总可以把旧模式的内容拷贝到一个临时的关系中，删除旧的关系和它的模式，然后再用原来的名字创建新的关系模式。然而，这种方法十分令人乏味且容易出错。为简化模式的维护任务，SQL提供了ALERT语句，其最简单的形式如下所示：

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

该命令在STUDENT关系中加入新的字段，并且将每个元组中该字段的值初始化为0。另外，读者还可以使用DROP COLUMN语句从关系中删除一列，还可以增加或删除约束。例如：

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

注意，如果已有的STUDENT实例违反这个新的GPARANGE约束，或TEACHING违反TEACHKEY约束，则约束的增加操作会遭到拒绝。

要从一个表定义中删除一个约束，则必须给约束命名（我们至今未用到过该选项[⊖]）。下面是修改过的TRANSCRIPT关系的定义，其中每个约束都已命名：

```
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 ))
```

现在，我们可以删除任一个指定的完整性约束，例如：

```
ALTER TABLE  TRANSCRIPT
    DROP CONSTRAINT TRKEY
```

当不再需要一个表时，可以把它的定义从系统目录中删除。此时，表的模式和它所有的实例都会丢失。以前定义过的断言和域也都被删除。例如：

⊖ 显式命名约束是可选的，后面我们会看到给约束命名的其他优势。

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

DROP TABLE命令有两个选项：RESTRICT和CASCADE。RESTRICT选项表示若在别的定义（如完整性约束）中用到该表，则删除表的操作会被拒绝。在本例中，约束THOUSHALTNOTFIREEVERYONE防止我们删除表EMPLOYEE。CASCADE选项则相反，它表示不仅删除该表定义还删除其他所有用到该表的那些定义。所以，如果我们在上面的DROP TABLE命令中使用CASCADE选项，则断言THOUSHALTNOTFIREEVERYONE也会被删除（表EMPLOYEE和约束WATCHADMINCOSTS也会被删除），这样，我们也不需要单独的DROP ASSERTION语句。

DROP DOMAIN有些微妙之处。比如，上面的例子删除域GRADES不会导致关系TRANSCRIPT的属性Grade不稳定。实际上，其效果就如同GRADES的定义首先被拷贝到所有表中用到它的地方，在DROP DOMAIN之后该定义才从系统目录中删去。

4.3.11 SQL-模式

数据库的结构由目录描述。目录的成员是表和域这样的模式对象。因此，图4-9是学生注册系统目录的简化版本。SQL-92允许目录分割为SQL-模式。**SQL-模式**（SQL-schema）[⊖]是一部分数据库的描述，该数据库由某位用户控制，而该用户有权创建和访问数据库中的对象。例如，语句

```
CREATE SCHEMA SRS_STUDINFO AUTHORIZATION JohnDoe
```

创建SRS_STUDINFO这个SQL-模式，它描述包含学生信息的部分学生注册系统数据库。AUTHORIZATION子句指定用户JohnDoe可访问在该SQL-模式中定义的表和其他对象。JohnDoe称为该SQL-模式的**授权Id**（authorization Id）。一般的做法是把同一个授权Id赋给几个不同的用户账户。

SQL-模式的命名机制类似于操作系统中的目录结构。例如，如果JohnDoe想在SQL-模式SRS_STUDINFO中创建一个STUDENT表，则他可用SRS_STUDINFO.STUDENT指代[⊜]它。

SQL-92没有规定SQL-模式的信息必须以何种格式存储，但它要求每个目录必须包含一个名为INFORMATION_SCHEMA的特定模式，该模式的内容需精确地指明。INFORMATION_SCHEMA包含一组SQL表，它们很精确地复制目录中所有其他SQL-模式的定义。INFORMATION_SCHEMA中的信息可以被任何授权用户访问。

最后，我们注意到SQL-92定义**聚簇**（cluster）作为目录的集合，聚簇用来描述可以被单个SQL程序访问的数据库集合。这样，在大学这个例子中，可以定义一个聚簇来描述大学维护的所有数据库，它们可以被单个的事务访问。

4.3.12 访问控制

数据库常常包含一些敏感信息。因此，必须保证只有经授权的用户才能访问系统，并且

⊖ 注意这里的“模式”在意义上不同于关系模式或数据库模式中的“模式”，我们用SQL-模式指代SQL用法。

⊜ SQL-92有精心构造的默认命名机制，所以在许多情况下，是不需要两层的命名限定的。

保证用户只能访问他们有权访问的信息。许多事务处理系统会提供大量的认证和授权机制，访问之前必须先进行认证。可能的手段包括提供口令给数据库管理系统，或者采用涉及到独立的安全服务器的更为复杂的方案（如第27章中所描述的例子）。当认证完成后，就认为用户已经和一个授权Id正确地联系在一起，可以开始对数据库的访问。

权限被赋予某个特定的授权Id。当然多个用户可以有相同的授权Id，此时他们拥有同样的权限。表或其他对象的创建者拥有该表或对象（他们被称为所有者），这样他们也就拥有关于该对象的所有权限。所有者可以使用GRANT语句把关于该对象的某项权限授予其他用户（即，其他授权Id），GRANT语句的形式如下：

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

这里，WITH GRANT OPTION意味着权限接受者可以进一步的把他得到的权限授予他人。

如果对象是表或视图，则*privilege-list*包括：

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

前面四个选项为执行指定的语句授予权限。包含(*column-comma-list*)的选项只为指定的列授权。比如，若INSERT权限已被授予，那么在插入的元组中中，只可以出现*comma-list*中的属性值。所有的*comma-list*都是可选的，这由中括号标志出来。

REFERENCES授予使用外键来引用一个表或列的权限。控制这种访问类型看起来有点奇怪，但是如果不控制外键约束则会存在安全隐患。随意设置外键约束会引发两个问题。假定一个学生可以创建如下的表：

```
CREATE TABLE  DontDismissMe  (
    Id   INTEGER,
    FOREIGN KEY (Id) REFERENCES Student)
```

如果该学生在DontDismissMe中插入一个包含她自己的标识Id的行，则注册人员不能“开除”该学生，也就是说，不能从表Student中删除该学生所在的行。这是因为删除会违反参照完整性，因而被数据库管理系统拒绝。

不受限制的外键访问还会导致安全漏洞。假如为保护学生信息，对Student表的SELECT访问权限只赋给大学注册办公室中的工作人员。然而，如果一个入侵者能够创建上面所示的表（表的名字可以是ProbeProtectedInfo），那么就需要这种限制不让该入侵者得逞。否则，入侵者可以试图在ProbeProtectedInfo表中插入某个Id，如果该插入操作被数据库管理系统接受，他就可以知道该Id在表Student中也存在，即对应此Id的个人是一个学生，否则根据参照完整性必然会导致违例。类似的，若该插入被数据库管理系统拒绝，那么这个Id对应的人不是学生。所以用这样的手段，即是入侵者没有权限对表Student执行SELECT操作，他也能截取到一些信息。

一个GRANT语句的例子如下所示：

```
GRANT SELECT, UPDATE (ProfId) ON StudRegSystem.Teaching
    TO JohnSmyth, MaryDoe WITH GRANT OPTION
```

该语句授予John Smyth和Mary Doe从表Teaching中读取一行和更新ProfId字段的权限，但不允许删除和增加行。并且，允许他们把自己获得的权限传递给别的用户。

SQL不仅允许控制对数据库的直接访问，还能够通过视图控制对数据库的间接访问。举个例子，下面的语句允许所有的校友（alumnus）对(4.5)中定义的视图ProfStud作无限制的查询访问：

```
GRANT SELECT ON ProfStud TO Alumnus
```

然而，校友们不能把查询权利传递给别人，也不能更新该视图。更有意思的是，尽管这些用户能间接访问到视图显示的表的信息，他们却可能没有权利访问Transcript和Teaching表，而正是这两张表给ProfStud提供内容。这方面的问题在第6章会进一步讨论。

除针对表之外，还能针对其他对象（比如域）授予权限，这里不再详述。

权限或者授予权限的选项可以用REVOKE语句撤销。

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

语句中CASCADE的含义是，如果某个用户比如U_1（他得到*authID-list*中一个权限Id）把权限赋予另外一个用户如U_2，则赋予U_2的权限也应该被撤销，如此一直进行下去直到没有用户拥有权限为止（假设U_2又将权限赋予其他用户）。而RESTRICT意味着，如果权限存在这样的相互依赖，那么REVOKE语句被拒绝。

在许多应用程序中，只在数据库操作级（比如SELECT或UPDATE）授权是不够的。比如，只有存款人能在他的银行账号存钱，只有银行的官员可以给该账号增加利息，但是存款事务和利息事务都可能用到同样的UPDATE语句。对这种应用，在子例程或事务这一级进行授权是更恰当的。许多事务处理系统会在该级控制访问。第27章我们会再介绍该主题。

4.4 参考书目

关系数据库模型由[Codd 1970, 1990]提出。[Codd 1979]在原始模型上提出多种扩展，这些扩展模型提供了丰富的语义信息。

早期有两个系统实现并扩展了关系数据库模型：System R[Astrahan et al. 1981]和INGRES[Stonebraker 1986]。最后，System R演变成IBM的商业产品DB2，而INGRES也成为一家同名的商业公司的产品（当前出售给Computer Associates, Intl.）。

关系数据库理论现在已经极为丰富，许多理论成功地形成研究原型甚至商业产品。更深入的讨论和本书没有涉及到的主题可以在[Maier 1983, Atzeni and Antonellis 1993, Abiteboul et al. 1995]中找到。

4.5 练习

4.1 联系关系数据库的定义，说明数据原子性，并和事务处理系统中的事务原子性进行比较。

4.2 证明每个关系都有一个键。

4.3 给出下面的概念的定义：

a. 键
b. 候选键
c. 主键
d. 超键

4.4 定义以下概念:

a. 完整性约束
b. 静态完整性约束，并和动态完整性约束进行比较
c. 参照完整性
d. 反应性约束
e. 包含依赖
f. 外键约束

4.5 在某个特定时刻，查看存储在表中针对某个特定应用程序的一些数据，解释你是否可以说出:

a. 该表的键约束是什么。
b. 某个属性是否是键。
c. 该应用程序的完整性约束是什么。
d. 某个特定的完整性约束的集合是否被满足。

4.6 使用SQL的DDL说明图4-4中的学生注册系统的模式，并包括图4-6和图4-8中的约束。为属性的域指定较小的范围，如DeptId和Grade属性。

4.7 考虑一个数据库模式，它有四个关系: SUPPLIER、PRODUCT、CUSTOMER和CONTRACTS。SUPPLIER和CUSTOMER有属性Id，Name和Address。Id是一个9位的数字。PRODUCT有属性PartNumber（整型，取值范围在1～999999之间）和Name。CONTRACTS的每个元组针对某个产品对应供应商和顾客之间的一份合同，合同中有数量和价格信息。

a. 用SQL DDL说明这些关系的模式，包括合适的完整性约束（主键，候选键和外键）以及SQL域。
b. 用SQL的断言语句说明下面的约束: 合同必须比供应商多。

4.8 假设系统中有2个账户: STUDENT和ADMINISTRATOR。用SQL的GRANT语句为他们指定合适的权限。

4.9 解释必须要REFERENCES权限的原因，并给出一个例子说明通过创建引用该关系的外键约束从而获得该关系的部分内容是可能的。

第5章 数据库设计I：实体-联系模型

既然学生注册系统的规格说明书已经被校方认可，现在我们就可以开始系统的数据库部分的设计。在第12章我们会介绍事务处理应用的完整设计过程，包括这个过程的最终产品——设计文档。这里我们讨论数据库部分的设计，它得到的结果是声明数据模式（如表、索引、域和断言等）的完整的（可编译的）CREATE语句的集合。学生注册系统的完整设计参见5.7节。

数据库设计的关键问题是必须准确地把一个企业多方面的问题建模到关系数据库中，该关系数据库可被众多应用中的大量并发执行的事务高效地访问和更新，这些应用甚至涉及决策支持查询。同其他的工程学科一样，利用特定的方法学设计过程会更为容易，而且可以根据某些客观标准来评估它。

在本章中，我们介绍一种设计关系数据库的流行方法：*实体-联系（E-R）方法*，它是由[Chen 1976]提出的。我们会在第8章中再次介绍数据库设计的内容，该章介绍*关系规范化理论*，它给出如何评估各种可选设计的客观标准。

还应当注意，E-R方法和关系规范化理论的许多基本机制已经由计算机程序完成，因此，它减轻了设计者解决常规设计问题的任务。尽管如此，设计数据库仍需要良好的创造力，专门的技术和经验，并且掌握数据库设计的基本理论。

5.1 E-R方法的概念建模

为消除一个常见的误解，我们必须强调E-R方法不是相关的、派生的或一般化的关系数据模型，实际上，它根本不是数据模型，而只是适用于（但不只限于）关系模型的*设计方法学*。这里，术语“联系”指的是该方法学中的两个主要组件之一，而不是关系数据模型。

E-R方法的两个主要组件是**实体**（entity）和**联系**（relationship）的概念。实体是对企业中涉及的对象的建模，比如大学中的学生、教授和课程。联系是对这些实体之间的关联的建模，比如教授“授”课。另外，实体和联系上的**完整性约束**也是E-R规格说明的重要方面，其重要性相当于完整性约束对于关系模型的重要性。比如，一个约束规定教授在一天中的某个时刻只能教一门课。

实体-联系图（E-R图）（请看一下图5-1和图5-3）是组成设计的实体、联系和约束的图形表示。同其他面向可视化设计的方法学一样，E-R图提供的图形化设计概括对设计者尤为重要，不仅可验证设计的正确性，而且可供他们和同事进行讨论，并用该图向程序员解释。但是，E-R图的画法还没有相应的标准，因此在数据库教材中E-R方法的许多方面有各种各样的变体。

当企业用E-R图建模完毕，那么将图转化为相应的CREATE TABLE语句就比较直接。然而，这样的转化不会生成唯一的模式，特别是对于约束来说情况更是如此，有些能用E-R图表

示的约束在SQL中没有直接的对应。这些相关问题会在适当时候再进行讨论。

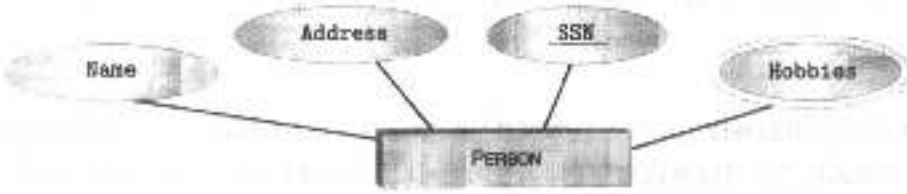


图5-1 实体类型PERSON的E-R图片断

在为企业建模时，E-R方法学中富于创造性的地方是决定使用哪些实体、联系以及约束。本书中列举的简单例子可能使人认为这很容易做到，但实际上，设计者不仅需要丰富的技术知识、判断和经验，还需要很好地理解企业的运作方式。

E-R方法学的一个重要好处是设计者可以集中精力考虑企业完整和精确的建模，而不必担心最终的数据库执行查询或更新时是否高效。此后，当E-R图被转化为CREATE TABLE语句时，设计者再用第8章中的规范化理论和本章以及第13章讨论的一些技术来考虑最后设计的表在性能方面的要求。

5.2 实体和实体类型

E-R方法的第一步是选择企业建模中要使用的实体。实体很类似于对象（见附录A），只是实体没有方法。实体可以是现实世界中的具体对象，比如John Doe这个人，停在Main大街123号的卡迪拉克轿车或帝国大厦等，也可以是抽象的对象，比如花旗银行的账户123456789，数据库课程CS305或SUNY Stony Brook大学的计算机科学系。

类似的实体可聚合成**实体类型**（entity type）。比如，John Doe，Mary Doe、Joe Blow和Ann White可聚合为实体类型PERSON（人），因为这些实体都表示人。John Doe和Joe Blow还可以属于实体类型STUDENT，因为在第4章的数据库例子中我们看到它们表示学生。同样地，Mary Doe和Ann White可归类到实体类型PROFESSOR中。

其他的实体类型有：

- CS305、MGT315和EE101，它们属于实体类型COURSE。
- Alf和E.T.，它们属于实体类型SPACEALIEN。
- CIA、FBI和IRS，它们属于实体类型GOVERNMENTAGENCY。

1. 属性

和关系（对象）相同，实体用属性描述。实体的每个属性确定该实体的某项性质。比如，PERSON实体的Name属性通常指定一串字符来代表由数据库实体表示的人在现实世界中的名字。类似的，Age这个属性指定从现实世界中的人出生那刻起地球围绕太阳已转过的次数。

上面所有实体类型的例子本质上是符合语义的，也就是说，实体类型由语义上相关的实体集合组成。比如，把人、车和纸夹归类到同一个实体类型通常是无意义的，因为在一个典型的企业建模中它们几乎没有共同点。把语义上相近的实体归为一类才更有用，因为它们很可能有共性。例如，在企业里，人总是有许多共性，比如名字、年龄、地址等。把实体类型分类可以允许我们将属性与实体类型关联而不是与实体本身关联。

当然，不同的实体类型有不同的属性集。比如，PAPERCLIP（纸夹）这个实体类型可能有Size（尺寸）和Price（价格）这些属性，而COURSE的属性可能是CrsName、CrsCode、Credits和Description。

2. 域

和关系模型中的域概念相同，实体属性的域指它可以取的值的集合。然而和关系模型不同的是，E-R属性的值可以是值集合，即属性的值可以取相应域上的一组值而不仅仅是单个值，比如，实体类型PERSON可以有取值集合的属性ChildrenNames和Hobby。

不能方便地取值集合是关系数据模型主要缺点之一，这也激发了对面向对象数据建模的研究。然而，在E-R模型中使用值集合的属性仅是方便而已。稍加努力，关系（如第4章中定义的）就能用于带值集合属性的实体建模。

3. 键

对照关系模型中键，在E-R方法中引入和实体类型相关的键约束是有用的。实体类型S的**键约束**（Key Constraint）是S的属性值$\overline{A}$，它满足以下条件：

1) 没有两个S的实体对于$\overline{A}$中的每个属性有相同的值（例如，两个不同的COMPANY实体其Name和Address属性值均不可相同）。

2) $\overline{A}$中没有子集具有上面第1条中的性质（也就是说，$\overline{A}$是关于该性质的最小集）。

很明显，实体键的概念类似于关系模型中的候选键。但有一个细微之处，E-R方法中的属性可以是值集合，而原则上这样的属性也可以是键的一部分。然而，实践中让键中的属性取值集合是不大自然的，这往往表示设计做得不好。

4. 模式

对照关系模型中的模式，E-R方法中的**模式**（schema）定义为包含类型名、属性集（以及相关联的域和每个属性取值是单值还是值集合的指示标志）和键约束的实体类型。

5. E-R图表示

在E-R图中，实体类型用矩形表示，属性用椭圆表示，而取值集合的属性用双椭圆表示。图5-1描述实体类型PERSON的E-R图，图中Hobby属性的值取值集合，“SSN”加下划线表示它是键。

6. 关系模型中的表示法

E-R模型中的实体和关系模型中的关系之间的对应是十分简单的。每个实体类型转化为一个关系，每个实体属性转化成关系的属性。

但是，这样简单的转换可能会令人怀疑，因为实体的属性可能会取值集合，而关系的属性却不能。如果不破坏关系模型的数据原子性（见4.2节），怎样才可以把取值集合的实体属性转化为相对应关系的单值的属性？

问题答案是，在转化时需要用一组元组代表拥有取值集合的属性的实体，每个元组对应属性的值集合中的一个元素。为说明这一点，假设图5-1中的实体类型PERSON具有下面的实体数据：

```
(111111111, John Doe, 123 Main St., {Stamps, Coins})
(555666777, Mary Doe, 7 Lake Dr., {Hiking, Skating})
(987654321, Bart Simpson, Fox 5 TV, {Acting})
```

转化后，我们获得图5-2所示的关系。

Person	SSN	Name	Address	Hobby
	111111111	John Doe	123 Main St.	Stamps
	111111111	John Doe	123 Main St.	Coins
	555666777	Mary Doe	7 Lake Dr.	Hiking
	555666777	Mary Doe	7 Lake Dr.	Skating
	987654321	Bart Simpson	Fox 5 TV	Acting

图5-2 实体类型PERSON转化成的关系

接下来的问题是，经过上面的转化，该为关系指派什么样的键。如果实体类型没有值集合的属性，那么答案是简单的：实体类型的键（它是属性的集合）就是对应关系模式的键。然而，如果实体的某个属性是取值集合的，则确定键会有点棘手。在实体类型PERSON中，因没有人会有相同的SSN号，所以属性SSN是键，而Hobby属性的取值是所有Hobby值的某个集合。转化为关系后，图5-2中John Doe和Mary Doe由一对元组表示，他们的SSN号都出现两次，因此，SSN就不能再作为关系的键。

显然，值集合属性Hobby是麻烦的始作俑者。为得到关系的键，我们就要把属性Hobby包括进来，这样，图5-2中关系PERSON的键是{SSN, Hobby}。

下面CREATE TABLE语句用于定义PERSON关系的模式：

```
CREATE TABLE  PERSON  (
    SSN       INTEGER,
    Name      CHAR(20),
    Address   CHAR(50),
    Hobby     CHAR(10),
    PRIMARY KEY (SSN, Hobby) )
```
(5.1)

即使我们最终找到这样的键，但若仔细分析一下上面的表还是会发现一些问题。首先，我们觉得不应该用Hobby属性确定PERSON关系中的元组。其次，在最初的实体类型PERSON中，任意一个SSN的值都可以唯一确定Name和Address的值，但到图5-2中，虽然该性质还存在，但它已不能由主键约束所反映，这说明为唯一确定一个元组，我们必须指定SSN和Hobby的值。而在原来的实体类型PERSON中，是不需要用Hobby的值来决定Name和Address的值的。换言之，这个重要的约束在转化过程中丢失!

前述的例子已经表明只使用E-R方法是不能保证良好的关系设计的。第8章提供许多客观标准，它们可以帮助数据库设计者评估从E-R图转化而来的关系模式。特别地，图5-2的关系存在的问题在于它不是一个第8章定义的所谓的“范式”。该章所介绍的算法能自动地调整这些问题，方法是将有问题的关系分割成较小的符合范式的关系。

5.3 联系和联系类型

在E-R方法中，在实体和关联这些实体的机制之间有明显的差别。该机制称为**联系**(relationship)。如同实体归类为实体类型，关联相同类型实体和有同样含义的联系也可以分

组为联系类型（relationship type）[⊖]。

比如，STUDENT实体和PROGRAM实体之间的联系类型是MAJORSIN（主修）。同样，PROFESSOR实体和他们工作的系之间的联系类型是WORKSIN（工作于）。

我们需再一次强调，E-R方法中联系（relationship）的概念不同于关系数据模型中的关系（relation）概念。联系只是E-R方法中的一个建模工具。当涉及到实现时，实体和联系都用数据库管理系统中的关系（也就是表）表示。

1. 属性和角色

与实体类似，联系也有属性。例如，联系MAJORSIN可能会有属性Since（自从）表示学生获准学习该专业的日期。联系WORKSIN可以用属性Since指示雇佣的起始日期。

属性不能提供有关联系的完整的描述信息。考虑实体类型EMPLOYEE和联系REPORTSTO（它是雇员间的联系），雇员的两种类型是下属和老板。这样，若说<John, Bill>是类型REPORTSTO的一个联系，我们仍然无法知道谁应该向谁汇报。

将EMPLOYEE实体类型分成SUBORDINATE（下属）和SUPERVISOR（主管）也无济于事，这是因为REPORTSTO可能表示公司结构中的整个汇报链，这使得某些雇员同时既是下属也是主管。

解决方法是必须认识到各种实体类型在联系中扮演不同的角色。对每种参与其中的实体类型，我们定义一种**角色**（role）并为之命名（比如，Subordinate）。角色类似于属性，但它不是说明联系的某种性质而是说明实体类型以什么样的方式参与到联系中去。角色和属性都是联系类型模式的一部分。

举个例子，联系类型WORKSIN有两个角色：Professor和Department。角色Professor确定WORKSIN联系中的PROFESSOR实体，Department角色确定该联系中的DEPARTMENT实体。类似的，联系类型MAJORSIN也有两种角色：Student和Program。

如果一个联系中的所有实体属于不同的实体类型（就像在WORKSIN和MAJORSIN中那样），则没有必要显式地指出角色，这是因为我们总能采取一些约定，如在相应的实体类型之后命名角色（实践中经常这么做）[⊜]。然而，若一些实体来自同一个实体类型，则这种简单的手段是不可行的，REPORTSTO联系就是一个例子。这里，我们都明确地标明角色，如Subordinate和Supervisor。图5-3演示联系的几个例子，其中将角色清楚地标明出来。

综上所示，**联系类型的模式**（*schema of a relationship type*）包括：

- 若干属性以及相应的域。属性可以是单值或值集合。
- 若干属性以及相应的实体类型。与属性不同，角色总是单值的。
- 一组约束。这在后面描述（在图5-3中，用箭头表示的是约束）。

一个联系类型中参与的角色的数量称为类型的**度**（degree）。

现在，我们更准确地定义联系的概念。度为n的联系类型$\mathbf{R}$由它的属性$A_1,\cdots,A_k$和角色$R_1,\cdots,R_n$所定义。$\mathbf{R}$的联系定义为如下形式的元组集：

$$< \mathbf{e}_1, \mathbf{e}_2, \cdots, \mathbf{e}_n, ; a_1, a_2, \cdots, a_k >$$

⊖ 后文若是在联系和实体之间，以及联系类型和实体类型之间出现导致混淆的地方，则就用术语“联系实例”和“实体实例”分别代替“联系”和“实体”。

⊜ 为免混淆，我们用不同的字体区分实体类型和它们在各个联系中扮演的角色。

其中，实体$e_1, e_2, \cdots, e_n$分别是角色$R_1, \cdots R_n$的值；$a_1, a_2, \cdots, a_n$分别是属性$A_1, \cdots A_n$的值。而且我们还假定联系中属性的所有值都处于它们各自的域内，所有的实体都有正确的实体类型。

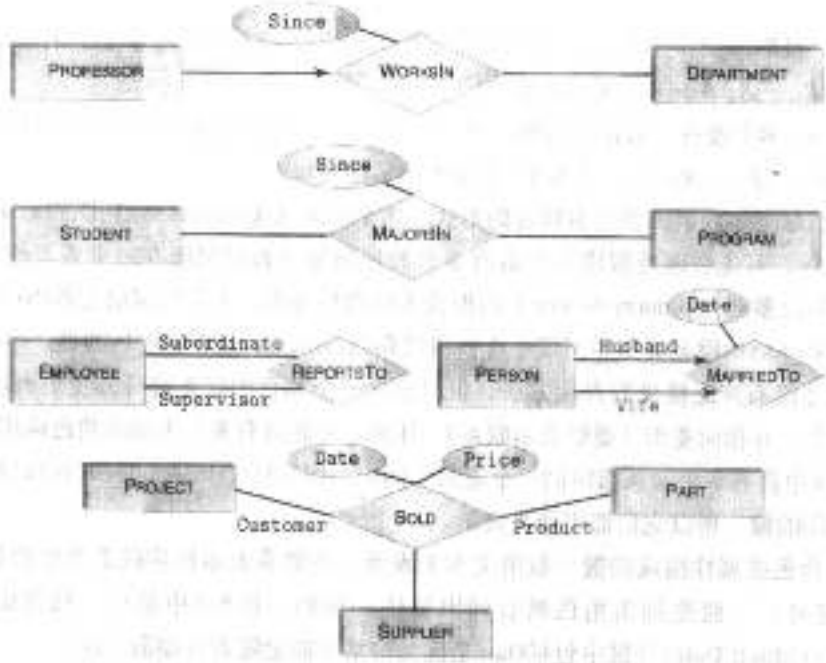


图5-3 几个联系类型的E-R图

比如联系类型MAJORSIN可有模式：

(Student, Program, Since)

这里Student、Program是角色，Since是属性。这个联系类型的一个实例可能是：

(Homer Simpson, EE, 1994)

该联系说明，实体Homer Simpson是一个学生，他在1994年登记入学，学习的专业课程用实体EE表示。元组中前两部分是实体，而最后一个是取自年份这个域的普通值。

2. E-R图表示

在E-R图中，联系类型用菱形表示，角色用连接联系类型和实体类型的边表示，如果角色必须显式地命名（这是由于联系涉及到的实体取自同样的实体类型），那么角色名字也应该在图里给出。图5-3显示我们讨论过的几个联系的E-R图（为清楚起见，所有的实体属性都省略）。图中前三个联系是二元（binary）的，因为每个联系都关联两个实体类型，最后一个联系是三元（ternary）的，因为它关联三个实体类型。最后一个图也说明若缺省的角色名字（该例中是Project和Part）被重命名（如分别是Customer和Product），那么图表的含义有时会更容易表达。

3. 键

联系的键能使设计者容易、自然和一致地表达许多约束[⊖]。对于实体类型来说，键是一组

⊖ 大多数教材利用多对一基数约束定义有限形式的联系键。

唯一标识实体的属性。然而，单单用属性不能完全地刻画联系，还应该把角色考虑进去。我们定义**联系类型R的键**为R的属性和角色的最小集，该最小集的值唯一地标识联系类型的联系实例。

换言之，设$R_1,\cdots,R_k$是R的所有角色集的子集，$A_1,\cdots,A_s$是R的属性子集，则当以下条件成立时，集合$\{R_1,\cdots,R_k, A_1,\cdots,A_s\}$是R的键：

1) 唯一性。R中没有一组联系实例在$\{R_1,\cdots,R_k, A_1,\cdots,A_s\}$中的属性和角色值都相同。

2) 最小性。$\{R_1,\cdots,R_k, A_1,\cdots,A_s\}$的子集都不满足性质1。

在某些情形下，联系的键会有特殊的形式。考虑实体类型PROFESSOR和DEPARTMENT之间的联系WORKSIN，可先合理地假设每个系有多个教授但每个教授只能在一个系工作。换言之，联系WORKSIN以**多对一**（many-to-one）的形式关联教授与系。由于任何给定的PROFESSOR实体最多只能在WORKSIN联系类型中出现一次，所以角色Professor是WORKSIN的键。尽管在E-R图中联系的键还没有广泛接受的表示方法，但只包含一个角色的联系键可方便地表示为：用箭头代表该角色，并指向菱形（菱形表示联系）。注意，可能会有多个不同的角色构成联系的键，所以在E-R图中会有多个箭头指向同一个菱形。例如，图5-3中{Husband}和{Wife}都是联系类型MARRIEDTO的键，所以它们都用箭头表示。

由多个角色或属性组成的键一般用文本来表示，并紧靠表示相应联系类型的菱形。当表示这样的键时，一般先列出角色然后列出属性。例如，图5-3中最后一张图的键可能是{Customer, Product; Date}（键中包括Date是因为价格可能会随着日期而浮动）。

然而在许多情况下，联系的键会正好是所有角色的集合并且是唯一的，这样我们就不需要在E-R图中指定键。

4. *在关系模型中的表示*

把联系类型R映射到关系模式的步骤如下：

- *关系模式的属性。*关系模式的属性是：

  1) R的自身属性。

  2) 对于R中的每个角色，相关联的实体类型的主键。

  注意，这里我们用实体类型的主键，而不是构造于该实体类型的关系模式的键。其原因是，我们的目标是确定联系中涉及的实体。这样，就和实体类型PERSON相联系的角色而言，只用主键SSN而忽略Hobby。

- *关系模式的候选键。*大多数情况下，关系模式的键从R自身的键的直接转换取得。即，若R的角色R属于R的键，则和R相关的实体的主键K的属性一定属于从R转化来的关系模式的候选键。

  如果R有值集合的属性，则会有些小问题。若是这样，我们需求助于早先用于转化实体键为关系键的小窍门：不管那些值集合的属性是否包含在R的键中，关系的候选键都应该包含它们（请参阅PERSON实体-关系转化的例子）。注意，角色总是单值的，所以这种处理方式不适合于角色。

- *关系模式的外键约束。*由于在E-R模型中，角色总是引用一些实体（实体映射到关系），所以角色被转化为外键约束。对应R的关系模式的外键构造方法如下：

对于R中的每个角色，其关联的实体类型的主键（它在与R对应的关系模式的属性之中）成为R的关系模式的外键，该外键引用到这个实体类型转化成的关系。

还有一个问题是实体类型的主键（例如SSN）不成为对应关系的主键（例如{SSN, Hobby}），这个问题由第8章中的规范化理论来消除。

图5-4中的CREATE TABLE命令创建图5-3中联系所对应的关系模式。图中，在关系MARRIEDTO中，没有定义外键约束，这是因为尽管SSNhusband和SSNwife引用PERSON关系的SSN属性，但SSN不是该关系的候选键。模式中的UNIQUE约束保证SSNwife在表的任何实例中是唯一的，这样它可作为候选键。

```
CREATE TABLE  WORKSIN  (
     Since          DATE,
     ProfId         INTEGER,
     DeptId         CHAR(4),
     PRIMARY KEY (ProfId),
     FOREIGN KEY (ProfId) REFERENCES PROFESSOR (Id),
     FOREIGN KEY (DeptId) REFERENCES DEPARTMENT )

CREATE TABLE  MARRIEDTO  (
     Date           DATE,
     SSNhusband     INTEGER,
     SSNwife        INTEGER,
     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) )
```

图5-4 部分联系的转化

我们看到，当E-R图转换为表时，一些表描述实体，而另外一些表描述联系。从而，图5-3的第一个表翻译成三张表：一张表描述教授的实体类型，一张表描述系的实体类型，第三张描述教授工作于不同系的联系类型。

5.4 E-R方法的高级特性

5.4.1 实体类型层次结构

某些实体集合之间会存在紧密的联系。比如，所有STUDENT实体集中的每个实体也都是PERSON实体集的一个成员。因此，PERSON实体集的所有属性也适用于STUDENT实体。而学生实体可能会有PERSON实体没有的属性（比如，Major，StartDate，GPA）。这种情况下，我们把STUDENT实体类型称为PERSON实体类型的子类型。

正式地说，实体类型**R**是实体类型**R'**的**子类型**（subtype）相同于指定一个实体间约束，它意味着:

1) **R**的每个实体实例也是**R'**的实体实例。

2) **R'**的每个属性也是**R**的属性。

上面定义的一个重要推论是超类型的任何键也是它所有子类型的键。

请注意，子类型化不仅是约束还是联系。该联系可以用角色Sub（类型）和角色Super（类型）表示，通常称之为IsA（是）联系。这样，在IsA联系中角色的名字是固定的，但它们的范围取决于和某个IsA联系相关的实体类型。

例如，在关联STUDENT和PERSON的IsA联系类型中，Sub的范围是STUDENT，Super的范围是PERSON。该联系的某个实例可能是<Homer Simpson, Homer Simpson>，它说明Homer Simpson既是学生也是人。注意，IsA中涉及的实体类型总是一致的。

那么IsA联系有什么用呢？回答是子类型约束引入E-R模型中的分类层次结构(classification hierarchy)。例如，FRESHMAN是STUDENT的子类型，而STUDENT又是PERSON的子类型。该性质是*传递*的，即FRESHMAN也是PERSON的子类型。传递性质使得E-R图更易读，其画法更加简洁。由子类型化的性质2可知，PERSON的每个属性也都是STUDENT的属性，根据传递性它们也是FRESHMAN的属性。该现象也经常表述为，STUDENT**继承**（inherit）PERSON的属性，而FRESHMAN则既继承PERSON又继承STUDENT的属性。

注意，这里并没有为实体类型STUDENT和FRESHMAN显式地指定被继承的属性（SSN，Name等），但因为是IsA联系所以它们还是有效的属性。除被继承的这些属性外，STUDENT和FRESHMAN可能还有它们自己的属性，而它们相应的超类型是没有这些属性的。图5-5解释了这一点，图中STUDENT是PERSON的子类型，它继承来自PERSON的所有属性，因此就没有必要为STUDENT类型重复指定属性Name和D.O.B.（出生日期）。类似的，FRESHMAN，SOPHOMORE等是STUDENT的子类型，因此也没有必要拷贝STUDENT和PERSON的属性给它们。可见，传递性大大地简化了E-R图。图中IsA树的EMPLOYEE分支给出了属性继承的又一个例子。联系“EMPLOYEE IsA PERSON”表明每个EMPLOYEE实体都有属性Department和Salary。此外，它还说明每个EMPLOYEE实体也是一个PERSON实体，因此也有属性Name，SSN等。

注意图5-5中的每个IsA三角形代表不同的联系类型。例如，最上面的三角形表示联系类型“EMPLOYEE IsA PERSON”和“STUDENT IsA PERSON”。尽管这种表示法使得IsA联系的表示不同于其他联系的表示，但由于实体类型层次结构中存在多种约束，用该符号表示它们显得比较自然。

例如，属于实体类型FRESHMAN，SOPHOMORE，JUNIOR和SENIOR的所有实体可能就是类型STUDENT的所有实体（比如在四年制的大学里）。这种约束称为**覆盖约束**（covering constraint），它很容易地附加到右下方的IsA三角形中。并且，这些实体集可能永不相交（在绝大多数的美国大学里是这样的），这样的**不相交约束**（disjointness constraint）也可以附加到右下方的三角形中[⊖]。

⊖ E-R图中没有普遍接受的表示覆盖和不相交约束的方法，所以读者可以用自己喜欢的形式来表示。

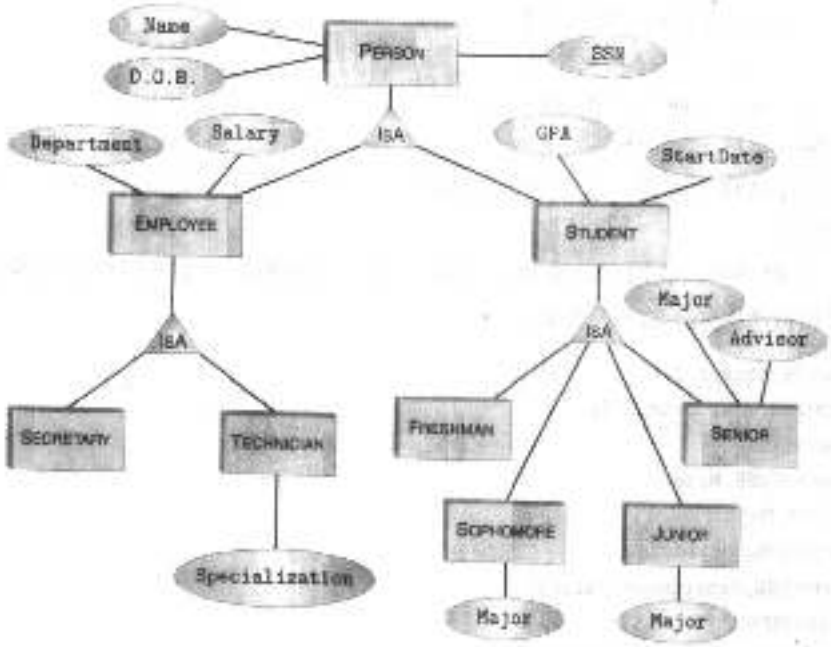


图5-5 IsA层次结构的例子

I. 实体类型层次结构和数据分段

上面针对IsA联系的讨论集中在概念组织和属性继承上，然而，这种层次结构还是处理物理**数据分段**（data fragmentation，或者**数据分区**（data partitioning））问题的一种好办法。数据分段的需求经常来自于分布式环境。在这种环境下，多个地理区域相异的实体访问共同的数据库。银行就是一个典型的例子，因为不同的城市存在很多分行。

在分布式企业里出现的问题是网络延时：由于频繁地执行事务，可能禁止从布法罗的银行分行访问数据库。然而，当地分行需要的数据块最好能从本地访问，所以比较好的办法是分发数据库中这样的信息，并让单个分行维护它们。第18章中我们会详细地讨论在分布式数据库中的数据分段，其他使用数据分段的地方在24.3.1节讨论。

我们看看在数据库设计阶段怎样处理数据分段。考虑实体类型CUSTOMER，它表示银行顾客的信息。对于每个分行，我们可创建子类型，例如NYC_CUSTOMER和BUFFALO_CUSTOMER，它们与CUSTOMER之间的关系如图5-6所示。

我们观察到，和类型层次结构相关的约束为确定数据如何分段提供了相当强大的表达功能。例如，图5-6可解释成这样的需求，即纽约市和布法罗的数据必须本地存储。这不是说纽约市的数据库和布法罗的顾客数据库必须是不相交的，而是说该限制可由早先阐述过的不相交约束所确定。另外，我们还可以增加覆盖约束来规定如果把所有的分行客

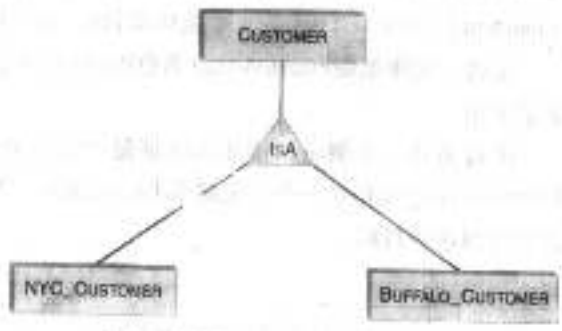


图5-6 用到IsA联系的数据片断

户合并起来那么就得到银行的所有客户的信息。

2. 在关系模型中表示IsA层次结构

有几种方法可表示使用关系表的IsA联系。普遍采用的一种办法是先为表示IsA联系的树的分支上的所有实体类型选择一个候选键，并为每个实体类型加入键属性，然后如5.2节所述，把分支上的实体转化为关系。这里这样选择键是可行的，因为正如前面提到的，超类型的键也是子类型的键。

例如，在图5-5中，我们可以选择{SSN}作为键，把属性SSN加入到每个实体类型中。之后的转化过程将产生如下的关系模式:

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)

当存在不相交和覆盖这样的约束时，会有更有效的方法来表示这些信息。例如，如果STUDENT的所有子实体有相同的属性集（就是说，如SENIOR没有它自己的属性Advisor），那么可以不用5个关系表示学生，而就用一个关系表示，该关系有一个特殊的属性Status，它的取值范围是常量集{Freshman, Sophomore, Junior, Senior}[⊖]。

5.4.2 参与约束

考虑实体类型PROFESSOR和DEPARTMENT之间的联系类型WORKSIN，前面已经说过，每个系可有多个教授，但每个教授只能为一个系工作，因此角色PROFESSOR是WORKSIN的键。

该键约束确保没有一个教授在类型WORKSIN的一个以上的联系中出现，然而它没有办法保证一个教授一定会出现在某个联系中，换言之，该键约束不能排除某教授没能为某个系工作的情况（这样他就可能因没有授课而被辞退！）。为解决这个问题，设计者可使用*参与约束*。

给定实体类型**E**，联系类型**R**和一个角色ρ，**R**中角色ρ的E的**参与约束**（participation constraint）表示，对于**E**的每个实体实例**e**，存在**R**中的一个联系**r**，满足**e**以角色ρ参与**r**。

显然，实体类型PROFESSOR以角色PROFESSOR参与联系类型WORKSIN，以保证每个教授都在某系工作。

再看另外一个例子，例如为保证每个学生至少要学习一门课，先假设STUDENT、COURSE和SEMESTER之间存在一个三元联系TRANSCRIPT，我们可以为STUDENT实体类型施加一个参与约束来达到这个目标。

⊖ 即使SENIOR有自己的属性该转化也能完成，但对那些不是大四（senior status）的学生他们的Advisor属性应该置NULL值。

在E-R图中，参与约束由连接参与实体和对应联系的粗线表示。如图5-7所示，连接PROFFSSOR与WORKSIN的粗箭头不仅表示每个教授至少参与一个联系（由粗线标出），还表示每个教授至多参与一个联系（由箭头标出），这也意味着PROFESSOR实体和联系之间的对应关系是一对一的。

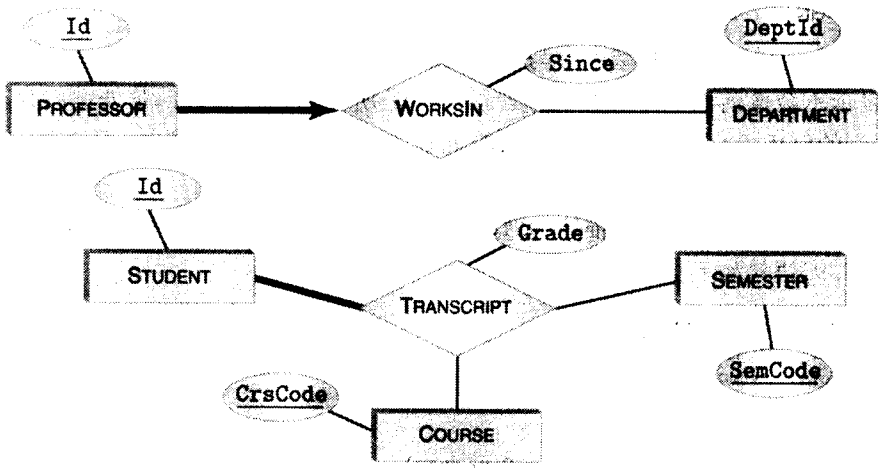


图5-7 参与约束

关系模型中的表示

从概念上说，使用关系模型表示参与约束是十分容易的。在图5-4中我们已经看到针对WORKSIN联系的CREATE TABLE语句。现在为实现WORKSIN中PROFESSORS的参与约束所要做的是指定如下的包含依赖[⊖]：

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

恰好，上面的约束也正好是外键约束，因为ProfId是WORKSIN的键。这样，我们只需简单地把PROFESSOR的Id属性声明为外键约束就实现上面的参与约束：

```
CREATE TABLE  PROFESSOR  (
    Id        INTEGER,
    Name      CHAR(20),
    DeptId    CHAR(4),
    PRIMARY KEY (Id),
    FOREIGN KEY (Id) REFERENCES WORKSIN (ProfId) )
```

注意，上面的外键约束没有排除Id是空值的可能性，而通常来说，应当要保证Id非空的约束。但是，本例中Id是PROFESSOR的主键，所以就没有必要再指定NOT NULL约束（至少，在SQL-92兼容的数据库中可以这样做）。

我们还可以使转换的效果更好一些。注意，Id是PROFESSOR的键，同时也是WORKSIN的键（间接的，通过外键约束来实现），所以可以把WORKSIN中的属性合并到PROFESSOR中，在删除ProfId属性之后。这是可能的，因为这两个表的共同键保证每个PROFESSOR元组有完全对应的WORKSIN元组，并且连接这些元组也不会导致冗余。产生的新表PROFESSORMERGEDWITH-WORKSIN如下：

⊖ 包含依赖参见4.2.2节的介绍。

```
CREATE TABLE  PROFESSORMERGEDWITHWORKSIN  (
    Id        INTEGER,
    Name      CHAR(20),
    DeptId    CHAR(4),
    Since     DATE,
    PRIMARY KEY (Id)
    FOREIGN KEY DeptId REFERENCES DEPARTMENT )
```

尽管从概念上说，关系模型中参与约束的表示就和指定包含依赖差不多，但是SQL-92中实际的表示不会像前面例子那样简单。因为不是所有的包含依赖都是外键约束（见4.2.2节），而它们的SQL-92表达式要用到复杂的机制（比如断言和触发器），而这些对性能都有负面影响。

该情形的一个例子是图5-7中描述的STUDENT在联系TRANSCRIPT中的参与约束。TRANSCRIPT转换成的SQL-92语句如下:

```
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))
```

同前面一样，为TRANSCRIPT表指定的外键约束不能保证每个学生都会选修一门课程。为保证每个学生都参与某个TRANSCRIPT联系，STUDENT关系必须有如下形式的包含依赖:

```
STUDENT(Id) references TRANSCRIPT(StudId)
```

然而由于StudId不是TRANSCRIPT的候选键，所以该包含依赖不能由外键约束表示。在第4章中，我们曾解释过如何用CREATE ASSERTION语句来说明包含依赖（参阅第4章的程序(4.4)）。

但是，验证普通的断言通常比验证外键约束的代价高很多，因此考虑到可能的开销，务必谨慎使用（4.4）那样的约束。例如，如果可以确定包括这样的断言减慢了数据库的关键操作，设计者可以把包含依赖的检查作为一个单独的、周期性而不是实时执行的事务的一部分。

5.5　一个经纪公司的例子

我们已经使用学生注册系统解释了E-R模型中的绝大多数的概念。在本节中，我们再用一个例子来解释这些概念。

PSSC是一个经纪公司，它为客户买卖股票。因此公司的主要参与者是经纪人和客户。PSSC在不同的城市拥有办事处，每个经纪人都在其中的某一个办事处工作。其中的某个经纪人还可以是他所工作的办事处的经理。

客户拥有账户，每个账户都可以有多个拥有者。每个账户也由某个经纪人管理。客户可以有多个账户，经纪人也能管理多个账户，但客户在某个确定的办事处只能有一个账户。

现在的需求是设计一个数据库维护上述的信息，以及在每个账户上执行的交易信息。

图5-8描述有关经纪人和客户的基本信息，更详细的信息请参见图5-9。这里我们还假设经

纪人只能最多管理一个办事处，而每个办事处也至多只有一个经理。注意，这里我们没有为MANAGEDBY联系中的OFFICE指定参与约束，因此某个办事处可能没有经理（比如，如果某个经理辞职，该位置则空缺）。因为每个账户只被某个办事处的某个经纪人所维护，所以图5-9显示一个参与约束，它对应联系ISHANDLEDBY中的ACCOUNT实体，箭头方向从ACCOUNT指向ISHANDLEDBY。这样，{Account}是ISHANDLEDBY的键。注意，图中形成实体键的属性用下划线表示，且不同的键分别有不同的下划线，例如OFFICE有两个键：{Phone#}和{Address}。

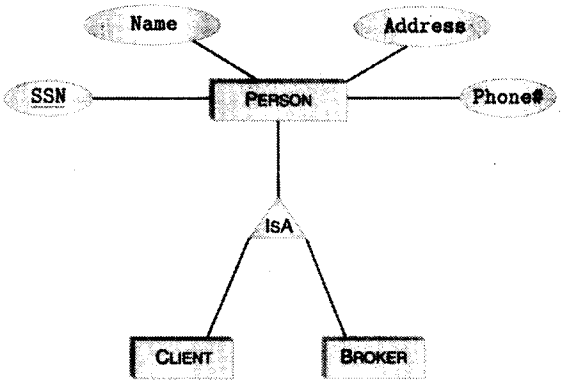


图5-8 PSSC企业的IsA层次结构

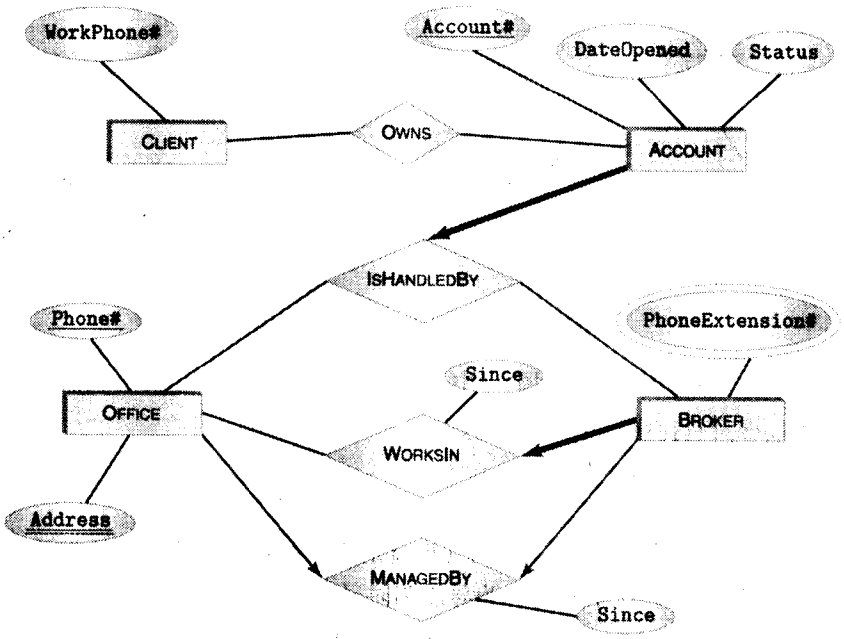


图5-9 客户/经纪人信息：首次尝试

但是，若仔细检查图5-9会发现存在某些问题，比如，它要求每个账户都要有一个经纪人，这可能是正确的也可能是不正确的，它依赖于公司的制度。又如，要求客户在同一个办事处不能有不同的账户这个需求没有在图中表示出来。

图5-10纠正了这些问题。这里我们用稍微不同的方法，引入三元联系HASACCOUNT，其键是{Client，Office}。而且，还包含一个联系HANDLEDBY，它关联账户和经纪人，但不要求每个账户都要有经纪人。

但是，该图还有问题。首先注意，连接ACCOUNT和HASACCOUNT的边没有箭头，这样的箭头将使得角色{Account}成为联系HASACCOUNT的键，但这会和一个账户可以有多个拥有者的

要求相冲突。然而，这个新的设计中还包含其他的问题：每个账户正好分配给一个办事处这个约束在图中没有表示出来。HASACCOUNT中的ACCOUNT的参与约束表示每个账户必须至少分配给一个办事处（和至少一个客户），但这没有说明办事处必须唯一。而且，我们也不能为该参与约束加上一个箭头来解决问题，因为前面已经说过，这会无法实现让同一个账户有多个所有者的要求。

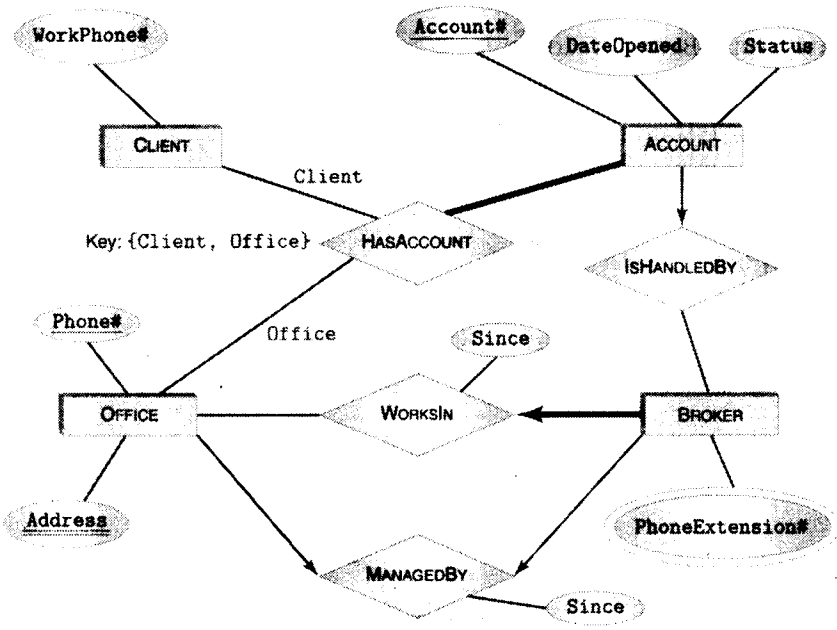


图5-10　客户/经纪人信息：第二次的修改

图5-10还有一个问题（该问题在图5-9中也存在）。假设我们有下面的联系：

⟨Client1, Acct1, Office1⟩ ∈ HASACCOUNT

⟨Acct1, Broker1⟩ ∈ HANDLEDBY

⟨Broker1, Office2⟩ ∈ WORKSIN

有什么能够保证Office1和Office2是一样的（也就是说，Account1的办事处就是掌管Account1的经纪人所在的办事处）？这个问题称为**导航陷阱**（navigation trap）：从给定的实体Office1开始，沿着由三个联系HASACCOUNT、HANDLEDBY和WORKSIN形成的三角形移动，我们可能会最终到达同一类型的另一个实体如Office2。在E-R模型中，这样的导航陷阱是很难避免的，因为若要这么做必须用到参与约束和所谓的*函数依赖*（functional dependency，见8.3节），但E-R图对它们只提供很有限的支持，即提供键和参与约束[⊖]。

注意，要避免导航陷阱，可以通过删除联系HASACCOUNT并引入客户和账户之间的OWNS

⊖ 为消除该例中的导航陷阱，我们要用到相当复杂的包含依赖。通俗地说，要保证每个元组t∈HASACCOUNT，其属性Account要引用由某个经纪人操作的账户，而属性Office要引用该经纪人的办事处。在学完第6章的基本关系代数之后，读者会发现包含依赖可规范地表示成$\pi_{Office,Account}$ (HANDLEDBY ⋈ WORKSIN) ⊇ $\pi_{Office,Account}$ (HASACCOUNT)。其中π是投影操作符（它去掉Office和Account之外的所有属性），⋈是连接操作符（它将对应于HANDLEDBY和WORKSIN关系中同一个代理的元组排列起来）。

联系来实现。然而，这又会回到前面的问题，即客户不能在某个办事处有多个账户的约束没办法表示出来。

即使我们还没有为该数据库设计出完全令人满意的方案，但现在不妨把注意力转到处理股票交易的部分上。图5-10和图5-11可通过实体ACCOUNT连接为一个更大的图。

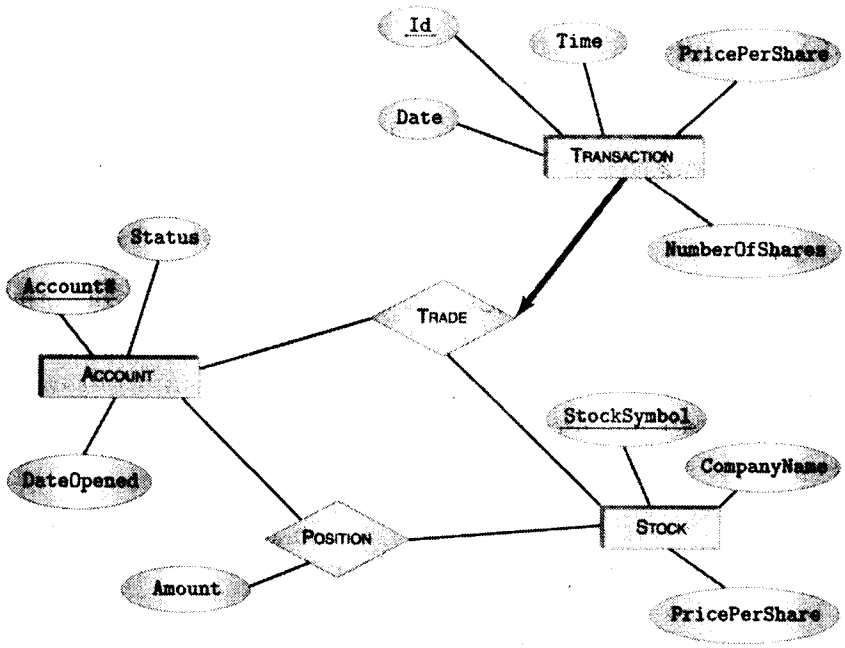


图5-11 PSSC公司的交易信息

交易信息用到三个实体：ACCOUNT、STOCK和TRANSACTION。这些实体通过联系POSITION和联系TRADE链接在一起，POSITION关联股票和持有它们的账户，TRADE表示实际购买和卖出的股票。这些信息在图5-11中描述。

注意，图中的角色Transaction是联系TRADE的键，且该事务不能在TRADE联系之外存在。该约束在图5-11中由粗箭头表示，它确定实体TRANSACTION和联系TRADE之间的一对一关系。

5.6 E-R方法的局限性

到目前为止，读者已经看到用实体-联系模型设计数据库的两个案例研究。如果你对E-R设计还没有完全理解，那么你有了一个很好的学习伙伴。尽管我们已经介绍的概念可以为组织企业数据提供一般性的指导，但若要在实际问题中应用它们仍然需要丰富经验和技巧。决定某个数据是否是实体、联系或属性具有很大的灵活性。而且，即使这些问题都解决，实体间联系的表现形式还是多种多样的。本节讨论数据库设计者经常会面对的困难问题。

1. 实体还是属性

在图5-7中，学期被表示为实体。然而，我们能把TRANSCRIPT变成二元（而不是三元）联系，并让SEMESTER成为它的一个属性。很明显，现在的问题是哪种表示办法更好。

在某种程度上，决定某个数据表示成实体还是属性通常取决于个人的习惯。除此以外，如何表示可能还取决于该数据是否有自己的内部结构。如果它没有自己的内部数据结构，把它作为独立的实体会使E-R图更加复杂，而且更糟的是，当把E-R图转化为关系模型时，还会增加额外的关系到数据库模式中。另一方面，如果数据有它自己的属性，若把该数据表示成属性，则数据的属性就无法表示出来。

例如在图5-7中，实体类型SEMESTER没有它自己的属性，所以把学期信息表示成实体显得没有什么必要。然而，如果需求文档中指出每个学期中必须要有以下信息：Start_date（开始日期），End_date（结束日期），Holidays（假期）和Enrollment（注册人数），那么学期信息就不能再是TRANSCRIPT联系的属性，因为没有指定关键的日期和学期相关的注册人数（比如，某个学期中大学所有的注册人数不是任何一个成绩单联系的属性）这些信息。

2. 实体还是联系

再考虑图5-7中的E-R图，图中我们把成绩单记录看作是STUDENT，COURSE和SEMESTER实体之间的联系。还有一种设计方案是把成绩单记录作为实体，并引入一个新的联系类型ENROLLED来连接它们，如图5-12所示。这里我们还合并实体SEMESTER的一些属性，此外还为联系ENROLLED增加属性Credits（学分）[⊖]。显然，这两张E-R图表达的信息是一样的（除图5-12中增加些属性外），那么哪个更好呢？

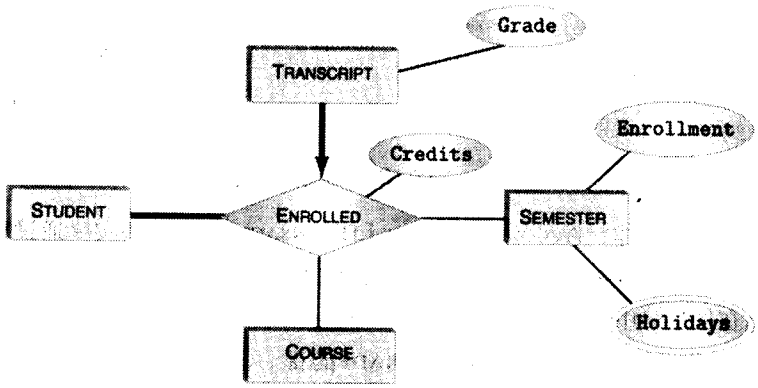


图5-12　成绩单信息的另一种表示方法

当面临“实体还是属性”这样的两难选择时，个人喜好是主要决定因素。但是还要考虑其他一些方面。例如，最好尽量减少实体和关系的数目，因为它们直接影响E-R图转换为关系模型时所生成的关系的数目。通常在这个阶段两个关系堆成一团也无大碍，因为利用第8章中的关系设计理论可确定关系是否需要分解，并且有相应的分解算法可供利用。另一方面，发现相反的问题是比较困难的，即发现不需要分解的关系比较困难。

再回到图5-12，我们注意到，实体TRANSCRIPT和联系ENROLLED之间存在一个参与约束。此外，箭头从TRANSCRIPT指向ENROLLED表示角色TRANSCRIPT构成ENROLLED联系的键，因此实体类型TRANSCRIPT和联系类型ENROLLED之间存在一对一的对应关系。这意味着可认为ENROLLED类型的联系是多余的，因为可以使用TRANSCRIPT实体来关联STUDENT、COURSE和

⊖ 例如，在Stony Brook大学，一些研究性课程需要累积一定的学分才可选修。

SEMESTER实体类型。为不丢失信息，所需做的是在把TRANSCRIPT转变为联系后，将ENROLLED的属性转移到TRANSCRIPT。

总结上面的讨论，我们得到下面的规则：

> 考虑实体类型$E_1,\cdots,E_n$和关联它们的联系类型**R**，假定E_1通过一个角色附属于**R**，该角色构成**R**的键，且E_1和**R**之间存在参与约束，那么可把E_1和**R**合并到一个只涉及到实体类型$E_2,\cdots,E_n$的新的联系类型上。

注意，该规则只是表明E_1可以合并到**R**，而不保证这是可行的。例如，如果E_1还牵涉到其他的联系**R'**，这样，如果合并E_1和**R**，则**R**和**R'**之间会存在一条边，但E-R图的构造规则不允许两个联系类型之间存在边。图5-10中的BROKER和ACCOUNT实体就有这样的情形，按照规则BROKER可以合并到WORKSIN，ACCOUNT可以合并到HANDLEDBY。但是，由于BROKER和ACCOUNT都涉及到两个不同的联系，因此合并后图中WORKSIN和HANDLEDBY，HASACCOUNT和HANDLEDBY这两对联系之间都会有一条边。另外，图5-11中的实体类型TRANSACTION可以合并到联系类型TRADE。

3. 信息丢失

我们已经看到，把实体降为属性或者合并实体到联系中都会使联系的度发生变化。但在上面给出的例子中，即使经过转化，E-R图的信息内容仍然保持不变。现在我们讨论**信息丢失**（information loss）的典型情况，即表面上看来无害的转换实际上会导致E-R图信息内容的改变，即信息丢失。

考虑图5-3的PARTS/SUPPLIER/PROJECT图，某些设计者可能不喜欢使用三元联系，而喜欢用多个二元联系来取代它，这种改变会形成图5-13所示的E-R图。

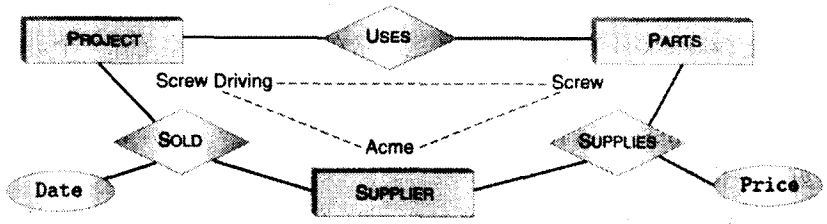


图5-13 用三个二元联系取代图5-3中的三元联系SOLD

尽管从表面上看，该图和原图差不多，但实际上有几个细微的差别。首先，新设计中引入了我们在股票交易例子中见过的导航陷阱，即有可能供应商Acme出售Screw（螺丝钉），另外它还供应商品给项目Screw Driving，而且Screw Driving项目可能正使用Acme所出售的螺丝钉。但是从该E-R图的联系中没法得出是Acme卖这些螺丝钉给这个项目的结论，我们所知道的只是Acme可能这么做。

图中的另一个问题是价格属性现在和联系SUPPLIES有关。这意味着每件产品的价格都固定，与该产品供应给哪个项目无关。与此相反，图5-3的原设计中支持不同的项目可获得不同的价格（例如，基于采购数量）。类似的，新的设计只允许供应商和项目之间在一天中只进行一次交易（因为这里每个交易由元组<p,s;d>表示），因此没有办法区分同一天中他们所做的多次交易。然而原设计中，若涉及不同的零件，那么一天中是可以做多次交易的。

认识到引入导航陷阱的危害后，我们可尽可能地使用度数较高的联系。例如，在图5-10中，为消除由联系HASACCOUNT、WORKSIN和HANDLEDBY引起的导航陷阱，我们将这些联系合并为一个联系。然而，这样的解决方法会引发更多的问题。例如，若该转化保留连接BROKER和WORKSIN的箭头，则我们于无意中引入“一个经纪人至多只能有一个账户和一个客户”这个约束。若不保持这个箭头则又丢失约束“每个经纪人被分配到一个办事处”。这个转化也没有办法实现经纪人没有账户和账户没有经纪人。

4. E-R和对象数据库

在第16章之前我们都不会讨论对象数据库，但这里我们要概述一下，一些从E-R图转化到模式的很难的问题在对象数据库中则会变得比较容易。

- 在5.2节，我们曾讨论过关系数据库中带有值集合属性的实体表示问题。在对象数据库中存储的对象可有值集合属性，因此在对象数据库中表示这样的实体会相当容易。
- 在5.4节，我们曾讨论过关系数据库中IsA联系的表示问题。对象数据库在模式中直接表示IsA联系，因此，这样的联系其表示也容易多。

从这些例子中我们可以看出，不仅把E-R图翻译到对象数据库的模式比翻译到关系数据库的模式通常要容易些，而且对许多应用而言用对象数据库建模要比用关系数据库建模直观得多。

5.7 案例研究：学生注册系统的设计

本节将介绍学生注册系统的数据库设计。该设计包含在完整的应用程序设计文档中，该文档在12.1节中描述。

设计过程的第一步是构造如图5-14所示的E-R图，该图是对3.2节的需求文档中学生注册系统的模型。从图中我们注意到：

- STUDENT通过TRANSCRIPT关联到CLASS，这表示在某个学期学生是在一个班级中注册、登记或完成的。
- FACULTY通过TEACHES关联到CLASS，这表示某个学期中一个教师教授一门课，而每个班级也正好由一位教师授课。
- COURSE通过联系REQUIRES关联到COURSE，即一门课程可能是某个学期中的另一门课程的预备课程。当预备课程有效时，属性EnforcedSince指定了课程的日期。
- CLASS通过TAUGHTIN关联到CLASSROOM，也就是说某学期中一个班要在某个教室上课。
- 一个班（即提供某课程）最多用一本教材，因为Textbook属性不是实体CLASS的键的一部分。所以在8.12节，我们将删除这个限制并讨论它对系统设计决策的影响。

以这个E-R图和3.2.4节需求文档中列出的完整性约束为基础，下一步将产生模式，如图5-15和图5-16所示。E-R图到关系模式的转化使用了本章所讨论的技术，其方式比较直接。注意，没有为联系TEACHES和TAUGHTIN创建表，因为它们都只有角色而没有属性，且都是多对一联系，因此它们的信息可以存储在表CLASS中——在CLASS表中与每个班级对应的元组包括授课教室的标识ClassroomId和授课老师的标识InstructorId。

注意，需求文档中规定的一致性约束在CREATE TABLE语句中检查（特别是来自3.2.4节的约束1、3、5和6）。在完整的模式设计中，其他的完整性约束用CREATE ASSERTION语句

和触发器检查。然而这部分设计会用到我们还没有讨论过的SQL构造，在12.6节学完这些构造后会继续完成这里的模式设计。

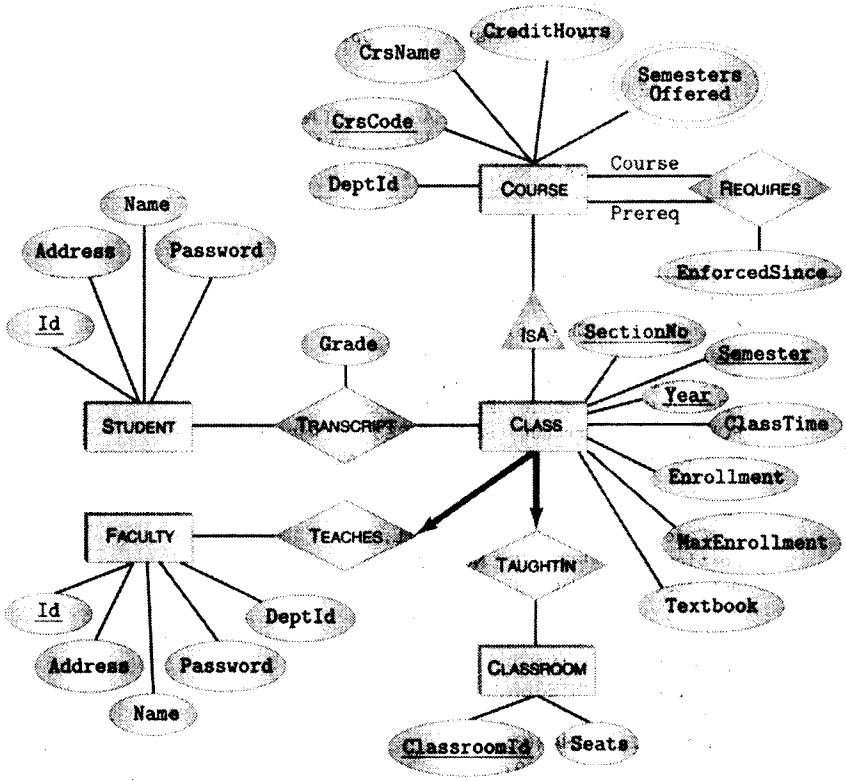


图5-14 学生注册系统的E-R图

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

CREATE TABLE FACULTY (
    Id              CHAR(9) NOT NULL,
    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) NOT NULL,
    DeptId          CHAR(4) NOT NULL,
    CrsName         CHAR(20) NOT NULL,
    CreditHours     INTEGER NOT NULL,
    PRIMARY KEY (CrsCode),
    UNIQUE (DeptId, CrsName) )
```

图5-15 学生注册系统的模式——第1部分

```
CREATE TABLE WHENOFFERED (
    CrsCode        CHAR(6) NOT NULL,
    Semester       CHAR(6) NOT NULL,
    PRIMARY KEY (CrsCode, Semester),
    CHECK (Semester IN ('Spring','Fall') ) )

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

图5-15 （续）

```
CREATE TABLE REQUIRES (
    CrsCode        CHAR(6)  NOT NULL,
    PrereqCrsCode  CHAR(6)  NOT NULL,
    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)  NOT NULL,
    SectionNo      INTEGER  NOT NULL,
    Semester       CHAR(6)  NOT NULL,
    Year           INTEGER  NOT NULL,
    Textbook       CHAR(50),
    ClassTime      CHAR(5),
    Enrollment     INTEGER,
    MaxEnrollment  INTEGER,
    ClassroomId    CHAR(3),
    InstructorId   CHAR(9),
    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)  NOT NULL,
    CrsCode        CHAR(6)  NOT NULL,
    SectionNo      INTEGER  NOT NULL,
    Semester       CHAR(6)  NOT NULL,
    Year           INTEGER  NOT NULL,
    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') ) )
```

图5-16 学生注册系统的模式——第2部分

我们在本书中选择这个设计例子是因为它是直观和清晰的。在实践中，它可能只是一个起点，需要进一步扩展来反映更多的特性，同时提高最终实现的效率。

让我们讨论一些可能需要加强和可被替换的地方。考虑课程依赖。我们设计的模式中一个明显的遗漏是“并修课程”（corequisite）这个联系，所有的约束都需要它。更微妙地是，大学课程时刻在变：新课程会加入，老的课程会被取消，而课程间的基本依赖也会变化。这样，图5-14中的REQUIRES联系可能需要另外两个属性Start和End来指定预备课程联系在哪段时间内有效。而且该预备课程联系可能会在不同时期存在。例如，1985年和1990年间（修完）课程A是（修）课程B的先决条件，且它在1999年和现在之间可能又成为预备课程。这种情况下的建模留到练习5.10。

另一个有趣的增强条件是，某些很受欢迎的课程可能只限于某些专业。在这种情况下，E-R图和对应的表中不得不包含学生专业的信息（这是值集合属性！），且还要和课程所允许的专业一致。该问题留到练习5.11讨论。

除加强我们的设计来表达更复杂的需求外，还可以利用各种替代方案，但它可能会影响系统的总体性能。下面我们讨论一个替代方案的例子和它的影响。

考虑实体COURSE的属性SemsetersOffered。因为它是值集合属性，我们把它翻译到独立的表WHENOFFERED。我们选择这个例子是因为它可以容易地表达约束“开设某个班的学期必须是提供该课程的学期”。例如，只在春季学期提供课程CS305，但却在2000年的秋季设置开设该课程的班级，这是不允许的。当然，如果春季和秋季两个学期都提供CS305，那就没有影响。

在我们的设计中，该需求可简单地用关系CLASS中的一个外键约束表达：

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

该约束说明，如果所开课程的课程代码是CrsCode的班级在学期Semester中提供，那么<CrsCode, Semester>应该是WHENOFFERED中的元组。也就是说，Semester必须是允许开设课程的学期。

抛开简化设计不谈，有人可能觉得为这么少的信息创建单独的表会导致不必要的开销。的确，这个单独的表有更多的存储需求，且对于查询需要额外的操作，等等[⊖]。一个替代方案是定义一个新的SQL域（参阅第4章），该域只包含三个可能的值：

```
CREATE DOMAIN  SEMESTERS CHAR(6)
    CHECK ( VALUE IN ('Spring', 'Fall', 'Both') )
```

现在，实体COURSE的值集合属性SemestersOffered就是单值的，但它取值范围在域SEMESTERS中。这样做的好处是到关系模型的转化更加直接，也不需要额外的关系WHENOFFERED。然而，若要说明约束“在已开设相应的课程时，一个学期只能开设这门课程的一个班”则更难。这个问题的细节留到练习5.12。

最后，我们考虑表示当前和下个学期的替代方案（这个需求由3.2节需求文档所提供的某些事务规定）。实际上，在我们的设计中没有明显的途径来表示这个信息。说明哪个学期是当前学期或下一个学期，一个简单的办法创建一个独立的表来存储这些信息。然而，这不是一

⊖ 我们的例子中，这些缺点好像都没有出现，这是由于通常情况下，关系WHENOFFERED都被用于验证上面的外键约束，而为“课程-学期”生成单独的关系使得检查该约束效率很高。

个好的设计，因为每个到当前或下一个学期的引用需要昂贵的数据库查询代价。解决这种问题的正确办法是使用SQL提供的CURRENT_DATE函数，并用EXTRACT函数从日期中提取某个字段，例如:

```
EXTRACT(YEAR FROM CURRENT_DATE)
EXTRACT(MONTHS FROM CURRENT_DATE)
```

上面语句返回当前年和月的数值。这样就足以确定给定的学期是当前的学期还是下一个学期。

5.8 参考书目

实体-联系方法在[Chen 1976]中提出。它自诞生之日起就得到广泛的关注，并且人们提出多种扩展（如研究论文集[Spaccapietra 1987]）。使用E-R模型的概念设计已经取得显著的进步。读者可参考[Teorey 1992, Batini et al. 1992, Thalheim 1992]来获得全面的介绍。

最近，出现了一种称为统一建模语言（Unified Modeling Language，UML）的新的设计方法学，并且广为流行[Booch et al. 1999]。UML借鉴E-R模型中的许多思想，并在面向对象建模方面进行了扩展。特别是，它不仅提供数据结构建模的方法，还提供程序行为方面的建模方法，以及在复杂的计算环境中如何配置复杂的应用。UML的大致介绍可以参考[Fowler and Scott 1999]。

许多现有的工具可以帮助数据库设计者完成E-R建模和UML建模。这些工具帮助用户完成指定图表，属性，约束等工作。当这些工作都完成后，它们可把概念模型自动地映射为关系表。这些工具包括Computer Associates公司的ERwin，Embarcadero Technologies公司的ER/Studio以及Rational Software公司的Rational Rose。此外，数据库管理系统供应商也提供它们自己的设计工具，如Oracle公司的Oracle Designer和Sybase公司的Power Designer。

5.9 练习

5.1 假设你要通过加入一个新属性（比如5.4.1节描述的STUDENT实体的Status属性）把IsA层次结构转换成对象模型，若子实体是不相交的则会存在什么问题（例如，如果秘书也可以是技术人员）？如果没有覆盖约束会存在什么问题（例如，如果一些雇员既不是秘书也不是技术员）？

5.2 构造一个E-R图的例子，该图直接转化成关系模型时也会像PERSON实体一样有一个异常（参见有关图5-1和5-2的讨论）。

5.3 在关系模型中表示图5-5的IsA层次结构。对每个IsA联系讨论你使用的技术（也就是说，使用“将层次结构中每个实体表示成一个单独的关系”这种技术，还是使用“实体和它的子实体都在一个关系中但用新的属性区分”这种技术），并讨论什么情况下用另外一种技术会更好。

5.4 将5.5节的经纪人这个例子翻译成SQL-92模式。用必要的SQL-92手段表达E-R模型中所有的约束。

5.5 说明图5-9中出现的导航陷阱。

5.6 考虑下面的数据库模式:

- SUPPLIER (SName, ItemName, Price)——供应商SName以价格Price出售商品ItemName。
- CUSTOMER (CName, Address)——顾客CName住在Address。
- ORDER (CName, SName, ItemName, Qty)——顾客CName从供应商SName处订购Qty件产品ItemName。
- ITEM (ItemName, Description)——产品的信息。

a. 根据上面模式画出相应的E-R图，并指定键。

b. 假设现在你要为该图增加一个约束：每种产品由某个供应商提供。修改E-R图使之包含该约束，并说明如何将新图翻译回关系模式。

5.7 用E-R方法为你所在社区的图书馆建模。图书馆中有书、CD、磁带等，它们可借给读者。读者有账户、地址等信息。如果借出的物品未按时归还，则累积罚款。然而有些读者是未成年人，因此必须有资助者负责支付罚款（或者赔偿丢失的书）。

5.8 为房产中介公司设计E-R图。该公司登记待售的房子的信息，并有想买房子的顾客的信息。待售的房子可由该公司或别的公司“列出”，房子被“列出”意味着该房子的主人和中介公司的代理人签署合同。市场上的每所房子都有地址、价格、主人以及一些特征列表（比如卧室和浴室的个数、供热方式、家用电器、车库大小等等）。这个列表对于不同的房子可以是不同的，并且房子的属性也随房子的不同而不同。同样地，每个顾客都有他的偏好，表达的形式一样（如卧室、浴室的数量等）。除此之外，顾客还可以指明他们感兴趣房子的价格范围。

5.9 一个连锁超市决定建立一个决策支持系统，以便分析不同时间在不同超市出售的各种商品情况。每个超市位于一个城市，城市处于某个州，而州又处于某个区域里。时间可由天，月，季度和年来度量。商品有名字和类别（如农产品，罐装食品等）。为该应用设计E-R图。

5.10 为图5-14所示的学生注册系统修改E-R图，加入并修课程和存在于多个时期的预备课程联系。每个时期由某年的某个学期开始，也结束于某年的某个学期；或者每个时期是持续到现在的。并转换为适当的关系模型。

5.11 为图5-14所示的学生注册系统修改E-R图，加入学生专业和某课程允许哪些专业的学生选修这些信息。一个学生可以有多个专业（大学中有些专业会这样做，比如CSE、ISE、MUS、ECO）。课程有一个可允许的专业列表或者该列表为空，列表为空说明任何人都可以选修该门课。用约束表达是“限制专业的课程只能由那些符合要求的专业的学生选修”。

一般地说，上面这个约束只能用SQL的断言语句（参见4.3节）来表示，且用到6.2节讨论的一些特性。然而，基于下面的简化假设还是可以表达这个约束的：当一个学生注册一门课程时，他必须声明自己的专业以便可以注册该课程。

修改模式以反映这个简化过的假设，并表达前述的完整性约束。

5.12 对学生注册系统的模式作必要的修改以反映使用SQL域SEMESTERS的设计（见5.7节最后）。表达这个约束，某班级只能在提供相应课程的学期里开设。例如，如果课程CS305的属性SemestersOffered值为Both，那么对应的班级可在春秋两个学期都开设。但是，如果该属性的值是Spring，那么班级只能在春季开课。

5.13 为下面企业设计E-R模型。各种企业组织之间互有业务往来（为简单起见，我们不妨假设任意一笔业务只有两方参与）。当洽谈业务（签协议）时，每个企业都由一个律师作代表。一个企业可以和多个不同的企业有业务往来，并指派不同的律师负责各笔业务。律师和企业有多个属性，如地址和名字。他们也有自己独有的属性，例如，律师有专长和收费情况的属性，企业有预算的属性。

说明当维大于2的联系被分解成二元联系时，信息丢失是如何发生的。讨论如何给出一个假设，在该假设下进行分割不会导致信息丢失。

第6章 查询语言 I：关系代数和SQL

既然知道了如何创建一个数据库，那么下一步就要学习如何查询这个数据库来为某个特定的应用程序检索需要的信息。*数据库查询语言*是一种用于特定目的的编程语言，它用来检索存储在数据库中的信息。

在关系数据库出现以前，数据库查询是一件非常麻烦的任务。即使是提交一个简单的查询（按照现在的标准），也必须使用传统的编程语言来编写一段程序，这段程序可能包含多个嵌套循环、错误处理和边界条件检查。另外，程序员还要处理繁杂的内部物理模式的细节问题，因为在当时数据独立性还只是个愿望而已。

我们最感兴趣的关系查询语言是SQL（结构化查询语言，Structured Query Language）。它和传统的编程语言有着很大的不同。在SQL中，你指定的是被检索的信息的性质，而不是检索所需要的具体算法。例如，在学生注册系统中的某个查询是要检索在某个学期里讲授某门课程的所有教授的姓名和Id，但是它并没有给出遍历多个数据库表来检索指定姓名的具体过程（包括while循环、if语句、指针变量等）。因此，SQL被称为是*声明性*（declarative）的，因为它的查询“声明”答案应该包含什么信息，而不是如何得到这些信息。而传统的编程语言（比如C或Java）因为在编写的程序中描述了为得到答案而要执行的严密步骤，所以称为*过程性*（procedural）的。

程序员就像使用其他的计算机语言一样，在简单了解SQL之后就能够设计出简单的查询，而要设计出实际应用程序中需要的复杂查询则需要对该语言及其语义有更深入的了解。因此，在介绍SQL之前，我们首先学习*关系代数*。关系代数是另外一种关系查询语言，它被数据库管理系统用作中间语言，SQL语句在被优化前先被翻译成这种形式。在第7章中将介绍其他的关系语言（特别是关系演算），并且将它们与SQL和关系代数关联起来。

6.1 关系代数：在SQL的覆盖之下

关系代数（relational algebra）被称为代数，是因为它基于一组为数不多的以关系（表）为运算对象的*运算符*。每个运算符对一个或多个关系进行运算，产生的结果是另外一个关系。查询就是包含这些运算符的表达式。表达式的结果是关系，也就是这个查询的答案。因此，在前面提交的学生注册系统的查询的结果是一个具有两个属性的关系，分别给出在指定学期里讲授指定课程的教授的姓名和Id。

尽管SQL是声明性语言，这意味着它不必指定用来处理查询的算法，但是关系代数却是过程性的。关系表达式可以看作是这种的算法的详细描述（尽管它比使用传统编程语言指定的算法的抽象等级要高得多）。

因此，即使程序员用SQL来指定查询，数据库管理系统也使用关系代数作为一种指定查询计算算法的中间语言。数据库管理系统解析SQL查询，并把它翻译成关系代数表达式，这

通常产生一个相当简单但很低效的算法。然后，**查询优化器**（query optimizer）把这个代数表达式转换成一个与之等价的代数表达式，但是（希望）可以使用更少的时间来执行[⊖]。查询优化器根据优化后的代数表达式，生成**查询执行计划**（query execution plan），然后在数据库管理系统中由代码生成器将它转换成可执行代码。因为代数表达式有着精确的数学语义，所以系统能够验证优化后的结果表达式是和原先的表达式是等价的。利用语义就能够比较几种不同的查询执行计划。查询处理示意图如图6-1所示。

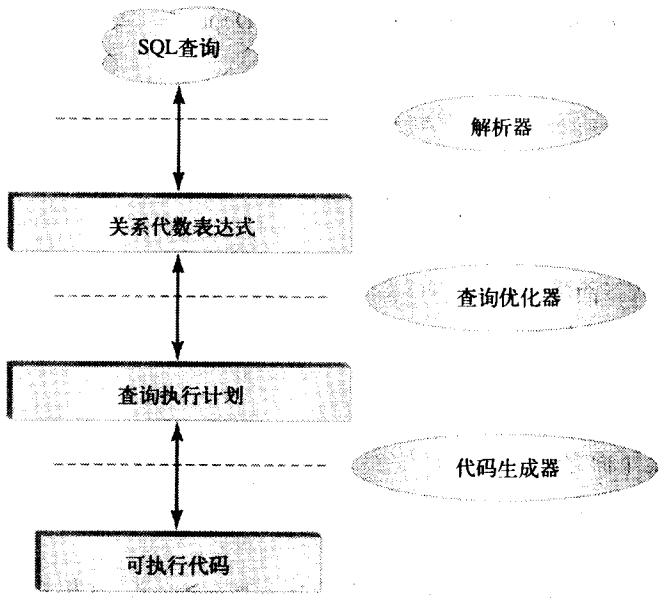


图6-1 查询处理示意图

关系代数是理解关系数据库管理系统内部运转机制的关键，而理解关系数据库管理系统的内部运行机制在设计可以高效处理的SQL查询时是非常重要的。

6.1.1 基本运算符

关系代数基于五个基本运算符：

- 选择（select）
- 投影（project）
- 并（union）
- 差（set difference）
- 笛卡儿积（cartesian product，也称为叉积（cross product））

我们将依次介绍这五个基本运算符。此外，还有三个导出运算符（即它们可以被表示为包含基本运算符的表达式）：**交**（intersection）、**除法**（division）和**联结**（join）。我们还讨论**重命名**（renaming）运算符，它在与笛卡儿积和联结结合使用时很有用。

⊖ 查询优化器并不真正做“优化”工作（即产生最有效的查询计算算法），因为这通常是不可能做到的。相反，它们使用已知的启发式方法来产生等价的表达式，这样的表达式的计算代价通常比较低廉。

1. 选择运算符

在关系上执行的最频繁的运算之一是对元组的子集的选择（也就是在一个表中对行的某个子集的选择）。例如，你可能想知道哪些人是计算机科学系的教授。当然，他们都列在PROFESSOR关系中，但是这个关系可能很大，对感兴趣的元组进行人工扫描是很困难的。使用选择运算符，这个查询可以表示为:

$$\sigma_{\text{DeptId = 'CS'}}(\text{PROFESSOR})$$

这个查询读作“从关系PROFESSOR中选择所有满足条件DeptId = ‘CS’的元组”。

选择运算符的语法是

$$\sigma_{\text{选择条件}}(\text{关系名称})$$

后面我们将看到，选择运算符的参数可能比一个简单的识别某个关系的名称要更具一般性，它也可以是能计算到一个关系的表达式。

选择条件可能是下面几种形式之一:

- 简单选择条件（下面将作解释）
- 选择条件 AND 选择条件
- 选择条件 OR 选择条件
- NOT（选择条件）

简单选择条件可能是下面之一:

- 关系属性 oper 常数
- 关系属性 oper 关系属性

其中，oper可以是下面的几种比较运算符之一: =、≠、>、≥、<和≤。参与比较的每个属性必须是选择运算符的关系名称参数里的一个属性。

认识到必须遵守上面的语法规则是很重要的，就像在任何其他的编程语言中遵守语法规则那样。编写查询来列出计算机科学系教授讲授的所有课程，一个常见的语法错误是:

$$\sigma_{\text{ProfId = PROFESSOR.Id AND PROFESSOR.DeptId = 'CS'}}(\text{TEACHING})$$

这个表达式可能看起来很自然，但是它在语法上是错误的，因为选择条件用到了PROFESSOR关系的属性，而在这里语法规则要求只能使用TEACHING关系（在选择的参数中指定的关系）的属性。

孤零的表达式$\sigma_{\text{选择条件}}(\mathbf{R})$是没有什么意义的，它只是一串字符。然而，在具体的数据库中，它可能有一个值，也就是这个表达式在那个数据库的上下文中的意义。因此，我们总是假设我们工作在某个具体数据库的上下文中，把具体的关系实例与每个关系名称相联系。

假设**r**是与**R**相联系的关系实例，我们定义上面的表达式关于**r**的值，记为$\sigma_{\text{选择条件}}(\mathbf{r})$，是**r**中满足选择条件的所有元组的集合构成的关系。因此，在一个关系上运算的选择运算符的值是另外一个关系。

例如，$\sigma_{\text{Dept Id = 'CS'}}(\text{PROFESSOR})$关于图4-5的数据库的值是关系CSPROF。

CSPROF	Id	Name	DeptId
	101202303	Smyth, John	CS
	555666777	Doe, Mary	CS

必须清楚r中的元组满足选择条件的含义。例如，如果条件是$A > c$，这里A是属性，c是常数，当（且仅当）t中属性A的值比c大时，元组t满足条件[⊖]。当选择条件较复杂的时候（如$cond_1$ AND $cond_2$），条件满足是通过递归来定义的：当且仅当t满足$cond_1$和$cond_2$时，t才满足选择条件。条件$cond_1$ OR $cond_2$是类似的，只是t只需要满足子条件中的一个即可。类似地，当t违反$cond$的时候，t满足NOT（$cond$）。

例如，关系CSPROF中的元组2满足复杂条件Id > 111222333 AND NOT（DeptId = 'EE'），因为它满足下面两个条件。

- Id > 111222333，因为555666777 > 111222333。
- NOT（DeptId = 'EE'），因为'CS' ≠ 'EE'（也就是说元组违反条件DeptId = 'EE'）。

相反，那个关系中的元组1违反了上面的复杂条件，因为它违反了条件的第一项Id > 111222333（因为101202303 ≯ 111222333）。

这里有一个较复杂的选择：

$$\sigma_{\text{StudId} \neq 111111111 \text{ AND (Semester = 'S1991' OR Grade =< 'B')}}(\text{TRANSCRIPT})$$

这里字符串间的比较（Grade =< 'B'）假设按字典顺序进行比较。在图4-5的数据库的上下文中，这个表达式的值是如下的关系：

SUBTRANSCRIPT	StudId	CrsCode	Semester	Grade
	666666666	MGT123	F1994	A
	666666666	EE101	S1991	B
	123454321	CS315	S1997	A
	023456789	CS305	S1996	A

选择条件的一个明显的泛化是允许使用形如$expression_1$ oper $expression_2$的条件，这里$expression$可能是一个涉及属性（作为变量）和常数的算术表达式，或是字符串表达式（如模式匹配，字符串连接）。

EmplSalary >（MngrSalary * 2）和（DeptId + CrsNumber）LIKE CrsCode（这里“+”表示字符串连接，LIKE表示模式匹配）是这种表达式的例子。这里的扩展选择的功效是很明显的，广泛用在数据库语言中。

2. 投影运算符

在讨论选择的时候，我们经常引用一个元组中某些属性的值。在关系代数中，这样的引用是很常见的，所以我们为它们引入特别的符号。假设A表示关系r的一个属性，t表示r中的一个元组，则$t.A$表示元组t中与属性A相应的分量。例如，如果t表示关系SUBTRANSCRIPT中的元组1，那么t.CrsCode就是MGT123。

我们经常还需要从一个元组中提取出子元组。子元组是依照某个属性序列从t中析取出的一个值的序列。它可以表示成

$$t.\{A_1, \cdots, A_n\}$$

⊖ 我们假设存在某种次序。在数值域中，这种次序是很明显的；在字符串域中，我们假定这种次序为字典顺序。

这里$A_1, \cdots, A_n$是属性。

例如，如果t是SUBTRANSCRIPT中的元组1，那么t.{Semester, CrsCode}就是元组{F1994, MGT123}。注意，这里属性的顺序没有（也不需要）遵照它们在关系SUBTRANSCRIPT中列出的顺序。此外，有时允许在列表中出现重复的属性是很方便的。因而，t.{Semester, CrsCode, Semester}是<F1994, MGT123, F1994>。

现在，我们可以定义投影运算符了，其通常的语法是

$$\pi_{\text{属性列表}}(\text{关系名称})$$

举个例子，如果**R**是关系名称，$A_1, \cdots, A_n$是**R**中的一些（或全部）属性，那么$\pi_{A_1, \cdots, A_n}$（**R**）就称为**R**在属性$A_1, \cdots, A_n$上的投影。（换句话说，投影选取表中列的某个子集。）

和选择的情况一样，上面的表达式在具体数据库的上下文中能被赋予一个值。假设**r**是这样的数据库中与**R**相应的关系实例。那么$\pi_{A_1, \cdots, A_n}$（**R**）的值，记为$\pi_{A_1, \cdots, A_n}$（**r**），是形为t.{$A_1, \cdots, A_n$}的所有元组的集合，这里t的取值范围是**r**中所有的元组。

例如，$\pi_{CrsCode, Semester}$（TEACHING）的结果如右图所示。

OFFERINGS	CrsCode	Semester
	MGT123	F1994
	EE101	S1991
	CS305	F1995
	CS315	S1997
	MAT123	S1996
	EE101	F1995
	CS305	S1996
	MAT123	F1997
	MGT123	F1997

注意，图4-5的最初的TEACHING关系有11个元组，而OFFERINGS只有9个元组。其他的元组去哪了呢？如果我们更仔细地检查一下最初的关系，答案就一目了然了。容易看出，元组1和元组4的CrsCode和Semester属性是相同的。它们之间的唯一区别是ProfId属性的值不同。元组10和元组11也是同样的情况。应用投影运算符到元组1和元组4（到元组10和元组11）会产生相同的元组，因为属性ProfId被消除了。关系是集合，所有不允许有重复；因此在每组相同元组中只保留其中一个元组。

投影运算符的目的是帮助我们关注感兴趣的联系，而忽略与某个查询无关的属性。例如，假设我们想知道哪些课程是在哪个学期开设的，则OFFERINGS关系可以清楚地给出这些信息，没有把答案和我们没有请求的数据（即ProfId）混在一起。投影也消除了重复，这也为我们分析答案节省了时间。

既然我们已经学习了两个关系运算符，就可以把它们组合成**关系表达式**(relational expression)。这并不只是抽象的数学练习。在关系数据库中，表达式是构造查询的一般方法。例如，我们看到查找所有计算机科学系教授的查询，结果是关系CSPROF。我们可以用投影运算符只得到那些教授的姓名，表达式是π_{Name}（CSPROF），但是代数的优势是能组合运算符，所以我们可以写出

$$\pi_{Name}(\sigma_{DeptId = 'CS'}(\text{PROFESSOR}))$$

而无需指定中间结果和创建临时的关系。

3. 集合运算符

下面两个运算符是熟悉的集合运算符**并**（union）和**差**（set difference）。显然，由于关系是集合，所以这些运算符可以应用到关系上面。语法是**R** ∪ **S**和**R**−**S**。如果**r**和**s**是与**R**和**S**相应的关系，那么以下的性质成立：

• **R∪S的值是r∪s**，即属于r或s的所有元组的集合。

• **R−S的值是r−s**，即r中不属于s的所有元组的集合。

我们也可以使用**交**（intersection）运算符，**R∩S**（其在r和s上的值自然是r∩s），但是这个运算符并不独立于其他的运算符。它可以用本节开头提到的五个基本的运算符构造的表达式来表示。

但是，我们并没有做完。尽管两个集合的并总是一个集合，但是这不能推广到关系上去。考虑计算PROFESSOR和TRANSCRIPT的并。一个问题是关系是表，所有的行有同样数目的项。然而在PROFESSOR关系中，每个元组有三个项，而在TRANSCRIPT关系中每个元组有四项。因为它们具有不同的元数，所有这些元组的集合并不构成一个关系。

即使元数相同，为了让并是有意义的，在相应列中的所有项必须属于同一个域。考虑PROFESSOR和TEACHING的集合论并集。如果我们要匹配列，在PROFESSOR中的Id、Name和DeptId将与ProfId、CrsCode和Semester相对应。在并集中，第一列的值是相同域的成员，所以是没有问题的。然而第二和第三列显然不属于同一个域（如人的姓名和课程代码）。

为了解决这个问题，我们限制并运算符的范围，只把它应用到*并相容*的关系上去。如果关系的模式满足下面的规则，那么它们是*并相容*（union-compatible）的[⊖]。

• 两个关系中列的数目相同。

• 两个关系中属性的名称相同。

• 两个关系中同名属性的域相同。

对于差集和交集运算符，我们也要求并相容。

图6-2说明了结合选择和投影运算符的集合运算符的一些非平凡的应用。第一个查询检索除了MAT123并且有某个学生的成绩得了C的所有的课程开设情况。第二个查询有点不太自然，但是一个很好的说明。它产生符合以下条件的课程：有人在这门课程里得了C或者这门课程是MAT123。第三个查询列出了有人得了C的MAT123的所有课程开设情况。

这些例子的一个有趣的方面是最初的关系TRANSCRIPT和TEACHING不是并相容的。然而，在经过投影去掉不相容的属性以后，它们就变成相容的了。此图中的所有表达式是在图4-5的运行实例的上下文中计算得到的。

4. *笛卡儿积和重命名*

笛卡儿积（也称为**叉积**）**R × S**，类似于在集合上的叉积运算：如果r和s分别是与R和S相应的关系实例，则这个表达式的值，记为**r × s**，是所有元组*t*的集合，*t*可以由某个元组$r \in$ r和元组$s \in$ s的连接得到[⊜]。

图6-3给出了$\pi_{Id, Name}$（STUDENT）和$\pi_{Id, DeptId}$（PROFESSOR）的某些子集的笛卡儿积。为了更清楚地看出乘积中的元组的哪个部分来自哪个关系，我们用双线标记了边界。

⊖ 一些教材中没有这样的要求，它们并不要求在两个关系中的属性的名称是相同的（但是仍然要求域是相对应的）。在本节的后半部分，我们将引入*重命名运算符*，它将有助于消除这个要求的不同表达之间的差异。

⊜ 关系上集合论的叉积运算和关系的叉积运算之间的差异就在于前者的结果是一组形如(<*a*, *b*>, <*c*, *d*>)的元组的集合，而后者的结果是一组连接的元组，如<*a*, *b*, *c*, *d*>。

CrsCode	Semester
CS305	F1995

$\pi_{\text{CrsCode,Semester}}(\sigma_{\text{Grade='C'}}(\text{TRANSCRIPT}))$
$-\pi_{\text{CrsCode,Semester}}(\sigma_{\text{CrsCode='MAT123'}}(\text{TEACHING}))$

CrsCode	Semester
CS305	F1995
MAT123	S1996
MAT123	F1997

$\pi_{\text{CrsCode,Semester}}(\sigma_{\text{Grade='C'}}(\text{TRANSCRIPT}))$
$\cup\ \pi_{\text{CrsCode,Semester}}(\sigma_{\text{CrsCode='MAT123'}}(\text{TEACHING}))$

CrsCode	Semester
MAT123	S1996

$\pi_{\text{CrsCode,Semester}}(\sigma_{\text{Grade='C'}}(\text{TRANSCRIPT}))$
$\cap\ \pi_{\text{CrsCode,Semester}}(\sigma_{\text{CrsCode='MAT123'}}(\text{TEACHING}))$

图6-2　集合运算符的例子

Id	Name
111223344	Smith, Mary
023456789	Simpson, Homer
987654321	Simpson, Bart

$\pi_{\text{Id, Name}}$（STUDENT）的子集

Id	DeptId
555666777	CS
101202303	CS

$\pi_{\text{Id, DeptId}}$（PROFESSOR）的子集

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

它们的笛卡儿积

图6-3　两个关系和它们的笛卡儿积

现在必须解决一下属性命名的问题，我们迄今为止都尽可能回避这个问题。作为代数表达式参数的关系的模式在系统目录中定义，所以它们属性的名称是已知的。相反，通过计算

表达式而产生的关系是临时创建的，它们的模式没有显式定义。对于一些运算，如σ和π，因为我们可以简单地重用参数关系的模式，所以并不会引起问题。对于∪、∩和−，我们也不会有命名的问题，因为在这些运算中涉及到的关系是并相容的，因此它们有着相同的模式，又可以为查询结果所重用。

笛卡儿积使得我们第一次要处理命名问题。在图6-3中，注意乘积关系的某些属性突然改变了名称。这是因为在运算$\pi_{\text{Id, Name}}$（STUDENT）和$\pi_{\text{Id, DeptId}}$（PROFESSOR）中涉及到的关系有一个同名的属性Id，它将在乘积中出现两次。关系模型不允许同一个关系的不同列具有相同的名称，所以我们通过指定属性来自的关系来为其重命名。

在上面的笛卡儿积的例子中，我们设法找到了一个既简单又自然的重命名规则，我们继续通过在合适的时候给关系名加上前缀来消除属性名的歧义性。然而，这个规则并不总是有效的。例如，在PROFESSOR关系的两个实例做叉积时，它将会失效。我们没有去发明更为复杂的命名规则，而是把这个工作交给程序员，由他们来负责重命名。为此，我们引入**重命名运算符**（renaming operator），它并不属于代数的核心内容，也没有标准的符号表示。我们选用了下面的简单符号表示：

$$expression\ [A_1, \cdots, A_n]$$

这里*expression*是关系代数表达式，并且$A_1, ..., A_n$是用于那个表达式结果中的属性的名称列表。

假设n表示在表达式结果中列的数目，此外，假设通过计算表达式而产生的关系中列之间存在某个标准顺序。例如，在π、σ、∪、∩和−的结果中，这个顺序就是参数关系的模式中属性排列的顺序。对于**R** × **S**，属性应该像图6-3中那样排列：**R**的属性后面排列**S**的属性。例如，

$$(\pi_{\text{Id, Name}}\ (\text{STUDENT}) \times \pi_{\text{Id, DeptId}}\ (\text{PROFESSOR}))$$
$$[\text{StudId, StudName, ProfId, ProfDept}]$$

把图6-3的乘积关系的属性从左到右重命名为StudId、StudName、ProfId、ProfDept。特别地，Id的两次出现分别重命名为StudId和ProfId。

重命名运算符也可以应用于子表达式。下面的例子类似于前面的例子，不同的是在应用其他运算符之前对PROFESSOR关系的属性进行了重命名。

$$\pi_{\text{Id, Name}}\ (\text{STUDENT}) \times \pi_{\text{ProfId, ProfDept}}\ (\text{PROFESSOR}\ [\text{ProfId, ProfName, ProfDept}])$$

结果与前面的例子一样，但是现在属性的名字（从左到右）是Id、Name、ProfId、ProfDept。还要注意的是必须改变第二个π运算符中的属性，因为参数关系的属性在重命名后有了变化。

笛卡儿积的独特之处在于它是关系代数中计算代价最高昂的运算符。考虑**R** × **S**，假设**R**有n个元组，**S**有m个元组，那么笛卡儿积有$n \times m$个元组。另外，乘积中每个元组的规模也比较大。举个具体的例子，假设**R**和**S**都有1000个元组，每个元组有100B。那么**R** × **S**有1 000 000个元组，每个元组有200B。所以，尽管最初的关系总的规模不到0.5MB，但乘积却有200MB。仅把这样的关系写到磁盘上就会付出极高的代价。这只是一个小例子，很快我们将会看到一个查询涉及到三个乃至更多关系是极为寻常的事情。即使只有四个很小的关系，每个关系有100个元组，一个元组有100B，它们的笛卡儿积有100 000 000个元组，每个元组

有400B，即40GB的数据。

在算法分析的课程中，我们已经学到过只要多项式的阶数不太高，具有多项式级时间复杂度的算法是可以接受的。从上面的例子中可以看出，在查询处理中，如果是运算涉及大量的数据，即使是二次多项式算法也不可接受的。因为潜在的巨大代价，查询优化器总是尽可能的避免叉积。

6.1.2 导出运算符

1. 联结

两个关系**R**和**S**的**联结**（join）是形如$\mathbf{R}\bowtie_{\text{联结条件}}\mathbf{S}$的表达式。

联结条件就是我们已经熟悉的σ运算符的选择条件的特殊形式：

$$\mathbf{R}.A_1\ \text{oper}_1\ \mathbf{S}.B_1\ \text{AND}\ \mathbf{R}.A_2\ \text{oper}_2\ \mathbf{S}.B_2\ \text{AND}\ \cdots\ \text{AND}\ \mathbf{R}.A_n\ \text{oper}_n\ \mathbf{S}.B_n$$

这里$A_1, \cdots, A_n$是**R**的属性的子集，$B_1, \cdots, B_n$是**S**的属性的子集。最后，$\text{oper}_1, \cdots, \text{oper}_n$是比较运算符=、≠、>等等。

这些限制意味着连接条件只能有AND连接词（没有OR和NOT），不允许有属性和常数之间的比较。

尽管我们用了新的符号来表示联结，但是它并不是一个全新的运算符。根据定义，上面的联结等价于

$$\sigma_{\text{联结条件}}(\mathbf{R}\times\mathbf{S})$$

然而，因为联结在数据库查询中频繁出现，所有它们就有了拥有自己的符号的特权。

图6-4是两个关系联结的例子。注意联结条件Id < Id，这意味着为了适合于作联结，STUDENT元组中Id属性的值必须比PROFESSOR元组中Id属性的值要小。

STUDENT.Id	Name	PROFESSOR.Id	DeptId
111223344	Smith, Mary	555666777	CS
023456789	Simpson, Homer	555666777	CS
023456789	Simpson, Homer	101202303	CS

$\pi_{\text{Id, Name}}(\text{STUDENT})\bowtie_{\text{Id<Id}}\pi_{\text{Id, DeptId}}(\text{PROFESSOR})$

图6-4　图6-3中关系的联结

注意，联结的定义涉及到作为中间步骤的笛卡儿积。因此，联结是潜在的代价昂贵的运算。但是，笛卡儿积隐藏在联结中。实际上，笛卡儿积很少独立地出现在典型的数据库查询中。因而，尽管中间结果（乘积）可能是很大的，但最后的结果却是可处理的，因为乘积中只有一小部分元组可能满足联结条件。

例如，图6-4中联结的结果是图6-3中笛卡儿积大小的一半。联结的大小只是相应叉积的大小的很小一部分是很常见的事情。这里需要技巧的部分是计算联结而不必计算中间的笛卡儿积！这可能听起来很不可思议，但是有若干种方法能完成这个工作，查询优化器会经常这么做。

上面描述的是普通的联结，有时也被称为theta**联结**（theta-join），因为在古代希腊字母θ

用来表示联结条件。尽管theta联结在查询处理中是很常见（查询列出薪水比其经理多的所有职员就涉及到theta联结），更常见的是所有的比较都是等号：

$$\mathbf{R}.A_1 = \mathbf{S}.B_1 \text{ AND } \cdots \text{ AND } \mathbf{R}.A_n = \mathbf{S}.B_n$$

使用这样条件的联结叫做**等值联结**（equijoin）。

等值联结实质上给关系数据库赋予了智能，因为它们把分散在数据库中的根本不同的数据联系到一起。使用等值联结，程序员用几行SQL代码就可以揭示隐藏在数据之下的复杂联系。

下面的例子说明如何找到在1994年秋季授课的教授的姓名：

$$\pi_{\text{Name}}(\text{PROFESSOR} \bowtie_{\text{Id = ProfId}} \sigma_{\text{Semester = 'F1994'}}(\text{TEACHING}))$$

内层的联结把PROFESSOR中的元组依照TEACHING中的元组来排列，TEACHING中的元组描述了各个教授在1994年秋季教授课程的情况。最后的投影去掉了不感兴趣的属性。找出在1995年秋季开设的课程名称和任课教授的姓名也很容易：

$$\pi_{\text{CrsName, Name}}(\\ (\text{PROFESSOR} \bowtie_{\text{Id = ProfId}} \sigma_{\text{Semester = 'F1995'}}(\text{TEACHING})) \\ \bowtie_{\text{CrsCode = CrsCode}} \text{COURSE} \\ )$$

上面的查询涉及到两个联结。我们在第一个连接的周围加上圆括号，来指出计算联结的次序。然而，这并不是必须的，因为联结恰好是结合（associative）运算符，这可以由其定义得以证明。

$$\begin{aligned}
&\mathbf{R} \bowtie_{\text{cond}_1} (\mathbf{S} \bowtie_{\text{cond}_2} \mathbf{T}) \\
&= \sigma_{\text{cond}_1}(\mathbf{R} \times \sigma_{\text{cond}_2}(\mathbf{S} \times \mathbf{T})) && \text{通过}\bowtie\text{的定义} \\
&= \sigma_{\text{cond}_1}(\sigma_{\text{cond}_2}(\mathbf{R} \times (\mathbf{S} \times \mathbf{T}))) && \text{因为}\sigma\text{和}\times\text{可交换（检验!）} \\
&= \sigma_{\text{cond}_2}(\sigma_{\text{cond}_1}(\mathbf{R} \times (\mathbf{S} \times \mathbf{T}))) && \text{因为两个}\sigma\text{可交换（检验!）} \\
&= \sigma_{\text{cond}_2}(\sigma_{\text{cond}_1}((\mathbf{R} \times \mathbf{S}) \times \mathbf{T})) && \text{通过}\times\text{的结合性（检验!）} \\
&= (\mathbf{R} \bowtie_{\text{cond}_1} \mathbf{S}) \bowtie_{\text{cond}_2} \mathbf{T} && \text{通过}\bowtie\text{的定义}
\end{aligned}$$

上面的双联结查询说明了另外一个要点。像TEACHING.CrsCode = COURSE.CrsCode这样用来测试同名属性（但在不同关系中）等值的联结条件是很常见的。这主要是因为给表示同一事物但分属不同关系的属性赋予相同的名字是一个好的设计思路。例如，在COURSE、TEACHING和TRANSCRIPT中CrsCode的语义是一样的（这就是在这三个关系中都使用相同名字的原因）。因为找出数据中的隐藏联系等同于在不同关系中比较类似属性，所以上面的设计思路导致了同名属性相等的等值联结条件[⊖]。

事实上，这个设计思路使得大多数等值联结的联结条件只是同名属性之间的相等。认识到它们重要性之后，这样的联结就有了它们自己的名称：**自然联结**（natural join）。自然联结

⊖ 我们很自然会问，为什么我们为STUDENT(Id)和TRANSCRIPT(StudId)中的Id属性使用了不同的名称，这显然违反了之前提到的设计惯例。这个答案是很简单的。我们之所以这样做是为了能够从一个合适大小的模式中举出更多的例子。在一个设计良好的数据库模式中，StudId可能在两个地方都用到；同样的，ProfId可能在PROFESSOR和TEACHING中都用到；Id可能在这四个地方都用到。

的作用实际上不仅仅如此。第一，联结条件是参与联结的两个关系中所有同名属性都分别相等。第二，因为相等的属性实际上表示两个关系中同一个事物（由它们名称的一致性来表示），所以没有必要把两列都保留下来。因此，其中一列通过投影而消除。总之，**R**和**S**的自然联结，记为**R**⋈**S**，由下面的关系表达式来定义：

$$\pi_{\text{attr-list}}\,(\sigma_{\text{join-cond}}\,(\mathbf{R}\times\mathbf{S}))$$

其中，

1) attr-list = attributes (**R**) ∪ attributes (**S**)，即用在投影运算符中的属性序列包含参数关系并集中的所有属性，但去除重复的属性名。因为删除了重复属性，所以没有必要对属性进行重命名。

2) 联结条件join-cond具有如下形式：

$$\mathbf{R}.A_1 = \mathbf{S}.A_1 \text{ AND } \cdots \text{ AND } \mathbf{R}.A_n = \mathbf{S}.A_n$$

这里$\{A_1, \cdots, A_n\}$ = attributes (**R**) ∩ attributes (**S**)。这就是说，它是**R**和**S**中公共属性的列表。注意，自然联结的符号省略了联结条件，因为条件是隐含的（并且是唯一的），由赋予要联结的关系的属性名所决定。

典型的使用自然联结的例子是下面的查询：

$$\pi_{\text{StudId, ProfId}}\,(\text{TRANSCRIPT} \bowtie \text{TEACHING})$$

这里列出了所有选修过课程的学生的Id和任课教授的Id。

为了进一步说明自然联结和等值联结的区别，比较下面两个表达式是很有意义的：

$$\text{TRANSCRIPT} \bowtie \text{TEACHING}$$

$$\text{TRANSCRIPT} \bowtie_{\text{Cond}} \text{TEACHING}$$

这里等值联结条件Cond是TRANSCRIPT.CrsCode = TEACHING.CrsCode AND TRANSCRIPT.Semester = TEACHING. Semester。两个表达式都是等值联结，并且都使用了相同的联结条件（自然联结是隐含使用的）。然而，结果关系却有着不同的属性集合。

自然联结

StudId, CrsCode, Semester, Grade, ProfId

等值联结

StudId, TRANSCRIPT.CrsCode, TEACHING.CrsCode,

TRANSCRIPT.Semester, TEACHING.Semester, Grade, ProfId

两个表达式实质上表示了相同的信息。然而，等值联结的模式多了两个属性（它们是其他属性的重复），自然联结的优势在于有较简单的属性命名规则。

除了发现数据中的隐藏联系，联结还能用于某些计数任务。下面的表达式说明如何找出至少选修两门不同课程的所有学生：

$$\pi_{\text{StudId}}\,(\sigma_{\text{CrsCode} \neq \text{CrsCode2}}\,(\text{TRANSCRIPT} \bowtie \text{TRANSCRIPT}\,[\text{StudId, CrsCode2, Semester2, Grade2}]))$$

这个技术的一个明显的限制是如果我们想得出选修了15门课程的学生，那么我们就不得不将TRANSCRIPT与其自身联结15次。一个更好的方法是扩展关系代数使之具有所谓的聚合（aggregate）函数，其中就有计数运算符。我们在这里并不讨论这种可能性，但是我们将在6.2节介绍SQL的内容中再来讨论聚合函数。

谈到联结，就不能不提到下面这一点——交运算是自然联结的特殊形式。假设**R**和**S**是并相容的，由定义可以直接得到$\mathbf{R} \cap \mathbf{S} = \mathbf{R} \bowtie \mathbf{S}$。

2. 除法运算符

尽管联结运算符使查询回答具有一定的智能性，除法运算符则是最难以理解和正确使用的。

当你很想找出哪个教授讲授计算机科学系开设的所有课程或者哪个学生选修过电子工程系每个教授讲授的课程，那么除法运算符就非常有用了。这里的关键在于我们想在一个关系中找出匹配另一个关系中所有元组的元组。

下面是精确的定义。假设**R**是具有属性$A_1, \cdots, A_n, B_1, \cdots, B_m$的关系模式，**S**是具有属性$B_1, \cdots, B_m$的关系模式。换句话说，**S**的属性集是**R**的属性集的一个子集。**R**除以**S**是形如**R/S**的表达式。如果**r**和**s**是数据库中与**R**和**S**相对应的关系实例，那么数据库中**R/S**的值（记为r/s）是属性集为$A_1, \cdots, A_n$的关系，它包含了所有这样的元组<*a*>：对于**s**中的每个元组<*b*>，联结后的元组<*a*, *b*>在**r**中。这个定义的示意图如图6-5所示。

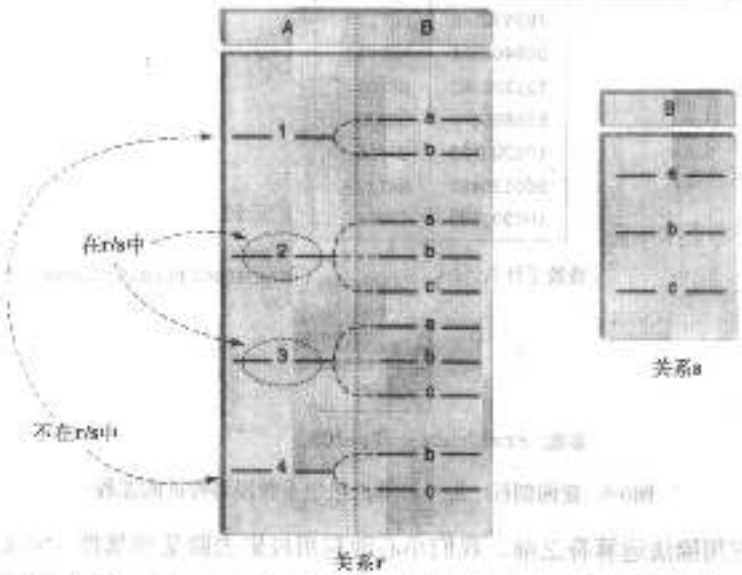


图6-5 除法运算符

定义除法的一个等价方法是

$$\langle a \rangle \in \mathbf{r/s} \text{ 当且仅当 } (\{\langle a \rangle\} \times \mathbf{s}) \subseteq \mathbf{r}$$

因此，如果把×看成是乘法，那么可以把除法看成是乘法的逆运算。

举个例子，考虑查询列出计算机科学系每个教授讲授过的所有课程。这里的"每个教授"是指"记录在数据库中的每个教授"。也就是说，我们假设数据库描述了我们所知道的一切情

况。不仅数据库中的每个元组代表了有关被建模的现实世界企业的一个事实，而且除此之外没有其他已知事实能表示成数据库关系中的元组。这被称为封闭世界假设（closed-world assumption），隐含在关系查询处理中。

图6-6给出了三个关系。关系

$$\text{PROFCS} = \pi_{\text{Id}}\,(\sigma_{\text{DeptId = 'CS'}}\,(\text{PROFESSOR}))$$

包含了所有已知的计算机科学系教授的Id。关系

$$\text{PROFCOURSES} = (\pi_{\text{ProfId, CrsCode}}\,(\text{TEACHING}))\ [\text{Id, CrsCode}]$$

是课程和任课教授（任何时候）之间的一个联系，它包含了所有已知的这种联系（注意，我们应用了重命名运算符来保证PROFCOURSES的第一个属性与PROFCS的属性同名）。第三个关系给出了PROFCOURSES/PROFCS，这是查询的结果。

PROFCS	Id
	101202303
	555666777

所有计算机系教授：$\pi_{\text{Id}}(\sigma_{\text{DeptId='CS'}}(\text{PROFESSOR}))$

PROFCOURSES	Id	CrsCode
	783432188	MGT123
	009406321	MGT123
	121232343	EE101
	555666777	CS305
	101202303	CS315
	900120450	MAT123
	101202303	CS305

谁教了什么：$(\pi_{\text{ProfId,CrsCode}}(\text{TEACHING}))\,[\text{Id,CrsCode}]$

CrsCode
CS305

答案：ProfCourses/ProfCS

图6-6　查询剖析：每个计算机科学系教授讲授过的课程

注意，在应用除法运算符之前，我们小心地利用投影去除某些属性（对某些其他的属性进行了重命名）来使得除法运算符是可应用的。在这种情况下显然必须进行投影。

为了说明问题，我们考虑下面的查询：检索选修过曾教过课的每个教授的课的所有学生。被除数需要的是一个能联系学生和在某门课程中教过他们的教授的关系。我们可以通过TRANSCRIPT和TEACHING的自然联结得到这个关系：

$$\text{STUDPROF} = \text{TRANSCRIPT} \bowtie \text{TEACHING}$$

STUDPROF具有属性StudId、CrsCode、Semester、Grade和ProfId，这些比我们所需要的信息多，所以我们用投影去掉不想要的列来得到被除数：$\pi_{\text{StudId, ProfId}}$（STUDPROF）。

因为我们感兴趣的是选修过“每个教授”的课程的学生，除数需要的是一个对“每个教授”都有一个元组的关系。教过课的教授在TEACHING关系的元组里都有他们的Id。（我们不想在这里使用PROFESSOR，因为它可能包含没有做过任何教学工作的教授的元组。）因此，除数是π_{ProfId}（TEACHING），我们查询的答案是

$$\pi_{StudId,\ ProfId}\ (\text{STUDPROF}) \ /\ \pi_{ProfId}\ (\text{TEACHING}) \tag{6.1}$$

这里还有一个例子：找出选修过计算机科学系所有教授都教过的课程的所有学生。这里需要两个除法：

$$\begin{aligned}&(\pi_{Id,\ Name}\ (\text{STUDENT}))\ [\text{StudId, Name}] \bowtie\\&(\pi_{StudId,\ CrsCode}\ (\text{TRANSCRIPT})\ /\\&((\pi_{ProfId,\ CrsCode}\ (\text{TEACHING}))\ [\text{Id, CrsCode}]\ /\\&\pi_{Id}\ (\sigma_{DeptId\ =\ 'CS'}\ (\text{PROFESSOR}))))\end{aligned} \tag{6.2}$$

这里的第二个除法来自图6-6中所应用的表达式，它会产生每个计算机系教授讲授过的所有课程。因此，表达式的最后三行定义了选修过所有这样课程的所有学生的Id。第一行的自然联结是用来得到这些学生的姓名。我们要在联结之前先对STUDENT关系的属性进行重命名。

我们总结一下，除法运算符如何用其他的关系运算符（投影、差、叉积）来表达。假设**R**是具有属性*A*，*B*的关系，**S**是具有单一属性*B*的关系（下面的构造即使在*A*和*B*没有交集的情况下也是正确的）。在不使用除法的情况下，表达式**R/S**可以计算如下：

$$\begin{aligned}&\mathbf{T}_1 = \pi_A\ (\mathbf{R}) \times \mathbf{S} && \text{在}\mathbf{R}\text{中}A\text{的值和}\mathbf{S}\text{中}B\text{的值之间所有可能的关联。}\\&\mathbf{T}_2 = \pi_A\ (\mathbf{T}_1 - \mathbf{R}) && \text{在}\mathbf{R}\text{中所有那些不和}\mathbf{S}\text{中每个}B\text{的值都有联系的}A\text{的值，}\\&&& \text{这些正好就是那些不应该出现在答案中的}A\text{的值。}\\&\mathbf{T}_3 = \pi_A\ (\mathbf{R}) - \mathbf{T}_2 && \text{答案：在}\mathbf{R}\text{中和}\mathbf{S}\text{中所有}B\text{的值都有联系的}A\text{的值。}\end{aligned} \tag{6.3}$$

6.2 SQL的查询子语言

SQL是使用最为广泛的关系数据库语言。1974年提出了最初版本，之后发展演化至今。使用最广泛的版本通常称为SQL-92，是美国国家标准协会（ANSI）的标准。语言在继续演化，SQL:1999最近也已经完成。SQL和任何一种快速演化的语言一样，其形式受众多用户的影响，所以也变得极其复杂了。本节和下一节就是向你介绍它的一些复杂性。但是，你应该知道，全面阐述该语言足可以写成一本书。例如，[Melton and Simon 1992, Date and Darwen 1997]是有关SQL-92的详细介绍的参考书，[Gulutzan and Pelzer 1999]描述了SQL:1999。然而，商业数据库并不总是遵守这些标准，与供应商有关的参考书几乎是重大的应用开发所必需的。

SQL可以通过从终端向数据库管理系统直接提交SQL语句来交互使用。然而，特别是在事务处理系统中，SQL语句通常嵌入在较大的程序中，在运行时将其提交给数据库管理系统，并处理返回的结果。有关嵌入的相关内容将在第10章中讨论。

6.2.1 简单的SQL查询

简单的SQL查询是很容易设计的。想要列出电子工程（EE）系的所有教授？可以利用下列查询：

```
SELECT   P.Name
FROM     PROFESSOR P
WHERE    P.DeptId = 'EE'
```

注意，符号P称为**元组变量**（tuple variable）。它的取值范围是关系PROFESSOR的元组[⊖]。实际上在这个语句中可以不使用元组变量。我们同样可以不在FROM子句中使用它，把这些属性称为Name和DeptId即可，而不用称为P.Name和P.DeptId，因为这样并不会引起混淆。我们也可以使用PROFESSOR.Name和PROFESSOR.DeptId，但是这样就相当麻烦。不过一旦引入了元组变量，就不必在语句中任何地方使用表的全名了。

我们很快就会看到，在某些情形下使用元组变量是很方便的，在某些情形下也是很必要的。所以，在SELECT语句中使用元组变量是一种很好的编程习惯，不会导致小错误。因此，在本书中我们总是声明所有的元组变量。

虽然这个语句非常简单，但是重要的是理解如何计算SELECT语句。SELECT语句可能会很复杂，对语句遵循一定的操作流程是了解当前处理进展的唯一方法。当然，不同的数据库管理系统可能在计算同一个语句的时候采用不同的策略，但是因为它们都产生相同的结果，所以描述一种特别简单的策略是很有用的。

1. *简单查询的计算策略*

计算SQL查询的基本算法可以描述如下:

第一步，计算FROM子句。它产生一个表，这个表是作为参数的那些表的笛卡儿积。如果一个表像查询（6.9）那样在FROM子句中出现多次，那么这个表在积中也会出现相同的次数。

第二步，计算WHERE子句。它对第一步中产生的表的每一行进行单独处理。行的属性值将替换条件的属性名，然后计算这个条件。由WHERE子句产生的表只包含那些条件为真的行。

第三步，计算SELECT子句。它利用第二步中产生的表，只保留作参数的那些列，SELECT语句输出结果表。

当我们讨论SQL语言新的特点的时候，我们将会给这个策略加上额外的步骤，但是基本思想是不变的。每个子句产生一个表，作为下一个要计算的子句的输入。在某个计算中可能不需要某些步骤。例如，因为SELECT语句可以没有WHERE子句，所以就有可能不必计算步骤2了。其他的步骤也可能是微不足道的。例如，虽然每个SELECT语句必须有FROM子句，但是如果该子句只指定了一个关系，那么就没必要做笛卡儿积了，关系也只是传给步骤2。

与上面的SQL查询等价的关系代数是

$$\pi_{\text{Name}}\left(\sigma_{\text{DeptId = 'EE'}}(\text{PROFESSOR})\right)$$

2. *联结查询*

表达两个关系之间联结的查询遵照的模式与上面相同。下面的查询涉及到联结，它返回在1994年秋季授课的所有教授的列表。

```
SELECT   P.Name
FROM     PROFESSOR P, TEACHING T                                    (6.4)
WHERE    P.Id = T.ProfId  AND  T.Semester = 'F1994'
```

⊖ 掌握SQL意味着要在头脑中塞满冗余的术语。例如，SQL变量也称作“表别名”。我们将在第7章中看到这些变量对应于元组关系演算的元组变量。关系演算是SQL的真实理论基础。

注意，这个例子中的元组变量澄清了这个语句的含义，因为它们确定了每个属性来源于哪个表。如果PROFESSOR的Id属性被称为ProfId，而不是简单的Id，我们就不得不用元组变量来区别这两个引用。

语句的计算遵循上面列出的步骤。与前面的例子相对比，这个例子处理FROM子句时涉及到计算笛卡儿积。

花点时间让自己确信这个查询等价于下面的关系代数表达式。

$$\pi_{\text{Name}}\,(\text{PROFESSOR} \bowtie_{\text{Id = ProfId}} \sigma_{\text{Semester = 'F1994'}}\,(\text{TEACHING})) \tag{6.5}$$

它把联结条件和选择条件区别开来。联结条件Id = ProfId确保组合两个表中相关的元组。组合关系PROFESSOR（描述了某个特定的教授）中的元组和关系TEACHING（描述了由不同的教授讲授的某门课）中的元组是没有意义的。因此，联结条件消除了没有意义的部分。另一方面，选择条件Semester = 'F1994'消除了与这个查询无关的元组。

3. SQL和关系代数之间的联系

关系代数表达式

$$\pi_{\text{Name}}\,(\sigma_{\text{Id = ProfId AND Semester = 'F1994'}}\,(\text{PROFESSOR} \times \text{TEACHING}))$$

与（6.5）等价，但是它不包含任何联结运算符。事实上，它合并了联结条件和选择条件。虽然结果表达式导致了一个计算相应的SQL查询的最低效的方法，但是它简单、一致，从语法的角度来看也更贴近于等价的SQL语句。更一般地，查询模板

```
SELECT   TargetList
FROM     REL1 V1, ..., RELn Vn
WHERE    Condition
```
(6.6)

基本上等价于代数表达式

$$\pi_{\mathit{TargetList}}\,\sigma_{\mathit{Condition}}\,(\text{REL}_1 \times \cdots \times \text{REL}_n) \tag{6.7}$$

“基本上”意味着还要把*Condition*转换成关系代数的形式。更具体的，考虑查询找出在1995年秋季开设的课程名称以及讲授这些课程的教授的姓名。用SQL来表达的形式如下所示：

```
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
```
(6.8)

以低效但一致的形式来表达上面描述的代数表达式是

$$\pi_{\text{CrsName, Name}}\,(\sigma_{\text{Condition}}\,(\text{PROFESSOR} \times \text{TEACHING} \times \text{COURSE}))$$

*Condition*表示WHERE子句的内容（可以做适当修改以适合关系代数）。

```
Id = ProfId AND TEACHING.CrsCode = COURSE.CrsCode AND
    Semester = 'F1995'
```

4. 自联结查询

让我们回到前面考虑的查询上来，找出所有至少选修过两门课程的学生。我们把它用关系代数表达为

π_{StudId} (

$\sigma_{CrsCode \neq CrsCode2}$ (

TRANSCRIPT $\bowtie_{StudId = StudId}$

TRANSCRIPT [StudId, CrsCode2, Semester2, Grade2]

))

注意，我们已经把关系TRANSCRIPT和它自身进行了联结。为了在关系代数中完成这一工作，我们必须对TRANSCRIPT的第二次出现应用重命名运算符。在SQL中，我们必须在FROM子句中两次提到TRANSCRIPT关系，声明这个关系上的两个不同的变量。每个变量将表示联结中TRANSCRIPT的一次出现。

```
SELECT   T1.StudId
FROM     TRANSCRIPT T1, TRANSCRIPT T2
WHERE    T1.CrsCode <> T2.CrsCode AND T1.StudId = T2.StudId
```
(6.9)

这个查询中，符号< >是SQL中表示“不相等”的方法。注意，我们确实需要两个不同的元组变量在TRANSCRIPT上使用；如果我们只使用一个变量T，那么条件T.CrsCode < > T.CrsCode将永远也得不到满足，.使得查询的结果是空关系。因此，我们无法在不使用元组变量的情况下还能清楚地表达这个查询。在这个例子中使用元组变量是必需的。

5. 检索不同的结果

我们从第4章已经知道关系是集合，所以不允许出现重复元组。然而，许多关系运算符可以产生**多集**（multiset，也就是类似于集合的对象，其中可能包含同一元素的多次出现）作为计算的中间结果。例如，如果我们从图4-5中描绘的关系TEACHING的实例中去掉属性Semester，我们将得到一个元组列表，其中包含<009406321, MGT123>的重复出现，其他元组也一样。因此为了让SQL查询

```
SELECT   T.ProfId, T.CrsCode
FROM     TEACHING  T
```
(6.10)

返回一个关系（关系数据模型是这样要求的），查询处理器必须对SELECT子句中指定的投影结果进行额外的一次扫描，以消除所有的重复。在许多情况下，程序员并不愿意为消除重复付出代价。因此，SQL的设计者决定，在默认情况下是不消除重复元组的，除非使用关键字DISTINCT显式地要求消除元组。

```
SELECT   DISTINCT T.ProfId, T.CrsCode
FROM     TEACHING T
```
(6.11)

注意，这个查询没有WHERE子句，WHERE子句在SQL中是可选的。当没有WHERE子句的时候，不管FROM子句中元组变量的值是什么，WHERE条件都被假设为真。

虽然WHERE子句是可选的，但是SELECT子句和FROM子句不是可选的。

6. 注释

和各种编程语言一样，程序员希望用注释来给查询加上评注。在SQL中，注释是以双减号（--）开头的字符串，以新行结束。例如，

```
-- An example of SELECT DISTINCT
SELECT   DISTINCT T.ProfId, T.CrsCode
FROM     TEACHING T    --No WHERE clause!
```
(6.12)

7. WHERE子句中的表达式

到目前为止，WHERE子句中的条件都是属性和常数或其他属性的比较。对于数值，SQL提供了下面的比较运算符：=（相等）、<>（不相等）、>（大于）、>=（大于等于）、<（小于）和<=（小于等于）。

所有这些运算符可以应用于数值，也可以运用于字符串（字符串也可以用LIKE运算符来与模式比较，我们将在后面描述这一点）。字符串从左到右按照字母顺序来比较。从本书的目的出发，我们把重点关注ASCII符号，并假定比较两个字符串时字符的顺序由分配给这些字符的ASCII代码决定。

这些比较运算符的运算项可以是**表达式**（expression），而不仅仅是单个的属性或常数。对于数值来说，表达式由通常的运算符（*、+等等）组成。对于字符串来说，可以使用连接运算符||。例如，假设EMPLOYEE关系具有属性SSN、BossSSN、LastName、FirstName、Salary。下面的查询

```
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
```

返回所有薪水高于老板薪水两倍并且姓是“Mc”和名的连接（如Donald McDonald）的职员。

8. SELECT子句中的表达式和特点

下面将会讨论SELECT子句的许多特点。在2.2节中，我们注意到星号（*）表示FROM子句中所有关系的所有属性的列表。注意，当使用星号的时候，如果一个关系在FROM子句中出现两次（或多次），那么它的属性也会出现两次（或多次）。例如，查询

```
SELECT   *
FROM     EMPLOYEE  E, EMPLOYEE M
```

和

```
SELECT   E.SSN, E.BossSSN, E.FirstName, E.LastName, E.Salary,
         M.SSN, M.BossSSN, M.FirstName, M.LastName, M.Salary
FROM     EMPLOYEE E, M
```

是一样的。

SQL也允许表达式作为目标列表的一部分，而不仅仅出现在WHERE子句中（这个问题我们已经看到）。例如，假设审计办公室想要得到职员和他们直接上司的薪水差异的列表。这可以用下面的查询来完成，这里目标列表的最后一个元素是表达式。

```
SELECT   E.SSN, M.SSN, M.Salary - E.Salary
FROM     EMPLOYEE  E,  EMPLOYEE M
WHERE    E.BossSSN = M.SSN
```

如果上面的查询是交互使用的，那么大部分数据库管理系统会给出一张表，这张表的前两列标记为SSN，最后一列根本没有标记。显然，这不是很令人满意的，因为它需要用户来记住SELECT子句中数据项的含义。为了解决这个问题，SQL允许程序员改变属性名称并可以分配不存在的名称。这要借助于关键字AS才能得以完成。例如，我们可以用如下方法修改前面的查询。

```
SELECT   E.SSN AS EmplId,
         M.SSN AS MngrId,
         M.Salary - E.Salary AS SalaryDifference
FROM     EMPLOYEE  E, EMPLOYEE M
WHERE    E.BossSSN = M.SSN
```

这个查询除了输出的形式外，其他方面都与前面的查询相同。最后一个查询的属性将会给显示为EmplId、MngrId和SalaryDifference。

9. 否定

WHERE子句的任何条件都可以用NOT来取非。例如，在查询（6.9）中我们可以用NOT（T1.CrsCode = T2.CrsCode）来替换T1.CrsCode < > T2.CrsCode。取非的条件不需要是原子的，即它可以包含任意个由AND或OR连接的子条件，甚至可以像下面的例子中那样嵌套使用NOT。

```
NOT (E.BossSSN = M.SSN AND E.Salary > 2 * M.Salary
    AND NOT (E.LastName = 'Mc'|| E.FirstName))
```

6.2.2 集合运算

SQL使用关系代数中的集合运算符。下面有一个简单的查询，它用到了集合运算符：找出所有在计算机科学系或电子工程系工作的教授。

```
(SELECT  P.Name
FROM     PROFESSOR P
WHERE    P.DeptId = 'CS' )
UNION                                                    (6.13)
(SELECT  P.Name
FROM     PROFESSOR P
WHERE    P.DeptId = 'EE' )
```

SQL的表示是自解释的。查询包含两个子查询：一个查询检索出所有计算机科学系的教授，另一个查询检索出所有电子工程系的教授。它们的结果使用代数中的UNION运算符合并到一个关系中。

虽然这个例子说明了集合运算符在SQL中的基本用法，但是在这里使用UNION的优势并不明显。这个查询不用UNION就可以改写为更加有效的形式。

```
SELECT   P.Name
FROM     PROFESSOR P                                     (6.14)
WHERE    P.DeptId = 'CS' OR P.DeptId = 'EE'
```

下一个例子是查询找出所有计算机科学系的教授和曾经教过计算机科学课程的教授，该查询更加复杂，集合运算符的优势也就更为明显了。

假定所有计算机科学课程的代码都是以CS开头。在设计查询的时候，我们需要根据字符串（课程代码）来匹配模式。为了检验一个字符串是否匹配一个模式，SQL提供了LIKE谓词。例如，T.CrsCode LIKE 'CS%'检验T.CrsCode的值匹配以CS开头并且还可以有零个或多个额外字符的模式。SQL的模式类似于UNIX或DOS中的通配符，但是SQL中建立模式的方法有限：

除了%，还有_，它可以和任意的某个字符匹配[⊖]。

如果不用UNION运算符，可以用如下方法找出计算机科学系教授或那些教过计算机科学课程的教授：

```
SELECT   P.Name
FROM     PROFESSOR P,  TEACHING T
WHERE    (P.Id = T.ProfId  AND  T.CrsCode LIKE 'CS%')
         OR (P.DeptId = 'CS')
```
(6.15)

我们看到，当WHERE条件变得较长较复杂的时候，它就变得比较难读和理解了。有了UNION运算符，我们可以用下面的方法重写这个查询：

```
(SELECT  P.Name
FROM     PROFESSOR P, TEACHING T
WHERE    P.Id = T.ProfId AND  T.CrsCode LIKE 'CS%')
UNION                                                    (6.16)
(SELECT  P.Name
FROM     PROFESSOR P
WHERE    P.DeptId = 'CS')
```

尽管这个查询并不比（6.15）简洁，但是它更加模块化，也更加容易理解。

如果我们想改变我们的查询来检索所有教授计算机课程但不是计算机系的教授的人，我们可以简单的修改（6.16）：用EXCEPT替换UNION，这里EXCEPT是关系代数中减法运算符对应的SQL名称。把（6.15）改为新的查询比较微妙：（6.15）的WHERE子句要改成

```
P.Id = T.ProfId AND T.CrsCode LIKE 'CS%' AND P.DeptId <> 'CS'
```

继续集合运算符的讨论，假设我们需要找出所有既选修过“事务处理”课程（CS315），又选修过“数据库系统”课程（CS305）的学生。首先，我们可能构造下面的查询：

```
SELECT   S.Name
FROM     STUDENT S, TRANSCRIPT  T
WHERE    S.StudId = T.StudId AND T.CrsCode = 'CS305'
         AND T.CrsCode = 'CS315'
```
(6.17)

然而，再细致地检查一下，我们就会发现这个SQL查询并不是我们所需要的，因为它要求在TRANSCRIPT中有一个元组能使T.CrsCode既等于CS305又等于CS315，而这是不可能的。这个简单的例子说明了一个常见的错误，即没有认识到需要使用另一个元组变量。针对我们的问题，正确SQL查询是

```
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'
```
(6.18)

注意，我们用了关系TRANSCRIPT上的两个不同的元组变量来表达学生S选修过两门不同课程的事实。

⊖ 假设你要构造一个模式，其中的特殊字符%和_就表示它们自己。例如，假设你需要匹配以“_%”开头的所有字符串。这是可以完成的，虽然有些麻烦：你必须先声明一个转义字符，然后把它作为特殊字符的前缀，来让SQL知道这些字符表示它们自己。例如，C.Descr LIKE '_\%__$' ESCAPE '\' 把C.Descr的值和下面的匹配字符串的模式做比较：以“_%”开头，跟着一对任意字符，最后以符号$结尾。这里通过ESCAPE子句把“\”声明为转义字符，然后用来转义“%”和“_”。

这和最初的话题——关系代数中的集合运算符有什么关系呢？原来，INTERSECT运算符让我们可以用一种更为模块化更不易出错的方式来重写（6.18）。

```
(SELECT  S.Name
FROM     STUDENT S, TRANSCRIPT  T
WHERE    S.StudId = T.StudId  AND T.CrsCode = 'CS305')
INTERSECT                                                    (6.19)
(SELECT  S.Name
FROM     STUDENT S, TRANSCRIPT  T
WHERE    S.StudId = T.StudId AND T.CrsCode = 'CS315')
```

注意，我们这里并不需要在同一个关系上安排多个变量，这稍微减少了一些出错的危险。实际上我们写了两个很简单，但本质上又很类似的查询，然后取了它们结果的交集。

1. 集合构造器

最后我们介绍一个相关的特点，即在SQL查询中能够构造有限集合的构造器。集合构造器的语法很简单：（set-elem$_1$, set-elem$_2$, …, sel-elem$_n$）。运算符IN让我们可以检查一个特定的元素是否在这个集合里。例如，借助于集合构造器可以把查询（6.14）简化为

```
SELECT   P.Name
FROM     PROFESSOR P                                         (6.20)
WHERE    P.DeptId  IN   ('CS','EE')
```

注意，如果我们把查询（6.17）的WHERE子句替换成

```
S.StudId = T.StudId  AND  T.CrsCode  IN ('CS305','CS315' )
```

我们得到一个含义不同于（6.17）的查询。这个问题将在练习6.13中作进一步研究。

2. 否定和中缀比较运算符

之前我们讨论过NOT运算符。对于一些像LIKE和IN这样的中缀运算符，SQL提供了两种形式的否定：NOT（X LIKE Y）和与之等价的X NOT LIKE Y。类似地，可以把NOT （X IN Y）写成X NOT IN Y。

6.2.3 嵌套查询

如果完成每个任务就只有一种方法，那么就先去了使用SQL的乐趣。考虑查询选择所有在1994年秋季授课的教授。（6.4）给出了以SQL表示的一种方法，但是（至少）还有一种完全不同的方法：首先用**嵌套子查询**（nested subquery）计算出在1994年秋季授课的所有教授的集合，然后提取他们的名字。

```
SELECT   P.Name
FROM     PROFESSOR P
WHERE    P.Id IN
         -- A nested subquery
         (SELECT T.ProfId
          FROM TEACHING T
          WHERE T.Semester = 'F1994')
```

注意，在这个例子中，嵌套子查询只计算一次，然后根据此计算结果在WHERE子句的条件中检验PROFESSOR的每一行。

上面的例子说明了嵌套子查询的一个优势，即增加了可读性。然而，仅仅能够增加可读

性并不能成为使用它的充足理由，因为大多数的查询处理器不能很好地优化嵌套子查询。使用嵌套子查询的主要原因是它增强了SQL的表达能力，没有嵌套子查询，一些查询就不能通过自然的方式表达出来。

考虑查询列出没有选修过任何课程的所有学生。这个查询听起来很简单，但是在SQL中不用嵌套子查询（或者EXCEPT语句，可以自己试试在SQL中使用EXCEPT语句），它就不能表达出来。

```
SELECT   S.Name
FROM     STUDENT  S
WHERE    S.Id NOT IN
         -- Students who have taken a course
         (SELECT  T.StudId
          FROM    TRANSCRIPT T)
```
(6.21)

作为最后一个例子，子查询可以用于从一张表中析取出标量值。假设你想知道哪个职员的薪水比你高（你的Id是111111111），你可以用子查询从EMPLOYEE中得到你的薪水值：

```
SELECT   E.Id
FROM     EMPLOYEE  E
WHERE    E.Salary >
         (SELECT  E1.Salary
          FROM    EMPLOYEE E1
          WHERE   E1.Id = '111111111')
```
(6.22)

上面的查询就可以找出薪水比你高的所有职员。

1. 相关嵌套子查询

一方面，嵌套子查询可以提高查询的可读性。另一方面，它也是SQL中最复杂、最昂贵、最容易出错的特性之一。这个复杂性从很大程度上来说要归因于**查询相关性**（query correlation），即在外层查询中定义变量，在内层查询中使用它们的能力。嵌套和相关性类似于编程语言中begin/end块的概念和变量的作用域的相关思想。

为了进行说明，我们假设需要找出为在下一个学期（为了明确起见，比方说是1999年秋季）计划教课的教授做助教的学生。对于每个教授，我们计算出他在那个学期要教授的所有课程的集合，然后找到选修其中一门课程的学生（也就有资格做助教）。可以在嵌套子查询里计算教授所教的课程的列表，在外层查询里把教授与做助教的学生联系起来。下面是这个计划在SQL中的实现：

```
SELECT   R.StudId,  P.Id,  R.CrsCode
FROM     TRANSCRIPT R,  PROFESSOR P
WHERE    R.CrsCode  IN
         -- Courses taught by P.Id in F1999
         (SELECT  T1.CrsCode
          FROM    TEACHING T1
          WHERE  T1.ProfId = P.Id AND T1.Semester = 'F1999' )
```

这里，变量T1的作用域被限制在子查询中。相反，变量P在外层查询和内层查询中都是可见的。这个变量是内层查询的参数，并把它的结果和外层查询的元组相关。也就是说，对于每个P.Id值，独立计算内层查询就好像P.Id是常数一样。每次计算内层查询时，将根据返回的结果检查R.CrsCode的值。如果这个值属于这个结果，则外层的SELECT查询就形成输出元组。

注意，内层查询必须对PROFESSOR的每一行重新计算。这是和不相关的嵌套查询相反的，也解释了与查询相关有关的代价。

即使嵌套查询对于查询优化器来说是一种挑战，但没有经验的数据库程序员有时会滥用它们，不使用比较简单的AND、NOT等而是用嵌套查询替代它们。这里有一个例子，说明如何不写前面的查询（即使它在语义上也是正确的）而使用可以完成相同功能的另一个查询:

```
SELECT   R.StudId,  T.ProfId,  R.CrsCode
FROM     TRANSCRIPT R, TEACHING T
WHERE    R.CrsCode  IN
         -- Courses taught by T.ProfId in F1999
         (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 = 'F1999' ) )
```

2. EXISTS运算符

检查嵌套子查询是否返回空结果是十分必要的。例如，我们想找出所有没有选修过计算机课程的学生。解决这个问题的一种途径是计算一个学生选修的所有计算机课程的集合，然后只列出这个集合为空的那些学生。这可以借助于相关嵌套子查询和EXISTS运算符来完成，如果集合非空就返回真值。

下面是这个查询的SQL表示:

```
SELECT   S.Id
FROM     STUDENT S
WHERE    NOT EXISTS (
         -- All CS courses taken by S.Id
         SELECT   T.CrsCode                                    (6.23)
         FROM     TRANSCRIPT T
         WHERE    T.CrsCode LIKE 'CS%'
                  AND  T.StudId = S.Id )
```

相对于内层子查询来说变量S是全局的。对于S.Id的每个值（在计算过程中被视为常数）计算这个子查询。内层查询没有结果的所有S.Id值构成外层查询的结果。

3. 表达除法运算符

我们现在来说明嵌套查询有助于表达关系除法运算符。具体来说，考虑查询列出选修过所有计算机课程的所有学生。我们可以通过首先计算一门计算机课程对应一行的单属性关系（这是除法运算符的被除数）来解决这个问题。然后，对于每个学生，我们检查这个学生的成绩单是否包含了所有这些课程。

为了使思路更加清晰，我们首先用虚拟的SQL（一种与SQL不同的不存在的语言，因为它有一个额外的集合运算符CONTAINS）来实现我们的计划。

```
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%' )
```

这个过程看起来很简单，只是现实的SQL缺少CONTAINS。然而，因为*A* CONTAINS *B*等价于NOT EXISTS (*B* EXCEPT *A*)，所以这个问题是能够解决的。但是上面查询的真正的SQL表示看起来就复杂得多了。

```
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 ) )
```

下面是一个更复杂的查询:

找出选修过计算机科学系每个教授教过的课程的所有学生。

产生这个查询结果的SQL语句是:

```
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                                                    (6.24)
    (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) )
```

变量S是全局的，每次计算子查询的时候，S的值是固定的。另一方面，R的范围是TRANSCRIPT的所有元组。因此，R.Semester和R.CrsCode的值取自学生S.Id的所有注册记录，T.ProfId的取值范围是所有教过S.Id的所有教授。

4. 集合比较运算符

假设STUDENT关系有另外一个数值属性GPA。我们可以提出这样的查询：大学里是否有一个学生的GPA高于所有大三的学生。

嵌套查询在解决这个问题时也很有用。

```
SELECT   S.Name, S.Id
FROM     STUDENT S
WHERE S.GPA >ALL  (SELECT  S.GPA                              (6.25)
                   FROM    STUDENT S
                   WHERE   S.Status ='junior')
```

这里“> ALL”是比较运算符，如果它左边的参数大于它右边的集合中的每个元素的时候，它返回真值。如果我们把它替换成“>=ANY”，我们得到一个查询，它检索出GPA高于或等于某个大三学生的学生。

关于这个查询还有一点需要注意：在外层查询和内层查询中都声明了变量S。那么内层查询的WHERE子句中提到的是哪一个S？答案是就像begin/end块一样，内层的声明在内层查询中是有效的（然而，这样的声明是不好的编程习惯）。

5. FROM子句中的嵌套子查询

即使查询（6.24）并不是很复杂，但SQL也有藏而待用的方法：你可以在FROM子句中使用嵌套子查询！它是按如下方式工作的：你编写一个嵌套子查询（它不能是相关的，因此不能用全局变量）。这个子查询可以用在FROM子句中，就好像它是关系名一样。你可以用关键字AS来给它附加一个元组变量。（实际上在这里，AS是可选的，但是为了提高可读性还是建议使用它。）

举个例子，我们可以表达类似于（6.24）的查询，但是不用NOT EXISTS（也就是说，答案包含了至少没有被一个计算机系教授教过的所有学生），它是通过把第一个嵌套子查询移进FROM子句来实现的（不能移动第二个子查询，因为它是相关的）。

```
SELECT   S.Id
FROM     STUDENT S,
         (SELECT  P.Id -- All CS professors
          FROM PROFESSOR P
          WHERE P.Dept = 'CS') AS C
WHERE C.ProfId NOT IN                                          (6.26)
     (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 )
```

应该尽可能避免在FROM子句中使用嵌套子查询，因为它可能产生作者自己也难以理解和检验的查询。一个更好的替代方法是使用视图机制，这将在6.2.6节中讨论。

除了嵌套查询外，SQL也允许在FROM子句中出现显式的表联结。然而，我们不讨论这个特点。

6.2.4 数据的聚合

在许多情况下，需要计算平均工资、最高GPA、每个部门的员工人数、总的购买金额等等。可以借助运算在元组集合上的所谓的**聚合函数**（aggregate function）完成这些任务。SQL使用图6-7所示的5个聚合函数。

聚合函数不能用纯粹的关系代数来表示。然而，可以扩展代数来使用它们（这些扩展超过了本书的讨论范围）。

为了说明聚合函数的使用，假设STUDENT和PROFESSOR关系都有属性Age，STUDENT关系还有属性GPA。我们先来看一些简单的例子。

```
-- 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'
```

COUNT ([DISTINCT] Attr)	统计查询结果Attr列中值的数目。可选的关键字DISTINCT指明每个值即使在不同的结果元组中出现多次，也只能统计一次
SUM ([DISTINCT] Attr)	累加Attr列的值。DISTINCT意味着每个值不管在Attr列中出现多少次，也只能累加一次
AVG ([DISTINCT] Attr)	计算Attr列值的平均值。DISTINCT意味着每个值只能用一次
MAX (Attr)	计算Attr列的最大值。DISTINCT不用于这个函数，因为它对结果不产生影响
MIN (Attr)	计算Attr列的最小值。DISTINCT不用于这个函数

图6-7 SQL聚合函数

上面的查询只是找出平均年龄和最小年龄，而不是找出与这些年龄对应的人。如果我们要找出管理系最年轻的教授，我们可以用嵌套子查询。

```
-- 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'  )
```

查询（6.25）返回GPA最高的大三学生的名字和Id（前面的查询中没有使用聚合），也可以用MAX以另外一种方式编写这个查询:

```
SELECT   S.Name,S.StudId
FROM     STUDENT  S
WHERE    S.GPA >= (SELECT  MAX(S1.GPA)                    (6.27)
                   FROM STUDENT  S1
                   WHERE S1.Status = 'Junior')
```

在聚合函数中使用DISTINCT有时会在查询语义上产生细微的差别。例如，

```
SELECT   COUNT(P.Name)
FROM     PROFESSOR P                                     (6.28)
WHERE    P.DeptId = 'MGT'
```

返回管理系的教授人数。另一方面，

```
SELECT   COUNT(DISTINCT P.Name)
FROM     PROFESSOR P
WHERE    P.DeptId = 'MGT'
```

返回管理系不同姓名教授的人数，而这可能与教授的实际人数不相符。类似地，

```
SELECT   AVG(P.Age)
FROM     PROFESSOR P                                     (6.29)
WHERE    P.DeptId = 'MGT'
```

返回管理系教授的平均年龄。然而，如果我们在上面的SELECT子句中用AVG（DISTINCT

P.Age)，查询结果就是不同年龄的平均年龄，这在统计上是没有意义的。

我们已经看到利用聚合函数的一些优势，那么你也应该避免聚合函数的一些缺点。在SELECT列表中混合使用聚合函数和属性是没有意义的，比如

```
SELECT   COUNT(*), S.Id
FROM     STUDENT S
WHERE    S.Name  = 'JohnDoe'
```
(6.30)

这是因为聚合函数为与John Doe相应的行的整个集合产生单个值，而属性S.Id为每一行产生一个不同的值（注意：可能有好几个人叫John Doe)。然而，在一些情况下这样的关联借助于稍后将定义的GROUP BY结构可能会变得有意义。

尽管在SELECT子句中混合聚合函数和属性通常并不十分有用，但是具有多个聚合函数确实是有意义的。比如，

```
SELECT   COUNT(*), AVG(P.Age)
FROM     STUDENT S
WHERE    S.Name = 'JohnDoe'
```
(6.31)

统计了在学生关系中叫John Doe的人数，也计算了他们的平均年龄。最后，你可能很想把语句（6.27）重写为

```
SELECT   S.Name,S.StudId
FROM     STUDENT  S
WHERE    S.GPA >= (MAX(SELECT  S1.GPA
                       FROM    STUDENT S1
                       WHERE   S1.Status = 'junior'))
```

但是你不敢这样做，因为它是一个无效的构造器。在WHERE子句中不允许使用聚合函数。

聚合和分组

到现在为止，我们知道了如何统计管理系教授的人数。但是如果我们想要知道大学里每个系的教授的人数，该怎么办呢？当然，我们可以为每个系构造类似于（6.28）的查询。除了WHERE条件中的MGT将被其他系的代码替换以外，每个查询都将是一样的。显然，这并不是可行的解决方法，因为即使在中等规模的大学里，系的数目也能达到几十个。另外，每次成立新系的时候，我们要构造新的查询，当系改名的时候，我们要做乏味的维护工作。

更好的解决方法是使用GROUP BY子句，它可以作为SELECT语句的一个组成部分。这个子句允许程序员把行的集合分成若干组，同一组中的所有行在某个特定的列的子集上的值是一致的。对组应用聚合函数，如图6-8所示每一组产生一行。例如，如果我们基于列StudId把图4-5所示的关系TRANSCRIPT的实例分组，结果产生了五个组。在任何一个指定的组中，所有行在StudId列上具有相同的值，但是在其他列上的值可能是不同的。

例如，下面的查询

```
SELECT   T.StudId, COUNT(*) AS  NumCrs,
         AVG(T.Grade)  AS  CrsAvg
FROM     TRANSCRIPT T
GROUP BY T.StudId
```

产生表

TRANSCRIPT	StudId	NumCrs	CrsAvg
	666666666	3	3.33
	987654321	2	2.5
	123454321	3	3.33
	023456789	2	3.5
	111111111	3	3.33

这里给出了如何得到每个系的教授人数和他们的平均年龄的方法。

```
SELECT   P.DeptId, COUNT(P.Name) AS  DeptSize,
         AVG(P.Age)  AS  AvgAge
FROM     PROFESSOR P
GROUP BY P.DeptId
```

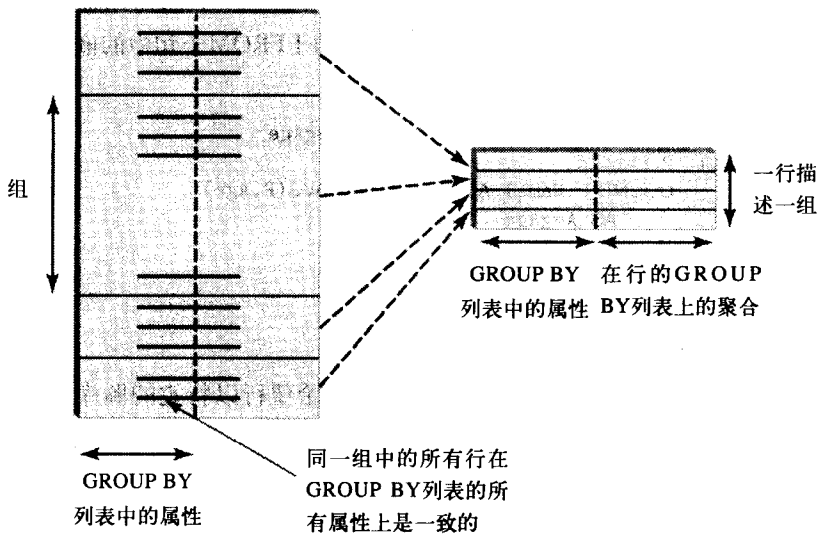


图6-8 GROUP BY子句的作用

在这两个查询中要注意的一个重要的问题是，SELECT子句中的每一列必须在GROUP BY子句中指定，或者是聚合函数的结果。

HAVING子句是与GROUP BY联合使用的。它使程序员可以指定一个条件来限制哪些组（在GROUP BY子句中指定）将被考虑作为最后的查询结果。在应用聚合之前，要去除不满足条件的组。假设我们想知道每个系教授的人数和教授的平均年龄，与前面的查询相似，但是这次我们只选取教授人数超过10人的系。这可以利用以下程序完成:

```
SELECT   P.DeptId, COUNT(P.Name) AS  DeptSize,
         AVG( P.Age)  AS  AvgAge
FROM     PROFESSOR P
GROUP BY P.DeptId
HAVING   COUNT(*) > 10
```

HAVING条件（不像WHERE条件！）在组上应用，而不是在单个的元组上应用。所以，对于由GROUP BY子句产生的每一组来说，COUNT（*）统计元组的数目。只有那些统计值

超过10的组才会做进一步处理。最后，应用聚合函数到每一组以便为每一组产生一个元组。

注意，上面的查询使用AS来给由聚合函数产生的列命名。另外，*和出现在HAVING子句中的COUNT函数一起使用。*和聚合函数可以方便地结合在一起使用，它可以在SELECT列表和HAVING子句中使用。然而，应该注意的是，上面的例子还有另外几种可供选择的方法：我们可以使用P.Name，甚至P.DeptId来代替*，因为不显式要求（DISTINCT）的话，SQL是不会消除重复的。

如果我们想基于1997～1998学年的成绩考虑系主任列表的候选人，那么可以使用下列程序：

```
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
```

通常，使用HAVING子句只是语法习惯而已，借助于FROM子句中的嵌套查询也可以产生同样的结果。

```
SELECT   Stats.DeptId, Stats.DeptSize, Stats.AvgAge
FROM     (SELECT  P.DeptId,
                  COUNT(P.Name) AS DeptSize, AVG(P.Age)
                      AS AvgAge
          FROM    PROFESSOR P
          GROUP BY P.DeptId) AS Stats
WHERE    Stats.ProfCount > 10
```

然而，应该避免在FROM子句中使用嵌套查询。

最后，通常不必指定查询结果中行的顺序。如果希望行以特定的顺序排列，就可以使用ORDER BY子句。例如，如果我们在产生系主任列表的SELECT语句中包含子句

```
ORDER BY CrsAvg
```

查询结果的行将按学生平均成绩的升序排列。这个子句通常以查询结果的列名列表作为参数。根据列表中指定的第一列有序地输出行。如果多个行在那个列上有相同的值，则列表中指定的第二列就用来决定顺序，依此类推。例如，如果我们想在输出系主任列表的候选人时首先以平均成绩为序，然后再以学号为序，我们可以使用下面的SELECT语句：

```
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
```

ORDER BY子句中指定的属性必须是查询结果中的列名。因此，我们在这个例子中把第二个元素称为StudId（而不是T.StudId），因为默认情况下，该名称就是查询结果中的列名。类似地，不在SELECT子句中引入列的别名CrsAvg，就不能以平均成绩为主序来给行排序，因为若没有别名这一列就没有名字了[⊖]。

⊖ 结果表的列也可以通过它们的次序位置来引用，但是通常不鼓励使用这种方式。

默认情况下使用升序，但是也可以指定降序。如果在上面的例子中我们把ORDER BY子句替换为

```
ORDER BY DESC CrsAvg, ASC StudId
```

则查询结果的行将会按照平均成绩的降序排列，对于平均成绩相同的学生，则按学号的升序排列。

6.2.5 简单查询计算算法

带有聚合和分组的查询计算的总体过程如图6-9所示。

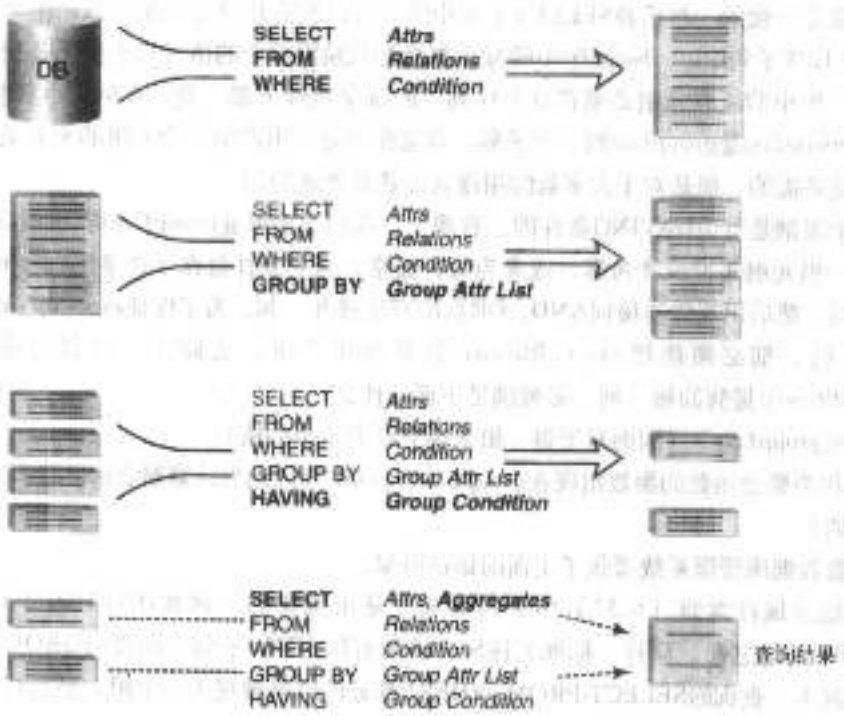


图6-9 带有聚合函数的查询计算

第一步，计算FROM子句。它产生一个表，这个表是作为参数列出的那些表的笛卡儿积。

第二步，计算WHERE子句。它对第一步中产生的表的每一行逐行单独处理。行的属性值将替换条件中的属性名，然后计算这个条件。由WHERE子句产生的表只包含那些条件计算为真的行。

第三步，计算GROUP BY子句。它把第二步中产生的表划分成元组的分组，每一组只包含在组属性序列表的所有属性上都一致的那些元组。

第四步，计算HAVING子句。它去除第三步产生的组中没能满足组条件的那些组。

第五步，计算SELECT子句。它对第四步产生的每一组计算目标序列表中的聚合函数，保留作为SELECT子句参数列出的那些列，并为每一组产生一行。

第六步，计算ORDER BY子句。它利用指定的列的列表为第五步中产生的行排序。SELECT语句输出结果表。

对GROUP BY和HAVING的限制

分组要遵守一些规则不同的是其中的一些限制实际上是很有意义的!

考虑带有聚合的SQL查询的通用形式:

```
SELECT    attributeList, aggregates
FROM      relationList
WHERE     whereCondition                                    (6.32)
GROUP BY  groupList
HAVING    groupCondition
```

GROUP BY子句的目的是把查询结果划分成组，同一组中的所有元组在*groupList*中的每个属性上都是一致的。然后将SELECT子句中的聚合函数应用到每一组，以产生单个的元组。因为对SELECT子句的*attributeList*中的每个属性都只输出单个的值（那个值不是聚合值），所以在任意一组中的所有元组必须在这个序列中的每个属性上都一致。在SQL中具有这个性质就要求*attributeList*是*groupList*的一个子集，这是保证每一组产生一个元组的充分条件。(这个条件并不是必需的，但是对于大多数的用途来说是很普通的。)

第二个限制是有关HAVING条件的。直观上，我们需要保证*groupCondition*对GROUP BY指定的每一组元组来说或者为真，或者为假。通常，这个条件包含了许多形如$expr_1$ op $expr_2$的原子比较，然后用逻辑连接词AND、OR和NOT连接在一起。为了保证$expr_1$ op $expr_2$有意义，对于每一组，都必须能把$expr_1$和$expr_2$计算为单个值。实际上，这就意味着对于在*groupCondition*中提到的每一列，必须满足下面条件之一:

1) 它在*groupList*中（因而对于每一组来说它都有单个的值)。

2) 它作为聚合函数的参数出现在*groupCondition*中（因而当计算聚合的时候，表达式将看到单个的值)。

大多数数据库管理系统增强了上面的语法限制。

我们也应该注意到（6.32）中子句的顺序是很重要的。例如HAVING子句不能放在GROUP BY子句之前。同样，标准允许SQL语句有HAVING子句，而没有GROUP BY子句。在这种情况下，查询的SELECT-FROM-WHERE部分的结果被视为一个组，然后将HAVING中的*groupCondition*应用到那一组。

6.2.6 再论SQL中的视图

到目前为止，我们所讨论的关系在技术上通常是指**基本关系**（base relation)。它们是“正常的”数据库关系。基本关系的内容物理上是存储在磁盘上的，独立于数据库中其他关系的内容。

在第4章我们已经讨论过，视图是一个关系，但是它的内容通常不是*物理存储*在数据库中，而是定义为SELECT语句的查询结果。每次使用视图的时候，就用相关的查询来计算其内容。因此，存储（在系统目录中）的是视图的定义（即决定这个视图内容的查询)，而不是视图的内容。

因为每个视图都是查询执行的结果，它的内容依赖于引用视图时数据库中基本关系的内容。

查询语言中视图的角色类似于传统编程语言中*子例程*（subroutine)的角色。视图通常表

示某些有意义的查询，它用于其他一些经常提交的查询，或者它有独立的作用。不管是哪种情况，都应该把这个查询抽象出来，并“假装”数据库中包含了一个关系，其内容就是查询的结果。

视图机制的另一个重要的作用就是控制用户对数据的访问。这个功能在4.1节和4.3节中简要介绍过，我们将在本节的后面部分更详细地讨论它。

1. *在查询中使用视图*

一旦定义了视图，就可以像任何其他表一样在SQL查询中使用它了。当查询中用到视图时，它的定义就自动替换到FROM子句中，如图6-10所示。

```
SELECT   ... ... ...                          用到视图VIEW1
FROM     VIEW1 V                              的查询
WHERE    ... AND V.Attr = 'abc' AND ...

         变成

SELECT   ... ... ...                          被视图定义修改
FROM     (definition of VIEW1) V              的查询
WHERE    ... AND V.Attr = 'abc' AND...
```

图6-10 使用视图的查询修改过程

假设想要找出大学里教授的平均年龄最低的系，应用程序设计者可能决定除了用上面的查询以外，还可以用许多其他的查询来计算平均年龄。在查询处理中，就像编程语言那样，这是创建计算平均年龄视图的很好的理由。

```
CREATE VIEW AVGDEPTAGE(Dept,AvgAge) AS
    SELECT      P.DeptId, AVG(P.Age)
    FROM        PROFESSOR P
    GROUP BY    P.Dept
```

这个视图就像子例程一样允许我们单独地解决一个较大问题的一个部分。例如，我们现在可以用以下代码找出教授平均年龄最小的系：

```
SELECT   A.Dept
FROM     AVGDEPTAGE A
WHERE    A.AvgAge =    (SELECT MIN(A.AvgAge)
                        FROM AVGDEPTAGE A )
```
(6.33)

视图可以使复杂的SQL查询变得容易理解。（并被调试！）考虑查询（6.24），它要找出选修过计算机科学系每个教授所讲授过的课程的所有学生。查询用了两个嵌套子查询，其中一个子查询和外层查询相关。因为嵌套问题和相关问题微妙地交织在一起，很难第一次就构造出正确的查询。

我们借助视图就可以简化这项工作。视图

```
CREATE VIEW ALLCSPROFIDS(ProfId) AS
    SELECT   P.Id
    FROM     PROFESSOR P
    WHERE    P.Dept = 'CS'
```

构造了所有计算机科学系教授的Id的集合。下一步，我们定义视图来表示（6.24）中的第二个

相关子查询。这个子查询的目标列表只有一个属性ProfId，但它还是用到了全局变量S，S是查询返回的结果集的参数。每次执行查询都返回教过某个学生的所有教授的集合。因为SQL不允许创建全局变量作为参数的视图，我们临时把这个变量的属性包括在目标列表中。更精确地说，我们只包括在子查询中实际用到的全局变量的属性。在我们的例子中，全局变量是S，额外的属性是StudId。结果就是我们熟悉的（4.5）中定义的视图PROFSTUD。

但是，我们还不能从ALLCSPROFID中把PROFSTUD减掉，因为它们不是并相容的。另外，SQL只允许将EXCEPT运算符应用到子查询的结果上，即我们不能简单地从一个表中减去另一个表。因此，我们仍然必须要使用嵌套子查询。然而，现在的子查询比（6.24）中的子查询更易管理，因为视图使我们能把一个复杂的问题分解为几个小的任务。

```
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))
```

尽管CREATE VIEW语句与CREATE TABLE语句在用于基本关系的时候不大相同，但是从系统目录中删除视图和表时却用了类似的语句：对于视图使用DROP VIEW，对于基本表则使用DROP TABLE。

如果我们想删除一个视图或者基本表，但是数据库中有（其他的）视图是通过这个视图或这张表定义的，此时该怎么办呢？现在的问题是删除这样的视图或表意味着所有通过它们定义的视图都将要“被遗弃”，也就没有办法再使用它们了。在这样的情况下，SQL把决定权留给了DROP语句的设计者。这个语句的一般形式为

DROP {TABLE | VIEW } *table or view* {RESTRICT | CASCADE}

如果使用RESTRICT选项，那么若有某个视图依赖于将要被删除的表或视图的时候，删除操作将会失败。如果使用CASCADE选项，所有依赖视图也都将被删除。

2. 物化视图

如果许多查询都用到一个视图，那么视图的内容就可以存储在高速缓存中。高速缓存的视图常常称为**物化视图**（materialized view）。视图缓存能够在很大程度上改善根据这些视图定义的查询的响应时间，但是更新事务时就要付出代价了。如果一个视图依赖于一张基本表，一个事务更新这张基本表，那么视图缓存也需要更新。

考虑视图（4.5），假设一个事务向TRANSCRIPT中增加元组<023456789, CS315, S1997, B>。这个元组与TEACHING中的元组<101202303, CS315, S1997>进行联结以产生一个视图元组<101202303, 023456789>。但是后一个元组已经在视图（见图4-10）中了，所以这个更新不会改变这个视图。如果我们向TRANSCRIPT中增加元组<023456789, MGT123, F1997, A>，这个视图将得到新的元组<783432188, 023456789>和<009406321, 023456789>。

现在考虑若从定义了物化视图的基本关系中删除元组时会发生什么呢？如果一个事务从TRANSCRIPT中删除了元组<123454321, CS305, S1996, A>，有人可能会认为元组<101202303,

123454321>也应该从视图缓存中删除。但实际上并不是这样，因为<101202303, 123454321>仍然可以通过联结<123454321, CS315, S1997, A>和<101202303, CS315, S1997>来得到。观察图4-10指出元组<101202303, 123454321>在视图中的两个理由，删除<123454321, CS305, S1996, A>只去除了其中一个理由！然而，如果事务从TRANSCRIPT中删除<123454321, MAT123, S1996, C>，则应该从视图缓存中删除元组<900120450, 123454321>。

商用数据库管理系统并不为物化视图的自动维护提供直接的支持，但是从视图缓存中自动删除元组或往视图缓存中自动插入元组可以使用触发器来实现（参见第9章）。即使这样，触发器只是为维护视图缓存提供了基本的方法。就像我们刚刚所说的，维护视图缓存是算法上非平凡的任务，特别是在对效率要求比较高的时候。我们在本书中不再深入讨论这些问题。相关文献中已经提出过许多算法，比如[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]。对这个问题，现在仍然进行着较多的研究，因为物化视图对于数据仓库来说显得日益重要。

数据仓库（data warehouse）是（非频繁更新的）数据库，通常由存储在单独的产品数据库中的数据的物化视图组成。它们通常用于在线分析处理（OLAP），OLAP在第1章简要讨论过并将在第19章中详细讨论。和大多数产品数据库相比，优化数据仓库是为了查询，而不是为了事务处理，它们是本章讨论的SQL高级查询功能的最大受益者。（在许多事务处理的应用程序中，快速响应时间和高吞吐量的要求使得不能使用复杂的查询。）

3. 访问控制和通过视图定制

数据库视图不仅可以用作一种子例程机制，还可以用作一种对数据访问进行控制的灵活工具。因此，我们可以允许某些用户来访问一个视图，而不是定义它的那些表。例如，可能不允许学生去查询PROFESSOR关系，因为该表存储着和SSN有关的信息。然而，可以让学生访问之前定义的AVGDEPTAGE视图。这样，尽管只有系统管理员才能访问PROFESSOR，但是可以允许学生查询视图AVGDEPTAGE:

```
GRANT SELECT ON AvgDeptAge TO ALL
```

注意通过使用视图，我们可以弥补GRANT语句的一个缺陷。在授予UPDATE（或INSERT）权限的时候，允许（可选的）修改指定的某些列（或者可以插入值），但是在授予SELECT权限的时候，SQL不能指定这样的列。然而，我们可以通过创建可访问列的视图并授予对那个视图（而不是基本表）的访问权限来达到相同的效果。

视图的创建者（也是所有者）不一定也是定义它的基本表的所有者。所必需的就是视图的创建者在定义视图所需的所有关系上具有SELECT权限。例如，如果PROFESSOR关系为*Administrator*所有，*Administrator*再给*Personnel*授予对PROFESSOR的SELECT权限，则*Personnel*就可以创建视图AVGDEPTAGE，而后提交上面的GRANT语句。

如果*Administrator*决定从*Personnel*那里收回SELECT权限，那将会怎么样呢？注意，如果*Administrator*收回了这个权限，则视图将会“被丢弃”，没有人可以再查询它了。实际的结果依赖于发布REVOKE语句的方式。如果系统管理员在REVOKE语句中使用了RESTRICT选项，收回操作将会失败。如果使用了CASCADE选项，就可以进行收回并且将会从系统目录中删除视图本身。

视图的另一个用途是进行定制，这是和访问控制共同作用的。一个实际的产品数据库可能包含几百个关系，每个关系又有几十个属性。然而，大多数的用户（包括“初级的”用户，以及应用程序开发者）只需要处理数据库模式的一小部分，也就是与由用户或应用程序执行的特定任务相关的那一部分。让所有用户去费劲地了解整个数据库模式是没有好处的。更好的策略是创建为不同的用户分类定制的视图，所以AVGDEPTAGE可能是为统计员定制的视图之一。这种方式有以下三个好处。

1) *便于使用和学习*。这可以加快应用程序开发，避免因为误解部分数据库模式而引起的错误。

2) *安全*。因为只授予各种用户和应用程序对特定视图的访问权限，因此减少了由于程序错误、人为错误和恶意行为所带来的可能的损失。

3) *数据逻辑独立性*（如第4章所讨论的）。如果以后需要改变数据库模式，这个优点将会节省大量的维护成本。假设模式的重构不会导致信息的损失，针对视图编写的应用程序就没必要修改了——唯一需要修改的是视图定义。

6.2.7 空值的窘境

在第4章中我们简要讨论了**空值**（null value）的概念。例如，如果我们用TRANSCRIPT表中的元组来代表过去选修过的课程或当前学期正在选修的课程，一些元组可能在Grade属性上没有有效的值。NULL就是SQL在这种情况下用到的占位符。

空值是查询处理中令人头痛（又是不可避免的）的问题。如果条件T.Grade = 'A'中T的值是在Grade属性上有空值的元组，那么该条件的真值是什么呢？

考虑到这个现象，SQL使用了所谓的*三值逻辑*，这里真值有true、false和unknown，当val_1或val_2中至少有一个值是NULL时，val_1 op val_2（op可以是<、>、< >、=等）就被认为是unknown。

空值不仅影响到WHERE子句和CHECK子句中的比较，还影响到算术表达式和聚合函数。当算术表达式遇到NULL时，它自身就被定值为NULL。COUNT把 NULL当作正常值（比如，语句（6.28）在计算管理系教授人数的时候把NULL和正常值一起统计）。所有其他的聚合则把空值丢掉（比如，语句（6.29）在计算平均值的时候忽略了Age列中的空值）。然而还是要注意，如果将这样的聚合函数应用到只有NULL的列上，那么结果还是空值。

但是，还有其他令人混淆的情况。当WHERE子句或CHECK子句形如$cond_1$ AND $cond_2$、$cond_1$ OR $cond_2$或NOT *cond*，并且子条件之一因为有NULL隐藏在其中而计算得到unknown时，我们也必须找到相应的解决方法。这个问题通过图6-11所示的真值表得以解决，它给出了依赖于子条件值的各种布尔函数的值。

这些表应该很清晰了。其基本思想是很简单的，假设我们需要计算true AND unknown的值。因为unknown可能是true，或者是false，所以整个表达式的值也就可能是true，或者是false（也就是unknown）。另一方面，不管unknown是true还是false，计算表达式false AND unknown和true OR unknown都会分别得到false和true。NOT的情况也可以做类似的处理。

SQL引入了另外一个谓词IS NULL，它用来测试某个值是否为NULL。例如，当赋予T的元组在Grade属性上有空值的时候，T.Grade IS NULL为真，否则为假。有趣的是，这是SQL

中唯一真正的二值谓词!

$cond_1$	$cond_2$	$cond_1$ AND $cond_2$	$cond_1$ OR $cond_2$
true	*true*	*true*	*true*
true	*false*	*false*	*true*
true	*unknown*	*unknown*	*true*
false	*true*	*false*	*true*
false	*false*	*false*	*false*
false	*unknown*	*false*	*unknown*
unknown	*true*	*unknown*	*true*
unknown	*false*	*false*	*unknown*
unknown	*unknown*	*unknown*	*unknown*

cond	NOT *cond*
true	*false*
false	*true*
unknown	*unknown*

图6-11 SQL用来处理unknown的真值表

那么当整个条件计算得到unknown时，将会怎么样呢？答案要取决于这是一个WHERE子句还是CHECK子句。如果WHERE子句计算得到unknown，就被视为false，相应的元组也不能加入查询结果。如果CHECK子句计算得到unknown，就要考虑遵守完整性约束了（也就是结果被视为true）。可以给在查询和约束中对unknown值的不同处理方法一个合理的解释。确实，我们假设用户希望查询只返回确定是真实的答案。另一方面，形如CHECK（condition）的约束被视为条件不应该计算得到false的语句。因此，unknown真值是可以接受的。

我们只是提出了SQL中空值的大体框架。许多细节还没有涉及，但是可以在大部分标准的SQL参考书中找到这些内容，像[Date and Darwen 1997]。例如，空值对于LIKE条件、IN条件、集合比较（如“> ALL”）、EXISTS特性、消除重复（如用到DISTINCT的查询）等等会产生怎样的影响？考虑在这些情况下怎样做才是合理的，然后把你的结论和[Date and Darwen 1997]中的结论进行比较。

6.3 在SQL中修改关系实例

到目前为止，我们已经讨论了查询子语言——SQL中最难的一部分。然而，数据库的作用不仅是为了查询，还要加入和修改适当的数据。本节主要讨论用于数据插入、删除和修改的那部分SQL。

1. *增加数据*

INSERT语句有几种形式，最简单的一种就是程序员只需指定要插入的元组。这个语句的第二个版本可以插入多个元组，并可以用查询来指定要插入的元组。要插入单个元组，程序员只要编写以下语句:

```
INSERT INTO   PROFESSOR(DeptId,Id,Name)
VALUES ('MATH','100100100','Bob Parker')
```

INTO子句中的属性的次序不一定是默认的次序（即它们在CREATE TABLE语句中的次序）。然而，如果你知道默认的属性次序，就可以省略INTO子句中的属性列表。当然，此时VALUE子句中项的次序必须与默认的属性次序相对应。尽管这样可能会节省一些时间，但在INTO子句中省略属性列表是容易出错的（比如，考虑一下如果以后模式发生改变将会怎样），因此这

是不好的编程风格。

INSERT语句的第二种形式可以将大量的元组插入到关系中。将要插入的元组是INSERT语句中查询的结果。注意，即使用查询来定义将要插入到关系中的元组，这种机制也从根本上不同于通过查询来定义视图（见练习6.18）。

举个例子，我们将一些元组插入到HARDCLASS关系中。困难班级是指有10%以上的学生没有通过的班级。HARDCLASS关系的属性是课程号、学期和未通过率。

这个查询实际上非常复杂（因为在SQL中表达未通过率需要多思考一下），所以我们一步一步地来处理，首先定义两个视图。

```
--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
```

类似地，我们可以定义视图CLASSENROLLED，它统计选修每门课程的学生的人数。

```
--Number of enrolled students per class
CREATE VIEW CLASSENROLLED(CrsCode, Semester, Enrolled) AS
    SELECT   T.CrsCode, T.Semester, COUNT(*)
    FROM     TRANSCRIPT T
    GROUP BY T.CrsCode, T.Semester
```

现在可以用如下方法向HARDCLASS中添加元组：

```
INSERT INTO HARDCLASS(CrsCode, Semester, FailRate)
    SELECT   F.CrsCode, F.Semester, F.Failed/E.Enrolled
    FROM     CLASSFAILURES F, CLASSENROLLED E
    WHERE    F.CrsCode = E.CrsCode AND F.Semester = E.Semester
                 AND (F.Failed/E.Enrolled) > 0.1
```

(6.34)

最后一个查询看起来很简单，但是想象一下如果没有视图它将会多么复杂！

我们必须还要提到一些小问题。第一，如果INSERT语句插入一个违反某个完整性约束的元组，那么整个操作将会异常中止，并且不会插入元组（假设没有对约束检查执行DEFERRED操作，这将在10.3节中描述）。

第二，有可能在VALUE子句的值的列表中省略了一个值（和属性列表中相应的属性），并且在SELECT子句中省略了一个属性（如果CREATE TABLE语句没有指定NOT NULL）。如果CREATE TABLE语句为省略的属性指定了默认值，那么就用默认值。除了省略值以外，更好的办法是用信息性更强的关键字DEFAULT或NULL（任何一个都是合适的），来替代省略的值（如（100100100, NULL, DEFAULT））。

2. 删除数据

删除元组类似于INSERT的第二种形式，只是现在要用DELETE关键字。例如，删除在1991年秋季和春季教授过的困难的班级，我们可以使用以下语句：

```
DELETE FROM HARDCLASS
WHERE Semester IN ('S1991','F1991')
```

注意，DELETE语句不允许在FROM子句中使用元组变量。

假设为了改善教学水平，学校决定解雇某门课程里有过高未通过率的所有教授。这是很困难的（并不是因为有终生教授的原因！）。DELETE语句的FROM子句确实有着很强的限制形式：程序员只可以指定一个关系，就是将要删除其元组的那个关系。（从两个关系的笛卡儿积中删除行意味着什么呢？）然而，为了找出要解雇的教授，我们需要看一下关系HARDCLASS，但是在FROM子句中没有这个关系的位置了。那么我们将怎么做呢？使用嵌套子查询！

```
DELETE FROM PROFESSOR
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)
```
(6.35)

3. 更新已有的数据

有时必须改变关系中已有元组的某些属性的值。例如，下面的语句把学生666666666选修的课程EE101的成绩由B改为A:

```
UPDATE   TRANSCRIPT
SET      Grade ='A'
WHERE    StudId ='666666666' AND CrsCode = 'EE101'
```

注意UPDATE语句和DELETE一样也不允许使用元组变量，它的FROM子句形式是受限制的（更准确地说，UPDATE本身就是一种FROM子句）。结果，一些较复杂的更新就要用到嵌套查询。例如，如果我们决定把所有表现欠佳的教授调任到行政部门而不是解雇他们，我们使用类似于（6.35）的子查询。

```
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)
```
(6.36)

下面我们最喜欢做的事情：把所有管理员的工资提高10%:

```
UPDATE   EMPLOYEE
SET      Salary = Salary * 1.1
WHERE    Department = 'Adm'
```

4. 视图的更新

因为视图经常用作定制工具，对用户和程序员隐藏了概念数据库模式的复杂性，所以让程序员更新他们的视图是很自然的。但是这项工作说起来容易做起来难，因为存在下面三个问题:

1) 假设我们在TRANSCRIPT上有一个简单的视图——在属性CrsCode、StudId和Semester上的投影。如果程序员想在这样的视图中插入一个新的元组，则Grade属性的值将丢失。这个问题并不严重。如果创建基本关系的Create Table语句允许的话，我们可以用空值填补上这个丢失的属性；如果不允许，插入命令就会被拒绝。

2) 考虑PROFESSOR关系上的视图CSPROF，它是通过在DeptId = 'CS'上的简单选择来得到的。假设程序员要插入<121232343, 'Paul Schmidt', 'EE'>。如果我们把这个插入传播到基本关系(即PROFESSOR)，我们可以观察到这样的异常：在插入之后查询这个视图，却没有显示刚刚插入的元组的任何踪迹。实际上，Paul Schmidt并不是计算机科学系的教授，所以他不会出现在这个通过选择DeptId = 'CS'来定义的视图中。

SQL在默认情况下不会禁止这样的异常，但是细心的数据库设计者可能会加上子句WITH CHECK OPTION，来保证在视图中新插入或更新的元组确实满足视图的定义。在我们的例子中，我们可以编写以下语句：

```
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) 这个问题更加复杂了。某些视图更新可能会转换为多个对定义它的基本关系的更新。这是一个潜在的非常严重的问题，因为由单个视图更新而产生的多种可能性对存储的数据来说可能产生完全不同的结果。

为了说明第三个问题，考虑之前讨论的视图PROFSTUD。

```
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
```

这个视图的内容在图4-10中加以描述。假设我们现在决定从视图中删除元组<101202303, 123454321>。这个更新应该如何传播回基本关系呢？这里有四种可能性。

1) 从关系TEACHING中删除<101202303, CS315, S1997>和<101202303, CS305, S1996>。

2) 从关系TRANSCRIPT中删除<123454321, CS315, S1997, A>和<123454321, CS305, S1996, A>。

3) 从TEACHING中删除<101202303, CS315, S1997>，并从关系TRANSCRIPT中删除<123454321, CS305, S1996, A>。

4) 从TEACHING中删除<101202303, CS305, S1996>，并从关系TRANSCRIPT中删除<123454321, CS315, S1997, A>。

对于这每一种可能性，由视图隐含的联结不包含元组<101202303, 123454321>。唯一的问题是应该将哪一种可能性应用到视图更新上？

这个例子说明，在缺乏额外信息的情况下，有时不能把视图更新翻译成对基本关系的更新。在为了消除视图更新的歧义而定义启发式方法方面，已经做了大量的工作，但是没有一个方法能够成为可接受的解决方案。SQL采用了最简单的方法，只允许更新一类有极强限制的视图。在视图定义上的这些限制的实质总结如下。

1) 在FROM子句中只能提到一个表（并且只能提到一次），FROM子句不能有嵌套子查询。

2) 不允许有聚合、GROUP BY子句、HAVING子句或集合运算。

3) 视图的WHERE子句中的嵌套子查询不能引用视图定义的FROM子句中用到的表。嵌套子查询既不能显式地在FROM子句中引用这个表，也不能隐式地通过定义在外层查询中的变

量来引用这个表。

4) 不允许在SELECT子句中有表达式或DISTINCT关键字。

满足这些条件（和其他一些很模糊的限制）的视图称为**可更新的**（updatable）视图（在SQL意义上）。这里是可更新视图的例子。

```
CREATE VIEW CANTEACH(Professor, Course)
    SELECT   T.ProfId, T.CrsCode
    FROM     TEACHING  T
```

参考图4-5的数据库，假设我们从视图CANTEACH中删除<09406321, MGT123>。在基本关系（TEACHING）中有两个元组会引起疑问：<09406321, MGT123, F1994>和<09406321, MGT123, F1997>。因此视图更新到TEACHING更新的翻译必须要删除这两个元组。

6.4 参考书目

Codd在学术论文[Codd 1972, 1970]中引入了关系代数。SQL是由IBM的System R研究组[Astrahan et al. 1981]开发的。[Melton and Simon 1992, Date and Darwen 1997]是SQL-92的参考书，[Gulutzan and Pelzer 1999] 描述了SQL:1999提供的扩展。

过去，视图更新问题引起了相当大的关注。下面是提出各种解决方案的部分著作：[Bancihon and Spyratos 1981, Masunaga 1984, Cosmadakis and Papadimitriou 1983, Gottlob et al.1988, Keller 1985, Langerak 1990, Chen et al. 1995]。物化视图的维护问题也是极受关注的研究领域[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]。因为在数据仓库中物化视图很重要，因而进行了较多的研究。

6.5 练习

6.1 假设**R**和**S**是关系，分别包含n_R和n_S个元组。下面每个表达式的结果中可能会有的元组的最大数目和最小数目是多少（假设**R**和**S**并相容）：

a. $\mathbf{R} \cup \mathbf{S}$

b. $\mathbf{R} \cap \mathbf{S}$

c. $\mathbf{R} - \mathbf{S}$

d. $\mathbf{R} \times \mathbf{S}$

e. $\mathbf{R} \bowtie \mathbf{S}$

f. $\mathbf{R}/\mathbf{S}$

g. $\sigma_{s=4}(\mathbf{R}) \times \pi_{s,t}(\mathbf{S})$

6.2 假设**R**和**S**是表示前面练习中关系的表，设计返回上面练习的每个表达式的结果的SQL查询。

6.3 验证笛卡儿积是可结合的操作符，即

$$\mathbf{r} \times (\mathbf{s} \times \mathbf{t}) = (\mathbf{r} \times \mathbf{s}) \times \mathbf{t}$$

对于所有的关系**r**、**s**和**t**都成立。

6.4 验证选择是可交换的，即对于任何关系r和任何一对选择条件$cond_1$和$cond_2$，有

$$\sigma_{cond_1}(\sigma_{cond_2}(\mathbf{r})) = \sigma_{cond_2}(\sigma_{cond_1}(\mathbf{r}))$$

6.5 验证对于任何一对关系**r**和**s**，如果选择条件*cond*只涉及到在关系**s**的模式中提到的属性，那么

$$\sigma_{cond}(\mathbf{r} \times \mathbf{s}) = \mathbf{r} \times \sigma_{cond}(\mathbf{s})$$

6.6 证明：如果**r**和**s**是并相容的，那么$\mathbf{r} \cap \mathbf{s} = \mathbf{r} \bowtie \mathbf{s}$。

6.7 用除法编写一个关系代数表达式，它产生所有选修了每个学期开设的所有课程的学生（这表明他们可能两次选修了同一门课程）。

6.8 用除法编写一个关系代数表达式，它产生所有选修了开设过的所有课程的学生。（如果一门课程开设了多次，那么他们必须至少选修一次。）

6.9 两个关系**r**和**s**的**外联结**（联结条件为*cond*）表示为$\mathbf{r} \overset{outer}{\underset{cond}{\bowtie}} \mathbf{s}$，其定义如下。就像以前一样，它是包含**R**（**r**的模式）和**S**（**s**的模式）的（可能进行了重命名）属性的并的模式上的关系。然而，$\mathbf{r} \overset{outer}{\underset{cond}{\bowtie}} \mathbf{s}$中的元组分为三类：（1）在**r**和**s**正常联结中出现的元组。（2）**r**的不与**s**中任何元组联结的元组。因为这些元组没有值对应于来自**S**的属性，所以它们在这些属性上被赋予NULL。（3）**s**的不与**r**中任何元组联结的元组。同样，这些元组在**R**的属性上被赋予NULL值。

构造一个关系代数查询，它产生和$\mathbf{r} \overset{outer}{\underset{cond}{\bowtie}} \mathbf{s}$相同的结果，要求只使用并、差、笛卡儿积、普通的联结（不是外联结）这些操作符。你也可以使用常数关系，即不依赖于数据库其他部分的具有固定内容的关系。

6.10 使用图4-4的学生注册系统模式，以关系代数和SQL的形式来表达下面的每个查询。

a. 列出由属于EE或者MGT系的教授讲授的所有课程。

b. 列出所有在1997年春季和1998年秋季都选修过课程的学生的名字。

c. 列出所有选修过至少两门来自不同系的教授讲授的课程的学生的名字。

d. 列出MGT系中被所有学生选修的所有课程。

*e. 找出符合下列要求的系：有一位教授教过那个系开设过的所有课程。

6.11 用关系代数来找出所有“可能”的班级，其不通过率高于20%。

因为关系代数没有聚合操作符，所以我们必须增加这样的操作符才能解决上面的问题。你应该使用的额外操作符是$count_{A/B}(\mathbf{r})$。

这个操作符的含义如下：*A*和*B*必须是**r**中属性的列表。$count_{A/B}(\mathbf{r})$的模式包含*B*的所有属性和一个额外的属性（表示计数值）。$count_{A/B}(\mathbf{r})$的内容定义如下：对于每个元组$t \in \pi_B(\mathbf{r})$（在*B*上的投影），进行$\pi_A(\sigma_{B=t}(\mathbf{r}))$，然后对结果关系中的元组的数目进行计数（$\sigma_{B=t}(\mathbf{r})$表示**r**中所有元组的集合，其在*B*的属性上的值为**t**）。我们把这个数目表示为$c(t)$。

关系$count_{A/B}(\mathbf{r})$定义为$\{<t, c(t)> \mid t \in \pi_B(\mathbf{r})\}$。

你应该能够看出上面的构造就是SQL中的GROUP BY到关系代数的直接转换。

6.12 阐述下面代数表达式的含义（其中一些查询可能在大学中产生空结果，但是这偏离了主题）：

a. $\pi_{\text{CrsCode, Semester}}$ (TRANSCRIPT) / π_{CrsCode} (TRANSCRIPT)

b. $\pi_{\text{CrsCode, Semester}}$ (TRANSCRIPT) / π_{Semester} (TRANSCRIPT)

c. $\pi_{\text{CrsCode, StudId}}$ (TRANSCRIPT) / (π_{Id} (STUDENT))[StudId]

d. $\pi_{\text{CrsCode, Semester, StudId}}$ (TRANSCRIPT) / (π_{Id} (STUDENT))[StudId]

6.13 考虑下面的查询

```
SELECT   S.Name
FROM     STUDENT S, TRANSCRIPT T
WHERE    S.Id = T.StudId
         AND T.CrsCode IN ('CS305','CS315' )
```

这个查询意味着什么（用一个简短的句子来表达其含义）？写出一个等价的SQL查询，不要使用IN操作符和集合构造。

6.14 写出查询（6.33），不要使用视图。

6.15 用SQL来表达下面的查询。假设给STUDENT表增加一个额外的属性Age，PROFESSOR表有额外的属性Age和Salary。

a. 找出在某门课程中得过A的学生的平均年龄。

b. 找出每门课程的成绩最优秀的学生的最小年龄。

c. 找出每门课程的成绩最优秀的学生（并且他还选修过CS305或者MAT123）的最小年龄。（提示：使用HAVING子句会有帮助。）

d. 把年龄低于40并且在1997年春季或者1997年秋季教过MAT123的每个教授的工资提高10%。（提示：试着在UPDATE语句的WHERE子句中使用嵌套子查询。）

e. 找出工资比所有教授的平均工资高至少10%的教授。（提示：使用视图，就像（6.34）所示的HARDCLASS例子那样。）

f. 找出工资比其所在系的所有教授的平均工资高至少10%的教授。（提示：使用视图，就像在（6.34）中那样。）

6.16 用关系代数表达下面的查询。

a.（6.14）

b.（6.18）

c.（6.21）

6.17 考虑下面的模式：

TRANSCRIPT (StudId, CrsCode, Semester, Grade)

TEACHING (ProfId, CrsCode, Semester)

PROFESSOR (Id, ProfName, Dept)

用关系代数和SQL写出下面的查询：找出所有选修过MUS系的每个教授教的一门课程的学生的Id。

6.18 把上面的查询定义为SQL视图，然后使用这个视图来回答下面的查询：对于选修过MUS系的每个教授所教的一门课程的学生，如果他选修的课程的数目大于10，那么显示出他所选修的课程数目。

6.19 考虑下面的模式：

BROKER (Id, Name) ACCOUNT (Acct#, BrokerId, Gain)

用关系代数和SQL写出下面的查询：找出所有在分配给他们的账户上都赚钱的经纪人的名字（即Gain>0）。

6.20 写出一个SQL语句（对于在练习6.19中给出的数据库模式）来解雇在其至少40%的账户上亏钱的所有经纪人。假设每个经纪人至少有一个账户。（提示：定义中间视图来方便地写出查询。）

6.21 解释为什么视图类似于子例程。

6.22 考虑下面的模式，它表示了待售的房屋和欲购房的顾客：

CUSTOMER (Id, Name, Address)

PREFERENCE (CustId, Feature)

AGENT (Id, AgentName)

HOUSE (Address, OwnerId, AgentId)

AMENITY (Address, Feature)

PREFERENCE是列出客户所要求的所有特性的关系（对于每个“顾客/特性”有一个元组，如<123, '5BR'>, <123, '2BATH'>, <432, 'pool'>），AMENITY是每个房屋的所有特性的关系（对于每个房屋/特性有一个元组）。

如果某顾客指定的所有特性的集合是某房屋所有特性集合的子集，那么该顾客就对购买该房屋感

兴趣。HOUSE关系中的元组说明了谁是所有者，谁是代理该房屋的房地产代理。用SQL写出下面的查询:

a. 找出对Id为007的代理所代理的每一所房屋都感兴趣的所有顾客。

b. 把前面的查询当作视图，检索形如<feature, number_of_customers>的一组元组，结果中的每个元组都给出了一个特性和要求这一特性的客户的数目，并且:

- 只考虑对Id为007所代理的每一所房屋都感兴趣的顾客。
- 对feature感兴趣的顾客的数目要大于3。(如果这个数目不大于3，那么相应的元组<feature, number_of_customers>就不需要加到结果中。)

6.23 考虑模式PERSON (Id, Name, Age)。写出一个SQL查询来找出这个关系中按年龄由大到小的顺序排在第100位的人，也就是说有99个人肯定比他老。(可能会有多个这样的人，他们的年龄相同，也可能没有这样的人。)

6.24 本章的最后一节给出了SQL中可更新视图的四个主要的规则。第三个规则说明了如果WHERE子句包含了嵌套子查询，那么在那个子查询中提到的表（显式或隐式的）不能是用在视图定义的FROM子句中的表。

构造一个视图，它违反可更新性条件3，但是满足条件1、2、4，以便存在一个对这个视图的更新，它涉及两个不同的对基本关系的更新。

6.25 举一个查询的例子，解释视图和用INSERT语句来对基本关系进行大量元组插入在概念上的区别。

第7章 查询语言 II：关系演算和可视化查询语言

在第6章，我们用关系代数来解释了SQL查询是如何计算的。事实上，数据库管理系统经常用关系代数作为SQL查询在优化前被翻译成的高级中间代码。然而，从概念上和语法上来说，SQL基于一种完全不同的正式查询语言，称为**关系演算**（relational calculus，“演算”对于这样一个相对简单的语言来说可能令人觉得很复杂）。关系演算是经典谓词逻辑的一个子集，谓词逻辑是在关系模型诞生之前已经被深入研究的一个课题。E.F.Codd的主要贡献之一是他预见到这个工具可能成为强大的数据库查询语言的基础。

有两种关系演算。在下一节中，我们介绍**元组关系演算**（Tuple Relational Calculus，TRC）的基础，它最早是在[Codd 1972]中引入的。TRC对于正确理解SQL的查询子语言是很重要的。我们将会看到SQL确实可以视为用英文单词替换一些数学符号的TRC。例如，[Date 1992]给出了TRC如何有效地作为构造复杂SQL查询的中间语言。

然后我们介绍**域关系演算**（Domain Relational Calculus，DRC），并且讨论基于它的可视化查询语言。如果你具备常见谓词逻辑的基础知识，那么DRC看起来就很熟悉了，因为它实质上是那个逻辑的一个子集。它是在[Lacroix and Pirotte 1977]中作为数据库查询语言提出的。

7.1 元组关系演算

TRC中的查询形式如下

$$\{T \mid Condition\}$$

“|”左边的查询部分称为查询**目标**（target）；右边的部分称为查询**条件**（condition，或查询体）。

目标包含一个**元组变量**（tuple variable）T，它就像代数中常见的变量一样，只是它的取值范围是值的元组（如关系元组），而不是单个的值（如数值和字符串）。

查询条件必须符合下面的条件：

- 它用到变量T，可能还有某些其他的变量。
- 如果用一个具体的值的元组替换*Condition*中T的每次出现，则条件计算得到一个布尔值*true*或*false*。

这里有一个简单的例子：找出1997年秋季开设的课程的所有教学记录。

```
{T | TEACHING(T) AND T.Semester = 'F1997'}
```

项TEACHING (T)用来测试元组T是否属于TEACHING的关系实例。T.Semester = 'F1997'是另一个测试，很容易看出这个TRC查询与下面的SQL查询相对应。

```
SELECT   *
FROM     TEACHING T
WHERE    T.Semester = 'F1997'
```

现在应该了解为什么我们之前说SQL实质上是TRC的语法变体。TRC查询的目标与SQL查询的SELECT列表相对应。TRC的查询条件在SQL中被拆分为两个子句：FROM子句（它有着形如*Relation*（*Variable*）的条件，这限制了特定的变量的取值范围为特定关系的元组）和WHERE子句（它包含所有其他的条件）。

理解TRC查询含义最简单的方法是*声明性地考虑它们，不要考虑特定的查询计算算法*。你可以把这个过程看作是为目标元组变量T枚举所有可能的选择，然后检查哪个选择在给定的数据库实例的情况下可以使查询条件为真。这个解释引出了下面的定义：

> *给定一个数据库，TRC查询的结果是能使查询条件关于此数据库为真的变量T的值的所有选择的集合。*

事实上，这个定义也是思考SQL查询的正确方法（有时也是检验SQL查询是否完成程序员想要做的事情的唯一确定的方法）。

我们来更详细地看一看这个定义的工作原理。假设我们选择<009406321, MGT123, F1980>作为上例中T的可能的答案。用它替换T产生

```
TEACHING(009406321, MGT123, F1980) AND 'F1980'= 'F1997'
```

这是个错误的猜测。这个元组不属于TEACHING的关系实例（查看图4-5），并且'F1980' = 'F1997'也不为真。所以放弃这个选择。

再看另一个选择：<009406321, MGT123, F1994>。用它替换T产生

```
TEACHING(009406321, MGT123, F1994) AND 'F1994' = 'F1997'
```

这是一个稍好的选择但是仍然是错误的猜测。尽管<009406321, MGT123, F1994>属于关系TEACHING的实例（所以部分查询条件为真），但命题'F1994' = 'F1997'仍然是错误的。

第三个选择<009406321, MGT123, F1997>产生

```
TEACHING(009406321, MGT123, F1997) AND 'F1997' = 'F1997'
```

这个查询在数据库的当前状态下为真，所以最后一个元组属于查询结果。事实上，只有三个元组满足这个查询：<009406321, MGT123, F1997>、<783432188, MGT123, F1997>和<900120450, MGT123, F1997>，这可以通过对所有域中所有可能的元素进行彻底的搜索来进行验证。

当然，数据库管理系统并不需要进行彻底的搜索。事实上，它们使用更加复杂的算法（对于不同的数据库管理系统，这些算法是不同的）。然而，我们用彻底搜索作为教学工具给出语言的语义，是因为它容易理解并且可以作为衡量较复杂的查询计算算法的标准。

另外，彻底搜索的语义在验证一个SQL查询的特定表达是否完成了我们希望它做的事情是非常有用的。因为当编写复杂的SQL查询的时候，我们已经看到人的直觉是多么不可靠。

现在我们可以来学习TRC的细节了。你会惊讶地发现我们已经给出了大多数基本的思想。只有两个问题还需要解释一下：

- 查询条件的语法。
- 用猜测元组替换查询目标中的元组变量后，条件为真的含义。

1. *查询条件的语法*

在查询条件中使用的基本构造块可以有以下几种形式：

- **P**(*T*)，**P**是关系名称，*T*是元组变量。从直观上看，**P**(*T*)用来测试元组*T*是否属于**P**的关系实例（当然，只有在给*T*选择了值以后测试才是有意义的）。例如：STUDENT(T)。
- *T.A* oper *S.B*，oper是比较运算符（=、>、≥、≠等），*T*和*S*是元组变量，*A*和*B*是属性。项*T.A*表示*T*中与属性*A*相对应的分量。这里的含义是元组的一个分量和另一个分量的比较。例如：T.StudId ≥ S.ProfId。
- *T.A* oper *const*。这和前面的形式类似，只是它是和常数比较，而不是和元组分量比较。例如：T.Semester = 'F1994'。

这些基本构造块称为**原子条件**（atomic condition）。较复杂的查询条件根据如下要求递归地构造：

- 如果*C*是原子条件，那么它是查询条件。
- 如果C_1和C_2是查询条件，那么C_1 AND C_2、C_1 OR C_2和NOT C_1也是查询条件。
- 如果*C*是查询条件，**R**是关系名称，*T*是元组变量，那么$\forall T \in \mathbf{R}(C)$和$\exists T \in \mathbf{R}(C)$是查询条件。

符号∀代表“任取”，符号∃代表“存在”。它们分别称为**全称量词**（universal quantifier）和**存在量词**（existential quantifier）。

前面已经解释了原子条件的含义，第二组中条件的含义应该是很明显的。例如，当且仅当C_1和C_2都判定为真时，C_1 AND C_2才为真。当且仅当C_1和C_2中至少有一个判定为真，C_1 OR C_2为真。当且仅当C_1为假时，NOT C_1为真。

量化公式的含义可以直接从公式读得。假设**r**是**R**的关系实例。那么$\forall T \in \mathbf{R}(C)$表示：*对于每个元组$t \in \mathbf{r}$，如果用$t$替换变量$T$，那么$C$将为真。*类似地，$\exists T \in \mathbf{R}(C)$表示：*存在一个元组$t \in \mathbf{r}$，在用$t$替换$T$之后，$C$将为真。*

我们已经介绍了语法，还有一个概念。在公式$\forall T \in \mathbf{R}(C)$和$\exists T \in \mathbf{R}(C)$中，变量*T*称为被量词所**绑定**（bound）。任何没有以这种方法显式绑定的变量都称为**自由**（free）变量。例如，在T.Name = S.Name中T和S都是自由变量，但是在∀S ∈ STUDENT（T.Name = S.Name）中，变量S是绑定的，T是自由的。

所有这些自由变量和绑定变量与查询的语义有什么关系呢？考虑句子*在X天下雨了*。这里*X*是自由变量，因此如果我们不把*X*具体化（也就是特定的值来替换它），那么这个句子就没有真值。为了比较，考虑句子*对于所有日子$X \in$ 七月，在X这一天下雨了*。句子的表述可能看起来有点别扭，但是这就是在逻辑中叙述它的方式（这样可以消除可能的混淆）。注意，第二个句子中仍然有一个变量，但是现在我们直观上能感觉到这个句子或者为真，或者为假（假设根据上下文可以弄清楚是哪年七月以及所说的是哪个地方）。这两个句子之间主要的区别是*X*在第一个句子中是自由变量，而在第二个句子中是绑定变量。

这个讨论指出，绑定变量用于生成关于数据库中元组的断言，而自由变量用于指定由查询返回的元组。这些自由变量只能用在查询目标中，因为查询结果是由为目标变量替换的具体值决定的。在替换后，查询条件就不应该再有任何自由变量了（否则我们不能辨别为目标变量选择的某个特定值是否让查询条件为真）。因此，再看一下TRC查询的定义{*T* | *Condition*}，我们应该加上下面的限制：

*T*在*Condition*中必须是唯一的自由变量。

2. 计算查询条件

那么，我们如何来决定选择的元组是否满足查询条件呢？让我们来看一个具体的例子。

```
{ E | COURSE(E) AND
    ∀S ∈ STUDENT (∃T ∈ TRANSCRIPT(                                    (7.1)
    T.StudId = S.Id AND T.CrsCode = E.CrsCode))}
```

这个查询返回每个学生都选修了的所有课程。

让我们来看一看E的某个选择是否满足查询条件。因为我们知道，正确的选择必须是COURSE中的元组，所以我们在图4-5中选择属于这个关系的一个具体的元组：<MGT123, MGT, Market, Analysis, Get rich quick>。在用这个元组替换了E之后，查询条件中不再有自由变量，但是仍然很复杂。

```
COURSE(MGT123, MGT, Market Analysis, Get rich quick) AND
    ∀S ∈ STUDENT (∃T ∈ TRANSCRIPT(                                    (7.2)
        T.StudId = S.Id AND T.CrsCode = 'MGT123'))
```

为了确定在我们的数据库中这个条件是否为真，我们通过逐步地切下语法构成的片段来递归地计算它，而这个顺序与从原子条件来构造该表达式的顺序相反。因为（7.2）的第一个分量在我们的数据库中为真（元组<MGT123, MGT, Market, Analysis, Get rich quick>描述了一个已存在的课程），我们只需要检查第二个分量是否为真。

第二个分量中最外层的片断是$\forall S \in$ STUDENT。这个量词的作用是说明，对于每一个具体的元组$s \in$ STUDENT，如果我们用元组s替换变量S，子条件

```
∃T ∈ TRANSCRIPT (T.StudId = s.Id AND T.CrsCode = 'MGT123')           (7.3)
```

必须为真。如果是这样的话，表达式（7.2）计算得到真。子条件（7.3）即使只对STUDENT中的一个元组为假，那么整个条件（7.2）也计算得到假。

因为我们需要对所有学生测试（7.3），让我们从描述John Doe的图4-2来试试这个元组。因为它的Id分量是111111111，所以我们就要用111111111替换s.Id来计算（7.3）。（7.3）中最外层的结构是∃T ∈ TRANSCRIPT，所以如果我们能找到一个元组$t \in$ TRANSCRIPT，并且在用t替换T之后条件

```
t.StudId = 111111111 AND t.CrsCode = 'MGT123'
```

为真，那么（7.3）的结果为真。顺便提一下，通过TRANSCRIPT中的一个元组可以看出John Doe在1997年秋季选修了MGT123，并成绩为B。所以，用这个元组替换T，我们得到

```
111111111 = 111111111 AND 'MGT123' = 'MGT123'
```

这在我们的数据库中为真，因为每个子条件都为真。

假设我们最初的选择<MGT123, MGT, Market Analysis, Get rich quick>是查询的一个答案，我们的工作是不是完成了呢？不！我们还要检验对于每个元组$s \in$ STUDENT，（7.3）都为真。到目前为止，我们只是确定了当s是描述John Doe的元组时，这个条件才为真。

现在假设s是描述Homer Simpson的元组，其Id为023456789，为了使s满足（7.3），我们需要验证，在TRANSCRIPT中存在一个元组，如果我们用它来替换T，那么在数据库中，下面的语句将为真：

```
T.StudId = 023456789 AND T.CrsCode = 'MGT123'
```

我们可以依次试试TRANSCRIPT中的每一个元组，但是我们通过验证这个关系没有元组的StudId属性是023456789且CrsCode属性是MGT123，从而节省一些时间。因此，（7.3）中Homer Simpson元组的替换产生了一个在我们数据库中为假的条件，因而（7.3）对于我们数据库中的每一个学生并不是都为真。这意味着（7.2）为假，所以我们最初的（对COURSE中某元组的）假想不是查询（7.1）答案。

如果你有时间，你可以检查一下没有选择满足（7.1）中的查询条件，所以结果为空。但并不是一定会这样。如果我们为MGT123增加适当的成绩记录，使我们的数据库中所有学生都选修了这门课程，那么描述MGT123的COURSE元组将会属于查询结果。

上面的过程尽管很乏味，但是实质上是专家用来检验一个选择是否属于查询结果的方法。有了一些经验之后，你可以开发各种便捷方法和省时的工具（“优化”），但是上面的过程仍然将作为“正确性准则”。查询设计者可以用某个类似的过程来验证一个特定的查询设计是否满足相应的要求。

3. 例子

上面检验TRC查询的过程无疑是乏味的，但是你不必在每次设计一个查询的时候都执行它（就好像你不用为文件中的每条记录都手工执行Java程序来确保这个程序是正确的）。在大多数的情况下，如果你严格地把TRC翻译成自然语言，那么可以通过阅读这些文字来验证查询。这个方法在许多情况下是很有效的，因为人对于逻辑正确性有着天生的敏感。然而，有时复杂的查询翻译可能也会非常复杂，这时上面的验证过程就起作用了⊖。

为了更好的感受一下TRC，我们考虑一些通常在关系代数中需要联结、投影和选择的查询。在TRC中，这样的查询需要∃和AND。为了简化我们的查询，我们稍微扩展一下语法，允许在目标中出现多个元组变量，对于不同的元组变量允许混合使用形如*T.attribute*的项。比如，

```
{S.Name, T.CrsCode | STUDENT(S) AND TRANSCRIPT(T) AND...}
```

这个扩展只是为了符号上的方便，因为我们能够用先前较严格的语法来编写同样的查询，这就要引入新的具有属性Name和CrsCode的元组变量R。

```
{R |∃S ∈ STUDENT (∃T ∈ TRANSCRIPT (
    R.Name = S.Name AND R.CrsCode = T.CrsCode AND...))}
```

查询列出教过MGT123的所有教授的名字可以表达为:

```
{P.Name | PROFESSOR(P) AND
  ∃T ∈ TEACHING (P.Id = T.ProfId AND T.CRSCODE = 'MGT123')}
```

注意，如果我们要把这个查询表达成关系代数形式的话，它将涉及到选择、投影和联结运算符。相应的SQL查询也只是语法的变体。

```
SELECT   P.Name
FROM     PROFESSOR P, TEACHING T
WHERE    P.Id = T.ProfId AND T.CrsCode = 'MGT123'
```

⊖ 比较一下验证TRC（或SQL）查询和验证Java程序是很有趣的。在TRC中，程序短小但是单个语句的功能却非常强大。因而，困难就在于验证少量功能强大的语句的组合含义。相比之下，在Java中，每个命令都很容易检查，但是程序却很长。因而，弄懂大量简单语句的执行结果是很困难的。

如果我们想得出教过Homer Simpson的教授的名字，我们可以编写以下语句:

```
{P.Name | PROFESSOR(P) AND
    ∃T ∈ TRANSCRIPT (∃S ∈ STUDENT (∃E ∈ TEACHING(
        P.Id = E.ProfId AND T.StudId = S.Id AND
        E.CrsCode = T.CrsCode AND E.Semester = T.Semester
        AND S.Name = 'Homer Simpson'))}
```

下面的查询找出选修过两次（但是在不同学期）同一门课程的所有学生的Id:

```
{T.StudId | TRANSCRIPT(T) AND
    ∃T1 ∈ TRANSCRIPT(
        T.StudId = T1.StudId AND T.CrsCode = T1.CrsCode
        AND T.Semester ≠ T1.Semester)}
```

注意，在这个查询中我们用到了两个元组变量，一个是自由变量，一个是绑定变量，两个都在TRANSCRIPT上变化。在关系代数中，我们可能就要将TRANSCRIPT与自身做联结了（不使用等值联结）。

让我们再来看一看在关系代数中需要用到除法运算符的查询。我们已经详细讨论了查询(7.1)，它产生每个学生都选修过的所有课程的列表。在讨论除法运算符时，我们构造查询(6.1)，它产生选修过教过课的每个教授的课的所有学生的列表。在TRC中，这个查询表达为

```
{R.StudId | ∀T ∈ TEACHING (∃T1 ∈ TEACHING (TRANSCRIPT(R)
    AND T.ProfId = T1.ProfId AND T1.CrsCode = R.CrsCode
    AND T1.Semester = R.Semester))}
```
(7.4)

下面将解释这个查询。T.ProfId的一个特定值确定了教过课的一个教授。如果TEACHING中存在一个元组T1，指出了这个教授教了一门课（即T.ProfId = T1.ProfId），并且这个学生选修过这门课（即T1.CrsCode = R.CrsCode AND T1.Semester = R.Semester），那么Id为R.StudId的学生就选修过这个教授的这门课。如果这个学生选修过所有这样教授（∀T ∈ TEACHING）的一门课的话，那么该学生就应该在查询结果中。

这个查询比等价的代数查询（6.1）看起来要稍微复杂一些。然而，我们在TRC中不需要引入特别的运算符来处理这样的查询（试试不用除法运算符来编写查询（6.1）!），并且TRC查询也更加灵活。例如，如果我们想列出的是学生的姓名而不是Id，那么我们所要做的就是把S.Id替换成S.Name。相比之下，在代数中这需要与COURSE关系再做一次联结。

注意，如果我们省略存在量化的变量T1，而在查询的其余部分用T来替换它，我们将得到选修过学校所有课程的所有学生（与在每个教授那里选修过一门课相反）。研究一下这个查询并理解它与查询（7.4）之间的区别是一个很有用的练习。

与（7.4）相应的SQL查询就更难编写了，因为SQL设计者考虑到全称量词难以正确使用，所以就从该语言中去掉了这些量词。（但是我们应该注意到SQL:1999已经引入了全称量词和存在量词的受限形式，即FOR ALL和FOR SOME运算符。）为了弥补这一点，SQL具有许多像嵌套子查询和EXISTS运算符这样的特性，但是这些特性更难使用。因而，用TRC和用SQL表达上面查询的主要的语法差异就在于需要把全称量词翻译成等价的使用嵌套子查询的语句。和SQL不同，TRC没有支持嵌套子查询的语法。然而，这样的子查询可以用本节后面讨论的视图来表达。

TRC查询有一些重要的特性值得注意。首先，相邻的存在量词可以互换位置。事实上，

我们既可以写成

∃R ∈ TRANSCRIPT (∃T1 ∈ TEACHING (...))

也可以写成

∃T1 ∈ TRANSCRIPT (∃R ∈ TEACHING (...))

这可以用之前给出的计算查询条件的方法来加以验证。

第二，全称量词和存在量词不可以互换位置。好好想一想就知道“对于每个TEACHING元组，存在一个TRANSCRIPT元组使得命题St为真”和“存在一个TRANSCRIPT元组，对于所有的TEACHING元组，命题St为真”是不一样的。

第三，在关系演算中，量词类似于begin/end块，因为它们定义了变量的作用域。例如，

$$\forall T \in \mathbf{R}_1 \left(U(T) \text{ AND } \exists T \in \mathbf{R}_2 \left(V(T)\right)\right)$$

是合法的。变量T的两次出现是处于不同量词的作用域内的。因此，就像同名变量出现在嵌套的begin/end块中那样，这两次出现是完全独立的，上面的表达式等价于

$$\forall T \in \mathbf{R}_1 \left(U(T) \quad \text{AND} \quad \exists S \in \mathbf{R}_2 \left(V(S)\right)\right)$$

这里S是某个其他的新变量。

4. TRC中的视图

现在，假设我们想稍微修改一下查询（7.4）。这次不是列出选修过每个教授的一门课程的学生，而是想看看选修过每个计算机科学系教授的一门课程的学生。这看起来好像只是很小的变化，但是可能会给TRC带来极大的困难（而利用代数却不会！）。这个查询的SQL公式在查询（6.24）中给出。

直观地，修改（7.4）来处理这个查询的最简单的方法是用∀T ∈ CSTEACHING来替换∀T ∈ TEACHING，这里CSTEACHING是TEACHING的一个子集，它与由计算机科学系教授教的课程相对应。如果我们有这样的一个关系，那么这个改变确实是正确的。然而，这个关系在我们的数据库中是不存在的，所以我们需要找到绕过这个问题的一个途径。

我们可能想通过向查询条件（在最内层的括弧中）增加

```
AND ∃P ∈ PROFESSOR (P.DeptId = 'CS' AND P.Id = T.ProfId)        (7.5)
```

来改变（7.4），但这是错误的。因为T是在整个TEACHING关系上变化，上面对（7.4）的增加必须对TEACHING中T.ProfId的所有值都为真，这只有在数据库中每个任课教授都在计算机系工作的情况下才有可能（而这并不是实际情况）。正确的查询比较复杂，这需要一些耐心来理解这样编写的原理。

```
{R.StudId | ∀T ∈ TEACHING (∃T1 ∈ TEACHING (TRANSCRIPT(R) AND
   NOT (∃P ∈ PROFESSOR (P.DeptId = 'CS' AND P.Id = T.ProfId))
    OR (T.ProfId = T1.ProfId AND T1.CrsCode = R.CrsCode
        AND T1.Semester = R.Semester)))}                        (7.6)
```

这个条件指出，R的值必须是TRANSCRIPT的元素，T.ProfId可能不是计算机系的教授，或者R.StudId选修过一门T.ProfId教的课。

幸运的是，有一种获得相同结果的更为简单的方法。让我们回到我们第一次的尝试中去：试着用∀T ∈ CSTEACHING。这里的问题是关系CSTEACHING并不在我们最初的数据库中。然而，

我们可以用查询来定义它。事实上我们定义了更简单的关系。

```
CSPROF = {P.ProfId | PROFESSOR(P) AND P.DeptId = 'CS'}
```

我们可以在较大的查询中把它用作数据库视图（或作为“子例程”）。

```
{R.StudId | ∀P ∈ CSPROF (∃T1 ∈ TEACHING (TRANSCRIPT(R)
    AND P.ProfId = T1.ProfId AND T1.CrsCode = R.CrsCode
          AND T1.Semester = R.Semester))}                    (7.7)
```

实际上，复杂查询（7.6）是类似于采用查询（7.7）并在视图CSPROF的定义中做了替换了的TRC。这样替换复杂的结果并不令人觉得奇怪，想一想如果Java程序中所有子例程都被替换了将会怎样！浏览一下代数查询的例子，它揭示了这种子例程的广泛使用，这就是为什么一些查询看起来比它们对应的TRC查询更简单的原因。

7.2 通过元组关系演算理解SQL

我们已经看到，SQL实质上就是元组关系演算的语言，其间点缀着别的语言，目的是隐藏与谓词逻辑的关系[⊖]。

除了来源上的关系外，TRC对于SQL来说是极为重要的，因为理解TRC有助于编写复杂的SQL查询。一些SQL书籍依靠关系代数来解释通常的SQL查询的意义（就像第6章那样）。但是，代数表达式并不比等价的TRC查询直观，对较复杂的SQL也没有多少帮助。事实上，把6.2.3节中的SQL查询（它使用嵌套子查询）翻译为代数查询不是一个简单的任务。更重要的是，关系代数并不是有助于把SQL数据库查询翻译成英语来检验它们是否如期工作的理解工具。

尽管英语描述没有数学规范那么正式，但是它也有自己的优点，就是顾客和程序员都很容易理解它并因此共同确认它与他们的目标相对应。但如何确定给定的SQL查询满足英语规范呢？我们可以进行如下推理。当且仅当下列条件满足时，SQL查询符合英语规范：

1) 对于根据英语表述必须在查询结果中的每个元组t，有办法为FROM子句中的元组变量分配元组，让这些元组满足WHERE条件并且查询目标列表判定为t（在用那些元组替换元组变量后）。

2) 由SQL查询产生的每个元组都在英语表述的结果中。

应用这些条件的主要困难是确定给变量分配特定的元组以满足WHERE条件意味着什么。在简单的情况下你可以靠直觉，但是对于（6.24）这样较复杂的SQL查询，那就越来越容易出错了。

幸运的是，这个困难是可以解决的，因为SQL和TRC之间存在紧密的对应关系。给定一个SQL查询，其思想是首先构造一个等价的TRC查询，然后把它翻译成英语。因为TRC相对英语来说，更加接近于SQL，所以更有可能正确地做好第一步。下一步是将TRC翻译为英语，这实际上是已经经过良好定义的过程（在很多关于谓词逻辑的书里已经进行了解释），它可以由计算机来执行。

⊖ 追溯到20世纪70年代，当时第一个关系系统还正在实验室中研制，那时就希望基于关系演算的语言能够提供相当高层次的编程，这样即使非技术用户（如公司的CEO）也能使用它们。不用说，甚至在二十年之后这个希望还没有实现，而SQL随着每一次新的发布却变得越来越复杂。

用这种方法，检验SQL查询的问题就简化为把SQL翻译成TRC的问题。在简单的情况下，这个翻译是很容易的。我们现在用带嵌套子查询的一个复杂SQL查询的例子来说明这一思想。考虑下面的查询模板，为了明确起见我们假设REL1的属性是A、B，REL2的属性是C、D，REL3和REL4分别具有属性E、F和G、H。

```
SELECT   R1.A, R2.C
FROM     REL1 R1, REL2 R2
WHERE    Condition1(R1, R2) AND
    R1.B IN (SELECT R3.E
             FROM REL3 R3, REL4 R4
             WHERE Condition2(R2,R3,R4))
```
(7.8)

在这个模板里，变量R3和R4对于子查询来说是局部的，变量R1和R2是全局的。另外，子查询的WHERE条件用到了变量R2、R3、R4（这就是符号*Condition*2（R2, R3, R4）想表达的）。因为R2是*Condition*2用到的全局变量，它是内层子查询的参数。

我们假设*Condition*1是很适合TRC的形式（即它是关系代数中的合法选择条件）。然而，嵌套子查询显然不是TRC可以理解的形式。不过，把它翻译成TRC是很简单的。为了说明这一点，回忆一下前面对关系视图的讨论。

我们来把（7.8）中的内层子查询表示成视图，即已命名的虚拟视图（称为TEMP），不存储视图的内容，而是由查询计算得到视图的内容。这里需要注意一个细节：可以使用的TEMP的正确的属性集是什么？我们不能只是从内层子查询（即E）的目标列表中取出属性，因为内层子查询（和它的结果）被全局变量R2参数化了。回忆一下，条件中的自由变量是那些在目标列表中已命名的变量（因为一旦给目标变量的值做了替换，条件必须判定为真或假）。R2必须是自由的，因为在外层查询中给它分配了一个值。因此，TEMP正确的属性集除了内层子查询的目标列表（即E）中的属性外，还必须包括REL2的属性（C和D）[⊖]。这个推理引出了下面TEMP的视图定义：

```
TEMP = {R3.E, R2.C, R2.D | REL2(R2) AND REL3(R3)
            AND ∃R4∈REL4 (Condition2(R2,R3,R4))}
```
(7.9)

在这里，变量R2和R3是自由的，但是R4必须被量化，因为它不在查询的目标列表中出现。

现在，将（7.8）翻译为TRC的结果是

```
{R1.A, R2.C | REL1(R1) AND REL2(R2) AND Condition1(R1,R2) AND
        ∃R∈TEMP (R.E = R1.B AND R.C = R2.C
            AND R.D = R2.D)}
```
(7.10)

这个TRC查询是如何表达条件R1.B IN（SELECT R3.E...）的呢？通过等式R.E = R1.B。事实上，当且仅当这些元组分别属于关系REL1和REL2，并且<b, c, d>是TEMP中的元组时，为R1指定元组<a, b>和为R2指定元组<c, d >满足（7.10）中的条件。TEMP表示（7.8）中内层子查询。但是b是R1.B的值，这意味着b必须是内层子查询的结果。

再看另一个例子，考虑较复杂的查询（6.24），它返回选修过计算机系每个教授教的一门课程的所有学生的Id。这里我们要构造两个视图，即为（6.24）的每个子查询构造一个视图。

⊖ 实际上，我们只需要包含REL2的那些在*Condition*2中实际用到的属性。也就是说，如果*Condition*2用到R2.C，但没有用到R2.D，那么TEMP的属性集就不需要包含R2.D（不过包含它也没有什么坏处）。

第一个视图CSPROF是很简单的:

```
CSPROF = {P.ProfId | PROFESSOR(P) AND P.Dept ='CS'}
```

这个视图的结果是计算机系所有教授的Id的集合。(6.24) 的第二个子查询返回教过Id为R.StudId的学生某门课的所有教授的Id的集合，这里R是参数化子查询的变量。因为之前解释过，与这个子查询相应的TRC视图必须在目标列表中包括R:

```
PROFSTUD = {T.ProfId, R.StudId | TEACHING(T) AND TRANSCRIPT(R) AND
            ∃R1∈TRANSCRIPT (T.CrsCode = R1.CrsCode AND
                    T.Semester = R1.Semester AND
                    R.StudId = R1.StudId)}
```

最后，整个查询 (6.24) 能够重述如下: 我们想找这样的学生s，在CSPROF中不存在某个教授c，使得元组<c, s>不在PROFSTUD中。用TRC来表达这个查询有点麻烦。

```
{S.StudId | STUDENT(S) AND NOT (∃C∈CSPROF (NOT(∃P∈PROFSTUD
                (P.ProfId = C.ProfId AND P.StudId = S.StudId))))}
```

如果我们回忆一下在标准逻辑里NOT ∃ XNOT等价于X，那么这个表达式可以被简化为:

```
{S.StudId | STUDENT(S) AND ∀C∈CSPROF ∃P∈PROFSTUD
                (P.ProfId = C.ProfId AND P.StudId = S.StudId)}
```

这个语句可以表述为: 当且仅当对于每个计算机科学系的教授c来说，元组<c, s>在视图PROFSTUD中，即c教过s，那么学生Id s在这个查询的结果中。

现在把它和 (6.24) 中最初的表述比较一下: 找出选修过计算机系每个教授教的一门课的所有学生。这两个句子具有相同的含义，这就意味着我们验证了这个复杂的SQL查询。

7.3 域关系演算和可视化查询语言

元组关系演算已经成为像SQL这样的文本数据库查询语言的基础。关系演算的另一个分支是**域关系演算** (Domain Relational Calculus，DRC)，它已经成为像IBM的Query-By-Example这样的可视化查询语言和像Microsoft的Access、Borland的Paradox这样的PC数据库语言的基础了[⊖]。

DRC和TRC非常相似。主要的区别在于DRC在查询中使用了**域变量** (domain variable)，而不是元组变量。回忆一下，元组包含了一组命名的属性，每个属性从某个域中取值。域变量从某个属性的域中取值。例如，TEACHING关系可能有名为ProfId的属性，它的域由有效的Id组成。Pid可以成为域变量，它从有效Id的域中取值。

和TRC相似，DRC查询的输出是一个关系，DRC查询的通常形式是

$\{X_1, \cdots, X_n \mid Condition\}$

其中，$X_1, \cdots, X_n$是一组域变量 (不必互不相同)，它们形成了查询的**目标**部分，*Condition* 是查询条件，看起来和TRC中的查询条件一样。这里有一个例子:

```
{Pid, Code | TEACHING(Pid,Code,F1997)}
```

⊖ 在TRC或者DRC中并没有什么固有的东西使得一种形式比另一种形式更适合于一种特定的查询语言。选择TRC作为SQL的基础只是历史的偶然。

它实质上和下面的TRC查询是一样的。

```
{T | TEACHING(T) AND T.Semester = 'F1997'}
```

但是它更为简单，因为我们可以把常数F1997直接放进条件TEACHING（Pid, Code, F1997）中，从而就不需要T.Semester = 'F1997'了。这样的替换经常使得DRC查询更加简洁，也比它们的TRC等价表达更加容易理解。

因为DRC和TRC十分相近，它们用到的许多特性和技术或者相同，或者非常类似。因此，我们将不像讨论TRC那样详细地讨论DRC。

1. DRC查询条件的语法

DRC中的查询条件的一般语法类似于在TRC所介绍的语法。基本的构造块有以下几种形式：

- **P** (X_1, ⋯, X_n)，这里**P**是关系名称，X_1, ⋯, X_n是域变量。直观地，**P** (X_1, ⋯, X_n)表示测试元组<X_1, ⋯, X_n>是否属于**P**的关系实例（就像在TRC中那样，这个表达式是在为X_1, ⋯, X_n选择了某个特定的值之后才计算的）。例如，STUDENT（Sid, N, A, S）。
- X oper Y，这里oper是比较运算符（=、>、≥、≠等），X和Y是域变量。这里是指将X的值和Y的值进行比较。例如：Sid ≥ Pid。注意，因为域变量直接表示元组分量，我们不需要像在TRC中那样给X和Y加上关系名称的前缀（比如，A.Sid = B.Pid）。
- X oper *const*，它类似于前面的形式，但是这里的比较是和常数比，而不是和变量比。例如：X = 'F1994'。

这些基本构造块称为**原子条件**（atomic condition），更为复杂的查询条件是由如下原则递归构造而成的：

- 如果C是原子条件，那么它是查询条件。
- 如果C_1和C_2是查询条件，那么C_1 AND C_2、C_1 OR C_2和NOT C_1也是查询条件。
- 如果C是查询条件，**R**是关系名称，X是域变量，那么$\forall X \in \mathbf{R}.A(C)$和$\exists X \in \mathbf{R}.A(C)$也是查询条件。

这里的量词类似于TRC中的量词，只有一点不同：表达式**R**.A表示关系**R**的A列，变量X的取值范围是出现于该列的值。换句话说，$\forall X \in \mathbf{R}.A(C)$表示对于出现在关系**R**（在当前的数据库实例中）的A列上的所有值，条件C必须为真。表达式$\exists X \in \mathbf{R}.A(C)$表示，在列**R**.$A$上至少有一个值$x$，使得如果用$x$替换出现的$X$，则$C$为真。

自由变量和绑定变量的规则和TRC中的规则是一样的，并且像在TRC中那样，只允许目标变量在查询条件中自由出现。为了让这个规则更加灵活，我们也允许常数出现在目标中。总结一下，在

$$\{X_1, \cdots, X_n \mid Condition\} \tag{7.11}$$

中，每个X_i（$i = 1, \cdots, n$）或者是出现在*Condition*中的自由变量，或者是常数。没有其他变量自由出现在*Condition*中。

查询（7.11）在给定数据库上定义的查询结果类似于TRC查询中的那样：它是元组<x_1, ⋯, x_n>的所有选择的集合，这样当用x_1替换X_1，用x_2替换X_2时，*Condition*结果为真。如果某个X_i是常数，就不进行替换：查询结果在所有元组的第i个位置上有这个常数。

2. 例子

为了说明DRC的使用以及说明它和TRC的区别，我们用了说明关系代数和TRC时用到的同一套查询。

DRC查询用到的变量通常比与它们对应的TRC查询更多，因为DRC变量是在简单的值上变化的，而TRC变量是在元组上变化的。因此，一个TRC变量可以有效地代表几个DRC变量。结果，DRC查询常常使用更多的量词。但是，有几种方法可以让符号变得更为简洁。一个办法是引入**全称域**（universal domain）U，它包含了所有域中的所有值，这样我们不用写成$\exists X \in U$（*Condition*），而可以用简略的表示方式$\exists X$（*Condition*）。

全称域如此有用的一个原因是任何形如$\exists X \in \mathbf{R}.A_i$（$\mathbf{R}$（…, X, …）…）的表达式（这里A_i是$\mathbf{R}$的属性，它相应于X的出现）等价于$\exists X$（$\mathbf{R}$（…, X, …）…）。换句话说，我们可以用全称域U来替换域$\mathbf{R}.A_i$。当然，U包含了$\mathbf{R}.A_i$，但是它可能大得多。然而，X的任何超出$\mathbf{R}.A_i$的选择x，都不能使$\exists X$（$\mathbf{R}$（…, X, …）…）结果为真，因为为了让$\mathbf{R}$（…, x, …）为真，x必须是$\mathbf{R}$的实例中某个元组的A_i的值，因此对某些i来说，x必须属于$\mathbf{R}.A_i$的域。

现在让我们回到7.1节的查询例子。与查询列出教过MGT123的所有教授的名字等价的DRC是

```
{N | ∃I∈PROFESSOR.Id ∃D∈PROFESSOR.DeptId (
        PROFESSOR(I,N,D) AND
        ∃S∈TEACHING.Semester (TEACHING(I, MGT123, S)))}
```

粗略地看一下这个查询就可以清楚地看出全称域的作用：舍弃了长长的域名称使得查询更为简洁。我们可以在下面的例子中利用这个性质。

下一个查询返回教过Homer Simpson的所有教授的名字。

```
{Pname | ∃Pid ∃Dept (PROFESSOR(Pid, Pname, Dept) AND
        ∃Grd ∃Crs ∃Sem ∃Sid ∃Addr ∃Stat (TEACHING(Pid, Crs, Sem)
            AND TRANSCRIPT(Sid, Crs, Sem, Grd)
            AND STUDENT(Sid, Homer Simpson, Addr, Stat)))}
```

把上面两个查询和它们相对应的TRC进行比较是很有指导作用的。DRC查询通常需要更多量化变量，但是需要较少形如$\mathbf{R}_1.Attr_1 = \mathbf{R}_2.Attr_2$的分量。然而，基于DRC和TRC而构建的商业数据库语言不显式使用量词。事实上，所有不显式出现在查询的目标列表中的变量都假设被∃隐式地量化，并在全称域上变化。利用这个规则，上面的DRC查询中的所有量词都可以舍弃。例如，第二个查询变成为

```
{Pname | (PROFESSOR(Pid, Pname, Dept)
        AND TEACHING(Pid,Crs,Sem)
        AND TRANSCRIPT(Sid, Crs, Sem, Grd)
        AND STUDENT(Sid, Homer Simpson, Addr, Stat))}
```

相比而言，TRC量词在不对查询做进一步修改的情况下是不能舍弃的，否则在某些情况下我们可能丢失关于相应量化变量的有用信息。为了说明这个问题，我们再来看看关于教过MGT123的教授的TRC查询。

```
{P.Name | PROFESSOR(P) AND
    ∃T∈TEACHING (P.Id = T.ProfId AND T.CrsCode = 'MGT123')}
```

这里舍弃∃T ∈ TEACHING将会使T变成“未声明”（编程语言的术语），因为我们丢失了T的变化范围是TEACHING这样的信息。变化范围信息在DRC中是不会丢失的，因为未声明的变量都假设是在全称域上变化。然而，注意到∃T ∈ TEACHING（…）等价于∃T ∈ TEACHING（TEACHING (T) AND …），我们可以把这个查询重写为

```
{P.Name | PROFESSOR(P) AND
    TEACHING(T) AND P.Id = T.ProfId AND T.CrsCode = 'MGT123'}
```

如果我们假设非目标变量的T隐式地由∃量化，那么这个公式就是正确的。

当在DRC或TRC中舍弃存在量词的时候，如果∃和∀都出现在同一个查询中，则很可能存在不明确性。例如，考虑∀Y（LIKES（X, Y）），这里LIKES（X, Y）意味着*X*类似于*Y*。*X*隐含地由∃来量化这条规则还让我们面临了两难的选择：这个表达式表示∀Y∃X（LIKES）（X, Y）还是∃X∀YLIKES（X, Y）？思考一下就可以知道，这两个表达式对应着两个完全不同的句子：*每个Y都与某个X相似*和*某个X与每个Y都相似*。因此，在一些情况下是不可以舍弃量词的。这种二义性不会在SQL中出现，因为SQL不允许使用∀，因为优化这样的查询是很困难的[⊖]。事实上，如果程序员提交的查询需要用到∀或除法运算符，那么它们就得从循环中跳出来。

我们在6.2节中研究了克服SQL的这个缺陷的技术，但是在利用这个技术之前，我们先看看如何用DRC来表示查询*找出选修过每个教授的一门课的所有学生*。

```
{Sid | ∀Pid∈TEACHING.ProfId (∃Crs ∃Sem ∃Grd (
        TEACHING(Pid, Crs, Sem) AND
        TRANSCRIPT(Sid, Crs, Sem, Grd)))}
```

(7.12)

这个DRC查询实质上比相应的SQL查询（6.24）和TRC查询（7.4）要简单。特别是不必两次提到TEACHING。（回忆一下在TRC中，TEACHING因为不明确的原因而必须出现两次。）注意，在（7.12）中我们显式地用到∃，来避免因为∀和∃出现在同一个查询中而引起的二义性。

还有一点必须指出，我们不能用全称域*U*来替换变量P.Id的域TEACHING.ProfId。通常，$\forall X \in \mathbf{R}.A_i$（**R**（…, *X*, …））并不等价于∀*X*（**R**（…, *X*, …））。用*U*而不是用TEACHING.ProfId意味着为了使某个特定的值sid在查询（7.12）的结果中，条件TEACHING（pid, crs, sem）AND TRANSCRIPT（sid, crs, sem, grd）对于在*U*中的所有pid值和某些crs、sem、grd值都必须为真。若pid是*U*中的一个值但它又不和任何教授的Id相对应是不可能的（*U*必须有这样的值，因为它完全包含TEACHING.ProfId）。然而，就像之前讨论的那样，在$\exists X \in \mathbf{R}.A_i$域中可以用*U*来替换$\mathbf{R}.A_i$。

7.4 可视化查询语言：QBE和PC数据库

QBE（Query-by-Example）是第一个受到广泛欢迎的可视化查询语言。它和SQL一样也是由IBM开发的，开发时间大约与[Zloof 1975]相同。QBE是IBM的DB2关系数据库产品的一部分，其他一些供应商也开发过QBE的类似产品。然而，可视化查询语言最大的成功是伴随PC数据库（如Borland的Paradox和Microsoft的Access）的出现而到来的。

在这一节中，我们要研究QBE。我们本节的目标是强调概念，而不是提供详尽的参考。在本节的最后一部分，我们简单讨论一下Microsoft的Access，它的界面也是基于域演算，但

⊖ SQL:1999引入了全称量词的受限形式。

是比QBE更为自由。

1. QBE的基本概念

QBE是纯粹的理论工具——域演算用于实际应用的最好例证之一。QBE的主要原理是用户（他可能不是程序员）通过从菜单中选择关系模板，然后再给这些模板填充“样例元组”（它们指定希望的结果）来指定查询。QBE中的样例元组包含变量和常数，但是即使是变量，看起来也像是“常数的样例”——QBE尽量避免了编程的符号[⊖]。

假设你想找出在MGT系中的所有教授。在QBE中，你选择一个与PROFESSOR关系相应的模板，然后填入如下样例元组：

PROFESSOR	Id	Name	DeptId
		P._John	MGT

很明显，这个QBE查询只是用于文本方式的DRC查询{Name | ∃ I PROFESSOR (I, Name, MGT)}的不同的可视化表示。符号_John是伪装的域变量，而MGT是常数。即使一般用户看起来这两个符号是相似的，但前缀“_”也显示出_John作为不同寻常的变量的特殊状态。另外，表示打印的运算符P.指明_John在关系演算的意义上是目标变量，因为它是由打印命令来输出的。QBE使用几个运算符，通常表示为关键字后面跟着一个句点，但是在这个概述中我们主要使用P.——所有QBE运算符之源，它把目标变量从其他部分分离开来。

我们之前提到商业的查询语言不会显式地量化变量，当然我们在QBE查询中找不到量词。事实上，所有的非目标变量都假设为隐含地由∃来量化。然而，QBE更进一步：可以省略一些非目标变量（如上面查询的DRC版本中的I）。当出现这种情况时，系统生成一个独一无二的变量名，并自动地把它替换进来。

实际上，即使变量_John也不是必需的，我们可以舍弃它而不会影响最终的结果。在那种情况下，Name列只包含P.。因而P.可以像独立运算符那样在查询中出现。如果查询结果必须是具有两个或两个以上属性的关系，我们可以把P.放到查询的几个地方。如果关系的所有属性都要输出，我们可以把P.直接放到关系名下，而不是放在每个列里。

PROFESSOR	Id	Name	DeptId
P.			MGT

2. 连接和高级查询

上面的查询没有显式地提到任何变量。然而我们通常又不能不使用变量。如果查询中需要用到某个变量两次或两次以上，那么它表示在相同或不同元组的属性之间的相等关系，因此不能省略它。经常在指定等值联结的时候出现这个情况。下面是类似的查询（列出教MGT123的所有教授的名字）的QBE版本：

PROFESSOR	Id	Name	DeptId
	_123456789	P._John	

⊖ QBE尽力消除任何编程术语的痕迹。它称常量为“例值”，称变量为“例元素”。

TEACHING	ProfId	CrsCode	Semester
	_123456789	MGT123	

这里的_John就像前面的一样是目标变量，_123456789是由∃隐式量化的变量。它在这个查询中用于指定两张表的等值联结，因此不能省略。（相反，_John像前面一样可以省略。）

我们已经看到，QBE让用户通过把适当的常数放入模板来指定简单的选择。QBE更进一步，允许用下面的语法来找出Id大于指定常数的所有教授的名字。

PROFESSOR	Id	Name	DeptId
	> 123456789	P.	

（注意常数123456789和变量_123456789之间的差别）。然而，较复杂的选择或联结条件不能通过这种方式来指定，所以QBE提供了特别的模板（称为**条件框**，condition box），用户可以在条件框里用类似于关系代数的语法编写任何复杂的选择和联结条件。例如，为了得到选修过CS305并且成绩为A或B的所有学生的Id，我们可以写成如下形式：

TRANSCRIPT	StudId	CrsCode	Semester	Grade
	P.	CS305		_G

CONDITIONS
_G = A OR _G = B

因为QBE查询的结果是关系，在**P.**运算符出现的位置上有某些限制。特别地，为了避免**P.**可能放在不同关系模板的两个同名属性之下，QBE要求**P.**必须都在同一个模板中出现⊖。那么我们如何来指定查询*找出所有教授以及他们教过的学生*？因为教授和学生存储在不同的关系中，所以如果**P.**的所有形式只可以出现在两个模板之一，它就不能回答这个查询。

一个解决方案是允许用户构造*新的*模板，所以他们就不会局限于数据库中已有的关系模板。为了回答上面的查询，用户使用菜单系统来按如下方式定义新的模板HASTAUGHT：

HASTAUGHT	Prof	Stud
I.	_123456789	_987654321

TRANSCRIPT	StudId	CrsCode	Semester	Grade
	_987654321	_CS305	_F1996	

TEACHING	ProfId	CrsCode	Semester
	_123456789	_CS305	_F1996

模板HASTAUGHT中的运算符**I.**（*意味着插入*）指定了应该位于相应表的元组，然后用户可以按如下方式使用**P.**运算符来查询HASTAUGHT。

HASTAUGHT	Prof	Stud
P.		

⊖ 这是一个相当严重的限制，引入它是为了解决相对小的问题。

3. 免费午餐的代价

前面的例子说明了正确选择可视化原语和规则可以让即使是非专业的人士也能理解域关系演算。但是和语言设计中出现的情况类似，让一些事情变得简单的特性却让其他的事情变得困难。就数据库查询语言而言，代价通常是表达能力有所损失（即不能指定某些类型的查询）或者是指定起来不方便。在QBE的情况下（实际上SQL也一样），当需要提交涉及到除法运算符的查询（例如找出数据库中每个学生都选修过的所有课程），就必须付出代价了。我们在后面再说到这个查询。

因为QBE不提供全称量词∀，所以它使用否定运算符来构造¬∃¬，它表达了同样的意思。为了说明这一点，让我们来考虑一下查询列出在1995年秋季没有教过一门课的所有教授。在DRC中，这个查询既可以表达为

```
{Name|PROFESSOR (Id, Name, DeptId)
        AND ∀CrsCode (NOT(TEACHING (Id, CrsCode, F1995)))}
```

也可以表达为:

```
{Name|PROFESSOR (Id, Name, DeptId)
        AND NOT (∃CrsCode (TEACHING (Id, CrsCode, F1995)))}
```

在这两种情况下，我们都假设Id和DeptId是存在量化的。QBE选择了第二个公式，它表示为

PROFESSOR	Id	Name	DeptId
	_123456789	P.	

TEACHING	ProfId	CrsCode	Semester
¬	_123456789		F1995

符号“¬”表示否定运算符，它出现在TEACHING模板中，因为我们要求在TEACHING中没有哪一行可以匹配PROFESSOR中某个特定Id。这个否定大致与差集相对应。第一个模板产生了PROFESSOR关系中所有教授的名字，第二个模板从第1个模板的集合中去除了在1995年秋季教过一门课的那些教授。这个查询看起来很简单，但是也有缺陷。前面我们说过，QBE中的所有变量都是由∃隐式量化的，模板中的空白列事实上是由系统产生的变量（也是由∃隐式量化的）填充的。对于否定的样例元组则不是这样！事实上，如果系统产生的变量（如隐式出现在列CrsCode中的_Crs123）是存在量化的，我们将会有下面的查询:

```
{Name | ∃Id ∃DeptId ∃CrsCode(
        PROFESSOR(Id, Name, DeptId) AND
        NOT TEACHING(Id, CrsCode, 'F1995'))}
```

如果有一门课，比如是MGT123，则行<111111111, MGT123, F1995>不在TEACHING中，那么这个查询返回一个Id，即11111111。也就是说，它返回在1995年秋季没有教授某门课的所有教授的Id——这与我们提交过的查询是不同的。正确的公式是

```
{Name | ∃Id ∃DeptId ∀CrsCode(
        PROFESSOR(Id, Name, DeptId) AND
        NOT TEACHING(Id, CrsCode, 'F1995'))}
```

(7.13)

也就是说，_Crs123希望的量化是全称的。

否定元组中对变量进行正确量化的不确定性是QBE中语义问题的一个来源。最终决定，假设否定元组中所有系统生成的变量都隐式地由∀来量化。做出这个决定的原因是我们认为就像在前面的例子中那样，在大多数情况下用户想要的是全称量化。

但是，这个简单的规则仍然没有完全解决这个问题，因为我们已经知道∃和∀不能互换位置。所以存在一个安排量化前缀的顺序问题。例如，在（7.13）中我们以不同的顺序安排量词，并得到了一个不同的查询：

```
{Name | ∀CrsCode ∃Id ∃DeptId(
        PROFESSOR(Id, Name, DeptId) AND
        NOT TEACHING(Id, CrsCode, 'F1995'))}
```
(7.14)

它找出具有下面性质的所有教授的名字：对于每门课来说，存在一个教授，他没有教过这门特定的课程。

最终解决了QBE的语义问题，这样量化前缀按存在量词优先的原则来安排顺序。因而之前的QBE查询必须解释为（7.13）那样。

既然我们理解了QBE中的否定，关于每个学生都选修过的课程的查询就可以按如下方式表达了。首先，我们要找到至少有一个学生没有选修过的课程。我们称包含所有这样的课程的关系为NOTANSWER，它的构造方法如下：

NOTANSWER	CrsCode
I.	_MGT111

COURSE	CrsCode	CrsName	Descr
	_MGT111		

STUDENT	Id	Name	Status	Address
	_123456789			

TRANSCRIPT	StudId	CrsCode	Semester	Grade
¬	_123456789	_MGT111		

我们又一次用到了插入运算符来指定我们想插入到关系NOTANSWER中的元组。因而，NOTANSWER的内容由下面的DRC查询来描述。

```
{CrsCode | ∃CrsName ∃Descr ∃Id ∃Name ∃Status ∃Address
    ∀Semester ∀Grade(
        COURSE(CrsCode, CrsName, Descr) AND
            STUDENT(Id, Name, Status, Address) AND
            NOT TRANSCRIPT(Id, CrsCode, Semester, Grade))}
```
(7.15)

最后，为了得到最初查询的结果，我们要从COURSE中减去NOTANSWER。

NOTANSWER	CrsCode
¬	_MGT111

COURSE	CrsCode	CrsName	Descr
	_MGT111	P.	

尽管我们已经介绍了QBE大多数的特性，但是关于QBE还有一些我们没有讨论到的方面（或者只是简略地介绍了一下）。例如，QBE有非常方便的数据定义子语言，其可视化的外观

和查询子语言保持一致。它也支持聚合函数，如统计、求平均值，这些聚合函数以运算符（COUNT、AVG等）的形式提供给用户使用。在6.2节中详细讨论过聚合函数，但是该节重点介绍的是SQL，而不是QBE。最后，我们已经看到了运算符I.，它是QBE数据操纵子语言的一部分，也可以用类似的运算符来删除和修改元组。

4. PC数据库

从概念上来说，如果你已经看到过一种可视化查询语言，你就能了解其他的可视化查询语言了。QBE和Borland Paradox或Microsoft Access之间的主要差异在于图形界面的细节的不同。图7-1描绘了用Access的“设计模式”指定的查询。查询返回在1995年秋季开设的所有课程以及教这些课的教授的名字。

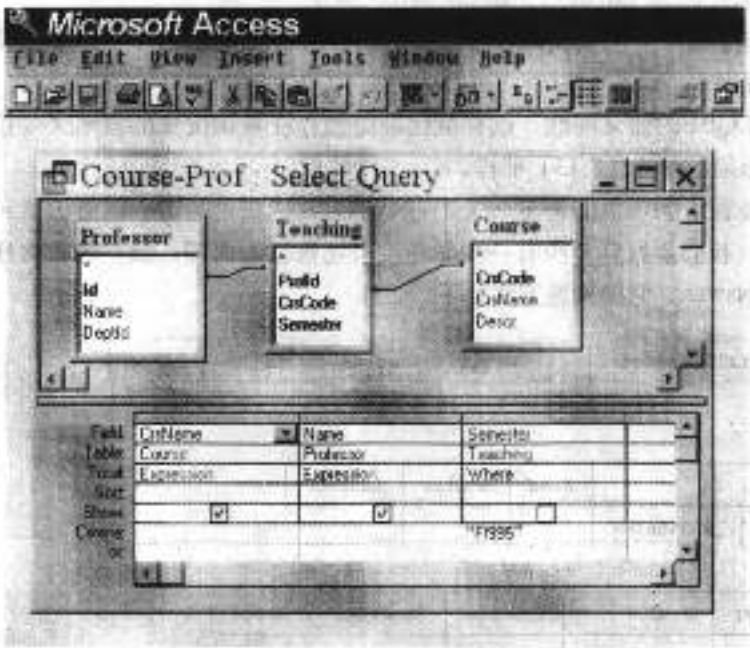


图7-1　Microsoft Access中的可视化查询

Access努力试图来隐藏其关系演算的本源。为了指定等值联结，用户不是使用变量，而是把一个属性拖到另一个属性那里去，创建如图所示的图形链接，用图形链接来表示联结条件（如PROFESSOR.Id = TEACHING.ProfId）。Access有类似于QBE的条件框的表示（如准则TEACHING.Semester = 'F1995'）。在Access中比在QBE中会更多地要用到这个框，特别是指定QBE中要借助于像P.这样的运算符来完成工作（见图7-1中标记为“Show:”的线）。

总之，QBE比Access更为灵活，通常对于熟悉关系语言的数据库专家也更为直观。然而，对于拖放的生成来说，Access的设计模式较容易使用，至少对于简单查询是这样。

7.5　关系代数和关系演算之间的联系

我们已经给出了关系代数和两种关系演算的详细描述。我们也看到一些查询在一种语言中看起来比在另一种语言中要简单，我们会很自然地提出这样的问题：是否某些查询只能用

其中的一种语言来编写？

注意，这并不是没有价值的理论问题。回忆一下数据库管理系统使用关系代数作为SQL查询翻译的中间代码，进而进行优化。如果TRC能够比代数表达更多的查询，那么一些SQL查询就可能没有代数的对应体，因而就不可实现了。类似的，如果DRC能够比TRC表达更多的查询，那么它就能够成为奠定数据库语言基础的更好平台。

上述问题的答案是，这三种语言有着完全相同的表达能力：能以一种语言提交的查询几乎都能以另一种语言来提交。事实上，演算的表达能力更强一些，但是在实际的数据库查询中很少用到那些额外的表达。

为了进行说明，考虑一下查询{T | NOT Q(T)}，它要求返回不在关系**Q**中的所有元组。从直观上看，这并不是通常针对数据库中的数据的查询。但是对于我们讨论来说，更重要的是它有一种非常奇怪的性质。例如，假设**Q**是包含一组学生名字的关系，名字的域是CHAR (20)。那么{T | NOT Q (T)}的结果就是除了**Q**中的那些字符串以外所有最多由20个字符构成的字符串的集合。现在假设名字的域后来成为所有最多由30个字符构成的字符串的集合。那么即使数据库实例没有变化，上面查询的答案也发生了改变。

在查询条件包含析取的时候可能出现查询答案依赖于域的情况。例如{T | ∃S(Q(T)OR R(S))}在**R**是空关系的时候，将返回**Q**中所有元组的集合。然而，如果**R**即使只包含一个元组，那么这个查询的答案就是所有可能的n元元组的集合，这里的n是**Q**中属性的数目。在这种情况下，查询的答案依赖于**Q**中属性的域。

当域变化而结果仍然保持不变的数据库查询称为**域独立的**（domain-independent）。练习7.4说明了用关系代数表达的查询总是域独立的。然而，我们刚刚看到可以用关系演算来表达域独立的查询。即使这样，也可以得到如下的结论

> *以两种演算中任何一种方法表达的每个域独立的查询都可以翻译成用关系代数编写的等价查询，反之亦然。*

这个结论的第一部分并不能明显地看出其正确性。在[Ullman 1988]中有它的证明。然而，另一个方向，从代数到演算的翻译是很容易的，也是一个很好的练习，因为它显示了关系代数的运算符是如何与关系演算的构造成分相关联的。

- *选择*

  代数：$\sigma_{Condition}(\mathbf{R})$

  TRC：{T | **R**(T) AND $Condition_1$}

  DRC：{$X_1, \cdots, X_n$ | **R** ($X_1, \cdots, X_n$) AND $Condition_2$}

在TRC中，$Condition_1$是通过用T的适当分量替换属性而从$Condition$得到的。例如，如果$Condition$是A = B AND C = d，这里A、B、C是属性，d是常数，那么$Condition_1$就是T.A = T.B AND T.C = d。

在DRC中，$Condition_2$是通过用适当变量替换出现在$Condition$中的属性而得到的。例如，如果A是R的第一个属性，B是第二个属性，C是第三个属性，那么$Condition_2$就是$X_1 = X_2$ AND X_3 = d。

• 投影

代数: $\pi_{A,B,C}(\mathbf{R})$

TRC: {T.A, T.B, T.C | **R** (T) }

DRC: {X, Y, Z | ∃V∃W (**R** (X, Y, Z, V, W)) }

这里我们假设**R**有五个属性，并且A、B、C是前三个属性。

• 笛卡儿积

代数: **R** × **S**

TRC: {T.A, T.B, T.C, V.D, V.E | **R** (T) AND **S** (V) }

为了明确起见，我们假设**R**有属性A、B、C，**S**有属性D、E。

DRC: {X, Y, Z, V, W | **R** (X, Y, Z) AND **S** (V, W) }

• 并

代数: **R** ∪ **S**

TRC: {T | **R** (T) OR **S** (T) }

DRC: { $X_1, \cdots, X_n$ | **R** ($X_1, \cdots, X_n$) OR **S** ($X_1, \cdots, X_n$)}

注意因为**R**和**S**必须是并相容的，这样∪运算符才有意义，所以两个关系模式必须具有相同的元数。

• 差集

代数: **R** − **S**

TRC: {T | **R** (T) AND (NOT **S** (T))}

DRC: 练习7.6

• 如果不讨论除法运算符，关系代数的讨论是不完整的。

代数: **R/S**，这里**R**具有属性A、B，**S**只有属性B。

TRC: {T.A | **R** (T) AND ∀ X ∈ **S** (∃Y ∈ **R** (Y.B = X.B AND Y.A = T.A))}

DRC: 练习7.6

7.6 SQL:1999中的递归查询

1. 关系查询语言的局限性

本章和前面的几章中的许多例子说明了关系查询语言的强大表达能力。然而这些语言的表达能力并不足以使程序员用SQL来编写全部的应用程序。令人惊奇的是，SQL甚至不能用来表达一些非常常见的查询，如一门课是否是另一门课的（可能是非直接的）预备课程。

为了说明这个问题，考虑一下图7-2中的关系PREREQ。假设我们想知道CS113是否是CS632的预备课程。我们可以利用关系代数试着用如下方法解决这个问题。让 PREREQ_2 (Crs, PreCrs) 表示表达式

$$\pi_{\text{Crs,PreCrs}}((\text{PREREQ} \bowtie_{\text{PreCrs=Crs}} \text{PREREQ})[\text{Crs,P1,C2,PreCrs}]) \cup \text{PREREQ}$$

PREREQ	Crs	PreCrs
	CS632	CS532
	CS505	CS213
	CS532	CS305
	CS305	CS213
	CS305	CS214
	CS214	CS114
	CS114	CS113
	CS305	CS220

图7-2　预备课程列表

这里 [Crs, P1, C2, PreCrs] 像前面一样对属性重命名。我们现在可以计算表达式 $\sigma_{Crs = 'CS632' \text{ AND } PreCrs = 'CS113'}(PREREQ_2)$，并看看结果是否不为空。如果不为空，CS113就是CS632的预备课程。然而，容易验证上面的表达式在我们这个具体的情况下计算得到一个空关系，所以我们还没有做完。

我们试着计算下面的表达式来看看CS113是否是CS632的预备课程，我们把它表示成 $PREREQ_3$ (Crs, PreCrs)。

$$\pi_{Crs,PreCrs}((PREREQ \bowtie_{PreCrs=Crs} PREREQ_2)[Crs,P1,C2,PreCrs]) \cup PREREQ_2$$

如果 $\sigma_{Crs = 'CS632' \text{ AND } PreCrs = 'CS113'}(PREREQ_3)$ 不为空，那么就证实了我们的假设。然而，这个表达式又一次为空，所以我们还没有在这两门课之间建立起间接的预备课程关系。如果我们给关系PREREQ赋予别名 $PREREQ_1$，我们可以看出在上面过程中每次重复的模式看起来都如下所示：

$$PREREQ_{i+1} = \pi_{Crs,PreCrs}((PREREQ \bowtie_{PreCrs=Crs} PREREQ_i)[Crs,P1,C2,PreCrs]) \cup PREREQ_i \quad (7.16)$$

我们继续用同样的方法创建 $PREREQ_4$、$PREREQ_5$ 等等。容易直接验证 $\sigma_{Crs = 'CS632' \text{ AND } PreCrs = 'CS113'}(PREREQ_5)$ 不为空，因而这两门课具有预备课程关系。

注意，上面的过程也可以用来提供否定的答案。例如，为了验证CS220不是CS505的直接或间接的预备课程，我们可以计算 $PREREQ_2$、$PREREQ_3$ 等等，来看看在每种情况下应用选择 $\sigma_{Crs = 'CS505' \text{ AND } PreCrs = 'CS220'}$ 都产生空的结果。我们怎么知道 $\sigma_{Crs = 'CS505' \text{ AND } PreCrs = 'CS220'}(PREREQ_{1000})$ 是不是为空呢？这很简单：$PREREQ_5$、$PREREQ_6$ 等等都是相等的，即用于构造这些关系的联结运算符在做几次联结之后就停止产生新的元组了。

鉴于上面的讨论，看起来只用关系代数就可以找出课程的间接预备课程。在我们的例子中，我们所要做的就是检查 $PREREQ_5$ 的内容。然而，在经过更为仔细的考虑之后，问题就变得更为复杂了。如果PREREQ关系有元组<CS220, CS214>而不是<CS305, CS214>，那将会怎么样呢？在这种情况下，会还需要进行一次（或多次迭代）以建立CS632和CS113之间的间接预备课程关系；即 $\sigma_{Crs = 'CS632' \text{ AND } PreCrs = 'CS113'}(PREREQ_5)$ 将仍然为空，但是 $\sigma_{Crs = 'CS632' \text{ AND } PreCrs = 'CS113'}(PREREQ_6)$ 不为空。

换句话说，达到稳定状态表达式（7.16）所需要重复的次数（随后的联结没有影响）是*由数据决定的*，不能事先预计。因此，对于像 $\sigma_{Crs = 'CS632' \text{ AND } PreCrs = 'CS113'}(PREREQ_5)$ 这样的表达式，不容易判断对于关系PREREQ的所有合法内容，CS113是否是CS632的间接预备课程。事实上，[Aho and Ullman1979]中说明了*关系表达式不能提供这样的答案*！这个结果说明，像关系代数、关系演算和SQL这样的关系语言的表达能力是有限的。事实上如后面的练习7.14所示，检查预备课程具有多项式级的时间复杂度，这意味着SQL不能够表达某些具有多项式级时间复杂度的查询。这个理论结果的直接实际后果是不能完全用SQL来编写即使很简单的数据库应用程序。这就是在现实世界中SQL为什么不用在提供一般应用逻辑的宿主语言（像C或Java）之内的一个原因。（第10章将会解释这是怎么做的。）

等式（7.16）被称为是**循环的**（recurrent），以它的术语表达的查询称为是**递归的**（recursive）。可以用如下方法计算递归查询：从某个已知的初始值（在我们的例子中就是

PREREQ）开始重复应用循环等式，直到达到一个稳定的状态为止，即直到对于某个N来说PREREQ_{N+1}等于PREREQ_N为止。换句话说，关于预备课程的查询的答案就是循环等式（7.16）的稳定状态。因而，[Aho and Ullman 1979]中上述结果的另一个解释是通常不能用关系代数表达式来表示循环等式的稳定状态。

2. SQL中的递归

探讨递归查询处理过程的数据库研究领域称为**演绎数据库**（deductive database），关于这个主题有很多文献（其简介见[Ramakrishnan and Ullman 1995]）。最终，SQL标准组认识到需要处理递归查询，并在SQL:1999中增加了适当的扩展。我们下一步将讨论这些扩展。

SQL:1999不使用（7.16）那样的循环等式来定义递归查询，因为这些等式与判定这样的查询的过程图相对应。与最初的观点相一致，SQL:1999是声明性地指定要检索的内容，而不是过程性地指定如何检索。不过，可以从语法上很清楚地看出循环等式和SQL:1999表达递归查询的方法之间的联系，特别是对于递归视图。例如，下面的视图指定了所有课程－预备课程对。

```
CREATE RECURSIVE VIEW INDIRECTPREREQVIEW(Crs, PreCrs) AS
SELECT * FROM PREREQ
UNION
SELECT P.Crs, I.PreCrs
FROM PREREQ P, INDIRECTPREREQVIEW I
WHERE P.PreCrs = I.Crs
```

普通视图和递归视图之间的差异是递归视图的定义包含了两个独立的部分。

- 非递归子查询　在我们的例子中，就是第一个SELECT语句（在UNION运算符之上）。它不能包含对要定义的视图关系的引用。
- 递归部分　这部分包含了第二个子查询（在UNION运算符之下）。和非递归子查询不同，它引用了关系INDIRECTPREREQVIEW，即正在由CREATE RECURSIVE VIEW语句定义的那个视图。

现在应该清楚，为什么这样的视图被称为是递归的：它们的定义看起来是循环的。然而，它们的内容有很好的定义，并根据前面提到的循环等式可以清楚地理解它们。

递归视图定义中非递归子查询的目的是指定用于循环等式中的递归关系的最初内容。在这个例子中，这个子查询说明INDIRECTPREREQVIEW的最初内容是关系PREREQ的内容。我们把这些内容表示为$\text{INDIRECTPREREQVIEW}_1$。

递归部分指定了实际的循环等式。它说明，为了得到INDIRECTPREREQVIEW内容的下一个近似，必须假定当前的近似来判定这个查询。换句话说，$\text{INDIRECTPREREQVIEW}_2$是由$\text{INDIRECTPREREQVIEW}_1$和下面查询的结果做并集而计算得到的关系。

```
SELECT P.Crs, I.PreCrs
FROM PREREQ P, INDIRECTPREREQVIEW1 I
WHERE P.PreCrs = I.Crs
```

这里我们用前面的近似$\text{INDIRECTPREREQVIEW}_1$来计算$\text{INDIRECTPREREQVIEW}_2$的值。这个循环等式的稳定状态就被认为是这个递归视图的含义。在这个例子中，当把INDIRECTPREREQVIEW的当前内容和最初的PREREQ关系做联结时，不再产生新的课程－预备课程对时，也就是说，

当下面查询的结果

```
SELECT P.Crs, I.PreCrs
FROM PREREQ P, INDIRECTPREREQVIEW_N I
WHERE P.PreCrs = I.Crs
```

包含在INDIRECTPREREQVIEW$_N$（$N>0$）中时，就达到了稳定状态。

在定义了递归视图之后，我们现在就可以用通常的SELECT语句来进行查询。例如，为了确定CS113是否是CS632的间接预备课程，我们可以编写如下查询：

```
SELECT *
FROM INDIRECTPREREQVIEW I                                    (7.17)
WHERE I.PreCrs = 'CS113' AND I.Crs = 'CS632'
```

SQL:1999也提供了不依赖于视图的递归查询语法。其主体思想是类似的：递归查询包含两部分：递归关系的定义和对这个关系的查询。第一部分的语法和我们之前看到的递归视图定义很相近，第二部分的语法就是普通的SQL（非递归的）查询。例如，关于CS113和CS632之间的间接预备课程关系的查询可以表达如下：

```
WITH RECURSIVE INDIRECTPREREQQUERY(Crs, PreCrs) AS
          ((SELECT * FROM PREREQ)
           UNION
           (SELECT P.Crs, I.PreCrs
            FROM PREREQ P, INDIRECTPREREQQUERY I
            WHERE P.PreCrs = I.Crs))
SELECT *
FROM INDIRECTPREREQQUERY I
WHERE I.PreCrs = 'CS113' AND I.Crs = 'CS632'
```

注意，这个查询最上面那一部分几乎和递归视图INDIRECTPREREQVIEW的定义完全相同。唯一的实质性区别是，视图定义是保存在系统目录中，以后在其他查询中可以重用，然而查询INDIRECTPREREQQUERY的定义在处理后就被系统丢弃了。上面查询的最下面一部分与针对视图INDIRECTPREREQVIEW的查询（7.17）相对应。

3. *互递归查询*

到目前为止，递归查询考虑的传递闭包还没有说明SQL:1999中递归查询功能的全部能力。下面考虑一个较复杂的例子，我们想（完全是出于好奇）找出中间间隔奇数门课程的预备课程。我们可以通过定义关系ODDPREREQ来表达这个查询，这个关系与另一个关系EVENPREREQ是**互递归的**（mutually recursive，即每个关系都要根据另一个关系来定义）。互递归可以借助于WITH语句表达如下：

```
WITH
    RECURSIVE ODDPREREQ(Crs, PreCrs) AS
          ((SELECT * FROM PREREQ)
           UNION
           (SELECT P.Crs, E.PreCrs
            FROM PREREQ P, EVENPREREQ E
            WHERE P.PreCrs = E.Crs)),
    RECURSIVE EVENPREREQ(Crs, PreCrs) AS
          (SELECT  P.Crs, O.PreCrs
           FROM PREREQ P, ODDPREREQ O
           WHERE P.PreCrs = O.Crs)
SELECT * FROM ODDPREREQ
```

WITH语句在这个查询中定义了两个临时关系OddPrereq和EvenPrereq。第一个关系有非递归子查询，它指出每个直接预备课程也是一门奇数预备课程。OddPrereq定义的递归部分指出奇数预备课程的其余部分是通过把Prereq关系和包含所有偶数预备课程的关系EvenPrereq进行联结而得到的。后一个关系是用第二个递归查询来定义的。这个查询没有非递归的部分，这意味着EvenPrereq的初始值是空关系。EvenPrereq的后继近似值是通过把Prereq和OddPrereq进行联结而得到的。因此，OddPrereq和EvenPrereq是彼此依赖的。

4. 递归使用的限制

SQL:1999在使用递归时有许多限制，看起来好像只是为减少销售者为了解释深奥的特性需要付出的代价。然而，一些限制是由于技术方面的考虑和保持较低的判定查询的时间复杂度而引起的。这个类别中最重要的限制与否定有关，它是用关键字EXCEPT和NOT来表达的。为了进行说明，假设我们想找出所有*真正的*奇数预备课程，即那些不可以是偶数的预备课程（对于一门预备课程来说，它可以既为奇数也可以为偶数，因为任何一对课程都可能是由多条预备课程链连接的，这样的链可能有不同的长度）。我们可以*试着*用以下方式表达这个查询：

```
WITH
    RECURSIVE OddPrereq(Crs, PreCrs) AS
        ((SELECT * FROM Prereq)
          UNION
         (SELECT P.Crs, E.PreCrs
          FROM Prereq P, EvenPrereq E
          WHERE P.PreCrs = E.Crs)
         EXCEPT
         (SELECT * FROM EvenPrereq)),
    RECURSIVE EvenPrereq(Crs, PreCrs) AS
        (SELECT P.Crs, O.PreCrs
         FROM Prereq P, OddPrereq O
         WHERE P.PreCrs = O.Crs )
SELECT * FROM OddPrereq
```

(7.18)

这个查询的问题与在OddPrereq定义的递归部分中去除EvenPrereq关系有关。就像我们之前看到的那样，这两个关系是彼此依赖的。所以，为了知道哪些元组在OddPrereq中，我们应该知道哪些元组*不在*EvenPrereq中，而这就需要知道哪些元组*在*EvenPrereq中。但是后者又需要知道哪些元组*在*OddPrereq中——这又回到了前面的问题。

这个问题通常可接受的解决方案（并且是用在SQL:1999中的解决方案）是要求否定（像EXCEPT和NOT）的使用是**分层的**（stratified）。也就是说，如果关系**P**的定义要和知道关系**Q**的分量，那么**Q**的定义一定*不能*依赖于**P**（或它的分量）。特别的，**P**不能依赖于它自己的分量。注意（练习7.15），非递归查询中否定的每次使用都是分层的，所以SQL-92中不必有这个限制。

查询（7.18）中否定的使用不是分层的，所以它在SQL:1999中是不合法的。为了构造检索所有真正奇数预备课程的合法查询，我们必须打破OddPrereq和EvenPrereq定义中的互递归。

```
WITH
    RECURSIVE OddPrereq(Crs, PreCrs) AS
        ((SELECT * FROM Prereq)
         UNION
         (SELECT P.Crs, E.PreCrs
          FROM Prereq P, Prereq P1, OddPrereq O
```

```
            WHERE P.PreCrs = P1.Crs AND P1.PreCrs = O.Crs)
          EXCEPT
          (SELECT * FROM EVENPREREQ)),
    RECURSIVE EVENPREREQ(Crs, PreCrs) AS
        ((SELECT P.Crs, P1.PreCrs
          FROM PREREQ P, PREREQ P1
          WHERE P.PreCrs = P1.Crs
          UNION
          (SELECT P.Crs, E.PreCrs
           FROM PREREQ P, PREREQ P1, EVENPREREQ E
           WHERE P.PreCrs=P1.Crs AND P1.PreCrs = E.Crs))
SELECT * FROM ODDPREREQ
```

在这个查询中，两个关系ODDPREREQ和EVENPREREQ仍然是递归的，但是不再相互依赖了。ODDPREREQ依赖于EVENPREREQ的分量（因为它定义中的EXCEPT子句），但是EVENPREREQ并不依赖于ODDPREREQ。因此，在这个查询中否定的使用是分层的，这个查询是合法的。

7.7 参考书目

[Codd 1972]中引入了元组关系演算，并把它视为研究关系代数作为查询语言的表达能力的理论工具。[Date 1992]说明了TRC如何有效地用作构造复杂SQL查询的中间语言。[Lacroix and Pirotte 1977]中提出域关系演算，并把它作为数据库查询的语言。[Zloof 1975]中引入了样例查询。用两种演算之一编写的域独立查询和关系代数的等价证明要归功于Codd，在[Ullman 1988]中可以找到这个证明。在[Van Gelder and Topor 1991, Kifer 1988, Topor and Sonenberg 1988, Avron and Hirshfeld 1994]等文献中研究了域独立及相关的问题。

传递闭包查询不能用关系代数表达的证明归功于[Aho and Ullman 1979]。[Ullman 1988, Abiteboul et al. 1995]中广泛地讨论了演绎数据库，[Ramakrishnan et al. 1994, Sagonas et al. 1994, Vaghani et al. 1994]中描述了许多演绎数据库系统的研究原型。[Apt et al.1988]中引入了分层否定。

7.8 练习

7.1 a. 解释为什么元组关系演算被称为声明性的，而关系代数被称为过程性的。

b. 解释为什么尽管SQL是声明性的，但是关系数据库管理系统中的查询优化器把SQL语句翻译成过程性的关系代数。

7.2 用语言来叙述下面每个表达式的含义（这里Took的含义很明显）。比如，其中一个表达式的答案是“每个学生至少选修过一门课程。”

a. $\exists S \in$ STUDENT ($\forall$C $\in$ COURSE TOOK(S, C))

b. $\forall S \in$ STUDENT ($\exists$C $\in$ COURSE TOOK(S, C))

c. $\exists C \in$ COURSE ($\forall$S $\in$ STUDENT TOOK(S, C))

d. $\forall C \in$ COURSE ($\exists$S $\in$ STUDENT TOOK(S, C))

**7.3 证明元组关系演算中的任何查询在域关系演算中都有等价的查询，反之亦然。

*7.4 证明关系代数查询是域独立的。提示：对关系代数表达式的结构进行归纳。

7.5 考虑与图5-5中的IsA层次结构对应的关系模式。假设这个模式对每个实体都有一个关系。（参考5.4节来更新你对这种IsA层次结构到关系模型的翻译的认识。）用元组关系演算和域关系演算来写出下面的查询:

a. 找出计算机科学专业所有大二学生的名字。

b. 找出计算机科学专业所有学生的名字。

c. 找出满足下面要求的所有系：存在某些技术员具有任何其他技术员（本系或外系）所具有的每门技术。

7.6 写出一个等价于下面代数表达式的域关系演算查询：

a. **R** − **S**

b. **R**/**S**，这里关系**R**有属性*A*、*B*，**S**只有一个属性*B*。

7.7 利用图4-4的模式，用关系代数、元组关系演算、域关系演算和QBE表达下面的每个查询。

a. 找出由属于EE系或者MGT系的教授讲授的所有课程。

b. 列出在1997年春季和1998年秋季都选修过课程的所有学生的名字。

c. 列出所有选修过至少两个来自不同系的教授讲授的课程的学生的名字。

d. 找出在MGT系被所有学生选修过的所有课程。

*e. 找出满足下面要求的所有系：有一个教授教过那个系开设的所有课程。

比较使用两种演算、QBE和SQL在构造上面查询时的难易程度。

7.8 如果你有一份Paradox、Access或者类似的数据库管理系统的拷贝，用附带的可视化语言来设计上面的查询。

*7.9 用TRC和DRC来写出练习6.17的查询。

*7.10 用TRC和DRC来写出练习6.19的查询。

*7.11 用TRC和DRC来写出练习6.22的查询。

a. 找出对代理号为007的代理所负责的每一所房屋都感兴趣的所有客户。

b. 把前面的查询当作视图，检索形如<*feature*, *customer*>的一组元组，这里结果中的每个元组说明了一个特性和要求这一特性的一个客户，而且：

- 只考虑对代理007负责的每一个房屋都感兴趣的客户。
- 对“feature”感兴趣的客户的数目要大于2。（如果这个数目小于等于2，那么就不把相应的元组<*feature*, *customer*>加进结果。）

不能方便地用TRC或者DRC来表达这一部分查询，因为它们缺少计数操作符。不过，完成上述工作仍然是可能的，而且也不难。

7.12 用TRC和DRC写出SQL查询（6.8）和查询（6.9）。

7.13 阐述SQL操作符EXISTS和TRC量词∃之间的逻辑关系。具体来说，请用TRC来表达查询（6.23）。

*7.14 说明计算PREREQ的传递闭包的迭代过程在有限次数的步骤之后会中止。假设略微修改这个过程以避免冗余的计算，那么说明这个过程可以在多项式级的时间里计算传递闭包。

7.15 说明在SQL-92查询中，否定的每次使用都是分层的。

7.16 考虑列出城市间所有直飞航线的关系DIRECTFLIGHT(StartCity, DestinationCity)。使用SQL:1999的递归功能来编写一个查询，它找出所有的配对<$city_1$, $city_2$>，要求从$city_1$到$city_2$有一个间接的航线，并且在二者之间至少有两站。

7.17 使用SQL:1999的递归功能来表达所谓的“同辈”的查询：给定PARENT关系，找出所有的人员配对，每一对人都有相同的祖先，并且从她到这一祖先隔了相同的代数。（例如，一个小孩相对她的父亲来说隔了一代，相对她的祖父亲来说隔了两代。）

7.18 考虑下面的材料单问题：数据库有一个关系SUBPART(Part, Subpart, Quantity)，它指出了每个零件直接需要的子零件以及需要的数量。例如，SUBPART(mounting_assembly, screw, 4)表示安装物包含了4个螺丝钉。为了简单起见，假设没有子零件的零件（原子零件）具有NULL作为唯一的子零件（例如，SUBPART(screw, NULL, 0)）。写出一个递归查询来列出所有的零件，并且对于每个零件来说，给出它拥有的原始子零件的数目。

第8章 数据库设计II：关系规范化理论

E-R方法可以很好地控制现实世界建模的复杂性。然而，它只是一组指导原则，掌握这些原则需要有相当的专业知识和直觉；而且，对于相同的情况它可以产生不同的设计方案。但是，E-R方法并不包括一些评价标准或工具，来帮助我们选择设计方案和对方案进行改进。在本章中，我们将介绍*关系规范化理论*（relational normalization theory），它包括一些概念和算法。这些概念和算法可以帮助我们对通过E-R方法得到的设计进行评价与改进。

在规范化理论中使用的主要工具是*函数依赖*（functional dependency）的概念（以及更低程度的*联结依赖*（join dependency）的概念）。函数依赖是对E-R方法中键依赖的一种推广，而联结依赖在E-R方法中没有相应的方法。设计者使用这两种依赖来发现在E-R设计中把两个不同实体类型的属性放到同一个关系模式中这种不正常的情况。这些情况我们用*范式*（normal form）来刻画，它起源于*规范化理论*这个术语。规范化理论通过使用*分解*（decomposition）使关系成为合适的范式；分解是指分离那些把不相关实体类型混在一起的模式。由于分解在关系设计扮演着中心角色，因此我们将要讨论的技术有时也称为*关系分解理论*（relational decomposition theory）。

8.1 冗余所带来的问题

示例是理解基于E-R方法进行关系设计所具有的潜在问题的最好方法。考虑（5.1）的语句：CREATE TABLE PERSON。这个关系模式是通过直接转换E-R图(图5-1)而得来的。对于这个转换，我们首先可以发现的错误是：SSN并不是PERSON关系的键，而(SSN ,Hobby)的组合才是键。换句话说，属性SSN并不能唯一决定PERSON关系中的元组，尽管它确实唯一决定了PERSON实体集中的实体。这不仅与我们的直觉相反，而且它对PERSON关系模式的实例也会产生一些不良的影响。

为说明这一点，我们仔细看一下图5-2中的关系实例。注意，John Doe 和 Mary Doe 均由多个元组表示，并且他们的地址，姓名和Id也多次出现。相同信息的冗余存储问题在这里表现得很明显。然而，空间的浪费并不是关键问题。真正的问题是，当对数据库进行更新时，我们必须保持相同数据的冗余拷贝之间的一致性，并且能使更新有效进行。特别地，我们应该考虑如下问题：

更新异常 如果John Doe搬到1 Hill Top Dr，更新图5-2中的关系时还必须修改描述John Doe实体的两个元组的地址。

插入异常 假如我们决定把Homer Simpson添加到PERSON关系中去，但在Homer的信息表中并没有说明他的任何爱好。解决此问题的一个方法是加入元组<023456789, Homer Simpson, Fox 5 TV, NULL>，即将缺失的字段用NULL填充。然而，Hobby字段是主键的一部分，大部分DBMS不允许主键出现空值。为什么？一方面，DBMS通常在主键上维护一个索引，而它并

不清楚索引应该如何引用空值。假定这个问题可以解决，有一个请求要求插入<023456789, Homer Simpson, Fox 5 TV, acting>。这个新的元组是应该被添加进去，还是应该替换现有的元组<023456789, Homer Simpson, Fox 5 TV, NULL>？有人很可能选择替换它，因为人们通常并不认为爱好是一个人特征的一部分。然而，计算机如何知道主键为<111111111,NULL>和<111111111,acting>的元组指同一个实体？(关于哪个元组来自哪个实体的信息在转换中已经丢失！）冗余是产生这种不确定性的根源：如果Homer最多只被一个元组描述，那么只有表演这门课程是符合要求的。

删除异常　假设Homer Simpson不再对表演感兴趣。我们如何删除这个业余爱好？当然，我们可以删除与Homer表演兴趣相关的元组。然而，由于只有一个元组引用Homer（参见图5-2），这会把关于Homer的Id和地址这些有用的信息也丢掉。为避免这种信息的丢失，我们可以把acting替换为NULL。这又会引起主键属性为空的问题。在这里，冗余又一次成为元凶。如果只用一个元组就可以描述Homer，那么属性Hobby将不会是键的一部分。

为方便起见，我们有时用术语“更新异常”来指代上面提到的所有异常类型。

8.2 分解

冗余所引起的问题（浪费空间和异常）可以使用如下的技术进行修正。不使用单个关系来描述所有关于某个人的信息，而用两个单独的关系模式。

PERSON1 (SSN, Name, Address)
HOBBY (SSN, Hobby)　　(8.1)

对图5-2中的关系进行投影可以得到图8-1所示的模式。这个新的设计有如下性质：

SSN	Name	Address
111111111	John Doe	123 Main St.
555666777	Mary Doe	7 Lake Dr.
987654321	Bart Simpson	Fox 5 TV

a) PERSON1

SSN	Hobby
111111111	stamps
111111111	hiking
111111111	coins
555666777	hiking
555666777	skating
987654321	acting

b) HOBBY

图8-1　图5-2中关系PERSON的分解

1) 尽管在所有人都有业余爱好的情况下，关系PERSON在缺失业余爱好的情况下无法描述个人信息，但它是PERSON1和HOBBY的自然联结。这个性质并不是我们例子的制品，即在某种常理假设下（即SSN唯一决定一个人的姓名和地址），对图5-2中的模式，任何关系**r**都等同于**r**在（8.1）中两个关系模式投影的自然联结。这个性质称为*无损性*（losslessness），它将会在8.6.1节进行讨论。这意味着我们的分解保留了代表PERSON关系的原始信息。

2) 在图5-2中原始关系的冗余和更新异常已经消除。存储次数多于一次的唯一项是SSN，它是类型PERSON的实体的标识符。因此，现在对地址、姓名、业余爱好的改变只会影响一个元组。同样，把Bart Simpson的爱好从数据库删除并不会删除他的地址信息，也不要求我们对

空值产生依赖。由于我们可以单独地添加人和业余爱好，因此插入异常也消除了。

注意，新的设计同样有一定的冗余，并且我们在某些情况下可能还要使用空值。首先，由于我们使用SSN作为元组的标识符，每个SSN可以出现多次，而且所有这些出现必须在数据库中保持一致。因此一致性维护问题并没有完全消除。然而，如果标识符是非动态的（比如SSN不会经常改变），那么一致性的维护会简单很多。其次，考虑某种情况：添加一个人到关系PERSON1但却不知道他的地址。很明显，即使对新的设计，我们不得不为相应的元组在字段Address中插入NULL。然而，Address并不是主键的一部分，因此这里使用NULL不会有太坏的影响（使用Address属性来联结PERSON1仍会有困难）。

并不是所有的分解都是等价的，理解这一点很重要。事实上，大部分分解尽管在消除冗余方面做的很好，但其并没有什么意义。分解

```
SSN(SSN)
NAME(Name)
ADDRESS(Address)                                    (8.2)
HOBBY(Hobby)
```

是消除最终冗余的分解。对图5-2的关系的投影会产生一个每个值只出现一次的数据库，如图8-2所示。但是，这个新的数据库几乎没有任何有用信息；我们不可能再知道John Doe住在哪里，或者谁把集邮作为业余爱好。这种情况与图8-1中的分解形成鲜明对比。对图8-1，通过自然联结我们能够恢复原始关系所表示的信息。

SSN
111111111
555666777
987654321

Name
John Doe
Mary Doe
Bart Simpson

Address
123 Main St.
7 Lake Dr.
Fox 5 TV

Hobby
stamps
hiking
coins
skating
acting

图8-2　最终的分解

精化模式的需求

从实体类型PERSON到关系模型的转换说明不能只依靠E-R方法来设计数据库模式。而且，例子PERSON所暴露的问题十分常见。考虑图5-10中的联系HASACCOUNT。从HASACCOUNT到关系模型的一个典型转换为

```
CREATE TABLE HASACCOUNT (
AccountNumber INTEGER NOT NULL,
ClientId      CHAR(20),
OfficeId      INTEGER,                              (8.3)
PRIMARY KEY (ClientId, OfficeId),
FOREIGN KEY (OfficeId) REFERENCES OFFICE
... ... ... )
```

回忆一下，一个客户在一个办事处最多只有一个账户，因此(ClientId, OfficeId)是一个键。同样，一个账户只能分配到一个办事处。仔细分析一下可以发现，这个要求会导致在PERSON例子中出现的同样问题。例如，对于记录一个特定的账户由特定的办事处管理这一事实的元

组，在没有记录客户信息的情况下（由于ClientId是主键的一部分），它是不能被添加进去的，这就是插入异常。由于这种特定应用的特殊性，这种情况（和删除异常）也许并不是一个严重的问题；但是更新异常会导致维护问题。把一个账户从一个办事处移到另一个办事处需要改变对应于那个账户的每个元组中的OfficeId。如果这个账户有多个客户，那么这可能就会有问题。

我们会在本章的后面再介绍这个例子，因为HASACCOUNT有一些PERSON例子所没有的有趣的性质。比如，尽管我们期望对HASACCOUNT进行分解，但这会导致额外的在PERSON的分解中没有出现的维护开销。

上述讨论产生两个要点：1）作为对E-R方法的一个补充，关系模式的分解在消除冗余的问题上是一个有用的工具；2）选择正确分解的标准并不是显而易见的，特别是在处理包含很多属性的模式时更是如此。由于这些原因，8.3～8.6节的目的就是研究相关的技术与标准，以识别需要分解的关系模式；以及理解在分解中不丢失信息的含义。

在分解理论的发展过程中，主要的工具是**函数依赖**（functional dependency），它是键约束思想的一个扩展。函数依赖用于定义**范式**（normal form），即在更新密集的事务系统中对关系模式的一组要求。这就是分解理论经常被称为**规范化理论**（normalization theory）的原因。8.5～8.9节将展示在规范化过程中所用的算法。

8.3　函数依赖

在本章剩下的部分，我们使用在关系规范化理论中常用的特定符号来表示属性，这些属性如下所示：从字母表开始的大写字母（A,B,C,D）代表单个属性；在字母表中从中间到最后的有上划线的大写字母（$\overline{P},\overline{V},\overline{W},\overline{X},\overline{Y},\overline{Z}$）代表属性的集合。同样，对字母组成的字符串，如$ABCD$，代表相应属性的集合（$\{A,B,C,D\}$）；带上划线的字母组成的字符串，如($\overline{X}\,\overline{Y}\,\overline{Z}$)，代表这些集合的并($\overline{X} \cup \overline{Y} \cup \overline{Z}$)。我们应该熟悉这些符号，它提供了一种简洁的语言，在实例与定义中使用这些符号会非常方便。

对关系模式R的**函数依赖**（FD）是形如$\overline{X} \rightarrow \overline{Y}$的一种约束，$\overline{X}$和$\overline{Y}$是**R**中的属性集。如果**r**是**R**的关系实例，并且满足如下条件，那么认为它满足函数依赖：

> 对**r**中每一对元组t和s，如果t和s在$\overline{X}$属性集上的值一致，那么t和s在$\overline{Y}$属性集上的值也一致。

换句话说，在**r**中不应该有一对元组，它们在$\overline{X}$属性集上的值一致，但对$\overline{Y}$中某些属性却有不同的值。

注意，第4章介绍的键约束是FD的一种特殊情况。假定key($\overline{K}$)是关系模式**R**中的键约束，**r**是**R**的关系实例。根据定义，当且仅当没有一对不同的元组，($t, s \in \mathbf{r}$)，使得t和s在key($\overline{K}$)上的每一个属性上有相同的值，那么**r**满足key($\overline{K}$)。因此，键约束等同于FD $\overline{K} \rightarrow \overline{R}$，$\overline{K}$是键约束中属性的集合，$\overline{R}$是模式**R**的所有属性的集合。

请记住，函数依赖与关系模式相关联，而当我们考虑是否满足一个函数依赖时，我们必须考虑那些模式的关系实例。这是因为FD是模式上的完整性约束（类似于键约束），即限制满足给定FD情况下允许的关系实例的集合。因此，给定一个模式**R**=($\overline{R}$;*Constraints*)，$\overline{R}$是属

性的集合，*Constraints*是函数依赖的集合，我们要寻找 $\overline{R}$ 的所有关系实例的集合，它们满足*Constraints*中的每个函数依赖。这样的关系实例称为**R的合法实例**（legal instance）。

函数依赖与更新异常

存在于关系模式中的特定的函数依赖会导致相应的关系实例的冗余。考虑8.1节与8.3节讨论的两个例子：模式PERSON和HASACCOUNT。每个模式有一个主键，这些主键分别在（5.1）与（8.3）相应的CREATE TABLE命令中定义。相应地，有如下的函数依赖：

```
PERSON:     SSN,Hobby → SSN,Name,Address,Hobby
HASACCOUNT: ClientId,OfficeId → AccountNumber,                    (8.4)
                                ClientId,OfficeId
```

然而，这些并不是最初的规格说明所蕴含的仅有的函数依赖。例如，在图5-1的E-R图中，Name和Address被定义为单值属性。这表明，一个PERSON实体（由它的SSN属性标识）最多只能有一个名字和一个地址。类似地，PSSC（5.5节所讨论的经纪人公司）的业务规则要求每个账户只能确切地分配到一个办事处，这意味着对于相应的关系模式，如下的函数依赖同样成立：

```
PERSON:     SSN → Name, Address
HASACCOUNT: AccountNumber → OfficeId                              (8.5)
```

很容易看出(8.4)中依赖的句法结构对应于我们在相应关系中所确定的更新异常。比如，在PERSON中的问题是对任何给定的SSN，我们无法单独改变属性Name和Address的值，而不考虑相应的个人是否有业余爱好，即如果某个人有多个爱好，那么这个改变将在多行发生。同样，对于HASACCOUNT，我们无法改变OfficeId（即把一个账户转到另一个办事处）的值而不去考虑与此账户关联的所有客户。由于若干个客户共享同一个账户，在HASACCOUNT中可以有多行引用同一个账户；因此，OfficeId所在的多行必须同时改变。

注意，对上述两个例子，涉及更新异常的属性均出现在函数依赖的左侧。我们可以看到，更新异常与特定类型的函数依赖有联系。那么什么样的依赖是不好的？请注意(8.4)与(8.5)中依赖的主要区别：前者说明相应关系的键约束，而后者不是。

然而，只是知道哪些依赖会导致异常还不够，我们必须能采取某些方式解决这些问题。我们不能只是抛弃不好的函数依赖，因为它们是数据库所建模的业务规则的一部分。它们是需求文档的一部分，在没有征得顾客同意的情况下不能改变。另一方面，我们看到模式分解是一个有用的工具。尽管分解不能抛弃函数依赖，但它可以使依赖工作正常。比如，（8.1）中的分解所产生的模式会把不良的函数依赖SSN→Name，Address变为一个行为良好的键约束。

8.4 函数依赖的性质

在进一步讨论之前，我们应该掌握一些函数依赖的数学性质和检测它们的算法。由于这些性质和算法依赖于8.3节开始所介绍的符号惯例，所以我们重新回顾一下这些惯例。

我们要研究的函数依赖的性质基于*蕴涵*（entailment）。考虑属性集 $\overline{R}$，$\overline{R}$ 的函数依赖的集合为F，$\overline{R}$ 的另一个函数依赖为f。如果对属性集 $\overline{R}$ 每个关系**r**有如下性质，则称F*蕴涵*f：

*如果**r**满足F中的每个函数依赖，那么**r**满足函数依赖f。*

给定一个函数依赖的集合F，F的**闭包**（closure）F^+是所有被F蕴涵的函数依赖的集合。很明显，F^+包含F为子集[⊖]。

如果F和G是函数依赖的集合，如果F蕴涵G中每个单独的函数依赖，那么我们说F蕴涵G。如果F蕴涵G且G蕴涵F，则F和G称为**等价**。

我们现在介绍几个蕴涵中简单但重要的性质。

1. 自反性（Reflexivity）

无论在什么情况下，每一个关系均满足某些函数依赖。这些依赖的形式为 $\overline{X} \rightarrow \overline{Y}$，其中 $\overline{Y} \subseteq \overline{X}$，它们称为**平凡**（trivial）函数依赖。

• 自反性说明，如果 $\overline{Y} \subseteq \overline{X}$，那么 $\overline{X} \rightarrow \overline{Y}$。

为说明平凡函数依赖为什么总被满足，考虑关系**r**，它的属性集包括 $\overline{X}$ $\overline{Y}$ 中的所有属性。假如t，$s \in \mathbf{r}$为在 $\overline{X}$ 上一致的元组。但是，由于 $\overline{Y} \subseteq \overline{X}$，这意味着$t$和$s$也在 $\overline{Y}$ 上一致。因此，**r**满足 $\overline{X} \rightarrow \overline{Y}$。

我们现在可以把平凡函数依赖和蕴涵联系起来。由于每个关系满足平凡函数依赖，因此每个函数依赖集均蕴涵它！特别地，F^+包含每个平凡函数依赖。

2. 增广性（Augmentation）

考虑函数依赖 $\overline{X} \rightarrow \overline{Y}$ 和另一个属性集 $\overline{Z}$。$\overline{R}$ 包含 $\overline{X} \cup \overline{Y} \cup \overline{Z}$。那么 $\overline{X} \rightarrow \overline{Y}$ 蕴涵 $\overline{X}\,\overline{Z} \rightarrow \overline{Y}\,\overline{Z}$。换句话说，对 $\overline{R}$ 中每个关系**r**满足 $\overline{X} \rightarrow \overline{Y}$ 则一定也满足函数依赖 $\overline{X}\,\overline{Z} \rightarrow \overline{Y}\,\overline{Z}$。

• 增广性说明，如果 $\overline{X} \rightarrow \overline{Y}$，那么 $\overline{X}\,\overline{Z} \rightarrow \overline{Y}\,\overline{Z}$。

下面来证明这个性质。如果t，$s \in \mathbf{r}$在 $\overline{X}$ $\overline{Z}$的每个属性上均一致，那么它们在 $\overline{X}$上也一致。由于**r**满足 $\overline{X} \rightarrow \overline{Y}$，所以$t$和$s$也必须在 $\overline{Y}$ 上一致。由于我们已经假定它们在更大的属性集 $\overline{X}\,\overline{Z}$ 上一致，那么它们在 $\overline{Z}$ 上也一致。因此，如果t，s在 $\overline{X}\,\overline{Z}$的每个属性上均一致，那么它们在 $\overline{Y}\,\overline{Z}$ 上也一致。由于这是我们在**r**中任意选择的一对元组，因此可以表明**r**满足 $\overline{X}\,\overline{Z} \rightarrow \overline{Y}\,\overline{Z}$。

3. 传递性（Transitivity）

函数依赖集$\{\overline{X} \rightarrow \overline{Y}$，$\overline{Y} \rightarrow \overline{Z}\}$蕴涵函数依赖 $\overline{X} \rightarrow \overline{Z}$。

• 传递性说明，如果 $\overline{X} \rightarrow \overline{Y}$ 且 $\overline{Y} \rightarrow \overline{Z}$，那么 $\overline{Y} \rightarrow \overline{Z}$。

这个性质可以用与前面类似的方法进行证明（见练习）。

这三个函数依赖的性质称为Armstrong公理[⊜]，它们主要用于证明各种数据库设计算法的正确性。然而，他们也同样是供（真正的）数据库设计者使用的有力工具，因为它们能帮助人们快速的找到关系模式中有问题的函数依赖。我们现在看一下如何用Armstrong公理来推导新的函数依赖。

4. 函数依赖的并

任何关系**r**，若满足 $\overline{X} \rightarrow \overline{Y}$ 和 $\overline{X} \rightarrow \overline{Z}$，那么它也一定满足 $\overline{X} \rightarrow \overline{Y}\ \overline{Z}$。为证明这一点，我们可以使用Armstrong公理定义的简单的语法操纵从 $\overline{X} \rightarrow \overline{Y}$ 和 $\overline{X} \rightarrow \overline{Z}$ 推导出 $\overline{X} \rightarrow \overline{Y}\ \overline{Z}$。这种操纵很容易在计算机中进行编程，不像我们在建立公理本身时基于元组进行考虑。下面是推理过程：

⊖ 如果$f \in F$，那么满足F中每个函数依赖的关系很明显满足f。因此，根据蕴涵的定义，f被F蕴涵。

⊜ 严格来说，它们是推理规则，因为它们从旧规则中产生新规则。只有自反性可以视为公理。

a) $\overline{X} \rightarrow \overline{Y}$　　已知条件

b) $\overline{X} \rightarrow \overline{Z}$　　已知条件

c) $\overline{X} \rightarrow \overline{Y}\ \overline{X}$　　把 $\overline{X}$ 加到a的两边：Armstrong增广规则

d) $\overline{Y}\ \overline{X} \rightarrow \overline{Y}\ \overline{Z}$　　把 $\overline{Y}$ 加到b的两边：Armstrong增广规则

e) $\overline{X} \rightarrow \overline{Y}\ \overline{Z}$　　应用于c和d的Armstrong传递规则

5. *函数依赖的分解*

同样我们可以证明如下规则：每个关系满足 $\overline{X} \rightarrow \overline{Y}\ \overline{Z}$，那么一定满足函数依赖 $\overline{X} \rightarrow \overline{Y}$ 和 $\overline{X} \rightarrow \overline{Z}$。可以通过下列简单的步骤进行证明：

a) $\overline{X} \rightarrow \overline{Y}\ \overline{Z}$　条件

b) $\overline{Y}\ \overline{Z} \rightarrow \overline{Y}$　根据Armstrong自反性规则，$\overline{Y} \subseteq \overline{Y}\,\overline{Z}$

c) $\overline{X} \rightarrow \overline{Y}$　根据a与b的传递性

同样可推出 $\overline{X} \rightarrow \overline{Z}$。

Armstrong公理明显是正确（sound）的。正确是指根据公理推出的任何表达式 $\overline{X} \rightarrow \overline{Y}$ 确实是函数依赖。正确性来源于如下事实：我们已经证明这些推理规则对每个关系都有效。然而，在它们是完整的(complete)这方面就不太明显，即如果一个函数依赖集F蕴涵另一个函数依赖f，那么在只依靠Armstrong公理的情况下，利用类似于上面的步骤顺序，可以从F推出f。这里我们不证明这个事实，但在[Ullman 1998]中可以找到相应的证明。Armstrong公理的正确性和完整性并不只是理论上有用，这个结论的实际价值也相当高，因为它保证函数依赖的蕴涵（即是否有$f \in F^+$）可以用计算机程序来进行验证。我们下面就介绍这样一个算法。

验证函数依赖集F蕴涵函数依赖f的一个直接方法是指导计算机对F应用Armstrong公理对F进行各种可能情况的尝试。由于F和f中属性集的个数是有限的，所以这个推导过程不会永远进行下去。当所有可能的推导都完成后，我们可以简单地查看f是否在这个过程所产生的函数依赖中。Armstrong公理的完整性保证，当且仅当f是推导出的函数依赖中的某一个时，$f \in F^+$。

为了了解这个过程的工作过程，考虑如下函数依赖集：$F=\{AC \rightarrow B,\ A \rightarrow C,\ D \rightarrow A\}$和$G=\{A \rightarrow B,\ A \rightarrow C,\ D \rightarrow A,\ D \rightarrow B\}$。我们可以用Armstrong公理来证明这两个集合是等价的，即G中的每个函数依赖被F蕴涵，反之亦然。例如，为证明$A \rightarrow B$被F蕴涵，我们可以用各种方法来应用Armstrong公理。大部分尝试将不会产生任何结果，但有一些尝试会产生结果。比如，下面的推导会建立所希望的蕴涵：

a) $A \rightarrow C$　F中的一个函数依赖

b) $A \rightarrow AC$　从a和Armstrong增广公理得出

c) $A \rightarrow B$　从b，$AC \rightarrow B \in F$和Armstrong传递公理得出

函数依赖$A \rightarrow C$和$D \rightarrow A$同时属于F和G，因此这个推导是很容易的。对G中的$D \rightarrow B$，计算机会尽力使用Armstrong公理直到推导出这个函数依赖。一段时间后，它会偶然发现如下的有效推导：

a) $D \rightarrow A$　F中的函数依赖

b) $A \rightarrow B$　先前所推导出的

c) $D \rightarrow B$　根据a，b和Armstrong传递公理得出

这表明F中被G所蕴含的每个函数依赖都可以用类似方法推导出来。

尽管利用Armstrong公理来检查蕴涵比较简单，但它不是很有效。事实上，F^+的数量是F数量的指数级大小，因此对大的数据库模式，设计者要花很长时间才能看到结果。因此我们将开发一个更有效的算法，它同样是基于Armstrong公理，但使用起来更方便。

6. 检测函数依赖的蕴涵

验证蕴涵的新算法的思想基于*属性闭包*（attribute closure）。

给定函数依赖集F和属性的集合$\overline{X}$，相对于F，$\overline{X}$的属性闭包$\overline{X}_F^+$定义为：

$$\overline{X}_F^+ = \left\{A \middle| \overline{X} \rightarrow A \in F^+\right\}$$

换句话说，$\overline{X}_F^+$是所有属性A的集合，其中$\overline{X} \rightarrow A$被$F$蕴涵。注意$\overline{X} \subseteq \overline{X}_F^+$，因为如果$A \in \overline{X}$，那么根据Armstrong自反性公理，$\overline{X} \rightarrow A$是平凡函数依赖而被包括$F$在内的每个函数依赖集所蕴涵。

理解F的闭包（即F^+）和$\overline{X}$的闭包（即$\overline{X}_F^+$）是相关且不同的概念是很重要的：F^+是函数依赖的集合，而$\overline{X}_F^+$是属性的集合。

检测函数依赖的蕴涵的更有效的算法的过程如下：给定函数依赖集F和函数依赖$\overline{X} \rightarrow \overline{Y}$，检测是否有$\overline{Y} \subseteq \overline{X}_F^+$。如果存在，那么$F$蕴涵$\overline{X} \rightarrow \overline{Y}$。否则，如果$\overline{Y} \not\subseteq \overline{X}_F^+$，那么$F$不蕴涵$\overline{X} \rightarrow \overline{Y}$。

这个算法的正确性可由Armstrong公理证明。如果$\overline{Y} \subseteq \overline{X}_F^+$，那么对每个$A \in \overline{Y}$有$\overline{X} \rightarrow A \in F^+$（根据$\overline{X}_F^+$的定义）。根据函数依赖的并规则，$F$蕴涵$\overline{X} \rightarrow \overline{Y}$。相反，如果$\overline{Y} \not\subseteq \overline{X}_F^+$，那么有$B \in \overline{Y}$，使$\overline{B} \notin \overline{X}_F^+$。因此，$\overline{X} \rightarrow B$不被$F$蕴涵。那么$F$不蕴涵$\overline{X} \rightarrow \overline{Y}$；如果蕴涵，那么根据函数依赖的分解规则，它将会也蕴涵$\overline{X} \rightarrow B$。

上述算法的核心是检测一个属性集是否属于$\overline{X}_F^+$。因此，我们的工作还没有完成：我们需要一个算法来计算$\overline{X}$的闭包，图8-3说明了此算法。该算法的基本思想是，应用F中的函数依赖来扩大属于$\overline{X}_F^+$的属性集。闭包的初始值为$\overline{X}$，因为我们知道$\overline{X}$总是$\overline{X}_F^+$的子集。

```
closure := X̄
repeat
    old := closure
    if there is an FD Z̄ → V̄ ∈ F such that Z̄ ⊆ closure and V̄ ⊈ closure then
        closure := closure ∪ V̄
until old = closure
return closure
```

图8-3　计算属性闭包$\overline{X}_F^+$

算法的正确性可以用归纳法进行证明。最初，闭包为$\overline{X}$，因此$\overline{X} \rightarrow closure$在$F^+$中。接着，假定在图8-3的repeat循环的某个中间步骤有$\overline{X} \rightarrow closure \in F^+$，并给定一个函数依赖$\overline{Z} \rightarrow \overline{V} \in F$有$\overline{Z} \subset closure$，我们可以应用*通用传递规则*（参见练习8.7）来推出F蕴涵$\overline{X} \rightarrow closure \cup \overline{V}$。因此，如果在计算结束时$A \in closure$，那么$A \in \overline{X}_F^+$。反之同样是正确的：如果$A \in \overline{X}_F^+$，那么在计算结束时$A \in closure$（参见练习8.8）。

和不加区别地使用Armstrong公理的思想的简单算法不同，图8-3算法的运行时间复杂度与F大小的平方成正比。事实上，在[Beeri and Bernstein 1997]中给出一个计算$\overline{X}_F^+$的算法，这个

算法的运行时间复杂度与F的大小是线性关系。这个算法更适合计算机程序，而它的内部工作过程更复杂。

例8.4.1　检测蕴涵

考虑关系模式，$\mathbf{R}=(\overline{R};F)$，其中 $\overline{R}=ABCDEFGHIJ$，函数依赖集F包含如下函数依赖：$AB\rightarrow C, D\rightarrow E, AE\rightarrow G, GD\rightarrow H, ID\rightarrow J$。我们希望检测$F$是否蕴涵$ABD\rightarrow GH$和$ABD\rightarrow HJ$。

首先计算 ABD_F^+。以$closure=ABD$开始。两个函数依赖可以用在图8-3的第一次迭代中。根据定义，使用$AB\rightarrow C$，得到$closure=ABDC$。在第二次迭代时，使用$D\rightarrow E$，得到$closure=ABDCE$。现在可以在第三次迭代中使用函数依赖$AE\rightarrow G$，产生$closure=ABDCEG$。这又允许将$GD\rightarrow H$应用到第四次迭代，得到$closure=ABDCEGH$。在第五次迭代中，我们不能应用任何新的函数依赖，因此$closure$不再改变，循环结束。有 $ABD_F^+=ABDCEGH$。

由于$GH \subseteq ABDCEGH$，我们可以推出F蕴涵$ABD\rightarrow GH$。另一方面，$HJ \not\subseteq ABDCEGH$，因此F不蕴涵$ABD\rightarrow HJ$。注意，F蕴涵$ABD\rightarrow H$。

通过上述测试蕴涵的算法可得到一个测试一对函数依赖集是否等价的简单算法。假定F和G是这样两个集合。为检测它们是否等价，我们必须检测G中每个函数依赖是否被F蕴涵，反之亦然。图8-4描述了此算法。

```
Input: F G – FD sets
Output: true, if F is equivalent to G; false otherwise
for each f ∈ F do
    if G does not entail f then return false
for each g ∈ G do
    if F does not entail g then return false
return true
```

图8-4　测试函数依赖集的等价性

8.5　范式

为消除冗余和潜在的更新异常，数据库理论提出几个模式的**范式**（normal form），如果一个模式是范式中的一种，它就有一些可以预测的性质。最初，[Codd 1970]提出三种范式，每一种范式能比前一种范式消除更多的异常。

正如Codd所介绍的，**第一范式**（First Normal Form，1NF）等价于关系数据模型的定义。**第二范式**（Second Normal Form，2NF）尝试消除某些潜在异常，但可以证明它没有实际用处，因此不讨论它。

第三范式（Third Normal Form，3NF）最初被认为是“最终”范式。但是，Boyce和Codd很快意识到3NF同样有一些不受欢迎的函数依赖，因此他们引入了所谓的Boyce-Codd范式（BCNF）。但是，尽管BCNF更受人欢迎，但有时要付出代价才能达到。在本节，我们定义BCNF和3NF。后续章节会提出一些算法，以便自动地将包含各种不好性质的关系模式转换为3NF和BCNF的模式集。我们同样会研究进行这些转换的代价。

在本章的最后，我们将看到，某些冗余不是由函数依赖所引起的，而是由别的依赖引起。为解决这个问题，我们引入第四范式（Fourth Normal Form，4NF），它是对BCNF的进一步扩展。

1. Boyce-Codd范式

对一个关系模式**R**=($\overline{R}$;*F*) ($\overline{R}$ 是**R**的属性集合，*F*是和**R**关联的函数依赖的集合）来说，如果每个函数依赖 $\overline{X} \rightarrow \overline{Y} \in F$，那么若下面两个条件有一个为真，则**R**是Boyce-Codd范式:

- $\overline{Y} \subseteq \overline{X}$ （即是平凡函数依赖)。
- $\overline{X}$ 是**R**的超键。

换句话说，唯一的非平凡函数依赖是那些键在功能上决定一个或多个属性的函数依赖。

很容易看出(8.1)中的关系模式PERSON1和HOBBY是BCNF，因为唯一的非平凡函数依赖是SSN→Name, Address。它应用到以SSN作为键的PERSON1。

另一方面，考虑(5.1)中CREATE TABLE语句所定义的模式PERSON和（8.3）的SQL语句所定义的模式HASACCOUNT。正如前面所讨论的，这些语句无法获取到一些重要的关系，像（8.5）中函数依赖所表示的。这些函数依赖中的每一个都违反了BCNF的要求：它们不是平凡的，而且它们的左侧（SSN和AccountNumber）不是各自模式的键。

注意，一个BCNF模式可以有多于一个的键。比如，**R**=(*ABCD*;*F*)，其中*F*={*AB*→*CD*, *AC*→*BD*}有两个键*AB*和*AC*。它也是BCNF，因为*F*中每个函数依赖的左边均是键。

BCNF模式的一个重要性质是它们的实例不包括冗余信息。由于我们只是通过实例来说明冗余问题，所以上述语句似乎不太有说服力。确切地说，什么是冗余信息？比如，对上述提到的BCNF模式**R**，这个抽象关系是否存储了冗余信息?

A	B	C	D
1	1	3	4
2	1	3	4

从表面上看，该模式似乎是存储了冗余信息，因为这两个元组除了一个属性外，其他的都一致。然而，在不同元组的某些属性上有相同的值并不一定表明元组存储了冗余信息。当某些属性集 $\overline{X}$ 的值确实决定另一个属性*A*的值，即存在函数依赖，此时才认为有冗余。如果两个元组在 $\overline{X}$ 上有相同的值，那么它们在*A*上一定有相同的值。如果我们只将 $\overline{X}$ 和*A*之间的关联存储一次（在一个单独的关系中），而不是在 $\overline{X}$ 上一致的模式**R**的一个实例所有元组上重复它，那么冗余就会被消除。由于**R**在属性BCD上没有函数依赖，所以没有冗余信息会被存储。关系中的元组在BCD上一致这一事实只是巧合。例如，在第一个元组中属性*D*的值可以从4变到5，而无需考虑第二个元组。

DBMS会自动消除一种冗余：两个元组在键字段有相同的值在模式的任何实例中是禁止的。这是一个特例：键确定一个实体，同时也决定描述那个实体的所有属性的值。由于BCNF的定义排除了不包括键的关联，所以BCNF模式的关系不会存储冗余信息。因此，删除与更新异常不会在BCNF关系中出现。

有多于一个键的关系仍会有插入异常。为说明这一点，假定对*ABD*和*ACD*的关联被加到下述关系中:

A	B	C	D
1	1	3	4
2	1	3	4
3	4	NULL	5
3	NULL	2	5

由于在第一个关联中的属性C和第二个中B的值是未知的，所以我们用NULL填充缺失信息。然而，我们不能分辨新加进的元组是否相同——这要依靠空值的真实值。这个问题的一个实际解决方案，即SQL标准所采纳的方案，是指定一个键为主键并禁止主键的属性出现空值。

2. 第三范式

对于一个关系模式**R**=($\overline{R}$; F) ($\overline{R}$是**R**的属性集合，F是和**R**关联的函数依赖的集合）来说，如果对每个函数依赖 $\overline{X} \rightarrow \overline{A} \in F$，那么在下面任何一个条件为真的情况下，**R**是第三范式:

- $A \in \overline{X}$，(即是一个平凡函数依赖)。
- $\overline{X}$是**R**的超键。
- 对**R**的某个键 $\overline{K}$，$A \in \overline{K}$。

注意，3NF定义的前两个条件与BCNF的定义相同。因此，3NF是对BCNF要求的一个放松。BCNF的模式一定是3NF，但反之则不然。例如，(8.3)所示的关系HasAccount是3NF，因为唯一不是基于键约束的函数依赖是AccountNumber→OfficeId，而OfficeId是键的一部分。然而，这个关系不是BCNF，如前所示[⊖]。

如果你想知道3NF定义中的三个条件的本质优点，那么答案是无。从某方面来说，3NF是在寻找BCNF过程中，被错误地发现的！它能保存下来的原因是，后来发现它有一些BCNF没有的算法性质。我们在随后的章节会讨论这些问题。

回忆8.2节，HasAccount的关系实例也许会存储冗余信息。现在我们可以看到这个冗余的出现，是因为AccountNumber和OfficeId之间的函数依赖，而它又不被键约束所蕴涵。

再例如，考虑先前讨论的模式Person。这个模式违反了3NF的要求，因为函数依赖SSN→NAME不基于键约束（SSN不是超键），而且Name不属于Person的某个键。然而，把这个模式分解成(8.1)中的Person1和Hobby会得到既是3NF又是BCNF的两个模式。

8.6 分解的性质

由于BCNF模式中没有冗余，3NF中冗余也是有限的，因此我们希望把一个给定的模式分解成一个模式的集合，其中每一个模式都是BCNF或3NF。

前一节讨论的主要问题是3NF没有完全解决冗余问题。因此，初看起来，没有理由考虑将3NF作为数据库设计的目标。然而，与冗余相关的维护问题并不是全部。正如我们将要看到的，维护也与完整性约束相关[⊜]，3NF分解在维护方面有时比BCNF分解更好。我们先来定义这些性质。

回忆8.2节，不是所有的分解都等价的。比如，对（8.1）所示的Person所进行的分解就是好的，而（8.2）则毫无意义。是否有客观方法来评价哪些分解有意义，哪些没有意义？并且这个客观方法是否能解释给计算机？这两个问题的答案都是肯定的。有意义的分解称为*无损的*（lossless）。在介绍这个概念之前，我们需要更清楚地了解什么叫“分解”。

⊖ 事实上，HasAccount是3NF但不是BCNF的最小可能例子（参见练习8.5）。

⊜ 例如，最初的表中的一个完整性限制只有在对被分解的表做连接后才能进行检测，但这会导致运行时的大量开销。

若有模式$\mathbf{R}=(\overline{R}; F)$，其$\overline{R}$是模式属性的集合，$F$是函数依赖的集合，那么**模式的分解**（decomposition of a schema）是模式的一个集合：

$$\mathbf{R}_1=(\overline{R}_1; F_1),\ \mathbf{R}_2=(\overline{R}_2; F_2),\ \cdots,\ \mathbf{R}_n=(\overline{R}_n; F_n)$$

它满足如下条件：

1) $\overline{R}=\mathrm{U}_{i=1}^{n}\overline{R}_i$

2) 对每个i=1,…,n，F蕴涵F_i

定义的第一部分是清晰的：一个分解不应该引入新的属性，它也不应该丢失原模式中的属性。定义的第二部分说明一个分解不应该引入新的函数依赖（但可能会丢失一些函数依赖）。后面会对第二个要求进行更详细的讨论。

一个模式的分解自然会导致它的关系的分解。对模式**R**的一个**关系的分解**（decomposition of a relation）会产生关系的集合

$$\mathbf{r}_1=\pi_{\overline{R}_1}(\mathbf{r}),\ \mathbf{r}_2=\pi_{\overline{R}_2}(\mathbf{r}),\ \cdots,\ \mathbf{r}_n=\pi_{\overline{R}_n}(\mathbf{r})$$

其中，π是投影运算符。可以看到（参见练习8.9），如果**r**是**R**的一个有效实例，那么每个$\mathbf{r}_i$均满足F_i中的所有函数依赖，因此每个$\mathbf{r}_i$是模式$\mathbf{R}_i$的有效关系实例。分解的目的就是把原来的关系模式**r**替换为$\mathbf{r}_1,\cdots,\mathbf{r}_n$的关系集合，其中它们的模式组成了分解的原模式。

根据上述定义，认识到*模式分解*和*关系实例分解*是两个不同但相关的概念是很重要的。

应用上述定义到我们的例子，我们可以看到把PERSON拆分为PERSON1和HOBBY（参见(8.1)和图8-1）会产生一个分解。把PERSON按(8.2)和图8-2所示进行拆分，同样是上述意义上的一个分解。它明显满足分解的第一个条件。它同样满足第二个条件，因为(8.2)只有平凡函数依赖，它们会被每个函数依赖集所蕴涵。

最后一个例子表明，上述分解的定义并不具备一个分解所有好的性质，正如8.2节所示，(8.2）中的分解是没有意义的。下面，我们将介绍分解的其他一些有意义的性质。

8.6.1 无损分解与有损分解

考虑上面定义的关系**r**和它的分解$\mathbf{r}_1,\cdots,\mathbf{r}_n$。由于分解后数据库不再存储关系**r**，而是维护它的投影$\mathbf{r}_1,\cdots,\mathbf{r}_n$，数据库必须能够从利用这些投影重构最初的关系**r**。不能重构**r**意味着分解没有表示原始数据库的信息。（就像银行丢失了谁拥有哪个账户的信息，或者更糟，把那些账户分给了错误的储户！）

从原则上说，我们可以用任何能够保证从它的投影中重构**r**的计算方法。然而，自然的（并且在大多数情况下比较实际）的方法是进行自然联结。我们假定**r**当且仅当

$$\mathbf{r}=\mathbf{r}_1\bowtie\mathbf{r}_2\bowtie\cdots\bowtie\mathbf{r}_n$$

时，**r**是可重构的。

重构是模式分解的一个性质，而不是这个模式的一个实例。在数据库设计阶段，设计者操纵的是模式，而不是关系，对模式的任何变换必须保证对它的所有有效的关系实例的可重构性。

这个讨论会产生如下概念。模式**R**=($\overline{R}$; F)分解为模式集

$$\mathbf{R}_1 = \left(\overline{R}_1;\ F_1\right),\ \mathbf{R}_2 = \left(\overline{R}_2;\ F_2\right),\ \cdots,\ \mathbf{R}_n = \left(\overline{R}_n;\ F_n\right)$$

是无损的，只要对模式**R**的每个有效实例**r**，都满足

$$\mathbf{r} = \mathbf{r}_1 \bowtie \mathbf{r}_2 \bowtie \cdots \bowtie \mathbf{r}_n$$

其中，

$$\mathbf{r}_1 = \pi_{\overline{R}_1}(\mathbf{r}),\ \mathbf{r}_2 = \pi_{\overline{R}_2}(\mathbf{r}),\ \cdots,\ \mathbf{r}_n = \pi_{\overline{R}_n}(\mathbf{r})$$

否则一个分解则是**有损的**（lossy）。

根据定义，无损模式分解可以保证原始模式的任何有效实例可以利用单独分解的模式的投影进行重构。注意，

$$\mathbf{r} \subseteq \mathbf{r}_1 \bowtie \mathbf{r}_2 \bowtie \cdots \bowtie \mathbf{r}_n$$

对任何分解都成立（练习8.10），因此无损性只是陈述相反的结论。

$$\mathbf{r} \supseteq \mathbf{r}_1 \bowtie \mathbf{r}_2 \bowtie \cdots \bowtie \mathbf{r}_n$$

事实

$$\mathbf{r} \subseteq \mathbf{r}_1 \bowtie \mathbf{r}_2 \bowtie \cdots \bowtie \mathbf{r}_n$$

无论在什么条件下成立，初看起来也许有些令人困惑。如果通过联结**r**的投影可以得到更多的元组，为什么这样一个分解会称为“有损”？我们得到更多的元组，而不是更少！为澄清这个问题，注意，我们这里所丢失的不是元组而是关于*哪些元组是正确的信息*。比如，考虑模式PERSON的分解（8.2）。图8-2表示图5-2的PERSON的一个有效关系实例的相应分解。然而，如果我们现在计算分解中关系的自然联结（由于这些关系没有公共属性，会变为笛卡儿积），我们将无法辨别每个人的住址和每个人的爱好。姓名和SSN之间的联系也会丢失。换句话说，当重构原始关系时，得到更多的元组与得到更少的元组同样不是好事情，我们必须得到原始关系的确切的元组集合。

现在我们明白了无损的重要性，我们需要一个计算机算法来验证这个性质，因为无损联结的定义并没有提供一个有效的测试，而只是告诉我们尝试每个可能的关系。这是不太可行的，而且效率也不高。

检测得到n个模式的一个分解是否为无损分解的通用测试是存在的，但有些复杂。我们可以在[Beeri et al. 1981]中找到这个算法。然而，有一个更简单的用来检测二元分解（即分解为一对模式）的算法。这个测试可以确定分解结果多于两个模式的无损性，只要这个分解可以通过一系列的二元分解得到即可。由于大部分分解是以这种方式获得的，下面介绍的简单的二元测试对大部分实际情况都是有效的。

测试二元分解的无损性

设**R**=($\overline{R}$; F)是一个模式，$\mathbf{R}_1$=($\overline{R}_1$; F_1)，$\mathbf{R}_2$=($\overline{R}_2$; F_2)是**R**的一个二元分解。当且仅当下面的某个条件为真时，此分解是无损的:

- ($\overline{R}_1 \cap \overline{R}_2$) → $\overline{R}_1 \in F^+$
- ($\overline{R}_1 \cap \overline{R}_2$) → $\overline{R}_2 \in F^+$

为说明其中的原理，假定$(\overline{R}_1 \cap \overline{R}_2) \rightarrow \overline{R}_2 \in F^+$成立。$\overline{R}_1$是**R**的超键，因为$\overline{R}_1$属性的一个子集的值决定$\overline{R}_2$所有属性的值(明显地，$\overline{R}_1$函数决定$\overline{R}_1$)。假定**r**是**R**的有效关系实例。由于$\overline{R}_1$是$\overline{R}$的一个超键，$\mathbf{r}_1 = \pi_{\overline{R}}(\mathbf{r})$的每个元组可延伸为**r**中确切的一个元组。因此，如图8-5所描述，$\mathbf{r}_1$的基数（元组的个数）等于**r**的基数，$\mathbf{r}_1$中的每个元组与$\mathbf{r}_2 = \pi_{\overline{R}_2}(\mathbf{r})$中确切的一个元组相联结（如果$\mathbf{r}_2$中多个元组与$\mathbf{r}_1$中一个元组联结，则$(\overline{R}_1 \cap \overline{R}_2) \rightarrow \overline{R}_2$将不会是一个函数依赖)。因此，$\mathbf{r}_1 \bowtie \mathbf{r}_2$的基数等于$\mathbf{r}_1$的基数，$\mathbf{r}_1$的基数又等于**r**的基数。由于**r**必须是$\mathbf{r}_1 \bowtie \mathbf{r}_2$的子集，因此有$\mathbf{r} = \mathbf{r}_1 \bowtie \mathbf{r}_2$。相反地，如果上述两个函数依赖都不成立，则很容易构造一个关系**r**，使$\mathbf{r} \subset \mathbf{r}_1 \bowtie \mathbf{r}_2$。这个构造的细节作为练习8.11。

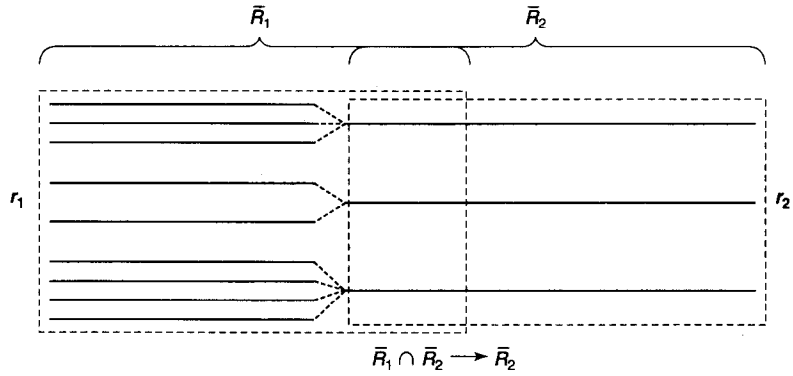


图8-5　无损二元分解的元组结构：$\mathbf{r}_1$的一行与$\mathbf{r}_2$的一行的联结

上述测试可以用来证实我们的直觉：PERSON分解为PERSON1和HOBBY是一个好的分解。PERSON1和HOBBY属性的交是{SSN}，SSN是PERSON1的键。因此，这个分解是无损的。

8.6.2　依赖保持分解

再次考虑模式HASACCOUNT。它有属性AccountNumber、ClientId、OfficeId，并且其函数依赖为：

```
ClientId, OfficeId → AccountNumber                    (8.6)
AccountNumber → OfficeId                              (8.7)
```

根据无损性测试，因为AccountNumber（两个属性集的交）是第一个模式ACCTOFFICE的键，如下分解是无损的：

```
ACCTOFFICE = (AccountNumber, OfficeId;
                    {AccountNumber → OfficeId})       (8.8)
ACCTCLIENT = (AccountNumber, ClientId; { })
```

尽管分解（8.8）是无损的，但似乎还是有些问题。ACCTOFFICE保持函数依赖（8.7），而与ACCTCLIENT关联的函数依赖集是空。这会导致原始模式中的函数依赖(8.6)的丢失。分解的两个模式都没有可以容纳这个函数依赖的属性；而且，这个函数依赖无法从属于模式ACCTOFFICE和ACCTCLIENT的函数依赖中导出。

从实际来说，这意味着尽管把模式HASACCOUNT的关系分解为ACCTOFFICE和 ACCTCLIENT的关系不会导致信息丢失，但它也许会由于丢失函数依赖而使完整性约束的维护代价提高。

和函数依赖（8.7）可以进行局部校验不同，对关系AcctOffice，函数依赖(8.6)的校验要求在校验前先计算AcctOffice和AcctClient的联结。在这种情况下，我们认为分解没有保持原始模式的依赖。我们现在对其进行定义。

考虑模式$\mathbf{R}=(\overline{R};F)$并假定

$$\mathbf{R}_1=(\overline{R}_1;\ F_1),\ \mathbf{R}_2=(\overline{R}_2;\ F_2),\ \cdots,\ \mathbf{R}_n=(\overline{R}_n;\ F_n)$$

是一个分解。根据定义，F蕴涵每个F_i，因此F蕴涵$\bigcup_{i=1}^{n}F_i$。然而，这个定义并不要求两个依赖集合相等，即$\bigcup_{i=1}^{n}F_i$必须蕴涵F。这个反向蕴涵正是在上述例子中所丢失的。包括依赖(8.6)和(8.7)的HasAccount的函数依赖集并没有被分解（8.8）的依赖的并所蕴涵，它们的并只有一个依赖AccountNumber → OfficeId。正因为这样，(8.8)不是一个依赖保持的分解。

正式地说，若当且仅当

$$\mathbf{R}_1=(\overline{R}_1;\ F_1),\ \mathbf{R}_2=(\overline{R}_2;\ F_2),\ \cdots,\ \mathbf{R}_n=(\overline{R}_n;\ F_n)$$

是一个分解而且函数依赖集F和$\bigcup_{i=1}^{n}F_i$相等，则称$\mathbf{R}=(\overline{R};F)$的一个**依赖保持的分解**(dependency-preserving decomposition)。

F中的一个函数依赖f不在任何F_i中并不意味着这个分解不是依赖保持的，因为f也许会被$\bigcup_{i=1}^{n}F_i$所蕴涵。在这种情况下，保持f作为函数依赖不需要做额外的努力：如果$\bigcup_{i=1}^{n}F_i$被保持，那么f同样也会被保持。只有当f不被$\bigcup_{i=1}^{n}F_i$蕴涵时，这个分解是非依赖保持的，而且f的维护要求一个联结。

根据定义，上述HasAccount分解不是依赖保持的，这正是错误所在。存在于原始模式中而在分解中丢失的依赖成为关系间的约束，它不能进行局部维护。每一次改变分解中的关系时，只有在原始关系重构后才能检测是否满足关系间的约束。为说明这个问题，考虑图8-6的HasAccount分解。

AccountNumber	ClientId	OfficeId
B123	111111111	SB01
A908	123456789	MN08

HasAccount

AccountNumber	OfficeId
B123	SB01
A908	MN08

AcctOffice

AccountNumber	ClientId
B123	111111111
A908	123456789

AcctClient

图8-6　HasAccount关系的分解

如果我们把元组<B567,SB01>加到关系AcctOffice中，把元组<B567,111111111>加到AcctClient中，这两个关系仍满足它们的局部函数依赖（事实上，从（8.8）可看出只有AcctOffice有依赖要被满足）。相反地，在更新后，关系间的函数依赖（8.6）没有被满足，但这并不很明显。为验证这个结论，我们必须联结这两个关系，如图8-7所示。我们现在可看到在更新过的HasAccount关系中，前两个元组违反了约束（8.6）。

AccountNumber	ClientId	OfficeId
B123	111111111	SB01
B567	111111111	SB01
A908	123456789	MN08

HASACCOUNT

AccountNumber	OfficeId
B123	SB01
B567	SB01
A908	MN08

ACCTOFFICE

AccountNumber	ClientId
B123	111111111
B567	111111111
A908	123456789

ACCTCLIENT

图8-7　在插入几行后的HASACCOUNT和它的分解

下一个问题是检测一个分解是否是依赖保持的难度。如果我们已经有了一个分解，它的结果是附属于局部模式的函数依赖的集合，那么可以在线性时间内检测出其是否是依赖保持的。我们只要检测原始集合的每个函数依赖是否被局部模式的函数依赖的并所蕴涵。对每个这样的测试，我们可以使用8.4节介绍的二次属性闭包算法。

实际上，这种情况更复杂。通常，我们（和计算机算法）必须先决定如何拆分属性集以组成分解，此时才能把函数依赖添加到那些属性集中。如果 $\overline{X} \cup \overline{Y} \subseteq \overline{S}$，则我们可以把函数依赖 $\overline{X} \rightarrow \overline{Y}$ 添加到属性集合 $\overline{S}$。下面我们会更正式地说明这个问题。

考虑模式**R**=($\overline{R}$; F)，对模式**R**的关系**r**和属性集合 $\overline{S}$，有 $\overline{S} \subseteq \overline{R}$。如果 $\overline{S}$ 是**R**分解的一个模式，在 $\pi_{\overline{S}}(\mathbf{r})$ 上可以保证的函数依赖是 $\overline{X} \rightarrow \overline{Y} \in F^+$，有 $\overline{X}\,\overline{Y} \subseteq \overline{S}$。这会产生如下概念:

$$\pi_{\overline{S}}(F) = \left\{\overline{X} \rightarrow \overline{Y} \mid \overline{X} \rightarrow \overline{Y} \in \overline{F}^+ \text{ and } \overline{X} \cup \overline{Y} \subseteq \overline{S}\right\}$$

这称为函数依赖集F在属性集 $\overline{S}$ 上的**投影**（projection）。

对函数依赖投影的概念找到了一种在只知道如何拆分原始模式时构造分解的方法。如果**R**=($\overline{R}$;F)是一个模式，$\overline{R}_1$，…，$\overline{R}_n$是属性的子集，有 $\overline{R} = \bigcup_{i=1}^{n} \overline{R}_i$，那么模式（$\overline{R}_1$; $\pi_{\overline{R}_1}(F)$），…，（$\overline{R}_n$; $\pi_{\overline{R}_n}(F)$）的集合是一个分解。给定分解中的一个属性集，因为通过投影，我们总能决定相应的函数依赖集，因此当指定模式分解时，通常忽略掉函数依赖。

因此构造分解涉及计算函数依赖的投影。这个计算涉及对F闭包的计算[⊖]，它与F大小的指数成正比。如果这个代价被算到计算依赖保持的代价中，这个检测同样是指数级的。因此，为了测试一个分解的无损性，令人感兴趣的是我们不需要计算属于局部模式的函数依赖。早期呈现的测试使用函数依赖的原始集合F，并且与F大小呈线性关系。

综上所述，我们考虑模式分解的两个重要性质：无损性和依赖保持。我们也可以看一个例子，一个关系（HASACCOUNT）是无损的，但分解为BCNF却不是依赖保持的（就像我们将要看到的，它没有一个BCNF分解会满足这两个性质）。这两个性质哪一个更重要？答案是无损性是必需的，而依赖保持尽管也是我们所希望的，但却是可选的。原因是有损分解会丢失

⊖ 有一种方法可以避免计算完全的闭包F^+，但最坏情况下的复杂度是一样的。

原始数据库中的信息，这是不可接受的。而不保持函数依赖的分解在数据库改变并且需要检测关系间约束时才会导致计算开销。

8.7 分解BCNF的一个算法

我们现在准备介绍第一个分解算法。假设**R**=($\overline{R}$;*F*)是一个关系模式，它不是BCNF的。图8-8的算法通过不断拆分**R**为更小的子模式来构造一个新的分解，因此在每一步，新的数据库模式中违反BCNF的函数依赖比前一次迭代更少（参见练习8.13）。因此，这个算法总会终止，并且结果中的所有模式都符合BCNF。

Input: $\mathbf{R} = (\overline{R};\ F)$
Decomposition := **R**
while there is a schema $\mathbf{S} = (\overline{S};\ F')$ in *Decomposition* that is not in BCNF **do**
　　/* Let $\overline{X} \rightarrow \overline{Y}$ be an FD in F such that $\overline{X}\,\overline{Y} \subseteq \overline{S}$
　　and it violates BCNF. Decompose using this FD */
　　Replace **S** in *Decomposition* with schemas $\mathbf{S}_1 = (\overline{X}\,\overline{Y};\ F_1)$
and $\mathbf{S}_2 = (\overline{S} - \overline{Y}) \cup \overline{X};\ F_2$, where F_1 and F_2 are all the FDs
from F' that involve only attributes in their respective schemas
end
return *Decomposition*

图8-8　无损分解为BCNF

为了理解BCNF分解算法的工作原理，再次考虑HASACCOUNT例子。这个模式不满足BCNF，因为函数依赖AccountNumber →OfficeId的左边不是超键。因此，在图8-8的while循环中我们使用这个函数依赖来拆分HASACCOUNT。结果正是我们在（8.8）中看到的分解。

下一个例子更复杂也更抽象。考虑关系模式**R**=($\overline{R}$; *F*)，其中 $\overline{R}$ =*ABCDEFGH*（回忆*A*、*B*等表示属性名），函数依赖集*F*是

$ABH \rightarrow C$

$A \rightarrow DE$

$BGH \rightarrow F$

$F \rightarrow ADH$

$BH \rightarrow GE$

为应用BCNF分解算法，我么首先需要确定违背BCNF的函数依赖，即左边不是超键的那些函数依赖。我们可以看到第一个函数依赖没有违背BCNF，因为属性闭包 $(ABH)^+$（用图8-3所示的算法进行计算）包含所有模式属性，因此*ABH*是超键。然而，第二个函数依赖，$A \rightarrow DE$违背了BCNF，因为*A*的属性闭包是*ADE*，因此*A*不是超键。我们可以使用这个函数依赖拆分**R**:

$\mathbf{R}_1 = (ADE;\ \{A \rightarrow DE\})$

$\mathbf{R}_2 = (ABCFGH;\ \{\ ABH \rightarrow C\ ,\ BGH \rightarrow F\ ,\ F \rightarrow AH,\ BH \rightarrow G\})$

注意，拆分$F \rightarrow ADH$为{ $F \rightarrow AH$，$F \rightarrow D$}，拆分$BH \rightarrow GE$为{ $BH \rightarrow G$，$BH \rightarrow E$ }，其中一些函数依赖$F \rightarrow D$和$BH \rightarrow E$就会丢失，因为没有新的模式包含这些函数依赖所用的所有属性。然

而，情况看起来还是不错，因为从新模式$\mathbf{R}_1$和$\mathbf{R}_2$中的函数依赖$F \rightarrow AH$, $A \rightarrow DE$可以推导出函数依赖$F \rightarrow D$。因此，如果维护这两个依赖的有效性，则$F \rightarrow D$的有效性也同样被维护。检验这两个依赖是容易的，因为每一个依赖的属性被限制在单个关系中。类似地，同样可以导出$BH \rightarrow E$，因为限制到嵌入在$\mathbf{R}_1$和$\mathbf{R}_2$中的函数依赖的BH的属性闭包中包含E。（使用图8-3的算法进行验证！）因此上述分解是依赖保持的。

容易看到，$\mathbf{R}_1$满足BCNF，因此BCNF分解算法必须关注$\mathbf{R}_2$。函数依赖$ABH \rightarrow C$ 和$BGH \rightarrow F$没有违背$\mathbf{R}$中的BCNF，因为ABH和BGH都是超键。因此，它们也不违背$\mathbf{R}_2$的BCNF（它只有$\mathbf{R}$属性的一个子集）。明显违背BCNF的函数依赖是$F \rightarrow AH$，因此算法会找到这个函数依赖，并对$\mathbf{R}_2$进行相应地拆分。

$\mathbf{R}_{21} = (FAH; \{F \rightarrow AH\})$

$\mathbf{R}_{22} = (FBCG; \{\})$

现在模式$\mathbf{R}_{21}$和$\mathbf{R}_{22}$都满足BCNF（$\mathbf{R}_{22}$是平凡的，因为它没有函数依赖）。然而，代价是$\mathbf{R}_2$中的函数依赖$ABH \rightarrow C$ ，$BGH \rightarrow F$和$BH \rightarrow G$现在没有容身之地。而且，这些函数依赖都不能用嵌在$\mathbf{R}_1$、$\mathbf{R}_{21}$和$\mathbf{R}_{22}$中的函数依赖推导出来。比如，在考虑其函数依赖集的情况下，计算$(ABH)^+$得到$ABHDE$，它不包括C，因此无法推出$ABH \rightarrow C$。这样，我们得到了$\mathbf{R}$的三个BCNF模式：$\mathbf{R}_1$、$\mathbf{R}_{21}$和$\mathbf{R}_{22}$的一个非依赖保持分解。

这个分解并不是唯一的。例如，如果我们的算法在第一次迭代中选择出$F \rightarrow ADH$，则第一个分解将是

$\mathbf{R}_1' = (FADH; \{F \rightarrow ADH\})$

$\mathbf{R}_2' = (FBCEG; \{\})$

这并不是最后的差别。尽管这两个模式都满足BCNF，但在分解$\mathbf{R}_1$、$\mathbf{R}_{21}$和$\mathbf{R}_{22}$中的一些函数依赖现在丢失了。

BCNF分解算法的性质

首先最重要的是，图8-8的BCNF分解算法总会产生一个无损分解。为说明这一点，考虑包含属性集 $\overline{X}\ \overline{Y}$ 和 $(\overline{S} - \overline{Y}) \cup \overline{X}$ 的两个模式，来代替算法中的模式$\mathbf{S}=(\overline{S}, F')$。注意，$\overline{X}\ \overline{Y} \cap ((\overline{S} - \overline{Y}) \cup \overline{X}) = \overline{X}$，由于 $\overline{X} \rightarrow \overline{Y} \in F$，因此 $\overline{X}\ \overline{Y} \cap ((\overline{S} - \overline{Y}) \cup \overline{X}) \rightarrow \overline{X}\ \overline{Y}$。根据二元分解的无损性测试可知，$\{\overline{X}\ \overline{Y}, (\overline{S} - \overline{Y}) \cup \overline{X}\}$是$\mathbf{S}$的无损分解。这意味着在我们算法的每一步，我们用无损分解来代替一个模式。这样，根据练习8.12，这个算法最终产生的分解是无损的。

BCNF算法产生的分解是否总是依赖保持的？我们可以看到情况并非如此。HASACCOUNT的分解（8.8）不是依赖保持的。而且，容易看到没有HASACCOUNT的分解（不仅是这个特定算法产生的）是既无损又依赖保持的。实际上，只有三个分解可以进行，我们可以把它们都检测一下。

最后，我们看到BCNF分解算法是非决定性的。最终的结果取决于While循环中函数依赖被选择的顺序。数据库设计者可以根据个人品味选择分解，也可以根据客观标准选择分解。比如，一些分解也许是依赖保持的，而另外一些分解则不是；一些分解也许会导致更少的函数依赖作为关系间限制（比如，分解$\mathbf{R}_1$、$\mathbf{R}_{21}$、$\mathbf{R}_{22}$在这种情况下就比 $\mathbf{R}_1'$、 $\mathbf{R}_2'$分解好）。一些

属性集更可能被一起查询，因此最好不要在分解中将它们分开。下一节描述了一个通用的方法，以帮助我们选择BCNF分解。

8.8 3NF模式的合成

我们已经看到，一些模式（如图8-7中的HASACCOUNT）在不能再分解为BCNF的情况下仍保持依赖。然而，如果我们愿意分解为3NF而不是BCNF，那么就有可能得到依赖保持分解（但3NF也许会包含冗余）。

在介绍3NF分解算法前，我们介绍*最小覆盖*（minimal cover）的概念，它非常简单。我们知道函数依赖集也许看起来完全不同、但却逻辑上等价。图8-4呈现了一个相当直观的方法来测试相等性。由于可能会有很多函数依赖集等于任何给定的集合，所以我们希望有一个函数依赖集可以视为是“规范的”。但定义一个唯一的规范集不是一件容易的事，但最小覆盖的概念与之很相似。

8.8.1 最小覆盖

F是函数依赖集。F的**最小覆盖**是一个函数依赖集G，它有如下性质：

1) G等价于F（但可能与F不同）。

2) G中的所有函数依赖形如$\overline{X} \rightarrow A$，$A$是单个属性。

3) 通过如下两个方法不可能使G“更小”（并且仍满足前两个性质）：

a. 删除一个函数依赖。

b. 从函数依赖中删除一个属性。

很明显，根据函数依赖分解的规则，很容易把F转换为等价的函数依赖集，使它的右边为单个属性。然而，性质3更微妙。在介绍计算最小覆盖的算法前，我们用一个具体的例子进行说明。

考虑属性集$ABCDEFGH$和如下的函数依赖集F：

$ABH \rightarrow C$

$A \rightarrow D$

$C \rightarrow E$

$BGH \rightarrow F$

$F \rightarrow AD$

$E \rightarrow F$

$BH \rightarrow E$

由于并非所有函数依赖的右边为单个属性，我们可以使用分解规则来获得满足最小覆盖的前两个性质一个函数依赖集：

$ABH \rightarrow C$

$A \rightarrow D$

$C \rightarrow E$

$BGH \rightarrow F$

$$F \rightarrow A \quad (8.9)$$
$$F \rightarrow D$$
$$E \rightarrow F$$
$$BH \rightarrow E$$

可以看到，$BGH \rightarrow F$被$BH \rightarrow E$和$E \rightarrow F$所蕴涵，$F \rightarrow D$被$F \rightarrow A$和$A \rightarrow D$所蕴涵。因此，还剩下

$$ABH \rightarrow C$$
$$A \rightarrow D$$
$$C \rightarrow E$$
$$F \rightarrow A \quad (8.10)$$
$$E \rightarrow F$$
$$BH \rightarrow E$$

通过计算属性闭包，容易验证这些函数依赖均不是冗余的；即我们不能简单地从这个集合丢掉一个函数依赖而不牺牲与原始集合F的等价性。然而，这个结果集是F的最小覆盖吗？答案是否定的，因为从第一个函数依赖可以删除属性A。$ABH \rightarrow C$被属性集

$$BH \rightarrow C$$
$$A \rightarrow D$$
$$C \rightarrow E$$
$$F \rightarrow A \quad (8.11)$$
$$E \rightarrow F$$
$$BH \rightarrow E$$

所蕴涵，$BH \rightarrow C$被集合(8.10)所蕴涵。第一个事实可以根据（8.11）通过计算ABH的属性闭包进行验证，第二个事实可以根据（8.10）通过计算BH的属性闭包进行验证。很容易得出这两个集合是等价的，因此(8.10)不是最小覆盖。令人感兴趣的是，在冗余属性A被移除后，集合（8.11）仍然不是最小覆盖，因为$BH \rightarrow E$是冗余的。去除这个函数依赖后最终得到了最小覆盖。

计算最小覆盖的算法如图8-9所示。步骤1根据函数依赖的右边对它进行简单拆分。比如，$\overline{X} \rightarrow AB$转换为 $\overline{X} \rightarrow A$和 $\overline{X} \rightarrow B$。

Input: a set of FDs F
Output: G, a minimal cover of F

Step 1: $G := F$, 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 G
Step 3: Remove all redundant FDs from G

return G

图8-9　最小覆盖的计算

步骤2检测G中左边的每个属性是否存在冗余。也就是说，对每个函数依赖 $\overline{X} \rightarrow A \in G$和每个属性$B \in \overline{X}$，我们需要检测$(\overline{X}-B) \rightarrow A$是否被$G$所蕴涵。如果手工进行的话，这是一项非常枯燥的工作。在上述例子中，当检测到$BH \rightarrow C$被函数依赖集（8.10）所蕴涵时，就执行这个步骤，这就允许我们删除$ABH \rightarrow C$中的冗余属性A。步骤三通过另一个枯燥的算法执行：对每个$g \in G$，检测函数依赖g是否被$G-g$所蕴涵。

注意，图8-9的算法中，步骤2和3不能反过来进行，在执行步骤2之前执行步骤3并不总是返回最小覆盖。事实上，我们已经看到这个现象：通过从（8.9）移去冗余函数依赖得到集合（8.10）；通过删除冗余属性得到集合（8.11）。然而，这个结果仍有冗余函数依赖（$BH \rightarrow E$）。另一方面，如果我们先从（8.9）移去冗余属性，可得到

$BH \rightarrow C$

$A \rightarrow D$

$C \rightarrow E$

$BH \rightarrow F$

$F \rightarrow A$

$F \rightarrow D$

$E \rightarrow F$

$BH \rightarrow E$

接着移去冗余的函数依赖$BH \rightarrow F$，$F \rightarrow D$和$BH \rightarrow E$会产生如下最小覆盖

$$\begin{aligned} &BH \rightarrow C \\ &A \rightarrow D \\ &C \rightarrow E \\ &F \rightarrow A \\ &E \rightarrow F \end{aligned} \tag{8.12}$$

8.8.2 通过模式合成的3NF分解

构造依赖保持的3NF分解的算法与BCNF的原理大不相同。3NF算法从单独属性开始把它们组合为模式，而不是从一个大的模式开始逐步分割它。因此，它被称为3NF**合成**（synthesis）。给定模式$\mathbf{R}=(\overline{R}; F)$，$\overline{R}$是属性的超集，$F$是函数依赖的集合，该算法分为四步。

1) 找到F的最小覆盖G。

2) 把G划分为函数依赖集$G_1,\cdots,G_n$，每个G_i的左半部相同。（没有必要假定不同的G_i左边的部分相同，但把左边部分相同的集合进行合并是一个好的思想。）

3) 对每个G_i，组成关系模式$\mathbf{R}_i=(\overline{R}_i; G_i)$，$\overline{R}_i$是$G_i$中所涉及到的所有属性的集合。

4) 如果某个$\overline{R}_i$是模式$\mathbf{R}$的超键（即$(\overline{R}_i)_F{}^+=\overline{R}$），则完成分解，即$\mathbf{R}_1,\cdots,\mathbf{R}_n$是所要的分解。如果没有$\overline{R}_i$是$\mathbf{R}$的超键，让$\overline{R}_0$成为包含$\mathbf{R}$的键的属性的集合，$R_0=(\overline{R}_0; \{\})$为一个新的模式。那么$\mathbf{R}_0, \mathbf{R}_1,\cdots,\mathbf{R}_n$就是所要求的模式。

注意，完成步骤3后所获得的模式的集合也许还不是一个分解，因为$\mathbf{R}$的一些属性可能会丢失（参见下面的例子）。然而，任何丢失的属性在步骤4中会重新找到，因为这些属性必须

是**R**键的一部分（练习8.25）。

容易看出，上述算法的结果是一个依赖保持的分解。通过构造，G中的每个函数依赖都被包括在内，因此$G=\cup G_i$。根据最小覆盖的定义，$G^+ = F^+$，F被保持了。

同样很明显，每个$\mathbf{R}_i$是一个3NF模式，因为只有与$\mathbf{R}_i$关联的函数依赖在G_i中，而且它们左边的部分相同（因此是$\mathbf{R}_i$的一个超键）。似乎每个$\mathbf{R}_i$也满足BCNF，但不要被蒙蔽了：G_i中的函数依赖也许并不是唯一在$\mathbf{R}_i$中成立的，因为G^+也许有其他的函数依赖，它们的属性集完全被包含在$\overline{R}_i$中。为说明这一点，考虑我们曾试过的真实模式HASACCOUNT的一个稍微的修改。这里$\overline{R}$包含属性AccountNumber、ClientId、OfficeId和DateOpened，F包含函数依赖ClientId、OfficeId→AccountNumber和AccountNumber→OfficeId、DateOpened。上述算法会得到两个模式：

```
R1 = ({ClientId,OfficeId,AccountNumber},
            {ClientId,OfficeId → AccountNumber})
R2 = ({AccountNumber,OfficeId,DateOpened},
            {AccountNumber → OfficeId,DateOpened})
```

经过仔细检查可以发现，尽管AccountNumber→OfficeId没有在$\mathbf{R}_1$中显式地指定，对那个模式的属性它是一定成立的，因为这个函数依赖被$\mathbf{R}_2$隐含说明。

为理解考虑$\mathbf{R}_1$时要考虑$\mathbf{R}_2$的函数依赖的原因，注意属性对AccountNumber，OfficeId表示$\mathbf{R}_1$和$\mathbf{R}_2$中同样的真实世界联系，因此如果$\mathbf{R}_2$中的元组遵从这对属性上的约束，而$\mathbf{R}_1$中的元组不遵从属性上的约束，这会导致不一致。

回到我们的最初问题，上述讨论表明我们的合成算法所产生的模式也许有左边部分不同的函数依赖，因此遵从3NF并不是很明显。然而，可以证明上述算法总会产生3NF分解（参见练习8.14）。

最后的问题是合成算法是否会产生输入模式的无损分解。答案是肯定的，但证明这一点比证明3NF性质更困难。尽管不是很明显，实际上获得无损性是那个算法的步骤4的唯一目的。

对一个3NF合成更复杂的例子，考虑（8.9）中带有函数依赖的模式。这个集合的最小覆盖是（8.12）。由于这里没有两个函数依赖左边的部分相同，我们以下列模式结束：$(BHC;BH\rightarrow C)$，$(AD;A\rightarrow D)$，$(CE;C\rightarrow E)$，$(FA;F\rightarrow A)$以及$(EF;E\rightarrow F)$。注意，这些模式没有一个组成全部属性集的超键。比如，BHC的属性闭包并不包括G。事实上，属性G没有包括在任何模式中！因此，根据我们对步骤4目的评价，这个分解并不是无损的（事实上，它不是一个分解！）。为使它是无损的，我们进行步骤4，加入模式$(BCGH;\{\})$。

8.8.3 通过3NF合成的BCNF分解

那么，如何利用3NF合成帮助设计BCNF数据库模式？答案很简单：为了把一个模式分解为BCNF关系，首先不要使用BCNF算法。应该使用3NF合成，它是无损的而且保证了保持依赖。如果结果模式符合BCNF（正如我们前面的例子），则不必再采取进一步的措施。然而，如果结果中的某个模式不在BCNF中，则使用BCNF算法来拆分它，直到没有违背BCNF的模式存在为止。对每个3NF合成产生的非BCNF模式重复这一步。

这个方法的优点是，如果一个无损且依赖保持的分解存在，则3NF合成很可能会找到它。如果在第一阶段后一些模式不是BCNF，那么一些函数依赖的丢失是不可避免的（练习8.26）。

但至少我们已经尽力了。下面用一个完整的例子来说明上述方法。

例8.8.1　模式合成与分解的结合

假定属性集是St(学生)、C(课程)、Sem(学期)、P(教授)、T(时间)和R(地点)，其中有下列函数依赖:

```
St C Sem -> P
P Sem -> C
C Sem T -> P
P Sem T -> C R
P Sem C T -> R
P Sem T -> C
```

这些函数依赖可以应用在一个大学中，同一门课程的多个班级可以在同一个学期开设(因此提供课程和学期名并不能唯一地确定一个教授)，一个教授一学期只教一门课程（因此提供教授和学期名可以唯一确定一门课程)，一门课程的所有班级在不同的时间教。

我们首先找到上述集合的一个最小覆盖；第一步是拆分函数依赖集的右边部分为单个属性。

```
St C Sem -> P
P Sem -> C
C Sem T -> P
P Sem T -> C
P Sem T -> R
P Sem C T -> R
P Sem T -> C
```

假定F表示这个函数依赖的集合。最后一个函数依赖是重复的，因此我们从集合中删除它。接着，我们通过消除冗余属性来减少左边部分。比如，为检测左边的部分St C Sem, 我们必须计算几个属性闭包: $(\text{St Sem})_{F^+}$ = {St,Sem}; $(\text{St C})_{F^+}$ = {St,C};$(\text{C Sem})_{F^+}$ = {C,Sem}，这表明在第一个函数依赖中没有冗余属性。同样地，P Sem, C Sem T和P Sem T也不能被减少。然而，检测P Sem C T会带来一些回报: $(\text{P Sem T})_{F^+}$ = P Sem T C R，因此C可以被删除。

上述分析得到如下的函数依赖集，我们加上编号以便进行索引。

```
FD 1: St C Sem -> P
FD 2: P Sem -> C
FD 3: C Sem T -> P
FD 4: P Sem T -> C
FD 5: P Sem T -> R
```

下一阶段是去除冗余函数依赖，同样利用属性集闭包进行检测。由于 $(\text{St C Sem})^{+}_{\{F-\text{FD1}\}}$ =St C Sem, FD1不能被消除。FD2、FD3、FD5均不能删除。然而，FD4是冗余的(由于FD2)，因此它可以被消除。这样，最小覆盖是

```
St C Sem -> P
P Sem -> C
C Sem T -> P
P Sem T -> R
```

这会得到如下依赖保持的3NF分解:

```
(St C Sem P; St C Sem -> P)
(P Sem C; P Sem -> C)
(C Sem T P; C Sem T -> P)
(P Sem T R; P Sem T -> R)
```

容易验证，上述模式没有一个会组成原始模式的一个超键；因此，为了使上述分解是无损的，我们需要加一个模式，它的属性闭包包含所有原始属性。模式(St T Sem P;{})是一个符合要求的属性。

如果你相信3NF合成算法是正确的，并且我们在应用它时没有出错，那么没有必要再对3NF进行检测。然而，我们回顾一下可以看到第一个模式和第三个模式不是BCNF，因为函数依赖P Sem→C 嵌入在第二个模式中。

考虑到P Sem → C，第一个模式的进一步分解产生(P Sem C;P Sem → C)和(P Sem St;{})，这是一个无损分解但函数依赖St C Sem → P没有保持。

考虑到P Sem → C，第三个模式的分解产生(P Sem C;P Sem → C)和(P Sem T;{})，这是另一个无损分解，同样丢失C Sem T → P。

因此，最终的BCNF分解是

```
(P Sem C; P Sem → C)
(P Sem St)
(P Sem T)
(P Sem T R; P Sem T → R)
(St T Sem P)
```

这个分解是无损的，因为我们首先获得了一个无损的3NF分解，接着应用BCNF算法，它可以保持无损性。然而，它不是依赖保持的，因为St C Sem→P和C Sem T→P没有表示在上述模式中。

8.9 第四范式

世界上所有的问题并非都是由不好的函数依赖引起的。考虑如下模式：

PERSON(SSN, PhoneN, ChildSSN)　　(8.13)

我们假定一个人可以有几个电话号码和几个孩子。如下所示是一个可能的关系实例。

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

(8.14)

由于没有非平凡的函数依赖，所以这个模式满足3NF也满足BCNF。然而，很明显它不是一个好的设计，因为它有相当大的冗余。除了通过SSN，电话号码和孩子之间没有特定的关联，因此每个关联于给定SSN的ChildSSN项必须与每个关联于同一个SSN的PhoneN出现在同一个元组中。因此，当添加或删除一个电话号码时候，也需要添加或删除几个元组。如果一个人丢弃所有电话号码，则关于孩子的信息也将会丢失（或者使用NULL值）。

这里似乎可以使用压缩技术。比如，我们可以决定只存储一些元组，只要有一个方法可

以重构原始信息。

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

尽管它更有效，但它并没有解决上述的异常。而且，它会给应用程序加上一些额外的负担，即应用程序必须清楚压缩模式。

在我们讨论BCNF时，我们知道当属性值间的一个特定的语义联系的存储多于一次时，就会产生冗余。对图5-2来说，一个人的SSN为111111111，他住在123 Main St.就是一个例子，这个信息被存储了三次。在这个例子中，问题被追溯到了SSN和Address之间的函数依赖，而SSN不是键（因此可以有多行对应一个SSN值）。然而，语义联系的冗余存储并不仅限于此。在（8.14）所示的关系**r**中，联系SSN-PhoneN和SSN-ChildSSN被存储了多次，但这里并没有涉及函数依赖。这里出现问题是因为有几个属性（本例中PhoneN和ChildSSN），它们的值的集合与另一个属性的单值相关联（本例中是SSN）。由于某个SSN的人有多个孩子，所以特定的SSN值和特定的PhoneN值之间的联系被存储了多次。注意，这个联系满足如下性质：

$$\mathbf{r} = \pi_{\text{SSN,PhoneN}}(\mathbf{r}) \bowtie \pi_{\text{SSN,ChildSSN}}(\mathbf{r}) \tag{8.15}$$

当一个模式的所有合法实例都要求满足（8.15）时，这个性质被称为联结依赖。当一个企业的特征被值的集合所描述时，就会引起联结依赖。用E-R方法进行数据库设计，我们看到这样的特征被表示为集合值（set-valued）属性，并且把它们转换为关系模式中的属性是不方便的。特别地，当一个实体类型或联系类型有几个集合值属性，就会产生联结依赖。

条件（8.15）看起来应该比较熟悉。它保证将**r**分解为两个表$\pi_{\text{SSN,PhoneN}}(\mathbf{r})$和$\pi_{\text{SSN,ChildSSN}}(\mathbf{r})$时是无损的。本例当然符合这个要求，但它并不是我们所关心的。这个条件还告诉我们一些**r**的情况：一个联结依赖表明语义联系的存储在**r**的一个实例中会有冗余。

从形式上说，$\overline{R}$是属性集合。**联结依赖**（Join Dependency，JD）是如下的一种约束：

$$\overline{R} = \overline{R}_1 \bowtie \cdots \bowtie \overline{R}_n$$

其中，$\overline{R}_1 \cdots \overline{R}_n$是$\overline{R}$的一个分解。回忆前面我们定义的关系实例满足函数依赖的定义。我们现在对JD定义同样的概念：$\overline{R}$的一个关系实例**r**满足上述联结依赖的条件为：

$$\mathbf{r} = \pi_{\overline{R}_1}(\mathbf{r}) \bowtie \cdots \bowtie \pi_{\overline{R}_n}(\mathbf{r})$$

设**R**=($\overline{R}$; Constraints)是一个关系模式，其中$\overline{R}$是属性的集合，Constraints是FD和JD的集合。和仅有FD的情况一样，当且仅当它满足Constraints中所有约束时，属性集$\overline{R}$的一个关系是**R**的一个合法实例。

特别应该关注的是所谓的**二元**（binary）联结依赖，也称为**多值依赖**（MultiValued Dependency，MVD）。它们是形如$\overline{R} = \overline{R}_1 \bowtie \overline{R}_2$的JD。关系模式PERSON(8.13)所展现的冗余就是由这种特定的联结依赖所引起的。*第四范式*（在[Fagin 1977]中引入）被用来防止这种类型的冗余。

MVD对关系实例的约束方式与FD一样，因此一个关系模式的描述必须包括这两者。所以我们可以描述一个关系模式**R**为($\overline{R}$; D)，D是FD和MVD的集合。JD**蕴涵**的定义与FD蕴涵的定义类似。设S是JD的集合（可能有FD），d是一个JD(或FD)。如果满足S中所有依赖的每个关系实例**r**同样满足d，那么S蕴涵d。在下一节，我们会介绍一个MVD如何被MVD集合所蕴涵。一个关系模式，**R**=($\overline{R}$; D)被认为是第四范式，如果对每个被D蕴涵的MVD $\overline{R} = \overline{X} \bowtie \overline{Y}$，下面两个条件中的一个为真:

- $\overline{X} \subseteq \overline{Y}$ 或 $\overline{Y} \subseteq \overline{X}$ （MVD是平凡的）
- $\overline{X} \cap \overline{Y}$ 是 $\overline{R}$ 的超键（($\overline{X} \cap \overline{Y}$)→ $\overline{R}$ 被D蕴涵）

那么关系模式**R** = ($\overline{R}$; D) 就是**第四范式**（Fourth Normal Form，4NF）

容易看出，PERSON不是一个4NF模式，因为MVD PERSON={SSN PhoneN} ⋈ {SSN ChildSSN}成立，而SSN={SSN PhoneN} ∩ {SSN ChildSSN}不是超键。这里直觉会告诉我们什么？如果SSN是一个超键，那么对每个SSN的值，最多只有一个PhoneN的值和一个ChildSSN的值，因此没有冗余。

1. 4NF和BCNF

可以看到，满足4NF模式同样满足BCNF模式（即4NF弥补了BCNF留下的漏洞）。为说明这一点，假定**R**=($\overline{R}$; D)是一个4NF模式，$\overline{X} \rightarrow \overline{Y}$ 是**R**中成立的一个非平凡函数依赖。为说明4NF模式同样是BCNF模式，我们必须声明 $\overline{X}$ 是**R**的超键。为简单起见，假定 $\overline{X}$ 和 $\overline{Y}$ 是不相交的。那么 $\overline{R}_1 = \overline{X}\ \overline{Y}$，$\overline{R}_2 = \overline{R} - \overline{Y}$ 是**R**的无损分解，利用8.6.1节的二元模式分解的无损性测试可以证明。因此，$\overline{R} = \overline{R}_1 \bowtie \overline{R}_2$是一个二元联结依赖，即**R**中成立的MVD。但根据4NF定义，下面任一个会成立：$\overline{X}\,\overline{Y} \subseteq \overline{R} - \overline{Y}$（不可能）或 $\overline{R} - \overline{Y} \subseteq \overline{X}\,\overline{Y}$（这意味着 $\overline{R} = \overline{X}\ \overline{Y}$ 和 $\overline{X}$ 是超键）或 $\overline{R}_1 \cap \overline{R}_2$ (= $\overline{X}$) 是超键。这意味着**R**中每个非平凡函数依赖满足BCNF的需求。

同样可以看到（但更难做到），如果**R**=($\overline{R}$;D)中D只包含FD，那么当且仅当**R**是BCNF时**R**是4NF（参见[Fagin 1977]）。换句话说，在（除了FD）MVD必须被说明的设计环境下，4NF是BCNF需求的一个扩展。

2. 设计4NF模式

因为4NF包含BCNF，我们不可能找到一个通用的算法，对一个任意的关系构造一个依赖保持且无损的4NF分解。和BCNF一样，无损分解为4NF总是可以达到的。这样一个算法与BCNF的算法很类似。它是一个从原始模式开始的一个迭代过程，每一步产生有更少MVD违背4NF的一个分解：如果$\mathbf{R}_i$=($\overline{R}_i$; D_i)是这样一个中间模式，D_i蕴涵一个违背4NF的MVD，$\overline{R}_i = \overline{X} \bowtie \overline{Y}$，这个算法把$\mathbf{R}_i$替换为一对模式($\overline{X}$; $D_{i,1}$) 和($\overline{Y}$; $D_{i,2}$)，它们两个都不包含那个MVD。最终，将不会留下任何违背4NF要求的MVD。

关于这个算法有两个要点要强调一下。第一，如果$\mathbf{R}_i$=($\overline{R}_i$; D_i)是一个模式，$\overline{S} \rightarrow \overline{T} \in D$（为简单起见，假定 $\overline{S}$ 和 $\overline{T}$ 不相交），那么这个FD蕴涵MVD $\mathbf{R}_i = \overline{S}\ \overline{T} \bowtie (\overline{R}_i - \overline{T})$。这样，4NF分解算法可以把FD看作MVD。4NF分解算法中另一个不太明显的问题是决定分解中成立的依赖集。即，如果根据MVD $\overline{R}_i = \overline{X} \bowtie \overline{Y}$ 分解$\mathbf{R}_i$=($\overline{R}_i$; D_i)，在分解的结果中，哪个依赖集会在 $\overline{X}$ 和 $\overline{Y}$ 上成立？答案是 $\pi_{\overline{X}}(D_i^+)$ 和 $\pi_{\overline{Y}}(D_i^+)$，即 D_i^+ 在 $\overline{X}$ 和 $\overline{Y}$ 上的投影。这里 D_i^+ 是D_i的闭包，即所有被D_i蕴涵的FD和MVD的集合（8.10节为MVD蕴涵提供了一个推理规则集）。FD在

属性集上的投影在8.6.2节已经定义。对一个MVD，$\overline{R}_i = \overline{V} \bowtie \overline{W}$ 属于 D_i^+，它的投影 $\pi_{\overline{X}}(\overline{R}_i = \overline{V} \bowtie \overline{W})$定义为 $\overline{X}=(\overline{X} \cap \overline{V}) \bowtie (\overline{X} \cap \overline{W})$（如果 $\overline{V} \cap \overline{W} \subseteq \overline{X}$），否则它无定义。很容易验证对MVD来说，这个投影规则是正确的（参见练习8.22）。

为说明这个算法，考虑一个模式，其属性为$ABCD$，MVD为$ABCD=AB \bowtie BCD$，$ABCD=ACD \bowtie BD$，$ABCD=ABC \bowtie BCD$。应用第一个MVD，我们得到如下分解：AB,BCD。剩余MVD在AB上的投影未定义。第三个MVD在BCD上的投影是$BCD=BC \bowtie BCD$，这是一个平凡MVD，第二个MVD在BCD上的投影是$BCD=CD \bowtie BD$。根据最后一个MVD，我们可以分解BCD产生如下最终结果：AB,BD,CD。注意，如果我们根据第三个MVD首先分解$ABCD$，最终结果将变为AB,BC,BD,CD。

4NF的设计理论并不像3NF和BCNF理论发展的那么好，并且没有多少已知的算法。一般建议读者先分解为3NF，接着根据上述算法进一步分解不好的（非4NF）模式。在更复杂的层次上，在[Beeri and Kifer 1986a, 1986b, 1987]中有论述，它们开发了一个设计理论和相应的算法，通过合成new(!)属性来改正设计问题。这些高级问题在下一节将会简要论述。

3. 第五范式

我们不打算在本书中讲述第五范式。知道存在第五范式就够了，数据库设计者通常不需要关心它。5NF与4NF类似，它是基于联结依赖的，但不像4NF，它会把没有被超键蕴涵的所有非平凡JD(不止是二元的)排除掉。

8.10 高级4NF设计*

8.9节提出4NF算法的目的是使你熟悉MVD和4NF，但它只是4NF设计过程的开端。在本节，我们提供一些更有深度的信息，解释在有FD和MVD的情况下，设计数据库模式的主要困难，并给出解决的方法。特别地，我们会解释为什么4NF分解算法并没有完全解决冗余问题，为什么在有MVD的情况下BCNF可能会存在某些不足。有关的更多细节，你可以查找参考文献。

8.10.1 MVD和它们的性质

多值依赖是二元联结依赖。然而，和通用的联结依赖不同，它们有一些类似于FD的好的代数性质。特别地，就像FD有Armstrong公理一样，可以用一套语法规则从给定的MVD集合推导出新的MVD。当我们使用MVD的一个特定符号时，这些规则的形式特别简单：按照惯例，对关系模式$\mathbf{R}=(\overline{R}; D)$的形如 $\overline{R} = \overline{V} \bowtie \overline{W}$ 的多值依赖表示为 $\overline{X} \twoheadrightarrow \overline{Y}$，其中 $\overline{X} = \overline{V} \cap \overline{W}$ 和 $\overline{X} \cup \overline{Y} = \overline{V}$ 或者 $\overline{X} \cup \overline{Y} = \overline{W}$。因此，$\overline{X} \twoheadrightarrow \overline{Y}$ 是 $\overline{R} = \overline{X}\ \overline{Y} \bowtie \overline{X}(\overline{R} - \overline{Y})$的同义词。

请思考一下这个概念背后所隐藏的一些简单结论。当一个属性A的单值与属性B的一个值的集合，以及属性C的一个值的集合相关联时，会出现MVD。包含在 $\overline{X}$ 中的属性A可以看作一个独立的变量，它的值决定（用符号 $\twoheadrightarrow$ 表示）关联的B和C的值集合，B和C一个包含在 $\overline{Y}$ 中，另一个包含在 $\overline{X}\ \overline{Y}$ 的补中。比如，(8.13)中PERSON关系的MVD：SSN PhoneN$\bowtie$SSN ChildSSN可以表示为SSN $\twoheadrightarrow$ PhoneN或SSN $\twoheadrightarrow$ ChildSSN。

另外，把左边相同的MVD进行合并是比较方便的。例如，$\overline{X} \twoheadrightarrow \overline{Y}$ 和 $\overline{X} \twoheadrightarrow \overline{Z}$ 可表示为 $\overline{X} \twoheadrightarrow \overline{Y} \mid \overline{Z}$。很容易看到这样一对MVD等价于形如 $\overline{X}\ \overline{Y} \bowtie \overline{X}\ \overline{Z} \bowtie \overline{X}(\overline{R} - \overline{Y}\ \overline{Z})$的联结依赖。表示为 $\overline{X} \twoheadrightarrow \overline{Y} \mid \overline{Z}$ 不仅对推理系统是方便的，同样可以表示哪里出现冗余：如果属性

$\overline{X}$，$\overline{Y}$，$\overline{Z}$包含在一个关系模式中，那么$\overline{Y}$和$\overline{Z}$之间的关联很可能会被冗余地存储。我们在PERSON关系的语境下看到了这个问题，后面还会对它进行讨论。

使用这个符号，我们现在引入一个推理系统，可以用来决定FD和MVD的蕴涵。这个扩展的系统包含Armstrong公理和如下规则。

1. FD-MVD合成

这些规则混合了FD和MVD。

- 复制（Replication）：$\overline{X} \rightarrow \overline{Y}$蕴涵$\overline{X} \twoheadrightarrow \overline{Y}$。
- 结合（Coalescence）：如果$\overline{W} \subset \overline{Y}$和$\overline{Y} \cap \overline{Z} = \varnothing$，那么$\overline{X} \twoheadrightarrow \overline{Y}$和$\overline{Z} \rightarrow \overline{W}$蕴涵$\overline{X} \rightarrow \overline{W}$。

2. 仅对MVD的规则

这些规则有些类似于FD规则，有些是新的。

- 自反性（Reflexivity）：对每个关系，$\overline{X} \twoheadrightarrow \overline{X}$成立。
- 增广性（Augmentation）：$\overline{X} \twoheadrightarrow \overline{Y}$蕴涵$\overline{X}\ \overline{Z} \twoheadrightarrow \overline{Y}$。
- 相加性（Additivity）：$\overline{X} \twoheadrightarrow \overline{Y}$和$\overline{X} \twoheadrightarrow \overline{Z}$蕴涵$\overline{X} \twoheadrightarrow \overline{Y}\ \overline{Z}$。
- 投影（Projectivity）：$\overline{X} \twoheadrightarrow \overline{Y}$和$\overline{X} \twoheadrightarrow \overline{Z}$蕴涵$\overline{X} \twoheadrightarrow \overline{Y} \cap \overline{Z}$和$\overline{X} \twoheadrightarrow \overline{Y} - \overline{Z}$。
- 传递性（Transitivity）：$\overline{X} \twoheadrightarrow \overline{Y}$和$\overline{Y} \twoheadrightarrow \overline{Z}$蕴涵$\overline{X} \twoheadrightarrow \overline{Z} - \overline{Y}$。
- 伪传递性（Pseudotransitivity）：$\overline{X} \twoheadrightarrow \overline{Y}$和$\overline{Y}\ \overline{W} \twoheadrightarrow \overline{Z}$蕴涵$\overline{X}\ \overline{W} \twoheadrightarrow \overline{Z} - (\overline{Y}\ \overline{W})$。
- 补（Complementation）：$\overline{X} \twoheadrightarrow \overline{Y}$蕴涵$\overline{X} \twoheadrightarrow \overline{R} - \overline{X}\ \overline{Y}$，其中$\overline{R}$是模式中所有属性的集合。

这些规则首先出现在[Beeri et al. 1977]，但[Maier 1983]就这个主题进行了更系统、更易理解的介绍。这些规则是正确的，对任何关系，只要规则前提成立，规则的结果就一定成立。例如，像对无损性一样，复制使用同样的推理规则：如果$\overline{X} \rightarrow \overline{Y}$，那么$\overline{R}$分解为$\overline{R}_1 = \overline{X}\ \overline{Y}$和$\overline{R}_2 = \overline{X}(\overline{R} - \overline{Y})$是无损的；因此$\overline{R} = \overline{X}\ \overline{Y} \bowtie \overline{X}(\overline{R} - \overline{Y})$且有$\overline{X} \twoheadrightarrow \overline{Y}$。

然而，一个显著的事实是给定一个包含MVD和FD的集合s，以及可以是FD或MVD的一个依赖d，s当且仅当通过对δ中依赖的上述规则（加上Armstrong公理）的一个纯粹的句法应用可以推出d。FD一个类似的性质之前称为*完备性*（completeness）。证明上述推导规则的正确性是一个不错的练习（参见练习8.21）。完备性则更难证明。有兴趣的读者可以参考[Beeri et al. 1977，Maier 1983]。

8.10.2 设计4NF的困难性

MVD推理规则是十分有用的，因为它们有助于消除冗余的MVD和FD。和只有FD的情况一样，使用非冗余的依赖集可以改善前面描述的4NF分解算法的设计。然而，即使在没有冗余依赖的情况下还有可能出错。我们使用一些例子来说明这些问题。考虑三个问题：依赖丢失、冗余、使用FD和MVD进行设计。

1. 一个合同的例子

考虑模式

CONTRACTS(Buyer, Vendor, Product, Currency)

其中，形如（John Doe, Acme, Paper Clips, USD）的元组表示购买者John Doe有一份合同，使用美元从Acme公司购买纸夹。假定我们的关系表示购买者和公司之间的一个国际网络合同。尽管这个合同使用美元结算，但如果Acme用DM（德国马克，假设它是一家德国公司）销售

它的一些产品，那么把这个合同存储为两个元组也许更方便：一个使用USD表示的金融信息，另一个使用DM。通常，CONTRACTS满足如下规则：如果一个公司使用几种货币结算，那么每个合同均用一个元组描述。这可以用如下的MVD描述：

```
Buyer Vendor ↠ Product | Currency
```
(8.16)

更直接地，这个MVD意味着

CONTRACTS = (Buyer Vendor Product) ⋈ (Buyer Vendor Currency)

这个国际网络的第二个规则是，如果两个供应商供应一种特定的产品和接受一种特定的货币，而且如果那个产品的购买者有一份与其中一个供应商签署的合同来买特定产品，那么这个购买者必须与另一个供应商也有购买该种产品的合同。例如，如果除了上述元组，CONTRACTS包含(Mary Smith, OfficeMin, Paper Clips, USD),那么它必须包含元组(John Doe, OfficeMin, Paper Clips, USD)和(Mary Smith, Acme, Paper Clips, USD)。这种类型的约束可以使用如下MVD表示：

```
Product Currency ↠ Buyer | Vendor
```
(8.17)

让我们试着用4NF分解算法。如果我们首先用依赖（8.16）进行分解，得到如下无损分解：

```
(Buyer, Vendor, Product)
(Buyer, Vendor, Currency)
```
(8.18)

可以观察到，一旦这个分解完成，第二个MVD不能再被应用，因为在上述两个模式中没有联结依赖成立[⊖]。这种情况非常类似于我们在BCNF分解算法中面临的问题：在这个过程中，也许会丢失一些依赖。在本例中，可能丢失的是依赖（8.17）。如果我们首先使用上面的第二个分解，会有同样的问题，但此时会丢失依赖（8.16）。

和在BCNF分解中丢失FD不同，丢失MVD是一个潜在的更严重的问题，因为这个结果可能还会有相当的冗余，即使分解中每个关系都是4NF!为说明这一点，考虑CONTRACTS模式的如下关系：

Buyer	Vendor	Product	Currency
B_1	V_1	P	C
B_2	V_2	P	C
B_1	V_2	P	C
B_2	V_1	P	C

容易检测出这个关系满足MVD(8.16)和（8.17）。例如，为验证（8.16），考虑对分解模式（8.18）进行投影。

Buyer	Vendor	Product
B_1	V_1	P
B_2	V_2	P
B_1	V_2	P
B_2	V_1	P

Buyer	Vendor	Currency
B_1	V_1	C
B_2	V_2	C
B_1	V_2	C
B_2	V_1	C

⊖ 这不是很明显，因为我们没有讨论验证这种情况的合适工具。然而，在这个例子中，我们声明，可以直接使用自然联结的定义进行验证。我们建议读者参考[Maier 1983] 这本优秀的参考书来了解这个技术。

联结（使用自然联结）这两个关系显然会产生CONTRACTS的原始关系。更进一步观察可以看到，上述关系仍包含相当大的冗余。例如，第一个关系两次说明产品P由供应商V_1和V_2供应。而且，该关系两次说明购买者B_1和B_2想要购买P。第一个关系似乎要求进一步分解为（Buyer, Vendor）和（Vendor, Product），第二个关系要求分解为（Buyer, Currency）和（Vendor, Currency）。但是，这两个要求都不能满足，因为没有一个分解是无损的（比如，Vendor不是第一个关系的健）。结果是，分解(8.18)会遇到同样的更新异常，即使每个关系都满足4NF! 更进一步，由于4NF包含BCNF，在有MVD的情况下，即使BCNF也不能保证完全消除冗余。

2. 一个词典的例子

下面一个例子考虑多语言词典关系，DICTIONARY(English, French, German)，它提供从一种语言到另一种的转换。正如所期望的，每个术语有到每种语言的一个翻译（也许多于一个），并且这些翻译彼此独立。使用MVD很容易得到这些约束。

```
English →→ French | German
French →→ English | German
German →→ English | French
```
(8.19)

和上面一样，问题是在4NF分解算法中应用任何一个MVD会丢失其他两个依赖，分解结果同样会有更新异常。

3. 一个多语言词典例子

让我们扩展前面的例子，以使一个术语只与一个概念关联，每个概念有一个关联的描述（由于本例的目的，忽略描述使用的语言）。相应的模式成为DICTIONARY（Concept, Description, English, French, German），其依赖是

```
English → Concept
French → Concept
German → Concept
Concept → Description
Concept →→ English | French | German
```
(8.20)

概念的一个例子是A5329，带有描述“homo sapiens”和翻译{human, man}, {homme}, {Mensch, Mann}。

4NF分解算法建议我们从挑选一个违背4NF的MVD开始，在分解过程中使用它。由于每个FD同样是MVD，我们可以先选择English→Concept，它会产生模式(English, Concept)和(English, French, German, Description)。使用MVD的传递性规则，我们可以推导出MVD English →→ French | German | Description，并进一步分解第二个关系为(English, French),(English, German),(English, Description)。

最终的模式有两个缺点。第一，它倾向于English，而原始模式是完全对称的。第二，双语关系中的每一个关系，比如（English, French），会冗余地列出所有可能翻译（从English到French，或从French到English）。例如，如果a和b是英语的同义词，c和d是法语的同义词，a翻译为c，那么英法词典（English, French）有所有四个元组：(a, c), (a, d), (b, c)和(b, d)。

使用4NF分解算法的一个更好的方法是，在考虑（8.20）中的函数依赖下，计算（8.20）中MVD的左边部分的属性闭包，根据增广规则得到如下MVD:

Concept Description ↠ English | French | German

我们可以使用这个MVD来应用4NF分解算法，这会产生分解(Concept, Description, English)(Concept, Description, French)(Concept, Description, German)。我们可以使用FD进一步分解这些关系为BCNF，产生（Concept, Description)。我们不但把它分解为4NF，还保持了所有依赖。

8.10.3 如何进行4NF分解

上述例子使我们明白，设计4NF并不是一个简单的过程。实际上，这个问题直到20世纪80年代的早期一直是一个活跃的研究领域[Beeri et al. 1978, Zaniolo and Melkanoff 1981, Sciore 1983]。最终，所有这些工作被合并为[Beeri and Kifer 1986b]中一个统一的框架。我们不会深入研究这个方法的细节，它的要点可以用我们的三个例子解释：合同、词典、辞典。

1. 分割左边的异常

当一个MVD拆分另一个MVD的左边部分时，表明会有设计问题，正如合同例子所示的那样。例如，MVD(8.17)划分（8.16）中左边的部分（Buyer, Vendor)，这表明Buyer和Vendor是不相关的属性（每个购买者与每个供应商的某个元组关联)，因此不能在同一个关系中。把CONTRACTS关系分解为（Buyer, Vendor, Product）和（Buyer, Vendor, Currency）会产生冗余，也是这个原因。

这个模式存在问题的原因也许是不正确地指定了依赖。不使用CONTRACT模式中的MVD，联结依赖

Buyer Product ⋈ Vendor Product
⋈ Vendor Currency ⋈ Buyer Currency

似乎更合适。它简单地说明每个购买者需要特定的产品，每个供应商供应特定的产品，一个供应商可以接受特定的货币，一个购买者可以支付特定的货币。只要一个购买者和一个供应商可以在产品和货币上匹配，一个交易就可以进行。这个英语的描述与合同例子描述中的需求相匹配，设计者也许没能认识到上述JD就是所需的。

2. 交异常

一个交异常是指一个模式有一对MVD，形如 $\overline{X} \twoheadrightarrow \overline{Z}$ 和 $\overline{Y} \twoheadrightarrow \overline{Z}$，但没有MVD $\overline{X} \cap \overline{Y} \twoheadrightarrow \overline{Z}$。注意，我们的词典例子就包含这种异常：有MVD English ↠ French和German ↠ French，但是没有依赖Ø ↠ French。[Beeri and Kifer 1986b]认为这是一个设计问题，可以通过创建new(!)属性来修正这个问题。在这里，属性Concept丢失了。事实上，我们的辞典例子就是基于词典例子，再添加这个特定的属性[⊖]和将它与老的属性关联起来的依赖。令人惊讶的是，这种类型的异常可以自动改正，即新的属性和关联的依赖可以用一个良好的算法来产生[Beeri and Kifer 1986a, 1987]。

3. 设计策略

假定拆分左边的部分不会产生异常[⊜]，模式**R**=($\overline{R}$; *D*)到第四范式的一个依赖保持分解可以利用以下五个步骤得到。

⊖ 另一个属性Description用来说明另一个问题，这里暂时先不考虑它。

⊜ 这样的异常意味着依赖是不正确或不完备的。

1) 使用被D蕴涵的FD，计算每个MVD $X \twoheadrightarrow Y$左边的属性闭包X^+。用$X^+ \twoheadrightarrow Y$替换$X \twoheadrightarrow Y$。

2) 找到MVD结果集的最小覆盖。如果D并不存在拆分左边的异常，这样一个覆盖是唯一的。

3) 通过加入新的属性，使用算法[Beeri and Kifer 1986a, 1987]消除交异常。

4) 只用MVD来应用4NF分解算法。

5) 只用FD在每个结果模式中应用BCNF设计算法。

在分解结果中每个关系都符合4NF，没有MVD丢失，这保证在结果模式中没有冗余。而且，如果最后一步的分解是依赖保持的，那么这5个步骤都是依赖保持的。

从词典例子到辞典例子的转变，以及辞典例子的最终分解是这五步过程的一个说明。为了简化例子，我们先应用步骤3，接着增加一个属性Description,以说明步骤1并使步骤5更令人感兴趣。由于我们的例子没有冗余的MVD，则没有必要进行步骤2。

8.11 范式分解的总结

我们总结一下讨论过的范式分解算法的一些性质。

- 第三范式模式也许会有一些冗余。我们讨论的产生3NF模式的算法是无损且依赖保持的。它不考虑多值依赖。
- 如果只考虑FD，那么*Boyce-Codd*分解没有冗余。我们讨论的产生BCNF模式的算法是无损的，但可能不是依赖保持的。（正如我们已看到的，一些模式并没有既是无损又依赖保持的Boyce-Codd分解。）它并不考虑多值依赖，因此这种依赖可能会造成冗余。
- 第四范式分解没有任何非平凡的多值依赖。我们讨论的产生4NF模式的算法是无损的，但可能不是依赖保持的。它会尝试消除与MVD关联的冗余，但它并不能确保所有这样的冗余都能消除。

注意，上述分解都不会产生拥有我们想要的所有属性的模式。

8.12 案例研究：学生注册系统的模式精化

我们花费了很大的精力来研究关系的规范化理论，我们现在可以用它来验证5.7节设计的学生注册系统。好消息是，把图5-14中的E-R图转换为图5-16和图5-17所示的关系这一步骤，我们完成得不错，因此大部分关系都满足Boyce-Codd范式。下面我们要开始新的工作。

为确定一个模式是否符合范式，我们应该收集所有相关的函数依赖。一个来源是PRIMARY KEY 和UNIQUE约束。然而，也许还有其他的依赖不能通过这些约束或E-R图所捕捉到。只能通过只能通过仔细地检查模式和应用程序的规格说明来发现这些依赖。如果没有找到新的依赖，模式中所有函数依赖都是主键和候选键（或者被它们蕴涵的函数依赖），那么这个模式是BCNF。如果发现其他的函数依赖，我们必须检测模式是否符合范式要求，如果不符合要求，就要进行适当的调整。

在我们的例子中，我们可以验证图5-16中所有关系模式（除了CLASS之外）都符合BCNF，因为它们没有函数依赖不被键所蕴涵。进行这个验证并不是很困难，因为这些模式有6个或更少的属性。困难的是验证CLASS，它有10个属性。

我们使用CLASS模式来说明这个过程。在这个过程中，我们发现一个丢失的函数依赖并规范化CLASS。首先，我们列出为那个关系在CREATE TABLE语句中指定的键约束。

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
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

验证那些大的模式中的额外依赖是很困难的，因为必须考虑CLASS属性的不是超键的每个子集，并检测它是否函数决定别的属性。这个“检测”并不是基于任何具体算法。严格地说，确定一个特定的FD是否在一个关系中成立，是设计者在用数据库建模真实世界时，对相应实体的语义的理解问题，本质上它是容易出错的。然而，现在已经开始相应的研究来解决这个问题。比如，FDEXPERT[Ram 1995]是一个专家系统，通过使用关于典型企业的知识和它们的设计模式，可以帮助数据库设计者发现FD。

但是，我们手边并没有这样一个专家系统，因此我们分析起来很费力。考虑如下的候选FD:

ClassTime ClassroomId InstructorId → CrsCode

容易看出为什么这个FD并不适用，因为不同的课程可以被同一个老师在同一个时间同一个教室教授，只要所有这些在不同学期和年份发生。利用类似方法，可以拒绝许多其他的FD。然而，正如5.7节所介绍的，我们假定对给定的课程，最多只有一种教材，如下的函数依赖是对先前指定的CLASS约束集一个合适的补充:

CsrCode Semester Year → Textbook　　(8.21)

尽管某门课程使用的教材每学期都可以不同，但如果一门课程在一个特定的学期开设，因为注册人数太多会被分为几个班，那么所有班都会使用同样的教材[⊖]。

现在容易看出CLASS的设计问题：上述依赖的左边部分不是键，Textbook不属于任何键。因此，CLASS不符合3NF。3NF合成算法建议，按如下方法划分原始模式可以纠正这个情况:

⊖ 当然，这个规则不一定适合所有的大学，但对大部分大学来说都为真。

- CLASS1，包含所有CLASS的属性（除了Textbook之外），以及FD 1,3~7,9~13,15~18（这些数字指前面提供的带编号的FD列表）。
- TEXTBOOKS（CrsCode, Semester, Year, TextBook），以及一个FD CrsCode Semester Year→Textbook。

这些结果得到的模式均符合BCNF，因为我们直接检查它们所有FD均被键约束蕴涵来验证。3NF合成算法同样保证上述分解是无损的，并且是依赖保持的。

让我们考虑一个更实际的情况，课程可以有多种推荐教材，在一个特定学期的课程的所有班级使用这些课本。在这种情况下，FD(8.21)并不成立。注意，任何特定班级使用的教材与上课时间、老师、注册等等无关。这种情况类似于8.9节的（8.14）所示的PERSON关系，其中，属性Textbook与属性ClassTime、InstructorId等等无关，上述情形可以形式地表示为如下的多值依赖:

```
(CrsCode Semester Year Textbook)
    ⋈ (CrsCode Semester Year ClassTime SectionNo
       InstructorId Enrollment MaxEnrollment ClassroomId)
```

或使用8.10节介绍的符号表示为:

```
CrsCode Semester Year →→ Textbook
```

和PERSON关系的模式一样，CLASS符合BCNF。尽管这样，由于上面提到的多值依赖，它包含冗余。解决这个问题的一个方法是使用更高层的范式4NF，幸运的是，使用8.10节的算法很容易。我们只是需要使用上述依赖分解CLASS，这会产生如下无损分解（8.10节的分解算法保证了无损性）:

- CLASS1(CrsCode, Semester, Year, ClassTime, SectionNo, InstructorId, Enrollment, MaxEnrollment, ClassroomId)和函数依赖1,3~7,9~13,15~18。
- TEXTBOOKS(CrsCode, Semester, Year, Textbook)没有FD。

注意，这个模式和先前在每个班级一种教材的假设下获得的模式的唯一区别是在第二个模式中缺少FD。

```
CrsCode Semester Year → Textbook
```

对5.7节开发的学生注册系统的初始设计应用关系规范化理论的结果是产生一个无损分解，且每个关系都符合4NF（因此也符合BCNF）。幸运的是，这个分解是依赖保持的，因为为模式指定的每个FD在分解中是嵌入在某个关系中的，这对带有4NF和BCNF的设计来说并不是总可以达到的。

8.13 性能调整问题：是否进行分解

在本章，我们学到了很多模式分解的理论。然而，这个理论的动机是在数据库频繁更新的情况下，避免冗余会导致的一致性维护问题。如果大部分事务是只读查询会怎么样？模式分解似乎使回答查询更困难，因为原本存在于一个关系中的关联也会被分为几个不同的关系。

例如，找到每个地址的平均爱好数，使用图5-2的单一关系比使用图8-1的一对关系更加有效，因为后者要求在计算聚合前先进行联结。这是经典的时空折中的例子，即增加冗余可以

改善查询性能。如果对一个特定应用的一个频繁执行的查询性能不是很好，就必须评估是否要进行这样的折中。在这种情况下，术语**反规范化**（denormalization）描述了这种情形：满足特定的范式会带来性能的下降，因此不应该进行分解。

通常，没有一种建议会适合所有情况。有时，模拟可以帮助我们解决问题。下面是一个有冲突指导的不完全列表，对每个特定事务的混合情况需要进行评估。

1) 分解通常会使回答复杂查询的效率更低，因为在评估查询时要执行额外的联结。

2) 分解使回答简单的查询更有效，因为这类查询通常涉及同一关系的很少几个属性。因为分解后的关系有更少的元组，所以在计算简单查询的过程中需要扫描的元组可能会更少。

3) 分解通常使简单的更新事务效率更高。然而，对复杂的更新事务也许并非如此（如给教计算机科学专业每门课程的所有教授涨工资），因为它们也许包含复杂的查询（因此也要求复杂的联结）。

4) 分解可以降低对存储空间的要求，因为它通常会消除冗余数据。

5) 如果冗余程度低的话，分解会增加存储需求。例如，在（8.14）的PERSON关系中，假定（有很少例外）大部分人只有一个电话号码和一个孩子。这时，模式分解通常会增加存储需求而不会带来看得到的好处。对图8-6的HASACCOUNT进行分解，会增加更新事务的开销。原因是对FD的验证

```
ClientId OfficeId → AccountNumber
```

在更新后要求进行联结，因为属性ClientId和OfficeId属于分解中不同的关系。

8.14 参考书目

关系范式和函数依赖是在[Codd 1970]引入的。Armstrong公理和它们的正确性和完整性证明首先出现在[Armstrong 1974]，更易理解的说明在[Ullman 1988, Maier 1983]。对FD蕴涵的一个有效算法首先出现在[Beeri and Bernstein 1979]中。对无损分解的一个通用测试方法首先出现在[Beeri et al. 1981]中。合成第三范式的的算法出现在[Bernstein 1976]中。

第四范式在[Fagin 1977]介绍，该书同时提出了一个朴素分解算法并探索了4NF和BCNF之间的关系。第五范式在[Beeri et al. 1977]中介绍，但我们推荐参考[Maier 1983]来系统地了解这个主题。[Kanellakis 1990]的综述同样是一个好的起点。关于4NF的其他论文有[Beeri et al. 1978, Zaniolo and Melkanoff 1981, Sciore 1983]。最后，在有FD和MVD时设计4NF模式的所有工作被扩展并集成在一个[Beeri and Kifer 1986a and 1986b, 1987]介绍的统一的框架中。关于4NF最新的著作有[Vincent and Srinivasan 1993, Vincent 1999]。

本书提供的深入介绍关系设计理论的教材有[Mannila and Raäihä 1992, Atzeni and Antonellis 1993]。

正如8.12节所介绍的，应用本章讨论的结果到数据库设计的最大的障碍是，找到在模式规范化过程中使用的正确依赖集。我们提到了FDEXPERT系统[Ram 1995]，它可以应用各种企业的知识来发现函数依赖。在使用机器学习和数据挖掘技术来发现FD、MVD和包含依赖的算法的领域，也进行了大量的研究工作[Huhtala et al. 1999, Kantola et al. 1992, Mannila and Raäihä 1994, Flach and Savnik 1999, Savnik and Flach 1993]。我们将在第19章介绍数据挖掘。

8.15 练习

8.1 函数依赖的定义并没有排除左边部分为空的情形，即它允许使用形如{}→A的FD。解释这种依赖的含义。

8.2 给出一个例子，说明一个模式不是3NF但只有两个属性。

8.3 一个关系键可以有的最小属性个数为多少？

8.4 一个表ABC有属性A,B,C和函数依赖$A \rightarrow BC$。写一个SQL CREATE ASSERTION语句以防止违背这个函数依赖。

8.5 证明只有两个属性的3NF关系模式都符合BCNF。证明最多只有一个非平凡FD的模式符合BCNF。

8.6 证明Armstrong传递性公理是正确的，即一个关系若满足FD $\overline{X} \rightarrow \overline{Y}$，$\overline{Y} \rightarrow \overline{Z}$，则它一定满足FD $\overline{X} \rightarrow \overline{Z}$。

8.7 证明如下的通用传递性规则：如果 $\overline{Z} \subseteq \overline{Y}$，那么 $\overline{X} \rightarrow \overline{Y}$，$\overline{Z} \rightarrow \overline{W}$ 蕴涵 $\overline{X} \rightarrow \overline{W}$。试着使用两种方法证明这条规则：

- 直接使用FD定义的方法，如8.4节所示。
- 试用Armstrong公理，通过一系列的步骤，从 $\overline{X} \rightarrow \overline{Y}$，$\overline{Z} \rightarrow \overline{W}$ 推导出 $\overline{X} \rightarrow \overline{W}$。

*8.8 我们已经证明了图8-3算法的的正确性，即如果$A \in closure$，则 $A \in \overline{X}_F^+$。证明这个算法的完整性。即如果 $A \in \overline{X}_F^+$，那么在计算结束时$A \in closure$。提示：利用Armstrong公理，对推导$X \rightarrow A$的长度进行归纳。

8.9 设$\mathbf{R}=(\overline{R}; F)$是一个关系模式，$\mathbf{R}_1=(\overline{R}_1; F_1)$，…，$\mathbf{R}_n=(\overline{R}_n; F_n)$是它的分解。$\mathbf{r}$是$\mathbf{R}$的一个有效关系实例，$\mathbf{r}_i = \pi_{\overline{R}_i}(r)$。证明$\mathbf{r}_i$满足FD的集合$F_i$，且是模式$R_i$的一个有效关系实例。

8.10 设 $\overline{R}_1$和 $\overline{R}_2$是属性的集合，$\overline{R} = \overline{R}_1 \cup \overline{R}_2$。$\mathbf{r}$是 $\overline{R}$ 的关系。证明 $\mathbf{r} \subseteq \pi_{\overline{R}_1}(\mathbf{r}) \bowtie \pi_{\overline{R}_2}(\mathbf{r})$。将这个结果推广到 $\overline{R}$ 分解为n>2的模式。

8.11 设$\mathbf{R}=(\overline{R}; F)$是一个模式，$\mathbf{R}_1=(\overline{R}_1; F_1)$，$\mathbf{R}_2=(\overline{R}_2; F_2)$是一个二元分解，$(\overline{R}_1 \cap \overline{R}_2) \rightarrow \overline{R}_1$或$(\overline{R}_1 \cap \overline{R}_2) \rightarrow \overline{R}_2$均不被$F$蕴涵。构造一个关系$\mathbf{r}$，使 $r \subset \pi_{\overline{R}_1}(\mathbf{r}) \bowtie \pi_{\overline{R}_2}(\mathbf{r})$，“$\subset$”表示严格子集。

8.12 设$\mathbf{R}_1$，…，$\mathbf{R}_n$是模式R的分解，它通过一系列二元无损分解得到（以$\mathbf{R}$的一个分解开始）。证明$\mathbf{R}_1$，…，$\mathbf{R}_n$是模式$\mathbf{R}$的无损分解。

8.13 证明图8-8的BCNF分解算法的循环有如下性质：每个下一次迭代的数据库模式比上一次迭代的模式有严格更少的违背BCNF的FD。

*8.14 证明8.8节的合成3NF分解的算法产生的模式满足3NF条件（提示：使用反证法。假定一些FD违背3NF，接着证明这与用最小覆盖合成的模式的算法相矛盾）。

8.15 考虑数据库模式，它的属性为A、B、C、D、E，函数依赖为$B \rightarrow E$，$E \rightarrow A$，$A \rightarrow D$，$D \rightarrow E$。证明这个模式分解为AB，BCD，ADE是无损的。它是依赖保持的吗？

8.16 考虑关系模式，它的属性为$ABCGWXYZ$，依赖集为$F=\{XZ \rightarrow ZYB$，$YA \rightarrow CG$，$C \rightarrow W$，$B \rightarrow G$，$XZ \rightarrow G\}$。使用合适的算法解决如下问题。

a. 找到F的最小覆盖。

b. 依赖$XZA \rightarrow YB$是否被F蕴涵？

c. 分解为$XZYAB$和$YABCGW$是无损的吗？

d. 上述分解是依赖保持的吗？

8.17 考虑属性集$ABCDEFGH$的如下函数依赖：

$A \rightarrow E$

$AD \rightarrow BE$

$AC \rightarrow E$

$E \rightarrow B$

$BG \rightarrow F$

$BE \rightarrow D$

$BDH \rightarrow E$

$F \rightarrow A$

$D \rightarrow H$

$CD \rightarrow A$

找到最小覆盖，接着分解为无损的3NF。完成后，检查结果关系是否符合BCNF。如果找到一个模式不是BCNF，把它分解为无损的BCNF。解释所有步骤。

8.18 找到如下依赖集在属性AFE上的投影:

$A \rightarrow BC$

$C \rightarrow FG$

$E \rightarrow HG$

$G \rightarrow A$

8.19 考虑模式，其属性集为$ABCDEFH$，函数依赖在（8.12）中描述。证明分解$(AD; A \rightarrow D)$，$(CE; C \rightarrow E)$，$(FA; F \rightarrow A)$，$(EF; E \rightarrow F)$，$(BHE; BH \rightarrow E)$不是无损的，通过对$ABCDEFH$一个具体的关系实例在这个模式上的投影来展现信息的丢失。

8.20 考虑模式$BCDFGH$有如下FD:$BG \rightarrow CD$，$G \rightarrow F$，$CD \rightarrow GH$，$C \rightarrow FG$，$F \rightarrow D$。使用3NF合成算法来获得3NF的一个无损的、依赖保持的分解。如果某个结果模式不是BCNF，继续分解它们为BCNF。

*8.21 证明8.10节所有FD和MVD的推理规则是正确的。换句话说，对每个关系**r**，满足对R的任何规则的假设中的依赖，R的结论同样被**r**满足（即，对于增广律，证明如果 $\overline{X} \twoheadrightarrow \overline{Y}$ 在**r**中成立，那么 $\overline{X}\ \overline{Z} \twoheadrightarrow \overline{Y}$ 同样在**r**中成立）。

8.22 设 $\overline{X}$、$\overline{Y}$、$\overline{S}$、$\overline{R}$ 是属性集合，有 $\overline{S} \subseteq \overline{R}$ 和 $\overline{X} \cup \overline{Y} = R$。设**r**是 $\overline{R}$ 的一个关系，满足非平凡MVD $\overline{R} = \overline{X} \bowtie \overline{Y}$（$\overline{X}$ 或 $\overline{Y}$ 均不是对方的子集）。

a. 证明：如果 $\overline{X} \cap \overline{Y} \subseteq \overline{S}$，那么关系 $\pi_{\overline{S}}(\mathbf{r})$ 满足MVD $\overline{S} = (\overline{S} \cap \overline{X}) \bowtie (\overline{S} \cap \overline{Y})$。

b. 设 $\overline{X}$、$\overline{Y}$、$\overline{S}$、$\overline{R}$ 满足上面所有条件（除了 $\overline{X} \cap \overline{Y} \not\subseteq \overline{S}$ 之外）。给出r的一个例子，满足 $\overline{R} = \overline{X} \bowtie \overline{Y}$，但不满足 $\overline{S} = (\overline{S} \cap \overline{Y}) \bowtie (\overline{S} \cap \overline{Y})$。

8.23 考虑关系模式，它有属性$ABCDEFG$和如下MVD:

$ABCD \bowtie DEFG$

$CD \bowtie ABCEFG$

$DFG \bowtie ABCDEG$

找到一个到4NF的无损分解。

8.24 对属性集$ABCDEFG$，MVD是

$ABCD \bowtie DEFG$

$ABCE \bowtie ABDFG$

$ABD \bowtie CDEFG$

找到一个到4NF的无损分解。它是唯一的吗？

8.25 考虑R的一个分解$\mathbf{R}_1,\cdots\mathbf{R}_n$，它是通过3NF合成算法的步骤1、2、3得到的（除了步骤4）。假定**R**中有一个属性A不属于任何$\mathbf{R}_i$, $i = 1,\cdots,n$。证明A必须是**R**的每个键的一部分。

8.26 考虑通过3NF合成得到的**R**的分解$\mathbf{R}_1,\cdots\mathbf{R}_n$。假定$\mathbf{R}_i$不符合BCNF，$X\rightarrow A$是$\mathbf{R}_i$的一个违规的FD。证明$\mathbf{R}_i$中一定有另一个FD：$Y\rightarrow B$，如果$\mathbf{R}_i$根据$X\rightarrow A$进一步分解的话，它将会丢失。

*8.27 这个练习依赖于8.10节介绍的技术。考虑关系模式，它的属性为$ABCDEFGHI$，并有如下MVD和FD:

$D\rightarrow AH$

$G\rightarrow I$

$D\twoheadrightarrow BC$

$C\twoheadrightarrow B$

$G\twoheadrightarrow ABCE$

找到一个到4NF的无损且依赖保持的分解。

第9章 触发器和动态数据库

第4章讨论的是如何在数据库反应性约束情况下使用触发器。除此之外，触发器还有其他的用途，例如在需要**动态数据库**（active database）的应用中会经常看到触发器，这些应用必须响应各种外部事件。在这些应用中，对可能出现的外部事件只知道总体情况，却不知道它们的确切时间安排，触发器特别适合这种情形。

虽然触发器不是SQL-92标准的一部分，但是许多数据库厂商都会在各自的产品中包括某种形式的触发器。本章所讨论的触发器是SQL:1999标准的一部分，关于SQL:1999中触发器的更多信息可以参考[Gulutzan and Pelzer 1999]。

9.1 触发器处理的语义

触发器（trigger）是数据库模式的元素之一，其结构如下所示：

ON *event* IF *precondition* THEN *action*

其中*event*是执行某个数据库操作的请求（例如在一张学生课程注册表中插入一行）；*precondition*是一个判定结果为真或假的表达式（例如某个班的学生人数已满）；*action*是描述当事件发生并且前提条件为真时的行为的语句（例如从数据库删除一些内容或者向管理员发送e-mail）。因为触发器是在以上三个因素的基础上构建的，所以有时也被称作ECA（Event-Condition-Action）规则。

令人惊奇的是，在这个简单的概念后面隐藏着许多复杂的问题。首先可能有不同类型的触发器，而每种类型的触发器可以用在不同的应用中。其次，怎样以及什么时候使用触发器存在许多细微的差别。第三，在某个时间点可能会同时激活若干个触发器，DBMS如何决定执行哪一个触发器并按照什么样的顺序执行？不同的选择会导致不同的执行结果。

最后，执行触发器可能会激活其他的触发器，因此一个事件会引起一连串触发器动作，并且不能确保这个过程什么时候会停止。实际上，每个DBMS会限制链式反应的深度。例如，如果深度超过32，那么抛出异常停止链式反应，同时回退先前的更新语句和触发器所作的修改。发生链式反应通常说明设计有问题，应当避免依赖于DBMS所施加的限制。在本章的后面将会讨论一些防止链式反应的方法。

1. 触发器判断

请求触发事件时会**激活**（activated）触发器，之后需要检查前提条件，触发器要判断什么时候检查这些条件。假设在请求触发事件的时候判断前提条件为真，从而执行触发器。但是不久之后，由于自身或者其他事务进行更新操作会使前提条件变为假。因此如果没有立即检查前提条件，那么就有可能不会执行触发器。

下面这个触发器的目的是保证注册的学生人数不会超过课程的名额限制：

ON *inserting a row in course registration table*
IF *over course capacity*
THEN *abort registration transaction*

当学生试图在课程注册表中添加自己名字的时候，课程名额可能已满。所以如果前提条件在试图注册的时候检查，那么会拒绝这个学生的请求。但是，与此同时可能会有另外一个学生执行放弃课程的事务（或者注册人员增加课程的学生名额），并且第二个事务是在第一个事务之前提交。因此如果触发器前提条件是在提交注册事务的时候检查，而不是在事件发生的时候检查，那么学生将会成功地注册这门课程。那么，在本例中延迟判断触发器前提条件的策略是恰当的。

但是，如果数据库正在监视核电厂，触发事件是压力增加，前提条件为压力不能超过某个临界值，那么最好立即判断触发器的前提条件。

总的来说至少有两种实用的策略：当请求触发事件的时候**立即**判断，或者**延迟**到提交事务的时候才判断。

事实上，触发器判断还有一些细节需要考虑。假设触发器T是由事件e激活，涉及关系R，C是与T有关的条件。在许多系统中（包括SQL:1999），C可以考虑在e发生之前和之后R的状态。因此，如果C仅使用R的两个状态而不涉及数据库中的其他关系，那么立即和延迟判断触发器T的结果是相同的。如果C仅考虑之前R的状态，那么这可以解释为在e发生之前判断C。但是如果C涉及数据库关系而不是R，那么两种判断模式的结果可能会不相同。

2. 触发器执行

如果触发器判断被推迟，那么也必须将触发器执行推迟到触发事务的结束为止。如果是立即判断，那么至少有两个选择，一是在判断后立即执行触发器，二是将执行推迟到触发事务结束后。对于原子能反应堆，立即执行是最佳选择，但是在一些不太重要的情况下推迟执行是比较好的选择。

在立即执行模式下有三种可能，既可以在触发事件之前或者之后执行，也可以在忽略触发事件的情况下执行。初看起来，有两种情况似乎很奇怪，怎么可能在事件之前或者忽略的情况下执行动作？原因在于事件是一个由事务向DBMS发出的请求，因此很有可能DBMS忽略请求而执行触发器，或者系统先执行触发器然后执行请求的动作。

3. 触发器粒度

这里是指事件的组成。**行级粒度**（row-level granularity）将对一行的修改看作一个事件，对不同行的修改被视为不同的事件，会引起触发器的多次执行。相反，**语句级粒度**（statement-level granularity）将事件看作一些语句，例如INSERT、DELETE和UPDATE，而不是每行的修改。因此一个没有进行修改的UPDATE语句（因为WHERE子句的条件没有影响数据库中的记录）也是引起触发器执行的事件。

在行级粒度上，触发器需要知道受影响的记录的原值和新值，这样才能正确判断前提条件。例如增加工资，原来的记录包含原来的薪水，新的记录包含新的薪水。如果两者都可以得到，那么触发器可以确认增长幅度没有超过10%并正确地执行相应的动作。行级粒度的触发器可以通过特殊的变量访问受到影响的元组的原值和新值。

在语句级粒度上，更新放在诸如原表和新表这样的临时结构里。这样允许触发器查询这

两张表并在结果的基础上执行操作。

4. 触发器冲突

一个事件可能会一次激活若干个触发器。例如，当学生注册课程的时候将触发下面这两个触发器：

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*

这时首先考虑哪一个触发器呢？有两种方案：

- 顺序原则 即依次判断触发器的前提条件。如果条件为真，那么就执行相应的触发器。当执行完成以后，再考虑下一个触发器。在这个例子中，学生可以选择接受另一门课程而放弃请求已满员的课程。这样在判断第二个触发器的时候，因为前提条件不再为真，因此不会执行。
- 组原则 也就是一次判定触发器的所有前提条件，然后安排执行前提条件为真的触发器。在本例中会执行（依次或者同时）所有的触发器，即使某些触发器的前提条件在判定后不久变为假。

对第一种方案，系统可以决定顺序选择或者随机选择触发器。第二种方案中的顺序无关紧要，因为可以安排同时运行所有的触发器（虽然大部分DBMS仍会排序）。

5. 触发器和完整性约束

第4章讨论过更新数据库可能会违反参照完整性约束。SQL可以指定修正动作（例如ON DELETE CASCADE），使DBMS借此恢复完整性。这些动作可以看作是特殊的具有严格语义的触发器：在执行结束时，必须恢复数据库的完整性。如果修正动作激活了其他会引发违反参照完整性的触发器，那么情况会变得比较复杂。这时必须以最终恢复完整性约束为目标安排所有的触发器。目前还没有一个明确的方案可以解决触发器安排问题。本章后面的部分将讨论SQL:1999是如何解决这个问题的。

9.2 SQL:1999中的触发器

制定SQL:1999当前的触发器语法和语义标准经历了一个相当漫长和痛苦的过程。首先，各个数据库厂商在自己的系统中都已配备了触发器，因此标准要能带来巨大的利益才能说服厂商改变他们的实现。其次，正如已经看到的，与触发器相关的语义问题并非无足轻重，除非为先前提到的问题提供良好的处理方案，否则不会接受这个标准。

对与触发器处理有关的问题有了全新的认识之后，现在开始系统地探讨SQL:1999。

- 触发事件 事件可以是将SQL INSERT、DELETE和UPDATE语句作为一个整体执行，或者是利用这些语句对单个记录进行修改。
- 触发器前提条件 SQL的WHERE子句中允许使用的任何条件。
- 触发器动作 可以是SQL查询，DELETE、INSERT、UPDATE、ROLLBACK以及

SIGNAL语句，或者是使用SQL/PSM（SQL's Persistent Stored Module，SQL永久存储模块）语言编写的程序。SQL/PSM使过程控制语句和SQL查询与更新语句无缝地集成起来。有关内容将在下一章讨论。

- *触发器冲突解决*　按照顺序原则，SQL:1999将所有的触发器进行排序，然后按照某个特定实现方式执行。因为顺序可能因数据库产品而异，所以必须使数据库产品与触发器的顺序无关。
- *触发器判断*　立即判断模式，可以指定在触发事件之前还是之后执行。
- *触发器粒度*　支持行级和语句级两种模式。

下面是SQL:1999中触发器的完整语法。方框号里的结构是可选的；而花括号包含若干个由竖线分隔的选项，需要指定其中一个选项：

```
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
```

SQL:1999触发器的语法与前面讨论的模型大致相同。触发器有一个名称；它由某个事件激活（INSERT-DELETE-UPDATE子句指定）；可以定义为BEFORE类型或者AFTER类型（表明是在事件之前还是在事件之后检查前提条件）；WHEN子句指定前提条件。子句FOR EACH ROW和FOR EACH STATEMENT指定触发器的粒度。如果指定前者，那么该触发器监视的表中元组的修改会激活触发器；如果指定后者（默认设置），那么在对被监视的表每次执行INSERT、DELETE或者UPDATE语句的时候都会激活触发器，不管这个执行会修改多少元组（即使没有修改也会激活触发器）。

WHEN子句后的*statement-list*定义触发器触发时执行的动作，通常这些动作是SQL语句（插入、删除或者更新记录），但是更一般的形式是SQL/PSM语句，它将SQL语句和if-then-else、循环和局部变量等融合在一起。有关SQL/PSM的内容在下一章将讨论。

REFERENCING子句引用关系*table name*更新前和更新后的内容，可以用在WHEN条件和紧随其后的语句列表中。

有两种类型的引用，这与触发器的粒度有关。如果触发器是行级的，那么可以使用OLD AS和NEW AS子句定义元组变量来引用激活触发器元组的原值与新值。如果事件是INSERT，那么不能用OLD；如果事件是DELETE，那么不能用NEW。如果触发器粒度是语句级的，那么SQL:1999提供一些方法访问触发语句所影响的表的原值和新值。OLD TABLE命名一张含有更新所影响的元组的原值的表，而NEW TABLE命名一张可以访问这些元组的新值的表。

注意　NEW AS和OLD AS定义的元组变量仅仅涉及更新所影响的行。也就是说，在插入元组的时候，NEW AS指向插入的元组；在更新的时候它指向被更新元组的新状

态。OLD AS指向被删除的元组或者被更新元组的原状态。

同样，OLD TABLE和NEW TABLE仅仅涉及更新所影响的行，而不是整张表的原状态和新状态。如果r是表更新前的状态，那么更新后的状态是（r-旧表）U新表。

最后，SQL:1999对BEFORE和AFTER触发器行为可以施加某些约束。

1. BEFORE触发器

所有BEFORE类型的触发器完全在触发事件发生前执行。它们不能更新数据库，但是可以检查WHEN子句中的前提条件并选择接受或者中止触发事务。因为BEFORE触发器不能更新数据库，因此也不能激活其他触发器。

BEFORE触发器的一个典型用途是保持与应用相关的数据完整性。例如，除了已经熟悉的TRANSCRIPT表外，另外一张表CRSLIMITS具有字段CrsCode、Semester和Limit（从名称可以看出每个字段的含义）。在下面的示例中，触发器对TRANSCRIPT的插入操作进行监视。INSERT语句可以插入不同课程的多条记录。触发器指定行级粒度，因此每个插入被视为单独的事件。触发器检查插入所影响的课程没有超过名额限制。如果超过，那么拒绝插入。

```
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
```

注意，WHEN子句中的第一个SQL语句同时引用TRANSCRIPT表的新行（通过行变量N）和所有行（通过变量T）。在检查WHEN条件的有效性时TRANSCRIPT表的状态是怎样的？对于N所引用的行，答案很清楚：必须是所引用行的新状态。但对于T所引用的状态，则并非那么显而易见，答案是，BEFORE触发器认为所引用的表处于原状态，而AFTER触发器则认为该表处于新状态。

2. AFTER触发器

AFTER触发器完全在触发事件对数据库进行修改以后执行。由于允许它们修改数据库，因此能激活其他触发器（这会引起前面所说的链式反应）。AFTER触发器可以作为应用逻辑的延伸，可以自动处理各种事件，从而减轻应用程序员在每个应用中编写处理这些事件的代码的负担。

下面的示例说明如何使用AFTER触发器。首先来看一下怎样执行约束：事务所执行的薪水增幅不得超过5%。假定数据库有一张具有属性Salary的表EMPLOYEE:

```
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
```

当更新EMPLOYEE表中某行的Salary的时候，触发器通知DBMS比较原值和新值。如果增幅超出所设置的界限，那么调整薪水增幅为5%。如果增幅没有超出所设置的界限或者薪水下降，那么不会执行触发器。注意，行变量O和N始终引用同一行，即数据库更新所影响的那一行。不同之处在于O指向行的原状态而N指向行的新状态。

注意，当触发器LIMITSALARYRAISE被触发，并执行动作的时候，该动作会覆盖原来事件（触发LIMITSALARYRAISE）的影响。另外，动作本身是一个触发事件，但是这个新事件不会导致链式反应：当第二次检查触发器的时候，薪水实际上下降了（正好比原来的薪水增加5%），因此WHEN的条件为假，触发器不会再执行。

目前所举的示例中的触发器的粒度都是行级的，但是有一些应用要求在完成所有的更新后触发器只执行一次。下面举两个示例说明如何使用语句级的触发器。

假设在全体增加薪水之后想记录所有员工新的平均薪水，可以使用触发器实现这个功能：

```
CREATE TRIGGER  RECORDNEWAVERAGE
    AFTER UPDATE OF  Salary ON EMPLOYEE
    FOR EACH STATEMENT
        INSERT INTO LOG
        VALUES (CURRENT_DATE,
                (SELECT AVG(Salary) FROM EMPLOYEE))
```

当触发器执行的时候，会在Log表中插入一行记录新的平均薪水。这条记录同时含有计算平均薪水的日期（CURRENT_DATE是返回当前日期的一个内嵌函数）。因为每修改一行记录就计算一次平均值没有任何意义，所以使用语句级粒度比行级粒度更适合。

第二个示例说明语句级的触发器如何维护包含依赖（inclusion dependency）。在第4章里介绍过利用包含依赖对外键约束进行泛化，在实际中会经常使用这样的设置。考虑下面的依赖：*教师去教授没有学生注册的课程*（见(4.1)）。这是一个不基于外键的参照完整性约束，SQL-92要求使用断言来表示它（见(4.4)），那么AFTER触发器是如何维护更新时的约束呢[⊖]？

关键的一点是构建SQL视图IDLETEACHING，该视图包含TEACHING表中那些在TRANSCRIPT表中没有相应记录的行。换句话说，该视图包含那些违反包含依赖的行。定义这个视图的工作留给大家作为练习。

触发器的工作过程如下所示：在学生放弃某门课程（某些课程）后，触发器从TEACHING表中删除所有能在IDLETEACHING发现的行。

```
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)
```

(9.1)

同样，我们可以构建触发器以便在插入记录到TEACHING表中时维护包含依赖。如果在TRANSCRIPT表中没有相应的记录，那么触发器将中止试图插入记录到TEACHING表中的事务。

⊖ 可以使用同样的方法在与SQL-92不匹配的系统中仿效外键约束里的ON DELETE和ON UPDATE子句。

注意，上面的触发器也许会引起类似于鸡生蛋还是蛋生鸡的问题。直到某个学生注册一门课程之后才能给这门课程安排某个教师。但是大部分学校下学期的课程表在学生注册之前就已经公布了，因此一些触发器需要创建并销毁。例如上面的触发器只需要在注册截止时间和课程增删截止时间之间有效即可。

3. 触发器评价过程概述

假设事件e在执行语句S的时候发生[⊖]，这个事件激活一组触发器$\mathbf{T} = \{T_1, \cdots, T_k\}$。下面是触发器处理的过程:

1) 将新激活的触发器放入触发器队列**Q**中。

2) 暂停执行S。

3) 如果使用行级粒度，则计算OLD和NEW；如果使用语句级粒度，则计算OLD TABLE和NEW TABLE。

4) 判断**T**中所有BEFORE触发器，执行前提条件为真的触发器，而将前提条件为真的AFTER触发器放入**Q**中。

5) 根据S更新数据库。

6) 按照优先权（依赖于实现）判断**Q**中的每个AFTER触发器，如果触发条件为真则立即执行。如果触发器执行激活了新的触发器，那么跳至步骤1重新执行算法。

7) 恢复语句S的执行。

4. 触发器和外键约束

当激活触发器的事件是在一个具有ON DELETE和ON UPDATE外键约束的关系上发生的时候，这些子句指定的修正动作可能会引起自身的更新，从而使系统的准确语义变得相当复杂。实际上，SQL:1999标准的设计者需要反复多次进行迭代以寻求满意的解决方案。

也许有人会问为什么不能将外键约束看作通常的触发器呢？实际上在第4章我们就是这样称呼它们的。答案是，这些约束是触发器，但是它们具有特殊的语义（它们是为了修正违反外键约束的状态），在触发器评价过程中需要包括这个语义。为了达到这个目的，我们修改步骤5:

5') 根据S更新数据库，对每一个当前状态所违反的外键约束执行下列操作:

a) 将相关的修正动作表示为**act**（也就是CASCADE、SET DEFAULT、SET NULL或者NO ACTION）。注意，**act**是一个事件，可能会激活其他的触发器$\mathbf{S} = \{S_1, \cdots, S_n\}$。

b) 判断**S**中的所有触发器，执行前提条件为真的BEFORE触发器，将前提条件为真的AFTER触发器放入**Q**中。

c) 根据**act**更新数据库。

注意，第5'步中的c并没有指明将**S**中未处理的触发器放在**Q**的队首还是队尾，因为SQL是根据优先级处理触发器的。

在步骤6中执行AFTER触发器可能会激活其他的触发器，也可能会违反外键约束，这时候循环执行步骤1 ~ 6。

⊖ 回忆一下，事件是一个执行数据库操作的请求。

5. 示例

上面的算法用来处理触发器和外键约束之间非常复杂的交互，这个交互也许涉及几十个触发器。下面所给的示例是比较简单的，只涉及两个触发器和一个外键约束。

假设TRANSCRIPT表中的课程也必须在COURSE表中列出，这两张表已在图4-4中描述过。这两个表之间的外键约束CHECKCOURSEVALIDITY可以表示如下：

```
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 )
```

如果COURSE的记录被删除或者更新，那么CHECKCOURSEVALIDITY会删除或者更新所有对应的TRANSCRIPT的记录。

假设有一个AFTER触发器WATCHCOURSEHISTORY记录COURSE表中元组的变化。如何使用SQL定义这个触发器留作练习。最后，触发器MAINTAINCOURSESNONEMPTY(9.1)也是数据库模式的一部分。注意，这个示例没有考虑其他外键约束（特别是与TEACHING相关的外键约束，因为包含这样的约束会使情况变得相当复杂）。

现在假设COURSE表中的一些课程发生变化，例如2000秋季开设的CS305变成CS405。这个变化会激活触发器WATCHCOURSEHISTORY和外键约束CHECKCOURSEVALIDITY。因为WATCHCOURSEHISTORY是AFTER触发器，所以它被放入Q。触发器处理算法首先按照步骤5'中c处理外键约束，因此TRANSCRIPT表中所有CS305的记录变成CS405。

这个修改激活了第二个触发器MAINTAINCOURSESNONEMPTY。因为CS305已经变为CS405，这样教授CS305的教师无课可上。换句话说，CS305在TEACHING表中，但是TRANSCRIPT表中没有相应的行，因此MAINTAINCOURSESNONEMPTY的WHEN条件为真，触发器被执行。

因为MAINTAINCOURSESNONEMPTY是一个AFTER触发器，所以被放在Q中（步骤5'的b），Q中已经包含WATCHCOURSEHISTORY。两个触发器执行的顺序跟所用的DBMS有关，因而无法预测[⊖]。当所有的触发器最终执行完毕的时候，课程的变化被记录在历史日志中而CS305的课程安排也被删除。

注意，以上两个触发器和外键约束的交互也许不能产生预想的结果。例如将CS305的课程安排变成CS405的安排也许更合理些。

9.3 避免链式反应

在触发器执行中出现永远没有结束的链式反应是一个十分严重的问题。在任何情况下触发都会最终停止的触发器系统称为是**安全的**（safe）系统。

⊖ 大部分DBMS使用调度策略基于反映触发器判断时间的时间戳。在本例中，时间戳顺序应该先是WATCHCOURSEHISTORY。

但是，没有算法能够告诉你给定的一组触发器是否是安全的。但是存在一些充分条件能够保证安全性（也就是说如果条件满足，那么触发器是安全的），但是它们不是必要的（即有一组触发器是安全的，但是它们不满足条件）。由于没有算法来测试安全性，所以也没有算法可以检验的安全性充要条件。

有鉴于此，在这里给出一个保证安全性的充分条件，但是它不能判断许多具有安全性的触发器系统。

在**触发图**（triggering graph）中，节点代表触发器（或者外键约束）。弧线从一个触发器T出发到另一个触发器T'终止，表示当且仅当T执行时会激活T'。

可以很容易地看出，使用简单的语法分析就能判断一个触发器是否会激活另一个触发器。实际上，激活触发器的事件在触发器定义的BEFORE/AFTER子句中列出（或者在外键约束定义的ON DELETE/UPDATE子句中列出）。触发器引发的事件可以从触发器主体中的语句确定。图9-1是本章讨论的部分触发器的触发图。

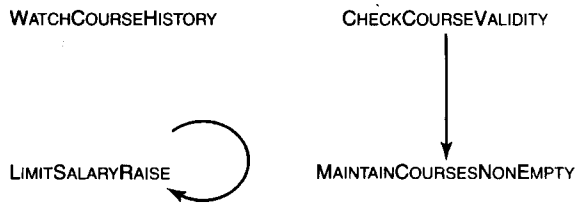


图9-1 触发器图

显然，如果触发器图是非循环的，那么就不可能以一种非终止的方式互相调用。根据这个准则，触发器WATCHCOURSEHISTORY、MAINTAINCOURSESNONEMPTY和CHECKCOURSEVALIDITY不可能产生链式反应。

虽然这个方法能够判断一些系统的安全性，但是在许多情况下它无能为力。实际上，触发图中LIMITSALARYRAISE部分是循环的，因为语法分析表明执行会激活自身。但是正如前面所说的，触发器不会执行第二次，因为WHEN条件为假。这个例子暴露出触发图的一个主要缺点：它没有考虑触发器的触发条件的语义。例如，如果可以保证在触发图中没有可以无限次运行的循环，那么这个触发器系统就是安全的。

还有许多改进触发图的方法，但由于这些内容超出了本书的范围，我们就不在这里讨论了。

9.4 参考书目

有关数据库触发器的基本概念可以参考[Paton et al. 1993]。集成触发器和外键约束的算法源自[Cochrane et al. 1996]。SQL:1999触发器的语法可以参考最近的SQL指导，例如[Gulutzan and Pelzer 1999]。

关于动态数据库的参考资料很多，本章中关于SQL:1999触发器的介绍只是冰山一角。感兴趣的读者可以参阅[Widom and Ceri 1996]进行更深入的学习。

9.5 练习

9.1 说明你的本地DBMS上应用的触发器的语义，描述定义触发器的语法。

9.2 给出从SQL触发器和外键约束的集合构建触发图的准确的语法规则。

9.3 补充MAINTAINCOURSESNONEMPTY（见9.1）的触发器设计，以便在TRANSCRIPT表中没有相应的记录

时禁止向TEACHING表中插入元组。

9.4 设计一个与MAINTAINCOURSESNONEMPTY触发器功能相同的行级触发器。

9.5 定义触发器WATCHCOURSEHISTORY，它使用LOG表记录事务对COURSE表中各门课程的所有修改。

9.6 定义一个触发器，在学生放弃某门课程、改变专业或者平均成绩低于某个阈值的时候执行（为简单起见，假设有函数grade_avg()，它以学生Id为参数，返回这名学生的平均成绩）。

9.7 考虑STUDENT (Id , Major)和PERSON(Id, Name)之间的IsA联系。写出合适的触发器来维护这种联系：当从PERSON删除一行的时候，必须从STUDENT表中删除具有相同Id的行；当插入一行到表STUDENT的时候，检查是否有相应的行存在于PERSON表中，如果没有则中止（不要使用FOREIGN KEY语句的ON DELETE和ON INSERT子句）。

9.8 证券公司的数据库包括表HOLDINGS(AccountId, StockSymbol, Price, Quantity)和表BALANCE(AccountId, Balance)。编写触发器维护买卖股票时账户余额的正确性，买股票时会插入一行到HOLDINGS表中或者增加Quantity，而卖股票时会从HOLDINGS表中删除一行或者减少Quantity。

9.9 在一个企业中，各个的项目使用的零件是由不同厂商供应的。定义一组合适的表和相应的外键约束，并定义在下面情况下执行的触发器：当项目改变某零件的供应商的时候；当供应商停止供应某零件的时候；当项目停止使用某零件的时候。

9.10 触发器可以立即判断并延迟执行，那么在判断和执行的时候OLD AS和NEW AS引用的是什么？

9.11 举例说明SQL:1999触发器不仅仅用来维护完整性约束。

第10章 真实世界中的SQL

在前面的章节中，我们将SQL作为一种交互式语言来讨论。每输入完一条查询语句，就开始焦急地等待，接着看到屏幕上出现结果（有时会因屏幕翻滚太快而看不清楚）。这种执行模式称为**直接执行**（direct execution），是最初的SQL的一部分。

在大部分事务处理应用中，SQL语句被集成到使用常用的编程语言（如C、Cobol、Java或者Visual Basic）编写的应用程序中，运行这些程序的计算机和数据库服务器所在的计算机不是同一台计算机。在本章中，我们将讨论SQL的一些高级特征，主要解决SQL在这种模式下运行时所涉及的问题。本章的目的在于介绍一些基本概念，因此不会详细讲述语法选项。

10.1 在应用程序中执行SQL语句

我们感兴趣的是生成一个包含SQL语句和通常的编程语言语句的程序，其中SQL语句能使程序访问数据库，而通常的编程语言（也叫做**宿主语言**(host language)）提供一些SQL没有的特征。这些特征包括诸如if语句和while语句等控制机制、赋值语句以及错误处理等。

当讨论如何在宿主语言中使用SQL语句时，一定会遇到下面两个问题:

- 在执行SQL语句之前，会执行一个**预备**（preparation）步骤。预备步骤首先分析语句，然后生成一个**查询执行计划**（query execution plan），这个计划决定执行语句所需步骤的顺序。例如，表按照什么样的顺序进行联结？是否需要首先为表进行排序？使用什么样的索引？要检查哪些约束？在第11章中，我们将介绍**访问路径**（access path）的概念，它决定如何访问表中各行。查询执行计划为查询中的每一张表建立一个访问路径。因为执行一个SQL语句需要大量的计算以及I/O资源，因此需要仔细设计。查询执行计划由DBMS使用数据库模式和语句结构设计，语句结构包括语句类型（例如SELECT和INSERT）、所要访问的表和列，还有列的域。表中的行数等因素在进行查询优化的时候也要加以考虑。SQL语句按照计划概述的步骤顺序执行，所涉及到的问题将在第14章中详细讨论。
- 有两种方式可以在应用程序中使用SQL结构:

语句级接口（Statement-Level Interface，SLI） SQL结构在程序中以新的语句类型的方式出现，这样程序不仅包含宿主语言而且包含新的语句类型。在使用宿主语言编译器编译程序之前，必须先用**预编译器**（precomplier）处理SQL结构，它将SQL结构转换成对宿主语言过程的调用。经过预编译之后的程序才能使用宿主语言编译器编译。运行时，这些过程与DBMS进行通信，DBMS会采取必要的动作执行SQL。

SQL结构有两种形式。第一种是**嵌入式SQL**（embedded SQL），也就是通常所说的SQL语句（例如SELECT和INSERT）；第二种是预备和执行SQL语句的指令，但是SQL语句是以字符串变量的值的形式出现在程序中的，而字符串变量是在运行时由程序的宿主语言部分构造

的。在第二种情况下，真正要执行的SQL语句在编译的时候并不知道，所以这种形式也被称为**动态SQL**（dynamic SQL）。而与此对应的嵌入式SQL在编译的时候已经知道并且被直接写进程序中，因此嵌入式SQL也被称作**静态SQL**（static SQL）。SQL-92是嵌入式SQL的一个标准，而SQLJ是一种专门为Java设计的SLI。

调用级接口（Call-Level Interface，CLI）与静态和动态SQL不同的是，应用程序完全是用宿主语言编写的。但是和动态SQL一样，SQL语句是在运行时生成的字符串变量的值，这些变量作为参数传递给CLI提供的宿主语言过程。因为没有使用特殊的语法，所以不需要经过预编译。

在本章中，我们会讨论两种调用级接口：Java数据库互连（Java DataBase Connectivity，JDBC）和开放式数据库互连（Open DataBase Connectivity，ODBC）。前者是专门为Java语言设计的，而后者被多种语言所支持。最近完成的SQL:1999对调用级接口进行了标准化，该标准与ODBC十分类似，所以有人预测两者将会最终合并。

10.2　嵌入式SQL

嵌入式SQL是一种语句级接口，可以将SQL语句嵌入到宿主语言程序中。程序要访问的数据库模式在编写程序的时候必须要知道，这样才能创建SQL语句。例如，程序员必须知道表的名称和列的名称以及域。

在宿主语言编译器编译程序之前，会有一个预编译器（通常由DBMS厂商提供）扫描应用程序并且确定嵌入式SQL语句的位置。因为这些语句不是宿主语言的一部分，所以宿主语言编译器不会处理它们。相反，通过某种特殊的语法将嵌入式SQL语句与其他部分分开，使得预编译器能够识别它们。预编译器将每条语句转换成一连串用宿主语言编写的对运行时链接库（run-time library）的子程序的调用，这样才能被宿主语言编译器在下个阶段进行处理。当程序运行并执行SQL语句的时候，会调用这些子程序，它们将原先嵌入在应用程序中的SQL语句发送给DBMS，由DBMS负责预备和执行这些SQL语句。

由预编译器检查每个SQL语句的形式并预备查询计划是合理的，因为这将会大大降低运行时的负载。不过，很多预编译器并不是这样做的。预备要求预编译器与DBMS通信（确定SQL语句所要访问的数据库模式），而这种通信在编译的时候不一定可行。另外，因为查询执行计划可能取决于表的规模，所以这种预备越在接近运行时进行效果越好。

在最好的情况下，嵌入式SQL结构使用SQL-92或者SQL:1999编写，然后DBMS的预编译器执行必要的转换，将它转换成该DBMS可以识别的SQL专用语言。但是在现实世界中，大部分编译器不作这样的转换，SQL结构必须使用DBMS的SQL专用语言。因而在实际中，必须在编写程序的时候知道DBMS和数据库模式。

要求应用程序使用DBMS的专用语言有一个缺点，就是如果将来需要转换到另外一种使用不同专用语言的DBMS的时候会带来一些问题。不过在某些情况下，使用DBMS的专用语言有一个优点，许多DBMS对SQL进行专有的扩展，如果嵌入到宿主语言中的SQL是SQL-92，那么程序员就不能使用这些扩展功能。当然，如果应用程序使用由DBMS提供的这种专门的扩展，那么就会增加转换到不同DBMS的难度。

SQL-92标准要求嵌入式SQL的实现至少要为以下7种语言提供预编译器：Ada、C、COBOL、Fortran、M（MUMPS）⊖、Pascal和PL/1。事实上，同样可以获得其他语言专用的预编译器。一些应用生成器提供自己专用的宿主语言和嵌入式SQL专用语言，并且提供一个编译器（或者解释器）使SQL结构能够访问某些DBMS。

图10-1是一个包含嵌入式SQL语句的C语言程序片段。每个嵌入式SQL语句由EXEC SQL开头，这样才能被预编译器定位。这里使用的是SQL-92语法，但是我们要知道许多数据库厂商使用它们自己的SQL专用语言（或者早期的SQL标准提供的语法）。

```
EXEC SQL BEGIN DECLARE SECTION;
         unsigned long num_enrolled;
         char *crs_code, *semester;
         ...
EXEC SQL END DECLARE SECTION;

    ... other host language declarations and statements ...
    ... get the values for 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 ...
```

图10-1　用C语言编写的嵌入式SQL程序

本章的所有例子均来自下面两个模式：

```
CLASS(CrsCode:CHAR(6), Semester:CHAR(6),
      Enrollment:INTEGER, ProfId:CHAR(9), Room:CHAR(10))
键为 CLASS: {CrsCode, Semester}
TRANSCRIPT(StudId:INTEGER, CrsCode:CHAR(6), Semester:CHAR(6),
           Grade:CHAR(1))
键为 TRANSCRIPT: {StudId, CrsCode, Semester}
```

属性CrsCode、Semester和Grade的取值范围和图4-5中所示的取值范围一致，即CrsCode是形如“MAT123”或者“CS305”的字符串，Semester是形如“F1999”和“S2000”的字符串，而Grade是形如“A”或者“B”的字母。

为了使应用程序作为一个整体和数据库通信，宿主语言语句和SQL语句必须能够访问公共变量。这样，由程序的宿主语言部分计算的结果才可以存储在数据库中，而从数据库中得到的数据也能被宿主语言语句处理。

图10-1的第一部分声明宿主程序的变量，也称为**宿主变量**（host variable），使用它们可以达到上面的目的。声明包含在EXEC SQL BEGIN DECLARE SECTION和EXEC SQL END DECLARE SECTION之间，这样才容易被预编译器发现并处理。但声明本身不是以EXEC SQL开头的。这样，声明以便能同时被预编译器和宿主预语言编译器处理。

⊖ MUMPS是Massachusetts General Hospital Utility Multi-Programming System的缩写。

图中的宿主变量用于SELECT语句中。注意，在SELECT语句中，每个宿主变量前面的冒号是用来与数据库模式的表名和列名相区别的。字段Enrollment的值被赋给宿主变量num_enrolled，在执行完SELECT语句后宿主语言语句可以以通常的方式访问。

因为CrsCode和Semester一起构成表CLASS的主键，所以SELECT语句只返回一行。这十分关键，因为如果SELECT返回多行，那么将哪一个赋值给变量num_enrolled呢？因此如果返回的行数多于一行，那么这个SELECT INTO语句就有错误。我们将在10.2.4节讨论返回多行的情况。

我们可以把宿主语言变量理解为将SQL语句参数化，它们用来传递标量值（scalar value），而不是表名、列名或者结构化的数据。WHERE子句中使用的宿主变量对应**输入参数**（in parameter），而INTO子句中的宿主变量对应**输出参数**（out parameter）。当执行语句的时候，输入参数的值用来构成一个完整的能被数据库管理器执行的SQL语句。但是要注意，SQL语句在输入参数的值确定之前可以被预处理，因为已知表名和列名（它们不可能作为参数）。所以第一次执行语句时使用的查询执行计划可以被保存下来以便在将来同一语句再一次执行时使用（因为每次执行时只是参数的取值不同），这是嵌入式SQL的一大优点。预编译器一个功能是在运行时调用子程序，传递值给宿主语言变量或者从宿主语言变量取得值，并以一种规范化的形式与DBMS进行通信。

10.2.1 状态处理

在现实世界中，事情不总是像想象的那样简单。例如当你试图连接远端服务器上的数据库的时候，服务器可能关闭或者拒绝连接。有时，试图执行INSERT语句也会被DBMS拒绝，因为它违反了约束。你可以把它们归类到错误情况中去，因为请求的操作没有被执行。在另外一些情况中，SQL语句被正确执行并且返回描述执行结果的信息。例如，DELETE语句返回删除的行数。SQL提供两种机制将描述这些情况的返回信息传递给宿主程序：长度为5的字符串SQLSTATE和**诊断域**（diagnostics area）。

图10-2在图10-1的基础上增加了状态处理部分。SQLSTATE在声明部分进行声明，因为它用来在DBMS和应用程序的宿主语言部分之间进行通信[⊖]。所有嵌入式SQL程序必须要有这个声明[⊖]。每当一个SQL语句执行结束，DBMS会将信息存储在这个字符串中。可以使用条件语句判断SQLSTATE的值，如果值为“00000”，则说明上一次的SQL语句执行成功，否则要判断出了什么样的异常，以便采取适当的措施。在上面的程序中，打印出一条状态信息。

除了在执行完SQL语句后判断状态外，还可以在执行第一个SQL语句前的任何一个地方加入下面的WHENEVER语句：

```
EXEC SQL WHENEVER SQLERROR GOTO label;
```

这时只要执行的语句返回非零状态，程序就会跳转到label标识的地方。在调用另外一个WHENEVER语句前，当前的WHENEVER一直有效。

⊖ 当在C语言中使用SQL的时候，SQLSTATE被声明为长度为6个字符的字符串，因为最后一个字符在C中用来标识字符串的终止。注意在SQL语句中，SQLSTATE前面没有冒号，因为预处理程序把它看作特殊的关键字。

⊖ 早期版本的SQL使用的技术稍有不同，状态是通过一个整型变量SQLCODE来传递的。

```
#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");
```

图10-2　增加状态处理

有关上一次SQL语句的执行结果的更详细的信息可以通过调用GET DIAGNOSTICS 语句从诊断域获得。一条SQL语句可能会抛出多个异常，诊断域可以记录所有异常的信息。

在这里要介绍一下字符串的标记方法，以免将来混淆。在SQL中（包括在嵌入式SQL中）字符串用单引号标识，而在很多宿主语言（例如C和Java）中用双引号标识，因此在一个包含嵌入式SQL语句的C语言程序中会同时有两种标识，例如:

```
semester = "F2000";
EXEC SQL SELECT C.Enrollment
     INTO :num_enrolled
     FROM CLASS C
     WHERE C.CrsCode = 'CS305'
          AND C.Semester = :semester;
```

字符串“F2000”出现在通常的赋值语句中，由C语言编译器处理，而“CS305”出现在SQL语句中，由SQL预处理程序处理。

10.2.2　会话、连接和事务

我们先介绍SQL标准的一些术语。SQL**代理**（agent）就是执行包含SQL语句的应用程序，SQL代理运行在SQL**客户端**（client）。SQL代理在对数据库执行任何操作之前，必须与SQL**服务器**（server）建立SQL**连接**（connection）。这个连接在服务器上创建一个SQL**会话**（session）。一旦建立起会话，SQL代理可以执行任意多个**事务**（transaction），直到从数据库断开连接、结束这个会话为止。

SQL连接通过执行CONNECT语句建立（也有可能隐式地建立），它的一般形式是:

```
CONNECT TO {DEFAULT | db-name-string}
     [AS connection-name-string] [USER user-id-string]
```

方括号里的选项是可选的，而花括号里有若干个选项，每个选项用竖线隔开，使用的时候必须有而且只能有一项被选用。

选项*db-name-string*是数据源的名称，*connection-name-string*是这个连接的名称，而*user-*

*id-string*是用户的名称（数据源可以使用它进行授权）。指定数据源的格式取决于不同的供应商。它可以是标识本地机上的一个数据库的名称的字符串，也可以是远程机上的一个数据库的名称，例如:

```
tcp:postgresql://db.xyz.edu:100/studregDB
```

程序可以通过CONNECT语句连接到不同的服务器上，一旦建立新的连接和会话，原来的连接和会话就会被替代。如果想在不同的连接和会话之间切换，可以通过执行下面的语句实现:

```
SET CONNECTION TO {DEFAULT | connection-name-string}
```

下面的语句终止SQL连接和会话（也有可能隐式地终止):

```
DISCONNECT {DEFAULT | db-name-string}
```

10.2.3 执行事务

SQL-92没有专门的语句来创建事务。每当第一条访问数据库的SQL语句在会话中执行的时候，就会自动创建一个事务[⊖]。事务通过COMMIT或者ROLLBACK结束，下一个SQL语句（COMMIT或者ROLLBACK之后）又会重新创建一个事务。这个过程称作链（chaining)，它将在第21章讨论[⊜]。

事务的默认执行模式是READ/WRITE，也就是说事务既可以读数据库也可以修改数据库。另外一种可选模式是READ ONLY，它只允许读数据库，以防止未授权用户对数据库进行更改。

第2章已经指出，虽然可序列化的调度可以保证所有应用正确执行，但是为了提高性能可以使用较低的隔离级别。默认的隔离级别是SERIALIZABLE，还有其他的可选项，我们将在第15章～第24章里详细讨论这些级别。

如果想改变默认的执行模式，可以使用SET TRANSACTION语句，例如:

```
SET TRANSACTION READ ONLY
    ISOLATION LEVEL READ COMMITTED
    DIAGNOSTICS SIZE 6;
```

这条语句设置模式为READ ONLY，同时设置隔离级别为READ COMMITTED。可以选择的隔离级别有:

```
READ UNCOMMITTED
READ COMMITTED
REPEATABLE READ
SERIALIZABLE
```

DIAGNOSITICS SIZE决定在诊断域中可以一次性描述的异常状况的数目，这些异常是由最近一次执行的SQL语句抛出的。

图10-3说明如何使用连接和事务语句以及状态处理。可以看到，声明之后是一个连接到

⊖ 在SQL:1999中，可以使用START TRANSACTION语句创建事务。它可以指定事务的特征，与SET TRANSACTION语句类似，后面将会描述这个语句。

⊜ SQL:1999也有COMMIT AND CHAIN和ROLLBACK AND CHAIN语句，这两个语句在完成提交和回退后立刻开始一个新的事务，而不需要等到执行下一个SQL语句才开始新的事务。

服务器上的语句，该语言没有提供显式创建事务的方式，相反一旦建立连接就会隐式地创建一个事务。

```
#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;
```

图10-3　在C程序中使用嵌入式SQL语句实现连接和事务处理，将学生从某门课程中注销

图中所示程序的作用是注销选修一门课程的学生。这里假设宿主语言变量semester存储当前的学期，可以通过调用操作系统的系统时间time()获得。程序对数据库状态进行两次更新，一次是从TRANSCRIPT表中删除学生注册某门课程的信息，另一次是在CLASS表中更新这门课程的人数Enrollment。只要DELETE和UPDATE语句中任何一个发生失败，就会导致SQLSTATE

的值不是“00000”，从而执行ROLLBACK语句[⊖]；如果没有发生错误，会执行COMMIT语句。最后事务断开连接。

当应用执行EXEC SQL COMMIT或者EXEC SQL ROLLBACK的时候，就会请求当前连接的数据库服务器S提交或者回退对数据库做的任何改变。但是应用程序执行的事务可能不只是访问单一数据库服务器。例如，通过建立若干个连接并在不同的连接之间进行切换，程序可以访问不同的数据库服务器，或者将计算的结果放到本地的文件系统中。因为COMMIT和ROLLBACK会通过连接被传递给数据库服务器S，所以相应的操作不能在这种情况下应用。

如果程序连接到多个数据库，不同服务器上的事务可以单独提交或者回退，然而包含单独事务的全局事务可能不具有原子性，这个问题将在第26章进一步讨论。

10.2.4 游标

作为数据库语言，SQL语言的一个优势是可以处理整张表，因此SELECT语句返回的可能是张表，这张表称为**查询结果**（query result）或者**结果集**（result set）。当语句是以直接的或者交互的方式执行而不是嵌入到程序里执行的时候，可能整个屏幕都无法完整地显示结果集。例如，下面的SELECT语句返回某个学期中参加某门课程的所有学生的Id和成绩：

```
EXEC SQL SELECT T.StudId, T.Grade
    FROM TRANSCRIPT T
    WHERE T.Semester = :semester
        AND T.CrsCode = :crs_code;
```

假如在宿主语言程序中要包含这样的查询，但是返回的行数只有在执行完语句后才知道，那么在程序里无法为未知的行数分配内存。例如如果使用数组，那么这个数组有多大呢？

为了处理这个问题，SQL引进了**游标**（cursor）机制，它允许应用程序一次处理结果集中的一行。游标类似于指针，可以指向结果集的任何一行。FETCH语句获得游标所指向的行并将该行的属性值赋给程序中的宿主语言变量。这种方式只需要将一行的值赋给变量。根据数据库模式可以知道每行的值的类型，从而声明适当类型的变量。

图10-4是使用了游标的嵌入式SQL程序片段。DECLARE CURSOR语句声明游标的名字为GETENROLLED，指定类型是INSENSITIVE（后面将会讨论这个限定词），并将它与某个SELECT语句联系在一起。但是这个SELECT语句现在并没有被执行，直到执行OPEN语句时才执行这条SELECT语句。

在示例中，SELECT有两个参数，分别是宿主变量crs_code和semester，它们将参数值传递给SELECT语句，要求返回属性CrsCode和Semester的值分别等于这两个参数值的行。当OPEN语句执行的时候会发生参数替代，接着执行SELECT语句。因此在打开游标后再对参数做修改不会对检索到的元组产生任何影响，OPEN会把游标定位到结果集的第一行之前。

执行FETCH语句会使游标向前移动，因此图10-4中的FETCH语句将使游标指向结果集的第一行，获取该行的值并将相应的值存储在宿主语言变量stud_id和grade中。CLOSE语句关闭

⊖ 如果DELETE语句失败，这时没有对数据库进行任何更新，也许有人会认为这里没有必要使用ROLLBACK。但是，必须通知系统事务已经结束以便在系统日志中记录适当的条目，而事务获得的锁也因此而释放。日志是系统确保事务原子性机制的一部分（见25.2节），而锁是系统保证隔离的机制的一部分（见23.5节）。

游标（这个示例仅仅获取结果集的第一行的值，下面的示例将更接近真实的情形）。

```
#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("Can't open cursor\n");
     exit(1);
}
EXEC SQL FETCH GETENROLLED INTO :stud_id, :grade;
if (strcmp(SQLSTATE,OK) != 0){
     printf("Can't fetch\n");
     exit(1);
}
EXEC SQL CLOSE GETENROLLED;
```

图10-4　使用游标

程序中的每个SQL语句都有一些可选项，DECLARE CURSOR语句也不例外，它的一般形式如下：

```
DECLARE cursor-name [ INSENSITIVE ] [ SCROLL] CURSOR FOR
     table-expression
     [ ORDER BY order-item-comma-list]
     [ FOR { READ ONLY | UPDATE [ OF column-commalist]} ]
```

其中，*table-expression*通常是一个表名，也可以是视图或者SELECT语句[⊖]。

选项INSENSITIVE的含义是执行OPEN语句将会有效地创建一个结果集的拷贝，所有通过游标的访问都指向这个拷贝。SQL标准使用“有效地”这个词的含义是标准中不指定使用什么样的方式来实现，但是必须和建立一个拷贝有相同的作用。这种返回数据有时也称作**快照**（snapshot）。

INSENSITIVE游标的语义很直观。当OPEN语句执行的时候，SELECT语句隐含的对基表的选择操作执行，并计算和存储结果集。在稍后可以通过游标访问这个拷贝。例如，图10-5显示的是执行图10-4得到的结果集，参数crs_code和semester的值分别是“F1997”和“CS315”，可以看到有一个游标指向结果集的拷贝，而不是指向原表。

⊖ 还有其他一些可选项，由于它们超出了本书的讨论范围，就不在这里讨论了。

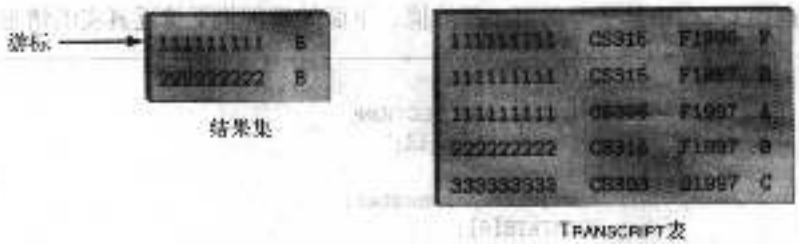


图10-5　使用INSENSITIVE游标有效地生成结果集，获取行的时候并没有访问原表

因为INSENSITIVE游标访问的是结果集的拷贝，因此打开INSENSITIVE游标后当前事务里任何对基表的操作都不会通过游标看到，当然通过游标进行的操作例外。例如，事务在打开游标后执行下面的语句：

```
INSERT INTO TRANSCRIPT
VALUES ('656565656', 'CS315', 'F1997', 'C');
```

它直接插入（不是通过游标）一行新的记录到表TRANSCRIPT中。虽然在GETENROLLED打开的时候crs_code的值是"CS315"而且当前的semester是"F1997"，但是游标不能获得这个新行的值。即使打开游标后事务使用UPDATE语句更新了在结果集中的某行的值，游标也看不到。类似地，在打开游标后由其他并发事务对基表进行的修改同样无法通过游标看到。

当没有指定INSENSITIVE选项的时候，SQL标准没有规定对基表的操作会有什么样的结果，数据库供应商可以自由选择他们认为合适的实现方式。许多供应商使用KEYSET_DRIVEN，它是ODBC标准的一部分，其相关内容将在10.6节中讨论。

如果没有指定INSENSITIVE，也没有声明游标为READ ONLY，并且游标声明中的SQL查询满足可更新视图（见6.3节）的要求，那么基表当前行的更新和删除可以通过游标实现，这时称游标是可更新的。UPDATE和DELETE语句分别用于实现更新和删除操作，不过WHERE子句被WHERE CURRENT OF *cursor-name*代替。下面是一般的语法：

```
UPDATE table-name
SET assignment-comma-list
WHERE CURRENT OF cursor-name
```

和

```
DELETE
FROM table-name
WHERE CURRENT OF cursor-name
```

因为INSENSITIVE游标指向结果集的拷贝，所以UPDATE和DELETE对用于计算出结果集的基表不会有任何影响。为了避免混淆，这些操作不能通过INSENSITIVE游标执行。

如果想对结果集的行进行排序，可以通过ORDER BY子句实现。例如在GETENROLLED的声明中指定

```
ORDER BY Grade
```

那么结果集将会按照Grade的升序排序。

FETCH语句的一般形式是：

```
FETCH [ [ row-selector ] FROM ] cursor-name
INTO target-commalist
```

其中，*target-commalist*是宿主语言变量的列表，变量的数量和类型必须与游标指向的结果集的属性的数量和类型一致，*row-selector*指定游标在获取下一个满足条件的行之前的移动模式，它有如下选项：

FIRST
NEXT
PRIOR
LAST
ABSOLUTE *n*
RELATIVE *n*

如果*row-selector*指定为NEXT，游标移向结果集的下一行，同时该行的值被存储在*target-commalist*中的变量里；如果指定为PRIOR，游标移向前一行，并获取这一行的值。与此类似，FIRST将使游标指向第一行，而LAST使游标指向最后一行。最后，ABSOLUTE *n*使游标指向表中的第*n*行，而RELATIVE *n*指向相对当前行之前或者之后的第*n*行。如果没有指定*row-selector*，那么默认的方式是NEXT。如果是默认方式并且GETENROLLED使用上面的ORDER BY子句，那么会按照成绩的升序获取每行。

游标声明中的选项SCROLL的含义是FETCH语句可以使用任何移动模式；如果没有指定SCROLL，那么只能使用NEXT移动模式。

图10-6中对图10-4进行扩充，使得参加某门课程的所有学生都能被处理。循环中使用FETCH语句，循环在FETCH执行失败时终止。后面的条件语句判断是不是结果集的所有行都被访问过，如果都访问过，那么SQLSTATE的值为“02000”，否则打印出错信息并退出程序。

```
#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("Can't open cursor\n");
     exit(1);
}
```

图10-6　使用游标扫描一张表

```
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;
```

图10-6　(续)

10.2.5 服务器存储过程

许多DBMS供应商将**存储过程**（stored procedure）作为数据库模式的一部分。这些过程能被客户端应用调用并在服务器端执行。存储过程有如下的优点:

- 由于存储过程在服务器端执行，因此只需要将执行结果从服务器端传到客户端的应用程序即可。例如，一个存储过程使用游标访问一个庞大的结果集并分析得出一个值，这个值被传给应用程序。如果不在服务器端运行存储过程而让应用程序使用游标，那么必须将这个庞大的结果集传给应用程序进行分析，这势必增加通信费用和响应时间。
- 存储过程中的SQL语句可以在执行应用程序前预备，因为它是作为模式的一部分存储在服务器端的，而嵌入式SQL语句通常都是在运行时准备的。因此即使存储过程仅含有一条SQL语句，也比直接嵌入在应用里运行效率高得多。

  在14.6节中说过，这个优点在某些情况下可能变成缺点，因为查询计划会变得越来越陈旧，因此，“旧”的存储过程虽然避免了查询预备的开销，但却由于低效和过时的查询计划而增加运行时的开销。一些供应商（例如Sybase）提供了WITH RECOMPILE选项，可以在调用存储过程的时候指定这个选项，这样应用程序就能定期地重新编译存储过程以保证查询执行计划总是保持最新状态。
- DBMS可以在存储过程的级别上使用GRANT EXECUTE语句对授权进行检查，GRANT EXECUTE对4.3节介绍的GRANT语句进行了扩展。这样即使那些没有被授权允许访问某些数据库关系的用户也可能被授权通过执行存储过程来访问这些关系。例如，注册事务和修改成绩事务可以通过调用使用SELECT语句的存储过程访问某张表中的相同行，但是一个存储过程被授权给学生，而另一个存储过程被授权给教职员工。

  另外，存储过程能超出SQL GRANT语句范围控制用户能做哪些事。例如，存储过程可以要求只有学生才能够执行为自己注册的事务。
- 应用程序员不需要知道数据库模式的细节，因为所有的数据库访问都封装在存储过程中。例如，注册办公室可以提供注册事务的存储过程，程序员只要知道如何调用即可。
- 系统的维护变得更加简单，因为只需要对服务器上的存储过程的拷贝进行维护和更新。相反，如果将存储过程的代码包含在许多不同的应用程序中，那么这些代码就需要同时

被维护和更新。

• 存储过程代码的物理安全性得到提高，因为代码被放在服务器端而不是在应用程序中。

在22.2.1节中讨论支持存储过程的系统体系结构的时候会进一步说明这些优点。

最初版本的SQL-92标准不支持存储过程，这种支持直到1996年才被加入。现在我们通过例子来说明存储过程语言。

图10-7是一个存储过程的DDL声明，这个存储过程用于注销学生选取的某门课程。这里假设在调用存储过程之前应用程序已经连接到DBMS，而在执行存储过程后会断开。

```
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;
```

图10-7　存储过程：注销学生选取的某门课程

这个过程是使用SQL/PSM（SQL Persistent Stored Module，持久存储模块）语言编写的，符合SQL-92扩展标准[⊖]。这个标准同时规定在其他语言（例如C语言）中编写存储过程的规范。注意，SQL/PSM仅仅是另一种宿主语言，SQL可以嵌入其中。示例中的过程共有3个输入参数，用关键字IN标识，同时有两个输出参数，用关键字OUT标识。标准也支持同时作为输入和输出的参数，用INOUT标识。

过程的主体包含在BEGIN/END块中，选项ATOMIC确保将整个块将作为一个原子单元执行。接下来是一组变量声明，可以使用DEFAULT语句（可选的）指定初始值。注意，不管是参数还是宿主变量（即在存储过程中声明的PSM变量）都没有前缀“:”，这是因为SQL/PSM是一种统一语言，其编译器能够知道哪些语句是宿主变量声明，哪些语句是控制语句，哪些语句是SQL查询和更新语句。这个例子说明，变量既可以在查询和更新语句之中使用，也可以在查询和更新语句之外使用。特别要指明的是，它们的值可以通过SET子句设置，并且可以作为算术和字符串表达式的一部分。

PSM是一个功能强大的成熟的编程语言，可以与SQL完美地结合起来。这里只是简单介绍，有许多特征没有加以介绍，例如循环结构、CASE语句以及游标等等。有关PSM和存储过程的详细内容可以参阅[Melton 1997]，这里仅仅介绍PSM的错误处理，它与嵌入式SQL稍有不同。

与每次执行更新语句之后检查变量SQLSTATE（或者使用WHENEVER语句）不同，在SQL/PSM中通过声明**条件处理程序**（condition handler）处理SQLSTATE的不同值。条件处理程序是一个程序，在SQL语句执行中断后如果SQLSTATE的值与某个条件处理程序相关联的值匹配，那么就会执行这个条件处理程序。在我们的例子中有两个处理程序与SQL EXCEPTION相关联，它是可以匹配任何“错误代码”（任何不以00、01和02开头的SQLSTATE值）的条件。每个处理程序都有处理范围，由BEGIN/END界定，这样就可以将不同的处理程序和不同的SQL语句联系在一起。两个处理程序的类型都是UNDO，这意味着DBMS在执行完处理程序后将回退存储过程的结果并退出。UNDO类型的处理程序只能出现在BEGIN ATOMIC模块中。另外，还可以指定CONTINUE与EXIT类型的处理程序，前者在执行完处理程序后继续进行似乎没有任何错误发生，而后者在执行完处理程序后将直接退出程序并且不会回退存储过程所作的操作。

在交互式SQL中（在一个存储过程中），可以使用下面的语句调用另外一个存储过程：

```
CALL procedure_name(argument-commalist);
```

在嵌入SQL的宿主语言中，CALL语句前要加EXE SQL。这时，过程参数是添加前缀“:”的宿主语言变量。例如，调用Deregister()存储过程的方式是：

```
EXEC SQL CALL Deregister(:crs_code,:semester,:stud_id);
```

其中crs_code、semester 和stud_id都是宿主变量。

10.3 再论完整性约束

一个一致的事务将数据库从初始状态变到最终状态，这两个状态都满足所有的完整性约

⊖ 其他供应商在此之前开发了类似于SQL/PSM的语言，但与SQL/PSM有一定的区别。例如，Oracle使用PL/SQL，Microsoft和Sybase提供Transact-SQL，而Informix提供SPL语言。

束。但是事务执行过程中的一个中间状态却可能违反某个约束。例如，在参照完整性的例子中，如果对行的引用在该行之前加入就违反了约束。同样，图10-7中的过程Deregister里DELETE语句产生的状态违反下列完整性约束，即TRANSCRIPT表中参加某门课程的学生数目必须与CLASS表中与这门课程对应的字段NumEnrolled值相等，如果DBMS在执行完每条语句后立即进行约束性检查，那么将会拒绝执行DELETE操作。

为了处理这种情况，SQL允许应用控制每个约束的模式。如果约束是**立即模式**（immediate mode），那么在事务中执行完任何一条可能违反该约束的SQL语句后会立即检查；如果约束是**延迟模式**（deferred mode），那么直到事务请求提交的时候才会检查该约束。

- 如果约束检查是立即模式并且某个SQL语句违反该约束，那么该语句会被回退，而且相应的错误代码通过SQLSTATE返回。事务可以从诊断域获得被违反的约束的名称。
- 如果约束检查是延迟模式，那么直到事务请求提交的时候才会检查约束。如果这时发现违反某个约束，那么将会中止事务并返回相应的错误代码。因此如果完整性约束在事务执行的过程中会被违反，那么显然选择延迟模式更合适。

当定义一个约束的时候，可以指定不同的选项，例如一个表约束遵守下面的规则：

```
[ CONSTRAINT constraint-name ] CHECK conditional-expression
    [ { INITIALLY DEFERRED | INITIALLY IMMEDIATE}]
    [ { DEFERRABLE | NOT DEFERRABLE } ]
```

第一个选项确定约束的初始模式。如果选定INITIALLY DEFERRED，那么将该约束设为延迟模式，直到使用SET CONSTRAINT语句重新设定为止。第二个选项确定是否以后能被SET CONSTRAINT语句设定为延迟模式，显然NOT DEFERRABLE和INITIALLY DEFERRED是相互矛盾的，不能同时使用。

一个DEFERRABLE约束可以在不同时间指定为IMMEDIATE和DEFERRED。通过使用下面的语句可以在立即模式和延迟模式之间切换：

```
SET CONSTRAINTS { constraint-list | ALL } { DEFERRED | IMMEDIATE }
```

其中*constraint-list*是CONSTRAINT语句中指定的约束的名称列表。

10.4 动态SQL

在编写程序的时候设计静态SQL并将它嵌入到应用程序中。语句的所有细节信息，例如类型（SELECT，INSERT）、模式信息（语句中引用的属性名和表名）以及作为输入输出参数的宿主语言变量，在编译的时候就已经知道。

但是在某些应用程序里，在编写的时候不一定知道上述所有信息。为了处理这种情况，SQL-92定义专门的语法，通过包含宿主语言程序中的**指令**（directive）来构造、预备和执行SQL语句。语句由程序的宿主语言部分在运行时构造。为了和静态SQL区别，这组指令被称为**动态SQL**（dynamic SQL），它的另一个含义是表示SQL语句可以在运行时动态创建。和静态SQL一样，指令使用语法和宿主语言相区别，因此动态SQL也是语句级的接口。由于静态和动态SQL使用相同的语法，因此能被相同的预编译器处理，一个应用程序可以同时包含静态和动态SQL结构。

构建的SQL语句在程序中是作为字符串类型宿主语言变量的值出现的，在运行时作为动态SQL指令的参数传递给DBMS做预备。一旦预备完毕，就可以执行语句。和静态SQL相同，语句必须使用DBMS能够识别的专门语言加以构建。

假设学校有一个学生注册系统，学生可以利用该系统在任何一个分校里注册任何一门课程。同时假定每个分校都拥有自己的课程数据库，并以自己的习惯来命名表和属性。当一个学生执行注册的时候，应用程序就会创建合适的SQL语句执行在指定分校的注册。例如，可以将每个分校相应的SELECT语句以字符串的形式存储在文件里，在运行的时候获取相应的字符串并赋值给宿主语言变量，从而预备并执行。程序也可以使用合适的SQL语句框架，提前预备，在运行的时候填充合适的表名和属性名即可。

上面的例子中，在模式和在所有分校注册某学生的SQL语句之间有某些共同点，因此应用程序知道将要执行的SQL语句的某些信息。再举一个例子，某个应用监控着终端，允许用户输入任意的SQL语句在某个数据库管理器上执行。现在，应用程序没有将要送往数据库执行的SQL语句的任何信息，仅仅将这个字符串赋值给变量并将它传递给DBMS进行处理。类似地，考虑将电子数据表格与数据库相连的一个应用。运行的时候，用户指定电子数据表格的某项的值包含数据库项目，这些项目必须通过查询的方式的到。查询可以由用户通过图形符号的方式表达，应用程序将这些符号转换成SELECT语句。同样，这里没有将要执行的SQL语句的信息，也没有语句所要访问的数据库模式的信息。

缺乏信息会导致一个问题，因为必须知道SQL语句输入输出参数的域才能确定宿主语言变量的合适类型，以便传递参数。在应用程序编译的时候不能获得信息的情况下，动态SQL提供指令允许程序在运行的时候查询DBMS以获得模式信息。

10.4.1　动态SQL的语句预备

下面的示例说明如何动态创建SQL语句[⊖]：

```
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;
```

这里除了CrsCode和Semester的值在编译的时候不知道外，SELECT的确切形式也不是很清楚，因为查询检索的列依赖于用户在运行时的输入。PREPARE语句将存储在变量my_sql_stmt中的查询字符串传递给数据库管理器进行预备，并将预备语句命名为st1。注意，这里st1是一个SQL变量（只能用在SQL语句中）而不是宿主语言变量，因此前面没有“:”前缀。

⊖ 对C不太熟悉的读者可以在这里得到帮助。函数scanf()读取用户的输入并将结果赋给变量column。SELECT语句中的反斜线表明下一行是这一行的继续。函数sprintf()使用变量column的值替换%s，最后将结果赋给变量my_sql_stmt。

EXECUTE语句执行语句st1。这个字符串包含两个输入参数，用“?”标识，这些参数将用USING子句里的宿主语言变量的值来替换。另外，INTO子句里的宿主变量用来接收结果。标记“?”被称为**动态参数**（dynamic parameter）或者**占位符**（placeholder），可以用在SELECT、INSERT、UPDATE以及DELETE语句中。一旦预备完毕，st1可以执行多次，并使用不同的宿主语言变量作为参数。PREPARE语句生成的查询执行计划用于当前会话中所有后续的执行中。

注意，与SELECT INTO一样，EXECUTE INTO要求查询结果为一行。如果结果包含多行，就必须使用游标而不能再使用EXECUTE INTO。关于动态SQL语句里的游标将在10.4.3节讨论。

动态SQL中的参数传递与静态SQL里的参数传递是不相同的，它使用占位符而不是宿主语言变量名称执行预备，并且INTO子句是和EXECUTE语句联系在一起而不是和SELECT语句联系在一起。为什么参数传递不同呢？

- 在静态SQL里，WHERE和INTO子句中的宿主语言变量名称是作为参数提供给预编译器。预编译器在编译的时候分析这些子句，这些变量在编译器的符号表中加以描述，符号表用于将变量名转换成地址（回忆一下，DECLARE SECTION中声明可以同时被预编译器和宿主语言编译器处理）。符号表中的项包含变量名和地址间的映射以及类型信息，这个类型信息被预编译器用来生成数据库里的数据和这些变量之间转换的代码，在宿主语言程序里执行SQL语句的时候执行这个代码。
- 在动态SQL里，正如示例所示，编译器可能得不到SQL语句，因此如果宿主语言变量作为参数嵌入到语句中，就不能使用符号表中的信息来处理这些参数。为了能够在编译的时候获得参数信息，有两种可选择的方法，一种方法是通过*SQLDA*机制，这将在稍后讨论；另一种方法是像示例所示那样使用USING和INTO子句提供输入输出变量。对于后者，预编译器生成代码在这些变量中存取参数的值从而和DBMS通信。

应用程序应该尽可能地使用静态SQL设计，因为动态SQL的效率通常比较低。将预备和执行分隔开会增加通信和处理的开销。如果语句多次执行，增加的开销会分摊到每次执行中去，因为预备只需要做一次。另外，在某些情况下可以避免这个开销。例如，对于那些只需要执行一次的SQL语句，可以通过EXECUTE IMMEDIATE指令将预备和执行绑定在一起[⊖]：

```
EXEC SQL EXECUTE IMMEDIATE
    'INSERT INTO TRANSCRIPT '
    || 'VALUES ("656565656", "CS315", "F1999", "C") ';
```

更常见的情况是，SQL语句可以被放在一个宿主语言的字符串变量中，这时EXECUTE IMMEDIATE语句的形式如下：

```
EXEC SQL EXECUTE IMMEDIATE :my_sql_stmt;
```

注意，变量my_sql_stmt前有符号“:”，前面介绍过，这表示my_sql_stmt是宿主语言变量而不

⊖ 注意字符串的处理。INSERT INTO是动态SQL语句的一部分，因此我们使用单引号标识字符串。由于语句太长，所以被分为两个字符串，并使用SQL里常用的连接符“||”连接。要在字符串中包含引用符号，它必须用双引号，例如“656565656”。这使得SQL分析器能够正确的解析字符串。最终每个双引号（“）被单引号代替，从而生成正确的SQL语句。

是SQL变量。

EXECUTE IMMEDIATE仅仅是将PREPARE和EXECUTE语句绑定在一起的快捷方式，并且在语句执行完后并不保存执行计划。这个快捷方式强加了一些语法限制，不允许使用某些逻辑。例如它不允许有INTO子句，因此不能有输出参数（也就是不能检索数据），即不能使用SELECT语句。

EXECUTE IMMEDIATE也不允许使用USING子句，不过这个限制并不很严格。EXECUTE语句使用USING的原因是一旦语句预备结束（其中动态输入参数标记为？），那么就可以执行多次而每次使用不同的输入参数。因为EXECUTE IMMEDIATE在一步中就将预备和执行完成（而且不需要保存预备语句），因此是不需要动态参数的，我们可以通过宿主语言将合适的参数值嵌入到SQL语句中，然后将创建好的完整语句传递给EXECUTE IMMEDIATE。

同静态SQL一样，SQLSTATE用来返回PREPARE、EXECUTE、EXECUTE IMMEDIATE语句和其他动态SQL语句的状态。

10.4.2 预备语句和描述符区

虽然10.4.1节的示例中查询是在运行时创建的，而且输出列的名字在编译的时候并不知道，但是这个例子仍然比较简单的，因为程序知道查询目标列表只含有一个属性名并且知道WHERE子句有两个动态参数。知道输入输出参数的个数就可以使用EXECUTE语句并为USING和INTO子句提供具体的变量名。这时，预编译器可以提供自动格式转化。例如，如果用户输入Enrollment的类型为整型，而预编译器确定INTO变量的类型是字符串型，所以需要自动转换为字符串型。因为DBMS在运行时提供SELECT语句返回值的类型，所以转换要到那个时候才确定。

现在假设在运行时用户可以指定目标列表中属性的个数和WHERE从句中的条件，这样在设计时就不知道输入和输出的个数，因此也就不能使用10.4.1节的EXECUTE语句形式，因为在编写程序的时候不知道要在INTO和USING子句中提供多少个变量。

为了处理这种情况，动态SQL提供了一种运行时机制，通过这种机制程序可以从DBMS中请求语句参数的描述信息。例如，程序允许用户查询数据库中的任意一个只有一行的关系[⊖]：

```
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
sprintf(my_sql_stmt,
        "SELECT * FROM %s WHERE COUNT(*)=1",
        table);
```

这条语句没有输入参数，但是输出参数的个数不确定，因为预先不知道FROM子句使用的表，所以也就不知道SELECT语句中的“*”表示多少个表属性（即输出参数）。虽然对这些参数一无所知，但是一旦语句预备完毕，DBMS就可以知道所有的信息，并将这些信息提供给程序，它是通过描述符区来完成的。应用程序首先要求DBMS分配一个**描述符区**（descriptor area），

⊖ 这个例子中使用EXECUTE语句，它能处理只有一行的查询。对于一般的情况，可以使用游标和FETCH语句。将在下一节讨论动态SQL中的游标。

有时候也被称作SQLDA，参数的信息就存储在这个空间里。当语句预备完毕后，应用程序可以请求DBMS将参数信息填充到这个描述符区中。对于上面的例子，可以按如下方式填充描述符区：

```
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';
```

这里st_output是 SQL变量，用于命名已分配的描述符区。ALLOCATE DESCRIPTOR语句在数据库管理器中创建最多能够描述21个参数的空间（由WITH MAX指定）。描述符区可以视为由一个一维数组加上一个记录实际参数个数的变量组成的。每一项都有固定结构，这些结构由一组描述特定参数的部分组成，这些特定参数包括名称、类型，以及值。所有字段最初都没有定义。DESCRIBE语句告诉DBMS将预备语句st（包含名称、类型和参数的长度）的第i个输出参数的元信息填充到描述符st_output的第i个项中，同时将参数的个数记录在描述符中。

现在回到我们的例子当中，程序使用下面的指令执行预备好的语句st:

```
EXEC SQL EXECUTE st
    INTO SQL DESCRIPTOR 'st_output';
```

它将返回行的第i个属性的值存储在st_output的第i项中。程序可以用GET DESCRIPTOR语句在循环中依次访问返回行的每个字段获得属性以及值等元信息，参考图10-8。最后，程序调用DEALLOCATE DESCRIPTOR来释放st_output占用的空间。

实际上，SQL语句执行时输入参数的个数不确定的情况是很少见的。如果遇到这种情况，程序可以使用ALLOCATE和DESCRIBE INPUT为占位符“?”标识的输入参数创建描述符区。和前面一样，这个描述符区是一个数组，每一项对应着一个占位符。然后程序使用DESCRIBE INPUT语句请求DBMS用每个输入参数的类型、长度以及名称等信息填充描述符区。这时需要使用SET DESCRIPTOR语句提供参数值。对具体的细节有兴趣的读者可以参考SQL手册，例如[Date and Darwen 1997, Melton and Simon 1992]。

```
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 i^th 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;
```

图10-8　使用GET DESCRIPTOR的例子

```
    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
```

图10-8　(续)

10.4.3 游标

与SELECT INTO相同，EXECUTE INTO也要求查询结果必须是单行的，而在一般情况下查询结果都为多行，这时需要使用游标机制去扫描。幸运的是，和静态SQL一样，在动态SQL中可以为预备语句定义游标，尽管语法稍有不同。例如，下面这个例子是和图10-4中的程序等价的（为了简便，本例中没有加入状态检查）：

```
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;
```

与静态SQL里的游标相同，DECLARE CURSOR语句有INSENSITIVE和SCROLL选项；FETCH语句允许为可卷的游标指定row-selector；UPDATE和DELETE语句可以通过非INSENSITIVE类型的游标执行。

10.4.4 服务器端的存储过程

一些DBMS允许使用动态SQL调用存储过程，下面的示例调用了图10-7中的存储过程：

```
my_sql_stmt = "CALL Deregister(?,?,?)";
EXEC SQL PREPARE st3 FROM :my_sql_stmt;
EXEC SQL EXECUTE st3
    USING :crs_code, :semester, :stud_id;
```

PREPARE语句准备调用Deregister存储过程，而EXEC SQL EXECUTE语句调用存储过程并提供宿主语言的变量的值作为参数。

10.5 JDBC和SQLJ

JDBC[1]是数据管理器的一组API，提供在Java程序中执行SQL语句的调用级接口。和动态SQL一样，在运行时，SQL语句可以作为字符串变量的值创建。JDBC由SUN公司开发，是Java语言的一部分。

而SQLJ是Java程序的声明级接口，类似于静态嵌入式SQL。不同于JDBC的是，SQLJ是由各公司的联盟开发的，已经成为一个独立的ANSI标准。

JDBC和SQLJ都用来通过因特网访问数据库。比其他各种嵌入式或者动态SQL更具可移植性。

10.5.1 JDBC的基本概念

回忆一下，在动态SQL和静态SQL里，必须在编译的时候知道目标DBMS，因为SQL语句要使用它的专用语言。对于动态SQL而言，在编译的时候可以不知道模式。而JDBC在编译的时候既不需要知道DBMS，也不需要知道模式。应用程序使用JDBC的SQL专用语言，而与所使用的DBMS无关。和动态SQL一样，JDBC提供相应的功能允许程序运行时向DBMS请求关于模式的信息。

图10-9显示在运行的时候JDBC处理不同DBMS的机制。应用程序通过**驱动程序管理器**（driver manager）模块与DBMS通信。当应用程序第一次连接到某个DBMS的时候，驱动程序管理器选择某个驱动程序（即一个JDBC模块）去执行以下的操作：将SQL语句从JDBC的SQL专用语言转换成DBMS的专用语言。JDBC为常用的DBMS保留专门的驱动程序。驱动程序管理器针对需要访问的DBMS选择相应的驱动程序。当执行SQL语句的时候，应用程序将表示该条语句的字符串发给相应的驱动程序。驱动程序经过必要的转化后将语句传给DBMS，语句在DBMS中进行预处理并执行。

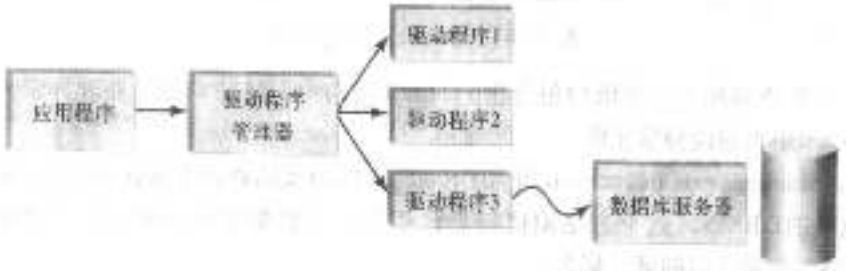


图10-9 通过JDBC连接数据库

JDBC的软件体系结构由一组预先定义的类组成，例如DriverManager和Statement。这些对象类的方法提供数据库的调用级接口。JDBC程序必须首先装入这些预先定义的类，接着创建对象类型的相应实例，最后调用合适的方法访问数据库。图10-10显示Java程序调用JDBC的一般框架，下面逐一进行解释：

- import java.sql.*导入包java.sql中的所有类，使得Java程序可以调用JDBC API。图中出

[1] JDBC是Sun公司的商标，虽然SUN声称JDBC不是缩写，然而通常认为它代表Java数据库连接（Java Database Connectivity）。

现的类Connection、Statement、DriverManager和Result都在这个包里，其他的类PreparedStatement、CallableStatement、ResultSetMetadata和SQLException也在这个包中，其中CallableStatement是PreparedStatement的子类，而PreparedStatement是Statement的子类。

- Class.forName()装入指定的数据库驱动程序，并在驱动程序管理器中注册。虽然命名方式不是很直观，但是在包java.lang package中确实包含一个名为Class的类，它提供在运行时装入某个类的方法。forName()是它的一个静态方法，用来装入和注册指定的驱动程序。如果应用程序想连接到多个数据库，那么可以为每个数据库单独调用Class.froName()方法。

```
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("Can't load driver\n");
     System.exit(1);
  } catch (SQLException e) {
     System.err.println("Can't 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
```

图10-10　JDBC的过程调用框架

这条语句和下一条语句包含在try/catch结构中，该结构用来处理异常。我们将在10.5.5节中再讨论异常处理。

- DriverManager.getConnection()使用DriverManager类的静态方法getConnection()连接给定地址的DBMS。这个方法测试每个已经装入的数据库驱动程序是否能连接到那个DBMS。如果可以的话，那么：

  1) 使用指定的用户ID和密码建立连接。

  2) 创建一个Connection对象，并将它赋给先前声明的变量con1。

  参数url含有目标DBMS的URL（统一资源定位符，Uniform Resource Locator）。这个URL是从数据库管理员那里获得的。例如：

```
jdbc:odbc:http://server.xyz.edu/sturegDB:8000
```

前缀jdbc指定连接数据库的主协议（毫无疑问就是JDBC）。第二部分odbc指定子协议（跟供应商和驱动程序有关）。接下来的一部分就是数据库服务器的地址，8000是服务器监听的通信端口号。

- con1.createStatement()使用Connection对象con1的方法createStatement()来生成Statement对象并将它赋给Statement的变量stat1。
- stat1.executeQuery()使用Statement对象stat1预处理并执行作为参数的SELECT语句。SQL语句可以没有输入参数，返回的结果集存储在ResultSet对象res1中，它是由executeQuery()方法创建的。这个方法类似于动态SQL中的EXECUTE IMMEDIATE指令，因为它也是将预备和执行绑定在一起完成。不同之处在于，JDBC方法为返回数据创建了一个对象，而EXECUTE IMMEDIATE没有这样做。在10.5.3节中将讨论ResultSet类。

  要执行UPDATE、DELETE、INSERT语句或者DDL语句，可以使用如下所示的形式：

  ```
  stat1.executeUpdate(···some SQL statement···);
  ```

  这个函数返回一个整数，表明有多少条记录受到影响，对于CREATE这样的DDL语句返回的是0。
- stat1.close()和con1.close()分别释放Statement对象和Connection对象（断开连接）。当执行完一条语句以后，Statement对象可以不关闭以执行另一条语句。

10.5.2 预处理语句

图10-10调用的executeQuery()和 executeUpdate()会预处理并执行指定的语句，为了将预处理和执行分开，可以先调用

```
PreparedStatement ps1 =
    con1.prepareStatement(···SQL preparable statement···);
```

返回一个PreparedStatement对象给ps1，然后调用

```
ResultSet res1 = ps1.executeQuery();
```

或者

```
int n = ps1.executeUpdate();
```

这里executeUpdate()返回一个整数，表明对多少行进行了更新。

PreparedStatement是Statement类的子类。注意，这两个类都有executeQuery()方法和executeUpdate()方法，但是PreparedStatement的方法没有参数。

与动态SQL相同，prepareStatement()的字符串参数可以包含占位符“?”标识的动态输入参数，并且在执行前必须为占位符指定一个具体值。这可以使用setXXX()方法完成。例如：

```
ps1.setInt(1, someIntVar);
```

会用宿主语言变量someIntVar的值替换第一个占位符。PreparedStatement类有一组setXXX()方法，XXX表示不同类型的输入参数，例如Int、Long和String等。

10.5.3 结果集和游标

执行查询语句后会将结果（输出参数）的值存储在ResultSet对象中，可以通过游标检索结果集的行。JDBC游标通过ResultSet类的方法next()实现，调用ResultSet对象的时候，它会依次扫描整个集合。在图10-11里，结果集对象存储在变量res2中，res2.next()方法将游标向前

移动一行。

```
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();
```

图10-11　在JDBC程序中使用游标

程序先预处理查询，提供参数，然后执行查询，接着使用while循环检索结果集的行。next()方法移动游标，当没有记录返回的时候返回false。在while循环的每次迭代中调用getLong()和getString()检索结果行，有两种方式，一种方式是以属性名为参数，另一种方式是以位置为参数。因此，如果studId和grade分别是记录集res2的第一个和第二个属性的话，那么程序可以使用getLong(1)和getString(2)分别获得这两个属性的值。ResultSet类有一组getXXX()方法，XXX为Java的基本类型。

JDBC提供三种类型的结果集，它们在滚动和灵敏度方面有所不同。

- 顾名思义，forward-only类型结果集是不可以滚动的，游标只能向前移动。它使用DBMS默认的游标类型（INSENSITIVE或者非INSENSITIVE）。
- scroll-insensitive类型结果集。它是可以滚动的，并且使用指定DBMS的INSENSITIVE类型的游标，因此得到结果集后（不管是生成结果集的事务操作的还是其他事务操作的），对数据表的操作都不会在结果集中看到。
- scroll-sensitive类型结果集。它是可以滚动的，但是使用了非INSENSITIVE类型的游标。

10.2.4节说过SQL标准没有定义当INSENSITIVE未被选择时的行为。数据库厂商可以自由地实现他们认为合适的语义。JDBC通常提供所访问的DBMS所支持的语义。许多厂商实现了KEYSET_DRIVEN语义，它是ODBC规范的一部分，我们将在10.6节中讨论这个规范。在这个语义中，创建结果集之后，记录的更新和删除是可见的，但是插入是不可见的。JDBC提供一系列的方法询问驱动程序以决定期望看到的内容。

如果目标DBMS不支持应用所请求的滚动和灵敏度，就会给出警告。

结果集可以是只读的也可以是可更新的。对于可更新的结果集，SQL查询必须满足可更新视图的条件（见6.3节），例如，下面的createStatement()的变体创建一个Statement对象s3:

```
Statement s3 =
    con1.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                         ResultSet.CONCUR_UPDATABLE);
```

如果稍后调用executeQuery()方法，那么所生成的结果集就是可更新的而且是scroll-sensitive。其他选项可以参阅JDK文档中关于ResultSet和Connection类的说明。

一个返回字符串属性Name的SELECT语句生成一个可更新的结果集res，它的当前行可以使用下面的方法将Smith赋给Name，从而更新属性Name的值:

```
res.updateString("Name", "Smith");
```

同setXXX()和getXXX()方法一样，对于每个基本类型都有相应的updateXXX()方法。

当需要更新的行的值被构建以后，执行下面的语句更新原表:

```
res.updateRow();
```

不仅可以将可更新结果集中的行更新或删除，而且还可以通过结果集插入新行，这是与静态SQL和动态SQL不同的。要插入的行的列值首先在与记录集关联的缓冲器中装配，然后调用res.insertRow()方法将缓冲器里的行插入到结果集res中，同时也插入到数据库中。

10.5.4 获取结果集的信息

和动态SQL相同，编写程序的时候结果集的信息是无从得知的，JDBC提供查询DBMS的机制来获取这类信息。例如，JDBC提供ResultSetMetaData类，其方法可以用于获取信息。

```
ResultSet rs3 = stmt3.executeQuery("SELECT * FROM TABLE3");
ResultSetMetaData rsm3 = rs3.getMetaData();
```

上面的语句创建ResultSetMetaData对象rsm3，并用关于结果集rs3的信息填充它，这时可以使用下面的方法查询该对象:

```
int numberOfColumns = rsm3.getColumnCount();
String columnName = rsm3.getColumnName(1);
String typeName = rsm3.getColumnTypeName(1);
```

第一个方法返回结果集的列数，后两个方法分别返回第一列的名称和类型。使用这些方法，甚至不需要知道结果集的模式，就可以迭代访问每行的每个字段，检查它们的类型，并获取数据。例如，如果rsm3.getColumnTypeName(2)返回“Integer”类型，那么程序就可以调用rs3.getInt(2)获得当前行第二列的值。

JDBC同样还有一个DatabaseMetaData类，可以通过它获得模式信息以及其他有关数据库

的信息。

10.5.5 状态处理

在JDBC中，状态处理使用Java的标准异常处理机制，其基本形式如下：

```
try {
     . . . code that might cause an exception goes here . . .
}
catch (SQLException e) {
     System.err.println("Bad things have happened:\n");
     System.err.println("Message: " + e.getMessage());
     System.err.println("SQLState: " + e.getSQLState());
     System.err.println("ErrorCode: " + e.getErrorCode());
};
```

实际上，在程序中，对executeQuery()，prepareStatement()等的调用都应该使用try这样的语句封装。

系统试图执行try子句中的语句，每个语句包含对Java或者JDBC对象方法的调用。方法在声明的时候可以指定如果在执行方法的时候发生错误，那么就**抛出**（thrown）一个或多个已命名的异常，该异常的名称表示发生的错误的类型。例如，在执行查询的时候如果DBMS返回一个访问错误，那么JDBC方法executeQuery()会抛出SQLException异常。当一条SQL语句的执行不成功或者不完全的时候会发生访问错误，更具体地说，返回的SQLSTATE的值是除成功之外的值。成功的时候返回00XXX，警告时返回01XXX，没有数据的时候返回02000。

如果在try子句中抛出异常，会被相应的catch子句捕获并执行。在例子中，一个SQLException对象e被创建，它的方法用于打印出错信息并返回SQLSTATE的值或者厂商特定的错误代码，当catch子句执行完毕后，程序继续执行后面的语句。

10.5.6 执行事务

默认情况下，当创建一个连接的时候数据库处于**自动提交模式**（autocommit mode）。每条SQL语句都被看作一个单独的事务，当语句执行完毕就被提交。为了将两条甚至多条语句放在一个事务里，可以将自动提交模式关闭：

```
con4.setAutoCommit(false);
```

其中，con4是一个Connection对象。

开始的时候，每个事务使用数据库管理器的默认隔离级别，通过下面的调用可以改变隔离级别：

```
con4.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
```

序列化级别TRANSACTION_SERIALIZABLE和TRANSACTION_REPEATABLE_READ等都是定义在Connection类中的常量（静态整数）。

提交或者终止事务使用Connection类的commit()方法和rollback()方法：

```
con4.commit();
con4.rollback();
```

当一个事务被提交或者回退后，下一个SQL语句执行时就会生成一个新的事务（程序的

第一条SQL语句执行也会生成一个事务)。这种建立事务的方法被称作链（chaining），将在21.3.1中进一步讨论这个问题。

如果程序连接到多个DBMS，那么在每个DBMS中的事务可以分别进行提交和回退。JDBC不支持确保事务集从全局上说是原子的提交协议。不过，可以使用一种新提议的Java包——JTS(Java Transaction Service)，它包括TP监视器和相应的API，以确保使用JDBC的分布式事务是原子提交的。TP监视器的内容将在第22章讨论。

10.5.7 服务器端的存储过程

如果DBMS支持这个功能，那么JDBC可以调用存储过程。例如，下面的程序调用图10-7定义的存储过程:

```
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();
```

其中，con5是一个Connection对象。

第一条语句声明CallableStatement对象cs5，并将存储过程Deregister()赋给它。结构{call Deregister(?,?,?,?,?)}外面的大括号表明该结构是SQL escape语法的一部分，使得驱动程序能使用特殊的方式处理括号里的代码。Deregister()的三个输入参数的值分别从Java变量crs_code、semester和stud_Id中获得，通过setXXX()方法指定（这里忽略输出参数和返回值的细节，它们的处理方式稍有不同）。最后一条语句执行调用。

除了更新数据库外，DBMS允许存储过程返回一个结果集。调用这样的存储过程被看作是一次查询，这时，上面的最后一条语句被替换为:

```
ResultSet rs5 = cs5.executeQuery();
```

JDBC也支持以字符串的形式创建一个存储过程并将它发送给DBMS。

10.5.8 示例

图10-12是一个调用JDBC API的Java程序，实现的功能和图10-3相似。有一点不同，在本例的SQL语句中使用的是常量，而图10-3使用的是占位符“?”。

10.5.9 SQLJ: Java的语句级接口

尽管调用级接口（如JDBC）可以在静态事务处理应用程序中使用（这时数据库模式和SQL语句的模式在编译的时候已经知道），但是运行时的效率要比语句级接口（如静态SQL）低，这是因为预处理和执行通常要与DBMS单独通信。正因为如此，各公司的联盟组织开发了Java的语句级SQL接口，叫做SQLJ，现在已经成为ANSI标准。SQLJ的一个重要目标就是获得与Java应用的嵌入式SQL相同的运行时效率，同时保留通过JDBC访问DBMS的优点。

SQLJ是嵌入式SQL的专用语言，可以包含在Java程序里。这些程序通过预编译器转换成

标准的Java，同时嵌入式SQLJ结构被替换成对SQLJ运行包的调用，这个包通过调用JDBC驱动程序访问数据库。SQLJ程序可以通过不同的JDBC驱动程序连接到不同的DBMS。和嵌入式SQL一样，预编译器会检查SQL语法和参数的数目和类型，以及结果的数目和类型。

```
import java.sql.*;
    ... ... ...
    try {
        // Use the right JDBC driver here
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    } catch (ClassNotFoundException e) {
        return(-1);  // Can't 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);  // Can't 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);  // Can't 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);  // Can't update
    }

    con6.commit();
    stat6.close();
    con6.close();
    return(0);  // Success!
```

图10-12　在Java程序中使用JDBC

这里不详细介绍SQLJ的语法，只是概括一下SQLJ、嵌入式SQL与JDBC之间的区别。

1) 对于嵌入式SQL来说，每个DBMS厂商支持自己的SQL，而SQLJ支持SQL-92，从而更

具可移植性。

2) 在Java程序中，SQL语句作为**SQLJ子句**的一部分，它是以#SQL开头，将SQL语句包含在大括号里，而不是像嵌入式SQL那样以EXEC SQL开头。例如，图10-1中的SELECT语句写成SQLJ时变成下面的形式:

```
#SQL {SELECT C.Enrollment
      INTO :numEnrolled
      FROM CLASS C
      WHERE C.CrsCode = :crsCode
        AND C.Semester = :semester};
```

3) 在静态SQL中，任何Java变量都可以作为SQL语句的参数，只要以“:”作为前缀即可。因为可以在编译时完成，所以这种参数传递方式在运行时比JDBC里的方法要有效的多。在JDBC里，每个参数的值在运行时必须与一个占位符“?”绑定。

4) 在SQLJ中，查询返回一个**SQLJ迭代器**（iterator）对象而不是ResultSet对象。SQLJ 迭代器与结果集相似，在结果集中提供游标机制[⊖]。事实上，SQLJ 迭代器对象和ResultSet对象实现了相同的Java接口java.util.Iterator。迭代器对象存储整个结果集，并且提供next()这样的方法扫描结果集中每行记录。例如，图10-13是图10-6的SQLJ版本。

```
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();
```

图10-13　在SQLJ中使用迭代器

程序的第一条语句告诉SQLJ预编译器生成Java语句，该语句定义一个GetEnrolledIter类，这个类实现sqlj.runtime.NamedIterator接口，它是标准Java接口java.util.Iterator的扩展，并提供next()方法。GetEnrolledIter类可以用来存储每行有两列的结果集，一列是整型，另一列是字符串型。声明将这两个字段命名为studentID和studGrade，同时隐式地定义了**列访问器**（column accessor）方法studentId()和studGrade()，它们用来返回存储在相应列中的值。

⊖ SQLJ迭代器可以转换成结果集，反之亦然。

第二条语句声明一个GetEnrolledIter类对象iter1。

第三条语句执行SELECT并将结果集存储在iter1中。注意，AS子句将SQL的属性名与迭代器中的列名联系在一起，这些名字不一定要相同，但是类型和数量上要一致。

最后，while语句将行的值存储在宿主变量id和grade中，以便处理。

5) SQLJ有自己的机制来定义连接对象和连接数据库，程序可以同时有几个活动的连接。与嵌入式SQL不同的是，每个SQLJ语句可以选择一个数据库连接，并将这个语句应用在那个连接上。例如，本节开头的第一个SELECT语句可以写成：

```
#SQL [db1] {SELECT C.Enrollment
            INTO :num_enrolled
            FROM CLASS C
            WHERE C.CrsCode = :crs_code
               AND C.Semester = :semester};
```

它指定将语句应用到先前定义的名为db1的数据库连接上。如果没有指定这个选项，那么将所有的SQL语句应用到默认的数据库连接上，这与嵌入式SQL相同。回忆一下，在JDBC中，每个SQL语句总是通过Statement对象显式地和一个指定的数据库连接关联在一起。

6) 正如静态与动态嵌入式SQL语句可以包含在相同的宿主语言程序中一样，SQLJ语句和JDBC调用也可以同时包含在同一个Java程序中。

10.6 ODBC*

开放式数据库互连（Open DataBase Connectivity，ODBC）是DBMS的API，提供执行SQL语句的调用级接口。下面的内容基于Microsoft开发的ODBC规范，但要注意，有些厂商并不支持所有的功能。

SQL标准组织长期以来一直致力于规范调用级接口（如SQL/CLI），以取代笨重的动态SQL，ODBC正是这种规范的早期版本的一个分支，它与最新发布的SQL/CLI有许多共同之处。SQL/CLI包括在SQL:1999中，Microsoft宣称ODBC会与这个新颁布的标准保持一致。

ODBC应用程序的软件体系结构和JDBC很相似，JDBC使用驱动程序管理器并且每个被访问的DBMS都会有一个独立的驱动程序。但是ODBC不是面向对象的，所以它提供对DBMS的较底层接口。例如，在ODBC里，应用必须专门分配和释放由驱动程序管理器和驱动程序使用的存储空间。而在JDBC里，当创建适当的对象的时候，存储空间会自动分配。因此，ODBC应用在调用函数SQLConnect()请求连接到数据库管理器前，必须先调用函数SQLAllocConnect()来请求驱动程序管理器分配连接所需的存储空间。而当应用调用SQLDisconnect()从数据库管理器断开之后，程序必须调用SQLFreeConnect()来释放存储空间。这样的操作使ODBC程序更容易发生**内存泄漏**（memory leak），它是因程序没有释放不再使用的ODBC结构而积累起来的垃圾内存。

图10-14显示了C程序中一些常用函数的调用方法[⊖]，每个函数返回一个值标识成功与否。

⊖ 这里以及以后的例子中我们都简化了语法，目的是强调ODBC交互的语义。例如在现实中，字符串参数（database_name）要跟一个长度的字段。

```
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);
```

图10-14　C程序中的ODBC过程调用框架

- SQLAllocEnv()在驱动程序管理器中为应用的ODBC接口分配和初始化存储空间。它返回一个标识句柄henv。**句柄**（handle）是一种应用程序用来引用该数据结构的机制。在C中，句柄即为指针，但是在其他宿主语言中，它的实现有些不太一样。ODBC驱动程序管理器内部使用环境域存放运行时信息。
- SQLAllocConnect()在驱动程序管理器中为连接分配内存，并返回连接句柄hdbc。
- SQLConnect()装载合适的数据库驱动程序，然后使用刚才分配的连接hdbc、数据库名称、用户Id和密码连接到DBMS服务器。

  如果应用想连接到多个数据库管理器，就要为每个管理器分别调用SQLAllocConnect()和SQLConnect()，驱动程序（可能有多个）为每个连接维护独立的事务。
- SQLAllocStmt()在驱动程序中为SQL语句分配存储空间，并返回这个语句的句柄hstms。
- SQLExecDirect()接受使用SQLAllocStmt()分配的语句句柄和一个含有SQL语句的字符串变量，请求DBMS预处理和执行语句。同一个句柄可以使用多次以执行不同的SQL语句。SQL语句可以是数据操纵语句（例如SELECT和UPDATE），也可以是DDL语句（例如CREATE和GRANT）。但是，它不能包含一个嵌入式的对宿主变量的引用，因为变量名只能在编译的时候转换成内存地址。注意，SQLExecDirect()与动态SQL中的EXECUTE IMMEDIATE相关联但是比它更通用，因为后者不能返回数据给应用（见10.4.1节），而前者可以使用SELECT生成一个结果集，并可以通过游标访问结果集（见10.6.2节）。
- SQLDisconnect()从服务器断开，这个函数使用连接句柄作为参数。
- SQLFreeStmt()、SQLFreeConnect()和SQLFreeEnv()释放句柄并释放由相应的Alloc函数分配的存储空间。

10.6.1　预处理语句

应用程序调用下面的函数，而不是调用SQLExecDirect()对语句预处理：

```
SQLPrepare(hstmt, ···SQL statement···);
```

然后调用

```
SQLExecute(hstmt);
```

执行这条语句。

和动态SQL一样，SQLPrepare()的语句参数可以包含占位符“?”。程序使用SQLBindParameters()函数提供变量给这些参数，例如，下面的调用[⊖]：

```
SQLBindParameters(hstmt, 1, SQL_PARAMETER_INPUT,
                  SQL_C_SSHORT, SQL_SMALLINT, &int1);
```

将语句hstmt的第一个参数和宿主语言变量int1绑定，hstmt是一个输入参数，而int1是C语言中的short类型。执行SQLBindParameters ()会将语句中的第一个占位符“?” 替换为这个参数的值，并将C中的short类型转换为SQL中的SMALLINT类型[⊜]。因为这个过程调用是宿主语言编译器编译的，所以对参数列表中宿主语言变量int1的引用可以在编译的时候处理。这与SQLExecDirect()不同，它不允许引用宿主变量。

10.6.2 游标

使用函数SQLExecDirect()或者SQLExecute()执行SQL语句不会给应用返回任何数据，而是通过游标返回数据，它由ODBC驱动程序维护并被SQLAllocStmt()函数返回的语句句柄hstmt引用。为了获取数据还必须调用相应的ODBC函数。

程序可以调用SQLBindCol()将结果集里的某一个列与程序里的某个宿主变量绑定起来。例如，下面的调用[⊝]将第一个列与整型变量int2绑定在一起：

```
SQLBindCol(hstmt, 1, SQL_C_SSHORT, &int2);
```

调用SQLFetch(hstmt)时，游标向前移动，当前行里列的值存储在与其绑定的变量里。如果某个列没有被绑定到指定的宿主变量，程序可以调用SQLGetData()检索和存储该列的值。例如[⊞]，下面的调用将当前行里第二列的值存储在整型变量int3中：

```
SQLGetData(hstmt, 2, SQL_C_SSHORT, &int3);
```

在执行SELECT语句之前，应用程序可以使用SQLSetStmtOption()指定游标的类型，共有三种类型可以选择：

1) STATIC。与嵌入式SQL中的游标类型INSENSITIVE一样，当执行SELECT语句的时候，驱动程序建立结果集的行的一个拷贝。游标访问的是这个拷贝，如图10-5所示。这种返回的数据类型称作**快照**（snapshot）。

2) KEYSET_DRIVEN。当执行SELECT语句的时候，驱动程序建立一组指针指向基表中满足WHERE子句的行。如果当前事务或者同步事务中的其他语句更新或者删除其中的某一行，这个变化会在下一次调用SQLFetch()时体现出来。但是，如果是插入一行，虽然它也满足WHERE子句，该变化却不能在下一次调用SQLFetch()时体现出来，这是因为指针指向的行没有位于集合中。另外，可以改变行里某个属性的值从而不再满足WHERE子句，但是仍能够通过游标访问。这种返回数据的类型称作**动态集**（dynaset），见图10-15。

⊖ 这里简化了语法，实际上SQLBindParameters()的参数不少于10个。

⊜ 与此相同，在许多ODBC结构中，程序员需要指明DBMS中的SQL数据类型与应用程序中的C语言数据类型之间的转化。

⊝ 这里简化了语法。

⊞ 这里简化了语法。

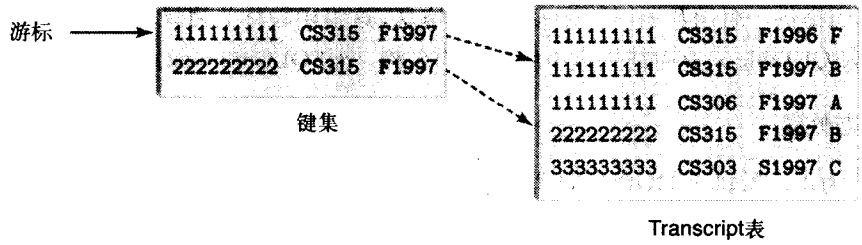


图10-15　使用KEYSET_DRIVEN游标检索TRANSCRIPT中的记录

3) DYNAMIC。结果集中的数据是完全动态的。执行完SELECT语句后，当前事务或者并发事务中的语句不仅可以修改和删除结果集中的行，而且还可以插入一行记录，这些变化都可以在下一次调用SQLFetch()时体现出来。

ODBC规范建议实现所有类型的游标。但是，ODBC可以得到的某个DBMS的机制可能会使实现某种类型的游标很困难，因此这个DBMS的驱动程序只能支持其中的一部分游标类型。很明显，DYNAMIC游标实现起来最困难，许多驱动程序并没有实现这种类型的游标。

语句SQLSetStmtOption可以用来请求游标的类型，形式如下:

```
SQLSetStmtOption(hstmt,SQL_CURSOR_TYPE, Option);
```

其中，*Option*是SQL_CURSOR_STATIC、SQL_CURSOR_DYNAMIC 或者SQL_CURSOR_KEYSET_DRIVEN中的一个常量。

ODBC也支持通过非静态的游标更新指定的位置，这时，SELECT语句必须用FOR UPDATE加以定义。例如:

```
SQLExecDirect(hstmt1,  "SELECT * \
                        FROM EMPLOYEE \
                        FOR UPDATE OF Salary");
```

它使用已经分配的语句句柄hstmt1预处理一个查询，该查询的游标允许通过Salary属性更新EMPLOYEE关系。

假设执行若干次SQLFetch(hstmt1)后游标位于employee属性值是“Joe Public”的这一行，现在要将Joe的薪水增加100美元，可以使用下面的语句:

```
SQLExecDirect(hstmt2,  "UPDATE EMPLOYEE \
                        SET Salary = Salary + 1000\
                        WHERE CURRENT OF employee_cursor");
```

其中hstmt2是以前分配的一个语句句柄。上面语句的一个问题是，WHERE CURRENT OF子句中的游标名称employee_cursor可能是无效的，因为它并没有同与hstmt1句柄相关的游标建立联系，因此执行上面的更新操作前必须给游标起一个名字。ODBC使用下面的方法指定名字:

```
SQLSetCursorName(hstmt1, employee_cursor);
```

和动态SQL一样，在编写程序的时候并不知道查询的结果集的信息，因此ODBC提供一组函数来获得这些信息。例如，预处理一条语句后，程序可以调用函数SQLNumResultCols()获得结果集中列的个数，而函数SQLColAttributes()和SQLDescribeCol()可以提供结果集中指定列的相关信息。

ODBC还提供一组函数返回数据库模式的信息，这组函数称为**目录函数**（catalog function）。例如，函数SQLTables()返回结果集中所有表的名称，而SQLColumns()返回字段的名称。

10.6.3 状态处理

到目前为止，所讨论的ODBC过程实际上都返回一个类型为RETCODE的值，表示指定的动作是否成功。例如，下面的示例使用这个值判断成功与否：

```
RETCODE retcode1;
...
retcode1 = SQLConnect(...);
if (retcode1 != SQL_SUCCESS)   {
     ... do something ...
}
```

关于错误的额外信息可以通过调用SQLError()获得。

10.6.4 执行事务

默认情况下，数据库在创建一个连接时处于自动提交模式。为了在一个事务中执行多条语句，可以通过下面的函数关闭自动提交模式：

```
SQLSetConnectionOption(hdbc,
                       SQL_AUTOCOMMIT,
                       SQL_AUTOCOMMIT_OFF);
```

其中，hdbc是一个连接句柄。初始状态下，每个事务使用数据库管理器默认的隔离级别，可以调用下面的方法改变级别：

```
SQLSetConnectionOption(hdbc,
                       SQL_TXN_ISOLATION,
                       SQL_TXN_REPEATABLE_READ);
```

使用下面的方法可以提交或者回退事务：

```
SQLTransact(henv, hdbc, Action);
```

这里，Action是SQL_COMMIT或者SQL_ABORT。

当事务提交或者回退以后，执行下一条SQL语句会开始一个新的事务（或者程序的第一条SQL语句执行的时候也会开始一个新事务）。

如果程序连接到多个数据库，每个数据库上的事务是独立提交和回退的。ODBC不支持确保事务集是全局原子的提交协议。不过，Microsoft引进了新的TP监控器——MTS（Microsoft Transaction Server），它包括一个事务管理器和一组API来保证使用ODBC的分布式事务是原子提交的。

10.6.5 服务器端的存储过程

如果DBMS支持这个特性，那么可以使用ODBC调用存储过程。例如，为了调用图10-7的存储过程，可以先用以下方法预处理调用语句：

```
SQLPrepare(hstmt, "{call Deregister(?,?,?,?,?)}");
```

Deregister ()外面的大括号表示转义语法，参见10.5.7节。Deregister()的参数可以通过SQLBindParameter()函数与宿主语言变量绑定。执行调用过程如下:

```
SQLExecute(hstmt);
```

和嵌入式SQL一样，如果DBMS允许存储过程返回一个结果集，那么应用程序可以使用游标从结果集检索数据。

10.6.6 示例

图10-16是一个包含ODBC过程调用的C语言程序，它执行的事务与图10-12相同。作为过程SQLConnect()和SQLExecDirect()中的参数的常量SQL_NTS表示一个以null结尾的空字符串。

```
HENV henv;
HDBC hdbc;
HSTMT hstmt;
RETCODE retcode;
SQLAllocEnv(&henv);
SQLAllocConnect(henv, &hdbc);
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, hdbc, 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, hdbc, SQL_ABORT);
     SQLFreeStmt(hstmt, SQL_DROP);
     SQLDisconnect(hdbc);
     SQLFreeConnect(hdbc);
     SQLFreeEnv(henv);
     return(-3);
}
SQLTransact(henv, hdbc, SQL_COMMIT);
SQLFreeStmt(hstmt, SQL_DROP);
SQLDisconnect(hdbc);
SQLFreeConnect(hdbc);
SQLFreeEnv(henv);
```

图10-16　用C编写的ODBC程序

10.7 比较

到目前为止，我们已经讨论了访问数据库的不同方法，每种方法都有自己的优点和缺点，下面对此加以总结。

- 在某些情况下（如静态SQL和SQLJ中），SQL语句使用特殊的语法。而在动态SQL、JDBC和ODBC中，SQL语句是作为变量的值。前者的优点是简单，但是与数据库的交互只能限定在编译的时候。在某些应用中，执行的语句要到运行时才能确定，这时候动态生成SQL语句是十分重要的。
- 有时候，应用程序必须使用所连接的DBMS的专用SQL，这使得将程序移植到另一个DBMS变得十分困难。而在SQLJ、JDBC和ODBC中，应用程序使用统一的专用语言，并提供模块将它转换到特定厂商的专用语言，因此提高了可移植性，但是这样的程序就不能使用某个厂商提供的特殊功能。
- 评估各种技术在运行时的开销涉及许多因素，如通信、预处理以及参数传递等等。虽然有时候开销与技术的实现方式有关，但是仍然可以大致总结如下。

对于语句级接口，SQL语句包含嵌入式参数的名称，在编译的时候可以处理该名称并生成参数传递代码。运行时，语句被传递到服务器进行预处理和执行，因此一次通信就足够了。对于调用级接口和动态SQL，参数名称不包含在语句中，因为在编译的时候（通过符号表可以将参数名称映射成地址）语句是得不到的。相反，如果它们是分别提供的，那么就可以在编译的时候进行处理。因此需要一次通信来发送预处理的语句，而另外一次通信请求执行，不过EXECUTE IMMEDIATE例外。这是非常重要的，因为通信的开销昂贵而且耗费时间。

如果SQL语句是动态生成的（动态SQL、JDBC和ODBC），那么预处理必须在运行的时候进行。但即使是静态SQL，预处理通常也是在语句被提交执行的时候进行的。在所有的情况下，如果语句执行多次，预处理的开销可以分摊到每次执行中去，因而不再是一个主要因素。最有效的避免运行时预处理操作的方法是使用存储过程，它可以让DBMS在程序执行前创建和存储一个查询执行计划。通常，存储过程为过程中的每个SQL语句单独生成一个计划，更为成熟的系统是将它看成一个整体生成一个优化计划，因为已知SQL语句的顺序。为一个语句生成的数据结构可以保留下来供下一条语句使用。

10.8 参考书目

嵌入式和动态SQL可以追溯到很早以前，任何SQL手册都或多或少地包含一些相关内容。下面的参考书值得一看：[Date and Darwen 1997, Melton and Simon 1992, Gulutzan and Pelzer 1999]。

介绍ODBC的文献很多，Microsoft出版了一本权威指南[Microsoft 1997]。此外，还可以参考[Signore et al. 1995]。[Venkatrao and Pizzo 1995] 讨论了SQL/CLI的历史和原理，这是一个与ODBC相似的新的SQL标准。SQL存储过程可以参阅[Eisenberg 1996, Melton 1997]中的介绍。有关JDBC和SQLJ的信息可以很容易地从相关网页[Sun 2000]和[SQLJ2000]获得，但是[Reese 2000, Melton et al. 2000]是学习这些技术的更好选择。

10.9 练习

10.1 解释你机器上的应用程序器使用什么方法来完成下面的工作:

a. 在宿主语言中包含SQL语句。

b. 创建数据库表。

c. 连接和断开数据库。

d. 创建、提交和中止事务。

e. 指定隔离级别。

f. 改变约束检查的模式。

g. 授予用户在某个表上执行SELECT语句的权限。

10.2 a. 为什么嵌入式SQL和SQLJ需要预编译器，而ODBC和JDBC不需要预编译器?

b. 嵌入式SQL的预编译器将SQL语句转换成过程调用，而调用级接口（如ODBC和JDBC）是将SQL语句作为过程调用的参数，两者的区别是什么?

10.3 a. 为什么约束检查通常在事务处理应用中被推迟进行?

b. 举例说明在事务处理应用中不希望进行即时约束检查的情况。

10.4 说明在事务处理应用中使用存储过程的优点和缺点。

10.5 写出使用下面的方法实现的程序:

a. 嵌入式SQL

b. JDBC

c. ODBC

d. SQLJ

这个程序实现学生注册系统中的注册事务，请参考图5-15和5-16中的数据库模式。

10.6 写出使用下面的方法实现的程序:

a. 嵌入式SQL

b. JDBC

c. ODBC

d. SQLJ

这个程序使用游标打印出学生注册系统中某个学生成绩单，请参考图5-15和5-16中的数据库模式。

10.7 a. 为什么嵌入式SQL和SQLJ使用宿主语言变量作为参数，而动态SQL、JDBC和ODBC使用占位符“?”?

b. 为什么在嵌入式SQL中使用宿主语言变量作为参数比使用占位符“?”要好?

10.8 解释动态SQL与JDBC和ODBC相比的优点和缺点。

10.9 写出使用下面的方法实现的程序:

a. 嵌入式SQL

b. JDBC

c. ODBC

d. SQLJ

这个程序在不同的DBMS中传递表中的记录。你的事务具有全局原子性吗?

10.10 编写一个在本地浏览器中运行的Java程序，使用JDBC连接到本地机上的DBMS，并对创建好的表执行一个简单查询。

10.11 假设在编译的时候，程序员已知要执行的SQL语句、DBMS和数据库模式的所有细节，比较使用

嵌入式SQL与使用JDBC和ODBC相比的优点和缺点。

10.12　10.6节讨论了KEYSET_DRIVEN类型的游标。

a. STATIC和KEYSET_DRIVEN游标之间有何不同？

b. 举例说明使用这些游标可以返回不同的结果，即使事务是独立执行的。

c. 为什么更新和删除操作可以通过KEYSET_DRIVEN类型的游标来完成？

10.13　写出一个包含游标的事务程序，程序中FETCH语句返回的结果与游标是否被定义为INSENSITIVE有关。假设这个事务仅仅是执行，不必关心并发事务可能带来的影响。

10.14　比较嵌入式SQL、SQL/PSM、JDBC、SQLJ和ODBC状态处理（异常处理）机制的优点和缺点。

第11章 数据的物理组织和索引

SQL最大的优势在于它是一种声明性的语言，一个SQL语句描述的是关于存储在数据库中信息的一个查询，但没有定义系统执行该查询所使用的技术，具体使用何种技术由数据库管理系统自身来决定。

这样的技术与**存储结构**（storage structure）、**索引**（index）和**访问路径**（access path）紧密相关。存储结构是定义一个文件里表中行的特定组织形式。索引是附属的数据结构，可能是单独存储的一个文件，它支持对表中行的快速访问。访问路径是指访问一个行集合的特殊技术。它使用一种基于表存储结构的算法，或基于所选择的表上可用的索引。

关系模型的一个重要特性是，SQL语句的执行结果不会受到执行时使用的访问路径的影响（也就是说，查询是作用在数据库上的，由数据库返回查询结果的信息），所以，在设计查询时，程序员不必关心系统的访问路径。

尽管访问路径的选择不会影响查询结果，但它对性能有重要的影响。根据所使用的访问路径，执行时间可能从几秒到几个小时不等，特别是查询涉及到大量的表，而这些表中有几千行，或者成千上万行时更是如此。而且，对同一个表的查询，不同的访问路径适合不同的SQL语句。

因为执行时间对访问路径敏感，所以大多数数据库系统允许数据库设计师和系统管理员自行确定每一个表的访问路径。一般来说，管理员会根据所观察到的各种SQL语句的执行频率来决定。

在这一章中，我们描述多种访问路径，通过不同的SQL语句的执行来比较它们的操作性能。

11.1 磁盘组织

由于多方面的原因，数据库通常存储在大容量的存储设备上，而不是存储在主存中。

容量（size） 描述大型企业的数据库通常包含海量信息，G数量级字节的数据库是不足为奇的，目前已存在许多T数量级字节的数据库。计算机的主存容纳不下如此规模的数据库，而且在将来也是不可能的。

代价（cost） 即使数据库能放在主存中（例如，当数据库的大小是兆数量级字节时），通常还是把数据库存放在大容量的存储设备上[⊖]，理由是经济方面的原因，主存上每字节的代价比磁盘上每字节的代价要高100倍。

易失性（volatility） 在第2章，我们介绍了持久性的概念，这是与事务的ACID性质相关的。但持久性不仅仅限于事务，它也是数据库系统的一个重要性质，即使在系统故障的情形

⊖ 由于应用要求快速的响应时间，数据库通常是存储在主存中的，我们把这样的系统称为主存数据库系统（main memory database system）。

下、描述企业的信息也必须持久地保存下来。但是，大多数主存中的信息是易失的，即在断电和系统崩溃时，主存中的信息就会丢失。所以，本质上来讲，主存不能满足数据库系统的要求。而另一方面，大容量存储器中的信息是**非易失的**（nonvolatile），因为在系统故障时，信息仍然能保留下来。所以，大容量存储器组是大多数数据库系统的基础。我们将在第25章予以详细介绍。

因为存储在大容量存储器上的数据是不能被系统处理器直接访问的。要访问数据库中的数据，首先必须从大容量存储器上读数据到主存中的缓冲区中（此时，两种存储设备上都有这一数据的拷贝），然后对缓存中的数据进行操作。如果访问涉及对数据的修改，则大容量存储器上数据的拷贝就被废弃，必须将缓存中的数据拷贝写回存储器。

最常用的大容量存储器是磁盘。由于提高一个数据库系统的性能是以磁盘的物理特性为基础的，所以为理解性能问题，有必要回顾一下磁盘的工作方式。

一个磁盘单元包含一个或多个圆形**盘片**（platter），这些盘片固定在一个旋转的轴上，如图11-1所示。每个盘片的一面或者两面涂有磁性材料，一个点的磁场方向决定该点记录的是0还是1。因为信息是靠磁来存储的，所以在断电的情况下信息也不会丢失。

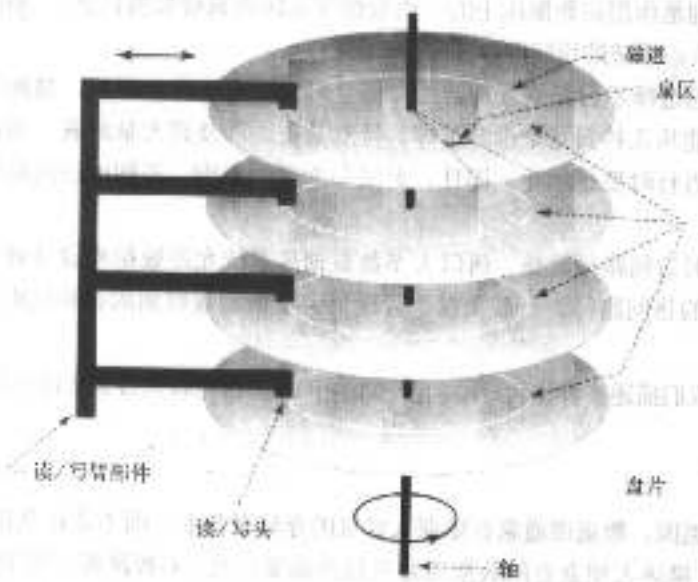


图11-1　一个磁盘存储单元的物理结构

每一个盘片（或表面）通过与之相连的**读/写头**（read/write head）来访问，读/写头能检测和设置它所处位置下方一点的磁场方向。读/写头与一个读/写臂相连，读/写臂沿半径方向移动，它既可以移向盘片的圆心，也可以移向盘片的边缘。由于盘片可以旋转，所以读/写头可以定位于盘面上方的任意一点。

事实上，数据是存储在**磁道**（track）上的，磁道是盘片上的同心圆，每一个盘片有相同的磁道数N，N是固定不变的，所有盘片上的第*i*个磁道组成的存储区域称为第*i*个**柱面**（cylinder）。因为每个盘片上都有一个读/写头，所以磁盘单元作为一个整体有一组读/写头，读/写臂部件一致地移动所有的读/写头。所以，在某个给定时间，所有读/写头定位于一个

特定柱面的上方，看起来好像能同时读写当前柱面的所有磁道。但是，通常由于机械方面的限制，在某一个时刻只允许有一个读/写头处于活动状态。最后，每一个磁道被分为多个**扇区**(sector)，扇区是能被硬件传输的最小单元。

在访问一个特定的扇区之前，读/写头必须定位于扇区起始处的上方，因此，访问一个扇区S的时间可分为三部分:

寻道时间（seek time） 读/写臂部件定位于包含扇区S的柱面上方所用的时间。

旋转延迟（rotational latency） 在读/写臂部件处于柱面的上方后，盘片必须旋转一定的角度，使得读/写头定位于扇区S开始处的上方。

传输时间（transfer time） 盘片旋转经过扇区S所花费的时间。

寻道时间取决于沿半径方向的距离，这个距离是当前读/写头所在的柱面到读/写扇区所在柱面之间的距离。在最坏的情况下，读/写头必须从第1道移动到第N道；在最好的情况下，根本不需要移动。如果磁盘请求均匀地分布在柱面上，按它们到达的先后顺序进行服务，那么可以证明，寻道平均要跨越的磁道数大约是$N/3$[⊖]。寻道时间通常是这三种时间中最长的，因为它不仅涉及机械移动，而且还包括克服读/写臂部件在启动和停止时克服惯性所用的时间。

旋转延迟仅次于寻道时间。平均来说，旋转延迟是绕盘片旋转一周所用时间的一半。其中同样也包括机械移动，不过在这种情形下，惯性不是问题。传输时间受旋转时间的限制。一个磁盘所支持的传输率是在某处读/写头一次能够传输数据的速率，这取决于盘片的旋转速度和盘片表面的位密度。术语**延迟**（latency）是指定位读/写头和访问盘片所用时间的总和。所以延迟是寻道和旋转延迟时间之和。

一个典型的磁盘能存储G数量级字节的数据，磁盘中一个扇区的大小为512B，平均寻道时间是5 ~ 20ms，平均旋转延迟是2 ~ 10ms，每秒的传输率是几兆字节，所以通常情况下，访问一个扇区的时间是20毫秒级。

磁盘的物理特性决定两个扇区之间的距离，它是度量访问延迟的尺度。如果相同磁道上的两个扇区相邻，此时扇区之间的距离最小，因为如果连续地读这两个扇区，则时间延迟为零。按距离增加的顺序，在同一柱面，同一磁道上的两个扇区相距最近。对不同的柱面，如果$|n_1-n_2|<|n_3-n_4|$，则磁道n_1和n_2上的扇区之间的距离比磁道n_3和n_4上扇区之间的距离小。

从一个磁盘单元的物理特性，我们可以得出下列结论:

- 最重要的是，和CPU相比，磁盘是一种低速设备。CPU可以利用访问一个扇区的时间执行成百上千条指令。因此，在试图优化一个数据库系统的性能时，有必要优化主存和磁盘之间的信息流。与优化磁盘流量所获得的性能相比，数据库使用有效的算法处理数据所提高的性能是微不足道的。认识到这一点后，当在后续章节讨论访问路径时，我们通过估计I/O操作来评价性能，而忽略处理器所用的时间。
- 在访问记录A_1后很可能访问记录A_2的实际应用中，如果存储A_1的扇区和A_2的扇区的距离越小，则时间延迟就越小。所以，应用程序的性能受磁盘上数据的物理存储方式的影响。例如，如果一个表存储在一个磁道上或一个柱面上，那么对这个表进行顺序扫描就能高效地完成。

⊖ 该问题用算术方式可描述为：假设有一间隔，在间隔内任意地放置两个标记，这两个标记平均相距多远?

- 为简化缓冲区的管理，数据库系统的每次I/O操作传输相同的字节数。那么每次传输多少字节呢？根据以前所记录的数字，和平均延迟相比，传输一个扇区所花的时间明显地要小得多，即使每次I/O操作传输多个扇区，这个结论也是正确的。尽管从物理的角度来讲，一般的传输单位是处在一个扇区上的数据，但如果系统能够传输更大单元上的数据，这可能会更加有效。

  页（page）通常指每次I/O操作所传输的数据单位，页在磁盘上是以一个**磁盘块**（简称为块）的形式存储的，一个块是一个磁道上顺序存储的多个相邻扇区，页的大小和这个块的大小相等。一次I/O操作能传输一页，只需一次时间延迟来将读/写头定位于包含该块的起始处，因为在这里没有必要将读/写臂或盘片从块的一个扇区移动到下一个扇区来重新定位。

  这时的折中涉及到选择块中的扇区数（即页的大小）。当一个特定的应用访问表中记录A时，应用很可能会访问表中与A相邻的另一条记录（如果表被经常扫描，那么实际情况可能就是这样），因此页应足够大，以便能容得下这些额外的记录从而避免对磁盘的另一次访问。另一方面，随着页面的增大，传输时间会增加，大的页面要求内存中有大的缓冲区。因此，太大的页面也是不可取的，因为这可能大大地超出与A相邻的实际要访问的信息。综合以上的考虑，一个典型页面的大小是4096B（4KB）。注意，我们要讨论的是减少延迟和寻道时间。在这种情况下，这样大小的页是合适的，它能将相关的信息包含在相同的页和连续的扇区上（它们之间的距离为0）。

在主存中，系统维护着一组页面大小的缓冲区，称为**高速缓存**（cache）。当应用请求访问表的一个特定记录A_1时，对页进行I/O操作，会有下列事件要发生。我们假设每一个表存储在一个**数据文件**（data file）中。首先，数据库系统检测文件，确定哪一页包含记录A_1，然后将这一页从大容量的存储设备传输到高速缓存的一个缓冲区中（我们立即对这部分描述进行修改），在后来的某个时间（通常是由大容量存储设备的中断信号来标识），系统确认传输已经完成，就将A_1从缓冲区的页拷贝到应用程序中。

由于几个方面的原因，系统试图将页在缓冲区中保存一段时间。第一，应用可能以后要对记录A_1进行修改，这样缓冲区中A_1的拷贝必须进行相应的修改（缓冲区中的其他记录保持不变），缓冲区的内容要用来修改大容量存储设备上的页。

第二，应用（或其他某个应用）可能请求访问这个表中的另外一条记录A_2。如果A_2和A_1处在相同的页，则系统无需额外的I/O操作就能返回一个拷贝（明显的成功之处）。数据库系统在执行I/O操作之前，总是先搜索高速缓存来试图满足应用的数据库访问请求（这就是我们前面所说的修改）。

当在高速缓存中发现请求的页时，这个事件称为一次**高速缓存命中**（cache hit），通过最大化高速缓存命中率，有可能极大地提高系统的性能。因此，应该关注高速缓存中缓冲区（有限数量的缓冲区）的管理。当高速缓存满（因为高速缓存早晚一定会满），必须提取新页时，应该清空高速缓存中的哪个缓冲区呢？解决此问题常用的一个算法是**最近最少使用算法**（Least Recently Used，LRU），它清空最长时间没被引用的页面所在的缓冲区。这一设想是基于最近被引用的页是活动的，它在不久的将来被再次引用的概率很高，所以这个页应保存在高速缓存中。

11.2 堆文件

我们假设每一个表按照某种存储结构存储在不同的文件中，其中最简单的存储结构是**堆文件**（heap file）。使用堆文件，新产生的行能有效地追加在文件的尾部。因此，文件中行的顺序是任意的。没有专门的规则来限制特定的行应放在什么位置。图11-2是表TRANSCRIPT以堆文件方式存储的物理图表示。这个表共有四列：StudId、CrsCode、Semester和Grade。我们已经修改了表的模式，以便Grade能存储实数，我们还假设一个页面能容纳四行。

666666666	MGT123	F1994	4.0	
123454321	CS305	S1996	4.0	第0页
987654321	CS305	F1995	2.0	
111111111	MGT123	F1997	3.0	
123454321	CS315	S1997	4.0	
666666666	EE101	S1991	3.0	第1页
123454321	MAT123	S1996	2.0	
234567890	EE101	F1995	3.0	
234567890	CS305	S1996	4.0	
111111111	EE101	F1997	4.0	第2页
111111111	MAT123	F1997	3.0	
987654321	MGT123	F1994	3.0	
425360777	CS305	S1996	3.0	
666666666	MAT123	F1997	3.0	第3页

图11-2 以堆文件方式存储的TRANSCRIPT表

迄今为止，我们所关心的是堆文件的行是无序的，这是堆文件最重要的特征。我们忽略一些重要的问题，因为它们是数据库系统内部机制的问题，程序员通常无法控制这些问题。但我们必须知道这些是什么样的问题。例如，我们假设在图11-2中，所有的行有相同的长度，所以每一个页面容纳的行数相同。但事实却不一定是这样。例如，如果与某列相关的域是VARCHAR(3000)，用于存储这列中的字节数从1～3000不等。因此，一页所能容纳的行数是不定的，这使得存储分配变得十分复杂。通常，不论行是定长还是不定长，每一个页面必须格式化，使之包含一定数量的头信息，头信息指明页面中每一行的起始点和页面中未被使用的区域。我们假设一个数据文件中一行的逻辑地址是由行的ID(rid)给出的，这个ID包含这行在该文件中的**页数**（page number）和**行数**（slot number）。行的实际位置是通过解析使用页的头信息的行数信息来确定的。当一行太长，一个页面容纳不下时，这会变得更加复杂。

堆文件的优点是它的简单性，行插入后直接追加在文件的尾部，行删除是通过将它所在页面的头信息定义为空来完成的。把图11-2中的ID为111111111的学生删除，ID为666666666的学生在1998年的春季修完CS305后的记录插入就得到图11-3。注意，删除后在文件中留下了空隙，最终会浪费很多的存储空间。由于要检查很多不必要的页，这些空隙使得文件的搜索时间变长。最终这些空隙会扩展到必须压缩文件的地步，当文件很大时，压缩是一个很费时的过程，因为不仅必须对每一页进行读操作，还必须对所压缩的页面进行写操作。

666666666	MGT123	F1994	4.0	
123454321	CS305	S1996	4.0	第0页
987654321	CS305	F1995	2.0	
123454321	CS315	S1997	4.0	
666666666	EE101	S1991	3.0	第1页
123454321	MAT123	S1996	2.0	
234567890	EE101	F1995	3.0	
234567890	CS305	S1996	4.0	
				第2页
987654321	MGT123	F1994	3.0	
425360777	CS305	S1996	3.0	
666666666	MAT123	F1997	3.0	第3页
666666666	CS305	S1998	3.0	

图11-3　在图11-2中插入和删除一些行后的TRANSCRIPT表

通过计算完成其他文件操作所需的I/O操作数，我们可以比较堆文件和其他存储结构文件的访问效率，这是我们后面要讨论的内容。用F表示一个文件中的页数。首先来看插入操作。在插入一行A时，我们必须确保A的键不会与表中已有行的键相同。如果存在相同的键，可能平均要读F/2个页面，然后放弃插入操作。为得出没有相同键值存在的结论，必须读整个文件，然后重写最后一页（插入A）。在这种情况下，总的代价是传输F+1个页面。

删除的情形与此类似。对一个有特定键值的元组A，平均要读F/2个页面才能找到它，然后重写这个页面（A被删除），代价是F/2+1。如果不存在这样的元组，代价将是F。如果指定被删除的元组的条件不涉及键，就必须扫描整个文件，因为在表中可能存在多个满足条件的行。

如果对表的查询涉及访问所有的行，且行的访问顺序并不重要，那么堆文件是一种有效的存储结构。例如，查询

```
SELECT    *
FROM      TRANSCRIPT
```

返回整个表。对一个堆存储结构，代价是F，显然这是一种理想的状态。当然，如果我们想将行按某种顺序打印出来，代价会更大。因为在输出之前，必须先将行排序。这样的一个查询例子是:

```
SELECT    *
FROM      TRANSCRIPT T
ORDER BY T.StudId
```

作为另外的一个例子，考虑下面的查询:

```
SELECT    AVG(T.Grade)
FROM      TRANSCRIPT T
```

它返回所有指定课程的平均成绩。不管表是如何存储的，都必须读取表中所有行来提取成绩值。在读取表中的行时，平均数的计算与顺序无关。其代价为F，这也是最优的。

假设一个查询要求找出ID为234567890，在1996年春季所修课程为CS305的学生的成绩：

```
SELECT   T.Grade
FROM     TRANSCRIPT T
WHERE    T.StudId = '234567890' AND
         T.CrsCode = 'CS305' AND T.Semester = 'S1996'
```
(11.1)

因为{StudId,CrsCode,Semester}是表TRANSCRIPT的键，所以最多只返回一个成绩。如果表TRANSCRIPT的值如图11-2所示，那么将返回一个确切的成绩（namely,4.0）。数据库系统必须扫描文件的页面来查找具有特定键的行，并返回相应的成绩，在这种情况之下，必须读取三个页面。一般情况下，必须平均读取F/2个页面，但如果有特定键的行不在表中，就必须读取所有F个页面。不论哪一种情形，考虑到实际请求访问的信息量，其代价都是很高的。

最后，考虑下面的查询，第一个查询返回ID为234567890的学生的课程、学期和所有课程的成绩，第二个查询返回所有某些课程的成绩位于2.0～4.0的学生的ID。因为在这两种情况之下，WHERE子句都没有指定一个候选键，所以可能返回任意多个行。因此，整个表的搜索代价是F。

```
SELECT   T.Course, T.Semester, T.Grade
FROM     TRANSCRIPT T
WHERE    T.StudId = '234567890'
```
(11.2)

```
SELECT   T.StudId
FROM     TRANSCRIPT
WHERE    T.Grade BETWEEN 2.0 AND 4.0
```
(11.3)

在语句（11.1）和（11.2）中，WHERE子句中的条件是相等条件，因为所请求的元组必须有特定的属性值。查找满足相等条件元组被称为**等值查找**（equality search）。语句（11.3）中的WHERE子句的条件是一个**范围条件**（range condition），它涉及一个属性，属性的域是有序的，所有属性值在特定范围的行都被检索出来，在条件范围内，并没有指定元组实际的属性值，一个满足范围条件的查询称为**范围查找**（range search）。

11.3 排序文件

最后的两个例子（11.2和11.3）说明了堆文件的缺点。即使请求访问的信息是一个行子集的情况下，也必须查找整个表。因为不扫描一个页，我们就不能确定它是否包含结果子集的一行。这些例子提出开发更好的存储结构的需求，其中之一就是本节我们所要讨论的结构。

假设我们不是在表中以任意的顺序存储各行，而是按表中某些属性上的值进行排序，我们称这样的结构为**排序文件**（sorted file）。例如，我们可以将表TRANSCRIPT存储在一个索引文件中，表中的各行是按StudId排序的，如图11-4所示。为查找满足（11.2）的条件的数据记录，一个简单的方法就是顺序扫描记录，直到找到第一个StudId为234567890的记录为止。所有包含这样值的记录是连续存储的，所以访问它们所需的I/O开销最小。特别地，如果连续的页在磁盘上邻近地存放，则可以使寻道时间最小化。在图中，描述学生234567890所修课程的行存储在一个页面上。一旦这些行的第一个记录在高速缓存中可用，那么所有其他满足查询条件

(11.2) 的记录就能被迅速检索出来。如果数据文件包含F个页面，为定位记录，平均要对F/2个页面进行I/O操作。和存储在堆文件中的表TRANSCRIPT相比，性能得到有效的改善。注意，同样的方法也适用于查询 (11.1)。

111111111	MGT123	F1997	3.0	第0页
111111111	EE101	F1997	4.0	
111111111	MAT123	F1997	3.0	
123454321	CS305	S1996	4.0	第1页
123454321	CS315	S1997	4.0	
123454321	MAT123	S1996	2.0	
234567890	EE101	F1995	3.0	第2页
234567890	CS305	S1996	4.0	
425360777	CS305	S1996	3.0	
666666666	MGT123	F1994	4.0	第3页
666666666	MAT123	F1997	3.0	
666666666	EE101	S1991	3.0	
987654321	MGT123	F1994	3.0	第4页
987654321	CS305	F1995	2.0	

图11-4　作为索引文件存储的TRANSCRIPT表

在某些情况下，使用**二分查找** (binary search) 技术会更加有效。首先检索处在中间的页，把它的Id值和234567890相比较。如果目标值在这个页面中，则查找成功。如果目标值小于（或者大于）页面的Id值，则在这个数据文件的前一半上（或后一半上）递归地重复这一过程。使用这一方法，在最坏的情况下，定位一个特定学生的Id需传输的页面数大约是$\log_2 F$。

传输的页面数并不一定能反映查找一个排序文件的实际开销。而使用二分查找法，我们可能要访问磁盘中不同柱面上的页，这可能会产生相当大的寻道延迟开销。实际上，在某些情况下，寻道延迟可能是主要的开销。考虑一个占用N个连续柱面的索引文件。其中，连续的页存储在相邻的块中，利用二分查找法可能先让磁盘头移动的距离是N/2个柱面，然后是N/4个柱面，接着是N/8个柱面，依此类推。磁盘头跨越的总的柱面数大约是N个，所以二分查找法总的开销将是:

$$N \times \text{寻道时间} + \log_2 F \times \text{传输时间}$$

另一方面，如果我们对文件作简单的顺序查找，磁盘头平均跨越的柱面数将是N/2，总开销为:

$$N/2 \times \text{寻道时间} + F/2 \times \text{传输时间}$$

由于传输时间中，寻道时间占很大一部分，所以只有在F远远大于N时[⊖]，二分查找才更有效率。纵观这些结论，对索引文件的等值查找而言，顺序查找和二分查找都不是最佳选择。一个更常用的方法是使用索引（参见下一节）对索引文件进行补充，然后在索引上使用二分查找法。因为可设计索引使之适于放在主存中，对索引实施二分查找是非常有效的，不会遇到前面所描述的磁盘延迟开销。

如果文件是在请求查找的同一属性上排序的，那么它也支持范围查找。因此，图11-4所示的TRANSCRIPT文件支持这样的查询：查找ID处在100000000和199999999之间的学生信息。在这一范围内，一个查找ID为100000000的等值查找定位该范围中的第一个元组（可能有ID为100000000的元组，也可能这样的元组不存在，此时元组的最小ID大于这个值）。这一范围内后续的元组存储在连续的槽内，在它们被检索时，很可能就会被高速缓存命中。如果B是存储在一页中的行数，R是一特定范围内的行数，那么为检索数据文件中包含在这一范围内（一旦第一行被定位后）的行所需的I/O操作数大约是R/B。

把这些数字和堆文件的数字相比较。在堆文件中，某一范围内的行可能处在不同的页面，为定位所有这些记录，有必要扫描整个文件。若数据文件有F个页面，则I/O操作数为F。类似地，索引文件支持满足（11.3）条件的元组检索，尽管在这种情况之下，表必须在Grade上建立索引，但不能同时在两个查找键上建立索引。

实际上，如果表是动态变化的，那么保持行索引有序是很困难的。当插入新行时，数据文件中插入点后面所有的行都要向下移动一个存储槽（平均要更新一半的页面），这要付出沉重的I/O代价。解决这个问题的一个（部分）办法就是：在建立数据文件时，为容纳以后要在行之间插入的记录，在每一页留出一些空的存储槽。术语**装填因子**（fillfactor）是指一个页面在初始状态存储槽被填满的百分比。例如，在一个堆文件中，装填因子是100%。在图11-4中，装填因子是75%，因为一个页面中的4个存储槽只填满3个。

但增加空的存储槽不能完全解决问题，因为在行被插入时，它们可能会被耗光。在后续行插入时，又出现同样的问题。**溢出页**（overflow page）可用于这种情形，如图11-5所示的TRANSCRIPT表。我们假设数据文件的初始状态如图11-4所示，每一页有一个指针字段，它包含溢出页的页号（如果存在的话）。在创建文件后，在第2页中插入新的两行。如果溢出页自身溢出，我们可以创建一个**溢出链**（overflow chain），即一个链接所有溢出页的链表。

在使用溢出链来使文件**逻辑排序**（logically sorted）时，却失去索引文件连续存储在磁盘空间上的优点，因为溢出页可能和与之相链接的页相距很远。这就导致顺序扫描读取记录中产生额外时间延迟，因此传输某一范围内记录的I/O操作数不再是R/B。如果顺序扫描的性能非常重要（实际就是这样），必须定期重新组织数据文件，以确保所有的记录在磁盘空间上是毗邻存储的。这里的教训是：如果文件是动态的，维持一个数据文件索引有序要付出代价。

11.4 索引

假设你为一个应用建立一个数据库，以供一组用户使用。但你不仅没有收到相应的回报，反而从用户那里收到许多系统太慢的报怨电话，这会使你非常生气。一些用户声称，他们要

⊖ 我们忽略了旋转延迟，和二分查找相比，它能更好地支持顺序查找。

等很长时间才得到查询的响应。还有一些用户声称，输出结果是不可接受的，提交查询的速率远远大于系统所能处理的能力。

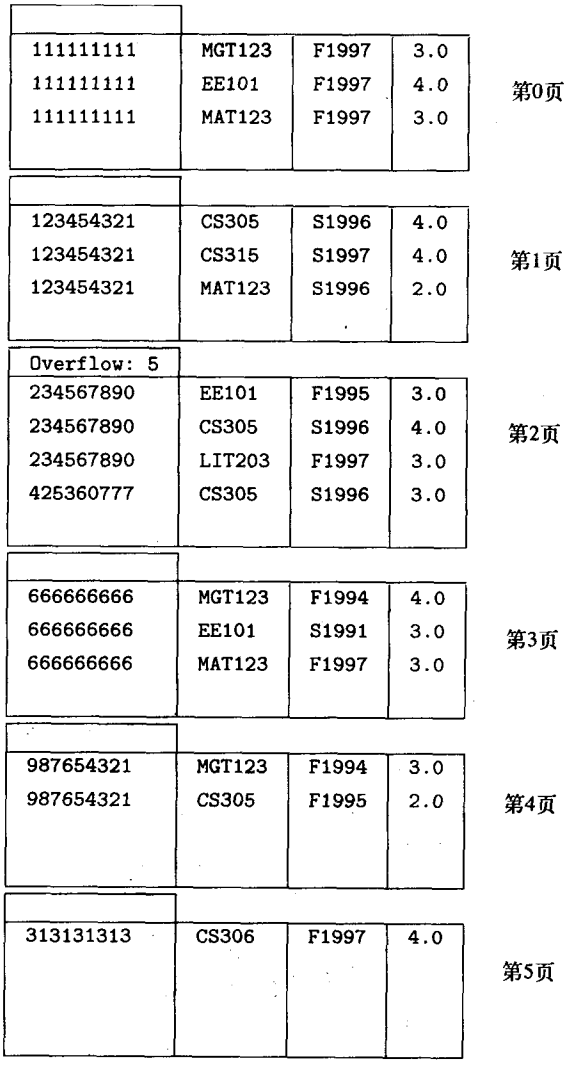


图11-5　在作为索引文件存储的TRANSCRIPT表中增加几行

索引可用来改善这种情况。表上的一个索引类似于一本书的索引，或者类似于图书馆的分类卡。对一本书来说，读者感兴趣的术语被挑选出来组成索引条目，每一个**索引条目**（index entry）包含一个术语（例如“gas turbines”）和一个指针（如“P.348”），指针用来定位术语在书中出现的位置。这些条目同时按术语进行排序，从而构成一个表，称为一个**索引**（index）。在讨论某个术语时，不是去扫描整本书，而是先访问索引，然后直接找到术语出现的地方。

这与分类卡的情形更加接近。在分类卡中，可能有基于书的不同属性而制成的多个索引：作者，书名，科目。在每个索引中，条目是按属性值排序的，每个条目指向书库中的一本书。因此，作者索引中的条目包含作者的姓名和指针（如section、shelf），它指向作者所写的一本书。

类似地，数据库表上的索引提供一种便利的方法来定位某一行（数据记录），而不必扫描整个表，因而可以大大地节约处理查询的时间。用来定位的属性是索引表中称为**查找键**（search key）的一列（或几列）。注意，不要把一个表上索引的查找键和表上的候选键相混淆。记住，一个候选键是一列或几列的集合，它有这样的属性，即任何表中的实例，没有两行在这些列上的值是相等的。查找键就没有这样的限制，因此，表中可有多个学生ID为111111111的行，这意味着StudId上的索引有多个索引条目，每个条目的查找键值都为111111111。

和候选键一样，一个查找键可能涉及几列，例如，查找键可能包括学生的StudId和Semester。但又和候选键不同，在查找键中，列顺序是不同的，所以把索引表中的查找键看成是列的一个顺序序列（和一个集合相对照），在我们讨论部分键查找时，你会发现它们的顺序是非常重要的。

索引是由一组索引条目和基于一个查找键值来有效地定位某个条目的机制所组成的。对某些索引，如ISAM索引和B^+树，定位机制依赖于索引条目在查找键上有序的事实。散列索引采用不同的方法。在这两种情况中，支持定位机制的数据结构和索引条目都被集成为数据文件包含在表中，如图11-6所示。这种集成后的数据文件被看作一种新的存储结构。以后我们所要讨论的ISAM、B^+树，以及散列存储结构，都可以作为堆文件和有序存储结构的一种替代。利用集成的存储结构，每条索引实际包含表的一行（不需要指针）。

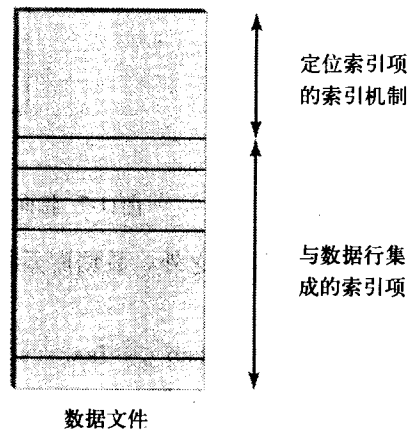


图11-6　索引与数据记录集成在一起的存储结构

另外，索引可以存储在一个单独的文件中，该文件称为**索引文件**（index file），文件中的每条索引包含一个查找键和一个rid[⊖]，这种组织形式如图11-7所示。例如，图11-2所示的TRANSCRIPT表在StudId上建立了索引，每条索引包含一个值和一个rid，值出现在表中某些行的StudId列上，rid是数据文件中页的ID。所以在索引中有（425360777，（3，1））这个项。

这种集成节约了存储空间，因为不需要指针，而且查找键既不必存储在索引项中，也不必存储在包含相应行的数据记录中。

SQL-92标准没有提供这种索引的建立和删除操作。但索引是数据库系统必需的部分，大多数供应厂商都会提供索引。在某些情形之下，带有查找键的索引是和主键等价的，在表被创建时自动产生。这种索引通常是集成的存储结构，能在添加或修改行时有效地保证主键的唯一性（能支持涉及主键的高效查询）。

⊖ 有些索引只存储了页的ID，因为访问数据文件的主要开销是传输页的开销，在索引项中只存储页的ID，使得准确定位页成为可能。一旦检索到页，使用顺序查找就能定位页上有目标查找值的记录，和传输页的时间相比，通常用于查找的附加计算时间是很少的，其合理性能在索引项上所节省的空间中得以证明。索引项更少的索引能装入更小的页面中，因而访问时只需更少的页I/O开销。在有些索引组织中，仅一个索引项就包含多个指针。由于这一特征没有增加新的概念，只会使我们的讨论复杂化，在此我们不进一步考虑。

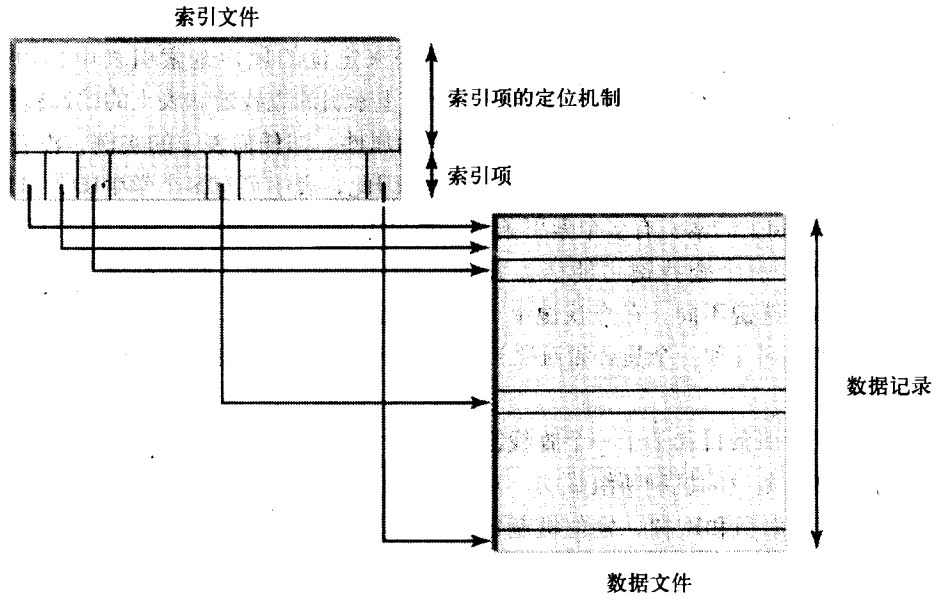


图11-7　指向一个单独的数据文件的索引

除自动创建的索引之外，数据库系统一般还提供一个语句（用它们的专用语言）来创建索引。例如:

```
CREATE INDEX TRANSGRD ON TRANSCRIPT (Grade)
```

这个语句将Grade作为查找键在表TRANSCRIPT上建立称作TRANSGRD的索引。如果CREATE INDEX没有提供一种选择来指定索引的类型，那么通常使用B^+树索引。在任何情况下，创建的索引被保存在一个索引文件中，如图11-7所示。它引用一个表，这个表已与存储结构中的另一个索引集成在一起。数据文件可以按某种存储结构来组织，它涉及一种定位机制，如图11-6所示。但为简单起见，在此我们不考虑这个问题。

使用一个合适的索引，能大大地减少查找时所要访问的数据文件中的页，但访问索引本身也是一种开销。如果索引很小，可以装入主存，那么这种开销是很小的。但如果索引很大，则必须从大容量的存储设备中检索索引页，这些I/O操作也必须视为查找净开销的一部分。

因为要对索引进行维护，所以必须有选择地添加索引。如果索引表是动态的，那么索引本身也必须随关系的变化而作相应的修改。例如，新插入一行到关系中，就要求在索引表中插入一条新的索引。所以，除在查找过程中访问索引表的开销外，索引作为数据库操作的一部分，也必须作相应的修改。由于这方面的开销，删除一条不能充分支持数据库查询的索引是可取的。因此，一个在CREATE INDEX中命名的索引名可以被DROP INDEX引用，这样可以删除一条索引。

在讨论特殊的索引结构之前，我们先介绍几个常见的用来分类索引的属性。

11.4.1　聚簇索引与非聚簇索引

在聚簇索引中，物理位置上相近的索引项在一定程度上预示相应数据记录也很相近。和非聚簇索引相比，这样的索引使得某些特定的查询能更加有效地执行。查询优化器能利用索

引的聚簇来改进查询执行的策略（在第13章和第14章将解释查询优化器是如何使用聚簇信息的）。聚簇可以采取不同的形式，这取决于索引项是有序的（如在ISAM和B^+树中）还是无序的（在散列索引中）。如果在相同的查找键上，索引项和数据记录都是有序的，那么就称有序索引是**聚簇的**（clustered），否则就称之为**非聚簇的**（unclustered）。聚簇的散列索引将在11.6.1节加以定义。这些定义表明，如果组织索引使索引项与数据记录集成在一起，那它一定是聚簇的。图11-7的索引就是聚簇索引，这里的索引和表存储在单独的文件中（因而数据记录不包含在索引项中）。在数据文件中，指针通常由索引项指向记录，这表明索引项和数据记录在同一列上排序。图11-8所示的索引是非聚簇的。

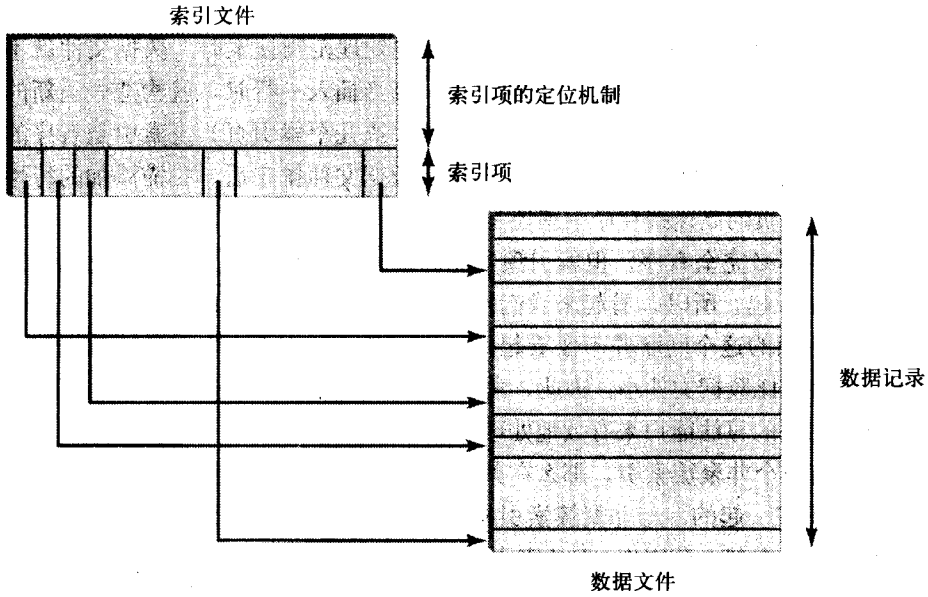


图11-8　数据文件上的非聚簇索引

聚簇索引通常称作**主索引**（main index），非聚簇索引通常称作是**辅助索引**（secondary index）[⊖]。最多只有一个聚簇索引（因为数据文件最多只能在一个查找键上排序），但可以有多个辅助索引。由语句CREATE TABLE创建的索引通常是聚簇索引。对某些数据库系统，聚簇索引总是与数据文件集成为一种存储结构。在这种情况下，索引项包含数据记录，如图11-6所示。由语句CREATE INDEX所创建的索引通常是辅助索引，非聚簇索引存储在一个单独的索引文件中（尽管有些数据库系统允许程序员请求创建一个聚簇索引，但这涉及存储结构的重组）。主索引上的查找键可能是表上的主键，但并不一定。

如果某列是查找键，同时又是辅助索引，那我们就说这个文件在这列上是反序的(inverted)。如果不被主键包含的所有列组成一个辅助索引，那么我们就说这个文件是完全反序的(fully inverted)。

⊖ 聚簇索引有时也称为主索引。我们避免使用这一术语，是因为在考虑关系上主索引与主键的关系时可能发生混淆。主索引并不一定是关系中主键上的索引，如关系PROFESSOR能按Department属性（甚至不是键）排序，建立聚簇索引，而关系中ID属性（同时也是主键）上的索引是非聚簇的，因为行并不在此属性上有序。

一个包含查找键的聚簇索引sk对涉及sk的范围查找是非常有效的。当索引项的查找键值是查找范围的一个边界值时，它的定位机制能有效地进行定位，因为索引项在sk上是有序的，在这一范围内，后续的各项处在同一个索引页中（或处在连续的页中，我们将在11.5节讨论这种情形），使用这些索引项就能检索数据记录。

聚簇索引的优点在于，数据记录本身是组织在一起的（而不是分散在整个数据文件）。它们或者包含在索引项中，或者按索引排序（见图11-7）。所以在某个范围内检索特定的数据记录时，高速缓存命中率是很高的，因为处在这一页的其他记录可能已经被访问过。如11.3节所述，在数据文件上所做的I/O操作数大约是R/B，其中R是在这一范围内的元组数，B是一页中的元组数。

使用一个与数据分开存储的聚簇索引，在有效地处理范围搜索时，数据文件就不必完全有序。例如，溢出页可用来存放动态插入的记录，每当插入一行时，就构造一条新的索引并将它放在索引文件的适当位置。由于在一个索引页（或几个索引页）上索引是有序的，所以索引的定位机制能有效地找到所有的索引。即使在数据文件排序后，扫描添加的行性能会有所下降，但检索数据页的高速缓存命中率还是很高。

尽管数据文件不必完全有序，但索引项不能简单地添加到索引文件中，它们必须与索引的定位机制集成在一起。所以，看起来我们把使数据文件有序的难题转换成使索引文件正确组织的问题。但后面的这个问题并不像看起来那样困难。一方面，索引项比数据记录数少得多，所以索引文件远比数据文件小。因此，索引的重组只需较少的I/O。而且，我们将要看到，一个与B^+树索引相关的算法能用来有效地处理索引的添加和删除。

如果在sk上有一个非聚簇索引，那么在某一范围内包含查找键值的索引项是可以定位的，因为索引项是组织在一起的。与非聚簇索引相关的问题是，相应的记录可能分散在整个数据文件中。因此，如果在某一范围内有R条索引，为检索数据记录，可能有必要进行R次I/O操作（在后面的小节中，我们将讨论对这一数据进行优化的技术）。

例如，一个数据文件可能包含10 000页，但对一个特定的查询来说，可能只有100条记录落在这一范围内。如果这个查询的WHERE子句的属性上有一个非聚簇索引是可用的，那么检索这个数据文件的页最多只需要100次I/O操作（如果数据存储在一个没有索引的堆文件中，那么可能需要10 000次I/O操作）。如果索引是聚簇的，每个数据文件的页平均包含20条数据记录，那么大约需要检索5个数据页，在比较的过程中，我们忽略索引文件上的I/O的操作。对不同的索引结构，我们分别讨论索引的I/O操作。

11.4.2 稀疏索引和稠密索引

我们已经假设索引是稠密的。稠密索引（dense index）是指索引中的项与数据文件中的每一条记录一一对应。非聚簇索引一定是稠密的，而聚簇索引却不一定是稠密的。一个有序文件上的稀疏索引是指对数据文件中的每一页，都有一个索引项，反之亦然。每条索引包含一个小于或等于它指向页的所有记录的值。稀疏索引和稠密索引的区别可以用图11-9中的表PROFESSOR来说明。为对图进行简化，我们没有显示出数据文件中的溢出页，并假设所有的存储槽都被填满。数据文件用两种不同的索引文件进行索引，在索引文件中只显示了索引项，不包括定位机制。而且我们假设每页有4个存储槽。在这个表上，属性ID是主键，同时也是表

示在图左边稀疏索引的查找键，属性Name是表示在图右边的稠密索引的查找键。

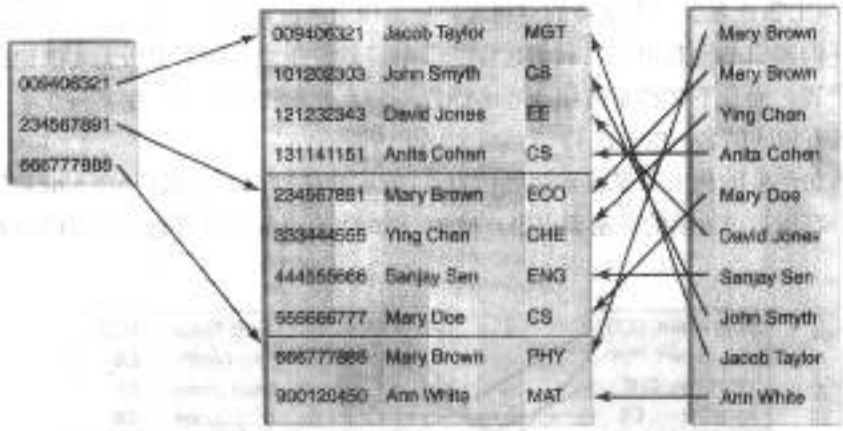


图11-9 有查找键Id的稀疏索引（左边），查找键为Name的稠密索引（右边），它们都指向表PROFESSOR，该表存储在一个文件中，并按Id排序

为检索Id为333444555的记录，我们可以使用稀疏索引来定位小于目标Id但值最大的索引项。在这里，符合上述条件的索引项的值为234567891，它指向数据文件的第一页。一旦检索到这一页，可通过向前搜索该页来找到目标记录。使用稀疏索引时，数据文件按作为索引的关键字排序是很重要的，因为有序文件允许我们定位不被索引引用的记录，因此稀疏索引必须是聚簇的。

非常重要的一点是，稀疏索引的查找键应是表上的一个候选键。因为如果数据文件中，对相同的查找键值，存在多个记录分散在不同的页，那么第一页中的记录就有可能被错过，该问题如图11-10所示。稀疏索引的查找键值是表的第一列，它不是候选键。通过寻找查找键值为20的记录的索引，指针指向一个查找键值为20的索引，向前搜索目标页，得到查找键值为20的一条记录，而错过数据文件的前一页中含有同样查找键值的记录。



图11-10 稀疏索引的一个问题：查找键不是候选键

可以应用多种技术来处理这类问题，最简单的方法是：当目标值和稀疏索引值相等时，搜索数据文件前一页。另一个办法是为表中每个不同的键值创建一个索引项（不同于每页对应一条索引）。这样，就可能有多条索引指向同一页，即使表中的行有很多重复的查找键值，这类索引也比一个稠密索引要小很多。

非聚簇索引是稠密的，图11-9所示的索引是非聚簇索引，因为每条索引对应一条记录，查找键不必是表上的候选键，因此，多个索引项可以有相同的查找键值（如图所示）。

11.4.3 包含多个属性的查找键

查找键可能包含多个属性。例如，可以通过执行下列语句

```
CREATE INDEX NAMEDEPT ON PROFESSOR (Name, DeptId)
```

在PROFESSOR上创建索引。当支持的应用客户经常请求通过姓名和系别Id来查询教授的有关信息时，这样的索引是很有用的，利用这样的索引，系统能直接检索出所需要的记录，不仅不必扫描整个表，而且不必检索有相同姓名而在不同系的教授的记录（在某一个系，不可能出现两名教授有相同的名字的情况）。

这条语句产生的稠密非聚簇索引如图11-11所示的结论，索引项是按Name和DeptId排序的（注意，在图11-9所示的稠密索引中，Mary Brown的两条索引与它们所在的位置顺序现在颠倒了）。

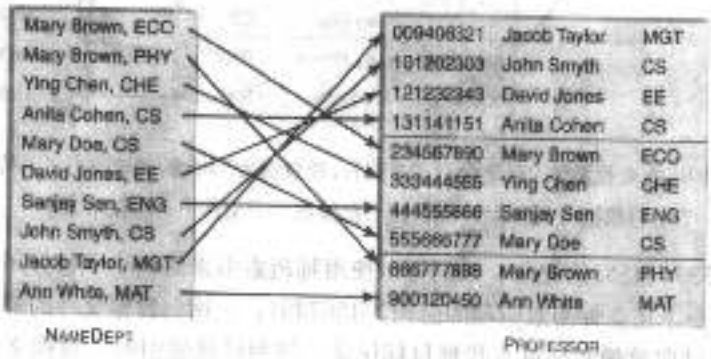


图11-11　表PROFESSOR上查找键值为Name、DeptId的稠密索引

在查找键中使用多个属性的第一个优点在于，结果索引支持更细粒度的查询。利用NAMEDEPT我们可以迅速检索出经济系中名为Mary Brown教授的信息，图11-9所示的稠密索引要求我们检查两条数据记录。

第二个优点在于，如果索引项是有序的（针对ISAM和B+树索引），它能支持多种类型的范围搜索。例如，我们可以检索出某个系所有名为Mary Brown的教授的记录（一个等值搜索），而且我们可检索出所有名为Mary Brown的教授，其所在系名按字母顺序处于经济系和社会学系之间的记录信息，或者可检索出所有系中名为Mary Brown的教授的记录信息，还可检索出所有系中名字按字母顺序处在Mary Brown和David Jones之间的教授的记录信息。以上所有的这些例子都是范围查询，因为索引项首先是按Name排序，其次是按DeptId排序，所以这些查询都被支持。因此，目标索引项在索引文件中是逻辑上连续的，我们能限制扫描的范围，后面的两个例子是部分键查找（partial key search），即查找键中一些属性的值没有被指定。

注意，NAMEDEPT不支持范围查找，其中没有给出姓名的值（例如，检索出所有经济系的教授）。因为在这种情况下，目标索引项在索引文件中不是连续的，这正是查找键区别于候选键的原因。在查找键中，属性的顺序是非常重要的，而对候选键却不是这样。使用部分键查找，查找键的前缀值可用于索引的查找，但索引却不支持键值的后缀查找。

例如，在设计5.7节给出的学生注册系统时，TRANSCRIPT表的主键是StudId、CrsCode、SectionNo、Semester和Year，可以在Student Grade交互上使用一个索引，它的查找键由给定顺序的上述属性组成，因为在给出一名学生成绩时，必须提供所有这些信息。但Class Roster交互不能使用这一索引，因为查找指定的是CrsCode、SectionNo、Semester、Year，而不是

StudId。由于有StudId，所以索引对查找学生的历史成绩是有帮助的，但这不是最优的，因为在交互过程中使用一个SELECT语句，其中的WHERE子句指定StudId和Semester/Year，遗憾的是，没有CrsCode和SectionNo，索引不支持使用Semester/Year。在主键中，将这些属性的顺序颠倒，就能支持Student Grade和Grade History的查询，Student Roster的交互还需要另一个索引。

一般地，表**R**上查找键为K的一个树索引支持下列形式的查找:

$$\sigma_{attr_1\ op_1\ val_1\ \wedge\ \cdots\ \wedge\ attr_n\ op_n\ val_n}(\mathbf{R}) \tag{11.4}$$

但其前提条件是K的一些前缀是查找键中命名的属性的一个子集。例如，如果（11.4）的属性是有序的，$attr_1$，…，$attr_s$，$s \leqslant n$，是K的一个前缀，那么对某个$s<n$，搜索定位满足$attr_1=val_1 \wedge \cdots \wedge attr_s = val_s$的最小索引项，并从满足（11.4）的项处开始向前扫描。因此，一个特定的索引可以按不同的方式支持多种形式的查找，可以按多个访问路径来访问**R**。

最后，含多个属性的查找键表明：一个索引可包含索引表中任意的信息。一个含有多个属性的稠密索引，其每一项对应表中的一行，它可以包含表中任意多个列的信息。这意味着，对一些查询，不需访问表就可从索引中找到所请求的信息。例如，执行下列查询

```
SELECT   P.Name
FROM     PROFESSOR P
WHERE    P.DeptId = 'EE'
```

结果可从索引NAMEDEPT中得到，而不必通过扫描索引项访问PROFESSOR。由于存储索引的页比表页要少，所以扫描索引的开销比扫描表PROFESSOR的开销要小。

11.5 多级索引

在前面的章节中，我们对索引项进行了讨论，而没有讨论用定位机制查找索引。在这一部分，我们来讨论树索引中使用的定位机制，然后描述**索引顺序存取方法**（ISAM）和B⁺树中特殊的定位机制。

为理解树索引中的定位机制的工作原理，考虑图11-9所示表PROFESSOR上的稠密索引。因为查找键是Name，所以索引项在这个字段上是有序的，因为数据文件中数据记录的顺序是不同的，所以索引是非聚簇的。我们假设索引项顺序存储在索引文件的页中。对这些索引而言，一个朴素的定位机制是二分查找法。因此，为定位描述Sanjay Sen的记录，我们先找到索引项列的中点，比较Sanjay Sen和这页查找键的值。如果目标值能在这页找到，就定位索引项；如果小于（或大于）这页上查找键的值，这个过程就在索引表的前一半（或后一半）上递归地重复进行。一旦定位到索引项，就将含有这条记录的数据页提取出来，仅有一次额外的I/O操作。

和11.3节所描述的将二分查找法用于有序的数据文件相比，把二分查找法用于索引文件是一个很大的改进，因为数据记录通常比索引项要多得多。所以，如果Q是包含索引项的稠密树索引的页面数，F是数据文件的页面数，则Q<<F。使用二分查找法来定位一条索引的I/O操作数是不可能大于$\log_2 Q$的。

通过为索引项列表建立索引，我们可以进一步减少定位索引项的开销。使用相同的查找

键，我们在索引项上构建一个稀疏索引（这样的稀疏索引是可能的，因为索引项在这个键上有序），这时我们再对索引使用二分查找法。二级索引上的索引项作为分类符，对查找一级索引上的索引项起引导作用。因此，我们使用术语**叶子项**（leaf entry）来指处于索引树上最低一级上的索引项，使用**分类符项**（separator entry）来指处在上层的索引项。

这一技术如图11-12所示，该图表示一个二级索引。为使图更加精确，我们假设表上的一个候选键是整型的，我们使用该键来作索引的查找键，我们还假设索引文件上的一页可以容纳4条索引项。当然，在实际的系统中，一个索引页可以容纳更多条索引（比如说100条），和数据文件的大小相比，索引的大小是很小的。

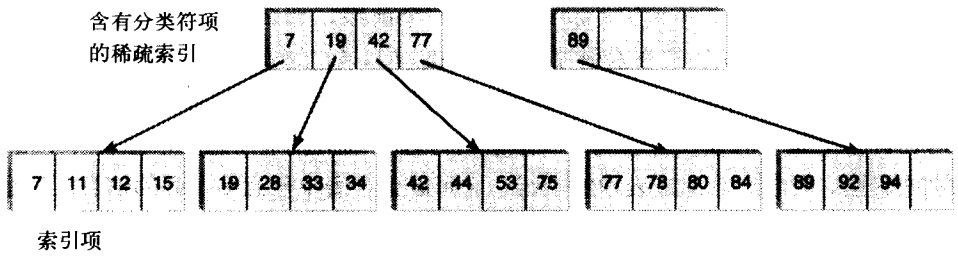


图11-12　二级的索引

在这个图和后面的图中，我们显式地显示数据记录。可以用两种方式来解释图：1）索引（叶子）项包含一个指向数据文件中数据记录的指针，2）叶子索引包含数据记录，图代表一种存储结构。在解释1中，索引文件包含叶子索引和二级索引，访问数据记录需要一次额外的I/O操作，在解释2中，定位一个叶子索引需提取相应的数据，不需要额外的I/O操作，我们所讨论的多级索引可应用于这两种情形。

最后，尽管图中二级索引的分类符看起来是和叶子索引相等的，但事实却并不总是这样。在解释2中，只有叶子索引包含数据记录，而分类符却没有包含数据记录，因而相比较而言，叶子索引比分类符大。在解释1中，分类符和叶子索引项看起来相等，只是第二级上的分类符指向第一级上的结点，而叶子索引则指向数据文件中实际的记录。这些指针的格式是不同的。

使用二级索引，我们把叶子的二分查找分为两步：第一步，对最上面一级的稀疏索引使用二分查找法，找到指向包含目标叶子索引页的分类符项；第二步，从页中找到相应的索引项（如果存在的话）。所以，如果我们要找查找键的键值为33的项时，我们首先使用二分查找法来定位值小于或等于33的最右边的分类符，这里的分类符值为19。然后我们顺着指针找到叶子索引的第二页，在这一页上作线性搜索找到需要的索引项。如果目标查找键值是32，我们将执行相同的步骤，然后得出结论：在候选键属性上，索引表中没有值为32的一行。

通过引入二级索引，我们得到什么？因为上面级别的索引是稀疏的，所以和第一级的叶子索引相比，分类符是较少的。如果我们假设数据和索引文件是分开的，那么分类符和叶子索引大小差不多。而且，我们假设在一个索引页上包含100条索引，那么二级索引就包含Q/100个页（这里的Q是指叶子页的数目），那么使用二分查找法访问二级索引上的一个叶子索引的最大开销大约是$\log_2(Q/100)+1$个页的I/O操作（这里的1是指包含叶子索引的那页），和对叶子索引实施二分查找法的开销$\log_2 Q$相比，如果Q相当大时，这能避免很大的开销。

这促使我们研究多级索引。如果一个二级索引是可取的，那为什么不使用多级索引呢？每一级索引使用更少的、更高一级的稀疏索引来进行索引，直到索引能被包含在一个页面中为止。搜索该索引的I/O开销是1，所以定位目标叶子索引的总开销和索引的级数是相等的。

一个多级索引如图11-13所示，每个长方形框代表一页。树的最低一级包含叶子索引，它被称作是**叶子级**（leaf level）。如果我们将所有页按顺序都放在这一级上，那么叶子索引就构成一个有序的列表，上层包含分类符，被称为**分类符级**（separator level）；如果我们在某一特定的分类符级上按顺序把所有的页都连接起来，我们就得到下一级的一个稀疏索引。树的根（最顶层分类符）是一个稀疏索引，它包含在索引文件的一个页面中。如果叶子索引包含数据记录，那么图就表示一个树索引文件的存储结构。

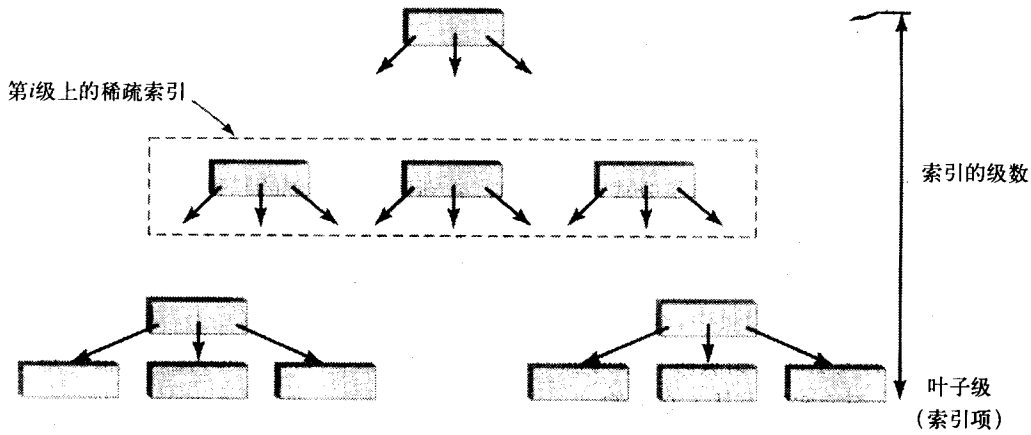


图11-13　一个多级索引的图示

我们使用术语**索引级**（index level）来指索引树上的任意一级，叶子或者是分类符。使用术语**扇出数**（fan out）来指一页中分类符的数目。扇出数可以控制树的级数：扇出数越小，树的级数就越多。树的级数是和提取一个叶子索引的I/O操作数相等的。如果扇出数用Φ来表示，提取一个叶子索引所需的I/O操作数是$\log_{\Phi}Q+1$。

例如，如果在叶子级上有10 000页，扇出数是100（在数据文件中，假设有10^6行，且叶子索引和分类索引的大小是相同的），那么检索一个特定叶子索引就需要对3个页进行I/O操作。因而，使用大的扇出数，即使对很大的数据文件，对索引的遍历都会减少到只需要很少的I/O操作。因为根索引只占一个页面，所以它能经常保存在主存中，可以进一步地减少开销。即使是二级索引，在这里也只需要100个页面，实际上也是可以放在主存中的。

多级索引构成树索引的基础，我们下面要讨论树索引，说明为什么树索引被经常使用，不仅是因为它提供对数据文件的有效的访问方法，正如我们将会看到的，而且它还支持范围查询。

11.5.1 索引顺序访问

索引顺序访问方法（ISAM）[⊖]是基于多级索引的。ISAM索引是一个主索引。因此它是

⊖ 通常在使用时，访问方法（access method）和访问路径（access path）是可互换的，但我们更偏向于使用访问路径，因为它在概念上更加一致。

记录上的聚簇索引，这些记录是按索引的查找键排序的。一般来说，记录是放在叶子索引中的，ISAM是数据文件的一个存储结构。

在分类符级上，一个页的格式如图11-14所示，每一个分类级是它下一级的稀疏索引。每一个分类索引由一个查找键值k_i和一个指向索引文件其他页的指针p_i组成。这一页可能在下一页，或处在下一分类符级上，或是包含叶子索引中的页。分类索引在页中是有序的，我们假设一页最多包含Φ条分类索引。

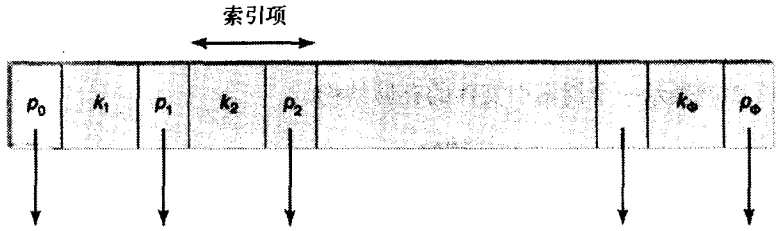


图11-14　一个ISAM索引在分类符级上的页

每一个查找键值k_i将查找键值集分成两棵子树，分别用两个相邻的指针p_{i-1}和p_i指向这两棵子树。如果一个查找键值k在由p_{i-1}所指向的子树中，则满足$k<k_i$，如果在由p_i所指向的子树中，则满足$k \geqslant k_i$。这就是为什么查找键值在索引页被称为分类符的原因。和一个多级索引中的稀疏索引页相比，在ISAM索引页中，似乎还包含一个额外的指针p_0[⊖]。实际上，比较一个ISAM索引页和分类索引级上的索引页时，较好的方法是分类符包含一个额外的查找键值：最小的查找键值在k_0页，而这实际上是不必要的。

图11-15是一个ISAM索引的例子。其中，树中含有两级分类符和一个叶子级索引，查找键值是学生的名字，名字是按字母顺序编排的。最左边子树上（由根上的p_0指向）所有的查找键值都小于judy，中间子树上（由根上的p_1指向）所有的查找键值都大于或等于judy。在创建一个ISAM文件时，首先顺序分配存储结构中的页作为叶子索引页（包含数据行），然后自底向上构建分类符级：根是最上面一级的索引。因此，ISAM索引有这样的属性，所有出现在分类索引级的查找键值，同时也出现在叶子索引级。

查找一个包含查找键值karen的数据记录是从根开始的，确定目标键值处在judy和rick之间，在由p_1指向的下一级索引的中间页上。在该页，确定karen小于mike，沿着p_0所指，在叶子索引页上找到记录所在的位置。如果要检索所有查找键值在karen和pete之间的数据记录，我们首先用刚才的方法找到包含karen的叶子索引，然后扫描叶子一级，直到遇到包含pete的索引项为止。因为在叶子级上的页是按索引顺序有序存储的，所以要找的项在文件中是连续的。如果要检索查找键值在kate和paul之间的所有数据记录，我们将扫描相同的叶子页。另外，ISAM索引还支持多属性键和部分属性键的查找。

ISAM索引以如下事实为特征：分类符级一旦被创建就不再改变。即使叶子级的页的内容可能改变，页本身不会被释放或重新分配。因此，它们在文件中的位置是固定不变的。如果表中的一行被删除，相应的叶子项也从叶子级中删除，但不对分类符级进行修改。这样的删除就会导致一个查找键值在分类符中存在，却没有相对应的叶子项。这是可能发生的。例如，如果

⊖ 注意，现在的扇出是Φ+1，由于Φ通常要比1大得多，因此在开销计算中，我们忽略Φ与Φ+1的差别。

从图11-15中将记录jane删除，就会出现这样的情况。这样的索引看起来很奇怪，但它却仍然能正确的工作，也不会产生什么严重的问题（除了浪费叶子项原先所在的存储空间）。

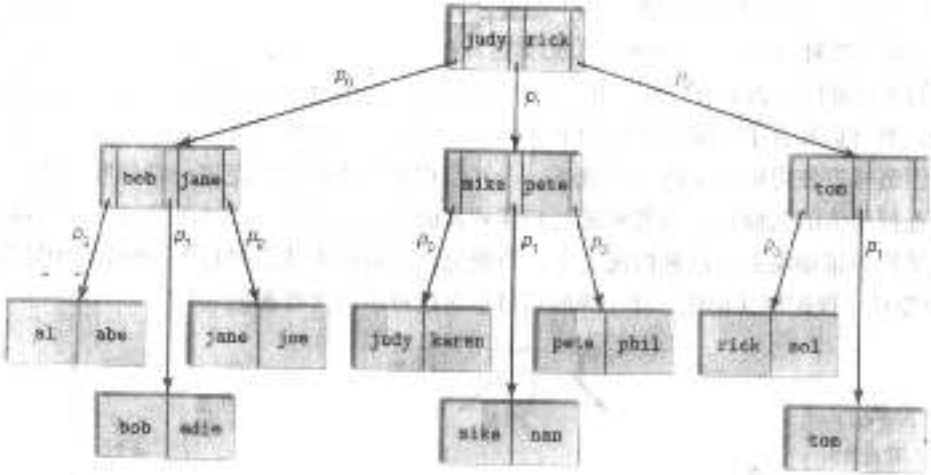


图11-15 一个ISAM索引的例子

当添加一条记录时，会出现很严重的问题，因为必须创建一个新的索引项，而相应的叶子页却可能已经满。通过使用一个小于1的装填因子（对一个ISAM而言，装填因子为0.75是比较合理的），可以避免这种情况的发生，但最终可能还是需要溢出页。例如，插入关于ivan的一行（jane行已经删除），结果索引最左边的子树结果如图11-16所示。注意，新页是一个溢出的叶子一级的页，不是一级新的索引或者一个新的叶子级的页。如果一个表是动态的，那么经常有插入操作，溢出链就会变得很长。结果，索引结构就变得越来越低效，因为要满足查询要求，就必须搜索溢出链。链上的各项可能不是有序的，溢出页在大容量存储设备上可能并不在物理位置上相邻。为消除溢出链，索引要定期重组。由于这方面的原因，尽管ISAM索引对相对静态的表是有效的，但动态表不常使用ISAM。

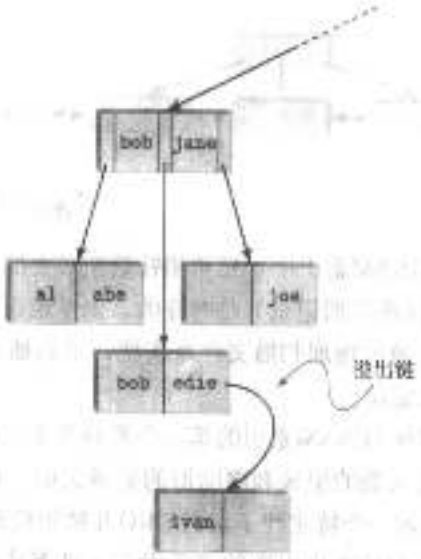


图11-16 在图11-15上进行一次插入和一次删除之后ISAM索引的一部分

11.5.2 B+树

B+树在索引结构中最常用。实际上，在某些数据库系统中，它是唯一可用的。和ISAM索引一样，B+树结构是基于多级索引的，它支持等值查找、范围查找和部分键查找。叶子索引可包含数据记录，在这种情况之下，B+树不仅作为索引，而且还可以作为一种存储结构，用来组织数据文件中的记录。树可能存储在一个索引文件中，其中叶子索引包含指向数据文件

记录的指针。在第一种情形中，因为它是聚簇的，所以B+树是一种类似于ISAM索引的主索引。在第二种情形中，它可能是一种主索引或辅助索引，数据文件不必在它的查找键上排序。

图11-17是一个B+树的示意图，其中每页含有一个有序的索引集。它与图11-13的唯一区别是添加了兄弟指针（sibling pointer）。兄弟指针把叶子级上的页连接起来。这样的链接表使得记录上的查找键值是索引有序的。和一个ISAM相反，B+树本身是动态改变的。当增加和删除记录时，索引页和页了叶会做相应的增加，删除或修改。因此，叶子页在文件中可能不是连续的。但链接表使得B+树支持范围查找。一旦使用等值查找定位范围一端的叶子索引，范围内包含查找键值的其他叶子项就可通过扫描列表而得到。注意，在这种模式下，B+树既作为主索引又作为辅助索引（这种情况之下，数据记录不必按查找键排序）。通过增加兄弟指针，图11-15变成一棵B+树（记住，叶子级的页可包含也可不包含数据记录）[8]。

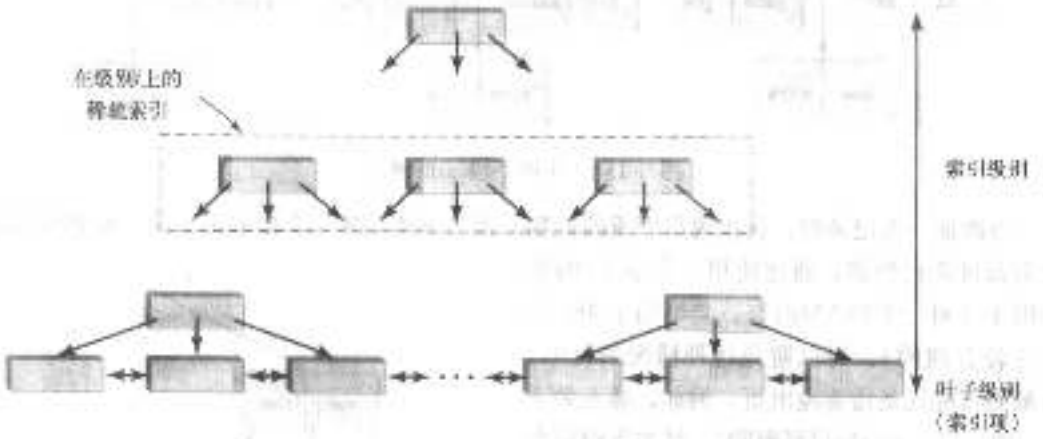


图11-17　B+树的示意图

在ISAM索引中，兄弟指针是不必要的，因为在文件创建时，存储在文件中的叶子索引页（通常包含数据记录）是有序的，由于索引是静态的，所以顺序保持不变。所以，一个范围查找可以通过物理扫描文件来实现。动态插入的索引项不是按索引顺序保存的，但可以通过溢出链来定位。

B+树与ISAM索引的第二个差异在于，B+树是一棵平衡树（balanced tree）。这意味着，即使在插入新的记录和删除旧的记录之后，任何从根到叶子索引的路径都具有相同的长度。因此，检索一个特定叶子索引的I/O开销和检索所有其他叶子的I/O开销是相同的。我们已经知道，使用具有适当扇出数的多级索引可显著减少开销。Φ是一个索引页中所能容纳的最大分类索引数。如果我们假设插入和删除项的算法（后面将简短地描述）保证一页中的分类符数至少是Φ/2（即最小扇出数是Φ/2）[9]，那么在一棵有Q个叶子页的B+树上的查找开销最多是$\log_{\Phi/2}Q+1$（一般来说，根节点上的分类索引数可以小于Φ/2，这个因素对公式有一点影响），这是和有溢出链的ISAM索引不同的。因为链的长度是无限的，所以链末端叶子索引查找的开

8　B树与B+树之差别在于，B树上的每一页都有一个指针指向实际数据文件的一条记录，即每个索引页都有一个分类符来叶子索引，这表明，一个特定的查找键值在树上只出现一次，而B+树却没有这样的属性。

9　对于Φ为奇数的情形，最小的分类符数应为（Φ/2），即大于Φ/2的最小整数。

销也是无限的。

因此，和ISAM索引相比，B⁺树的一个主要优点在于它能适应表的动态改变。当添加一个新记录时，不是创建一条溢出链，而是修改索引结构使树保持平衡。当结构发生改变时，从页中添加和删除索引项，使一页中的分类索引数始终保持在Φ/2和Φ之间。

让我们来看一下在图11-15的索引中进行一系列的插入操作时B⁺树是如何工作的。这种情况下，Φ=2。图11-18显示在插入vince记录之后的最右边的子树。因为按查找键的顺序，vince是在tom之后，在最右边的叶子页上有空的叶子索引，所以不需对B⁺树的结构做修改。

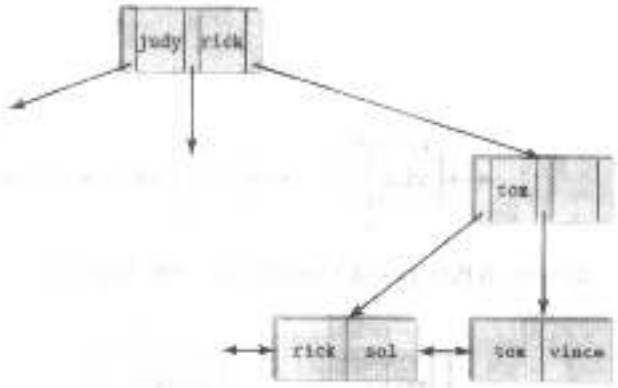


图11-18　在图11-15中插入vince后的部分索引

假设下一个要插入的是vera，它在tom之后。因为最右边的叶子页已满，这时需要一个新页。但不是使用一个溢出页（像ISAM索引那样），我们创建一个新的叶子页，这要求对索引的结构进行修改，这就是B⁺树区别于ISAM的地方。因为必须保持查找键的值在叶子一级有序，所以vera必须插入在tom和vince之间。因此简单地创建一个包含vera的新叶子页放在最右边是不够的。相反，我们必须分配一个新的叶子页，将有序的查找键分成大致相等的两部分，一半放在已存在的叶子页中，另一半放在新的叶子页中，结果如图11-19所示。在图中，新叶子页中最小的一项vince标为C。因此，vince在索引页D中变成一个新的分类符（叶子页B中所有的索引项都小于vince）。现在可以有足够的空间来容纳那项索引。一般来说，当一个（满的）叶子页包含Φ项索引时，为容纳新插入的项，就把它分为两个叶子页，其中一页包含Φ/2+1条索引，另一页包含Φ/2条索引，并在它的上一级索引上插入一条分类索引，我们把这个规则称为规则1。注意，对于vince，有一条分类符和一条叶子索引。

如果再插入一条rob，问题就更加严重，新的叶子索引必须处在rick和sol之间，这就要求对页面A进行分裂，在页D上需要4个指针（指向新分裂的两个页和B、C）。但一个页面只能有三个指针，因而我们还必须对D进行分裂。并且，按照规则1，我们必须为sol创建一条分类符。在一般情况下，当一个索引页要存储Φ+1条分类符时（在这里是sol、tom、vince），就要分裂它以得到Φ+2个指针指向下一级的索引页。分裂后的每一个页分别含有Φ/2+1指针和Φ/2个分类符，可能看起来好像有一条分类符放错位置，但实际上并不是这样。

将页A分裂成A1和A2，D分裂成D1和D2之后的情形如图11-20所示。注意，页D1和D2中分类符数之和等于页D中的分类索引数，尽管值可能不同，用sol代替了tom，这个数字看起来很奇怪，因为叶子级上分裂页所需的分类符数是3（sol、tom、vince），可以用tom来解释，它

将D1和D2中的值分开放在两个子树中，自身变成更高一层上的分类符。一般地，在分裂一个页以容纳Φ+1个分类符时，中间的那个分类索引不是存储在分裂后的两个页面里，而是被推到树的上一层。我们把这个规则称为规则2。

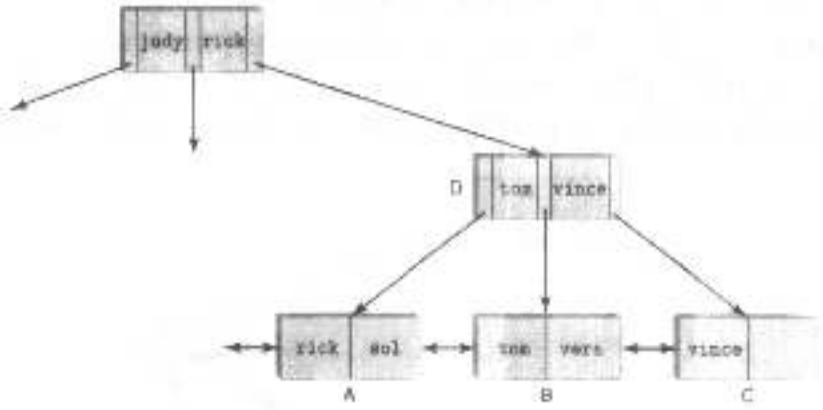


图11-19　在图11-18中插入vera后导致一个叶子页的分裂

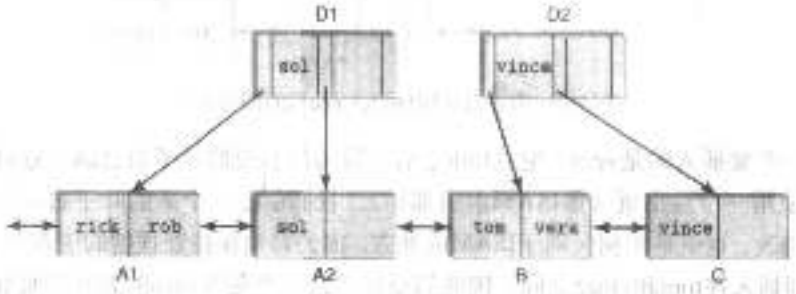


图11-20　图11-19的索引子树在插入rob之后，导致一个叶子页和一个索引页的分裂

因此，插入操作并没有完成。分类索引必须向上推，在更高一层的索引级上，对索引页D2必须指定一条新的参照索引。换句话说，这个过程必须继续重复。一般来说，这个过程要一直重复达到某个索引级，这个索引级能够容纳一条新的分类索引而不必再进行分裂为止。在我们的例子中，下一个索引级是根索引，因为它不能容纳一条新的分类索引，也必须分裂，这里的分类索引顺序是judy、rick和tom。根据规则2，rick必须推到新产生的根索引页中。

这时，插入过程完成，新产生的B*树如图11-21所示。注意，虽然层数增加1，但树仍然是平衡的。访问任何一个叶子索引需要4次I/O操作，如果表被频繁地访问，那么让根索引放在主存中，可使I/O操作次数减1。还要注意，在分裂A时，指向B的兄弟指针必须更新，这要求一次额外的I/O操作。

节点的分裂使得开销增加，所以应该尽量避免节点分裂。一种办法就是在创建B*树时使用一个小于1的装填因子（装填因子为0.75比较合适）。当然，这会使得树的层数增加。另一种插入算法是试图避免分裂，这涉及叶子级上索引项的重新分布。例如，在图11-9中我们插入叶子项tony，为在B页上腾出空间，需要把vera移到C页上（在D页上把分类符项vince替换成vera）。重新分布通常是在叶子一级的相邻页上完成的，这些叶子有相同的父亲。这样的一些页被称为兄弟（sibling）。

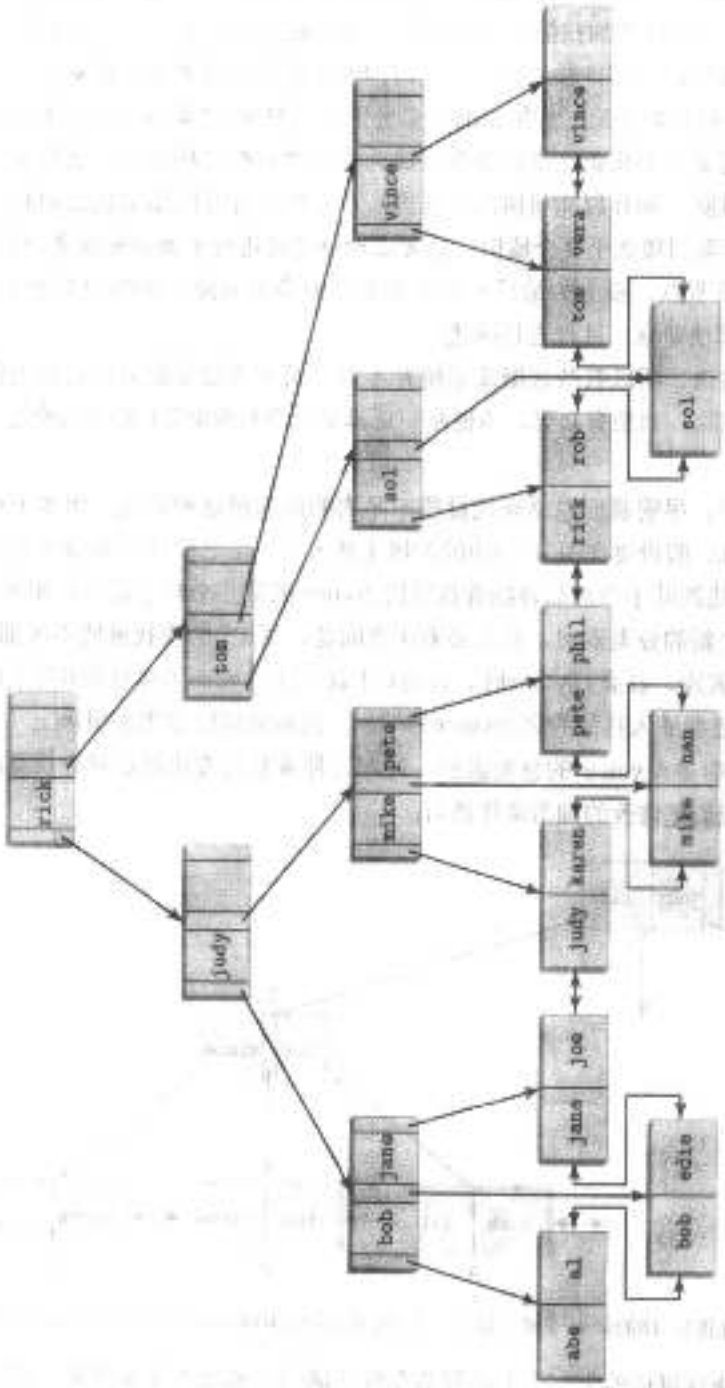


图11-21　在图11-15中插入vince，vera和rob后所得到的B+树

以上的讨论解释在B+树上插入一项的主要过程。练习11.12要求用实际的算法来说明这一过程。

删除是另一类问题。当页变得稀疏时，树会变得越来越深，这是不必要的。因为这不仅增加向下查找的代价，而且增加扫描叶子的代价。为避免这种情况，可以对页进行压缩处理，要求每一页至少包含Φ/2个索引项。当对一个含有Φ/2个索引项的页进行删除操作时，先对它兄弟页上的索引项进行重新分配。如果它的兄弟页上也只有Φ/2个索引项时，重新分配是不可能的，这时就把它与某一个兄弟页进行合并，成为一个含有Φ-1项的页，然后将一条索引项从树的更高一层上删除。和分裂能将树向上推进一层一样，合并的结果也是如此。删除可能导致一个索引页上的索引项低于某个域值，这要求对分类符进行重新分配或者对页进行合并，如果这种影响传到根节点，根上的最后一个分类索引就会被删除，树的深度就减少1。练习11.13要求给出一个算法对这一过程进行描述。

由于表增长得很快，所以有些数据库系统并不对一页中所包含的最小索引项作要求，只是在页为空时进行删除。如果有必要，为使每一页满足至少包含Φ/2个索引项的这一要求，可以对树进行重构。

在前面的讨论中，尽管我们忽略查找键值不是表的候选键这种情况，但多个行可能有相同的查找键值。例如，假设要在图11-18中的B+树上插入另一位名叫vince的学生记录。使用规则1，必须分裂最右边的叶子节点以容纳查找键值为vince的第二个叶子索引，如图11-22所示。因而就必须创建一个新的分类索引。首先必须注意的是，页B上的查找键值不再都小于D页上索引项vince的值。其次，在查找vince时，在页C上就终止，因而不会找到B页上的叶子索引vince。最后，如果还要插入其他的名为vince的学生，在最低层的分类索引级上（也可能是更高的一层上），可能有多个vince 的分类索引。处理这种重复的方法就是对查找算法进行修改以适应这些变化。我们把修改的细节留作练习。

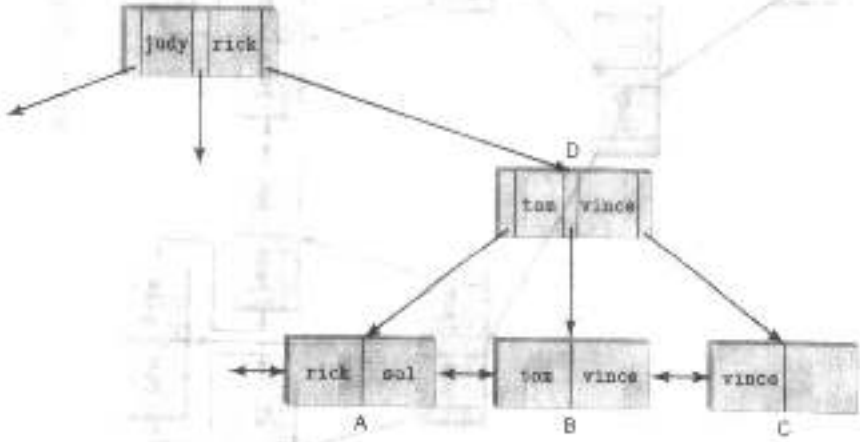


图11-22　在图11-18的索引子树上插入一个相同的索引项vince后，导致叶子页的分裂

处理插入相同查找键值的另一个方法就是在叶子满时，创建一个溢出页。使用这种方法，就不必对查找算法进行修改。但溢出链会变得很长，前面所描述的树查找代价估计方法将不再适用。

11.6 散列索引

在很多的计算机应用中，散列是一个非常重要的查找算法。在这一节，我们讨论它在数据库关系索引上的使用。**静态散列**（static hashing）中，散列表的大小是一个常量，而在**动态散列**（dynamic hashing）中，表可以增大和缩小。当一个关系在大部分情况下是静态时，使用第一种技术比较合适。当关系索引涉及经常的插入和删除操作时，利用第二种技术较好。

11.6.1 静态散列

散列索引把对应于一个表上的数据记录分成多个不相邻的子集，这些子集称为**桶**（bucket），这与某个**散列函数**（hash function）h相对应。当一条新的索引插入某个特定的桶时，这取决于在索引项上对查找键值v应用函数h后的结果。因此，$h(v)$是桶的地址。由于不同的查找键值数目通常比桶的数目要大得多，所以某一个桶将包含具有不同查找键值的索引项。每一个桶一般存储在一个页上（可能加以扩展包含一个溢出链），由$h(v)$标识。这种情况如图11-23所示。

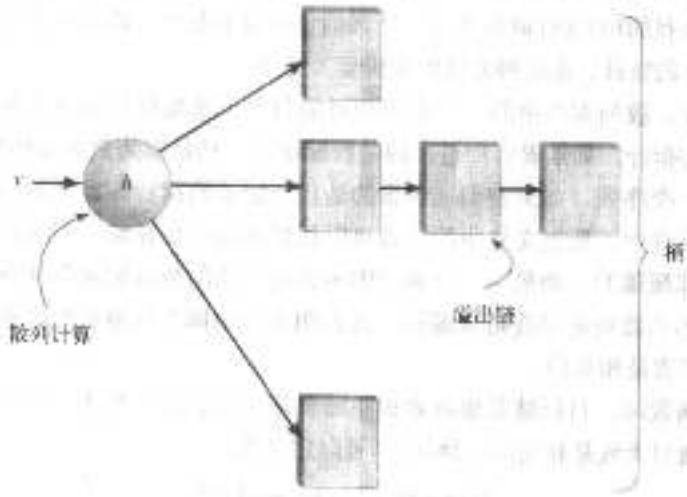


图11-23 一个散列索引的示意图

对查找键值为v索引项进行等值搜索时，先计算$h(v)$，找到存储在相关页上的桶，然后扫描桶的内容来定位查找键值为v的索引项（如果有的话）。因为没有其他的桶具有包含同样键值的项，所以如果在桶中没有找到相应的索引项，那它就不在文件中。因此不需要维护类似于树的索引结构，目标索引项的等值查找可以通过只有一次I/O操作的检索来实现（假设没有溢出链）。

和树索引相比，正确设计的散列索引可以使等值查找更为有效。因为对一个树索引来说，在到达叶子级之前，必须访问多个索引级的页。但如果要执行一连串的等值查找，可能树索引更加合适。例如，假设有必要搜索查找键值为$k_0, k_1, \cdots k_n$的记录，该序列在查找键值上是有序的（即$k_i<k_{i+1}$）。序列的顺序表示检索记录的顺序。因为一棵索引树上的叶子索引是按键值排序的，当检索索引项时，高速缓存的命中率就高。但对散列索引而言，每一个查找键值可能散列在不同的桶中，因而在检索序列中连续的索引项时，高速缓存命中率就会很低。这个

例子表明：在评价哪种索引能够改善应用的性能时，应从整体上进行考虑。

尽管散列索引与等值搜索相比，优势很明显，但树索引通常更加适用，因为树索引有多种用途：散列索引不支持范围和部分键值查找。它不支持部分键值查找是因为散列函数必须应用于整个键。为理解它不能有效地支持范围查找的原因，考虑前面所讨论过的学生表。假设使用一个散列索引，从文件中找出名字在paul和tom之间的所有学生的记录，散列得出的结论是没有名为paul的学生，并找出名为tom的索引项。但为定位这一查找范围内的相应的索引项，这是不够的，因为我们无法运用散列函数来搜索这一范围内的每一个可能的值——仅有一小部分可能出现在数据库中。在某一范围内，连续的索引项随机地分布在各个桶中，因而对这样的一个范围查找所花费的代价是和在这一范围内索引项的数目成比例的。在最坏的情况下（当文件中索引项的数目超出页的数目时），使用索引后的代价可能大于只是扫描整个文件所要付出的代价！

相反，使用聚簇ISAM或B^+树搜索范围内的数据记录时，先用索引定位范围内的第一条数据记录，然后进行简单的扫描。进行简单的扫描是可能的，因为文件中的数据记录是按查找键顺序维护的。查找的代价和这个范围内的页数成正比，再加上查找索引的代价。最重要的差别在于，B^+树查找的I/O代价取决于这一范围内叶子页的数目，而散列索引查找的代价取决于这一范围内记录的数目，而这种方法的代价要大得多。

和树索引一样，散列索引中的一个索引项可能包含一条数据记录或者存储一个指向数据文件中数据记录的指针。如果索引项包含的是数据记录，桶是作为数据文件的一种存储结构：数据文件是桶的一个序列。如果索引项包含的是指向记录的指针，那么索引项作为桶的序列保存在一个索引文件中，数据文件中的记录可以按任意的顺序存储。在这种情况下，散列索引是辅助索引和**非聚簇**的。如果同一个桶中的索引项引用的数据记录存储在相同的或者是相邻的磁盘块上，那么散列索引就是**聚簇**的。这表明含有相同查找键值的数据记录在大容量存储设备上的物理位置是相邻的。

在选取散列函数时，目标就是要将索引项随机地分配在各个桶中，这样，索引表上的实例在每个桶中的数目大致是相同的。例如，h可以定义为：

$$h(v) = (a * v + b) \mod M$$

其中，a和b是常数，用于优化函数随机分布查找键值的方式，M是桶的数目，v是一个字符串，在计算散列值时将它当作二进制数进行处理。为简化起见，假设以后所有的算法中，M是2的倍数（实际上它是一个比较大的素数）。

我们刚才所描述的索引模式称作**静态散列**（static hash），因为在索引被创建时，M是固定的。静态散列表的效率取决于每个桶中的索引适于放在一个页面上的程度。一个桶中索引的数目是和M成反比的：桶越少，桶的平均填充率就越高。桶的平均填充率越高，一个桶能放在一个页上的可能性越低，因而有必要使用溢出页。对索引操作的效率而言，M的选择是关键。

如果Φ是一个页面所能容纳的最大索引数，L是总的索引数，选择值为L/Φ的M可以产生只有一个溢出页的桶。一方面，h通常不能让索引精确地分布在各个桶中，所以我们希望某些桶可以分配到多于Φ条的索引。另一方面，如果表是随时间增长的，那么桶有可能溢出。处

理这种增长的一种解决办法就是使用一个小于1的装填因子来扩大桶的数目。每个桶的平均填充度为Φ与装填因子之积，以便那些装填数大于平均数的桶能容纳在一个页中。M就变成L/(Φ*装填因子)。装填因子小于0.5是不合理的，扩大M的缺点是空间的需求增大，因为有些桶中只有为数很少的几条索引。

这一技术能够减少溢出问题，但没有完全解决溢出问题，部分原因是因为一些表的增长事先是不能预测的。因而必须处理桶的溢出，这可以使用如图11-23所示的溢出链来解决。但是，对一个ISAM索引来说，溢出链是低效的。因为对一个等值查找，必须扫描所有的桶，查找一个存储在n个页面上的桶需要n次I/O操作。这在一个理想的查找代价上乘上n，但研究表明，对一个好的散列函数而言，n=1.2。

11.6.2　动态散列算法

正如调整树索引得到B^+树以适应动态表一样，也可以调整静态散列得到动态散列方案来处理同样的问题。动态散列的目标就是动态地改变桶的数目，从而在插入行和删除行时减少或消除溢出链。有两种动态散列算法倍受关注，它们是可扩展的散列和线性散列，下面将简短地讨论这两种算法。

静态散列使用一个固定的散列函数h，它把所有可能的查找键值分成S_i（$1 \leqslant i \leqslant M$）个子集，将每一个子集映射到一个桶$B_i$，$S_i$中的每一个元素$v$可用$h(v)$来标识桶$B_i$。动态散列方案可以将$S_i$在运行过程中被分成不相邻的$S_i'$和$S_i''$，将$B_i$分成$B_i'$和$B_i''$，这样，$S_i'$上的元素映射到桶$B_i'$，$S_i''$上的元素映射到桶$B_i''$。通过减少映射到桶上的键值数，一个分裂可以将一个溢出的桶用两个非满的桶来代替。映射的变化预示在分裂一个桶时（或合并两个桶）考虑了散列函数的变化。可扩展的散列和线性散列以两种不同的方法来完成这个映射。

1. 可扩展的散列

可扩展的散列使用一个基于散列函数h的散列函数序列$h_0, h_1, \cdots, h_b$，它把查找键值散列成一个b位的整数。对每一个k，$0 \leqslant k \leqslant b$，$h_k(v)$是由$h(v)$的最后$k$位组成的一个整数。其数学表达式为

$$h_k(v) = h(v) \bmod 2^k$$

因此，每个函数h_k的范围是h_{k-1}的两倍。在任何给定的时间内，序列中的一个特定函数h_k支持所有的查询。

和静态散列不一样，动态散列使用一个二次散列来确定与桶相关联的查找键值v。如图11-24所示，这种映射是通过一个目录来间接执行的。由$h_k(v)$所产生的值作为一条索引放在目录中，目录中的指针指向与v相关联的桶。目录中不同项可能指向同一个桶，所以即使$h_k(v1)$和$h_k(v2)$的值不同，$v1$和$v2$也可能映射到同一个桶。

为理解这一过程，假设散列函数h是0到$2^{10}-1$之间的一个整数集。在图11-24中，我们假设一个桶的页能容纳两个索引项。该图描述某一时刻用h_2对查找键值进行散列的算法（因此，目录中包含有2^2=4个索引项）。因为h_2仅用h范围内的值的最后两位，所以数字可以由下面的函数h来产生:

v	h(v)
pete	1001111010
mary	0100000000
jane	1100011110
bill	0100000000
john	0001101001
vince	1101110101
karen	0000110111

因而h将pete散列成1001111010，但h_2只用最后的两位，表示目录的第三项，它指向B_2。

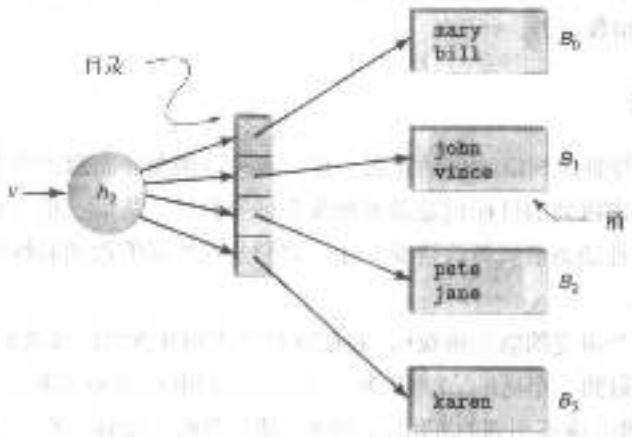


图11-24　可扩展的散列通过一个目录将散列结果映射到桶

现在我们假设要插入一个记录sol到表中，h(sol)=0001010001，h_2将john、vince、sol都映射到B_1，导致溢出。不是创建一个溢出链，而是可扩展的散列将B_1分裂，产生一个新桶B_5，如图11-25所示。为容纳5个桶，有必要使用一个包含多于4个值的散列函数，因此用h_3来代替h_2。由于h_3的最高一位h_3(john)=0，h_3(vince)=1是不同的（而低两位01是相同的），h_3通过把john和vince映射到不同的桶而避免溢出（h_2做不到）。在图11-25中，注意，如果v1和v2是B_1（或B_5）的元素，则h(v1)和h(v2)的最后三位是相等的。

需要一个目录来补充只有B_1需要分裂的事实。实际上，h_3不仅把john和vince区分开来，而且对pete(010)和jane(110)产生不同的值。因此，如果没有目录，那么当用h_3替换h_2时，就有必要对B_2进行分裂。通过在散列计算和桶之间插入一个目录，我们可以将pete和jane都映射到B_2，这是通过在目录中的第三项(010)和第七项(110)中存储一个指向B_2的指针来实现的。结果是，与B_1和B_5相反，如果v1和v2都是B_2的元素，那么h(v1)和h(v2)只是在它们的最后两位上相等。

把图11-24转换到图11-25的算法相当简单：

1) 分配一个新桶B'，使用序列中的下一个散列函数将溢出桶B中的内容分裂到B和B'中。

2) 通过与复制旧目录中相关内容来创建一个新目录。

3) 将拷贝中指向B的指针用指向B'的指针来代替。

因此，除第二项和第六项的指针外（这是桶被分裂后的两部分），图11-25中目录的两部分是一致的。

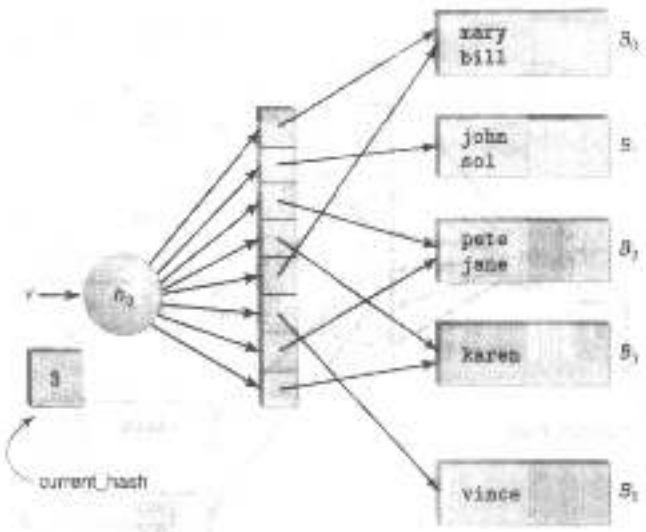


图11-25 使用可扩展的散列，图11-24中的桶B_1被分裂

为给出算法的一个完整的描述，我们必须处理一些其他问题。首先，我们怎么能够知道执行查找操作时使用的是序列中的哪一个散列函数？这是很容易实现的。我们只需要将当前散列函数的索引与目录一起存储即可。我们假设用一个变量current_hash来完成这一功能，它被初始化为0（仅有一个目录项）。一般来说，目录项的数目是$2^{current_hash}$。在图11-24中，current_hash的值是2（没有显示出来），在图11-25中，current_hash的值是3。

如果下面要插入关于judy的一行，h(judy)=.110000110，会出现一个更加复杂的问题，这是我们应该考虑的。在图11-25中，judy被映射到B_2，这导致桶溢出，必须对桶进行分裂。但这却不同于导致我们将h_2替换成h_3的情形，因为h_3本身能够将映射到B_2的索引项区分开来（B_2中索引项的散列值只在它们的最后两位上是相等的）。在这个例子中，h_3将judy和jane（110）与pete(010)区分开来。所以不是使用一个新的散列函数来处理溢出，我们只需要为judy和jane创建一个新的桶，然后对目录中相应的指针进行更新。如图11-26所示，新桶标记为B_6。

这是目录此时不必扩大的一个原因。每次扩大目录时，目录能存储该时刻指向每一个桶的指针，事实上，仅有一个桶分裂。因此，在扩展一个目录时，几乎只有一个指针指向原来的桶。图11-25中的目录是为B_1的分裂而创建的，所以在新目录中只有一项101指向新桶。因为项010和110都指向B_2，所以目录不需要进一步扩大就能适应B_2的分裂。

通过存储每一个桶和桶已经分裂的次数，对我们可以检测出这种情形。我们把这个值称为bucket level，并与一个变量bucket_level[i]相关联。在开始时，bucket_level[0]的值为0，当B_i分裂时，我们使用bucket_level[i]+1来作为B_i的bucket level和新产生的桶。因为每次目录扩大时，B_i不进行分裂，我们让目录中指向B_i的指针加倍。目录中指向B_i的指针数是$2^{current_hash-bucket_level[i]}$。而且，当一个桶分裂时，索引项也分配到两个分裂后的桶中，所以如果v1和v2都是B_i的元素，那么h(v1)和h(v2)在它们的最后的bucket_level[i]位上是相等的。

在图11-25和图11-26中，bucket_level[1]都是3（在图11-25中没有表示出来），而图11-25中的bucket_level[2]是2，图11-26中的bucket_level[2]增加到3。

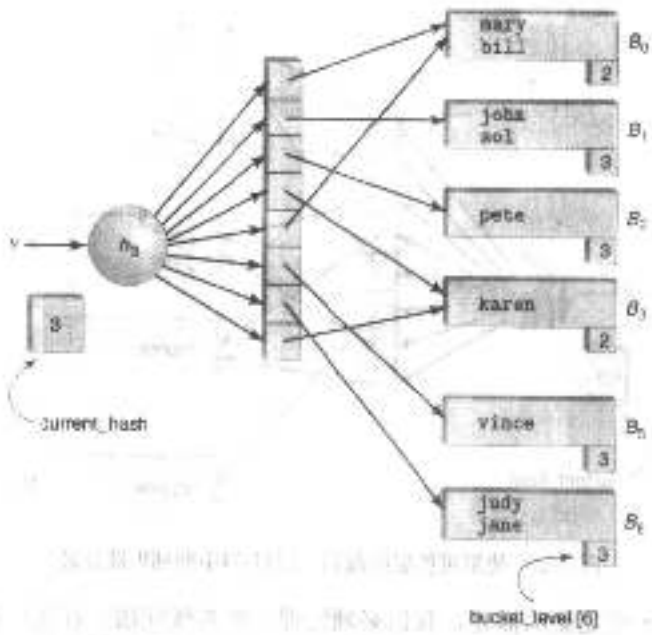


图11-26　不扩充目录，分裂图11-25中的桶B_5

使用分裂算法的逆可以合并两个桶。如果删除一个索引项导致一个桶B'为空，而B'和B''是在B分裂时产生的，则可以让目录中指向B'的指针指向B''，让B''的bucket_level减1就可以释放B'。另外，当合并之后所产生的目录上半部分和下半部分相同时，其中一半可以释放，current_hash减1。一般不进行合并操作，因为假设尽管一个表暂时会收缩，但从长远来看，它还是可能会扩大，所以合并最终会跟随着一个分裂。

当索引项的数目增加时，可扩展的散列消除大多数用静态散列开发的溢出链，但它有几个缺点。第一，它需要额外的空间来存储目录。第二，由目录间接定位缓冲区需要额外的时间开销。如果目录很小，它可以保存在主存中，所以这些缺点都不是主要问题。这时，目录的访问不需要额外的I/O操作。但一个目录变得相当大的时候，如果它不能放在主存中，那么可扩展散列的I/O开销是静态散列的两倍。最后，分裂桶并不能将一个桶中的内容分开而消除溢出。例如，当一个溢出由多个具有相同查找键值的索引项引起时，分裂并不能消除溢出。

2. 线性散列

由于可扩展散列的缺点是由目录的引入而引起的，我们自然会想到开发不需要目录的动态散列模式。可扩展散列需要一个目录是因为当一个桶分裂时，有必要切换到一个有更大范围的散列函数：存储在不同桶中的查找键值必须散列成不同的值。因此，h_{i+1}所包含的元素数是h_i的两倍，所以$h_i(v)$和$h_{i+1}(v)$是不相等的。但如果v是非分裂桶中的一个元素，那么在分裂前和分裂后，索引必须能让查询找到这个桶。目录可以解决这个问题。

这个问题可以通过不使用目录来解决吗？一种可能的解决办法就使用相同的散列函数序列$h_0, h_1, \cdots, h_n$，这和可扩展的散列一样，但不同的是h_i和h_{i+1}同时使用：h_i用于没有分裂的桶中的值，h_{i+1}用于分裂的桶中的值。这种方法的优点是：它为分裂前和分裂后与分裂无关的桶的查找键值提供相同的映射。诀窍在于当通过索引开始一个查找时，它能告诉我们使用的是哪

一个散列函数。

使用线性散列，桶的分裂分为多个阶段。处于开始阶段的桶连续分裂，所以最后阶段桶的数目是开始阶段桶数目的两倍。这一过程如图11-27所示。阶段是有编号的，当前阶段的编号保存在变量current_stage中。图中current_stage的值是i，这表明使用的散列函数是h_i和h_{i+1}。在开始时，current_stage的值为0，只有一个桶。变量next表示序列中下一个要分裂的桶。

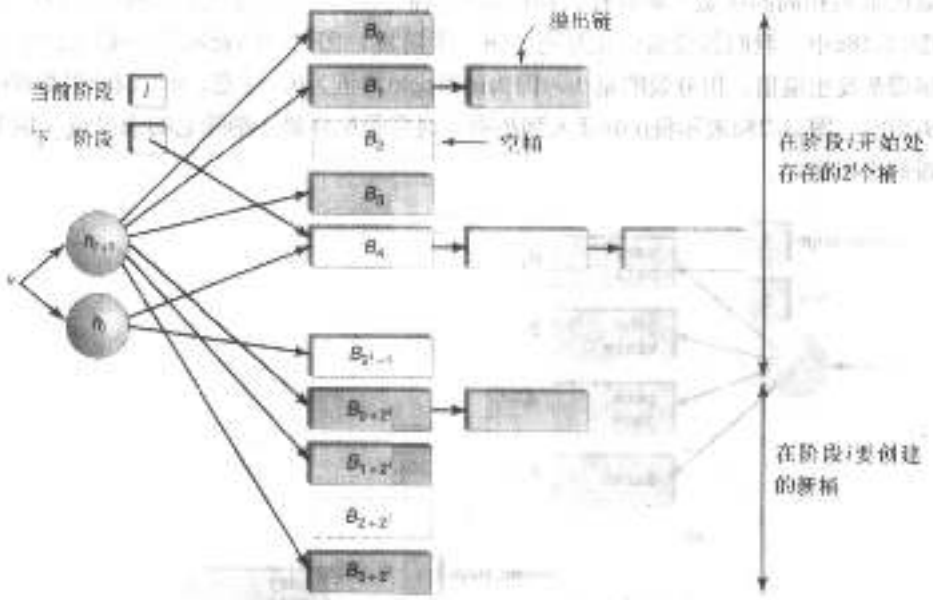


图11-27 线性散列连续地对桶进行分裂，阴影部分的桶是通过h_{i+1}来访问的

当第i个阶段开始时，有2^i个桶，范围为$\{0\cdots 2^i-1\}$的散列函数h_i散列查找键值。定期做出分裂桶的决策（我们暂时回到这个问题上来），next的值用来决定对开始阶段的哪一个桶进行分裂。在开始阶段，next初始化为0，在发生分裂时加1。因此，桶连续地进行分裂，next把那些当前阶段已经分裂的桶和那些没有分裂的桶区分开来。由于next指向下一个要分裂的桶，索引号小于next的桶在这一阶段已经分裂（分裂的桶在图中以阴影部分的形式出现）。当有2^i个桶存在时，第i个阶段已经完成。

当桶B_{next}分裂时，新创建的桶有next+2^i条索引，B_{next}的元素使用h_{i+1}来分割：如果v是B_{next}的一个元素，且$h_{i+1}(v)=h_i(v)$，那么它将留在原来的桶中，否则就移到B_{next+2^i}中。当next的值达到2^i时，一个新的阶段开始，将next重新设置为0，current_stage加1。

当一个查找作用在查找键v上时，就计算$h_i(v)$。如果$next \leqslant h_i(v)<2^i$，就为了$v$扫描$h_i(v)$表示的桶。如果$0 \leqslant h_i(v)<next$，说明由$h_i(v)$表示的桶已经分裂，$h_i(v)$不能提供足够的信息来确定是应该扫描$B_{h_i(v)}$桶还是$B_{h_i(v)+2^i}$桶。但在分裂时，使用$h_{i+1}$来对元素进行分割，我们可以使用$h_{i+1}(v)$确定在哪个桶中查找。

重要的一点是，在线性散列中，被分割的桶不一定溢出。当一个桶B_j溢出时，我们可能决定对桶进行分裂，但被分裂的桶是B_{next}，next的值可能不同于j。注意，在图11-27中，next=4，这表明B_0、B_1、B_2和B_3已经分裂。特别是当B_2为空，没有发生溢出时，也被分裂。所以，线性散列不能消除溢出链（这样，就必须为B_j创建一个溢出页），但最终B_j将被分裂，因

为处在当前开始阶段的每一个桶在这一阶段结束时都要分裂。所以尽管可能要为一个桶B创建一个溢出链，但链通常很短，一般在下次分裂B时就消除。一个溢出页的平均生命期随着频繁的分裂而缩短。尽管这样做的代价只需要很少的空间，因为每一个阶段，没有溢出的桶都要分裂。

假设分裂每次都在一个溢出页被创建之后发生。图11-28显示当一个线性索引从图11-24的与可扩展的散列相同的状态开始并经历相同的后续插入时，一个线性索引所经历的一系列阶段。在图11-28a中，我们假设索引正好进入第二个阶段，图11-28b表示将sol插入到B_1之后的结果。尽管B_1发生溢出，但分裂的是B_0，因为此时next的值为0。注意，mary和bill都被h_1散列到B_0，B_4为空。图11-28c表示将judy插入到B_3中，现在是B_1分裂，消除它的溢出链，但为B_2创建一个新的溢出链。

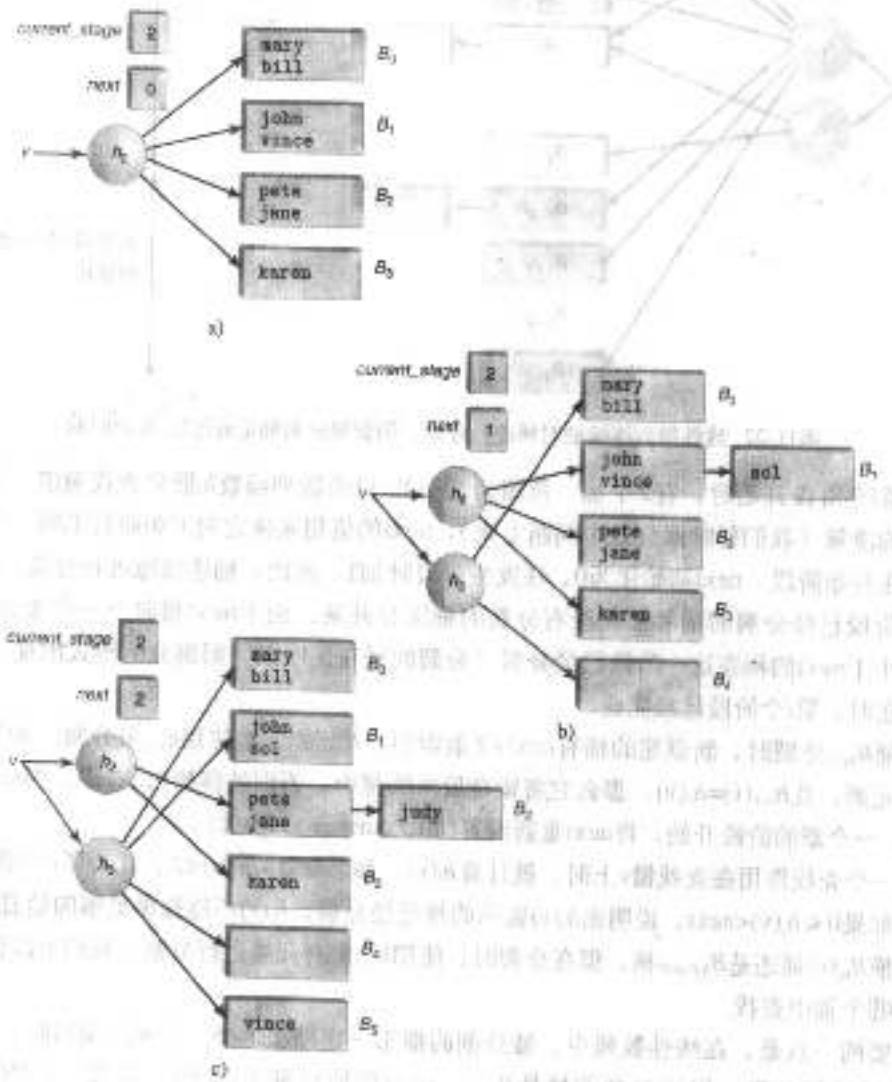


图11-28　a) 在开始阶段一个线性散列索引的状态；
b) 插入sol之后的状态；c) 插入judy之后的状态

可以看到，线性散列分裂的不是所要分裂的桶，这就是为避免使用目录而付出的代价。按顺序分裂桶很容易将已分裂的桶和未分裂的桶区分开来。尽管线性散列有溢出链，但不必为目录的存取而付出额外的开销。

11.7 特殊用途的索引

到现在为止，所介绍的索引结构可用于多种情形，但还有一些索引结构只适用于一些特殊的情况，但能节约大量的存储空间和处理时间。我们来讨论两种这样的技术：位图索引和联接索引。

11.7.1 位图索引

位图索引（Bitmap index）[O'Neil 1987]是作为一个或多个位向量来实现的。这对一些只能取很少值的属性来说非常合适。例如，关系PERSON上的Sex属性只能取两个值：Male和Female。假设PERSON共有40 000行，PERSON上的位图索引是一个含有两位的向量，其中一位用于Sex可能的值，每一个位向量包含40 000个位，另一位用于PERSON的每一行。因此，如果PERSON中的第*i*行的性别属性值为Male，那么Male位向量的第*i*位是1。结果，我们可以通过扫描Male位向量来确定属性为Male的行数。如果给定行数，我们就能计算出从数据文件开始端的位移，直接从磁盘上访问想要找的行。

注意，例子中存储索引的空间正好是80 000位，即10K字节，它是可以放在主存中的。因为全部在主存中实现，所以查找这样的索引可以通过顺序扫描来快速实现。

你可能已经注意到，和位图索引实际需要的空间相比，位图索引浪费了更多的空间。事实上，编码Sex属性我们使用两位，而实际只要1位就能编码所有可能的值。这样做的原因是，位图索引的目的是提高效率，特别是提高在两个属性或多个属性上进行选择时的效率。为更加清楚地进行说明，假设关系PERSON表示一次健康调查的结果，其中有属性Smoker和HasHeartDisease，这两个属性只接受Yes和No两个属性值。一个查询是在一特定的年龄群体中找出吸烟但没有心脏病的男性公民人数。作为查询的一部分，我们有以下的条件选择：

```
Sex = 'Male' AND Smoker = 'Yes' AND HasHeartDisease = 'No'
```

如果以上的三个属性都有位索引，那么我们就能很容易地找到满足这一条件的所有元组：在Sex属性为Male、Smoker属性为Yes、HasHeartDisease属性为No的值的位串取逻辑AND，满足条件的元组在结果位串中相应的位上取1。

更一般的情况是，位图索引能处理包含OR和NOT的条件选择——所有我们要做的就是计算相应位串中合适位的布尔值。例如，假设Age属性有一个120位的位图索引，PERSON中的每一条记录需要120位的开销。这是可以接受的，无论如何，PERSON每条记录的一个常规索引至少要8个字节（64位）。现在，我们应能高效地找到年龄在50岁～80岁之间吸烟的男性公民，而他们从来没有犯过心脏病。首先计算以上所描述的三个位串的逻辑AND值，然后计算Age属性上位图索引中年龄在50岁～80岁之间相应位串的逻辑OR值，最后计算这两个结果的逻辑AND值。还有，位串的最后一位给出我们所要查询记录的ID值。

正如第19章将要讨论的，位图索引在数据挖掘和OLAP（联机分析处理）应用中起着重要

的作用。它们受欢迎的原因是，这样的一些应用通常处理低基数属性，例如：性别、年龄、公司地点、财政期等属性，而且这些应用所操作的数据是相对静止的，当数据频繁变化时，位图索引的维护是比较昂贵的（考虑数据文件记录的插入和删除对一个位图索引的修改）。

11.7.2 联结索引

假设我们要加快两个关系的等值联结速度，如$P\bowtie_{A=B}Q$。**联结索引**[Valduriez 1987]是一个由形如<p,q>的对所组成的一个集合，这里的p是关系P中的一个元组t的rid，q是关系Q中的一个元组s的rid，此时t.A=s.B（即这两个元组联结时）。

一个联结索引通常按词典顺序升序排列。因此，<3,3>在<4,2>之前，这样的索引也可以组织成B^+树或散列表的形式。一棵B^+树中各项的查找键值是关系P中行的rid，为查找到关系Q中与关系P的一行p联结的所有行的rid，p的叶子索引（如果它们存在的话）包含请求的rid。类似地，为找到包含查找项的桶，散列索引对p进行散列。这些项包含关系Q中与关系P的一行p联结的所有行元素。

一个联结索引可以被看作是一个预先计算的联结，它以压缩的形式保存。尽管计算它首先可能需要执行常规联结，它的优点（除它的压缩尺寸）在于它可以递增地保存：当有新的元组添加到P或Q中时，新的索引项可以添加到联结索引，而不必从头开始重新计算整个索引（练习11-24）。

给定一个联结索引J，通过简单地扫描J来寻找匹配的p和q，系统能够计算出$P\bowtie_{A=B}Q$。注意，如果J在P的字段上是有序的，则相应于P的元组的rid是以升序的形式出现的。因此，联结是以一次扫描索引和P来完成计算的。对Q中一个元组的每一个rid，都要访问一次Q。

值得一提的是，联结索引的一种变体被称为**位图联结索引**（bitmapped join index）[O'Neil and Graefe 1995]。还记得一个联结索引通常组织成一个顺序文件，一个散列文件，或是一棵B^+树，以便很容易地找到Q中与关系P的一行p联结的所有行的元素。我们可以以下列方式合并Q中的这些rid：对P中的每一个rid p，在J中用<p,Q中相匹配的位图>来替换所有的<p,q>形式的元组。当且仅当Q中的第i个元组与P中的元组p相联结时，位图的第i位为1。这样看起来好像浪费了大量的空间，因为每一个位图的位数是和Q中的元组数相等的。但位图容易被压缩，所以这不是主要问题。另一方面，位图联结索引能加速多个关系的联结，正如13.5节所阐述的，用空间来换时间是值得的。

11.8 调整问题：为一个应用选择索引

每一类索引都能改善某类特定查询的性能，因此，在选择支持某种应用的索引时，了解可能要执行的查询和它们执行的频率是非常重要的。为一个很少执行的查询添加一个索引以改善其性能是不明智的，因为每添加一个索引就会添加额外的开销，对于那些更新数据库的操作来说更是如此。

例如，把StudId指定为一个参数的查询（11.2）是经常执行的，增加一个索引就能确保查询的快速响应。这样一个索引的查找键是从WHERE子句（不是SELECT子句）中的列选择得到的，因为这些列定义查找的方向。这样，在查找键值为StudId的TRANSCRIPT表上的一个索引是非常有用的。与扫描整个表来查找StudId为111111111的行相反，定位机制先找到包含查找

键值的索引项。每一个索引项既包含记录又包含记录的元素。因此，当一个扫描要求检索数据文件一半（平均）以上的页时，通过索引访问仅需一次检索。另一方面，对查询（11.3），在StudId上建立索引是无益的。

但如果我们要支持查询

```
SELECT   T.Grade
FROM     TRANSCRIPT T                                          (11.5)
WHERE    T.StudId = '111111111' AND T.Semester = 'F1997'
```

我们可以选择在StudId和Semester上创建一个多属性索引。但如果我们选择把索引限制在一个属性上（或许是为支持其他查询，避免太多的索引），那么哪一个属性是它的查找键呢？一般来说，选择最具可选性的属性。如果在StudId上创建一个索引，那么数据库系统使用它提取所有的行来找一名学生，然后搜索结果，并把这些行作为目标Semester保存下来。在搜索Semester时，这会比提取所有的行更加有效。因为对给定的学生，这里有更多的行是针对给定的学期的。一般来说，仅有为数不多的几个值的属性（例如Sex）是不可能作为查找键的。

在选择查找键时，可参考下面的几点指导:

1) 用作联结条件的一列可用作索引。

2) 用在ORDERBY子句中，一列上的聚簇B^+树索引可用来按特定顺序检索行。

3) 候选键上的索引能有效地加强唯一性约束。

4) 用于范围查找中的聚簇B^+树索引能快速检索特定范围内的元素。

如果性能问题是吞吐量的问题，我们首先必须确定导致问题的查询类型。一种是经常使用的查询，另一种是长时间运行的查询，它访问多个表，并做大量的计算。对这两种情况，一旦查询的问题被确定，就可以添加合适的索引（或多个索引）可用来加速查询的执行。

正如我们将在24.3.1节中要看到的，索引通过增加允许的并发数来提高性能。一些并发控制使用加锁协议，其中包括为索引页加锁。如果存在合适的索引，协议就能避免对整个文件上锁。因为和锁住整个文件相比，对一个索引加锁可以减少限制，加锁增加数据库上并发执行的操作数，因而改善了性能。这类性能的改善有时是选择索引的基础。

11.9 参考书目

B树是在[Bayer and McCreight 1972]中介绍的，B^+树首先出现在[Knuth 1973]中。这本书的最新版本[Knuth 1998]涵盖本章介绍的大部分内容。散列作为一种数据结构首先在[Peterson 1957]讨论。线性散列是在[Litwin 1980]中提出的，可扩展的散列是在[Fagin et al. 1979]中提出来的。[Larson 1981]分析了顺序索引文件。

联结索引首先是在[Valduriez 1987]中提出的，[O'Neil 1987]第一次描述了位图索引。位图映射联结索引是在[O'Neil and Graefe 1995]中介绍的。

11.10 练习

11.1 说明下列机器的存储容量、扇区大小、页大小、寻道时间、旋转延迟和磁盘的传输时间。

a. 你的本地PC机

b. 你所在学校提供的服务器

11.2 试解释等值搜索和范围搜索的区别。

11.3 a. 对你所在的城市或城镇的电话号码簿，执行一个二分查找，试给出必须查找页数的一个上界。

b. 如果给电话号码簿每页的第一项准备一个索引，上面得出的数字会减少多少？（这条索引适合电话号码簿的这一页吗？）

c. 使用常见的方法（非正式的）做一个试验，查找本地电话号簿上的John Lewis，把这个数字和a中的数字进行比较。

11.4 试解释为什么一个文件只能有一个聚簇索引。

11.5 试解释为什么一个辅助的、非聚簇索引必须是稠密的。

11.6 一棵B^+树的最终结构取决于添加项的顺序吗？试对你的答案作出解释，并举出一个例子。

11.7 从一棵每个节点最多只能有两个查找键的空B^+树开始，按顺序插入下列键值，看树是如何增长的？

18,10,7,14,8,9,21

11.8 思考图11-29中某B^+树的一部分。

a. 不添加新的键，把所有内部节点都填充上。

b. 增加键bbb，说明树的变化。

c. 从结果b中删除键abc，说明树的变化。

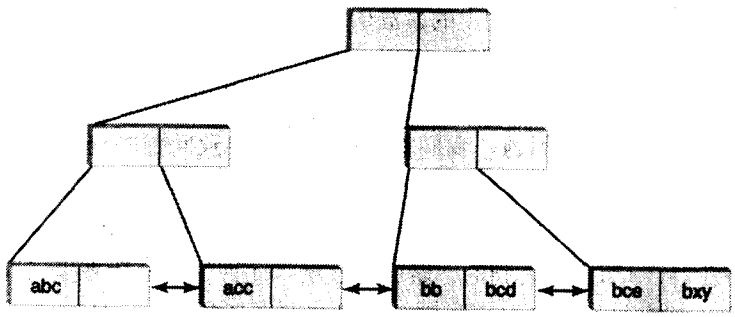


图11-29 某B^+树的一部分

11.9 思考图11-30的B^+树，假设它是从其他树插入一个键到叶子节点后，导致一个节点的分裂后得到的。树原来是什么样子？新插入的键是什么？这是唯一的解决办法吗？试对你的答案做出解释。

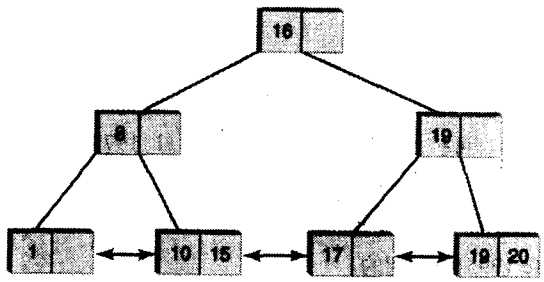


图11-30 B^+树

11.10 为一棵B^+树描述一个查找算法，其中的查找键不是候选键。假设在处理相同的键时，没有使用溢出页。

11.11 思考一个散列函数h，它把复合查找键的值作为参数，复合查找键是r属性的一个序列$a_1, a_2, \cdots, a_r$。如果h有下面的形式

$$h(a_1 \circ a_2 \circ \cdots \circ a_r) = h_1(a_1) \circ h_2(a_2) \circ \cdots \circ h_r(a_r)$$

这里h_i是属性a_i上的一个散列，其中$\circ$是串接符，h称为可以**分割的散列函数**（partitioned hash function）。对等值查找、部分键查找和范围查找，试描述这样一个散列函数的优点。

11.12 使用伪代码写出一棵B^+树的插入算法。

11.13 使用伪代码写出一棵B^+树的删除算法。

11.14 使用伪代码写出可扩展散列模式上插入和删除索引项的算法。

11.15 试给出下列例子的SQL语句，它们是：

a. 由于增加B^+树索引，结果使查找速度加快（如图11-21所示）。

b. 由于增加B^+树索引，结果使查找速度降低（如图11-21所示）。

11.16 对图11-21的索引，试画出插入alice、betty、carol、debbie、edith和zelda后的B^+树。

11.17 关系数据库中的一个表包含100 000行，每行需要200B的内存，一条SELECT语句返回表中某一属性上满足等值查找的所有行。试估计在这一属性上建立下列索引后完成这一查询所需的时间（单位为ms）。对页的大小、磁盘的访问时间等作出理想的估计。

a. 没有索引（堆文件）。

b. 一个静态散列索引（没有溢出页）。

c. 一个聚簇的、非集成的B^+树索引。

11.18 对上题中的表，当使用下列索引时，试估计插入一行的时间（单位为ms）：

a. 没有索引（文件按查找键排序）。

b. 一个静态散列索引（没有溢出页）。

c. 一个聚簇的、非集成的B^+树索引（不需要对节点分裂）。

11.19 对练习11.17中的表，当使用下列索引时，试估计更新一行的查找键值所需要的时间：

a. 没有索引（文件按查找键排序）。

b. 静态散列索引（区别于原来行所在的页，更新的行处在不同的桶中，但不需要溢出页）。

c. B^+树索引（区别于原来行所在的页，更新的行处在不同的页中，但不需要分裂节点）。

11.20 试估计存储练习11.17中B^+树所需的空间，并将它和存储表所需的空间进行比较。

11.21 试解释你的本地数据库管理系统所支持的索引类型，并给出创建每一种索引的命令。

11.22 试为学生注册系统的表设计索引，并给出所有设计决策的依据（包括那些应该使用索引但没有使用索引的情况）。

11.23 在下列情况下，试解释使用小于1的装填因子的依据：

a. 有序文件

b. ISAM索引

c. B^+树索引

d. 散列索引

11.24 试为保持递增的联结索引设计一个算法，也就是说，在加入一个新的元组到关系中时，不需要从头开始重新计算整个联结索引。

第12章 案例研究：实现学生注册系统

由于客户已经认可学生注册系统的规格说明文档，所以可以继续进行这个项目的下一步工作，即设计。

12.1 设计文档

规格说明描述系统应该是什么样子，而设计描述系统如何实现其设计的功能，因而设计一个事务处理系统会包括如下步骤：

- 系统使用的全局数据结构的声明，包括数据库模式和应用程序在事务调用之间保存的数据结构。
- 将规格说明文档所描述的每个交互分解成事务和过程。
- 详细描述系统中每个模块、对象、过程和事务的行为。

设计阶段的结果呈现在设计文档中，从某种意义上来说，它是规格说明文档的延伸。规格说明文档详细描述了系统的功能，而设计文档则详细描述了如何实现这些功能。

设计过程本身通常被视为是项目实现过程中最富有创造性的一部分，好的设计应该是简单而优雅的。设计人员阐明和评价可以实现所需功能的各种方案（例如，各种各样表的设计），然后根据判断和经验做出决策，这些决策将极大地影响系统的实现以及最终的性能。但是，尽管做出决策是件愉快的事情，但是在设计文档中详细记录却是设计人员最为单调乏味的任务之一，然而这个过程是必不可少的。

设计文档的使用者包括：

- 编程人员，他们把设计文档作为唯一的编程信息来源。
- 质量控制人员，他们借助设计文档和规格说明文档来设计测试，并在发现错误的时候确定是哪里出了问题。
- 维护人员，他们会在以后的某个时间使用设计文档来提升系统性能。

设计过程中的一个重要部分是做出全局的决策（即那些会影响多个事务和过程的决策），这样在以后的编程阶段，编程人员不需要知道整个系统的情况（除了那些在设计文档中提供的）就可以实现每个事务和过程。如果设计文档不完整，但某个事务或者过程的编程人员根据它做出全局的决策，那么这个决策很可能会与另一个事务或者过程的编程人员对同样全局问题做出的决策不一致，从而导致错误。

假设某个交互I被分解成两个事务T_1和T_2，事务T_1读取数据库项X，将它的值存储在变量x中，对于I来说x是全局的。当提交T_1后，T_2被初始化并使用x的值，但是X的值可能已经被并发的另一个交互更新了。如果T_2仍认定x是X的当前值，那么执行就会出错。

在设计过程中，必须做出全局决策决定是否允许T_1和T_2之间通信，或者是否需要将T_1和T_2放在同一个事务中。假定执行T_1和T_2之间不可能改变通信的值，那么设计者或许会允许通信。

例如在学生注册系统中，X可以是Semester Id，虽然一个并行的交互可能改变Semester Id，但是设计人员认为这种情况发生的可能性很小，所以可以使用这样的方式通信。这个决策应该记录在设计文档中，以便每个程序员能够理解，同时如果报告错误并发现假设是不正确的，那么在必要的时候就应该进行修改。

12.1.1 文档结构

一个事务处理系统的设计文档包括下面几个部分：

A. **标题、作者、日期以及版本号。**

B. **引言**　简单介绍系统的目标。

C. **相关文档**　包括对需求和规格说明文档的引用，以及在设计或者实现中使用的其他文档，例如Visual Basic用户手册、ODBC规格说明文档或者设计时使用的某个对象库的规格说明文档。

D. **高级设计**　设计的非正式描述，使得读者能很容易地理解后面部分的详细说明。有关项目包括：

1) 将系统分解成模块和对象。

2) 将规格说明文档中定义的交互分解成事务和过程。

3) 过程调用树。

4) 数据库设计的E-R图，包括设计图表的依据。

这一部分中还要记录设计决策的依据，例如：

5) 为什么使用这种方法将会话分解成事务，而不使用另一种也许更直观的方式？

6) 为什么表要使用这种方式进行规范化、反规范化或者分片[⊖]，以满足性能要求？

7) 为什么要定义某些索引？

8) 为什么某些完整性约束要由事务进行检查，而不是嵌入在数据库模式中？

9) 为什么可以在更低的隔离级别上执行事务？

10) 其他的以便达到性能要求的决策。

E. **数据库模式和其他全局数据结构的声明**

1) *数据库模式*

a. 一组完整的可编译语句，用来声明包括表、索引、域名规格说明、断言和访问权限在内的数据库模式，要记录每张表每个列的用途和选择每个索引的依据。

b. 完整性约束列表，并描述在哪里执行约束：是在模式中执行，还是在单独的事务中执行。

2) *全局数据结构*

其他任何全局数据结构的完整可编译的声明，例如应用程序保存的由其创建的事务所使用的数据结构。必须记录每个项目的用途及其值的限制。

F. **图形用户接口**　如果在项目的规格说明阶段已经使用应用生成器设计和实现图形用户接口，并且在规格说明中对接口进行了描述，那么只需要引用相应的文档即可。否则，任何

⊖ 分片将在第18章讨论。

缺失的用户接口细节必须在这里提供，包括所有事件、对象及其方法的规格说明。

G. 每个事务和过程的详细说明

1) *事务或者过程名称。*

2) *描述*　用一个语句非正式地描述事务或者过程完成什么工作，例如“这个事务检查这个学生完成指定课程的所有预备课程后允许其注册这门课程”，描述的目的是大致说明事务的用途，而不是详细说明它的功能。

3) *参数*　事务或者过程的输入、输出参数，记录每个参数的类型和用途。

4) *返回值*　事务或者过程的返回值，记录每个返回值的类型和用途。

5) *调用口*　调用事务的过程或者GUI事件，例如在某个表单对象上点击鼠标时会调用一个事务。

6) *调用*　事务所调用的过程，包括它引发的事件和产生的异常。这两项在需要改变设计和代码的时候是有用的，设计人员和编码人员必须将变化传播到整个设计中。

7) *先决条件*　事务或者过程所做的假设（不需要在运行的时候检查），包括数据库的状态、全局的数据结构以及它运行时的参数。例如，学生注册课程的事务可以假设先前执行的事务已经对这名学生进行了认证。再比如，鼠标点击事件调用的过程可以假定所需的输入参数已经被用户存储在某个表单对象的特定字段中。在大型系统实现中，大部分全局错误出现的原因在于对先决条件理解不同。

8) *隔离级别*　执行事务所处的隔离级别。

9) *动作*

a. 对事务或者过程所采取的动作的文字描述，这段描述可能有一两段而不是象先前那样只有一句话，目的是帮助程序员编写代码。

b. 所访问的数据库表和全局数据结构，以及事务和过程假定会发生的变化。例如，在一个注册事务成功完成之后，学生一定会出现在相应的注册课程的表中，同时这门课程的注册人数也一定会增加。

c. 错误情形

- 正确性检查，事务或者过程必须保证参数、全局数据结构或者数据库的正确性。例如，要求学生注册课程事务检查学生已经完成（或者正在进行）该课程所有预备课程。一个访问允许空的字段的事务（例如注册事务读课程的天数和时间）在执行的时候要检查该字段是否为空。必须描述当检查失败的时候所采取的动作。
- 事务更新时系统将要做的自动约束检查，以及更新失败时应采取的动作。
- 其他任何可能发生的错误和异常情况以及这些情况下所采取的动作。例如，当数据库CONNECT语句失败时会发生什么？什么情况下会抛出异常，是什么样的异常？

d. 各种情形下显示的表单，例如在成功完成的情形下或者发生指定错误情形下应显示的表单。

12.1.2 设计评审

在设计的最后阶段，通常会举行正式的**设计评审**（design review），所有的设计和质量保证人员将参加这一活动，可能还包括一些其他人员。参加的人员在评审前会得到最新的规格

说明文档和设计文档，要求在开会之前已经研究过这些文档。

设计人员做正式陈述，要求与会人员能理解设计并找出以下几个方面存在的问题：

- 设计文档和规格说明文档不一致的地方。
- 设计文档不正确、不一致、不完整或者模糊不清的地方。
- 可以提高设计效率的地方（也许是使用不同的表结构或者索引结构就可以实现）。
- 项目达到目标所冒的风险。例如某个搜索算法满足响应时间要求吗？设计需要那些无法及时获得而影响进度的新版的数据库驱动程序吗？存在一些不可靠的假设作为全局决策的基础吗？

设计评审的目的在于找出问题而不是解决问题，找出的每个问题会被分配给设计组的某个成员，要求他在指定时间内解决，并添加到设计文档的以后版本中。

在设计评审时发现的错误比在以后编程和测试时发现的错更容易修改，而且代价也更小。在将系统交付给客户以后才发现问题，这时更改的代价将会更大。

设计评审会议可能会包括**测试计划**（test plan）的评审，下一节将讨论这个问题。

12.2 测试计划

测试是所有软件工程项目中重要的一部分，而不是测试人员在编程完成以后才开始考虑的一种非正式的特殊行为。测试要按照正式的测试计划文档来进行，这个文档是在项目的设计和编程阶段准备的。测试计划文档规定所要执行的测试、所使用的测试数据以及对执行测试所需的测试驱动程序和脚本软件的设计。

完整的测试计划包括**模块测试**（module test），是由每个模块的编程人员在提交模块进行系统整合前进行的测试；**集成测试**（integration test），是由将模块整合到系统的人员负责的测试；以及**QA测试集**（QA test set），是由质量保证人员在完成整合的系统上进行的测试。

本节将重点介绍最终的QA测试集，但在讨论之前先注意模块测试的一个重要方面，即模块中每个SQL语句的测试。这些测试包括**代码检查**（code check），由相关人员检查每个SQL语句并确认符合英语语言规范，此外还包括常规的测试，即在实际或者测试数据库上执行SQL语句。

为商业事务处理系统设计合适的QA测试集是件非常辛苦的事情。测试集中的测试用例可以用下面两种方法进行测试。

黑盒测试（black box test）是使用规格说明文档设计出来的，而不需要查看设计文档和代码。这种测试假设系统是一个看不到内部结构的“黑盒子”，目标是确认系统符合规格说明。因此规格说明文档中的每一个规格说明（包括错误情形）至少有一个测试用例相对应，一些规格说明可能有好几个测试用例。例如，为了测试注册某门课程的学生数没有超过规定的最大数目，测试人员必须测试注册的学生数比最大数目少1和正好等于最大数目的时候执行注册事务的情况。测试用例也应该包括数目远小于最大数目的情况。如果最大数目指定为0或者1，会发生什么情况？由于在整数范围内测试所有可能的已经注册学生数和最大数目是不实际的（这是一个完全的穷举测试集），因此设计人员必须根据经验设计测试集，它能够代表可能发生的情形，满足适当的边界条件，并且能在规定的测试时间内完成。

因为在规格说明文档中指定了用户接口，因此黑盒测试必须测试用户接口。

白盒测试（glass box test）是使用设计文档和代码设计的。之所以叫“白盒”，是因为可以看到系统内部结构，目的在于确认编程的细节是正确的。因此，白盒测试应当访问每一行代码，遍历每一个分支，检查每个循环的边界条件，调用每个事件，执行每个完整性检查，测试每个算法的方方面面。例如，检查学生是否完成某门课程的所有预备课程的代码包含一个while循环，测试设计人员应当测试循环的退出条件是正确的。注意，while循环的存在条件以及退出条件在规格说明文档中并没有特别指出，这就是为什么除了黑盒测试之外还需要白盒测试。不过黑盒测试也是必要的，因为设计人员可能会对规格说明文档中的某些部分理解有误，而基于设计文档的测试集不会发现这些错误。例如，白盒测试显示while循环的退出条件是符合设计文档的，但是设计文档没有正确地解释规格说明。

测试计划可能还包括**压力测试**（stress test），即将系统置于模拟的或者真实的情况下确认它满足事务通过量和响应时间的规格说明。这类测试可以显示大量死锁发生的情形，以及因某些原因需要调整数据库设计来增加通过量和减少响应时间的情形。

如果程序是建立在保证ACID性质的系统上，并且假设已经单独测试过每个事务，那么就不需要测试事务是否能正确地并发执行。但是如果程序是在低于SERIALIZABLE（见10.2.3节）的隔离性级别上执行，就需要测试在并发执行的情况下是否正确。

在测试计划中经常被忽略的是测试用户手册或者其他交付文档，确保它们与规格说明和交付的系统一致。

测试计划文档包括所有要执行的测试的脚本和相应的正确结果。因为测试计划包括大量的测试并且在测试和维护阶段（修改一些错误和增加一些新的功能之后）需要运行多次，因此使用测试驱动程序或者脚本机制来自动执行测试计划是十分必要的。如果使用了这样的机制，那么测试计划文档需要包括测试驱动程序的相应输入。如果测试驱动程序是作为项目的一部分实现的，那么也必须包括它的设计。

测试计划文档需要包含测试人员执行测试时使用的测试协议（或者是引用公司的标准测试协议文档）的描述。协议应当包括在发现错误的时候必须填写的**错误报告表单**（error report form）。错误报告包括测试人员的名字、测试日期以及错误描述，最重要的是对如何再现错误的描述。错误报表被发送给负责修复错误的人员，由他填写什么时候，如何修复错误以及在哪一个版本中包含修复版本的信息。整个协议必须确保所有发现的错误最终被修复并且在某一版本的代码中包括这些修复。

设计一个全面的而且便于管理的测试文档需要相当多的技巧和经验。对任何一个软件而言，测试集的实际大小和测试的范围通常是市场和技术上的决策，有时依赖于相互矛盾的因素，例如：

- 系统的正确性重要到何种程度？人们生活在危险之中吗？
- 时间对系统所在的市场重要到何种程度？竞争对手的产品将要发布吗？或者系统必须在即将到来的展览会上展出吗？

在某些重要的应用中，差不多有一半的时间是花费在项目的测试上。当管理层施加压力要求发布未经完全测试的产品时，有时候也会产生职业道德问题。

在应用中，一个小组实现系统并将它交付给另一个小组，由后者负责运行和维护，这时测试集、驱动程序或者脚本机制是与代码和文档一起作为可交付产品的一部分。

接受测试和beta测试

测试计划是由系统实现人员准备和执行的，而客户通常在接受系统之前要准备和执行**接**

受测试（acceptance test）。接受测试通常由实际输入和客户的数据库组成（实现人员的测试计划通常涉及边界测试用例的输入和测试数据库）。它可以增强客户对系统实现目标的信心。一个精明的客户会花费相当多的精力去设计接受测试。这个测试可以全面测试在真实情况下系统的运行情况。

如果系统是一个有许多客户的产品，那么可以选择其中的一些客户进行beta测试（beta test）。先前由实现人员进行的测试称作alpha测试（alpha test）。当alpha测试完成时，会将beta测试版（beta test version）提供给客户，他们在实际应用中使用这个版本并将错误报告给实现人员。虽然beta测试版仍可能包含严重的错误，客户需要谨慎使用（实现人员已经将alpha测试量最小化，依赖beta测试来发现系统中的错误），但是客户可以从早期的版本中受益并且一些经济上或者技术上的因素也会激发客户进行beta测试的兴趣。beta测试需要持续一段时间，之后系统的最初发布版会提供给所有的客户。

即使经过所有的测试，一些重要应用的客户通常仍会将新的系统和已有的系统一起运行一段时间，直到能确认它可以正确可靠地执行任务为止。

12.3 项目计划

项目计划是软件工程的另一个重要组成部分。在准备规格说明文档的时候，项目经理会给出项目计划的初始版本。为了制定项目计划，经理将项目分解成一组**任务**（task），估计完成每项任务所需的时间，然后将它们安排给个人或者小组（并告之预计的开始和结束日期）。

任务可以包括设计、编码、测试和制作文档，而完成这个属性必须准确地予以描述，例如可以说“模块3的编码已经完成”，但不能说“模块4已经有90%调试过”。估计完成一项任务的时间是十分困难的，因为要理解任务的复杂性和负责实现这项任务的人员的能力。

安排任务的一个很重要的方面是**任务依赖性**（task dependency），即某些任务要在其他任务完成以后方能进行。例如，模块的测试不可能在还没编程就开始。

根据依赖性和估计的任务持续时间可以生成**依赖图**（dependency chart），也叫做PERT图（Program Evaluation and Review Technique）。见图12-1。矩形代表活动，上方标识的是持续时间（有时还包括估计的任务持续的最长和最短时间），弧线代表依赖关系。使用这种方式，



图12-1 依赖图

可以表示哪些任务可以并行执行，而哪些任务必须依次执行。从起点到终点最长的一条路径称作**关键路径**（critical path），是估计完成项目所需的最短时间。

记录项目计划的其他图包括：

- **活动图**（activity chart），也称作甘特图（Gantt chart），这个名称是以发明它的人Henry Gantt命名的。它是一个表示任务什么时候开始和什么时候结束的条形图（见图12-2）。

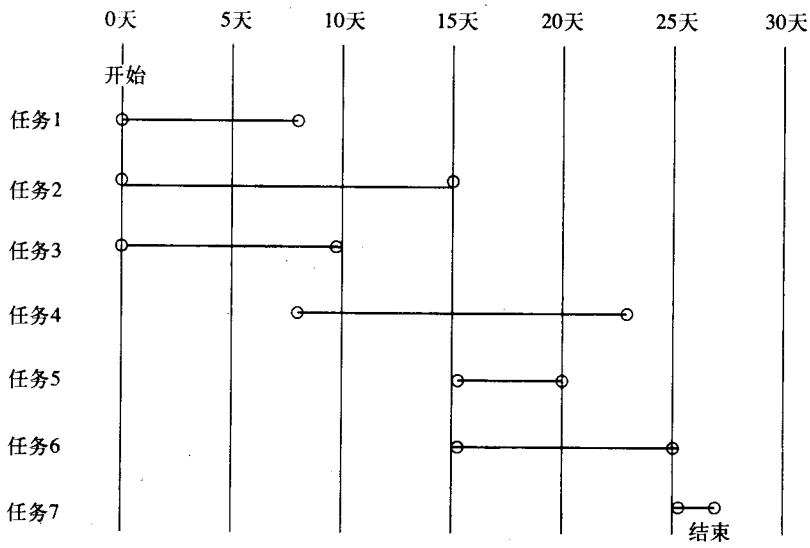


图12-2　活动图

- **人员安排图**（staff allocation chart），表示将某项任务安排给某人以及计划的起始和终止时间的条形图（见图12-3）。

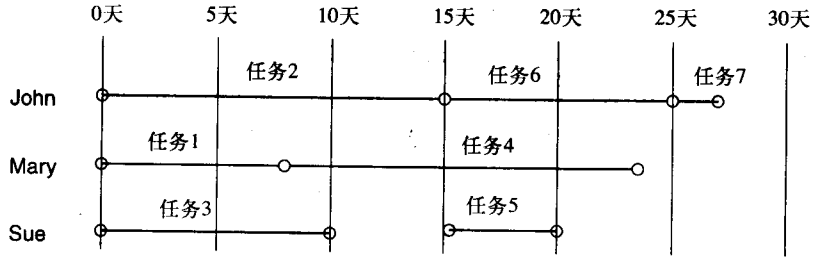


图12-3　人员安排图

在项目进行的过程中，项目经理将定期召开项目会议（可能是每周一次），会上每个实现小组成员报告所负责任务的状态和预期结束时间，并与项目计划规定的完成时间相比较。项目经理应当创造轻松的气氛使小组成员在对所分配的任务感到困难而难以在规定时间内完成时能够坦诚地提出来。如果有必要，项目经理需要做出适当的决策以保证项目按时完成。要特别关注关键路径上的任务，通常最好的人员都被安排完成这些任务。在会议上记录所做出的决策和任务安排是十分重要的。

当项目完成的时候，许多项目经理会举行总结会议，整个小组一起讨论在项目计划和其他方面的经验和教训。目的在于从错误中吸取教训，并提高经理和小组成员软件工程的能力。

尽管项目管理软件可以自动生成项目计划图，但是准备和监督项目计划需要丰富技巧和经验。实际上，许多软件项目失败或者延迟的主要原因在于项目经理缺乏项目计划和监督的能力。

12.4 编程

在大部分软件项目中，编程花费的时间少于整个项目完成时间的六分之一。当应用生成器和可重用对象包能够自动地从设计生成代码的时候，所花费的时间将更少。

一个运作良好的IT部门应当有每个程序员都遵守的编程指导。这些指导每个公司都不相同，但是很多规则是通用的。一些编程语言（如Java）有它自己的编程风格，但是这通常仅仅是典型编程标准要求的一小部分。一个众所周知的例子是GNU编程指导[Stallman2000]。在这里给出几个生成具有专业质量代码的建议：

- 编程最重要的两点是*正确*（correctness）和*清楚*（clarity）[⊖]。除了正确之外，代码必须能被很多人所理解（例如质量保证和维护组的人员），他们将在项目的整个生命周期里阅读它。
- 所有的程序员应当使用相同的变量定义和缩进风格等等。好的文本编辑器和集成开发系统会提供根据规则自动缩进排版程序的工具，整个系统的程序应当看起来似乎是一个程序员编写的。
- 变量和过程应当有面向应用的名称，能够从它的名称知道用途。例如不要使用S或者Stu甚至Student，而要使用Student_Name或者Student_ID_Number。
- 注释的使用。
  - 每个模块、过程和事务的代码都应当首先有**导言**（preamble），它与设计文档中的详细描述相同，可以包括作者、日期和修订编号。有的指导要求在程序文件中包括**修订历史**（revision history）记录，例如：

```
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
```

不过在程序文件中保留修订历史显得过时了，因为已经开发出成熟的版本控制系统，而且软件工程的复杂性一直在增加。一个比较好的方法是将相同目录下的所有文件的修订历史保存在一个单独的文件里，这个文件通常叫做ChangeLog，这样开发人员可以在一个地方看到对所有文件的修改，并且按照时间顺序排序。当相同目录下的程序文件相互依赖的时候，这种方式更具优越性。单独文件的历史可以从版本控制系统中获得，这比放在程序文件里要好。

- 模块、过程和事务中的注释应当是面向应用的。例如，下面的注释就是没有意义的：

```
/* Increment number_registered */
number_registered = number_registered+1;
```

因为可以从代码看出注释的含义。如果这行代码是注册事务中的一部分，并需要记录下来，那么更好的注释是：

⊖ 第三个c是指聪明（cleverness），重要性最低。

```
/* Another student has registered */
```

不是所有的语句都需要注释，一些关于编程风格的书建议给所有的循环和条件语句添加注释。例如，可以为循环和条件语句分别添加以下注释：

```
/* Give all employees making < $10,000 a 5% raise. */
```

和

```
/* If the customer has exceeded the credit limit,
** then abort transaction. */
```

- 当因修复bug和扩展系统而需要修改代码的时候，相应的注释也要同时更新。
- 清楚地记录系统相关的数据结构和代码。例如一些厂商提供它们自己的嵌入式SQL CONNECT语句：

```
EXEC SQL CONNECT student_database IDENTIFIED BY pml
    DBMS_PASSWORD = 'z9t.56';
    /* ***System Specific: Ingres version of CONNECT */
```

如果在将来系统要移植到另一个DBMS，通过文本编辑器能够很容易地发现与系统相关的语句。如果CONNECT语句在系统中出现多次，那么可以将它封装在一个过程中，在适当的地方调用这个过程即可。这样，当系统需要移植的时候，只要对这个函数进行修改就可以了。

原型编程

到目前为止，项目的编程阶段和需求分析以及设计阶段是完全区分开来的。但是实际上，为帮助制定决策，通常在早期阶段就实现系统的部分原型。例如，在事务处理系统中实现下面的原型可能是必要的：

需求分析阶段

- 用户接口的原型，用来评价接口的清晰度和可用性，并合并客户的反馈。
- 访问DBMS代码的原型，用来比较不同设计方案的速度，例如存储过程和嵌入在应用程序中的过程相比较。通过这些试验可以确认最终系统的预期事务吞吐量，或者在编程和测试阶段评价（减少）执行这些任务所需的时间。

设计阶段

- 一张表的不同设计原型，用来评价不同的索引模式执行某个SELECT或者UPDATE语句所需的时间。
- 设计的其他部分的原型，用来评价或者减少某个设计决策带来的风险。

有时原型代码最终会丢弃，但也有可能使用在最终的产品中。

12.5 渐进式开发

许多大型项目往往需要几年的时间才能完成，在这段时间，资助项目的企业的需求和目标可能会发生很大的变化，因此在项目过程中，企业经理也许会要求对正在构建的系统的规格说明进行大的修改。一个重要问题是实现小组如何对这样的要求做出响应。

一种方法是按照原来的要求继续构建系统。不过即使项目成功地完成，最终的系统可能也无法完全符合企业的要求。当然，系统的下一个版本会包括部分或者所有要求的变化，并

且新的版本可以在第一个版本完成后立即开始，并且有望在相对较短的时间里完成。

另一个方法是严格按照经理的要求，在项目过程中多次修改规格说明、设计和代码。遗憾的是，这样永远也不能成功地完成项目，因为设计始终在修改，代码始终在重写，项目完成时间一拖再拖，而且需要越来越多的钱维持项目的进行，直到最后取消项目。根据Standish Group的报告，31%的信息系统项目在完成之前就取消了。要想成功地使用这种方法，需要限制修改的次数和范围，还需要协商预算的增加和时间的延长。

还有一种方法称作**渐进式开发**（incremental development），该方法一步步地构建系统。首先在部分初始规格说明的基础上建立系统的**核心版本**（core version），这项工作可以在短时间内实现。接着开发下面的版本，基于企业需求的改变和从运行先前版本中获得的经验，每个版本会包括更多的功能和一些经理要求的变化。经过几次渐进式开发后，系统包括了大部分的需求。当然它永远也不可能包含所有的需求，因为经理会不断修改它们。

渐进式开发通常优于其他两种方法。因为可以在相对较短的时间内得到一个可运作的系统，所以风险被降低到最小，同时可以基于实际运行的经验来扩展系统。一个潜在的缺点是需要仔细设计初始版本，否则早期版本的设计决策也许会导致在以后不太适合增加新的功能。那时设计者可以选择改变设计（重写大部分代码）或者使用旧的版本，这有可能使系统效率不高。

然而，有些设计人员把渐进式开发当作不做任何设计的借口，只在前一版本上随意删改，这种方式无疑会导致灾难性的后果。

12.6 学生注册系统的设计和编程

5.7节讨论表的设计和一些简单的适合学生注册系统的约束，本节将完成设计并提供更复杂约束的细节和注册事务的部分代码。

12.6.1 完成数据库设计：完整性约束

数据库设计的一个重要部分是列出数据库完整性约束并决定怎样检查约束：是由DBMS（在CREATE TABLE、CREATE ASSERTION或者CREATE TRIGGER语句中）自动检查还是在一个或者多个事务中检查。在5.7节数据库的初始设计中并没有完全讨论约束，是因为还没有介绍一些所需的SQL结构。

下面列出数据库完整性约束，并给出在哪里实施约束：在模式中（见图5-15、图5-16和图12-4）还是在独立的事务中。它们与3.2节需求文档中给出的约束一致，只是对它们加以扩充指定每个约束涉及的属性和表。

- Id唯一性。STUDENT表中的Id，FACULTY表中的Id和COURSE表中的CrsCode都必须是唯一的，这由相应表中的主键约束执行，见5.7节。
- 如果学生在某年/学期注册了一门课程，那么学校必须在该年/学期必须提供这门课程。如果在TRANSCRIPT表中存在具有某个CrsCode、Semester和Year的一行，那么也必须在CLASS表有一行与其对应。这是由注册事务Register()执行的，见12.6.3节。
- 注册限制性。CLASS表中，属性Enrollment的值不能大于同一行属性MaxEnrollment的值。这由CLASS表的CHECK约束执行。

```
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
```

图12-4　学生注册系统的部分约束

- 注册一致性。Class表中课程的Enrollment属性值必须与Transcript表中在那个学年/学期注册这门课程的学生数目相同。这是由图12-4中的EnrollmentConsistency断言执行的。
- 一名教师不可能在同一学期内同时教授两门课程。在Class表中不存在两行有相同的ClassTime、Semester、Year和InstructorId。这由5.7节中Class表的UNIQUE约束执行。
- 两门课程不可能在同一学期内同一时间在同一教室里教授。在表Class中不存在两行有相同的ClassroomId、Semester、Year和ClassTime。这由5.7节中表Class的UNIQUE约束执行。
- 注册某门课程的学生必须完成该课程所有的预备课程并且成绩在C以上。Transcript表中的每行t_i具有属性StudId、CrsCode、Semester和Year，如果在Requires表中存在某些行t_r具有相同的CrsCode值，并且EnforcedSince的值在t_i中的Semester前，那么对每个t_r在Transcript表中必存在一行t_n，它具有相同的StudId，其CrsCode值与t_r的PrereqCrsCode相同，具有更早的Semester和Year，并且Grade在C以上。这是通过在12.6.3的注册事务中调用方法checkPrerequisites()执行的。
- 一个学生不能注册在同一时间进行的不同课程。Transcript表中不存在具有相同StudId的两行，以至于其CrsCode、SectionNo、Semester和Year在Class表中有相同的ClassTime。这个约束还没有实现，留作练习。

- 在任一学期学生注册的课程不能超过20个学分。设S为TRANSCRIPT表中具有相同StudId、Semester和Year的所有行的集合，那么COURSE表中与S中CrsCode值对应的CreditHours的总和必须小于或者等于20。这是通过在12.6.3节的注册事务中调用方法checkRegisteredCredits()执行的。
- 分配给某门课程的教室的座位数不能少于该门课程允许的最大注册人数。如果cl和cr分别是CLASS表和CLASSROOM表中的行，它们具有相同的ClassroomId，那么cl中MaxEnrollment的值不能大于cr中Seats的值。这是由图12-4中的ROOMADEQUACY断言执行的。
- 一个用字母表示的有效成绩不能改成Incomplete。一旦A、B、C、D或者F被赋给表TRANSCRIPT中某行的Grade属性，就不能在以后改成I，这是通过CANTCHANGEGRADETOI触发器执行的，见图12-4。

图12-4通过定义模式的断言和触发器部分完成数据库设计。

12.6.2 设计注册事务

现在使用12.1.1节G部分所给的形式设计注册事务。当一个学生想注册某门课程的时候，他先启动程序，这时GUI出现下一学期可选择的课程。选定某门课程以后，GUI通过构造器ClassTable()生成Java类ClassTable的一个实例对象。然后学生在GUI上按下注册按钮，这个新对象将调用注册事务（该事务的代码在这个类的Register()方法中）。

1) 事务名称

```
public int Register(Connection con, String sid)
```

这个事务是作为ClassTable类的一个方法实现的，由这个类的具体实例调用。变量courseId和sectionNo通过类的构造器ClassTable()设置，以对应某门课程。

2) 描述　事务经过注册正确性检查后批准一名学生注册某门课程。

3) 参数

a. Connection con　数据库连接标识符。

b. String sid　注册学生的Id。

4) 返回值

a. 如果注册成功，返回常量OK。

b. 如果在动作部分描述的任何一个检查失败，返回一个对应于那个失败的字符串。

c. 如果数据库操作失败，返回用户定义的常量FAIL。

5) 调用口　因为没有给出完整的设计和其他类与方法的名称，所以这里没有相应的内容。

6) 调用

a. checkCourseOffering()

b. checkCourseTaken()

c. checkTimeConflict()

d. checkRegisteredCredits()

e. checkPrerequisites()

f. addRegisterInfo()

前5个函数执行动作部分所描述的检查。如果所有检查成功，最后一个函数将更新数据库以完成注册。

7) 先决条件

a. 参数sid对应的学生已经经过认证。

b. 数据库已经打开，并已经创建连接con。

c. 已经在连接上执行JDBC方法setAutoCommit(false)。

8) 隔离级别　SERIALIZABLE：通过下面的调用设置：

```
setTransactionIsolation (Connection.TRANSACTION_SERIALIZABLE)
```

9) 动作

a. 文本描述

i. 事务检查下列内容：

a) 在下学期提供的课程。

b) 学生还没有注册这门课程，目前没有参加这门课程，也没有在这门课程获得C或者以上的成绩。

c) 学生没有注册在同一时间进行的课程。

d) 学生在下学期选择的课程学分总和将不会超过20。

e) 学生已经完成（或者正在进行）这门课程的所有预备课程，成绩都在C或者以上。

ii. 如果学生通过所有的检查，事务完成注册，相应地在TRANSCRIPT表中为这个学生和课程增加一行记录，并在CLASS表中将这门课程的Enrollment属性加1。最后提交返回状态OK。

b. 访问的表

i. 检查包括表STUDENT、COURSE、REQUIRES、CLASS和TRANSCRIPT。

ii. 更新包括表TRANSCRIPT和CLASS。

c. 错误情形

i. 正确性检查

如果动作部分中的任何一个检查失败，事务会中止并返回标识这个失败的状态。例如，如果学生已经学完这门课程并且成绩在C以上，那么checkCourseTaken()方法就会返回用户定义的常量CourseTaken作为状态，Register()事务返回该状态。同样，如果检测出时间安排有冲突，checkTimeConflict()方法返回TimeConflict，事务返回该状态。调用方法负责生成相应的错误消息。

ii. 系统执行的自动约束检查

注册的学生数不能超过课程最大允许注册人数MaxEnrollment（这是由断言执行的）。如果检查失败，DBMS会中止事务，调用过程生成相应的消息。

iii. 其他异常情况

如果数据库操作失败，事务中止并返回FAIL，调用过程负责生成相应的错误消息。

12.6.3 部分注册事务程序

下面的代码是用Java编写的部分注册事务程序。程序定义ClassTable类和关键的方法Register()，这个方法指定课程注册事务。另外，程序提供一致性检查的方法checkCourseTaken()的代码，但同时忽略其他类似的方法。

这个方法的结构易于编程和理解。首先进行必要的检查决定是否允许注册请求，每个检查都由一个独立的过程完成。如果所有的检查通过，过程addRegisterInfo()会更新表。如果更新成功，就提交事务。检查和更新执行的数据库动作可以作为数据库服务器上的存储过程实现。

正如设计指出的那样，所有错误和成功信息不是由注册事务生成的，而是由调用注册事务的GUI程序根据事务返回的值产生的。这种设计的优点是允许系统在二层或者三层的体系结构上实现，这种体系结构由控制屏幕显示的表示服务器（presentation server）和完成实际工作的应用服务器（application server）组成（在第22章将讨论事务的多层体系结构）。使用三层体系结构的学生注册系统，在表示服务器上执行GUI程序，在应用服务器上执行注册方法，而在数据库服务器上执行设计文档G.6节中列出的存储过程。

```
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;
      } 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()
```

下面的程序实现Register()中进行的一个检查，检查学生已经学过课程并获得符合要求的成绩。

```
// 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
```

12.7 参考书目

本章的许多主题在标准的软件工程书籍中都有更详细的讨论，例如[Summerville 1996, Pressman 1997, Schach 1990]。而建立模型、设计数据库和事务处理应用中涉及的特殊的主题在[Blaha and Premerlani 1998]中进行讨论。

12.8 练习

12.1 为一个简单的计算器准备设计文档和测试计划。

12.2 a. 为什么黑盒测试不能测试规格说明中的所有方面?

b. 为什么白盒测试通常不能测试代码中的所有执行路径（这不意味着白盒测试不能访问代码的所有行和分支）?

12.3 为什么并发系统（例如操作系统）很难测试？为什么事务处理应用即使是并发的，却不那么难测试?

12.4 从以下方面说明渐进式系统开发的优点:

a. 资助项目的企业经理

b. 项目经理

c. 实现团队

12.5 在12.6.2节给出的注册事务设计中，一些必需的检查是在模式中执行的，而一些检查是在事务程序中执行的。

a. 哪些在程序中执行的检查可以在模式中执行?

b. 修改模式执行这些检查。

12.6 重写12.6.3节的注册事务程序。

a. 使用存储过程执行注册检查。

b. 使用一个存储过程执行所有的检查，而不是像程序中每个检查都有相应的过程。

c. 使用C和嵌入式SQL。

d. 使用C和ODBC。

12.7 评价12.6.3节中注册事务程序的编程风格。

12.8 为12.6.2节中注册事务程序准备测试计划。

12.9 为学生注册系统的注销事务做以下的准备工作:

a. 准备设计

b. 编写程序

c. 准备测试计划

第13章 查询处理基础

理解查询处理的原理和方法有助于应用程序设计者设计出更好的系统。我们在这一章中研究用于判定基本关系运算符的方法，并讨论它们对物理数据库设计的影响。第14章将讨论查询处理更高级的部分：查询优化。

13.1 外部排序

排序是用于计算机编程的许多算法中重要的一部分，也是支持关系运算的核心算法。例如，排序是消除重复元组最有效的方法之一，也是一些联结算法的基础。关系数据库管理系统中用于处理查询的排序算法不是在基本算法课程中介绍的那些排序算法。后一类算法在所有数据都存储在主存中时执行排序，而这通常在数据库环境中是不适用的。当文件很大只能存放在外存（如磁盘）上时，我们就要用到**外部排序**（external sorting）。

外部排序的主要思想是把文件的一部分放入主存，用一种已知的内存排序算法（如快速排序）来对它们进行排序，然后把结果存回磁盘。这样就产生了若干有序的文件片段，之后必须合并这些文件片段来产生一个有序的文件。因为执行一次输入/输出操作的时间比执行一条指令的时间高出几个数量级，所以可以认为输入/输出的代价决定了内存排序的代价。因此，外部排序的计算复杂性通常只用磁盘读写的次数来衡量。典型的外部排序算法包含了两个阶段：*部分排序*和*合并*。

1. *部分排序*

部分排序阶段非常简单。假设我们在主存中有一个缓冲区，可以容纳下M页用于排序，而文件有F页。F通常比M要大得多。第一个阶段的算法如下：

```
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 for in-memory sorting)
    dump the sorted file segment into a new file
} until (end-of-file)
```

我们用术语**运行**（run）来指代由上面循环的一次迭代产生的有序文件片段。运行的大小是片段中页的数目。因此，第一阶段产生了$\lceil F/M \rceil$个有序的运行，代价是$2F$次磁盘输入/输出操作（为简单起见，我们假设每次输入/输出操作只在磁盘和内存之间传输一页，这里的符号$\lceil\ \rceil$表示取大于或等于F/M的最邻近的整数的操作）。部分排序阶段如图13-1所示，其中我们假设每个块包含两个记录。

2. *k路合并*

算法的下一个阶段是取出有序的运行，把它们合并成更大的有序运行。重复这个过程直

到我们只剩下一个有序运行为止。这就是我们的最终目的：最初文件的有序版本。k路合并算法取k个大小为R页的有序运行来产生一个大小为kR的有序运行，如图13-2所示。实际的算法如下所示：

```
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
}
```

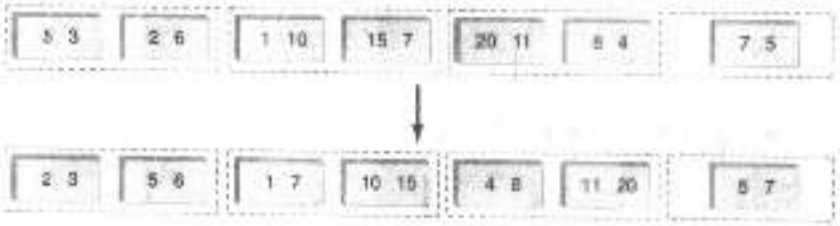


图13-1 文件的部分排序，$M=2$，$F=7$

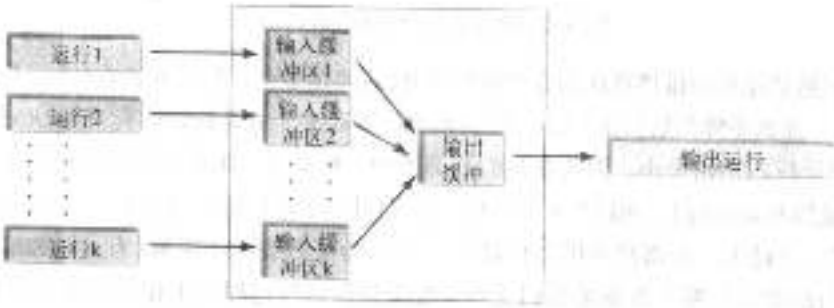


图13-2 k路合并

因为每个运行都是有序的，所以选择步骤是很简单的，其原因是一个运行中余下的最小元素总是它当前的头元素。图13-3说明了2路合并算法的重复应用，图13-4说明了3路合并的情况。

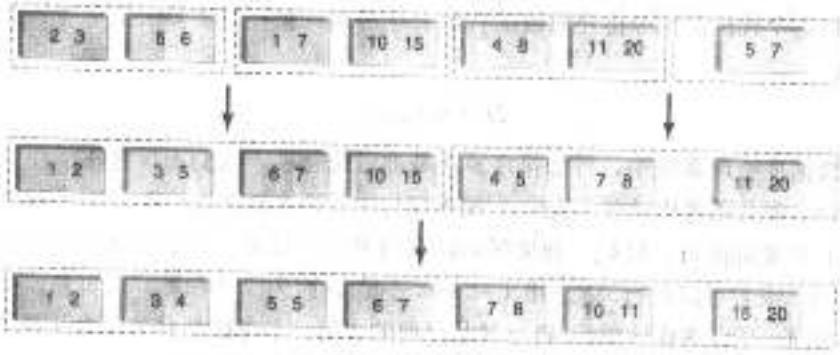


图13-3 在2路合并中合并有序运行

大小为R的k个运行的k路合并的代价是多少？显然，必须扫描每个运行一次，然后整个输

出必须写回磁盘。因而代价是$2kR$。因为最初可能有多于k个的运行，所以我们把这些运行划分为若干组，每组有k个运行，再分别对每组应用k路合并。如果把这个过程也作为一个步骤，并且开始的时候有N个运行，那么就有$\lceil N/k \rceil$个组，合并步骤总代价的上界是$2RN$。注意，这个值并不依赖于k。另外，因为在下一个合并步骤中，开始有$\lceil N/k \rceil$个运行，每个运行的最大大小是kR，所以合并的代价不会超过$2RN$，并且得到$\lceil N/k^2 \rceil$个运行。实际上，通过推理很容易看出这个输入/输出代价的上界对于合并算法的每一步都是成立的。

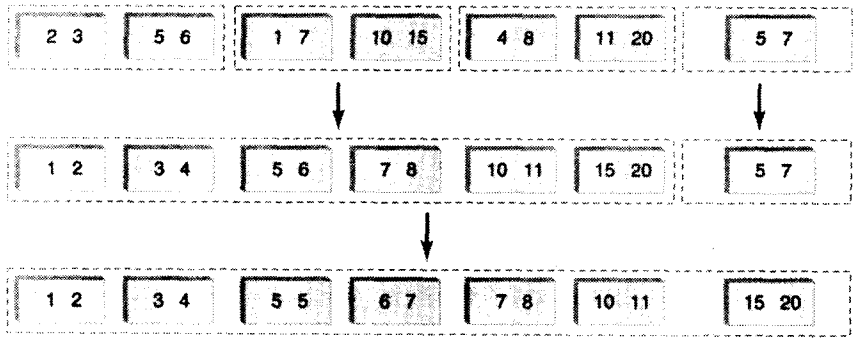


图13-4　3路合并中的合并选择运行

下一个问题是在外部排序算法的合并阶段k的值应该是多少。如果开始有N个运行，每一步执行k路合并，那么步骤的数目是$\lceil \log_k N \rceil$。因而，算法整个合并阶段的代价是$2RN * \log_k N$，这里R是有序运行的初始大小。因为在我们的例子中，$R = M$（即我们可以使用整个缓冲区来产生最大可能的初始运行），所以$N = \lceil F/M \rceil$，我们可以得出结论，代价是$2F * \log_k \lceil F/M \rceil$。

由此可知，k越大，外部排序的代价越小。那么为什么我们不能取k为可能的最大值（即初始有序运行的数目）呢？答案是我们受到分配给外部排序过程的主存缓冲区大小M的限制。这暗示我们应该让k尽可能地大以便充分利用这块内存。因为在合并的时候必须至少分配一页来收集输出（周期性的整体传向磁盘），所以k的最大值就是$M-1$。把这个值替换到前面的代价估计中，就得到了下面的估算：

$$2F(\log_{(M-1)}F - \log_{(M-1)}M) \approx 2F(\log_{(M-1)}F - 1)$$

最后，把这个代价和部分排序阶段的代价合在一起，就可以估算出整个外部排序过程的代价：

$$2F * \log_{(M-1)}F \tag{13.1}$$

在商用数据库管理系统中，外部排序算法是高度优化的。它们不仅考虑了初始运行的内存排序的代价，而且还考虑到在一次输入/输出操作中传输多页比执行多次输入/输出操作（一次传输一页）要更加高效。例如，如果缓冲区可以容纳下12页，那么或者可以执行11路合并（一次读一页），或者可以执行3路合并（一次读三页）。因为读（和写）三页和读一页所需要的时间几乎相同，所以考虑这样的3路合并是合理的。另一个考虑是有关在合并操作的过程中将输出缓冲区整体写回磁盘时的延时问题。可以用像双缓冲或三缓冲这样的技术来减少这种延时。然而尽管有着这样那样的简化，但是在本节中讨论的算法和代价估计和真实系统中的情况极为近似。本章其余部分对算法讨论就依赖于对代价估计的理解以及这里讨论的排序算

法的细节。

3. 排序和B+树

上面描述的基于合并的算法是查询处理中最常用的排序方法，因为它适用于所有的情况，并且不需要辅助的数据结构。然而，当存在这样的结构时，则执行排序的代价会更低。

例如，假设在排序键上有二级B+树索引。对树的叶子实体进行遍历可以产生实际数据文件的记录识别号（rid）的有序列表。原则上，我们可以简单地顺着指针，以搜索键的顺序来检索数据文件中的记录。令人奇怪的是，它并不比基于合并的算法效率高。

在决定是否用B+树索引来排序文件时，主要应该考虑索引是聚簇的还是非聚簇的。简单地说，如果索引是聚簇的，最好用B+树索引，否则不会有很好的效果。如果索引是聚簇的，那么数据文件一定已经几乎排好序了（通过定义），所以我们就不需要做什么了。然而，如果索引是非聚簇的，那么遍历B+树的叶子并顺着数据记录指针将以随机的顺序检索主文件的各页。在最坏的情况下，这可能意味着我们必须为索引叶子节点中的每个记录传输一页（回忆一下，我们之前的分析是基于文件中页的数目，而不是记录的数目）。练习13.1讨论了用非聚簇的B+树来进行外部排序的代价估计。

13.2 计算投影、并和差

计算投影、并和差操作初看上去是很简单的。例如，就投影来说，可以扫描这个关系，然后删除不想要的字段。然而，如果用户查询中有DISTINCT指令，情况就比较复杂了。这里的问题是必须清除可能由投影操作导致的重复元组。例如，如果我们把图4-5中的关系TRANSCRIPT的StudId和Grade属性投影掉，那么元组〈MGT123, F1994〉将在结果中出现两次。因而，我们必须找到高效的技术来消除重复元组。

在计算两个关系的并的时候也可能出现与重复一样的问题。在计算两个关系的差**r**−**s**的时候是不会引起重复的，除非最初的关系**r**已经有重复了。然而，我们面临的问题是相似的：我们必须识别**r**中与**s**中相同的元组。

有两种技术可以用来找出相同的元组：排序（我们在前一节中讨论过这种技术）和散列。我们首先把这些技术应用到投影操作符上，然后讨论对并和差操作符来说所需要做的修改。

1. 基于排序的投影

这个技术首先扫描初始关系，删除要被投影掉的元组分量，再把结果写回磁盘（我们假设没有足够的内存来存储这个结果）。这个操作的代价是$2F$数量级，这里F是关系中页的数目。然后排序这个结果，代价是$2F*\log_{(M-1)}F$，这里M是用来排序的主存页的数目。最后，我们再扫描一次结果（代价是$2F$），因为相同的元组彼此相邻（因为关系是排好序的），所以我们很容易删除重复的元组。

实际上，如果把排序和扫描结合起来效果会更好。首先，我们在排序算法的部分排序阶段从元组中删除不想要的分量。在该阶段，我们要扫描初始关系，所以去除这些元组分量不需要额外的磁盘输入/输出开销。其次，我们把消除重复元组和输出有序运行到磁盘的步骤合并起来，这样就省去了最后用来删除重复所需要的那遍扫描。因为每个这样的步骤都要写出有序元组的块，所以消除重复可以在主存中完成，不需要额外的输入/输出开销。

因而，基于排序的投影的代价是$2F*\log_{(M-1)}F$。另外，如果我们考虑到第一次扫描可能

产生较小的关系（大小是αT，这里$\alpha<1$是减小因子），那么投影的开销就可能更低了（见练习13.2）。

2. 基于散列的投影

快速识别重复的另一种方法是用散列函数。假设散列函数产生范围在1～$M-1$之间的整数，并且在主存中有M个缓冲页，它包含了（$M-1$）页的散列表和一个输入缓冲。算法的工作原理如下。在第一阶段，扫描初始关系。在这次扫描中，我们去除要投影掉的元组分量，元组的其余分量在余下的属性上进行散列。每当散列表的一页写满时，就把它整体传输到磁盘上相应的桶中去。这一步如图13-5所示。

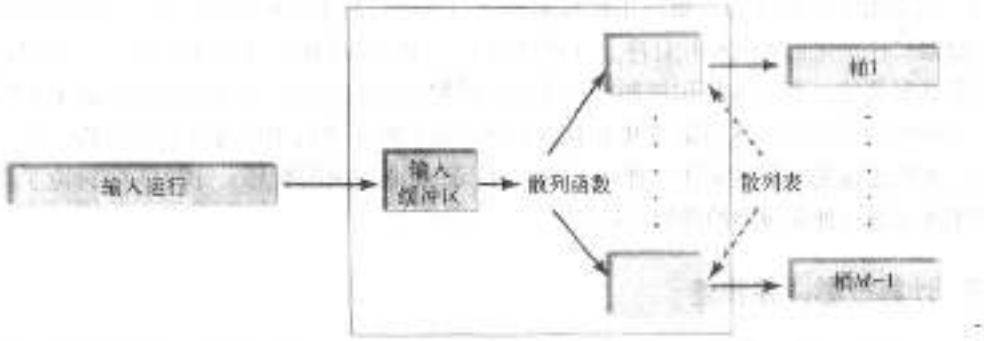


图13-5　散列输入关系到桶

显然，重复元组总是散列到同一个桶中去，所以我们可以分别在每个桶中消除重复。在算法的第二个阶段完成重复消除。假设每个桶都可以放进主存，那么可以用如下方法完成第二个阶段：整体读取每个桶，在主存中对它进行排序来消除重复，再把它整体写进磁盘。整个过程的输入/输出复杂度是$4F$（如果考虑投影的减小因子$\alpha(\alpha<1)$，那么结果是$F+3\alpha F$）。如果单个的桶不能放进内存中，那么必须用外部排序来对它们进行排序，而这将引起额外的输入/输出开销。练习13.3讨论了这种情况下的代价估计。

3. 基于排序方法和基于散列方法的比较

假设每个桶都可以放进主存即使对于很大的文件也是可以实现的。例如，假设做投影的程序有10 000页的缓冲区。这样的缓冲区只需要40M的内存。廉价的台式电脑也会有这么大的内存。我们可以先用这个缓冲区来存储散列表，然后用它来读取桶。假设每个桶都可以放进缓冲区，那么处理$10\,000\times 10\,000=10^8$页的文件（400G）只需要不到$4\times 10^8$次页传输的开销。基于排序的投影算法真的能表现得更好吗？在这种情况下，它的开销是$2\times 10^8\log_{10^4-1}10^8$，稍微高了一些。

因此，要外部排序散列桶的可能性就很小了。更大的风险是使用的散列函数不会把元组平均分配到各个桶中去。虽然一般来说，在这种情况下桶是可以放进内存的，但是其他一些桶却不能。在最坏的情况（这是不太可能的）下，所有元组可能都落入了同一个缓冲页里。这将要求进行外部排序。扫描初始关系并把它复制到单个桶的代价将是$2F$，再加上排序桶并消除重复的代价将是$2F\log_{M-1}F$。这与基于排序的投影相比就浪费了$2F$次页传输。

4. 计算并和差

计算并类似于计算投影，只是不需要去掉不想要的属性。为了计算差r-s，我们把r和s都

排好序，然后类似于合并过程那样并行扫描它们。然而，每当我们发现r的一个元组t也在s中时，我们不是合并元组，而是不把它加进结果中。

在基于散列的差计算中，我们可以像前面所说的那样把r和s散列到桶。然而，在每个桶中我们必须区分来自r的元组和来自s的元组。在第二阶段，必须分别对每个桶应用差操作。

13.3 计算选择

计算选择运算符可能比计算投影和集合运算复杂得多，可能要用到更多的技术。针对特定的选择运算符选择技术时取决于选择条件的类型和涉及的关系的物理组织。通常，数据库管理系统自动决定它将要使用的技术，所采用的技术基于我们下面要描述的启发式方法。然而，理解这些启发式方法就给了程序员一个要求物理组织的机会，所要求的物理组织对于在特定的应用程序中最频繁出现的选择类型是最佳的。

我们首先考虑形如$attr$ op $value$的简单选择条件，这里op是=、>、<等比较符中的一个，然后把我们的技术推广到涉及布尔运算符的复杂条件。

数据库查询常常导致两种不同的选择：基于等式（$attr = val$）的选择和基于不等式的选择（像$attr < val$）。后者称为**范围查询**（range query），因为它们通常成对出现来指定值的范围，如$\sigma_{c_1<attr\leqslant c_2}(\mathbf{r})$。

13.3.1 具有简单条件的选择

计算选择$\sigma_{attr\ op\ value}(\mathbf{r})$的一个很显而易见的方法是扫描关系r，对其中的每个元组检查选择条件，输出满足选择条件的元组。然而，如果只有一小部分元组满足条件，那么扫描整个关系的代价就显得太大了。在了解更多关于r的结构信息的情况下，就不需要做全部扫描了。我们考虑三种情况：1）在$attr$上没有索引；2）在$attr$上有B^+树索引；3）在$attr$上有散列索引。在第三种情况下，只能高效的处理等值选择（即op是“=”）。在第二种情况下，可以高效处理等值选择和范围选择，但是第三种情况在处理等值选择时效果会更好一些。在第一种情况下，只能对r做全部扫描了，除非关系r已经在$attr$上排好序了。在这种情况下，可以处理等值选择和范围选择，但是没有使用B^+树索引时那么高效。

1. 没有索引

通常，我们只能扫描整个关系r，代价是传输F页（r中磁盘块的数目）。然而，如果r在$attr$上已经排好序了，那么我们可以使用二分法来找到r中包含满足条件$attr = value$的第一条元组的页。我们可以以适当的方向扫描文件来检索所有满足$attr$ op $value$的元组。

这种二分查找的代价是与$\log_2 F$成正比的。为了做到这一点，我们还必须加入扫描包含合格元组的块的代价。例如，如果r有500页，那么查找的代价是$\lceil \log_2 500 \rceil$，即传输9页（加上包含合格项的磁盘块的数目）。

2. B^+树索引

有了$attr$上的B^+树索引，算法类似于针对排好序的文件的算法。然而，我们不是用二分查找，而是用索引来找到r中满足条件$attr = value$的第一个元组。更准确地说，我们找到包含或者指向满足条件的第一个元组的B^+树的叶子节点。我们从那里扫描B^+树索引的叶子来找到指向包含满足条件$attr$ op $value$的元组的页的所有索引项。

找出索引的第一个合格的叶子节点的代价等于B+树的深度。和以前一样，我们还要再加上扫描索引的叶子来确定所有合格项的代价。当然这个代价依赖于合格项的数目，而合格项的数目又依赖于选择条件和关系中实际的数据。

然而，事实并没有这么简单。到目前为止，我们只是描述了得到索引项的过程。得到实际元组的代价取决于索引是否是聚簇的。如果索引是聚簇的，那么所有感兴趣的元组存储在一页或相邻的几页中（不管索引是否集成到存储结构中这都是正确的）。例如，如果有1000个合格的元组，每个磁盘块存储100个元组，那么得到所有这些元组（假设我们已经找到了适当的索引节点）需要10次页传输。另一方面，如果索引是非聚簇的，那么每个合格的元组可能是在单独的一块中，所以检索所有合格元组可能要进行1000次页传输!

这就引起了令人不悦的前景：要进行的页传输次数与选择中合格元组的数目相同，这就很有可能超过整个关系r的页的数目。幸运的是，稍微想一想，我们可以有更好一些的方法。让我们首先对从索引中得到的合格元组的记录Id进行排序。然后我们可以按合格记录Id的升序来从关系中检索数据页，这就保证了每个数据页最多被检索一次。因而，即使是非聚簇索引，其代价也是和包含合格元组的页的数目成正比的（再加上搜索索引和排序记录Id的代价）。在最坏的情况下，这个代价可以和初始关系的页的数目一样高。(但是不会像元组数目那么大!) 这是因为合格元组并不是像使用聚簇索引那样集中在检索到的页中：检索到的页可能只包含一个合格的元组。因而，应该优先考虑聚簇索引。

3. *散列索引*

在这种情况下，我们可以使用散列函数来找出含有满足条件*attr* = *value*的元组的桶。因为两个在*value*上只有稍许差别的元组可能被散列到不同的桶中去，所以这个方法不能有效地用于像*attr* < *value*这样的范围条件。

通常，找出正确的桶的代价是一个常数（对于好的散列函数来说接近1.2)。然而，检索元组的实际代价依赖于合格元组的数目。如果这个数目比1大，那么和B+树索引的情况一样，实际的代价依赖于索引是否是聚簇的。在聚簇的情况下，所有合格元组集中在若干邻近的页中，那么检索的代价就是扫描这些页的代价。在非聚簇索引的情况下，元组分散在整个数据文件中，我们就面临了和非聚簇的B+树相同的问题——排序记录Id导致了和包含合格元组的页的数目成正比的代价。

13.3.2 存取路径

上面讨论的实现关系运算符的算法都是假设对于要处理的关系存在（或不存在）某种辅助的数据结构（索引)。这些数据结构和使用它们的算法称为**存取路径**（access path)。到现在为止，我们已经看到几个可以用于处理特定查询的存取路径的例子：*总可以使用文件扫描*；在查询中指定的属性上已经排好序的文件可以使用二分查找；如果*散列索引*或B+树有涉及那些属性的搜索键，则可以使用这两种索引。

必须仔细地为给定的关系运算选择存取路径。例如，考虑图4-5中的TRANSCRIPT关系，并假设我们在搜索键<StudId, Semester>上有散列索引。这个索引在计算$\pi_{StudId,Semester}$(TRANSCRIPT)时是很有用的，因为我们可以确定由于投影而产生的任何重复都出自存储在同一个桶中的元组。这就极大地简化了消除重复的工作，因为搜索重复可以用一次处理一个桶的方式完成。

另一方面，如果我们计算$\pi_{StudId,CrsCode}$（TRANSCRIPT），那么索引在消除重复方面就没有什么帮助了。尽管元组t_1和t_2在<StudId, CrsCode>上的投影可能是相同的，但是t_1和t_2可能被散列到不同的桶中去，因为它们在Semester属性上的值可能不同。为了使用散列来消除重复，散列索引的整个搜索键必须包含在没有被投影掉的属性集合中（见练习13.4）。

然而，上面的散列索引在计算表达式$\sigma_{StudId=666666666 \wedge Grade='A' \wedge Semester='F1994'}$（TRANSCRIPT）时是很有用的。我们可以使用散列索引来检索满足部分条件StudId=666666666 ∧ Semester='F1994'的元组，然后扫描得到的结果（估计可能会比较小）来找出还满足Grade='A'的元组。然而，这个索引在计算表达式$\sigma_{StudId=666666666}$（TRANSCRIPT）的时候是没有多大用处的，因为满足条件的元组可能分散在不同的桶中。

最后，在<Grade, StudId>上的散列索引对于计算关系表达式$\sigma_{Grade>'C'}$（TRANSCRIPT）是没有多大用处的，但是在搜索键<Grade, StudId>上的B⁺树索引是有帮助的（尽管搜索键是〈StudId, Grade〉的B⁺树可能没有帮助）。为了使用散列函数，我们要为StudId提供所有可能的值，还要提供高于C的Grade的值，而这是不实际的。相比之下，因为Grade是B⁺树搜索键的前缀，所以我们可以使用这棵树来有效地找出所有具有搜索键<g, id>（这里的g高于C）的索引项。

这个讨论引出了一个概念：什么时候存取路径才能*覆盖*一个特定关系运算符的使用。我们只为满足下面条件的选择运算符准确地定义这个概念：其选择条件是形如*attr* op *value*的项的合取。投影和差运算符留作练习。

覆盖把存取路径和可以用这些路径来计算的关系表达式联系起来。考虑如下的关系表达式

$$\sigma_{attr_1\ op_1\ val_1 \wedge \ldots \wedge attr_n\ op_n\ val_n}(\mathbf{R}) \tag{13.2}$$

这里**R**是关系模式。当且仅当下面的条件之一成立时，这个表达式被一个存取路径**覆盖**：

- 存取路径是文件扫描。（文件扫描显然可以用来计算任意表达式。）
- 存取路径是散列索引，该索引的搜索键是属性$attr_1, \cdots, attr_n$的一个子集，在这个子集中的所有op_i都是等号。（我们可以使用散列索引来识别满足选择条件中一些子条件的元组，然后扫描这个结果来验证余下的子条件。）
- 存取路径是有搜索键$sk_1, \cdots, sk_m$的B⁺树索引，搜索键的某个前缀$sk_1, \cdots, sk_i$是$attr_1, \cdots, attr_n$的一个子集。（11.4.3节解释了如何使用B⁺树来做部分键搜索。这将有助于我们找出满足选择中一些子条件的元组，余下的子条件可以通过对得到的结果做顺序扫描来验证。）
- 存取路径是二分查找，**R**的关系实例在属性$sk_1, \cdots, sk_m$上排序。在这种情况下，覆盖的定义和B⁺树索引上的定义是相同的。

注意，仅当与搜索键中的属性对应的所有比较都是=时，才可以使用基于散列的存取路径。即使这些比较涉及到像 ≤、<、>和 ≥ 的不等号，也可以使用其他的存取路径。如果比较运算符是 ≠，那么唯一可用的存取路径就是文件扫描，因为在枚举关系中的所有合格元组时没有哪个索引是很有效的。

举个例子，考虑表达式$\sigma_{a_1>5 \wedge a_2=3.0 \wedge a_3='a'}(\mathbf{R})$，假设**R**上存在B⁺树索引，搜索键是$a_2$、$a_1$、$a_4$。我们可以使用这个索引来找出叶子项$e$，其中$a_2$的值是3.0，$a_1$的值是5。如果这样的项不存

在，那么就找出索引中紧跟e的第一个项。我们可以从那一点开始扫描叶子项来找出所有还能使a_3有a值的叶子。

在我们继续之前还要注意一个概念：如果我们使用与存取路径对应的计算方法，那么存取路径的**选择度**（selectivity）就是将要检索到的页的数目。选择度越小，存取路径就越好。选择度与计算查询的代价是紧密相关的，不过查询代价可能还涉及其他因素。例如，可能用到多个存取路径（见13.3.3节），或者在输出之前需要排序结果（如果用到ORDER BY子句）。显然，对于任意给定的关系表达式，存取路径选择度依赖于那个表达式结果的大小，并且总是大于或等于包含结果元组的页的数目。然而，一些存取路径具有的选择度更加接近理论上的最小值，而其他一些存取路径更加接近扫描整个文件的代价。什么时候存取路径覆盖表达式的概念有助于确定出那些选择度对给定类别的表达式很“合理”的存取路径。

13.3.3 具有复杂条件的选择

我们现在开始讨论用于计算任意选择的方法。

1. 带合取条件的选择

这些是上面考虑的形如（13.2）的表达式。我们有两种选择：

- 使用最有选择性的存取路径来检索相应的元组。这样的存取路径通过尽可能多地使用在选择条件中提到的属性来构造搜索键的前缀。它以这种方式检索到需要的元组的最小可能的超集，我们可以扫描这个结果来找出满足整个选择条件的元组。例如，假设我们需要计算$\sigma_{\text{Grade>'C'} \wedge \text{Semester='F1994'}}$（TRANSCRIPT），并且存在搜索键为<Grade, StudId>的B^+树索引。因为这个存取路径覆盖了选择条件Grade>'C'，所以我们可以用它来计算$\sigma_{\text{Grade>'C'}}$（TRANSCRIPT）。然后我们可以扫描这个结果来识别与1994年秋季的学期对应的成绩记录。如果存在搜索键为<Semester, Grade>的B^+树索引，那么我们可以把它当作存取路径来用，因为这个键覆盖了两个选择条件。
- 使用覆盖该表达式的若干存取路径。例如，我们可能有两个二级索引，它们的选择度都小于直接文件扫描的选择度。我们可以使用这两个存取路径来找出可能属于查询结果的元组的rid，然后计算出它们rid集合的交集。最后我们可以检索选中的元组，再用余下的条件对它们测试。例如，考虑表达式$\sigma_{\text{StudId=666666666} \wedge \text{Grade='A'} \wedge \text{Semester='F1994'}}$（TRANSCRIPT），假设存在两个散列索引，一个在Semester上，另一个在Grade上。用第一个存取路径，我们可以找出1994年秋季的那个学期的成绩记录的rid。然后我们使用第二个存取路径来找出成绩是'A'的记录的rid。最后，我们可以找出同时属于这两个集合的rid，再去检索相应的页。当扫描元组的时候，我们可以进一步选择与学生的Id是666666666相对应的那些元组。

2. 带析取条件的选择

当选择条件包含析取的时候，我们必须首先把它们转换成析取范式（disjunctive normal form）。如果一个条件形如$C_1 \vee \cdots \vee C_n$，这里每个C_i是比较项的合取（就像表达式（13.2）中的那样），那么这个条件就是**析取范式**（disjunctive normal form）。

从原子谓词演算可以得知，每个条件都有等价的析取范式。例如，对于条件

```
(Grade='A' ∨ Grade='B') ∧ (Semester='F1994' ∨ Semester='F1995')
```

相应的析取范式是:

```
(Grade='A' ∧ Semester='F1994') ∨ (Grade='A' ∧ Semester='F1995')
∨ (Grade='B' ∧ Semester='F1994') ∨ (Grade='B' ∧ Semester='F1995')
```

对于在析取范式中的条件，查询处理器必须对单个的析取项检查存在的存取路径，并选择适当的策略。这里是一些可能情况。

- 必须使用文件扫描来计算其中一个析取项C_i。在这种情况下，我们可以在这个扫描的过程中计算整个选择表达式。
- 每个C_i都存在优于直接文件扫描的存取路径。其中又包含两种情况:
  1) 所有这些路径的选择度的总和接近文件扫描的选择度。在这种情况下，我们应该优先使用文件扫描，因为索引搜索的开销和其他一些因素可能会超过由于使用较复杂的存取路径而获得的很小的潜在收益。
  2) 所有析取项的存取路径的组合选择度比文件扫描的选择度要小得多。在这种情况下，我们应该使用适当的存取路径来分别计算$\sigma_{C_i}(\mathbf{R})$，再对结果取并集。

13.4 计算联结

计算投影、选择等关系运算符的方法只是计算关系联结的前奏。我们把注意力都放在比较不同存取路径上，在计算投影或者选择时可能发生的最坏情况是我们可能要扫描或者排序整个关系。这样的表达式的结果也是很规则的: 它不可能比最初的关系还要大。

把这个结论和关系联结作一下比较，关系联结中要扫描的页的数目和结果的大小都可能是输入大小的二次方倍。尽管“二次方”在某些算法具有指数级复杂度的应用程序中看起来并不是太糟糕，但是这在数据库查询计算中是不允许的。因为数据量太大，而磁盘的输入/输出又相对较慢。例如，联结两个只是跨越1000页的文件可能就要进行10^6次输入/输出操作，这即使对于批作业也是不可接受的。因此，在数据库查询处理中要特别关注联结。

考虑联结表达式$\mathbf{r} \bowtie_{A=B} \mathbf{s}$，这里$A$是r的属性，$B$是s的属性。计算连接有三种主要的方法: 嵌套循环、排序-合并和基于散列的联结。我们依次来考虑这三种方法。

13.4.1 用嵌套循环来计算联结

计算联结$r \bowtie_{A=B} \mathbf{s}$的一个显而易见的方法是使用下面的循环:

```
foreach t ∈ r do
    foreach t' ∈ s do
        if t[A] = t'[B] then output ⟨t,t'⟩
```

这个过程的代价可以用如下方法估算。令β_r和β_s分别是r和s中页的数目，τ_r和τ_s分别是r和s中元组的数目。容易看出，对于r中的每个元组都必须从头到尾扫描关系s，这将导致$\tau_r\beta_s$次页传输。另外，在外循环中必须扫描一次r。结果，有$\beta_r + \tau_r\beta_s$次页传输。(在所有对联结运算的代价估算中，我们忽略了把最终结果写进磁盘的代价，因为这一步对于所有的方法来说都是一样的，而且对这一阶段的估计依赖于结果的实际大小。估计这个大小会使我们偏离主题，还会在这一阶段分散注意力。)

上面的代价估计给了我们两个教训:

- 它涉及到大量的页传输！令β_r=1000，β_s=100，τ_r=10 000。我们的代价估计说明计算可能要求1000+10 000 × 100=1 001 000次页传输——对于联结这样相对较小的表来说这也太大了（如果一次页输入/输出需要10ms，那么大约要花费166分钟）。
- 循环的顺序。假设不是在外层循环中扫描r，而是在外层循环中扫描s，在内层循环中扫描r。假设τ_s=1000。在代价估算中互换r和s将会产生：100+1000 × 1000=1 000 100。虽然这个例子在运算上的减少是最小的了（令人震撼的9分钟！），但是容易看出，如果每页上的元组的数目在两个关系中是一样的，那么在外层循环中要被扫描的关系应该较小，而内层循环中被扫描的关系应该较大。

1. 块嵌套循环联结

如果我们不是对r的每个元组都扫描s一次，而是对r的每一页扫描一次s，那么嵌套循环连接的复杂性就会大为降低。这将把代价估计减小到$\beta_r+\beta_r\beta_s$——比上面的例子降低了一个数量级。达到这个效果的方法是为当前在内存中的r的页中的所有元组输出连接的结果。

如果我们可以把对s的扫描次数减小为r的每一页进行一次，那么我们能不能进一步把这个次数减小到一组页一次呢？如果我们能有更多内存的话，那么答案是肯定的。假设查询处理器有M页主存缓冲区来做这个联结。我们可以为外层循环关系r分配$M-2$页，为内层循环关系s分配一页，而把最后一页留作输出缓冲区。这个过程在图13-6中做了描述。

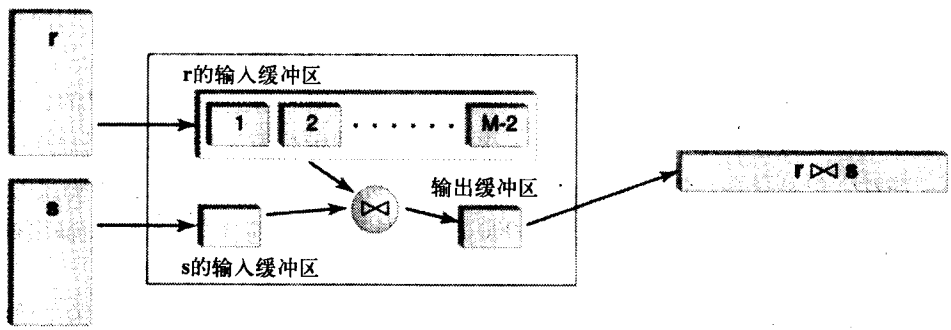


图13-6　块嵌套循环联结

块嵌套循环连接的代价可以用类似前例的方法进行估算：外层循环r只扫描一次，代价是β_r次页传输，对于r的每$M-2$个页才扫描关系s一次，即$\left\lceil\frac{\beta_r}{M-2}\right\rceil$。因而，代价（除去把输出写进磁盘的代价）是$\beta_r+\beta_s\left\lceil\frac{\beta_r}{M-2}\right\rceil$。在我们的例子中，如果$M$=102，那么这个代价将减小到1000+100 × 10=2000。如果在外层循环中扫描较小的关系s，那么代价将会更低，即100+100 × 10=1100（或者假设每次页输入/输出花费10ms，那么需要11秒）——相对于最初的简单实现的代价10^6已经减小了很多。

2. 索引嵌套循环联结

下一个方法是使用索引。如果r和s中在属性A和B上匹配的元组的数目相对文件的大小来说比较小的话，那么利用这个技术会取得特别好的结果。

假设关系s在属性B上有索引，那么我们不是在内层循环中扫描s，而是使用索引找出匹配的元组：

```
foreach t ∈ r do {
    use the index on B to find all tuples t' ∈ s such that
    t[A] = t'[B]; output ⟨t,t'⟩
}
```

为了估计索引嵌套循环联结的代价，我们要考虑到索引的类型以及它是否是聚簇的。在s中匹配元组的数目也是要考虑的因素。

如果索引是B^+树，那么找到s中匹配元组的第一个叶子节点的代价是2～4，这依赖于关系的大小。在基于散列的索引中，如果很好地选择散列函数的话，那么这个代价大约是1.2。下一个问题是需要多少次输入/输出操作来取得s中的匹配元组，这个答案依赖于匹配元组的数目以及索引是聚簇的还是非聚簇的。

如果索引是非聚簇的，那么检索所有匹配元组所需要的输入/输出的数目可能和匹配元组的数目一样多。所以非聚簇索引对于索引嵌套循环联结并不是很有用，除非匹配元组的数目很小（比如，如果B是s的候选键）。对于聚簇索引来说，所有匹配元组通常处于同一个或者相邻的磁盘块中，所以检索它们所需要的输入/输出的数目通常是1或者2。因此，在聚簇索引的情况下，这个代价估计是

$$\beta_r + (\rho + 1) \times \tau_r$$

这里ρ是检索B^+树索引的叶子节点或者找出散列索引正确的桶所需要的输入/输出的数目（我们假设索引不是和数据文件集成的，而且所有的匹配元组都能放进一页中，这就是为什么会有1的原因）。在非聚簇索引的情况下，这个代价是

$$\beta_r + (\rho + \mu) \times \tau_r$$

这里μ是对于r中每个元组来说s中匹配元组的平均数目。

我们再回到我们的例子，把这个代价和块嵌套循环联结的代价作一个比较。假设ρ是2（我们的关系很小），那么在聚簇索引的情况下的代价为1000 + 3 × 10 000 = 31 000——这比块嵌套循环中的代价要大得多。然而，如果我们在嵌套循环中把r和s互换，那么索引嵌套循环和块嵌套循环的代价就很接近了：100 + 3 × 1000 = 3100和1100。

在这个例子中，索引看起来对于块嵌套循环是不起什么作用的。那为什么要考虑索引呢？这是因为索引循环有一个很重要的性质：代价不容易受内层关系大小的影响，这可以从上面的公式中看出。所以，如果我们在内层循环中使用了s，它的大小增长到10 000页（100 000个元组），那么块嵌套循环联结的代价就增长到1000 + 10 000 × 10 = 101 000次页传输。相比之下，索引嵌套循环联结的代价的增长比较适当：1000 + 3 × 10 000 = 31 000。因而，索引联结在联结的关系比较大，并且一个比另一个大很多的时候会取得比较好的效果。

13.4.2　排序-合并联结

排序-合并的基本思想是首先对每个关系在连接属性上排序，然后使用各种合并过程来找出匹配的元组，即同时扫描两个关系，比较联结属性。当找到匹配元组的时候，就把联结后的元组加入结果中。然而，这样做的实际过程有点细致。我们在图13-7中给出了这一算法。

```
输入：在属性A上排好序的关系r；在属性B上排好序的关系s
输出：r ⋈A=B s

Result := {}                                    // 初始化结果
tr := getFirst(r)                               // 得到第一个元组
ts := getFirst(s)
while !eof(r) and !eof(s) do{
    while !eof(r) && tr[A] < ts[B] do
        tr := getNext(r)                        // 得到下一个元组
    while !eof(s) and tr[A] > ts[B] do
        ts := getNext(s)
    if tr[A] = ts[B] = c then {                 // 对于某个常数c
        Result := (σA=c(r) × σB=c(s)) ∪ Result;
        tr := the next tuple t ∈ r where t[A] > c;}
}
return Result;
```

图13-7　排序-合并联结的合并步骤

根据图中所示，算法中的合并步骤过程如下：扫描关系r和s，直到找到属性A和B上的匹配元组为止。当找到相应的元组以后，就把匹配元组的所有可能的组合（笛卡儿积）加入结果中。

我们现在根据页传输的数目来估计排序-合并联结的代价。显然，我们必须为排序关系r和s付出代价：$2\beta_r\lceil\log_{(M-1)}\beta_r\rceil+2\beta_s\lceil\log_{(M-1)}\beta_s\rceil$（假设有$M$个缓冲区可供使用）。计算归并步骤的代价比较细致。初看起来，似乎只要扫描一次r和s就可以了。然而，如果我们不能把$\sigma_{A=c}(r)$和$\sigma_{B=c}(s)$都放进内存中，那么计算笛卡儿积可能要对子关系$\sigma_{B=c}(s)$做多次扫描。计算这个乘积的最好方法是使用块嵌套循环联结。这里，页传输的实际数目依赖于这些子关系的大小和可用内存的数量。然而在通常情况下，匹配的子关系可以放进内存，这就可以避免对关系s做多次扫描（例如，如果A和B是键）。

对于我们的实例来说，我们在块中的关系大小如下：$\beta_r=1000$，$\beta_s=100$，用于联结操作的缓冲区有102页。这意味着s的排序可以在200次页传输中完成，r的排序可以在$2\times1000\times\lceil\log_{101}1000\rceil$，即4000次页传输中完成。合并还要进行一次扫描（假设匹配元组都可以放进（M-1）/2个页中，所以连接它们并不涉及额外的扫描）。因此，整个联结应该要花费5300次页传输。实际上，合并可以在外部排序算法的最后阶段进行（就像之前对于基于排序的投影所做的那样——具体内容留作练习13.9）。在这种情况下，排序-合并联结花费了4200次页传输。

在这个特殊的情况中，块嵌套循环算法看起来优于排序-合并算法。然而，只就索引嵌套循环方法来说，排序-合并的渐进行为优于块嵌套循环的。当r和s的大小增长时，块嵌套循环的代价呈二次方增长，即$O(\beta_r\beta_s)$，而排序-合并联结的代价的增长要缓慢得多（假设像上面讨论的$\sigma_{A=c}(r)$和$\sigma_{B=c}(s)$都很小），即$O(\beta_r\log\beta_r+\beta_s\log\beta_s)$。

13.4.3　散列联结

计算联结$r\bowtie_{A=B}s$的一个方法是预处理关系r和s，以使可能匹配的元组位于相同的或者临

近的页中。进行这样的预处理后就不需要对内层循环关系做重复扫描了。这也是上面讨论的排序-合并技术的基本思想。然而，排序只是一种可能的预处理技术。除此之外，我们还可以使用散列来保证匹配元组彼此之间的位置很接近。我们之前使用过这个技术，那时候我们需要把重复元组（因为投影或者集合运算而引起的）放到彼此都近的位置。显然，识别重复的问题是我们现在面对的这个问题的一种特殊情况：识别一个或多个属性有着相同值的元组（如上面的A和B）。图13-8说明了这一思想。

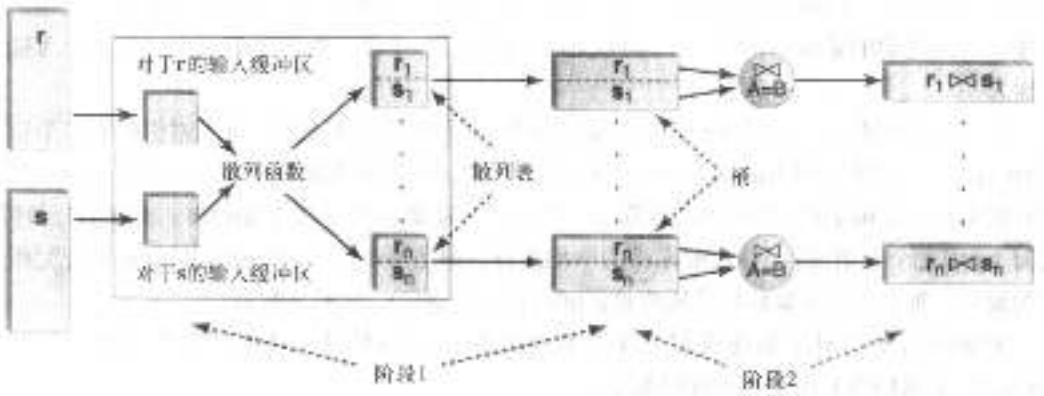


图13-8 散列联结

散列联结方法首先把每个输入关系散列到散列表中去，这里r在属性A上散列，s在属性B上散列。这样做的结果是r和s中可能匹配的元组被放进了同一个桶中。

在第二个阶段，连接每个桶的r部分和s部分以产生最后的结果。如果两个部分都可以放进内存，那么完成所有这些联结的代价就是对r和s进行单遍扫描的代价。如果桶对于内存来说太大了，那么可以尝试其他的联结技术。通常在这种情况下，在属性A上使用不同的散列函数来对每个桶的r部分作进一步划分。然后扫描相应桶的s部分，在这个过程中，使用新的散列函数来在属性B上散列s的每个元组，并确定r中匹配的元组。

假设每个桶都可以放进内存，那么联结r和s的代价就是对这些表做二次扫描。必须输入r和s，必须输出得到的桶（对每个关系作两次扫描）。然后必须输入每个桶来找出匹配的元组（对每个关系作一次扫描——回忆一下我们没有把最后的输出步骤包含在这个代价中）。在我们的实例中，代价是3300次页传输，这比块嵌套循环的代价要高，但是散列联结的渐进行为要更优。实际上，如果在算法第一阶段产生的每个散列桶都可以放进内存中，那么代价相对r和s的大小来说是线性的。这使得散列联结在迄今为止所讨论的所有方法中是最好的。然而，要认识到散列联结在很大程度上依赖于散列函数的选择，并且很容易被不合适的数据而破坏。（如果所有的元组都被散列到同一个桶中将会怎么样呢？）另外，散列联结只能用于等值联结中，所以散列联结对于更为普遍的联结条件（如不等）就不适用了。

13.5 多关系联结

我们在11.7节中讨论了联结索引，并把它们用在计算联结中。计算形如$p \bowtie_{r.a=s.b} q$的联结的算法首先对联结索引进行扫描，取出rid可以在索引项中找到的元组。

实际的计算实质上类似于索引循环联结，在外层循环中扫描**p**，只是我们使用联结索引来定位**q**的元组，而不是用q的在属性B上的通常索引。在这种情况下，联结索引相比其他索引类型的优势在于不需要对它进行搜索，因为所有匹配元组已经在彼此之间相联系了，所以可以简单地扫描索引，取得匹配元组的rid对，最后联结元组。

关于联结索引的思想可以扩展到多关系联结上，这里要创建索引来把多于两个元组的rid联系起来。例如，在3路联结$\mathbf{p}\bowtie\mathbf{q}\bowtie\mathbf{r}$中，联结索引包含了形如<p, q, r>的三元组，这里**p**是关系p中元组的rid，**q**是q中匹配元组的rid，**r**是r中匹配元组的rid，按rid的升序来对三元组进行排序，首先考虑的是索引的第一个字段，然后是第二个字段，然后是第三个（索引也可能是B^+树）。

有了这样的索引，可以用扫描联结索引的简单循环来计算联结。对于索引中的每个三元组<**p**, **q**, **r**>，取回与rid为**p**，**q**和**r**相应的元组。因为索引首先在第一个字段上排序，所以可以在对索引和关系**p**的单次扫描中进行联结。然而，可能要多次存取关系**q**和**r**。实际上，如果N是对于**p**中每个元组来说**q**中匹配元组的平均数目，M是对于p中每个元组来说**r**中匹配元组的平均数目，那么为了计算联结可能要检索**q**的$|\mathbf{p}|\times N$页，**r**的$|\mathbf{p}|\times M$页。

多路联结索引对于加速所谓的星型联结的处理特别有用，星型联结是联机分析处理（OLAP，见第19章）中常用的联结类型。

星型联结（star join）是形如$\mathbf{r}\bowtie_{cond_1}\mathbf{r}_1\bowtie_{cond_2}\mathbf{r}_2\bowtie_{cond_3}\cdots$的多路联结，这里每个$cond_i$总只涉及**r**和$\mathbf{r}_i$的属性的联结条件。换句话说，没有条件能把$\mathbf{r}_i$和$\mathbf{r}_j$的元组直接联系起来，所有的匹配都是通过**r**的元组来完成的。星型联结的一个例子如图13-9所示，这里“卫星”关系COURSE、TEACHING和STUDENT是用等值联结条件和“恒星”关系TRANSCRIPT联结的，这些联结条件没有把卫星关系的属性彼此匹配。

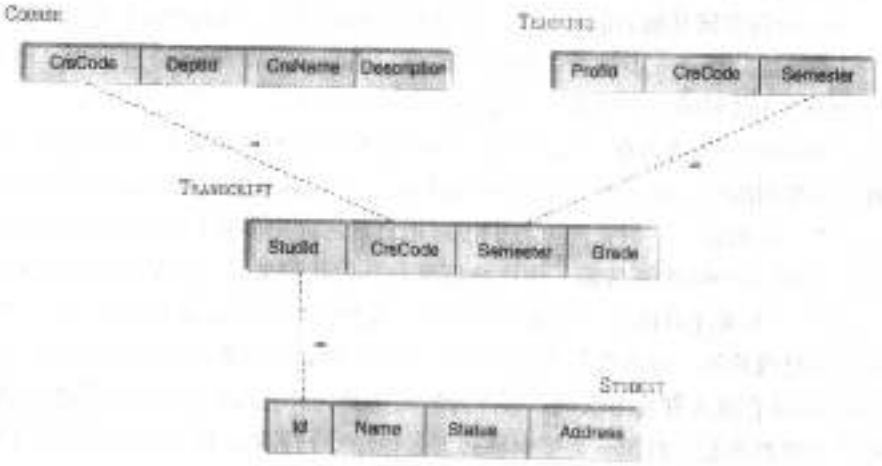


图13-9　星型联结

多路联结索引适合用于计算星型联结的一个原因是星型联结的联结索引比通常的多路联结（练习13.13）的联结索引更容易维护。另外，使用联结索引计算通常的多路联结的代价很大。考虑联结$\mathbf{r}\bowtie\mathbf{r}_1\bowtie\mathbf{r}_2\bowtie\cdots\bowtie\mathbf{r}_n$，假设N是对于**r**中每个元组来说匹配$\mathbf{r}_i$中元组的平均数目。那么，根据类似于对二路联结的分析，计算联结索引可能需要存取$|\mathbf{r}|\times N\times n$个页。

幸运的是，星型联结有着更为有效的方法。在[O'Neil and Graefe 1995]中描述的一种方法利用了在11.7.2节中介绍的位图联结索引。我们对每个部分连接$\mathbf{r}_i \bowtie \mathbf{r}$不是用涉及$n$个关系的一个联结索引，而是用一个位图联结索引$\delta_i$。每个$\delta_i$是<v, bitmap>对的集合，这里v是$\mathbf{r}_i$中一个元组的rid，当且仅当$\mathbf{r}$中第$i$个元组与用v表示的$\mathbf{r}_i$的元组联结时，bitmap在第$i$个位置上是1。然后我们可以扫描$\delta_i$，再对所有的位图进行逻辑或运算。这使我们得到了$\mathbf{r}$中所有可以和$\mathbf{r}_i$中某些元组联结的元组的rid。在得到每个卫星关系$\mathbf{r}_i$（$i$=1, …, n）的这样的逻辑析取的位图后，我们可以对这些位图进行逻辑与运算来得到$\mathbf{r}$中可以和每个$\mathbf{r}_i$中某些元组联结的所有元组的rid。换句话说，这个过程删去了$\mathbf{r}$中不参与星型联结的所有元组。原理是只会剩下少量元组，所以可以利用像嵌套循环那样的原始技术来计算联结，而代价却不大。

主要的商用数据库管理系统厂商（如IBM的DB/2，Oracle的Oracle 8i和Microsoft的SQL Server）的最新版本都支持联结索引和星型联结优化。

13.6 计算聚合函数

通常在查询中计算聚合函数（像AVG或者COUNT）涉及对查询输出作完整的扫描。这里唯一的问题是在有GROUP BY attrs语句时聚合的计算问题。问题又一次减小到找到根据某些属性的值来划分元组的有效技术。我们到目前为止已经确定了三种这样的技术。

- 排序
- 散列
- 索引

这三种技术都提供了存取由GROUP BY子句指定的元组的分组的有效方法。所剩下的就是对这些分组的成员元组应用聚合函数。

13.7 调优问题：对物理数据库设计的影响

应用程序设计者如何把关系运算的计算技术知识翻译成更好的系统设计呢？他们能够起作用的一个领域是物理数据库设计。尽管通常由数据库管理系统来决定如何组织数据，但是它也可以从数据库设计者那里得到提示。考虑到我们的讨论，特别感兴趣的提示是索引的声明。尽管SQL标准没有涉及这一领域，但是大多数数据库管理系统厂商允许用户指定要被索引的属性、索引的类型（散列、B^+树）以及索引是否是聚簇的。可以裁剪这些声明以便适合在每个特定的安装时期望的查询的特定组合。

为了说明这一思想，考虑TRANSCRIPT关系（参见图4-5），并假设大多数的查询是在属性Semester上的选择和联结。这时，最好的策略是要求数据库管理系统在那个属性上创建聚簇索引。B^+树索引保证可以快速找到满足任意特定选择（即使是对于范围选择）的索引叶子。因为索引是聚簇的，所以可以只用少数的几次页传输来检索实际的关系元组。另外，当数据库管理系统需要在Semester上作联结时，它可以使用排序-合并联结。因为关系在这个属性上已经排过序（因为索引是聚簇的），所以算法的排序步骤的主要部分就不会产生什么开销了。最后，因为表是动态的，所以对ISAM组织来说应该优先使用B^+树。假设还要在StudId属性上做选择和联结。不可能针对StudId属性提交范围查询，所以使用二级散列索引是合适的。

对于更加有意思的场景（仍然涉及TRANSCRIPT关系），假设最频繁的存取路径是通过StudId和CrsCode的组合（即大多数查询使用这个路径），较少用到的路径是通过Semester。显然，我们必须在<StudId, CrsCode>上创建一个索引，在Semester上创建另一个索引。然而，问题是哪一个索引应该是聚簇的？初看一下，似乎<StudId, CrsCode>上的索引应该是聚簇的，因为这是到达关系主要的存取路径。然而，仔细看一下这个关系的语义就使我们确信尽管<StudId, CrsCode>不是这个关系的候选键，但是在这两个属性上一致的TRANSCRIPT记录的数目在几乎所有的情况下都是1——只有当学生重修了一门课程的时候，这个数字才可能大于1。而且不可能针对这一对属性提出范围查询，所以B^+树索引的开销看起来并不是很合理——散列索引可能是这种情况下最好的解决方法。另一方面，Semester上聚簇的B^+树索引可以在很大程度上提高在那个属性上选择和联结的效率，它是二级存取路径的一个很好的选择[⊖]。

13.8 参考书目

在[Blasgen and Eswaran 1977]中讨论了关系运算符的基于排序的计算技术，在[DeWitt et al.1984, Kitsuregawa et al. 1983]中讨论了基于散列的技术。在[Graefe 1993, Chaudhuri 1998]中可以找到计算关系运算符技术的简介和其他的参考资料。在[Valduriez 1987]中研究了使用联结索引来计算多关系联结的内容，在[O'Neil and Graefe 1995, O'Neil and Quass 1997]中讨论了在位图索引的帮助下计算各种关系运算符的技术。

13.9 练习

13.1 考虑用非聚簇的B^+树来做外部排序。假设R表示每个磁盘块上的数据记录的数目，F表示数据文件中块的数目。估计这样的排序过程的代价（表示为R和F的函数）。把这个代价和基于合并的外部排序的代价进行比较。考虑R=1、10和100的情况。

13.2 假设在初始扫描（在这里删除了元组分量）的过程中，原来关系的大小的缩小率为因子$\alpha<1$，估计基于排序的投影的代价。

13.3 考虑投影运算符基于散列的计算方法。假设所有桶的大小都基本相同，但是不能放进主存。设N是内存页中散列表的大小，F是页中初始关系的大小，$\alpha<1$是因为投影而导致的减小因子。估计为计算投影而需要向磁盘传输的和从磁盘传输的页的数目。

13.4 举例说明TRANSCRIPT关系（图4-5）的实例和属性序列<StudId, Grade>上的散列函数，这个散列函数可以把$\pi_{StudId,Semester}$(TRANSCRIPT)中两个相同的元组发送到不同的散列桶中去。（这表明这样的基于散列的存取路径不能用于计算投影。）

13.5 存取路径的选择度理论上的最小值显然是包含涉及到的关系运算符的输出的页的数目。当涉及选择或者投影运算符时，存取路径的选择度的最合理的理论上界是多少？

13.6 基于13.3.2节中的讨论，给出何时存取路径将覆盖投影、并和差运算符的使用的准确定义。

13.7 考虑表达式

$$\sigma_{StudId=666666666 \wedge Semester='F1995' \wedge Grade='A'}(\text{TRANSCRIPT})$$

假设存在下面的存取路径：

- 在StudId上的非聚簇的散列索引

⊖ 注意，这种情况下在主键<StudId, CrsCode, Semester>上的主索引可能不是聚簇的。

- 在Semester上的非聚簇的散列索引
- 在Grade上的非聚簇的散列索引

这些存取路径中，哪个路径有着最合理的选择度，哪个路径的选择度最差？把最差的存取路径（在上面三种中）的选择度和文件扫描的选择度作一比较。

13.8 用下面的方法计算$r \bowtie_{A=B} s$的代价。

- 嵌套循环
- 块嵌套循环
- 索引嵌套循环

这里r有2000个页，每页上有20个元组，s有5000个页，每页上有5个元组，可用于块嵌套循环联结的内存数量是402页。

*13.9 在排序-合并联结$s \bowtie r$中，合并阶段中的额外扫描看起来是没有必要的。我们应该能够在排序算法的最后合并阶段中匹配s和r的元组。具体设计这一算法。

13.10 估计用排序-合并联结来计算$r \bowtie_{A=B} s$所需要传输的页的数目，假设：

- r的大小是1000页，每页上有10个元组；s的大小是500页，每页上有20个元组。
- 用于计算这个联结的内存缓冲区的大小是10页。
- r和s（见图13-7）中匹配元组的笛卡儿积是用块嵌套循环联结来计算的。
- r[A]有100个不同的值，s[B]有50个不同的值。这些值或多或少平均地分布在这些文件中，所以$\sigma_{A=c}(r)$的大小（$c \in r[A]$）不会随c有大的变化。

13.11 本章讨论的计算联结的方法都涉及等值联结。讨论计算像$r \bowtie_{A<B} s$这样的不等值联结问题的适用性。

13.12 考虑关系模式R(A, B)，它具有下面的特性：

- 元组的总数目：1 000 000。
- 每页有10个元组。
- 属性A是候选键，在1 ~ 1 000 000之间变化。
- 在A上有深度为4的聚簇的B^+树索引。
- 属性B有100 000个不同的值。
- 在B上的散列索引。

对于下面每种建议的方法来说，估计计算下面每个查询所需要传输的页的数目：

- $\sigma_{A<3000}$：顺序扫描；在A上有索引。
- $\sigma_{A>3000 \wedge A<3200 \wedge B=5}$：在A上有索引；在B上有索引。
- $\sigma_{A \neq 22 \wedge B \neq 66}$：顺序扫描；在A上有索引；在B上有索引。

13.13 设计对于多路星型联结的联结索引进行增量维护的算法。

13.14 设计使用联结索引的联结算法。定义聚簇联结索引（在二分联结的情况下有三种可能的情况！）的概念，并考虑聚簇对联结算法的影响。

第14章 查询优化概述

本章是通常用于数据库管理系统的关系查询优化技术的概述。本章的目的并不是要把你培养成数据库管理系统的实现者，而是想让你成为更好的应用程序设计者或者数据库管理员。就像了解用于关系代数的计算技术的知识有助于更好地进行物理设计一样，了解查询优化的原则有助于编写更有可能被查询优化器改善的SQL查询。

关系查询优化是使用相对简单的启发式搜索算法来处理广泛的计算复杂性的一个很好的例子。在[Garcia-Molina et al. 2000]中可以找到关于这个主题更为全面的讨论。

14.1 查询处理概述

当用户提交一个查询的时候，首先由数据库管理系统的解析器来解析这个查询。解析器验证这个查询的语法，并使用系统目录来决定属性引用是否正确。例如，TRANSCRIPT.Student不是一个正确的引用，因为关系TRANSCRIPT并没有Student这个属性。同样，把AVG操作符应用于CrsCode属性也与这个属性的类型不相匹配。

因为SQL查询是声明性的，而不是过程性的，所以它并不指定任何特定的实现。因而，必须首先把解析过的查询转换为关系代数表达式，然后可以直接使用第13章中给出的算法来计算它。就像6.2.1节所解释的那样，下面这个典型的SQL查询：

```
SELECT   DISTINCT TargetList
FROM     REL1 V1, ···, RELn Vn
WHERE    Condition
```
(14.1)

通常被翻译成下面的关系代数表达式：[⊖]

$$\pi_{TargetList}(\sigma_{Condition'}(REL_1 \times \cdots \times REL_n)) \tag{14.2}$$

这里，Condition'是Condition从SQL语法转换到关系代数表达式的形式。例如，WHERE子句

```
T.Semester = 'F1995' AND P.Id = T.ProfId
        AND T.CrsCode = C.CrsCode
```

可翻译成选择条件

```
Semester = 'F1995' AND Id = ProfId
        AND TEACHING.CrsCode = COURSE.CrsCode
```

但是，上面从SQL直接进行翻译而产生的代数表达式（14.2）可能要花费很长时间（确实的！）来计算。一方面，它包含有笛卡儿积，所以联结四个含100个块的关系的将产生含10^8个块的中间关系，假定磁盘速度是10ms/页，那么用50小时才仅仅能把它全部写出去。即使我们设法把笛卡儿积转换成等值联结（就像本章后面所解释的那样），我们仍然要花费很长的时

⊖ 为了简化符号，我们忽略在笛卡儿积中一些属性可能需要重命名的情况。

间转换上面的查询（几十分钟）。**查询优化器**（query optimizer）的作用就是要把这个时间降到几秒钟（对很复杂的查询来说降到几分钟）。

典型的查询优化器结合使用启发式算法和代价估计方法来选择查询执行计划。因而，查询优化器的两个主要组成部分是**查询执行计划生成器**（query execution plan generator）和**计划代价估计器**（plan cost estimator）。**查询执行计划**（query execution plan）可以看作是一个关系表达式，这个表达式中关系操作符的每次出现都附加有具体的计算方法（即我们在第13章中所说的存取路径）。因而，查询优化器的主要任务是产生一个查询执行计划来计算给定的关系表达式。然后，这个计划可以传递给查询执行计划解释器，这个软件组件根据给定的计划直接负责计算查询。查询处理的总体体系结构如图14-1所示。

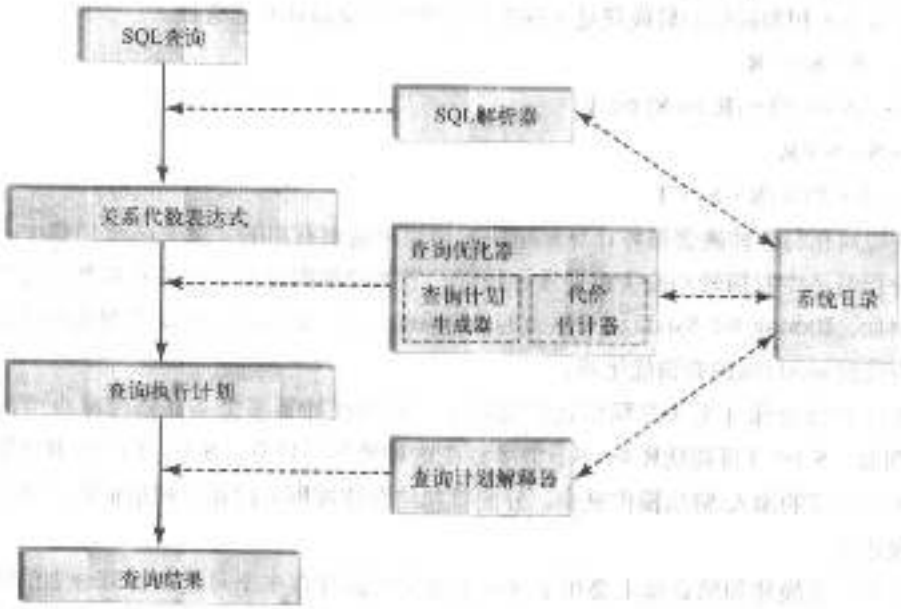


图14-1　数据库管理系统查询处理的典型体系结构

14.2　基于代数等价的启发式优化

用于计算关系查询的启发式算法（大多数）是基于简单的观察，比如联结较小的关系优于联结较大的关系，等值联结优于笛卡儿积，在一次关系扫描中计算若干操作优于在多次扫描中计算同样多次操作。大多数这样的启发式算法都可以用关系代数转换的形式来表达，它把一个表达式转换成另一个不同但等价的表达式。并不是所有的转换都会产生优化的等价表达式，有时它们产生“不太有效”的表达式。然而，关系转换与其他转换一起产生总体上更好的表达式。

我们现在给出一些用于查询优化器的启发式转换。

1. 基于选择和投影的转换

* $\sigma_{cond_1 \wedge cond_2}(\mathbf{R}) = \sigma_{cond_1}(\sigma_{cond_2}(\mathbf{R}))$。这个转换称为**选择的级联**（cascading of selection）。它本质上并不是优化，但是它在与其他转换一起使用时很有用（参见下面关于通过联结

来推动选择和投影的讨论）。

- $\sigma_{cond_1}(\sigma_{cond_2}(\mathbf{R})) \equiv \sigma_{cond_2}(\sigma_{cond_1}(\mathbf{R}))$。这个转换称为**选择的交换**（commutativity of selection）。像级联一样，它在与其他转换一起使用时很有用。
- 如果 $attr \subseteq attr'$，则$\pi_{attr}(\mathbf{R}) \equiv \pi_{attr}(\pi_{attr'}(\mathbf{R}))$。这个等价称为**投影的级联**（cascading of projection），主要是与其他的转换一起使用。
- 如果*attr*包含了用于*cond*的所有属性，则$\pi_{attr}(\sigma_{cond}(\mathbf{R})) \equiv \sigma_{cond}(\pi_{attr}(\mathbf{R}))$。这个等价称为**选择和投影的交换**（commutativity of selection and projection）。它通常用作通过联结操作符推动选择和投影的一个准备步骤（下面将做描述）。

笛卡儿积和联结的转换

用于笛卡儿积和联结的转换就是这些操作符常用的交换律和结合律。

- $\mathbf{R} \bowtie \mathbf{S} \equiv \mathbf{S} \bowtie \mathbf{R}$
- $\mathbf{R} \bowtie (\mathbf{S} \bowtie \mathbf{T}) \equiv (\mathbf{R} \bowtie \mathbf{S}) \bowtie \mathbf{T}$
- $\mathbf{R} \times \mathbf{S} \equiv \mathbf{S} \times \mathbf{R}$
- $\mathbf{R} \times (\mathbf{S} \times \mathbf{T}) \equiv (\mathbf{R} \times \mathbf{S}) \times \mathbf{T}$

这些规则在与各种嵌套循环计算策略一起使用时是很有用的。就像在第13章中看到的那样，在外层循环中扫描较小的关系通常比较好，上面的规则有助于把关系调整到适当的位置上去。例如，BIGGER ⋈ SMALLER可以改写成SMALLER ⋈ BIGGER，这就直观地对应决定在外层循环中使用SMALLER的查询优化器。

交换律和结合律（至少在联结这种情况下）可以在计算多关系联结时减少中间关系的大小。例如，$\mathbf{S} \bowtie \mathbf{T}$可能比$\mathbf{R} \bowtie \mathbf{S}$小得多，在这种情况下计算$(\mathbf{S} \bowtie \mathbf{T}) \bowtie \mathbf{R}$可能比计算$(\mathbf{R} \bowtie \mathbf{S}) \bowtie \mathbf{T}$的输入/输出操作更少。分配律和结合律规则可以用于把后面的表达式转换成前面的表达式。

实际上，交换律和结合律主要用于同一个查询可能存在多个可选的计算计划的情况。涉及N个关系联结的查询可能有$T(N) \times N!$个只是处理联结的查询计划，这里的$T(N)$是有N个叶子节点的不同二叉树的数目。（$N!$是N个关系的排列的数目，$T(N)$是为一个特定的排列加上括弧的不同方式的数目。）这个数目即使对于很小的N来说也会增长得非常迅速，并且很大[⊖]。类似的结果对于其他可交换和可结合的关系也是成立的（比如并），但是我们关注的是联结，因为它的计算代价最大。

查询优化器的任务就是估计这些计划的代价（这些计划的代价可能会有很大的不同），并选择一个“好的”计划。因为计划的数目很大，所以找到一个好的计划的时间可能比直接计算查询所花的时间还要多（进行10^6次输入/输出比15!次内存操作还要快）。为了让查询优化变得实用一些，优化器通常只是检查所有可能的计划的一个小的子集，所以它的代价估计最多也是近似的。因此，查询优化器很可能没有找到最优的计划，而实际上只是找到“合理的”计划。换句话说，在“查询优化器”中的“优化”并不总是满足需要，因为它并不足以描述数据库管理系统的体系结构的某个组件所完成的事情。

⊖ 当N=4时，$T(4)$是5，所有计划的数目是120。当N=5时，$T(5)$是14，所有计划的数目是1680。

2. 通过联结和笛卡儿积推动选择和投影

- $\sigma_{cond}(\mathbf{R} \times \mathbf{S}) \equiv \mathbf{R} \bowtie_{cond} \mathbf{S}$。当*cond*与R和S的属性都相关的时候使用这个规则。这种启发式方法的基础就是认为不应该物化笛卡儿积，选择必须和笛卡儿积组合使用，并且应该使用计算联结的技术。只要创建$\mathbf{R} \times \mathbf{S}$的一行就应用选择条件，我们可以节省一遍扫描，并且避免存储很大的中间关系。
- 如果用于*cond*的属性都属于R，则$\sigma_{cond}(\mathbf{R} \times \mathbf{S}) \equiv \sigma_{cond}(\mathbf{R}) \times \mathbf{S}$。这个启发式方法基于这样的想法：如果必须计算笛卡儿积，那么我们也应该让所涉及的关系尽可能地小。通过把选择下推到R，我们希望在R应用到交积之前就减少它的大小。
- 如果*cond*中的属性都属于R，则$\sigma_{cond}(\mathbf{R} \bowtie \mathbf{S}) \equiv \sigma_{cond}(\mathbf{R}) \bowtie \mathbf{S}$。这里的原理和对于笛卡儿积的方法的原理是一样的。计算联结的代价可能非常大，我们必须努力减小所涉及的关系的大小。注意，如果*cond*是比较条件的合取，那么我们可以把每个合取分量分开推到R或者S上去，只要在分量中命名的属性只属于一个关系。
- 如果$attributes(\mathbf{R}) \supseteq attr' \supseteq (attr \cap attributes(\mathbf{R}))$，那么$\pi_{attr}(\mathbf{R} \times \mathbf{S}) \equiv \pi_{attr}(\pi_{attr'}(\mathbf{R}) \times \mathbf{S})$。这里$attributes(\mathbf{R})$表示R的所有属性的集合。这个规则的原理是通过把投影推进笛卡儿积，我们可以减小其中一个操作数的大小。在第13章中，我们看到联结操作（“×”是一个特例）的输入/输出复杂性是和涉及的关系的页数成比例的。因此，较早地应用投影可以减少计算叉积所需传递的页的数目。
- 如果$attributes(\mathbf{R}) \supseteq attr' \supseteq (attr \cap attributes(\mathbf{R}))$，并且*attr'*包含在*cond*中提到的R的所有那些属性，那么$\pi_{attr}(\mathbf{R} \bowtie_{cond} \mathbf{S}) \equiv \pi_{attr}(\pi_{attr'}(\mathbf{R}) \bowtie_{cond} \mathbf{S})$。这里潜在的好处是和叉积一样的。另一个重要的需求是*attr'*必须包含在*cond*中提到的R的所有那些属性。如果这些属性中的一部分被投影掉了，那么表达式$\pi_{attr'}(\mathbf{R}) \bowtie_{cond} \mathbf{S}$将会出现句法错误。这个需求在笛卡儿积的情况下是没有必要的，因为在笛卡儿积中不涉及联结条件。

通过联结和叉积推动选择和投影的规则在与级联σ和π规则结合使用时特别有用。例如，考虑表达式$\sigma_{c_1 \wedge c_2 \wedge c_3}(\mathbf{R} \times \mathbf{S})$，这里$c_1$涉及R和S的属性，$c_2$只涉及R的属性，$c_3$只涉及S的属性。我们通过首先级联选择，然后把它们下推，最后消除笛卡儿积来把这个表达式转换成能够更有效计算的表达式：

$$\sigma_{c_1 \wedge c_2 \wedge c_3}(\mathbf{R} \times \mathbf{S}) \equiv \sigma_{c_1}(\sigma_{c_2}(\sigma_{c_3}(\mathbf{R} \times \mathbf{S}))) \equiv \sigma_{c_1}(\sigma_{c_2}(\mathbf{R}) \times \sigma_{c_3}(\mathbf{S})) \equiv \sigma_{c_2}(\mathbf{R}) \bowtie_{c_1} \sigma_{c_3}(\mathbf{S})$$

我们可以用类似的方式来优化涉及投影的表达式。例如，考虑$\pi_{attr}(\mathbf{R} \bowtie_{cond} \mathbf{S})$。假设$attr_1$是R中属性的一个子集，使$attr_1 \supseteq attr \cap attributes(\mathbf{R})$，$attr_1$包含*cond*中的所有属性。假定$attr_2$是对于S来说类似的集合，那么

$$\pi_{attr}(\mathbf{R} \bowtie_{cond} \mathbf{S}) \equiv \pi_{attr}(\pi_{attr_1}(\mathbf{R} \bowtie_{cond} \mathbf{S})) \equiv \pi_{attr}(\pi_{attr_1}(\mathbf{R}) \bowtie_{cond} \mathbf{S}) \equiv$$
$$\pi_{attr}(\pi_{attr_2}(\pi_{attr_1}(\mathbf{R}) \bowtie_{cond} (\mathbf{S})) \equiv \pi_{attr}(\pi_{attr_1}(\mathbf{R}) \bowtie_{cond} \pi_{attr_2}(\mathbf{S}))$$

这样得到的表达式更为有效，因为它联结的是较小的关系。

使用代数等价规则

通常上面的代数规则用于把关系代数表达的查询转换成比最初的查询更好的表达式。这里“更好”这个词并不总是真的，因为用来指导转换的标准就是启发式的。实际上，在下一节中，我们将看到用建议的转换并不一定产生最好的结果。因而代数转换步骤的结果应该产

生一组候选查询，然后再用下一节中讨论的代价估计技术进行进一步检查。这里给出了应用代数等价方法的典型的启发式算法。

1) 使用选择的级联规则来打破选择条件的连接。结果，将单个的选择转换成一系列的选择操作符，然后可以单独应用每个操作符。

2) 上一个步骤将会使通过联结和笛卡儿积推动选择具有更多的自由。我们现在可以使用选择的交换律和通过连接推动选择等规则来尽可能的把选择传播进查询中去。

3) 把笛卡儿积操作和选择结合起来形成联结。就像在第13章中看到的那样，计算联结有十分有效的技术，但是可以改善笛卡儿积的计算的方法就很少了。因而，把这些乘积转换成联结可以节省时间和空间。

4) 使用联结和笛卡儿积的分配律来重新安排联结操作的顺序。这里的目的是设法得到一个产生最小的中间关系的顺序。(注意，中间关系的大小直接影响到开销，所以减小它们的大小会加速查询的处理速度。) 在下一节中将会讨论估计中间关系大小的技术。

5) 使用级联投影和把它们推进查询的规则来尽可能地把投影传播到查询中去。这可以通过减小操作数的大小来加速联结的计算。

6) 找出可以在同一遍扫描中处理的操作，以节省把中间关系写进磁盘的时间。这一技术称为**管道**，将在下一节中进行介绍。

14.3 估计查询执行计划的代价

就像前面定义的那样，查询执行计划从某种意义上说就是关系表达式，其中的每一个操作都附加有具体的计算方法（存取路径）。在这一节中，我们将进一步探讨这一概念，并讨论计算计划（为了计算查询结果）的代价的方法。

为了讨论的方便，我们把查询表示成树。在**查询树**（query tree）中，每个内部节点用关系操作符来标记，每个叶子节点用关系名来标记。一元关系操作符只有一个孩子，二元操作符有两个孩子。图14-2分别表示了对应下面等价关系表达式的四棵查询树:

$$\pi_{Name}\ (\sigma_{DeptId = 'CS' \wedge Semester = 'F1994'}\ (\text{PROFESSOR} \bowtie_{Id = ProfId} \text{TEACHING})) \tag{14.3}$$

$$\pi_{Name}\ (\sigma_{DeptId = 'CS'}\ (\text{PROFESSOR}) \bowtie_{Id = ProfId} \sigma_{Semester = 'F1994'}\ (\text{TEACHING})) \tag{14.4}$$

$$\pi_{Name}\ (\sigma_{Semester = 'F1994'}\ (\sigma_{DeptId = 'CS'}\ (\text{PROFESSOR}) \bowtie_{Id = ProfId} \text{TEACHING})) \tag{14.5}$$

$$\pi_{Name}\ (\sigma_{DeptId = 'CS'}\ (\text{PROFESSOR} \bowtie_{Id = ProfId} \sigma_{Semester = 'F1994'}\ (\text{TEACHING}))) \tag{14.6}$$

关系PROFESSOR和TEACHING在图4-5中描述过。

对应图14-2a的表达式（14.3）是查询处理器从下面的SQL查询最初生成的。

```
SELECT   P.Name
FROM     PROFESSOR P, TEACHING T
WHERE    P.Id = T.ProfId AND T.Semester = 'F1994'
         AND P.DeptId = 'CS'
```
(14.7)

对应图14-2b的第二个表达式（14.4）是从第一个表达式中得来的，即像前一节的启发式规则建议的那样通过联结来推动选择。对应图14-2c和14.2d的第三个和第四个表达式是从（14.3）中得来的，它们只是把选择条件的一部分下推到实际的关系上去而得到的。

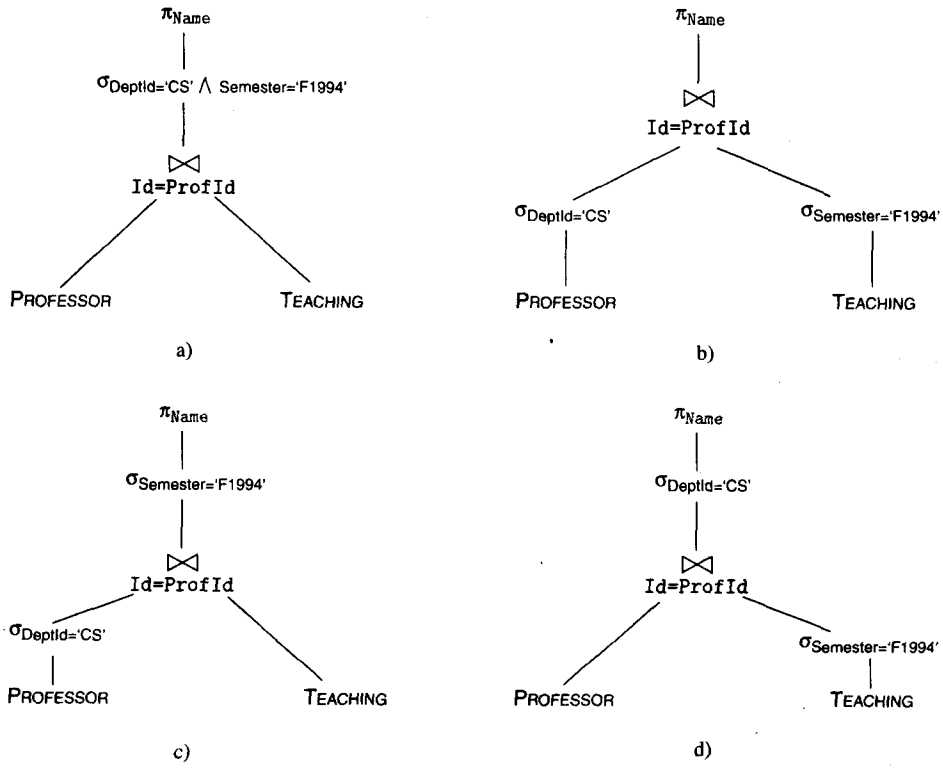


图14-2　关系表达式（14.3）～（14.6）的查询树

我们现在来用各种查询执行计划来扩充这些查询树，并估计每棵树的代价。

假定下面的信息在系统目录中的这些关系上是可用的。

• Professor

大小：200页，关于50个系中的教授的1000个记录（5个元组/页）。

索引：在DeptId上聚簇的2级B^+树索引，在Id上的散列索引。

• Teaching

大小：1000个页，四个学期的10 000个教学记录（10个元组/页）。

索引：在Semester上聚簇的2级B^+树索引，在ProfId上的散列索引。

在继续之前，我们还需要另一个信息：关系PROFESSOR中属性Id的权重和关系TEACHING中属性ProfId的权重。通常，关系r中属性A的**权重**（weight）是匹配属性A的不同值的元组的平均数目。换句话说，权重是$\sigma_{A = value}(r)$中元组的平均数目，这里的“平均”是就r中A的所有值来说的。

各种属性的权重通常是从存储在系统目录中的静态信息中导出的，并由数据库管理系统来维护。当前的查询优化器进一步为一个特定属性中的值的分布维护了*柱状图*。关于对于给定的属性值来说有多少元组可能被选择到，柱状图给出了更为精确的信息。需要用属性权重来估计使用基于索引的技术计算联结的代价，以及在我们的例子中估计所有操作的结果的大小。因为各种操作的中间结果可能会在后来用作其他操作符的输入，所以知道其大小对于估计每个具体计划的代价是重要的。

回到我们的例子，我们首先要找到属性Id和ProfId的实际权重。对于PROFESSOR中的属性Id来说，权重必须是1，因为Id是键。对于TEACHING中的ProfId的权重来说，我们假设每个教授可能教了相同数目的课程。因为有1000个教授和10 000个教学记录，所以ProfId的权重必须大约是10。现在我们来考虑图14-2中的四种情况。在这四种情况中，我们都假设有51页的缓冲区来计算联结，外加少量的内存来存放一些索引块（具体的数量在必要的时候将会指定）。

1. 情况a：没有推动选择

计算联结的一种可能方法是索引嵌套循环方法。例如，我们可以在外层循环中使用较小的关系PROFESSOR。因为在Id和ProfId上的索引不是聚簇的，并且因为PROFESSOR中的每个元组可能匹配TEACHING中的某个元组（通常每个教授总要教点什么），所以可以用如下过程来估计代价。

- 扫描PROFESSOR关系　200页传输。
- 在TEACHING中找出匹配的元组　我们可以使用缓冲区中的50页来存放PROFESSOR关系的页。因为在每个这样的页中有5个PROFESSOR的元组，又因为每个元组匹配10个TEACHING元组，所以PROFESSOR关系的50页的块平均可以匹配50 × 5 × 10 = 2500个TEACHING的元组。因为在TEACHING的ProfId属性上的索引不是聚簇的，所以不对从其中检索出的记录Id排序。结果，取得数据文件中匹配的行的代价（暂时不考虑从索引中取得Id的代价）就可能是2500页传输。然而通过首先对这些匹配元组的记录Id排序，我们可以保证在不超过1000页的传输（TEACHING关系的大小）中取得这些元组（这要用到13.3.1节描述的技术）。因为这个技巧必须执行四次（对PROFESSOR的每个50页的块来说），所以取得TEACHING中匹配元组的总的页传输的数目是4000。
- 搜索索引　因为TEACHING在ProfId上有散列索引，所以我们假设每次索引搜索有1.2次输入/输出。对于每个ProfId来说，搜索找出包含所有匹配元组（平均10个）的记录Id的桶。这些Id可以在一次输入/输出中检索到。因此，在TEACHING中元组的1000个匹配记录Id可以以每次输入/输出10个元组的速度来检索，总共要1000次输入/输出。对所有元组进行索引搜索总的代价是1200。
- 总的代价　5200次页传输。

我们还可以使用块嵌套的循环联结或者排序-合并联结。对于利用51页内存缓冲区的块嵌套循环来说，必须扫描4次内层关系TEACHING。这就使得页传输的数目变小：200+4 × 1000 = 4200。但注意，如果ProfId的权重比较低，那么索引嵌套技术和块嵌套技术之间的区别就很大了（练习14.4）。

联结的结果将有10000个元组（因为Id是PROFESSOR的键，每个PROFESSOR元组匹配大约10个TEACHING元组）。因为每个TEACHING元组的大小是PROFESSOR元组大小的两倍，所以得到的文件将会比TEACHING大50%。换句话说，它将有1500页。

下一步我们要应用选择和投影操作符。因为联结的结果没有任何索引，所以我们选择文件扫描存取路径。另外，我们可以在同一遍扫描中应用选择和投影。依次检查每个元组，如果它不满足选择条件，我们就丢弃它；如果它满足选择条件，我们就丢弃没有在SELECT子句中命名的属性，并输出结果。

我们可以把联结阶段和选择/投影阶段分开处理，输出联结的结果到中间文件，然后输入

这个文件来做选择/投影，但是有一个更好的办法。通过使这两个阶段交错进行，我们可以省去与创建和存取中间文件的输入/输出操作。利用称为**管道**（pipelining）的技术，联结和选择/投影可以作为协同例程来操作。执行联结阶段直到用完内存中可用的缓冲区为止，然后由选择/投影接管，清空缓冲区，输出结果。然后重复联结阶段，填满缓冲区，处理继续进行，直到选择/投影输出最后的元组为止。在管道中，一个关系操作符的输出“通过管道传送”到下一个关系操作符的输入，从而省去了磁盘上的中间结果。

得到的查询执行计划如图14-3a所示。总之，使用块嵌套循环策略计算这个计划花费了4200+α × 1500页输入/输出，这里1500是联结的大小，α是0 ~ 1之间的一个数字，它是因为选择和投影而导致的减小因子。我们在本章的后面要研究估计这个减小因子的技术。

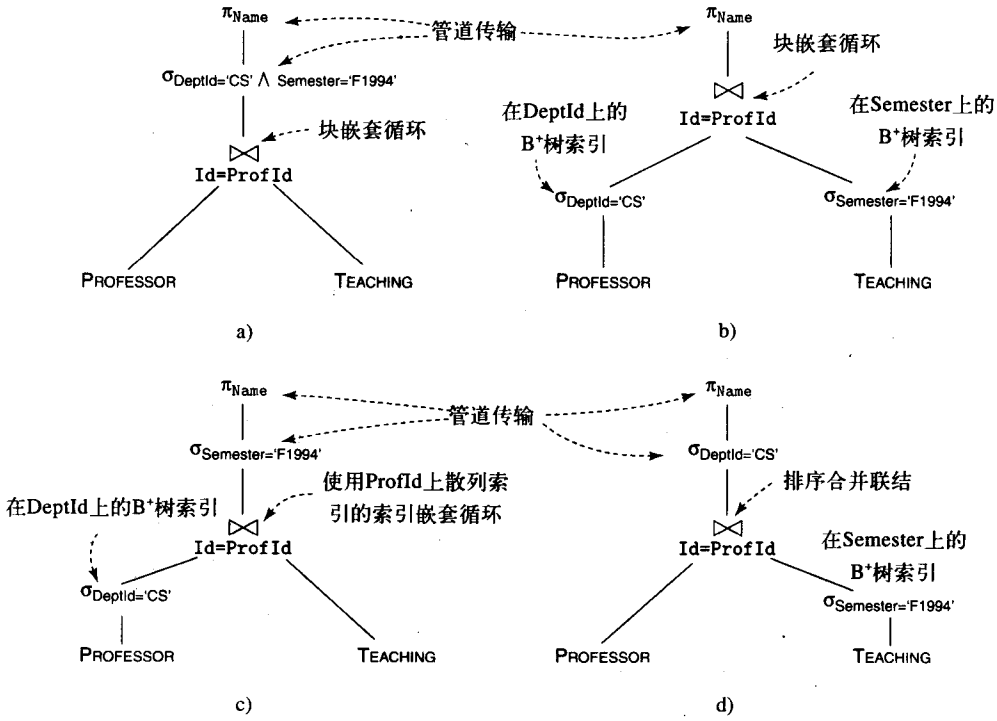


图14-3　（14.3）到（14.6）关系表达式的查询执行计划

2. 情况b：全部推动选择

图14-2b中查询树给出了很多可选的查询执行计划。首先，如果我们把选择下推到树的叶子节点（关系PROFESSOR和TEACHING），那么我们可以使用已存的在DeptId和Semester上的B+树索引计算关系$\sigma_{DeptId = 'CS'}$(PROFESSOR)和$\sigma_{Semester = 'F1994'}$(TEACHING)。然而，得到的关系就没有任何索引了（除非数据库管理系统决得值得创建它们）。特别的，我们不能利用PROFESSOR.Id和TEACHING.ProfId上的散列索引。因此，我们必须使用块嵌套循环或者排序-合并来计算联结。然后当把联结的结果写到磁盘上的时候，即时地应用投影到联结的结果上。换句话说，我们又一次使用了管道来把应用投影操作的开销最小化。

我们估计图14-3b中所示的计划的代价，这里是使用块嵌套循环来进行联结。因为在50个系里有1000个教授，所以PROFESSOR关系中DeptId的权重是20。因此，$\sigma_{DeptId = 'CS'}$（PROFESSOR）

的大小是20个元组左右，即4页。TEACHING中Semester的权重是10000/4＝2500个元组，即250页。因为DeptId和Semester上的索引是聚簇的，所以计算选择将花费4次输入/输出（来存取这两个索引）+4+250次输入/输出（来存取两个关系中符合条件的元组）。

可以在每个文件的一遍扫描中进行联结操作（因为缓冲区可以容纳整个PROFESSOR关系）。因此，总的代价是4+4+250+4+250＝512。

3. 情况c：推动选择到PROFESSOR关系

对于图14-2c中的查询树，可以用如下方法构造查询执行计划。首先，使用PROFESSOR.DeptId上的B^+树索引计算$\sigma_{DeptId = 'CS'}$(PROFESSOR)。就像在情况b中那样，这将使得我们在随后的联结计算中不能进一步使用PROFESSOR.Id上的散列索引。然而和情况b不同的是，关系TEACHING保持不变，所以我们仍然可以使用索引嵌套循环（利用在TEACHING.ProfId上的索引）来计算连接。其他的可能是块嵌套循环和排序-合并联结。最后，我们可以利用管道传输联结的输出到选择操作符$\sigma_{DeptId = 'F1994'}$，并在同一遍扫描中应用投影。

上面的查询执行计划如图14-3c所示。我们现在来估计这个计划的代价。

- $\sigma_{DeptId = 'CS'}$(PROFESSOR)。有50个系，1000名教授。因此，这个选择的结果将包含大约20个元组，即4个页。因为PROFESSOR.DeptId上的索引是聚簇的，所以检索这些元组应该花费大约4次输入/输出。索引搜索将会为2级B^+树索引花费2次额外的输入/输出。因为我们想把选择的结果通过管道输入到接下来的联结步骤，所以没有输出代价。
- 索引嵌套循环连接。我们使用前面选择的结果并把它直接作为输入通过管道输出给联结。这里要考虑的一个重要问题是，因为我们选择的是利用了TEACHING.ProfId上的散列索引的索引嵌套循环，所以选择的结果即使很大也不需要保存在磁盘上。一旦选择产生足够多的元组来填满缓冲区，我们就可以马上使用散列索引把这些元组和匹配的PROFESSOR元组联结起来，并且输出连接后的元组。然后我们重新进行选择，并再次填满缓冲区。和以前一样，每个PROFESSOR元组匹配大约10个TEACHING元组，它们将要存储在一个桶里。所以，为了找出20个元组的匹配，我们不得不搜索索引20次，每次搜索的代价是1.2次输入/输出。另外需要200次输入/输出来从磁盘上取得匹配的元组（因为索引是非聚簇的）。总之，这应该要花费1.2 × 20+200＝224次输入/输出。
- 总的代价。因为连接的结果通过管道传输给下一个选择和投影，所以这些后面的操作就输入/输出来说并不花费什么。因此，总的代价是4+2+224＝230次输入/输出。

4. 情况d：推动选择到TEACHING关系

这种情况类似于情况c，只是现在将选择应用到TEACHING而不是PROFESSOR。因为TEACHING上的索引在应用选择后就丢失了，所以我们不能在索引的嵌套循环联结的内层循环中使用这个关系。然而，我们可以在索引嵌套的循环联结（它在内层循环中利用了PROFESSOR.DeptId上的散列索引）的外层循环中使用它。这个联结也可以使用块嵌套循环和排序合并来计算。在这个例子中，我们选择了排序合并。之后对结果应用选择和投影可以像前面的例子中那样使用管道来完成。得到的查询计划如图14-3d所示。

- 联结：排序阶段。第一步是在Id上对PROFESSOR进行排序，在ProfId上对$\sigma_{Semester = 'F1994'}$(TEACHING)进行排序。

- 为了对PROFESSOR进行排序，我们必须首先对它进行扫描，创建排序的运行。因为PROFESSOR可以放进200块中，所以将会有200/50＝4个排序的运行。因此，四个排序的运行的创建和把它们排序回磁盘要花费2 × 200＝400次输入/输出。这些运行可以只在一遍扫描中合并，但是我们推延这个合并，把它和排序合并联结的合并阶段结合起来（如下所示）。
- 为了对$\sigma_{Semester = 'F1994'}$(TEACHING)进行排序，我们必须首先计算这个关系。因为TEACHING保存了大约4个学期的信息，所以选择的大小约为10000/4＝2500个元组。因为索引是聚簇的，所以元组连续地存储在这个文件的250个块中。因此选择的代价大约是252次输入/输出操作，其中包含为了搜索索引所做的2次输入/输出操作。

  选择的结果不是被写进磁盘。每次内存中的50页缓冲区充满的时候，就立即对它进行排序以产生一个运行，然后将其写到磁盘上去。用这种方式，我们创建了250/50＝5个排序的运行。这要花费250次输入/输出。

  $\sigma_{Semester = 'F1994'}$(TEACHING)的5个已排序的运行可以在一遍扫描中合并。然而，我们不是单独地进行这个操作，而是把这一步骤和联结的合并步骤（以及之前被推延的排序PROFESSOR的合并步骤）结合起来。

- 联结：合并阶段。不是将PROFESSOR的4个已排序的运行和$\sigma_{Semester = 'F1994'}$(TEACHING)的已排序的运行合并到两个已排序的关系中去，而是将运行直接通过管道传输到排序合并联结的合并阶段，而不是把中间排序的结果写到磁盘上去。用这种方法，我们在排序这些关系的时候把最终的合并步骤和联结的合并步骤结合起来。

  组合的合并为PROFESSOR的每个已排序的运行使用了4个输入缓冲区，为$\sigma_{Semester = 'F1994'}$(TEACHING)的每个已排序的运行使用了5个输入缓冲区，并为联结的结果使用一个输出缓冲区。选择元组p和p.Id在4个PROFESSOR的运行的头中最低的值，并把它和元组t和t.ProfId在5个与$\sigma_{Semester = 'F1994'}$(TEACHING)对应的运行的头中最低的值相匹配。如果p.Id＝t.ProfId，那么p和t就从运行中去除，它们的联结就放置在输出缓冲区中。如果p.Id<t.ProfId，那么就丢弃p；否则就丢弃t。然后重复这个过程直到用完所有的输入运行为止。

  组合合并的代价是读取两个关系的排序运行：用于PROFESSOR的运行的200次输入/输出和用于$\sigma_{Semester = 'F1994'}$(TEACHING)运行的250次输入/输出。

- 其他的。然后联结的结果直接通过管道传输给下一个选择（在DeptId上）和投影（在Name上）操作。因为不用把中间结果写到磁盘上，所以这些阶段的输入/输出代价为零。
- 总的代价。将以上各个操作的代价求和，我们得到：400+252+250+200+250＝1352。

最佳的计划

总结一下结果，我们可以看到（在考虑到的那些计划中，这些计划只是所有可能的计划的一小部分）最好的计划是计划C。这里，有趣的是，这个计划优于计划b，即使计划b联结的是较小的关系（因为选择充分下推了）。其中的原因是，当选择下推到TEACHING关系的时候，索引丢失。这又一次说明了14.2节的启发式规则就是启发性的。尽管利用它们可以产生更好的查询执行计划，但是它们必须在一个更为普遍的代价模型中计算。

14.4 估计输出的大小

在前一节中的例子，说明了对各种关系表达式的输出大小进行精确估计的重要性。一个表达式的结果作为另一个表达式的输入，输入的大小对于计算的代价有着直接的影响。为了更好地理解这样的估计是如何完成的，我们给出一个简单的技术，这个技术基于以下的假设：所有的值在关系中出现的几率是相等的。

对每个关系名**R**来说，系统目录可以包含下面一组统计信息：

- Blocks（**R**）　表**R**的实例占用的块的数目。
- Tuples（**R**）　**R**的实例中元组的数目。
- Values（**R**.*A*）　**R**的实例中属性*A*的不同值的数目。
- MaxVal（**R**.*A*）　**R**的实例中属性*A*的最大值。
- MinVal（**R**.*A*）　**R**的实例中属性*A*的最小值。

之前我们介绍了属性权重的概念，并使用它来估计选择和等值联结的大小。我们现在定义更为普遍的概念——减小因子。考虑下面的通用查询：

```
SELECT   TargetList
FROM     R1 V1, ···, Rn Vn
WHERE    Condition
```

这个查询的**减小因子**（reduction factor）是比率

$$\frac{\text{Tuples（结果集）}}{\text{Tuples}(R_1) \times \cdots \times \text{Tuples}}$$

尽管这个定义看起来是循环的（为了找出减小因子我们需要知道结果集的大小），但是减小因子容易通过归纳查询结构来估计。

我们假设减小因子因查询的不同部分而彼此独立的。因而，

$$reduction\ (Query) = reduction\ (TargetList) \times reduction\ (Condition)$$

这里，*reduction*（*TargetList*）是因为各行在SELECT子句中的属性上的投影而导致的大小减小，*reduction*（*Condition*）是因为去掉不满足*Condition*的行而导致的大小减小。

如果$Condition = Condition_1$ AND $Condition_2$，那么

$$reduction\ (Condition) = reduction\ (Condition_1) \times reduction\ (Condition_2)$$

因而，因为复杂条件而导致的大小减小可以根据因为那个条件的分量而导致的大小减小来估计。

剩下的是估计因为SELECT子句的投影和WHERE子句中的原子条件而导致的减小因子。我们在这个讨论中忽略嵌套子查询和聚合。

- $reduction\ (\mathbf{R}_i.A = value) = \dfrac{1}{Values\ (\mathbf{R}_i.A)}$，这里$\mathbf{R}_i$是关系名，*A*是$\mathbf{R}_i$中的属性。这个估计是基于均匀假设——所有的值是等值可能的。
- $reduction\ (\mathbf{R}_i.A = \mathrm{R}_j.B) = \dfrac{1}{\max\ (Values\ (\mathbf{R}_i.A), Values\ (\mathbf{R}_j.A))}$，这里$\mathbf{R}_i$和$\mathbf{R}_j$是关系，*A*和*B*

是属性。使用均匀假设，我们可以把$\mathbf{R}_i$ ($\mathbf{R}_j$)分解成若干子集，这些子集具有以下性质：子集的所有元素在$\mathbf{R}_i.A$（$\mathbf{R}_j.B$）上有着相同的值。如果我们假设$\mathbf{R}_i$中有$N_{\mathbf{R}_i}$个元组，$\mathbf{R}_j$中有$N_{\mathbf{R}_j}$个元组，$\mathbf{R}_i$中的每个元素匹配$\mathbf{R}_j$中的一个元素，那么我们可以得出结论：满足条件的元组的数目是$Values\ (\mathbf{R}_i.A) \times (N_{\mathbf{R}_i} / Values\ (\mathbf{R}_i.A)) \times (N_{\mathbf{R}_j} / Values\ (\mathbf{R}_j.B))$。通常，计算减小因子是假设（并不实际）在较小范围内的每个值匹配较大范围内的一个值。假设$\mathbf{R}A$是较小的范围，用$N_{\mathbf{R}_i} \times N_{\mathbf{R}_j}$来除这个表达式，就得到减小因子。

- $reduction\ (\mathbf{R}_i.A > value) = \dfrac{MaxVal\ (\mathbf{R}_i.A) - value}{Values\ (\mathbf{R}_i.A)}$。对于$\mathbf{R}_i.A < value$来说，减小因子也可以类似定义。这些估计也是基于这样假设：所有的值都是均匀分布的。
- $reduction\ (TargetList) = \dfrac{\text{number} - \text{of} - \text{attributes}\ (TargetList)}{\text{number} - \text{of} - \text{attributes}\ (\mathbf{R}_i)}$。这里为了简单起见，我们假设所有的属性对应相等的元组大小。

14.3节的属性$\mathbf{R}_i.A$的权重现在可以用减小因子的概念来估计：

$$weight\ (\mathbf{R}_i.A) = Tuples\ (\mathbf{R}_i) \times reduction\ (\mathbf{R}_i.A = value)$$

例如，查询PROFESSOR.DeptId = *value*的减小因子是1/50，因为有50个系。因为PROFESSOR中元组的数目是1000，所以PROFESSOR中属性DeptId的权重是20。

14.5 选择计划

在前一节中，我们看了一些查询执行计划，并且说明如何计算它们的代价。然而，我们还没有给出产生候选计划的通用技术。但是，可能的计划的数目会非常大，所以我们需要一种有效的方法来选择相对较小的但是比较符合要求的子集。我们可以计算每一个计划的代价，然后选择最好的。在此过程中至少涉及三个主要的问题。

- 选择逻辑计划。
- 减小搜索空间。
- 选择启发式的搜索算法。

我们依次来讨论这些问题。

1. 选择逻辑计划

我们把查询执行计划定义为查询树，这棵树的每个内部结点都附加有关系实现方法。因此，构造这样的计划涉及两个任务：选择树和选择现实方法。选择正确的树是更为困难的工作，因为涉及的树的数目是由二元可分配和可交换的操作符（如联结、笛卡儿积、并等等）决定的，可以用很多不同的方式来处理。我们在14.2节中提到的其中N个关系由这样的操作符组合的查询树的子树可以有T（N）× N! 种方法。我们想单独处理这种指数级的复杂性，所以我们首先关注**逻辑查询执行计划**（logical query execution plan），它通过把同一种的连续二元操作符组合到一个节点，避免了这个问题，如图14-4所示。

不同的逻辑查询执行计划是通过下推选择和投影以及将选择和笛卡儿积组合成联结而利用式（14.2）的“主计划”创建的。所有可能的逻辑计划中只有一部分用来做进一步的考虑。通常，被选中的计划是充分推动树（因为希望它们产生最小的中间结果）和所有“几乎”是

充分推动的树。从前一节中的讨论可以很明显地看出后一种情况的原因：下推选择和投影到查询树的叶子节点可以消除在联结计算中使用索引的选择。

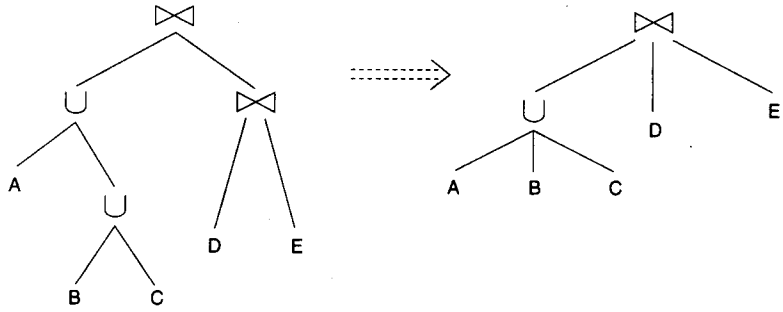


图14-4　把查询树转换成逻辑查询执行计划

根据这个启发式方法，图14-2a的查询树就不会被选中，因为其中没有推动任何东西。14.3节说明了图14-3b中的充分下推查询计划不如图14-3c中的几乎充分下推计划（它有着最小的代价估计）。在这个例子中，所有的连接都是二元的，所以图14-4所示的转换并不适合。

2. 减小搜索空间

选择了候选的逻辑查询执行计划后，查询优化器必须决定如何来计算涉及可交换和可分配的操作符的表达式了。例如，图14-5说明了几种可选的但是等价的把组合多个关系的逻辑计划中可交换和可分配的节点a转换成查询树b、c和d的方法。

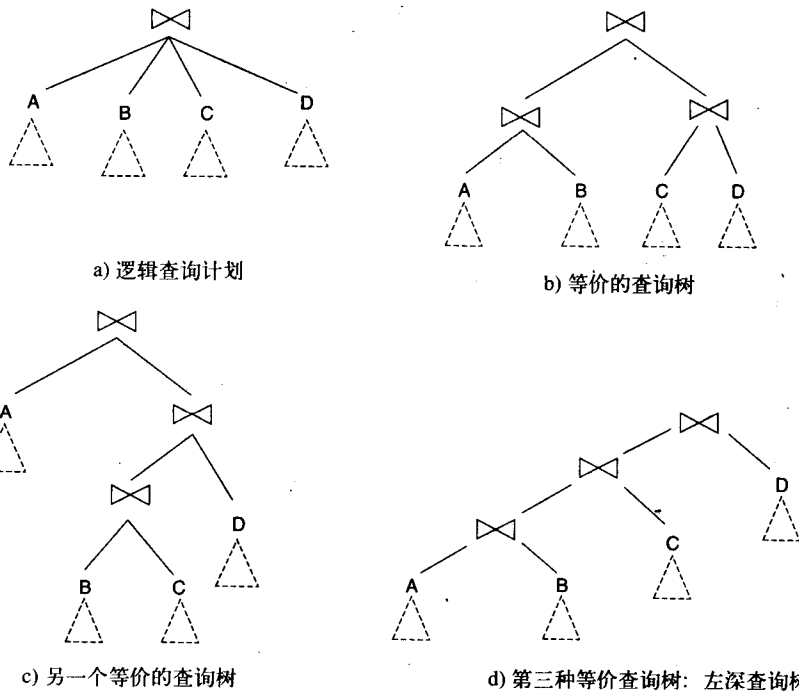


图14-5　逻辑计划和三种等价查询树

对应逻辑查询计划中一个节点的所有可能的等价查询（子）树空间是二维的。首先，

我们必须选择想要的树的形状（即我们忽略节点的标号）。例如，图14-5的树有不同的形状，其中d是最简单的。这样形状的树被称为**左深查询树**（left-deep query tree）。树的形状对应涉及可分配和可交换的操作符的关系子表达式的加括号的特定方法。因而，图14-5a中的逻辑查询执行计划对应于表达式A⋈B⋈C⋈D，而查询树b、c和d分别对应于表达式（A⋈B）⋈（C⋈D），A⋈（B⋈C）⋈D和（（A⋈B）⋈C）⋈D。左深查询树总是对应形如（…（（E_{i_1}⋈ E_{i_2}）⋈ E_{i_3}）⋈…）⋈的代数表达式。

查询优化器通常为查询树决定一个特定的形状：左深。这是因为即使有了固定的树形，查询优化器还是要做大量的工作。给定图14-5d的左深树，仍然有4！个可能的排序联结的方法。例如，（（B⋈D）⋈C）⋈A是图14-5d另一种联结的排序，它会产生不同的左深查询执行计划。所以，如果为4！个查询执行计划计算代价估计，那么看起来不会花费很多时间，但是想一想，如果要估计10！或者12！甚至16！个计划的代价，那要花费多少时间。顺便提一下，所有商用查询优化器最多处理16个左右的联结。

除了减小搜索空间常见的需要以外，选择左深树而不选形如图14-5b的树的另一个原因是管道传输。例如，在图14-5d中我们可以首先计算A⋈B，然后把结果通过管道传输给和C的下一个联结。第二个联结的结果也可以在树中向上进行管道传输，而不需要将中间结果物化到磁盘。能够这样做对于大关系来说是很重要的，因为从一个联结得到的中间输出可能会很大。例如，如果关系A、B和C的大小是1000页，那么中间的关系可能会达到10^9个页，只是在与D连接后。在磁盘中存储这样的中间结果的开销是巨大的。

3. *启发式的搜索算法*

选择左深树已经把搜索空间的大小由极大减小到很大。下一步我们必须给左深树的叶子节点分配关系。因为有N！种分配方法，所以估计所有的代价仍然是很难处理的事情。因此，需要利用启发式的搜索算法，通过查看整个搜索空间中极小的一部分来找出合理的计划。一个这样的算法是基于*动态编程*，并应用于多个商用系统中（如DB2）。我们下面要解释这个算法主要的思想。[Griffiths-Selinger et al. 1979]给出了具体内容。[Wong and Youssefi 1976]描述了另一种不同的启发式搜索算法，这一算法应用于Igres（过去很有影响力的一个数据库管理系统）。

动态编程启发式搜索算法的一个简化版本如图14-6所示。它首先通过评估用于计算N元联结E_1⋈…⋈E_N（每个E_j都是1元关系表达式）的每个参数的所有计划的代价，从而创建一个左深查询树。这些就被称为*1元关系计划*。注意，每个E_j都可能有若干这样的计划（因为有不同的存取路径，如一个人可能使用一遍扫描，另一个人可能使用索引），所以1元关系计划的数目可能是N或者更大。所有这些计划中*最好的*（即那些具有最小代价的）可以扩展到二元关系计划，然后三元关系计划等等。为了扩展最好的一元关系计划p（为了明确起见，假定p是E_{i_1}的一个计划）为二元关系计划，p和除了E_{i_1}的计划之外（因为我们已经选择p作为E_{i_1}的计划）的每个一元关系计划做联结。然后我们计算所有这些计划的代价，保留最好的二元计划。通过把q和每个一元关系计划（除了用于E_{i_1}和E_{i_2}的计划，因为它们已经在q中考虑进去了）作联结而扩展每个最好的二元计划q为一组三元关系计划。再强调一次，为下一个阶段只保留具有最低代价的计划。这个过程继续下去，直到与逻辑计划E_1⋈…⋈E_N对应的左深表达式完全构造出来为止。

Input: *A logical plan* $E_1 \bowtie \cdots \bowtie E_N$
Output: *A "good" left-deep plan* $(\ldots((E_{i_1} \bowtie E_{i_2}) \bowtie E_{i_3}) \bowtie \ldots) \bowtie E_{i_N}$

```
1-Plans := all 1-relation plans
Best := all 1-relation plans with lowest cost
for (i := 1; i < N; i++) do
    // Below, ⋈(meth) denotes join marked with an implementation method, meth*
    Plans := { best ⋈(meth) 1-plan | best ∈ Best; 1-plan ∈ 1-Plans, where
                                        1-plan is a plan for some E_j that has not
                                        been used so far in best }
    Best := { plan | plan ∈ Plans, where plan has the lowest cost }
end
return Best;
```

图14-6 查询执行计划的启发式查找

我们用实例查询（14.7）来说明整个过程。首先查询处理器生成许多合理的逻辑计划，在我们的情况下，这些计划是图14-2b的充分下推的树和两棵部分下推树c和d。

图14-2所示的树的形状是左深，但是每棵这样的树有两个查询执行计划与之对应：它们的差别在于联结中关系的顺序不同。我们从图14-2c的逻辑计划开始，来考虑用图14-6的算法生成的查询执行计划。

对于$\sigma_{DeptId = 'CS'}$(PROFESSOR)的一元关系计划可以使用下面的存取路径：一遍扫描，PROFESSOR.DeptId上的聚簇索引，或者二元搜索（因为PROFESSOR在DeptId上排序）。最好的计划使用这个索引，所以保留它。对于表达式TEACHING只需扫描。我们现在有了两个一元关系计划。在下一个迭代中，算法扩展选中的一元计划到二元计划。这等于生成两个表达式$\sigma_{DeptId = 'CS'}$(PROFESSOR)$\bowtie_{Id=ProfId}$ TEACHING和TEACHING $\bowtie_{ProfId = Id}$ $\sigma_{DeptId = 'CS'}$(PROFESSOR)，并决定在每个联结中使用的计算策略。我们在14.3节估计了前一个表达式不同的计划，并得出结论：索引嵌套的循环联结是最好的。不能用相同的方法计算第二个表达式，因为参数的顺序指出，要首先扫描关系TEACHING。可以使用排序-合并或块嵌套来计算这个表达式。这两种方法的开销都比较大，所以要丢弃它们。

一旦选择了计算联结的最好的计划，那么我们可以把联结的结果看成一元表达式E，我们现在要为π_{Name}（$\sigma_{Semester = 'F1994'}$（$E$））找出一个计划。因为E的结果不需要排序和索引，又因为不需要消除重复，所以我们选择顺序扫描存取路径来计算两个选择和投影。另外，因为E在内存中生成结果，所以我们选择管道传输来避免把中间结果保存到磁盘上。

利用动态编程算法，可能会错过一些很好的计划，因为它关注在当前时刻什么是最好的，而没有尝试往前再看一看。可以改进一下，不仅保留最好的计划，还保留某些“有趣的”计划。如果一个计划的输出关系是排序的，或者如果它有一个索引，那么即使这个计划的代价不是最小的，这个计划也可以被看作是有趣的。这个启发式方法说明一个事实：排序的关系可以在很大程度上减小随后的操作的代价，比如排序合并联结、消除重复和重组的代价。类似的，索引的关系可以减小随后的联结的代价。

14.6 调整问题：对查询设计的影响

数据库管理之所以成为高薪工作是有原因的。它要求的一个重要技能就是深入了解查询

优化器的工作原理，并能用这一知识去调整数据库设计和查询以提高性能。

1. 检查计划

在13.7节中，我们讨论了了解关系操作符的实现是如何在物理等级上影响设计决定的。其中的讨论注重涉及单个操作符的查询，但是通常的数据库工作负载都包含更为复杂的查询。因此，为了决定用额外的索引可以加快哪个操作的速度，需要了解数据库管理系统为每个查询选择的查询执行计划。在很多情况下，这要求对典型的关系大小和联结、选择属性中值的分布作人工的估计。显然，不可能用这种方法来分析查询执行计划的数目，所以这个过程需要很多直觉和经验。

数据库管理系统厂商常常提供各种工具来帮助进行调整。使用这些工具通常需要创建模型数据库，以在其中试验不同的计划。在大多数数据库管理系统中，典型工具是EXPLAIN PLAN语句，它使得用户可以看到数据库管理系统生成的查询计划。这个语句不是SQL标准的一部分，所以各厂商的语法是有差异的。基本的思想是首先执行下面形式的语句:

```
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'
```

它让数据库管理系统生成一个查询执行计划，并把它存储到称为PLAN_TABLE的关系的一组元组中去。queryno是该表的一个属性。一些数据库管理系统使用不同的属性名，比如Id。可以查询PLAN_TABLE来检索这个计划:

```
SELECT * FROM PLAN_TABLE WHERE queryno=123
```

检查查询计划的基于文本的工具是相当强大的，但是最近它们却很少被使用。一个忙碌的数据库管理员只把基于文本的工具作为最后的手段，因为很多厂商为他们的调整工具提供了华丽的图形界面。例如，IBM的DB/2有Visual Explain，Oracle有Oracle Diagnostics Pack，微软的SQL Server有Query Analyzer。这些工具不仅给出了查询计划，而且也给出了可以加速各种查询的索引。

2. 只有索引的查询

确定了支持各种查询需要的可能的索引之后，数据库设计者必须决定将把哪个索引放进实际的系统中。例如，同一关系的两个不同的查询可能需要两个不同的聚簇索引。因为每个关系只有一个索引可以是聚簇的，所以必须要有解决的办法。在这种情况下，需要对查询的“重要性”进行排序，可以考虑这个查询发生的频繁程度，希望的响应时间，等等。

另外一种避免聚簇冲突的方法是强制优化器使用**只有索引的策略**（index-only strategy）。假设TEACHING已经在Semester上有聚簇B^+树索引，但是另一个重要的查询需要在ProfId上的聚簇索引以快速访问与给定教授相关的课程代码。因为数据库管理系统不能在TEACHING上再创建另外一个聚簇索引，所以我们可以使用属性序列ProfId、CrsCode在TEACHING上创建非聚簇B^+树索引来避免这个问题。因为我们的查询只需要ProfId和CrsCode，所以优化器并不需要去查看TEACHING关系——所有的信息都可以从索引中获得！因为索引是B^+树，所以CrsCode的值在ProfId的相应值上是聚簇的，这就产生了和聚簇索引相同的效果（实际上，它更为有效，因为索引更小，并且不需要检索实际关系的页面）。

只有索引的查询处理可能看起来是最终的数据库调整设备。然而，很少有免费的午餐：维护额外的索引意味着存储的开销。更重要的是，额外的索引可能严重地降低更新事务的性能，因为每当改变对应的关系，就必须更新每个索引。因此，添加索引对于高度动态的关系来说不是一个好的方法。

3. *嵌套查询和查询优化*

嵌套查询是SQL最强大的特性之一，也是最难优化的一部分。考虑下面的查询：

```
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='F1995' AND T.ProfId = P.Id
```

这个查询返回一组行，这些行的第一个属性值是在1995年秋季教过一门课的计算机科学系教授的名字，第二个属性的值是这门课程的名字。

通常，查询处理器把这个查询分成两个部分，并将内层查询作为一个独立的优化的单位，同时也对外层查询进行独立优化（把内层SELECT语句的结果集视为数据库关系）。因为进行了这样的分解，所以查询优化器可能不再考虑某些优化——例如，不考虑TEACHING上的聚簇索引和搜索键（ProfId, CrsCode），因为相关嵌套查询只是为提供的P.Id的每个值产生一组课程代码。另一方面，可以看出，上面的查询等价于

```
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
```

并且在优化这个查询的时候考虑使用聚簇索引。

值得注意的是，一些查询优化器实际上努力消除嵌套子查询，并采取其他的步骤来减少处理它们的代价。然而，任何时候都避免查询嵌套是个很好的方法。

4. *存储过程*

存储过程的优势在第10章讨论过。它们的重要性质是它们是由数据库管理系统编译的，为它们嵌入式的SQL语句产生了查询执行计划。这一点很好，因为不需要在运行时执行查询优化器。然而，要认识到，在编译时产生的查询执行计划表示优化器基于那时可以得到的统计信息和物理组织做出的最好的猜测。这意味着如果之后加入、删除索引或者相关统计信息发生大的改变时，之前生成的查询计划可能就不再是好的了，甚至是不可行的。因此，不应该对涉及非常动态的关系的查询使用存储过程，并且最好在索引变化的时候重新编译这些过程。

14.7 参考书目

在[Garcia-Molina et al. 2000]中可以找到针对查询优化的详细的讨论。在[Griffiths-Selinger et al. 1979, Wong and Youssefi 1976]中描述了启发式搜索算法。

如果想进一步阅读关于最新的查询优化技术和其他的参考，请看[Ioannidis 1996, Chaudhuri 1998]。

14.8 练习

14.1 对于投影操作可能有可交换的转换吗？请解释原因。

14.2 写出把$\pi_A((R \bowtie_{B=C} S) \bowtie_{D=E} T)$转换成$\pi_A((\pi_E(T) \bowtie_{E=D} \pi_{ACD}(S)) \bowtie_{C=B} R)$所需要的一系列步骤。为了让上面的转换正确进行，请列出模式R、S和T都必须要有的属性以及这些模式（或者某些）不可以有的属性。

14.3 使用14.2节给出的启发式规则，在什么条件下可以把表达式$\pi_A((R \bowtie_{cond_1} S) \bowtie_{cond_2} T)$转换成$\pi_A(\pi_B(R \bowtie_{cond_1} \pi_C(S)) \bowtie_{cond_2} \pi_D(T))$?

14.4 考虑用在14.3节的实例中的联结$\text{PROFESSOR} \bowtie_{\text{ID=PROFID}} \text{TEACHING}$。我们稍微改变一下统计数字，假设TEACHING.ProfId的不同值的数目是10 000（它翻译成这个属性较低的权值）。

a. PROFESSOR关系的基数是多少?

b. 假定在ProfId上有非聚簇的散列索引，并且假设就像以前一样，5个PROFESSOR元组可以放进一个页，10个TEACHING元组可以放进一个页，TEACHING的基数是10 000。估计在有51页的缓冲区的情况下使用索引嵌套循环和块嵌套循环计算上面联结的代价。

14.5 考虑下面的查询

```
SELECT DISTINCT E.Ename
FROM    EMPLOYEE E
WHERE   E.Title = 'Programmer' AND E.Dept = 'Production'
```

假设

- 10%的雇员是程序员。
- 5%的雇员是为生产部门工作的程序员。
- 有10个部门。
- EMPLOYEE关系有1000个页，每页有10个元组。
- 有51页的缓冲区可以用来处理这个查询。

为下面的每个情况找出最好的查询执行计划:

a. 在Title上有唯一的索引，并且是聚簇的2级B^+树。

b. 在属性序列Dept、Title、Ename上有唯一的索引，并且是聚簇的，它有2级。

c. 在Dept、Ename、Title上有唯一的索引，并且是聚簇的3级B^+树。

d. 在Dept上有非聚簇的散列索引，在Ename上有2级聚簇树索引。

14.6 考虑下面的模式，其中键属性加了下划线:

EMPLOYEE(<u>SSN</u>, Name,Dept)
PROJECT(<u>SSN,PID</u>,Name,Budget)

PROJECT中的SSN属性是为这个项目工作的雇员的Id。每个项目中有若干雇员，但是函数依赖Pid→Name, Budget是成立的（所以关系不是规范化的）。考虑查询

```
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
```

假设有下面的统计信息:

- 在EMPLOYEE关系中有10 000个元组。
- 在PROJECT关系中有20 000个元组。
- 在每个关系中每页有40个元组。
- 有10页的缓冲区。
- E.Name有1000个不同的值。
- Budget域包含了1 ~ 100的整数。
- 索引:
  - EMPLOYEE关系
    在Name上：非聚簇，散列。
    在SSN上：聚簇，3级B+树。
  - PROJECT关系:
    在SSN上：非聚簇，散列。
    在Budget上：聚簇，2级B+树。

a. 画出充分下推的查询树。

b. 找出“最好的”执行计划和次好的计划。每个计划的代价是多少？解释你是如何得到答案的。

14.7 考虑下面的模式，其中键属性加了下划线（不同的键加的下划线不一样）:

PROFESSOR(Id, Name, Department)
COURSE(CrsCode, Department, CrsName)
TEACHING(ProfId, CrsCode, Semester)

考虑下面的查询:

```
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
```

假设有下面的统计信息:

- 在PROFESSOR关系中有1000个元组，每页上有10个元组。
- 在TEACHING关系中有20000个元组，每页上有10个元组。
- 在COURSE关系中有2000个元组，每页上有5个元组。
- 有5页的缓冲区。
- 对于Department有50个不同的值。
- 对于Semester有200个不同的值。
- 索引:
  - PROFESSOR关系:
    在Department上：聚簇，2级B+树。
    在Id上：非聚簇，散列。
  - COURSES关系:
    在CrsCode上：已排序（没有索引）。
    在CrsName上：散列，非聚簇。
  - TEACHING关系:
    在ProfId：聚簇，2级B+树。
    在Semester、CrsCode上：非聚簇，2级B+树。

a. 首先，给出与上面的SQL查询对应的没有优化过的关系代数表达式，然后画出相应的完全下推的查询树。

b. 找出最好的执行计划和它的代价，并解释你是如何得到答案的。

14.8 考虑下面的关系，它表示了一个房地产数据库的一部分：

AGENT(Id, AgentName)
HOUSE(Address, OwnerId, AgentId)
AMENITY(Address, Feature)

AGENT关系保存了房地产代理的信息，HOUSE关系具有谁卖房子和相关的代理的信息，AMENITY关系给出了每所房子的特性的信息。每个关系的键属性都加了下划线。

考虑下面的查询：

```
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'
```

假设这个查询可用的缓冲区空间有5页，而且有下面的统计信息和索引：

- AMENITY:

  在1000个房子上有10 000条记录，每页有5条记录。

  在Address上有聚簇的2级B^+树索引。

  在Feature上有非聚簇的散列索引，50个特性。

- AGENT:

  有200个代理，每页有10个元组。

  在Id上有非聚簇的散列索引。

- HOUSE:

  有1000个房子，每页上有4条记录。

  在AgentId上有非聚簇散列索引。

  在Address上有聚簇的2级B^+树索引。

回答下面的问题（并解释你是如何得到答案的）

a. 画出与上面的查询对应的完全下推的查询树。

b. 找出最好的查询计划来计算上面的查询，并估计它的代价。

c. 找出次优的计划，并估计它的代价。

14.9 图14-3中的查询执行计划对于查询14.3 ~ 14.6都没有消除重复。为了说明这一点，我们增加一个额外的关系操作符δ，它表示消除重复的操作。通过在适当的地方增加δ来修改图14-3中的计划，从而使计算的代价最小。请估计每个新的计划的代价。

14.10 选择一个数据库管理系统来为练习14.5中的情景创建一个数据库。使用EXPLAIN PLAN语句（或者由你的数据库管理系统提供的等价的语句）来比较你手工找到的最优计划和由数据库管理系统实际产生的计划。

14.11 仿照练习14.10，但是使用练习14.16中的情景。

14.12 仿照练习14.10，但是使用练习14.17中的情景。

第15章 事务处理概述

事务处理应用程序中的事务应该满足第2章中讨论的ACID性质——原子性、一致性、隔离性和持久性。事务的设计者负责保证系统中事务的一致性。我们必须保证，如果每个事务独立执行（没有其他的事务并发执行），那么它能正确运行，即它会维护数据库的完整性约束并执行其规格说明中的操作（例如，学生注册系统中的注册事务正确更新学生和课程信息）。其余的性质（原子性、隔离性和持久性）是底层的事务系统的责任。我们将会在本章中概述如何实现这些特性。

本书的第四部分将给出这些问题的更为详细的解释。如果你想研究那一部分，你可以跳过这一章。特别要注意的是，本章是数据库简介部分的最后一章，而不是事务处理部分的第一章。

15.1 隔离性

事务设计者负责设计每个事务，所以如果每个事务独立执行，并且最初的数据库正确地对现实企业的当前状态进行了建模，那么事务就可以正确执行，最终的数据库也就可以正确地对企业的（新）状态进行建模。然而，如果事务处理系统并发地执行一组这样的事务（以某种交叉执行的方式），那么后果可能就是数据库无法转换到和现实世界企业相对应的状态，或者给用户返回错误的结果。

图2-4的调度是错误的并发调度的例子。该调度成功地完成了两个注册事务，但是该课程超员了，而注册学生的总数目只增加了一个。失败的原因是交叉执行。两个事务都读取了cur_reg相同的值，所以它们都没有考虑到对方的影响。我们把这种情况称为缺少隔离性（即ACID中的I）。

系统实现隔离性的一种方法是以某种串行的顺序一个接着一个地执行事务——每个事务只有在前一个事务完成之后才可以开始。假设独立执行的事务都可以正确完成，那么串行调度将是正确的。假设数据库在调度开始的时候可以对现实世界进行正确地建模，那么它在每个事务完成后也将会对现实世界进行正确地建模。

但是，对于很多应用程序来说，串行执行将导致无法容忍的很小的事务吞吐量（用每秒的事务数来衡量）和用户无法接受的很长的响应时间。尽管限制事务处理系统只执行串行调度是不实际的，但是串行调度是很重要的，因为它们可以作为衡量正确性的主要标准。如果非串行调度具有和串行调度相同的结果，那么就可以保证它也是正确的。

注意，这里的推理只能朝一个方向进行。和串行调度具有不同效果的非串行调度不一定就是错误的。我们将会看到大多数数据库管理系统允许应用程序设计者灵活执行非串行调度。然而，我们首先讨论可串行化调度：和串行调度等价的调度。

15.1.1 可串行化

提高性能以优于串行执行并且实现隔离性的一种方法是保证调度是可串行化的。*可串行化调度*（serializable schedule）是等价于串行调度的调度。下面我们将讨论等价的含义。

先举一个简单的例子，假设事务T_1只对数据库的数据项x和y进行读和写，事务T_2只对数据库的数据项z进行读和写。即使事务是交叉执行的，如调度

$$r_1(x)\ w_2(z)\ w_1(y)$$

那么T_2的访问对T_1没有影响，T_1的访问对T_2也没有影响。因此，调度的总体结果和按顺序$T_1\ T_2$或$T_2\ T_1$串行执行事务的结果是一样的，即下面的串行调度之一：

$$r_1(x)\ w_1(y)\ w_2(z)$$

或

$$w_2(z)\ r_1(x)\ w_1(y)$$

注意，这两个等价的串行调度都是由最初的调度通过互换操作的位置得来的。在第一个例子中，互换了两个写操作。它们可以互换是因为它们在不同的数据项上操作，因此不管它们以何种顺序执行，都将使数据库具有相同的最终状态。在第二个例子中，我们互换了$r_1(x)$和$w_2(z)$。这些操作也可以互换是因为它们在不同的数据项上操作，因此T_1在两个顺序中都得到了相同的x值，数据库具有相同的最终状态。

假设除此之外，T_1和T_2还读取一个共同的数据项q。总体结果还是和以上述两种顺序之一串行执行事务的结果是相同的。因此调度

$$r_1(x)\ r_2(q)\ w_2(z)\ r_1(q)\ w_1(y)$$

和串行调度（T_1的所有操作都在T_2的操作之前完成）

$$r_1(x)\ r_1(q)\ w_1(y)\ r_2(q)\ w_2(z)$$

具有相同的结果。这两种调度之间的等价又是基于互换性。这个例子说明的一个新特性是互换不一定要求操作必须访问不同的数据项。在这种情况下，$r_1(q)$和$r_2(q)$可以互换是因为它们在两种执行顺序下都给事务返回相同的值。

图2-4所示的两个注册事务的调度是不可串行化的。我们不能通过一系列相邻操作的互换来得到等价的串行调度，因为在同一个数据项上的读操作和写操作是不可互换的，而且在同一个数据项上的两个写操作也不可以互换。

通常，我们感兴趣的是说明什么时候某组并发执行的事务的调度S等价于（即具有相同的执行结果）那组事务的某个串行调度S_{ser}。通俗地说，我们所要求的就是在两个调度中

- 由相应的读操作返回的值是相同的。
- 以相同的顺序对每个数据项进行更新。

如果S和S_{ser}中每个读操作返回相同的值，那么由事务完成的计算在两个调度中将是相同的，因此事务将把相同的值写回数据库中。因为在两个调度中以相同的顺序进行写操作，它们将使数据库具有相同的最终状态。因而，S和S_{ser}具有相同的结果（因此是等价的）。

如果一组事务的调度如上所述的那样等价于一个串行调度，那么称这个调度是**可串行化**（serializable）调度。因为在串行调度中一次只执行一个事务，事务之间不可能相互影响。因

而，称它们是隔离的。另外，因为串行调度和可串行化调度之间是等价的，所以把可串行化调度中的事务也称为是隔离的。

数据库系统可以保证调度是可串行化的。通过允许可串行化调度，它们允许程度更高的并发性，因此性能得到提高。这些系统也提供了不要求调度为可串行化的这样较为宽松的隔离观点（稍后讨论），因此它们支持程度更高的并发性并得到更好的性能。因为这样的调度不一定等价于串行调度，所以并不能对所有程序都保证正确性。因此，必须慎重使用较为宽松的隔离观点。

负责保证隔离性的事务处理系统部分称为**并发控制**（concurrency control）。并发控制通过控制数据库操作的调度来保证隔离性。当事务想读写数据库数据项时，它把这个请求发送给并发控制。并发控制知道到那一时刻为止已经批准的请求序列，但是它不知道将会有什么请求到达，并发控制会决定如果在那个时刻批准那个请求是否能保证隔离性。如果不能保证隔离性，请求就不能批准，因此事务要么被迫等待，要么异常中止。

15.1.2 两段锁

商用系统中大多数并发控制用**严格的两段锁协议**（strict two-phase locking protocol）[Eswaran et al. 1976]来实现可串行性。这个协议使用与数据库中数据项相关的锁，要求事务在访问数据项之前要拥有适当的锁。当事务想读（或写）数据库的数据项时，它向并发控制发送一个请求，并发控制在把这个请求传递给执行这个访问的数据库系统模块之前必须授予该事务在那个数据项上的**读锁**（read lock）或**写锁**（write lock）。系统依据下面的规则来完成锁的请求、授予和释放：

1) 如果事务T请求读一个数据项，并且没有其他的事务拥有这个数据项上的写锁，那么控制就把那个数据项上的读锁授予T，允许操作继续进行。注意，因为其他事务可能在那时拥有该数据项上的读锁，所以读锁常常也称为**共享**（shared）锁。

2) 如果事务T请求读一个数据项，但是另一个事务T'拥有那个数据项上的写锁，那么T必须等到T'完成（并释放它的锁）为止。我们称这个读请求与之前批准的写请求**冲突**（conflict）。

3) 如果事务T请求写一个数据项，并且没有其他的事务在那个数据项上拥有读锁或写锁，那么控制就把那个数据项上的写锁授予T，允许操作继续进行。因为写锁排斥了**所有**其他的锁，所以也常常称之为**排他**（exclusive）锁。

4) 如果事务T请求写一个数据项，但是另一个事务T'拥有那个数据项上的读锁或写锁，那么T必须等到T'完成（并释放它的锁）为止。我们称这个写请求与之前批准的读请求或写请求**冲突**。

5) 一旦授予事务锁，事务就拥有了这个锁。数据项上的读锁允许事务对那个数据项进行读。数据项上的写锁允许事务对那个数据项进行读或写。当事务完成时，它释放授予它的所有锁。

上述规则规定，如果请求不和之前批准的来自另一个活动事务的请求冲突，就可以批准这个请求。（来自不同事务的）请求如果属于下面的情况之一就不会冲突：

- 它们引用不同的数据项。

• 它们都是读请求。

图15-1以表格的形式说明了数据项之间的冲突关系。× 表示冲突。

请求的模式 \ 已批准的模式	读	写
读		×
写	×	×

图15-1　两段锁并发控制的冲突表。锁模式之间的冲突用X表示

并发控制使用锁来记住当前的活动事务之前执行的数据库操作。事务要在数据项上执行操作，只有当这个操作和当前的活动事务之前在那个数据项上执行的所有其他操作可以互换（不冲突）的条件下，系统才能授予该事务锁来执行相应操作。例如，因为数据项上的两个读操作是可以互换的，所以即使一个事务当前拥有那个数据项上的读锁，也可以授予另一个事务在那个数据项上的读锁。用这种方法，控制保证活动事务的操作可以互换，所以在任何时候这些操作都可以以任何顺序串行化。这个结果是证明任何由并发控制产生的调度都和串行调度等价的基础。

如果每个事务都是先经历获得锁的阶段，然后经历释放锁的阶段，那么锁控制就称为是**两段的**（two-phase）。之所以称它为**严格的**（strict）两段锁控制，是因为每个事务都保持它的锁直到事务完成为止：第二阶段就被压缩成时间上的一个点。在**非严格**（nonstrict）的两段锁控制中，第二阶段在事务得到它需要的所有锁之后就开始了。在第二阶段中，事务可以在任何时候释放锁。

尽管非严格的两段协议保证了可串行性，但是当事务异常中止的时候就出问题了。如果事务T_1修改了数据项x，然后在它的第二阶段释放在其上的锁，第二个事务T_2读了这个新的值随后提交了。因为T_1在完成之前就释放了x上的锁，所以它随后可能会异常中止。原子性要求异常中止的事务不能对数据库状态有任何影响，所以如果T_1异常中止，x要恢复到最初的值。这些事件可以记录为下面的调度：

$$w_1(x)\ r_2(x)\ w_2(y)\ \text{commit}_2\ \text{abort}_1 \tag{15.1}$$

T_2已经对y写入了新值，而这可能就是基于它读到的x值。因为T_2读到的x值是由随后异常中止的事务产生的，所以违反了原子性。因此，即使在非严格的控制中事务也要保持写锁，直到它提交为止。在23.2节中可以找到关于这个问题的更为完整的描述，在23.3和23.4节中可以找到并发控制更为完整的介绍。

使用严格两段锁的并发控制会产生在提交顺序上可串行化的调度。也就是说，调度S等价于串行调度S_{ser}，在S_{ser}中事务的顺序和它们在S中提交的顺序是相同的。为了理解为什么会这样，注意写锁直到提交的时候才释放，所以不允许事务读（或写）尚未提交的事务写过的数据项。因此，每个事务“看到”的是在其完成之前提交的事务序列所产生的数据库。非严格的两段锁协议产生可串行化的调度，但是不必在提交顺序上可串行化。

对于很多应用程序来说，用户希望事务在提交顺序上是可串行化的。例如，你希望在你的银行存款事务提交之后，任何之后提交的事务将看到那笔存款的结果。

在23.1节中可以找到关于可串行性更为完整的讨论。

两段锁协议的原理是拥有锁直到可以安全地释放它们为止。较早地释放锁可能导致不一致的数据库状态或使事务给用户返回错误的结果。数据库领域已经使用专用的术语来描述可能会遇到的一些异常。

- *脏读*　假设事务T_2读数据项x，事务T_1在完成之前对x进行过写操作。如果T_1在它提交之前释放它获得的x上的写锁，那么就可能发生这种情况。因为由读操作返回的x的值不是被已提交的事务所写（我们把这样的值称为提交的），它不可能反映在数据库中。这就是我们所说的**脏读**（dirty read）。例如，T_1可能在T_2读了x之后异常中止，产生调度

$$w_1(x)\ r_2(x)\ abort_1$$

当T_1异常中止，它对x的修改就要回退。因此，T_2读了本不应该在数据库中出现的值。（15.1）中的问题是由脏读引起的。

- *不可重复读*　假设事务T_1读了数据项x，然后它在完成之前释放它获得的读锁。另一个事务T_2可能写了x并提交。如果T_1重新获得x上的读锁并读了它的值，那么第二次读返回的值和第一次读返回的值就不相同。我们把这种情况称为**不可重复读**（nonrepeatable read）。这种情况可以由下面的调度来说明。

$$r_1(x)\ w_2(x)\ commit_2\ r_1(x)$$

注意，在这个例子中，T_2在第二次读之前已经提交，所以第二次读不是脏读。尽管不可重复读可能看起来并不是重要的问题（为什么事务要两次读相同的数据项？），但是它是较为严重的一个问题的征兆。例如，假设x是航班上的一组乘客，y是这组乘客的人数。在下面的调度中，T_2在这个航班上预订了一个座位，因此向x添加了一个条目，并使y加1。T_1读了x和y，并且在添加新乘客之前就看到乘客列表，在增加乘客人数之后看到乘客人数——这就产生了不一致。

$$r_1(x)\ r_2(x)\ w_2(x)\ r_2(y)\ w_2(y)\ commit_2\ r_1(y)$$

这个调度直接与前一个相关。在两种情况下，锁不是两阶段的，T_2覆盖了活动事务读过的数据项。

- *丢失更新*　假设事务读了数据项x的值，根据读到的值，把新值写回x。银行系统的存款事务就是这种活动的实例，其中x是要存款账户的余额。如果这样的事务在获得写锁之前释放它已经获得的读锁，那么在同一个账户上的两个存款事务就可能交叉执行，如下面的调度所示：

$$r_1(x)\ r_2(x)\ w_2(x)\ commit_2\ w_1(x)\ commit_1$$

我们又要考虑T_2在T_1的最后一个操作之前提交的情况。这种情况称为**丢失更新**（lost update）：T_2的结果丢失了，因为T_1写入的值是基于x最初的值而不是T_2写入的值。因此，如果T_1试图存入\$5，T_2试图存入\$10，而最后数据库只增加了\$5。

这些异常和其他未命名的异常一样，都会使事务返回错误的结果，数据库可能会变得不一致。

15.1.3 死锁

假设事务T_1和T_2都想对数据项x和y进行写操作，但是以相反的顺序进行写操作。在一种可能的调度中，T_1封锁了x并进行写操作；T_2封锁了y并进行写操作；T_1请求写y，但是因为T_2封锁了y，所以必须等待；T_2请求写x，但因为T_1封锁了x，所以也必须等待。

$$w_1(x)\ w_2(y)\ Request_w_1(y)\ Request_w_2(x)$$

此时，T_1等待T_2完成，T_2等待T_1完成。两者将会永远等待下去，因为谁也不可能完成。

这种情况称为**死锁**（deadlock）。通常，只要存在等待循环，即一系列事务T_1，T_2，…，T_n，其中每个事务T_i都在等待访问被T_{i+1}封锁的数据项，T_n在等待访问被T_1封锁的数据项，这时就会出现死锁。虽然两段锁特别容易出现死锁，但是死锁可能出现在任何并发控制中，只要它允许事务在请求一个数据项上的锁的时候，还保持另一个数据项上的锁。这样的控制必须有检测死锁，然后异常中止等待循环中一个事务的机制，以使得循环里剩余的事务中至少有一个可以继续进行下去。

利用这样的机制，不管什么时候只要事务被迫等待，控制就会检测是否将会形成等待事务的循环。因此，如果T_1必须等待T_2，控制就会检测T_2是否也在等待，如果是的话，它为什么而等待。这个过程继续下去，就会发现一连串等待的事务，如果这个串循环回到事务T_1，那么就检测到一个死锁。另一个机制是使用**超时**（timeout）。只要事务已经等待了“很长”时间，控制就假设存在死锁，并异常中止这个事务。可以在23.4.2节中找到关于死锁的较为完整的讨论。

即使有了检测和异常中止这些手段，我们仍然不希望有死锁，因为它们浪费了资源（被异常中止事务执行过的运算必须重做），并减慢了系统执行速度。应用程序设计者应该设计表和事务以减少死锁的可能的数目。

15.1.4 关系数据库中的锁

到现在为止，我们关于加锁的讨论都是假设事务请求访问某个命名的数据项（比如x）。加锁在关系数据库中有着不同的形式，在关系数据库中事务访问的是元组。尽管可以对元组加锁，但是事务不是用名称来描述它想访问的元组，而是用元组满足的条件来描述。例如，事务用SELECT语句读到的元组的集合是由WHERE子句中的选择条件来指定的[⊖]。

例如，在银行系统中，ACCOUNTS表可能对于每个独立的账户都用一个元组表示，事务T_1可能读取描述储户Mary控制的账户的所有元组，如下所示：

```
SELECT *
FROM ACCOUNTS A
WHERE A.Name = 'Mary';
```

在这种情况下，T_1读取ACCOUNTS中满足条件（Name属性的值是Mary）的所有元组。条件A.Name = 'Mary'称为**谓词**（predicate）。

至于非关系数据库，我们可以用加锁协议来保证可串行性。在设计这样的协议时，我们必须决定要封锁什么样的数据项。一种办法是封锁整张表，即使只访问其中的一些元组。和元组不同，表在要访问它们的语句中指出名称。因此，SELECT语句可以被视为在这些数据项（即表，在FROM子句中指出名称）上的读。类似的，DELETE、INSERT和UPDATE可以被视为在已命名表上的写。因此可以使用前面几节中描述的并发控制协议，并将产生可串行化调度。问题是表锁是粗粒度的：表可能包含成千上万个元组，因为要访问其中很少的一些元组

⊖ 在事务用主键来存取元组的特殊情况下，我们可以说该键值就是元组的名称，并且可以考虑基于那个名称来封锁元组。然而，即使在这一特殊情况下，其他的考虑也适用于关系数据库（来避免后面讨论的幻影异常）。

而封锁整张表可能会导致并发度的严重损失。

第二种办法是只封锁由SELECT语句返回的元组。例如，在处理上面的SELECT语句时，只封锁描述Mary账户的元组。这种办法的问题是它可能导致称为**幻影**（phantom）的异常。在处理完这个语句之后，但在T_1提交之前，另一个事务可能为Mary创建一个新的账户，插入描述ACCOUNTS中那个账户的新的元组t，然后提交。那个元组称为幻影。要注意的重要一点是T_1拥有的锁并不能阻止这个插入。如果T_1重新执行SELECT，它除了返回之前返回的元组外，还将返回t，因此读是不可重复的（尽管在这种情况下T_1没有释放锁）。至于不可重复读，问题不只只是事务执行了两次相同的读操作。因为可能出现幻影，所以封锁元组并不能保证可串行化调度。通常，在事务T读了表R中满足谓词P的元组之后，另一个事务又在T结束之前向R中插入满足P的元组，这时就会发生幻影。

注意，虽然我们刚描述的简单的元组封锁算法确实可能导致幻影，但是当很多商用数据库管理系统使用术语“元组封锁”（或者是我们稍后将会看到的“页封锁”）来描述并发控制算法时，意味着它们使用元组锁（或者页锁）作为更为复杂封锁算法的一部分，就像在24.3.1节中描述的那样，它确实保证了可串行性。记住下面的警告“当所有方法都失败时，就去读手册吧。”

15.1.5 隔离级别

加锁会阻止并发，因此会影响到性能。所以尽量减少它们的使用。在两段协议中用到的表封锁虽然产生了可串行化调度，但是由于封锁的数据项的大小，它对并发有着极大的影响。元组封锁更为有效，但是即使用在两段协议中也可能（因为幻影）导致不可串行化调度。因为锁一直保持到提交的时候，所以严格协议比非严格协议更限制并发度。因为这些原因，大多数商用数据库管理系统允许应用程序设计者选择封锁协议，例如要封锁哪些数据项，这些锁将要保持多久。我们通常用ANSI标准隔离级别来描述这些选项。

数据库系统经常支持若干隔离级别[Gray et al. 1976]，并且允许应用程序设计者选择适合于特定应用程序的级别。设计者应该选择一个可以保证应用程序正确执行并且能使并发度最大的级别。这些选择可能允许不可串行化的调度。ANSI标准隔离级别是根据每一级别要避免的某种现象（或异常）来指定的。在一个级别中要避免的现象在每个更高的级别中也要避免。这些级别按强度递增的顺序依次是:

- READ UNCOMMITTED　可以有脏读。
- READ COMMITTED　不允许脏读（但是可以有不可重复读）。
- REPEATABLE READ　特定事务执行的对同一个元组连续的读不会产生不同的值（但是可以有幻影）。
- SERIALIZABLE　不允许幻影。事务执行必须是可串行化的。

SQL标准规定同一个应用程序的不同事务可以在不同的隔离级别上执行，每个这样的事务可能看到也可能看不到与这个级别对应的现象。例如，在REPEATABLE READ级别上执行的事务读取特定的元组若干次都将会看到相同的值，即使其他的事务在其他的级别上执行。类似的，在SERIALIZABLE级别上执行的事务看到所有其他事务对数据库做的改变都是串行的，而不管这些事务的级别如何。

尽管某一级别可以通过很多方法来实现，但是封锁是常用的技术。数据库系统独立于隔离级别，它通常可以保证每个SQL语句执行的原子性并且它的执行是和其他语句的执行相隔离的。封锁更是可以控制不同事务的语句交叉执行的方法。

每个级别通过不同的方法来使用锁。对于大多数的级别，事务在执行SQL语句的时候获得它们要存取的数据项上的锁。根据如何存取这个数据项，锁可以是读锁或者写锁。一旦获得了锁，就会将锁保持到提交的时候（在这种情况下称为**长期锁**（long duration）），或者只保持到该语句执行完毕（在这种情况下称为**短期锁**（short duration））。在所有隔离级别上的写锁都是长期锁。读锁在每个级别上的处理方式不同。

- READ UNCOMMITTED　没有获得读锁也可以执行读操作。因此，在这个级别上执行的事务可以读取其他事务已经在其上加过写锁的元组。因此这个事务可能会读取没有提交的（脏的）数据。
- READ COMMITTED　在允许读操作之前，要获得每个元组上的短期锁。因此，将会检测到和写锁的冲突，事务将会被迫等待直到释放写锁为止。因为写锁是长期锁，所以只能读取提交过的数据。然而，当读操作完成之后就释放读锁，所以特定的事务对同一个元组两次连续的读操作可能会被另一个事务（更新这个元组然后提交）的执行所分隔。这意味着读可能是不可重复的。
- REPEATABLE READ　在由SELECT返回的每个元组上获得长期读锁。因此尽管可能存在幻影，但是不可能存在不可重复读。
- SERIALIZABLE　如果在进行所有表的读操作之前都要求获得长期读锁，那么就可以保证它是可串行化调度。尽管这种方法降低了并发度，但是它消除了幻影。在15.1.7节描述了涉及到使用索引的更好的实现。

当事务使用游标来指向元组时会发生一种特别麻烦的不可重复读现象。尽管游标正指向某元组，但在READ COMMITTED级别上并发执行的事务也可以修改那个元组。一些商用数据库系统支持称为CURSOR STABILITY的隔离级别，实质上它就是READ COMMITTED再加上只要有游标指向元组就在元组上维持读锁这一特性。因此，如果不移动游标的话，读那个特定元组是可重复的。

24.1节和24.2节给出了关于关系数据库系统中隔离级别的更为完整的讨论，包含在每个隔离级别上正确的和错误的调度的例子。

SNAPSHOT隔离级别

有一个隔离级别虽然并不是ANSI标准的一部分，但是常用的数据库管理系统Oracle却提供了这一隔离级别，它就是SNAPSHOT隔离级别。SNAPSHOT隔离级别使用**多版本**（multiversion）数据库，即当（提交了的）事务更新一个数据项时，并不丢弃这个数据项的旧值，而是保留这个旧值（或版本），并创建新的版本。因此在任意时刻，数据库中存在同一数据项的多个版本。对于任意的i，系统可能构造每个数据项的值，它包含了要提交的第i个事务和之前提交的所有事务的结果。假设事务是以它们提交的顺序连续索引，图15-2说明了这种情况。这里，数据项的每个版本都用产生它的事务的索引来标记，并用虚线来从整体上指出数据库的版本。因此事务T_4完成时的数据库版本包含了对x的三次更新，对y的两次更新和对z的一次更新。每个数据库版本都称为**快照**（snapshot）。

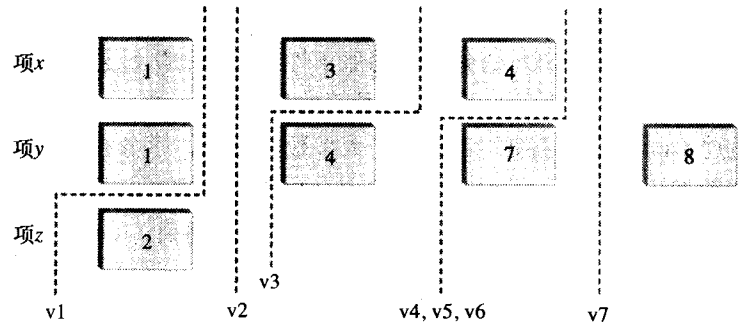


图15-2　多版本数据库

利用SNAPSHOT隔离级别，就不必用读锁了。事务T请求的所有读操作都通过当T第一次请求读操作之时提交的事务序列产生的快照来满足。因为快照一旦产生就无法修改，所以就没有必要使用读锁，读请求也就不需要延迟了。

每个事务的更新是由称为**先提交者赢**（first-committer-wins）的协议来控制的。允许事务T提交需要满足下面两个条件：（1）没有其他的事务在T第一次请求读和它请求提交之间提交；（2）没有其他的事务更新T已经更新过的数据项。否则，T只能异常中止。SNAPSHOT隔离级别的一个实现（在24.5节中描述）有着一个很好的性质：不需要写锁，因此读和写都不需要等待。然而，当事务完成的时候可能会异常中止。

先提交者赢的特性消除了丢失更新，当两个并发的活动事务读了同一个快照又去写同一个数据项的时候，就会发生丢失更新。然而即使有了先提交者赢协议，SNAPSHOT隔离级别也允许不可串行化（因此可能是错误的）调度。例如调度

$$r_1(x)\ r_1(y)\ r_2(x)\ r_2(y)\ w_1(y)\ w_2(x)$$

是允许的，但它不是可串行化的，因为不可能通过一系列的交换相邻操作来产生等价的串行调度。T_1和T_2读取同一个快照，但是它们写了不同的数据项，所以两个操作都可以提交。

SNAPSHOT隔离级别的实现是很复杂的，因为必须要维护多版本数据库。然而，实际上不可能维护所有的版本，最终都要丢弃老的版本。这对于长时间运行的事务来说是一个问题，因为如果它们请求访问不再存在的版本，它们就必须要异常中止。在24.5节可以找到关于多版本并发控制和SNAPSHOT隔离级别的较为完整的讨论。

15.1.6　锁粒度和意向锁

为了减少维护大量锁的开销，很多商用并发控制封锁了比单个数据项要大的单元。例如，并发控制可能不是封锁只占几个字节的数据项，而是封锁该数据项所在的整个磁盘页。因为这样的控制封锁了比实际需要更多的数据项，所以它产生了和只封锁请求加锁的数据项的并发控制相同的或者更高的隔离级别。类似的，并发控制可能不是在包含表中若干元组的页上加锁、而是在整张表上加锁。

封锁单元的大小决定了锁的**粒度**（granularity）。如果封锁单元小，则称为**细**（fine）粒度，否则就是**粗**（coarse）粒度。粒度形成了基于包含关系的层次结构。通常，细粒度锁是元组锁，中等粒度锁是在包含指定元组的页上的锁，粗粒度锁是覆盖表的所有页的表锁。

细粒度锁比粗粒度锁具有更好的并发度，因为事务只封锁它们实际使用的数据项。然而，与细粒度锁相关的开销就比较大。事务通常拥有更多的锁，因为单一的粗粒度锁可能准予访问此事务用到的几个数据项[⊖]。因此，在并发控制中需要更多的空间来保存关于细粒度锁的信息。在为每个单元请求锁时也花费了更多的时间。经过这些权衡之后，数据库系统经常在不同的级别上提供不同的粒度，在同一个应用程序中可以使用不同的级别。

设置多粒度锁引入了一些新的实现问题。假设事务T_1已经取得了一张表的某个特定元组的写锁，事务T_2请求在整张表上的读锁（它想读取这张表上所有的元组）。并发控制不应该授予它表锁，因为不应该允许T_2读取被T_1封锁的元组。问题是并发控制如何检测到这个冲突，因为冲突的锁是在加不同的数据项上的。

解决的方法是有层次地组织锁。在取得细粒度数据项（如一个元组）上的锁之前，事务必须首先取得包含粗粒度数据项（如表）上的锁。但是这种锁是什么类型的呢？显然，不是读锁，也不是写锁，因为在那种情况下获得额外的细粒度锁是没有意义的，有效的锁粒度应该是粗糙的。因此，数据库系统提供了新型的锁——**意向锁**（intention lock）。例如，在支持元组锁和表锁的系统中，在事务获得元组上的读（共享）锁或者写（排他）锁之前，它必须获得包含那个元组的表上的意向锁。意向锁是比读锁和写锁更弱的锁，一般分为三种。

- 如果事务想取得某个元组上的共享锁，它必须首先得到这个表上的**意向共享**锁（Intention Shared，IS）。IS锁指出这个事务想取得那张表中的某个元组上的共享锁。
- 如果事务想取得某个元组上的排他锁，它必须首先得到这个表上的**意向排他**锁（Intention Exclusive，IX）。IX锁指出这个事务想取得那张表中的某个元组上的排他锁。
- 如果事务想更新表中某些元组，但是需要读取所有的元组来决定到底修改哪一些（例如，它想修改某一属性的值小于100的所有元组），它必须首先取得这个表上的**共享意向排他**锁（Shared Intention Exclusive，SIX）。SIX锁是这个表上共享锁和意向排他锁的组合。这允许事务读取这张表上所有的元组，随后得到它想更新的那些元组上的排他锁。

需求的模式 \ 已授权模式	IS	IX	SIX	S	X
IS					×
IX			×	×	×
SIX		×	×	×	×
S		×	×		×
X	×	×	×	×	×

图15-3　意向锁的冲突表。有冲突的锁模式之间用×表示

在图15-3中给出了意向锁的冲突表。例如，它表明在已经授予事务T_1在一张表上的IX锁后，就不能授予另一个事务T_2在那张表上的共享锁。为了理解这一点，注意共享锁允许T_2读取那张表中的所有元组，这和T_1正在更新它们中的一个（或多个）元组相冲突。另一方面，可以授予T_2在不同元组上的排他锁，但是它必须首先获得表上的IX锁。这不会产生问题，因为如图所示IX锁并不冲突。在24.3.1节中将更详细地讨论多粒度锁和意向锁。

15.1.7　用意向锁的可串行化封锁策略

如果事务在两阶段的方式中只使用表锁，那么结果就是可串行化调度。然而，这样的封

⊖ 例如，一个磁盘页上的元组通常都是某张表中的元素，存取一个这样的元组的事务也可能存取另一个这样的元组。页锁允许存取这两个元组。

锁原则会使并发度降低。另一方面，虽然页封锁或者元组封锁更加有效，但是可能产生幻影。很多商用系统使用增强的细粒度封锁策略来防止幻影从而产生可串行化调度。防止幻影的基本条件是在事务T_1读取了表R中满足WHERE谓词P的元组之后，直到T_1终止都没有其他并发执行的事务T_2可以向R中插入满足P的（幻影）元组t。

该策略与T_1如何存取R有关。在构造存取满足P的元组的SQL语句的查询执行计划的时候，系统决定是否可以使用已存在的索引。如果没有这样的索引，系统必须搜索R中的每一页，来定位满足P的元组。在这种情况下，T_1获得R上的S锁并且保持这个锁直到T_1提交为止。T_2不能插入t，因为它首先需要获得R上的（相冲突的）IX锁。因此，当不使用索引的时候，不可能出现幻影。

如果T_1通过索引来存取R，那么就不需要对R进行全部扫描，这种情况就更加棘手了。例如，如果T_1用谓词P来执行SELECT语句，那么它只获得R上的IS锁。假设系统实现了细粒度页锁，那么T_1也必须获得包含它通过索引存取的其中包含满足P的元组的R所在页上的S锁。但是，这些锁并不能防止幻影。如果T_2试图插入幻影t到R中，它可以获得R上的IX锁，因为那个锁并和T_1拥有的IS锁并不冲突。因此在表这个级别上没有冲突。如果t存储在与T_1封锁的页不同的页上，那么在页这个级别上也没有冲突。因此，T_2可能插入t。因而，需要有一些机制来防止这样的幻影。

用到的机制涉及封锁索引结构本身的部分。当插入元组到R中时，要更新用来存取R的索引以包含对新元组的引用。因此，如果T_1在T_2为了插入幻影而必须更新的索引结构的页上保持着共享锁，那么就不能进行插入。特别的，因为t满足P，T_2必须更新的索引页就是T_1必须存取的页，并且T_1在那些页上保持着共享锁。封锁协议的细节依赖于用到的索引的类型，这将在24.3.1讨论。

15.1.8 总结

如何为一个特定应用程序选择隔离级别是比较复杂的问题。商用数据库管理系统厂商常常支持很多可选项（包括但不仅限于上面讨论的那些），并且希望应用程序设计者为他们特定的应用程序选择一个适当的隔离级别。

可串行化调度可以保证应用程序的正确性，但是可能无法满足应用程序的性能要求。然而，一些应用程序在隔离级别SERIALIAZABLE时也可以正确执行。例如，打印储户邮件列表的事务可能不需要最新的数据库视图。因此，它可以使用短期读锁而不使用长期读锁，并且可以在READ COMMITTED上执行。24.2.2节给出了在不同隔离级别上正确运行的应用程序的例子。

15.2 原子性和持久性

原子性要求事务或者成功的完成并提交或者异常中止，即撤销它对数据库做过的任何修改。事务可能被用户异常中止（可能是使用取消按钮），可能被系统异常中止（可能因为死锁或者因为某个数据库更新违反了约束检查），或者是自己异常中止（比如当它发现不想要的数据的时候）。事务不能成功完成的另一个原因是在事务执行的过程中系统崩溃。崩溃比用户或系统引发的异常中止要更复杂一些，因为存储在主存中的所有信息在系统崩溃的时候都会丢

失。因此事务必须异常中止，而且必须只使用存储在大规模存储器中的信息来回退它所做过的修改。

持久性要求在事务提交以后，即使存储数据库的大规模存储器设备失效了，它对数据库的修改也不能丢失。

15.2.1 先写日志

我们在同一节中讨论原子性和持久性，是因为它们经常由同一个机制——先写日志来实现。

日志是一系列描述事务对数据库更新的记录。随着事务的执行，记录就会追加到日志中，而且不会改变。系统会参考日志来实现原子性和持久性。具备了持久性，在存储数据库的大规模存储器设备失效以后，可以用日志来恢复这个数据库。因此，日志通常存储在不同的大规模存储器设备上。日志常常是磁盘上的顺序文件，它通常是双工的（即具有存储在不同设备上的拷贝），所以它在任何一个介质失效以后都还是存在的。

实现原子性和持久性常用的技术涉及到**更新**（update）记录的使用。在事务执行改变数据库状态的操作时，更新记录追加到日志后面。如果操作只是读取数据库，那么不需要追加记录。每个更新记录描述了所做的改变，特别是要包含足够的信息在之后事务异常中止的时候来使系统撤销所做的改变。

有几种方法来撤销改变。最常见的是更新记录中包含要修改的数据库数据项的**前像**（before image），即数据项在修改之前的物理拷贝。如果事务异常中止，就可以用更新记录来恢复这个数据项到它最初的值——这就是有时称更新记录为**撤销记录**（undo record）的原因。除了前像，更新记录还要用**事务识别号**（transaction Id）识别进行改变的事务和改变了的数据库数据项。

如果事务T异常中止，用日志来回退是很方便的。从后向前扫描日志，当遇到T的更新记录时，就把前像写到数据库中，以便撤销这个改变。因为日志可能非常长，所以从后向前扫描到最开始来保证处理T的所有更新记录是不实际的。为了避免完整地从后往前扫描，当T初始化的时候，包含事务识别号的**开始记录**（begin record）就要追加到日志中去。从后向前扫描在遇到这个记录的时候就会停止。

因崩溃而导致的回退比某个事务的异常中止要复杂一点，因为现在系统必须首先识别出异常中止的事务。特别的，系统必须区分已经完成的事务和在发生崩溃时仍然活动的事务。所有活动的事务都必须异常中止。

当事务提交时，它把**提交记录**（commit record）写到日志中。如果它异常中止，那么就回退它的更新，然后把**异常中止记录**（abort record）写到日志中。使用这些记录，从后向前的扫描就可以记录下在崩溃前完成的事务的识别号，因此在后面遇到它们的更新记录时就可以忽略掉。如果在从后向前的扫描中与T相关的第一个记录是更新记录，那么当崩溃发生的时候T是活动的，因此必须异常中止。

注意，当T提交时把提交记录写进日志是很重要的。如果我们假设当T提出写请求的时候数据库是立即更新的（并不总是这种情况），那么当T请求提交的时候，所有T请求的数据库修改将会在大规模存储器中记录下来。然而，提交请求本身并不能保证持久性。如果崩溃发生

在事务提出这个请求之后，但是在提交记录写进日志之前，那么事务就会被恢复过程异常中止，这就不是持久的了。因此，实际上事务要等到它的提交记录已经追加到大规模存储器上的日志中后才可以提交。

和崩溃相关的最后一个问题是在恢复时避免做从后向前完整扫描的机制问题。没有这样的机制，恢复过程就无法知道什么时候才能停止扫描，因为在崩溃时仍然活动的事务可能在日志中前面的某点上已经写下了更新记录，然后就没有做进一步更新了。因此恢复过程除非往回扫描到那条记录，否则就找不到这个事务存在的痕迹。

为了处理这种情况，系统周期性地追加**检查点记录**（checkpoint record）到日志中去，在其中列出当前活动的事务。恢复过程至少必须从后向前扫描到最近的检查点记录。如果T在那条记录中命名，并且从后向前扫描没有遇到T的提交记录或异常中止记录，那么T在系统崩溃的时候仍然是活动的。从后向前扫描必须在经过检查点之后还继续进行，直到遇到T的开始记录为止。扫描在考虑了所有这样的事务之后终止。

我们已经假设在数据库数据项x的新值写进包含数据库的大规模存储器设备的时候，x的更新记录也写进包含日志的大规模存储器设备。实际上，x的更新和更新记录的追加是以某种顺序进行的。考虑在执行这些操作的时候发生崩溃的可能性。如果没有一个操作完成，那么没有问题。更新记录不会出现在日志中，恢复过程也不会撤销任何东西，因为x还没有被更新。如果崩溃发生在两个操作都完成之后，那么恢复过程可以用如上所述的方法正确进行恢复。然而，假设首先更新x，然后在更新和日志追加之间崩溃发生。那么恢复过程就无法回退这个事务并把数据库恢复到一致的状态，因为大规模存储器上没有x最初的值——这是不可接受的情况。另一方面，如果首先追加更新记录，就可以避免这个问题。在重启的时候，恢复过程只是用更新记录来恢复x，崩溃发生在x的新值写进数据库之前还是之后都没什么区别。如果崩溃发生在追加更新记录之后，但是在x更新之前，那么数据库中x的值和更新记录中的前像在系统重启的时候是相同的。恢复过程使用这个前像来覆盖x（这并不改变它的值），但是恢复后的最终状态是正确的。

因此，更新记录必须总是在更新数据库之前追加到日志中去。这被称为先写特性，日志被称为**先写日志**（write-ahead log）。

从崩溃中恢复实际上比我们所描述的要复杂一些。因为性能的原因，大多数数据库系统在主存的页缓冲区中维护正在使用的数据库页的拷贝。对数据库做的修改就记录在这些页中，它们不需要立即传送到大规模存储器上的实际数据库中去。事实上，它们可能在主存中保存一段时间，以便再次访问它们时可以快速处理。另外，因为类似的原因，日志记录也不是实时地写到日志中，而是临时存储在主存的日志缓冲区中。为了避免为每个更新记录都对日志进行一次写操作，当缓冲区满的时候，才把整个缓冲区写到日志中去。这些缓冲区的操纵和它们在更新大规模存储器上日志和数据库的使用必须小心调整以实现原子性和持久性。可以在25.1和25.2节找到这个问题较完整的讨论。

15.2.2　从大规模存储器失效中恢复

持久性要求不可以丢失提交事务对数据库所做的修改。因为大规模存储器设备也可能失效，所以数据库必须冗余地存储在不同的设备上。

实现持久性的一个简单途径是在不同的磁盘上维护数据库单独的拷贝（可能由不同的电源支持），以保证两个磁盘不可能同时失效。**镜像磁盘**实现了这个方法。镜像磁盘是大规模存储器系统，它对用户是透明的，只要应用程序请求磁盘写操作，系统就把相同的信息写到两个不同的磁盘上，因此一个磁盘是额外的拷贝，或者是另一个磁盘的镜像。

在事务处理应用程序中，镜像磁盘系统可以实现增强的系统**可用性**（availability），因为如果其中一个镜像磁盘失效，系统可以使用另一个磁盘继续运行。当更换了失效磁盘后，系统必须重新同步这两个磁盘。相比之下，当只是使用日志来实现持久性的时候（就像下面要描述的那样），磁盘故障后的恢复可能需要比较长的一段时间，在这段时间里，用户无法使用系统。

即使当事务处理系统使用镜像磁盘的时候，它也必须使用先写日志来实现原子性——例如在崩溃后回退事务。

实现持久性的第二种方法涉及到从日志中恢复磁盘上的信息。因为我们不希望恢复过程花费太长的时间或者日志变得太大，所以计划是周期性地拷贝或者**转储**（dump）整个数据库内容到大规模存储器，因此就得到了数据库的快照。另外，除了前像，包含被更新数据项的新值的**后像**（after image）存储在每个更新记录中。

为了从磁盘故障中恢复，系统首先拷贝转储文件到（新的）磁盘上。然后它对日志做两遍扫描——从后向前的扫描（它列出转储后提交的所有事务）和从前向后的扫描（它使用第一遍扫描得到的事务的更新记录的后像，来从转储中记录的状态向前回退数据库到发生故障时它的提交值）。

在这个方法中，一个很重要的问题是如何产生转储。对于一些应用程序来说，可以在关闭系统之后离线产生，此时不允许有新的事务进入，允许所有现存的事务完成。当进行转储的时候，不会有事务在执行，数据库的状态变成在转储开始之前提交的所有事务执行结果的快照。然而对于很多应用程序来说，系统是不能关闭的，转储必须在系统运行的时候进行。

模糊转储（fuzzy dump）是当系统正在运行，事务可能正在执行的时候进行的转储。这些事务后来可能提交也可能异常中止。用模糊转储恢复磁盘的算法必须适当地处理这些事务的影响。可以在25.4节中找到关于介质故障和转储的较完整的讨论。

15.3 实现分布式事务

我们已经假设事务存取的信息存储在一个数据库系统中，但并不总是这样，支持大企业的信息可能存储在多个数据库中。例如，制造企业可以有多个数据库来分别描述库存、生产、员工、资金等等。当这些企业向更高阶段集成的时候，事务就必须存取位于多个数据库中的信息了。因此，启动一个新的零件装配的事务可能通过更新库存数据库来找到零件，通过存取员工数据库来为这个工作指派特定的员工，还要在生产数据库中创建一条记录来描述这个新的活动。这种类型的系统称为**多数据库**（multidatabase）系统。我们将在第18章讨论这样的多数据库系统的数据库和查询设计问题。在这一节中，我们讨论与事务相关的问题。

存取多数据库系统的事务通常被称为**全局**（global）事务，因为它们可以存取企业的所有数据。因为数据库经常位于网络的不同站点，所以这样的事务也称为**分布式**（distributed）事务。设计全局事务时的很多考虑并不依赖于这些数据是否是分布的，所以这两个术语经常是

互换使用的。跨网络的分布引入了新的故障模式（如丢失信息、站点崩溃）和性能问题，这些问题在所有数据库都存储在同一个站点时是不会出现的。

我们假设，多数据库中的每个数据库输出一组可以用来存取它的数据的（局部）事务。例如，银行支行可能通过储蓄和取款事务来存取它维护的账户。可以在本地调用这样的事务来反映纯本地的事件（比如，在本地支行发生的存储到本地支行账户的储蓄），或者通过远程调用来作为分布式事务的一部分（比如，把钱从一个支行的一个账户转到另一个支行的一个账户）。当一个站点的事务作为分布式事务的一部分执行的时候，我们称其为**子事务**（subtransaction）。

每个站点的数据库都有它自己的与存储该站点的数据相关的本地完整性约束。多数据库可能也有与不同站点的数据相关的全局完整性约束。例如，银行总部的数据库可能包含一个数据项，它的值是所有本地支行的数据库中存款的总和。我们假设每个分布式事务都是一致的。每个子事务维护着执行它的站点的局部完整性约束，分布式事务的所有子事务一起来维护全局完整性约束。

在多数据库系统上实现分布式事务的目标是保证它们是全局原子的、隔离的、持久的和一致的。我们已经看到设计者经常选择在一个站点降低隔离级别来执行事务以增强系统性能。就分布式事务来说，有时不仅需要降低隔离级别，而且还需要牺牲原子性和隔离性。为了更好地理解下面的问题，我们首先给出保证全局原子性的可串行化的分布式事务所要求的技术。

15.3.1　原子性和持久性——两阶段提交协议

为了保证分布式事务T的原子性，T的所有子事务必须或者都提交，或者都异常中止。因此，即使一些子事务成功完成，它也不能马上提交，因为T的另一个子事务可能异常中止。如果是这样的话，T的所有子事务都必须异常中止。例如，如果T是在不同站点的两个账户之间转账的分布式事务，假定在一个站点上存款的子事务异常中止，我们不希望在另一个站点上的取款子事务提交。

负责保证分布式事务原子性的事务处理系统的一部分是**事务管理器**（transaction manager）。它的一个任务就是跟踪每个分布式事务有哪个站点参与。当T的所有子事务都完成并且T请求提交的时候，它会通知事务管理器。为了保证原子性，事务管理器和执行子事务的数据库系统遵循一个协议，即**两阶段提交协议**（two-phase commit protocol）[Gray 1978, Lampson and Sturgis 1979]。在描述两阶段提交协议的时候，习惯上称事务管理器为**协调者**（coordinator），称数据库系统为**参与者**（cohort）。

在协议的第一阶段，协调者给T的所有参与者发送**准备好**（prepare）消息，通知它们准备好提交。每个参与者用**投票**（vote）消息来回复。肯定投票表明参与者已经准备好提交。否决投票表明参与者已经异常中止了这个子事务。例如，来自协调者的准备好消息可能会使参与者计算一个推迟了的完整性约束。如果参与者发现违反了约束，那么就回退子事务并发送否决的投票。参与者崩溃或者没有回复消息会被协调者解释为否决投票。

一旦站点S的参与者回复说它准备提交，那么决定就不能逆转了（即它必须保持准备提交的状态），因为协调者根据它收到的投票的准确性来决定将分布式事务提交还是异常中止。特别的，不允许站点S的并发控制随后异常中止这个子事务。另外，所有由子事务执行的更新必

须在发送准备好投票之前存储到非易失性存储器中去，这样，如果事务管理器决定提交分布式事务但是参与者随后崩溃了，那么还可以在恢复的时候提交子事务。

如果协调者收到了所有参与者的准备好投票，那么它就提交T，并给每个参与者发送**提交**（commit）消息，通知它们提交各自的子事务。因为在投票之后就不可能异常中止子事务，所以可以在收到消息的时候提交它。然后，参与者给协调者发送**完成**（done）消息，表明它已经完成协议中它的那部分。当协调者收到每个参与者的完成消息时，协议就终止了。

如果协调者从一个参与者那里收到异常中止投票，那么它就异常中止分布式事务，并给其他的参与者发送**异常中止**（abort）消息。当参与者收到异常中止消息时，它就异常中止子事务。

对每个参与者来说，给协调者发送肯定投票消息和从协调者那里收到提交或异常中止消息的间隔称为**不确定时期**（uncertain period），因为参与者不能确定协议的结果。参与者在这段时期依赖于协调者，因为它不能提交或异常中止子事务。子事务拥有的锁不能释放，直到协调者回复为止（因为协调者可能决定异常中止，由子事务写的新值在那种情况不应该是可见的）。这就给性能带来了负面影响，参与者被称为**阻塞**（blocked）了。不确定时期可能很长，因为在协议中有通信延时。

协调者也可能会崩溃或者因为在这段不确定时期通信失败而变成不可用。因为协调者可能已经决定提交或者异常中止事务，然后在它发送提交或者异常中止消息给所有参与者之前崩溃了，所以参与者不得不维持阻塞直到它发现（如果可能的话）已经做出了什么决定为止。这个过程可能也要持续很长的时间。因为这么长的延时通常在性能上是不可接受的，所以很多系统当不确定时期超过某个事先设定的时间时就抛弃这个协议（可能还有原子性），而只是提交或异常中止本地参与者的子事务（即使其他子事务可能已经以不同的方法终止了）。

在一些情况下，站点管理者可能拒绝数据库管理系统参与两阶段提交协议，因为它对性能可能有负面的影响。在另一些情况下，站点不能参与是因为数据库管理系统是较早的**遗留**（legacy）系统，它不支持这个协议。也可能客户端的软件（如ODBC或JDBC）不支持这个协议。在这种情况下，就不能实现全局原子性了。

可以在26.2节找到关于两阶段提交协议完整的讨论。

15.3.2 全局可串行性和死锁

多数据库系统的每个站点都可能维护着它自己本地的严格两阶段封锁并发控制，这个并发控制调度操作以使在那个站点上的子事务是可串行化的。而且，控制要保证解决那个站点的子事务之间的死锁。

1. 全局可串行性

为了保证分布式事务的可串行性，我们必须保证不仅每个站点上的子事务是可串行化的，而且在所有站点上都有某种全局可串行性。例如，我们不希望站点A的并发控制以某个顺序串行化A上的T_1和T_2的子事务，而站点B的并发控制以相反的顺序串行化B上的T_1和T_2的子事务。在这种情况下，不可能有全局的可串行性。

令人奇怪的是，除了我们已经讨论过的机制，没有其他的机制可以实现全局可串行性：

> 如果每个站点的并发控制独立地使用严格的两段锁协议来本地串行化子事务，并且用两阶段提交协议来提交全局事务，那么全局事务在它们提交的顺序上（全局）是可串行化的。[Weihl 1984]

我们将在26.5节给出这段话的证明。

我们已经讨论了不能实现两阶段提交协议的情况。在这种情况下，不能保证全局事务是全局原子性的或者全局可串行化的。

2. 全局死锁

分布式系统容易产生涉及不同站点的子事务的某种类型的死锁。例如，站点A上的T_1的一个子事务可能在等待A上T_2的一个子事务拥有的一个锁，然而站点B上的T_2的一个子事务可能在等待B上的T_1的一个子事务拥有的一个锁。因为全局事务的所有子事务必须同时提交，所以T_1和T_2就进入了分布式的死锁——它们都将永远等待下去。系统必须有某种机制来检测全局死锁并异常中止等待循环中的一个事务。一个简单的机制就是超时。如果一个站点上的子事务等待了足够长的时间，那么在那个站点上的控制就异常中止这个事务。26.4节将会更详尽地讨论这个问题。

15.3.3 复制

在分布式系统中，数据项的拷贝可以存储在网络的不同站点上。移动系统中的站点可能断开网络连接很长一段时间，所以它们就经常这么做。在断开连接之前，站点将制作一份它在断开连接的状态下可能会用到的数据的拷贝。复制有两个潜在的优点。它可以减少存取数据项所花的时间，因为事务可以存取最近的（甚至可能就是本地的）拷贝。它还可以在发生故障的情况下改善数据项的可用性，因为如果一个包含复制的站点崩溃，那么仍然可以用另一个站点上的不同复本来存取这个数据项。

在复制的大多数实现中，单独的事务不会感知到复本的存在。系统知道复制了哪个数据项以及这个复本存储在哪里。应用**读一个/写全部**（read-one/write-all）的复制算法，当事务请求读取数据项的时候，系统从最近的复本那里得到它的值；当事务请求更新数据项的时候，系统更新所有的复本。系统负责实现复制算法的部分称为**复本控制**（replica control）。

读一个/写全部的复制可以提高速度，利用它可以满足非复制系统上的读请求，因为没有复制，读请求可能要求与远程的数据库进行通信。然而，写请求的性能可能会降低，因为必须要更新所有的复本。因此，在读比写要频繁得多的应用中，利用读一个/写全部复制会提高性能。

读一个/写全部的系统可以分为同步更新和异步更新。

- **同步更新系统**　当事务更新一个数据项的时候，要在事务提交之前更新所有的复本。因此对复本的更新和对其他数据项更新的方法相同。基于两段锁和两阶段提交协议的同步更新系统会产生全局可串行化的调度。
- **异步更新系统**　在事务提交之前只更新一个复本，其他的复本在事务提交之后由另一个程序来更新，这个程序由提交操作来触发或者每隔固定的一段时间周期性执行。因此，即使在基于两段锁协议和两阶段提交协议的系统中，调度也可能不是全局可串行化的。例如，事务T_1可能更新数据项x和y，然后提交。事务T_2可能从已经更新了的一个复本中读取x的值，而从没有更新的一个复本中读取y的值。

异步更新系统又分为两类。

- 事务可以封锁并更新任何复本（假定是最近的一个）。在事务提交之后，更新异步地传

播到其他的复本。结果，并发执行的事务可能更新了同一个数据项不同的复本。由这些事务产生的新值最终会到达每个复本，但是可能是以不同的顺序到达。每个复本站点必须决定首先执行哪个更新，或者是否丢弃整个更新。这就要求利用冲突解决策略来处理这种情况。因为没有策略可以保证对所有的应用程序都是正确的，所以商用系统提供了几种特别的策略，包括“最早更新赢”、“最新更新赢”、“最高优先级站点赢”以及“用户提供冲突解决的过程”。注意，这种形式的复制可能会引起丢失更新问题。

- 每个数据项有一个唯一的复本被指派为**主拷贝**（primary copy），所有其他的复本被指派为**副拷贝**（secondary copy）。要求更新一个数据项的事务必须封锁并更新它的主拷贝。利用这种方法，可以检测到写冲突，并且可以用特定的顺序来进行更新。当事务提交的时候，对主拷贝的更新就传播到其他的复本。当更新主拷贝的事务提交时，可能通过触发单个事务来更新副拷贝。通过主拷贝来排序更新，这个算法可以保证所有的复本看到相同次序的更新，因而不需要任何冲突解决策略。通过存取最近的复本来满足读请求是常见的方式。

因为异步更新比同步更新有着更大的事务吞吐量，所以它是使用最为广泛的复制形式。然而，设计者应该知道异步更新系统（即使是主拷贝系统）也可能产生不可串行化的调度，因此可能产生错误的结果。可以在26.7节中找到关于复制的更为完整的讨论。

15.3.4 总结

分布式事务处理系统的许多方面非常简单。每个站点上的并发控制可以是严格的两段锁协议。两阶段提交协议可以在提交时用来同步参与者。可以用超时来解决全局死锁。

简单的结果是全局支持可串行化调度的分布式事务处理系统可以通过将包含不同厂商开发的数据库管理器的站点互连来实现，只要满足如下要求即可：

- 所有的数据库管理器实现严格的两段锁并发控制。
- 每个站点参加两阶段提交协议。

这些思想大大的简化了分布式系统的设计。

通常这些条件是不成立的。在一些站点上，应用程序可能使用较低的隔离级别，站点可能不参与两阶段提交协议，可能使用异步复制。在这种情况下，就不能保证分布式事务的执行是可串行化的。然而，可能必须在这些约束下设计系统，应用程序设计者也必须仔细评估不可串行化调度对于数据库的正确性和应用程序的最终效用的影响。

15.4 参考书目

可以在[Gray and Reuter 1993]中找到有关实现分布式事务问题的全面论述。[Ceri and Pelagatti 1984]在理论上加以深化，并描述了实现的算法。[Eswaran et al. 1976]中引入了两段锁。[Gray et al. 1976]中讨论了隔离级别。[Gray 1978, Lampson and Sturgis 1979]介绍了两阶段提交协议。[Weihl 1984]中有两阶段提交协议和两段锁的局部并发控制可以保证全局可串行性的证明。可以在[Gray 1978]中找到关于日志和恢复技术的讨论。[Haerder and Reuter 1983]、[Bernstein and Newcomer 1997]和[Gray and Reuter 1993]中对技术进行了详细的总结。[Stonebraker 1979]中引入了主拷贝复制。

15.5 练习

15.1 说明下面哪个调度是可串行化的。

a. $r_1(x)\ r_2(y)\ r_1(z)\ r_3(z)\ r_2(x)\ r_1(y)$

b. $r_1(x)\ w_2(y)\ r_1(z)\ r_3(z)\ w_2(x)\ r_1(y)$

c. $r_1(x)\ w_2(y)\ r_1(z)\ r_3(z)\ w_1(x)\ r_2(y)$

d. $r_1(x)\ r_2(y)\ r_1(z)\ r_3(z)\ w_1(x)\ w_2(y)$

e. $w_1(x)\ r_2(y)\ r_1(z)\ r_3(z)\ r_1(x)\ w_2(y)$

15.2 在学生注册系统中，举出一个会发生死锁的调度的例子。

15.3 给出一个调度的例子，它可能是由非严格的两段锁并发控制产生的，它是可串行化的，但不是以提交的顺序。

15.4 给出一个你接触过的事务处理系统（除银行系统以外）的例子，对于它你希望串行顺序就是提交顺序。

15.5 假设你的大学的事务处理系统包含了一张表，表中为每个当前的学生设置一个元组。

a. 估计存储这张表需要多少磁盘空间。

b. 给出学生注册系统中事务的例子，如果使用表封锁并发控制，那么就要封锁整张表。

15.6 给出一个在READ COMMITTED隔离级别上会发生丢失更新的事务的例子。

15.7 给出在REPEATABLE READ隔离级别上的调度的例子，它在运行SELECT语句之后就插入了一个幻影，并且

a. 得到的调度是非可串行化的，也是不正确的。

b. 得到的调度是可串行化的，因而也是正确的。

15.8 给出在SNAPSHOT隔离级别上调度的例子，并且

a. 它是可串行化的，因而也是正确的。

b. 非可串行化的，也是不正确的。

15.9 给出满足下面条件的调度的例子：

a. 在SNAPSHOT隔离级别上，但不是REPEATABLE READ。

b. 在SERIALIZABLE隔离级别上，但不是在SNAPSHOT隔离级别上（提示：T_2在T_1提交后执行了一个写操作）。

15.10 a. 给出一个涉及两个事务的调度的例子，其中两段锁并发控制使得其中一个事务等待，但是实现SNAPSHOT隔离级别的控制使其中一个事务异常中止。

b. 举出一个涉及两个事务的调度的例子，其中两段锁并发控制使得其中一个事务异常中止（因为死锁），但是实现SNAPSHOT隔离级别的控制允许两个事务都提交。

15.11 某个只读事务读取了前一个月中输入数据库的数据，并用它来准备一个报告。可以运行这个事务的最低的隔离级别是什么？解释你的原因。

15.12 当使用在15.1.4节中给出的封锁实现时，在REPEATABLE READ上运行的事务中的读操作必须获得什么意向锁？

15.13 解释事务的提交是如何在日志系统中实现的。

15.14 解释为什么先写日志需要先写特性。

15.15 解释为什么在两阶段提交协议中的参与者直到它的不确定时期结束后才能释放子事务获得的锁。

15.16 两个分布式事务运行在相同的两个站点上。每个站点使用严格的两段锁并发控制，整个系统使用两阶段提交协议。给出一个运行这些事务的调度，它们在每个站点上的提交顺序是不同的，但是全局调度是可串行化的。

15.17 给出一个不正确调度的例子，它可能是由异步更新复制系统产生的。

15.18 解释如何用触发器来实现同步更新复制系统。

第三部分

数据库的高级主题

第三部分讨论一些更高级的主题。

这一部分将具体讨论面向对象数据库、半结构化数据和XML数据库、分布式数据库、联机分析处理和数据挖掘。

作为独立的数据库产品以及已有的关系数据库产品的扩充，对象数据库开始找到其通向主流数据库的道路。我们将在第16章学习对象数据模型的基本原理和相应的查询语言，以及这些原理在现有标准中的体现。

XML数据库代表一个正在形成的领域，一旦基本标准和工具发展成熟，XML将是一个非常重要的领域。第17章将讨论一些正在形成的标准和应用。

与对象数据库一样，分布式数据库在许多应用中变得愈加重要。第18章将讨论分布式数据库设计、查询设计和查询优化。

数据库技术的其他一些应用包括数据仓库、联机分析处理(OLAP)、数据挖掘。数据仓库是一种为处理复杂只读查询进行了优化的数据库系统，这种只读查询主要应用于复杂的决策支持系统(如对公司销售情况按区域和时段的合计查询)。第19章将讨论可以简化这些查询和提高系统性能的数据结构、语言构造和算法。

第16章 对象数据库

本章介绍对象数据库。首先讨论关系数据模型的缺点，这些缺点的存在推动了对更加丰富的数据模型的需求。与关系数据库不同，对象数据库有许多标准，这使得不同技术之间的关系难以理解。为了更好地学习对象数据库，我们先介绍一个不参考其他具体的标准和语言的对象数据库的概念数据模型。然后给出该领域的两个主要标准：ODMG和SQL:1999的对象关系扩充，并通过概念对象模型对他们各自的特点进行解释。本章的最后将介绍CORBA——由对象管理组织为推动分布式客户/服务器应用的开发而提出的标准。尽管CORBA不是一个纯数据库标准，但是它可以作为开发分布式对象数据服务的框架，因此与本章相关。

16.1 关系数据模型的缺点

由于关系数据库管理系统的基本关系模型比较简单（对应用开发者具有吸引力）以及表成为商业应用中的大多数数据的合适的表示形式，20世纪80年代，关系数据库系统风靡整个数据库市场。受这种商业上的成功的鼓舞，人们试图把关系数据库应用于其他一些非关系模型的领域，如计算机辅助设计（CAD）和地理数据。关系数据库很快显示出并不适用于这些“非传统”的应用。甚至在核心应用领域，关系数据库也有一些缺陷。本节用一些简单示例来说明关系数据模型的问题。

1. 集合-值属性

考虑如下的使用社会保险号、姓名、电话号码、子女情况来描述的人的关系模式：

PERSON (SSN: String, Name: String, PhoneN: String, Child: String)

假设一个人可以有多个电话号码和多个子女，并且Child是关系Person的一个外键。所以这个模式的键包含属性SSN、PhoneN和Child。下图所示是该模式的一个可能的关系实例[⊖]：

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

⊖ 为了简单起见，忽略了一些用于满足外键约束的元组。

由于存在如下的函数依赖，这个模式不是第三范式。

$$SSN \rightarrow Name$$

因为SSN不是键并且Name不是键所包含的属性。而且容易验证，如果首先在SSN、NAME、PhoneN和SSN、Name、Child上进行投影来分解这个关系，然后再对投影进行联结，又可以得到原来的关系。所以根据8.9节的论述，这个关系满足如下的联结依赖:

$$(SSN, Name, PhoneN) \bowtie (SSN, Name, Child)$$

在8.9节中，我们讨论过如果一个关系满足非平凡的联结依赖，则它可能含有许多冗余信息（事实上，比函数依赖引起的冗余信息多）。由于信息冗余是引起更新异常的一个原因，所以关系设计理论要求我们把原始的关系分解为如下的3个关系:

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

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

尽管这样进行分解确实去除了更新异常，但还存在其他一些困难。考虑查询找到Joe的所有孙子的电话号码。下面的SQL语句:

```
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)

对原始模式进行查询，而如下语句

```
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)

对于分解后的模式进行同样的查询。这两个SQL表达式在实现我们刚才给出的查询时显得比较麻烦。

问题在于PERSON的原始关系模式中的冗余只是因为关系数据模型不能以一种自然的方法处理集合-值属性。对于原始表来说，更合适的模式是

```
PERSON(SSN: String, Name: String,
       PhoneN: {String}, Child: {String})
```

其中，括号{}代表集合-值属性。例如，Child:{String}表示一个元组中的属性Child的值是String类型的元素集合。这样的表中的行具有如下的形式（注意其中的集合-值元素）：

```
(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})
```

关于上述例子的第二个问题是SQL处理查询16.1和16.2显得比较笨拙。假设属性Child的类型是{PERSON}而不是{String}，并且SQL可以处理作为PERSON元组集合（而不仅仅是表示SSN的字符串集合）的属性Child的值。这样就可以使查询更简明和自然，因为表达式P.Child.Child可以被赋予精确的含意：由所有对应于P的孩子的孩子的元组组成的集合。这就使得我们可以用如下所示的简洁方法写出上述查询：

```
SELECT   P.Child.Child.PhoneN
FROM     PERSON P                                    (16.3)
WHERE    P.Name = 'Joe Public'
```

P.Child.Child.PhoneN形式的表达式称为**路径表达式**（path expression）。

2. IsA *层次结构*

假设数据库中的一些（而不是所有的）人是学生。因为学生也是人，我们想提炼出所有人共同具有的一般性信息，并且使STUDENT的模式只包含学生的特定信息：

```
STUDENT(SSN: String, Major: String)
```

则可以推论，因为STUDENT也是PERSON，所以每个学生都有一个Name属性。例如，查询找出计算机科学系的所有学生可以写成如下形式：

```
SELECT   S.Name
FROM     STUDENT S
WHERE    S.Major = 'CS'
```

在SLQ-92中，上述查询将被拒绝，因为属性Name没有明确地包含在STUDENT的模式中。但是，如果系统知道学生与人之间的IsA联系，它可以推断STUDENT从PERSON继承了Name属性。

从实体-联系模型中，我们对IsA层次结构的概念已经非常熟悉。但是，E-R模型并不能用于查询语言。所以我们不得不使用关系模型和标准的SQL，写出如下的更复杂的查询：

```
SELECT   P.Name
FROM     PERSON P, STUDENT S
WHERE    P.SSN = S.SSN AND S.Major = 'CS'
```

从本质上说，SQL-92程序员必须使每一个查询都包含一个IsA联系的实现。

3. blob

blob代表二进制大对象（binary large object）。实质上，所有关系DBMS都允许关系具有blob类型的属性。例如，一个电影数据数据库可能拥有如下的模式：

```
MOVIE (Name: String, Director: PERSON, Video: blob)          (16.4)
```

属性Video可能拥有视频流，这些视频流可以包含几千兆字节的数据。按照关系的观点，视频流是一个巨大的、无结构的位序列。

blob存在一些问题。考虑查询:

```
SELECT   M.Director
FROM     MOVIE M
WHERE    M.Name = 'The Simpsons'
```

为了计算WHERE 子句，一些系统可能会把包含blob的所有元组都从磁盘上取出放入内存中。这会产生巨大的开销。即使数据库在处理blob方面得到优化，它的可选择余地仍然有限。假设我们只需要从20 000 ~ 50 000帧的数据。如果仍使用传统的关系模型，将无法得到这些信息。为了处理这样的查询，我们需要一个专门的例程frameRange(from，to)，它可以实现为存储过程，这样对于特定的视频blob返回指定范围内的帧。

把frameRange()作为一个操作符加入关系数据模型有意义吗？这种添加可能会使任何人都可以使用视频blob，但是它对于那些存储DNA序列和VLSI芯片设计的blob毫无帮助。所以，与其用这些专门的操作来束缚数据模型，不如提供一种机制使得用户可以为每一类blob单独定义操作。

4. 对象

刚才讨论的SQL的缺点使人们产生了开发存储和检索对象的数据库的想法。如附录中所作的解释，对象包含一组属性和可以存取那些属性的一组方法，以及相关的继承层次结构。

属性值可以是复合数据类型或其他对象的实例。例如，对象person可能包含一个表示他的地址的属性和表示他的配偶的属性。

因为属性的值可能是对象的实例，为对象定义的操作可以用在查询中。所以，视频对象可能有一个方法frameRange()用于查询中。

```
SELECT   M.frameRange(20000,50000)
FROM     MOVIE M
WHERE    M.Name = 'The Simpsons'
```

5. 数据库语言中的阻抗不匹配

不太可能完全用SQL编写一个完整的应用程序，所以典型的数据库应用程序都是用宿主语言（如C或Java）编写的，并且通过执行嵌入宿主程序的SQL查询来访问数据库。第10章讨论了从宿主语言访问数据库的一些机制[⊖]。

这个方法的问题在于，SQL面向的是集合，意味着它的查询返回的是元组集合。相反，C、Java以及其他宿主语言不理解关系并且不支持关系上的高层次的操作。除了类型不匹配之外，SQL的声明性（指明作什么）与宿主语言的过程性（程序员必须说明怎么作）存在明显的不同。这种现象加重了数据访问语言和宿主语言之间的**阻抗不匹配**（impedance mismatch），所以人们发明了游标机制作为过程性宿主语言和SQL的适配器。

早期开发对象数据库的一个原因就是它有可能消除阻抗不匹配。其基本思想比较简单：选择一种典型的面向对象语言（C++和Smalltalk是当时主要的候选对象；20世纪90年代增加

⊖ 存储程序使用的语言（SQL/PSM，见第10章）在计算上是完整的，但是很难说它是C、C++或Java语言的合适的替代者。

了Java）作为数据操纵语言。因为对象语言和对象数据库的基础都是对象，所以不会出现阻抗不匹配的问题。

当时，这种想法似乎是非常吸引人的，但是其中也存在一些问题。首先，仍然需要一种强大的、声明性的查询语言（如SQL）来支持复杂的数据查询。由于C++和Java并不能容易地扩展为声明性的查询语言，因此阻抗不匹配仍然存在[⊖]。

另一个困难是，不再只有一种数据操作语言，SQL（尽管与宿主语言不匹配），我们现在具有和宿主语言一样多的数据操作语言。这本身并没什么坏处，因为这些宿主语言中没有任何一个是新的并且多数程序员都比较熟悉这些语言。问题是不同的对象语言具有稍有不同的对象模型。所以，难于定义一个，统一数据模型。例如，如果一个对象由使用某种宿主语言的应用程序所产生，那么使用由另一种宿主语言编写的应用程序进行访问就可能出现问题。

本章讨论的两个主要的对象数据库标准（SQL:1999和ODMG）对于阻抗不匹配问题的影响具有不同的观点。其中一部分原因是上述困难，另一部分原因是设计理念的不同，SQL:1999并不以消除阻抗不匹配为目标。相反，ODMG把消除阻抗不匹配视为主要的（即使还比较难以琢磨）设计目标之一。

对象数据库与关系数据库

前面的例子概括了对象模型及其与关系模型的关系。

- 关系数据库包含关系，即元组的集合，而对象数据库包含类，即对象的集合。所以，关系数据库可能有一个称为PERSON的关系，拥有包含每一个人的信息的元组，而对象数据库可能有一个称为PERSON的类，拥有包含每一个人的信息的对象。通过为每一个关系定义一个类，特定的关系数据库可以在对象模型中实现。特定类的属性为对应的关系的属性，每一个经过类的实例化得到的对象对应一个元组。
- 在关系数据库中，元组的组成部分必须是基本类型（字符串、整数等）；在对象数据库中，对象的组成部分可以是复杂类型（集合、元组、对象等）。
- 对象数据库具有一些关系数据库不具备的性质：
  - 对象可以组成继承层次结构，即允许低类型的对象从高类型的对象继承属性和方法。这有助于减少类型规格说明中的混乱并且产生更简洁的查询。
  - 对象可以有方法，并且可以从查询中调用这些方法。例如，前面提到的关于MOVIE类的规格说明可能包含一个方法frameRange，并具有如下的形式的声明：

    ```
    list(VIDEOFRAME) frameRange(Integer,Integer);
    ```

    其中指出frameRange具有两个整数类型参数并且返回视频帧的列表。这样的声明通过专门的对象定义语言实现，该语言与SQL的数据定义子语言非常类似。
- 方法的实现可以使用标准的宿主语言（如C++或Java）预先进行编写，并存储在服务器上。在这一点上，方法类似于SQL数据库中的存储过程（见10.2.5节）。但是，存储过程与任何特定的关系无关，而存储的方法是相应类中的一个部分并且可以沿着对象类型层

⊖ 在这里，我们应该提及正在进行的关于**Java数据对象规格说明**（Java Data Objects Specification）的工作，它的目标是开发一种能更好地混合到Java语法中的查询语言。

次结构被继承，就像面向对象程序设计语言中的方法一样。

- 在一些对象数据库系统中，数据操纵语言和宿主语言为同一语言。

16.2 发展历史

下面的历史回顾可以帮助我们更好地理解对象数据库的演化发展。

1. 20世纪80年代早期：嵌套关系（也称为非1NF）

嵌套关系的思想是早期为解决关系数据模型的一些限制而开发的方法[Makinouchi1977, Arisawa et al.1983, Roth and Korth 1987, Jaeschke and Schek 1982, Ozsoyoglu and Yuan 1985]。在嵌套关系中，属性可以为relation类型。所以，在表

111-22-3333	Joe Public	516-345-6789 516-123-4567		222-33-4444 333-44-5555	Bob Public Sally Public
222-33-4444	Bob Public	212-987-6543 212-987-1111		444-55-6666 555-66-7777	Maggy Public Mary Public

中，每一个元组有四个组成部分。和传统关系模型一样，前两个组成部分是原子值。但是，第三和第四个属性的值不是原子的：它们分别是一元和二元关系。第一个元组的第三部分中的一元关系代表Joe Public的电话号码列表，而这个元组中的第四部分中的关系描述了Joe的子女（通过社会保险号和名字）。

经验表明，嵌套关系模型具有很多限制，并且其查询语言和模式设计理论十分复杂。

2. 20世纪80年代中期：持久对象

持久对象的想法来自于程序设计语言并且可以追溯到20世纪60年代。从概念上说，早期的对象数据库系统就是持久的C++或Smalltalk。

持久对象的基本思想简单而优雅。

- 用户可以声明一些C++或Smalltalk对象为**持久**对象。
- 当应用程序在程序中引用持久对象时，系统检查对象是否在内存缓冲区内。如果不在，产生一个**对象故障**（object fault），系统把对象放入内存中，这一过程对用户程序来说是透明的。
- 当程序执行完毕，如果更新了持久对象，则对象的新版本将以对用户透明的方式返回到大容量存储器中。

具有持久对象的程序设计语言消除了数据库数据类型和宿主语言的数据类型间的阻抗不匹配的问题[⊖]。但是，它们不提供类似于SQL中的SELECT语句那样的高层声明性查询机制。其结果是所有的查询都必须使用宿主语言中可用的结构过程化地编程。我们将在16.4.5节看到ODMG标准用某种程度的阻抗不匹配换取了对“豪华”的声明式查询语言的支持。

⊖ 注意，持久数据类型仅在语言具有丰富的数据类型时才能减少阻抗不匹配。例如，使C的数据类型持久化后，由于它缺乏集合数据类型，所以不太可能对这个问题有所帮助。

3. 20世纪90年代早期：对象-关系数据库

对象-关系数据库（object-relational database）的基本思想是：关系仍然是元组的集合，但是元组的组成部分可以是对象。对象-关系模型在两个早期产品UniSQL和Illustra中应用。对象-关系数据库并不像近来的一些对象数据库那样通用。在近来的这些数据库中，任意对象（不仅是关系）都可以位于数据模型的最高层。但是，它们在接近。现在，多数关系数据库产品的最新版本都支持对象-关系特征，并且SQL:1999标准也有了对象-关系扩展。16.5节将对这些内容进行介绍。

4. 20世纪90年代中期：对象DBMS的增殖

一些早期的对象DBMS系统，包括O_2、GemStone、ObjectStore、Poet、Versant和其他系统使得这个领域趋于成熟。每一个实现都对基本的概念框架作出了贡献。

5. 20世纪90年代的中期到后期：对象数据管理标准

这个领域发展到了一个新的阶段，需要新的标准来推动用户接受新的模式。我们将讨论其中三个标准。

ODMG

ODMG标准（由对象数据库管理组织开发）涉及对象数据库和（作为特例）对象-关系数据库系统。它的数据模型支持对象，关系是简化的特殊对象类型。该标准有如下几个主要部分：

- ODL——对象定义语言：如何指定数据库模式。
- OQL——（和SQL类似）对象查询语言。
- 宿主语言绑定——如何从过程化的语言内部使用ODL和OQL。标准定义了C++、Smalltalk和Java的绑定。在ODMG中，宿主语言也作为对象操纵语言。

SQL:1999

SQL的最新版本，SQL:1999支持对象-关系数据模型的一个子集。顶层结构是包含了元组集合的关系，但是元组的组成部分可以是对象。SQL:1999标准与ODMG在对象-关系数据库系统上有部分重叠，但是语法却有极大不同。另外，为了把关系模型推广到对象-关系模型，SQL:1999为语言添加了一些其他的特性，如：

- 增强SQL数据定义、操纵和查询语言（7.6节和16.5节）。
- 触发器（第9章）。
- 在10.2.5节讨论的SQL持久存储模块（SQL/PSM）语言，可以使程序员编写存储过程[⊖]。
- 调用级接口（SQL/CLI）的定义，类似于ODBC（10.6节）。
- 多媒体扩充。

CORBA

在对象数据库的演化过程中，人们把相当大的努力投入到了客户/服务器系统的标准的开发上，在这种系统中客户可以访问服务器上的对象。对象管理组（OMG）[⊖]提出了这样一个标准，即CORBA，表示公共对象请求代理体系结构。这个标准的最近的版本允许客户以事务

⊖ 定义存储过程的能力十分重要，所以SQL/PSM重新包含在了SLQ-92标准中。

⊖ 对象管理组织OMG和对象数据库管理组织ODMG是不同的机构。

的方式访问位于远程服务器上的持久对象，并且使用声明性的查询语言执行这些访问。所以，CORBA可以提供对象数据库管理系统的功能。16.6节将对CORBA做进一步的讨论。

16.3 概念上的对象数据模型

为了避免迷失在ODMG和SQL:1999标准的细节中，我们首先介绍适合对象数据库的数据模型的概念上的观点。**概念对象数据模型**（Conceptual Object Data Model）(CODM)来自于O_2的研究小组的工作[Bancilhon et al. 1990]，O_2对象DBMS对ODMG标准产生了非常大的影响。事实上，CODM接近于ODMG。但是，它不受一些实际细节以及为了兼容现有产品和标准而进行的低层实现的束缚。另外，SQL:1999的对象-关系扩展可以用CODM方便地进行解释，这使得两个标准间的关系更易理解。

在CODM中，每一个对象都有唯一不可更改的标识，称为**对象Id**（oid），它独立于对象的实际值。oid在创建对象时由系统赋予并且在对象的整个生命周期中都不改变。注意，oid与关系中的主键不同。同oid一样，主键唯一确定对象。但是与oid不同的是，主键的值可以改变（一个人可能会改变他的社会保险号码）。另外，oid是一个内部对象名，正常情形下对程序员不可见（在一些语言中甚至不能通过属性访问）。如果需要反复引用一个对象，那么在对象被创建和检索后需要把oid存储在一个变量中。而另一方面，主键是可见的，并且为了检索一个特定的对象可以在查询中显式地提及主键。

16.3.1 对象和值

下面是一个描述，Joe Public的对象的例子：

```
(#32, [  SSN: 111-22-3333,
         Name: Joe Public,
         PhoneN: {"516-123-4567", "516-345-6789"},
         Child: {#445, #73}] )
```
(16.5)

符号#32就是数据对象的oid，它代表真实世界中的Joe Public。余下的部分说明了对象的值。oid在其他对象中确定了这个对象，值提供了关于Joe的实际信息。注意，Child属性的值是一组PERSON对象的oid，这些PERSON对象描述了Joe的两个孩子。

形式上，一个**对象**（object）具有二元组的形式(oid,val)，其中oid代表对象Id，val代表值。**值**（value）部分val可以拥有如下形式：

- 基本值（primitive value）——如整数型、字符串型、浮点型或布尔型的数据，例如："516-123-4567"。
- 引用值——对象的oid。例如，#445。
- 元组值——为$[A_1:v_1,\cdots,A_n:v_n]$形式，其中$A_1,\cdots,A_n$是不同的属性名，并且$v_1,\cdots,v_n$为相应的值。例如，在（16.5）的对象#32中的全部值部分（括号内的）。
- 集合值——为$\{v_1,v_2,\cdots,v_n\}$形式，其中$v_1,v_2,\cdots,v_n$是值。例如：{"516-123-4567", "516-3456-6789"}。

为了区别于基本类型，引用值、元组值以及集合值称为**复杂值**（complex value）。实际的ODMG数据模型包含额外的复杂值类型，如包（同一元素可以具有多个副本的集合）、列表、

结构、枚举类型和数组。但是，在此不考虑这些类型，因为它们没有为整个概念框架添加新的内容。

注意，对象的oid部分是不能改变的（如果改变，就可能表示不同的对象）。相反，对象的值作为更新的结果可以被其他值替换。例如，如果要变更Joe Public的电话号码，将PoneN属性的值用一个新值替换，但是对象仍保持同样的oid并被认为是同一个对象。

16.3.2 类

在面向对象系统中，语义相似的对象组织成为**类**（class）。例如，所有代表人的对象组成了PERSON类。

类在CODM中扮演的角色与关系在关系数据库中扮演的角色一样。但是，在SLQ-92中，数据库是一组关系，每个关系又是一组元组。在CODM中，数据库是一组类，每一个类是一组对象。所以，在SQL-92中，我们可以有称为PERSON的关系，其元组包含了一个人的信息，在CODM中可以有一个称为PERSON的类，其对象含有每一个人的信息。

所以，类是一种把对象组成不同类别的方法。类具有**类型**（type）以及方法签名（method signature），类型描述类中所有对象的公共结构，方法签名是可以用于类对象的操作的声明。稍后将更详细地讨论这些概念。这里注意，只有方法签名是CODM的一部分，方法的实现并不是CODM的一部分。方法的实现是一个存储在数据库服务器上的由宿主语言编写的过程。ODBMS必须提供一种无论何时在程序中使用该方法时，都可以调用到适当的实现的机制[⊖]。

在关系数据模型中，两个表可以通过关系间约束（如外键约束）建立关联。在对象数据模型中，有一种联系扮演着特殊的角色。假设除了把所有代表人的对象组织在一起的PERSON类外，我们还有STUDENT类，它把所有表示学生的对象组织在一起。自然地，每一个学生都是人，所以组成STUDENT类的对象集合必须是组成PERSON类的对象集合的子集。这是**子类联系**（subclass relationship）的一个例子，其中STUDENT是PERSON的子类。子类联系也称为IsA联系或**继承联系**（inheritance relationship）。

赋予类的所有对象的集合称为类的**外延**（extent）。为了充分反映我们关于子类联系的直觉，**外延**必须满足如下的性质：

如果C_1是C_2的子类，那么C_2的外延包含C_1的外延。

例如，因为STUDENT是PERSON的子类，所有学生的集合是所有人的集合的子集。

查询语言和数据操纵语言都明白子类联系。例如，如果假设查询返回所有具有某些性质的PERSON对象，并且如果一些STUDENT对象具有那些性质，查询将返回查询结果中的STUDENT对象，因为每一学生都是人。

综上所述，类具有类型，类型描述了类的结构，方法签名（经常被认为是类型的一部分），外延，列出属于类的所有对象。这样，类、类型，以及外延就构成了彼此相关而又不同的概念。

⊖ 通常，这样的代码在服务器端执行。但是，用Java实现的方法可能在客户端执行，因为Java提供了在机器间传送代码的机制。

16.3.3 类型

任何数据模型的一个重要需求是数据必须适当地结构化。由于基本数据结构简单，所以在关系模型中完成类型化并不是一个大问题。在对象数据库中情况比较复杂。例如可以考虑(16.5)中的对象。可以说它的类型（称为PERSON）由下面的表达式表示:

```
[SSN: String, Name: String, PhoneN: {String}, Child: {PERSON}]        (16.6)
```

这个类型定义指出属性SSN和Name从基本域String中取值；属性PhoneN的值必须是串的集合；属性Child的值是PERSON对象的集合。

简单地说，对象的类型是对象组成部分的类型的集合。更准确地说，适合于结构化对象的复杂类型可以定义如下:

- 基本类型。字符串型，浮点型，整数型等。
- 引用类型。用户定义的类名，如PERSON和STUDENT。
- 元组类型。$[A_1:T_1,\cdots,A_n:T_n]$形式的表达式，其中，A_i是互不相同的属性名，T_i是类型。（例如，在16.6中给出的类型。）
- 集合类型。{T}形式的表达式，其中T是类型。例如{String}。

注意，16.6描述了一种类型，其中一个关系嵌套在另一个关系中。Child的每一个值都是一个关系：一组PERSON类型的元组。另一方面，字符串为基本类型，所以PhoneN的值是一组基本类型值。

一个对象符合某一类型是什么意思呢？由于对象的递归结构以及IsA层次结构，使这个问题的答案不太简单。我们将在下面的段落中逐步介绍类型的概念。

1. 子类型

除了结构化地组织对象之外，类型系统可以确定与其他类型相比，哪一种类型更“结构化”。例如，假设PERSON对象的类型是

```
[SSN: String, Name: String, Address: [StNumber: Integer,
                                       StName: String]]
```

这是一个元组类型，其中前两个组成部分具有基本类型，第三个组成部分为元组类型。

现在考虑STUDENT类型的对象。显然，学生具有姓名、地址，以及PERSON对象所具有的其他所有内容。但是，学生还有其他的属性，所以一个适当的类型可能是

```
[SSN: String, Name: String,
 Address: [StNumber: Integer, StName: String],
 Majors: {String}, Enrolled: {COURSE}]
```

直觉提示我们STUDENT类型比PERSON类型含有更多的结构。第一，它具有PERSON的所有属性。第二，这些属性的值至少与相应的PERSON的属性有同样多的结构。第三，STUDENT具有PERSON中没有的属性。这种直觉导出**子类型**（subtype）与**超类型**（supertype）的概念：如果$T \neq T'$并且如下的条件之一成立，那么类型T是T'的子类型（超类型）。

- T和T'是引用类型，T是T'的子类。
- $T=[A_1:T_1,\cdots,A_n:T_n, A_{n+1}:T_{n+1},\cdots,A_m:T_m]$ 和$T'=[A_1:T_1',\cdots, A_n,:T_n']$是元组类型（注意，T包含T'的所有属性，即$m \geqslant n$），且$T_i = T_i'$或$T_i$为$T_i'$的子类型（对于每一个$i=1,\cdots,n$）。
- $T=\{T_0\}$和$T' = \{T_0'\}$都是集合类型，且T_0是T_0'的子类型。

根据这个定义，STUDENT是PERSON的子类型，因为它包含了为PERSON定义的所有结构并且具有自己额外的属性。但要注意，具有额外的属性不是类型成为子类型所必须的条件。例如

```
[SSN: String, Name: String,
 Address: [StNumber: Integer,  StName: String,
           POBox: String] ]
```
(16.7)

也是PERSON的子类型，尽管它不具备超出PERSON定义范围的属性。16.7中的Address属性比PERSON中的Address属性具有更多的结构。

2. 类型的域

类型的**域**（domain）决定所有符合该类型的对象。直观地说，类型T的域（用domain(T)表示）是T的各组成部分的域的适当组合。更准确地说，

- 基本类型的域（如整数型或字符串型）就是我们所期望的（分别是所有整数和字符串的集合）。
- 引用类型T的域是T的外延，即类T中的所有对象的Id的集合。例如，如果T是PERSON，它的域是所有PERSON对象的oid集合。
- 元组类型的域$[A_1:T_1,\cdots,A_n:T_n]$为$\{[A_1: w_1,\cdots, A_n : w_n] \mid n \geq 0$且$w_i \in \text{domain}(T_i)\}$，即所有元组值的集合，这些元组值的组成部分符合属性相应的类型。所以，(16.6)的类型的域是(16.5)形式的所有值的集合。
- 集合类型的域{T}是

$$\{\{w_1,\cdots, w_k\} \mid k > 0 \text{ 且 } w_i \in \text{domain}(T)\}$$

即，它包含符合给定类型T的值的集合。例如，类型{COURSE}的域是这样的集合，其成员是COURSE对象的oid的有限集合。

容易看出，域是以如下的方式定义的，即子类型的域S（从某种意义上说）是超类型S'的域的子集。更准确地说，对于S中任何特定的对象o，或者o已经在S'中，或则可以通过去除o中包含的子对象元组的一些组成部分得到S'中的对象o'。例如，PERSON对象可以通过去除STUDENT对象中的Major和Enrolled属性得到。类似地，PERSON对象可以通过去除（16.7）类型的对象中所含的嵌套地址的POBox组成部分而获得。

3. 数据库模式和实例

在对象数据库中，**模式**（schema）包含了可以存储在数据库中的对象的每一个类的规格说明。每一个类C包含以下部分：

- 与C相关的类型。该类型决定了C的每个实例的结构。
- C的方法签名。**方法签名**（method signature）指定方法的名字、类型和方法的参数的顺序，以及方法产生的结果的类型。例如，方法enroll()具有下面的签名：

```
Boolean enroll (STUDENT, COURSE);
```

方法enrolled()具有签名

```
{COURSE} enrolled (STUDENT);
```

enroll()的签名表示要登记一个学生，必须提供给一个STUDENT类的对象和一个COURSE类的对象。该方法返回一个布尔值来指出操作的结果。enrolled()的签名表示，为了检查学

生的登记情况，必须提供一个STUDENT对象，返回的结果是一组学生登记过的COURSE的对象。

- *subclass-of* 联系，确定C的超类。
- 完整性约束，如类似于关系数据库中的约束的键约束、引用约束，或更一般的断言[⊖]。

数据库**实例**（instance）包含模式中指定的特定类的对象。对象必须满足模式包含的所有约束，且每一个对象都必须具有唯一的oid。

这样就完成了概念对象模型的定义。如你所见，CODM中使用的大多数概念都是从我们熟悉的关系模型扩充而来，由于其基础的数据模型十分丰富，所以需要小心地使用。

16.3.4 对象-关系数据库

对象-关系数据库于20世纪90年代初期出现，那时许多厂商开始倡导对象-关系数据库系统，认为它是从关系数据库迁移（到对象数据库）的一条比较安全的途径。主要的卖点是这些数据库可以通过扩展已经存在的关系数据库来实现。经长时间的考虑，SQL:1999工作组最终采用了对象-关系数据模型的一个有限子集，该子集对应于CODM的一个子集。一般情况下，一个对象-关系数据库包括：

- 一个关系集合（可以看作一个类）。
- 每个关系包含一个元组集（可以看作表示关系的类的实例）。
- 每一条元组的形式为（oid，val），其中oid代表对象ID，val代表元组值，其组成部分可以是任意值（例如，基本值、元组集、对其他对象的引用）。

对象-关系模型和CODM模型的主要区别是前者中的每一个对象实例的顶层结构总是一个元组，而后者的顶层结构可以是任意值。但是，对象-关系DBMS的这种限制并没有显著削弱它对现实世界的建模能力。

对象-关系模型与传统关系模型的区别在于关系模型中的元组组成部分必须是基本值，而对象-关系模型中的元组组成部分则可以是任意值。所以，关系模型可以视为对象-关系模型的一个子集（所以也是CODM的一个子集）。

16.5节将讨论SQL:1999支持的对象-关系模型的子集。

16.4 ODMG标准

本节给出ODMG标准的各个方面的一个概述。关于ODMG3.0的权威性的指南可参阅[Cattell and Barry 2000]。我们将讨论如下内容：

- 数据模型 什么样的对象能够存储在ODMG数据库中？
- 对象定义语言（ODL） 如何在数据库管理系统中描述这些对象？
- 对象查询语言（OQL） 如何对数据库进行查询？
- 事务机制 如何指定事务的边界？
- 语言绑定 如何在“现实世界”中使用ODMG数据库？

⊖ ODMG标准提供指定键和参照完整性的方法，但是没有定义一般的SQL式的断言的语言。SQL:1999标准从SQL-92继承了约束规格说明语言。

尽管ODMG的基础数据模型与CODM非常近似，下面的小节还是先简要地讨论一下它们的不同之处。然后介绍ODL，即ODMG的对象定义语言，它是一种描述类及其类型的具体语法，接着讨论ODMG的查询语言OQL，并强调它与16.1节中讨论的查询语言的不同之处。

ODMG数据库体系结构

图16-1和16-2描述ODMG数据库的总体体系结构。同关系数据库一样，使用ODMG数据库的应用程序用如C++这样的宿主语言编写。为了访问数据库，应用程序必须与ODBMS库以及实现类方法的代码链接。在对象数据库中，大多数操纵对象的代码都是数据库自身的一部分：每个类都有适用于其类对象使用的方法集。这些方法的代码存储在数据库服务器上。方法签名则通过ODL定义为模式的一部分。当一个方法被调用时，ODBMS负责启用适当的代码。原则上，方法在服务器端或客户端都可以执行。但是，在客户端执行方法需要ODBMS提供向客户端计算机运送代码的机制。由于实现这样的机制十分困难，所以商用DOBMS一般在服务器端执行方法。幸运的是，Java语言的出现令代码运送变得非常容易，我们可以在将来的产品中看到更多的客户端方法调用。

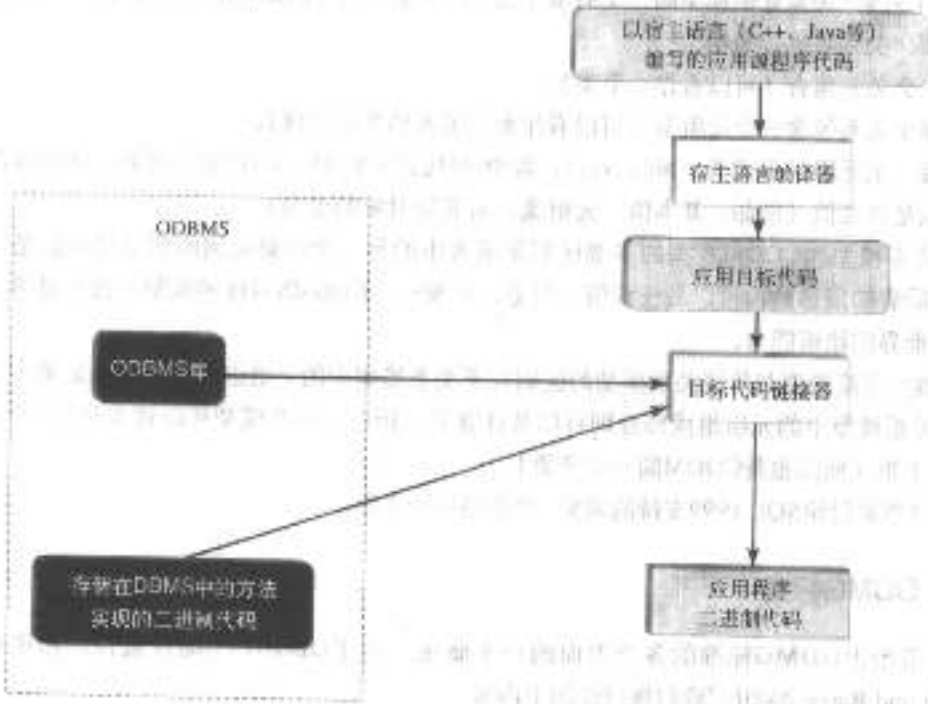


图16-1　ODMG应用结构

在服务器上存储代码会令人想起关系数据库中的存储过程（参阅10.2.5节）。但是，二者最大的不同之处在于方法的实现是对象整体的组成部分，而存储过程是为了性能和安全的原因存储在数据库中的外部应用程序。

消除数据存取语言和宿主语言间的阻抗不匹配是ODMG的最重要的设计目标之一。这个设计目标的直接现实含义是宿主语言也用作数据操纵语言。这可以通过**语言绑定**（language binding）来实现。所谓语言绑定是一组标准（每种宿主语言一个标准），它定义了C++、Java

和Smalltalk中的对象定义如何映射到数据库对象。这种映射使得宿主程序通过标准的宿主语言语句（赋值、增加等）或通过执行为相应类定义的方法来直接操纵数据库对象。一旦发生了这种交互，ODBMS负责确保事务提交后的结果改变是持久的。

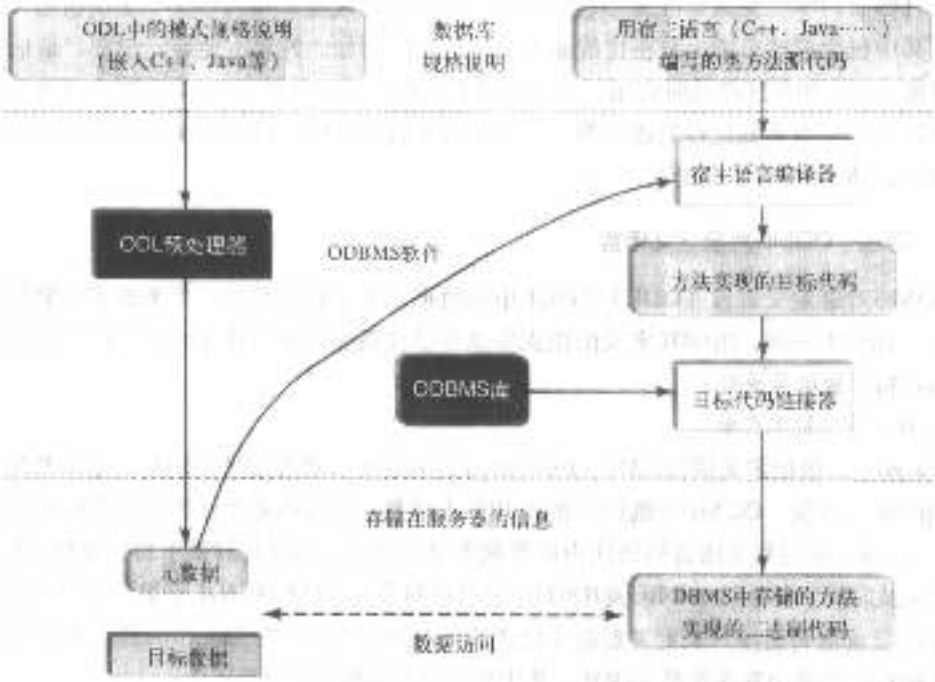


图16-2 ODMG数据库体系结构

通常，应用程序用OQL查询来寻找对象或对象集合，然后，应用程序使用程序设计语言语句和对象的方法来操纵和更新那些对象。此时不需要SQL那样专门的更新语句，因为在程序中对任何对象的更新都会引起数据库中该对象副本的更新。例如，如果宿主语言变量Stud包含持久对象STUDENT的oid，则如下的C++或Java指令

```
Stud.Name = "John";
```

将使得数据库和程序中学生的姓名都更改为“John”[⊖]。

注意，这种数据访问的方式与SQL数据库中使用的数据访问方式极为不同。在SQL数据库中，对数据的任何查询和修改都是通过SQL这种与典型的宿主语言极为不同的语言来完成的。即使宿主语言是面向对象语言，情况也是这样，例如在第10章讨论的Java语言使用JDBC和SQLJ。相反，在ODMG数据库中，数据直接用宿主语言进行修改，程序中其他的对象也用同样的方式进行修改。

ODMG提供专门的方法允许应用程序向数据库发送OQL查询（即SELECT语句）。这种设计被认为是整个ODMG方案中的“丑小鸭”，因为它产生了声名狼藉的阻抗不匹配问题。为了解决这个问题，目前正在努力形成中的Java**数据对象规格说明**（Java data object specification）的目标是开发一种能更好适应Java语法的查询语言。完成后，这个规格说明可能会替代现有

⊖ 注意，ODMG类似C++和Java，使用双引号表示字符串，相反，在SQL中单引号表示字符串。

的ODMG Java绑定语言。

根据上述讨论可知，典型的应用使用两类对象：存储在数据库中的**持久**（persistent）对象和不在数据库中存储的保持应用逻辑需要的辅助信息的**瞬时**（transient）对象。例如，一个向还未付学费的学生发送提醒通知的邮件合并应用查询数据库并为每一个学生创建一个临时对象，其中包含那些必须出现在提醒通知中的信息（例如，姓名、地址、学费、最后期限）。这些对象只在应用程序活动时有用，而且不需要保存。另一方面，STUDENT对象可能需要更新和保存以反映学费通知已经发送的事实。定义对象持久性的严格的语法是语言绑定规格说明的一部分，16.4.5节将予以讨论。

16.4.1 ODL：ODMG对象定义语言

ODMG对象定义语言（ODL）与SQL中的数据定义子语言类似。用来描述对象数据库中的模式。和SQL一样，用ODL定义的模式信息存储在系统目录中并在系统运行时使用，以评价查询并执行数据库更新。

1. ODL与阻抗不匹配

在SQL中，数据定义语言（Data Definition Language）的作用非常清楚，它是数据库中描述数据的唯一方法。ODMG中的ODL的作用不太清楚。语言绑定允许程序员直接在宿主语言（例如，Java）中用宿主语言的语法声明数据库对象和类，所以此时就不再需要使用ODL了。这与嵌入式SQL、SQLJ、ODBC或JDBC通过单独的语言（SQL的DDL子集）与宿主语言结合采用调用层或语句层接口来定义数据库模式的机制非常不同。更加复杂的是，很少有ODMG厂商在他们的系统中真正实现了ODL。其中的原因在这里不作讨论。

如果我们回想起完全消除阻抗不匹配是有问题的，那么ODL的角色就变得更清晰了。在人们相信C++是唯一可信赖的编程方法的时代，使用统一语言来编写应用程序和进行数据库访问听起来非常吸引人。对象将在宿主语言中统一声明，与其是否为用来存储在数据库中的持久对象或是在应用结束时将被释放的空间的临时对象无关。然而，Java的出现使事情变得更加复杂。其问题在于Java的对象定义方法不像C++那样详尽。即使在Java出现之前，ODBMS应用开发团体中就有相当多的Smalltalk（也有其自身的限制）的拥护者。所以问题是：所有这些使用不同对象定义方式的语言如何访问同样的对象存储？Java应用程序能操纵C++应用程序生成的对象吗？ODMG的数据模型到底是由C++语言绑定来定义的还是由Java绑定定义的？

SQL数据库中没有上述问题的原因在于它有一个与所有宿主语言分离的数据定义语言。在ODMG中，由于要去除阻抗不匹配，难以要求每一个宿主语言都有同样的ODL。然而，ODL作为指定参照数据模型的方法以及由ODMG定义的三种语言绑定的目标语言还是有用的。所以，虽然ODL不能作为一个软件包，但是通过提供统一的概念基础，它在保持ODMG标准的统一方面起着重要作用。特别是ODMG指明只要采用属于两者共同拥有的ODL的子集进行对象定义，以一种语言编写的对象可以被其他语言编写的应用程序访问。

2. ODL和ODMG数据模型

ODL中使用的一些术语在某些方面与CODM不同，下面着重讨论它们的不同之处。

ODL描述每个对象类型的属性和方法，包括它继承的属性。方法由它的**签名**（signature）

确定，签名包含了方法名、名称、序号、参数的类型、返回值的类型，以及会产生的异常（错误和特殊条件）。

ODL 是CORBA标准中用于指定对象的**接口定义语言**（Interface Definition Language，IDL）的扩展。在Java中，ODL中的类分为两种，第一种叫做接口，第二种叫做类。为了避免与CODM类混淆，将其分别称为**ODMG接口**（ODMG interface）和**ODMG类**（ODMG class）。

按照CODM，ODMG接口和类定义指定以下方面：

- 类名（在CODM意义下）和相关的继承层次结构。
- 类的相关类型。

这些定义的集合指定了全部ODMG数据库模式。ODMG接口和ODMG类之间的主要区别如下：

- ODMG接口不能包括方法的代码；只允许含有方法的签名（这就是为什么称其为接口的原因）。ODMG包含类方法的代码。方法的签名和代码既可以使用宿主语言（如C++或Java）显式地指定，也可以从更高的类层次结构中继承过来。ODMG接口也不指明属性，而ODMG类指明属性。
- ODMG接口不能有它自己的对象成员，即不能创建接口的对象实例。相反，ODMG类可以（通常有）对象成员。所以，ODMG接口的外延包含属于它的ODMG子类的对象。
- ODMG接口不能从ODMG类继承，只能从其他ODMG接口继承。
- ODMG类可以继承多个ODMG接口，但最多只能有一个直接ODMG超类。

区分类和接口的原因有两个。第一，可以使ODL成为CORBA的IDL的超集，因此提供了与分布式系统的这一重要标准的一定程度的兼容性。第二，使ODL避开了一些由多继承引起的问题，例如，当一个类从不同的超类继承了同一方法的不同的定义时就会出现多继承。

与CODM类似，ODMG区分对象与值。在ODMG的术语中，值称为**文字**（literal）。对象如前所述用二元组（oid，val）表示。另一方面，文字可以有复杂的内部状态，但是它们没有oid，以及任何相关方法。在ODL中，对象创建为ODMG类的实例，而文字是由用struct 关键字指定的类型的实例。

类Person的定义如下：

```
// An interface.
// Note: Object is the top interface in ODMG, but a class in Java.
interface PERSONINTERFACE: Object
{    String Name();
     String SSN();
     enum SexType {m,f} Sex();
}
// An ODMG class
// PERSON inherits from PERSONINTERFACE, but has no ODMG
// superclass
class PERSON: PERSONINTERFACE
(    extent PERSONEXT
     keys ( SSN, (Name, PhoneN) ): PERSISTENT; )
// properties of the instances of the type
{    attribute ADDRESS Address;
     attribute Set<String> PhoneN;
// relationships among instances of the type
```

```
        relationship PERSON Spouse;
        relationship Set <PERSON> Child;
    // methods of instances of the type
        void add_phone_number(in String phone);
    }

    // A literal type
    struct ADDRESS
    {   String StNumber;
        String StName;
    }
```

这个例子定义了一个接口、一个类和一个文字类型。PERSONINTERFACE: Object语句说明PERSONINTERFACE是继承层次结构中object的孩子。Object是内建的接口定义，它提供所有对象共有的方法（如delete()、copy()和same_as()）的签名。（一个对象same_as 另一个对象在当且仅当它们有相同oid时成立。）类似地，PERSON: PERSONINTERFACE声明类PERSON从接口PERSONINTERFACE接口继承。在这个例子中，PERSON不是任何ODMG类的子类。

以extent和key开始的语句指定PERSON的类型。extent 给出了所有PERSON对象组成的集合的名字。按照关系DBMS的观点，数据定义语言中的类名和外延名之间的不同相当不寻常。这就像给关系模式赋予一个名字（即用于语句CREATE TABLE语句的符号），而该模式下的实际关系实例被赋予另一个名字。无论按照概念的观点还是实际的观点，外延的值都比较可疑。

key语句声明extent有两个不同的候选键：SSN和<Name,PhoneN>。换句话说，这种类型没有具有相同SSN或相同的姓名和电话号码的不同对象。

规格说明余下的部分定义了与接口PERSONINTERFACE、类PERSON和文字类型ADDRESS相关的类型（CODM意义上）。这些类型的每一个实例都有数个方法和属性。PERSONINTERFACE的方法Name()返回基本类型String。方法Sex()返回枚举类型，其值可能为m或f。类PERSON的Address属性有用户定义类型ADDRESS，即在PERSON定义中给出的文字类型。PhoneN属性为集合类型，集合的每个成员为string类型，用ODMG中的接口set表示。注意，因为接口中只能定义方法，所以例子中Name()、SSN()和Sex()都是方法。

ODMG区分*属性*（attribute）和*联系*（relationship）。**属性**的值是存储在特定对象中的文字类型（不是对象）。**联系**指存储在数据库中的其他对象。它指出对象与特定对象如何相关。PERSON定义包含两个联系。Spouse是对应于特定对象的资助人的另一个PERSON对象（单独存储的）。Child为PERSON对象集合（也是单独存储的），它列出关于特定的人的所有子对象。

但是，ODMG使用的术语“联系”与E-R模型中使用的“联系”相冲突。除了联系涉及对象而非值以外，ODMG的联系与属性非常相似。相反，E-R联系与对象相似并且通过属性和角色来表示自身的结构。事实上，E-R角色的概念对应于ODMG联系的概念，因为角色本质上是一个定义域为实体集合的属性（而不是基本类型，如整型和字符串）。

除了指定属性与联系外，类定义也可以有方法签名（在宿主语言中单独提供了方法的实现）。在我们的例子中，方法add_phone_number()的签名已经指定了。方法的参数列表中的关键字in指定**参数的传递模式**（parameter passing mode）；这里是说参数phone是输入参数。这个特性是从CORBA的接口定义语言（在第10章的SQL/PSM中也可见到）“借来的”。其他的参数传递关键字是out和inout（输出参数和输入输出都可用的参数）。

上面的ADDRESS规格说明定义了一个ODMG文字类型（它的实例是CODM术语中的“值”）。对于这个定义，ODMG使用关键字struct 而不是class。文字只有属性没有联系，所以关键字attribute 和relationship在文字的定义中并不需要。

定义另一个ODMG类STUDENT来继续开发上述例子:

```
class STUDENT extends PERSON {
    (extent STUDENTEXT)
    ...
    attribute Set<String> Major;
    relationship Set<COURSE> Enrolled;
    ...
}
```

子句 STUDENT extends PERSON使STUDENT成为类PERSON的子类。注意，ODMG使用“:”表示从接口继承，关键字extends代表从类继承。从类的定义继承意味着除了属性和方法签名外还继承了方法的实现。此外，STUDENT的外延成为PERSON外延的子集。

ODMG的继承模型与Java的继承模型类似。一个ODMG类只能继承另一个ODMG类，但是可以从多个ODMG接口继承方法签名。然而，ODMG禁止名字重载（name overloading）：具有特定名字的方法不能从超过一个类或接口继承。例如，如果LIBRARIAN定义为FACULTY和STAFF的子类，则FACULTY 和STAFF不能拥有一样的方法名，如weeklyPay。（注意，在LIBRARIAN的例子中，由于不能从多个类继承，所以FACULTY或STAFF或二者都必须是接口。）

STUDENT定义中的联系Enrolled说明学生参加了一组课程，其中COURSE 是一个单独定义（存储）的对象。COURSE的定义是:

```
class COURSE : Object {
    (extent COURSEEXT)
    attribute Integer CrsCode;
    attribute String Department;
    relationship Set<STUDENT> Enrollment;
    ...
}
```

联系Enrollment 建立了一门课程和参加该课的学生间的关联。

这些联系声明自动地强制执行我们在SQL中知道的参照完整性。例如，如果从数据库中删除一个课程对象，则系统也会自动地从由STUDENT对象的Enrolled属性返回的课程集合中删除相应的课程对象。类似的，如果删除一个STUDENT对象，系统也将从参加相应课程的学生集合中删除该对象。

ODMG标准不提供类似SQL中ON DELETE CASCADE语句的机制，对这类删除起作用。然而，可以通过ODMG的异常（exception）机制获得几乎同样的功能（这里就不再讨论了）。

ODL也可以自动维护由联系Enrolled和Enrollment返回的值之间的一致性。如果john是类STUDENT的一个对象，john.Enrolled 包含课程CS532,但是CS532.Enrollment的值不包含john，则数据库是不一致的。为了避免出现这种情况，数据库设计者可以指出Enrollment联系是Enrolled关系的inverse，反之亦然。

```
class STUDENT extends PERSON {
    (extent STUDENTEXT)
    ...
    attribute Set<String> Major;
    relationship Set<COURSE> Enrolled;
            inverse COURSE::Enrollment;
    ...
}                                                              (16.8)
class COURSE : Object {
    ...
    relationship Set<STUDENT> Enrollment;
            inverse STUDENT::Enrolled;
    ...
}
```

这里，语句relationship Set<COURSE> Enrolled; inverse COURSE::Enrollment 说明，如果一个COURSE对象c在s.Enrolled集合中（其中s是一个STUDENT对象）则s必须在集合c.Enrollment中。语句relationship Set<STUDENT>Enrollment;inverse STUDENT::Enrolled 说明对等包含（opposite inclusion）。注意，这种约束的功能比参照完整性更加强大，引用一致性仅能保证没有学生注册一门假的课程，并保证课程花名册（rosters）不包含幻影（phantom）学生。

16.4.2 OQL：ODMG 对象查询语言

尽管对象能通过它们的方法和属性直接进行查询，ODMG还是提供了一个强大的声明性查询语言来访问对象数据库。和在关系数据库中一样，ODMG查询可以交互式地发出或嵌入到应用（程序）中。然而，由于许多厂商不支持交互模式，所以嵌入式查询是实际中最为常用的。嵌入式查询的语法依赖于宿主语言。16.4.5节解释了如何在Java 中实现嵌入式查询。本节讨论独立于上述模式的查询语言本身的内容。

ODMG 对象查询语言（OQL）在许多方面与SQL查询语言的子集类似。例如，查询

```
SELECT DISTINCT S.Address
FROM PERSONEXT S
WHERE S.Name = "Smith"
```

返回所有名为Smith的人的地址集合[⊖]。更准确地说，它返回一个Set<Address>类型的值。集合类型与内建方法使得程序员可以存取集合的元素以获得SQL中游标提供的功能。

回想一下，ODMG清楚地区分类名和类的外延名。在FORM子句中出现的是类的外延名而不是类名。所以上例使用符号PERSONEXT，表示类PERSON的外延（即所有对象的集合）。

在上述查询中，如果关键字DISTINCT 从SELECT语句中略去，那么查询将返回一个Bag<Address>类型的值，地址包（bag）（包类似于集合，但是可以有重复元素）。

方法也能在SELECT语句中调用。使用（16.4）中定义的MOVIE对象示例，可以利用如下查询

```
SELECT M.frameRange(100,1000)
FROM MOVIE M
WHERE M.Name = "The Simpsons"
```

⊖ 注意，不同于SQL，OQL使用双引号表示字符串常量，即表示为“Smith”而不是‘Smith’。

来调用frameRange()方法，该方法返回一组帧的范围对象，每一个对象对应一个名称为“Simpsons”的电影。如果frameRange()已经在继承层次结构中进行了重新定义，那么系统将根据继承规则选择正确的版本。有或没有参数的方法都可以在查询的任何位置使用[⊖]。

调用方法的SELECT 语句可能有副作用。例如，如果PERSON类有一个更新个人电话号码的方法，那么可以写出如下的OQL查询，其中在SELECT子句中调用更新方法:

```
SELECT S.add_phone_number("555-1212")
FROM PERSONEXT S                                    (16.9)
WHERE S.SSN = "123-45-6789"
```

这个查询改变了数据库，但是并不向调用者返回任何信息。

在OQL SELECT语句中调用更新方法的能力模糊了数据操纵语言与数据查询语言的界限。

SELECT 语句的语法和SQL一样，也包含复杂WHERE谓词、联结、聚合、分组、排序等。

1. 路径表达式

路径表达式是OQL的关键部分；它使得查询可以深入到复杂属性以及执行对象的方法。如16.1节所指出的，该技术可以相当大地简化查询的表达。例如，

```
SELECT DISTINCT S.Address.StName
FROM PERSONEXT S
WHERE S.Name = "Smith"
```

返回所有名为Smith的人的地址中的街道名对应的字符串集合（类型为Set<String>的值）。

在SELECT语句中，任何可以使用属性的地方都可以使用联系。例如，下面的查询用联系Spouse代替属性Address。

```
SELECT S.Spouse.Name()
FROM PERSONEXT S
WHERE S.Name = "Smith"
```

返回名为Smith的人（数据库中可能有几个Smith）的相应的资助人的名字（类型为Set<String>的值）。

形式化地说，**路径表达式**（path expression）为如下形式:

$$P.name_1.name_2. \ldots . name_n$$

其中P可以是对象或定义域为对象的变量，并且每一个$name_i$可以是属性名，方法调用（带参数），或联系名。例如，表达式M.frameRange(100,1000).play()就调用了方法。

路径表达式必须具有**类型一致性**（type consistent）。在上面例子中，我们用

```
S.Address.StName
```

返回所有名为Smith的人的地址中的StName。该表达式具有类型一致性，原因如下:

- 变量S引用PERSON类型的对象。
- 类型PERSON具有Address属性。
- 子表达式S.Address返回ADDRESS类型的值。
- 类型ADDRESS具有StName属性。

⊖ 如果方法的结果用于其他一些表达式，那么会存在类型化限制。如X.age()=Y.frameRange(100,1000)为类型错误。

在（16.3）用到了路径表达式

```
P.Child.Child.PhoneN                                    (16.10)
```

该表达式表示对象P的所有孙子的电话号码集合。但是，该路径表达式又有了新的问题。一个人可能有多于一个的子女，所以，这里的问题由子表达式P.Child返回的结果的性质所引起。它是一组对象吗？其中每一个对象都属于PERSON类型，还是属于Set<PERSON>类型的单个集合对象？

在一些面向对象的查询语言中（如XSQL[Kifer et al. 1992]），P.Child 是一组对象。所以在XSQL中，可以连续应用Child，然后应用每一个孩子的PhoneN。最后，得到P的所有孙子的电话号码集合。但是，在OQL中，P.Child 返回一个Set<PERSON>类型的集合对象。由于这个集合对象不是PERSON类型的，所以没有为其定义属性Child。只要涉及OQL，表达式（16.10）就会有类型错误！

OQL引入了一个专门的操作符flatten，其目的是分解集合对象和其他聚合对象（如列表）。例如，flatten(Set<1,2,3>)是一个对象的集合{1,2,3}，而不是单个的集合对象。所以，在XSQL中，表达式（16.10）是正确的。而在OQL中，相应的表达式会比较麻烦：

```
flatten(flatten(P.Child).Child).PhoneN
```

2. OQL中的嵌套查询

和在SQL中一样，一些OQL查询需要嵌套子查询机制。回想一下，在SQL中，嵌套子查询出现在两个地方：

- 出现在FROM子句中，指明元组变量的有效取值范围（例如，查询（6.26））。
- 出现在WHERE子句中，指明复合查询条件（例如，查询（6.22））。

与SQL中的原因相同，嵌套子查询可以出现在OQL的FROM和WHERE子句中。但是，如果需要构造一个集合对象并且作为查询结果返回，那么OQL查询在SELECT子句中也可以有嵌套子查询。显然，这种情况是面向对象查询语言所特有的，因为在关系模型中集合对象不能作为元组的组成部分出现。

为了进行说明，假设需要得到所有学生以及他们所学的计算机科学课程的列表。我们希望输出是一组元组，其中第一个属性为基本类型String，而第二个属性为Set<Course>类型。

```
SELECT struct{name:   S.Name,
              courses: (SELECT E
                        FROM S.Enrolled E
                        WHERE E.Department="CS")                (16.11)
             }
FROM StudentExt S
```

这里，SELECT 语句中的嵌套查询产生一个赋予属性courses的集合。这个嵌套查询的目的是生成一个复杂的嵌套结构，这在SQL中不可能发生（所以除了返回单个标量值的子查询外，SQL中不允许复杂的嵌套查询）。注意我们使用了C语言风格的struct结构告诉OQL，查询的输出将视为复杂值的集合。

3. 聚合和分组

OQL提供常见的聚合函数（如sum、count、avg等）。从第6章中，我们知道聚合主要与分

组一起使用。在SQL中，分组需要一个专门的子句GROUP BY。有趣的是，OQL分组可以使用SELECT 语句中的嵌套子查询完成，不需要专门的分组语句。

具体地说，考虑查询（16.11）。它的输出为元组集合，其中第一个组成部分是学生的姓名，第二个部分是学生参加的课程集。所以，输出已经进行了分组。通过一些小的改动，可以把上面的例子转换为产生一个学生以及每个学生参加的计算机科学课程的总数的列表。结果与使用GROUP BY语句的SQL语句一样。

```
SELECT   name:    S.Name,
         count:   count(SELECT E.CrsCode
                        FROM   S.Enrolled E
                        WHERE  E.Department="CS")
FROM StudentExt S
```

即使GROUP BY 在OQL中并非必须的，但它仍与HAVING语句一起被提供。然而，它的存在并没有增加语言的表达能力或作为语法修饰成分，即语言的表达能力没有变化。在大多数情况下，GROUP BY 并没有简化查询的表述。相反，它可以作为查询优化器的提示（hint），用它可以构建一个较好的查询执行计划。

下面进行一些解释，考虑查询处理器如何执行上述查询。为了有效地回答上述查询，查询处理器必须把课程对象分组，其中每一组代表某个学生参加的课程的列表。然而，查询优化器不太可能自己判断出需要以学生为单位对课程分组。所以可行的计划是扫描StudentExt并对S的每一个值执行子查询。另一方面，优化器会以不同的方法处理下面的查询[⊖]：

```
SELECT S.Name, count: count(E.CrsCode)
FROM StudentExt S, S.Enrolled E
WHERE E.Department = "CS"
GROUP BY S.SSN
```

在这个查询中，FROM语句产生oid对的序列<s,e>，其中e为课程的oid，具有相应oid的学生s选修了该课程。

由于使用了GroupBy语句，优化器现在知道课程必须按选修它的学生分组，所以可以用使结果元组正确分组的方法联结Student和Course类。一种实现方法是扫描类Course的实例，并且对于每一门课程c，按每一个参加课程c的学生的oid进行散列。结果，课程的oid将被放置在桶中，按这种方法同一学生参加的所有课程将在同一桶中。所以，可以通过扫描同一个桶来完成联结计算并统计每一个学生选修的课程数。

16.4.3 ODMG中的事务

事务在初始化之前，必须打开引用事务所访问的数据库的**数据库对象**（database object）。通过一个内建的接口DatabaseFactory实现数据库对象的生成、打开和关闭。所以，调用

```
db1 = DatabaseFactory.new();
```

产生新的数据库对象并返回一个引用db1。这个引用能够用于下面的调用：

```
db1.open(in String database_name);
```

⊖ 这里我们为了兼容SQL使用GROUP BY的所谓的“替换语法”。OQL中的正式的GROUP BY语法比较复杂，但是功能更强。这里就不介绍了。

它将打开名为*database_name*的数据库。语句

```
db1.close();
```

关闭该数据库。一次只能打开一个数据库。

ODMG有一个内建的接口TransactionFactory，它用来产生新的事务对象以访问当前打开的数据库。所以，调用

```
trans1 = TransactionFactory.new();
```

产生一个新的事务对象并返回一个引用trans1。这个引用能够用于下面的调用

```
trans1.begin();
trans1.commit();
trans1.abort();
```

来执行指定操作。ODMG假设软件厂商将提供实现上述接口的类。

16.4.4 ODMG中的对象操纵

对象操纵语言处理对象的产生、删除和更新。然而，ODMG数据库通常不支持单独的数据操纵语言，因为宿主语言提供了该功能。ODMG语言绑定（下节介绍）说明用宿主语言的语法如何完成对象操纵。例如，为了生成新的数据库对象，C++和Java使用new 构造函数。在宿主语言中，删除和修改对象可以通过调用相应类的方法来完成。

另一种更新对象的方法是调用SELECT语句中的方法，如（16.9）所示。这种技术能够用来更新满足某些条件的对象集合。

16.4.5 语言绑定

ODMG的目标之一就是鼓励厂商实现基于ODMG数据模型的商用对象DBMS。和关系数据库系统一样，对象数据库的大部分应用用宿主语言编写。ODMG的一个重要目标是为这些程序提供统一的访问数据库的方法，特别是C++，SMALLTALK，Java。所有这些语言都是面向对象的，并且包含指定和产生类似那些在ODL中的对象的功能。为了让这些语言访问ODMG数据库，为它们各自定义了**语言绑定**（language binding）。语言绑定解决如下一些问题：

- *把ODL对象定义映射到宿主语言固有的语法*　为了更好地理解这个问题，回想一下Java、C++以及其他宿主语言不理解ODL的语法。所以，如何用这些语言来定义ODMG模式呢？一种方法是使用语句级的接口，以便ODL语句能够直接插入到宿主语言程序中。然后预处理器把这些语句转换为对相应的数据库库程序的调用。这是嵌入式SQL采用的方法，如10.2节所讨论的。但是，ODMG采用了不同的方法。不是修改宿主语言，而是在宿主语言中定义一系列的类和接口，表示ODL中相应的概念。DBMS库必须确保对于这些在宿主语言中的类的实例的操作在数据库对象上得到了正确的反映。
- *从宿主语言内访问和查询对象*　访问数据库的方法是把运行时对象（根据宿主语言的固有的机制定义）绑定到数据库对象。然后通过将带有数据库对象的类的方法应用到运行时对象对数据库对象进行查询和修改。访问对象的更有效的方法是向服务器发送OQL查询（即SELECT语句），发送方法在概念上与ODBC和JDBC中的方法类似。ODMG绑定

采用了上述两种方法。

例如，考虑下面Java中的类:

```
public class STUDENT extends PERSON {
    public DSet Major;
    // Plus, possibly, other attributes
}
```

这里，DSet是ODMG定义的对应于ODL接口Set的Java接口。定义DSet的原因是现有的Java接口未提供ODL中的Set的全部功能。要为学生添加一个专业，可以调用STUDENT对象上的固有的Java方法:

```
STUDENT X;
....
X.Major.add("CS");
.....
```

其中add是一个应用于DSet对象的方法。所有语言绑定的主要设计原则是在ODMG对象上使用固有的宿主语言功能（可以稍有变化），从而消除阻抗不匹配。但是，实现标准所需要的许多功能在某些或所有宿主语言中都不存在。结果，ODMG语言绑定缺乏语法（有时为概念上的）的统一性，而且其细节在不同的宿主语言上也有很大的不同。为了说明其中的困难，可以考虑如下的问题:

- 语言绑定如何区分特定类的持久对象和瞬时对象？例如，一个使用数据库中的数据项的应用同时具有持久对象和瞬时对象。持久对象直接绑定到要更新的数据库对象，而瞬时对象并不直接对应于数据库中存储的信息，但是它们由应用逻辑使用（如显示表单，保存计算的中间结果）。

  对于此问题，每一种语言都有自己的解决方法。首先，类必须声明为**可持久化**(persistence capable)，三种语言的实现各不相同。其次，需要自动存储在数据库中的可持久化类的任何对象必须显式地进行持久化。为此，C++绑定使用一个专门形式的new()方法。在Java绑定中，通过定义在接口Database中的专门方法makePersistent()实现对象的持久化，对象持久化的另一种方法是它被已经持久化的对象所引用。此后，由数据库负责确保对象存储在数据库中。
- 绑定如何表示和实现联系？这3种语言在原始形式下都没有实现联系。Java绑定的现在版本不支持联系，C++和Smalltalk绑定通过定义专门的类和方法实现了联系。
- 如何表示ODMG的文字？在C++中，文字用关键字struct表示，但是在Smalltalk和Java中文字不能直接表示，必须映射到对象。
- 如何执行OQL查询？每一种语言都有自己的实现，但是它们基本上都依靠类似JDBC中采用的机制（参见10.5节）：实例化一个查询对象，把OQL查询作为一个参数化的字符串提供给查询对象，通过使用专门的方法可以把查询中的参数实例化为特定的对象。然后，通过利用查询对象的execute()方法执行查询。
- 数据库是如何打开与关闭的？事务是如何执行的？每一种语言都有预先定义的对应于ODL中的database和transaction对象的类，这些类包含执行查询操作的适当的方法。

Java 绑定示例

为了使上述讨论更加具体，我们探讨一下如何用Java绑定执行OQL查询[⊖]。

ODMG宿主语言绑定为查询对象提供两种补充机制：OQL查询（使用OQLQuery类）、由DCollection接口定义的方法。前者从数据库中抽取对象集并赋给Java变量；后者对使用第一种方法从数据库中检索到的对象集（并且存储在Java变量中）进行查询，下面将对两种机制进行解释[⊖]。

为了使用OQLQuery方法，要用OQLQuery类的构造器方法创建一个查询对象。这个类中的一些方法是：

```
class OQLQuery {
    public OQLQuery(String query); // query constructor
    public bind(Object parameter); // supplies arguments
    public Object execute();  // executes queries
    ... ... ...
}
```

OQLQuery构造器接收一个包含OQL查询（一条SELECT语句）的字符串作为输入并生成一个查询对象。OQL查询本身可以用**占位符**（placeholder）（$1, $2等）参数化，它们分别对应动态SQL、JDBC和ODBC中的“?”占位符（第10章）。这些自变量可以使用bind()方法替换为实际的Java对象。当查询对象中的所有占位符都绑定后，可以通过execute()方法执行查询。

例如，下面的程序段计算1999春季只有计算机科学专业的学生参加的课程：

```
DSet students, courses;
String semester;
OQLQuery query1, query2;
query1 = new OQLQuery("SELECT S FROM STUDENT S "
                      + "WHERE \"CS\" IN S.Major");
students = (DSet) query1.execute();
query2 = new OQLQuery("SELECT T FROM COURSE T "
                      + "WHERE T.Enrollment.subsetOf($1) "
                      + "AND T.Semester = $2");
semester = new String("S1999");
query2.bind(students); // bind $1 to the value of students
query2.bind(semester); // bind $2 to the value of semester
courses = (DSet) query2.execute();
```

变量students和courses定义为类型DSet，该类型是对应于ODL中的接口Set的Java接口。从概念上讲，可以想象Java绑定把接口DSet映射到了ODL的接口Set。

接下来，变量students 被赋予了一个集合对象，该集合对象包含代表计算机科学系中学生的所有对象。这个集合对象可以通过先生成一个OQL查询对象，然后调用它的execute()方法得到。

⊖ 预计目前的Java绑定将被前面提到的即将出现的Java数据对象规格说明代替。但是，对于现在的和即将出现的规格说明，从应用程序中对数据库对象进行访问的原则是一样的。

⊖ 注意，这种设计具有相当严重的阻抗不匹配，因为该设计使用单独的语言OQL从数据库中检索对象。我们在前面讨论了这个问题，提到即将出现的Java数据对象规格说明的目的就是为了克服这个问题。

注意，第一个OQL查询的目标列表只含有一个DSet类型的变量。查询返回一个oid集合，把结果赋予一个DSet变量，使之能够存取相应的对象并且能够对它们应用方法。实际上，execute()的签名说明该方法返回一个类Object的成员，所以必须把结果转换为DSet类。

下一步生成另一个查询对象，并将其保存在变量query2中。方法subsetOf()根据集合T.Enrollment是否为作为参数$1的集合的子集而返回值true或false。第二个查询没有完全指定，它是一个查询模板（query template），因为它包含两个占位符。第一个（$1）已经通过调用query2.bind(students)绑定到变量students中的对象上（students 现在表示所有计算机科学专业的学生）。第二个占位符（$2）通过第二个调用bind()绑定到变量semester提供的String对象上[⊖]。至此，查询已全部指定，并且可以使用execute()方法执行查询。作为执行结果的Object在赋予courses前需要转换为DSset类型。

得到了想要的课程集合后，可能想进一步挑选这些对象的一个子集，检查具有特定性质的对象是否存在，或者使用类似游标的机制逐个处理对象。所有这些功能都由DCollection接口提供，该接口是接口DSet的超类。和DSet类似，DCollection也是ODMG Java绑定的一部分。我们给出DCollection接口的部分内容:

```
public interface DCollection extends java.util.Collection {
    public DCollection query(String condition);
    public Object selectElement(String condition);
    public Boolean existsElement(String condition);
    public java.util.Iterator select(String condition);
}
```

这里令人感兴趣的方法是提供了功能强大的生成对象的子集合的query()，以及select()。query()方法类似于关系代数中的选择操作符，但更为一般化。例如，上述方法中的condition变量可以包含量词forall和exits。DCollection 的select()方法生成一个集合，该集合由以参数形式提供的条件和关于该集合的iterator对象指定。iterator是我们熟悉的游标概念的具体实现。Java Iterator接口定义了允许宿主语言逐个处理集合中对象的方法。

回到我们的程序，我们可以获得由第二个查询计算出的courses集合，进一步挑选那些少于3个学分的课程:

```
DSet seminars;
seminars = (DSet) courses.query("this.Credits < 3")
```

其中this是一个变量，其取值范围是query()所作用的集合的所有元素。同样假设类Course拥有属性Credits。当然，新的集合semianrs可能已经通过一个OQL查询直接计算得到。但是，如果需要计算course变量保存的集合中的不同的子集合，则首先计算较大的集合，然后使用DCollection接口进一步查询结果的效率可能会更高。

由接口DCollection指定的其他方法以同样的方式进行工作。例如，selectElement（）选择一些满足条件的集合成员，这些条件作为参数传递。方法exitsElement()测试由condition条件确定的子集合是否为空。

⊖ 注意语句query2.bind()的顺序。

16.5 SQL: 1999中的对象

SQL:1999中的面向对象扩展已经经过了许多次修订。最终的结果是对象-关系模型的一个比较简洁，但相当有限的版本。这是一次困难的标准化处理，因为需要保持向后兼容SQL-92，SQL-92是一种设计时根本没有考虑对象概念的语言。

在这一节中，我们讨论SQL:1999新的对象-关系扩充。但是，在写作本书时，数年前写作的许多参考书都已经过时，而关于新标准的信息并不容易获得。SQL:1999对象模型的一些内容的较好的介绍在[Fuh et al. 1999]给出。许多细节可以从[Gulutzan and Pelzer 1999]得到。

SQL:1999数据库包含一个关系集合。每一个关系既是一个元组集合也是一个对象集合。一个**SQL:1999对象**（object）是一个二元组（o,v），其中o代表oid，v代表SQL:1999的元组值。SQL:1999**元组值**（tuple value）的形式为$[A_1:v_1,\cdots,A_n:v_n]$，其中$A_1,\cdots,A_n$是不同的属性名，每一个v_j取如下的值（使用16.3.1节介绍的术语）：

- *基本值*——普通SQL的基本类型的常量，如CHAR(18)、INTEGER、FLOAT和BOOLEAN。
- *引用值*——对象Id。
- *元组值*——$[A_1:v_1,\cdots,A_n:v_n]$形式，其中A_i是不同的属性名，每一个v_i是一个值。
- *集合值*——仅包含ARRAY结构，这里就不再介绍了。曾经提出的SETOF和LISTOF结构没有包含在标准中，但是可能会出现在以后的版本中。特别是，没有包含集合-值属性极大地限制了SQL:1999对象模型的有效性，取消了一些16.1节讨论的关于对象的优点。

作为对象-关系模式中期望的数据模型，SQL:1999中的每一个对象的顶层值是一个元组。相反，ODMG允许顶层为任何（类型的）值。因为SQL:1999不支持元组内的组成部分为集合值，所以对于多数实际应用，元组构造器是构造复杂值的唯一途径。但是，元组可以嵌套在其他元组中至任意深度。

16.5.1 行类型

构造用于属性规格说明中的元组类型的最简单方法是用ROW**类型构造**（type construct）。例如，可以用如下方法定义关系PERSON:

```
CREATE TABLE PERSON (
   Name CHAR(20),
   Address ROW(Number INTEGER, Street CHAR(20), ZIP CHAR(5)))
```

我们使用通常的路径表达式来引用行类型的组成部分。

```
SELECT P.Name
FROM PERSON P
WHERE P.Address.ZIP = '11794'
```

一个具有行类型的表可以通过ROW**值构造器**（value constructor）装入数据，如下所示：

```
INSERT INTO PERSON(Name, Address)
VALUES ('John Doe', ROW(666, 'Hollow Rd.', '66666'))
```

更新表用行类型也非常简单。

```
UPDATE PERSON
SET Address.ZIP = '12345'
WHERE Address.ZIP = '66666'
```

当John Doe更换地址后，可以用如下方法改变整个地址:

```
UPDATE PERSON
SET Address = ROW(21, 'Main St.', '12345')
WHERE Address = ROW(666, 'Hollow Rd.', '66666')
      AND Name = 'John Doe'
```

16.5.2 用户定义类型

回想16.3.3节中讨论过，类型（CODM中）是一组关于数据结构化的规则。遵循这些规则的对象集合就是这个类型的定义域。一个类有多个模式（包含类型和方法签名）和一个外延，外延是类型的定义域的子集。当为与该类型相关的方法签名添加方法体时，我们就得到了一个**抽象数据类型**（abstract data type）。在SQL:1999中，抽象数据类型称为**用户定义类型**（UDT）。下面给出一些UDT定义的例子:

```
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';
```

除了现在定义的是类型而不是表之外，第一个CREAT TYPE语句类似于前面的表PERSON的定义。这个类型没有任何显式定义的方法，但是很快将会看到DBMS自动为我们生成许多方法。

第二条语句更有趣。它定义STUDENTTYPE作为PERSONTYPE的子类型，PERSONTYPE由子句UNDER指定。这样，它就继承了PERSONTYPE的属性。另外，STUDENTTYPE定义了属于自己的属性和方法award_degree()。类型定义仅包含方法的签名。实际的定义是通过使用CREATE METHOD语句完成的（通过FOR子句与STUDENTTYPE相关联）。该语句说明方法体是用C语言编写的（所以DBMS知道如何与这个程序链接），并且告诉我们在哪可以找到它的可执行体。如果指定LANGUAGE SQL，我们可以在一个附带的BEGIN/ENDC程序块中使用SQL/PSM（存储过程语言（见第10章））定义方法的程序代码。

用户定义类型出现在两种主要的上下文中。首先，它们可以用来指定表中属性的定义域，就像基本类型，如整数型和字符型:

```
CREATE TABLE TRANSCRIPT (
    Student STUDENTTYPE, -- a previously defined UDT
    CrsCode CHAR(6),
    Semester CHAR(6),
    Grade CHAR(1) )
```

(16.12)

这里使用CREATE TABLE语句的唯一的不同之处是属性可以拥有复杂类型（如上面的StudentType）。

其次，UDT能够用来指定整个表的类型，这可以通过使用一种新的CREAT TABLE 语句实现，不是对表的每一列进行枚举。这意味着表中的所有行都必须具有UDT指定的结构。例如，可以定义如下的基于前面定义的UDT StudentType的表:

```
CREATE TABLE Student OF StudentType;                    (16.13)
```

通过CREATE TABLE…OF 语句创建的表（如上所示）称为**类型表**（typed table）。类型表的行是**对象**（object），下面将进行介绍。

16.5.3 对象

在SQL:1999中生成对象的唯一方法是在类型表中插入一行。换句话说，这样的表中的每一行都被作为一个对象，拥有自己的oid。表自身则被视为一个类（如CODM中所定义的），它的行的集合对应于类的外延。

把如下的声明与（16.13）对比一下是有启发的:

```
CREATE TABLE Student1 (
    Name CHAR(20),
    Address ROW(Number INTEGER, Street CHAR(20), ZIP CHAR(5)),
    Id INTEGER,
    Status CHAR(2) )
```

注意，其中Student1包含的属性与Student完全一样（名字和类型也一样）。但是，Student是一个类型表而Student1不是，这意味着SQL:1999把Student中的元组（而不是Student1中的元组）视为对象。这种不一致（甚至可以称为矛盾）的两种创建表的方法是为了向后兼容SQL-92。还要注意，在（16.12）和（16.13）中使用StudentType的不同。StudentType的实例在前者中不是对象，而在后者中则是对象。

下一个问题是如何获得一个对象。为了理解这个问题，让我们回到（16.12）的Transcript表，考虑其中的属性student。因为同一学生可能参加几门课程，所以他在Transcript中就有几条记录（元组）。问题在于声明

```
Student StudentType
```

的意思是这个属性返回StudentType值，而不是引用StudentType对象。所以，同一学生的信息（姓名、地址等）必须重复存储在该生在Transcript中的每一条记录中。显然，这同我们在第8章要消除的冗余一样。SQL:1999通过引入显式的**引用类型**（reference type）解决了这个问题，我们将在16.5.6节更深入地讨论这个问题。现在，我们只要记住引用类型的定义域是oid集合就行了。为了引用SQL:1999中的对象，需要得到它的oid，所以我们必须考察一下为此提供的解决机制。

SQL:1999标准说明每一个类型表（如（16.3））都有一个**自引用列**（self-referencing column）用来保存元组的oid。当元组创建时，oid会自动生成。但是，为了存取保存在自引用列中的oid，必须显式地给该列一个名字。上述StudentType的声明中没有为自引用列命名，所以在该表中无法引用元组的oid。

下面是考虑自引用列的一个方法:

```
CREATE TABLE STUDENT2 OF STUDENTTYPE
REF IS stud_oid;
```
(16.14)

REF 子句给出自引用列一个显式的名字stud_oid。(注意，这个列也存在于（16.13）中，但是是隐藏的，所以不能被引用。）在大多数的使用方式中，stud_oid和其他属性一样。特别地，可以把它用于查询（SELECT和WHERE 子句都可以），但是我们不能改变它的值，因为oid是系统赋予的，不可改变。

SQL:1999通过区分对象和它的引用使用C和C++方法。这种区分在ODMG和纯面向对象语言（如Java）中是不存在的，并被领域中的许多研究者认为是一个倒退。

16.5.4 查询用户定义类型

查询UDT并未产生任何新的问题。我们只要使用路径表达式在对象中向下抽取需要的信息即可。例如，

```
SELECT T.Student.Name, T.Grade
FROM TRANSCRIPT T
WHERE T.Student.Address.Street = 'Hollow Rd.'
```
(16.15)

查询TRANSCRIPT关系，并返回居住在Hollow Road的学生的姓名和分数。注意，T.Student返回类型STUDENTTYPE的复杂值以及从PERSONTYPE继承允许我们存取为它定义的Name 和Address属性。

还要注意，尽管STUDENT和STUDENT1具有不同的结构，但与一个学生相关的具体查询在两种情形下都一样。所以，

```
SELECT S.Address.Street
FROM X S
WHERE S.Id = '111111111'
```

返回具有Id 111111111的学生所居住的街道名而不管X是STUDENT还是STUDENT1。

16.5.5 更新用户定义类型

我们已经讨论了UDT的数据定义问题，现在介绍向基于UDT的关系增添数据的问题。我们已经看到（在16.5.1节中）如何向PERSON表中插入元组。类似地，我们可以使用同样的方法向表STUDENT和STUDENT2中插入元组。这些关系包含对象（以及额外的自引用属性）都不是问题，因为oid是系统创建的。必须担心的仅仅是那些实际属性。我们可以尝试着用类似INSERT的语句向关系TRANSCRIPT添加元组:

```
INSERT INTO TRANSCRIPT(Student, Course, Semester, Grade)
VALUES (????, 'CS308', '2000', 'A')
```

但是，VALUES子句的第一部分是什么呢？为了回答这个问题，可以参阅16.5.6节。

UDT被**封装**（encapsulated）的事实，即它的组成部分只能通过类型提供的方法进行存取，使得问题变得更加复杂。虽然我们没有为STUDENTTYPE定义任何方法，但数据库为我们进行了定义。也就是说，对每一个属性，系统提供一个observer方法（可以用来查询属性的值）和一个mutator方法（用来改变属性的值）。observer和mutator都有和属性同样的名字。在

STUDENTTYPE的例子中，系统提供如下observer方法：

- Id: ()→INTEGER。这个方法返回一个整数，并且同所有observer一样，它不接受任何参数[⊖]。
- Name和Status。两种方法都有type()→CHAR(*length*)，其中*length*是一个整数。
- Address: ()→ROW(INTEGER，CHAR((20)，CHAR(5))。

但明显，查询（16.15）使用了observer方法Name和Address。另一方面，在同一查询中使用的表TRANSCRIPT的Grade属性不是UDT的一部分，所以它没有observer方法。但是，这种区别是概念上的而不是语法上的，因为从语法意义上说，引用Grade的方法与引用Name的方法是一样的。

Mutator方法都是在与STUDENTTYPE对象有关的情形下被调用的，并返回同样的对象。但是，它们拥有不同的参数，返回的对象在状态上发生了如下的变化：

- Id: INTEGER→STUDENTTYPE。这个方法接受一个整数并用它替换对象的Id值。换句话说，mutator方法改变了学生的Id并且返回修改后的对象。
- Name: CHAR(20)→STUDENTTYPE。这个方法接受一个字符串并替换学生对象中的Name属性的值。它返回更新后的学生对象。Status的mutator也是一样。
- Address: ROW（INTEGER，CHAR（20），CHAR（5））→STUDENTTYPE。该方法接受一个表示地址的行并且用它替换学生的地址。

注意，SQL不使用C++和Java的pubic和private描述符来控制方法的存取。相反，它使用EXECUTE的权限以及4.3节介绍的常用的GRANT/REVOKE机制来控制存取。

现在准备第一次向UDT中插入元组。

```
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')
```
(16.16)

这里应该注意两件事。在插入的元组的第一部分通过调用StudentType()生成了一个空白学生对象，其中StudentType()是DBMS为每一个UDT生成的构造器。然后，新生成的对象上的mutator方法被逐个地调用以便进行数据填充[⊜]。

如果要更改学生的地址、姓名和课程的分数，可以用下面的更新语句：

```
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'
```

⊖ 符号（）表示该方法无参数。

⊜ 注意，上面的语法只是NEW StudentType().Id(……).Status(……).Name(……).Address(……)的缩进的易读形式。

我们使用DBMS生成的STUDENTTYPE的mutator函数来更新学生对象的值。首先，用Address() mutator更新地址，然后应用Name ( ) mutator。

你一定已经注意到向涉及UDT的关系中插入新元组是相当繁琐的。但是，把方法关联到复杂数据类型的功能可以在某种程度上使上述问题得到简化。也就是说，我们可以定义一个只接受标量值的特殊的构造器方法。这样，一个复杂对象可以通过调用一次该构造器来产生。

我们用SQL存储过程的语言来解释上述想法。首先需要把下面的声明加入到STUDENTTYPE[⊖]的早期定义中：

```
METHOD StudentConstr(name CHAR(20), id INTEGER,
                     streetNumber INTEGER,
                     streetName CHAR(20),
                     zip CHAR(5), status CHAR(2))
RETURNS StudentType;
```

然后，定义方法主体，方法如下：

```
CREATE METHOD StudentConstr(name CHAR(20), id INTEGER,
                            streetNumber INTEGER,
                            streetName CHAR(20),
                            zip CHAR(5), status CHAR(2))
RETURNS StudentType
LANGUAGE SQL
    BEGIN
        RETURN NEW StudentType()
                .Name(name)
                .Id(id)
                .Status(status)
                .Address(ROW(streetNumber,streetName,zip));
    END;
```

有了新的构造器，向对应于（16.16）的TRANSCRIPT关系中插入新元组就变得不太繁琐了。

```
INSERT INTO Transcript(Student, Course, Semester, Grade)
VALUES (StudentConstr('Vlad Dracula', 666666666, 666,
                      'Transylvania Ave.', '66666', 'G5'),
        'HIS666',
        'F1462',
        'D')
```

16.5.6 引用类型

在关系TRANSCRIPT的模式（16.12）中，属性Student具有STUDENTTYPE类型。16.5.3节解释过，这阻碍了学生对象的共享，因为每个学生的记录都物理存储在对应的脚本记录中。为了能够共享对象，SQL:1999使用**引用数据类型**（reference data type）。引用是一个oid，REF(*udt-xyz*)形式的类型的域包含*udt-xyz*类型对象的所有的oid。可以用以下方式重写（16.12）的TRANSCRIPT的定义：

```
CREATE TABLE Transcript1 (
        Student REF(StudentType) SCOPE Student2,
        CrsCode CHAR(6),
        Semester CHAR(6),
        Grade CHAR(1) )
```
(16.17)

⊖ 类似于属性，方法可以用SQL中的ALTER语句进行添加。

我们需要对Student属性做更多的解释。首先，类型REF(STUDENTTYPE)的意思是Student的值必须是一个类型为STUDENTTYPE的对象的oid。然而，我们可以创建许多不同类型的表并把它们与STUDENTTYPE相关联，每一张表都可以包含各种类型的学生。我们可能不希望Student指向任意的学生。相反，我们需要某种参照完整性来保证这个属性指向特定表所描述的学生。SCOPE子句恰好起到了这样的作用：限定Student的值不是任意一个STUDENTTYPE类型对象的对象号，而是属于表STUDENT2[⊖]中一个已存在的对象。注意，(16.14) 中定义STUDENT时，选择STUDENT2不是偶然的。

新的特征通常带来特征的混淆。对于 (16.17) 中定义的表TRANSCIPT1，(16.15) 中的查询可以写为下面的形式。这个查询中包含一个引用类型的属性— STUDENT:

```
SELECT T.Student->Name, T.Grade
FROM TRANSCRIPT1 T
WHERE T.Student->Address.Street = 'Hollow Rd.'
```

注意，我们使用→来指向STUDENTTYPE对象的属性。这是因为T.Student返回的是一个学生对象的oid，而不是对象本身。由于历史原因，用“.”和用“→”表示的引用在C/C++ 中是不同的。而在面向对象语言中没有必要对这两者进行区别，所以在OQL和Java中不存在两种不同的符号[⊜]。

在C和C++中，选择“.”或“→”的原则是一样的。如果一个属性是引用类型，则在路径表达式中使用“→”；如果是对象类型，则使用“.”。在我们的例子中，T. Student 有一个引用类型REF(STUDENTTYPE)，所以用T. Student→Address来指向student object里面。相反，查询 (16.15) 使用T. Student.Address，因为这里T.Student的类型是STUDENTTYPE，它不是一个引用类型。

另外一个重要的问题是怎样填充表TRANSCRIPT1。在16.5.5节中我们看到了怎样把元组插入到复杂类型中的例子。但是，那些例子并不处理对象引用。为了向TRANSCRIPT1中插入一个元组，必须找到一个方法来存取Student对象的oid，然后把它们赋给Student属性。这就是第16.5.3节中所说的自引用列。回想一下该节定义的表STUDENT2，它有一个由子句REF IS[⊝]定义的自引用列stud_oid。同样，在 (16.13) 中定义的表STUDENT与STUDENT2有相同的类型，只是它的自引用列是隐藏的 (因为我们没有给它命名)。这样就没办法得到STUDENT表中的元组的oid，并把它赋给TRANSCRIPT1表中的STUDENT属性了。正是因为这个原因，在定义TRANSCRIPT1时，应使用STUDENT2而不是STUDENT。

假设STUDENT2中的Id属性是一个键，我们能够向TRANSCRIPT1中插入一个student，方法如下:

```
INSERT INTO TRANSCRIPT1(Student, Course, Semester, Grade)
SELECT S.stud_oid, 'HIS666', 'F1462', 'D'
FROM STUDENT2 S
WHERE S.Id = '666666666'
```

⊖ 为了使得这些要求保持一致，SCOPE中提到的关系必须和REF中提到的类型相关联，就像STUDENT2通过CREATE TABLE...OF语句与STUDENTTYPE相关联一样。

⊜ OQL中允许使用“→”，但这只是用点表示的的另一种写法。

⊝ 注意，stud_oid不使用属性Id，它是STUDENTTYPE的一个常规属性，该属性是由程序员显式赋值的。

注意，我们用一个SELECT语句来检索所有想要的Student对象的oid。这个oid成为Student属性的值。而属性Course、Semester和Grade的值是直接添加到目标列表中去的。

16.5.7 集合类型

最有用的集合类型是set和list，目前已经被标准忽略。如果有了这些类型，数据建模会变得更加自然，更接近于ODMG中的各种可能情况。例如，可以使用Enrolled属性扩展STUDENTTYPE而不必重新创建一个新的TRANSCRIPTS表，方法如下[⊖]：

```
CREATE TYPE StudentType UNDER PersonType AS (
    Id INTEGER,
    Status CHAR(2)
    Enrolled SETOF(REF(CourseType)) SCOPE Course)
METHOD StudentConstr(name CHAR(20), id INTEGER,
                     streetNumber INTEGER,
                     streetName CHAR(20),
                     zip CHAR(5), status CHAR(2))
RETURNS StudentType
METHOD award_degree() RETURNS BOOLEAN;
```

在这个例子中，Enrolled属性的值必须是关系COURSE中元组的一组oid。

通过使用上面的collection类型- set，可以写出一种新的查询。下面的查询列出每一个学生的Id、居住的街道和所选课程。

```
SELECT S.Id, S.Address.Name, C.Name
FROM Student S, Course C
WHERE C.CrsCode IN
        (SELECT E -> CrsCode
         FROM S.Enrolled E)
```

这里，WHERE子句测试STUDENT和COURSE的笛卡儿积的每一个元素。条件使用一个嵌套的SELECT语句来产生笛卡儿积中某一行所描述的学生所学的课程代码的集合。

嵌套SELECT语句的FROM子句展示了一个新的特征。变量E的范围是由路径表达式S.Enrolled给定的一个集合。这个集合里面每一个元素都是对学生S所参与的一门课所对应的COURSE对象的引用。这和SQL-92中利用FROM子句使用查询结果是类似的，只是这里使用路径表达式，而不是查询来返回对象引用的集合。

因为E的范围是对象引用，所以必须通过“→”操作符来访问每一个对象的属性，它将间接引用E并产生COURSE中的一个特定对象。

16.6 公共对象请求代理体系结构

CORBA是为客户端访问驻留在服务器上的对象的环境所设计。客户端能够获得这样的服务的一种方法是利用**远程过程调用**（Remote Procedure Call，RPC）[Birrell and Nelson 1984]，22.4.1节将对RPC进行讨论。客户端进程执行过程调用，使过程在服务器进程（可能是另一台机器）中被执行。

对象管理组（Object Management Group，OMG）提出了一个新的中间件标准：公共对象

⊖ 为了使这个例子更加简单，我们在新的设计中忽略成绩存储在哪里的问题。

请求代理体系结构（Common Object Request Broker Architecture，CORBA）。与RPC类似，CORBA能够让客户端访问驻留在服务器上的对象，而且它更加灵活，更加一般化。不过，它并不是一个RPC的替代品或类似的机制。事实上，CORBA通常是在RPC之上实现的。

CORBA与RPC的另一个相似之处是它提供分布式计算资源的**位置透明性**（location transparency）。这意味着客户端访问资源时是与位置无关的，资源位置的变化不会影响客户端。在RPC中，远程资源被指定成不相关的过程的集合，而CORBA把资源看成是包含一组相关操作的对象。它同时为客户端应用提供发现和使用远程服务的机制，即使这些服务在编写应用时还不存在或没有设计出来。

在CORBA中包含一个叫做CORBAservices的层，它为提供持久性、查询和事务服务提供基础设施。持久性、查询和事务服务都是本节所要讨论的重点。CORBA已成为制造、电子商务、银行和医疗等领域的各种应用框架的平台。这些框架是CORBAfacilities层的一部分。本节不讨论这一层，详细信息可见 [Pope 1998]。

16.6.1 CORBA基础

每一个服务器使用**接口定义语言**（Interface Description Language，IDL）来定义驻留在它上面的对象的接口。IDL是ODMG的对象定义语言（Object Definition Language，ODL，参见16.4.1节）的一个子集。和ODL一样，IDL用于指定类和方法的签名。然而，IDL类没有扩展，它们被称为“接口”（和ODL术语保持一致）。IDL里也没有约束和集合类型（比如Set），而ODL中则有存在。

为了解释这个概念，考虑一个公用图书馆的服务器，它提供对图书馆藏书进行检索的一个接口。客户端应用可以使用这些搜索机制来为用户提供服务。

```
/* File: Library.idl */
module Library {
  interface myTownLibrary {
    string searchByKeywords(in string keywords);
    string searchByAuthorTitle(in string author, in string title);
  }
}
```

接口通常根据用途组成各个模块。使用模块的一个极大的好处是它能避免因为开发接口的组织或部门不同所造成的命名冲突。因为模块名称总是作为方法名称的前缀。这样，客户端对searchByKeywords () 方法的引用可以写做:

```
Library_myTownLibrary_searchByKeywords(...)
```

客户端应用的请求是怎样导致服务器代码执行的呢？这是通过对象请求代理完成的，下面将进行详细讨论。

1. 对象请求代理

CORBA体系结构中的一个新的组件是**对象请求代理**（Object Request Broker，ORB），它处于客户端和各个服务器之间，如图16-3所示。当客户端根据IDL的描述执行一个方法调用，调用就会传给ORB。ORB负责定位包含那个对象的服务器，然后在客户端方法调用和服务器类定义所需的方法调用之间做必要的转化。也就是说，ORB将IDL语言映射到实现对象的语

言。CORBA标准定义IDL可以映射到C、C++、Java、Cobol、Smalltalk和Ada。

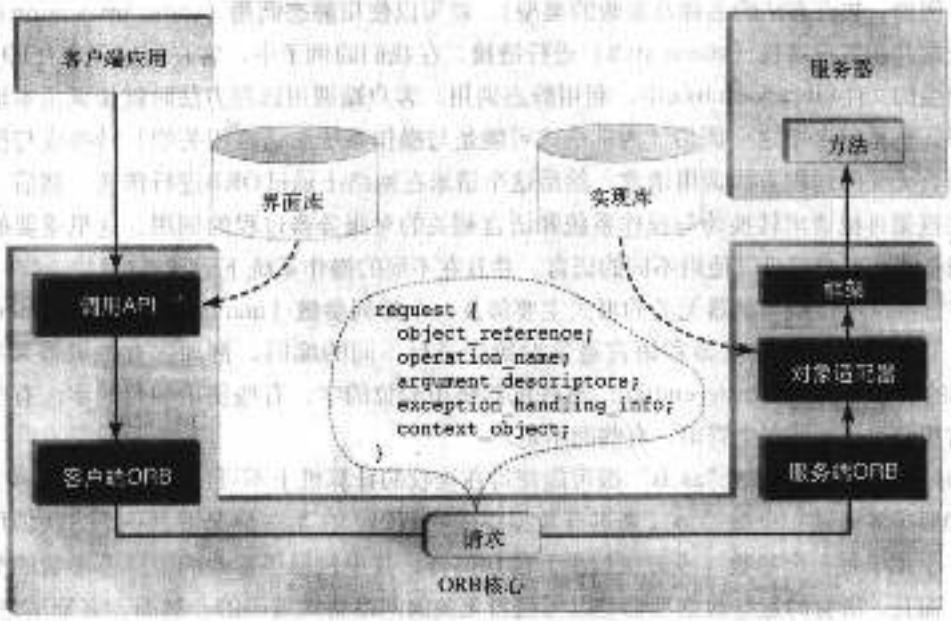


图16-3 CORBA体系结构

远程调用服务器对象的详细的过程如下所示[⊖]。在部署服务器的时候，IDL定义、Library.idl被CORBA厂商提供的IDL编译器所编译。编译后产生一对文件：Library-stubs.c和Library-skeleton.c。而且，接口定义的方法签名以及相关的IDL信息存储在接口仓库（interface repository）中。

文件Library-skeleton.c中包含一个服务器框架。这部分代码将与操作系统和语言无关的客户端请求映射成具体的（操作系统、语言相关）对图书馆服务器上的searchByKeywords() 和searchByAuthorTitle() 方法的调用。框架在服务器部署前就被编译并与服务器链接。当服务器启动时，它将在**对象适配器**（object adapter）中注册自己。对象适配器是驻留在服务器上的ORB的一部分。通过注册，服务器通知ORB，它可以用接口仓库中所描述的特定接口中的特定方法处理调用[⊜]。有意思的是，可以注册不同的实现来处理同一个方法调用（在同一个接口中），由对象适配器来选择一个特定的实现。例如，如果图书馆的目录是分布式的，对象适配器通过使用一个搜索目录的本地高速缓存副本的实现来满足对图书馆顾客的请求。同样，图书馆工作人员的查询也可以用执行分布式搜索的实现来完成。对象适配器可以根据查询中所包含的客户端的**上下文对象**（context object）来决定使用哪一种实现[⊛]。ORB维护**实现仓库**（implementation repository）来跟踪服务器上不同方法的可用实现。

⊖ 我们描述整体的过程，某些方面可能与特定的CORBA实现有关。

⊜ 原则上讲，同一个接口的不同方法可以由不同的服务器处理。相似的，不同接口的方法也可以由同一个服务器来执行。

⊛ CORBA定义一个接口Context来提供为任意性质设置名/值对的方法。Context对象包含客户端所定义的所有的名/值对。服务器可以使用Context接口来检查Context对象的可用属性，并进行相应的操作。这样，客户端就可以使用Context对象来给服务器提供有关请求的元信息。

在客户端，可以通过以下几种方式进行方法调用。如果客户端应用知道怎样调用服务器方法（例如，知道方法的名称及参数的类型），就可以使用**静态调用**（static invocation）来编译客户端并和**客户端桩**（client stub）进行链接。在我们的例子中，客户端桩保存在IDL编译器所产生的文件Library-stubs.c中。利用静态调用，客户端调用远程方法时就像调用本地子例程一样。桩里包含把这个调用（内部格式可能是与操作系统和语言相关的）转换成与操作系统和语言无关的远程方法调用请求，然后这个请求在网络上通过ORB进行传送。然后，服务器上的框架再把请求转换为与操作系统和语言相关的对服务器过程的调用。这里重要的一点是，服务器和客户端可以使用不同的语言，并且在不同的操作系统下运行。

将方法调用转换为机器无关的形式主要涉及一个**排列参数**（marshaing the argument）的过程。这是因为不同的机器和语言通常对数据进行不同的编码。例如，有些机器采用big-endian的表示，有些是little-endian；有些机器使用32位的字，有些使用64位的字；有些计算机语言用空字节来结束字符串，有些则不是。

如你所见，发送数据“as is”很可能使得在接收的计算机上不可使用。在CORBA中，客户端和服务器通过一个握手协议来进行数据编码和解码。在某些情况下，一个方法调用请求包含一个用于每一个参数（或方法结果）的解码器，其中包括了参数的值以及参数的类型和长度。而且，所有的这些数据项都是以与机器无关的网络格式编码的。然而，必须清楚的是，程序员并不需要亲自解决这个转换问题。相反，它是由桩调用（间接）的CORBA库例程来完成的。CORBA中的一个方法调用请求的整个结构如图16-3所示。

在有些情况下，客户端不知道怎样调用服务器上的方法。这乍听起来好像不可能；然而有很多实际发生的例子，或许还很有用。让我们重新考虑那个图书馆搜索应用，不过这次假设它要给用户提供对不同的图书馆的不同目录进行搜索的能力。一种可能的情况是强制所有的图书馆服务器使用同样的接口myTownLibrary，但这显然是不现实的，因为不同的图书馆可能有不同的遗留系统，而且变更需要的花费很大。而且，不同的图书馆可能提供不同的搜索能力，例如，有些图书馆可能会提供利用模式及通配符进行搜索的功能。

如果检索涉及的图书馆一直保持不变，我们可以把不同的接口编码在客户端应用里，从而解决这个问题。但是，我们希望我们的应用能够允许新的图书馆加入或已有的图书馆退出，而且新加入的图书馆也能被我们的应用正常搜索。例如，如果 yourTownLibrary加入了系统，我们可以将Library IDL 模块改写为：

```
/* File: Library.idl */
module Library {
  interface myTownLibrary {
    string searchByKeywords(in string keywords);
    string searchByAuthorTitle(in string author, in string title);
  }
  interface yourTownLibrary {
    void searchByTitle(in string title, out string result);
    void searchByWildcard(in string wildcard, out string result);
  }
}
```

在上面的yourTownLibrary接口中，不仅仅是方法名称和调用顺序不同，结果也是以输出参数而不是以函数结果的形式返回的。在编译完Library.idl并且将框架与yourTownLibrary的服务器

链接后，就能让客户端搜索应用搜索两个图书馆了。

为了达到这些目标，我们将客户端应用设计成让用户填写一个基本信息的表格。表格中包含书籍的ISBN、作者、关键词及通配符。然后客户端应用根据参数名分析所有在Library.idl中定义的接口，创建适合服务器的调用。例如，如果用户填入了作者、关键词、通配符等信息，客户端应用就可以选择接口myTownLibrary中的searchByKeywords ()和接口yourTownLibrary中的searchBywildcard()。如果用户输入了一个标题和一个通配符，可能就不会选择myTownLibrary（因为它没有正确的参数），而是选择yourTownLibrary中的searchByTitle ()。为了实现这个调用策略，我们可以要求成员数据库在写接口的IDL描述时从一个固定的词汇表里选择参数名。这个要求不一定会给遗留数据库带来变化，而仅仅是在接口设计中加入一些原则。

怎样实现这个灵活的方法调用策略呢？这时就要引入接口仓库。客户端应用可以使用CORBA提供的一个特殊的**动态调用API**（dynamic invocation API）而不是桩去查询接口仓库。这个API允许应用决定模块Library中的所有接口，每一个接口中定义的方法，每一个方法的参数名及类型以及方法的返回值的类型。根据参数名，搜索应用可以决定应该调用哪些方法及提供哪些参数。它通过使用API调用CORBA_Object_create_request()生成适当的请求对象。请求对象包含被调用的方法名，其参数的名字和类型，以及返回结果的类型。一个完全创建的请求可以作为一个CORBA API子例程CORBA_Request_invoke()的参数，来执行对服务器上方法的实际调用。这个请求采用与机器相关的格式，并且由上面的API子例程将请求通过网络传输到ORB的服务器端。然后，ORB用服务器skeleton将请求转换为对服务器方法的具体调用。

值得注意的是，静态调用实际上执行的是为ORB创建请求的动作。然而，因为对服务器方法的调用序列是预先知道的，所以需要的操作序列是由IDL编译器自动生成的，并组成了客户端桩的核心部分。

在对客户端方法调用的讨论中，忽略了一点，即客户端要求服务器执行操作的实际对象。客户端给ORB的请求中包含一个请求对象的**对象引用**（object reference）。客户端是怎样得到这个引用的呢？

一种方法是让客户端和服务器的设计者遵循某些协议。例如，服务器可能会将对象引用作为字符串在网络上经常访问的地点发布。在我们的例子中，对参加对象的所有引用都可以在网络上的某些协议文档里发布。然后可以通过API调用CORBA_ORB_string_to_object ()把这些字符串格式的引用转换为CORBA中使用的内部二进制格式。

一个更加方便的获得对象引用的方法是使用**命名服务**，我们将在下一节中进行讨论。

2. CORBA中的互操作性

如果世界上只有一个ORB对象，那么每一个对象都可以和其他对象进行交谈。然而事实情况是，不同的组织和公司喜欢部署自己的ORB，其中绑定了实现它们的业务过程的对象。如果一组企业在一个项目中进行合作，它们必须为那个项目重新部署一个ORB对象。于是就存在怎样和不同的ORB所控制的对象进行通信的问题。我们知道，一个ORB对象可以向自己控制的对象传递请求，然而，如果一个对象属于另外一个ORB，就需要利用协议来传送请求。

幸运的是，CORBA对这个问题有一般性的回答——**通用ORB间协议**（General Inter-ORB Protocol，GIOP）。GIOP实质上是一个特殊格式的消息，它使得一个ORB可以向另一个ORB

所控制的对象发送请求，并接受回答。因为这些消息必须在实际的网络中进行传输，所以必须把它们映射为已有网络中的消息格式。IIOP（Internet inter-ORB Protocol，因特网ORB间协议）就是这样一个协议，它指定怎样将GIOP消息翻译成TCP/IP消息以便通过Internet传送，如图16-4所示。

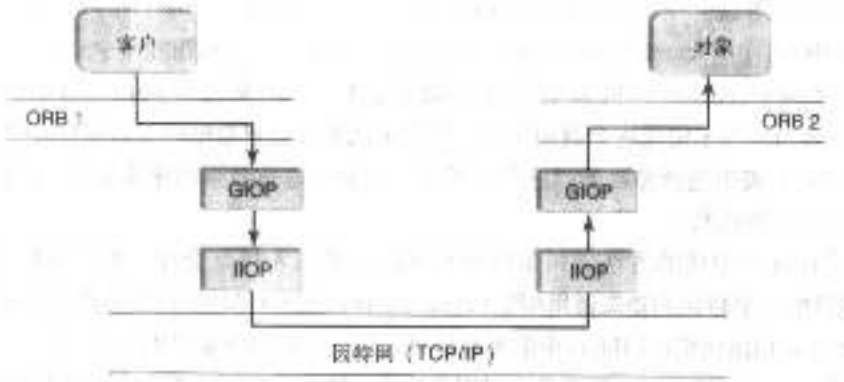


图16-4　ORB间体系结构

对于网络上其他流行的协议也有相似的翻译协议，比如点对点协议（Point-to-Point Protocol），它经常在使用modem的情况下用于连接到网络。对于一些中间件标准，如分布式计算环境（Distributed Computing Environment，DCE）也有相似的翻译协议。

16.6.2　CORBA和数据库

CORBA的作用已经超出使用它的最初目的，并迅速成为分布式计算的一个无处不在的标准。最初，CORBA的目的是允许客户端程序访问驻留在服务器上的对象，然而，因为对象是持久性的，所以就有可能将这些对象的集合作为数据库来使用。为了实现数据库系统的全部功能，除了最基本的服务，还需要再提供一些附加的CORBA服务。这些服务现在集成在一个体系结构层CORBAservices中，它位于基本的ORB机制之上。通过使用CORBAservices，我们就可以使用ORB对象来建造一个功能性数据库，它包含了分布在Internet或其他网络上的异质对象。

从概念上讲，CORBAservices是一组提供特定服务的API，其中包括事件通知服务、许可证服务（控制使用特定的对象，或对使用这些对象的用户进行收费）、生命周期服务（允许拷贝、移动和删除对象）。在数据库上下文中常用的服务有：

- 持久性服务（persistence service）
- 命名服务（naming service）
- 对象查询服务（object query service）
- 事务服务（transaction service）
- 并发控制服务（concurrency control service）

CORBA通过提供这些服务，使得不同位置服务器上的一组对象能够形成一个数据库，并提供满足全部ACID事务性质的访问[9]。

[9] 没有必要提供一个显式的“对象更新服务”，因为对象只有在客户调用它们的方法时才更新状态。这样的调用是ORB提供的，它位于CORBA服务层的下面。

下面是一个使用持久性对象的CORBA应用的例子。一些公司要联合开发一个新的产品（所谓的虚拟公司）。每一个公司都将自己所负责的设计部分（图表、部件列表、文档等等）作为持久对象存放在本地机器上。这些对象可能是用不同语言，在不同的操作系统下开发的。通过使用ORB，一个公司的工程师可以访问其他公司所设计的对象，并可以将它们集成到本地的文档中。而且，工程师可以使用名字访问某个设计对象，而不需要知道它究竟存储在哪个公司的机器上。查询服务允许用户对整个虚拟企业的对象集进行查询，ORB使得用户能够调用对象的方法。而事务和并发控制服务使得编写要求事务属性的应用成为可能。

1. 持久性服务

CORBA提供一个让应用与提供可用IDL接口的远程对象进行交互的机制。但是，对于数据存储（例如ODMG数据库）来说缺少一种标准方式来向它的数据对象导出接口并允许CORBA客户端对它们进行操纵。而且，也没有一个标准规定CORBA客户端怎样在数据存储中创建和删除一个对象。当然，客户端可以在CORBA之外，例如，用ODBC与数据库进行交互，但是这违背了CORBA为构建分布式应用提供一致性的基于对象的接口的目标。而CORBA**持久性状态服务**（PSS）恰好解决了这个问题。

CORBA持久性状态服务的整体架构如图16-5所示。它的基本思想是服务器端的持久性对象被组织成storage home，然后再将storage home的集合组成数据存储（data store）。数据存储可以是提供非易变存储的任意形式，如数据库或文件系统。

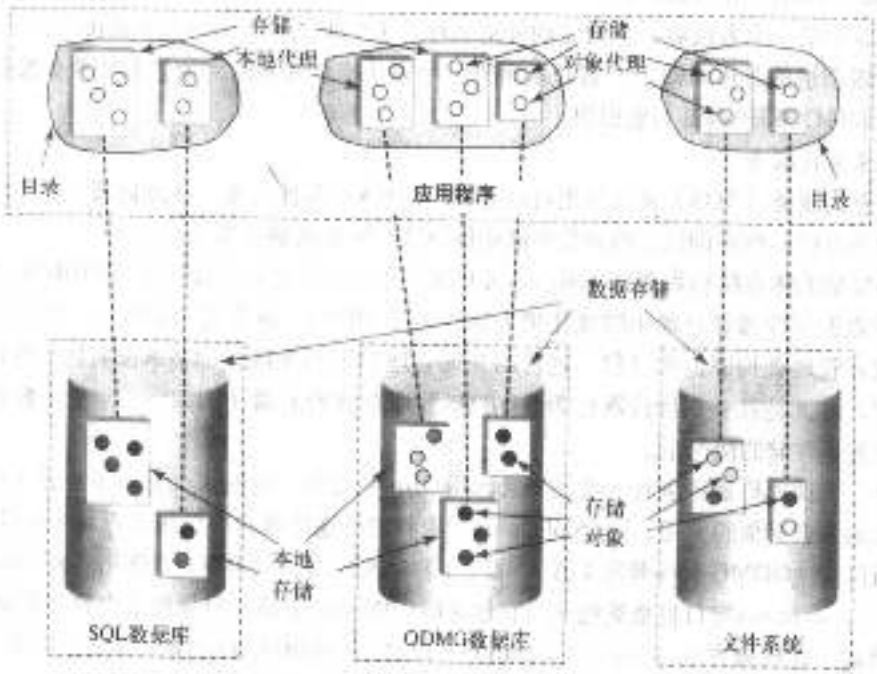


图16-5 CORBA持久性状态服务的体系结构

在应用端，程序可以同时使用多个数据存储，如图16-5所示。应用通过一个特殊的方法调用来连接到存储，从而开始对一个数据存储的使用。根据编程语言的不同，可以应用不同的机制来访问storage home。在面向对象的语言（如Java或C++）中，一个storage home对应一

个公共类。为了访问一个storage home，应用必须有一个与storage home名称相同的公共类，称为**storage home 代理**[⊖]。为了创建storage home，程序员必须提供一个包含访问方法（签名及代码）的类定义。然后这个类需要在服务器上编译并在PSS上注册。

将storage home 代理聚合起来的对象就像服务器上的存储对象的代理。客户端应用直接使用宿主语言通过调用这些代理的方法来操纵它们。这些方法可能直接访问存储对象的高速缓存拷贝，或者和服务器通信。PSS必须保证代理的变更能够反映在实际的存储对象中。

很容易看出，PSS的设计受到ODMG体系结构及其减少或消除阻抗不匹配的目标的影响。事实上，通过操纵代理，会让客户端程序感觉是在直接操纵持久性的CORBA对象，就好像它们是本地对象一样，它不需要使用特殊的API来发送和更新对远程对象的请求。

2. 命名服务

命名服务使用良好定义的命名机制为程序引用（持久）对象提供了方便的途径。这个机制与操作系统中命名文件的方式相似。CORBA的命名空间是一个有向无圈图，其中每一条边用一个简单名字来标注。对象的名称由图中从根到该对象的路径上的所有名称连接起来后组成。路径不同，一个对象可以有多于一个的全名。

在图书馆例子中，除了用前面讲的home-grown协议来导出对象引用外，服务器可以决定使用通过CORBA接口NamingContext定义的ORB的命名机制。这两个重要的方法是：

- void blind (in Name n, in Object o);
- Object resolve (in Name n);

服务器用第一个方法将一个特定的外部名称[⊜]（比如/Library/myTownLibrary/search）绑定到服务器创建的具体对象上。客户端用第二个方法将已发布的对象名称作为参数传递给方法，然后取得CORBA内部对象引用。

3. 对象查询服务

对象查询服务（OQS）使得应用可以查询CORBA持久性对象。OQS依赖于ODMG和SQL标准，给不支持这些查询语言的对象存储提供OQL和SQL风格的查询功能。

OQS完整的体系结构如图16-6所示。客户端应用将查询传递给作用类似ODBC的查询评价器。然后查询评价器经过最小的变化把查询传递给DBMS，或者它可能把一个查询分成一系列的请求并管理查询评价的过程。这样的查询管理对于不提供查询请求所有特征的DBMS就非常必要，尤其是在查询评价器作为非DBMS数据存储的前端（例如：一组电子数据表格），或有多个数据存储的情况下。

另外，查询评价器还创建一个类型为collection的对象，它的成员是对所有属于该查询请求结果的服务器对象的引用。这个collection对象将被传递给客户端应用进行后续处理。

前面讨论过ODMG Java绑定（第16.4.5节）和SQLJ（第10.5.9节）中使用的collection接口。CORBA 的collection接口也是类似的。它包含向collection中插入对象和从中删除对象的方法，另外还增加了**迭代器方法**（iterator method），以提供类似游标的功能，并允许应用一次处理collection中的一个对象。

注意，尽管表示查询结果的collection对象通常是由查询评价器在客户端创建，但

⊖ 在PSS的当前版本中，这个类叫做storage home实例。我们不使用这个易混淆的术语，因为在面向对象中，实例是属于某个类的一个对象，而不是代表另一个类的类。

⊜ Name数据结构是在CORBA中定义的，它包含了一个已发布的对象的字符串名。

collection中的成员对象还是保存在它们的数据存储中。当客户端调用查询结果对象的方法时，ORB将这些请求传送给服务器上的成员对象。这样的服务器端的处理在CORBA中是很常见的。在这种情况下，ORB只是作为对象间的一个智能通信通道，而对象是保留在服务器上的。

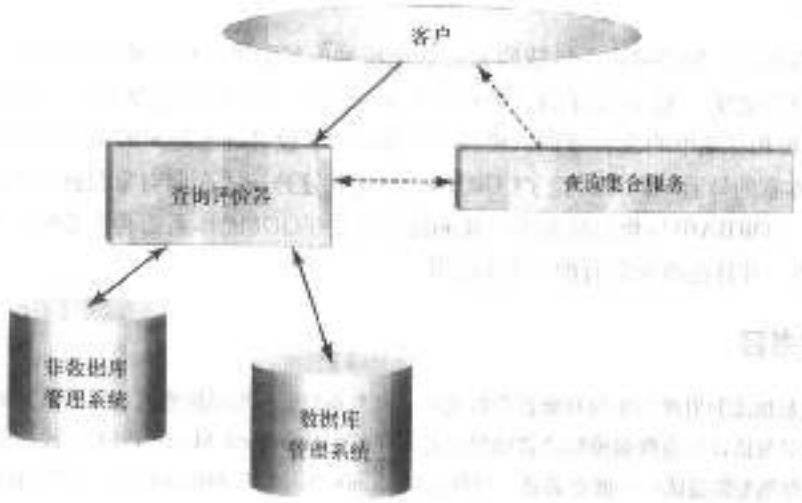


图16-6 对象查询服务的体系结构

4. 事务和并发控制服务

15.3.1节介绍了事务管理器（transaction manager）模块，它组织并协调一个事务的运行（该事务可能访问多个分布在不同地理位置的数据库），并且保证事务是原子的（22.3.1节中会详细讲述这部分内容）。CORBA通过**事务服务**（也称作对象事务服务，object transaction service或 OTS）来提供相似的功能。事务服务适用于访问一组不同的持久性对象或数据库的事务，如图16-5所示。

事务服务允许应用将一个执行线程转变为事务。一个线程可以在本线程的事务上下文被创建后，调用一个特殊的事务对象上的begin ()方法，事务服务将在整个事务运行期间保存此上下文。线程还可以调用commit () 和rollback ()方法来执行相应的事务操作。

被事务修改了的CORBA对象必须被声明为transactional和recoverable，这意味着它们可正确地响应commit和rollback命令。Transactional和recoverable对象必须驻留在transactional和recoverable的服务器上，以提供实现commit和rollback命令所需要的登陆和其他服务。

CORBA也提供了**并发控制服务**（concurrency control service），它允许应用对CORBA对象加锁和解锁，并执行两阶段锁协议（参见第15.1.2节），这样就保证了事务的隔离性。注意，CORBA级的锁是独立于DBMS中的锁，并在其之上的。

事务服务使用两阶段提交协议（参见第15.3.1节）来保证整个事务的全局原子性，该事务可以访问多个对象，包括存储在数据库里的持久性对象（参见图16-5）。所有用这种方式访问的对象必须支持包含commit和rollback数据库的命令的X/Open标准API（参见第22.3.1节）。

事务服务不但实现了通常的平坦事务模型（第21.1节），而且实现了嵌套事务模型（第21.2.3节）。

注意，CORBA只支持并发和事务语义，但并不强制执行，也就是说，它并不阻止非事务

的CORBA应用访问对象，也不阻止不使用CORBA的数据访问对象。这样，为了保证CORBA对象语义具备ACID性质，所有的应用必须都使用事务和并发控制服务。

16.7 小结

在本章的开头，我们讨论了传统的关系数据模型的局限性，以及面向对象的数据模型是怎样解决这些问题的。接下来介绍了面向对象数据库的通用概念数据模型，给出了不同的ODBMS的实现和标准中的公共原则。我们还具体讨论了设计理念迥异的两个标准：ODMG和SQL:1999。本章的最后一部分讨论了CORBA，它是构建分布式面向对象数据存储的一个新兴的基础结构。CORBA中与数据相关的功能的设计受到了ODMG体系结构的影响，但是它还处在不断发展中，并且在将来会有细节上的变化。

16.8 参考书目

面向对象数据库的出现与面向对象语言的流行，对关系数据模型局限性的认识等因素密不可分。使用现有的面向对象语言作为数据操纵语言的想法首先在[Copeland and Maier 1984]中被提出。一种早期的扩充关系数据模型的尝试——嵌套关系，出现在[Makinouchi 1977, Arisawa et al. 1983, Roth and Korth 1987, Jaeschke and Schek 1982, Ozsoyoglu and Yuan 1985, Mok et al. 1996]中。

POSTGRES[Stonebreaker and Kemnitz 1991]是用抽象数据类型扩充关系数据库的早期代表。现在它叫做PostgreSQL，是一个功能强大的、公开源码的对象-关系DBMS[PostgreSQL 2000]。而对ODMG数据模型及其查询语言具有很大影响的对象数据库O_2在[Bancilhon et al. 1990]中有详细的描述。[Cattell and Barry 2000]中介绍了ODMG标准的最新版本。[Alagic 1999]中讨论了ODMG标准设计中要考虑的若干问题（及一些可用的解决方案）。

16.3节中提到的概念数据模型及相关问题在[Abiteboul et al. 1995]中有更详细的讨论。而面向对象查询语言的逻辑基础的发展可以参考[Kifer et al. 1995, Abiteboul and Kanellakis 1998]。

用路径表达式查询对象式的结构的方式最早出现在GEM系统中[Zaniolo 1983]。路径表达式后来用于所有主要的对象查询语言中，包括ODL和各种SQL的面向对象的扩展，如[Kifer et al. 1992]中所讨论的XSQL。

早期支持对象-关系数据模型的数据库有UniSQL、POSTGRES和O_2。现在，大多数主流的关系数据库厂商（如Oracle、Informix和IBM），都提供它们产品的对象-关系扩展。这些系统背后的很多设计理念都基于SQL:1999标准。更多关于SQL:1999对象关系扩展的内容可以参考[Gulutzan and Pelzer 1999]。

由于SQL:1999对象扩展还是新生事物，现在还没有产品完全支持这一标准。IBM的DB/2在语法和支持特性方面是最接近这一标准的。

[Pope 1998]是CORBA的一个带有C语言示例的全面介绍。Java和C++程序员可以参考[Orfali and Harkey 1998, Henning and Vinoski 1999]来详细了解CORBA。

本章略去了有关面向对象数据库设计的讨论，而关系数据库相对应的内容在第5章和第8章进行过介绍。这部分没有介绍的内容可以作为附加阅读材料。逐渐流行的面向对象数据库设计方法是**统一建模语言**（Unified Modeling Language，UML）[Booch et al. 1999, Fowler and Scott 1999]。

UML类似于E-R建模，但是它在一些方面扩展了E-R模型，尤其是通过提供方法来规定数据库对象的外部行为（如，对此类对象的公共方法应该有什么要求）。然而，E-R建模具有简单性的优点，已经发

展了许多技术使之适应于面向对象数据库设计[Biskup et at. 1996, Biskup et al. 1996b, Gogola et al. 1993, Missaoui et al. 1995]。

尽管面向对象的E-R建模已经发展得比较完善，但相应的标准化理论看起来比关系模式下的理论更为复杂。这个理论的产生可以参考[Weddell 1992, Ito and Weddell 1994, Biskup and Polle 2000b, Biskup and Polle 2000a]。

16.9 练习

16.1 请给出有关学生注册系统的例子:

a. 方便地使用名/值对属性。

b. 方便地表达两个对象间的关系（用ODMG风格）。

c. 方便地使用继承。

16.2 请给出适合学生注册系统的一组类。

16.3 给出一个实际的例子，说明一个对象可以直接存储在对象数据库（比如ODMG）中，但是不能存储在对象-关系数据库中。

16.4 第16.4.1节中有对PERSON对象及其联系Spouse的ODL描述。

a. 怎样用ODL来表达一个人（可能）有资助人?

b. 给出返回某个人的资助人名字的OQL查询。

16.5 解释对象数据库中的对象的oid和关系数据库中元组的主键的区别。

16.6 给出对象数据库中的对象被认为相同的不同意义。

16.7 考虑银行系统中的ACCOUNT类和TRANSACTIONACTIVITY类。

a. 为它们设计ODMG ODL类定义。其中，ACCOUNT类必须包含与TRANSACTIONACTIVITY中一组对象的联系，这组对象对应该账户的存取事务。

b. 给出一个满足描述的对象实例的例子。

c. 给出一个对该数据库的OQL查询: 它返回至少取过$10 000的所有账户的账户号码。

16.8 一个关系数据库有一个叫做ACCOUNTS的表，其中的每一个账户的元组可能支持存储过程deposit () 和withdraw ()。一个对象数据库有一个叫做ACCOUNTS的类，其中每一个账户对应这个类的一个对象，并且包含deposit () 和 withdraw ()两个方法。解释这两种方法的优点和缺点。

16.9 假设前面例子里的对象数据库的ACCOUNT类有子类SAVINGSACCOUNTS和CHECKINGACCOUNTS，并且CHECKINGACCOUNTS有子类ECONOMYCHECKINGACCOUNTS。解释继承语义怎样影响对每一类对象的检索（例如，当检索满足某条件的所有checking account对象时，哪些类需要被访问)。

16.10 解释一个set对象和一个由对象所组成的set的区别。

16.11 a. 解释ODMG属性和联系之间的区别。

b. 解释ODMG联系和E-R联系之间的区别。

16.12 解释路径表达式的类型一致性的概念。

16.13 在16.4.1节的PERSON定义中，加入恰当的SPOUSE联系的反关系。

16.14 给出一个OQL查询: 返回16.4.1节定义的PERSON中SSN为123-45-6789的人的所有的孙子孙女的资助人的姓名。

16.15 假设PERSON类有另一个属性 age。编写OQL查询产生每一个年龄的人数。要求使用2种方法: 使用GROUP BY子句，及不用GROUP BY而用嵌套SELECT语句。描述每一情况下的查询的分析策略并解释哪种查询会运行得更快。

16.16　写一个OQL查询来产生每个专业的学生人数。使用（16.8）中定义的STUDENT类。

16.17　使用SQL:1999（如果需要，使用SET OF结构）来完成16.5节中部分定义的数据库模式。必须包含下面的UDT:STUDENT、COURSE、PROFESSOR、TEACHING和TRANSCRIPT。

16.18　使用上题定义的模式回答下面的查询:

a. 找出在数学系选课超过5门的所有学生。

b. 用一个UDT GRADE代表成绩，并且包含返回成绩数值的方法value ()。

c. 编写一个方法为每一个学生计算平均成绩。这个方法要求使用前面所编写的value ()方法。

16.19　使用SQL:1999及附加的SETOF结构表示一个银行数据库，要求包含账户、客户和事务的UDT。

16.20　使用SQL:1999和前面所创建的模式回答下面的查询:

a. 找出住址邮编为12345的客户的账号。

b. 找出账户余额超过$1 000 000的所有客户。

16.21　E-R图可以用来设计对象数据库的类定义。给出图5-1、图5-5及图5-12的ODMG类定义的E-R图。

第17章 XML和Web数据

Web的出现为信息技术开启了一个全新的领域，也为现有的数据库框架带来了新的挑战。和传统数据库不同的是，Web上的数据一般不遵循任何广为使用的结构（如关系模式或对象模式）。因此，仅仅使用传统数据库的存储和操作技术是不足以解决Web数据的存储和操作的。鉴于这个原因，我们需要扩充现有的数据库技术以支持教育、商务和政务等领域中基于Web的电子信息的传送和交换。

17.1 半结构化数据

初看起来，Web数据和传统数据库中的数据没有相似之处。然而，根据它的一些特征，我们也可以使用传统数据库技术和信息检索技术来处理Web数据。Web数据的第一个特征就是，它们大多以某种结构化的形式来表示。如图17-1所示，HTML（超文本标记语言，Hypertext Markup Language）使用树状的层次结构来显示学生信息列表。在这棵树中，不同的HTML标记对应着不同的数据元素。

```
<html>
  <head><Title>Student List</Title></head>
  <body>
      <h1>ListName: Students</h1>
      <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>
```

图17-1 HTML描述的学生信息列表

从表面上看（虽然机器不会这样识别数据），HTML页面上的信息是结构化的学生列表，它可以用16.3节介绍的概念对象数据模型（CODM）来表示（我们可以在理解图17-1的基础上

将其转换为图17-2所示的模式)。在这个模型中，实际对象出现在图的上半部分，和第16章一样，对象被表示为一个Oid-Value对。学生列表对应的模式并未显式地出现在HTML中，但是却符合图17-2下半部分所显示的模式。

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] ]
           ]
```

图17-2　学生信息列表的对象形式

这里，我们发现了一个有趣的事实，那就是，尽管模式（包括属性名）没有随数据发送过来，但我们可以根据数据本身的特征合理地推断出数据遵循的数据结构。这是因为，Web网页的设计者意识到数据结构应该易于理解，所以构造出能够**自描述**（self-describing）的数据。数据的这种自描述性是通过在数据域中包含属性名来实现的。

现在，假设同样的信息通过Web传送给机器（而不是人）进行处理。与人不同，机器不太可能对所接受到数据的结构进行智能化的推测。而且，数据模式也未必有良好的定义，例如，在列表中，有的学生可能具有额外的属性（如电话号码），或者某个属性不能确定，如住址（可以使用邮箱号码来取代街道地址）。因此，为了方便计算机之间的信息交换，遵循统一的格式，通过在数据中区分属性名和值来实现数据的自描述非常有意义。

总之，用于机器处理的Web数据可能具有以下特征:

- 对象特征（object-like），即可以使用16.3节描述的CODM模型表示的对象集合来表示Web数据。
- 无模式（schemaless），即与16.3节讨论的对象不同，Web数据并不保证遵循任何确定的类型结构。
- 自描述性（self-describing）。

具备上述特征的数据称为**半结构化**（semistructured）数据。其中的自描述性特征可能会造成某种程度的误导，因为它可能被误解为：数据的含义伴随着数据本身。事实上，半结构化数据仅仅具有属性的名称，而且，与传统数据库中的数据相比，组织化的程度不高。特别地，因为缺乏模式，所以不能保证所有的对象都具有一样的属性，也不能保证不同对象的同名属性具有同样的含义。

根据观察，图17-2不足以完整地表达图17-1中的数据。原因在于，CODM的对象表示法和模式表示法并不是用来表示自描述数据。然而，通过引进CODM的对象表示法与模式表示法中的元素，可以开发出一种合适的表示方法。利用新的表示法，我们的学生列表可以表示为

如下的无模式但具备自描述性的形式:

```
(#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)

上面的描述能恰如其分地表示半结构化数据，而且它还很好地遵循了数据库的习惯。然而，这并不是Web数据交换所选用的格式。Web数据交换选用的是可扩展标记语言(Extensible Markup Language，XML)——1998年被W3C(World Wide Web Consortium)采纳的一套标准。

自问世以来，XML被越来越多的人认同和接受，并逐步成为人类世界和机器世界中信息表示的标准格式。下一节我们将介绍XML语言的各个组成部分并给出相关的实例。

尽管在本质上，XML数据是无模式的，但是符合一定模式的数据通常有更大的用处。特别是，根据电子数据交换的需要，数据在传输时对数据格式严格性的要求超出了半结构化数据所提供的严格性。为此，XML提供一个说明文档结构的可选机制。我们讨论两种这样的机制: **文档类型定义语言**(Document Type Definition，DTD)，它是XML标准的一个组成部分。另一种是XML Schema，它是最近出现的一种建立在XML基础上的规格说明。在本章的最后，我们还将介绍三种XML查询语言: XPath(一种轻量级的查询语言)、XSLT(文档翻译语言)，以及XQuery(完整的XML查询语言)。想进一步研究半结构化数据的读者，建议参考[Abiteboul et al.2000]以及本书参考文献部分包含的其他文章。

17.2 XML概述

XML不能解决世界上所有的问题，也不是一个革命性的思想。实际上，它甚至不是一个新想法，那么，为什么它正在引起一场革命呢? 简单地说，XML是人和机器都能读懂的数据格式，它易于解析，所以简化了数据交换过程。过去，人们也提出过几种数据交换格式，但它们要么是非开放的专有标准，要么不够优秀，不足以吸引大家为发展它而努力。而XML却在合适的时间出现在了合适的地方。人们已经看到，通信、教育和商业领域中Web和开放标准的作用，同时也明确了简化软件代理之间的数据交换的要求。一个值得信任的标准组织，W3C(不隶属于任何团体和政府)也帮助了该标准的推广。这是第一次出现了一个简单的、开放的、被广泛接受的数据标准，它对于Web应用起到了极大的推动作用。

XML是一种类似于HTML的语言，它支持任意数量的用户自定义标记(tag)，而且这些标记事先并没有确定的语义。为了更好地理解这句话意思，让我们再来看看HTML，在这种文档格式中，我们使用标记来标明各个文本块，这些标记将影响到文本在浏览器中的显示。重要的一点是，HTML中标记的数量被HTML定义所固定，每一个标记都具有明确的定义。例如，标记<table>和</table>之间的文本将被浏览器显示为一张表，而标记<p>则告诉浏览器开始一个新的段落。和HTML不同的是，XML没有预定的标记指令系统，用户可以自由地引入

新的标记名，而且，任意一个XML标记都没有预定的语义。

XML缺乏语义似乎是一个倒退。浏览器怎样才能知道如何处理接收到的文件呢？答案在于，应该在XML文档之上建立不同的语义层，并且可以为不同类型的应用进行定制。浏览器显示只是其中的一种。用于浏览器显示的文档的语义用样式表（stylesheet）表示（参见17.4.2节）。然而，大部分XML文档都由应用程序中进行交换，这些应用程序并不会显示在屏幕上供人类阅读。这时，语义由应用领域而定。（因此，同一个应用领域可以共享同一个语义系统。）整个行业就可以开发出不同的语义层，以表示目录信息、商业信息、工程信息以及其他信息。然而，这些工作都需要有定义数据模式的能力。我们将在17.2.4节和17.3节中讨论XML中可用的结构化机制，但必须注意，尽管定义了模式，但XML数据本身仍然是半结构化的。对模式的遵循仍然是可选的，应用程序也可以自由地忽略部分或全部数据模式定义。

举个例子，让我们来看看图17-3的文档，它是图17-1中学生信息列表的一种XML表示方法。第一行是强制性语句，告诉程序（这种程序被称为**XML处理器**）接收文档，而且规定该文档的版本为XML1.0版本。其他部分除了以下两点（这两点非常重要）之外，和HTML文档有着相同的结构:

```
<?xml version="1.0" ?>
<PersonList Type="Student" Date="2000-12-12">
    <Title Value="Student List"/>
    <Contents>
        <Person>
            <Name>John Doe</Name>
            <Id>111111111</Id>
            <Address>
                <Number>123</Number>
                <Street>Main St</Street>
            </Address>
        </Person>
        <Person>
            <Name>Joe Public</Name>
            <Id>666666666</Id>
            <Address>
                <Number>666</Number>
                <Street>Hollow Rd</Street>
            </Address>
        </Person>
    </Contents>
</PersonList>
```

图17-3　学生信息列表的XML表示

- XML文档作者可以使用各种不同的标记构造XML文档。而在HTML中却不可以这样做，只有在HTML官方规格说明中明确定义的标记才是正确的标记，其他标记浏览器均不识别。
- XML中每一个起始标记都要有一个结束标记和它匹配，而且这些标记的嵌套顺序必须正确（即，<a>顺序就是不正确的）。而一些HTML的标记并不要求有匹配的结束标记（如<p>），即使遗漏了结束标记，浏览器也能正确地处理HTML文档。

• XML文档必须有一个**根元素**（root element）。根元素包含其他所有的元素。在图17-3中，根元素是PersonList.

任意一个形如<sometag>.....</sometag>且嵌套正确的文本块均是一个**XML元素**（XML element），其中sometag是元素的**名字**。起始标记（<sometag>）和结束标记（</sometag>）之间的文本叫做元素的**内容**（content）。包含在A元素中的元素是A元素的**子元素**（children），例如在上例中，Name、Id和Address是Person的子元素，而Person是Contents的子元素，Contents又是Personlist的子元素。相应地，我们说PersonList是Contents和Title的**父元素**(parent)，Contents是Person的父元素。

XML也定义了元素之间的**祖先**（ancestor）/**后代**（descendant）关系，这对于XML的查询来说是很重要的，我们将在17.4节向大家介绍这部分内容。这种祖先/后代关系和人类家族中的祖先/后代关系一样，祖先是父亲、祖父或曾祖父等等，后代是儿子、孙子、曾孙等等。例如，PersonList是Person和Address的祖先，而Address是PersonList的后代。

起始标记可以有**属性**（attribute）。如图17-3所示，<PersonList Type = "Student">标记中的Type就是元素PersonList的一个属性名，Student是属性的值。和HTML不同，所有的属性值必须用“”括起来，但标记之间的字符串不需要用“”括起来。再看另一个例子，<Title Value = "Student List"/>说明Title包含了一个属性Value。和一般的表示方法不一样，Title元素没有对应的结束标记，而是直接放入到<.../>中。由于不含有其他内容，所以称其为**空元素**(empty element)。在XML中，上面的声明方式是<Title Value=“Student List” > </Title>组合的简写。

除了元素和属性外，XML还允许使用**处理指令**（processing instruction）和**注释**(comments)。处理指令是如下形式的语句

```
<?my-command go bring coffee?>
```

文档作者可以使用处理指令告诉XML处理器该怎样处理文档。处理指令很少使用，在17.4.2节中，与XML样式表结合时用到了一个处理指令。

注释的一般形式如下:

```
<!-注释内容-->
```

除了在**标记**中，即在开始标记（<）和结束标记（>）之间以及类似的地方不能使用注释外，任何位置均可以使用注释。注释是XML文档的一个不可缺少的组成部分——文档的撰写者在传送之前不会将它删除，接收者可以浏览注释以获得一定的信息。尽管这种处理注释的方法和当前流行的编程语言以及数据库语言中处理注释的方法不一样，但文档处理中并不鲜见。例如，JavaScript程序经常被放在HTML文档中担任注释的角色，HTML处理器不一定忽略它们。相反，除非JavaScript特性被关闭，否则处理器将会执行注释中发现的JavaScript程序。

XML的另一个需要简单说明的特征就是CDATA构造。CDATA允许字符串包含标记元素，这些标记元素可能会导致不正确的文档格式。举个例子，如果我们使用XML写一个关于Web发布的结构化的指南，可能会用到下列文本:

```
Web浏览器试图纠正发布者诸如标记嵌套错误等错误。例如<b><i>Attention!</b></i>
这样的错误，大部分的浏览器都能正确地处理它。
```

然而，这样的字符串因为会导致不正常的文档结构而被所有遵循XML标准的浏览器所拒绝。一个方便的、能够正确包含这样的串的方法如下:

```
<! [CDATA[ Web浏览器试图纠正发布中诸如标记嵌套错误等错误。例如
<b><i>Attention!</b></i>这样的错误，大部分的浏览器都能正确地处理它。]]>
```

最后，每一篇XML文档都可以有一个可选的**文档类型定义**（Document Type Definition，DTD）文件。DTD文件决定XML文档的结构。我们将在17.4.2节讨论DTD。

17.2.1 XML元素和数据库对象

现在，我们对图17-3中的XML文档作为Web上传送半结构化数据的格式进行一些评价。显然，元素名可以有效地用作对象的属性名（XML的属性名也有同样的作用）。所以，该文档本质上是（17.1）中自描述对象的另一种等价的文字描述。

1. XML元素到对象的转换

思考一下，就会发现，XML的嵌套标记结构非常适合表示树结构的自描述对象。XML文档的每一个元素可以看作一个对象。其子元素的标记名对应于对象的属性，子元素自身则是该属性的值。例如，图17-3中的第一个Person元素就可以部分地转换为如下对象:

```
{#6543, Name: "John Doe",
    Id: "111111111",
    Address: <Address>
                <Number>123</Number>
                <Street>Main St</Street>
             </Address>
}
```

转换过程具有递归性。像Name和Id这样简单的元素可以通过抽取它们的内容直接进行转换。元素Address暂时不变，因为它具有复杂的内部结构，这些内部结构可以通过递归地调用相同的转换进程来进一步分解。这样会产生一个新的地址对象:

```
{#098686, Number: "123",
          Street: "Main St"
}
```

2. XML元素和对象之间的区别

尽管XML元素和结构化数据库对象之间存在明显的对应关系，但它们之间还是存在着根本性的区别。首先，XML是由SGML[SGM1986]发展而来，而且深受SGML的影响，而SGML是一种文档标记语言而不是数据库语言。举个例子，XML允许下列形式的文档:

```
<Address>
    Sally lives on
    <Street>Main St</Street>
    house number
    <Number>123</Number>
    in the beautiful Anytown, USA.
</Address>
```

该文档既有数据结构又有文本结构，这在XML中是允许的，但这种文档不便于自动化处理，而且这种风格的文档是真正的数据库设计者所不愿意使用的。

其次，XML元素具有有序性，而数据库对象的属性却不具备这种性质。因此，下面两个对象可以看作是同一个：

```
{#098686, Number: "123",          {#098686, Street: "Main St",
          Street: "Main St"                   Number: "123"
}                                 }
```

而下面两个XML文档却是不同的：

```
<Address>                         <Address>
    <Number>123</Number>              <Street>Main St</Street>
    <Street>Main St</Street>          <Number>123</Number>
</Address>                        </Address>
```

再次，XML只有一种基本数据类型——字符型，而且规定文档约束的机制并不强。幸运的是，这些缺点中的大部分可以通过17.3节的XML Schema规范来解决。

17.2.2 XML属性

在图17-3中，我们已经看到了Type和Value这样的XML属性的用法了。一个元素可以有任意数量的用户自定义属性。然而，回忆一下前文所举的例子中XML元素的表达能力，我们陷入了对作为数据表示工具的XML属性所扮演的角色的困惑之中，更确切地说，它们对于数据表示是否有用呢？它们能表达元素不能表达的内容吗？

答案是，XML属性有时更便于表示数据，但属性所能做到的，元素都也可以做到。而且，利用新出现的基于XML的标准，如XML Schema（将在17.3节中介绍），我们期望用属性来表示数据的情况逐渐消失。但是，属性还是广泛应用于基于XML的规范，如XML Schema和XSLT（我们将在本章后面的内容中介绍他们）中。我们的例子中也大量地使用了属性，这是为了阐明XML的各种特性，而且使用属性能更简洁地表示数据。

在文档的处理中，属性经常用于解释起始标记和结束标记之间带有值（该值不是文本的一部分，但和文本内容紧密相关）的文本，在下面的程序中：

```
<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>
```

我们使用属性来解释那些本质上不是程序内容但又和程序内容相关的“元”信息。本例中，

属性的使用非常方便，这是因为，这里的属性并不影响数据流。另一方面，在数据处理中，文本流不太受关注，因为目前的计算机技术不太可能在不久的将来重视这种风格的文档。这里我们关注的是用XML属性描述另外一个数据表示中的非必要的，但却是数据库程序员所忧虑的方面。

另外，属性值只能是字符串，这一点严格地限制了属性的使用，相比较而言，元素可以有子元素，从而使之呈现出多样性。

对属性进行了这些坦率的评价后，我们应该提及属性的一些优点。首先，元素中多个属性之间的顺序对文档不产生影响，因此，文档<thing price="2" color="yellow" >foobar</thing>和<thing color="yellow" price="2" >foobar</thing>完全一样，如同它们在数据库中一样。其次，同一属性只能出现一次（即<thing price="2" price="2" >是错误的）。然而，正如我们在图17-3中看到的那样，具有同样标记的元素可以重复出现。再次，使用属性可以将数据表达得更为简洁。例如，<thing price="2" color="yellow" />比<thing><price>2</price><color>yellow</color></thing>简洁得多。

XML属性的一个非常有用的性质就是，可以将属性值声明为唯一值。这一性质可以用来强制执行有限类型的参照完整性。当然，仅仅使用XML文档本身并不能实现参照完整性（但是，利用17.3节中讨论的XML Schema的帮助，我们就能作到这一点，甚至更多）。介绍完DTD以后，就可以明白属性可以被定义成ID、IDREF和IDREFS等类型。

ID属性的一个特征就是：在整个文档范围内，其属性值必须唯一。这意味着，假设有两个ID类型的属性attr1和attr2，如果<elt1 attr1="abc" >和<elt2 attr2="abc" >出现在同一个XML文档中，那就是错误的（无论elt1和elt2是否为同样的标记，attr1和attr2是否为同样的属性）。从某种意义上来说，ID相当于关系数据库中的键。IDREF类型的属性必须引用在同一文档中声明的有效的ID。也就是说，它的值必须作为ID类型的另一个属性的值在文档的其他地方出现。因此，IDREF类似于关系数据库中的外键。

为了进一步说明，我们考察图17-4中的报告文档。IDREF类型的属性表示一个独立的指向有效ID的字符串列表。在图17-4中，我们为元素Student定义了一个ID类型的属性StduID，为元素Course定义了一个ID类型的属性CrsCode，元素CrsTaken的属性CrsCode是IDREF类型的，元素ClassRoster的属性Members也属于IDREF类型。结果，XML处理器会验证同一个学生或同一门课不会被重复定义，同时还能保证参照完整性，如CrsRoster不能引用不存在的Course，而所有Members列表中的Student都必须存在。

现在，我们可以定义一个重要的正确性的要求。一个XML文档如果满足下列条件，那么它就是一篇形式良好（well formed）的文档：

- 有一个根元素。
- 每一个起始标记后面都有一个与之匹配的结束标记，而且元素的嵌套关系正确。
- 在同一个起始标记中，任何属性最多只能出现一次，同时要给定属性值，并且属性值应该用引号括起来。

注意，关于ID、IDREF和IDREFS的限制并不是定义的一部分，因为这些属性类型由DTD进行指定，而形式良好的定义却完全忽略DTD。

```
<?xml version="1.0" ?>
<Report Date="2000-12-12">
  <Students>
    <Student StudId="111111111">
      <Name><First>John</First><Last>Doe</Last></Name>
      <Status>U2</Status>
      <CrsTaken CrsCode="CS308" Semester="F1997"/>
      <CrsTaken CrsCode="MAT123" Semester="F1997"/>
    </Student>
    <Student StudId="666666666">
      <Name><First>Joe</First><Last>Public</Last></Name>
      <Status>U3</Status>
      <CrsTaken CrsCode="CS308" Semester="F1994"/>
      <CrsTaken CrsCode="MAT123" Semester="F1997"/>
    </Student>
    <Student StudId="987654321">
      <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="666666666 987654321"/>
    </Class>
    <Class>
      <CrsCode>CS308</CrsCode><Semester>F1997</Semester>
      <ClassRoster Members="111111111"/>
    </Class>
    <Class>
      <CrsCode>MAT123</CrsCode><Semester>F1997</Semester>
      <ClassRoster Members="111111111 666666666"/>
    </Class>
  </Classes>
  <Courses>
    <Course CrsCode="CS308">
      <CrsName>Market Analysis</CrsName>
    </Course>
    <Course CrsCode="MAT123">
      <CrsName>Algebra</CrsName>
    </Course>
  </Courses>
</Report>
```

图17-4 具有交叉引用的报告文档

17.2.3 命名空间

最初的XML规范并没有考虑**命名空间**（namespace），后来才把命名空间补充进来。然而，时至今日，命名空间却成为在XML之上构建的许多重要标准的核心部分，所以从实用的角度上看，我们认为它是XML必不可少的特征。

引入命名空间源于人们相信，在不远的将来，企业会构造适合不同领域的术语词汇表

（如教育、商业、电子），并使用这些词汇作为XML标记。在这种情况下，不同词汇表之间的名字冲突是不可避免的，集成来自不同信息源的信息将变得非常困难。例如，依据谈论的对象是人或公司，术语Name可能具有不同的含义和结构，如同下面看到的两个文档片段：

```
<Name><First>John</First> <Last>Doe</Last></Name>
<Name>IBM</Name>
```

因此，应用程序对那些在含有冲突标记的单词表上构建的文档的处理将变得非常困难。

为了解决这个问题，XML标记名必须由两部分组成：命名空间（namespace）和本地名（local name），其一般结构如下：*命名空间*：*本地名*。除了其中多了一个“:”外，本地名和一般的XML标记没有什么区别。命名空间以一个统一资源标识符（Uniform Resource Identifier，URI）形式的字符串表示，这样的URI可以是一个抽象标识串（用作唯一标识符的通用字符串）或者以统一资源定位器（Uniform Resource Locator，URL）的形式表示命名空间。

这个想法看起来比较简单：不同的作者使用不同领域的不同的命名空间标识符，因此就能避免术语的冲突。自从命名空间问世以来，大家就一直遵循这一规则，即作者选择他们掌握的URL作为命名空间的标识符。例如，如果Joe Public为Acme公司服务的学校制定了一个单词表，所使用的命名空间为：

```
http://www.acmeinc.com/jp#supplies
```

而把玩具的命名空间定为：

```
http://www.acmeinc.com/jp#toys
```

WWW组织（W3C）建议书[⊖]中包含的XML[Nam 1999]的命名空间并不仅仅是简单的两部分命名模式，它也确定了专门的语法来声明命名空间、它们的使用及作用范围。下面是一个例子：

```
<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>
```

这里，命名空间用属性xmlns定义，这是一个保留字。实际上，W3C建议所有以xml开始的名字都保留为该组织使用。本例中，我们在item元素的作用范围内声明了两个命名空间。第一个是**默认的命名空间**（default namespace），用语句“xmlns=”进行声明。自然地，每一个开始标记都只能定义一个默认命名空间，这不仅是由默认命名空间的语义决定的，而且因为同一个开始标记中的同名属性只能有一个。语句xmlns:toy=declaration，定义了第二个命名空间，它由**前缀**toy确定。可以定义不同前缀的命名空间，只要这些前缀是不同的即可[⊜]。

⊖ W3C最后确定的文档一般称为“建议书”，但事实上，人们都把这些建议书作为标准。

⊜ 然而，当且仅当两个标记的前缀指向相同URI（相同意味着URI中的字符串完全相同）时，处理器会认为它们属于同一个命名空间（即使它们有不同的前缀）。

命名空间的声明

属于命名空间http://www.acmeinc.com/jp#toys的元素都应该以toy:为前缀。本例中，它们是内部标记toy:item和toy:name，而没有前缀的标记（外部的item、name和feature）被认为属于默认命名空间。

命名空间有一个作用范围，就像嵌套的程序模块一样。为了解释这个问题，我们来看看下面的例子:

```
<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>
```

其中，我们为最外层元素item增加了一个子元素，这个子元素也叫做item，但它有自己的默认命名空间以及重新声明的命名空间前缀toy。所以，最外层的item标记属于默认命名空间http://www.acmeinc.com/jp#supplies。

内部无前缀标记item及其无前缀的子标记name都在默认命名空间http://www.acmeinc.com/jp#supplies2的作用范围内。

同样，feature元素的内部标记toy:item和item:name属于命名空间http://www.acmeinc.com/jp#toys。

而子元素item的子元素toy:name的命名空间则是http://www.acmeinc.com/jp#toys2。

我们注意到，正像最内层声明的默认命名空间会覆盖最外层声明的默认命名空间一样，最内层的前缀toy的声明会覆盖最外层同样的前缀。能识别命名空间的XML处理器应该明白这些微妙之处。特别地，这两个没有前缀的元素item和name是不同的标记，因为它们属于不同的命名空间（对带有前缀名的标记name也一样）。不识别命名空间的XML处理器也能解析上述文档，但它会认为所有没有前缀的item和name是一样的，而所有出现的有前缀的标记toy:name是一样的。它仅仅会对名字中奇怪的“:”符号感到困惑。

尽管命名空间的概念似乎得到了广泛的认同（可能有人不赞同），但它却成为W3C提出的最令人费解的建议之一[Bourret 2000]。每个人都赞同标记名应该分成两个部分，但人们总想找出建议中的不足。最令人费解的一点就是使用URL作命名空间。以我们的日常经验而言，URL指的是一些Web资源，如果URL被用作命名空间，人们很可能将它理解为一个包含描述对应名字集的模式的真正的Web地址。实际上，命名空间的名字仅仅是一个碰巧是URL的字符串，如果将浏览器指向这样的URL而引出一个错误信息，那么会令人非常失望。

使用命名空间的最初目的是建立一种区分标记名字的机制。读取用命名空间编码的文档

的XML处理器应该知道如何解析它，换言之，就是知道如何发现它的模式（以DTD或XML Schema来表示，XML Schema是后面介绍的一种规范语言）。模式的位置信息可以在一个特殊属性中给出，或是特定的企业、领域的习惯作法的一部分。举个例子，玩具行业可能赞成将所有和玩具相关的XML文档都用位于某个URL的DTD来解析。目前的习惯作法是用某些“熟知”的命名空间来识别某些词汇（如那些用于XML Schema 规范的单词，见17.3节），这些命名空间可以唯一地指示出文档模式。

17.2.4 文档类型定义

作者必须遵循一些固定的规则来创立XML文档，以便这些文档能在浏览器中正确地表示出来。比如说，元素table不能出现在元素form中。而在XML中，作者能够为任何文档指定这样的规则，这就使得XML可以用于描述各种文档类型，如账单、目录和订单等，它们一般用于机器处理，而不是用于人工处理。

创建XML文档时应该遵守的规则集叫做**文档类型定义**（Document Type Definition，DTD）。DTD可以指定为XML文档的一部分，也可以在文档中给出能够找到其DTD的URL。符合自己的DTD的XML文档被认为是正确的（valid）。XML规范并不要求XML处理器检查每一篇文档是否都遵循自己的DTD，因为有些应用程序可能并不关心XML文档是否正确。在某些情况下，处理器并不检查文档的正确性，而是依赖于文档的发送者（如在电子账单传递中，传送双方使用的软件能保证产生的是正确的文档）。XML甚至不要求文档必须有DTD，但却要求所有的XML文档必须都具备形式良好性（形式良好的条件，即正确的元素嵌套与属性约束，已在17.2.2节中进行了讨论）。

上述关于正确性的两个说明可以使XML处理器得到明显的简化，并使处理器加快处理速度。HTML浏览器总是尽量纠正HTML文档中的bug，并尽可能地把文档中的bug报告出来。而XML处理器却只是要求拒绝那些形式不好的文档，而要求XML文档必须正确的处理器则根本不会处理那些不符合DTD的文档，即便它们是形式良好的文档。

对于熟悉形式语言的人来说，DTD就是在（定义）文档的标记与属性基础上指定合法XML文档的语法。下面就是图17-3的文档对应的DTD:

```
<!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>
]>
```

这个例子中，我们可以看到最常见的DTD组件：名称（例中是PersonList）和一组ELEMENT和ATTLIST语句。DTD的名字必须和符合该DTD的XML文档的根元素的标记名一

致。每一个允许的标记（包括根标记）有一个ELEMENT语句。此外，对于每一个可以有自己属性的标记来说，ATTLIST语句指定允许的属性和它们的类型。

本例中，第一个ELEMENT语句表示PersonList元素包含Title和跟随其后的Contents元素。Title元素（第二个ELEMENT语句）不包含任何元素（它是空元素）。而在Contents元素的定义中，“*”表示Contents有0个或多个Person类型的元素。如果不用“*”而用“+”，则表示Contents至少有1个Person类型的元素。元素Name、Number和Street都被声明为#PCDATA类型，即它们是字符串类型[⊖]。

在元素列表之后，DTD还包含允许的元素属性的描述。本例中，PersonList有属性Type和Date，而Title则只有一个属性Value，其他的元素没有属性。此外，#IMPLIED指定的PersonList的两个属性都是可选的，而#REQUIRED则表明Title的Value属性是必需的，这三个属性的类型都是CDATA，即字符串类型（注意，定义元素的字符串类型和定义属性的字符串类型的语法不一样）。

根据上面的DTD，图17-3中的XML文档是正确的。但如果我们将文档中的某个元素Address删除，那么它就不正确了，因为DTD要求每一个Person都必须有一个Address。另一方面，我们也可以上述DTD的ELEMENT Person的声明改为:

```
<!ELEMENT Person (Name,Id,Address?)>
```

使得Address是可选的，因为“?”指示Address元素出现0或一次。

我们也可以声明在person的描述中，元素的顺序是无关紧要的。这通过使用表示选择的连接符“|”实现:

```
<!ELEMENT Person
    ((Name,Id,Address)|(Name,Address,Id)|(Id,Address,Name)
      |(Id,Name,Address)|(Address,Id,Name)|(Address,Name,Id))>
```

然而，这样非常不方便。

DTD允许作者为一个属性指定多种类型，我们已经看到了CDATA，其他经常使用的类型是与图17-4的报告文档相关的ID、IDREF和IDREFS。我们指出这种类型的文档需要满足参照完整性，就像第4章中给出的数据库示例一样。

特别地，我们希望确保Student的属性StudId的值和Course的属性CrsCode的值在整个文档范围内具有唯一性，确保CrsTaken中的属性CrsCode引用同一文档中的课程（通过Course的CrsCode属性值的匹配），并确保由ClassRoster的属性Members说明的列表成员引用同一文档中给出的Student记录（同样通过StudId和Members的值匹配来实现），图17-5的DTD用于达到这一目的，这里忽略了那些容易推想出的部分。

要求文档正确性的XML处理必须根据该DTD的要求来确保不存在两个Student元素的StudId属性具有相同的值（Course元素也一样）[⊖]。这是因为这些属性被声明为ID类型。同样地，元素CrsTaken的属性CrsCode定义为IDREF，因此保持了参照完整性。ClassRoster的属性Members定义为IDREFS类型，即表示IDREF类型值的列表。该声明是为了保证对学生Id的参照完整性。

⊖ PCDATA表示已解析的字符串。

⊖ 事实上，还要保证一个更严格的条件，即Student元素的属性StudId和Course元素的属性CrsCode的值不能相同。

```
<!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 #IMPLIED>
    <!ATTLIST Student StudId ID #REQUIRED>
    <!ATTLIST Course CrsCode ID #REQUIRED>
    <!ATTLIST CrsTaken CrsCode IDREF #REQUIRED>
    <!ATTLIST CrsTaken Semester IDREF #REQUIRED>
    <!ATTLIST ClassRoster Members IDREFS #IMPLIED>
]>
```

图17-5　图17-4中报告文档的DTD

在文档中还有一些要求注意，但又不能通过DTD强制执行的约束，我们将在下一节讨论这些问题。

17.2.5　DTD作为数据定义语言的不足

XML被认为是SGML[SGM1986]的一种简化版本，SGML在XML开始使用的数年前就已经成为了标准。SGML语言用来指定可交换且能被软件代理自动处理的文档，这也是XML最初的目标。DTD及其使用的基本原理都借鉴于SGML。众所周知，XML的技术基础来自于形式语言理论和通用（语法）分析算法，分析算法能够验证任意给定的文档是否遵守DTD。这种验证对文档处理软件具有重要意义。例如，如果XML处理器希望它接收的文档经过验证，并符合上述Report DTD，它就不需要考虑那些特殊或例外情况，如某个学生可能选修了并不存在的一门课程或者丢失了一条街道的地址。

XML还在发展之中，新的方法正在出现。特别是XML给出了这样一种可能性，把Web文档作为一种可以查询的数据源（如同数据库中的关系），并且通过具有语义含义的链接使之联系起来（如同外键约束）。在这点上，XML超越了它所借鉴的SGML。由于出现的太晚而没有包含在XML 1.0中的命名空间的概念是XML的第一次扩展，而更有意义的扩展是发展出了XML Schema规范（下一节介绍），XML Schema是为了克服作为数据定义语言的DTD的许多不足之处而开发出来的。这些不足之处如下：

- DTD中没有用到命名空间的概念。DTD认为xmlns与其他的属性一样，并没有特别的意思。虽然扩展DTD以包含命名空间并不难，但这样就存在反向兼容性的问题，考虑到DTD的其他限制，这种扩展可能实际上并没有什么作用。

- DTD的语法和XML文档的语法有很大的差别。虽然这并不是什么致命的缺点，但也不是XML 1.0中好的特性。
- DTD有一个非常有限的基本类型指令系统（本质上仅仅是字符串）。
- DTD提供的表达数据一致性约束的手段有限。没有键（除了非常有限的ID类型），并且指定参照完整性的机制非常弱。唯一一种进行引用的方法是通过IDREF和IDREFS属性，即使这种引用只能基于一种基本类型——串。特别地，不能指定引用的类型。不能指定图17-4的报告文档中的元素CrsTaken的属性CrsCode只能引用Course元素。因此，John Doe可能会有一个子元素

```
<CrsTaken CrsCode="666666666" Semester="F1999"/>
```

它引用的是Joe Public的学生ID而不是一门课程，并且任何遵循XML 1.0的处理器都不会检测出这个问题。

- DTD拥有强制执行属性的参照完整性的途径，但元素却缺乏相应的特性。例如，元素Class的内容包含子元素CrsCode和Semester（不要和标记CrsTaken的属性混淆）。显然，我们希望元素CrsCode的内容能引用一门有效的课程，并和某些Course的属性CrsCode的值相匹配。而且，对于元素CrsTaken的每一对属性值，在某些Class元素中必须有与之对应的CrsCode/Semester标记对。而这些约束无法使用DTD来强制执行。
- XML数据是有序的，而数据库的数据是无序的（如元组之间的顺序就无关紧要）。同样，数据库关系或对象的属性之间的顺序也无关紧要，但XML的元素之间的顺序是有意义的。我们已经知道，DTD允许指定不同的组合，据此我们可以声明元素可以无序（如前面所举的Person元素的Name、Address、Id子元素的例子）。但是，当属性数量很多时，这种方法就变得非常不方便。例如，为了说明N个子元素可以无序，我们必须指出N!个可能的组合。
- 元素定义对于整个文档具有全局性。例如，某个DTD中指定了一个Name元素，它有First和Last两个子元素，那么在这个文档中，就不能定义一个名字同样为Name但结构不同的元素。这是因为，DTD中的每一个元素名只能有一个ELEMENT子句。无法根据父元素来局部化Name，以便根据Name出现的位置的不同而应用不同的定义。

17.3 XML Schema

XML Schema是XML文档的一种数据定义语言，它已经成为W3C组织推荐的一个标准。为了克服前述的DTD机制的不足，人们提出并发展了Schema，它具有以下主要特征：

- 其语法和一般XML文档的语法一致。
- 它结合了命名空间的机制。特别地，不同的模式可以从不同的命名空间中导入并合成一个模式。
- 它提供了许多内置的数据类型，如字符串、整型和时间型，这类似于SQL中的类型。
- 它提供了通过简单数据类型定义复杂数据类型的方法。
- 允许为不同嵌套结构的元素定义相同的元素名称。
- 支持键和参照完整性约束。

• 对于元素类型的顺序无关紧要的文档，它提供了一种更好的定义机制。

遵从某个模式的XML文档被称为**模式有效**（schema valid）的文档，并且该文档叫做该模式的一个**实例**（instance）[⊖]。和DTD相似的是，XML Schema规范并不要求XML处理器真正处理文档的模式。处理器也可以选择忽略文档的模式或者使用一个不同的模式。例如，XML处理器可能只考虑能够更好地遵守给定的参照完整性约束的文档，或者可能只要求执行模式的一部分。这一宽松的特点和数据库的不一样，数据库要求所有的数据必须满足数据库的模式。从这个意义上来说，尽管模式可能描述了部分XML文档，但XML数据从整体上来说是半结构化的（参见17.1节）。

17.3.1 XML Schema和命名空间

一个XML Schema文档（和DTD一样）是用来描述其他XML文档（实例）的结构的。Schema文档是以模式中使用的相关命名空间的声明开头，其中有三个命名空间非常重要。

- http://www.w3.org/2001/XMLSchema: 该命名空间规定了在模式中用到的标记和属性的名字。这些名字不和该模式的任何特定文档（XML文档）相关，也不出现在该模式的任何具体的文档中。相反，它们用于从总体上描述XML文档的结构属性。实际上，在http://www.w3.org/2001/XMLSchema中可能根本就没有文档。该命名空间的名字能被所有利用XML处理器的模式识别，并被认为是XML Schema规范中定义的名字。例如，与该命名空间有关的名字有schema、attribute和element，这些名字并不会像图17-4那样出现在实例文档中（如果实例文档真的包含了这几个名字，那么它们不是用来描述文档结构的，含义也不同，且应该属于另一个命名空间）。因此，这个命名空间确实是模式文档的一部分，但不能出现在实例文档中。
- http://www.w3.org/2001/XMLSchema-instance: 和http://www.w3.org/2001/XMLSchema相联的另一个命名空间，它规定了一些特殊的名字，这些名字在XML Schema规范中定义，不过这些名字在实例文档中使用，而不是在它们的模式中使用（因此该命名空间的名字叫作XMLSchema-instance）。例如，名字schemaLocation表示的是XML文档中模式所在位置。另一个名字定义了文档中的空值。在用到这个功能时，我们会向大家介绍这些特性。这个命名空间是实例文档规范的一部分。
- 目标命名空间: 规定了一个特定的模式文档所定义的一组名字，换句话说，是一组用户定义的名字，这些名字用于那个特定模式的实例文档。比如，在图17-4的模式文档中，名字CrsTaken、Student、Status等等就可以和目标命名空间联系在一起（下文中我们会介绍模式的各个部分）。我们通过模式文档的根元素schema的开始标记的属性targetNamespace来声明目标命名空间。

DTD的一个主要缺陷就是不能整合命名空间: 虽然DTD可以定义任意数量的标记，但却不能将这些标记和某个命名空间关联起来。

现在我们开始为图17-4的报告文档开发与之匹配的模式。在第一个例子中，我们只声明模式中要用到的命名空间。

⊖ 注意，这里所说的模式和实例与关系数据库和面向对象数据库中的术语模式和实例类似。

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://xyz.edu/Admin">

    <!-- 现在这里是空的 -->
</schema>
```

本例中，首先将标准命名空间XMLSchema声明为默认的命名空间。这样做是便捷的，因为在创建模式时，我们需要使用到许多XML Schema规范中定义的特殊标记，将XMLSchema定义为默认命名空间可以避免在使用这些标记时加上命名空间的前缀。如果希望将另一个命名空间设置为默认命名空间，那么我们可以使用以下语句:

```
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

根据约定，xsd是XMLSchema命名空间中所有名字的前缀，不过该约定不是强制的。当声明xsd作为XMLSchema命名空间中名字的前缀时，只要使用该命名空间中的标记就要加上前缀xsd，如下例所示:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xsd:targetNamespace="http://xyz.edu/Admin">

    <!-- 现在这里是空的 -->
</xsd:schema>
```

这里的第一个属性表示和命名空间XMLSchema相关的名字的前缀是xsd。第二个属性声明上述模式文档中定义的新标签和新属性属于命名空间http://xyz.edu/Admin。注意，因为targetNamespace是由XML Schema规范定义的，所以它的前缀是xsd。

```
<!-- 一个XML 模式文档; 位于http://xyz.edu/Admin.xsd -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://xyz.edu/Admin">

    <!-- 现在这里是空的 -->
</schema>

<!-- 符合上述模式的实例文档; 它使用了上述模式中定义的目标命名空间 -->
<?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">

<!-- 和图17-4的报告文档的内容相同 -->
</Report>
```

图17-6 模式和实例文档

假如我们已经填满了上述模式中的空缺部分，那么构造遵从该模式的XML文档（图17-4中的文档）时，该做那些事情呢？我们需要在实例文档中添加三项内容：声明它使用的命名空间（本例中为http://xyz.edu/Admin），报告文档的模式的位置，以及命名空间XMLSchema-Instance。需要添加最后一项是因为实例文档中出现属性schemaLocation，它是属于命名空间

XMLSchema-instance的，该属性指定了模式的位置。为了更好地理解模式、实际的实例文档和各种命名空间之间的关系，我们在图17-6中同时给出了报告文档和它遵循的模式。

可以看到，图中实例文档的默认命名空间是http://xyz.edu/Admin，它是模式文档中属性targetNamespace所定义的命名空间[⊖]。在这个URL中可以不存在任何内容，因为命名空间只是用于消除文档标记名和属性的歧义的标识符。因为图17-6中的文档和图17-4中的报告文档的内容一致，大多数的标记和属性名属于这个命名空间，所以选它作为默认命名空间可以最大限度减少文档中前缀的使用。

属性xsi:schemalocation是XML Schema规范的一部分，属于命名空间http://www.w3.org/2001/XMLSchema-instance。该属性的值是一个命名空间-URL对，这表明命名空间http://xyz.edu/Admin的模式文档可以在URL http://xyz.edu/Admin.xsd所指的XML模式文档中找到。然而，正如前面提到的那样，XML处理器并不受这些限制。它们可以忽略这个模式，也可以使用另一个模式。

在具体讲解如何定义模式之前，我们来看另一个重要的细节，这就是include语句。从图17-4中我们可以看到，图中的XML文档有三个不同的部分：学生列表、班级列表和课程列表。因为每个部分的结构都很不一样，所以假设它们单独出现在其他上下文中并拥有自己的模式也是合理的。鉴于这一点，仅仅为了创建图17-4的文档的模式，而将它们各自的模式全部拷贝过来就不合理了。我们可以使用XML Schema规范中定义的include语句来解决这个问题：

```
<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>
```

include语句的作用就是将给定文档的指定地址的模式包含进来。这一技术使XML模式更灵活以及模块化程度更高。被包含进来的模式文档必须和包含它的模式文档具有同一个目标命名空间。上例可能还有一个容易混淆的地方，属性schemaLocation并没有使用xsi前缀，而且和前面例子不同的是，它没有把命名空间XMLSchema-instance包含进来。这样做是有道理的。标记include的属性SchemaLocation是属于标准XMLSchema命名空间的（就像include本身一样）。也就是说，这个属性和前面的报告文档中的同名属性是两个不同的属性。和报告文档不一样的是，这里的模式文档不需要使用XMLSchema-instance命名空间中的名字，所以这个命名空间就不需要声明。

17.3.2 简单类型

1. 基本类型

缺少基本类型是DTD一大缺陷。为了克服这一缺陷，除string、ID和IDREF之外，XML

⊖ 本例中使用的大部分命名空间的位置和文档位置都不是真实的，这是为了保护那些真实位置上的信息。然而XMLSchema和XMLSchema-instance这个命名空间是真实的。

Schema规范增加了许多有用的基本类型，如decimal、integer、float、boolean和date。更重要的是，它提供类型构造器，如*list*和*union*，并且可以根据已有的基本类型构造出新的基本类型，这一机制和SQL中的CREATE DOMAIN语句相似（参见4.3.6节）。

2. 使用list和union构造器构造简单类型

注意，正如在DTD中一样，IDREFS并不是一种基本类型。下面展示了如何使用list构造器构造该类型[⊖]。

```
<simpleType name="myIdrefs">
    <list itemType="IDREF"/>
</simpleType>
```

若需要以两种或更多种方式输入数据，那么可以使用union类型。举个例子，美国的电话号码可以是7位也可以是10位，这可用以下方法来表示:

```
<simpleType name="phoneNumber">
    <union memberTypes="phone7digits phone10digits"/>
</simpleType>
```

马上我们会给出类型phone7digits和phone10digits的定义。

3. 使用限制来构造简单类型

构造新类型的一个更有意思的方法是使用**限制**（restriction）机制，这种机制使得我们可以使用固定的指令系统中一条或多条约束来限制基本类型，该指令系统在XML Schema规范中定义。我们就是这样定义类型phone7digits的:

```
<simpleType name="phone7digits">
    <restriction base="integer">
        <minInclusive value="1000000"/>
        <maxInclusive value="9999999"/>
    </restriction>
</simpleType>
```

可以用类似方法定义10位数字类型。在phone7digits的定义中，我们使用了标记maxInclusive和minInclusive来定义正确的数值范围。XML Schema提供了大量的内置约束，如maxInclusive/minInclusive，这些内置约束可以和[XMLSchema2000a，XMLSchema2000b]一起使用。这里我们只介绍几个比较常见的内置约束。另外，如果允许用户以XXX-YYYY的格式自己指定电话号码，那么可以通过几种方法来实现，下面给出其中的一种方法:

```
<simpleType name="phone7digitsAndDash">
    <restriction base="string">
        <pattern value="[0-9]{3}-[0-9]{4}"/>
    </restriction>
</simpleType>
```

这里我们使用pattern标记限制字符串必须和给定的模式相符。构造该模式的语言和Perl编程语言中用到的是相似的，但只要你熟悉诸如Vi或Emacs等文本编辑器，就不会对模式构造的基本原理感到陌生了。在上例中，[0-9]意味着“0到9之间的任何数字”，{3}是一种模式修饰符，

⊖ 除非特别说明，否则所有XML Schema例子中的默认命名空间都是标准命名空间http://www.w3.org/2000/10/XMLSchema。

它意味着匹配的字符串有且只有三位数字。

从基本类型string构造简单类型的方法还有以下几种:

- <length value="7">: 限定字符串长度只能为7。
- <minLength value="7">: 限定字符串长度至少为7。
- <maxLength value="14">: 限定字符串长度最多为14。
- <enumeration value="ABC">: 限制字符串的域为一个确定的字符集（下面将介绍）。

除字符串之外，其他类型也可以使用上述限制。事实上，enumeration可以应用于任何基本类型，下面就是一个这样的例子:

```
<simpleType name="emergencyNumbers">
    <restriction base="integer">
        <enumeration value="911"/>
        <enumeration value="333"/>
        <enumeration value="5431234"/>
    </restriction>
</simpleType>
```

4. 报告文档中的简单类型

现在我们为图17-4的报告文档定义一些简单类型。后面我们会将这些类型和文档模式中适当的属性联系起来。为了便于引用，我们将所有与学生相关的类型归纳在图17-8中，所有和课程相关的类型归纳在图17-9中。

```
<simpleType name="studentId">
    <restriction base="ID">
        <pattern value="[0-9]{9}"/>
    </restriction>
</simpleType>
<simpleType name="studentRef">
    <restriction base="IDREF"
        <pattern value="[0-9]{9}"/>
    </restriction>
</simpleType>
<simpleType name="studentIds">
    <list itemType="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>
```

第一个类型studId将学生Id定义为长度为9的数字字符串，该类型用于描述报告文档中studId的值域。第二个类型定义了引用学生Id的类型，第三个类型定义了引用学生Id的列表，第四个类型定义课程编码为三个大写字母后跟三位数字的字符串，第五个类型定义的是课程的引用类型。注意，我们将ID和IDREFS作为基类型，它们的语义和DTD中的语义相同，因而

确保了唯一性和参照完整性。

上面的定义和图17-5中的DTD相比，已经有了很大的改进。在DTD中不可能说明属性Members返回的是对学生的引用列表而不是对课程的引用列表，也不可能对标记CrsTaken的属性CrsCode施加相似的限制。相比较而言，上述简单类型能够避免这种无意义的引用，这是因为类型courseRef同studentId的值域不相交，类型studentRef同courseCode的值域也不相交。

5. 简单元素和属性的类型声明

到目前为止，我们还没有谈到不和元素以及属性联系在一起的类型。这里给出了一些简单的例子，用于描述报告文档中标记的类型声明，这些将成为报告的模式文档的一部分:

```
<element name="CrsName" type="string"/>
<element name="Status" type="adm:studentStatus"/>
```

第一个声明表示元素CrsName的内容是string类型的。最后一个声明是个假设，将标记Status与导出类型studentStatus相关联，studentStatus被定义成枚举类型，值为字符串U1、U2、U3、U4、G1、G2、G3、G4和G5，用以表示各级本科生和研究生。

```
<simpleType name="studentStatus">
    <restriction base="string">
        <enumeration value="U1"/>
        <enumeration value="U2"/>
        .
        .
        .
        <enumeration value="G5"/>
    </restriction>
</simpleType>
```

本例中非常重要的一处是studentStatus的前缀adm，adm是因为考虑到命名空间而引入的。为了更好地理解这点，让我们看一下上述声明所在的上下文:

```
<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>
```

在模式文档中，默认的命名空间通常是XMLSchema命名空间。这使得我们频繁地使用诸如element、simpleType、name和type等时，可以不加前缀。另外，模式文档中定义了许多属于目标命名空间（本例中为http://xyz.edu/Admin）的类型（如studentStatus）、元素（如Status）

和属性（参见后续章节）。定义新元素或类型的时候，并不需要加上前缀（如name="Status"和name="studentStatus"），这是因为这些名字是新定义的，正因为是新定义的，所以不可能属于默认命名空间。它们被自动加入到目标命名空间中。然而，怎样才能引用该模式中定义的名称呢（比如说，在type属性中引用studentStatus）？如果不使用前缀，XML处理器就会认为该名字属于默认命名空间。这正是处理string类型的CrsName元素的情况。因为string没有前缀，所以它被认为来自标准的XML Schema命名空间，这是正确的。但是，若不加前缀地使用studentStatus，那就会导致处理器认为studentStatus也是出自默认命名空间，这是错误的，因为XML Schema并没有定义studentStatus。因此我们应该为目标命名空间定义前缀，每次引用到其中的标记时都要加上该前缀。上例中第二个xmlns属性（schema元素中）的定义就是为了将adm和目标命名空间联系起来。从现在起，我们就假设目标命名空间对应的前缀是adm，以后在已定义的类型中使用它时就不作说明了。

接下来，让我们看看该如何定义文档中的属性类型：

```
<attribute name="Date" type="date"/>
<attribute name="StudId" type="adm:studentId"/>
<attribute name="Members" type="adm:studentIds"/>
<attribute name="CrsCode" type="adm:courseCode"/>
```

注意，这些定义并没有将属性和元素联系在一起，在这一点上属性并没有什么意义。这里还不能建立这种联系，因为具有属性的元素被认为是*复杂类型*（即使内容为空，如CrsTaken），所以我们需要先熟悉这些复杂类型。

17.3.3 复杂类型

1. *基本的例子*

到目前为止，我们已经知道如何定义简单类型，即不包含属性或子元素的元素类型。图17-7的模式片段定义了一个复杂数据类型，该类型能表示报告文档中的Student元素。

```
<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>
```

图17-7　复杂类型studentType的定义

这个简单的例子包含两个类型声明，并具备多个新的特征。第一，使用了标记ComplexType而不是SimpleType，这样XML处理器就知道将处理的是复杂类型。第二，标记

sequence用于指定元素name、Status和CrsTaken必须以给定的顺序出现。第三，元素CrsTaken（稍后给出它的类型定义）出现的次数可能是0次、1次或多次。一般来说，minOccurs和maxOccurs的值可以是任何数字。而DTD中作这样的说明虽然可行，但是很不方便，因为DTD必须使用选择方案（规定使用的是“|”符号）罗列出所有的可能，这样会导致模式冗长。对于其他元素，我们没有定义minOccurs和maxOccurs，因为它们的默认值都为1（总之是我们想要的结果）。最后，该复杂类型定义结尾的属性声明将StudId和studentId类型关联在一起（参见图17-8），而且因为它出现在studentType的定义中，所以意味着每一个studentType类型的元素都必须包含这个属性（除此之外没有其他属性）。

图17-7中的第二个类型声明给出了Name元素的类型，Name元素在studentType类型的定义中需要用到，这个声明中没有涉及schema语言的新特性。

定义一个元素为复杂类型与定义一个元素为简单类型没什么区别。下面的声明将Student元素定义为复杂类型studentType:

```
<element name="Student" type="adm:studentType"/>
```

2. 特殊情况

如果考虑下面的两种特殊情况，刚才介绍的简单情况就变得复杂了：怎样才能定义一个具有简单内容（如没有子元素的文本类型）和属性的元素类型，以及怎样才能定义只有属性而没有内容的元素（在DTD中定义为EMPTY）？我们已经在Romeo和Apothecary的对话文档中接触过第一种情况，图17-6中的CrsTaken元素表示出第二种情况。

第一种元素定义起来不太方便，因为这种情况在使用XML的数据表示中很少出现，所以这里我们就不予介绍。而诸如CrsTaken的第二种元素的类型定义就很简单了，如下所示:

```
<complexType name="courseTakenType">
    <attribute name="CrsCode" type="adm:courseRef"/>
    <attribute name="Semester" type="string"/>
</complexType>
```

3. 构造元素组

从图17-7中的studentType的定义中，我们知道怎样使用sequence将多个元素组合成一个有序元素组。像sequence这样能够将多个元素合成组的标记叫做**合成器**（compositor）。当标记具有**复杂的内容**，即它至少有一个子元素，这时我们必须使用合成器来完成元素的定义。XML Schema提供了几种合成器，其中一种允许将元素组成一个无序元素组。注意，在17.2.5节中，缺少一种实用的方法来定义无序元素集是DTD的一个重要的缺陷。

假设我们希望街道、门牌号码和城市可以以任意的顺序出现在地址中，那么我们可以使用合成器all:

```
<complexType name="addressType">
    <all>
        <element name="StreetName" type="string"/>
        <element name="StreetNumber" type="string"/>
        <element name="City" type="string"/>
    </all>
</complexType>
```

但是，使用all必须满足许多限制条件，这就使得它不能广泛地使用。这些限制条件包括：第一，all必须直接出现在complexType的下方，所以下面的定义是不正确的：

```
<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"/>
    </sequence>
    <attribute name="StudId" type="adm:studentId"/>
</complexType>
```

第二，all包含的所有元素都不能重复。换句话说，all的所有子元素的maxOccurs值必须为1，所以下面的定义是不允许的：

```
<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>
```

XML Schema中的第三种合成器是choice合成器，在复杂类型中使用choice的作用和在简单类型中使用union的作用是一样的。下面就是说明其用法的一个例子：

```
<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允许我们使用邮箱来代替街道地址。也就是说，一个合法的地址或者是一个邮箱，或者是一个街道地址。

注意，即使类型只有一个子元素，也需要使用合成器之类的内容描述器，例如：

```
<complexType name="foo">
    <element name="bar" type="integer"/>
</complexType>
```

是错误的，而

```
<complexType name="foo">
    <sequence>
        <element name="bar" type="integer"/>
    </sequence>
</complexType>
```

才是正确的。

4. 局部元素名字

在DTD中，所有的元素声明都是全局的，因为它只允许一个元素名对应一个ELEMENT语句。因此，（对任何一个DTD而言）无法定义出一个有效的报告文档，其中元素Student和Course都有一个名为Name的子元素。事实上，图17-4中课程名是一个字符串，而学生名却是一个复杂类型personNameType。这就是在报告文档中使用CrsName而不使用Name的最根本的原因。因为同样的原因，在DTD中不允许使用元素名Course替代CrsCode作为Class元素的子元素名，因为Courses中已经定义了一个子元素Course，而且它的结构和Class的子元素CrsCode并不一样。所以，如果我们用Course替换CrsCode，那么在DTD中Course必须对应两个不同的ELEMENT语句，这是不可能的。

XML Schema规范通过提供局部范围的元素声明来解决这个问题。这种做法和许多编程语言一样。元素类型的作用域是距离最近的包含<复杂类型…>…</复杂类型>块，在报告文档中，局部范围机制使得我们可以将标记CrsName改名为Name并定义下面的模式：

```
<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>
```

这里studentType和courseType都包含子元素Name。前一个Name子元素是一个复杂类型personNameType，包含First和Last两个元素。后一个Name子元素是简单类型String。然而，和DTD不一样的是，因为这两个声明的作用域不一样，所以它们不会冲突。

5. 导入模式

在17.3.1节中，我们介绍了怎样使用include将分散在不同文件中的模式导入构成一个模式。这一语句使得合作开发小组可以模块化地构造复杂的模式。正因为这种使用方法，被包含的模式文档和包含它的模式文档必须属于同一个命名空间。

同时，XML Schema规范的设计者也意识到，只有将不同的组或组织构造的模式文档合并起来，Web的潜在性能才能得以实现。这就是import语句的目标。和include语句一样，import也提供了schemaLocation属性，只不过该属性对于import是可选的。import语句中唯一必须的

属性是namespace，因为导入具有不同命名空间的模式文档是可能的。假如import语句中没有指定schemaLocation的值，那么XML处理器就会自己寻找该模式文档，处理器可能会根据一些约定从命名空间中得到模式文档。即使指定了schemaLocation的值，XML处理器也可以忽略它或者使用另一个模式，XML处理器唯一不能忽略的就是命名空间。

在下例中，我们用import语句替代include语句:

```
<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>
```

这里，我们假设上述模式文档的实例文档包含学生记录和课程，它们属于不同的命名空间，而且报告文档的处理软件能找到描述课程的模式。因此，在第二个import语句中并没有提供schemaLocation属性（但第一个语句中提供了该属性）。第一个import语句将目标命名空间为http://xyz.edu/Registrar的schema文档Sch导入。而上面的模式声明该目标命名空间对应的前缀是reg，这样我们就可以用reg:x作为对Sch中的元素x的引用。

6. 通过extension和restriction构造新的复杂类型*

某些情况下，用户可能需要对用include或import包含进来的模式文档作某些修改。这在include中是很容易做到的，因为文档作者可以控制所有的文档。然而，在import中，包含进来的外部文档的控制权通常是属于某个外部机构的，导入它们的模式文档可能无权复制它们，或者也不想这样做。例如，在许多情况下，导入者只想得到原模式的一个视图，这样导入者的模式可以和原模式同步变化。

XML Schema规范提供两种机制来修改导入的模式: extension和restriction。它们都是第16章介绍的*subtype*概念的特例。扩展模式意味着向文档中增加新的元素或属性。限制模式文档意味着进一步限定其定义，以便过滤掉一些不符合条件的实例文档。

假设foo.edu打算以xyz.edu为例，以XML语言构建他们的注册系统。他们的模式总的来说和xyz.edu的相同，但希望为每个课程记录添加一个简短的课程提纲。因为xyz.edu会经常改进它的学生注册工具，所以foo.edu希望导入该模式并使用扩展机制，而不是将它直接拷贝过来，这样就可以利用到原模式的改进。特别地，foo.edu想要扩展类型courseType（参见图17-9），在该类型中增加一个新的元素syllabus（课程大纲）。最后，他们建立了如下的模式文档:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xyzCrs="http://xyz.edu/Courses"
        xmlns:fooAdm="http://foo.edu/Admin">
        targetNamespace="http://foo.edu/Admin">
    <!-- fooAdm is the prefix to be used with the target namespace -->
    <import namespace="http://xyz.edu/Courses"/>

    <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>
```

注意，现在的目标命名空间是http://foo.edu/Admin，这是大学客户的命名空间，对应的前缀是fooAdm。文档使用import语句将xyz.edu包含进来作为构造新的模式的基础。因为新的模式引用了xyz.edu的命名空间中的名字（如courseType），所以我们需要为导入的命名空间xyz.edu定义一个前缀，这里使用的是xyzCrs。

定义好命名空间之后，我们需要使用导入的模式中的对应类型，定义新的类型courseType。新定义的类型没有前缀，因为我们希望它属于目标命名空间。然而，从xyz.edu中导入的基类型需要加前缀，以xyzCrs: courseType作为对它的引用。为了告诉XML处理器将要处理的是一个由另一个类型修改得到的复杂类型，XML Schema规范要求必须给出<complexContent>…</complexContent>标记对。在这个标记对中，需要指定extension或restriction中的一种。上例中使用的是extension，它表示指定元素syllabus将被加入到类型xyzCrs:courseType中以构成新类型courseType（新类型属于目标命名空间http://foo.edu/Admin）。

foo.edu可能会对原模式文档做其他的修改。例如，它们可能需要定义一个像命名空间http://xyz.edu/Admin（见图17-8）中定义的studentType一样的元素，有一点不同的是，不允许学生选修任意数量的课程（原模式中maxOccurs="unbounded"）。因此，foo.edu会将原模式的选修课程数目限制为63，如下所示:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:xyzCrs="http://xyz.edu/Courses"
    xmlns:fooAdm="http://foo.edu/Admin"
    targetNamespace="http://foo.edu/Admin">

  <import namespace="http://xyz.edu/Courses"/>
  .
  .
  .
  <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>
```

和类型扩展机制类似，我们在complexContent块中使用了restriction标记。然而，它们的主要区别在于，在限制复杂类型时，我们必须重复基类型中所有元素和属性的声明。同时，我们可以在基类型的各部分上加限制，例如，将maxOccurs="unbounded"作更严格的限制，改为maxOccurs="63"。

17.3.4 一个完整的Schema文档

在前面的例子中，我们已经接触了很多定义模式的技巧，现在让我们将这些部分合并起来组成一个完整的模式，该模式包含许多类型定义并至少有一个全局元素（即文档的根元素）的定义，该定义将根元素名和一个复杂类型联系起来。这个复杂类型可能还包含其他元素和属性的声明，而这些子元素和属性的定义同样又和它们的类型联系起来。从根元素开始，我们可以在它的下一层找到它所有的属性和子元素。这样重复下去，我们可以找到处于文档结构任意层次的元素和属性。

```
<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/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>
```

我们略去了较低层的classOfferings类型和courseCatalog类型的定义，它们的定义和studentList的定义类似。和studentList一样，这些类型也是根据图17-8和图17-9中所示类型的形式来定义的。

```
<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"/>
            <element name="CrsTaken" type="adm:courseTakenType"
                     minOccurs="0" maxOccurs="unbounded"/>
        </sequence>
        <attribute name="StudId" type="rpt: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="studentRef"/>
    </simpleType>
    <simpleType name="studentRef">
        <restriction base="IDREF">
            <pattern value="[0-9]{9}"/>
        </restriction>
    </simpleType>
</schema>
```

图17-8 http://xyz.edu/StudentTypes.xsd处的Student类型

和以前一样，我们必须小心处理命名空间，所以这里将目标命名空间的前缀定义为adm，引用该模式文档中的任何名字都需要加上这个前缀（除了使用属性name定义这些名字的声明语句）。前面提过，包含的模式和被包含的模式应该属于同一个命名空间，所以前缀adm既是包含模式（如adm:courseCatalog）中名字的前缀又是被包含的模式（如adm:studentType）中名字的前缀。

```
<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>
```

图17-9　http://xyz.edu/CourseTypes.xsd处的Course类型

匿名类型

到目前为止，我们定义的所有类型都是**有名字的类型**（named type），因为每个类型定义都有一个相关的名字，而每一个新元素都有一个有名字的类型与之匹配。当我们需要在几个元素或属性的定义间共享同一个类型时，对类型进行命名就很有用。然而，在很多情况下，某个类型可能只有一个，不会被再次用到。例如，在上述报告文档的完整的模式中，reportType类型（以及几个其他的类型，如studentList和classOfferings）就不需要被共享，在这种情况下，使用**匿名类型**（anonymous type）更方便。

匿名类型和有名字的类型的定义很相似，只是匿名类型的定义中没有name属性，而且匿名类型的定义必须和使用它的element或attribute的定义写在一起。这些附上匿名类型的定义和原来的定义有些不同。首先，定义时不使用type属性引入匿名类型。其次，定义时使用标记对而不是使用空标记<element…/>和<attribute…/>，匿名类型的定义被包含在开始和结束标记对中，这样我们就可以使用匿名类型将上面的Report元素的定义修改为：

```
<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>
```

类似地，我们可以匿名类型来修改元素Students、Classes和Courses的定义。在这种情况下，相应类型定义的内容必须直接包含在上述模式中。

17.3.5 完整性约束

前面我们谈到了XML文档中的参照完整性的问题，并介绍了XML Schema规范是怎样克服DTD在这方面的缺陷的。然而，即使在XML Schema规范中，我们仍然使用从DTD继承的特殊类型ID、IDREF和IDREFS。为了说明这种机制的局限性，我们定义图17-4中的Class元素的类型。

```
<element name="Class" type="adm:classType"/>
<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="classListType">
  <attribute name="Members" type="adm:studentIds"/>
</complexType>
```

显然，如果某个学生通过Student的子元素CrsTaken宣称他选了某门课程，相应的课程必定在特定的学期中开设。课程给出的方式在我们文档中通过Class元素描述。假定有下面的元素:

```
<CrsTaken CrsCode="CS308" Semester="F1997"/>
```

那么就一定存在下列形式的元素

```
<Class>
  <CrsCode>CS308</CrsCode><Semester>F1997</Semester>
  :
</Class>
```

问题是，CrsCode和Semester都不能唯一地决定一个Class元素，因此这时ID/IDREF机制就不能发挥作用[⊖]。而且，使用ID/IDREF机制，我们不能将文档某部分的一组值（如CrsTaken标记的CrsCode和Semester属性）关联到文档另一部分的一组值（Class的子元素CrsCode和Semester）之上。这是经常会碰到的问题，在传统数据库中，我们使用多属性键（键由多个属性组成）来解决这个问题。

1. XML的键

为了解决上述问题，XML Schema规范允许使用多属性键和外键约束，使用方法和SQL中的方法相似。但稍微复杂一些，SQL处理的是平面的关系，所以指定多属性键时，只要将属于该键的属性列出即可。同样地，在SQL中定义外键约束时，只要列出引用和被引用的关系中的相关属性就可以了。但是，在XML中处理的却是复杂结构的数据，键的概念就更加复杂。事实上，键可能是由位于元素内部不同层次的值的序列组成的。

假设引用的上下文环境是Class元素的父元素，那么我们可以认为Class元素集合的键是由

⊖ 在XML Schema中，类型ID和IDREF不仅限于属性。

路径表达式对Class/CrsCode和Class/Semester能访问到的值所组成的。从第16章的介绍可知，我们已经熟悉了路径表达式的概念，但在XML中，路径表达式更加细致。事实上，XML路径表达式是另一种规范——XPath——的一部分，我们将在17.4.1节中学习XPath的相关内容。

为了说明XML中键规范的复杂性，我们扩展Class元素的定义，在其中增加班级，并从学期的年份中分解出季节。

```
<Class>
  <CrsCode Section="2">CS308</CrsCode>
  <Semester><Season>Fall</Season><Year>1997</Year></Semester>
  .
  .
  .
</Class>
```

这里，能唯一决定class的一系列值分散在不同地方（属性、元素内容）和不同层次（在CrsCode标记的Section属性，CrsCode标记的内容，Semester元素的子元素Season以及Semester的子元素Year中都有）。下面就是用Xpath描述的访问它们的路径表达式:

```
CrsCode/@Section
CrsCode
Semester/Season
Semester/Year
```

所有这些路径表达式都和报告文档中的Class元素有关。第一个路径选出标记CrsCode的属性Section的值，这里CrsCode必须是当前元素（设为Class元素）的子元素。第二条路径选出CrsCode元素的内容（即CrsCode节点的开始标记和结束标记之间的文本）。第三条路径得到的是元素Season的内容，这里Season必须是元素Semester的子元素，而元素Semester又必须是当前元素的子元素。第四条路径也是类似的情况。

XML Schema提供了两种定义键的方法。一种是使用标记unique，和SQL中的限定词UNIQUE类似。标记unique指定了*候选键*（第4章中的术语）。另一种是使用标记key，对应SQL中的PRIMARY KEY约束。在XML中，unique和key之间的唯一区别就在于key不能有空值（在XML中，形如<footag></footag>的元素的值是空字符串但不是空值）。若要指定footag的值为空值（在XML Schema中称为nil），那么需要下面的定义:

```
<footage xsi:nil="true"></footag>
```

这里的nil是在以下命名空间中定义的:

```
http://www.w3.org/2001/XMLSchema-instance
```

(这里我们设xsi是这个命名空间对应的前缀)。

下面是报告文档的主键声明的例子。除了用unique标记代替key标记外，候选键的定义是相似的。引用本节之前定义的classType类型，我们想指定CrsCode和Semester的一对值用于在该文档中唯一确定Class元素。这可以通过以下方法做到:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:adm="http://xyz.edu/Admin">
        targetNamespace="http://xyz.edu/Admin">

  <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>
        <element name="Class" type="adm:classType"
                 minOccurs="0" maxOccurs="unbounded"/>
        </sequence>
      </complexType>
    </element>
    <element name="Courses" type="adm:courseCatalog"/>
  </sequence>

  <key name="PrimaryKeyForClass">
    <selector xpath="Classes/Class"/>
    <field xpath="CrsCode"/>
    <field xpath="Semester"/>
  </key>
</complexType>

<complexType name="classType">
  <sequence>
    <element name="CrsCode" type="adm:courseCode"/>
    <element name="Semester" type="string"/>
    <element name="ClassRoster" type="adm:classListType"/>
  </sequence>
</complexType>
.
.
.
</schema>
```

上面的模式给出了报告文档的相关类型定义。命名空间声明和类型classType在前面已经讨论过了。类型reportType用于定义报告文档的根节点Report。它由三个元素组成：Students、Classes和Courses 。Students类型在17.3.4节的开始已经定义过了。为方便引用，我们将元素Classes的类型扩展为匿名类型，并将类型定义直接和元素定义写在一起。Courses元素的类型courseCatalog的定义比较简单，读者可以作为练习。

这里最有趣的特征就是使用key标记来声明键，并通过其属性name的值PrimaryKeyForClass来表明这是一个主键。请注意，即使键只涉及ClassType中的部分，键的定义仍然出现reportType的定义中而不是出现在classType的定义中。这样写是故意的，目的是说明XML键的定义是和对象集（通常是元素集）关联，而不是和类型关联。selector标记的属性xpath指定了一个路径表达式。该路径表达式确定了键声明所应用的对象集。对象集不一定对应一个类型。例如，XPath路径表达式可以返回两个或者更多种不同类型的元素集合（如17.4.1节中的CrsTaken和Class的并）。这样，在XML中，键不一定要和某个类型相关联，这和我们以前所看到的很不一样[⊖]。在我们的例子中，键对应的路径表达式是Classes/Class，该路

⊖ 回忆一下，在关系模型中，键是为某个模式的元组集合定义的。在E-R模型中，键是为某个实体或联系而定义的。

径表达式和类型reportType相关。选择器确定的是文档中所有的Class元素的集合（正巧和所有classType类型的所有元素集合相一致）。

指定了合适的对象集合之后，接下来我们使用子元素field来描述键的组成。正如前文解释的那样，这些字段可以来自一个对象的不同地方，也可以具有复杂的嵌套层次。然而，本例比较简单，键的第一个字段是元素Class的子元素CrsCode的内容，第二个字段是子元素Semester的内容（在field子句的xpath属性中所描述的路径表达式和选择器确定的对象集合是相关的。这就是为什么第一个路径表达式简单的表述为CrsCode，而不是Classes/Class/CrsCode）。注意，作为对键的某个字段的具体描述，路径表达式如果要有意义，那么它必须为每个它应用的对象只返回一个值。举例来说，对于任意给定的Class元素，路径表达式CrsCode只返回一个值，因此字段描述

```
<selector xpath="Classes/Class"/>
<field xpath="CrsCode"/>
.
.
.
```

是允许的。相反，元素Student的范围内的路径表达式CrsTaken/@CrsCode会返回一组学生选修的课程（请参见图17-8中studentType的定义），因此字段描述

```
<selector xpath="Students/Student"/>
<field xpath="CrsTaken/@CrsCode"/>
.
.
.
```

是不允许的。

2. XML中的外键约束

接下来，我们希望学生记录中的每个CrsTaken元素能引用同一个报告文档中确实存在的课程元素。这就类似于外键约束，如图17-10所示，这里使用keyref元素定义这种关系。

外键约束由约束名、引用标识符、选择器以及一系列的字段组成。约束名在这里无关紧要。引用通过属性refer来定义，它的值必须和key或unique约束的名字一致。在本例中，它和Class元素的主键约束PrimaryKeyForClass匹配。

在SQL中，引用的值相当于外键约束中的REFERENCES关系名部分。接下来我们讨论一下选择器。在声明键约束时，选择器通过它的属性xpath定义**源集**（source collection）。而这里所讨论的元素集由所有CrsTaken元素组成。每一个CrsTaken元素都要引用键约束PrimaryKeyForClass定义的**目标集**（target collection）中的某个对象的键。

最后，我们来介绍一下外键本身，即CrsTaken元素（源集）内部的各个字段，这些字段才真正引用目标集合（主键约束中定义）中的字段。我们使用大家已经熟悉的field标记来描述。和前面一样，field标记提供了能得到某个值的路径表达式（该路径相对于所选择的对象集）。我们希望CrsTaken元素的源集属性CrsCode和Semester能引用主键的字段，该主键是Class元素的目标集的主键。在XPath中，我们使用@CrsCode和@Semester来确定这些属性（参见图17-9）。和组成键的字段一样，源集中任何对象上的路径表达式的值必须是唯一的。

相似地，我们可以在报告文档中定义其他的主键和外键约束，也可以使用数据类型ID和IDREF替换前面的主键和外键的约束定义（参见练习17.5和17.7）。从另一方面来看，没有一

个很好的方法能够使用key和keyref来指定IDREFS风格的参照完整性。比如，元素ClassRoster（在17.3.5节的起始部分）的属性Members的类型是studentIds，studentIds是由一系列studentRef类型的值组成的。因为studentRef类型是由基类型IDREF通过施加限制而得来的（参见图17-8），所以studentIds可以看作是IDREFS的一个特例。我们可以尝试使用下面的方法来定义这种参照完整性:

```
<keyref name="RosterToStudIdRef" refer="adm:studentKey">
  <selector xpath="Classes/Class"/>
  <field xpath="ClassRoster/@Members"/>
</keyref>
```

这里，studentKey是Student元素的一个主键约束。但问题是，属性Members的值是一个列表，而Student标记的键属性StudId的值却是一个项。而且，在XPath语言中，不可能建立这样的引用，引用者是列表数据类型（如属性Members）的某个对象，而被引用的是文档中的其他实体（也就是Student元素中定义的学生Id）。能解决这一问题的唯一方法就是使得Members中的学生Id不出现在列表中，而是以单值的形式出现（参见练习17.9）。

```
<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"/>
      <element name="Classes" type="adm:classOfferings"/>
      <element name="Courses" type="adm:courseCatalog"/>
    </sequence>

    <key name="PrimaryKeyForClass">
      <selector xpath="Classes/Class"/>
      <field xpath="CrsCode"/>
      <field xpath="Semester"/>
    </key>
    <keyref name="NoBogusTranscripts" refer="adm:PrimaryKeyForClass">
      <selector xpath="Students/Student/CrsTaken"/>
      <field name="@CrsCode"/>
      <field name="@Semester"/>
    </keyref>
  </complexType>
</schema>
```

图17-10 带有主键和外键约束的模式的一部分

17.4 XML查询语言

为什么需要查询XML文档？现有的数据库技术能存储并发布XML吗？

将XML数据存储在专门为XML而设计的数据库中，已经不成问题。人们已经找到能够有效地存储和检索树结构的对象，当然也包括XML文档[Deutsch et al.1999，Zhao and Joseph 2000]的办法。然而，XML文档多半会被映射成现有的关系或面向对象的格式而被存储起来。另一方面，目前所有主流DBMS厂商的产品都已经兼容XML。它们能够接受XML文档并将XML文档转换成关系或对象形式，并提供工具将已有的关系数据库或对象数据库中的数据转换成XML文档对外发布。一旦生成XML文档，就可以将它传递到别的机器上，接收到该文档的机器可以自动地处理它，也可以将它呈现在用户面前。为了达到这一目的，W3C组织为XML定义了**文档对象模型**（Document Object Model，DOM）[DOM2000]，DOM将客户端程序访问XML文档各个部分的接口标准化，因而简化了此类程序的编写过程。

XML查询语言和这些有什么关系呢？假设你正在准备下个学期的课程表，并需要找出该学期下午3点到7点的所有课程。如果大学的数据库服务器允许你进行这样的查询，那么问题当然好解决，但更大的可能是，数据库仅提供了一个的固定接口，这个接口仅只持有限的一组查询。此时，要查找出需要的结果，你可能需要填写一连串的表单并且靠眼睛来分析结果，甚至可能需要使用诸如笔和纸这样的工具将一些需要的信息记录下来。

一种替代方案就是，向服务器请求包含本学期所有课程列表的XML文档，并且使用一个客户端程序找到想要的信息。正如前面介绍的那样，DOM在很大程度上简化了这一过程。尽管如此，它提供的仅仅是底层的XML接口。如果你的查询需要联结存储在文档不同部分的信息或者存储在不同文档中的信息，那么你可能需要编一大段程序（可能等程序调试完，这个学期就将结束了）。一种可行的方案就是使用嵌套循环和if语句来代替复杂的SQL查询语句。在后面，大家将看到一种功能更强的高级查询语言，它可以简化这一查询过程，这种查询语言允许客户端程序以智能、可定制的方式来处理信息。

在本节后面的内容中，我们将讨论XML的三种查询语言：XPath[XPath1999]、XSLT[XSL 1999]和XQuery[XQuery 2001]，其中XQuery是W3C最近提出的一种XML查询语言并将最终成为W3C推荐的官方查询语言，它是在Quilt语言的基础上建立起来的[Chamberlin et al.2000,Robie et al.2000]。

XPath的提出者希望XPath语言简单而有效。它建立在路径表达式的概念之上（第16章我们已经介绍了路径表达式的含义），并被设计为紧凑的并且可以和URL合并的语言。将XPath表达式和URL合并是XPointer规范[XPointer 2000]的一部分，这部分内容我们也将做简略的介绍。XSLT是一门成熟的编程语言，具有强大的查询功能（尽管有限）。XQuery是一种SQL风格的查询语言，它的概念符合人们对传统数据库查询语言的理解。在本章所介绍的所有XML查询语言中，XQuery是功能最强大，设计最完善的一种。

17.4.1 XPath：一种轻量的XML查询语言

在面向对象的语言（如OQL，见16.4.2节）中，路径表达式就是对象属性的序列，这个序列指明了到达嵌套在对象结构内部的数据元素所需的确切路径。如果编程者知道了数据库的

模式并且这个模式改变的可能性不大的话，给出这样的确定路径并不困难。然而，当数据库模式不明确并且数据结构需要被挖掘的时候（在Web应用中，这种情况是很普遍的），只采用面向对象语言中的路径表达式的概念是不够的。

Xpath使用查询机制扩展了路径表达式，扩展的方法就是将元素路径的一部分用搜索条件来替换。在检查数据的时候，XPath解释器会在运行时查找出路径缺少的部分。这种扩展路径表达式的想法并不是新的，这种想法最早出现在[Kifer and Lausen 1989，Kifer et al.1992，Frohn et al.1994]中的面向对象数据库的环境中，并在半结构化数据的研究中得以发展，这方面的著作有[Buneman et al.1996,Abiteboul et al.1997，Deutsch et al.1998，Abiteboul et al.2000]。XPath建立在这些概念的基础上，并成为许多XML扩展的重要基础。

1. XPath数据模型

XPath将XML文档看作是树，将元素、属性、注释和文本看作是树的节点。但这里有一点要说明，那就是树的根节点并不是XML文档的根元素。图17-11就阐明了这一点，它是下列XML文档对应的树：

```
<?xml version="1.0" ?>
<!-- Some comment -->
<Students>
    <Student StudId="111111111">
        <Name><First>John</First><Last>Doe</Last></Name>
        <Status>U2</Status>
        <CrsTaken CrsCode="CS308" Semester="F1997"/>
        <CrsTaken CrsCode="MAT123" Semester="F1997"/>
    </Student>
    <Student StudId="987654321">
        <Name><First>Bart</First><Last>Simpson</Last></Name>
        <Status>U4</Status>
        <CrsTaken CrsCode="CS308" Semester="F1994"/>
    </Student>
</Students>
<!-- Some other comment -->
```

（该文档是图17-4的报告文档的一个片段。）

注意，XPath树的根节点和对应着文档的根元素——Students元素的节点不同。从图中可以看出，我们显然需要指定特殊的根节点——这个特殊的根节点可以将文档的所有部分包括在内，包括允许出现在根元素以外的地方的注释。

和一般的树一样，除了根节点以外，所有的节点都有一个父节点。节点P出现在另一节点C的上一层，它就是节点C的父节点，而节点C则是节点P的子节点。然而，Xpath规范有一个重要而有时又令人迷惑的特殊情况：属性不是它的父节点的子节点。也就是说，如果C是P的一个属性，那么P是C的父节点，但C不是P的子节点。正是因为XPath这个特殊的术语可能造成迷惑，我们采用树结构的标准术语，认为属性也是其父元素的一个子节点。为了避免概念混淆，当我们要强调子元素的特定类型时，有时也会将它们分别称为元素子节点（e-children）、属性子节点（a-children）和文本子节点（t-children）。例如，Name是Student的元素子节点，StudId是Student的属性子节点，而John是First的文本子节点。当我们既要指元素子节点又要指文本子节点时，我们称之为元素文本子节点（et-children）。同样，ta-children指的是文本属性

子节点，依次类推。

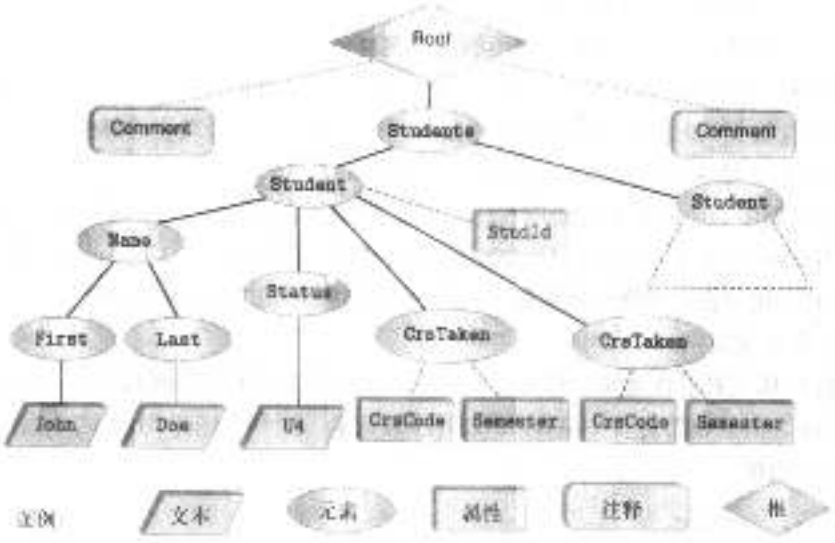


图17.11　XPath文档树

XPath数据模型提供一些操作来浏览文档的树结构并访问树的各个部分。这些操作包括访问根节点、访问节点的父节点、访问节点的子节点、访问节点的内容、访问属性的值等等。

讨论XML模式的约束的时候，我们已经介绍了部分操作，基本的语法是UNIX操作系统的文件命名机制：符号“/”表示根节点，“.”表示当前节点，“..”表示当前节点的父节点。XPath表达式以文档树为参数并返回树的节点集。因此，/Students/Student/CrsTaken是一个**绝对路径表达式**（absolute path expression），返回一组对节点的引用，这些节点对应CrsTaken元素，可以从根节点通过子节点Students以及孙节点Student到达该元素。文档树中共有三个这样的元素（上图中已经给出了其中的一个）。假设当前节点对应着元素Name，那么First和./First指的是同一个子元素。假设当前节点对应着元素First，那么../Last指的就是元素First的兄弟元素Last。这些都是**相对路径表达式**（relative path expression），因为它们都不是从根节点开始的。

为了能访问属性，我们使用符号“@”。比如，我们可以使用路径/Students/Student/CrsTaken/@CrsCode来获得上述XML文档中属性CrsCode的值的集合。在本例中，这样的CrsCode的值有两个：CS308和MAT123。文本节点可以使用text()函数来访问。比如，/Students/Student/Name/First/text()表示的是这样的节点集，其中每一个成员都表示一个First类型元素的文本内容。本例中，我们有两个这样的节点，一个是John，另一个是Batt。如果你要得到文档中的两个注释，那么你可以使用路径/comment()函数来实现。

2. XPath中的高级导航

XPath导航中的高级特性包括选择XML文档中的特定的节点和跳过不定数量的子节点。例如，要找出John Doe选修的第二门课，我们使用路径/Students/Student[1]/CrsTaken[2]，这里[1]选择文档树中两个Student节点中的第一个节点，路径/Students/Student[1]/CrsTaken选出第一个Student节点下所有的CrsTaken节点，最后，[2]指明选择的是这些节点中的第二个节点。

另一个例子用于选择某个特定的元素的表达式是/Students/Student/CrsTaken[last()]，它的作用机制和上面的例子一样，/Students/Student/CrsTaken选出所有的CrsTaken节点，而[last()]则选出按标准的XML节点顺序排序的节点中的最后一个（标准的XML节点顺序是按文档树深度优先遍历各节点形成的顺序）。在本例中，它得到的结果是

```
<CrsTaken CrsCode="CS308" Semester="F1994"/>
```

有时候，我们不知道文档的确切结构，或者指定准确的导航路径比较麻烦，因此XPath提供几种通配符机制。一种是descendant-or-self操作（用“//”表示），例如，路径//CrsTaken选择的是整个树中所有的CrsTaken元素。本例中，它的结果和/Students/Student/CrsTaken产生的结果是一样的。然而，如果XML文档的不同深度和不同元素类型下都有CrsTaken元素，那么不使用通配符机制来选出所有这样的元素是很困难的，也是不明智的。

descendant-or-self操作也可以用在相对路径中。例如，.//CrsTaken将在当前节点的所有子孙节点中寻找CrsTaken元素[⊖]。XPath同样允许访问给定节点的祖先节点（父节点、祖父节点等等），这里我们不介绍这一特性。

通配符*可以让我们选择一个节点的所有元素子节点（不管类型如何）。例如，Student/*将选出当前节点的子节点Student的所有元素子节点（本例中，当前节点是Students节点，上述路径将选出2个Name节点、2个Status节点和3个CrsTaken节点），而表达式/*//*将选出根的所有元素孙子节点，以及孙子节点的所有元素后代节点。

通配符*也可以用于属性。例如，CrsTaken/@*将选择当前节点下一层的CrsTaken节点的所有属性。注意，通配符*不能包括文本节点（如，Student元素子节点中的文本节点）。要选择Student节点的文本子节点，我们需要使用表达式Student/text()。

3. XPath查询

我们特别关心的是XPath中使用查询来选择节点的相关特性。XPath查询语句允许将一定的查询条件放在路径表达式的任何一个步骤中。下面我们会给出关于图17-4的报告文档的几个带有查询的XPath表达式。

下面的路径表达式查找所有选修1994年下半年课程的学生列表。

```
//Student[CrsTaken/@Semester = "F1994"]
```

这里我们使用了符号“//Student”，它选出根节点的所有后代节点Student。“[]”中括起来的表达式叫做**选择条件**（selection condition），它仅仅选择那些特定的Student节点，在这些Student节点上能够应用路径表达式CrsTaken/@Semester并且表达式返回的集合中包括F1994[⊖]。为了基于元素内容而不是属性来选择元素，我们可以使用下面的表达式

```
//Student[Status = "U3" and starts-with(.//Last, "P")
        and not(.//Last = .//First)]
```

从这个例子中我们可以了解到XPath的几个特性。第一，选择条件可以是一个由and、or和not

⊖ 注意，./CrsTaken和CrsTaken是一样的。但是.//CrsTaken、CrsTaken和//CrsTaken是不同的。第一个表达式返回当前节点的所有CrsTaken后代，第二个表达式只返回当前节点的CrsTaken子节点，第三个表达式返回文档中所有CrsTaken元素。

⊖ 注意，如果Student节点有几个CrsTaken子节点，那么路径表达式CrsTaken/@Semester返回一个节点集合。

组合而成的表达式。第二，基于子节点或后代节点的内容进行节点查询时，只要将路径表达式等价地替换为另一个表达式或常量即可。第三，XPath语言有丰富的函数库，该函数库提供的各种函数极大地提高了XPath路径表达式的表达能力。关于这些函数的详细介绍，大家不妨参阅XPath的规范[XPath 1999]。

在上例中，我们使用函数starts-with选择姓的第一个子母是P的学生。简言之，上面的例子选择所有状态为U3、姓的第一个子母是P且姓和名不同的学生。在其他的字符串操作函数中，有的用于判断包含问题，有的用于执行字符串连接，有的用于判断字符串的长度等等。例如，下面的查询能找出所有名字中包含van的学生。

```
//Student[contains(concat(Name//text()), "van")]
```

这里，Name//text()返回Name元素下的所有文本节点，而连接函数将这些文本连接为一个文本，即将学生的姓和名连接起来，然后再检查姓名中是否包含字符串“van”。

XPath语言中定义的聚合函数有sum()和count()。例如，下面的例子选出至少选修了5门课的学生名单:

```
//Student[count(CrsTaken) &gt;= 5]
```

在本例中，CrsTaken返回当前节点（必须为Student节点）的CrsTaken类型的元素子节点集合。因此，count(CrsTaken)返回的是某个Student的所有CrsTaken的个数，这个数将和5进行比较。符号“>=”表示“≥”。这样表示是由于“>”和“<”必须被转译为“&gt”和“&gt”，因为这些符号是作为标记的分隔符而被保留的。

值得注意的是，选择条件可以出现在路径表达式的各个不同的层次，也可以多次出现在路径表达式中，因此，下面的表达式是合法的:

```
//Student[Status="U4"]/CrsTaken[@CrsCode="CS305"]
```

这个表达式选择文档中所有的CrsTaken元素，但它必须满足两个条件，即它的父节点Student的属性Status必须为U4以及它的属性CrsCode的值必须为CS305。

路径表达式的同一层中也可以有多个选择条件，正如下面的表达式那样。下面的表达式找出所有在1994年秋天选修了课程MAT123的学生:

```
//Student[CrsTaken/@CrsCode="MAT123"]
         [CrsTaken/@Semester="F1994"]
```

这个表达式等同于:

```
//Student[CrsTaken/@CrsCode="MAT123"
          and CrsTaken/Semester="F1994"]
```

or运算也可以在条件表达式中使用，如CrsTaken/@CrsCode=“MAT123” or CrsTaken/@Semester=“F1994”也是合法的。

```
//Student[CrsTaken/@Grade]
```

选择条件还有另一种有趣的形式。假设Grade是CrsTaken的一个可选的属性，那么表达式//Student[CrsTaken/@Grade]将选出所有这样的Student节点，其子节点CrsTaken的属性Grade必须存在。同样，

```
//Student[Name/First or CrsTaken]
```

则选出所有这样的Student节点，它要么有孙子节点First，要么有子节点CrsTaken。

最后，在SQL中可以进行UNION和EXCEPT等代数查询操作。作为一门简单的语言，XPath只允许union操作，该操作用“|”表示，如下面的表达式所示：

```
//CrsTaken[@Semester="F1994"] | //Class[Semester="F1994"]
```

这个查询所选择的结果是所有属性Semester的值为“F1994”的CrsTaken节点和所有子节点Semester的值为“F1994”的Class节点的组合。这个例子也表明了路径表达式返回的集合可以包含不同类型的元素。

4. XPointer——一种智能的URL

虽然XPath拥有这么多的特性，但它并不是一种表达能力强大的语言。它不能表达联结操作，且只适合于树结构文档的导航。然而，正是它的这一缺点，使得XPath成为许多XML应用的嵌入部件。在17.3.5节中，我们就在XML模式中使用了XPath来表达约束。XPath还经常用简单的查询机制来丰富URL。为此，人们已经提出了许多XPath的扩展，这些扩展将在W3C组织即将推出的XPointer中得以标准化。

XPointer由URL和XPath发展而来。为了理解XPointer的使用方式，我们假设某篇文档中需要创建一个能连接到另一篇文档特定位置的超链接。这种链接在现在的HTML文档中很常见（一个典型的例子就是内容的目录，当点击了目录中的一个链接，用户就能进入相应的章节）。在HTML中，假如需要链接到一篇文档的中间部分，那么就将这个特殊部分用锚点标识[⊖]，比如，被链接的部分是interesting-place，那么使如下的URLdocument-url#interesting-place就能将用户引导到interesting-place部分。问题是，只有文档的作者预先创建好适当的锚点后，我们才能创建这样的链接，但一般来说，外部的文档浏览者是不能随便在文档的某些感兴趣的地方标记锚点的。

这就是要引入XPath的原因了。如果我们将URL和路径表达式能够连接起来使用，那么上述问题就得到解决了。浏览者可以先使用URL检索到这篇文档，然后使用路径表达式找出需要链接到的地方。这就是XPointer语言，其表达式的一般形式是：

```
someURL#xpointer(XPath表达式1)xpointer(XPath表达式2)…
```

这种表达式的处理过程为：首先找出Some URL指定的文档，然后根据它计算出XPath表达式1，如果它返回的是一个非空的树节点集合，那么该地址就是需要链接到的锚点，否则计算XPath表达式2，如果它仍返回一个空集合，那么计算第三个XPath表达式，以此类推。举个例子，假设图17-4的文档的URL为http://www.foo.edu/Report.xml，那么下面的表达式直接链接到第二个学生信息：

```
http://www.foo.edu/Report.xml#xpointer(//Student[2])
```

如果没有XPath的查询节点的语言能力，那么XPointer语言的功能就要减半。比如，有了XPath，很容易地就能找到选修了1994年秋天开设的MAT123课程的学生信息：

```
http://www.foo.edu/Report.xml#
        xpointer(//Student[CrsTaken/@CrsCode="MAT123"
                           and CrsTaken/@Semester="F1994"])
```

⊖ 在HTML中，这是利用标记<A name = "…">指定的。

（上面的表达式应该写在同一行，它之所以占据了好几行是因为版面的限制。）

总结一下，我们介绍了XPointer对XPath的几种扩展，其中还有一些扩展这里没有介绍，比如指定文档节点的范围，感兴趣的读者可以参考[XPointer2000]。

17.4.2 XSLT: XML的一种转换语言

XSLT（XSL Transformation）是一种转换语言，也是XSL（XML的可扩展样式语言）的一部分。它的最初目的只是将XML文档转换为HTML格式，以便用常见的浏览器查看文档。事实上它可以把XML源文档转换成任何类型的文档（HTML、XML以及纯文本）。因为XSLT具有这种能力，所以它可以用来查询XML文档。

作为一种查询语言，XSLT不同于本书中已经提到过的其他查询语言。第6章描述的关系代数是一种**命令式**（imperative）语言，它的查询是一系列为了得到结果而必须执行的有序操作。第6章和第7章中提到的SQL和QBE是基于谓词逻辑的子集——关系演算的，因此它们属于描述性语言（有时也称为**逻辑编程**语言）。第16章讨论的ODMG的面向对象查询语言OQL也属于这一类。

在这些语言中，用户必须给数据库源指定必需的信息和信息之间的联系，系统从数据库源中获得这些信息。另一方面，XSLT是使用XML语法的**功能性**程序设计语言(functional programming language)。和SQL类似（和关系代数不同），用户间接指定需要的结果，但是，这个结果是用一个递归函数集而不是基于逻辑的语言（如关系演算）来指定的。基于功能性程序设计的查询语言几乎和那些基于代数和逻辑的查询语言一样古老[Shipman 1981]。然而，在XSLT之前，它们在数据库世界里很难得到一块立足之地。

XSLT只是XSL的两个组成部分之一。另一个是*格式化语言*（formatting language），它指定了一个文档在浏览器或纸上显示时的外观。显示一个XML文档的主要过程是将该XML文档转变为另一个文档，这个文档在原来的XML文档基础上增加了显示方面的指令。结果可能丢失一些原始内容（例如，为使文档能在诸如Palm Pilot的PDA中使用），也可能会得到一些额外信息（例如，一个内容目录）。在使用了XSL格式化语言后，输出文档将是可以被一些浏览器或者文字处理器显示的XML格式。而XSLT可以被单独使用，来把输入的XML文档转换成一种完全不同的格式，例如HTML（可以用常见的浏览器显示）或LaTex（适用于高质量的论文文档）。

格式化不是本节要讨论的主题。我们将着重讨论XSLT，尤其是它作为查询语言的使用。此外，我们会关注XSLT查询功能中最吸引人之处——基于模式的文档转换，读者可以在[Kay2000, XSL1999]中查到XSLT的详细信息。

1. XSLT基础

一个XSLT程序（通常称为样式表）指定了从一种类型的文档到另一种类型的转换。像以前一样，我们将通过图17-4中的报告文档来说明它的各种特性。为了处理一个文档，XML处理器需要知道相应的样式表的位置。这个位置由xml-stylesheet处理指令来指定，它必须出现在文档的**序言**（preamble）部分，在初始的<?xml…?>指令和第一个XML标记之间。例如，为了给我们的报告提供一个样式表，文档开头应该如下所示:

```
<?xml version="1.0" ?>
<?xml-stylesheet type="text/xsl"
                 href="http://xyz.edu/Report/report.xsl" ?>
<Report Date="2000-12-12">
.
.
.
</Report>
```

xml-stylesheet指令中的type属性指明样式表是一个XSL文本文件，所以XML处理器能够选择合适的解析器。href属性指明样式表的位置。处理器就可以从指定位置取得样式表并且相应地转换文档格式。

下面是一个样式表的例子，它可以从一个报告（例如图17-4中的报告）中提取出所有学生的列表:

```
<?xml version="1.0" ?>
<StudentList xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
             xsl:version="1.0">
    <xsl:copy-of select="//Student/Name"/>
</StudentList>
```

我们的报告在使用这个样式表定义的转换后结果如下:

```
<StudentList>
    <Name><First>John</First><Last>Doe</Last></Name>
    <Name><First>Joe</First><Last>Public</Last></Name>
    <Name><First>Bart</First><Last>Simpson</Last></Name>
</StudentList>
```

一个样式表由显示格式样例（它是在样式表中定义的标记，在此例中是StudentList）和**XSLT指令**（例如copy-of）组成。样例被简单地复制到结果文档中，而指令从源文档中提取数据项，并将其放置到结果文档中。注意，当把我们的样式表看作XML文档时，StudentList是它的根标记。和大多数XML文档一样，样式表的根标记包含命名空间的声明。StudentList也包含XSLT需要的version属性（XSLT强制的属性和命名空间声明不复制到结果文档中）。

xmlns声明将前缀xsl（XSLT惯用的一个前缀）和定义XSLT词汇表的标准命名空间关联起来。这个词汇表包括copy-of和其他我们马上就会看到的XSLT指令。由于一些原因，我们不能把XSLT的命名空间作为默认的命名空间。如果这样做，StudentList将属于这个命名空间（根据命名空间声明的范围规则）。然而，这个标识符在命名空间http://www.w3.org/1999/XSL/Transform中并不存在，事实上，它只是在此例中作为结果文档的顶层标记而创造的。

copy-of语句是一个XSLT指令，其功能只是复制其select属性中的XPath表达式选出的元素。在我们的例子中，路径表达式选择所有Student的子节点的Name元素，产生了上面显示的Name元素的列表。

XSLT包含一些条件指令和循环指令，例如if（一个没有“else”的“if-then”）、choose（C/C++和Java中的switch语句的增强版本）和for-each。下面是一个使用了其中一部分特性的例子:

```
<?xml version="1.0" ?>
<StudentList xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
             xsl:version="1.0">
```

```
    <xsl:for-each select="//Student">
      <xsl:if test="count(CrsTaken) &gt; 1">
        <FullName>
          <!-- Last is two levels below Student; use * to skip one level -->
          <xsl:value-of select="*/Last"/>,
          <xsl:value-of select="*/First"/>
        </FullName>
      </xsl:if>
    </xsl:for-each>
</StudentList>
```

对我们的报告应用这个转换后的结果如下:

```
<StudentList>
   <FullName>
     Doe, John
     Public, Joe
   </FullName>
</StudentList>
```

for-each语句选择了被相应路径表达式指定的所有节点（在我们的例子中是所有Student元素的集合），并且对这些节点应用所有出现在for-each元素中的语句。对每个学生而言，这个程序首先检查这个学生是否修完多于一门的课程（像前面讨论的那样，我们必须把“>”表示为“>”）。我们的文档中只有两个学生满足这个条件: John Doe和Joe Public，对于每个人，显示格式样例的标记<FullName>…</FullName>被从样式表复制到结果文档。

下一步，我们打破报告中Name元素的结构，从中提取出姓和名，丢弃掉标记。学生名以无结构的文本形式显示。抽取一个元素的文本内容是使用XSLT指令value-of达到的，它类似于copy-of，不同之处在于它只提取元素的内容而不提取元素本身。如果我们使用copy-of而不是value-of，那么将会有不同结果:

```
<StudentList>
   <FullName>
     <Last>Doe</Last>, <First>John</First>
     <Last>Public</Last>, <First>Joe</First>
   </FullName>
</StudentList>
```

2. XSLT模板

XSLT的过程化特性看似强大，但它们并不足以处理所有的XML文档。问题在于尽管for-each指令可以迭代路径表达式选择的所有元素，但是它不能递归访问下面的元素并用递归方法转换文档。因为XML文档可以是任意的深度，所以结构的递归遍历对提取希望的信息是必需的。对XML树的遍历由基于模式的模板（pattern-based template）提供。先前描述的StudentList样式表使用了一种简单的语法，这种语法是下面这个模板的简化:

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xsl:version="1.0">
    <xsl:template match="/">
        <StudentList>
            <xsl:for-each select="//Student">
            .......
```

```
            </xsl:for-each>
        </StudentList>
    </xsl:template>
</xsl:stylesheet>
```

事实上，在这里老的样式表用<xsl:template match="/" >…</xsl:template>标记对包了起来，它们说明标记对之间的显示格式样例部分和样式表中的指令都应该应用在文档根节点下。

一个XSLT模板是一个转换函数，经常根据其他的模板递归定义。在模板内部，它可以使用我们已经见过的过程化特性（如copy-of、for-each等），也可以调用其他的模板。有趣的是，模板通常不直接被调用，而是当它们的XPath表达式与文档树中当前节点匹配时才被调用[⊖]。图17-12显示了一个与先前的样式表功能相同的模板程序，而上个模板程序是用for-each语句实现的。尽管新的样式表比原来的样式表复杂，但是它也足够简明并且能阐明主要的思想。

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xsl:version="1.0">
  <xsl:template match="/">
    <StudentList>
      <xsl:apply-templates/>
    </StudentList>
  </xsl:template>
  <xsl:template match="//Student">
    <xsl:if test="count(CrsTaken) &gt; 1">
      <FullName>
        <xsl:value-of select="*/Last"/>,
        <xsl:value-of select="*/First"/>
      </FullName>
    </xsl:if>
  </xsl:template>
  <xsl:template match="text()">
    <!-- Empty template -->
  </xsl:template>
</xsl:stylesheet>
```

图17-12　递归样式表

计算开始时将图17-11中文档树的根节点和每个模板中match属性指定的路径表达式进行匹配。在我们的例子中，匹配根的唯一路径表达式是第一个模板中的"/"。这个模板首先发布StudentList标记对，设置**当前**节点为树的根节点，然后用apply-templates指令递归调用其他的模板。这个调用的结果会插入到StudentList标记对之间。

apply-templates指令驱动对文档树的递归遍历。它构造了当前上下文节点的所有元素文本子节点（忽略属性节点）集合，并且对每个子节点应用一个匹配的模板[⊜]。

根节点唯一的元素文本子节点是Students元素，因为我们的样式表中没有一个模板匹配此节点，所以我们好像进入了一个死胡同。为避免这种情况，我们可以指定下面的模板:

⊖ XSLT也有已命名的模板，它们是显式调用的。但我们不讨论这种机制。

⊜ 这些模板应用时互不相关，但是结果按照子节点在XML树中出现的顺序输出。

```
<xsl:template match="*|/">
    <xsl:apply-templates/>
</xsl:template>
```
(17.2)

这个模板匹配所有的e-node（因为“*”）和根节点（因为“/”）。幸运的是，在没有其他模板匹配当前元素节点时，XSLT会自动调用这个默认的模板，所以我们不需要显式指定它（文本节点和属性节点的默认模板是不同的，下面会解释这一点）。上面的默认模板不发布任何东西，只是继续对报告的元素文本子节点（Students、Classes、Courses）应用模板。这个递归过程在遍历文档树的Classes和Courses分支时会进入死胡同，但是在遍历Students分支时会产生有用的结果。

让我们先考虑一下对Students分支的转换。因为我们的样式表中没有模板匹配Students，所以会再次应用默认规则。它引导XML处理器对三个Students子元素应用匹配的模板。很幸运，我们这次有了一个相匹配的显式模板——样式表的第二个模板。（这个模板的实际转换与我们前面讨论的使用了for-each指令的样式表相同。）

对Classes和Courses两个分支的转换的过程相似，所以我们只考虑Classes分支。因为没有模板显式匹配Classes，所以XSLT应用默认的模板，它又对元素文本子节点应用模板。注意，默认的模板忽视属性和文本节点。然而，当它应用到CrsCode和Semester节点时（Classes的子孙），它们的文本子节点将会受到apply-templates语句的影响。这个问题是由于XSLT还有一个可以应用到属性和文本节点的默认模板而造成的:

```
<xsl:template match="text()|@*">
    <xsl:value-of select="."/>
</xsl:template>
```

这意味着，当这个模板匹配一个属性或文本节点时，属性或文本值被处理器复制到结果文档中。但是，在我们的例子中，它产生了我们不需要的表示学期和课程代码的错误文本（我们只需要学生的名字）。

第三，加入图17-12的样式表中的空模板可以解决这个问题。这个模板只匹配文本节点并且什么也不发布（在这里我们不需要考虑属性，因为<xsl:apply-templates>忽视所有当前节点的属性子节点）。因为我们有一个显式的规则匹配文本节点，所以就不会再使用这些节点的默认规则。

因为XSLT中有大量可用的特性，所以对文档应用样式表的算法非常复杂，连W3C提出的标准[XSL 1999]都没有关于它的详细描述。这里我们提供一个用于简化的基于模式的模板的计算过程的概要，这些简化的模板至多在结果文档增加一个元素节点并且可能调用apply-templates。特别地，我们去掉了for-each循环。

1) 转换首先建立结果文档的根。在此过程中把源文档的根复制到结果树中，并且使它成为结果文档根节点的一个孩子。源文档的根被设置成*当前节点*（Current Node，CN），并且*当前节点列表*（Current Node List，CNL）被初始化为只包含源文档的根节点。

2) CNL保存源文档中将要应用模板的节点集。在算法中，CN始终是CNL的第一个节点。当有一个节点N被放置在CNL中时，它的副本（我们记作N^R）就会被放置在结果文档树中。之后，虽然节点N^R可能被删除，或者被一个模板的应用替换，但是这个节点起了标记符的作用，它表明接下来结果树的哪部分会发生变化。

3) 程序找到CN的**最优匹配模板**（best-matching template）并且把它应用到CN。最优匹配模板就是其路径表达式返回的源文档节点集最小并且当前节点也包含在这个节点集中的模板[⊖]。如果没有模板的路径表达式返回包含当前节点的集合，那么就会调用合适的默认模板。

4) 应用模板能带来下列改变：

a. 结果文档树中的CN^R可以被一棵子树替换。举个例子，假设CN是图17-11的Students节点，那么图17-12的样式表中的最优匹配模板是

```
<xsl:template match="//Students">
    <StudentList>
        <xsl:apply-templates/>
    </StudentList>
</xsl:template>
```

在这个例子中，CN^R被替换成StudentList元素节点并且源树中CN的每个元素文本子节点被复制到结果树中成为StudentList的孩子。这个转换过程见图17-13a。

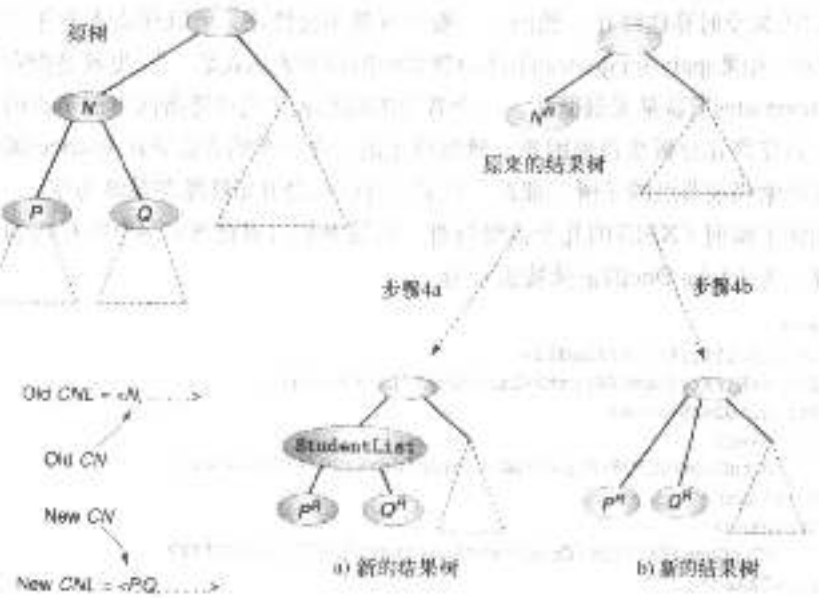


图17-13 apply-templates对文档树的作用

b. 结果文档树中的CN^R可以删除，它的父亲可以导到其他的孩子。举个例子，图17-12中样式表的默认规则

```
<xsl:template match="*|/">
    <xsl:apply-templates/>
</xsl:template>
```

删除了CN^R。源树中CN的元素文本子节点被复制到结果树中成为CN^R父节点的孩子。这个转换见图17-13b。

在这两种情况中，如果CN没有元素文本子节点，那么CNL会变短成为模板应用的结果。

⊖ 如果有这样的几个模板，就使用其他的规则来找到其中最优的模板，参见[XSL 1999]。

否则（如果它没有这样的子节点），CNL可能会变大，因为CNL中的CN会被它的元素文本子节点的列表替换。这些孩子在结果树中和在CNL中放置的顺序与在源树中的顺序相同。特别的，CNL的第一个节点成为新的CN。

一般来说，apply-templates指令有一个select属性:

```
<xsl: apply-templates select = "some path expression"/>
```
(17.3)

事实上，<xsl: apply-templates/>是<xsl: apply-templates select= “node()” />的简写，node()是一个XPath函数，它返回给定的元素节点的元素文本子节点集。例如，<xsl: apply-templates select= “@*|text()” />可以应用在属性（因为通配符“@*”）和文本（因为函数text()）节点上，但是不能应用到元素上。

一般来说，转换（17.3）把CN替换成select属性指定的节点，select属性不是必需的，默认情况是替换成CN的元素文本子节点。就像在情况a和b中解释的那样，这些节点也被复制到结果树中。算法继续进行第 3 步。

5) 当CNL为空时算法终止。然而，一般它可能不会终止，所以样式表的作者要确保它能够终止。例如，如果apply-templates的select属性中的路径表达式是“.”（也就是指向当前节点），那么apply-templates明显是无限循环。一个普通的路径表达式可能指向当前节点的父亲、祖先或者兄弟，这使终止分析变得很困难。确保终止的一个简单的方法让所有select属性中的路径表达式只能搜索当前节点的子树。那么，在某点时CNL会开始收缩并最终为空。

下面的例子阐明了XSLT的几个高级特性。假设我们打算把图17-4中的报告的属性都替换成元素。那么关于John Doe的记录被重写为:

```
<Student>
    <StudId>111111111</StudId>
    <Name><First>John</First><Last>Doe</Last></Name>
    <Status>U2</Status>
    <CrsTaken>
        <CrsCode>CS308</CrsCode><Semester>F1997</Semester>
    </CrsTaken>
    <CrsTaken>
        <CrsCode>MAT123</CrsCode><Semester>F1997</Semester>
    </CrsTaken>
</Student>
```

此外，我们打算写一个样式表，使它不必知道源文档的确切属性名称。这样，如果以后Grade属性被加入到CrsTaken元素中，我们的程序仍能工作，可以把<CrsTaken...Grade="A"/>转换成

```
<CrsTaken> ··· <Grade>A</Grade></CrsTaken>
```

为达到这个目的，样式表必须能发布预先不知道名字的元素，这些元素的名字在运行时通过源文档中的属性计算出来。XSLT的两个新特性使这一目标变成了可能。

- element指令使样式表把一个具有给定名字的元素复制到结果文档中。例如:

```
<xsl:element name="name(current())">
    I do not know where I am
</xsl:element>
```

上述代码会在执行的时候根据当前节点的不同复制不同的元素。例如，如果在执行时，当前节点是一个名字为foo的属性，这条指令会把下面的元素复制到结果中：

```
<foo>
    I do not know where I am
</foo>
```

在这个样式表中，current()是一个运行时在任何特定点返回当前节点的XSLT函数。函数name()是一个返回节点集（被传递给函数作为参数）中第一个节点名字[⊖]的XPath函数。在本例中，在执行过程中，参数始终是当前节点，所以函数返回当前节点的名字。

- copy指令在运行过程中的特定点输出当前节点。不要把这个指令和copy-of指令混淆，copy-of指令输出的是它的select属性返回的节点集。

此外，copy-of指令复制节点及其所有从属（属性、孩子元素等），而copy指令不返回当前节点的属性和孩子[⊜]。这个特性很重要，因为它提供了对输出的控制程度，特别是使样式表可以截取属性节点并把它们作为元素发布。

copy指令也不同于current()函数。因为current()函数不是一个指令，它不能复制任何内容到结果文档中。相反，作为一个函数，它返回当前节点（包括它内部的所有内容），并且返回结果可以被XSLT指令使用。

我们问题（找到一个把属性转换成元素的样式表）的一个可行的解决方法如图17-14所示。这个样式表有两个模板。源文档的处理和平常一样从根节点开始。因为没有显式的模板匹配根，所以调用合适的默认模板。像前面看到的那样，默认模板（17.2）把CNL上的CN替换成它的元素子节点列表并且让处理器对列表应用匹配的模板。在图17-11的文档中，根节点唯一的孩子是Students元素，所以处理器试图找到一个与之相匹配的模板。

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xsl:version="1.0">
    <xsl:template match="node()">
        <xsl:copy>
            <xsl:apply-templates select="@*"/>
            <xsl:apply-templates/>
        </xsl:copy>
    </xsl:template>
    <xsl:template match="@*">
        <xsl:element name="name(current())">
            <xsl:value-of select="."/>
        </xsl:element>
    </xsl:template>
</xsl:stylesheet>
```

图17-14 把属性转换为元素的XSLT样式表

我们的样式表中只有第一个模板匹配：XPath函数node()匹配文档树中的除了根和属性节点以外的所有节点。（第二个模板选择了匹配路径表达式@*的文档节点，@*匹配当前节点的

⊖ 对于元素节点，节点名是标记名。对于属性节点，节点名是属性名。

⊜ 当然，如果当前节点是文本节点或属性，就不需要跳过什么，直接把整个节点复制。

每个属性。）

第一个模板把当前节点N复制到结果中。回想一下，copy指令去掉了孩子和属性，所以只有N被复制。对N的属性和元素文本子节点的处理由copy指令中的两条apply-templates指令实现。

第一个apply-templates指令把N（当前节点）的属性加入到CNL的开头。CNL的第一个节点成为新的当前节点CN。假设N有属性，只有第二个模板匹配CN，所以使用此模板。它输出与当前属性具有相同名字的元素（因为模板只对属性应用，所以current()一定返回属性节点），并且把当前属性的值作为这个元素的内容。（例如，Attr="something"被转换成<Attr>something</Attr>。）

我们对copy指令中的第二个apply-templates指令已经很熟悉了。它对当前节点的每个元素文本子节点应用模板，但不对属性节点应用。和以前一样，只有第一个模板匹配这些节点，所以它们被复制，它们的属性被转换成元素，然后重复过程。

3. XSLT的局限

XSLT具有一些使其成为数据库查询语言的特性：它能从一个文档中提取出节点集合，重新整理它们，通过递归遍历文档树来应用转换。然而，XSLT在一些重要方面仍有欠缺，这使它在作为查询语言使用时有一定局限。

最重要的问题与文档间或同一文档各部分间的联结有关，这和关系数据库中的联结操作类似。作为一个简单的例子，考虑利用图17-4中的记录产生一个学生记录列表，在列表中给学生选的每门课程附上课程名。为此，我们需要关联具有相同课程号的CrsTaken元素和Course元素。

事实上，要用XSLT表达这样的联结查询是很麻烦的。一种方法是下降到Student元素，对每个CrsTaken使用带有select属性的apply-templates，select中的路径指向对应的Course元素。尽管不自然，但这种方法是可行的，只是需要使用XSLT变量（在本节中没有讨论这种机制）。另一种方法是使用for-each循环嵌套，并用XSLT变量保存文档节点集。这种方法与在嵌入式SQL中把联结操作显式地编码成嵌套循环来查询数据库的方法相似。

幸运的是，XSLT不是XML查询的最新成果，还有一些其他的XML查询语言已经开发出来了。

17.4.3 XQuery: XML的一个功能完善的查询语言

XPath和XSLT提供了对XML文档的一定的查询能力。然而，我们已经发现XPath被设计成轻量级的，所以只能表达简单的查询。XSLT有更强的表达能力，但它不是作为一种查询语言来设计的，所以，它在处理复杂的查询时有些困难。

在众多的为查询XML而设计的语言中，应该关注一下XQL和XML-QL。XQL[Robie et al.1998]是XPath的一个扩展，XML-QL[Deutsch et al.1998,Florescu et al.1999]是一种SQL式的查询语言，它同时借鉴了其他一些语言的思想，例如OQL（参见第16章）和Lorel[Abiteboul et al.1997]。最近，XQL和XML-QL的优秀特性被组合成为另一种叫做XQuery的语言，它是目前W3C对XML的正式的查询语言[XQuery 2001]的设计基础。像XSLT一样，XQuery使用了XPath作为它路径表达式的语法。然而，XQuery在查询的时候通常比XSLT更加简洁和透明。

本节将介绍XQuery的主要特性。

1. 选择和连接

和XSLT不同，XQuery不使用XML冗长的语法，因为没有理由让任何查询语言都这样做[⊖]。相反，XQuery语句跟SQL有一些相似:

```
FOR 变量声明
WHERE 条件
RETURN 结果
```

FOR语句的作用与SQL中的FROM语句作用相同，WHERE语句也同SQL中的WHERE语句功能一致。RETURN语句类似于SELECT: 在SQL中，它定义结果关系的模板；在XQuery中，它指定结果文档的模板。

更深入地说，XQuery受ODMG数据库（参见第16章）的面向对象查询语言OQL影响很深。这种联系可以从例子中看出来。我们从对图17-15的文档的简单查询开始，此文档在URL http://xyz.edu/transcripts.xml 中。

第一个查询查找所有选修过课程MAT123的学生。

```
FOR $t IN document("http://xyz.edu/transcripts.xml")//Transcript
WHERE $t/CrsTaken/@CrsCode = "MAT123"
RETURN $t/Student
```

FOR语句声明了一个变量$t和它的范围——一个文档节点集合。这个集合由XPath表达式“//Transcript”指定，这个XPath表达式将应用在用函数document()得到的文档上。document()函数借鉴自XSLT，它返回URL指定的文档的根。所以$t的范围是文档中的所有Transcript节点。XQuery依赖于用变量扩展了的XPath表达式来导航文档树，所以，$t/CrsTaken/@CrsCode 和 $t/Student是扩展后的XPath表达式。当$t绑定在文档树的一个Transcript节点时，第一个表达式返回那些对应于这个节点的元素子节点CrsTaken的CrsCode属性的节点。第二个表达式返回这个节点的元素子节点Student。

WHERE条件选出了Transcript节点的一个子集，这个集合中每个节点都有一个CrsCode属性值为MAT123的CrsTaken元素作为孩子节点。变量$t被轮流绑定到每个Transcript节点上，RETURN语句分别执行$t的每个绑定。每次执行输出结果文档的一部分，在本例中是包含在$t当前值的Transcript节点中的Student元素。下面是本例的输出:

```
<Student StudId="111111111" Name="John Doe"/>
<Student StudId="123454321" Name="Joe Blow"/>
```

这里，我们的问题是查询没有产生结构良好的XML文档。它产生了一个Student元素的列表，但是没有把它们包含在一个父元素中。把上面的查询嵌入一对标记中就可以轻松地解决这个问题。

```
<StudentList>
(
    FOR $t IN document("http://xyz.edu/transcripts.xml")
                          //Transcript
```

⊖ 至少，不是给人类使用的。然而，XQuery中基于XML的语法正在发展中。首先，它可以简化为XQuery构建解析器的工作。

```
    WHERE $t/CrsTaken/@CrsCode = "MAT123"
    RETURN $t/Student
)
</StudentList>
```

结果是FOR语句逐个输出Student元素，并且把它们作为StudentList节点的孩子。

在下面的例子中，当最顶层的标记对的作用仅仅是给查询结果提供一个根元素时，就像上例中的StudentList，我们就会省略它。

下面的例子说明了XQuery的结构重组能力。图17-15的文档Transcripts将学生选过的课程放在相应的学生周围。然而，阅读这个文档的读者可能想为每门课程重新构建班级列表。换句话说，对某个学期的开设的每门课程，他可能想得到选修这门课的学生列表。在XQuery中，这可以直接用图17-16的transcripts.xml来得到。在此查询中，变量$c的范围是transcripts.xml文档中所有不同的CrsTaken节点集合。对每个这样的节点，将会像RETURN语句中描述的那样建立结果文档的一个片段。

```
<?xml version="1.0" ?>
<Transcripts>
    <Transcript>
        <Student StudId="111111111" 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="987654321" Name="Bart Simpson"/>
        <CrsTaken CrsCode="CS305" Semester="F1995" Grade="C"/>
        <CrsTaken CrsCode="CS308" Semester="F1994" Grade="B"/>
    </Transcript>
    <Transcript>
        <Student StudId="123454321" 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="023456789" Name="Homer Simpson"/>
        <CrsTaken CrsCode="EE101" Semester="F1995" Grade="B"/>
        <CrsTaken CrsCode="CS305" Semester="S1996" Grade="A"/>
    </Transcript>
</Transcripts>
```

图17-15　http://xyz.edu/transcripts.xml上的成绩单

图17-16中的查询几乎是正确的，但是它有一个缺点，这个问题我们在讨论了这个查询中展示的新特性后再作解释。

首先注意，这个查询是嵌套的。然而，有趣的是嵌套出现在RETURN语句中，它对应的是SQL中的SELECT语句。回想一下，SQL在SELECT语句中禁止嵌套查询（如果嵌套查询返回多于一个的元组），因为嵌套查询在关系模型中没有意义。的确，一个嵌套查询通常返回一

组元组，然而SQL中SELECT语句的目标列表是一个元组的模板。与在SQL中相反，RETURN语句中的嵌套查询对XML查询语言极有意义，因为这里的目的是构造一个具有任意复杂嵌套结构的文档。在本例中，嵌套查询的目的是把学生列表嵌入到班级花名册中。我们已经见过在一个查询的目标列表中使用嵌套查询的情况，这在面向对象查询语言OQL（见查询（16.11））中是允许的。在那里使用嵌套的原因和在XQuery中使用嵌套的原因是相同的。

```
FOR $c IN distinct(document("http://xyz.edu/transcripts.xml")
                                    //CrsTaken)
RETURN <ClassRoster CrsCode=$c/@CrsCode Semester=$c/@Semester>
    (
        FOR $t IN document("http://xyz.edu/transcripts.xml")
                                    //Transcript
        WHERE $t/CrsTaken/@CrsCode = $c/@CrsCode
                AND $t/CrsTaken/@Semester = $c/@Semester
        RETURN
                $t/Student
                SORTBY($t/Student/@StudId)
    )
</ClassRoster>
SORTBY($c/@CrsCode)
```

图17-16 从transcripts构造班级花名册：第一次尝试

我们再仔细看看图17-16中查询的RETURN语句，它对于变量$c的每个值分别执行。首先，它构造ClassRoster元素并为CrsCode和Semester属性设置适当的值。然后调用嵌套子查询来构造这个班级（即对给定学期中的给定课程）的学生的有序列表。其中，$t涵盖Transcript元素，然后WHERE语句选出的那些特定的Transcript元素，这些Transcript中有一个CrsTaken元素与$c中指定的学期和课程相匹配。输出由如下元素组成：

```
<ClassRoster CrsCode="CS305" Semester="F1995">
    <Student StudId="111111111" Name="John Doe"/>
    <Student StudId="987654321" Name="Bart Simpson"/>
</ClassRoster>
```

在每个花名册中，这些元素已经根据CrsCode属性排好序了。

到现在为止，一切都很好，只是John Doe和Bart Simpson在1995年秋天的CS305中得到了不同的分数，所以FOR语句为那个班级把$c绑定在用一个课程的两个不同的CrsTaken元素上。

```
<CrsTaken CrsCode="CS305" Semester="F1995" Grade="A"/>
<CrsTaken CrsCode="CS305" Semester="F1995" Grade="C"/>
```

这意味着上面的ClassRoster元素将输出两次。一般来说，每个花名册对相应的班级中每个不同的分数输出一次。解决这个问题的方法是建立一个包含所有班级列表的新文档，然后把$c绑定在列表的元素上。这个任务可以很容易完成，方法是从Transcripts文档中选出所有的CrsTaken元素并去掉Grade属性。

我们等会再来考虑这个想法，现在我们通过假设已经存在图17-17所示的文档来避开这个问题，此文档在URL http://xyz.edu/classes.xml。下面的查询是将图17-16那个有缺点的查询稍加改动而形成的，目的是说明XQuery中的联结操作：

```
FOR $c IN document("http://xyz.edu/classes.xml")//Class
RETURN
    <ClassRoster CrsCode=$c/@CrsCode Semester=$c/@Semester>
        $c/CrsName
        $c/Instructor
        (
            FOR $t IN document("http://xyz.edu/transcripts.xml")
                                    //Transcript
            WHERE $t/CrsTaken/@CrsCode = $c/@CrsCode
                  AND $t/CrsTaken/@Semester = $c/@Semester
            RETURN
            $t/Student
            SORTBY($t/Student/@StudId)
    )
</ClassRoster>
SORTBY($c/@CrsCode)
```

```
<?xml version="1.0" ?>
<Classes>
    <Class CrsCode="CS308" Semester="F1997">
        <CrsName>Market Analysis</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>
```

图17-17　http://xyz.edu/classes.xml上的班级

新查询中的改变是变量$c的取值范围变为图17-17的文档中所有Class节点的集合。因为在这里班级不会出现多次，所以我们不再有输出同一个花名册的多个实例的问题（我们甚至不需要应用distinct()函数）。新查询的结果为每个花名册增加了课程名（$c/CrsName）和教师（$c/Instructor）作为子元素。这样，这个查询在图17-15文档的Transcript元素和图17-17文档的Class元素之间在CrsCode和Semester属性上做了*等值联结*。像17.4.2节结尾提到的那样，XSLT难以实现这种格式转换。

注意，在上面的查询中，即使有些班级没有学生，对应的ClassRoster元素仍然会在结果中出现，只是内容为空。这与关系数据库中的联结操作不同，在关系数据库中，CLASS关系

中的一个元组如果在TRANSCRIPT关系中没有匹配元组，那么不会考虑在CrsCode和Semester属性上做等值联结。上面XQuery例子中的联结在关系数据库中相当于外连接（在第6章的练习6.9中定义）。不过，在最外层的FOR语句中加入一个WHERE语句以使其实现“真正的”联结是很简单的:

```
FOR $c IN document("http://xyz.edu/classes.xml")//Class
WHERE document("http://xyz.edu/transcripts.xml")
                    //CrsTaken[@CrsCode = $c/@CrsCode
                                and @Semester = $c/@Semester]
RETURN
    <ClassRoster CrsCode=$c/@CrsCode Semester=$c/@Semester>
        $c/CrsName
        $c/Instructor
        (
            FOR $t IN document("http://xyz.edu/transcripts.xml")
                                    //Transcript
            WHERE $t/CrsTaken/@CrsCode = $c/@CrsCode
                  AND $t/CrsTaken/@Semester = $c/@Semester
            RETURN
                  $t/Student
                  SORTBY($t/Student/@StudId)
    )
</ClassRoster>
SORTBY($c/@CrsCode)
```

新WHERE语句的目的是测试http://xyz.edu/transcripts.xml处的文档是否有与用于变量$c当前值的CrsCode和Semester属性匹配的CrsTaken元素。这个测试由第一个WHERE语句的路径表达式“//CrsTaken[…]”实现。只有当这样的CrsTaken元素存在时，对应的ClassRoster元素才会输出。所以，没有学生的班级花名册不会在结果中出现。

2. XQuery的语义

到目前为止，我们已经讨论了不同的例子，却还没有解释实际查询计算机制是如何工作的。我们现在来解释这些问题。

FOR语句有如下功能:

- 指明查询中使用的文档。
- 声明变量。
- 把每个变量绑定到由XQuery表达式指明的文档节点的有序集合上。通常，这个XQuery表达式是一个XPath表达式，但是，正如我们即将看到的，它也可以是一个能够返回一个节点列表的查询或者函数。

FOR语句中产生的绑定被翻译成一个有序的元组列表，每个元组含有FOR语句中提到的每个变量的具体绑定。举个例子，如果FOR语句声明变量$a和$b，并且分别把它们绑定到文档节点{*v*,*w*}和{*x*,*y*,*z*}上，那么就会产生有序元组列表{*v*,*x*}，{*v*,*y*}，{*v*,*z*}，{*w*,*x*}，{*w*,*y*}，{*w*,*z*}。每个元组（比如{*w*,*x*}）提供一个具体的绑定（$*a*/*w*，$*b*/*x*）给我们的变量。

下一步，由WHERE条件对绑定的元组进行筛选。因此，如果条件是$a/CrsTaken/@CrsCode = $b/Class/@CrsCode，并且相应的文档节点w和x满足w/CrsTaken/@CrsCode = x/Class/@CrsCode，则$a和$b的绑定元组{*w*,*x*}就会被保留，否则，它会被丢弃。WHERE语

句的作用是选出原来元组列表的一个有序子列表。

最后，对每一个保留的元组绑定，RETURN表达式都会实例化一次。结果是创建输出文档的一部分。重复这个过程，直到所有合格的元组绑定被处理完为止。

3. 过滤

改正图17-16的查询来建立班级花名册的最简单方法是使用XQuery的filter()函数。这个函数有两个参数，每个参数都是一个文档节点集合。第一个参数中的每个节点代表一个文档片段（一个以这个节点为根的子树），第二个参数中每个节点代表它本身（也就是不包括它的孩子节点）。

filter()通过删除所有在第二个参数中没有出现的节点来裁剪作为第一个参数的文档片段。这些文档树中剩余的节点被互相连接起来以保留原来的祖孙关系，并且返回文档片段的结果集。例如，下面的filter()操作应用在图17-17的文档上[⊖]。

```
filter(//Class, //Class|//Class/CrsName)
```

会产生下面的文档片段：

```
<Class><CrsName>Market Analysis</CrsName></Class>
<Class><CrsName>Electronic Circuits</CrsName></Class>
<Class><CrsName>Database Systems</CrsName></Class>
<Class><CrsName>Transaction Processing</CrsName></Class>
<Class><CrsName>Algebra</CrsName></Class>
```

注意，尽管节点集“//Class”包含在filter()表达式的第二个参数中，但是它的属性节点没有显式出现，所以这些属性不会在输出中出现。同样，因为元素子节点Instructor没有显式出现，所以它也没有出现在结果中。

回想一下图17-16中导致查询问题的原因。由于Grade属性，变量$c可能会被绑定到两个不同的CrsTaken元素上，但是这两个元素的CrsCode和Semester却有相同的值。filter()函数能排除Grade属性，这样就去掉了$c的无关绑定：

```
LET $trs := document("http://xyz.edu/transcripts.xml")//Transcript
LET $ct := $trs/CrsTaken
FOR $c IN distinct(filter($ct, $ct | $ct/@CrsCode | $ct/@Semester))
RETURN
    <ClassRoster CrsCode=$c/@CrsCode Semester=$c/@Semester>
       (
        FOR     $t IN $trs
        WHERE   $t/CrsTaken/@CrsCode = $c/@CrsCode
                AND $t/CrsTaken/@Semester = $c/@Semester
        RETURN
                $t/Student
                SORTBY($t/Student/@StudId)
       )
</ClassRoster>
SORTBY($c/@CrsCode)
```

上面的查询和图17-16中查询的本质不同是使用了filter()函数和LET语句。LET语句只是把所有Transcript节点集赋值给了变量$trs，把Transcript节点的子节点CrsTaken集合赋值给了

⊖ “|”在路径表达式中表示表达式的并。

变量$ct。这么做减少了对$c进行绑定的语句长度，并且简化了第二个FOR语句。换句话说，如果我们把出现$trs和$ct的地方替换成它们的值，那么我们可以去掉LET语句。

然而，filter表达式

```
filter($ct, $ct | $ct/@CrsCode | $ct/@Semester)
```

并不是一个语法上的优点。这里的第一个参数是所有的CrsTaken元素集（包括它们的孩子），第二个参数的三个选择部分对第一个参数的集合进行了剪除，只保留了元素节点本身和它的两个属性，去掉了Grade属性（CrsTaken没有元素子节点，所以没有其他要剪除的）。结果是一个元素列表，如:

```
<CrsTaken CrsCode="MAT123" Semester="F1997">
<CrsTaken CrsCode="CS305" Semester="F1995">
.
.
.
```

所以，剪除之后，元素

```
<CrsTaken CrsCode="CS305" Semester="F1995" Grade="A"/>
<CrsTaken CrsCode="CS305" Semester="F1995" Grade="C"/>
```

变得等同了，所以在调用了distinct()函数后只有一个能保留下来。这正是我们在图17-16的查询中想得到的。

4. 用户定义的函数

XQuery提供了大量的内置函数，包括所有XPath中可用的核心函数。它也提供了其他一些有用的函数，如distinct()和document()，以及其他我们下面将会看到的函数。

更吸引人的是，XQuery中一个的查询中可以定义许多函数，这些函数可以在FOR-WHERE-RETURN主查询中被调用。函数可以递归地调用它们自身；它们可以把单独的节点或节点集作为参数；它们可以返回基本类型、文档节点或者这些类型的集合。函数体是一个通用的XQuery表达式（甚至查询也可以当成是表达式）。一个表达式可以处理整数、元素、元素列表等等，然后函数返回它的结果。

下面是一个函数的例子，这个函数计算以$e为根的文档片段中子孙元素和文本节点的个数:

```
FUNCTION countNodes(AnyElement $e) RETURNS integer {
    RETURN
        IF empty($e/*) THEN 0
        ELSE sum(countNodes($e/*)) + count($e/*)
}
```

这个函数定义阐明了以下一些特性:

- 声明AnyElement $e说明函数的参数必须是一个元素。它不能是一个属性或文本节点，也不能是整数。函数会返回一个整数。我们在后面会解释这些类型的来源。

  XQuery函数体是一个**XQuery表达式**，它将会计算出一个值（整数、字符串、文档节点、节点列表等等）表。这个表达式的值被返回。下面会解释XQuery表达式。
- IF-THEN-ELSE语句是一个XQuery条件表达式。在IF和THEN之间必须是一个布尔表达式——在本例中是empty($e/*)，它用内置函数empty()检查路径表达式$e/*是否返回文档

节点的一个空集[⊖]。

THEN和ELSE部分必须是XQuery表达式。这里我们没有定义XQuery表达式的完整语法[⊜]。一般它们会是路径表达式（返回一个文档节点集）、函数调用（返回一个基本类型，比如integer或string，一个文档节点，或者是调用用户定义函数情况下一个节点集）、算术表达式、IF-THEN-ELSE条件表达式或完整的查询（返回一个文档或者返回文档片段的列表）。

在本例中，THEN表达式是一个常数，ELSE表达式是一个调用聚合函数sum()和count()的算术表达式。利用递归调用countNodes()（下面会解释）得到的数字值，sum()函数将对它们进行求和。count()函数对路径表达式$e/*返回的节点进行统计。

- 递归调用countNodes()很有趣。注意，在签名中，这个函数的参数是一个元素。可是，似乎与这里的声明矛盾，这个调用应用到会返回文档节点集的路径表达式。这不是错误。

XQuery中一般规定，在这种情况下，函数会对集合中每个元素应用一次，返回每次应用的结果组成的集合。　　(17.4)

所以，countNodes()被应用到它的参数集中的每个文档节点，并且返回一组整数，这组整数由内置聚合函数sum()求和并传递给算术表达式。

在下个例子中，我们再看一看图17-16中“几乎正确”的查询。回想一下，其中问题是如果一个班级中至少有两个学生得到了不同的分数，那么结果中班级花名册就会出现多次。我们提出的一个解决方案是把Transcripts与图17-17中的文档做联结。用XQuery函数可以建立一个与图17-17相似的中间文档，并且在查询中把它和Transcripts做联结，这样可以不依赖其他外部文档。解决方法见图17-18。

查询的第一个部分定义了函数extractClasses()。这个函数接受一个元素，并返回一个元素列表。返回的类型是用LIST(AnyElement)指定的，其中LIST是一个列表类型构造器，AnyElement是所有元素的类型。函数结果由一个简单查询得到，它遍历函数参数的所有CrsTaken子孙，略过Grade属性，把Class元素列表作为结果输出。

函数定义下面是主查询。它返回一个顶层标记是Rosters的文档，顶层标记包含用ClassRoster标记的不同元素的列表。这个查询与图17-16的查询几乎相同，只是变量$c的范围变成了由extractClasses()函数产生的结果。同时，WHERE语句也被去掉，替换成了与变量$t1绑定的XPath表达式上的选择条件。（XPath表达式上的选择条件在17.4.1节中介绍。）

用户定义函数的最后一个例子说明XQuery如何实现在讲述XSLT时讨论的文档转换。我们重写了图17-14的XSLT样式表，这个样式表可以遍历一个XML文档，并且把属性换成名称和内容相同的元素。该程序的等价查询如下：

```
FUNCTION convertAttribute(AnyAttribute $a) RETURNS AnyElement{
  LET $name := name($a)
  RETURN
    <$name>value($a)</$name>
```

⊖ 回忆一下，通配符“*”选择当前节点的所有元素子节点，所以$e/*返回赋给$e的节点的子节点的所有元素集。

⊜ 有兴趣的读者可以参考[XQuery 2001]。

```
}
FUNCTION convertElement(AnyElement $e) RETURNS AnyElement{
  LET $name := name($e)
  RETURN
    <$name>
      convertAttribute($e/@*)
      IF empty($e/*) THEN $e/text()
      ELSE convertElement($e/*)
    </$name>
}

RETURN convertElement(document("....")/*)
```

```
  FUNCTION extractClasses(AnyElement $e)
                    RETURNS LIST(AnyElement){
    FOR $c IN $e//CrsTaken
  RETURN <Class CrsCode=$c/@CrsCode Semester=$c/@Semester/>
  }

  <Rosters>
  (
    LET $trs := document("http://xyz.edu/transcripts.xml")
    FOR $c IN distinct(extractClasses($trs))
    RETURN
      <ClassRoster CrsCode=$c/@CrsCode Semester=$c/@Semester>
        (
          FOR $t1 IN  $trs//Transcript[CrsTaken/@CrsCode=$c/@CrsCode and
                                       CrsTaken/@Semester=$c/@Semester]
          RETURN
            $t1/Student
            SORTBY($t1/Student/@StudId)
        )
      </ClassRoster>
  )
  </Rosters>
```

图17-18. 使用用户定义的函数来构造班级花名册

这个查询包含一个对预先定义的函数convertElement()的调用，这个函数的参数是一个元素节点。在本例中，参数是文档根的孩子节点[⊖]。注意，因为具有良好结构的XML文档只有一个顶层元素，所以没有必要用FOR语句这样的迭代结构[⊖]。

第一个函数convertAttribute()把属性节点作为参数并且把它转换成同名的元素（名字通过调用XPath函数name()得到）。属性值（通过XQuery函数value()得到）则作为元素的内容。

第二个函数convertElement()是这个查询的主要部分。它用来把一个元素节点转换成无属性的同名元素。对给定的节点来说，convertElement()输出同名标记对（又用到了name()函数），然后转换元素属性，最后处理它的元素文本子节点。为转换属性，它对元素的属性列表调用

⊖ 我们忽略了根可能有其他孩子的可能性，例如注释和处理指令。

⊖ 即使文档的根有多个元素子节点，但（17.4）使我们无需使用FOR语句。

convertAttribute()函数[⊖]。因为convertAttribute()的参数是一个属性，所以函数会分别对每个属性应用，并产生一个元素列表。这是按照XQuery的一般惯例（17.4）完成的。

处理完属性之后，convertElement()转向处理文本节点和元素。如果元素没有元素子节点（由empty()函数决定）[⊜]，那么唯一的孩子一定是文本节点（如果元素为空，那么什么也没有）。在第一种情况下，如果文本节点存在就将它发布；在第二种情况下，它通过递归调用自身来转换当前节点的元素子节点。一个元素列表再一次传递给参数为一个元素的函数，所以对convertElement()的调用输出了转换后的元素列表。

上面的查询与图17-14的XSLT样式表产生的转换不完全一样。例如，如果一个元素具有混合内容（它们孩子既包含元素又包含文本节点[⊜]），那么查询略过文本（因为XPath表达式$e/*只选择元素子节点），并转换元素。我们在解释了XQuery、XML Schemas和命名空间的关系后再说明如何解决这个问题。

5. XQuery和数据类型

前面的查询中展示了像integer、AnyElement和AnyAttribute这些基本类型的用法。然而，通过与XML Schema规范的紧密结合，以及允许引入和使用XML Schema中定义的类型，XQuery能发挥更大的作用。事实上，前面使用的基本类型integer不是XQuery中固有的类型，而是在XML Schema规范中定义的，所以它应该和对应的命名空间共同使用。同样地，AnyElement和AnyAttribute类型也一定是在某个模式中定义的。简单地说，我们把它们当成XML Schema规范的一部分。我们也假装AnyText类型（包含文档树中的所有文本节点）是在标准XML Schema命名空间中定义的。

下面是一个例子，它说明XQuery与XML Schema和命名空间的结合，并且提供了一个真正等价于图17-14中XSLT样式表的XQuery来完成了前面的例子。首先，我们需要定义一个新的简单数据类型，它可以接受属性、元素和文本节点。我们假设这个数据类型在http://types.r.us/auxiliary的命名空间中描述。

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://types.r.us/auxiliary">
    <simpleType name="AnyNode">
        <union memberTypes="AnyElement AnyAttribute AnyText"/>
    </simpleType>
</schema>
```

假设这个模式存储在URL http://types.r.us/auxiliary/types.xsd文档中。图17-19显示了我们正在寻找的XQuery表达式。

第一个语句schema告诉XQuery处理器去哪里寻找查询中使用的模式。在本例中，这个模式包含联合类型AnyNode的定义，我们在函数convertNode()中使用了这个类型。两个NAMESPACE语句引入了查询中使用的命名空间。命名空间XMLSchema使处理器知道查询中的AnyElement、AnyAttribute和AnyText类型是在XML Schema规范中定义的基本类型，而其

⊖ 回忆一下，@*是一个返回元素的所有属性的通配符。XPath函数text()返回当前节点的所有文本子节点。

⊜ 在这种情况下，元素有这种<foo> some text </foo>格式。

⊜ 例如，<foo> some text <bar> more text </bar> even more </foo>有混合内容，有两个文本子元素和一个元素子元素。

他的数据类型则不是。另一个命名空间auxiliary把AnyNode和上个模式中定义的简单类型关联在一起。注意，这个命名空间和定义AnyNode的模式文档中的目标命名空间相匹配。

```
SCHEMA "http://types.r.us/auxiliary/types.xsd"
NAMESPACE aux = "http://types.r.us/auxiliary"
NAMESPACE xsd = "http://www.w3.org/2001/XMLSchema"
FUNCTION convertNode(aux:AnyNode $n) RETURNS aux:AnyNode {
    LET $name := name($n)
    RETURN
        IF $n INSTANCEOF xsd:AnyElement THEN {
            <$name>
                convertNode($n/@*)
                convertNode($n/node())
            </$name>
        } ELSE IF $n INSTANCEOF xsd:AnyAttribute THEN {
            <$name>
                value($n)
            </$name>
        } ELSE $n
}

RETURN convertNode(document("....")/*)
```

图17-19　与图17-14的样式表功能相同的XQuery转换

查询中唯一的新特性是操作符INSTANCEOF，它测试一个给定的节点（本例中是$n的值）是否属于一个给定的类型。我们用这个操作符在函数convertNode()中测试这个函数的参数是否是属性节点或元素。

convertNode()函数的工作原理类似于先前我们提过的convertElement()。它先检查参数类型。如果是一个元素，就创建合适的标记对。在标记之间，convertNode()被递归调用，从而把当前元素的属性转换成元素；然后被调用来转换当前节点的所有元素文本子节点[⊖]。如果参数是属性，那么属性值被输出，其中具有与当前属性同名的标记对。如果参数既不是属性也不是元素，那么它一定是文本节点，在这种情况下就简单的把节点复制到结果文档中。

6. 分组和聚合

和SQL不同，XQuery没有使用单独的分组操作符；取而代之的是，它使用一种更为可靠的机制，这种机制应用聚合函数来显式地构造文档节点的集合。这是利用我们已经熟悉的LET语句来实现的，LET语句声明了一个变量并且用一个由XQuery表达式指定的节点集来初始化变量。有趣的是，就像我们在前面看到的例子那样，在FOR语句范围外使用LET语句仅仅是语法上的优点，但是我们必须把它放在FOR语句的范围内，因为别的方法不能实现分组。

为了说明这个问题，下面的查询利用图17-15的Transcripts文档产生了一个文档，新文档列出了学生和目前为止该学生选修过的课程数。

```
FOR $t IN document("http://xyz.edu/transcripts.xml")//Transcript,
    $s IN $t/Student
LET $c := $t/CrsTaken
```

⊖ 回忆一下，node()是一个返回当前节点的所有元素文本子节点的XPath函数。

```
RETURN
    <StudentSummary StudId=$s/@StudId Name=$s/@Name
                    TotalCourses=count(distinct($c))/>
SORTBY(StudentSummary/@TotalCourses)
```

FOR语句通过把$t绑定在文档的每个Transcript元素上来实现迭代。这里的窍门是，每当$t绑定到一个新的元素上时，变量$c被赋予一个新的CrsTaken元素列表，因为LET是在绑定$t的FOR语句范围内发生的。换句话说，对$t的每个Transcript绑定，$c被绑定在那个transcript元素提到的班级列表上，随后调用count()来计算列表中不同的元素数。

FOR语句和LET语句之间的相似性使它们容易被搞错。它们都是把变量绑定在一个文档节点集上，但是FOR把变量逐个绑定在集合中的节点上，而LET一次就把变量绑定在整个节点集上。

下一个查询稍微复杂一点，因为它涉及到图17-15中Transcripts文档与图17-17中Classes文档的联结。它创建了一个班级列表，每个班级带有它的平均分数。为得到分数的数字值，我们使用numericGrade()函数，这个函数可以很容易地定义成XQuery函数（这里省略了）。这个例子也说明了另一个重要的特性：LET语句（或FOR语句）中的变量绑定不一定需要指定为XPath表达式，但可以是任何返回一列节点的XQuery表达式。特别地，它可以是下面的查询：

```
FOR $c IN document("http://xyz.edu/classes.xml")//Class
-- $g gets the collection of all numeric grades in the class bound to $c
LET $g := (
            FOR $ct IN document("http://xyz.edu/transcripts.xml")
                                        //CrsTaken
            WHERE $ct/@CrsCode = $c/@CrsCode
                  AND $ct/@Semester = $c/@Semester
            RETURN numericGrade($ct/@Grade)
          )
RETURN
    <ClassSummary CrsCode = $c/@CrsCode Semester=$c/@Semester
                  CsrName = $c/CrsName Instructor=$c/Instructor
                  AvgGrade = avg($g)/>
SORTBY(ClassSummary/@CrsCode)
```

这本质上与前面介绍的通过联结transcripts.xml和classes.xml来构造班级花名册的查询相同。然而，在这个例子中，我们并没有在结果文档中列出班级所有的学生，而是计算出班级的分数列表并用LET语句把它赋值给变量$g。然后，求分数的平均值，而后把结果当作属性AvgGrade的值。

注意，当LET语句在FOR语句的范围内出现时，它引入了一个关系到查询语义的新问题。这是因为FOR和LET语句都可以绑定变量造成的。LET语句用下面的方式合并到计算机制中。对FOR语句中变量的每个元组绑定，LET语句中变量绑定是确定的。和FOR变量的绑定不同，LET语句是绑定在一列节点上，这一列节点一般作为聚合函数的参数。因此，LET变量的绑定是完全由FOR变量的绑定决定的。

查询计算过程的其余部分保持不变：WHERE语句把FOR语句中产生的绑定元组过滤。现在唯一的不同是，WHERE中的条件也可以使用LET绑定的变量。最后，对FOR变量绑定的每个元组，RETURN语句为结果文档产生一个片段。

7. 量化

假设我们想查询选修课程MAT123的所有学生。在SQL中，我们需要使用EXISTS操作符（类似于XQuery中的empty()函数）。XQuery通过量词SOME和EVERY提供了相似的能力，它们对应于关系演算（参见第7章）中的存在量词（∃）和全称量词（∀）。第7章介绍过，显式地使用量词使查询的构造过程比SQL中查询的构造过程简化很多。

下面的查询使用了SOME量词来返回选修MAT123的学生列表。

```
FOR $t IN document("http://xyz.edu/transcripts.xml")//Transcript
WHERE SOME $ct IN $t/CrsTaken
        SATISFIES $ct/@CrsCode = "MAT123"
RETURN $t/Student
```

在很多情况下，查询中可以不用SOME量词。例如，假设一个学生不能选两个同样的课程，那么上面的查询可以改为

```
FOR $t IN document("http://xyz.edu/transcripts.xml")//Transcript,
    $c IN $t/CrsTaken
WHERE $c/@CrsCode = "MAT123"
RETURN $t/Student
```

但是，SQL和XQuery之间不是能完全对应的。考虑下面的查询，它返回至少有一个学生注册的所有课程名。SQL查询是

```
SELECT   C.CrsName
FROM     CLASS C, TRANSCRIPT T
WHERE    C.CrsCode = T.CrsCode AND C.Semester = T.Semester
```

XQuery查询是

```
FOR      $c IN document("http://xyz.edu/classes.xml")//Class,
         $t IN document("http://xyz.edu/transcripts.xml")//CrsTaken
WHERE    $c/@CrsCode = $t/@CrsCode AND $c/@Semester = $t/@Semester
RETURN   $c/CrsName
```

因为T没有出现在SELECT语句中，$t没有出现在RETURN语句中，所以假设这两者存在量化。在这两个例子中，即使只有一条匹配的transcript记录，课程名也会被输出。如果找到了多条记录，SQL查询处理器根据查询优化器的内部工作原理决定是否输出重复的课程名。（因为SQL是一个关系查询语言，它的语义意味着对每门课程名字只输出一次。重复的可能性是实现的一个细节。）相反，XQuery中的FOR语句的语义表示对每个匹配的transcript记录都输出一次课程名。这是因为，FOR指定了一个循环，在这个循环中，RETURN语句对每个$c和$t的绑定都要执行一次，而且对每个$c的绑定可能会有多个对$t的匹配绑定。

注意，在SQL中用SELECT DISTINCT可以确保没有重复，但在XQuery中使用distinct()函数却不能轻松地达到相同的目的（见练习17.30）。在本例中，消除重复的一种方法是使用distinct()和filter()函数（见练习17.31）。另一种方法是使用SOME。然而，不同于前面查询选修了MAT123的所有学生的例子，在这个例子中量词很关键，不能只是把变量从WHERE移动到FOR。

```
FOR    $c IN document("http://xyz.edu/classes.xml")//Class
WHERE
  SOME $t IN document("http://xyz.edu/transcripts.xml")//CrsTaken
  SATISFIES $c/@CrsCode = $t/@CrsCode AND $c/@Semester = $t/@Semester
RETURN $c/CrsName
```

与存在量词相反，全称量词EVERY在大多数查询中都不能被去掉。第7章讨论过，用全称量词可以自然地表达涉及关系代数中除法操作的查询。这样的查询在SQL中看起来很笨拙，因为SQL不直接支持全称量词[⊖]。为说明这一点，下面的例子查询其中所有学生都选修MAT123的班级：

```
FOR $c IN document("http://xyz.edu/classes.xml")//Class
-- $g gets bound to the set of all Transcript elements
-- corresponding to the particular class that binds $c
LET $g := (
     FOR $t IN document("http://xyz.edu/transcripts.xml")
                              //Transcript
     WHERE $t/CrsTaken/@CrsCode = $c/@CrsCode
          AND $t/CrsTaken/@Semester = $c/@Semester
     RETURN $t
          )
-- Take only those $g in which every transcript
-- has a CrsTaken element for MAT123
WHERE EVERY $tr IN $g
     SATISFIES NOT empty($tr[CrsTaken/@CrsCode = "MAT123"])
RETURN $c SORTBY($c/@CrsCode)
```

这里，每次把$c绑定到Class元素上时，$g就被绑定到选修这门课的学生的成绩单列表上。外层的WHERE语句检查是否$g中的每个学生都选修了MAT123。（回想XPath表达式$tr[CrsTaken/@CrsCode="MAT123"]返回一个非空节点集，条件是当且仅当$tr被绑定到一个含有与课程MAT123对应的CrsTaken元素的transcript元素上。）

17.4.4 小结

本节介绍了XML的三种查询语言。XPath是一种轻量级的语言，它已经成为一些XML相关技术的不可缺少的一部分。我们在本章中已经见到了一些这样的技术：XML Schema、XPointer、XSLT和XQuery。

XSLT是一种XML转换语言。它没被设计成一种查询语言，但是，因为它使用了XPath和它全面的设计，所以它适合很多种XML查询，特别是那些不涉及文档联结的查询。

XQuery是一种折中的语言，是建立在数据库世界中产生的思想上的。它还没有完全成熟，一些语法和语义都有可能改变。然而，即使在目前的形态上，XQuery也阐明了怎样把关系数据库和面向对象数据库中产生的一些思想应用到查询和重构文档上。

目前，XPath和XSLT都受到W3C的推荐，并且它们都已经得到大多数Web浏览器的支持。在写这本书时，XQuery的规范还只是一个初步的草稿[XQuery 2001]。

17.5 参考书目

XML试图给Web信息处理事务的混乱状态带来一些秩序。从概念上说，它是SGML标准[SGM 1986]的重写和简化。版本1在1998年通过，并且成为一种被广泛接受的标准[XML 1998]。和每个新的热门话

⊖ 我们在6.2.3节说明了在SQL中表示全称量词需要使用EXISTS、嵌套子查询和双重否定。然而，SQL:1999引入了全称量词的一种有限形式——FOR ALL操作符，它减轻了编写需要关系代数中除法操作符的查询的痛苦。

题一样，在短时间内出现了很多讨论它的出版物。因为其数量太多而不能一一列出这些出版物，所以我们只提及最近的两本著作：[Ray 2001，Bradley 2000a]。

XML Schema在很多书中都有所介绍，但是因为它到目前为止还尚未成熟，所以我们推荐原始资料[XMLSchema 2000a，XMLSchema 2000b]。

XPath是XML路径表达式语言，在大多数最近关于XML的著作中都有所描述。可以在[XPath 1999]中找到W3C的官方推荐标准。路径表达式的最初想法来源于[Zaniolo 1983]。利用查询能力增加路径表达式的想法在[Kifer and Lausen 1989, Kifer et al. 1992, Frohn et al. 1994, Abiteboul et al. 1997, Deutsch et al. 1998]及其他著作中均有提及。

XSLT在1999年成为W3C的官方推荐标准[XSL 1999]，并且大多数主流浏览器的最新版本都支持它，如Internet Explorer、Mozilla、Netscape。一些著作中含有XSLT的主题，如[Kay 2000, Bradley 2000b]。

XML查询语言XQuery在[XQuery 2001]中描述。它是一种折中的语言，建立在之前的SQL、OQL[Cattell and Barry 2000]、XQL[Robie et al. 1998]、XML-QL[Deutsch et al. 1998, Florescu et al. 1999]以及 Quilt[Robie et al. 2000, Chamberlin et al. 2000]等几种语言的思想之上。XQuery仍然处于不断的完善之中，了解XQuery进展的最佳地方是W3C XML 查询工作组的Web站点 http://www.w3.org/XML/Query。

最后，[Abiteboul et al. 2000]是一本优秀的参考书，它采用Web上信息处理的数据库观点，并且涵盖了半结构化数据的最新研究，包括Lorel和XML-QL（XQuery的两个先驱）。

17.6 练习

17.1 使用XML表示图4-2中STUDENT关系的内容。为这个文档定义一个合适的DTD。

17.2 设某文档包含图5-15的COURSE表和图5-16的REQUIRES表中的数据，为该文档定义一个合适的DTD。根据这些DTD写出相应的约束。给出一个符合你的DTD的文档。

17.3 重新组织图17-4的文档结构，用合适标记将元素Name、Status、CrsCode、Semester和CrsName全部替换成属性。给出适合这个文档的DTD。描述所有可用的ID和IDREF约束。

17.4 定义下列简单类型：

a. 这种类型的域由字符串列表组成，每个列表由7个元素组成。

b. 这种类型的域由字符串列表组成，每个字符串长度为7。

c. 这种类型的域是一个字符串列表集合，每个字符串有7 ~ 10个字符，并且每个列表有7 ~ 10个元素。

d. 这种类型适合用字母表示学生在完成课程之后取得的成绩——A、A−、B+、B、B−、C+、C、C−、D和F。用两种方式表示这种类型：用枚举型或用XML Schema 中的pattern标记。

17.5 使用XML Schema中的key语句来为图17-4的文档定义下面的键约束：

a. 所有Student元素集的键。

b. 所有Course元素集的键。

c. 所有Class元素集的键。

17.6 假设图17-4文档中任何学生都可以由last name和status唯一确定。定义这个键约束。

17.7 使用XML Schema的keyref语句为图17-4的文档定义下面的参照完整性：

a. CourseTaken元素中每个课程代码必须对应一门有效的课程。

b. Class元素中的每个课程代码必须对应一门有效的课程。

17.8 为图17-4的文档描述以下的约束：在相同的Student元素中，没有一对CourseTaken元素有相同的CrsCode属性值。

17.9 重新安排图17-4中Class元素的结构，使得该元素可以定义下面的参照完整性：在Class元素中涉及的每个学生Id对应同一文档中的一个学生。

17.10 编写一个统一的XML Schema来涵盖图17-15和17-17的文档。提供合适的主键和外键约束。

17.11 用XML Schema来表示图4-6中的关系模式片段，其中包括所有的键和外键约束。

17.12 用XPath来表示以下对图17-15文档的查询：

a. 查找所有Id以987结尾并且选修课程MAT123的Student元素。

b. 查找所有first name是Joe并且选修少于3门课程的Student元素。

c. 查找所有对应学期S1996并且属于名字以P开头的学生的CrsTaken元素。

17.13 为图17-17的文档构造下列XPath查询：

a. 查找在1995年秋季由Mary Doe教授的所有课程名字。

b. 查找所有在1996年秋季开设的课程名字对应的节点和所有教授MAT123的教师对应的节点。

c. 查找在1997年春季由John Smyth教授的所有课程的代码的集合。

17.14 用XSLT把图17-15中的文档转换成一个具有良好格式的XML文档，这个文档要包含所有选修1996年春季所开设课程的Student元素的列表。

17.15 用XSLT把图17-17中的文档转换成一个具有良好格式的XML文档，这个文档要包含在1997年秋季由Ann White教授的课程列表。输出中不要包含子元素Instructor和Semester属性。

17.16 编写一个遍历文档树的XSLT样式表，忽略属性、元素拷贝和文本节点。例如，<foo a="1">the<best quality="S"/>bar</foo>将被转换成<foo>the<best />bar</foo>。

17.17 编写一个遍历文档树的XSLT样式表，忽略属性、复制元素拷贝并使文本节点翻倍。例如，<foo a="1">the<best />bar</foo>将被转换成<foo>thethe<best/>barbar</foo>。

17.18 编写一个遍历文档树的XSLT样式表，保留所有元素、属性、文本节点，不过要去掉CrsTaken元素。样式表的构造不允许依赖CrsTaken元素在源文档中的位置。

17.19 编写一个遍历文档树的XSLT样式表，略过所有属性，保留树的其他的方面（元素节点和文本节点间的父子关系）。不过，当样式表碰上foobar元素时，它的属性和它下面的整个结构被保留，元素本身被重复两次。

17.20 编写一个遍历文档树的样式表，保留文档所有内容。但是，当它碰上foo元素时，要求把foo元素的每个文本子节点转换成名字为text的元素。例如：

```
<foo a="1">the<best>bar<foo>in the</foo></best>world</foo>
```

应该转换成

```
<foo a="1">the<best>bar<foo> <text>in the</text>
</foo></best>world</foo>
```

17.21 考虑图4-6的关系模式，假设Web服务器用下面的XML格式发布这些关系的内容：关系名是顶层元素，每个元组用一个tuple元素表示，每个关系属性用一个带有value属性的空元素表示。例如，STUDENT关系表示如下：

```
<Student>
    <tuple>
        <Id value="111111111"/> <Name value="John Doe"/>
        <Address value="123 Main St."/> <Status="U1"/>
    </tuple>
    .
    .
    .
</Student>
```

用XQuery表示下列查询：

a. 得到所有住在Main Street的学生的列表。

b. 找到CrsCode值与开设这门课程的院系Id不一致的所有课程。(例如，信息系统中的课程IS315可能是由计算机科学(CS)系开设的。)

c. 利用TEACHING关系，对所有秋季开设的课程的记录创建一个列表。

17.22 对练习17.21描述的文档结构，用XQuery写出下面的查询:

a. 为教授过MAT123的所有教授创建列表。其中必须包含PROFESSOR关系中的所有属性。

b. 为选修过John Smith的课并且成绩为A的所有学生（包括学生Id和name）创建列表。

c. 为Joe Public成绩为A的所有课程创建列表。

17.23 对练习17.21描述的文档结构，用XQuery写出下面的查询:

a. 建立一个XML文档，列出所有学生的名字，对每个学生，分别列出该生的成绩单。成绩单记录必须包括课程代码、学期和成绩。

b. 建立一个XML文档，列出每个教授的名字和他所教的课程。

c. 建立一个文档，列出每门课程和教授这门课的教授。课程信息必须包括课程名，教授信息应该包含教授的名字。

17.24 对练习17.21描述的文档结构，用XQuery写出下面的查询:

a. 列出选修的课程多于3门的所有学生。

b. 列出三门课程以为成绩为A的所有学生。

c. 列出有3个以上学生成绩为A的所有教授。

d. 列出平均成绩高于B的所有课程。

e. 列出所给平均成绩（每个学生所有成绩的平均值）等于或高于B的所有教授。对这个问题，我们必须写一个XQuery函数来比较用字母表示的成绩。

f. 列出平均成绩小于教这门课程的教授所给的平均成绩的所有课程。

17.25 对练习17.21描述的文档结构，用XQuery写出下面的查询:

a. 列出每个学生的成绩高于B或为B的所有课程。

b. 列出成绩不低于B的所有学生。

17.26 写一个XQuery函数来遍历一个文档，并计算出文档树的最大分支因子，即文档所有元素中的子节点（文本或元素节点）的最大数目。

17.27 写一个XQuery函数来遍历一个文档并去掉所有元素标记。例如:

```
<the>best<foo>bar</foo>in the world</the>
```

将被转换成

```
<result>bestbarin the world</result>
```

17.28 考虑一个文档，它包含一个教授列表（姓名、Id、院系Id）和每个教授教的课程列表（课程代码、学期)。使用聚合函数生成下面的文档:

a. 一个教授列表（姓名、Id)，包含该教授所教的课程数目。(不同学期的相同课程算一门课程。)

b. 一个院系列表（院系Id)，包含该院系的教授曾经上过的不同的课程数。(不同学期的相同课程或者相同学期由不同老师教的同一门课程都算作一门课程。)

17.29 用filter()函数重做练习17.28。

17.30 考虑下面的查询:

```
FOR     $c IN document("http://xyz.edu/classes.xml")//Class,
        $t IN document("http://xyz.edu/transcripts.xml")
            //CrsTaken
WHERE   $c/@CrsCode = $t/@CrsCode
AND     $c/@Semester = $t/@Semester
RETURN  $c/CrsName
```

解释一下为什么每个课程名都会输出一次以上。用distinct()函数能解决这个重复问题吗？解释你的答案。

17.31 考虑练习17.30中的查询。用filter()和distinct()来去掉输出中的重复部分。

第18章 分布式数据库

越来越多的应用要求访问多个在不同地点，甚至在地理上很分散的数据库。这些应用可以分为两大类，我们分别用例子来说明。

- 第一类：一个网络零售商建立了一个全国性的仓库网络来加快货品递送的速度。每一个仓库都有自己的本地数据库，而在该零售商的总部也有一个数据库。一个计算所有仓库的库存的应用在总部的数据上运行，并访问各个仓库的数据库。
- 第二类：当一个顾客从该网络零售商处购买物品时，这项交易可能涉及这个零售商和信用卡公司。关于交易的信息必须同时保存于零售商和信用卡公司两者的数据库中。

这两种应用都涉及分布式数据。不同之处在于它们在单独站点访问数据库的方式。在第一类中，应用是以模式（schema）来编写的，它能在SQL语句级访问数据库站点。因此，它可以给每一个仓库发送SELECT语句来获得它想要的信息，然后再合并返回的元组。

第二类应用使用不同的方法访问数据。零售商和信用卡公司是独立的企业，而且它们的数据库都含有双方不愿意共享的敏感信息。此外，双方都不愿意让对方（也许是无意的）在自己的数据库中引入不一致性。于是，信用卡公司提供一个子例程（也许是一段作为事务执行的存储过程）以供调用来更新数据库，记录对某一顾客账户的扣款。远程站点可以调用这个子例程，但是不能直接访问数据库。因为子例程是由信用卡公司编写的，所以其数据库的安全性和完备性就得到保证。和第一类应用相比较，直接远程访问信用卡公司的数据库是不可能的。

这两类应用都涉及分布式数据，而且如果应用要求有事务特性，则还涉及分布式事务。在第四部分和第15章的事务处理概论中我们已经讨论过事务的问题了。分布式系统的安全性问题将会在第27章中讨论。

只有第一类的应用可以直接访问数据，因此需要使用面向数据库的策略来提高性能和实用性。本章将讨论这些策略。例如：

- 应该如何设计分布式数据库？
- 各个数据项或表应该存储在哪个站点？
- 哪些数据项应该被复制，数据项备份应该存储在哪个站点？
- 如何处理访问多个数据库的查询？
- 分布式查询优化涉及哪些问题？
- 用于查询优化的技术如何影响数据库的设计？

为什么数据是分布的

既然分布带来新的问题，那么为什么要分布数据呢？为什么不把分布式企业的所有数据集中到一个中心站点呢？下面的理由（也许互相冲突的）解释了为何数据必须是分布的和它们应该放在哪里。

- 这样放置数据可以减少通信代价和响应时间，这通常意味着数据被放在最经常访问它的站点。
- 数据分布放置可以平衡工作量，使得单个站点不会因为过载而影响到吞吐量。
- 数据可以存放在产生它的站点，以便它的创造者保持对它的控制并保证安全性。
- 某些数据项会在多个站点复制以增加它们在系统崩溃时的可用性（如果一个备份不可用，那么其他的备份还可以被访问）或增加吞吐量和减少响应时间（数据可以通过本地或邻近的副本而被更快地访问）。

18.1　应用设计者对数据库的观点

直接访问数据库的应用提交根据某些模式构造的SQL语句。这些模式描述应用所看见的数据库的结构。我们来考虑三种这样的模式。

1. 多个本地模式

在应用程序看来，分布式数据库是一系列独立的数据库，每一个数据库都有自己的模式，如图18-1a所示。此类系统是**多数据库系统**（multidatabase）的一个例子（关于多数据库系统的完整讨论将在21.2.2节给出）。如果各个数据库管理系统都是由不同的厂商供应的，则称此系统是**异构的**（heterogeneous）；如果是由同一厂商供应的，则称此系统为**同构的**（homogeneous）。

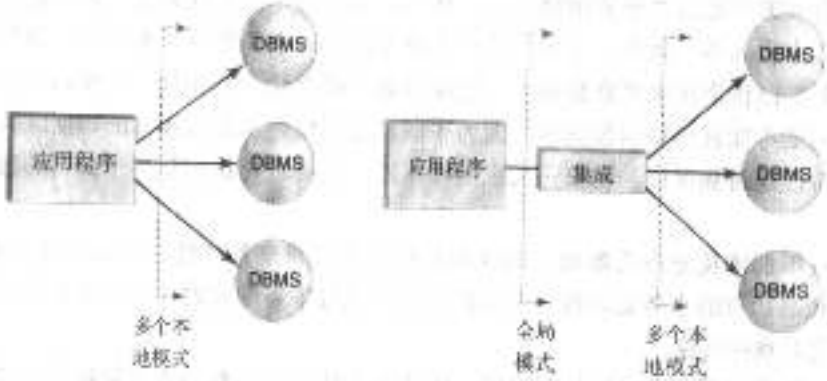


图18-1　分布式数据的视图

对每一个包含需要访问的数据项的站点，应用程序都必须明确地建立一个到此站点的连接。连接建立后，程序可以用根据此站点模式构造的SQL语句访问数据库。如果数据项从一个站点移到另一个站点，那么必须修改程序。

SQL不支持在单个语句中引用不同站点上的表，例如，进行全局的联结操作。如果应用程序要联结不同站点的表，它必须把每张表中的元组读入应用所在站点的缓冲区中（用不同的SELECT语句），并且显式地为每一对元组测试联结条件。

不同站点的数据可能是以不同格式储存的。例如，在一个站点，某个人的姓可能被存放在最前面，但在另一个站点，姓可能被存放在最后。此外，各个模式中的类型可能会不同。

例如，在一个站点，Id可能被存储为一串字符；而在另一个站点，Id却被存储为整数。在这些情况下，应用必须提供对话例程以便在运行时整合数据。

应用还必须管理副本。当要查询一个已被复制的数据项时，应用必须决定应该访问哪一个副本，如果这个数据项被更新，它必须保证所有的副本都被更新。

我们在第10章讨论的用嵌入式SQL、JDBC、SQLJ和ODBC访问分布式数据库的方法都是指这种模式的数据库。

2. 全局模式

本地模式的方案几乎对应用程序设计者不提供任何支持：所有因数据库的分布特性而产生的问题都必须通过程序来解决。另一个极端就是，所有的问题都被隐藏于应用之外且自动解决。这很有趣，因为它提供了衡量分布式数据库系统的一个理想状态。

在这个方法中，应用设计者看到的是一个综合了各本地模式的模式。因此，我们称这个模式是**全局模式**（global schema），并称这个系统是**集成**（integrated）分布式数据库系统。

集成的工作是由中间件完成的（见图18-1b），它将各个模式统一成一个包含所有站点数据的全局模式。全局模式可能含有不在任何本地模式中出现的表，但这种表可以由本地模式中的表用适当的SQL语句计算出来。换句话说，全局模式是本地模式的一个视图。

当全局模式中的元素被访问时，中间件会自动建立到各个站点的连接。因此，应用程序是不知道表所在的位置的（这称作**位置透明性**（location transparency））。如果数据项从一个站点移到另一个站点，那么全局模式可保持不变而且应用程序也不必修改。全局模式到本地模式的映射需要在中间件中修改，但这比修改众多的应用程序容易多了。

就像在本地模式中一样，不同站点的相关数据可用不同的格式或类型来储存，这可能与全局模式中的格式和类型不符。在这种情况下，中间件提供了转换例程以便整合系统。

一个与其相关的问题是**语义集成**（semantic integration），它至少涉及值转换和名转换的问题。考虑一个在欧洲、日本和美国都有站点的分布式数据库。货币值在所有站点都可用双精度数表示，所以不需要进行格式转换。但是，1000日元和1000欧元是不同的，1000欧元也不同于1000美元。因此，对东京的销售总额的查询可能需要按汇率转换成日元，同样对阿姆斯特丹的查询就需要转换成欧元。而属性名的转换必须根据各地的文化差异和习惯而定。即使不考虑阿姆斯特丹的站点和纽约的站点使用不同语言的可能性，我们还是必须处理两个不同的美国站点用item#和part#来表示同一属性的可能性。

应用程序根据全局模式来执行SQL语句。例如，应用可能会在全局模式中请求两个表T_1和T_2的联结。如果这两个表在同一个站点存储，那么语句会被送到该站点处理。如果它们在不同的站点存储（全局联结），那么中间件必须根据单个数据库的模式把联结翻译成一系列恰当的SQL语句并且为计算联结执行其他必要的操作。更复杂的情况是，T_1和T_2可以看作是由中间件用查询填充的，查询用来联结储存在不同站点的关系。我们将在18.3.1节讨论执行这些全局联结的技术。

应用设计者可能会复制一些数据项并为这些副本指定保存的站点。但是，对应用程序而言，复制是隐藏的。程序访问逻辑数据项，而中间件自动管理复制，提供适当的副本来满足查询并且在适当的时候更新所有副本。这被称为**复制透明性**（replication transparency）。

3. 限制的全局模式

应用设计者看到的是一个全局模式，但这个模式是各个数据库模式的并（相对于视图）。因此，限制的全局模式包含各数据库中所有的表。

一些同构系统的软件厂商支持限制的全局模式。由这样的厂商提供的数据库服务器可以直接合作，无需中间件，但仍如图18-1b那样运作[⊖]。

应用使用一种命名约定来表示各数据库中的表。因此，表的位置就可被隐藏（位置透明度）。当一个站点的表被访问时，到该站点的联结就会自动建立。

应用可以执行引用不同站点的表的SQL语句——例如，一个全局联结。系统还包含一个全局查询优化器来设计高效的查询计划并提供复制透明性。

18.2 在不同数据库中分布数据

在许多情况下，数据在不同站点分布不受应用设计者的控制。例如，由于安全的原因某些数据项必须保存在特定的站点。在另一些情况下，设计者可以参与决定在哪里存放或复制数据。在这一节中，我们将描述一些关于数据分布的问题。

18.2.1 分段

分布数据的最简单方法就是把单个的表存储在不同的站点。然而，表不一定是分布数据的最好单位。通常一个事务访问的只是表的部分行或表的一个视图，而不是整个表。如果不同的事务访问表的不同部分且在不同的站点运行，那么将表的一部分储存在执行相应事务的站点上将会提高性能。当以这种方式分解表时，称表的这些部分为**段**（fragment）。对于此类应用，段是更好的数据分布的单位。

将表的段作为分布单位还有其他的好处。例如，在一个很大的表上处理一个查询需要很多时间，而把它分布到几个保存表段的站点上去运行就会大大减少处理时间。又例如，考虑一个处理一所大学中所有学生的姓名和平均成绩的查询。如果STUDENT表和TRANSCRIPT表都保存在中心管理站点的话，那么所有的处理都将在该站点进行。如果这些表被分段存储在各校区，那么每个校区的数据库管理系统就可以并行执行并且用较少的时间产生结果。分布的表段甚至可能改善系统吞吐量，前提是在一个站点运行的查询只访问本地的表段。分段可以是水平或垂直的。

1. 水平分段

一个表T被分成若干段:

$$T_1, T_2, \cdots, T_r$$

其中每一段包含T的一部分行并且T的每一行都会出现在一个段中。例如，网络零售商可能有如下关系:

INVENTORY(StockNum, Amount, Price, Location)

用来描述他的库存，以及存储库存的仓库地点。该零售商可以按城市对此关系作水平分

⊖ Oracle实现的视图支持一定程度的异构性，即允许其他厂商的产品对数据库进行有限访问。除此之外，这里描述的视图与Oracle提供的视图类似。

段。例如，将所有满足

```
Location = 'Chicago'                                    (18.1)
```

的元组存储在一个叫INVENTORY_CH并位于芝加哥的仓库的段中，其模式为

```
INVENTORY_CH(StockNum, Amount, Price, Location)
```

（Location属性现在是多余的了，可以被省略。）因为每一个元组都被存放在某段中，因此水平分段是无损的。可以合并所有段来重建表。

更一般地，每一个段都满足：

$$T_i = \sigma_{C_i}(T)$$

其中C_i是选择条件，并且T中的每一个元组对某个i值都满足C_i。表达式（18.1）是选择条件的一个例子。

2. *垂直分段*

一个表T被分成若干段：

$$T_1, T_2, \cdots, T_r$$

每一段包含T的一部分列。每一列必须至少被一个段包含，并且每一段必须包含候选键的列（对所有段都一样）。

于是，网络零售商可能有如下的关系

```
EMPLOYEE(SSnum, Name, Salary, Title, Location)
```

来描述所有仓库的雇员。它可能会将EMPLOYEE关系垂直分段为

```
EMP1 (SSnum, Name, Salary)
EMP2 (SSnum, Name, Title, Location)
```

其中EMP1存储在总部站点（职工工资在该站点计算）而EMP2存储在别的地方。因为每一列至少被一个段包含，且所有的段都含有同样的候选键，所以垂直分段是无损的。对所有段作自然联结就可重建原来的表。因为每一段都含有候选键，所以垂直分段涉及复制。其他列也可以被复制。在上述例子中，两个段中都含有Name列，因为它在各站点的本地应用中都需要用到。

需要注意的是，垂直分段的原理和我们在第8章中讨论的关系规范化的原理是不同的。实际上，这三个关系（EMPLOYEE、EMP1和EMP2）都是满足Boyce-Codd范式的，所以第8章提出的算法不适用于EMPLOYEE。

3. *混合分段*

同样可以组合水平分段和垂直分段，但是一定要保证可以从其分段中重建原表。一种方法是先进行一种分段，再进行另一种分段。于是，当网络零售商将EMPLOYEE垂直分段成EMP1和EMP2之后，他可以按照地点水平分段EMP2。对应芝加哥和布法罗仓库的分段就是

```
EMP2_CH (SSnum, Name, Title, Location)
EMP2_BU (SSnum, Name, Title, Location)
```

（再强调一次，属性Location可以被省略。）EMP1被存放在总部站点，而EMP2_CH和EMP2_BU被存放在相应的仓库站点。

4. 导出的水平分段

在某些情况下，可能需要水平分段一个关系，但决定哪些行属于哪个分段的信息并不包含在这个关系中。例如假设网络零售商在一个城市有多个仓库，每个仓库用一个编号来表示。数据库包含两个表:

INVENTORY(StockNum, Amount, Price, WarehouseNum)
WAREHOUSE(WarehouseNum, Capacity, Street-address, Location)

该零售商在每一个城市有一个数据库站点并且想水平分段INVENTORY使某个城市中的分段含有那座城市的所有仓库中所有货品的信息。问题在于，指出仓库所在城市的属性Location不是INVENTORY的属性。所以要如何分割元组就不是很清楚。

为解决这个问题，我们应该知道用仓库编号表示的每个仓库的位置。而这些信息保存在WAREHOUSE表中。因此，我们要先联结这两个表的信息然后再做分段。因为仓库编号用同样的属性名存放在这两个表中，我们可以使用自然联结。INVENTORY中描述芝加哥的仓库的分段称为INVENTORY_CH，其中的行是联结了WAREHOUSE中的满足谓词Location = 'Chicago'的行。于是，

$$\text{INVENTORY_CH} = \pi_A(\text{INVENTORY} \bowtie (\sigma_{\text{Location='Chicago'}}(\text{WAREHOUSE})))$$

其中A是INVENTORY中所有属性的集合。这个联结的作用仅仅是帮助我们找到要包含在INVENTORY_CH中的INVENTORY的行。我们并不想保留WAREHOUSE中的任何列，因此我们将结果在A中的属性上作投影。因为

$$\sigma_{\text{Location = 'Chicago'}} \text{WAREHOUSE}$$

是WAREHOUSE的一个段，INVENTORY的分段是从WAREHOUSE的分段导出的，所以我们称这种分段为**导出分段**（derived fragmentation）。INVENTORY_CH是INVENTORY和WAREHOUSE的一个段的*半联结*（semijoin）。我们将在18.3.1节讨论半联结。

水平分段在一个站点的（大多数）应用都只需要访问关系中*元组的子集*时使用；垂直分段在一个站点的（大多数）应用只需要访问关系中*属性的子集*时使用。我们将在18.3节讨论用数值方法比较涉及分段的不同数据库设计。

基于全局模式的体系结构可以提供**分段透明性**（fragmentation transparency）。这意味着未分段的原始关系出现在全局模式中，而中间件会将对关系的访问转换成对其在不同数据库中相应的分段的访问。相反，多数据库系统就不能提供分段透明性——每个应用程序都必须知道分段并且在查询中恰当地使用它。

18.2.2 更新和分段

尽管我们的兴趣主要集中在查询上，但我们也注意到，当关系被分段时，更新操作有时会要求元组从一个分段移到另一个分段，也就是从一个数据库站点移到另一个数据库站点。假设那个网络零售商的一个雇员从芝加哥仓库调到了布法罗仓库。在未分段的EMPLOYEE关系

EMPLOYEE (SSnum, Name, Salary, Title, Location)

中，该员工的元组中的Location属性的值必须加以修改。如果EMPLOYEE如前所述被分段成EMP2_CH和EMP2_BU的话，更新该雇员的Location属性要求将相应的元组从芝加哥的数据库移

到布法罗的数据库。

因此，在决定将数据放在分布式数据库的什么位置时，应该考虑更新和查询操作要求在站点间移动数据的可能性。

18.2.3 复制

复制是分布式数据库中最常用和最有用的机制之一。在几个站点保存数据副本增加了可用性，因为即使有些站点发生故障，数据还是可以被访问。复制还可以改善性能：查询可以被更有效地执行，因为可以从本地或附近的副本读到数据。但是，更新速度通常会变慢，因为数据的所有副本都需要更新。因此只有在那些更新比查询少得多的应用中性能才会因复制得到提高。在本节中，我们将讨论和单个SQL语句执行相关的性能问题。在26.7节中我们将进一步讨论访问包含副本数据的数据库的事务性能。

示例一：为跟踪其顾客，网络零售商可能有如下关系：

```
CUSTOMER(CustNum, Address, Location)
```

其中Location表示由一个特定仓库服务的区域。总部站点的一个每月给所有顾客邮寄广告的应用需要查询该关系，而每个仓库站点查询该关系来获得其所在区域的送货信息。当一个新顾客在该公司注册或某一顾客的信息改变（这经常发生）时，该关系在总部站点更新。直观地说，按Location将关系水平分段，且把特定的分段存放在相应的仓库和总部站点，这看来很正确。于是，需要复制该关系，并在总部站点存放一个完整的副本。我们对此进行分析，并和另外两种不需要复制数据的设计方案做一下比较。这三种设计方案是：

1) 将整个关系存放在总部站点，在仓库站点不存放任何信息。

2) 将所有分段存放在仓库站点，在总部站点不存放任何信息。

3) 在两种站点都存放分段的副本。

比较这些方案的一种方法是在特定应用执行时分别估计每一种方案下必须在站点之间传送的信息量。为了做出比较，我们对于表的大小和各应用执行的频度做出如下假设：

- CUSTOMER关系大约有100 000个元组。
- 总部的邮寄广告应用每月向每一个顾客寄一份广告。
- 每天需要大约500次送货服务（在所有仓库），且每一次送货对应一个元组。
- 公司每天大约会增加100个新的顾客（相对而言，各个顾客信息的变化数目是可以忽略的）。

现在我们可以衡量这三种方案了。

1) 如果我们将关系存放在总部站点，则每当要送货时，信息就必须从总部传送到相应的仓库站点——大约每天500个元组。

2) 如果我们将分段存放在仓库站点，信息必须以如下方式传送：

- 当执行邮寄广告应用时，从仓库到总部大约每月传送100 000个元组，即每天传送3300个元组。
- 当一个新顾客注册时，从总部到仓库大约每天传送100个元组。

所以每天一共约需传送3400个元组。

3) 如果我们在仓库和总部站点都复制分段，仅在有新客户注册时，信息需要从总部传到

仓库——大约每天100个元组。

从上述比较中可看出，复制是最好的方案。但是，其他的问题也可能很重要，例如事务的响应时间。

1) 如果我们将关系存放在总部，则处理送货的时间很长，因为它需要远程访问。但这可能不会被视为重要问题。

2) 如果我们将分段存放在仓库且每月邮寄广告是由单个应用完成的，那么有100 000个元组必须从仓库传送到总部。这将会阻塞通信系统并且使其他应用的运行速度变慢。可以在其他应用较少执行的周末的晚上运行该邮寄应用来避免这个问题。

3) 如果我们复制分段，则注册新顾客的时间就比较长，因为需要更新总部和相应仓库两方面的表中的数据。这个问题很重要，因为更新时客户是在线的，而更新时间过长会减慢整个注册的速度而变得不可接受。但是，在这个应用中，当总部数据库完成更新后，可以认为与顾客的交互已经完成了。仓库站点的更新可以待日后再完成，因为仅当需要执行送货事务时才会需要那些信息。我们将在15.3.3节和26.7节中讨论**异步更新复制**（asynchronous-update replication）。

我们可以从上述讨论中看出，复制分段仍然是最好的方案。

18.3 查询策略

多数据库系统由一组互相独立的DBMS组成。在一种多数据库系统中，每个DBMS对应用都有一个SQL接口。为了查询储存在多个站点的信息，必须把查询分解为一系列SQL语句，每一条语句都由特定的DBMS来处理。当接收到一条SQL语句后，该站点的查询优化器就会产生一个查询执行计划，执行语句，将结果返回到这个应用。

支持全局模式的系统含有一个全局查询优化器，它用全局模式分析查询并将其翻译成一系列在单个站点执行的步骤。每一个步骤又可在相应站点进一步用本地查询优化器优化并执行。因此，全局查询处理涉及一个分布式算法，分布式算法又涉及DBMS之间的直接数据交换。结果，DBMS除了对应用有一个SQL接口，还要有一个支持与其他DBMS间数据交换的接口。

在这两种情况下，我们感兴趣的都是如何更有效地执行查询。由于输入输出的代价远大于计算的代价，所以我们在第13章中衡量查询执行计划的效率的方法是估计所要求的输入输出操作的次数。同样的道理，衡量分布式数据库查询的代价基于所需要的通信代价，因为通信是昂贵而费时的。

我们对查询优化的兴趣在三个方面。熟悉全局查询优化算法可以帮助我们设计：

- 全局查询，能够在包含全局查询优化器的系统中有效地执行。
- 全局查询，能够在不包含全局查询优化器的多数据库系统中有效地执行（见18.3.2节）。
- 分布式数据库，使全局查询可以在其上有效地执行（见18.3.3节）。

18.3.1 全局查询优化

1. 联结的计划

涉及不同站点的表联结（全局联结）的查询其代价是尤其昂贵的，因为必须在站点间传

送信息来决定结果中的元组。例如，A站点的一个应用要联结站点B和C的两个表，并把结果传回A站点。可用全局优化器来评估的两个执行该联结的直观方法是:

1) 把两个表都传到A站点，然后在那里执行联结。A站点的应用程序就可以显式地测试联结条件。这是在不支持全局联结的多数据库系统中使用的方法。

2) 把较小的表（例如B站点的表）传到C站点，在C站点执行联结，然后将结果传到A站点。

为了更详细地说明，我们考虑两个表STUDENT (Id, Major)和TRANSCRIPT (StudId, CrsCode)，其中STUDENT记录了每个学生的专业，TRANSCRIPT表记录了学生在本学期选修的课程。这两个表分别存放在B站点和C站点。假设A站点的应用要计算等值联结，其联结条件为

$$\texttt{Id = StudId} \tag{18.2}$$

为了比较这两种查询计划，我们必须对表的大小和（某些情况下）对这些表所进行的操作的相对频度做出一些假设。对于本例，我们假设

- 属性的长度是:
  - Id和StudId: 9个字节。
  - Major: 3个字节。
  - CrsCode: 6个字节。
- STUDENT大约有15 000个元组，每个元组的长度是12个字节（9+3）。
- 大约有5000个学生至少选修了一门课程，且平均每个学生选修4门课。因此，TRANSCRIPT大约有20 000个元组，每个元组的长度是15个字节（9+6）。注意，有10 000个学生没有选修任何课程。

联结的结果中有大约20 000个元组（TRANSCRIPT中的每一个元组对应联结结果中的一个元组），每个元组的长度是18个字节（9+3+6）。

基于这些假设，我们比较以上的查询策略。

1) 如果把两个表都送到A站点来做联结，我们要传送480 000个字节（即15 000 × 12 + 20 000 × 15）。

2) 如果把STUDENT表送到C站点，在那里计算联结，然后把结果送到A站点，我们需要传送540 000个字节（即15 000 × 12 + 20 000 × 18）。

3) 如果把TRANSCRIPT表送到B站点，在那里计算联结，然后把结果送到A站点，我们需要传送660 000个字节（即20 000 × 15 + 20 000 × 18）。

于是，我们可以看出最好的策略是1。

2. *半联结的计划*

全局查询优化器可能考虑的另一个更为有效的方法是只将STUDENT表中那些将会实际参与联结的元组从B站点传到C站点，然后在C站点对这些元组和TRANSCRIPT表做联结。这个方法涉及被称为*半联结*（semijoin）的操作。这个过程由三个步骤组成。

1) 在C站点，计算表P。P是TRANSCRIPT表在StudId上的投影，即在联结条件中将会涉及到的列，并把P送到B站点。于是，P中包含那些目前至少选修一门课的学生的Id。

2) 在B站点，根据联结条件(18.2)对STUDENT表和P做联结，并把结果表Q送到C站点。Q中

包含了STUDENT表中所有参与联结的元组。在本例中，Q包含了对应于STUDENT表中所有至少选修了一门课的学生的元组。

3) 在C站点，根据联结条件对TRANSCRIPT表和Q做联结。然后把STUDENT表和TRANSCRIPT表的联结结果送回A站点。

步骤2的结果关系Q被称为STUDENT表和TRANSCRIPT表的半联结。更普遍地，两个关系T_1和T_2的半联结（是基于一个联接条件的）被定义为T_1和T_2的联结在T_1的列上的投影:

$$\pi_{attributes(T_1)}(T_1 \bowtie_{join\text{-}condition} T_2)$$

换言之，半联结是由T_1中参与与T_2的联结的元组组成的。于是就可想到为计算式子

$$T_1 \bowtie_{join\text{-}condition} T_2$$

可先计算半联结，然后再和T_2联结:

$$(\pi_{attributes(T_1)}(T_1 \bowtie_{join\text{-}condition} T_2)) \bowtie_{join\text{-}condition} T_2$$

以上两式相等的证明留作练习18.10。

看起来我们似乎是倒退了一步，用两个联结来代替一个联结。然而，步骤1为我们提供了这样的线索，即执行半联结潜在的经济性与下面的等式有关:

$$\pi_{attributes\ (T_1)}(T_1 \bowtie_{join\text{-}condition} T_2) = \pi_{attributes\ (T_1)}(\pi_{attributes\ (join\text{-}condition)}(T_2) \bowtie_{join\text{-}condition} T_1)$$

换句话说，为计算T_1和T_2的半联结，我们可以先在联结条件中的属性上对T_2做投影，然后将结果与T_1联结（用相同的联结条件）。我们将这个等式的证明留作练习18.9。第一步潜在地减少了通信代价，因为T_2的投影要比T_2小得多，所以我们就可以避免在通信链路上传送大量数据。但是，我们也必须为额外的对半联结结果和T_2的联结付出代价。如果在通信上节省的代价超过了额外联结的代价，则我们还是成功的。

在我们的例子中，计算的三个步骤对应于下列的代数表达式:

$$\pi_{attributes\ (STUDENT)}(\pi_{attributes\ (join\text{-}condition)}(TRANSCRIPT) \bowtie_{join\text{-}condition} STUDENT) \bowtie_{join\text{-}condition} TRANSCRIPT$$

根据我们上述的讨论，上式等价于计算STUDENT表和TRANSCRIPT表的联结。

将这个方法与之前的几个方法比较是很有启发的。

1) 在第一步，我们传送了45 000个字节（即5 000 × 9）。

2) 在第二步，我们传送了60 000个字节（即5 000 × 12）。

3) 在第三步，我们传送了360 000个字节（即20 000 × 18）。

因此我们总共传送了465 000个字节（即45 000 + 60 000 + 360 000）。所以，就通信代价而言，半联结的方法比我们之前讨论过的方法都要好。

3. 用复制实现全局联结

还有另一种实现全局联结的方法是，在另一个站点保存此站点的表的副本，于是把全局联结转化为本地联结。在这个例子中，我们可以在C站点中保存一个STUDENT表的副本，在C站点与TRANSCRIPT表做联结，然后返回360 000字节的结果到A站点。

这个方法加快了联结操作的速度，但是减慢了被复制的表的更新操作的速度。在这个例子中，更新可能很少发生，因为很少有学生会改变他们的专业。

4. 涉及联结和投影的查询

大多数的查询不仅涉及联结，还包含其他的关系操作符。在上述例子中，假设A站点的一个应用执行一个查询，该查询要求返回所有至少选修一门课程的学生的专业和课程代码。于是，如果至少有一个计算机系(CS)的学生选修了CS305，那么行(CS, CS305)就会在结果表中。该查询先对两个表STUDENT和TRANSCRIPT做联结，然后在Major和CrsCode上投影来得到结果表R。我们的计划是在做联结的站点执行投影操作，然后将结果R送到A站点。

为重新衡量上一节提出的四个策略的通信代价，需要多加一个假设，即R有1000个元组。每个元组的长度是9个字节，因此R的大小是9000字节，比联结后的表要小得多。这在实际查询中是很常见的。

1) 如果把所有的表都送到A站点并在那里完成所有的操作，那么需要传送480 000个字节，同前面一样。

2) 如果把STUDENT表送到C站点，在那里执行操作，然后把R送回A站点，那么需要传送189 000个字节（即15 000 × 12 + 1000 × 9）。

3) 如果把TRANSCRIPT表送到B站点，在那里执行操作，然后把R送回A站点，那么需要传送309 000个字节（即20 000 × 15 + 1000 × 9）。

4) 如果如前面所述做半联结，在C站点做投影，然后把R送回A站点，那么需要传送114 000个字节（即5000 × 9 + 5000 × 12 + 1000 × 9）。

所以，半联结的方法仍然是最好的选择。

可以修改第二步再对过程加以优化。在执行完半联结得到Q之后，对Q做投影只保留查询中需要的STUDENT表的列。只将这些列从B站点传送到C站点。例如，假设STUDENT表还有其他的一些属性（如Address、Date_of_Birth、Entrance_date），并且查询还是像以前一样要求对学生选修的每一门课找到相应的Id、CrsCode和Major。那么在第二步，就可以在WHERE和SELECT语句中提到的这些属性上作投影，并只把这些属性送到C站点。

这个想法可以运用到之前讨论过的所有方案中。在把一个表送到另一个站点进行联结之前，所有不需要的属性都可以被省略掉。

5. 涉及联结和选择的查询

类似的想法也可以用在涉及联结和选择的查询中。例如，假设在网络零售商的应用中只有一个仓库，且EMPLOYEE关系被垂直分段成

```
EMP1 (SSnum, Name, Salary)
EMP2 (SSnum, Title, Location)                                    (18.3)
```

这些关系分别存放在B站点（总部）和C站点（仓库）。假设在第三个站点A有一个应用，要查询所有薪水在$20 000以上的经理的名字。（如果我们假设有多个仓库，其原理是相似的，只是计算上复杂一点儿，见练习18.14）。

一个比较直观的方法是先将两个表联结起来（重新产生EMPLOYEE表）然后利用选择和投影操作来获得：

$$\pi_{\text{NAME}}\left(\sigma_{\text{Title='manager' AND Salary>'20000'}}(\text{EMP1} \bowtie \text{EMP2})\right)$$

但是，用半联结的方法来优化联结不会降低通信代价。这是因为，这两个表都是EMPLOYEE表的垂直分段，所以EMP1和EMP2都包含了EMPLOYEE的一个键SSnum。因此每个表中的所有元组都必须参与重建EMPLOYEE。

但是，我们注意到选择条件可以被分割为单个表上的选择条件。

- 在EMP1上的选择条件是Salary > '20000'。
- 在EMP2上的选择条件是Title = 'Manager'。

现在可以用关系运算符的数学性质来改变联结和选择操作的顺序（回想14.2节中选择的级联和递推规则）：

$$\pi_{NAME}((\sigma_{Salary>'20000'}\ EMP1) \bowtie (\sigma_{Title='manager'}\ EMP2))$$

详细地说，

1) 在B站点，从EMP1中选择所有工资大于$20 000的元组，结果称为$R_1$。

2) 在C站点，从EMP2中选择所有职位是经理的元组，称结果为R_2。

3) 在某个站点（由后面的规则决定）执行R_1和R_2的联结并把结果在Name属性上投影。称结果为R_3。如果这个站点不是站点A，那么把结果R_3送到A站点。

余下的唯一问题就是，在哪里执行第三步中的联结。有三种可能：

1) 计划1：把R_2送到B站点，并在那里做联结。然后把姓名送到A站点。

2) 计划2：把R_1送到C站点，并在那里做联结。然后把姓名送到A站点。

3) 计划3：把R_1和R_2送到A站点，并在那里做联结。

像从前一样，为了决定哪个是最好的方案，必须考虑不同的表的大小和结果的大小。我们做出如下假设：

- 属性的长度为：
  - SSnum: 9个字节。
  - Salary: 6个字节。
  - Title: 7个字节。
  - Location: 10个字节。
  - Name: 15个字节。

因此，EMP1中的每个元组长度是30个字节，而EMP2中的每个元组长度是26个字节。

- EMP1（和EMP2）有大约100 000个元组。
- 大约有5 000名雇员的薪水超过$20 000。因此，$R_1$大约有5000个元组（每个元组的长度为30字节），共有150 000个字节。
- 有大约50名经理。因此，R_2有大约50个元组（每个元组的长度为26字节），共有1300个字节。
- 大约90%的经理薪水超过$20 000。因此，$R_3$有大约45个元组，每个元组长度为15个字节，共有675个字节。

现在可以估计每个计划的代价了。

1) 如果在B站点做联结，需要从C站点传送1300个字节到B站点，还要从B站点传675个字节到A站点，共传送1975个字节。

2) 如果在C站点做联结，需要从B站点传送150 000个字节到C站点，还要从C站点传675个字节到A站点，共传送150 675个字节。

3) 如果在A站点做联结，需要从B站点传送150 000个字节到A站点，还要从C站点传送1300个字节到A站点，共传送151 300个字节。

正如你所见的，第一个计划要远远好于另外两个计划。

为充分理解设计良好的查询计划的重要性，将这个计划的代价与未优化的计划的代价作比较。在未优化的策略中，EMP1和EMP2被传送到A站点，在那里评估查询。那样的话，需要传送5 600 000个字节（即2 600 000 + 3 000 000）！

18.3.2 多数据库系统的策略

访问多数据库系统的应用不能在全局模式上提交SQL语句。涉及多个站点数据的查询必须用一组SQL语句构造，每一个语句根据特定的DBMS的模式来构造并在该站点处理。尽管这样的环境不存在全局查询优化器，但是应用设计者可以参考在18.3.1节讨论的一些想法来选择实现查询的合适的语句序列。遗憾的是，设计者在两个重要的方面受到限制。我们用网络零售商的例子来说明。

1) 在一个多数据库系统中，数据只能在数据库站点和提交查询的站点（在本例中是A站点）间通信。另一方面，利用全局查询优化器，数据库站点互相合作并且直接通信。

2) 即使数据不能在数据库站点之间直接传送，我们还可以考虑将数据通过A站点间接地从一个站点传送到另一个站点。但这个方法是不可能的。尽管A站点可以从DBMS接收数据（作为提交SELECT语句的结果），但它不能向DBMS送出数据，因为应用的接口只关心处理SQL语句（而不是接收数据）。

这使得它从本质上不可能去模仿半联结方法的第一步，即传送一个表的投影（在该步骤中是P）[⊖]。

示例　重新考虑前一节中被分段的表EMPLOYEE，以及在A站点提交的请求所有工资大于$20 000的经理的姓名的查询。如果这个查询在一个有全局优化器的系统中执行的话，我们希望优化器选择计划1，其通信代价只有1975字节。

如果同样的查询在一个多数据库系统中执行，则应用设计者可先在每个站点执行SELECT语句，返回R_1和R_2到A站点。然后程序会在这些元组上执行必要的处理来实现联结操作。其通信代价和计划3相同，都是151 300字节。虽然这个策略不如全局查询优化器可选择的最佳策略那么有效率，但它比把EMP1和EMP2整个传送到A站点，花费5 500 000字节代价的原始策略要好得多了。

18.3.3 调整问题：分布式环境下的数据库设计和查询计划

和集中式的情况一样，分布式数据库的查询计划也涉及评估选择方案，其中有：

⊖ A站点可能会通过动态构造一个基于P中的行的SELECT语句试着绕开这个限制。例如，如果P有一个属性attr，且其列值为a, b,…，那么WHERE语句可能含有式子(attr = 'a' OR attr='b' OR…)。注意，投影$\pi_{attr}(P)$被编码到该查询中的WHERE语句中。于是，向远程站点发送这个查询就等价于A发送$\pi_{attr}(P)$到该站点。于是，A站点收到查询结果就等价于从该远程站点收到半联结的结果。遗憾的是，这个方法只适用于attr值域很小的情况。

- 在不同的站点执行操作。
- 在查询执行时将部分结果或整个表从一个站点传送到另一个站点。
- 执行半联结。
- 使用关系代数的启发式优化规则（参见14.2节）对操作重新排序。

应用设计者不能控制全局查询优化器使用的策略。但设计者可以控制分布式数据库的设计，而设计可以改变全局查询优化器可选择的方案，从而在很大程度上影响查询策略。这对于应用设计者手工完成多数据库系统的查询计划也成立。

在集中式的情况下，应用设计者可能通过加入索引或反规范化表（见8.13节、13.7节和14.6节）来改变数据库的模式。在分布式的情况下，设计者可能有更多的选择，例如:

- 把表放在不同的站点。
- 用不同的方法将表分段，然后把分段放在不同的站点。
- 复制表或表中的数据（例如，反规范化的），并把副本放在不同的站点。

和在集中式数据库中一样，这些选择可能会加快一些操作的速度，同时减慢另一些操作的速度。因此，设计者在衡量一个数据库设计时要考虑应用中各操作的相对频度以及该操作的吞吐量和响应时间的重要程度。

在网络零售商的应用中，将INVENTORY关系分段加快了处理送货的本地应用的速度，但也减慢了需要联结分段的全局应用（例如，计算公司总的库存的速度。在权衡这些选择方案时，公司可能要求送货应用必须执行得很快，而计算总库存的应用不会经常执行，而且对其响应时间也没有很高的要求。

应用设计者可能会考虑在总部数据库中复制仓库库存信息来加快全局库存应用的速度。但这个方案很可能被拒绝，因为仓库存货的库存信息总是更新得很快（每一次送货时都要更新），而且为更新副本花费的通信代价远远大于执行全局库存应用的代价。权衡这些利弊对于设计一个高效运行且满足公司需求的应用是很重要的。

18.4 参考书目

我们关于分布式查询处理的描述基于两个系统的实现，即SDD-1[Wong 1977, Bernstein et al. 1981]和System R*[Griffiths-Selinger and Adiba 1980]。半联结的理论在[Bernstein and Chiu 1981]中讨论。[Chang and Cheng 1980, Ceri et al. 1982]中讨论了分段。更深入的关于分布式数据库问题的研究（特别是数据库设计和查询处理）可以在一些专业的文献中找到，例如[Ceri and Pelagatti 1984, Bell and Grimson 1992]和[Ozsu and Valduriez 1991]。

18.5 练习

18.1　讨论应用设计者把应用设计为同构系统的好处，即所有的数据库由同一个厂商提供。

18.2　解释为什么一个表可以按照集中式系统的模式来分段。

18.3　解释下列说法正确与否：通过对表T做分段（垂直或水平）得到的两个表，其联结不可能包含T以外的元组。

18.4　考虑18.3.2节给出的两个多数据库系统中查询设计的例子。用Java和JDBC编写程序实现它们。

18.5　A站点的一个程序要求联结分别在B站点和C站点的两个表，写出该程序。陈述使以下结论成立的

假设条件：只考虑通信代价，那么最好的实现方法是将B站点的表送到C站点，在那里做联结，然后把结果送到A站点。

18.6 你正在考虑按Location水平分段如下关系：

EMPLOYEE (SSN, Name, Salary, Title, Location)

并把每个分段存放在那个地点的数据库中，同时可以在别的站点复制一部分分段。讨论可能会影响你的决定的查询类型和更新（以及它们的频度）。

18.7 假设有如下关系

EMPLOYEE2 (SSnum, Name, Salary, Age, Title, Location)

把它分段成

EMP21 (SSnum, Name, Salary)
EMP22 (SSnum, Title, Age, Location)

其中EMP21存放在B站点，而EMP22存放在C站点。A站点的一个查询要求找出会计部门中薪水大于他们的年龄的所有经理的姓名。使用18.3.1节对表和属性大小的假设，为这个查询设计一个计划。假设Age列的数据长度为2个字节。

18.8 设计一个多数据库查询计划和一组SQL语句来实现上题中的查询。

18.9 用关系代数来证明18.3.1节中用半联结来实现联结的方法的第2步确实产生一个半联结。为简明起见，假设我们要做的联结是自然联结。即证明

$$\pi_{\text{attributes}(T_1)}(T_1 \bowtie T_2) = \pi_{\text{attributes}(T_1)}(\pi_{\text{attributes(join-condition)}}(T_1) \bowtie T_2)$$

18.10 用关系代数来证明18.3.1节中用半联结来实现联结的方法的第3步确实产生一个联结。为简明起见，假设我们要做的联结是自然联结。即证明

$$\pi_{\text{attributes}(T_1)}(T_1 \bowtie T_2) \bowtie T_2 = (T_1 \bowtie T_2)$$

18.11 为18.3.1节中联结的例子设计一个查询计划，假设表的大小和该节中表的一样，但有如下的区别：

a. B站点的一个应用请求联结。

b. C站点的一个应用请求联结。

18.12 用关系代数证明18.2.1节中设计水平分段的方法的正确性。

18.13 证明半联结操作是不可交换的，即T_1半联结T_2和T_2半联结T_1是不同的。

18.14 用式(18.3)的模式来设计找到所有工资大于$20 000的经理（Title = 'manager'的雇员）的姓名，但是假设有三个仓库。同时假设雇员总数约为100 000人，其中5 000人的工资在$20 000以上，经理的总人数是50，且其中90%的人工资在$20 000以上。

第19章 OLAP和数据挖掘

本章介绍了OLAP（联机分析处理）、数据仓库和数据挖掘这些新兴领域的概念和技术。OLAP的相关内容可参考[Chaudhuri and Dayal 1997]。数据挖掘较之OLAP包含了更多的内容，涉及数据库、统计分析和机器学习。通过本章的介绍，读者可以对这些知识有大致的了解。如果想进行深入学习相关内容，请参考介绍这一领域最新应用和研究的论文集[Fayyad et al. 1996]和[Han an Kamber 2001]。

19.1 OLAP和数据仓库

因特网上为什么会有那么多的免费材料——只需要填一张表格，就可以获得这些材料？事实上，不会有真正的免费午餐，你付出的是你的个人信息，而且，当你在因特网上购物的时候，你还提供了更多的个人信息，即你的购买习惯。

当你在超市和其他商场用信用卡购物的时候，你也同样提供了个人信息。在这些情况下，你的个人信息被存储到了事务处理系统（transaction processing system）中，系统将来会用这些数据进行分析。

你不禁要问，存储这些信息有什么用呢？一般来说，这些信息会与从其他途径得到的关于你的信息汇总起来，存储到数据库中。使用这些信息的方式有以下几种:

- 一些企业把能搜集到的所有用户的信息和他们的购买情况结合起来，进行分析，在此基础上计划它们的货存、广告以及其他方面的企业经营策略。
- 企业可能会用这些信息分析出你的购买习惯，然后通过电子邮件和其他途径为你提供适合你的商品信息。可能不久以后，你和你身边的人就会看到不同的电视广告（这些广告就是基于你们的购买习惯而安排的）。

信息收集、理解中的这些发展趋势已经严重地涉及到了个人隐私问题。但是，那些问题并不是本书所关心的。

这类型的应用称为**联机分析处理**（Online Analytic Processing，OLAP），与之相对的是**联机事务处理**（Online Transaction Processing，OLTP），这两种应用在目的和技术要求上都是不同的。

- OLTP的目的是维护数据库该数据库一个企业的准确模型。这个系统应提供足够大的事务吞吐量和较短的响应时间以满足负载的要求，并且避免用户失败。它的特征包括:
  - 简短的事务。
  - 相对频繁的更新。
  - 只涉及数据库很小部分的事务处理。
- OLAP的目的是使用数据库中存储的信息来指导企业制定决策。因此所涉及的数据库非常巨大而且不需要完全准确或时刻保持为最新数据，.对响应时间的要求也不是很苛刻。

它的特征包括:

- 复杂的查询。
- 不频繁的更新。
- 涉及数据库很大部分的事务处理。

我们在1.4节的OLAP例子是针对一个连锁超市的经理群的，他们需要的是对数据库进行突发的查询（不是预先设计好的），来指导他们做出某方面的决策。这个例子很形象地说明了OLAP的特点——即席的查询（事先未知的），它的用户是一些不具有很多计算机知识的企业管理人员。

本节开头部分的例子描述了OLAP的新型的应用。企业使用预先设计好的查询来检索OLAP数据库，在持续运营的基础上定制他们的营销以及企业的其他方面的策略。这些查询通常是复杂的，另一方面，由于这些查询在决策支持中的关键性和使用的频繁性（例如每天或每周使用），因此这些查询要由专业人员来设计和实现。

在传统的OLAP应用中，OLAP数据库中的信息很多情况下仅仅是企业每天的交易数据（可能来源于OLTP数据库）；而在一些新型的应用中，企业管理者需要根据他们的具体应用主动地收集——甚至可能是购买一些另外的数据。

正如前面描述，OLAP的目的是针对一些应用来分析数据，所以提出了两种单独的却彼此相关的技术上问题:

- *要执行的分析过程和这些过程所需要的数据。*例如，一个公司想决定下一阶段生产哪些产品。根据这个应用，它设计一种操作，这种操作需要前一阶段和过去五年中相应阶段的产品销售情况的数据。
- *获得分析所需的大批数据的方法。*例如，公司如何从下属部门的数据库抽取所需数据？在OLAP数据库中以何种形式存储这些数据？怎样才能对分析所需要的数据进行高效率的检索？

第一个问题，由于它需要针对公司的具体业务来设计特殊的算法，所以不是数据库方面的问题，故我们主要讨论第二个问题，即针对分析操作，我们需要设计什么样的数据库。假设检索出的数据只是显示在屏幕上。然而，在很多情况下，尤其是在一些新的应用中，这些数据还要作为复杂的分析操作的输入。

数据仓库

OLAP数据库通常存储在专门的OLAP服务器上，这个服务器称为**数据仓库**（data warehouse）。我们构造数据仓库来支持OLAP查询。这些查询可能非常复杂，如果它们在OLTP数据库上进行，可能慢得让人无法忍受。

我们将在19.7节讨论填充数据仓库涉及的一些问题。首先，我们看看数据仓库中要存储哪些数据。

19.2 OLAP应用的多维模型

1. 事实表和维表

许多OLAP应用和1.4节的超级市场的例子很相似：分析不同时间段中不同超级市场的不同商品的销售额。我们用图19-1中的关系表来描述这样的销售数据。Market_Id表示一个超级

市场，Product_Id表示一种商品，Time_Id表示一个特定的时间段，Sales_Amt表示在指定时间段、指定超级市场中特定产品的销售总额（以美元计）。这种表称为**事实表**（fact table），因为它包含了分析所需要的所有事实。

SALES	Market_Id	Product_Id	Time_Id	Sales_Amt
	M1	P1	T1	1000
	M1	P2	T1	2000
	M1	P3	T1	1500
	M1	P4	T1	2500
	M2	P1	T1	500
	M2	P2	T1	800
	M2	P3	T1	0
	M2	P4	T1	3333
	M3	P1	T1	5000
	M3	P2	T1	8000
	M3	P3	T1	10
	M3	P4	T1	3300
	M1	P1	T2	1001
	M1	P2	T2	2001
	M1	P3	T2	1501
	M1	P4	T2	2501
	M2	P1	T2	501
	M2	P2	T2	801
	M2	P3	T2	1
	M2	P4	T2	3334
	M3	P1	T2	5001
	M3	P2	T2	8001
	M3	P3	T2	11
	M3	P4	T2	3301
	M1	P1	T3	1002
	M1	P2	T3	2002
	M1	P3	T3	1502
	M1	P4	T3	2502
	M2	P1	T3	502
	M2	P2	T3	802
	M2	P3	T3	2
	M2	P4	T3	333
	M3	P1	T3	5002
	M3	P2	T3	8002
	M3	P3	T3	12
	M3	P4	T3	3302

图19-1　超级市场应用程序的事实表

Sales_Amt属性的取值是Market_Id、Product_Id、Time_Id这三个维的值的函数，所以我们可以把这个销售数据看作是**多维的**（multi-dimensional）。在这张表中，前三列表示三个维，第四列表示实际的销售数据。

我们也可以考虑把事实表中的数据存放到一个多维的立方体中。图19-2显示用数据立方体表示超级市场例子中的数据的情况。这个立方体是三维的，三个维分别是Market_Id、Product_Id、Time_Id，立方体的每个单元（cell）表示相应的Sales_Amt的值。这种多维的视图可以直观地表示OLAP的查询和结果。

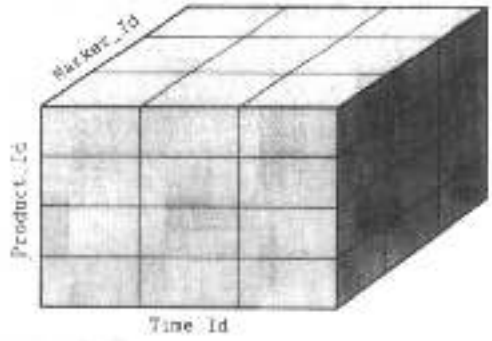


图19-2 超级市场应用程序的三维立方体

每一维的另外的信息可以存储到维表（dimension table）中，维表描述每个维的属性，比如Market_Id，Product_Id和Time_Id。在上面例子中，可以有三张维表，分别称为MARKET、PRODUCT和TIME，如图19-3所示。MARKET表说明超级市场的地理位置，位于哪个地区、州、城市。在一个更真实的例子中，这张表包含了一个超级市场构成的链，每个区包含多个州，每个州包含多个城市，每个城市包含多个超市。

MARKET	Market_Id	City	State	Region
	M1	Stony Brook	New York	East
	M2	Newark	New Jersey	East
	M3	Oakland	California	West

PRODUCT	Product_id	Name	Category	Price
	P1	Beer	Drink	1.98
	P2	Diapers	Soft Goods	2.98
	P3	Cold Cuts	Meat	3.98
	P4	Soda	Drink	1.25

TIME	Time_id	Week	Month	Quarter
	T1	Wk-1	January	First
	T2	Wk-24	June	Second
	T3	Wk-52	December	Fourth

图19-3 超级市场应用程序的维表

2. 星型模式

可以用一张图（如图19-4所示）表示超级市场这个例子所蕴含的关系，这张图就像一颗星星，中间是一张事实表，向周围辐射出维表，所以称之为星型模式。在OLAP中，这种模式是很常见的。值得注意的是，星型模式和实体-联系（E-R）图很相似，事实表对应一个联系，维表对应实体。

如果把维表都规范化（一张维表可能变成几张表），那么这幅图会变得更复杂，从而形成雪花的结构，这种模式被称为**雪花模式**（snowflake schema）。但是，由于以下两个原因，我们很少规范化维表。

- 通过规范化维表所节约的空间和事实表所占的空间相比是微不足道的。

- 很少会更新维表，所以不必考虑由于维表不规范所带来的更新异常，而且，如果把维表中的关系分解为3NF或BCNF，会使得查询非常复杂，如8.13节所述。

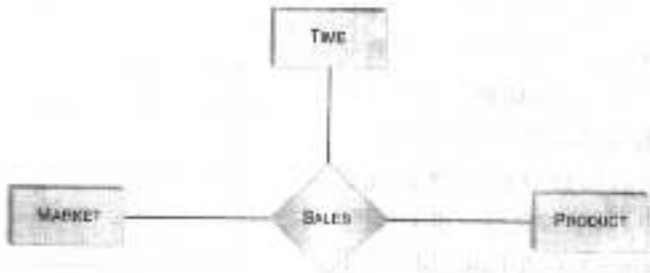


图19-4　超级市场示例的星型模式

除了星型模式外，很多OLAP应用还使用**星座模式**（constellation schema）。在这种模式中，可能多个事实表共享一张或多张维表。例如，超级市场的应用还需要另外一张事实表INVENTORY，它需要维表WAREHOUSE、PRODUCT和TIME，如图19-5所示。维表PRODUCT和TIME就被事实表INVENTORY和SALES共享，而WAREHOUSE没有被共享。

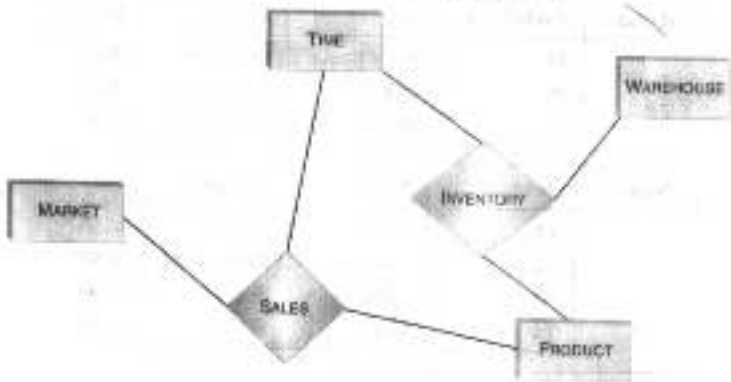


图19-5　扩展的超级市场示例的星座模式

19.3　聚合

很多OLAP查询涉及对事实表中数据的**聚合**（aggregation）。例如，下面的SQL语句表示查询每个超级市场中每种产品的销售总量（所有时间段销售额的和）。

```
SELECT      S.Market_Id, S.Product_Id, SUM(S.Sales_Amt)
FROM        SALES S
GROUP BY    S.Market_Id, S.Product_Id
```

这个查询返回的结果如图19-6所示，它可以看成一个二维的立方体，每个单元的值是对Sales_Amt的值的聚合。由于聚合是对整个时间维进行的（即结果不依赖于时间坐标），所以通过聚合，产生了一个压缩后的维视图，由原来的三维变为现在的两维。

19.3.1　下钻、上卷、切片和切块

一些维表中包含有**聚合层次结构**（aggregation hierarchy），例如，维表MARKET表示层次

结构。

```
Market_Id → City → State → Region
```

这意味着地区包含州，州包含城市，城市包含超级市场。我们可以在层次结构的不同层上进行查询，例如:

```
SELECT      S.Product_Id, M.Region, SUM(S.Sales_Amt)
FROM        SALES S,   MARKET M
WHERE       M.Market_Id = S.Market_Id
GROUP BY    M.Region, S.Product_Id
```

查询结果如图19-7所示，每个单元的值是每个地区中每件产品的销售总额。

SUM(Sales_Amt)		Market_Id		
		M1	M2	M3
Product_Id	P1	3003	1503	15003
	P2	6003	2402	24003
	P3	4503	3	33
	P4	7503	7000	9903

图19-6 在时间维上聚合Sales_Amt的查询结果

SUM(Sales_Amt)		Region			
		North	South	East	West
Product_Id	P1	0	0	4506	15003
	P2	0	0	8405	24003
	P3	0	0	4506	33
	P4	0	0	14503	9903

图19-7 在地区上下钻的查询结果

当我们执行的查询从层次结构中较高的层次移动到粒度更小的层次，即从比概括到具体(例如从“年”上的聚合到“月”上的聚合)，这种操作称为**下钻**(drilling down)。下钻需要更具体的信息。因此，为了得到“月”上的聚合，我们可能需要事实表或在“天”上聚合了的表。当我们从层次结构中较低的层次移到较高的层次(从“天”上的聚合到“周”上的聚合)，我们称为**上卷**(rolling up)。像这样在时间维上进行下钻或上卷是很常见的，因为我们总是要总结每天、每月或者每季度的销售额。

还有一些相关的OLAP术语。当我们观察多维立方体时，我们可能选择其中的坐标轴的一个子集，我们称为执行一次**转轴**(pivot)。被选择的坐标轴对应于GROUP BY子句中的那些属性。从另一个角度看，转轴是对GROUP BY子句中没有出现的维的聚合。

下面的查询就是对多维立方体执行转轴，以便从产品和时间维上查看该立方体。查询找出今年每个季度的每件产品的销售总额(在所有超级市场的聚合)，查询结果如图19-8所示。

```
SELECT      S.Product_Id, T.Quarter,  SUM(S.Sales_Amt)
FROM        SALES S,  TIME T
WHERE       T.Time_Id = S.Time_Id
GROUP BY    T.Quarter, S.Product_Id
```

(19.1)

如果我们在GROUP BY子句中用年代替季度，这就是在时间层次结构中上卷。

```
SELECT      S.Product_Id, T.Year, SUM(S.Sales_Amt)
FROM        SALES S,  TIME T
WHERE       T.Time_Id = S.Time_Id
GROUP BY    T.Year, S.Product_Id
```

SQL:1999和其他一些OLAP厂商支持一种扩展的SQL语句: ROLLUP，用它来简化上卷操作(参见19.3.2节)。值得注意的是，相应的下钻操作却没有对应的语句，这是因为上卷不仅给用户设计查询提供便利，它还能优化查询处理。比如，用户首先在“季度”上进行了聚合，

然后要在所得结果上进行上卷在“年”上聚合，处理器只要在第一步的查询结果上再进行聚合就行了，这样可以大大减少第二步的处理时间。这样的优化是不可能用于下钻的。

不是所有的聚合层次结构都像地理位置层次结构那样是线性的，图19-9中的时间层次结构是一个格。“月”不包含“星期”，因为一个星期可能包含在两个月中。所以，我们可以把“天”上卷到“星期”或“月”，但从“星期”只能上卷到“季度”中。

SUM(Sales_Amt)		Quarter			
		First	Second	Third	Fourth
Product_Id	P1	6500	6503	0	6506
	P2	10800	10803	0	10806
	P3	1510	1513	0	1516
	P4	9133	9136	0	6137

图19-8　表示每个季度产品销售总额的查询结果

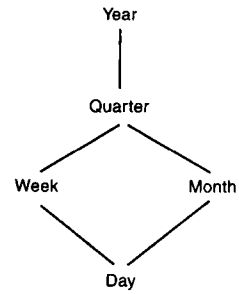


图19-9　以时间层次结构作为一个格

注意，上面的所有查询都需要访问事实表中的大量数据。与之相反，本地超级市场数据库上的OLTP查询*现在番茄汁罐头有多少存货*只需要访问数据库中相应的一条记录。

切片和切块

维上的每个层次结构都把多维立方体分成几个子立方体。例如，时间维上的Quarter（季度）把整个立方体分成四个子立方体。每个立方体存储一个季度的数据。返回这些子立方体信息的查询称为**切片**（slice）或**切块**（dice）。

- 通过在查询中使用GROUP BY子句来限定层次的某一部分（或全部），我们把整个多维立方体分成若干个子立方体，这就是一个**切块**操作。查询(19.1)在表SALES中的时间维上执行了切块，它把时间维按季度切成四块，相应地把立方体切成四个子立方体，每个子立方体表示一个季度的数据。
- 通过使用WHERE子句，我们用等式来把维属性和一个常量比较，这样就可以在查询中给某个维指定特定值，这就是一个**切片**操作。如果在上面查询中，我们需要观察时间段T1上每个市场的销售情况，那就要在时间维上进行切片。

可以看出，切块是按某个维把立方体分成相应的子立方体，而切片是只取出其中的某一个子立方体。

19.3.2　CUBE操作符

很多OLAP查询需要使用聚合函数，一般是通过在SELTCT语句中使用GROUP BY子句来实现聚合。SELECT语句的规则限制OLAP查询的类型，而这些查询在SQL中是很容易编写的。OLAP厂商（以及SQL:1999）在SQL中加入了另外的聚合函数，并且有的厂商允许用户使用自己设计的聚合函数。

一种扩展是ROLLUP子句，另一种扩展是CUBE操作符[Gray et al.1997]。下面用一个例子来讲述CUBE操作符的使用。为了得到这样一张表（如图19-10所示），它和图19-6中的表很相似，不同之处是它要显示每行、每列的总数。如果使用SQL语句构造这张表，我们需要执行

下面的三个SELECT语句来检索必要的信息。

SUM(Sales_Amt)		Market_Id			
		M1	M2	M3	Total
Product_Id	P1	3003	1503	15003	19509
	P2	6003	2402	24003	32408
	P3	4503	3	33	4539
	P4	7503	7000	9903	24406
	Total	21012	10908	48942	80862

图19-10 以电子表格形式出现的销售应用程序的查询结果

第一个查询得到表中间的数据（得不到总数）。

```
SELECT      S.Market_Id, S.Product_Id, SUM(S.Sales_Amt)
FROM        SALES S
GROUP BY    S.Market_Id, S.Product_Id
```

第二个查询得到每行的总数。

```
SELECT      S.Product_Id, SUM(S.Sales_Amt)
FROM        SALES S
GROUP BY    S.Product_Id
```

第三个查询得到每列的总数。

```
SELECT      S.Market_Id, SUM(S.Sales_Amt)
FROM        SALES S
GROUP BY    S.Market_Id
```

这几个SELECT语句是必需的，因为我们要得到三个聚合，分别是按时间、按产品Id和时间、按市场Id和时间，每一个都需要不同的GROUP BY子句。

独立地计算这三个查询无疑是对时间和计算资源的浪费，第一个查询的很多结果在后两个查询中都会用到。如果我们保存第一个查询的结果然后在Market_Id和Product_Id上聚合，就会节省很多资源。这种形式的计算高效性在OLAP中是非常重要的，研究者们做了很多这方面的工作。例如[Agrawal et al.1996，Harinarayan et al.1996，Ross and Srivastava 1997，Zhao et al.1998]。

CUBE子句的主要目的是可以有效地进行扩展[Gray et al.1997]，SQL:1999标准中包含有这个子句。当在GROUP BY子句中使用CUBE时，形式如下：

```
GROUP BY CUBE(v1, v2, ···, vn)
```

这个查询相当于一组GROUP BY查询，其中每一个对应于$(v1,v2,\cdots,vn)$的一个子集。举个例子：

```
SELECT    S.Market_Id, S.Product_Id, SUM(S.Sales_Amt)
FROM      SALES S
GROUP BY CUBE(S.Market_Id, S.Product_Id)
```

得到的结果如图19-11所示，它相当于上面三个查询，故等价于图19-10。NULL表示是一个聚合。比如Product_Id列中的第一个NULL表示这是超市M1的所有产品的销售总额[⊖]。

⊖ 在这种情况下，SQL中的NULL的使用有些混乱，因为这时NULL的实际含义是所有（它是某一维上的聚合）。

RESULT SET	Market_Id	Product_Id	Sales_Amt
	M1	P1	3003
	M1	P2	6003
	M1	P3	4503
	M1	P4	7503
	M2	P1	1503
	M2	P2	2402
	M2	P3	3
	M2	P4	7000
	M3	P1	15003
	M3	P2	24003
	M3	P3	33
	M3	P4	9902
	M1	NULL	21012
	M2	NULL	10908
	M3	NULL	48092
	NULL	P1	19509
	NULL	P2	32409
	NULL	P3	4539
	NULL	P4	23406
	NULL	NULL	80862

图19-11　用CUBE操作符返回的结果集合

ROLLUP和CUBE有点相似，它们的区别是CUBE在参数的所有子集上做聚合，而ROLLUP选择一些子集进行聚合，选择规则是对GROUP BY ROLLUP子句中的维从右向左依次聚合。SQL:1999标准也包含有ROLLUP子句。

把上面的查询中的CUBE换成ROLLUP，得到下面的查询:

```
SELECT   S.Market_Id, S.Product_Id, SUM(S.Sales_Amt)
FROM     SALES S                                          (19.2)
GROUP BY ROLLUP(S.Market_Id, S.Product_Id)
```

上述查询首先从最小的粒度开始聚合，相当于GROUP BY S.Market_Id, S.Product_Id，然后使用“GROUP BY S.Market_Id”聚合，最后，计算出总数，它相当于空的GROUP BY子句。结果如图19-12所示。如果是更复杂的情况（ROLLUP子句中包含更多的属性），相应的结果表中首先会有一些最后一列是NULL的元组，然后是最后两列为NULL的元组，接下来是最后三列为NULL的元组，依此类推。

OLAP的扩展SQL语句中的ROLLUP操作符其实是在聚合层次结构上的上卷操作的概化。而且，用小粒度上的聚合结果计算较大粒度上的聚合可以节约成本显然也适用于ROLLUP操作符。例如，在查询（19.2）中，子句GROUP BY S.Market_Id, S.Product_Id的聚合结果会在GROUP BY S.Market_Id的聚合计算中用到。这些计算结果会在更大粒度的聚合上用到。

用CUBE操作符物化视图

可以用CUBE操作符来预先把事实表所有维上的所有聚合都计算出来，然后把结果存储到数据库中，供以后的查询使用。

```
SELECT   S.Market_Id, S.Product_Id, SUM(S.Sales_Amt)
FROM     SALES S
GROUP BY CUBE(S.Market_Id, S.Product_Id, S.Time_Id)
```

上述查询产生的结果是图19-1中的所有元组再加上图19-13中的所有元组。将这个结果表存储为物化视图，那么在以后的查询中使用时会提高查询的速度。

RESULT SET	Market_ Id	Product_Id	Sales_Amt
	M1	P1	3003
	M1	P2	6003
	M1	P3	4503
	M1	P4	7503
	M2	P1	1503
	M2	P2	2402
	M2	P3	3
	M2	P4	7000
	M3	P1	15003
	M3	P2	24003
	M3	P3	33
	M3	P4	9902
	M1	NULL	21012
	M2	NULL	10908
	M3	NULL	48092
	NULL	NULL	80862

图19-12 利用ROLLUP操作符返回的结果集

RESULT SET	Market_Id	Product_Id	Time_Id	Sales_Amt
	...	...	...	...
	NULL	P1	T1	6500
	NULL	P2	T1	10800
	NULL	P3	T1	1510
	NULL	P4	T1	9133
	NULL	P1	T2	6503
	NULL	P2	T2	10803
	NULL	P3	T2	1513
	NULL	P4	T2	9136
	NULL	P1	T3	6506
	NULL	P2	T3	10806
	NULL	P3	T3	1516
	NULL	P4	T3	6137
	M1	NULL	T1	7000
	M2	NULL	T1	4633
	M3	NULL	T1	4610
	M1	NULL	T2	7004

图19-13 由CUBE操作符附加到事实表的元组

RESULT SET	Market_Id	Product_Id	Time_Id	Sales_Amt
	M2	NULL	T2	4634
	M3	NULL	T2	16314
	M1	NULL	T3	7008
	M2	NULL	T3	1639
	M3	NULL	T3	16318
	M1	P1	NULL	3003
	M1	P2	NULL	6003
	M1	P3	NULL	4503
	M1	P4	NULL	7503
	M2	P1	NULL	1503
	M2	P2	NULL	2402
	M2	P3	NULL	3
	M2	P4	NULL	7000
	M3	P1	NULL	15003
	M3	P2	NULL	24003
	M3	P3	NULL	33
	M3	P4	NULL	9903
	NULL	NULL	T1	16243
	NULL	NULL	T2	27952
	NULL	NULL	T3	24967
	NULL	P1	NULL	19509
	NULL	P2	NULL	32408
	NULL	P3	NULL	4539
	NULL	P4	NULL	24406
	M1	NULL	NULL	31012
	M2	NULL	NULL	10908
	M3	NULL	NULL	48942
	NULL	NULL	NULL	80862

图19-13　（续）

一些物化了的视图（使用或不使用CUBE操作符）可以在整个OLAP应用中提高查询性能。当然，要存储这些物化视图就需要另外的空间，所以，要考虑物化的视图的个数。由于数据更新很不频繁，故不用考虑物化视图的更新问题。

19.4 ROLAP和MOLAP

可以把OLAP数据库按照星型模式存储在关系数据库中。这种实现称为**关系OLAP**（ROLAP）。

另一些OLAP服务器的厂商使用多维实现（不是关系实现）以**数据立方体**（data cube）的方式实现事实表，同时存储大量预先计算好的聚合结果。这种实现称为**多维OLAP**（MOLAP）。值得注意的是，在ROLAP实现中，数据立方体是看待数据的一种方式；在MOLAP实现中，数据存储在某种可以表示数据立方体的结构中。

CUBE操作符的用途之一就是从SQL数据库中提取数据用来生成MOLAP。许多MOLAP系

统允许用户指定要物化哪些视图。它的数据库中实现了一些在OLAP中常用到的操作（不是关系型的），比如在层次结构的不同级别上的聚合。

对于MOLAP实现，没有统一的查询语言标准。不过很多MOLAP厂商（以及ROLAP厂商）提供了其专用的语言（有的是可视化的），使得那些缺乏计算机相关技术的用户可以只使用简单的查询就得到图19-10那样的表，然后可以在表上执行转轴、上卷、下钻等操作，有时仅仅需要点击鼠标就可以实现查询。

并不是所有的商业决策支持应用都必须用ROLAP或MOLAP数据库服务器，很多应用仍使用传统的关系数据库，只是添加一些针对具体应用的特殊模式。比如，本书中一些复杂的SQL查询可视为OLAP查询（如，*列出教授所有课程的所有教授……*）。实际上，所有带有复杂的联结和嵌套的SELECT语句的查询仅仅可以用来做分析，因为在OLTP数据库上执行这种查询的速度太慢。

19.5 实现中的一些问题

大多数OLAP系统的实现技术都是针对OLAP应用的特点设计的：

OLAP有大量的数据，这些数据比较稳定，很少更新。

而且，很多技术都包括了预先计算部分查询和使用索引，如果设计系统时知道主要执行哪些查询，可以使预先计算和索引有很好的针对性。例如，当它们被嵌入到一个运作OLAP应用中。如果数据库设计者或管理员对要执行的查询有一定的了解时，也可以在即席查询中应用它们。

一种技术是预先计算一些常用的聚合并把结果存储在数据库中。其中包括一些对维层次结构的聚合。由于数据不经常改变，所以维护这些结果的开销不会太大。

另一种技术是针对要执行的查询使用相应的索引。由于数据比较稳定，所以维护索引的开销也是很小的。*联结索引*和*位图索引*是比较常用的两种索引。

1. *星型联结和联结索引*

星型模式中关系的联结叫做**星型联结**（star join），可以用联结索引（在11.7.2节中介绍）来优化这种联结。所有大型商业数据库管理系统的最新版本都支持星型联结。

2. *位图索引*

位图索引（11.7节介绍）对取值个数较少的那些属性尤其有效。在OLAP中，这种属性是很常见的。比如，表MARKET中Region属性只取四个值：North、South、East和West。如果表MARKET有10 000行，属性Region上的位图索引包含4个位向量，大约需要40 000b或5KB的存储空间。这种大小的可以放在内存中用以优化查询。

19.6 数据挖掘

数据挖掘（data mining）的目的是知识发现，即在大数据集上寻找模式和结构，而不是查询特定的信息。如果说OLAP是验证已知的结论，那么数据挖掘就是探索未知的规律。

数据挖掘使用的技术来源于很多领域，例如统计分析和人工智能。我们将引领大家了解

这些技术以及它们如何用于处理大数据集。

1. 关联

数据挖掘中一个非常重要的应用是寻找关联。**关联**（association）就是在数据库中特定值之间的相关性。在1.4节中我们给出过这样的例子：

每天傍晚，大多数在便利店购买了尿布的顾客会同时购买啤酒。

这条关联规则表示为

$$\texttt{Purchase_diapers} \Rightarrow \texttt{Purchase_beer} \tag{19.3}$$

怎样来判定一条关联规则呢？假定便利店有一张PURCHASES表，该表可以从OLTP系统中提取出来（如图19-14所示），数据挖掘系统根据这张表中计算两个度量，用它们来考察关联规则：

- 关联规则的**置信度**（confidence）：同时包含规则两边的项的交易数在包含规则左边的项的交易数中占的百分比。在图19-14的例子中，前三个交易都包含“尿布”，其中前两个交易同时又包含有“啤酒”，所以关联规则(19.3)的置信度是66.66%。
- 关联规则的**支持度**（support）：同时包含规则两边的项的交易数在所有交易中占的百分比。在图19-14的例子中，四个交易中有两个同时包含“尿布”和“啤酒”，所以这条关联规则的支持度是50%。

PURCHASES	Transaction_Id	Product
	001	diapers
	001	beer
	001	popcorn
	001	bread
	002	diapers
	002	cheese
	002	soda
	002	beer
	002	juice
	003	diapers
	003	cold cuts
	003	cookies
	003	napkins
	004	cereal
	004	beer
	004	cold cuts

图19-14　用于数据挖掘的PURCHASES表

置信度用来衡量如果一个交易包含规则左边的项（Purchase_diaper），那么它会同时包含规则右边的项（Purchase_beer）的概率。如果这个概率很高，便利店老板就可以考虑在尿布货架的旁边摆放啤酒。

但是，只凭置信度还不足以确定这条规则的可靠性，这条规则还应该在统计上是有意义的。例如，关联规则Purchase_cookies ⇒ Purchase_napkins的置信度是100%，但只有一条交易满足这条关联规则，那么它是没有意义的。支持度就是衡量规则是否是有意义的，它衡量满足关联规则的交易在所有交易中所占的百分比。

为了保证一条关联规则是有意义的，置信度和支持度都要大于某个阈值。选择合适的阈值属于统计分析领域，本书不讲述这方面的内容。

一条关联规则可能涉及不止两个项。例如，如果一个顾客购买了奶酪和熏鲑鱼，那么他很可能也购买面包圈（如果他只买奶酪，则可能另有它用，那他就不会买面包圈）。

$$\texttt{Purchase_creamcheese AND Purchase_lox} \Rightarrow \texttt{Purchase_bagels}$$

对于这个关联规则，置信度就是包含规则中所有项的交易在包含规则前两项的交易中所占的百分比。

给出一个关联规则，然后计算它的置信度和支持度是很简单的，只需要执行一次OLAP查询就可以了。如果是要寻找所有置信度和支持度大于阈值的关联规则，则要困难得多，这就是一个数据挖掘查询。[Agrawal et al. 1993]首先提出了关联规则的数据挖掘和早期的一些算法。

下面介绍一种有效的算法，用该方法检索相关数据以得到支持度大于给定阈值T的所有关联规则。正如我们将要看到的，一旦找到了这些规则，计算它的置信度是否大于给定阈值就比较容易了。

假设我们要寻找所有支持度大于阈值T的关联规则A⇒B，一种最原始的方法就是计算A_i和B_j的所有可能的不同项的组合的支持度，选出那些大于T的组合。但是如果一共有n个项，则要验证$n(n-1)$种组合，这显然是不可行的。

有一种更为有效的算法称为apriori算法，它基于下面的结论:

如果关联规则A⇒B的支持度大于T，那么A和B的支持度都大于T[⊖]。

(显然，如果在R个交易中A和B都出现，那么A和B分别单独出现的次数肯定不小于R。)

所以首先计算每个项的支持度，找出那些大于T的项，共需要计算n次，假设有m个项满足条件（m很可能比n小得多），然后在这m个项的组合上找出那些支持度大于T的组合（共需计算$m(m-1)$种组合），假设找到p个规则满足条件，那么需要验证$2p$个规则的置信度是否大于阈值。

2. 序列模式

如果在图19-14的表中加入两列Customer_Id和Date，我们还可以挖掘出有关顾客在不同时间的购买情况的规则。

一个顾客买了某个牌子的奶酪，那么以后他还会买这种牌子的奶酪吗?

一个顾客买了垃圾桶，那么过一段时间他会买垃圾桶的盖子吗?

这种就是**序列模式**（sequence pattern），很多挖掘关联规则的技术可以应用到挖掘序列模式上。

3. 分类规则

分类指的是使用数据库中的数据来寻找规则，利用这些规则对将来的结果进行预测。例如，抵押贷款机构利用顾客的年收入和净资产来预测他是否会拖欠偿还贷款。为了达到这个目的，抵押贷款机构可能会分析关联规则

```
((10 000≤P.Income≤50 000) AND (P.Networth≤100 000))⇒(P.Default='yes')
```

是否成立（其中P指顾客）。如果这条规则成立，那些满足左边条件的顾客就会被划分到“可能拖欠”的类别中。分类规则和关联规则的不同之处在于分类规则一般使用属性值的范围而关联规则使用单个值，但支持度和置信度的衡量方法对于二者是相同的。

4. 机器学习

机器学习（属于人工智能的一个分支）领域中有大量技术可以应用在数据挖掘中。接着前面的例子，假如抵押贷款机构想预测申请者是否会拖欠，他可能考虑更多的因素，并且这些因素的重要性是不一样的，要给每个因素赋以不同的权重。

⊖ 单个项（如A）的支持度是包含A的事务的百分比。

要想了解如何应用权重制定决策，假定仍只考虑两个因素：年收入和净资产。赋权重a_1给条件(10 000 ≤ P.Income ≤ 50 000)，将权重a_2赋给条件(P.Networth ≤ 100 000)，就得到了一个表达式

$$a_1x_1 + a_2x_2$$

如果第一个条件满足，则x_1=1，反之x_1=−1；第二个条件也一样。如果对于一个顾客，表达式的值不小于阈值t，即

$$a_1x_1 + a_2x_2 \geqslant t$$

那么，这个顾客就属于“可能拖欠”的顾客，反之就属于“不会拖欠”的顾客。实际上，在计算时还会包含其他许多的因素。现在的问题是：怎样来决定这些权重和阈值？

一种称为**神经网络**（neural net）的技术通过考察已有数据中的信息来“学习”相关的权重，以便预测顾客的行为，并对未来的顾客的行为进行预测。“学习”指的是系统利用数据库以往顾客拖欠或按时还款的信息来逐渐的调整这些权重使预测效果达到最佳。

上面的不等式可以看作基本**神经元**（neuron或者nerve cell）行为的一个模型。如果输入（x_1和x_2）的加权求和大于阈值，则称这个神经元被激活。使用数据库中以前顾客的信息，一个神经元的简单学习算法如下所示：

1) 把所有权重和阈值初始化为0。

2) 每次把一个顾客的信息输入到这个神经元模型（计算上述表达式），看求得的结果是否正确（正确是指如果顾客拖欠了贷款，则表达式的值大于阈值；如果没拖欠贷款，则表达式的值小于阈值）。

a. 如果神经元结果正确，则不对权重和阈值进行改动。

b. 如果表达式本应大于阈值，而计算结果相反，则修改权重和阈值。修改规则为：依据每个输入项是正还是负，把相应的权重和阈值在同方向上乘一个系数，使表达式大于阈值。即，如果x_i=1，则把a_i放大d倍（d是放大/缩小因子）；如果x_i=−1，则把a_i缩小d倍，把t缩小d倍。

c. 如果表达式不应大于阈值，而计算结果却大于阈值，同样要修改权重和阈值。修改规则为：依据每个输入项是正还是负，把相应的权重和阈值在反方向上乘一个系数，使表达式大于阈值。即，如果x_i=1，则把a_i缩小d倍（d是放大/缩小因子）；如果x_i=−1,则把a_i放大d倍，把t放大d倍。

3) 重复第二步（即每次考察一个顾客），直到神经元错误率较小并且权重变得比较稳定为止。这说明神经元“学习到”了顾客的信息，现在就可以用这些权重对未来顾客进行预测了。

这个算法有一个性质[Novikoff 1962]：如果一个神经元总是能做出正确的决定，那么可以通过有限次调整，使得权重和阈值收敛于正确的值。但在实际应用中，结果往往不是这样的，因为只有一个神经元的模型很难真实地模拟现实应用。所以提出用多个神经元组成网络并利用相应的更为复杂的计算权重的算法，这里不详细描述。

应该指出的是，神经网络在信用调查上的使用是机器学习和数据挖掘最成功的商业应用之一。

19.7 数据仓库的数据载入

用于OLAP和数据挖掘的数据通常存储在一个特殊的数据库中，这种数据库称为**数据仓库**(data warehouse)。数据仓库非常大，可能包含在多个时间段从不同的数据源提取出的TB级的数据，还可以包含不同的厂商生产的具有不同模式的数据库。把这些数据集成到一个数据库中不是一个简单的工作。当这些数据需要周期性地更新时，还会产生很多别的问题。

在把这些数据装载到数据仓库前，一般要对它执行两个很重要的操作。

1) *变换*（transformation） 必须对来自不同数据库管理系统中的数据进行语法或语义的变换，使之变为数据仓库需要的统一的格式。

a. *语法变换* 不同数据库管理系统表示相同数据的语法很可能是不同的。比如，社会安全号在有的数据库中用属性SSN表示，而在其他数据库中是用Ssnum表示；可能一个是字符串型，另一个是整数型。

b. *语义变换* 不同数据库管理系统表示相同数据的语义很可能是不同的。比如，数据仓库要求对数据按“天”来进行综合，而有的数据库是按“小时”综合的，而有些数据库则根本不综合，只是记录每次交易的信息。

2) *数据清理*（data cleaning） 必须对数据的错误和缺失进行处理。我们可能会认为其他OLTP数据库中的数据肯定是正确的，但在现实中通常不是这样。另外，一些错误也可能来源于OLTP数据库以外的途径，比如因特网上表格信息的压缩代码错误。

通常情况下，这两种操作都称为“数据清理”。尽管没有明确的设计理论来执行这些操作，但大多数厂商都提供了一些工具来执行这些操作。

如果数据源的模式是关系数据库，和数据仓库的模式非常相似，没有必要进行数据清理，那么可以仅仅使用SQL语句来从数据元提取数据，然后存储到数据仓库中。例如，在一个连锁超市的每家分店的数据库M中，都有表M_SALES模式为M_SALES(Product_Id, Time_Id, Sales_Amt)，它记录了分店M的每个时间段的每件产品的销售情况。那么，在时间段T4后，我们可以使用下面的语句来更新数据仓库中的事实表（图19-1），把M在时间段T4的销售情况添加进去。

```
INSERT INTO SALES(Market_Id, Product_Id, Time_Id, Sales_Amt)
    SELECT Market_Id = 'M', S.Product_Id, S.Time_Id, S.Sales_Amt
    FROM M_SALES S
    WHERE S.Time_Id = 'T4'
```

如果需要数据清理或重新格式化，可以把数据源的数据看做没有物化的视图。在这个视图上通过数据清理程序提取数据（不需要数据源数据模式的信息），然后进行进一步的操作。

和其他类型的数据库一样，OLAP数据库也需要有**元数据库**（metadata repository），用来记录数据的逻辑和物理结构，包括模式、索引等。对于数据仓库，元数据库中还应该包括数据源的信息以及装载和刷新数据的信息。

OLAP数据库的大量数据使得数据的载入和更新都变得很困难。为了提高效率，一般采取增量式的更新，即在不同的时间更新数据库的不同部分。然而，这种更新方式会使得数据库有时处于不一致的状态。一般情况下这不会给OLAP造成很大问题，因为大多数数据分析只是综合性的和统计意义上的分析，不一致状态对这种分析的影响不大。

图19-15描述了装载OLAP数据库的过程。



图19-15　OLAP数据库的数据装载

19.8　参考书目

术语OLAP由Codd在[Codd 1996]提出。[Chaudhuri and Dayal 1997]对联机分析处理进行了很好的介绍。[Fayyad et al. 1996]中包含关于数据挖掘的应用和研究的最新文章。

[Gray et al. 1997]介绍了CUBE操作符。[Agrawal et al. 1996,Harinarayan et al. 1996,Ross and Srivastava 1997,Zhao et al. 1998]讨论了如何高效地计算数据立方体。[Agrawal et al. 1993]提出了关联规则的挖掘思想和一些早期算法。[Han and Kamber 2001]是一本综合介绍数据挖掘的书。

19.9　练习

19.1　典型的BCNF事实表是否符合BCNF？说明原因。

19.2　为什么把星型模式看成E-R模型时，事实表是联系而维表是实体？

19.3　设计一个超级市场进行OLAP时会用到的事实表和相应的维表（不同于前面讲述的例子）。

19.4　解释为什么不能把学生注册系统的数据库建模为星型模式？

19.5　为超级市场设计一个SQL查询，返回结果和表19-10相似，不同之处是在“州”上聚合商店，在“月”上聚合时间。

a. 使用CUBE或者ROLLUP操作符。

b. 不使用CUBE或者ROLLUP操作符。

c. 计算结果表。

19.6　a. 为上述的超级市场的例子设计一个查询，返回每个分店的销售总额。

b. 计算出结果表。

19.7　假设一个应用有4张维表，每张维表有100行。

a. 计算事实表有多少行。

b. 假设一张表中有一个可以取10个值的属性，如果要为这个属性建立一个位图索引，需要多少字节？

c. 计算用来联结一张维表和事实表的联结索引的最大大小。

d. 如果要用CUBE操作符来对事实表的4个维进行聚合，那么结果表一共有多少行？

e. 如果要用ROLLUP操作符来对事实表的4个维进行聚合，那么结果表一共有多少行？

19.8 为ROLLUP操作符设计一个查询评估算法，这个算法的目标是重用以前聚合得到的结果而不是从头重新计算。

19.9 为CUBE操作符设计一个查询评估算法，这个算法使用以前得到的聚合结果来计算新的聚合。(提示：把所有的GROUP BY子句组织成一个格。例如，GROUP BY A > GROUP BY A,B > GROUP BY A,B,C以及GROUP BY B > GROUP BY B,C > GROUP BY A,B,C。描述格中低层次的聚合怎样用来计算高层次的聚合。)

19.10 假设图19-1中的事实表进行了CUBE操作，返回结果存储在一个视图SALES_V1中。设计一个对该表的查询，返回图19-10和19-7的表。

19.11 建立一个OLAP应用来分析我们大学中的分数情况。分数用整数0 ~ 4表示（4代表A），我们想知道不同课程、不同教授、不同系在不同学期、不同学年的平均成绩。给这个应用设计一个星型模式。

19.12 讨论分别用多维数组和事实表实现数据立方体在存储需求上的区别。

19.13 对图19-14中的表执行一个apririo算法，找出所有的二项关联规则。

19.14 在什么情况下，关联规则的置信度总是大于支持度。

19.15 试举出本书所给例子之外的装载数据到数据仓库的语义变换、语法变换的例子。

第四部分

事务处理

现在我们开始学习事务处理。

在第20章，我们将对事务的ACID性质以及这些性质如何与正确的调度相关联给出详细的描述。

在第21章和第22章，我们将描述多种事务模型以及事务处理系统的体系结构。

从第23章～第26章，我们将看到每一个ACID性质如何在集中式系统和分布式系统中实现。

- 在第23章和第24章，我们讨论隔离性的实现，首先是普遍意义上的实现，然后是在关系数据库上的实现。我们将看到SQL支持的各种隔离性级别，以及除了SERIALIZABLE隔离级别，在其他隔离级别下产生正确调度的方法。
- 在第25章，我们将讨论在事务异常中止、系统崩溃和数据库存储介质损坏几种情况下，原子性和持久性的实现。
- 在第26章，我们探讨事务的ACID性质在分布式环境下的实现。

在第27章，我们将研究安全性和因特网商务，包括众所周知的因特网协议（如SSL和SET）。

第20章 事务的ACID性质

我们假设事务在一些应用程序的内部执行，而这些应用程序包含一个或多个作为真实世界企业模型的数据库[⊖]。事务是能完成下列功能的程序:

1) 事务能更新数据库，以反映影响数据库所建模企业状态的真实世界事件的发生。一个例子就是银行的储蓄事务: 事件是顾客把现金交给出纳，事务在数据库中更新顾客的账户信息来反映储蓄。

2) 事务能确保一个或多个真实世界事件的发生。一个例子就是在自动取款机(ATM)上的取款事务: 事件是现金的提取（当且仅当取款事务成功执行时）。

3) 事务能从数据库中获取信息。如打印顾客银行账户信息的事务。

前两项功能之间的差异在于: 在第一项功能里，真实世界的事件已经发生，事务只是更新数据库以反映这一事实; 而在第二项功能里，真实世界的事件是由事务触发的。

一个事务能完成以上三项功能。例如，存款事务能执行以下功能:

1) 更新数据库，以反映顾客把现金交给出纳这一真实世界的事件。

2) 当且仅当事务成功执行时，触发打印存款收据这一真实世界的事件。

3) 从数据库中返回有关该顾客账户的信息。

在2.3节，我们看到一组事务的执行是受某些特殊性质约束的（如ACID性质），它们不适用于一般的程序。在本章中，我们将讨论这一主题。

20.1 一致性

一个数据库相对于其所表示的真实世界企业模型，既充当主动的角色又充当被动的角色。在被动角色中，它维护数据库状态和企业状态之间的对应性。例如，学生注册系统必须准确地维护每名学生的Id和选修每门课程的学生人数，因为这里没有纸质的注册记录。从主动性角色来讲，它强制执行企业特定的规则。例如，选修一门课程的学生人数不能超过数据库中存储的这门课程的最大注册学生数（2.3节约束IC2）。试图选修一门选课人数已满的事务是不能成功执行的。

一致性可以用来说明这些问题，它包含两个方面的内容。

1. *数据库必须满足所有完整性约束*

不是所有的数据库状态都是被允许的，有以下两个方面的原因:

- 内部一致性（internal consistency）。便于在不同的表单中存储相同的信息。例如: 我们可以存储选修一门课程的学生人数以及选修该课程学生的姓名列表，列表的长度不等于选课学生人数的数据库状态是不允许的(2.3节的约束IC3)。

⊖ 一些人可能认为数据库不是对真实世界的建模，而是对真实事件一部分的建模，即数据库的内容实际上决定着真实世界的状态。

- 企业规则（enterprise rule）。企业规则限制企业可能的状态。当企业规则存在的时候，数据库可能的状态也相应地受到限制。以上的选课学生人数和最大注册人数的关系规则就是一个例子。选课人数大于最大注册人数的状态是不允许存在的。

这些约束称为**完整性约束**（integrity constraint），有时也称为**一致性约束**（consistency constraint）。每个事务的执行必须保持所有完整性约束。

2. 数据库必须基于真实世界企业状态来建模

从这一意义上来讲，事务必须是正确的，并按其语义来更新数据库。新的数据库状态必须反映真实世界新的状态。例如，注册事务必须增加数据库中存储选课学生人数的变量的值，并将学生添加到选修课程学生姓名列表中。注册事务成功执行但不更新数据库而使数据库保持一致性状态，这时数据库的状态并没有显示学生选课成功。同样，在某储户账户上记录存款的储蓄事务，不更新数据库而使数据库保持一致性，数据库的这种状态就不能反映真实世界的状态。

我们可以通过检查数据库快照（或许在这一时刻没有事务执行）中数据项的值，来判断它是否满足所有完整性约束。但这并不表明数据库的状态能准确地反映真实世界中企业的状态。问题在于仅数据库处于一致性状态是不够的，每个事务也必须满足一致性。

事务的一致性（transaction consistency）　事务设计者可以假设，在事务开始执行时，数据库满足所有完整性约束。事务设计者必须保证，事务执行完成后，数据库仍然会满足所有完整性约束，数据库的新状态是事务规格说明中描述的变换的反映。

这里的一致性有两层含义。一是数据库的一致性，指数据库满足所有完整性约束。另一层含义是事务的一致性，如果处于隔离状态事务的执行使处于一致性状态的数据库变化到新的一致性状态，那么数据库的新状态也要满足事务的规格说明要求。

记住，编写一致性的事务是编写事务程序的应用程序员的基本职责。其他事务处理系统也假设满足一致性并提供原子性、隔离性和持久性，这些性质在确保一致性事务的并发执行时是必要的，即使在出现故障的情形下，也能维持数据库状态和企业状态之间的关系。

完整性约束的检查

SQL提供一些维持完整性约束的支持。在设计数据库时，特定的完整性约束可以组织成SQL断言、键约束等。例如，PRIMARY KEY约束可以消除记录在数据库中的两名学生具有相同ID的可能（2.3节的约束IC0）。约束CHECK能维持注册人数与最大注册人数之间的关系。ASSERTION能确保，某门课程的教室能容纳的学生人数大于最大选课学生人数。在一个事务执行时，如果要访问模式的约束中包含的数据，数据库管理系统会自动检查以防止事务违反约束，并阻止任何违反约束事务的执行。

但不是所有的完整性约束都能写入模式中。即使能写入模式，有时我们也不这样设计。相反，约束是在事务程序内部检查的，这样决策的理由是检查约束非常耗时。如果把约束编入模式中，修改任何约束所引用的表都会引起约束的自动检查，从而导致这种不必要的检查经常进行。把约束检查放在事务中后，只有在可能违反约束时，约束检查才会执行。例如，选课人数限制不可能在注销选课时违反，因而在注销事务执行过程中，即使选课人数发生改

变时，也不会做约束检查（既不做自动检查，也不必做其他检查）。这样做的优点是事务设计者只需把约束检查的代码放在有可能违反约束的事务中。

事务内部的约束检查有一些严重的弊端：增加编程错误的可能性，维护代价高。假设在某些情形下，将约束IC2修改为，选修某门课程的学生数不能超过该课程的教室所能容纳的学生数。利用内部约束检查，可能要修改多个事务，并重新编译和测试。但如果IC2完全在模式内定义，不依赖于任何事务，则容易修改，且不必修改其他事务。

作为一个工作单元的事务

每个事务都要满足完整性约束的要求限制了设计者向应用程序中的每个事务分派任务。为解释这种限制，一些作者把事务定义为完成一个“工作单元”的程序，这表示在真实世界中的事件发生时，应用程序中的每个事务必须按要求完成更新数据库，并维持完整性的所有工作（以及完成其他必要的活动）。所以当一名学生注册一门课程时，更新选修课程人数的事务和更新课程花名册的事务必须同时执行，因为这两个事务都不满足一致性约束要求。在这种情况下，“工作单元”要求二者同时更新，必须由一个事务来完成以保持一致性的约束。

20.2 原子性

系统中负责事务管理和数据库管理系统访问控制的部分称为TP监控器（TP monitor）。除一致性以外，TP监控器还必须为事务如何执行提供其他保证，其中一项保证就是*原子性*。

原子性（atomicity）　*系统必须确保事务一直运行到完成，或如果没有运行完成，将不产生任何影响（就像没有运行一样）。*

常规的操作系统通常不保证原子性。在（常规）程序执行期间，如果系统崩溃，系统崩溃前程序对文件所作的任何修改，在系统重新启动后都将保留下来。如果这些变化使文件处于一种不正确的状态，那么操作系统不负责纠正。

这种行为在事务处理系统中是不可接受的。一名学生要么注册选修一门课程，要么不选修课程。不完整的注册选修过程不但没有任何意义，还会使数据库处于一种不一致的状态。例如约束IC3说明，一名学生在注册选修课程时，数据库中的两项信息都必须同时更新。如果选课事务部分执行，在一部分信息更新后，另一部分信息更新之前系统崩溃，这就会导致数据库处于不一致的状态。

如果事务成功执行，且系统保存执行结果，我们说事务已**提交**（committed）。如果事务没有成功完成，我们说事务**异常中止**（aborted），这时系统必须撤销（或回退，rolled back）事务对数据库所做的任何修改。我们将在25.2节看到，一个事务处理系统包含复杂的异常中止事务处理机制和回退事务影响的机制。

从上面的讨论可以得出以下重要的结论：

原子性的执行表明事务要么提交，要么异常中止。

让我们再看另外一个例子。自动取款机上的取款事务至少包含两个动作：在顾客的账户上记载取款数目和提取相应的现金。事务原子性的执行表明，如果事务提交，这两个动作都必须发生；如果事务异常中止，这两个动作都不发生（其中一个动作是数据库的更新动作，

另一个动作是真实世界的现金提取动作)。

分布式事务是在不同地点访问数据库的事务。例如分布式银行事务可以在两个银行的两个账户上转移存款。原子性要求：若事务提交，则同时更新，若事务异常中止，则不做任何修改。

1. 事务为什么会异常中止

事务异常中止可能有以下几方面的原因，一种可能是在事务执行期间系统崩溃（事务提交前)，另一种可能是在分布式事务中，系统中的某个数据库崩溃。其他可能的原因有:

1) 事务进入死锁状态，不能获得继续执行的资源。

2) 事务的继续执行可能违反完整性约束。

3) 事务的执行可能违反隔离性要求，这意味着可能发生20.4节所描述的与另一个事务错误地交互执行。

最后，事务本身可能决定异常中止。例如，用户可能按取消键，或事务程序可能遇到一些导致事务放弃计算的条件（与应用相关的)。大多数事务处理系统有一个异常中止处理过程，在事务异常中止时调用。严格来讲，这样的一个过程是没有必要的。因为事务本身能进行相应的异常中止处理，撤销任何对数据库所做的修改，然后提交。不过这是一个细致而又易出错的任务，它要求事务记住对数据库所做的修改，并保存足够的信息使数据库能恢复到修改前的状态。既然系统必须包含一个异常中止处理过程，以处理崩溃和其他的一些情形，所以这样的过程适于所有的事务。事务设计者可以不必为异常中止处理编程。

2. 事务编写规范

每一个事务处理系统必须提供一套编程规范，以便程序员界定一个事务。这些规范在不同的系统中是不同的。例如，事务以begin-transaction 命令开始，它的成功执行可能以commit 命令表示。

调用commit命令表示一个提交要求。系统可以决定提交事务，或由于前面讨论的理由而异常中止。rollback命令用来事务自身中止，和提交请求相反，回退请求总是系统所称道的。

当一个分布式事务请求提交时，参与事务执行的各个地点的计算机执行**提交协议**(commit protocol)，以决定请求是否被批准。不管是分布式事务还是非分布式事务，如果系统决定提交事务，就执行提交操作。在提交操作执行前，事务必须处于一种未提交状态（或还有可能异常中止的状态)。提交操作执行后，事务处于提交状态（不可能再处于异常中止状态)。从这个意义上来讲，提交操作是原子性的，不存在把提交状态和未提交状态分开的中间状态。如果在提交的过程中系统崩溃，待恢复的事务要么已经提交，要么尚未提交。我们将在25.2节对提交操作进一步讨论。

20.3 持久性

事务处理系统的第二个要求是不丢失信息。例如，如果你选修一门课程，不论是发生硬件故障还是发生软件故障，你都希望系统保留这一信息。即使第二天因冰雹导致断电和计算机瘫痪（或者即使这在你提交事务后的一微秒发生)，你还是能上这门课。常规的操作系统不提供这种持久性保证。备份可以被保存，但不能确保最近的修改是持久的。硬件故障不仅仅局限于CPU和内存的故障，在大容量的存储设备发生故障时，数据也有可能丢失。因此，我

们给出如下的持久性定义：

持久性（durability）　系统必须确保一旦事务提交，不管后来计算机和存储介质是否发生故障，其执行结果必须反映在数据库中。

持久性可通过在不同备份设备上冗余存储数据来实现。这些设备的特性导致系统有不同程度的**可用性**（availability）。如果设备速度快，系统可以提供**无间断**（nonstop）服务。例如，将带**磁盘镜像**（mirrored disk）的数据库备份保存在两个不同的大容量存储设备上，能对两个数据库快速更新。即使一个设备故障，保存在另一个设备上的数据库仍然能够提供服务。结果，故障对用户是不可见的。电话系统就有这样的要求（尽管在实际情况下并不一定总是满足这样的要求）。

如果备份设备速度慢，那么发生故障后，在数据库**恢复**（recovery）期间，不能为用户提供服务。大多数的航班预定系统属于这一类型，当系统暂时不能提供服务时，就会让预定航班的乘客非常气愤。学生注册系统也属于这一类型。

在真实世界里，持久性是相对的。在发生以下事件时我们仍希望已提交的数据仍能保留在系统中。

- CPU 崩溃。
- 磁盘故障。
- 多个磁盘故障。
- 火灾。
- 恶意攻击。

对于不同的事件，维持持久性的代价是不同的。每一个企业有必要确定其要保持的持久性的程度，确定可能影响持久性的某些特定的故障，以及为保证持久性所愿意付出的代价。

20.4　隔离性

在讨论一致性的时候，我们注重单个事务的影响。下面我们来看一组事务集执行所产生的结果。如果一组事务集里的一个事务在另一个事务开始执行前执行结束，那么我们说事务集是按顺序执行的，即**串行**（serially）执行的。串行执行的优点是，如果所有事务都满足一致性，那么开始处于一致性状态的数据库就会一直维持在一致性状态。当事务集的第一个事务开始执行时，数据库处于一致性的状态，因为事务也是一致性的，所以在事务执行结束后数据库仍满足一致性要求。第二个事务的执行也是如此，一直到事务集中的事务全部执行结束为止。

串行执行对性能要求一般的应用程序是可行的，但对有严格时间响应和吞吐量的应用程序却不适用。幸运的是，现代计算机系统的CPU和输入输出处理器是由一组处理器组成的，能并发执行多个计算和多个输入输出。另一方面，事务通常是一个**顺序执行的程序**（sequential program），即需要CPU参与的局部变量计算和要求I/O从数据库读信息与向数据库写信息交替执行的程序。对这两种情况，每次都需要单独的处理器服务。现代计算系统能同时为一个以上的事务提供服务，我们称这种执行模式为**并发执行**（concurrent execution）。并发执行适合在事务处理系统中为多个用户提供服务。这样，在任意给定的时间内，会有多个

正在执行和部分已执行完的事务。

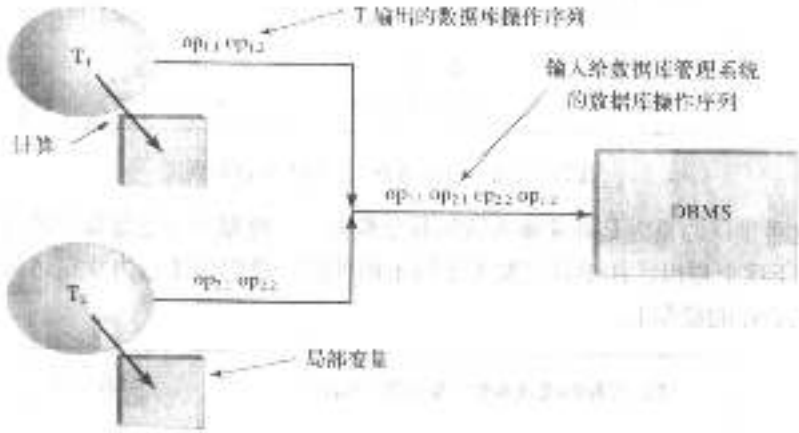


图20-1　在并发调度中，两个事务对数据库的输出操作交错进行。
注意，其中op_{11}先于op_{21}到达数据库管理系统

在并发执行过程中，如图20-1所示，不同事务的数据库操作随时间交错进行。事务T_1在使用局部变量计算和请求数据库操作之间交替操作。例如，一个操作可能是在局部变量和数据库之间传输数据，也可能是对某些数据库变量执行更新操作。请求由操作序列op_{11} op_{12}的序列组成，我们称之为**事务调度**（transaction schedule）。T_2的计算方式也同样如此。因为两个事务的执行不是同步的，所以传输操作到达数据库的顺序（称之为一个调度(schedule)）是两个序列出现的某种可能情况。在图20-1中，这个序列是op_{11}, op_{21}, op_{22}, op_{12}。

当事务并发执行时，每个事务保持一致性并不能保证数据库保持一致性。实际上，开始处于一致性的状态的事务虽然在提交时保证数据库处于一致性状态，但其执行过程中的中间状态不一定是一致性的。另外，一个开始处于一致性状态的事务，在其中间状态读取变量值的行为是不可预测的。假设注册人员定期执行审计事务打印注册学生和注册课程记录。如果事务在注册事务更新选课记录后，更新班级花名册表之前执行，则此时打印出的信息是不一致的，因为注册某门课程的学生记录总数就会少于注册记录中的学生数。在这个例子里面，尽管审计事务打印出来的信息是不一致的，但最终数据库还是会达到一致性的状态。

并发执行可能会破坏一致性，在图2-4中，我们阐明两个注册事务的并发调度，导致更新丢失而产生的不一致。因为有31名学生注册了某门课程，但只有30名学生选修这门课程，所以最终的数据库状态没有反映这一现实事实，因为班级花名册中有31条记录，所以数据库状态不满足完整性约束IC3。

图20-2也说明了这种**丢失更新**（lost update）。在这种情况下，两个银行存款事务交错进行。假设这里唯一的完整性约束是账户余额大于零。事务T_1试图存款5元，T_2存款20元。第一步，这两个事务读到的余额都是10元。第二步，T_2写入30，T_1写入15。最终的值是15元，事务T_2的更新被丢失。注意，数据库最终状态是保持一致性的，因为余额大于零。但实际是不正确的，余额应该是35元。如果事务顺序执行，那么T_1执行完成才允许T_2执行，T_2读到的余

额是15元，那么这两个事务的更新都会反映到最终的数据库状态中。

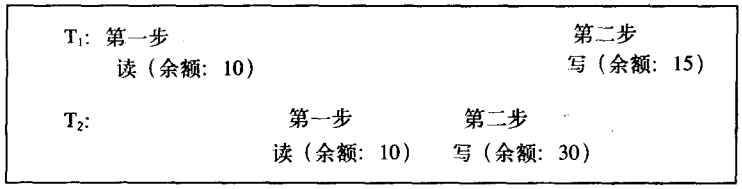


图20-2　两个存款事务没有相互隔离的调度

导致这种错误的原因是并发事务访问共享数据——数据库（这是操作系统中讨论的关键问题）。基于这个原因，在学习并发事务的正确性时，我们把注意力放在访问共享变量的操作——对数据库的操作上。

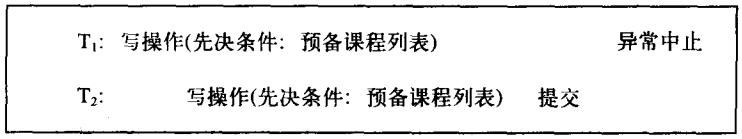


图20-3　不保持隔离性的事务也不保持原子性的调度

并发执行将原子性复杂化。在图20-3中，事务T_1是从$course_2$中删除预备课程$course_1$，T_1先把新的预备课程列表通过w（prereq: new_list）操作写入数据库中，新列表被注册事务T_2读取，基于新列表，学生注册成功完成。但在T_2提交后，T_1转入异常中止状态，旧列表被恢复。学生没有学完课程$course_1$就成功注册$course_2$的事实(违反完整性约束IC1)表明：尽管T_1异常中止，但它已经产生影响，因而T_1的执行是不保持原子性的。

以上例子说明，我们必须为事务的并发执行指定相应的约束，以便保持一致性又保持原子性。这样的约束就是：

隔离性（isolation）　*尽管事务是并发执行的，但并发调度执行的结果就像事务按某种顺序串行执行时所产生的结果一样。*

这一要求的确切含义我们将在第23章和第24章更加详细地阐述。显然，如果事务是一致性的，并且事务在并发调度时所产生的结果与按照某种顺序串行调度所产生的结果一样，我们就说并发事务保持一致性。满足这一条件的并发调度称为是**可串行化的**(serializable)。

注意，常规操作系统通常不保证隔离性。不同的程序可能读写由文件系统维护的共享文件，由于操作系统对读写操作的顺序不加任何约束，所以程序并发执行时不能保证隔离性。

20.5　事务的ACID性质

区分事务和一般程序的特性可简写为ACID［Haerder 和Reuter 1983］特性，即我们前面所讨论的事务特性。

原子性　每个事务要么执行完成，要么不执行。

一致性　一个事务的独立执行将保持数据库的一致性，数据库的新状态是企业新状态的反映。

隔离性 一个事务集并发执行所产生的结果与这些事务串行执行所产生的结果一样。

持久性 提交事务的结果永久地保留在数据库中。

设计出一致性的事务是事务设计者的工作。假设TP监视器提供原子性、隔离性和持久性的抽象。这一假设大大简化了设计者的设计任务，因为它不必考虑事务的异常中止和并发执行。保证事务的原子性、隔离性和持久性是TP监视器的职责。在下列情况下，系统保证每一个调度都将使数据库处于一致性的状态:

当事务满足ACID性质时，事务处理系统保证数据库处于一种正确、一致和实时反映真实世界的状态，提供给用户的数据是正确和实时更新的。

许多应用程序要求保证正确性，例如在线的世界范围的货币交易系统，它必须始终提供正确的服务，因为随时都有可能遭受千百亿的金钱损失。

真实世界中的ACID性质

在后续的几章中，我们将阐述实现原子性、隔离性和持久性可能使系统性能受到影响。例如:

- 实现隔离性时要求事务对它所要访问的数据项加锁。在释放锁之前，阻止其他事务访问该数据项。长期封锁导致长期等待，因而使系统性能遭受损失。
- 原子性和持久性通常是通过维护更新操作的日志来实现的，日志维护是需要开销的。
- 分布式事务的原子性要求事务要么在所有地点的计算机上都成功提交，要么在所有地点的计算机上都异常中止。所以当事务在一个地点的计算机上执行完后不会单独提交，而是要等到所有其他地点的计算机上的事务都执行完成后才提交。在事务提交前，锁必须保持，这可能导致长时间等待。

虽然事务的ACID性质的实现会给性能带来损失，但许多应用处理的事务在执行时还是设计成遵循ACID性质的。对某些应用来说，有些性能损失是无法接受的，如系统无法获得期望的吞吐量和响应时间。在这种情况下，经常牺牲隔离性从而减弱ACID性质来提升性能。例如:

- 有些应用并不要求获得真实世界的准确信息，例如，一个全球供应链决策支持系统，可能允许仓储经理获得每个仓库的库存信息，这些信息对决定何时购买额外商品是很有价值的。在事务完全隔离的状态下，系统产生所有仓库库存的一个瞬时快照，经理们可以根据这一大致的快照做出购买决定，但在快照中，某些仓储的库存报表已经在一小时前更新过或一些仓库根本没有提供它们的库存清单。
- 有些事务处理应用必须为真实世界准确建模，即使事务不是完全隔离的，也要求正确执行。完全隔离是确保任何应用正确执行的充分条件，但不是必要条件。因为有些应用虽然为真实世界准确建模，但事务并不是隔离的（我们将在24.2.2节给出这样的应用实例）。

为改进这类应用执行的性能，大多数商业系统减弱了隔离性级别，不保证可串行化的调度。随后几章将探讨这些问题。基于以上考虑，设计者应该决定应用是否要保持ACID性质，以及是否负担得起保持ACID性质的代价。

20.6 参考书目

关于事务及其实现的精彩描述在[Gray and Reuter 1993]，[Lynch et al 1994,and Bernstein and Newcomer 1997]中可以找到。术语ACID在[Haerder and Reuter 1983] 中给出，但ACID性质的各个部分在早期的论文中就有介绍，例如[Gray et al. 1976], [Eswaran et al. 1976]。

20.7 练习

20.1 一些分布式事务处理系统在两个或两个以上地理上分开的地点复制数据。通常的做法是事务更新一个场地的数据时也必须更新所有场地的数据。因此，在这样一个备份系统中，可能的完整性约束就是所有场地的数据必须保持完全相同。试解释：在这样一个系统上运行时，事务怎样才会违反：

a. 原子性

b. 一致性

c. 隔离性

d. 持久性

20.2 考虑上题所描述的复制存储系统。

a.复制存储系统对只读事务的性能有什么影响？

b.复制存储系统对有读写操作要求的事务的性能有什么影响？

c.复制存储系统对通信系统有什么影响？

20.3 试举出三个除自动取款机取款事务之外的例子，要求当且仅当事务提交时，真实世界中的事件才发生。

20.4 学生注册系统模式包括当前注册的人数和每门课程已注册学生的列表。

a. 数据有哪些完整性约束？

b. 非原子性的注册事务在什么情况下会违反这一约束？

c. 假设系统执行一个事务显示一门课程的当前信息。试给出一个非隔离性的调度，其中，事务显示不一致的信息。

20.5 试举出三个应用实例，其中特定事务不是完全隔离的，尽管不是可串行化调度的结果，但返回的结果足以满足应用的需要。

20.6 试描述程序在本地操作系统上执行时分别不满足以下三种性质的情形。

a. 原子性

b. 隔离性

c. 持久性

20.7 某天中午12点时，有100人同时从某银行的100台自动取款机前从他们各自的账户上提取现金。假设事务是顺序执行的，每个事务花费0.25秒的时间进行计算和输入输出，试估计执行完100个事务需要多长时间，对这100名顾客的平均响应时间是多少？

20.8 你可以选择执行单个事务将300元钱从同一银行的一个账户转移到另一个账户上，也可以运行两个事务，其中一个事务从账户上取300元钱，另一个事务再将这300元钱存入另外的账户。在第一种选择中，传输是原子性的，而第二种方法中的传输不是原子性的。试描述一种在传输后，在传输完成的那一瞬间两个账户的余额不同的情形（和两个事务开始执行时比较）。提示：在资金传输时，同时可能还执行其他事务。

第21章 事务模型

在学生注册系统中，所有的事务都是短事务，且只对单个数据库服务器中存储的数据有少量的访问，但多数应用与长事务有关，这些长事务涉及对多个数据库的访问。例如，一个学生缴费系统中的事务可能要处理一个大学里10 000名学生的学费和房租。在真正的大系统中，事务可能要访问一个网络中存储在不同地点的多个数据库服务器上的数百万条记录。为处理这样的长而且复杂的事务，许多事务处理系统引入相关机制，给事务强加某种结构，或把一个任务分解成多个相关事务。这一章中，我们从应用设计者的角度来介绍这些事务的结构机制。在后续几章，我们将阐述这些机制的实现。

21.1 平坦事务

我们所讨论的事务模型仅涉及单个服务器上的单个数据库，这种事务模型没有内部结构，我们称之为**平坦事务**（flat transaction）。其形式如下:

```
begin_transaction();
    S;
commit();
```

在这里，我们介绍begin_transaction()语句。你可能还记得，在10.2.3节我们所讨论的事务中，SQL-92标准里没有这样的语句(一个事务必须在它的前一个事务结束后才开始执行)。在本节及其后续部分，我们对事务作更加详细的介绍。对事务的抽象作清晰的描述是有益的，为此，我们使用新的语法。begin_transaction()语句通知数据库管理系统，有一个新的事务开始，后续的SQL语句包含在S部分。

事务在使用局部变量进行计算和执行SQL语句二者之间交替进行，这些语句使得数据和状态信息在数据库和局部变量之间得以传递，包括SQL语句和描述符中的输入输出参数。在这个简单的模型中，我们假设事务只访问一个数据库管理系统。计算完成时，事务要求服务器提交或撤销事务对数据库所作的修改。数据库管理系统保证事务的原子性、隔离性和持久性。

为理解这一模型的局限性，考虑以下几种情形。

1) 设想一个旅行计划事务必须为某次旅行预定从伦敦到得梅因的航班。可以采取以下策略:先预订从伦敦到纽约的航班，然后预订从纽约到芝加哥的航班，最后预订从芝加哥到得梅因的航班。现在假设已经预订好前两个航班，但发现从芝加哥去得梅因的航班已经没有座位。事务可能决定放弃预订从纽约去芝加哥的航班，而选择从纽约去圣路易，然后再去得梅因。

设计这样一个事务有几种选择，当预订从芝加哥到得梅因的航班失败时，事务可能异常中止，下一个事务选择经过圣路易的路线。这样设计的困难在于，从伦敦到纽约航班的预订结果将丢失，下一个事务发现从伦敦到纽约的航班已没有座位。另一个方法是事务取消从纽约到芝加哥的航班预订，修改为经圣路易的路线。若该方法可行，就需要撤销以前的操作，

其代码相当复杂。而且，看起来这个事务处理系统还要能够提供一个撤销某些操作的机制。系统中已经提供完全回退的操作，这里需要的是部分回退。

2) 由于得梅因没有国际航班，我们的旅客必须在某个地方换机，可能要预订旅馆和汽车。因此事务不得不访问分布在全世界各数据库服务器上的多个数据库。尽管涉及多个数据库服务器，但我们还是希望事务保持ACID性质。例如，假设我们已经成功预订到一个国际航班，但维护航线的数据库的服务器崩溃（这样无法安排一次完整的旅行），我们需要取消预订。通常，要应用新技术来保证访问多个服务器的事务的原子性、隔离性和持久性。

3) 安排一次旅行不仅包括必要的预订，而且还包括票据的打印和邮寄。这些工作必须完成，但不必同时完成，因为一些工作要求机械操作和人工干预。所以事务操作可能在时间上扩展，正和在空间上扩展一样。允许一个事务创建其他事务的模型是有意义的，这些新创建的事务可以在以后的某个时间里执行。更一般地，一个模型应该能描述整个企业的活动，涉及不同地点和不同时间里由计算机和人共同完成的多个相关工作。

4) 银行在每季度末付利息。一种办法是为每个账户执行一个事务，更新余额和其他与账户相关的信息。如果有10 000个账户，则必须执行10 000个事务。这种做法的问题在于数据库在两个连续事务之间会处于一种不一致的状态：在一些账户上已计入利息而其他账户却没有计入利息。如果这时审计员运行一个事务统计所有账户的余额，总额将是一个毫无意义的数字。较好的办法是在一个事务中计算所有账户的利息。假设这是由平坦事务完成的，在事务为前9000个账户计息后，系统崩溃。由于事务异常中止，前面所有计算结果丢失，这就需要一个即使在系统崩溃时也能保存部分结果的事务模型。

下一节给出解决这类问题及与其相关问题的事务模型。

21.2 提供事务的结构

由平坦事务的介绍可知，应用设计者必须在全做或全不做之间做出选择：运用平坦事务达到原子性、隔离性和持久性，或设计一个不依赖于以上任何性质的应用程序。在本章后面的部分（和后续几章），我们将描述一个可以更好地获得原子性、隔离性和持久性的事务模型和机制，使原子性、隔离性和持久性在不同程度上得以满足。这种灵活性是在事务内部引入结构而获得的。结构化意味着分解，即一个事务分解成以各种方式相互关联的不同部分。在某些情况下，事务的内部结构对其他事务是不可见的。在另外一些情况下，事务ACID性质中的隔离性被违反。

在本节中，我们描述把事务看作是单一的、紧密集成在一起的工作单元的模型。在21.3节，我们描述应用中子任务是松散连接的模型。

21.2.1 存储点

一般来说，数据库提供**存储点**（savepoint）[Astrahan et al.1976]，是部分事务要回退到的事务中的一点。存储点用来标记事务在执行过程中某个特殊的点，一个事务可以定义多个不同的存储点，它们是按数字顺序标记的，以便于区分，使以后的事务能引用某个特定的点。存储点通过对服务器的调用来创建，如

```
sp := create_savepoint()
```

在返回值的地方创建存储点标记。有多个存储点的事务形式如下:

```
begin_transaction();
    S1;
    sp1 := create_savepoint();
    S2;
    sp2 := create_savepoint();
     . . .
    Sn;
    spn := create_savepoint();
     . . .
    if (condition) {
        rollback(spi);
         . . .
    }
     . . .
commit();
```

一个事务能通过发出rollback(sp)请求回退到先前创建的某个存储点处。这里变量sp包含目标存储点的标记。

回退意味着被事务访问过的数据库项的值（或称为**数据库上下文**（database context））返回到创建存储点时的状态，事务撤销自存储点后对数据库所作的任何修改。事务从回退语句之后的语句处继续执行（不是create_savepoint()之后的语句）。

例如，旅行计划事务可能在完成每个单独的航班预订后，创建一个存储点。当发现从芝加哥到得梅因的航班没有座位时，事务就回退到预订完从伦敦到芝加哥航班后创建的存储点处，从数据库中撤销预订纽约到芝加哥航班时所做的修改。其结果就好像事务从来没有想让旅客途经芝加哥一样。在23.7.1节，我们将讨论存储点的实现（特别是如何维护隔离性）。

注意，尽管数据库返回到创建存储点时的状态，但事务的局部变量的状态并没有因事务的回退而改变（即它们没有被回退）。因此这些局部变量的值可能受到影响，因为在存储点创建后，它们可能读取数据库中数据项的值。例如，一个事务在创建存储点后读数据库中数据项x的值，将x的值保存在局部变量$x1$中，然后计算新的局部变量$x2$的值，并将该值写回x中，如果后来事务回退到存储点，x恢复为最初的值，$x1$是当前x的值，$x2$的值不再在数据库中，$x1$和$x2$的值可能影响到以后事务的执行。这意味着回退到存储点给我们一种从存储点到回退点会对事务的执行产生影响的假象。实际上，这种假象通常与实际是不符的，这里重做同样的计算没有任何意义。事务只需知道回退操作已经发生，以后采取不同的执行路径。

注意，数据库在存储点的状态不是持久的。如果事务异常中止或系统崩溃，数据库返回到事务开始执行时的状态。尽管在某种意义上，事务异常中止可以看作回退到初始存储点（没有明确定义），事务异常中止和事务回退到一个存储点有重要的区别：异常中止的事务在执行中止后不可能继续执行，而回退到存储点的事务能继续执行。

同样，在事务执行回退语句rollback(sp_i)后，所有在sp_i之后，回退之前创建的存储点都是不可访问的，因为在以后的计算中回退到这些存储点没有任何意义。

21.2.2 分布式事务

许多事务处理应用的演化方式非常相似。多年来，企业致力于开发使某些活动自动完成

的专用事务处理系统。这些活动包括编制存货清单、账单处理和工资核算等。这样的系统可能由不同的集团在不同的时间、不同的地点，使用不同的硬件和软件平台以及不同的数据库管理系统单独开发出来的。每个事务处理系统导出一个事务集T（可能是存储过程）。在大多数情况下，这些系统运行多年并且很可靠，因此，管理人员不允许对其进行[⊖]任何修改。

随着自动化要求的增加，企业发现，有必要把这些系统集成在一起，以完成更加复杂的活动。在这里，这些系统被称作**遗留系统**（legacy system），因为它们被当作完整的、不可修改的单元，供应用设计者建造更大的系统。同样，系统中的事务也称作**遗留事务**（legacy transaction）。通常，遗留系统的特性使得在用它们来集成大系统时是比较困难的。

例如，库存系统和账单系统可作为构建自动销售系统的组件。如果服务器位于不同的计算机上，集成的第一步就是通过网络连接各个不同的站点。在一些网络站点上，这样一个系统上的事务开始执行时，可能通过启动遗留事务访问多个事务处理系统。尽管可能所有的个人系统都驻留在同一机器上，但通常实际情况不是这样。我们把这样的事务称为**分布式事务**（distributed transaction）或**全局事务**（global transaction）。

上述情况在图21-1中说明，该图显示了管理分布式事务的新语法，该语法是基于X/Open标准的，我们将在22.3.1节讨论这个标准。和begin_transaction()相比，tx_begin()不是对某数据库服务器的调用，而是TP监控器API的一部分，TP监控器控制整个事务处理系统。

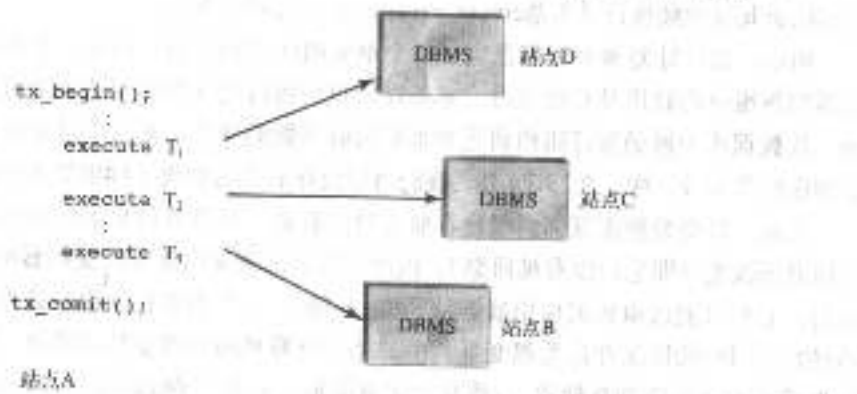


图21-1　分布式事务在多个服务站点调用遗留事务

我们应该了解用这种方法构造分布式事务的优点。事务把账单和库存看作一种抽象，它的逻辑专注于集成遗留事务产生的结果，而不考虑本地数据库模式，不考虑记账和库存过程的处理细节，以及每个站点的如原子性、并发性和持久性等问题。

例如，一个公司可能在不同的地点有多个仓库。对每一个仓库，都有一个事务处理系统为控制库存维护本地数据库。公司的总裁可能想在总部（图21-1中的站点A）执行一个事务产生所有仓库的库存信息。这个事务导致遗留事务（图中的T_1、T_2、T_3）在每个仓库（图中的地点B、C、D）执行以收集库存信息。每个遗留事务将其结果返回给总部的事务，由它采集成信息并产生报表。

⊖ 在某些极端情况下，由于原来执行特定事务的人离开公司很久了，相应的文档已不存在，没有人还能理解事务是怎样工作的。

分布式事务处理应用也可以从头开始构造，而不是由遗留系统集成。一种可能就是整个分布式事务位于一个地点，它建立几个数据库管理系统之间的连接，嵌入SQL语句，然后这些SQL语句被送到这些服务器上执行。在这种情况下，一个服务器上的子事务就是它执行的SQL语句集。

通常，一个分布式事务涉及网络上不同地点的服务器上执行的子事务集。服务器可以提供除数据库（如文件）之外对其他资源的访问。对它所访问的服务器而言，每一个子事务是一个事务，因而保持ACID性质。如果这些服务器是数据库系统，那么我们说分布式事务在多数据库系统之上执行。**多数据库**（multidatabase）（有时也称为**联邦数据库**（federated database））是包含相关信息的数据库的松散联合。

我们假设，各个站点的数据库（称为本地数据库）满足**局部完整性约束**（local integrity constraint）。因为每个站点保持原子性和隔离性，所以多个分布式事务的子事务在那些站点并发执行时，它们是满足约束的。另外，多数据库（由所有本地数据库组成）会有与各站点数据相关的**全局完整性约束**（global integrity constraint）。我们假设一个分布式事务是全局一致的，因此在它隔离执行时，也满足这些约束。

作为全局完整性约束的一个例子，假设一个银行维护各分行的本地数据库，每一个数据库中含有该分行的全部财产数据。银行同时在总行维护着一个数据库。总行数据库中含有银行的总的财产数据。该银行的全局完整性约束可能是总行的全部财产数据必须是各分行中财产数据之和。注意，这一约束不是由各个子事务维护的。在一个分行存款，则在该分行的数据库上启动一个子事务增加该分行的财产值，而不是总行的财产值。为保持全局完整性约束，分行的存款子事务必须伴随着总行的一个子事务，在总财产值中增加同样的值。

除全局一致性外，TP监控器的另一个目标是确保每一个分布式事务(包括它所有的子事务)是原子性、隔离性和持久性的。

- 一个分布式事务的原子性表明它的每一个子事务在其所访问的服务器上是原子性的，还意味着要么提交全部子事务，要么异常中止全部子事务。因此，当分布式事务T中的一个子事务完成操作后不能立即提交，因为T事务的其他子事务可能异常中止（在这种情形下，T事务的所有子事务也必须异常中止）。我们把这种要么都提交要么都不提交的特性称作**全局原子性**（global atomicity）。
- 事务的隔离性不仅表明每一个子事务在其所执行的服务器上是隔离的（即每一台服务器串行化其上执行的所有子事务），而且表明每一个分布式事务相对于其他事务来说也是隔离的（即这里存在所有分布式事务的一些全局串行化序列）。因此，**全局串行化**（global serializability）表示，分布式事务T_1和T_2的子事务在所有服务器上以这样一种方式执行，T_1的所有子事务先于T_2的子事务执行，或者T_2的所有子事务先于T_1的所有子事务执行。
- 分布式事务的持久性表明它的所有子事务也是持久性的。

全局原子性和隔离性足以保证，分布式事务集并发执行所产生的结果与它们按某一串行方式执行所产生的结果一样。因此，我们可以认为，作为一个整体的每一个分布式事务的一致性和可串行化表明，分布式事务的并发调度是正确的。稍后，我们将讨论不同模型，这些模型中的事务不一定满足全局原子性和隔离性，其正确性也是不能保证的。

分布式事务模型

一个分布式事务可以被看作一个树，树的根产生一个子事务集，每一个子事务又产生一个事务集，结果产生一个任意深度的树。一般来讲，它的结构有下面的一些选项。

- 某一子事务的孩子可能或不能并发执行。
- 子事务集的父亲可能或不能与其孩子并发执行。若可以并发执行，父亲可能或不能与其孩子通信。
- 在一些模型中，只有根能请求提交事务。在其他一些事务模型中，任意子事务可以请求提交事务，事务设计者必须确保只有一个这样的事务能提出提交请求。还有一些事务模型，请求提交的权利可以从一个子事务传递到另一个子事务。

在以上这些选项中，可能存在多种事务模型。有两种特定的模型变体占据着主导地位，它们被视为极端情形的例子。

(1) 层次模型（hierarchical model）

事务内部不允许并发执行。启动一个子事务后，父事务必须等到所有子事务都执行完后才能继续执行。结果父事务既不能产生其他与其孩子并发执行的子事务，也不能与其孩子通信。事务是由根来提交的。过程调用是一般模型内部通信的一般形式。TP监控器通常提供一种特殊的过程调用，这种过程调用称为**事务远程过程调用**（TRPC），它除了能调用过程，还支持分布式事务。TRPC将在22.4.1节讨论这部分内容。

(2) 对等模型（peer model）

父亲与孩子、孩子与孩子之间可以并发执行。父事务与其孩子事务的层次关系被最小化：一旦创建孩子事务，孩子与父亲就处在同等的位置上，特别是父亲能与其孩子对等地进行通信，任何参与者都可以提出事务提交请求。在对等模型中，**对等通信**（peer-to-peer communication）是常用的形式。一对子事务可以显式地建立连接，然后通过连接发送和接收消息。TP监控器通常支持对等通信，我们将在22.4.2节讨论这部分内容。

最后，有两种技术通常用来定义分布式事务的边界，这被称作**事务分界标记**（transaction demarcation）。借助于**程序分界标记**（programmatic demarcation），一个应用模块通过监视器提供的API函数，明确告知TP监控器事务的开始和结束。例如，使用X/Open标准，应用程序调用tx_begin()开始一个事务。

声明性分界标记（declarative demarcation）的目的就是从组成应用程序的模块中删除事务边界的规格说明。每一个这样的模块在使用声明性分界标记系统中称为一个**组件**（component），它只处理与应用相关的问题，如访问数据库和维护业务规则，并不对事务执行的上下文进行定义。使用组件的**容器模块**（container module）可定义组件是否作为一个事务执行。因此，不同的模块能决定把同一个组件作为一个事务，或是作为一个子事务，或是作为一个更大事务的非事务的一部分。Microsoft Transaction Server（MTS）是一个使用声明性分界标记的TP监控器。EJB（Enterprise Java Bean）也提供声明性分界标记接口，所以使用EJB构造的TP监控器也能使用这一特性。

21.2.3 嵌套事务

分布式事务从一个需求发展而来，该需求就是将从多个服务器上导出的事务组合成一个

事务单元。因为每一个服务器都支持事务的概念，所以每一个（子）事务都能单独决定是提交还是异常中止。但事务设计者不能控制分布式事务的结构，每一个导出事务的功能基本上是由数据在各服务器上的分布方式来确定的。数据的分布可能受控于这样一些因素，如数据在哪里产生，经常在哪里被访问等。由于采用自底向上的方法，事务可能在分解后不能反映应用原来的功能。

因为嵌套事务没有被视为处理多个服务器和分布式数据的方法，所以它的发展也是不同的。它的目标是让事务设计者利用自顶向下的方式设计一个复杂事务。事务从功能上适当地分解成多个子事务（这种分解方式不是分布式数据专用的），而且，与使用分布式模型相同，由子事务决定提交或异常中止，但决定的处理方式不同。分布式模型使用要么全做或全不做的方法，而嵌套模型中单个子事务能中途异常中止而不会使整个事务都异常中止。即使这样，作为一个整体，嵌套事务仍保持全局隔离性和原子性。

现在已经提出一些具体的、详细的嵌套事务模型，我们来看[Moss 1985]所描述的一个这样的模型。一个事务与它所有的子事务被看作一棵树，树的根称为顶层（top-level）事务，术语父、子、祖先、后代和兄弟的含义与平常无异。没有孩子的子事务称作叶子，不是所有的叶子都处在同一层上。我们假设事务和它的所有子事务都在同一个地点执行。

嵌套事务模型的语义可以总结如下：

1）父事务能顺序地创建孩子事务，使一个孩子事务执行结束后再开始执行另一个孩子事务，它还能定义子事务集并发执行。父事务不能与孩子事务并发执行，它要等到所有子事务都执行完毕后才开始执行，它还能创建新的孩子。注意图21-2所描述的嵌套事务树结构，在图中并没有把并发执行的子事务与顺序执行的子事务区分开来。

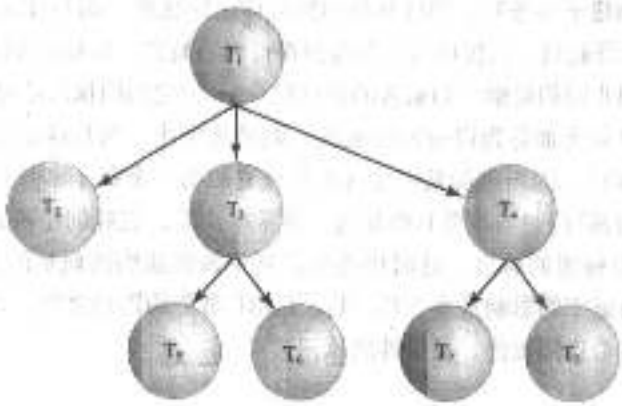


图21-2 一个旅游计划事务的结构

2）子事务（及其所有的后代）相对其并发执行的兄弟事务来说，是一个隔离执行的单元。在图21-2中，如果T_2和T_3并发执行，那么T_2把子树（T_4、T_5、T_6）看作是隔离的事务，而不管其内部结构。兄弟事务集并发执行的结果与它们按照某种串行顺序执行的结果是一样的，因此，兄弟事务对它们自身来讲是可串行化的。

在某些情形下，兄弟事务串行化的顺序影响数据库最终状态。例如，在一个银行应用程序中，两个并发的兄弟事务从同一个账户提款，不管它们的实际执行顺序如何，其执行结果

是一样的（假设账户的初始余额允许这两次提款操作）。但写入的值依赖于它们的执行顺序，因此，嵌套事务是不确定的，因为在不同时间运行同一个事务会产生不同的结果。但如果设计正确，所有的结果对应用来说都是可以接受的。

我们可以把所有的嵌套事务看作是树结构，“嵌套事务中的所有顶层事务的母亲”是根，所有的顶层事务是她的孩子。一个顶层事务（和它所有的后代）相对于其他所有的顶层事务（和它们所有的后代）来说，可以被看作一个独立执行的事务单元。嵌套事务的这种层次结构对事务的外部来说是不可见的。

3) 子事务是原子性的。每一个子事务不能独立地决定是提交还是异常中止。子事务的提交和持久性取决于其父亲事务的提交。因此，当一个子事务的所有祖先事务（包括顶层事务）提交时，这个子事务提交并保持持久性，这时我们说整个嵌套事务被提交。如果一个父亲事务异常中止，那么它所有的子事务（即使已经提交）也异常中止。

4) 如果一个子事务异常中止，那么它的执行结果就像对数据库没有作任何操作一样，同时又回到父事务的状态，父事务可以采取适当的措施。这样一个异常中止的子事务可能会改变父事务的执行路径，对数据库的状态产生影响。这和一般平坦事务的情形相反，一般事务在异常中止时是不产生任何影响的。

5) 子事务不一定保持一致性，但嵌套事务作为一个整体是保持一致性的。

嵌套事务隔离的实现比平坦事务隔离的实现要复杂，因为并发不仅可能发生在嵌套事务之间，而且可能发生在嵌套事务的内部，我们将在23.7.4节讨论这个问题。

下面我们用旅游计划的例子来阐述嵌套事务的模型，事务的嵌套结构如图21-2所示。

事务T_1预订从伦敦到得梅因的航班。它可以创建子事务T_2，预订从伦敦到纽约的航班，T_2执行完成后，它又创建子事务T_3，预订从纽约到得梅因的航班。而T_3可以创建子事务T_5和T_6，T_5预订从纽约到芝加哥航班，T_6预订从芝加哥到梅得因的航班。T_5和T_6可以定义成并发执行的事务，它们可能访问共同的数据（如旅客的银行账户），但它们的执行是可串行化的。

如果T_6不能预订从芝加哥到得梅因的航班，则异常中止。当T_3得知这一异常中止后就放弃经芝加哥的航班预订，因而也异常中止（也导致它的另一个孩子事务T_5异常中止，取消从纽约到芝加哥的航班预订）。当事务T_1得知这一异常中止后，它就创建新的子事务T_4，预订途经圣路易从纽约到得梅因的航班，这时仍然保留从伦敦到纽约的航班的预订。如果T_4提交，T_1就能提交，它对数据库的影响是事务T_2、T_4、T_7和T_8事务的影响之和。事务作为一个整体相对于其他嵌套事务可看作隔离的、原子性的单元。

21.2.4 多级事务

多级事务在某些地方和分布式事务及嵌套事务是相似的。一个事务被分解成嵌套的子事务集。但和嵌套事务不同，多级事务的动机是提高操作性能，目标是允许更多的独立执行的事务并发执行。为理解它如何提高性能，下面将深入讨论这一问题。

隔离通常用锁来实现。当一个事务访问数据项时就对该数据项加锁，迫使其他事务在访问这些项之前等待，一直到锁被释放为止。这样可以防止一个事务看到另一个事务执行的中间结果。如果事务把锁一直保持到事务提交为止，隔离就实现了，但只许有限数量的事务并发，因而这种操作性能的提高也是有限的（和串行执行相比）。

多级事务模型通过允许多级事务中的子事务在整个事务提交前提交，因而释放锁，使得其他等待的事务能较早地继续执行，从而提高操作性能。这样做虽然可以提高操作性能，但导致一个多级事务能看到另一个事务产生的部分结果。相反，在嵌套事务模型中，某个子事务只能有条件地提交，锁不被释放，一个嵌套事务也不会看不到另一个嵌套事务产生的部分结果。但我们将看到多级事务的执行是原子性的和隔离性的。

在这一节中，我们基于[Weikum 1991]的研究来讨论多级事务模型。我们将在23.7.5节讨论它的实现，从操作性能的角度看，这一模型的优点是很明显的。和嵌套事务模型一样，我们假设一个多级事务的子事务在一个单独的地点执行。

1. 多级事务模型

多级事务访问数据库涉及一系列已定义的抽象。例如，在最低级别，数据库可被看作是一些页面的集合，通过读（r）和写（w）操作来访问这些页面。在较高级别，我们可以看到元组的抽象，通过SQL语句可访问这些元组（这些抽象级别通常是由数据库管理系统来提供的）。在更高级别中，我们可以看到面向接口的应用程序。例如，在学生注册系统中，我们可以定义一组对象代表课程的班级，利用抽象的操作将学生从一个班级移动到另一个班级中。这些操作包括测试添加操作TestInc，如果空间足够，有条件地增加学生；减操作Dec从一个班级中将学生删除。

假设在这一抽象级别上，一个事务Move(sec_1，sec_2)将一名学生从一个人数较多的班级$section_1$移动到班级$section_2$中，该事务结构如图21-3所示。应用程序在图中处于最高一级，它设置事务边界，调用TesInc进行测试，并增加$section_2$中的学生人数。如果成功，就调用Dec对$section_1$的学生人数做减法操作。在级别L_2上，TesInc是由一个程序来完成的，该程序使用一个SELECT语句来筛选信息，以决定学生是否可以加入班级$section_2$。用UPDATE语句来增加学生数。在级别L_2，Dec是由一个UPDATE语句来完成的，该语句无条件地对学生数进行减操作。我们假设$section_1$和$section_2$的信息分别存储在元组t_1和元组t_2中。在级别L_1上，这些SQL语句是在读写数据库页面的程序中实现的。元组t_1存储在P_1页中，元组t_2存储在P_2页中。在这个例子里，我们忽略索引页的访问。

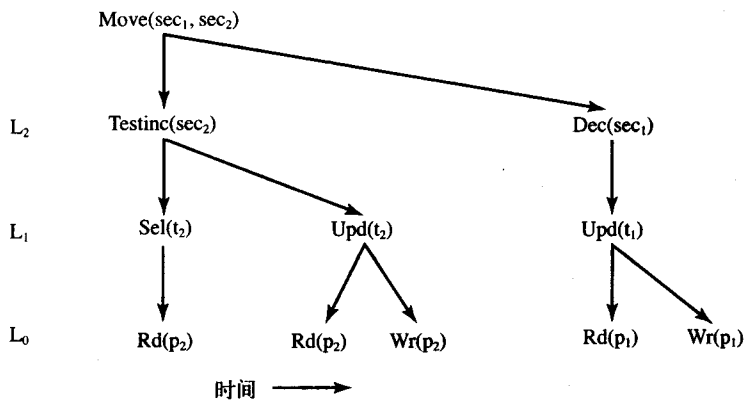


图21-3 多级事务模型中的Move事务

在某些级上调用的操作（L_0除外）可以视为这个级别下面的子事务。因此，在L_2级上调用Upd导致L_1级的子事务的执行，L_1的子事务激活Rd和Wr操作，它们都作为L_0级上的子事务

来执行。类似地，L_2级上的TestInc调用L_1级上执行Sel和Upd操作的子事务。

在一个父子事务创建孩子子事务时，它要一直等到孩子子事务执行完成后才能执行。不过，与嵌套事务模型相比，孩子子事务不是并发的，这意味着多级事务是顺序展开的。多级事务不同于嵌套事务模型还有另外两方面的原因：

1) 事务树上所有叶子子事务都处在同一级别。

2) 只有叶子子事务访问数据库。

2. 多级事务的提交

提交操作是多级事务模型区别于嵌套事务模型的关键所在，也是多级事务能提升性能的关键所在。和嵌套事务模型相比，其子事务的提交是无条件的。多级事务模型中的任意级别上的子事务T在提交时，它对包含它的数据抽象所做的改动对与它处于同一级别且并发执行的子事务来讲是可见的。

无条件提交必须解决两个问题，以便确保多级事务保持原子性和隔离性。

- 隔离性　在整个多级事务提交前，由一个子事务产生的数据库中间状态，对与其并发执行的事务来讲是可见的。例如，在图21-3中，$section_2$中学生数这个数据项，在TesInc提交后（在Dec开始前），对与其并发执行的多级事务来说是可用的。

事务作为一个整体保持完整性约束，但单个子事务却不一定能保持完整性约束。因此，只有在多级事务的最后一个子事务提交后，我们才能确信数据库处于一致性状态。在图21-3中，L_2级上的两个子事务执行期间，被移动的学生在$section_1$和$section_2$中都没有记载。这就导致班级中的人数与课程人数总和不一致的情形，这里假设课程人数总和保存在其他表中。这意味着我们必须考虑并发执行的事务可能看到不一致的状态这个问题。

尽管这个例子中各多级事务可能出现相互之间不保持隔离性的情形（即一个事务能访问与其并发执行的事务的中间结果），但多级事务模型的实现（将在23.7.5节介绍）仍能确保可串行化。从这一意义上说，若事务保持隔离性，则调度的结果与按照某种串行顺序执行的结果一样。

- 原子性　如果一个多级事务必须中止，那么它对数据库所做的所有操作都必须撤销。对平坦事务和嵌套事务T而言，这是没有问题的。即使事务T更新某个数据项x，在新值还没被其他并发事务访问之前，异常中止事务T的一个办法就是将原来的值写回x。这种方式可视为物理恢复（在25.2.3节，我们将对这一方法作详细的阐述）。

撤销多级事务所做的修改更加复杂，因为它的一些子事务可能已经提交，与其并发执行的事务已经看到更新后的数据。设想在Dec子事务D_1提交后，Move事务T_1检测到一个异常中止条件。因为D_1有其孩子事务，所以D_1提交后t_1的新值对与D_1并发执行的事务来说是可访问的。简单地物理恢复t_1到它原来的值不一定有效。例如，有另外一个与T_1并发执行的Move事务T_2，它从同一个班级中移走一名学生，在D_1提交后，它的Dec子事务D_2立刻被调用，然后T_2提交，如图21-4所示。在这种情形下，班级学生数被减两次。由于在D_1执行前t_1的值没有增加，所以如果在T_1异常中止的情况下简单地恢复t_1到它原来的值，则D_2的修改将被丢失，从而使数据库处于不一致的状态。

用于解决这种问题的技术称为**补偿**（compensation）。这种技术不是从物理上恢复原来的值，而是使用**补偿子事务**（compensating subtransaction）从逻辑上进行撤销。Dec可以从逻辑

上成功撤销TestInc操作，因此它是一个补偿子事务。同样，Inc可以从逻辑上撤销Dec操作。类似地，在一个航班预定系统中，cancellation可以逻辑上撤销reservation操作，这样在预订子事务系统中，cancellation就成为reservation子事务的一个补偿子事务。

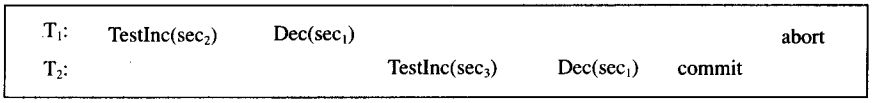


图21-4 在T_1异常中止时，sec_1的数量减少两次，此时，物理恢复产生错误的结果

作为应用设计的一部分，应用程序设计者必须为每一个子事务提供补偿子事务，这并不是一件容易的事。如果CT是ST的补偿子事务，那么它取决于由ST执行后的状态。例如，Dec只有在实际增加后才能补偿TestInc（否则，补偿是没有必要的）。在ST提交后，必须在日志中存储足够的信息使补偿成为可能。对一些应用来说，一个补偿子事务没有必要在逻辑上撤销所有子事务对数据库的更新来实现补偿。例如，在航班应用中，cancellation不会从列表中将旅客的名字也删除。因此，如果不要求原子性，由应用决定的补偿可以不必精确地定义。

一般地，在L_{i-1}级的子事务$ST_{i-1,1},\cdots,ST_{i-1,k}$，提交后，要在$L_i$级异常中止子事务$ST_{i,j}$，则补偿子事务按相反的顺序来执行：$CT_{i-1,k},\cdots,CT_{i-1,1}$，这里$CT_{i-1,j}$补偿$ST_{i-1,j}$。在图21-3中，如果$TestInc(sec_2)$在调用$Upd(t_2)$前异常中止，则可以不采取任何操作（因为$Sel(t_2)$不需要补偿）。但如果在调用$Upd(t_2)$之后中止，就必须执行补偿更新操作，对$t_2$作减法。如果Move在$Dec(sec_1)$提交后中止，就必须按照它的顺序执行$TestInc(sec_2)$补偿。在23.6节，将对补偿做更加详细的讨论。

21.3 把应用分解成多个事务

在很多情况下，把一个普通的事务分解成更小的事务是有必要的。例如将一个长事务分解，使它在中间状态释放锁，可达到改善性能的目标，或者通过在一些中间点提交避免在发生系统崩溃时丢失更多的工作。在有些情况下，一个事务可能包括在不同时间发生的子任务。在本节中，我们探讨提供这种结构的方法以及把复杂业务看作子任务集的工作流管理系统，其中这些子任务是以依赖于应用的某种方式执行的。

21.3.1 链式事务

通常情况下，一个应用程序由一系列事务组成。例如，一个分类订购应用可能是由三个事务的序列组成的应用程序，这三个事务是订购商品、运输、结账。在它们执行期间，程序把订购的商品条目、定购人信息存储在局部变量中，也可以用这些信息进行一些计算。

一个小的优化（称为链（chaining））可以在序列中的前一个事务提交时，自动开始执行一个新事务，从而达到除第一个事务之外，不必对其他事务使用begin_transaction()(也减少了调用数据库管理系统的开销）[⊖]。启用事务链后，由事务序列组成的应用程序有如下的形式：

⊖ 实际上，可能不再需要begin_transaction()了，因为与服务器的第一次交互能自动地开始一个事务。

```
begin_transaction();
    S1;
commit();
    S2;
commit();
     . . .
    Sn-1;
commit();
    Sn;
commit();
```

这里，S_i是第i个事务ST_i的主体部分。

每一个提交语句的执行使前一个事务对数据库所做的修改持久地反映在数据库中。因此，如果在事务ST_i的执行过程中发生崩溃，那么当系统重启时，事务ST_1，…，ST_{i-1}对数据库所做的修改就保留在数据库中。当然，应用程序存储在局部变量中的信息会丢失。

在设计长事务（如前面介绍的学生缴费系统）时，可以从另一个角度来看待链式事务。和自动开始执行一个新事务相反，我们关心的是在系统崩溃时避免回退整个事务。使用链，一个长事务就可以分解成代码片段序列S_1，S_2，…，S_n，这些序列像上面描述的那样链接在一起，其中每一个片段是一个子事务。例如，学生缴费事务可以分解成10个子事务，每一个子事务负责1000名学生的缴费。

在把一个长事务分解成链时需要考虑以下问题：

1）如果在执行子事务时系统发生崩溃，那么只有当前的子事务结果丢失，因为该子事务之前的各子事务结果已持久化（相反，使用存储点的事务在系统发生崩溃时整个事务的结果都丢失）。但是，事务作为一个整体不再保持原子性。在恢复后，系统不负责从发生崩溃的事务重启链。

2）由于各子事务本来是事务的一部分，并且它单独完成一项任务，所以相互之间通常需要进行通信。通过访问共享的局部变量集合可以轻松实现通信。例如，如果学生缴费事务按上述方式进行分解，则一个局部变量可能包含最后一个已缴费学生的索引。当新的子事务开始时，就用这个变量来确定下一个该缴费的学生。使用局部变量通信的问题在于崩溃发生时局部变量不能保存下来，在执行恢复时，确定下一个应缴费的学生是很困难的。

作为一种替代，各子事务可以借助数据库变量来进行通信。例如，最后一个已缴费的学生的索引在子事务提交前保存在数据库的项中。如果在下一个子事务执行期间发生崩溃，就可由数据库中数据项的值指明恢复的地点。在这种情况下，数据库中用于通信的数据项的值，对其他应用程序中正在执行的事务（并发的）来说是可用的，但必须注意确保它不被其他应用程序所篡改。

3）数据库中被链式事务访问的那部分状态称为数据库上下文（database context），它并不是维持在事务链中一个子事务和下一个子事务之间。例如，如果用锁来实现隔离，那么子事务ST_i的所有锁在子事务提交后都必须释放。因此，如果ST_{i+1}是链中的下一个子事务，它访问ST_i访问过的数据项，那么在ST_{i+1}看来，数据项的值可能不同于ST_i提交时的值（因为有不在该链中的其他事务，在ST_i提交后和ST_{i+1}访问前的期间，对值进行修改并提交）。问题在于，和一个长事务相反，虽然链式事务中的每一个子事务是隔离的，但整个链式事务却不具有隔离性。

另一方面，数据库上下文是任何一个事务打开游标后的一种状态。事务提交则关闭游标，由ST_i打开的游标对ST_{i+1}来说是不可用的。

尽管隔离性受到损害，但构造链却能使性能受益。长事务利用锁使得它访问的那部分数据库长时间不可访问。这可能产生性能瓶颈，因为与其并发执行的事务必须等到该事务提交并释放锁后才能访问数据库。通过将事务分解成链接的子事务，锁可以很快释放，瓶颈也就消除。

4) 关于链的最初观点在于，序列中的各个事务是相继自动执行的，前一个事务执行完毕，下一个事务自动开始执行。这样，每一个事务都保持一致性，并且在事务执行期间数据库也处于一致性的状态。当我们把链看作分解一个长事务的机制时，情况就发生变化。尽管整个长事务是一致性的，但各个子事务并不一定保持一致性。这就产生一个问题，因为每一个子事务在提交后都释放数据库上下文，使得它对与其并发执行的事务来说是可见的。这些事务希望数据库处于一致性的状态，因此必须要求各子事务也是一致性的。使用存储点时不会发生一致性问题。因为整个事务是保持隔离性和原子性的。在存储点，数据库上下文并没有被释放，与其并发执行的事务不会看到不一致的状态。

为应对系统崩溃，子事务也必须是一致性的。如果发生崩溃时，链式事务没有执行完，那么系统重启时，部分结果对其他应用的事务来说是可见的。因此链式事务既不具有隔离性，同时它也不具有原子性。

1. saga

[Garcia-Molina and Salem 1987] 提出一个事务模型，称作saga，其中的链式事务原子性是通过补偿来实现的。链中的每一个子事务都有一个补偿事务。如果一个链式事务T_i由子事务$ST_{i,j}(1 \leqslant j \leqslant n)$组成，$CT_{i,j}$是$ST_{i,j}$的补偿事务，那么$T_i$的执行会有以下两种形式。如果$T_i$成功执行，则子事务序列如下:

$$ST_{i,1},\ ST_{i,2},\ \cdots,\ ST_{i,n}$$

如果在$ST_{i,j+1}$执行期间系统发生崩溃，则子事务就会异常中止，执行序列如下:

$$ST_{i,1},\ ST_{i,2},\ \cdots,\ ST_{i,j},\ CT_{i,j},\ \cdots,\ CT_{i,1}$$

这样，就从一方面获得了原子性: 如果T_i的任意子事务异常中止，那么已提交子事务所做的所有变化都会撤销。但它可能不是原子性的，因为在补偿事务执行前，可能有并发子事务读取已提交子事务所做的修改。相反，在多级事务模型中，使用补偿来保证子事务的原子性和可串行化来控制子事务（将在23.7.5节进行描述）。

2. 链式事务的另一种语义

从教学的角度来看，考虑链式事务的另一种语义是有意义的，它能处理传统解释所引起的问题，为将新的语义与常规语义加以区分，我们使用函数调用chain()，现在一个链式事务可以有以下形式:

```
begin_transaction();
    S1;
    chain();
    S2;
    chain();
    ...
```

```
    S_{n-1};
    chain();
    S_n;
commit();
```

Chain()提交一个子事务ST_i（因而是持久的），开始一个新的子事务ST_{i+1}，但它并不释放数据库上下文，并且保持游标。例如，当用锁实现隔离性时，ST_i保持的锁不是被释放而是传递给ST_{i+1}，因此，当ST_{i+1}访问ST_i访问过的数据时，它所看到的数据值和ST_i提交后的值是一样的。因为ST_i对数据库的修改对与其并发执行的事务来说是不可见的，所以单个子事务不再保持一致性，如果ST_{i+1}按这种方法设计，ST_i就可以让数据库处于一种不一致的状态。因而链式事务作为一个整体是隔离的，虽然其性能会受到损害。

从这一语义来讲，故障恢复是非常复杂的。如果恢复只是回退在系统发生崩溃时处于活动状态的子事务，那么隔离性就得不到保证，因为在系统恢复后启动的事务会看到一种不一致的状态。链式事务作为一个整体是不能回退的，因为链中前面的子事务已经提交。这意味着，利用新的语义，当一个链式事务的任意一个子事务提交后，它必须向前回退。如果在执行ST_{i+1}期间系统发生崩溃，那么恢复过程重启ST_{i+1}，并传送ST_i提交后的数据库上下文给ST_{i+1}（即数据库上下文和ST_i锁定时是一样的）。这种重启为作为整体的链式事务提供隔离性和原子性。和链式事务相反，一个平坦事务在发生崩溃后，系统不负责自动重启。

21.3.2 用可恢复队列调度事务

在链式事务中，事务以序列形式组织和处理，前一个事务执行完后下一个事务开始执行。但有时，一个应用要求事务按顺序执行，但不是作为一个单元来依次执行，相反，它要求一个事务执行完后，下一个事务开始执行并运行到结束。和链式事务不同，在一个事务执行完成与下一个事务执行前的间隔里系统可能发生崩溃。

正如我们在前面所看到的，一个分类订购活动可能包含三个任务：下订单、运输和结账，这些任务可能由三个独立的事务来完成，运输和结账事务的工作可以在下订单事务提交后的任何一个时间来完成。不过重要的是，在订购后即使系统发生崩溃，这些事务在后来的某个时间也能够执行。

作为另外一个例子，我们可以考虑一个分布式应用，其中本地站点上的一个事务T要在远程站点上完成某些操作。如果远程操作和事务T整合在一起，涉及动作的调用和接收确认消息的网络延迟都成为事务T的响应时间的一部分。但若不要求远程操作和事务T一起作为一个独立的单元来执行，而是只要它最终执行就行，那么它就可以设计成一个由T来调度后续执行的单独事务，T的响应时间也不会因网络延迟而受到影响。

这样一些应用需要极为可靠的机制来确保将来事务的实际执行，这样的一个机制就是**可恢复队列**（recoverable queue）。可恢复队列有常规队列的语义。它的API允许事务执行队列项的入队和出队操作。事务可以让描述某些工作的信息项入队，在一个后续的时间里，如果事务提交，这些工作必须执行。信息项中的信息和本地状态信息一样必须传递给链中的后续事务。在后来的某个时间，信息项由另一个事务执行出队。第二个事务可能是由服务器启动的过程，它不断地将信息项从队中取出并进行处理。

作为一个例子，考虑用户想对网上第二天即将拍卖的某件物品投标的情形。用户运行一

个事务，将他所投的标放在一个可恢复的队列中。后来，在拍卖发生之前，另一个事务将他所投的标从队列中去除并放置在拍卖的数据库中（注意，拍卖数据库中的项可以依拍卖规则按投标顺序保存）。这样安排的好处在于，只要该用户投的标进入队列，系统就可以对用户的投标快速反应。而另一种安排方式将买主投的标直接放在拍卖数据库中，但由于数据库的操作开销很大，所以会有一个较长的响应时间。

为让一个由已提交事务执行入队的项最终得以执行，队列必须是持久的。持久性意味着队列必须在出现各种故障时都能幸存下来。所以，和数据库一样，队列必须在大容量数据库上冗余存储。事务的原子性要求入队和出队操作必须以下列形式与事务的提交保持协调：

- 如果事务插入一项到队列中，在事务异常中止后，该项必须从队列中删除。
- 如果事务从队列中删除一项，在事务异常中止后，该项必须重置在该队列中。
- 一个事务T插入的项在它提交之前不能被另一个事务从队列中删除（因为我们不知道事务T会不会提交）。

注意，直接在数据库中实现具备上述性质的队列是可能的，将数据项插入队列的事务只是更新用于实现队列的表。问题在于这些表被过多的事务访问。在第23章和第24章我们将看到，大多数的隔离是使用锁协议来实现的，这里事务对共享数据经常延迟更新。从性能的角度来看，把队列作为一个独立的模块来处理是可取的，这种独立的模块是从隔离角度看的一种特殊情况。

队列的恢复可以使用多种调度策略，包括先进先出（FIFO）和按优先权排序。系统还可以使用一个过程检查队列，将某个项从队列中删除。注意，这里即使队列是FIFO的，也并不一定按FIFO的顺序处理其中的项。例如，事务T_1可能从一个FIFO队列中删除队列的头部E_1，在后来的某个时间里，事务T_2使新的队列的头部E_2出队。如果后来T_1异常中止，E_1仍然是队列的头部，结果是第二项E_2先于E_1得到服务，这和FIFO队列的要求是相矛盾的。

组织分类订购应用的一种方式就好像管道一样，如图21-5所示。订单输入职员执行一个事务，它按订购的顺序将订单项插入到一个可恢复队列中供运输事务使用，然后提交。后来，运输服务器执行一个事务让该项出队，按要求完成指定的操作，将要结账的订单插入到第二个可恢复队列中供结账时使用，然后提交。接着，结账服务器执行一个出队事务，完成指定的操作后提交。这样组织方式就像一个管道，因为每一项对应一个购买过程，按顺序从一个队列到下一个队列。

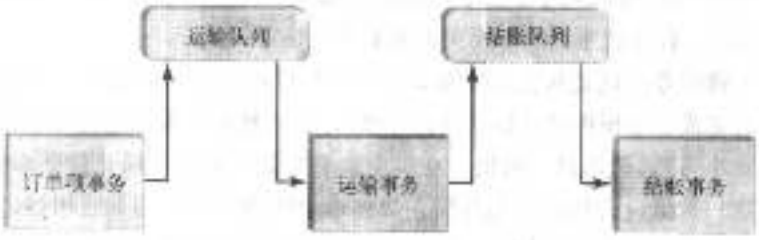


图21-5 以管道方式组织使用可恢复队列的系统

另一种组织方法如图21-6所示，订购事务同时将订单项插入到两个可恢复队列中，一个是结账队列，一个是运输队列，然后提交。后来，处理结账的事务将订单项从结账队列中删

除，完成适当的操作后提交；运输事务将订单项从运输队列中删除，完成适当的操作后提交。这样组织使得同一订单的结账事务和运输事务并发执行（当然，顾客在收货之前先收到账单可能会不高兴）。

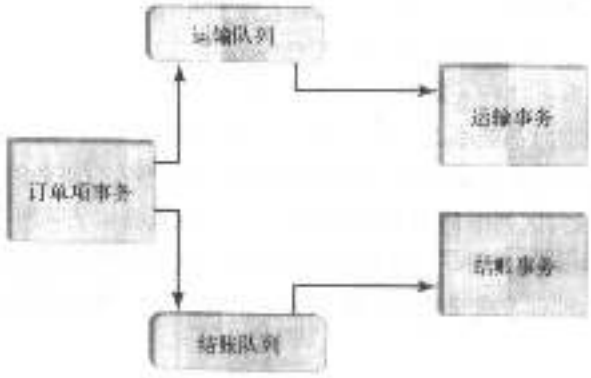


图21-6　使用可恢复队列来获得并发的系统

在分布式系统的例子中，中心地点A的一个事务可能插入一项到本地可恢复队列中，该队列所描述的动作必须在远程地点B执行。后来，一个分布式事务在B启动，同时可能在A产生一个子事务，它从队列中删除那一项，然后发送到B来执行。

1. 用于调度现实事件的可恢复队列

当一个事务要求完成一些实际操作（如打印取款收据）时，可恢复队列可用于获得原子性特征。和数据库的更新不同，现实世界的动作是不能回退的。一旦事务执行，就不能因事务的异常中止（如系统崩溃或死锁引起的异常中止）而回退。这意味着事务异常中止是不能维持原子性的（如一个事务从ATM系统上取现金，在事务提交前系统崩溃会怎样呢？即使在现金已被取出的情况下，数据库中相应于取款的修改也会被回退）。

使用可恢复队列就能获得期望的语义——当且仅当事务提交时，现实动作才会发生。事务在提交前将一个完成现实世界动作的请求插入到一个可恢复队列中。如果事务异常中止，请求就被删除。如果事务提交，请求就被保存下来，由后来完成所需操作的实际事务满足该请求。

但这种解决方法仅适于现实事务。如果事务正在执行的时候系统崩溃又会怎样呢？我们知道请求将保存在可恢复队列中，但我们怎么能知道在系统崩溃之前，事务是否完成操作呢？我们必须知道，在系统重启时，现实事务是否应该重新执行。

解决这个问题的办法就是从完成实际动作的物理设备那里获得帮助。假设设备保留着一个计数器，每次完成一个操作时计数器就会增加1，且计数器的数值可被现实事务读取。在操作完成后，现实事务T_{RW}读取这一数值，在提交前将更新后的值存储在数据库中。在恢复时，系统就会读取这一数值，并将该值与存储在数据库中的值比较。如果值相等，就不需做其他操作，因为上一次现实事务已经提交。如果值不相等，物理设备上的值一定比存储在数据库中的值大1，这表明实际动作已经执行，但由于系统崩溃，相应的现实事务T_{RW}异常中止。因此，在队列头部恢复T_{RW}删除的项，T_{RW}可能已将更新后的值存储在数据库中，也可能没有。即使已经存储，也必须回退。因此系统能够推断出队首项所要求的实际操作已经完成，但相

应的现实事务却没有提交。

2. 用于支持前向代理的可恢复队列

另一个与可恢复队列相关的机制是**前向代理**（forwarding agent）。考虑一个客户机想调用一个服务，却发现目标服务器不可用的情形。如果所要求的服务可推迟，那么客户机可以将它的服务请求插入到队列中，等到目标服务器可用时得到服务。前向代理是在后来的某个时间激活目标服务器的机制。它定期地启动一个事务从请求服务队列中删除一项，激活目标服务器，然后等待响应。如果事务没有成功执行（如目标服务器仍然不可用），它就异常中止，并将删除项恢复到原来的队列中供以后服务。如果事务成功提交，一个响应项就插入到回复队列中供以后客户机使用（如图21-7所示）。注意，目标服务器是不能区分激活它的这两种情况的。它的操作是独立的，不论它是直接由客户机激活的还是由代理间接地激活的。

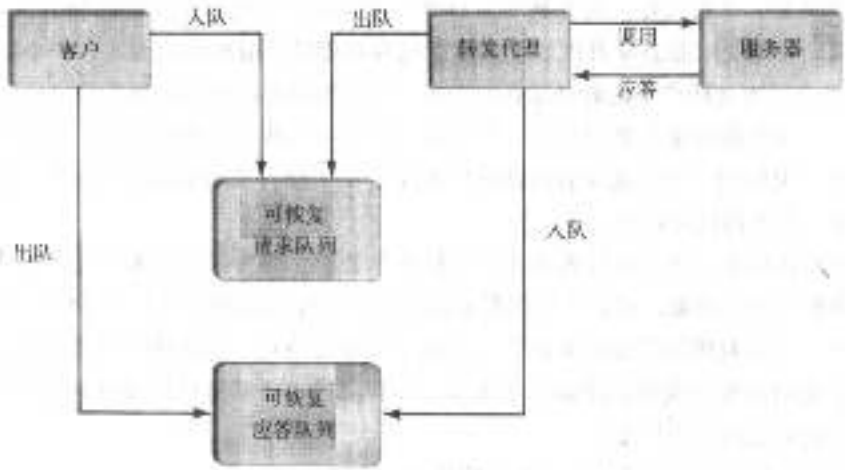


图21-7 使用前向代理激活服务器

3. 可恢复队列作为一种通信机制

可恢复队列可以看作一种可靠的通信机制，通过它，各模块之间可以相互通信。和前面讨论的联机的、直接的通信方法不同，而使用队列的通信是延迟的。这类似于把消息留在电话答录机中。因为队列是可恢复的，即使系统发生崩溃，延迟通信还是能实现。

21.3.3 扩展事务

传统的事务概念及其相关的ACID性质在很多应用中是非常有用的，特别是执行时间非常短（几秒钟或可能是几分钟）、事务访问数据项相对较少、数据库是本地的情形。学生注册系统就是一个典型的例子。

但大部分的应用并不满足这些规则。例如在CAD/CAM、办公自动化、软件开发环境中，事务可能持续几个小时或几天，可能访问不同服务器上的多个数据。为这样的长事务保持隔离性和原子性会严重损害系统的性能。隔离性意味着事务T对数据库所作的修改在它提交之前对与其并发执行的事务来说是不可见的，并发事务必须等待，当事务T是一个长事务并访问多个数据时，系统的性能就会受损。原子性要求一个事务的所有子事务要么全部提交，要么全部异常中止。保证原子性要求服务器工作在原子性提交协议下(我们将在第26章讨论)。但不是

所有的服务器都愿意或能够参与这一协议的。

由于上述原因，事务模型被推广为一种**扩展的**（extended）（或**高级**（advanced））**事务模型**（transaction model）。这里有几种保持原义的事务模型，这无疑是有趣的，它们为商务系统中的实际工作流模型提供支持。

扩展事务模型的共性是它们不再保持ACID性质，这些模型允许应用程序员设计在不同程度上满足原子性和隔离性的事务。因此，必须小心设计扩展事务以免出错。

利用和分布式事务相似的方式，扩展事务被设计成运行在由多个数据库服务器组成的多数据库系统上的事务。扩展事务是由多个子事务组成的，为简单起见，假设每个服务器输出一个事务集，所有事务集组成一个子事务库，从而形成一个扩展的事务。

扩展事务模型将**执行状态**（execution state）与每个子事务相关联。可能的执行状态包括：*异常中止*、*提交*、*正在执行*、*没有执行*、*补偿*、*准备提交*（准备提交状态将在26.2节讨论）。子事务可以处在提交状态意味着扩展事务模型没有必要保证隔离性。因此，一个扩展事务(未提交)的已提交子事务所产生的数据库状态对另一个扩展事务的子事务来说是可见的。

另外，一些扩展的事务模型允许一个已提交的子事务报告其逻辑状态，如*成功执行*或*没有成功执行*。银行的一个不成功的取款操作可能是由于没有足够的资金引起的。注意，在这两种情况下子事务都已经提交。

利用分布式事务，应用设计者必须定义扩展事务的子事务的执行顺序。不是按先后顺序嵌入到事务程序里（例如，对客户机到服务器的远程过程调用的处理），扩展事务模型使用外部规格说明。可以利用图的形式来说明，这里子事务是节点，边表明子事务的先后顺序。也可以用一个规则的集合或控制流语言来表示。不管怎样，子事务可以通过其他子事务的功能输出，也可能是它们的执行状态。

程序员可以用以下方式来规定子事务的启动：

`initiate` $ST_{i,j}$ `when` $ST_{i,k}$ `committed` (21.1)

这里，$ST_{i,j}$和$ST_{i,k}$是扩展事务ET_i子事务。$ST_{i,j}$只有在$ST_{i,k}$提交后才能开始执行。如果$ST_{i,k}$异常中止，$ST_{i,j}$就根本不会启动。下列规格说明

`initiate` $ST_{i,r}$, $ST_{i,k}$ `when` $ST_{i,h}$ `committed`

使一个扩展事务内的子事务$ST_{i,r}$ $ST_{i,k}$并发执行。另外一种设计方法如下所示：

`initiate` $ST_{i,q}$ `when` $ST_{i,k}$ `aborted`

这里$ST_{i,q}$是一个子事务，它用另一种方法来完成$ST_{i,k}$所承担的任务。使用这种技术，就可以指定多条成功执行扩展事务的路径。

例如，$ST_{i,k}$可能是一个旅行计划扩展事务的子事务，它预定一张从纽约到华盛顿的机票，$ST_{i,q}$是一个预定从纽约到华盛顿的火车票事务的子事务，扩展事务可能并发执行$ST_{i,k}$和$ST_{i,q}$，并要求只有在$ST_{i,k}$异常中止时$ST_{i,q}$才能提交。

子事务是可以**重复尝试**（retriable）的，这表明即使在开始时异常中止，但只要它重试足够多的次数，它最终是能提交的。所以当一个可重复尝试的事务异常中止时，没有必要给它指定其他路径。只要规定子事务可重复尝试一直到提交为止即可。例如，一个存款事务是可重复尝试的，而取款事务却是不可重复尝试的（不一定总有足够的资金）。如果一个扩展事务

的所有子事务都是可重复尝试的，那么这个扩展事务总能成功执行。

因为扩展事务模型不一定保证提交的原子性，所以当一个扩展事务异常中止时就引起一个问题。例如，一个用户可能决定在一个扩展事务执行期间取消它，或者一个没有其他可选路径的子事务（非可重复尝试的）可能中途中止。那么正在执行的扩展事务的子事务可能中止执行，其他已经提交的子事务又会怎样呢？它们的影响必须撤销。例如，如果扩展事务ET_i已经预定旅馆和交通工具，在旅行被取消之前，预定航班的子事务ST_{ij}可能已提交。在多级事务模型和saga中也遇到过这样的问题。我们看到，在这些模型中，应使用补偿而不是物理恢复。在扩展事务模型中也应用补偿。区别在于，多级事务模型的补偿确保原子性和可串行化。而在扩展事务模型中（或在saga模型中）却不保证原子性和可串行化。

补偿子事务是扩展事务模型的一部分。只包含补偿子事务的扩展事务总是能撤销的。但有些子事务既不能补偿又不能重复尝试。例如，一个不能退票的预定子事务是无法恢复和重做的（这里不能确保票永远可用），这样的子事务通常被称为pivot。

在非补偿的子事务提交后撤销扩展事务是不可能的。相反，在这点一定有可能执行完。所有后续执行的子事务必须是可重复尝试的（或必须提供其他适当的路径）。因此，一个扩展事务的执行是由可补偿子事务的执行加之可重复尝试的子事务执行的。单个pivot子事务在这两种情况下都能执行。

扩展事务模型假设存在**运行控制器**（run-time controller），它解释先后顺序规则及子事务的性质（如可重复尝试或是可补偿），使每个扩展事务执行时能满足上述要求。

21.3.4 工作流和工作流管理系统

工作流（workflow）是企业里复杂处理的模型，它是一个必须按特定的顺序执行的任务集。工作流中的任务不一定是子事务。例如，分类订购系统可能包括一个由人执行的任务（不是由计算机来执行），它的目的就是在运输之前包装商品。其他任务可以是从库存数据库中删除这一商品的数据库事务。

工作流可以视为扩展事务的泛化。在第22章，我们将看到工作流角色在应用服务器中起的作用。本节所描述的工作流和其他模型有很多共性，但它们的侧重点明显不同。

和我们所讨论的各种模型相比，工作流很少考虑数据库和ACID性。它所考虑的是复杂的自动执行，长时间运行的商务处理，包括在分布式或异构环境下的计算和非计算任务。工作流主要关心的不是并发执行和原子性的维持。工作流中的任务可以是数据库事务，它局部满足ACID性质，但工作流不区分这样的任务和非事务型任务。

工作流里面的每一个任务由一个**代理**（agent）来完成，代理可能是一个程序、一个硬件设备或是一个人。销售跟踪中的代理可以是一个软件系统，而商品包装的代理则可能是一个人。一个工作流是在一段比较长的时间里由多个代理来完成的。并且，通常工作流是在分布式且异构程度很高的环境下工作的，这种环境中可能还涉及遗留系统 。在分类订购的实例中，运输和结账部门可能是完全分开的，并且使用不同的数据库系统。

每一个任务都有一个物理状态，如正在执行、提交、或异常中止。另外，任务的完成可能产生逻辑状态信息，它指示事务执行成功还是失败。例如，在分类订购系统中，T_4可能发现一个错误的顾客贷款利率。由于与应用程序相关的条件，它产生一个逻辑失败状态。

在扩展事务模型中，工作流的任务必须是协同工作的。例如，某个任务可能只有在两个或多个任务执行完成后（一个与条件）才能调度，或多个任务并发执行。

同样，在工作流执行的某个时间点，可能有完成同一个目标的几个任务，但只有其中的一个任务应该执行（一个或条件），执行哪个任务依赖于其逻辑状态，或工作流中前面任务的输出，或依赖于某些外部变量的值（如时间）。

图21-8所示是一个描述分类订购系统的工作流。T_1取出订单，任务T_2从库存数据库中删除订购的商品，同时启动任务T_3和T_6并发执行。T_3把订购项从仓库中删除，T_4执行结账功能。删除订购商品后，T_5进行包装。在结清账务和包装好后，T_7安排运输，可能空运（任务T_8），也可能陆运（任务T_9），但只能选择其中的一种方式。最后，当客户在送货单上签字接收后，数据库更新，表示订购已经完成（任务T_{10}）。

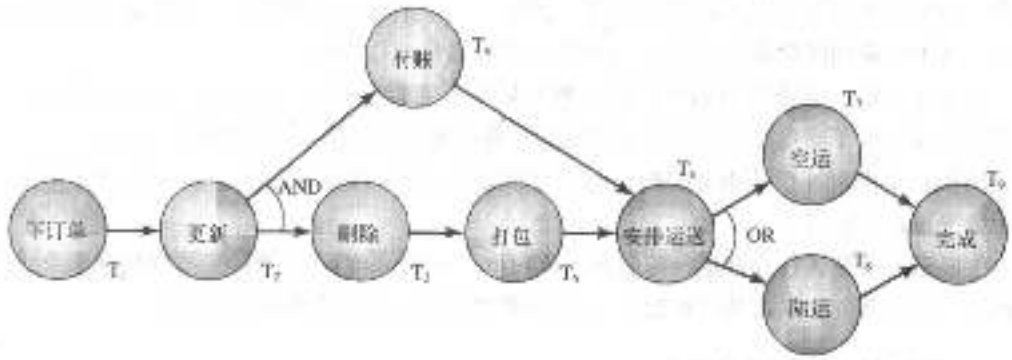


图21-8　表示分类订购系统中任务之间的发生执行关系的工作流

除逻辑失败以外，一个任务的失败可能是一些系统相关的条件造成的，例如服务器关闭。在一些应用中，这样的任务是可重试的，这种情况下，该任务在以后又可以重新运行。顾客可以在任意时刻决定取消购买，这种情形下就说工作流失败。异常中止用在这里不合适，因为一些任务已经完成，且它们的结果是可见的。

处理这种失败的一种方法是使用补偿，为工作流中的每一个任务定义一个补偿任务，发生运行失败时，对每一个已完成的任务按相反的顺序执行补偿任务。例如，T_3的补偿任务将订购项恢复到库存记录中。如果在执行T_7和T_8时发生失败，这些任务就回退，先补偿任务T_2，然后补偿任务T_1。

处理这种失败还可以采取其他策略。如果一个工作流的任务序列为先执行T_A后执行T_B，如果在运行时T_B返回一个逻辑失败状态，工作流设计师就可以指定恢复T_A的任务，然后恢复任务序列中和T_A、T_B一样完成同一目标的其他任务，但这也许不是一种理想方式。

一个任务产生的结果通常是另一个任务的输入，但这不一定和控制流相同。例如，在图21-8中，在T_1执行的过程中，顾客的姓名和地址一起发送给T_4和T_6，而订购商品的ID发送给T_2和T_3，T_2又把订购商品的费用发送给T_4。分类订购系统中的数据流和商品流如图21-9所示。这里就有一个格式问题，一个任务输出的数据在输入到另一个任务之前可能要转换为另一种格式。当信息在遗留系统之间传输时，这样的问题经常出现。

一般来说，工作流遵从这样的事实，即并不是所有的任务都是计算型的，并不是所有的代理都是软件系统。但即使所有的任务都是计算型的，工作流模型和扩展事务模型一样，也

不遵循ACID性质。考虑隔离性。在工作流执行时，某个任务可能释放它访问过的资源，使工作流中的其他事务可以利用这些资源。这意味着一个工作流的中间状态对与其并发执行的工作流来说是可见的。例如，任务T_2和T_3更新不同的数据库。考虑分类订购工作流的两个实例W_1和W_2，它们并发处理两次不同的销售。在W_1中，如果T_2和T_4执行期间，已经执行完成的W_2，那么W_2可以看到这两个数据库的中间状态，这在串行执行的工作流中是不可能出现的。在这类应用中违反隔离性是可以接受的。

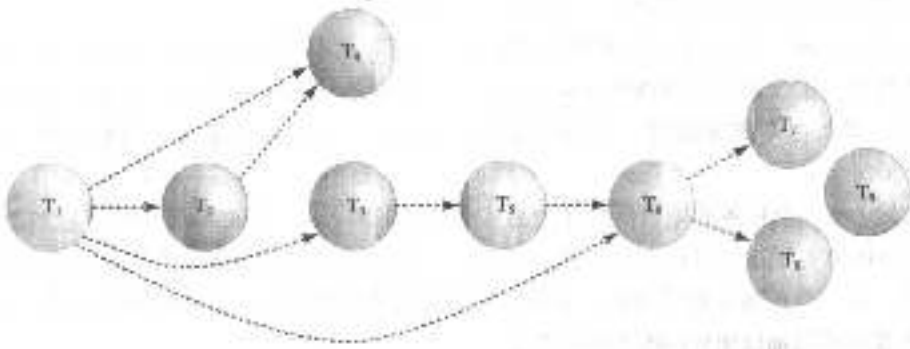


图21-9 图21-8中分类订购系统中的数据流

作为另外一个例子，考虑商务环境中的合作：作为代理的人在完成不同的任务时互相交流信息。这样的任务不是隔离的，因为其中一个任务产生的信息可能影响其他任务的执行。但这种通信方式对某些应用来说是可取的。

另一个非隔离的例子就是对文件更新任务。如果文件没有加锁，工作流中一个任务产生的中间文件状态对其他任务来说是可用的。而且，文件系统一般不提供原子性，即如果因某种原因任务异常中止，文件将停留在这个中间状态。

最后，也可以适当地削弱原子性，因为原子性对一些任务来说是不必要的。我们的分类订购工作流可能包括一个任务，它把顾客的姓名加到邮件列表中。这一任务失败（如邮件列表数据库崩溃）不应该使整个工作流异常中止，也就是说，即使后来广告手册无法送出，但销售必须完成。

工作流管理系统

工作流管理系统（WFMS）为工作流的定义（在设计时）和调度、监控执行提供支持。工作流的规格说明通常不涉及任务细节，但关心任务的顺序和它们之间数据的流动。工作流规格说明涉及图形用户接口（GUI），通过它设计者可以画出类似21-8和21-9的图，该规格说明也可以支持一种语言来表达任务协调规则，如式（21-1）所示。对一个事务处理系统的工作流控制器（见22.2.1）来说，该规格说明能让设计者使用队列或类似方法，来指明事务是顺序执行还是并行执行。

工作流控制涉及根据其规格说明的解释自动执行工作流。这里所说的解释可能涉及以下问题：

- 角色、代理、工作列表　每一个代理有一个属性指定一个角色集。一个角色（role）描述代理所能完成的任务。而每一个任务都有其相关的角色。工作流控制器用角色来确定能完成将要执行的任务的代理。例如，在图21-8中，多个不同的销售代表都可以完成任

务T_1。WfMS可以使用一个算法在各代理之间作出选择，以平衡各代理之间的负载。为方便这样的分配，每个代理可能与当前赋这项任务(来自不同的处于活动状态的工作流)的工作表相关联。

- *任务的激活*　WfMS监视每个任务的物理状态和逻辑状态，当状态发生变化时，由它决定是否满足启动新任务的条件。(前面的任务都完成了吗？所有的输出信息对这项任务可用吗？)它选择并通知已被选中的代理，并把任务加入其工作列表。WfMS还对逻辑和物理失败的情形作出评价，必要时启动补偿任务。
- *状态的维持*　假设执行任务的服务器为每个任务产生的结果提供持久性服务，唯一与持久性相关的问题就是WfMS自身的状态。如果它的状态与各任务的输入输出都能持久维持，那么在系统崩溃时，WfMS就能恢复执行。这样的恢复称为**向前恢复**(forward recovery)。
- *过滤*　当一个任务的输出信息作为另一个任务的输入时，有必要将这些信息重新格式化。WfMS为实现这一目标提供过滤。注意，一个任务的输入可能是由先前的多个任务提供的。在一个任务开始启动前，WfMS要确保当前所有输入可用并且格式正确。而且，过滤器还可以抽取出WfMS所用的信息。
- *可恢复队列*　可恢复队列可以被WfMS当作用来存储活动工作流中任务的信息的机制以及任务的排队机制。

当一个活动比较复杂且必须满足特定的业务规则时，工作流活动的规格说明特别重要。例如，使用分类订购系统的企业管理可能有这样的规则：没有通过信用检查的顾客所订购的商品是不能运送的。工作流中包含这样的规则后，从管理上就能确保即使一天启动很多活动实例(或许经最简单的培训后使用人作代理)，但每个实例都是按已有规则来执行的。为更进一步地与业务规则一致，很多工作流的任务涉及书面批准及先前任务中其他活动的管理。

21.4 参考书目

平坦事务的基本概念已经发展了一段时间，早期的描述包含在[Eswaran et al. 1976,Gray 1981,Gray et al. 1976]，一个更早的实现在[Gray 1978]中介绍。最近的全面描述在[Gray and Reuter 1993,Lynch et al. 1994,Bernstein and Newcomer 1997]中可以找到。存储点在[Astrahan et al. 1976]中有所介绍。多数据库上的分布式事务的相关问题在[Breitbart et al. 1992]中进行了简要介绍。嵌套事务是在[Moss 1985]中提出的。特殊的嵌套事务模型是在[Moss 1985,Beeri et al. 1989,Fekete et al. 1989,Weikum and Schek 1991,Garcia-Molina et al. 1991]中介绍的。嵌套事务是在[Transarc 1996]所介绍的Encina TP监控器中实现的。多级事务在多篇论文中都有所涉及，包括[Weikum 1991,Beeri et al. 1989,Beeri et al. 1983,Moss 1985]。补偿事务的讨论在[Korth et al. 1990]中可以找到，saga是在[Garcia-Molina and Salem 1987]介绍的。一些扩展事务模型的例子包括ConTracts[Reuter and Wachter 1991]，ACTA[Chrysanthis and Ramaritham 1990]和FLEX[Elmagarmid et al. 1990]。多个事务模型的比较可参阅[Elmagarmid 1992]。扩展事务模型的新发展能在[Jajodia and Kerschberg 1997]中找到。

工作流管理的概要介绍能在[Georgakopoulos et al. 1995,Khoshafian and Buckiewicz 1995,Bukhres and Kueshn,Eds. 1995,Hsu 1995]中找到。两个主要问题得到了极大的关注：适合工作流的事务模型的开发和工作流规格说明语言的开发。可以从[Rusinkiewicz and Sheth 1994,Alonso et al. 1996,Georgakopoulos et al. 1994,Alonso et al. 1997,Worah and Sheth 1997,Kamath and Ramamritham 1996]中找到这些问题的讨

论。如果把业务规则作为工作流的一部分，会出现更多复杂的问题。首先，规格说明语言必须丰富足以定义这些规则，第二，工作流必须遵守这些规则，这是一个很大的进步。为让工作流正确执行，很多研究组对工作流正式的定义方法和算法进行了研究，部分研究成果可参考[Orlowska et al. 1996,Attie et al. 1993,Wodtke and Weikum 1997,Singh 1996,Attie et al. 1996,Davulcu et al. 1998,Adam et al. 1998,Hull et al. 1999,Bonner 1999]。Workflow Management Coalition已经开始另一个重要的问题——工作流之间的互操作方面的研究，并在[Workflow Management Coalition 2000]中发布多个标准。

21.5 练习

21.1 本章12.1节描述的学生缴费系统既可以采用在每100名学生后产生存储点的单个事务的结构，也可以采用在每100学生后有提交点的链式事务结构。试解释在为每一位学生打印账单时，这两种结构在执行时语义上的差别。

21.2 试解释在下列两个事务模型在异常中止、提交、回退、隔离性方面的差异:

a. 含一系列存储点的事务和链式事务。

b. 含一系列存储点的事务和由一系列串行执行子事务组成的嵌套事务。

c. 含一系列存储点的事务和由可恢复队列链接的一系列事务。

d. 一系列链式事务和由可串行调度子事务组成的嵌套事务。

e. 一系列链式事务和由可恢复队列链接的一系列事务。

f. 由可恢复队列链接的一系列事务和由串行执行的子事务组成的嵌套事务。

g. 由并发执行的兄弟事务集组成的嵌套事务和由并发执行的同级子事务集组成的分布式事务。

h. 由串行执行的子事务组成的嵌套事务和多级事务。

21.3 把12.6节给出的学生注册系统中的注册事务分解成并发嵌套事务。

21.4 把12.6节给出的学生注册系统中的注册事务重新设计成为多级事务。

21.5 怎样才能把ATM系统上的取款事务构建为由并发子事务组成的嵌套事务。

21.6 a. 试解释书中所讨论的两种链在语义上的差别。

b. 它们中的哪一种可以在SQL内实现?

c. 试举出一个应用的例子，在该应用中第二种方法更为适用。

21.7 试解释如何使用一个可恢复队列让学生注册系统与学生缴费系统相接。

21.8 试举出三个使用可恢复队列但不要求隔离性的应用实例。

21.9 试解释不使用可恢复队列时，完成打印操作的困难。假设事务在提交前可以一直等到成功打印为止。

21.10 考虑在21.3.2节讨论的提取现金的现实事务。在现实事务执行前、执行期间和执行后的每一个时刻系统都有可能崩溃，描述系统（恢复后）如何确定现金是否被取走。讨论现金取款机制本身也可能失败的几种情况，而且这种情况下系统不能判断现金是否被取走。

21.11 试解释在什么情况下，事务中每一条SQL语句的执行与嵌套子事务相似。

21.12 说明如何将第1章第一段描述的确定信用卡的合法性的事务构建为一个分布式事务。

21.13 将学生注册系统和缴费系统（包括伙食和住宿缴费）集成在一起，这两个系统的数据库驻留在不同的服务器上，因此，集成后的系统是分布式的。运用你的想象能力:

a. 举出这个系统的全局数据库上的两个全局完整性约束的例子。

b. 举出都访问数据库的两个事务。

21.14 试把大学的入学办公室接纳新生的过程描述为一个工作流。把这一过程分解成一些任务，用类似

于21-8的图来描绘这些任务之间的相互作用。

21.15　考虑一个在两个银行之间转账的事务。它可以分解为两个子事务：从第一个账户中取款和将款项划入第二个账户。试描述这个过程在层次化事务模型和对等事务模型中如何实现。

21.16　试解释嵌套事务如何使用存储点和过程调用实现。假设每一个子事务的孩子事务不会并发执行。

21.17　试描述一个工作流和一个扩展事务的相似之处和不同之处。

21.18　学生注册系统中的一次会话能否被看作一个扩展事务？说明你的理由。

第22章 事务处理系统的体系结构

事务处理系统是现有最大的软件系统之一，它们应用在广泛的应用领域中，因此对它们的要求差别也很大。一种极端的情况是单用户系统访问一个本地数据库，而另一种极端的情况是多用户系统，在这样的系统中，一个事务要访问分布在网络上异构的多个资源管理器，每一秒可能有成千上万个事务在执行。许多这样的系统都有严格的性能要求。这些系统是企业发展的基础。

为创建和维护这样一个复杂的系统，将其分解为完成不同任务的功能模块是很有必要的。在本章中，我们将探讨这些系统中的模块化结构，从最简单的开始，然后逐步增加结构的复杂性。我们将描述各种模块必须完成的任务，说明这些模块是怎样构成的，并解释它们之间的通信方式。

22.1 集中式系统中的事务处理

在集中式事务处理系统中，所有的模块都驻留在一台计算机上。例如，为一个用户服务的计算机或工作站，或一台连有多个终端为多个用户提供并发服务的主机。我们分别对这两种情况进行讨论。

22.1.1 单用户系统的组织

图22-1显示的是一个用户的事务处理系统在一台PC机或工作站上可能的组织形式。用户模块完成**表示和应用服务**（presentation and application service）。表示服务通常是由应用程序生成器创建的，这已经在3.4节进行过描述。它在屏幕上显示表单，并通过表单处理从用户和计算机之间的信息流。一个典型的循环活动开始于显示表单的表示服务。在表单上，用户在文本框中输入信息，然后单击鼠标或者某个按钮，表示服务把单击作为一个事件，并调用相关程序，这样的程序在3.6节被称为**事件程序**（event program）。表示服务把GUI上输入到文本框中的信息传递到事件程序，由事件程序来完成应用服务。程序检查没有在模式中定义的完整性约束，并按照企业的规则执行一系列的步骤，适当地更新数据库。

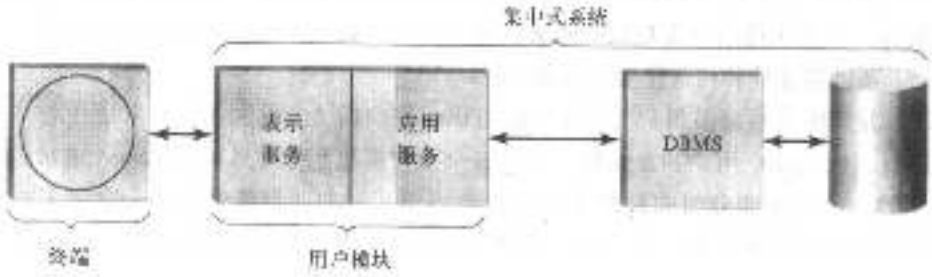


图22-1 单用户事务处理系统

为完成应用服务，事件程序必须与数据库服务器通信。例如，在学生注册系统中负责课程注册的事件程序，必须确保学生已经学完了预备课程，并且必须把学生的名字添加到班级花名册中去。它访问数据库的请求可能定义在嵌入式SQL语句中，这些SQL语句被发送到数据库服务器，数据库服务器将请求转换成一系列访问数据库的读写操作命令（数据库存储在大容量的存储设备上）。这些命令由操作系统来执行。

注意，用户并不直接与数据库服务器交互，而是通过调用事件程序，由事件程序向数据库服务器发送请求。让用户直接访问数据库服务器（如通过执行指定的SQL语句）是危险的，因为粗心的或恶意的用户容易通过写入错误的数据来破坏数据库的完整性。即使只允许用户读数据也有缺陷。一方面，构建一个SQL查询语句返回用户想要的信息不是一件容易的事。另一方面，数据库中可能有不允许用户看到的信息，如一名学生的成绩信息不能被其他学生看到，但教师却可以访问学生的成绩信息。

要求用户必须使用由应用程序（假设用户不能篡改它们）所提供的事件程序来间接地访问数据库可以解决这个问题。这些程序是由应用程序员所实现的，他们设计这些程序来验证对服务器的访问是正确的和适当的。

尽管事件程序（或其一部分）可以被视为事务，但此时并不要求它具有事务处理系统的全部功能。例如，单用户系统一次只调用一个事务，因此自动保持隔离性。但保证原子性和持久性的机制仍然是必要的，不过这些机制实现起来相对简单，因为同一时间只有一个事务在执行。

尽管有人认为单用户系统太简单，可以不包含在事务处理系统中，但它表明在所有系统中必须提供两个必要的服务：表示服务和应用服务。这是我们将要讨论的。

22.1.2 集中式多用户系统的组织

支持任何规模企业的事务处理系统必须允许多个用户从多个地点对它进行访问。这样系统的早期版本使用了连接到一台中央计算机的终端。在终端和计算机局限在一个小区域内（如一幢建筑物内）的情况下，可以通过硬件连接来进行通信。不过，在大多数情况下，终端分布在远程地点，它们与计算机的通信是通过电话线来完成的。早期系统和当前多用户系统的主要区别是终端是“哑”终端，即没有计算能力。它们作为输入输出设备来进行服务，为用户提供简单的文本接口。图22-1的表示服务（除此之外都内置到终端硬件中）是最小的，且必须在中心地点执行。当然，这样的系统在地理上可能是分布的，但通常不认为它们是分布式系统，因为所有的计算和智能业务都驻留在同一地点。

引入多用户处理系统主要源于事务的发展和ACID性质的要求。因为多个用户并发访问系统，必须有一种方法将用户交互彼此隔离。相对于个人数据库，因为现在的系统对企业来说至关重要，所以原子性和持久性变得更加重要。

图22-2显示的是早期多用户处理系统的组织。用户模块包含表示服务和应用程序，运行在中心地点。因为多个用户并发执行，所以每个用户模块在独立过程中执行。这些过程的执行是异步的，它们可能随时向数据库提交服务请求。例如，当服务器为一个应用程序执行SQL语句的请求提供服务时，另一个应用程序可能正在提交执行其他SQL语句的请求。高级数据库服务器能同时为多个这样的请求服务，并保证每个SQL语句作为隔离的和原子的单元

执行。

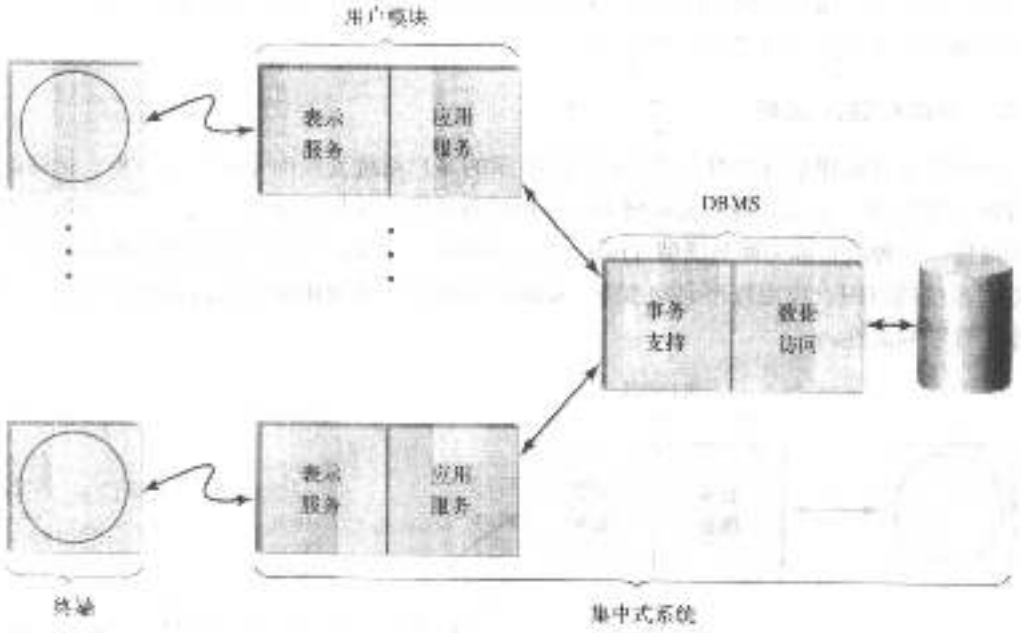


图22-2　集中式多用户事务处理系统

但这并不足以确保不同用户之间的交互是隔离的。为提供隔离性，应用程序需要运用事务的概念。因此，事务支持模块在服务器上提供，以执行begin_transaction、commit和rollback这样的命令，从而提供原子性、隔离性和持久性。模块还包括并发控制和日志，在后面几章我们将详细讨论这些内容。

22.2　分布式系统上的事务处理

现代事务处理系统通常在分布式的硬件上实现，这些硬件包括多个地理位置上分布的独立计算机。ATM机是与银行计算机分离的，售票代理点的计算机是与航班预定主系统相分离的。在有些情况下，应用可以与存储在不同计算机上的多个数据库通信。计算机连接在一个网络上，各模块分布在任意的地点，它们以一种统一的方式交换消息。

这些系统的体系结构是基于C/S模型的。利用分布式硬件，客户机和服务器模块不必处在同一个地点。数据库服务器的分布可能取决于以下一些因素。

- 通信费用和响应时间最小化　例如，一个工业系统是由中心办公室、仓库和生产车间组成的。如果大多数访问员工记录的事务是在中心办公室启动的，那么这些记录就可以保存在本地，而库存记录则可以保存在仓库。
- 数据的所有权和安全性　例如，在每个支行办公室都有计算机的分布式银行系统中，支行可能要求账户信息保存在自己本地的计算机上。
- 计算和存储设备的可用性　例如，一个包含多个大容量存储设备的复杂数据库服务器只能处在一个地点。

记住，在分布式系统中，不同的软件模块处在不同的计算机上，当事务处理系统是分布

式的时候，它的数据库部分（database portion）可以是集中式的，也就是说，可以利用驻留在一台计算机或不同计算机上的多个数据库服务器。因此，事务处理系统可以是分布的（distributed），但有一个集中式的数据库。

22.2.1　分布式系统的组织

分布式事务处理系统的组织是由图22-2所示的多户系统发展而来的。第一步，把终端替换成客户计算机。这样，负责表示服务和应用服务的用户模块位于客户机上，它与数据库服务器通信。这种结构称为**两层模型**（two-tiered model）。图22-3展示拥有集中式数据库的分布式处理系统的组织。应用程序启动事务，如图22-2所示。事务由事务支持模块来处理，以确保原子性和隔离性。

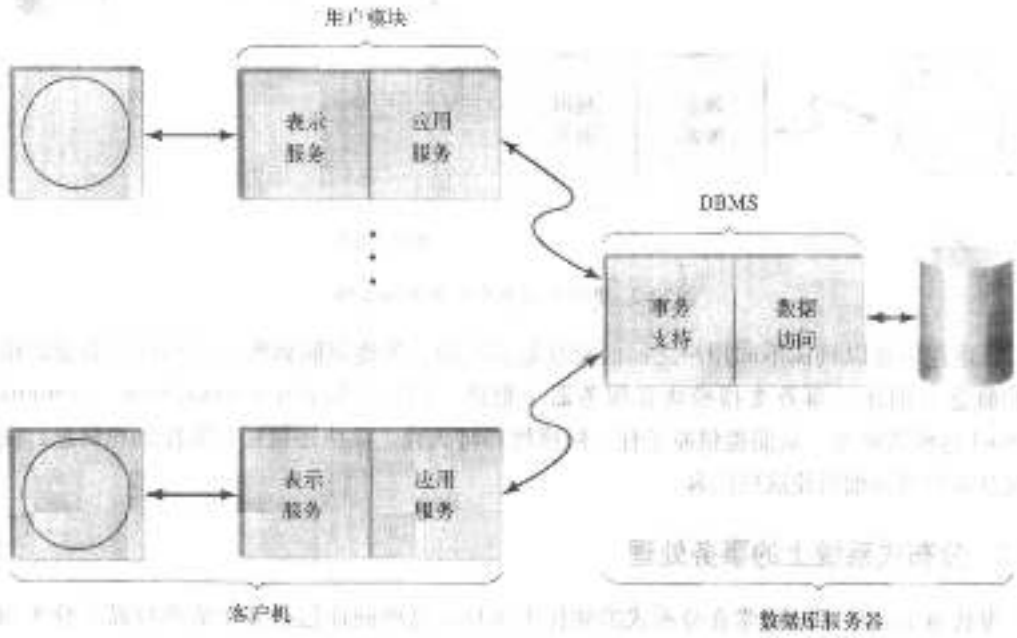


图22-3　两层的多用户分布式事务处理系统

数据库服务器可以提供一个SQL接口，以便客户机上的应用程序向数据库服务器发送执行特定SQL语句的请求。但在客户机直接使用这样的接口时，会出现几个重要的问题。一个问题就是数据库完整性的处理。对单用户系统，我们考虑的是处在客户端的客户机可能不安全和不可靠。例如，一个错误的应用程序可能通过发送一条不正确的更新语句破坏数据库的完整性。

另一个重要的问题与网络通信和系统处理大量客户的能力有关。用户可能希望执行SQL语句来扫描一张表。这导致将表中所有的数据从数据库服务器传送到客户机上。因为这些机器是分布的，即使在事务执行的结果中包含的信息很少时，也要求大量的网络通信。当不得不处理这样的要求时，网络只支持少量的客户。

解决这个问题的办法是采用存储过程。与发送单个SQL语句相反，客户机上的应用程序要求数据库服务器执行某个存储过程。实际上，应用程序现在有高级的或更加抽象的数据库

接口。考虑银行的数据库服务器，它可能为应用程序提供deposit()和withdraw()存储过程。

存储过程有以下几个优点：

- 它们被假设是正确的，在数据库服务器上可以保持完整性。在客户机上运行的应用程序禁止提交单个SQL语句，因为客户机只能通过存储过程访问数据库，因而一致性也得到保护。
- 一个过程的SQL语句可以事先写好，因而执行效率比解释代码更高。
- 服务提供商能更加容易地授权用户执行特定的应用功能。例如，只有银行出纳才能给出有效的支票，而不是顾客。在单个SQL语句级，管理授权是困难的。取款事务可能使用与产生有效支票事务相同的SQL语句。通过拒绝顾客直接访问数据库来解决授权问题。相反，允许它执行取款存储过程却不允许它执行有效支票的存储过程。
- 通过提供更加抽象的服务，网络通信的信息量减少。在客户机上的应用程序和数据库服务器之间，不传递中间变量和单个SQL语句的执行结果，所有的中间结果由数据库服务器上的存储过程来处理，只有初始变量和最终结果通过网络传输。这就是以上的表扫描完成的方式。这样，数据通信量减少，可以有更多的顾客得到服务。但数据库服务器必须有足够的能力来处理存储过程所要求的额外工作。

1. 三层模型

为客户机提供更高层次服务的思路可更进一步实现。对照图22-3的两层系统，图22-4展示的是分布式事务处理系统的**三层模型**（three-tiered model）。用户模块可以分成表示服务器和应用服务器，它们在不同的计算机上执行，客户端的表示服务集成用户的输入信息对信息的有效性进行检查（如类型检查），然后发送请求消息给应用服务器，在网络的某个地方执行。

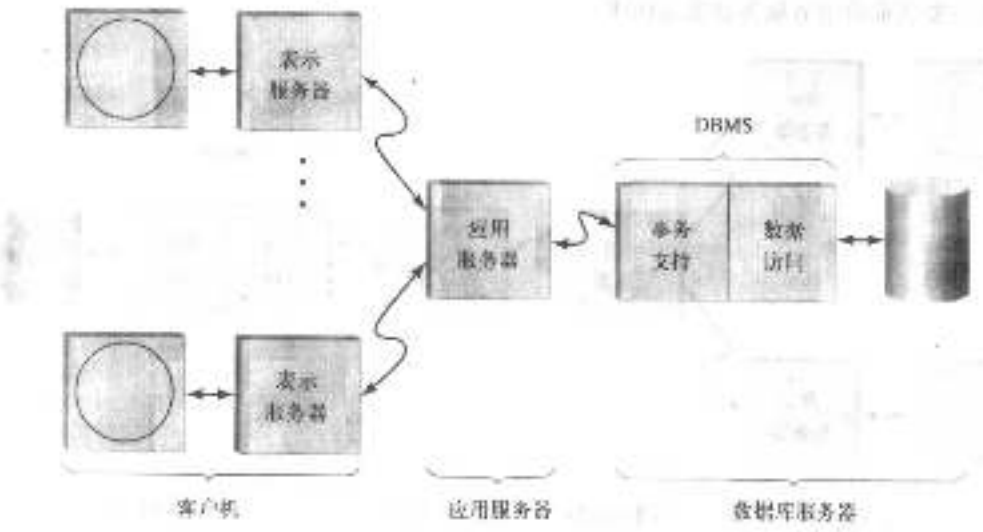


图22-4 三层的分布式事务处理系统

应用服务器执行与请求的服务相关的应用程序，和以前一样，该程序实现企业的规则，依次检查必须满足的条件，调用数据库服务器上相应的存储过程来完成服务请求。应用程序把请求的服务当作任务序列。因此，在12.6.2节给出的学生注册系统设计中，在执行注册课程的任务之前，注册事务检查该课程是否提供，学生是否修完所有的预备课程，且学生选修的

课程总数没有超过限制。

每个任务可能要求复杂程序的执行，每个程序可能是数据库服务器上的一个存储过程。应用程序控制事务的边界，使过程在事务内部执行。并且，它鼓励任务重用，即如果任务被选择来执行通用的有用功能，它们能作为不同应用程序的组件来调用。一般情况下，事务是分布的，应用程序被作为根，存储过程是在不同服务器上执行的子事务的组成部分。这样，就需要更多精心设计的事务支持。我们将在22.3.1节讨论。

应用可以当作工作流（见21.3.4节），它由必须按特定顺序（尽管这种情况下所有的任务都是计算型的）执行的一个任务集合组成。应用服务器可被看作工作流控制器，它控制任务流，按照用户的要求执行。

图22-4展示有多个客户机的应用服务器。因为每个客户机请求的服务都要花一定时间，多个其他客户请求会被挂起。因此，应用服务器必须并发处理请求以维持操作性能。例如，它可以是**多线程的**（multithreaded）[8]，每个线程处理一个客户，通过使用线程，能避免过多地为每个客户创建过程带来的开销。这样，当客户机数量增加时，系统规模可以更好。

如果有多个客户机，可能存在应用服务器的多个实例。每个实例可能驻留在不同的计算机上，每台计算机可能与不同的客户子集相连接。例如，一台应用服务器和它服务的客户可能在地理位置上是相邻的。

在一些组织中，实现应用程序调用的任务的存储过程被移出数据库服务器，在称为**事务服务器**（transaction server）的模块里执行，如图22-5所示。为最小化网络通信，事务服务器通常位于与数据库服务器物理上相近的计算机上，而应用服务器位于接近用户地点。事务服务器现在可以做大量的工作，因为它向数据库服务器提交SQL语句和处理返回的数据。应用服务器主要负责给事务服务器发送请求。

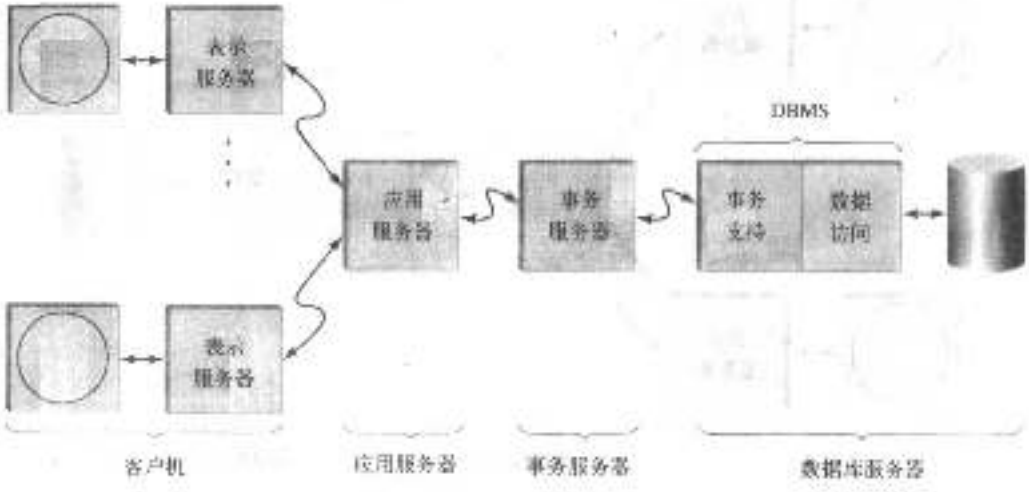


图22-5　在事务服务器中执行任务的三层分布式事务处理系统

尽管在图22-5中没有显示，但这里可能存在多个事务服务器过程的实例。这是一个**服务器类**（server class）的例子，服务器类在服务器过载时使用，类成员可能在与数据库服务器

[8] 在附录A中能找到有关线程的讨论。

相连接的不同服务器上运行，这样可以共享并发执行的工作流出现的事务负载。每个事务服务器过程可能是一个服务器类实例。更一般地，可能有多个数据库服务器存储企业数据库的不同部分，例如，一个服务器上是缴费数据库，而另一个服务器上是注册数据库。某个事务服务器可能连在数据库服务器的子集上，只能执行例程的一个子集。这样，一个应用服务器必须调用合适的事务服务器来完成特定的任务，这种选择调用通常称作**路由**（routing）。分布式事务的应用程序将请求路由到多个事务服务器上，如图22-6所示。

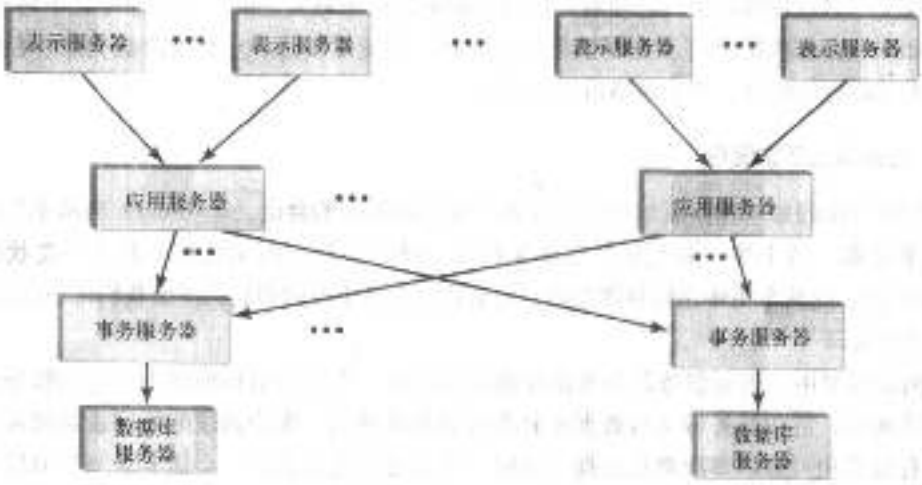


图22-6 三层体系结构中表示服务器、应用服务器和事务服务器的相互连接

在下列情况下，将事务服务器和数据库服务器分开是有益的：

1）企业不同的组件使用不同的过程集访问公共数据库。把这些过程集分别放在不同的事务服务器上可以使每个组件更容易地控制自己的过程。例如，一个大公司的账务系统和个人系统可能使用完全不同的过程访问同一数据。

2）过程必须访问不同服务器上的不同数据库。

3）数据库服务器不支持存储过程。

4）数据库服务器必须处理大量用户的请求，因而有可能产生瓶颈。把存储过程从数据库机器移出可减轻负载。

将客户机和应用服务器分开的优点如下：

1）客户机可更小（因而也更便宜）。当应用包含成千上万个客户机时，这一点尤为重要。

2）系统维护更加简单，因为企业规则的改变（导致工作流程序的改变）局部化在应用服务器上，不会反映到所有的客户机上。

3）系统安全有所加强，因为用户不对应用服务器做物理访问，因而不易改变应用程序。

2. 案例研究：三层体系结构和软件抽象级别

从软件工程的角度来看，设计复杂应用的方法是形成多级抽象的体系结构。后一级使用前一级实现的抽象进行构建。最底层应用完成有意义的工作，在此基础上直接构建应用的功能。在学生注册系统中，这样的活动包括预备课程检查和缴费。按照12.6.3节的实现方法，这层由代码组成，其方法包括：checkCourseOffering()、checkCourseTaken()、checkTimeConflict()、checkPrerequisites()和addRegisterInfo()。它们通过SQL语句直接与数据

库进行交互，在数据库的概念级的顶层进行操作。第二层抽象由事务组成，如课程注册。其实现由方法Register()来表示。注意，这个方法使用低级任务，如checkCourseOffering()和checkPrerequisites()，从数据库相关的角度考虑是得到了防护（如实际的SQL查询）。顶层抽象包括与用户交互的模块。例如，让学生填写选课表单的GUI。

现在我们看到，应用设计的抽象级别是与三层结构体系相对应的。例如，在学生注册系统中，检查预备课程的实际代码（即方法checkPrerequisites()）可以作为存储过程保存在数据库服务器上。当过程被调用时，所有处理都在数据库服务器上完成，只返回是或不是的回答。相反，如果预备课程的检查在应用服务器上执行，它就要发送SQL请求给数据库，数据库可能返回大量的查询结果，增加网络阻塞的可能。

22.2.2 会话和上下文信息

我们所讨论的每一种体系结构都涉及客户机与服务器的**会话**（session）。当两个实体要彼此通信来完成一些工作，并且每个实体保持着一些与工作中的角色相关的**上下文状态信息**（context）时，这两个实体之间存在会话。会话可以存在于不同的层。下面我们将予以讨论。

1. 通信会话

在两层模型中，表示服务器与数据库服务器通信。在三层结构模型中，表示服务器与应用服务器通信，应用服务器又与数据库和事务服务器通信。客户请求的处理通常涉及通信模块之间有效的和可靠的多次消息交换。这时，通常建立通信会话。会话要求每个通信实体保持上下文信息，如在每个方向上发送的消息的顺序号（为了可靠地发送消息）、信息寻址、密钥和当前通信的方向。上下文信息保存在称为**上下文块**（context block）的数据结构里。

信息交换通过建立和取消会话来实现，这使得这些活动是有代价的。结果，每当客户发出请求时，表示服务器就建立会话，每当应用服务器要从事务或数据库服务器得到服务时，也要建立会话，这是很浪费的。相反，服务器之间应建立长期会话，在这段时间内，每个服务器为客户或事务传递消息。

关于会话，使用三层结构模型的好处从图22-6中可以看出来。大的事务处理系统可能涉及成千上万的客户机和成百上千的事务服务器。在最坏的情况下，两层模型的每个客户机与每个事务服务器都有一个连接（长期的），总的连接数将是2的次幂形式，这可能是一个非常巨大的数字。建立这些连接的总开销及上下文块的存储费用会大幅增加。

通过引进应用层，每个客户机只需要与应用服务器建立一个连接，应用服务器只需与事务服务器建立连接。但应用服务器与事务服务器之间的每个连接都能被应用服务器上运行的所有的事务使用。更精确地说，如果有n_1台客户机，n_2台应用服务器和n_3台事务服务器。最坏的情况下，在两层体系结构上有n_1*n_3个连接，而三层结构上连接数是$n_1+n_2*n_3$。因为n_2远远小于n_1，所以这大大减少了连接数。因此三层体系结构模型非常适用于处理大量客户机的系统。

2. 客户机/服务器会话

服务器维护它为之提供服务的每个客户机的上下文信息。考虑一个通过游标访问表的事务。它执行一个SQL语句序列（OPEN，FETCH）产生一个结果集，检索其中的行。上下文必须保存在服务器上，以便它能处理这些语句。因此，对FETCH语句，服务器必须知道最后返回哪一行。更一般地说，如果典型的客户交互包含一系列的请求，服务器不想认证客户，

并决定为哪一个请求服务。

供客户机/服务器会话使用的上下文信息可以以多种不同的方式保存。

(1) 存储在本地服务器

服务器能在本地保存客户机的上下文信息。每次客户机发出请求时，服务器检查客户机的上下文，并用上下文信息解释请求。注意，当客户机总是访问同一服务器时，这种方法的效果很好。但当请求的服务由服务器类的任意实例提供时，必须谨慎使用这种方法。此时，后续服务可能由不同的实例提供，存储在本地实例中的上下文信息对其他实例来说是不可访问的。而且，当客户机很多，会话时间很长时，保持会话是沉重的负担。上下文信息保持的形式经常与对等通信一起使用（参见22.4.2节）。

(2) 存储在数据库上

如果服务器是一个数据库管理系统，那么上下文信息能保持在数据库中。这种方法可避免服务器类带来的问题，因为类的所有实例可以访问同一个数据库。

(3) 存储在客户机上

上下文信息可以在客户机和服务器之间来回传递。服务器完成服务请求后给客户机返回其上下文信息。客户机不会试图解释上下文信息，而是将它保存下来，当下次发出请求时，将它再传给服务器。这种方法既能减轻服务器保存上下文信息的负担，又能避免服务器类带来的问题。上下文信息数据结构的来回传送称作**上下文处理**（context handle），这种上下文信息保持的形式经常与过程调用通信一起使用。

到目前为止，在对客户机/服务器上下文信息的讨论中，我们主要关注与一系列客户机对服务器请求相关联的上下文信息。更一般地，上下文信息必须与事务关联在一起作为一个整体（这可能包括对不同服务器的多个请求，也可能包括多个会话）。这种需求如图22-7所示。在这里，一个客户机与服务器S1建立一个会话，与服务器S2建立另一个会话。这两个服务器都使用服务器S3来完成客户机的请求。但问题在于，隔离性和原子性是和作为一个整体的事务联系在一起的，而不是和某个特定的会话联系在一起。例如，锁（在第23章讨论）用来实现隔离，S3必须使用事务的上下文信息来确定锁，它是在处理S1发出的请求时获得的，对特定的事务而言，可用于为同一事务中从S2来的请求服务。

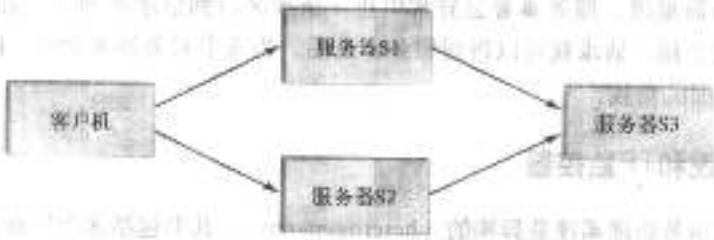


图22-7 当两个服务器访问代表同一个事务的同一个服务器时，必须保持事务的上下文

22.2.3 队列事务处理

图22-3中所给出的两层模型称为以**数据为中心的**（data centered），而图22-5中给出的三层模型称为是以**服务为中心的**（service centered）。在这两种情况下，客户机和服务器都参与**直接事务处理**（direct transaction processing），即客户机调用服务器，然后等待结果。系统会尽

快提供服务，并返回结果。此时客户机和服务器是同步工作的。

通过使用**队列事务处理**（queued transaction processing）机制，客户机把请求插入到服务器上待服务的请求队列中，然后做其他工作。当服务器准备提供服务时，就从队列中删除请求。例如，表示服务器的请求经常被插入到应用服务器队列的前面，直到指定应用服务器线程来处理这个请求为止。在服务完成后，服务器就将结果插入到结果队列中。客户机会将结果从队列中删除。因而，客户机和服务器异步工作。

使用可恢复队列，队列操作是事务型的。处理一个请求涉及三个事务，如图22-8所示。在得到服务前，客户机通过执行事务T_1将请求插入到队列中。提供服务时，服务器执行事务T_2从队列中删除请求，并把结果插入到应答队列中。最后，客户机执行事务T_3从应答队列中将结果删除。比较图22-8与图21-7，其主要差别在于图21-7用前向代理从请求队列中删除请求并调用服务器（被动地），而在图22-8中，由服务器(主动地)来删除请求。

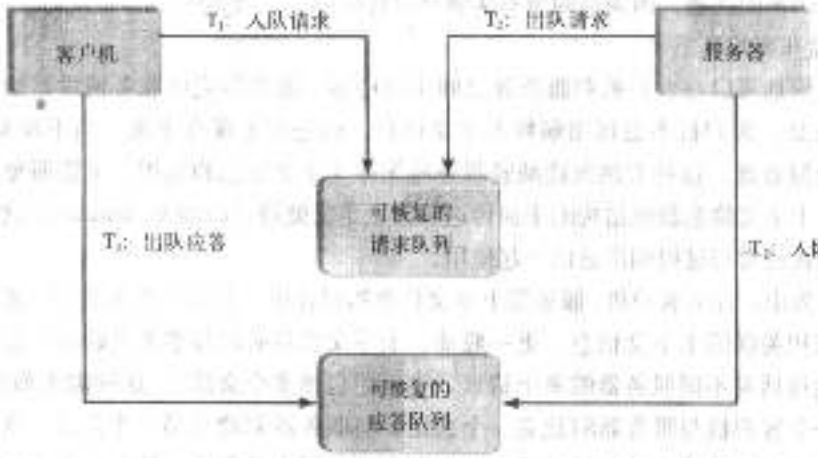


图22-8　排队事务的处理涉及二个队列和三个事务

队列事务处理带来一些好处。客户机能在服务器忙或不工作时输入请求。类似地，即使在客户机因为忙或不工作而没有准备好接收结果，服务器也能将结果返回。而且，如果请求正在服务时服务器崩溃，服务事务会异常中止，请求又回到请求队列中。在服务器重启后，不需要客户机的干预，请求就可以得到服务。最后，当多个服务器可用时，队列会调用算法平衡所有服务器间的负载。

22.3　异构系统和TP监控器

现在，许多事务处理系统是**异构的**（heterogeneous），其中包括多个厂商的产品：硬件平台、操作系统、数据库管理器以及通信协议。和异构相对应，早期的**同构**（homogeneous）系统的软硬件来自一个厂商，通常使用其**专用接口**（proprietary interface），且只与自己的软硬件产品相互连接，即使发布其专用接口，这些专用接口与系统中其他厂商的产品进行集成时也有困难，因为这些产品使用不同的接口。

同构系统逐渐发展成为现在的异构系统有以下几方面的原因：

- 新的应用经常要求与原来独立操作的老系统和遗留系统互相连接。例如，近几年来，公

司中每个部门可能都有自己的采购系统，现在公司需要与各部门相连接的全公司范围的采购系统。

- 有很多厂商供应硬件和软件组件，用户需要集成功能最好的软硬件，而与厂商无关。

异构系统需要各厂商同意实现**开放接口**（open interface），即非专有接口，使各软件能互相通信的通信软件。如果遗留系统（比如一个老的采购系统）使用非标准的、专有的接口，在接口之间就必须有一个**包装程序**（wrapper program）作为接口之间的桥梁。

为促进开放的分布式系统的发展，特别是事务处理系统的发展，现在开发出多种被称为中间件的软件产品。**中间件**（middleware）是软件，它支持C/S之间的交互。例如，满足下列条件的软件模块是中间件：

- 它支持不同的通信协议。
- 分布式应用的安全，包括认证和加密。
- 将一个模块的程序和数据翻译到另一个模块中去。
- 分布式事务的原子性、隔离性和持久性。

因为有标准的接口，所以这些软件模块可以在不同的分布式应用中使用。

JDBC和ODBC（10.5节和10.6节）是两个中间件的例子，它允许应用程序与不同厂商的数据库服务器交互。CORBA（16.6节）是另一种中间件的例子，它允许应用访问分布在网络上的对象。

22.3.1 事务管理器

分布式事务实现的主要问题是全局原子性。特定服务器上的原子性可以由这台服务器来实现，但全局原子性需要涉及该事务的所有服务器的合作，合作可以通过使用协议来实现。

当事务处理系统是同构系统时，实现全局原子性的算法通常集成到各厂商提供的服务器上。这样的系统有时也被称作TP轻负荷（TP-Lite）。在异构系统中，这样的算法通常包含在不同的中间件模块上，如图22-9所示。这种中间件模块称作**事务管理器**（transaction manager），它是TP监控器的一部分（参见22.3.2节）。图中表示的是一个两层系统，也可以表示为三层系统。异构系统比同构系统更加复杂，它常被称作TP重负荷（TP-Heavy）。

在这两种情形下，支持全局原子性的模块响应来自应用程序的命令，该应用程序设置分布式事务的边界，对子事务的提交进行协调。为达到上述目的，该应用程序必须知道何时事务是作为一个整体，何时开始执行它的子事务。这里，我们就来描述工作在TP-Heavy系统中的过程，因为系统是异构的，所以它依赖于接口的标准。X/Open标准API就是这样一个标准。

X/Open是由X/Open有限公司定义的，该公司是一个独立的被许多大型的信息系统供应商和软件公司支持的世界范围的组织。其API的一部分就是tx接口，它支持事务的概念，包括tx_begin()、tx_commit()和tx_rollback()。这些过程从应用程序中调用，在事务管理器中执行。在描述与分布式事务相关的概念时，我们的讨论基于这一标准。

当应用程序想让事务管理器知道它开始一个分布式事务时，它就调用tx_begin()过程。事务管理器记录新的分布式事务的存在，并返回一个标识符，该标识符唯一地标识该事务。后来，当应用向资源管理器发出服务请求时，通信层（由TP监控器提供）就通知事务管理器，有新的子事务存在。

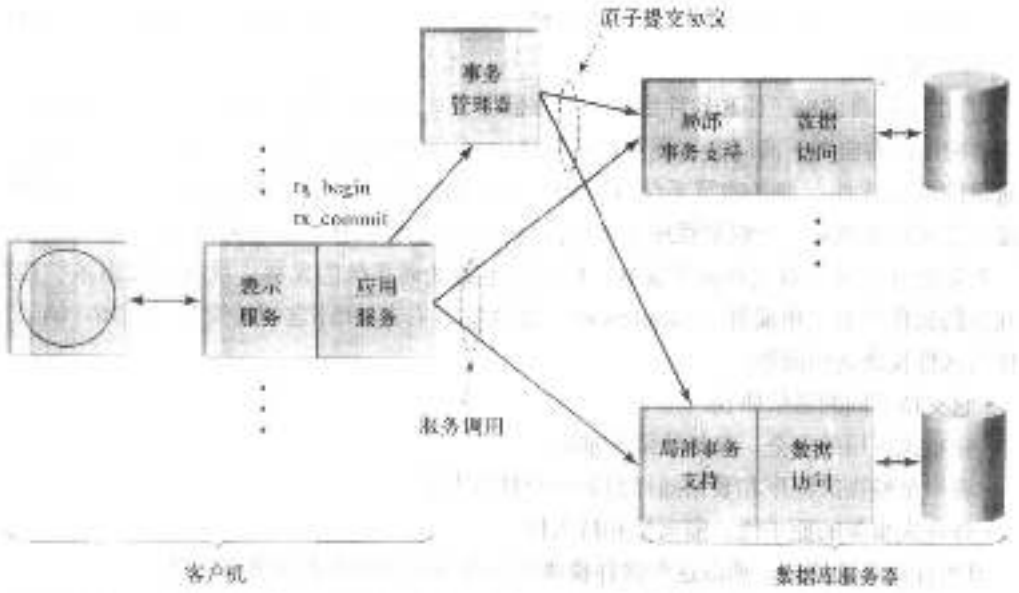


图22-9　事务能访问多个数据库服务器的两层多数据库事务处理系统

应用通过调用tx_commit()来通知事务管理器分布式事务已成功执行。然后事务管理器与服务器通信，参与事务执行的子事务要么全部提交，要么全部异常中止，从而保证分布式事务的全局原子性。为实现这一目的，使用**原子提交协议**（atomic commit protocol)控制消息的交换。在这里，事务管理器通常称为**协调器**（coordinator)。在26.2节，我们将讨论最常用的原子提交协议。

每个这样的过程是一个函数，它返回一个值来表示请求是失败还是成功。特别地，tx_commit()的返回值表示分布式事务是提交还是异常中止，以及异常中止的原因。例如，如果事务管理器发现其中一台数据库服务器崩溃或到该服务器上的通信被中断，分布式事务可能会异常中止。

22.3.2　TP监控器

事务管理器是能用来构建TP-Heavy系统的现成的中间件的例子，TP监控器是这样组件的集合，存在多个TP监控器。Tuxedo和Encina是早期的例子，Microsoft Transaction Server（MTS）是比较新型的TP监控器，Java Transaction Server（JTS）是基于TP监控器的一个规格说明。所有这些产品都包括或定义一个事务管理器和一个独立于应用的服务集，尽管在事务处理系统中这些组件是必需的，但通常不由操作系统提供。如图22-10所示，TP监控器可以视为操作系统和应用例程之间的一个软件层。

TP监控器提供的服务类似于操作系统所提供的一些服务。操作系统创建并发执行过程的抽象，这些过程可以相互通信，操作系统为这些过程提供对计算机系统资源的共享访问。TP监控器基于这些抽象创建事务的抽象，这些事务并发地在异构的分布式环境下执行。

下面这些服务通常是由TP监控器来提供的。

1. 通信

TP监控器支持应用程序模块使用的抽象来相互通信，这些抽象是通过在内核传递的分布

式消息提供的消息传递设施传递消息建立起来的（参见附录A）。通常提供两种抽象：远程过程调用和对等通信。远程过程调用由于它的简单性，以及它与C/S模型的自然关系，经常被应用程序使用。对等通信更加复杂，但更加灵活。系统级模块倾向于使用对等通信，以便使用它的灵活性来优化性能。大多数与数据库管理系统的通信是通过对等连接来实现的。我们将在22.4节探讨通信问题。

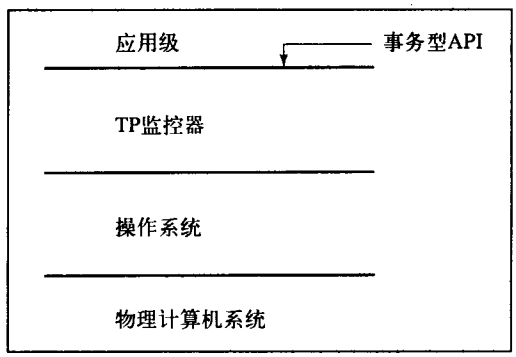


图22-10 事务处理系统的层次结构

2. 全局原子性和隔离性

TP监控器的一个目标就是确保全局原子性和隔离性。分布式事务的子事务可以在不同的资源管理器上执行，经常是数据库管理器本地实现原子性和隔离性，其他资源管理器可能不提供对这些特性的支持。例如，事务可以访问文件服务器上的几个文件，对文件的访问应该是满足隔离性和原子性的。如果文件服务器不支持隔离性，事务可以使用由TP监控器提供的锁管理器。锁管理器中的锁和文件服务器中的文件相关联。访问文件的事务在访问文件服务器之前，要求对访问的文件加锁（通过调用锁管理器），事务结束后，就调用锁管理器释放锁。如果所有事务都遵守这一协议，对文件的访问就可以同步化，这和访问提供并发控制的数据库服务器中的数据项一样（用锁实现隔离性将在23.5节中做更加详细的讨论）。类似地，如果文件服务器不支持原子性，可以使用日志管理器。

在将某个服务器上的局部隔离性和局部原子性扩展到全局隔离性和全局原子性时，事务管理器是很重要的。为提供这一服务，分布式事务所创建的子事务在启动时必须通知事务管理器。因此，事务管理器的作用与应用使用的通信抽象紧密相关。每当应用启动新的子事务时（通过第一次与一个服务器通信），就通知事务管理器。因此，支持全局原子性涉及事务管理器和通信设备。这是一个复杂的问题，我们将在22.4.1节分别讨论。

3. 负载平衡和路由

大型事务处理系统使用服务器类。如果类中的服务器分布在网络上，其可用性有所增加，因为它们能并发执行，所以性能也得到提升。当客户机调用这样的服务时，TP监控器能把请求路由给类中的任何一个服务器。有些TP监控器使用负载平衡作为选择标准。它们使用一个循环或随机算法来对服务器上的负载进行分配，或保留类中正在处理的每个服务器会话信息（也许是通过对等连接的数量来衡量的），从中选择负载最小的服务器。当提供队列事务处理时，负载平衡可以与队列机制集成在一起。

4. 可恢复队列

除支持队列事务处理外，可恢复队列通常用于应用模块之间的异步通信，许多TP监控器提供可恢复队列。

5. 安全服务

一个事务处理系统中的信息经常是需要保护的，加密、认证和授权是保护的基础。由于这样的原因，TP监控器经常支持安全服务。我们将第27章探讨这些服务。

6. 线程

我们知道，（例如，与应用服务器连接时）在事务处理系统中，线程能减少处理大量客户的开销。但不是所有的操作系统都支持线程。这种情况下，有些TP监控器提供自己的线程。这样，操作系统在调度过程执行时，就不必考虑多线程。在处理过程中，由TP监控器代码选择特定的线程来执行。

7. 服务器的支持

TP监控器提供多个服务器，这些服务器在事务处理系统中是有益的。例如，时钟服务器可让不同计算机上的时钟同步，文件服务器能像一般设备那样提供服务。

8. 嵌套事务

有些TP监控器支持嵌套事务。

本章中我们介绍了很多术语，它们是容易混淆的。下面我们对这些定义进行回顾。

- **事务服务器**　执行应用子例程，子例程完成基本单元的工作，这些基本的单元构成整个应用程序。
- **事务管理器**　支持分布式事务原子执行的资源管理器。
- **TP监控器**　提供事务管理器，处于中间件的下层，将事务处理系统的组件连在一起。
- **事务处理系统**　包括TP监控器、应用程序代码和各种资源管理器，如数据库管理系统，构成整个系统。

22.4　TP监控器：通信和全局原子性

在21.2.2节给出的常规分布式事务模型包括一组相关计算，这些计算是由许多模块完成的，各模块位于网络上的不同地点，相互之间可以进行通信。例如，应用模块可以从数据库服务器请求服务，或者这个应用本身是分布的，应用模块可能从其他模块请求服务。

考虑课程注册应用程序A_1（可能在某应用服务器上的事务的上下文中执行），它调用数据库服务器D_1上的存储过程为一名学生选修一门课程，然后调用远程应用程序A_2的一个过程，完成这名学生的缴费工作。A_2又可以调用处在不同数据库服务器D_2上的存储过程记录费用情况。D_1和A_2都被A_1看作是服务器（D_2被A_2看作是服务器）。但在这一意义上，只有D_1和D_2是资源管理器，它们所控制的资源状态必须保持。尽管这些模块之间的通信可以使用相同的通信范型（因此，在这一章，我们可以不区分这两种情况），但只有资源管理器参与原子提交协议。如果A_1在事务T的上下文中执行，我们说事务T从A_1**传播**（propagate）[⊖]到A_2，D_1[⊜]作为A_1调用的结果。

尽管消息的传递是基于通信完成的，TP监控器通常为应用提供几个高层通信抽象。它的选择取决于应用的类型。在一些应用中，各模块之间有严格的层次结构关系。例如，客户机模块向服务器模块请求服务，然后等待回答。在这种情况下，利用模块调用机制非常方便。在其他应用中，相互通信的模块是对等的，所以使用对等的通信模式是比较合适的。在这一

⊖ 有时也用术语spread或infect。

⊜ 有些TP监控器允许违反这一规则。例如，A_1可以定义A_2不包含在A中。类似地，A_2自身也可以定义它自己不包含在T中。而且，已定义的命令允许一个程序暂时不包含在当前事务中，而在以后恢复，参与事务的执行。

节中，我们来探讨通信的抽象及其实现。另外，我们还将对事件通信进行讨论，它对处理异常情况是非常有用的。

22.4.1 远程过程调用

分布式消息传输内核提供传输工具，利用这种网络消息传递功能可以支持模块之间的通信（参见附录A）。但由于模块接口的缺陷使得它使用起来不够方便。而调用操作系统原语来实现消息的收发既不可取也不是件容易的事。用户喜欢使用高级语言的过程接口，能从编译器提供的自动类型检查中受益。因此，我们的目标就是创建过程调用工具，使得调用模块（可能是远程的）中的一个过程和调用一个本地过程（该过程链接在调用者的代码中）一样方便。这样的一个工具支持**远程过程调用**（Remote Procedure Call，RPC）[Birrell and Nelson 1984,1990]。

使用RPC，分布式计算就可以采用树型结构，因为一个模块中的过程调用可以调用另一个模块中的过程。启动计算的过程作为一个整体被称为树的**根**（root）。

当一个过程在模块中被调用时，局部变量位于一个临时工作区，完成计算需要的所有必要信息作为参数传给过程。在计算完成和过程返回后，局部变量的空间被释放，因此不能用来存储上下文信息。由于这个原因，RPC通信称为**无状态的**（stateless）。如果经过几次调用后，上下文信息必须保存，调用模块就将它作为全局变量保存，或者使用一个上下文处理机制来传递它（参见22.2.2节）。

1. 远程过程调用的实现

RPC是使用桩（stub）来实现的。客户桩是与客户机相链接的例程，服务器桩是与服务器相链接的例程。如图22-11所示，桩程序作为客户机和服务器的中介。客户机调用服务器过程时，服务器过程使用过程的全局唯一的名字。名字是系统中所有用户都知道的字符串。但调用并不直接激活过程。相反，它调用处在服务器上的客户桩（使用目录服务，我们将在后面讨论），建立连接，然后把参数转换成一种标准格式[⊖]，连同被调用过程的名字一起打包成调用消息，这称作**参数序列化**（marshaling of argument）过程，然后使用低层通信协议把消息发送给服务器。

在图中，服务器处在不同的计算机上。服务器桩从所有的客户接收调用消息，并以一种特殊的消息格式对参数重新处理，转换成服务器所希望的格式，然后使用标准的过程调用（本地）方法调用合适的过程。调用结果返回给服务器桩，由它使用消息传递工具传回给客户机桩，再由客户机桩返回给客户（客户机桩是由使用常规过程调用的客户应用程序来激活的）。因此，RPC是高层通信工具，是由过程调用机制耦合消息传递工具来实现的。

桩机制有很多优点。只有桩和操作系统接口。对客户机和服务器而言，就好像是通过通常的过程调用来实现相互之间的通信。而且，对服务器上的过程，桩使得本地机器上的过程调用和远程机器上的过程调用是一样的，就像调用链接在客户机代码的本地过程一样。因此，客户机的代码不必关心服务器所处的物理位置，它使用同样的机制与本地服务器和远程服务器通信。最后，如果组件移动，客户机代码与服务器代码是不需改变的，这一特性称为**位置**

⊖ 在另一种组织形式中，客户端并不进行转换，但如果有必要，就让服务器端将接收的参数转换成服务器所需要的形式。

透明性（location transparency）。

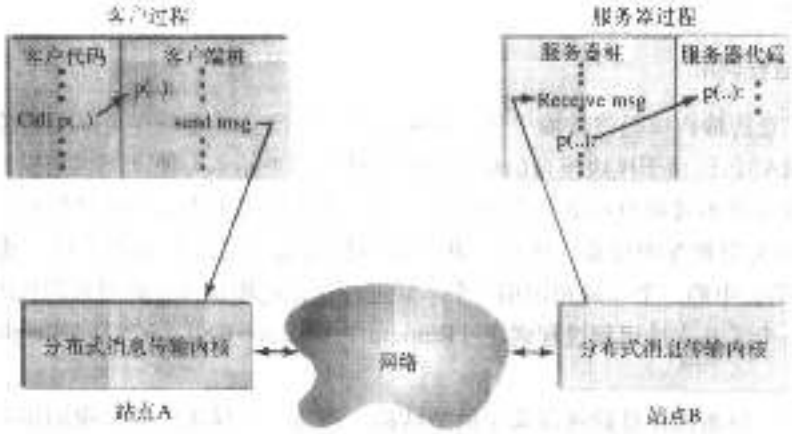


图22-11　用桩支持远程过程调用

在把客户机连接到服务器时，必须考虑桩可能失败的情况。一个地点的系统失败不经常出现，但通常会出现崩溃的情况。当系统崩溃时，主存（不是大容量的存储设备）中的内容丢失，系统必须重启。在恢复时，通常不去恢复系统在崩溃时正在处理的过程。

分布式系统的失败不仅包括特定计算机的崩溃，而且包括通信的失败。相比较而言，这样的失败可能是持久的（通信线路的长期故障），也有可能是暂时的（一个消息丢失，而后续消息可能正确传输）。系统中有成百上千或成千上万的计算机，通信线路、路由器等等。在任何一个特定的时间，所有设备都正常运行的概率远比某一个地点发生崩溃的概率小。这就是为什么在分布式系统中经常出现故障的原因。

不可能所有的部分都出现故障，系统的大部分仍会正常工作，应用可能要求在某些组件出现故障时仍然可用。与单个地点的系统相比，在分布式系统中，如果一个故障干扰到客户机与服务器的交互，但它还有可能继续提供服务。例如，在向服务器桩发送服务器调用参数后，客户机桩必须等待一个应答消息。如果在规定的时间里没有收到应答消息，就可能出现了如下情况：

- 来自客户机的调用消息还在传输过程中。
- 来自客户机的调用信息丢失。
- 服务器崩溃。
- 服务器已开始服务，但还没有完成。
- 服务器已经完成服务，但来自服务器的应答消息还在传输过程中。
- 服务器已经完成服务，但来自服务器来的应答消息丢失。

基于以上情况，客户端的桩程序采取以下措施中的一种：

- 继续等待。
- 发送另一个调用消息给服务器。
- 发送一个调用消息给处在不同机器上的服务器，以便完成同样的功能。
- 异常中止客户机。
- 返回一个错误指示信息给客户机。

但是，客户机桩不知道引起延迟的原因，当用适于另一种情形的应对措施解决某个问题时，可能会把情况弄糟。例如，客户机可能会不明原因地等待，或者提前中止，也有可能因应答消息丢失而再次发送一个调用消息，这导致服务器执行两次相同的服务。例如，若客户机只是简单地读服务器中的一些信息，上面所说的情况还是可以接受的。而在另一些情况下却是不可接受的，例如，客户机请求购买机票。由于以上这些原因，处理失败的桩算法可能相当复杂。

2. 目录服务

服务器通过接口知道它的客户机。接口是公开的，它包含客户机用来调用服务器上过程的必要信息：过程名和参数描述。正如我们在16.4.1和16.6节所看到的，**接口定义语言**（Interface Definition Language，IDL）是一种描述接口的高级语言。IDL编译器将接口描述信息编译到一个头文件和特定的客户机与服务器桩程序中。进行编译时，头文件包含在应用程序中，客户机桩链接到结果模块的客户代码中。使用IDL文件来描述服务器接口促进了开放式系统的开发。客户机和服务器可以由不同的厂商来制造，只要它们在通信接口上达成一致即可。

但服务器接口并不指定服务器过程的Id和位置，服务器过程实际执行服务器代码。因此在编译时并不知道它的位置，或者说它是动态改变的。而且，服务器以具有实例的类的形式实现，它们在网络上的不同节点执行。因此，系统必须提供运行机制来对客户机和服务器进行动态绑定。

绑定要求，一台服务器S在准备好为它的客户提供服务时，利用某些命名服务来注册自己，使它对所有客户可用。这样的服务通常由**名字**（name）、**目录**（directory）和**服务器**（server）提供。

为了注册，S将它的名字（全局唯一）、网址、它所支持的接口及用来与客户机交换消息（用来实现RPC通信的消息）的通信协议提供给目录服务器。例如，S可能只通过TCP连接来接受服务请求。相对于其他服务器（目录服务器）而言，这是服务器充当客户机的例子。

客户机桩向目录服务器提交服务器的名字，或者是一个接口的Id，向目录服务器请求网络地址和通信协议，用来与有这样名字或支持这种接口的服务器通信。一旦提供这些信息，客户机桩就能与服务器直接通信。

尽管目录服务器自身也是一台服务器，但它必须有特定的状态，因为客户机和服务器必须能与它相连，而不必去使用另外的目录服务器（避免出现先有鸡与还是先有蛋的问题）。因此，它应位于一个众所周知的网络地址上，不用使用目录服务器就能确定该地址。

目录服务器可能是一个复杂的实体。复杂的一个原因就是它在分布式计算中扮演中心角色。如果目录服务器失败（可能是因为它驻留的主机崩溃），客户机就不能定位服务器，新的分布式计算就无法建立。为避免这样的灾难，目录服务自身必须是分布式的或者在网络上是重复存储的。但这又引起新的问题：必须确保备份是最新的，网络上任意客户机都能找到包含它所需信息的服务器。

分布式计算环境（Distributed Computing Environment，DCE）[Rosenberry et al. 1992]是一个中间件的例子，中间件支持分布式系统如TP监控器[⊖]。DCE所提供的服务是运行在不同操作系统平台上各模块之间的远程过程调用。和其他基本特征一样，如我们将要在第27章讨

⊖ TP监控器是基于DCE的。

论的安全服务，目录服务是在连接的过程中提供的。

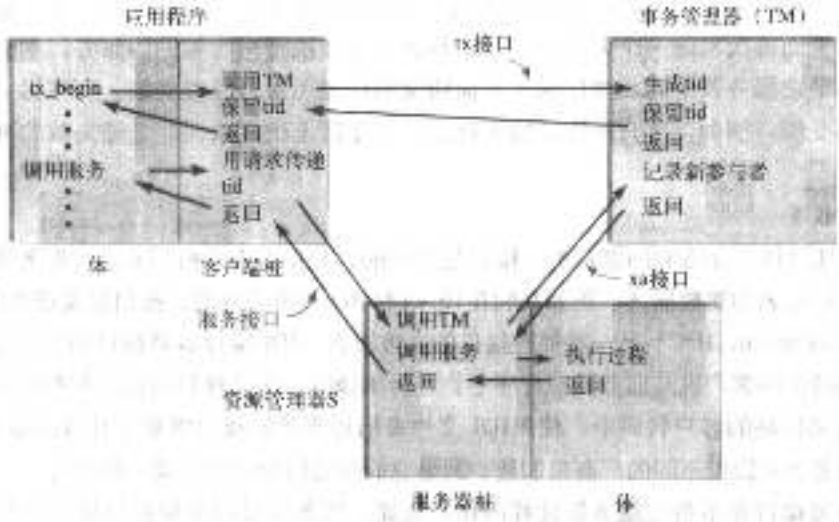


图22-12　通过与事务管理器通信实现事务的远程过程调用

3. 事务管理器和事务的RPC

在实现全局原子性时，事务管理器的角色如图22-12和图22-13所示。在启动和中止事务的事务管理器（TM）内，X/Open系统调用tx_begin()、tx_commit()和tx_rollback()来调用过程。当应用启动一个事务T时，它调用tx_begin()。如图22-12所示，TM返回一个事务标识符tid，它唯一地标识T。事务标识符保留在客户机的桩数据空间里。

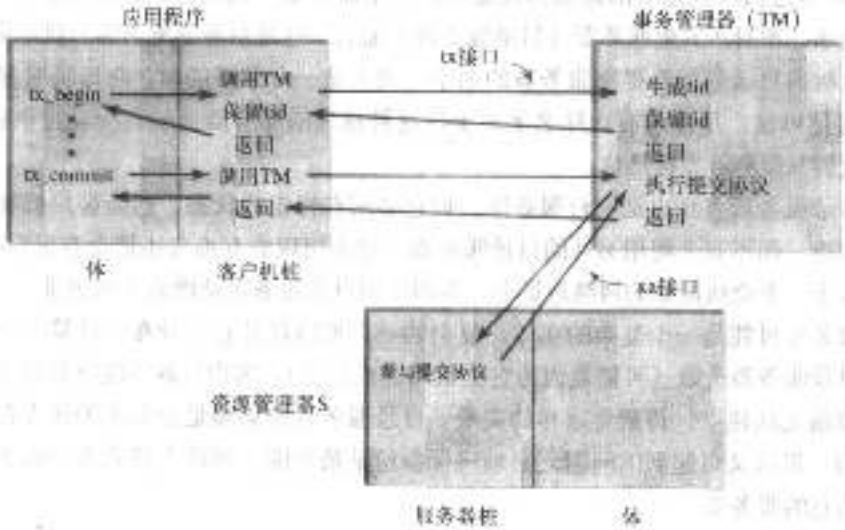


图22-13　与事务管理器通信来实现原子提交协议

随后，一旦有客户机调用资源管理器S，客户机桩就把tid附加到调用消息中，供S来识别调用的事务。如果这是T向S发出的第一个请求，服务器桩就通知TM。现在通过调用TM的过程xa_rcg()并传递tid为T提供服务。因此，TM能记录所有参与T服务的资源管理器标识。S称

作T的一个**队**（cohort）。xa_reg()是X/Open的，处在事务管理器和资源管理器之间的xa接口的一部分。当事务管理器和资源管理器都支持xa接口时，它们能共同支持分布式事务；即使它们是不同厂商的产品。

注意，定义tx接口和xa接口的标准，tx接口是应用和事务管理器之间的接口，xa接口是事务管理器和资源管理器间的接口，该标准没有定义应用和资源管理器之间的接口，这样的接口是由资源管理器自身来定义的。

因为过程调用是一种同步通信机制（调用者一直等到被调用者执行完毕为止），当子事务完成计算时，它所调用的所有子事务也执行完毕。因此，当事务树的根完成计算时，整个分布式事务也执行完毕。根可以要求，事务通过调用事务管理过程tx_comit()来提交，如图22-13所示（如果客户机希望异常中止事务T，它就调用tx_rollback()）。事务管理器必须确保事务的终止是全局原子性的，即被事务T调用的所有服务器要么全部提交要么全部异常中止。事务管理器是通过参与原子提交协议来实现上述要求的，它在其中充当协调员，而资源管理器作为支持者。被广泛使用的原子提交协议是*两段提交协议*，该协议将在26.2节描述。

为执行提交协议，事务管理器必须与所有的参与者（如S）通信，如图22-13所示。这是通过使用资源管理器所登记的回执来实现的。除资源管理器提供给应用的标准服务接口外，它还给事务管理器提供一组**回调**（callback），它们是资源管理器的过程，可被事务管理器用于不同的目的。每个资源管理器提供不同的回调，这取决于它所能提供的服务。为此，那些参与原子提交协议的事务管理器就注册回调[⊖]。如果支持者不注册这样的回调，就不能参与协议，全局原子性也就不能得到保证。

因此，我们刚才所讨论的过程调用机制（涉及使用tid和xa接口）是RPC机制的一个升级，是用来实现全局原子性的一个组成部分。它由TP管理器来提供，称作**事务远程过程调用**（Transactional Remote Procedure Call，TRPC）[⊜]。

全局原子性由桩和事务管理器实现。它涉及以下方面：

- T启动时，由事务管理器为事务T建立一个标识。
- T的标识作为参数包含在过程调用消息中。
- 只要调用新的资源管理器，就通知事务管理器。
- 事务完成时执行原子提交协议（参见26.2节）。

为获得全局原子性（必须由TRPC来处理），必须克服由网络中的失败引起的另一个障碍。在22.4.1节，我们看到一些失败的情形，这些失败阻止客户桩接收它向服务器S所发出请求消息的应答消息，但客户桩是无法区分这些失败的。桩所采取的动作会产生不同的结果。特别是当调用消息被重发时，服务将被执行两次，即使调用消息没有重发，桩也不能认为服务没有完成。

在这种情况下，桩的动作必须确保被请求的服务在S上执行一次，允许事务继续，或确保所请求的服务根本没有做（假设其他服务器不能提供这样的服务），事务被异常中止。因此，桩必须确信什么叫精确执行一次的语义。

在执行所请求的服务时，S可能要调用其他服务器S'，这种情况就会更加复杂。如果S所在的计算机发生崩溃，S'在网络的其他地方执行，这就出现一个**孤儿**（orphan），即S'没有报

⊖ 支持分布式存储点的回调为另一个实例。

⊜ TRPC以有效RPC增强形式的执行将在27.6节讨论。

告它执行结果的父进程（即S）。在这种情况下，桩的工作就非常重要，必须保证精确执行一次的语义。

4. 多个域的分布式事务

这里，我们假设整个分布式事务的提交协议是由事务管理器来控制的。事实上，事务管理器的职责仅局限于一个域（domain），这个域通常包含事务管理器执行的地点的所有资源管理器（尽管可能还有其他的组织）。

假设域D_A的分布式事务的一个成员（可能是应用模块，也可能是资源管理器）是由事务管理器TM_A控制的，该成员调用域D_B的资源管理器R_B，R_B是由事务管理器TM_B控制的。以后，TM_B将作为R_B及所有在D_B与R_B相关事务的协调员来参加原子提交协议。对TM_A而言，TM_B是作为所有这些事务的代表，也是TM_A的支持者。因此，TM_A必须被告知，TM_B是它的一个组成成员。对R_B和TM_A而言，TM_B也必须知道它的这种双重角色。在分布式事务的X/Open模型中，由一个称为**通信资源管理器**（communication resource manager）的服务器来负责传播这类信息。域中事务管理器和通信资源管理器之间的接口称作xa+接口。

因此，分布式事务的结构就像一棵树，如图22-14所示。资源管理器作为叶子节点，事务管理器作为内节点（注意，应用模块不是树的一部分，因为只有资源管理器需要参与原子提交协议，以确定是否提交对它所管理的资源的修改，这些资源是由它们来管理的）。树的根是事务管理器，它控制分布式事务启动的域。设置事务边界的应用程序发送提交请求给事务管理器，由事务管理器执行协议。图中显示仅有两个分支的树，一般来说，分布式事务树的分支数是任意的。沿着树边缘的实际消息交换在26.2.1节描述。

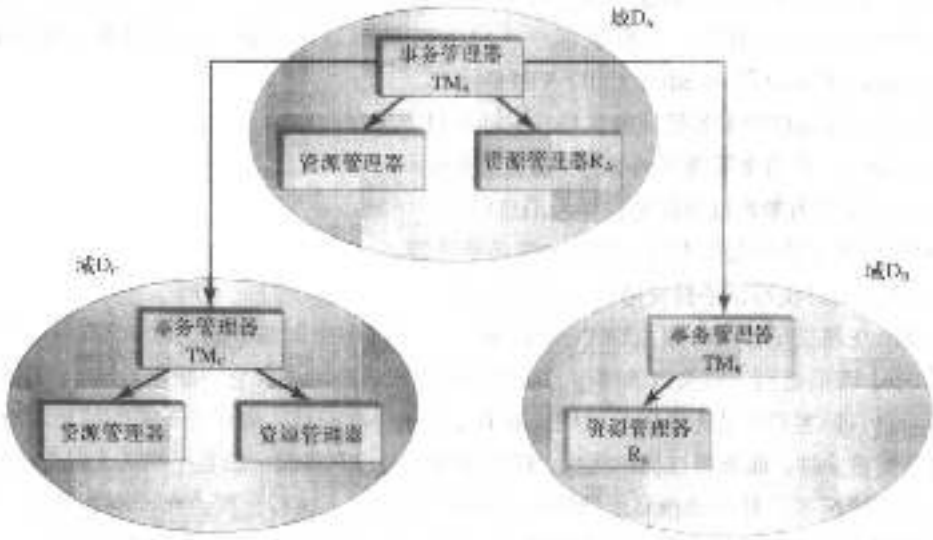


图22-14　分布式事务的树结构

22.4.2　对等通信

在远程过程调用中，客户机通过发送请求消息给服务器来激活一个过程，然后等待应答，因此通信是同步的，于事务在服务器上执行，当服务完成后，就发送一个应答消息，这种请

求/应答的通信模式是不对称的，客户机和服务器不是并发执行的。

相反，**对等通信**（peer-to-peer communication）是对称的：一旦建立连接，双方使用名字发送和接收命令。这种方式会更加灵活，因为通信双方支持各种消息模式。发送是异步操作，这意味着系统要对所发消息进行缓冲，然后把控制权交给发送者。和RPC不同，发送方和接收方能并发执行。发送方可能要完成一些计算，或者发送额外的信息，因此在接收方应答之前，一个消息流可能从发送方流到接收方。和RPC一样，对等通信是由低级消息传输协议（如TCP/IP或者SNA）支持的。

同样，灵活性是以增加复杂性为代价的。通信双方都必须能够处理对方使用的消息模式，这意味着有更多出错的可能。例如，每一个进程在继续执行之前，期望从另一个进程得到消息，这就会导致死锁发生。TP监控器支持对等通信不仅是因为它的灵活性更高，而是因为许多数据库服务器（经常运行在主机上）都使用这种通信模式。

1. *建立连接*

目前有多种对等通信协议。我们在这一节的讨论基于IBM常用的SNA协议LU6.2[IBM 1991]和与它接口的API。

在开始对话之前，模块必须和它所要通信的程序建立连接。这是通过一个以目标程序名字作为参数的allocate()命令来完成的。通过创建一个新的程序实例来结束对话。和RPC相反，位置透明性不是它的目标，请求通信方会明确提供接收程序的地址。

我们再回到22.4节分布式计算的例子，A_1可能与A_2建立对等通信连接。启动新的缴费程序的实例来接收由A_1发来的消息。

在LU6.2中的连接是半双工的，连接通常是半双工或是全双工。利用**半双工**（half duplex），消息只能从通信双方的一方流到另一方，在任意给定的时间里，模块A是发送方（当前连接对A来说是**发送模式**），另一个模块B是接收方（对B来说是**接收模式**）。A发送任意多的消息给B，然后传递发送许可给B。这时，B为发送方而A为接收方。当B完成消息发送后，它又把发送许可权返回给A，这样的过程重复继续下去。这是和一个**全双工**（full duplex）连接相反的。在全双工通信中，任意时间里双方都能发送。在半双工通信中，当前连接的方向只是它的上下文的一部分。

当模块A中的例程作为事务的一部分执行时，它与模块B中的一个程序实例建立连接，事务从A传播到B。一旦连接建立，A与B是对等的双方，流过连接的所有消息都从属于那个事务。客户机/服务器用来通过连接通信的上下文信息可以方便地保存在局部变量中，因此，每个到来的消息就能按照它的上下文信息进行解释。因此，使用对等交互的协议经常是**有状态的**（stateful）。

参与任意多个不同的对等连接的模块是并发的。因为事务中的任意一个模块都能与另一个模块建立新的连接，分布式事务的常见结构是典型的非循环图。如图22-15所示，其中节点代表参与通信的事务，边代表连接。图是非循环的，因为当建立连接时，就创建一个新的

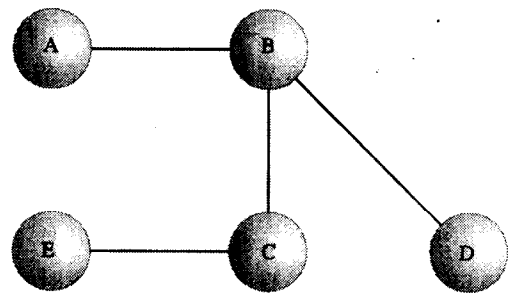


图22-15 通过对等连接通信的模块集

程序实例。因此，图中的节点代表实例，为一个程序和已有的实例建立一个连接是不可能的。如图中的C和D，它们代表同一程序的两个不同实例。图中没有哪个节点的作用类似于一个过程调用层次的根节点。

2. 分布式提交

由于使用对等连接，提交的规则是和事务中参与通信方地位平等的规则一致的，任意一个结点都能请求提交事务。在分布式事务的层次模型中，只有根模块能请求提交，因此（因为通信是同步的）所有子事务在那一时刻必须完成。与之相反，对等事务不是同步的，当其中一个事务决定提交时，与它对等的事务可能还在执行，并不一定准备提交。因此，作为整体来提交的事务确保所有的参与者都已执行完毕。这样，系统执行一个原子提交协议，它保证所有参与者要么全部提交，要么中止。以下的描述是基于[Maslak et al. 1991]的。

在一个组织合理的事务中，一个模块A开始提交。此时A的所有连接都必须处在发送模式。A通过声明一个**同步点**（syncpoint）来开始提交，这使得一个**同步点消息**（syncpoint message）发送给A的所有连接，然后A等待，直到提交协议完成为止。

当收到同步点消息时，连接到A的B可能还没有完成它的那部分事务。当B执行完成后，它的所有连接（除与A的连接外）都处在发送模式，它也定义一个同步点，发送给它的所有连接（除与A的连接外）。这样，同步点消息传遍整个事务图。通过它仅有的一个连接(如图中的E)收到一个同步点消息的节点也可以定义一个同步点，但没有额外的同步点消息。最终，所有的对等方都定义一个同步点，通过它们的同步点定义实现同步，现在可以实施原子提交协议。

同步点协议要求只有一个模块启动它，只有当它所有的连接都处在发送模式，并且没有收到任何同步点消息，或者除一个连接之外，所有与它的连接都处在发送模式状态，它处在接收模式状态，通过连接收到一个同步点消息，一个模块才能定义一个同步点。如果协议不能正确执行，事务就异常中止。例如，如果两个参与方都通过定义一个同步点来启动协议，通过连接，同步点消息会在某些中间节点会聚，这违反协议，导致事务异常中止。

全局原子性可以通过使用类似于事务管理器的模块来实现，这种模块称为**同步点管理器**（syncpoint manager）。当一个程序接受对等连接时，同步点管理器与程序A的新实例相关联。同步点管理器还有相关的域。这和RPC使用的技术相类似，每当A首次调用域中的一个资源管理器时，A的同步点管理器就被告知事务中加入新成员。类似地，如果A与B建立一个连接，A的同步点管理器就被告知，B的同步点管理器加入到事务中。同步点管理器必须对域中事务的所有参与者的原子提交协议进行管理（这里，A是作为参与者），这和图22-14中所表示的一样。树的根是参与者的同步点管理器，初始同步点是由该参与者来定义的。

当同步消息到达一个叶子节点时，若处在该叶子节点的子事务已经执行完，一个应答消息就返回给树的根节点。当所有叶子节点都回复后，根节点就认为整个事务已经执行完毕。这个协议的详细描述在26.2.4节给出。

22.4.3 事务中异常情况的处理

当模块使用RPC通信时，被调用者（或服务器）组织起来接收服务请求。它导出一个过程用于对请求进行服务。当没有请求时，服务器空闲，等待下一个请求。这和对等通信有点类似。在这种情况下，服务器接收一个连接，执行一个接收命令。当没有服务请求时，服务

器处于空闲状态。

在设计为另一模块的请求服务的模块时，过程通信和对等通信都是有用的。但在模块还必须处理异常情况时，这些通信模式通常是不合适的，因为模块不能只是等待请求的发生，它还必须执行其他任务。

例如，假设模块M_1重复地读取和记录一个炉子的温度。模块M_2对炉子的使用过程进行控制（可能在另一台计算机上执行），当温度达到某特殊的限定值时，M_2就被告知这一异常情况（可能M_2将炉子关掉）。因为M_2是作为控制者，它需要一种方法来确定何时发生异常情况。实现这种要求的一种办法就是，M_2定期使用过程化通信访问M_1，要求返回当前的温度记录。对M_2来说，另一种方法就是建立一个对等连接，定期向M_1发送温度请求消息。这些方法称为**轮询法**（polling）。如果值不可能达到限定值，那么使用轮询法是一种资源的浪费，因为M_2不停地发送请求，但应答却指示不用采取任何措施。而当温度达到这一极限值时，M_2必须快速反应，轮询法就显得更加浪费，因为这时需要更加频繁的通信。

作为第二个例子，考虑一个POS机系统，其中应用模块M在销售终端上，由它来对客户从键盘上输入的服务请求做出响应。如果系统准备暂时离线，那么不能服务的消息必须在终端显示出来。但如果M只对键盘上的输入做出反应，而不能识别来自中心站点的通信信息，它就不会被通知打印离线的消息。可见，对异常情况做出响应的机制是必需的。

在以上的例子中，使用标准的客户机/服务器范式来设计M，使之对来自中心地点的请求（还有来自键盘的请求）做出反应是可能的。但如果要求对中心地点做出响应是受时间限制的，那么仅用这种方式可能还不够充分。如果要求的响应时间比分配给键盘请求服务的时间还少，那么采用中断机制是必要的。

异常情况称作**事件**（event）。一些TP监控器提供**事件通信**（event communication）机制，利用它模块就能识别事件，并能方便地通知另一个模块M对事件进行处理。通知可能采用中断的形式，促使M执行事件处理例程[⊖]。如果事件识别发生在一个事务的上下文中，事件处理器的执行就变成事务的一部分。这样，事件处理（像RPC和对等通信一样）使得一个分布式事务从一个模块扩展到另一个模块[⊜]。

可以中断的模块使用TP监控器的事件处理API函数来注册**事件处理器**（event handler）（或回调），在通知事件发生时就开始执行事件处理器。如图22-16所示，M使用TP监控器注册事件处理器foo。因为M不等待事件的发生，而是继续进行它的正常工作，所以事件通知被称为是**主动提供的**（unsolicited）。

当要处理的事件发生时，系统打断M的执行。它保存M的当前状态，并将控制传递给foo。当foo退出时，系统利用已保存的状态将控制返回M被中断的点。当事件发生时，M可以立即响应。响应是异步的，因为无法事先确定事件何时发生，也不能确定在M执行（即两个指令之间）的哪个时间点foo会执行。虽然快速响应非常重要，但需要仔细设计，以确保处理器的执行不会干扰中断的计算。

如图22-16所描述的那样，事件生成模块N使用事件API来指示系统，事件已经发生，模块

⊖ M可能选择延迟执行程序处理事件，直到它明确通知TP监控器，它已准备就绪，能处理任何发送给它的事件为止。

⊜ 我们的讨论基于Tuxedo事务处理系统中的事件通信[Andrade et al. 1996]。

M被中断。这个过程常被称作**通知**（notication）。假设M已经注册事件处理器，TP监控器促使它执行这个事件处理器。API允许事件生成模块将消息作为一个参数传给它。

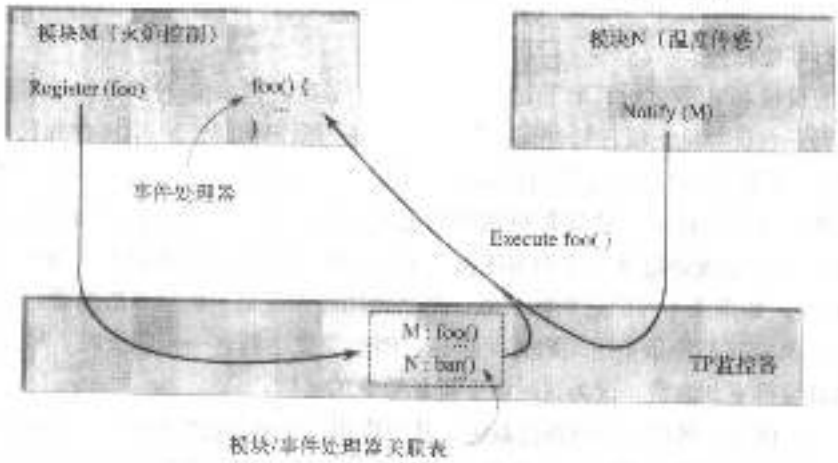


图22-16　将一个事件生成模块与一个事件处理模块相链接的事件处理过程

这种简单的事件通信形式是有用的，但它有一个重要的缺点：事件生成模块必须知道被通知模块的ID（可能不止一个模块）。应用中对事件做出响应的模块可能会变化。让目标事件处理模块对事件生成模块而言是透明的，因而是值得的，这可以通过使用事件代理来做到。

事件代理（event broker）是TP监控器提供的服务器。它是连接事件生成模块和事件处理模块的桥梁。使用事件代理时，给每个事件起一个名字。事件处理模块M在用TP监控器注册事件处理例程后，它通过使用事件API，调用事件代理来订阅（subscribe）事件E。事件的名字是作为参数来提供的。代理记录已命名事件和M之间的关联。一个事件可能有多个订阅方，这种情况下，它就与一个事件处理模块列表相关联。当事件发生时，事件生成模块N就将事件发送给代理（再次使用代理的API，把事件的名字作为参数来传递）。然后代理通知所有事先订阅事件的模块。这种情况如图22-17所示。尽管这幅图看起来和图22-16有点相似，但要注意，代理保留已命名事件和事件处理模块之间的关联，并使用通知来与事件处理模块通信。正如前面所述，事件处理模块和事件处理例程的关联保存在TP监控器中。

和其他形式的通信一样，通过事件通信可使事务从一个模块扩展到另一个模块。例如，在事件生成模块中，如果post（Event）动作是作为事务的一部分来完成的，结果处理器的执行也可作为那个事务的一部分。但在有些情况下，处理器的执行不应作为那个发送事件事务的一部分。例如，执行数据库操作的事务可能发现一个不可修正的数据库错误，它请求异常中止。在异常中止之前，事务可能想发送一个事件，把这个错误通知给数据库管理员。此时，事件处理器不应该是事务的一部分，否则，在事务异常中止的时候，它也被异常中止，数据库管理员就不会得到通知。控制事务何时扩展为一个事件发送结果的规则，是事务处理系统协议实现的一部分。

22.5　因特网上的事务处理

因特网的增长促进了许多异构分布式事务处理应用的发展。这些应用使用已存在的客户

软件（Web浏览器）和标准的因特网通信协议与Web服务器通信。Web服务器提供事务处理的功能。一个Web服务器有因特网地址（即，它在因特网上是可访问的），对使用标准因特网协议（例如HTTP、SMTP或FTP）的Web提供访问服务。

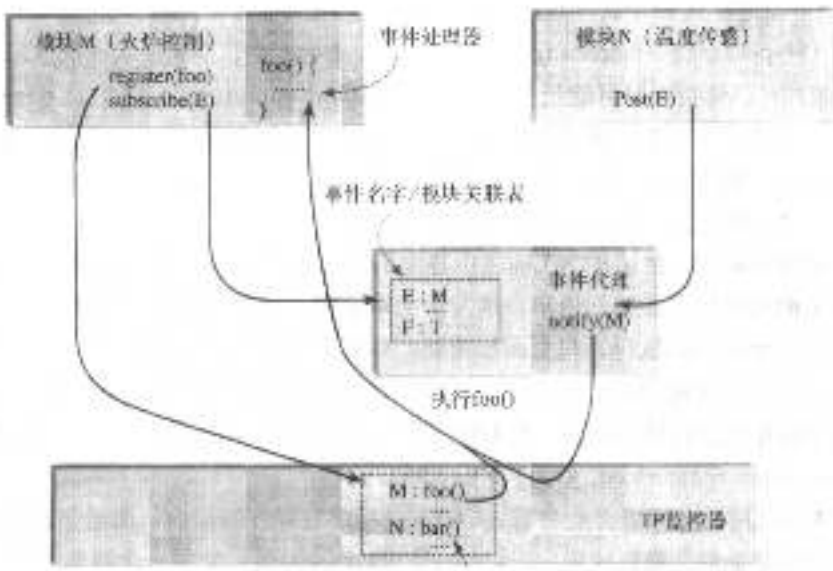


图22-17 使用事件代理来处理事件传递

利用这样的系统，通过Web服务器，一个用户可以启动事务来访问全世界的数据库，把它的浏览器作为表示服务器。对各种多媒体，只要点击鼠标就可访问，再点击几下鼠标就能找到提供某件商品的最便宜的厂商，再点击几下鼠标就能将商品买下来，所有这些都是用户在家中完成的。

另外，很多企业都使用因特网技术来构建企业内部网和企业外部网。

- 企业内部网　企业内部网是通过专用网络将企业内部各个站点互连而得到的，它脱离了因特网，但使用的是因特网技术，企业通过内部网来开展企业业务。在加利福尼亚见到的一个产品，可能是在纽约设计的，在香港制造的，在芝加哥上市，在新泽西集中进行账户结算。
- 企业外部网　企业厂商和业务客户使用它们的企业外部网进行连接。为满足位于得梅因的企业部门的要求，全世界的办公用品厂商都能来竞标。

对网络上的商务活动而言，事务起着关键作用。此时出现的一个重要问题就是安全性问题，因为通过因特网发送的信息容易被截取和改变，许多网站都发现有未经授权的用户闯入。为解决安全问题，大多数Web浏览器支持高级的基于加密的安全协议。各种应用通过使用防火墙和类似机制来增加安全性。我们将在第27章对因特网事务的安全问题进行讨论。

22.5.1 一般的体系结构

在描述因特网事务处理之前，我们简短地介绍普遍使用的体系结构，通过它，信息在

Web浏览器和Web服务器之间传递。希望与Web服务器交互的用户在浏览器中给出服务器的地址，这个地址以URL（Uniform Resource Locator，统一资源定位器）的形式给出，例如，http://www.somecompany.com。浏览器使用**超文本传输协议**（HyperText Transfer Protocol；HTTP）与服务器建立连接。服务器再次使用HTTP返回浏览器要显示的页面。页面是用**超文本标记语言**（HyperText Markup Language，HTML）来书写的，它定义页面的显示格式。

服务器使用HTML页面时可能包含一个或多个被称作**applet**的程序，它是用Java这样的语言来书写的。applets在浏览器提供的程序环境里执行，它们能激活HTML页面，对特定的事件（如鼠标单击）做出反应，能与用户和网络以其他方式交互，其中的一些方式我们将在后面进行讨论。发送页面后，服务器与浏览器断开连接。

浏览器收到页面后，使用HTML描述中规定的页面格式显示页面，也可能执行一些指定的applets。页面可能包含按钮、由用户填写的文本框和一些诸如此类的控件。用户与页的交互可能触发其他applets的执行。与页面相关联，但通常对用户来说是隐藏的，这就是Web服务器（可能不是发送页面的那个服务器）的URL，用户需要的信息发送给该服务器，服务器上程序的名字被用来进行信息处理。调用Web服务器上程序的两个最流行的API是**通用网关接口**（Common Gateway Interface，CGI）和servlets。

CGI*程序*可以用任意的语言来书写，CGI不是指编写程序的语言，而是指如何调用程序，从页面来的信息是如何传递到该程序。servlet是用Java编写的，它用一个特殊的Java API与客户和服务器上的其他程序通信。除语言之外，这两个API的主要差别是servlet API的功能更丰富。尤其是它提供对会话管理的支持，而CGI没有此功能。浏览器的每个请求调用一个CGI程序，而一个servlet能处理来自同一个客户的多个请求。尽管有这些差别，但这两个API还是很相似的，我们用CGI作为例子。

当用户点击“提交”按钮时，浏览器使用HTTP与指定的因特网服务器建立连接，并发送消息。CGI程序在服务器上启动，信息以参数的形式传给它。程序对这些信息进行处理，也可能执行其他动作（例如访问数据库，向一个事务处理系统提交请求）。这时，程序准备一个新的HTML页面并将页面送回浏览器。又一次，服务器与浏览器断开连接，并且不保留客户的上下文信息。在用户与新的HTML页交互后，单击“提交”按钮，页面重新与同一个服务器或者不同的服务器连接，并启动新的CGI程序。

在一些应用中，浏览器可能为了同一个用户与同一个Web服务器重新连接多次（例如，第一次从一个目录中读信息，然后定购）。由于这个原因，服务器可能希望访问这些交互上下文信息（例如，描述用户的信息），因此，它保留与用户的会话信息，这个用户与它有过多次的连接。在其他一些应用中，服务器可能希望保留几天内的连接上下文信息，甚至保留几周的信息（例如，以前订单中显示出的用户的喜好）。这些上下文信息通常是不保存在服务器上的，因为服务器可能同时处理成千上万的客户。在服务器上保存所有客户的上下文信息是浪费资源的，特别是一个浏览器可能不回复（或许用户无聊，把它的浏览器指向迪斯尼的站点）。在22.2.2节，我们讨论客户机上存储上下文信息的可能性。对浏览器，它可能采取以下几种方式。

1) *在一次会话内——隐藏区域* 当发送一个HTML页给浏览器时，CGI程序C1把上下文信息保存在页面内的**隐藏区域**（hidden field），这样的区域不在屏幕上显示，所以它对用户来说

是不可见的。在用户把可见区域填好后，单击“提交”按钮，页面上的信息（包括可见的和不可见区域的信息）传给页面上指定的CGI程序C2，这样，C1知道的上下文信息就传递给C2。

2) *在一次会话内或会话之间*——cookie 一个用户浏览器包含对应于服务器的文件中cookie的文件，这些服务器是浏览器以前所访问过的。每一个cookie是一个上下文信息的处理，最多包含255个字符数据，是由服务器创建的，对用户进行描述，但对用户来说一般是不可访问的。当浏览器与一个服务器连接时，它把相应的cookie随用户输入的信息传给服务器。当服务器应答时，把cookie（可能作修改）返回给浏览器。

3) *使用*servlet servlet有一个生命周期，存在于整个客户会话期间，因此它包含客户的上下文信息。另外，Java servlet API支持会话管理，它能极大地简化服务器端的处理。

22.5.2 因特网上事务系统的组织

对因特网上的应用，我们讨论事务处理系统三种可能的组织方式。

1) 浏览器作为表示服务器，应用服务器、事务服务器和数据库服务器驻留在因特网上。数据库地点的Web服务器对来自浏览器的请求做出响应，浏览器使用一个HTML页面和一个写成Java applet的应用程序与用户交互。用户在页面上适当的位置填写好后，applet就在浏览器内执行。applet可能访问因特网上的数据库，可能启动一个事务，提交SQL语句给数据库服务器进行处理。applet使用JDBC（参见10.5节）与数据库通信。这个模型和图10-9所示的模型类似。JDBC驱动程序可以从包含有Java applet的Web服务器上下载，此时它与数据库服务器建立一个网络连接。驱动程序也可以驻留在Web服务器上，通过CGI程序把命令作为传输数据从浏览器接收。

2) 浏览器作为表示服务器，Web服务器上的CGI程序作为应用服务器。当与Web服务器接触时，它向与用户交互的浏览器发送一个HTML页面。用户填好页面后，提交给服务器，执行企业规则的CGI程序在Web服务器上启动。程序可能启动包含SQL语句的事务，或者调用驻留在数据库服务器上的存储过程，如图22-4所示，也可能包含对事务服务器的调用，如图22-5所示。在第一种情形下，CGI程序可能使用嵌入式SQL语句访问数据库管理系统，也可能使用一个调用接口（如JDBC或者ODBC）。执行完成后，它返回一个HTML页面给浏览器[⊖]。

3) 在一些应用中，一个三层事务处理系统可能接收来自基于因特网或不基于因特网的客户请求。不基于因特网的客户与应用服务器连接，如图22-4所示。基于因特网的客户通过Web服务器上的CGI应用程序与应用服务器的连接，如图22-18所示。这种情况下，浏览器提供表示服务，但应用服务器把浏览器和CGI程序与不基于因特网的客户一样看待：只是作为另一个表示服务器。对这样的系统，Web服务器上的CGI程序对从HTML页面返回的信息进行处理，使用应用服务器所希望的通信协议，启动应用服务器上相应的应用程序。当应用程序完成后，将信息返回给CGI程序，CGI准备相应的HTML页面，然后将页面返回给浏览器。

⊖ 有些商业应用生成程序通过生成应用来有选择地支持这一模型，这些应用是在Web服务器上执行的CGI程序。应用设计员使用应用生成程序来设计显示在客户端的界面，以及与之相对应的信息处理应用程序。然后，应用生成程序将此设计转换成等价的HTML页面和CGI程序。运行时，系统按如上所述方式执行。

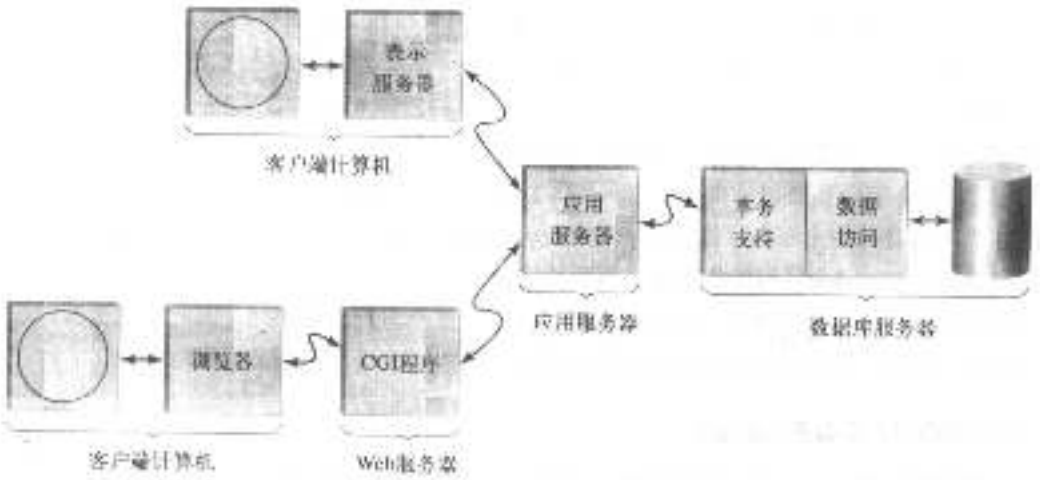


图22.18　基于因特网的客户与三层事务处理系统相连接

22.6　参考书目

本章中的许多材料来自两本书。[Gray 1978]是一本百科全书，它涵盖事务处理系统，学生们应该去看这本非常好的额外信息资源。另一个非常优秀的资源是[Bernstein and Newcomer 1997]。[Gray 1978]着重于实现细节，但[Bernstein and Newcomer]专注于更高层次的相同材料。它们的著作包括关于两层和三层模型非常有用的讨论，以及多种TP监控器的描述。关于Tuxedo的其他信息能在[Andrade et al. 1996]中找到。关于Encina的内容能在[Transarc 1996]中找到。RPC是由[Birrell and Nelson 1984]介绍的。最常用的对等协议——IBM的LU6.2的描述能在[IBM 1991]中找到。通信技术的一般讨论（包括RPC）能在[Peterson and Davies 2000]中找到。基于Tuxedo系统中使用的模型的异常情况处理的讨论在[Andrade et al. 1996]中有所介绍。分布式事务处理系统中X/Open模型的描述能在[X/Open CAE Specification Structured Transaction Definition Language (STDL) 1996]和[X/Open Guide Distributed Transaction Processing: Reference Model, Version 3 1996b]中找到。

有很多著作介绍了使用CGI和Java servlets编写Web应用程序（包括数据库应用）。最近的一些著作包括[Hall 2000,Hunter and Crawford 1998,Sebesta 2001,Berg and Virginia 2000]。

22.7　练习

22.1　试解释对银行的账户数据库的访问只通过存储过程（如deposit()和withdraw()）来完成的好处。

22.2　试解释为什么说三层事务处理系统（包括事务服务器）的组织可扩展成大型企业系统，从下销、安全、维护、认证和授权问题上加以讨论。

22.3　试解释在三层体系结构的事务处理系统中，当事务正在执行时，如果表示服务器崩溃，会发生什么情况。

22.4　试解释在ATM机上，为什么按下submit按钮之后，cancel按钮不工作。

22.5　试解释在事务处理系统的体系结构中包含事务服务器的好处。

22.6　试举出一个你使用的事务处理系统的例子，它是作为包含集中式数据库的分布式系统实现的。

22.7　试举出一个分布式数据库系统的例子，其中事务的提交不满足原子性（它访问的一个数据库管理器提交而另一个异常中止），使数据库处于不一致的状态。

22.8　试解释项目中使用的系统是否可以描述为TP-Lite或TP-Heavy。

22.9　列举出在设计异构分布式事务处理系统时会出现，而在设计同构分布式系统时不会出现的五个问题。

22.10　试解释TP监控器与事务管理器的区别。

22.11　试举出三个不同于事务服务器、数据库服务器、文件服务器的服务器例子，该服务器可能在分布式事务处理系统中被一个应用服务器调用。

22.12　试描述你所在学校的学生注册系统的体系结构。

22.13　试给出两种方法，它们在事务的远程过程调用中是不同于一般的远程过程调用的。

22.14　假设事务使用TRPC从一个远程地点的数据库中更新数据，结果成功返回。在事务执行完成之前，远程地点崩溃，当事务请求提交时，会发生什么情况。

22.15　试解释X/Open中的tx_commit()与嵌入在SQL中的COMMIT语句的不同之处。

22.16　对事务管理器，给出一个使用tx和xa来实现分布式存储点的建议。假设每一个子事务（作为一个事务的一部分）都能定义一个存储点，当它这样做时，也迫使它的孩子事务创建相应的存储点。当一个事务或者子事务回退到存储点时，它的孩子事务也回退到相应的存储点。

22.17　在客户机/服务器系统中，试给出使用应用服务器体系结构的三个好处。

22.18　试给出一个事件的例子，它与书中给出的例子不同，例子中回调函数不是事务的一部分。

22.19　试解释对等通信如何实现远程过程调用。

22.20　试解释使用信用卡在因特网上订购商品时所涉及的身份认证问题。

22.21　实现一个Web上tic-tac-toe游戏，它的显示是由浏览器中的表示服务器来实现的，而游戏的逻辑却在服务器中的CGI程序中实现。

22.22　打印出本地Web浏览器上的cookie文件。

22.23　考虑一个与集中式数据库管理系统接口的三层系统，其中n1个表示服务器与n2个应用服务器相连，n2个应用服务器还与n3个事务服务器相连。假设对每个事务，应用服务器平均调用k个过程，这些过程在任意一个事务服务器中执行，同时每个过程平均执行s条必须由数据库管理系统处理的SQL语句。如果表示服务器平均每秒处理r 个请求，（每个请求在应用服务器上产生一个事务），数据库管理系统平均每秒要处理多少条SQL语句？有多少条消息要流经通信线路？

第23章 隔离性的实现

我们的学校有超过10 000多名本科生。每到注册时，可能有成百上千的学生同时使用学生注册系统。系统必须确保大量用户的并发使用不会破坏数据库的完整性。例如，假设由于教室空间所限，每门课程只允许50名同学选修（这是数据库的一个完整性约束），而目前已经有49名同学选修这门课程。如果有两名同学同时注册这门课程，那么系统必须保证它们之中最多只有一人能成功。

为确保正确的并发调度，一个办法就是顺序运行所有事务，一次执行一个事务。因此，当有两名同学同时试图注册以得到课程的最后一个名额时，先运行注册事务的那名同学就能成功注册，而后运行注册事务的那名同学就会被告知，选课人数已满。这种事务执行的方式称为**串行的**（serial），每个事务的执行是**隔离的**（isolated），即ACID中的I。

让我们回顾为什么会出现隔离性问题。回忆一下，我们假设事务是一致的，即ACID中的C。它表明，如果数据库处于一致性的状态，并且事务的执行是隔离的，那么它就能正确执行。由于数据库已经返回到一个一致性的状态，我们可以开始第二个事务的执行。同时因为第二个事务也是一致的，所以它也能正确执行。因此，如果数据库的初始状态是一致性的，那么事务集的顺序执行（即一次只有一个事务在执行）是正确的。

但是，顺序执行是不实际的。数据库是许多应用操作的中心，所以会被经常访问。让事务顺序执行的系统是不能满足负载要求的。而且，很容易看到，在许多情况下，顺序执行是不必要的。例如，如果事务T_1访问表X和Y，事务T_2访问表U和V，则T_1和T_2的操作可以交叉进行，此时最终的结果（包括数据库管理系统返回给事务的信息以及数据库的最终状态）在先执行T_1后执行T_2和先执行T_2后执行T_1的顺序执行的情况下是一样的。由于已知顺序执行是正确的，则交叉调度也一定是正确的。

事务集的交叉执行比该事务集的顺序执行可能高效得多。事务的执行要求多个系统资源（主要有CPU和I/O信道）的服务。但一个事务的执行一次通常只使用其中的一个资源，多个事务的并发执行就能同时利用这些资源，因此也就提高系统的吞吐量。例如，当CPU在为一个事务做计算时，I/O信道可以为另一个事务提供I/O服务。

但有些交叉调度会使一致的事务运行不正确，给应用返回错误的结果，使数据库处于不一致性状态。因此，我们不允许任意的交叉。第一个问题是我们怎样决定哪种交叉是好的，哪种交叉不好。下一个问题是我们怎样实现一个算法来执行好的交叉，同时禁止不好的事务交叉。我们把这样的算法叫做**并发控制**（concurrency control）。这些就是本章要讨论的问题。

在大多数商业事务处理系统中，并发控制是自动完成的，对应用程序员来说是不可见的，应用程序员设计每个事务，就好像事务在一种非并发环境中执行一样。并发控制允许并发执行，它必须确保每一个事务相对于另一个事务来说是隔离的，毫无疑问，理解并发操作控制的基本概念很重要，这是因为:

1) 使用并发控制来实现隔离（而不只是任意交叉）能有效地缩短响应时间，明显降低事务的吞吐量（用每秒的事务量来衡量）。因此，许多商业系统允许在运行多个事务与降低隔离的级别之间做出选择（有时是作为缺省值）。因为有些设计人员可能试图选择其中的一种方案来提高系统效率，所以理解降低隔离级别如何导致数据库出现不一致的状态和不正确的结果是很重要的。

2) 设计者可以在实现完全隔离和降低隔离级别之间进行选择，并发控制与应用程序内表和事务设计之间的交互会严重影响应用的执行效率。

隔离性是一个复杂的问题。因此，我们的讨论分成两部分。在这一章，我们主要对抽象数据库系统上的隔离性感兴趣。这里的抽象是指：数据库中的每个数据项都有一个名字，对它们的访问可以使用名字来进行读写操作。第24章着重介绍关系数据库系统上的隔离，其中数据是使用SQL语句来访问的，而SQL语句使用条件来确定要处理的行。在抽象的数据库上学习隔离性，有助于我们关注并发控制的关键问题。对特定的关系数据库，就能用一个完好的技术来处理抽象问题。

23.1　调度和等价调度

我们感兴趣的并发控制可以在任何应用中使用。我们并不讨论为某特定应用设计的并发控制。而且，我们也不关注利用特定事务执行的计算信息进行控制的问题，我们关注的是这样的控制，它在不知道事务将做什么的情况下，一定能把好的事务交叉调度与不好的事务交叉调度区分开来。例如，事务可能读取数据库中一个变量的值。如果并发控制知道这个变量表示某银行账户的余额，那么作为储蓄的第一步，事务将请求读数据，它可能选择可接受的交叉调度来使用信息。但我们假设，这样的信息对并发控制来说是不可用的。

如果不能使用特定应用程序的信息，我们怎样才能判断出哪种交叉调度是正确的呢？答案在我们的假设中，即每个事务是一致的，因而串行调度必定是正确的。从这里我们可以看出，我们使用的正确性准则是：所产生的结果与串行调度执行的结果一样的交叉调度必定是正确的。以后我们将对“和一个串行调度有相同的结果”进行提炼，但你必须理解，这是传统的正确性的概念。对大多数应用来说，可能有很多可接受的非串行执行的方法，因此，“不正确”是相对于我们的准则而言的，因为我们假设事务正在处理的信息是不能被利用的。

我们假设，一个事务是一个程序，它的数据空间包括数据库和局部变量。而局部变量仅能被这个事务访问。数据库是全局性的，所有的事务都可以访问它。事务使用不同的机制访问数据空间中的数据库和局部变量。对事务来讲，局部变量是直接访问的（即局部变量在它的实际内存中），但数据库只能通过调用数据库管理器提供的过程来访问。例如，在一个层次非常低的实现细节上，事务请求数据库管理器从数据库中复制一个数据块到它的局部变量中，这是**读请求**（read request），或者请求把存储在局部变量中的数据写到数据库中去，这是**写请求**（write request）。在这一层次上，我们把数据库看作是一个数据项集，并且假设不知道数据项中数据的类型。而且，我们对数据库的存储位置不做假设。大多数情况下，数据存储在大容量的存储设备上，但要求它做出快速响应时，它就可能存储在主存中。

一个事务是一个程序，在这个程序内，使用局部变量的计算和提交给数据库管理器的数据库访问请求是交织在一起的。因为计算是在主存中完成的，对数据库管理器来说是不可见

的，所以它把事务的执行看作是一个读写请求序列，我们把这个序列叫做**事务的调度**（transaction schedule）。如果p_{ij}是T$_i$所做的第j个请求，则

$$p_{i1},\ p_{i2},\ \cdots,\ p_{in}$$

是事务T$_i$的调度，它由n个对数据库的请求组成。

因为事务是并发执行的，数据库管理器必须处理事务调度的合集，我们把它简单地称为**调度**（schedule）。数据库管理器负责为每个到来的请求提供服务，但按它们到达的先后顺序提供服务可能是不正确的。因此，当一个请求到达时，必须决定是否需要立即为其提供服务。这样的决定是由并发控制来完成的。如果并发控制决定立即为一个请求提供服务可能导致一个不正确的调度，它就延迟一段时间后再提供服务，或者它可能连同请求事务一起中止。

因此，在并发控制下，数据库管理器所提供的服务调度可能和请求到达的序列是不一样的。一般情况下，并发控制会记录这些请求。当然，它不会只记录一个事务的请求，因为我们假设事务是一个顺序执行的程序，在前一个请求得到服务之前，它不会再提交另一个请求。并发控制的目的就是按到达的调度记录不同事务的请求，以便产生一个正确的调度，让数据库管理器提供服务。系统的组织如图23-1所示。

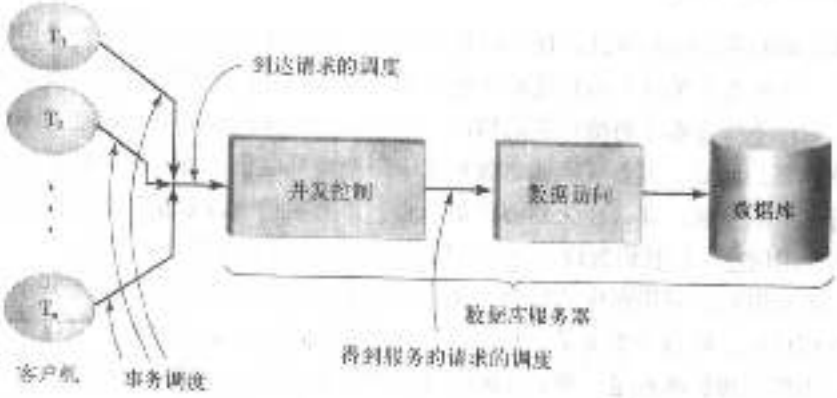


图23-1　数据库系统中并发控制的角色

注意，传输到达的调度表是有开销的：事务可能被延迟，或被异常中止。事务的延迟会降低系统中并发执行的程度，因此增加了系统的平均反应时间，同时降低了吞吐量。异常中止事务是较差的一种情形，因为它要求计算必须重做。所以，并发控制s做必要的转换，但必须是有尽可能多的调度表。并发控制一般不能识别出所有正确的调度，因此有时会作一些不必要的转换操作。设计一个并发控制的目标就是要最小化这种开销。

我们假设，相对于其他的数据库操作而言，每个数据库的操作是原子性和隔离性的。尽管我们已经假设并发控制不知道事务的语义（即计算的实质），但我们假设它知道每个数据库操作产生的影响，我们把它称为**操作语义**（operation semantics）。在这一章中，我们主要考虑读写操作。在下一章中，我们考虑关系数据库上的操作，如SELECT 和UPDATE。

等价调度

操作的语义可用来决定可行的调度。为解释是如何确定可行的调度，我们必须先解释两个等价调度的含义。如果对于所有可能的初始数据库状态可以满足下列条件，我们就说两个

数据库的操作p_1和p_2是可以交换的。

- 按顺序p_1,p_2或者p_2,p_1执行，p_1返回同样的结果。
- 按顺序p_1,p_2或者p_2,p_1执行，p_2返回同样的结果。
- 按这两种顺序执行所产生的数据库状态是一样的。

假设p_1和p_2是两个不同事务发出的请求，并且在调度S_1中是连续的操作，则S_1有下面的形式:

$$S_{1,1},\ p_1,\ p_2,\ S_{1,2}$$

这里$S_{1,1}$是S_1的前缀，$S_{1,2}$是S_1的后缀。假设两个操作p_1和p_2是可以交换的，在调度S_2中有:

$$S_{1,1},\ p_2,\ p_1,\ S_{1,2}$$

调度S_2中所有事务执行的计算与在调度S_1所有事务完成的计算相同，因为这两个调度返回给每个事务的值是一样的。而且，这两个调度使数据库处于相同的终态，因此，我们就说调度S_1和S_2是**等价的**（equivalent)。不可交换的操作是冲突的。

最重要的是，两个对不同数据进行的操作总是可以交换的。对同一个数据的操作有可能可以互换。例如，两个对同一个数据的读操作是可互换的，但对同一个数据的读和写操作是冲突的，因为尽管数据的最终状态独立于它们的执行顺序，但读到的值取决于操作的执行顺序。类似地，两个写操作也是冲突的，因为数据的最终状态取决于写操作发生的顺序。

在任意的调度中，可交换的、属于不同事务的连续操作可以相互交换以产生与原来调度等价的新调度。因为等价是传递的，所以我们能够证明两个调度的等价性，这两个调度可合并成同一事务的调度集，但使用这样简单的顺序交换，实质是不同的。且这种交换对并发控制却是件难事。

大多数并发控制设计是基于以下原理的，它是用另一种方法来证明调度的等价性:

定理（等价调度)：当且仅当冲突操作在这两个调度中按相同的顺序执行，同一操作集的两个调度是等价的。

注意，如果我们能证明下面的命题，我们就能证明这个定理:

当且仅当冲突操作在这两个调度中以相同的顺序执行时，通过交换可交换的操作，调度S_2可从S_1得到。

因为我们知道，当且仅当其中的一个调度可以通过交换可交换的操作从另一个调度得到，两个调度是等价的。

“仅当”是定理的一部分，在交换过程中，保留冲突操作的执行顺序。因此，如果冲突操作在两个调度中的执行顺序不同，那么S_2就不能通过交换从S_1中得到。

“当”是定理中更加复杂的一部分。可以这样来证明，若任意的调度S_2（和S_1有相同的操作集）的冲突操作和S_1的冲突操作有相同的顺序，那么就可以通过交换从S_1产生。为表示这个过程，考虑调度S_1:

$$\cdots p_i,\ p_{i+1},\ p_{i+2},\ \cdots,\ p_{i+r},\ \cdots$$

假设S_2是与S_1有相同操作集的调度，冲突操作的顺序也和S_1相同。而且，假设所有的j满足$1 \leqslant j < r-1$，p_i和p_{i+j}在S_1和S_2中有相同的顺序，但p_i和p_{i+r}的顺序不同。因此p_{i+r}是S_1中p_i之后

的第一个操作，而在S_2中却有不同的顺序，p_{i+r}是先于p_i的。操作p_i和p_{i+r}必须能够交换，因为冲突操作在S_1和S_2中有相同的顺序。

现在假设有k满足$1 \leqslant k \leqslant r-1$，这样$p_{i+r}$就不能与$p_{i+k}$交换。在$S_2$中，操作必须有这样的顺序:

$$\ldots p_{i+k},\ \ldots p_{i+r},\ \ldots,\ p_i,\ \ldots$$

这是因为冲突操作在两个调度中有相同的顺序。但这是与假设相矛盾的，在假设中，p_{i+r}是S_1中p_i之后的第一个操作，这和S_2中的操作顺序是不同的。

由于这个原因，假设存在一个k，满足$1 \leqslant k \leqslant r-1$，使操作$p_{i+r}$不能与$p_{i+k}$交换是错误的。因此，$p_{i+r}$能与$p_i$，…，$p_{i+r-1}$中所有的操作交换。而且，在通过交换相邻操作产生的不同于S_1却和S_1等价的调度中，交换后的操作中p_{i+r}先于p_i，而不是p_{i+r}在p_i之后（在S_2中也是如此）。交换过程是可重复的，操作顺序不同就使S_1变成S_2。

23.1.1 串行化

我们已经看到，如果冲突操作在两个调度中的顺序相同，则调度是等价的。使用这一规则，我们就可以定义和串行调度等价的交叉调度，这是并发控制所允许的。我们把这样的调度称为可串行化调度[Eswaran et al. 1976]。

如果一个调度和一个串行调度等价，即这两个调度中的冲突操作有相同的顺序，那么这个调度是可串行化的（serializable）。

可串行化调度的概念为我们的第一个问题提供答案：我们怎样确定哪种交叉调度是正确的。

因为可串行化调度是和串行调度等价的，并且因为我们假设所有事务都是一致性的，所以任何一个事务的可串行化调度是正确的[⊖]。

可串行化调度对任何应用来说都是正确的。但对特定应用而言，可串行性的条件要求太高（这个应用事务的非串行化调度也有可能是正确的），并且有可能导致不必要的性能损失。因此，并发控制通常实现不同的隔离级别，最强的隔离级产生可串行化的调度，应用设计人员可以为特定应用选择合适的隔离级别。在本章中，我们只处理可串行化调度。我们将在第24章讨论最严格的隔离级别。

为说明串行的含义，假设$p_{1,1}$和$p_{1,2}$是事务T_1的两个连续的数据库操作请求，而$p_{2,1}$和$p_{2,2}$是事务T_2的两个连续的数据库操作请求。一个交叉操作序列是:

$$p_{1,1},\ p_{2,1},\ p_{1,2},\ p_{2,2}$$

如果交换$p_{2,1}$和$p_{1,2}$，那么这个交叉序列和串行调度

$$p_{1,1},\ p_{1,2},\ p_{2,1},\ p_{2,2}$$

等价，因而是正确的。

⊖ 与此相反的结论是，如果调度是不可串行化的，那么一定存在某个完整性约束，而调度违反了这个完整性约束。[Rosenkrantz et al. 1984]证明了这一结论。

图23-2a显示了两个事务调度，每个事务在不同的线上表示，时间从左到右递增。总调度是这两个事务调度的汇集，在空间上表示它们交叉进行。其中$r(x)$表示读数据项x，$w(x)$表示写数据项。在调度中，某一时刻数据项的值是上一次写入的那个值，如果前面没有写操作，则数据的值为初值。

图23-2的a中调度是交叉（非序列的）的，因为T_1的有些操作发生在T_2之前，另一些在T_2之后。图23-2b中的调度是串行的，在T_2开始之前，T_1已经结束。图23-2a中事务T_2对x的读写操作与T_1对y的读写操作是可以交换的，所以，通过一系列相邻操作的交换，图23-2a可以转换成图23-2b。因此，图23-2a中的调度是可串行化的。

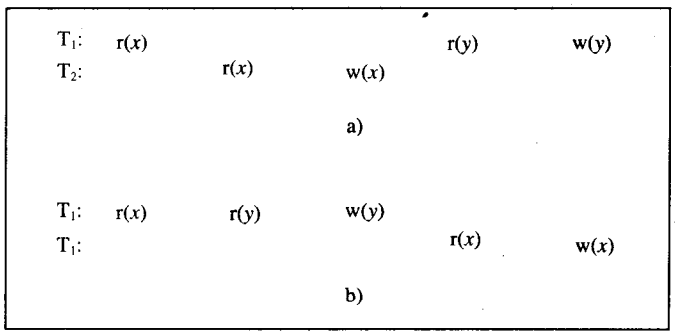


图23-2　a) 可串行化调度；b) 等价的串行调度

现在我们从调度等价定理来考虑这两个调度，两个事务中唯一的冲突操作是T_1的$r(x)$和T_2的$w(x)$，因为它们在图23-2a和图23-2b中的顺序相同，所以这两个事务是等价的。

最后请注意，尽管图23-2a的调度和图23-2b串行调度T_1T_2是等价的，但却和串行调度T_2T_1不等价。

作为另一个例子，图23-3中的调度不是可串行化的，因为T_1读x之后，T_2写x，在任何等价的串行调度中，T_2必须跟在T_1的后面（因为它的写操作是不能和T_1的读操作交换的，所以不能改变它们的顺序）。类似地，因为T_1在T_2读y之后写y，所以在任何等价的串行序列中，T_2必须跟在T_1之后。因为T_1不能同时既处在T_2之前，又处在T_2之后，所以没有串行的等价序列。

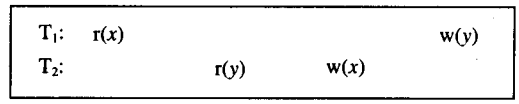


图23-3　非可串行化的调度

尽管可串行化调度和串行调度之间等价问题的争论是基于操作的可交换性和重排序，但在执行一个调度前，并发控制并不对可串行化调度的操作重新排序。因为可串行化调度和串行调度所产生的结果是一样的，所以没有必要对操作进行重新排序。如果并发控制能确定已到达的操作序列是可串行化调度的前缀，那么不管以后提交什么操作，它都按操作到达的顺序执行。但如果不是这样，操作就必须推迟执行，可能需要重新排序以便保证整个调度可串行化。稍后我们对如何完成重排序加以阐述。

23.1.2　冲突等价与观察等价

我们已经知道，如果冲突操作在这两个调度中有相同的顺序，那么两个调度是等价的。

但实际上，这里有两个不同的等价概念：一个是我们已描述过的，因为它特定的属性而称为**冲突等价**（conflict equivalence）的等价概念，第二个是称为**观察等价**（view equivalence）的等价概念。如果两个调度满足下列两个条件，那么它们是观察等价的：

1) 每个调度中相对应的读操作返回相同的值（因此，在这两个调度中，所有的事务完成相同的计算，写相同的值到数据库中）。

2) 两个调度产生相同的最终数据库状态。

第一个条件表明，这两个调度中的事务有相同的数据库视图，视图等价因此而命名。第二个条件是必要的，因为尽管两个调度中的事务写相同的值到数据库中，但如果写操作以不同的顺序发生，那么这两个调度可能使数据库最终处于不同的状态。第二个条件对写语句的顺序进行限制，要求数据库的最终状态是相同的：在每一个调度中，最终的写数据项的操作必须相同。

冲突等价条件是保证视图等价的*充分条件*，但不是*必要条件*。在某些情形下，它的要求更加严格。相应地，视图等价比冲突等价更具一般性（即条件要弱一些）。尽管两个冲突等价调度是视图等价的（在练习23.6中要求证明），但视图等价并不一定是冲突等价：它不可能通过交换相邻操作而从另一个调度得到。

例如，如图23-4所示的调度，它与任何一个串行化的调度都不是冲突等价的。在T_2和T_1中，各自对y的读写操作是不能交换的，因此，如果存在冲突等价的串行调度，T_2必须在T_1之前。另一方面，这些事务对x的两个写操作也是不能交换的，这表明如果存在一个冲突等价的串行调度，T_1必须先于T_2，这是前后矛盾的。注意，事务按T_2、T_1、T_3的顺序执行，串行调度所产生的结果与图23-4中显示的调度所产生的结果相同：x和y最终有相同的最终状态，返回给T_2的作为读取y的结果的值相同。因此图23-4中的调度与串行调度$T_2T_1T_3$是视图等价的，所以它是可串行化的。

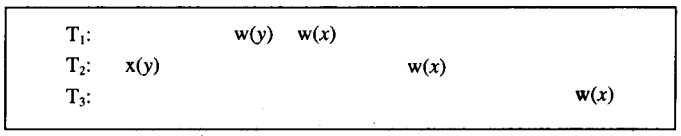


图23-4　表明观察等价不一定是冲突等价的调度

尽管可以基于视图等价来设计并发控制（或许可以获得另外的并发，因为允许串行化程度更高的调度），但这样的控制实现起来是很困难的。因此，并发控制通常基于冲突等价在余下的部分中，除特殊声明外，我们使用“等价”表示冲突等价。

23.1.3　串行图

另外一个考虑冲突串行调度的办法就是基于串行图。某个已提交事务的调度S的**串行图**（serialization graph）是一个有向图，其中的节点代表参与调度的事务。

如果在调度S中，

1) T_i中的某些数据库操作p_i与T_j中的某些操作p_j冲突。

2) 在S中，p_i出现在p_j之前。

那么在有向图中存在有向边，从代表事务T_i的节点指向代表事务T_j的节点，即$T_i \rightarrow T_j$。

例如，对应图23-2a的串行图由下面一条边组成:

$$T_1 \rightarrow T_2$$

因为T_2写x的操作在T_1读x之后。

如果有一条从T_i到T_j的有向边出现在表示S的串行图中，则对于任何调度，我们得出结论，T_i一定要在T_j之前，它才会与S冲突等价。从这里我们可以看出，如果我们想通过交换可交换的操作来得到冲突等价的串行调度，那么我们不能交换产生有向边的冲突操作。例如，在任何一个冲突等价于图23-2a的调度中，T_1必须先于T_2，因为T_2对x的写操作不能与T_1对x的读操作交换。

一个特定调度的串行图可用来推导这个调度的可串行化方法。如图23-3所示，调度的串行图有两条边:

$$T_1 \rightarrow T_2$$

因为T_2对x的写在T_1对x的读之后，另一条边是

$$T_2 \rightarrow T_1$$

因为T_1对y的写在T_2对y的读之后，这两条边构成一个循环:

$$T_1 \rightarrow T_2 \rightarrow T_1$$

因此，我们可以得出结论，在任何等价的串行调度中，T_1必须先于T_2，同时T_2又必须先于T_1。很明显，这是不可能的，因此我们说不存在等价的串行调度，图23-3所示的调度是不可串行化的。更一般地，我们可以得出下面的结论:

定理（串行图）: 当且仅当它的串行图不存在环时，调度是冲突可串行化的。

要证明这个定理，需注意，如果调度S的串行图有环，那么我们可以运用以上的推理来证明它不是可串行化的。另一方面，假设图中不存在环，$T_{i_1}, T_{i_2}, \cdots, T_{i_n}$是一个图的拓扑有序[⊖]事务，它构成一个与之相应的串行调度S^{ser}，调度S^{ser}冲突等价于调度S，因为如果串行图中存在一条从T_r到T_s的边，就存在T_r的p_r操作与T_s的p_s操作相冲突，在调度S和S^{ser}中，p_r必须先于p_s。因为在这两个调度中，冲突操作是有序的，因而它们是等价的（按照前面的调度等价定理），S是冲突可串行化的。

图23-5a是一个比较大的串行图，它没有环，因此可对应于一个可串行调度。图中有许多拓扑序列，因此它的事务存在许多冲突等价串行调度。其中的两个调度如下:

$$T_1T_2T_3T_4T_5T_6T_7$$

和

$$T_1T_3T_5T_2T_6T_7T_4$$

在图23-5b中，串行图没有相应的可串行化调度，因为存在环:

$$T_2 \rightarrow T_6 \rightarrow T_7 \rightarrow T_2$$

⊖ 非循环有向图的拓扑排序是图中任意有序的节点序列，这些顺序与边所表示的顺序是一致的。一个非循环图可能有多个拓扑排序。

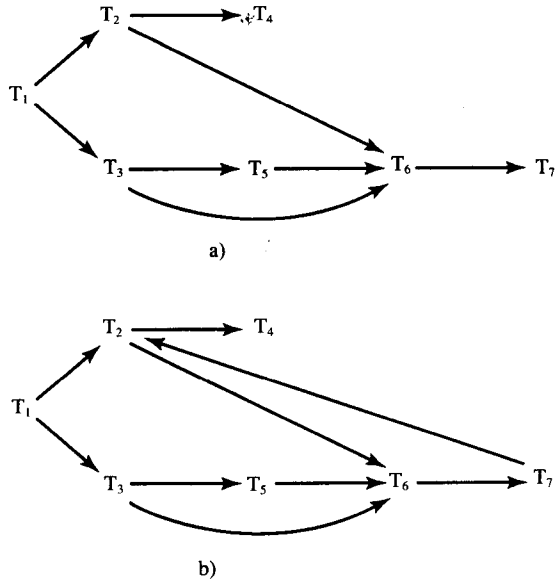


图23-5　两个串行图

23.2　可恢复性、级联异常中止和严格性

迄今为止，我们的讨论是由可串行性引发的，我们假设所有的事务最后都提交。当我们考虑可能有一个事务异常中止时，这个过程就变得更加复杂。这时必须对调度施加额外的限制。当事务提交时，持久性要求它对数据库所做的任何变化是长久的和不易丢失的。当事务异常中止时，必须和它没有被启动所产生的结果一样，但没有提交和没有异常中止都被说成是**活动的**（active）。

当事务异常中止时，它对数据库所做的任何修改必须清除，即事务必须**回退**（roll back）。但回退事务对数据库写入的值并不足以保证异常中止的事务没有产生任何影响。假设T_2将新值写入x，在T_2终止前我们允许T_1读取x的值，如图23-6a所示。T_1的读操作称作**脏读**（dirty read），因为T_2还没有提交。如果我们允许事件序列“T_1提交，T_2异常中止”发生，即使将x的值回退到T_2写入之前，T_2还是会影响T_1。而且，因为T_1已经提交，信息已经写入数据库，即y的新值已经保存。这就产生严重的问题：T_1从T_2读到值，即使T_2异常中止，它也会间接地影响数据库的状态。

注意，如果T_2不异常中止，图23-6a中的调度是可串行化的，“脏读”可能在可串行化调度中发生。因为我们假设不能阻止一个事务异常中止，所以我们必须施加额外的限制来设计并发控制，以保证以上情况不会发生。特别地，在以上的情况下，我们不允许T_1提交。如果T_1没有提交，我们就可以在T_2异常中止的时候，也异常中止T_1，如图23-6b所示。

如果一个事务T_1读事务T_2所写的数据，在事务T_2提交之前，不允许T_1提交（因此，如果T_2异常中止，T_1也异常中止），那么并发控制称作**可恢复的**（recoverable）[Hadzilacos 1983]。我们要求所有的并发控制是可恢复的。由可恢复的并发控制所产生的调度称为可恢复调度。

注意，即使事务不异常中止，也不希望读取脏数据。事务T_2可能多次更新一个数据项，

并发事务T_1读取脏数据可能看到它的中间值，如图23-7所示。从直觉上我们可以得出结论，它不是可串行化调度，因为在串行调度中，中间值是不能被读取的。由于T_1的读操作与T_2的两个写操作冲突，所以图23-7中的调度是不可串行化的。

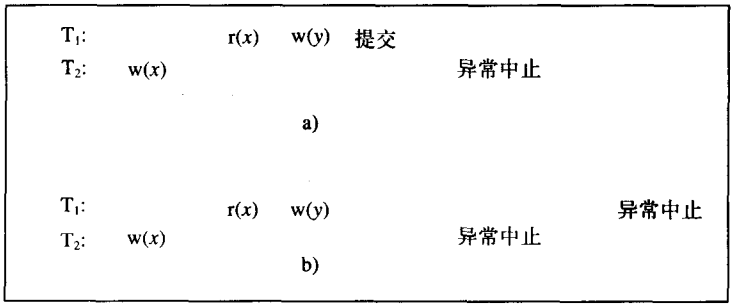


图23-6　a) 不可恢复的调度；b) 相应的可恢复调度

即使一个并发控制是可恢复的，它还可能有另外一种不好的性质：级联异常中止。当一个事务的异常中止引起一个或多个其他事务的异常中止时，并发控制就出现**级联异常中止**（cascaded abort）。

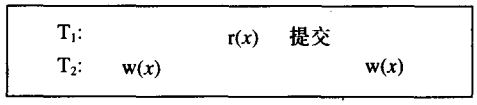


图23-7　不可恢复的调度

假设我们允许事务T_3写入数据库的值在T_3终止前，可以被事务T_2读取，如图23-8所示。如果现在T_3异常中止，那么T_2也必须异常中止，因为它读T_3写入的值。由于同样的原因，T_2的异常中止又引起T_1的异常中止。在一般的情形下，任意多个事务可能被迫异常中止，这是不可取的。因此，我们要求并发控制不会产生级联异常中止。

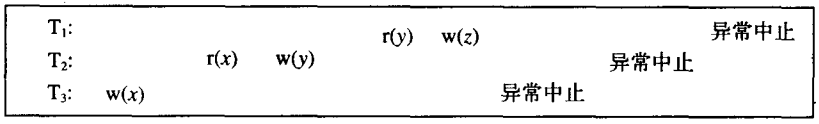


图23-8　说明一个级联异常中止的可恢复调度，T_3异常中止导致T_2异常中止，T_2异常中止又使得T_1异常中止

如果禁止读取脏数据，那么并发控制就不会导致级联异常中止。这个条件比可恢复条件更加严格，可恢复性的条件允许事务T_1读一个活动事务T_2写入的值，但在T_2提交前，不允许T_1提交。但我们选择一个更严格的条件，称之为严格性。如果它不允许事务读或写由一个活动事务写入的数据，那么并发控制是**严格的**（strict）[Hadzilacos 1983]。一个由活动事务写入数据的写操作称为**脏写**（dirty write）。

显然，严格的并发控制是可恢复的，不会出现级联异常中止，但为什么我们要对写施加额外的条件呢？答案在于，在特定的情况下，这样能更加有效地实现回退。一般地，当我们回退一个写数据x所产生的影响时，我们希望将x的值复原到写操作发生之前的状态。假设我们允许x首先被T_1修改，然后被T_2修改，如图23-9所示。如果T_1异常中止，我们根本不必恢复x的值，因为它的值已经被T_2写过，而T_2没有异常中止。如果现在T_2也异常中止，我们必须将x的值恢复到它被T_1写之前的值（不是T_2写之前的值）。尽管我们能够设计系统正确地处理这

样的情况，但严格的系统设计起来会更加简单，因为，我们总是能通过将x的值恢复到它被写之前的值来回退写操作。在非严格的系统中，恢复算法更加复杂，需要对多个事务的写进行分析。

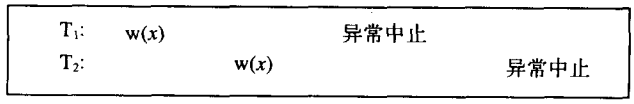


图23-9　表明处理回退的困难性的调度

23.3　并发控制的模型

在这一节中，我们给出几个事务能与并发控制和数据库交互的方法。与数据库的交互可以有立即更新和延迟更新两种。

- 在**立即更新**（immediate-update）系统中，事务的写操作导致数据库中相应数据项的立即更新，读操作返回数据库中数据项的值。可能出现读操作返回还未提交事务写入的值，这意味着并发控制不是严格的，但我们将看到，并发控制算法能阻止这种情况的发生。
- 在**延迟更新**（deferred-updated）系统中，事务的写操作并不立即更新数据库中相应的数据项。相反，信息被保存在事务中称为**意向表**（intentions list）的特殊地点。事务的读操作将返回数据库中相应数据项的值，若数据已被修改过，则返回意向表中的值。当事务提交时，意向表用来更新数据库。注意，读操作返回的值既可能是事务本身写入的值，也可能是已提交事务写入的值。

如图23-1所示，并发控制的目的就是将对数据库的请求序列转换成一个严格的可串行化的调度。当事务发出请求时，控制必须决定是否允许处理请求。控制知道，到现在为止已经被数据库系统服务的（部分）调度，但不知道将要到达的请求。因此，它必须确保，不论随后有什么样的请求序列到达，由数据库系统处理的调度都是可串行化的。控制对特殊请求的响应可能是下列几种形式:

1) 允许请求被处理。

2) 让发出请求的事务等待，直到有其他事件发生为止。

3) 拒绝请求（和异常中止事务）。

并发控制对请求的响应分为同意请求（如上面的第一种响应）、延迟或拒绝。基于这一点，并发控制分为悲观型和乐观型两种。

- 在**悲观**（pessimistic）系统中，无论事务试图对数据库进行什么操作，它必须得到并发控制的许可，但事务不需要请求被允许，可以在任何时间提交。
- 在**乐观**（optimistic）系统中，事务不需得到并发控制的许可，就可以对数据库进行任何操作，但事务的提交必须得到并发控制的许可。

在这两种系统中，事务在它提交前不需要任何请求，可以在任何时间异常中止。

悲观并发控制的设计是基于不利情况有可能出现的心态，即事务对数据库的访问可能发生冲突。因此，悲观控制对每个请求进行检查，只有当后续的请求不会使调度变成非可串行化时，才允许对请求进行处理。控制做出“最坏情况”的决定，因此被称为悲观控制。因为

它能确保最终的调度是可串行化的，所以提交请求总是能被批准。

另一方面，乐观控制的算法是基于不利情况不可能出现的心态，即事务对数据库的访问不可能发生冲突。因此，乐观控制立即批准每个访问请求。但当事务请求提交时，控制必须检查以确保不利情况不可能发生，否则，事务的提交请求不被批准。

对大型数据库而言，乐观算法是比较适用的，因为每个事务只对数据库的某些数据项进行访问，而这些访问随机地分布于整个数据库。这样的假设违反得越多，就越有可能发生冲突。因此，提交请求被拒绝。特别地，乐观算法对存在**热点**（hotspot）的数据库是不合适的，因为数据项会被可能产生冲突的多个事务频繁地访问。

并发控制通常以这两类交互选择为特征，最常见的并发控制是立即更新的悲观系统，在这里我们会予以详细的阐述。我们也会对延迟更新的乐观系统作简短描述，因为它在一些情况下是非常有用的。我们不对延迟更新的悲观系统进行讨论。

23.4　立即更新的悲观并发控制策略

我们要求并发控制是严格的。决不允许事务读或写数据库中被另外一个仍处在活动状态的事务写过的数据项。我们认为这一限制能有效地确保可恢复性、防止级联异常中止以及有效地回退。在这一节，我们对与立即更新悲观系统相关的其他难点进行讨论，并提出解决的办法。

23.4.1　避免冲突

假设事务T_2已对x写入新值，在T_2仍处于活动状态时，事务T_1发出一个冲突请求（如请求读x的值）。我们看到，为确保严格性，这样的请求是不被批准的，但让我们在这时忽略严格性，而只考虑可串行化要求。因为我们假设在立即更新系统中，如果T_1的读请求被批准，将返回T_2写入的值。因此，如果并发控制批准这个请求，那么任何串行序列中，T_1必须在T_2之后。如果后来T_1和T_2要求访问另一个数据y，那么并发控制必须记住它们之间已存在的顺序，确保对y的访问不与这一顺序矛盾。因此，如果T_1和T_2对y的访问使得T_2必须跟在T_1之后，那么这两个矛盾的序列就不可能使任意一个串行序列与最终的结果调度等价。

以上问题还可能有更严重的危害。假设T_1访问x后，对y写入一个新值，然后T_1提交，最后T_2请求读y的值，如图23-10所示。并发控制不能批准读操作，因为结果调度不是可串行化的。因为T_2不能完成，它必须异常中止，但这太不可思议，因为T_1读T_2写的数据，T_1不可能异常中止（它已经提交）。

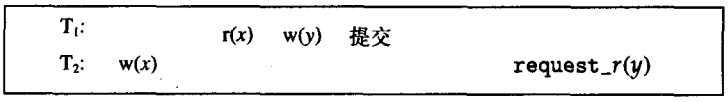


图23-10　说明如果要保持可串行性，活动事务的冲突请求不被批准的调度

这个问题可以通过延迟T_1的提交来避免，但这只会导致级联异常中止。为避免这个问题又能确保可串行性，我们要求并发控制遵守下列规则：

并发控制以下列方式来批准一个请求，它不规定活动事务的顺序。

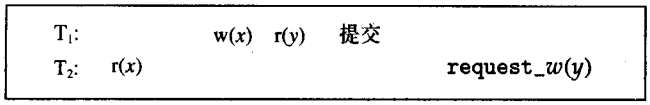


图23-11　说明如果要保持可串行性，活动事务的冲突请求不被批准的调度

为实行这一规则，如果控制已批准一个仍处在活动状态的事务的冲突请求，控制不批准其他事务的请求。在这个例子中，T_1的读x请求被延迟，直到T_2不再处于活动状态为止。记住，因为事务是顺序执行的程序，所以延迟请求实际上是延迟整个事务。这种情况下产生的调度是

$$w_2(x)r_2(y)commit_2\ r_1(x)w_1(y)commit_1$$

这里每一个操作下角的字母表示执行这个操作的事务。

注意，这个规则仅排除脏数据的读取，如图23-10所示。任何与前一个仍处在活动状态事务的请求冲突的请求将被延迟，所以图23-11所示的调度，从规则的角度来看，它和图23-10的问题一样。

如果T_1和T_2的请求不冲突，那么它们都将被批准。如果下列条件之一为真，请求就不会冲突:

1) 请求操作不同数据。

2) 两个请求都是读操作请求。

图23-12以表格的形式显示冲突关系。

	允许的模式	
请求模式	读	写
读		×
写	×	×

图23-12　立即更新的悲观并发控制冲突表，× 表示锁模式之间的冲突

需要说明的是，基于冲突的并发控制所产生的调度既是严格的，又是可串行化的。下面的定理陈述这个结论:

定理（*串行提交顺序*）: *如果并发控制已经批准一个仍处在活动状态的事务的冲突请求，且不批准其他事务的请求，那么它产生的调度是严格的，并且事务的提交顺序是可串行化的*（*称为提交顺序*（commit order））。

显然，这样的并发控制所产生的调度都是严格的，因为没有事务能够读或者写由另一个仍处在活动状态的事务写入的数据。难点在于，证明提交顺序的可串行化。首先我们必须处理这样的事实，即调度可能包含还没有完成的事务的操作（在前面，我们的讨论限制在已完成事务所产生的调度）。当我们说一个调度是可串行化的时候，意味着它和符合以下条件的调度是等价的，即在这个调度中，已提交事务的操作是不能交叉执行的，它发生在未提交事务之前。以后，我们的串行调度的概念中包含这样的调度。

可用归纳法来证明这一结果，归纳法是基于数字i（它是调度中已提交的事务数）来完成的。考虑最基本的情况，i=1，表示仅有一个事务T_1已提交，其他事务都处在活动状态。这时，调度有下列形式: *pre-commit*、$commit_1$、*post-commit*。这里，$commit_1$是T_1的提交操作，*pre-commit*是发生在$commit_1$操作之前的操作序列（所有事务的），*post-commit*是发生在$commit_1$之后的所有操作。*pre-commit*和*post-commit*都不包含任何提交操作。因为在$commit_1$之前所有的事务都处在活动状态，所以它遵从我们关于并发控制的假设，即*pre-commit*中的请求不和调度中的其他请求相冲突。因此，*pre-commit*中所有的操作是可互换的。特别地，T_1的操作（所有

在*pre-commit*中的操作）可以和其他事务的所有操作互换来产生一个等价的串行调度。

现在假设所有恰好含有i个提交事务的调度在提交顺序上是可串行化的。考虑一个调度S，它含有i+1个已提交事务，设T是最后一个提交的事务，且设$S=S_1 \cdot S_2$，这里S_1是到T的提交操作（但不包括T）S的前缀，因此S_1包含i个已提交事务。从假设可知，它在提交顺序上是可串行化的。设S_1^{ser}是等价于S_1的可串行化调度，因此，S等价于调度$S_1^{ser} \cdot S_2$。调度有如下的性质，即T的所有操作都在第i个提交操作之后完成。而且，T的所有操作必须能与S中未提交事务的所有操作互换，所以T的操作能采取下列方式进行交换，即这些操作在所有已提交事务之后，未提交事务之前。再强调一次，每对交换使结果调度是串行的，与S也是等价的。

因为那些仅访问不相交数据项的事务可以以任意的顺序排列，所以一个可串行化的调度能和不止一个串行调度等价，但正如我们以上所述，其中的一个串行顺序是提交顺序。

尽管我们要求的是调度按照某种顺序可串行化，但用户可能希望事务按提交顺序执行。例如，事务经常有一些外部操作对用户来说是可见的（一个存款事务输出一个收据），用户希望等价的串行顺序与这些动作一致。因此，用户在观察到T_1的外部动作后启动T_2，他希望一个等价的串行顺序，其中T_2在T_1之后执行（收据打印好后，启动一个取款事务应该能看到数据库中存款的结果）。为确保这些外部动作与等价的串行序列相同，一个办法就是按提交顺序将其串行化。并发控制通常按提交顺序串行化事务。

23.4.2 死锁

当事务发出的请求与另一个活动事务执行的操作冲突时，串行性要求此时不批准这个请求。发出请求的事务会一直等待，直到冲突消失（因为其他事务提交或异常中止）为止，但这会导致死锁的发生，如图23-13所示。当T_1请求读y时，系统要求它等待（直到T_2提交或异常中止为止），当后来T_2请求读x时，系统也要求它等待（直到T_1提交或异常中止为止）。如果不采取措施，这两个事务都将永远等待，这是我们非常不愿意看到的情况。

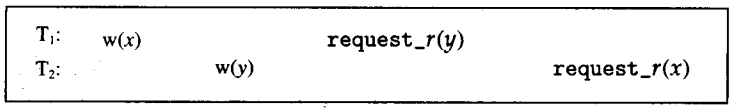


图23-13 说明死锁的调度

更一般地，当有事务相互等待的环时，我们就说存在**死锁**（dead lock）。

使事务等待的并发控制必须提供一些机制来处理死锁。常见的办法就是构建一个数据结构来表示等待（waits_for）关系：如果（T_1，T_2）是一个waits_for元素，它表示T_1在等待T_2。当冲突被检测到时，就能创建waits_for。如果T_1的请求与另一个已得到批准的活动事务T_2的操作相冲突，那么（T_1，T_2）就被插入到waits_for中。类似地，如果T_2正在等待T_3，那么（T_2，T_3）就被插入到waits_for中，这个过程一直继续下去，直到又回到它自身为止，因而就检测到一个死锁，或者在一个不是处于等待状态的事务那里终止。如果允许T_1等待会导致死锁发生，那么并发控制就异常中止，然后重新启动环中的某个事务（通常是T_1）。异常中止对启动这个异常中止事务的用户来说是不可见的。

第二个处理死锁的机制是**超时**（time-out）。如果事务等待执行一个操作的时间超过某阈值，控制就认为发生死锁。结果，它会异常中止这个事务或者重新启动这个事务。

最后，时间戳技术[Rosenkrantz et al. 1978]能用来防止死锁的发生（与死锁检测相对应）。并发控制使用当前的时钟值在一个事务启动时作为事务的时间戳。假设时钟比事务启动的频率向前走的更快，每个事务的时间戳是唯一的。如果发生冲突，并发控制就使用两个事务的时间戳来决定是等待还是重新启动。例如，它可能采取年长的事务不等待年轻事务的策略（通过异常中止年轻事务来实现）。如果一个事务在因死锁而导致的异常中止后重启，它将维持其原有的时间戳。这样，控制就能确保事务最终得以完成，因为它最终会变成最年长的事务。

23.5 立即更新的悲观并发控制的设计

实现立即更新的悲观并发控制的标准技术称为**加锁**（locking）。当事务请求对数据库某特定数据项执行操作时，系统试图为这个事务在这个数据上获得一个适当的锁。例如，对于读操作，它就试图获得**读锁**（read lock），对于写操作，它就试图获得**写锁**（write lock）。在获得锁之前，事务不能对数据进行操作。写锁比读锁的要求更加严格，因为一旦事务对数据加写锁，它就既能对数据进行读操作，又能对数据进行写操作。

基于我们以前的讨论，我们可以得出以下结论：

- 并发控制仅在没有其他活动事务对数据施加写锁时允许事务对特定数据加读锁。因为即使其他事务也对这个数据施加了读锁，事务仍可以对这个事务加读锁，所以读锁常被称为**共享锁**（shared lock）。
- 并发控制只有在没有其他活动事务对这个数据施加读锁或写锁时允许事务对特定数据加写锁。因此，写锁常被称为**排他锁**（exclusive lock）。

23.5.1 锁集和等待集的实现

为实现加锁，我们假设与并发控制相关联的每个已加锁的数据项x相关数据结构称为锁集（lock set）$L(x)$，它描述并发的活动事务在x上加的锁。在以上的规则中，若$L(x)$非空，则它可以包含描述在数据x上的读锁的多个项或描述x上一个写锁的数据项。

类似地，我们把与每个已加锁的数据项x相关的数据结构称为**等待集**（wait set）$W(x)$，它表示有事务对数据库中的x发出操作请求，但还没有获得锁。因为在一个数据库中通常存储大量的数据，保持锁和对那些没有被引用的等待集会使效率低下。因此，这些集合通常是在它第一次引用时动态分配的，处理锁的总开销包括管理存储的时间和存储集合中各项的空间。

最后，我们把与每个活动事务T_i相关的数据结构称为一个**锁列表**（lock list）L_i，它是所有已获得锁和其他数据等待集中的事务项的列表，注意，T_i的锁列表可能最多有等待集中的一个事务，因为一旦事务的请求被放入到等待集中，事务就被挂起，直到请求从等待集中删除，事务才会被恢复。

T_i访问x的请求可以被当作是检查和批准锁的并发控制中的例程的调用。例程通过执行下列步骤对$L(x)$、$W(x)$和L_i进行操作：

1) 如果T_i已经获得对x的读锁，读请求就被批准。如果T_i已经获得对x的写锁，读写请求都被批准。

2) 如果T_i还没有被批准对x加合适的锁，就搜索$L(x)$和$W(x)$，寻找与请求访问冲突的事务。例如，如果T_i请求读x，一个冲突项可能是$T_j(i \neq j)$的写锁，或者是T_j已经发出的对x的写请求，

该写请求悬挂在W(x)中。如果这里没有冲突项，就批准T_i的请求，在L(x)和L_i中插入项，说明T_i和所需的锁类型，恢复T_i，我们说T_i*对x加锁*。

如果在L(x)中存在冲突项，就延迟请求，将T_i和请求的锁类型插入到W(x)和L_i中，阻塞T_i。类似地，如果在W(x)中有一个冲突项，那么以同样的方法延迟请求。在后面的这种情形下，通常T_i会等待，以避免使另一个正在等待对x进行访问的事务T_j处于一种饥饿（starving）状态。如果在W(x)中的项无限制地等待，就会出现饥饿的情形，这是由于后来的事务抢先得到服务而造成的。例如，T_j在等待一个数据项的写锁，而这个数据已经被别的事务加读锁，如果T_i也有一个读请求，它就能立即得到服务，因为它不与已存在的锁发生冲突，但如果调度算法在这种情况下对T_j的请求置之不理，一系列的读请求就会使T_j无限期等待下去，T_j处在一种饥饿的状态，我们说这种算法是**不公平的**，因此T_i必须等待。

3) 如果让T_i等待，则可能导致死锁。如果T_j等待T_i（暂时地），那么有可能出现这样的一种情形。而T_i使用23.4.2节描述的*waits_for*关系检测。如果检测到死锁，T_i就重新启动（或者在这个环中的其他事务）。

4) 当T_i提交或异常中止时，就使用L_i来定位和删除锁集中T_i所有的项（因为T_i不再处于活动状态）。如果将从L(x)中删除一个锁，且W(x)非空，T_i对x的锁至少与一个正在等待的请求发生冲突，那么该请求就可以被批准（如果T_i对x施加一个读锁，并且还有其他事务施加的读锁，那么就不可能批准W(x)中的一个请求）。在这里，有几种策略可用来改进（promote）W(x)和L(x)。例如，一个公平的策略是以FIFO（先来先服务）的顺序检查W(x)的请求。如果准许一个请求，就把它移到L(x)中，并检查下一个W(x)请求，如果不批准请求，就得检查更多的请求。另外，如果列表中的第一个请求是读请求，就批准列表中所有的读请求。如果列表中的第一个请求是写请求，就只批准这一个请求。这是按FIFO顺序从服务请求中分离出来的，它不会导致饥饿。当推进过程完成后，L_i就被删除。

并发控制保证它批准的所有调度都是可串行化的，因为它只批准这样一些请求，这些请求可与先前已经批准给活动事务的请求进行交换。

加锁算法有自动获得锁的性质。事务只要发出访问数据项的请求，并且当请求被批准时，并发控制会自动地记录事务所获得的锁。在事务完成前，所有的锁都必须保持，在事务执行完成时，控制自动地释放对这个事务所加的锁，因为写锁是排他的，所以必须保持到最后，自动加锁遵守严格调度。

23.5.2 两段锁

一般来说，加锁是自动完成的，但有些系统允许手动加锁和解锁。在这种情况之下，事务在发出另一个访问请求之前，它明确地向并发控制发出锁请求。和以前一样，只有并发控制确信请求的事务当前可以获得合适的锁时，访问才会被批准。

除在事务终止时，事务持有的所有的锁会自动解除外，解锁也可以手动完成。手动解锁可以有更大的灵活性，因为和自动解锁相比，如果事务在终止之前释放锁，并发事务就能在更早的时候访问数据。但是，为加强严格性，事务不能提前释放写锁（如果是这样，其他事务就能对这个数据项加锁并访问，而这时第一个事务还处在活动状态）。因此，提前释放仅限于读锁。

而且，除非能正确地提前释放读锁，否则会导致不可串行化的调度。例如，考虑图23-14所示的调度。其中，l(*x*)是对*x*的加锁请求，u(*x*)是对*x*的解锁请求。T_1读*x*，解锁，然后读*y*。在这两个访问之间，T_2对这两个数据项作冲突访问。调度不是可串行化的，因为每个事务必须以串行的顺序跟在另一个事务之后。如果锁是由并发控制来自动处理的，这种情形就不会发生，因为锁必须一直保持到提交时才能释放。下面我们说明在手动系统中，通过要求事务在释放锁之后，不能再对数据加锁就可以避免这种情况。

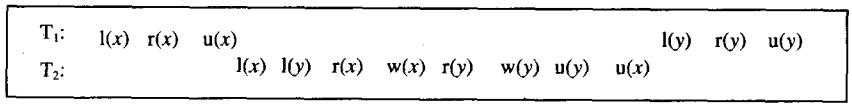


图23-14　涉及不是两段锁事务的不可串行化调度

如果一个事务在解锁之前，保持所有的锁（首先是加锁阶段，然后是解锁阶段），则该事务被说成保持**两段锁协议**（two-phase locking）[Eswaran et al.1976]。图23-14的调度不是两阶段。自动加锁是两阶段。

定理（两段锁）：使用两段锁协议的并发控制产生可串行的调度。

在23.1.3节有使用串行图定理对这个定理的证明[Ullman 1982]。当且仅当调度的串行图无环时，这个调度是冲突可串行化的。这是用矛盾法来证明的，假设由两段锁的并发控制所产生的调度串行图包含环：

$$T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$$

边$T_1 \rightarrow T_2$表示在调度中T_1有操作与前面T_2的操作相冲突。因为操作冲突，所以在执行这两个操作时，T_1必须释放锁，T_2获得锁。T_2和T_3的情况类似，所以可以得出这样的结论，在T_n获得锁之前，T_1释放锁。因此，T_1释放一个锁之后又获得一个锁，这违反了两段锁协议。因此，我们得出矛盾，这样就得出结论，由一个两段锁并发控制所产生的串行图的所有调度必定是无环的，因而是可串行化的。

在使用两段锁协议时，一个可能的等价串行顺序是事务各自执行第一个释放锁的顺序（在练习23.8中要求证明这一结论）。因此，如果调度包含事务T_1和T_2，T_1的第一个解锁请求发生在T_2的第一个解锁请求之前，则在一个等价的串行顺序中，T_1就先于T_2。如果事务将锁一直保持到提交时，那么串行顺序就是提交顺序。

两段式并发控制将锁保持到提交时（例如，自动加锁），那么说它是满足**严格的两段锁协议**（strict two-phase locking protocol）。严格的两段锁协议的定义比严格的并发控制的定义更加严格，因为后者要求控制保持“写锁”到提交时刻，但允许“读锁”提前释放（但仍处在一个两阶段行为中）。

因为等价的串行顺序是在运行时确定的，所以两段锁也称作**动态协议**（dynamic protocol），它和**静态协议**（static protocol）相反。在静态协议中，顺序是由事务的启动顺序来决定的。23.8.1节所讨论的时间戳顺序的并发控制是一个静态协议。

23.5.3　锁的粒度

我们把被加锁的实体称作数据项，但并没有解释它是什么样的数据项。现在我们把数据

库中被并发控制算法加锁的数据项定义成任意的一个实体，它可以是变量、记录、行、表、文件等。在本章中，所有的算法都假设，一个数据项有唯一标识它的名字（在第24章，我们将看到情况并不总是这样）。这个名字用来在访问数据项的时候引用这个数据项。数据项的锁集和等待集经常按它的名字散列定位。

被锁定的实体大小决定锁的**粒度**（granularity）。实体越小，则锁的粒度**越细**（fine），否则**越粗**（coarse）。锁的粒度越粗，则加锁的算法越保守。因此，在数据库管理系统中，只支持表的锁，当访问一行时，会将整个表锁定。显然，串行化没有受到锁粒度的影响。因为只要被访问的数据项被锁定，两段锁并发控制就能产生串行化的调度，即使锁定了其他不需要锁定的数据项。细粒度的锁可以使事务的并发程度更高，因为事务只需对它实际要访问的数据项加锁。不过，与细粒度加锁相关的开销会更大。通常事务要保持的锁越多，就要有更多的空间来保存锁的信息。当在相同的锁定了的实体内，事务要访问多个数据项时，粗粒度的锁可以解决这些问题。例如，一个事务可能访问一个表的多行，只要对这个表加锁就可以。

许多系统实现页锁作为一种折中，即对存储数据项的页面加锁，而不是对数据项本身加锁。页地址是被锁定的实体名。页锁是保守的，因为不仅要给需访问的数据项加锁，还要给存储在该页上的其他数据项也加锁。

23.6 对象和语义交换*

立即更新的悲观并发控制设计是基于数据库操作的交换性的。在简单系统中，我们已经讨论过，只有读和写操作，对特定的数据项而言，只有两个读操作是可以交换的。

如果我们想要执行更加复杂的数据库操作，且保证每个复杂操作的执行与其他复杂操作的执行相隔离，我们就可以使用这些操作的语义来决定哪些操作可以交换，使用这些信息来设计并发控制。在本节中，我们讨论对象数据库（第16章），这里的操作是在对象上定义的方法，这些对象存储在数据库中。

作为对象数据库的例子，考虑一个银行的应用，其中一个账户对象有deposit(x)和withdraw(x)操作。我们假设账户余额不能为负，因此，当账户上至少还有x元时，withdraw(x)就会返回一个OK值，取款操作成功。如果账户上的余额低于x元，就会返回一个NO值，取款操作不成功。

这两个操作的实现都涉及对数据库的读操作（获得账户余额）和写操作（存储新的余额）。因为并发控制在非对象数据库中对读和写的处理是分开的，所以它检测读写操作的冲突，这些读写操作是不同事务对同一账户的操作。但假设并发控制能接受调用方法的请求。它就会发现两个对同一账户的deposit()操作是可交换的，不管它们以什么样的顺序执行，它们会返回相同的信息，数据库的最终状态也是相同的（账户中增加两次存款的值）。我们的分析错在哪里呢？为什么更高级别操作的交换性明显而低级别操作的交换性却不是呢？

答案在于，在决定两个存款操作是否可交换时，我们抽取的是算法信息：存款语义。但读和写几乎没有包含语义信息：我们不知道在事务计算中是如何读信息的，我们也不知道读信息和写信息之间的联系。而且，低级别存款的读写操作是单独的操作，我们不能保证其他事务的操作没有插入到这两个操作之间。

教训在于，在更高级别的操作上，获取的操作语义信息越多，并发控制就能识别更多的可交换性。于是，我们可以总结出，一个大型交叉调度集和串行调度是等价的，而且需要的重新排序很少。因为重新排序涉及延迟，操作语义的使用能提供更多的并发性，并改善性能。

两个存款操作是可以交换的，但对同一账户的deposit(y)和withdraw(x)操作却是冲突的。因为如果withdraw()在deposit()之后执行，可能返回OK值，但如果withdraw()在deposit()之前执行，则可能返回NO值（例如，在操作执行之前，如果余额是$x-y$元）。所以我们可以构造一个图23-15所示的冲突表，使用这张表来作并发控制设计的基础。表中的行和列对应于数据库中的操作，对每个这样的操作，都有相应的锁来控制。例如，当事务想在一些账户上调用deposit()时，它就请求一个该账户对象上的存款锁。如果在这个对象上没有其他事务的取款锁，则请求就被批准，否则，事务就必须等待。注意，与基于读写操作的控制来说，这种控制能获得更强的并发性。因为前者不允许并发事务对同一对象执行存款操作，而基于图23-15的控制却可以执行。

	授权模式	
请求模式	deposit()	withdraw()
deposit()		×
withdraw()	×	×

图23-15　账户对象上的冲突表。
× 表示锁模式之间的冲突

如果我们把另外的两个概念运用到并发控制设计中，就能获得更多的并发性。

部分操作和向后交换

根据数据库的初始状态，我们可以将一个能产生几个可能结果的操作用其他几个操作来代替。例如，我们能用以下两个操作替换withdraw(x):

- withdrawOK(x): 它在账户对象的余额大于或等于x元时执行。
- withdrawNO(x): 它在账户对象的余额小于x元时执行。

我们假设，当事务提交执行取款操作的请求时，并发控制检查账户中的余额，决定是执行withdrawOK()还是执行withdrawNO()。这些新的操作称为**部分操作**（partial operation），因为每个操作的定义只是针对初始状态的一个子集（为所有状态定义的操作称为**全部操作**(total operation)）。

我们可以为部分操作定义新的交换——**向后交换**（backward commutativity），并使并发控制设计基于这些新的定义，考虑下面的操作序列:

$\text{withdrawOK}_1(x)$, $\text{deposit}_2(y)$

这个操作仅在余额z至少是x元的情况下定义。在所有的状态中，可以定义操作序列

$\text{deposit}_2(y)$, $\text{withdrawOK}_1(x)$

因为在存款操作之后，余额是$z+y$，这肯定比x要大（因此定义withdrawOK()）。而且，对这两个序列而言，数据库的最终状态是一样的：余额是$z-x+y$。我们说deposit()向后与withdrawOK()交换。

更确切地说，如果在所有的数据库状态中，定义操作序列q、p和操作序列p、q。操作序列p和q返回同样的结果，数据库最终的状态相同，那么就说操作p与操作q**向后交换**

(backward-commute) [Weihl 1988]⊖。如果p不能向后与q交换，就说它与q**冲突**(conflict)，这个交换的定义和在23.1节给出的完全操作的交换定义是不同的。在完全操作的交换定义中，两个操作若满足以下条件，就是可交换的：它们从*任意的数据库状态*开始执行，以不同操作序列完成等价的动作。相反，操作p向后与操作q交换的条件是，它们从*定义序列*q，p的*任意数据库状态*开始执行，以不同顺序完成相同的动作。因此，新的定义使用部分操作的定义。

注意，向后交换不是对称关系。对特定的操作p和q，p可能向后能与q交换，但q却不一定能向后与p交换。例如，我们已经看到，deposit()能向后与withdrawOK()交换，但withdrawOK()却不能向后与deposit()交换，因为这里定义序列

$deposit_2(y)$, $withdrawOK_1(x)$

的状态与定义序列

$withdrawOK_1(x)$, $deposit_2(y)$

的状态不同。但对有些成对的操作而言，向后交换是对称的（例如，两个并发的withdrawOK()）。

和完全操作上使用的标准交换一样，并发控制设计可以运用同样的方法来使用部分操作上向后交换的概念。例如，事务在一个账户对象上有withdrawOK()锁，另外一个事务在这个对象上请求deposit()锁，该请求就能被批准，因为deposit()能向后与withdrawOK()交换。

更一般地，如果事务T_1在一个对象上有一个q锁，另一个事务T_2在这个对象上请求一个p锁，如果p向后能与q交换，则请求被批准。注意，因为T_2的操作与T_1的操作向后交换，所以这两个操作可能以反序执行，因此控制不用确定T_1和T_2之间的顺序。

基于这个想法，我们把图23-15的冲突表扩充为图23-16的冲突表。在图中插入一行，列表示相应行的操作能与相应列的操作向后交换。表中的 × 表示相应行的操作不能向后与相应列的操作交换。

	授权模式		
请求模式	deposit()	withdrawOK()	withdrawNO()
deposit()			×
withdrawOK()	×		
withdrawNO()		×	

图23-16　在一个账户上使用部分操作和向后交换的冲突表

× 表示相应行的操作不能与相应列的操作向后交换

表中少量的冲突表明，使用这张表的并发控制将允许其他并发。但获得这种并发涉及额外的运行时开销。当一个取款操作调用时，并发控制必须访问数据库来确定账户余额，这样它就知道是请求一个withdrawOK()锁还是一个withdrawNO()锁。当控制是基于图23-15的表时，并不会出现额外的开销，它与完全操作（而不是部分操作）相关，因为完全操作可以在所有的状态下交换。

⊖ 概念foward commutativity的定义参见练习23.19。

原子性、可恢复性和补偿操作

在一个对数据库的访问只使用读和写两种操作的系统中，只有写操作会改变数据库的状态。因为写入一个数据项x与其他在x上的操作相冲突，一旦事务T写x，在这个操作提交之前，其他事务不能访问x（假设是严格的两段并发控制）。因此，如果T失败，只需将x恢复到写之前的值即可，我们把这个过程称为物理恢复。

在支持抽象操作的系统中，异常中止更加复杂，因为两个要修改同一个数据项（如两存款操作）的操作不能冲突。假设T_1使用操作p修改x，然后T_2使用一个非冲突操作q修改x，然后T_1异常中止。我们不能只将x的值恢复到执行p操作之前的值，因为p和q 的操作结果都会丢失。我们已在21.2.4节的多级事务中讨论过这个问题，并总结出补偿是撤销一个异常中止事务影响合适的办法。对每一个操作p，必须有一个补偿事务p^{-1}，这样，在所有定义p的数据库状态中，也定义序列p^{-1}，执行序列p^{-1}会使数据库回到未修改状态。因而，如deposit(x)能补偿withdrawOK(x)。

如果事务T_1和T_2分别使用操作p和q访问同一个抽象数据x，然后T_1异常中止，那么问题就很复杂。假设操作以p、q的顺序出现，p的补偿操作是p^{-1}，因此在调度中的补偿结果是p，q，p^{-1}。我们怎样才能确保在q修改x之后，执行p^{-1}能正确地撤销p的影响？我们怎样才能确保定义p^{-1}的数据库状态是q执行之后的数据库状态？幸运的是，只有当q与p可向后交换时，我们的并发控制可以调度q。因此这个调度是和调度q，p，p^{-1}等价的。因为p^{-1}在这个调度中定义，所以它必须也在最初的调度中定义。这个调度和q等价，因而补偿能正确的工作。

我们所讨论的这个简单的例子是可恢复的（第23.2节），其中补偿能正确地工作，产生原子性。更一般地，考虑一个事务T和事务调度

$$p_1,\ p_2,\ \ldots,\ p_n \tag{23.1}$$

如果在执行完某个操作p_i后，T异常中止，有必要反序执行p_1，p_2，…，p_i的补偿事务，从而撤销异常中止对T的影响，则事务调度就是下列情形:

$$p_1,\ p_2,\ \ldots,\ p_i,\ p_i^{-1},\ p_{i-1}^{-1},\ \ldots,\ p_1^{-1} \tag{23.2}$$

因此对事务T，有n个可能的“异常中止”调度，每一个i（取决于T异常中止的时间）值对应一个异常中止调度。

所有这些调度对数据库都没有任何影响。当我们执行T时，我们不能预测它的这n+1个调度（(23.1）或（23.2)）哪一个会发生。而调度中的异常中止操作意味着要完成一个复杂的动作，涉及物理撤销一个事务先前对数据库所作的所有变更，(23.2）明确地描述异常中止操作所应做的工作，因此，在调度中，我们不必定义异常中止操作。当调度提交时，事务就异常中止，我们把这个过程称作**逻辑回退**（logical rollback)。

如果每个异常中止的事务对数据库没有产生任何影响，或者对事务的并发执行没有产生任何影响，则调度是可恢复的。更精确地说，调度S在满足下列条件时是可恢复的: 如果对S中每一个异常中止的事务T而言，S和一个删除所有T的操作的调度等价。

考虑一个并发控制，它以下列的方式来处理补偿操作:

1) 它不检查冲突就批准补偿操作[⊖]。

2) 只有当它们可以向后与已经批准为活动事务交换时，它才批准向前操作（没有考虑可能和先前已经批准活动事务的补偿操作冲突）。

我们必须证明，这一设计能正确地工作。这样，由控制产生的任意调度就是可恢复的，在恢复后，没有异常中止的事务调度结果是可串行化的。

让我们来看任意的调度S，它可能由这样的并发控制产生，涉及提交事务和异常中止事务（因而即包含向前操作，也包含补偿操作）。考虑S中的第一个异常中止事务，在事务执行时，设p_i^{-1}是第一个补偿操作。因为p_i^{-1}是S中的第一个补偿操作，所以S有下列形式:

$$S_{prefix},\ p_i,\ S',\ p_i^{-1},\ S_{suffix}$$

这里，S'是把p_i与p_i^{-1}分开的一个操作序列（注意，S'只包含向前操作，因为p_i^{-1}是S中的第一个补偿操作）。因为除非操作可向后与p_i交换，否则并发控制不会在S'中调度一个操作，所以这个调度是和下列调度等价的:

$$S_{prefix},\ S',\ p_i,\ p_i^{-1},\ S_{suffix}$$

这个调度又和调度

$$S_{prefix},\ S',\ S_{suffix}$$

等价。

这一变换使p_i和p_i^{-1}相互消除影响，因而使S和一个更加短的调度等价。

这一变换现在也能用来消除S中的第二个补偿操作和与之匹配的向前操作。这样进行下去，就能从S中消除异常中止事务的所有操作（向前的和补偿的操作），最终产生一个没有异常中止事务的调度。因此，S是可恢复的。而且，结果调度与等价于一个串行调度冲突。这种变换总是可能的事实就可以证明并发控制的正确性。

通过这样的变换，一个调度可以缩减成没有异常中止事务的可串行化调度，我们把这个调度称为是**可缩减的**（reducible）。并发执行的事务T_1，T_2，…，T_n是可恢复的，条件是调度（其中的每个事务T_j可用它的事务调度来表示）是可缩减的。本节所描述的所有由并发控制产生的调度都是可缩减的，因而也是可恢复的[⊜]。

23.7　结构化事务模型中的隔离

在第21章中介绍了一些事务模型。讨论过串行化和锁之后，现在我们来看如何在这些模型中实现隔离。我们将在第26章中讨论隔离在分布式事务中的实现，所以在这里我们不对这一内容进行讨论。

23.7.1　存储点

存储点（参见21.2.1节）是在事务T内部实现部分回退的机制，它使事务的部分操作对数据库所作的更新回退到更新之前。回退完成后，恢复的数据项可能立即被解锁，可供与其并

⊖ 如果由于冲突而延迟补偿操作，死锁可能导致不能实现异常中止，这是无法接受的。

⊜ [schek et al. 1993] 中有关于这一问题更加详细的讨论。

发的事务使用。因此在并发控制中，我们有下列基于锁的存储点规则。

1) 当创建存储点S时，对任何锁集没有影响。但并发控制系统必须记住事务T_i在创建S之前所有已获的锁的标识，如果T_i回退到S，它就能释放在创建S后所获得的锁。为实现上述目标，控制在锁列表L_i中作标记，其中包含一个代表存储点Id的数字。这个标记以后的锁项对应于在创建S之后获得的锁。

2) 当事务回退到S时，L_i中所有处在标记之后对应于锁项的锁被释放。

在两段锁并发控制中，如果上述规则能保持隔离性，那么是比较好的结果。事务在产生一个存储点后可能读一个数据项x，然后回退，释放这个读锁。结果是提前释放一个读锁，这很容易产生非两段调度。为保持隔离性，必须修改第二个规则，以便将写锁降级为读锁，读锁不被释放。

23.7.2 链式事务

链用来把一个事务T成分解成更小的子事务，从而在发生崩溃时，避免整个事务回退。在第21章，我们讨论两个不同的语义，以便说明当控制从链中的一个子事务移向另一个子事务时，链式事务是如何处理事务访问的数据库项的状态的。使用提交，不必保持两个子事务之间的状态，尽管单个子事务是隔离的和可串行化的，但事务T作为一个整体却不是。使用链，可以在一个子事务和下一个子事务之间维持状态，T相对其他事务而言是隔离的和可串行化的。

提交操作是按正常方式处理的，即释放所有的锁。对于链操作来说，锁不被释放，而是传给链中的下一个子事务。在这两种情况下都可以得到持久性（在第25章描述）。

23.7.3 可恢复队列

可恢复队列能在数据库内部实现（因为数据库是持久的），但性能因此受损。队列是一个热点，它被许多事务访问，因为并发控制是严格的，锁要一直保持到提交时刻，因而产生瓶颈。

由于这个原因，可恢复队列是作为一个模块来实现的。一种可能的实现方式是，给队列中的每一个元素、队首指针、队尾指针分别加锁。事务想将一个元素插入到队列或删除出队列时，它必须首先给队尾指针和队首指针分别施加“写锁”。给指针加锁只能保证入队或出队操作的持久性，而对入队或出队元素施加的锁会一直保持到提交时。因此，例如事务T可以从队列中去除一个元素，释放队首指针的锁。另一个事务可以在T提交或异常中止之前使下一个元素出队。

注意，因为队列是作为与数据库分开的一个模块来实现的，所以可消除严格性和两段锁要求的限制。显然，并发控制对指针的锁的操纵方式不是严格的，也不必是两段的。队列实质上是具有已知语义的对象。队列用于调度，所以定义在队列上的操作是可交换的——尽管实际并不是这样（例如，如果出队的操作顺序是相反的，那么将返回不同的信息给调用者）。结果，并发程度的加强是以牺牲隔离性为代价的。并发事务可能以不同的顺序让一个队列集合中的元素入队和出队，但在实现串行执行时是不能实现这一点的。

与在队首和队尾指针的锁对应，T总是对它所访问的元素施加“写锁”，直到提交为止。

例如，这能保证在T插入一个元素以后，直到T提交之前，没有其他的事务T'能使这个元素出队。

23.7.4 嵌套事务

嵌套事务支持子事务的并发执行，也就是说多个高级事务的子事务能并发执行，能请求冲突的数据库操作。因此，除给并发事务加锁的授权规则外，我们必须引入新的规则，用来控制对单个（嵌套）事务的子事务锁的授权。

在第21章讨论的嵌套事务模型遵守下列规则:

1) 作为整体的每个嵌套事务必须是隔离的，因而相对其他嵌套事务而言是可串行化的。

2) 子事务的父事务不与其子事务并发执行。

3) 每个子事务（及其所有后代）必须是隔离的，因而相对于每个兄弟事务（及其所有后代）是可串行化的。

为执行这个语义，我们必须施加以下规则[Beeri et al. 1989]:

1) 当嵌套事务的子事务T请求读一个数据项时，如果没有其他嵌套事务对这个数据有“写锁”，或者T中对那个数据有写锁的所有子事务是它的祖先（因而没有执行）时，就批准一个读锁。

2) 当嵌套事务的子事务T请求写一个数据时，如果没有其他嵌套事务对这个数据已有一个读锁或写锁，或者T中对这个数据有一个读锁或写锁的所有子事务是它的祖先（因而没有执行）时，则批准一个写锁。

3) 子事务所获得的锁一直保持到它提交或异常中止为止。当子事务提交时，这个子事务没有被它的父事务所持有的锁会由其父事务继承。当子事务异常中止时，这个子事务没有被它的父事务持有的锁就被释放。

因为以上规则是保证并发事务隔离性规则的一个超集，所以并发嵌套事务的调度是可串行化的。要了解这些规则如何加强兄弟之间所希望的语义，可以观察在同一子树下，活动兄弟事务的锁不会与另一活动兄弟事务的锁发生冲突。因此，对子树内的并发活动兄弟事务而言，它们不是以在其子树内执行的数据库操作来排序的，兄弟事务相对于其他事务而言是隔离的，是以它们的提交顺序可串行化的。

23.7.5 多级事务*

多级事务的并发控制可以是常规控制和第23.4节与23.5节[Weikum]描述的严格两段锁并发控制的泛化。第一个泛化利用各个级别上的操作语义，第二个泛化依赖于多级的事实。

1. 操作语义和交换

在23.6节，我们讨论了对象上操作的交换性如何用于立即更新的悲观并发控制的冲突表设计，这种思想构成了多级模型的核心部分。

在一个支持多级事务的系统中，每一级有它自己的（交叉）调度表。例如，在图21-3中，L_2这一级上的调度Move(sec_1, sec_2)是序列TestInc(sec_2), Dec(sec_1)（提示: Move将选修一门课程的学生从一个班移到另一个班，TestInc是有条件地在一个班中增加学生，Dec减少一个班中注册的人数），一个更加有意义的调度涉及多个多级事务的并发执行。例如，L_2的调度

$$TestInc_1(sec_2),\ TestInc_2(sec_2),\ Dec_2(sec_1),\ Dec_1(sec_1) \tag{23.3}$$

涉及两个事务$Move_1(sec_1, sec_2)$和$Move_2(sec_1, sec_2)$的交叉执行，这两个事务都是将班级1中的一个学生移到班级2中去。这个调度的有趣之处在于，因为两个减操作可交换，所以（23.3）的调度等价于

$$TestInc_1(sec_2),\ Dec_1(sec_1),\ TestInc_2(sec_2),\ Dec_2(sec_1) \tag{23.4}$$

因而按$Move_1$，$Move_2$的顺序是可串行化的。注意，因为两个针对相同元组的TestInc操作不能交换，所以按$Move_2$，$Move_1$的顺序不可串行化（初始状态可能是这样的，在加的过程中，第一个成功，而第二个却失败，如果顺序颠倒，可能返回不同的结果给调用者）。

现在假设按在L_1上执行的操作检查调度（23.3），我们来看下面的序列:

$$Sel_1(t_2),\ Upd_1(t_2),\ Sel_2(t_2),\ Upd_2(t_2),\ Upd_2(t_1),\ Upd_1(t_1) \tag{23.5}$$

和我们在23.6节讨论的银行应用一样，因为在相同元组上的更新操作一般不能交换，所以这两种顺序的调度都是不可串行化的。而且，对并发控制而言，可用的语义越多，就能检测到更多并发。

2. L_2的并发控制

为利用语义，我们使用冲突表C_2为L_2构造并发控制，冲突表中行和列上的操作都是L_2级所支持的操作。例如，一个Move事务是应用级L_3上的程序，包含有对L_2级支持的TestInc和Dec的调用操作。收到这些调用后，L_2级上的并发控制使用冲突表C_2来决定是否服务，C_2如图23-17所示。为实现这一目标，它使用TestInc和Dec锁。因此，在Move调用$Dec(sec_2)$时，如果没有其他事务在sec_2上加TestInc锁，并发控制就批准在sec_2上加一个Dec锁。否则，Dec请求必须等待，一旦得到锁，L_2上的程序就得以执行，它通过调用L_1级上的Dec操作来实现。

	授权模式	
请求模式	TestInc	Dec
TestInc	×	×
Dec	×	

图23-17　L_2级上的并发控制调度TestInc和Dec操作的冲突表

现在让我们回到原来的例子。尽管（23.3）的调度按$Move_1$，$Move_2$的顺序是可串行化的，因为两个TestInc操作不能交换，所以对这两个事务加一个顺序。因为对23.4节所描述的常规的、悲观的并发控制而言，当在活动事务上强加一个顺序时，它们就不会批准一个请求。对L_2也不会产生（23.4）这样的调度：和常规的并发控制一样，它并不能找出所有的冲突可串行调度。但考虑下面事务$Move_1(sec_1, sec_2)$，$Move_2(sec_1, sec_3)$的交叉调度

$$TestInc_1(sec_2),\ TestInc_2(sec_3),\ Dec_2(sec_1),\ Dec_1(sec_1) \tag{23.6}$$

L_2级的并发控制允许这个交叉调度，因为两个增加操作是可交换的。当事务请求提交时，为获得更多的并发，在执行完减操作后，不是立即提交，$Move_2$可能在一段额外的时间内继续处于活动状态。通过找到可交换的减操作，L_2并发控制就能避免延迟Dec_1，一直到$Move_2$释放它的锁为止。

3. 多级并发控制

现在看来，获得最佳性能的策略是执行一个并发控制来最有效地利用语义，这一并发控

制处在层次结构中的最高层L_n（图21-3的L_2）。但我们忽略重要的一点。在23.4节，我们在讨论常规的悲观并发控制时，我们假设由并发控制调度的读写操作是完全有序的。因此，如果控制在一个读操作之后调度一个写操作，它假设在写操作开始前，读操作已经结束。利用多级并发控制，当操作冲突时，这是正确的，但操作不冲突时，却不一定正确。因此一个事务T执行TestInc(sec_1)时，若第二个事务请求同样的操作，那么它就等待，一直到T执行完为止，因为同一班上的两个TestInc操作冲突，因此这两个操作是完全有序的。

但考虑两个不冲突的操作，它们并发执行是并发控制所允许的。在图23-18中，L_2并发控制调度两个减操作并发执行，这两个操作由应用级调用，即运行在L_2上的实际事务程序（事务还可以调用其他操作，但在这里我们不考虑这种情况）。减操作是由L_2上的一个程序（子事务）来实现的，程序调用操作被L_1支持。在这种情况下，减法程序的每个实例只作一次这样的调用——Upd。L_1上每个Upd程序的调用都调用读和写操作，它们在L_0实现。

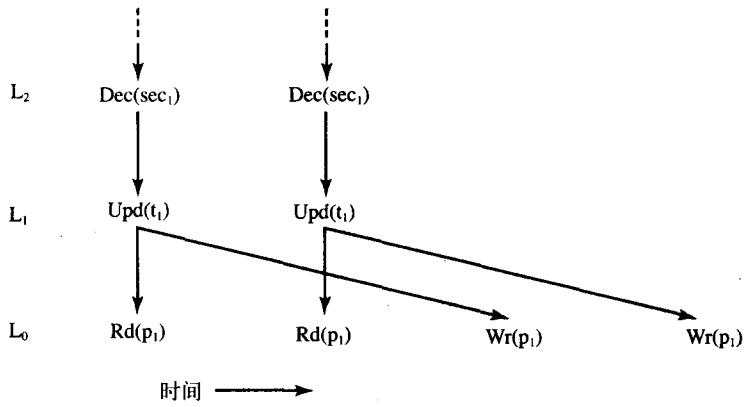


图23-18 在L_2上的减操作不冲突，但低级别上的任意的交叉可能导致问题

如图所示，每个更新语句读取存储在元组t_1中的选修人数的值，对值做减法，因此存回t_1的值也是相同的。尽管执行两次减操作，但选课人数只减一次，这是在2.3节介绍的丢失更新问题的例子。丢失更新的原因在于这两个Upd操作之间不是相互隔离的。

为避免发生这类问题，多级事务使用多级并发控制，它保证在每一级上的操作是可串行化的，因而是完全有序的。**多级并发控制**（multilevel concurrency control）由各级上的控制组成，因此，处在L_i的控制按照冲突表C_i调度来自L_{i+1}的子事务调用操作，冲突表决定这些操作是否可以并发执行。和前面所描述的一样，控制给每个操作加锁，当子事务在L_{i+1}级提交时就释放锁，图23-19就是这样的一个组织。

多级并发控制的目标就是要确保任意级别上的程序调用操作能有效地相互隔离。换句话说，在应用级上，事务调用的操作是由L_n中的一个程序来执行的，可以把这个程序看作一个子事务。这个子事务调用一系列的操作，每个操作由L_{n-1}上的程序来实现。如果L_n的并发控制决定op_1和op_2（由两个应用事务来调用）可以并发地执行，事务的调度由L_n上的子程序产生，这些调度在L_{n-1}上是交叉的。为保证op_1和op_2有效地隔离，L_{n-1}上由子事务产生的交叉调度必须是可串行化的，即它和单个序列的串行调度是等价的。注意，等价的串行顺序并不重要，因为op_1和op_2是可以交换的，因此这两种顺序产生相同的结果。L_n上的并发控制要求L_{n-1}上的

调度只和一些串行调度等价。

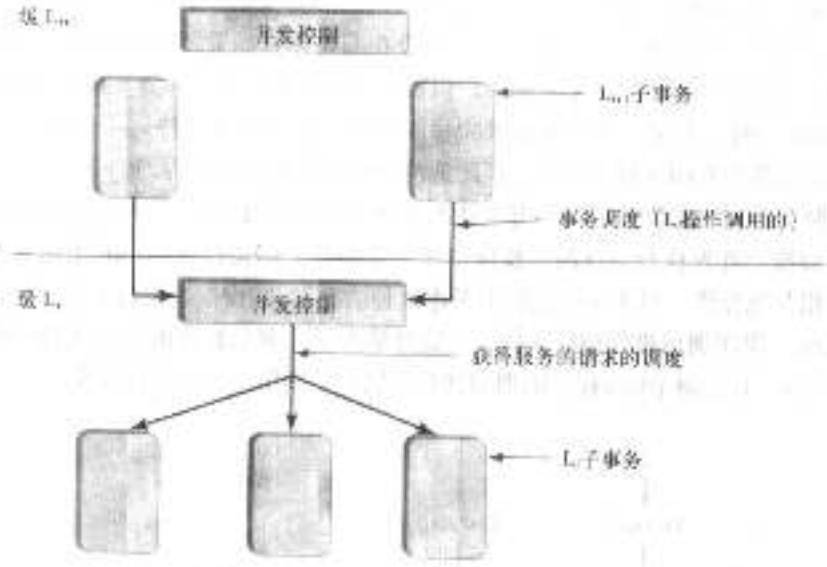


图23-19　多级并发控制中各级之间的联系

为达到这一目标，我们需要L_{n-1}上的并发控制保证这一级上的调度是可串行化的。L_{n-1}控制假设它调度的操作是隔离的单元，而事实却不是这样：它们是L_{n-2}上的子事务。因此，L_{n-2}上的并发控制有必要保证这些事务是可串行化的。

我们依此进行推理，把图23-18作为一个例子，来看为什么图中L_0上的调度不能由多级并发控制产生。应用级上的两个事务（不在图中）并发地调用Dec(sec)。L_2的控制看到这两个操作是可交换的，因此它批准这两个事务各自在sec_1上加Dec锁，允许L_2上实现Dec的子事务的两个调用并发执行。L_2上的每个子事务是调用Upd(t_1)的程序，这些调用被传给控制L_1。L_2的第一个子事务获得在t_1上的Upd锁，但第二个子事务却要等待，因为Upd锁冲突。因此，在L_0上，只有一个更新子事务的调用被启动，它调用p_1上的读写操作。这些被传给L_0上的控制，它批准更新子事务和调度操作在p_1上的锁。当更新子事务提交时，它释放页锁，并返回给调用它的Dec子事务。当Dec提交时，它释放Upd锁，因而允许更新子事务开始第二次调用，并返回给调用它的应用事务。这样，和图23-18相反，L_0上的调度是可串行化的。

使用和23.6节相同的推理，我们看到所有由这个并发控制产生的调度是可恢复的（即补偿操作能正确地工作，在事务异常中止时能保证原子性）。

23.8　其他的并发控制

在商业系统中，锁是大多数（但不是所有）并发控制算法的基础。在本节中，我们讨论两个不是锁的算法：时间戳顺序的并发控制和乐观并发控制。时间戳顺序算法，是一个我们在前面提到的算法，它说明使用时间戳来获得同步。乐观并发控制是一个新出现的方法，对一些特定的应用有前景。在第24章中，我们将讨论另外的算法——多版本的并发控制，它在商业关系数据库系统中实现。

23.8.1 时间戳顺序的并发控制

时间戳顺序并发控制（timestamp-ordered concurrency control）中，事务T在启动时被分配一个唯一的时间戳TS(T)，并发控制保证存在一个等价的串行调度，该调度中的事务是按时间戳排序的。由于这个原因，时间戳顺序控制是**静态的**（static），也就是说，等价的串行顺序在它们启动时就确定了。事务按它们的启动顺序串行化，而不必按提交的顺序串行化。用锁可以产生唯一的时间戳，事务启动时的时钟值作为时间戳。只要时钟的滴答比事务启动的速度快，那么每个事务就能获得一个唯一的时间戳[⊖]。我们描述一个立即更新的时间戳顺序控制，延迟更新也可能使用时间戳顺序控制。

时间戳顺序控制用一个数据项x来存储下列信息：

- $rt(x)$：任何读x的事务的最大时间戳。
- $wt(x)$：任何写x的事务的最大时间戳。
- $f(x)$：一个标志，它表明最后写x的事务是否提交。

保持这些信息意味着会有额外的开销，这是该策略的缺点。需要用额外的空间来将它们分别存在数据库中。而且，因为这些信息是存储在数据库中，必须像本地数据一样进行更新，即它们必须被存储在磁盘上。而且，如果事务异常中止，它们必须被回退。和其他的控制相反，这意味着对数据x的读就导致$rt(x)$的写。结果，时间戳顺序算法没有获得广泛使用。

当事务T_1请求读x时，并发控制执行以下动作：

(1) R_1

如果$TS(T_1)<wt(x)$，那么在等价的串行（时间戳）顺序中，必须在T_1之后的某些事务T_2（$TS(T_2)>TS(T_1)$）可以写一个新值到x中，T_1的读操作应在T_2写操作之前返回x的值，但这个值在数据库中已经不存在。因此，T_1太老（有一个太小的时间戳）不能读x，它就被异常中止并重启（有一个新的时间戳）。

(2) R_2

如果$TS(T_1)>wt(x)$，那么分两种情形：

- 如果$f(x)$表示x的值已经提交，请求就被批准，如果$TS(T_1)>rt(x)$，就把$TS(T_1)$的值赋给$rt(x)$。
- 如果$f(x)$表示x的值还没有提交，T_1必须等待（避免脏读）。

当一个事务T_1请求写x时，并发控制执行以下动作：

(1) W_1

如果$TS(T_1)<rt(x)$，在等价的串行（时间戳）顺序中，必须在T_1之后的某些事务T_2读x以前的值。若允许T_1提交，那么T_2也必须读T_1请求写的值。因此，T_1太老不能写x，它就异常中止并重启（有一个新的时间戳）。

(2) W_2

如果$rt(x)<TS(T_1)<wt(x)$，那么在按时间戳顺序的串行调度中，事务存新值到x中，以覆盖

⊖ 在网络中，每个站点使用它自己的本地时钟产生时间戳，这样的算法就不能保证时间戳的唯一性。为了保证时间戳的唯一性，必须对算法进行修改，给每个站点设置一个唯一的标识。站点把这个标识添加到时钟值来构成时间戳。因此站点i的时间戳为(c_i, id_i)，这里c_i是当前的时钟值，id_i是第i个唯一的标识符。

T_1请求写的值（因为$TS(T_1)<wt(x)$）。而且，没有时间戳处在$TS(T_1)$和$wt(x)$之间请求读x的事务（因为$rt(x)<TS(T_1)$）：

- 如果$f(x)$表示x的值已经提交，任何时间戳处于$TS(T_1)$和$wt(x)$之间的试图读x的后续事务都将异常中止（参见R_1）。因此，T_1请求写的值将不能被任何其他事务读，对数据库的最终状态也就没有影响。因此，请求被批准，但写实际上没有完成。这被称为Thomas写规则[Thomas 1979]（这种情况下不执行写操作）。
- 如果$f(x)$表示x不是一个已提交的值，T_1必须等待（因为最后写x的事务可能异常中止，T_1请求写的值变成当前的值）。

(3) W_3

如果$wt(x)$，$rt(x)<TS(T_1)$，那么分两种情形：

- 如果$f(x)$表示x不是一个已提交的值，那么T_1必须等待，因为批准请求会使回退变得复杂（参见23.2节的讨论）。
- 如果$f(x)$表示x是一个已提交的值，那么请求就被批准，将$TS(T_1)$的值赋给$wt(x)$，$f(x)$的值设成未提交（以后当T_1提交时，$f(x)$就设成已提交的状态）。

图23-20显示的请求序列说明了这些规则。假设$TS(T_1)<TS(T_2)$，在时刻t_0，x和y读写的时间戳都小于$TS(T_1)$，x和y都有已提交的值。然后在t_1时刻，应用规则R_2，批准读请求，把$rt(y)$设成$TS(T_1)$。在t_2时刻，使用W_3，批准写的请求，把$wt(y)$设成$TS(T_2)$，把$f(y)$设成表示y未提交。在t_3时刻，再次应用W_3，批准写的请求，把$wt(x)$设成$TS(T_2)$，因为T_2立即提交，所以$f(x)$和$f(y)$都表示已提交的值。在t_4时刻，应用W_2，因为$rt(x)$没有变化，$wt(x)$现在的值是$TS(T_2)$。请求被批准，但写实际上没有完成，$wt(x)$没有被更新。

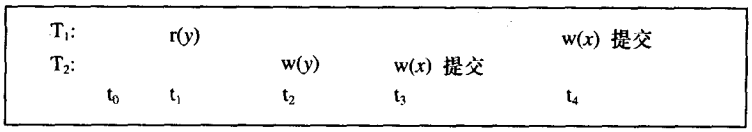


图23-20　时间戳顺序并发控制接受的请求的顺序。假设$TS(T_1)<TS(T_2)$，读写x和y的时间戳的初始值小于这两个事务的时间戳，T_1最终的写操作没有完成

因此，时间戳控制接受一个时间戳序列，但我们并不把它当作是一个调度，因为控制并没有将最终写的结果提交给数据库。注意，这个序列不冲突等价于一个串行调度，也不被一个基于冲突等价的并发控制所接受。这意味着时间戳顺序控制只接受不被基于两段锁的控制所接受的请求序列。练习23.38要求你给出一个被两段锁并发控制接受的调度，但它不被时间戳顺序控制所接受。这样，这两类控制是无法比较的，因为其中一类控制所接受的调度不被另一类控制所接受。

23.8.2　乐观的并发控制

通常，乐观算法由以下几个步骤组成。第一步，在一些简化任务性能的假设（乐观的）下执行一个任务。例如，在并发控制中，任务是一个事务，假设是不会出现冲突的并发事务。因此，我们不需考虑加锁和等待。事务的读和写不需要得到并发控制的许可，因此，与基于锁的（悲观的）算法相反，它不会延迟。第二步是通过检查假设是否为真来对第一步进行验

证。如果假设不为真，任务必须重做，这是并发控制中回退的情形。如果假设为真，验证结果导致结果提交。

这个方法和悲观的方法相反，在悲观的方法中，任务的执行是谨慎的，它并不在第一步中简化假设，每一个对数据库的访问都事先进行检查，如果检测到冲突，就立即采取适当的措施。因此在悲观的方法中，没有第二个步骤（验证）。

因为第一步中没有对数据库的访问进行检查，所以实际上是可能发生冲突的，乐观并发控制[Kung and Robinson 1981]通常使用延迟更新的方法来防止不正确事务执行的结果在数据库中传播（如果使用一个立即更新的方法，将在以后回退的事务的更新操作对与其并发执行的事务来说是可见的，这会导致一个级联异常中止），写入的新值存储在一个意向表中，但并不用来对数据库作立即更新。因此，修改数据库的事务需要第三步，并且验证意向表是否写入数据库中。

因为回退是有代价的，所以只有在乐观假设有效的情况下，乐观算法才是合适的。注意，在乐观算法中，回退的代价比时间戳顺序算法的回退代价更高，因为在事务完成后，要做出是否回退的决定。对时间戳顺序控制而言，回退决定是在事务还在执行时做出的，它耗费较少的系统资源。由于死锁，回退在悲观算法中还是可能发生的。乐观算法的一个重要优点是不会发生死锁，因为一个事务不会等待另一个事务。比较乐观算法和悲观算法的效率，相对锁的代价，验证和意向表的管理必须加权。

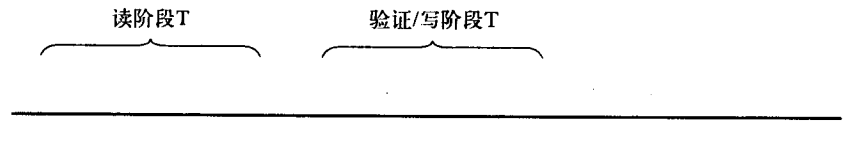
因为数据库在第一步中不会被修改（可能请求写，但实际上没有执行），这一步被称作**读阶段**（read phase），第二步被称为**验证阶段**（validation phase），第三步被称作**写阶段**（write phase）。因此，事务的写是在写阶段完成的（在一个调度中出现）。为简单起见，我们起初假设验证和写阶段是关键部分，因此一个时刻，只有一个事务处在验证阶段或写阶段。但我们稍后将修改这个假设，一个事务的三个阶段如图23-21a所示。

验证阶段确保由乐观并发控制产生的任意调度S是和调度S^{ser}等价的。在S^{ser}中，已提交事务是按提交的顺序执行的，因此它们的操作先于那些未提交事务的操作。

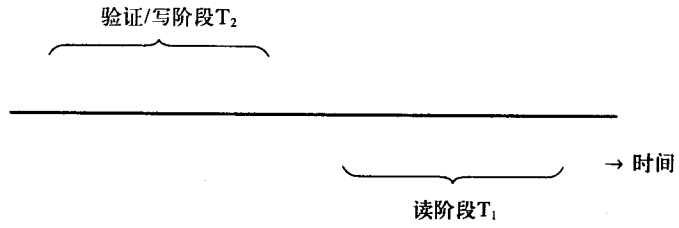
如果事务T_1在读阶段执行的操作（回忆一下，在数据库的读阶段是没有写操作的）不与其他事务的操作相冲突，这些事务在T_1的读阶段处于活动状态，在T_1进入验证阶段之前提交，那么T_1就成功通过验证，并提交。如果T_2在T_1进入读阶段之前完成它的写阶段，如图23-21b所示，那么这两个事务不会并发地处在活动状态。这种情形之下，不论是在调度S还是在S^{ser}中，T_1的所有操作都在T_2之后，且不会发生冲突。但假设T_2在T_1进入验证阶段之前完成，在T_1开始它的读阶段之前，T_2还没有完成它的写阶段，如图23-21c所示。而且，假设T_1读T_2写的数据，T_1的读可能先于T_2的写，在这种情况下，冲突操作的顺序是与提交顺序T_2T_1相反的。因此，不允许T_1成功通过验证。由于这个原因，一些并发事务的读写操作在T_1处在读阶段时提交，事务与这样一些事务的冲突验证测试如下：T_1读的数据集不与任何事务写的数据集相交，这些事务的写阶段与T_1的读阶段相重叠。

注意，当T_1验证时，唯一的检查只考虑T_1的读操作，但不包括T_1写操作的冲突检查。尽管T_1写一个T_2读或写的记录可能发冲突，但在一个时刻只有一个事务处于验证阶段或写阶段的

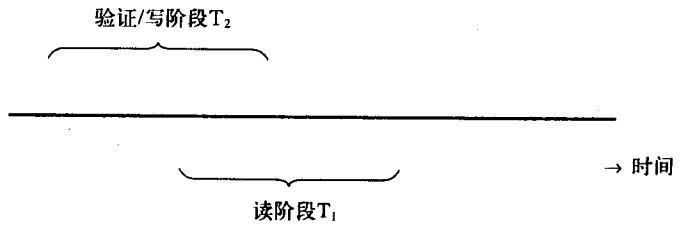
事实保证T_1的写（在它的写阶段）是在T_2的读（在读期间）或写（在写期间）之后。因此，假设T_1提交，则操作以提交的顺序执行（即在S和S^{ser}中，它们的顺序是一样的）。所以，即使T_2的读或写与T_1的写发生冲突，T_1也能成功地通过验证，因为构成S^{ser}的操作不必交换（也可以用另一种方法：当T_2不再处于活动状态时，T_1的写已经完成，因此不要考虑冲突）。



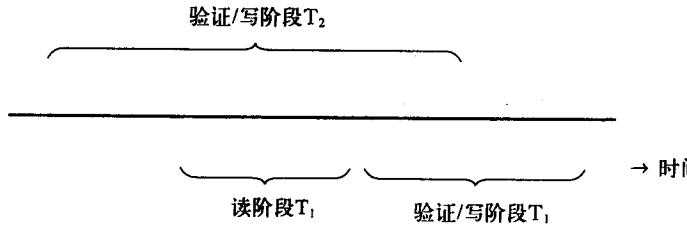
a) 一个事务的三个阶段



b) 非并发的事务



c) 如果T_1读取的数据集与T_2写的数据集重叠，那么T_1与T_2冲突



d) 如果T_1读或写的数据集与T_2写的数据集重叠，那么T_1与T_2冲突

图23-21　在一个乐观并发控制中的事务

我们已经假设，在任何时刻，只有一个事务处在验证阶段和写阶段（这被称为**串行验证**（serial validation））。因此，等价的事务提交顺序就是事务进入验证阶段的顺序（即提交顺序），没有两个事务同时处于写阶段。尽管这一假设简化了验证，但它产生了瓶颈，从而限制了并发。

另一个办法是**并行验证**（parallel validation），它可以通过允许事务并发地执行它们的验证阶段或写阶段来避免瓶颈。利用串行验证，事务进入验证阶段的顺序是等价的串行顺序。所以，除必须满足顺序验证的条件外，一个事务T_1在进入它的验证阶段时，必须阻止其他事务T_2在同一时刻也进入验证阶段或写阶段（因为这些事务在等价的串行序列中是先于T_1的）。对这样的事务集，必须考虑它们之间的写操作冲突（发生在并发的写阶段）。在T_1提交之前，必须满足两个条件。第一个条件与串行验证的相同：如果T_1的读阶段与T_2的验证阶段或写阶段重叠，那么T_1读的数据集必须与T_2写的数据集不相交。另外，如果T_2在T_1之前进入验证阶段或写阶段，这些阶段是重叠的，如图23-21d所示。为保证T_1能成功地通过验证，T_1所写的数据集必须与T_2写的数据集不相交。

23.9 参考书目

介绍并发控制的两本非常好的书是[Bernstein et al. 1987,Papadimitriou 1986]。更多的理论上的研究在[Lynch et al. 1994]中给出。

串行化的概念和两段锁在[Eswaran et al. 1976]中有所介绍。如果调度不是可串行化的，则存在一个完整性约束，调度违反这个完整性约束，这在[Rosenkrantz et al. 1984]中给出证明。可恢复性和严格性是在[Hadzilacos 1983]中介绍的。[Weihl 1988]介绍了向前和向后交换，在[Lynch et al. 1994]有更加详细的讨论。本书关于嵌套事务执行的介绍取自于[Beeri et al. 1989]，多级事务的介绍来自[Weikum 1991]。时间戳顺序并发控制最先是在[Thomas 1979]中描述的，这本书还介绍了Thomas写规则。[Kung and Robinson 1981]介绍了乐观并发控制。[Lampson et al. 1981]首次提出了意向表。基于应用的语义（而不是操作语义）设计并发控制的其他方法在[Bernstein et al. 1999b,Bernstein et al. 1999a,Bernstein et al. 1998,Bernstein and Lewis 1996]中进行了讨论。

23.10 练习

23.1 试述下列哪些调度是可串行化的

a. $r_1(x)r_2(y)r_1(z)r_3(z)r_2(x)r_1(y)$

b. $r_1(x)w_2(y)r_1(z)r_3(z)w_2(x)r_1(y)$

c. $r_1(x)w_2(y)r_1(z)r_3(z)w_1(x)r_2(y)$

d. $r_1(x)r_2(y)r_1(z)r_3(z)w_1(x)w_2(y)$

e. $r_1(x)r_2(y)w_2(x)w_3(x)w_3(y)r_1(y)$

f. $w_1(x)r_2(y)r_1(z)r_3(z)r_1(x)w_2(y)$

g. $r_1(z)w_2(x)r_2(z)r_2(y)w_1(x)w_3(z)w_1(y)r_3(x)$

23.2 给出对应于23-5所示的串行图的所有可能的冲突等价串行调度。

23.3 使用一个串行图来阐明图23-4所示的调度不是冲突可串行化的。

23.4 假设我们在数据库模式上定义所有的数据库完整性约束，以便当事务的更新违反这些完整性约束时，数据库管理系统不允许事务提交。而且，即使我们不使用任何并发控制，数据库仍然保持一致性，试解释我们为什么还是要使用并发控制。

23.5 试举出一个包含两个事务的调度的例子，它保持数据库的一致性（数据库满足它所有的完整性约束），但数据库最终没有反映出这两个事务的影响。

23.6 试证明冲突等价包含视图等价。

23.7 试举出一个调度的例子，其中学生注册系统中的事务出现死锁。

23.8 利用两段锁协议，试证明一个可能的等价串行顺序是这些事务第一次释放锁的顺序。

23.9 试举出一个你所见到的不同于银行处理系统事务处理系统的例子，在这个系统中，串行顺序就是事务提交的顺序。

23.10 试举出一个调度的例子，该调度是可串行化的，但不是严格的。

23.11 试举出一个调度的例子，该调度是严格的，但不是可串行化的。

23.12 试举出一个调度的例子，该调度是由非严格两段锁并发控制所产生的，但不是可恢复的。

23.13 试举出一个由可恢复的但非严格的并发控制所产生的调度，它涉及三个事务，当死锁发生时，导致所有三个事务级联异常中止。

23.14 试举出一个调度的例子，它由非严格的两段锁并发控制产生，是可串行化的，但不是以提交的顺序可串行化的。

23.15 假设account对象（它的冲突表如图23-16所示）有一个额外的操作balance，该操作返回账户的余额。为这个对象设计一个包含新操作的新冲突表。

23.16 考虑下面这个包含三个事务的调度

$$r_2(y)r_1(x)r_3(y)r_2(x)w_2(y)w_1(x)r_3(x)$$

a. 定义T_1和T_2之间的串行序列。

b. 按照什么顺序T_3能看见数据库的内容。

c. 调度是可串行化的吗？

d. 假设每个事务都是一致的，数据库的最终状态满足所有的完整性约束吗？

e. T_3所看到的数据库状态满足完整性约束吗？请说明原因。

23.17 假设除read(x)和write(x)操作外，数据库还有copy(x, y)操作，该操作自动地将存储在x中的值复制到y中去。为使用在立即更新的悲观并发控制中使用的这些操作设计一个冲突表。

23.18 假设有一个queue对象，有enqueue和dequeue操作，如果队列中没有数据项，dequeue返回NO。而enqueue总是成功完成。使用部分操作和向后交换为这个对象设计一个冲突表。

23.19 设一对数据库操作（部分）p和q。在定义p和q的每个数据库状态中，如果定义序列p,q和q,p，而且序列p和q都返回相同的值，数据库的最终状态也是相同的，这时我们说操作p和q是可**向前交换的**（forward-commute）。

a. 描述在设计延迟更新的悲观并发控制时，向前交换是如何实现的？

b. 对图23-16中描述的account对象，为这样的控制设计一个冲突表。

23.20 设计一个延迟更新的悲观并发控制。

23.21 在实现链事务时，链接下一个子事务的子事务释放它的某些锁，这些锁对后续的子事务是不必要的，解释这怎么能影响到链式事务结果的ACID性质。

23.22 我们希望确保使用可恢复队列的事务集是隔离的。试解释队列的实现是如何在队列指针和队列中的数据项上加锁的。

23.23 试给出系统中一个不可串行化的调度，该系统使用可恢复队列，其中在enqueue和dequeue操作完成之后，事务提交之前，加在队首指针和队尾指针上的锁都被释放。

23.24 假设ATM系统上的取款事务是包含两个子事务的嵌套事务，其中一个子事务验证用户提供的PIN码，与所给账户相关，另一个子事务完成从账户余额中取款。假设两个这样的事务试图并发地从同一账户提取现金。试解释为什么这样的调度是可能的，调度中先执行两个子事务的PIN码验证，然后执行两个取款子事务。

23.25 试给出这样一个调度，在悲观并发控制中，它先让一个事务等待，但之后又允许这个事务提交，而在乐观并发控制中，它却重启事务。

23.26 试给出这样一个调度，在悲观并发控制中，它先让一个事务等待，但之后又允许这个事务提交，而在乐观并发控制中，它却允许事务不必等待就提交。

23.27 试给出这样一个调度（不会因为锁而延迟），它在立即更新的、悲观的、严格的两段锁并发控制中是可以接受的，但乐观并发控制却重启其中的一个事务。

23.28 死锁可能在本章中描述的时间戳顺序控制中发生吗？

23.29 试给出一个由时间戳顺序并发控制所产生的调度例子，其中串行顺序不是提交的顺序。

23.30 试给出一个严格的、可串行化调度的例子，但不是按提交顺序，它既可以由时间戳顺序并发控制产生，也可以由两段锁并发控制产生。

23.31 试述下面提出的协议对时间戳顺序并发控制来说是不可恢复的。

为每个被事务（不必是已提交的）读过的数据项保存最大的时间戳，为每个被事务（不必是已提交的）写过的数据项保存最大的时间戳。

当一个事务请求读（写）一个数据项时，如果请求事务的时间戳比所要写（读）的数据项的时间戳小，就重启事务，否则批准请求。

23.32 Kill-Wait并发控制结合了立即更新并发控制和时间戳顺序控制的概念，在时间戳顺序系统中，当一个事务T_i启动时，给它分配一个时间戳$TS(T_i)$。但系统使用与立即更新悲观并发控制相同的冲突表，解决冲突使用的规则如下：

如果事务T_i发出一个请求，它与活动事务T_2的操作冲突，若$TS(T_i)<TS(T_2)$，就异常中止T_2，否则使T_i等待，直到T_2终止。

异常中止T_2被称作是kill，因为T_i杀死了T_2。

a. 试证明Kill-Wait按提交顺序串行化。

b. 试给出一个由Kill-Wait控制产生的调度，它不是按时间戳顺序可串行化的。

c. 试解释为什么在Kill-Wait控制中不会发生死锁。

23.33 Wait-Die并发控制是另一种控制，它结合立即更新并发控制和时间戳顺序控制：

如果事务T_i的请求与活动事务T_2的操作冲突，若$TS(T_i)<TS(T_2)$，则让T_i等待直到T_2终止，否则异常中止T_i。

这里异常中止T_i被称作是die，因为T_i杀死了它自己。

a. 试证明Wait-Die控制是按提交顺序串行化的，并且能防止死锁的发生。

b. 试比较执行Kill-Wait和Wait-Die控制的优点。

23.34 为并行验证乐观并发控制给出一个完整的算法描述。

23.35 比较事务处理系统的两个模型：

a. 嵌套模型，和书中描述的类似，只是不允许子事务的并发执行。在某一时刻，父事务只有唯一的一个子事务在执行。

b. 常规事务模型，当嵌套模型中的父事务调用孩子事务时，常规模型则调用一个子例程，嵌套事务模型中的子事务异常中止时，常规模型中相应的子例程手工撤销任何数据库的更新，并返回一个失败的状态。

解释为什么嵌套模型更加有效，它能允许事务更多的吞吐量。

23.36 假设事务T_1和T_2能分解成下面的子事务，T_1：$T_{1,1}$，$T_{1,2}$和T_2：$T_{2,1}$，$T_{2,2}$，这样$T_{1,1}$和$T_{2,1}$访问的数据项与$T_{1,2}$和$T_{2,2}$访问的数据项不相交。不要求保证所有涉及T_1和T_2的调度是可串行化的，假设某个

并发控制可以保证$T_{1,1}$总是与$T_{2,1}$串行执行，$T_{1,2}$总是与$T_{2,2}$串行执行。

a. T_1总是和T_2串行执行吗？请说明原因。

b. 为保证T_1和T_2并发执行的结果与串行调度的执行结果一样，子事务上最低的限制条件是什么？

c. 假设保持b的限制条件，在保证可串行化调度方面，和并发控制相比，新的并发控制有什么优点？

23.37 假设事务T_1和T_2能分解成下面的子事务，T_1：$T_{1,1}$，$T_{1,2}$和T_2：$T_{2,1}$，$T_{2,2}$，这样每一个子事务都保持数据库的一致性约束。不保证所有涉及T_1和T_2的调度是可串行化的，假设并发控制保证所有的子事务总是可串行化执行的。

a. T_1总是和T_2串行执行的吗？请说明原因。

b. 所有可能的调度都将保持完整性约束吗？

c. 如果并发控制以这样的方式调度事务，可能出现什么样的问题？

23.38 试举出一个调度的例子，它被两段锁并发控制支持，但不被时间戳顺序并发控制所支持。

第24章 关系数据库中的隔离性

24.1 加锁

迄今为止，在有关隔离性的讨论中，我们总是假设事务是通过名字显式地访问数据项的。在关系型环境里，事务所访问与描述的都是元组；它要求利用该元组所满足的条件，而不是通过给出其名字，来隐式地访问元组。因此，SELECT语句读取的元组集合满足WHERE子句中的选择条件[⊖]。

在某银行系统中，考虑对于每个不同账户都包含有一个元组的表ACCOUNTS。我们可以利用下述SELECT语句来读取描述由储户Mary控制的ACCOUNTS中的所有元组：

```
SELECT *
FROM ACCOUNTS A
WHERE A.Name = 'Mary'
```

上述WHERE子句中的表达式称为**读谓词**（read predicate），FROM子句中命名的表的属性作为该谓词的变量），该语句返回（读取）满足该读谓词的所有元组。

这类操作的其他形式将会产生矛盾。例如，假设表ACCOUNTS具有属性AcctNumber（键）、Name和Balance。同时假设存在表DEPOSITORS，对于每个储户都包含一个元组，它具有属性Name（键）和TotalBalance，其中属性TotalBalance的值是该储户所有账户的余额之总和。为Mary执行的某个审计事务T_1可能需要执行下述SELECT语句：

```
SELECT SUM(Balance)
FROM ACCOUNTS A
WHERE A.Name = 'Mary'
```

并将其同执行下列语句得到的结果值进行比较：

```
SELECT D.TotalBalance
FROM DEPOSITORS D
WHERE D.Name = 'Mary'
```

最后再检查TotalBalance的值是否等于其总和。

同时，Mary的某个新的账户事务T_2可能会通过下述语句对表ACCOUNTS追加某个新的元组：

```
INSERT INTO ACCOUNTS
VALUES ('10021', 'Mary', 100)
```

然后利用下述语句将表DEPOSITORS中相应的元组的TotalBalance增加100。

⊖ 对于事务是通过其主键（例如关系STUDENT里的Id）访问元组的特殊情况，我们可以将此主键的值称为该元组的名字，并且可以考虑在该名字的基础上对元组加锁。然而，正如我们将会看到的，即便在这样的特殊场合，也还是需要应用附加条件的。

```
UPDATE DEPOSITORS
SET TotalBalance = TotalBalance + 100
WHERE Name = 'Mary'
```

表ACCOUNTS上由T_1和T_2所完成的操作是相互冲突的，因为T_2的INSERT和T_1的SELECT是不可交换的。倘若INSERT是在SELECT之前执行的，那么插入后的元组将由SELECT语句返回；反之，则不会被返回。因此，倘若在T_1读取表ACCOUNTS和T_2读取表DEPOSITORS之间，允许T_2交错执行，那么就会得到无效结果。第2个语句里由T_1所读取的TotalBalance之值就不会等于它在第1个语句里读取到的Balance的总和。（表DEPOSITORS上的操作也有类似的冲突。）

24.1.1 幻影

就像使用非关系数据库一样，通过应用某个加锁算法，我们可以确保实现可串行化。在设计这类算法时，我们首先必须判断对什么加锁。一种做法是对表加锁。表都有名字，并在访问表的读写语句中使用这些名字。SELECT语句可以处理成对FROM子句中命名的数据项（表）的读取，而DELETE、INSERT和UPDATE语句也可以处理成对已命名的表的写。因此，在第23章里所描述的并发性控制算法就可以用来实现可串行化调度。就像采用页加锁一样，可以采用表加锁。这种方法的问题是，加锁粒度太粗。一张表可能会包含成千上万（甚至几百万）个元组。仅仅因为要访问其中某一个元组，就对整个表加锁，这样将会导致严重的并发性损失。

如果不对表加锁，而是对每个元组分别加锁，那么锁的粒度是细了，但其最终的调度可能不是可串行化的。例如，假设T_1对ACCOUNTS已读取的所有元组（满足谓词Name = 'Mary'的元组）加锁。而一个事务在某张表中插入新元组的能力是不会受到其他事务对该表的现有元组所加之锁的影响的。因此，T_2可以构造一个元组t，它满足该谓词，而且描述了Mary的一个新账户，并将它插入ACCOUNTS中。这样一来，就有可能会产生下述调度：

T_1加锁并读取ACCOUNTS中描述Mary的账户的所有元组。

T_2向ACCOUNTS追加t，并对其加锁。

T_2对Mary在DEPOSITORS中的所有元组加锁并更新。

T_2提交并释放其拥有的全部锁。

T_1对DEPOSITORS中的Mary的元组加锁并读取。

其中，由于追加了t，因此T_2已改变了谓词Name = 'Mary'所引用的元组集的内容。在这种情况下，t称为**幻影**（phantom），因为T_1认为它锁住了满足该谓词的所有元组，可是T_1不知道并发事务插入了一个满足该谓词的新元组t。幻影可能会导致非可串行化的调度，因此会产生不确定的结果。在本例中，T_1在其第2个语句里读取到的TotalBalance的值不等于其在第1个语句里读到的元组的Balance属性值的总和。

出现问题的原因是SELECT语句并不命名特定的数据项。相反，它指定一个可能由一些元组满足的条件或谓词，这些元组有的也许就在某个特定表里，而有的可能不在。然而，我们只能对已经在某个表里的元组设置锁，却难以对不在表里的那些元组设置锁。为了消除产生幻影的可能性，我们需要有一种加锁机制，得以防止将那些满足谓词、但当前并不在表里的元组（即幻影）添加到表里来。

虽然我们以SELECT语句为例说明幻影问题，但只要是对数据库更新的语句，都可能出现这个问题。例如，对满足谓词P的所有元组进行更新的UPDATE语句，不能和向该表里插入满足P的一个元组的某个INSERT语句交换。遗憾的是，即使更新事务要求对所有被更新的元组加锁，仍然可能有某个并发事务对其进行插入。

防止幻影的一种办法是对整张表加锁，这肯定可以防止包括幻影在内的任何新的元组被插入。然而，正如在24.3.1节将会看到的，由于存在可以防止幻影、但不要求对整张表进行加锁的协议，因此对表加锁并非总是必要的。所以，当许多商用DBMS使用术语“元组加锁”（或“页加锁”）来描述并发性控制算法时，意味着它们采用元组锁（或页面锁）来作为这类协议的一部分，从而保证可串行性。不过万事小心为妙，“当一切努力均告失败时，请阅读用户手册”。但在一些场合下，商用DBMS还是会锁住整张表，以防止幻影并实现可串行化。

24.1.2 谓词加锁

处理幻影的一种技术是**谓词加锁**（predicate locking）[Eswaran et al. 1976]。一个谓词P指定了元组的一个集合。当且仅当某元组的属性值使P为真时，该元组属于该集合。例如，Name = 'Mary'是一个谓词，它指定ACCOUNTS里存在的、其Name属性具有值Mary的所有可能元组的集合。这个集合是所有曾经存放在ACCOUNTS里的元组集D的一个子集。因此，P指定D的一个子集，其中有些元素可能在ACCOUNTS里，而有些则未必在ACCOUNTS里。

1. 和SQL语句有关的谓词

每个SQL语句都有一个相关的谓词。与SELECT或DELETE语句相关的谓词是在其WHERE子句里指定的。在最简单情况下，该子句描述了对FROM子句中命名的表的属性的约束，而该谓词则能够与此表相关联。然而，事情有时候会变得更为复杂，比如，WHERE子句中包含某个嵌套的SELECT，或者命名多张表的FROM子句时。在这种情况下，就会涉及多张表，每张表都有一个相关的谓词。尽管可以将这种情况描述得更具一般性，但是我们并不打算将讨论搞得复杂化，因为我们当前的主要目的是描述谓词加锁的概念。

与INSERT语句相关的谓词描述了打算插入的元组集。最简单的情况就是插入单个元组，谓词是

$$(A_1 = v_1) \wedge (A_2 = v_2) \wedge \cdots \wedge (A_n = v_n)$$

其中A_i是第i个属性的名字，而v_i则是打算插入的元里该属性的值。该谓词指定了已被插入的元组的集合。更一般地，INSERT可以包含某个嵌套的SELECT语句

```
INSERT  INTO TABLE1(···属性列表···)
SELECT ···属性列表···
FROM TABLE2
WHERE P
```

这时，谓词P既与TABLE2相关（因为满足P的元组是从该表读取的），又与TABLE1相关（因为满足P的元组要写入此表）。

一个UPDATE语句可以看成是一个DELETE语句，再跟随一个INSERT语句，因此，它具有两个相关谓词。第1个是UPDATE语句里WHERE子句中的谓词P，它指定了需要删除的元组。SET子句描述了如何修改这些元组。然后，再插入由某个谓词P'所描述的最终的元组集。例如，

下述UPDATE语句对Mary的所有账户加入利息：

```
UPDATE Accounts
SET Balance = Balance * 1.05
WHERE Name = 'Mary'
```

这种情况下，要删除的元组和被插入的元组满足相同的谓词——Name = 'Mary'，然而，若Mary想要在其名字里加上她的姓，我们可以执行以下的语句：

```
UPDATE Accounts
SET Name = 'Mary S'
WHERE Name = 'Mary'
```

现在，P是Name = 'Mary'，而P'则是Name = 'Mary S'。

2. 谓词锁

谓词锁（predicate lock）是加在谓词P上与表R相关的锁，它锁定由P所指定的所有元组，无论它们是否在表R内。如图24-1所示，SELECT语句在FROM子句中指定表R，并在WHERE子句中指定谓词P，表R内所有的元组，凡满足P者，皆由该SELECT语句返回，但是，D中的所有元组，凡满足P者都被锁定。

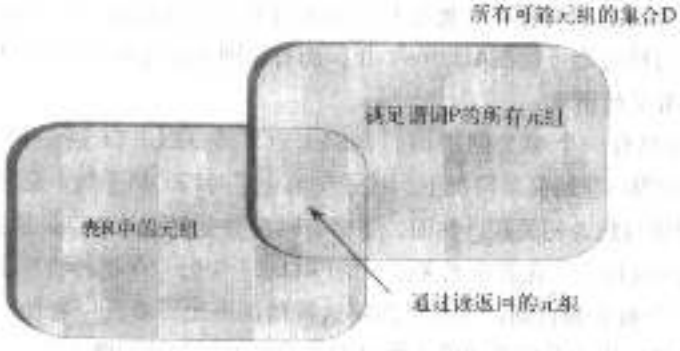


图24-1　谓词P指定了D的一个子集，该子集中的元组有些在R里，有些可能不在

类似地，对于DELETE语句，D中的所有元组，凡满足P者皆被锁定，而R中满足P的元组则被删除。在使用INSERT语句的情况下，锁住的是所插入的元组。例如，与所插入元组t（它描述Mary在Accounts里的新账户）相关的谓词锁恰好锁住元组t。

$$P_t:\{AcctNumber = '10021'\} \wedge \{Name = 'Mary'\} \wedge \{Balance = 100\} \tag{24.1}$$

使用UPDATE语句，P和P'两者都必须加锁。

现在我们来解释一下更一般的冲突的概念。取代冲突操作命名着相同的数据项之要求，现在我们规定如下：

> 两个操作冲突的条件如下：其中至少有一个是写操作，并且与这两个操作相关的谓词所描述的元组集具有非空的交集。

例如，返回所有满足谓词Name = 'Mary'的元组的SELECT(read)语句与DELETE(write)语句冲突：

```
DELETE
FROM Accounts
WHERE Balance < 1
```

后者删除所有满足谓词Balance < 1的账户。因为D中存在同时满足两个谓词的元组，例如，满足下述要求的元组:

```
AcctNumber = '10000' ∧ Name = 'Mary' ∧ Balance = .5
```

这些元组有可能在ACCOUNTS里，因此，我们无法确定该SELECT语句与DELETE语句有无交集。另一方面，该SELECT语句与下述语句并不冲突:

```
DELETE
FROM ACCOUNTS
WHERE Name = 'John'
```

因为与这两个语句相关的谓词的交集为空集。最后，下述语句

```
SELECT *
FROM ACCOUNTS
WHERE Name = 'Mary S'
```

与向其在ACCOUNTS里的Name属性添加Mary的姓的UPDATE语句相冲突。

前面我们已经讲过，谓词加锁能解决幻影问题。当T_1读取ACCOUNTS时，它对于谓词Name = 'Mary'施加一个读取锁。后来，当T_2试图向ACCOUNTS插入描述Mary新账户的元组t时，它将请求对于谓词P_t(24.1)施加某个写锁。由于这两个谓词的交集非空，因此存在冲突，因而该写锁无法获得授权。元组t是一个幻影（它在ACCOUNTS里并不存在），对P_t施加谓词锁防止了出现并发性的追加。

谓词加锁的一种实现方式如下：对于每张表R，都有一个相关的锁集L(R)，其中包括所有获得授权的当前活动事务已请求的相关锁。L(R)的每一个元素都对应一个谓词。当与某个数据库操作所请求的相关锁同L(R)的某一个元素起冲突时，并发控制就会让该请求事务等待。

利用谓词加锁，我们就能用比表加锁更为精细的加锁粒度来保障可串行调度，因为我们是对可能在R内的、而非整个表内的元组的集的子集加锁。遗憾的是，冲突的测试（谓词交集）实现起来比较费事，它限制了谓词加锁的实用性。由于这个原因，商用的DBMS一般并不实现谓词锁。

24.2　加锁与SQL隔离级别

大多数商用DBMS通过对整个表加锁，或者利用与索引加锁有关的粒度更细的锁（我们稍后将会描述）来保障可串行性。遗憾的是，对于某些应用来说，更富有刺激性的是，比实现可串行性，更快地释放锁，对此，以上两种处理都不太适宜。

如果采用较弱的隔离级别[Gray et al. 1976]，较快地释放锁是有可能的。正如我们在10.2.3节看到的，SQL标准定义了四种隔离级别，每个事务可以选取四个级别中的一个。按照强度递降的顺序，它们是:

- SERIALIZABLE (可串行级)
- REPEATABLE READ (可重复读取级)
- READ COMMITTED (读取已提交级)
- READ UNCOMMITTED (读取未提交级)

SERIALIZABLE级是与本书讨论的可串行化执行的概念相对应的，它是唯一能够保证所

有应用的正确性的一种隔离级别。利用较弱的隔离级别来实现性能上的提高，是以不正确执行为代价的。

给定的某个DBMS未必支持所有的隔离级别。通常是提供某个特定级别作为默认值，并有请求获得其他支持的级别的机制。

SQL标准[SQL 1992]根据每个级别应防止的一定现象（有时称为异常）来指定隔离级别，但SERIALIZABLE级是例外。某个现象若是某一级别所防止的，那么它也是更高级别所防止的。

- 在READ UNCOMMITTED级，脏读是有可能的（参见23.2节）。
- 在READ COMMITTED级，脏读是禁止的，但特定事务对同一个元组连续读取有可能会产生不同的值。
- 在REPEATABLE READ级，通过特定的事务对同一个元组连续读取不可能会产生不同的值，但是有可能会出现幻影。
- 在SERIALIZABLE级，禁止出现幻影。事务执行必定是可串行化的。

图24-2里汇总了允许及不允许的各种现象。

隔离级别	脏读	不可重复读	幻影
READ UNCOMMITTED级别	Yes	Yes	Yes
READ COMMITTED级别	No	Yes	Yes
REPEATABLE READ级别	No	No	Yes
SERIALIZABLE级别	No	No	No

图24-2　每种隔离级别允许与不允许的现象

SQL隔离级别并不处理脏写，正如我们在23.2节所看到的，出于众多原因，脏写是我们不希望遇到的。尽管在所有事务均在SERIALIZABLE级运行的情况下，脏写是不可能出现的，然而SQL标准却并没有在较低隔离级别中明文加以禁止（当然，在某个特定的实现里不允许它们出现是有可能的）。请注意，图23-9里的调度解释了某个脏写入，它并不涉及脏读或不可重复读，因此它没有违反三个较低隔离级别的任何一个的要求。由于级别是根据某些不希望出现的现象来定义的，那么人们也许会想，若脏读、不可重复读和幻影都消除了，那么调度一定是可串行化的了。脏写现象的存在说明这个观点是不正确的。

SQL标准规定，同一个应用里的不同事务可以在不同的隔离级别上执行，而每个这类事务都会看到或看不到与其级别相对应的现象。例如，一个在REPEATABLE READ级执行的事务读取某个元组若干次，它总是得到相同的值，尽管其他的事务在另外的隔离级别上执行。类似地，一个按照SERIALIZABLE级执行的事务，它所看到的数据库视图，对于由所有其他事务所做出的变更，总是可串行化的，完全可以无视这些事务的隔离级。

1. 隔离级别的加锁实现

由于SQL标准是通过行为来定义隔离级别的，所以它并未对并发控制的实现作出约束。特别地，该定义并没有隐含地说明并发控制必须通过锁来实现。然而，锁确实形成了绝大多数并发控制的基础，因此，考察一下在基于锁的系统里如何支持SQL隔离级别，是很有用处的。

我们所描述的实现是在[Berenson et al. 1995]里提出的[⊖]。

撇开隔离级别不说，DBMS一般总是保证每个SQL语句能够原子性地执行，而且这种执行是与其他语句的执行相分离的。加锁功能比控制不同事务的语句交叠的方法的强度更高。

各个隔离级别都是采用不同的办法加锁来实现的。因此，一个锁既可以是对某个项（元组、页或表）施加的常规锁，也可以是某个谓词锁。我们描述的是采用谓词锁的实现，虽然我们已指出过，鉴于其复杂性，一般并不采用谓词锁实现。之所以这样使用，是出于教学上的考虑。这样，我们更精确地谈论，需要对什么进行加锁。实际的实现可能是以表锁取代谓词锁，或者是采用其他的技术（参见24.3.1节）。

有的锁能够一直持有到提交时刻，我们称这类锁为**长期**（long duration）的；而有的锁可能会在该语句访问数据项或者谓词之后便释放，这类锁称为**短期**（short duration）的。短期锁不足以保证可串行性。然而，通过要求一个事务请求短期锁，并发控制能够检查它是否与其他事务所持有的锁冲突，一旦出现冲突，便强行要求请求者等待。如果不提出加锁请求（就像READ UNCOMMITTED级那样），就会忽视锁集合里存在锁冲突的情况。

所有的隔离级别都以同样的方式使用写锁。长期写锁是施加在与UPDATE、INSERT和DELETE语句相关的谓词上的。

某个特定级别的实现，不仅会排除相应的不希望出现的现象，同时还可能会排除其他的现象。由于写锁在任何隔离级别上都是长期的，因此，脏写也是所有级别的规则所排斥的。通过某个SELECT语句获得的读锁在各个隔离级别上的处理是不一样的。

- **READ UNCOMMITTED级**　完成读取，无须获得读锁。因为读取并不涉及加锁机制，一个事务可以对某些数据项或谓词持有写锁，哪怕别的事务正在读取它们。因此，一个事务可能会读取未提交的（脏）数据。
- **READ COMMITTED级**　对SELECT语句所返回的每个元组t，获得短期读锁。其结果是，与写锁起冲突的都被检测出来，由于写锁是长期的，所以脏读是不可能发生的。然而，由于施加在t上的读锁在读取完成时便释放了，在某个特定的事务里，两个相继执行的同样是返回t的SELECT语句，有可能会被另行执行的某个其他事务隔开，而后者或许将在更新t后提交。因此，返回的t值，有可能是不一样的。
- **REPEATABLE READ级**　对SELECT语句所返回的每个元组t，获得长期读锁。其结果是，t的不可重复读是不可能出现的。可是，由于与该SELECT语句相关的谓词没有加锁，所以有可能出现幻影。
- **SERIALIZABLE级**　所有的读（和写）锁都是长期谓词锁，因此，幻影是不可能出现的。所有的事务都是可串行化的。

图24-3汇总了各个隔离级别的读锁的用法。值得注意的是，由于所有的写锁都是长期的，而所有的隔离级别除READ UNCOMMITTED级以外都要求，在读取一个数据项之前先获得读锁，所以级别高于READ COMMITTED的所有事务的调度都是严格的。

由于所有的事务都使用长期谓词写锁，所以它们的写入操作都是可串行化的。因此，一

⊖ 要知道，尽管许多DBMS是利用下述的加锁协议来实现隔离级别的，但还有一些DBMS是使用不同的实现方法的。在特定情况下，实现不同，展现出的行为可能也不同。

个在SERIALIZABLE级运行的（因此是采用长期读谓词锁的）事务，要么能看到由某个并发事务所完成的全部更新，要么一个也看不到。也就是说，对于所有其他独立于其执行的隔离级别的事务来说，它是受到串行化处理的。然而，在较低隔离级别执行的事务就不一定能看到一致的状态，因此，它们的更新可能会造成不一致。SERIALIZABLE级事务一旦看到这种不一致，它们的计算就会受到影响。

隔离级别	读　锁
READ UNCOMMITTED级别	无
READ COMMITTED级别	对返回元组施加短期锁
REPEATABLE READ级别	对返回元组施加长期锁
SERIALIZABLE级别	对语句指定的谓词施加长期锁

图24-3　各个隔离级别的加锁实现中使用的读锁。所有级别都对谓词使用长期写锁

较低隔离级别所要求的读锁是弱于SERIALIZABLE级采用的长期谓词读锁的，因此是这些级别可以改善性能的原因所在。由于各个级别读锁的管理消除了相应级别所禁止的异常现象，因此在不同隔离级别上执行的事务能够并发地运行，并满足各个级别的规格说明。

2. *在低隔离级别执行的危险*

由于允许事务在弱于SERIALIZABLE级的级别运行，就有可能会产生不可串行化的调度。因此，一个事务一旦看到不一致的数据，它就可能会将不一致的数据写入数据库。为了说明在较低隔离级别下这种情况是如何发生的，我们考虑如下的例子。

(1) READ UNCOMMITTED级

在READ UNCOMMITTED级执行的某个事务T_2可能会读取到由另一个活动事务T_1所产生的脏值。这些值可能从未成为数据库的真正组成部分，因此是毫无意义的。例如，T_1可能向某个数据项写入过一个值v，但后来异常中止了。T_2有可能是在T_1异常中止之前读取该数据项的，然后向用户返回v。或者，T_2可能在值v的基础上计算出某个新值，并将其存入另一个数据项里。因此，由于产生值v的事务异常中止，导致数据库出现混乱[⊖]。或者，T_1可能先将值v写入该数据项，然后又用不同的值改写了值v。这时，对于外部消费来说，值v纯粹是个没有意义的中间值。即使T_1只将最终值写入数据库，但若T_2在T_1提交之前看到它们，那么还是有可能会产生不可串行化的调度。例如，图24-4中的调度就是不可串行化的。在图中，T_1是银行事务，它从余额保存在元组t_1里的一个账户（初始为$1000）中取出$100，并将$100存入余额保存在元组$t_2$里的一个账户（初始为$500）。T_2是在READ UNCOMMITTED级执行的一个只读事务，它打印所有账户的总余额。该调度表明，T_2读取了一个未提交的值t_1，因此，两个账户里正在传递着$100没有得到报告。

(2) READ COMMITTED级

在READ COMMITTED级执行的事务T_1，对各个元组使用的是短期读锁。因此，如图24-5所示，事务T_2可以更新元组t，然后在T_1的相继两次读取之间提交。人们可能会认为这是个严重问题，因为一个事务两次读取相同的元组不太可能。然而，即便事务不打算第2次读取，也

⊖　为了防止数据库发生混乱，通常要求在READ UNCOMMITTED级运行的事务是只读的。

可能会出现不正确的结果[⊖]。图24-6给出了一个调度的例子，其中T_1和T_2都是在READ COMMITTED级执行的银行存款事务，它们都在同一个账户上操作，该账户的余额也全都存入元组t。由于T_1的读锁是短期的，因此，对于T_2来说，更新t后提交是不可能的。结果，T_2更新的效果被丢失了。值得注意的是，两个事务读取的都是已提交的数据。这就是第2章所介绍的更新丢失的问题。

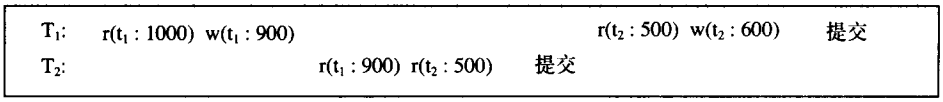


图24-4　涉及读取某未提交数据的调度。T_2在READ UNCOMMITTED级执行

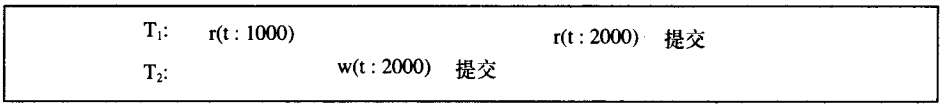


图24-5　涉及不可重复读的调度。T_1在READ COMMITTED级执行

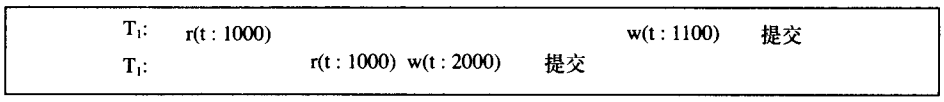


图24-6　说明在READ COMMITTED级执行更新会丢失的调度例子

图24-7给出了在READ COMMITTED级不正确调度的又一个例子。其中，事务T_1看到的是不一致的数据库视图。假设某个完整性约束是：x和y的值必须满足$x \geqslant y$，而初始时x=10，y=1。T_1既读取x也读取y，但在两次读取之间（在T_1对x已放弃读锁之后），另一个事务T_2对x和y值进行了修改（以便使新值能满足完整性约束），然后又提交了。T_1看到的x值，是在T_2对其修改之前的值，而y值则是在T_2对其修改之后的值，这个数据库视图一般是不满足其完整性约束的。由于事务只有在看到数据库的一致性视图时，才能确保执行正确，因此T_1的执行方式可能无法预料，也许还会再向数据库写入错误的数据。即便T_2在写入时有可能使读取的两个值刚好满足完整性约束，但是鉴于它们是来自数据库的两个不同的版本的事实（一个是在T_2执行之前的，而另一个是T_2执行之后的），就有可能使得T_1的执行不正确，并向数据库写入错误的数据。

(3) REPEATABLE READ级

由于只是元组(而非谓词)具有长期锁，因此，幻影是有可能出现的。我们在24.1.1节已看到，这可能会导致不正确的行为。值得注意的是，其中的例子也涉及看到了数据库的不一致的视图的事务。

24.2.1　更新丢失、游标稳定性和更新锁

图24-6是更新丢失行为的一个例子，出现该行为的原因是在READ COMMITTED级仅仅

⊖ 这是由于（根据所防止的现象）描述隔离级别的方法的非形式化特性造成不确定性的一个例子。是否不可重复读仅仅出现在一个事务试图第2次读，并获得不同的值的时候？抑或只要试图第2次读取，就会出现不可重复读，使其读取到不同的值？

应用短期读锁。更新丢失问题的一种特例（就是读取是通过游标来完成的情况）可以通过一种CURSOR STABILITY级（游标稳定级）的隔离级别来防止更新的丢失，在许多SQL实现里，都用它来代替READ COMMITTED级。

我们尚未讨论通过游标进行的访问是如何受到在不同隔离级别执行的事务的影响的。当某个事务T_1打开关于表R的一个INSENSITIVE（非敏感）游标时，就会作出其结果集的一份拷贝，随后的通过游标而运行的所有FETCH语句都是用该拷贝完成的。因此，无论T_1在哪个隔离级别执行都不成问题，该FETCH语句决不会看到由某个并发执行的事务T_2（甚至包括由T_1本身）随后对R作出的任何更新。

然而，倘若T_1打开的游标未声明为INSENSITIVE型，比如它是KEYSET_DRIVEN（关键字集驱动）型的，那么便返回一组指示R上元组的指针（结果集便是根据R构造的），随后的所有FETCH操作都是通过指针作出的。如果T_1在READ COMMITTED级执行，那么它只要求对元组施加短期读锁。如果T_1和T_2是并发执行的，那么T_1可以（在T_2更新之前）通过游标先取回某些元组，其他的元组则等待T_2对其更新并提交之后再取回[⊖]。

特别地，T_2可能会更新一个元组，而此时某个游标正在指向它。如果T_1先读取该元组，后来又在移动游标之前对它进行了更新，那么这种情况特别麻烦。因为在读取和更新之间，T_2可能会读取该元组，更新它然后提交。在T_1更新之后，T_2的更新将被丢失。CURSOR STABILITY级（游标稳定级）可以防止这一类更新丢失。

CURSOR STABILITY级是READ COMMITTED级的一种扩展。因此，它提供的隔离级的强度介于READ COMMITTED级和REPEATABLE READ级之间。使用CURSOR STABILITY级，只要由事务T_1打开的游标指向某个元组，其他事务T_2就不能修改或删除该元组。然而，一旦T_1移动或者关闭此游标，T_2就又可以修改该元组了。

正如在其他隔离级中加锁一样，CURSOR STABILITY级也可以通过使用长期写谓词锁（或等价物）实现。读锁的处理如下：

> CURSOR STABILITY　*应为读取的每个元组施加短期读锁，除非该元组是通过游标访问的。通过游标而对元组施加的读锁是一种*中期（medium-duration）锁，*它在游标指向元组时获得，在游标移动或关闭时释放。*

如果采用CURSOR STABILITY级，就有可能出现图24-6所示的调度，因为我们隐式地假设T_1和T_2都不通过游标来引用t。然而，假设T_1是向银行的所有账户添加利息。它通过一个游标不断地访问各个元组，首先读取t的余额，接着就用新的余额来更新t。采用CURSOR STABILITY级，T_1对t所获取的读锁会一直维持到它请求更新t为止。这时，该锁升级为写锁，并一直保持到T_1提交为止（因为写锁是长期的）。因此，对于其他事务T_2来说，想在T_1的读取和更新之间进行更新是不可能的。

⊖ 这里可能会产生一些混淆，因为SQL标准要求，每个SQL语句都应该以原子性和隔离性的方式执行。在使用游标的情况下，OPEN和FETCH语句的执行满足原子性和隔离性。可是，正在取回（非INSENSITIVE型游标的结果集里的）数据行的事务T可能会读取到被其他并发事务更新过的一些数据行，而其他的事务则不会。这时，事务T便不满足隔离性，它就可能会看到结果集的不一致的视图，哪怕此集合是由一个满足隔离性的OPEN语句创建的也无济于事。

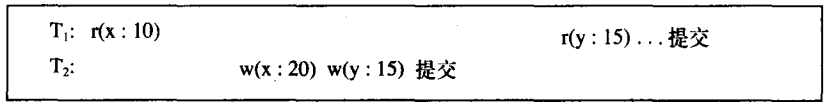


图24-7　说明在READ COMMITTED级执行事务可能看到不一致的数据库视图的调度的例子

考虑图24-8所示的情况。假设T_2是存款事务，它直接通过一个索引来访问t，并假定它是在READ COMMITTED级或CURSOR STABILITY级执行的（因为它们没有使用游标时，它们没有任何差别）。T_2在T_1读取t之后再读取t。当T_1请求更新t时，它的读锁可能已被升级为写锁，因为T_2的读锁是短期的，故而在读取完成时就被释放了。遗憾的是，更新丢失问题并未得到解决。T_2写入的值（在T_1提交之后）是建立在其原先读取到的值，而并非是T_1写入的新值的基础上的，因此，T_1的更新被丢失了。

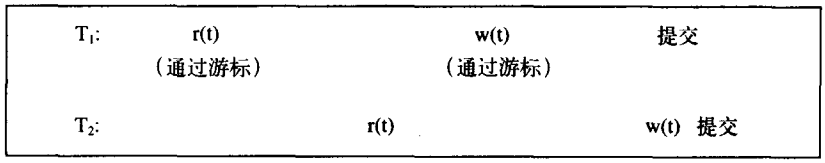


图24-8　说明CURSOR STABILITY级并非灵丹妙药的调度的例子

在我们描述如何解决这个问题之前，先考虑另一种情况：T_2也是在CURSOR STABILITY级执行的，并通过一个游标来访问t。这一次的情况又很让人失望，因为T_1和T_2都在读取后持有各自的（中期）读锁，当两者都试图升级为写锁时，就可能出会现死锁。很显然，CURSOR STABILITY级仅仅提供了更新丢失问题的部分解决办法。

一些商用DBMS为处理这些问题提供了若干补充机制。

- 在某些系统里，一个事务T可以在读取数据项时请求对该数据项施加一个写锁，以便使得T能够在随后对数据项进行更新。这就避免了升级读锁的要求（因此也避免了死锁），可是这样就会拒绝T首次访问开始后的其他事务访问数据项，哪怕这些事务仅仅只是读取数据而已。
- 某些系统提供一种称为**更新锁**（update lock）的新型锁，事务可以在开始需要读取某个数据项，随后打算对其进行更新的场合使用它[⊖]。该锁允许一个事务读取数据项但不允许写入数据项，并指明以后它很可能升级成一个写锁。更新锁与其他同类锁或写锁都冲突，但与读锁没有冲突。因此，倘若T_1和T_2都请求更新锁，那么第一个请求的将被批准，而另一个请求则需等待。这样，丢失更新和死锁都得到了避免。由于更新锁和读取锁并不冲突，所以倘若T_2仅仅要求读取t，那么它可以在T_1的读取和写入之间获得某个读锁，因为T_1所持有的只是这段时间里的更新锁。T_1必须在它打算写该数据项之前，将更新锁升级成写锁。当T_1请求升级时，假若正好有其他事务持有读锁，那么T_1必须等待。
- 某些系统提供一个称为OPTIMISTIC READ COMMITTED级的READ COMMITTED级的版本。如果T_1在这个级别执行，它将获得与在READ COMMITTED级执行时获得的相同的短期读锁。然而，如果T_1后来试图写入原先读取过的元组t时，有某个别的事务，

⊖　更新锁有时也称为带写入意图的读取（read-with-intention-to-write）锁。

在T_1读取元组t到试图写入t的这段时间里曾经修改过t并且提交了，那么T_1就将异常中止。这个方法之所以称为“乐观”，是因为事务乐观地假定：各个事务不必在元组上维护读锁，却仍可防止更新的丢失[⊖]。就像采用其他的乐观算法一样，倘若乐观假设失败，则事务便异常中止。虽然OPTIMISTIC READ COMMITTED级别防止了更新的丢失，但是它仍可能导致不正确的调度（见练习24.17）。

24.2.2 案例研究：正确性和非可串行级调度——学生注册系统

虽然我们已经给出过在比SERIALIZABLE级低的级别上执行可能会造成不正确行为的例子，可是往往可以用一个应用的语义来说明不正确行为是不会发生的，或者说没有任何严重影响。在这类应用里，事务可以安全地在低于SERIALIZABLE的级别执行，从而具有较高的并发性和良好的性能。我们假设DBMS采用本节前面介绍的每个级别的加锁的方法。

例如，我们曾经谈到，较低隔离级别允许的非可串行级调度将会违反某些完整性约束。然而，回想一下，许多种完整性约束都能够在数据库模式里声明，故而可以由DBMS自动地检查。因此，由不完整的隔离级别所造成的对约束的破坏完全可以由DBMS检测到，而该事务（姑且称之为约束性检查），在其请求提交时将异常中止。

因此，我们现在假设，大多数完整性约束都是在数据库模式里声明的，我们现在必须讨论的不正确调度仅仅包括：

- 产生未在模式中声明的违反完整性约束的数据库状态。
- 产生虽然一致却不正确的数据库状态，因为它们不能反映事务的期望结果。例如，由于更新丢失而产生的数据库状态。
- 返回用户的数据建立在不是从某个一致性快照所得到的数据库视图基础上。比如，从某个在READ UNCOMMITTED级执行的只读型事务场合。

当我们考虑某个应用的一个特定的事务是否能够安全地在某个较低隔离级别执行时，必须弄清楚它与同一应用里的其他事务之间的交互关系。学生注册系统里的下列事务则说明了这样的一些可能性，该系统的数据库模式在5.7节给出。

(1) READ UNCOMMITTED级

考虑某个打印当前已注册的某一学生的课程信息的事务T_1。它通过使用一串FETCH语句的某个游标来读取与该学生Id相对应的表TRANSCRIPT里的行记录。这类游标大多声明为INSENSITIVE型，因此，一旦打开游标，就会获得与该学生相对应的行记录的一个快照。然而，不正确调度问题依然会碰到，这与如何声明游标无关。

假设T_1在READ UNCOMMITTED级运行。我们必须考虑它可能会与某个并发事务T_2交互的情况。下面列出一些这样的情况：

- 假设T_2通过某个UPDATE语句修改了表TRANSCRIPT里由T_1读取到的行记录集合里的若干字段。由于任何SQL语句的执行都是由DBMS保证隔离性的，这就保证了T_1所看到的任何行都不会处于因T_2的UPDATE语句部分执行而导致的状态。T_1所看到的值要么来源于

⊖ 这个方法有时也称为*先提交者赢*（first-committer-wins）方法，因为对于写入某个元组的第一个事务可以提交（赢），而第二个试图对该元组写入的事务便会异常中止（输）。我们将在24.5.3节再深入讨论先提交者赢方法。

T_2的更新出现之前的行记录快照，要么来源于T_2的更新完成之后的行记录快照。因此，这样的交互是不会造成什么问题的。

- 假设T_2通过不同的UPDATE语句修改了表TRANSCRIPT里某一行的若干字段。由于处于READ UNCOMMITTED级，T_1既没检查，也没加锁，所以完全有可能检索到被T_2部分地更新过的行的属性值。这种情况是不能接受的，所以我们必须检查该系统的设计文档，断定没有任何事务是以这种方式运行的。
- 假设T_2通过不同的UPDATE语句修改了表TRANSCRIPT里的若干行，那么T_1就有可能读取到T_2更新之前的行记录和另一些已被T_2更新过的行记录。类似地，如果T_2通过不同的SQL语句插入或删除了若干行记录，那么T_1也许会报告某种绝不可能存在的状态。因此，我们再次必须确定，在设计文档里绝无任何事务是以这种方式运行的。
- 假设T_2更新了某行记录，接着T_1又在T_2完成之前读取了该行数据（一个脏读），可是T_2后来却异常中止了。于是，T_1读取的信息后来进行了回退。虽然T_1极少有可能会把这个（已异常中止的）信息返回用户，但我们也必须保证这时没有产生严重的后果。

如果上述情况都没有任何问题，T_1就可以在READ UNCOMMITTED级正确地执行。结果，加锁既不会造成延迟，也不会遇到任何的延迟。

(2) READ COMMITTED级

考虑为某个学生注册一门课程c的事务T_1。它是通过执行以下的步骤实现的（其中有些步骤省略了）:

1) 通过读取表REQUIRES确定c的预备课程，对于每门课程的每一门预备课程，REQUIRES表都有一行记录。

2) 通过读取表TRANSCRIPT，检查该学生应完成的c的预备课程，以确定该学生是否已经完成每一门预备课程，并获得过至少为C的成绩。

3) 检查班级c是否有足够大的教室，若有，便增加当前注册人数。为了完成这项工作，T_1执行如下语句:

```
UPDATE CLASS
SET Enrollment = Enrollment + 1
WHERE CrsCode = :courseId AND Enrollment < MaxEnrollment
```

其中班级c的课程代码存放在宿主变量courseId里，同时，我们假定CLASS包含属性Enrollment和MaxEnrollment，前者包含下学期已注册该课程的学生数目，而后者则包含c中最多可以注册的人数。（值得注意的是，这里给出的方法与12.6.2节设计的在模式中检查该条件的方法有所不同。）

4) 在表TRANSCRIPT里插入一行记录，表示该学生已注册了班级c 。

假设T_1是在READ COMMITTED级执行的。我们必须考虑以下情况:

- 当第1步中T_1在利用行的短期读锁读取REQUIRES时，可能有某个并发的事务T_2也正在通过插入与c的预备课程相对应的新的行记录（包括这些预备课程的通过日期）更新REQUIRES，然后提交。

为了增加例子的趣味性，我们假定T_1在读取REQUIRES时采用DYNAMIC型游标。如果T_1和T_2并发地执行，那么T_1可以看到一部分（但不是全部）由T_2所插入的行记录。因此，T_1对于T_2

来说就不是可串行化的。

然而，在这种情况下，不可串行性并不会造成任何问题，因为在第3.3节曾经规定过，不对当前注册者应用新的预备课程。所以，在第1步中确定c的预备课程时，使用各个预备课程行的日期属性的T_1忽略了那些在本学期追加的预备课程。因此，T_1读取到一部分（而非全部）由T_2追加的新的预备课程时不会有任何问题，因为不管读到什么内容，它都将其忽略的。这样一来，T_1在READ COMMITTED级也能够正确地执行。

值得注意的是，即便T_1采用KEYSET_DRIVEN型游标，在READ COMMITTED级执行仍然是正确的，因为T_1看不到插入的任何行。（不过，其理由却令人扫兴，它是出于KEYSET_DRIVEN型游标的语义，而不是出于该事务的语义。）

- 在第2步中，当T_1读取TRANSCRIPT，并放弃读到的行记录上的短期读锁之后，某个并发事务T_2可能会通过改变该学生某门课程的成绩，或者追加该学生尚未修读过的某门新课程来更新TRANSCRIPT。这一修改也许会影响到c的预备课程。T_1是看不到这种修改的影响的，然而，倘若它在SERIALIZABLE级运行的话，那么它就要在第2步里对TRANSCRIPT加上长期谓词锁（或其等价物）。该锁用于防止T_2在T_1完成之前更新TRANSCRIPT。因此，在这种情况下，T_1是不会看到T_2的影响的，所以，在这种情况下，两个事务（T_1和T_2）都可以在READ COMMITTED级运行。
- 可能会发生试图将学生注册到c里的两个注册事务并发执行的情况，这样会导致丢失更新（如图24-6所示）。然而，在本例中，这种更新丢失是不会出现的，因为第3步中的检查和增加是作为某个SQL语句的隔离执行的一部分而完成的(值得注意的是，它不必依赖于由长期写锁提供的保护机制，就能确保这一点)。类似的论据也可以应用到并发执行某个注销事务的场合。

由于这些情况都不会产生不正确的调度，所以该注册事务能够在READ COMMITTED级正确运行。因此，在步骤1和步骤2中所要求的读锁将很快释放，以改善性能。

(3) REPEATABLE READ级

在REPEATABLE READ级执行的事务中，唯一可能出现的不利情况是由幻影引起的。考虑完成重新分配教室的某个事务。当新学期来临时，学校可能想腾出大教室，决定把原先分配到大教室、但注册人数较少的课程重新分配到较小的教室。新教室必须能够容纳该课程当前已注册的学生，该课程的最大允许注册人数必须低于该教室的所能容纳的人数。

重新分配教室事务T_1是在某个特定时间段里为所有课程执行这一功能的。为此，它需要完成以下步骤:

1) 在指定时间段讲授的课程里，标识出满足条件（最大注册人数超出其当前注册人数达某个阈值以上）的课程。为了完成这个任务，T_1使用了建立在下述查询基础上的游标:

```
SELECT   C.CrsCode, C.Enrollment, C.MaxEnrollment,
         C.ClassroomID
FROM     CLASS C
WHERE    C.ClassTime = :timeSlot
         AND C.MaxEnrollment - C.Enrollment > :thresh
```

其中时间段和阈值分别存放在宿主变量timeSlot和thresh里。正如5.7节一样，我们假定CLASS具有ClassroomId和ClassTime两个属性，前者标识出下个学期讲授该课程所在的教室，而后者

则规定该课程在下个学期将占用的时间段。

2) 对步骤1所确定的每门课程，读取表CLASSROOM和CLASS，以确定是否存在更小的教室能够容纳当前已注册的学生，而且在所规定的时间段里未被占用。

3) 对于存在更小教室的每门课程，更新CLASS的ClassroomId属性，以反映出新的教室分配；同时更新CLASS的MaxEnrollment属性，以反映新教室的大小。

我们首先观察到，在READ COMMITTED级，T_1是不能正确执行的。在第1步中，T_1在CLASS上施加的短期读锁并不能让注册事务防止在步骤1到步骤3之间使注册人数增加。因此，可能有些课程分配到的教室太小，不能容纳在册学生。

现在假设T_1是在REPEATABLE READ级执行的。我们必须考虑以下三种情况：

- T_1完成了第1步之后，在所考虑的时间段中，某个并发的事务T_2可能会在表CLASS里插入与某个新班级（幻影）相对应的新的行记录。追加这样的新班级不仅不太可能，而且即便发生了，也不会对T_1的目标造成太大的影响。重新给这个新班级安排教室的要求是不会被考虑的，而所有现有的班级都会得到正确的处理。此外，其效果也与T_2排在T_1之后顺序执行的效果相同。
- 类似地，T_1完成了第2步之后，在所考虑的时间段中，某个并发的事务T_2可能会在表CLASSROOM里插入与某个新教室（幻影）相对应的新的行记录。然而，追加新的教室是不太可能的，也不会对T_1的目标造成太大的影响。此外，其效果与T_2排在T_1之后顺序执行的情况完全相同。
- 由于T_1减少了MaxEnrollment，不难想象，它会干扰在READ COMMITTED级执行的注册事务的正确操作，因为该操作要访问MaxEnrollment，并为某个特定的课程c增加Enrollment。如果两个事务并发执行，是否有可能最终会处于(对于某个c) MaxEnrollment < Enrollment的状态呢？这是不会的，因为唯一可能的途径是，该注册事务在T_1执行第1步和第3步之间，增加了Enrollment。然而，这是不可能发生的，因为在第1步里，T_1就已对关系CLASS里c的行施加了某个（长期）读锁。

由于上述情况都不会造成不正确的调度，所以T_1就可以在REPEATABLE READ级运行，而不是在SERIALIZABLE级运行。只有T_1实际访问的那些页或行，才需要加上读锁，因而，其性能也得到了改善。

(4) SERIALIZABLE级

事务T_1检查与本学期某门课程在册生所对应的TRANSCRIPT表的行数是否与CLASS表中该课程的行所记录的当前注册人数相等。因此，对于各门课程来说，它完成以下的步骤：

1) 利用下述语句，对TRANSCRIPT里的行进行计数，以确定该课程的在册生人数。

```
SELECT COUNT(*)
INTO :registered
FROM TRANSCRIPT T
WHERE T.CrsCode = :courseId
    AND T.Semester = :this_sem
```

2) 利用下述语句，从CLASS的合适行检索Enrollment值，以确定课程的在册生人数。

```
SELECT C.Enrollment
INTO :enrolled
FROM CLASS C
WHERE C.CrsCode = :courseId
    AND C.Semester = :this_sem
```

3) 比较enrolled与registered。

T_1必须在SERIALIZABLE级执行。倘若它在REPEATABLE READ级执行，则某个注册事务就可能会在步骤1和2之间交叉。虽然T_1可以对与某门课程的在册生相对应的所有行设置长期读锁，但是，注册事务可能会为注册该门课程的学生插入某个幻影行，更新该课程在CLASS里的行记录，然后提交，并释放其全部锁。这样，第2步里，T_1读取到的该课程的值，就不会等于它在第1步里所获得的值。为了防止出现这种幻影，T_1必须在SERIALIZABLE隔离级运行。值得注意的是，倘若将步骤1和2颠倒一下，那么这种特殊的交互可能就不会发生。

我们已经给出过若干个在低于SERIALIZABLE隔离级执行将产生不正确结果的例子，而其他的例子则不会产生不正确的结果。对于任意给定的应用来说，一个谨慎的设计师总是假定，使用较低的隔离级别可能会产生不正确的结果，除非已被明确地告知，根据该应用的语义，这类结果是完全不可能产生的。

此外，我们注意到，凡是由于选择使用较低的隔离级别而导致的错误都是难以追查的。它们往往出现在事务错误地交叉的调度中。倘若特定事务不常调用的话，就不会经常出现这种出错情况，而这种不正确调度的后果，可能直到其执行后的很长一段时间里才会显现出来。正是由于这个缘故，系统的工作在很长一段时间里仿佛都很正常，直到有一天突然检测到存在某个不一致的状态。要确定造成这一错误的事件序列是极端困难的。

从软件工程的观点看，还有一些值得注意的地方。即便一个应用的初始版本的语义可以保证在较低的隔离级别也只会产生正确的调度；但后来的版本的语义却不能做出这样的保证，因为系统可能追加了新的事务，或者修改了某些旧的事务。因此，应当仔细地在文档（或许是设计文档）中记录在较低隔离级别运行的理由，以便以后让系统维护人员判断对于更新后的版本，该理由是否仍然有效。

(5) 修订的注册事务

在12.6.2节里给出的注册事务的设计与这里所描述的设计是不一样的，前者假设，一个班级中能够注册的学生人数受到数据库模式中某个完整性约束的限制。故而，在该种设计里，两个并发执行的注册事务引起超出限制的情况，也是不可能存在的（不过，理由却是不一样的）。因此，我们可能会认为，在那种设计里的注册事务在READ COMMITTED级也可以正确地执行。

然而，在12.6.2节里我们曾经考虑过某些附加条件，这里却还没有考虑到。例如，该事务检查（在事务代码内部）注册不至于造成某个学生在该学期选修课程的总学分超过20的极限。它通过读取TRANSCRIPT来确定当前注册的学生的学分情况。倘若该约束得到满足，它就在TRANSCRIPT里插入一个与新课程相对应的新的元组。

如果该事务在READ COMMITTED级（或在REPEATABLE READ级）执行，就有可能存在学生学分超过极限的情况。倘若该学生启动了两个并发的注册事务，以注册两门不同的课程，但这两个事务都没有看到对方插入TRANSCRIPT里的（幻影）元组，它们又都提交了，结果，学分的极限就有可能被超过。

也许设计师觉得发生这种情况的可能性很小，该事务可以安全地在READ COMMITTED级运行。若不是，该注册事务就应当在SERIALIZABLE级执行，以便消除幻影。

(6) 学生注册系统中的死锁

现在考虑对于同一门课程的两个事务（注册事务T_1和注销注册事务T_2）并发执行，同时假定DBMS采用页加锁。

- T_2首先删除TRANSCRIPT里描述注册该课程的学生的行记录，然后，减少CLASS里的Enrollment（因为若该学生还没有实际注册的话，那么就不存在让Enrollment减少的情况）。于是，T_2首先获得对于TRANSCRIPT的某个长期谓词写锁，然后获得对于CLASS的长期谓词写锁。尽管我们还没有讨论过谓词锁的实现，但不妨假设，这两个锁都是锁住正要更新的页面。
- 不妨回忆一下，注册事务T_1首先更新CLASS（因为倘若该课程没有可用的教室的话，那么在TRANSCRIPT里根本就不存在地方让它插入元组），然后再在TRANSCRIPT里插入一个元组。于是，T_1首先获得对于CLASS的某个长期谓词写锁，然后再获得对于TRANSCRIPT的长期谓词写锁。我们再一次假设，两个锁都是锁住正要更新的页面。

两个事务都是更新CLASS里的同一个元组。如果T_1试图把TRANSCRIPT里的新元组插入到包含被T_2删除的元组的页面里，就可能产生死锁，因为处在对立状态中的T_1和T_2都想获得各自的锁。由于所有的隔离级别都采用的是长期写锁，所以这个死锁与选择的隔离级别无关。

值得注意的是，两个注册事务其实都不会死锁，因为它们都以相同的顺序要求对两张表加锁：首先对CLASS加锁，然后对TRANSCRIPT加锁。因此，要求事务按照相同的顺序对某个项加锁，是标准的避免死锁的一种手段。

24.2.3 可串行化、SERIALIZABLE和正态的

在描述并发控制所产生的调度时，我们已经使用过三种术语：

- *可串行化*（Serializable）　等价于某个串行化调度。
- SERIALIZABLE级（可串行级）　一种SQL隔离级别。它不允许脏读取、不可重复型读取以及幻影，而且调度必须是可串行化的（如[SQL 1992]中的ANSI规格说明所述）。
- *正态的*（Correct）　让数据库处于一种正确地模拟现实世界，并且满足企业的业务规则（如同其规格说明文档所述）的状态。

这些定义有以下关系（假定每个事务都是一致的）：

- 如果某个调度是可串行化的，那么它一定是正态的。
- 如果某个调度是由一组在SERIALIZABLE隔离级执行的事务所产生的，那么它是可串行化的（因而也是正态的）。

上面两个结论反过来不一定成立。

- 一个调度可以是正态的，即使它不是可串行化的。
- 一个调度可以是可串行化的，即使它是通过在低于SERIALIZABLE隔离级执行的事务所产生的。

前一节里已经提供了证明这两个结论的一些例子。

- **正态的，但非可串行化的**　注册事务可以正确地在READ COMMITTED级执行，尽管相对于追加新预备课程的事务来说，它不是可串行化的。
- **可串行化的，但有些事务不是SERIALIZABLE级的**　除了教室重分配事务是在

REPEATABLE READ级执行的以外，其他所有事务都是在SERIALIZABLE级执行的，这种调度是可串行化的。这是因为，当教室重分配事务T正在执行时，某个事务有可能会创建出幻影（通过追加新的教室或新的课程），该事务可能实际上被串行地排在T的后面，因为T不会看到幻影。

这些例子再一次说明可以获得正确性，但不一定要通过严格的加锁协议来保证串行化调度。

24.3 粒度加锁：概念锁和索引锁

在前一节里，我们讨论了如何通过使用并不能确保可串行化执行的隔离级别来改善事务处理系统的性能。而加锁粒度也同样可以影响系统的性能。在本节里，我们考虑允许根据事务的需要调整粒度的加锁算法。这样既可以提高性能，又可以保留可串行化行为。

加锁系统的设计者在选择加锁粒度时，需要在一致性和开销之间进行权衡。因此，当对某个应用程序（其中有一个事务需要访问大量的数据（比如，整张表），而其他的事务却只访问少量的数据（比如，几个元组））执行并发控制时，我们希望使用一种允许不同粒度的加锁机制。

粒度加锁[Gray et al. 1976]就是为了满足这一需要而设计的。一个需要访问大量数据的事务可以用一个简单的命令来给数据加锁；而一个需要访问少量数据的事务可以给每个数据单独的加锁。在后面的事务中，几个事务可以同时在同一数据组中对小的数据项加锁。

举一个简单的例子，一个系统既可以对多个记录加锁，也可以仅对某个记录里的特殊字段加锁。这两种锁的粒度是不同的。假设事务T_1在记录R_1内对字段F_1施加了一个写锁，随后事务T_2需要对于整个记录R_1施加一个写锁。但由于T_2要求访问字段F_1，于是后一个写锁就不能被批准。现在的问题是，需要在申请对R_1加锁时，设计一个有效的机制，使并发控制可以识别对F_1加的锁。

解决的方法是，分层次地安排加锁。在对F_1加锁之前，T_1必须首先对R_1加锁。随后，当T_2请求对R_1加锁时，并发控制将会发现冲突。但是T_1要得到一个什么类型的锁呢？很显然，既不是读锁也不是写锁，因为在这种情况下没有必要对F_1追加粒度细的锁，而且有效锁粒度也会显得很粗糙。因此，DBMS提供了一种新型的加锁方式——意向锁（intention lock）。在一个事务能够对某个数据项加共享锁或排他锁之前，它必须在粒度的层次结构上，对所有需要包含的数据项施加适当的意向锁。于是，在T_1可以对F_1加锁之前，它必须首先对R_1施加意向锁。意向锁有三种类型：

1) 如果T_1要读取R_1的一些字段，它必须首先获得R_1的某个**意向共享**（intention shared，或简写为IS）锁。然后它才可以申请对那些字段施加某个**共享**（shared，S）锁。

2) 如果T_1要更新R_1的一些字段，它必须首先获得R_1的某个**意向排他**（intention exclusive，或简写为IX）锁。然后它才可以申请对那些字段施加某个**排他**（exclusive，X）锁。

3) 如果T_1要更新R_1的一些字段，但是要先读取所有的字段，以便决定对哪些字段进行更新（比如，它要更新所有值小于100的字段），那么它必须首先获得R_1的**共享意向排他**（shared intention exclusive，或简写为SIX）锁。然后它才可以读取R_1内所有的字段，并申请对其需要更新的字段施加某个X锁。（SIX锁是对R_1施加的S锁和IX锁的一种组合。）

	已授权的模式				
申请的模式	IS	IX	SIX	S	X
IS					×
IX			×	×	×
SIX		×	×	×	×
S		×	×		×
X	×	×	×	×	×

图24-9　意向锁的冲突表。× 表明加锁模式之间有冲突

尽管现在事务必须要获得附加的锁，然而由于意向锁可以和许多其他类型锁交换，我们还是可以利用它提高性能。图24-9给出了粒度锁的冲突表。例如，它表明，如果该数据项已经加有S锁，那么再申请对该数据项加IX锁就会遭到拒绝。其理由是，S锁允许读取所有数据项，可是IX锁却允许事务对某些数据项施加写锁。相反地，如果某个数据项已加的是IS锁，那么对其加IX锁的请求就可能会得到批准。其理由是，IS锁允许其子集包含已加S锁的数据项，而IX锁也允许其子集包含已加X锁的数据项。这些子集是不相交的，如果这样的话，就不会有冲突了。然而，倘若它们并非是不相交的，那么由于事务将不得不对所包含的数据项逐个施加S锁和X锁，因此在低级别中就会发现冲突。

在前面的例子中，当T_1对R_1施加IX锁之后，就不再会批准任何事务对R_1施加排他锁了。

一般情况下，需要加锁的数据项是按层次结构组织的，这可以用一棵树来表示。其中，树的节点表示的每个数据项都是包含在此树中其父节点表示的数据项内的。因此，毫无疑问，对该树中的某个数据项加锁意味着对其所有子项加锁（对记录加锁无疑意味着对其所有字段的加锁）。一般规则是，在对某个特定的数据项（不一定是树叶）加锁之前，必须对该层次结构中所有包含它的数据项（祖先）施加某个适当的意向锁。因此，一个事务为了对某个处于S模式的数据项加锁，它首先必须从树根开始，依次对直到该数据项的路径上所遇到的所有数据项施加IS模式的锁。最后才要求加S锁，以便确保该事务只有在所有的锁到位后才能实际访问目标对象。锁的释放则以相反的顺序进行。类似地，要对某个数据项施加X锁，也必须首先对从树根开始到该数据项的路径上的所有的数据项施加IX锁。

我们特意把我们的例子建立在一个使用记录和字段的系统的基础上，而不是建立在使用表和元组的系统的基础上，因此幻影不会是一个问题。然而，当我们把粒度加锁的概念应用到关系数据库上时，这个问题就会凸显出来，届时我们必须要确保幻影不会出现。在下一节，我们将讨论粒度加锁的一种手段，它确实不会导致幻影，并且已用于很多商业DBMS中。

24.3.1　索引锁：无幻影的粒度加锁

我们讨论了在关系数据库中保证可串行化调度的两种方法：谓词加锁和表加锁。我们指出过它们各自的缺陷：谓词加锁具有计算复杂性，而表加锁的粒度粗糙。表加锁的粒度粗糙这个缺点可以通过对个别元组加锁来克服，但是这会导致幻影和不可串行化的现象。对存储元组的页面（而不是元组本身）加锁，在某些方面更加有效，但是同样也会导致幻影。

很多商用DBMS采用一种涉及元组、页面与表格的扩展的粒度加锁方法来消除幻影并确

保可串行化调度。防止幻影的最主要需求就是，在事务T_1采用谓词P访问某个表R以后，没有其他任何并发执行的事务T_2将一个同样满足P的（幻影）元组t插入R中，直到T_1终止为止。如果DBMS使用页面锁，那么当T_1访问R时，它只锁住那些在访问过程中实际被扫描到的页面（除非R的所有页面都被扫描到了，只有在这种情况下，它才对R加锁）。当访问涉及索引时，短语“那些在访问过程中实际被扫描到的页面”并不像看起来那么简单。

我们这里所描述的方法取决于T_1是如何访问R的。在为访问满足P的元组的SQL语句构建查询执行计划时，系统要确定是否有现成的索引可以利用。如果T_1对R执行了一个SELECT语句，并且任何索引可用，那么系统就必须搜索R中的每一个页面，以定位满足P的元组。为了进行搜索，T_1需要对R加S锁。如果直到T_1提交为止仍然保持该锁，那么T_2就不可能插入t，因为它首先必须对R施加（相冲突的）IX锁。同样地，如果T_1利用谓词P对R执行某个DELETE语句（我们后面再考虑UPDATE语句），也没有任何索引可用，那么它同样也必须要搜索R中的每个页面，以定位满足P的元组。在这种情况下，T_1首先需要对R施加某个SIX锁（再次提醒一下，SIX锁相当于S锁加上IX锁）。然后，它就对包含满足P的元组的页面施加X锁[⊖]。同样地，倘若不先对R施加IX锁（它与T_1拥有的S锁相冲突），T_2是不能插入t的。因此，在没有任何索引可用时，利用粒度加锁就能防止幻影的出现。

如果T_1通过一个索引来访问R，那么情况就更加复杂了。在这种情况下，不需要对R进行整体扫描。如果T_1使用谓词P对R执行某个SELECT操作，它只需要对R施加一个IS锁，并对R中含有满足P的元组的页面获得S锁，这些页面可以通过索引访问到。同样，如果T_1使用谓词P对R执行某个DELETE操作，也只需对R施加一个IX锁，并对R中含有满足P的元组的页面施加X锁，这些页面可通过索引访问到。

遗憾的是，这个加锁协议并不能防止幻影的出现。如果T_2试图在R中插入一个幻影，它可以对R施加IX锁，因为这个锁与T_1所获得的IS或IX锁都不会产生冲突。因此，在表层次上没有任何冲突。此外，倘若T_2要插入的幻影存储在另外一个不同的页面上，而并非是T_1加锁的页面，那么在页层次也不会有冲突。无论如何，对于T_2来说，插入幻影是有可能的。因此需要有某种机制来防止上述情况。

例如，在学生注册系统的模式里，表STUDENT有一个属性Address（为简单起见，我们假设地址仅仅指城镇），T_1可以执行下列SELECT语句：

```
SELECT *
FROM STUDENT S
WHERE S.Address = 'Stony Brook'
```

如果对于关系STUDENT中的Address存在索引ADDRIDX，就可以用它来查找住在Stony Brook的学生。需要对STUDENT加IS锁，而对那些包含描述居住在Stony Brook的学生的元组的页面则需要加S锁。然而，这些锁并不能防止T_2插入一个元组t，该元组描述某个住在Stony Brook的新来的学生，因为T_2只需要对STUDENT加IX锁，并对t所插入的页面加X锁。（该页面也许和用于存储居住在Stony Brook的学生的元组的所有页面都不相同）。

为了防止幻影，除了对表应用适当的意向锁并对被访问的数据页面施加页面锁以外，事务还需要对于索引结构本身所在的页面加锁。商用DBMS通常使用两类索引。一类索引包含

⊖ 如果DBMS支持元组加锁，那么它先对R和包含满足P的元组的页面施加SIX锁，然后再对该元组施加X锁。

一个指针，该指针仅仅指向可能存储描述了居住在Stony Brook的学生的元组的那些页面[⊖]。带碰撞的索引，数据文件里每一行的位置都由其搜索键的值控制的聚簇索引就属于这一范畴。集中的静态散列索引（其中在同一个散列桶内的所有行都存储在同一个页面里）也属于这种索引。在第二类索引方式里，那种用来存放所有的描述居住在 Stony Brook的学生的元组的唯一的STUDENT页面是不存在的。当T_2要求插入t时，存储t的页面是确定的，而且一个指向该页面的指针就存储在ADDRIDX上。任何二级索引都属于这一范畴。

我们用于防止幻影的策略取决于索引的类型。如果索引属于第一类，那么通过对包含有描述居住在Stony Brook的学生的元组的STUDENT页面施加长期锁（至于是S还是X，则取决于T_1是执行SELECT还是DELETE），T_1能够使T_2推迟。由于t必须插入到该页面，所以T_2就延迟到T_1完成再执行。例如，在静态散列里，T_1需要对Stony Brook散列到的散列桶加一个S锁。

如果索引是第二类的，那么T_1要在使用索引期间，对访问的整个ADDRIDX叶页面加长期读锁，而T_2要求对所有需要更新的（如它向其中插入指针）叶索引页面加长期X锁。图24-10说明了这种情况。其中B+树作为STUDENT表中Address属性的二级索引。我们假设地址值为$a_i, i>0$，以及$a_i < a_{i+1}$。在执行SELECT语句时，T_1对该索引的叶页面A加长期S锁[⊖]。此索引包含指向STUDENT数据文件中（其中包含描述居住在Stony Brook的学生的元组）的页面B、C、D的指针。以后，当T_2想要插入一个新的描述居住于Stony Brook的学生的元组时，就需要插入一个指针，该指针指向包含STUDENT的每个索引中元组的数据页面。

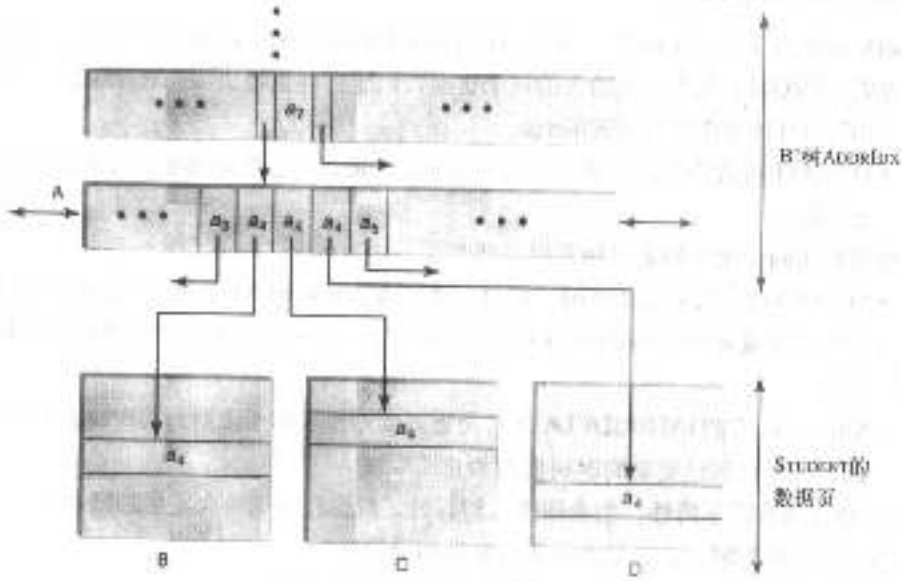


图24-10 STUDENT表上的无碰撞B+树二级索引

在ADDRIDX的情况中，该指针必须要插入索引页面A中，因为叶页面是按Address分类的。这就需要对A加X锁，这和T_1产生了冲突，所以T_2不得不等待。从而新的元组得以防止成为幻影。尽管实际上，它最终也许还是储存于某个不同于B、C或D的数据页面上；而T_1和T_2对

⊖ 注意，当一个页面加锁后，任何和该页面对关的溢出链也会被锁住。

⊖ 一般情况下，也许会有多个这样的叶页面，但是算法是不变的。

STUDENT所加的也只需要是兼容锁(IS和IX)。T_1必须一直保留对A所加的锁，直到它提交为止。从直观上说，通过这种方式对索引加锁，事务能有效地获得对于谓词Address = 'Stony Brook'的谓词锁。

虽然这种方法可以消除由INSERT语句引起的幻影，然而在使用UPDATE语句时，还是会碰到其他的一些问题。一个UPDATE语句和先处理一个删除需要更新的元组的DELETE语句，后跟处理一个插入更新后元组的INSERT语句等价。因此，UPDATE有双重特性，由于它具有DELETE部分，所以它要防止幻影；而由于它又具有INSERT部分，所以它又可能引起幻影。如果使用的是第一类索引，那么T_1所更新的元组也许就不得不移到数据文件的新页面中。例如，在采用散列方法的情况下，如果包含在散列键里的更新后的元组t'的某个属性被修改，那么t'必须被移到某个新的散列桶中。在这种情况下，为了防止T_2插入满足UPDATE语句的WHERE子句的幻影元组，T_1必须保持原始的包含t'的散列桶上的锁。（此外，T_1还需要对t'所移往的散列桶加X锁。）类似地，如果使用的是第二类索引，而包含在搜索键里的t'的某个属性被修改，那么指向t'的指针必须移到一个新的位置。T_1必须始终锁住原始的包含指针的页面，以防止随后插入指向幻影元组的指针。（此外，T_1还必须用X锁锁住指向t'的指针所移往的索引页面。）

我们现在总结一下，无索引可用以及可以使用B^+树索引时的协议。

关系数据库的粒度加锁协议

- 如果在执行某个SELECT、UPDATE或DELETE语句时，没有索引可用，那么系统必须搜索在FROM子句中命名的表的每个页面，以定位满足该语句的WHERE子句的元组。
  - SELECT语句需获得对表的S锁。
  - UPDATE或者DELETE语句则需对表加SIX锁，对包含需要更新或删除的元组的页面加X锁。
- 如果在访问中使用的是B^+树索引，那么
  - SELECT语句需对表加IS锁，对包含满足该语句WHERE子句的元组的页面加S锁。
  - UPDATE或者DELETE语句则需对表加IX锁，对包含需要更新或删除的元组的页面加X锁。
  - SELECT、UPDATE和DELETE语句还需要对搜索期间读取到的B^+树的叶页面加长期S锁，并对任何已更新的B^+树页面加长期X锁。

这一协议具有以下特性，当企图插入幻影时，就会在下列场合发生加锁冲突：

- 当不使用索引时，在表层次可能会发生加锁冲突。
- 当使用索引时，则沿着索引的路径可能会发生加锁冲突。

因此，它确实不允许幻影，并且可以产生可串行化调度。

即便没有索引可用，粒度加锁也比我们以前所描述的方法的并发性更高。在对所有隔离级别（包括SERIALIZABLE级）加锁的过程中，写入语句要求对某个谓词施加长期写入锁。在没有索引和粒度锁时，其实现方法是对相应表施加长期写入锁。通过粒度加锁，一个不使用索引的写入语句只需要对表加SIX锁，并对包含需要更新、插入或删除的行的页面加X锁就可以了。这些锁可以防止并发事务读取那些行及修改表中的任何行，但是允许并发事务读取

该表的任何行，而不是已更新页面里的行。因此，粒度加锁可以防止幻影在SERIALIZABLE级别出现（因而确保了可串行性），但这种方法锁定的数据项比对整个表加写锁要少（因此提高了并发性）。如果写入语句使用索引的话，则就可以实现的并发程度会更高。

值得注意的是，粒度加锁可以用于（幻影不成问题的）较低的隔离级别。比如，在REPEATABLE READ级执行某个SELECT语句，需要对表施加IS锁，并对包含满足此语句的WHERE子句的元组的页面加S锁。但是，与SERIALIZABLE级相比，它不需要对该语句所用的索引的叶页面施加长期写锁。

1. 锁的逐步升级

当一个事务积聚了太多的细粒度锁时，粒度加锁的开销就会非常大。这种开销有两种形式：DBMS中用来记录每次请求的锁的信息的空间开销；用于处理每个加锁请求的时间开销。当某个事务需要对表中大量的页面（或者元组）加锁时，它很可能会这样不断继续下去。因此，把这些锁换成对整个表的单一锁是有好处的。

这种技术就是我们所说的**锁的逐步升级**（lock escalation）。应该在并发控制中设定某个阈值，以限制在某个表中一个事务可能得到的页面锁的数量。当事务达到这一限制，并发控制就会试图对整个表加锁（页面锁的情况也与此类似）。一旦该表锁得到批准，那么对此表施加的页锁和意向锁就可以释放。必须注意的是，在这一模式中存在死锁的危险。如果两个事务都在请求页锁，其中至少有一个是写入者，而两者都达到了它们的阈值，这时就会产生死锁，因为它们都不能升级到表锁。

2. 多级控制*

数据的抽取一层接一层可以达到任意深度，这样建立起来的多级并发控制还没有被广泛采用，但是可以在DBMS中使用一种缩略的形式。关键的问题是，当不得不更新、插入或者删除元组时，对于包含它的整个页面一般是施加排他锁，以便页面中关于存储分配的信息可以被调整。因此，为了确保该语句的隔离性，就需要对某个SQL语句访问到的页面加锁。但是，一旦该语句继续维持那些页锁，就会不必要地降低并发性，因为它们阻止了其他事务以总体上无冲突的方式访问该页面。

为了增加并发性，可以把SQL语句看成是某个事务的子事务。一旦该事务看到了SQL语句的抽取，那么此子事务就也会看到读取和写入的页面的抽取。于是，此子事务可以对其需要访问的数据页面加锁，然后在结束的时候释放它们。这样，该事务就能够维持对元组和访问路径的较高级别的锁，以确保合适的隔离性。

3. 表分段

表分段（table fragmentation）是一种和粒度加锁相关的有用技术。再次以表STUDENT为例。一个希望抽取居住在Stony Brook的学生的信息的应用程序可能执行下述语句：

```
SELECT *
FROM STUDENT S
WHERE S.Address = 'Stony Brook'
```

另一种数据组织结构是将STUDENT划分成一些称为**片段**（fragment）的单独的表，每个城镇对应一个表。比如，我们可以把所有的满足谓词Address='Stony Brook'的元组放到一个表STUD_SB里，而所有满足谓词Address='Smithtown'的元组则放到另一个表STUD_SM里，依此

类推。某个想要检索居住在Stony Brook的学生的信息的事务，现在可以用如下语句执行:

```
SELECT *
FROM STUD_SB
```

我们可以看到，该片段上的表锁等价于用来执行分段的谓词的某个谓词锁。这就消除了用索引加锁替代谓词加锁的要求。由于原始表格已经不再存在，我们就不用实现两种不同粒度的加锁。所以，一个申请对某个片段加表锁的事务，就不再需要更高级的意向锁了。在第18章我们已经详细讨论过分段。它的优势在于，表分段比原始表的粒度更细，所以我们可以获得一个更高级别的并发性。它的劣势在于，贯穿多个片段的查询会变得难以处理。例如，某个要检索所有居住在Stony Brook或是Smithtown的学生的元组的查询，或者是使用并不涉及属性Address的某个谓词的查询都必须要访问多个表。

24.3.2 对象数据库里的粒度加锁*

关系数据库的粒度加锁的许多思想同样也适用于对象数据库。我们考虑某个银行账户的应用程序。假设一个关系数据库中拥有某个其元组代表个人账户的表ACCOUNTS。同样地，假设一个对象数据库拥有某个其对象代表个人账户的类ACCOUNTSCLASS。正如元组包含在表里一样，我们可以认为对象包含在类里。而且，我们还可以使用相同的加锁模式（共享、排他以及相应的意向模式），并用关系数据库中的解释方式来在对象数据库中解释它们:

- 在关系数据库中，对表加锁意味着对其中的所有元组加锁。
- 在对象数据库中，对类加锁意味着对其中所有的对象加锁。

因此，在银行的对象数据库中，某个粒度加锁协议就要求我们在对某个账户对象加锁之前，首先必须对类ACCOUNTCLASS施加合适的意向锁。

对象数据库也支持继承性。因此，在银行应用程序中，类的层次结构可能包括这样的事实：SAVINGSACCOUNTSCLASS和CHECKINGACCOUNTSCLASS都是ACCOUNTSCLASS的子类，ECONOMYCHECKINGACCOUNTSCLASS是CHECKINGACCOUNTSCLASS的子类。由于类ECONOMYCHECKINGACCOUNTSCLASS中的对象也是其父类CHECKINGACCOUNTSCLASS和ACCOUNTSCLASS的对象，所以对ACCOUNTSCLASS的加锁就隐含着对CHECKINGACCOUNTSCLASS和ECONOMYCHECKINGACCOUNTSCLASS中的所有对象加锁。同样地，在我们对ECONOMYCHECKINGACCOUNTSCLASS类加锁前，我们必须对CHECKINGACCOUNTSCLASS类和ACCOUNTSCLASS类都施加合适的意向锁。因此，对一个类加锁，就意味着对下列内容加锁:

- 该类的所有对象。
- 该类的所有子孙类（因而也包括那些类中的所有对象）。

我们现在从上述讨论中归结出对象数据库的粒度加锁的一个（简化的）协议[⊖]。

⊖ 一些DBMS也许允许不同的粒度——例如，属性级（一个对象的个别属性），或者数据库级。在某些DBMS中，一个类的写锁允许程序修改该类的声明，包括它的方法。其他的一些DBMS会区分对类实例加锁（类似于某个表锁）与对类本身加锁（类似于某个模式锁），前者类实例指当前在该类中的所有对象，而后者类本身则指允许修改该类的定义。

对象数据库的粒度加锁协议

- 在给对象加锁之前，系统必须先对该对象所属的类及其所有父类施加合适的意向锁。
- 在给类加锁之前，系统必须先对该类的所有父类施加合适的意向锁。

有了这些观念后，我们就会发现，对关系数据库的隔离性和粒度加锁的许多讨论同样也可应用于对象数据库。

24.4 系统性能的改进

性能是系统设计的关键问题。在本节中，我们会给出一些技术的例子，这些技术可以用来提高在加锁并发性控制下运行的应用程序的性能。

- 事务应该在与应用程序需求一致的最低的隔离级别执行。
- 对在模式中包含完整性约束加以权衡，所以必须仔细考察DBMS强制的规定与事务中执行强制的代码。例如，修改某个在约束中命名的数据项的一些事务，可以用不会违反约束的方式去修改它，然而，如果约束是模式的一部分，那么它将在该事务提交时（不必要地）进行检测。如果这样的事务需要频繁执行，最好限制一下对事务代码的约束检测，因为事务代码也许会违反某种约束。另一方面，这样一个决策要和潜在的维护消耗相互比较考虑：如果我们稍后必须要修改此约束，那么所有检测该约束的事务的代码也必须改变并重新编译。但如果约束是模式的一部分，那么这些就没有不必要了。
- 某些完整性约束应该在数据库模式内加以声明，以便它们会自动地由DBMS检验，然后允许某个事务在比应用程序需求相一致的更低的隔离级别上正确执行。（如果那些完整性约束不是由DBMS检验的，该事务按照那个隔离级别执行可能是不正确的。）这基本上是一种乐观型方法，它假设了某种程度上的交叉，这种交叉不大可能造成数据库的不一致性。万一出现这样的交叉情况（极其难得），DBMS将在侦测到违反约束时异常中止该事务。然而，不是由违反完整性约束所导致的错误是侦测不到的。
- 事务应该尽可能地短，以限制保持锁的时间。特别重要的是，在初始化事务前，需要从用户处收集所需的信息。由于与用户交互要花费很长的时间，因此在其进行过程中，不应该加锁。把一个较长的事务分解成一系列较短的事务（假设可以在维持一致性的情况下完成的话）也是不错的。在极端的情况下，每个SQL语句就是一个事务。
- 设计数据库，以便使最频繁调用的事务能够有效地执行。这可以包括反规范化（denormalization）（参见8.13节），以避免大量的联结。
- 在频繁执行搜索时应该考虑索引。在某些情况下，这意味着为某个表建立若干个二级索引。尽管在表经常更新的情况下，索引的维护开销也要被考虑进来。为了避免多重索引，索引应该能够支持尽量多的搜索。在B^+树索引中，这意味着要仔细地选择和排列搜索键的属性。对于幻影敏感的应用程序来说，采用索引锁而不是表锁，可能会增加并发性。
- 若逐步升级的阈值很容易达到，经常需要表锁，那么这种锁的逐步升级是低效的。有些数据库允许事务在访问表之前明确地申请表锁（手动加锁）。作为一种选择，如果需要的页面（或者元组）锁的数量可以估计出来，而且也不是很大，那么阈值就可以设置成高于该值。

- 加锁粒度可能会下降，因此通过对一个或者多个表分段，往往可以增加并发性。
- 在使用页面加锁的系统里，如果两个事务分别访问存储在同一个页面上的两个不同的元组，就会发生加锁冲突。把这些元组存储于不同的页面会降低这种冲突发生的概率。同样，如果某个事务访问一系列的元组，即便所有那些元组都聚合于少量页面内，但和其他事务的加锁冲突也可能降低。
- 如果一个事务以某种顺序访问两个表，而另一个事务以相反的顺序访问它们，那么就可能会产生死锁。只要可能，访问公共资源的事务都应该以相同的顺序对那些资源加锁。

24.5 多版本并发控制

数据库的**版本**（version）就是指在某个事务提交时所得到的一个数据库的快照，其中包含该事务的结果和所有先前提交的事务的结果。因此，版本只包含已提交的数据。一个数据库的许多版本都是在事务的某个调度的执行过程中产生的。在多版本DBMS中，不同的版本都被保留下来，而并发控制也未必使用当前的版本来满足读取某个数据项的请求。

在本节中，我们将讨论三个多版本并发控制。这些算法都已在商用系统中得到实现，它们的优势在于，（在绝大多数情况下）读取者无须设立读锁。因此，读取某个数据项的请求不需要等待；而写入某个数据项的请求也不需要等待读取者。这是一个很重要的优势，特别是在许多应用程序中，读取发生的频率要比写入高得多。这些优势的代价是高昂的，为了维持数持数据库的多版本，就需要额外地提高系统的复杂性。

在我们讨论的三个算法中，后面两种算法可能会产生不可串行化的调度，从而导致不正确的数据库状态。第一种算法总是产生可串行化调度，但是行为可能不太直观。

事务级读一致性

在指定一个多版本并发控制时，必须确定的第一个问题就是，“对一个请求读取数据库中的某个数据项的事务而言，它返回的是怎样的值？”当采用READ COMMITTED隔离级别时，多版本算法确保返回的只能是已提交的数据（因为根据定义，一个版本所包含的只能是已提交的数据）。但回忆一下，在READ COMMITTED级连续的读取可能会从不同的版本返回数据，而且（根据我们在24.2.1节的讨论来看），一旦使用了某个游标，那么从某个SELECT语句产生的结果集所返回的元组很有可能来自不同的版本；哪怕该结果集是在此游标打开时，就以某种隔离级别计算出来的也是如此。因此，该事务会看到数据的某种不一致的视图，即一种并非来自某个数据库版本的视图。

有些多版本算法确保通过这类查询所返回的数据具备某个较强的条件。**事务级读一致性**（transaction-level read consistency）确保由某个事务执行的所有SQL语句返回的数据都来自数据库的同一版本。然而，事务级读一致性未必能确保可串行性。

另一个必须确定的问题是，“一个SQL语句所访问的是数据库的哪个版本？”一个多版本并发控制可能需要访问的版本不是最近提交的事务所产生的版本。例如，假设事务T_1和T_2在某个常规（单一版本）的立即更新悲观系统中是活动的。如果T_1写入了某个数据项，恰好T_2打算读取该数据项，那么就存在冲突，于是T_2就需要等待。在一个多版本系统中，利用在T_1写入之前就已创建的某个版本（值得注意的是，这不需要是最近提交的版本），T_2的请求可能会得到满足。因此，按照任何等价的串行顺序，T_2都先行于T_1。

24.5.1 只读型的多版本并发控制

在一般情况下，设计一种能够确保可串行化调度的多版本并发性控制是相当复杂的。然而，有一种特殊的多版本控制的情况，称为**只读型的多版本并发控制**（read-only multiversion concurrency control），它可以比较容易地实现，并产生出可串行化的调度。

一个只读型的多版本并发控制事先需要区别两种类型事务：**只读型**（read-only）（即不包含写操作）和**读/写型**（read/write），即至少包含一个写操作（也可以包含读操作）。

- 读/写型事务采用立即更新的悲观并发控制。读锁和写锁可以有多种管理方法，这取决于所选择的隔离级别。事务总是访问需要读/写的数据项的最新版本。
- 只读型事务T_{RO}中的所有读取，则是采用T_{RO}首次发出读取请求时就存在的数据库的（已提交的）版本来满足其需求的。因此，T_{RO}直接就是可串行化的，而那个版本所创建的事务，未必都是遵循着提交的顺序的。其效果就像是在T_{RO}第一次读取时所获得的数据库的（已提交的）一个**快照**（snapshot）；而所有随后的读取都可以利用此快照来满足。因此，只读型事务是可以提供事务级读一致性的。

如果读/写型事务可以串行执行，那么所有的事务都可以提供事务级读一致性。在这种情况下，只读型和读/写型事务的组合调度是可串行化的。等价的串行顺序是按照读/写型事务的提交顺序排列，而每个只读型事务都插入到创建其快照的读/写型事务的后面。

为了实现这一控制，DBMS维护该数据库的多个版本。我们将在第25章中看到，DBMS一般都在其日志里保存着版本信息，以供恢复使用。因此，维护这一信息并不仅仅是为了多版本系统。不过，多版本系统还有其他的需求，就是能以一种有效的方式访问较早的版本。

问题在于如何来提供合适的版本来满足某个读取请求。为了做到这一点，系统使用一种延迟更新的技术，并维护一个所谓的**版本计数器**（Version Counter，VC），每当某个读/写型事务$T_{R/W}$提交时，它就会递增。这时，每个被修改的数据项的新版本就会存入该数据库中，并将加入版本号N，这个N值是从（递增后的）VC获得的。此数据项的旧版本仍然（也许在日志里）保存着。$T_{R/W}$创建了该数据库的一个新版本（即快照），标有快照号N，其中包括被更新的数据项的新版本和其他数据项的最新版本（在$T_{R/W}$提交时刻）。快照包含的都是已提交的值。每个只读型事务赋予快照的是其第一次请求读取时的值，而所有后来的读取请求都可以利用从该快照所获得的值得到满足。

图24-11解释了这种情况，其中显示了数据库的三个数据项x、y和z。每个数据项都标记着一个版本号。T_{RO}在VC值为4的时候提出它的首次读请求。因此，它的快照号是4，它当即得到了虚线上方的该版本的x、y和z。在T_{RO}正在执行时创建了版本5和6，但是T_{RO}是无法看到它们的。

存储每个数据库项的版本号都会产生管理开销。此外，考虑到实用性，可以访问的早期版本的数目也许会有限制；因此，对于从很陈旧的版本抽取信息的某个(长时间运行的)只读型事务来说，若那个版本不可以再使用的话，该事务不得不异常中止。

该控制具有一个非常好的特点，那就是只读型事务不需要获得任何锁。正是由于这个原因，只读型事务从不需要等待，而读/写型事务也从不需要等待只读型事务。这一特点的代价是产生更加复杂的并发控制，维护带版本号的多版本需要额外的存储空间，以及由于只读型事务的可串行化顺序不同于提交顺序会产生有些不自然的行为。例如，一个报告银行账户余

额的只读型事务可能比某个银行存款事务的提交时间晚，但是它不会报告存款的结果，因为它在存款事务开始之前就已执行了它的首次读取。

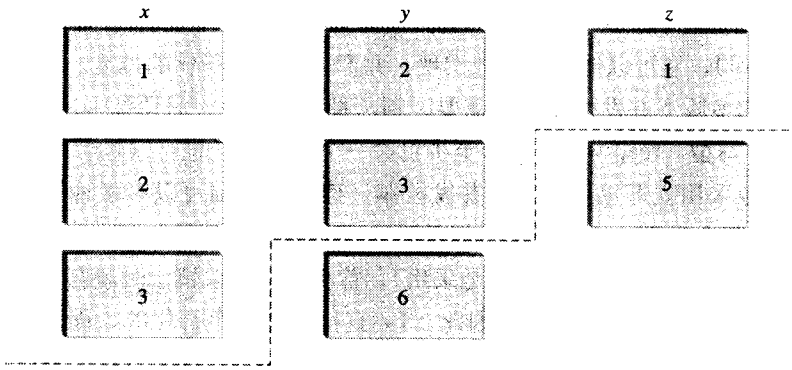


图24-11　在多版本数据库中满足某个读取请求

24.5.2　读取一致性的多版本并发控制

对于可以容忍不可重复读取（因此也是不可串行化调度）的应用程序来说，一些商用DBMS（比如Oracle）都采用一种称为**读取一致性**（Read-Consistency）的算法，它拓展了处理读/写型事务的只读型控制。

- 对于只读型事务的处理与在只读型控制中一样，因此也提供事务级的读一致性。
- 读/写型事务里的写入语句对于被写入的数据项，则是对其最新版本使用长期写入锁。某个试图改写已被其他事务施加写锁的数据项的事务，则必须等待。
- 读/写型事务里的读取语句不需要使用读锁。相反，每个读请求都是由被请求项的最近版本值提供的。

读一致性供了READ COMMITTED隔离级别的一个扩展的实现（同时也是Oracle提供的READ COMMITTED的实现）。正如24.2节给出的READ COMMITTED级的加锁实现一样，写锁是长期的，而且读取返回的是已提交的值。然而，读一致性对于只读型事务所提供的是事务级的读取一致性，而这却是READ COMMITTED级的加锁实现所不能提供的（也不是READ COMMITTED的ANSI定义所要求的）。

读一致性的一个很好的特点是，任何事务都不需要对读操作加锁。因此，读取从不需要等待写入，而写入也从不需要等待读取。就像在READ COMMITTED级那样，由读/写型事务执行的读取是不可重复的；因此，调度有可能是不可串行化的。例如，图24-6的调度就显示了这种控制有可能会产生的更新丢失问题。

24.5.3　SNAPSHOT隔离级别

相同思想的还有另一种变形方案，称之为SNAPSHOT隔离性[Berenson et al. 1995]。包括Oracle在内的一些数据库供应商已经实现了SNAPSHOT隔离性的变形方案。该方法是建立在以下两个原则的基础上的:

- 每个事务的所有读取都是利用该事务首次提出读取请求时的数据库的快照来满足的。因

此，所有的事务都提供了事务级的读一致性。

- 每个事务的写入必须满足**先提交者赢**（first-committer-wins）的原则。事务T_1只有当没有任何其他事务会在T_1第一次提出读取请求与要求提交之间请求提交，或者更新了某个T_1也对之更新的数据项的时候，才允许提交。否则，T_1就会异常中止。数据项既可以是一个行，也可以是一张表。但由于表对确定事务之间的交互所提供的是较粗的粒度，这就会对性能产生负面的影响，所以今后我们总是假设数据项是行。

先提交者赢原则对消除更新丢失起着重要作用。在图24-6中，T_1是不允许提交的，因为它要求更新数据项t，可是T_2对t的更新和提交请求，却是在T_1首次请求读取后和T_1请求提交前提出的。

先提交者赢原则可以在没有写入锁的情况下实现。一旦某个事务T_1完成，就验证是否强制贯彻先提交者赢原则（就像在某个乐观型并发控制里一样，但却是另外的一种验证标准）。验证可以通过比较T_1的快照号和T_1更新过的每个数据项的版本号来实现。

- 假设当T_1请求提交时，某些T_1更新的数据项的版本号比T_1的快照号要大。这就意味着，存在事务T_2，它在T_1的快照产生之后写入了那些数据项，并已提交了。在这种情况下，T_1必须异常中止，因为T_2是先提交者，所以它赢。
- 假设当T_1请求提交时，所有T_1写入的数据项的版本号都小于或等于T_1的快照号。只有在这种情况下，才允许T_1提交。

正如图24-11所显示的，如果T_1的快照号是4，而T_1已经写入了x和y，那么T_1的提交的请求会遭到拒绝。因为尽管x还没有更新的版本，可是y的一个更新的版本已被在T_1执行时就已提交的另一个事务创建出来了。

与采用乐观型并发性控制算法一样，这种控制具有不需要任何锁的特性。因此，无论是读取还是写入都不需要等待，但事务也可能在其任务完成时异常中止。

尽管SNAPSHOT隔离性消除了很多异常，可是它不能确保所有的调度都是可串行化的，因此，事务可能会不正确地执行。例如，在图24-12中，T_1和T_2是从同一个储户d所拥有的余额为a_1和a_2的两个不同的账户取款的银行取款事务。银行有一条业务规则：个人账户的余额可以是负的，但是每个储户所拥有的所有账户的总余额必须是非负的。因此，如果d有两个账户，就有$a_1+a_2>0$的约束。T_1和T_2都是一致的：在进行取款前，它们都要读取两个账户中的余额；所以，当单独执行时，它们都符合该约束。在此例中，每个账户最初都有\$10，因此，每个事务都得出“取款\$15是安全的”结论。然而，正如我们所看到的，该调度是允许SNAPSHOT隔离性的（因为T_1和T_2是对不同的数据项写入），可是最终a_1和a_2的值都是\$-5，这是违背约束的。请注意，这种调度不是可串行化的。因为一方面T_2必须在T_1的后面（因为T_2写入a_1在T_1读取a_1之后发生），而另一方面T_1又必须在T_2的后面（因为T_1写入a_2是在T_2读取a_2之后发生）。（这个例子的后续讨论，请见练习24.27。）

T_1:	$r(a_1:10)$	$r(a_2:10)$			$w(a_2:-5)$	提交
T_2:			$r(a_1:10)$	$r(a_2:10)$ $w(a_1:-5)$	提交	

图24-12　不可串行化并导致某个不一致数据库的SNAPSHOT隔离性调度

1. SNAPSHOT隔离性与隔离级现象

尽管图24-12的例子显示，SNAPSHOT隔离性可能会产生不可串行化（因而不正确）的调度，但值得注意的是，SNAPSHOT隔离级的调度并不展示任何与较低隔离级有关的坏现象——脏读、不可重复读和幻影（还有不属于隔离级定义的脏写和更新的丢失）。

必须澄清的是，SNAPSHOT隔离性确实不允许幻影。在SNAPSHOT隔离级调度中，一个事务T可能执行基于某个谓词的SELECT语句，而一个并发事务也有可能插入满足那个谓词的某个元组t（这似乎是一个幻影）。可是，T是看不见t的，哪怕它在插入之后立即再读取同一个谓词。因为它的所有的读取都是通过T第一次读取时的快照来得到满足的。正是基于这一原因，我们可以说，t不是幻影。我们给出两个例子，一个是插入这样的一个元组并不会导致不正确的调度，而另一个则会导致不正确的调度。

- 24.1.1节的幻影的例子涉及Mary和她的账户，该例子在SNAPSHOT隔离级别是可以正确执行的。在该例中，将TotalBalance（总余额）与Mary的所有账户总余额进行比较的事务T_1看到了不一致的数据，因为有一个幻影插入到它的两个SELECT语句中间。然而，SNAPSHOT隔离级别能够确保事务级的读一致性。T_1看到的是一个与T_1刚开始时的一样的一致的数据库快照。它确实看不到幻影，因此也不会看到更新过的TotalBalance值。所以，它可以正确地执行。
- 假设银行数据库有一个完整性约束（与银行的某个业务规则相对应），即任何储户不能拥有10个以上的账户。为了保证这个约束，add_new_account事务总是先使用谓词Name = 'Mary'执行一个SELECT语句，以确定Mary的账户数目。如果该数目小于等于9，它便会插入与Mary的新账户相对应的一个元组。现在假设add_new_account事务有两个实例T_1和T_2，它们在SNAPSHOT隔离级别并发执行，Mary的最初账户数为9。由于每个事务都确定账户数为9，因此都插入了与新账户相对应的一个元组。于是Mary现在有了11个账户，这和完整性约束是冲突的，所以调度是不可串行化的，也是不正确的。

对于幻影的组成，在文献中还没有一致的定义。有些资料认为，如果某个事务两次执行同一个SELECT语句，第二次执行可能返回一个包含有某个幻影元组的结果集合，而这个元组在第一次执行返回的结果集合中却没有，那么隔离级别允许幻影。采用这样的定义，幻影在REPEATABLE READ级是允许的，但在SNAPSHOT隔离级是不允许的。因为执行同一个SELECT语句两次的事务总是会获得同样的结果集（因为两个SELECT语句访问的是同一个快照）。

然而，上述第二个例子说明，幻影的影响在SNAPSHOT隔离级还是存在的，虽然根据以上的定义，插入的确没有构成幻影。倘若我们在那个例子中在REPEATABLE READ级执行该事务，就会允许出现同样的（不可串行化的）调度，因此我们说幻影确实存在。

一旦采用基于相继执行SELECT语句的幻影定义，就允许SNAPSHOT隔离级不展示较低层的隔离级所定义的任何不好的现象。然而，它并不满足ANSI定义的SERIALIZABLE[SQL 1992]，该定义认为，SERIALIZABLE必须提供“在完备的可串行化执行场合里众所周知的”那些东西（此外还不允许以下三种现象中的任何一种）。这当然不是SNAPSHOT隔离级别的情况。不包含脏读、不可重复读和幻影三种现象的调度，有时候称为**异常可串行化**（anomaly serializable）。因此，SNAPSHOT隔离级的调度是异常可串行化，但不一定是可串行化的。

这再次表明，执行的正确性不应该通过有没有出现某一特定的现象来定义。有些作者采用**写入偏差**（write skew）的说法来描述图24-12所例举的不好现象，但是即便把那些现象都添加到列表中，也不能认为这样的列表就是完备的。例如，图24-13中的调度在SNAPSHOT隔离级别是允许的，但它是不可串行化的，因为它的**可串行化图**（serialization graph）有一个循环：

$$T_3 \rightarrow T_2 \rightarrow T_1 \rightarrow T_3$$

然而，这个调度却没有展示出任何命名过的现象，包括（通常定义的）写入偏差。

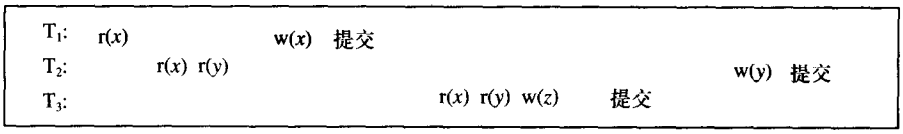


图24-13　不可串行化但并不展示任何已命名异常情况的SNAPSHOT隔离级调度

2. SNAPSHOT隔离级别中的冲突

我们在第23章讨论带有冲突的操作时强调过，在读取和写入操作之间有三种冲突形式（尽管当时我们并没有对它们进行命名）。

- 读/写型冲突　T_2写入了T_1先前已读取的某个数据项，因此无论按什么顺序，它都必须排在T_1的后面。
- 写/读型冲突　T_2读取了T_1先前已写入的某个数据项，因此无论按什么顺序，它都必须排在T_1的后面。
- 写/写型冲突　T_2写入了T_1先前已写入的某个数据项，因此无论按什么顺序，它都必须排在T_1的后面。

与其他隔离级别有关的任何现象在SNAPSHOT隔离级别所产生的调度中不会展示的一个原因是：在并发执行的事务中可能出现的唯一冲突是读/写型冲突（在彼此不是并发执行的事务中间，还允许出现其他类型的冲突）。

- 写/读型冲突不可能发生，因为T_2是通过它开始时所得到的数据库快照来读取数据项值的。由于T_2和T_1是并发执行的，因此那个快照确实不包含T_1写入该数据项后的值。
- 写/写型冲突不可能发生，这是由于先提交者赢特性的缘故。

在图24-12的调度中，有一个写入偏差，

- T_1与T_2有一个读/写型冲突，因为T_1是在T_2读取后才写入a_2的。
- T_2与T_1有一个读/写型冲突，因为T_2是在T_1读取后才写入a_1的。

图24-13的调度更加有趣。

- T_1与T_2有一个读/写型冲突，因为T_1在T_2读取后才写入x。
- T_2与T_3有一个读/写型冲突，因为T_2在T_3读取后才写入y。
- T_3与T_1有一个写/读型冲突，因为T_3在T_1写入后才读取x。

最后的这种冲突，同并发事务中间不能发生写/读型冲突的说法并不矛盾。因为T_3与T_1不是并发执行的，它是在T_1提交之后才开始的。

这两个例子可以概括地说明如下[Fekete et al. 2000]：在每个不可串行化的SNAPSHOT隔离级调度中，其串行图中包含环。这个环中至少包含两个并发执行的事务之间的读/写型冲突，

也可以包含非并发执行的事务之间产生的一些读/写型、写/读型或者写/写型冲突。

3. 在SNAPSHOT隔离级别中正确执行

注意，在24.2.2节给出的四个例子都是可以在SNAPSHOT隔离级别正确运行的。

- 一个打印TRANSCRIPT的事务能够正确工作，因为它是只读的，并可看到该数据库的某个快照。
- 注册事务T能够正确地工作，因为
  - 如果某个事务T'在T第一次发出读取请求后修改REQUIRES或者TRANSCRIPT，那么它所产生的最终调度等价于串行调度T，T'。
  - 如果某个并发的注册事务试图给学生注册同一门课程，它会更新CLASS的同一行，但只有一个注册事务允许提交。
- 重新分配教室的事务T能够正确地工作，因为
  - 如果某个事务T'在T第一次发生读取请求后修改CLASS或者CLASSROOM，那么它所产生的最终调度等价于串行调度T，T'。
  - T不可能干涉并发注册事务，因为两者都是更新CLASS的同一个行，而只有一个允许提交。
- 审计事务可以正确地工作，因为它是只读型的，并能够看到该数据库的某个快照。

一个应用程序在SNAPSHOT隔离级别总是能够正确运行，即便它的一些调度出现了写入偏差，或者是不可串行化的。例如，考虑某个包含一个音乐会座位预订事务的应用程序。该事务检查座位的数目，并预订一个座位。完整性约束规定：同一个座位不能被多个人预订。假设两个票务预订事务并发执行，并产生如图24-14所示的调度（事实上等价于图24-12的调度）。每个事务都读取对应于座位s_1和s_2的元组，并确定它们都未被预订(U)。然后，T_1通过在数据库中将其状态更新为R来预订s_1；T_2也用同样的方式预订s_2。对于该应用程序来说，这一调度是正确的，尽管出现了某个写入偏差（因此该调度不是可串行化的）。此外，由于先提交者赢的原则在维护着完整性约束，所以，如果两个事务都想要预订座位s_1，那么只有其中的一个事务会提交。因此，票务预订事务的任何调度在SNAPSHOT隔离级都能够正确地执行。

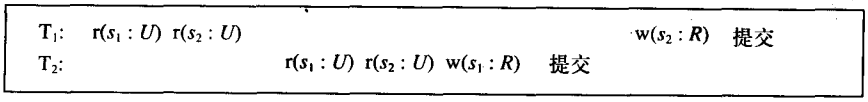


图24-14　某票务预订应用程序的SNAPSHOT隔离级调度。它展示了写入偏差，是不可串行化的，但却是正确的

由于SNAPSHOT隔离级的多版本特点，可串行化SNAPSHOT隔离级调度有时候可能会产生不太直观的行为。例如，图24-15的调度按照顺序

$$T_3 \rightarrow T_2 \rightarrow T_1$$

是可串行化的，其中T_3先于T_1（即使它在T_1提交后才开始）。

事实上，许多应用程序在SNAPSHOT隔离级都能正确运行，倘若绝大多数的完整性约束以代码形式融入到数据库模式里的话（见练习24.27），那就更是如此。然而，谨慎的设计者在制定设计决策之前，都会进行仔细的分析。

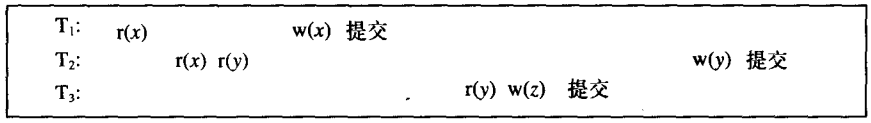


图24-15　可串行化的SNAPSHOT隔离级调度，尽管T_3在T_1提交后开始，但在等价串行顺序中却先于T_1

24.6 参考书目

幻影与利用谓词锁消除幻影是在[Eswaran et al. 1976]里介绍的。SQL隔离级别的定义可以在[Gray et al. 1976]和ANSI SQL标准[SQL 1992]中找到。隔离级的加锁实现在[Berenson et al. 1995]中讨论。在[Gray et al. 1976]里包含关于粒度加锁的详细论述。多版本并发控制在[Bernstein and Goodman 1983，Hadzilacos and Papadimitriou 1985]中加以介绍。多版本乐观型并发控制的设计在[Agrawal et al. 1987]中有所描述。SNAPSHOT隔离级是在[Bernstein et al. 2000]中首先提出的。[Fekete et al. 2000]中讨论了SNAPSHOT隔离级的冲突问题，并给出了SNAPSHOT隔离级别的调度为可串行化的一个充分条件。[Bernstein et al. 2000]还讨论了基于事务的语义证明低隔离级调度正确性的一种方法。[Bernstein et al. 1987]对包括索引加锁在内的许多并发控制算法进行了精辟论述。关于对象数据库的并发性问题可参见[Cattell 1994]。

24.7 练习

24.1　假设学校的事务处理系统包含一张表，每个目前已注册学生都对应于该表中的一个元组。

a. 估计一下，要存储这张表需要多大的磁盘空间？

b. 请给出在对表进行并发控制场合，必须对整个表加锁的事务的例子。

24.2　请给出在一组记录组成的非关系数据库里出现幻影的例子，该数据库中每个记录包含若干字段。

24.3　对于比SERIALIZABLE级别低的每个隔离级别，请对曾经遇到过的产生出错的情况，给出事务处理系统的排序的例子(不能是银行或学生注册系统)。

24.4　假设事务在REPEATABLE READ级执行。请给出出现幻影的事务例子，其中事务执行着用WHERE子句来指定主键值的某个SELECT语句。

24.5　假设事务在REPEATABLE READ级执行。请给出出现幻影的事务例子，其中事务执行着删除满足某谓词P的若干元组的某个DELETE语句。

24.6　假设事务在REPEATABLE READ级执行。请给出出现幻影的例子，其中事务T_2执行的UPDATE语句将导致事务T_1执行UPDATE语句会产生一个幻影。

24.7　假定每个表都拥有三个属性attr1、attr2和attr3的两张表TABLE1与TABLE2组成一个模式，并假定有如下语句：

```
SELECT T1.attr1, T2.attr1
FROM TABLE1 T1, TABLE2 T2
WHERE T1.attr2 = T2.attr2 AND T1.attr3 = 5
        AND T2.attr3 = 7
```

请给出将产生幻影的一个INSERT语句的例子。

24.8　试述不可重复读与幻影之间的区别。特别地，请给出能阐释这两种情况的若干事务的SQL语句的调度的例子。指定每种情况的隔离级。

24.9　请给出采用SELECT语句的应用例子，要求其中应用的语义蕴涵着不可能出现幻影。

24.10 我们在24.2.2节里提到，在READ COMMITTED级执行的注册事务不会干扰在REPEATABLE READ级执行的教室重分配事务的正确性。请说明原因。

24.11 在实现意向锁的DBMS里，请说明为什么两个不同的事务可以同时对同一个表施加IX锁，而不会引起冲突？

24.12 请说明，一个事务所请求的对表施加的谓词锁，在什么条件下会由于另一个事务拥有对同一张表的谓词读锁而遭到拒绝？

24.13 对于实现隔离级的每种锁，请叙述为什么需要有IS锁以及何时可以将其释放？

24.14 下述过程是为获得读锁而提出的：

当一个SQL语句读取某表中满足某谓词的元组集时，系统首先对包含这些元组的表施加IS锁，然后再对其中每一个元组施加S锁。

请解释，为什么这个过程允许出现幻影？

24.15 请给出通过一个只读多版本并发控制所产生的调度的例子，要求其中的读/写事务是按其提交的顺序串行化的，然而其只读型事务却是按不同的顺序串行化的。

24.16 请给出一个可以为多版本并发控制所接受的读/写请求调度的例子，其中事务T_1在事务T_2提交之后才开始，但按串行顺序T_1却在T_2之前。这样的调度可能会有以下的违反直觉的行为（尽管它是可串行化的）：你向自己的银行账户里存钱；提交你的事务；后来你又开始了一个新的事务，以读取账户里的总额，结果却发现你存的钱没有在总额中反映出来。（提示：此调度允许包含有附加的事务。）

24.17 请说明，图24-12所示的在SNAPSHOT隔离级不正确执行的调度可能在OPTIMISTIC READ COMMITTED级执行（见24.2.1节）时出现，它也将是不正确的。

24.18 请给出在以下隔离级别可以接受的调度的例子：

a. SNAPSHOT隔离级别，但不是REPEATABLE READ级。

b. SERIALIZABLE级，但不是SNAPSHOT隔离级别（提示：在T_1提交后T_2完成一次写。）

24.19 一个特定的只读型事务读取上个月输入数据库里的数据，并用这些数据来准备一份报告。这个事务可以最低可以在哪种隔离级别执行？请加以说明。

24.20 请说明，由使用某个INSENSITIVE游标的单个SELECT语句组成的只读型事务，为什么总是能够正确地在READ COMMITTED级执行？

24.21 假设REPEATABLE READ级的加锁实现要求事务在请求写入时，对某表施加X锁；当请求读取时，要求事务对该表施加IS锁，以及对返回的元组施加S锁。请说明，这时幻影不可能出现。

24.22 考虑对行和表采用长期粒度锁所实现的一种隔离级别。在什么条件下有可能出现幻影？

24.23 a. 请给出两个事务调度的一个例子，其中两阶段加锁并发控制会造成一个事务等待，但SNAPSHOT隔离级控制则会使一个事务异常中止。

b. 请给出两个事务调度的一个例子，其中两阶段加锁并发控制会造成一个事务异常中止（因死锁缘故），但SNAPSHOT隔离级控制则允许两个事务都提交。

24.24 请说明，为什么SNAPSHOT隔离级调度不会出现脏读、脏写、丢失更新、不可重复读以及幻影？

24.25 考虑由不同的隔离级的事务所组成的一个应用。请证明，SERIALIZABLE级的事务相对其他事务来说总是可以串行化的。换句话说，就是证明，对于任意一个SERIALIZABLE级事务T和其他任何事务T'，两个事务的任何调度都等同于，T的所有操作要么都在T'的所有操作的前面，要么都在T'的所有操作的后面。

24.26 在某些特殊的应用里，所有的事务都将其读取的全部数据项改写。请证明，如果这些应用是在SNAPSHOT隔离级下执行的话，那么已提交事务的所有调度都是可串行化的。

24.27 考虑如图24-12所示的两个银行取款事务的调度，其中SNAPSHOT隔离级将会导致不一致的数据库。假设银行编制事务规则“同一个储户所拥有的全部账户的余额的总和必须为非负”，并将其作为数据库模式的完整性约束。那么这种特殊的调度是不可能出现的。

虽然现在保证了完整性约束，但特殊事务的规格说明还规定，当事务提交时，该数据库的状态会满足某个更严格的条件。请给出更严格条件的一个例子，它是某个取款事务在终止时强行施加的，并给出两个事务在SNAPSHOT隔离级运行就可能使它们的行为不正确的调度的例子。

24.28 已有人提出了如下的多版本并发控制:

在事务第一次提出读取请求时利用数据库已经存在的（已提交）版本，读取能够得到满足。而写入则是受到表的长期写锁的控制。

这样控制是否始终产生可串行化调度？若答案否定，请给出一个可能产生不可串行化调度的例子。

24.29 我们已经给出过实现READ COMMITED隔离级的两种不同的办法：一是在24.2节的加锁实现，二是24.5节的读取一致性实现。请给出一个调度的例子，其中这两种实现会产生出不同的结果。

24.30 粒度加锁协议能够杜绝如下两个事务之间的死锁，其中一个事务执行某个SELECT语句，而另一个事务则执行某个UPDATE语句。例如，假设一个事务包含如下的SELECT语句:

```
SELECT COUNT (P.Id)
FROM EMPLOYEE P
WHERE P.Age = '27'
```

它返回年龄为27的雇员人数，而另一个则包含如下的UPDATE语句:

```
UPDATE EMPLOYEE
SET Salary = Salary * 1.1
WHERE Department = 'Adm'
```

它给管理部门的所有雇员加薪10%。假定Department和Age都存在着索引，而与部门Adm对应的元组，同年龄27所对应的元组一样，都是存放在多个页面上的。

请说明，在非READ UNCOMMITTED级上为何可能出现死锁?

24.31 请给出在SNAPSHOT隔离级执行的调度的一个例子，其中的两个事务都会引入另一个事务看不到的幻影，从而产生不正确的行为。假设在SNAPSHOT隔离级描述里所说的数据项是行。

24.32 在某个Internet选举系统里，每个投票者都以邮件形式发送PIN(个人身份识别号)。当投票者需要在某个Web站点投票时，她就输入其PIN和投票，然后就会执行某个投票事务。

在该投票事务里，首先检查PIN，以证实它是有效的，并非是已经使用过的。然后将投票累计到相应的候选人的积分榜上。此时用到两张表，一张表包含有效的PIN，同时附带着该PIN是否已被用过的指示标记，而另一张表则包含候选人的名字，以及每人的投票积分榜。请讨论为投票事务选择合适的隔离级所涉及的问题。倘若引入一个新的只读型事务来输出整个投票积分榜表，请讨论选择合适的隔离级时会涉及的问题。

24.33 某个航空订票数据库有两张表：一张表为FLIGHTS，具有属性flt_num、plane_id、num_reserv；另一张表为PLANES，具有属性plane_id、num_seats。

这些属性都有直观的语义。预订事务包含以下步骤:

```
        SELECT F.plane_id, F.num_reserv
        INTO :p, :n
        FROM FLIGHTS F
        WHERE F.flt_num = :f
A.      SELECT P.num_seats
        INTO :s
        FROM PLANES P
        WHERE P.plane_id = :p
B.      ...check that n < s...
C.      UPDATE FLIGHTS F
        SET F.num_reserv = :n + 1
        WHERE F.flt_num = :f
D.      COMMIT
```

假定每一个SQL语句都在一定的隔离级执行，DBMS使用意向锁，并且对表和行加锁，而宿主变量f则包含被预订的航班号。该事务应当不会额外预订航班。

a. 假定事务在READ COMMITTED级运行，在A、B和D点应持有哪种锁？

b. 倘若并发执行在READ COMMITTED级运行的预订事务以某种方式交叉，即一个事务要等到另一个事务执行到B点完成才能开始执行，那么数据库会处于不正确的状态。请描述其存在的问题。

c. 为了避免b中指出的问题，将UPDATE语句里的SET子句改为

F.num_reserv = F.num_reserv + 1

现在预订事务是否正确地在READ COMMITTED级运行？为什么？

d. 假定事务在REPEATABLE READ级运行，而表是通过索引访问的，那么在A、B和D点应对表施加哪种锁？

e. 倘若在REPEATABLE READ级运行，b中所说的交叉会导致什么问题？

f. 若事务（两个版本）都采用SNAPSHOT隔离级运行，那么b中所说的交叉是否也会导致不正确状态？为什么？

g. 为了跟踪每位乘客，引入一张新的表PASSENGER，它用一行描述每个航班的每位乘客，其中包括属性name、flt_num和seat_id。在事务的最后，添加SQL语句，以便完成以下任务：(1) 读取f所规定的航班的每位乘客的seat_id，(2) 为每位新乘客插入一行新记录，它为该乘客分配一个空座位。使得该事务不会产生不正确状态（即两位乘客占据同一座位）的最低的ANSI隔离级是什么？为什么？

24.34 两个事务并发地运行，其中每个事务既可能提交，也可能异常中止。事务如下：

T_1: $r_1(x)$ $w_1(y)$

T_2: $w_2(x)$

T_3: $r_3(y)$ $w_3(x)$

T_4: $r_4(x)$ $w_4(x)$ $w_4(y)$

针对以下的每一种情况，请指出其最终调度是否总是可串行化和原子化的(yes或no)？

a. T_1和T_2都在READ UNCOMMITTED级运行。

b. T_2和T_3都在READ UNCOMMITTED级运行。

c. T_1和T_2都在READ COMMITTED级运行。

d. T_1和T_3都在READ COMMITTED级运行。

e. T_1和T_3都在SNAPSHOT隔离级运行。

f. T_1和T_4都在SNAPSHOT隔离级运行。

第25章 原子性和持久性

在以前的各章里，我们一直作出不切实际的假设：事务总是提交而且系统永远不会失灵。现实是完全不一样的：事务可能会由于各种原因而异常中止，而硬件和软件也会有故障。这类事件必须小心应对，以确保事务的原子性。此外，海量存储设备很容易出现故障，造成已提交事务写入数据库的信息丢失，从而威胁到持久性。

在本章里，我们探讨为了实现原子性和持久性所必须解决的基本问题，以及实现原子性和持久性的一些技术。我们的目的并不是介绍任何特定的故障恢复系统的设计。相反，我们着重强调这些系统设计背后的若干指导原则。

25.1 崩溃、异常中止和介质故障

虽然计算机系统的可靠性在这些年有了很大的提高，但是出现故障的可能性还是存在的。故障可能是由处理器中的问题引起的，也可能是由主存中的问题引起的（比如断电），还可能是由软件中的错误引发的。这些故障会导致处理器的运行不可预测；比如，在执行某种导致处理器停止工作的操作之前，处理器有可能随意地把错误的数据写在存储器的某个地方。我们把这种故障称作**崩溃**（crash）。我们假设一旦出现崩溃，主存中的数据就丢失了。正因为如此，主存又称为**易失型存储器**（volatile storage）。发生故障的处理器也有可能向海量存储设备写入错误的数据。不过这种事件不太可能出现，于是不妨假定海量存储器的内容并不受崩溃之影响。

一般来说，当事务处理系统崩溃时，有一些事务仍处于活动状态。这就意味着数据库将处于不一致的状态。当系统崩溃后重新启动时，在尚未执行**恢复程序**（recovery procedure）把数据库复原到一致状态之前，系统的服务是无法继续的。设计恢复程序的一个主要问题是，如何处置发生崩溃时还处于活动状态的事务T？原子性要求恢复程序要么让T继续执行，直到它全部完成——称之为**前滚**（rollforward）；要么撤销崩溃前T所产生的任何影响——称之为**回退**（rollback）。

前滚虽然并非不可能，但实现起来是非常困难的。如果T是一个交互性的事务，那么继续执行就还需要终端用户的配合。用户必须知道，崩溃前用户所请求的更新操作中，有哪些已实际记录到数据库里；并从该点开始继续提交请求。在崩溃时刻，事务的本地状态（即T的本地变量的状态）有可能是存放在易失型存储器里的，故而很可能已经丢失。这样，前滚一个事务就变得更加困难。除非T的本地变量的状态得到复原，否则从崩溃发生的那一点继续执行事务T是不可能的。这样，为了在崩溃后能够前滚事务T，必须采取特别的措施，在事务执行期间，定期地把事务的本地状态保存到海量存储设备里。

基于上述原因，在恢复时，往往对崩溃时还处于活动状态的事务实施回退，以利于实现事务的原子性。注意，这里我们关心的是崩溃前事务T给数据库所带来的变动。事务还可能会

产生一些其他的（外部）影响。例如，在屏幕上显示一条消息，或在工厂里启动一台控制器等。尽管我们在21.3.2节讨论了处理它们的一种办法，但是外部行为一般是很难逆转的。

回退机制不仅可以用来处理崩溃，还可以用来处理事务异常中止。一个事务可能由于众多原因而异常中止：

- 事务可能因用户而异常中止。例如，用户输入了错误的数据。
- 事务可能因本身而异常中止。例如，它在数据库里遇到了某个意外的信息。
- 事务可能因系统而异常中止。例如，一个事务可能会与其他事务发生死锁，或者一个事务可能由于缺乏足够的系统资源而无法完成，或者由于违反了约束条件而异常中止。

持久性要求事务一旦提交，它对数据库的影响就不会消失。数据库通常储存在磁盘这样的海量存储设备里。由于通常系统崩溃不会波及这些设备，所以海量存储设备通常称为**非易失型**（nonvolatile）的。然而，这些设备将受到其固有的故障形式，即**介质故障**（media failure）的影响。介质故障可能会影响设备上储存的全部或部分数据。应对这种故障并保持持久性的一种方法就是数据的冗余存储。冗余拷贝越多，可以容许的介质故障也就越多。因此，持久性并非是绝对的，它依赖于数据的价值以及公司在数据保护方面的投资规模。因为在事务执行过程中有可能会发生介质故障，所以介质故障的恢复程序也必须提供回退的能力。

25.2　直接型更新系统和先写型日志

直接型更新系统和延迟型更新系统的回退机制是不一样的。因为直接型更新系统（immediate-update system）的应用比较普遍，所以首先介绍这种系统。对这种系统的介绍将分为几个阶段。在本节中，我们描述一个简单但却不太现实的系统，以说明其主要的思想。在以后各节我们再讨论在商用系统中必须处理的一些难题，以及为解决这些难题必须对简单系统所做的改进。

直接型更新系统维护着一个由一系列记录构成的**日志**（log）。每当事务执行时，其记录就被添加到日志里，这些记录从不修改或删除。系统就是引用日志而实现原子性和持久性的。对于持久性而言，该日志是在存放数据库的海量存储设备发生故障后用来复原数据库的。因此，日志必须保存在非易失型存储设备里。一般情况下，日志是储存在磁盘上的顺序文件。此外，日志通常都被复制（并将拷贝存放在不同的设备上），以避免单个介质故障所产生的影响。

存储器的组织结构如图25-1所示。从效率的角度考虑，数据库和数据库服务器之间数据传送的单位是页，最近访问的页面存放在易失型存储器的**高速缓存**（cache）里。此外，最终将存放到日志里的信息通常首先储存在易失型存储器的**日志缓冲区**（log buffer）里。高速缓存和日志缓冲区的存在使得回退和提交的处理愈加复杂。在本节中，我们始终假设高速缓存和日志缓冲区均未使用，并且假设信息是直接从数据库读取或写入数据库，并直接写入日志里的。

当事务执行某个会改变数据库状态的操作时，系统便在日志中添加一条**更新记录**（update record）（倘若是纯粹读取数据库的操作，那就不需要添加记录）。每个更新记录都描述了已经做出的变更，尤其是它包含有足够的信息，万一事务异常中止的话，系统可以撤销所做的变更。由于每当做出变更便添加记录，于是日志里就汇聚了所有事务的更新记录。

更新记录以最简单的形式记载着被修改的数据库项的**前像**（before-image），即修改实施

前该数据项的一份物理拷贝。一旦事务异常中止，这个更新记录就可用于把该项数据复原成其原先的值。因此，前像有时又称为**撤销记录**（undo record）。倘若并发性控制令操作可串行化地执行（即修改该项数据时，该数据项锁住，并严格地进行并发性监控），那么其他的并发事务是访问不到该新数据值的。因此，异常中止的事务对其他事务没有任何影响：一旦恢复，对数据库也没有任何影响了。除了前像之外，更新记录还确认采用**事务Id**（transaction Id）作出变更的事务以及改变了的数据项。后面我们还会介绍更新记录中包含的其他信息。

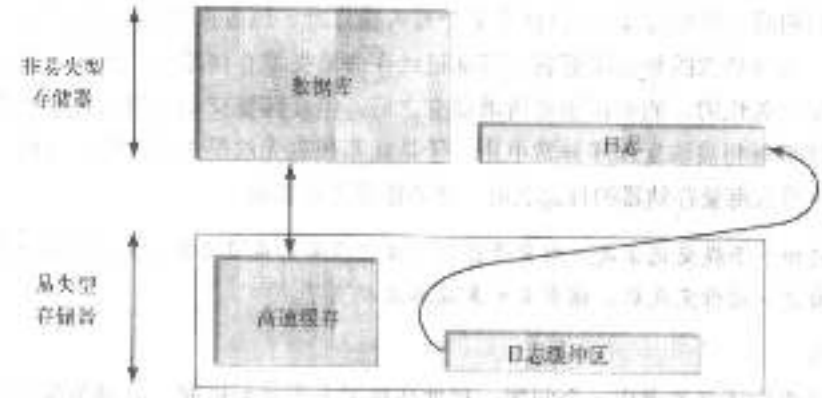


图25-1 存储器的组织结构

由于更新记录中包含着数据项的一份物理拷贝，所以登录的这种形式称为**物理型登录**（physical logging）。

如果系统让事务T异常中止或事务T自行异常中止，那么使用日志完成回退是相当简单的。从后向前地扫描日志，一旦遇到T的更新记录，就把前像写入数据库，以此来撤销更改。因为日志可能特别长，所以反向地扫描整个日志以确保所有T的更新记录都被处理的方法是不现实的。为了避免这种从尾到头的扫描，每当事务开始执行时，就向日志追加一条包含该事务Id的**开始记录**（begin record）。一旦遇到T的开始记录，从尾开始的扫描就可以结束（为了便于访问，可以把某个事务的日志记录链接在一起，并且把最近访问的记录放在该链表的头部）。

把这种技术扩展一下就可以实现**存储点**（savepoint）。每当事务声明某个存储点时，就有一条**存储点记录**（savepoint record）写入日志。这条记录不仅包含此事务的Id和该存储点的Id，还可能包含声明该存储点时打开的任何游标的信息。为了回退到某个特定的存储点，将反向地扫描日志，直到到达该存储点的记录为止。扫描过程遇到的每条事务更新记录的前像直接应用到此数据库上。记录里的游标信息则用来重新确定声明该存储点时的游标位置。

崩溃所引起的回退要比单个事务的异常中止复杂一些，因为在恢复的时候，系统必须首先分辨出哪些事务需要异常中止，尤其是，系统对崩溃发生时已完成的事务（包括已提交、已异常中止的）和还在活动的事务必须加以区分。而所有还处于活动状态的事务则必须使之异常中止。

当某个事务提交的时候，它就会把一条**提交记录**（commit record）写到日志里。如果某个事务异常中止，那么该事务使回退它的更新操作，并把一条**异常中止记录**（abort record）写到日志里。这两种记录都包含着该事务的Id。当写完提交记录或异常中止记录之后，该事

务就可以释放其持有的任何锁。

利用这些记录，恢复程序可以确定崩溃时还有哪些事务处于活动状态。在反向扫描过程中，如果与T有关的第一条记录是更新记录，那么可以断定，崩溃发生时T还处于活动的状态，因此T必须被异常中止。如果第一个记录是提交记录或异常中止记录，那么就可以断定，崩溃发生时T已经处于完成状态，那么以后再遇到它的更新记录时，便可以忽略。

值得注意的是，出于持久性的考虑，每当T提交时就向日志写入一条提交记录是非常重要的。因为我们的简化模型假设，当T请求某个写入操作时，该数据库是直接更新的；所以当T请求提交时，它所请求的数据库更新全部都记录在非易失型存储器里。然而，提交请求本身却是无法保证持久性的。倘若在事务请求提交之后，但在其提交记录写入日志之前就出现了崩溃，那么该事务将被恢复程序异常中止，于是此系统也无法提供持久性。因此，一个事务在其提交记录写入海量存储器的日志之前，并不能算真正提交。

向日志追加一条提交记录是一种原子操作（该记录要么在日志里，要么不在日志里），当且仅当这一动作完成后，该事务才算是真正提交了。

1. 检测点

关于崩溃我们还必须考虑一个问题，那就是必须采用某种机制，以避免恢复过程中反向地扫描整个日志。倘若没有这种机制，恢复过程就无法知道该何时停止扫描。因为一个在崩溃时刻还处于活动状态的事务，也许它在很早的时候曾向日志写入过一条更新记录，可是其后一直到崩溃为止都没有做过任何数据库更新操作。除非持续反向扫描直到到达该更新记录，否则恢复过程不能发现该事务存在的任何迹象。为了解决这种情况，系统需要周期性地向日志写入一条**检测点记录**（checkpoint record），以列出所有当前活动事务的标识。恢复过程（至少）必须反向地扫描到最近一次的检测点记录。如果T的标识列在某个检测点记录中，但恢复过程从该检测点记录开始一直到日志结束，都没有遇到该事务的结束记录，那么可以认为当崩溃发生时，该事务还是活动的。于是，反向扫描必须继续进行，直到遇到该事务的开始记录为止。只有在所有这样的事务都处理完成后，反向扫描才能够结束。扫描过程用到的仅仅是最近一次的检测点记录（每个检测点记录将取代它以前的那个检测点记录）。检测点记录写入日志的频度将会影响到恢复的速度，因为检测频率高意味着需要反向扫描的日志少。

图25-2给出了日志的一个例子。在恢复过程的反向扫描中，它发现T_6和T_1在崩溃时还处于活动状态，因为它们最后追加的记录都是更新记录。恢复过程按照反向扫描时遇到的先后顺序利用这些记录里的前像，来回退这些记录所指出的数据库项。由于首先遇到的T_4记录是提交记录，所以恢复过程知道当崩溃发生时T_4处于非活动状态，因此可忽略T_4的更新记录。当遇到检测点记录时，恢复过程便得知，在设定该检测点时，T_1、T_3和T_4是活动的（T_6没有在检测点记录中提及，因为它是在该检测点设定之后才开始的）。于是，恢复过程可得出以下结论：当崩溃发生时，除了T_1和T_6以外，T_3也处于活动状态（因为没遇到T_3的结束记录）。再也没有其他事务处于活动状态了，因此，必须异常中止的事务就是这几个。恢复过程必须继续其反向扫描，按照遇到T_1、T_3和T_6的更新记录的顺序来处理这些更新记录。只有遇到这些事务的开始记录后，反向扫描才能宣告结束。

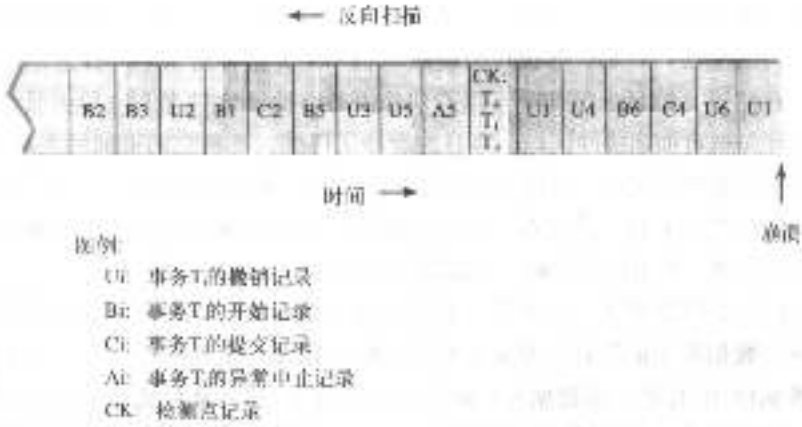


图25-2 日志的一个例子

2. 先写型登录

我们已经假设，当更新数据库项x的时候，数据项x的更新记录也要写入日志里。实际上，更新x和追加其更新记录必须遵循一定的顺序。如果操作顺序不同，又有什么差别呢？我们不妨假设在执行这些操作时发生了崩溃的情况。如果崩溃发生在这两种操作都尚未完成之前，那就没有问题。日志里没有任何更新记录，x没有被更新，因此也不需要恢复过程对x执行任何撤销操作。如果崩溃发生在这两种操作之后，恢复过程也会像上面所说那样正确地处理。可是，倘若首先更新的是x，随后却发生了崩溃，而其更新记录还没来得及追加到日志里；那么由于日志里没有其前像，恢复过程无法引用，所以就不能对该事务进行回退。因此，恢复无法把数据库回归到某个一致状态——这是一种不可接受的状况。

另一方面，如果首先把更新记录追加到日志里，那么问题就解决了。当重新启动时，恢复过程只需要引用该更新记录对x进行复原就可以了。如图25-3所示，无论崩溃是发生在该事务将x的新值写入数据库之前还是之后，两者的结果都是一样的。x的初值是3，事务把它改成5，在事务提交之前系统崩溃了。如果崩溃在更新记录写入日志之后，但又在x值更新之前（图中崩溃1处）发生，那么当系统重新启动时，数据库中x的值和其更新记录中的x前像里的值是一样的。恢复过程按照前像来重写x，并没有改变x的值，但完成恢复后的x最终状态是正确的。如果崩溃发生在更新x的值之后（图中崩溃2处），恢复过程也会把x的值复原为3。

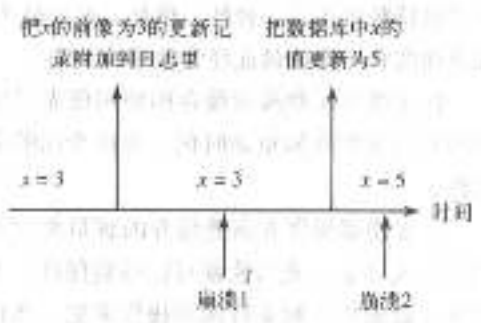


图25-3 利用先写型日志，恢复程序可以正确地处理数据库的恢复

因此，始终必须在更新数据库之前首先把更新记录追加到日志里。这一特性称为**先写**（write-ahead）特性。这种日志也就称为**先写型日志**（write-ahead log）。

25.2.1 性能和先写型登录

尽管上一节介绍的先写型登录方法工作正确，然而从性能的观点来看，这种方法是不能

接受的，因为它使更新数据库的I/O操作的数量增加了一倍。每次数据库更新都需要向日志追加一条记录。为避免这一开销，数据库系统经常利用易失型存储器中的日志缓冲区，作为日志记录的临时存储器。该日志缓冲区可以看成是海量存储器中日志的一种扩展。如图25-1所示，日志记录首先被追加到缓冲区，一旦日志缓冲区填满，再将它们追加到或**刷入**（flushed）日志里。使用日志缓冲区以后，写日志的代价就由缓冲区中的所有日志记录来分摊。

从崩溃恢复的观点来看，储存在易失型存储器上的日志缓冲区与储存在海量存储器上的日志有着很大的区别。当系统崩溃时，日志缓冲区中的内容将会丢失。

此外，我们的介绍忽略了一个事实，那就是为了提高性能，绝大多数的数据库系统都支持将最近访问的数据库页面储存在易失型存储器的高速缓存里。这样，当一个事务访问某个数据库中的数据项x的时候，该数据库系统首先将把包含x的海量存储器中的数据库页面带到高速缓存中，然后把x的值拷贝到该事务的缓冲区中。将该页面存放在高速缓存里是基于这样的假设：此事务很可能不久就会更新x的值，或者读取该页里的其他数据项。若是这样，就可避免一次页面传送，因为可以直接访问高速缓存里的页面(无须I/O操作)。例如，一个事务可以通过游标来扫描某张表。当从某页里检索了一行数据之后，事务将要获取的下一行很可能就在同一页里。

如果更新了x，高速缓存里该页的拷贝（并非数据库中该页的原始拷贝）将被修改。在高速缓存里该页将标记为脏（dirty），意思是它所包含的数据库记录版本要比海量存储器中的版本新。当高速缓冲空间不足时，对于标记为干净（clean，非脏）的高速缓存页面，即仅仅读取其内容的那些页面，可以用数据库的新页面直接覆盖。然而，脏页面在其能够被覆盖之前，首先必须回写到数据库里。至于哪些页面应该保留在高速缓存里的决策，是由某种页面置换算法来确定的。该算法的目标是使由高速缓存页面满足的数据库访问的数目尽可能地多，同时又保持数据库的一致性。例如，最近最少使用（Least-Recently-Used，LRU）算法可以用于在高速缓存里保持最近经常使用的页面。

日志缓冲区和高速缓存的使用使先写型登录愈加复杂，因为它们影响到海量存储器中数据库和日志的实际更新时间。前面介绍的简单方案的两条性质必须保持：先写性和提交的持久性。

只要能够确保在高速缓存的脏页面回写到数据库之前，首先把日志缓冲区里对应的更新记录写入日志，先写性就可以得到保持。为达到这一目的，一般采用两种机制。先看第一种机制。数据库一般支持两种操作来把一条记录追加到日志里。一种操作只是单纯地把该记录添加到日志缓冲区里，而另一种操作则是先把该记录追加到缓冲区里，接着立即把缓冲区里的内容写入日志。后一种操作称为**强制性**（forced）操作。一个正常（非强制性）的写入操作只不过是登记一个请求，请求操作系统向海量存储器写入某个页面（该I/O操作要在以后的某个时间里完成），而强制性写入操作在该写入未完成以前是不会把控制权返还给其调用者的。由于日志是串行排列的，因此每当一个例程请求对某条更新记录进行强制性写入操作时，就需要把其前的所有记录也都全部写入日志。一旦该例程恢复执行，就可以确保这些记录已被储存到海量存储器里了。这样就可以安全地提出请求，把相应的高速缓存脏页面写入数据库。

第二种机制采用**日志序列号**（Log Sequence Number，LSN）对所有的日志记录顺序地编号，该序列号是存储在其日志记录里的。此外，对于每个数据库页面，其中还存储着对该页

最近一次更新的更新记录的LSN。这样，如果某个数据库页面包含数据项x、y、z，而且最近一次的更新项是y，那么该页存放的LSN的值将是y的最近更新记录的LSN。

利用强制性写入和LSN机制，我们可以保证先写性。当需要高速缓存的空间，并且决定把脏页面P的内容回写到海量存储器里时，系统需要确定日志缓冲区是否继续保存其LSN值等于储存在P中的LSN值的那些更新记录？若是，那么P中的这个LSN值必然要比海量存储器里日志的最后一条记录的LSN值大。因此，日志缓冲区的内容必须在该页写入数据库之前强制性地写入海量存储器。若不是，那么与该页的某个数据项的最近一次更新相对应的更新记录已经追加到海量存储器的日志里了，于是，立即可以覆盖高速缓存中的该页面。

这样，我们看到，为了实现先写性，有时候我们必须推迟包含某个更新项的高速缓存页面的写入操作，直到包含该相应项更新记录的日志缓冲区已经写入日志为止。为了实现持久性，我们必须确保在T的提交记录追加到海量存储器的日志里以前，T所更新的所有数据项的新值都已经储存在海量存储器里。否则，如果崩溃出现时该提交记录已经追加了，可是其新数据值尚未来得及存放到海量存储器里的话，那么该事务虽然已经提交，但是其新数据值将丢失。因此，尽管系统的崩溃得到了处理，可是持久性却没能实现。

有两种策略可以确保持久性：**强制**（force）策略和**非强制**（no-force）策略。采用强制策略是把页面写入高速缓存扩展为强制操作。在T的提交记录追加到海量存储器日志之前，高速缓存中被T更新过的数据库页面都被强制性地移到数据库中。其事件序列如下：

1) 如果该事务的最后一条更新记录仍然存放在日志缓冲区中，那么强制性地将其写入海量存储器日志中。这能确保所有的前像都是持久的。

2) 如果由该事务的更新所产生的任何脏页面仍然存放在高速缓存中，那么强制性地将它们写入其数据库里。这能确保所有的新数据值都是持久的。

3) 向日志缓冲区追加该提交记录。一旦该记录写入海量存储器日志中（见下文），就能确保该事务是持久的。

图25-4说明了更新某个数据项x的事务T的事件序列。包含LSN值j和前像x_{old}的更新记录位于日志缓冲区里，而包含新值x_{new}和更新记录LSN的更新页位于高速缓存里。该页是脏的，它还没有写入海量存储器。该页的原始版本（包含值x_{old}和LSN值$s(s<j)$）还在海量存储器里。为了满足先写性，必须在覆盖该脏页面之前(图中的步骤2)，将此更新记录先行移入海量存储器里(图中的步骤1)。为了确保这一点，可能必须对日志缓冲区实施强制写入。类似地，在包含LSN值$k(k>j)$的提交记录能够追加到日志缓冲区之前，该脏页面的内容必须先行写入海量存储器，以便确保它在提交记录(图中的步骤3)之前就已经在海量存储器里了。为确保这一点，强制写入脏页面是有必要的。

值得注意的是，对提交记录采取强制写入并非是必需的。然而，在提交记录尚未写入海量存储器日志以前，该事务仍然还是未提交的。在有些系统里，每当提交记录追加到缓冲区后，日志缓冲区的内容就被强制写入海量存储器，于是提交立即生效。然而，另外一些系统此时并不强制写入日志缓冲区，故而避免了一次写入日志的操作。不过，该提交操作只有等到以后在将其日志缓冲区移往海量存储器时才能奏效。这种协议称为**成组提交**（group commit），因为这一组事务的提交记录全都在日志缓冲区里，一旦出现了下一次的写入，它们就一次全部提交。

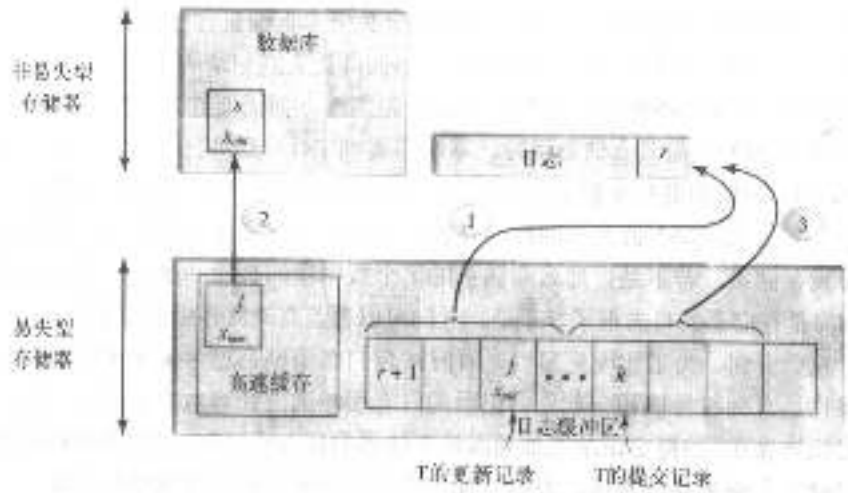


图25-4 采用强制策略实现持久性。在该事务的提交记录写入日志之前，可能必须强制把事务修改过的页面从高速缓存写入数据库。在事务T更新过的包含LSN值n的脏页面能够写入数据库之前，可能必须强制日志缓冲区将LSN值等于n的更新记录移往海量存储器日志(1)。在此脏页面被覆盖(2)之后，T的提交记录就可以追加到日志缓冲区里了(3)。该日志缓冲区的内容则可以在以后的某个时间再写入日志

采用强制策略确保持久性的主要缺点是：覆盖高速缓存脏页面与提交这两个操作是同步的。被某个事务修改过的页面必须在该事务提交之前写到数据库里。由于页面的写入操作比较缓慢，因此事务的提交就被推迟了，从而影响了响应时间。

强制策略的另一个缺点是要与被不同事务频繁修改的页面打交道（例如，那些含有系统相关信息的页面）。LRU页面置换算法可能不会选择把这样的页面换出高速缓存；然而，由于强制策略，每当包含着对此页面进行修改的某个事务提交时，该页面都要被重写。另一方面，强制策略的优点在于：崩溃发生后，恢复已经提交的事务时根本无需采取任何动作。届时，该事务的提交记录已经写入海量存储器日志里，同时所有由该事务更新而产生的新值也已经拷贝到海量存储器数据库里。下一节中讨论的非强制策略未必能保证这样的结果。

25.2.2 检测点和恢复

在上一节，我们指出，事务T所更新的某个数据项x的新值，在该事务请求提交时，仍然可以留在高速缓冲区的某个脏页面里（这意味着该数据库页面的拷贝还没有被更新）。为了保持T的持久性，系统必须在T的提交记录写入海量存储器日志之前，先把x的新值记录到海量存储器里。确保该值在非易失型存储器里的常用办法是，除了前像外，把后像也储存在该日志的更新记录里。

简单地说，更新的数据项的**后像**（after-image）（有时称为**重做记录**（redo record））就是该数据项新值的一份物理型拷贝。因为在日志中，T的所有更新记录都先于其提交记录，所以每当T的提交记录写入海量存储器日志时，海量存储器里已经储存着T所创建的所有数据库项的新值了。这样，即使海量存储器中包含x的数据库页面在事务T提交时尚未更新，而在T提交后系统崩溃，那么正如图25-5所示，恢复时可以通过后像在数据库页中安装x的新值。先写性

依然要求在脏页面刷入数据库之前，先将更新记录写入海量存储器日志。但是，对于提交记录写入的时间，不再有任何的顺序规定，特别地，提交记录也可以在T修改过的所有高速缓存页面写入数据库之后，先行写入持久性存储器日志。

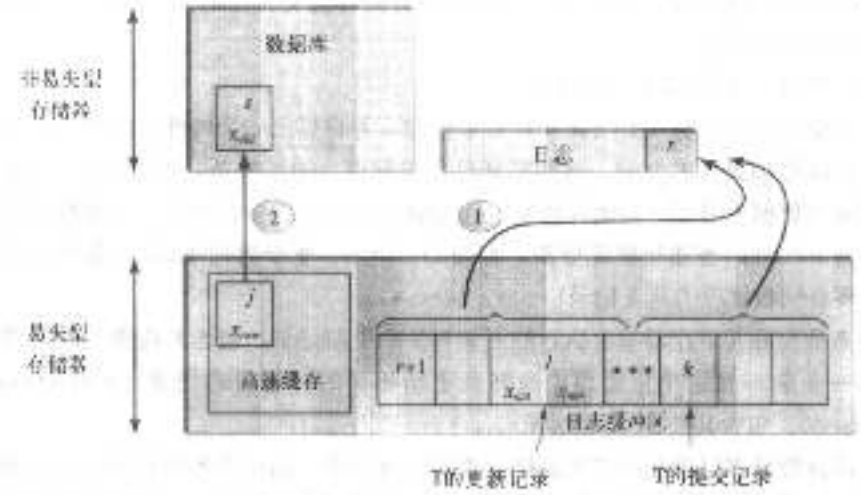


图25.5 利用带有非强制策略的后像实现持久性。先写性要求在事务T所更新的脏页面可写入数据库之前(2)，对应的更新记录必须已经写入海量存储器日志(1)。虽然该提交记录不能在更新记录之前写入日志，但是提交记录的写入时间和脏页面的写入时间之间的关系是不受制约的

非强制策略的显著优点是：事务T马上可以提交，无须等到高速缓存中更新过的所有页面都得到强制为止。其缺点在于，一旦发生崩溃，恢复过程将由于以下的情况而变得复杂：

- 数据库中的某些页面可能包含由尚未提交的事务所写入的更新。这些页面必须通过日志中的前像来回退。无论使用强制策略还是非强制策略，这个问题都是存在的。
- 数据库中的某些页面可能尚未包含所有的由已提交事务所写入的更新。这些页面必须通过日志中的后像来前滚。仅当使用非强制策略时，才会发生这个问题。

我们已经处理了回退问题。现在的问题是，如何识别出数据库中必须前滚的页面？

一种办法是使用**精确检测点**（sharp checkpoint）。在把检测点记录CK写入日志缓冲区之前，处理过程将中断，并把高速缓存中的所有脏页面写入数据库。因此，当恢复过程扫描到CK时，可以断定日志里CK前的所有更新记录在崩溃之前就已写入了数据库。如果CK是最近的检测点记录，那么在日志里，只有CK以后的那些更新记录才有可能尚未写入数据库。基于上述推断，恢复处理可以分为三遍：

- 第1遍　反向扫描日志直到遇到最近的检测点，由此确定哪些事务在崩溃发生时刻还处于活动状态（因而必须被回退）。
- 第2遍　从该检测点开始正向扫描（重演），利用所有遇到的更新记录（无论其事务已提交还是尚未提交）的后像来更新数据库中对应的数据项。本遍结束时，凡是由已提交或未提交事务所引起的变化都已到位，因此，数据库的状态就是崩溃发生前那一刻的状态。
- 第3遍　反向扫描日志，以回退崩溃时还处于活动状态的所有事务，利用这些事务的更

新记录的前像来逆转数据库里相应的更新。当到达所有事务的开始记录时，本遍就宣告结束。其效果就等同于异常中止所有未提交的事务。

DO-UNDO-REDO（做-撤销-重做）就是这种恢复处理的一般形式的名字。DO是指事务在更新数据项时所做的原始动作，UNDO是指倘若事务没有提交，在第3遍中出现的回退，REDO是指第2遍中出现的前滚。

采用这一技术，需要考虑三个问题：

- 在该检测点之后更新数据项且又异常中止的事务将提出一个特殊的问题。在它们的异常中止记录追加到日志之前，这些更新已经被回退。遗憾的是，在第2遍里，这些更新又被恢复到数据库里了。为确保恢复过程能够正确地处理异常中止，回退操作应该被看作该事务所实施的普通数据库更新。如图25-6所示，更新数据项x的异常中止事务在其日志中将有该数据项的两条记录：
  - 一条是与异常中止前事务执行的更新有关的更新记录，包含其前像x_{old}和后像x_{new}。
  - 另一条是与异常中止处理中逆转此更新相关的**补偿日志记录**（compensation log record），包含其前像x_{new}和后像x_{old}。

  该补偿日志记录在日志里紧接在更新记录后面，而该事务的异常中止记录则紧接在最后一条补偿日志记录之后。第2遍扫描首先处理更新记录，并把它的后像写入数据库；然后处理补偿日志记录，并把它的前像写入数据库。数据库中x的最终值为x_{old}。由于崩溃时该事务未处于活动状态，于是第3遍扫描可以忽略它的更新记录和补偿日志记录。因为补偿日志记录也可以看成是更新记录，所以它们有一个良好的性质：将日志中事务的提交和异常中止同等看待。
- 在最后的检测点记录写入后，更新的高速缓存中的某些页面可能已经写入数据库里。因此，第2遍中所遇到的一些更新记录描述了已经被传递给数据库的那些更新。使用这些记录中的后像来更新数据库是不必要的，但并非不正确的。对于这种情况，数据库中数据项的值和该更新记录中的后像是等同的，所以使用后像毫无意义[⊖]。
- 在恢复过程中，系统有可能再次崩溃，在这种情况下，恢复过程可以重新启动。根据第二次崩溃发生时恢复过程进行到哪一遍，可以决定第二次应用到该数据库项上的究竟是前像还是后像。然而，这些映像的使用是**幂等的**（idempotent）。也就是说，采用一个特定的后像来多次更新某数据库项，与更新一次有着相同的效果[⊜]。因此，恢复过程中的一次崩溃（实际上可能多次崩溃）对结果并不产生影响。当恢复最终完成时，未提交的事务已经异常中止，而已提交事务所作的更新则已经写入了数据库。

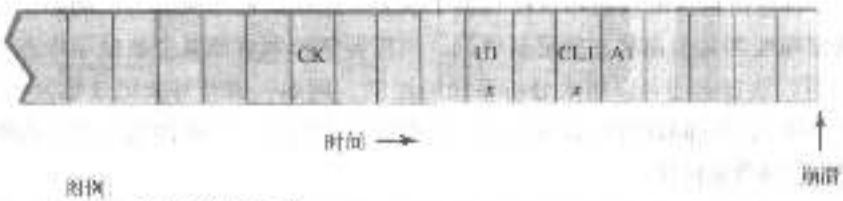
1. 模糊检测点

使用精确检测点有一个很大的缺点。在把检测点记录写入日志缓冲区之前，系统必须中断，并把脏页面从高速缓存写入海量存储器。而这种服务的中断是许多应用程序所不能接受

⊖ 需要注意的是，在该检测点记录写入日志之后，数据库中同一个数据项被多次更新的情况。对于这种情况，其后像与该数据项的值是不相同的。然而，在第2遍里处理了最近更新记录之后，该数据项的值便又恢复成崩溃前的该项值。

⊜ 在前面的段落中我们使用过幂等性这一概念。当采用后像来前滚崩溃发生前最终更新的页面时，后像的幂等性确保了恢复过程中的更新不会产生影响。

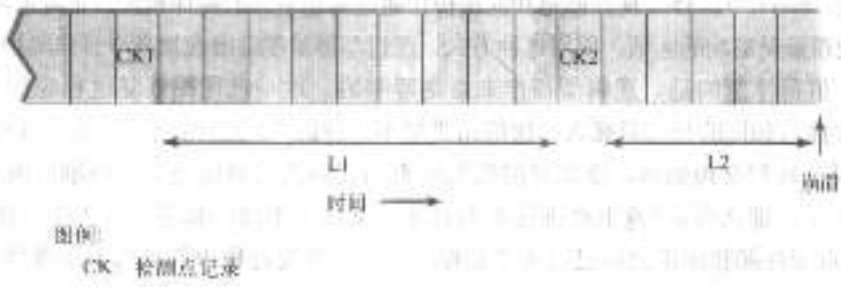
的。利用**模糊检测点**（fuzzy checkpoint），只需对恢复过程作少许改动，就能解决这一问题。当一个检测点记录写入日志时，并没有把脏页面从高速缓存移出；取而代之的等价操作是，只在易失型存储器中对这些页面做个标记，随后在正常的处理中由后台处理将它们写入数据库。这种方法唯一的限制是：在前一个检测点记录里所标记的脏页面全部写入数据库之前，不能设置下一个检测点。



图例：
U1：事务T1的更新记录
A1：事务T1的异常中止记录
CL1：事务T1的补偿日志记录
CK：检测点记录

图25-6 显示更新数据库变量x的某个异常中止事务的记录的日志，该日志同时包含了x的一条更新记录和一条补偿日志记录

图25-7解释了模糊检测点。当CK2追加到日志缓冲区时，原先追加CK1时高速缓存中的所有脏页面都已经写入数据库。这些页面的修改对应着更新该日志早出现在CK1之前的记录。与该日志L1区域的某条更新记录所对应的数据库更新，产生出脏页面P。P很可能在追加CK2到日志缓冲区时仍然还在高速缓存里（届时P也许已经写入海量存储器，但我们不能肯定）。如果当追加CK2到日志缓冲区时，P仍然在高速缓存中，那么它的标识将被打上标记。直到下一个检测点记录追加之前，我们并不能保证该页已经写入数据库。由于图中崩溃发生在该页写入数据库之前，因此我们必须做最坏的打算：L1和L2区域的更新记录在数据库里没有反映出来。



图例：
CK 检测点记录

图25-7 模糊检测点的使用

为了使数据库恢复到崩溃前的所有更新已处理完毕的状态，恢复过程的第2遍必须作如下修改，使前向扫描从CK1开始而不是从CK2开始。第1遍依旧在CK2处结束，因为其目的只是确定出直到崩溃时还处于活动状态的事务。于是，恢复过程比使用精确检测点时要慢多了。现在必须权衡恢复速度和常规操作的可用性，以决定对于特定场合到底是使用精确检测点更合适还是使用模糊检测点更合适。

2. 日志的存档

我们必须考虑另外一个问题，以使我们对日志登录和恢复的讨论(较为)完整。我们说过，日志记录追加到日志后就从不删除。一旦海量存储器装满了日志记录以后，又会发生什么事情呢？你可能猜想，可以抛弃该日志的开始部分；然而，出于众多的原因，日志记录经常需要保存相当长的一段时间。

一方面，日志所包含的信息不仅仅有助于恢复，还在其他方面有用。例如，日志保存了使得每个数据项变为其当前状态的更新序列，当需要该公司解释某个数据项状态时，这些信息就非常有用。该日志还可以用来分析系统的性能。例如，倘若每条记录包含一个时间戳，那么每个事务的响应时间就可以计算出来。不抛弃日志的另一个原因则与介质故障有关，这一点我们将在25.4节里讨论。

既然不能抛弃日志，那么它的开始部分就应该移送到第三类存储器（例如磁带）里。这一过程称为**存档**（archiving）。只有最近的日志记录需要保留在线，于是现在的问题是确定从该日志哪一点开始移去一部分。当然，活动事务的记录必须保留在线，以便迅速处理异常中止和恢复过程。于是，包含其LSN值比最早活动事务的开始记录的LSN值小的记录的日志部分可以存档。然而，介质故障恢复还需要引入其他的约束，这一点我们将在25.4节里讨论。

25.2.3 逻辑型登录和物理逻辑型登录*

物理型登录有着重大的缺陷，在关系型数据库场合尤为突出。一次简单的更新可能导致数据库中大量页面的修改。对于这种情况，前像和后像可能会很大，并且难以管理。例如，在某表中插入一个元组可能需要重新组织其所追加的页面，并在与该关系有关的索引中插入新的索引项。受影响的整个区域都必须记录在其前像和后像中，这样使更新记录变得很大，并且增加了管理日志所必需的I/O开销。*逻辑型登录*就是弥补这一缺陷的一种技术。

采用**逻辑型登录**（logical logging）时，更新记录里储存的不是需要更新的数据项的快照，而是更新操作本身及其逆操作。这样一来，把行r插入表T的操作的撤销记录是<*delete*，r，T>，重做记录是<*insert*，r，T>。现在回退和前滚操作通过实施适当的操作构成，不再像物理型登录那样简单地覆盖受影响的区域。使用这种方式，逻辑型登录能够潜在地减少日志维护的开销。

然而，值得注意的是，逻辑型操作未必是幂等的，这一点使得恢复过程变得复杂起来。例如，两次插入相同的行与只插入一次的结果是不一样的[⊖]。这样，在第2遍处理更新和补偿日志记录时，我们必须知道，更新过的数据库页面在崩溃前到底是否已经刷入海量存储器。如果已刷入了，那么在第2遍里再进行重做将导致某种不正确的状态。在上面的例子中，如果包含x的页面在濒临崩溃之际已写入了数据库，那么恢复过程中的逻辑型重做将导致x插入两次。

幸运的是，使用页面中的LSN可以很容易地解决这个问题。如果在第2遍中发现，某一页中的LSN大于或等于该页中某一数据项的更新记录的LSN（表示该页包含了实施该更新操作的结果），那么就不要执行重做操作。

第二个问题比较严重。例如，无论元组t是否已经插入某个表，我们都隐式地假设逻辑型

⊖ 不妨回想一下，关系数据库未必一定检查重复行。

操作是自动地进行的。然而，为了执行一个插入操作，可能需要修改许多页面，因此相对下述故障来说，该逻辑型操作就不是原子化的。该故障为系统在部分（而非所有）页面写入以后就崩溃了。因此，数据可能在恢复过程中处于不一致的状态。这里的不一致性的形式有所不同。例如，包含t的数据页面可能已经写入海量存储器，但是缺少包含有指向t的指针的索引页。这和不一致的事务所引起的不一致状态是不一样的，因为这里的不一致并非是数据值的不一致，而是数据储存方式的不一致。

应用逻辑型操作来解决不一致状态问题很可能会失败。在本例中，第2遍逻辑型重做记录的应用可能导致该表中t有两份拷贝。这在采用物理型登录时是没有问题的。在采用逻辑型登录时，倘若逻辑型操作仅仅影响到一个页面，这也不成问题。因为这时操作的执行是原子化的，而且其LSN指示出该操作究竟有没有发生。

为了解决这一问题，可以使用**物理逻辑型登录**（physiological logging）。这项技术是物理型登录和逻辑型登录的折中（其名字是复杂描述*physical-to-a-page, logical-within-a-page*，即“物理到页、逻辑页内”的一种缩写）。一个涉及多页更新的逻辑型操作被分解为多个（逻辑型）**微型操作**（mini-operation）。通过这种方式，每一个微型操作被限制在单一页面里（这是物理逻辑型登录操作的物理维度），并且保持了页面的一致性。这样一来，无论何时发生故障，都可以在页面上执行微型操作。逻辑操作“把t插入表T”可以被分解为微型操作“把t插入包含T的文件的某个特定页面”，该操作又有一个或多个后继的微型操作，“在T的某个特定索引页中插入一个指向t的指针”。因为每一个微型操作都拥有各自的日志记录，所以即使在微型操作执行过程中发生了崩溃，恢复过程也可以正确地处理。由于逻辑型微型操作未必是幂等的，因此可以使用LSN（如上所述）确定第2遍中有哪些微型操作已经被应用到某一页面上。

在上述例子中，每个微型操作都是限定在单一页面内的逻辑型操作。把一个数据项实际插入某页面的工作，除了把该数据项储存到该页面以外，实际上还涉及更新该页的页头信息（用于找到数据项以及该页的空闲空间）。因此，就可以全部重新组织该页面。使用物理型登录时，所有这些更改的物理映像都必须保存在该日志记录中。而使用物理逻辑型登录时，只有微型操作的性质（例如插入）、它的变量（例如t）以及该页的标识必须保存。如果微型操作不能方便地逻辑表达，那么可以使用物理型日志记录。

25.3　延迟更新系统的恢复

在**延迟更新系统**（deferred-update system）里，一个事务的写入操作并没有更新数据库中的相应的数据项。相反，这些信息保存在一个称为该事务的**意向列表**（intention list）的特殊的内存区域里。该意向列表并不长期存储。如果该事务提交的话，它的意向列表就用来更新数据库。

当我们要异常中止这样的事务的时候，只要抛弃它的意向列表就可以了。类似地，如果系统崩溃的话，我们无需采取特别的措施来异常中止活动的事务，因为它们并没有修改数据库。

对于已经提交的事务，为了实现持久性，可以使用日志和日志缓冲区体系结构。为简单起见，在以下讨论中，我们假设采用物理型登录方式。

当事务更新一个数据项时，系统除了把更新保存到该事务的意向列表里外，还要把包含

后像的更新记录追加到日志缓冲区里。在这种情况下，既不需要先写性，也无需在更新记录中包含前像，因为该数据库项一直要等到其事务提交后才更新。当此事务提交时，系统把一条提交记录追加到日志缓冲区里，随后它被强制移到非易失型存储器的日志中。由于在日志中更新记录是先于提交记录的，所以该提交记录的强制，保证所有的更新记录也都移到了非易失型存储器里。当提交操作完成以后，系统便根据该事务的意向列表来更新数据库内容，然后释放它所持有的锁，并且向该日志写入一条**结束记录**（completion record）。

系统可能在该事务已经提交，但尚未完成所有的数据库更新时崩溃。由于该事务的意向列表已丢失，恢复过程只得使用其日志来完成该事务的更新安装。为加速这一过程，系统周期性地向该日志追加一条检测点记录，其中包含了其意向列表正被用于更新此数据库的已提交事务的标识。重新启动后，恢复过程需要确定在崩溃发生时，其意向列表还未处理完成的已提交事务的标识。为了实现这一目标，需要利用结束记录及与上述精确检测点类似的某个算法，通过反向扫描该日志，找到最近的检测点记录。然后，再利用这些事务的更新记录来更新此数据库。因此，延迟更新系统的恢复类似于立即更新系统恢复过程的第2遍。

值得注意的是，恢复过程不再关注回退由崩溃时尚在活动的事务所履行的数据库更新（第3遍），因为活动事务并没有更新数据库。因此，检测点记录只列出了未完成的事务。

25.4　介质故障的恢复

持久性要求已提交事务写入的所有信息都不得丢失。因此，我们现在需要考虑介质故障。

实现持久性的一种简单的方法是，在两台不同的非易失型设备（可能由不同的电源供电）上维护该数据库的两份不同的拷贝，这样的两台设备不太可能同时发生故障。镜像化磁盘就是实现这种方法的方式之一。**镜像化磁盘**（mirrored disk）是一种大容量存储系统，每当提出写入一条记录的请求时，该系统就在两个不同的磁盘中都写入同一条记录。因此每一个磁盘都是另一个磁盘的精确拷贝，或称为**镜像**（mirror image）。此外，这个双重写入操作对于请求者是透明的。

如果出现的只是单一介质故障，那么储存在镜像化磁盘中的数据库就是持久的。此外，如果其中一个镜像化磁盘发生了故障，该系统仍然保持可用，因为可以利用另一个磁盘继续操作。当置换发生故障的磁盘以后，系统必须对两个磁盘实施再同步化。与之相反，当采用日志来达到持久性时（下文进行描述），恢复一次介质故障可能需要花费相当长的时间，在此期间，用户不能使用该系统。

即便是使用镜像化磁盘的立即更新系统，也必须利用先写型日志来实现原子性。因此，每当一个事务异常中止时，仍然需要用前像来回退数据库项；每当一个事务提交时，也仍然需要用后像来前滚数据库项。

实现持久性的第二种方法是，一旦出现介质故障，就根据其日志来对数据库进行复原。达到这个目标的一个办法是从该日志起点开始，通过更新记录的后像正向推演该日志。然而，考虑到日志的大小，这个办法是不现实的。这需要耗费大量的时间，而且在这一过程中，系统是无法使用的。另一个解决办法是，周期性地对数据库做一份**存档拷贝**（archive copy），即**转储**（dump）。

使用转储进行恢复取决于转储的实施方式。对于某些应用程序，可以脱机进行转储。系

统在适当的时候就不允许开始新的事务，并且等到现有活动事务都已经终止，这时系统就关闭。然后开始实施转储。一旦转储实施完毕，系统就重新启动。为了在介质故障后恢复数据库，重新启动后，系统首先使用转储文件，然后对转储记录之后追加的日志记录处理两遍：1) 一次反向扫描，产生所有转储后已提交的事务的一张列表，2) 一次正向扫描，把上述列表中所有事务的重做记录复制到数据库里。

模糊转储

在许多应用程序中，系统是不能关闭的。这就需要在系统的运行过程中进行**模糊转储**（fuzzy dump）。模糊转储就是无视锁的存在，顺序读取该数据库里的所有记录。这样，事务可以一边执行转储，一边更新记录，以后再提交或异常中止事务。该转储程序在这些记录写入以前或者以后都可以读取它们。

让我们考虑一个使用物理型登录的立即更新系统。如果上述的两遍恢复过程采用模糊转储，那么第2遍（正向）扫描（从该日志）恢复自**启动转储**（start dump）后提交的事务所写入的每一个数据库项的值，而不管转储是否真正读取过该值。如图25-8a所示，转储记录的x的值反映了T的影响，但是y的值却不能反映这一影响。然而，由于T的提交是在转储开始之后，所以，所有修改的后像都将用于由该转储启动的数据库重构。这对于x是不需要的，可是需要前滚y到它的正确值。该恢复过程也可处理图25-8b所示的情况。其中某个事务在**转储完成**后再启动，但后来异常中止了，因为它的更新记录在第2遍中忽略了。

但是两遍扫描过程并不能正确地处理以下两种情况：

- 在转储开始前提交的事务T所写的数据库页面在转储完成之前可能未被写入数据库。在这种情况下，转储不包含T写的新值，但是T不包含在第1遍扫描获得的记录已提交事务的列表里，这些事务的更新记录在第2遍扫描中用来前滚数据库的值。这种情况的发生是因为T的提交记录位于转储记录之前，而且正向扫描是从转储记录开始的。
- 转储有可能读取在转储时活动但后来异常中止的事务所写入的值。在这种情况下，转储中记录的该值不会被回退。这种情况如图25-8c所示。

为了解决这些问题，模糊转储采取与模糊检测点相同的策略。

1) 在开始转储之前，先将图25-7所示的检测点记录CK2追加到日志里，其后紧接着一条**转储开始**（begin dump）记录。正如此图所示，日志中CK2的出现确保当CK1追加时，出现在高速缓存中的所有的脏页面已经写入数据库，因而也记录在转储里。

2) 补偿日志记录用来记录异常中止处理时对于更新的逆操作。

为了恢复数据库，系统首先重载转储文件，然后对日志做如下三遍扫描：

- **第1遍**　扫描处理从日志尾部开始，反向扫描到最近的检测点记录。在这一遍扫描中，系统产生一个包含所有故障发生时还处于活动状态的事务列表L。
- **第2遍**　扫描处理从转储开始时第2个最近检测点记录（图25-7中的CK1）开始，正向扫描到日志的结尾。在这一遍扫描中，系统使用所有事务的重做记录。从转储中记录的状态开始来前滚数据库。
- **第3遍**　扫描处理从日志尾部开始，反向扫描到L中最早的某一事务的开始记录。在这一遍扫描中，系统使用L中所有事务的撤销记录来回退它们的影响。

对于那些可能未包含在转储中的更新，它们的所有重做记录(包括补偿日志记录)在第2遍

扫描处理中都被重演。这样，在第2遍扫描结束时，数据库的状态将从转储中记录的状态前滚为故障发生时的状态。仅有介质故障发生时仍处于活动状态的事务所产生的影响需要消除，这些工作在第3遍扫描里完成。

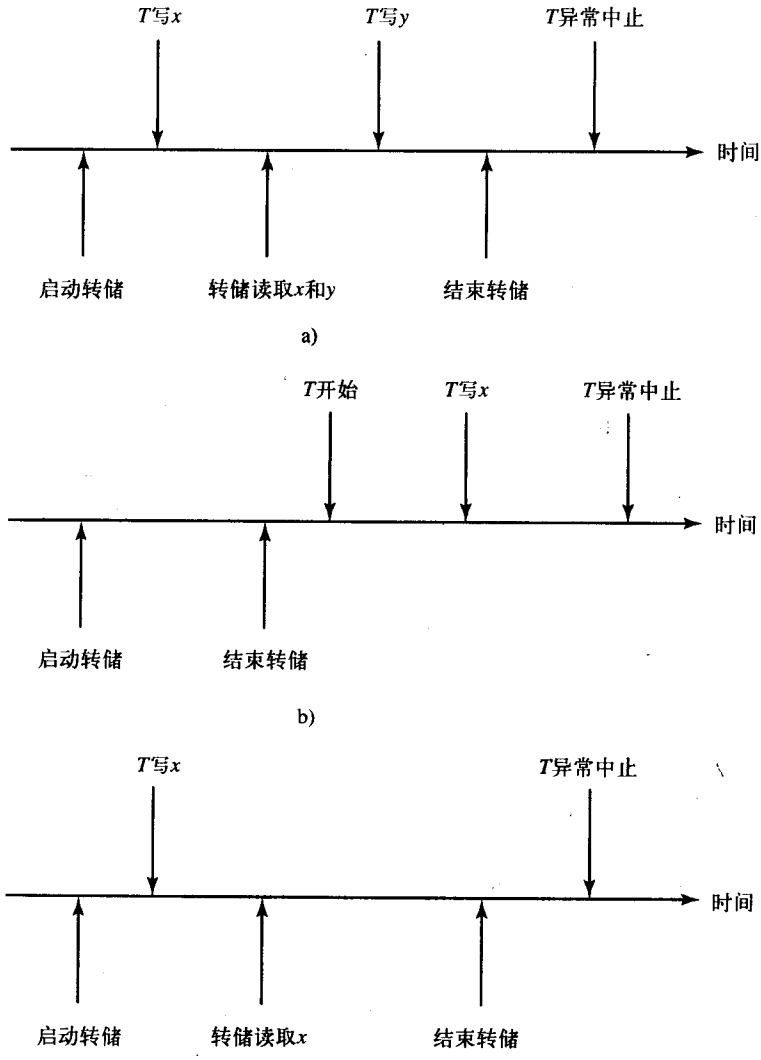


图25-8　实施转储时活动事务的作用

总之，介质故障的恢复需要数据库的最近转储以及一部分日志，这部分日志包括了没有包含在转储中的所有的更新操作的全部更新记录。应该注意到，该部分日志通常比崩溃恢复需要的部分大一些。这两部分都必须包括最早活动事务的开始记录。然而，崩溃恢复还需要包括两个最近检测点记录的那部分日志，而介质恢复需要转储开始前的两个检测点记录。此外，如果最近存档的数据库拷贝被损坏，那么利用相同的算法可以把数据库恢复到某个更早的拷贝。

使用物理逻辑型登录，不用在第2遍和第3遍扫描中不分青红皂白地应用前像和后像，可

以像25.2.3节描述的那样使用LSN来确定是否应该应用某项操作。

25.5 参考书目

[Gray 1978]是最早讨论登录和恢复技术的文献之一。当前技术大多是建立在System R[Gray et al. 1981]和Aries（Algorithm for Recovery and Isolation Exploiting Semantics）[Mohan et al. 1992]的实现的基础之上的。在[Haerder and Reuter 1983，Bernstein and Newcomer 1997，Gray and Reuter 1993]里包含有关于该技术的精彩概述。关于故障的一个更为抽象的观点，即把恢复和可串行化集成到单个模型中，是在[Schek et al. 1993]里描述的。

25.6 练习

25.1 请描述以下日志记录的内容，以及在从崩溃和介质故障中回退与恢复时，如何利用这些记录？

a. 异常中止记录
b. 开始记录
c. 转储开始记录
d. 检测点记录
e. 提交记录
f. 补偿日志记录
g. 结束记录
h. 重做记录
i. 存储点记录
j. 撤销记录

25.2 假设并发性控制采用表锁，某个事务是完成对一张表里的一个元组的一个属性的值进行更新的操作。那么该更新记录应当包含整张表的映像还是仅仅一个元组的映像？

25.3 假设由于两个活动事务写入而在高速缓存里产生了某个脏页，其中一个事务想要提交。请描述高速缓存过程此时该如何工作？

25.4 假设数据库崩溃发生在某个事务提交(通过向日志追加一条提交记录)和它要释放其锁时刻之间。请描述在这种情况下系统是如何恢复的？

25.5 请解释为什么在追加异常中止记录时，日志缓冲区不必进行刷新？

25.6 请解释为什么当物理型登录同时使用高速缓存和日志缓冲区时，在数据库保存的页面上无须包括LSN？

25.7 假设每个数据库页都包含最后一个把某数据库项提交并写入该页的事务的提交记录的LSN，而该系统采取的策略是，在日志缓冲区中的最早记录的LSN大于该页的LSN之前，不使用高速缓存对该页进行刷新。是否需要强制使用先写性策略？

25.8 精确检测点恢复过程的第2遍扫描如下：从检测点正向扫描日志。使用所有更新记录中的后像来更新相应的数据库项。证明这种更新可以按下述两种顺序完成：

a. 每当正向扫描碰到一个更新记录时，便完成相应的数据库更新操作（即使有不同的事务的更新记录在日志中交叠）。
b. 在正向扫描时，每个事务的更新记录保存在易失型存储器中，而对于每个事务的数据库更新，则是在正向扫描碰到该事务的提交记录时，一次性地全部完成的。

25.9 请说明在准确检测点恢复过程里，当利用日志的后像更新数据库时，系统是否必须获得锁？

25.10 考虑下述利用精确检测点和物理型登录进行崩溃恢复的两遍策略：第1遍是反向扫描，在这一遍中对活动事务进行回退。至于如何确定活动事务，请参见25.2节。第1遍至少一直延伸到碰到最早活动事务的开始记录或最近的检测点记录为止，无论哪个在日志里更早出现。在扫描时，每当碰到这些事务的更新记录，便对数据库应用它们的前像。第2遍是从最近检测点记录开始正向扫描，利用后像来前滚从已写入的检测点记录以来，事务所作出的所有变更（补偿日志记录是按与

通常的更新记录相同的方式进行处理的，以便使已异常中止的事务得到正确的处理）。该过程能否正常工作？

25.11 为了使逻辑型登录能工作，一个逻辑型数据库操作必须要有一个逻辑型逆操作。请给出没有任何逆操作的数据库操作的例子。请建议一个包括能够处理这种情况的逻辑型日志的过程。

25.12 请考虑在使用逻辑型登录时，利用在25.2节所描述的崩溃恢复程序（意指物理型登录）。请说明该过程应当如何修改来处理在恢复时出现的崩溃？假设每次更新的影响局限于单个页面。

25.13 请说明在延迟更新系统里，作为立即更新系统一部分的先写性，为什么在数据库项更新时却没有使用？

25.14 请说明为什么在延迟更新系统里，系统不是首先把意向列表拷贝到数据库里，然后向日志追加提交记录？

25.15 假设系统支持SNAPSHOT隔离。请说明为什么无须关闭系统就可以实施精确(非模糊)转储？

25.16 a. 请说明你的本地DBMS里，日志是如何实现的？

b. 请估计你的本地DBMS里，提交一次事务需要多少毫秒？

25.17 存放在数据库页里的LSN可看作日志里的描述该页最近更新的一个更新记录。假设某个事务完成了对某页的最后的更新，而后来却异常中止了。既然它对该页的更新已经被逆转，那么该页的LSN就不再能看作为合适的更新记录。为什么在本书的登录描述里，这不是一个问题呢？

25.18 一个机票预售系统需要具备性能标准和可用性标准。以下角色是否在增强性能方面起作用？它们能提高可用性吗？请说明你的理由。

a. 页高速缓存

b. 日志缓冲区

c. 检测点记录

d. 物理逻辑型登录

e. 镜像化磁盘

第26章 分布式事务的实现

26.1 ACID特性的实现

分布式事务（distributed transaction）是指通过网络在不同的站点上访问资源管理器。当此资源管理器是数据库管理程序时，我们把该系统称为**分布式数据库系统**（distributed database system）。每个本地数据库管理程序都可以导出预先存储的子程序，供分布式事务T作为子事务调用。事务T也可以提交单独的SQL语句，供数据库管理程序执行。在这种情况下，所提交的SQL语句序列就会在此管理程序中成为T的子事务。一般来说，T的子事务既可以顺序执行，也可以并发执行，而且一个子事务的结果可能会影响到其他子事务的执行。

当数据库系统所支持的组织是分散的，并且包含着各自固有的部分数据时，分布式数据库是十分有用的。可以通过把数据存放于访问频度最高的站点上将通信成本降到最小。而且，因为单个站点的故障失效并不妨碍其余站点上的继续操作，所以系统的有效性也得到提升。例如，学生注册系统就可以作为一个分布式系统来实现。学生的信息是存储在某台电脑上的，而课程的信息则可能是存储在另外一台电脑上的。这两台电脑也许被放置在校园内不同的大楼里。

我们在第18章讨论过与分布式数据库系统有关的数据库和查询设计的话题。在本章中，我们将讨论与分布式事务有关的话题。我们把单个的、执行某个全局性事务的子事务的数据库管理程序称为该事务的**伴随程序**（cohort）。一个为全局性事务执行子事务的数据库管理程序，是这个全局性事务的伴随程序，并且有可能同时是许多全局性事务的伴随程序。

我们希望每一个分布式事务都能满足ACID特性。主要负责处理此工作的模块被称为**协调程序**（coordinator）。在绝大多数系统中，事务管理程序就是协调程序。

分布式事务的伴随程序分散在网络各处。数据管理程序不仅仅作为网络中任意站点的分布式事务的伴随程序，同时还为本地的传统的（单一来源）事务的提供服务。鉴于数据库管理程序并不区分本地事务与全局性事务的子事务，所以我们常常把它们统称为“事务”。图26-1显示了某个分布式事务的数据访问路径。

现考虑某硬件制造厂商在全国范围内的分布式系统。该公司维护着一个其站点遍布全国的仓库网络，每个站点拥有各自的本地数据库，本地数据库中包含着本站点的库存信息。某站点的一个客户可能会启动一个事务来申请100打零件。该事务读取当前本地仓库中包含零件数量的数据项，发现本地仓库中只有10打。它（暂时地）先预订了10打，随后访问其他的一个或多个站点的数据，以预订其余的90打零件。当所有的100打零件都已找到且预订后，相应站点中的有关数据便会相应减少，并产生相应的装运命令。倘若事务无法找到100打零件，它便会释放所有预订的零件并提交，并向客户返回失败状态。在这种方式下，要么所有的站点都在其本地数据库中减掉被预定的零件数量，要么就全都什么也不做。

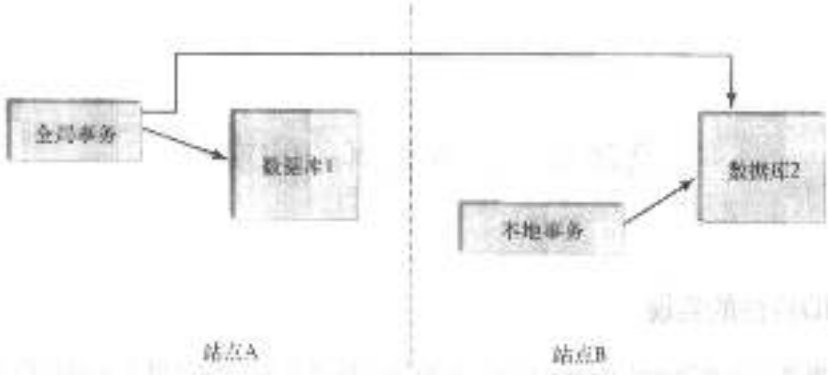


图26-1 某分布式事务的数据访问路径

当事务是分布式事务时，物理型故障会更加复杂一些。我们在第25章中已看到，系统崩溃是故障的一种最常见形式。在集中式系统里，一旦计算机崩溃，则与该事务有关的所有模块便全部失败。采用分布式事务后，网络中的某些计算机的崩溃，只会引起若干模块子集的失败，而其余的模块将继续执行。必须设计一些特殊的协议来处理这种新的故障模式。

当数据通信故障导致网络**被分割**（partitioned）时，也会出现类似的情况：运作站点不能彼此通信。我们将在本章的后面讨论此类故障的处理方法。我们假设一个事务可以异常中止（因此必须是可复原的）；同时假设一旦事务（在所有的站点）提交，系统必须确保所有的站点的数据库变更都是持久性的。

如果我们假设每一个站点都支持本地ACID特性，并且确保没有任何本地的死锁，那么分布式事务处理系统同样也会确保有如下的结果：

- **原子化终止** 分布式事务的所有伴随程序，必然是要么全部提交，要么全部异常中止。
- **没有任何全局死锁** 必然没有任何涉及多个站点的全局（分布式）死锁。
- **全局可串行化** 必然存在包括（分布式或本地的）所有事务在内的一种（全局型）可串行化实施。

我们将在本章中讨论以上话题，同时考虑数据复制以及与网络中数据的分布有关的话题。

26.2 原子终止

为了保证全局原子性，分布式事务只有在其所有子事务都提交时，才能提交。即使某个子事务已经完成其所有的操作，而且已做好提交的准备，事务也不能单方面地决定提交，因为其他的子事务有可能异常中止（或许已经异常中止）。在那种情况下，整个分布式事务也必须异常中止。因此，当某个子事务已成功地完成时，它也必须等待，直到所有其他子事务都完成后，才能提交。

协调程序是通过执行**原子提交协议**（atomic commit protocol）来保证全局原子性的。图26-2显示的是与该协议有关的通信路径。当一个应用程序启动某个分布式事务T时，它先通知其协调程序，然后设定其初始事务边界。每当T首次调用某个资源管理器的服务时，该管理程序便立即通知其协调程序它将要与该事务相联结。当事务T完成时，此应用程序便会通知其协调程序，接着设定其最终的事务边界。然后，其协调程序再启动此原子提交协议。

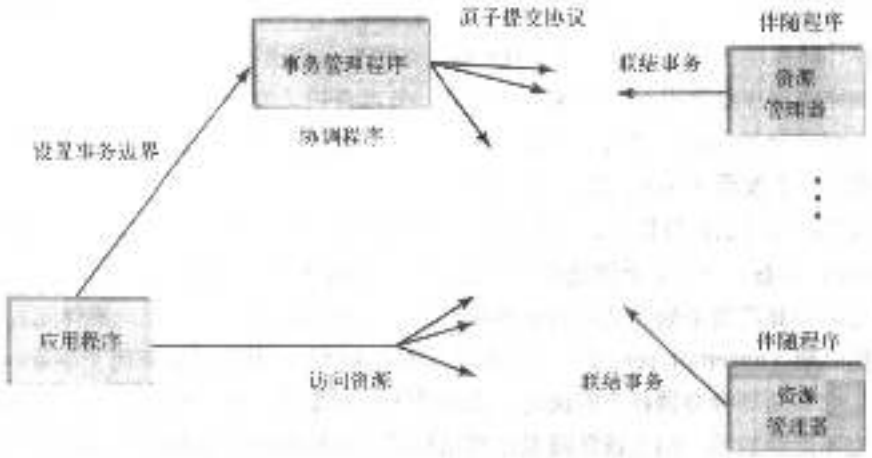


图26-2 与协议有关的通信路径

原子性要求当T完成时，要么所有的伴随程序都提交它们的变更，要么全都异常中止。因此，在处理T的提交要求时，协调程序必须首先确定是否所有的伴随程序都同意提交。以下是一个伴随程序可能无法提交的一些原因：

- 某伴随程序站点的模式可能规定有延迟约束检测（参见10.3节）。当该站点的子事务完成时，数据库管理程序可能判定此约束遭到破坏，并异常中止该子事务。
- 某伴随程序站点的数据库管理程序可能采用乐观型并发性控制。当该站点的子事务完成时，该管理程序便执行确认过程。一旦确认失败，该管理程序就异常中止此子事务。
- 因为（本地的）死锁或者与其他子事务（或本地事务）发生（本地）冲突，子事务可能被(本地的)并发性控制异常中止。
- 该伴随程序站点可能崩溃，因此无法响应协调程序发出的协议消息。
- 某些通信线路可能已失效，这妨碍了该伴随程序站点响应协调程序发出的协议消息。

26.2.1 两阶段提交协议

我们所要描述的特殊的原子提交协议被称为**两阶段提交协议**（two-phase commit protocol）[Gray 1978，Lampson和Sturgis 1979]。一旦事务请求提交，协调程序便启动此两阶段提交协议。为了执行协议，协调程序应该知道该事务的所有伴随程序的标识。因此，当事务启动时，协调程序便在易失型存储器里设置一条**事务记录**（transaction record）。每当有某个资源管理器联结到该事务时，它的标识便被添加到此事务记录里。因此，一旦事务请求提交，此事务记录立即便可拥有其所有伴随程序的一份清单。

我们不妨把该协议描述成在协调程序和伴随程序之间互相传递的一系列消息。这些消息是通过调用由事务管理程序和资源管理器所提供的过程而发出的。

当提交事务请求时，协调程序向每个伴随程序发送一条**准备**（prepare）消息，并始了此两阶段提交协议的第一阶段。发布此消息的目的是确定伴随程序是否打算提交。若打算提交，则要求它在非易失型存储器上储存所有子事务的更新记录以准备提交[⊖]。把更新记录保存在

⊖ 只完成读取操作的伴随程序站点，可以实现简化的协议版本，参见练习26.14。

非易失型存储器中可以保证，若协调程序随后决定该事务应当被提交，则其伴随程序也能提交；哪怕该伴随程序在肯定回答了准备消息后，系统发生崩溃也没有问题。

如果该伴随程序打算提交，它通过强制将一条**准备**记录写入日志，确保其更新记录是储存在非易失型存储器上的。然后，可以认为它已进入**准备**状态，并能够响应准备消息。每个伴随程序都用一个**投票**（vote）消息来回复此准备消息。

倘若该伴随程序正打算提交，其投票就处于**就绪**（ready）状态，否则，处于**正在异常中止**（aborting）状态。一旦某个伴随程序的投票处于就绪状态，它就不得再改变主意；因为协调程序就是利用此投票来确定是否将该事务作为一个整体提交。可以说，该伴随程序进入了一个**不确定时期**（uncertain period）；因为它不知道协调程序最终是提交此子事务、还是将其异常中止。它必须等待协调程序的决定，在这段等待时期里，在下述意义上，该伴随程序受到阻塞：它不能释放锁，而且该伴随程序数据库的并发性控制不能异常中止此子事务。这是一种令人遗憾的情况，我们将在26.6节再讨论这个问题。如果该伴随程序投票异常中止，它就立即异常中止子事务，并退出该协议。此协议的第一阶段到此就宣告完成。

该协调程序接收每个伴随程序的投票，并将其记录在事务记录里。如果所有的投票都是就绪，就可决定事务T能够整体地提交；同时在此事务记录里记下该事务已被提交的事实，并强制在其日志中写入一条提交记录。在此提交记录里包含该事务记录的一份拷贝。

对于采用单一资源的事务T，一旦提交记录被安全地保存在非易失型存储器里，T就被提交了。这时，所有伴随程序的全部更新记录都已在非易失型存储器里，因为每个伴随程序在投票之前都已强制写入了准备记录。值得注意的是，我们始终假设存在一个总的日志系统，其中，事务管理程序和每个本地数据库管理程序都拥有自己的独立的日志。

然后，该协调程序向每个伴随程序发送一条**提交**（commit）消息，催促其提交。本地提交过程如25.2节所描述的那样执行，涉及在该数据库管理程序的日志中强制写入此提交记录（以表示这一子事务已被提交），释放锁，本地清空。每个伴随程序在完成了这些操作后，便向其协调程序返回一个**完毕**（done）消息，表示它已完成了协议。

当该协调程序接收到了每个伴随程序的完毕消息后，它就在日志中添加一个**结束**（complete）记录，并且从易失型存储器里删除该事务记录。到此，该协议完成。对于一个已提交的事务，其协调程序要在日志中进行两次写入，其中只有一次是强制写入。

一旦该协调程序接收到任何正在异常中止的投票，它就将释放易失型存储器中的事务T的记录，并向投票提交的每个伴随程序发出**异常中止**（abort）消息（凡原先投票过异常中止的伴随程序都已经异常中止，并退出了此协议）。该协调程序并不在其日志里记录异常中止，因为此协议有一种预想的异常中止特性，我们将会在后面加以讨论。接着，该数据库管理程序便异常中止此伴随程序，并在其日志中写入一条异常中止记录。提交或异常中止消息的到达，标志着这个伴随程序结束了不确定时期。

图26-3显示了在应用程序、协调程序（事务管理程序）和伴随程序（资源管理器）之间交换的消息序列。

1. 两阶段提交协议小结

我们该两阶段提交协议小结如下。

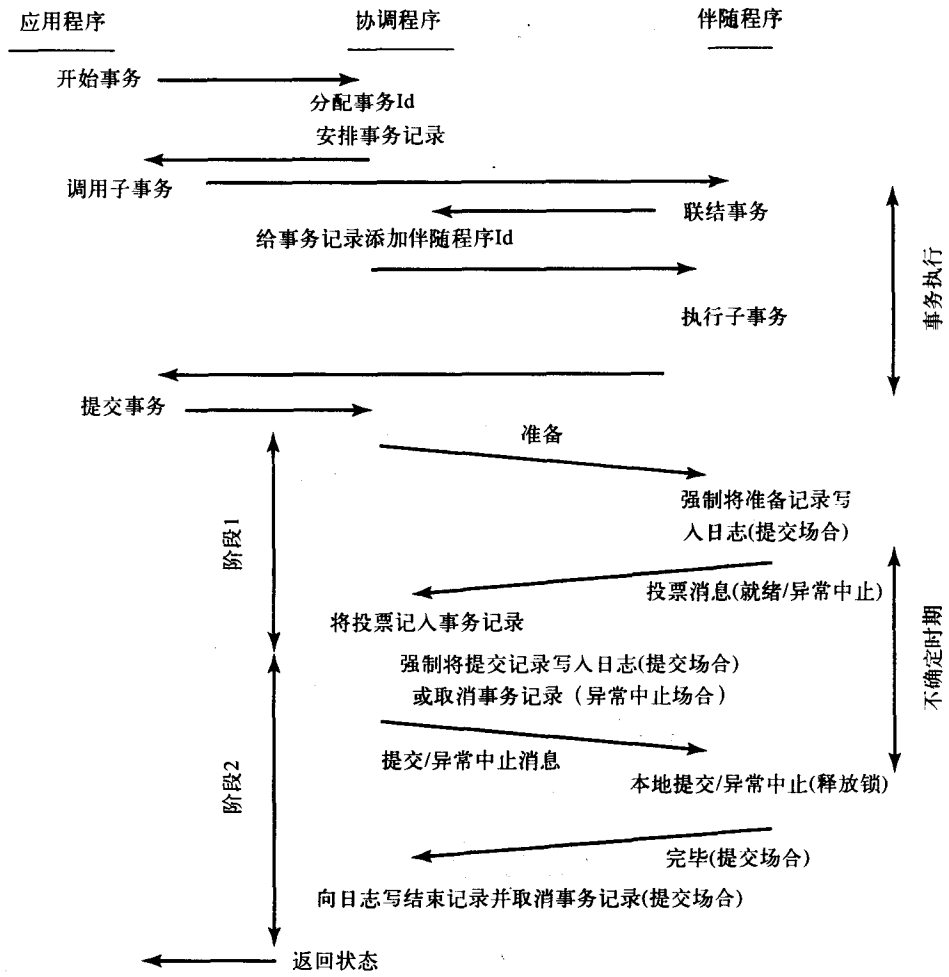


图26-3　两阶段提交协议中消息的交换

第1阶段

1) 该协调程序向所有的伴随程序发送一条准备消息。

2) 每个伴随程序等待直到收到从协调程序发出的准备消息为止。如果它已准备好提交，则向其日志中强制写入准备记录，并进入一种本地控制无法使其异常中止的状态，然后以投票消息形式向协调程序发出已就绪的信息。

倘若此伴随程序无法提交，它将在其日志中添加一条异常中止记录。或许它本来就已经异常中止。无论是哪种情况，它都将以投票消息形式向其协调程序发送正在异常中止的信息，接着回退此子事务对数据库作出的任何变更，并解除该子事务的锁，然后终止参与此协议。

第2阶段

1) 该协调程序等待直到接收到所有伴随程序的投票为止。如果至少收到一条正在异常中止的投票，它将决定异常中止：向所有投票就绪的伴随程序发出异常中止消息，释放其在易失型存储器里的事务记录，并终止参与此协议。

如果所有投票都是就绪，该协调程序将决定提交（并在其事务记录中记下这一事实）：强

制在其日志中写入提交记录（包括此事务记录的一份拷贝），并向每个伴随程序发送提交消息。

2）每个投票就绪的伴随程序都在等待接收该协调程序发出的消息。如果伴随程序接收到的是一条异常中止消息，它将回退该子事务对数据库作出的任何变更，并在其日志中添加一条异常中止记录，再解除该子事务的锁，然后终止参与此协议。

如果该伴随程序收到的是一条提交消息，它将在其日志中强制写入一条提交记录，并释放所有的锁，再向协调程序发送一条完毕消息，然后终止参与此协议。

3）如果该协调程序已提交此事务，它将等待直到接收到所有伴随程序发出的完毕消息。然后，在其日志中添加一条结束记录，并从易失型存储器中删除此事务记录。最后终止参与此协议。

2. 多重域上的原子化提交协议

在分布式事务中，协调程序的角色可能会由多个事务管理程序来担当，每个事务管理程序都在不同的域内。在这种情况下，原子提交协议中的消息将如图26-4那样沿着一棵树的边缘遍历。叶节点代表伴随程序，根代表控制域的事务管理程序，这个域中包含着启动事务的应用程序。树的内节点代表事务管理程序，它在自己的域中协调资源管理器，相对于其双亲事务管理程序来说，它起到伴随程序的作用。

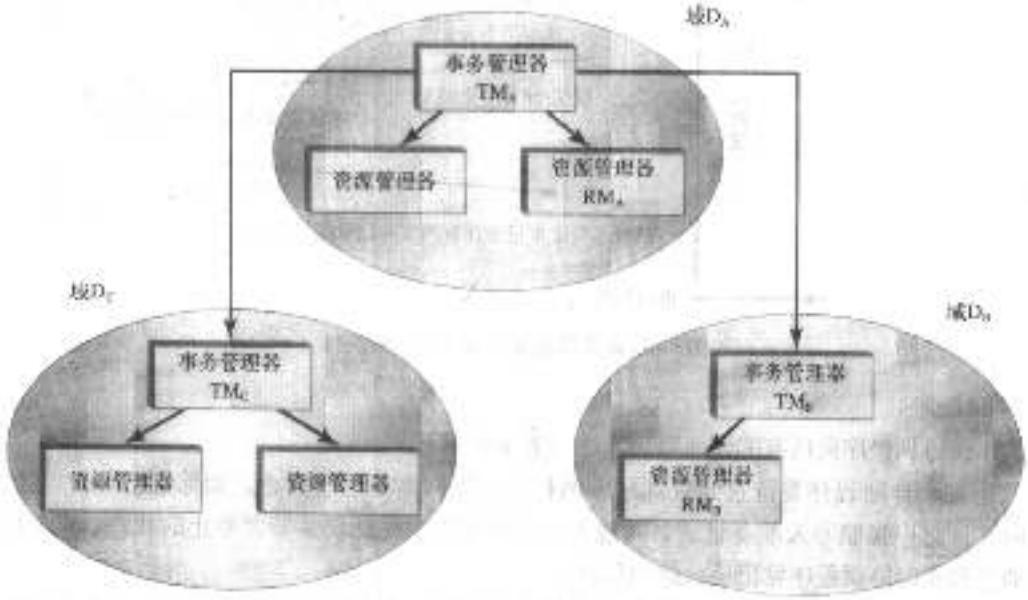


图26-4　树结构的分布式事务

启动该事务的应用程序向随后启动该协议的根事务管理程序发出提交请求。它向其所有的子管理程序（包括本地资源管理器和远程事务管理程序）都发出准备消息。一旦某个事务管理程序接收到一条准备消息，它就开始了协议的第1阶段，此时向其子事务管理程序发送准备消息并等待它们的投票消息。根据收到的投票，它再以适当的投票来响应其父程序。本图仅展示两层的协调程序，一般的分布式事务可以有任意的层数。

在本图中，当域D_A的事务管理程序TM_A收到提交请求后，便向其每个子程序发送准备消

息。当域D_B中的事务管理程序TM_B接收到此准备消息，它就与其子程序RM_B开始执行此两阶段提交协议的第1阶段。如果它从RM_B处接收到的是就绪投票，那么它就向TM_A发送就绪投票，作为对其准备消息的回复。类似地，TM_C在其子程序中执行此两阶段提交协议的第1阶段。倘若TM_A从TM_C处（以及站点A的两个资源管理器处）接收到就绪投票，它就提交该事务；并向其所有的子程序发送提交消息。然后，TM_B与TM_C在它们的子程序中执行此协议的第2阶段。通过这样的方式，提交消息就沿着树向下传播，而后，完毕消息又沿着树向上传播。采用这种一目了然的方法，该协议可以推广到更多层数的树。

26.2.2　两阶段提交协议中故障的处理

两阶段提交协议包括用来处理在分布式系统中可能出现的各种故障的协议。

- 假如一个站点等待消息超时，那么执行的是**超时协议**（timeout protocol）。（超时可能是由于发送的站点崩溃、消息丢失或消息投递系统速度缓慢造成的。）
- 当某个站点从崩溃中恢复时，执行的是**重启动协议**（restart protocol）。

如果直到某些故障修复为止，某站点仍无法完成提交协议，那么我们就称这个站点**阻塞**（blocked）。当某个站点阻塞后，提交或异常中止的决定将被推迟一段时间。当某个伴随程序站点采用加锁并发控制时，非常不希望出现这种推迟，因为在此站点中被该子事务锁定的数据项无法被其他事务读取。因此，了解该两阶段提交协议在怎样的状况下会导致阻塞是很重要的。我们会在下面讨论各种故障的情况。我们在描述这些情况时，假设该协议树恰好包含两层。不过，对于多层树的任意一对邻接层次，在其父程序与子程序之间履行同样的决策。

1. *伴随程序在等待准备消息时超时*

伴随程序可以断定，没有一个站点已决定提交（因为这些站点并没有投票）。因此，它可以决定异常中止。倘若随后到达一条准备消息，该伴随程序就可以简单地用正在异常中止来回复，这能防止所有其他的站点确定提交（因为提交决定需要得到所有站点的提交投票）。在这种方式下，全局的原子性就得到了保证。

2. *协调程序在等待投票消息时超时*

此时的情况同上例一样，该协调程序可以决定异常中止，并向所有的伴随程序发送一条异常中止消息[⊖]。

3. *伴随程序在等待提交或异常中止消息时超时*

由于伴随程序已投票就绪，并进入了不确定时期，所以这种情况更严重。它受到了阻塞，除非它能够确定其协调程序已经做出某个决定并且知道决定的内容。伴随程序无法单方面地作出异常中止或提交的决定，因为其协调程序可能会做出不同的决定，那样将违反全体一致性。

伴随程序可以试着联络其协调程序询问该事务的状态。如果无法询问（其协调程序已经崩溃或者网络被分割），那么伴随程序可以试着联系其他的伴随程序。（为了便于实现这种联系，协调程序可以在其准备消息中提供一份所有伴随程序的清单，伴随程序可以将其保存在写入日志的准备记录中。）如果伴随程序发现还有其他的伴随程序尚未投票，那么双方都可以

⊖ 值得注意的是，所有的站点都可能发送提交消息，可是在超时时期并未传递过该消息。在这种场合，该协调程序就异常中止该事务，哪怕其所有伴随程序都在运行，并投票提交——这是一种违背直觉的状况。

决定异常中止。这样的异常中止决定是安全的，因为协调程序无法在有伴随程序尚未投票之前决定提交。如果它发现某伴随程序已经提交或异常中止，它也可以做出同样的决定。如果它发现其余所有伴随程序都处在不确定时期，那么它将继续维持阻塞，直到能够建立与协调程序或某个提交/异常中止的伴随程序的联系为止。

4. 协调程序在等待某个完毕消息时超时

协调程序与请求完毕消息的伴随程序联系。一旦收到了所有伴随程序的完毕消息，它就释放其事务记录。

5. 协调程序崩溃

当协调程序在崩溃后再重新启动时，它将搜索其日志。对于某个事务T，如果它发现有一条提交记录但没有任何结束记录，那么它必定处于事务T的协议的第2阶段。协调程序就把（在T的提交记录中找到的）T的事务记录复原到易失型存储器中，并向其所有伴随程序发送提交消息来继续该协议，因为该协调程序可能在发送提交消息之前已崩溃。如果此协调程序没有发现某事务的提交记录，那么存在两种可能情况：

- 当崩溃发生时，该协议仍处在第1阶段。
- 其协调程序已经异常中止了此事务。

由于以上两种情况下协调程序都没有向其日志写入记录，所以它就无法辨别到底是其中的哪一种情况。幸运的是，协调程序无须区分这两种情况，因为无论哪种情况下事务都已异常中止。如果发生的是第一种情况，那么不久后伴随程序就会向其协调程序询问事务的状态。因为协调程序在其易失型存储器中找不到此事务的事务记录，它就会回复一条异常中止消息。这是允许的，因为当协调程序崩溃时，事务仍处于协议的第1阶段，因此协调程序在崩溃前还没有来得及做出任何决定。下面要介绍的预想异常中止协议就包含了这部分的原因。

6. 伴随程序在准备阶段崩溃或超时——预想的异常中止特性

当伴随程序在崩溃后重启动时，它将搜索其日志。如果它找到的是一条准备记录，而不是提交或异常中止记录的话，它就知道在崩溃发生时它正处于准备状态。它就会向协调程序询问该事务的状态。协调程序可能也已经重新启动，或者在伴随程序崩溃时，它仍是活动的。如果协调程序在易失型存储器中找到了事务的事务记录，那么协调程序立即可以回复上述请求。然而，倘若它没有找到事务记录，它将（不去查看其日志）假设该事务已经异常中止，并通知伴随程序。协议的这种特性称为**预想的异常中止**（presumed abort）[Mohan et al. 1986]。

预想的异常中止特性是很有用的[⊖]，因为该事务记录短缺意味着下述情况之一为真：

- 协调程序已经提交了事务，也接收到了所有伴随程序的完毕消息，并已在易失型存储器中删除了事务记录。
- 协调程序已崩溃，而它的重启动协议则在其日志中发现了该事务的提交记录和结束记录。
- 协调程序异常中止了该事务，并删除了事务记录。
- 协调程序已崩溃，而它的重启动协议在其日志中确实没有发现事务的任何记录（它异常

⊖ 预想（presume）一词的一种字典定义是："就像'预想是清白无罪的，除非证明有罪'那样，将希望或假定当作可能是真的"。然而，在本场合，该协调程序一定知道，该事务已经异常中止。

中止过该事务，或者当协调程序崩溃时处于协议的第1阶段）。

因为伴随程序仍处于准备状态（故而没有发送完毕消息），前面的两个可能性就被排除了。因此，协调程序就可以报告事务已异常中止。当伴随程序在其准备状态下超时，并向协调程序询问事务状态时，此原理同样适用。（注意，当在多重域内使用此提交协议时，该协议树里的每个协调程序都可以使用此预想的异常中止特性，来响应其伴随程序的请求。）

7. 用于两阶段提交协议的超时协议

TO1 伴随程序在等待准备消息时超时。该伴随程序决定异常中止。

TO2 协调程序在等待投票消息时超时。协调程序决定异常中止，对向其投票就绪的每个伴随程序发送一条异常中止消息，删除易失型存储器中的事务记录，并终止参与此协议。

TO3 伴随程序在等待提交/异常中止消息时超时。伴随程序试着与其协调程序通信以确定事务的结果。如果它能与协调程序通信，协调程序就会运用预想的异常中止特性来产生其回答。倘若不能联系与协调程序通信，它将试着与其他伴随程序通信。如果发现某个已提交或异常中止的站点时，它将做出同样的决定。倘若它发现还有一个站点没有投票，它们都可以决定异常中止。否则，此伴随程序就阻塞。

TO4 协调程序在等待完成消息时超时。协调程序向请求发送完毕消息的伴随程序发送一条消息。它会在易失型存储器中维持事务记录，直到接收到所有伴随程序的完毕消息为止。

这个在一次崩溃后重启动协调程序或伴随程序的协议，可以建立在25.2节所描述的崩溃恢复过程上面。

8. 用于两阶段提交协议的重启动协议

RES1

如果重启动站点是某个伴随程序站点，并且此站点的崩溃恢复过程在其日志中发现了某个事务的异常中止或提交记录，那么这个事务已经完成，其崩溃恢复过程也不会采取25.2节已讨论过的行为之外的其他行为。

RES2

如果该重启动站点是某个伴随程序站点，并且此站点的崩溃恢复过程在其日志中发现了某个事务的事务开始记录却没有发现准备记录（因此，此伴随程序还没有投票），那么其崩溃恢复过程也不会采取25.2节已讨论过的行为之外的其他行为（即，它将异常中止该子事务）。

RES3

如果该重启动站点是某个伴随程序站点，并且此站点的崩溃恢复过程在其日志中发现了某事务的准备记录却没有发现提交或异常中止记录（因此，伴随程序可能在崩溃之前就已经发送过就绪投票），那么该事务是在伴随程序处于不确定时期时崩溃的。崩溃恢复程序将复原该子事务所作的全部数据库更新。然后伴随程序将遵循协议TO3。

RES4

如果重启动站点是协调程序，并且它在其日志中只发现了一些事务的提交记录而没有发现结束记录，那么此崩溃是出现在协调程序提交事务之后，但在接收到所有伴随程序的完毕消息之前。当协调程序重启动后，将根据提交记录把事务记录复原到易失型存储器里。然后，

协调程序将遵循协议TO4。

26.2.3 格式和协议：X/Open标准

两阶段提交协议是与软件模块（如由不同厂商提供的DBMS和事务管理程序等）联系在一起的。如果这些模块之间要有效地沟通，或应用程序要与这些模块沟通的话，它们都必须要遵守通信约定，有时称为**格式与协议**（Format And Protocol，FAP）。FAP的标准化促进了不同厂商提供的产品之间的**互操作性**（interoperability）。

通过定义一组用于交换协议消息的函数调用以及这些消息的格式，X/Open标准允许互操作性。利用X/Open标准，由事务管理程序（协调程序）实现并由事务调用的函数的名字都带有前缀tx。例如，图26-3里启动某个事务所调用的函数是tx_begin()。类似地，从事务管理程序到资源管理器所调用的X/Open函数都带有前缀xa。于是，当资源管理器（伴随程序）打算联结到某个事务上时，它调用xa_reg()。当一个事务打算提交时，它便调用tx_commit()。事务管理程序通过xa_prepare()来调用每个资源管理器（以便发送准备消息），并等待着此函数的返回。每次调用xa_prepare()的返回值便是资源管理器的就绪或异常中止投票。倘若所有的投票都是就绪，那么事务管理程序就设定调用tx_commit()的事务程序的返回值为提交，并通过xa_commit()调用每个资源管理器。如果有一个或多个xa_prepare()的返回值是正在异常中止，则该事务管理程序就设定该事务调用tx_commit()的返回值为异常中止，并通过xa_abort()调用每个资源管理器。

X/Open标准提供了在采用RPC方式调用子事务的情况下实现全局型原子性的结构。例如，在本例场合，剩下的事情就是调用tx_begin和xa_reg。

26.2.4 对等原子提交协议

实现事务原子提交的两阶段提交协议有一种变体是应用对等通信（见22.4.3节）[⊖]。其中，同步点管理程序执行事务管理程序的作用。与某应用程序A相关联的同步点管理程序持有一份记录，记载着A能与之直接通信的所有应用程序和资源管理器。

假设A通过声明某同步点启动了该协议，那么A的所有连接必定处于发送模式。与A相关联的同步点管理程序通过向A所调用的所有资源管理器发送准备消息并向A连接的其他应用程序发送同步点消息开始协议的第1阶段。然后，A处于等待状态，直到该协议完成为止。当同步点消息到达另一个程序B时，该程序可能还没有完成其负责的事务部分。一旦它完成，而且它的所有其他连接（除了连接A以外）也都处于发送模式时，它便同样地声明一个同步点。这将导致与B相关联的同步点管理程序向B所调用的所有资源管理器发送准备消息，并向与B通信的其他应用程序（除A以外）发送同步点消息。然后，B处于等待状态，直到该协议完成为止。

同步点消息就以这种方法传播，并定义出与图26-4类似的一种树结构。假定所有的对等点都希望提交，那么每个对等点最终都将声明一个同步点。于是，所有的点都在其同步点声明处实施同步，而且与之关联的资源管理器也处于准备状态。投票就绪消息沿着该树自下向上传播，完成了该两阶段提交协议的第1阶段。接着，第2阶段开始，该事务将由同步点管理

⊖ 此处的描述基于[Maslak et al. 1991]。

程序树实施提交。提交的状态也会返回给每个程序，它们然后就可以启动某个新的事务来继续其执行过程。

如果某个点决定异常中止，那么它将会启动局部的回退，并将异常中止状态发送给所有的资源管理器，以使得它们也回退。此异常中止状态也会返回给每个程序，这些程序又可以再启动某个新的事务来继续其执行过程。

26.3 协调的传递

一般说来，与事务启动所在站点相关联的事务管理程序将成为协调程序。它与伴随程序通信，以便执行原子提交协议。这些伴随程序有可能是资源管理器，但它们一般更有可能是某个分布式事务树中的事务管理程序。

基于启动站点的协调，未必就是最佳的做法，其理由如下。第一，此启动站点可能并非是该事务中最可靠的站点。考虑到该事务有可能因为某个POS终端的某些操作而启动，并且还可能涉及到该商店服务器及客户所在银行的服务器。所以让协调位于这些服务器上可能更安全些。这样的话，可以修改协议，使协调程序的状态从一个参与者传递到另一个参与者。

协调的传递的第二个原因是，应当对必须在协议中交换的消息的数量加以优化。带有预想异常中止特性的两阶段提交协议在协调程序和每个伴随程序之间一般交换四条消息。对此进行优化是有可能的。例如，下述修改协议涉及两个参与者P_1与P_2，它们可以看成是（控制着一组伴随程序资源管理器的）事务管理程序或者服务器。

1) P_1进入准备状态，从而开始协议（如果P_1是事务管理程序，则其伴随程序全都处于准备状态）。接着，它向P_2发送一条消息，同时说明P_1已准备好本地提交，请求P_2准备并整体提交该事务。因此，该消息是投票就绪与准备消息的一种组合，并把协调程序的角色传递给P_2。

2) P_2接收到该消息，假设它打算提交，则进入准备状态。既然P_2知道P_1此时已准备，那么P_2就可以决定整体提交该事务，并采取本地提交该事务的必要操作。然后以提交消息回应P_1。

3) P_1收到此消息后，先本地提交，再以完毕消息回应P_2。

由于预想的异常中止特性，该完毕消息是必不可少的：作为新协调程序的P_2，必须能够记得该事务的结果，以防P_1没有收到提交消息的情况发生。届时，它就能够回答P_1对于该事务结果的询问。此完毕消息指示P_2，可以从易失型存储器中删除该条事务记录。此协议只交换了三条消息(而不是四条)就完成了，这一事实表明，需要交换的消息的数量是能够进行优化的。

26.3.1 线性提交协议

线性提交协议（linear commit protocol）是一种采用协调传递的两阶段提交协议的变形。我们设想伴随程序是通过某条（线性）链彼此连接的。不妨（任意地）假设最左端伴随程序C_1启动了该协议。一旦准备好提交，它就将进入准备状态，并向其右侧的伴随程序C_2发送一条消息，表明其已准备好提交，同时把协调传递给C_2。收到该消息后，如果C_2打算提交的话，它也进入准备状态，把该消息转播给其右侧的伴随程序，并再一次传递协调。这样的过程一直持续下去，直到该消息到达最右端的伴随程序C_n为止。如果C_n同意提交，它将实施提交，并向C_{n-1}发送一条提交消息。C_{n-1}又再提交并沿着此链传递此消息，直到此消息到达C_1为止，

然后C_1提交。最后，一条完毕消息又沿着此链从C_1传到C_n以最终完成此协议。

在收到第一个协议消息后，如果某个伴随程序决定异常中止事务，则事务就异常中止，并将异常中止消息发送给其左侧和右侧的伴随程序。那些伴随程序也异常中止，并将异常中止消息继续沿着链传播直到到达其两端为止。

用来在发生各种故障时实现原子性的登录、超时和恢复协议等技术与两阶段提交协议中的相应技术很相似（参见练习26.11）。

线性提交协议比两阶段提交协议包含更少的消息，因此节约了通信开销。假设有n个伴随程序，那么线性提交仅需要$3(n-1)$条消息，然而两阶段提交协议则需要$4n$条消息（并涉及一个单独的协调程序）。另一方面，两阶段提交只需经过4次消息交换即可完成（与伴随程序数目无关），因为协调程序与所有伴随程序的通信都是平行的。可是线性提交却需要$3(n-1)$次消息交换才能完成，因为其消息是串行发送的。

线性提交的概念我们将在第27章讨论的因特网事务协议中介绍。

26.3.2 无准备状态的两阶段提交协议

可以用协调传递的基本思想来调整两阶段提交协议以适应下述情形，其中(恰好)有某个伴随程序C并没有领会该准备消息，因而也没有准备状态（当C是老式遗留系统时，有可能发生这种情况）。在这种情况下，协调程序将和那些支持准备状态的伴随程序一起按通常的方式执行此协议的第1阶段。如果所有的伴随程序都赞成提交，那么该协调程序将向C发送一条提交消息，此消息将有效地允许C来决定是否整体提交该事务。倘若C向该协调程序响应说其已经提交，那么该协调程序将向其他伴随程序发送提交消息，并完成协议的第2阶段。倘若C响应说它已经异常中止，那么该协调程序将向其他伴随程序发送异常中止消息。

值得注意的是，C实际上确实没有协调程序的功能，因为它并没有采取处理故障的必要步骤。它既不维护该分布式事务的事务记录，也不能理解完毕消息。因此，它无法响应其他伴随程序的是否发生故障的询问。因此，协调没有得到完全的传递。

26.4 分布式死锁

使用等待的悲观型并发性控制易发生死锁。假设每个本地并发性控制都不允许有本地死锁，我们期望确保整个系统不会遭受**分布式死锁**（distributed deadlock）。例如，在站点A和B中都拥有其伴随程序的两个分布式事务T_1和T_2之间，将会产生一种简单的分布式死锁。若在A站点的并发性控制使得T_1的伴随程序T_{1A}等待T_2的伴随程序T_{2A}，与此同时，在B站点的并发性控制使得T_2的伴随程序T_{2B}又等待T_1的伴随程序T_{1B}，此时就会发生死锁。请注意，在分布式事务的一般模式中，伴随程序是允许并发运行的，而访问单一数据库的事务却只能纯串行地进行。因此，在上例中，T_{2A}不仅持有T_{1A}所等待的资源，而且由于它并没因为等待T_{2B}而推迟，所以它实际上还能够运行。然而死锁仍然将出现，因为T_2不能在其全局化提交之前释放锁，而这在T_{2B}仍在等待的情况下是不会发生的。在这类情况下，陷入死锁事务的所有过程中止。

一般说来，分布式死锁是无法通过异常中止或重启动单个的伴随程序来消除的。某个伴随程序所执行的语句，实际上是该事务整体所执行的语句的一个子序列。因为其他的伴随程序可能已经执行了逻辑上在这些子序列之后的语句，所以它们就不能再重复执行。例如，在

死锁发生之前，T_{1A}计算的结果可能已经传输给了T_{1B}。在这种情况下，重启动T_{1A}但不同时重启动T_{1B}的操作是毫无意义的，因此，整个分布式事务必须重新启动。

用于侦测分布式死锁的技术是23.4.2节讨论的技术的简单延伸。其一，该系统将建立一个分布式的waits_for关系，并搜索是否存在让某个伴随程序等待的循环。例如，T_1的伴随程序通知其协调程序，它正在等待T_2的某个伴随程序。T_1的协调程序则随后向T_2的协调程序发送一条**探查**（probe）消息。如果T_2的协调程序收到过其某个伴随程序发出的正在等待T_3的某个伴随程序的通知，那么这个探查消息就会通过T_2的协调程序转送给T_3的协调程序。倘若该探查消息又返回到了T_1的协调程序处，那么就侦测到了一个死锁。

其二是使用超时技术。一旦一个站点的某个伴随程序的等待时间超过了某一阈值，则此站点的并发性控制就认定存在死锁，并异常中止此伴随程序。

最后，将23.4.2节所描述的**时间戳技术**（timestamp technique）[Rosenkrantz et al. 1978]稍加推广，也可以应用到分布式事务中。一旦启动某个分布式事务，其协调程序就以此协调程序站点上的时钟作为时间戳的基础。为了确保所有的时间戳都是全局型唯一的，时间戳是通过把其协调程序（唯一的）站点的标识符添加到时钟值的低位上而形成的。当该事务在一个站点上创建了某个伴随程序时，它将同时发送其时间戳的值。由于分布式事务的所有伴随程序都使用单一、唯一的时间戳，所以利用不允许旧事务等待新事务的策略，就能够消除分布式死锁。

26.5　全局可串行化

在集中式系统中，并发控制的目标是响应各种访问数据库项的请求，以便产生某个特定的隔离级别。在一个分布式事务中可能会涉及多个DBMS，而各个DBMS又可能支持不同的隔离级别。在这样的情况下，我们就只能很粗略地定义并发分布式事务之间的隔离级别。然而，现在假定我们需要考虑如何实现全局可串行化调度的问题。最简单的办法就是在某个中央站点提供单一控制。任何站点上的所有的请求都发送到这个站点，此站点维护着对所有站点的数据项实现加锁的数据库结构。

这样的系统只是一种数据为分布式的集中式并发控制系统。但是，这种简单的办法存在着很大的缺点：它需要大量的通信（这就意味着会带来延迟），中央站点是个瓶颈，而且整个系统会因为中央站点的故障而变得极其脆弱。

另一种更好的、常用的办法是，每个站点维持其本地的并发控制。一旦一个（子）事务在某些站点请求实施某项操作，此站点的并发性控制则并不与其他任何站点通信，而是仅仅依据本地可利用的信息做出决策。每个并发性控制都分别地运用先前所描述的技术来确保其所产生的调度至少等价于其站点上的全局事务的子事务和事务的一种串行化调度。该系统的总体设计必须确保全局事务是全局串行化的，即至少存在一个所有站点都赞成的、等价的串行顺序。

因为站点独立运行，甚至有可能使用不同的并发性控制算法，所以我们关心的是是否存在所有站点都赞成的排序。不妨考虑在A站点和B站点中都拥有伴随程序的全局型事务T_1和T_2。在站点A，T_{1A}和T_{2A}之间可能存在冲突操作，而按照T_{1A}，T_{2A}的顺序是可串行的；同时，同样的两个事务在站点B的操作也起冲突，而按照T_{2B}，T_{1B}的顺序也是可以串行的。在这种情况下，

T_1和T_2之间总体上不存在任何等价的串行调度。

我们可以肯定，只要采用个别的并发控制和两阶段提交协议，每个站点的本地可串行性就蕴涵着全局型可串行性。特别地，在[Weihl 1984]中可以看到：

> 如果每个站点的并发性控制独立地使用严格的两段锁或采用乐观算法，并且系统使用某种两阶段提交协议，那么任何全局调度都是可串行化的（按照各自协调程序提交它们的顺序）。

尽管我们在此没有证明结果，但不难看到可以扩展成证明的基本论据。我们假设A站点和B站点使用严格的两阶段加锁并发控制，其两阶段提交算法用来确保全局原子性，事务T_1和T_2则如上所述。我们用反证法来证明。假定上述结论不正确（即这些事务不是可串行化的），并且两个事务都已提交。假设在A站点的某个数据项上T_{1A}和T_{2A}发生了冲突，于是，在T_{1A}释放对那个数据项所加的锁以前，T_{2A}是无法完成的。由于该并发控制是严格的，而且使用了两阶段提交算法，所以T_{1A}要到T_1提交后才会释放该锁。但由于T_2在T_{2A}完成之前是无法提交的，所以T_1必须在T_2之前提交。然而，如果我们在站点B也进行同样的推理，我们就会得出T_2必须在T_1之前提交的结论。因此，我们得出了矛盾的结论，于是两个事务不可能都已提交。事实上，我们在站点A和站点B上所假设的冲突产生了一个死锁，因此必须异常中止其中的某个事务。

26.6　不能保证全局原子性的场合

在实际操作中，有许多原子提交协议不能完成的场合，因此也无法保证全局原子性。

- **伴随程序站点不参与两阶段提交**　特定的站点可能不支持两阶段提交协议。例如，某个资源管理器属于遗留系统，该遗留系统并不支持准备状态。另外，一个站点可以选择不参与，从而避免因阻塞而产生的性能降级。事务在不确定时期里是受阻塞的，并且一直持有锁，以防止其他事务访问已锁定的数据项。由于该站点无法控制不确定时期的长度，所以它不再是独立的，因为控制对加锁资源访问的因素是受到别处控制的。例如，不确定时期的长短取决于其他伴随程序对准备消息的响应速度以及消息传递系统的效率。如果该协调程序崩溃或是网络被分割，那么伴随程序就可能会在相当长的一段时间里持续阻塞。

选择参与协议的伴随程序站点常常通过单方面决定提交或是异常中止某个受阻塞的子事务来释放锁，从而解决这样的问题。这类决策称为**启发式决策**（heuristic decision）。

但是，给以上的动作指定某个技术性的名称并不能改变以下事实：此动作的结果可能会影响全局原子性。一个启发式决策可能会导致该数据库的全局不一致性（在某站点上更新数据项的某个子事务可能提交，而同一事务在另外的站点上更新相关数据项的另一个子事务就可能异常中止）。这样的不一致性有些时候可以利用即席方式，通过不同站点上的数据库管理员之间的通信来解决。虽然启发式决策并不能保证全局原子性，但是它是解决这个重要的实际问题的唯一方法。

伴随程序也可能出于性能以外的原因而不参与协议。例如，某站点对执行子事务收费，当该子事务完成并将结果返回给应用程序时，哪怕整个事务随后异常中止，此站点仍然会要求收费。为此，该站点管理员可能坚持该子事务一完成后就马上提交，而不用等待此事务在整体上是提交还是异常中止。

- **语言不支持两阶段提交** 在应用程序方面，虽然JDBC、ODBC和大多数版本的内嵌式SQL允许事务连接到多个DBMS，但它们并不支持两阶段提交。当一个事务完成时，它一次一条地向每个DBMS发送单个的提交命令。这样，两阶段提交协议所提供的all-or-none（全部或没有）提交并没有得到强制实施，因此，其事务不能确保全局原子性。然而，一个新提议的包括TP监控器（和相应的API）在内的新Java API——Java事务服务（Java Transaction Service，JTS），可以确保采用JDBC的分布式事务的原子化提交。微软也推出一款新的TP监控器——微软事务服务器（Microsoft Transaction Server，MTS），其中包含一个事务管理程序（及其相应的API），它确保采用ODBC的分布式事务的原子化提交。此外，某些数据库厂商提供的内嵌式SQL在该厂商生产的DBMS中（即在同构系统中）支持两阶段提交。
- **系统不支持两阶段提交** 如果要求支持两阶段提交协议，就需要**系统中间件**包括一个协调程序（事务管理程序），而且要求应用程序、协调程序和服务器遵守消息交换协定。例如，X/Open标准为此目的而专门指定了API（如tx和xa接口）。对于许多应用程序来说，这样的系统支持是无法提供的，所以两阶段提交也就无法实现了。

弱提交协议

如果站点没有参与两阶段提交协议（或某个其他的原子提交协议），那么分布式事务的执行就无法保证其隔离性或原子性。尽管如此，设计系统时仍然需要遵守这种约束。在这种情况下，应用程序的设计者必须仔细地评估，此约束是如何影响该数据库的正确性，以利于客户最大程度地使用该应用程序。

在不支持两阶段提交时，可以使用以下的某一项**弱提交协议**（weaker commit protocol）。

- 在**一阶段提交协议**（one-phase commit protocol）中，应用程序在其所有的子事务完成之前，并不向任何站点发送提交命令。届时，它再向其访问的每个站点单独地发送提交命令。有些站点可能提交，有些则可能异常中止。用JDBC实现的分布式事务就能够以这种方式运作。
- 在**零阶段提交协议**（zero-phase commit protocol）中，每个子事务只要完成在某站点中的所有操作，就立即提交。同样，有些站点可能会提交，有些则可能异常中止。注意，仅当子事务在某站点提交后，应用程序才可以在其他的站点启动新的子事务。在这类协议里，应用程序每次只启动一个子事务。只有当这个子事务完成并提交后，应用程序才启动另一个子事务。
- 在**自动提交协议**（autocommit protocol）中，在每个SQL语句之后立即完成一次提交操作（由系统自动地执行）。在其ODBC或JDBC中，自动提交总是默认的。

以上这些协议无一能确保全局原子性，因为子事务（或操作）有可能在某些站点提交，但在另外的某些站点却异常中止。

在所有的子事务全部提交的正常情况下，如果每个DBMS都使用严格的两段锁算法或某个乐观型并发控制算法，那么上述一阶段提交协议可以确保全局可串行性。

如果满足以下三个条件：

1) 每个站点的并发性控制都独立地使用严格的两段锁算法或乐观型算法。

2) 应用程序使用一阶段提交协议。

3) 所有站点的子事务都提交。

那么，所有的全局调度都是可串行化的（按照应用程序提交此事务的顺序）。

因此，该事务维护了所有全局和本地的完整性约束。

根据同样的推理，通过增加所有站点的子事务都必须提交这一约束，我们可以证明这里应用的两阶段提交有类似的结果。如果某个站点的子事务异常中止，使得其全局事务无法保证原子性；那么再次根据同样的推理可知，该事务中已提交的那部分是可串行化的。

采用零阶段协议，某站点的子事务在启动另一站点的某个子事务并请求锁之前，可以提交并释放其站点上的锁。所以，从全局观点来看，即便所有站点的子事务都提交，加锁也不是两阶段的，因此全局可串行性也得不到保证。事务有可能在不同的站点按不同的顺序实现可串行化。在每个站点上执行的子事务保持其本地完整性约束，但是其全局事务则未必能保持全局完整性约束。

零阶段提交协议持锁的时间要比一阶段提交协议少。因此，它可以提供较好的性能，但并不能确保全局可串行化。因此，零阶段提交协议特别适合于那些没有任何全局完整性约束的应用，所以全局型可串行性也不成问题。然而，本地可串行性还是应当保持的。

自动提交协议持锁的时间最短，因此性能最佳。但是，它甚至连每个站点的本地可串行性都无法保证。正因为这个原因，无论是全局完整性约束还是本地完整性约束，它都不能确保。

在一阶段提交协议和零阶段提交协议中，如果其应用程序要求某站点的子事务提交，但该子事务反而异常中止的话，该应用程序就会收到异常中止的通知，并有可能采用某些其他方法（或许通过在另外的站点启动某个新的子事务）来实现预期的结果。对于某些应用程序来说，这个特点与两阶段提交协议相比具有明显的优势。因为，在两阶段提交协议中，只要任意站点中的某个子事务异常中止的话，整个全局事务也就异常中止了。

26.7 复制数据库

处理故障的常用技术是在网络中的其他站点上复制每个数据库的部分内容。这样，如果某个站点崩溃或者由于分割而与网络分开，此站点所拥有的那部分数据库仍然可以通过连接其他拥有副本的站点来访问。因此，数据的**可用性**（availability）提高了。

复制也可以提高访问数据的效率（因此增加了事务吞吐量，减少了响应时间），因为一个事务可以访问存储与其距离最近的（或许就是该事务启动的站点里的）副本。例如，在18.2.3节讨论过的因特网食品商应用程序里，该公司拥有一张描述其顾客的表，它在其总部站点和交付顾客订单的本地仓库站点里都有副本。涉及商品交付的事务在仓库站点执行，该事务使用的是存储在仓库站点的副本；而每月向所有顾客发送邮件的事务在总部站点执行，使用的是存储在总部站点的副本。

当然，复制是需要代价的。首先，需要更多的存储空间。其次，因为我们需要正确地安排对副本数据的访问，所以系统变得更加复杂了。例如，如果我们允许两个事务访问，并且它们可能同时更新同一个数据项的不同副本，而每个事务可能都不知道将会给对方带来什么影响，那么结果就会产生与更新丢失相类似的问题。因此，一个复制系统必须要确保每个数

据项的副本都能正确地更新，并且向请求读取那些数据项的事务提供相应的值。在因特网食品商应用程序中，复制后的顾客信息只需要在顾客信息变动（例如，地址改变，但这不经常发生）后更新即可。

如果某数据项的副本在每个站点中都存在，那么称将此数据项**完全复制**（totally replicated）。如果某数据项的副本仅在部分而不是全部站点中存在，那么称将此数据项**部分复制**（partially replicated）。

如果DBMS本身不支持复制，那么应用程序也能够复制数据项。那时DBMS将不会意识到，不同站点中各个数据项其实互相都是副本。如果x1和x2都是某同一个数据项的副本，那么很明显，每个事务都必须保持完整性约束x1 = x2。访问复制数据项的每个事务必须规定之需要的是哪个副本，更新复制数据项的事务也必须明确地启动了事务来更新此数据项的每个副本。

绝大多数商用DBMS不是通过事务来管理复制的，而是为管理复制提供了一个特别的子系统——副本控制。它使得应用程序无法看到复制。副本控制知道每个数据项的所有副本的位置。当某个事务请求访问数据项时，它并不指定某个特定的副本。副本控制自动地将此请求翻译成访问相应的副本的请求，并将该请求传递给本地并发性控制（倘若该副本在本地站点）或（该副本驻留其上的）远程站点。我们假设该并发性控制实施某个加锁协议，所以在访问副本时，仿佛它们采用与针对普通的数据项一样的方法来加锁。即读取访问时加共享锁，写入访问时加排他锁。并发性控制并不知道本站点中的某个表实际上是另一个站点的另一个表的一份拷贝。图26-5显示了副本控制与并发性控制之间的关系。

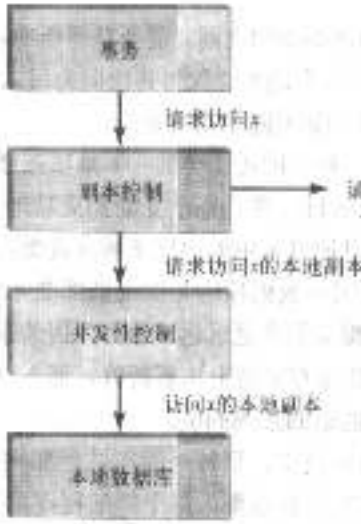


图26-5 在复制数据库中副本控制与并发性控制之间的关系

商用DBMS中的副本控制试图维持某种形式的**相互一致性**（mutual consistency）。**强相互一致性**（strong mutual consistency）要求每个数据项的任何一个已提交副本的值始终要与其他已提交副本完全相同，但是，考虑到性能，这个目标很难达到，因此，绝大多数的副本控制采用更诱惑的目标——**弱相互一致性**（weak mutual consistency），即虽然在任意的特定时间里，每个数据项的所有已提交副本可能会具有不同的数值，但最终它们将会具有相同的值。

我们可以使用很多算法来达成这个目标。

最简单的副本控制系统称为**单读/全写系统**（read-one/write-all system）。当某一事务请求读取某个数据项时，副本控制可能返回任意一个副本的值，不妨假设是最近的副本。利用一个完全复制的系统，没有更新复制数据项的事务不必进行任何远程访问，因此能够快速地响应用户。然而，当一个事务请求更新某个数据项，副本控制就必须执行某种最终可以更新此数据项的所有副本的算法。这是一种相当困难的情况，不同的算法具有不同的特性。一般来说，如果读取操作远比更新操作频繁的话，那么单读/全写系统将会改进无复制系统（其中，读取可能需要访问远程的数据项）的性能。

根据副本控制用来维持相互一致性的算法，单读/全写系统能够被刻画为同步更新或异步更新。我们将在下一节讨论同步更新系统，在26.7.2节讨论异步更新系统。

26.7.1　同步更新复制系统

在**同步更新系统**（synchronous-update system）中，当一个事务更新某个数据项时，该副本控制将在该事务提交之前锁定所有的副本并予以更新。结果就维护了强形式的相互一致性。此外，因为副本是立即更新的，所以同步复制也称为**热切**（eager）复制。当事务在读取某个数据项时，仅对其中一个副本加锁。

假设使用两阶段加锁策略，在该事务提交之前，所有已访问的副本都已加上相应的锁。因此，事务可以串行执行，而数据库的一致性也得以保持。

可以采取以下两种形式加锁:

- *悲观型*　在事务处理超出访问所请求的语句之前，需要获得所有必要的锁。
- *乐观型*　当执行该语句时，只锁上一个副本，仅当其访问为写入操作时才实施更新。其余的副本则仅在该事务提交前，才加锁并随后再更新。

无论利用哪种形式，都有可能产生一种新的死锁类型——**单项死锁**（one-item deadlock）。如果同一数据项的两个更新程序并发地运行，并且每个更新都成功地锁定该数据项的副本的某个子集，那么就会发生上述问题。可以通过常用的协议来解决这类死锁。

当一个事务提交时，我们必须确保同一数据项的每次更新都是永久的。仅在该事务启动的站点提交此事务，并向副本站点发送提交消息是远远不够的。因为副本站点A可能在收到此消息之前崩溃。如果在恢复期间，在A处没有安装上该更新值，那么对于在A处履行的另一个事务所提出的读取该数据项的请求就不能返回正确的值。

两阶段提交协议可用来确保如果事务提交，则每个副本站点都将提交其新的值。伴随程序就是该事务访问过的副本站点[⊖]。虽然没有必要为每个伴随程序单独评估其各个子事务的结果，但此准备状态却是必需的，以便确保在该事务真正提交之前，所有的更新都是持久的。

但是，热切复制会要求该事务获得额外的锁，这就增加了产生死锁的可能性。此外，由于为处理远程副本加锁请求所需的时间，以及直到所有的副本站点都能够确保持久性时该事务才能完成的这一事实，其响应时间极大地延长了。这些因素影响了性能，因此，同步复制的应用性相当有限。

⊖ 仅完成读取操作的副本站点只要一收到此准备消息就立即放弃其读取锁。参见练习26.14。

合议庭协议

虽然同步更新单读/全写复制可以为读取者增加可利性，但它对更新者是没有帮助的。因为一个（同步型）更新必须访问所有的副本，所以如果有任何站点崩溃，则其更新就无法完成。现在我们介绍一种不必访问所有副本的同步复制，使得每个数据项仍是可用的，即使某些副本是无法访问的。为了达到这个目标，我们不再坚持要维护相互一致性（即使是弱相互一致性）。因为副本不再是相同的，所以一个数据项的状态必须按其副本来构筑，而副本控制就是负责这个任务的。我们的目标与单读/全写复制一样：所有通过使用副本控制而产生的调度，都应当与每个数据项只有一份拷贝时的可串行化调度相同。

合议庭协议（Quorum Consensus Protocol）[Gifford 1979]的基本思想是，当一个事务请求读取（或写入）某个复制的数据项时，其并发控制在批准该请求之前，首先对那些称为**读取法官**（read quorum）或**写入法官**（write quorum）的子集加锁。如果某读取法官的规模是p，而写入法官的规模是q，那么我们要求$p+q>n$且$q>n/2$，其中n是副本的总数。因此，我们确保在任何写入法官和读取法官之间、某特定的数据项的任何两个写入法官之间存在某个非空交集。其结果是，每当并发事务执行某个复制的数据项有冲突的操作时，在至少有一个副本的站点上，是不会批准加锁请求的，因此，其中的某一个操作将被迫等待。值得注意的是，单读/全写系统可以看成是一种合议庭协议，其中读取法官的副本数为1，写入法官的副本数为n。

合议庭协议使得我们可以权衡可用性和在某个数据项上的操作代价。p的值越小，读取的数据项的可用性就越高，读取代价也越低。同样地，q的值越小，写入的数据项的可用性就越强，写入代价也越低。然而，读写的可用性是彼此相关的。读取的可用性与效率越高，那么写入的可利用性与效率就越低。

当某个写入要求得到批准（因此该事务提交）时，仅有写入法官中的数据项得以更新。因此，没有保持相互一致性，在不同站点的副本可能有不同的值。我们假设，每当事务提交时，就为它分配一个唯一的时间戳，而每个数据项的所有副本都包含着最后写入此副本的（已提交）事务的时间戳。因为每个读取法官都与写入法官相交，所以读取法官与最近写入的写入法官相交。因此，每个读取法官中至少有一个数据项拥有最后更新此数据项的事务的时间戳，从而拥有当前值。此时间戳是法官中的最大时间戳。副本控制向读取请求返回相应的副本的值。

我们现在总结一下合议庭协议。我们假设某个数据项R在该系统中存储为一系列的副本，每个副本包含有一个值及一个时间戳。我们还假设采用某个直接更新的悲观型并发性控制以及某个严格的两阶段提交协议。最后，我们假设每个站点维护一个本地时钟，而且该时钟至少被同步到事务的时间戳与其提交顺序是相一致的。

合议庭副本控制协议

1）当在A站点中执行的一个事务发出读取/写入某个特定数据项的请求时，A站点的副本控制会把此项请求发送给包含其副本的站点的读取/写入法官。如果该副本站点的并发控制批准给该副本加上相应的锁的话，那么该请求操作得到实施，并将应答返回给A站点的副本控制。若是读取请求，那么此应答将包含该副本的值及其时间戳。

2）一旦A的副本控制收到某个站点法官的应答，该事务便继续推进。如果是读取请求，

副本控制就把拥有最大时间戳的副本值返回给此事务。

3) 事务通过两阶段提交协议实施提交，其中伴随程序是持有读/写锁的所有站点[⊖]。协调程序为那些采用本地时钟的事务获取时间戳，并将它和准备消息一起发送。如果该事务提交，那么有写入动作的每个伴随程序在更新其副本时，都将采用在其解锁以前的时间戳值。

只要控制程序可以为所有的操作集中必要的法官，那么即便出现故障，该协议仍然可以进行。当某个站点故障，并随后重启动时，它的某些副本可能会持有非常早的时间戳。然而，该站点无须采取任何特别的恢复动作，因为在没有被后来的事务覆盖之前，这些副本是不会被任何事务使用的，而一旦覆盖，这些值就是当前值了。

26.7.2 异步更新复制系统

在**异步更新系统**（asynchronous-update system）中，当事务更新某个数据项时，副本控制在该事务提交之前仅仅更新部分而不是全部的副本。最常见的是仅仅更新一个副本。其余的副本将在该事务提交后才更新。因此，只能保持弱相互一致性。这些更新可以由提交操作而触发，或是以固定的时间间隔周期性地执行的。由于副本并没有立即更新，所以异步复制又称为**惰性**（lazy）复制。更新并不是作为事务本身的一部分来完成的，所以*系统整体上可能不是可串行化的，并且事务可能会看到某种不一致的状态*。

例如，图26-6显示的是一个调度，其中事务T_1更新x_A（x在A站点的拷贝）和y_B（y在B站点的拷贝）。在T_1提交后，事务T_2读取y_B（因此得到了y的新值）和站点C中的x的副本x_C（尚未来得及更新）。因此，T_2将（可能）看到该数据库的不一致视图。然后，将执行某个副本更新事务T_{ru}来更新x、y以及x_C的所有副本。

T_1:	$w(x_A)$	$w(y_B)$	提交			
T_2:			$r(x_C)$	$r(y_B)$	提交	
T_3:					$w(x_C)$	提交

图26-6　显示采用惰性复制可能会出现不一致的调度的例子。T_1更新站点A和B的x和y。T_{ru}在T_1提交后传播该更新。由于传播是异步的，所以T_2看到的y是新值，但x却是旧值

在复制场合，**捕获**（capture）是指副本控制确定某个应用程序的副本的更新已经出现，并需要将此更新传播到其他的副本的过程。捕获可以采取两种形式：监视日志并密切注意对于复制的数据项的更新，供以后传播之用；在数据库中设置触发器，以记录变更。**应用**（apply）是指通知副本站点必须实施该更新，以确保它们的副本是当前最新的过程。

不同形式的异步复制适合不同的应用程序。在某些情况下，重点需要放在使副本尽可能紧密的同步。虽然并不保证可串行化，但我们的目标是尽量减少从一个副本更新到将此更新应用到其他副本之间的时间间隔。维护账户或客户服务记录的分布式应用程序就属于这个范畴，它们有着各种名称：**成组**（group）**复制**、**对等**（peer-to-peer）**复制**、**多主**（multi-master）**复制**。在其他的情况下，紧密的同步性并不是最重要的。例如，某公司在该领域拥有巨大的销售能力，但它只需周期性地登录主站点，并下载最新的数据视图，就能满足其需要了。在

⊖ 如前所述，仅仅实施读取操作的副本站点只要一收到准备消息，就能够立即放弃其读取锁。

这种情况下经常使用的复制形式是**主拷贝复制**（primary copy replication）。当大批量的数据必须更新时，往往应用第三种复制形式，即所谓**过程式**（procedural）复制。

1. 主拷贝复制

数据项的某个特定的副本可以指定为**主拷贝**（primary copy）[Stonebraker 1979]，其余的副本都是**次拷贝**（secondary copy）。次拷贝通过**订阅**（subscribing）主拷贝所做的变更而产生。虽然事务可以读取任意的拷贝，但它只能更新主拷贝。如果在A站点的事务T希望更新数据项x，一种办法是在x的主拷贝x_p上设置一个排他锁，然后再更新这个主拷贝。哪怕在A站点有次拷贝x_A，也不会去更新这个次拷贝。其结果是，若有两个事务要更新x，x_p上的锁也只能等到第一个事务提交后才能授予第二个事务。因此，事务的写入操作是串行化的，但读取操作却不是。另一种办法是，应用程序只需要把更新传递给主拷贝，以便稍后再处理。

当T提交后，T的更新将异步地、非串行化地传播给次拷贝（包括x_A）。因为T和应用步骤没有设立单一的隔离级单位，所以副本是以非串行化方式更新的。对于读取来说，使用任意的副本都可以满足，因此，并发性事务有可能看到该数据库的某个不一致的视图（如图26-6所示），结果造成功能的不正确。

利用主拷贝复制，所有的更新都通过主拷贝汇集起来，此拷贝所在的站点然后执行副本更新事务，以实现其应用步骤。某个更新事务可以更新所有的副本（如图26-6所示），或者每个副本都需要使用单独的更新事务。如果每个副本的更新事务都是按照主拷贝的更新顺序可串行化地执行，那么该副本控制系统就能够确保：每个次拷贝都可以按照主拷贝的顺序看到该更新。这就确保了弱相互一致性。

在主拷贝复制的某些方案中，如果站点A的T请求更新x，且x_A不是主拷贝，那么它将对x的主拷贝和x_A两者都加锁，并把对它们的更新作为该事务的一部分。其余的次拷贝将如前所述，在T提交以后再更新。采用这种方法，在A站点的用户就可以直接执行随后的、需要读取x_A的事务，而不必等待从主拷贝处向A传播回该更新。

采用主拷贝复制实现应用步骤的另一个办法是，副本控制系统可以周期性地向所有次拷贝广播（broadcast）主拷贝的当前值，而不是在事务完成后再传播其主拷贝的更新。这种值广播应当在事务上是一致的，它包含着自从上次广播以来由已提交事务所完成的全部更新。

在有些实现中，每个次站点可以声明主数据项的某个视图，并且仅仅传递此视图。若次站点与主站点之间是利用低带宽（例如电话）来通信的，那么每个站点仅仅复制那些相关的数据就显得很重要，此时利用这个方法就很重要。

还有一些实现应用步骤的办法，其更新并不是自动地从主站点传播的，而是由副本站点明确地请求刷新他们的视图。这被称作**上拉策略**（pull strategy），因为次站点是从主站点拉取这些数据。相反地，在我们以前所描述的**下推策略**（push strategy）算法中，主站点是将数据向下推给次站点。下推策略缩小了副本值可能不一致的时间间隔。无论使用哪种策略，一个数据项的所有副本最终都会包含同样的值，所以弱相互一致性得到支持。

当次站点是该地区的某个大型销售团队使用的移动（可能是手提的）电脑时，利用下推策略可能是很合适的。每个销售人员可以在连接到网络时更新他的数据项的副本。因为带宽较窄，他就可以定义一个仅仅包含与其销售地区相关的信息的视图。在这样的应用程序中，副本可以稍为陈旧些（例如滞后几分钟或几小时）。读取访问占大多数。只有在签署新的合约

时，该新信息首先发送到主站点，而后再传播到（或是被上拉到）所有的次站点，这时才可能会出现写入访问。倘若更新出现时次站点是断开的，那么此更新就必须保持到建立起某个新的连接为止。

2. 成组复制

对于任何复制品（不妨假设是最近的副本）都可以实施更新。（因此，如果数据库是完全复制的，那么仅利用本地站点的数据就能够完成读/写事务或只读事务。）某个事务T所作的更新，将在以后才传播给其他的副本，由于传播是异步的，所以全局可串行化无法得到保证。

和使用主拷贝复制一样，成组复制可能会导致非串行化（从而引起不正确的）调度。例如，有可能出现与图26-6相似的调度。此外，如果没有进一步的改进，可能连弱相互一致性都无法保持。图26-7说明了在站点A和D执行的两个并发性事务T_1和T_2分别更新数据项x的不同的副本的情况。如图所示，传播这些更新的消息的到达顺序在不同的副本站点是不一样的。由于副本的最终值是其最后一次写入的值，所以若副本是按不同的顺序更新的，那么就无法保持相互一致性。按复制的术语来说，就是出现了某个**冲突**（conflict），因此副本控制系统必须采用**冲突化解策略**（conflict resolution strategy）来保证**收敛性**(convergence）和相互一致性。

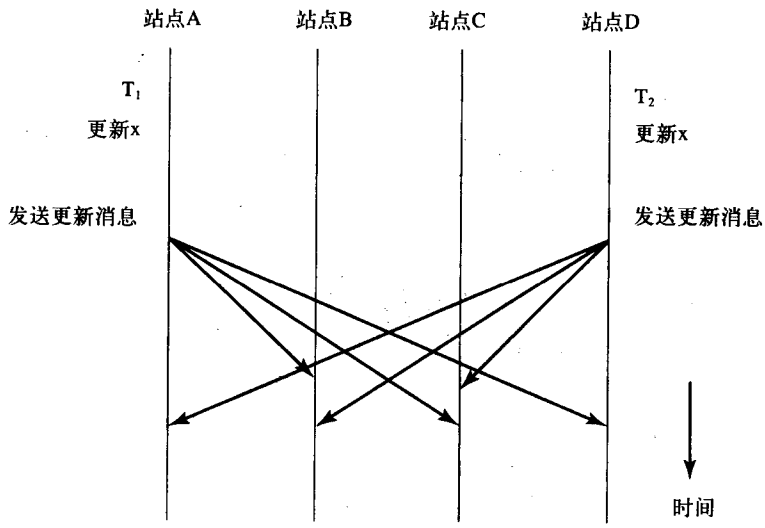


图26-7　采用成组复制，更新可能以不同的顺序到达副本，从而使相互一致性遭到破坏

一种保证弱相互一致性的算法是给每个更新和每个副本都加上唯一的时间戳。所谓更新的时间戳就是将更新提交给系统的时刻，复制品的时间戳是其上应用的最近一次更新的时间戳。如果副本站点只是将那些时间戳晚于副本时间戳的新到达的更新抛弃，那么就可以维持弱相互一致性。这是我们在23.8.1节中讨论的托马斯写入规则(Thomas Write Rule)[Thomas 1979]的一个例子。采用这种算法，每个副本的值最终将收敛到时间戳最大的(最新的)那些更新的值。虽然此规则保证了弱相互一致性，但有可能会丢失更新，因为两个事务读取并更新了不同的副本，然而只留下了其中一个的结果。因此，这种算法可能不适合于某些应用程序。

遗憾的是，在这种情况下，不存在一个可以为所有的应用程序正确地融合不同事务效果的

算法。在某些应用程序中，明显地存在合适的冲突化解策略。例如，如果一个目录已复制好，而且并发事务向不同的副本分别追加了不同的条目，那么最终目录中的所有副本都包含这些条目。这样的话，我们就可以为某一特定应用程序设计一种冲突化解策略。在一般情况下，每当副本控制系统侦测到冲突时，它就会通知用户并允许用户来化解冲突。因为没有任何冲突化解策略可以确保始终正确，所以某些商用系统就提供了一些可供选择的即席策略，包括“最早更新赢”、“最新更新赢”、“最高优先级站点赢”和“用户提供的冲突化解过程”等。

3. 过程复制

当更新需要以面向批处理方式应用到许多数据项时（如在每个银行账户中加入利息，以及在多个站点中复制银行记录），使用这种复制形式是很有用处的。如果每个更新是在网络中单独传递的话，通信成本会较高。一种可供选择的方法是在每个次站点上都复制一个**存储过程**，一旦该数据需要更新时，便在所有的副本上调用此过程。

4. 小结

同步更新系统与异步更新系统之间的权衡就是正确性与性能的一种对全。许多商用DBMS提供了这两种类型的复制。设计者应当清楚，异步更新有可能会产生不正确的、非串行化调度，并导致不一致的数据库。

26.8　现实世界里的分布式事务

虽然分布式事务处理系统的设计理论看上去较复杂，但最终运行结果却出乎意料的简单且很实用。如果每个站点的并发性控制是严格的两阶段悲观型加锁控制或乐观型控制，并使用两阶段提交协议来保证提交时伴随程序同步，那么分布式事务将是可串行化的。全局死锁是由于悲观型控制所强制的等待造成的，但可以通过时间戳、等待图或者超时来解决。

这种简化的结果是，拥有着不同厂商生产的DBMS的互连站点也可以实现分布式事务处理系统，但必须满足如下的条件:

1) 所有的DBMS实现两个指定的并发控制之一。

2) 每个站点都要参与此两阶段提交协议。

这些想法大大地简化了系统设计。

原子性和隔离性是事务处理系统的特点，这种事务处理系统确保只要应用程序的事务是一致的，那么此应用程序就能正确地执行。我们已经看到，许多分布式事务处理系统为了提高性能，常常不是完全地支持原子性和隔离性。某些站点的事务可以不遵循SERIALIZABLE隔离级别运行，在提交分布式事务时不运用两阶段提交协议，采用异步更新技术支持复制过程等等，这些情况都是有可能的。这些让步是否会产生错误的操作，在很大程度上取决于其应用程序的语义。正因为这个原因，我们在构建一个特定的应用程序的事务处理系统时，必须认真地考虑这些问题。

26.9　参考书目

分布式事务的全面讨论可以在[Gray and Reuter 1993]里找到。在这一个方面，[Ceri and Pelagatti 1984]的理论性更强一些。

两阶段提交协议是由[Gray 1978，Lampson and Sturgis 1979]引入的。在[Mohan et al. 1986]里讨论了两阶段提交协议的预想的异常中止特性。一个称为三阶段提交的功能更强大的提交协议是在[Skeen 1981]中介绍的。采用时间戳技术以避免死锁的wound-wait和kill-wait系统是在[Rosenkrantz et al. 1978]介绍的。两阶段提交协议连同两阶段加锁本地并发性控制以确保全局可串行化的证明，是在[Weihl 1984]中介绍的。合议庭协议是在[Gifford 1979]中介绍的。主拷贝复制是在[Stonebraker 1979]中引进的，而托马斯写入规则则来自[Thomas 1979]。

26.10 练习

26.1 若在两阶段提交协议内，处于以下状态时有一个伴随程序或协调程序崩溃，请描述其恢复过程该如何进行？

a. 在其协调程序发送准备消息之前

b. 在某一伴随程序已投票，但其协调程序尚未决定提交还是异常中止之前

c. 在其协调程序已决定提交，但该伴随程序尚未接收到提交消息之前

d. 在该伴随程序已提交，但其协调程序尚未把结束记录写入日志之前

26.2 请说明在两阶段提交协议里，为什么一个伴随程序确实不该强制将异常中止记录写入日志？

26.3 请说明模糊转储恢复过程该如何扩展，以便在两阶段提交协议里处理日志中可能存在准备记录的情况？

26.4 请描述当若干个数据库管理程序正在使用乐观型并发控制时的两阶段提交协议。

26.5 假定没有任何故障，请描述如何将两阶段提交协议扩展到分布式事务是任意的子事务树的情况？

26.6 请描述在上述例子的协议里如何实现预想的异常中止特性？

26.7 请描述本书怎样扩展本书给出的两阶段提交协议使之可以处理采用时间戳顺序的并发控制的伴随程序站点？

26.8 请给出正在两个不同的站点执行的分布式事务的调度，使得每个站点的提交顺序不同，但全局调度却是可串行化的。

26.9 假设某个分布式系统的每个站点的并发性控制是独立地使用严格的两阶段加锁并发控制、某种乐观型并发控制或者某个时间戳顺序控制的，以便使冲突事务按提交顺序串行化，并采用某个两阶段提交协议。这样的分布式系统是否是全局可串行化的吗？

26.10 扩展的两阶段提交协议的第1阶段同我们已经描述过的两阶段提交协议的第1阶段相同。回想本书描述的两阶段提交协议，其提交记录是在第1阶段完成之后才写入日志的。而在新的协议里，第2阶段如下：

第2阶段　若协调程序在第1阶段至少收到过一条异常中止投票，它就决定异常中止，并向投票就绪的所有伴随程序发送一条异常中止消息。若所有的投票都是就绪，那么它就向所有的伴随程序发送一条**提交**消息。

如果某个伴随程序接收到一条**异常中止**消息，它便将该子事务回退。如果某伴随程序接收到一条**提交**消息，它便强制将提交记录写入其日志，释放全部锁，并向协调程序发送**完毕**消息，然后终止。

协调程序一直等待，直到它从某个伴随程序处接收到第一条**完毕**消息为止。此后，它强制地向其日志写入一条提交记录。

协调程序等到接收到所有伴随程序的**完毕**消息后，就向其日志写入一条结束记录，然后终止。

请为这个协议描述其超时过程和重启动过程。请证明，这个协议仅在两种情况下会阻塞：协调程

序和至少一个伴随程序出现（或部分出现）崩溃时以及所有的伴随程序都崩溃时。

26.11 请为线性提交协议设计一个登录、超时和重启动过程。切勿假定存在一个单独的协调程序模块。假设伴随程序之间的所有通信都是沿着一条链执行的。

26.12 考虑一个分布式事务处理系统，其中每个站点都采用串行验证的乐观型并发控制，以及某个两阶段提交协议。请证明，在这些条件下，仍然有可能发生死锁。

26.13 如果所有的站点都使用乐观型并发控制，并且采用某个两阶段提交协议，请证明这时分布式事务是全局可串行化的。

26.14 如果一个分布式事务的伴随程序已完成的仅仅是读取操作，那么两阶段提交协议是可以简化的。当该伴随程序接收到准备消息时，它便放弃其锁，并且终止参与协议。请解释为什么这种简化后的协议能正常工作？

26.15 考虑下述原子提交协议，它试图消除在两阶段提交协议里出现的阻塞。每个伴随程序都包含所有伴随程序的地址，协调程序向每个伴随程序发送一个准备消息。每个伴随程序又直接向其他的伴随程序发送它的投票。当一个伴随程序接收到其所有伴随程序的投票时，它便可以按照通常的做法来决定是提交还是异常中止。

a. 假定没有任何故障，请比较这个协议里与两阶段提交协议里所发送的消息的数量。假定发送所有消息所需的时间长度都相同，哪个协议会运行得快些？为什么？

b. 当出现故障时，该协议是否仍然能杜绝阻塞？

26.16 练习23.32中的kill-wait并发性控制是建立在加锁的基础之上的。当将它用到某个分布式系统里时，通常将它称为wound-wait协议。我们假定分布式事务在伴随程序之间是采用RPC（远程过程调用）进行通信的，以便当两阶段提交协议启动时，所有的伴随程序都已完成。用wound原语取代kill原语。

如果某个站点的事务T_1的伴随程序发出与那个站点的某个主动性事务T_2的伴随程序的操作相冲突的请求，那么

if $TS(T_1) < TS(T_2)$ then 使T_2 wound **else** 让 T_1 等待

其中wound T_2意味着T_2被异常中止（正如在kill-wait协议里一样），除非T_2已经进入两阶段提交协议，那时T_1将等待T_2完成协议。

请说明，这个协议为什么可以防止事务之间的全局死锁？

26.17 假定嵌套的事务模型得到扩展，使得子事务分布于网络的不同站点上。执行某个分布式嵌套事务时，在哪个时刻会使某个伴随程序进入准备状态？请说明理由。

26.18 在本书中我们介绍过，如果分布式数据库系统里每个站点都采用严格的两阶段加锁并发控制，并且系统使用两阶段提交协议，那么事务是全局型可串行化的。倘若并发控制不是严格的（然而却是两阶段的），这个结论是否还成立？

26.19 请说明如何利用触发器来实现同步更新复制？

26.20 请设计一个合议庭协议，其中没有时间戳字段，但每个数据项都有一个版本号字段，它每逢数据项写入时进行更新。

26.21 考虑一个合议庭协议，其中每个数据项保存5份拷贝，每次读取和写入法官的数目均为3个。请给出一个满足以下条件的调度：

3个不同的事务欲对数据项执行写操作。然后有2个副本站点出现故障，于是仅剩下3份拷贝——它们全都包含着不同的值。

请解释，为什么该协议仍然能正确执行？

26.22 考虑一个合议庭协议，其中每个数据项保存n份拷贝，每次读取和写入法官的数目分别为p个和q个。

a. 出现故障但仍然能使协议正确工作的副本站点的数量最大是多少？

b. 能够使得$p = q$的p和q的最小值是多少？

c. 请选择p和q，使得出现故障但仍然能使协议正确工作的副本站点数量为最大。对于这样的选择，最多允许有多少个站点故障？

26.23 请说明在合议庭复制算法里，为什么所有站点的时钟都必须同步化？

26.24 请描述复制数据项的一个应用程序，其中可以不需要可串行性。

26.25 以什么方式使用你存放在家里的支票本，才能使你的支票账户如同一个异步更新复制系统？

26.26 请给出由某个主拷贝异步更新复制系统产生出非串行化调度的一个例子。

26.27 有人建议采用下述主拷贝异步更新复制协议的方案来完全复制系统。

a. 在站点s_i处执行的事务在提交以前，仅仅更新站点A处的副本。

b. 在事务提交之后，再启动第2个事务来更新站点A更新过的所有数据项的主拷贝。

c. 当步骤b中的事务完成之后，每个主站点就把在此站点上做出的全部更新传播到其所有的次站点（包括站点A上的拷贝）。而由若干事务对主拷贝所做出的更新，则按照原来主拷贝更新的顺序再传播到其次站点。

请说明，为什么在步骤b中，所有的主站点必须在某个单独的事务内更新，而在步骤c中更新才被传播到站点s_i。

26.28 请说明如何才能应用触发器来实现主拷贝复制？

26.29 请为分布式保存点设计一个登录协议。

第27章 安全性与因特网商务

27.1 认证、授权与加密

安全性是处理具有权限的信息或者与生命和财产息息相关的支持系统的应用程序设计中的一个主要的问题。在绝大多数应用程序中，每个事务的参与者必须能极为肯定地确认彼此的身份，并且希望确认没有任何第三方在观察或者修改正在交换的信息。对于通过**因特网**执行的事务来说，安全性问题尤其重要。因为冒名顶替者伪装成特定的客户或者服务器，以及窃听者偷听事务参与者之间的交换内容都相对比较容易。

认证（authentication）就是指确认参与者的身份的过程。当你在一台ATM（Automatic Teller Machine，自动柜员机）上实施某项事务的时候，系统通过你的ATM卡信息和**个人识别号**（PIN）来识别你的身份。另外，你可能希望确认，你是在和一个真实的ATM机，而不是某个设计用来窃取你的PIN的“特洛伊木马”（Trojan horse）对话。同样地，当你考虑执行某个利用Macy（一种零星付款）装置的因特网事务时，将不得不向服务器提供信用卡号，这时你希望确认，你确实是在和该服务器而不是躲在寝室里伪装成Macy事务的三个学生对话。

授权（authorization）就是指确定允许特定客户访问由某个服务器控制的特定资源的模式的过程。假定这个客户事先已经过认证。授权可以是对某个人授权（允许你从自己银行账户里取款）或者是在小组范围内授权（允许所有的出纳员写入已认证的支票），并且根据个人或者小组能够执行的功能进行授权。例如，“允许John从某个指定的账户中提款。”

加密（encryption）常用来防止未经授权用户访问站点间传输的或存储在特定站点的信息。可以用许多复杂算法将信息转化成某种比特流，但这些比特流只有少数选定的用户才可能理解。

鉴于加密在认证和授权中起着很重要的作用，所以我们先浏览一下其大体结构。然后再讨论认证和授权，最后则讨论这三个概念如何在因特网商务事务中应用。

27.2 加密

图27-1为我们展示了一般的加密系统的模型。需要传输的信息一般是称为**明文**（plaintext）的一串字符。由某个装置（或程序）将它加密成**密文**（ciphertext），而实际上传输的正是密文。在接收端，再通过另外的装置对密文进行解密，把它变回原来的明文。

加密系统的目标是防止**入侵者**（intruder）接触信息。一般假定入侵采取两种形式。第一种是**被动式**（passive），此时入侵者只能够复制传输中的信息。第二种是**主动式**（active），此时入侵者还可能修改传输中的信息，重新发送（被它复制过的）原先发送的消息，甚至发送新的消息。然而，目前还没有任何实际的加密系统能够绝对杜绝任何具有无限计算机资源（可用来分析密文）的入侵者。因此，加密的目标只能是使入侵变得很困难，以至于每个入侵

者几乎都不可能成功。

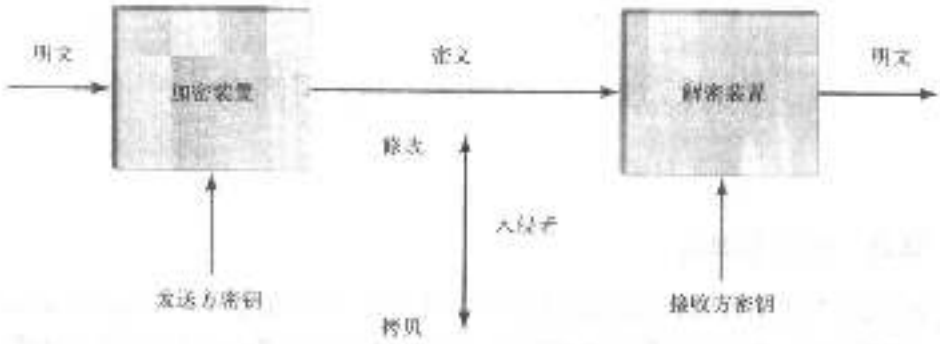


图27-1　一个加密系统的模型

目前存在很多加密算法，而且一般总是假定入侵者知道被攻击系统所采用的特定算法。然而，每个算法都可参数化为（控制加密和解密过程的）一个或多个密钥（key）。要想将密文恢复为明文，若无解密密钥，则是极端困难的。因此，技术的威力取决于确保解密密钥的机密性[⊖]。

我们采用下述表示法

$$\text{密文} = K_{sender}[\text{明文}]$$

来表示密文是由发送者利用密钥K_{sender}加密明文而得到的。包括加密和解密在内的整个加密系统可以用这种表示法表达成：

$$\text{明文} = K_{receiver}[K_{sender}[\text{明文}]]$$

它表示，如果首先用K_{sender}对明文加密，然后再用$K_{receiver}$解密，其结果就是原来的明文。

1. 对称加密法

所谓对称加密法（symmetric cryptography），就是加密方和解密方采用相同的密钥（$K_{sender} = K_{receiver}$）。此密钥仅为正在通信的两个过程所知（因为它可以用来对密文解密）。

在对称加密法中有一些常用的技术。所谓**块密码**（block cipher），指首先将明文划分成固定大小的若干块，然后将它们一对一地映射成密文块。该特定的映射是此加密算法及其密钥的一个函数。密文块序列就是加密后的消息。由于此映射是一对一的，所以其明文能够得到恢复。

替换密码（substitution cipher）是块密码的一种类型，目前已提出过很多种替换密码，用来映射明文块。例如，某个**单表**（monoalphabetic）（有时称为简单）**替换密码**，其块长度为一个字符，而构建密文则采用某个从字符集合到其本身的一对一映射。因此，倘若将a映射成c，那么在明文中凡是出现a的地方，在其密文的相应位置中都替换成c。

多表（polyalphabetic）**替换密码**是由多个单表替换密码组成的。例如，可以有十个不同的单表密码。第一个字符采用第一个密码加密，第二个字符则用第二个密码加密，依此类推，直到第十个字符，在此之后，从第11个字符开始又再依次采用第一个密码加密等等。

⊖　一个真正不可破解的加密系统是一次插入（one-time pad）[Stallings 1999,Schneier 1995]。它采用某个随机生成的，与该消息一样长的密钥，而且也仅用于一条消息，用后就被丢弃。

多重（polygram）**替换密码**中块的长度大于1。例如，若块的长度为3，则块“cde”可以加密为“zyy”，而“dce”可以加密为“dtr”。

换位（transposition）**密码**也是一种块密码，但是与替换密码不同的是，每一个明文块中字符的顺序在产生密文块时进行更改，即对于每个明文块中的字符重新排序，排序方法通过密钥来刻画。例如，如果将块“cde”加密为“dec”，那么“dce”就会加密为“ced”。

将明文块映射成密文块所用的密码是频度分析攻击的攻击对象。假设某入侵者知道块长度，并能够测定每个明文块在正常（非加密的）通信中的使用频度。那么此入侵者就可以将这个频度同某个加密后的比特流中密文块的频度进行比较。通过将明文和密文中具有相近频度的字符相匹配，入侵者就可以大大减少为了决定哪个明文块映射到某个特定的密文块所必须测试的可能组合的数目。入侵者能够监控的加密比特流越长，对密文块的频度估计就越准确，就会使计算量大大降低。频度分析攻击使得许多短长度块的替换密码和换位密码失去用武之地，因为对于小型明文来说，可以获得较准确的频度文档。

ANSI的**数据加密标准**（Data Encryption Standard，DES）是一种对称加密法技术，它采用一连串的阶段来对每一明文块进行加密，而其中每一个阶段则都是对前一个阶段的结果进行加密。所有的阶段都采用长度相同的块，而且每一个阶段都是采用替换密码或是换位密码。将这两种密码技术组合起来的结果有时候称为**产品密码**（product cipher）。目前广泛应用的DES（例如在银行和金融服务业），它采用的就是64比特的块长度和一个56比特的密钥[⊖]。

比特流密码（bit stream cipher）则是不以块为基础的加密技术的一个例子。其密文流是对明文流与某个伪随机数序列进行异成操作的结果，这个随机数序列是以密钥为初始种子的随机数发生器产生的。通过采用同一发生器的密钥，在接收端就可产生同样的伪随机数序列来解读此密文。

2. 非对称加密法

与对称加密法不同，**非对称加密法**（asymmetric cryptography）为每一个用户都配有一个加密密钥和一个解密密钥。加密密钥是不保密的。用户可以将它发布给任何想要给他发送消息的人。所以加密密钥是一个公开的密钥（公钥），潜在的入侵者也可以知道该密钥。然而，解密密钥是不公开的，只有用户知道解密密钥。如果M是一个（明文）消息，而K_C^{pub}和K_C^{priv}分别是客户C的公钥和私钥，那么非对称加密法就可以用下述关系来描述：

$$M = K_C^{priv}\left[K_C^{pub}[M]\right]$$

因为利用K_C^{pub}加密的某个消息，只要通过K_C^{priv}解密就能导出其原始的明文。换句话说，发送者是采用接收者的公钥来加密消息的，而接收者则利用其私钥对它进行解密。

一般情况下，如果消息必须在两个进程之间双向传递，那么每个进程都采用对方的加密密钥来加密消息。由于加密密钥是可以公开的，所以非对称加密法又称为**公钥加密法**（public key cryptography）。在对称加密法中，密钥仅为彼此通信着的进程所知，故而称为**私钥加密法**（secret key cryptography）。公钥加密的概念是由[Diffie and Hellman 1976]提出的，但几乎所有的公钥加密系统都是建立在RSA算法[Rivest et al. 1978]的基础上的。在[Schneier 1995]中给出了对于RSA算法的数学原理以及许多其他加密算法与协议的详细的描述。

⊖ 加密专家已经批评该密钥太短了。

我们简要地描述一下RSA算法。为了设计一个加密/解密密钥组，首先选择两个很大的（随机）素数p和q，另外再选一个整数d，与$(p-1)*(q-1)$互为素数。最终，计算整数e使得它满足下式：

$$e * d \equiv 1 \pmod{(p-1)*(q-1)}$$

加密密钥是(e, N)，而解密密钥是(d, N)，其中$N = p * q$，因而也称为**模**（modulus）。

可以将需要加密的消息分割成若干块，使得每个块M都可以看成是0到$(N-1)$之间的某个整数。为了将M加密成为密文块C，我们实施如下运算：

$$C = M^e \ (mod\ N)$$

为了解密C，我们实施

$$M = C^d \ (mod\ N)$$

该协议是可以正确地运行的，因为

$$M = (M^e \ (mod\ N))^{\,d} \ (mod\ N) \tag{27.1}$$

这可以通过数论的一些基本概念进行证明。

尽管d和e在数学上是相关的，然而通过因子分解N（一个大整数）来得到p和q（它们可用于从e计算出d），那是极其困难的。因此，只有接收者才能够解读发送给他的消息。

公钥加密法是一种功能非常强大的技术，但是它要比对称加密法的计算强度更高（若计算某个大数的指数幂，尽管不像因子分解一个大数那样难，后者则是破译加密所必须的，但当幂为很大的数时，计算起来也是很费力的）。正因为这个原因，公钥加密一般仅用来加密作为某个协议的一部分而交换的小块的消息，而不是用来加密大块数据，大块数据一般采用对称技术加密。

27.3 数字签名

非对称加密法的一个非常重要的作用就是实现**数字签名**（digital signature），数字签名可以作为某个文档的出处证明，就像日常生活中应用的手写签名一样。和公钥加密法一样，这一概念是在[Diffie and Hellman 1976]中首次采用的，但是和公钥系统一样，绝大多数数字签名系统也是建立在RSA算法[Rivest et al. 1978]或者是特别为签名而开发的其他算法的基础上的。

建立在加密算法基础上的数字签名应用了许多非对称加密法算法的一种特性，即公钥和私钥的角色可以对换。私钥可以用来加密明文，由此而得的密文则可以通过采用相对应的公钥来解密。因此，我们有

$$M = K_C^{pub}\left[K_C^{priv}[M]\right] = K_C^{priv}\left[K_C^{pub}[M]\right] \tag{27.2}$$

与加密不同，这里发送者用私钥来加密该签名，而接收者则用发送者的公钥来恢复签名。

如果公钥算法有(27.2)的特性，那么C（发送者）就可以证明是她通过对M用K_C^{priv}加密而发出的消息M。如果接收者可以通过K_C^{pub}恢复M，那么接收者就知道M只能是C发出的，因为仅有C才知道K_C^{priv}。（其他人也许确实发送过此消息，但是只有C才能生成它。）这一技术假设

接收者知道C的公钥 K_C^{pub}。发布公钥的过程和相关的问题将在27.4和27.7.1节中讨论。

值得注意的是，由于任何人都可以用公钥来解密消息，所以该消息就不是隐蔽的。因此，隐蔽消息不是任何数字签名协议的目的。

这种技术的一个问题是，采用公钥算法来加密和解密整个的消息可能造成计算强度很高而且花费时间的后果。为了减少计算时间，可以计算M的某个函数 f（例如，某种检查和或者某个散列函数），它将产生比M自身小得多的某个结果。$f(M)$有时候称为M的**消息摘要**（message digest）。也假设入侵者（像通信者一样）知道消息摘要函数f。用 K_C^{priv} 对$f(M)$加密，并称其为数字签名，它是随M一起传播的。因此，C向接收者发送了两项内容，即 $K_C^{priv}[f(M)]$ 和M。接收者采用 K_C^{pub} 解密第一项，并将其结果与对第二项应用 f 所得到的结果进行比较。如果两者是相同的，那么接收者能够得出结论：只有C才可能发出M。但是，为了安全起见，要得出这样一种结论，我们还必须要处理一些其他的问题。

不妨考虑某个入侵者正在窃听和复制从C到其服务器所传输的内容。

1) 该入侵者可以利用C发送消息M时附带的签名 $K_C^{priv}[f(M)]$ 来签署另外一个不同的消息M'，以便能够迷惑接收者，使其相信C发送的是M'。若他能够构建出M'使得$f(M) = f(M')$的话，那么入侵者的这一攻击是有可能成功的。为了杜绝这类攻击，我们要求f是一个**单向函数**（one-way function），即f具有以下特点：已知某个结果y，要求构建出自变量x，使得$f(x) = y$在计算上是不可行的。例如，某个单向消息摘要函数可以产生满足以下特性的结果串：

a. 在f值域内的所有值都是等概率的。

b. 若修改该消息的任何比特，那么消息摘要中的每个比特都有50%的概率会改变。

特性b保证了$f(M)$和$f(M')$不会相同，因为M和M'是相关的或者类似的消息。特性a确保不可能很容易地找到满足$f(M) = f(M')$的M'，因为f将很大比例的消息映射到$f(M)$。

对于该入侵者而言，要在这些条件下构建出某个消息M'，使得$f(M')$等于$f(M)$是极其困难的。因此，要找出一个可以依附到消息M'上的签名 $K_C^{priv}[f(M)]$，也是很难的。另外，该入侵者也不可能伪造出可以依附于M'的签名 $K_C^{priv}[f(M')]$，因为他并不知道 K_C^{priv}。

2) 该入侵者也许是企图复制，然后第二次地再发送一个带有此签名的消息[⊖]，这称为**回放攻击**（replay attack）。可以通过让客户构建一个时间戳（采用在第26章中提到的技术），并将它包含在该消息里，来处理回放攻击。数字签名是针对整个消息而计算出来的。假设在网络中所有站点上的时钟是大致同步的，如果接收者保持有一份最近接收到的所有消息的时间戳列表，并且拒绝接收包含在此列表中的时间戳的到达消息（注意，时间戳在全局范围内是唯一的），那么它就可以侦测到该消息是第二次到达的。这里的关键在于，时间戳是不重复的。如果每个消息都有一个唯一的顺序号，那么也可以破解回放攻击问题。

除了确认签名者的身份之外，数字签名还保证了该消息的*完整性*。尽管该消息是透明传输的，并因此可能被某个入侵者读取，但它不可能被入侵者修改，因为修改过的消息的签名是不同的。（如果希望保持私密性，那么签过名的消息可以用另一个密钥加密。）

数字签名还防止了**否认行为**（repudiation）。签名者不能否认构建过此消息，因为数字签名和消息只能用该签名者的私人密钥构建出来。数字签名经常应用于商业的安全协议中。

⊖ 若该消息是C请求将资金转入该入侵者的银行账户中的话，那么有可能利用这种攻击。

27.4 密钥发布与认证

无论是对称加密法还是非对称加密法，在双方能够通信之前，它们必须就所用的加密/解密密钥达成共识。由于完成以上工作涉及在消息中讨论通信时的密钥，因此协议的这一阶段就称为**密钥发布**（key distribution）。在大多数情况下，密钥发布也涉及认证。在双方就密钥达成共识的同时，它们也都确认了对方的身份。

有人也许会认为密钥发布对于非对称算法而言很容易，因为接收者的加密密钥是公开的。也就是说，如果*C*想要给*S*发送某个加密的消息，他只要向*S*发送一个透明的(不加密的)请求，要求得到*S*的公钥。然后*S*就可以把他的密钥（透明地）发送给*C*。但是，事情不是那么简单的。入侵者也许会在中途拦截*S*的消息，并将消息换成他自己的公钥。一旦*C*采用了入侵者的密钥，那么入侵者就能够解读*C*发送给*S*的所有的消息。因此，通过在采用公钥加密时进行认证，可以使密钥发布变得复杂，因为*C*必须能够鉴别密钥发送者的身份。

在对称加密的情况下也存在类似的问题。如果对于每个进程*P*都有一个唯一（对称）的密钥，用于加密和解密消息，那么所有要向*P*发送的进程就都应当知道这个密钥。但是，如此一来，每个发送者就都可以解读由其他人发送给*P*的所有消息了，因为该密钥也能用来解密——这是令人无法接受的。这一问题的常用解决方法是，给各进程之间的每次对话或会话配上称为**会话密钥**（session key）的某个唯一（对称）的密钥。会话密钥必须在会话启动前创建并发布给进程各方。它已成为通信上下文的一部分，并在会话结束时抛弃。再强调一次，采用某个会话密钥的进程都需要弄清持有该密钥拷贝的其他进程的身份。

因此，除了解决密钥发布的问题外，我们还必须考虑认证的问题。直观而言，每当我们说到某个客户时，其实我们同时也在考虑该客户进程的运作所代表的个人，所以，我们也可互换地使用这些概念。

认证的一个目的就是使一台服务器可以明确地辨别发送消息的客户的身份，以便它能够决定是否授权其请求的服务项目。例如，是否允许请求者从Macy的银行账户中提款？认证的另一个目的则是使得每个客户在向某台服务器发送任何重要的消息之前，能够认证该服务器。例如，是在向Macy还是一个Macy假冒者发送信用卡号码？

为了对这些情况建模，我们需要涉及**主角**（principal）的概念。所谓主角，既可以是一个人，也可以是一个进程，认证的目的在于证实主角真是自己所声称的对象。

一般来说，某个人会通过提供一些只有她本人才可能持有的东西来证明她就是她自己所声称的人。其中最简单又最不安全的就是**口令**（password），在这种情况下，她个人所持有的是某些唯一的知识。但是，口令往往很短又容易联想（例如某个宠物的名字），以利于记忆，因此很容易被入侵者破译，只要他愿意尝试足够多的备选项的话。同样，倘若某个口令通过网络来发送的话，它就有可能被截获，从而被破译。

安全性可以通过要求某个惟独该人才持有的物理项目（例如一张令牌卡片）来得到加强。一种更加安全的技术涉及到采用某些生物学标识，如指纹或嗓音。遗憾的是，生物学机制的代价很高，而且这些特征的任何计算机表示也都可能被复制。

然而，口令在密钥发布和身份鉴别协议中仍然起到重要作用，这些协议包含一些支持交换已加密消息的新技术。由于只需要少量的消息，因此加密的开销一般不太大，所以这些协

设既可以基于对称加密法（例如Kerberos，参见27.4.1节的讨论），也可以基于公钥加密法（例如SSL，参见27.7.1节的讨论）。然而，由于公钥加密法的计算要求很多，因此实际的数据交换一般都是采用建立在会话密钥基础上的对称技术。

27.4.1 Kerberos协议：票据

作为协议的一个例子，该协议采用对称加密法来为服务器鉴别每个客户的身份，并为随后的数据交换发布一个对称的会话密钥。我们叙述的是广泛采用的Kerberos系统的一种简化版本。该系统是由MIT（Steiner et al. 1988，Neuman and Ts'o 1994）设计的。Kerberos是一种商业规用的中间件模块，它能够整合到某个分布式计算系统，并可以由某个TP监视器来提供服务。

Kerberos协议采用一个称为**密钥服务器**（key server）的中间过程。实际上，Kerberos将它的密钥服务器称为KDS（Key Distribution Server，密钥发布服务器）。该密钥服务器根据需要创建会话密钥，并以一种仅有正在通信着的进程才知道的方式来发布密钥。因此，它被看成是一种**可信任第三方**（trusted third party）。

每一个想要在某一时刻通信的用户首先要向密钥服务器KS注册一个对称的**用户密钥**（user key）。用户密钥并不是会话密钥，而是仅在某次会话启动时用来通报会话密钥的。

假设客户C想要与某个服务器S通信。C和S事前各自分别向KS注册了用户密钥$K_{C,KS}$和$K_{S,KS}$，$K_{C,KS}$仅为C和KS所知，同样，$K_{S,KS}$也仅为S和KS所知。KS是C所信任的，即C假设KS从不向任何其他进程通报$K_{C,KS}$，而KS用来储存$K_{C,KS}$的数据结构足以杜绝任何未经授权批准的访问。同样地，KS也是S所信任的。

Kerberos引入了**票据**（ticket）的概念来发布会话密钥。为了理解票据的作用，不妨考虑图27-2所示的以下的处理步骤，它们组成了该协议的核心部分。（与这里介绍的其他协议一样，我们省略了一些次要的细节。）

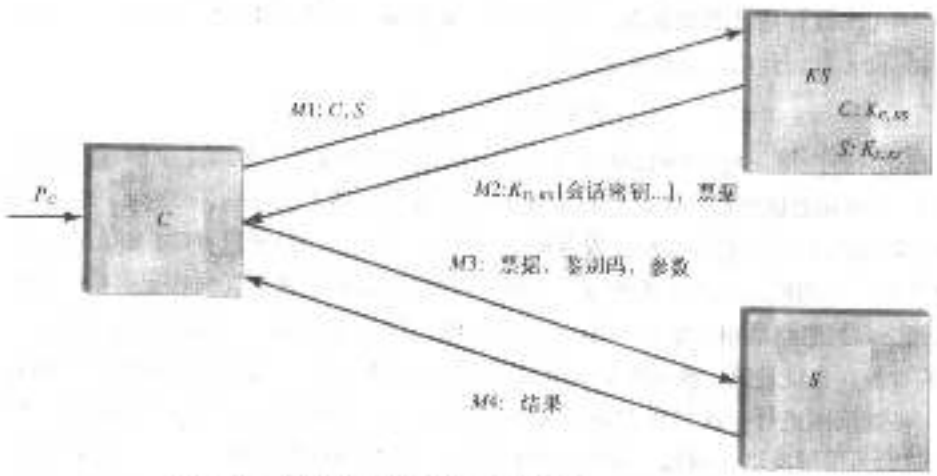


图27-2 对称加密法中用于鉴别某个客户身份的消息序列

1）C向KS发送（透明地）一条消息$M1$，请求一张票据来向S表明C的身份。$M1$包含通信双方的名称(C, S)。

2) 当KS收到$M1$时，它采取以下的动作:

a. KS先（随机地）构建某个会话密钥$K_{sess,C\&S}$。

b. 接着KS向C发送一条包含下述两项内容的消息$M2$:

i. $K_{C,KS}[K_{sess,C\&S}, S, LT]$

ii. $K_{S,KS}[K_{sess,C\&S}, C, LT]$——即实际的票据

其中LT是该票据生效的生命周期（时间间隔）。

3) 当C收到$M2$后，它实施以下步骤:

a. C根据$K_{C,KS}$的第一项内容恢复$K_{sess,C\&S}$（它无法解读该票据）。

b. C储存该票据直到它准备好向S请求某些服务。

值得注意的是，KS并不知道$M1$的实际来源，$M1$完全可能是一个冒充C的入侵者I所发送的。然而，由于KS加密了$M2$，使得其返回的消息只能为C和S所访问。

当需要采用主角的口令P_C以支持C的登录时，C则在某种保护模式下构建它的用户密钥$K_{C,KS}$，而不是储存其用户密钥。这是在某个单向函数f的帮助下完成的。

$$K_{C,KS} = f(P_C)$$

因而，只有C可以利用f创建出$K_{C,KS}$，并且只有C才能够从$M2$的第一项检索出$K_{sess,C\&S}$: 该消息是KS发送给声称为C的进程的。注意，该协议并未在网络上发送P_C，因此避免了口令被复制的可能性。另外，由于C并不储存$K_{C,KS}$，因此它被窃取的可能性也降低了。

以后，当C想要请求S支持某种服务器操作时，就可以采取以下的动作:

4) C向S发送某个消息$M3$，其中包含请求调用的参数（既可以通过$K_{sess,C\&S}$对其加密，也可以不加密）、该票据和一个鉴别码（参见下文）。

只有S才可以解读该票据，并复原已加密项。然而，单凭（包含C的）票据是不足以向S认证C的身份，因为I可能在步骤2中复制了该票据，并在它自己的调用中向S回放。利用时间戳也许可以防止回放，可是时间戳不能存储于此票据中，因为此票据可以由C在其生命周期内多次使用。所以C就伴随其票据发送一个鉴别码。**鉴别码**（authenticator）包含一个（当前）时间戳TS和C的名称，经$K_{sess,C\&S}$加密而成的:

$$\text{鉴别码} = K_{sess,C\&S}[C, TS]$$

这意味着它只能使用一次。S可以通过（由解密该票据而确定的）$K_{sess,C\&S}$来解读此鉴别码。

至此，S便知道该票据只能是KS所创建的，因为只有KS才知道$K_{S,KS}$。另外，由于S信任KS，而且每次传递的$K_{sess,C\&S}$都是由$K_{C,KS}$或者$K_{S,KS}$加密的，所以，S知道只有C才知道$K_{sess,C\&S}$。鉴别码里包含着一些由$K_{sess,C\&S}$加密的明文（例如C），它可以和该票据（同样包含C）所包含的内容相对照。如果它们是相匹配的，S就可得出结论: 肯定是C创建了此鉴别码。然而，为了向S证实C的身份（也就是说，使其相信该调用请求确实来自于C），还必须排除几种可能的攻击。

1) I可能试图进行一次回放攻击，其中它拷贝了该票据和来自于$M3$的鉴别码，并在以后使用它们。为了解决这个问题，我们必须使得每个鉴别码只能使用一次（与票据相反）。C为它的每一次调用创建一个新的（带有某个唯一时间戳的）鉴别码。只要它的时间戳处在相关票据的生命周期（LT）内，它就是*活动的*（live）。为了确保拷贝鉴别码是没有任何价值的，S采用以下的协议，以使其保护自身免受回放攻击:

a. 如果收到的鉴别码不是活动的，S就拒绝它。

b. S维护一份其所接收到的、目前仍然在活动的鉴别码的列表。如果接收到的鉴别码是活动的，则S便将它和该列表进行对比，一旦发现已经有拷贝，就拒绝接受。通过保持生命周期信息，S便能够限制必须保存在此列表中的鉴别码的数量。

2) I截获了消息$M3$（它没有到达S），并试图采用该票据和鉴别码来为其自身调用某个服务。然而，如果C选择了利用$K_{sess,C\&S}$来加密其调用的参数，那么I就不可能用自己的参数来替代它，因为它并不知道$K_{sess,C\&S}$。对于I来说，在稍后的时间里发送整个被截获的消息是没有什么实际意义的，因为那只不过导致满足C的原始请求。

3) I截获$M1$并将其置换成消息(C, I)。在这一攻击中，I的目的是为了迷惑C，使其误认I为S，并且让C以为它是在通过$M3$向S发送其参数，从而获得包含在该参数中的关于C的私密消息。协议通过将服务器的名称包含在$M2$的第一项里，防止了这一攻击。C利用这一消息来确定该进程的身份，即它应当能够解读其调用消息。

该协议可以提供很多级别的保护。客户可以要求仅当首次建立对服务器的连接时才进行认证。也可以要求每次请求服务时都进行认证，还可以要求利用$K_{sess,C\&S}$来加密（在$M3$中的）调用的参数、（在$M4$中的）返回的结果，以及同时加密其鉴别码。

我们已经给出了参数，以阐释Kerberos如何防止一个入侵者试图进行的各种攻击。不要以为我们的讨论已经证明Kerberos是安全的，其实并没有。这样的证据是正在进行研究的课题。

一次性签名

Kerberos提供了一种称为**一次性签名**（single sign-on）的特性，随着客户交互变得越来越复杂，它也变得越来越重要。复杂的交互经常涉及访问多种资源，以及访问多台服务器。每一台服务器都需要认证该客户的身份，而在最坏的情况下，为了完成这项任务还要有它自身固有的接口。另外，该客户所使用的口令，对于它所访问的每一台服务器来说可能都是不同的。因此，每个客户都必须记住多套口令，并涉足多个认证协议。而每当客户信息有所变动时，系统管理员还必须保持着当前与每一台服务器相关联的认证信息。

采用一次性签名方式，该客户只需要认证其身份一次。Kerberos通过将认证集中在**认证服务器**（authentication server）（类似于KS）中，来提供这一特性。它在登录时通过该客户提供的口令来认证C的身份，就像我们前面所描述的一样。由于该客户想要访问的服务器的身份在这一时刻可能还不知道，因此，认证服务器不可能创建出合适的票据（每一张票据都是用某个特定的服务器密钥加密的）。相反，它向C返回一张**票据授权票据**（ticket-granting ticket），这是用来向某台服务器请求服务的，这台服务器称为**票据授权服务器**（ticket-granting server），即TGS，也是Kerberos的一部分。其后，C就可以通过票据授权票据向TGS请求它所需要的特定的票据（例如，用于S的一张票据）。

认证服务器生成C用来与TGS通信的一个会话密钥$K_{sess,C\&TGS}$，并将其返回给C（其格式与上述简化协议中的$M2$的相类似）。

- $K_{C,KS}[K_{sess,C\&TGS}, TGS, LT]$——$K_{sess,C\&TGS}$是用于与$TGS$通信用的一个会话密钥。
- $K_{TGS,KS}[K_{sess,C\&TGS}, C, LT]$——用于$TGS$的票据授权票据。

其中$K_{TGS,KS}$是TGS向KS注册的密钥。

然后，当C想要访问某台特定的服务器S时，它便将该票据授权票据的一份拷贝连同该服

务器的名称（和一个鉴别码）一起发送给*TGS*。*TGS*然后再返回给*C*（其格式仍然与*M*2的相类似）。

- $K_{sess,C\&TGS}[K_{sess,C\&S}, S, LT]$——$K_{sess,C\&S}$是用于与*S*通信用的一个会话密钥。
- $K_{S,KS}[K_{sess,C\&S}, C, LT]$——用于*S*的票据。

于是，*C*为它打算访问的每一台服务器都获得了一张不同的票据。它参与的是一个认证协议，而且，由于票据的使用在用户级是不可见的，其用户接口也得到了简化。此外，由于认证集中在单独的一台服务器里，所以认证信息的管理也得到简化。

27.4.2 临时串

临时串（nonce）是由一个进程所创建的某个比特串，其创建的方式使得其他的进程很难再创建出相同的串。例如，由某一个进程随机创建的有足够长度的比特串，不大可能以后再被其他进程创建。临时串有多种用途，其中之一是和认证有关的。

当进程P_1和P_2共享一个会话密钥K_{sess}，而且P_1向P_2发送一个加密的消息*M*1并期望得到一个加密的应答*M*2的时候，就会碰到需要通过临时串解决的问题。当P_1收到*M*2时，它如何肯定这是由P_2发出的？看起来似乎P_1只要采用K_{sess}解密*M*2再看其结果是否有意义就可以了。然而，要确定某个串是否有意义，一般都需要人工的干预；而在有些情况下，甚至连人工干预都不起作用。不妨考虑下述情况，即*M*2只包含由P_2计算出来的数据串（一种完全任意的比特串）。*I*可能会用某个随机的串来替换*M*2。当P_1采用K_{sess}来解读该串的时候，它可能产生看来像是数据串的另一个串。但是，不重复P_2的计算，P_1就无法确定该串是否正确。作为一种选择，*I*可能会回放在同一个会话期间发送的并且是用K_{sess}加密的某个较早的消息。在某些情况下，这样的一种回放也许可能是对*M*1的一种正确的回应，从而，P_1会由于接受它而造成错误。

在该问题的临时串解决方案中，P_1在*M*1中包含一个临时串*N*，而P_2则在*M*2中包含有$N+1$。一旦接收到*M*2后，P_1就知道发送者一定能解读*M*1，因为返回的是$N+1$，而不是简单地回放*N*。这意味着发送者知道K_{sess}，所以必定是P_2。

在Kerberos中，时间戳*TS*（已经是鉴别码的一部分）就可以作为一个临时串，因此不再需要添加附加项。该服务器可以在*M*4中包含$TS+1$。

临时串经常会由于一种完全不同的原因而应用于加密协议中。若在加密消息前对明文附加一个大的随机数，那么对于该入侵者来说，指望通过猜测其部分内容（例如，猜测截止日期或者某个信用卡号中的一些冗余信息）并通过这些信息来降低硬性搜索密钥的开销，从而破译该密钥是极端困难的。临时串的这种作用有时候可看成向某个消息添加**调料**（salt）。我们后面将讨论的很多协议中，都采用着加过调料的消息。但在我们的讨论里，则省略了协议的这部分内容。在某些协议中，用于此目的的临时串称为**盅惑者**（confounder）。

27.5 授权

认证了某个客户的身份后，服务器必须接下来要决定是否应该批准其所请求的服务。因此，当一个来自于某个客户的访问特定对象的请求到达时，系统的认证组件必须决定是否应该允许此主角去访问其所请求的对象。这称为**授权策略**（authorization policy）。某个对象可以被访问的模式取决于该对象受保护的类型。

例如，如果该资源是一个文件，那么访问的模式可能是读取、写入、追加和执行。操作系统一般通过以下方式来控制对存储在文件系统中的文件的访问，即首先要求在登录时认证主角自己的身份，然后检查对该文件的每一次访问是否事先获得此文件的所有者的授权。通常（在Windows NT和UNIX的许多版本中）用来记录系统的保护策略的数据结构是ACL（Access Control List，访问控制列表）。

每个文件都有一个相关的ACL，该ACL的每一项都标识着某个主角，并为每一种可能的访问模式都包含1位。第i位对应于第i种访问模式，并说明是否允许某主角以该种模式来访问此文件。图27-3显示了某个文件的ACL。前5位包含一个Id。随后的每一个位对应于一种访问模式：读取、写入、追加和执行。该图说明Id为11011的用户有权读取和写入该列表所对应的文件。

Id	r	w	a	x
11011	1	1	0	0
00000	1	0	0	0

图27-3 某个文件的访问控制列表

从实用的角度考虑，有必要介绍**组**（group）的概念。例如，单独地在某个普遍可以读取的文件的ACL中列出所有的主角是一种很笨拙的做法。简单地讲，组就是一系列的主角，它对应于ACL的某一项。包含在某项中的访问权限，是允许该组的所有成员使用的。因此，正如图27-3所说明的那样，一个普遍可读取、但只有它的所有者才能够写入的文件，可能持有一份由两项组成的ACL。其中一项是针对Id为11011的所有者，设定读取、写入位置为1；而另一项则是针对包含所有主角的组，仅将读取位置1。Id 00000标识由所有用户组成的组。通过引入组，某个ACL中的几个项可能都是指向同一个特定的主角（例如，某个主角可能属于好几个组）。在这种情况下，该主角的权限是每个项的并。

事务处理系统中存储和实施某个授权策略的结构，只不过是操作系统保护文件时采用的ACL/组结构的一种小的推广。资源是由服务器控制的，所以每个服务器（而不是操作系统）都要负责存储ACL，并对它所控制的资源实施授权策略。服务器将输出某些方法，使它们可供客户调用。这些方法构成访问该服务器所封装的资源的模式。简而言之，授权策略就是针对某个特定主角可以调用的方法以及此方法可以调用的特定资源的一种规定。例如，在学生注册系统中，修改分数的方法不能让任何学生调用，修改描述学生个人信息（如学生的地址）的方法也只能为此学生本人所调用，而设定对班级注册人数限制的方法可以由任何学校的教师调用。

ACL用来支持这种推广。在某种方法里，学生注册系统服务器可能会采用某个单独的ACL，该ACL包含一个针对学生组的项和一个针对教师组的项。在每一个项中的权限位对应于该客户可以调用的方法。一个项只需要有足够的位，就可以使所有输出方法都和不同的位相关联。在这种情况下，教师组被授予调用修改分数和班级注册人数方法的权限，而学生组则被授予调用个人信息方法的权限。某个学生的个人信息只能由该学生本人更改的附加性要求应当单独地进行检查。当调用某个方法时，服务器就会检查ACL，若没有给用户授予必要的权限的话，就会给调用者返回一个错误代码。

另外，我们可以认为服务器管理一些不同的资源（班级记录、个人记录）并分别为每一条记录分配一张ACL。在这种情况下，就可以实施一种粒度更为精细的保护。例如，有关每个学生的个人记录的ACL都拥有一个仅包含该学生本人的项。

利用SQL GRANT语句是客户为某张表指定授权策略的一种方法。在这种情况下，访问模式是对应于各种SQL语句类型的。SQL服务器可以采用ACL来实现授权策略。此时，利用GRANT语句可修改与该语句所指定的表。

每一台服务器一般负责提供它自身的授权模块，因为它所控制的对象以及对这些对象的访问都是该服务器专有的。相反，一个主角的身份标识和组成员标识可以是对所有的服务器都通用的。每台服务器中的授权模块称为**引用监视器**（reference monitor），它负责构建、检索和解释访问控制列表。实现这一目标的中间件模块可能是由某个TP监控器提供的。

27.6　已认证的远程过程调用

许多要求认证和授权的分布式应用程序的目标是，对其主角隐藏所需协议的复杂性。一旦该主角执行了登录过程，服务调用应该是很简单的，而认证和授权也应该是不可见的（除非侦测到某个入侵）。实现这一目标的一种方法是，在RPC存根中实施认证，就象图27-4中所显示的，并对于客户和服务器表示为已认证RPC（authenticated RPC）的抽象物。这是在分布式计算的DCE模型中常用的一种实现方式。

在这种实现方式里，中间件提供了该客户可以用来登录和调用服务器的API。此API和用于交换消息（图27-2所示的消息）的存根相连接。密钥服务器现在称为**安全服务器**（security server），因为它既实施了认证，同时又在实施授权中发挥着重要作用：它跟踪所有每一个主角所属的组，而且它在发送给该主角的一张票据中还包含了该主角所属组的Id的一份清单，安全服务器可以像商业现用中间件模块那样容易地获取。

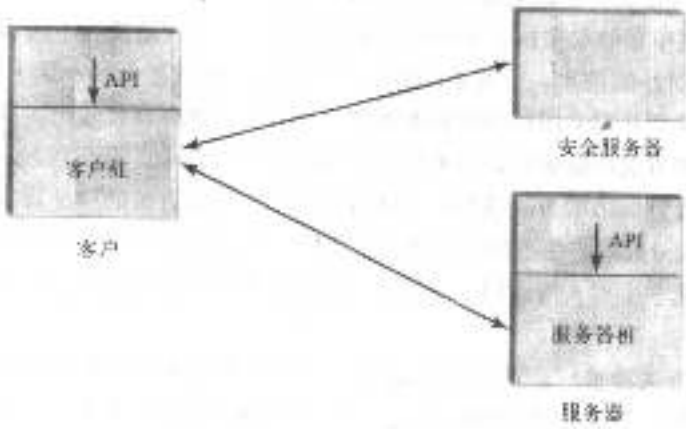


图27-4　桩和身份认证之间的关系

每台服务器负责安全地在本地存储其服务器密钥$K_{S,KS}$的一份拷贝，同时使得每当接收到某个调用消息时，该拷贝都可供其服务器存根使用，然后，该存根就可以解读此消息中包含的票据，以确定该客户的身份及其所属的组。

服务器桩API提供了一些调用，服务器可以使用这些调用来检索该客户的身份以及在服务器桩里的票据中所包含的组成员信息。对于每个客户调用的授权是由引用监视器在服务器（而不是服务器桩）中提供的，其中使用了客户的（已认证的）身份、组成员信息和包含在服

务器里的访问控制列表。

27.7 因特网商务

当事务通过因特网执行的时候，安全性问题尤其重要。认证对于防止一个站点成功地假扮成另一个站点是很重要的。加密对于防止窃听也是很重要的。另外，在参与因特网事务的团体之间普遍存在着一种猜疑，这或许是因为没有面对面地交流的缘故，也可能是因为可以在一夜之间就冒出某个耸人听闻的网站的缘故。正因为这些原因，经常需要认证彼此的身份。

我们需要区分两类因特网商务事务：C2B（Customer-to-Business，顾客对企业）和B2B（Business-to-Business，企业对企业）。这两类事务的安全性要求是相似的，但我们假设顾客仅拥有和其浏览器配置的安全软件，而企业则可以有更复杂和更专业的保护，比如他们自己的公钥和私钥，而这些都是顾客所没有的。

27.7.1 SSL协议：证书

1. 证书

作为某个因特网事务的一部分，若服务器（也许是代表企业）想要对其他各方表明它的身份，那么可以利用某个CA（Certification Authority，认证机构），后者扮演着可信任第三方的角色。有很多公司都在从事着认证机构的业务。

CA采用公钥加密来产生**证书**（certificate），它将证明某个主角的名称(例如Macy)和它的公钥之间的关联性。该证书（在其他项中）包含该主角的名称和公钥，而且它是用CA的私钥签署的。由于CA的公钥是公开的（而且很可能事先已存储在用户的浏览器里），所以系统中的任何进程都可以据此来确定该证书的有效性。因此，如果某个客户想要安全地与Macy通信，它就可以利用在包含名称“Macy”的有效证书中找到的这个公钥来加密某段消息，并且可以肯定仅有知道Macy私钥的进程才能解读那段消息。因而，该证书解决了可靠地发布公钥的问题，这是非对称加密的密钥发布问题。它们可以在下面描述的协议中采用。

任何想要从CA获得一份证书的因特网服务器S首先要产生一个公钥和私钥对，并将其公钥再加上其他的信息发送给CA。CA会采用多种方法来证实该服务器的身份（也许在Dun和Bradstreet上寻找它，也许是通过电话和邮件与该企业的服务器所在地的人员进行通信），然后向它颁发一份证书，在其他项中包括

- CA的名称。
- S的名称。
- S的URL。
- S的公钥。
- 时间戳和截止日期信息。

CA签署此证书，并可能通过电子邮件不加密地向S发送已签名的证书。然后，S证实它的正确性（例如，存储于该证书中的公钥确实是S的公钥）。值得注意的是，对于入侵者而言，一份证书就是其可以轻易地获得的公共信息。但它对于入侵者是没有任何作用的，因为每个想要与S通信的客户将会采用S的公钥来对消息加密。由于S的私钥并没有包含在证书中，这样仅有S才能够解读该消息。

2. SSL协议

SSL（Secure Sockets Layer，安全套接层）协议[Netscape 2000]利用证书来支持因特网上的客户与某台因特网服务器之间（或者几台服务器相互之间）的安全通信和认证。通过使用证书，SSL可以消除对于某个联机密钥服务器（就像Kerberos场合那样）的需求，在一个每秒处理数千个事务的事务系统中，联机密钥服务器可能是一种瓶颈[㊀]。

SSL的一个目的是向客户认证某台服务器。由于这是利用证书来完成的，所以每台想要认证的服务器必须首先获得一份证书。另一方面，客户一般不需要向CA注册，因而也不必持有证书或与它们相关的加密密钥[㊁]。客户的登录一般是通过浏览器表示的，浏览器（通常）没有它自己的私钥。然而，浏览器往往包含和该浏览器的供应商达成了共识的所有CA的公钥。浏览器在SSL协议期间实际上并没有和某个CA进行通信，而每个CA也确实不知道关于浏览器的任何私有信息。

SSL协议向客户认证服务器的身份，并创建一把会话密钥供它们使用[㊂]。

假设浏览器C连接到一台声称代表某企业E（例如，Macy）的服务器S。在这种情况下，该协议由以下步骤组成:

1) S向C发送带有CA签名的证书的一份拷贝（以未加密方式发送）。

2) C验证采用包括在其浏览器里的CA公钥的证书，并因此知道证书中的公钥确是E的公钥。

3) C产生一个用E的公钥加密的会话密钥，并发送给S[㊃]。

请注意，这里是由C（而不是S）来产生会话密钥的，因为在该协议的这一点上，C可以采用E的公钥安全地与S通信，但是S却不能安全地与C通信（也不存在任何联机密钥服务器（像Kerberos中那样）来产生会话密钥）。

一旦创建了会话密钥，C和S（现在已经知道它是代表E的）就可以用它来交换加密消息。协议是在数据传输层（TCP/IP）和应用程序层中间的通信层上运行的，对于应用程序来说协议的执行是不可见的。

如果浏览器采用会话密钥向S通报某个信用卡号码，那么对于通信的安全性，用户可以有足够的自信。浏览器本身产生出会话密钥，它采用E的公钥与S通信，而且该用户也知道它采用的确是E的公钥，因为它是从一份仅可能由CA产生的证书里获得的。用户相信CA已经验证过E的身份，并在证书中包含关于E的正确的信息。同时相信浏览器的供应商已经将每个CA的正确的公钥和SSL协议的一种正确的实施包括在其浏览器中[㊄]。用户也必须相信它的浏览器没

㊀ 但是，请注意：证书有一个显著的缺陷，一份证书一旦通过CA授权给某台服务器，那么以后需要吊销时将是很困难的。相反，一个联机密钥服务器则可以简单地中止向某台特定的服务器提供密钥。

㊁ SSL对于确有证书的客户有一份可供选用的认证协议。

㊂ 出现在SSL协议中的一台浏览器在它连接到一台URL以https:（而不是一般的http:）开头的服务器时，就表明那是一个采用SSL加密的HTTP协议。

㊃ SSL实际上还要复杂一些。C产生一个**先掌握的秘密**（pre-master secret）并发送给S,据此，C和S采用同样的算法，独立地产生两个会话密钥——每个方向的通信各一个。这将有效地提高安全程度。C和S还可以用它来验证该协议的应用程序部分中消息的完整性。

㊄ 尽管SSL 协议解决了使客户获得某台服务器的公钥的密钥发布问题，但在其解决方案里又包含了另一个密钥发布问题：需要使该客户获得CA的公钥。

有遭受某个恶意程序（也许它在早些时候曾经下载过的）的侵害。

至此，协议已经向此客户认证了服务器的身份，但客户还没有向服务器认证自己的身份。对于很多应用程序来说，客户认证并不是必需的。例如，大多数服务器会接受（可以提供信用卡号的任何浏览器的）某张信用卡的采购，而无需判断该浏览器是否真正代表该卡的持有者。（大多数电话订购目录公司都以类似的条件接受订单。）

对于另外一些应用程序来说，S确实想要确定它是否在和某个特定客户对话（例如，在发送它的私人证券信息或者接收某个股票交易事务之前）。向客户和服务器提供这样的认证的一种方法是商定某个口令（也许通过电话），服务器存储这个口令，客户在创建会话密钥后提供该口令。对于客户来说还有另一个方法，即持有一份证书，于是客户和服务器双方就可以彼此认证身份。

27.7.2 SET协议：对偶签名

许多零售商在顾客采购事务中采用SSL协议。在创建会话密钥后，顾客发送需要采购的物品的细节和信用卡信息给零售商的服务器，通过在代表信用卡公司的某个其他站点上验证信用卡来完成该事务。其中唯一的漏洞是让零售商知道了顾客的信用卡号码。

当然，在大多数的非因特网顾客—零售商事务中，零售商之所以能够获得某个信用卡号码，是因为顾客将她的信用卡交给了零售商[⊖]。然而，将信用卡号码透露给零售商，在电子商务中是会产生问题的，这有两方面的原因。

- 因为采购时只需要信用卡号码（而并非信用卡本身），所以某个不诚实的零售商有可能在信用卡持有者完全不知情的情况下使用该号码，直到该问题暴露为止。
- 另外，因特网商务的匿名性不利于促进零售商和顾客之间的信任关系。因此，如果零售商不知道顾客的信用卡号码，那么顾客将会更加安心些。

为了促进因特网上电子商务的发展，技术人员已经开发出了更加安全的协议。SET（Secure Electronic Transaction，安全性电子事务）协议[VISA 2000]就是其中之一，它是由Visa和MasterCard共同开发的。SSL是一个**会话级安全协议**（session-level security protocol），它确保了会话期间的安全通信，而SET则是一个**事务级安全协议**（transaction-level security protocol），它确保了采购事务（其中包括一个原子化提交的）的安全性。

SET协议是很复杂的，需要用很多签名和许多交叉检验来提高总体的安全性。这里我们介绍一个简化的版本，它展示了对零售商隐藏信用卡号码的技巧以及如何原子化提交采购事务。

该协议涉及到两个新的概念。

- 每个顾客都有自己的证书，因此也有自己的公钥和私钥。这些密钥用来提供该协议的一种独特的特性，即**对偶签名**（dual signature）；它在很大程度上提高了事务的安全性[⊖]。顾客的证书还包含有一个他信用卡号码和信用卡有效日期的消息摘要。请回忆一下，证书中的这些信息是不加密的。因此，只有摘要（而不是信用卡号码本身）可以被包括在其中。摘要可用来证明此信用卡号码是该顾客提供的，也就是说它对应于该顾客的某张

⊖ 例如，当你去一家餐厅就餐的时候，你将你的信用卡给了一名侍者，她验证该事务并返回一张收据让你签名。可是你怎么知道她没有复制你的信用卡号码呢？

⊖ 有些SET版本采用一种改进的对偶签名，它不需要顾客持有自己的证书(以及公钥和私钥)。

信用卡。

- 一个新的服务器，即代表信用卡公司操作的**支付网关**（payment gateway）G。SET是一种涉及顾客、零售商和支付网关的三方协议，支付网关在其协议期间担当可信任第三方的角色，并在该事务结束时完成提交操作。

该协议的基本思想是，顾客C向零售商M发送包含两个部分的消息。消息的第一部分包括了采购的总额和采用G的公钥加密的C的信用卡信息（所以M看不到该信用卡信息）；消息的第二部分包括采购的总额和采购的细节（但没有信用卡信息），这一部分是采用M的公钥加密的。然后M将该消息的第一部分转发给G，G将它解密，验证该次信用卡采购，并提交该事务。

对于这种由两部分组成的消息可能有一种攻击，某个入侵者可能会将某一个消息的第一部分和另一个消息的第二部分连在一起。例如，一旦截获Joe和Mary采购的消息，入侵者就可以将Joe消息的第一部分与Mary消息的第二部分连接起来，使Joe支付Mary货物的费用。阻止这类攻击的一种方法是，让M对每个事务都关联一个唯一的Id，并要求C在消息的两部分里都包含它。这样，任何试图合并不同消息的两部分的企图就很容易侦测到了。然而，这还不足以解决任何不诚实的零售商的问题，他可以让两个不同采购事务都以相同的Id联系起来，从而使这两个结果部分的消息有可能结合在一起。我们需要采用一种新的机制来克服这类问题。这种机制就是我们接下来要介绍的对偶签名。

在SET开始以前，C和M协商采购的事项。该协议以一次握手开始，此时C和M交换证书并认证彼此身份。C将其证书发送给M，而M则将它自己的和G的证书一起发送给C；通过握手，C和M都知道了对方的和G的公钥。然后便开始其采购事务。

1) M发送已签名消息给C，其中包括某个（唯一的）事务Id（这可用来防止回放攻击）。C采用M的证书里的公钥来检验签名，从而知道该消息确实来自M，而且在传输中没有被改变。

2) C向M发送一条消息，其中包括两部分以及对偶签名:

a. 事务Id，C的信用卡信息和采购总额（但没有采购物品的说明）——以G的公钥加密:

$$m_1 = K_G^{pub}[trans_Id,\ credit_card_inf,\ \$_amount]$$

b. 事务Id，采购总额和采购物品的说明（但没有C的信用卡信息）——以M的公钥加密:

$$m_2 = K_M^{pub}[trans_Id,\ \$_amount,\ desc]$$

其对偶签名包含三个字段:

a. 消息的第一部分的消息摘要MD_1: $MD_1 = f(m_1)$

其中$f()$是消息摘要函数。

b. 消息的第二部分的消息摘要MD_2: $MD_2 = f(m_2)$

c. C对于MD_1连接MD_2的签名: $K_C^{pri}[f(MD_1 \cdot MD_2)]$

于是，完整的对偶签名是

$$dual_signature = MD_1,\ MD_2,\ K_C^{pri}[f(MD_1 \cdot MD_2)]$$

从而，由C发送给M的完整消息是$(m_1, m_2, dual_signature)$。

对偶签名将该消息的两个部分捆绑在一起。所以，例如，某个入侵者或者M试图将m'_2与m_1联系在一起的企图是不会奏效的，因为它们的消息摘要MD'_2与MD_2是不一样的。虽然在对

偶签名中可以用MD'_2替换MD_2，可是$K_C^{pri}[f(MD_1 \cdot MD_2)]$不能够作为$MD_1 \cdot MD'_2$的签名，因为只有C才可以为重组后的消息计算出正确的对偶签名。

3) M可以用它的私钥解读此消息的第二部分（但是它不能破解包含着信用卡号的第一部分）。该零售商然后

a. 利用对偶签名来验证m_2在其传输过程中没有被更改。它首先计算出m_2的消息摘要，并检查它是否和数字签名(MD_2)的第二部分一样。然后采用C的证书中的公钥来检验第三部分确实是第一和第二部分的连接的正确签名。

b. 核对该事务的Id，订单总额和采购物品的说明。

接着M给G发送包括两个部分的一条消息:

a. m_1和从C处接收到的对偶签名: $m_3 = m_1$, *dual_signature*

b. 该事务Id和订单总额——以M的私钥签名并采用G的公钥加密:

$$m_4 = K_G^{pub}\left[trans_Id,\ \$_amount,\ K_M^{pri}[f(trans_Id,\ \$_amount)]\right]$$

M发送给G的完整的消息是(m_3, m_4)，以及C和M的证书的拷贝。

4) G采用它的私钥来解读该消息。

a. 利用对偶签名和C的证书里的公钥，来验证m_1是否是由C准备的，而且没有被更改过（就像在步骤3a中一样）。

b. 利用C的证书中的信用卡信息的消息摘要，来验证其信用卡信息是由m_1提供的。

c. 利用m_4中M的签名和M的证书中的公钥，来检查m_4没有被更改过。

d. 检查其事务Id和采购总额在m_1和m_4中是否相同（以验证M和C是否就采购达成一致）。

e. 检验事务Id是否从来不曾提交过（以防止回放攻击）。

f. 进行任何必要的工作，以验证此项信用卡请求。

然后G给M返回一个签过名的*已验证消息*。至此，该事务才被提交。

5) 一旦M接收到此*已验证消息*，它就知道该事务已经提交了。然后，它给C发送一个签过名的消息: *事务完成*。于是，C也知道该事务已经提交。

请注意，该协议是如何处理一些其他的攻击的。

1) M无法替换不同的货物，因为该对偶签名是建立在C所认同的描述之上的。一旦向G转发了此对偶签名，M就已经自己提交了该描述。

2) C也无法利用从其他客户提交的消息中拷贝出来的m'_1，以图通过将其和m_2进行粘合而使那个客户支付C的采购。在这种情况下，对偶签名不会起作用，因为它由C计算的。然而，m'_1和m_2会有不同的事务Id，所以该事务将会遭到G的拒绝。

用于SET的原子化提交协议

当G提交事务时，它将记录合适的日期数据，使该事务持久化。M也可能希望在接收到G的验证消息时提交它的子事务。许多顾客可能不希望进行正式的提交，但是C、M和G之间的消息交换可以看成是一种线性提交协议，在这种语境下

- 步骤2中C发给M的消息和步骤3中M发给G的消息都是*投票消息*。在它们发送之前，C和M必须处在某种准备状态下。
- 在步骤4中G发给M的消息和步骤5中M发给C的消息都是*提交消息*。在发送那些消息之前，G和M必须将相应的提交记录输入到它们的日志里。

值得注意的是，G是一个可信任第三方，在完成提交时是受到其他两方的信任的。

27.7.3 货物原子性、托管与已认证交付

某些因特网事务涉及到采购物品的实际交付。涉及采购下载软件的事务就属于这一范畴。这样的事务[⊖]应当是**货物原子化**（goods atomic）的，也就是说，当且仅当货款支付后，货物才能交付。在SET协议的语境中，“已付款”意味着采购事务已经为支付网关所验证；在27.7.4节将讨论的电子现金协议的语境中，它意味着电子现金已交付给零售商，并已为银行所接受。

涉及采购实际货物的事务一般都不是货物原子化的。顾客订购了货物，然后事务提交，（大多数情况下）在这之后，零售商便装运货物。然而，在因特网商务中，当该事务提交后，顾客可能不信任让零售商来发送（下载）货物。

货物原子性实际上并不是一个新概念。其原始思想是，交付货物这一事件也是事务的一部分（其中已为货物付款）。当且仅当已付款时才交付货物的要求，恰恰是原子化事务执行的通常的定义。实现货物原子性的难点在于交付无法回退，这就意味着若货物在事务提交前就交付了，而该事务随后异常中止了，那么没有任何办法可以撤销交付，故而执行不是原子化的。

在21.3.2节中，我们谈论过一种类似于货物原子性的情况，其中，当且仅当该银行提交此取款事务时，某个ATM才发放现金。我们讨论过如何利用一个可恢复队列来达到此目的。现金发放和货物原子性都涉及当且仅当该事务提交时发生的某个外部事件。

货物原子性的概念及实现它的某个协议，是在开发NetBill系统时提出来的[Cox et al. 1995]。该协议与涉及一位客户、一个零售商和某个可信任第三方的SET有些相似之处，后者通过某个线性提交协议来有效地完善某个信用卡事务。我们并不描述NetBill，而是要描述如何加强SET协议，以实现货物原子性。

在C和M就某项事务达成共识后，但在C向M发送确认之前（SET的步骤2），M用某个新的对称密钥$K_{C,M}$（M创建这个密钥是为此目的）向C发送（下载）加过密的货物。M同时还发送该加密货物的一份消息摘要，所以C就可以证实已正确地收到此加密货物。请注意，此刻C尚不能使用这些货物，因为她并不知道$K_{C,M}$。

在C发送给M（在m_2中）的描述*desc*里，既包括该货物的一份规格说明，又包括用C的私钥签名的已加密货物的消息摘要。实际上，C知道它已经收到了（以加密的形式）与该摘要相对应的货物。正如SET的步骤2一样，完整的消息是(m_1, m_2, *dual_signature*)，而这也是C投票提交的。当收到了C的消息后，如果M认为其从C处收到的描述是准确的，那么如SET的步骤3一样，M便创建消息(m_3, m_4)，不过在m_4中还包括如下两个附加项：

- $K_{C,M}$
- 用C的私钥签名的已加密货物的消息摘要（它是在步骤2中从C处收到的）再一次用M的私钥进行签名（会签）。

M然后将此消息发送给G（SET的步骤3）。

一个不道德的零售商也许想报一个更高的价格，但他无法更改该事务的项，因为C所建立

⊖ 在此语境中“事务”一词的用法是宽松的。货物的实际交付也可能在该事务提交之后完成，但它是包含该事务的协议的一部分。

的对偶签名已经包含了价格信息。通过在m_4中添加M对该消息的签名，并将其转发给G，M又向该事务提交了它自己。再说明一次，这一消息仍是M投票提交的。

当G从M处收到双重签名的消息（在SET的步骤4中）后，它便知道双方都已准备好提交该事务的内容。与SET一样，如果G对C提供的信用卡信息满意，它便提交该事务，持久地存储m_3、m_4和对偶签名，并给M发送一条已验证消息。M则向C发送一条包含$K_{C,M}$的事务完成消息，于是，C就可以解读其货物。

该协议是货物原子化的，其原因如下：

- 如果在G提交该事务之前出现了故障，那么不会有任何资金传递，而C也不会得到此货物，因为它没有获得$K_{C,M}$。
- 如果在提交之后出现故障，那么资金将会传递，G有一个（持久的）$K_{C,M}$的拷贝，而C有该货物的一份加密的拷贝。C可以从M处得到$K_{C,M}$，或者（如果因为某些原因M没有发送$K_{C,M}$）从G处得到$K_{C,M}$，因为G知道该事务已经提交，而C的证书也证实了它创建该对偶签名的主角的身份。

该协议的一个重要的特点是，当该事务提交时，G便拥有了$K_{C,M}$的一份拷贝，因此即便M“忘记”了发送包含$K_{C,M}$的事务完成消息，C仍然可以解密其货物。

1. 已认证交付

因特网上货物交付的另一个问题是**已认证交付**（certified delivery）。一个货物原子化的事务确保了对顾客的交付，但是我们还想要有附加的保证：交付的是正确的货物。一旦出现发送的货物和先前达成共识的规格说明不符，M如何为它自身辩白？C又如何肯定接收到的就是订购的货物呢？尤其是，如果在M和C之间出现关于所交付货物的争执，而这个争执将由另一位仲裁者来解决，那么M和C如何将各自的情况向仲裁者表达呢？例如：

- 假设在解密所交付货物后，C发现它们不符合其规格说明，并想要回她的钱。C可以向仲裁者阐明该软件确实无法工作，但是她如何表明该软件就是M发送的软件呢？
- 假设C企图欺骗M，而她展示给仲裁者的无法使用的软件并不是M所发送的。那么，M又如何来揭露这一欺诈的企图呢？

加强型的SET协议符合已认证交付的要求。不妨回忆一下，G在该事务提交的时候持久地储存了m_3、m_4和对偶签名。由C创建的对偶签名涵盖了包含在m_2中的货物的规格说明。通过将该签名转发给G，M便可确认该规格说明是准确的（因为它可以通过解读m_2来审核此规格说明，并可以检验其签名是涵盖m_2的）。

- 通过运行针对该货物的摘要函数，并将其结果和G存储于m_4中的摘要进行比较，C就可以向仲裁者展示她声称从M处接收到的加密货物实际上就是M所发送的货物。仲裁者然后就可以利用$K_{C,M}$来解读此货物，并对它们进行测试。C还可以提供包含此规格说明的m_2，并展示其对偶签名是涵盖m_2的，因此也是经M同意的。仲裁者现在就可以对C的声明作出裁决了。
- M也可以反对声明（所收到的货物不符合它们的规格说明），为自己作辩护，因为M也能够产生m_2。仲裁者可以再一次确认该货物并没有被篡改，然后针对其规格说明来测试它们。

2. 托管服务器

另一种要求货物原子性的应用程序是某个陌生人或某个拍卖网站通过因特网对实际(非电子的)货物的采购。该货物不能被下载，但必须通过某个装运代理发送。该事务的某一个参与者可能怀疑另一方会不遵守此采购协议议定的条件。双方如何确定该事务是货物原子化的，以及当且仅当已付货款时，如何确定货物已交付呢?

符合这些需求的一种实现方法是采用某个称之为**托管代理**（escrow agent）的可信任第三方。有很多公司都从事因特网上的托管代理的业务。其基本思路是，在顾客和零售商就采购事项达成一致之后，顾客支付给托管代理而不是支付给零售商，由托管代理持有这些资金，直到货物被交付并且顾客收到货物为止。此后，托管代理才将此支付款转交该零售商。

我们将介绍一种简化的托管代理协议形式[⊖]，其中不考虑很多细节，包括认证、加密等等。该协议涉及一个客户C、一个零售商M和一个托管代理E。在C和M就采购事项达成一致之后，该协议开始。

1) C可以采用在本章中描述过的某种安全付款方法向E发送已达成共识的货款。

2) E持久地存储货款以及该协议其余部分所需要的其他信息，并提交该事务。

3) E通知M，C已经付款。

4) M通过某些可追踪的——同意向E提供其运输的状态消息（包括交付的确认）的——货运代理（如FedEX或UPS），向C发送其货物。

5) 一旦C收到货物，他便对货物进行检测，看它们是否符合订单的要求。这种检测必须在规定的时间内完成，这段时间从货物交付时开始，由货运代理的追踪机制记录。

a. 如果C对货物满意，则发生以下情况:

i. C通知E货物已经收到并且满意所收到的货物。

ii. E将货款转交给M。

b. 如果C不满意此货物，则将发生以下情况:

i. C通知E货物已经收到，但是不满意所收到的货物。

ii. C通过某个可追踪货运代理将货物返回给M。

iii. M接收到货物，并通知E[⊜]。

iv. E将货款返回给C。

c. 如果在检测时间结束后，C并没有通知E他是否满意货物，那么E将货款转给M[⊜]。

该协议基本上可以实现货物原子性和已认证交付。例如，C有一个规定的检查时间来确定所交付的货物是否是其订购的货物。与原先的货物原子性协议一样，这个协议也是依靠某个受C和M都信任的第三方，在该事务提交之后完成某个规定的操作。

27.7.4 电子现金：盲签名

我们迄今所讨论的因特网采购事务都是使用某种信用卡，信用卡（包括支票）是**符号货**

⊖ 我们所描述的协议是建立在i-Escrow协议[i-Escrow 2000]基础上的。

⊜ M可以检测返回的货物，以查看它们是否是所发送的，并且必须在某个规定的检查时间内完成其检测。我们忽略了若M对返回的货物不满意时会发生的情况。

⊜ 在步骤4中M可能没有发送货物给C，也有可能在步骤5bii中C没有将货物返回给E。再说明一次，这些情况可以通过货运代理的追踪机制来解决。

币（notational money）的一个例子。也就是说，你实际的财富是通过银行账户中的余额来表示的；你的信用卡（或者支票）就是相对于那些财富的一种符号。当进行某项采购的时候，你提供了一个符号来标识你和你正在采购的货物的价值。零售商相信你会遵循该采购协定，并最终会用实际的货币支付这次采购（在你的支票账户中有足够的资金，或者你在信用卡公司具有良好的信誉）。无论是哪一种情况，你的银行余额最终会减少，以反映出你的采购。

与符号化货币不同，现金是由政府支持的。虽然它并没有真实的价值（毕竟，那只是一张纸而已）；但是公众对政府的稳定性的信任会导致它好像是有真实价值那样（例如，仿佛它是金子一样）。因此，零售商知道，他完全可以将现金存入其账户里，或者用它来采购其他的货物，而无需信任该顾客。现金通常称作**令牌货币**（token money）。在因特网世界里，令牌货币就是**电子化**（electronic）或**数字化**（digital）的现金。

令牌货币向某个事务的参与者提供了超越符号化货币的优势。

- *匿名性*　由于不需要顾客提供任何签名记录来完成某项现金事务，所以事务可以匿名地进行。银行和信用卡公司都不知道该顾客的身份。相反，信用卡公司则保存着所有顾客的采购记录，而银行也可以访问作废的支票。在稍后的某个时间，这些记录可能用于政府、某个法庭程序等方面。
- *小额采购*　对于每一项信用卡事务，信用卡公司要在固定费用的基础上根据采购金额加收一定的手续费用。因此，信用卡对于那些只涉及少量金额的采购是不合适的，但仍有些因特网供应商会为提供给浏览者的一页消息而收取几分钱。小额的电子现金对于这样的事务很有用处。

因此有必要支持建立在电子现金基础上的事务。这样的事务应当满足**货币原子性**（money atomicity）的需求：货币不应该被创造或销毁。然而，由于电子现金在系统中是通过某个数据结构来代表的，所以可能存在几种违反货币原子性的情况。例如：

- 一个不诚实的客户或零售商可能会制作该数据结构的一份拷贝，并同时使用原始的数据结构和拷贝。
- 若出现故障（例如，丢失了某个消息或者系统崩溃了），则货币就可能被创造或销毁。例如，一个将数据结构拷贝发送给零售商的顾客，无法确认零售商是否接收到这条消息，所以决定在另一次采购中重复使用该数据结构——尽管实际上已收到资金。另外，有可能消息还没有接收到，但该客户已不再使用此数据结构了，因为顾客并没有意识到此项付款尚未完成。

下面将讨论用来支持任意（未必是电子化）货物采购的电子化现金协议。货物原子性并不是这种协议的一个特点。相反地，顾客相信零售商会在事务提交之后发送货物。

1. 令牌与冗余谓词

本协议是建立在Ecash协议[Chaum et al. 1988]基础上的。现金用各种单位的电子令牌来代表。每张令牌都由用银行才知道的某个私钥加过密的唯一序列号n组成。这里的术语容易令人困惑，因为我们经常说到，在电子化现金协议中，银行通过“签署”序列号来创建令牌。这里的签名的意思与27.3节中签名的意思是不一样的，后一种签名由被签名项紧随一个对其摘要的加密项组成。在本节中，当我们说银行签署了某个序列号时，意思是指银行用某个私钥

加密了此序列号，而产生的结果则是一张令牌。

这种方案如何防止入侵者创建伪造令牌呢？毕竟，令牌只不过是具有一定长度的比特串而已。该比特串是对某个序列号进行加密而得到的事实，并不能提供这种保护。你可能会认为，我们可以通过用银行的公钥对它解密，并检验结果的方法来测试该令牌的真实性。然而，那个密钥可能会应用到任何有效的或无效的令牌，产生比特串，而我们没有任何办法来辨别某个比特串是有效的序列号还是无效的序列号。

为了防止入侵者创建伪造的令牌，我们采用一种技术，它要求将序列号变成具有某种特点的比特串，从而使得它们可以和任意的比特串区分开来。例如，它可能要求序列号的前半部分应该是随机地创建的，而后半部分则应该是前半部分利用某种固定的、已知的编码函数来编码的形式。形式上，我们说，存在一些众所周知的谓词，称为**冗余谓词**（redundancy predicate）*valid*，使得对于所有有效的序列号n，其谓词*valid*(n)都为真。

尽管我们假设伪造者也知道冗余谓词和银行解密令牌的公钥，但他并不知道银行的私钥，所以无法通过加密某个有效的序列号来产生出令牌。因此，在伪造令牌时，他面临着需要寻找到一张（假的）令牌，使其解密得到的比特串能满足*valid*的问题。伪造者可以采用尝试-出错技术来完成这一工作，但是解密一个任意选择的比特串，其结果会满足*valid*的可能性是极小的，若该比特串足够长，或者*valid*使得满足*valid*的比特串仅占所有该长度比特串总数的很小一部分的话。

在创建令牌的方案中，银行保持有一系列的公钥/私钥对，并选择满足*valid*的序列号。它在该系列中，用同一把私钥 K_j^{priv} 来签署所有用于创建某个特定单位j的序列号，而对于不同的单位则采用不同的私钥。银行并不持有其创建令牌的序列号的列表，然而，如果数量足够大，并对每种单位允许铸造的令牌数量有所限制的话，那么银行会两次选择相同的序列号的可能性就会缩到很小。银行持有所有已储存令牌的序列号的一份列表，因此可以拒绝任何已储存的令牌的拷贝。通过这种方法，它就可以侦测到试图使用某个复制令牌的企图。任何客户或零售商，都可以通过用 K_j^{pub} 解密，并对其结果应用*valid*的方法，来检查某张令牌的真实性及其单位。

2. 简单数字化现金协议

如果匿名性不成问题，那么客户C、零售商M和银行B就可以采用如下的简单数字化现金协议。

(1) 创建令牌

1) C向B认证他的身份，并发送一条消息，请求以令牌形式从C的账户提取一定数额的现金。

2) B在C的账户上记入借方，然后通过建立某个序列号n_i来铸造所要求的令牌，对于每一张令牌，使得*valid*(n_i)为真。然后，与该令牌的单位j相对应，用私钥 K_j^{priv} 来加密n_i，从而产生令牌 $K_j^{priv}[n_i]$。

3) B向C发送用会话密钥（为供B和C在通常情况下使用而生成的）加密的令牌。任何令牌都不能以未加密形式发送，因为每个入侵者都可能会拷贝它，并在C尚未消费原始的令牌之前就消费掉该拷贝。（在这种情况下，当C在后来存款时，原始的令牌反而会遭到B的拒绝。）这时，提交此创建令牌事务。

4) C接收到此令牌，并将其储存在他“电子钱包”里。

(2) 消费令牌

1) 当C想要采用他的一些令牌从M处采购货物的时候，他先与M建立一次会话，并按常规方式产生一把会话密钥。然后他向M发送包含货物的采购订单和相应的令牌号码的一条消息，所有的通信都用会话密钥进行了加密。

2) 当接收到了来自于C的消息，M解读此令牌并检查它们是否有效，是否足够用于采购所要求的货物。M然后用会话密钥加密此令牌后转发给B。

3) 当收到来自于M的消息后，B解读此令牌并检查它们是否有效。然后它检查其所存储的令牌列表，以此来确定所接收到的令牌尚未存储过。然后B将刚才收到的令牌添加到其存储的令牌列表中，对M账户增加此令牌之金额数，提交该事务并向M发送一条完成消息。

4) 一旦收到B的完成消息，M便执行本地提交操作，然后向C发送一条完成消息（及已采购的货物）。

该协议并未确保货物原子性或已认证交付。M可能并没有向C发送货物。

正如SET协议一样，在B、M和C之间的消息交换可以看成是一种线性提交协议。

3. 匿名协议和盲函数

简单数字化现金协议并未向客户提供匿名性，因为银行能够记录顾客取款用的令牌的序列号。当那些令牌被某个零售商存储时，银行可以得出结论：该零售商卖给了那个顾客一些货物。这就透露出了有关那个顾客的行动的一些消息，而他本人可能并不想让人知道这些事情。为了提供匿名性，协议改为由顾客（而不是银行）来产生序列号n，使之满足$valid(n)$，对它编码，然后将它提交给银行。银行采用某个对应于所取款的令牌单位的私钥，签署已编码的序列号(它并不知道此序列号是什么)，创建令牌。这样一种签名称为**盲签名**（blind signature）[Chaum et al. 1988]。银行并不知道该序列号，所以当该令牌稍后再由零售商存储时，它并不能回溯出其顾客。当顾客从银行收到盲令牌时，便将它还原，以获得原令牌。

为了实现盲签名，该协议采用**盲函数**（blinding function）b，它有时候称为兑换函数(commuting function)。函数b和它的逆函数b^{-1}有如下两个性质。

- 已知$b(n)$是很难确定n的。
- b可与银行所采用的——涉及（秘密的）单位密钥 K_j^{priv} 的——加密函数进行兑换。即

$$K_j^{priv}[b(n)] = b\left(K_j^{priv}[n]\right)$$

作为结果，有

$$b^{-1}\left(K_j^{priv}[b(n)]\right) = b^{-1}\left(b\left(K_j^{priv}[n]\right)\right) = K_j^{priv}[n]$$

因此，C可以从盲令牌中复原出原令牌。

创建令牌

1) C创建一个满足$valid(n)$的有效序列号n。

2) C选择一个盲函数b（仅有C知道），并通过计算$b(n)$来**盲化**（blind）此序列号。

3) C向B认证自己的身份，并发送一条包含$b(n)$的消息，请求从他的账户中以令牌的形式提取一定数额的现金。(和在简单数字化现金协议中一样，该消息是用会话密钥加密了的。)

由于B不知道盲函数，所以它无法确定n。

4) 对应于该令牌的单位，B用私钥 K_j^{priv} 来签署$b(n)$，创建盲令牌 $K_j^{priv}[b(n)]$。它相应地向C的账户记入借方，并再次采用会话密钥向C返回该盲令牌。尽管B无法检验C确实选出了一个有效的序列号（满足$valid(n)$），但C没有任何理由去创建一个无效的号码，因为C知道，当他试图消费它的时候，B会将该令牌的金额数记入他账户的借方，而爱令牌的有效性也将会受到检验。这时，提交该令牌创建事务。

5) C通过使用 $b^{-1}\left(K_j^{priv}[b(n)]\right)$ 复明（unblind）盲令牌，以获得 $K_j^{priv}[n]$，这就是其请求的由某个已签名的有效序列号所组成的令牌。

4. 创建盲函数

协议要求C创建他自己的（B不知道的）盲函数b。这似乎是一项很困难的任务，但对于公钥加密来说，这很容易在RSA算法语境下实现。在完成此任务的一个方案中，C首先产生某个随机数u，它与银行密钥的模N是互素的[⊖]（参见27.2节）。因为u与N是互素的，所以它相对于N有一个**乘性逆**（multiplicative inverse）u^{-1}，使得

$$u * u^{-1} \equiv 1 \,(mod\, N)$$

为了盲化序列号n，C计算

$$K_j^{pub}[u] * n \,(mod\, N)$$

并将其结果发送给B。因此，盲函数可以被简单地看成乘以某个随机数。

由B返回给C的已签名的结果sr是

$$sr = K_j^{pri}\left[K_j^{pub}[u] * n\right]$$

根据式(27.1)，可推出

$$sr = u * K_j^{pri}[n] \,(mod\, N)$$

通俗地讲，我们可以把复明令牌C说成是“用u去除sr”，但实际上，C是通过乘以其乘性逆来复明该令牌的:

$$K_j^{pri}[n] = u^{-1} * sr \,(mod\, N)$$

现在就可以通过 K_j^{pub} 获得序列号n。

5. 消费令牌

消费和验证令牌的协议涉及和先前叙述的简单协议一样的处理步骤。幸运的是，我们不必假设B持有其所产生令牌的系列号的一份列表，因为在匿名协议中，它确实不知道这些序列号是什么。当M向B提交该令牌要求履约时，B总是简单地假定如果该序列号满足$valid$，那么它较快地（盲目地）签署那个序列号。与以前一样，B持有已存储令牌的序列号的一份列表，所以它不会重复接收同样的令牌。

6. 货币原子性

问题是该电子化现金协议是否达到了货币原子性。货币原子性有两个方面:

1) 若某个进程能够产生某个令牌的一份拷贝，并随后消费它的话，货币是有可能（在任

⊖ Euclid的算法可以用来测试该随机数是否与N互素(见[Stalling 1997])。

何事务之外）创建出来的。然而，当银行检测出序列号与其先前提交的令牌的列表不相符时，就会发现这一欺骗的企图。创建货币的另一种方法是伪造，但是，就像我们先前所看到的一样，这种方法也是不可能成功的。

2) 货币也有可能因为故障而销毁，但是这样的问题一般也是可以处理的。

- 在令牌生成事务中，银行会记入该客户账户的借方，发送此令牌，然后提交。但通信系统则丢失了此令牌，并且它从不交付给顾客。不过，这些协议有一个有趣的性质，如果顾客声称从未收到过银行发送的某个令牌，那么银行可以很简单地向此顾客发送它从其日志里复原的（盲）令牌的一份拷贝。即便该顾客是不诚实的，他现在拥有该令牌的两份拷贝，但是他只能够消费一个。
- 在令牌消费事务中，当某顾客发送了令牌、但尚未接收到该事务提交消息时，系统出现了崩溃。顾客并不知道事务是否在崩溃之前提交，因此也不知道该令牌是否实际消费了。倘若顾客试图再次消费令牌时，就可能被怀疑欺诈。然而，顾客可以稍后询问银行，是否已消费了某个特定的令牌（即在已消费令牌的列表中），但是这样会破坏该顾客的匿名性。

27.8 参考书目

[Stallings 1999]中涵盖了本章的绝大部分内容。公钥加密的概念是由[Diffie and Hellman 1976]首先提出的，但几乎所有的公钥加密系统都是建立在RSA算法[Rivest et al. 1978]基础上的。[Schneier 1995]对RSA算法以及其他许多加密算法与协议的数学基础进行了详细的描述。与公钥加密类似，数字签名的概念也是在[Diffie and Hellman 1976]中引进的，但虽然大多数数字签名系统都是建立在RSA算法[Rivest et al. 1978]基础上的，但还是有些系统是建立在专门为数字签名而开发的其他算法基础上的，它们不适合用于加密。Kerberos系统是在[Steiner et al. 1988]和[Neuman and Ts'o 1994]里讨论的，它形成了DCE[Hu 1995]中提供的安全性的基础。NetBill系统是在[Cox et al. 1995]里介绍的。[Chaum et al. 1988]介绍了Ecash协议和盲签名。SSL和SET协议的描述可在Web的相关网站上找到，按照公布时间，SSL见[Netscape 2000]，SET见[VISA 2000]。托管代理协议是建立在通过eBay提供的i-Escrow系统基础上的，按照公布时间，在其Web网站[i-Escrow 2000]上有描述。

27.9 练习

27.1　请讨论在因特网上运行事务时所涉及的安全性问题。

27.2　任何采用计算机键盘的人，应当都能解决下述简单的替换密码:

Rsvj ;ryyrt od vjsmhrf yp yjr pmr pm oyd tohjy pm yjr lrunpstf/

27.3　请说明，为什么一般短密钥的安全性低于长密钥?

27.4　为什么在Kerberos协议里，从KS向C发送的消息（即消息M2）里必须包含S? 并请描述，倘若不包含S，可能被I利用的一种攻击。

27.5　请说明，如何在安全性协议里用时间戳来防止某个回放攻击。

27.6　请说明，在对短消息或者对字段内包含着入侵者可能熟悉的明文进行加密时，如何利用临时串来提高安全性。

27.7　在采用公钥加密的系统里，站点B打算通过假冒A来愚弄C。B等待着，直到A请求与B通信为止。

A是通过向B发送一条“我想要通信”的消息而做到这点的，该消息指出其名字(A)，并采用B的公钥加密。然后B突然提出陷阱。他向C发送一条“我想要通信”的消息，声称他是A，并采用C的公钥进行加密。为了弄清是否确实在与A通信，C (向B)回复了一个通过对某个大随机数N用A的公钥加密而获得的一段消息。如果C得到的是包含着用C的公钥加密的数N+1的消息，他就会得出对方为A的结论，因为只有A才能解读C发送的消息。C确实得到了这样的响应。然而，这是错误的，因为响应来自B。请说明，这是怎么发生的？（提示：如果每个消息的密文都包含发送者的名字的话，该协议就可以被校正。）

27.8 假设入侵者获得了某零售商证书的一份拷贝。

a. 请说明，为什么该入侵者不能轻易地利用该证书，并冒充他是零售商？

b. 请说明，为什么入侵者不能用自己的公钥取代零售商在证书里的公钥？

27.9 假设你采用了SSL协议，并与某零售商站点M相连接。该站点向你发送M的证书。当SSL协议完成时，如何才能确定新的会话密钥只有M知道（也许是某个入侵者向你发送的M证书的拷贝）？如何能确保确实连接到了M？

27.10 利用你的本地因特网浏览器：

a. 请描述，当你与某个站点相连接时，如何告知对方正在采用SSL协议？

b. 假设你已经与一个采用SSL的站点相连接。请描述，如何确定提供该站点证书的CA的名字？

c. 你的浏览器为SSL加密所用的密钥是多少位的？

27.11 假设你得到了自己的证书。请说明，如何采用该证书来处理某个入侵者可能偷走了你的信用卡号码的情况？

27.12 假设某个入侵者将病毒放到电脑里，从而改变了你的浏览器设置。请描述，该入侵者可能冒充你打算与之通信的某台服务器站点S（哪怕你采用了SSL协议）并获得你的信用卡号码的两个不同办法。

27.13 一个采用SSL协议（不带SET）的零售商可以通过如下过程实现信用卡交易：客户采购某项物品，而零售商要求他用在SSL协议里创建的会话密钥将其信用卡号码加密后发送过来。当零售商接收到此信息后，他便与信用卡公司开始一个单独的事务，以验证该次采购。当该事务需要提交时，零售商就与顾客一起提交该事务。请说明，这个协议与SET之间的异同。

27.14 假设SET协议里的零售商是不诚实的。请说明，他为什么无法欺骗顾客？

27.15 请说明，为什么在已认证交付协议中需要采用某个可信任的第三方？

27.16 请描述零售商电脑可以用来处理SET协议中出现崩溃情况的重启动处理步骤。

27.17 请说明，为什么SET协议中的MD_2（见27.7.2节）必须是对偶签名的一部分？

27.18 请说明，为什么伪造者不能轻易地提交一个任意随机数，将它作为电子化现金协议的令牌？

27.19 假设在匿名电子化现金协议里的银行是诚实的，但客户和零售商不一定诚实。

a. 在接收到银行的令牌后，顾客声称他从未收到过此令牌。请说明，银行应当如何做？为什么这样做是正确的？

b. 在从顾客处接收到包含采购订单和客户令牌的消息之后，零售商声称从未接收到此类消息。请说明，客户应当如何做？为什么？

27.20 请描述，以下的每个协议为防止回放攻击所采用的方法：

a. Kerberos认证协议

b. SET协议

c. 电子化现金协议

附　录

附录A 关于系统的问题

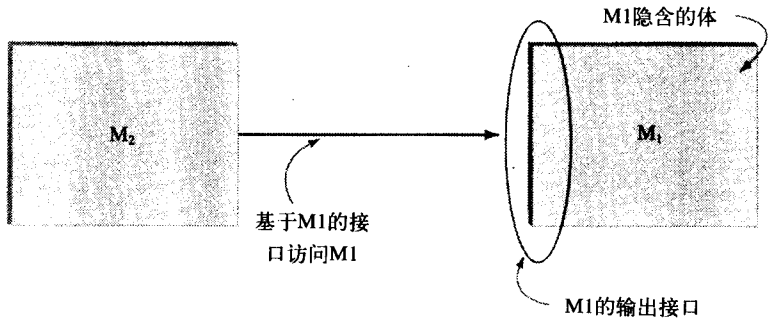
A.1 基本系统问题

事务处理系统既可以在单台计算机上实现，也可以在许多地理上广阔分布的计算机上实现。事务既可以访问同一台计算机上的单个（本地）数据库，也可以访问另外半球上的许多（远程）数据库。但是无论如何，事务都必须支持ACID性质。在这一节中，我们回顾设计这类系统的一些基本概念和机制。

A.1.1 模块和对象

多数大型系统都采用“分而治之”的方法进行设计。系统被分解为一些简单的、可以单独实现与调试的模块。支持模块结构的程序设计语言允许程序员指定模块接口和模块体。模块M1的**接口**（interface）包含M2模块访问M1所需要的全部信息。它包括M1中可以被M2调用的过程名，以及它们的参数和类型，上述内容组成了调用语句的语法（有时也称为过程的**签名**（signature））。接口还应该包含语义信息：每个过程作用的描述（可能是文字的）。

接口可作为模块的实现者和它的用户之间的一个契约。实现者保证模块符合接口中包含的规范。当M1为应用级的程序提供服务时，它的接口通常称为**应用编程接口**（Application Programming Interface，API）。图A-1给出了这种情形。



图A-1 模块结构

接口也包含向调用者返回错误情况的机制的描述。这些情况在事务处理系统中频繁发生。例如，当M2调用M1时，M1可能还没有初始化，或提供给M1的自变量发生错误。

模块**体**（body）包含可以被其他模块调用的过程，仅供内部使用的过程，以及变量声明。M1的接口可以在模块体的实现和存储入库之前进行设计。M1的接口规范与模块体的实现分离所带来的便利是，调用M1的模块可以仅根据该接口进行设计和实现（甚至部分调试）。这样，M1和M2的实现就可以并行进行。

每个模块的体对其他模块来说是隐藏的。有时我们也说过程和变量的声明**封装**（encapsulated）到模块中。所以，M1的模块体可以随时重新实现，只要它的接口仍保持不变，M2就无需改变。这种性质通常称为**实现透明性**（implementation transparency）。模块体中的变量声明对于M1的过程是全局性的。与局部变量不同，全局变量具有持久性，并在不同的调用间保持它们各自的值。全局变量的值构成模块的**状态**（state）。

实现的透明性意味着模块的用户无需关注模块提供的服务是如何实现的。而且，模块的用户不能有意或无意地破坏模块体中声明的变量的完整性，因为它们不能直接访问这些变量。用户通过调用接口过程可以间接影响这些变量的状态，而那些过程会保持数据的完整性。

模块的思想可以进一步改进，从而产生对象的概念。**对象**（object）是一组应用中有意义的某个实体的计算机表示，例如，一个下压堆栈，或表单上由图片表示的一个按钮，该按钮可作为事务处理系统的图形界面的一个组成部分。

对象由一组**属性**（attribute）和**方法**（method）来刻画，这些属性的值组成了对象的状态，而在对象接口中描述的这些方法是一些能够被其他模块调用的过程。一个按钮的属性可能包含它的尺寸、位置和颜色。与按钮对象相关的方法可能包含在屏幕上绘制按钮的图形表示的过程，以及当用户按下按钮后改变按钮表示的过程。过程和属性编码的数据结构都封装在对象内部。

一些属性是**公共的**（public），并且可以从对象外部通过常用的**赋值**语句进行访问。例如，一个时钟对象可能产生可以直接被外部程序读取的存储当前时间的变量。另外一些属性则是**私有的**（private），并且只能被公共对象方法访问。例如，按钮对象拥有一组与它的图形显示有关的私有属性，这些属性被绘制在屏幕上显示按钮的方法所使用。由于用户不能访问这些私有属性，所以显示的完整性就不会遭到破坏。

一个给定的应用可能包含许多相似的对象，如按钮，这些对象具有同样的属性和方法。它们是**对象类**（object class），或**类**（class）的实例。所以，可能存在一个具有上述的属性和方法的按钮类。某个按钮对象是按钮类的一个**实例**（instance）。与传统的程序设计语言相比，对象类就像数据类型，类的对象就像该类型的变量。

支持对象的语言也支持**继承**（inheritance）。当对象类在系统中实现后（可能由系统开发商提供），应用程序员可以实现一个新的对象类，它是原始对象类的一个**子类**（child）。子类继承其父类的所有属性和方法，并且可以有自己另外的属性和方法。

例如，按钮类的子类可能是一个开关按钮类，当按下按钮后，按钮的颜色在红、绿之间切换产生不同的效果。由于开关按钮自动继承了其父类的属性和方法，因此设计者可以使用继承的方法来绘制和改变开关按钮的颜色，并不需要理解私有的图形属性或其他维护图形显示完整性的细节。所以，继承提供了一种手段，使得应用程序员可以轻松地为特定应用定制

通用对象类。

对象可以视为对**抽象数据类型**（abstract data type）的程序语言概念的推广。通俗地说，抽象数据类型是没有继承性的对象。一个例子是内置在计算机硬件上的整数数据类型。整数具有值（它的属性）和一些操作：加法、减法、乘法以及除法（它的方法）。它之所以被认为是抽象数据类型，是因为它的实现（一进制补码、二进制补码等）被封装了，对程序员不可见。大部分程序设计语言提供了其他的内置抽象数据类型，这些数据类型不在硬件中实现，如数组和记录。再强调一次，这些类型的实现是封装起来的，对于使用它们的程序员而言是不可见的。一些程序设计语言允许程序员定义自己的抽象数据类型，例如，队列。这只是实现程序员定义自己的对象过程中在编程语言的概念上取得的一个小进步，即具有继承性的抽象数据类型，例如，作为队列的子类的优先级队列。

A.1.2 客户与服务器

事务处理系统中的一类重要模块是**资源管理器**（resource manager）。我们在广义上使用“资源”这个词来描述系统中硬件或数据，这些硬件或数据可以被许多模块所利用，例如，数据库，含有注册标识符和相应口令的注册文件，不重复数字源（唯一数产生器），或打印设备。“资源管理器”是调节资源使用的模块。它封装资源以及相关的数据结构和一组向其他模块进行输出的过程。这些过程是希望访问这些资源的其他模块可获取的方法或抽象操作。只有通过这些过程才能访问资源。

例如，管理数据库的资源管理器（称为**数据库管理器**（database manager）或数据库管理系统（DBMS））是允许访问含有数据库的大容量存储器上区域的唯一模块。另外，可能在它的地址空间中含有缓冲区来放置最近访问过的页，这部分内存称为**高速缓存**（cache memory）。它可能提供执行SQL语句的服务。注册管理器可能封装注册文件并实现注册新用户、变更口令、检查口令的服务。资源管理器的接口在API中描述。

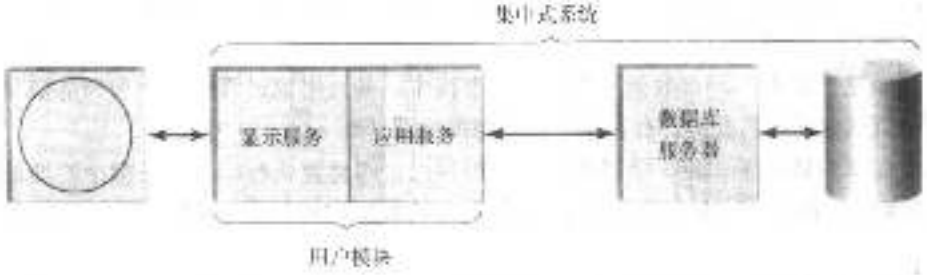
资源管理器为其他正在同步执行的模块提供服务。所以，当资源管理器RM正为某个模块M_1提供服务时，另一模块M_2，可能也要求RM执行某些服务。在最简单的情形下，服务按先进先出的方式提供：M_2的服务请求要一直等到由M_1调用的过程返回以后才能开始执行。然而，这种规定可能在大型系统中引发性能问题，因为在大型系统中，其他许多模块都需要RM提供的服务。顺序执行事务可能引起同样的问题：RM会成为瓶颈。允许并发执行资源管理器的过程可以解决上诉问题。**线程**（thread）是这种机制的一个例子。我们在A.3节讨论线程的细节时再讨论这问题。

资源管理器通常被称为**服务器**（server）。所以，我们称管理数据库的模块为数据库管理器或则数据库服务器。调用服务器的模块称为**客户**（client）。这些名字可能会令人有些误解，因为模块通常按一定的层次结构进行组织。为一个模块担当服务器的模块可能用到其他模块的服务，在这种情形下，该模块又成为了那些其他模块的客户。

事务处理系统通常组织成客户/服务器系统。事务被客户模块调用，当它们需要访问某些资源时就调用相应的服务器模块。

考虑图A-2所示的简单的事务处理系统。用户模块由一个控制屏幕上的信息显示的显示服

务器和一个执行应用程序的应用服务器组成。数据库服务器为应用程序执行SQL语句。



图A-2 简单的事务处理系统

我们知道，术语**客户**（client）和**服务器**（server）经常用在不同的语境中。这里，我们用这两个术语表示存在于同一计算机、或者不同计算机上的模块。在其他的语境下，这些术语可能指执行这些模块的计算机。所以，可以说我们的学生注册系统就是一个客户/服务器系统，其中客户是学生使用的计算机，服务器是注册办公室的主机。在这种情形下，仅当客户和服务器模块驻留在不同的计算机上时，事务处理系统才是一个客户/服务器系统。反之，客户与服务器模块都在同一计算机上的系统不能被称为客户/服务器系统。为了区分这两种用法，我们使用术语"客户"和"服务器"来指示我们已经描述过的模块结构。当提到硬件时，我们使用客户机与服务器机。注意，客户/服务器结构的一个特点（也是它的优势之一）是它可以轻松地在单台机器上或计算机网络上实现。

A.2 多道程序的操作系统

现代计算机系统包含多个独立的可同时执行的处理器。例如，中央处理器（CPU）和I/O处理器（有时称为信道）。**单处理器系统**（uniprocessor system）有一个CPU，而**多处理器系统**（multiprocessor system）有多个CPU。传统的多处理器系统的所有处理器共享一个公共主存，所以它也称为**共享内存系统**（shared memory system）。这种体系结构的一个变体是其中的每一个CPU都有自己的私有主存并且仅共享大容量存储器。这种系统称为**磁盘共享系统**（shared disk system）。

还有一种**无共享系统**（shared nothing system），即不同的计算机系统通过通信线路互连起来。每一个系统都是设备齐全的，它们具有自己的私有主存和大容量存储设备，无任何共享。这种体系结构的一类变体是，同类的个体系统被包装在同一个单元中。每一个系统都构建在同样的硬件上，被同一操作系统的拷贝控制，并且通过高速连接（如，总线）进行通信。在另一类变体中，个体系统是一个**分布式系统**（distributed system）的一部分。分布式系统一般包括异构硬件和软件，它们可能是地理上分布的，进行低速通信（可能通过电话线路）。我们在下一节会讨论这样的系统。

操作系统（operating system）是管理计算机系统资源的程序。这些资源包含在系统处理器上的执行时间、存储空间、对物理设备（如大容量的存储设备或通信线路）的访问。操作系统也可能管理信息资源，如文件系统。操作系统的目标之一是高效地使用这些资源。

由操作系统控制的用户与系统任务以进程方式执行。**进程**（process）是可以存取某些数

据的正在执行的程序。存储在程序的虚拟内存中的数据可以直接被程序访问，数据也可以存储在文件中，通过操作系统访问。另外，数据可以是全局性的或局部性的。全局性数据能够被进程执行的所有过程访问，当特定的过程开始执行时可以动态生成局部数据，并且当过程返回时释放这些数据。局部数据一般存储在堆栈中，并且组成过程的参数和局部变量。程序和所有可以被它访问的数据都存在于进程的**数据空间**（data space）中。

在最简单的情形下，每一个进程有一个控制点，即将要执行的下一条指令的地址（我们在A.3节考察具有多个控制点的进程）。控制点与其数据空间中的变量的值合称为进程的**状态**（state）。

在**单道程序**（uniprogrammed）的操作系统中，一个进程在其他进程开始之前可以一直运行到结束。这不能充分利用系统资源，因为许多程序被设计成一次只能利用一个物理处理器。例如，程序在计算（请求CPU）和与文件交换信息（请求I/O处理器）之间进行轮换。在计算阶段，I/O设备空闲，而在I/O阶段，CPU空闲。

有效地利用计算机系统要求使尽可能多的物理CPU处于忙的状态。为了达到目标，理想的方法是同时执行多个进程。进程在一个特定的时间点可能仅从一个（或者，可能一小部分）处理器请求服务。但是，如果许多进程同时执行，就可以同时利用许多处理器，从而提高系统的吞吐量。能够同时执行多个进程的操作系统被称为**多道程序**（multiprogrammed）的操作系统。

从用户观点看，并发执行是重要的，因为为一个用户提供服务的进程不需要等到为另一个用户提供服务的进程结束后才能开始，就如同在单道程序系统中一样。结果是改进了系统的响应时间。

并发在同时为数千个用户服务的事务处理系统中特别重要。通过事务的并发执行，计算机系统固有的能力可以在减少单个事务的响应时间，同时提高系统的吞吐量。

事实上，尽管进程共享那些由操作系统分配的现有资源（包括CPU时间、存储空间、I/O处理器的服务请求等），但在多道程序操作系统控制下执行的每个进程看起来就像在自己的机器上执行一样。操作系统允许一个进程P_1执行一段时间，称为**时间片**（time quantum），然后，中断P_1让另一个进程P_2执行，从而促使CPU被共享。P_1可以在随后的时间继续执行。这产生了一种称为**交叉执行**（interleaved execution）的现象。在多处理器系统中，每个CPU都按这种方式共享。

如果P_1和P_2的数据空间是不相交的，那么它们的执行就不会受到交叉执行的影响。P_1分辨不出它已经被中断了，因为当它继续执行时，它的数据空间的状态与其被中断前的状态是完全一致的[⊖]。数据空间并不一定是不相交的，然而，可能出现交叠。一种极端的情形是，数据空间的交叠可能被限制在大容量存储器的文件上。但是操作系统也允许处理虚拟内存中的共享部分。任何情形下，操作系统负责确保每一个进程只能够访问自己数据库空间中的信息。

如果P_1和P_2的数据空间不是不相交的，那么这些进程**共享**访问交集中的数据项。如果P_2中断了P_1，并且改变了交集中的一个数据项的值，则P_1被中断时的数据空间的状态与其随后继续执行时的状态将不完全一致，所以P_1的执行可能受到交叉执行的影响。例如，如果P_2向P_1读取

⊖ 这里我们假设进程不访问系统时钟。如果能检测时间的流逝，进程就能分辨出自己已经被中断了。

的数据项写入新值，则P_1是在读取数据之前还是之后被P_2所中断，决定了返回给P_1的值。注意，进程不能控制中断何时发生，这由操作系统在其资源分配例程中决定。所以，同样的两个进程的不同次执行，其交叉操作可能会不同。这意味着，如果数据空间相交，则其进程可能在不同的执行中具有不同的表现。

我们在第23章的事务隔离性中对交叉执行带来的影响进行了讨论，并且指出事务的不同表现由它们的数据库操作如何交叉进行所决定。这里，共享的数据空间是指数据库（不同事务的工作空间没有交集）。

A.3 线程

在多道程序的操作系统中，线程管理是一个复杂的问题。操作系统必须维护其表中关于每个进程的相当数量的信息，这需要使用大量的内存。这些信息包括进程可以使用的资源总量（如计算时间），目前的使用量（出于计数和调度的目的），虚拟内存目前所在的位置（即其在主存或大容量存储器中的物理地址），调度的优先级，可以访问的文件（因此进程和用户都可以受到保护），等等。另外，这些表会被频繁地扫描，使得操作系统占用了大量的计算时间。例如，系统不得不频繁扫描进程表来确定进程的执行顺序。

由多道程序操作系统引起的空间与时间的开销是它所支持的处理器数量的函数，因此处理器的数量受到了限制。这种限制提出了对低开销进程的需求，并导致了线程和多线程进程的发展。

传统的（或**单线程**（single-threaded））进程只有一个控制点。当持续执行时，控制点从一条指令或I/O操作转移到另一条指令或I/O操作，并且定义了一条执行路径。执行是**顺序**（sequential）进行的，这意味着在任何给定的时间，进程的状态决定了下面将要执行的唯一的指令或I/O操作。**多线程**（multithreaded）进程包含多个控制点，它们定义了称为**线程**（thread）的独立执行路径。如果操作系统支持线程，它为线程分配CPU的方法与为进程分配CPU的方法一样。这样，进程中的线程就能够并发执行。

由于在任何给定的时间，在多线程进程中需要执行的下一条指令可能是它的任何一个线程的下一条指令，所以从总体上看，进程的执行不再是顺序的。同样，因为每一个线程的控制点可以指向任何指令，所以几个线程可能同时执行同一个过程。进程中的每一个线程都有权访问进程的数据库空间中的全局信息（数据和过程）以及存放局部数据的线程私有堆栈。

操作系统管理含n个线程的多线程进程的开销明显低于管理n个进程的开销。

- 操作系统从总体上为进程维护一个数据空间描述。
- 操作系统从总体上考虑进程对资源的利用。
- 进程中的所有线程共享对文件、设备等的访问路径。
- 操作系统可以把程序的执行从进程中的一个线程切换到另一个线程，而不会出现进程之间开销昂贵的**运行环境切换**（context switch）。

多线程的优势需要付出代价。线程必须被谨慎地规划以同步它们对进程内全局数据的访问。而且，在进程中执行的线程之间没有内存保护，所以一个线程执行中的错误可以破坏数据空间中的所有其他线程的信息。

应用服务器是可以被多线程化的模块的一个例子。当用户在客户机上注册后，应用服务

器上就建立一个进行相应的处理会话。每一个会话都有相关的局部数据用于处理用户的请求（如用户的姓名和Id）。会话可能会持续相当长的一段时间，因为用户可能想执行一系列的交互，每个交互一般都需要与用户通信。例如，许多学生可能同时使用注册系统进行数门课程的注册。结果，很多会话可能会同时进行，因此把整个进程致力于处理一个会话将导致处理的低效。而应用服务器中的线程可以通过存储在其私有堆栈中的局部信息与每一个会话关联起来。

A.4 通信

多道程序操作系统通常可以使进程间的相互通信成为可能。所以，系统允许这样的可能性：进程不是各自代表不同的用户进行分离的或互不相关的计算，而是合作完成一些复杂的任务。例如，一个二维空间上的复杂的科学计算可以分解为在不同的空间区域上的数个计算，其中每一个计算都被单独的进程执行。在一个含有几TB字节信息的数据库上进行的查询可能被分配到不同进程的许多查询器中进行。在这些情况下，协作的进程之间需要进行通信。

共享内存体系结构中进程间通信的一种方法是通过它们的数据空间的共享区进行通信。一个进程可以在一些共享位置写一条消息，另一个进程可以在随后进行读取。类似地，进程可以只访问它们的数据空间的共享部分的公共数据结构，以便一个进程可以观察到其他进程对数据的修改。一组线程也可以按这种方式通信。如果进程交互密切，并且有相当数量的信息交换，那么采用这种通信方式特别有效。

共享内存是一种开销非常低的通信技术。然而，它自身也存在问题。

- 必须为进程提供一些机制来同步它们对共享数据的访问。例如，消息的读取方直到写入方的写入完成后才能开始读取。类似地，当进程更新数据结构时，更新过程中所产生的中间状态可能还没有彻底形成。所以，如果两个进程试图并发访问一个数据结构，并且至少一个是执行更新，则其中之一或二者都可能出现运行错误。访问数据结构的代码段被称为**关键段**（critical section）。与访问公共数据库项的事务需要被隔离一样，对数据段的访问也需要被隔离。这类隔离通常被称为**互斥**（mutual exclusion），并且我们说临界区的执行必须互斥。
- 共享内存在分布式系统中并不普遍，进程往往存在于无共享数据空间的不同的计算机上。

为了克服这些问题，多道程序操作系统提供了**进程间通信工具**（Interprocess Communication Facility，IPC），允许进程使用send和receive操作进行消息交换。进程P_1可以调用send，指定另一个进程P_2接收它创建的一条消息。P_2可以调用receive表示它愿意接收发送给它的下一条消息。从被发送直到被接收，消息一直高速缓存在操作系统中，所以发送者和接收者不需要任何共享内存。

网络中的计算机通过通信线路相互连接。就操作系统来说，通信线路是一种新型的I/O设备。为了支持分布式应用的发展，每台计算机的操作系统中实现的IPC被扩展为允许处于网络任何位置的进程都可以使用统一的方式交换消息。所以，无论进程是否在同一台机器上，它们都可以用同样的命令进行通信。这类系统称为**分布式消息传递内核**（distributed message-passing kernel）。因为每一台机器并不必直接与其他机器相连，所以**存储-转发**（store-and-forward）策略得到了利用。消息通过从发送端经中间机器到达接收端。路线由在内核上执行

的分布式算法决定。

通常有数个级别的消息传递服务提供给用户，最基本的是**数据报**（datagram）。这样，在发送端机器上的操作系统在消息发送到网络后不再保存消息的记录。如果消息丢失（由于系统失败或中间机器的资源不足），接收端不会知道消息已丢失。尽管未必会有丢失发生，但是由于这个原因，数据报被认为（相对地）不太可靠。因特网协议（IP）是提供数据报服务的协议的一个范例。

消息传递系统提供的**虚拟电路**（virtual circuit）更为可靠。在这种情形下，发送端将保留消息的一个拷贝直到消息被接收端确认，如果消息丢失则发送端可以重新发送。构建在IP之上的传输控制协议（TCP）是提供虚拟电路服务的协议的范例。UNIX和Windows的**套接字**（socket）结构是分布式消息发送核的用户接口，提供数据报和虚拟电路服务。

对等计算和第22章描述的**远程过程调用通信**（RPC）都可以构建在分布式消息传送内核提供的消息发送服务之上。例如，当一个远程过程调用产生后，调用端（发送端）发出一条包含被调用端（接收端）自变量的消息，并且执行一个接收来等待响应。当（远程）过程结束后，被调用端发送一个包含执行结果的响应消息给调用端。类似地，对等通信是在操作系统级提供的增强型的消息发送机制。其中一方面增强是22.4.2节描述的事物同步机制（同步点）。

与共享存储通信相反，分布式系统中普遍使用消息传递。它也解决了与共享内存有关的同步问题。例如，数据结构可以由单个服务器来管理，而不是由客户进程直接访问共享数据结构。服务器从客户处接受消息（使用对等通信或RPC）并且一次一个地进行处理，这些消息描述了被请求的访问。如果服务器是单线程的，那么不存在对数据结构的并发访问以及同步问题。如果是多线程服务器，那么线程对数据结构的访问必须被同步。受通信开销（包含那些与发生在系统缓冲区、发送端、接收端的地址空间之间的消息拷贝相关的开销）以及与传递消息相关的通信延迟的制约，消息传递只适用于进程间松散交互，并且通信的信息量比较小的情形。

附录B 参考文献

Abiteboul, S., and Kanellakis, P. (1998). Object identity as a query language primitive. *Journal of the ACM* **45**(5): 798–842.

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*, Addison-Wesley, Reading, 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.

Abiteboul, S., Buneman, P., and Suciu, D. (2000). *Data on the Web*, Morgan Kaufmann, San Francisco.

Adam, N., Atluri, V., and Huang, W. (1998). Modeling and analysis of workflows using Petri nets. *Journal of Intelligent Information Systems* **10**(2): 131–158.

Agrawal, D., Bernstein, A., Gupta, P., and Sengupta, S. (1987). Distributed optimistic concurrency control with reduced rollback. *Distributed Computing* **2**(1): 45–59.

Agrawal, R., Imielinski, T., and Swami, A. (1993). Database mining: A performance perspective. *IEEE Transactions on Knowledge and Date Engineering* **5**(6): 914–925.

Agrawal, S., Agrawal, R., Deshpande, P., Gupta, A., Naughton, J., Ramakrishnan, R., and Sarawagi, S. (1996). On the computation of multidimensional aggregates. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Mombai, India, 506–521.

Aho, A., and Ullman, J. (1979). Universality of data retrieval languages. *ACM Symposium on Principles of Programming Languages (POPL)*, 110–120.

Alagic, S. (1999). Type-checking OQL queries in the ODMG type systems. *ACM Transactions on Database Systems* **24**(3): 319–360.

Alonso, G., Agrawal, D., Abbadi, A. E., Kamath, M., Günthör, R., and Mohan, C. (1996). Advanced transaction models in workflow contexts. *Proceedings of the International Conference on Data Engineering (ICDE)*, New Orleans, 574–581.

Alonso, G., Agrawal, D., Abbadi, A. E., and Mohan., C. (1997). Functionality and limitations of current workflow management systems. *IEEE-Expert, Special issue on Cooperative Information Systems* **1**(9).

Andrade, J. M., Carges, M. T., Dwyer, T. J., and Felts, S. D. (1996). *The TUXEDO System, Software for Constructing and Managing Distributed Business Applications*, Addison-Wesley, Reading, MA.

Apt, K., Blair, H., and Walker, A. (1988). Towards a theory of declarative knowledge. In J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, San Francisco, 89–148.

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., Chamberlin, D., Eswaran, K., Gray, J., Griffiths, P., King, W., Lorie, R., McJones, P., Mehl, J., Putzolu, G., Traiger, I., and Watson, V. (1976). System R: A relational approach to database management. *ACM Transactions on Database*

Systems **1**(2): 97–137.

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.

Attie, P., Singh, M., Sheth, A., and Rusinkiewicz, M. (1993). Specifying and enforcing intertask dependencies. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Dublin, 134–145.

Attie, P., Singh, M., Emerson, E., Sheth, A., and Rusinkiewicz, M. (1996). Scheduling workflows by enforcing intertask dependencies. *Distributed Systems Engineering Journal* **3**(4): 222–238.

Atzeni, P., and Antonellis, V. D. (1993). *Relational Database Theory*. Benjamin-Cummings, San Francisco.

Avron, A., and Hirshfeld, J. (1994). Query evaluation, relative safety, and domain independence in first-order databases. *Methods of Logic in Computer Science* **1**: 261–278.

Bancilhon, F., and Spyratos, N. (1981). Update semantics of relational views. *ACM Transactions on Database Systems* **6**(4): 557–575.

Bancilhon, F., Delobel, C., and Kanellakis, P. (eds.) (1990). *Building an Object-Oriented Database System: The Story of 02*. Morgan Kaufmann, San Francisco.

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., 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., 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., 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., Mendelson, A., Sagiv, Y., and Ullman, J. (1981). Equivalence of relational database schemes. *SIAM Journal of Computing* **10**(2): 352–370.

Beeri, C., Bernstein, P., Goodman, N., Lai, M.-Y., and Shasha, D. (1983). A concurrency control theory for nested transactions. *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, Montreal, Canada, 45–62.

Beeri, C., Bernstein, P., and Goodman, N. (1989). A model for concurrency in nested transaction systems. *Journal of the ACM* **36**(2): 230–269.

Bell, D., and Grimson, J. (1992). *Distributed Database Systems*. Addison-Wesley, Reading, MA.

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. (1995). A critique of ANSI SQL isolation levels. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Jose, CA, 1–10.

Berg, C., and Virginia, C. (2000). *Advanced Java 2 Development for Enterprise Applications*, (2nd ed.). Prentice-Hall, Englewood Cliffs, NJ.

Bernstein, P. (1976). Synthesizing third normal form from functional dependencies. *ACM Transactions on Database Systems* **1**(4): 277–298.

Bernstein, P., and Chiu, D. (1981). Using semi-joins to solve relational queries. *Journal of the ACM* **28**(1): 28–40.

Bernstein, P., Goodman, N., Wong, E., Reeve, C., and Rothnie, J. (1981). Query processing in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems* **6**(4): 602–625.

Bernstein, P., and Goodman, N. (1983). Multiversion concurrency control—Theory and algorithms. *ACM Transactions on Database Systems* **8**(4): 465–483.

Bernstein, P., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA.

Bernstein, A. J., and Lewis, P. M. (1996). High-performance transaction systems using transaction semantics. *Distributed and Parallel Databases* 4(1).

Bernstein, P., and Newcomer, E. (1997). *Principles of Transaction Processing*. Morgan Kaufmann, San Francisco.

Bernstein, A. J., Gerstl, D., Leung, W.-H., and Lewis, P. M. (1998). Design and performance of an assertional concurrency control system. *Proceedings of the International Conference on Data Engineering* (ICDE), Orlando, 436–445.

Bernstein, A. J., Gerstl, D., and Lewis, P. (1999). Concurrency control for step decomposed transactions. *Information Systems* **24**(8): 673–698.

Bernstein, A. J., Gerstl, D., Lewis, P., and Lu, S. (1999). Using transaction semantics to increase performance. *International Workshop on High Performance Transaction Systems*, Pacific Grove, CA, 26–29.

Bernstein, A. J., Lewis, P., and Lu, S. (2000). Semantic conditions for correctness at different isolation levels. *Proceedings of the International Conference on Data Engineering*, San Diego, 507–566.

Birrell, A., and Nelson, B. (1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems* **2**(1): 39–59.

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-Base Systems*. In Vol. 1762 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, 31–49.

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.

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.

Bonner, A. (1999). Workflow, transactions, and datalog. *ACM SIGACT-SIGMOD-SIGART*

Symposium on Principles of Database Systems (PODS), Philadelphia, 294–305.

Booch, G., Rumbaugh, J., and Jacobson, I. (1999). *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, 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, Reading, MA.

Bradley, N. (2000b). *The XSL Companion*. Addison-Wesley, Reading, MA.

Breitbart, Y., Garcia-Molina, H., and Silberschatz, A. (1992). Overview of multidatabase transaction management. *VLDB Journal* **1**(2): 181–240.

Bukhres, O., and Kueshn, E. (eds.) (1995). *Distributed and Parallel Databases—An International Journal*, Special Issue on Software Support for Workflow Management.

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. (1994). *Object Database Management* (rev. ed.), Addison-Wesley, Reading, MA.

Cattell, R., and Barry, D. (eds.) (2000). *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann, San Francisco.

Ceri, S., and Pelagatti, G. (1984). *Distributed Databases: Principles and Systems*. McGraw-Hill, New York.

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, 128–136.

Chamberlin, D., Robie, J., and Florescu, D. (2000). Quilt: An XML query language for heterogeneous data sources. In *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany (*http://www.almaden.ibm.com/cs/people/chamberlin/quilt_lncs.pdf*).

Chang, S., and Cheng, W. (1980). A methodology for structured database decomposition. *IEEE-TSE* **6**(2): 205–218.

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., and Dayal, U. (1997). An overview of data warehousing and OLAP technology. *SIGMOD Record* **26**(1): 65–74.

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.

Chaum, D., Fiat, A., and Noar, M. (1988). Untraceable electronic cash. *Advances in Cryptology: Crypto'88 Proceedings*. In *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, 319–327.

Chen, P. (1976). The Entity-Relationship Model—Towards a unified view of data. *ACM Transactions on Database Systems* **1**(1): 9–36.

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.

Chrysanthis, P., and Ramaritham, K. (1990). ACTA: A framework for specifying and reasoning about transaction structure and behavior. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Atlantic City, NJ, 194–205.

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, Reading, MA.

Codd, E. (1995). Twelve rules for on-line analytic processing. *Computerworld*, April 13.

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.

Cox, B., Tygar, J., and Sirbu, M. (1995). Netbill security and transaction protocol. *Proceedings of the 1st USENIX workshop on Electronic Commerce*, New York, Vol. 1.

Date, C. (1992). Relational calculus as an aid to effective query formulation. In C. Date and H. Darwen (eds.), *Relational Database Writings*, Addison-Wesley, Reading, MA.

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

Davulcu, H., Kifer, M., Ramakrishnan, C.R., and Ramakrishnan, I.V. (1998). Logic based modeling and analysis of workflows. *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Seattle, 25–33.

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., Fernandez, 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.

Diffie, W., and Hellman, M. (1976). New directions in cryptography. *IEEE Transactions on Information Theory* **IT-22**(6): 644–654.

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.

Elmagarmid, A. (ed.) (1992). *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, San Francisco.

Elmagarmid, A., Leu, Y., Litwin, W., and Rusinkiewicz, M. (1990). A multidatabase transaction model for interbase. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Brisbane, Australia, 507–518.

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.

Fayyad, U., Piatetsky-Shapiro, G., Smyth, P., and Uthurusamy, R. (eds.) (1996). *Advances in Knowledge Discovery and Data Mining*. The MIT Press, Cambridge, MA.

Fekete, A., Lynch, N., Merritt, M., and Weihl, W. (1989). Commutativity-based locking for nested transactions. *Technical Report MIT/LCS/TM-370.b*, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., and Shasha, D. (2000). Making snapshot isolation serializable. *http://www.cs.umb.edu/poneil/publist.html*.

Flach, P. A., and Savnik, I. (1999). Database dependency discovery: A machine learning approach. *AI Communications* **12**(3): 139–160.

Florescu, D., Deutsch, A., Levy, A., Suciu, D., and Fernandez, M. (1999). A query language for XML. *Proceedings of the Eighth International World Wide Web Conference*, Toronto, Canada.

Fowler, M., and Scott, K. (1999). *UML Distilled*. Addison-Wesley, Reading, 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.

Fuh, Y.-C., Dessloch, S., Chen, W., Mattos, N., Tran, B., Lindsay, B., DeMichiel, L., Rielau, S., and Mannhaupt, D. (1999). Implementation of SQL3 structured types with inheritance and value substitutability. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Edinburgh, Scotland, 565–574.

Garcia-Molina, H., and Salem, K. (1987). Sagas. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Francisco, 249–259.

Garcia-Molina, H., Gawlick, D., Klien, J., Kleissner, K., and Salem, K. (1991). Modeling long-running activities as nested Sagas. *Quarterly Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* **14**(1): 14–18.

Garcia-Molina, H., Ullman, J., and Widom, J. (2000). *Database System Implementation*, Prentice-Hall, Englewood Cliffs, NJ.

Georgakopoulos, D., Hornick, M., Krychniak, P., and Manola, F. (1994). Specification and management of extended transactions in a programmable transaction environment. *Proceedings of the International Conference on Data Engineering (ICDE)*, Houston, 462–473.

Georgakopoulos, D., Hornick, M., and Sheth, A. (1995). An overview of workflow management: From process modeling to infrastructure for automation. *Journal on Distributed and Parallel Database Systems* **3**(2): 119–153.

Gifford, D. (1979). Weighted voting for replicated data. *Proceedings of the ACM 7th Symposium on Operating Systems Principles*, Pacific Grove, CA, 150–162.

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. (1981). The transaction concept: Virtues and limitations. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Cannes, 144–154.

Gray, J., and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco.

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., McJones, P., and Blasgen, M. (1981). The recovery manager of the System R database manager. *Computer Surveys* **13**(2): 223–242.

Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., and Venkatrao, M. (1997). Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In Fayyad et al. (eds.), *Data Mining and Knowledge Discovery*, The MIT Press, Cambridge, MA.

Griffiths-Selinger, P., and Adiba, M. (1980). Access path selection in distributed data base management systems. *Proceedings of the International Conference on Data Bases*, Aberdeen, Scotland, 204–215.

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 Subrahmanian, V. (1993). Maintaining views incrementally. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, DC, 157–166.

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.

Hadzilacos, V. (1983). An operational model for database system reliability. *SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Atlanta, 244–256.

Hadzilacos, V., and Papadimitriou, C. (1985). Algorithmic aspects of multiversion concurrency control. *SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Portland, OR, 96–104.

Haerder, T., and Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Computing Surveys* **15**(4): 287–317.

Hall, M. (2000). *Core Servlets and JavaServer Pages (JSP)*. Prentice-Hall, Englewood Cliffs, NJ.

Han, J., and Kamber, M. (2001). *Data Mining: Concepts and Techniques*. Morgan Kaufmann, San Francisco.

Harinarayan, V., Rajaraman, A., and Ullman, J. (1996). Implementing data cubes efficiently. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, 205–216.

Henning, M., and Vinoski, S. (1999). *Advanced CORBA Programming with C++*. Addison-Wesley, Reading, MA.

Hsu, M. (1995). Letter from the special issues editor. *Quarterly Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, **18**(1): 2–3, Special Issue on Workflow Systems.

Hu, W. (1995). *DCE Security Programming*. O'Reilly and Associates, Sebastopol, CA.

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.

Hull, R., Llirbat, F., Simon, E., Su, J., Dong, G., Kumar, B., and Zhou, G. (1999). Declarative workflows that support easy modification and dynamic browsing. *Proceedings of the ACM International Joint Conference on Work Activities Coordination and Collaboration (WACC)*, San Francisco, 69–78.

Hunter, J., and Crawford, W. (1998). *Java Servlet Programming*. O'Reilly and Associates, Sebastopol, CA.

IBM (1991). System network architecture (SNA) logical unit 6.2 (LU6.2): Transaction programmer's reference manual for LU6.2. *Technical Report GC30-3084*.

i-Escrow (2000). i-Escrow. (*http://www.iescrow.com*).

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.

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.

Jajodia, S., and Kerschberg, L. (eds.) (1997). *Advanced Transaction Models and Architectures*, Kluwer Academic Publishers, Dordrecht, Netherlands.

Kamath, M., and Ramamritham, K. (1996). Correctness issues in workflow management. *Distributed Systems Engineering Journal* 3(4): 213–221.

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.

Kay, M. (2000). *XSLT Programmer's Reference*. Wrox Press, Paris.

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.

Khoshafian, S., and Buckiewicz, M. (1995). *Introduction to Groupware, Workflow, and Workgroup Computing*. John Wiley & Sons, New York.

Kifer, M. (1988). On safety, domain independence, and capturability of database queries. *Proceedings of the Third International Conference on Data and Knowledge Bases*, Jerusalem, Israel, 405–415.

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., 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., 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, Reading, MA.

Knuth, D. (1998). *The Art of Computer Programming: Vol III, Sorting and Searching*, (3rd ed.), Addison-Wesley, Reading, MA.

Korth, H., Levy, E., and Silberschatz, A. (1990). A formal approach to recovery by compensating transactions. *Proceedings of the International Conference on Very Large Data Bases*, Brisbane, Australia, 95–106.

Kung, H., and Robinson, J. (1981). On optimistic methods for concurrency control. *ACM Transactions on Database Systems* **6**(2): 213–226.

Lacroix, M., and Pirotte, A. (1977). Domain-oriented relational languages. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, 370–378.

Lampson, B., and Sturgis, H. (1979). Crash recovery in a distributed data storage system. *Technical Report*, Xerox Palo Alto Research Center, Palo Alto, CA.

Lampson, B., Paul, M., and Seigert, H. (1981). *Distributed Systems: Architecture and Implementation (An Advanced Course)*. Springer-Verlag, Heidelberg, Germany.

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.

Lynch, N., Merritt, M., Weihl, W., and Fekete, A. (1994). *Atomic Transactions*, Morgan Kaufmann, San Francisco.

Maier, D. (1983). *The Theory of Relational Databases*. Computer Science Press. (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., and Simon, A. (1992). *Understanding the New SQL: A Complete Guide*. 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.

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.

Mohan, C., Lindsay, B., and Obermarck, R. (1986). Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems* **11**(4): 378–396.

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwartz, P. (1992). Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks

using write-ahead logging. *ACM Transactions on Database Systems* **17**(1): 94–162.

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.

Moss, J. (1985). *Nested Transactions: An Approach to Reliable Computing*. The MIT Press, Cambridge, MA.

Nam (1999). *http://www.w3.org/TR/1999/REC-xml-names-19990114/*.

Netscape (2000). SSL-3 specifications. *http://home.netscape.com/eng/ssl3/index.html*.

Neuman, B. C., and Ts'o, T. (1994). Kerberos: An authentication service for computer networks. *IEEE Communications* **32**(9): 33–38.

Novikoff, A. (1962). On Convergence Proofs for Perceptrons. *Proceedings of the Symposium on Mathematical Theory of Automata*, New York, 615–621.

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.

Orfali, R., and Harkey, D. (1998). *Client/Server Programing with Java and CORBA*. John Wiley, New York.

Orlowska, M., Rajapakse, J., and ter Hofstede, A. (1996). Verification problems in conceptual workflow specifications. *Proceedings of the International Conference on Conceptual Modeling*. Vol. 1157 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany.

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.

Ozsu, M., and Valduriez, P. (1991). *Principles of Distributed Database Systems*, Prentice-Hall, Englewood Cliffs, NJ.

Papadimitriou, C. (1986). *The Theory of Concurrency Control*. Computer Science Press, Rockville, MD.

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.

Peterson, L., and Davies, B. (2000). *Computer Networks: A Systems Approach* (2nd ed.). Morgan Kaufmann, San Francisco.

Pope, A. (1998). *The CORBA Reference Guide*. Addison-Wesley, Reading, MA.

PostgreSQL (2000). PostgreSQL. *http://www.postgresql.org*.

Pressman, R. (1997). *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York.

Ram, S. (1995). Deriving functional dependencies from the entity-relationship model. *Communications of the ACM* **38**(9): 95–107.

Ramakrishnan, R., and Ullman, J. (1995). A survey of deductive databases. *Journal of Logic Programming* **23**(2): 125–149.

Ramakrishnan, R., Srivastava, D., Sudarshan, S., and Seshadri, P. (1994). The CORAL deductive database system. *VLDB Journal* **3**(2): 161–210.

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.

Reuter, A., and Wachter, H. (1991). The contract model. *Quarterly Bulletin of the IEEE Computer Society Technical Commmttee on Data Engineering* **14**(1): 39–43.

Rivest, R., Shamir, A., and Adelman, L. (1978). On digital signatures and public-key cryptosystems. *Communications of the ACM* **21**(2): 120–126.

Robie, J., Lapp, J., and Schach, D. (1998). XML query language (XQL). *Proceedings of the Query Languages Workshop*, Boston (*http://www.w3.org/TandS/QL/QL98/pp/xql.html*).

Robie, J., Chamberlin, D., and Florescu, D. (2000). Quilt: An XML query language. *XML Europe* (*http://www.almaden.ibm.com/cs/people/chamberlin/robie_XML_Europe.pdf*).

Rosenberry, W., Kenney, D., and Fisher, G. (1992). *Understanding DCE*, O'Reilly and Associates, Sebastopol, CA.

Rosenkrantz, D., Stearns, R., and Lewis, P. (1978). System level concurrency control for distributed database systems. *ACM Transactions on Database Systems* **3**(2): 178–198.

Rosenkrantz, D., Stearns, R., and Lewis, P. (1984). Consistency and serializability in concurrent database systems. *SIAM Journal of Computing* **13**(3): 505–530.

Ross, K., and Srivastava, D. (1997). Fast computation of sparse datacubes. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Athens, Greece, 116–125.

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.

Rusinkiewicz, M., and Sheth, A. (1994). Specification and execution of transactional workflows. In W. Kim (ed.), *Modern Database Systems: The Object Model, Interoperability, and Beyond*, ACM Press, New York, 592–620.

Sagonas, K., Swift, T., and Warren, D. (1994). XSB as an efficient deductive database engine. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Minneapolis, MN, 442–453.

Savnik, I., and Flach, P. (1993). Bottom-up induction of functional dependencies from relations. *Proceedings of the AAAI Knowledge Discovery in Databases Workshop (KDD)*, Ljubliana, Slovenija, 174–185.

Schach, S. (1990). *Software Engineering*. Aksen Associates, Homewood, IL.

Schek, H.-J., Weikum, G., and Ye, H. (1993). Towards a unified theory of concurrency control and recovery. *ACM SIGACT-SIGMOD-SIGART Conference on Principles of Database Systems (PODS)*, Washington, DC, 300–311.

Schneier, B. (1995). *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley, New York.

Sciore, E. (1983). Improving database schemes by adding attributes. *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, New York, 379–383.

Sebesta, R. (2001). *Programming the World Wide Web*. Addison-Wesley, Reading, MA.

SGM (1986). Information processing—text and office systems—Standard Generalized Markup Language (SGML). *ISO Standard 8879*.

Shipman, D. (1981). The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems* **6**(1): 140–173.

Signore, R., Creamer, J., and Stegman, M. (1995). *The ODBC Solution: Open Database Connectivity in Distributed Environments*. McGraw-Hill, New York.

Singh, M. (1996). Synthesizing distributed constrained events from transactional workflow specifications. *Proceedings of the International Conference on Data Engineering*, New Orleans, 616–623.

Skeen, D. (1981). Nonblocking commit protocols. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Ann Arbor, MI, 133–142.

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*.

SQLJ (2000). SQLJ. *http://www.sqlj.org*.

Stallings, W. (1999). *Cryptography and Network Security: Principles and Practice*, (2nd ed.). Prentice-Hall, Englewood Cliffs, NJ.

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.

Steiner, J. G., Neuman, B. C., and Schiller, J. I. (1988). Kerberos: An authentication service for open network systems. *USENIX Conference Proceedings*, Dallas, 191–202.

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, Reading, MA.

Stonebreaker, M., and Kemnitz, G. (1991). The POSTGRES next generation database management system. *Communications of the ACM* **10**(34): 78–92.

Summerville, I. (1996). *Software Engineering*, Addison-Wesley, Reading, MA.

Sun (2000). JDBC data access API. *http://java.sun.com/products/jdbc/*.

Teorey, T. (1992). *Database modeling and design: The E-R approach*. Morgan Kaufmann, San Francisco.

Thalheim, B. (1992). *Fundamentals of Entity-Relationship Modeling*. Springer-Verlag, Berlin.

Thomas, R. (1979). A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems* **4**(2): 180–209.

Topor, R., and Sonenberg, E. (1988). On domain independent databases. In J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, Los Altos, CA, 217–240.

Transarc (1996). Encina monitor programmer's guide and reference. *Technical Report ENC-D5008-06*, Transarc Corporation, Pittsburgh.

Ullman, J. (1982). *Priniciples of Database Systems*. Computer Science Press, Rockville, MD.

Ullman, J. (1988). *Principles of Database and Knowledge-Base Systems, Volumes 1 and 2*. Computer Science Press, Rockville, MD.

Vaghani, J., Ramamohanarao, K., Kemp, D., Somogyi, Z., Stuckey, P., Leask, T. and Harland, J. (1994). The Aditi deductive database system. *The VLDB Journal* **3**(2): 245–288.

Valduriez, P. (1987). Join indices. *ACM Transactions on Database Systems* **12**(2): 218–246.

Van Gelder, A., and Topor, R. (1991). Safety and translation of relational calculus queries. *ACM Transactions on Database Systems* **16**(2): 235–278.

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.

VISA (2000). SET specifications. *http://www.visa.com/nt/ecomm/set/intro.html*.

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.

Weihl, W. (1988). Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers* **37**(12): 1488–1505.

Weikum, G. (1991). Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems* **16**(1): 132–180.

Weikum, G., and Schek, H. (1991). Multi-level transactions and open nested transactions. *Quarterly Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* **14**(1): 55–66.

Widom, J., and Ceri, S. (1996). *Active Database Systems*. Morgan Kaufmann, San Francisco.

Wodtke, D., and Weikum, G. (1997). A formal foundation for distributed workflow execution based on state charts. *Proceedings of the International Conference on Database Theory (ICDT)*, Delphi, Greece, 230–246.

Wong, E. (1977). Retrieving dispersed data from SDD-1: A system for distributed databases. *Proceedings of the 2nd International Berkeley Workshop on Distributed Data Management and Data Networks*, Berkeley, CA, 217–235.

Wong, E., and Youssefi, K. (1976). Decomposition—A strategy for query processing. *ACM Transactions on Database Systems* **1**(3): 223–241.

Worah, D., and Sheth, A. (1997). Transactions in transactional workflows. In S. Jajodia and L. Kerschberg (eds.), *Advanced Transaction Models, and Architectures*, Kluwer Academic Publishers, Dordrecht, Netherlands, 3–45.

Workflow Management Coalition (2000). WfMC standards. *http://www.aiim.org/wfmc/standards/docs.htm*.

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*.

X/Open CAE Specification Structured Transaction Definition Language (STDL) (1996a). X/Open Co., Ltd., London.

X/Open Guide Distributed Transaction Processing: Reference Model, Version 3 (1996b). X/Open Co., Ltd., London.

XPath (1999). XML Path Language (XPath), version 1.0. *http://www.w3.org/TR/xpath/*.

XPointer (2000). XML Pointer Language (XPointer), version 1.0. *http://www.w3.org/TR/xptr/*.

XQuery: (2001) A query language for XML. Editors: D. Chamberlin, , D. Florescu, T. Robie, T. Simeon, M. Stefanescu, *http://www.w3.org/TR/xquery*.

XSL (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/*.

Zhao, Y., Deshpande, P., Naughton, J., and Shukla, A. (1998). Simultaneous optimization and evaluation of multiple dimensional queries. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Seattle, WA, 271–282.

Zloof, M. (1975). Query by example, *NCC*, AFIPS Press, Montvale, NJ.