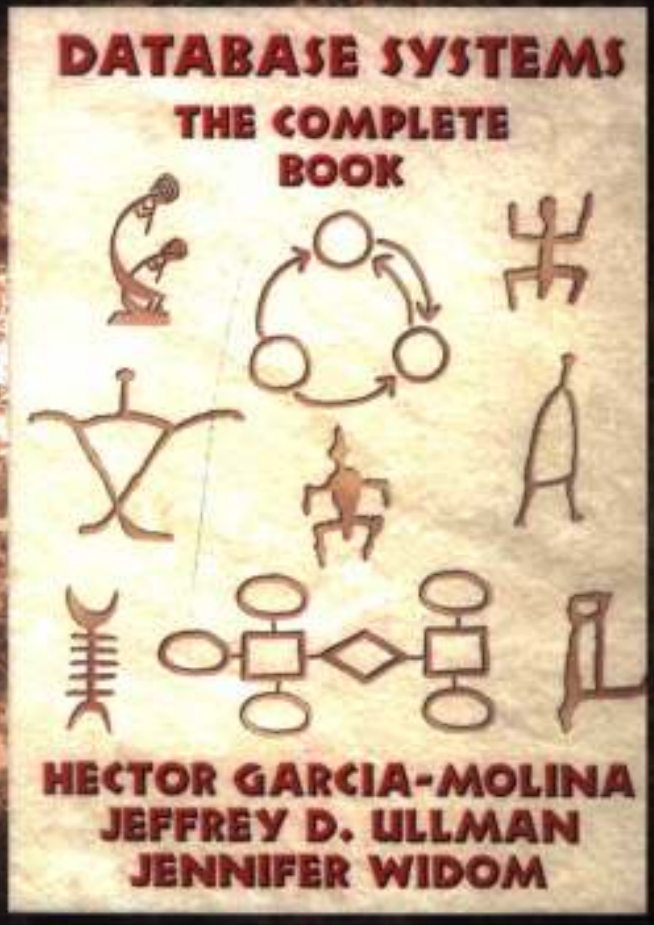
计 算 机 科 学 丛 书

数据库系统全书

（美） Hector Garcia-Molina Jeffrey D. Ullman Jennifer Widom 著
杨冬青 唐世渭 等译



Database Systems
The Complete Book

机械工业出版社
China Machine Press

本书是斯坦福大学知名计算机科学家Hector Garcia-Molina，Jeffrey D.Ullman和Jennifer Widom合作编写的一本数据库系统引论书籍。书的前半部分从数据库设计者、用户和应用程序员的角度深入地介绍了数据库，包括最新数据库标准SQL-1999，SQL/PSM，SQL/CLI，ODL和XML，相比其他大多数书籍，更多地介绍了SQL内容。本书的后半部分是从DBMS实现的角度来介绍数据库的，覆盖了这个领域内的基本技术，并且比其他大多数书籍更多地介绍了查询优化。高级论题包括多维和位图索引、分布式事务处理和信息集成技术。本书既可用作大学教科书，也可作为该领域专业人员的参考书。

本书显著特色：

- 使用人们普遍关注的、现实世界的例子提高可读性
- SQL/PSM（持久存储模块）、JDBC（Java接口）和SQL/CLI（ODBC或开放式数据库连接）等内容为本书所特有
- 用ODMG标准ODL介绍了面向对象设计，用SQL-99标准介绍了对象-关系设计
- 借助关系代数，讲述了查询处理和查询优化的扩展内容
- 讨论了信息集成技术，包括数据仓库、协调器、OLAP、数据立方体和数据挖掘技术
- 解释了很多重要的专门技术，如RAID盘的错误纠正、位图索引、统计数据的应用以及指针混合
- 通过主页http://www-db.stanford.edu/~ullman/dscb.html提供本书更多的附加资料

作者简介

Hector Garcia-Molina 是斯坦福大学计算机科学与电气工程系教授，发表过大量关于数据库系统、分布式系统和数字图书馆领域的论文。

Jeffrey D. Ullman 是斯坦福大学计算机科学教授，他独立或合作出版了15本著作，发表了170篇技术论文，其中包括《A First Course in Database Systems》（Prentice Hall出版社，1997）和《Elements of ML Programming》（Prentice Hall出版社，1998）。他的研究兴趣包括数据库理论、数据库集成、数据挖掘和利用信息基础设施进行教育，他获得了Guggenheim Fellowship等多种奖励，并被推选进入美国国家工程院。他还被授予1996年Sigmod贡献奖和1998年Karl V. Karstrom杰出教育家奖。

Jennifer Widom 是斯坦福大学计算机科学与电气工程系的副教授，她还是《A First Course in Database Systems》的作者之一。她的研究兴趣包括半结构化数据的数据库系统和XML、数据仓库以及主动数据库系统。

PEARSON Prentice Hall www.PearsonEd.com

ISBN 7-111-12541-X

9 787111 125419

HZ BOOKS 华章图书

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

http://www.hzbook.com

ISBN 7-111-12541-X/TP·2791

定价：65.00元

计 算 机 科 学 丛 书

数据库系统全书

(美) Hector Garcia-Molina Jeffrey D. Ullman Jennifer Widom 著
岳丽华 杨冬青 龙启昌 唐世渭 徐其均 等译



Database Systems
The Complete Book

机械工业出版社
China Machine Press

本书是斯坦福大学计算机科学专业数据库系列课程教科书。书中对数据库系统基本原理以及数据库系统实现进行了深入阐述，并对ODL、SQL、关系代数、面向对象查询、事务管理、并发控制等内容展开具体讨论。对该领域内的一些最新技术，诸如数据仓库、数据控掘、数据立方体系统等，也给予了介绍。

本书适合作为高等院校计算机专业研究生的教材或本科生的教学参考书，也适合作为从事相关研究或开发工作的专业技术人员的高级参考资料。

Simplified Chinese deition copyright © 2002 by PEARSON EDUCATION ASIA LIMITED and China Machine Press.

Original English language title: Database Systems: The Complete Book, ISBN: 0-13-031995-3, 1st Edition by Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom, Copyright 2002

All rights reserved.

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

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。

本书版权登记号：图字：01-2002-1880

图书在版编目（CIP）数据

数据库系统全书 /（美）加西亚－莫里纳（Garcia-Molina, H.），（美）沃尔曼（Ullman, J. D.），（美）威德姆（Widom, J.）著；岳丽华等译. －北京：机械工业出版社，2003.8

（计算机科学丛书）

书名原文：Database Systems: The Complete Book

ISBN 7-111-12541-X

Ⅰ.数… Ⅱ.①加… ②沃… ③威… ④岳… Ⅲ.数据库系统 Ⅳ.TP311.13

中国版本图书馆CIP数据核字（2003）第055263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：蒋 祎

北京瑞德印刷有限公司印刷 · 新华书店北京发行所发行

2003年10月第1版第1次印刷

787mm × 1092mm 1/16 · 46.5印张

印数：0 001-5 000册

定价：65.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线电话：（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

数据库系统课程是计算机科学与技术专业的一门必修课程，有一本好的数据库系统课程教材对计算机科学技术专业的教学与学习是非常重要的。

本书是斯坦福大学计算机系Jeffrey D. Ullman教授，Jennifer Widom副教授与Hector Garcia-Molina教授等学者在重新整理其使用多年的数据库课程教材的基础上，精心编辑成的一本适用于20周数据库系统课程教学的教材。Jeffrey D. Ullman教授已从教近40年。自1980年编写了其第一本数据库教材《数据库系统原理》以来，已出版过多本数据库系统方面的教材。本教材主要特点如下：（1）内容全面。与数据库系统相关的内容书中都有讨论。所介绍的语言不仅仅介绍了标准SQL语言、关系代数语言，还介绍了较新的面向对象查询语言OQL和逻辑语言DATALOG。（2）内容深入。如讨论查询处理时，不仅给出了查询逻辑计划与物理计划处理，而且还给出了有关查询计划执行时的多种处理算法，这些在一般的数据库系统书中是不常见到的。（3）举例多、习题多。书中通过反复举例和在每一小节（而不是每一章）之后给出大量标有难度的习题，可以有效地引导学生循序渐进地深入掌握教学内容。（4）在讨论一些重要的结论时，不仅陈述了该结论，而且还给出了该结论的证明，这反映了Ullman教授扎实的数学背景的渊源，为维护数据库理论的严格性提供了范例。

本书是Ullman教授把斯坦福大学季度学制中的本科生数据库基础课（CS145）和本科生高年级数据库系统实现课（CS245）合二为一，以适应学期学制教学而准备的教材。我国大学基本上都实行学期学制，数据库课程的学时一般都在60学时以上。随着教学手段的改进（如使用多媒体教学），大大节省了板书时间，使得教学内容的扩展成为可能。因此本书的大部分内容可以在一学期中完成。如果不需要对数据库系统的某个方面做深入讨论，可以省略某些内容而不影响教学的系统性。数据库系统课程教学中需要给学生足够多的、难易程度合适的实验练习，在Ullman教授的Web主页上还给出了与本书配合的课程实验内容，这为使用本书作为数据库系统课程教学提供了非常大的帮助。

本书的翻译组织安排如下，岳丽华负责翻译审校了1-8章，第11章及前言，龚育昌负责翻译审校了9-10章，其余章节由杨冬青、唐世渭、徐其钧诸位老师负责翻译审校。

另外参加翻译工作的人员还有：周英华、刘晓红、陆岚、杨洋、柳建平、杨晓宁、韦鹏、齐冀、陈安、杨良怀、王爱华、王腾蛟、叶茂盛、赵绍军、赵畅、高桂英。

限于译者的水平，译文中难免有错误与不足之处，欢迎读者批评指正。

译　者

2003年5月

前　言

在斯坦福，因为执行的是一年四学期制，所以数据库引论课被分为两门课程。第一门课程是CS145，该课程中只要求学生学会使用数据库系统，而不必做关于DBMS实现方面的实验。CS145课是 CS245的预修课。CS245是介绍DBMS实现。学生若想进一步学习数据库方向课程，则可以学CS345（此课是理论课）、CS346（该课是 DBMS实现实验课），以及CS347课程（该课介绍事务处理及分布式数据库）。

从1997年开始，我们已经出版了两本配套的书。《数据库系统基础教程》是为CS145课程编写。《数据库系统实现》是为CS245课程以及部分CS346课程编写。由于很多学校实行学期制，或者是将这两门数据库引论课组合成一门引论课程，因此，我们感到有必要将上述两本书合成一本书。同时，数据库系统的发展迫使要开出新的课程以介绍很多新课题。这样，我们加入了一些新的内容，大多是与应用程序设计领域相关。如对象关系数据、SQL/PSM（存储程序）、SQL/CLI（C/SQL接口标准）和JDBC（与JAVA/SQL相同）等。

如何使用该书

我们建议用两个学季来讲该书。如果你遵照斯坦福方法，则在第一学季中讲前十章内容，在第二学季讲后十章内容。如果你想在一学季中讲完本书，则要省略一些章节不讲。

通常，我们建议第2~7章，11~13章，以及17~18 章应该给定较高的优先数，而这些章中有一些可以跳过不讲。

若如同我们在CS145课程中所做的那样，你想给学生一个真实的数据库应用设计和实现项目作业，则应该对书的讲解顺序做某些调整，使SQL的介绍较早开始。虽然学生在做数据库设计时需要规范化知识，但你可以推迟有关函数依赖的介绍。

预备知识

我们曾经将此书作为本科生和一年级研究生都选的课程的教材。该课程的正规的预修条件是将其作为二年级课程，在此之前已学习过：（1）数据结构、算法、离散数学。（2）软件系统、软件工程和程序设计语言等。所有这些内容中最重要的是学生至少要对如下内容有基本的理解：代数表达式和代数定律、逻辑、基本的数据结构（如查询树和图）、面向对象程序设计概念和程序设计环境。可是我们相信最合适的知识基础是在修完典型的计算机科学专业课程体系的三年级后。

习题

本书几乎在每一节都包括了一些扩充练习，我们用感叹号对难题做了标记，对最难的习题用双感叹号做了标记。

有些习题标有星号，对这些习题我们将努力通过该书的网页提供有关该题的解法，以方便读者访问。这些解是公开有效的，并且可用于自测。注意，在有些情况下，习题B是你对另

一习题A的修改和改造，于是如果 A 的某一部分有解，则你也应该能解出B的对应部分。

WWW网上支持

本书的主页地址是：

http: //www-db.stanford.edu/~ullman/dscb.html

这里有对加星号习题的解的勘误表及书的备份。同时还有与课程CS145和CS246 相关的作业、课程实现及考卷等内容。

致谢

有很多人曾帮助过我们，他们或是提供了本书及它的后续版内容的最初材料，或者是提供了本书或其他基于网页材料的勘误表。我们对所有这些帮助过我们的人表示感谢。他们是：

Marc Abromowit, Joseph H. Adamski, Brad Adelberg, Gleb Ashimov, Donald Aingworth, Jonathan Becker, Margaret Bentiez, Larry Bonham, Phillip Bonnet, David Brokaw, Ed Burns, Karen Buter, Christopher Chan, Sudarshan Chawathe, Per Christensen, Ed Chang, Surajit Chaudhuri, Ken Chen, Rada Chirkova, Nitin Chopra, Bobbie Cochrane, Arturo Crespo, Linda DeMichiel, Tom Dienstbier, Pearl D'Souza, Oliver Duschka.

还有Xavier Faz, Greg Fichtenholtz, Bar Fisher, Jarl Friis, John Fry, Chiping Fu, Tracy Fujieda, Manish Godara, Meredith Goldsmith, Luis Gravano, Gerard Guillemette, Rafael Hernandez, Antti Hjelt, Ben Holtzman, Steve Huntsberry, Leonard Jacobson, Thulasiraman Jeyaraman, Dwight Joe, Seth Katz, Yeong-Ping Koh, Gyorgy Kovacs, Phillip Koza, Brian Kulman, Sang Ho Lee, Olivier Lobry, Lu Chao-Jun, Arun Marathe, Le-Wei Mo, Fabian Modoux, Peter Mork, Mark Mortensen.

还有Ramprakash Narayanaswami, Hankyung Na, Marie Nilsson, Torbjorn Norbye, Chang-Min Oh, Mehul Patel, Bert Porter, Limbek Reka, Prahash Ramanan, Ken Ross, Tim Roughgarten, Mema Roussopoulos, Richard Scherl, Catherine Tornabene, Anders Uhl, Jonathan Ullman, Mayank Upadhyay, Vassilis Vassalos, Qiang Wang, Kristian Widjaja, Janet Wu, Sundar Yamunachari, Takeshi Yokukawa, Min-Sig Yun, Torben Zahle, Sandy Zhang。当然书中的错误是我们的。

H. G.-M

J. D. U.

J. W.

斯坦福，加州

作者简介



Hector Garcia-Molina是斯坦福大学计算机科学与电气工程系的Leonard Bosack和Sandra Lerner教授。他在数据库系统、分布式系统和数字图书馆领域中发表了大量论文。他的研究兴趣包括分布式计算系统、数据库系统和数字图书馆。



Jeffrey D. Ullman是斯坦福大学的Stanford W. Ascherman计算机科学教授。他作为作者或合作者出版了15本著作，发表了170篇技术论文，其中包括《A First Course in Database Systems》(Prentice Hall 出版社，1997)和《Elements of ML Programming》(Prentice Hall 出版社，1998)。他的研究兴趣包括数据库理论、数据库集成、数据挖掘和利用信息基础设施进行教育。他获得了Guggenheim Fellowship等多种奖励，并被推选进入国家工程院。他还被授予1996年Sigmod贡献奖和1998年Karl V. Karstrom杰出教育家奖。



Jennifer Widom是斯坦福大学计算机科学与电气工程系的副教授。她是多个编辑委员会和程序委员会的成员，在计算机科学会议和杂志上发表了许多文章。她还是《A First Course in Database Systems》的作者之一。她的研究兴趣包括半结构化数据的数据库系统和XML、数据仓库以及主动数据库系统。

目　录

出版者的话
专家指导委员会
译者序
作者简介
前言
第1章　数据库系统世界 ……1
1.1　数据库系统的发展 ……1
1.1.1　早期的数据库管理系统 ……1
1.1.2　关系数据库系统 ……2
1.1.3　越来越小的系统 ……3
1.1.4　越来越大的系统 ……4
1.1.5　客户-服务器和多层体系结构 ……4
1.1.6　多媒体数据 ……5
1.1.7　信息集成 ……5
1.2　数据库管理系统概述 ……6
1.2.1　数据定义语言命令 ……6
1.2.2　查询处理概述 ……6
1.2.3　存储器和缓冲区管理器 ……8
1.2.4　事务处理 ……8
1.2.5　查询处理器 ……9
1.3　数据库系统研究概述 ……9
1.3.1　数据库设计 ……10
1.3.2　数据库程序设计 ……10
1.3.3　数据库系统实现 ……11
1.3.4　信息集成概述 ……12
1.4　小结 ……12
1.5　参考文献 ……12
第2章　实体-联系数据模型 ……15
2.1　E/R模型的要素 ……15
2.1.1　实体集 ……15
2.1.2　属性 ……16
2.1.3　联系 ……16
2.1.4　实体-联系图 ……16
2.1.5　E/R图实例 ……17
2.1.6　二元E/R联系的多样性 ……17
2.1.7　多路联系 ……18
2.1.8　联系中的角色 ……18
2.1.9　联系的属性 ……19
2.1.10　多路联系到二元联系的转换 ……20
2.1.11　E/R模型中的子类 ……21
2.1.12　习题 ……22
2.2　设计原则 ……24
2.2.1　忠实性 ……24
2.2.2　避免冗余 ……24
2.2.3　简单性考虑 ……25
2.2.4　选择正确的联系 ……25
2.2.5　选择正确的元素种类 ……26
2.2.6　习题 ……27
2.3　约束的建模 ……28
2.3.1　约束的分类 ……29
2.3.2　E/R模型中的键 ……29
2.3.3　E/R模型中键的表示 ……30
2.3.4　单值约束 ……31
2.3.5　引用完整性 ……31
2.3.6　E/R图中的引用完整性 ……31
2.3.7　其他类型的约束 ……32
2.3.8　习题 ……32
2.4　弱实体集 ……33
2.4.1　弱实体集的来源 ……33
2.4.2　弱实体集的要求 ……34
2.4.3　弱实体集的符号 ……35
2.4.4　习题 ……35
2.5　小结 ……35
2.6　参考文献 ……36
第3章　关系数据模型 ……37
3.1　关系模型的基础 ……37
3.1.1　属性 ……37
3.1.2　模式 ……37
3.1.3　元组 ……38
3.1.4　域 ……38
3.1.5　关系的等价描述 ……38
3.1.6　关系实例 ……38
3.1.7　习题 ……39

3.2 从E/R图到关系设计……39
3.2.1 实体集到关系的转化……40
3.2.2 E/R联系到关系的转化……41
3.2.3 组合关系……42
3.2.4 处理弱实体集……43
3.2.5 习题……45
3.3 子类结构到关系的转化……46
3.3.1 E/R方式转化……46
3.3.2 面向对象的方法……47
3.3.3 使用空值组合关系……48
3.3.4 各种方法的比较……48
3.3.5 习题……48
3.4 函数依赖……49
3.4.1 函数依赖的定义……50
3.4.2 关系的键……50
3.4.3 超键……52
3.4.4 找出关系中的键……52
3.4.5 习题……53
3.5 函数依赖的规则……54
3.5.1 分解/结合规则……54
3.5.2 平凡函数依赖……55
3.5.3 计算属性的闭包……55
3.5.4 为什么能用闭包算法……57
3.5.5 传递规则……58
3.5.6 函数依赖的闭包集合……59
3.5.7 投影函数依赖……59
3.5.8 习题……60
3.6 关系数据库模式设计……61
3.6.1 异常……62
3.6.2 分解关系……62
3.6.3 Boyce-Codd范式……63
3.6.4 分解为 BCNF……65
3.6.5 从分解中恢复信息……68
3.6.6 第三范式……69
3.6.7 习题……71
3.7 多值依赖……71
3.7.1 属性独立及伴随其产生的冗余……71
3.7.2 多值依赖的定义……72
3.7.3 多值依赖的推论……73
3.7.4 第四范式……74
3.7.5 分解到第四范式……75
3.7.6 范式间的联系……76
3.7.7 习题……76
3.8 小结……77
3.9 参考文献……78
第4章 其他数据模型……81
4.1 面向对象概念的复习……81
4.1.1 类型系统……82
4.1.2 类与对象……82
4.1.3 对象标识……82
4.1.4 方法……82
4.1.5 类的层次……83
4.2 ODL简介……83
4.2.1 面向对象设计……83
4.2.2 类声明……84
4.2.3 ODL中的属性……84
4.2.4 ODL中的联系……85
4.2.5 反向联系……86
4.2.6 联系的多重性……87
4.2.7 ODL中的方法……88
4.2.8 ODL中的类型……89
4.2.9 习题……90
4.3 ODL中的其他概念……91
4.3.1 ODL中的多路联系……92
4.3.2 ODL中的子类……92
4.3.3 ODL中的多继承……93
4.3.4 范围……94
4.3.5 ODL中键声明……94
4.3.6 习题……96
4.4 从ODL设计到关系设计……96
4.4.1 从ODL属性到关系属性……97
4.4.2 类中的非原子类型属性……97
4.4.3 集合类型属性的表示……98
4.4.4 其他类型构建器的表示……99
4.4.5 ODL中联系的表示……100
4.4.6 如果没有键会怎样?……102
4.4.7 习题……102
4.5 对象关系模型……103
4.5.1 从关系到对象关系……104
4.5.2 嵌套关系……104
4.5.3 引用……105
4.5.4 面向对象与对象关系的比较……106

4.5.5 ODL设计到对象关系设计的转化 …107
4.5.6 习题 …107
4.6 半结构化数据 …107
4.6.1 为何需要半结构化数据模型 …108
4.6.2 半结构化数据表示 …108
4.6.3 信息集成与半结构化数据 …109
4.6.4 习题 …110
4.7 XML及其数据模型 …110
4.7.1 语义标签 …111
4.7.2 格式规范的XML …111
4.7.3 文档类型定义（DTD） …112
4.7.4 使用DTD …113
4.7.5 属性列表 …114
4.7.6 习题 …115
4.8 小结 …116
4.9 参考文献 …116
第5章 关系代数 …119
5.1 一个数据库模式的例子 …119
5.2 关系代数操作 …120
5.2.1 关系代数基础 …121
5.2.2 关系中的集合操作 …121
5.2.3 投影 …122
5.2.4 选择 …123
5.2.5 笛卡儿积 …124
5.2.6 自然连接 …124
5.2.7 θ连接 …125
5.2.8 使用组合操作生成查询 …126
5.2.9 重命名 …127
5.2.10 依赖的和非依赖的操作 …128
5.2.11 关系代数表达式中的线性符号 …129
5.2.12 习题 …129
5.3 包上的关系操作 …134
5.3.1 为什么采用包 …134
5.3.2 包的并、交、差 …135
5.3.3 包的投影操作 …136
5.3.4 包的选择 …137
5.3.5 包的笛卡儿积 …137
5.3.6 包的连接 …137
5.3.7 习题 …138
5.4 关系代数的扩展操作 …139
5.4.1 消除重复 …139
5.4.2 聚集操作符 …140
5.4.3 分组 …140
5.4.4 分组操作符 …141
5.4.5 扩展的投影操作符 …142
5.4.6 排序操作符 …143
5.4.7 外连接 …143
5.4.8 习题 …144
5.5 关系的约束 …145
5.5.1 作为约束语言的关系代数 …145
5.5.2 引用完整性约束 …146
5.5.3 其他的约束举例 …147
5.5.4 习题 …148
5.6 小结 …149
5.7 参考文献 …149
第6章 数据库语言SQL …151
6.1 SQL中的简单查询 …151
6.1.1 SQL中的投影 …152
6.1.2 SQL中的选择 …154
6.1.3 字符串比较 …155
6.1.4 日期和时间 …156
6.1.5 空值和涉及空值的比较 …157
6.1.6 布尔值UNKNOWN …158
6.1.7 输出排序 …159
6.1.8 习题 …159
6.2 多个关系上的查询 …160
6.2.1 SQL中的积和连接 …161
6.2.2 避免属性歧义 …161
6.2.3 元组变量 …162
6.2.4 多关系查询的解释 …163
6.2.5 查询的并、交、差 …165
6.2.6 习题 …166
6.3 子查询 …167
6.3.1 产生标量值的子查询 …167
6.3.2 含有关系的条件表达式 …168
6.3.3 含有元组的条件表达式 …169
6.3.4 关联子查询 …170
6.3.5 FROM子句中的子查询 …171
6.3.6 SQL 的连接表达式 …171
6.3.7 自然连接 …172
6.3.8 外连接 …173
6.3.9 习题 …174

6.4 全关系操作 ……176
6.4.1 消除重复 ……176
6.4.2 交、并、差中的重复 ……176
6.4.3 SQL 中的分组和聚集 ……177
6.4.4 聚集操作符 ……177
6.4.5 分组 ……178
6.4.6 HAVING子句 ……179
6.4.7 习题 ……180
6.5 数据库更新 ……181
6.5.1 插入 ……181
6.5.2 删除 ……183
6.5.3 更新 ……183
6.5.4 习题 ……184
6.6 SQL中的关系模式定义 ……185
6.6.1 数据类型 ……185
6.6.2 简单表定义 ……186
6.6.3 修改关系模式 ……186
6.6.4 默认值 ……187
6.6.5 索引 ……187
6.6.6 索引选择简介 ……188
6.6.7 习题 ……190
6.7 视图定义 ……191
6.7.1 视图声明 ……191
6.7.2 视图查询 ……192
6.7.3 重命名属性 ……193
6.7.4 视图更新 ……193
6.7.5 涉及视图的查询解释 ……195
6.7.6 习题 ……197
6.8 小结 ……197
6.9 参考文献 ……198
第7章 约束和触发器 ……201
7.1 键和外键 ……201
7.1.1 主键声明 ……201
7.1.2 用UNIQUE声明键 ……202
7.1.3 强制键约束 ……203
7.1.4 外键约束声明 ……203
7.1.5 维护引用完整性 ……204
7.1.6 延迟约束检查 ……205
7.1.7 习题 ……207
7.2 属性和元组上的约束 ……208
7.2.1 非空值约束 ……208
7.2.2 基于属性的 CHECK 约束 ……209
7.2.3 基于元组的 CHECK 约束 ……210
7.2.4 习题 ……211
7.3 修改约束 ……212
7.3.1 给约束命名 ……212
7.3.2 修改表上约束 ……212
7.3.3 习题 ……213
7.4 模式层的约束和触发器 ……214
7.4.1 断言 ……214
7.4.2 事件-条件-动作规则 ……216
7.4.3 SQL中的触发器 ……216
7.4.4 替换触发器（Instead of Triggers）…219
7.4.5 习题 ……219
7.5 小结 ……221
7.6 参考文献 ……221
第8章 SQL 的系统特征 ……223
8.1 编程环境下的 SQL ……223
8.1.1 阻抗不匹配问题 ……224
8.1.2 SQL/宿主语言接口 ……224
8.1.3 DECLARE节 ……225
8.1.4 使用共享变量 ……225
8.1.5 单元组选择语句 ……226
8.1.6 游标 ……226
8.1.7 游标修改 ……229
8.1.8 防止并发更新 ……229
8.1.9 卷型游标 ……230
8.1.10 动态SQL ……231
8.1.11 习题 ……232
8.2 模式中的存储过程 ……233
8.2.1 创建PSM函数和过程 ……233
8.2.2 PSM中的简单语句格式 ……234
8.2.3 分支语句 ……235
8.2.4 PSM中的查询 ……236
8.2.5 PSM中的循环 ……236
8.2.6 For 循环 ……238
8.2.7 PSM的异常处理 ……238
8.2.8 使用PSM函数和过程 ……240
8.2.9 习题 ……240
8.3 SQL 环境 ……242
8.3.1 环境 ……242
8.3.2 模式 ……242

8.3.3 目录 ……243
8.3.4 SQL 环境中的客户和服务器……244
8.3.5 连接 ……244
8.3.6 会话 ……245
8.3.7 模块 ……245
8.4 使用调用层接口 ……245
8.4.1 SQL/CLI简介 ……246
8.4.2 处理语句 ……247
8.4.3 从查询结果中取数据 ……248
8.4.4 向查询传递参数 ……250
8.4.5 习题 ……250
8.5 Java数据库连接 ……250
8.5.1 JDBC 简介 ……250
8.5.2 JDBC 中的创建语句……251
8.5.3 JDBC 中的游标操作……252
8.5.4 参数传递 ……252
8.5.5 习题 ……253
8.6 SQL 中的事务 ……253
8.6.1 可串行性 ……253
8.6.2 原子性 ……255
8.6.3 事务 ……256
8.6.4 只读事务 ……257
8.6.5 读脏数据 ……258
8.6.6 其他隔离级别 ……259
8.6.7 习题 ……260
8.7 SQL 中的安全机制和用户认证 ……261
8.7.1 权限 ……261
8.7.2 创建权限 ……262
8.7.3 检查权限的处理 ……263
8.7.4 授权 ……264
8.7.5 授权图 ……265
8.7.6 销权 ……266
8.7.7 习题 ……268
8.8 小结 ……269
8.9 参考文献 ……270
第9章 面向对象查询语言 ……271
9.1 OQL简介 ……271
9.1.1 一个面向对象的电影例子……271
9.1.2 路径表达式 ……271
9.1.3 OQL 中 Select-From-Where 表达式 ……273
9.1.4 修改结果的类型 ……273
9.1.5 复杂输出类型 ……274
9.1.6 子查询 ……275
9.1.7 习题 ……276
9.2 OQL 表达式的其他格式……278
9.2.1 量词表达式 ……278
9.2.2 聚集表达式 ……279
9.2.3 分组表达式 ……279
9.2.4 HAVING 子句 ……281
9.2.5 并、交和差操作 ……281
9.2.6 习题 ……282
9.3 OQL 中对象的赋值与创建 ……283
9.3.1 宿主语言变量的赋值 ……283
9.3.2 集合元素的提取 ……283
9.3.3 获取集合的每一个成员 ……283
9.3.4 OQL 中的常量 ……284
9.3.5 创建新对象 ……285
9.3.6 习题 ……286
9.4 SQL 中的用户定义类型 ……286
9.4.1 在SQL 中定义类型 ……286
9.4.2 用户定义类型中的方法 ……287
9.4.3 用UDT声明关系 ……288
9.4.4 引用 ……288
9.4.5 习题 ……290
9.5 对象关系数据上的操作 ……290
9.5.1 引用的跟随（Following Reference）……290
9.5.2 访问UDT类型元组的属性……291
9.5.3 生成器和转换器函数 ……292
9.5.4 UDT类型联系的排序 ……293
9.5.5 习题 ……294
9.6 小结 ……295
9.7 参考文献 ……295
第10章 逻辑查询语言 ……297
10.1 一种关系逻辑 ……297
10.1.1 谓词和原子……297
10.1.2 算术原子 ……297
10.1.3 Datalog 规则和查询 ……298
10.1.4 Datalog 规则的意义 ……299
10.1.5 扩展谓词和内涵谓词 ……300
10.1.6 Datalog规则应用于包 ……301
10.1.7 习题 ……302

10.2 从关系代数到Datalog ……302
10.2.1 交 ……302
10.2.2 并 ……302
10.2.3 差 ……303
10.2.4 投影 ……303
10.2.5 选择 ……303
10.2.6 积 ……305
10.2.7 连接 ……305
10.2.8 用 Datalog 模拟多重操作 ……306
10.2.9 习题 ……307
10.3 Datalog 的递归编程 ……308
10.3.1 递归规则 ……309
10.3.2 计算递归Datalog 规则 ……309
10.3.3 递归规则中的非 ……313
10.3.4 习题 ……315
10.4 SQL 中的递归 ……316
10.4.1 在SQL 中定义IDB关系 ……316
10.4.2 分层非 ……318
10.4.3 有问题的递归SQL表达式 ……319
10.4.4 习题 ……321
10.5 小结 ……322
10.6 参考文献 ……322
第11章 数据存储 ……325
11.1 Megatron 2002数据库系统 ……325
11.1.1 Megatron 2002实现细节 ……325
11.1.2 Megatron 2002如何执行查询 ……326
11.1.3 Megatron 2002有什么问题 ……327
11.2 存储器层次 ……327
11.2.1 高速缓冲存储器 ……327
11.2.2 主存储器 ……328
11.2.3 虚拟存储器 ……329
11.2.4 二级存储器 ……329
11.2.5 三级存储器 ……330
11.2.6 易失和非易失存储器 ……332
11.2.7 习题 ……332
11.3 磁盘 ……332
11.3.1 磁盘结构 ……332
11.3.2 磁盘控制器 ……334
11.3.3 磁盘存储特性 ……334
11.3.4 磁盘访问特性 ……335
11.3.5 块的写操作 ……338
11.3.6 块的修改 ……338
11.3.7 习题 ……338
11.4 有效使用二级存储器 ……339
11.4.1 计算的I/O模型 ……339
11.4.2 二级存储器中的数据排序 ……340
11.4.3 归并排序 ……341
11.4.4 两趟多路归并排序 ……342
11.4.5 更大型关系的多路归并 ……343
11.4.6 习题 ……344
11.5 加速二级存储的访问 ……345
11.5.1 按柱面组织数据 ……346
11.5.2 使用多个磁盘 ……346
11.5.3 磁盘镜像 ……347
11.5.4 磁盘调度和电梯算法 ……348
11.5.5 预取和大规模缓冲 ……350
11.5.6 对策略和折中的小结 ……351
11.5.7 习题 ……352
11.6 磁盘故障 ……353
11.6.1 间断性故障 ……353
11.6.2 校验和 ……353
11.6.3 稳定存储 ……354
11.6.4 稳定存储的错误处理能力 ……354
11.6.5 习题 ……355
11.7 从磁盘崩溃中恢复 ……355
11.7.1 磁盘的故障模型 ……355
11.7.2 镜像冗余技术 ……356
11.7.3 奇偶块 ……356
11.7.4 一种改进：RAID 5 ……359
11.7.5 多个盘崩溃时的处理 ……359
11.7.6 习题 ……361
11.8 小结 ……363
11.9 参考文献 ……364
第12章 数据元素的表示 ……365
12.1 数据元素和字段 ……365
12.1.1 关系数据库元素的表示 ……365
12.1.2 对象的表示 ……366
12.1.3 数据元素的表示 ……366
12.2 记录 ……368
12.2.1 定长记录的构造 ……368
12.2.2 记录首部 ……370
12.2.3 定长记录在块中的放置 ……371
12.2.4 习题 ……371
12.3 块和记录地址的表示 ……372

12.3.1 客户-服务器系统 ……372
12.3.2 逻辑地址和结构地址 ……373
12.3.3 指针混写 ……374
12.3.4 块返回磁盘 ……376
12.3.5 被固定的记录和块 ……377
12.3.6 习题 ……378
12.4 变长数据和记录 ……379
12.4.1 具有变长字段的记录 ……379
12.4.2 具有重复字段的记录 ……380
12.4.3 可变格式记录 ……381
12.4.4 不能装入一个块中的记录 ……382
12.4.5 BLOBS ……383
12.4.6 习题 ……383
12.5 记录的修改 ……384
12.5.1 插入 ……384
12.5.2 删除 ……385
12.5.3 更新 ……386
12.5.4 习题 ……386
12.6 小结 ……387
12.7 参考文献 ……387
第13章 索引结构 ……389
13.1 顺序文件上的索引 ……389
13.1.1 顺序文件 ……390
13.1.2 稠密索引 ……390
13.1.3 稀疏索引 ……391
13.1.4 多级索引 ……392
13.1.5 重复查找键的索引 ……393
13.1.6 数据修改期间的索引维护 ……395
13.1.7 习题 ……398
13.2 辅助索引 ……399
13.2.1 辅助索引的设计 ……399
13.2.2 辅助索引的应用 ……400
13.2.3 辅助索引的间接性 ……401
13.2.4 文档检索和倒排索引 ……403
13.2.5 习题 ……405
13.3 B树 ……406
13.3.1 B树的结构 ……406
13.3.2 B树的应用 ……409
13.3.3 B树中的查找 ……410
13.3.4 范围查询 ……410
13.3.5 B树的插入 ……411
13.3.6 B树的删除 ……413
13.3.7 B树的效率 ……415
13.3.8 习题 ……416
13.4 散列表 ……417
13.4.1 辅存散列表 ……418
13.4.2 散列表的插入 ……418
13.4.3 散列表的删除 ……419
13.4.4 散列表索引的效率 ……419
13.4.5 可扩展散列表 ……419
13.4.6 可扩展散列表的插入 ……420
13.4.7 线性散列表 ……421
13.4.8 线性散列表的插入 ……422
13.4.9 习题 ……423
13.5 小结 ……425
13.6 参考文献 ……425
第14章 多维索引和位图索引 ……427
14.1 需要多维的应用 ……427
14.1.1 地理信息系统 ……428
14.1.2 数据立方体 ……428
14.1.3 SQL多维查询 ……428
14.1.4 使用传统索引执行范围查询 ……430
14.1.5 利用传统索引执行最邻近查询 ……430
14.1.6 传统索引的其他限制 ……431
14.1.7 多维索引结构综述 ……432
14.1.8 习题 ……432
14.2 多维数据的类散列结构 ……433
14.2.1 网格文件 ……433
14.2.2 网格文件的查找 ……434
14.2.3 网格文件的插入 ……434
14.2.4 网格文件的性能 ……435
14.2.5 分段散列函数 ……436
14.2.6 网格文件和分段散列的比较 ……438
14.2.7 习题 ……438
14.3 多维数据的树形结构 ……440
14.3.1 多键索引 ……440
14.3.2 多键索引的性能 ……441
14.3.3 kd树 ……441
14.3.4 kd树的操作 ……442
14.3.5 使kd树适合辅存 ……444

14.3.6 四叉树 ……444
14.3.7 R树 ……446
14.3.8 R树的操作 ……446
14.3.9 习题 ……448
14.4 位图索引 ……449
14.4.1 位图索引的诱因 ……449
14.4.2 压缩位图 ……451
14.4.3 游程长度编码位向量的操作 ……452
14.4.4 位图索引的管理 ……452
14.4.5 习题 ……454
14.5 小结 ……454
14.6 参考文献 ……455
第15章 查询执行 ……457
15.1 物理查询计划操作符介绍 ……458
15.1.1 扫描表 ……458
15.1.2 扫描表时的排序 ……459
15.1.3 物理操作符计算模型 ……459
15.1.4 衡量代价的参数 ……459
15.1.5 扫描操作符的I/O代价 ……460
15.1.6 实现物理操作符的迭代器 ……461
15.2 数据库操作的一趟算法 ……463
15.2.1 一次多元组操作的一趟算法 ……464
15.2.2 全关系的一元操作的一趟算法 ……464
15.2.3 二元操作的一趟算法 ……466
15.2.4 习题 ……468
15.3 嵌套循环连接 ……469
15.3.1 基于元组的嵌套循环连接 ……469
15.3.2 基于元组的嵌套循环连接的迭代器 ……470
15.3.3 基于块的嵌套循环连接算法 ……470
15.3.4 嵌套循环连接的分析 ……471
15.3.5 迄今为止的算法小结 ……472
15.3.6 习题 ……472
15.4 基于排序的两趟算法 ……472
15.4.1 利用排序消除重复 ……473
15.4.2 利用排序进行分组和聚集 ……474
15.4.3 基于排序的并算法 ……475
15.4.4 基于排序的交和差算法 ……475
15.4.5 基于排序的一个简单的连接算法 ……476
15.4.6 简单排序连接的分析 ……478
15.4.7 一种更有效的基于排序的连接 ……478
15.4.8 基于排序的算法小结 ……479
15.4.9 习题 ……479
15.5 基于散列的两趟算法 ……480
15.5.1 通过散列划分关系 ……481
15.5.2 基于散列的消除重复算法 ……481
15.5.3 基于散列的分组和聚集算法 ……481
15.5.4 基于散列的并、交、差算法 ……482
15.5.5 散列连接算法 ……482
15.5.6 节省一些磁盘I/O ……483
15.5.7 基于散列的算法小结 ……484
15.5.8 习题 ……485
15.6 基于索引的算法 ……485
15.6.1 聚簇和非聚簇索引 ……485
15.6.2 基于索引的选择 ……486
15.6.3 使用索引的连接 ……488
15.6.4 使用有排序索引的连接 ……488
15.6.5 习题 ……489
15.7 缓冲区管理 ……490
15.7.1 缓冲区管理结构 ……491
15.7.2 缓冲区管理策略 ……491
15.7.3 物理操作符选择和缓冲区管理的关系 ……493
15.7.4 习题 ……494
15.8 使用超过两趟的算法 ……494
15.8.1 基于排序的多趟算法 ……494
15.8.2 基于排序的多趟算法的性能 ……495
15.8.3 基于散列的多趟算法 ……495
15.8.4 基于散列的多趟算法的性能 ……496
15.8.5 习题 ……496
15.9 关系操作的并行算法 ……497
15.9.1 并行模型 ……497
15.9.2 一次一个元组的并行操作 ……498
15.9.3 全关系操作的并行算法 ……499
15.9.4 并行算法的性能 ……500
15.9.5 习题 ……501
15.10 小结 ……502
15.11 参考文献 ……503
第16章 查询编译器 ……505

16.1 语法分析……505
16.1.1 语法分析与语法分析树 ……505
16.1.2 SQL的一个简单子集的语法 ……506
16.1.3 预处理器 ……508
16.1.4 习题 ……509
16.2 用于改进查询计划的代数定律 ……510
16.2.1 交换律与结合律 ……510
16.2.2 涉及选择的定律 ……512
16.2.3 下推选择 ……513
16.2.4 涉及投影的定律 ……514
16.2.5 有关连接与积的定律 ……516
16.2.6 有关消除重复的定律 ……517
16.2.7 涉及分组与聚集的定律 ……517
16.2.8 习题 ……518
16.3 从语法分析树到逻辑查询计划 ……520
16.3.1 转换成关系代数 ……520
16.3.2 从条件中去除子查询 ……520
16.3.3 逻辑查询计划的改进 ……524
16.3.4 结合/交换操作符的分组 ……525
16.3.5 习题 ……525
16.4 操作代价的估计 ……526
16.4.1 中间关系大小的估计 ……526
16.4.2 投影大小的估计 ……527
16.4.3 估计选择的大小 ……527
16.4.4 连接大小的估计 ……529
16.4.5 多连接属性的自然连接 ……531
16.4.6 多个关系的连接 ……532
16.4.7 其他操作的大小估计 ……533
16.4.8 习题 ……534
16.5 基于代价的计划选择介绍 ……535
16.5.1 大小参数估计值的获取 ……535
16.5.2 计算统计量 ……538
16.5.3 减少逻辑查询计划代价的启发式 ……538
16.5.4 物理计划的枚举技术 ……539
16.5.5 习题 ……541
16.6 连接顺序的选择 ……543
16.6.1 连接的左右变元的意义 ……543
16.6.2 连接树 ……543
16.6.3 左深连接树 ……544
16.6.4 通过动态规划来选择连接顺序和分组 ……546
16.6.5 带有更具体的代价函数的动态规划法 ……549
16.6.6 选择连接顺序的贪婪算法 ……549
16.6.7 习题 ……550
16.7 物理查询计划选择的完成 ……551
16.7.1 选取选择方法 ……551
16.7.2 选取连接方法 ……552
16.7.3 流水线操作与实体化 ……553
16.7.4 一元流水线操作 ……553
16.7.5 二元流水线操作 ……554
16.7.6 物理查询计划的符号 ……556
16.7.7 物理操作的顺序 ……557
16.7.8 习题 ……558
16.8 小结 ……559
16.9 参考文献 ……560
第17章 系统故障对策 ……561
17.1 可回复操作的问题和模型 ……561
17.1.1 故障模式 ……561
17.1.2 关于事务的进一步讨论 ……562
17.1.3 事务的正确执行 ……563
17.1.4 事务的原语操作 ……564
17.1.5 习题 ……566
17.2 undo日志 ……566
17.2.1 日志记录 ……567
17.2.2 undo日志规则 ……568
17.2.3 使用undo日志的恢复 ……569
17.2.4 检查点 ……571
17.2.5 非静止检查点 ……571
17.2.6 习题 ……573
17.3 redo日志 ……574
17.3.1 redo日志规则 ……574
17.3.2 使用redo日志的恢复 ……575
17.3.3 redo日志的检查点 ……576
17.3.4 使用带检查点的redo日志的恢复 ……577
17.3.5 习题 ……577
17.4 undo/redo日志 ……578
17.4.1 undo/redo规则 ……578

17.4.2 使用undo/redo日志的恢复 ……579
17.4.3 undo/redo日志的检查点 ……579
17.4.4 习题 ……580
17.5 防备介质故障 ……581
17.5.1 备份 ……581
17.5.2 非静止转储 ……582
17.5.3 使用备份和日志的恢复 ……583
17.5.4 习题 ……584
17.6 小结 ……584
17.7 参考文献 ……585
第18章 并发控制 ……587
18.1 串行调度和可串行化调度 ……587
18.1.1 调度 ……587
18.1.2 串行调度 ……588
18.1.3 可串行化调度 ……588
18.1.4 事务语义的影响 ……589
18.1.5 事务和调度的一种记法 ……590
18.1.6 习题 ……590
18.2 冲突可串行性 ……591
18.2.1 冲突 ……591
18.2.2 优先图及冲突可串行性判断 ……592
18.2.3 优先图测试发挥作用的原因 ……593
18.2.4 习题 ……594
18.3 使用锁的可串行性实现 ……595
18.3.1 锁 ……596
18.3.2 封锁调度器 ……597
18.3.3 两阶段封锁 ……598
18.3.4 两阶段封锁发挥作用的原因 ……598
18.3.5 习题 ……599
18.4 用多种锁方式的封锁系统 ……600
18.4.1 共享锁与排他锁 ……601
18.4.2 相容性矩阵 ……602
18.4.3 锁的升级 ……602
18.4.4 更新锁 ……603
18.4.5 增量锁 ……604
18.4.6 习题 ……605
18.5 封锁调度器的一种体系结构 ……607
18.5.1 插入锁动作的调度器 ……607
18.5.2 锁表 ……609
18.5.3 习题 ……611
18.6 数据库元素层次的管理 ……611
18.6.1 多粒度的锁 ……611
18.6.2 警示锁 ……612
18.6.3 幻像与插入的正确处理 ……613
18.6.4 习题 ……614
18.7 树协议 ……615
18.7.1 基于树的封锁的动机 ……615
18.7.2 访问树结构数据的规则 ……615
18.7.3 树协议发挥作用的原因 ……616
18.7.4 习题 ……618
18.8 使用时间戳的并发控制 ……618
18.8.1 时间戳 ……619
18.8.2 物理上不可实现的行为 ……619
18.8.3 脏数据的问题 ……620
18.8.4 基于时间戳调度的规则 ……621
18.8.5 多版本时间戳 ……622
18.8.6 时间戳与封锁 ……623
18.8.7 习题 ……623
18.9 使用有效性确认的并发控制 ……624
18.9.1 基于有效性确认的调度器的结构 ……624
18.9.2 有效性确认规则 ……625
18.9.3 三种并发控制机制的比较 ……627
18.9.4 习题 ……627
18.10 小结 ……628
18.11 参考文献 ……629
第19章 再论事务管理 ……631
19.1 读未提交数据的事务 ……631
19.1.1 脏数据问题 ……631
19.1.2 级联回滚 ……632
19.1.3 可恢复调度 ……633
19.1.4 避免级联回滚的调度 ……633
19.1.5 回滚的管理 ……634
19.1.6 成组提交 ……635
19.1.7 逻辑日志 ……636
19.1.8 根据逻辑日志恢复 ……637
19.1.9 习题 ……638
19.2 视图可串行性 ……639
19.2.1 视图等价性 ……639
19.2.2 多重图与视图可串行性的判断 ……640

19.2.3 视图可串行性的判断 ……643
19.2.4 习题 ……643
19.3 死锁处理 ……643
19.3.1 超时死锁检测 ……644
19.3.2 等待图 ……644
19.3.3 通过元素排序预防死锁 ……645
19.3.4 时间戳死锁检测 ……646
19.3.5 死锁管理方法的比较 ……648
19.3.6 习题 ……649
19.4 分布式数据库 ……649
19.4.1 数据的分布 ……649
19.4.2 分布式事务 ……650
19.4.3 数据复制 ……651
19.4.4 分布式查询优化 ……651
19.4.5 习题 ……652
19.5 分布式提交 ……652
19.5.1 分布式原子性的支持 ……652
19.5.2 两阶段提交 ……653
19.5.3 分布式事务的恢复 ……654
19.5.4 习题 ……655
19.6 分布式封锁 ……656
19.6.1 集中封锁系统 ……656
19.6.2 分布式封锁算法的代价模型 ……656
19.6.3 封锁多副本的元素 ……657
19.6.4 主副本封锁 ……658
19.6.5 局部锁构成的全局锁 ……658
19.6.6 习题 ……659
19.7 长事务 ……659
19.7.1 长事务的问题 ……659
19.7.2 saga（系列记载） ……661
19.7.3 补偿事务 ……662
19.7.4 补偿事务发挥作用的原因 ……663
19.7.5 习题 ……663
19.8 小结 ……663
19.9 参考文献 ……665
第20章 信息集成 ……667
20.1 信息集成的方式 ……667
20.1.1 信息集成的问题 ……667
20.1.2 联邦数据库系统 ……668
20.1.3 数据仓库 ……669
20.1.4 协调器 ……671
20.1.5 习题 ……673
20.2 基于协调器系统的包装器 ……674
20.2.1 查询模式的模板 ……674
20.2.2 包装器生成器 ……675
20.2.3 过滤器 ……676
20.2.4 其他在包装器上进行的操作 ……677
20.2.5 习题 ……678
20.3 协调器基于能力的优化 ……678
20.3.1 数据源能力有限的问题 ……678
20.3.2 描述数据源能力的符号 ……679
20.3.3 基于能力的查询计划选择 ……680
20.3.4 增加基于代价的优化 ……681
20.3.5 习题 ……681
20.4 联机分析处理 ……682
20.4.1 OLAP应用 ……682
20.4.2 OLAP数据的多维视图 ……683
20.4.3 星型模式 ……684
20.4.4 切片和切块 ……685
20.4.5 习题 ……687
20.5 数据立方体 ……687
20.5.1 立方体操作符 ……687
20.5.2 通过物化视图实现立方体 ……689
20.5.3 视图的格 ……691
20.5.4 习题 ……692
20.6 数据挖掘 ……693
20.6.1 数据挖掘的应用 ……694
20.6.2 寻找频繁项目集 ……695
20.6.3 A-Priori算法 ……696
20.6.4 习题 ……698
20.7 小结 ……698
20.8 参考文献 ……699
索引 ……701

第1章 数据库系统世界

今天数据库已是每一项业务的基础。数据库被应用于维护商业内部记录，在万维网上为顾客和客户显示数据，以及支持很多其他商业处理。数据库同样出现在很多科学研究的核心中。天文学家、人类基因研究者和探索蛋白质医药性质的生化学家，以及其他很多科学家搜集的数据也是用数据库表示的。

数据库的能力来自于已发展了数十年的知识和技术，这些知识和技术蕴藏在被称做*数据库管理系统*（database management system）的专业化软件中。该软件也被称做DBMS，或更通俗地称为“数据库系统”。DBMS是一个能有效建立和管理大量数据的强大工具，并且能安全地长期保存这些数据。数据库系统是最复杂的软件系统之一。DBMS为用户提供的功能如下：

1. *持久存储* 如同文件系统，DBMS支持对独立于应用的超大量数据的存储。然而，DBMS不仅只是在灵活性上优于文件系统，而且在数据结构上，还支持对超大量数据的有效访问。

2. *程序设计接口* DBMS允许用户或应用程序通过强有力的查询语言对数据进行访问和修改。而且，DBMS比文件系统更具优点，它不仅仅提供对文件的读写，而且还具备以更复杂的方式管理数据的灵活性。

3. *事务管理* DBMS支持数据的并发存取，即可以同时有很多不同的进程（称做“事务”）对数据访问。为了避免同时存取产生不期望的结果，DBMS要支持*独立性*（isolation）——一次执行一个事务，*原子性*（atomicity）——事务要么全部执行要么全部不执行。DBMS还支持*持久性*（durability），即具有能从很多类故障和错误中恢复的能力。

1.1 数据库系统的发展

数据库是什么？本质上讲，数据库就是信息的集合。这种集合可以存在很长时间，通常是很多年。一般讲，数据库是指由DBMS管理的数据的集合。DBMS需要有如下功能：

1. 允许用户使用专门的*数据定义语言*（data-definition language）建立新的数据库，并说明它们的*模式*（schema），即数据的逻辑结构。

2. 使用合适的*查询语言*（query language）或*数据操作语言*（data-manipulation language），为用户提供*查询*（query）和*更新*（modify）数据的能力。“查询”是数据库关于数据的提问的术语。

3. 支持超大数据量（吉字节或更多）数据的长时间存储，防止对数据意外的或非授权的访问，并且在数据库查询和更新时支持对数据的有效存取。

4. 控制多个用户对数据的立即存取，不允许一个用户的操作影响另一个用户，也不允许同时存取对数据的意外破坏。

1.1.1 早期的数据库管理系统

第一个商用数据库管理系统出现在20世纪60年代末。这些系统都是来自于文件系统，它们提供某些上述第3项功能；文件系统可以长期地存储数据，并且允许大数据量存储，可是，如果数据不做备份，文件系统通常并不保证数据不会丢失。当不知道数据项在某个文件中的存储位置时，文件系统也不提供数据的有效访问。

更进一步说，文件系统不直接支持第2项功能，即没有对文件的查询语言。对第1项功能的

⊖ 边栏数字为原书页码，此页码数与索引所示页码数呼应。

支持（即数据模式的支持）也只限于文件目录结构的建立。最后，文件系统不满足第4项功能。当有多个用户或进程对文件并发访问时，文件系统不能防止两个用户对同一个数据的修改，于
2 是将出现一个用户的修改被丢失的情形。

在DBMS第一批重要的应用中，数据由很多小数据项组成，并且完成很多查询或修改。下面是一些例子。

飞机订票系统

这类系统通常包括如下数据项：

1. 单个旅客预定某个航班的机票，包括的信息有座位分配、饮食习惯等。
2. 关于航班的信息，有飞机起飞和到达的机场、飞机起飞和到达时间或运营的飞机等等。
3. 有关机票价格、机票需求和剩余机票等数据。

典型的查询有：在某个时间范围内从一个城市到另一个城市的航班，有什么样的舱位，价格如何。典型的数据修改包括：旅客航班预定、坐席分配或注明饮食习惯等。在任一给定时间，很多代理机构将访问上述部分数据。DBMS必须允许这样的并发存取，并防止两个代理将同一个坐席同时分配给两个旅客这样的问题发生。当系统突然发生故障时还要保护数据不丢失。

银行系统

银行系统包括的数据有：顾客姓名、地址、账号、贷款、余额，以及顾客和他们的账号与贷款间的关联，例如，谁在哪个账号上有签字权。关于账目余额的查询是最常见的应用，但更常见的应用是修改，表现为对某个账号的付款或存款。

如同在飞机订票系统中一样，人们期望很多出纳员和顾客（通过ATM机或Web）能即时查询和修改银行数据。同时对一个账号的访问不会引起事务丢失的作用也极其重要。故障是不能容忍的。例如，一旦钱已从ATM机中弹出，即使是立即发生电源断电，银行也必须记录下这笔借方账目。另一方面，如果银行已在借方记账，但由于断电而未能付款，这种情况也是不允许的。处理这类操作的恰当方法并不那么简单，它被看做是DBMS方式的优点之一。

公司资料/数据

很多早期的应用涉及公司资料/数据，例如每次的销售记录、收支账目或职工信息（如
3 他们的名字、地址、工资、福利选择、缴税状况，等等）。查询应用包括：打印应收款项、职工每周的薪金。每次销售、采购、账单、收据、职工雇用、辞退及晋升等，都导致数据库的修改。

早期的DBMS是从文件系统发展而来，它助长用户用接近数据的方式显现数据。这些数据库系统使用多个不同的数据模型描述数据库中的信息结构，主要的有基于树结构的模型、“层次”模型和基于图结构的“网状”模型。网状模型通过CODASYL（数据系统及语言委员会）的报告在20世纪60年代末被标准化[⊖]。

早期模型和系统的一个问题是它们不支持高级查询语言。例如，CODASYL查询语言的语句只允许用户通过指向数据元素的指针，从一个数据元素跳到另一个数据元素。因此，即使是写一个非常简单的查询程序，用户也要花费很大的力气。

1.1.2 关系数据库系统

随着1970年Ted Codd著名论文的发表[⊖]，数据库系统有了重大的改变。Codd提出数据库系统应该将数据组织成表的形式呈现给用户。这种形式称做关系（relation）。在关系的后面，可

⊖ CODASYL “Data Base Task Group” April. 1971报告,ACM, New York。

⊖ Codd, E. F.,“大量共享数据库的关系模型” Comm. ACM 13:6 pp. 377-387。

能是一个复杂的数据结构，实现对各种查询问题的快速响应。但是，不同于早期数据库系统的用户，关系系统的用户将不必关心数据的存储结构，查询可以用非常高级的语言表述，因此可以极大地增加数据库程序员的工作效率。

本书从第3章开始介绍基本关系概念，书中的大部分内容也都与关系数据模型相关。从第6章开始将介绍SQL（“结构化查询语言”），它是最重要的基于关系模型的查询语言。对关系的简要介绍，将使读者对该模型的简洁性有个初步了解，下面的SQL例子，也将使读者了解到关系模型是如何用非常高级的语言书写查询的，从而绕开了数据库“航行”的细节。

例1.1 关系是表，它的列以属性（attribute）为标题，属性描述列中的项。例如，记录银行账户的账号、余额和类型的Accounts关系，其结构如下：

accountNo	*balance*	*type*
12345	1000.00	savings
67890	2846.92	checking
...	...	...

4

这里列标题是三个属性：accountNo（账号）、balance（余额）和type（类型）。属性下是行，或称做元组（tuple）。上例中显示出了两个元组值，元组之下的省略号表示还有更多的元组。每个元组对应银行中的一个账号。第一个元组是说账号12345有余额1000.00美元，它是储蓄型账号。第二个元组的账号是67890，其余额是2846.92美元，是支票型账号。

假如想知道账号67890的余额，可以用SQL查询如下：

```
SELECT balance
FROM Accounts
WHERE accountNo = 67890;
```

另一个例子，查询所有余额为负值的储蓄账号的语句是：

```
SELECT accountNo
FROM Accounts
WHERE type = 'savings' AND balance < 0;
```

上述两个例子并不足以使读者成为一个SQL专家级程序员。但是，这两个例子表达了SQL语言的“select-from-where”语句特征。原则上，以上例子要求DBMS完成如下动作：

1. 检查FROM子句中提到的Accounts关系的所有元组；
2. 选出所有满足WHERE子句中给出条件的元组；
3. 从选出的元组中，以SELECT子句指明的属性作为查询结果。

实践中，系统必须“优化”查询，寻找一个有效的方法回答查询，尽管查询中涉及的关系可能非常大。 □

到1990年，关系数据库系统已成为标准。但是，数据库领域继续在发展，数据管理的新课题、新方法不断出现。本节后续部分将讨论数据库系统的某些新趋势。

1.1.3 越来越小的系统

最初，DBMS是运行在大型计算机上的既庞大又昂贵的软件系统。因为存储吉字节（gigabyte）数据需要大的计算机系统，所以大容量是必需的。如今，单个磁盘的容量就可达若干吉字节，DBMS运行在个人计算机上已成为可能。因此，关系模型数据库可以在非常小的机器上运行。而且，如同以前的电子表格和字处理系统，关系数据库正在成为计算机应用的普通工具。

5

1.1.4 越来越大的系统

另一方面，吉字节数据量还不够大。一个公司的数据库常常有几百吉字节。再者，存储器已经很便宜，人们也就更有理由存储更大量的数据。例如，零售连锁店常常存储太字节（terabyte，1太字节是1000GB，或是10^{12}B）信息，记录长时间内每一笔销售业务的来龙去脉（用于规划库存，对此将在1.1.7节中作进一步讨论）。

更进一步，数据库不再集中于存储简单数据项如整数或短字符串，它还可以存储图像、声频信息、视频信息和其他很多种数据，这些数据通常都要占用相当大的空间。例如，1个小时的视频信息就要消耗1GB。数据库存储的卫星图像可涉及拍字节（petabyte，即1000TB，或10^{15}B）数据。

管理如此庞大的数据库需要多种先进技术。譬如，如今中等大小的数据库都存储在磁盘组上，这种磁盘组称做*第二级存储器*（secondary storage device，此名称是与内存相比较而言，内存称为“主”存储器）。人们可能要问，数据库与其他软件究竟有何不同。最主要的不同是数据库系统一般都假定数据量非常大，数据不能存放在主存中，主要是存储在磁盘上。下面介绍的两种技术将允许数据库系统更快地处理大量数据。

三级存储器

磁盘已不能满足现代最大型数据库系统的需要。多种*三级存储器设备*（tertiary storage device）已经出现。每一个三级存储设备可以存储太字节数据，但是它比磁盘需要更长时间存取设备上的数据。典型的磁盘系统需要10~20毫秒存取盘上数据，而三级存储设备将需要数秒钟。三级存储设备存取数据时，要将存储所需数据项的部分输送到一个读取设备上。这种输送动作通过一种自动机器传送工具完成。

例如，CD或DVD都可以作为三级存储设备中的存储介质。一个机械臂安装在轨道上，存取数据时，机械臂移动到某个特定的盘上，将盘举起，携带着数据移到读出器，将该盘装载到读出器上。

并行计算

虽然存储大量数据的能力很重要，但是若不能快速地存取这些大量数据，也就没有什么用
6 处。因此，非常大的数据库也需要增强其处理速度。加速的一个重要手段是通过索引结构，对此将在1.2.2节提及。另外一个可以在给定时间处理更多数据的方法就是利用并行方式。并行可以用各种方式实现。

例如，从一个给定磁盘上读取数据的速度是相当慢的，每秒钟只有几兆字节，如果使用多个磁盘，并且并行地读（即使数据原本在三级存储器上存储，在被DBMS存取之前也要“贮存”到磁盘上），就能加速处理。这些磁盘可以是并行机的一部分，或者是分布式系统的组成成分，在这种系统中有很多机器，每个机器负责一部分数据库，当需要时，信息在这些机器组成的网络中高速传输。

当然，如同存储大量数据的能力，快速移动数据的能力本身并不能保证快速响应查询。仍然需要利用算法将查询分解，以便能允许并行计算机，或分布式计算机网络有效地使用所有这些资源。于是，这给超大数据库的并行和分布式管理的研究与发展留下了一个充满活力的领域。在15.9节中将讨论其某些重要的思想。

1.1.5 客户-服务器和多层体系结构

很多各种现代软件都使用*客户-服务器*(client-server)体系结构。在这种结构中，一个处理器（客户）的请求被送到另一个处理器（服务器）上执行。数据库系统也不例外。将DBMS的

工作分成服务器和一个或多个客户处理器处理的方式已逐渐普及开来。

在最简单的客户/服务器体系结构中，除了与用户交互和发送查询或其他命令到服务器的查询界面，整个DBMS就是一个服务器。例如，通常关系型系统使用SQL语言表示客户到服务器的请求。然后，数据库服务器用表或关系的形式将结果返回给客户。客户与服务器间的联系可以很复杂，特别是当查询结果的数量极其巨大时更是如此。这个问题将在1.1.6节中详细讨论。

如果有很多同时访问的数据库的用户，服务器就将成为瓶颈，因此也有一种倾向是将更多的工作放在客户端。在新近普遍采用的系统中，数据库被用来为Web站点提供动态生成信息（dynamically-generated content）时，这种系统将两层（客户-服务器）结构转为三层（或更多层）结构。这时，DBMS继续作为服务器，其客户端则成为一个典型的应用服务器（application server），管理对数据库的连接、事务、权限以及其他方面的工作。应用服务器本身又有客户端，如Web服务器，支持终端用户或其他应用。 7

1.1.6 多媒体数据

数据库系统中的另一个重要趋势是多媒体数据的加入。“多媒体”的意思是指表示某种信号的信息。多媒体数据形式通常包括视频、音频、雷达信号、卫星图像以及各种编码的文档或图片。这些形式的共同点是它们都大于早期的数据形式——如整数，定长字符串等——并且，它们的大小也存在很大的差异。

多媒体数据的存储迫使DBMS向多个方向扩展。例如，多媒体数据的操作不同于传统数据形式。人们虽然可以在银行数据库中通过银行账户余额与实数0.0比较，查询所有余额为负值的账号，但是不能查询图像数据库，寻找与某个特定形象“看起来相似”的图片。

为了允许用户创建和使用复杂的数据操作，如图像处理，DBMS必须提供用户引入他们自己选择的功能的能力。这类扩展常常用到面向对象技术，在关系系统中，它们被命名为“对象关系”。面向对象数据库程序设计本书还会多次讨论，包括第4章和第9章。

多媒体对象的大小也迫使DBMS修改存储管理器，以便能处理吉字节或更多字节的对象或元组。在如此巨大的数据元素所带来的许多问题中，查询结果的传送是其中之一。通常关系数据库的查询结果是元组集，这些元组被数据库服务器作为一个整体传送到客户端。

可是，假如查询的结果是吉字节长的视频剪辑，服务器就不能将其作为一个整体传送给客户。理由之一是，这样做需要很长时间，从而使服务器不能为其他请求服务。另一个理由是，客户可能只是想要电影剪辑的一小部分，但是如果未看剪辑的起始部分又无法确切地指明需要的部分。第三个理由是，即使客户想要电影的整个剪辑，为了能在屏幕上放映，那也需要在一个小时内按某个固定的速度传送（1小时是放映1GB压缩视频所需的时间）。因此，支持多媒体数据的DBMS存储系统，必须以交互方式提交查询结果，也就是根据用户请求或按照一个固定速度传送一小块查询结果到客户端。

1.1.7 信息集成

当信息已成为人们工作和娱乐的基础时，人们发现原来已存在的信息资源有了新的使用方式。比如，一家公司想要为其产品提供联机目录，以方便用户利用WWW浏览产品并联机定购。 8
通常，一家大公司有很多分部，每个分部可能已经独立地建立了自己的产品数据库。这些分部可能使用的是不同的DBMS，使用不同的数据结构描述信息，甚至可能用同一术语表示不同的事物，或者用不同术语表示相同的事物。

例1.2 想像一家有多个部门的磁盘制作公司。其中一个部门的产品目录中，转速以转数每秒为单位，而另一个部门是以转数每分钟为单位，还可能有另外一个部门根本就忽略了转速。

一个生产软盘的部门可能将软盘称为“盘”，而生产硬盘的部门也可能将硬盘称为“盘”。磁道数可能在一个部门是指“磁道”的数目，而在另一个部门是指“圆柱面”的数目，等等。 □

中央控制方式不是解决问题的万能方法。各个部门可能在认识到部门间信息集成成为问题之前，已经投资了大量资金开发自己的数据库。一个部门也可能是最近刚并入的一个独立公司。为了这些或其他原因，难于取代这些所谓的*遗留数据库*（legacy database）。因此，公司必须在这些遗留数据库的顶层构建某种结构，为顾客提供一个公司产品的统一视图。

通常解决此问题的方法是创建*数据仓库*（data warehouse），通过合适的转换技术，将来自多个遗留数据库的信息复制到一个中央数据库。随着遗留数据库的改变，数据仓库被更新，但是数据仓库的更新并不一定要同步进行。解决更新问题的一个普遍模式是每天晚上，当遗留数据库不太忙时重新构造数据仓库。

经过上述处理，遗留数据库能够继续为公司服务。新的功能，诸如通过WWW提供一个联机目录服务，是在数据仓库上完成的。另外，数据仓库也用于计划和分析。例如，公司分析员可以在数据仓库上通过查询销售趋势，更好地计划库存和生产。同样，也由于数据仓库的建立，*数据挖掘*（data mining）可以在其数据中寻找有价值的和特殊的模式，并通过利用以这种方式发现的模式，找到提高销量的办法。有关这些问题和其他信息集成主题将在第20章中讨论。

1.2 数据库管理系统概述

在图1-1中给出了一个完整的DBMS结构，其中单线框表示系统成分，双线框表示内存中
9 的数据结构，实线表示控制和数据流，虚线只表示数据流。由于图很复杂，在此分几个步骤来考虑细节。首先，在顶部有两个不同的命令源将命令发给DBMS：

1. 通常的用户和应用程序，发出查询数据或修改数据命令。

2. *数据库管理员*（database administrator），是对数据库结构或*模式*（schema）负责的一个人或一批人。

1.2.1 数据定义语言命令

第二种命令的处理比较简单，图1-1的右上方显示命令行踪的开始。例如，大学注册数据库的管理员或DBA，可以决定该数据库中应该有一个表或关系，其中的列有学生，学生选修的课程，以及学生取得的课程成绩。DBA也可以决定，有效成绩只能是A、B、C、D和F。这些结构和约束信息都是数据库的一部分，在图1-1中显示为由DBA输入。由于这些命令能深深地影响数据库，所以DBA具有特定的权限去执行模式修改命令。模式修改DDL命令（“DDL”代表“数据定义语言”）由DDL处理器分析，并且传送给执行引擎，执行引擎然后通过索引/文件/记录管理器去修改*元数据*（metadata），也就是数据库的模式信息。

1.2.2 查询处理概述

与DBMS交互最主要的工作是沿着图1-1左边的路径。用户或应用程序启动某个不影响数据库模式的操作，但是这些操作可能会影响数据库的内容（如果是修改操作），或者是从数据库中抽取数据（如果是查询操作）。1.1节中已指出，表述这些命令的语言称做数据操作语言（DML），或口语化地称做查询语言。有多种数据操作语言，在例1.1中使用的SQL是如今最广泛使用的语言。DML语言由两个独立的子系统处理，有关这两个系统的叙述如下。

查询处理

查询通过*查询编译器*（query compiler）完成语法分析和优化。编译的结果是*查询计划*（query plan）或是由DBMS执行并获得查询结果的操作序列，它们将被送到*执行引擎*（execution

engine）。执行引擎向资源管理器发出一系列获取小块数据的请求，典型的小块数据是关系的记录或元组。资源管理器知道数据文件（data file，存放关系的文件）、数据文件的格式和记录大小和索引文件（index file），这些对于快速从数据文件中找到相应数据元素是有用的。 10

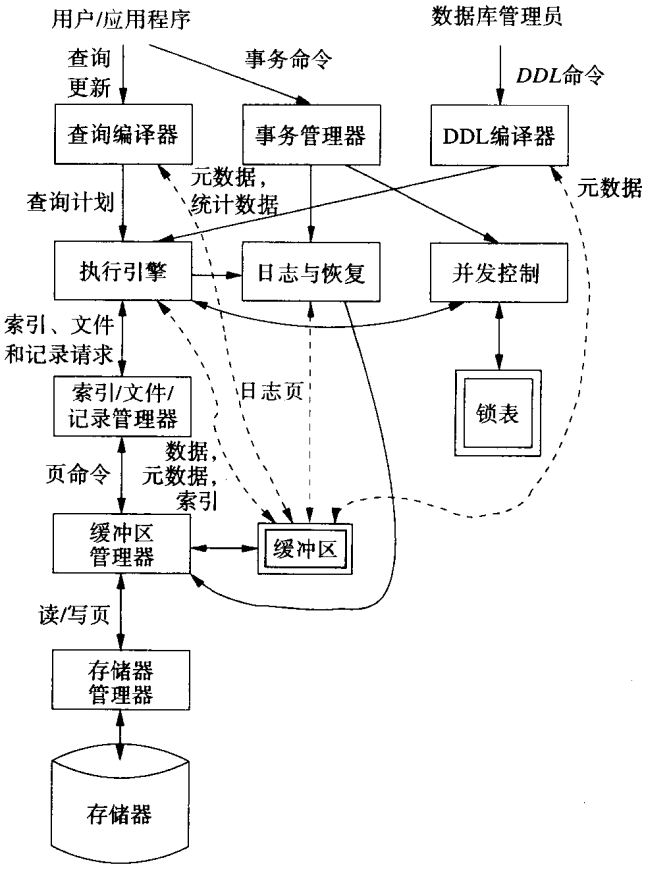


图1-1 数据库管理系统组成 11

数据请求被转换成页请求，页请求又被传送给缓冲区管理器（buffer manager）。1.2.3节将讨论缓冲区管理器，但是简单说来，它的任务是从永久保存数据的二级存储器（通常是磁盘）中获取数据送入主存缓冲区中。一般地，页或“磁盘块”是缓冲区和磁盘间的传送单位。

为了从磁盘中得到数据，缓冲区管理器与存储器管理器通信。存储器管理器可能包含操作系统命令，但是更典型的是DBMS直接向磁盘控制器发命令。

事务处理

查询或其他DML操作被组织成事务（transaction）。事务是必须原子性执行的单位，执行中的事务之间还必须互相隔离。常常一个查询或修改操作本身就是一个事务。另外，事务的执行必须持久（durable），也就是说任何已完成事务的作用必须被保持，即使是事务刚刚完成时系统就出现某种故障也应如此。事务处理器被分成两个主要部分：

1. 并发控制管理器（concurrency-control manager），或调度器（scheduler），保证事务的原子性和独立性；

2. 日志（logging）和恢复管理器（recovery manager），负责事务的持久性。1.2.4节中将进一步讨论这两个部分。

1.2.3 存储器和缓冲区管理器

数据库数据平常存储在二级存储器中。今天的计算机系统中“二级存储器”一般指磁盘。可是，对数据的操作只能在主存中执行。*存储器管理器*（storage manager）的任务就是控制数据在磁盘上的位置存放和在磁盘与主存间的移动。

在一个简单数据库系统中，存储器管理器可以就是操作系统下的文件系统。可是，为了有效性，DBMS常常直接控制磁盘上的存储，至少是在某些环境下如此。*存储器管理器*跟踪磁盘上的文件位置，根据请求从缓冲区管理器中获取含有请求文件的一个或多个磁盘块。回忆一下，*磁盘*通常被分成*磁盘块*（disk blocks），块是含有大量字节的连续存储区域，其大小可以是2^{12}或2^{14}（大约4 000到16 000字节）。

缓冲区管理器负责把可用主存分割成*缓冲区*（buffer），缓冲区是包含若干个页面的区域，
12 其中可以传输磁盘块。于是，所有需要从磁盘中获取信息的DBMS组件，或是直接，或是通过执行引擎的方式，与缓冲区和缓冲区管理器交互。各个组件可能需要的信息种类有：

1. *数据*：数据库本身的内容。
2. *元数据*：描述数据库结构的数据模式和数据库上的语义限制。
3. *统计数据*：由DBMS收集和存储的关于数据特征的数据。例如，数据库大小，数据库中的值，数据库中的各种关系和其他成分。
4. *索引*：支持对数据库中数据有效存取的数据结构。

有关缓冲区管理器，以及它的任务与角色的更完全讨论，将在15.7节中给出。

1.2.4 事务处理

通常将一个或一组数据库操作组成一个*事务*。事务的执行满足原子性，并且与其他事务的执行互相隔离。另外，DBMS还要保证事务的持久性：已完成事务的工作永不丢失。*事务管理器*（transaction manager）接收来自应用的*事务命令*（transaction command），这些命令告诉事务管理器事务何时开始，何时结束，以及应用期望的信息（例如，某些应用可能不需要原子性）。事务处理器执行如下一些任务：

1. *记日志*：为了保证持久性，数据库的每一个变化都记录在单独的磁盘上。*日志管理器*（log manager）遵循一种设计原则，无论何时系统出现故障或“崩溃”，恢复管理器都能够通过检查日志中的修改记录，把数据库恢复到某个一致状态。日志管理器先把日志写入缓冲区，然后与缓冲区管理器协商以确保缓冲区在合适的时间被写入磁盘（磁盘中的数据可以在系统崩溃后幸存下来）。

2. *并发控制*（concurrency control）：事务必须表现为以孤立的方式执行。但是在大多数系统中，很多事务都是同时在执行。因此，调度器（并发控制管理器）必须保证，多个事务的单个动作是按某个顺序在执行，由此获得的效果应该与系统一次只执行一个事务一样。典型的调
13 度器是通过对数据库的某些片段加锁（lock）的方式工作。锁将防止两个以有害方式交互的事务对同一数据片段的存取。如图1-1所示，锁通常保存在主存的*锁表*（lock table）中，调度器通过阻止执行引擎存取加锁数据，来影响查询和其他数据库操作。

3. *消除死锁*（deadlock resolution）：当事务通过调度器授权锁竞争资源时，它们可能会陷入一种状态，由于每个事务需要的资源都被另一个事务占有，没有一个事务能够继续执行。此时，事务管理器的任务是进行干御，并删除（“回滚”或“终止”）一个或多个事务，以便其他事务可以继续执行。

事务的ACID性质

正确执行的事务通常被说成符合“ACID测试”，这里：

- “A”表示“原子性”；事务的操作要么全部被执行，要么全部不被执行。
- “I”表示“独立性”，每个事务的执行必须显现为如同没有其他事务在同时执行。
- “D”表示“持久性”，一旦事务已经完成，则该事务对数据库的影响就永远不会消失。

剩下的字母“C”表示“一致性”。也就是说，所有数据库中数据元组之间的联系具有一致性限制，或说满足一致性期望（例如，账户余额不能是负数）。要求事务保持数据库的一致性。在第7章中将讨论数据库模式中的一致性限制表述。在18.1节开始讨论DBMS如何维护一致性。

1.2.5 查询处理器

用户可以见到的最影响系统性能的DBMS部分是查询处理器（query processor）。图1-1中查询处理器用两个组件表示：

1. 查询编译器 它把查询转换成称做查询计划的内部形式。查询计划是在数据上的操作序列。通常，查询计划中的操作用“关系代数”运算实现。5.2节将讨论关系代数。查询编译器 14
主要由三个模块组成：

（a）查询分析器（query parser） 查询分析器是从查询的文本结构中构造一个查询树结构。

（b）查询预处理器（query preprocessor） 查询预处理器对查询作语义检查（例如，保证查询中提到的关系已确实存在），并且将查询语法树转换成表示初始查询计划的代数操作符树。

（c）查询优化器（query optimizer） 查询优化器将查询初始计划转换成在实际数据上执行最有效的操作序列。

查询编译器使用关于数据的元数据和统计数据以确定哪种操作序列最快。例如，索引是一种特殊的便于数据存取的数据结构，如果索引存在，并且给定索引数据项值，则利用索引的查询计划将比其他计划更快。

2. 执行引擎（execution engine） 执行引擎负责执行选定查询计划的每一步。执行引擎与DBMS中的其他大多数组件直接地或通过缓冲区交互。为了操作数据，它必须从数据库中将数据取到缓冲区，必须与调度器交互以避免存取已加锁的数据，它还要与日志管理器交互以确保所有数据库的变化都被日志正确地记录下来。

1.3 数据库系统研究概述

有关数据库系统的研究被分成三大类：

1. 数据库设计 研究如何开发一个有用的数据库？将什么样的信息送入数据库？信息组织是怎样的？对数据项的类型或值有何假定？数据项之间如何连接？

2. 数据库程序设计 研究如何在数据库上表述查询和其他操作？如何在应用中使用事务或约束等DBMS的其他功能？数据库程序设计如何与普通程序设计相结合？

3. 数据库系统实现 研究如何建立一个DBMS，包括查询处理，事务处理和高效访问的存储组织？ 15

如何实现索引

读者可能在数据结构课程中已知，散列表是建立索引的非常有效的方法。早期DBMS确实广泛使用散列表。今天，最通用的数据结构称做B-树，“B”的意思是“平衡”。B-树是平衡二叉查询树的推广，一棵二叉树的每个节点的子节点可以多达两个，而B-树的节点数可以包含大量的子节点。已知的B-树通常驻留在磁盘上而不是主存中，B-树的一个节点占据整个一个磁盘块。由于一般系统使用的磁盘块是2^{12}B（4096B）数量级，B-树的一块可以容纳几百个指针，因此，B-树查询涉及的树层次很少。

一般磁盘操作的开销与存取磁盘的块数成正比。由于B-树典型查询只涉及几个磁盘块，二叉树的节点驻留在很多不同磁盘块中，所以B-树查询比二叉树查询要快得多。那些最适于磁盘存储的数据结构和适于主存中运行的算法的数据结构有多处不同，B-树与二叉树查询之间的差别是其中之一。

1.3.1 数据库设计

第2章首先给出了表述数据库设计的高层表示法，称为实体联系模型（entity-relationship model）。第3章介绍关系模型，这是目前流行DBMS采用的模型，1.1.2节中对此已有非常简单的介绍。另外将介绍如何把实体-联系设计转换为关系设计，或“关系数据库模式”。在6.6节，将展现如何把关系数据库模式用SQL语言的数据定义语句写出。

第3章还为读者介绍“函数依赖”概念，这些依赖在形式上被表述为关系中对元组之间的联系的假设。“函数依赖”使我们通过关系“规范化”处理，可以改进关系数据库设计。

第4章介绍面向对象数据库设计方法。将介绍ODL语言，该语言允许用户以高层次的面向对象方式描述数据库。我们也要考察，面向对象设计已经与关系模型相结合，产生了所谓“对象-关系”模型。最后，第4章还介绍特别灵活的数据库模型，即“半结构化数据”（semistructured data），同时也可以看到它被具体化为文本语言XML。

16

1.3.2 数据库程序设计

从第5章到第10章都是讨论数据库程序设计。第5章从关系模型查询抽象化处理开始，引入关系上的操作符集，形成了“关系代数”。

第6章到第8章介绍SQL程序设计。正如已经提到的，SQL是当今主要查询语言。第6章介绍SQL中关于查询的基本思想和SQL中数据库模式的表达。第7章介绍SQL中对数据的约束条件和触发器。

第8章介绍SQL程序设计的某些高级内容。首先，虽然SQL程序设计的最简单方式是独立的、通用的查询界面，但实际上，大多数SQL程序设计是嵌入在一个更大程序中，这个程序用传统的程序设计语言编写，诸如C。第8章中学习如何将SQL语句与一个主程序连接，如何将数据库数据传给程序变量以及相反的过程。本章还介绍如何使用SQL的事务说明特性连接客户到服务器，如何授权用户对数据库数据的访问。

第9章转入介绍面向对象数据库程序设计的标准语言。这里考虑两个方面：第一，OQL（对象查询语言），这种语言可以看做是使C++，或其他面向对象程序设计语言，与高层的数据库程序设计相容的一种尝试。第二，也是目前在SQL标准中采纳的面向对象特征，可以看做是试图使关系数据库和SQL与面向对象程序设计相匹配。

最后，第10章又回到第5章中开始的抽象查询语言的研究。研究基于逻辑的语言，并且看它如何被用于扩展现代SQL的能力。

1.3.3 数据库系统实现

本书第三部分是关于DBMS实现。数据库系统的实现可以粗略地分成三部分。

1. 存储器管理（storage management）：研究二级存储器的有效存储数据和快速访问技术。
2. 查询处理：研究用高级语言（如SQL）表述的查询如何能有效地执行。
3. 事务管理：研究如何支持事务的ACID性质。

本书中将分别用几个章节的篇幅对上述每一个问题进行讨论。 17

存储器管理概述

第11章中引入了存储的层次结构。但是因为二级存储器，特别是磁盘，是DBMS管理数据的主要方式，所以对磁盘上数据的存储和存取方式给出了特别详细的描述。书中引入了基于磁盘数据的“块模型（block model）”，这个模型几乎对数据库系统中的每个部分都有影响。

第12章是关于数据元素存储对数据块模型的需求分析。这里数据元素指关系、元组、属性值和在其他数据模型中与它们对应的数据项。然后，讨论用于构造索引的重要数据结构，索引是支持数据做有效访问的数据结构。第13章中讨论重要的一维索引结构——顺序索引文件、B-树和散列表。这些都是DBMS中常用的索引，它们支持查找满足某个给定属性值的元组的查询要求。B-树用于存取已按某个属性值排序的关系。第14章讨论多维索引，这类索引支持一些专用数据结构，如地理数据库，其典型的查询是查找某个区域内的信息内容。这些索引结构也支持具有两个或多个属性值限制的复杂SQL查询，目前商用DBMS中已经开始出现某些这样的结构。

查询处理概述

第15章讨论查询执行的基本内容。将研究多个有效实现关系代数操作的算法。这些算法对磁盘数据很有效，在某些情况下，它们与主存数据的同类算法很不同。

第16章考虑查询编译和优化的体系结构。先从查询的语法分析和语义检查开始，然后，考虑将查询从SQL转换到关系代数，以及逻辑查询计划（logical query plan）选择。逻辑查询计划表示在数据上执行的操作的代数表达式，以及必要的操作顺序约束。最后，实施物理查询计划（physical query plan）选择。物理查询计划给出了特定的操作序列，以及用于实现每个操作的算法。

事务处理概述

第17章中讨论DBMS如何支持事务的持久性。其中心思想是把数据库的所有变化都记录在日志上。当系统崩溃时（比如当系统断电时），任何在主存但不再磁盘上的信息都被丢失。因此，必须仔细地按某种次序将记录了数据库改变的日志从缓冲区移到磁盘。有多个可用的日志策略，但是每一种策略都不同程度地对操作带来一定的约束。 18

然后，在第18章中讨论并发控制——保证事务的原子性和独立性。事务是读或写数据库元素的操作序列。该章的主要内容是讨论如何管理数据库元素上的锁：可以使用不同类型的锁，事务以何种方式获取与释放数据元素上的锁。另外，还要研究几种不用锁也能保证事务原子性和独立性的方法。

第19章是关于事务处理研究的总结。讨论第17章中给出的日志技术需求与第18章中给出的并发需求之间的相互作用。并发控制管理的另一个重要课题——死锁处理也在本章中讨论。第19章还要讨论并发控制在分布式环境下的扩展。最后，讨论引入“长事务”的可能性。所谓“长”事务，是说事务的执行时间不是几毫秒，而是要几小时或几天。长事务对数据加锁，将在使用该数据的其他用户之间引起混乱，因此，迫使研究者对涉及长事务应用的并发控制有新的思考。

1.3.4 信息集成概述

数据库系统的很多最新发展趋向是允许不同数据源（data source），可以是数据库和/或不由DBMS管理的信息源，一起作为一个整体工作。对此在1.1.7节中已有简单的介绍。本书最后一章，第20章，将研究信息集成的某些重要内容。讨论集成的基本方法，包括称做“数据仓库”的经转换的和集成的数据源副本，以及通过“协调器”收集的数据源虚拟“视图”集合。

1.4 小结

- *数据库管理系统*：DBMS的特征，是支持可以长期保存的大量数据有效存取的能力，支持强有力的查询语言和按原子性和与其他事务独立的方式并发执行的持久事务能力。
- *与文件系统比较*：因为文件系统不能支持有效查找，不支持对小片数据的有效修改，不支持复杂查询，也不支持主存中有用信息的缓冲控制以及事务的原子性和独立执行，所以，通常的文件系统不适宜作数据库系统。
- *关系数据库系统*：今天，大多数数据库系统基于关系数据模型，这种模型将数据组织成
19 表，SQL是这些系统中最常使用的语言。
- *二级和三级存储器*：大的数据库是存储在二级存储器设备上，通常是在磁盘上。最大的数据库需要三级存储器设备，它的容量比磁盘要高几个数量级，但是它的速度也要比磁盘慢几个数量级。
- *客户-服务器系统*：数据库管理系统一般支持客户-服务器体系结构，数据库的主要部分是在服务器上，客户机用于对用户提供界面。
- *未来系统*：数据库系统的主要趋势，一是支持非常大的“多媒体”对象，如视频或图像。另外是将来自多个独立的信息源信息集成为单个数据库。
- *数据库语言*：介绍了定义数据结构的语言或语言成分（数据定义语言），以及数据查询和更新语言（数据操作语言）。
- DBMS组成：数据库管理系统的主要组成是存储器管理器、查询处理器和事务管理器。
- *存储器管理器*：该部件的责任是存储数据、元数据（关于数据模式或结构的信息）、索引（加速数据存取的数据结构）和日志（记录数据库变化）。这些数据结构被保存在磁盘上。存储器管理器的重要部分是缓冲区管理器，该管理器在主存中保存部分磁盘内容。
- *查询处理器*：该部件分析查询，通过选择查询执行计划优化查询，然后在存储数据上执行选择的查询计划。
- *事务处理器*：该部件负责记录数据库的变化的日志，以支持系统故障后的恢复。它也能保证并发事务执行的原子性（事务或是完全执行，或是完全不执行）和孤立性（事务执行时如同不存在其他并发执行的事物）。

1.5 参考文献

当前，联机可查询的目录基本覆盖了所有关于数据库系统的最新文章。因此，本书并不试
20 图给出完全的引述，但是将给出历史上重要的文章和主要的辅助文献或有用的综述。Michael Ley[5]已给出了可查找的数据库研究论文的索引。AlfChristian Achilles维护了一个与数据库领域有关的可查找的索引目录[1]。

有很多对数据库领域技术有贡献的原型实现，其中最有名的两个是IBM Almaden研究中心的System R项目[3]和伯克利大学的INGRES项目[7]。它们都是早期的关系系统，并且使此类系

统成为主流数据库技术。许多使数据库领域定形的研究文章可在文献[6]中找到。

1998年的“Asilomar 报告”[4]是关于关系数据库系统一系列研究和指导报告中最新的一份。它也引用了早期的这类报告。

从文献[2]、[8]和[9]中可以发现比这里提到的更多关于数据库的理论。

1. `http://liinwww.ira.uka.de/bibliography/Database` .

2. Abiteboul, S., R. Hull, and V. Vianu, *Foundations of Databases*, Addison-Wesley, Reading, MA, 1995.

3. M. M. Astrahan et al., “System R: a relational approach to database management,” *ACM Trans. on Database Systems* **1**:2 (1976), pp. 97–137.

4. P. A. Bernstein et al., “The Asilomar report on database research,” `http://s2k-ftp.cs.berkeley.edu:8000/postgres/papers/Asilomar_Final.htm` .

5. `http://www.informatik.uni-trier.de/~ley/db/index.html` . A mirror site is found at `http://www.acm.org/sigmod/dblp/db/index.html` .

6. Stonebraker, M. and J. M. Hellerstein (eds.), *Readings in Database Systems*, Morgan-Kaufmann, San Francisco, 1998.

7. M. Stonebraker, E. Wong, P. Kreps, and G. Held, “The design and implementation of INGRES,” *ACM Trans. on Database Systems* **1**:3 (1976), pp. 189–222.

8. Ullman, J. D., *Principles of Database and Knowledge-Base Systems, Volume I*, Computer Science Press, New York, 1988.

9. Ullman, J. D., *Principles of Database and Knowledge-Base Systems, Volume II*, Computer Science Press, New York, 1989.

第2章　实体-联系数据模型

设计数据库时，要分析这个数据库必须存储的信息及这些信息组成部分之间的关系。通常数据库的结构为*数据库模式*（database schema），它是用一种语言或一些设计符号来描述的。设计人员经过适当的考虑，设计出一份可以被输入到DBMS的格式，数据库也以物理形式建立起来。

本书会用到几种设计符号。在这一章，先开始介绍一种传统且流行的方法，叫做“实体-联系”（E/R）模型。这种模型实际上是用矩形和箭头表示基本数据元素及其联系的图形。

第3章将集中讨论关系模型。在关系模型中，现实领域被表示为一个表集合。关系模型虽然只能表达有限的结构，然而简单、实用，当今主要的商业DBMS都是建立在这种模型上。数据库设计者通常先用E/R模型或是面向对象的模型设计出模式，再把模式转换成关系模型。

其他模型会在第4章涉及到。4.2节会介绍面向对象数据库的标准——ODL（对象定义语言）。然后，再来看面向对象的思想对关系DBMS的影响，由此产生常说的“对象-关系”模型。

4.6节介绍另一种建模方法，叫做“半结构化数据”。这种模型在组织数据的结构上有很大的灵活性。在4.7节，还要讨论XML标准，它把数据转化成具有层次式结构的文档模式，用“标签”（像HTML的标签）来标识文本元素的作用。XML是半结构化数据模型的一个重要体现。

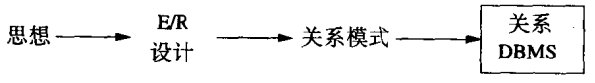


图2-1　数据库建模与实现过程

图2-1显示了如何用E/R模型进行数据库设计。先是提出对信息建模的思想，并用E/R模型描述它们。然后，将抽象的E/R设计转化成用某个DBMS的数据说明语言描述的模式。这种DBMS最常使用的是关系模型。通过这样一种相当机械的处理过程，把抽象的设计转化为具体的关系设计，设计的结果叫做“关系数据库模式”。这些内容将在3.2节中讨论。

应当注意到，有一些DBMS使用非关系或非对象-关系的模型，而且没有一个DBMS是直接用E/R模型的。原因是E/R模型并不是足够有效的数据结构，不能作为数据库的基础。

2.1　E/R模型的要素

数据库结构抽象表示的最常用模型是*实体-联系模型*（entity-relationship model），或E/R模型。在E/R 模型中，数据的结构被表示为“实体-联系”图，图中有三个主要的元素类型：

1. 实体集；
2. 属性；
3. 联系。

下面将依次介绍这三个元素。

2.1.1　实体集

实体（entity）是某个抽象事物，相似实体的集合形成实体集（entity set）。从面向对象程序设计的意义上讲，实体和“对象”有某种相似性。同样，实体集和对象类也有相似性。但是，E/R模型是个静态的概念，它只包括数据的结构而不包括对数据的操作。所以，实体集中不会

像类那样有方法（method）出现。

例2.1 用一个关于电影的数据库作为持续使用的例子。数据库中包括有电影、影星、电影制作公司和电影其他方面的内容。每部电影是个实体，所有电影构成一个实体集。同样，影
24 星也是实体，影星的集合也是一个实体集。电影公司是另一类实体，它的集合是出现在例子中的第三个实体集。 □

E/R模型的变化

在E/R模型的某些版本中，属性的类型可以是：

1. 原子的，如本书E/R模型中的属性。
2. 如C语言中的“结构”或具有固定数目的原子性成分的元组。
3. 某种类型的一组值：或为原子类型的，或为“结构”类型的。

2.1.2 属性

实体集有相关的*属性*（attribute），属性是这个实体集中的实体所具有的性质。比如，实体集Movies的属性可能有title（电影名）或length（片长）即电影放映多少分钟。在本书的E/R模型中，假定属性都是原子值，比如字符串、整数或实数。但是这个模型可以有一些其他变化，其中属性可以是限定的结构，请见上面方框中的“E/R模型的变化”。

2.1.3 联系

联系（relationship）是两个或多个实体集间的连接。如，Movies和Stars是两个实体集，Stars-in就是连接Movies和Stars的联系。其目的是如果影星实体s出现在电影实体m中，m和s就被*Stars-in*联系在一起。二元联系是目前为止最一般的联系类型，它联系两个实体集，E/R模型允许联系连接任意数目的实体集。2.1.7节将讨论这种多路联系。

2.1.4 实体-联系图

E/R图（E/R diagram）是用来描述实体集、属性和联系的图形。图中每种元素都用结点表
25 示。我们用特殊形状的结点来表示特定的元素类别：

- 矩形表示实体集
- 椭圆表示属性
- 菱形表示联系

用实线来连接实体集与它的属性以及联系与它的实体集。

例2.2 图2-2是一个E/R图，表示一个简单的电影数据库。实体集是Movies、Stars和Studios。

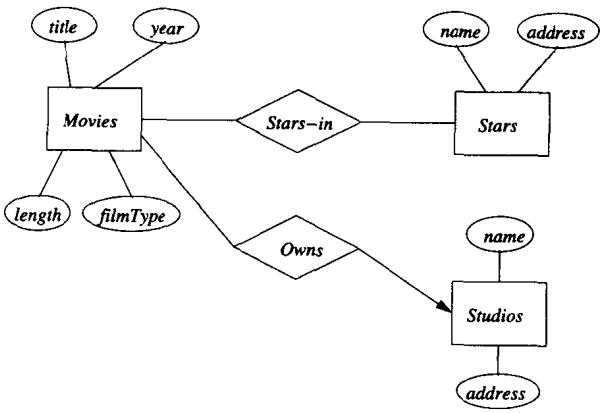


图2-2 电影数据库的实体联系图

Movies实体集有四个属性：title、year（电影制作日期）、length和filmType（“彩色”或“黑白”）。另外两个实体集Stars和Studios正好有两个相同的属性：name和address，都有其显而易见的含义。图中还有两个联系：

1. Stars-in是电影及其影星的联系。因此这也是影星及其参演电影的联系。

2. Owns是电影及其所属电影公司的联系。图2-2中指向实体集Studios的箭头暗示每部电影只属于惟一的电影公司。2.1.6节将讨论像这样的惟一性约束。 □ 26

2.1.5 E/R图实例

E/R图是一种描述数据库模式（即数据库结构）的符号。用E/R图描述的数据库包含特定的数据，称做数据库实例（instance）。特别地，对每个实体集，数据库实例有一个特定的有限实体集合。实体集中的每个实体对每个属性都有特定的值。记住，这种数据是抽象化的；人们并不会直接把E/R数据存入数据库。在把数据转化为关系数据库和物理存在形式之前，想像一下这种数据的存在有助于对设计的思考。

数据库实例也包含联系的具体选择。连接n个实体集E_1、E_2、…、E_n的联系R有一个实例，由列表（e_1、e_2、…、e_n）的有限集构成，其中e_i是从实体集E_i的当前实例中选出的。这样的列表由联系R“连接”起来。

这个列表集叫做R当前实例的联系集（relationship set）。把联系集直观地表示为一张表很有帮助。表的列标题是包含在联系集中的实体集名，表的行是被联系连接起来的一串实体集。

例2.3 下表表示联系Stars-in的一个实例：

Movies	*Stars*
Basic Instinct	Sharon Stone
Total Recall	Arnold Schwarzenegger
Total Recall	Sharon Stone

联系集的成员是表的行。例如：

```
(Basic Instinct, Sharon Stone)
```

是联系Stars-in的当前实例的联系集中的一个元组。 □

2.1.6 二元E/R联系的多样性

总体来说，二元联系能将一实体集中任意数目的实体与另一实体集中任意数目的实体连接。可是，对联系的多样性通常会有所约束。假设R是连接实体集E和F的联系，那么：

- 如果E中的任一实体可以通过R与F中的至多一个实体联系，那么说R是从E到F的多对一（many-one）联系。当从E到F是一种多对一的联系时，F中的每一个实体都能与E中的若干个实体联系。类似地，如果F中任一实体可通过R与E中至多一个实体联系，则说R是从F到E的多对一联系。（或者说是从E到F的一对多联系。） 27
- 如果R既是从E到F的多对一联系，又是从F到E的多对一联系，那么R就是一对一（one-one）联系。在一对一联系中，实体集中的一个实体最多可以和另一实体集中的一个实体联系。
- 如果R既不是的从E到F的多对一联系，也不是从F到E的多对一联系，则说R是多对多（many-many）联系。

正如在例2.2中提到的，箭头可用来表示E/R图中联系的多样性。如果从实体集E到F是多对一联系，就把箭头指向F。箭头表明实体集E中每个实体与实体集F中的最多一个实体联系。除非还有一个箭头指向E，F中的每个实体可以与E中的若干个实体联系。

例2.4 根据这个原则，如果实体集E和F是一对一联系，就把箭头同时指向E和F。例如，图2-3中有两个实体集Studios和Presidents以及它们之间的联系Runs（属性省略）。假设一位经理只管理一家电影公司，一家电影公司只有一位经理，那么这种联系就是一对一的，可以用两个箭头分别指向两个实体。



图2-3 一个一对一联系

记住一个箭头是表示“最多一个”，但它并不保证箭头指向的实体集中的实体存在。所以，在图2-3中，你可以认为一位经理一定会和某个电影公司有联系；否则他怎么会是经理？但是，电影公司可能会在某一特定时期没有经理，所以从Runs指向Presidents的箭头含义是“最多一个”，而不是“有且只有一个”。2.3.6节中将对此作进一步讨论。 □

2.1.7 多路联系

E/R模型使人们能够更方便地描述两个以上实体集之间的联系。实际上，三重（三路）或更多路的联系很少，但是有时这些联系对于反映事物的真实情况很有必要。E/R图中的多路联系由从菱形到联系涉及的每个实体集的连线表示。

不同联系类型的含义

应当意识到多对一联系是多对多联系的一种特殊情况，而一对一联系是多对一联系的一种特例。也就是说，多对多联系的任何有用的特性也同样适用于多对一联系，多对一联系的任何有用的特性也同样适用于一对一联系。例如，表示多对一的数据结构也可以用来表示一对一联系，但不适用于多对多联系。

例2.5 图2-4是一个Contracts联系，包括电影公司、影星和电影三个实体集。这种联系表明电影公司和某一影星签约，让他出演一部电影。一般而言，E/R联系的值被当作元组的一个
28 联系集，元组的组成成分是加入到联系中的实体，在2.1.5中已经谈过这点。所以，Contracts联系可以由三元组

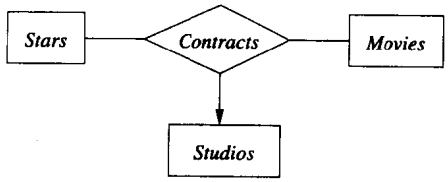


图2-4 一个三路联系

```
(studio, star, movie)
```

来描述。

在多路联系中，指向实体集E的箭头表示：如果从此联系中除E之外的其他每个实体集选择一个实体，它们至多与E中的一个实体有联系（注意，这种规则是对二元联系所用的多对一表示法的推广）。在图2-4中，有一个箭头指向Studios，表明对于某一影星和电影来说，只有一家电影公司与这位影星签订了出演此电影的合同。 □

2.1.8 联系中的角色

在一个联系中一个实体集可能出现两次或多次。如果是这样，根据实体集在联系中出现的次数，把联系与实体集用同样多的连线连起来。每一条连向实体集的连线代表实体集在联系中
29 扮演的不同角色（role）。因而人们给实体集和联系之间的线命名，称之为“角色”。

多路联系中箭头符号的限制

当一个联系连接三个或更多的实体集时，仅选择有无箭头的连线是无法表示这种情况的。箭头无法描述每种可能出现的情况。如图2-4所示，电影制作仅仅是电影公司才具有的功能，而不是影星和电影二者的结合。已有的符号之所以不能把这种情况与三路联系区分开来，是因为在三路联系情况下，被箭头指向的实体集是其他两个实体集的函数。在3.4节中采用了一种正式符号——函数依赖，它可以描述出一个实体集如何被其他实体集惟一决定的所有可能。

例2.6 图2-5给出了一个实体集Movies和一个由它本身组成的联系Sequel-of。每个联系连接两部电影，其中一部是另一部的续集。为了在一种联系中区别两部电影，一条线标以Original，另一条标以Sequel，分别代表最初的电影及续集。假设一部电影有许多部续集，但对于每部续集来说只存在一部初集。所以Sequel电影与Original电影之间是多对一联系，如同已在图2-5 的E/R图中用箭头表明的那样。

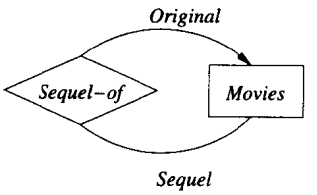


图2-5 带有角色的联系

□

例2.7 这是最后一个包含多路联系和具有多重角色的实体集的例子。图2-6所示联系比例2.5中介绍的Contracts联系更为复杂。这里，Contracts联系包括两家电影公司、一位影星和一部电影。其含意是，一家跟某个影星签约（通常并非仅为某部影片）的电影公司，可能与另一家电影公司签约使该影星能出演某部电影。因而，此联系被描述为四元组

```
(studio1, studio2, star, movie)
```

的形式，表示studio2与studio1签约借用studio1的影星出演电影。

30

在图2-6中可以看到，箭头指向Studios的两种角色——影星的“拥有者”和电影的制作公司。但是，没有箭头指向Stars或Movies。其理由如下：给定一位影星、一部电影和制作这部电影的电影公司，就可确定“拥有”此影星的惟一一家电影公司。（假定一位影星只与一家电影公司签约。）类似地，一部电影只由一家电影公司制作，因此给定一位影星、一部电影和此影星的签约电影公司，就可以惟一决定出此电影的制作公司。注意，在这两种情况下，只需要知道其他实体集中的任一个实体就可以惟一决定一个实体——例如，只需要知道电影就可以惟一确定制作电影公司——但这并不改变多路联系的多样性。

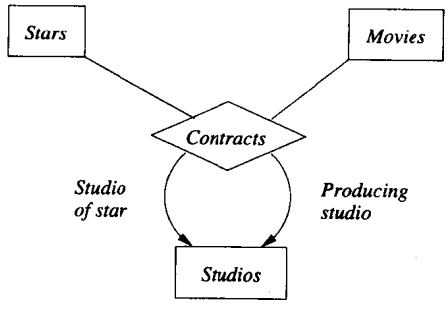


图2-6 一个四路联系

没有箭头指向Stars或Movies。给定一位影星、影星的签约电影公司和电影制作公司，可能会有几个不同的合同使该影星能出演几部电影。因此，一个联系四元组的另外三个组元并不足以惟一决定一部电影。类似地，电影制作公司可能与另外一些电影公司签约以利用它们的多位影星出演同一部影片。因此，影星并不能由其他三个组元惟一决定。

□

2.1.9 联系的属性

有时把属性与联系相连，较之与联系连接的任何一个实体集相连更加方便，甚至至关重要。

例如，图2-4中的联系代表影星和电影公司就一部电影签订的合同[⊖]。人们可能会希望从合同中记录下片酬。但是不能把它与影星联系起来，影星可能在不同的电影中片酬不同。类似地，把
31 片酬与电影公司（它们会付不同的片酬给不同的影星）或电影（不同的影星在同一部电影中的片酬不同）联系起来也是毫无意义的。

但是，把片酬与三元组

```
(star, movie, studio)
```

联系起来是合适的。图2-7是在图2-4的基础上增加了一些属性。此处联系有了属性salary，而图2-2中的实体集也有这样的属性。

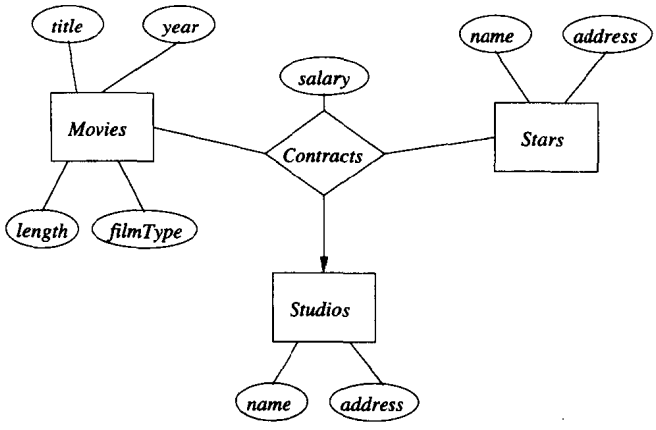


图2-7 一个有属性的联系

其实并没有必要为联系添加属性。人们可以创建一个新的实体集，将原属于联系的属性连到这个实体集上，把这个实体集包含在联系中，省去联系本身的属性。但是联系上带属性是个有用的惯例，在适当的场合会继续使用它。

例2.8 修改图2-7中的E/R图。原图中的联系Contracts有属性salary。创建一个有属性salary的实体集Salaries。Salaries变成了联系Contracts的第四个实体集。全图显示在图2-8中。 □

2.1.10 多路联系到二元联系的转换

有一些数据模型限制联系必须是二元的，如ODL（对象定义语言），这将在4.2节介绍。而E/R模型不要求联系必须是二元的，所以有必要看一下连接多个实体集的联系怎样转化为一组
32 二元的多对一联系。这里介绍一种新的实体集，它的实体被看做是多路联系的联系集的元组。这个实体集叫做连接实体集。接下来介绍这种多对一联系，它把连接实体集与组成原来多路联系元组的每个实体集相连。如果一个实体集扮演多个角色，那么每个角色的一个联系就指向它。

例2.9 图2-6中的四路联系Contracts可以用一个也称为 Contracts的实体集代替。如图2-9，合同涉及了四种联系。如果联系Contracts的联系集有一个四元组

```
(studio1, studio2, star, movie)
```

那么实体集Contracts就有一个实体e。这个实体由联系Star-of连向实体集Stars中的实体star；由联系Movie-of连向Movies中的movie实体；再分别由联系Studio-of-star和Producing-studio连向Studios中的实体studio1和studio2。 □

⊖ 这里，已将其恢复成例2.5中的三路合同的符号，而不是例2.7中的四路联系。

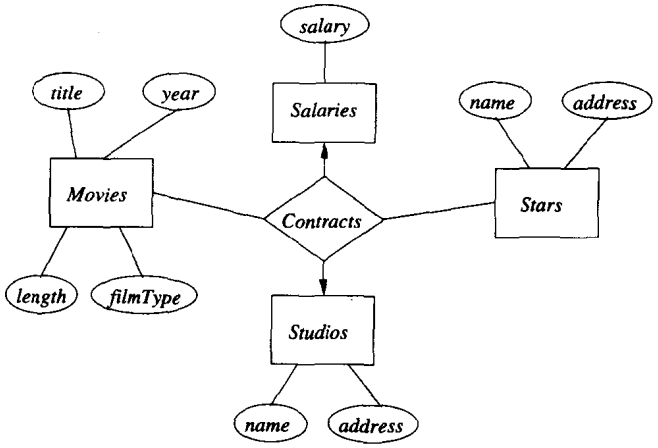


图2-8 将属性移至实体集

注意，虽然图2-9中的其他实体集都有隐含属性，但是假定为实体集的Contracts没有属性。然而，也可能对实体集Contracts加上属性，比如签约的日期。 □

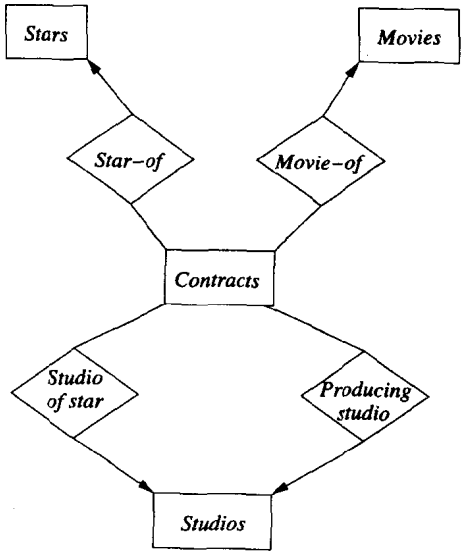


图2-9 用实体集和二元联系代替多路联系

2.1.11 E/R模型中的子类

经常地，一个实体集中含有一些实体，这些实体拥有集合中其他实体成员没有的特殊性质。如果这种情况存在，那么，定义一些特例实体集或子类（subclass）就很有用。这里每个子类有它自己的特殊的属性和/或联系。我们用一个被称做isa的联系连接实体集和它的子类（也就是，“*A*是*B*”表达了从实体集*A*到实体集*B*的“isa”联系）。

isa联系是一种特殊的联系，为了强调它与其他联系不同，用一种特殊符号表示。每个isa联系用一个三角形表示。三角形的一边与子类相连，与此边相对的一角与父类相连。尽管没有标出两个箭头，但每个isa联系都是一对一联系。

例2.10 在电影实例数据库中可以存储的电影种类有卡通片、凶杀片、惊险片、喜剧片以及其他种类的电影。对每一种电影，都可以定义Movies实体集的子类。例如，假定有两个子类：Cartoons和Murder-Mysteries。卡通片除了拥有所有Movies共同的属性和联系外，还有一个额外的联系叫Voices，它给出了并不出演电影的配音演员的集合。除卡通片外的电影没有配音演员。凶杀片的一个附加属性是weapon。实体集Movies、Cartoons和Murder-Mysteries之间的联系如图2-10所示。 □

并列联系可以不同

图2-9说明联系的一个细节。在实体集Contracts和Studios上，有两个不同的联系，Studio-of-Star和Producing-Studio。人们不能因此就认为二者存在相同的联系集。事实上，两种联系中，同一份合同和相同的电影公司相连接是不可能的，如果这样，电影公司就

只能与它本身签约。

更一般地，E/R图中相同实体集存在多种联系是正常的。在数据库中，这些联系实例一般都不同，反映了联系的不同意义。事实上，如果两个联系的联系集是相同的，那么它们就是相同的联系，不应该赋予不同的名字。

34 当然，原则上说，由isa联系连接起来的一组实体集可以是任何一种结构，但是在这里是把isa结构限制为树形结构，即只有一个根（root）的实体集（如图2-10中的Movies），它是最具有普遍性的，其他逐渐具体化的实体集就由树形结构的根向下延伸。

假设有一个由isa联系连接的树形实体集。单个实体由来自于一个或多个实体集的组成部分（component）构成，这些组成部分在包括根的子树中。也就是说，如果一个实体e在实体集E中有组成部分c，E在树中的父实体集是F，那么e也有F中的组成部分d。另外，c和d必须在从E到F的isa联系集中配对出现。e的组成部分有什么属性，实体e就一定有什么属性；它们参与什么联系，e就一定参与什么联系。

例2.11 一般的电影(既不是卡通片也不是凶杀片)仅有在如图2-10的根实体集Movies中的一个组成部分。这些实体只有Movies的四个属性（图2-10中没有显示Movies的两个联系——Stars-in和Owns）。

一部非凶杀类卡通片有两个组成部分，一个在Movies中，一个在Cartoons中。故而它的实体不仅有Movies的四个属性，还有联系Voices。类似地，凶杀片的实体有两个组成部分，一个在Movies中，一个在Murder-Mysteries中，因此有五个属性，包括weapon。

最后，像Roger Rabbit这样的电影既属于卡通片也属于凶杀片，它在三个实体集Movies、Cartoons和Murder-Mysteries中都有组成部分。三个组成部分被isa联系连接为一个实体。这些组成部分给予了Roger Rabbit实体Movies的四个属性和Murder-Mysteries的weapon属性以及
35 Cartoons的Voices联系。 □

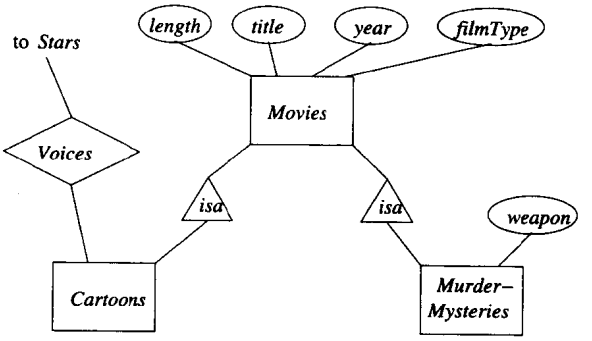


图2-10 E/R图中的isa联系

2.1.12 习题

* **习题2.1.1** 为一家银行设计一个数据库，包括客户以及他们账户的信息。客户信息包括姓名、地址、电话、社会保障号，账户信息包括号码、类型（如存款、支票）和余额。另外还需要记录有账户的客户。为该数据库画一幅E/R图。在适当的地方画上箭头，以表示联系的多样性。

习题2.1.2 按如下要求修改习题2.1.1的解决方案：

a) 修改图使一个账户只有一位客户。

b) 使一位客户只有一个账户。

! c) 修改习题2.1.1中的原图，使客户可以有多个地址（街道、城市、州组成的元组）和多个电话。记住E/R模型不允许属性有非原子类型，如集合。

! d) 进一步修改你的图使客户可以有一组地址，每个地址有一组电话号码。

习题2.1.3 为数据库设计E/R图，该数据库记录球队、队员和球迷的信息，包括：

a) 每个球队的名称、队员、队长（是队员的一员）和队服颜色。

b) 每名队员的名字。

! c) 每位球迷的名字、最喜欢的球队、最喜欢的球员、最喜欢的颜色。 36

有一些颜色不适合球队的属性，你如何避开这些约束呢？

习题2.1.4 假设希望在习题2.1.3的模式上加一个联系Led-by，连接两名队员和一个球队。意思是联系集由三元组

```
(player1, player2, team)
```

构成，表示player2当队长时，player1在此队踢球。

a) 修改E/R图。

b) 把三元联系用一个新的实体集和二元联系替换。

! c) 新的二元联系是否和以前存在的某个联系相同？注意这里假设两名队员是不同的，也就是说，队长不能自己领导自己。

面向对象系统中的子类

E/R模型中的isa和面向对象语言中的子类有惊人的相似之处。从某种意义上说，“isa”是把子类与父类联系起来。但是，传统的E/R观点与面向对象方法仍有根本区别：在实体集树中一个实体由其他实体代表是允许的，但对象只存在于一个类或子类中。

在例2.11中，当讨论如何处理电影Roger Rabbit时，这种区别变得很明显。运用面向对象方法，需要给这部电影第四个实体集：cartoon-murder-mystery，它继承了Movies、Cartoons和Murder-Mysteries的所有属性和联系。然而，在E/R模型中，只要把Roger Rabbit的所有成分放进Cartoons和Murder-Mysteries实体集中，就能获得第四个子类的作用。

习题2.1.5 修改习题2.1.3，为每个队员记录他们踢球的历史，包括为每个球队效力的开始日期和结束日期（如果他们是转会过来的）。

! **习题2.1.6** 假设维护一部家谱需要一个实体集Person。希望记录的成员信息包括他们的姓名（一个属性）以及以下的联系：母亲、父亲和孩子。给出E/R图，包括Person实体集和它涉及的所有的联系，包括与母亲、父亲、孩子的联系。当一个实体集在一个联系中被多次使用时别忘了指出其角色。 37

! **习题2.1.7** 修改习题2.1.6中的数据库设计以包含下列类型的成员：

1. 女性
2. 男性
3. 为人父母者

如果你还想根据其他类型来设计，请用联系连接适当的成员子类。

习题2.1.8 另外一种表示习题2.1.6的信息的方法是用三元联系Family，Family联系集中的三元组

(person, mother, father)

分别是一个人和他的母亲与父亲。当然，这三者都是People实体集中的实体。

* a) 画出E/R图，把箭头标在适当的线上。

b) 用一个实体集和一个二元联系代替三元联系Family，并用箭头显示联系的多重性。

习题2.1.9 为大学注册设计一个适当的数据库。这个数据库应当包含的信息有学生、系别、教授、课程、哪些学生选哪些课、哪些教授教哪些课、学生年级、助教（助教也是学生）、一个系开哪些课，等等，以及任何你认为合适的信息。这个问题比上面的问题提的更自由，你需要考虑联系的多重性，适当的类型，甚至什么样的信息需要表示。

! **习题2.1.10** 非正式地讲，如果反映现实世界情况的两个E/R图的实例，一个能从另一个推出，那么说这两个E/R图包含相同的信息。考虑图2-6的E/R图。利用一部电影必然只由一家电影公司制作这个事实，这种四路联系可以变为一个三路联系和一个二元联系。不利
38 用四路联系，画出与图2-6包含信息相同的E/R图。

2.2 设计原则

虽然还需要了解E/R模型的更多细节。但是在学习是什么组成一个好的设计，什么应当避免这些重要问题之前还有许多需要讨论。在本节，先介绍一些有用的设计原则。

2.2.1 忠实性

首要的也是最重要的，设计应当忠实于应用的具体要求。也就是说，实体集和它们的属性应当反映现实。你不能把属性number-of-cylinders与Stars联系，然而却可以把它作为汽车的属性。根据所了解的那一部分真实世界去建模；无论设计哪一种联系都应当有意义。

例2.12 如果在Stars和Movies之间定义联系Stars-in，它应该是一种多对多联系。原因是根据对现实世界的观察，影星可以在不止一部电影中出现，出演一部电影的影星也不止一位。认为Stars-in是任一方向的多对一或一对一联系都是不正确的。 □

例2.13 另一方面，有时并不清楚现实世界要求在E/R建模中做什么。例如，思考一下实体集Courses和Instructors，它们之间有联系Teaches。从Courses到Instructors，Teaches是一种多对一联系吗？答案在于创建数据库的组织的政策和意图。有没有可能学校有这样一个政策：对每一门课只有一名教师？即使有许多教师组成小组轮流教授一门课程，学校可能要求只将一名教师的名字列入数据库中，作为这门课的负责人。上述任何一种情况，都会使从Courses到Instructors的联系Teaches是多对一的。

相反，学校可能定期聘用一组教师，并且希望数据库允许几名教师教同一门课。或者，Teaches联系的意图可能并不是表示出目前教这门课的教师，而是那些曾经教过或者能够教这门课的人。因此，不能简单地从联系的名称上做出判断。在这两种情况中，把Teaches作为多对多联系更为恰当。 □

2.2.2 避免冗余

应当小心：每件事只说一次。例如，在电影和电影公司之间用了联系Owns。也可以把电影公司的名称studioName选作电影实体集的一个属性。虽然这样做并不违反规定，但却危险，
39 原因有以下几点：

1. 当数据被存储时，同一家电影公司的两次表达，比只表达一次占用了更多的空间。
2. 当电影卖出去后，可能改变了被联系Owns连接的拥有者电影公司，但是忘了改变属性

studioName，或者出现相反的情况。当然有人会说人们不会犯这种不小心的错误，但实际上错误是常有的，把一种东西用两种方式说出来是在自找麻烦。

在3.6节将更正式地讨论这个问题并学习重新设计数据库模式的方法，以去掉冗余及其附带问题。

2.2.3 简单性考虑

除非有绝对需要，不要在你的设计中添加更多成分。

例2.14 假定用“电影所有权”来代替Movies和Studios之间的联系。可再创建一个实体集Holdings，一个一对一联系Represents就在每部电影和它的所有权之间建立起来。从Holdings到Studios之间的多对一联系如图2-11所示。

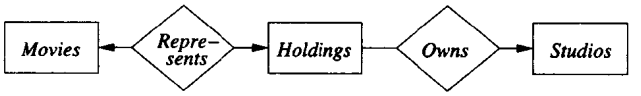


图2-11 含有不必要实体集的拙劣设计

从技术上说，图2-11真实地反映了现实世界，因为一部电影由联系Holdings连向它的惟一的拥有者是可能的。然而，Holdings在这里没有实际用途，没有它可能更好。它使使用“电影-电影公司”联系的程序变得更复杂、浪费空间和更容易犯错。 □

2.2.4 选择正确的联系

实体集可以用多种方式连接起来。但是，把每种可能的联系都加到设计中却不是个好办法。首先，它导致冗余，即一个联系连接起来的实体对或实体集可以从一个或多个其他联系中导出。第二，使得数据库可能需要更多的空间来储存冗余元素，而且修改数据库会更复杂，因为数据的一处变动会引起储存联系的多处变动。这些问题和2.2.2节讨论过的问题本质上一致，尽管导致问题的原因不同。 40

可以用两个例子来解释这个问题，并说明解决办法。在第一个例子中许多联系表示相同的信息。第二个例子中一个联系可以从另外几个中导出。

例2.15 首先再来看一下图2-7，图中电影、影星和电影公司被三路联系Contracts连接起来。这里省略了图2-2中的二元联系Stars-in和Owns。是否需要在Movies和Stars，Movies和Studios之间分别加上这些联系呢？答案是：“不知道。这取决于对问题中这三个联系的假定。”

人们可以从Contracts推出Stars-in。如果只有在合同中包含影星、电影、电影所属的电影公司诸项的条件下，影星才会在这部电影中出现，那么Stars-in的确不需要存在。观察联系Contracts中的star-movie-studio三元组，再把影星、电影两个成分抽出来就可以得到star-movie对了。但是，如果一个影星没有签合同就可以演电影——或者更有可能的情况是在数据库中没有此项合同——那么Stars-in中就可能存在这样的star-movie对，它们不是Contracts中的star-movie-studio三元组的一部分。这时，需要保留Stars-in联系。

类似的情况也适用于联系Owns。如果对于每部电影，都至少有一个包含电影、所属电影公司、参演影星的诸项内容合同，就可以不用Owns。但是，如果有电影公司的电影没有影星签约出演或数据库中没有这个合同，就必须保留Owns。

总之，无法明确告诉你一个特定联系是否多余。你必须从希望创建数据库的人那里找出他们需要什么。只有这样，你才能对是否包括像Stars-in或Owns这样的联系作出明智的选择。□

例2.16 现在，再看一下图2-2。在此图中，影星和电影公司之间没有联系。然而可以用联系Stars-in和Owns，并通过组合处理它们来创建影星和电影公司之间的连接。就是说，一位

影星被Stars-in连向某些电影，那些电影又被Owns连向电影公司。因此，可以说一位影星被连向一些电影公司，该公司拥有这个影星参演的电影。

如图2-12，在Stars和Studios之间创建一个联系Works-for有意义吗？在知道更多信息之前还不能判断。第一，这个联系的意思是什么？如果是“至少出演这个电影公司的一部电影的影星”，那么可能就没有理由把它包含进这幅图中了。因为可以从Stars-in和Owns推出这个信息。

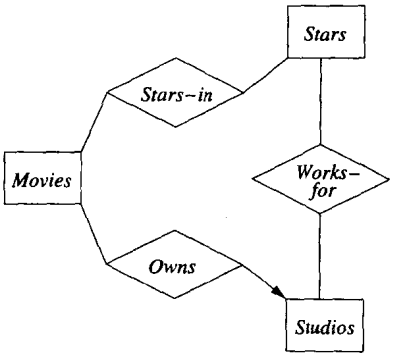


图2-12 在Stars和Studios之间加一个联系

然而，可以想像那些为电影公司工作的影
41 星的其他信息，并不被连向的电影所需要。如果那样的话，将影星和电影公司直接连接可能更有用，而不冗余。另一种选择，人们可能用一个影星和电影公司之间的联系来表达完全不同的意思。如，它可能表示这个影星与这个电影公司签约，而且在某种意义上跟任一部电影没有任何关系。正如在例2.7中所说的，一个影星与一家电影公司签约而为另一家电影公司的一部影片工作。这样的话，在新的联系Works-for中的信息就独立于联系Stars-in和Owns，而且肯定不是冗余。 □

2.2.5 选择正确的元素种类

有时人们可以选择不同的设计元素的类型表示现实世界。这样的选择很多介于是用属性还是使用实体集/联系之间。一般来说，属性比实体集或联系都易于实现。然而，并不能把一切事物都作为属性。

例2.17 考虑一个具体的问题。在图2-2中，把电影公司作为实体集是否明智？是否应该把电影公司的名字和地址作为电影的属性，并把实体集Studios除去？这样做的一个问题是得为每一部电影重复电影公司的地址。这是另一种冗余——与2.2.2和2.2.4节中看到的相似。此外，还面临着这样的危险，如果没有指定电影公司所拥有的任何一部电影，就将失去电影公司的地址。

另一方面，如果没有记录电影公司的地址，那么把电影公司的名字作为电影的一个属性并
42 没有危害，没有重复地址所产生的冗余；必须说出电影的名字（如像对迪斯尼拥有的每部电影说出Disney），这一事实并不是真正的冗余。 □

通过对例2.17所作的观察，可以总结出在哪些情况下宁愿使用属性而不使用实体集。假设E是个实体集。如果要把E用一个属性或几个其他实体集的属性代替，必须遵守下列条件：

1. 所有与E有关的联系都必须有指向E的箭头。就是说，E必须是多对一联系中的“一”，或者它对多重联系情况的推广。

2. E的属性必须总体上标识一个实体。如只有一个属性则肯定符合。然而，如果有几个属性，那么不能有属性依赖于其他属性，如像Studios中addrss依赖于name一样。

3. 没有联系包含E多次。

如果这些条件都符合，那么可以用下面的方式代替实体集E：

a）如果从实体集F到E有多对一联系R，那么删除R并把E的属性作为F的属性，如果名称跟F原来的属性名冲突则重命名。实际上，每个F实体把与E实体有关的惟一[⊖]名字作为属性，如

⊖ 在一个F实体不与任何E实体联系的情况下，F的新属性将被赋予特殊值“null”，以指明关联E实体不存在。相似的规律也适用于（b）中R的新属性。

同电影对象可以把电影出品公司的名字作为一个属性一样，此时无需考虑电影公司地址。

b）如果有多路联系R的箭头指向E，把E的属性作为R的属性，并删除从R到E的弧。一个转换的例子是用图2-7代替图2-8，在图2-8中引进了一个新实体集Salaries，它有一个数字作为惟一的属性。

例2.18 将使用多路联系和使用附带若干二元联系的连接实体集进行一下比较。图2-6中影星、电影和两家电影公司之间有四路联系。在图2-9中是机械地把它转换成实体集Contracts。它是否关系到我们所作的选择?·

从问题所表述的内容看，两种方式都是适合的。然而，如果将问题稍稍变化，就不得不选择连接实体集的方式。假设合同包括一位影星、一部电影和任何一组电影公司。情况比图2-6中的复杂，在那里电影公司可承担两个角色。在这里，合同中可以包含任意数目的电影公司，可 43
能一个负责制作，一个负责特别效果，一个负责发行，等等。此时无法给电影公司分配角色。

看起来，联系Contracts的联系集必须包括如下形式的三元组：

```
(star, movie, set-of-studios)
```

联系Contracts本身不仅包含通常的Stars和Movies实体集，而且还有一个新的实体集，它的实体是电影公司类型。虽然不能阻止采用这种方法，但把电影集合看做基本实体显得并不自然，不建议这样做。

一个更好的方法是把合同看做为一个实体集。如图2-9，合同实体连接影星、电影和一组电影公司，但是现在无须限制电影公司的数目。因此，如果合同是真正的“连接”实体集，合同和电影公司之间的联系是多对多，而不是多对一。图2-13画出了该E/R图。注意与合同相联系的有单独一位影星和单独一部电影和任意数目的电影公司。 □

2.2.6 习题

习题2.2.1 图2-14是一家银行数据库的E/R图，包括客户和账户。因为每个客户可以有几个账户，而账户可以被几个客户共同拥有，故将每个客户与一个“账户集”相关联，而账户是一个或几个账户集的成员。假设各种联系和属性的意思正如字面意思所示，请评判这个设计。它违反了什么设计规则？为什么？你建议怎么修改？ 44

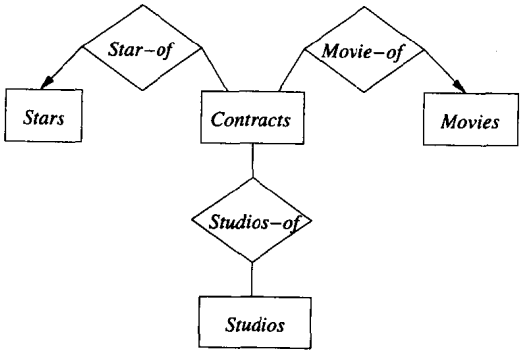


图2-13 合同连接一位影星、一部电影和一组电影公司

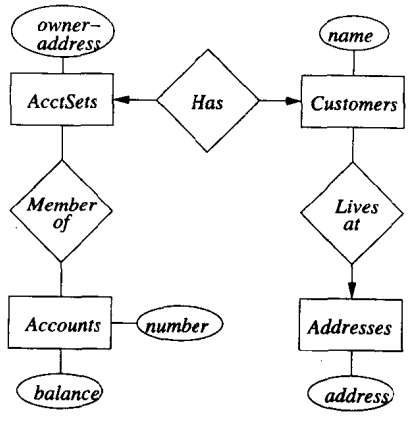


图2-14 银行数据库的一个拙劣设计

* **习题2.2.2** 你认为在什么情况下（考虑Studios和Presidents的隐含属性）可以把图2-3中的两个实体集和联系结合起来成为单一的实体集和属性？

习题2.2.3 假设在图2-7中删除Studios的属性address。考虑如何用一个属性代替一个实体集。那个属性应出现在什么地方?

习题2.2.4 给出可能的替代图2-13中下列实体集的属性:

a) Stars。

b) Movies。

! c) Studios。

!! **习题2.2.5** 在本题和下题中，考虑用E/R模型表示的两种描述出生的设计方案。在一次出生中，有一个婴儿（双胞胎用两次出生来表示）、一位母亲、任意数目的护士和任意数目的医生。因此假设，有实体集Babies、Mothers、Nurses和Doctors。假设用一个联系Births连接上述四个实体集，如图2-15所示。注意Births的联系集元组为如下形式:

```
(baby, mother, nurse, doctor)
```

如果有多个护士或医生接生，那么就会有若干个包含同一婴儿和母亲的元组，对应于
45 一种护士和医生的组合。

可将某些假设合并到设计中。为了表示假设，对每一个假设，说明如何把箭头或其他元素加到E/R图中。

a) 每个婴儿只有惟一的母亲。

b) 每一个婴儿、护士和医生的组合，有相应的惟一母亲。

c) 每一个婴儿和母亲的组合，有相应的惟一医生。

! **习题2.2.6** 习题2.2.5的另一种解决方案是用实体集Births连接四个实体集Babies、Mothers、Nurses和Doctors，在Births和它们之间分别用四种联系，如图2-16所示。用箭头（表示某些联系是多对一）表示下列条件:

a) 每个婴儿是惟一一次出生的结果，每次出生只有惟一的婴儿。

46 b) 除条件(a)外，每个婴儿有惟一的母亲。

c) 除条件(a)、(b)外，每次出生只有惟一的医生。

你从两种方案中分别看出哪些缺陷?

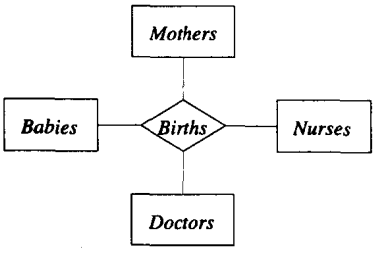


图2-15 用多路联系表示出生

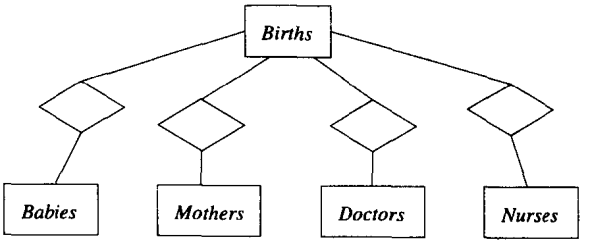


图2-16 用一个实体集表示的出生

!! **习题2.2.7** 假设改变观点，允许一个母亲一次生产多个婴儿。在此情况下，你怎样用习题2.2.5和2.2.6的方法去表示每个婴儿仍然只有惟一的母亲?

2.3 约束的建模

迄今人们已经知道了怎样把现实世界中的一小部分用实体集和联系来模拟。但仍有一些现实世界的重要方面无法用现有工具模拟。这些信息经常表现为附加在数据上的约束（constraint）形式，它已不再是实体集、属性、联系的定义所限制的结构和类型约束。

2.3.1 约束的分类

下面是常见约束的大致分类，而并非全部的分类。剩下的部分将在5.5节讲关系代数和第7章讲SQL编程时涉及。

1. 键（key） 是实体集中惟一标识一个实体的属性或属性集。不存在两个实体其构成键的所有属性值都相同，但部分相同是允许的。

2. 单值约束（single-value constraint） 是指值在某种情况下有惟一性的要求。键是单值约束的主要来源，因为它要求一个实体集中的每个实体在键属性上都有惟一值。也有其他单值约束来源，如多对一联系。

3. 引用完整性约束（referential integrity constraint） 是指要求某个对象所引用的值必须在数据库中实际存在。引用完整性与禁止悬挂指针或传统编程中的悬挂引用很相似。

4. 域约束（domain constraint） 要求属性的值必须在一个具体的值集或范围里。

5. 一般约束（general constraint） 是需要在数据库中得到满足的任意要求。例如，可以要求任一部电影中列出的影星数不超过十位。在5.5和7.4节中介绍一般约束表达式语言。 47

这些约束的重要性体现在几个方面。它们讲述了要模拟的现实世界各个方面的结构。例如，键让用户无混淆地标识实体。如果知道属性name是实体集Studios的键，那么就可以通过指定电影公司的名字来指定惟一的实体。另外，因为存储单值比存储集合容易，即使是那个集合中只有一个成员也如此[⊖]，所以，知道惟一值会节省空间和时间。引用完整性约束和键还支持可使数据访问更快的存储结构。这将在第13章中讨论。

2.3.2 E/R模型中的键

实体集E的键（key）是有一个或多个属性的集合K，对来自于E的不同实体e_1和e_2，它们包含在键K中的属性没有完全相同的值。如果K由多个属性组成，对于e_1和e_2虽然它们可以部分相同，但决不会全部相同。有三点需要记住：

- 键可以由多个属性组成，如例2.19。
- 一个实体集也可以有多个键，如在例2.20中将会看到的那样。但是，习惯上只选择一个作为“主键”，就好像是仅有的键。
- 当一个实体集处于一个isa层次中时，要求根实体集拥有键所需的所有属性，并且每个实体集的键都可在根实体集中找到它的组成部分，而不论在层次中有多少个实体集有它的组成部分。

例2.19 考虑例2.1中的实体集Movies。人们一开始可以假定属性title单独构成键。可是，有几个片名用于两部或多部电影，如King Kong。所以，title单独作为键是不明智的。如果一定要这样做，就不能在数据库中同时包含两部名为King Kong的电影了。

约束是模式的一部分

人们观察某一时刻存在的数据库时，可能会因为那里没有两个实体在某一属性上相同，就错误地决定用此属性作为键。例如，在创建电影数据库时没有电影同名。所以看起来title可以作为键。然而，如果人们决定了用title作键，而且设计了一个用title作键的数据库存储结构后，就会发现他不能再把第二个名为King Kong的电影输入数据库了。

因此，键约束和一般约束都是数据库模式的一部分。它们是数据库设计者连同结构

⊖ 类似地，注意在C编程中表示一个整数比表示一个整数链表简单，即使链表中只含有一个整数。

设计（如实体和联系）一块声明的。一旦一个约束被声明了，对数据库的任何违反约束的插入或修改操作都是不允许的。

因此，虽然一个特定的数据库的实例可能会满足某个特定约束，但是惟一“真正”的约束是由设计者指明的，能正确模拟现实世界的所有数据库实例的约束。这些约束是可以由用户和数据库存储结构采用的约束。

一个更好的选择是把由属性title和year构成的集合作为键。即使这样仍然存在两部同名电影在同一年出品的危险（此时不能同时将它们存入数据库），尽管这种情况不太可能出现。

关于例2.1中的另外两个实体集Stars和Studios，必须仔细考虑用什么可以作键。对电影公
48 司，有理由假定没有两家电影公司同名，所以可以把name作为Studios的键。但影星只用名字标识就不那么明确了。可以肯定，一般来说名字不能完全区分不同的人。可是，由于影星一般都有一个艺名，所以也可用这个名字作为Stars的键。不然就用name和address一对属性作为键，这应该令人满意了，除非有两位同名影星住在同一个地方。 □

例2.20 例2.19可能会导致人们认为找键是一件很困难的事，或者确信要由属性集构成键。现实中事情往往更简单。在数据库所模拟的现实世界中，人们经常不厌其烦地针对实体集创造出一些键。例如，公司一般为所有职工分配职工号ID，这些ID都是经过精心挑选以达到惟一性的。ID的目的之一就是在公司数据库中把每一位职工同其他职工区分开——即使有人同名。这样，职工号ID就可作为数据库中的键了。

49 在美国的公司，一般每位职工都有社会保障号。如果数据库有一个属性为社会保障号，那么这个属性也可以作为键。注意，一个实体集有几个键的选择并没有错，就像职工可以同时有职工号ID和社会保障号一样。

创造一个属性作为键的想法是非常普遍的。除了职工号ID，人们发现学号ID可以区分大学的学生。在机动车部门，驾照号码和汽车注册号码可以分别区分驾驶员和汽车。读者无疑可以找到更多这样的例子。 □

2.3.3 E/R模型中键的表示

在E/R图中，一个实体集的键属性用下划线标出。例如，图2-17是图2-2的重画，其中的键属性带有下划线。属性name是Stars的键。类似地，Studios的键是它的属性name。这些键的选择与例2.19的讨论一致。

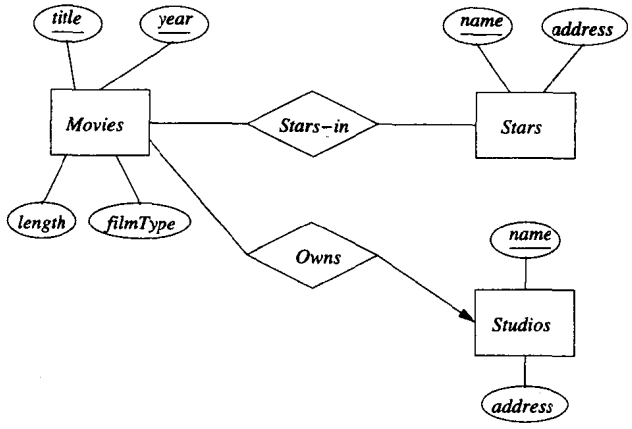


图2-17 用下划线标出键的E/R图

如例2.19中所讨论，属性title和year一起构成Movies的键。注意，有下划线的属性都是键的成员，如图2-17所示。当一个实体有几个键时，只在主键下划线。还要注意在一些不常见的情况下，一个实体集的键属性并不完全属于它自己，这叫做“弱实体集”，将稍后在2.4节讨论。 50

2.3.4 单值约束

通常，数据库设计的一个重要特征是至多只有一个值起着特殊的作用。例如，每个电影实体有惟一的名字、出品年份、片长和电影类型，并且一部电影只属于惟一的电影公司。

在E/R模型中有几种表示单值约束的方式。

1. 一个实体集的每个属性仅有一个值。有时，某些实体的某一属性可以没有值，此时就得创建一个“空值”作为此属性的值。例如，人们可以想像，数据库中的某些电影片长未知。此时可以用-1代表片长未知。另一方面，人们不希望一电影实体的键属性title或year为空值。某一属性不可为空值的要求在E/R模型中还没有特殊的表示。如果有此要求，可以在属性旁加个记号声明。

2. 从实体集E到F的多对一联系R隐含着一个单值约束。就是说，对E中的每一个实体e，在F中最多只有一个关联实体f。更一般地，如果R是多路联系，每个从R出去的箭头都意味着一个单值约束。特殊情况下，如果从R到实体集E有一个箭头，那么与E相关联的实体集中每个实体至多对应E的一个实体。

2.3.5 引用完整性

单值约束是要求对一给定角色至多只有一个值存在，而*引用完整性约束*（referential integrity constraint）是要求对应那个角色正好只有一个值存在。人们可以见到某个约束要求属性具有非空的值，并且单值作为一种引用完整性要求是需要的。但是“引用完整性”更多地用来指实体集之间的联系。

考虑一下图2-2中从Movies到Studios的多对一联系Owns。简单地说，多对一要求是指没有电影可以被一个以上电影公司所拥有，但并不是说一部电影必须被一个电影公司所拥有，或者，如果它被某个电影公司所拥有，当存入数据库时，那个电影公司就必须出现在实体集Studios。

联系Owns上的引用完整性将要求对每部电影，其所属的电影公司（由这部电影的联系“所引用”的实体）必须存在于数据库中。可以通过几种方式来强制这个约束。 51

1. 可以禁止被引用实体的删除（如例子中的电影公司）。即不能从数据库中删除任何电影公司，除非它不拥有电影。

2. 如果一个被引用的实体被删除，那么引用它的所有实体都要被删除。例子里要求如果删除了一家电影公司，还要删除其拥有的所有电影。

除了要求这样一种删除策略之外，还要求当一个电影实体插入数据库时，给定一个已经存在的Studios实体，由联系Owns与其相连。如果那个联系的值变了，新值也必须是一个已经存在的Studios实体。强制执行这些规则以保证联系的引用完整性属于数据库实现范畴，在此不讨论其细节。

2.3.6 E/R图中的引用完整性

人们可以扩展E/R图中的箭头标记，来表示一个联系是否在一个或多个方向上支持引用完整性。假设R是从实体集E到实体集F的联系，就可以用圆箭头指向F表示此联系从E到F不仅是多对一或一对一，而且要求与给定的E实体相联系的F实体必须存在。当R是多个实体集之间的联系时同样如此。

例2.21 图2-18显示了实体集Movies、Studios和Presidents之间一些引用完整性的约束。对这些实体集和联系的初次介绍是在图2-2和2-3中。图中可以看见一个圆箭头从联系Owns指向Studios。此箭头表示了每部电影必须被一个电影公司所拥有的引用完整性约束，而且要求相连的电影公司必须在Studios实体中。

类似地，图中还出现一个从Runs到Studios的圆箭头。它表示的引用完整约束是说每位经理经营一家存在于Studios实体集中的电影公司。

注意，从Runs到Presidents仍然是尖箭头。这个选择反映了一个电影公司及其经理间的合
52 理假定。如果一个电影公司不存在了，相应的经理也就不需要了，此经理应从实体集Presidents中删去，因此有一个圆箭头指向Studios。另一方面，如果从数据库中删去经理，那家电影公司会继续存在。因此使用通常的尖箭头，表示每家电影公司有至多一位经理，有时会没有经理。 □



图2-18 显示引用完整性约束的E/R图

2.3.7 其他类型的约束

如本节开头所说，一个数据库中可能会有其他类型的约束。这里只简单提及，其详细讨论是在第7章。

域约束（domain constraint）是指属性值必须取自一个有限集的约束。一个简单的例子便是属性类型的声明。更强的域约束可能是为属性声明枚举类型或值的范围，如电影的length属性必须是从0到240之间的整数。在E/R模型中域约束没有特殊的记号，如果需要，可以在属性旁边放一个记号，说明想要的约束。

仍有很多的常见约束没有包含入本节的分类中。例如，可以选择对联系的程度加以约束，如电影实体连到的影星实体不能多于10位。在E/R模型中可以在联系到实体的连线上加一个数字，表示相关实体集中可被联系到的实体数目约束。

例2.22 图2-19显示了如何在E/R模型中表示出演一部电影的影星不能超过10位的约束。作为另一个例子，还可以把箭头认为是约束“≤1”的同义词，可以把图2-18中的圆箭头认为是约束“=1”。 □



图2-19 每部电影的影星数目约束

2.3.8 习题

习题2.3.1 对题目：

* a) 习题2.1.1

53 b) 习题2.1.3

c) 习题2.1.6

的E/R图完成如下工作：

i) 选择并指明键

ii) 指出适当的引用完整性约束

! **习题2.3.2** 可以认为联系像实体集一样也有键。设R是实体集E_1、E_2、…、E_n之间的联系。那么R的键是从E_1、E_2、…、E_n的属性中选出的属性集K，使得如果（e_1、e_2、…、e_n）和

（f_1、f_2、…、f_n）是R联系集中的两个不同元组，那么这两个元组就不可能在K的所有属性上全相同。现在，假设$n=2$，也就是说，R是二元联系。并且，对所有i，设K_i是一属性集，表示实体集E_i的键。根据E_1和E_2的项，对下述R给出一个最小的键：

a) R是多对多联系

* b) R是从E_1到E_2的多对一联系

c) R是从E_2到E_1的多对一联系

d) R是一对一联系

!! **习题2.3.3** 继续考虑习题2.3.2的问题，允许n是任意值而不仅是2。仅利用从R到E_i的某些连线有箭头的信息，说明如何根据K_i的项找到R的一个最小键K。

! **习题2.3.4** 举出在实际生活中主要是为用作键而创建属性的例子（与例2.20不同）。

2.4 弱实体集

偶尔有这样的情形，一个实体集的键是由另一个实体集的部分或全部属性构成。这样的实体集叫做*弱实体集*（weak entity set）。

2.4.1 弱实体集的来源

弱实体集主要有两个来源。第一，有时实体集处在一个与2.1.11节的“isa 层次”无关的层次体系中。如果集合E中的实体是集合F中实体的一部分，那么可能仅仅只考虑E实体的名字将不具有惟一性，直到再考虑了E实体所属的F实体的名字后惟一性才成立。下面用几个例子说明这个问题。

54

例2.23 一个电影公司可能有几套拍摄班子（或称工作室）。这些拍摄班子可能被一家电影公司指定为crew1、crew2等等。可是，其他电影公司也可能用相同的编号，于是，number属性不能成为拍摄班子的键。为了惟一地命名一套拍摄班子，需要同时给出它所属电影公司的名字和它本身的编号。图2-20显示了这种情况。弱实体集Crews的键是它自己的number属性和Studios的name属性，Crews和Studios通过多对一联系Unit-of[⊖]相连接。 □

例2.24 一个种由它的属和它的种名所确定。例如，人是Homo sapiens（智人类）种；Homo（人）是属名，sapiens（现代人）是种名。一般而言，属由几个种构成，每个种都有一个名字，它以属名开头，继以种名。但是，种名自身并不惟一。不同的属可能有相同种名的种。为了惟一地指定一个种，需要将种名和它被Belongs-to联系连接到的属名结合起来，如图2-21所示。Species是个弱实体集，部分键来自于它的种。 □

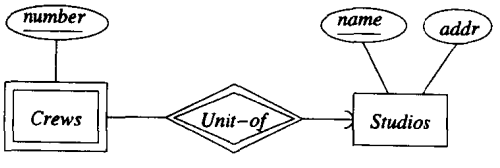


图2-20 弱实体集Crew和它的连接

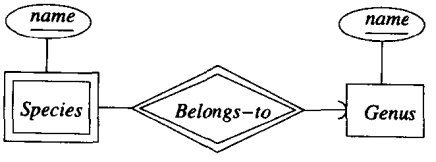


图2-21 另一个弱实体集：属

第二个弱实体集的来源是2.1.10节介绍的连接实体集，它被作为消除多路联系的一种方法[⊖]。这些实体集经常没有自己的属性。它们的键是由它们所连接的实体的键属性构成。

55

⊖ 双边菱形和双边方框将在2.4.3节解释。

⊖ 记住，在E/R模型中对多路联系的消除没有特殊要求，虽然在某些设计模型中有这种要求。

例2.25 在图2-22中可以看到连接实体集Contracts代替了例2.5中的三重联系Contracts。Contracts有一个属性salary，但并不是键。合同的键由电影公司的名字、参演影星的名字、电影名字和出品日期组成。 □

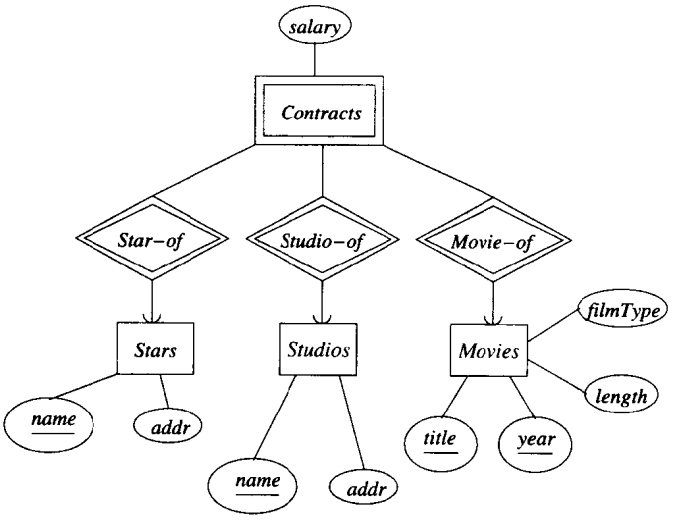


图2-22 连接实体集是弱实体集

2.4.2 弱实体集的要求

人们不能不加选择地去获取弱实体集的键属性。相反，如果E是弱实体集，则它的键组成包括：

1. 若干个它自己的属性（也可没有）；

2. 通过某些多对一联系从E到达其他实体集的键。这些多对一联系称为E的*支持联系*（supporting relationship）。

为了使从E到某个实体集F的多对一联系R成为E的一个支持联系，必须具备下面的条件：

56 a) R必须是从E到F的二元的多对一联系[⊖]。

b) R必须有从E到F的引用完整性。也就是说，对于每个E实体，经R与它相联系的F实体都必须存在于实际数据库中。换言之，必须有一个从R到F的圆箭头。

c) F提供给E作键的属性必须是F的键属性。

d) 然而，如果F本身就是弱实体集，那么F提供给E的部分或全部键属性是F由支持联系连接的一个或多个实体集G的键属性。同样的，如果G是弱实体集，则G的某些键属性又将由另一个实体集提供，如此继续下去。

e) 如果从E到F有多个不同的支持联系，那么每个联系都会被用来提供一份F的键的拷贝以帮助E的键的形成。注意，E中的实体e通过不同的支持联系被连接到F中的不同实体。因此，F的几个不同实体的键可能会同时出现在标识E的一个特定实体e的键值中。

这些条件的直观解释如下。考虑弱实体集中的一个实体，比如例2.23中的拍摄班子。抽象地讲，每个拍摄班子是惟一的。人们能够区分不同的班子，即使它们有相同的编号，但是属于不同的电影公司。所以拍摄班子的信息中仅包含编号还是不够的。如果存在某种决定性过程能够导出附加属性值，使得摄影班子能被惟一指定，那么这就是能够把这些附加信息与拍摄班子

⊖ 记住，一对一联系是多对一联系的特例。当说联系必须是多对一时，总是也包括了一对一联系在内。

关联起来的仅有方法。但是，仅与一个抽象的拍摄班子实体联系的惟一值是：

1. 实体集Crews的属性值；

2. 依据从一个摄影班子实体到某个其他实体集的惟一实体的联系所获取的属性值，这里其他实体有某种类型的惟一关联值。就是说，这个依据的联系必须是指向另一个实体集F的多对一（或特殊情况下的一对一）联系，并且相关联的值必须是F的键的一部分。

2.4.3 弱实体集的符号

用下面的约定来描述一个实体集是弱实体集，并且声明它的键属性。

1. 如果一个实体集是弱实体集，它就被显示为双边的矩形，例如图2-20中的Crews和图2-22中的Contracts。

2. 它的多对一支持联系显示为双边的菱形，例如图2-20中的Unit-of和图2-22中全部三个联系。 57

3. 如果一个实体集提供了构成它自己键的属性，则那些属性带有下划线。如图2-20给出的例子，图中摄影班子的编号虽然不是Crews键的全部，但是它是其自身的键的一部分。

可以用下面的规则概述这些约定：

- 无论何时用到，但凡实体集E有双边，它就是弱实体集。E中带下划线的属性（如果有的话），加上E被双边的多对一联系指向的实体集的键属性，必定对E的实体是惟一的。

人们应该记住双边的菱形只用于支持联系。从弱实体集引出的多对一联系有可能不是支持联系，因此也就不是双边的菱形。

例2.26 在图2-22中，联系Studio-of不必要作为Contracts的支持联系。原因是每部电影只属于一家电影公司，由从Movies到Studios的多对一联系（未显示）决定。因此，如果知道了一位影星和一部电影的名字，那么在这位影星、这部电影和它们所属的电影公司之间最多只有一项合同。根据记号所表达的意思，图2-22中的Studio-of更适合用一个普通的单边菱形而不是双边菱形。 □

2.4.4 习题

* **习题2.4.1** 表示学生和他们的课程成绩的一种方式是用相应于学生、课程和“注册”的实体集。注册实体集构成学生和课程之间的“连接”实体集，它不仅可以用来表示一个学生选了某门课，而且可以表示选此课的学生的成绩。为此情况画一个E/R图，指出弱实体集和它的键。成绩是注册实体集键的一部分吗？

习题2.4.2 修改你给出的上题的答案，以便可以记录学生一门课的每次作业的成绩。同样，指出弱实体集和它的键。

习题2.4.3 在习题2.2.6（a）-（c）的E/R图中，指出弱实体集、支持联系和键。

习题2.4.4 为下面的情况画出包括弱实体集的E/R图。对每种情况指出实体集的键。

a) 实体集Courses和Departments。一门课只有惟一的系开设，但它仅有的属性是它的编号。不同系可以开设具有相同编号的课。每个系有惟一的名称。 58

*! b) 实体集包括Leagues、Teams和Players。体育协会的名字是惟一的。体育协会中没有同名的队。队中也没有两个相同编号的选手。可是，不同队中可以有相同编号的选手，不同体育协会中可以有同名的队。

2.5 小结

- *实体/联系模型*：在E/R模型中描述了实体集、实体集之间的联系、实体集和联系的属性。实体集的成员叫做实体。

- E/R图：分别用矩形、菱形和椭圆来表示实体集、联系和属性。
- 联系的多样性：二元联系可以是一对一联系、多对一联系、或多对多联系。在一对一联系中，两个实体集中的任一个实体至多只能与另一个实体集中的一个实体关联。在多对一联系中，“多”一方的每个实体至多只能与另一方的一个实体关联。多对多联系对多重性不加限制。
- 键：可以惟一决定一个给定实体集中实体的一组属性是该实体集的键。
- 好的设计：高效地设计数据库，要求忠实地表达现实世界，选择合适的元素（如联系、属性），避免冗余——冗余是指一件事表示了两次，或者是用一种间接的或过度复杂的方式表示一件事。
- 引用完整性：当一个实体由给定的联系连向另一实体集中的一个实体时，要求后者必须存在于数据库中称为引用完整性约束。
- 子类：E/R模型用一个特殊的联系isa表示一个实体集是另一个实体集的特例。实体集可能连在一个层次结构中，其中每个子节点都是它的父节点的特例。只要子树包含根，那么实体集可以拥有属于此层次中任意子树的组成部分。
- 弱实体集：在E/R模型中有一种复杂情况是弱实体集，它需要用该实体集所连接的实体集
59 的属性来确定它自己的实体。用双边的矩形和菱形来区分弱实体集。

2.6 参考文献

[2]是实体/联系模型的第一篇文章，[1]和[3]是关于E/R设计的较新的书。

1. Batini, Carlo., S. Ceri, and S. B. Navathe, and Carol Batini, *Conceptual Database Design: an Entity/Relationship Approach*, Addison-Wesley, Reading MA, 1991.

2. Chen, P. P., “The entity-relationship model: toward a unified view of data,” *ACM Trans. on Database Systems* **1**:1, pp. 9–36, 1976.

3. Thalheim, B., “Fundamentals of Entity-Relationship Modeling,” Springer-Verlag, Berlin, 2000.
60

第3章 关系数据模型

尽管第2章中所介绍的实体-联系数据模型方法对于描述数据结构来说既简单又适合，但现今的数据库实现通常使用另一种方法，称做*关系模型*（relational model）。关系模型非常有用是因为它只有一个数据模型概念："关系"，即组织数据的一个二维表。在第6章将介绍关系模型怎样支持一种高级编程语言SQL（结构化查询语言）。SQL使我们只需编写简单的程序就可以对存储在关系中的数据进行强有力的操作，而E/R模型通常被认为不宜作为数据操作语言的基础。

另一方面，使用E/R符号易于设计数据库。因此，首先要考虑怎样把使用E/R符号描述的设计转换为关系，然后，研究关系自己的设计理论，即关系的"规范化"理论。该理论主要基于"函数依赖"，它包含并扩展了2.3.2节中已简要讨论的"键"概念。使用规范化理论，在对于在设计中如何选择关系去描述一个特定的数据库有改进作用。

3.1 关系模型的基础

关系模型为人们提供了一种描述数据的方法：一个称之为*关系*（relation）的二维表。图3-1就是一个关系的例子。关系名是Movies，它的目的是保存本书例子中实体集Movies的实体信息。图中的每一行对应一个电影实体，每一列对应电影实体集合的一个属性。除了描述实体集合之外，关系还能作更多的事情。

title	*year*	*length*	*filmType*
Star Wars	1977	124	color
Mighty Ducks	1991	104	color
Wayne's World	1992	95	color

图3-1 关系Movies

3.1.1 属性

位于关系最上一行的是*属性*（attribute）。图3-1中的属性是title，year，length以及film Type。关系的属性作为关系的列标题。通常，属性用来描述所在列的项目。例如，length属性列表示了以分钟为单位的每部电影放映时间。

注意，图3-1中关系Movies的属性和实体集合Movies的属性相同。虽然通常是把一个实体集合转化为具有同一属性集合的关系，但是，关系中的属性并不一定要与E/R图中的任一特定分量对应起来。

3.1.2 模式

关系名和其属性集合的组合称为这个关系的*模式*（schema）。描述一个关系模式时，先给出关系名，其后是用圆括号括起的所有属性。图3-1的Movies关系模式如下所示：

```
Movies(title, year, length, filmType)
```

关系模式中的属性是集合，而不是列表。然而，为了讲述关系，常常赋予属性一个"标准"顺序。当需要介绍具有一系列属性的关系模式时，常以这个标准次序将关系或关系的任意一行

显示出来。

一项关系模型设计包括一个或多个关系模式。这个关系模式的集合叫做*关系数据库模式*（relational database schema），或者就称做*数据库模式*（database schema）。

3.1.3 元组

关系中除含有属性名所在行以外的其他行称做*元组*（tuple）。每个元组均有一个分量（component）对应于关系的每个属性。例如，图3-1中，第一个元组有四个分量Star Wars、
62 1977、124和color，它们分别对应于属性title、year、length和filmType。若要单独表示一个元组，而不是把它作为关系的一部分时，常用逗号分开各个分量，并用圆括号括起来。例如：

```
(Star Wars, 1977, 124, color)
```

是图3-1的第一个元组的表示形式。从这个形式可看到，当单独表示元组时，属性不出现，因此要给出元组所在关系的标志，这个标志通常就是属性在关系模式中的排列次序。

3.1.4 域

关系模式要求元组的每个分量具有原子性。也就是说，它必须属于某种元素类型，如integer或string，而不能是记录、集合、列表、数组或其他任何可以被分解成更小分量的组合类型。

进一步假定与关系的每个属性相关联的是一个域（domain），即一个特殊的元素类型，关系中任一元组的分量值必须属于对应列的域。例如，图3-1关系Movies中元组分量所对应的域分别为：string、integer、integer、常量color或blackAndWhite。虽然域是关系模式的内容，但在6.6.2节中才给出定义域的方法。

3.1.5 关系的等价描述

关系是元组的集合，而不是元组的列表。因此关系中元组出现的顺序不是实质问题。例如，图3-1中三个元组有六种可能排列，却均表示同一个关系。

关系的属性次序可以任意重排，而关系不会改变。但重新排序关系模式时，要记住属性是列标题。因此，改变属性的次序时，也要改变它们所在列的次序。与此同时，元组的分量也要进行相应的移动，其排列方式应与属性的排列方式一致。

63 例如，图3-2是图3-1的行和列的多种重排方式中的一种。两个图表示的是同一个关系。

year	*title*	*filmType*	*length*
1991	Mighty Ducks	color	104
1992	Wayne's World	color	95
1977	Star Wars	color	124

图3-2 关系Movies的另一种表示

3.1.6 关系实例

关系movies不是静态的，而是会随时间改变的。人们希望变化会通过关系的元组表现出来。例如：插入新元组将电影加入到数据库中；对已存在的元组信息进行修改或更正；或者因为某种原因删除元组将电影从数据库去除等。

关系模式并不经常改变。然而，在有些情况下需要对属性进行添加和删除。在商业数据库系统中，模式可以改变，但代价是很昂贵的，因为可能会导致需要重写上百万个元组以添加或删除元组分量。若要添加一个属性，在既存元组中为对应分量找到正确赋值也很困难，甚至是不可能的。

一个给定关系中元组的集合叫做关系的实例（instance）。例如，图3-1中的三个元组就形成了关系Movies的一个实例。随着时间的流逝，Movies已经发生了改变，而且还将继续改变下去。例如，在1980年，Movies中并不包括Mighty Ducks和Wayne's World这两个元组。通常的数据库系统仅仅只维护关系的一个版本，即关系的“当前”元组集合。这个关系实例称做当前实例（current instance）。

3.1.7 习题

习题3.1.1 图3-3给出了构成部分银行数据库的两个关系实例。请指出：

a) 每个关系的属性。

b) 每个关系的元组。

c) 每个关系中元组的分量。

d) 每个关系的模式。

e) 数据库模式。

f) 每个属性的域。

g) 每个关系的另一个等价描述。

64

acctNo	*type*	*balance*
12345	savings	12000
23456	checking	1000
34567	savings	25

关系 Accounts

firstName	*lastName*	*idNo*	*account*
Robbie	Banks	901-222	12345
Lena	Hand	805-333	12345
Lena	Hand	805-333	23456

关系 Customers

图3-3 银行数据库中的两个关系

!! **习题3.1.2** 可用几种方法表示以下关系实例（考虑元组和属性的顺序）？

* a) 与图3-3中的关系Accounts一样有三个属性和三个元组。

b) 有四个属性和五个元组。

c) 有n个属性和m个元组。

3.2 从E/R图到关系设计

考虑一个新的数据库的建立过程，比如电影数据库。首先是设计阶段，这个阶段要考虑数据库存储什么信息、信息单元之间有什么联系、可以有哪些约束，如键约束或引用完整性约束，等等。因人们要对各种情况进行评价并在意见上达成一致，这个阶段需要很长一段时间。

接着是实施阶段，这时要使用一个已存在的数据库系统。因为大多数的商业数据库系统使用的是关系模型，故可以假定在设计阶段使用的也是关系模型，而不是E/R模型或其他面向设计的模型。

但在实践中先使用E/R图进行设计，再将其转化为关系模型会更加简单。这么做的主要原

65 因是关系模型仅有一个概念：关系，而不是像E/R模型那样具有许多复杂的概念（如实体集和联系）。这样，在设计方案确定后就可以充分利用它的不变性。

模式和实例

不要忽视关系模式和关系实例之间的区别。模式是关系名和关系属性，是相对不变的。实例是关系元组的集合，是经常变化的。

在数据模型中模式和实例的区别是很常见的，例如，实体集和联系在E/R模型中被用来描述模式，实体集和联系集形成了一个E/R模式的实例。不管怎样，要记住当设计数据库时，数据库实例不属于设计部分，只需想像有什么样的典型实例即可。

可简单明了地将E/R设计转换为关系数据库模式：

- 把每个实体集合转化为具有同一属性集合的关系。
- 用关系替换联系，关系的属性就是联系所连接的实体集的键集合。

这两条规则适用于大多数情况，要考虑的几种特殊情况是：

1. 弱实体集不能直接转化为关系。
2. “isa”联系和子类要特殊对待。
3. 有时需要把两个关系合并为一个，特别是当一个关系是从实体集*E*转化而成，而另一个关系由*E*到其他实体集的一个多对一的联系转化而来时，要考虑这种情况。

3.2.1 实体集到关系的转化

先不考虑弱实体集。弱实体集的转化需要作些修改，对此将在3.2.4节中讨论。对其他任一实体集，可创建一个同名且具有相同属性集的关系。因为实体集参与的联系不会在转化后的关
66 系中体现出来，所以还需要用单独的关系处理联系，这一点将在3.2.2中进行讨论。

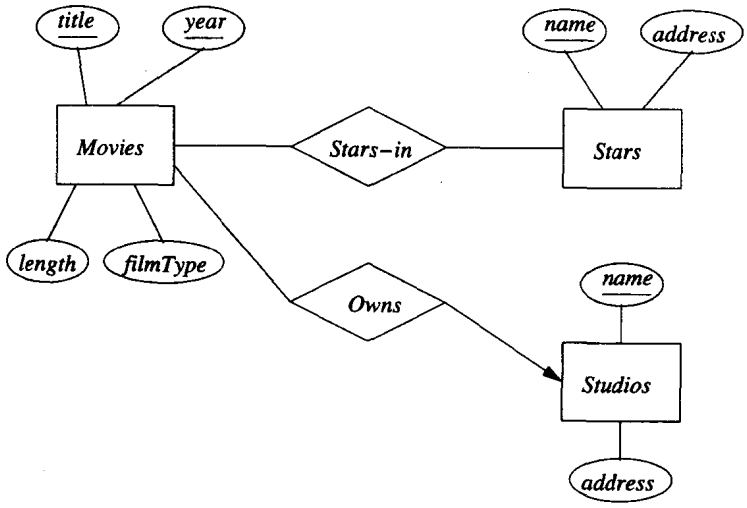


图3-4 电影数据库的E/R图

例3.1 考虑图2-17（重画为图3-4）中的三个实体集Movies、Stars和Studios。Movies实体集的属性分别是title、year、length和filmType。转化后的结果就是关系Movies，它和3.1节中给出的图3-1相似。

接着考虑图3-4中实体集Stars。它有两个属性name和address。因此，相应的关系`Stars`的模式为`Star(name, address)`。这个关系的一个典型实例如下所示：

name	*address*
Carrie Fisher	123 Maple St., Hollywood
Mark Hamill	456 Oak Rd., Brentwood
Harrison Ford	789 Palm Dr., Beverly Hills

□

3.2.2 E/R联系到关系的转化

E/R模型中的联系也可以用关系表示。一个给定联系R的关系有下列属性：

1. 联系R涉及的每一个实体集的键属性或键属性集都是由R转化成的关系模式的一部分。

2. 如果这个联系有属性，则它们也是由R转化成的关系中的属性。 67

如果一个实体集以不同的角色在联系中多次出现，则它的键属性出现的次数与角色出现次数相同。为了避免重名，必须对这些键属性重新命名。更普遍的情况是，只要R本身的属性和与其相连的实体集的键属性有同名，就要对这些属性重命名。

关于数据质量 :-)

为使举例中的数据尽可能的精确并尊重影星们的隐私权，我们使用了一些伪造的关于影星的地址和其他个人信息的数据。

例3.2 考虑图3-4中的联系Owns。它连接了实体集Movies和Studios。则Owns的关系模式中含有Movies的键属性title和year，以及Studios的键name。于是，它的关系模式为：

```
Owns(title, year, studioName)
```

这个关系的实例样本为：

title	*year*	*studioName*
Star Wars	1977	Fox
Mighty Ducks	1991	Disney
Wayne's World	1992	Paramount

为了清晰起见，这里选择studioName作为属性名，与Studios中的属性name相对应。 □

例3.3 同样，图3-4中的联系Star-in可转化成具有属性title、year（Movies的键属性集）和starName(Stars的键属性)的关系，图3-5给出了关系样本Star-In。

因为图3-5中电影的title值是惟一的，因此看上去year属性是多余的。但是，也会存在很多同名的电影，如“king kong”，所以有必要同时使用年份来区分哪位影星在电影的哪个版本中出演。 □

title	*year*	*starName*
Star Wars	1977	Carrie Fisher
Star Wars	1977	Mark Hamill
Star Wars	1977	Harrison Ford
Mighty Ducks	1991	Emilio Estevez
Wayne's World	1992	Dana Carvey
Wayne's World	1992	Mike Meyers

图3-5 从联系Stars-In得到的关系

例3.4 把多路联系转化为关系也很容易。考虑图2-6（在图3-6中重新给出）中的四路联系Contracts，涉及一个Star，一个Movie，两个
68 Studios——一个拥有影星，另一个制作电影。若用一个关系Contracts来描述，则关系模式包含以下四个实体集属性：

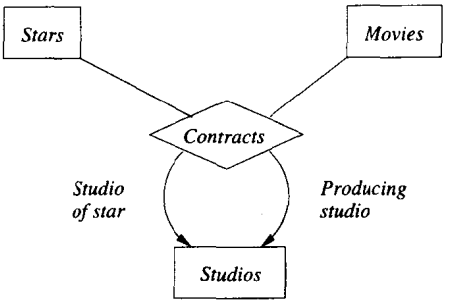


图3-6 联系Contracts

1. Star的键starName。
2. Movie的键title和year。
3. 标识第一家电影制片厂名字的键studioOfStar。前面曾假设电影制片厂的名字是实体集Studios的键。
4. 标识电影公司名字的键producingStudio，而这家制片厂制作某位影星出演的某部电影。

这就是说，这个关系模式为：

```
Contracts(starName, title, year, studioOfStar, producingStudio)
```

注意，在这个关系模式中使用的属性名并不是任何属性的“名字”，因为若这样的话，就
69 不知道它到底指的是影星的名字，还是制片厂的名字，若是制片厂名字，又是指哪一家制片厂？并且，如果实体集Contracts也有属性，比如salary，那么这些属性都要添加到关系Contracts的关系模式中。 □

3.2.3 组合关系

有时从实体集和联系转化而来的关系，对于给定的数据而言并不是最好的。一个较普遍的例子是，如果存在一个实体集E和一个从E到F的多对一联系R，则经转化后所得到的关系模式E和R都含有E的键属性。而且，从E得到的关系模式中还包含E中的非键属性，从R得到的关系模式中也包含F中的键和R中的所有属性。由于R是多对一联系，E的键已确定了这些属性的惟一值，因此可把它们组合在一个关系中，相应的模式包含：

1. E的所有属性。
2. F的键属性。
3. 联系R的任何属性。

对于一个不与F相连的E中的实体e而言，上述的2，3类属性在e元组中的相应分量上为空值。空值在2.3.4中已有过介绍，它的引入是为了描述失去值或未知值。空值不是关系模型的正式内容，记作NULL，在SQL语言中可用。本书采用E/R设计对关系数据库模式进行描述时使用了空值。

例3.5 在电影数据库的例子中，Owns是一个连接Movies和Studios的多对一联系。例3.2中给出了它到关系的转化，实体集合Movies到关系的转化也已在例3.1中给出。取出它们的属性进行连接，所得的关系模式如图3-7所示：

title	*year*	*length*	*filmType*	*studioName*
Star Wars	1977	124	color	Fox
Mighty Ducks	1991	104	color	Disney
Wayne's World	1992	95	color	Paramount

图3-7 组合关系Movies和Owns

之所以按这种方式组成关系，是因为把依赖于E的键属性的所有属性组合在一个关系中有
70 很多优点，即使是有多条从E到其他实体集的多对一的联系时也如此。例如，涉及一个关系的

属性查询比涉及多个关系的属性查询效率更高。实际上，一些基于E/R模式的设计系统可以自动地为用户组合这些关系。

另一方面，有人可能想把E和涉及E但并不是从E到其他实体的多对一联系R组合成关系。这么做是危险的，因为这样常常会造成冗余，关于冗余的问题将在3.6节讨论。 □

例3.6 为了说明可能产生错误组合，假设要将图3-7中的关系和多对多联系Stars-in组合起来，Stars-in在图3-5中给出。所得到的组合关系在图3-8中给出。

title	*year*	*length*	*filmType*	*studioName*	*starName*
Star Wars	1977	124	color	Fox	Carrie Fisher
Star Wars	1977	124	color	Fox	Mark Hamill
Star Wars	1977	124	color	Fox	Harrison Ford
Mighty Ducks	1991	104	color	Disney	Emilio Estevez
Wayne's World	1992	95	color	Paramount	Dana Carvey
Wayne's World	1992	95	color	Paramount	Mike Meyers

图3-8 带有影星信息的关系Movies

因为一部电影可以有多个影星，所以对于每一个影星，就不得不重复这部电影的所有信息。在图3-8中，*Star Wars*的length对应三个影星重复了三次，Fox也一样，因为这部电影是由Fox制作。这种冗余是不希望出现的，3.6节中讲述的关系数据库设计理论其目的就是要分离类似图3-8所示的关系以消除冗余。 □

3.2.4 处理弱实体集

若需转化E/R图中的一个弱实体集时，需要做下面三件事：

1. 从弱实体集W得到的关系不仅要包含W的属性，还包含有助于形成W的键的其他实体集的键属性。显而易见这样做有利于实体集，因为可从W引出的支持联系（双边菱形）访问到它们。

71

2. 与弱实体集W相连的联系，经转化后所得的关系必须把那些和W相连的，以及对W的键有用的，实体集的键属性作为W的键。

3. 然而，从弱实体集W指向其他有助于形成W的键的实体集的支持联系R不必转化为关系。理由就是，如同3.2.3节讨论的，由多对一联系R转化得到的关系的属性可从W的属性，也可以与W的关系模式进行组合（在R有属性的情况下）。

当然，当引入附加属性来建立一个弱实体集的键时，要注意不要重复使用同一个名字。如果有必要，要对其中部分或所有的属性进行重命名。

例3.7 考虑图2-20中的弱实体集Crews，该图在图3-9中重新给出。图表有三个关系，关系模式分别为：

```
Studios(name, addr)
Crews(number, studioName)
Unit-of(number, studioName, name)
```

第一个关系Studios是从同名实体集直接转化而来。第二个关系Crews是由弱实体集Crews转化得到，它的属性是相应弱实体集的键属性，如果弱实体集Crews还有非键属性，则也要包含在关系Crews模式中。可以用studioName作为关系Crews的属性来与实体集Studios的name相对应。

第三个关系Unit-of也由同名联系转化而来。根据前面的描述，它的属性由与联系

Unit-of相连的实体集合的键属性组成。在这个例子中，Unit-of的属性有number，studioName（弱实体集合Crews的键属性）和name（实体集合Studios的键属性）。要注意，由于Unit-of是个多对一的联系，studioName等同于name。

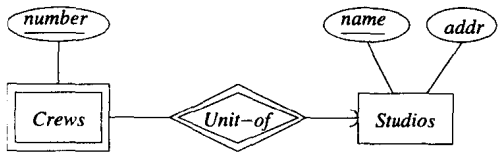


图3-9 弱实体集crews

带有子集模式的关系

根据例3.7，似乎只要关系R含有的属性集合是另一个关系S属性集合的子集，就可消除R。但是，这种说法并不完全正确。因为S的附加属性不允许把R的元组扩展成S的元组，所以R可能含有S中没有的信息。

例如，国家税收局（Internal Revenue Service）要维护关系People（name，ss#），其中含有可能的纳税人的名字和他们的社会保险号，而不管他们是否有收入。税收局还要维护关系TaxPayers（name，ss#，amount），其中的属性amount表示今年以来纳税人所交的税金。People的属性集合是TaxPayers属性集合的子集，但在People中保存着的纳税人的社会保险号在TaxPayers中却可能没有。

事实上，即使是同一个属性集合也会有不同的语义，所以不可能合并它们的元组。例如，对于两个关系Stars（name，addr）和Studios（name，addr），虽然它们看起来相似，却不能把star的元组改为studios的元组，反之亦然。

另一方面，如果两个关系分别由弱实体集构造和对应的支持联系转化过来，那么对于属性集合较小的关系而言，它就不包含另外的信息。这是因为支持联系得到的关系的元组与弱实体集所得到关系的元组一一对应。此时可以除去具有较少属性的关系。

72 假设3#工作室隶属于迪斯尼公司。那么联系unit-of应包含

(Disney-crew-#3, Disney)

而转化为关系后，应包含元组

(3, Disney, Disney)

从这个例子可以看到，属性studioName和name在转化后的关系中具有相同的含义。这就是说，可以把属性studioName和name合并起来，得出它的简单模式：

Unit-of(number, name)

73 此时由于Unit-of和Crews一样，就不用对它进行转化。 □

例3.8 下面考虑例2.25给出的弱实体集Contracts和2.4.1节中的图2-22。图2-22在图3-10中重新给出。Contracts转化后的关系模式为

Contracts(starName, studioName, title, year, salary)

其中，starName是对Stars键的重命名，studioName是对Studios键的重命名，title和year这两个属性形成了Movies的键，salary是Contracts自身的属性。联系Star-of，Studio-of，Movie-of没有相应的关系，但它们各自所形成的关系模式都是Contracts的真子集。

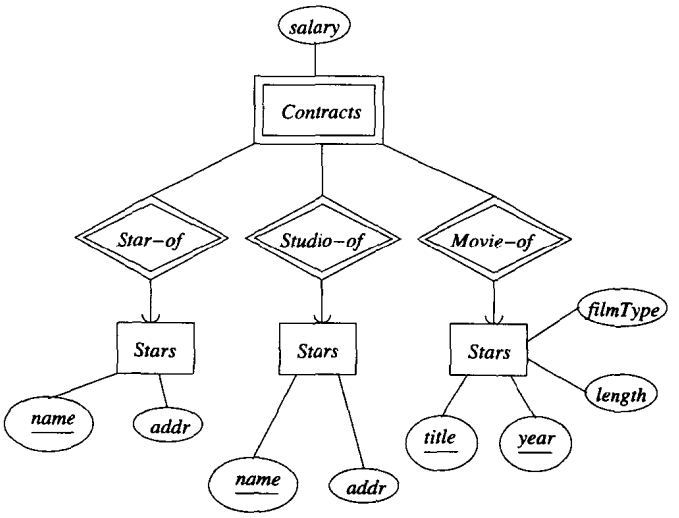


图3-10 弱实体集Contracts

顺带再说一下，上述关系与从图2-7所示E/R图获得的关系完全相同。图2-7中的Contracts是个三路联系，分别连接Stars，Movies和Studios，本身含有一个属性salary。 □

从例3.7和例3.8中可看出：支持联系不需要转化为关系。这一点对于弱实体集是通用的。下面给出将弱实体集转化为关系的修改规则。

• 若*W*是一个弱实体集，则*W*转化为关系后的模式由以下项目组成：

1. *W*的所有属性。
2. 与*W*相连的支持联系的所有属性。 74
3. 对每一个连接*W*的支持联系（即从*W*到实体集的多对一联系），要包含*E*的所有键属性。

为了避免同名冲突，必要时要对某些属性进行重命名。

• 不要为与*W*相连的支持联系构造关系。

3.2.5 习题

* **习题3.2.1** 把图3-11中的E/R图转化为一个关系数据库模式。

! **习题3.2.2** 存在另一个E/R图描述图3-11中的弱实体集Bookings。注意，机票可由航班号（flight number）、航班日期（day）、座位的排号（row）和座位号(seat)惟一确定，而与顾客的信息无关。

a) 对图3-11中的图进行修改，使其可以反映新的观点。

b) 把（a）中所得的图转化为关系。所得的关系数据库模式与习题3.2.1中的相同吗？

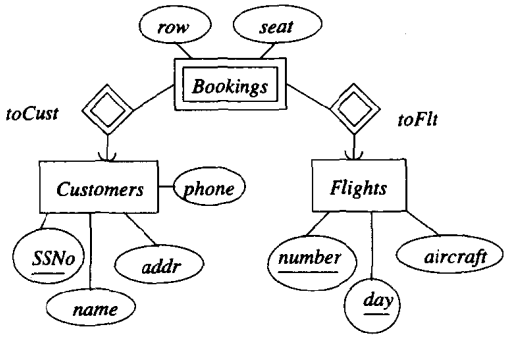


图3-11 关于飞机航班的E/R图

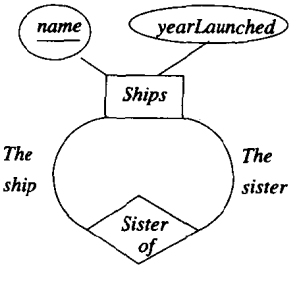


图3-12 姊妹船的E/R图

* **习题3.2.3** 图3-12中的E/R图表示船只，如果不同船只（ship）出自于同一份设计方案，则认为它们是姊妹（sister）。把这个图转化为关系数据库模式。

习题3.2.4 把下面的E/R图转化为关系数据库模式。

75

a) 图2-22。

b) 习题2.4.1的结果

c) 习题2.4.4(a)的结果。

d) 习题2.4.4(b)的结果。

3.3 子类结构到关系的转化

有几种策略把一个isa层次实体集转化为关系。对于这种层次，已有如下假定：

- 这个层次中有一个根实体集。
- 这个层次实体集有一个可惟一确定层次中每个实体集的键。
- 一个给定的实体可能会包含属于这个层次中某些子树的实体集的分量，只要这个子树包含根。

主要的转化策略是：

1. 遵照E/R观点。为任一个在层次中的实体集创建一个关系，它包含根的键属性和实体集自身属性。

2. 把实体看做属于单个类的对象。为每个包含根的子树创建一个关系，这个关系模式包括子树中所有实体集的所有属性。

3. 使用空值（null value）。创建一个包含层次中所有实体集属性的关系。每个实体由一个元组表示，对于实体不具有的属性，则设该元组的相应分量为空。

76

下面依次讨论上述方法。

3.3.1 E/R方式转化

第一种方法是同往常一样为每个实体集建立关系。如果实体集*E*不是层次中的根，为了能够区别用元组表示的实体，由*E*转化成的关系要包含根的键属性和*E*本身属性。并且，如果*E*和其他实体集间存在联系，就在此联系生成的关系中，使用这些键属性识别*E*中的实体。

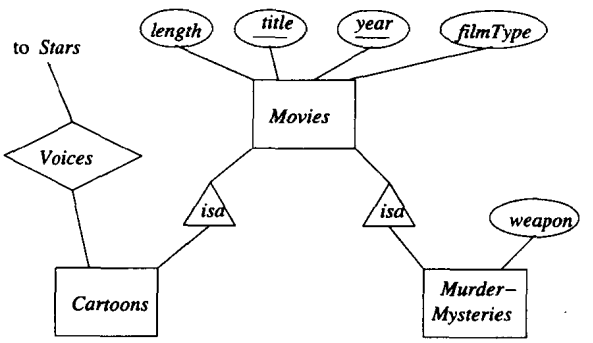


图3-13 movie层次

要注意的是，虽然“isa”被认为是联系，但它与其他联系不同。它关联的是单个实体的分量，而不是实体。因此，不能为“isa”创建关系。

例3.9 考虑图2-10给出的层次，图2-10由图3-13重新给出。这个层次中四个不同实体转化成的关系为：

1. `Movies(title, year, length, filmType)`。这个关系已在例3.1中进行过讨论，在这里每部电影由一个元组表示。

2. `MurderMysteries(title, year, weapon)`。前面两个属性是`Movies`的键，最后一个属性是实体集本身的属性。每部凶杀片在此关系中和在`Movies`关系中都对应一个

元组。

3. Cartoons(title, year)。这个关系是卡通片的集合。它只包含了Movies的键属性，这是因为关于卡通片的其他信息均包含在联系Voices中。每部卡通电影在此关系中和在Movies关系中分别对应一个元组。

要注意的是：第四种电影，那些既属于卡通片又属于凶杀片的电影在这三个关系中均有元组存在。 77

另外，还要构造与Stars和Cartoons之间联系Voices相对应的关系Voices(title, year, starName)。其中最后一个属性是Stars的键属性，前两个属性组成实体集Cartoons的键。

例如，电影*Roger Rabbit*作为元组将会在这四个关系中出现。它的基本信息将会在Movies中出现，凶手的凶器（weapon）在关系MurderMysteries中出现，为此电影配音的演员会在关系Voices中出现。

要注意关系Cartoons的模式是关系Voices模式的子集。在很多情况下，我们急于消去Cartoons关系，因为它并未包含与Voices有所不同的信息。然而，数据库中可能会有一些无声卡通片。这些卡通片没有配音，于是这些电影也是卡通片的信息就会被忽略掉。

3.3.2 面向对象的方法

把isa-层次转化为关系有另一个方法，就是枚举层次中所有可能的子树。为每一个子树构造一个可以描述该子树中实体的关系。这个关系模式含有子树中所有实体集的所有属性。因为这种方法的前提是假设这些实体是属于且仅属于一个类的“对象”，所以这种方法被称为“面向对象”的方法。

例3.10 考虑图3-13的层次。有四个可能的包含根的子树：

1. Movies本身。
2. 仅有Movies和Cartoons。
3. 仅有Movies和Murder-Mysteries。
4. 所有三个实体集。

下面给出这四个类构造的关系，因为只有Murder-Mysteries有自身的属性，因此实际上存在着重复，这四个关系是：

```
Movies(title, year, length, filmType)
MoviesC(title, year, length, filmType)
MoviesMM(title, year, length, filmType, weapon)
MoviesCMM(title, year, length, filmType, weapon)
```

如果Cartoons含有自身特有的属性，则以上四个关系将会有不同的属性集合。然而情况并不是这样，所以，虽然会丢失一些信息，比如丢失了属于卡通片的电影信息。人们仍将可以将Movies和MoviesC组合为不包含凶杀片的关系，也可以将MoviesMM和MoviesCMM组合为包含所有凶杀片的关系。 78

人们还需要考虑怎样处理连接Cartoons和Stars的联系Voices。如果Voices是一个从Cartoons引出的多对一的联系，就可以增加一个voice属性到MoviesC和MoviesCMM中，这个属性可以描述联系Voices，并且有使这四个关系模式不同的副作用。可是，Voices是一个多对多的联系，因此就要为这个联系单独创建一个关系。如同惯例，它的关系模式包含了与其相连的实体集的键属性。这种情况下它的模式为

```
Voices(title, year, starName)
```

人们可以考虑是否有必要建立两个这样的关系，一个连接不是凶杀片的卡通片到它的配音演员，另一个连接是凶杀片的卡通片与配音演员相关联。然而，这样做似乎并没有高明之处。□

3.3.3 使用空值组合关系

有很多种表示实体集层次信息的方法。如果允许NULL（就像SQL中的空值）作为元组值，就可以对一个实体集层次只创建一个关系。这个关系包含了层次中所有实体集的所有属性，一个实体表现为关系中的一个元组。元组中的NULL表示该实体没定义的属性。

例3.11 若把这种方法应用于图3-13，就可以得到相应的关系模式为：

```
Movie(title, year, length, filmType, weapon)
```

那些非凶杀类的电影在元组中的weapon为NULL。但有必要创建一个如例3.10中的voices关系，因为要用它来连接卡通片和为该卡通片配音的演员。□

3.3.4 各种方法的比较

上述三种方法分别被称为“直接E/R”，“面向对象”和“空值”方法。它们各自均有优缺点，这里列出其主要的几点。

1. 由于涉及几个关系的查询代价高昂，所以人们宁愿在一个关系中寻找查询需要的所有属性。“空值”方法对于所有的属性只使用一个关系，在这一点上它有很好的性能。而其他两种
79 方法更适合于其他类型的查询。例如：

(a) 若要查询“1999年放映时间长于150分钟的是哪部影片？”，可以直接从使用例3.9中的“直接E/R”方法得到的关系Movies中查到。可是，若使用例3.10中的“面向对象”的方法，就需要对Movies、MoviesC、MoviesMM和MoviesCMM进行查询，因为一部片长较长的电影可能会在这四个关系中出现[⊖]。

(b) 另一方面，若要查询“放映时间长于150分钟的卡通片中使用了什么武器？”，使用“直接E/R”方法会很麻烦。因为必须要访问Movies以找到长于150分钟的影片，接着访问Cartoons以查证这部影片是否存在，然后访问MurderMysteries以找到凶器。若使用面向对象的方法，则只需访问关系MoviesCMM就可以找到所需的所有信息。

2. 如果倾向于用尽量少的关系，就可使用“空值”的方法，因为它只需一个关系。然而其他两种方法之间也有不同点，在“直接E/R”方法中层次的每个实体集只转化为一个关系。而在“面向对象”的方法中，如果层次中有一个根和n个孩子(共有$n+1$个实体集)，则就会有2^n个不同的实体类，此时就要创建同样多个关系。

3. 有时可能倾向于减少空间和避免重复的信息。因为“面向对象”的方法对每个实体只使用一个元组，并且这个元组只含有对实体有意义的属性的分量，所以这种方法占用的空间最少。虽然“空值”法也是每个实体一个元组，但是这些元组的分量“太长”了。也就是说，它们含有针对任一属性的分量，而不管对于给定实体这些分量是否适合。如果层次中有很多的实体集，而这些实体集又有很多的属性，那么使用“空值”法就会浪费大量的空间。而“直接E/R”法中虽说每个实体对应多个元组，但只有键属性重复了多次。因此，“直接E/R”法相对于“空值”法而言，使用的空间可能多也可能少。

3.3.5 习题

* **习题3.3.1** 使用下面的方法，把图3-14中的E/R图转化为关系数据库模式：

⊖ 即使把这四个关系合并为两个，为了得到查询结果也必须访问所有的关系。

a)“直接E/R”法。
b)“面向对象”法。
c)“空值”法。

80

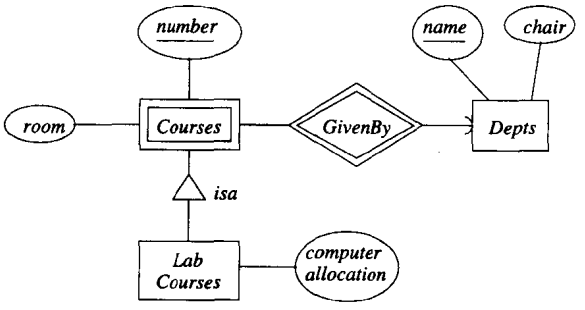


图3-14 习题3.3.1的E/R图

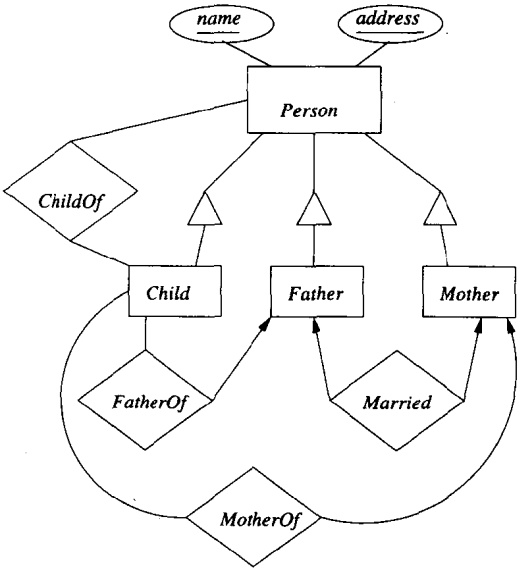


图3-15 习题3.3.2的E/R图

81

! **习题3.3.2** 使用下面的方法，把图3-15中的E/R图转化为关系数据库模式：
a)“直接E/R”法。
b)“面向对象”法。
c)“空值”法。

习题3.3.3 使用下面的方法，把习题2.1.7中设计的E/R图转化为关系数据库模式：
a)“直接E/R”法。
b)“面向对象”法。
c)“空值”法。

! **习题3.3.4** 假设有一个涉及实体集e的isa-层次。每个实体集都有a个属性，其中k个是根实体的属性，形成所有实体集的键。给出使用下面的方法转化为关系时，计算符合如下要求的公式（i）使用的最少和最多关系数（ii）单个实体元组最少和最多的分量数。
* a)“直接E/R”法。
b)“面向对象”法。
c)“空值”法。

3.4 函数依赖

3.2节和3.3节给出了怎样把E/R设计转化为关系模式的方法。对于数据库设计者来说，虽然从应用需求中直接产生关系模式会很困难，但并非不可能。不管关系设计如何产生，人们经常发现可基于某种约束类型对设计系统地加以改进。在关系模式设计上使用的最重要的约束类型是单值约束，称为“函数依赖”（缩写为FD）。在对数据库进行重新设计以减少冗余的过程中，这种约束类型的相关知识至关重要，这一点将在3.6节中见到。还有其他可使人们设计出优良的

82 数据库模式的约束类型。例如，3.7节中给出的多值依赖，以及5.5节中提到的引用完整性约束。

3.4.1 函数依赖的定义

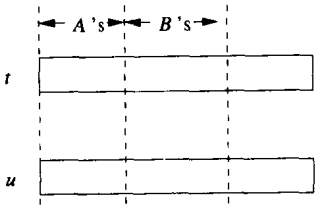
一个关系R上的函数依赖（functional dependency），即FD，是指“如果R的两个元组在属性A_1，A_2，...，A_n上一致（也就是，它们对应于这些属性的分量值相同），那么它们在其他分量B上的值必定也相同。写做：$A_1A_2\cdots A_n \rightarrow B$，叫做“$A_1$，$A_2$，…，$A_n$函数决定$B$。”

如果属性集A_1，A_2，…，A_n函数决定多个属性，即：

$$A_1A_2...A_n \rightarrow B_1$$
$$A_1A_2...A_n \rightarrow B_2$$
$$...$$
$$A_1A_2...A_n \rightarrow B_m$$

则可把这个FD集合缩写为

$$A_1A_2...A_n \rightarrow B_1\ B_2\ ...\ B_m$$



如果t和u在此一致
则它们必定在此也一致

图3-16 两个元组上函数依赖的影响

图3-16是对关系R中任意两个元组t和u的解释。

例3.12 考虑图3-8中的关系

```
Movies(title, year, length, filmType, studioName, starName)
```

该关系的一个实例在图3-17中给出。从Movies中可以合理地导出几个FD。例如，下面这三个FD：

```
title year → length
title year → filmType
```
83
```
title year → studioName
```

title	*year*	*length*	*filmType*	*studioName*	*starName*
Star Wars	1977	124	color	Fox	Carrie Fisher
Star Wars	1977	124	color	Fox	Mark Hamill
Star Wars	1977	124	color	Fox	Harrison Ford
Mighty Ducks	1991	104	color	Disney	Emilio Estevez
Wayne's World	1992	95	color	Paramount	Dana Carvey
Wayne's World	1992	95	color	Paramount	Mike Meyers

图3-17 关系Movies（title，year，length，filmType，studioName，starName）的实例

因为这三个FD箭头左边有相同的title和 year，所以可把它们缩写在一行上：

```
title year → length filmType studioName
```

非正式地讲，这个FD集合的含义是，若两个元组在分量title和year上具有相同的值，则这两个元组在分量length，filmType和studioName上的值也分别相同。这种断言与关系模式最初的设计意义一致。属性title和year形成了Movies实体集合的键。这样，只要给定片名和年份，就可惟一地确定一部电影，并确定一部电影的片长和类型。此外，还有一个从Movies到Studios的多对一联系。人们也希望，如果给定一部电影，则只有一家出品该影片的公司。

另外，下面的式子

```
title year → starName
```

是错误的，它不是一个函数依赖。因为完全有可能有很多影星参加同一部电影的演出。□

3.4.2 关系的键

如果下列条件满足，就认为一个或多个属性的集合{A_1，A_2，...，A_n}是关系R的键：

1. 这些属性函数决定关系的所有其他属性。因为关系是一个集合，也可以说，不可能存在

两个不同的元组，它们具有相同的A_1，A_2，…，A_n值。 84

2. 没有一个{A_1，A_2，…，A_n}的真子集能函数决定R的其他属性。也就是说，键必须是最小的（minimal）。

当键只包括一个属性A时，把它写为A，而不是{A}。

函数依赖对模式的解释

FD和任何一种约束一样，针对的是关系模式，而不是某个特定实例。不能仅通过一个实例确定FD。否则，从图3-17可以得到一个FD：title→filmType，因为对于关系Movies实例中的每一个元组来说，只要字段title值相同，则字段filmType也相同。

但是，不能就此断定这个FD对关系Movies成立。如果实例中包含了title为*King Kong*的两个元组，而它们的filmType属性值分别为color和blackAndWhite，那么所假设的FD就不成立。

例3.13 图3-17中关系Movies的键为{title、year、starName}。首先要证明它们函数决定了所有其他属性。也就是说，假设有两个元组在属性title、year和starName上的值相同，则相应的其他属性如length、filmType和studioName上的值也应该相同，这一点同例3.12中讨论的一样。因此，不同的元组在title、year和starName上取值应不完全相同；否则，应是指同一个元组。

下面将讨论的是{title、year、starName}的任一真子集都不能函数决定其他的属性。首先看，图中的属性title和year不能确定starName，这是因为有许多电影是由多个影星参演。因此，{title、year}不是键。

{year, starName}也不是键，因为在同一年中一个影星可以演两部电影。因此

year starName → title

不是FD。同样，{title, starName}也不是键，理由是在不同的年份中可有两部同名且由同一个影星演出的电影[⊖]。 □ 85

有时一个关系可能会有多个键。如果是这样的话，通常就要指定其中一个为主键（primary key）。在商业DBMS中，对主键的选择会影响数据库的实现，例如怎样在磁盘中存储关系。一条通用的惯例是：

- 当显示关系模式时，在主键属性下划线。

键的最小化

虽然在关系模型中要求键最小化，但是在E/R模型中没有此项需求。可以假设E/R模型的设计者只把必需的属性作为键，但是没法知道所给的E/R键是否已最小化。只有使用一种如FD的正式的描述后，才能了解所给的属性集合是否为键的最小化集合。

再补充一点，最小化（minimal）与最小值（minimum）是不同的概念，最小化是说不能从此集合中拿出任何东西，而最小值是说它是所有可能键当中最小的。对于一个给定的关系，最小化键的属性个数不一定是键的最小属性数。比如，*ABC*与*DE*均是最小化的键，然而只有*DE*是所有键中属性数最小的。

⊖ 在早期的书中，由于认为没有关于这种情况的例子，一些读者指出那样的认定不正确。影星可在同一部片子的不同版本中演出，这是一个有趣的挑战。

3.4.3 超键

一个包含键的属性集就叫做超键（superkey），它是“键的超集”的简写。因此，每个键都是超键。然而，某些超键不是（最小化的）键。注意，每个超键都满足键的第一个条件：它函数决定了关系中所有其他属性。但超键不需要满足第二个条件：最小化。

例3.14 在例3.13给出的关系中，有许多超键。除了键{title, year, starName}是超键外，还有任何含有键的超集，如

{title, year, starName, length, studioName}

86 也是超键。 □

函数依赖中的“函数”是什么意思？

$A_1 A_2 \ldots A_n \rightarrow B$被称为“函数”依赖是因为在这条规则中，有一个含有一系列值的函数，对每一个属性A_1，A_2，...，A_n都产生一个惟一的B值（或根本没有值）的函数。例如，在关系Movies中，可以想像有一个含有字符串类型（如“*Star Wars*”）和整数类型（如1977）的函数，它确定了一个惟一的length的值，即124。这个函数与数学中给出的函数不同，因为数学中的函数是可计算的，然而这条规则无法进行计算。也就是，不能由字符串“*Star Wars*”和整数1977算出正确的length值，它只能通过对关系的观察才能得出结果。我们观察一个元组，看在给定title和year属性值时，这个元组有什么样的length值。

3.4.4 找出关系中的键

当把E/R设计转化为关系模式时，通常要预知关系的键。产生键的第一条规则是：

- 如果这个关系来自于一个实体集，则它的键就是相应实体集的键属性。

例3.15 例3.1描述了怎样把实体集Movies和Stars转化为关系。这些实体集的键分别为{title, year}和{name}。因此，这也是它们相应的关系的键。下面给出它们的关系模式，下划线强调了关系的键。

Movies(<u>title</u>, <u>year</u>, length, filmType)
Stars(<u>name</u>, address)

□

第二条规则是与二元联系相关。若一个关系R是从联系转化而来，那么联系的多样性就会影响关系的键。有三种情况：

- 如果联系是多对多的，则与其相连的实体集的键属性都是R的键属性。
87 - 如果联系是从E_1到E_2的多对一联系，则只有E_1的键属性是R的键属性。
- 如果联系是一对一的，则与其相连的任一个实体集的键属性都是它的键属性。这也就是说，R的键不惟一。

其他的键术语

在某些书籍和文献中，对键有不同的叫法。将本书中的“超键”称为“键”，也就是键的属性集合只有函数决定所有其他属性的要求，没有最小化限制。而将最小化的键集合，也就是本书中的“键”称为“候选键”。

例3.16 例3.2中给出的Owns是一个从实体集Movies到Studios的多对一的联系，那么Owns

的键属性就是Movies的键属性title和year。下面给出了Owns的模式，用下划线是为了强调它的键属性。

Owns(title, year, studioName)

而例3.3中给出的Stars-in是一个处于Movies和Stars之间的多对多的联系，它的模式是：

Stars-in(title, year, starName)

从这可看出，它的所有属性都是键属性。事实上，除非这种联系有自身的属性，否则它的所有的属性都是键属性。自身的属性不是键属性。 □

最后讨论一下多路联系。因为不可能通过联系的箭头所指对所有可能存在的函数依赖进行描述，所以存在着这样的情形：此时键并不那么显明，如果不经过深思熟虑，不可能知道实体集的哪些集合能函数决定其他实体集。但可以确定的是：

- 如果一个多路联系R有一个箭头指向实体集E，那么至少有一个相应不包含E的键的关系的键。

函数依赖要注意的其他方面

现在对于FD的了解只是FD的左边可有多个属性，其右边只有一个属性。而且，位于右边的属性不能出现在左边。可以对多个左边相同的FD进行简写，简写后的FD右边是一个属性集合。另外，有时允许右边属性是其左边属性的一部分，这样的FD被称为“平凡”（trivial）FD。

另一种观点是FD左边和右边可以是任意属性组合，属性可以同时在左边和右边出现。这两种观点没有什么重要的不同。本书中除了特别声明外，不允许属性同时出现在左右两边。

3.4.5 习题

习题3.4.1 考虑一个关于美国公民信息的关系，这个关系的属性有：人名、社会保险号、街道地址、城市、州、邮编、地区代码和电话号码（7位数字）。这个关系有哪些FD？关系的键是什么？为了回答这些问题，就要知道是如何分配这些数据的。比如，一个地区代码是否可以用于两个州？一个邮编能否适用于两个地区？两个人可否有相同的社会保险号？他们能有相同的地址和电话号码吗？ 88

* **习题3.4.2** 考虑在一个密封容器中分子的方位，属性有分子的ID，分子方位的x、y、z坐标，以及在x、y、z方向上的速率。你认为这个关系上有哪些FD成立？键是什么？

! **习题3.4.3** 习题2.2.5中给出了三种不同的关于联系Births的假设。请分别指出它们相应关系的键。

* **习题3.4.4** 指出由习题3.2.1得出的数据库模式中每个关系的键。

习题3.4.5 指出由习题3.2.4的四个小题所得关系的键。

!! **习题3.4.6** 假设R是含有属性A_1，A_2，…，A_n的关系。如果给出下列条件，指出R有多少超键。

* a) A_1是仅有的键。

b) A_1和A_2均为键。

c) $\{A_1, A_2\}$和$\{A_3, A_4\}$都为键。

d) $\{A_1, A_2\}$和$\{A_1, A_3\}$都为键。 89

3.5 函数依赖的规则

在这一节将要学习如何推导（reason）FD。也就是，假设已经知道关系满足一些FD集合，通常可从这些已知FD中推导出这个关系中必定存在的其他FD。发现其他FD的能力，对于3.6节中讨论的设计一个良好的关系模式很有必要。

例3.17 如果一个含有属性A、B、C的关系R满足FD：$A \rightarrow B$和$B \rightarrow C$，那么就可以推断出R也满足FD：$A \rightarrow C$。这是怎么得到的呢？为了证明$A \rightarrow C$，必须要对R中A的分量值相同的两个元组进行考虑，证明它们C的分量值也相同。

假设两个在A上取值相同的元组（a，b_1，c_1）和（a，b_2，c_2）。假定元组属性的次序是A，B，C。因为R满足$A \rightarrow B$，又已知两个元组在A上的值相同，所以它们在B上的值也相同。也就是$b_1 = b_2$，这两个元组实际上就是（a，b，c_1）和（a，b，c_1），其中b既是b_1也是b_2。同样，因为R满足$B \rightarrow C$，而这两个元组在B上的值相同，所以它们在C上的值也相同。这也就证明了R中只要两个元组在A上取值相同，则它们在C上取值也相同，即存在FD：$A \rightarrow C$。 □

FD在不改变关系的合法实例集的前提下，有多种不同的描述方法。其中：

- 对于FD集合S和T而言，若满足S的关系实例集与其满足T的关系实例集完全相同，就认为S和T等价（equivalent）。
- 更普遍的情况是，若满足T中的所有FD的关系实例集必然同时满足S中的所有FD，则认为S是从T中推断（follow）而来。

注意，当且仅当S从T中推断而来，并且T也从S中推断而来时，S与T才等价。

在这一节中将给出关于FD的很多有用的规则。这些规则保证了可以用一个FD集合替换另一个等价的FD集合，或者可以添加从原有FD集合推断出的新的FD集合。例如，例3.17中给出的传递规则（transitive rule），可以用来跟踪FD链。还可以给出一个用来判断一个FD是否可以由一个或多个FD推断出来的算法。

3.5.1 分解/结合规则

3.1.4节中对FD的定义是：

$$A_1 A_2 \ldots A_n \rightarrow B_1$$
$$A_1 A_2 \ldots A_n \rightarrow B_2$$
$$\ldots$$
$$A_1 A_2 \ldots A_n \rightarrow B_m$$

它的缩略形式是：

90

$$A_1 A_2 \ldots A_n \rightarrow B_1 B_2 \ldots B_m$$

所以，可把缩略形式右边的属性分解开，使得每个FD的右边只有一个属性。同样，也可以把左边具有相同属性的多个FD组合起来，形成一个左边相同而右边为原来右边所有属性集合的FD。此时FD的新形式与原形式等价。等价的转化方式有两种：

- 可用一个FD的集合$A_1 A_2 \ldots A_n \rightarrow B_i$（$i = 1$，$2$，…，$m$）替换FD $A_1 A_2 \ldots A_n \rightarrow B_1 B_2 \ldots B_m$。这种转化称为分解规则（splitting rule）。
- 可用一个FD $A_1 A_2 \ldots A_n \rightarrow B_1 B_2 \ldots B_m$替换FD集合$A_1 A_2 \ldots A_n \rightarrow B_i$（$i = 1$，$2$，…，$m$）。这种转化称为组合规则（combining rule）。

例如，在例3.12中指出了FD集合：

```
title year → length
title year → filmType
title year → studioName
```

如何等价于单个FD

```
title year → length filmType studioName
```

分解规则只能在FD的右边使用，而不能在左边使用。可用下例来说明原因。

例3.18 考虑例3.12中关系Movies的一个FD：

```
title year → length
```

如果要把它进行左边分解，则为

```
title → length
year → length
```

那么就得到了两个错误的FD。也就是说，title不能函数决定length，原因是可以存在两部同名（例如，*King Kong*）但片长不同的电影。同样，year也不能函数决定length，是因为在任一年代可以存在不同片长的电影。 □ 91

3.5.2 平凡函数依赖

若在一个FD：$A_1 A_2 \ldots A_n \rightarrow B$中，$B$属于$A$，则认为这个FD是平凡的（trivial）。比如，

```
title year → title
```

就是这样一个FD。

每个关系中都会存在平凡 FD，因为平凡FD是说“两个元组在属性A_1，A_2，...，A_n上取值相同，则它们在这n个属性中的任一个上取值都相同。”因此，不需知道关系中的FD就可以假设出任意一个平凡FD。

在FD的最初定义中不允许存在平凡FD。然而，包括它们没有坏处，因为它们总是真的，而且有时它们能使规则的描述简单化。

当允许存在平凡 FD时，也就允许存在左边属性出现在右边的FD。而FD $A_1 A_2 \ldots A_n \rightarrow B_1 B_2 \ldots B_m$为称为：

- 平凡的（trivial），仅当其右边的属性集合是左边集合的子集。
- 非平凡的（nontrivial），仅当其右边属性集中至少有一个属性不属于左边的集合。
- 完全非平凡的（completely nontrivial），仅当其右边集合中的属性均不在左边集合中。

因此

```
title year → year length
```

是非平凡的，而不是完全非平凡的。若除去右边中的year，得到的就是一个完全非平凡的FD。

还有一种情况是，人们总是可以从右边除去那些在左边出现的属性，也就是：

- FD $A_1 A_2 \ldots A_n \rightarrow B_1 B_2 \ldots B_m$等价于

  $A_1 A_2 \ldots A_n \rightarrow C_1 C_2 \ldots C_k$

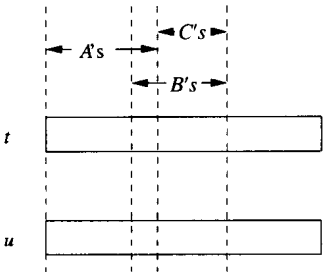
这里的C是所有不在A中而在B中的属性。

这个规则被称为平凡依赖规则（trivial-dependency rule），用图3-18给以说明。

3.5.3 计算属性的闭包

在讲述其他规则前，先介绍一个基本的规则，其他规则都是从它引伸出来的。假设{A_1，

A_2，…，A_n}是一个属性集合，S是一个FD的集合。则S集合下的属性集合{A_1，A_2，…，A_n}的闭包（closure）是集合B，使得每一个满足S中所有FD的关系，也同样满足A_1，A_2，…，$A_n \rightarrow B$。也就是说，$A_1 A_2 \ldots A_n \rightarrow B$是从$S$的FD中推
92 断出来的。属性集{A_1，A_2，…，A_n}的闭包记为{A_1，A_2，…，A_n}$^+$。为了简化闭包计算的讨论，将允许存在平凡FD，于是A_1，A_2，…，A_n总是在{A_1，A_2，…，A_n}$^+$中。



如果t和u在A部分一致
则它们一定也在B部分一致
因此它们在C部分一致

图3-18 平凡依赖规则

图3-19给出了计算闭包的过程。从一个给定的属性集合出发，重复地扩展这个集合，只要某个FD左边的属性全部包含在这个集合中，就把此FD的右边的属性也包含进去，依次使用这个方法，直到不再产生新的属性为止。最后的结果集合就是给定属性集合的闭包。下面给出计算属性集合{A_1，A_2，…，A_n}关于某已知FD集合的闭包的详细算法。

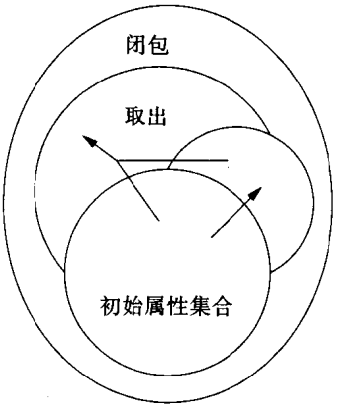


图3-19 计算属性集合的闭包

1. 设X是结果的属性集合，也就是闭包。首先，把X初始化为{A_1，A_2，…，A_n}。

2. 在FD集合中查找B_1 B_2 … $B_n \rightarrow C$这样的式子，这里B_1，B_2，…，B_n在X中，而C不在X中，若找到，则把C加入X。

3. 反复使用第二步，直到不再有其他的属性加入到X。因为X的元素只能增长，而任何一个关系模式中的属性都是有限的，所以最后肯定存在不能再加入属性的结果。

4. 当不能添加任何属性时，集合X就是{A_1，A_2，…，A_n}$^+$。

例3.19 考虑含有属性A，B，C，D，E和F的关系。假设此关系有FD：$AB \rightarrow C$，$BC \rightarrow$
93 AD，$D \rightarrow E$和$CF \rightarrow B$。那么{A，B}的闭包{A，B}$^+$是什么？

从X = {A，B}出发。首先注意到FD $AB \rightarrow C$左边的属性都在X中，而C不在X中，把C加入X。因此，第二步运行一次后的X为{A，B，C}。

接着，注意到FD $BC \rightarrow AD$左边的属性都在X中，所以可往X中添加A和D[⊖]。但由于A在X中，而D不在，由此X为{A，B，C，D}。同样，根据FD $D \rightarrow E$可把E加入X中，于是X为{A，B，C，D，E}。至此，再没有属性可以添加到X中了。要注意不能使用FD $CF \rightarrow B$，原因是左边集合中的F永远不会在X中。因此，{A，B}$^+$ = {A，B，C，D，E}。 □

如果知道怎样计算任一属性集合的闭包，那么就可以判断任一给定的FD $A_1 A_2 \ldots A_n \rightarrow B$是否是来自FD集合$S$的推断。首先可用$S$计算{$A_1$，$A_2$，…，$A_n$}$^+$。如果$B$在{$A_1$，$A_2$，…，$A_n$}$^+$中，则$A_1 A_2 \ldots A_n \rightarrow B$可从$S$推断得来。如果$B$不在{$A_1$，$A_2$，…，$A_n$}$^+$中，则该FD不能从$S$推断得来。更普遍地，如果一个右边为属性集合的FD是FD集合的缩略形式，将其分解就可以判定这个FD

⊖ $BC \rightarrow AD$是函数依赖集$BC \rightarrow A$和$BC \rightarrow D$的缩略形式，根据需要可以独立处理这些依赖集。

是否从S推断出来。比如对于FD $A_1\ A_2 \ldots A_n \rightarrow B_1\ B_2 \ldots B_m$来说，只有当$B_1$，$B_2$，…$B_m$都在$\{A_1, A_2, \ldots, A_n\}^+$中时，才认为此FD是从$S$推断而来。

例3.20 考虑例3.19中的关系和FD集合。假设要证明$AB \rightarrow D$是从这些FD集合中推出来的。先计算$\{A, B\}^+$，由上例可知，它的值为$\{A, B, C, D, E\}$。因为D是闭包的一个元素，则可以认为$AB \rightarrow D$是从FD集合得来的。

另一方面，考虑FD $D \rightarrow A$。为了判断这个FD是否能从给定的FD集合得来，先要计算$\{D\}^+$。为了计算这个闭包，令$X = \{D\}$。接着可以使用FD $D \rightarrow E$把E加入X中。此时，由于没有其他 94
FD的左边属性包含$X = \{D, E\}$，所以$\{D\}^+ = \{D, E\}$。因为A不在$\{D, E\}$中，结论是FD $D \rightarrow A$不能从FD集合得来。 □

3.5.4 为什么能用闭包算法

在这节中，将证明为什么闭包算法能正确判断一个FD $A_1\ A_2 \ldots A_n \rightarrow B$是否能从给定的FD集合$S$推断出来。证明分两部分：

1. 必须要证明闭包算法没有多余的FD。也就是说，如果$A_1\ A_2 \ldots A_n \rightarrow B$通过了闭包算法测试（即$B$在$\{A_1, A_2, \ldots, A_n\}^+$中），那么$A_1\ A_2 \ldots A_n \rightarrow B$在任何满足FD集合$S$的关系中成立。

2. 必须要证明通过闭包算法可以找到从S推断出来的所有FD。

为什么闭包算法只给出正确的FD

这一点可以通过对算法第二步的使用次数作归纳证明。第二步是说对每一个X中的属性D，FD $A_1\ A_2 \ldots A_n \rightarrow D$成立（特殊情况，$D$是$A$中一员。该FD为平凡FD）。这样，每个满足$S$中所有FD的关系$R$都满足$A_1\ A_2 \ldots A_n \rightarrow D$。

基础：最基础的情况是没有进行任何计算。于是D必定是A_1，A_2，…，A_n中一员，则$A_1\ A_2 \ldots A_n \rightarrow D$是个平凡FD，它在任何关系中都成立。

归纳：假设当使用FD $B_1\ B_2 \ldots B_m \rightarrow D$时已把$D$加入到$X$。那么由归纳假设可知$R$满足$A_1\ A_2 \ldots A_n \rightarrow B_i$（$i = 1, 2, \ldots, m$）。换言之，$R$的任何两个元组在$A_1$，$A_2$，…，$A_n$上的取值相等时，它们在$B_1$，$B_2$，…，$B_m$上的取值也相等。由于$R$满足$B_1\ B_2 \ldots B_m \rightarrow D$，还可以得出这两个元组在$D$上的取值也相等。因此，$R$满足$A_1\ A_2 \ldots A_n \rightarrow D$。

为什么闭包算法可以找到所有的FD

假设闭包算法测试说FD $A_1\ A_2 \ldots A_n \rightarrow B$不能从$S$中推断。也就是说，$\{A_1, A_2, \ldots, A_n\}$关于$S$的闭包中不包含$B$。那么，如果要证明FD $A_1\ A_2 \ldots A_n \rightarrow B$确实不能从$S$中推断，也就是要证明至少有一个满足$S$中所有FD集合，但不满足$A_1\ A_2 \ldots A_n \rightarrow B$的关系实例存在。

构造这样一个实例I非常简单，步骤见图3-20。假设I只有两个元组t和s。这两个元组在$\{A_1, A_2, \ldots, A_n\}^+$的所有属性上取值相同，但是在其他属性上的值不同。首先要证明I满足S中所有的FD，接着证明它不满足$A_1\ A_2 \ldots A_n \rightarrow B$。 95

	$\{A_1, A_2, \ldots, A_n\}^+$	Other Attributes
t:	1 1 1 ··· 1 1	0 0 0 ··· 0 0
s:	1 1 1 ··· 1 1	1 1 1 ··· 1 1

图3-20 满足S但不满足$A_1\ A_2 \ldots A_n \rightarrow B$的实例$I$

假设在集合S中存在一些I不满足的FD $C_1\ C_2 \ldots C_k \rightarrow D$。因为$I$只有两个元组$t$和$s$，那么肯定是这两个元组违反了$C_1\ C_2 \ldots C_k \rightarrow D$。也就是说，$t$和$s$在$\{C_1, C_2, \ldots C_k\}$上的取值相同，但在$D$上的取值不同。从图3-20中可以看出，$C_1$，$C_2$，…$C_n$肯定属于$\{A_1, A_2, \ldots, A_n\}^+$，这是因为$t$和$s$只在这个闭包中的取值相同。同样，因为$t$和$s$只在其他属性上的取值不同，$D$肯定属于其

他属性。

但是这样就不能正确地计算出闭包。因为当X为$\{C_1, C_2, \ldots C_n\}$时，就可以运用$C_1\ C_2 \ldots C_k \rightarrow D$把$D$加入$X$。由此得出的结论是，不存在$C_1\ C_2 \ldots C_k \rightarrow D$；即实例$I$满足$s$。

其次，要证明I不满足$A_1\ A_2 \ldots A_n \rightarrow B$。这个比较简单，已知$A_1, A_2, \ldots, A_n$属于$t$和$s$取值相同的属性集合，并且$B$不在$\{A_1, A_2, \ldots, A_n\}^+$中（即$B$不属于$\{A_1, A_2, \ldots, A_n\}^+$）。由此可知，$I$不满足$A_1\ A_2 \ldots A_n \rightarrow B$。上述分析的结论就是，闭包算法不会找到过多或过少的FD，而是不多不少的能从s推断的所有FD。

3.5.5 传递规则

传递规则联结了两个FD

- 若关系R中FD $A_1\ A_2 \ldots A_n \rightarrow B_1\ B_2 \ldots B_m$和$B_1\ B_2 \ldots B_m \rightarrow C_1\ C_2 \ldots C_k$都成立，那么FD $A_1\ A_2 \ldots A_n \rightarrow C_1\ C_2 \ldots C_k$也在$R$中也成立。

如果C中有属性属于A集合，则可根据平凡依赖把它们从右边消除。

下面用3.5.3节中的测试来证明传递规则的正确性。为了证明$A_1\ A_2 \ldots A_n \rightarrow C_1\ C_2 \ldots C_k$成立，就要根据所给的两个FD来计算$\{A_1, A_2, \ldots, A_n\}^+$。

从FD $A_1\ A_2 \ldots A_n \rightarrow B_1\ B_2 \ldots B_m$可知，$B_1, B_2, \ldots, B_m$属于$\{A_1, A_2, \ldots, A_n\}^+$。然后，使用FD $B_1\ B_2 \ldots B_m \rightarrow C_1\ C_2 \ldots C_k$把$C_1, C_2, \ldots, C_k$加入到$\{A_1, A_2, \ldots, A_n\}^+$。因为$C$集合中所有的元素都属于$\{A_1, A_2, \ldots, A_n\}^+$，所以可以得出结论：对于任何满足$A_1\ A_2 \ldots A_n \rightarrow B_1\ B_2 \ldots B_m$和$B_1\ B_2 \ldots B_m \rightarrow C_1\ C_2 \ldots C_k$的关系而言，$A_1\ A_2 \ldots A_n \rightarrow C_1\ C_2 \ldots C_k$成立。

96

闭包和键

注意当且仅当$A_1, A_2, \ldots, A_n$是关系的超键时，$\{A_1, A_2, \ldots, A_n\}^+$才是这个关系的所有属性的集合。只有这样，$A_1, A_2, \ldots, A_n$才能函数决定所有其他的属性。如果要验证$\{A_1, A_2, \ldots, A_n\}$是一个关系的键，可以先检查$\{A_1, A_2, \ldots, A_n\}^+$是否包含了关系的全部属性，然后再检查是否是从$\{A_1, A_2, \ldots, A_n\}$中移走一个属性，就不会形成包含所有属性。

例3.21 先从图3-7中关系Movies开始，该关系来自于例3.5，表示出实体集Movies的四个属性以及Movies与Studios之间的联系Owns。下面给出这个关系和它的一些数据样本。

title	*year*	*length*	*filmType*	*studioName*
Star Wars	1977	124	color	Fox
Mighty Ducks	1991	104	color	Disney
Wayne's World	1992	95	color	Paramount

假设要在此关系中加入描述制片厂的相关数据，为了简单起见，仅仅加上制片厂所在的城市以表示它的地址。于是这个关系变为：

title	*year*	*length*	*filmType*	*studioName*	*studioAddr*
Star Wars	1977	124	color	Fox	Hollywood
Mighty Ducks	1991	104	color	Disney	Buena Vista
Wayne's World	1992	95	color	Paramount	Hollywood

在这个关系中存在有两个FD：

```
title year → studioName
studioName → studioAddr
```

第一个FD成立是因为Owns是多对一的联系。而第二个FD成立是因为address是Studios

的一个属性，并且studioName是Studios的键。

运用传递规则，上面两个FD可组合为

```
title year → studioAddr
```

这个FD说明了title和year（也就是一部movie）函数决定了一个地址——拥有这部电影的制片厂的地址。 □ 97

3.5.6 函数依赖的闭包集合

由上面的分析可知，只要给定一个FD集合，就可以推断出一些其他FD集合，其中包含平凡和非平凡FD集合。在以后的章节中，还将介绍怎样区分关系中原已给定的FD（given FD）集合和运用这节中给出的规则或属性集合的闭包算法导出的FD（derived FD）。

而且，在有些情况下，还要判断应使用哪一个FD集合来描述一个关系的完全FD集合。如果一个给定FD集能把关系中的其他FD集合推出来，就认为这个FD是关系的基本集（basis）。如果一个基本集的任和真子集都不能完全地把其他FD集推出来，就称这个基本集为最小化的（minimal）基本FD集。

例3.22 考虑关系（A，B，C），它的任一个属性都能函数决定其他两个属性。此时它的全部导出FD集包含了六个左边和右边都只有一个属性的FD：$A\rightarrow B$、$A\rightarrow C$、$B\rightarrow A$、$B\rightarrow C$、$C\rightarrow A$、和$C\rightarrow B$，以及三个左边有两个属性的FD：$AB\rightarrow C$、$AC\rightarrow B$、$BC\rightarrow A$。导出FD集中还包括有用缩略形式表示的FD：$A\rightarrow BC$，类似$A\rightarrow A$的平凡FD和类似$AB\rightarrow AC$的非完全平凡FD（虽然在严格的FD定义中，不要求列出平凡的或部分平凡的FD，和右边有多个属性的依赖）。

这个关系和它的FD集合有多个最小化的基本FD集。其中一个是

$$\{A\rightarrow B,\ B\rightarrow A,\ B\rightarrow C,\ C\rightarrow B\}$$

另一个是

$$\{A\rightarrow B,\ B\rightarrow C,\ C\rightarrow A\}$$

这个关系还有一些其他最小化的基本FD集，把它作为习题留给读者。 □

3.5.7 投影函数依赖

当学习关系模式设计时，还应回答下面有关FD的问题。假设有一个含有FD集合F的关系R，通过除去R模式中某些属性获得R的“投影”（project）。设S是从关系R的投影中获得的关系。因为S是集合，在它的实例中没有相同的元组。那么S中有哪些FD呢？

这个问题的答案原则上可以通过计算所有满足如下条件的FD集合得来：

- 从F推断出来。
- 只包含S的属性。 98

推理规则的全集

若要知道一个FD是否从一个给定的FD集合中导出，经常使用的方法是运用3.5.3节中介绍的闭包算法。可是，还可运用被称为Armstrong 公理（Armstrong's axioms）的一组规则得到一个给定集合能推断出的FD。这些公理是：

1. 自反律（reflexivity）：如果$\{B_1, B_2, \ldots, B_m\}$是$\{A_1, A_2, \ldots, A_n\}$的子集，则$A_1 A_2 \ldots A_n \rightarrow B_1 B_2 \ldots B_m$。这就是通常所说的平凡FD。
2. 增广律（augmentation）：如果$A_1 A_2 \ldots A_n \rightarrow B_1 B_2 \ldots B_m$，那么

$A_1 A_2 \ldots A_n\ C_1 C_2 \ldots C_k \rightarrow B_1 B_2 \ldots B_m C_1 C_2 \ldots C_k$

对于任何属性C_1，C_2，...，C_k的集合都成立。

3. 传递律（transitivity）：如果

$A_1 A_2 \ldots A_n \rightarrow B_1 B_2 \ldots B_m$和$B_1 B_2 \ldots B_m \rightarrow C_1 C_2 \ldots C_k$

都成立，那么

$A_1 A_2 \ldots A_n \rightarrow C_1 C_2 \ldots C_k$

也成立。

由于存在大量这样计算出的FD集合，而且其中很多可能是冗余（也就是，它们是从相同的FD推出），因此可以对它们进行简化。通常，在最坏情况下，计算S中的全部FD集合的复杂度是S中属性数目的指数幂。

例3.23 假设R（A，B，C，D）中有FD：$A \rightarrow B$，$B \rightarrow C$和$C \rightarrow D$。假设要把R投影到S（A，C，D）。原则上，为了找到S的FD集合，需要运用FD集合的完全集，包括涉及B的FD，求出{A，C，D}的八个子集的闭包。但实际上可以使用一些明显的简化规则。

- 除去空集和不能推出非平凡FD的属性集合。
- 如果已知集合X的闭包包含了全部的属性，那么就不能通过X的超集来寻找新的FD。

因此，可先从单元素集的闭包出发，如有必要的话再接着从双元素集合的闭包出发。对于
99 X中的任一个闭包，要增加关于在X^+和在S的模式中，但不在X中的属性E的FD $X \rightarrow E$。

首先，$\{A\}^+ = \{A, B, C, D\}$。因此，FD $A \rightarrow C$和$A \rightarrow D$存在于S中。要注意$A \rightarrow B$在R中有效，但在S中毫无意义，这是因为B不是S的属性。

接着，考虑$\{C\}^+ = \{C, D\}$，从这个集合可以为S得到新的FD $C \rightarrow D$。因为$\{D\}^+ = \{D\}$，不能添加新的FD。于是，单元素集闭包计算完成。

由于$\{A\}^+$包含了S的所有属性，因此就没有必要考虑$\{A\}$的超集。原因是不管找到的是什么样的FD，如$AC \rightarrow D$，它是通过增广律从S中已有的，左边只有A的FD推导出来。此时，仅有的双元素集的闭包是$\{C, D\}^+ = \{C, D\}$。它意味着不能再添加任何FD。闭包计算到此为止，所得的FD是：$A \rightarrow C$，$A \rightarrow D$和$C \rightarrow D$。

若仔细观察的话，还可发现$A \rightarrow D$是运用传递律从其他两个FD得到的。因此，S中的一个简单的，等价的FD集合就是$A \rightarrow C$和$C \rightarrow D$。 □

3.5.8 习题

* **习题3.5.1** 考虑模式为R（A，B，C，D）的关系R和FD：$AB \rightarrow C$，$C \rightarrow D$和$D \rightarrow A$。

a) 从给定的FD集合推出的非平凡FD是什么？必须限制FD的右边只有一个属性。

b) R的键有哪些？

c) 不包含R键的超键有哪些？

习题3.5.2 根据下列各条件重复回答习题3.5.1提出的问题：

a) 模式为S（A，B，C，D），FD：$A \rightarrow B$，$B \rightarrow C$和$B \rightarrow D$。

b) 模式为T（A，B，C，D），FD：$AB \rightarrow C$，$BC \rightarrow D$，$CD \rightarrow A$和$AD \rightarrow B$。

c) 模式为U（A，B，C，D），FD：$A \rightarrow B$，$B \rightarrow C$，$C \rightarrow D$和$D \rightarrow A$。

习题3.5.3 运用节3.5.3中的闭包测试方法，证明下面的规则。

* a) 增广左边（augmenting left sides）。如果FD $A_1 A_2 \ldots A_n \rightarrow B$成立，且$C$是另一个属性，
100 那么可推出$A_1 A_2 \ldots A_n C \rightarrow B$。

b) 全部增广（full augmentation）。如果FD $A_1 A_2 \ldots A_n \rightarrow B$成立，且$C$是另一个属性，那么可推出$A_1 A_2 \ldots A_n C \rightarrow BC$。注意，利用这个规则，可以很容易地证明3.5.6节中方框中所提及的增广（augmentation）律。

c) 假传递（pseudotransitivity）。假设FD $A_1 A_2 \ldots A_n \rightarrow B_1 B_2 \ldots B_m$和$C_1 C_2 \ldots C_k \rightarrow D$成立，且$B$中元素都在$C$中。则$A_1 A_2 \ldots A_n E_1 E_2 \ldots E_j \rightarrow D$成立，其中$E$的元素都在$C$中，而没有任何元素在$B$中。

d) 加法（addition）。如果FD $A_1 A_2 \ldots A_n \rightarrow B_1 B_2 \ldots B_m$和$C_1 C_2 \ldots C_k \rightarrow D_1 D_2 \ldots D_j$成立，那么FD $A_1 A_2 \ldots A_n C_1 C_2 \ldots C_k \rightarrow B_1 B_2 \ldots B_m D_1 D_2 \ldots D_j$也成立。但要先确保$A$、$C$合集和$B$、$D$合集中无相同的元素。

! **习题3.5.4** 通过给出关系例子证明下列规则无效，例子要满足给定FD集（跟在“if”后的），但不满足导出FD集（跟在“then”后的）。

* a) if $A \rightarrow B$ then $B \rightarrow A$。

b) if $AB \rightarrow C$ and $A \rightarrow C$, then $B \rightarrow C$。

c) if $AB \rightarrow C$, then $A \rightarrow C$ or $B \rightarrow C$。

! **习题3.5.5** 证明若一个关系不具有由其他所有属性函数决定的属性，那么这个关系根本就没有非平凡FD。

! **习题3.5.6** 令X和Y是属性集合。证明如果X是Y的子集，那么X^+也是Y^+的子集，其中X^+和Y^+分别是X和Y关于同一个FD集的闭包。

! **习题3.5.7** 证明$(X^+)^+ = X^+$。

!! **习题3.5.8** 如果$X^+ = X$，就认为属性集合X封闭（关于一个指定的FD集合）。考虑模式为$R(A, B, C, D)$的关系R和一个未知的FD集合。若能得知哪个属性集合是封闭的，就可以找到FD。根据下列条件，求出FD集合。

* a) 这四个属性的所有集合都是封闭的。

b) 只有Φ和$\{A, B, C, D\}$是封闭的。

c) 封闭集是Φ，$\{A, B\}$和$\{A, B, C, D\}$。

! **习题3.5.9** 找出例3.22中关系的所有最小化的基本FD集。 101

! **习题3.5.10** 假设有一关系$R\{A, B, C, D, E\}$和一些FD，把它投影到$S(A, B, C)$。下面给出R中的FD集合，求出S中的FD集合。

* a) $AB \rightarrow D$E，$C \rightarrow$ E，$D \rightarrow C$和E $\rightarrow A$。

b) $A \rightarrow D$，$BD \rightarrow E$，$AC \rightarrow E$和$DE \rightarrow B$。

c) $AB \rightarrow D$，$AC \rightarrow E$，$BC \rightarrow D$，$D \rightarrow A$和$E \rightarrow B$。

d) $A \rightarrow B$，$B \rightarrow C$，$C \rightarrow D$，$D \rightarrow E$和$E \rightarrow A$。

对于每种情况，给出S的最小化基本FD集合。

!! **习题3.5.11** 证明可使用Armstrong公理证明一个FDF是从给定FD集合中推出的。提示：研究计算属性集合闭包算法，给出算法的每一步怎么与Armstrong公理相对应。

3.6 关系数据库模式设计

选择关系数据库模式不仔细的话会带来问题。例如，例3.6中组合一个多对多联系的关系和一个与此联系相连的实体集的关系，就产生了问题。它的主要问题就是冗余，也就是同一个事实在多个元组中重复。这个问题在图3-17中已见到，现在把它再现在图3-21中。如果出演影

星不同，*Stars Wars*和*Wayne's World*的类型和长度就要重复一次。

这节将解决怎样设计一个好的关系模式的问题，下面是设计步骤：

1. 首先当模式有问题时，要对这个问题进行深入地研究。

2. 其次，引进一种“分解”的思想，也就是把一个关系模式（属性集合）分解为两个较小的模式。

3. 再下一步，引进“Boyce-Codd范式”，即“BCNF”，这是一种在关系模式上消除上述问题的条件。

102 4. 把上面的几点结合起来解释怎样通过分解关系模式来确保BCNF。

title	*year*	*length*	*filmType*	*studioName*	*starName*
Star Wars	1977	124	color	Fox	Carrie Fisher
Star Wars	1977	124	color	Fox	Mark Hamill
Star Wars	1977	124	color	Fox	Harrison Ford
Mighty Ducks	1991	104	color	Disney	Emilio Estevez
Wayne's World	1992	95	color	Paramount	Dana Carvey
Wayne's World	1992	95	color	Paramount	Mike Meyers

图3-21 显示异常的关系Movies

3.6.1 异常

当一个关系中含有过多信息时产生的问题如冗余就叫做异常（anomaly）。异常的基本类型有：

1. 冗余（redundancy）。信息没有必要地在多个元组中重复。如图3-21中Movies关系的length和filmType字段。

2. 更新异常（Update Anomaly）。可能修改了一个元组的信息，但是没有改变其他元组中的相同信息。例如，*Star Wars*的实际放映时间为125分钟，这时可能对图3-21中的第一个元组的length作了改变，但是没有改变第二个和第三个元组的信息。是的，没有人会这么粗心的。但是，如果重新对关系Movies进行设计，那么引起这种错误的危险就不复存在。

3. 删除异常（Deletion Anomaly）。如果一个值集变成空集，就可能丢失信息。例如，假如要从*Mighty Ducks*的影星集合中删除Emilio Estevez，则数据库中就不再存在出演这部电影的影星的记录。在关系Movies中的最后一个*Mighty Ducks*元组就会消失，并且它的其他信息如片长104分钟和类型为color等也会在数据库中消失。

3.6.2 分解关系

一般用分解（decompose）关系的方法来消除异常。分解R不但要分离R的属性，使其组成两个新的关系，还要对R进行投影以转移元组的信息到这两个关系中。描述完分解过程后，将介绍怎样分解才能消除异常。

给定一个模式为$\{A_1, A_2, \ldots, A_n\}$的关系$R$，要把它分解为关系$S$和$T$，它们的模式分别为

103 $\{B_1, B_2, \ldots, B_m\}$和$\{C_1, C_2, \ldots, C_k\}$，并且满足：

1. $\{A_1, A_2, \ldots, A_n\} = \{B_1, B_2, \ldots, B_m\} \cup \{C_1, C_2, \ldots, C_k\}$。

2. S中的元组是R中所有元组在$\{B_1, B_2, \ldots, B_m\}$上的投影（projection）。也就是，对于$R$的当前实例的任一元组$t$，取出$t$在属性$B_1, B_2, \ldots, B_m$上的相应分量。这些分量就组成了一个属于$S$当前实例的元组。由于$S$为集合，从$R$中两个不同的元组投影到$S$的元组有可能是重复的，这时就只需保留一份。

3. 同样，T中的元组也是R的当前实例的所有元组在属性集合$\{C_1, C_2, \ldots, C_k\}$上的投影。

例3.24 分解图3-21中的关系Movies。首先要对模式进行分解。怎么分解更有好处在

3.6.3节有介绍，在这里Movies分解为：

1. 关系Movies1，它的模式包含了除starName外的其他属性。

2. 关系Movies2，它的模式包含了属性title，year和starName。

下面将给出分解图3-21中关系实例的过程。第一步是投影到模式Movies1：

{title, year, length, filmType, studioName}

图3-21中的前三个元组在这五个属性上取的值都相同，为

(Star Wars, 1977, 124, color, Fox)

第四个元组产生了在这五个属性上取值均不同的元组。而第五个和第六个元组也产生了在这几个属性上取值相同的元组。投影后的结果Movies1见图3-22。

title	*year*	*length*	*filmType*	*studioName*
Star Wars	1977	124	color	Fox
Mighty Ducks	1991	104	color	Disney
Wayne's World	1992	95	color	Paramount

图3-22 关系Movies1

下一步是投影到模式Movies2。这六个元组中每一个在三个属性title、year 和starName上的取值至少有一个与其他五个不同，因此投影结果如图3-23所示。 □ 104

title	*year*	*starName*
Star Wars	1977	Carrie Fisher
Star Wars	1977	Mark Hamill
Star Wars	1977	Harrison Ford
Mighty Ducks	1991	Emilio Estevez
Wayne's World	1992	Dana Carvey
Wayne's World	1992	Mike Meyers

图3-23 关系Movies2

接着分析在这个例子中怎样用分解来消除3.6.1节中所讲的异常。在这里，冗余消除了。例如，关系Movies1中每部电影的片长只出现一次。更新异常消除了。因为在这里只要对Movies1中的一个元组*Star Wars*的*length*进行改变，不会产生同一部电影有不同片长的情况。

最后，删除异常消除了。比如，删除所有*Mighty Ducks*中的影星，将导致这部电影从Movies2中消除，但是关于这部电影的其他信息仍可以从Movies1中得到。

因为一部电影的片名和年份可重复多次，在Movies2中仍存在冗余。但是这两个属性是Movies的键，没有更简洁的方法可以描述一部电影了。此外，Movies2根本不会出现更新异常。例如，可能会有人认为若把starName为Carrie Fisher的年份改为2003，而不对*Star Wars*的其他两个元组进行改变，那么就会引起更新异常。然而有可能存在这样一部名为*Star Wars*，影星为Carrie Fisher而年份为2003年的电影。因此，不能阻止在*Star Wars*的某个元组中改变year，也不能保证这种改变一定正确。

3.6.3 Boyce-Codd范式

分解的目的就是将一个关系用多个不会产生异常的关系替换。也就是说，在一个简单的条件下保证前面讨论的异常不产生。这个条件称为Boyce-Codd范式，或称为BCNF。

- 关系R满足BCNF当且仅当：如果R中非平凡FD $A_1 A_2 \ldots A_n \rightarrow B$成立，则$\{A_1, A_2, \ldots, A_n\}$是关系$R$的超键。

换言之，每个非平凡FD的左边都必须是超键。由于超键不一定是最小化，因此，BCNF的

105 一个等价描述是，每个非平凡FD的左边必须包含键。

当发现一个FD违反BCNF时，一个更简单的将关系完全分解到BCNF关系的方法是，只要在这个FD右端添加与其左端相同的所有其他FD的右端，而不管那些FD是否违反BCNF，就能使其对应的关系模式满足BCNF。下面是BCNF的另一种定义，这里人们寻找具有相同左边的FD集合，其中至少有一个是非平凡FD，并且不满足BCNF：

- 关系R满足BCNF当且仅当：如果R中非平凡FD $A_1\ A_2 \ldots A_n \rightarrow B_1\ B_2 \ldots B_m$成立，则{ A_1, A_2, ..., A_n }是关系R的超键。

这个定义与前一个定义等价。由于FD $A_1\ A_2 \ldots A_n \rightarrow B_1\ B_2 \ldots B_m$是FD集合$A_1\ A_2 \ldots A_n \rightarrow B_i$（$i$=1, 2, ..., m）的缩略形式，因此至少有一个B_i不在A中（否则，$A_1\ A_2 \ldots A_n \rightarrow B_1\ B_2 \ldots B_m$是平凡的）。而根据最初的定义，$A_1\ A_2 \ldots A_n \rightarrow B_i$是BCNF侵犯。

例3.25 图3-21的关系Movies不属于BCNF，下面将给以证明。首先要确定键，例3.13中已求出{title, year, starName}是键。因此，任何包含这三个属性的集合就是超键。从例3.13可以得出任何不包含这三个属性的集合都不是超键的结论。因此，{title, year, starName}是Movies的惟一键。

可是，根据例3.13可知Movies中存在FD

title year → length filmType studioName

对这个FD进行考虑，可看出它的左边不是超键。特别是，title和year不能函数决定属性starName，因此这个FD违反了BCNF条件，说明Movies的模式不是BCNF。而且，根据BCNF的原始定义（FD的右边是单个的属性），可得出这三个FD的任何一个，如title year →length，违反了BCNF。 □

例3.26 另一方面，图3-22中的Movies1模式是BCNF。因为该关系中存在FD

title year → length filmType studioName

并且已知Movies1的惟一键是{title, year}。另外，仅有的非平凡FD左边至少要有title和year，因此这个非平凡FD的左边一定是超键。这也就是说，Movies1模式满足
106 BCNF。 □

例3.27 任一个二元关系属于BCNF。对此必须审查所有可能的右边是单个属性的非平凡FD。由于没有太多的情况可以讨论，下面将依次列出这些情况。假设属性为A和B。

1. 没有非平凡FD。于是这个关系肯定满足BCNF。因为只有非平凡FD才能违反BCNF。在这种情况下{ A, B }是惟一的键。

2. $A \rightarrow B$成立，但$B \rightarrow A$不成立。在这种情况下，A是惟一的键，每个非平凡FD的左边都包含A（事实上，左边只能是A）。此时没有FD违反BCNF。

3. $B \rightarrow A$成立，但$A \rightarrow B$不成立。这种情况与第二种情况类似，只不过是A，B对调。

4. $A \rightarrow B$和$B \rightarrow A$都成立。于是A和B都是键。任一FD的左边至少包含A和B的其中一个。此时，没有FD违反BCNF。

值得注意的是，对于第四种情况可能会有多个键。此外，BCNF条件要求的是任一个非平凡FD的左边只需要有键则可，不一定是全部的键。对于只有两个属性的关系，每个都函数决定另一个的情形并不难以置信。例如，一个公司会分配给它的员工惟一的ID，并且记录他们的社会保险号。一个只有empID和ssNO的关系中的每个属性都函数决定其他的属性。换言之，就是每个属性都是键，因此也就没有两个元组在某个属性上的值相同。 □

3.6.4 分解为BCNF

重复使用适当的分解，可以把任何一个关系模式分解为具有下列重要性质的多个真子集：

- 这些真子集都是满足BCNF的关系模式。
- 原始关系中的数据都正确地反映在分解后的关系上，对此3.6.5节的说法更精确。简单讲，是要求原始关系实例应能从分解后的几个关系实例中重构。

从例3.27可看出一个关系被分解为多个二元关系后必然满足BCNF。但是这样武断的分解无法满足上述第二个性质，3.6.5节会给出关于这一点的说明。所以分解关系模式时必须非常小心，事实上可利用违反BCNF的条件来进行分解。 107

下面给出分解的步骤。首先寻找违反BCNF条件的非平凡FD $A_1 A_2 \ldots A_n \rightarrow B_1 B_2 \ldots B_m$，也就是要找，$\{A_1, A_2, \ldots, A_n\}$不是超键的FD。一个启发是：尽可能地往FD的右边增加足够多的由$\{A_1, A_2, \ldots, A_n\}$决定的属性。图3-24说明了属性集合如何被分解到两个重叠的关系模式，其中一个模式包含了上述FD的所有属性，而另一个包含了位于这个FD左边的属性和不属于FD的所有属性，即除了只属于B而不属于A的属性的所有属性。

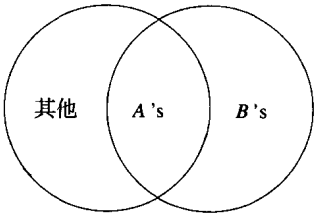


图3-24 基于不满足BCNF的关系模式分解

例3.28 考虑图3-21中的关系Movies。从例3.25知FD

```
title year → length filmType studioName
```

违反BCNF。在这个FD中，右边已经包含了由title和year函数决定的所有属性，所以可以把Movies分解为：

1. 含有上述FD所有属性的模式，也就是：

```
{title, year, length, filmType, studioName}
```

2. 除了上述FD右边的三个属性外，包含了其他所有属性的模式。也就是：

```
{title, year, starName}
```

上面两个模式就是例3.24中给出的Movies1和Movies2。例3.26中也证明了它们都满足BCNF。 □ 108

例3.29 考虑例3.21中的关系MovieStudio，它存储了关于电影（Movies），及其所属制片厂和这些制片厂地址的信息。图3-25给出了这个关系的模式和一些典型的元组。

title	*year*	*length*	*filmType*	*studioName*	*studioAddr*
Star Wars	1977	124	color	Fox	Hollywood
Mighty Ducks	1991	104	color	Disney	Buena Vista
Wayne's World	1992	95	color	Paramount	Hollywood
Addams Family	1991	102	color	Paramount	Hollywood

图3-25 关系MovieStudio

注意MovieStudio包含冗余信息，这是因为studioName为“Paramount”的两个元组中studioAddr相同。然而这个问题与例3.28中出现的问题不同，在例3.28中，问题的缘由是一个多对多的联系（给定的一部电影中有多个影星）同电影的其他信息存储在一起。而在这个例子中，每个属性都是单值：如影片的length，连接电影和其惟一的制片厂的联系ownedBy和

制片厂的地址studioAddr。

在这个例子中，问题是出于存在一个“传递依赖”。也就是，与例3.21中所讨论的，关系MovieStudio含有FD集合：

title year → studioName
studioName → studioAddr

对这两个FD使用传递规则可得：

title year → studioAddr

也就是说，title和year（关系Movies的键）函数决定制片厂的地址，即拥有该影片的制片厂的地址。因为

title year → length filmType

显然是关系中存在的另一个函数依赖，所以可得出{title, year}是MovieStudio的键。事实上，它也是惟一的键。

另一方面，传递规则的其中一个FD：

studioName → studioAddr

是左边不是超键的非平凡FD。这就说明了MovieStudio不满足BCNF。于是可以根据上
109 述分解规则解决该冗余问题。分解后的第一个模式是这个FD本身属性的集合，即{studioName, studioAddr}。第二个模式是除属性studioAddr外的MovieStudio中所有属性的集合。不包含studioAddr的原因是它位于FD的右边，这个模式为：

{title, year, length, filmType, studioName}

图3-25在这些模式上投影得出的两个关系是MovieStudio1和MovieStudio2，分别显示在图3-26和图3-27中。这两个关系都满足BCNF。回想3.5.7节的讨论，对于分解后的每个关系，还需通过计算它们属性集合的每个子集的闭包，使用给定FD的完全集合来计算它们的FD集合。通常，这个过程的复杂度为分解后关系中属性数目的指数幂，但也可根据3.5.7节进行简化。

在这个例子中，找出MovieStudio1的基本依赖很容易，它是：

title year → length filmType studioname

而在MovieStudio2中，仅有的非平凡FD为：

studioName → studioAddr

因此，MovieStudio1的惟一键是{title, year}，MovieStudio2的惟一键是{studioName}。在这两个关系中所存在的非平凡FD的左边都包含了键。□

title	*year*	*length*	*filmType*	*studioName*
Star Wars	1977	124	color	Fox
Mighty Ducks	1991	104	color	Disney
Wayne's World	1992	95	color	Paramount
Addams Family	1991	102	color	Paramount

图3-26 关系MoveiStudio1

虽说从前面的每个例子看出，已有足够的分解规则支持将一个关系分解为满足BCNF的多个关系，但情况并非如此。

studioName	*studioAddr*
Fox	Hollywood
Disney	Buena Vista
Paramount	Hollywood

图3-27 关系MovieStudio2

110

例3.30 将例3.29中的关系加以总结并生成长度大于2的FD链。考虑关系模式

{title, year, studioName, president, presAddr}

这个关系的每个元组给出信息包括片名、所属制片厂、制片厂经理（president）及经理地址（presAddr）的信息。这个关系上可能有的三个FD是：

title year → studioName
studioName → president
president → presAddr

关系的惟一键是{title, year}。因此上述最后两个FD都违反了BCNF条件。假设利用下面的FD开始分解，即：

studioName → president

首先，要往这个函数依赖的右边添加在studioName闭包中的其他属性。在FD studioName→president和FD president→presAddr上运用传递规则，可得到FD

studioName → presAddr

组合左边都是studioName的两个FD，得到

studioName → president presAddr

它的右边已达到最大化，于是把关系分解为下面两个关系模式：

{title, year, studioName}
{studioName, president, presAddr}

若使用3.5.7节中的投影算法，就可确定第一个关系有基本FD：

title year → studioName

而第二个关系有基本FD：

studioName → president
president → presAddr

因此，第一个关系惟一的键是{title, year}，它满足BCNF。第二个关系也有惟一键{studioName}，但是FD：

111

president → presAddr

违反BCNF。因此，必须再进一步利用这个FD对第二个关系进行分解。所得的三个关系均满足BCNF，其模式为：

{title, year, studioName}
{studioName, president}
{president, presAddr}

□

通常，必须反复使用分解规则，直至所得的关系均满足BCNF。这样做一定会成功，因为每次运用分解规则后，所得到的模式中的属性个数都比R的属性个数少。例如例3.27，当分解为只有两个属性的集合后，所得关系肯定满足BCNF。但很多情况下，有多个属性的关系也满足BCNF。

3.6.5 从分解中恢复信息

下面要考虑的是为什么3.6.4节中的分解算法仍保持原关系中的信息。它的思想是如果遵循这个算法，则原始元组的投影可被重新连接，恢复成与原元组完全相同的元组。

为了简化起见，下面只考虑关系R（A，B，C）和一个违反BCNF条件的FD $B \rightarrow C$。如同例3.29中的关系一样，FD $B \rightarrow C$与另一FD $A \rightarrow B$构成传递依赖链。在这种情况下，{A}是惟一的键，而$B \rightarrow C$的左边不是超键。另一种可能的情况是，$B \rightarrow C$是惟一的非平凡FD，此时惟一的键是{A，B}。同样，$B \rightarrow C$的左边仍不是超键。在这两种情况下，根据FD $B \rightarrow C$可把R的模式分解为{A，B}和{B，C}。

令t是R的一个元组，并可写为t = {a，b，c}，其中a，b和c是t相应于属性A，B和C的分量。元组t在关系模式{A，B}上投影是（a，b），而在关系模式{B，C}上投影是（b，c）。

由于元组在B上的分量相同，可以连接{A，B}的元组和{B，C}的元组。尤其当连接（a，b）和（a，c）后，可以得到原始元组t =（a，b，c）。也就是说，不管哪个元组t都可以恢复原来的信息。

然而，恢复那些被分解的关系元组，并不足以确保原始关系R一定能被分解的关系正确表示。如果R中有元组t =（a，b，c）和v =（d，b，e），那会有什么样的结果？如图3-28所示，把t投影到{A，B}上可得u =（a，b），而把v投影到{B，c}上可得w =（b，e）。

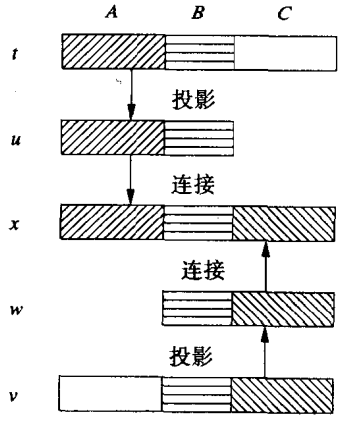


图3-28 连接投影后的两个元组

由于u和w在B上的分量相同，可把它们进行连接。则所得的元组为x =（a，b，e）。x可能是个伪元组吗？也就是说，x可能不是R的一个元组吗？

112

因为已假设R中存在FD $B \rightarrow C$，所以答案是“否”。这个FD意味着R中的两个元组只要在B上的分量相同，它们在C上的分量一定相同。而t和v满足这个条件（都有b），于是它们在C上的分量也应相同。这意味着，$c = e$。也就是说，假设的不同值其实是相同的。因此，（a，b，c）就是（a，b，e），即$x = t$。

由t在R中可知x一定也在R中。换言之，只要R中存在FD $B \rightarrow C$，连接两个投影后的元组不会生成一个伪元组。而且，每一个连接形成的元组肯定属于R。

通常这个论断是正确的。虽然这里假设A，B和C都是单个的属性，事实上，它对属性集合同样也成立。也就是，对于任何违反BCNF条件的FD，令B是位于它左边的集合，C是位于它右边的但不在左边出现的属性集合，A是不出现在任何一边的属性集合。则有下列的结论：

- 如果根据3.6.4节所提的方法分解一个关系，不论用何种方式连接新生成关系都可以正确地恢复原始关系。

如果使用一种不基于某个FD的方法分解关系，可能不能恢复原始关系。下面是一个这样的例子。

例3.31 假设有一个与上面关系相同的R（A，B，C），但是关系中不存在FD $B \rightarrow C$。R可能包含了下面两个元组

A	B	C
1	2	3
4	2	5

113

则R在{A，B}和{B，C}上的投影分别为

A	B
1	2
4	2

和

B	C
2	3
2	5

因为这四个元组在B上的分量相同，值均为2，所以可把这两个关系进行连接。此时，就会得到

A	B	C
1	2	3
1	2	5
4	2	3
4	2	5

从图中可看出，所得关系的元组多于原始关系中的元组，即存在两个伪元组（1，2，5）和（4，2，3）。□

3.6.6 第三范式

人们有时会不想对一个不满足BCNF的关系继续分解。下面给出一个典型的例子。

例3.32 假设关系Bookings含有属性：

1. title，电影的名字。
2. theater，放映电影的影院名。
3. city，影院所处的城市。

元组（m，t，c）的意思是名为m的电影正在城市c中的影院t放映。

对于这个关系可声明下面的FD：

theater → city
title city → theater

114

第一个FD说明一个影院位于一个城市，而第二个FD的含义虽不明显，但也可基于这样的假设，即一部电影不能在同一个城市中的两个影院中放映。这样假设只是为了说明这个例子。

首先要找到键。其中没有单个的属性是键。例如，title不是键是因为一部电影可同时在多个影院或在多个城市放映[⊖]。同样，theater不是键，虽然theater函数决定city，但仍存在可以同时放映多部电影的影院。这也就是说，theater不是键。而因为在一个城市内通常有多个影院和多部电影放映，所以city也不是键。

另一方面，两个具有两个属性的集合是键。很明显，{title，city}是键，这是因为给

⊖ 该例中假定没有同名的两个“当前”的电影，即使是在前述已经看到在不同年份中可以制作了两个同名的电影。

定的FD已说明了这两个属性函数决定了theater。

{theater, title}也是键。观察FD theater→city，运用习题3.5.3(a)的增广规则就可推出theater title→city。因为直觉上，若theater可函数决定city，那么theater和title肯定能函数决定city。

而剩下的属性对{city, theater}不能函数决定title，因此不是键。由此，仅有的两个键是

{title, city}
{theater, title}

现在，立即见到了一个违反BCNF条件的FD theater→city。根据这个FD，原关系模式可被分解为下列两个模式。

{theater, city}
{theater, title}

但是这个分解存在一个问题。考虑FD

title city → theater

分解后的关系中存在满足FD theater→city的模式（如关系{theater, city}），但若要对分解后的关系进行连接会得不到满足title city→theater的关系。例如，根据上述关系中的FD，如下两个关系

theater	*city*
Guild	Menlo Park
Park	Menlo Park

115

其他范式

既然有“第三范式”，那前两个“范式”是什么呢？确实它们也有定义，但现在已很少使用它们了。第一范式（first normal form）只简单地要求每个元组的各分量是原子值。而第二范式（second normal form）的限制比3NF少。它允许关系中存在传递FD，但不允许有左边是键真子集的非平凡FD存在。还有将在3.7节中介绍的“第四范式”。

和

theater	*title*
Guild	The Net
Park	The Net

是允许的。若连接这两个关系，就可以得到下面只含有两个元组的关系

theater	*city*	*title*
Guild	Menlo Park	The Net
Park	Menlo Park	The Net

但是在这个关系中不存在FD title city→theater。 □

解决上述问题的方法就是稍稍放宽对BCNF的限制。以能够允许例3.32中的关系模式存在，在这种关系模式中，人们不可能保证分解后的BCNF关系使每一个原有的FD仍然成立。这种限制较少的条件就是第三范式条件。

• 若在关系R中存在非平凡FD $A_1\ A_2 \ldots A_n \rightarrow B$，且要么$\{A_1, A_2, \ldots, A_n\}$是超键，要么$B$属

于某个键，则认为R属于第三范式（3NF）。

一个属于某个键的属性常被称为是主属性。因此第三范式的另一种说法是“3NF关系是那些仅存在左边是超键，或者右边是主属性的非平凡FD的关系”。

注意，3NF与BCNF条件的区别是句子：“B属于某个键（例如，主属性）”。这个句子使得例3.32中的FD theater→city成为合法，因为其右边city是主属性。

证明3NF满足该假定已经超出了本书范围。也就是说，一个关系模式总是可以无信息丢失地分解到3NF，并且原关系中的FD仍存在于分解后的多个关系中。虽说3NF已够用，但不分解到BCNF就会产生冗余。 116

3.6.7 习题

习题3.6.1 对于下列的关系模式和FD集合：

* a) $R(A, B, C, D)$，含有FD：$AB \rightarrow C$，$C \rightarrow D$和$D \rightarrow A$。

* b) $R(A, B, C, D)$，含有FD：$B \rightarrow C$和$B \rightarrow D$。

c) $R(A, B, C, D)$，含有FD：$AB \rightarrow C$，$BC \rightarrow D$，$CD \rightarrow A$和$AD \rightarrow B$。

d) $R(A, B, C, D)$，含有FD：$A \rightarrow B$，$B \rightarrow C$，$C \rightarrow D$和$D \rightarrow A$。

e) $R(A, B, C, D, E)$，含有FD：$AB \rightarrow C$，$DE \rightarrow C$和$B \rightarrow D$。

f) $R(A, B, C, D, E)$，含有FD：$AB \rightarrow C$，$C \rightarrow D$，$D \rightarrow B$和$D \rightarrow E$。

完成以下作业：

i) 指出所有违反BCNF条件的FD。不要忘记考虑那些不在上述集合中，但是由它们推出的FD。可是没必要考虑那些右边有多个属性的FD。

ii) 根据需要把关系分解为满足BCNF的关系集合。

iii) 指出所有违反3NF条件的FD。

iv) 根据需要把关系分解为满足3NF的关系集合。

习题3.6.2 在3.6.4节中曾指出，若可能的话，可以增广一个违反BCNF条件的FD的右边集合。但这只是个可选的步骤。考虑模式为{A，B，C，D}，并含有FD $A \rightarrow B$和$A \rightarrow C$的关系。因为R的惟一键是{A，D}，所以这两个FD都违反了BCNF的条件。假设根据$A \rightarrow B$分解R，那么最终所得的结果是否与对违反BCNF条件的FD $A \rightarrow BC$进行分解所得结果相同？若相同，则为什么相同，否则为什么不同？

! **习题3.6.3** 假设R与习题3.6.2中的R相同，但是它含有的FD为$A \rightarrow B$和$B \rightarrow C$。再次比较使用$A \rightarrow B$进行分解和使用$A \rightarrow BC$进行分解的结果。

! **习题3.6.4** 假设有一个关系模式$R(A, B, C)$，它含有FD $A \rightarrow B$。并假设要把它分解为$S(A, B)$和$T(B, C)$。给出R的一个实例，使其投影到S和T后再将两者进行连接而所得的结果与原实例不同。 117

3.7 多值依赖

“多值依赖”(multivalued dependency)是两个属性或属性集合之间相互独立的断言。它是广义的函数依赖，在某种意义上每个FD就意味着一个相应的多值依赖。但是仍然存在有不能用FD解释的属性集合相互独立的情况。这一节就要说明引起多值依赖的原因和在数据库模式设计中如何使用多值依赖。

3.7.1 属性独立及伴随其产生的冗余

在设计关系模式时，有时会有这样的情况发生，即某个模式是BCNF，但在相应的关系中

还有与FD无关的冗余存在。在BCNF模式中最常见的导致冗余的情形，是试图把两个或多个多对多联系置于同一个关系中。

例3.33 在这个例子中，假设影星可有多处地址。地址分为街道（street）和城市（city）两部分。在这个关系中，除了影星的名字（name）和地址属性外，还有Stars-in信息，即含有某个影星所演电影的片名（title）和电影放映的年代（year）。图3-29给出了这个关系的一个典型的实例。

name	*street*	*city*	*title*	*year*
C. Fisher	123 Maple St.	Hollywood	Star Wars	1977
C. Fisher	5 Locust Ln.	Malibu	Star Wars	1977
C. Fisher	123 Maple St.	Hollywood	Empire Strikes Back	1980
C. Fisher	5 Locust Ln.	Malibu	Empire Strikes Back	1980
C. Fisher	123 Maple St.	Hollywood	Return of the Jedi	1983
C. Fisher	5 Locust Ln.	Malibu	Return of the Jedi	1983

图3-29 独立于电影的地址集合

图中给出了假想的Carrie Fisher的两个地址和他主演的三部著名的电影。把某个地址和某部影片进行连接是没有理由的。由此，表达影星地址和电影独立性的惟一途径是把地址和电影相连的各种组合都罗列出来。但是这样的组合显然有冗余。例如，图3-29中Carrie Fisher的地址重复了三次（每个地址对应一部电影），电影重复了两次。

图3-29中不存在违反BCNF条件的FD，也不存在非平凡FD。例如，属性city并非由其他
118 四个属性函数决定。因为一个影星可在不同城市的同一个街道有两处居所。那么就存在两个除了city分量值不同外，其他分量值均相同的两个元组。因此，

```
name street title year → city
```

不是该关系的FD。同样，这五个属性中的任一个都不由其他四个属性函数决定，这一点留给读者来证明。因为不存在非平凡FD，所以键由五个属性形成，于是关系中不存在违反BCNF条件的FD。 □

3.7.2 多值依赖的定义

多值依赖（常缩写为MVD）是指在关系R中，当给定某个属性集合的值时，存在有一组与关系中所有其他属性值独立的属性值。精确一点说，R中MVD

$$A_1 A_2 \ldots A_n \twoheadrightarrow B_1 B_2 \ldots B_m$$

成立，是指若给定R中的A属性值，则存在有一组B中的值，这些值独立于R中既不是A也不是B的属性集合的值。更精确的说法是，若要MVD存在，则

对于R中每个在A上取值相同的元组对t和u，能找到满足下列条件的元组v：

1. 在A属性上的取值与t和u相同；
2. 在B属性上取值与t相同；
3. 与不同于A和B的其他所有属性上取值与u相同。

若t和u交换也能使用这个规则推断出有第四个元组存在，这个元组同u在B属性上的取值相同，而与t在其他属性上的取值相同。结果是，对于任一组A的值，B和其他属性值的各种组合出现在不同元组。图3-30给出了当MVD存在时，v是如何与t和u相关的。

通常，可以假设一个MVD中的A和B（左边和右边）不相交。然而，作为一个FD，允许把A中的某些属性添加到B中。也要注意到它与FD不同，FD是从右边为单个属性开始，并且允许

用缩略形式表示右边的属性集合，而MVD一开始就考虑右边是属性集合的情形。如同例3-35所见，并不总能把MVD的右边集合分解为单个属性。

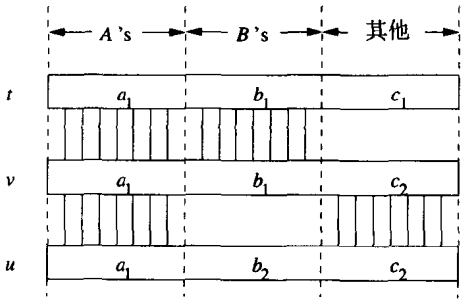


图3-30 确保v存在的一个多值依赖

119

例3.34 例3.33给出了一个MVD，用符号表示就是

name ↠ street city .

也就是说，对于任一个影星的name，地址与影星所演的电影可以不同组合形式出现。举一个例子来说明怎样应用MVD的正式定义，考虑图3-29中第一个和第四个元组：

name	*street*	*city*	*title*	*year*
C. Fisher	123 Maple St.	Hollywood	Star Wars	1977
C. Fisher	5 Locust Ln.	Malibu	Empire Strikes Back	1980

设上图中的第一个元组为t，第二个元组为u，那么根据这个MVD，可知R中存在这么一个元组，其name为C.Fisher，street和city与元组t取值相同，而其他属性（title和year）与元组u取值相同。观察图3-29，其中确实存在这么一个元组，即第三个元组。

同样，令t为上图中的第二个元组，而u为第一个元组。那么根据这个MVD，可知存在这么一个元组，name，street和city与t取值相同，而name，title和year与u取值相同。观察图3-29，确实也存在这么一个元组，即第二个元组。 □

3.7.3 多值依赖的推论

有很多关于MVD的规则，它们与3.5节中所给的FD规则相似。例如，MVD遵循

- 平凡依赖规则（trivial dependency rule），即如果某个关系中存在MVD

$$A_1 A_2 \ldots A_n \longrightarrow\!\!\!\!\!\rightarrow B_1 B_2 \ldots B_m$$

120

则$A_1 A_2 \ldots A_n \longrightarrow\!\!\!\!\!\rightarrow C_1 C_2 \ldots C_k$也存在，其中$C$是$B$中属性和$A$中一个或多个属性的并。相反地，也可以移走$B$中属于$A$的属性，推导出MVD $A_1 A_2 \ldots A_n \longrightarrow\!\!\!\!\!\rightarrow D_1 D_2 \ldots D_r$成立，其中$D$是$B$中不属于$A$的属性集合。

- 传递规则（transitive rule），即如果关系中存在$A_1 A_2 \ldots A_n \longrightarrow\!\!\!\!\!\rightarrow B_1 B_2 \ldots B_m$和$B_1 B_2 \ldots B_m \longrightarrow\!\!\!\!\!\rightarrow C_1 C_2 \ldots C_k$，则

$$A_1 A_2 \ldots A_n \longrightarrow\!\!\!\!\!\rightarrow C_1 C_2 \ldots C_k$$

成立。然而，任何既属于B又属于C的属性要从右边集合中除去。

另一方面，MVD不遵循分解/结合规则中的分解部分，下面给出了这样的一个例子。

例3.35 再次考虑图3-29，图中关系存在MVD：

name ↠ street city

若把分解规则应用在这个MVD上，就会有

name ↠ street

也成立。这个MVD意味着每个影星的街道地址（street）独立于其他属性，其中包括属性city。然而，这是错误的结论。例如，考虑图3-29中的前两个元组，根据上述假想的MVD可推出R中存在street相互交换的元组：

name	*street*	*city*	*title*	*year*
C. Fisher	5 Locust Ln.	Hollywood	Star Wars	1977
C. Fisher	123 Maple St.	Malibu	Star Wars	1977

但是这种元组是不存在的，因为street为5 Locust Ln.是处于城市Malibu中，而不是在Hollywood。 □

还有一些需要知道的关于MVD的规则，第一条是：

- 每个FD也是MVD。也就是说，若存在$A_1\,A_2\ldots A_n \rightarrow B_1\,B_2\ldots B_m$，那$A_1\,A_2\ldots A_n \twoheadrightarrow B_1\,B_2\ldots B_m$也成立。

为了说明这条规则成立的理由，假设关系R中存在FD

$$A_1\,A_2\ldots A_n \rightarrow B_1\,B_2\ldots B_m$$

121 而t和u是R中在A上的值相同的元组。为了证明MVD $A_1\,A_2\ldots A_n \twoheadrightarrow B_1\,B_2\ldots B_m$成立，就要证明也包含这样的元组$v$，它与$t$和$u$在$A$上的取值相同，与$t$在$B$上的取值相同，并且与$u$在其他属性上的取值相同。但是由于$v$可能是$u$，那么$u$与$t$和$u$在$A$上的取值肯定相同，这由假设可得。而FD $A_1\,A_2\ldots A_n \rightarrow B_1\,B_2\ldots B_m$则确保了$u$与$t$在$B$上的取值相同。当然$u$与自身在其他属性上的取值相同。因此，当存在FD时，肯定存在相应的MVD。

另一个规则是*互补规则*（complementation rule），但是不存在与该规则对应的FD规则。

- 如果关系R中存在MVD $A_1\,A_2\ldots A_n \twoheadrightarrow B_1\,B_2\ldots B_m$，则$R$中也有$A_1\,A_2\ldots A_n \twoheadrightarrow C_1\,C_2\ldots C_k$成立，其中$C$属性是$R$中不存在于$A$和$B$中的所有其他属性。

例3.36 再次考虑图3-29，图中关系存在MVD：

```
name →→ street city
```

则互补规则说明了

```
name →→ title year
```

在R中也成立，这是因为title和year是不在第一个MVD中被提及的属性。第二个MVD直觉上意味着每个影星可主演很多部片子，而这是独立于影星的地址。 □

3.7.4 第四范式

3.7.1节中由MVD引起的冗余，可通过依赖使用新的分解算法来消除。这一节将引入一种新的范式，即“第四范式”。在这个范式中，同在BCNF中消除所有违反BCNF的FD一样，消除了所有的非平凡（下面有定义）MVD。这样做的结果是，分解后的关系中既不存在3.6.1节中讨论的FD带来的冗余，也不存在3.7.1节中讨论的MVD带来的冗余。

当下列条件成立时，关系R中MVD $A_1\,A_2\ldots A_n \twoheadrightarrow B_1\,B_2\ldots B_m$是非平凡的：

1. B中没有一个属性属于A。
2. R中存在除了A和B之外的属性。

“第四范式”条件本质上是BCNF条件，但它不是应用于FD，而是应用于MVD。正式的定义是：

122 - 在R中，如果任一非平凡MVD $A_1\,A_2\ldots A_n \twoheadrightarrow B_1\,B_2\ldots B_m$成立时，都有$\{A_1, A_2, \ldots, A_n\}$是超键，则$R$属于第四范式（4NF）。

也就是说，若一个关系满足4NF，则每个非平凡MVD确实是左边为超键的FD。注意，键和超键概念只与FD有关，添加MVD不会改变“键”的定义。

例3.37 图3-29中的关系不属于4NF。例如

```
name →→ street city
```

虽说是一个非平凡MVD，但其中的name不是超键。事实上，这个关系仅有的键是所有属性的集合。 □

第四范式是广义的BCNF，每个FD又是一个MVD。所以只要违反BCNF条件，也必然违反了4NF条件。换而言之，只要满足4NF，也必然满足BCNF。

然而也存在一些满足BCNF但不满足4NF的关系。图3-29就是一个例子。这个关系仅有的键是所有属性的集合，因此也不存在非平凡FD，而它确实满足BCNF。例3.37给出的关系同样也不属于4NF。

3.7.5 分解到第四范式

4NF的分解算法与BCNF的分解算法非常类似。对于$A_1\ A_2 \ldots A_n \rightarrow\!\!\!\rightarrow B_1\ B_2 \ldots B_m$，若{ A_1，A_2，…，A_n}不是超键，则认为此MVD违反了4NF条件。这个MVD可能确实符合定义，或者只是从相应的FD $A_1\ A_2 \ldots A_n \rightarrow B_1\ B_2 \ldots B_m$ 对应而来，因为FD也是MVD。那么可把存在违反4NF条件的MVD关系R分解为两个模式：

1. 含有A和B的全部属性。
2. 含有A的全部属性和不在A和B中的全部属性。

例3.38 继续考虑例3.37，图中存在违反4NF条件的MVD

```
name →→ street city
```

由分解规则可知，把含有五个属性的模式分解为两个模式，其中一个是只含有上述三个属性的模式，而另一个模式含有name和不在MVD中出现的属性，即title和year。于是，分解后的两个模式为 123

```
{name, street, city}
{name, title, year}
```

因为在这两个模式中不存在非平凡多值（或函数）依赖，所以它们都满足4NF。

投影多值依赖

当把某个模式分解为4NF时，有必要在分解后的关系中找到MVD。人们希望找寻MVD的过程简单。然而，不存在类似于计算属性集合闭包来找FD这样简单的方法（参见3.5.3节）。事实上，推导函数依赖和多值依赖的完全规则集相当复杂，并且已超出了本书的范围。3.9节中有几处地方提及了怎么对待这样的问题。

幸运的是，通常可以通过使用传递规则、互补规则和交集规则[习题3.7.7(b)]得到分解后关系的相应MVD。建议读者在例子和习题中使用这种方法。

注意，在模式为{name, street, city}的关系中存在平凡MVD：

```
name →→ street city
```

其原因是因为它包含了所有的属性。同样，在模式为{name, title, year}的关系中，MVD：

```
name →→ title year
```

也是平凡的。若分解出的两个关系不属于4NF，则还要对它继续进行分解。 □

对于BCNF分解，每步分解后所得关系模式的属性个数都严格少于原始关系的属性个数。

因此，最后肯定能得到不需继续分解的模式，也就是，它们满足4NF。此外，这也说明了3.6.5节中给出的分解法对于MVD同样适用。即，根据MVD $A_1 A_2 \ldots A_n \rightarrow\!\!\!\rightarrow B_1 B_2 \ldots B_m$分解一个关系时，这个依赖就足以证明从分解后的关系中重构原始关系的正确性。

3.7.6 范式间的联系

如前所述，4NF暗含于BCNF，同样BCNF暗含于3NF。图3-31给出了满足这三个范式的关系模式（包括函数依赖）集合之间的联系。也就是说，若含有特定依赖的关系满足4NF，则它也满足BCNF和3NF。而对于满足BCNF的含有特定依赖的关系而言，它也满足3NF。

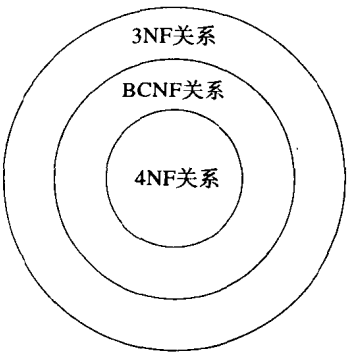


图3-31 4NF暗含了BCNF，BCNF暗含了3NF

比较这些范式的另一种方法是比较分解到各范式的关系的性质。图3-32的表中给出了对这几种范式性质的总结。也就是，BCNF（因此4NF）消除了由FD带来的冗余和其他异常，而只有4NF才能消除由非平凡MVD（不是FD）带来的附加冗余。通常情况下3NF就足以消除这些冗余，但是仍然存在有它不能消除的例子。总是选择分解到3NF，是因为这样做能保持FD，即在分解后的关系中仍存在FD（虽然在本书中没有讨论相应的算法）。BCNF不保证能保持FD。虽然在典型的例子中仍然能保持MVD，但是没有一个范式能做这种保证。

性质	3NF	BCNF	4NF
消除FD冗余	大多数	Yes	Yes
消除MVD冗余	No	No	Yes
保持FD	Yes	可能	可能
保持MVD	可能	可能	可能

图3-32 范式的性质和分解

3.7.7 习题

* **习题3.7.1** 假设关系R（A，B，C）中存在MVD $A \rightarrow\!\!\!\rightarrow B$。若$R$的当前实例中含有元组（$a$，$b_1$，$c_1$），（$a$，$b_2$，$c_2$）和（$a$，$b_3$，$c_3$），那么$R$中必存在其他哪些元组？

* **习题3.7.2** 假设有一个记录了人名、其社会保险号和生日的关系。对于他们的每个孩子也记录了他们的名字、社会保险号、每人的生日，还记录了每人拥有的汽车、车牌号和厂家。下面给出这个关系的字段：

$$(n,\ s,\ b,\ cn,\ cs,\ cb,\ as,\ am)$$

其中

1. n是人名，而s是社会保险号。
2. b是n的生日。
3. cn是n的某个孩子的名字。
4. cs是cn的社会保险号。
5. cb是cn的生日。
6. as是n的某部车的车牌号。
7. am是as的厂家。

对这个关系：

a) 指出希望含有的函数依赖和多值依赖。

b) 分解到4NF。 126

习题3.7.3 对于下面的关系模式和依赖

* a) $R(A, B, C, D)$，存在MVD $A \twoheadrightarrow B$和$A \twoheadrightarrow C$。

b) $R(A, B, C, D)$，存在MVD $A \twoheadrightarrow B$和$B \twoheadrightarrow CD$。

c) $R(A, B, C, D)$，存在MVD $AB \twoheadrightarrow C$和FD $B \rightarrow C$。

d) $R(A, B, C, D, E)$，存在MVD $A \twoheadrightarrow B$和$A \twoheadrightarrow C$，以及FD $A \rightarrow D$和$AB \rightarrow E$。

分别回答下面问题：

i. 指出违反4NF的依赖。

ii. 把关系分解为多个属于4NF的关系。

! **习题3.7.4** 在习题2.2.5中已对联系Births的不同假设进行过讨论。对于不同的假设，指出在结果关系中存在的MVD（不是FD）。

习题3.7.5 非正式地证明为什么例3.33中五个属性中的任一个属性都不能由其他四个属性函数决定。

! **习题3.7.6** 使用MVD的定义，说明为什么存在互补规则。

! **习题3.7.7** 证明下面给出MVD规则：

* a) 联合规则（union rule）。X，Y和Z都是属性集合，若$X \twoheadrightarrow Y$和$X \twoheadrightarrow Z$成立，则$X \twoheadrightarrow (Y \cup Z)$成立。

b) 交集规则（intersection rule）。X，Y和Z都是属性集合，若$X \twoheadrightarrow Y$和$X \twoheadrightarrow Z$成立，则$X \twoheadrightarrow (Y \cap Z)$成立。

c) 差异规则（difference rule）。X，Y和Z都是属性集合，若$X \twoheadrightarrow Y$和$X \twoheadrightarrow Z$成立，则$X \twoheadrightarrow (Y - Z)$成立。

d) 平凡MVD（trivial MVD）。若$Y \subseteq X$（Y是X的子集），则在任一个关系中都存在$X \twoheadrightarrow Y$。

e) 平凡MVD的另一来源。若$X \cup Y$包含了关系R的所有属性，则R中存在$X \twoheadrightarrow Y$。

f) 通过移动左边和右边来移动共享属性集合（removing attributes shared by left and right side）。若$X \twoheadrightarrow Y$，则$X \twoheadrightarrow (Y - X)$成立。

! **习题3.7.8** 给出一个反例，说明为什么下面关于MVD的规则不正确。

* a) 若$A \twoheadrightarrow BC$，则$A \twoheadrightarrow B$。

b) 若$A \twoheadrightarrow B$，则$A \rightarrow B$。

c) 若$AB \twoheadrightarrow C$，则$A \twoheadrightarrow C$。

3.8 小结

- 关系模式（relational model）：关系是表示信息的表。属性位于每列的上部，每个属性都有相应的域，或者数据类型。行被称为元组，每一行都有一个分量与关系属性对应。
- 模式（schema）：关系名和该关系所有属性的结合。多个关系模式形成一个数据库模式。一个关系或多个关系的特定数据叫做关系模式实例或数据库模式的实例。 127
- 把实体集转化为关系：实体集关系的属性与相应实体集的属性相对应。而弱实体集E例外，它的属性必须包含所有有助于识别E实体集的键属性。
- 把联系转化为关系：E/R联系关系的属性与每个连接此联系的实体集的键属性相对应。若

一个联系连接弱实体集，则不需为这个联系生成关系。

- *把isa层次转化为关系*：一种方法是拆分层次中不同的实体集，为每个实体集创建相应的含有所有必要属性的关系。另一种方法是为层次中实体集的每个可能的子集创建相应的关系，每个实体形成关系中的一个元组。关系中的所有元组对应于该实体所属的实体集。第三种方法是利用空值仅只创建一个关系，每个实体对应于关系中的一个元组，若该实体没有某个属性，则在关系相应的分量处填入空值。
- *函数依赖*（functional dependency）：若关系中的两个元组在某些属性上取值相同，则它们在另一些属性上取值也相同。
- *关系的键*（key）：关系的*超键*（superkey）是函数决定该关系中所有属性的属性集合。若一个超键不存在任何能函数决定所有属性的真子集，则它就是键。
- *函数依赖的推论*：一组规则，这些规则能推出，在满足给定FD集的任一关系实例中有另一个FD $X \rightarrow A$成立。证明存在FD $X \rightarrow A$的最简单方法是计算X关于给定FD的集合闭包，并判断X闭包中是否包含A。
- *关系分解*：一个关系能无损地被分解为两个关系，只要分解后两个关系模式中的相同属性集至少能构成其中一个模式的超键。
- Boyce-Codd*范式*：若关系中的非平凡FD都指明某个超键函数决定其他属性，则认为此关系满足BCNF。任何关系都可以无损地分解为BCNF关系集。BCNF的主要优点是它消除了由FD引起的冗余。
- *第三范式*：有时分解到BCNF会丢失某些FD。一种比BCNF限制较宽的范式为3NF，它允许FD $X \rightarrow A$（其中X可以不是超键，而A是键属性）的存在。3NF不保证消除所有FD引起的冗余，但实际上大多数情况下可以消除冗余。
- *多值依赖*（MVD）：关系中两个属性集的值用其所有可能的组合方式出现。
- *第四范式*：关系中MVD也能引起冗余。4NF同BCNF一样，也不允许存在非平凡MVD（除非它们是BCNF中允许存在的FD）。一个关系有可能无损地分解为4NF关系集。

128

3.9 参考文献

经典的关于关系模式的文章是由Codd写的[4]。这篇文章介绍了函数依赖的思想和基本的关系概念。文中也描述了第三范式，而Boyee-Codd范式出现在Codd后来的文章中。

多值依赖和第四范式由Fagin在[7]中给出了定义。另外，多值依赖的思想也独立出现于[6]和[9]中。

Armstrong是第一个研究推出FD规则[1]的人。本书中给出的关于FD（包括Armstrong公理）和MVD的规则来自于[2]。通过计算属性集的闭包来验证FD的方法则是参考[3]。

有许多算法和/或者证明并未在本书中给出，其中包括如何推出MVD，如何把多值依赖投影到分解后的关系，以及如何无丢失FD地把关系分解到3NF。这些或其他与依赖有关的问题见[8]。

1. Armstrong, W. W., "Dependency structures of database relationships," *Proceedings of the 1974 IFIP Congress*, pp. 580–583.

2. Beeri, C., R. Fagin, and J. H. Howard, "A complete axiomatization for functional and multivalued dependencies," *ACM SIGMOD International Conference on Management of Data*, pp. 47–61, 1977.

3. Bernstein, P. A., "Synthesizing third normal form relations from functional dependencies," *ACM Transactions on Database Systems* **1**:4, pp. 277–298, 1976.

4. Codd, E. F., "A relational model for large shared data banks," *Comm. ACM* **13**:6, pp. 377–387, 1970. 129

5. Codd, E. F., "Further normalization of the data base relational model," in *Database Systems* (R. Rustin, ed.), Prentice-Hall, Englewood Cliffs, NJ, 1972.

6. Delobel, C., "Normalization and hierarchical dependencies in the relational data model," *ACM Transactions on Database Systems* **3**:3, pp. 201–222, 1978.

7. Fagin, R., "Multivalued dependencies and a new normal form for relational databases," *ACM Transactions on Database Systems* **2**:3, pp. 262–278, 1977.

8. Ullman, J. D., *Principles of Database and Knowledge-Base Systems, Volume I*, Computer Science Press, New York, 1988.

9. Zaniolo, C. and M. A. Melkanoff, "On the design of relational database schemata," *ACM Transactions on Database Systems* **6**:1, pp. 1–47, 1981. 130

第4章 其他数据模型

实体联系模型（E/R模型）和关系模型仅是当今数据库系统中的两个重要的模型。这一章，将再介绍几个越来越重要的其他数据模型。

本章先从讨论面向对象（Object-Oriented）数据模型开始。数据库系统面向对象化的一种方法，是扩展面向对象编程语言（如C++，Java等），使其支持持久性。通常程序中的对象在程序结束后将消失，但是，DBMS的基本要求是：对象被永久保存（除非如同文件系统一样被用户修改）。这里先研究一个“纯”面向对象数据模型：称做ODL（对象定义语言），它已经被ODMG（对象管理组织）标准化。

接下来，将讨论对象关系（Object-Relational）模型。该模型是SQL-99（或叫SQL:1999、SQL3，是SQL的最新标准）的一部分。它扩展了第3章中介绍的关系模型，包括了许多常用的面向对象概念。尽管在具体实现上和提供给用户的用途方面千差万别，但是这个标准是目前许多主要数据库经销商的对象关系DBMS的基础。第9章中讨论了对象关系模型SQL-99。

然后是“半结构化”数据模型。这个模型主要用于处理数据库中存在的一些问题，包括如何将数据库与具有不同模式的其他数据源，如Web页，连接起来。面向对象或对象关系系统的基础是坚持为每个类或关系提供一个固定的模式，而半结构化模型采用了一种灵活得多的模式。以电影对象为例，有些对象会列出导演，有些对象则会列出不同版本的片长，还有些对象会包括评论等等。

XML（eXtensible Markup Language）是半结构化数据模型最重要的一种实现。本质上，XML是“文档”的一种规范，这种文档是嵌套数据（可以包含其他数据集合作为其元素的数据）的集合，其中每个数据可以用一个标记来标识其角色。人们认为，在不同的数据源之间处理或传递数据的系统中，XML数据将成为一个核心的组成部分。此外，XML还可以成为在数据库中灵活地存储数据的一种方法。

4.1 面向对象概念的复习

在介绍面向对象数据库模型之前，先来复习一下面向对象的一些主要概念。面向对象的编程方法已经被广为接受，因为它提供了更好的程序组织方式，从而提高了程序的可靠性。面向对象的编程语言从Smalltalk开始流行，在开发出C++并在C++上移植了许多原先用C实现的软件之后得到了长足发展。最近的Java语言（提供了在WWW上共享程序的一种方式）也是面向对象程序设计语言。

面向对象概念对数据库系统很有吸引力，特别是在数据库设计以及如何扩展关系数据库管理系统（DBMS）的新特性等方面。这一节，将复习一些面向对象的基本概念。

1. 一个功能强大的类型系统；

2. 类（class）是与范围（extent）或属于类的对象集相联系的数据类型。类有一个基本的特征：它可以拥有方法（method）——属于这个类的对象可以使用的过程函数，这个特性使类不同于传统的数据类型；

3. 对象标识（object identity），每个对象都有一个与其具体的值无关的ID；

4. 继承（inheritance），继承将类层次化，每个子类从父类中继承各种特性（包括属性与方法）。

4.1.1 类型系统

面向对象编程语言为用户提供了丰富的类型集合，它预先定义了一些原子类型（atomic type），例如整型、实型、布尔型和字符串类型。用户可以使用类型构建器（type constructor）来创建新类型，一般而言，类型构建器允许用户创建：

1. 记录结构 若给定一个类型序列T_1、T_2 ... T_n以及对应的域名序列f_1、f_2 ... f_n（在Smalltalk
132 中叫做实例变量），用户可以创建一个包含n个属性的记录类型，第i个属性的类型是T_i，可以用其域名f_i来引用。记录类型在C或C++中被称为“结构”，下面将常常用到这个术语；

2. 集合类型 给定一个类型T，人们就可以对T使用集合操作符（collection operator）创建新类型，不同的编程语言使用不同的集合操作符，但是也有一些是共有的，如数组、链表和集合等等。这样如果T是整型原子类型，就可以创建集合类型：“整型数组”、“整型链表”和“整型集合”；

3. 引用类型 对类型T的一个引用是这样一种类型，它可以用来指向该类型的一个特定的值，在C或C++中引用是指向一个值的指针（也就是值的虚拟内存地址）。

显然，可以反复使用记录结构和集合操作符来构造更加复杂的数据类型，例如，一个银行系统可能会定义一个记录结构，其中第一个分量为字符串类型，称为`customer`，第二个分量为整型的集合类型，称为`accounts`，这样的类型就可以适用于将银行的顾客与其账户集联结起来。

4.1.2 类与对象

一个类由一个类型构成，同时可能还有一个或多个函数或过程（两者统称为方法，见下文），其中这些方法可以由类的对象调用。类的对象可以是该类型的值，称为不变对象（immutable object），也可以是该类型的变量，称为可变对象（mutable object）。例如，有一个C类，它的类型为整型集合，这样{2,5,7}就是类C的一个不变对象，而变量s可以被声明为类C的一个可变对象，然后被赋予一个值，如{2,5,7}。

4.1.3 对象标识

对象都有对象标识（object identity，OID）。两个对象不能有相同的OID，并且每个对象只有一个标识。对象标识在建模时的一些作用很有意思。例如，对一个实体而言，往往要求有一个键，这个键一般来自实体的某些属性值（如果是一个弱实体集，则键来自相关实体集的某些属性值）。但是在类中，由于保证了类的OID不会重复，这样就可以区分所有属性值都相等的两个对象。

4.1.4 方法

常常有一些函数与类相关，称为方法（method）。任一个类C的方法都至少有一个参数，
133 即类C的一个对象，当然可能还有其他各种类型（包括类C在内）的参数。例如，包含“整型集合”类型的类，可能需要一个方法来计算集合中所有元素的总和，可能需要一个方法来处理与另一个整型集合的合并，还可能需要一个返回布尔值的方法来判断集合是否为空。

在有些情况下，类被视为“抽象数据类型”，也就是说，类限制了对其对象的访问权限（只有类中定义的方法可以直接修改类的对象），或者说，类实现了封装（encapsulate）。这样就保证了对类的对象的修改不会超出类的设计者设计时所考虑到的情况（这样，对象就始终处于“安全”状态）。封装是开发可靠软件的关键技术。

4.1.5 类的层次

可以将类*C*定义为类*D*的子类（subclass），这样的话，类*C*将继承（inherit）类*D*的所有特性，包括*D*的数据类型以及*D*中定义的函数。此外，*C*还可能包含额外的特性。比如，可以为*C*的对象定义新的方法，它们可以是新增的方法，也可以是取代原先*D*中定义的方法。类*C*还可以扩展原先*D*的类型定义，如果*D*中的数据类型是一个记录结构类型，那么可以为它增加新的属性，这些属性只在*C*的对象中可见。

例4.1 考虑一个银行账户对象的类。它的数据类型定义描述如下：

```
CLASS Account = {accountNo: integer;
                 balance: real;
                 owner: REF Customer;
                }
```

Account（账户）类的数据类型是一个拥有三个字段的记录结构：一个整型的账户号字段，一个实型的余额（balance）字段和一个指向Customer类对象的户主字段（假设类Customer是银行数据库系统中一个已定义的类，这里不对其类型进行介绍）。

人们还可以为这个类定义一些方法，如：

```
deposit(a: Account, m: real)
```

该方法将账户的余额（balance）增加m。

此外可能还要为Account类定义一些子类，如一个定期储蓄的Account类，该类拥有一个额外的字段dueDate（到期时间），这个字段定义了户主可以取钱的时间。假设这个新定义的类叫做TimeDeposit，它是Account类的子类。子类可以有它自己的方法，如定义方法：

```
penalty(a: TimeDeposit)
```

134

这个方法以一个TimeDeposit对象为参数，计算过早提款行为将导致的金钱损失，这个函数将使用到字段dueDate的值和系统的当前日期。 □

4.2 ODL简介

ODL（对象定义语言）是一种用面向对象术语描述数据库结构的标准化语言。它是IDL（接口描述语言）——CORBA（通用对象请求代理）的一个组件——的扩展。CORBA是分布式面向对象计算的一种标准。

4.2.1 面向对象设计

在一个面向对象的设计中，世界被描述成由形形色色的对象构成。对象是某种可见的实体。例如，人可以被看做对象，银行账户也不例外，还有航班，大学的课程，建筑物等等。每个对象都有一个惟一的对象标识（OID），使之区别于其他对象（见4.1.3节的讨论）。

为了组织信息，人们往往要把拥有“相似特性”的对象归成一个对象的类。不过，在讲ODL面向对象设计时，所谓类的对象拥有“相似的特性”，有以下两种含义：

- 类对象在现实世界中表示的概念应当相似。如，可以把一个银行的所有客户归成一类，而将银行的所有账户归成另一类；若把两者混为一类就失去了意义。原因很简单，两者基本上无共同点可言。并且在银行业领域的角色也完全不一样。
- 一个类中的对象的特性应当相同。用面向对象语言编程时，常把对象看做记录。如图4-1所示，对象有存放值的字段，这些字段的类型在不同对象之间应当保持一致，可以同为 135

整型、字符串型、数组型或对另一个对象的引用。

在设计ODL类时，要描述三类特性：

1. 属性（attribute），它是与对象相关的值，在4.2.8节将讨论ODL属性的合法类型；

2. 联系（relationship），它是当前对象与其他对象的连接；

3. 方法（method），它是可以应用于类的对象的函数。

属性、联系、和方法都被作为特性。

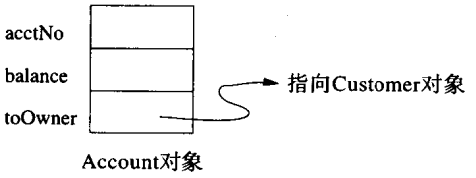


图4-1 一个表示账户的对象

4.2.2 类声明

ODL中，一个最简单的类声明包括：

1. 键class；

2. 类名；

3. 用大括号包含的类的特性列表，这些特性可以是属性、联系或者方法，它们可以按任意次序出现。

也就是说，最简单的类声明形式如下：

```
class <name> {
   <list of properties>
}
```

4.2.3 ODL中的属性

最简单的特性是属性（attribute），通过将固定类型的值与一个对象相联系，属性从一个角度上描述了该对象。例如每个人类对象，都有一个属性名字，其类型为字符串，值为人的名字，人的对象可能还有一个属性出生日期（birthdate），它是一个整型的三元组（一个记录结构），分别代表对象出生的年、月和日。

在ODL中，属性不必为整型、字符串型等简单类型，这一点与E/R模型不同。刚才提到的birthdate就是一个例子。再举一个例子，像电话号码（phone）这样的属性，可以用一组
136 字符串来表示，或者还可以是更复杂的类型，在4.2.8节将总结ODL的类型系统。

例4.2 图4-2中是一个电影类的ODL声明，它并不完整，以后还会加以丰富。第（1）行声明了该类的类名为Movie，接下来是四个属性的声明，它们是所有Movie对象所共有的。

```
1)  class Movie {
2)     attribute string title;
3)     attribute integer year;
4)     attribute integer length;
5)     attribute enum Film {color,blackAndWhite} filmType;
   };
```

图4-2 Movie类的ODL声明

第一个属性名为title（标题），在第（2）行声明，类型为string——长度未定的字符流。假设在Movie对象中该属性的值代表电影的名字。后面两个属性，年份（year）和长度

(length)分别在第(3)、(4)行声明，都为整型，分别代表电影的出品年份和片长(以分钟记)。第(5)行是另一个属性filmType(电影类型)，它用来标识电影是彩色的还是黑白的。其类型为枚举型(enumeration)，枚举类型名为Film，枚举型的属性值从一个文字(literal)列表中选取，本例中是color和blackAndWhite。

到目前为止，一个Movie类的对象可以看做是一个拥有四个分量的记录或元组，每个分量都对应一个属性。如：

```
("Gone With the Wind", 1939, 231, color)
```

就是一个Movie对象。 □

例4.3 在例4.2中，每个属性都是原子类型的，本例将给出非原子类型的属性。定义Star类如下：

```
1)  class Star {
2)      attribute string name;
3)      attribute Struct Addr
            {string street, string city} address;
    };
```

第(2)行声明属性name为字符串型，第(3)行声明了另一个属性address，这个属性的类型是一个记录结构，属性名为Addr，它由两个域构成：street和city，都是字符串型。一般而言，在ODL中，用户可以使用关键字Struct定义一个记录结构，用大括号将其域名以及相应的类型列表括起来。与枚举型一样，结构类型也要有名字，这样其他地方就可以使用此类型。 □ 137

4.2.4 ODL中的联系

尽管通过属性，人们可以知道许多关于对象的信息，有时，对象的主要信息可能在于它与其他对象(相同类的或不同类的)的连接。

为何要为枚举型和记录类型命名?

图4-2的第(5)行中，将枚举型命名为Film似乎没有必要。可是，在给它起了名字之后，人们就可以在Movie类的声明范围之外引用此类型。要引用此类型，就必须提供域的名字(scoped name)Movie::Film。例如，在声明camera(照相机)类时，可以用以下代码：

```
attribute Movie::Film uses;
```

该行声明了属性uses，它使用了Movie类中同样的枚举型，它的值为color和blackAndWhite。

命名枚举型(对于记录类型也一样，它与枚举型声明的方式相同)的另一个原因是：命名了该类型之后，人们就可以在任何特定类之外的一个“模块”中声明类型，这样该模块中的所有类都可以使用被声明的类型。

例4.4 现在，假设要给例4.2中的Movie类增加一个属性：一组影星。更精确地说，是要求每个Movie对象都与一组Star对象(该电影中的影星)相连接。表达两个类之间联系的最好的方式就是使用一个联系。可以用以下代码行实现：

```
relationship Set<Star> stars;
```

该行是在Movie类的声明中，可以加在图4-2中第（1）到第（5）行中任意一行之后。这样就可以说，在每个Movie类的对象中，都有一组对Star对象的引用。这组引用命名为stars。前面的关键字relationship表示stars包含对其他对象的引用，而<Star>前面的Set关键字表示stars包含了一组（而不是一个）Star对象的引用。一般地，一组*T*类型元素
138 在ODL中的定义方式为：Set < T >。 □

4.2.5 反向联系

上面的联系提供了一种方法，使得人们可以从一部给定的电影中求出在该部影片中演出的影星。同样，有时也希望能知道某个特定的影星演过的影片。要在Star对象中加入这些信息，可以将以下代码：

```
relationship Set<Movie> starredIn;
```

加入Star类的声明中（见例4.3）。但是这种方式忽略了Movie对象与Star对象之间联系的一个重要方面。如果一位影星*S*在电影*M*的stars集合中，那么理所应当地，*M*也应该在*S*的starredIn集合中。要表示这种联系就要在其声明中加上关键字inverse，以及其反向联系的名字。如果反向联系定义在其他类中（通常情况如此），则在引用时，就要在联系名之前加上它所在类的名字以及“::”符号。

例4.5 要将Star类的联系starredIn定义为Movie类中stars的反向联系，修改这些类的声明，见图4-3（其中还包含一个Studio类，以后将讨论到），第（6）行代码是Movie类的联系stars的声明，并且注明了其反向联系为Star::starredIn。由于联系starredIn在另一个类中定义，联系名前加了该类名以及“::”。读者可以回忆一下，当要引用定义在其他类中的某些事物时（如特征和类型名，等等），总要使用符号“::”。

类似地，联系starredIn在第（11）行声明；其反向联系则被定义为Movie类的联系stars。因为反向联系必须成对出现，所以如果在Movie类中定义stars联系时注明其反向联系为starredIn，那么也就决定了Star类中的starredIn联系的反向联系必然是Movie::stars。 □

总的规则如下：如果类*C*的联系*R*将类*C*的*x*对象与类*D*的y_1、y_2、…、y_n对象联结起来，那么*R*的反向联系将y_i与*x*联系起来（可能同时还与其他对象相联系）。有时将类*C*到类*D*的联系*R*形象地表示成关系的元素对的列表（或元组集），这种思想与2.1.5节讨论的、用于描述E/R模型的“联系集”很相似。其中每对元素由一个类*C*的*x*对象和与其相关的类*D*的y对象构成。如下表：

C	*D*
x_1	y_1
x_2	y_2
…	…

139 这样*R*反向联系就是相反的元素对的集合，如下表：

D	*C*
y_1	x_1
y_2	x_2
…	…

注意，当类*C*与类*D*相同时，以上规则一样有效。有一些联系，逻辑上将一个类与其自身相联系。例如联系“child of”（是…的子女）就是Persons（人）类和它自身之间存在的

联系。 □

```
1)  class Movie {
2)     attribute string title;
3)     attribute integer year;
4)     attribute integer length;
5)     attribute enum Film {color,blackAndWhite} filmType;
6)     relationship Set<Star> stars
                    inverse Star::starredIn;
7)     relationship Studio ownedBy
                    inverse Studio::owns;
    };

8)  class Star {
9)     attribute string name;
10)    attribute Struct Addr
           {string street, string city} address;
11)    relationship Set<Movie> starredIn
                    inverse Movie::stars;
    };

12) class Studio {
13)    attribute string name;
14)    attribute string address;
15)    relationship Set<Movie> owns
                    inverse Movie::ownedBy;
    };
```

图4-3 ODL的类及它们之间的联系

4.2.6 联系的多重性

像E/R模型中的二元联系一样，ODL中，一对互为反向的联系也可以被分为：多对多、多对一、一对多以及一对一，具体是哪一种，可以看联系的类型声明。 140

1. 若类*C*与类*D*之间有一个多对多的联系，那么类*C*中相应联系的类型应当为Set<*D*>，而*D*中相应联系的类型则为Set< *C* >[⊖]；

2. 如果联系是类*C*到类*D*的多对一联系，那么在*C*中相应联系的类型是*D*，而在*D*中的相应联系的类型是Set< *C* >；

3. 如果联系是类*D*到类*C*的多对一的联系，那么联系的类型正好与2中的相反；

4. 如果联系是一对一的，那么类*C*中相应联系的类型为*D*，而类*D*中相应联系的类型为*C*。

注意，像在E/R模型一样，这里允许多对一或一对一联系包括以下情况：即对某些对象而言联系中的“一”，事实上是“零”。例如，一个*C*到*D*的多对一联系中，某些类*C*对象中的联系值为空（null），因为*D*可以与任意的*C*对象的集合相联结，当然这个集合也可以为空。

例4.6 图4-3定义了三个类：Movie、Star和Studio。前两个已经在例4.2和例4.3中介绍过了。另外还讨论了联系对stars和starredIn。例子中，联系对的类型声明中都使用了Set，可见这是一个多对多联系。

制片厂（Studio）对象有属性name和address，见第（13）和（14）行。注意，这里address类型为字符串型，而不是第（10）行Star类的address属性所用的结构类型。在不

⊖ 这里的Set可以用其他的“集合类型”代替，如链表、包，这些将在4.2.8中讨论，在讨论关系时只用Set作为代表。

同的类中定义具有不同类型的相同名字属性是合法的。

第（7）行定义了一个从Movie类到Studio类的联系ownedBy，由于联系的类型是Studio而不是Set<Studio>，所以，每部影片仅属于一个制片厂。该联系的反向联系见第（15）行，定义了Studio类到Movie类的联系owns。它的的类型为Set<Movie>，这就意味着每个制片厂拥有一个电影集合——可能是0部、1部，或者许多部。 □

4.2.7 ODL中的方法

ODL类中，第三种特性是方法。与其他面向对象语言一样，方法是一段可执行代码，可以由类的对象调用。

在ODL中，人们可以在声明中指定类的方法的名字及其输入输出参数的类型。这些声明，
141 称为签名（signature），就好像C或C++中的函数声明（definition）（不是函数的代码实现——这称为函数的定义）。这些方法的代码将由宿主语言（host language）实现，这不在ODL的范畴之内。

为何要提供签名？

提供签名的意义在于：当用具体的编程语言实现设计的模型时，可以自动地检验实现是否与设计模型相一致。尽管无法从语义上检验操作的实现，但至少可以检验输入与输出参数的个数以及类型是否正确。

方法的声明与属性以及联系一起出现在类声明中。与面向对象编程语言一样，每个方法属于某个类，并且由该类的对象调用。因此类对象是方法的隐含参数。这种方式就使得不同类中使用相同的方法名成为可能，因为调用方法的对象决定了具体是调用哪一个方法。这种情况下，就说方法被重载（overloaded）了。

方法声明的语法与C语言中的函数声明一样，但有两点重要补充：

1. 方法的参数由in、out或inout确定，分别表示该参数是输入参数、输出参数或者输入输出参数。其中，后两种参数类型可以在方法中被改变；而in参数不能。out与inout参数是通过引用传递，而输入参数则是通过值传递。注意，方法还可以有返回值，这是方法返回结果的另一种方式（除了把值分配给out或inout参数两种方式之外）。

2. 方法可能会引起异常（exception），这是与方法通信的一种特殊情况（与一般的传参、值返回方式相对而言）。异常通常表示一种不正常或未预料到的情况，将由调用该方法的某个方法来“处理”（可能间接地通过一系列的调用）。被零除是一种可能导致异常的例子。在ODL中，一个方法的声明之后可以加上关键字raises，再在其后加上由括号括起来的一个或多个该方法可能引起的异常。

例4.7 图4-4给出了Movie类定义的更新版本，它是在图4-3基础上的改进。下图是带方法
142 声明的类声明。

第（8）行声明了方法lengthInHours（以小时计的片长），顾名思义，该方法返回调用该方法的电影对象的片长。其处理办法是将以分钟为单位的属性length换算成一个以小时为单位的浮点数。该方法的声明中没有参数，但是不要忘记，类的方法始终有一个隐含的参数，即调用该方法的Movie对象。只有提供了这个参数，方法才可能知道到底要换算哪个电影对象的length属性[⊖]。

⊖ 在方法lengthInHours的实际定义中将使用一个特殊的关键字（如self），来引用调用方法的对象。但在方法的声明中不必关心它。

```
1)  class Movie {
2)     attribute string title;
3)     attribute integer year;
4)     attribute integer length;
5)     attribute enumeration(color,blackAndWhite) filmType;
6)     relationship Set<Star> stars
                  inverse Star::starredIn;
7)     relationship Studio ownedBy
                  inverse Studio::owns;
8)     float lengthInHours() raises(noLengthFound);
9)     void starNames(out Set<String>);
10)    void otherMovies(in Star, out Set<Movie>)
                  raises(noSuchStar);
    };
```

图4-4 为Movie类增加方法签名

方法lengthInHours可能会导致一个叫做noLengthFound的异常。假设当调用该方法的Movie对象的length值无意义（比如，是一个负数）或未定义时，会引起这个异常。

第（9）行可以看到另一个方法签名，方法名为starNames。这个方法没有返回值，只有一个输出参数，其类型为一个字符串集合。假设输出参数的值被方法starNames设定为对应电影对象的影星的名字集。当然，也不保证该方法一定是用来执行这个操作的。

最后在第（10）行定义了otherMovies方法。这个方法有一个输入参数。该方法的一个可能的操作如下：假设该方法认为输入参数代表一位参与该电影的影星（如果不是，那么将导致一个noSuchStar的异常）。如果该影星出演了该电影，那么输出参数（类型为电影集合） 143
将被赋值为该影星演过的所有影片的集合。 □

4.2.8 ODL中的类型

ODL为数据库设计者提供了一个类似于*C*或其他传统编程语言的类型系统。一个类型系统通过一些本身提供的基本类型以及一些递归的规则（由这些规则可以构造出很复杂的类型）构建起来。ODL类型系统的基本类型有：

1. *原子类型*。含有整型、浮点型、字符型、字符串型、布尔型和枚举型（是一张文字列表，其中每个文字被赋予一个抽象的值）。图4-3的第（5）行就是一个枚举型的例子，其文字包括color和blackAndWhite。

2. *类名*。如Movie、Star等。此类型实际上是一个包含类的所有属性和联系的结构。

可以使用下列*类型构建器*（type constructor）将基本类型组合成结构化的类型：

1. *集合*（set）。设*T*为任意类型，则Set<T>表示的类型是类型为*T*的元素的一个有限集合，Set类型构建器的例子见图4-3的第（6）、（11）以及（15）行。

2. *包*（bag）。设*T*为任意类型，则Bag<T>表示类型为*T*的元素的一个有限包，或有限多集(multiset)，它允许相同的元素在集合中出现多次。例如，{1, 2, 1}是一个多集，但不是一个集合，因为1在其中出现了不止一次。

3. *链表*（list）。设*T*为任意类型，则List<T>表示一个类型为*T*的元素的有限长链表（链表中元素个数可以为0）。一个特殊的例子：类型string是List<char>的简写形式。

4. *数组*（array）。设*T*为任意类型，*i*是一个整数，则Array< T, i>表示的类型是长度为*i*的类型为*T*的元素的数组。如Array<char, 10>表示一个长度为10的字符串。

5. *字典*（dictionary）。设*T*，*S*为任意类型，则Dictionary< T, S >表示的是*T*，*S*元素

对的有限集合。其中每对元素由一个键类型（key type）T的值和一个值域类型（range type）S的值构成。字典中不能存在两对键类型值一样的元素对。字典一般都假设是用很高效的方式实现了从一个给定的键类型值t，找到与之对应的值域类型值。

144 6. 结构（structure）。若T_1，T_2，...，T_n是类型，F_1，F_2，...，F_n分别是其域名，那么：

```
Struct N {T1 F1, T2 F2,..., Tn Fn}
```

表示一个名为N的结构类型，它包含n个域，其第i个域名为F_i，类型为T_i。例如图4-3的第（10）行定义了一个名为Addr的结构类型，它有两个域，均为string类型，名字分别是street和city。

集合，包和链表的区别

集合是无序的，每个元素至多出现一次。包也是无序的，但它允许元素出现多次。而链表是有序的，它也允许一个元素出现多次。所以{1, 2, 1}，{2, 1, 1}是同样的包，但（1, 2, 1），（2, 1, 1）是不同的链表。

前五种类型：集合、包、链表、数组和字典统称为集合类型（collection type）。至于哪些类型可以用于声明属性，哪些可以用于声明联系，有不同的规则：

- 联系的类型要么是一个类要么是应用于一个类的集合类型构建器（单独使用一种集合类型构建器）。
- 属性的类型先由原子类型构造。也可以使用类，但一般而言，这些类与“结构”差不多，如图4-3中的Addr结构。人们一般在联系中使用类，因为联系总是双向的，这样就简化了数据库中的查询。而对属性而言，只能通过对象找到其属性，而无法反过来通过属性找到其所属对象。进行原子类型或类构造后，可以没有次数限制地重复使用结构和集合类型构建器。

例4.8 属性可能取的一些类型是：

```
1. integer.
2. Struct N {string field1, integer field2}.
3. List<real>.
```

145 4. `Array<Struct N {string field1, integer field2}, 10>.`

例子中，（1）是原子类型，（2）是结构类型，（3）是原子类型的集合类型，（4）是一个由原子类型构成的结构的集合类型。

现在假设Movie和Star是可用的基本类型的类名，然后可以创建诸如Movie、Bag<Star>等等的联系类型。可是，如下是一些非法的联系类型：

1. Struct N{ Movie field1, Star field2 }。联系不能使用结构类型；
2. Set<Integer>。联系不能使用原子类型；
3. Set<Array<Star, 10>>。联系类型中不能含有多种集合类型。

□

4.2.9 习题

* **习题4.2.1** 习题2.1.1是一个银行数据库系统的非正式描述，现在将它用ODL描述。

习题4.2.2 用习题2.1.2中列举的方式修改习题4.2.1中的设计，说明你做的修改，不必重新

设计。

习题4.2.3 在ODL中实现习题2.1.3中的球队-队员-球迷数据库系统。为何原先习题中很复杂的球队颜色集问题，在ODL中却显得很简单？

*! **习题4.2.4** 假设要保存一本家谱。人们将使用一个Person类，对应每个Person对象，要记录其名字（这是Person类的一个属性）以及以下联系：mother、father和children。给出Person类的一个ODL设计。记住，要指明mother、father和children联系的反向联系（是从Person类到Person类的联系）。mother联系的反向联系是否就是children联系？请给出你的回答以及理由，将每个联系连同其反向联系一道描述成一个联系对集合。

! **习题4.2.5** 为习题4.2.4中的设计增加一个属性education，这个属性用来保存各个Person对象所获得的学位的集合，一个学位信息包括：学位名、所属学校以及获得日期。学位的集合类型可以是Set、Bag、List或Array。解释这四种选择可能导致的设计结果，从保存信息的角度考虑它们各自应如何取舍？舍去的信息在实际中是否很重要？

习题4.2.6 图4-5是用ODL定义的Ship类和TG（任务组，由一组Ship构成）类，要求对其作些修改（可以将要修改的行标出，然后给出修改后的行，或者在原先的基础上增加一些新行）：

146

a) 属性commander（指挥官）改为由一对字符串类型组成的结构，前者是其级别，后者是其名字；

b) 一个Ship对象可以出现在多个任务组中；

c) *姊妹船*（sister ship）是指设计方案相同的不同船只，为每艘船增加姊妹船的集合（不包括自身），假设姊妹船都是Ship对象。

```
1)  class Ship {
2)      attribute string name;
3)      attribute integer yearLaunched;
4)      relationship TG assignedTo inverse TG::unitsOf;
    };

5)  class TG {
6)      attribute real number;
7)      attribute string commander;
8)      relationship Set<Ship> unitsOf
            inverse Ship::assignedTo;
    };
```

图4-5 船与任务组的一个ODL描述

*!! **习题4.2.7** 在什么情况下一个联系是自反的？*提示*：将联系看成是对象对的集合（就像4.2.5节讨论的那样）。

4.3 ODL中的其他概念

要在ODL中描述所有能在E/R模型或关系模型中描述的内容的话，还需要了解ODL中的一些其他特性。这一节将讨论：

1. 多路联系的表示。ODL中所有的联系都是二元的，因此，必须进一步讨论三路及多路联系的ODL表示，而这些在E/R图和关系中可以很简单地将它们表示出来。

2. 子类和继承。

147

3. 键。在ODL中这是可选项。

4. 范围（extent），一个给定数据库中类的对象集合。这是ODL中等价于实体集或关系的概念。不要将其与类本身混淆，类是一种模式。

4.3.1 ODL中的多路联系

ODL仅仅支持二元联系。有一个技巧（在2.1.7节介绍），可以将一个多路联系转化成几个二元的多对一联系。假设在类（或实体）集合C_1，C_2，...，C_n之间有一个多元联系R，可以将R替换为一个类C和n个多对一的二元联系（从类C到类C_i）。类C的每个对象可以被看做R联系集中的一个元组t，而t对象通过n个多对一的联系与类C_i相联系。

例4.9 考虑如何在ODL中表示三元联系contracts（合同），其E/R图在图2-7中给出。这里先定义类Movie、Star和Studio，这三个类由图4-3中的Contracts联系。

必须创建类Contract与这个三路联系Contracts相对应。则三个多对一的联系（从Contract类到其他三个类）分别为theMovie、theStar和theStudio，见图4-6：

```
1)    class Contract {
2)        attribute integer salary;
3)        relationship Movie theMovie
              inverse ... ;
4)        relationship Star theStar
              inverse ... ;
5)        relationship Studio theStudio
              inverse ... ;
      };
```

图4-6 用类Contract描述三路联系Contracts

Contract类有一个属性：工资，因为它仅由合同、而不是合同三方中的任一方决定。回忆一下，在图2-7中曾类似地将属性salary放在联系Contracts中而不是任何一个与合同相关的实体集中。Contract对象的其余特性就是刚才提到的三个联系。

注意，至此尚未命名上述联系的反向联系。这先要修改Movie、Star和Studio类的声明，
148 增加与Contract类中的联系互为相反的联系。例如theMovie的反向联系可以叫做contractsFor，这样图4-6的第（3）行可以换作：

```
3)  relationship Movie theMovie
        inverse Movie::contractsFor;
```

并且在Movie的声明中增加：

```
relationship Set<Contract> contractsFor
    inverse Contract::theMovie;
```

注意，Movie类中的联系contractsFor的类型是Set<Contract>，这是因为每部影片需要多份合同。每份合同本质上是一个由一部电影、一位影星以及一个制片厂构成的三元组，另外再加上制片厂支付给影星的工资这个属性。 □

4.3.2 ODL中的子类

回忆一下，在2.1.11节E/R模型中讨论过子类。ODL中同样允许将一个类C定义为另一个类D的子类。只要在类C的声明之后，加上关键字extends，再加上类名D即可。

例4.10 例2.10中将卡通片类定义为电影类的子类，该类与父类相比有一个额外的特性：一个与一组配音影星之间构成的联系，可以用ODL将卡通片类声明如下：

```
class Cartoon extends Movie {
   relationship Set<Star> voices;
};
```

这里没有给出联系voices的反向联系，但实际上必须给出。

子类继承父类的所有特性，所以每个卡通片对象都从Movie类继承title、year、length和filmType（回忆图4-3）以及联系stars和ownedBy。

同样，在本例中，再定义一个凶杀片（murder mystery）类，该类有属性weapon：

```
class MurderMystery extends Movie {
   attribute string weapon;
};
```

这是一个合法的子类定义，Movie类的所有特性都被该类继承。 □ 149

4.3.3 ODL中的多继承

有时候，在将像《兔子罗杰》这样的影片归类时，人们可能需要另外一个新的类，它同时是两个或者更多类的子类。在E/R模型中，可以用三个实体集Movies、Cartoons和MurderMysteries的结合（这三个实体集由一个“isa”层次连接起来）来表示《兔子罗杰》。而对面向对象系统而言，这是行不通的，该系统的一条基本的原则就是一个对象属于并且仅仅属于一个类，因而，要描述一部既是卡通片，又是凶杀片的电影，就还要定义一个新类。

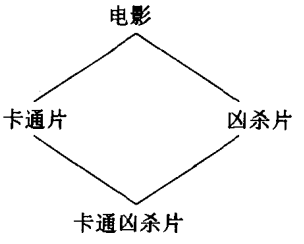


图4-7 多继承的图示

类卡通凶杀片（CartoonMurderMystery）必须同时继承Cartoon类和MurderMystery类，如图4-7所示。也就是说，一个CartoonMurderMystery类的对象拥有Movie类对象的所有特性，此外，还拥有联系voices和属性weapon。

ODL中，可以在extends关键字之后加上多个类名，类名之间用“:”分开[⊖]，因而新类的定义如下：

```
class CartoonMurderMystery
    extends MurderMystery : Cartoon;
```

一个类*C*继承多个父类时，可能导致冲突。*C*的两个或更多的父类可能有同名但不同类型的特性。例子中CartoonMurderMystery类没有这个问题，因为父类Cartoon和MurderMystery同名的特性是Movie类中的特性，它们是一致的。但在下面的例子中会导致这种问题。

例4.11 假设有两个Movie类的子类，分别为浪漫剧（Romance）类和法庭剧（Courtroom）类，另外假设这两个子类都有一个名为ending的属性。在Romance类中，属性ending 是一个枚举型{happy, sad}，而Courtroom类中属性ending是另一个枚举型{guilty, notGuilty}。如果另外再创建一个类Courtroom-Romance，它有两个父类Courtroom和Romance，那么类Courtroom-Romance的属性ending的类型就是不明确的。 □ 150

⊖ 技术实现上，第二个以及后续的名字必须代表“接口”，而不是类。大致说来，ODL中的接口就是一个没有相关对象集（或称为范围）的类声明。在4.3.4节中还将进行进一步讨论。

ODL标准并未规定如何去消除这种冲突，一些可能的解决方式有：

1. 不允许多继承，这种限制方式过于严格；

2. 在子类中指明有效的特性，如例4.11中对于一部Courtroom-Romance类型的电影，人们更感兴趣的可能是其结局是喜剧还是悲剧，而不是法庭的判决到底是有罪还是无罪，这样就可以指定类Courtroom-Romance从父类Romance（而不是从父类Courtroom）继承ending属性。

3. 对与父类中重名的属性，在子类中将其重命名。如例4.11，如果Courtroom-Romance从父类Romance继承ending属性，那么可以指定Courtroom还有另一个属性verdict，它是子类从Courtroom类继承的属性ending重命名而得。

4.3.4 范围

当ODL中类是已定义的数据库系统的一部分时，需要将类的定义与其对象集合区分开。这种区别类似于一个关系模式与一个关系实例之间的区别（尽管在不同的上下文环境中，两者都可以用关系名来指代）。另外在E/R模型中也要区分实体的定义与该实体当前存在的实体集合。

ODL中，这种区别是通过赋予类以及其范围（extent）不同的名字得以实现。因此类名代表类的设计模式，而范围是该类当前存在的实体集合。类的范围的命名方式是用小括号将关键字extent以及紧跟的范围名括起来，加在声明的类名之后。

例4.12 一般地，一个很有效的命名习惯是将类命名为一个单数名词，而将对应的范围命
151 名为名词的复数形式。根据这种习惯，可以将Movie类的范围命名为Movies。为了声明范围的名字，类Movie的声明代码如下：

接 口

ODL提供接口（interface）定义的支持，接口定义实质上是没有定义相关范围（因而也没有相关对象集合）的类定义。4.3.3节中首次提到接口，指出它们可以支持一对多继承。当几个类有不同的范围、但有相同的特性时，接口就有用武之地。就像几个关系拥有相同的设计模式，但元组集不同的情况一样。

定义一个接口*I*之后，就可以定义机构继承*I*特性的类，它们各自拥有不同的范围，这样人们就可以在数据库中保存几个有相同类型但属于不同类的对象集合。

```
class Movie (extent Movies) {
    attribute string title;
        ...
```

以后在学习查询ODL数据的OQL语言时就会看到：当要查找现存于数据库中的电影时，系统处理的是Movie类的范围而不是类本身。 □

4.3.5 ODL中键声明

ODL与到目前为止学习过的模式的差别在于，键的声明和使用是可选的。也就是说，在E/R模型中，实体集合需要用键来区分各个元素。在关系模型中（关系都是集合），所有属性一起共同组成一个键（除非某个适当的属性子集可以担当键），无论是哪一种方式，对于每个关系而言都至少需要一个键。

但是，对象有一个独特的对象标识（4.1.3节讨论过），所以ODL中，声明一个键（或键集合）是可选项。对于一个类而言，可以有所有特性都相等的多个对象，即便如此，仍然可以通过其内部对象标识来区分它们。

ODL中，可以使用保留字key或keys（都一样），后面跟着一个属性或一组属性来定义对象的键。如果一个键中包含多个属性，那么必须将属性列表用括号括起来。键声明紧跟在 152
extent声明之后。

例4.13 为类Movie声明一个由两个属性title和year构成的键：

```
class Movie
    (extent Movies key (title, year))
{
    attribute string title;
        ...
```

也可以用keys代替key，即便键只由一个属性构成。

如果类Star的键是名字name，那么类似地声明如下：

```
class Star
    (extent Stars key name)
{
    attribute string name;
        ...
```

□

可能有多个属性集构成多个键的情形。这样的话，可以将这些属性集加在关键字key（s）后面，并且用逗号分开。一般地，一个由多个属性构成的键在声明中必须将这些属性用括号括起来，这样就可以区分声明的是由多种属性集构成的多个键，还是一个由多个属性构成的键。

例4.14 作为多个属性集构成多个键的例子，考虑类Employee。这里并不将该类的全部信息都罗列出来，仅仅考虑其属性工号（empID）和社会保险号（ssNo），它们中任意一个都可以被声明为键：

```
class Employee
   (extent Employees key empID, ssNo)
      ...
```

因为属性表没有用括号括起来，ODL认为上述声明是把empID和ssNo都声明为键。而假如为属性表加上括号：（empID, ssNo），那么ODL认为，这两个属性共同构成一个键。以下代码：

```
class Employee
   (extent Employees key (empID, ssNo))
      ...
```

153

就意味着，任意两个员工的empID和ssNo的组合都必须独一无二，不过empID或ssNo有可能相等。 □

ODL标准还允许用属性以外的特性来构成键。将一个方法或者联系声明为键（或者键的一部分）原则上不会造成什么问题，因为键是DBMS可以使用（也可不用）的一种辅助性的声明。例如，将一个方法声明为键，这就意味着对于类中不同的对象，该方法的返回值不会相同。

如果允许在键声明中出现多对一联系，那么这就和E/R模型中的弱实体集合有相同的效果。假设对象O_2和对象O_1处于一个多对一的联系中，其中O_2位于“多”的那一边，O_1位于“一”那一边，那么可以将对象O_1声明为O_2中的键（的一部分），对于一个类似于O_2的对象的集合而言，

O_1是独一无二的。但是，应当记住，类不必要有键，这样也就不必去用特别的办法来处理那些缺乏构成键的属性的类，就好像在处理弱实体集合时所做一样。

例4.15 先复习一下图2-20中弱实体集的例子*Crews*。假设摄制组是由他们的成员号以及他们所在的制片厂来标识。虽然两个不同的制片厂可以有相同的摄制组编号，但是可以如图4-8所示那样对Crew类定义。注意，这里需要修改类Studio的声明，让它包含联系crewsOf，该联系是Crew类中的partOf联系的反向联系。这里没有给出其修改内容。

```
class Crew
    (extent Crews key (number, partOf))
{
    attribute integer number;
    relationship Studio partOf
        inverse Studio::crewsOf;
}
```

图4-8 联系crews的ODL声明

这里的键声明意味着不存在两个摄制组有相同的摄制组编号，又在同一个制片厂工作。请注意，这个声明与图2-20中的E/R图很相似，该图中，一个摄制组实体由一个编号与相关制片
154 厂的名字决定。 □

4.3.6 习题

* **习题4.3.1** 给习题4.2.1中的ODL设计增加范围和键；

习题4.3.2 给习题4.2.3中的ODL设计增加范围和键；

! **习题4.3.3** 假设要修改习题4.2.4中的ODL声明，原习题中声明了一个关于人的类和一组联系mother-of，father-of和child-of，使这个类包含特定的子类：（1）男性；（2）女性；（3）为父母的。此外，我们希望联系mother-of，father-of和child-of存在于（并且仅仅存在于）尽可能小的类中，为此你可能要定义其他的子类，写出其声明，在合适的地方请使用多继承。

习题4.3.4 图4-6中的Contract类有没有合适的键？有的话，是什么？

习题4.3.5 习题2.4.4中有两个例子，其中的弱实体集是必要的。使用ODL实现这些数据库，包括范围和键的声明。

习题4.3.6 用ODL设计习题2.1.9中的注册数据库。

4.4 从ODL设计到关系设计

E/R模型一般在数据库的具体实现时需要被转换为另一种模型，比如关系模型，而ODL则主要是用于真实的、面向对象的DBMS系统的描述语言。但是ODL就像所有面向对象的设计系统一样，也可以用于初步设计，然后在具体实现之前被转成关系。这一节将考虑怎样将ODL设计转化为关系设计。这个过程在很多方面与在3.2节介绍的、将E/R图表转化为关系数据库模式的方法相似。不过，对于ODL而言，会存在一些新的问题，包括：

1. 实体集一定要有键，但在ODL中没有这种要求。因此，有些情况下在为类创建关系时，必须自创一个属性作为键。

2. E/R属性和关系属性都要求必须是原子类型，但ODL没有这样的限制。集合类型的属性转化为关系的方法具有技巧性，并且常常产生不规范的关系，并为此不得不使用3.6节介绍的
155 办法重新设计。

3. ODL允许声明方法作为设计的一部分，但是不存在简单的办法将方法直接转化为一个关系模式。读者可以参见4.5.5节，以及第9章。现在假设所有要转化为关系设计的ODL设计中都不包含方法。

4.4.1 从ODL属性到关系属性

作为开始，先假设每个类对应一个关系，而类的每个特性都对应关系的一个属性。后面将看到，这种假设在很多方面需要改进。但暂时先只考虑这种最简单的情况，在这种情况下确实可以做到类被转化为与关系对应，特性可以转为属性。假设的限制如下：

1. 类中的所有特性都是属性（而不是联系或者方法）；

2. 属性的类型都是原子类型（而不是结构或集合）。

例4.16 图4-9就是这样一个类，它有四个属性，每个属性都是原子类型的：title的类型是字符串型，year和length是整型，filmType是一个枚举型（包含两个值）。

```
class Movie (extent Movies) {
   attribute string title;
   attribute integer year;
   attribute integer length;
   attribute enum Film {color,blackAndWhite} filmType;
};
```

图4-9 类Movie的属性

例中创建了一个与该类的范围名一样的关系名：Movies；这个关系有四个属性，各对应于类中的每个属性。关系的属性名与类中对应的属性名相同，因此该关系的模式为： 156

```
Movies(title, year, length, filmType)
```

Movies范围中每个对象都对应于关系Movies中的一个元组，元组中的每个部分都对应于类对象的一个属性（并且有相同的值）。 □

4.4.2 类中的非原子类型属性

但是，即便是只有属性的类，也很难将类转化为关系。原因是，ODL的类属性可以是复杂类型，诸如结构、集合、包和列表等等。而关系模型的一个基本要求是：属性必须是整型、字符串型等原子类型。所以必须设法在关系中表示非原子类型的属性。

类型是原子类型的记录结构比较容易处理。只要扩展结构的定义，为结构的每个字段定义一个关系的属性即可。这里惟一可能出现的问题是，在一个类中的两个结构类型有相同名字的字段，如果是这样，重新选择一个属性名即可解决这个问题。

```
class Star (extent Stars) {
   attribute string name;
   attribute Struct Addr
       {string street, string city} address;
};
```

图4-10 有结构类型的属性的类

例4.17 图4-10是类Star的声明，该类的特性都是属性。其中name属性是原子类型的，但是address属性是一个有两个字段（street和city）的结构类型。对于这个类，可以使用一个包含三个属性的关系来替代。第一个属性命名为name，对应Star类中的同名属性，第二和第三个属性分别命名为street和city，分别对应于Address结构类型的两个同名的字

段。至此就得到关系的设计模式：

```
Stars(name, street, city)
```

图4-11给出了这个关系的一些典型元组的例子。 □

name	*street*	*city*
Carrie Fisher	123 Maple St.	Hollywood
Mark Hamill	456 Oak Rd.	Brentwood
Harrison Ford	789 Palm Dr.	Beverly Hills

157

图4-11 表示影星的关系

4.4.3 集合类型属性的表示

记录结构还不是ODL类声明中类型最复杂的属性。在4.2.8节中提到，类型还可以使用类型构建器Set、Bag、List、Array和Dictionary来构造。在将其转化为关系模型时，各自有各自的问题。这里只对最常用的Set进行详细讨论。

一种处理办法是，对于属性*A*的值集合，为集合中的每个值都构造一个元组，该元组中包括属性*A*和类中的所有其他属性的相应值。下面先来看一个用这种办法可以处理的例子，然后再看这个办法的问题所在。

```
class Star (extent Stars) {
   attribute string name;
   attribute Set<
          Struct Addr {string street, string city}
      > address;
};
```

图4-12 有一组地址的影星

例4.18 假设类Star如图4-12所示被定义，每个Star对象都有一个地址的集合。另外假设Carrie Fisher另外拥有一栋海滩别墅。而图4-11中的其他两位影星都只有一个住处。于是将为名字（name属性）“Carrie Fisher”创建两个元组如图4-13所示，其他元组不变。 □

name	*street*	*city*
Carrie Fisher	123 Maple St.	Hollywood
Carrie Fisher	5 Locust Ln.	Malibu
Mark Hamill	456 Oak Rd.	Brentwood
Harrison Ford	789 Palm Dr.	Beverly Hills

图4-13 允许地址集合的例子

但是，这种将拥有一个或者多个值集合属性的对象替换为元组集合的技巧可能会导致不规范的关系（3.6节讨论过的）。事实上，即便只有一个值集合类型的属性也可能会导致违反BCNF规则。见下一个例子。

158

```
class Star (extent Stars) {
   attribute string name;
   attribute Set<
          Struct Addr {string street, string city}
      > address;
   attribute Date birthdate;
};
```

图4-14 拥有一个地址值集合属性以及一个生日属性的Star对象

原子类型的值：是特点还是不足？

ODL允许使用结构类型，这种灵活的表示方法使得关系模型显得碍手碍脚。有人可能想把关系模型彻底放弃（至多把它作为一种被淘汰了的思想），而只采用面向对象的ODL。但是，现状是关系型数据库至今为止仍然在市场上占主导地位。一个重要的原因是这种模型的简单性使得其查询语言的功能十分强大，特别是SQL（见第6章），它是现在数据库系统中的查询语言标准。

例4.19 假设在图4-14例子的基础上增加一个birthdate属性，其类型Date是ODL的一个原子类型。于是Birthdate属性可以是Star关系的一个属性。如图4-12所示设计模型变成：

```
Stars(name, street, city, birthdate)
```

若把图4-13的数据修改一下（因为地址集属性可以为空，例如假设Harrison Ford没有地址信息），得到图4-15的结果。这里出了两个问题：

1. Carrie Fisher的出生属性重复，导致了信息的冗余。（尽管名字也出现了两次，但这不是冗余，没有名字信息就无法知道具体的地址信息是关于哪位影星）；

2. 由于Harrison Ford没有地址信息，也就失去他的其他信息。这是在3.6.1节讨论过的删除异常的一个例子。

159

name	*street*	*city*	*birthdate*
Carrie Fisher	123 Maple St.	Hollywood	9/9/99
Carrie Fisher	5 Locust Ln.	Malibu	9/9/99
Mark Hamill	456 Oak Rd.	Brentwood	8/8/88

图4-15 增加属性birthdate之后的元组

尽管name属性是类Star的一个键，但是由于对于每个影星，要求使用多个元组来表示其所有的住址，这就使得在关系Stars中，name不是一个键，事实上该关系的键是 { name, street, city }，因此函数依赖

$$\mathbf{name} \rightarrow \mathbf{birthdate}$$

违反了BCNF条件，这也就是为什么会出现以上异常的原因。 □

至于如何处理在类声明中出现的集合类型，有几种选择。首先，可以把所有的属性（不管是不是集合类型）都放在关系的设计模式中；然后，使用3.6节和3.7节提到的一般化方法来消除第一步导致的对BCNF和4NF条件的违反。注意，一个集合类型的属性与一个单值属性的联合将导致违反BCNF条件，见例4.19。在同一个类声明中的两个集合类型的属性将导致违反4NF条件。

另一种方式是把所有的集合类型的属性分离出来，将这些属性的值域与类的对象集合之间的关系看做是一个“多对多”的联系。对此将在4.4.5节讨论联系时使用这个方法。

4.4.4 其他类型构建器的表示

除了记录结构和集合，一个ODL类定义还可以使用Bag、List、Array或者Dictionary来构建类型。为了要用关系模型表示一个相同的元素可以出现n次的包（多集），不能简单地将这种情况处理成n个完全相同的元组[⊖]。另外也可以在设计模式中增加一个count属性用于记

⊖ 精确地讲，在第3章所介绍的抽象的关系模型中我们无法引入完全相同的元组。不过在基于SQL的关系型DBMS中这是可以的。也就是说，SQL中关系是包而不是集合。参见5.3节和6.4节。假如查询中有可能要求得到元组的个数，我们建议还是使用这里描述的设计模式（不管你的DBMS是否允许重复元组）。

录包中每个元素出现的次数。例如假设图4-12中的address是一个包类型（而不是集合类型）
160 的属性，那么可以通过如下这样的元组集合：

name	*street*	*city*	*count*
Carrie Fisher	123 Maple St.	Hollywood	2
Carrie Fisher	5 Locust Ln.	Malibu	3

表示123 Maple St., Hollywood是Carrie Fisher的“二次”地址，5 Locust Ln., Malibu 是Carrie Fisher的“三次”地址，而不管这到底意味着什么。

如果地址是一个列表类型（而不是刚才提到的那两种类型），那么我们可以使用一个新属性position，该属性用于指示对应的地址在列表中的位置。例如，Carie fisher的地址作为列表，Hollywood是第一个：

name	*street*	*city*	*position*
Carrie Fisher	123 Maple St.	Hollywood	1
Carrie Fisher	5 Locust Ln.	Malibu	2

如果地址是定长数组，则可以用数组中的位置表示。例如地址是数组类型，结构为两个城市街道，可以把Star对象用如下形式表示：

name	*street1*	*city1*	*street2*	*city2*
Carrie Fisher	123 Maple St.	Hollywood	5 Locust Ln.	Malibu

最后，字典（dictionary）可以表示成一个由键域和值域组成的二元组的集合。例如，对于每个影星，除了地址外，若还要保存其各个房产的抵押契据持有者。那么，该字典类型就以地址作为其键域，而房产的抵押契据持有者银行就构成其值域。下面的图表是一个例子：

name	*street*	*city*	*mortgage-holder*
Carrie Fisher	123 Maple St.	Hollywood	Bank of Burbank
Carrie Fisher	5 Locust Ln.	Malibu	Torrance Trust

当然，ODL中属性的类型可能是由多重类型构建器定义而成。如果一个类型是在记录类型的基础上使用集合类型构建器（字典类型除外）定义而成的（也就是说，一组结构），那么使用4.4.3节或者4.4.4节介绍的技巧，可以首先将记录类型看做原子类型，然后再将结构类型展开成多个属性。上面例子中，已经使用了这个技巧（原先address属性是一个结构类型，例子中已经把它展开成了两个属性street和city）。字典类型的情况类似，留作习题。

很多情况下应当把属性类型的复杂度控制在一定程度（在一个结构声明和一个集合类型构建器所能构造的范围之内）。2.1.1节中提到过，尽管E/R模型要求每个属性都是原子类型，然而
161 在某些E/R模型的实际实现中，类型定义的复杂度有所扩展，不过还是被限制在这个范围以内。有一个建议，就是在使用ODL对关系数据库模式作设计时，尽可能地限制自己不要使用太多的特性。在习题中将考虑在一些属性类型更加复杂的情况下，如何选择处理方式。

4.4.5 ODL中联系的表示

一般地，ODL中的类会包含与其他类之间的联系。这就像E/R模型中，可以为每一个联系创建一个新关系，该关系连接两个相关类的键。不过在ODL中，联系互为相反地成对出现，对于每对联系，只需建立一个关系。

```
class Movie
    (extent Movies key(title, year)
{
    attribute string title;
    attribute integer year;
    attribute integer length;
    attribute enum Film {color,blackAndWhite} filmType;
    relationship Set<Star> stars
                inverse Star::starredIn;
    relationship Studio ownedBy
                inverse Studio::owns;
};

class Studio
    (extent Studios key name)
{
    attribute string name;
    attribute string address;
    relationship Set<Movie> owns
                inverse Movie::ownedBy;
};
```

图4-16 Movie类与Studio类的完全定义

例4.20 考虑图4-16中重新给出的类Movie和Studio的声明。可以看到title和year属性构成了Movie类的键，而name属性构成了Studio类的键。可以为联系对owns和ownedBy创建一个关系。关系需要一个名字，在此把它叫做StudioOf。该关系的设计模式中包括Movie类的键（title和year）和Studio类的键name，但在设计模式里name被改名为studioName。于是，关系的设计模式如下：

162

```
StudioOf(title, year, studioName)
```

下面是该关系的一些典型的元组例子：

title	*year*	*studioName*
Star Wars	1977	Fox
Mighty Ducks	1991	Disney
Wayne's World	1992	Paramount

□

当一个联系的类型是多对一时，可以选择将该联系与联系中“多”的那一方的类建立一个关系。这样做的效果是，组合了两个拥有共同键的关系（这在3.2.3节讨论过）。这样就不会导致破坏BCNF条件，因而也就是一个合法的、并且常用的选择。

例4.21 如果不为联系对owns和ownedBy创建一个关系StudioOf（像例4.20中做的那样）的话可以修改设计模式中关系Movies的内容，在其中增加一个studioName属性来表示Studio中的键。如果这样，Movies的设计模式就变成

```
Movies(title, year, length, filmType, studioName)
```

一些典型的元组例子：

title	*year*	*length*	*filmType*	*studioName*
Star Wars	1977	124	color	Fox
Mighty Ducks	1991	104	color	Disney
Wayne's World	1992	95	color	Paramount

注意，Movie类的键title和year也是关系Movies的键，因为每一部电影的长度、类型和制片厂具有惟一性。 □

应该记得，尽管处理多对多联系时，可以使用例4.21中处理多对一联系时的方式，但这并不明智。事实上，3.2.3节的例3.6中考虑了，如果stars联系是电影与影星之间的多对多联系，那么当为这个联系与关系Movies中的其他信息建立一个关系时，会出现什么问题。这种方式下的模式如下：

```
Movies(title, year, length, filmType, studioName, starName)
```

由于{ title, year, starName }是键，而属性length、filmType和studioName都仅仅只由title和year惟一确定，因此，该模式违反了BCNF条件。

同样，如果真的将多对一联系与一个类所转化后的关系组合，那么该类必然是属于联系中
163 “多”的那一方。例如，用关系Studios将owns和ownedBy组合起来违反BCNF条件(见习题4.4.4)。

4.4.6 如果没有键会怎样?

因为ODL中的键是一个可选项，所以也有可能会面对这样一种情况：即不存在可以充当键的属性。当类*C*是某个或者某些联系的组成部分时，可能就会有这样的问题出现。

这里建议创建一个新的属性（或者叫“证书”），用于在关系设计中标识类*C*的对象。这很像一个隐藏的面向对象系统中的对象标识。证书是类*C*在关系中的一个附加属性，同时在所有出现类*C*的关系中用于标识类*C*的对象。注意，实际中许多重要的类就是用这样的证书来表示的：学生的学号，司机的驾驶执照等等。

例4.22 假设名字不足以标识一个影星，因此将为每位影星赋予一个“证书号”，用以惟一地标识自己。这样Stars关系的模式是：

```
Stars(cert#, name, street, city, birthdate)
```

如果要用一个关系StarsIn表示电影与影星之间的多对多的联系的话，可以使用Movie中的title和year属性再加上表示影星的证书号。则联系模式是：

```
StarsIn(title, year, cert#)
```

□

4.4.7 习题

习题4.4.1 将以下习题中你的ODL设计转化为关系设计

* a) 习题4.2.1；

b) 习题4.2.2（包括该习题描述的四个修改方式）；

c) 习题4.2.3；

* d) 习题4.2.4；

e) 习题4.2.5；

习题4.4.2 将图4-5中的ODL描述转化为一个关系数据库模型的设计模式。习题4.2.6中的
164 修改对你的关系设计模式将产生怎样的影响?

! **习题4.4.3** 考虑一个字典类型的属性，其中字典类型的键类型以及值类型都是原子类型。怎样将拥有这样一个属性的类转化为一个关系?

* **习题4.4.4** 文中假设，如果你把图4-16中所示的类Studio的关系，与联系对owns和ownedBy组成另一个关系，将会违反BCNF规则。举例说明之。

习题4.4.5 我们提到过，把一个比记录的集合类型更复杂的类型，转化为关系需要一些技巧，特别是需要定义一些中间概念和关系。下面的一组习题会逐步地增加类型复杂性，如何将它们表示成为关系。

* a) 一张扑克牌（card）可以用一个结构来表示，结构包含一个牌面大小（rank）字段（2，3，...，10，Jack，Queen，King和Ace）和一个花色（suit）字段（梅花、方块、红心、黑桃）。给出结构类型Card的合理定义，并要求该类型的定义与其他任何类的声明相独立，但可以被它们使用。

* b) 一手牌（hand）是一个牌集，牌的数量不定。对类Hand进行声明，类的对象是一手牌，也就是说类声明中包含一个属性theHand，其类型为一手牌。

*! c) 将(b)中的Hand类声明转化为一个关系模式。

d) 一手扑克牌（poker hand）是指五张牌的集合，对于这个概念，重复(b)、(c)。

*! e) 发牌（deal）是一个“对”的集合，其中每“对”由玩家的名字和玩家的一手牌构成。声明类Deal，它的对象是发牌，也就是说，该声明中包括一个类型为发牌的属性theDeal。

f) 重复(e)，将其中的“一手牌”限制为“一手扑克牌”。

g) 重复(e)，对于其中的“发牌”使用字典，可以假定一次发牌中的各个玩家的名字互不相同。

*!! h) 将(e)中的类声明转化为一个关系数据库模式。

*! i) 假设定义发牌为牌集的集合（没有玩家与任一手牌相关）。建议使用如下关系模式来描述发牌：

```
Deals(dealID, card)
```

其中，card是某次发牌（deal）中某一手牌（hand）中的一张牌，这样表示有没有问题？如果有，问题在哪里？怎样改正？ 165

习题4.4.6 假设类*C*的定义如下

```
class C (key a) {
    attribute string a;
    attribute T b;
}
```

这里*T*是某种类型。如果*T*的类型如下，请给出类C的关系模式，并且指出关系的键：

a) Set<Struct S {string f, string g}>

*! b) Bag<Struct S {string f, string g}>

! c) List<Struct S {string f, string g}>

! d) Dictionary<Struct K {string f, string g}, Struct R {string i,string j}>

4.5 对象关系模型

关系模型以及ODL这样的面向对象的模型是两个重要的DBMS的模型。很长时间以来，主流的商业DBMS一直属于关系模型。在90年代，面向对象模型的DBMS有了一定的发展，但随即又消失了。数据库系统没有从关系模型转化为面向对象模型，相反关系模型的数据库开发商把许多面向对象模型中的概念融入到关系模型中。结果就是，许多过去叫做关系模型的DBMS

系统，现在被称为是对象关系模型的DBMS系统。

第9章中会遇到一个新的SQL标准，它是专门为对象关系模型数据库而制定。这一章只进行抽象的讨论。4.5.1节介绍对象关系概念，4.5.2节讨论它最早的一个实现（嵌套关系），对象关系中的类似于ODL的引用在4.5.3节讨论，4.5.4节比较对象关系模型与纯面向对象模型。

4.5.1 从关系到对象关系

关系的基本概念没有改变，但在加入了以下的特点之后，关系模型就被扩展成为对象关系模型（Object-Relational model）：

1. *结构类型的属性*。对象关系模型系统不再把属性的类型限制在原子类型，它支持一个与
166 ODL相似的类型系统：可以使用原子类型和类型构建器（如结构，集合和包等等）来创建类型。其中特别重要的类型是记录的集合[⊖]，这种类型本质上是一个关系，也就是说，一个元组的一个分量值可能就是完整的一个关系。

2. *方法*。特别定义的操作，可以用于用户定义的类型值。虽然还没有着手讨论如何操作关系模型，或者对象关系模型中的值和元组，但是在第5章，刚刚开始讨论这个问题时就会发现一些惊人的结果。例如，数值类型的值可以用加或小于这样的算术运算符来操作。不过，在对象关系模型中，还可以为一个类定义特别的操作，就像在例4.7中Movie类的ODL方法。

3. *元组的标识符*。在对象关系系统中，元组就是面向对象系统中的对象。所以在某些情况下，让每个元组都有一个独特的标识，以区别于其他元组，甚至是区别于所有组成分量都相等的元组会很有用。这个标识，就像ODL中的对象标识一样，对于用户而言一般不可见，不过在某些特殊情况下还是可见的。

4. *引用*。纯粹的关系模型系统没有引用（指向元组）的概念，而对象关系模型的系统可有不同的方式使用引用。

下一节，将集中讨论这些对象关系模型系统中新增的特点。

4.5.2 嵌套关系

上面第（1）点扩展的关系常常被称做“嵌套关系”。在*嵌套关系模型*（nested-relational model）中，允许非原子类型的关系属性。特别地，一个类型可以是一个关系模式。这样，就可以方便地递归定义关系属性的类型和关系的类型（模式）。

基础：属性的类型可以是一个原子类型（如整型、实型和字符串型等等）。

归纳：关系的类型可以是，任何包含一个或者多个合法类型属性的名字的模式。此外，模式也可以是任何属性类型。

167 在讨论关系模型时没有指出与每个属性联结的特殊原子类型，因为这些类型（整型、实型和字符串型等等）之间的差别并不会影响讨论的问题（比如，函数依赖和规范化）。在此将继续这条原则，不过在描述一个嵌套的关系模式时，必须指明哪些属性本身就是一个关系模式。为了做到这一点，将把这些属性当作关系名一样处理，用在属性后面加上括号括起来的属性（它们是属性的属性）列表来表示以上情况。于是，这些属性拥有自己的属性列表，并且可以有任意多级这样的嵌套关系。

例4.23 为影星设计一个嵌套的关系模式，其中包含了一个属性movies，它是一个代表所有该影星出演的电影集合的关系。Movies的关系模式包括title、year和影片的长度length。关系Stars的模式包括name、address和birthdate，还有就是movies中的信息。另外，address属性也是一个关系类型（包含street和city两个属性），可以在这个关

⊖ 严格地说，应该是一个包而不是一个集合，因为商业DBMS一般都支持有重复元组（是包不是集合）的关系。

系中记录影星的多个地址。Stars的模式可以设计为：

```
Stars(name, address(street, city), birthdate,
   movies(title, year, length))
```

图4-17显示了嵌套关系Stars的一个例子，其中有两个元组，一个是Carrie Fisher的，另一个是Mark Hamill的。为了节省空间简写了元组的分量值，以便于阅读。用虚线将元组之间分开，仅仅只是为了阅读方便，没有其他特别的意思。

name	*address*	*birthdate*	*movies*
Fisher	*street* / *city*: Maple / H'wood; Locust / Malibu	9/9/99	*title* / *year* / *length*: Star Wars / 1977 / 124; Empire / 1980 / 127; Return / 1983 / 133
Hamill	*street* / *city*: Oak / B'wood	8/8/88	*title* / *year* / *length*: Star Wars / 1977 / 124; Empire / 1980 / 127; Return / 1983 / 133

图4-17 一个关于影星以及他（她）出演的电影的嵌套关系

在Carrie Fisher的元组中，可以看到她的名字是原子类型，接下来是address的值。address是一个关系，这个关系有两个属性street和city，并有两个元组，每个都对应于她的一个住址。然后是她的birthdate（它也是原子类型的）。最后是movies属性，它的类型是一个关系，这个关系有title，year和length等三个属性，其值包含了Carrie Fisher最著名的三部影片。 168

第二个元组是关于Mark Hamill，其结构与上一个元组相同。其中address关系只有一个元组，因为他只有一个住址。Mark Hamill 的Movies关系的内容与Carrie Fisher的movies关系的内容很像，因为正巧两位影星的代表作一样。要注意的是，这两个关系（Movies）是元组的不同分量，只不过它们恰好有相同的值，这与两个不同分量恰好有相同的整数值124的情况类似。 □

4.5.3 引用

一部影片（如“*Star Wars*”）可能在嵌套关系Stars中的多个元组的movies关系中出现，这将导致冗余。实际上，例4.23中的模式就是一个不属于BCNF的嵌套关系模式。可是，即便分解Stars关系也无法避免冗余。于是必须设法令任何电影在Movies关系中最多出现一次。

要解决这个问题，对象关系模型需要提供引用元组（如元组t引用元组s，而不是直接合并t和s）的支持。因此需要为类型系统增加以下的归纳规则：一个属性的类型可以是对另外一个给定关系模式中某个元组的引用。

如果属性A的类型是对一个名为R的关系的元组的引用，在设计模式时将A表示为$A(*R)$。注意，这种情况与ODL中的联系相似：联系名为A，类型为R。也就是说，A属性被连接到一个R类型的对象。类似地，如果属性A的类型是一组对模式R的元组的引用，将它表示为：$A(\{*R\})$。这与ODL中联系A的类型是Set< R >的情况相似。

例4.24 消除图4-17中冗余的一种有效方式是使用两个关系：一个用于影星，另一个用于

电影。关系Movies是一个一般关系，其模式与例4.23中的属性Movies一样。关系Stars的模式与该例中的嵌套关系Stars相近，只是其Movies属性的类型变成一组对Movies元组的引用。两个关系的模式设计如下：

```
Movies(title, year, length)
Stars(name, address(street, city), birthdate,
    movies({*Movies}))
```

169

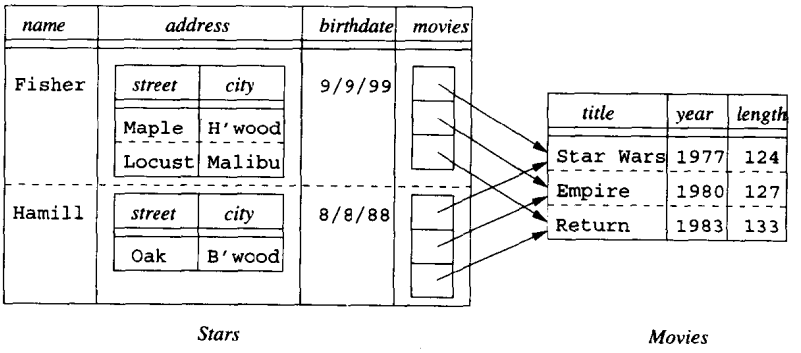


图4-18 引用集合作为属性的值

图4-17的数据填入这个新的模式中，就得到图4-18的结果。注意，因为每部电影只存在一个元组，虽然有许多对它的引用，但是消除了例4.23模式中的冗余。 □

4.5.4 面向对象与对象关系的比较

面向对象的数据模型（以ODL为典型）和这里讨论的对象关系模型极其相似！下面是几个显著方面的比较：

对象与元组

一个对象的值就是一个带有属性和联系结构的组成，ODL中没有规定联系表示方式的标准，不过可以假设对象之间的连接是通过某些指针集合建立起来。元组也是一个结构，不过在传统的关系模型中，它只包含属性。联系必须通过另一个关系的元组来表示，见3.2.2节。但是对象关系模型允许元组中包含引用集合，同样也允许联系直接与表示一个“对象”（或实体）的元组合并。

范围和关系

ODL中把类中存在的所有对象都认为是处于该类的“范围”中，而对象关系模型允许几个模式完全相同的关系并存。这可能令人感觉对象关系模型更加容易区别同一个类中的不同成员

170

（对象）。不过，ODL中支持接口（不包含范围的类的声明，见4.3.4节“接口”框中的说明）。这样ODL就允许用户定义任意多的类，它们都继承这个接口，而每个子类都可以有一个专有的范围。因此，当多个集合间共享同一个声明时，两者很类似。

方法

对象关系模型中我们没有讨论方法的使用。不过事实上，SQL-99标准以及面向对象的思想的使用，都使得对象关系模型和ODL一样，具有为任何类声明及定义方法的能力。

类型系统

两者的类型系统也相当地相似：都是在基于原子类型基础上使用结构、集合类型构建器来创建新类型。集合类型的可选部分可能有些不同，但至少都包含集合和包。而且，集合类型（或者包类型）和结构类型在两个模型中都有一个特殊的作用，那就是它们是ODL中的类和对

象关系模型中的关系类型。

引用和对象标识

一个纯粹的面向对象的模型使用一个用户完全不可见的对象标识，该标识不能由查询得到。对象关系模型允许类型中包含引用，所以，在有些情况下用户可以见到这些值，甚至在以后使用它们。这种情况是好是坏，也不能一概而论。在实际中，两个模型基本上没有差别。

向下兼容性

既然两种模型的差别微乎其微，那么为什么市场上，是对象关系模型而不是纯粹的面向对象的系统占主导地位呢？原因可能在于，在面向对象模型被提出以前，已经有相当数量的关系模型的商业系统。当关系数据库管理系统发展成对象关系管理系统时，开发商总是特别注意向下兼容性：系统的新版本仍然应当支持以前版本中可以使用的代码，并且接受同一个数据库模式，而用户并不关心是否采用了任何面向对象的特性。另一个原因还在于，要将一个系统完全转化为面向对象管理系统的工作量巨大。所以，尽管面向对象系统在技术上更有优势，但还不足以使开发商将大量已有的数据库转换到纯粹的对象关系管理系统上。 171

4.5.5 ODL设计到对象关系设计的转化

在4.4节，已经学习了将ODL设计转化为关系模型的模式的方法。当时的难点在于，ODL有很丰富的建模方式：非原子属性类型、联系和方法。现在要将其转化为对象关系模型，某些难点已经不再存在。针对不同的对象关系模型（考虑第9章中的具体的SQL-99模型），可以将大部分非原子类型转化为一种对应的对象关系类型，如结构、集合、包、链表以及数组。

如果一个ODL设计中的类型在对象关系模型中没有对应的类型的话，可以使用4.4.2节到4.4.4节介绍过的技巧。第4.4.5节中，已知对象关系模型中的联系的表示方式基本与关系模型中的一样（不过人们更习惯于使用引用，而不是键）。最后，如果ODL设计中包含方法，那么就无法将它转化为一个纯粹的关系模型，但对象关系模型大多数支持方法，所以这个方面的限制也可以放宽。

4.5.6 习题

习题4.5.1 使用嵌套关系和带引用的关系中的概念，设计包含以下信息的关系。对于每种情况，判断关系中应当包括哪些属性，请注意参考电影的例子。同时指出你的设计有无冗余，有的话，应当如何修改来避免？

* a) 电影：包括通常的属性，另外加上该电影的影星以及影星的通常信息。

*! b) 制片厂：包括制片厂制作的所有电影、每部电影中的所有影星以及电影、影星和制片厂的通常信息。

c) 电影和电影的制片厂、电影中的影星以及各自的通常信息。

* **习题4.5.2** 用对象关系模型表示习题2.1.1中的银行信息。要求可以方便地通过顾客的元组，得到其账户，同时也可以方便地通过账户的元组来找到该账户的户主。注意避免冗余。

! **习题4.5.3** 如果在习题4.5.2中每个账户只能有一个户主，那么怎样简化你在习题4.5.2中的设计？

! **习题4.5.4** 用对象关系模型实现习题2.1.3中的队员、球队、球迷。 172

! **习题4.5.5** 用对象关系模型实现习题2.1.6中的家谱。

4.6 半结构化数据

半结构化的数据（semistructured-data）模型在数据库系统中有一个独特的地位。

1. 它是一种适于数据库集成（integration）的数据模型，这是说，它可以用来描述多个数据库（这些数据库包含不同模式中的相似数据）中的数据。

2. 它是一种文档的模型，用于在Web上共享信息。4.7节要讨论的XML就是这样的一个例子。

这一节，将介绍半结构化的基本思想，理解为什么这种模型要比以前讨论过的模型更灵活。

4.6.1 为何需要半结构化数据模型

先来回忆一下E/R模型以及其基本的数据种类——实体集和联系。关系模型只有一种数据——关系。在3.2节还讨论了用关系来代替联系和实体集的方法。使用两种数据有其优点：在为现实世界建模时可以选择实体集或联系进行E/R模型设计，做到更加合适地匹配要模拟的概念。但是使用一种数据类型也有好处：它简化了表示模式的符号，而且查询数据库中各种类型数据也更加高效。从第11章开始将学习DBMS的实现，关系模型的优势将会更加明显。

现在来考虑4.2节介绍的面向对象模型，它有两个主要的概念：类（以及其范围）和联系。类似地，4.5节中介绍的对象关系模型中也有两个概念：属性（可以是类）和关系。

可以将半结构化数据看成是两个概念的混合：类-联系或者类-关系，就像关系模型将实体集和联系这两个概念混合起来一样，只是混合的原因不相同。关系模型成功的一大原因在于它简化了系统的高效实现，而人们对半结构化的模型尤为关注的原因则是其灵活性。虽然其他模型都有一个设计模式的概念（E/R图、关系模式或者ODL声明都是设计模式的例子），但是半结
173 构化的数据是“无模式”的。更加恰当的说法是，半结构化数据本身就指明了其模式，而且这样的模式会随着时间和数据库而改变。

4.6.2 半结构化数据表示

半结构化数据的数据库是节点（node）的集合，每个节点都是一个叶子节点（leaf）或者是一个内部节点（interior）。叶子节点与数据相关，数据的类型可以是任意原子类型，如数字和字符串。每个内部节点至少都有一条向外的弧。每条弧都有一个标签（label），该标签指明弧开始处的节点与弧末端的节点之间的关系。有一个名为root的内部节点，其上端没有弧，它代表整个数据库。每个节点都可从root可达，但是这个图未必是一棵树。

例4.25 图4-19是一个关于电影与影星的半结构化数据库。顶部的节点名为Root，这是整个数据库的入口，可以认为这个节点表示了整个数据库。中心对象（或者叫实体），（此例中为影星和电影）是Root节点的子节点。

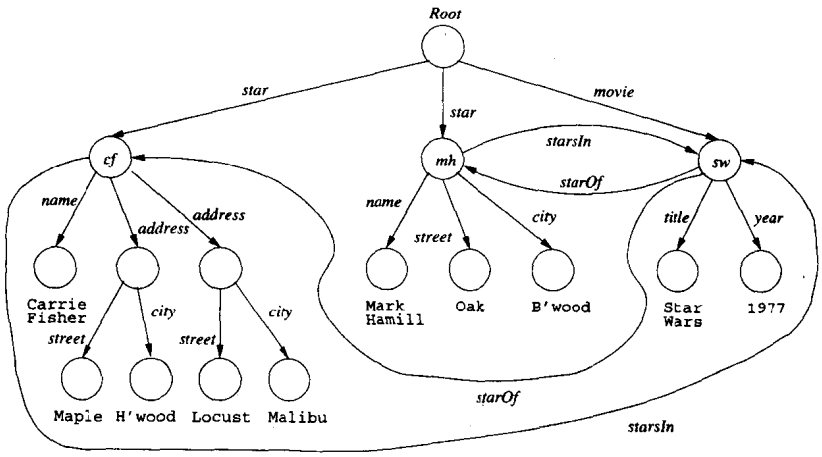


图4-19 表示一部电影和其中影星的半结构化数据

图中还看到许多叶子节点：例如，最左边的叶子节点旁边写着Carrie Fisher，最右边

的叶子节点旁边写着1977。还有许多内部节点，有三个特别的节点分别标着*cf*，*mh*和*sw*，分别代表“*Carrie Fisher*”，“*Mark Hamill*”和“*Star Wars*”。这些标签并不属于数据库，在这里给出只是为了能够方便地引用这些节点，否则它们将没有名字。例如考虑节点*sw*，它代表了概念“*Star Wars*”：该电影的名称和年份在此给出，而其他信息如长度和出演该部影片的影星没有给出。 □ 174

弧上的标签将类或者联系结合起来，有两个意义：（假设有一个名为*L*，从节点*N*出发到达节点*M*的标签）

1. 如果*N*是一个对象或者结构，而*M*是*N*的属性（*N*为对象）或者域（*N*为结构），那么*L*代表属性或者域的名字；

2. 若*N*和*M*都是对象，*L*就是从*N*到*M*的一个联系的名字。

例4.26 再看图4-19。*cf*代表的节点可以看做是代表Carrie Fisher的`Star`对象。从这个节点出发的一条弧的标签是name，它表示属性`name`，连接到代表名字的叶子节点。另外还有两条弧，其标签都是address，末端是两个表示地址的叶子节点。这两条弧对应于图4-12中的集合类型属性`address`。

这些地址都是结构，有子域`street`和`city`，图4-19中从这些节点出发的弧的标签是street和city，它们的末端都是对应数据的叶子节点。

图4-19中也有第二种弧。例如有一个从*cf*节点出发到*sw*节点，其标签名为`starsIn`的弧。对节点*mh*而言，也有一个类似的弧。节点*sw*有两个分别到*cf*和*mh*的弧。这些弧代表的是电影和影星之间的联系。 □

4.6.3 信息集成与半结构化数据

与以前讨论的模型不同，半结构化数据模型是*自描述*（self-describing），模式与数据附着在一起。每个节点（root节点除外）都是一个或者多个弧的末端，而这些弧的标签则说明了末端节点在出发节点中的角色。其他模型中，设计模式是固定的，它与数据本身分开，数据的角色信息隐含在设计模式中。

读者可能会想，这样的系统有些什么优点。事实上，有些小型的信息系统（如Lotus Notes）就是采用这种自描述数据的方式实现的。但是，数据库是为了大规模的数据而设计，一般而言，固定格式的数据更有优势，比如：固定的设计模式可以将数据组织成便于高效查询（第13章开始讨论）的形式。 175

半结构化的灵活性使之在两个应用中显得很重要，相关问题还会在4.7节讨论，这里仅仅考虑它作为信息集成的使用。实际应用中可能有许多数据库，一个很普遍的要求就是希望像在一个数据库中一样访问两个或者更多数据库中的数据。例如公司可能会合并，它们都有各自的关于人事、销售、存货、产品设计以及许多其他方面的数据库。如果这些数据库的模式都一致，那么要合并它们就很容易：比如只要把模式相同的两个关系中的元组合并即可。

但是，一般情况下都没有那么简单。各自开发的数据库不太可能有相同的设计模式，即便其中包含的都是人事信息：一个数据库中可能会有配偶姓名的记录，而另一个则没有；一个可能允许包含多个地址、电话和电子邮件的信息，而另一个可能只允许记录一个；一个数据库可能是关系型的，而另一个则是面向对象的。

更糟糕的是，数据库一般都是持续运行，不允许因为要复制数据到另一个数据库而将其关闭（即使可以得出从某一模式到另一模式的最有效的途径）。这些都被称做*遗留数据库问题*（legecy-database problem）；一旦数据库存在了，就不可以将其与应用分开，不允许它中途退役。

一种可能的解决方案见图4-20。显示了两个数据库外加一个接口，它也可包含多个遗留数据库。这两个数据库都不可改变，
176 这样它们就可以支持日常应用。

为了集成上的灵活性，接口支持半结构化数据，用户可以使用适于该数据的查询语言查询接口。半结构化的数据可以通过翻译数据源的数据来创建，每个数据库都有一个称做包装器(wrapper，或者叫做适配器“adapter”)的组件，负责翻译工作。

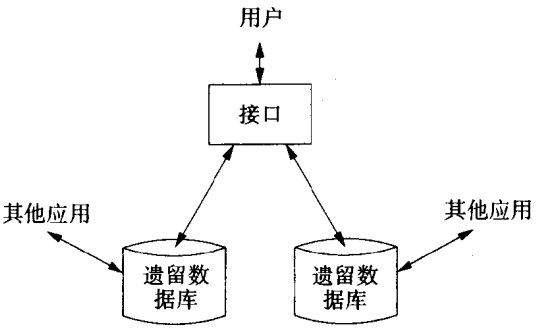


图4-20 通过一个支持半结构化数据的接口集成两个遗留的数据库

或者，根本就不存在接口处的半结构化数据。但是用户查询接口时仿佛有半结构化数据存在，接口通过把查询要求传递给数据源，并返回在数据源中相应模式的引用。

例4.27 在使用结构如图4-19的系统时，假设是从几个不同的数据源收集影星的信息。注意到Carrie Fisher地址信息是一个结构，有`street`和`city`两个组成部分。这种结构与嵌套关系`Stars ( name, address(street, city) )`的模式相似。

另一方面，Mark Hamill的地址信息不是结构，只是分开的`street`和`city`。这样的信息可能源于`Stars( name, street, city )`的设计模式，这样的模式仅仅支持每个影星一个地址。如果数据来自多个不同的数据库的话，还可能存在其他一些模式上的不同：可选的胶卷类型、导演、制片人、所属制片厂等等。 □

4.6.4 习题

习题4.6.1 由于在半结构化数据模型中不用设计模式，所以不要求你来设计模式。不过将要问你为了表现某种现实情况，应当如何组织数据：

* a) 在图4-19的基础上增加信息：“*Star Wars*”是由George Lucas导演，Gary Kurtz制作。

b) 在图4-19的基础上增加关于*Empire Strikes Back*和*Return of the Jedi*这两部影片的信息，其中Carrie Fisher和Mark Hamill参演了这些影片。

c) 在(b)的基础上增加制片厂福克斯（Fox）的信息，并包括该制片厂地址（Hollywood）。

* **习题4.6.2** 如何将习题2.1.1中的银行与客户数据用半结构化模型表示。

177 **习题4.6.3** 如何将习题2.1.3中的队员、球队和球迷的数据用半结构化模型表示。

习题4.6.4 如何将习题2.1.6中的家谱数据用半结构化模型表示。

*! **习题4.6.5** E/R模型和半结构化模型都是图形化的(使用了节点、标签以及节点间的连接)，但是这两个模型有一个本质的区别，请指出这个区别。

4.7 XML及其数据模型

XML（扩展标记语言）是一个基于标签的用于标记文档的符号，很像大家都熟悉的HTML（或者SGML）。一个文档（document）不过是一个包含许多字符的文件。但是，尽管HTML中的标签定义了文档中信息的表示方式（如哪部分信息用斜体，一张表的记录是什么等），而文档中子字符串的意义却定义在XML的标签中。

这一节，介绍XML的基本原理。它与4.6节给出的半结构数据的图形有相同结构，特别是

其标签的功能与半结构化数据图中的弧功能相同。接下来介绍DTD（文档类型定义），这是一种形式灵活的设计模式，可以用于使用了XML标签的文档。

4.7.1 语义标签

XML中标签是用尖括号，如<...>，括起来的文本，这与HTML中一样。而且，如同HTML，一般情况下标签成对出现：有一个开始标签形如<FOO>，和一个配对的结束标签形如</FOO>。HTML中可以有单个出现的标签，如分段用的<P>，XML中不允许这样。当标签在配对的开始-结束中出现时，还有另外一个要求，其开始-结束对必须嵌套，也就是说，在<FOO >和</FOO >之间，可以有任意数目的其他配对，但是如果该配对的开始是在这个域中，则其结束标签也必须在同一个域中出现。

XML可以使用于两种不同的模式：

1. *格式规范*（well-formed）的XML允许用户自定义标签，就好像半结构化数据中的标签一样。这个模式与半结构化模型很相似：没有设计模式，文档中的标签的形式任意。

2. *合法*（valid）的XML包括一个DTD，它定义了允许使用的标签和使用标签的语法（如何嵌套使用）。这种形式的XML是严格模式模型（如关系模型和ODL模型）与无模式模型（半结构模型）之间的一个折衷。在4.7.3节将看到DTD比传统的模式更加灵活，比如DTD可以定义可选的（或者是丢失的）域。 178

4.7.2 格式规范的XML

这种形式的XML的最简单形式是一个XML声明和一个*根标签*（root tag），它将整个文档包含在内。形式如下：

```
<? XML VERSION = "1.0" STANDALONE = "yes" ?>
<BODY>
    ...
</BODY>
```

第一行表明文件是XML文档。参数STANDALONE = “yes”说明本文档没有DTD：即本文档是一个完全形式的XML。注意这个声明使用了一个特别的标记<?...?>。

```
<? XML VERSION = "1.0" STANDALONE = "yes" ?>
<STAR-MOVIE-DATA>
    <STAR><NAME>Carrie Fisher</NAME>
        <ADDRESS><STREET>123 Maple St.</STREET>
            <CITY>Hollywood</CITY></ADDRESS>
        <ADDRESS><STREET>5 Locust Ln.</STREET>
            <CITY>Malibu</CITY></ADDRESS>
    </STAR>
    <STAR><NAME>Mark Hamill</NAME>
        <STREET>456 Oak Rd.</STREET><CITY>Brentwood</CITY>
    </STAR>
    <MOVIE><TITLE>Star Wars</TITLE><YEAR>1977</YEAR>
    </MOVIE>
</STAR-MOVIE-DATA>
```

图4-21 一个关于电影和影星的XML文档

例4.28 图4-21是一个与图4-19基本对等的XML文档。根标签是STAR-MOVIE-DATA。图中看到有两个用标签对<STAR>和</STAR>括起来的段，其中包含影星名字的子段。一个是关于Carrie Fisher，它有两个用<ADDRESS>和</ADDRESS>括起来的子段，每段都包含她的一个住址的信息。Mark Hamill的子段只用了一个关于街道和城市的子段，而没有使用<ADDRESS> 179

标签。图4-19中也有这个差别。

注意，图4-21的文档并不表现电影和影星之间的联系starsIn。但是可以将信息放在影星的段中，如：

```
<STAR><NAME>Mark Hamill</NAME>
    <STREET>Oak</STREET><CITY>Brentwood</CITY>
    <MOVIE><TITLE>Star Wars</TITLE><YEAR>1977</YEAR></MOVIE>
    <MOVIE><TITLE>Empire</TITLE><YEAR>1980</YEAR></MOVIE>
</STAR>
```

不过这种方式导致冗余。由于影片信息是重复地出现在所有参演的影星的信息中，这里仅仅包含了该影片的键信息，即电影名字和年份，这不是实际地表示冗余的例子。4.7.5节会看到XML在一个本质上是树形结构的标签系统中，如何处理这个问题。 □

4.7.3 文档类型定义（DTD）

为了让计算机自动处理XML文档，需要类似于设计模式的信息。也就是说，须要指明哪些标签可以出现在文档集中，标签如何被嵌套。这种模式的描述是使用一组类似于语法的规则集，称之为*文档类型定义*（document type definition），或称为DTD。使用DTD主要是考虑到，需要共享数据的公司或者团体可以通过创建各自的DTD来展示其标签的语义。例如可以使用一个DTD来描述蛋白质，也可能使用一个DTD来描述采购和销售，等等。

DTD的基本结构是：

```
<!DOCTYPE root-tag [
    <!ELEMENT element-name (components)>
        more elements
]>
```

其中根标签与其配对的结束标签之间包括所有符合该DTD定义规则的整个文档。*元素*（element）有一个名字和括号括起来的一些组件，名字用于标签，在其对应的XML文档中，该标签包含的区域就是元素代表的内容。括号中的组件代表的是元素中可以或者必须出现的标签。对每个组件有确切要求的定义方式将很快介绍。

有一个很重要的特例。元素名字后面的（#PCDATA）意味着元素后面的是值是文本，其中不再含有标签。

180 **例4.29** 图4-22中是关于影星的一个DTD[⊖]，其名字和最外层的标签是STARS（XML与HTML一样都是大小写无关的，所以STARS就是根标签）。第一个元素定义是说，在标签对<STARS></STARS>之间可能会发现零或多个STAR标签，每对STAR标签间的内容都对应一位影星，其中*号代表“零或多个”，也就是任意多个。

第二个元素是STAR，声明表示它包含3个子元素：NAME、ADDRESS和MOVIES。它们必须出现，而且必须以这个次序出现。ADDRESS后面的+号表示“一个或多个”，也就是说，对于一位影星，可以有一个或者多个地址存在。NAME被定义为“PCDATA”，即为文本。第四个元素是地址元素，它由street和city两个字段顺序构成。

MOVIES元素被定义为可以有零个或者多个MOVIE（*的意义是任意多个）。MOVIE元素有title和year两个字段，都是文本类型。图4-23是与图4-22中的DTD相符合的文档的例子。 □

⊖ 注意，图4-21中的影星和电影数据不符合这个DTD。

```
<!DOCTYPE Stars [
    <!ELEMENT STARS (STAR*)>
    <!ELEMENT STAR (NAME, ADDRESS+, MOVIES)>
    <!ELEMENT NAME (#PCDATA)>
    <!ELEMENT ADDRESS (STREET, CITY)>
    <!ELEMENT STREET (#PCDATA)>
    <!ELEMENT CITY (#PCDATA)>
    <!ELEMENT MOVIES (MOVIE*)>
    <!ELEMENT MOVIE (TITLE, YEAR)>
    <!ELEMENT TITLE (#PCDATA)>
    <!ELEMENT YEAR (#PCDATA)>
]>
```

图4-22 影星文档的DTD

*E*元素的组成通常是一些其他元素，这些元素必须在标签对<*E*></*E*>中出现，有一些操作符号可以控制元素出现的次数。

1. * 号表示该元素可以出现任意多次，包括零次。

2. + 号表示该元素可以出现一次或者多次。

3. ? 号表示该元素只可以出现零次或一次。 181

4. 符号 | 可以在元素之间或者在用括号括起来的元素组之间出现，用于表示“或者”，要么是符号 | 左边的元素（组）要么是符号 | 右边的元素（组）出现，但是不能同时出现。例如表达式（#PCDATA |（STREET CITY））作为ADDRESS元素的组成意味着地址可以是简单文本，或者是由street和city分量构成。

```
<STARS>
    <STAR><NAME>Carrie Fisher</NAME>
        <ADDRESS><STREET>123 Maple St.</STREET>
            <CITY>Hollywood</CITY></ADDRESS>
        <ADDRESS><STREET>5 Locust Ln.</STREET>
            <CITY>Malibu</CITY></ADDRESS>
        <MOVIES><MOVIE><TITLE>Star Wars</TITLE>
            <YEAR>1977</YEAR></MOVIE>
            <MOVIE><TITLE>Empire Strikes Back</TITLE>
            <YEAR>1980</YEAR></MOVIE>
            <MOVIE><TITLE>Return of the Jedi</TITLE>
            <YEAR>1983</YEAR></MOVIE>
        </MOVIES>
    </STAR>
    <STAR><NAME>Mark Hamill</NAME>
        <ADDRESS><STREET>456 Oak Rd.<STREET>
            <CITY>Brentwood</CITY></ADDRESS>
        <MOVIES><MOVIE><TITLE>Star Wars</TITLE>
            <YEAR>1977</YEAR></MOVIE>
            <MOVIE><TITLE>Empire Strikes Back</TITLE>
            <YEAR>1980</YEAR></MOVIE>
            <MOVIE><TITLE>Return of the Jedi</TITLE>
            <YEAR>1983</YEAR></MOVIE>
        </MOVIES>
    </STAR>
</STARS>
```

图4-23 与图4-22中的DTD对应的文档例子

4.7.4 使用DTD

若要一个文档与一个特定的DTD相一致，可以使用如下的任一种方法：

a) 在文档之前包含DTD。

182 b) 在开始行引用DTD。DTD必须在文件系统中分开存放，处理文档的应用程序应当可以访问到。

例4.30 下面是图4-23中文档要引用图4-22中DTD时，必须增加的内容。

```
<?XML VERSION = "1.0" STANDALONE = "no"?>
<!DOCTYPE Stars SYSTEM "star.dtd">
```

参数STANDALONE = "no" 表示使用了DTD。以前的例子中这个参数的值为 "yes"，因为那时不想使用DTD。DTD文件的位置在!DOCTYPE子句中给出，其中关键字SYSTEM后面的文件名就是DTD的位置信息。 □

4.7.5 属性列表

XML文档与半结构化数据之间有一种很强的联系。假设对于文档中的标签对<*T*>和</*T*>，创建一个节点*n*，而对于嵌在其中的标签对<*S*>和</*S*>（S与*T*之间不再有其他的标签层），画一条标签为*S*的从节点*n*到*S*标签对的弧，这样可以得到一个本质上与文档的结构一致的半结构化数据。

但是反过来就没有这样的对应（如果仅仅使用到现在为止学习到的XML的话）。在XML中，需要一种方式来表示有多条弧指向元素实例。显然无法将一个标签对嵌套在一个以上的标签对中，因为，嵌套不足以表达一个节点有多个前驱节点的情况。XML中还有另外一些特性，如：标签内的属性、标识符（ID）和标识的引用（IDREF）等，能够表示所有的半结构化数据。

开始标签可以有属性，这些属性出现在标签中，这与HTML中的<A HREF = ...>形式类似。保留字!ATTLIST声明了元素的一列属性及其类型。属性的一种典型用法是将值与标签联结起来，这种方式与使用类型为PCDATA的子标签可以达到同样的目的。

属性另一个重要的用途就是表示一个非树型的半结构化数据。类型为*E*的元素的ID属性的值，将惟一地标识每个由标签对<*E*>和</*E*>包含的段。用半结构的说法就是：ID为节点提供了一个独特的名字。

其他的属性可以被声明为*IDREF*，其值为与本标签表示的内容相关的其他标签的ID。只要给标签实例一个值为*v*的ID，给另一个标签实例一个值为*v*的IDREF，就可以有效地给出一个弧，从前者出发到后者结束。下面的例子展示了声明ID和IDREF的句法，以及在数据中使用它们
183 的重要性。

```
<!DOCTYPE Stars-Movies [
    <!ELEMENT STARS-MOVIES (STAR* MOVIE*)>
    <!ELEMENT STAR (NAME, ADDRESS+)>
        <!ATTLIST STAR
            starId ID
            starredIn IDREFS>
    <!ELEMENT NAME (#PCDATA)>
    <!ELEMENT ADDRESS (STREET, CITY)>
    <!ELEMENT STREET (#PCDATA)>
    <!ELEMENT CITY (#PCDATA)>
    <!ELEMENT MOVIE (TITLE, YEAR)>
        <!ATTLIST MOVIE
            movieId ID
            starsOf IDREFS>
    <!ELEMENT TITLE (#PCDATA)>
    <!ELEMENT YEAR (#PCDATA)>
]>
```

图4-24 一个使用ID和IDREF的关于影星和电影的DTD

例4.31 图4-24展示了一个修改过的DTD，其中影星和电影的地位相同。ID-IDREF用于描述电影和影星之间的多对多联系。类似地，图4-19中节点电影和节点影星之间的弧也描述了多对多的联系。根标签的名字变为STARS-MOVIES，元素为一系列的影星加上一系列的电影。

影星不再有表示电影集合的子元素（如图4-22），其属性只包括名字和地址。在<STAR>标签中，可以看到一个类型为ID列表的属性starredIn。注意，starredIn的类型是IDREFS，结尾的*S*表示这是一个ID的列表。

<STAR>标签还有一个属性starId，其类型为ID。该属性的值可能会被<MOVIE>标签引用，以表示影片中的影星。在图4-24的MOVIE属性列表中，有一个类型为ID的movieId属性，这是starredIn标签的值的来源。对称地，MOVIE的starsOf属性是影星的ID列表。

图4-25是一个符合图4-24中DTD的例子。它很像图4-19中的半结构化数据，不过它包含了更多的数据（有3部电影而不是一部电影）。惟一的结构上的差别在于，所有的影星都有一个 184
ADDRESS子元素，就算只有一个地址也如此。而图4-19中，表示Mark Hamill的节点只有street和city节点。 □

```
<STARS-MOVIES>
    <STAR starId = "cf" starredIn = "sw, esb, rj">
        <NAME>Carrie Fisher</NAME>
        <ADDRESS><STREET>123 Maple St.</STREET>
            <CITY>Hollywood</CITY></ADDRESS>
        <ADDRESS><STREET>5 Locust Ln.</STREET>
            <CITY>Malibu</CITY></ADDRESS>
    </STAR>
    <STAR starId = "mh" starredIn = "sw, esb, rj">
        <NAME>Mark Hamill</NAME>
        <ADDRESS><STREET>456 Oak Rd.<STREET>
            <CITY>Brentwood</CITY></ADDRESS>
    </STAR>
        <MOVIE movieId = "sw" starsOf = "cf, mh">
            <TITLE>Star Wars</TITLE>
            <YEAR>1977</YEAR>
        </MOVIE>
        <MOVIE movieId = "esb" starsOf = "cf, mh">
            <TITLE>Empire Strikes Back</TITLE>
            <YEAR>1980</YEAR>
        </MOVIE>
        <MOVIE movieId = "rj" starsOf = "cf, mh">
            <TITLE>Return of the Jedi</TITLE>
            <YEAR>1983</YEAR>
        </MOVIE>
</STARS-MOVIES>
```

图4-25 符合图4-24中的DTD的文档例子

4.7.6 习题

习题4.7.1 将以下情况增加到图4-25中：

* a) Harrison Ford同样出演了图中提到三部影片，另外他还出演了电影*Witness*（1985）。

b) Carrie Fisher同样出演了影片*Hannah and Her Sisters*（1985）。

c) Liam Neeson出演影片*The Phantom Menace*（1999）。 185

* **习题4.7.2** 为习题2.1.1中银行和客户的典型数据设计一个DTD。

习题4.7.3 为习题2.1.3中队员、球队和球迷的典型数据设计一个DTD。

习题4.7.4 为习题2.1.6中家谱的典型数据设计一个DTD。

4.8 小结

- *对象定义语言*（object definition language）：该语言用于面向对象方式描述数据库的模式设计。用户可以定义类，它有三种特性：属性、方法和联系。
- *ODL联系*：ODL中的联系必须是二元的。它在两个类（联系的两端）中通过名字来声明（同时声明其反向联系）。联系可以是多对多、多对一或者一对一的，这取决于联系的类型是被声明为单个对象还是对象的集合。
- *ODL的类型系统*：ODL运行自定义类型，从类名和原子类型开始使用类型构建器：结构、集合、包、链表、数组和字典构建器。
- *范围*（extent）：一个类可以有一个范围，它是当前数据库中存在的所有的类的对象的集合。因而范围与关系模型中的关系相对应，而类则对应于关系的设计模式。
- *ODL中的键*：ODL中键是可选的。用户可以定义一个或者更多的键。但是由于每个对象都有一个对象标识符，所以ODL的实现系统可以区别不同的对象，就算对象所有的属性都有相同的值时也如此。
- *将ODL设计转化为关系设计*：如果只把ODL作为一种设计语言，最终的结果将转化为关系的话，最简单的方式是为类的每个属性都创建一个关系，为每对联系也都创建一个关系。不过，也可以把一个多对一的联系与“多”的类的属性的关系结合起来。对于没有键的类，必须为其创建新的可以充当键的属性。

186

- *对象关系模型*：与ODL这样纯粹的面向对象数据库模型类似的，是扩展关系模型（包含面向对象的一些主要特性）而得到的对象关系模型。这些扩展包括了嵌套的关系（支持关系的复杂类型的属性，包括将关系转化为类型）。其他的扩展有：定义类型的方法，以及通过引用类型使一个元组具有引用另一个元组的能力。
- *半结构化数据*：这种模型中，数据使用图形来表示。节点类似于对象或者是对象的属性的值，带标签的弧可以将对象与它的属性值和由联系连接的其他对象相连接。
- XML：它是一个WWW Consortium制定的标准。它在文档（文本文件）中实现了半结构。节点对应于文本的段，（有些）带标签的弧在XML中使用成对出现的标签来表示。
- *XML中的标识符和引用*：为了表示非树形结构的图，XML允许在标签对的开始标签中使用类型`ID`和`IDREF`。标签（与半结构化数据的一个节点对应）可以有一个标识符，该标识符可以在其他的标签中使用，以建立一个连接（半结构化数据中的弧）。

4.9 参考文献

定义ODL的手册见[6]，该工作由ODMG（对象数据管理组织）继续完成。[4]、[5]、[8]中也可以找到面向对象数据库系统的由来。

将半结构化数据作为模型是来自Stanford开发的TSIMMIS和LORE项目。该模型的原始描述见[9]。LORE和它的查询语言在[3]中描述。对半结构化数据研究的最新详述见[1]、[10]以及[2]。关于半结构数据的书目清单正在Web上整理，见[7]。

XML标准由WWW Consortium制定。XML的主页见[11]。

1. S. Abiteboul, “Querying semi-structured data,” *Proc. Intl. Conf. on Database Theory* (1997), Lecture Notes in Computer Science 1187 (F. Afrati

and P. Kolaitis, eds.), Springer-Verlag, Berlin, pp. 1–18.

2. Abiteboul, S., D. Suciu, and P. Buneman, *Data on the Web: From Relations to Semistructured Data and Xml*, Morgan-Kaufmann, San Francisco, 1999.

3. Abiteboul S., D. Quass, J. McHugh, J. Widom, and J. L. Weiner, "The LOREL query language for semistructured data," In *J. Digital Libraries* **1**:1, 1997.

4. Bancilhon, F., C. Delobel, and P. Kanellakis, *Building an Object-Oriented Database System*, Morgan-Kaufmann, San Francisco, 1992.

5. Cattell, R. G. G., *Object Data Management*, Addison-Wesley, Reading, MA, 1994.

6. Cattell, R. G. G. (ed.), *The Object Database Standard: ODMG-99*, Morgan-Kaufmann, San Francisco, 1999.

7. L. C. Faulstich,

   `http://www.inf.fu-berlin.de/~faulstic/bib/semistruct/`

8. Kim, W. (ed.), *Modern Database Systems: The Object Model, Interoperability, and Beyond*, ACM Press, New York, 1994.

9. Papakonstantinou, Y., H. Garcia-Molina, and J. Widom, "Object exchange across heterogeneous information sources," *IEEE Intl. Conf. on Data Engineering*, pp. 251–260, March 1995.

10. D. Suciu (ed.) Special issue on management of semistructured data, *SIGMOD Record* **26**:4 (1997).

11. World-Wide-Web-Consortium, `http://www.w3.org/XML/`

第5章 关系代数

这一章开始学习关于数据库编程的设计，即用户如何对数据库进行查询以及修改数据库的内容等。重点是对于关系数据库的讨论，特别是那些用于关系代数查询的符号。

尽管面向对象的数据库语言ODL在原则上可以执行任何数据操作，而E/R模型不提供有效的途径来操作数据，但是关系模型的数据库已经形成了数据操作的“标准”操作集。这些操作并不是像其他编程语言那样是“图灵完全的”，因此，有些操作在关系代数中不能完成，但是在ODL方法中用C++实现。这些并不是关系数据模型或关系代数的缺点，因为限制数据库操作范围使得我们可以使用任何高级的语言（如SQL）来优化查询。在第6章中将要介绍的SQL语言就是这种语言。

本章从介绍关系代数的操作开始。这类代数形式地应用于元组集，即关系。可是，商业数据库管理系统（DBMS）使用的关系模型稍微有些不同，是用包而不是集合。也就是说，关系中可以含有重复的元组。虽然把关系代数看成是集合代数有好处，但是，人们仍然需要对关系代数操作结果中重复出现的元组非常小心。在本章的最后一节，将进一步考虑怎样约束关系的问题。

下面的几章将介绍现在常用的一些商用数据库管理系统使用的语言。关系代数的所有操作都用SQL数据库查询语言来实现，SQL将在第6章中介绍。这些代数操作同样在OQL语言当中出现，OQL语言是一种面向对象的查询语言，它建立在ODL模型的基础上，第9章将介绍有关它们的情况。 189

5.1 一个数据库模式的例子

在继续探讨有关关系模型数据库编程的话题之前，有必要为将要使用的例子建立一个模式。所造的例子是从movie、stars、studio例子中得出。这很像3.6节得到的规范化关系。但是，例子中有一些以前没有的属性，另外MovieExec关系也是以前没有提到的。这样做的目的是为了能学习更多的数据类型和表示数据的方法。图5-1给出了这个模式。 190

该模式含有五个关系。每个关系的属性和属性的域都在图中列出，并且用大写字母表示键属性（正文中仍使用小写字母）。比如说，关系StarsIn由三个属性组成，关系Movie有六个属性：title和year共同组成了关系Movie的键。属性title是字符串类型，而year是整型。

跟前面的模式相比，这个模式所作的主要改动有：

- 每一个电影制片人有一个授权证书号码，这个授权号码是具有惟一性的整型数。授权书可以想像为由一个外部的权利机构（或许是一个电影制片人的专门登记处，或许是一个其他的协会）来维护。
- 用授权证书号码作为电影制片人的键，由于影星并不一定具有这个授权，所以仍然用name作影星的键。虽然这样的设计可能跟实际情况有所不同，比如两个影星可能有同样的名字，但是可以以此来解释一些特殊的操作。
- 电影制片人是电影的一个属性，用producterC#来表示，表示该电影制片人的授权证书号码。电影制片人被认为是电影的执行者，或者是工作室主管。在MovieExc关系中

还可以有其他的执行者。

- Movie的属性filmType被从枚举型改为布尔型，称做inColor，该属性取真值表示彩色片，取假值表示黑白片。
- 影星关系中增加了性别属性gender，它属于字符串类型，“M”代表男性；“F”代表女性。影星关系中还增加了生日属性，它的类型是日期型。日期型是很多商用的数据库支持的特殊类型，其实也可以简单地把它看成是字符串。
- 所有的地址都用字符串类型string来表示，而不是用“街道”和“城市”数据对组成。目的是方便不同的关系中的地址的比较，同时也可以简化对不同关系中地址的操作。

```
Movie(
    TITLE: string,
    YEAR: integer,
    length: integer,
    inColor: boolean,
    studioName: string,
    producerC#: integer)

StarsIn(
    MOVIETITLE: string,
    MOVIEYEAR: integer,
    STARNAME: string)

MovieStar(
    NAME: string,
    address: string,
    gender: char,
    birthdate: date)

MovieExec(
    name: string,
    address: string,
    CERT#: integer,
    netWorth: integer)

Studio(
    NAME: string,
    address: string,
    presC#: integer)
```

图5-1 关于电影的数据库实例

5.2 关系代数操作

在开始学习关系上操作时，首先应该学习一种称做*关系代数*（relational algebra）的特殊代数，包括一些从给定的关系中建立新的关系的简单却具有强大功能的方法。当给定的关系存有
191 数据时，新得到的关系就可以作为原有关系查询的结果。

关系代数的发展已经历了一段历史时期，下面的叙述将大致跟踪这段历史。最初，关系代数是由T.Codd提出，它是一个元组集合（即关系），能用来进行典型的基于关系的查询。它由集合上的5个操作组成：并，差，笛卡儿积及另外两种不太常见的操作：选择和投影。除此之外，还有在这些操作之上定义的附加操作，其中各种join操作是最重要的。

Bag 为什么比Set更高效

有一个简单的例子可以说明为什么bag的实现比set更高效。如果在求两个关系的并时不消除重复元组，bag方法则就是直接把关系复制到输出，如果你坚持用set来实现的话，就必须先对结果中的元组排序，或者使用其他相似的操作，检查来自两个不同集合的重复元组，以保证得到的结果是一个set。

当最初开发关系模型的DBMS时，它们的查询语言大多以关系代数作为工具。但为了提高效率，这些系统把关系看做是包（bag）而非集合（set）。这就是说，除非用户明确地说明把重复的元素减少为一个（也就是除去重复元素的副本），关系就允许重复元组的存在。这样，在5.3节中将研究关于包的这些操作，看看都需要做哪些必要的改变。

对于商用的关系模型实现来说，另外一个需要的改进就是要加入一些其他操作。最重要的是加入了聚合操作，比如求某一个列的平均值。这些内容将在5.4节中介绍。

5.2.1 关系代数基础

通常，一门代数，总是由一些操作符和一些原子操作数组成。比如说，算术代数中原子操作数是像变量X和常量15这样的操作数。而加减乘除是其中的操作符。任何一门代数都允许把操作符作用在原子操作数或者是其他代数表达式上构造表达式（expression），一般地，括号被用来组合操作符和操作数。例如算术中有表达式：$(x+y)*z$和$((x+7)/(y-3))+x$。

关系代数是另外一门代数，它的原子操作数是：

1. 代表关系的变量。

192

2. 代表有限关系的常量。

正像前面提到的那样，在经典的关系代数当中，所有表达式的操作数和结果都是集合。传统关系代数的操作主要有以下四类：

a) 通常的关系操作：并、交、差。

b) 除去某些行或者列的操作。选择是消除某些行的操作，而投影是消除某些列的操作。

c) 组合两个关系元组的操作。包括有“笛卡儿积运算“（Cartesian product），该操作尝试两个关系的所有可能的元组配对方式，形成一个关系作为结果。另外还有许多连接（join）操作，它是从两个关系中选择一些元组配对。

d) 重命名（renaming）操作。不影响关系中的元组，但是它改变了关系的模式。即，属性的名称或者是关系本身的名称被改变。

人们一般把关系代数的表达式称为查询（query）。虽然到目前为止还没有更多表示关系代数表达式的符号，但是应该熟悉（a）类操作，知道“$R \cup S$”是关系代数表达式。R和S是表示关系的原子操作数，它们的具体元组未知。这个操作是查询R，S中所有元组的并集。

5.2.2 关系中的集合操作

三个最常用的集合操作是：并（union）、交（intersection）、差（difference）。这里假设读者已经熟悉这三种操作。下面是这些操作在任意集合R和S上的定义：

- $R \cup S$，表示关系R和S的并，所得到的结果关系的元素是来自R或者S或者在R和S中都出现过，但是对于最后一种情况，结果关系中的这个元素只出现一次。
- $R \cap S$，表示关系R和S的交，就是同时在R和S中存在的元素的集合。
- $R-S$，是关系R和S的差，它是由在R中出现，但是不在S中出现的元素构成的集合。注意，$S-R$与$R-S$不同，前者表示由只在S中出现但不在R中出现的元素构成的关系。

193 当在关系上应用这些操作的时候，需要对关系R和S另外加些条件：

1. R和S必须是具有同样属性集合的表，同时，R和S的各个属性的类型（域）也必须匹配。

2. 在做相应的集合操作（指并、交、差）之前，R和S的列必须经过排序，这样保证他们的属性序对于两个关系来说完全相同。

人们有时希望对具有相同属性个数、相应的块都相同、但是只有不同属性名的关系进行并、交和差运算等等，这时候，就要用到*重命名*（renaming）操作，对这两个关系模式修改使其具有相同属性名。5.2.9节将对此进行介绍。

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

Relation R

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

Relation S

图5-2 两个关系

例5.1 假设我们有两个关系R和S和5.1节中的MovieStar关系实例。R和S两个关系实例如图5-2所示，这样，他们的并运算$R \cup S$的结果是：

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

注意，两个表中均有Carrie Fisher出现，但是在结果中却仅有一个。

$R \cap S$的运算结果是：

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99

194

现在只有Carrie Fisher出现，因为只有这个元组同时在两个关系中出现。

$R-S$是：

name	*address*	*gender*	*birthdate*
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

也就是说，Fisher和Hamill这两个元组都是$R-S$的候选元素，但是Fisher在S中也出现，所以，就不在$R-S$中了。 □

5.2.3 投影

投影（projection）操作用来从关系R生成一个新的关系，这个关系只包含原来关系R的部分列。表达式 $\pi_{A_1, A_2, \ldots, A_n}(R)$ 的值是这样的一个关系：它只包含关系R属性A_1，A_2，...，A_n所代表的列。结果关系模式的属性集合为：$\{A_1, A_2, \ldots, A_n\}$，习惯上按所列出的顺序显示。

例5.2 考虑关系Movie，它的关系模式在5.1节有描述。图5-3给出了关系的一个实例。投影到关系前三个属性的表达式是：

$$\pi_{title,year,length}(\text{Movie})$$

title	*year*	*length*	*inColor*	*studioName*	*producerC#*
Star Wars	1977	124	true	Fox	12345
Mighty Ducks	1991	104	true	Disney	67890
Wayne's World	1992	95	true	Paramount	99999

图5-3 关系Movie

所得到的结果是：

title	*year*	*length*
Star Wars	1977	124
Mighty Ducks	1991	104
Wayne's World	1992	95

另外的一个例子是用表达式$\pi_{inColor}$(Movie)投影属性inColor。结果是一个单列关系：

inColor
true

注意这里只有一个元组，因为所有三个元组都有共同的inColor属性值，在关系代数集合中，重复元素总是被排除。 □ 195

5.2.4 选择

当选择（selection）操作应用到关系R上时，产生原关系的的新关系。结果关系的元组必须满足某个涉及R中属性的条件C。这个操作表示为：$\sigma_C(R)$。结果关系和原关系有着相同的模式，习惯上用跟原关系相同的顺序列出这些属性。

C是某个类型的条件表达式，它与人们熟悉的程序设计语言条件表达式类似。例如：Java或C语言中if关键字后面的条件表达式。所不同的是C中的操作数要么是一个常量要么是R的一个属性。假设t是R中的任意一个元组，把t代入到条件C中，如果代入的结果为真，那么这个元组就是$\sigma_C(R)$中的一个元组，否则此元组不在结果中出现。

例5.3 关系Movie还是像图5-3中表示的那样，那么表达式$\sigma_{length \geqslant 100}$(Movie)的结果是：

title	*year*	*length*	*inColor*	*studioName*	*producerC#*
Star Wars	1977	124	true	Fox	12345
Mighty Ducks	1991	104	true	Disney	67890

第一个元组满足表达式length≥100，因为当把第一个元组的length属性用它的实际值124代入后，它满足这个条件表达式124≥100。用同样的方法知道图5-3的第二个元组也满足这个表达式，所以也应该包括在结果当中。

第三个元组的length属性值是95。同样把这个值代入到表达式中，得到：95≥100，显然这个表达式的值为false。因此，图5-3的最后一个元组不在结果集中。 □

例5.4 假设想要得到这样的元组集合：在关系Movie中的所有Fox公司出品的至少有100分钟长的电影。就必须用更复杂的，包含AND和两个子条件的表达式来实现这样的查询。这个表达式可以写成：

$$\sigma_{length \geqslant 100 \text{ AND } studioName = \text{'Fox'}}(\text{Movie})$$

元组

title	*year*	*length*	*inColor*	*studioName*	*producerC#*
Star Wars	1977	124	true	Fox	12345

196 是结果关系中惟一的元组。 □

5.2.5 笛卡儿积

关系R和S的笛卡儿积（或者称为叉积或者就称做积），是一个有序对的集合，有序对的第一个元素是关系R中的任何一个元组，第二个元素是关系S中的任何一个元组，表示为$R \times S$。当R和S都是关系的时候，积本质上仍是关系。但是由于R和S这两个关系的属性未必只有一个，所以结果产生了更长的元组，包括了R和S中的所有属性。习惯上，在结果中关系R的属性出现在关系S的属性的前面。

A	B
1	2
3	4

关系 R

B	C	D
2	5	6
4	7	8
9	10	11

关系 S

结果关系模式是R和S关系模式的并，但是如果R和S恰好有同样的属性，就需要把至少一个属性名更改成不同的名称。为了使意义清楚，如果属性A在R和S中均出现，则结果关系模式中分别用$R.A$和$S.A$表示来自R和S的

197 属性。

例5.5 为了简单起见，利用一个抽象例子来解释积操作。令R和S的模式和元组如图5-4所示。于是$R \times S$就有如图所示六个元组。注意这里是怎样把每一个R中的元组跟三个S中的元组分别配对，因为属性B在R和S中均出现，$R \times S$中分别是用$R.B$和$S.B$表示。其他几个属性都不会引起混淆，所以保持原来名字不变。 □

A	R.B	S.B	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

$R \times S$ 的结果

图5-4 两个关系和它们的笛卡儿积

5.2.6 自然连接

跟积相比，人们更经常对两个关系做连接（join）操作，连接时相应的元组必须在某些方面一致。最简单的就是所谓的自然连接（natural join）。关系R和S的自然连接表示为$R \bowtie S$。此操作仅仅把在R和S模式中有某共同属性，且此属性有相同的值的元组配对。举例来说：假设R和S的模式有公共属性A_1，A_2，…，A_n，r和s是分别来自R和S的元组，则当且仅当r和s的A_1，A_2，…，A_n属性值都一样时，r和s才能配对，作为结果关系中的元组。

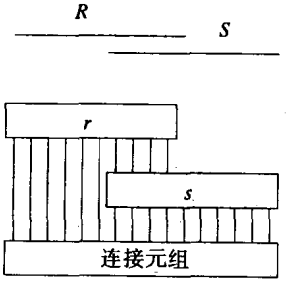


图5-5 连接两个元组

如果把r和s连接作为$R \bowtie S$结果的元组，则这个元组被称为连接元组（joined tuple）。连接元组具有R和S连接的所有成分。连接之后的元组跟元组r在模式R的所有属性上有相同值，同样，跟s在模式S的所有属性上有相同的值。一旦r和s被成功配对，那么连接之后的元组跟原来的两个元组在相同的属性上有相同的值。这一点在图5-5中可以清楚看到。

同样要注意，这里所说的连接操作跟3.6.5节用来重组关系的操作一样。后者重组的是一

个被映射到不同属性集上的关系。当时的目的是要解释为什么BCNF分解是有意义的。在5.2.8节中，还将要看到另一个用到自然连接的例子：联合两个关系，以便进行跟这两个关系都相关的查询。

198

例5.6 图5-4中关系R和S的自然连接是：

A	B	C	D
1	2	5	6
3	4	7	8

惟一一个R和S共同的属性是B。只要在属性B上相同的元组就可以成功连接。如果这样的话，结果元组中就有属性A（来自R），B（来自S或者R），C（来自S），D（来自S）。

在这个例子中，R的第一个元组成功地和S的第一个元组组成一对;他们在属性B上有共同的值2。这个配对产生了连接结果的第一个元组：（1，2，5，6）。关系R的第二个元组只可以跟S的第二个元组配对，结果是（3，4，7，8）。注意，关系S的第三个元组不能跟关系R的任何一个元组配对，所以不在结果$R \bowtie S$中出现。在一个连接当中，如果一个元组不能和另外关系中的任何一个元组配对的话，这个元组就被称为*悬挂元组*（dangling tuple）。□

例5.7 前面的例子并没有阐明自然连接操作所有可能的情况。例如，没有元组可以跟超过一个元组配对成功，并且两个关系模式只有一个共同属性。图5-6中，有另外两个关系U和V，他们的关系模式有共同的属性B和C。例子中，一个元组可以与另外几个元组连接。

只有在属性B和C上一致的元组才能配对成功。这样，U的第一个元组可以和V的第一和第二个元组相连接。而U的第二和第三个元组和V的第三个元组配对。这四个配对的结果在图5-6中给出。□

5.2.7 θ联接

自然连接必须根据某些特定的条件来把元组配对。虽然，相等的公共属性是关系连接最常见的基础。但是人们有时候需将满足其他条件的元组配对。为此目地，就有了相应的θ连接操作（theta-join）。历史上θ是指任意条件，但现在一般用C而不是θ表示这个条件。

关系R和关系S的θ联接可以这样用符号来表示：$R \underset{C}{\bowtie} S$。这个操作的结果是这样构造的：

1. 先得到R和S的积。
2. 在得到的关系中寻找满足条件C的元组。

就像在积操作中一样，结果关系的模式是模式R和模式S的并。如果有必要的话，需要在重名的属性前面加上“$R.$”或者“$S.$”。

例5.8 考虑这样的操作$U \underset{A<D}{\bowtie} V$。其中$U$和$V$是图5-6中的关系。这里必须考虑9个元组的配对方案。看一看来自U的元组的属性A是不是比来自V的元组的属性D的值小。U的第一个元组

A	B	C
1	2	3
6	7	8
9	7	8

关系 U

B	C	D
2	3	4
2	3	5
7	8	10

关系 V

A	B	C	D
1	2	3	4
1	2	3	5
6	7	8	10
9	7	8	10

$U \bowtie V$ 的结果

图5-6 关系的自然连接

199

A属性值是1，这个元组可以和任何来自V的元组配对。但是第二和第三个元组的A属性值分别是6和9，分别只能和V的最后一个元组配对。这样，所得到的结果关系就只有五个元组，即是刚才配对成功的那些。这个关系在图5-7中给出。 □

注意，连接的结果关系模式是由所有的六个属性组成。公共属性B和C前面加了关系名U和V以示区别，如图5-7所示。θ 连接和自然连接相比较，后者把公共属性合并成一个属性，这是由于参加运算的两个元组可以配对当且仅当它们的公共属性有相同的值。但是在 θ 连接当中，
200 并不能保证进行比较的属性有相同的值，因为它们有可能不是用“=”进行比较。

A	U.B	U.C	V.B	V.C	D
1	2	3	2	3	4
1	2	3	2	3	5
1	2	3	7	8	10
6	7	8	7	8	10
9	7	8	7	8	10

图5-7 $U \overset{\bowtie}{_{A<D}} V$的结果

例5.9 下面是关系U和V的更为复杂的 θ 连接，$U \overset{\bowtie}{_{A<D \text{ AND } U.B \neq V.B}} V$。结果不仅仅要求U关系元组的A属性小于V关系元组的D属性，而且U和V元组的B属性不能有同样的值。这里只有元组：

A	U.B	U.C	V.B	V.C	D
1	2	3	7	8	10

惟一满足两个条件，所以它就是最后的结果。 □

5.2.8 使用组合操作生成查询

如果只能在单个或者两个关系上进行一个操作，那么关系代数就不那么有用。可是，如同其他的所有代数一样，关系代数允许任意复杂的表达式，其操作符可以用于任何关系之上，这个关系既可以是某个给定关系，也可以是操作得到的结果关系。

可以在子表达式上应用算符来构造新的关系代数表达式，必要的时候用括号把操作数分割开。也可以用表达式树来表示这种表达式，虽然这种表达式树对机器来说比较难以处理，但是它更易于理解。

例5.10 考虑例3.24中分解的Movie关系。假设人们想知道“由Fox制作的片长至少100分钟的电影的名称（title）和制作年代（year）”。计算该查询的一种方法是：

1. 选择length ≥ 100分钟的 Movie关系中的
201 元组。
2. 选择studioName=“Fox”的Movie元组。
3. 计算(1)和(2)两个查询结果的交集。
4. 把（3）得到的关系投影到title和year属性上。

在图5-8中人们看到可以用表达式树来表示前面所说的几个步骤。两个选择操作的节点对应着第一和第二步。交操作的节点对应第三步，投影节点是第四步。

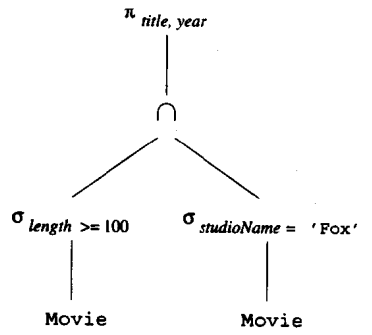


图5-8 关系代数表达式的表达式树

习惯上用线性符号来表示同样的表达式。如下公式

$$\pi_{title,\ year}(\sigma_{length\geqslant 100}(\text{Movie}) \cap \sigma_{studioName='\ Fox'}(\text{Movie}))$$

表示的是同一个表达式。

经常地，同一个计算可用多个不同的关系代数表达式来描述。例如，上边的查询可以用逻辑AND操作来替换“交”。即

$$\pi_{title,year}(\sigma_{length\geqslant 100\ \text{AND}\ studioName='\ Fox'}(\text{Movie}))$$

□

等价的表达式和查询优化

所有数据库系统都有查询应答系统，其中许多是基于接近关系代数表达能力的语言。这样，用户提交的查询可以有许多等价表达式（equivalent expression）（即只要给出同样的操作数，这些表达式就产生同样的结果），其中有的表达式能更快地被计算出结果。在1.2.5节中简要讨论过的查询“优化器”的重要作用就是把某个关系代数表达式替换为更加有效计算的等价形式。

例5.11 连接操作的一个应用就是将分解为BCNF的关系重新组合。回想例3.24中被分解的关系[⊖]。 202

Movie1有模式：{title, year, length, filmType, studioName}

Movie2有模式：{title, year, starName}

现在考虑编写查询“超过100分钟长的电影中的影星是谁？”的查询表达式。这个查询与Movie2的starName属性和Movie1的length属性有关。通过连接操作可以把两个关系的属性连接起来。自然连接只对在title和year属性上相同的元组才能成功配对，也就是说，仅仅那些代表同一个电影的元组才可以配对。于是，关系代数表达式Movie1 ⋈ Movie2的结果就是在例3.24中被称做Movie的关系。这是个非BCNF关系，其模式有六个属性，并且当有多个影星参与同一部影片时，该关系用多个元组来表示。

对于Movie1和Movie2连接后的关系还要作选择操作，选出那些长于100分钟的元组。最后再将其投影到人们感兴趣的属性starName上。于是表达式：

$$\pi_{starName}(\sigma_{length\geqslant 100}(\text{Movie1} \bowtie \text{Movie2}))$$

用关系代数实现了人们期望的查询。 □

5.2.9 重命名

关系代数操作的结果也是一个关系。为了有效管理生成的结果关系的属性名字，重命名操作就非常必要。算符$\rho_S(A_1, A_2, \ldots, A_n)(R)$表示对关系$R$重新命名。重命名后的关系与关系$R$有完全相同的元组，只不过关系的名字变成了$S$。另外，$S$关系的各个属性分别命名为$A_1, A_2, \ldots, A_n$，按从左到右的顺序排列。如果只是想把关系$R$的名字改变为$S$，并不改变其中属性的名字，就可以简单的使用$\rho_S(R)$即可。 203

例5.12 在例5.5中曾经对图5-4中的关系R和S进行积操作，并约定当两个关系存在同名属性时，分别把关系的名字作为结果属性名称的前缀。为方便起见，在图5-9中列出这两个关系。

⊖ 记住，该例中关系Movie的模式与5.1节中介绍并在例5.2中使用的关系Movie的模式有点不同。

假如不希望把两个属性B称做$R.B$或者是$S.B$，而是仍然希望来自R的属性B保持原来的名称，来自S的属性B改为X。操作$\rho_{S(X,\ C,\ D)}(S)$的结果是一个名为S的关系，它看起来与S很相似。惟一的不同是此关系的第一个属性名是X而不是B。

A	B
1	2
3	4

关系R

B	C	D
2	5	6
4	7	8
9	10	11

关系S

A	B	X	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

结果 $R \times \rho_{S(X,\ C,\ D)}(S)$

图5-9 在积运算之前重命名

用重命名后的关系和关系R进行积操作就不会有任何命名冲突了，也就不需要再进一步更改属性名。$R \times \rho_{S(X,\ C,\ D)}(S)$的结果与图5-4中$R \times S$的结果相同，只是结果的列标题从左到右分别
204 是：A，B，X，C，D。图5-9中示出了这个关系。

人们也可以不进行重命名就直接进行积操作，就像在例5.5中所做的那样，到最后再对结果进行重命名。表达式$\rho_{RS(A,\ B,\ X,\ C,\ D)}(R \times S)$可以产生跟图5-9同样的结果，包括其中属性的名字。但是这次关系有了新的名字——RS，而图5-9中的关系没有名字。 □

5.2.10 依赖的和非依赖的操作

在5.2节中介绍的操作，有一些可以用其他关系代数式来表达。例如，交运算可以如下表示：

$$R \cap S = R - (R - S)$$

意思就是，如果R和S是任意的两个关系，他们有同样的模式，那么R和S的交运算可以通过如下步骤实现：将S从R中减去，形成了一个新的关系T，它包含那些在R中但是不在S中出现的元组。然后再从R中将T减去，这样就只剩下那些既在R中又在S中出现的元组了。

两种形式的连接同样可以用其他表达式来实现。比如θ 连接，可以用积操作和选择操作来实现：

$$R \overset{\bowtie}{C} S = \sigma_C (R \times S)$$

自然连接可以通过先做一个积操作，然后再按如下形式的条件C进行选择来实现：

$$R.A_1 = S.A_1 \text{ AND } R.A_2 = S.A_2 \text{ AND } \ldots \text{ AND } R.A_n = S.A_n$$

这里，A_1，A_2，...，A_n是所有同时出现在R和S中的属性。最后，对于相同的属性必须投影出一份拷贝。令L是所有R中的属性和在S中但不在R中的属性的列表，那么

$$R \bowtie S = \pi_L(\sigma_C(R \times S))$$

例5.13 图5-6中U和V的自然连接可以写成含有积、选择、投影的式子：

$$\pi_{A,U.B,U.C,D}(\sigma_{U.B = V.B \text{ AND } U.C = V.C}(U \times V))$$

也就是说先得到积$U \times V$。然后在其中寻找使得B，C属性相等的元组。最后，将其投影到除去一个B和一个C之外的所有属性上。这里是选择除去关系V和U中同时出现的属性。

另外一个例子是关于图5-9中的θ连接，它可以写成：

$$\sigma_{A < D \text{ AND } U.B \neq V.B}(U \times V)$$

205

也就是说，先进行积操作，然后用θ连接中的条件进行选择。 □

本节中谈到的重写规则是已介绍的所有操作中惟一的“冗余”。其余六个操作：连接、差、选择、投影、积运算和重命名构成了一个独立的集合，这个集合中的每一个操作都不能被这个集合中余下的操作来实现。

5.2.11 关系代数表达式中的线性符号

在5.2.8节，曾用树表示复杂的关系代数表达式。另外的方法就是生成若干临时关系，用来表示树的内节点，并写一系列的赋值语句使其具有正确的值。这些赋值语句的顺序可变，只要保证在对节点N赋值之前已经对节点N的节点赋值完毕就可以。

在赋值过程中用到的符号有：

1. 关系的名字和用括号括起的关系属性的列表。名字Answer习惯上表示最后一步运算的结果。也就是在表达式树根结点上的关系名。

2. 赋值符号：=。

3. 写在赋值号右边的任何代数表达式。可以采用每个赋值语句只用一个算符的方法，这样每一个内部结点都有自己对应的赋值语句。可是，如果方便的话，仍然可以把几个代数运算组合到一起写在表达式的右端。

例5.14 考虑图5-8中的树，计算该表达式的一个可能的结果赋值序列如下：

```
R(t,y,l,i,s,p) := σ_length≥100(Movie)
S(t,y,l,i,s,p) := σ_studioName='Fox'(Movie)
T(t,y,l,i,s,p) := R ∩ S
Answer(title, year) := π_t,i(T)
```

第一步，计算图5-8中标号为$\sigma_{length \geq 100}$的内部节点关系，第二步计算标号是$\sigma_{studioName='Fox'}$的节点关系。注意，因为只要人们愿意就可以对关系的左边使用任何属性和关系的名字，所以可以自由重命名。最后两步显然是进行交运算和投影运算。

如前所述，可以对某些操作步骤进行合并。例如，可以对最后两步进行合并，写成：

```
R(t,y,l,i,s,p) := σ_length≥100(Movie)
S(t,y,l,i,s,p) := σ_studioName='Fox'(Movie)
Answer(title, year) := π_t,i(R ∩ S)
```

□ 206

5.2.12 习题

习题5.2.1 在这个习题中，介绍一个正在运行的关系数据库模式的例子和一些示例数据[⊖]。

⊖ 来源：厂商的网页和Amazon.com。

数据库模式由四个关系组成，这四个关系的模式是：

```
Product(maker, model, type)
PC(model, speed, ram, hd, rd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

关系Product给出了各种产品的制造厂商，型号，类型（PC、手提电脑或者打印机）等。这里简单假设所有产品的型号都惟一，而不管它是由哪个制造商生产的。这个假设有些不符合实际情况，真实的数据库中，制造商的号码应该作为型号的一部分。关系PC对于不同型号给出了如下属性：速度（处理器的速度，单位是MHz）、RAM的容量（单位是MB），硬盘的容量（单位是GB）、光盘驱动器的速度和型号（可能是CD也可能是DVD）、价格等。关系Laptop也类似，除了把移动硬盘属性换成了显示器尺寸外，没有变化。关系Printer对于每种模型，有如下属性：是否彩色（如果是的话，这个值是true）、处理类型（激光的还是喷墨的，或者是点阵的）、价格。

关系Product的一些数据的例子在图5-10中给出。另外三个关系的样例数据在图5-11中给出。生产厂商和型号被忽略了。这些数据只能反映是2001年年初的行情。

试写出下列查询的表达式。可以用5.2.11节介绍的线性算符来写这些表达式。针对图5-10和图5-11的数据，给出查询的结果。不过，你的答案应该在任何数据上都能正确工作，而不仅限于图中的数据。

maker	*model*	*type*
A	1001	pc
A	1002	pc
A	1003	pc
A	2004	laptop
A	2005	laptop
A	2006	laptop
B	1004	pc
B	1005	pc
B	1006	pc
B	2001	laptop
B	2002	laptop
B	2003	laptop
C	1007	pc
C	1008	pc
C	2008	laptop
C	2009	laptop
C	3002	printer
C	3003	printer
C	3006	printer
D	1009	pc
D	1010	pc
D	1011	pc
D	2007	laptop
E	1012	pc
E	1013	pc
E	2010	laptop
F	3001	printer
F	3004	printer
G	3005	printer
H	3007	printer

图5-10 关系Product的数据取样

model	*speed*	*ram*	*hd*	*rd*	*price*
1001	700	64	10	48xCD	799
1002	1500	128	60	12xDVD	2499
1003	866	128	20	8xDVD	1999
1004	866	64	10	12xDVD	999
1005	1000	128	20	12xDVD	1499
1006	1300	256	40	16xDVD	2119
1007	1400	128	80	12xDVD	2299
1008	700	64	30	24xCD	999
1009	1200	128	80	16xDVD	1699
1010	750	64	30	40xCD	699
1011	1100	128	60	16xDVD	1299
1012	350	64	7	48xCD	799
1013	733	256	60	12xDVD	2499

（a）关系PC的数据取样

model	*speed*	*ram*	*hd*	*screen*	*price*
2001	700	64	5	12.1	1448
2002	800	96	10	15.1	2584
2003	850	64	10	15.1	2738
2004	550	32	5	12.1	999
2005	600	64	6	12.1	2399
2006	800	96	20	15.7	2999
2007	850	128	20	15.0	3099
2008	650	64	10	12.1	1249
2009	750	256	20	15.1	2599
2010	366	64	10	12.1	1499

（b）关系Laptop的数据取样

model	*color*	*type*	*price*
3001	true	ink-jet	231
3002	true	ink-jet	267
3003	false	laser	390
3004	true	ink-jet	439
3005	true	bubble	200
3006	true	laser	1999
3007	false	laser	350

（c）关系Printer的数据取样

图5-11 练习5.2.1的样例数据

* a) 哪一种型号的PC速度大于1000?

b) 哪一个厂商生产的所有手提电脑有大于1G的硬盘?

c) 查询厂商*B*生产的所有产品的型号和价格。

d) 查询所有激光打印机的型号。

e) 查询那些只出售手提电脑、不出售PC的厂商。

*! f) 查询在一种或者两种PC机中都具有的硬盘的容量。

! g) 查询有同样处理速度和同样内存大小的PC对。每对只列一次，即列表给出（*i*,*j*）但不给出（*j*,*i*）。

*!! h) 查询那些至少生产两种处理速度大于700的PC或者手提电脑的厂商。

!! i) 查询平均处理速度（PC或者是手提电脑）最高的厂商。

207 ~ 209

!! j) 查询至少生产三种不同处理速度电脑的厂商。

!!k) 查询恰好出售三种型号PC的厂商。

习题5.2.2 画出上题中每个查询的表达式树。

习题5.2.3 用5.2.11节的线性算符重写习题5.2.1中的那些查询。

习题5.2.4 这个习题引入了另外的一个连续的实例。涉及二战中的大型舰船。它由以下几个关系组成：

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

相同设计的舰船组成一个“类”，类别的名称通常就是这个类的第一艘船的名字。关系Classes记录了“类”的名字、型号（bb代表主力舰，bc代表巡洋舰）、生产国家、火炮的门数、火炮尺寸（口径，单位是英寸）和排水量（重量，单位是吨）。关系Ships记录了舰船的名字、舰船类属名字、开始服役的日期。关系Battles给出了这些舰船参加的战役的时间。关系Outcomes给出了各个舰船在各场战役中的结果（是沉没，还是受伤，或者完好）。

图5-12和5-13给出了这四个关系的数据取样[⊖]。需要注意的是，跟习题5.2.1不同，在这个习题中存在着“悬挂元组”，比如，在关系Outcomes中出现的船只可能在关系Ships中查不到。

编写实现下面这些查询的关系代数表达式。对照图5-12和图5-13给出查询结果。可是，你的答案应该不只是对图中的数据有效，应该对任意的数据都有效。

210

a) 查询那些火炮口径至少十六英寸的舰船所属类的名称和拥有此类舰船的国家。

b) 查询那些在1921年之前服役的舰船。

c) 查询在北大西洋战役中沉没的舰船。

d) 在1921年签署的华盛顿条约禁止制造超过35 000吨的大型军舰，请列出那些违反华盛顿条约的军舰。

e) 列出参加了Guadalcanal岛海战的舰船的名称、排水量及火炮的数目。

f) 列出所有的在此数据库中提到的舰船。

! g) 列出只包含一艘舰船的类。

! h) 列出那些既有主力舰又有巡洋舰的国家。

! i) “留得青山在，不怕没柴烧”，列出那些在某次战役中受伤、但是又参加了其他战役的舰船。

习题5.2.5 画出习题5.2.4中的查询表达式树。

习题5.2.6 把习题5.2.4中的每个表达式用5.2.11节介绍的线性表达式重写。

* **习题5.2.7** 连接$R \bowtie S$和自然连接$R \overset{\bowtie}{C} S$的区别是什么？（这里，条件C为$R.A = S.A$。其中A是任何一个同时出现在R和S模式中的属性。）

! **习题 5.2.8** 一个关系操作符被称做是单调的，当且仅当对其运算参数增加任何元组时，其结果中始终至少包含那些原有的元组（也可能包含其他更多元组）。本节中的几个操作符中，哪个是单调操作符？对于每个操作符，如果是单调的，给出理由；如果不是单调的，

⊖ 来源：J. N. Westwood, *Fighting Ships of World War II*, Follett Publishing, Chicago, 1975 and R. C. Stern, *US Battleships in Action*, Squadron/Signal Publications, Carrollton, TX. 1980.

举例说明原因。

! **习题5.2.9** 假设关系R和关系S各有n个和m个元组。对于下面的表达式，给出结果中可能出现的最多和最少的元组数目。

* a) $R \cup S$。

b) $R \bowtie S$。

c) $\pi_C(R) \times S$，对于某个条件C。

d) $\pi_L(R) - S$，对于某个属性列表L。

class	*type*	*country*	*numGuns*	*bore*	*displacement*
Bismarck	bb	Germany	8	15	42000
Iowa	bb	USA	9	16	46000
Kongo	bc	Japan	8	14	32000
North Carolina	bb	USA	9	16	37000
Renown	bc	Gt. Britain	6	15	32000
Revenge	bb	Gt. Britain	8	15	29000
Tennessee	bb	USA	12	14	32000
Yamato	bb	Japan	9	18	65000

(a) 关系Classes的数据取样

name	*date*
North Atlantic	5/24-27/41
Guadalcanal	11/15/42
North Cape	12/26/43
Surigao Strait	10/25/44

(b) 关系Battles的数据取样

ship	*battle*	*result*
Bismarck	North Atlantic	sunk
California	Surigao Strait	ok
Duke of York	North Cape	ok
Fuso	Surigao Strait	sunk
Hood	North Atlantic	sunk
King George V	North Atlantic	ok
Kirishima	Guadalcanal	sunk
Prince of Wales	North Atlantic	damaged
Rodney	North Atlantic	ok
Scharnhorst	North Cape	sunk
South Dakota	Guadalcanal	damaged
Tennessee	Surigao Strait	ok
Washington	Guadalcanal	ok
West Virginia	Surigao Strait	ok
Yamashiro	Surigao Strait	sunk

(c) 关系Outcomes的数据取样

图5-12 习题5.2.4的数据

*! **习题5.2.10** 关系R和S的半连接（semijoin）写作$R \ltimes S$，它表示由R中的满足如下条件的元组t组成的包：t至少跟S中的一个元组在R和S的公共属性上相同。用关系代数表达式给出$R \ltimes S$的等价表示。

! **习题5.2.11** 反半连接（antisemijoin）$R \overline{\ltimes} S$是由R中的元组t构成的包：t在R和S的公共属性上的值，不能跟S中的任何元组相等。给出一个等价于$R \overline{\ltimes} S$的关系代数表达式。

!! **习题5.2.12** 假设R是具有如下模式的关系：

$$(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$$

关系S具有如下模式（B_1，B_2，...，B_m）；它的属性是关系R的属性的子集。R和S的商，记做$R \div S$，是具有属性A_1，A_2，...，A_n（即那些是R中的属性但却不是S中的属性）的元组集，使得对于S中的任何元组，元组ts（ts中包括t中A_1，A_2，...，A_n属性的值和s中$B_1, B_2, \ldots, B_m$属性的值）都是关系R中的元组。用前面已经定义过的关系操作算符写出与$R \div S$等价的关系代数表达式。

name	*class*	*launched*
California	Tennessee	1921
Haruna	Kongo	1915
Hiei	Kongo	1914
Iowa	Iowa	1943
Kirishina	Kongo	1915
Kongo	Kongo	1913
Missouri	Iowa	1944
Musashi	Yamato	1942
New Jersey	Iowa	1943
North Carolina	North Carolina	1941
Ramillies	Revenge	1917
Renown	Renown	1916
Repulse	Renown	1916
Resolution	Revenge	1916
Revenge	Revenge	1916
Royal Oak	Revenge	1916
Royal Sovereign	Revenge	1916
Tennessee	Tennessee	1920
Washington	North Carolina	1941
Wisconsin	Iowa	1944
Yamato	Yamato	1941

211
~
213

图5-13 关系Ships的数据取样

5.3 包上的关系操作

虽然元组集合（即关系）像它在数据库中一样，是一个简单的自然的基于集合的数据模型，但是在商业数据库系统中，很少是完全基于集合。在某些情况下，关系允许相同的元组出现多次，就像在数据库系统当中那样。当一个集合允许同样的元组出现多次的时候，它通常被称为包（Bag）或者多集（multiset）。这一节我们将讨论基于包的关系，也就是说，允许同样的元组在一个关系当中出现多次。当提到集合的时候，就意味着一个没有重复元组的关系，而包则意味着可能有重复元组出现在同一个关系当中（当然，也可能没有）。

例5.15 图5-14中的关系就是一个基于包的实例。其中，元组(1，2)出现了三次，元组(3，4)出现了一次。如果图5-14表示的是一个基于集合的关系，将必须消去两个(1，2)元组。在基于包的关系当中允许重复元组出现，但是，与基于集合的关系一样，元组通常没有顺序。 □

A	*B*
1	2
3	4
1	2
1	2

图5-14 包

5.3.1 为什么采用包

当考虑高效实现一个关系的时候，如果采用基于包的关系的话，就会有好几种方法加速查

询。在5.2节的开始曾经提到过怎样通过允许结果是包来加快两个关系的并操作。再举一个例子，执行投影操作的时候，允许结果关系是包（不管原来的关系是否基于集合)就可以独立地处理每一个元组。如果想得到一个基于集合的结果，就必须将每一个元组跟其余所有的元组比较，以保证这个元组在关系中只出现一次。但是如果允许结果是一个包的话，就可以简单地对每个元素作投影，然后把它加入到结果当中，不用跟已经得到的其他元组比较。

例5.16 如果允许结果是包，并且不去掉重复元组(1，2)的话，那么把图5-15中的关系投影到A，B属性上就得到图5-14中所示的包。 214

A	B	C
1	2	5
3	4	6
1	2	7
1	2	8

图5-15 例5.16的包

如果使用通常意义上的投影操作，就不允许重复元组在结果中出现，那么结果是

A	B
1	2
3	4

注意，尽管基于包的结果会大一些，可计算起来却比较快，因为不用把元组(1，2)、(3，4)跟已经得到的那些元组相比较。

另外，当对一个关系做投影操作，然后再进行“求A属性的平均值”这样的聚合操作时（将在5.4节讨论），不能采用基于集合的关系作为把原关系投影到A属性上的结果。例如在图5-15中，A属性的平均值是2，因为这个属性只有两个值：1和3。但是如果把图5-15中的A属性看成是包{1，3，1，1}的话，就得到了A属性的正确平均值——1.5。 □

5.3.2 包的并、交、差

当把两个包并起来时，要把相应元组出现的次数相加。就是说，如果R是一个包，其中元组t出现了n次；S也是一个包，其中元组t出现了m次。那么在$R \cup S$中，元组t出现了$n+m$次。注意，这里的n和m都可以是0。

当执行两个包R和S的交时，假定其中元组t分别出现了n次和m次，则在$R \cap S$中，元组t出现了$\min(n, m)$次。当计算$R-S$时，元组t在$R-S$中出现了$\max(0, n-m)$次。也就是说，如果元组t在R中出现的次数更多，则$R-S$中t出现的次数就是t在R中出现的次数，减去t在S中出现的次数。反之，如果t在S中出现的次数更多或者跟在R中出现的次数一样多，那么t在$R-S$中就不出现了。直观上，t在S中的每次出现，都抵消了它在R中的一次出现。

例5.17 R是一个跟图5-14中一样的基于包的关系，元组(1，2)出现了三次，元组(3，4)出现了一次。而S是如下的基于包的关系： 215

A	B
1	2
3	4
3	4
5	6

那么$R \cup S$就是这样的一个关系：其中元组(1，2)出现了四次（3个是由于其在R中的出现，

1个是由于在S中的出现）；元组(3，4)出现了三次，(5，6)出现了一次。

R和S的交$R \cap S$的结果是

A	B
1	2
3	4

这里(1，2)和(3，4)各出现了一次。就是说(1，2)在R中出现了三次，在S中出现了一次，min(3，1)=1，所以(1，2)在$R \cap S$中出现一次。(5，6)在S中出现一次，但是在R中没有出现，所以在$R \cap S$中出现min(0，1) = 0次。

基于包的$R-S$是

A	B
1	2
1	2

原因是元组(1，2)在R中出现了三次，在S中出现了一次，所以在R-S中出现max(0，3-1) =2次。元组(3，4)在R中出现了一次，在S中出现了两次，所以在R-S中出现max(0，1-2) =0次。R中没有其他的元组再出现了，所以$R-S$中也没有其他的元组了。

作为另一个例子，基于包的差$S-R$是包

A	B
3	4
5	6

这里，元组(3，4)出现了一次，因为这个元组在R中出现的次数减去它在S中出现的次数是1。(5，6)在$S-R$中出现了一次也是同样的原因。这里，其结果包恰好是一个集合。 □

5.3.3 包的投影操作

上面已经解释了包上的投影操作。在例5.16中，每个元组在投影操作时被独立处理。如果
216 R是如图5-15的关系，做包投影操作$\pi_{A,B}(R)$时得到的将是如图5-14的关系。

在集合上的包操作

假设现有两个集合R和S。它们可以设想成恰好是没有重复的元组的包。假定做交操作：$R \cap S$，并且是运用包操作规则来计算。这个操作的结果就与跟我们把R和S看成是集合的结果一样。也就是说，把R和S看成是包，$R \cap S$结果中的元组t的个数是t在R和S中出现的个数较小的那个。但是，当R和S都是集合的时候，那么结果中t在每个集合中出现次数只能是0或1。无论用包还是集合的规则，结果中t出现的次数至多一次，如果t在R和S中都出现的话，那么结果中V出现一次。相似地，如果运用包上的差规则来计算$R-S$或者$S-R$，其结果与跟利用集合上的差规则来计算同样的表达式得到的结果相同。

但是，并操作的情况就不是这样了，结果依赖于把R，S看成是包还是集合。如果用包的规则来计算$R \cup S$，就算R和S都是集合的话，结果也不一定是集合。例如，元组t在R和S中各出现一次，运用包规则，那么在结果中V出现两次。但是如果运用集合规则，t在结果中只出现一次。这样，进行并操作的时候，就必须特别小心的弄明白是用包规则还是用集合规则。

如果在投影操作过程中，除去了一个或者多个属性后，产生了多个同样的元组，那些重复的元组将不会被从包投影结果中除去。来自图5-15的三个元组(1，2，5)、(1，2，7)、(1，2，

8)在投影操作之后，都给出了同样的结果(1，2)。在结果包当中，元组(1，2)出现了三次，但在集合投影操作上，这个元组仅出现一次。

5.3.4 包的选择

在包上应用选择操作的时候，要独立地测试每个元组。就像对包所作的其他操作一样，在结果中不去掉重复元组。

例5.18 如果R是包

A	B	C
1	2	5
3	4	6
1	2	7
1	2	7

那么包选择$\sigma_{C\geq 6}(R)$的结果是 217

A	B	C
3	4	6
1	2	7
1	2	7

除了第一个元组之外，其余的都满足条件。最后两个元组是重复的，都在结果当中。 □

关于包的代数定律

一个代数定律就是两个关系代数表达式之间的恒等式，其中的参数是表示关系的变量。无论用什么样的关系去代替等式中的变量，等式都依然成立。一个例子是并操作的交换率：$R\cup S=S\cup R$。这个定律无论在R和S是包还是集合时都成立。但是有很多定律只适用于集合的情况。一个简单的例子是：集合差对于并的分配率，$(R\cup S)-T = (R-T)\cup (S-T)$。这个定律只适用于集合而不适用于$R$和$S$是包的情况。假设$R$和$S$，还有$T$都含有元组$t$。那么左边的表达式有一个$t$在结果中，但是右边的表达式结果中没有$t$。如果作为集合来考虑的话，那么结果中都没有$t$。在包情况下一些关系代数表达式的例子将在习题5.3.4和5.3.5中讨论。

5.3.5 包的笛卡儿积

包的笛卡儿积的规则正如所想像的那样。一个关系中的每个元组跟另外一个关系中的每个元组配对，而不问这个元组是不是重复出现。结果是，如果元组r在关系R中出现了m次，元组s在关系S中出现了n次，那么元组rs在$R\times S$中将出现mn次。

例5.19 关系R和S在图5-16中给出。乘积$R\times S$包括六个元组，就像在图5-16中那样。注意，对于属性名的约定，在基于包的情况与以前基于集合的情况完全相同。这样的话，同时属于R和S两个关系的属性B，在积中出现两次，于是属性的前面要加上关系名的前缀。 □ 218

5.3.6 包的连接

连接包的操作也跟预想的一样。首先对比两个关系当中的元组，看是不是能组成一对，如果可以的话，这个配对起来的元组就是结果中的一员。当产生结果的时候，不需要去掉重复元组。

例5.20 图5-16中的关系R和S的自然连接结果是：

A	B	C
1	2	3
1	2	3

元组(1，2)跟S中的元组(2，3)连接。因为在R中有两个(1，2)元组，在S中有一个(2，3)元
219 组，这样就有两个配对成功的元组对得到(1，2，3)。其他R和S中的元组均没有成功的连接。

另一个关于关系R和关系S的例子是θ连接

$$R \underset{R.B<S.B}{\bowtie} S$$

将包进行积操作

A	$R.B$	$S.B$	C
1	2	4	5
1	2	4	5
1	2	4	5
1	2	4	5

连接运算如下。来自R的元组(1，2)和来自S的元组(4，5)满足连接的条件。因为他们都在各自的关系当中出现了两次，则连接后的元组出现了$2 \times 2 = 4$次。另外可能连接结果是R的元组(1，2)和S的元组(2，3)，但是这个不满足连接条件，所以不在结果当中。 □

A	B
1	2
1	2

(a) 关系R

B	C
2	3
4	5
4	5

(b) 关系S

A	$R.B$	$S.B$	C
1	2	2	3
1	2	2	3
1	2	4	5
1	2	4	5
1	2	4	5
1	2	4	5

(c) $R \times S$的结果

图5-16 计算包的乘积

5.3.7 习题

* **习题5.3.1** PC是图5-11中的关系，假设要计算投影π_{speed}(PC)。分别给出用集合和用包表示的结果值，以及该投影的平均值。

习题5.3.2 对习题5.3.1的情况计算投影π_{hd}(PC)。

习题5.3.3 该习题参照习题5.2.4中的舰船模式。

a) 表达式π_{bore}(Classes)产生了一个单列的关系，当把结果看成是包和集合时，分别给出这个关系。

! b) 给出所有船只的火炮口径(不是船只类属)。你的表达式必须是对包有意义，也就是

说，一个值b出现的次数必须等于具有这个值的火炮口径的船只数目。

! **习题5.3.4** 一些对集合有效的代数运算规则对包也同样有效。解释下面的规则为什么对包和集合都有效。

* a) 对于并的结合律：$(R \cup S) \cup T = R \cup (S \cup T)$

b) 对于交的结合律：$(R \cap S) \cap T = R \cap (S \cap T)$

c) 对于自然连接的结合律：$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$ 220

d) 并的交换律：$R \cup S = S \cup R$

e) 交的交换律：$R \cap S = S \cap R$

f) 自然连接的交换律：$R \bowtie S = S \bowtie R$

g) $\pi_L(R \cup S) = \pi_L(R) \cup \sigma_L(S)$ 这里L是任意的属性列表。

* h) 并对于交的分配定律：$R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$

i) $\sigma_{C \text{ AND } D}(R) = \sigma_C(R) \cap \sigma_D(R)$ 这里，C、D是任意的对于R中元组的条件。

!! **习题5.3.5** 下面这些代数规则仅适用于集合，不适用于包，请说明为什么他们适用于集合，并举出不适用于包的反例。

* a) $(R \cap S) - T = R \cap (S - T)$

b) 交对于并的分配定律：$R \cap (S \cup T) = (R \cap S) \cup (R \cap T)$

c) $\sigma_{C \text{ OR } D}(R) = \sigma_C(R) \cup \sigma_D(R)$

5.4 关系代数的扩展操作

在5.2节中介绍了经典的关系代数，5.3节介绍了基于包的关系的一些必要的改动。这些内容形成了现代查询语言的基础。但是像SQL这样的语言还有许多其他的操作，这些在应用中更为重要。因此，这一节将全面介绍关系代数的其他操作。增加的内容有：

1. *消重复操作符*δ把包中的重复元素都去掉，只保留一个副本在关系当中。

2. *聚集操作*（aggregation operator）。例如求和或者求平均值。这些不是关系代数的操作但却是被*分组*（grouping）操作所使用（下面会讲到）。当聚集操作应用到某个关系的属性时，比如说是sum操作，就把这一列的所有值加起来求和计算出结果。

3. 根据元组的值对他们在一个或者多个属性上*分组*（grouping），有把某个关系的元组分成“组”的效果。这样，聚集操作就可以对分好组的各个列进行计算。这给我们提供了在经典 221
的关系代数表达式中不能表达的方式来描述许多查询。分组算符γ是组合了分组和聚集操作的一个算符。

4. *排序算符* τ把一个关系变成一个元组的列表，根据一个或者多个属性来排序。这个操作使用时要心中有数，因为其他的关系代数操作都是对集合或者包进行操作，所以这个操作一般作为一系列操作的最后一个来使用。

5. *扩展投影*（extended projection）在普通π操作上增加了一些功能。它可以以原有的列作为参数来进行计算，并产生新的列。

6. *外连接算符*（outerjoin operator）是连结算符的变体，它防止了悬挂元组的出现。在外连接的结果中，悬挂元组用null补齐，这样悬挂元组就可以在结果中被表示出来。

5.4.1 消除重复

有时候，需要用一个算符把包转化为集合。为此目的，用$\delta(R)$来返回一个没有重复元组的集合。

例5.21 如果R关系是：

A	B
1	2
3	4
1	2
1	2

那么，$\delta(R)$为

A	B
1	2
3	4

注意元组(1，2)在R中出现了三次，但是在$\delta(R)$中仅出现了一次。 □

5.4.2 聚集操作符

有许多应用在内含原子值的集合或包上的操作符。这些算符被用来汇总或者“聚集”关系
222 某一列中出现的值，所以被称为聚集操作。这些操作的标准算符是：

1. SUM用来产生一列的总和，得到的是一个数字值。
2. AVG用来产生一列的平均值，结果也是数字值。
3. MIN和MAX，当用于数字值列的时候，产生的分别是这一列中最小的和最大的值;当应用于字符值列的时候，产生的是字典序的最大者和最小者。
4. COUNT产生一列中的“值”的数目（并不一定指不同的值）。同样，COUNT应用于一个关系的任何一个属性时，产生的是这个关系的元组数，包括重复的元组。

例5.22 考虑关系：

A	B
1	2
3	4
1	2
1	2

一些应用在这个关系属性上聚集操作的例子是：

1. SUM(B)=2+4+2+2=10。
2. AVG(A)=(1+3+1+1)/4=1.5。
3. MIN(A)=1。
4. MAX(B)=4。
5. COUNT(A)=4。

□

5.4.3 分组

通常，人们不仅希望简单的对一整列求平均值或者是其他的聚集操作，还需要按照其他某一属性分组，然后考虑各分组中元组的聚集操作。比如，要计算每一个制片厂出品的电影总长度是多少分钟，其关系是：

studio	*sumOfLengths*
Disney	12345
MGM	54321
...	...

223

从关系Movie(title, year, length, inColor, studioName, producerC#)开始，对于5.1节中的数据库例子，先必须根据属性studioName的值把元组分组。然后计算每一组中length属性值的和。可以把Movie的元组想像成图5-17中那样，然后对每一组应用操作SUM(length)。

	studioName	
	Disney Disney Disney	
	MGM MGM	
	○ ○ ○	

图5-17 想像把一个关系分组

5.4.4 分组操作符

现在介绍一个允许把关系分组和（或）聚集的算符。如果有分组，那么聚集操作就在组中进行。

算符γ的下标是一个元素的列表L，其中每一个元素是下面情况之一：

a) 是应用γ操作的那个关系的一个属性，这个属性是那些可以把R分组的属性其中之一。这个元素就被称为是分组属性。

b) 应用到关系的一个属性上的聚集操作符。为了在结果中对应此聚集，给属性一个名称，一个箭头和一个新的名字附在这个聚集的后面。加了下划线的属性被称做是聚集属性。

表达式$\gamma_L(R)$返回的关系是这样产生的：

1. 把关系R的元组分组。每一组由具有L中分组属性为特定赋值的所有元组构成。
2. 对于每一组，产生一个如下内容的元组：

224

i. 那个组的分组属性值。

ii. 本组中所有元组对列表L的属性聚集操作的结果。

δ是特除情况

技术上讲，δ操作是冗余操作。如果$R(A_1, A_2, \ldots, A_n)$是关系，则$\delta(R)$等价于$\gamma_{A_1, A_2, \cdots, A_n}(R)$。也就是说，为了消除重复，用关系的所有属性分组，但是没有聚集操作。于是，每一组只有一个元组对应关系R中一次或多次出现的元组。由于γ中每一组只含有一个元组，该分组的效果就是消除重复。可是，因为δ操作很普遍和重要，所以在研究代数定律和操作符实现算法时仍然将δ单独考虑。

也可以把γ看做是集合上投影操作的扩展。也就是说，如果R是集合，$\gamma_{A_1, A_2, \cdots, A_n}(R)$与$\pi_{A_1, A_2, \cdots, A_n}(R)$相同。可是如果$R$是包，那么$\gamma$操作将消除$\pi$操作结果中的重复。因此，$\gamma$操作常常被看做是广义投影（generalized projection）。

例5.23 假设有如下的关系：

StarsIn(title, year, starName)

现想找出那些至少出演了三部电影的影星，以及他们最先出演的电影的拍摄时间。第一步就是分组，以starName作为分组的条件。显然，需要对每一组计算MIN(year)。但是，为

了确定满足至少出演三部电影的条件，还需进行COUNT(title)的操作。

先从如下的分组表达式开始：

$$\gamma_{starName,\ MIN(year)\rightarrow minYear,\ COUNT(title)\rightarrow ctTitle}(StarsIn)$$

此表达式结果的前两列是需要的，第三列是辅助属性，用ctTitle来标记。该列是用来确定一个影星是否在三部影片中出现。也就是说，在进行选择ctTitle>=3之后继续进行代数表达式的计算，把它投影到前两个属性上边。

225 这个查询的表达式树如图5-18所示。 □

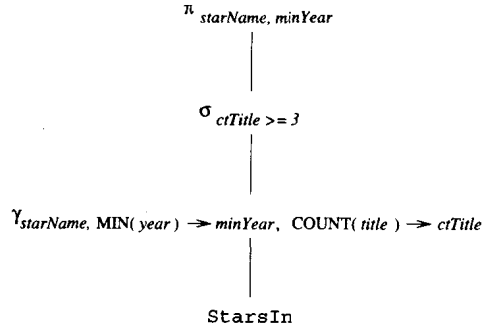


图5-18 例5.23中的查询的表达式树

5.4.5 扩展的投影操作符

现在重新考虑5.2.3节中介绍的投影操作$\pi_L(R)$。在经典的投影操作当中，L是R的一些属性的集合。扩展这个投影运算，使它支持在元组上的计算。仍然用$\pi_L(R)$来表示投影操作，其中，投影列表可以是以下所列出的元素之一：

1. R的一个属性。

2. 形如$x\rightarrow y$的表达式，其中，x和y都是属性的名字。$x\rightarrow y$表示把R中的x属性取来并重命名为y。

3. 形如$E\rightarrow z$的表达式，其中E是一个涉及R的属性、常量、代数运算符或者字符串运算符的表达式。z是表达式E得到的结果属性的新名字。例如，$a+b\rightarrow x$作为一个列表元素表示a和b属性的和，并重命名为x。元素$c||d\rightarrow e$表示连接字符串类型的属性c和d，并重命名为e。

投影操作的结果是通过依次考虑R的每一个元组得到。用元组中的成分代替L中相应的属性，并对其施以适当的运算。结果模式的属性名就是L中指定的名字。如果在R中有重复元组的话，结果当中肯定也有重复的元组，但是就算在R中没有重复的元组，在结果当中同样有可能产生重复元组。

226 **例5.24** 令R是如下的关系

A	B	C
0	1	2
0	1	2
3	4	5

那么$\pi_{A,\ B+C\rightarrow X}(R)$的结果是

A	X
0	3
0	3
3	9

结果模式有两个属性。一个是A，即R中的第一个属性，没有进行重命名。另一个是R中的第二个和第三个属性的和，被重命名为X。

另一个例子，$\pi_{B-A\rightarrow X,\ C-B\rightarrow Y}(R)$的结果是

X	Y
1	1
1	1
1	1

注意，这个投影操作的列表恰好对每一个元组产生了相同的结果，这样，元组(1，1)就在结果中出现了三次。 □

5.4.6 排序操作符

有很多时候人们希望对关系当中的元组按一个或者多个属性排序。通常，当进行查询的时候，希望结果是排好序的。例如，当查询Sean Connery所出演的电影的时候，可能希望结果是按着title的字典序排列，这样就可以很容易地找到关心的某一部电影。先对关系当中的元组排序可以提高DBMS的效率。

表达式$\tau_L(R)$，其中R是关系，L是R中某些属性的列表。这个表达式表示的就是关系R本身，只不过结果中的所有元组是按照L排序。如果L是这样的：A_1，A_2，...，A_n，那么R 的元组就先按属性A_1的值先排序，对于A_1属性相等的元组，就按A_2的值排序，依此类推。如果A_n属性也相同的话，则这些元组的顺序可以是任意的。

例5.25 如果R的模式是$R(A, B, C)$，那么$\tau_{C,B}(R)$就把R中的元组按着C 的值排序，对于C属性值相同的元组，以B属性的值确定他们的顺序。如果在B和C属性上都相同的话，这些元组之间的顺序可以是任意的。 □ 227

τ操作符特殊之处在于：它是关系代数中惟一一个结果是元组列表的操作符。这样，从查询的表达方面来讲，它只能作为一串操作的最后一个算符来使用时才有实际意义。因为，如果在τ之后还有另外一个关系操作，τ产生的结果就只能当作是一个包或者是集合，其中的元组也不再有序[⊖]。

5.4.7 外连接

连接操作的一个性质就是有可能产生*悬挂元组*。也就是说，这些元组不能跟另外关系的任何一个元组匹配。因为悬挂元组在结果中没有任何痕迹，所以这样的连接操作并不能完全反映原始关系的全部信息。在某些场合，比如某些商用系统中，一个连接的变种*外连接*（outerjoin）被广泛使用。

先来考虑自然连接的例子，其连接发生在两个关系中同值同名的属性上。外连接$R \overset{\circ}{\bowtie} S$开始进行的操作就是$R \bowtie S$，然后再把来自$R$或者$S$的悬浮元组加入其中。用*null*的表示符号$\perp$补齐结果元组中那些不具有值的属性[⊖]。

例5.26 图5-19中有两个关系U和V。U的元组(1，2，3)可以和V的元组(2，3，10)、(2，3，11)连接，那么这三个元组是不悬浮的。但是，另外三个元组——来自U的(4，5，6)、(7，8，9)和来自V的(6，7，12)——都是悬浮的。这三个元组没有匹配的元组。这样，在$U \overset{\circ}{\bowtie} V$中，三个悬浮元组中没有值的属性被赋予$\perp$，它们是$U$元组对应的$D$属性和$V$元组对应的$A$属性。 □

基本的外连接有几种不同的变体。*左外连接*（left outerjoin）$R \overset{\circ}{\bowtie}_L S$类似于外连接，只是将左变量$R$的悬浮元组补齐$\perp$加入到结果中。类似地，*右外连接*(right outerjoin) $R \overset{\circ}{\bowtie}_R S$中，右变量$S$的悬浮元组用$\perp$补齐加入结果。

例5.27 如果U和V是如图5-19所示，那么$U \overset{\circ}{\bowtie}_L V$的结果是：

A	B	C	D
1	2	3	10
1	2	3	11
4	5	6	$\perp$
7	8	9	$\perp$

228

⊖ 然而，如果对中间结果进行排序的话，有时会加快查询速度。

⊖ 在学习SQL语言时，将会看到空值符号被写成NULL。这里如果你想用NULL代替$\perp$的话也是可以的。

$U ⟕_R V$的结果是

A	B	C	D
1	2	3	10
1	2	3	11
⊥	6	7	12

□

A	B	C
1	2	3
4	5	6
7	8	9

关系U

B	C	D
2	3	10
2	3	11
6	7	12

关系V

A	B	C	D
1	2	3	10
1	2	3	11
4	5	6	⊥
7	8	9	⊥
⊥	6	7	12

结果 $U ⟕ V$

图5-19 关系的外连接

另外，所有的三个自然外连接算符都有其相应的θ连接版本。θ连接的操作是，先进行θ连接，然后将那些不能匹配其他关系的元组，用⊥补齐。可以用$U ⟕_{C} V$表示一个带条件C的θ外连接。同样也可以用L或者R来修饰这个算符，使其表示左外连接或者右外连接。

例5.28 U和V是图5-19中的关系。考虑$U ⟕_{A>V.C} V$。U的元组(4，5，6)、(7，8，9)分别和V
229 的元组(2，3，10)、(2，3，11)满足匹配条件。这样它们都不是悬浮元组。但是，U的元组(1，2，3)和V的元组(6，7，12)是悬浮的。结果在图5-20中列出。 □

A	U.B	U.C	V.B	V.C	D
4	5	6	2	3	10
4	5	6	2	3	11
7	8	9	2	3	10
7	8	9	2	3	11
1	2	3	⊥	⊥	⊥
⊥	⊥	⊥	6	7	12

图5-20 θ外连接的例子

5.4.8 习题

习题5.4.1 已知有关系

$R(A, B)$: {(0,1), (2,3), (0,1), (2,4), (3,4)}

$S(B,C)$: {(0,1), (2,4), (2,5), (3,4), (0,2), (3,4)}

计算下面的表达式:

* a) $\pi_{A+B,\ A^2,\ B^2}(R)$;

b) $\pi_{B+1,\ C-1}(S)$;

* c) $\tau_{B,\ A}(R)$;

d) $\tau_{B,\ C}(S)$;

* e) $\delta(R)$;

f) $\delta(S)$;

* g) $\gamma_{A,\ \mathrm{SUM}(B)}(R)$;

h) $\gamma_{B,\ \mathrm{AVG}(C)}(S)$;

! i) $\gamma_{A}(R)$;

! j) $\gamma_{A,\ \mathrm{MAX}(C)}(R \bowtie S)$;

* k) $R \overset{\circ}{\bowtie}_{L} S$;

l) $R \overset{\circ}{\bowtie}_{R} (S)$;

m) $R \overset{\circ}{\bowtie} S$;

n) $R_{R.B \overset{\circ}{<} S.B} S$。

! **习题5.4.2** 一元操作f如果满足下面的条件则被称为是幂等(idempotent)的:对于任何关系R,$f(f(R)) = f(R)$。也就是,应用f若干次跟应用f一次的结果是一样的。判断下面哪个操作是幂等的(给出理由):

* a) δ; * b) π_L; c) σ_C; d) γ_L; e) τ。

*! **习题5.4.3** 一个能用扩展投影操作来实现、但是不能用一般的投影操作来实现的例子是复制列。例如,如果$R(A, B)$是一个关系,那么$\pi_{A,\ A}(R)$对于R中的元组(a, b)产生元组(a, a)。试问这个操作能否用5.2节中定义的传统的关系代数操作来实现,说明你的理由。 230

5.5 关系的约束

关系代数提供了一种表示通常的约束的方法,例如2.3节中介绍的引用完整性约束。事实上,应该看到,关系代数提供了方便的表示各种约束的方法。甚至函数相关也可以用关系代数表示出来,就像在例5.31中的那样。在数据库编程中,约束非常重要,在第7章中,会看到SQL数据库系统能够像关系代数那样强制约束。

5.5.1 作为约束语言的关系代数

用关系代数表示约束有两种方法。

1.如果R是关系代数表达式,那么$R = \phi$表示"R的值必须为空"的约束,与"R中没有元组"等价。

2.如果R和S是关系代数表达式,那么$R \subseteq S$表示"任何在R中出现的元组都必须在S中出现"的约束,当然,S也可以包括其他的元组。表示约束的方法虽然很多,但是在作用上来说都是等价的,只是某些方法更清楚简洁。比如,约束$R \subseteq S$也可以写成$R - S = \phi$。因为如果每一个在R中出现的元组在S中也出现的话,那么$R-S$就是空。反过来,如果$R-S$没有任何元组的话,那么任何在R中的元组也必然在S中出现。

另一方面，第一种形式的约束$R=\phi$也可以写成$R\subseteq\phi$。从技术上讲，ϕ不是关系代数当中的表达式，但是既然有跟ϕ相等的表达式，例如$R-R$，那么把ϕ当作一个关系代数表达式也没有什么不好。注意，这种等价关系在当R和S都是包的时候也成立。

接下来的几节中，将介绍怎么用这两种不同的形式表示重要的约束。在第7章中将看到，第一种风格的约束是SQL编程当中最常用的约束。但是同样也可以用集合包含风格来考虑问题，
231 然后再转换成与空集等价的风格。

5.5.2 引用完整性约束

2.3节当中的“引用完整性约束”是一类普通约束。它规定在某个上下文中出现的值也必须在另外一个相关的上下文中出现。也就是说，如果实体A与实体B相关，那么B一定要真实存在。例如，在ODL术语中，如果对象A的一个联系是由一个指针来进行物理表示，那么该联系的引用完整性约束设定这个指针不为空，并且一定指向一个真实的对象。

在关系模型中，引用完整性约束看起来有些不同。如果在关系R的某个元组当中，有一个值是v，则由于设计上的原因，可以期望v是另一个关系S的某个元组的成分。可以用一个例子来解释如何用关系代数表达关系模型中的“引用完整性约束”。

例5.29 考虑前面的电影数据库模式，特别是如下两个关系：

```
Movie(title, year, length, inColor, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

有理由假设每个影片的制作者都必须在关系MovieExex里出现。如果不是这样的话，就会出错。于是，至少可以要求系统实现一个关系数据库，以便在对某个影片的制作者一无所知的时候能够告知用户。

更精确地说，每个Movie的producerC#属性必须在MovieExec关系的cert#中出现。因为制作人是以证书号码来惟一确定的。这个约束可以表达为：

$$\pi_{producerC\#}(\mathrm{Movie})\subseteq\pi_{cert\#}(\mathrm{MovieExec})$$

表达式左边的值是所有Movie元组的证书号码属性producerC#值的集合。类似地，表达式右边的值是所有关系MovieExec元组中cert#属性的证书号值集合。约束要求所有在第一个集合中的证书号必须也在第二个集合当中。

同样地，可以用另外一种风格来表达同样的约束：

$$\pi_{producerC\#}(\mathrm{Movie})-\pi_{cert\#}(\mathrm{MovieExec})=\phi$$

□

例5.30 可以用一种简单的方式表达多个属性表示同一个值的引用完整性约束。例如，人
232 们可能希望保证每一个在关系

```
StarsIn(movieTitle, movieYear, starName)
```

中出现的电影也出现在关系

```
Movie(title, year, length, inColor, studioName, producerC#)
```

中。在上述两个关系中，电影都是用电影名-年代对来表示，因为一个属性并不足以标识一部电影。约束

$$\pi_{movieTitle,movieYear}(\mathrm{StarsIn})\subseteq\pi_{title,year}(\mathrm{Movie})$$

是通过把两个关系都投影到合适的属性列表上，然后比较这些电影名-年代对，来表达引用完

整性约束。

□

5.5.3 其他的约束举例

除引用完整性约束外，一个约束符号还可以表达许多其他的约束。例如，可以将任意的函数依赖表示为代数约束，只不过表达起来非常繁琐罢了。

例5.31 在关系

```
MovieStar(name, address, gender, birthdate)
```

使用代数约束的方法表达如下FD：

```
name → address
```

其思路是：如果构造所有可能的MovieStar关系的元组对(t_1，t_2)，一定不存在在属性name上相同，但在address上不同的元组对。为了构造元组对，必须进行笛卡儿乘积。并且选择那些违反FD的对，最后用空集等价约束判断该约束是否成立。

首先，由于是把一个关系跟自己做笛卡儿积，所以必须至少先对该关系的一个副本重命名，以便能够方便地表示乘积的属性名。为了简洁，使用两个新名：MS1，MS2表示对MovieStar关系的引用。这样，FD就可以用下面的代数约束表达：

$$\sigma_{MS1.name=MS2.name \text{ AND } MS1.address \neq MS2.address}(\text{MS1} \times \text{MS2}) = \phi$$

在上面的表达式中，乘积MS1 × MS2中的MS1是下面重命名的缩写

$$\rho_{MS1(name,address,gender,birthdate)}(\text{MovieStar})$$

233

MS2是类似的MovieStar的重命名。 □

一些域约束也可以用关系代数表示。通常，域约束简单要求某一个属性的值有特定的类型，比如整型或者长度是30的字符串类型。这样就可以把属性和域联结起来。但是，通常一个域约束涉及对属性的特定值的需求。如果能接受的值的集合可以由选择条件语言表示，那么，该域约束就可以用代数约束语言表示。

例5.32 假设关系MovieStar中gender属性的合法值只有‘F’和‘M’。于是，这个约束就可以表示为：

$$\sigma_{gender \neq \text{'F' AND } gender \neq \text{'M'}}(\text{MovieStar}) = \phi$$

意思是MovieStar的元组中，gender成分既不等于‘F’也不等于‘M’的结果是空集。

□

最后，有一些约束既不属于2.3节中给出的任何一类，也不是函数依赖或多值依赖。代数约束语言能够表达很多种类的约束。下面给出一个例子。

例5.33 假设要求一个电影公司的经理至少要拥有$10 000 000的资产。这样的约束既不是域约束、单值约束，也不属于引用完整性约束，但是它仍然可以用如下代数约束表达。在描述代数约束之前，首先要对下面两个关系做θ连接

```
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

利用Studio中的presC#和MovieExc中的cert#相等为条件。这个连接将把某个制片人元组与以该制片人为经理的电影公司的相应元组合并为一个元组。如果根据约束要求在合并

起来后的元组上选择那些净资产值小于1000万的元组，其结果应该是空集。因此，该约束可以表达为：

$$\sigma_{netWorth<10000000}(\texttt{Studio} \underset{presC\#=cert\#}{\bowtie} \texttt{MovieExec})=\phi$$

另外一种表达该约束的方法是比较电影公司经理证书号集合与净资产值大于等于1000万的制片人的证书号集，前者应该是后者的子集。约束表达如下：

$$\pi_{presC\#}(\texttt{Studio}) \subseteq \pi_{cert\#}(\sigma_{netWorth\geq 10000000}(\texttt{MovieExec}))$$

234 □

5.5.4 习题

习题5.5.1 对习题5.2.1中的关系表达约束要求，习题5.2.1的关系如下：

```
Product(maker, model, type)
PC(model, speed, ram, hd, rd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

可以用包含的方式或者是空集等价的方式表达你的约束，指出习题5.2.1的数据中哪些违背约束。

* a) 处理速度低于1000的PC机，出售价格不能超过$1500。

b) 一个屏幕超过14英寸的手提电脑至少应有10G硬盘或者出售价格不超过$2000。

! c) 生产PC机的厂家不能同时生产手提电脑。

*!! d) 生产PC的厂家至少生产了一种手提电脑，其最高主频不低于其生产的PC机。

! e) 如果一种手提电脑的内存容量超过PC，那么其价格一定也高于PC。

习题5.5.2 用关系代数表达下列约束，这些约束是基于习题5.2.4的关系。

```
Classes(class, type, country, numGuns, bore)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

可以用两种风格来表达你的约束，即集合包含或者空集等价方式。指出习题5.2.4中哪些数据不符合以下约束：

a) 不存在火炮口径超过16英寸的舰船类别。

b) 如果一个类别中的舰船拥有的火炮超过9门，那么火炮的口径一定不超过14英寸。

! c) 没有所拥有的舰船超过两只的类。

235 ! d) 没有既拥有主力舰（battleship）又拥有巡洋舰（battlecruiser）的国家。

!! e) 在一艘火炮少于9门的舰船被击沉的战役中，不会有火炮超过9门的舰船参战。

! **习题5.5.3** 假设R和S是两个关系。C是引用完整性约束：只要R在属性A_1，A_2，...，A_n上具有特定的值v_1，v_2，...，v_n，那么在关系S中一定有元组在相应的属性B_1，B_2，...，B_n上有值v_1，v_2，...，v_n。用关系代数来表达这个约束。

! **习题5.5.4** R是关系，假定A_1，A_2，...，$A_n \rightarrow B$是涉及R的属性的FD。用关系代数说明这个FD在R中一定成立的约束。

!! **习题5.5.5** R是关系。假设A_1，A_2，...，$A_n \rightarrow\!\!\!\!\rightarrow B_1$，$B_2$，...，$B_m$是涉及$R$的属性的MVD。用关系代数表达该MVD在$R$中成立的约束。

5.6 小结

- *经典关系代数*（Classical Relational Algebra）：这种代数给出了大部分关系模型的查询语言。它的基本操作有并、交、差、选择、投影、笛卡儿积、自然连接、θ连接、重命名等。
- *选择和投影*：选择操作得到的结果关系是所有满足选择条件的元组。投影操作把不感兴趣的列从关系中去掉，形成最终结果。
- *连接*：通过比较两个关系的每一对元组来进行连接操作。在自然连接当中，把那些在两个关系的共同属性上相等的元组接合起来。在θ连接中，则是连接来自两个关系的一对满足θ连接指定的选择条件的元组。
- *基于包的关系*：在商用数据库系统当中，关系实际上是包，也就是在关系中，同一个元组允许重复出现。那些基于集合理论的关系代数操作可以扩展到包，但是其中某些代数规则将不再适用。
- *关系代数的扩展*：为了与SQL和其他一些查询语言的能力相适应，需要一些传统关系代数中不具备的算符。关系排序是一个例子，扩展投影也是，它支持在关系列上进行的操作。分组、聚合、外连接等操作也都需要。 236
- *分组和聚集*：聚集操作对关系的一列加以汇总。典型的聚集操作是求和、求平均、求最小值和最大值。分组操作算符允许对某个关系的元组按照他们在一个或者多个属性上的值分组，以便进一步对每个组进行聚集计算。
- *外连接*：两个关系的外连接是先执行这两个关系的连接，然后将那些悬浮的元组(不能跟任何元组匹配的元组)用null值补齐后，也加入到结果当中。
- 关系代数约束：许多常见的约束可以用某个关系代数表达式被另外一个所包含的形式来表达，或者用某个关系代数表达式等于空集的等价形式表达。这些约束还包括函数依赖和引用完整性约束。

5.7 参考文献

“关系代数”是论述关系模型的基础论文[1]所作出的另外一个贡献。扩展投影，使之包括分组和聚集在论文[2]中有论述。关于表示约束的查询的最初文献是[3]。

1. Codd, E. F., “A relational model for large shared data banks,” *Comm. ACM* **13**:6, pp. 377–387, 1970.

2. A. Gupta, V. Harinarayan, and D. Quass, “Aggregate-query processing in data warehousing environments,” *Proc. Intl. Conf. on Very Large Databases* (1995), pp. 358–369.

3. Nicolas, J.-M., “Logic for improving integrity checking in relational databases,” *Acta Informatica* **18**:3, pp. 227–253, 1982. 237

第6章 数据库语言SQL

关系数据库中最普遍地用于对数据库进行查询和修改操作的语言叫做SQL（有时读作“sequel”）。SQL的含义是结构化查询语言（Structured Query Language）。SQL中支持查询的部分在功能上十分接近在5.4节中介绍的扩展关系代数。SQL还包括了修改数据库的语句（如在关系中插入和删除元组）和定义数据库模式的语句。因此，SQL既是数据操作语言，也是数据定义语言。除此之外，SQL还包括许多其他标准的数据库命令，这些将在第7章和第8章介绍。

和自然语言的方言一样，存在许多不同类型的SQL。首先，存在着三个主要的标准，即ANSI（美国国家标准机构）SQL；对ANSI SQL修改后在1992年采纳的标准，称为SQL-92或SQL2；最近的SQL-99（以前也称为SQL3）标准。SQL-99从SQL2扩充而来并增加了对象关系特征和许多其他的新功能。其次，各大数据库厂商提供不同版本的SQL。这些版本的SQL不但都包括原始的ANSI标准，而且还在很大程度上支持新推出的SQL-92标准。另外，它们均在SQL2的基础上做了修改和扩展，包含了部分SQL-99标准。

本章和接下来的两章着重讲述如何使用SQL作为查询语言。本章的论述基于一个通用（或针对性的）SQL查询界面，即用户通过一个计算机终端向数据库提交查询和修改请求，查询结果在使用的终端上显示。在这里把SQL作为一个独立的查询语言。

下一章讨论约束和触发器——用户对数据库的内容加以控制的另外一种方式。第8章的内容包括在传统的程序设计语言中如何进行有关数据库程序设计。本章和下两章讨论使用的SQL遵从SQL-99标准，几乎所有的商用数据库系统以及较早的标准都包括书中介绍的SQL主要功能。 239

这几章的内容是让读者对SQL有一个初步的了解，难度接近入门教程，因此所讲解的主要内容集中在SQL最常用的部分上。要想了解一些语言细节和不同版本的语言差异可以阅读本章的参考文献。

6.1 SQL中的简单查询

SQL中最简单的查询是找出关系中满足特定条件的元组，这种查询和关系代数中的选择操作类似。和所有的SQL查询差不多，简单查询使用三个保留字 SELECT、FROM和WHERE来表示一个SQL查询语句。

```
Movie(title, year, length, inColor, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

图6-1 和前面相同的数据库模式例子

例6.1 下面例子中使用5.1节描述的数据库模式。在图6-1中这些模式再次出现。6.6节中将介绍怎样使用SQL表达模式。这里先假设在SQL中使用5.1节的关系和域（数据类型）。

第一个查询从关系

```
Movie(title, year, length, inColor, studioName, producerC#)
```

中找出由Disney制片公司在1990年制作的电影，其SQL语句为

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

该查询语句显示了大部分SQL查询语句的结构特征，即select-from-where 形式，它的特点是：

240 • FROM子句给出查询所引用的关系，在本例中查询引用的关系是Movie。

阅读和书写查询语句的技巧

检查一个select-from-where查询的最简单的方式是，首先查看FROM子句，找出该查询涉及到了哪些关系。接着查看WHERE子句，了解要找出的有关元组，它对查询很重要。最后再通过SELECT子句得到输出结果。同样地，在写查询语句的时候也要遵照同样的顺序，即先FROM再WHERE最后SELECT。这样做对理解语句非常有用。

• WHERE子句是一个条件子句，就像关系代数中的选择条件。和查询匹配的元组必须满足此条件。在本例中，条件是：元组中studioName属性值为'Disney'；year属性值为1990。满足这两个约束的元组符合条件，否则不符合条件。

• SELECT子句决定满足条件的元组的哪些属性应该在结果中列出。例子中的*表示列出元组所有的属性。查询结果就是处理后的元组所形成的关系。

解释查询的一种方法是逐个考虑FROM子句中涉及的关系元组。首先WHERE子句应用到元组上，更准确地说，所有在WHERE子句出现的属性由元组中对应该属性的值来代替，如果计算表达式为真，该元组在SELECT中指定的字段值作为结果中的一个元组。这样，查询的结果就是Movie元组中那些由Disney在1990年制作的电影。例如*Pretty Woman*。

当SQL查询处理器碰到Movie里的一个元组

title	*year*	*length*	*inColor*	*studioName*	*producerC#*
Pretty Woman	1990	119	true	Disney	999

（这里，999是一个虚构的电影制片人的证书号）将WHERE子句条件中的studioName换成值'Disney'，year换成值1990，因为这是所说元组的那些属性的值。于是WHERE子句变成了

```
WHERE 'Disney' = 'Disney' AND 1990 = 1990
```

241 显然该条件为真，这样*Pretty Woman*通过了WHERE子句的检查并成为查询结果的一部分。□

6.1.1 SQL中的投影

如果需要，可以减少选定的元组的字段，即将SQL查询中产生的关系投影到它的一些属性上去。在SELECT子句中 * 所在的位置上，如果列出的是FROM子句中涉及的关系的部分属性，那么结果将被投影到所列出的属性之上[⊖]。

例6.2 修改例6.1的查询，仅仅输出电影的标题和长度。其SQL语句是

```
SELECT title, length
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

⊖ SQL中的SELECT保留字实际上和关系代数中的投影操作符对应，而关系代数中选择操作符则对应于SQL查询中的WHERE子句。

该结果是一个有两列的表。两个列的标题分别为title和length。表中列出的元组是成对的，每个元组包括电影的标题和长度。每部电影由Disney在1990年制作。这个关系模式和它的一个元组表现为：

title	*length*
Pretty Woman	119
...	...

□

有时人们希望结果关系的列标题和FROM子句中给出的关系的属性有不同的名字。这可以通过在属性后面跟一个AS保留字和一个别名完成。该别名成为结果关系的列标题。保留字AS是可选的，即别名可以直接跟在它所代表的表达式之后而不必在两者间插入任何标记。

例6.3 修改例子6.2，产生一个关系，用属性name和duration替换title和length。

```
SELECT title AS name, length AS duration
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

结果与例6.2中的元组集相同，但是列标题变成了name和duration。结果关系可能为：

name	*duration*
Pretty Woman	119
...	...

242

□

SELECT子句的另一种选项是用表达式取代属性。换句话说，SELECT列表的功能与5.4.5节中的扩展投影中的列表一样。在6.4节还可以看到SELECT列表中包含5.4.4节中讲到的γ聚集操作。

例6.4 假定需要例6.3的输出，但是要把长度由分钟表示改为小时表示，可以把该例的SELECT子句改为

```
SELECT title AS name, length*0.016667 AS lengthInHours
```

该查询结果是与前例查询同样的电影，但是它的长度是以小时计算，第二列对应的标题改名为lengthInHours，如下所示：

name	*lengthInHours*
Pretty Woman	1
98334. ...	...

□

例6.5 人们甚至可以将一个常量作为表达式在SELECT子句中列出，这样做似乎没有什么意义，但有些应用需要输出一些有意义的词语到SQL的显示结果中去。下面的查询

```
SELECT title, length*0.016667 AS length, 'hrs.'AS inHours
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

产生的元组如下：

title	*length*	*inHours*
Pretty Woman	1.98334	hrs.
...	...	...

□

这里将第三列命名为inHours，使得它和第二列的标题在意义上相符。输出结果中的每一个元组在第三列包含常量hrs.这样看起来好像它是表示第二列值的计量单位。 □

6.1.2 SQL中的选择

关系代数中的选择操作符在SQL中是通过SQL的WHERE子句表示。WHERE子句中的表达式
243 （包括条件表达式）和普通的计算机语言（如C和Java）中的表达式类似。

不区分大小写

SQL是大小写无关的，它把大写和小写字母作看做相同。举个例子，对于保留字FROM，无论写成From或者from甚至FrOm都是正确的。关系名、属性名和别名都同样是大小写无关的。只有引号里面的字符才是区分大小写的。所以'FROM'和'from'是不同的字符串。当然，它们都不是保留字FROM。

可以通过值比较运算来建立表达式，比较运算使用六个常用的比较运算符：=、<>、<、>、<=和>=。除了<>在SQL中表示“不等于”，相应于C语言中的!=运算符之外，其他均和C语言中相同。

常量以及跟在FROM后面的关系的属性都是可以比较的。在进行比较之前也可以使用普通的算术运算符如+、*等作用于在数值上。例如(year-1930)*(year-1930)<100对于那些和1930之间的差别小于等于9的年份的值为真。也可以通过||连接运算符对字符串作连接运算。例如'foo'||'bar'的运算结果为'foobar'。

例子6.1中提到的一个比较运算

```
studioName = 'Disney'
```

将判断关系Movie的studioName属性是否与常量'Disney'相等。这是一个字符串常量，在SQL中，字符串常量是由单引号括起的字符串表示。数值常量，如整数和实数，都可以作为常量。SQL使用普通的记数法表示实数，如 -12.34或1.23E45。

比较运算的结果是一个布尔值：要么TRUE要么FALSE[⊖]。布尔值可以通过逻辑运算符AND、OR和NOT来进行组合，AND、OR、NOT分别为并、或和非运算。例如，在例子6.1中已经看到使用AND运算符来组合两个比较表达式。该例的WHERE子句为真，当且仅当两个比较都为真，即当制片厂名称为'Disney'并且制作年份是1990时整个WHERE子句为真。下面再看一些包含复杂WHERE子句的查询语句。

例6.6 下面的语句查找所有在1970年以后制作的黑白影片。

244
```
SELECT title
FROM Movie
WHERE year > 1970 AND NOT inColor;
```

SQL查询和关系代数

到目前为止所看到的简单的SQL查询都具有以下形式

SELECT *L*
FROM *R*
WHERE *C*

⊖ 关于布尔值的一些特殊情况，参阅6.1.6小节。

其中L是一个表达式列表，R是一个关系，C是一个条件。这种表达式和如下形式的关系代数表达式的意义相同：

$$\pi_L(\sigma_C(R))$$

即，首先对FROM子句关系中的每个元组，使用WHERE子句中指定的条件进行筛选，然后投影到SELECT子句中的属性或表达式列表上。

这个条件中，再一次对两个布尔表达式进行AND运算。第一个是一个普通的比较，而第二个是对属性inColor作NOT运算。在这里直接使用inColor属性是可以的，因为inColor是布尔类型。

下面，考虑查询

```
SELECT title
FROM Movie
WHERE (year > 1970 OR length < 90) AND studioName = 'MGM';
```

这个查询要找出由 MGM制片厂制作的影片名称，它们要么在1970以后制作要么影片长度小于90分钟。注意，比较表达式可以用括号括起来。这里使用括号是因为逻辑运算符优先级的原因。在SQL中逻辑运算符的优先级和其他高级语言相同：AND的优先级高于OR，NOT具有最高优先级。 □

6.1.3 字符串比较

当两个字符串里的字符序列完全相同时称两个字符串相等。SQL允许不同类型的字符串，例如固定长度字符数组和可变长度的字符链表[⊖]。因此需要考虑不同字符串类型之间的转换。 245
例如，字符串 foo可以作为一个固定长度为10同时包括7个填充字符的字符串存储，也可以使用一个变长字符串存储。比较时可能期望两种类型的字符串是相等的，并且等于字符串常量 'foo'。更多关于字符串的物理存储将在12.1.3小节中介绍。

位串的表示

二进制位串是一个以B开头，后面跟着由单引号括起来的0和1串，例如B'011' 表示包含3个位的串，第一位为0，其他两位为1。也可以用十六进制表示，即用X打头后面跟着由单引号括起来的十六进制数字（0到9及a到f，a到f表示10到15）。例如，X'7ff' 表示一个十二位长的位串，由一个0和后面的十一个1构成。每一个十六进制数字表示4比特位，开头的0不能省去。

当使用如 < 或 >= 等比较运算符对字符串作比较运算时，实际上比较的是它们的词典顺序（如字典顺序或字母表顺序）。如果$a_1a_2...a_n$和$b_1b_2...b_m$是两个字符串，如果$a_1<b_1$或$a_1=b_1$且$a_2<b_2$或 $a_1=b_1$，$a_2=b_2$且$a_3<b_3$，如此下去时，则前者小于后者。当$n<m$并且$a_1a_2...a_n = b_1b_2...b_n$时，字符串$a_1a_2...a_n<b_1b_2...b_m$，也就是第一个字符串正好是第二个字符串的一个前缀。例如'fodder' <'foo'，因为每个字符串的头两个字符相同都是fo，而且fodder中的第三个字符在字母表中顺序是在foo的第三个字符之前。同样的，'bar' <'bargin' 因为前者正好是后者的一个前缀。对于字符串的等值比较，有时需要对不同类型的字符串进行类型转换。

⊖ 至少可以把两种字符串分别看作以数组和链表的方式存储。字符串的实际存储方式依赖于具体实现，SQL标准不作具体描述。

SQL也提供了一种简单的模式匹配功能用于字符串比较，字符串比较的另一种方式是

```
s LIKE p
```

其中s是一个字符串，p是模式，即一个可能使用了两个特殊字符% 和 _的字符串 。p中普通字符仅能匹配s中与其相同的字符，而%能匹配s中任何任意长度（包括零长度）的字符串，p中的_则能匹配s中任何一个字符。该表达式为真当且仅当字符串s匹配模式p。同样的，s NOT LIKE p为真时当且仅当s不匹配模式p。

例6.7 如果记得某个电影的名字是“*Star something*”而且知道something是一个四个字母的单词。则可以通过如下查询找到电影的名字

246

```
SELECT title
FROM Movie
WHERE title LIKE 'Star ____';
```

这个查询询问电影名字由九个字符组成的，开始的五个字符是Star加一个空格。后面四个字符可以是任意四个字符，因为每一个_可以匹配任何字符。查询结果可能是满足完全匹配要求的电影名字的集合，如*Star Wars*或*Star Trek*。 □

例6.8 找出所有的电影名中含有所有格’s的电影，查询语句为

```
SELECT title
FROM Movie
WHERE title LIKE '%''s%';
```

要理解这个模式，需要先理解SQL中的单引号。括在字符串两边的单引号是不能代表字符串中的单引号的。SQL约定，字符串中两个连续的单引号表示一个单引号，不作为字符串的结束符。所以模式中的’’s表示一个单引号后面跟一个s。

在’s两边的两个%符号可以匹配任何形式的字符串。所以只要包括子串’s的字符串都能匹配该模式。该查询结果包括如*Logan's Run*或者*Alice's Restaurant*的电影。 □

6.1.4 日期和时间

SQL的实现版本通常将时间和日期作为特殊的数据类型。其值常常表示成不同形式如5/14/1948 或 14 May 1948。这里仅仅描述SQL标准中时间和日期的特定表示法。

日期常量是由保留字DATE后跟一个单引号括起的特定形式字符串组成。例如DATE’1948-05-14’是符合规范的表示。前四个数字字符表示年份，紧接着后面跟一个连字符号，接下来的两个数字字符表示月份。注意，在给出的例子中单数字的月份前补上了一个0。最后是一个连字符和两个数字字符表示日。和月份的表示一样，必须用两位数字表示日。

同样，时间常量由保留字TIME后跟一个单引号字符串组成。该字符串用两个字符表示小时，采用24小时制。接着是一个冒号后跟两个字符表示分钟。接着又是一个冒号后跟两个字符表示秒。如果还要将秒细分，则继续在后面跟一个小数点和任意多的数字字符。例如：TIME

247 ’15:00:02.5’表示下午3点过两秒半。

LIKE表达式中的转义字符

如果要在LIKE表达式的模式中直接使用%和_字符该怎么办呢？SQL没有采用特殊的字符作为转义字符（如UNIX中的反斜线），对于单个模式SQL允许使用任何一个字符作为转义字符。通过在模式后面跟一个保留字ESCAPE和一个用单引号括起来的字符指定转义字符。跟在转义字符后面的%和_将作为其本身所表示的字符出现在模式字符串中，

而不表示匹配一个或多个字符。例如，

```
s LIKE 'x%%x%' ESCAPE 'x'
```

使得x成为模式x%%x% 中的转义字符。字串x%表示单个字符%。这个模式匹配任何以字符%开头并同时以它结尾的字符串。注意，这里仅仅中间的%能匹配“任何字符串”。

作为一个可选项，可以表示为在格林威治(GMT)时间之前（用一个加号表示）和格林威治时间之后（用一个减号表示）若干小时和分钟的时间。例如，TIME' 12:00:00-8:00' 表示太平洋标准时间的正午，正好比格林威治时间晚八个小时。

如果要将日期和时间组合起来就要用到TIMESTAMP类型。通过关键字TIMESTAMP，一个日期值后跟一个空格和一个时间值来组合表示。例如，TIMESTAMP' 1948-05-14 12:00:00' 表示1948年5月14号正午。

可以使用字符串和数值运算中的比较运算符对日期和时间进行比较运算。即 < 对于日期比较意味着前一个日期比第二个早；<对时间而言也是前者早于后者（对于同一天的时间来说）。

6.1.5 空值和涉及空值的比较

SQL允许属性有一个特殊值NULL称做空值。对于空值有许多不同的解释。下面是一些最常见的解释：

1. *未知值*（value unknown）：即知道它有一个值但不知道是什么，例如一个未知的生日。

2. *不适用的值*（value inapplicable）：“任何值在这里都没有意义”，例如，对于MovieStar关系，如果有一个spouse属性表示其配偶。对于一个未婚的影星这个属性可能为NULL值，不是因为不知道其配偶的名字，而是因为没有配偶。 248

3. *保留的值*（value withheld）：“属于某对象的但我们无权知道的值”。例如，未公布的电话号码在phone属性中显示为NULL值。

在5.4.7节中已看到使用外连接操作符如何导致某些元组中产生了NULL值。SQL允许外连接并且在外连接中产生空值。参见6.3.8节。SQL还有一些其他的方式产生空值，例如，如在6.5.1节中将会看到，元组的某些插入产生空值。

在WHERE子句中，要考虑到元组中的空值可能带来的影响。当对空值进行运算时，有两个重要的规则要记住：

1. 对NULL和任何值（包括另一个NULL值）进行算术运算如 × 和 +，其结果仍然是空值。

2. 当使用比较运算符，如 = 或 >，比较NULL值和任意值（包括另一个NULL值），结果都为UNKNOWN值。值UNKNOWN是另外一个和TRUE和FALSE相同的布尔值。我们将简介地介绍UNKNOWN值的操作。

可是，我们要记住虽然NULL也是一个可以出现在元组中的值，但是它不是一个常量。因此可以利用上面的规则对值为NULL的表达式进行运算，但是不可以直接将NULL作为一个操作数。

例6.9 如果x的值是NULL，那么x+3的值也是NULL，但是NULL+3不是合法的SQL表达式。同样地，表达式x = 3的值是UNKNOWN，因为x值为NULL，不能确定x是否等于3。而比较表达式NULL=3是非法的SQL表达式。 □

顺便提一下，正确判断x的值是否为NULL的方式是用表达式 x IS NULL 表示。如果x的值为NULL那么该表达式为TRUE否则为FALSE。例如，x IS NOT NULL 值为TRUE除非x的值是NULL。

6.1.6 布尔值UNKNOWN

在6.1.2节的比较运算结果要么是TRUE要么是FALSE值。这两种布尔值可以通过逻辑运算符AND、OR和NOT组合在一起。由于上面刚刚讲到的NULL值出现，比较结果可能产生第三个布尔值：UNKNOWN。现在必须知道逻辑运算符对于三种布尔值进行运算的规则。

可以用一种简单的方式来记住理解。把TRUE看做1（完全真），FALSE看做0（完全假），
249 UNKNOWN看做1/2（处于真假之间）。这样：

使用空值的小缺陷

NULL值在SQL中通常表示“一个未知的但是的确存在的值”。然而这样表示有时和人们的直觉不符合。例如，假定x是某个元组的一字段，该字段的域类型是整型。因此可以推断无论整数x的值是什么 $0*x$的结果肯定是0。但是如果x的值是NULL，应用6.1.5节的规则(1)，0和NULL的积却是NULL。同样地$x-x$的值肯定是0，因为一个整数减去它本身结果为0，而应用规则（1）再次导致结果为NULL。

1. 两个布尔值之间的AND运算结果取两者之间最小的值。如果x或y两者之一为FALSE，则x AND y 为 FALSE；如果两者都不为FALSE但是至少有一个是UNKNOWN，则为UNKNOWN；当二者皆为TRUE时结果为TRUE。

2. 两个布尔值的OR运算取两者之间较大值。当两者之一为真，则x OR y为TRUE；当两者都不为TRUE但是至少有一个为UNKNOWN，则结果为UNKNOWN；当两者都为FALSE，则结果为FALSE。

3. 布尔值v的非值为$1-v$。当x为FALSE，则NOT x的值为TRUE，当x的值为TRUE，则NOT x为FALSE，当x的值为UNKNOWN时，其结果仍然为UNKNOWN。

图6-2是对操作数x和y赋予布尔值，形成九种不同组合时三种逻辑运算的结果。最后一个操作符NOT的计算结果仅仅作用于x。

x	y	x AND y	x OR y	NOT x
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	UNKNOWN	UNKNOWN	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
UNKNOWN	TRUE	UNKNOWN	TRUE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	FALSE	FALSE	UNKNOWN	UNKNOWN
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	UNKNOWN	FALSE	UNKNOWN	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

250 图6-2 三值逻辑真值表

SQL的条件，即出现在select-from-where语句中的WHERE子句，被应用到关系的每一个元组。对于每一个元组，条件可以有三种值，TURE、FALSE或UNKNOWN。但是只有条件为TRUE时，元组才符合要求。而那些值为FALSE和UNKNOWN的元组则不在查询结果之中。这种情形导致了一个类似“使用空值的小缺陷”框中提到的问题。下面的例子说明这一点

例6.10 假定对关系Movie (title, year, length, inColor, studioName, producerC#) 进行如下查询：

```
SELECT *
FROM Movie
WHERE length <= 120 OR length > 120;
```

直观地看，查询期望返回关系Movie的所有元组，因为电影的长度要么小于或等于120要么大于120。

但如果该关系某个元组的length字段具有NULL值，那么表达式length<=120和表达式length>120的结果都为UNKNOWN。两个UNKNOWN的OR运算仍然是UNKNOWN。这样WHERE子句也为UNKNOWN。从而导致该元组不出现在结果集合里面。这样，该查询的真正意义变成了“从关系Movie找出所有length字段为非空的元组”。 □

6.1.7 输出排序

有时需要对查询结果以某种顺序表示。可以基于任何一个属性来排序，并且将其他的属性跟在它之后进行约束。当第一属性值相同时，将第二个属性作为排序的依据，如此类推。类似5.4.6节中的τ操作。在select-from-where语句后加上如下子句用于排序。

```
ORDER BY <list of attributes>
```

排序的默认序为按升序排列，但也可以通过给某个属性加上保留字DESC（descending）按照其降序排列。同样的可以指定保留字ASC按升序排列，ASC可以省略。

例6.11 下面的例子是对例6.1的查询进行重写，找出Disney在1990年制作的电影，该关系为

```
Movie(title, year, length, inColor, studioName, producerC#)
```

251

其结果按照电影的长度排列，短的排在前面。如果两部电影长度相同，则按电影名的字典顺序排列。查询按如下方式写

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990
ORDER BY length, title;
```

□

6.1.8 习题

* **习题6.1.1** 如果一个查询的SELECT子句为

```
SELECT A B
```

A和B是不同的属性还是B是A的别名？

习题6.1.2 根据给出的电影数据库样例用SQL语句写出后面的查询。

```
Movie(title, year, length, inColor, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

* a) 找出电影制片厂MGM的地址。

b) 找出Sandra Bullock's的生日。

* c) 找出那些1980年制作，或者电影名中包括“Love”单词的电影中出现的所有电影明星。

d) 找出所有净产值（netWorth）为$10 000 000的出品人。

e) 找出所有是男性或是住在Malibu(地址中包括Malibu字符串)的电影明星。

习题6.1.3 使用习题5.2.1中提供的数据库模式用SQL语句写出后面的查询，并使用习题

5.2.1提供的资料写出查询结果。

```
Product(maker, model, type)
PC(model, speed, ram, hd, rd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

252

* a) 找出所有价格在$1200以下的PC机的型号、速度和硬盘大小。

* b) 要求同(a)，但是重命名列speed为megahertz，以及列hd为gigabytes。

c) 找出所有打印机制造厂商。

d) 找出价格在$2000以上的手提电脑的型号、内存大小和屏幕尺寸。

* e) 找出关系Printer中所有彩色打印机元组，注意属性color是一个布尔类型。

f) 找出价格少于$2000并且拥有12x或16x DVD的PC机的型号、速度和硬盘尺寸。把rd属性看做一个字符串类型。

习题6.1.4 基于习题5.2.4给出的数据库模式和资料写出后面的查询语句以及查询结果。

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

a) 找出至少装备10门火炮的船只所属类别名和制造国家。

b) 找出在1918年以前下水的舰船的名字，并且把结果列名改为ShipName。

c) 找出所有在战役中被击沉的船只和那次战役的名字。

d) 找出具有相同类别名的所有船只。

e) 找出所有以“R.”字符开头的船只的名字。

! f) 找出所有包括三个或三个以上单词的船只名字(例如 King George V)。

习题6.1.5 假定*a*和*b*是可能为NULL的整型属性。对于以下的条件（出现在WHERE子句中），准确地描述满足这些条件的(*a*,*b*)元组集合。包括那些*a*和（或）*b*可能为NULL值的元组。

253

* a) `a = 10 OR b = 20`

b) `a = 10 AND b = 20`

c) `a < 10 OR a>= 10`

*! d) `a = b`

! e) `a <= b`

! **习题6.1.6** 在例子6.10中讨论的查询

```
SELECT *
FROM Movie
WHERE length <= 120 OR length > 120;
```

当某部电影的长度属性为空值时，该查询显得不是很直观。请写出一个WHERE子句，其中只包括一个条件（条件中不使用AND和OR）的和上面等价的更简单的查询。

6.2 多个关系上的查询

关系代数的强大在于它能够通过连接、笛卡儿积、并、交和差来组合多个关系。在SQL中也可以做相同的操作。集合理论中的操作-并、交和差-在SQL中直接出现。这些将在6.2.5节中

讨论。首先学习如何在SQL的select-from-where句型中进行关系的笛卡儿积和连接运算。

6.2.1 SQL中的积和连接

SQL用简单的方式在一个查询中处理多个关系：在FROM子句中列出每个关系，然后在SELECT子句和WHERE子句中引用任何出现在FROM子句中关系的属性。

例6.12 找出电影*Star Wars*的制片人名字。要回答这个问题至少要使用以前例子中出现的两个关系：

```
Movie(title, year, length, inColor, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

制片人的证书号(procucerC#)在Movie关系中给出，因此可以对Movie做一个简单查询取得这个证书号码。然后对关系MovieExec做第二次查询找出具有该证书号的人名。

不过还有更好的方法，即把这两步合并成对关系Movie和MovieExec的一个查询，如下所示： 254

```
SELECT name
FROM Movie, MovieExec
WHERE title = 'Star Wars' AND producerC# = cert#;
```

这个查询检查关系Movie和MovieExec的所有元组对。这一对元组的匹配条件在WHERE子句中给出：

1. 关系Movie中元组的title字段值必须为'Star Wars'。

2. 关系Movie中元组的producerC#属性和MovieExec关系中的cert#属性必须具有相同的证书号，即这两个元组必须指的同一个电影制片人。

当找到符合上面两个条件的一对元组的时候，输出MovieExec中元组的name属性作为答案的一部分。当来自Movie中的元组是*Star Wars*并且来自MovieExec中的元组是George Lucas时，这时电影名是正确的并且证书号也符合，两个条件都满足，于是获得所需要的资料。这时George Lucas作为惟一输出的值。图6-3说明了整个过程。6.2.4节中将更详细地解释多关系查询。 □

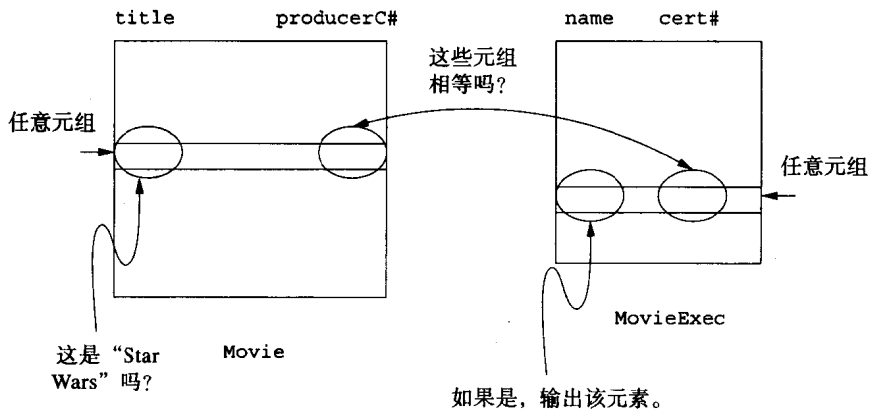


图6-3 例6.12的查询处理过程示意图

6.2.2 避免属性歧义

有时当查询涉及到几个关系的时候，关系中可能会有两个或两个以上的属性具有相同的名字，如果是这样，就需要用明确的方式指定这些相同名字的属性的意义。SQL通过在属性前面 255

加上关系名和一个点来解决这个问题。如R.A表示关系R的属性A。

例6.13 两个关系

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

都有属性name和address。假定希望找出地址相同的影星和制片人。下面的查询符合要求。

```
SELECT MovieStar.name, MovieExec.name
FROM MovieStar, MovieExec
WHERE MovieStar.address = MovieExec.address;
```

在这个查询里，查找一对元组，一个来自MovieStar，另一个来自MovieExec，它们的address字段相同。WHERE子句指明了来自两个关系的address字段必须相同。对于每一对匹配的元组，从MovieStar和MovieExec的元组里分别提取name属性，结果是元组配对的一个集合，如下所示

MovieStar.name	*MovieExec.name*
Jane Fonda	Ted Turner
...	...

□

即使属性没有二义性，在它前面加上关系名和一个点也是允许的。例如，也可以把例6.12的查询写成

```
SELECT MovieExec.name
FROM Movie, MovieExec
WHERE Movie.title = 'Star Wars'
      AND Movie.producerC# = MovieExec.cert#;
```

和前面例子不同的是，这里在属性前面给出了属性所属的关系。

6.2.3 元组变量

只要查询涉及多个关系，可以通过在属性名前面加上属性的关系名和一个点来区分不同关
256 系同名的属性。但有时查询可能使用到同一个关系中的两个或更多的元组。由于查询需要，关系*R*可能在FROM子句里被列出任意次，但是对每一个*R*的出现需要一种方式来区分。SQL允许为FROM子句中出现的每个*R*定义一个别名，称之为元组变量。方法是在FROM子句中的每一个*R*的后跟一个可选的保留字AS和元组变量的名字。下面的讨论中将忽略保留字AS。

元组变量和关系名字

从技术角度上来说，对SELECT子句和WHERE子句中的属性引用都是针对元组变量的。但如果一个关系名仅在FROM子句中出现一次，则可以将该关系名作为它的元组变量。也可以认为FROM子句中的关系是*R* AS *R*的简写。进一步还可以看到，当一个属性明确地属于某个关系时，这个关系名（元组变量）可以省略。

在SELECT和WHERE子句中，可以通过在属性前加上一个正确的元组变量和一个点符号来消除关系*R*的属性歧义。这样，元组变量可以作为关系*R*的另外一个名字用在需要的地方。

例6.14 6.13例子中是查找具有相同地址的影星和制片人。本例是想找出具有相同地址的两个影星。查询本质上是相同的，但本例中要考虑两个来自MovieStar中的元组，而不是分

别来自MovieStar和MovieExec。这里用元组变量作为别名区分对MovieStar的两次使用。查询语句可以写成如下形式。

```
SELECT Star1.name, Star2.name
FROM MovieStar Star1, MovieStar Star2
WHERE Star1.address = Star2.address
      AND Star1.name < Star2.name;
```

可以看到在FROM子句中声明了两个元组变量Star1和Star2，每一个都是关系MovieStar的别名。这两个元组变量在SELECT子句中引用两个元组的name字段。而在WHERE子句中，由Star1和Star2表示的两个MovieStar元组的address字段的值相同。

WHERE子句中的第二个条件Star1.name < Star2.name表示一个影星的名字的字典顺序在第二个影星的名字之前。如果这个条件漏掉的话，那么Star1和Star2可能引用的是同一个元组。两个元组变量引用address字段值相同的元组，这样就会产生一对相同的影星[⊖]。第二个条件也使得对于每一对具有相同地址的影星仅以字典顺序输出一次。如果使用 <>(不等于)作为比较运算符，将会把那些住在一起的已婚影星重复输出。如：

257

Star1.name	*Star2.name*
Alec Baldwin	Kim Basinger
Kim Basinger	Alec Baldwin
...	...

□

6.2.4 多关系查询的解释

有几种不同的方式定义前面讲到的select-from-where所表达的意义。如果每个查询应用到相同的关系实例上并且返回相同的结果，则称它们都等价。下面依次讨论它们：

嵌套循环

到目前为止，在例子中一直隐含使用的语义是元组变量。一个元组变量包括了相应关系的所有元组。如同在方框“元组变量和关系名字”提到的，没有使用别名的关系名也是元组变量，它包括了该关系的所有元组。如果使用了几个元组变量，可以把它们想像成嵌套循环，每一个元组变量为一个循环，访问了所代表的关系里的每一个元组。当每次把元组的值赋给元组变量的时候，要判断WHERE子句是否为真，如果是，则产生一个由跟在SELECT语句后面的表达式值构成的元组。注意，用当前赋给元组变量的元组中的值取代表达式中的每一项。查询-回答算法由图6-4给出。

并行赋值

可以通过另外一个等价的定义来说明，该定义中不直接通过元组变量进行嵌套扫描，而是以一种任意的顺序，或者说并行的顺行从适当的关系中把所有可能的元组都赋给元组变量。对于每一个赋值，考虑WHERE子句是否为真。每一种产生真值WHERE子句的赋值给答案贡献一个元组。该元组由SELECT子句中的属性构造给出，通过赋给的值计算相应字段的值。

258

转换成关系代数

第三种方案是把SQL查询和关系代数关联起来。从FROM子句后面的元组变量开始，求它们表示的关系的笛卡儿积。如果两个元组变量表示同一个关系，那么这个关系在笛卡儿乘法中出现两次，并且对它的属性重新命名以保证所有属性的名字都不相同。同样，对来自不同关系

⊖ 在例6.13中，当某个人既是影星又是制片人的时候也会出现类似问题。可以通过要求两个名字不同来解决。

的同名属性也重命名以防止歧义。

```
LET the tuple variables in the from-clause range over
        relations R1, R2, ..., Rn;
FOR each tuple t1 in relation R1 DO
    FOR each tuple t2 in relation R2 DO
        ...
        FOR each tuple tn in relation Rn DO
            IF the where-clause is satisfied when the values
            from t1, t2, ..., tn are substituted for all
            attribute references THEN
                evaluate the expressions of the select-clause
                according to t1, t2, ..., tn and produce the
                tuple of values that results.
```

图6-4 回答一个简单查询

在获得笛卡儿积之后，通过直接把WHERE子句转换成一个选择条件对它进行选择操作。即WHERE子句中引用的属性由笛卡儿积中对应的属性所取代。最后，利用SELECT子句中提供的表达式列表作最后的（扩展的）投影操作。和WHERE子句一样，直接把SELECT子句中引用的每个属性作为关系积中的相应属性。

例6.15 把例6.14的查询转换成关系代数。首先，FROM子句中存在两个元组变量，每一个都引用关系MovieStar。这样表达式（没有重命名）开始为

```
MovieStar × MovieStar
```

结果关系拥有八个属性，前面四个来自关系MovieStar第一份拷贝的属性name、address、gender和birthdate。后四个来自关系MovieStar第二份拷贝的同名属性。通过在属性前加上元组变量的别名和一个点来重命名（例如Star1.gender）。为了简洁，这里使用新的符号把这些属性称为A_1，A_2，...，A_8这样，A_1对应Star1.name，A_5对应Star2.name如
259 此类推。

SQL语义学导致的意外结果

假定R，S和T是一元关系（仅包含一个字段）。每个关系仅仅包含一个属性A。如果希望找出那些在R中也同时在S或T中（或者在二者中）的元素。即计算$R \cap (S \cup T)$。我们可能会指望下面的SQL查询完成这项工作。

```
SELECT R.A
FROM R, S, T
WHERE R.A = S.A OR R.A = T.A;
```

然而当关系T为空的时候情况有些特别。由于$R.A = T.A$不可能满足，我们可能直观地根据“OR”操作符认为此查询产生$R \cap S$。然而无论使用6.2.4小节使用的哪种定义，不管R和S有多少相同的元素，结果都是空。如果用图6-4嵌套循环语义来解释，会发现元组变量T循环重复0次，这是因为该关系中没有元组可以扫描。这样，for循环的if句子就不会执行。因此不会产生任何结果。同样地，如果采用把元组赋给元组变量的方式，因为没有任何可以赋给元组变量T的元组，所以没有任何赋值。最后，如果使用笛卡儿积的方式，开始计算$R \times S \times T$，结果也会是空，因为T是空的。

在对属性重命名后，从WHERE子句从获得的选择条件是A_2=A_6和A_1<A_5。投影列表是A_1、A_5。

这样关系代数运算

$$\pi_{A_1,A_3}(\sigma_{A2=A6 \text{ AND } A_1<A_5}(\rho_{M(A_1,A_2,A_3,A_4)}(\text{MovieStar}) \times \rho_{N(A_5,A_6,A_7,A_8)}(\text{MovieStar})))$$

描述了整个查询。 □

6.2.5 查询的并、交、差

在关系代数中可以用集合操作的并、交和差来组合关系。在查询结果上，SQL提供了对应的操作，条件是这些查询结果提供的关系具有相同的属性和属性类型列表。保留字UNION，INTERSECT和EXCEPT分别对应∪，∩和—。当UNION这样的保留字用于两个查询时，查询应该分别用括号括起来。 260

例6.16 假定要找出那些既是女影星又同时是具有超过$10 000 000资产的制片人的名字和地址。使用下面两个关系：

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

可以使用图6-5的查询来完成任务。这里第一到第三行产生一个模式为（name,address）的关系，该关系的元组中是女影星的名字和地址。

```
1)  (SELECT name, address
2)   FROM MovieStar
3)   WHERE gender = 'F')
4)       INTERSECT
5)  (SELECT name, address
6)   FROM MovieExec
7)   WHERE netWorth > 10000000);
```

图6-5 女影星和富有制片人的交集

类似地，第五到第六行产生那些身价在$10 000 000的“富有”的制片人的元组集合。这个查询也同样产生一个只具有属性name和address的关系模式。由于两个查询产生的模式是相同的，可以对它们取交，如图第四行所示。 □

例6.17 按照相同的格式，也能够从两个关系中找出两类人员组成的集合的差异。查询语句

```
(SELECT name, address FROM MovieStar)
   EXCEPT
(SELECT name, address FROM MovieExec);
```

给出了不是电影公司制片人的影星的名字和地址，这里没有考虑性别和资产。 □

在上面的两个例子中，进行交或者差运算的关系的属性恰好完全相同。然而，如果有必要得到相同的属性集的话，可以像例子6.3中那样重新命名属性。

例6.18 假定想得到所有出现在Movie或StarsIn关系中的电影的名字和年份。其关系定义是：

```
Movie(title, year, length, inColor, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
```

261

易读的SQL查询

通常，我们在写SQL查询时把重要的保留字如FROM或WHERE作为每一行的开头。这种风格使得读者能直观地理解查询的结构。但如果一个查询或子查询非常短的时候，我们可以直接把它写成一行，就像例子6.17所示的那样。这种风格使得查询语句很紧凑，也具有很好的可读性。

理想情况下，电影的元组集应该相同，但是实际使用的关系中通常不同，例如可能有的电影没有列出影星，或者StarsIn中的元组提到某部电影但是在Movie关系中却找不到[⊖]。其查询语句可以这样写：

```
(SELECT title, year FROM Movie)
    UNION
(SELECT movieTitle AS title, movieYear AS year FROM StarsIn);
```

结果是返回所有出现在Movie或StarsIn中的影星。查询结果的关系属性为title和year。□

6.2.6 习题

习题6.2.1 使用给出的关于电影数据库模式例子。

```
Movie(title, year, length, inColor, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

用SQL写出下面的查询：

* a) 谁是电影*Terms of Endearment*中的男影星？

b) 哪些影星在MGM于1995年制作的电影里演出？

c) 谁是MGM制片厂的经理？

*! d) 哪些电影比 *Gone With the Wind*长？

! e) 哪些制片人的资产比Merv Griffin多？

习题6.2.2 根据习题5.2.1的数据库模式写出下面的查询，并用那个习题给出的数据算出查询结果。

262

```
Product(maker, model, type)
PC(model, speed, ram, hd, rd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

* a) 查询硬盘容量至少30G的手提电脑制造商及电脑速度。

* b) 查询制造商B制造的任意类型的所有产品的型号和价格。

c) 查询只卖手提电脑不卖PC的厂商。

! d) 查询出现在两种或两种以上PC中的硬盘的尺寸。

! e) 查询每对具有相同速度和RAM的PC机，每一对只出现一次。例如，如果（i, j）符合，则（j, i）就不能在结果中出现。

!! f) 查询生产至少两种速度大于等于1000的计算机（PC或手提）的厂商。

⊖ 在7.1.4小节中介绍了防止这种分歧的方法。

习题6.2.3 根据习题5.2.4的数据库模式写出下面的查询，并用那个习题给出的数据算出查询结果。

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

a) 找出重量超过35 000吨的船只。

b) 找出参加过Guadalcanal战役的船只的名字、排水量和火炮数量。

c) 列出所有数据库中提到的船只（注意，并非所有的船只都出现在Ships关系中）。

! d) 找出同时具有战列舰和巡洋舰的国家。

! e) 找出曾在某次战役中受创但后来又在其他战役中出现的船只。

! f) 找出参战船只至少有三艘来自同一个国家的战役。

*! **习题6.2.4** 一个关系代数查询的一般形式为：

$$\pi_L(\sigma_C(R_1 \times R_2 \times \cdots \times R_n))$$

263

这里，L是任意属性列表，C是任意条件。关系列表中R_1，R_2，...，R_n可能重复包括某个关系，这种情况下可以对R_i进行重命名。用SQL给出这种形式的任意查询。

! **习题6.2.5** 另一个常用的关系代数查询格式是

$$\pi_L(\sigma_C(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n))$$

使用与习题6.2.4中同样的假定。不同的是这里使用的是自然连接而不是笛卡儿积。用SQL给出这种形式的任意查询。

6.3 子查询

在SQL中，一个查询可以通过不同的方式使用另一个查询。当某个查询是另一个查询的一部分时，称之为子查询。子查询还可以拥有下一级的子查询。如此递推，可以随需要拥有多级子查询。前面已经看到过使用子查询的例子。在6.2.5节中，通过连接两个子查询形成一个新的查询可以完成关系的并、交和差。还有一些使用子查询的其他方式：

1. 子查询可以返回单个常量，这个常量能在WHERE子句中和另一个常量进行比较。
2. 子查询能返回关系，该关系可以在WHERE子句中以不同的方式使用。
3. 像许多存储的关系一样，子查询形成的关系能出现在FROM子句中。

6.3.1 产生标量值的子查询

一个能成为元组字段值的原子值称为标量。一个select-from-where句子产生的关系可以有任意多的属性和元组。可是，人们经常只对单属性的值感兴趣。另外，有时可以从键或其他信息推导信息。这时那个属性仅有单个值。

如果这样的话，可以把这个select-from-where句子括起来看做一个标量。特定条件下，它可以出现在WHERE子句中任何常量或者表示元组字段的属性的地方。例如，可以用子查询的结果和一个常量或属性比较。

264

例6.19 回忆在例6.12中，为了要找出*Star Wars*这部影片的制片人，不得不对如下两个关系进行查询。

```
Movie(title, year, length, inColor, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

因为只有前者包含电影片名信息，而后者含有制片人的名字。两个信息通过证书号连接在一起。证书号码惟一地标志了制片人。查询写为：

```
SELECT name
FROM Movie, MovieExec
WHERE title = 'Star Wars' AND producerC# = cert#;
```

这里还可以用另一种方式去理解这个查询。使用关系Movie仅仅是为了取得电影*Star Wars*制片人的证书号。一旦获得这个号码，就可以用它在关系MovieExec中查到对应的制片人名字。第一个问题，取得制片人的证书号可以写成一个子查询。该子查询的结果可以作为单个值用在主查询中，从而取得和上面查询同样的结果。图6-6给出了该查询语句。

```
1)  SELECT name
2)  FROM MovieExec
3)  WHERE cert# =
4)      (SELECT producerC#
5)       FROM Movie
6)       WHERE title = 'Star Wars'
        );
```

图6-6 通过嵌套子查询找出影片*Star Wars*的制片人

图6-6的第(4)到第(6)行是子查询。单独地看这个简单查询，它的结果是一个具有惟一属性producerC#的关系，我们期望从这个关系中仅找到一个元组。该元组为(12345)的形式，也就是说，是某个整型值，可能是12345或是George Lucas的证书号。如果子查询返回零个或多个元组，就会出现运行错误。

执行完这个子查询后，可以看做先用12345替换了整个子查询，然后再执行第一到三行。即主查询按照如下的方式执行：

```
SELECT name
FROM MovieExec
WHERE cert# = 12345;
```

265

查询的结果是George Lucas。 □

6.3.2 含有关系的条件表达式

有许多SQL运算符可以作用在关系*R*上并产生布尔值结果。通常像select-from-where这样的子查询的结果是返回一个关系*R*。一些运算符，如IN、ALL和ANY，将首先在包含标量值*s*的简单形式中被解释。这种情况下，关系*R*要求是一个单个列的关系。这里给出这些操作符的定义。

1. EXISTS *R*是一个条件，当且仅当*R*非空时为真。

2. *s* IN *R*为真，当且仅当*s*等于*R*中的某一个值。反之，*s* NOT IN *R*为真，当且仅当*s*不等于*R*中的任何一个值。这里假定*R*是一元关系。在6.3.3小节中将讨论IN和NOT IN运算符的扩展情况，在那里*R*可以有一个以上的属性，*s*是一个元组。

3. *s* > ALL *R*为真，当且仅当*s*大于一元关系*R*中的任何一个值。> 运算符可以替换成其他的五个比较运算符之一。根据运算符来决定该条件表达式的意义。但都是要求*s*和*R*中所有元组比较结果为真时条件才为真。例如*s* <> ALL 和 *s* NOT IN *R*的意义相同。

4. *s* > ANY *R*为真，当且仅当*s*至少大于一元关系*R*中的一个值。> 运算符可以替换成其他五个比较运算符之一。条件的意义也根据运算符的意义类似。但都必须保证*s*和*R*中至少一个元组的比较为真时该条件表达式才为真。

EXISTS、ALL和ANY可以在使用它们的表达式前面加上NOT以否定掉整个表达式，就像它们是布尔表达式一样。这样，NOT EXISTS R为真当且仅当R为空时；NOT s > ALL R为真当且仅当s不是R中的最大值；NOT s > ANY R为真当且仅当s是R中的最小值。在后面会看到几个使用本节所讲到的操作符的例子。

6.3.3 含有元组的条件表达式

元组在SQL中通过括号括起来的标量值列表来表达。例如(123,'foo')和(name,address,networth)。前者用常量作为元组成分，后者用属性作为元组成分。属性和常量混合在一起也是可以的。

如果一个元组t和关系R的元组有相同的组成分量个数，那么使用6.3.2节的运算对t和R进行比较是有意义的。例如t IN R或t<>ANY R。如果第二个比较表达式为真，意味着R中存在着与t不同的元组。注意，当一个元组和关系R的成员比较时，必须按照关系属性的指定排列顺序来比较元组各字段值。 266

```
1)  SELECT name
2)  FROM MovieExec
3)  WHERE cert# IN
4)     (SELECT producerC#
5)      FROM Movie
6)      WHERE (title, year) IN
7)          (SELECT movieTitle, movieYear
8)           FROM StarsIn
9)           WHERE starName = 'Harrison Ford'
            )
        );
```

图6-7 找出Harrison Ford演过的电影的制片人

例6.20 在图6-7的查询中，使用了如下三个关系

```
Movie(title, year, length, inColor, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieExec(name, address, cert#, netWorth)
```

要找出Harrison Ford主演的电影的制片人。它包括一个主查询和一个嵌套在主查询中的子查询，以及嵌套在该子查询中的另一个子查询。

分析嵌套查询应当从里向外分析。先来看嵌在最里层的子查询，位置从第7行到第9行。这个查询检查关系StarsIn并找出所有starName字段为'Harrison Ford'的元组。该子查询获得所有满足检查条件的电影名字和制作年份。记住title和year属性合起来才是关系Movies的键，所以需要产生包括这两个属性的元组去惟一标志一部电影。第7到第9行的结果类似图6-8所示。

现在，考虑中间的查询，即第4到第6行。它从Movie关系中检索元组，这些元组的title和year包含在图6-8所形成的关系中。对于每个找到的元组，返回制片人的证书号，所以中间子查询的结果是所有Harrison Ford出演的电影制片人的证书号集合。

最后，考虑从第1到第3行的主查询。它检查关系MovieExec中的元组，从中找出cert#字段值与中间查询返回的证书号集中的某一个相等的元组。对于每一个这样的元组，返回制片人的名字，也即给出了想要的Harrison Ford出演影片的制片人的名字。□ 267

顺便说一下，图6-7的嵌套查询以及其他的一些嵌套查询都可以写成select-from-where 形式

的单个查询语句。在嵌套查询中的关系，无论是主查询还是子查询中的，都将其写在FROM后面。而嵌套查询的IN联系由WHERE子句中等于联系所代替。如图6-9的查询和图6-7中的查询本质上是相同的。不同之处在于图6-9中的查询可能会返回重复的制片人-例如George Lucas会重复。6.4.1节中将讨论这个问题。

title	*year*
Star Wars	1977
Raiders of the Lost Ark	1981
The Fugitive	1993
...	...

图6-8 内层查询返回的title-year属性对

```
SELECT name
FROM MovieExec, Movie, StarsIn
WHERE cert# = producerC# AND
      title = movieTitle AND
      year = movieYear AND
      starName = 'Harrison Ford';
```

图6-9 不用子查询找出Ford的电影制片人

6.3.4 关联子查询

最简单的子查询只需计算一次，它返回的结果用于高层查询。复杂的嵌套子查询要求一个子查询计算多次，每次赋给查询中的某项来自子查询外部的某个元组变量的值。这种类型的子查询叫做关联子查询。下面通过例子来说明。

例6.21 找出被两部或两部以上电影使用过的电影名。使用一个外查询从下面关系中查找。

```
Movie(title, year, length, inColor, studioName, producerC#)
```

对于该关系中的每个元组，在一个子查询中询问是否有一部电影具有相同的名字和更早的年份。
268 整个查询如图6-10所示。

和分析其他的子查询一样，开始从第4到第6行的最里层子查询开始分析。如果第6行的Old.title被一个常量字符串如'King Kong'所替换的话，那么该子查询很容易理解，即要找出影名为 ***King Kong***的电影的制作年份。这个嵌套查询和以前的稍有不同。惟一的问题在于不能确定Old.title的值。然而，当第1到第3行的外层查询中对Movie中的元组遍历的时候，每一个元组可以为Old.title提供一个值。这样可以用提供的Old.title去执行第4到第6行的子查询，从而决定第3到第6行的WHERE子句是否为真。 □

```
1)  SELECT title
2)  FROM Movie Old
3)  WHERE year < ANY
4)      (SELECT year
5)       FROM Movie
6)       WHERE title = Old.title
        );
```

图6-10 找出多次出现的电影名

当某部电影具有和Old.title相同的电影名，并且制作年份晚于元组变量Old当前所表示的元组的电影制作年份时，第3行的条件为真。只有当Old元组的年份是该同名电影的最近一次制作时间时，该条件为真。接下来的第1到第3行输出比具有同名电影数少一次的电影名称。

也就是说，某部电影制作了两次只输出一次，制作了三次的电影只输出两次，如此类推[⊖]。

写关联子查询时很重要的一点是要注意名字的作用范围（scoping rule），通常子查询中的某个属性是属于该子查询FROM子句中的某个元组变量，只要该元组变量所表示的关系模式中定义了该属性。如果没有定义该属性，就直接查找该子查询的外层查询，如此类推。这样，图6-10第4行的year和第6行的title是第5行中引用关系Movie副本的元组变量属性。也就是第4到第6行子查询所使用的Movie关系的副本。

另外，可以通过在属性前加上某个元组变量名和一个点表明该属性属于该元组变量。这就是为外层查询的Movie关系取Old别名的原因，也是在第6行引用Old.title的原因。注意，如果第2行和第5行FROM子句中的两个关系不同，就有可能不用别名，在子查询中可以直接使用第2行中提到的关系的属性。 269

6.3.5 FROM子句中的子查询

子查询的另一个作用是在FROM子句中当关系使用。在FROM列表中，除了使用一个存储关系以外，还可以使用括起来的子查询。由于这个子查询的结果没有名字，必须给它取一个元组变量别名。于是可以像引用FROM子句中关系的元组一样引用子查询结果中的元组。

例6.22 重新考虑例6.20中的问题。在该例中使用一个查询来找出Harrison Ford演过的电影的制片人。如果有一个关系给出了这些电影的制片人的证书号，那么找出关系MovieExec中制片人的名字将变得很简单。图6-11给出这样一个查询。

```
1)  SELECT name
2)  FROM MovieExec, (SELECT producerC#
3)                   FROM Movie, StarsIn
4)                   WHERE title = movieTitle AND
5)                         year = movieYear AND
6)                         starname = 'Harrison Ford'
7)                  ) Prod
8)  WHERE cert# = Prod.producerC#;
```

图6-11 通过在FROM子句中使用子查询找出Ford出演电影的制片人

图中第2到第7行是外查询的FROM子句。除了MovieExec关系以外，它还有一个子查询。该子查询在第3到第5行连接关系Movie和StarsIn。加上第6行的条件要求影星是Harrison Ford。在第2行返回制片人的集合。这个集合在第7行取了一个别名Prod。

第8行将关系MovieExec和别名为Pord的子查询通过证书号相同连接在一起。第1行语句从MovieExec中返回制片人的名字，该制片人的证书号在别名为Prod的集合中。 □

6.3.6 SQL的连接表达式

可以通过不同的连接运算符作用在两个关系上创建新的关系。这些不同的运算包括笛卡儿积、自然连接、θ连接和外连接。连接结果本身可以看做是一个查询。另外，由于连接表达式产生的是一个关系，所以可以在select-from-where句子中的FROM子句中当作子查询使用。 270

形式上最简单的连接是*交叉连接*（cross join）；这个术语和在5.2.5节中提到的笛卡儿积或“积”都是同一个意思。例如，如果需要下面两个关系的积

```
Movie(title, year, length, inColor, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
```

⊖ 这是第一次出现这种情况。记住在SQL中，关系不是集合而是要想像成一个包。还有几种其他的情况需要在SQL关系中用到复制。在第6.4节中将讨论这个问题。

可以写作

```
Movie CROSS JOIN StarsIn;
```

结果是包含关系Movie和StarsIn所有属性的有九个列的关系。连接结果关系中的每个元组由一对分别来自Movie和StarsIn中的元组组成。

关系积中的属性可以称为$R.A$，其中R是两个连接关系中的一个，A是它的一个属性。像以前提到的一样，如果只有一个关系具有属性A那么R和后面的点可以省略。在上面的例子中，两个关系没有同名属性，积中直接使用属性名就可以了。

通常关系积运算很少在实际中使用。更常用的是通过保留字ON来使用θ连接运算，即在关系R和S之间放一个JOIN保留字后面再跟一个保留字ON和一个条件。JOIN...ON的意义是在积运算$R \times S$的基础上再使用ON后面的条件进行选择运算。

例6.23 使用下面两个关系

```
Movie(title, year, length, inColor, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
```

进行连接运算，连接条件是同一部电影的元组才进行连接，也就是说来自两个关系电影名和年份属性相同。该查询语句如下：

```
Movie JOIN StarsIn ON
        title = movieTitle AND year = movieYear;
```

结果仍然是具有九列的关系和同样的属性名。不同的是对于来自StarsIn的元组和来自Movie的元组，只有当它们的电影名和年份相同时才合并成为新的关系中的元组。结果中有两个列是冗余的，因为结果中每个元组的title和movieTitle字段值及year和movieYear字段值相同。

对于上面连接中包含有冗余而不满意的话，可以把整个连接表达式作为一个子查询在
271 FROM子句中使用，然后使用SELECT子句去掉不需要的属性。这样，该查询可写成

```
SELECT title, year, length, inColor, studioName,
        producerC#, starName
FROM Movie JOIN StarsIn ON
        title = movieTitle AND year = movieYear;
```

该查询返回一个七个列的关系，其中每个元组可以看做是一个Movie关系的元组，加上演过该部电影的某个影星，并以所有可能的方式扩展而成。 □

6.3.7 自然连接

回忆5.2.6节所讲，自然连接不同于θ连接之处在于：

1. 自然连接是对两个关系中具有相同名字并且其值相同的属性作连接，除此之外再没有其他的条件。

2. 两个等值的属性只投影一个。

SQL自然连接也是按照这种方式进行。NATURAL JOIN出现在两个关系之间所表达的意义和$\bowtie$操作符的意义相同。

例6.24 假定需要计算下面两个关系的自然连接

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

结果关系模式中包括属性name和address，再加上所有出现在这两个关系中的其他属性。结果中的元组表示既是影星也是制片人的那些人。并且还包括所有相关的属性：姓名、地址、性别、生日、证书号和净资产等。表达式

```
MovieStar NATURAL JOIN MovieExec;
```

以简洁的方式表达了该关系。 □

6.3.8 外连接

外连接在4.7节中介绍过，它是一种通过在悬浮元组里填充空值来使之成为查询结果。在SQL中，也可以指定外连接。NULL用来表示空值。

例6.25 对下面两个关系进行外连接

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

SQL使用标准的外连接，它对两个关系的悬浮元组都进行填充，即它是一个完全外连接。语法形式为： 272

```
MovieStar NATURAL FULL OUTER JOIN MovieExec;
```

该运算结果和例6.24相同，也是一个具有6个属性的关系。关系中的元组可以分为三类。第一类表示既是影星又是制片人的人，这部分元组包含的六个属性都不为空，和例6.24中返回的结果相同。

第二类元组表示是影星但不是制片人的一类人。来自关系MovieStar的属性name、address、gender和birthdate包含有值，而来自关系MovieExec的属性cert#和netWorth都是NULL值。

第三类元组表示是制片人但不是影星的一类人。来自关系MovieExec的属性都包含有值，但是来自关系MovieStar的属性如gender和birthdate都是NULL值。例如，图6-12包含的三个元组分别对应了三种类型的人。 □

name	*address*	*gender*	*birthdate*	*cert#*	*networth*
Mary Tyler Moore	Maple St.	'F'	9/9/99	12345	$100···
Tom Hanks	Cherry Ln.	'M'	8/8/88	NULL	NULL
George Lucas	Oak Rd.	NULL	NULL	23456	$200···

图6-12 对MovieStar和MovieExec进行外连接产生的三个元组

在5.4.7节提到的所有不同的外连接在SQL中都提供。如果想要左或右外连接，用保留字LEFT或RIGHT放在FULL的位置即可。例如：

```
MovieStar NATURAL LEFT OUTER JOIN MovieExec;
```

输出图6-12中的前两个元组但不输出第三个。同样地

```
MovieStar NATURAL RIGHT OUTER JOIN MovieExec;
```

输出图6-12的第一个和第三个元组，不输出第二个。

如果希望是一个θ外连接而不是一个自然外连接。这时不使用保留字NATURAL，而是在连接后面加ON保留字和一个所有元组都服从的条件。如果指定了FULL OUTER JOIN，那么在对两个连接关系进行匹配以后，对每个关系中的悬浮元组填充NULL值，并把它们包括在结果中。

例6.26 重新考虑6.23的例子，该例中对关系Movie和StarsIn进行连接操作，条件是分

别来自两个关系的title和movieTitle属性值相同，以及year和movieYear属性值相同。
273 如果修改该例使之成为一个完全外连接:

```
Movie FULL OUTER JOIN StarsIn ON
    title = movieTitle AND year = movieYear;
```

那么结果关系中不但包括那些至少有一个影星在StarsIn中出现的电影的元组，同时也包括没有影星的电影的元组，这些元组的movieTitle、movieYear和starName属性都为NULL值。同样地，对于没有出现在关系Movie的任何一部电影中的影星也用一个元组列出，其来自Movie关系的六个属性为NULL。 □

外连接中的保留字FULL可以由RIGHT或LEFT取代。例如

```
Movie LEFT OUTER JOIN StarsIn ON
    title = movieTitle AND year = movieYear;
```

给出了有影星出现的Movie元组和用NULL值填充的没有影星出现的Movie元组。但是不会包括不在任何电影中出现的影星。相反地，

```
Movie RIGHT OUTER JOIN StarsIn ON
    title = movieTitle AND year = movieYear;
```

将会省去那些没有任何影星出现的电影的元组，但是会包括没有在任何一部电影中出现的影星，并在相应字段填上NULL值。

6.3.9 习题

习题6.3.1 基于习题5.2.1的数据库模式写出后面的查询

```
Product(maker, model, type)
PC(model, speed, ram, hd, rd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

每题的答案中，你应当至少使用一个子查询，并且使用两种不同的方法写出每个查询（例如，使用各种不同的操作符EXISTS、IN、ALL和ANY）

* a) 找出速度在1200以上的PC的制造商。

b) 找出价格最高的打印机。

! c) 找出速度比任何一台PC都慢的手提电脑。

! d) 找出具有最高价格的产品（PC、手提电脑或打印机）的型号。

274 ! e) 找出最低价格的彩色打印机的制造商。

!! f) 在所有的PC中，找出具有最快速度并具有最少RAM的PC制造商。

习题6.3.2 基于习题5.2.4的数据库模式写出后面的查询

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

每题的答案中，你应当至少使用一个子查询，并且使用两种不同的方法写出每个查询（例如，使用各种不同的操作符EXISTS、IN、ALL和ANY）。

a) 找出拥有火炮数量最多的船只所属的国家。

*! b) 找出至少有一艘船在战役中被击沉的船只种类。

c) 找出具有16英寸口径火炮的船只的名字。

d) 找出Kongo类型船只参加的战役。

!! e) 找出具有相同口径火炮的船只中火炮数量最多的船只名字。

! **习题6.3.3** 不用子查询写出图6-10的查询。

! **习题6.3.4** 考虑关系代数中表达式$\pi_L(R_1 \bowtie R_2 \bowtie ... \bowtie R_n)$，其中$L$是仅属于$R_1$的属性的列表。证明仅用子查询SQL语句也能写出该表达式。然后再用一个等价的SQL语句写出查询，该句中每个FROM子句列表中不超过一个关系。

! **习题6.3.5** 不使用交或差运算符写出下面的查询。

* a) 图6-5的交查询。

b) 例6-17的差查询。

!! **习题6.3.6** 注意到SQL的某些运算符是冗余的，即它们可以被其他的一些运算符替代。例如，*s* IN *R* 能被*s* = ANY *R* 替换。证明EXISTS和NOT EXISTS是冗余的，即用一个不含有EXISTS的表达式来替换形如EXISTS *R*或NOT EXISTS *R*的表达式（不考虑*R*本身包含的EXISTS）。提示：记住在SELECT子句中可以使用常量。

习题6.3.7 对于使用的电影数据库模式 275

```
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

描述下列SQL表达式表示的元组：

```
a) Studio CROSS JOIN MovieExec;
b) StarsIn NATURAL FULL OUTER JOIN MovieStar;
c) StarsIn FULL OUTER JOIN MovieStar ON name = starName;
```

*! **习题6.3.8** 使用数据库模式

```
Product(maker, model, type)
PC(model, speed, ram, hd, rd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

写出SQL查询，该查询将会返回所有的产品（PC、手提电脑和打印机）信息，包括它们的制造商（如果有的话），以及尽可能多的其他相关信息。

习题6.3.9 使用习题5.2.4中给的两个关系

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
```

用SQL查询输出所有的关于船只的信息，包括可以获得的Classes关系中的信息。如果在关系Ships中该类型的船只不出现，就不必输出类型信息。

! **习题6.3.10** 重做习题6.3.9，在结果中增加对于任何没有在Ships中提到的类型*C*，提供船的信息并用*C*作为其类型。

! **习题6.3.11** 本节学习的连接运算符是一个冗余的运算符，它可以通过select-from-where的形式表达出来。解释如何使用select-from-where的形式写出下列查询。

* a) R CROSS JOIN S;

b) R NATURAL JOIN S;

c) R JOIN S ON C ; *C* 是一个SQL条件。 276

6.4 全关系操作

本节中，将学习把关系作为一个整体而不是单个元组或一定数量的元组进行操作(如几个关系的连接)的操作符。首先，SQL把关系当作包而不是集合使用，一个元组可以在关系中多次出现，在6.4.1节中将讨论如何把操作的结果强制转换成集合。在6.4.2节中将讨论如何阻止在SQL系统中缺省地消除重复元组。

接着，开始讨论SQL对5.4.4节中的聚集操作运算符γ的支持。SQL也有聚集运算符和GROUP-BY子句。还有一个“HAVING”子句允许以某些方式对群组进行选择。它把每个群组看做一个操作的实体而不是单个的元组。

6.4.1 消除重复

如在6.3.4节提到的，SQL中关系的概念不同于第3章中提出的关系的抽象概念。一个关系是一个集合，不能包含某个元组一份以上的拷贝。当SQL查询创造了一个新关系时，SQL系统正常情况下不消除重复的元组，这样，SQL查询返回的关系中可能多次重复出现某个元组。

回忆6.2.4节讲到SQL的一个等价的select-from-where 定义，该定义首先对FROM子句中列出的关系做积运算，积中的每一元组先使用WHERE子句中的条件进行测试，通过测试的元组再使用SELECT子句进行投影。投影可能造成积中不同的元组形成某个重复相同的元组，如果这种情况发生，这些元组都将作为投影后的结果输出。还有，SQL关系中本身可以有重复元组，做笛卡儿积后的关系就会产生重复的元组。因为每一元组还要和来自其他关系中的元组配对，所以在积运算后可能产生更多的重复元组。

如果希望结果中不出现重复的元组，可以在保留字SELECT后跟上DISTINCT。该保留字告诉SQL仅产生每个元组的一份拷贝。这个和5.4.1节中对查询结果进行δ运算相似。

例6.27 考虑图6-9的查询，在那里不使用子查询来查找Harrison Ford出演电影的制片人。该例查询结果中George Lucas的名字会在输出结果中出现多次，如果希望仅仅看到每一个制片人的名字一次，可把第一行改成

277

```
1) SELECT DISTINCT name
```

那么，制片人列表中重复出现的名字在打印输出前就去掉了。

消除重复的代价

或许有人想在每个SELECT后面放上一个DISTINCT消除重复，这样在理论上似乎不费事，但实际上从关系中消除重复的代价是非常昂贵的。关系必须排序或者分组才能保证相同的元组紧挨在一起。从15.2.2节开始将讨论这些算法。只有按该算法分组后才能决定某个元组是否可以去掉。为消除重复对元组进行排序的时间通常比执行查询的时间都长。所以如果想要查询运行得快就要谨慎地使用消除重复。

顺便提一下，图6-7的查询中由于使用到了子查询，不会出现结果中元组重复的问题。虽然图6-7中的第4行会产生George Lucas的证书号多次，但是在第一行的主查询中，MovieExec中的每个元组仅被检查一次。假定该关系中只有一个关于George Lucas的元组，该元组就是惟一满足第3行WHERE子句中条件的元组。这样，George Lucas只打印一次。 □

6.4.2 交、并、差中的重复

SELECT语句中缺省保留重复的元组，除非使用DISTINCT保留字指明。与之不同的是在6.2.5节介绍的集合操作中的并、交和差操作在缺省情况下消除重复。即将包又变成了集合。操

作的集合版本在这里适用。如要要阻止消除重复元组，必须在UNION、INTERSECT和EXCEPT后跟上保留字ALL。这样，就又回到了5.3.2节中讨论的包的并、交、差操作。

例6.28 再次考虑例6.18中的集合表达式，但是加上保留字ALL使之成为：

```
(SELECT title, year FROM Movie)
    UNION ALL
(SELECT movieTitle AS title, movieYear AS year FROM StarsIn);
```

现在，把出现在关系Movie和StarsIn中的title和year组合放在一起，导致它们将会在结果中多次出现。例如，如果某部电影在Movie关系中有一条记录，并且StarsIn关系包含出演这部电影的三位影星（因此这部电影将会在StarsIn三个不同的元组中出现），最后这 278
部电影的名字和年份将会在集合的并运算结果中出现4次。 □

对于并运算，操作INTERSECT ALL和EXCEPT ALL是包的交和差。这样，如果R和S是关系的话，那么表达式

R INTERSECT ALL S

表示的关系中，元组t出现的次数是它在R中出现的次数和在S中出现的次数的较小者。表达式

R EXCEPT ALL S

表示的关系中如果元组t多次出现，那么它出现的次数是在R中出现的次数减去在S中出现的次数，结果差为正数。以上讨论的每一个定义都是使用了5.3.2节关系的包语义。

6.4.3 SQL中的分组和聚集

在5.4.4节中，分组和聚集操作符γ被引用来扩展关系代数。在5.4.3的讨论中，使用该操作能根据元组的一个或多个属性将关系中的元组划分成“组”。然后对关系中的其他列可以使用聚集操作符进行聚集操作。如果该关系已经分组，那么聚集操作是对每个分组单独进行。SQL通过在SELECT子句中的聚集操作和特定的GROUP BY子句提供了γ操作的所有功能。

6.4.4 聚集操作符

SQL提供在5.4.2节中提到的五个聚集操作符，分别是SUM、AVG、MIN、MAX和COUNT。这些操作符通常作用在一个标量表达式上，典型的是SELECT子句中的一个列名。一个例外是表达式COUNT(*)，它计算由查询中FROM子句和WHERE子句所创建的关系中的元组个数。

另外，通过使用保留字DISTINCT可以在使用聚集操作符之前从列中消除重复元组。即如COUNT(DISTINCT x)这样的表达式将只计算列x中不重复的x的个数。可以在把COUNT替换成其他的操作符。不过像SUM(DISTINCT x)这样的表达式几乎没有什么意义，它要求计算列x中不同值的和。 279

例6.29 下面的查询是找出所有的电影制片人资产的平均值。

```
SELECT AVG(netWorth)
FROM MovieExec;
```

注意，这里没有使用 WHERE子句，保留字WHERE没有出现在句中。该查询检查关系MovieExec(name,address,cert#,netWorth)，对找到的值求和，每元组计算一次（即使该元组与其他元组重复）。然后对求得的和除以元组的数目。如果没有重复的元组，查询就会给出制片人的平均资产，这正是查询所求。但如果该关系中有重复的元组，即一个经理的元组出现了n次，那么求得的平均值中此人的资产重复计算了n次。 □

例6.30 下面的查询

```
SELECT COUNT(*)
FROM StarsIn;
```

计算StarsIn关系中元组的数目。类似的查询

```
SELECT COUNT(starName)
FROM StarsIn;
```

计算starName列中的值出现的次数。SQL中对starName列进行投影时不消除重复。这个计数的结果和COUNT(*)的结果相同。

如果想保证不对重复的值多次计数，在聚集的属性前加上DISTINCT保留字就可以了。如：

```
SELECT COUNT(DISTINCT starName)
FROM StarsIn;
```

现在，每个影星无论在多少部电影中出现都只计数一次。 □

6.4.5 分组

在WHERE子句后面加上一个GROUP BY 子句可以对元组进行分组。保留字GROUP BY后面跟着一个分组属性列表。最简单的情况是，FROM子句后面只有一个关系，根据分组属性对
280 它的元组进行分组。SELECT子句中使用的聚集操作符仅应用在每个分组上。

例6.31 从关系Movie(title,year,length, inColor,studioName, producerC#)中找出每个制片公司制作的电影总长度。可用如下语句表达

```
SELECT studioName, SUM(length)
FROM Movie
GROUP BY studioName;
```

该语句是将Movie关系的元组重新组织，并且进行分组使得所有的Disney公司的元组被组织在一起，如所有MGM公司的元组被组织在一起等，就像图5-17所展示的那样。然后计算每个分组里面的所有元组的length字段值之和。对于每一个分组，制片公司的名字和求得的和一起被打印出来。 □

从例6.31中可以看出，SELECT子句有两种概念。

1. 聚集。每个聚集运算符作用在一个属性上或涉及属性的表达式上。上面已提到，这些项目是以分组为单位计算的。

2. 属性。如例子中的studioName，这些属性同时在GROUP BY中出现。如果SELECT子句中有聚集运算，那么GROUP BY子句中出现的属性可以在SELECT子句中以非聚集的形式出现。

当查询中有GROUP BY保留字的时候，SELECT子句中通常有聚集属性和聚集运算符。但也不必两者都出现，例如

```
SELECT studioName
FROM Movie
GROUP BY studioName;
```

这个查询对Movie关系根据制片公司名称进行分组，并打印每个分组的制片公司名称。无论多少个元组使用了这个制片公司名称，它只打印一次。这样，上面的查询和语句

```
SELECT DISTINCT studioName
FROM Movie;
```

有同样的效果。

也可以在多关系查询中使用GROUP BY子句。这种查询可以由下面几步来说明。

1. 计算由FROM和WHERE子句所表达的关系R。也就是说，R是对FROM子句中的关系进行积运算后再用WHERE中的条件进行选择操作后形成的关系。 281

2. 根据GROUP BY子句中的属性对R进行分组操作。

3. 把SELECT子句中的属性和聚集运算作为查询结果输出，就好像查询是在一个存储关系R上进行的。

例6.32 假定想要打印每个制片人制作电影的总长度，那么需要用到两个关系

```
Movie(title, year, length, inColor, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

因此，开始使用θ连接，要求两个关系的证书号（cert#和producerC#）相同。这一步给出了一个关系，该关系中每一个MovieExec元组和Movie中的元组连接，条件是MovieExec元组表示的人是这些电影的制片人，否则元组不会和任何Movie中的元组配对的，也就不会在结果关系中出现。现在，可以根据制片人的名字对从关系中选择出来的元组进行分组。最后，对分组中电影的长度求和。图6-13给出了查询。 □

```
SELECT name, SUM(length)
FROM MovieExec, Movie
WHERE producerC# = cert#
GROUP BY name;
```

图6-13 计算每个制片人的电影长度之和

6.4.6 HAVING子句

假设不希望查询所有在例6.32的结果中出现的制片人。此时，元组在分组之前就按某种方式加上限制，使得不需要的分组为空。例如，如果只统计资产在$10 000 000以上制片人的制作的电影长度，则将图6-13的第三行变为

```
WHERE producerC# = cert# AND networth > 10000000
```

另外，如果分组时需要考虑某些聚集操作，可以在GROUP BY子句后面加上一个HAVING子句。后者由保留字HAVING后跟一个分组的条件组成。

例6.33 假定需要打印至少曾在1930以前制作过一部电影的制片人制作的电影总长度，则在图6-13后面加上子句HAVING MIN(year) < 1930 282

分组、聚集和空值

当元组含有空值时，要记住下面几个规则：

- 空值在任何聚集操作中都被忽视。它对求和、取平均和计数都没有影响。它也不能是某列的最大值和最小值。例如，COUTN(*)是某个关系中所有的元组数目之和，但是COUNT(A)却是A属性非空的元组个数之和。
- 另一方面，NULL值又可以在分组属性中看做是一个一般的值。例如SELECT a, AVG(b) FROM R 中当a的属性值为空时，就会统计a=NULL的所有元组中b的均值，如果R中至少有一个元组的a属性值为空的话。

最后的查询语句在图6-14中给出。该查询将会去掉那些year字段值大于或等于1930的元组分组。 □

```
SELECT name, SUM(length)
FROM MovieExec, Movie
WHERE producerC# = cert#
GROUP BY name
HAVING MIN(year) < 1930;
```

图6-14 计算各个早期制片人的电影长度和

下面几个关于HAVING子句的几条规则应当记住。

- HAVING子句中的聚集只应用到正在检测的分组上。
- 所有FROM子句中关系的属性都可以在HAVING子句中用聚集运算，但是只有出现在GROUP BY子句中的属性，才能以不聚集的方式出现在HAVING子句中（和SELECT子句同样的规则）。

283

SQL查询中的子句顺序

到目前为止，已经讨论了六种可能出现在select-from-where查询中的子句。分别是：SELECT、FROM、WHERE、GROUP BY、HAVING和ORDER BY。但是，只有前两个是必须的，而且，在不使用GROUP BY子句的情况下不能使用HAVING子句。无论哪个子句都应该按照上面给定的顺序出现。

6.4.7 习题

习题6.4.1 用SQL写出习题5.2.1中的查询，结果中去掉重复的元组。

习题6.4.2 用SQL写出习题5.2.4中的查询，结果中去掉重复的元组。

! **习题6.4.3** 查看习题6.3.1的答案，看看你写的查询结果中是否有重复元组出现。如果有的话重写该查询保证结果没有重复；如果没有，不使用子查询，再重写一个结果中不出现重复的查询。

! **习题6.4.4** 按习题6.4.3的要求重做习题6.3.2。

*! **习题6.4.5** 在例子6.27中，提到查询“找出Harrison Ford出演电影的制片人”的不同版本。它们产生的结果集合是相同的，但是却有不同的“包”。考虑例6.22中的查询版本，它在FROM子句中使用了一个子查询。这个版本的查询是否会产生重复？如果是，为什么？

习题6.4.6 根据习题5.2.1的数据库模式写出后面的查询，再用该习题给出的数据算出该查询结果。

```
Product(maker, model, type)
PC(model, speed, ram, hd, rd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

* a) 查询PC速度的平均值。

b) 查询价格在$2000以上的手提电脑的平均值。

c) 查询制作商A生产的PC的平均价格。

! d) 查询制造商D生产的PC和手提电脑的平均价格。

284

e) 求不同速度的PC平均价格。

*! f) 找出各个制造商生产的手提电脑的平均价格。

! g) 找出至少生产3种不同型号PC的制造商。

! h) 找出各个制造商制造的PC的最高销售价格。

*! i) 找出速度在800以上的PC的平均价格。

!! j) 对于所有生产打印机的制造商，查询其生产的PC的硬盘平均大小。

习题6.4.7 基于习题5.2.4给出的数据库模式和数据写出后面的查询语句以及查询结果。

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

a) 找出战舰类型的数量。

b) 找出不同类型战舰拥有的平均火炮数量。

! c) 找出战舰的平均火炮数量。注意c）和b）的不同在于：在计算均值的时候，是使用战舰的数目还是战舰的类型数目。

! d) 找出每一类型(class)的第一艘船下水的年份。

! e) 找出每一类型中被击沉船的数目。

!! f) 找出至少有3艘船的类型中被击沉的船的数目。

!! g) 军舰火炮使用的炮弹的重量（以磅为单位）大约是火炮的口径（以英寸为单位）的一半。找出各个国家的军舰炮弹重量的平均值。

习题6.4.8 在例子5.23中，给出了一个查询："找出每个至少演过3部电影的影星最早出演的时间。"该例中使用了γ运算符完成查询，现在请用SQL完成这一功能。

*! **习题6.4.9** 扩展的关系代数中的γ运算符并不和SQL中的HAVING子句完全对应。是否可能在关系代数中模仿SQL中HAVING子句的功能？如果可以，通常情况下该怎样处理？ 285

6.5 数据库更新

到目前为止本章的讨论都集中于一般的SQL查询形式，即select-from-where句型。还有许多其他形式的不返回查询结果的句子，这些句子只改变数据库的状态。在本节中，将具体讨论如下三种类型的句子。

1. 插入元组到关系中去。
2. 从关系中删除元组。
3. 修改某个元组的某些字段的值。

6.5.1 插入

插入语句的基本形式由以下几项构成。

1. 保留字INSERT INTO。
2. 关系R的名字。
3. 关系R中括起来的属性列表。
4. 保留字VALUES。
5. 一个元组表达式，即括起来的具体的值的列表。每一个值针对列表(3)中的属性。

基本的插入格式为：

$$\text{INSERT INTO } R(A_1,\ldots,A_n) \text{ VALUES}(v_1,\ldots,v_n);$$

该插入将创建一个元组，其属性A_i值是$v_i, i=1,2,\ldots,n$。如果属性列表不包括关系R中所有的属性，那么对这些没有赋值的属性给一个缺省的值。最常使用的缺省值是空值NULL，在6.6.4节中还有一些其他的选项。

例6.34 如果要把Sydney Greenstreet加到The Maltese Falcon演员列表中去，可以这样写插入语句

```
1) INSERT INTO StarsIn(movieTitle, movieYear, starName)
2) VALUES('The Maltese Falcon', 1942, 'Sydney Greenstreet');
```

该语句的执行结果是具有第2行的三个字段值的元组被插入到关系StarsIn中。由于关系StarsIn中所有的属性都在一行列出，也就没有必要使用缺省的属性值。第2行的值和第1行的属性按给定的顺序对应。因此'The Maltese Falcon'成为属性movieTitle的值，其
286 他的也是如此类推。 □

如果像在例子6.34那样，关系的所有属性值都给出了，那么可以省掉跟在关系名后面的属性列表。其插入语句可以写成

```
INSERT INTO StarsIn
VALUES('The Maltese Falcon', 1942, 'Sydney Greenstreet');
```

注意，当省略属性列表时，一定要保证提供的属性值的顺序和关系属性的标准顺序一致。6.6节中将介绍如何声明关系模式。声明关系的同时给出了关系中的属性排列顺序，这个顺序在省略属性列表时作为默认的排列顺序。

- 如果不能确定属性的标准排列顺序，最好在INSERT句子中按照VALUES子句中属性值的顺序给出属性列表。

上面给出的简单INSERT语句仅仅插入一个元组到关系中去。除了明确地指定值来插入一个元组以外，还可以往关系中插入使用子查询计算出的元组集合。该子查询语句替换了上面给出的INSERT语句中的保留字VALUES和后面的元组表达式。

例6.35 往关系Studio(name, address, presC#)中添加在关系Movie(title, year, length, inColor, studioName, producerC#)中提到的，但没有在Studio出现的所有的制片厂。由于Movie中没有给出制片厂的地址和制片经理，插入到Studio的元组的属性address和presC#只好使用NULL值。图6-15给出了一种插入方式。

```
1)  INSERT INTO Studio(name)
2)      SELECT DISTINCT studioName
3)      FROM Movie
4)      WHERE studioName NOT IN
5)          (SELECT name
6)           FROM Studio);
```

图6-15 添加新的制片厂

287 和其他的嵌套SQL语句一样，图6-15从里层往外分析较容易。第5和第6行在关系Studio中产生所有的制片厂的名字。第4行检查来自Movie关系的制片厂的名字是否包含在Studio中。这样，从第2行到第6行输出存在于关系Movie中但不在Studio中的制片厂名字的集合。第2行的DISTINCT保留字保证无论拍过多少部电影，每个制片厂在集合中只出现一次。最后，第1行语句把这些制片厂插入到关系Studio中，属性address和presC#用NULL值填充。 □

插入的时机选择

图6-15指出了SQL语句语义中的一个细节问题。原则上，第2行到第6行的查询计算应当在执行第1行的插入前完成。这样插入到Studio中的新元组不会影响到第4行的执

行。然而，有时出于提高效率的考虑，某些实现可能会使得在执行第2行到第6行时，一找到新的元组就插入到关系Studio中，使之立刻发生变化。

这个例子比较特殊，插入是否要等到查询完全计算结束后对结果没什么影响。但有一些其他查询的结果受插入时间的影响。例如，如果从图6-15中去掉DISTINCT保留字。在插入前计算第2行到第6行的查询。结果会导致某个多次出现在Movie中的新制片厂的名字在查询结果中多次出现，并多次插入到关系Studio中去。然而，如果在计算第2行到第6行查询的同时将找到的新的制片厂插入到Studio中去，那么新的制片厂就不会重复插入。因为新的制片厂一旦添加一次，StudioName的值就不会满足第4行到第6行的条件，也就不会在第2行到第6行的查询结果中再次出现。

6.5.2 删除

删除语句由以下几项构成

1. 保留字DELETE FROM;
2. 关系名*R*;
3. 保留字WHERE;
4. 一个条件。

288

删除语句的格式为

```
DELETE FROM R WHERE <condition>;
```

该语句的查询结果将是每个满足条件（4）的元组从关系*R*中删掉。

例6.36 从关系StarsIn(movieTitle,movieYear,starName)删掉Sydney Greenstreet出演电影*The Maltese Falcon*这一事实。删除语句为

```
DELETE FROM StarsIn
WHERE movieTitle = 'The Maltese Falcon' AND
      movieYear = 1942 AND
      starName = 'Sydney Greenstreet';
```

注意，和例子6.34中的插入语句不同，不能简单地指定把某个元组删掉。必须通过WHERE子句明确地描述要删除的元组。□

例6.37 这里是删除语句的另一个例子。这次是从关系

```
MovieExec(name, address, cert#, netWorth)
```

一次删掉多个元组。给出的条件可能有多个元组满足。语句

```
DELETE FROM MovieExec
WHERE netWorth < 10000000;
```

删掉所有的资产低于一千万的制片经理。□

6.5.3 更新

虽然可以把对数据库的更新想成是插入和删除操作的组合，但在SQL中，更新（update）是数据库一种特定改变。数据库中已存在的一个或多个元组的某些字段值发生改变。更新语句的一般构成为：

1. 保留字UPDATE;
2. 关系名*R*;

289

3. 保留字SET;

4. 一组公式，设置关系*R*中的属性等于某个表达式或常量值;

5. 保留字WHERE;

6. 一个条件。

更新语句的格式为

UPDATE *R* SET <new-value assignments>WHERE <condition>;

每一个赋值（第4项）由一个属性、一个等号和一个公式组成。如果存在多个赋值，就用逗号隔开。

该语句的执行结果是找出所有满足条件（6）的元组，然后对于找到的元组根据(4)列出的公式进行更新，赋给*R*中元组相应属性指定的值。

例6.38 更新关系

```
MovieExec(name, address, cert#, netWorth)
```

在每个还是制片厂制片经理的制片人名字前加上称呼Pres.。MovieExec中待修改元组满足的条件是导演的证书号出现在关系Studio中某些元组的的presC#字段值上。该修改语句为:

```
1)  UPDATE MovieExec
2)  SET name = 'Pres. ' || name
3)  WHERE cert# IN (SELECT presC# FROM Studio);
```

第3行检查MovieExec中的某个证书号是否作为关系Studio中某个制片厂经理的证书号出现。

第2行对选定的元组执行修改操作。前面讲过||操作符表示字符串的连接。这样，在元组旧的name字段值前加上了字符串Pres.和一个空格。新的字符串成为该元组新的name字段值，效果上就是'Pres.'加在了旧的name字段值前面。 □

6.5.4 习题

习题6.5.1 根据习题5.2.1给出的数据库模式，写出下面的数据库修改。描述对该习题数据修改后的结果。

```
Product(maker, model, type)
PC(model, speed, ram, hd, rd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

290

a) 通过两条INSERT语句在数据库中添加如下信息：制造商C生产的型号为1100的PC，速度为1800，RAM为256，硬盘大小80，具有一个20x的DVD，售价为$2499。

! b) 加入如下信息：对于数据库中的每台PC，都对应一台与其速度、RAM、硬盘相同，具有一个15英寸的屏幕，型号大于1100、价格高于$500的相同厂商制造的手提电脑。

c) 删除所有硬盘不超过20G的PC。

d) 删除所有不制造打印机的厂商生产的手提电脑。

e) 厂商A收购了厂商B。将所有B生产的产品改为由A生产。

f) 对于每台PC，把它的内存加倍并且增加20G的硬盘容量。（记住UPDATE语句中可以同时更改多个属性的值）

! g) 把厂商B生产的手提电脑的屏幕尺寸增加一英寸并且价格下调$100。

习题6.5.2 根据习题5.2.4给出的数据库模式，写出下面的数据库更新语句，描述对该习题数据修改后的结果。

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

* a) 两艘Nelson类型的英国战舰-Nelson和Rodney-在1927年下水。两者都具有16英寸口径的火炮，排水量为34 000吨。把这条信息加入到数据库中。

b) 两艘Vittorio Veneto类型的意大利战舰-Vitorio Veneto和Italia-在1940年下水；第三艘同样类型的船-Roma在1942年下水。每艘船都有15英寸口径火炮和41 000吨的排水量。把这些信息加入到数据库中。

* c) 删除所有在战役中沉没的船只。

* d) 更新Classes关系使得火炮口径使用厘米作为单位（1英寸 = 2.5厘米），排水量使用公制吨。（1公制吨 = 1.1吨）。

e) 删除所有少于3艘船的类型。

6.6 SQL中的关系模式定义

本节中将讨论*数据定义*（data definition），这部分的SQL涉及到数据库里面的信息结构的描述。与之对应的前面所讲的SQL部分-查询和修改-称为*数据操纵*（data manipulation）[⊖]。

本节的主题是存储关系模式的声明。在此将会看到如何描述一个新的关系或SQL中提到的表（table）。6.7节讲述视图的声明，它是一种并不实际存储在数据库中的虚关系。一些关于关系约束的更复杂的问题放到了第7章讲述。

6.6.1 数据类型

先介绍SQL系统支持的基本原子类型。关系中所有的属性都必须有一个数据类型。

1. 可变长度或固定长度字符串。类型CHAR(n)表示*n*个字符的固定长度字符串。如果某个属性的类型是CHAR(n)，那么关系中每个元组的该属性值都是具有*n*个字符的字符串。VARCHAR(n)表示最多可以有*n*个字符的字符串。SQL允许合理地在不同字符串类型之间作类型转换。通常，当某字段值的字符个数比定义的类型少时在后面补上空格字符。例如，字符串'foo'成为某个类型为CHAR(5)的字段值的时候，就认为它是值为'foo'的字符串（后面跟上两个空格）。但和其他字符串比较的时候，后面的空格又可以忽略（参考6.1.3节）。

2. 固定或可变长度的位串。位串和字符串类似，但是它们的值是由比特而不是字符组成。类型BIT(n)表示长为*n*的位串。BIT VARYING(n)表示最大长度为*n*的位串。

3. BOOLEAN表示具有逻辑类型的值。可能是TRUE、FALSE和UNKNOWN（这一点可能会令George Boole吃惊）。

4. 类型INT和INTEGER（两者为同义词）表示典型的整数值。类型SHORTINT也表示整数，但是表示的位数可能小些，具体取决于实现。（类似C语言中的int和short int）。

5. 浮点值能通过不同的方式表达。类型FLOAT和REAL（两者为同义词）表示典型的浮点

[⊖] 从技术的角度来说，本节讨论的内容应该是数据库设计领域的内容，应当在本书的前面部分就应该讲到。它和ODL在面向对象数据库中情况类似。然而为了使得SQL的内容成为一个整体，这样安排也是合理的。所以在本书的内容安排上做了一点调整。

数值。需要高精度的浮点类型可以使用DOUBLE PRECISION；这些类型的区别也和C语言中类似。SQL还提供指定小数点后位数的浮点类型。例如DECIMAL(n,d)允许可以有n位有效数字的十进制数，小数点是在右数第d位的位置。例如 0123.45就是符合类型DECIMAL(6,2)定义的数值。NUMERIC几乎是DECIMAL的同义词，尽管存在某些依赖于实现的小差别。

6. 日期和时间分别通过DATE和TIME数据类型来表达。在6.1.4节中讨论过日期和时间值。它们实际上是某个特定格式的字符串。可以把时间和日期强制转换成字符串类型。反之也可以把某些格式字符串转换成日期和时间。

6.6.2 简单表定义

最简单的关系模式的定义形式是由保留字CREATE TABLE后面跟上关系名和括起来的由属性名和类型组成的列表。

例6.39 前面使用的例子里面的一个关系MovieStar，在5.1节中给出了不是很规范的描述。图6-16用SQL给出了定义。头两个属性name和address都声明成字符串类型。但对于name属性，使用了长度为30的固定长度字符串，长度小于30的名字在后面填上空格，长度大于30的字符串则截断成为30个字符长的字符串。与name属性不同，address属性声明为最大长度为255[⊖]的可变长度字符串，这种做法不一定是最好的选择，这里做的目的是为了说明两种不同字符串类型。

属性gender只用一个字母M或F描述。这样，用一个字符长度的字符串足以描述此属性。最后面的birthdate属性自然要使用DATE类型。如果某系统不完全支持SQL标准，没有提供DATE数据类型，可以使用CHAR(10)来代替。事实上日期类型就是用10个字符长的字符串表
293 示，即八个数字和两个连字符。 □

```
1)  CREATE TABLE MovieStar (
2)      name CHAR(30),
3)      address VARCHAR(255),
4)      gender CHAR(1),
5)      birthdate DATE
    );
```

图6-16 定义MovieStar的关系模式

6.6.3 修改关系模式

要删除某个关系可以使用SQL语句

```
DROP TABLE R;
```

此后关系R不再是数据库模式中的一部分。关系中的元组也再无法访问。

对于一个长期使用的数据库，一般很少删除其中的某个关系。通常可能需要对某些已存在的关系模式进行修改。修改操作的语句通常以保留字ALTER TABLE加上关系的名字开头。后面可以跟几种选项。最重要的两种是：

1. ADD后面跟上列的名字和数据类型。
2. DROP后面跟上列的名字。

例6.40 修改关系MovieStar，给它增加属性phone，语句为：

⊖ 255并不是某些说法中作为专门使用的特殊意义的数字。单个字节能保存0到255大小的整数。所以可以使用一个字节表示最大长度为255的可变长度的字符串的字符个数。再加上表示字符串本身的字符来一起表示变长字符串。但是商业数据库系统通常都支持更大长度的可变长字符串。

```
ALTER TABLE MovieStar ADD phone CHAR(16);
```

该语句的结果是MovieStar现在具有5个属性；前4个在图6-16中提到。第5个属性phone具有固定长度为16的字符串。在实际的关系中，元组都具有phone字段。但是该字段中没有电话号码值。所以，该字段的值将会为空。在6.6.4节中，将介绍不使用NULL而使用其他的方式来去确定默认值。

下面的例子中，关系的birthdate属性将被删掉。

```
ALTER TABLE MovieStar DROP birthdate;
```

□ 294

6.6.4 默认值

当元组被创建或修改时，并非总是给它的每个字段指定值。例如，在例子6.40中给关系增加一列时，关系中已存在的元组对应此列没有一个具体的值，通常就使用NULL值来替代该位置上某个实际值。在例子6.35中插入到关系Studio中的新元组仅知道制片厂名字而不知道地址和经理的证书号。还有，后面的两个属性有时的确要使用某个值表示“目前未知”而不用某个实际的值。

为了处理这些情况。SQL提供了NULL值，它表示关系属性值还没有指定。也有一些例外，在那里NULL值不允许被使用（见7.1小节）。然而，有些情况下需要选择使用默认值(default)，它是在值未指定或未知的情况下使用的值。

通常，在声明属性和其数据类型的地方，都可以加上保留字DEFAULT和一个合适的值。该值一般要么是NULL，要么是常量。系统还提供其他几种值，如当前时间，也可以是一种选择。

例6.41 重新考虑例6.39。希望使用字符？作为属性gender未知时的默认值，使用一个可能最早的日期DATE' 0000-00-00' 作为某个未知birthdate的默认值。这只需把图6-16的第(4)和第(5)行换成下面两行即可：

```
4)    gender CHAR(1) DEFAULT '?',
5)    birthdate DATE DEFAULT DATE '0000-00-00'
```

至于其他的例子，如例6.40中，当增加新属性phone时，可以把它的的默认值声明成' unlisted'。改变默认值的语句为：

```
ALTER TABLE MovieStar ADD phone CHAR(16) DEFAULT 'unlisted';
```

□

6.6.5 索引

关系属性A上的索引是一种数据结构，它可以提高查找在属性A上具有某个特定值的元组时的效率。索引通常有助于包含有属性A和常量比较的查询，例如$A=3$或$A\leqslant 3$。大型关系数据库中索引的实现技术是数据库管理系统实现中最重要的核心问题。第13章讲述这方面的内容。

当关系变得很大时，通过扫描所有关系中所有的元组来找出那些（可能数量很少）匹配给定条件的元组的操作方式代价太高。例如，考虑例6.1的查询： 295

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

关系中可能存在大约10 000部电影元组，但只有大约200部是1990年制作的。

实现这个查询最简单的方式是取得所有的10 000个元组，并用WHERE子句逐个测试每个元

组。但如果有某种方法取得1990年的200个元组并逐个测试制片厂是否是Disney，那么查询效率就大大提高。更有效的方法是能直接取得满足两个条件的10个左右的元组-制片厂是Disney，制作年份是1990年。具体参阅后面的“多重索引”介绍。

虽然到目前为止，索引的创建还不是任何SQL标准（包括SQL-99）的一部分，但是大部分商用系统提供给数据库设计者一些机制，用来对关系的某个属性创建索引。下面的句子是典型的索引创建语句。假定要给关系Movie的Year属性创建一个索引，创建语句可以写成：

```
CREATE INDEX YearIndex ON Movie(year);
```

该语句的结果是在关系Movie的属性year上创建一个名为YearIndex的索引。这样，SQL查询处理器在处理指定年份的查询的时候，仅仅对指定年份的元组进行测试，使获得查询结果的时间大大缩短。

通常，DBMS允许对多个属性创建一个索引。这种类型的索引使用几个属性的值进行查找，并能有效地找到给出这些属性值的元组。

例6.42 由于title和year是Movie的键，所以通常这两个属性要么同时指定，要么一个也不指定。下面是对这两个属性声明的一个索引。

```
CREATE INDEX KeyIndex ON Movie(title, year);
```

既然（title,year）作为键，所以当给出某个title和year时，只有一个符合需要的元组返回。如果查询同时指定title 和year，但是只有一个YearIndex可以使用，那么最好的情况是，系统先返回该年份的元组，然后逐个检查每个元组的title是否为给定值。

通常情况是，如果多重索引中的键值是某些属性按特定顺序的组合，那么可以使用这个索引找出给定属性列表中前面的属性值的全部元组。这样，多重索引的设计即是属性列表顺序的选择。例如，如果查找时对电影名的指定比年份的指定多，就使用上面定义的多重索引。如果对电影年份的指定比对电影名的指定多，最好创建一个建立在（year,title）上的索引。□

296

如果想删除索引，则使用下面这样的语句删除索引名即可。

```
DROP INDEX YearIndex;
```

6.6.6 索引选择简介

数据库设计者需要对索引做一个折中的选择。这种选择是衡量数据库设计成败的一个重要因子。设计索引时要考虑两个重要的因素：

- 对某个属性使用索引能极大的提高对该属性上的值的检索效率，使用到该属性时，还可以加快连接。
- 另一方面，对关系上某个属性的索引会使得对关系的插入、删除和修改变得复杂和费时。

索引的选择是数据库设计中最困难的一部分，因为它需要估计对数据库上使用什么样的查询组合以及其他操作。如果对某个关系的查询操作比对它的更新操作多，那么建立在该关系上的索引具有较高的效率。对于经常和查询where子句中的常量作比较的属性，以及频繁出现在连接条件中的属性建立索引非常有用。

例6.43 在图6-3中，计算连接时使用了穷举的方式查找配对元组。建立在字段Movie.title上的索引有助于快速地找出*Star Wars*对应的电影元组。在找到它的证书号后，建立在MovieExec.cert#上的索引，有助于快速地找到出现在MovieExec关系中对应证书号的人。□

如果更新操作特别频繁，那么创建索引要谨慎。即便如此，对于频繁使用的属性创建索引也是值得的。事实上，某些更新操作也需要查找数据库。（例如，带有select-from-where子查询的INSERT语句和带有条件的DELETE语句）。必须仔细地估算更新和查询数量的相对比例来决定对索引的使用。

数据如何存储以及索引是如何实现的这些细节问题还没有讲到，因为这需要对数据库有一个总体认识。在下面的例子中能看出部分问题。因为关系通常是存储在多个磁盘块上的，查询和更新的主要开销是把磁盘块的数据传到主存涉及的磁盘块数（参阅11.4.1节）。利用索引就可以不 297
用检查整个关系找到某个元组，将极大地节省时间。然而，索引本身也需要占用磁盘空间，访问和更新索引结构本身也需要磁盘操作。事实上，由于更新操作需要一次磁盘块读操作和一次写操作改变盘块的内容，它所需的磁盘访问次数是单独的读操作访问索引或查询数据时的两倍。

例6.44 考虑下面的关系

```
StarsIn(movieTitle, movieYear, starName)
```

假定在关系上执行3种数据库操作。

Q_1：查找某个影星演过的电影的名字和年份，使用下面形式的查询：

```
SELECT movieTitle, movieYear
FROM StarsIn
WHERE starName = s;
```

其中s是一个常量。

Q_2：找出给定的某部电影的名字。使用下面形式的查询：

```
SELECT starName
FROM StarsIn
WHERE movieTitle = t AND movieYear = y;
```

其中t和y是常量。

I：插入新的元组到关系StarsIn.使用下面形式的插入语句：

```
INSERT INTO StarsIn VALUES(t, y, s);
```

其中t，y和s是常量。

对于数据作下列假定：

1. StarsIn存储在10个磁盘块中，如果要检查整个关系，代价是10。

2. 平均每部电影有3个影星，每个影星出现在3部电影中。 298

3. 由于对于某个给定的影星或某部给定的电影，其相关元组可能分布在10个盘块之中，即使在starName或movieTitle和movieYear的组合上建有索引，平均也需要3个盘块访问操作找出某个影星或某部电影的3个元组。如果没有对影星或影片分别建索引，则需要10次磁盘访问操作。

4. 每次对于给定的属性值，利用索引找到对应元组的磁盘块，需要对索引盘块做一次读操作。如果索引盘块需要更新（如在插入的情况下），那么还要一次磁盘操作，写回修改后的盘块。

5. 同样地，在进行插入操作时，需要一次磁盘操作读取将要插入新元组的盘块。另一次磁盘操作用于写回这个盘块。假定即使在没有索引的情况下，也不需要扫描整个的关系，就能找到可以继续添加元组的盘块。

图6-17给出了三种操作的代价：Q_1（给出影星的查询），Q_2(给出电影的查询)，和I（插入

操作）。如果不使用索引，对于Q_1和Q_2则必须扫描整个关系（代价为10），而插入仅需要访问一个有空闲空间的盘块并对它重写入一个新的元组。（代价为2，假定盘块不需要索引也可以访问到），以上的分析解释了“No Index”列。

动作	无索引	Star索引	Movie索引	全索引
Q_1	10	4	10	4
Q_2	10	10	4	4
I	2	4	4	6
	$2+8p_1+8p_2$	$4+6p_2$	$4+6p_1$	$6-2p_1-2p_2$

图6-17 三种操作在使用不同索引的情况下的操作代价

如果仅仅对影星建立索引，那么Q_2仍然需要扫描整个关系（代价为10）。然而，Q_1能通过访问一个索引块找到某个给定影星的3个元组，找到这些元组还需要另外3次盘块访问操作。插入操作I要求对某一索引盘和某一数据块既要进行一次读操作，又要进行一次写操作，总共是4次磁盘操作。

仅对电影建立索引的情况和仅对影星建立索引的情况类似。最后，如果对影星和电影都建立索引。那么回答Q_1和Q_2都需要4次磁盘操作。但是插入I就需要对两个索引盘块和一个数据盘块进行读写操作，总共需要6次磁盘操作。上面的分析解释了图6-17的最后一列。

图6-17的最后一行给出了操作的平均代价。假定执行Q_1的时间比例是p_1，执行Q_2的时间比
299 例是p_2，因此，I的时间比例是$1-p_1-p_2$。

随着p_1和p_2的取值不同，给出四种方案对于三种操作都可能产生最低的平均代价。例如，如果$p_1=p_2=0.1$，那么表达式$2+8p_1+8p_2$最小，所以这时可以不用索引。也就是如果插入操作是主要的，只有少量的查询，那么就不需要索引。另一种情况，如果$p_1=p_2=0.4$，那么公式$6-2p_1-2p_2$将会是最小的，这时可以选择对starName和(movieTitle,movieYear)组合都建立索引。直觉上，如果进行大量的查询，并且对指定电影的查询数目和对指定影星的查询数目大致相同时，则两个索引都需要。

如果$p_1=0.5$，$p_2=0.1$，那么仅有影星索引给出了最好平均性能值，因为$4+6p_2$取到最小值。同样地，$p_1=0.1$和$p_2=0.5$要求创建一个仅建立在电影上的索引。直观意义上，如果某类查询非常频繁，那么就仅仅创建有助于该查询的索引。 □

6.6.7 习题

* **习题6.6.1** 在本节中，仅对例子中使用的一个关系MovieStar给出了规范定义。请对其他四个关系也给出规范定义。

```
Movie(title, year, length, inColor, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

习题6.6.2 下面再次给出习题5.2.1中的不规范的数据库模式的定义，写出符合下面要求的声明语句。

```
Product(maker, model, type)
PC(model, speed, ram, hd, rd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

a) 关系Product的一个适当模式。

b) 关系PC的一个适当模式。

* c) 关系Laptop的一个适当模式。

d) 关系Printer的一个适当模式。

e) 修改(d)定义的模式，删掉属性color。 300

* f) 对(c)定义的模式进行修改，增加属性cd。如果某个手提电脑没有CD的话，令该属性的默认值为'none'。

习题6.6.3 下面是习题5.2.4中的不规范的模式定义

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

写出如下声明。

a) 关系Classes的一个适当模式。

b) 关系Ships的一个适当模式。

c) 关系Battles的一个适当模式。

d) 关系Outcomes的一个适当模式。

e) 修改关系Classes，删掉属性bore。

f) 修改(b)中的关系Ships，加入新的属性yard，它给出了该船的船坞。

! **习题6.6.4** 解释语句DROP R和语句DELETE FROM R的不同。

习题6.6.5 假定例子6.44讨论的StarsIn关系需要100个块而不是10个，但该例其他条件都不变。用包含p_1和p_2两项的公式，给出在使用索引和不使用索引的四种组合情况下，查询Q_1、Q_2及插入I的代价。

6.7 视图定义

用CREATE TABLE语句定义的关系实际存储在数据库中。也就是说，SQL系统以物理组织方式存储表。它们是持久的，即它能以某种方式存在，除非对它们明确地使用6.5节中的INSERT语句或其他修改语句进行更新，否则它将保持不变。

还有另一类称为视图（view）的SQL关系，它不以物理形式存在。而且，视图通过类似查询的表达方式定义。可以将视图当作物理存在进行查询，在某些情况下，视图也可以更新。 301

6.7.1 视图声明

最简单的视图定义形式由以下几项构成：

1) 保留字CREATE VIEW；

2) 视图的名字；

3) 保留字AS；

4) 一个查询Q。此查询为视图的定义。任何时候对视图查询时，SQL的表现好像是Q在此时立即执行，查询是作用在Q产生的关系上。

这样，简单视图定义的形式为

```
CREATE VIEW <view-name> AS <view-definition>;
```

例6.45 在关系Movie(title, year, length, inColor, studioName, producerC#)上创建一个视图，该视图包含Paramount制片厂出品的电影名字和年份。可以这

样定义视图

```
1)  CREATE VIEW ParamountMovie AS
2)     SELECT title, year
3)     FROM Movie
4)     WHERE studioName = 'Paramount';
```

首先，在第1行中给出了视图的名字ParamountMovie。在第2行中列出了视图的属性，即title和year。视图定义为第2到第4行的查询。 □

6.7.2 视图查询

关系ParamountMovie并不真正地包含元组。对ParamountMovie查询时，需要从基表Movie中取得适当的元组才能获得查询结果。在使用过程中，可能两次对ParamountMovie的查询结果都不同。原因是虽然没有改动视图ParamountMovie的定义，但是基表Movie在此期间改动了。

例6.46 可以把视图ParamountMovie当一个存储的表来使用，例如

```
SELECT title
FROM ParamountMovie
WHERE year = 1979;
```

302

> **关系、表和视图**
>
> SQL程序员喜欢把术语“关系”叫做“表”。原因是区分存储关系(称为表)和虚拟关系（称为视图）在实际中很重要。现在我们知道了表和视图的关系。以后将在表和视图使用的地方使用关系这个术语。当需要重点强调关系是实际存储的物理存在而不是一个视图时，就使用术语“基关系”或“基表”。
>
> 还有第三类关系，它既不是视图也不是固定存储的物理存在。这些关系是如某些子查询所创建的暂时结果。它们也可以称为关系。

利用视图ParamountMovie的定义把上面的查询转换成一个新的查询，新查询仅仅作用在基表Movie上。6.7.5节中将会介绍如何把对视图的查询转换成对基表的查询。这里的简单例子很容易推出对视图查询的含义。 注意到ParamountMovie和Movie在两方面不同：

1. 属性title和year是关系ParamountMovie产生的。

2. 条件studioName = 'Paramount' 是对ParamountMovie任意查询的WHERE子句的一部分。既然结果中只需要title属性，(1)显然能满足要求。对于(2)需要把条件studioName='Paramount' 加到查询中的WHERE子句中去。这样，在FROM子句中的ParamountMovie都可以用Movie来替代，并且不改变查询的意义。这样查询

```
SELECT title
FROM Movie
WHERE studioName = 'Paramount' AND year = 1979;
```

虽然是对基表的查询，但是和原来对视图ParamountMovie的查询有同样的效果。注意，这部分转换工作是由SQL系统完成，这里的推理过程是为了讲解视图的查询过程。□

303

例6.47 可以在查询中同时使用视图和基表，下面给出一个例子

```
SELECT DISTINCT starName
FROM ParamountMovie, StarsIn
WHERE title = movieTitle AND year = movieYear;
```

这个查询查找出所有Paramount电影的名字。注意这里出现的DISTINCT保留字保证每个影星只列出一次，无论他在多少部Paramount公司的电影里出现过。 □

例6.48 考虑一个用来定义视图的复杂查询。目的是生成一个具有电影名字和制片人的关系。定义视图的查询使用了两个关系。从关系Movie(title, year, length, inColor, studioName, producerC#)可以得到制片人的证书号，而关系MovieExec (name, address, cert#, netWorth)则可以通过证书号得到制片人的名字。整个语句可以写作：

```
CREATE VIEW MovieProd AS
    SELECT title, name
    FROM Movie, MovieExec
    WHERE producerC# = cert#;
```

可以把这个视图看做一个存储的关系来查询。例如，要找出*Gone With the Wind*，查询语句为

```
SELECT name
FROM MovieProd
WHERE title = 'Gone With the Wind';
```

对于任何视图查询，都可以将其当作等价关系上对基表的查询处理，例如，上述查询可以看做：

```
SELECT name
FROM Movie, MovieExec
WHERE producerC# = cert# AND title = 'Gone With the Wind';
```

□

6.7.3 重命名属性

有时想把视图的属性名改成其他的名字，而不使用定义视图时来自查询的属性名。这时，可以在CREATE VIEW语句中视图名后面用括号括起一个指定的属性列表来达到目的。例如，可以这样重写例子6.48的视图定义： 304

```
CREATE VIEW MovieProd(movieTitle, prodName) AS
    SELECT title, name
    FROM Movie, MovieExec
    WHERE producerC# = cert#;
```

视图还是原来那个，但是列名从title和name改成了movieTitle和prodName。

6.7.4 视图更新

在某些特定条件下可以对视图进行插入、删除和修改。这种说法听起来没什么意义，因为视图不像基表（存储关系）那样实际存在。那么插入元组到视图里面意味着什么？插入的元组放到哪里了？数据库系统怎么记住这个操作？

对于多数视图，答案是“不能这样做”。然而，对于那些“充分简单”的视图，有时也称为“*可更新视图*（updatable view）”，可以把对视图的更新转变成一个等价的对基表的更新，更新操作最终作用在基表上。如果某视图允许更新，SQL提供规范更新定义。SQL规则是复杂的，但大致上，SQL允许通过从某个关系*R*中（这个关系本身也可能是一个可更新的视图）选择（使用SELECT，而不是SELECT DISTINCT）某些属性来定义可更新视图，这类视图有两个重要的特点：

- WHERE子句中的子查询中不能使用关系*R*。

- 视图定义语句中，SELECT子句中的属性列表必须包括足够多的属性，以保证对视图进行插入时，将未列出的属性换成NULL值或其他的默认值。并将构成新元组插入到基表中，该新元组能产生插入的视图元组。

例6.49 假定对例子6.45中的ParamountMovie插入一个元组

```
INSERT INTO ParamountMovie
VALUES('Star Trek', 1979);
```

视图ParamountMovie基本满足SQL的更新条件，因为视图只是包含了基表的部分元组。基表为

```
Movie(title, year, length, inColor, studioName, producerC#)
```

惟一的问题是关系Movie的属性studioName不是视图的一个属性。对此，插入的元组将使用NULL值而不是'Paramount'作为studioName的属性值。这个元组因而不满足制片厂
305 Paramount的条件。

为了使得视图ParamountMovie可以更新。即使是知道制片厂肯定是Paramount，仍然在它的视图定义语句的SELECT子句中增加一个属性studioName。视图ParamountMovie新的定义为：

```
CREATE VIEW ParamountMovie AS
   SELECT studioName, title, year
   FROM Movie
   WHERE studioName = 'Paramount';
```

然后，对可更新的视图ParamountMovie插入一个元组：

```
INSERT INTO ParamountMovie
VALUES('Paramount', 'Star Trek', 1979);
```

为完成插入，创建了一个新的Movie元组，该元组满足定义在关系Movie上的视图，最后实际插入到基表Movie中。对于上面的特定插入，studioName的字段值是'Paramount'，title字段值是'Star Trek'，year字段值是1979。

其他三个没有在视图中出现的属性length、inColor和producerC#也存在于插入的元组中。然而，这里无法推断出它们的值。作为结果，新的Movie元组必须给这三个字段填上某个默认值：要么是NULL，要么是对属性声明过的其他的默认值。例如，如果对属性length声明某个默认值0，其他的两个使用NULL作为默认值。那么插入到Movie中的元组则为：

title	*year*	*length*	*inColor*	*studioName*	*producerC#*
'Star Trek'	1979	0	NULL	'Paramount'	NULL

□

也可以从可更新视图中删除元组。如同视图的插入一样，删除操作也是在底层的关系*R*上真正实现。对视图中元组的删除，实际是删除关系*R*中的元组。

例6.50 删掉可更新视图ParamountMovie中所有电影名中包含“Trek”字符串的电影。其删除语句是

```
DELETE FROM ParamountMovie
WHERE title LIKE '%Trek%';
```

这个删除转换成对Movie基表的一个等价删除。惟一差别是把定义视图ParamountMovie的条件加到WHERE子句的条件中。

```
DELETE FROM Movie
WHERE title LIKE '%Trek%' AND studioName = 'Paramount';
```

□ 306

同样地，对可更新视图的更新操作也是通过底层关系完成。对视图的更新结果是更新其底层关系中的元组，从而引起视图元组的更新。

为什么某些视图是不可更新的

考虑例6.48中的视图，它将电影名字和制片人的名字联系起来。根据SQL定义，这个视图是不可更新的，因为它的FROM子句中包含两个关系：Movie和MovieExec。如果想往视图中加入一个元组

```
('Greatest Show on Earth', 'Cecil B. DeMille')
```

就必须往关系Movie和MovieExec中加入元组。可以给属性length和address提供默认的属性值，但是对于表示DeMille未知证书号的producerC#和cert#两个视为相等属性却不好处理。即使对它们都使用NULL值，由于SQL不认为两个NULL值是相等的（见6.1.5节），所以无法使用NULL进行连接。这样，'Greatest Show on Earth'不会和视图MovieProd中的'CecilB.DeMille'连接。于是插入操作不成功。

例6.51 视图更新

```
UPDATE ParamountMovie
SET year = 1979
WHERE title = 'Star Trek the Movie';
```

转换成对基表的修改。

```
UPDATE Movie
SET year = 1979
WHERE title = 'Star Trek the Movie' AND
    studioName = 'Paramount';
```

□

最后一类对视图的更新是删除视图。无论视图是否可更新，这种更新操作总是可以进行。一个典型的DROP语句是

```
DROP VIEW ParamountMovie;
```

307

注意这条语句删除了视图的定义，因此不再可以对此视图进行查询或修改操作。但删除视图并不影响其底层关系的任何一个元组。相反地，

```
DROP TABLE Movie
```

不但使得表Movie从此消失，也使得视图ParamountMovie不可用。这是因为使用该视图的查询会间接引用一个不存在的关系。

6.7.5 涉及视图的查询解释

分析如何处理涉及视图的查询有助于更好地理解视图。在16.2节中介绍一般的查询处理的时候，这个问题还会继续深入讨论。

图6-18描述的是基本的思想。查询Q在关系代数中通过表达式树来表示。视图关系是表达式树的叶子。这里使用了两个叶子，视图V和W。要用基表解释查询Q，就需要找到视图V和W的定义。这些定义也表示成关系代数中的表达式树。

为了使查询只包括基表，将树Q中每个表示视图的叶子节点替换成定义该视图的树的根节点。在图6-18中，标记W和V的两个叶子替换成了视图的定义。最后的结果是该树成为一个基于基表的查询，并且它和原始查询等价。

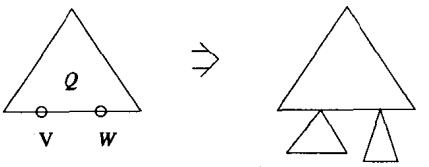


图6-18 用视图的定义来替换视图引用

例6.52 考虑例6.46中的视图定义和查询。其中视图ParamountMovie的定义是：

```
CREATE VIEW ParamountMovie AS
   SELECT title, year
   FROM Movie
   WHERE studioName = 'Paramount';
```

定义这个视图的查询表达式树如图6-19所示。

308 例6.46的查询为

```
SELECT title
FROM ParamountMovie
WHERE year = 1979;
```

它查找Paramount在1979年制作的电影。它的表达式树如图6-20所示。注意，该树的一个叶子表示视图ParamountMovie。

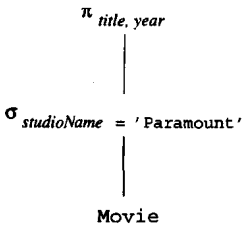


图6-19 视图ParamountMovie的表达式树

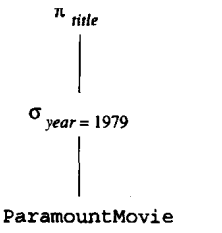


图6-20 查询的表达式树

因此图6-20中的叶子可以替换成6-19中的树来解释查询。替换后的树如图6-21所示。

图6-21的树是查询的一个可以接受的解释。但是它的表达过于复杂。SQL系统将会对这棵树进行转换，使之成为例6.46中的查询表达式树：

```
SELECT title
FROM Movie
WHERE studioName = 'Paramount' AND year = 1979;
```

例如，可以将投影操作$\pi_{title,year}$移到选择操作$\sigma_{year=1979}$上。原因是通常当选择操作不再改变表达式的结果时再做投影操作。这样，在一行上将有两个投影操作，首先是对title和year投影接着投影title。很明显第一个投影是一个多余的操作可以去掉。结果是两个投影操作合并成

309 一个对title的投影操作。

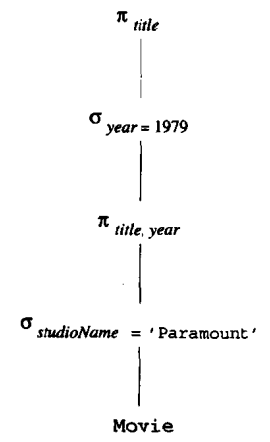


图6-21 在基表上表达查询

两个选择也可以合并。通常，可以对两个连续的选择操作的条件做AND，合并成一个选择。

合并选择操作后的表达式树如图6-22所示。 □

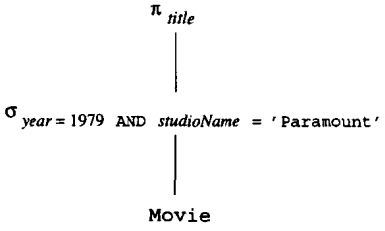


图6-22 在基表上简化查询

6.7.6 习题

习题6.7.1 根据下面的基表构造满足如下条件的视图。 310

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

* a) 视图RichExec给出了所有资产在$10 000 000以上的制片人的名字、地址、证书号和资产。

b) 视图StudioPres给出了既是制片厂经理（Studio president），又是制片人（Movie Executive）的人的名字、地址、证书号。

c) 视图ExecutiveStar给出既是制片人，又是影星的那些人的名字、地址、性别、生日、证书号和资产总值。

习题6.7.2 习题6.7.1中的视图哪些可更新?

习题6.7.3 使用一个或多个习题6.7.1中定义的视图，写出下面的查询，但不要使用基表。

a) 找出既是影星又是制片人的女性。

* b) 找出是制片厂经理，同时资产至少有$10 000 000的制片人。

! c) 找出是影星，同时资产至少有$50 000 000的制片厂经理。

*! **习题6.7.4** 对于例子6.48的视图和查询：

a) 给出视图MovieProd的表达式树。

b) 给出该例中查询的表达式树。

c) 使用基表从(a)和(b)的答案中来创建表达式，以表示该查询。

d) 解释如何改变(c)的表达式，使之和例6.48中的解决方式等价。

! **习题6.7.5** 用关系代数表达式来表示习题6.7.3中的查询和视图，替换查询表达式中的视图，同时尽可能地简化结果表达式。只使用基表，用SQL写出对应最后结果表达式的查询语句。

习题6.7.6 使用习题5.2.4中的基表

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
```

311

a) 定义一个视图，它包括所有英国船只的类（class）、类别(type)、火炮数量、口径、排水量和下水年份。

b) 利用(a)定义的视图写一个查询，找出在1919年下水的英国战舰火炮数量以及排水量。

! c) 用关系代数的形式表达查询(b)和视图(a)，试着替换表达式中的视图并且最大程度地化简表达式。

! d) 仅使用基表Classes和Ships写出一个对应(c)中表达式的SQL语句。

6.8 小结

- SQL: SQL语言是关系数据库上使用的主要查询语言。目前最新标准是SQL-99或SQL3。商用数据库管理系统通常和SQL标准略有出入。
- Select-From-Where查询：SQL查询最常用的形式是select-from-where。它允许使用几个关

系的积（在FROM子句中），对结果元组施加过滤条件（在WHERE子句中），并产生需要的字段值（SELECT子句）。

- 子查询：select-from-where查询能在其他查询中的WHERE子句或FROM子句中作为子查询使用。操作符EXISTS、IN、ALL和ANY都可以作用在WHERE子句中子查询形成的关系上，并形成布尔表达式。
- 关系的集合操作符：可以通过分别使用保留字UNION、INTERSECT和EXCEPT连接关系或者产生关系的查询，达到实现关系的并、交和差的目的。
- 连接表达式：SQL提供的如NATURAL JOIN的操作符，作用在关系上。其本身可以看做是一个查询或是在FROM子句中定义的一个新关系。
- 空值：SQL在元组的字段值没有指定具体值的时候，提供一个特殊的值NULL。NULL的逻辑和算术运算规则都比较特殊。任何值和NULL值比较的结果都是布尔值UNKNOWN，即使该值也是NULL值。UNKNOWN值也能在布尔运算中出现，但把它看做处于TRUE和
312 FALSE之间的一个值。
- 外连接：SQL提供一个OUTER JION操作符连接关系。连接结果中包括了来自连接关系的悬浮元组。在结果关系中悬浮元组被填上NULL值。
- 关系的包（bag）模型：SQL实际上把关系看做装满元组的包而不是元组的集合。可以使用DISTINCT保留字来消除元组重复。而保留字ALL允许在某些不认为关系是包的情况下保留重复。
- 聚集：关系中某列的值可以通过使用保留字SUM、AVG（平均值）、MIN、MAX和COUNT进行统计（聚集）。在进行聚集前元组可以通过GROUP BY进行分组。利用保留字HAVING可以过滤掉某些分组。
- 更新语句：SQL允许改变关系中元组的值。可以在SQL语句中使用INSERT（插入新元组），DELETE（删除元组）和UPDATE（改变某些存在的元组）来达到目的。
- 数据定义：SQL提供语句对数据库中的一些对象的模式进行定义。语句CREATE TABLE允许为存储的关系（称为表）定义模式，定义时指定关系的属性、数据类型和默认值。
- 模式修改：可以使用ALTER语句改变数据库模式的某些方面，包括增加或去掉关系模式的属性以及改变属性的默认值。还可以使用DROP语句完全删除关系和其他的对象模式。
- 索引：索引不是SQL标准的一部分，商业SQL系统允许对关系的属性声明索引。当查询或更新操作使用了被索引的属性时，能加快操作。
- 视图：视图定义了如何从存储在数据库中的表创建新的关系（视图）。可以像被存储的关系那样来查询视图。SQL系统修改对视图的查询，就可以修改对定义视图的基表的查询。

6.9 参考文献

SQL2和SQL-99标准都是可以通过匿名FTP从网上获得。最主要的地址是ftp://jerry.ece.umassd.edu/isowg3，它的镜像网站有ftp://math0.math.ecu.
313 edu/isowg3和ftp://tiu.ac.jp/iso/wg3。它们都使用子目录db1/BASEdocs来存放。在本书印刷时，上面列出的网站有的还不接受FTP访问。我们设法在本书的网站上提供信息使读者能了解到最新的内容。

有几本书给出了SQL编程的一些细节内容。在此推荐列出[2]，[4]和[6]。[5]中含有关于最近SQL-99标准的较早介绍。

[3]最先给出了SQL的定义。它是最早实现的关系数据库原型Systme R（参阅[1]）的一部分。

1. Astrahan, M. M. et al., "System R: a relational approach to data management," *ACM Transactions on Database Systems* **1**:2, pp. 97–137, 1976.

2. Celko, J., *SQL for Smarties*, Morgan-Kaufmann, San Francisco, 1999.

3. Chamberlin, D. D., et al., "SEQUEL 2: a unified approach to data definition, manipulation, and control," *IBM Journal of Research and Development* **20**:6, pp. 560–575, 1976.

4. Date, C. J. and H. Darwen, *A Guide to the SQL Standard*, Addison-Wesley, Reading, MA, 1997.

5. Gulutzan, P. and T. Pelzer, *SQL-99 Complete, Really*, R&D Books, Lawrence, KA, 1999.

6. Melton, J. and A. R. Simon, *Understanding the New SQL: A Complete Guide*, Morgan-Kaufmann, San Francisco, 1993.

314

第7章 约束和触发器

本章讨论在SQL中创建“主动”（active）元素的相关内容。主动元素是一个表达式或语句。该表达式或语句只需编写一次，存储在数据库中，然后在适当的时间执行。主动元素的执行可以是由于某个特定事件引发，如对关系插入元组，或者是当修改数据库的值引起某个逻辑值为真。

应用程序编写者面临的一个问题是，当更新数据库时，新的信息有可能是错误的。例如，手工录入数据时常常有抄写或印刷错误。能保证数据库拒绝接受不适当元组的最直接方法，是对应用程序中的每个插入、删除或更新命令都编写与其结合的检查，以保证其正确性。但是，正确性需求常常是复杂的，也是有重复的，这样应用程序必须在每次修改后都做相同的测试。

幸好SQL提供了各种技术把*完整性约束*（integrity constraint）作为数据库模式的一部分。本章讨论其基本方法。首先是键约束，将一个或一组属性声明为一个关系的键。然后考虑引用完整性。引用完整性也被称为“外键约束”（foreign-key constraint），是指一个关系中的一个属性或一组属性的值（如Studio中的presC#）必须在另一个关系的一个或一组属性的值中出现（如MovieExec中的cert#）。

然后，将属性上、元组上和关系上的约束作为整体考虑。关系之间的约束称为“断言”。最后，讨论“触发器”（trigger），触发器是主动元素，它在某个特定事件发生时被调用，如对一个特定关系的插入事件。

7.1 键和外键

数据库中最重要的约束，是声明一个或一组属性形成关系的键。如果一组属性S是关系R的键，则R的任何两个元组的S中至少有一个属性值不同。注意，该规则也可应用于重复元组约束，即，如果R有一个键，则R不能有重复。

与很多其他约束一样，键约束在SQL的CREATE TABLE命令中声明。有两种相似的声明键的方法：用保留字PRIMARY KEY或保留字UNIQUE。一个关系中只能有一个主键，但是可以有多个“惟一性”声明。

在与某个引用完整性约束连接中，SQL也使用“键”这一术语。判定一个关系中出现的值，也必须在另一个关系的主键中出现的约束称做“外键约束”。7.1.4节将关注外键约束。

7.1.1 主键声明

关系中只能有一个主键。CREATE TABLE中有两种声明主键的方法。

1. 当属性被列入关系模式中时，声明其是键。

2. 在模式声明的项目表中增加表项（目前仅能有属性项）声明一个或一组属性是键。

方法（1）是将PRIMARY KEY保留字加在属性类型之后。方法（2）是在属性列表中引入一个新元素，该元素包含保留字PRIMARY KEY和用圆括号括起的形成该键的属性或属性组列表。注意，键是由多个属性构成时，使用此方法。

关系R的属性集S被声明为键后有两点影响：

1. R中的两个元组不能有完全相同的S属性值。任何违反了该规则的插入或更新操作都将引

起DBMS拒绝执行。

2. S中的属性不允许是NULL值。

例7.1 考虑例6.39中关系MovieStar的模式。该关系的主键是name。于是，在name声明行上加入此事实。图7-1给出了图6-16的修改版。

316

```
1)  CREATE TABLE MovieStar (
2)     name CHAR(30) PRIMARY KEY,
3)     address VARCHAR(255),
4)     gender CHAR(1),
5)     birthdate DATE
    );
```

图7-1 name主键声明

另外，也可以使用独立的主键定义声明。在图6-16的第（5）行后面增加主键声明，而不必在第（2）行中声明。结果模式声明如图7-2所示。 □

```
1)  CREATE TABLE MovieStar (
2)     name CHAR(30),
3)     address VARCHAR(255),
4)     gender CHAR(1),
5)     birthdate DATE,
6)     PRIMARY KEY (name)
    );
```

图7-2 独立的主键声明

例7.1中，主键是单个属性时，无论是用图7-1的形式，还是用图7-2的形式都可以接受。可是，当主键是由多个属性组成时，必须要使用图7-2方法。例如，如果声明Movie关系模式，它的键是一对属性title和year，则应该在所有属性列表后增加一行：

```
PRIMARY KEY (title, year)
```

7.1.2 用UNIQUE声明键

另一种声明键的方法是使用保留字UNIQUE。该保留字可以出现在PRIMARY KEY出现的位置，或是跟在属性类型的后面，或是在CREATE TABLE语句中作为一个独立项出现。

317

UNIQUE声明的意思与PRIMARY KEY声明的意思几乎一样，不同点是：

1. 一个表中可以有多个UNIQUE声明，但是只能有一个PRIMARY KEY声明。

2. 声明为PRIMARY KEY的属性（组）不允许取空值（NULL），但是UNIQUE声明允许取空值。而且，对声明为UNIQUE的属性取空值时，可以会违反两个元组中声明为UNIQUE的属性值不能完全相同的规则。事实上，两个元组中，声明为UNIQUE的多个属性都取空值也允许。

DBMS的实现可以选择不同的规则。例如，数据库厂商可以总是为声明为PRIMARY KEY的属性建立索引（即使是该PRIMARY KEY是由多个属性构成时也如此），而其他属性则需要用户明确调用索引。另外，也可以让表在它的PRIMARY KEY上总保持有序。

例7.2 图7-1中的第（2）行可以被改写为

```
2)     name CHAR(30) UNIQUE,
```

如果两个电影明星不能有相同的地址，也可以将第（3）改为：

```
3)     address VARCHAR(255) UNIQUE,
```

类似地，可以将图7-2的第（6）行改为：

```
6)    UNIQUE (name)
```

□

7.1.3 强制键约束

第6.6.5 节关于索引的讨论中已知，虽然SQL标准没有规定，但是每个SQL的实现都将创建索引作为数据库模式定义的一部分。为了支持对主键的查询，系统通常在主键上建立索引。而且，用户也需要为声明为UNIQUE的属性建立索引。

这样，当查询中WHERE短语包含键值条件时，如例7.1中MovieStar关系上的name = 'Audrey Hepburn'，就可以不需要查询关系中的所有元组，很快找到与条件匹配的元组。

很多SQL的索引创建语句中提供保留字UNIQUE，也就是说，在为该属性建立索引的同时，该属性也被声明为键。例如有语句：

```
CREATE UNIQUE INDEX YearIndex ON Movie(year);
```

318

该语句与6.6.5节中索引创建语句有相同的效果，但是该语句同时也对Movie关系的year属性增加了惟一性约束（此处假定并不合理）。

现在思考一下SQL系统是如何实现强制键约束的。原则上，每当数据库被改变时都要做检查。但是，事实上，关系R只有在插入或更新时才可能会破坏键约束，R上的删除不会破坏键约束。因此，SQL系统的实际做法是仅当对关系作插入和更新时才检查键约束。

如果SQL系统有效地强制键约束，那么把属性（组）上的索引声明为键是很重要的。因为如果索引有效，则无论何时对关系插入元组，或更新元组中键属性值，都可以使用索引检查元组是否已有相同的键属性值。如果有，系统将阻止这种更新发生。

如果在键属性（组）上没有索引，仍然可以强制键约束。另外，通过对键排序也可有助于查询。在没有辅助的查找中，系统必须检查整个关系，查找与给定值匹配的元组。这个过程是极其耗时的，以致大型关系数据库的更新实际上不可能。

7.1.4 外键约束声明

数据库模式上第二种重要约束是要求某个属性的值必须有意义。比如，关系Studio中presC#属性的意思是要指定某个制片人。这个隐含的“引用完整性”约束是说，如果Studio元组在presC#属性上有特定数值C，那么就声明C是一真实的制片人。用数据库的术语来说，“真实”制片人的意思是认为他是在MovieExec关系中提到的一个制片人。因此，必定有一些MovieExec的元组，它的cert#属性值是C。

在SQL中可以将关系的一个属性（或属性组）声明为外键（foreign key），该外键引用另一个关系（也可以是同一个关系）的属性（组）。外键声明隐含着如下两层意思：

1. 被引用的另一个关系的属性在它所在的关系中，必须被声明为UNIQUE或PRIMARY KEY。否则，就不能由外键声明。

2. 在第一个关系中出现的外键值，也必须在被引用关系的某些元组的属性中出现。更精确地说，若令外键F引用某个关系的属性集G，并假定第一个关系中的元组t在F属性（组）上的值非空，t元组的F属性值记为$t[F]$。于是，在被引用的关系上必定有元组s，s的G属性（组）值与$t[F]$值相等。也就是说，$s[G] = t[F]$。 319

对于主键，有两种方法声明外键：

a）如果外键是单个属性，则可以在此属性的名字和类型之后，声明其“引用”的属性

（被引用的属性必须有主键或惟一性声明）。格式如下：

```
REFERENCES <表名>  (<属性名>)
```

b）若外键是由一组属性构成，则在CREATE TABLE语句的属性定义之后，用单独的声明语句声明，给出外键引用的表和属性（这些属性必须是键）。声明的格式为：

```
FOREIGN KEY (<属性名列表>) REFERENCES <表名>  (<属性名列表>)
```

例7.3 假设要说明关系

```
Studio(name, address, presC#)
```

其主键是 name，外键是presC#，被引用的属性是关系MovieExec的cert#，MovieExec关系如下：

```
MovieExec(name, address, cert#, netWorth)
```

外键presC# 可以直接引用cert#：

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT REFERENCES MovieExec(cert#)
);
```

另外也可以如下使用单独的外键声明语句完成：

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT,
    FOREIGN KEY (presC#) REFERENCES MovieExec(cert#)
);
```

注意，被引用的属性MovieExec中的cert#，必须是MovieExec的键。无论上面使用哪种形式的声明，其意义都是说无论何时，Studio元组中presC#的属性值，都必须也在MovieExec元组的cert#分量中出现。例外情况是当Studio的元组中presC#取空值时，并不要求cert#的值也是NULL（事实上，cert#是主键，因此它永远不会有NULL
320 值）。 □

7.1.5 维护引用完整性

从外键的声明中已知，外键声明意味着当其属性值非空时，就必须保证这些值也应在被引用关系的相应属性中出现。但是，数据库被更新时，如何能保证其约束？数据库系统实现者可以选择如下三种不同实现方法。

缺省原则：拒绝无效更新

SQL有缺省原则，即当任何更新违反引用完整性约束时，系统拒绝执行此更新。例如，例7.3中要求关系Studio中的presC#值也必须是MovieExec中cert#属性值，于是如下的动作都将被系统拒绝执行：

1. 对Studio插入一新元组，其presC#值非空，但是它不是MovieExec关系中任何元组的cert#值。插入动作被系统拒绝，元组插入不成功。

2. 更新Studio关系元组的presC#属性为非空值，但是该值不是MovieExec关系中任何元组的cert#值，该更新操作被拒绝，元组没有被更新。

3. 删除MovieExec元组，该元组的cert#值是某个（些）Studio元组的presC#值。该删除动作被拒绝，元组仍留在MoivExec关系中不变。

4. 更新MoivExec元组的cert#值。没有更新前的cert#值是Studio关系中某个元组的presC#值。系统再次拒绝这个更新动作，cert#值保持不变。

级联原则

另外有一种处理MovieExec关系的删除和更新（即上述3和4两种操作）操作方法。这种方法称做*级联原则*（cascade policy）。直观上看，引用属性(组)的改变被仿造到外键上。

在级联原则下，当对制片厂经理删除MovieExec元组时，为了维护引用完整性，也同时删除Studio中的相应元组。修改也类似处理，当将cert#的值从c_1改为c_2时，若存在有Studio元组的presC#值是c_1，则系统也把presC#值从c_1改为c_2。 321

置空值原则

处理该类问题的另一种方法是，把与被删除或更新的制片厂经理presC#值设置为空值（NULL），该原则称为*置空*（set-null）原则。

通过外键声明，可以为删除和更新独立地选择上述各种方法。声明的方法是在ON DELETE或ON UPDATE短语后面跟着SET NULL或CASCADE选项。

例7.4 修改7.3例中对Studio(name, address, presC#)的删除声明，以及对MovieExec(name, address, cert#, networth)的更新声明。

图7-3使用例7.3的CREATE TABLE语句，并用ON DELETE和ON UPDATE对其扩展。第（5）行声明，删除MovieExec元组的同时，也从Studio中将该制片人是经理的presC#值改为NULL。第（6）行声明，修改MovieExec的cert#值时，Studio中具有该值的presC#值也被同时修改。

```
1)  CREATE TABLE Studio (
2)     name CHAR(30) PRIMARY KEY,
3)     address VARCHAR(255),
4)     presC# INT REFERENCES MovieExec(cert#)
5)         ON DELETE SET NULL
6)         ON UPDATE CASCADE
    );
```

图7-3 选择不同原则保持引用完整性

注意，例子中置空原则使删除具有更多的含义，而级联原则更适合于更新。例如，制片厂经理退休时，制片厂仍然存在，其经理属性值在经理没有确定前要取空值。可是，制片厂经理经营证书号的修改更像是办事员的改变。人员继续存在，而且将是该制片厂的经理，因此Studio中presC#属性值也应该随着改变。 □ 322

7.1.6 延迟约束检查

假定在例7.3中，Studio的presC#是引用MovieExec中的cert#外键。Bill Clinton决定，在他卸任美国总统后，建立一个电影制片厂，称做Redlight制片厂，当然他就是该厂的经理。但是，执行如下插入语句会有些麻烦。

```
INSERT INTO Studio
VALUES('Redlight', 'New York', 23456);
```

原因是，MovieExec没有证书号码为23456的元组（假定23456是最新为Bill Clinton颁发的证书号），显然这违反了外键约束。

悬挂元组和更新原则

外键值在引用关系中不出现的元组称为悬挂元组（dangling tuple），在连接运算中不满足条件的元组也称为“悬挂”。这两个概念是紧密相关的。如果元组的外键值在引用关系中不出现，则该元组就不会与引用关系满足连接条件。

悬挂元组是那些违反引用完整性约束的外键元组。

- 引用关系的删除和修改缺省原则是说，当且仅当引用关系中产生了一个或多个悬挂元组时，该动作被阻止。
- 级联原则是删除或修改所有新产生的悬挂元组（分别依赖于对引用关系是更新还是删除）。
- 置空原则是把每个悬挂元组中外键值设为NULL。

解决此问题的一种方法是先插入Redlight元组，但是经理证书的值为空。如：

```
INSERT INTO Studio(name, address)
VALUES('Redlight', 'New York');
```

这种改变就避免违反约束。因为Redlight元组插入时，presC#的值是NULL，系统对空的外键
323 不检查其引用的列是否有已存在的值。可是，在系统执行如下修改语句前，还必须要把带有证书号的Bill Clinton元组插入MovieExec关系。

```
UPDATE Studio
SET presC# = 23456
WHERE name = 'Redlight';
```

如果不首先插入MovieExec元组，该修改语句同样也违反了外键约束。

当然，在这种情形，把Redlight元组插入Studio之前，先把Bill Clinton及他的证书元组插入MovieExec，将防止违反外键约束。但是，当循环约束（circular constraint）发生时，如上仔细安排的数据库更新顺序也不能解决其违反约束问题。

例7.5 如果电影制片人被约束为是制片厂经理，则要声明cert#是引用Studio（presC#）的外键。于是presC#必须被声明为UNIQUE。该声明意味着，两个制片厂的经理不能同时是同一个人。

现在，不可能对制片厂关系插入新元组。因为不能对Studio插入其presC#值是新值的元组，这样做将违反presC#是引用MovieExec（cert#）的外键约束。也不能对MovieExec插入其cert#值是新值的元组，因为这将违反cert#是引用Studio（presC#）的外键约束。 □

7.5例中的问题可以解决，但是将涉及还没有介绍的多个SQL的元素。

1. 首先，需要有将多个SQL语句组成一个单元的能力（两个插入——一个插入Studio，另一个插入MovieExec），这个单元称做“事务”。在8.6节中，事务被看做是不可见的工作单元。

2. 另外，必须要有一种方法通知SQL语句，当事务没有结束（针对事务的术语是“提交”）前，不要检查其约束。

对于第1点，可以先认为是成立的，但是要能处理第2点，还要有如下几点知识：

a) 任何约束——键，外键，或其他将在本章中可见到的约束——都可声明为DEFERRABLE或NOT DEFERRABLE。后者是缺省值，也即意味着每次数据库更新发生时，如果需要检查，系统立即检查约束。可是，如果约束被声明为DEFERRABLE，则是告诉系统，其约束检查可以

推迟到事务结束时进行。 324

b) 如果约束检查可以推迟，则需要声明INITIALLY DEFERRED 或者INITIALLY IMMEDIATE。前者意味着除非告诉系统停止约束推迟，检查将推迟到当前事务结束时进行。

例7.6 图7-4给出了将Studio的外键约束检查修改为推迟到事务结束时进行的模式定义。该定义中，将presC#声明为UNIQUE，是为便于其他关系外键约束引用。

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT UNIQUE
        REFERENCES MovieExec(cert#)
        DEFERRABLE INITIALLY DEFERRED
);
```

图7-4 修改presC#为UNIQUE，并推迟外键约束检查

如果对例7.5中提到的，假设MovieExec（cert#）是引用Studio（presC#）的外键约束给出类似的声明，则能够写一事务，分别为每个关系插入一元组。但是这两个外键的约束检查将推迟到两个插入动作完成之后进行。这样一来，当插入新的制片厂和它的新经理，并且这两个元组具有相同的证书号时，没有违反这两个外键约束中的任何一个。 □

对于推迟约束检查，有两点要记住：

- 任何类型的约束都可以命名。7.3.1节中将讨论如何做这件事。
- 如果一个约束有名字，比如MyConstraint，就可以用如下SQL语句将该约束从立即检查改为推迟检查。

```
SET CONSTRAINT MyConstraint DEFERRED;
```

同样，也可以把上面的DEFERRED检查改为IMMEDIATE。 325

7.1.7 习题

* **习题7.1.1** 5.1节的电影数据库例子中，对所有关系都定义了键，

```
Movie(title, year, length, inColor, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

修改习题6.6.1的模式定义，包括每个关系的键声明。注意关系StarsIn中的键是由它的所有属性组成。

习题7.1.2 声明习题7.1.1中电影数据库的如下引用完整性约束。

* a) 电影的制片人必须是MovieExec中的某个制片人。任何对MovieExec的更新，若违反约束则拒绝该操作。

b) 重复a)，但是当违反约束时，Movie中的producerC#置为NULL。

c) 重复a)，但是当违反约束时，Movie中违反约束的元组被删除或修改。

d) 出现在StarsIn中的电影，也必须出现在Movie中。当违反约束时，拒绝其更新。

e) 在StarsIn中出现的电影，也必须在MovieStar中出现。当违反约束时，删除违规的元组。

*! **习题7.1.3** 在关系Movie中的每部电影都至少在关系StarsIn中出现一次，这样的约束

能否用外键约束声明？并说出其理由。

习题7.1.4 对习题5.2.1中的PC数据库，

```
Product(maker, model, type)
PC(model, speed, ram, hd, rd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

给出适合每个关系的键。并且修改习题6.6.2的SQL模式定义，包括这些键的声明。

326 **习题7.1.5** 对习题5.2.4中的战船数据库

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

给出适合每个关系的键，并修改习题6.6.3的SQL模式定义，包括这些键的声明。

习题7.1.6 对习题7.1.5中的战船数据库，写出其引用完整性约束。根据习题7.1.5中有关键的假定，通过设置引用属性值为NULL处理所有违反约束之处。

* a) 在Ships中提到的每一类，也必须在Classes中出现。

b) 在OutComes中提到的每一次战役，也必须在Battles中出现。

c) 在OutComes 中提到的每艘战船，也必须在Ships中出现。

7.2 属性和元组上的约束

前一节中的键约束是迫使关系中某个属性的值必须具有惟一性，外键约束是迫使两个关系的属性之间有引用完整性。现在，介绍第三种约束：限制某些属性值的约束。这类约束可以用如下任一种方式表达：

1. 在关系模式定义中给出属性上的约束。

2. 在整个元组上的约束。该约束是关系模式的一部分，不与任何属性相关。

7.2.1节中将介绍属性上的简单型约束：属性值不可是NULL的约束。7.2.2节中给出类型(1)的基本形式，基于属性的CHECK约束（attribute-based CHECK constraint）。第二种类型——基于元组的约束将在7.2.3中给出。

更一般性的约束将在7.4节中见到。这种约束将用于约束整个关系或多个关系上的改变，

327 以及在单个属性或元组值上的约束。

7.2.1 非空值约束

与属性相连的简单约束是NOT NULL，其作用是不允许该属性取空值。约束声明方法是在CREATE TABLE 语句的属性声明之后用保留字NOT NULL声明。

例7.7 假定关系Studio中需要presC#不取空值，可以将图7-3中的第(4)行改为：

```
4)    presC# INT REFERENCES MovieExec(cert#) NOT NULL
```

这个假设会造成几种结果。例如：

- 对Studio关系插入元组时，不能只给出名字和地址，因为此时其presC#的值是NULL。
- 图7-3中第（5）行的置空值原则此处不能用。因为该原则通知系统，当外键违反约束时要将presC#值置空。

□

7.2.2 基于属性的CHECK约束

更复杂的约束是通过在属性声明之后附加保留字CHECK给出。CHECK保留字之后是用圆括号给出该属性应满足的条件。实际中，基于属性的CHECK约束是值上的简单约束。如合法值的排列或算术不等式。可是，原则上CHECK条件可以是任何在SQL语句的WHERE子句中允许的描述。条件可以通过表达式中属性名字引用被约束的属性。但是，如果条件要引用其他关系，或是其他属性，则该关系必须是子查询的FROM子句中出现的关系（即使该关系是被检查属性所在的关系）。

基于属性的CHECK约束是在元组为该属性获得新值时被执行。新值可能是由修改元组的语句给出，或者是插入元组的一部分。如果新值违反约束，则该更新被拒绝。例7.9将解释，如果数据库更新并不改变与约束相连的属性值，则基于属性的CHECK检查将不被执行，因为这样做将违反约束。

例7.8 设证书号必须至少有6位数字。图7-3中关系 328

```
Studio(name, address, presC#)
```

模式定义的第(4)行可以改为：

```
4)    presC# INT REFERENCES MovieExec(cert#)
                CHECK (presC# >= 100000)
```

另外一个例子，图6-16中关系

```
MovieStar(name, address, gender, birthdate)
```

的gender属性的数据类型被声明为CHAR(1)——也就是说，是一个单字符。可是，该字符的期望值只能是'F'和'M'。于是图6-16中第4行改为：

```
4) gender CHAR(1) CHECK (gender IN ('F', 'M')),
```

上面给定的条件是使用显示的两个值，声明任何gender的值必须是此集合中的值。 □

CHECK检查的条件中允许使用本关系中的其他属性或元组，甚至可以使用其他关系。但是如要这样做，就必须在条件中给出子查询。如已经提到的，条件可以是select-from-where型SQL语句中WHERE子句里的条件。应该明白约束检查仅仅只与要检查的属性相关，与约束中提到的每一个关系或属性无关。这样，如果某些与被检查不同的成分被改变时，复杂的条件可能取“假”值。

例7.9 用基于属性的CHECK约束模拟引用完整性约束。下面是模拟

```
Studio(name, address, presC#)
```

关系中presC#值必须在关系

```
MovieExec(name, address, cert#, netWorth)
```

的cert#之中出现的约束。假定图7-3第（4）行改为：

```
4)    presC# INT CHECK
          (presC# IN (SELECT cert# FROM MovieExec))
```

该语句是一个合法的基于属性的CHECK约束，它的作用如下： 329

- 如果对Studio插入新元组，新元组的presC#值不是任何电影制片人的证书号时，插入

被拒绝。

- 如果更改Studio元组的presC#值，新值不是电影制片人的cert#时，更改被拒绝。
- 可是，如果改变MovieExec，删除制片厂经理元组，此变化对上面的CHECK约束不可见。于是，删除动作被执行。即使这样一来违反了presC#上的CHECK约束。

在7.4.1节中，将见到如何使用更强有力的约束形式来正确表达上述条件。□

7.2.3 基于元组的CHECK约束

为了对单个表*R*的元组声明约束，在用CREATE TABLE 语句定义表时，可以在属性列表、键或外键声明上附加CHECK保留字，其约束条件用括号括起。括号中的条件可以是WHERE子句中出现的任何表达式。表达式被解释为表*R*元组上的条件，*R*的属性可以出现在该表达式中。这一点与基于属性的CHECK约束不同，在基于属性的CHECK约束中，子查询中的条件可以引用其他关系或同一关系*R*的其他元组。

每次向*R*插入元组时，以及当*R*的元组被修改时，都要基于元组的CHECK约束检查这个新元组或被更新的元组。如果该元组的约束条件计算结果是假，则表明违反约束，插入或更新被拒绝。可是，如果条件的子查询中引用某个关系（即使是关系*R*本身），而那个关系的改变将使关系*R*的某些元组对条件的计算结果是假，CHECK却不能阻止这样的事情发生。也就是说，类似基于属性的CHECK约束，基于元组的CHECK约束对其他关系不可见。

虽然基于元组的检查允许一些非常复杂的条件，但最好还是将复杂检查留给SQL的“断言”完成。关于“断言”将在7.4.1中讨论。正如上面已讨论的那样，把复杂检查留给“断言”的理由是，在某些条件中，基于元组的约束可以在不知晓的情况下违反。可是，如果基于元组的检查只涉及对元组的属性检查，而不涉及子查询，那么这类约束总可以保持。下面是涉及元组中多个属性的基于元组的CHECK约束的简单例子。

例7.10 对于例6.39的MovieStar表模式声明，图7-5重复了那个CREATE TABLE语句，另外又增加了主键约束和另一个约束声明，这些约束是将要检查的“一致性条件”之一。约束
330 的意思是，如果影星的性别是男性，则他的名字不能以’Ms.’开头。

```
1)  CREATE TABLE MovieStar (
2)     name CHAR(30) PRIMARY KEY,
3)     address VARCHAR(255),
4)     gender CHAR(1),
5)     birthdate DATE,
6)     CHECK (gender = 'F' OR name NOT LIKE 'Ms.%')
    );
```

图7-5 MovieStar的约束声明

在第（2）行，name被声明为关系的主键。第（6）行声明约束。对于每一个女影星和名字开头不是’Ms.’的影星元组，该约束取真值。使条件不为真的元组值名字开头是’Ms.’的男影星。这些正是应该从MovieStar中去除的元组。□

正确地书写约束

很多约束与例7.10相同，都是为了使元组满足两个或更多个条件。紧跟CHECK之后的表达式是每个条件否定的OR运算，或肯定的OR运算。该变换是“摩根定律”（DeMorgan's laws）之一：AND项的否定是各项否定的OR。因此，例7.10中第一个条件是声明影星是男性，使用gender=’F’作为合适的否定（虽然gender<>’M’应该是更

一般的表述否定的方法）。第二个条件是说，name开头必须是'Ms.'，使用NOT LIKE比较运算。该比较运算本身包含否定成分，在SQL中可以写成name LIKE 'Ms.%'。

7.2.4 习题

习题7.2.1 对关系

```
Movie(title, year, length, inColor, studioName, producerC#)
```

写出如下关于属性的约束。

* a) 年份不能是1895年以前。

b) 长度不能少于60也不能多于250。

* c) 制片厂的名字只能是Disney、FOX、MGM或者Paramount。 331

约束检查的局限：缺陷或特征？

人们可能奇怪，如果基于属性和基于元组的约束引用了其他关系或同一个关系的其他元组时，为什么能允许它被违反。理由是，这样的约束实现可以比更通用的约束（如断言，7.4.1节将讨论）的实现更有效。对于基于属性或基于元组的约束，仅仅需要计算被约束关系新插入或修改的元组的约束。而另一方面，断言则必须在其所提及的任一个关系每次被改变时都要做计算。仔细的数据库设计者，仅仅当这类约束不可能被违反时才使用基于属性和基于元组的约束。否则，将使用其他机制，如断言或触发器等。

习题7.2.2 对习题5.2.1中的关系模式例子写如下关于属性的约束。习题5.2.1的例子模式是：

```
Product(maker, model, type)
PC(model, speed, ram, hd, rd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

a) 手提电脑的速度至少是800。

b) 活动磁盘要么是32x或40x的CD，要么是12x或16x的DVD。

c) 打印机的类型只能是激光（laser）、喷墨（ink-jet）和点阵（bubble）。

d) 产品类型只能是PC、手提电脑和打印机。

! e) 产品模型也必须是PC、手提电脑和打印机。

习题7.2.3 例7.13对图7-7的基于元组的CHECK约束只完成了图7-6断言的部分工作。写出全部关于MovieExec的CHECK约束。

习题7.2.4 对给出的电影例子，写出如下基于元组的CHECK约束。

```
Movie(title, year, length, inColor, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

332

如果约束涉及两个关系，则应在两个关系中都给出约束声明。这样，无论哪个关系被改变，约束都将对插入和更新做检查。由于删除操作不可能维护基于元组的约束，所以暂不考虑。

* a) 如果是1939年以前制作的电影，则该电影不能是彩色影片。

b) 电影明星不可能出现在制作于其出生日期之前的影片之中。

! c) 两个制片厂不能有相同的地址。

*! d) 在MovieStar中出现的名字不能也出现在MovieExec中。

! e) Studio中出现的制片厂名字至少要在Movie的一个元组中出现。

!! f) 如果某人既是某部电影制片人又是制片厂经理，那么必须由这家制片厂来制作这部电影。

习题7.2.5 关于“PC”模式写出如下基于元组的CHECK约束。

a) 处理器速度低于1200的PC价格不能超过$1500。

b) 显示器小于15英寸的手提电脑要么硬盘至少有20G，要么售价低于$2000。

习题7.2.6 关于习题5.2.4中战船模式写出如下基于元组的CHECK约束。

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

a) 没有哪一类战船具有大于16英寸口径的火炮。

b) 如果某类船只的火炮多于9门，则这些火炮的口径不能大于14英寸。

! c) 船没下水前不能参战。

7.3 修改约束

任何时候都可以添加、修改、删除约束。表示这种修改的方式依赖于该约束是涉及属性、表还是（如同7.4.1节）数据库模式。

333

7.3.1 给约束命名

为了修改或删除一个已经存在的约束，约束必须有名字。为了命名，在约束前加保留字CONSTRAINT和该约束的名字。

例7.11 重写图7-1的第2行，为主键约束命名，如：

```
2)    name CHAR(30) CONSTRAINT NameIsKey PRIMARY KEY,
```

同样，对例7.8中的基于元组CHECK约束命名如下：

```
4) gender CHAR(1) CONSTRAINT NoAndro
                  CHECK (gender IN ('F', 'M')),
```

最后，如下约束：

```
6)    CONSTRAINT RightTitle
          CHECK (gender = 'F' OR name NOT LIKE 'Ms.%');
```

是为图7-5中第6行基于元组CHECK约束命名重写的语句。 □

7.3.2 修改表上约束

7.1.6节中提到，通过SET CONSTRAINT语句，可以将约束检查从立即执行转换到延期执行，或者反过来，从延期执行转为立即执行。对约束的其他改变使用ALTER TABLE语句。6.6.3节中已讨论过某些ALTER TABLE语句的应用，那里用它来添加和删除属性。

这些语句也能用于修改约束。ALTER TABLE用于基于属性和基于元组的检查。用保留字DROP和要删除的约束的名字可以删除约束。也可以用保留字ADD，后跟要添加的约束实现约束添加。注意，除非要添加的约束满足表的当前实例，否则不能对表添加约束。

为你的约束命名

记住，最好为你的每个约束都起个名字，即使你认为不会引用到它也要如此。如果约束创建时没有命名，再想为它起名就晚了。但是，当你必须要改变一个没有名字的约束时，你会发现DBMS可能已经为你提供了查找此约束的一种方式，它给出了一个列有你所有约束的列表，并为其中未命名的约束给出了DBMS的内部名称，你可以用此名称来引用约束。

例7.12 对例7.11中关系MovieStar添加和删除约束。下面是三个删除的语句序列：

```
ALTER TABLE MovieStar DROP CONSTRAINT NameIsKey;
ALTER TABLE MovieStar DROP CONSTRAINT NoAndro;
ALTER TABLE MovieStar DROP CONSTRAINT RightTitle;
```

在删除了这些约束后，如果想要恢复这些约束，可以通过对MovieStar关系修改模式，添加相同的约束。例如： 334

```
ALTER TABLE MovieStar ADD CONSTRAINT NameIsKey
    PRIMARY KEY (name);
ALTER TABLE MovieStar ADD CONSTRAINT NoAndro
    CHECK (gender IN ('F', 'M'));
ALTER TABLE MovieStar ADD CONSTRAINT RightTitle
    CHECK (gender = 'F' OR name NOT LIKE 'Ms.%');
```

这些约束都是基于元组而不是基于属性的检查，不能将其恢复到基于属性的约束。

重新引入的约束的名字可以任意给定。可是，不能信赖SQL能记住已删除的约束，因此，当添加以前的约束时，需要再次写出该约束，不能仅仅引用它以前的名字。 □

7.3.3 习题

习题7.3.1 按照如下要求修改电影例子的关系模式:

```
Movie(title, year, length, inColor, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

* a) 将title和year作为Movie的键。

b) 在MovieExec中每个影片制片人都必须出现的引用完整性约束。

c) 影片的长度不能少于60，也不能多于250。 335

*! d) 同一个名字不能在影片中的影星和影片制片人中同时出现（该约束在删除中不必维护）。

! e) 两个制片厂不能有同一个地址。

习题7.3.2 修改战舰数据库模式，使其有如下基于元组的CHECK约束：

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

a) 关系Classes 中class和country 形成键。

b) 在Battles中出现的每艘船也在Ships中出现的引用完整约束。

c) 在Outcomes 中出现的每艘船也在Ships中出现的引用完整约束。

d) 没有船装备有多于14门火炮。

! e) 不允许船下水前就参战。

7.4 模式层的约束和触发器

SQL中主动元素的最强有力形式与特定的元组或元组的组成并不相关。这些元素称做“触发”和“断言”，是数据库模式的一部分，等同于关系和视图本身。

- 断言是SQL逻辑表达式，并且总是为真。
- 触发是一系列与某个事件相关的动作，例如向关系插入元组。触发总是当这些事件发生时被执行。

由于断言只要求程序员声明什么是真，所以断言很便于程序员使用。但是，触发是DBMS的特征，特别提供作通用目的主动元素。理由是，断言的有效实现非常困难。DBMS必须推断数据库的任何更新是否影响断言的真假。另一方面，触发则确切地告知DBMS需要在何时处理
336 这些影响。

7.4.1 断言

SQL标准提出了一种允许强制任何条件的简单的断言形式（也称做“通用约束”）（WHERE之后的表达式）。断言与其他模式成分一样，用CREATE语句声明。断言的形式是：

1. 保留字CREATE ASSERTION;
2. 断言名；
3. 保留字CHECK;
4. 圆括弧括起的条件。

也就是说，语句的形式是：

```
CREATE ASSERTION <断言名> CHECK (<条件>)
```

当断言建立时，断言的条件必须是真，并且要永远保持是真。任何引起断言条件为假的数据库更新都被拒绝。已经介绍过的其他类型CHECK约束，如果涉及子查询，可以在某些条件下避免操作被拒绝。

基于元组的CHECK约束和断言约束有差别。基于元组的检查声明能引用关系的属性，例如，在图7-5的第6行中，使用属性gender和name而不必指明其来自何处。因为表是在CREATE TABLE语句中声明，所以它们可以引用并插入或更新表MovieStar元组的属性值。

断言条件没有如此特权。断言条件中引用的任何属性都必须要介绍，特别是在select-from-where表达式中要提及对应的关系。由于条件必须是逻辑值，因此，必须用某种方式聚集条件的结果，以获得单个的真/假值选择。例如，可能有一些条件表达式的结果产生一个关系，此时用NOT EXISTS。也就是说，约束该关系永远是空。另外，也可以在关系的一个列上使用SUM一类聚集操作，将其结果与一常数比较。例如，在这种方法中，要求SUM值总是小于某个限定值。

例7.13 假如希望其净资产值少于$10 000 000的人不能成为制片厂经理。可以写一个断言，声明经理净资产值少于$10 000 000的电影制片厂集合是空。该断言涉及两个关系：

```
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```
337

断言描述如图7-6所示。

```
CREATE ASSERTION RichPres CHECK
    (NOT EXISTS
        (SELECT *
         FROM Studio, MovieExec
         WHERE presC# = cert# AND netWorth < 10000000
        )
    );
```

图7-6 保证制片厂经理拥有相当规模资产的断言

虽然该约束涉及两个关系，但是使用断言的价值不大，因为可以在两个关系上使用基于元组的CHECK约束，而不使用两个关系上的单个断言。例如，在例7.3的CREATE TABLE语句上如图7-7所示对Studio加约束。

可是注意，图7-7中的约束仅仅只在对Studio更新时才做检查。而不能查找到关系MovieExec中制片厂经理净资产值少于$10 000 000的情况。为了能获取断言的所有作用，必须在MovieExec表上加另一个约束，声明如果电影制片人是制片厂的经理，那么其净资产值至少要有$10 000 000。 □

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT REFERENCES MovieExec(cert#),
    CHECK (presC# NOT IN
        (SELECT cert# FROM MovieExec
         WHERE netWorth < 10000000)
    )
);
```

图7-7 Studio上与断言约束的镜像

约束的比较

下面的表格列出了基于属性的检查约束、基于元组的检查约束和断言之间的主要区别。

约束类型	声明的位置	动作的时间	确保成立?
基于属性的CHECK	属性	对关系插入元组或属性修改	如果是子查询，则不能确保
基于元组的CHECK	关系模式元素	对关系插入元组或属性修改	如果是子查询，则不能确保
断言	数据库模式元素	对任何提及的关系改变	是

例7.14 另一个断言的例子。涉及关系：

```
Movie(title, year, length, inColor, studioName, producerC#)
```

338

声明对一个给定制片厂，其所有电影的总长度不能超过10 000分钟。

```
CREATE ASSERTION SumLength CHECK (10000 >= ALL
    (SELECT SUM(length) FROM Movie GROUP BY studioName)
);
```

由于该约束只涉及关系Movie，所以可以用基于元组的CHECK约束，而不是用断言来表达。也就是说，对表Movie的定义增加如下基于元组的CHECK约束。

```
CHECK (10000 >= ALL
    (SELECT SUM(length) FROM Movie GROUP BY studioName));
```

原则上，该条件对Movie表的每个元组有效。可是，它并不显示地提及元组的属性，并且所有工作都是在子查询中完成。

另外还要看到，如果作为基于元组的约束实现，对关系Movie元组的删除并不检查是否违反约束。该例中，这个差别并不带来危害。因为如果删除前该约束满足，那么，删除后仍满足约束要求。可是，如果约束是总长度的下限，而不是上限，则将发现由于做基于元组的检查而
339 不是断言，导致违反约束。 □

最后一点，断言可以被删除。删除断言的语句与删除任何数据库模式元素的格式一样，其语句格式是：

```
DROP ASSERTION < 断言名>
```

7.4.2 事件-条件-动作规则

触发器，有时也称做*事件-条件-动作规则*（event-condition-action rule），或者*ECA规则*。触发器与前面已介绍的几种约束有如下三点不同。

1. 当数据库程序员声明的事件发生时，触发器被激活。事件可以是对某个特定关系的插入、删除或更新。很多SQL系统中允许的事件是事务的结束（7.16节中曾对事务做了简洁的介绍，在8.6节将做更详细的声明）。

2. 当触发器被事件激活时，不是立即执行，而是首先由触发器测试触发条件。如果条件不成立，则响应该事件的触发器不做任何事情。

3. 如果触发器声明的条件满足，则与该触发器相连的动作（action）由DBMS执行。动作可以阻止事件发生，或者可以撤销事件（例如删除元组）。事实上，动作可以是任何数据库操作序列，甚至可以与触发事件毫无关联。

7.4.3 SQL中的触发器

SQL触发器语句在事件、条件和动作等部分都为用户提供了多种选择。主要特征有：

1. 动作可以在触发事件之前或之后被执行。

2. 在被触发的事件中，动作既可以指向被插入、删除、修改元组的新值，也可以指向其旧值。

3. 更新事件可以被局限到某个特定的属性或某一些属性。

4. 条件由WHEN短语给出。仅仅当规则被触发，并且触发事件的发生使条件成立时，动作才能被执行。

5. 程序员可以选择动作执行的声明方式：

(a) 一次只对一个更新元组，或者

340 (b) 一次针对在数据库操作中被改变的所有元组。

例7.15 编写应用在下面MovieExec关系上的SQL触发。

```
MovieExec(name, address, cert#, netWorth)
```

当修改netWorth属性时，激活触发器。该触发器的作用是阻挠降低电影出品人净资产值的企

图。触发器声明如图7-8所示。

图中第（1）行用保留字CREATE TRIGGER和触发器名引入触发声明。第（2）行给出名为MovieExec关系的netWorth属性被修改的触发事件。第（3）行到第（5）行建立了一个在触发的条件和动作部分声明旧元组（修改前的元组）和新元组（修改后的元组）的方法。根据第（4）行和第（5）行的声明，新旧元组分别用NewTuple和OldTuple引用。在条件和动作中，这些名字就如同通常SQL查询的FROM短语中的元组变量声明一样使用。

图中第（6）行，FOR EACH ROW短语，表达了该触发器是每修改一个元组执行一次的方式。如果没有该短语，或它被缺省值FOR EACH STATEMENT替换，则该触发器将是每执行一句SQL语句执行一次，而不管这个语句将使多少元组因触发事件被改变。对于新旧元组没有声明别名，但是可以使用下面引入的OLD TABLE 和 NEW TABLE。

第（7）行是触发的条件。声明动作的执行仅仅当新的净资产值低于旧的净资产值时，触发器被激活，也就是出品人净资产值收缩的时候被激活。

第（8）行到第（10）行是动作部分。该动作是通常的SQL修改语句，其作用是把制片人 341
的净产值重新还原为修改前的值。注意，原则上认为每个MovieExec的元组都要被修改，但是第（10）行的WHERE短语保证了该动作仅仅只对那些被修改的元组（即只与新元组的cert#值相等的元组）有作用。 □

```
 1)  CREATE TRIGGER NetWorthTrigger
 2)  AFTER UPDATE OF netWorth ON MovieExec
 3)  REFERENCING
 4)      OLD ROW AS OldTuple,
 5)      NEW ROW AS NewTuple
 6)  FOR EACH ROW
 7)  WHEN (OldTuple.netWorth > NewTuple.netWorth)
 8)      UPDATE MovieExec
 9)      SET netWorth = OldTuple.netWorth
10)      WHERE cert# = NewTuple.cert#;
```

图7-8 SQL 触发器

当然，例7.15仅仅只解释了SQL触发器的一部分特征。下面给出了触发器具有的其他选项，以及如何表达这些选项。

- 通过保留字AFTER，图7-8中的第（2）行指出该条规则将在触发事件之后被执行。AFTER可以用BEFORE替换，替换后，WHEN的条件将在触发事件之前测试，也就是说，是在能唤醒触发器的数据库更新之前执行。如果条件是真，执行触发的动作。然后，唤醒触发器的事件被执行，而不问其条件是否为真。
- 除了UPDATE之外，其他可能的触发事件是INSERT和DELETE。图7-8中第（2）行中OF netWorth短语是UPDATE的选项，它指出目前的事件仅仅是OF保留字后列出的属性的更新。OF短语在INSERT或DELETE中不可使用，因为这两个事件都是作用在整个元组上。
- WHEN短语是可选项。如果该短语缺省，则不问触发器是否被唤醒，都要执行动作。
- 虽然在例子中只显示了单个SQL语句作为动作，但实际上，动作可以是任意多个这样的语句组成，语句由BEGIN……END括起，并且语句之间用分号分隔。
- 当触发事件是修改时，则有旧元组和新元组之分，它们分别是修改之前和修改之后的元组。用OLD ROW AS 和NEW ROW AS短语命名这些元组，如同第（4）行和第（5）行中见到的。相反，删除时，OLD ROW AS被用于命名要被删除的元组，而NEW ROW AS

不可使用。

- 如果忽略第（6）行的FOR EACH ROW，则图7-8中的行级触发（row-level trigger）就变成了语句级触发（statement-level trigger）。一旦有合适类型的语句被执行，语句级触发就被执行，而不问它实际上会影响多少元组——零个、一个或多个。例如，如果用SQL更新语句更新整个表，语句级的修改触发将只执行一次，而元组级触发将对要修改的元组一次一个地执行。在语句级触发中，不能像第（4）行和第（5）行那样，直接引用旧
342 元组和新元组。可是，任何触发器——无论是元组级或语句级——都可以引用旧元组的关系（删除元组或更新元组的旧版本）和新元组的关系（插入元组或更新元组的新版本），声明方式是用保留字OLD TABLE AS OldStuff和NEW TABLE AS NewStuff。

例7.16 假定要阻止电影制片人的平均净资产值降到$500 000。在对关系

```
MovieExec(name, address, cert#, netWorth)
```

的netWorth列做插入、删除或修改时可能会违反上述约束。

该例的细微之处是，当一个INSERT或UPDATE语句导致在很多MovieExec关系元组变更期间，平均净资产值可能暂时低于$500 000，然后，当所有变更结束时，其净资产值将超过$500 000。约束要做的工作是，若语句执行结束后，平均净资产值仍然是低于$500 000，则该更新操作被拒绝。

对于关系MovieExec的插入、删除和更新这三个事件要分别写触发。图7-9给出了更新事件的触发。插入和删除事件的触发与此类似，比较而言更简单一些。

```
 1)  CREATE TRIGGER AvgNetWorthTrigger
 2)  AFTER UPDATE OF netWorth ON MovieExec
 3)  REFERENCING
 4)      OLD TABLE AS OldStuff,
 5)      NEW TABLE AS NewStuff
 6)  FOR EACH STATEMENT
 7)  WHEN (500000 > (SELECT AVG(netWorth) FROM MovieExec))
 8)  BEGIN
 9)      DELETE FROM MovieExec
10)      WHERE (name, address, cert#, netWorth) IN NewStuff;
11)      INSERT INTO MovieExec
12)          (SELECT * FROM OldStuff);
13)  END;
```

图7-9 平均净资产值约束

图中第（3）行到第（5）行声明NewStuff和OldStuff分别是包含新元组和旧元组的关系名，这些元组是唤醒上述触发操作涉及的数据库元组。由于一个数据库语句可以更新关系的很多元组，所以，如果执行这样的语句，在NewStuff和OldStuff中可能有很多元组。

如果是更新操作，则NewStuff和OldStuff中分别是被更新元组的新版本和旧版本。如
343 果类似地给出删除触发，则删除元组是OldStuff，也不需要像本触发那样为NEW TABLE声明NewStuff。同样，在类似的插入触发中，新元组是NewStuff，也不需要声明OldStuff。

第（6）行声明本触发的执行是一语句一次，而不问有多少元组被更新。第（7）行是条件，声明如果修改之后平均净资产值少于$500 000，则条件成立。

第（8）行到第（13）行是动作，由两个语句组成。当WHEN短语中的条件成立时，即新的平均值太低时，该语句将恢复关系MovieExec的原有值。第（9）行到第（10）行删除所有新元组，即元组的被修改过的版本。而第（11）行到第（12）行恢复修改之前的值。 □

7.4.4 替换触发器（Instead-Of Triggers）

替换触发器虽然不符合SQL-99标准，但是非常有用。在标准化讨论中曾提及它，一些商用系统也给予支持。它允许BEFORE或AFTER由INSTEAD OF替换，其意思是说当某个事件唤醒触发器时，完成触发动作以替换事件本身。

当触发器针对存储表时，作用很小。但是，若触发器是用于视图时，起到的作用将是巨大的。原因是，视图不能真正地被更新（参见6.7.4）。替换触发器将终止更新视图的企图，转为执行数据库设计者认为合适的动作。下面是该特征的一个典型例子。

例7.17 派拉蒙（Paramount）电影公司所拥有的所有电影的视图定义如下：

```
CREATE VIEW ParamountMovie AS
   SELECT title, year
   FROM Movie
   WHERE studioName = 'Paramount';
```

如同在例6.49中讨论的那样，该视图可以更新。但是，它有不期望看到的缺点。当插入ParamountMovie元组时，系统不能保证studioName属性确实是Paramount，于是，Movie元组中该属性值是NULL。

如果在视图上创建替换触发器，如图7-10所示，则可以获得较好的结果。这么长的触发器不奇怪，其中第（2）行中的保留字INSTEAD OF使得要插入ParamountMovie的企图永远不会发生。 344

```
1)  CREATE TRIGGER ParamountInsert
2)  INSTEAD OF INSERT ON ParamountMovie
3)  REFERENCING NEW ROW AS NewRow
4)  FOR EACH ROW
5)  INSERT INTO Movie(title, year, studioName)
6)  VALUES(NewRow.title, NewRow.year, 'Paramount');
```

图7-10 使用触发器在基本表上插入以替换对视图的插入

第（5）行和第（6）行是替换企图插入的动作。这里有一个对Movie的插入，并给出了三个已知属性。属性title和year是来自试图插入视图的元组，其值是通过元组变量NewRow引用，元组变量在第（3）行中声明，表示要插入的元组。属性StudioName的值是常数'Paramount'，该值不是插入元组的内容。另外，对于插入的电影，由于其是来自视图ParamountMovie，所以这样假定制片厂是正确的。 □

7.4.5 习题

习题7.4.1 为MovieExec的删除和插入事件编写类似于图7-9的触发器。

习题7.4.2 将如下要求写成触发器或断言。在每种情况中，如果不满足声明的约束，则拒绝或撤销更新。数据库模式是习题5.2.1中的“PC”例子。

```
Product(maker, model, type)
PC(model, speed, ram, hd, rd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

* a) 当修改PC的价格时，检查不存在速度与其相同，但价格更低的PC机。

* b) 没有同时也制造手提电脑的PC制造商。

*! c) PC制造商只能制造处理器速度至少等于PC的手提电脑。

d) 插入新打印机时，检查其型号是否已在Product中存在。

! e) 当对Laptop关系做任何更新时，要求每个制造商生产的手提电脑的平均价格至少是$2000。

! f) 当修改任何PC机的RAM或硬盘时，要求被修改的PC机的RAM速度至少是其硬盘速度的100倍。

345

! g) 如果手提电脑的内存多于PC，则手提电脑的价格也应高于PC。

! h) 当插入新的PC、手提电脑或打印机时，要确保型号与以前已有的PC、手提电脑或打印机型号不重复。

! i) 如果Product关系中有某个型号和它的类型，则该型号必须也出现在对应的某个合适的关系中。

习题7.4.3 将如下要求编写为断言或触发器。在每种情形，如果不满足描述的约束，则拒绝或撤销相应的更新。数据库模式是习题5.2.4中的战船例子。

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

* a) 当插入一新类到Classes时，也插入一具有该类名字和NULL下水日期的舰船。

b) 允许插入一排水量超过35 000吨的新类，但是要改变其排水量为35 000。

c) 同一类型的舰船不能多于2艘。

! d) 同一国家不能同时具有战船和战斗巡洋舰。

! e) 战斗中，多于9门火炮的战船不能被少于9门火炮的战船击沉。

! f) 当插入Outcomes元组时，要分别检查Ships和Battles关系中的船与战役元组。如果没有这样的船和战役存在，则在插入这些元组时，其中不确定的属性要赋以NULL值。

! g) 在向Ships中插入或更新Ships的class属性时，核查没有国家拥有20艘以上的战船。

! h) 没有船可以比与类同名的船只早下水。

! i) 对一个类，应该有一艘船具有该类的名字。

!! j) 在所有可能引起违反约束的环境下检查，没有船可以在此船已被击沉之后又出现在战役中。

! **习题7.4.4** 将如下要求编写为断言或触发器。在每种情况，如果不满足所要求的约束，则拒绝或撤销相应的更新。所有要求是在有关电影例子的关系上提出的。

346

```
Movie(title, year, length, inColor, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

你可以假定所有条件在数据库被改变之前成立。另外，当更新破坏条件时，系统宁可选择更新数据库，即使是用NULL值或缺省值插入元组，也不拒绝更新。

a) 保证在所有时间里，任何在StarsIn中出现的影星也出现在MovieStar中。

b) 保证在所有时间里，每个电影制片人是一制片厂经理、电影制片人或二者兼而有之。

c) 保证每个电影至少有一个男明星和一个女明星。

d) 保证在任一年中，任何制片厂制作的电影数量不能多于100。

e) 任何年代的电影平均长度不超过120。

7.5 小结

- 键约束（key constraint）：在关系模式中可以用UNIQUE或PRIMARY KEY定义一个属性或一组属性为键。
- 引用完整性（referential Integrity constraint）：在关系模式中，出现在某个属性或一组属性中的值，必须也出现在用REFERENCES或FOREIGN KEY声明的、另一个关系的某些元组的相应属性中。
- 基于属性的CHECK约束（attribute-based check constraint）：关系模式属性声明的后面加保留字CHECK和条件，可以实现对属性值的约束。
- 基于元组的CHECK约束（tuple-based check constraint）：通过在关系本身的声明中加CHECK保留字和要检查的条件，实现对关系元组的约束。
- 修改约束（modifying constraint）：用ALTER语句可以添加或删除基于元组的检查约束。 347
- 断言（assertion）：可以用保留字CHECK和要检查的条件声明断言，使其成为数据库模式的元素。该条件可以涉及一个或多个数据库模式关系，既可以用聚集将整个关系作为一个整体，也可以只对单个的元组。
- 断言执行（invoking the check）：断言涉及的关系被改变时，断言声明的条件被检查。基于属性和基于元组的检查仅仅当属性或关系被插入或更新操作改变时执行。因此，这些约束有子查询时将被侵犯。
- 触发器（trigger）：SQL标准包括触发器声明唤醒该触发器的特定事件（例如对某个关系的插入、删除或更新）。一旦触发器被唤醒，触发条件被检查。如果条件是真，则执行指明的动作序列（SQL语句，如查询和数据库更新）。

7.6 参考文献

读者必须参看第6章的书目摘记，以获知如何得到SQL2或SQL-99标准文档。参考文献[5]和[4]概述了数据库系统中主动元素的所有方面。[1]讨论了SQL-99和将来标准中最新的主动元素。引文[2]和[3]讨论HiPAC，这是一个早期主动数据库元素的原型系统。

1. Cochrane, R. J., H. Pirahesh, and N. Mattos, "Integrating triggers and declarative constraints in SQL database systems," *Intl. Conf. on Very Large Database Systems*, pp. 567–579, 1996.

2. Dayal, U., et al., "The HiPAC project: combining active databases and timing constraints," *SIGMOD Record* **17**:1, pp. 51–70, 1988.

3. McCarthy, D. R., and U. Dayal, "The architecture of an active database management system," *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 215–224, 1989.

4. N. W. Paton and O. Diaz, "Active database systems," *Computing Surveys* **31**:1 (March, 1999), pp. 63–103.

5. Widom, J. and S. Ceri, *Active Database Systems*, Morgan-Kaufmann, San Francisco, 1996.

348

第8章 SQL的系统特征

本章讨论如何把SQL应用到一个完整的编程环境中。8.1节讨论SQL如何嵌入到普通程序设计语言，比如C语言，编写的程序中。一个关键问题是在SQL关系与环境变量，或者“宿主”语言之间如何交换数据。

8.2节考虑用另一种方式将SQL和被称为持久性存储模块的一般通用程序结合起来，这些模块是以数据库模式形式存储的代码段，由用户以命令的形式执行。8.3节涵盖了系统的附加问题，例如对于客户-服务器计算模型的支持。

第三种编程方式叫“调用级界面”，以常规语言编程，用函数库来访问数据库。为了从C程序中做调用，8.4节讨论叫做SQL/CLI的SQL标准库。接着，8.5节将接触Java的JDBC（数据库连接），这是一种可供选择的调用级界面。

然后，8.6节介绍了“事务”，它是工作的一个原子单元。很多数据库应用程序，如银行系统，要求数据是原子性的，或者是不可分的，甚至有时许多并发操作同时进行。SQL提供了指定事务的功能，而且SQL系统的机制可以确认所调用的事务真正以原子方式执行。最后，8.7节讨论SQL对未经授权访问数据如何控制，以及SQL系统如何得知哪些是经过授权的访问。

8.1 编程环境下的SQL

到目前为止，例子中都使用了类SQL界面。也就是说，假定有一个SQL解释器来接受和执行各种已经学习过的SQL查询和命令。虽然这种操作模式几乎是由所有的DBMS作为一个选项提供的，但实际很少用到。实际上，大多数SQL语句是一个更大程序段的一部分。更现实的观点是在一些常规的宿主语言，如C，编写的程序中有SQL语句的功能。这一节将描述SQL在常规程序中运作的一种方式。 349

包含SQL语句的典型编程系统的框架如图8-1所示。图中，程序员用一种宿主语言编程，但是程序中用到了一些特殊的并不是宿主语言部分的 “嵌套”SQL语句。整个程序被发送给预处理器，预处理器将嵌套SQL语句转化成可以被宿主语言理解的形式，SQL的表达可以简化为调用函数，此函数把SQL语句当作字符串参数，并且执行这个SQL语句。

SQL标准语言

符合SQL标准的实现要求支持以下七种宿主语言中的至少一种：ADA，C，Cobol，Fortran，M（也叫Mumps），Pascal，和PL/I。计算机系的学生应该熟悉这些语言，M（或者叫Mumps）可能除外，因为它主要应用于医疗业。本书例子中用的是C语言。

经过预处理的宿主语言程序随后以通常的方式编译。一般情况下DBMS销售商提供了支持必要的函数定义库。这样，实现SQL的函数被执行，并且整个程序像一个整体一样运作。另外，图8-1中也显示这样一种可能性：程序员直接用宿主语言写程序，只是在必要时使用这些函数调用。这种方式通常称为调用级界面或者CLI，它将在8.4节介绍。

8.1.1 阻抗不匹配问题

连接SQL语句和那些常规的编程语言的基本问题就是*阻抗不匹配*，即SQL的数据模式与其他语言的模式差别甚大。众所周知，SQL的核心使用的是关系数据模型。然而，C和其他普通的编程语言使用的数据模型有整型、实型、算术型、字符型、指针、记录、数组等等。集合在C或者其他语言中不能直接表示，相对地，SQL不使用指针、循环和转移，或者其他普通编程语言的结构体。因此，在SQL和别的语言之间不能直接转移数据，必须设计一种机制允许程序的开发既可以使用SQL，也可以使用一
350 种其他的语言。

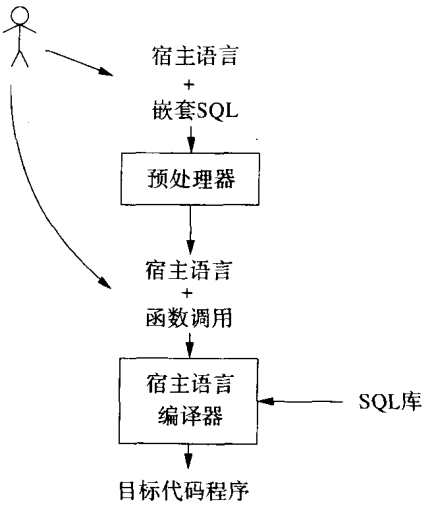


图8-1 处理嵌套SQL语句的过程

第一个可能的猜想是只用一种语言看来更可取，要么使用SQL完成所有的计算，要么不用SQL，只用常规语言完成所有的计算。然而，当涉及到数据库操作时，忽略SQL的想法很快被放弃了。SQL系统很大程度上帮助了程序员编写数据库操作，使这些操作可以有效地执行，并且操作的表示级别很高。SQL降低了程序员对于数据在存储器中如何组织或者如何利用这个存储结构在数据库中高效运行的理解需求。

另一方面，有许多重要的事情SQL根本不能完成。例如，不能用SQL查询来计算数n的阶乘[$n! = n \times (n-1) \times \ldots \times 2 \times 1$]，这个问题用C或者类似的语言[⊖]就可以很轻松地完成。另外一个例子是，SQL不能把输出直接格式化到图表等常规的格式中。所以，真正的数据库编程既要有
351 SQL，也要有常规语言。后者常常被叫做*宿主语言*（host language）。

8.1.2 SQL/宿主语言接口

数据库只能由SQL语句访问，在数据库和宿主语言程序之间的信息转移是通过宿主语言变量实现，这种变量可以被SQL语句读写。在SQL语句中引用这些*共享变量*（shared variable）时，变量前面要加上冒号作为前缀，而在宿主语言中这些变量并不需要冒号。

在宿主语言中使用SQL语句时，SQL代码语句必须紧跟在关键字EXEC SQL之后。系统将预处理这些语句，用宿主语言中合适的函数调用来代替这些语句，并且充分利用与SQL相关的函数库。

在SQL标准中，SQLSTATE这个特殊变量用于连接宿主语言程序与SQL执行系统。SQLSTATE是5个字符的数组类型。每次调用SQL库函数，向SQLSTATE变量中存放一个代码，该代码表示调用过程中出现的问题。SQL标准同时指定了大量的5个字符的代码和它们的意义。

例如，'00000'（五个零）表示没有产生任何错误，'02000'表示缺少作为SQL查询结果组成部分的一个元组。后面这个代码非常重要，因为它允许在宿主语言中创建的一个循环，每次检查关系中的一个元组，直至检查最后一个元组后结束。宿主语言可以读取SQLSTATE的值，并根据读取的值做出相应的决策。

⊖ 这儿要注意，如在10.4节中讨论的递归SQL或者8.2节中讨论到的SQL/PSM一样，基本SQL语言的扩张确实提供了“图灵完备性”，也就是说，能够计算用其他编程语言计算的任何问题。然而，这些扩展不准备用于通用目的计算，因此它们不被看作是通用性语言。

8.1.3 DECLARE节

共享变量的声明加入到如下两个嵌套SQL语句之间：

```
EXEC SQL BEGIN DECLARE SECTION;
    ...
EXEC SQL END DECLARE SECTION;
```

两个语句之间称为declare节。Declare节的变量声明形式可以是宿主语言要求的任何形式。更进一步讲，为使声明的变量有意义，其类型可以是宿主语言和SQL都能够处理的，如整型，实型和字符型，或者数组类型。

例8.1 下面的语句是更新Studio关系的C函数的一部分： 352

```
EXEC SQL BEGIN DECLARE SECTION;
    char studioName[50], studioAddr[256];
    char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
```

第一个和最后一个语句是DECLARE节必需的开头和结束。中间的语句声明了两个变量studioName和studioAddr。它们都是字符型数组，如将看到的，是用来保存一个制片厂的名称和地址，它们组成一个元组并插入到Studio关系式中。第三个语句将SQLSTATE声明为包含6个字符的数组[⊖]。 □

8.1.4 使用共享变量

共享变量可以用在SQL语句中任何需要和允许常量的地方。回忆一下，共享变量这样使用时，都要加前缀冒号。下面的例子使用了例8.1中的变量作为元组的一部分插入到Studio关系中。

例8.2 图8-2中显示了一个C函数：getStudio，该函数要求用户提供一个制片厂的名称和地址，读取结果，并将合适的元组插入到Studio。第1行到第4行是例8.1中已知的声明。图中省略了打印要求的C代码以及扫描所输入的文本并填写到两个数组studioName和studioAddr的C代码。

接着，第5行和第6行是常规的INSERT语句。这条语句以EXEC SQL开头，表明这实际上是一个嵌套SQL语句，而非不符合语法的C代码。图8-1中的预处理器将寻找EXEC SQL，以便检测必须预处理的语句。

第5行和第6行插入的值不是显式常量，和前面例子如例6.34一样。而且，第6行中出现的值是共享变量，这些共享变量的当前值构成了被插入元组的一部分。 □

除了INSERT语句外，还有许多SQL语句可以通过共享变量作为接口，嵌入到宿主语言中。在宿主语言中，每个嵌套SQL语句以EXEC SQL打头，并且可以共享变量代替常量。任何不返回结果的SQL语句（也就是说，非查询语句）都可以被嵌套。可嵌套的SQL语句包括DELETE和UPDATE语句以及那些创建、更新、或者删除表和视图等模式元素的语句。 353

然而，由于“阻抗不匹配”，select-from-where查询不能直接嵌套到宿主语言。查询产生的结果是元组集合，但是大多数宿主语言均不直接支持集合数据类型。因此，为了将查询结果连接到宿主语言程序，嵌套SQL不得不从下面两种机制中选择一种。

⊖ 对于5字符值的SQLSTATE，使用了6个字符，因为下面的程序中，使用C函数strcmp来测试SQLSTATE是否是确定的值。既然strcmp希望字符串以'\0'结束，那么对于这个结束符，就需要第6个字符。第六个字符必须初始化为'\0'，后面的程序中将不再说明这个赋值。

```
    void getStudio() {

1)      EXEC SQL BEGIN DECLARE SECTION;
2)          char studioName[50], studioAddr[256];
3)          char SQLSTATE[6];
4)      EXEC SQL END DECLARE SECTION;

        /* print request that studio name and address
           be entered and read response into variables
           studioName and studioAddr */

5)      EXEC SQL INSERT INTO Studio(name, address)
6)               VALUES (:studioName, :studioAddr);
    }
```

图8-2 使用共享变量插入新的制片厂

1. 只有一个结果元组的查询可以将该元组存储到共享变量中，一个变量对应元组的一个字段值。为此，使用了select-from-where语句的一种改进形式：单元组选择。

2. 对于结果元组多于一个的查询，可以为它声明一个游标。游标扫描了结果关系中的所有元组，每个元组依次被提取到共享变量，并由宿主语言进行处理。

下面依次对每种机制进行讨论。

8.1.5 单元组选择语句

单元组选择的形式类似于普通的select-from-where语句，只是SELECT子句后紧跟着关键字INTO和一连串的共享变量。与SQL语句中的所有共享变量一样，这些共享变量以冒号作为前缀。如果查询结果是个单一元组，那么这个元组的组成成分将被赋予这些共享变量。如果结果没有元组或者多于一个元组，那么不会分配给这些共享变量，同时一个相应的错误码被写入到
354 SQLSTATE变量中。

例8.3 写一个C函数来读取一个制片厂的名称并输出制片厂经理的资产。函数如图8-3所示。它开始于第1行到第5行的声明部分，声明了所需的变量。接着，从标准输入取得制片厂的名称，这个C语句并没有明确写出。

第6行到第9行是单元组选择语句，这与以前看到的查询十分相似。有两个不同：一是在第9行的条件中，变量studioName的值被用来代替常量字符串；二是，第7行有一条INTO子句显示了查询结果放置的位置。这种情况下，只希望产生单一的元组，且元组只有一个组成部分，即属性netWorth。这个元组的这个组成成分的值被存储在共享变量preNetWorth中。 □

8.1.6 游标

将SQL查询连接到宿主语言中最通用的方法是使用游标，游标快速遍历关系的元组。这个关系可以是一个被存储的表，也可以是由查询产生的结果。为了创建和使用游标，下列语句是
355 必须的：

1. 游标定义。游标声明的最简形式要包含：

(a) 引导词EXEC SQL，如同所有的嵌套SQL子句。

(b) 关键字DECLARE。

(c) 游标的名称。

(d) 关键字CURSOR FOR。

(e) 诸如关系的名称或者select-from-where语句之类的表达式，其值是一个关系。被声明的游标扫描这个关系的元组。也就是说，当使用游标“取值”时，游标依次指向这个关系的每个元组。

```
    void printNetWorth() {

1)      EXEC SQL BEGIN DECLARE SECTION;
2)          char studioName[50];
3)          int presNetWorth;
4)          char SQLSTATE[6];
5)      EXEC SQL END DECLARE SECTION;

        /* print request that studio name be entered.
           read response into studioName */

6)      EXEC SQL SELECT netWorth
7)               INTO :presNetWorth
8)               FROM Studio, MovieExec
9)               WHERE presC# = cert# AND
                       Studio.name = :studioName;

        /* check that SQLSTATE has all 0's and if so, print
           the value of presNetWorth */
    }
```

图8-3 嵌套在C函数中的单行元组选择

总之，游标的定义形式如下：

EXEC SQL DECLARE<cursor> CURSOR FOR<query>

2. 语句EXEC SQL OPEN，其后跟随着游标的名称。这个语句初始化游标的位置，使游标指向其扫描的那个关系中第一个元组，并从那里开始检索。

3. 一次或者多次使用fetch子句。fetch子句的目的是得到游标扫描的那个关系中的下一个元组。如果元组已经被遍历过，那么不会返回任何元组，且SQLSTATE被赋值为’02000’，这个代码表示“没有发现任何元组”。fetch子句的构成如下：

(a) 关键字EXEC SQL FETCH FROM。

(b) 游标的名称。

(c) 关键字INTO。

(d) 共享变量的列表，由逗号分隔。如果要提取一个元组，那么这个元组的组成部分被依次存入到这些变量中。

也就是说，fetch子句的形式如下：

EXEC SQL FETCH FROM<cursor> INTO<list of variables>

4. EXEC SQL CLOSE子句，其后跟随着游标的名称。这条语句关闭游标，游标将不再扫描关系的元组。然而，游标可以由另外一条OPEN语句重新初始化，它将重新扫描这个关系的元组。

356

例8.4 查询满足以下条件的电影制片人的个数，这些制片人的净资产成指数型增长，每个数值段对应代表净资产的位数。因此设计了一个查询，该查询用来检索所有MovieExec元组的netWorth字段，并存入worth共享变量中。游标execCursor将扫描所有的元组。每取一个元组时，计算整数worth的位数，并将数组counts中相应元素加1。

C函数worthRanges开始于图8-4的第1行。第2行声明了只能被C函数使用，而不能被嵌套SQL语句使用的变量。数组counts保存了在不同带中的制片人数目，digits计算净资产的位数，而*i*是一个下标，用来检索数组counts的元素。

```
1)  void worthRanges() {

2)      int i, digits, counts[15];
3)      EXEC SQL BEGIN DECLARE SECTION;
4)          int worth;
5)          char SQLSTATE[6];
6)      EXEC SQL END DECLARE SECTION;
7)      EXEC SQL DECLARE execCursor CURSOR FOR
8)          SELECT netWorth FROM MovieExec;

9)      EXEC SQL OPEN execCursor;
10)     for(i=0; i<15; i++) counts[i] = 0;
11)     while(1) {
12)         EXEC SQL FETCH FROM execCursor INTO :worth;
13)         if(NO_MORE_TUPLES) break;
14)         digits = 1;
15)         while((worth /= 10) > 0) digits++;
16)         if(digits <= 14) counts[digits]++;
        }
17)     EXEC SQL CLOSE execCursor;
18)     for(i=0; i<15; i++)
19)         printf("digits = %d: number of execs = %d\n",
                i, counts[i]);
    }
```

图8-4 分组出品人净资产呈现出指数型增长

第3行到第6行是SQL的声明节，声明了共享变量worth和SQLSTATE。第7行和第8行定义了游标execCursor，该游标遍历由第8行的查询所产生的值。这个查询仅仅给出MovieExec中所有元组的netWorth字段。游标在第9行打开。第10行通过数组counts的元素的置零来完成初始化。

357

主要的工作由第11行到第16行的循环完成。在第12行，一个元组被读取到共享变量worth中。虽然，一般说，变量的数目和被检索的元组的组成数目一样，但由于第8行的查询产生的元组只有一个成分，所以只需一个共享变量。第13行测试提取是否成功。这里，使用了一个宏NO_MORE_TUPLES，其定义如下：

```
#define NO_MORE_TUPLES !(strcmp(SQLSTATE,"02000"))
```

回忆一下，”02000”是一个SQLSTATE代码，表示没有找到任何元组。这样，第13行测试是否所有由查询返回的元组以前已经被扫描过，能否得到“下一个”元组。如果是的话，就中断循环，跳转到第17行。

如果提取了一个元组，那么第14行将净资产的位数初始化为1。第15行是一个循环，此循环重复地用10除净资产，并且将digits重复加1。当净资产被10除以后为零时，digits保存了那些原本被检索过的worth值的正确位数。最后，第16行将数组counts中对应元素加1。假定位数不超过14。然而，假定净资产有15或者更多的数位，第16行对数组counts的元素不作任何增加，因为没有对应的范围；也就说，过大的净资产被丢弃但并不影响统计。

函数的结束开始于第17行。关闭游标。第18行和第19行输出数组counts中的值。 □

8.1.7 游标修改

游标扫描一个基本表的元组（也就是说，存储到数据库中的关系，而不是查询建立的视图或者关系）时，不仅可以读和处理每个元组的值，而且可以更新或者删除元组。UPDATE和DELETE语句的语法除了WHERE子句外和第6.5节中的一样。这里WHERE子句只能是WHERE CURRENT OF，其后跟着游标的名称。对于读取元组的宿主语言而言，在决定删除或者更新这个元组之前，它当然可能将任何条件应用到该元组中。

例8.5 图8-5中的C函数查看MovieExec的每个元组并决定是删除元组还是将净资产翻倍。第3行和第4行定义了与MovieExec的四个属性相对应的变量，以及必需的SQLSTATE。然后，在第6行中声明的execCursor扫描存储模式MovieExec自身。注意，当试图通过游标修改元组时，该游标扫描某个作为查询结果的临时关系。如果游标扫描MovieExec之类的存储模式，就将对数据库产生持久的影响。 358

```
1)  void changeWorth() {

2)      EXEC SQL BEGIN DECLARE SECTION;
3)          int certNo, worth;
4)          char execName[30], execAddr[256], SQLSTATE[6];
5)      EXEC SQL END DECLARE SECTION;
6)      EXEC SQL DECLARE execCursor CURSOR FOR MovieExec;

7)      EXEC SQL OPEN execCursor;
8)      while(1) {
9)          EXEC SQL FETCH FROM execCursor INTO :execName,
                :execAddr, :certNo, :worth;
10)         if(NO_MORE_TUPLES) break;
11)         if (worth < 1000)
12)             EXEC SQL DELETE FROM MovieExec
                        WHERE CURRENT OF execCursor;
13)         else
14)             EXEC SQL UPDATE MovieExec
                        SET netWorth = 2 * netWorth
                        WHERE CURRENT OF execCursor;
        }
15)     EXEC SQL CLOSE execCursor;
    }
```

图8-5 修改制片人的净资产

第8行到第14行是循环，循环中游标execCursor依次指向MovieExec的每个元组。第9行将当前元组提取到为此设置的四个变量中，通常只用到了worth。第10行测试MovieExec的元组是否都检索过了。这里再次使用了宏ON_MORE_TUPLES，作为变量SQLSTATE包含“没有元组”的”02000”代码的条件。

第11行查询净资产是否低于$1000。如果是，元组被第12行的DELETE语句删除。注意涉及到游标的WHERE子句，这样刚刚提取的MovieExec的当前值从MovieExec中删除了。如果净资产至少是$1000，那么在第14行，同一个元组中的净资产被翻倍。 □ 359

8.1.8 防止并发更新

假定用图8-4中的函数worthRanges检查电影制片人的净资产时，某个其他进程正在修改底层的MovieExec关系。在第8.6节中讨论事务时，将就一些进程同时访问同一个数据库做更详细地说明。然而目前，仅仅说是存在这样的可能性：别的进程可以修改正在使用中的关系。

对于这种可能性，该怎么办呢？也许什么都不用做。可能近似的统计就足够了，例如，人们并不必关心正在被删除的制片人是否已统计过了。这样，只通过游标读取元组就可以了。

可是，人们并不希望游标读取的元组被并发的变化所影响。而是强调统计是针对某个时刻已存在的关系。在收集统计数据之前，并不能准确地把握MovieExec发生了哪些修改，但是，希望要么所有的修改语句被在函数worthRanges运行之前已经彻底完成，要么在该函数运行之后才开始进行，而不用关心一条修改语句影响了多少个制片人。为了保证这一点，有并发变化时，可以将游标声明为不敏感(insensitive)。

例8.6 图8-4中的第7行和第8行修改成如下形式：

```
7)     EXEC SQL DECLARE execCursor INSENSITIVE CURSOR FOR
8)         SELECT netWorth FROM MovieExec;
```

这样定义的execCursor，使得SQL系统保证了在execCursor打开和关闭之间对关系MovieExec所作的变化不会影响提取到的元组集合。 □

声明游标不敏感的代价很高，因为SQL系统必须花大量的时间管理数据访问，以确信游标是不敏感的。同样，8.6节将再次讨论对数据库并发操作的管理。然而，支持不敏感游标的一个简单方法就是，使SQL系统挂起那些访问了不敏感游标查询所用到关系的进程。

某些关系R上的游标可以确信不会改变R。这样的游标可以与R的不敏感游标同时运行，而不用担心会改变不敏感游标读到的关系R。如果游标声明为FOR READ ONLY，那么数据库系统可以确信基本关系不会因为游标对这个关系的访问而被修改。

例8.7 在图8-4的第8行以后加上下面一行：

```
FOR READ ONLY;
```

360 这样，任何试图通过游标execCursor所做出的修改都会产生错误。 □

8.1.9 卷型游标

游标提供了怎样遍及一个关系的元组的选择。缺省的、并且最通常的选择是从关系顶端开始，然后依次提取元组，直至末尾。不过，还可以按别的顺序提取元组，在游标关闭前可以扫描元组好几次。为了能获得这样选择的优越性，必须作如下两件事情。

1. 声明游标时，将关键字SCROLL置于保留字CURSOR之前。这个变化告诉了SQL系统，游标的使用方式不只是按照元组的顺序向前移动。

2. FETCH语句中，关键字FETCH后面跟着的是下面所列选项中的一个，选项决定所期望的元组的位置。

(a) NEXT或者PRIOR按顺序提取下一个或者上一个元组。记住这些元组是相对于游标的当前位置。如果没有指定选项，NEXT是缺省的选择，也是最常使用的。

(b) FIRST或者LAST提取顺序中的第一个或最后一个元组。

(c) RELATIVE后面跟随一个正整数或负整数，这个整数表示所期望的元组按顺序向前（对于正整数）或向后（对于负整数）所要移动的元组数。如，RELATIVE 1与NEXT意义相同，RELATIVE -1与PRIOR意义相同。

(d) ABSOLUTE后面跟随一个正整数或负整数，这个整数表示所期望的元组的位置是在从头部（对于正整数）或从尾部（对于负整数）开始计算多少个。如，ABSOLUTE 1与FIRST意义相同，ABSOLUTE -1与LAST意义相同。

例8.8 重写图8-5的函数，使之从最后一个元组回溯遍历全部元组。首先，游标声明

execCursor为卷型，即在第6行加上关键字SCROLL：

```
6) EXEC SQL DECLARE execCursor SCROLL CURSOR FOR MovieExec;
```

另外，需要用FETCH LAST语句初始化元组的提取，循环中使用FETCH PRIOR。图8-5的第8行到第14行的循环在图8-6中被重写。读者不要以为逆序读取在MovieExec中的顺序存储元组有什么优势。 □

```
EXEC SQL FETCH LAST FROM execCursor INTO :execName,
    :execAddr, :certNo, :worth;
while(1) {
    /* same as lines (10) through (14) */
    EXEC SQL FETCH PRIOR FROM execCursor INTO :worth;
}
```

图8-6 逆序读取MovieExec

8.1.10 动态SQL

目前为止，在宿主语言中的SQL嵌套形式就是在宿主语言程序中使用专门的SQL查询和命令。嵌套SQL另一种形式是自身可以被宿主语言计算的语句。这种语句编译时不可知，因此，不能被SQL预处理器和宿主语言编译器处理。 361

有这样一个例子说明这种情况。有一个程序，它提示用户输入SQL查询，然后读这个查询，并执行这个查询。第6章中假定的针对这种特定的SQL查询的基本界面就是这样的一个程序。每个商用SQL系统均提供了这种类型的基本SQL界面。如果查询是在运行时读取和执行，那么编译时什么都不必做。查询被读到后，将立即进行语法分析，并且由SQL系统寻找合适执行该查询的方式。

宿主语言程序必须指导SQL系统接受刚刚读到的字符串，并将字符串转化为可执行的SQL语句，最后执行这条语句。下面两条动态SQL（Dynamic SQL）语句完成这两步工作：

1. EXEC SQL PREPARE 其后跟随SQL变量*V*、保留字FROM和宿主语言变量或者字符串类型的表达式。这条语句使字符串被当作SQL语句。也许，将对SQL语句进行语法分析且由SQL系统寻找到一个不错的执行该语句的方案。但是，语句并没有被执行，执行该SQL语句的计划变成了*V*的值。

2. EXEC SQL EXECUTE 后面跟随的是如（1）中*V*的一个SQL变量。这条语句引起*V*所代表的SQL语句的执行。

上述两步可以用下面的语句合二为一：

```
EXEC SQL EXECUTE IMMEDIATE
```

跟随其后的是一个字符串型的共享变量或者一个字符串型的表达式。如果一条语句被编译一次，然后执行很多次时，就会看到合并这两步是不利的。使用EXECUTE IMMEDIATE，每次语句执行时都要付出准备该语句的代价，而不是只付出一次。 362

例8.9 图8-7中是一个C程序。该程序从标准输入中读取文本输送到一个变量query中，准备并执行这个查询。SQL变量SQLquery存储准备好的查询。既然查询只执行一次，那么语句：

```
EXEC SQL EXECUTE IMMEDIATE :query
```

可以替代图8-7中的第6行和第7行。 □

```
1)  void readQuery() {

2)      EXEC SQL BEGIN DECLARE SECTION;
3)          char *query;
4)      EXEC SQL END DECLARE SECTION;

5)      /* prompt user for a query, allocate space (e.g.,
           use malloc) and make shared variable :query point
           to the first character of the query */
6)      EXEC SQL PREPARE SQLquery FROM :query;
7)      EXEC SQL EXECUTE SQLquery;
    }
```

图8-7 准备并执行一条动态SQL查询

8.1.11 习题

习题8.1.1 写出下列基于习题5.2.1的数据库模式的嵌套SQL查询

```
Product(maker, model, type)
PC(model, speed, ram, hd, rd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

可以使用你所熟悉的任何宿主语言，宿主语言编程的细节可以用清楚的注释代替。

* a) 询问用户一个价格，找出价格与这个价格最相接近的PC。输出该PC的maker、model和speed。

b) 询问用户能接受的速度、RAM、硬盘大小和显示器尺寸的最小值。找出满足要求的所有手提电脑。输出它们的规格（laptop的所有属性）和它们的制造商。

363

! c) 询问用户一个制造商。输出制造商的所有产品的规格。即输出model number, product-type以及适合这个类型的关系的所有属性。

!! d) 询问用户的“预算”（PC和打印机的总价格）和PC机的最小速度。找出最便宜的“系统”（PC加上打印机）使其在预算之内和满足最小速度，但尽可能使打印机为彩色打印机。输出所选系统的型号。

e) 询问用户制造商、型号、速度、RAM、硬盘的大小、速度和种类或是否为可移动磁盘，以及新PC的价格。如果没有检查到这种型号的PC，输出一个警告，否则将信息插入到表Product和PC中。

*! f) 所有“旧”PC降价$100。确定程序运行时插入的“新”PC没有降价。

习题8.1.2 写出下列基于习题5.2.4的数据库模式的嵌套SQL查询

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

a) 船的火力大约与火炮的数目与火炮口径立方的乘积成比例。找出具有最大火力的类别。

! b) 向用户询问战役的名称。找出该战役中参战船只的所属国。输出沉船最多的国家和损坏船只最多的国家。

c) 向用户询问一个类别的名称和表Classes中元组所要求的其他信息。接着给出那

个类别的船的名字和下水日期的列表。但是用户给出的第一个名字不必是类别的名字。将收集的信息插入到Classes和Ships中。

! d) 检查关系Battles、Outcomes和Ships中在下水前已经参战的船只。当发现错误时，提示用户并提供改变下水日期和战役日期的选项，按要求进行改动。

*! **习题8.1.3** 本习题的目的是找出关系 364

```
PC(model, speed, ram, hd, rd, price)
```

中所有符合条件的PC，要求至少存在两种与其速度相同但价格更贵的PC。虽然有很多方法可以实现，但是在本习题中要使用卷动游标。先按speed再按price的顺序读取PC元组。提示：对于每个元组的读取，向前跳过头两个元组以查看速度是否改变。

8.2 模式中的存储过程

这一节介绍最新的*持久性存储模块*（SQL/PSM 或PSM 或PSM-96）。商业性的DBMS均向用户提供了以数据库模式的形式存储函数或过程的方式，这些函数或过程可用于SQL查询或别的SQL语句中。这些代码使用简单通用的语言编写，可以在数据库中完成不能用SQL查询语言表达的计算。本书描述了SQL/PSM标准。该标准表明了这些功能的主要思想，有助于读者理解与任何特定系统相关的语言。

PSM中定义了模块，这个模块是如下内容的集合，它们是：函数和过程定义、临时关系声明和其他声明。8.3.7节将更进一步讨论模块，这里只讨论PSM的函数和过程。

8.2.1 创建PSM函数和过程

过程声明的主要成分如下：

```
CREATE PROCEDURE <name> (<parameters>)
    local declarations
    procedure body;
```

这种形式与许多编程语言相似。它由过程名、用圆括号括起的参数列表、一些局部变量声明和定义函数代码的执行体组成。函数定义基本上与过程定义的方式相同，除了使用的是保留字FUNCTION和必须指定返回值的类型。函数声明的主要成分如下：

```
CREATE FUNCTION <name> (<parameters>) RETURNS <type>
    local declarations
    function body;
```

过程的参数是模式-名字-类型的三段式，很像4.2.7节提到的ODL方法的参数。也就是，不仅要如同编程语言一样，在参数名后跟随参数所定义的类型，而且要加一个“模式”前缀。 365
该前缀要么是IN，要么是OUT，或者INOUT。这三个关键字分别表明参数是仅输入的、仅输出的、或者即可输入又可输出的。缺省值IN可以省略。

另一方面，函数的参数只可以是IN模式。也就是，PSM阻止了函数中的副作用，所以从函数中得到信息的惟一方式是通过函数的返回值。虽然在过程定义中常常指出IN模式，但是在函数参数中将不指明IN模式。

例8.10 虽然还没有学习过程体和函数体的各种各样语句，但SQL语句的出现并不令人感到意外。对这些语句的限制和8.1.4节介绍的嵌套SQL一样：只允许查询进行单元组选择语句和基于游标的访问。图8-8是接受一新一旧两个地址的PSM过程，每个住在旧地址的影星的

address属性改变为新地址。

```
1)  CREATE PROCEDURE Move(
2)      IN oldAddr VARCHAR[255],
3)      IN newAddr VARCHAR[255]
    )
4)  UPDATE MovieStar
5)  SET address = newAddr
6)  WHERE address = oldAddr;
```

图8-8 改变地址的过程

第1行显示了过程及其名称Move。第2行和第3行包含了两个参数，两个都是输入参数且类型是最大长度为255的变长度字符串。注意，该类型和图6-16中定义的MovieStar的属性address的类型匹配。第4行到第6行是传统的UPDATE语句。参数名可以被当作常数使用。宿主变量在SQL中使用时必须加上前缀冒号（8.1.2节），但是PSM过程和函数中的参数或别的局部变量不要求加冒号。 □

8.2.2 PSM中的简单语句格式

先看如下这些容易掌握的语句格式。

1. *调用语句*：过程调用的形式：

366

```
CALL <procedure name> (<argument list>);
```

也就是，保留字CALL后跟随过程名和用圆括号括起的参数表，这与大多数语言一样。可是，这种调用语句在不同的位置使用不同的形式：

i. 例如，在宿主语言中的调用形式是：

```
EXEC SQL CALL Foo(:x, 3);
```

ii. 作为另一个PSM函数或过程的语句。

iii. 作为发送给基本SQL界面的SQL命令。例如，把

```
CALL Foo(1, 3);
```

这样的语句发送给该界面，并且分别用1和3作为赋值过程的两个参数，执行存储过程Foo。

注意，不允许调用函数。在PSM中调用函数和在C中一样：使用函数名和匹配的参数作为表达式的一部分。

2. *返回语句*：格式如下：

```
RETURN <expression>;
```

该语句只能出现在函数中。它计算表达式的值，并将函数的返回值设置为计算结果。然而，和普通编程语言不同的是，PSM的返回语句不结束这个函数。甚至，继续控制后面的语句，而且在函数完成之前返回值可能被改变。

3. *局部变量定义*：语句格式为

```
DECLARE <name> <type>;
```

用给定的类型声明给定名称的变量。这个变量是局部的，在函数或者过程运行后，DBMS不再保存其值。函数或过程体中的声明必须在可执行语句之前。

4. *赋值语句*：格式如下：

367

```
SET <variable> = <expression>;
```

除了引导保留字SET外，PSM中的赋值和别的语言完全相似。计算等号右边的表达式，并赋值给等号左边的变量。表达式可以是NULL，也可以是查询，只要查询是返回一个单值。

5. *语句组*：语句组以分号结束，并置于保留字BEGIN和END之间。这种构造被当作单个语句，可以出现在单个语句可以出现的任何地方。特别是，由于过程或函数体相当于单个语句，所以在过程和函数体中可插入任何语句序列，只要它们被置于BEGIN和END之间。

6. *语句标号*：8.2.5节将说明为什么有些语句需要标号。用名字（标号名）和冒号作为前缀来标识语句。

8.2.3 分支语句

复杂的PSM语句类型中，先考虑if语句。其形式有点奇怪；与C和其他类似语言的不同是：

1. 用保留字END IF结束

2. 嵌套在If语句的Else子句以单词ELSEIF开始。

因此，if语句的一般格式如图8-9。条件可以是任何boolean类型的表达式，如同SQL语句的where子句。语句列表由以分号结束的语句构成，但不必置于BEGIN...END之间。最后的ELSE和它的语句是可选项。也就是说，要么只有IF...THEN...END IF，要么只带有ELSEIF。

```
IF <condition> THEN
    <statement list>
ELSEIF <condition> THEN
    <statement list>
ELSEIF
    ...
ELSE <statement list>
END IF;
```

图8-9 if语句的格式

368

例8.11 编写一个关于年份y和制片厂s的函数，它返回一个布尔值，其值为true当且仅当制片厂s在第y年至少制作了一部黑白电影，或者在那一年没有制作任何电影。其代码见图8-10。

```
1)  CREATE FUNCTION BandW(y INT, s CHAR[15]) RETURNS BOOLEAN

2)  IF NOT EXISTS(
3)      SELECT * FROM Movie WHERE year = y AND
            studioName = s)
4)  THEN RETURN TRUE;
5)  ELSEIF 1 <=
6)      (SELECT COUNT(*) FROM Movie WHERE year = y AND
            studioName = s AND NOT inColor)
7)  THEN RETURN TRUE;
8)  ELSE RETURN FALSE;
9)  END IF;
```

图8-10 如果制作了电影，则至少有一部是黑白电影

第1行引入了函数和它的参数。参数模式不必指定，因为对于函数只能是IN模式。第2行和第3行判断制片厂s在第y年是否没有任何电影，如果是，则第4行返回值赋为TRUE。注意第4行并没有使函数返回。从技术上讲，正是由if语句规定的数据流引起了从第4行到第9行的跳转，在第9行函数完成并返回。

如果制片厂s在第y年制作了电影，那么第5行和第6行测试是否至少有一部电影不是彩色的。

如果是的话，返回值在第7行再次被置为TRUE。其余情况下，制片厂制作了电影但都是彩色的，则第8行返回值被置为FALSE。 □

8.2.4 PSM中的查询

PSM中有多种select-from-where的查询方式：

1. 子查询可用于条件语句中，或者一般而言，SQL中任何地方使用子查询都是合法的。例如，图8-10中第3行和第6行的子查询。

369 2. 返回单一值的查询可用在赋值语句的右边。

3. PSM中单元组选择语句是合法语句。回忆含有INTO子句的语句，INTO子句将变量赋值为单个的返回元组的组成部分。这些变量可以是局部变量或PSM过程的参数。它的一般形式曾在8.1.5节的嵌套SQL内容中讨论过。

4. 游标的定义和使用，和8.1.6节嵌套SQL的描述差不多。游标的定义及OPEN、FETCH和CLOSE语句等都和原先描述的一样，但是下面几点是不同的：

(a) 语句中不出现EXEC SQL

(b) 局部变量不使用冒号前缀。

```
CREATE PROCEDURE SomeProc(IN studioName CHAR[15])

DECLARE presNetWorth INTEGER;

SELECT netWorth
INTO presNetWorth
FROM Studio, MovieExec
WHERE presC# = cert# AND Studio.name = studioName;
    ...
```

图8-11 PSM中的单元组选择

例8.12 图8-11是用PSM重做图8-3所示的单元组选择，并且是置于假设的过程定义正文中。注意，因为单元组选择只返回一个组成部分的元组，故下面的赋值语句可以起到同样的作用：

```
SET presNetWorth = (SELECT netWorth
    FROM Studio, MovieExec
    WHERE presC# = cert# AND Studio.name = studioName);
```

使用游标的例子被延迟到下一节学习了PSM循环语句之后给出。 □

8.2.5 PSM中的循环

PSM中的基本循环结构如下：

```
LOOP
    <statement list>
END LOOP;
```

370

LOOP语句常常被标识出来，所以可以使用下面的语句中断循环：

```
LEAVE <loop label>;
```

通常情况下，循环涉及用游标读取元组，当没有更多的元组时，就希望离开这个循环。为了表示没有找到元组的SQLSTATE值（回忆一下，是'02000'），可以定义一个条件名。其方法是用如下语句：

```
DECLARE Not_Found CONDITION FOR SQLSTATE '02000';
```

更一般地，可以用如下语句声明任何希望的名字和任意的SQLSTATE值相对应的条件：

```
DECLARE <name> CONDITION FOR SQLSTATE <value>;
```

下面研究一个在PSM中结合使用游标操作和循环的例子。

例8.13 图8-12显示了一个PSM过程，该过程将制片厂名称*s*作为输入参数，并且在输出参数mean和variance给出制片厂*s*拥有的所有电影的长度的平均值和方差。第1行和到第4行声明了过程和参数。

```
 1)  CREATE PROCEDURE MeanVar(
 2)      IN s CHAR[15],
 3)      OUT mean REAL,
 4)      OUT variance REAL
     )
 5)  DECLARE Not_Found CONDITION FOR SQLSTATE '02000';
 6)  DECLARE MovieCursor CURSOR FOR
         SELECT length FROM Movie WHERE studioName = s;
 7)  DECLARE newLength INTEGER;
 8)  DECLARE movieCount INTEGER;

     BEGIN
 9)      SET mean = 0.0;
10)      SET variance = 0.0;
11)      SET movieCount = 0;
12)      OPEN MovieCursor;
13)      movieLoop: LOOP
14)          FETCH MovieCursor INTO newLength;
15)          IF Not_Found THEN LEAVE movieLoop END IF;
16)          SET movieCount = movieCount + 1;
17)          SET mean = mean + newLength;
18)          SET variance = variance + newLength * newLength;
19)      END LOOP;
20)      SET mean = mean/movieCount;
21)      SET variance = variance/movieCount - mean * mean;
22)      CLOSE MovieCursor;
     END;
```

图8-12 某电影制片厂拥有的影片长度的平均值和方差

第5行到第8行是局部声明。定义Not_Found作为条件的名称，该条件表示FETCH在第5行返回读取元组失败。接着，第6行，游标MovieCursor被定义为返回制片厂*s*拥有的电影的长度集合。第7行和第8行声明了所需的两个局部变量。整数newLength保存了FETCH的结果，而movieCount计数制片厂*s*拥有的电影的数目。之所以需要movieCount，是在最后将长度的和转变为长度的平均值，长度的平方和转化为方差。

其余行是过程体。Mean和variance为临时变量，以及最后的“返回”结果。在主循环中，mean实际保存长度的和，variance实际保存长度的平方和。因此，第9行到第11行初始化这些变量和电影的数目为0。第12行打开游标，第13行到第19行形成了标识为movieLoop的循环。

第14行执行读取元组，第15行检查是否找到了另一个元组。如果不是，结束循环。第15行到第18行计算值，将MovieCount加1，长度加到mean上（这里mean实际是计算长度和），长度的平方加到variance上。

当制片厂s的所有电影被检索过后，循环结束，转到第20行。在第20行，用电影的数目除
371 去长度和得到mean的正确值。第21行，电影的数目除去长度的平方和，再减去mean的平方，使variance得到真正的方差。参见习题8.2.4关于为什么这么计算是正确的讨论。第22行关闭游标，过程结束。 □

8.2.6 For循环

PSM中也有for循环结构，不过其惟一重要的目的是：游标的迭代。其语句形式为：

其他的循环结构

PSM中也有while-和repeat-循环，其含义与C相同。也就是说，可以创建如下的形式循环：

```
WHILE <condition> DO
    <statement list>
END WHILE;
```

或者

```
REPEAT
    <statement list>
UNTIL <condition>
END REPEAT;
```

附带地，如果要标识这些循环，包括由loop语句或for语句形成的循环，可以把标识置于END LOOP或其他标识符之后。这样做的优点是使循环结束的位置更清楚，促使PSM解释器能检测到忽略了END的语法错误。

```
FOR <loop name> AS <cursor name> CURSOR FOR
    <query>
372 DO
    <statement list>
END FOR;
```

该语句不但声明了游标，而且处理了许多“麻烦的细节”：打开和关闭游标、取值、检测是否没有元组可以获取。然而，既然不必自己去取元组，所以不能为元组的组成指定存储变量。这样，查询结果的属性名也是由PSM作为相同类型的局部变量处理。

例8.14 使用for循环重编写图8-12的过程。代码见图8-13。很多事情没有变化。图8-13中第1行到第4行的过程声明不变，第5行中局部变量movieCount的声明也一样。

然而，在过程的声明部分中不必声明游标，也不必定义条件Not_Found。第6行到第8行像前面一样初始化这些变量。接着，第9行的for循环，也定义了游标MovieCursor。第11行到第13行是循环体。注意，在第12行和第13行，所涉及的长度是游标用属性名length来检索
373 的，而不是用局部变量名newLength检索，这个版本的过程中不存在newLength。第15行和第16行为输出变量计算正确的值，与前一个版本相同。 □

8.2.7 PSM的异常处理

SQL系统通过在5个字符的字符串变量SQLSTATE设置非零的数字来表明错误条件。前面已经看到这些代码中的一个：’02000’表示“没有找到元组”。另外，’21000’表示“单元组选择返回了多个元组”。

PSM可以定义叫做异常处理（exception handler）的代码，在语句或语句组执行过程中，

当错误代码列表中的任何一个出现在SQLSTATE中时，就调用异常处理。每一个异常处理都和一个由BEGIN...END描述的代码块有关。处理器出现在代码块中，并且仅仅只针对代码块中的语句。

```
1)  CREATE PROCEDURE MeanVar(
2)      IN s CHAR[15],
3)      OUT mean REAL,
4)      OUT variance REAL
    )
5)  DECLARE movieCount INTEGER;

    BEGIN
6)      SET mean = 0.0;
7)      SET variance = 0.0;
8)      SET movieCount = 0;
9)      FOR movieLoop AS MovieCursor CURSOR FOR
            SELECT length FROM Movie WHERE studioName = s;
10)     DO
11)         SET movieCount = movieCount + 1;
12)         SET mean = mean + length;
13)         SET variance = variance + length * length;
14)     END FOR;
15)     SET mean = mean/movieCount;
16)     SET variance = variance/movieCount - mean * mean;
    END;
```

图8-13 用for循环计算平均值和方差

异常处理的组成是：

1. 一组异常条件，当这些条件成立时调用异常处理。
2. 当异常发生时，与该异常相联的执行代码。 374
3. 指明处理器完成处理后的转移去处。

异常处理声明形式如下：

```
DECLARE <where to go> HANDLER FOR <condition list>
    <statement>
```

转移的选择有3种：

a) CONTINUE，表示执行过异常处理声明中的语句之后，继续执行产生异常的语句之后的那条语句。

For循环中为什么需要名字？

注意movieLoop和MovieCursor，它们虽然在图8-13的第9行声明，却从未在这个函数中使用过。但是，仍然不得不为for循环本身和用来迭代的游标起名。原因是PSM解释程序将for循环解释为一个普通的循环，就像图8-12中的代码，代码中需要这两个名字。

b) EXIT，表示执行过异常处理语句后，控制离开声明异常处理的BEGIN...END块。下一步执行该代码块之后的语句。

c) UNDO，与EXIT差不多，只是有一点不同：到目前为止，该块语句的执行对数据库或局部变量的改变被“取消”。也就是，所产生的影响被取消，就好像这些语句没有执行过。

“条件列表”是由逗号分隔的条件的列表，可以是如图8-12的第5行中Not_Found之类被

声明的条件，也可以是SQLSTATE和5位字符串的表达式。

例8.15 编写一个PSM函数，以电影标题作为参数，返回电影的年份。如果该标题的电影不存在或是不止一个，则返回NULL。代码见图8-14。

```
1)  CREATE FUNCTION GetYear(t VARCHAR[255]) RETURNS INTEGER

2)  DECLARE Not_Found CONDITION FOR SQLSTATE '02000';
3)  DECLARE Too_Many CONDITION FOR SQLSTATE '21000';

    BEGIN
4)      DECLARE EXIT HANDLER FOR Not_Found, Too_Many
5)          RETURN NULL;
6)      RETURN (SELECT year FROM Movie WHERE title = t);
    END;
```

图8-14 单元组选择返回的元组个数不为1的例外处理

第2行和第3行声明了符号条件，这些定义不是必须要写的，也可以使用第4行所代表的SQL状态。第4、5、6行是一个代码块，它先为两个条件声明了异常处理：返回零个元组的条
375 件和返回多于一个元组的条件。第5行异常处理动作仅仅是把返回值置为NULL。

第6行的语句做了函数GetYear的工作。它是一个希望正好只返回一个整数的SELECT语句，该整数亦是函数GetYear要返回的。如果对于标题t（函数的输入参数）恰好只有一个，那么就返回这个值。然而，如果第6行发生了异常，要么因为与标题t对应的元组没有，要么因为对应的元组不止一个，那么调用异常处理，且返回值是NULL。既然处理是EXIT类型的，故流程的下一步到达END之后的那一处。因为此处是函数的结束处，此刻GetYear结束，返回NULL值。 □

8.2.8 使用PSM函数和过程

如8.2.2节提到的，可以在嵌套SQL程序、PSM代码本身或提供给类界面的普通SQL命令中调用PSM函数和过程。这些函数和过程的用法和大多数编程语言相同：过程调用用CALL，函数作为表达式的一部分出现。下面给出从特定的界面中调用函数的例子。

例8.16 假定模式中包括了图8-14的GetYear函数。想像面对特定的界面，准备输入Denzel Washington是*Remember the Titans*中的影星这个事实。可是，却忘记了电影的年份。只要这个名称的电影只有一部，并且它就在关系Movie中，那么，就不必预先查询去找出年份。而且，可以将下面的语句插入到这个特定的SQL界面中：

```
INSERT INTO StarsIn(movieTitle, movieYear, starName)
VALUES('Remember the Titans', GetYear('Remember the Titans'),
376     'Denzel Washington');
```

如果具有*Remember the Titans*名称的电影不是一部时，GetYear将返回NULL，那么有可能使该分量插入值是NULL。 □

8.2.9 习题

习题8.2.1 在电影数据库上：

```
Movie(title, year, length, inColor, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

用PSM过程或函数完成下列任务：

* a) 给定电影制片厂的名称，计算其经理的净产值。

* b) 给定名称和地址，如果这个人是影星而不是制片人，返回1，如果是制片人而不是影星，则返回2，如果既是影星又是制片人，返回3，如果既不是影星又不是制片人，返回4。

*! c) 给定制片厂名称，用该制片厂的两部最长的电影标题给输出参数赋值。如果没有这样的电影则参数中的一个或两个赋值为NULL（例如，如果制片厂只有一部电影，则没有“第二长的”电影）。

! d) 给定一位影星的名字，找出他出演的且超过120分钟的最早的电影。如果没有这样的电影，则返回年份0。

e) 给定地址，如果这个地址的影星恰好只有一位时，找出这位惟一影星的名字，如果找到的影星不止一位或没有找到，则返回NULL。

f) 给定影星名字，将其从MovieStar中删除，并从StarIn和Movie中删除他们出演的所有电影。

习题8.2.2 基于习题5.2.1中的数据库模式

```
Product(maker, model, type)
PC(model, speed, ram, hd, rd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

写出下列的PSM过程或函数。

* a) 将价格作为参数，返回价格最接近的PC的型号。

377

b) 将制造商、型号和价格作为参数，返回这种型号的任何类型的产品的价格。

! c) 将型号、速度、ram、硬盘、可移动磁盘和价格信息作为参数，将该信息插入到关系PC中。然而，如果已经有这种型号的PC（假定插入违反键约束时报错，这时SQLSTATE为’23000’），那么型号加1直到找到一个不存在的PC型号。

! d) 给定价格，输出超过这个价格卖出的PC、手提电脑和打印机的数目。

习题8.2.3 基于习题5.2.4的数据库模式

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

写出下列PSM函数或过程。

a) 船的火力大约与火炮的数目与火炮口径立方的乘积成正比。给定一个类别，得出它的火力。

! b) 给定战役的名称，输出该战役的参战船只所属的两个国家。如果所涉及的国家多于或少于两个，则输出时两个国家都为NULL。

c) 将新的类别的名称、类型、国家、火炮的数目、口径和排水量等作为参数。将这些信息插入到Classes而且将带有这个类别名称的船加入到Ships中。

! d) 给定船的名字，判断该船参战日期是否比船下水日期早。如果这样，参战日期和下水的日期都置为0。

习题8.2.4 在图8-12中，使用棘手的公式来计算数字序列x_1，x_2，...，x_n的方差。方差是这些数

字减去其平均值的平方的均值，即，方差是$\left(\sum_{i=1}^{n}(x_i-\bar{x})^2\right)/n$，这里平均值$\bar{x}$是$\left(\sum_{i=1}^{n}x_i\right)/n$。证明图8-12中使用的方差公式

$$\left(\sum_{i=1}^{n}(x_i)^2\right)/n-\left(\left(\sum_{i=1}^{n}x_i\right)/n\right)^2$$

378 产生相同的结果。

8.3 SQL环境

这一节尽可能广泛地查看DBMS和其所支持的数据库与程序。从这里可以看到数据库是如何定义的，如何组成簇、目录和模式，以及程序如何与需要操作的数据连接。由于许多细节依赖于特定的实现，因此这里着重于SQL标准包含的一般思想。8.4节和8.5节阐述了这些高层的概念如何出现在“调用层界面”，它要求程序员作出对数据库显式的连接。

8.3.1 环境

SQL环境是一个框架，该框架下数据可以存在，在数据上的SQL操作可以执行。实际上，SQL环境被当作某些DBMS的运行环境。例如，ABC公司买了Megatron 2002 DBMS的许可证以便在ABC的机器上运行它，于是运行在这些机器上的系统组成了SQL环境。

前面已经讨论了数据库的所有元素——表、视图、触发器、存储过程等等，它们都是在SQL环境中定义的。这些元素组成了层次性结构，每个元素在该结构中扮演了不同的角色。图8-15给出了SQL的标准结构。

简短地说，该组织方式由下面的结构构成：

1. 模式[⊖]。模式是表、视图、断言、触发、PSM模块和其他在书中没有讨论的信息(参见8.3.2节的“更多模式元素”框)的集合。模式是组织的基本单元，可近似地当作“数据库”，不过事实上，在某种程度上比数据库少，下面第（3）点可以看到。

2. 目录。这是模式的集合，是用来支持惟一的、可访问术语的基本单元。每个目录有一个或多个模式，一个目录中的模式名必须是惟一的，每个目录包含一个叫INFORMATION_SCHEMA的特殊模式，这个模式包含了该目录中所有模式的信息。

3. 簇。这是目录的集合。每个用户有一个相联的簇：用户可访问的所有目录的集合（参见8.7节解释如何访问目录和其他受控的元素）。SQL对簇的定义不是很严格，例如，不同用户相
379 联的簇是否可以重叠并没有被标识。簇是查询能发布的最大范围，故在一定程度上，簇是特定用户所看到的“数据库”。

8.3.2 模式

模式声明的最简单形式如下：

1. 保留字CREATE SCHEMA。
2. 模式名称。
3. 模式元素如基本表、视图和断言等的声明列表。

也就是说，模式可以声明为：

```
CREATE SCHEMA <schema name> <element declarations>
```

元素声明采用的是如6.6节、6.7.1节和8.2.1节等不同地方讨论的形式。

⊖ 注意这儿的术语“模式”指的是数据库模式，而不是关系模式。

例8.17 定义一个模式，该模式包括关于本书一直使用的电影例子中的5个关系，加上一些其他已经介绍的元素，如视图。图8-16描述了这样的定义。 □

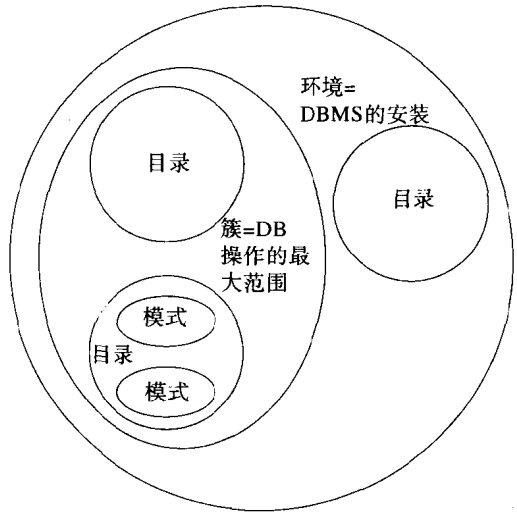


图8-15 环境下数据库模式的组织

不必要一次就将模式声明完全。可以使用合适的CREATE、DROP或ALTER语句来修改或增加模式，例如，CREATE TABLE后跟随模式的一张新表的定义。问题是SQL系统需要知道新表属于哪个模式。如果修改或删除表或其他模式元素，那么由于2个或更多的模式可以有同名的不同元素，故需要消除元素名的歧义。

使用SET SCHEMA语句改变“当前”的模式。例如，

```
SET SCHEMA MovieSchema;
```

将使图8-16描述的模式成为当前模式。于是，任何模式元素的定义被追加到该模式，任何DROP或ALTER语句都是指该模式中存在的元素。

```
CREATE SCHEMA MovieSchema
    CREATE TABLE MovieStar ... as in Fig. 7.5
        Create-table statements for the four other tables
    CREATE VIEW MovieProd ... as in Example 6.48
        Other view declarations
    CREATE ASSERTION RichPres ... as in Example 7.13
```

图8-16 一个模式声明

8.3.3 目录

像表一类的模式元素创建在模式中一样，模式在目录中创建和修改。原则上，希望目录的创建、增加处理与模式的创建、增加处理类似。不巧的是，SQL没有定义一个标准来这么做，如语句

```
CREATE CATALOG <catalog name>
```

使得后面紧跟着的是属于该目录的模式列表和那些模式的声明。

然而，SQL又规定了语句

```
SET CATALOG <catalog name>
```

这条语句允许设置“当前”目录，这样的话，新的模式将进入那个目录，模式修改也是对那个目录中的模式进行，这时可能有名字冲突。并且如果存在名字冲突的话，都是指新加入、修改的模式与当前目录冲突。

更多模式元素

有些没有提到的模式元素偶尔也能使用到，如：

- 域：值或简单数据类型集合。现在很少用到了，因为对象-关系数据库提供了更强大的创建类型的机制，参见9.4节。
- 字符集：符号以及如何编码的方法的集合。ASCII是最著名的字符集，但是SQL工

具可以支持许多其他的集合，如不同的外语集。

- 核对：回忆6.1.3节的字符串按字典序比较，假定任两个字符可以用“小于”关系比较，“小于”被标识为“<”。核对指定哪些字符“小于”另外哪些字符。例如，可以用ASCII隐含的顺序进行比较；或可以把小写和大写字符看做是一样的；不比较非字母的字符。
- 授权语句：它关心的是谁可以访问模式元素。8.7节将讨论授权优先级问题。

8.3.4 SQL环境中的客户和服务器

SQL环境不仅仅是目录和模式的集合。它还包含这样的元素：元素的目的是支持数据库上操作或者是那些目录和模式代表的数据库中的操作。SQL环境有两种特殊的进程：SQL客户和服务器。服务器支持对数据库元素的操作，客户允许用户连接到服务器上对数据库进行操作。可以这样想像：服务器运行在一个大的存储数据库的主机上，客户运行在其他主机上，也许是远离服务器的个人工作站。然而，也可能是客户和服务器都运行在一台主机上。

8.3.5 连接

如果在SQL客户端主机上运行包含了SQL的程序，那么将通过下面的SQL语句打开客户和
382 服务器之间的连接：

模式元素的全名

形式上，模式元素（如表）的名称是它的目录名，它的模式名和它自己的名称，并以这种顺序用点连接表示。因此，目录MovieCatalog中的模式MovieSchema的表Movie的引用如下：

```
MovieCatalog.MovieSchema.Movie
```

如果目录是缺省的或是当前的目录，那么可以省去目录名。如果模式也是缺省的或当前的模式，那么模式部分也可以省去，只留下元素自己的名称是很经常的。然而，当需要访问当前模式或目录以外的元素时，就不得不使用完全名。

```
CONNECT TO <server name> AS <connection name>
    AUTHORIZATION <name and password>
```

服务器名依赖于安装。单词DEFAULT可以替代一个名称且将用户连接到任何被作为“缺省服务器”安装的SQL服务器。授权子句后跟随着用户名和密码。虽然AUTHORIZATION后可以跟随其他的字符串，但密码是服务器识别用户的常见方式。

连接名可以在以后用来引用连接。引用连接的原因是SQL允许用户打开好几个连接，但是任何时候只有一个连接有效。为了切换连接，下面的语句将conn1变成为有效连接：

```
SET CONNECTION conn1;
```

任何当前有效的连接进入休眠状态后，只有显式提到它的另一条SET CONNECTION语句才能激活它。

当断开连接时也要用连接名。断开连接conn1的语句如下：

```
DISCONNECT conn1;
```

现在，conn1被中止。它不是休眠，也不能被激活。

然而，如果连接创建后再也不被引用，那么CONNECT TO子句中的AS和连接名可以省略。完全省略连接语句也是可以的。如果仅仅在SQL客户端的宿主机上执行SQL语句，那么可以为自己建立一个缺省的连接。

383

8.3.6 会话

连接有效时，执行的SQL操作形成了一个*会话*。会话和创建它的连接具有状态同时改变的特点。例如，当连接处于休眠状态时，它的会话也处于休眠态，SET CONNECTION语句对连接的恢复也使会话重新有效。因此，会话和连接是客户和服务器之间链路的两个方面，见图8-17。

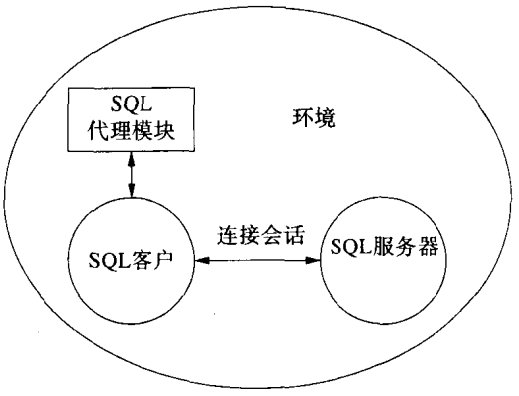


图8-17 SQL客户-服务器的交互图

每个会话有一个当前目录和该目录中的一个当前模式。这由语句SET SCHEMA和SET CATALOG设置，和8.3.2节和8.3.3节中讨论的一样。每个会话也都有一个授权用户，对此将在8.7节讨论。

8.3.7 模块

模块（module）是对应用程序而言的SQL术语。SQL标准建议了三种模块，要求一个SQL的实现中至少提供一种给用户。

1. *基本SQL界面*。用户可以键入SQL服务器执行的SQL语句。这种模式下，每个查询或其他语句本身是一个模块。虽然这种模式实际上很少使用，但是本书中大多数例子都是这种模块。

2. *嵌套SQL*。这种类型已在8.1节讨论过，由EXEC SQL引导的SQL语句出现在宿主语言程序中。预处理器将这个嵌套SQL语句转变为对SQL系统的对应函数或过程的调用。编译后的宿主语言程序（包括这些函数调用）是一个模块。

384

3. *真模块*。SQL设想模块的最一般形式是一个含有存储函数或过程集合的模块，这些函数或过程一部分是宿主语言代码，一部分是SQL语句。它们之间可以通过参数也可以通过共享变量进行通讯。PSM模块（8.2节）就是这样的一个例子。

模块的执行被称为SQL*代理*。图8-17中模块和SQL代理组成一个单元，它调用SQL客户建立与数据库的连接。然而，模块和SQL代理的区别与程序和进程的区别相似：前者是代码，后者是代码的执行。

8.4 使用调用层接口

这一节回到SQL操作和宿主语言程序协调的问题。8.1节中讨论了嵌套SQL；8.2节中讨论了存储在模式中的过程。本节中，采用第三种方法。使用*调用层界面*（CLI）时，只要编写普通的宿主语言代码，并使用可连接和访问数据库的函数库，就可把SQL语句传递到数据库。

这种方法和嵌套SQL编程的区别在某种程度上说是表面上的。如果观察了预处理器对嵌套SQL语句的处理，那么就会发现这些语句被库函数调用所代替，库函数很像标准SQL/CLI中的函数。然而，当SQL被CLI函数直接传递到数据库服务器时，就获得了一定级别的系统独立。也就是说，原则上可以在使用不同DBMS的多个站点运行相同的宿主语言程序。只要这些DBMS接受标准SQL（不妙的是，不总是这种情况），那么相同的代码可以在所有这些站点运行，而不需要特别设计的预处理器。

下面给出调用层界面的两个例子。本节考虑标准SQL/CLI，它是ODBC（开放数据库连接）的变型。8.5节讨论JDBC（Java数据库连接），一个类似的标准，它以面向对象的方式将Java程序连接到数据库。对这两种情况，本书都没有给出全部标准，而只是显示与本书讨论相关的内容。

8.4.1 SQL/CLI简介

用C和SQL/CLI（以后只写CLI）编写的程序包括头文件sqlcli.h，从而得到大量的函数、
385 类型定义、结构和符号常量。这样程序可以创建和处理四种记录（C中称做结构）：

1. *环境记录* 这种类型的记录由应用（客户）程序创建，为对数据库服务器的一个或多个连接做准备。

2. *连接记录* 这种记录创建的目的是连接应用程序和数据库。每个连接记录要存在于某个环境记录中。

3. *语句记录* 应用程序可以创建一个或多个语句记录。每个语句记录保存了一条有关SQL语句的信息，如果语句是一个查询则还包括隐含的游标。不同时刻，同一个的CLI语句代表不同的SQL语句。每条CLI语句存在于某一连接记录中。

> **环境记录和语句记录的内容是什么？**
>
> 我们不审查代表环境和连接的记录的内容。可是，有些有用的信息包含在这些记录的字段中。这些信息一般不是标准的一部分，而是依赖于实现。不过，作为一个例子，环境记录需要指明字符串如何表示，例如，C语言中以'\0'表示结束或定长字符串等。

4. *描述记录* 这种记录保存元组或参数的信息。应用程序或数据库服务器适当地设置描述记录的组成成分，以指定属性或属性值的名称和类型。每条语句都有一些隐式创建的描述记录，用户需要时可创建更多。在CLI的表示中，描述记录一般都是不可见的。

每个记录在应用程序中是用*句柄*来表示，句柄是记录的指针[⊖]。头文件sqlcli.h分别提供了环境记录、连接记录、语句记录和描述记录的句柄类型：SQLHENV、SQLHDBC、SQLHSTMT和SQLHDESC，虽然它们都被看做是指针或整型。除了使用这些类型外，另外还使用其他有明显含义的定义类型，如sqlcli.h中提供的SQLCHAR和SQLINTEGER。

在此不详细讨论如何设置和使用描述记录。然而，其他三种记录的句柄是用如下语句创建：

SQLAllocHandle(*hType,hIn,hOut*)

这里有三个参数：

1. *hType*是所希望的句柄类型。SQL_HANDLE_ENV表示一个新的环境；SQL_HANDLE_DBC表示一个新的连接；SQL_HANDLE_STMT表示一个新的语句。

2. *hIn*是高层元素的处理，该高层元素存放了新近分配的元素。如果要得到一个环境句柄，
386 该参数就是SQL_NULL_HANDLE。这是一个常量，告诉SQLAllocHandle这里没有相关值。如果要得到连接句柄，那么*hIn*是该连接所在环境的句柄。如果要得到语句句柄，那么*hIn*是语句所在连接的句柄。

3. *hOut*是SQLAllocHandle所创建的处理的地址。

SQLAllocHandle返回一个SQLRETURN（一个整数）类型的值。值是0表示没有错误发生，发生错误时返回某一非零值。

⊖ 不要将此处术语“句柄”的用法与8.2.7节讨论的例外处理混淆。

例8.18 图8-4中函数worthRanges曾作为嵌套SQL的例子，现在来考虑该函数如何在CLI中开始的。这个函数检查了MovieExec的所有元组，将他们的净产值分成不同范围。最初的步骤见图8-18。

```
1)  #include sqlcli.h
2)  SQLHENV myEnv;
3)  SQLHDBC myCon;
4)  SQLHSTMT execStat;
5)  SQLRETURN errorCode1, errorCode2, errorCode3;

6)  errorCode1 = SQLAllocHandle(SQL_HANDLE_ENV,
        SQL_NULL_HANDLE, &myEnv);
7)  if(!errorCode1)
8)      errorCode2 = SQLAllocHandle(SQL_HANDLE_DBC,
            myEnv, &myCon);
9)  if(!errorCode2)
10)     errorCode3 = SQLAllocHandle(SQL_HANDLE_STMT,
            myCon, &execStat);
```

图8-18 定义和创建环境、连接和语句记录

第2行到第4行分别声明了环境、连接和语句的句柄。它们的名称分别为myEnv，myCon和execStat。execStat代表SQL语句：

```
SELECT netWorth FROM MovieExec;
```

很像图8-4中游标execCursor所做的，但是目前还没有与execStat相连的SQL语句。第5行声明3个变量，存放函数调用的结果并指明出现的错误。值0表示调用过程中没有出现错误，这正是所希望的情况。 387

第6行调用SQLAllocHandle，需要一个环境处理（第一个参数），第二个参数提供一个空句柄（因为当请求环境句柄时，什么都不需要），提供地址myEnv作为第三个参数，产生的句柄放在此处。如果第6行成功，第7行和第8行用环境句柄来得到连接句柄myCon。假定这个调用也成功，第9行和第10行获得语句句柄execStat。 □

8.4.2 处理语句

图8-18的结尾，句柄是execStat的语句记录已经被创建。不过，还没有该记录相联结的SQL语句。使用语句句柄来联结和执行SQL语句的处理与8.1.10节描述的动态SQL相似。8.1.10节中，使用PREPARE将SQL语句的内容与所谓的“SQL变量”联结，接着使用EXECUTE执行这条语句。

如果将“SQL变量”作为语句句柄，则CLI中的情况十分相似。函数

SQLPrepare(*sh*,*st*,*sl*)

具有：

1. 语句记录句柄*sh*。

2. 指向SQL语句的指针*st*。

3. *St*指向的字符串的长度s_l。如果不知道长度，定义过的常量SQL_NTS将通知SQLPrepare从字符串本身计算出长度。或许，该串是个“以null结束的字符串”，这样SQLPrepare可以扫描这个字符串直至遇到结束符'\0'。

该函数的影响是使得被句柄*sh*引用的语句现在代表特定的SQL语句*st*。

388 另一个函数

SQLExecute(*sh*)

引起句柄*sh*涉及的语句的执行。对于很多形式的SQL语句，如插入和删除，这条语句的执行对数据库的影响是显而易见的。当*sh*涉及到的SQL语句是一个查询语句时其影响就不是那么明显。如8.4.3节所讨论的，该类查询语句有一个隐含的游标，它是语句记录的一部分。语句原则上被执行，所以可以想像所有返回的元组存放在某个位置，等待着被访问。使用隐式游标每次可以读取一个元组，这与8.1节和8.2节使用真实游标所做的差不多。

例8.19 继续图8-18开始的函数worthRanges。查询语句

```
SELECT netWorth FROM MovieExec;
```

与处理execStat涉及语句的联结可以用下面两个函数的调用实现：

```
11)  SQLPrepare(execStat, "SELECT netWorth FROM MovieExec",
         SQL_NTS);
12)  SQLExecute(execStat);
```

它们可以紧接在图8-18的第10行之后。记住SQL_NTS通知SQLPrepare确定它第二个参数所指的以null结束的字符串的长度。 □

与动态SQL相似，使用函数SQLExecDirect可以将准备和执行步骤合二为一。合并上面第11行和第12行的一个例子如下：

```
SQLExecDirect(execStat, "SELECT netWorth FROM MovieExec",
    SQL_NTS);
```

8.4.3 从查询结果中取数据

与嵌套SQL或PSM中FETCH命令相当的函数是

SQLFetch(*sh*)

这里*sh*是一个语句句柄。假定*sh*涉及的语句已经被执行，或者这个取数据操作将产生一个错误。像所有的CLI函数一样，SQLFetch返回一个表明成功或失败的SQLRETURN类型的值。

389 特别应该关注符号常量SQL_NO_DATA代表的返回值，它表示查询结果中不再有元组。如同以前读数据的例子，这个值用来跳出从查询结果中重复取新元组的循环。

然而，如果图8-19中SQLExecute后跟随一个或更多的SQLFetch调用，那么元组是否会出现在此？答案是其组成部分存入了一个描述记录中，该记录与句柄出现在SQLFetch调用中的语句相联结。在每次取数据时，可以通过在开始取数据之前把组成部分绑定到宿主语言变量来抽取相同的组成部分。完成这个工作的函数是：

SQLBindCol(*sh,colNo,colType,pVar,varsize,varInfo*)

这六个参数的意义为：

1. *Sh*是所涉及到的语句的句柄。

2. *ColNo*是要得到获取值的（元组内部）组成部分的数目。

3. *colType*是一个代码，表示存放组成部分值的变量类型。由sqlcli.h提供的代码SQL_CHAR表示字符数组和字符串；SQL_INTEGER表示整型。

4. *pVar*是一个指针，指向存放值的变量。

5. *varSize*是*pVar*指向的变量值的字节长度。

6. *varInfo*是一个整型指针，SQLBindCol用于提供关于产生值的附加信息。

例8.20 用CLI调用代替嵌套SQL，重写图8-4中的整个函数worthRanges。开始时如图8-18，不过为了简明的缘故，省去了所有的错误检查，只保留了测试SQLFetch是否表明目前没有更多的元组。代码见图8-19。

```
1)  #include sqlcli.h
2)  void worthRanges() {

3)      int i, digits, counts[15];
4)      SQLHENV myEnv;
5)      SQLHDBC myCon;
6)      SQLHSTMT execStat;
7)      SQLINTEGER worth, worthInfo;

8)      SQLAllocHandle(SQL_HANDLE_ENV,
            SQL_NULL_HANDLE, &myEnv);
9)      SQLAllocHandle(SQL_HANDLE_DBC, myEnv, &myCon);
10)     SQLAllocHandle(SQL_HANDLE_STMT, myCon, &execStat);
11)     SQLPrepare(execStat,
            "SELECT netWorth FROM MovieExec", SQL_NTS);
12)     SQLExecute(execStat);
13)     SQLBindCol(execStat, 1, SQL_INTEGER, &worth,
            size(worth), &worthInfo);
14)     while(SQLFetch(execStat != SQL_NO_DATA) {
15)         digits = 1;
16)         while((worth /= 10) > 0) digits++;
17)         if(digits <= 14) counts[digits]++;
        }
18)     for(i=0; i<15; i++)
19)         printf("digits = %d: number of execs = %d\n",
                i, counts[i]);
    }
```

图8-19 出品人净产值分组：CLI版本

用SQLGetData抽取组成部分

另外一种将程序变量绑定到查询结果关系输出上的方式是不作任何绑定地读取元组，然后在需要时再将组成部分转化为程序变量。使用的函数是SQLGetData，它的参数与SQLBindCol相同。不过，它只复制一次数据，并且必须在读取数据之后使用，这样才能和开始时就把列绑定到变量产生相同的作用。

图中第3行声明了与嵌套SQL版本中函数使用的局部变量相同的局部变量，第4行到第7行为sqlcli.h提供的类型声明附加的局部变量，这些变量都是与SQL相关的变量。第4行到第6行与图8-18中相同。不同的是第7行新加了worth（对应于图8-4中的同名共享变量）和worthInfo的声明，这是SQLBindCol所要求却没有用到的。

第8行到第10行如图8-18分配所需的句柄，第11行和第12行准备和执行SQL语句，与例8.19所讨论的相同。在第13行，查询结果的第一列（也是惟一的列）被绑定到变量worth。第一个参数是语句所涉及的句柄，第二个参数是涉及到的列，本例中是1。第三个参数是列的类型，第四个参数是一个指针，指向值所放的位置，即变量worth。第五个参数是变量的长度，最后一个参数指向worthInfo，是为SQLBindCol放置附加信息（此处没有用到）的地方。 390

函数的计算与图8-4的第11行和第19行十分相近。While循环开始于图8-19的第14行。注意：

读取一个元组并检验是否超出元组范围的工作，都是在第14行的while循环条件中。如果存在一个元组，那么第15行到第16行确定整数（被绑定为worth）的位数并将合适的计数加1。循环结束后，也就是，第12行语句执行所返回的所有元组已经被检查过，第18行和第19行输出计
391 算的结果。 □

8.4.4 向查询传递参数

嵌套SQL使得SQL语句可以执行，而且语句的一部分可以是由当前共享变量决定的值组成。CLI有相似的能力，但是更复杂。所需的步骤如下：

1. 用SQLPrepare准备一条语句，该语句的某些称为参数的部分用问号取代。第i个问号代表第i个参数。

2. 使用函数SQLBindParameter将值绑定到问号上。这个函数有十个参数，只解释其中所必需的。

3. 通过调用SQLExecute来执行带绑定的查询。注意如果改变了一个或多个参数的值，那么需要再次调用SQLExecute。

下面的例子将解释这个过程，并指出SQLBindParameter所需的重要参数。

例8.21 重新考虑图8-2的嵌套SQL代码，在那个图中得到了两个变量studioName和studioAddr的值，并将它们作为插入到Studio元组的组成部分。图8-20描述了这个过程在CLI中是如何工作的。它假定对于插入语句有一个语句句柄myStat可供使用。

代码从赋给studioName和studioAddr的值（图中没有给出步骤）开始。第1行语句myStat准备一条插入语句，该插入语句是带有两个参数（问号）的VALUE子句。接着，第2行和第3行分别绑定第一个和第二个问号至studioName和studioAddr的当前内容。最后，第4行执行插入语句。如果图8-20中步骤的整个序列，包括没写出的赋给studioName和studioAddr新值的工作，被置于一个循环中，那么每执行一次循环，一个带有制片厂新名称
392 和地址的新元组就被插入到Studio中。 □

```
    /* get values for studioName and studioAddr */

1)  SQLPrepare(myStat,
        "INSERT INTO Studio(name, address) VALUES(?, ?)",
        SQL_NTS);
2)  SQLBindParameter(myStat, 1,..., studioName,...);
3)  SQLBindParameter(myStat, 2,..., studioAddr,...);
4)  SQLExecute(myStat);
```

图8-20 通过将参数绑定到值来插入一个新的studio

8.4.5 习题

习题8.4.1 使用带有CLI调用的C语言重做习题8.1.1。

习题8.4.2 使用带有CLI调用的C语言重做习题8.1.2。

8.5 Java数据库连接

JDBC表示“Java Database Connectivity”，它是一个与CLI类似的软设备，允许Java程序访问SQL数据库。内容与CLI十分相似，只是JDBC中Java的面向对象的特性非常明显。

8.5.1 JDBC简介

使用JDBC首先要做以下工作：

1. 加载将要使用的数据库系统的“驱动器”。这一步可能依赖于安装和实现。然而，结果产生了一个叫做DriverManager的对象。这个对象在很多方面与使用CLI时的环境相近，那时第一步得到的是环境的句柄。

2. 建立与数据库的连接。如果将方法getConnection应用到DriverManager，那么就创建了一个Connection类型的变量。

创建连接的Java语句如下：

```
Connection myCon = DriverManager.getConnection(<URL>,
    <name>, <password>);
```

393

也就是说，方法getConnection将希望连接的数据库的URL、用户名和密码作为参数。它返回Connection类型的对象，对象名称是myCon。注意Java的风格，在一条语句中指定myCon类型和值。

这个连接和CLI连接非常相似，而且它们的目的相同。通过把合适的方法应用到诸如myCon的连接中，可以创建语句对象。将SQL语句“置入”这些对象，然后将值绑定至SQL语句参数。执行SQL语句并且逐个检查元组。既然JDBC和CLI之间的区别更多的在于语法而不是语义，那么只需简要叙述这些步骤。

8.5.2 JDBC中的创建语句

为了创建语句，有两种方法可以施加于一个连接。它们的名字相同，但是参数数量不同：

1. createStatement()返回Statement类型的对象。该对象没有相关的SQL语句，所以方法createStatement()被认为与CLI中SQLAllocHandle的调用相似，该调用接受一个连接句柄，返回一个语句句柄。

2. createStatement(*Q*)返回PreparedStatement类型的对象，此处*Q*是一个作为字符串被传递的SQL查询。因此，JDBC中createStatement(*Q*)的执行和两个CLI步骤的执行相似，这两个CLI步骤是使用SQLAllocHandle得到一个语句句柄，接着在这个句柄和查询*Q*中应用SQLPrepare。

有4种不同的执行SQL语句方法。像上述的方法，它们的区别在于是否接受一个语句作为参数。不过，这些方法在查询和其他SQL语句之间有区别，其他SQL语句统称为“更新”。注意SQL中UPDATE语句仅是JDBC中术语“更新”的一个实例。JDBC中的更新包括所有修改语句，其中有插入语句、所有模式相关语句如CREATE TABLE。这四种“执行”方法是：

a) executeQuery(*Q*)输入参数是查询语句*Q*，用于Statement对象。这种方法返回ResultSet类型的对象，它是查询*Q*得到的元组的集合（准确地说是包）。8.5.3节中将看到如何访问这些元组。

b) executeQuery()被用于PreparedStatement对象。既然准备语句已经含有相关的查询，故没有参数。这种方法也返回ResultSet类型的对象。

c) executeUpdate(*U*)接受非查询语句*U*。当应用到Statement对象时，执行*U*。语句只对数据库产生影响，没有结果集返回。

394

d) executeUpdate()没有参数，用在PreparedStatement中。在这种情况下，执行与准备语句相关的SQL语句。这条SQL语句当然不能是查询语句。

例8.22 假定有一个连接对象myCon，希望执行查询：

```
SELECT netWorth FROM MovieExec;
```

一种方法是创建execStat语句对象，接着用它直接执行查询。结果集被置于Result-

Set类型的对象Worths中，8.5.3节将看到如何对净产值进行平方和处理。完成这个任务的Java代码如下：

```
Statement execStat = myCon.createStatement();
ResultSet Worths = execStat.executeQuery(
    "SELECT netWorth FROM MovieExec");
```

另一种方法是立即准备查询然后再执行查询。与CLI中情形相类似，如果要重复地执行相同的查询，这种方法更可取。它只做一次准备但是多次执行，而不需要DBMS重复准备相同的查询。遵循这种方法的JDBC所需步骤是：

```
PreparedStatement execStat = myCon.createStatement(
    "SELECT netWorth FROM MovieExec");
ResultSet Worths = execStat.executeQuery();
```

□

例8.23 如果要执行一个无参的非查询，那么两种风格的执行步骤相似。不过没有结果集。例如，假定想把如下事件插入到StarsIn中：2000年Denzel Washington出演*Remember the Titans*。可以用下面两种方式中的任一种来创建和使用语句starStat：

```
Statement starStat = myCon.createStatement();
starStat.executeUpdate("INSERT INTO StarsIn VALUES(" +
    "'Remember the Titans', 2000, 'Denzel Washington')");
```

或是

```
PreparedStatement starStat = myCon.createStatement(
    "INSERT INTO StarsIn VALUES('Remember the Titans'," +
    "2000, 'Denzel Washington')");
starStat.executeUpdate();
```

注意，每个Java语句序列都利用了“+”是一个连接字符串的Java算符这一事实。因此，在
395 必要时能够将Java中的SQL语句扩展为多行。 □

8.5.3 JDBC中的游标操作

当执行查询得到一个结果集对象时，可以运行一个游标遍历结果集的元组。为达到这个目的，类ResultSet提供了如下有用的方法：

1. Next()，当将其应用到结果集对象中时，引起隐式游标移向下一个元组（对第一个元组，它是第一次应用）。如果没有下一个元组了，这种方法返回FALSE。

2. getString(*i*)、getInt(*i*)、getFloat(*i*)，以及类似的获取其他类型SQL值的方法，它们每一个都是返回游标所指的当前元组的第*i*个成分。所使用的方法必须适合于第*i*个成分类型。

例8.24 在例8.22中已经得到结果集Worths，可以逐个访问它的元组。由于这些元组只有一个组成成分，且为整型。循环的形式为：

```
while(Worths.next()) {
    worth = Worths.getInt(1);
    /* process this net worth */
}
```

□

8.5.4 参数传递

在CLI中，是使用问号替代查询的部分，然后将值绑定到这些参数。在JDBC中的做法是创

建一个准备语句，用方法setString(*i*, *v*)或setInt(*i*, *v*)将值*v*绑定到查询的第*i*个参数上，而且值*v*必须是方法中一种合适的类型。

例8.25 模仿例8.21的CLI代码，在那里准备了一条语句将新的制片厂插入到关系Studio中，该语句带有表示那个制片厂名称和地址的值的参数。准备这条语句、设置参数的Java代码，见图8-21。继续假定连接对象myCon有效。

第1行和第2行，创建和准备插入语句。该语句有表示被插入的每个值的参数。第2行以后，开始了一个循环，该循环重复地向用户索取制片厂的名称和地址，并将这些字符串置于变量studioName和studioAddr中。第3行和第4行分别将第一个和第二个参数指向保存studioName和studioAddr当前值的字符串。最后，第5行用参数的当前值执行插入语句。第5行之后，从含有注释的步骤再次开始循环。 □ 396

```
1)  PreparedStatement studioStat = myCon.createStatement(
2)      "INSERT INTO Studio(name, address) VALUES(?, ?)");
    /* get values for variables studioName and studioAddr
       from the user */
3)  studioStat.setString(1, studioName);
4)  studioStat.setString(2, studioAddr);
5)  studioStat.executeUpdate();
```

图8-21 在JDBC中设置和使用参数

8.5.5 习题

习题8.5.1 用带JDBC的Java代码重做习题8.1.1。

习题8.5.2 用带JDBC的Java代码重做习题8.1.2。

8.6 SQL中的事务

目前，在数据库上的操作模型是用户查询或修改数据库。这样，数据库上的操作一次只执行一个，一个操作留下的数据库状态正是下一个操作所要起用的。进一步，我们假定操作的执行是个实体（“原子性地”），也就是说操作过程中硬件和软件都不会出错，不会留下操作的结果不能解释的数据库状态。

实际情况比这更复杂。首先应考虑是什么导致数据库处于这样一种不能反映在其上执行的操作的状态，接着考虑SQL提供给用户的工具，以确保不会出现这些问题。

8.6.1 可串行性

象银行业务或机票预定这样的应用，每秒钟都会有上百个操作在数据库上执行。这些操作从成千上百的地方启动，如自动出纳机或旅行社的台式机、航空公司雇员或旅客自己。完全可能有两个操作影响同一个账目或航班，并且这些操作时间可能重叠。如果这样的话，它们会以古怪的方式相互影响。例如：如果DBMS完全没有约束对数据库操作的顺序，那么将会出现错误，下面给出了一个这样的例子。要强调的是数据库系统一般不会以这种方式运转，要使商用数据库系统发生这种错误还得费些周折。 397

例8.26 假定用嵌套SQL的C语言写一个函数chooseSeat()，读取关于有效航班和座位的关系，确定特定的座位是否有效，如果有效则占用这个座位。在其上进行操作的关系是Flights，它有属性fltNum、fltDate、fltSeat和occ，它们的意义很明显。选择座位的程序见图8-22。

图8-22的第9行到第11行是一个单元组选择，根据指定的座位占有与否来设置共享变量occ为真或假（1或0）。第12行检测座位是否被占用，如果没有，这个座位更新为被占用过。更新

398 由第13行到第15行完成，而第16行将这个座位分配给顾客。实际上，很可能将座位分配信息存储在另一个关系中。最后，在第17行，如果座位被占用了，则告知顾客。

```
1)  EXEC SQL BEGIN DECLARE SECTION;
2)      int flight; /* flight number */
3)      char date[10]; /* flight date in SQL format */
4)      char seat[3]; /* two digits and a letter represents
                         a seat */
5)      int occ; /* boolean to tell if seat is occupied */
6)  EXEC SQL END DECLARE SECTION;

7)  void chooseSeat() {
8)      /* C code to prompt the user to enter a flight,
           date, and seat and store these in the three
           variables with those names */
9)      EXEC SQL SELECT occupied INTO :occ
10)             FROM Flights
11)             WHERE fltNum = :flight AND fltDate = :date
                      AND fltSeat = :seat;
12)     if (!occ) {
13)         EXEC SQL UPDATE Flights
14)                  SET occupied = TRUE
15)                  WHERE fltNum = :flight
                         AND fltDate = :date
                         AND fltSeat = :seat;
16)         /* C and SQL code to record the seat assignment
               and inform the user of the assignment */
        }
17)     else /* C code to notify user of unavailability and
                ask for another seat selection */
    }
```

图8-22 选择座位

现在，记得函数chooseSeat()可以被2个或多个顾客同时执行。假设很巧合地，有两个代理程序几乎在同一时刻试图预定同一日期和同一航班的同一个座位，如图8-23。它们同时到达第九行，它们的局部变量occ的副本都是0值，也就是，座位目前没有被分配。在第12行，chooseSeat()的每次执行使得occ更新为TRUE，即占用该座位。这些更新可能一个接一个地执行，每次执行都在第16行告诉顾客这个座位属于他们。 □

在例8.26中，可以想像这两个操作都可以正确地执行，但是从全局看结果不对：两个顾客都相信自己得到了这个有争议的座位。这个问题可以用一些SQL机制来解决，这些机制将函数的执行串行化。如果一个函数在别的函数开始之前执行完毕，则说作用在同一个数据库上的函数的执行是串行的。如果函数表现得好像串行运行的一样，那么就说该执行是可串行化的，虽然函数的执行有可能在时间上重叠。

很清楚，如果chooseSeat()的两个引用是串行的（或是可串行化的），那么我们前面看到的错误就不会发生。一个顾客的启用先发生，这个顾客看到一个空的座位并预定它。接着其他顾客的启用开始且看到该座位已经被占用。

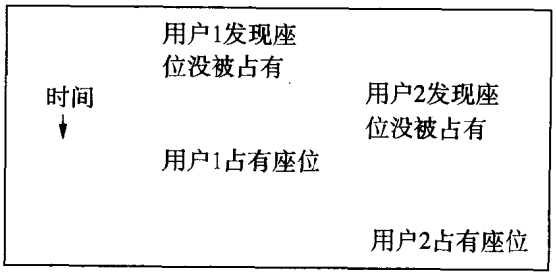


图8-23 两个顾客试图同时预定同一座位

对顾客而言谁占了座位可能是要紧的，但是对数据库而言最重要的是一个座位只能被分配一次。

8.6.2 原子性

如果两个或多个数据库操作在同一时间执行，可能出现不可串行化的行为，除此之外，如果在一个操作执行过程中出现硬件或软件“崩溃”，这个操作有可能使那个数据库处于不可接受的状态。例8.26将表明会发生些什么。要记住实际的数据库系统不允许出现这种错误。 399

保证串行化行为

实际上要求操作连续运行是不可能的：操作数目太多并且存在一定程度的并行。因此，DBMS采用了一种机制来保证串行化行为。即使操作不是连续的，对用户而言结果看起来好像操作是在连续执行。

对于DBMS，一个普通的方式是锁定数据库的元素防止两个函数同时访问这些元素。1.2.4节提供了锁技术，这个思想在18.3节将被广泛的提到。但是这个思想的深入讨论超出了本书范围。例如，如果例8.26中的函数chooseSeat()锁定了Flights关系以外的其他操作，那么不访问Flights的操作可以和chooseSeat的调用并行，但是调用chooseSeat的其他操作不可以运行。

例8.27 现在来看另一个通常的数据库：银行的账目记录。用带属性acctNo和balance的关系Accounts来表示这种情况。该关系中账号和账户余额是成对的。

希望写一个函数transfer()，该函数读取两个账户和一定数额的款项，检查第一个账户存储的款项是否大于此数额，如果是，将钱如数从第一个账户转移到第二个账户。函数transfer见图8-24。

```
1)  EXEC SQL BEGIN DECLARE SECTION;
2)      int acct1, acct2; /* the two accounts */
3)      int balance1; /* the amount of money in the
                         first account */
4)      int amount; /* the amount of money to transfer */
5)  EXEC SQL END DECLARE SECTION;

6)  void transfer() {
7)      /* C code to prompt the user to enter accounts
           1 and 2 and an amount of money to transfer,
           in variables acct1, acct2, and amount */
8)      EXEC SQL SELECT balance INTO :balance1
9)               FROM Accounts
10)              WHERE acctNo = :acct1;
11)     if (balance1 >= amount) {
12)         EXEC SQL UPDATE Accounts
13)                  SET balance = balance + :amount
14)                  WHERE acctNo = :acct2;
15)         EXEC SQL UPDATE Accounts
16)                  SET balance = balance - :amount
17)                  WHERE acctNo = :acct1;
        }
18)     else /* C code to print a message that there were
                insufficient funds to make the transfer */
    }
```

图8-24 从一个账户向另一个账户转账

图8-24的工作是直接的。第8行到第10行查询第一个账户的余额。在第11行，判断余额是

否足够提取所希望数额的钱。如果是的话，那么第12行到第14行将这个钱数加到第二个账户上，第15到第17行从第一个账户提取出同样多的钱。如果第一个账户的余额不够，那么不发生转移并且在第18行输出一个警告。

现在，考虑如果第14行之后失败了会如何。也许是计算机失效，也许是连接数据库和正在进行转移的处理机的网络失效。然后，数据库处于这样的状态：钱已经转移到第二个账户，但还没有改动第一个账户。银行实际上损失了那些要转移的钱。 □

400 例8.27说明的问题是数据库操作的某些结合（如图8-24的两个更新）需要原子地完成的，即，要么都做要么都不做。例如，一个简单的解决方案是将对数据库的所有改变都在一个本地工作区中完成，而且只有在这些工作都完成后才对数据库提交，于是这些改变成为数据库的一部分，并且对其他操作可见。

8.6.3 事务

8.6.1节和8.6.2节提出的对可串行化和原子性问题的解决方案将把数据库操作分组成事务(transaction)。事务是一个或多个必须被原子地执行的数据库操作的集合，也就是说每个操作要么都执行要么都不被执行。另外，SQL要求事务缺省地以可串行化方式执行。DBMS允许用
401 户对两个或多个事务的操作交叉指定一个不那么严格的约束条件。

使用基本SQL界面时，每条语句其自身都是一个事务[⊖]。不过，当编写带有嵌套SQL或带有SQL/CLI、JDBC的代码时，通常要显式地控制事务。当查询以及操纵数据库或模式的任何语句开始时，事务自动开始。SQL命令START TRANSACTION也可以在需要时使用。

在基本界面中，除非事务用START TRANSACTION命令启动，否则均以语句结束。在所有其他的情况中，有两种方式可以结束一个事务：

1. SQL语句COMMIT导致事务成功地结束。由这条SQL语句或从当前事务开始以来的语句组引起的、对数据库的任何改变都被永远建立在数据库中（即，它们被提交了）。在COMMIT事务执行之前，改变是试探性的或者说对其他事务不可见。

2. SQL语句ROLLBACK导致事务夭折或不成功结束。任何由该对事务的SQL语句所引起的改变都被撤消（即它们被回滚），故它们不再出现在数据库中。

上述观点有一个例外。如果试图提交一个事务，但是需要延迟约束检查（见7.1.6节），且这些约束现在不满足，那么即使用COMMIT语句告诉事务，事务也不提交。甚至会造成事务回滚，并且SQLSTATE中有一个指示告诉应用程序事务因为这个原因而夭折。

事务中数据库如何变化

不同的系统可以采用不同的方法实现事务。事务执行时，它可能引起数据库的变化。如果事务夭折，那么（如果没有预防）这些变化可能已经被其他的事务看到。对数据库系统而言，最普通的方式是锁定被改变的项目直到选择了COMMIT或者ROLLBACK，这样就阻止了其他事务看到这些暂时的变化。如果用户想让事务以串行化方式运行，那么一定使用了锁或者其等效物。

但是，如在8.6.4节看到的，SQL提供了关于数据库暂时变化的几个选项。被改变的数据有可能没有被加锁并且是可见的，即使随后的回滚使变化消失。它是由事务的程序

⊖ 然而，很多由语句唤醒的触发器也是这同一个事务的一部分。一些系统也允许触发器唤醒其他的触发器，如果是这样，所有这些动作也是该事务的一部分。

设计者来决定是否需要避免暂时变化的可见性。有鉴于此，所有SQL的实现都提供一种方法，如锁技术，以保持数据库的变化在提交前不可见。

例8.28 假设要以单个事务的形式执行图8-24中的函数transfer()。事务从第8行读取第一个账户的余额时开始。如果第11行的测试为真，完成资金的转移，接着提交所做的改变。这样，在第12行到第17行if分程序块的末尾附加上一条SQL语句：

```
EXEC SQL COMMIT;
```

如果第11行的测试为假——也就是，没有足够的资金可供转移——那么可以选择夭折事务。在第18行else分程序块的末尾附加完成该工作的语句

```
EXEC SQL ROLLBACK;
```

402

实际上，既然在这个转移中没有执行数据库修改语句，无论是提交还是夭折事务都无关紧要，因为没有改变被提交。 □

8.6.4 只读事务

例8.26和例8.27都包含了一个先读然后写数据到数据库中的事务。这种事务容易产生串行化问题。因此在例8.26中看到，如果两个操作试图在同一个时刻预定同一个座位会发生什么情况，在例8.27中又看到，如果在函数执行过程中出现崩溃，那么会发生什么情况。然而，当一个事务只读取而不写数据时，就可以更自由地让事务与别的事务并行执行[⊖]。

应用产生的回滚与系统产生的回滚

在讨论事务时，假定对事务提交还是回滚的判定是产生此事务的应用程序的一部分。也就是说，如同例8.30和8.28，一个事务可以完成很多数据库操作，接着通过发布COMMIT来决定导致永久变化，或是通过发布ROLLBACK返回到初始状态。然而，系统也可以执行事务回滚，在其他的并发事务或系统故障出现时遵照它们特定的独立层次自动执行。通常，如果系统夭折一个事务那么会产生一个特殊的错误代码或异常。如果应用程序希望确保事务成功执行，就必须捕捉这些条件（如通过SQLSTATE的值）并且重启这个有问题的事务。

例8.29 假定写了一个读数据来判断某个座位是否有效的程序；这个函数的动作如图8-22的第1行到第11行。可以立即执行这个函数的多次调用，不用担心会对数据库造成永久伤害。最糟的情形是，在读取某个有效座位时，别的函数正在预定或释放这个座位。因此，依赖于执行该查询时的微观区别，可能得到答案是“有效”或“被占用”，但是这个答案在某个时刻将具有确定的意义。 □

403

如果SQL执行系统被告知当前的事务是只读的，即，它不会改变数据库，那么SQL系统能够充分利用这一点。一般地，许多访问同一数据的只读事务可以并发地运行，但是写同一数据的事务不可以。

通过以下语句告知SQL系统下面的事务是只读事务：

```
SET TRANSACTION READ ONLY;
```

⊖ 一方面在事务，另一方面在游标管理之间有个比较。例如，注意到在8.1.8节使用只读游标比一般游标能得到更高的并行性。同样，只读事务能并行；而读写事务禁止并行。

这条语句必须在事务开始之前执行。例如，如果有个函数由图8-22的第1行到第11行组成，那么通过在事务开始处第九行之前添加下面的语句，可以声明其是只读事务：

```
EXEC SQL SET TRANSACTION READ ONLY;
```

如果在第9行后面的话就太晚了，会导致事务不是只读事务。

可以通过语句

```
SET TRANSACTION READ WRITE;
```

404 通知SQL即将开始的事务可以写数据。不过，这个选项是缺省选项，因此不是必须的。

8.6.5 读脏数据

脏数据（dirty data）是表示未提交的事务所写的数据的通用术语。*脏读取*（dirty read）是对脏数据的读取。读脏数据的风险是写数据的事务最终可能被放弃。如果这样，那么脏数据将从数据库中移走，就像这个数据不曾存在过。如果别的事务读取了这个脏数据，然后该事务可能用提交或别的措施反映它对脏数据的计算结果。

读脏数据有时会带来问题，有时却无关紧要。当读脏数据带来的影响足够小时，偶尔读一次脏数据也并非不可以，由此还可以避免：

1. DBMS用来预防读脏数据所做的费时的工作。
2. 因等待而未去读脏数据造成的并发性丢失。

下面例子表明当允许读脏数据时将会发生什么情况。

例8.30 再次考虑例8.27的账户转移。不过，假定转移由程序*P*实现，*P*执行下面一系列步骤：

1. 将这些钱加到第二个账户。
2. 检测账户1是否有足够的钱。

(a) 如果没有足够的钱，将这些钱从第二个账户减去并结束。[⊖]

(b) 如果有足够的钱，从第一个账户减去这些钱并结束。

如果程序*P*是串行执行，那么把钱临时放入第二个账户是无关紧要的。没人看到这笔钱，当转移不成功时将被减去。

不过，假定有可能读脏数据。假想有三个账户：*A*1、*A*2和*A*3，各有$100、$200、$300。假设事务*T*1执行程序*P*将$150从*A*1转移到*A*2。大概在同一时间，事务*T*2运行程序*P*将$250从*A*2转移到*A*3。这是一个可能的事务顺序：

405 1. T_2执行步骤1将$250加到*A*3，现在*A*3有$550。
2. T_1执行步骤1将$150加到*A*2，现在*A*2有$350。
3. T_2执行步骤2的测试，发现*A*2有足够的资金（$350）允许从*A*2到*A*3的$250转移。
4. T_1执行步骤2的测试，发现*A*1没有足够的资金（$100），不允许从*A*1到*A*2的$150转移。
5. T_2执行步骤2b。从*A*2中减去$250，*A*2现在有$100，并结束。
6. T_1执行步骤2a。从*A*2中减去$150，*A*2现在有-$50，并结束。

钱的总数没变。这三个账户中总共仍有$600。但是因为*T*2在上述6个步骤中的第三个步骤中读取了脏数据，故不能阻止一个账户变为负值，而本来的目的是希望检测第一个账户是否有足够的资金。 □

⊖ 读者应知道程序P试图完成比DBMS更常见的功能。特别地，像P在这一步所做的那样，当P决定不完成这个事务时，它将发出一个回滚（夭折）命令给DBMS，让DBMS消除P执行造成的影响。

例8.31 假想例8.26中的座位选择函数有个变化。在新的方法中：

1. 发现一个有效座位，通过设置occ为TRUE来取得这个座位。如果没有有效座位，则夭折。

2. 询问顾客是否要这个座位。如果是，提交。如果否，通过设置occ为FALSE来放弃这个座位，重复步骤1取得另一个座位。

如果两个事务几乎同时执行这个算法，一个事务可能取得一个座位*S*，*S*后来被顾客拒绝。如果在座位*S*被标记为占用时第二个事务执行步骤1，第二位顾客不能得到座位*S*的选择权。

例8.30中的问题是读了脏数据。第二个事务看到一个由第一个事务所写的元组（*S*被标记为占用），该元组后来又被第一个事务修改。 □

读到了脏数据会怎样？在例8.30中，它引起账户变为负值，尽管显而易见这种情况禁止发生。在例8.31中，问题看起来没这么严重。事实上，第二个顾客可能没得到他喜欢的座位，或者甚至被告知没有座位了。不过，后一种情况中，再次运行事务很有可能就显示了座位*S*的有效性。用允许读脏数据的方式实现座位选择函数是有意义的，可以加速预定要求的平均处理时间。

SQL允许明确指定一个给定的事务可以读脏数据。使用8.6.4节讨论的SET TRANSACTION语句。对于像例8.31中描述的事务而言，合适的形式为：

406

```
1)  SET TRANSACTION READ WRITE
2)      ISOLATION LEVEL READ UNCOMMITTED;
```

上述语句做了两件事：

1. 第1行声明事务可以写数据。

2. 第2行声明事务在读不提交（read-uncommitted）的“隔离级别”运行。即，允许事务读脏数据。8.6.6节将讨论四种隔离级别。目前为止，已经学习了两种：串行化和读不提交。

运行在不同隔离级别的事务之间的交互

微妙之处在于事务的隔离级别只影响事务可以看到的那些数据，不影响其他事务所看到的。作为佐证，如果事务T正在串行化级别运行，那么T的执行必须看起来好像所有其他事务的执行要么完全在T之前要么完全在T之后。但是，如果一些事务正运行在其他的隔离级别上，那么他们可以看到T所写的数据。如果他们运行在读不提交隔离级别，他们可以看到来自T的脏数据，且T夭折。

注意，如果事务不是只读的（即，有可能修改数据库），且指定了独立层次READ UNCOMMITED，那么也必须指定READ WRITE。回忆8.6.4节的缺省假设是事务是读写的。不过，SQL对于允许读脏数据时有一个例外。那么，缺省假设是只读，因为如已经看到的那样，带有读脏数据的读写事务有很大的风险。如果让读写事务在读不提交的隔离级别上运行，那么需要像上述那样显式地指定READ WRITE。

8.6.6 其他隔离级别

SQL一共提供了四种隔离级别。有两种已经看到了：串行化和读不提交（允许读脏数据时）。其余两种是读提交和重复读。它们可分别通过下面语句指明：

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

或

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

对于每条语句，事务缺省是读写的，所以在适当的情况下，可在每条语句后面加上READ ONLY。另外，指定时也可以选择

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

但是，这是SQL的缺省情况，不必显式指明。

读提交禁止读取脏数据（没有提交的数据），如它的名字所示。但是，它却允许一个事务发
407 布同一个查询若干次并得到不同的答案，前提是只要答案反映了已经提交的事务所写的数据。

例8.32 重新考虑例8.31的座位选择函数，但是假定声明函数运行在读提交隔离级别。那么在第一步寻找座位时，如果另外的事务正在预约座位而没有提交，那么它不会将这些座位看做是预定过的。[⊖]。但是，如果旅客拒绝了座位，并且多次执行查询有效座位，那么每次查询时可以看到不同的有效座位集，因为和这个事务并发执行的其他事务可以成功地预定或取消座位。 □

现在，考虑重复读隔离级别。这个术语有点像写错了名字，因为不止一次发布的同一个查询不能保证得到相同的答案。在重复读下，如果元组是第一次被检索到，那么可以确信重复这个查询时指定的元组再次被检索到。不过，执行同一个查询的第二次查询或子查询也有可能检索到幻象元组（phantom tuple）。后者是当该事务执行时插入到数据库中的结果元组。

例8.33 继续例8.31和例8.32中的座位选择问题。如果在重复读隔离级别下执行这个函数，那么在第一次查询第一步时有效的座位在子查询中仍保持有效性。

但是，假定关系Flights有了新的元组。例如航线可能将航班转为较大的飞机，创建了以前不存在的新元组。于是在重复读隔离级别下，查询有效座位的子查询也可能检索到这些新
408 座位。 □

8.6.7 习题

习题8.6.1 本题以及下一题的程序都涉及到对下面两个关系的操作

```
Product(maker, model, type)
PC(model, speed, ram, hd, rd, price)
```

使用嵌入的SQL和恰当的宿主语言，简要写出下列程序。记住在恰当的时候使用COMMIT和ROLLBACK语句，如果你的事务是只读的，要告诉系统。

a) 给定speed和RAM的容量（作为函数的参数），查询有此速度和RAM的PC，打印出model和price。

* b) 给定一个model值，从PC和Product中删除model值与给定值相同的元组。

c) 给定model值，将有此model值的PC的price值减少$100。

d) 给定maker、model、处理器速度、RAM的大小、硬盘的大小、可移动磁盘的类型以及价格，检查是否有model值为给定值的product。如果有这样的model，为用户打印一条出错信息。如果没有，将关于此model的信息输入PC和Product表。

! **习题8.6.2** 对于上题中的每个程序，讨论原子性问题，在程序执行过程中会出现系统崩

⊖ 真正发生了什么看起来有些神秘，因为人们不能确定实施不同隔离级别的算法。很可能，如果两个事务都看到一个座位有效且试图预定，那么系统必须强迫回滚一个事务以避免死锁（见8.6.3节的方框“应用引起的回滚以及系统引起的回滚”）。

溃吗?

! **习题8.6.3** 假设执行习题8.6.1四个程序中的一个作为事务*T*，同一时间也可以执行同一个或不同的程序作为其他事务。如果所有的事务都在读不提交隔离级别运行（如果它们都在可串行化隔离级别运行，这种情况就不可能出现了），事务*T*的运行状况如何？分别考虑事务*T*是习题8.6.1中的任何一个。

*!! **习题8.6.4** 假设有一个事务*T*它是“永久”运行的函数，每小时检查一次是否有速度为1500或更高、售价在$1000以下的PC机。如果找到，打印出相关信息并终止。在此期间，可能运行了习题8.6.1描述的四个程序。说出在每一个隔离级别——可串行化、重复读、读提交、以及读未提交——下的事务*T*的效果。 409

8.7 SQL中的安全机制和用户认证

SQL假定授权ID的存在，这些ID基本上都是用户名。SQL也有一个特殊的授权ID，PUBLIC，它包含了所有用户。授权ID可以被授予权限，很像操作系统里文件系统维护中的授权。例如，UNIX系统一般可以控制三种权限：read，write，以及execute。这些权限有意义，是因为UNIX系统中受保护的对象是文件，这三种操作已囊括了可以对文件所作的事。但是数据库比文件系统要复杂的多，相应的SQL中权限的种类也要复杂得多。

8.7.1 权限

SQL中定义了九种类型的权限：SELECT、INSERT、DELETE、UPDATE、REFERENCE、USAGE、TRIGGER、EXECUTE、UNDER。前四个应用到关系上，这里关系可以是表或视图。正如这四种权限的名字所示，它们分别赋予了权限拥有者查询关系、往关系中插入数据、删除关系中的数据以及更新关系中的元组这些权力。

一个包含了一条SQL语句的模块如果没有相应的权限是不能被执行的。例如，一个select-from-where语句要求对它访问的每个表有SELECT权限。在后面可以看到模块如何获得这些权限。SELECT、INSERT以及UPDATE也可以有一些相关的属性，例如，SELECT(name,addr)。这样的话，只有这些属性可以在查询中看到，在做插入的时候也只有这些属性可以被说明，或者修改时只能修改这些属性。注意，授权的时候，这些权限会和一个特定的关系相联结，此时关系属性name和addr属于什么关系是清楚的。

关系上的REFERENCE权限是指在完整性约束下引用关系的权力。这些约束可以使用第7章提到的所有形式，像断言、基于属性或元组的约束或引用完整性约束。REFERENCE权限也有一些附加的属性，此时，只有这些属性在约束中可以被引用。一个约束只有在它所在模式的拥有者拥有它涉及的所有数据的REFERENCE权限时才能被检查。

USAGE权限主要应用在多种模式元素上而不是关系和断言上（见8.3.2）。它给出了在声明中使用这些元素的权利。关系上的TRIGGER权限是定义这个关系上的触发器的权力。 410
EXECUTE是执行如PSM过程或函数之类的代码的权力。最后，UNDER是创建给定类型的子类型的权力。这个问题将延迟到第9章，那时开始讨论SQL面向对象的细节。

触发器和权限

触发器如何处理权限有一点微妙。首先，如果你拥有一个关系的TRIGGER权限，那么可以试图创建这个关系的任何触发器。不过，既然触发器的条件和动作部分与数据库的查询和/或修改部分很相似，触发器的创建者必须拥有这些操作的基本权限。当有人执

行唤醒触发器的操作时，他不需要具备触发器条件和动作所要求的权限。触发器是在其创建者的权限下执行。

例8.34 考虑执行图6-15的插入语句所需要的权限，该语句再次在图8-25中生成。首先是插入到关系Studio中的语句，所以需要Studio的INSERT权限。但是，既然插入指定的只是属性name的成分，那么拥有关系Studio的INSERT权限或INSERT（name）权限都是可以的。第二个权限仅仅允许插入指定了成分name的Studio元组，这些元组的其他成分接受的是缺省值或NULL，如图8-25。

```
1) INSERT INTO Studio(name)
2)     SELECT DISTINCT studioName
3)     FROM Movie
4)     WHERE studioName NOT IN
5)         (SELECT name
6)          FROM Studio);
```

图8-25 加入新制片厂

但是，注意到图8-25的插入语句还包括两个子查询，这两个子查询分别开始于第2行和第5行。为了执行这些选择语句，子查询要求必需的权限。因此，包含在FROM子句中的两个关系Movie和Studio需要SELECT权限。要注意的是，仅仅因为拥有Studio的INSERT权限并不
411 意味着拥有Studio的SELECT权限，反之亦然。既然选择的只是Movie和Studio的特殊属性，那么拥有Movie的SELECT（studioname）权限和Studio的SELECT（name）权限或者拥有包含这些属性的属性列表的权限就足够了。 □

8.7.2 创建权限

前面已经看到SQL权限是什么，并且也知道了SQL操作的执行必须要有相应的权限。现在学习如何取得执行一个操作的必要权限。取得权限有两个方面：最初是如何创建的，以及如何从一个用户传递到另一个用户。这里只讨论初始化，权限的传递将在8.7.4节讨论。

首先，SQL元组如模式或模块都有一个属主。属主拥有其所属事物的所有权限。SQL中建立属主身份的三种情况是：

1. 假定模式创建时，模式和所有的表，以及该模式中其他的模式元素的所有权都属于创建这个模式的用户。这样这个用户拥有模式元素的所有可能的权限。

2. 会话被CONNECT语句初始化时，有机会用AUTHORIZATION子句指定用户。例如，连接语句

```
CONNECT TO Starfleet-sql-server AS conn1
    AUTHORIZATION kirk;
```

表示用户kirk创建了一个连接到名字为Starfleet-sql-server 的SQL服务器的链路conn1。也许，SQL的实现将验证用户名是有效的，例如通过询问密码。也可以将密码包含在AUTHORIZATION子句中，如8.3.5节讨论的那样。这种方式有点不安全，因为密码可以被别人从Kirk的背后看到。

3. 模块创建时，可通过AUTHORIZATION子句选择其属主。例如，模块创建语句中的子句

```
AUTHORIZATION picard;
```

使得用户picard成为该模块的属主。模块也可以不指定属主，这种情况下模块公开执行，

执行模块中的任何操作所必需的权限必须从别处取得，例如在模块执行过程中连接和会话与用户的联结。 412

8.7.3 检查权限的处理

如上所述，每个模块、模式和会话有一个相关用户。用SQL术语来说就是，每个都有一个相关的授权ID。任何SQL操作包括两部分：

1. 数据库元素，操作将在其上执行。
2. 产生操作的代理。

对代理有效的权限来自一个叫做当前授权ID的特定授权ID。这个ID可以是

a）模块授权ID，如果代理正在执行的模块有一个授权ID，否则的话就是

b）会话授权ID。

只要当前授权ID拥有执行操作所涉及到的数据库元素所必需的权限时，可以执行这个SQL操作。

例8.35 为了明白检查权限的机制，重新考虑例8.34。假定所引用的表——Movie和Studio——是用户janeway创建和拥有的模式MovieSchema的一部分。这样，用户janeway拥有这些表和模式MovieSchema中任何其他元素的所有权限。他可以通过8.7.4节描述的机制将一些权限授权给别人，但是目前假定还没有授权给任何人。例8.34的插入执行方式有好几种。

1. 用户janeway创建了一个包含AUTHORIZATION janeway子句的模块，这个插入可以作为模块的一部分来执行。模块授权ID，如果有的话，通常变为当前授权ID。接着，模块和它的SQL插入语句的权限几乎与用户janeway拥有的权限相同，包括了表Movie和Studio的所有权限。

2. 插入可能是一个没有属主的模块的一部分。用户janeway在CONNECT语句中使用AUTHORIZATION janeway子句打开了一个连接。现在，janeway再次成为授权ID，所以插入语句拥有所需的所有权限。

3. 用户janeway将表Movie和Studio的所有权限授权给用户sisko，或是代表“所有用户”的特殊用户PUBLIC。插入语句存在于带有子句

```
AUTHORIZATION sisko
```

413

的模块中。由于当前授权ID现在是sisko，该用户拥有所需的权限，故插入再次被允许。

4. 同（3），用户janeway已经将所需的权限给了用户sisko。插入语句存在于没有属主的模块中，在一个会话中执行，该会话的授权ID由AUTHORIZATION sisko子句设置。这样，当前授权ID是sisko且这个ID拥有所需的权限。

□

例8.35说明了几条准则。将其摘要如下：

- 如果数据的属主与拥有当前授权ID的用户是同一个的话，所需的权限通常总可以得到。上述（1）和（2）说明了这一点。
- 如果数据的属主把这些权限授给拥有当前授权ID 的用户，或者这些权限被授给用户PUBLIC，所需的权限也可得到。情况（3）和（4）说明了这一点。
- 数据的属主，或者已经取得数据权限的用户执行该模块使得所需权限都可得到。当然，用户需要模块本身的EXECUTE权限。情况（1）和（3）说明了这点。
- 如果一个会话的授权ID是拥有所需权限的用户的授权ID时，在该会话中执行一个公用的

模块是合法执行这个操作的另一种方式。情况（2）和（4）说明了这点。

8.7.4 授权

在例8.35中看到了用户（也就是，授权ID）拥有所需权限的重要性。但是目前为止，拥有数据库元素权限的惟一方式是成为创建者和元素的属主。SQL提供了GRANT语句以允许一个用户将权限授给另一个用户。第一个用户仍然保留了所授予的权限。因此，GRANT被认为是“复制权限”。

权限的授予与复制之间有一个重要的区别。每个权限有一个相关的*授权选项*（grant option）。也就是说，用户可以拥有一个权限，如带有“授权选项”的表Movie上的SELECT权限。而第二个用户可以有相同的权限，但没有“授权选项”。于是，第一个用户可以将Movie的SELECT权限授权给第三个用户，甚至授权时可以有也可以没有授权选项。但是，第二个用户
414 没有授权选项，所以他不可以将Movie的SELECT权限授权给其他人。如果第三个用户以后得到这个带授权选项的相同权限，那么这个用户还可以将权限授权给第四个用户，可以有或没有授权选项，等等。

授权选项由下面的元素组成：

1. 关键字GRANT。
2. 一个或多个权限列表，如SELECT或INSERT（name）。关键字ALL PRIVILEGES可以选择性地出现在这儿，它表示对正在讨论的数据库元素（下面第四点提到的元素）授权者可合法授予的所有权限。
3. 关键字ON。
4. 数据库元素。这个元素一般是一个关系，或者是基本表或是视图。也可以是没有讨论过的域或别的元素（见8.3.2节的方框“更多的模式元素”），不过这种情况下元素名之前必须带有保留字DOMAIN或其他适当的保留字。
5. 保留字TO。
6. 一个或多个用户列表（授权ID）。
7. 保留字WITH GRANT OPTION，这是可选的。

即，授权语句的格式如下：

GRANT <privilege list> ON <database element> TO <user list>

其后也可以加上WITH GRANT OPTION。

为了合法地执行这条授权语句，执行它的用户必须拥有被授予的权限，而且这些权限还必须带有授权选项。但是，授权者可以拥有比授出的权限更通用的权限（带有授权选项）。例如，表Studio的INSERT（name）权限被授出，同时授权者拥有表Studio更通用的带有授权选项的权限INSERT。

例8.36 用户janeway是MovieSchema模式的属主，该模式包括表

```
Movie(title, year, length, inColor, studioName, producerC#)
Studio(name, address, presC#)
```

Janeway将表Studio的INSERT和SELECT权限以及表Movie的SELECT权限授予了用户
415 kirk和picard。甚至包括这些权限的授权选项。授权语句如下：

```
GRANT SELECT, INSERT ON Studio TO kirk, picard
    WITH GRANT OPTION;
GRANT SELECT ON Movie TO kirk, picard
    WITH GRANT OPTION;
```

现在，picard授予用户sisko相同的权限，但是没有授权选项。picard执行的语句为：

```
GRANT SELECT, INSERT ON Studio TO sisko;
GRANT SELECT ON Movie TO sisko;
```

Kirk授予sisko图8-25所需的最少权限，即Studio的SELECT和INSERT(name)权限以及Movie的SELECT权限。语句为：

```
GRANT SELECT, INSERT(name) ON Studio TO sisko;
GRANT SELECT ON Movie TO sisko;
```

注意到sisko从两个不同用户处接受了Movie和Studio的SELECT权限。他还两次接受了Studio的INSERT(name)权限：直接从kirk处获得和从picard处获得的更广泛的INSERT权限。 □

8.7.5 授权图

由于授权网的复杂以及一系列授权产生的重叠权限，用授权图（Grant diagram）表示授权很有意义。SQL系统维护这个图并跟踪权限和它们的起始点（以防权限被收回，见8.7.6节）。

授权图的节点对应一个用户和一个权限。要注意带或者不带授权选项的权限必须用两个不同的节点表示。如果用户*U*将权限*P*授予用户*V*，这个授权是基于*U*拥有权限*Q*（*Q*可以是带授权选项的*P*，或是*P*的带有授权选项的泛化），那么从节点*U*/*Q*到节点*V*/*P*画一条弧。

例8.37 8-26显示了例8.36的一系列授权语句产生的授权图。这里使用了一个约定：用户权限组合后，一个*表示该权限带有授权选项，而且，用户权限组合之后的**表明了该权限来自正在讨论的数据库元素的所有权而不是由于别处的权限授予。在8.7.6节讨论收权时这个区别很重要。被标两个星的权限自动包括了授权选项。 □ 416

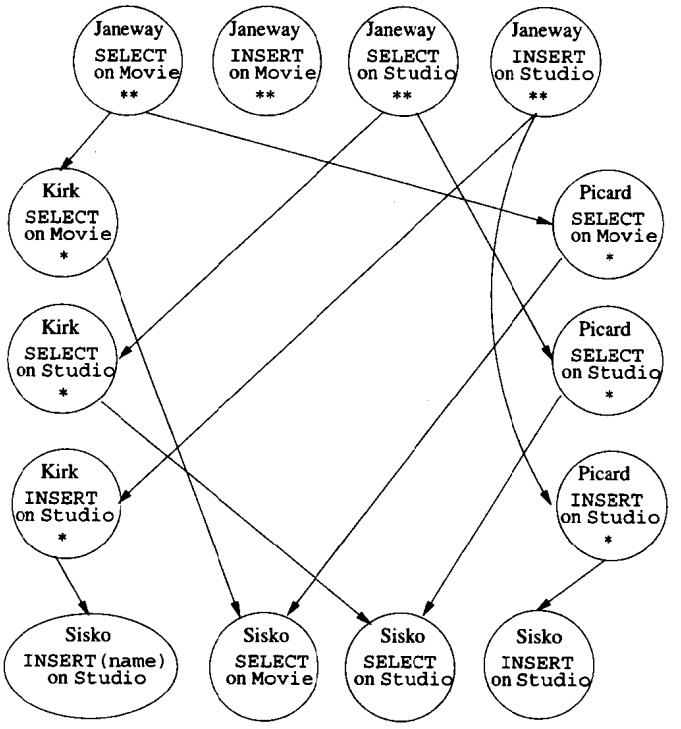


图8-26 授权图

8.7.6 销权

被授予的权限可以随时收回。事实上，权限的收回可能要求级联。级联的意思是，当收回已经被传递给其他用户的带授权选项的权限时，可能也要收回那些被此授权选项授予的权限。收权语句的简单形式如下：

1. 保留字REVOKE。
2. 一个或多个权限列表。
3. 保留字ON。
4. 数据库元素，如授权语句中描述的（4）。
5. 保留字FROM。

417 6. 一个或多个用户的列表（授权ID）。

即，收权语句格式为：

REVOKE <privilege list> ON <database element> FROM <user list>。

不过，语句中还需加入下面所列的选项之一：

1. 以CASCADE结尾的语句。如果这样的话，那么当收回指定的权限时，也要收回那些仅仅由于要收回的权限而被授予的权限。更严格地说，如果用户*U*要从用户*V*处收回权限*P*，基于权限*Q*是属于用户*U*，那么在授权图中删去从节点*U*/*Q*到节点*V*/*P*的弧。现在，那些无法从某个属主节点（被标为两颗星的节点）到达的节点也被删去。

2. 以RESTRICT结尾的语句，意味着：如果在前一项描述的级联规则由于要收回的权限被传递给其他人而造成了任何权限收回，那么该销权语句将不被执行。

允许用REVOKE GRANT OPTION FOR代替REVOKE，这种情况只保留自身的核心权限，但是它们授予他人的授权选项将被删除。可能不得不修改节点、重定向弧或创建新的节点来反映受影响的用户的变化。REVOKE的形式也可以与CASCADE 或RESTRICT结合使用。

例8.38 继续讨论例8.36，假定janeway使用下面的语句收回她授予picard的权限：

```
REVOKE SELECT, INSERT ON Studio FROM picard CASCADE;
REVOKE SELECT ON Movie FROM picard CASCADE;
```

删除图8-26中从janeway权限到picard权限对应的弧。既然规定了CASCADE，还要看一下在图中是否存在权限无法经标为双星的权限（基于属主的权限）到达。检查图8-26，看到picard的权限不再能从双星节点到达（如果还有另外的路径到达picard节点，那么也是可以到达）。Sisko对于Studio的INSERT权限也是不可达的。因此不仅从图中删除picard的权限，还要删除sisko的INSERT权限。

注意没有删除sisko关于Movie和Studio的SELECT权限或者他关于Studio的INSERT(name)权限，因为它们都可以通过kirk的权限从janeway基于属主的权限到达。最后的授权图见图8-27。 □

例8.39 用抽象的例子阐述微妙之处。首先，删除了通用权限*p*，没有删除*p*的特例。例如，
418 考虑下面的系列步骤，为何用户*U*，关系*R*的属主，将关系*R*的INSERT权限授予用户*V*，而且还授予了*R*的INSERT(A)权限。

Step	By	Action
1	*U*	GRANT INSERT ON *R* TO *V*
2	*U*	GRANT INSERT(A) ON *R* TO *V*
3	*U*	REVOKE INSERT ON *R* FROM *V* RESTRICT

当U从V中删除INSERT权限时，INSERT(A)权限仍被保留。第二步和第三步之后的授权图见图8-28。

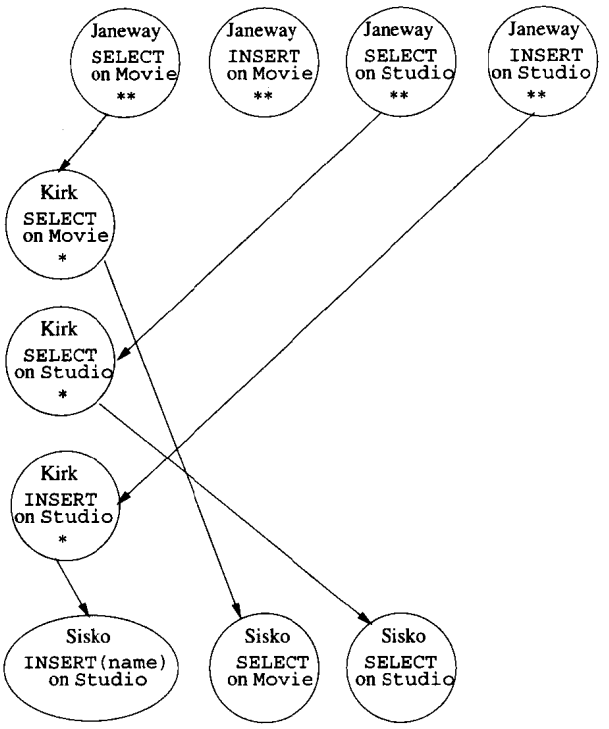
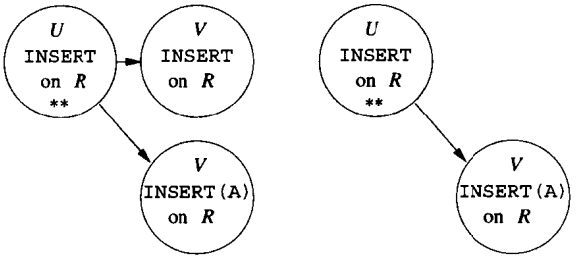


图8-27 删除picard权限的授权图

注意到第二步之后对于用户V拥有的相似但不同的权限有两个独立的节点。而且第三步的RESTRICT选项并没有阻止销权，因为V没有将这个选项授予别人。事实上，V不可能授出任何权限，因为V得到它们时没有授权选项。 □ 419



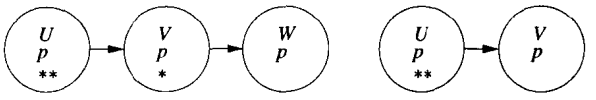
(a) 第二步之后 (b) 第三步之后

图8-28 收回通用权限保留特定权限

例8.40 现在考虑类似的例子，U授予V带授权选项的权限p，接着只收回授权选项。本例中，必须改变V的节点以反映授权选项的删除，而且由V作出的任何授权必须取消，这通过清除从V/P节点发出的弧完成。步骤顺序如下：

Step	By	Action
1	U	GRANT p TO V WITH GRANT OPTION
2	V	GRANT p TO W
3	U	REVOKE GRANT OPTION FOR p FROM V CASCADE

第一步，*U*授予*V*带授权选项的权限*p*。第二步，*V*使用授权选项将*p*授予*W*。此时授权图如图8-29(a)。



(a) 第二步之后　　(b) 第三步之后

图8-29 收回授权选项保留基本权限

接着第三步，*U*从*V*处收回权限*p*的授权选项，但是没有收回权限本身。因此，取消了节点*V*和*p*的星。但是没有*的节点不可能发出一条弧线，因为没有*的节点不可能是权限授权的起点。这样，还必须删除从节点*V*/*p*到节点*W*/*p*的弧线。

420 现在，节点*W*/*p*没有从代表权限*p*开始的**节点到*W*/*p*的路径。结果，从图中删除了节点*W*/*p*。但是，节点*V*/*p*仍保留，它只是被修改了，删除了代表授权选项的*。最后的授权图见图8-29(b)。 □

8.7.7 习题

习题8.7.1 指出执行下列查询需要何种权限。每种情况，说出最特别的权限以及一般性的权限就足够了。

a) 图6-5的查询

b) 图6-7的查询

* c) 图6-15的插入

d) 例6.36的删除

e) 例6.38的更新

f) 图7-5中基于元组的约束

g) 例7.13的断言

* **习题8.7.2** 显示图8-30中给出的操作序列步骤(4)至(6)的每一步执行以后的授权图。假设*A*是权限*p*所指向关系的属主。

Step	By	Action
1	*A*	GRANT *p* TO *B* WITH GRANT OPTION
2	*A*	GRANT *p* TO *C*
3	*B*	GRANT *p* TO *D* WITH GRANT OPTION
4	*D*	GRANT *p* TO *B*, *C*, *E* WITH GRANT OPTION
5	*B*	REVOKE *p* FROM *D* CASCADE
6	*A*	REVOKE *p* FROM *C* CASCADE

图8-30 习题8.7.2的操作序列

习题8.7.3 如图8-31所示，显示步骤(5)以及(6)以后的授权图。假设*A*是权限*p*所指向关系的属主。

Step	By	Action
1	*A*	GRANT *p* TO *B*, *E* WITH GRANT OPTION
2	*B*	GRANT *p* TO *C* WITH GRANT OPTION
3	*C*	GRANT *p* TO *D* WITH GRANT OPTION
4	*E*	GRANT *p* TO *C*
5	*E*	GRANT *p* TO *D* WITH GRANT OPTION
6	*A*	REVOKE GRANT OPTION FOR *p* FROM *B* CASCADE

图8-31 习题8.7.3的操作序列

习题8.7.4 显示经过以下步骤以后最终的授权图，假设A是权限p所指向关系的属主。

Step	By	Action
1	A	GRANT p TO B WITH GRANT OPTION
2	B	GRANT p TO B WITH GRANT OPTION
3	A	REVOKE p FROM B CASCADE

421

8.8 小结

- *嵌套SQL*：不采用基本查询界面来表达SQL查询以及修改，而是在一个传统的宿主语言中嵌套SQL查询，通常编写这样的程序更为有效。处理器将嵌套SQL语句转换成合适的宿主语言的函数调用。
- *阻抗不匹配*：SQL的数据模型和传统的宿主语言的数据模型有着很大的不同。因此，两者之间信息的传递是通过共享变量来实现的，共享变量可以表示程序中SQL部分中元组的组成分量。
- *游标*：游标是一个SQL变量，它指示关系中的一个元组。游标遍历关系的每个元组，使得宿主语言与SQL的连接变得较为容易。检索到当前元组的组成分量后存入共享变量中，并使用宿主语言进行处理。
- *动态SQL*：宿主程序中不是嵌套特定的SQL语句，而是创建字符串，由SQL系统将该字符串作为SQL语句解释并执行。
- *永久存储模块*：可以建立一些过程和函数的集合，作为数据库模式的一部分。这些过程和函数是用拥有所有常见的控制原语的特殊语言编写，如同SQL语句。它们可以由嵌套SQL或者一个基本查询界面触发。
- *数据库环境*：使用SQL DBMS的安装就创建了一个SQL环境。在此环境中，数据库元素(比如关系)组成（数据库）模式、日志以及簇。日志是模式的集合，而簇是用户可以看到的最大的元素集合。

422

- *客户/服务器系统*：一个SQL客户通过创建一个连接（两个进程之间的链接）和会话（操作序列）来建立到服务器的连接。在会话中执行的代码来自一个模块，此模块的执行叫做SQL代理。
- *调用层接口*：有一个称做SQL/CLI或者ODBC的标准函数库可以链接到任何一个C程序。功能与嵌套SQL相似，但它不需要预处理器。
- JDBC：Java数据库连接系统与CLI类似，只不过它使用了面向对象风格的Java。
- *并发控制*：SQL提供了两种机制来防止两个操作彼此干扰：事务和游标限制。游标限制包括声明一个游标是“不敏感”的，在此情况下，对关系的任何变化游标都不会看到。
- *事务*：SQL允许程序员把SQL语句组成事务，这些事务可以提交或者回滚（夭折）。事务的回滚或者是应用程序为了取消所作的变动而做的，或者是系统为了保证原子性和独立性而做的。
- *隔离级别*：SQL允许事务以四个隔离级别运行，从最串行的到最不串行的：“串行化”(事务必须完全在另一个事务之前或之后运行)、“读可重复”（查询得到的每个元组，在此查询再次被执行时必须重现）、“读提交”（只有那些被已提交事务写入的元组才可以被这个事务看到）、“读不提交”(对事务可以看到的信息不加限制)。
- *只读游标和事务*：游标和事务可以声明为只读的。该声明保证了游标和事务不会对数据

库造成任何变动，因此，通知SQL系统：该事务或游标不会以可能违反不敏感性、串行性或别的要求的方式来影响其他的事务或游标。

- 权限：出于安全性的考虑，SQL系统允许对数据库元素拥有多种不同的权限。这些权限包括选择（读）、插入、删除、更新关系的权力，还包括引用关系的权力（在限制的条件下引用他们），以及创建触发器的权力。
- 授权图：权限可以被其属主转授给其他用户或者一般用户PUBLIC，如果授权的时候带
423 有授权选项，那么，这些权限可以继续往下传。权限可以收回。授权图是一种非常有用的方法，用以记住足够多的授权和收权的历史记录来跟踪谁拥有何种权限以及他们是从何处获得的这些权限。

8.9 参考文献

读者要再次参考第6章的书目摘记以得到关于SQL标准的信息。[4]是PSM标准，[5]是一本关于这个主题的全面的书。[6]是关于JDBC的普及参考书。

[1]从事务和游标的范围讨论这个标准问题。

SQL授权机制的思想来自[3]和[2]。

1. Berenson, H., P. A. Bernstein, J. N. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," *Proceedings of ACM SIGMOD Intl. Conf. on Management of Data*, pp. 1–10, 1995.

2. Fagin, R., "On an authorization mechanism," *ACM Transactions on Database Systems* **3**:3, pp. 310–319, 1978.

3. Griffiths, P. P. and B. W. Wade, "An authorization mechanism for a relational database system," *ACM Transactions on Database Systems* **1**:3, pp. 242–255, 1976.

4. ISO/IEC Report 9075-4, 1996.

5. Melton, J., *Understanding SQL's Stored Peocedures: A Complete Guide to SQL/PSM*, Morgan-Kaufmann, San Francisco.

6. White, S., M. Fisher, R. Cattell, G. Hamilton, and M. Hapner, *JDBC*
424 *API Tutorial and Reference*, Addison-Wesley, Boston, 1999.

第9章 面向对象查询语言

本章将讨论把面向对象程序设计方法融入查询语言的两种途径。OQL（Object Query Language）是一种面向对象数据库的标准查询语言。它将SQL的高层次、说明性的编程方法和面向对象的编程范例结合起来。OQL对ODL（已在4.2节中介绍过的一种面向对象的数据描述语言）所描述的数据对象进行操作。

如果说OQL试图将SQL中最好的部分引入面向对象的领域，那么SQL-99标准所具有的相对较新的对象-关系特征，则是将面向对象方法引入到关系领域。在某种意义上，两种语言有着相同的地方，但对某些特定的事情，使用其中一种语言处理会比另一种更简单些。

本质上，两种面向对象的方法的主要区别在于他们对于关系的重要性的理解。以ODL与OQL为中心的面向对象的方式，认为关系并不是很重要。因此，在OQL中我们可以发现各种类型的对象，其中有一些是结构（即关系）的集合或包。对于SQL而言，依旧认为关系是最基本的数据结构概念。在4.5节所介绍的对象-关系方法中，关系模型的扩展是通过允许使用更复杂的关系元组和属性来实现的。这样，对象和类就可以被引入到关系模型中，但总是在关系的上下文中。

9.1 OQL简介

OQL（Object Query Language），向我们提供了一种类似SQL的符号化查询表达式。值得注意的是，OQL将作为对诸如C++，Smalltalk和Java等面向对象的*宿主*（host）语言的扩展。
对象由OQL查询以及传统的宿主语言语句来处理。宿主语言语句与OQL查询之间的结合不需通过显式的值传递来实现，这种方式优于8.1节所讨论的将SQL嵌入宿主语言的方式。

9.1.1 一个面向对象的电影例子

为了说明OQL的风格，我们引入一个实例，它包含我们已经熟悉的类`Movie`，`Star`和`Studio`。我们将使用如图4-3所示的定义，再增加一些键以及扩展声明。如图4-4所示，只有`Movie`包含方法。而完整例子如图9-1所示。

9.1.2 路径表达式

类似于C语言和SQL，我们通过一个点符号来访问对象或结构的成分。通用的规则如下：如果*a*表示一个属于类*C*的对象，*p*是该类的某个属性、关系或方法，那么*a.p*表示*p*作用于*a*的结果。即：

1. 如果*p*是一个属性，那么*a.p*是对象*a*中该属性的值；
2. 如果*p*是一个联系，那么*a.p*是表示通过联系*p*与对象*a*相关的对象或对象的集合；
3. 如果*p*是一个方法（可能有参数），那么*a.p*(…)表示*p*作用于*a*的结果。

例9.1 假设`myMovie`表示类型`Movie`的一个对象，那么：

- `myMovie.length`的值是该电影的长度，也就是`myMovie`所代表的`Movie` 对象的`length`属性的值。
- `myMovie.lengthInHours()`的值是一个实数，是以小时为单位的电影长度，通过将方法`lengthInHours`作用于对象`myMovie`而计算得到的。
- `myMovie.stars`的值是通过联系`stars`与`myMovie`相关联起来的`Star`对象的集合。

- 表达式myMovie.starNames(myStars)没有返回值（即在C++中，这种表达式的类型是空值）。然而，这有一个副作用，它将方法starNames的输出变量myStars的值设置为字符串的集合，那些字符串则是该电影中所有影星的名字。

426

```
class Movie
    (extent Movies key (title, year))
{
   attribute string title;
   attribute integer year;
   attribute integer length;
   attribute enum Film {color,blackAndWhite} filmType;
   relationship Set<Star> stars
                inverse Star::starredIn;
   relationship Studio ownedBy
                inverse Studio::owns;
   float lengthInHours() raises(noLengthFound);
   void starNames(out Set<String>);
   void otherMovies(in Star, out Set<Movie>)
                raises(noSuchStar);
};


class Star
    (extent Stars key name)
{
    attribute string name;
    attribute Struct Addr
        {string street, string city} address;
    relationship Set<Movie> starredIn
                 inverse Movie::stars;
};


class Studio
    (extent Studios key name)
{
    attribute string name;
    attribute string address;
    relationship Set<Movie> owns
                 inverse Movie::ownedBy;
};
```

427

图9-1 面向对象的电影数据库的一部分

箭头与点

OQL可以使用箭头符号‘->’来代表点符号。这种习惯继承了C语言的风格，点符号与箭头符号都可以获取结构的某个成分。然而，在C语言中，箭头符号与点符号略有区别，而在OQL中二者是一样的。在C语言中，表达式a.f中a是一个结构类型，而p->f中p是一个指向结构类型的指针。但二者都表示该结构中域f的值。

在保证有意义的前提下，表达式可以含有多个点符号，例如：如果myMovie表示一个电影对象，那么myMovie.ownedBy表示拥有该电影的Studio对象。而myMovie.ownedBy.name是一个字符串，代表该电影公司的名字。 □

9.1.3 OQL中Select-From-Where表达式

OQL允许使用类似于SQL查询语言的select-from-where语法来书写查询表达式。例如，查询影片*Gone With the Wind*的年份：

```
SELECT m.year
FROM Movies m
WHERE m.title = "Gone With the Wind"
```

可以看到，除了那个字符串常量的双引号外，该查询很类似于SQL。

通常，OQL的select-from-where表达式包含：

1.关键字SELECT，后接一列表达式。

2.关键字FROM，后接一个或多个变量声明。一个变量的声明可以为：

(a) 一个值为集（collection）类型（如：集合或包）的表达式；

(b) 可选关键字AS；

(c) 变量的名字。

典型地，(a)中的表达式是某些类的扩展，如上述的类Movie的扩展Movies。一个扩展就相当于SQL的FROM子句中的关系。然而，也可以将生成集类型的表达式，如另一个select-from-where表达式，放到变量声明中。 428

3.关键字WHERE和一个布尔表达式。同SELECT子句后面的表达式一样，该表达式只可以使用常量或在FROM子句中声明的变量。除了不等号用‘<>’而不是用‘!=’之外，比较操作符和SQL中的是一样的。逻辑运算符和SQL中的一样，是AND、OR和NOT。

查询产生一个对象包。考虑FROM子句中所有可能的变量值，我们通过嵌套的循环来计算这个包。如果有任何一个变量值的组合满足WHERE子句中的条件，那么这个由SELECT子句描述的对象将加进包中，最后得到的这个包就是select-from-where语句的结果。

例9.2 一个更为复杂OQL查询：

```
SELECT s.name
FROM Movies m, m.stars s
WHERE m.title = "Casablanca"
```

这个查询查找电影*Casablanca*中影星的姓名。注意FROM子句中各项的顺序：首先我们定义类Movie的任意一个对象m，这是通过把m声明在类Movie的扩展Movies中得到的。接着，对于m的每个值，我们定义s为一个Star对象，作为电影m的电影明星集合m.stars的一员。也就是，我们考虑两重嵌套循环中所有的(m, s)偶对，其中m是一部电影，而s是这部电影中的一个影星。其等价的描述为：

```
FOR each m in Movies DO
    FOR each s in m.stars DO
        IF m.title = "Casablanca" THEN
            add s.name to the output bag
```

WHERE子句将我们的考虑限制在含有特定对象m的偶对上，该对象m是片名为*Casablanca*的电影对象。因此，SELECT子句生成满足WHERE子句条件的(m, s)偶对中的所有影星对象s的名字属性的一个包（在本例的情况下应是一个集合）。这些名字是集合m_c.stars中影星的名字，其中m_c是电影*Casablanca*的电影对象。 □

9.1.4 修改结果的类型

类似于例9.2的查询，查询的结果是生成一个字符串的包。也就是说，OQL和SQL中的默

认方式一样，不会去除重复的结果，除非指示它这样做。然而，如果我们想这样做的话，可以强制结果为集合或列表。

429 • 与SQL类似，为了使结果变为一个集合，可以在SELECT后面使用关键字DISTINCT。

• 仍然与SQL类似，为了使结果变为一个列表，可以在查询的末尾加上ORDER BY。

FROM列表的替换形式

除了在FROM子句中使用SQL风格的元素（即一个集合类型后面跟着一个元素名）之外，OQL可以使用完全等价的、更具有逻辑性的、而少一些SQL风格的格式。我们可以给定一个常用的元素名，然后是关键字IN，最后是集合名。例如：

```
FROM m IN Movies, s IN m.stars
```

就是例9.2中查询中一个等价的FROM子句。

下面的例子将阐述正确的语法。

例9.3 假设我们查询迪斯尼电影中影星的名字。如下的查询可以满足要求。如果在几个影片中出现同一位影星，那么在结果中去除重复的姓名。

```
SELECT DISTINCT s.name
FROM Movies m, m.stars s
WHERE m.ownedBy.name = "Disney"
```

这个查询的策略类似于例9.2。我们仍然考虑以两重嵌套循环得到的所有由电影和该电影中影星组成的偶对。但现在，偶对(m,s)上的条件改为电影公司的名字“Disney”，其Studio对象是m.ownedBy。 □

在OQL中，ORDER BY子句类似于SQL。关键字ORDER BY后面紧接着一列表达式。对查询结果中的每个对象，计算第一个表达式的值，并根据该值对每个对象进行排序。如果存在相等的值，那么再根据第二个表达式的值进行排序，如此类推。与SQL类似，在缺省情况下是升序，也可以分别使用关键字ASC或DESC来选择升序或降序。

例9.4 假设我们查询迪斯尼电影的集合，但要求结果是根据影片长度排序的一个列表。如果有相等的情况，则等长的影片再根据字母序进行排序。该查询为：

```
SELECT m
FROM Movies m
WHERE m.ownedBy.name = "Disney"
430 ORDER BY m.length, m.title
```

在前三行中，我们考察每一个Movie对象*m*。如果那些拥有这部电影的公司名是“Disney”，那么，整个对象*m*将成为输出包的一员。而第四行中指明，由select-from-where查询生成的结果对象*m*根据m.length（电影的长度）进行排序。如果排序的结果中有相同的，那么再根据m.title（电影的名字）进行排序。这样，查询结果将生成Movie对象的一个列表。 □

9.1.5 复杂输出类型

在SELECT子句中的元素不一定是简单对象。它们可以是任意的表达式，包括由类型构造函数生成的表达式。例如，可以把Struct类型构造函数作用于几个表达式，得到生成这些结构的集合或包的select-from-where查询语句。

例9.5 假如我们想得到居住在同一地址的影星的偶对，可以通过如下查询：

```
SELECT DISTINCT Struct(star1: s1, star2: s2)
FROM Stars s1, Stars s2
WHERE s1.address = s2.address AND s1.name < s2.name
```

这就是说，我们考虑影星的所有偶对，s1和s2。其中，WHERE子句检查他们是否有相同的地址；同时依据字母序检查，第一个影星的姓名是否在第二个影星之前。这样就不会生成含有同一个影星出现两次的偶对，也不会生成以不同次序出现的同一对影星。

每一个经过两遍检查的偶对，都会生成一个记录结构。该结构的类型是一个有两个域的记录，分别是star1和star2。每个域的类型是Star类，因为给这两个域提供值的变量s1和s2的类型是Star类型。也就是说，该结构类型的规范形式为：

```
Struct{star1: Star, star2: Star}
```

查询结果的类型为以下结构的集合：

```
Set<Struct{star1: Star, star2: Star}>
```

□

9.1.6 子查询

在任何一个集出现的地方，我们都可以使用select-from-where表达式。我们将给出一个出现在FROM子句中的例子。其他一些子查询的例子将在9.2节中给出。 431

在FROM子句中，可以使用子查询生成一个集。于是我们可以生成一个表示这个集的元素的变量，它可以遍历集内的所有成员。

长度为1的SELECT列表是特例

注意，当一个SELECT列表仅有一个表达式时，结果的类型是该表达式类型的值的集合。然而，如果在SELECT列表中有多个表达式，那么就会有一个隐式结构生成，每个表达式都作为这个结构的成分。因此，即使我们让例9.5的查询以下面的语句开始：

```
SELECT DISTINCT star1: s1, star2: s2
```

结果的类型将会是：

```
Set<Struct{star1: Star, star2: Star}>
```

但例9.3中结果类型是Set<String>，而不是Set<Struct{name: String}>。

例9.6 让我们重做例9.3中的查询，该查询查找出现在迪斯尼公司制作的电影中的影星。首先，同例9.4一样，通过以下查询获得迪斯尼电影的集合：

```
SELECT m
FROM Movies m
WHERE m.ownedBy.name = "Disney"
```

我们可以利用这个查询作为子查询语句来定义一个集合，使得一个表示迪斯尼制作的电影的变量d可以遍历该集合中的所有成员：

```
SELECT DISTINCT s.name
FROM (SELECT m
      FROM Movies m
      WHERE m.ownedBy.name = "Disney") d,
     d.stars s
```

这种“查找迪斯尼电影的影星”的方式不比例9.3简洁，甚至可能更差。但是，它说明了

一种在OQL中有效的构造查询序列的新方式。在上面的查询中，FROM子句中有两个嵌套循环。在第一个循环中，变量d遍历了所有的迪斯尼电影，子查询的结果放在FROM子句中。在嵌套于第一个循环的第二个循环中，变量s遍历了迪斯尼电影变量d中的所有影星。注意，在外查询
432 中不需要WHERE子句。 □

9.1.7 习题

习题9.1.1 图9-2是一个用ODL描述产品的例子。在主类Product下分别设置了3个子类。读者要注意，一个产品的类型既可以从type属性获得，也可以从它所属的子类获得。这并不是一个很好的设计，因为其中可能出现这样的情况，如一个PC机对象的type属性可能与“laptop”或“printer”相同。然而，这种设计可以使你了解如何表示查询。

因为Printer的类型是从其父类Product继承而来的。我们不得不将Printer的类型属性改名为printerType。后面的属性是打印方式（如激光和喷墨打印机）而Product的类型的值将是PC机、手提电脑或打印机。

在图9-2的ODL代码中加入方法名（参见4.2.7节），完成下列事情：

* a) 把产品的价格减掉x。假设x是函数的一个输入参数。

* b) 如果该产品是PC机或者手提电脑，返回产品的速度，否则返回notComputer异常。

c) 根据特定的输入值x设置手提电脑的屏幕大小。

! d) 给定一个输入产品p，判断产品q其速度是否高于p而价格低于p。如果p不具有速度属性（如，p不是PC机也不是手提电脑），则产生一个badInput异常；如果q不具有速度属性，则产生一个noSpeed异常。

```
class Product
    (extent Products
     key model)
 {
    attribute integer model;
    attribute string manufacturer;
    attribute string type;
    attribute real price;
 };


class PC extends Product
    (extent PCs)
 {
    attribute integer speed;
    attribute integer ram;
    attribute integer hd;
    attribute string rd;
 };


class Laptop extends Product
    (extent Laptops)
 {
    attribute integer speed;
    attribute integer ram;
    attribute integer hd;
```

图9-2 ODL描述的product模式

```
        attribute real screen;
    };


    class Printer extends Product
        (extent Printers)
    {
        attribute boolean color;
        attribute string printerType;
    };
```

图9-2 （续）

习题9.1.2 使用习题9.1.1和图9-2的ODL模式，写出以下的OQL查询：

* a) 找出所有价格低于$2000的PC机的型号。

b) 找出所有内存大于128兆的PC机的型号。

*! c) 找出至少生产两种激光打印机的厂家。

d) 找出偶对(*r*, *h*)的集合。其中*r*表示某台PC机或手提电脑有*r*兆内存，*h*表示这台PC机或手提电脑有*h*吉字节的硬盘。

e) 根据处理器的速度，按升序生成PC机（对象，不是型号）的列表。

! f) 将至少有64兆内存的手提电脑，按屏幕的大小，降序生成其型号列表。

习题9.1.3 图9-3是一个战舰数据库的ODL描述，试向其中加入以下的方法：

a) 计算一艘舰的火力，即火炮的门数乘上口径的立方。

b) 找出一艘舰的姐妹舰。如果该舰是类中惟一的舰，产生一个noSisters异常。

c) 给定战役*b*和舰只*s*作为输入参数，假设舰只*s*参加了战役*b*，找出在这场战役中沉没的舰只。如果该舰只没有参加战役*b*，则产生一个didNotParticipate异常。

d) 给定舰名以及下水的年份作为参数，将该舰加入方法作用的类中去。

```
class Class
    (extent Classes
     key name)
{
    attribute string name;
    attribute string country;
    attribute integer numGuns;
    attribute integer bore;
    attribute integer displacement;
    relationship Set<Ship> ships inverse Ship::classOf;
};

class Ship
    (extent Ships
     key name)
{
    attribute string name;
    attribute integer launched;
    relationship Class classOf inverse Class::ships;
    relationship Set<Outcome> inBattles
                 inverse Outcome::theShip;
};
class Battle
```

图9-3 ODL描述的战舰数据库

```
    (extent Battles
     key name)
{
    attribute name;
    attribute Date dateFought;
    relationship Set<Outcome> results
                 inverse Outcome::theBattle;
};

class Outcome
    (extent Outcomes)
{
    attribute enum Stat {ok,sunk,damaged} status;
    relationship Ship theShip inverse Ship::inBattles;
    relationship Battle theBattle inverse Battle::results;
};
```

图9-3 （续）

! **习题9.1.4** 重复习题9.1.2，要求在每个查询中至少使用一个查询子句。

习题9.1.5 使用习题9.1.3和图9-3中的ODL模式，写出如下OQL查询操作：

a) 找出至少有九门炮的舰的类名。

b) 找出至少有九门炮的舰（对象，不是舰名）。

c) 找出排水量在30 000吨以下的舰的名字。按照下水的时间先后生成列表，如果有相同，再按照舰名的字母序排序。

d) 找出姐妹舰（如，同一类的舰）的对象配对。注意，配对中是舰对象本身，而不是舰的名字。

! e) 找出至少有两个国家的舰在战斗中沉没的所有战役的名字。

!! f) 找出没有舰受创的所有战役的名字。

9.2 OQL表达式的其他格式

在这一节中，除了select-from-where语句，我们将看到OQL所提供的用来构造表达式的其他一些操作符。其中包括：逻辑量词（全称量词和存在量词）、聚集操作符、分组操作符和集合操作符（并集，交集和差操作符）。

433 ≀ 436

9.2.1 量词表达式

我们可以测试是否一个集的所有成员都满足某一条件，也可以测试是否至少存在一个成员满足某个条件。为了测试集合S中是否所有的成员x都满足某一条件$C(x)$，可以用以下OQL表达式：

FOR ALL x IN S: $C(x)$

若每个在S中的x都满足$C(x)$，则说明表达式的结果为TRUE，否则为FALSE。类似地，对于表达式

EXISTS x IN S: $C(x)$

如果在S中至少有一个x使得$C(x)$的值为TRUE，则表达式的值为TRUE；否则，表达式的值为FALSE。

例9.7 图9-4给出了查询“找出迪斯尼电影中的所有影星”的另一种表达方式。这里，我们重点放在影星s上，查询他们是否是某部迪斯尼电影中的影星。从第(3)行可知，我们要考虑电影集合s.starredIn中所有电影m，该集合由影星s主演的所有电影构成。第(4)行则查询电

影m是否为一部迪斯尼电影。如果我们找到这样的电影m，即使只有一个，第(3)行及第(4)行中的EXISTS表达式值为TRUE；否则为FALSE。 □

```
1)  SELECT s
2)  FROM Stars s
3)  WHERE EXISTS m IN s.starredIn :
4)     m.ownedBy.name = "Disney"
```

图9-4 使用关于存在的子查询

例9.8 让我们用全称操作符写一个查询，查找出只主演过迪斯尼电影的所有影星。从技术上来讲，该集合包含那些没主演过任何一部电影的影星（就我们的数据库而言）。但是可以在查询语句中加入新的条件，要求影星至少主演过一部电影，这个改进留作习题。图9-5描述了这个查询。 □

```
SELECT s
FROM Stars s
WHERE FOR ALL m IN s.starredIn :
   m.ownedBy.name = "Disney"
```

图9-5 使用含有全称量词的子查询

9.2.2 聚集表达式

OQL使用与SQL相同的5个聚集操作符：AVG、COUNT、SUM、MIN和MAX。但是在SQL中，这些操作符只作用于一个表的指定的列，而在OQL中可以作用于所有包含合适类型成员的集合。 437
也就是说，COUNT可以作用于任何的集合；SUM和AVG可以作用于算术类型（如整型）的集合；而MIN和MAX则可以作用于可比较类型（如整型与字符串）的集合。

例9.9 为了计算所有电影的平均长度，我们需要建立一个所有电影长度的包。注意，不是生成电影长度的集合，因为如果是那样的话，相同长度的电影将只记一个。查询语句为：

```
AVG(SELECT m.length FROM Movies m)
```

这里，我们使用一个子查询从电影中来抽取长度成员。其结果是电影长度的一个包，而且使用AVG操作符得到希望的结果。 □

9.2.3 分组表达式

SQL中的GROUP BY子句被带到OQL中，但是确切地说，该子句发生了一点变化。GROUP BY子句在OQL中的格式为：

1. 关键字GROUP BY。

2. 一个用逗号隔开的、含有一个或多个划分属性（partition attribute）的列表。每一个属性包含：

(a) 一个域名

(b) 一个冒号

(c) 一个表达式

也就是说，GROUP BY子句的格式为：

GROUP BY f_1:e_1, f_2:e_2, ..., f_n:e_n

每一个GROUP BY子句都跟在一个select-from-where查询之后，表达式e_1，e_2，...，e_n可以引用FROM子句中的变量。为了说明GROUP BY是如何工作的，让我们看看在FROM子句中只有 438

一个变量x的一般情况。x的取值范围在某一个集C中，对于满足WHERE子句条件的C的成员i，我们计算所有的表达式获得值$e_1(i)$，$e_2(i)$，...，$e_n(i)$。这些值的列表也就是值i所属的组。

中间集

由GROUP BY子句返回的值实际上是一个结构的集合，我们称之为中间集（intermediate collection）。其中的每个成员有以下的格式：

Struct(f_1:v_1, f_2:v_2, ..., f_n:v_n, partition:P)

前n个域指明了这个组。也就是说，(v_1, v_2, ..., v_n)是值 ($e_1(i)$, $e_2(i)$, ..., $e_n(i)$)的列表，使得集C中至少有一个i值满足WHERE子句中的条件。

最后一个域有一个特殊的名字partition。直观上，它的值P是属于这个组的i的值。更准确的说法是，P是一个包含有形如Struct(x:i)的结构的包，其中x是FROM子句中的变量。

输出集

含有GROUP BY子句的select-from-where表达式中的SELECT子句，可以只引用中间集结构中的域，即f_1, f_2, ..., f_n和partition。对于形成partition值的包P中的成员，我们可以通过partition引用这些成员结构中的域x。这样，我们可以引用出现在FROM子句中的变量x，但是我们只能在对包P的成员进行聚集操作时才可以这样做。SELECT子句的结果将作为输出集（output collection）来引用。

例9.10 让我们为每个电影公司每一年的电影的总长度创建一个表。在OQL中，我们所构造的实际上是结构的一个包，其中每个结构有3个成分：电影公司、年份和该电影公司在该年制作的电影的总长度。该查询如图9-6所示：

```
SELECT stdo, yr, sumLength: SUM(SELECT p.m.length
                                FROM partition p)
FROM Movies m
GROUP BY stdo: m.ownedBy.name, yr: m.year
```

图9-6 用电影公司和年份对电影进行分组

为了理解这个查询，我们从FROM子句开始讨论。在该子句中，变量m遍历了所有的
439 Movie对象。因此，在我们的讨论中，m是扮演x的角色。在GROUP BY子句中，有两个域stdo与yr，分别对应于表达式m.ownedBy.name和m.year。

例如，*Pretty Woman*是迪斯尼公司在1990拍摄的一部电影。如果m表示该电影对象，那么m.ownedBy.name的值是“Disney”，而m.year的值是1990。结果，中间集中有这样一个成员，其结构为：

Struct(stdo:"Disney", yr:1990, partition:P)

其中，P是一个结构的集合。例如，它包含

Struct(m:m_{pw})

其中m_{pw}是电影*Pretty Woman*的Movie对象。仅有一个成分的结构也在P中，这些结构的域名m表示迪斯尼公司在1990年出品的其他电影。

现在，让我们看看SELECT子句。我们为中间集中的每个结构在输出集中创建一个结构。每个输出结构的第一个成分是stdo。也就是说，该成分的域名是stdo，而它的值等于中间集相应结构的stdo域的值。类似地，结果的第二个成分拥有域名yr以及一个与中间集相应结构的yr域的值相等的值。

每个输出结构的第三个成分是：

```
SUM(SELECT p.m.length FROM partition p)
```

为了理解这个select-from表达式，我们首先要认识到，变量p遍历GROUP BY结果中结构partition域上的成员。每个p的值都是Struct(m:o)形式的结构，其中o是一个电影对象。因此，表达式p.m指向这个对象o。所以，p.m.length是指该电影对象的长度成员。

所以，select-from查询子句生成特定组中电影的长度的包。例如，如果stdo的值为"Disney"，yr的值为1990，那么结果就是迪斯尼公司在1990年出品的这些电影长度的包。当我们对这个包使用操作符SUM时，我们就可以获得组中的电影长度的总和。因此，输出集中的一个成员可能为：

```
Struct(stdo:"Disney", yr:1990, sumLength:1234)
```

其中1234为迪斯尼公司在1990年出品的所有电影的总长度。 □

当FROM子句有多个集时的分组

如果在FROM子句中有多个变量，那么查询的解释就必须有一些相应的变化，但是总的原则同上述一个变量的情况是一样的。假设FROM子句中出现的变量为x_1，x_2，...，x_k，那么： 440

1. 所有的变量x_1，x_2，...，x_k，都可以在GROUP BY子句中的表达式e_1，e_2，...，e_n中使用。
2. 以partition域为值的包，其结构中有以下这些域：x_1，x_2，...，x_k。
3. 假设i_1，i_2，...，i_k分别是变量x_1，x_2，...，x_k的值，并且使WHERE子句为真，那么在中间集中有如下形式的结构：

$$\text{Struct}(f_1{:}e_1(i_1,\ldots,i_k),\ldots,f_n{:}e_n(i_1,\ldots,i_k),\text{partition:P})$$

而在包P中的结构为：

$$\text{Struct}(x_1{:}i_1, x_2{:}i_2, \ldots, x_k{:}i_k)$$

9.2.4 HAVING子句

像SQL查询中一样，OQL中的GROUP BY子句后面可以跟着HAVING子句。也就是说，以下形式的子句

```
HAVING <condition>
```

用于删除GROUP BY子句中生成的某些组。其中的条件将作用到中间集中每个结构的partition域的值。如果为真，那么就对该结构作和9.2.3节中一样的处理，生成输出集中的一个结构。若为假，则该结构不放到输出集中。

例9.11 让我们重做例9.10，查询电影公司制作的电影的总长度和制作年份，但是要求这家电影公司制作过片长至少为120分钟的电影。图9-7所示的查询完成这项工作。注意，在HAVING子句中，我们使用了和SELECT子句相同的查询语句，以获得给定电影公司和年份后得到的电影长度的包。在HAVING子句中，我们取这些长度的最大值，并与120进行比较。 □

```
SELECT stdo, yr, sumLength: SUM(SELECT p.m.length
                                FROM partition p)
FROM Movies m
GROUP BY stdo: m.ownedBy.name, yr: m.year
HAVING MAX(SELECT p.m.length FROM partition p) > 120
```

图9-7 组的约束 441

9.2.5 并、交和差操作

对于集合或包类型的两个对象，可以使用并、交和差操作符。像SQL一样，这三个操作符

分别用UNION、INTERSECT和EXCEPT这三个关键字来表示。

```
1)      (SELECT DISTINCT m
2)       FROM Movies m, m.stars s
3)       WHERE s.name = "Harrison Ford")
4)  EXCEPT
5)      (SELECT DISTINCT m
6)       FROM Movies m
7)       WHERE m.ownedBy.name = "Disney")
```

图9-8 使用集合差操作的查询

例9.12 如图9-8所示，通过对两个select-from-where查询进行差操作，我们可以查找出由Harrison Ford主演的不是迪斯尼公司出品的电影的集合。第(1)行到第(3)行查找Ford主演的电影，而第(5)行到第(7)行查找由迪斯尼公司出品的电影。第(4)行的EXCEPT取得它们的差集。 □

我们应该注意图9-8中第(1)行与第(5)行中的关键字DISTINCT。这个关键字使得两个查询的结果成为集合类型，如果没有DISTINCT，结果将是包（多重集合）类型。在OQL中，操作符UNION、INTERSECT和EXCEPT作用于集合或包。当两个变量均为集合时，这些操作符有通常集合的意义。

然而，当两个变量都是包类型，或者一个是集合、一个是包时，这些操作符则有包的含义。回忆5.3.2节，那里解释了包的并、交和差操作的定义。

从图9-8中的查询可知，某部电影出现在子查询结果中的次数是0或1，所以无论有没有DISTINCT，结果都是一样的。然而，结果的类型却不一样。如果使用了DISTINCT，那么结果类型是Set<Movie>；如果DISTINCT在一个地方不用或两个地方都不用时，结果类型则是Bag<Movie>。

9.2.6 习题

习题9.2.1 使用习题9.1.1和图9-2中的ODL模式，用ODL语言写出以下的查询序列：

* a) 查找同时生产PC机和打印机的厂家。

442 * b) 查找硬盘至少有20G的PC机生产厂家。

c) 查找只生产PC机而不生产手提电脑的厂家。

* d) 查找PC机的平均速度。

* e) 分别根据每种CD或DVD的速度，查找出PC机的平均内存的大小。

! f) 查找产品至少有64MB内存并且价格低于$1000的厂家。

*!! g) 对所制造的PC的平均速度至少为1200的每个厂家，查找他们在PC机中配置的最大内存。

习题9.2.2 使用习题9.1.3和图9-3中的ODL模式，用ODL语言写出以下的查询序列：

a) 查出所有在1919年之前下水的舰只的类别。

b) 找出每种类别舰只的最大排水量。

! c) 对于每一门大炮，查出配有此炮的舰的最早下水年份。

!! d) 根据所有在1919年之前下水的战舰的类别，查出此类别中的战舰在战役中沉没的数目。

! e) 查出各类舰中拥有船只的平均数。

! f) 查出船只的平均排水量。

!! g) 查找出那些至少有一艘来自美国的舰只参战，且至少有两艘舰沉没的战役。

! **习题9.2.3** 我们在例9.8中讲过，图9-5的OQL查询将返回那些没有主演过一场电影的影星，因此从技术上描述为："只会出现在迪斯尼电影中"。重写该查询，查找出至少主演过一部

电影的影星，而且这些影星只主演过迪斯尼电影。

! **习题9.2.4** 如果EXISTS x IN S : $C(x)$是假，那么FOR ALL x IN S : $C(x)$有没有可能为真？给出你的理由。

9.3 OQL中对象的赋值与创建

这一节将讨论OQL如何与宿主语言连接。我们将以C++为例子，虽然在某些系统中宿主语言可能是另外的面向对象的通用编程语言（如Java）。

9.3.1 宿主语言变量的赋值

与SQL在元组分量和宿主语言变量之间传送数据的方式不同，OQL很自然地嵌入到其宿主语言中。更确切地说，我们所学过的一些OQL的表达式，如select-from-where，是生成对象作为值的。这样就可以把OQL表达式的结果作为值，赋给具有合适类型的宿主语言变量。

例9.13 下面的OQL表达式

```
SELECT DISTINCT m
FROM Movies m
WHERE m.year < 1920
```

生成所有在1920年之前出品的电影的集合，它的类型是Set<Movie>。如果oldMovies是相同类型的宿主语言变量，那么可以（使用经过OQL扩展的C++语言）写出如下查询：

```
oldMovies = SELECT DISTINCT m
            FROM Movies m
            WHERE m.year < 1920;
```

这样oldMovies的值将成为Movie对象的集合。 □

9.3.2 集合元素的提取

由于select-from-where与group-by表达式都生成集（集合、包或列表），因此，如果我们需要该集内的某个元素，就得做一些额外的工作。即使我们肯定某个集仅有一个元素，这个说法也是成立的。OQL提供操作符ELEMENT，把一个单元素集转化成单独成员。例如，知道某个查询返回一个单元素的结果，就可以用这个操作符。

例9.14 假设我们想给类型为Movie（即是Movie类）的变量gwtw赋值为电影对象，表示电影*Gone With the Wind*。以下查询的结果是一个仅包含一个对象的包：

```
SELECT m
FROM Movies m
WHERE m.title = "Gone With the Wind"
```

我们不能直接将这个包赋给变量gwtw，否则将会出现一个类型错误。但是如果我们先使用ELEMENT操作符：

```
gwtw = ELEMENT(SELECT m
               FROM Movies m
               WHERE m.title = "Gone With the Wind"
       );
```

则变量的类型就和表达式匹配了，赋值就是合法的。 □

9.3.3 获取集合的每一个成员

获取集合或包中的每个成员更加复杂，但是仍然比SQL中基于游标的算法简单。首先，需

要使用含有ORDER BY子句的select-from-where表达式将集合或包转化为列表。回顾一下9.1.4节可知，这种表达式的结果就是选出的对象或值的列表。

例9.15 假设我们想要获得Movie类中的所有电影对象的列表，我们可以利用电影的名字以及年份（为了排除相同的情况），因为(title, year)是Movie的键。以下的语句把按照影片名和年份排好序的所有电影对象的列表，赋值给宿主语言变量movieList。 □

```
movieList = SELECT m
            FROM Movies m
            ORDER BY m.title, m.year;
```

一旦我们有了一个排好序或没有排好序的列表，就可以通过序号访问每个元素：$L[i-1]$获取列表L的第i个元素。注意，列表和数组的序号都假定是从0开始的，类似C或C++。

例9.16 假设我们想写一个C++函数，打印出每部电影的名字、年份和长度。该函数的简单框架如图9-9所示。

```
1)  movieList = SELECT m
                FROM Movies m
                ORDER BY m.title, m.year;
2)  numberOfMovies = COUNT(Movies);
3)  for(i=0; i<numberOfMovies; i++) {
4)      movie = movieList[i];
5)      cout << movie.title << " " << movie.year << " "
6)           << movie.length << "\n";
    }
```

图9-9 查看并打印每部电影

第(1)行对Movie类进行排序，将结果放进类型为List<Movie>的变量movieList中。
445 第(2)行使用OQL的操作符COUNT计算电影的数目。第(3)行到第(6)行是一个for 循环，在这个循环中，变量i遍历这个列表的每个位置。为了方便起见，列表的第i个元素赋给了变量movie。这样，在第(5)和第(6)行中，movie相应的属性就被打印出来。 □

9.3.4 OQL中的常量

OQL中的常量（有时作为不变对象来引用）是通过一个基和递归构造函数来构造的，这样的方式与ODL类型的构造方式类似。

1. 基本值（basic value），可以是下面任一个：

(a) 原子值（atomic value）：整数、浮点数、字符、字符串和布尔值。这些都已经在SQL中预定义过了，除非用双引号括起来作为字符串。

(b) 枚举值（enumeration）。在一个枚举中的值实际上是在ODL中声明。任何一个这样的值都可以用作常量。

2. 复合值（complex value），通过以下的类型构造函数来构造：

(a) Set(...)
(b) Bag(...)
(c) List(...)
(d) Array(...)
(e) Struct(...)

前四种称为集类型（collection type）。集类型和Struct类型可以随意作用到相应类型（基本类型或复合类型）的任何值上。但是，在应用Struct操作符的时候，需要具体给出域的名字和相应的值。每个域的名字后面紧跟一个冒号和域的值，而各个域-值对之间用逗号隔

开。注意同样的类型构造函数也在ODL中使用，但是在这里，我们用圆括号而不用尖括号。

例9.17 表达式bag(2, 1, 2)表示一个包，在这个包中整数2出现两次，整数1出现一次。而表达式

```
Struct(foo: bag(2,1,2), bar: "baz")
```

表示一个有两个域的结构。域foo的值是前面描述的包，而域bar的值是字符串“baz”。 446

□

9.3.5 创建新对象

我们已经看到，OQL表达式如select-from-where，允许我们创建新对象。通过把常量或者别的表达式显式地组合成结构和集合，也可以创建对象。我们从例9.5中已看到过这种惯例，其中的一行

```
SELECT DISTINCT Struct(star1: s1, star2: s2)
```

用来说明查询的结果是一些对象的集合，而这些对象的类型是Struct{star1: Star, star2: Star}。我们给出域的名字star1和star2来说明这个结构，而这些域的类型可以由变量s1和s2的类型推导出来。

例9.18 正如在9.3.4节中看到，常量的构造可以赋值到变量中去，这种方式类似于其他的编程语言。例如，考虑下面的赋值语句序列：

```
x = Struct(a:1, b:2);
y = Bag(x, x, Struct(a:3, b:4));
```

第一行赋给变量x一个值，其类型是：

```
Struct(a:integer, b:integer)
```

这个结构有两个整数值的域，名字分别为a和b。我们可以把这种类型的值表示成偶对，仅以整数作为成员，而不是域名a和b。因此，x的值可以表示成偶对（1，2）。第二行把y定义成一个包，该包的成员是与x有相同类型的结构。偶对（1，2）在这个包中出现了两次，而（3，4）出现了一次。 □

类或其他的类型可以通过*构造函数*（constructor function）创建自己的实例。一般情况下，类有许多不同形式的构造函数，选择哪种构造函数则取决于哪些属性被显性地初始化，以及哪些属性是用默认值。例如，方法不会被初始化，大部分的属性将得到初始值，而联系可能被初始化为一个空的集合，待以后再扩充。每个构造函数的名字都是该类的名字，依靠参数表中所用到的域名来区分。这些构造函数定义的细节取决于宿主语言。

例9.19 让我们考虑Movie对象的一个可能的构造函数。假设，这个函数的输入是属性title、year、length和ownedBy的值，输出是一个对象，该对象的域的值是输入值，还有一个影星的空集合。这样，如果变量mgm的值是MGM Studio对象，那么我们可以用下面的方式创建一个*Gone With the Wind*电影对象： 447

```
gwtw = Movie(title: "Gone With the Wind",
             year: 1939,
             length: 239,
             ownedBy: mgm);
```

这条语句将产生两个影响：

1. 它创建了一个新的Movie对象，成为Movies扩展的一部分。

2. 它使这个对象成为宿主语言的变量gwtw的值。

□

9.3.6 习题

习题9.3.1 把下面的常量赋给一个宿主语言变量x：

* a) 集合{1，2，3}。

b) 包{1，2，3，1}。

c) 表(1，2，3，1)。

d) 有两个成分的结构。第一个成分是名为a的集合{1，2}，第二个成分是名为b的包{1，1}。

e) 一个含有结构的包，每个结构都有两个名字分别为a和b的域。这个包含有三个结构，其值分别为(1，2)，(2，1)和(1，2)。

习题9.3.2 运用习题9.1.1和图9-2的ODL模式，写一段用OQL扩展的C++（或你选择的一种面向对象的宿主语言）程序，完成以下事情：

* a) 把型号1000的PC机对象赋给宿主语言变量x。

b) 把至少有64兆内存的手提电脑对象的集合赋给宿主语言变量y。

c) 把售价低于1500美元的PC的平均速度赋给宿主语言变量z。

! d) 找出所有的激光打印机，输出它们型号和价格的列表，接着是一条说明型号和最低价格的消息。

448 !! e) 输出一个表格，显示每个PC机厂商的产品的最低和最高价格。

习题9.3.3 在这个习题中，我们将使用习题9.1.3和图9-3的ODL模式。假设该模式下的四个类都有一个与该类同名的构造函数。这些构造函数为每个属性和单值联系取值，而对于多值联系则初始化为空值。对关联到其他类的单值联系，你可以假定一个宿主语言变量，变量的当前值就是相关的对象。创建下列对象，并把对象作为值赋给一个宿主语言变量。

* a) Maryland级战舰Colorado号，在1923年下水。

b) Lützow级战舰Graf Spee号，在1936年下水。

c) Malaya战争的一个后果是导致了战舰Prince of Wales号的沉没。

d) Malaya战争在1941年12月10号爆发。

e) Hood级的英国巡洋舰有8门15英寸大炮，排水量达41 000吨。

9.4 SQL中的用户定义类型

现在我们回头看看SQL-99是怎么把面向对象的多个特点结合到一起的。在ODL和OQL中已经讲述了这些特点。正因为这些对SQL的新扩展，一个符合这些标准的DBMS常常被认为是“对象关系的”。在4.5节中，我们遇到过很多抽象的对象关系概念。现在是研究这个标准的细节的时候了。

在OQL中没有具体的关系表示方法，只有一个结构的集合（或包）。然而，关系对SQL来说是居中心地位的，所以SQL中的对象仍以关系作为核心概念。在SQL中，ODL的类都变为用户定义类型（user-defined type，或UDT）。我们发现UDT类型用在两种截然不同的方面：

1. 一个UDT类型可以是一个表格的类型。
2. 一个UDT类型可以是某个表格中的某个属性的类型。

9.4.1 在SQL中定义类型

可以粗略地认为，SQL中用户定义类型的声明同ODL中类的声明是很相似的，但有一些差

别。首先，用来声明用户定义类型的关系的键是表格定义的一部分，而不是类型定义的一部分；也就是说，很多SQL的关系可以被声明为相同的（用户定义）类型，但有不同的键和其他限制。其次，在SQL中，我们不把联系看做是属性。一个联系必须被表示成一个独立的关系，正如在4.4.5节中讨论的那样。UDT定义的一个简单的形式如下： 449

1. 关键字CREATE TYPE，
2. 类型的名字，
3. 关键字AS。
4. 用小括号括起来、逗号隔开的属性及其类型的列表。
5. 用逗号隔开的方法的列表，这些方法包括参数类型和返回类型。

这样，类型*T*的定义形式为：

```
CREATE TYPE T AS <attribute and method declarations>;
```

例9.20 我们可以创建一个与图9-1中OQL例子的Star类相似的电影明星的类型。然而，我们不能直接在Star元组中把电影的集合表示成一个域。因此，我们将从Star元组的分量name和address开始讨论。

首先要注意，在图9-1中的address类型本身就是一个元组，含有分量street和city。因此，我们需要两个类型定义，一个用来定义地址，另一个定义影星。所需的定义如图9-10所示。

```
CREATE TYPE AddressType AS (
    street  CHAR(50),
    city    CHAR(20)
);

CREATE TYPE StarType AS (
    name    CHAR(30),
    address AddressType
);
```

图9-10 两个类型定义

AddressType类型的元组有两个分量，其属性是street和city。这两个分量的类型是长度分别为50和20的字符串。StarType类型的元组同样有两个分量。第一个的属性是name，其类型为30个字符的字符串；第二个分量的属性是address，它本身的类型就是一个AddressType的UDT类型，也就是一个含有street和city成员的元组。 □ 450

9.4.2 用户定义类型中的方法

方法的声明同8.2.1节中介绍的PSM中的函数类似，但方法没有与PSM过程类似的东西。也就是说，每个方法都返回某种类型的一个值。在PSM中函数的声明和定义是组合在一起的，而一个方法既需要一个满足自己类型定义的声明，又需要有一个用CREATE METHOD语句说明的单独的定义。

方法的声明类似于PSM的函数声明，只要用关键字METHOD代替关键字CREATE FUNCTION即可。但是通常SQL方法都没有参数，方法都作用在表的行上面，就像ODL的方法作用在对象上一样。在方法的定义中，如果需要，SELF可表示该元组本身。

例9.21 假设我们在图9-10中类型AddressType的定义中增加一个方法houseNumber，这个方法在住址的street分量中取得一部分数据。例如，street分量是“123 Maple St.”，那么，方法houseNumber将返回“123”。经过修改的类型定义为：

```
CREATE TYPE AddressType AS (
    street  CHAR(50),
    city    CHAR(20)
    )
    METHOD houseNumber() RETURNS CHAR(10);
```

我们可以看到在关键字METHOD后面，紧跟着方法的名字和一个用括号把参数和参数类型括起来的列表。在这个例子中，方法没有参数，但是括号仍然是必需的。如果方法有参数，那么这些参数会出现在列表中，后面紧跟参数的类型。例如，(a INT, b CHAR(5))。 □

在个别的情况下，我们需要定义方法，方法定义的一个简单形式包括：

1. 关键字CREATE METHOD。
2. 方法名、参数和参数类型以及返回语句，就像方法的声明一样。
3. 关键字FOR和方法声明所处的UDT的名字。
4. 方法体，使用与PSM函数体相同的语言来书写。

例如，我们可以如下定义例9.21中的方法houseNumber：

```
CREATE METHOD houseNumber() RETURNS CHAR(10)
FOR AddressType
BEGIN
    ...
END;
```

451

我们省略了方法体，因为从字符串address中分离出想要的子串并不是一件困难的事，即便在PSM中也是如此。

9.4.3 用UDT声明关系

在声明了一个类型之后，我们就可以声明一个或多个关系，这些关系中元组的类型就是刚声明过的那个类型。关系声明的形式类似于6.6.2节中所讲的那样，但是我们用

```
OF <type name>
```

来代替普通SQL表声明中的属性声明列表。表声明中的其他元素，如键、外键和基于元组的语义限制，如果需要都可以加进表的声明中，而且只作用到这张表上，不作用到UDT本身。

例9.22 我们可以通过下面的方式来声明MovieStar是一个元组类型为StarType的关系：

```
CREATE TABLE MovieStar OF StarType;
```

这样就使得表MovieStar拥有两个属性：name和address。第一个属性name是一个普通字符串，而第二个属性address的类型本身就是一个UDT，即AddressType类型。 □

通常，我们会为每个类型声明一个关系，认为这个关系是对应于该类型的类扩展（参见4.3.4节中的含义）。但是，对一个给定类型，声明多个关系或是不声明关系都是允许的。

9.4.4 引用

对于面向对象语言中的对象标识，SQL是通过*引用*（reference）来实现的。类型为UDT的表可以用一个*引用列*（reference column）作为它的标识符。例如，如果这个表有主键，引用列可以作为该表的主键，或者，引用列的值由DBMS单独生成和维护。我们将先了解怎样使用引用类型，之后再讲定义引用列的事情。

为了引用含有引用列的表中的元组，必须有一个可以引用其他类型 的属性。如果*T*是一个

UDT，那么REF(*T*)的类型就是对类型*T*的元组的引用。此外，可以为引用加上一个作用域（scope），该作用域就是被引用元组所在的关系的名字。因此，如果属性*A*的值是对关系*R*中元组的引用，而*R*是一个UDT类型*T*的表，那么*A*可以声明为： 452

A REF(*T*) SCOPE *R*

如果没有说明作用域，那么该引用就可以作用于类型*T*的任何一个关系。

例9.23 在MovieStar对象中，引用属性不足以记录影星主演的所有电影。但是，引用属性可以记录下每位影星主演的最佳电影。假设我们声明了一个关系Movie，关系的类型是UDT类型MovieType。我们将在图9-11中定义MovieType和Movie。下面是StarType一个新的定义，它包含了一个用来引用电影的属性bestMovie。

```
CREATE TYPE StarType AS (
    name      CHAR(30),
    address   AddressType,
    bestMovie REF(MovieType) SCOPE Movie
);
```

此时，如果关系MovieStar定义为包含上面内容的UDT，那么每个影星元组都会有一个分量来引用一个Movie元组——该影星的最佳电影。 □

接下来，我们将调整类似例9.23的表，加进一个引用列。这样的表称为可引用（referenceable）的。在一个表的类型为UDT类型的CREATE TABLE语句中（见9.4.3节），可以添加如下形式的子句：

REF IS <attribute name> <how generated>

"attribute name"是赋给的列名，用作元组的"对象标识符"。而"how generated"子句可以是下面任何一种常见形式：

1. SYSTEM GENERATED，这表示DBMS负责维护每个元组在该列有惟一的值。
2. DERIVED，这表示DBMS将使用关系的主键为该列产生惟一的值。

例9.24 图9-11说明了怎样声明UDT类型MovieType和关系Movie，以使得Movie是可以被引用的。第(1)到(4)行是对这个UDT的声明，第(5)到(7)行把关系Movie定义为这种UDT类型。注意，我们在第(7)行声明了title和year，两者一起作为关系Movie的键。

在第(6)行当中我们可以看到，Movie"标识"列的名字是movieID。这个属性将自动成为title、year以及inColor之后的Movie的第四个属性。和其他任何属性一样，该属性也可以用在查询中。

```
1)  CREATE TYPE MovieType AS (
2)      title   CHAR(30),
3)      year    INTEGER,
4)      inColor BOOLEAN,
    );

5)  CREATE TABLE Movie OF MovieType (
6)      REF IS movieID SYSTEM GENERATED,
7)      PRIMARY KEY (title, year)
    );
```

图9-11 创建可被引用的表

第(6)行还说明，每当一个新的元组插入到Movie的时候，DBMS都负责为其生成一个

453 movie-ID的值。如果我们用“DERIVED”代替“SYSTEM GENERATED”，那么在产生新的元组时，系统将由该元组的主键属性title和year的值计算出一个新值，作为新元组movieID的值。□

例9.25 现在，让我们看看怎样使用引用来表示电影和影星之间的多对多联系。先前，我们用关系来表示这种联系，例如用关系StarsIn，它包含以Movie和MovieStar作为键的元组。作为一种选择，我们重新定义StarsIn，使它可以引用这两个关系中的元组。

首先，我们需要重新定义MovieStar，使它成为一个可引用的表：

```
CREATE TABLE MovieStar OF StarType (
    REF IS starID SYSTEM GENERATED;
)
```

然后，我们可以声明关系StarsIn包含两个属性，即两个引用，其中一个引用电影的元组，另一个引用影星的元组。下面是这个关系的一个直接的定义：

```
CREATE TABLE StarsIn (
    star    REF(StarType) SCOPE MovieStar,
    movie   REF(MovieType) SCOPE Movie
);
```

也可以选择另一个方式，我们根据上述内容定义一个UDT类型，然后把StarsIn声明为该类型的一个表。□

9.4.5 习题

习题9.4.1 写出以下类型的类型声明：

454 a) NameType，包含名字、中间名、姓以及一个头衔。

* b) PersonType，包含人名和对其父亲、母亲对象的引用。在声明中必须使用(a)中的类型。

c) MarriageType，包含结婚日期和对丈夫、妻子对象的引用。

习题9.4.2 在适当的地方使用类型声明和引用属性，重新设计习题5.2.1中产品数据库的模式。特别是要在关系PC、Laptop和Printer中，把model属性改为对其Product元组的引用。

! **习题9.4.3** 在习题9.4.2中，我们假定表PC、Laptop和Printer中的型号是对表Product中元组的引用。是否也可以把Product中的model属性改为对该类型产品关系的元组的引用？为什么？

* **习题9.4.4** 在适当的地方使用类型声明和引用属性，重新设计习题5.2.4中战舰数据库的模式。习题9.1.3的模式会提示在什么地方引用属性。试找出多对一联系，并用一个引用类型的属性来表示它们。

9.5 对象关系数据上的操作

从前面的章节中看到，在用UDT声明或含有UDT类型属性的表上，可以运用所有适当的SQL操作。还可使用一些全新的操作，如引用跟随。但是我们熟悉的某些操作，特别是访问或者修改类型是UDT的列，将涉及到新的语法。

9.5.1 引用的跟随（Following Reference）

假设x类型为REF(T)的一个值，那么x引用类型T的某个元组t。通过以下两种方法，我们

可以获得元组t本身，或者t中的成员：

1. 操作符->本质上和在C语言中的含义相同。也就是说，如果x是对元组t的一个引用，且a是t的一个属性，那么x->a就是元组t中属性a的值。

2. DEREF操作符作用于一个引用，并且生成被引用的元组。

例9.26 使用例9.25中的关系StarsIn查找Mel Gibson扮演过的电影。回顾一下，该关系模式如下：

```
StarsIn(star, movie)
```

455

其中，star和movie分别是对MovieStar和Movie中元组的引用。可能的查询如下：

```
1)  SELECT DEREF(movie)
2)  FROM StarsIn
3)  WHERE star->name = 'Mel Gibson';
```

在第(3)行中，表达式star->name生成MovieStar元组中的name分量的值，而这个MovieStar元组是被任意给定的一个StarsIn元组的分量star所引用的。因此，WHERE子句标识这样一些StarsIn元组，它们的分量star都是对name值为Mel-Gibson的MovieStar元组的引用。第(1)行则生成那些相应的StarsIn元组中movie分量所引用的电影元组。所有三个属性——title、year和inColor——都将出现在输出结果中。

注意我们可以用下面的语句代替第(1)行：

```
1)  SELECT  movie
```

但是，如果我们这样做的话，将会得到一些无用的数据。这些数据是系统产生的，被用作这些元组的惟一的内部标识符。在被引用的元组中，我们是不会看到这些信息的。 □

9.5.2 访问UDT类型元组的属性

当我们定义一个含有UDT的关系的时候，必须把元组看做是单独的对象，而不能看做是以UDT属性为成员的列表。这里有一个恰当的例子，我们来考虑图9-11中声明的关系Movie。这个关系有一个UDT类型MovieType，它有三个属性：title、year和inColor。但是，Movie当中的元组t只有一个分量，而不是三个。这个分量就是这个对象本身。

如果我们深入到这个对象内部，可以从中取得类型MovieType的三个属性的值，并且可以使用该类型定义的任何方法。但是，我们有时访问的属性不属于这个元组本身。这样，UDT都为自己的每一个属性隐含地定义了一个*观测方法*(observer method)。属性x的观测方法的名字是x()。我们可以像使用该UDT的任何其他方法一样地使用这个方法，用点号把它接到计算该类型对象的表达式后面。因此，如果t是类型T的变量，x是T的一个属性，那么$t.x$()就是t代表的元组（对象）中x的值。

例9.27 让我们从图9-11的关系Movie中找出电影*King Kong*的年份。一种解决方案是：

```
SELECT m.year()
FROM Movie m
WHERE m.title() = 'King Kong';
```

456

虽然元组变量m出现在这里不是必要的，但是我们需要一个变量，它的值是类型为MovieType（关系Movie的UDT）的对象。WHERE子句的条件表达式把常量'King Kong'与m.title()进行比较。后者就是类型MovieType的属性title的观测器方法。同样的，SELECT子句中的值用m.year()来表示，这个表达式把属性year的观测器方法作用到对象m

上。 □

9.5.3 生成器和转换器函数

为了生成UDT中的数据，或者改变UDT对象的成员，我们可以使用两类方法。每当定义一个UDT时，这些方法连同观测器方法一起自动创建。这两类方法是：

1. *生成函数*（generator method）。这个方法的名字就是类型名，并且没有参数。它还有一个特别的性质是调用不需要作用到某个具体的对象上。也就是说，如果T是一个UDT，那么$T()$将返回一个类型为T的对象，而该对象的各个成分是没有值的。

2. *转换函数*（mutator method）。UDT类型T的每个属性x，都有一个转换器方法$x(v)$。当这个方法作用到T的一个对象上时，它把该对象的属性x的值改为v。注意，一个属性的转换函数和观测函数的名字都是该属性的名字，区别仅在于转换函数有一个参数。

例9.28 我们将写一段PSM的过程，取街道、城市和名字作为参数，调用正确的生成函数和转换函数来构造一个对象，并把它插入到关系MovieStar（依照例9.22，类型是StarType）中。回顾一下例9.20，StarType类型的对象有一个字符串类型的成分name，但成分
457 address本身就是类型为AddressType的一个对象。过程InsertStar如图9-12所示。

```
1)  CREATE PROCEDURE InsertStar(
2)      IN s CHAR(50),
3)      IN c CHAR(20),
4)      IN n CHAR(30)
    )
5)  DECLARE newAddr AddressType;
6)  DECLARE newStar StarType;

    BEGIN
7)      SET newAddr = AddressType();
8)      SET newStar = StarType();
9)      newAddr.street(s);
10)     newAddr.city(c);
11)     newStar.name(n);
12)     newStar.address(newAddr);
13)     INSERT INTO MovieStar VALUES(newStar);
    END;
```

图9-12 创建和存储StarType对象

第(2)到(4)行引进参数s、c和n，分别提供街道、城市和影星名字。第(5)到(6)行声明了两个局部变量，两者的类型都是关系MovieStar所涉及的UDT类型。在第(7)行和第(8)行，我们创建了这两个类型的空对象。

第(9)行和第(10)行从过程的参数中取得两个真正的值，并赋给对象newAddr，这些参数提供街道和城市名。类似地，第(11)行把参数n作为对象newStar中成分name的值。而第(12)行把整个newAddr对象作为newStar中成分address的值。最后，第(13)行把构造好的对象插入到关系MovieStar中。注意，与通常一样，一个以UDT为类型的关系只有一个单独的成分，即使这个成分有多个属性（比如这个例子中的name和address）也是如此。

要将一个影星插入MovieStar中，我们可以调用InsertStar过程。下面是一个例子：

```
InsertStar('345 Spruce St.', 'Glendale', 'Gwyneth Paltrow');
```

□

如果DBMS提供或者自己创建一个生成函数，使该生成函数以UDT属性作为输入，并返回一个合适的对象，那么把对象插入含有UDT的关系中将变得更简单。例如：如果我们已经有函数AddressType(s, c)和函数StarType(n, a)，它们返回指定类型的对象，那么就可以在例9.28的末尾使用如下的INSERT语句进行插入操作：

```
INSERT INTO MovieStar VALUES(
    StarType('Gwyneth Paltrow',
        AddressType('345 Spruce St.', 'Glendale')));
```

9.5.4 UDT类型联系的排序

某些UDT类型的对象本质上是抽象的。在这个意义上，没法对同一个UDT的两个对象进行比较，不管是测试它们是否“相等”，还是测试其中一个比另一个小。即使两个对象的所有成分都相同，也不会认为这两个对象是相等的，除非我们告诉系统把它们看做是相等的。同样，也无法对一个含有UDT的关系的元组进行排序，除非我们定义一个函数，指明该UDT的两个对象中哪一个先于另一个。 458

然而，SQL中有许多操作，需要进行相等关系测试或同时需要进行相等关系测试和小于关系测试。例如，如果不能指出两个元组是否相等，那么我们将无法排除重复的元组。如果无法对一个UDT做相等关系测试，那么我们就不能对一个UDT类型的属性进行分组。我们在WHERE子句中不能使用ORDER BY子句或比较符‘<’，除非我们能对两个元素进行比较。

为了说明一个排序或比较操作，SQL允许我们用CREATE ORDERING语句来声明一个UDT。其中有许多种声明的格式，这里仅列出最简单的两种：

1. 语句

```
CREATE ORDERING FOR T EQUALS ONLY BY STATE;
```

指明UDT类型T的两个成员，如果它们相应的分量是一样的，那么就可以认为是相等的。这里在UDT类型T的对象上没有定义‘<’操作。

2. 下列语句

```
CREATE ORDERING FOR T
ORDERING FULL BY RELATIVE WITH F
```

指明了6种比较：<、<=、>、>=、=和<>，都可以作用到UDT类型T的对象上。为了说明对象x_1和x_2是怎样作比较的，我们把函数F作用到这些对象上。函数F的实现必须这样写：如果$x_1 < x_2$，那么$F(x_1, x_2) < 0$；而$F(x_1, x_2) = 0$表示$x_1 = x_2$；$F(x_1, x_2) > 0$表示$x_1 > x_2$。如果将ORDERING FULL换为EQUALS ONLY，那么$F(x_1, x_2) = 0$表示$x_1 = x_2$，而$F(x_1, x_2)$的其他值则表示$x_1 \neq x_2$。在这种情况下，不能用‘<’符号进行比较。

例9.29 让我们考虑例9.20中UDT类型StarType的一个可能的排序。如果仅要进行等于操作，可以如下声明：

```
CREATE ORDERING FOR StarType EQUALS ONLY BY STATE;
```

该语句指明两个StarType类型的对象是相等的，当且仅当它们的名字是相同的字符串，而且地址是相同的UDT类型AddressType对象。

这里的问题在于，除非我们定义了AddressType的排序，否则该类的对象甚至不能等于自身。因此，我们至少还要测试AddressType对象是否相等。一个简单的做法是：声明两个AddressType对象是相等的，当且仅当这两个对象的street和city是相同的。做法如下： 459

```
CREATE ORDERING FOR AddressType EQUALS ONLY BY STATE;
```

另一种做法是：我们也可以定义AddressType对象的一个完整排序。一个合理的地址排序是，先以城市的字母序进行排序，有相同城市的地址再通过街道名的字母序进行排序。为此，我们必须定义一个函数AddrLEG，该函数以两个AddressType对象作为参数，比较后返回一个负值、零或正值，分别表示第一个小于、等于或大于第二个对象。我们可以做如下声明：

```
CREATE ORDERING FOR AddressType
ORDER FULL BY RELATIVE WITH AddrLEG
```

图9-13给出了函数AddrLEG。注意，如果我们可以到达第(7)行，那么两个city成分是相同的，因此可以接着比较street成分。同样地，如果我们到达第(9)行，那么剩下的惟一可能就是两个city是相同的，而且在字母序上，第一个street排在第二个的前面。 □

```
1)  CREATE FUNCTION AddrLEG(
2)      x1 AddressType,
3)      x2 AddressType
4)  ) RETURNS INTEGER

5)  IF x1.city() < x2.city() THEN RETURN(-1)
6)  ELSEIF x1.city() > x2.city() THEN RETURN(1)
7)  ELSEIF x1.street() < x2.street() THEN RETURN(-1)
8)  ELSEIF x1.street() = x2.street() THEN RETURN(0)
9)  ELSE RETURN(1)
    END IF;
```

图9-13 地址对象的一个比较函数

9.5.5 习题

习题9.5.1 使用例9.25中的StarsIn关系以及通过StarsIn可以访问到的Movie和MovieStar关系，写出以下的查询：

* a) 查找电影*Ishtar*中所有影星的姓名。

*! b) 查找出演影星中至少有一个住在Malibu的所有电影的片名和年份。

460 c) 查找影星Melanie Griffith演过的电影（MovieType类型的对象）。

! d) 查找至少有5位影星出演的电影（包括名字与年份）。

习题9.5.2 使用你在习题9.4.2中采用的模式，写出以下的查询，切记随时使用引用。

a) 查找硬盘大于60 GB的PC机的制造商。

b) 查找生产激光打印机的制造商。

! c) 创建一张表，对每一款手提电脑，给出同一个制造商生产的、具有最快处理速度手提电脑的型号。

习题9.5.3 使用习题9.4.4中的模式，写出以下的查询。切记随时使用引用，并避免含有连接操作（即子查询或FROM子句中含有不止一个元组变量）：

* a) 查找排水量大于35 000吨的舰只。

b) 查找至少有一艘战舰沉没的战役。

! c) 查找在1930年之后下水的战舰的类别。

!! d) 查找至少有一艘美国舰船受创的战斗。

习题9.5.4 假设图9-13的AddrLEG函数是可用的，写一个适当的函数来比较StarType类型的对象，并把该函数声明为StarType对象排序的基准。

*! **习题9.5.5** 写一个过程，使该过程以影星名字为参数，并且删除StarsIn与MovieStar中含有该影星的元组。

9.6 小结

- OQL中的select-from-where语句：OQL提供与SQL类似的 select-from-where表达式。在FROM子句中，我们可以声明集内任意变量，包括类扩展（类似于关系）和以对象的属性作为值的集。
- 公用的OQL操作符：OQL提供与SQL中类似的操作符，如全称量词、存在量词、IN、并、交、差和聚集操作符。但是，聚集操作始终在一个集上进行，而不在关系的某一列上进行。
- OQL的分组操作：与SQL类似，OQL在select-from-where语句中也提供GROUP BY子句。但是在OQL中，每一组的对象的集是通过一个称为partition的域名进行显式访问的。 461
- 从OQL集合中提取元素：我们可以使用操作符ELEMENT从一个单元素集中提取出该成员。如果集含有多个成员，那么先在select-from-where子句中使用ORDER BY子句将集转化为一个列表；然后利用宿主语言中的循环语句依次访问每个元素。
- SQL中的用户定义类型：SQL中的对象关系功能都是围绕着UDT（用户定义类型）展开的。这些类型的声明与表声明类似，是通过列出它们的属性和其他信息进行的。此外，对UDT类型可以声明方法。
- 有UDT类型的关系：我们可以声明一个关系含有某个UDT类型，而不是声明该关系的各个属性。如果这样做，那么该关系的元组将有一个UDT对象的成分。
- 引用类型：一个属性的类型可以是一个对UDT的引用，这些属性实质上是指向该UDT类的对象的指针。
- UDT的对象标识：当创建一个类型为UDT的关系时，我们将为每一个元组声明一个属性作为该元组的“对象标识”。该成分是指向这个元组本身的指针。与面向对象系统不同，用户可以访问该OID列，尽管它没有什么意义。
- 访问UDT中的分量：SQL为UDT的每一个属性提供了观测器和转化器函数。当它们作用于UDT类的任何对象时，这些函数将分别返回或修改给定的属性值。

9.7 参考文献

OQL的参考文献与ODL是一样的可参见[1]。可以从第6章给出的书目中得到关于面向对象特征的资料。

1. Cattell, R. G. G. (ed.), *The Object Database Standard: ODMG-99*, Morgan-Kaufmann, San Francisco, 1999.

462

第10章 逻辑查询语言

与我们在5.2节介绍过的代数相比较，有些关系模型查询语言更像是一种逻辑。然而基于逻辑的语言对于许多程序员来说似乎很难掌握，所以我们把查询语言的逻辑研究放在最后。

我们下面要介绍Datalog，一种为关系模型设计的形式最为简单的逻辑。Datalog的非递归形式和传统的关系代数有同样的能力。然而如果允许递归，我们可以用Datalog表示出SQL2所不能表示的查询（除非添加例如PSM的过程编程）。我们还考虑了因允许否定递归而带来的复杂性，最后我们介绍Datalog所提供的实现在最新的SQL-99标准中、更有意义的递归的方法。

10.1 一种关系逻辑

我们可以利用代数，将查询语言抽象表示为一种逻辑形式。逻辑查询语言Datalog（database logic）由if-then规则组成。这些规则表示：从特定关系特定元组的组合中可以推断出另一关系中的某个元组，或是一个查询结果中的某个元组。

10.1.1 谓词和原子

关系在Datalog中由谓词表示。每个谓词拥有固定数目的参数，一个谓词和它的参数一起被称为原子。原子的语法就像传统编程语言中的函数调用。例如：$P(x_1, x_2, \ldots, x_n)$即是一个由谓词P和参数$x_1, x_2, \ldots, x_n$组成的原子。

实质上，谓词就是一个返回布尔量的函数名。如果R是一个包含n个具有固定顺序的属性的关系，那么我们也应该用R作为对应这个关系的谓词名。如果$(a_1, a_2, \ldots, a_n)$是满足R的元组，那么原子$R(a_1, a_2, \ldots, a_n)$的值为TRUE，否则原子的值为FALSE。 463

例10.1 如关系R是：

A	B
1	2
3	4

那么$R(1, 2)$为true，$R(3, 4)$也为true。当x和y为其他任意值时，$R(x, y)$都为false。 □

谓词可以用变量和常量作为参数。如果一个原子以变量作为它的一个或多个参数，那么它是一个以变量为参数并返回TRUE或FALSE的布尔值函数。

例10.2 如果R是例10.1的谓词，那么函数$R(x, y)$表示：对任意x和y，元组(x,y)是否在关系R中。对于在例10.1中给出的特定例子，当

1. $x=1$ 且 $y=2$ 或
2. $x=3$ 且 $y=4$

$R(x, y)$返回TRUE，否则返回FALSE。另举一例，若$z=2$，原子$R(1, z)$返回TRUE，否则返回FALSE。 □

10.1.2 算术原子

在Datalog中另有一种很重要的原子：算术原子。这种原子表示两个算术表达式之间的比较，例如$x<y$或$x+1 \geq y+4 \times z$。相对地，我们把在10.1.1节中介绍的原子称为关系原子。它们都

是“原子”。

注意，算术原子和关系原子都将所有出现在原子中的变量值作为参数，并且都返回一个布尔值。实际上，诸如“<”或“≥”之类的算术比较符号象关系名一样地包含所有满足它的元组。因此，我们可以将关系“<”看做第一组元小于第二组元的所有元组，如(1,2)和(-1.5,65.4)。然而要记住数据库中的关系总是有限的，而且通常是随着时间变化的。相反地，算术比较关系，
464 如“<”，却是无限且不变的。

10.1.3 Datalog规则和查询

与5.2节经典关系代数中所述类似的操作在Datalog中称做规则，它包括：

1. 一个称为头部的关系原子。

2. 符号←，位于头部之后，我们经常读作“if”。

3. 一个主体部分，位于“←”之后，由一个或多个称为子目标的原子组成。原子可以是关系原子或算术原子。子目标之间由AND连接，任何子目标之前都可随意添加逻辑算子NOT。

例10.3 Datalog规则

```
LongMovie(t,y) ← Movie(t,y,l,c,s,p) AND l ≥ 100
```

定义了“long” movie的集合是时间长于100分钟的。它使用了我们的标准关系movie，其模式是：

```
Movie(title, year, length, inColor, studioName, producerC#)
```

这个规则的头部是原子LongMovie(t, y)。规则的体部包括如下两个子目标：

1. 第一个子目标包括谓词Movie和六个参数，对应于Movie关系的六个属性。每个参数都对应不同的变量：t是title组元，y是year组元，l是length组元，依次类推。我们可以把子目标看做：“把(t, y, l, c, s, p)作为Movie关系当前实例的一个元组。”更确切地说，当这六个变量就是一个Movie元组的六个组元的值时，Movie(t, y, l, c, s, p)为真。

2. 第二个子目标，$l \geq 100$。当一个Movie元组的length组元不小于100时该目标为真。

这个规则可以完整地说成：当我们发现在Movie中有一个元组满足以下条件时LongMovie(t, y)为真：

a) t和y是前两个组元（作为title和year）

b) 第三个组元l（作为length）至少为100

c) 组元四到六为任意值。

注意，这个规则等价于关系代数中的“赋值语句”：

465
$$\text{LongMovie} := \pi_{title,year}(\sigma_{length \geq 100}(\text{Movie}))$$

其右边是一个关系代数表达式。 □

匿名变量

Datalog规则中经常会有一些只出现一次的变量。这些变量所用的名字互不相干的。只有当一个变量出现不止一次时我们才注意它的名字，我们会发现这是同一个变量的第二次出现或更多次之后的重新出现。因此，为了方便，我们将允许把下划线_作为原子的参数，代替仅在那里出现的变量。多个_的出现代表不同的变量，而不是同一个变量。举例来说，例10.3的规则可以写作：

```
LongMovie(t,y) ←Movie(t,y,l,_,_,_) AND l≥100
```

这三个仅出现一次的变量c，s，p都被下划线代替。因为其他变量都在规则中出现两次，我们无法替换它们。

Datalog中的查询是一个或多个规则的组合。如果只有一个关系出现在规则头部，那么这个关系的值就是查询的结果。因此在例10.3中，LongMovie就是查询的结果。如果规则头部有不止一个关系，那么除一个是查询结果外，其余的用来辅助定义查询结果。我们必须指定哪一个关系是所期望的查询结果，或通过给关系命名，如Answer，来指定。

10.1.4 Datalog规则的意义

例10.3引伸出Datalog规则的意义。更确切地说，假设规则的变量涉及所有可能的值，只要这些变量的值使得所有子目标为真，那么我们即得到对应于这些变量的规则头部的值，并可把结果元组加入到头部谓词的关系中，关系的谓词可由规则头部得到。

举例来说，我们假设例10.3中的六个变量涉及所有可能值。仅当(t, y, l, c, s, p)的值以顺序构成Movie的一个元组时，这些值的组合才能使全部子目标为真。而且，子目标$l \geq 100$也必须为真，元组长度组元的值l至少为100。当我们发现这样一种值的组合时，则把元组(t,y)放在规则头部的LongMovie关系中。

然而我们在规则中使用变量还是有限制的，因此一条规则的结果是一个有限的关系，而且包含算术子目标和否定子目标（前面有NOT算子）的规则的意义是很明显的。我们称以下条件为安全条件： 466

- 每个在规则中任意位置出现的变量都必须出现在某个非否定的关系子目标中。

尤其是，任何在规则头部、否定关系子目标或算术子目标中出现的变量，也必须出现在一个非否定的关系子目标中。

例10.4 考虑例10.3中的规则

```
LongMovie(t,y) ← Movie(t,y,l,_,_,_) AND l ≥ 100
```

第一个子目标是非否定的关系子目标，它包含了所有在规则中出现的变量，尤其是在头部中出现的两个变量t和y都在体部的第一个子目标中出现。同样地，变量l在一个算术子目标中出现，但它也出现在第一个子目标中。 □

例10.5 下面的规则有三处不安全：

```
P(x,y) ← Q(x,z) AND NOT R(w,x,z) AND x<y
```

1. 变量y出现在头部但不在任何非否定关系在中出现。注意，y出现在算术子目标$x<y$中并不能把y的可能值限定在一个有限集合内。我们若取值a、b、c分别对应w，x，z以满足前两个子目标，则头部的关系P中满足$d>b$的元组(b, d)有无限多个。

2. 变量w出现在一个否定的关系子目标中，而不在非否定的关系子目标中。

3. 变量y出现在一个算术子目标中，但不在非否定的关系子目标中。

因此，这不是一条安全规则，不能用在Datalog中。 □

还有另一种方法定义规则的意义。不去考虑所有可能的变量赋值，而是考虑对应于每个非否定关系子目标关系的元组集合。如果某种元组的赋值对每个非否定关系子目标都是一致的，也就是说对同一个变量的每次出现都赋同一个值，则考虑对规则的所有变量赋值。注意，这是因为规则是安全的，一个变量赋一个值。 467

对每种一致性赋值，我们考虑否定关系子目标和算术子目标，看看变量的赋值是否使得它们都为真。记住，若一个否定关系子目标的原子为假，该子目标为真。如果所有子目标为真，那么我们则知道了在这种变量赋值下的规则头部的元组。这个元组被加入谓词头部所在的关系中。

例10.6 考虑Datalog规则

```
P(x,y) ← Q(x,z) AND R(z,y) AND NOT Q(x,y)
```

设关系Q包含两个元组(1, 2)和(1, 3)。设关系R包含元组(2, 3)和(3, 1)。这里有两个非否定的关系子目标Q(x, z)和R(z, y)，所以我们必须分别考虑这两个子目标的关系Q和R中元组的所有赋值组合。图10-1中的表格考虑了所有四种组合。

	Q(x,z)的元组	R(z,y)的元组	一致性赋值?	NOT Q(x,y)为真?	头部的结果
1)	(1,2)	(2,3)	是	不是	—
2)	(1,2)	(3,1)	不是; $z=2,3$	无关	—
3)	(1,3)	(2,3)	不是; $z=3,2$	无关	—
4)	(1,3)	(3,1)	是	是	$P(1,1)$

图10-1 $Q(x,z)$和$R(z,y)$中元组的所有可能赋值

图10-1中的第二和第三选项是不一致的。每个选项都赋给变量z两个不同的值。因此，我们不再考虑这些元组的赋值。

第一个选项中，子目标$Q(x, z)$被赋值为元组(1,2)而子目标$R(z, y)$被赋值为元组(2,3)，这是一致性赋值，x、y和z分别被赋值为1、3、2。我们下面开始检验其他不是非否定关系的子目标。只有一个：NOT Q(x,y)。对这一赋值，该子目标成为NOT Q(1,3)。由于(1,3)是Q的一个元组，这个子目标为假，对于这个元组－赋值(1)可以生成没有头部的元组。

最后的选项是(4)。该选项是一致性赋值：x、y、z分别被赋值为1、1、3。子目标NOT $Q(x,y)$的值为NOT $Q(1,1)$。由于(1,1)不是Q的元组，这个子目标为真。这样我们根据这一变量赋值计算头部$P(x,y)$得到$P(1,1)$，所以这个元组(1,1)在关系P中。既然我们已经讨论过所有可能的
468 赋值，那么这个元组就是P中的惟一元组。 □

10.1.5 扩展谓词和内涵谓词

对以下两者加以区分是必要的：

- *扩展谓词*：这种谓词的关系存放在数据库中。
- *内涵谓词*：这种谓词的关系是由一个或多个Datalog规则计算出来。

这两种谓词之间的区别等同于关系代数表达式的操作数与关系代数表达式计算出的关系之间的区别。前者，关系代数表达式操作数是“可扩展的”（由它的扩展，也就是“关系的当前实例”来定义）；后者，关系代数表达式计算出的关系可以是最终结果也可以是对应某些子表达式的中间结果，这些关系是“内涵的”（由程序员的意图而决定）。

当谈论Datalog规则时，我们也应当提及扩展谓词和内涵谓词所对应的关系。我们用缩写IDB (intensional database)来表示内涵谓词和它对应的关系。同样地，我们用EDB(extensional database)来表示扩展谓词或关系。

那么在例10.3中，Movie是一个EDB关系，由它的扩展来定义。谓词Movie同样是一个EDB谓词。关系和谓词LongMovie都是内涵的。

一个EDB谓词不可能出现在规则头部，但它可以出现在规则体部。IDB谓词可以出现在规则的头部和体部，或者同时出现在这两个位置。利用头部为同一谓词的多个规则建立一个关系的方法也被普遍使用。我们将在例10.10中描述这个合并两个关系的方法。

通过使用一系列的内涵谓词，我们可以用EDB关系逐步建立更为复杂的功能。这个过程近似于用几个算子建立关系代数表达式。

10.1.6 Datalog规则应用于包

Datalog本质上是关于集合的逻辑。然而只要没有否定的关系子目标，关系是集合的情况下计算Datalog规则的方法对于关系是包时同样适用。当关系是包时，使用我们在10.1.4节给出的计算Datalog规则的第二种方法在概念上更加简单。这个方法是对每个非否定的关系子目标，找出它的谓词关系的所有元组。如果某种元组选择对每个子目标给出了一致性变量赋值，并且 469
算术子目标都为真[⊖]，那么我们能够知道这个赋值的规则头部是什么样的。这个结果元组被放入头部关系之中。

既然我们现在要处理包，我们并不排除头部关系中元组的重复现象。而且，当我们考虑子目标的所有元组的组合时，在一个子目标的关系中重复出现n次的元组被作为该子目标元组与其他子目标元组合并了n次来考虑。

例10.7 考虑规则

```
H(x,z) ← R(x,y) AND S(y,z)
```

其中关系$R(A,B)$包括以下元组：

A	B
1	2
1	2

而$S(B,C)$包括元组：

B	C
2	3
4	5
4	5

当第一个子目标被赋值为关系R中的元组(1,2)，且第二个子目标被赋值为关系S中的元组(2,3)时，我们惟一一次得到对所有子目标的一致赋值（也就是y变量的赋值对每个子目标都是一样的）。由于(1,2)在R中出现两次，(2,3)在S中出现一次，那么有两种元组赋值使得变量x=1, y=2, z=3。头部的元组(x,z)是两个(1,3)。所以元组(1,3)在头部关系H中出现两次，并且没有其他元组出现。也就是说，关系

1	3
1	3

就是这个规则定义的头部关系。更普遍地，若元组(1,2)在R中出现n次，元组(2,3)在S中出现m次，那么元组(1,3)在H中出现nm次。 □

如果一个关系由若干规则定义，那么结果是每个规则生成的元组的包联合（bag-union）。

⊖ 注意规则中不能有任何否定的关系子目标。对于在包模式下，任意拥有否定关系子目标的Datalog规则的含义还没有明确的定义。

470 **例10.8** 考虑一个由以下两条规则定义的关系H

```
H(x,y) ← S(x,y) AND x>1
H(x,y) ← S(x,y) AND y<5
```

其中关系$S(B,C)$和在例10.7中一样，也就是S={(2,3)，(4,5)，(4,5)}。由于S中三个元组的第一个组元都大于1，因此第一条规则把这三个元组依次放入H中。由于(4,5)不满足条件$y<5$，第二条规则只把元组(2,3)放入H中。这样，结果关系H有元组(2,3)的两个拷贝和元组(4,5)的两个拷贝。 □

10.1.7 习题

习题10.1.1 用Datalog写出练习5.2.1的所有查询。只能使用安全规则，但可以使用与复杂关系代数表达式的子表达式对应的若干IDB谓词。

习题10.1.2 用Datalog写出练习5.2.4的所有查询。只能使用安全规则，可以使用若干IDB谓词。

!! **习题10.1.3** 我们对Datalog规则的安全性要求是充分保证：如果关系子目标的谓词是有限关系，那么头部谓词是一个有限关系。但是这个要求太严格了。给出一个Datalog规则的例子，使其虽然违反这个条件，但只要我们将其中关系谓词赋值为有限关系，头部关系就始终是有限的。

10.2 从关系代数到Datalog

第5.2节中的每个关系代数算子都可以由一条或多条Datalog规则模拟。在这一节我们依次考虑每个算子。我们将考察如何合并一些Datalog规则来模拟复杂代数表达式。

10.2.1 交

两个关系的交集可由以这两个关系为子目标且对应参数为同一变量的一条规则来表示。

例10.9 让我们使用关系R:

姓 名	地 址	性别	生 日
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

471 和关系S:

姓 名	地 址	性别	生 日
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

作为例子，它们的交集由以下Datalog规则计算出来：

```
I(n,a,g,b) ← R(n,a,g,b) AND S(n,a,g,b)
```

这里，I是一个IDB谓词，我们应用上述规则后它的关系是$R \cap S$。也就是说，如果一个元组(n,a,g,b)要使得两个子目标都为真，那么该元组必须同时在R和S中。 □

10.2.2 并

两个关系的并是由两条规则建立。每条规则都有一个对应于其中一个关系的原子作为它的惟一子目标，并且这两条规则头部都有同一IDB谓词。 每条规则头部的参数都和规则子目标的参数完全一样。

例10.10 为得到例10.9中两个关系R和S的并，我们使用两个规则

```
1. U(n,a,g,b) ← R(n,a,g,b)
2. U(n,a,g,b) ← S(n,a,g,b)
```

规则(1)表示R中每个元组都是IDB关系U中的元组。类似地，规则(2)表示S中每个元组都在关系U中。这样，这两个规则一起暗示在$R \cup S$中的每个元组都在U中。如果我们不再加入头部为U的规则，那么其他任何元组都不能加入关系U中，这样我们可以确认U就是$R \cup S$[⊖]。注意，与交的建立只对集合操作不同，上述规则对可以对集合或包进行操作，这取决于我们怎么解释这两条规则结果的并。我们在不特别说明时指的是集合的并。 □

10.2.3 差

关系R和S的差集可通过一条包含否定子目标的规则计算出来。也就是说，规则中非否定的子目标含有谓词R而否定子目标包含谓词S。这些子目标和头部相应参数的变量都是相同的。

例10.11 如果R和S都是例10.9中的关系，那么规则

```
D(n,a,g,b) ← R(n,a,g,b) AND NOT S(n,a,g,b)
```

定义的D就是关系$R-S$。 □ 472

变量对规则的局部作用

注意我们选择的规则中变量名是任意的，与其他规则中使用的变量无关。无关的原因是每条规则都是独立计算，并且向其头部关系提供元组，而与其他规则无关。例如，我们将例10.10中的第二条规则替换成

```
U(w,x,y,z) ← S(w,x,y,z)
```

而同时第一条规则保持不变，这两条规则仍是计算R和S的并。但需注意，如果在一条规则中用一个变量a代替另一个变量b，我们必须在该规则中用a替换所有出现的b。而且，我们所选择的替换变量a不可以是该规则中已经出现的变量。

10.2.4 投影

为了计算关系R的投影，在我们要使用的规则中，只有一条谓词为R的子目标。这个子目标的参数是不同的变量，每个代表关系的一个属性。规则头部有一个原子，其参数按照期望顺序对应于投影列表的属性。

例10.12 假设我们要将关系

```
Movie(title, year, length, inColor, studioName, producerC#)
```

投影到它的前三个属性`title`、`year`和`length`上。规则

```
P(t,y,l) ← Movie(t,y,l,c,s,p)
```

可以满足要求，它定义了一个名为P的关系作为投影的结果。 □

10.2.5 选择

用Datalog来表示选择略微有些困难。当选择条件定为对一个或多个算术比较来作AND操作时，则是一种比较简单的情况。在这种情况下，我们创建一条规则包含：

⊖ 事实上，我们应该假设在这一节所有例子中除了直接给出的IDB谓词的规则外没有其他规则存在。如果有其他规则，那么我们不能排除该谓词的关系中存有其他元组。

1. 一个对应于我们要进行选择的关系的子目标。这个原子对每个组元有不同变量，它们分
473 别对应于关系的每个属性。

2. 对选择条件中的每个比较都有一个与该比较对应的算术子目标。一旦在选择条件中使用了一个属性名，我们就根据关系子目标建立的对应关系，在算术子目标中使用对应的变量。

例10.13 例5.4中的选择

$$\sigma_{length \geq 100 \text{ AND } studioName = \text{'Fox'}}(\text{Movie})$$

可以用Datalog规则重写为

```
S(t,y,l,c,s,p) ← Movie(t,y,l,c,s,p) AND l ≥ 100 AND s = 'Fox'
```

其结果就是关系S。注意l和s依照Movie属性的标准次序对应于属性length和studioName。 □

现在，让我们考虑那些包含对条件进行OR操作的选择。我们不能用一条Datalog规则代替这样的选择。然而，两个条件OR操作的选择等价于分别按每个条件进行选择，然后得到结果的并。因此，n个条件的OR可以用n条规则表示，每条规则都定义同样的头部谓词。第i条规则按照n个条件中的第i个进行选择。

例10.14 我们对例10.13中的选择进行修改，把AND替换成OR得到如下选择：

$$\sigma_{length \geq 100 \text{ OR } studioName = \text{'Fox'}}(\text{Movie})$$

也就是找出所有长度至少为100的电影或Fox出产的电影。我们可以写两条规则，每个对应一个条件：

```
1. S(t,y,l,c,s,p) ← Movie(t,y,l,c,s,p) AND l ≥ 100
2. S(t,y,l,c,s,p) ← Movie(t,y,l,c,s,p) AND s = 'Fox'
```

规则(1)得出至少有100分钟长的电影，规则(2)得到Fox出品的电影。 □

有的应用中甚至还有更为复杂的选择条件，它们由逻辑算子AND、OR和NOT按照任意顺序组成。然而有一种已很普及的技术，可以将这样的逻辑表达式重新整理成“析取范式”，即表达式是“合取”的析取(OR)。合取是AND的“文字表示”(literal)，“文字表示”是一个比较
474 或否定比较。我们在这里不多作介绍[⊖]。

我们可以用一个子目标表示任一“文字表示”，这个子目标的前面可能有一个NOT。如果子目标是算术子目标，这个NOT可以合并到比较算子中。例如：NOT x≥100可以写作x<100。那么任意合取都可以由一条单独的Datalog规则表示，一个子目标对应一个比较。最后，每个析取范式都可写成若干Datalog规则，一条规则对应一个合取。这些规则对每个合取的结果作并操作或OR操作。

例10.15 我们根据例10.14为这个算法给出一个简单的例子。对该例子中的条件进行否定就得到一个更复杂的例子。我们下面给出表达式：

$$\sigma_{\text{NOT } (length \geq 100 \text{ OR } studioName = \text{'Fox'})}(\text{Movie})$$

也就是，找出所有既不长又不是Fox出品的电影。

这里，一个NOT放在一个并非简单比较的表达式之前。因此，我们必须依照DeMorgan定律，即OR的否定也就是否定的AND，把NOT合并到表达式中。也就是说，这个选择可以被重写成：

⊖ 可参考A.V.Aho and J.D.Ullman, *Foundations of Computer Science*, Computer Science Press, New York, 1992.

$$\sigma_{(\text{NOT } (length \geq 100)) \text{ AND } (\text{NOT } (studioName = \text{' Fox' }))}(\text{Movie})$$

现在我们可以把NOT放在比较中以得到表达式

$$\sigma_{length<100 \text{ AND } studioName \neq \text{' Fox'}}(\text{Movie})$$

这个表达式可以转化为Datalog规则

```
S(t,y,l,c,s,p) ← Movie(t,y,l,c,s,p) AND l < 100 AND s ≠ 'Fox'
```

□

例10.16 让我们考虑一个类似的例子，其中的否定表达式中含有AND。现在，我们使用DeMorgan定律的第二形式，即AND的否定也就是否定的OR。我们从代数表达式

$$\sigma_{\text{NOT } (length \geq 100 \text{ AND } studioName = \text{' Fox' })}(\text{Movie})$$

开始，也就是，找出所有不同时满足Fox出品且放映时间较长的影片。

我们用DeMorgan定律把NOT放到AND下，得到：

$$\sigma_{(\text{NOT } (length \geq 100)) \text{ OR } (\text{NOT } (studioName = \text{' Fox' }))}(\text{Movie})$$

我们再一次把NOT放在比较中得到：

$$\sigma_{length<100 \text{ OR } studioName \neq \text{' Fox'}}(\text{Movie})$$

最后，我们写两条规则，分别对应OR的两个部分。其结果Datalog规则是：

```
1. S(t,y,l,c,s,p) ← Movie(t,y,l,c,s,p) AND l < 100
2. S(t,y,l,c,s,p) ← Movie(t,y,l,c,s,p) AND s ≠ 'Fox'
```

□ 475

10.2.6 积

两个关系的积$R \times S$可以用一条Datalog规则表示出来。这条规则有两个子目标，一个对应R，一个对应S。这两个子目标有不同的变量，分别对应R和S的属性。头部的IDB谓词拥有在这两个子目标中出现的所有参数，在R的子目标中的变量列在S的子目标中变量的前面。

例10.17 我们来考虑例10.9中的两个四属性的关系R和S，规则

```
P(a,b,c,d,w,x,y,z) ← R(a,b,c,d) AND S(w,x,y,z)
```

定义了P为$R \times S$。我们任意使用字母表中前面的字母作为R的变量，而最后的字母作为S的变量。这些变量都出现在规则头部。 □

10.2.7 连接

我们可以用一条近似于积的Datalog规则来表示两个关系的自然连接。不同之处在于：如果我们想要得到$R \bowtie S$，必须小心地对R和S的同名属性使用同样的变量，不同名属性使用不同的变量。例如：我们可以用属性名来作为变量名。规则头部是一个IDB谓词，它内部的每个变量出现一次。

例10.18 考虑模式为$R(A,B)$和$S(B,C,D)$的关系，其自然连接可以由规则

```
J(a,b,c,d) ← R(a,b) AND S(b,c,d)
```

来定义。注意子目标中使用的变量与关系R和S的属性有很明显的对应关系。 □

我们还可以把θ-连接转换成Datalog。回忆一下5.2.10节中θ-连接是怎样表示为一个积后接一个选择操作的。如果选择条件是合取，也就是比较的AND，那么我们可以简单的从积的Datalog规则开始，为每个比较添加算术子目标。

例10.19 让我们考虑例5.9中的关系$U(A,B,C)$和$V(B,C,D)$，其中使用了θ-连接

$$U \overset{\bowtie}{}_{A<D \text{ AND } U.B \neq V.B} V$$

我们可以建立Datalog规则

```
J(a,ub,uc,vb,vc,d)  ←  U(a,ub,uc) AND V(vb,vc,d) AND
                       a < d AND ub ≠ vb
```

476

来进行同样的操作。我们使用ub作为对应U的B属性的变量，ub、uc和vc也有类似的含义，尽管用任意六个不同的变量都可以表示这两个关系的六个属性。前两个子目标引入了这两个关系，后面两个子目标则执行这个θ-连接的条件中出现的两个比较。 □

如果这个θ-连接的条件不是合取，那么我们则像10.2.5节中那样把它转化成析取范式。然后我们为每个合取建立一条规则。在规则中，我们从积的子目标开始，为合取中每个文字添加一个子目标。所有规则的头部是一样的，并且每个参数对应于进行θ-连接的两个关系的每个属性。

例10.20 在这个例子中，我们将对例10.19的代数表达式作一个简单的修改。AND将被改成OR。这个表达式中没有否定，所以它总是析取范式。其中有两个合取都是文字表示单文字。这个表达式为：

$$U \overset{\bowtie}{}_{A<D \text{ OR } U.B \neq V.B} V$$

使用与例10.19中同样的变量名方法，我们得到两条规则：

```
1. J(a,ub,uc,vb,vc,d) ← U(a,ub,uc) AND V(vb,vc,d) AND a < d
2. J(a,ub,uc,vb,vc,d) ← U(a,ub,uc) AND V(vb,vc,d) AND ub ≠ vb
```

每条规则都有对应两个关系的子目标以及一个对应于$A<D$和$U.B \neq V.B$两个条件之一的子目标。 □

10.2.8 用Datalog模拟多重操作

Datalog规则不仅仅可以模拟关系代数中的单个操作。事实上我们可以模拟任何代数表达式。其技巧是观察关系代数表达式的表达树并为树的每个内部节点创建一个IDB谓词。IDB谓词的一条或多条规则就是我们需要在对应节点上使用的算子。树的扩展操作数（它们是数据库的关系）由对应谓词表示。作为内部节点的操作数则由对应的IDB谓词表示。

例10.21 考虑例5.10中的代数表达式

477

$$\pi_{title,year}(\sigma_{length \geq 100}(\text{Movie}) \cap \sigma_{studioName = \text{'Fox'}}(\text{Movie}))$$

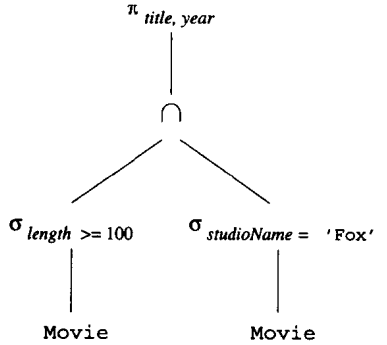


图10-2 表达树

```
1. W(t,y,l,c,s,p) ← Movie(t,y,l,c,s,p) AND l ≥ 100
2. X(t,y,l,c,s,p) ← Movie(t,y,l,c,s,p) AND s = 'Fox'
3. Y(t,y,l,c,s,p) ← W(t,y,l,c,s,p) AND X(t,y,l,c,s,p)
4. Z(t,y) ← Y(t,y,l,c,s,p)
```

图10-3 进行若干代数操作的Datalog规则

例5.10中的表达树与图5-8相同。我们重画该图为图10-2。该树共有四个内部节点，所以我们需要创建四个IDB谓词。每个谓词有一条Datalog规则，我们在图10-3中列出了这些规则。

最低的两个内部节点对EDB关系Movie进行简单的选择操作，所以我们可以创建IDB谓词W和X来表示这些选择。图10-3中的规则(1)和(2)描述了这些选择。例如，规则(1)定义W为Movie中长度至少为100的元组。

规则(3)按照我们在10.2.1节中给出的交的规则形式，定义谓词Y为W和X的交。最后，规则(4)定义谓词Z为Y在title和year属性上的投影。这里我们使用了在10.2.4节中学过的模拟投影技术。谓词Z是“答案”谓词，即无论关系Movie的值是什么，Z定义的关系就是我们在这个例子开始所给出的代数表达式的结果。

注意，因为Y只有一条规则定义，所以我们可以替换图10-3规则(4)中的Y子目标，将它换成规则(3)中的主体。接着，我们可以用规则（1）和（2）的主体替换W和X子目标。既然Movie子目标出现在W和X主体中，我们可以消去一个拷贝。结果，Z可以用一个单独的规则来定义：

```
Z(t,y) ← Movie(t,y,l,c,s,p) AND l ≥ 100 AND s = 'Fox'
```

478

然而，一个复杂的关系代数表达式只等于一条Datalog规则的情况并不常见。 □

10.2.9 习题

习题10.2.1 设$R(a,b,c)$、$S(a,b,c)$和$T(a,b,c)$是三个关系。写出一条或多条Datalog规则，分别定义出下列关系代数表达式的结果：

a) $R \cup S$.

b) $R \cap S$.

c) $R - S$.

* d) $(R \cup S) - T$.

! e) $(R - S) \cap (R - T)$.

f) $\pi_{a,b}(R)$.

*! g) $\pi_{a,b}(R) \cap \rho_{U(a,b)}(\pi_{b,c}(S))$.

习题10.2.2 设$R(x,y,z)$为一个关系。写一条或多条Datalog规则来定义$\sigma_C(R)$，其中C代表下列每个条件：

a) $x=y$.

* b) $x<y$ AND $y<z$.

c) $x<y$ OR $y<z$.

d) NOT $(x<y$ OR $x>y)$.

*! e) NOT $((x<y$ OR $x>y)$ AND $y<z)$.

! f) NOT $((x<y$ OR $x<z)$ AND $y<z)$.

习题10.2.3 设$R(a,b,c)$、$S(b,c,d)$和$T(d,e)$为三个关系。为每个自然连接写一条Datalog规则：

a) $R \bowtie S$.

b) $S \bowtie T$.

c) $(R \bowtie S) \bowtie T$. (注意：既然自然连接是相关联的和可交换的，这三个关系的连接顺序是无关的。)

习题10.2.4 设$R(x,y,z)$和$S(x,y,z)$是两个关系。写出一条或多条Datalog规则来定义每个θ-连接$R \overset{\bowtie}{C} S$，这里C是习题10.2.2中的一个条件。要求用左边R的一个属性和右边S的一个属性的比较来解释每个条件中的算术比较。例如：用$x<y$代替$R.x<S.y$。

479

! **习题10.2.5** 将Datalog规则转化为等价的关系代数表达式也是可行的。当我们还没有一般性的讨论到这样做的方法之前，也可以举出一些简单的例子。对下列每个Datalog规则，写出一个关系代数表达式来定义与该规则头部相同的关系。

* a) `P(x,y)←Q(x,z) AND R(z,y)`

b) `P(x,y)←Q(x,z) AND Q(z,y)`

c) `P(x,y)←Q(x,z) AND R(z,y) AND x<y`

10.3 Datalog的递归编程

虽然关系代数可以表示很多关系上的有效操作，但还是有很多计算不能够写成关系代数表达式。通常，我们不能用关系代数表示无限的、通过递归定义出的相似表达式序列。

例10.22 一个成功的电影经常会有续集。如果续集成功的话，则续集还会有续集，不断重复。于是，一部电影可能是另一部电影的远祖。假设我们有一个关系`SequelOf(movie, sequel)`包含一部影片和紧接着它的续集的配对。这个关系中元组的例子为：

movie	*sequel*
Naked Gun	Naked Gun 2$_{1/2}$
Naked Gun 2$_{1/2}$	Naked Gun 33$_{1/3}$

我们可能对电影后续有着一个更全面的理解：即续集、续集的续集，依次递推。在上述的关系中，Naked Gun 33$_{1/3}$是Naked Gun的一个后续，但不是我们在这儿使用的严格意义上的续集。如果我们在关系中只存放紧靠的续集，在需要时再建立后续，这样可以节约空间。在上面的例子中，我们只少存储了一个配对，但对五集的电影*Rocky*我们可以少存六个配对，对18集电影*Friday the 13th*我们则少存了136个配对。

然而，怎样从关系`SequelOf`建立后续的关系却不是立刻就能明白的。我们可以把`SequelOf`中的元组依次连接得到续集的续集。关系代数中这种表达式的一个例子如下，它用重命名使连接成为自然连接：

$$\pi_{first,third}(\rho_{R(first,second)}(\texttt{SequelOf}) \bowtie \rho_{S(second,third)}(\texttt{SequelOf}))$$

480

在这个表达式中，`SequelOf`被重命名两次，一次将其属性叫做`first`和`second`，第二次则被叫做`second`和`third`。这样，自然连接在`SequelOf`中寻找满足$m_2=m_3$的元组(m_1,m_2)和(m_3,m_4)。于是我们得到配对(m_1,m_4)，注意m_4是m_1的续集的续集。

同样地，我们可以对三个`SequelOf`作自然连接得到续集的续集的续集（例如*Rocky*和*Rocky IV*）。事实上我们可以对任意固定的i值，通过对`SequelOf`自身连接i-1次得到第i个续集。然后我们可以对`SequelOf`和这些连接的有限序列进行并操作得到在某个固定限制之内的所有续集。

在关系代数中，当i = 1,2,...时，第i个续集的表达式组成的无限序列无法进行“无限并”操作。注意，关系代数的并只允许我们进行两个关系的并操作，而不是无限个关系。通过在关系表达式中进行任意有限次的并操作，我们可以得到任意有限个关系的并，但我们不能在关系表

达式中得到无限个关系的并。 □

10.3.1 递归规则

通过在规则的头部和体部使用IDB谓词，我们可以用Datalog表示一个无限并。我们还是先来看一些怎样在Datalog中表示递归的例子。在10.3.2节中我们要检验对这些规则的IDB谓词关系的最小不动点计算。因为在10.1.4节中的直接规则计算方法假设所有规则体部的谓词是有固定关系的，所以计算递归规则需要一个新的规则计算方法。

例10.23 我们可以用下列两个Datalog规则定义IDB关系FollowOn:

```
1. FollowOn(x,y) ← SequelOf(x,y)
2. FollowOn(x,y) ← SequelOf(x,z) AND FollowOn(z,y)
```

第一个规则是基础，它告诉我们每个续集都是一个后续；第二个规则是说电影x的一个续集的每个后续也是x的后续。更确切地说：如果z是x的一个续集，并且我们发现y是z的一个后续，那么y则是x的一个后续。 □

10.3.2 计算递归Datalog规则

为了计算递归Datalog规则的IDB谓词，我们依照的原则是：只有像10.1.4节中那样应用规则时，我们才能判定一个元组是在一个IDB关系中。于是，我们：

1. 开始即假设所有IDB谓词都有空关系。

2. 执行一定数量的循环，在循环中为IDB谓词逐渐建立更大的关系。在规则的部分使用在前面的循环中建立的IDB关系。应用这些规则重新判定所有IDB谓词。 481

3. 如果规则是安全的，任何IDB元组的组元值都会在某个EDB关系中出现。因此，所有IDB关系都有一定数目的可能关联到的元组，这些元组最后会形成一个循环，其中不再为任何IDB关系加入新元组。这时我们可以结束计算得到结果——已经不会再产生新的IDB元组了。

这个IDB元组的集合称为规则的最小不动点。

例10.24 当关系SequelOf由如下三个元组组成时，让我们给出对关系FollowOn最小不动点的计算：

movie	*sequel*
Rocky	Rocky II
Rocky II	Rocky III
Rocky III	Rocky IV

在计算的第一个循环中，假设FollowOn为空。这样，规则（2）不会生成任何FollowOn元组。然而，规则（1）表明每个SequelOf元组都是一个FollowOn元组。因此在第一个循环结束后，FollowOn的值等同于上面列出的SequelOf关系，在第一循环后的情况如图10-4(a)所示。

在第二个循环中，我们以图10-4(a)所示关系作为FollowOn，并对这个关系和给定的SequelOf关系使用这两个规则。第一个规则给出我们已经得到了的三个元组，很明显，事实上规则(1)不可能为FollowOn产生任何不同于这三个元组的元组。对于规则(2)，我们从SequelOf中寻找一个元组，它的第二个组元与FollowOn的某个元组的第一个组元相同。

因此，我们可以从SequelOf中取得元组(Rocky, Rocky II)，并将它与FollowOn的元组(RockyII, Rocky III)配对并为FollowOn得到一个新元组(Rocky, RockyIII)。同样地，我们可以从SequelOf取出元组

(Rocky II,Rocky III)

并从FollowOn取出元组(RockyIII,RockyIV)，从而为FollowOn得到新元组(RockyII,RockyIV)。然而，没有其他的从SequelOf和FollowOn中取出的元组配对可以作连接了。这样，在第二个循环结束后，FollowOn有如图10-4(b)所示的五个元组。直观上，象图10-4(a)包括那些基于一个续集的后续那样，图10-4(b)包含了那些基于一个或两个续集的后续。

在第三个循环中，我们以图10-4(b)中的关系作为FollowOn再一次利用规则(2)进行计算，就得到了所有以前得到过的元组，当然，又多了一个元组。当我们连接SequelOf的元组
482 (Rocky,RockyII)和FollowOn当前值中的元组(RockyII,RockyIV)时，会得到一个新元组(Rocky,RockyIV)。这样，在第三个循环结束后，FollowOn的值如图10-4(c)所示。

当我们继续进入第四个循环时，没有得到新的元组，于是我们停止计算。正确的关系FollowOn如图10-4(c)所示。 □

x	y
Rocky	Rocky II
Rocky II	Rocky III
Rocky III	Rocky IV

(a) 循环1结束后FollowOn的值

x	y
Rocky	Rocky II
Rocky II	Rocky III
Rocky III	Rocky IV
Rocky	Rocky III
Rocky II	Rocky IV

(b) 循环2结束后FollowOn的值

x	y
Rocky	Rocky II
Rocky II	Rocky III
Rocky III	Rocky IV
Rocky	Rocky III
Rocky II	Rocky IV
Rocky	Rocky IV

(c) 循环3结束及以后得到的FollowOn正确值

图10-4 关系FollowOn的递归计算

一个可以简化所有递归Datalog计算的重要技巧是：

- 在任何循环中，向任何IDB关系中加入的新元组只能通过应用这样的规则得到：规则中
483 至少有一个IDB子目标与一个元组相匹配，而匹配的元组是在前面的循环中加入关系的。

递归的其他形式

在例10.23中我们使用右递归形式形成递归，它的递归关系FollowOn出现在EDB关系SequelOf的后面。我们也可以用类似的左递归规则，把递归关系放在前面。这些规则是：

```
1. FollowOn(x,y) ← SequelOf(x,y)
2. FollowOn(x,y) ← FollowOn(x,z) AND SequelOf(z,y)
```

简略地说，y是x的一个后续，当且仅当它是x的续集或x的后续的续集。

我们甚至可以使用两次递归关系，就像非线性递归：

```
1. FollowOn(x,y) ← SequelOf(x,y)
2. FollowOn(x,y) ← FollowOn(x,z) AND FollowOn(z,y)
```

简略地说，y是x的一个后续，当且仅当它是x的续集或x的后续的后续。这三种形式都向关系FollowOn给出同样的配对(x,y)：使得y是x的续集的续集的续集（重复若干次）…

对这个规则的证明是：假设所有子目标被匹配到“旧”元组上，头部的元组应该在前面的循环中就已经加入了。下面的两个例子对这个策略进行说明并且向我们展示更为复杂的递归的例子。

例10.25 在图的路径研究中有许多使用递归的例子。图10-5给出了一张图，上面标识出两家假定的航空公司在旧金山、丹佛、达拉斯、芝加哥和纽约之间的航班，这两家公司的名称分别为Untried Airlines(UA)公司和Arcane Airlines(AA)公司。

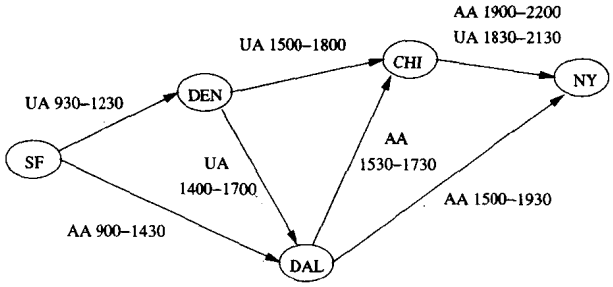


图10-5 航空公司的航班图

我们可以想像用一个EDB关系表示这些航班：

```
Flights(airline, from, to, departs, arrives)
```

图10-6给出了从图10-5中得出的关系的元组。

airline	*from*	*to*	*departs*	*arrives*
UA	SF	DEN	930	1230
AA	SF	DAL	900	1430
UA	DEN	CHI	1500	1800
UA	DEN	DAL	1400	1700
AA	DAL	CHI	1530	1730
AA	DAL	NY	1500	1930
AA	CHI	NY	1900	2200
UA	CHI	NY	1830	2130

图10-6 关系Flights中的元组

我们可以提出的最简单的递归问题是：“满足条件的(x,y)有哪些？x，y代表的意义是：乘坐一架或多架班机从城市x到达城市y。”下面两条规则描述了一个恰好包含这些城市配对的关系Reaches(x,y)。

```
1. Reaches(x,y) ← Flights(a,x,y,d,r)
2. Reaches(x,y) ← Reaches(x,z) AND Reaches(z,y)
```

484

第一个规则表示Reaches包含这样的城市配对：有一趟直接从第一个城市通向第二个城

市的航班。航空公司a与规则中的出发时间d和到达时间r是任意的。第二个规则表示如果你可以从城市x到城市z，又可以从城市z到城市y，那么你就可以从x到y。注意我们这里使用了递归的非线性形式，就像我们在上文“递归的其他形式”框中描述的那样。在这里使用这个形式更为方便，因为Flights在递归规则中的其他地方出现时会为没有用到的Filghts组元多引入三个变量。

我们按照例10.24中介绍的迭代过程计算关系Reaches。开始我们用规则(1)得到在Reaches中的以下配对：(SF,DEN)、(SF,DAL)、(DEN,CHI)、(DEN,DAL)、(DAL,CHI)、(DAL,NY)和(CHI,NY)。这些就是在图10-5中用边表示的七个配对。

在下一个循环中，我们使用递归规则(2)来对边进行配对，使得一条边的头是另一边的尾。这样我们就得到增加的配对(SF,CHI)、(DEN,NY)和(SF,NY)。第三个循环对所有一条或两条边的配对进行组合，形成最长包含四条边的路径。在这个特定的图中，我们没有得到新的配对。
485 这样Reaches由十个配对(x,y)组成，表示在图10-5中由x可以到达y。在图10-5里，配对(x, y)中的y在x的右边。 □

例10.26 一个更为复杂的限定情况是：什么时候可以把两个航班合并到一个更长的航班序列，其要求是第二航班离开机场的时间至少比第一航班抵达机场的时间晚一个小时。这里，我们使用一个称为Connects(x,y,d,r)的IDB谓词，它表示我们可以乘坐一个或多个航班，在d时刻从城市x出发，在r时刻抵达城市y。如果中间有连接，那么连接至少需要一个小时。

Connects的规则如下[⊖]：

```
1.  Connects(x,y,d,r)  ←  Flights(a,x,y,d,r)
2.  Connects(x,y,d,r)  ←  Connects(x,z,d,t1) AND
                          Connects(z,y,t2,r) AND
                          t1 <= t2 - 100
```

在第一个循环中，规则(1)给我们八个连接，即图10-7中第一条横向分隔线以上的连接（这条横向分隔线不是关系的一部分）。每个连接对应图10-5中的一趟航班。注意，该图中七条边中的一条代表不同时间的两趟航班。

我们现在尝试用规则(2)来合并这些元组。例如，这些元组的第二和第五个合并成为元组(SF,CHI,900,1730)。然而，第二和第六元组不能合并，因为到达拉斯的抵达时间是1430，而从达拉斯出发时间是1500，只多了半个小时。Connects关系在第二个循环结束后由图10-7中的第一和第二条横向分隔线以上的所有元组组成。从循环1得到的初始元组在第一条横向分隔线之上，在循环2中添加的六个元组位于第一和第二条横向分隔线之间。

在第三个循环中，我们主要考虑将图10-7的第一和第二条横向分隔线之上元组的所有配对作为规则(2)主体部分的两个Connects元组的候选者。然而，如果两个元组都是在第一条横向分隔线以上，那么它们在第二个循环中就已经被考虑过了，因此不会产生一个我们没有见过的Connects元组。得到一个新元组的惟一方法是在规则(2)使用的两个Connects元组中至少有一个是在前一循环中加入的，即它在图10-7的两条横向分隔线之间。

第三个循环只给我们三个新元组，位于图10-7的底部。在第四循环中没有新元组加入，所以我们的计算结束。这样，完整的关系Connects如图10-7所示。

□

⊖ 这些规则仅服从这样的假设：连接会发生在午夜。

x	y	d	r
SF	DEN	930	1230
SF	DAL	900	1430
DEN	CHI	1500	1800
DEN	DAL	1400	1700
DAL	CHI	1530	1730
DAL	NY	1500	1930
CHI	NY	1900	2200
CHI	NY	1830	2130
SF	CHI	900	1730
SF	CHI	930	1800
SF	DAL	930	1700
DEN	NY	1500	2200
DAL	NY	1530	2130
DAL	NY	1530	2200
SF	NY	900	2130
SF	NY	900	2200
SF	NY	930	2200

图10-7 在第三循环之后的关系Connects

□

10.3.3 递归规则中的非

有时，我们需要在递归规则中使用“非”操作。将递归和非进行混合的方法有两种，一种安全，一种不安全。一般来说，只有在不动点操作中不出现否定的情况下，使用否定才是合适的。为了观察它们的不同，我们要考虑递归和非的两个例子，一个是恰当的而另一个是自相矛盾的。我们会看到当有递归时，只有“分层的”非是有用的；关于术语“分层的”，我们会在例子之后作出确切的定义。 486

例10.27 假设我们要在图10-5中找出城市的配对(x,y)，使得UA可以从x飞到y（也许要经过某些其他城市），但AA却不可以。我们可以像例10.25中定义Reaches那样递归定义一个谓词UAreaches，但仅针对UA航班，如下所示：

```
1. UAreaches(x,y) ← Flights(UA,x,y,d,r)
2. UAreaches(x,y) ← UAreaches(x,z) AND UAreaches(z,y)
```

类似地，我们可以递归定义谓词AAreaches为城市配对(x,y)，使得人们可以只坐AA的航班从x到y，定义如下：

```
1. AAreaches(x,y) ← Flights(AA,x,y,d,r)
2. AAreaches(x,y) ← AAreaches(x,z) AND AAreaches(z,y)
```

现在，计算UAonly谓词，即计算人们可以坐UA航班而不坐AA航班可从x到y的城市配对(x,y)的集合。有了如下的非递归规则，计算就是一件简单的事情了。

```
UAonly(x,y) ← UAreaches(x,y) AND NOT AAreaches(x,y)
```

487

这条规则计算UAreaches和AAreaches两集合的差。

对图10-5中的数据，UAreaches包括下列配对：(SF, DEN)、(SF, DAL)、(SF, CHI)、(SF, NY)、(DEN, DAL)、(DEN, CHI)、(DEN, NY)和(CHI, NY)。这个集合是由10.3.2节中给出的迭代不动点过程计算出来的。类似地，我们可以计算AAreaches的值为以下数据：(SF, DAL)、(SF, CHI)、(SF, NY)、(DAL, CHI)、(DAL, NY)和(CHI, NY)。当我们计算这些配对集合的差时我们得到：(SF, DEN)、(DEN, DAL)、(DEN, CHI)和(DEN, NY)。这个四个配对的集合就是关系UAonly。 □

例10.28 现在，让我们考虑一个抽象的反面例子。假设：EDB谓词R是一元关系（一个参数），并且它只有一个元组(0)。另外两个IDB谓词P和Q也是一元关系，它们由两个规则定义：

```
1. P(x) ← R(x) AND NOT Q(x)
2. Q(x) ← R(x) AND NOT P(x)
```

简言之，这两个规则告诉我们：R中的一个元素x要么在P中，要么在Q中，但不会同时在两者中。注意P和Q在递归定义中互相关联。

当我们在10.3.2节定义递归关系的含义时，曾经说我们需要最小不动点，也就是最小的IDB关系，关系中包含了规则要求我们承认的所有元组。规则(1)是生成P的惟一规则，作为关系，$P=R-Q$；同样，规则(2)定义$Q=R-P$。由于R仅包含元组(0)，这个仅有的(0)可以在P或Q中。但(0)在哪里？它不可能不在P和Q中，因为这将使等式不满足。例如，$P=R-Q$意味着$\phi=\{(0)\}-\Phi$，这是错误的。

如果我们让$P=\{(0)\}$，$Q=\phi$，那么我们得到两个等式的解法。$P=R-Q$成为$\{(0)\}=\{(0)\}-\phi$，这是对的，$Q=R-P$成为$\phi=\{(0)\}-\{(0)\}$，这也是对的。

然而，我们还可以让$P=\phi$，$Q=\{(0)\}$。这个选择也满足这两条规则。于是我们有两个解：

a) $P=\{(0)\}$ $Q=\phi$

b) $P=\phi$ $Q=\{(0)\}$

两者都是最小的，因为如果我们从任意关系中取出任意元组，结果关系就都不再满足这两条规则。因此，我们不能在这两个最小不动点(a)和(b)中作决定，我们也就不能回答“$P(0)$是否为真？”这样简单的问题。 □

在例10.28中我们发现，当递归和非太过紧密地缠绕在一起时，我们定义递归规则为最小不
488 动点的思路已不再适合。可能有一个以上的最小不动点，这些不动点可以互相矛盾。如果有其他更合适的方法来定义递归非当然更好。然而不妙的是，目前还未就此达成广泛一致的意见。

因此，我们习惯于将递归限制在分层的非中。例如，在10.4节中讨论的递归的SQL-99标准就作了这样的限制。我们看到，当非是“分层的”时，有一种算法可以计算一个特定的、直观上与我们关于规则的含义相符合的最小不动点（也许是出自于很多个这样的不动点之中）。我们定义分层非的属性如下：

1. 画一张图，其中节点对应IDB谓词。

2. 如果一个规则的头部有谓词A，并且有一个谓词B的否定子目标，那么从节点A画一条边到节点B。为这条边作一个标记“-”，表示它是一个否定的边。

3. 如果一个规则的头部有谓词A，并且有一个谓词B的非否定子目标，那么从节点A画一条边到节点B。这条边不用减号作标记。

如果这张图有一个环包括一个或多个否定的边，那么这个递归则不是分层的。否则，这个递归是分层的。我们可以把一张分层图中的IDB谓词划分为阶层。一个谓词A的阶层值是从A开始的一条路径上的否定边的最大数量。

如果递归是分层的，那么我们可以依照分层顺序来计算IDB谓词，阶层最低的优先计算。运用这种方法产生了规则的一个最小不动点。更重要的是，依照IDB谓词的层次顺序来计算这些谓词总是有意义的并且使我们得到“正确的”不动点。相反，我们在例10.28中已经看到，不分层的递归可能导致根本没有“正确的”不动点，尽管可能有很多不动点可以选择。

例10.29 例10.27中谓词的图如图10-8所示。因为以自身为起始点的路径没有一条包含否

定的边，所以AAreaches和UAreaches是在阶层0。UAonly为阶层1，因为有一条包含否定边的路径从它开始，但这样的路径只有一条。因此，我们必须在开始计算UAonly之前完成AAreaches和UAreaches的计算。 489

比较一下我们为例10.28中的IDB谓词构造图的情况，如图10-9所示。由于规则(1)有头部P和否定子目标Q，因此有一条否定边从P到Q。由于规则(2)有头部Q和否定子目标P，因此也有一条相反方向的否定边。这样就有了一个否定环，这两条规则不是分层的。

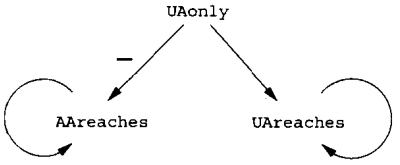


图10-8 从一个分层递归建立的图

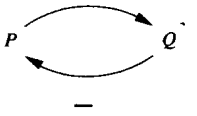


图10-9 由一个非分层递归构造的图

□

10.3.4 习题

习题10.3.1 如果我们添加或删除图10-5中的边，可能会改变例10.25中关系Reaches的值、例10.26中关系Connects的值或例10.27中关系UAreaches和AAreaches的值。如果我们作出如下增、删改动，试重新给出这些关系的值。

* a) 添加一条边从CHI到SF，标记为AA，1900-2100。

b) 添加一条边从NY到DEN，标记为UA，900-1100。

c) 按(a)和(b)所示添加两条边。

d) 删除从DEN到DAL的边。

习题10.3.2 针对例10.22中概念“后续”所作的下列修改，写出Datalog规则（如果必须用非，使用分层非）来描述。可以使用例10.23中定义的EDB关系SequelOf和IDB关系FollowOn。

* a) P(x,y)表示电影y是电影x的一个后续，但不是x的一个续集（就像EDB关系SequelOf所定义的那样）

b) Q(x,y)表示y是x的一个后续，但既不是一个续集也不是续集的续集。

! c) R(x)表示电影x有至少两个后续。注意这两个后续可以都是续集，而不仅是一个为续集而另一个为续集的续集。

!! d) S(x,y)表示y是x的一个后续且y最多只有一个后续。 490

习题10.3.3 ODL类和它们之间的关系可以由关系Rel(class,rclass,mult)来描述。这里，对一个多值关系mult值是multi，而对一个单值关系mult值则是single。前两个属性是相关的类：从class到rclass（相关类）。例如，关系Rel代表图4-3中正在播放的电影例子中的三个ODL类，如图10-10所示。

class	*rclass*	*mult*
Star	Movie	multi
Movie	Star	multi
Movie	Studio	single
Studio	Movie	multi

图10-10 用关系数据表示ODL关系

我们也可以用一张图来表示这些数据，其中节点代表类，而边从一个类到一个相关类，相应的标记为multi或single。图10-11是根据图10-10中的数据描绘得来。

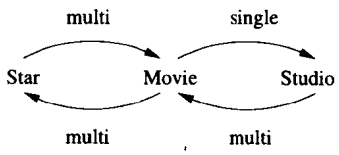


图10-11 用图表示关系

写出Datalog规则来表达下列谓词，如果必须用非就使用分层非。你可以把Rel作为一个EDB关系来使用。对图10-10的数据，按照循环步骤为你的规则计算出结果。

a) 谓词P(class,eclass)，它表示在类的图中有一条路径从class到eclass[⊖]。第二个类可以看做是“嵌入”在class中，因为它在某种意义上是第一个类对象的部分的部分的部分……

*! b) 谓词S(class,eclass)和M(class,eclass)，前者表示在class中有一个“单值的嵌入”的eclass，即从class到eclass的路径上的每条边都有标记single。后者*M*表示在class中有一个“多值的嵌入”的eclass，即一条从class到eclass的路径上至少有一条边标记为multi。

491

c) 谓词Q(class,eclass)表示有一条路径从class到eclass，但不是单值路径。你可以使用在这个练习前面定义过的IDB谓词。

10.4 SQL中的递归

SQL-99标准所包含的对递归规则的规定基于在10.3节描述的递归Datalog。尽管这一特点并不是所有DBMS都必须实现的“核心”SQL-99标准的一部分，但至少有一个重要的系统——IBM的DB2——采纳了SQL-99的建议。SQL-99的建议与我们的描述主要存在两个方面的差别：

1. 只强制限制线性递归，即规则中最多只有一个递归子目标。在后续讨论中我们将忽略这个限制。但要记住，有些标准SQL实现时禁止非线性递归而允许线性递归。

2. 我们在10.3.3节中讨论过的否定算子的分层要求也同样适用于其他可能引发同样问题的SQL算子，如聚集。

10.4.1 在SQL中定义IDB关系

WITH语句允许我们定义等价于IDB关系的SQL表达式。这些定义还可用在WITH语句本身。一个简单形式的WITH语句是：

WITH *R* AS <definition of *R*> <query involving *R*>

也就是说，先定义一个暂时的关系名为*R*，接着在某些查询中使用*R*。更一般地，可以在WITH后先定义若干关系，把它们的定义用逗号分开。这些定义都是可递归的。某些定义过的关系之间存在相互递归，即通过定义，每一个关系都可以与一些其他关系相关联，甚至可以关联它自身。然而，任何一个在递归中出现的关系前都必须有关键词RECURSIVE。这样，一个WITH语句的形式如下：

1. 关键词WITH。

2. 一个或多个定义。定义由逗号分开，每个定义包括：

(a) 一个可选的关键词RECURSIVE，如果该关系被定义为递归的，则该关键词是必须的。

(b) 被定义关系的名字。

⊖ 在这个练习中我们不考虑把空路径作为路径。

(c) 关键词AS。 492

(d) 定义该关系的查询。

3. 一个查询，它可以任意引用前面的定义，生成WITH语句的结果。

注意，与其他关系的定义不同，WITH语句中的定义只在该语句中有效，不能在别处使用，这一点很重要。如果想要一个关系持续有效，就应该在数据库模式中定义这个关系，而不是在任何WITH语句中定义。

例10.30 让我们重新考虑在10.3节中用作例子的航班信息。航班数据满足一个关系[⊖]：

```
Flights(airline, frm, to, departs, arrives)
```

图10-5给出了该例子中的确切数据。

在例10.25中，我们计算了IDB关系Reaches，以表示乘坐EDB关系Flights中的航班从第一个城市飞到第二个城市的城市配对。Reaches中的两个规则为：

```
1. Reaches(x,y) ← Flights(a,x,y,d,r)
2. Reaches(x,y) ← Reaches(x,z) AND Reaches(z,y)
```

利用这些规则，我们可以写出一个可生成关系Reaches的SQL查询。这个SQL查询把Reaches的规则放在一个WITH语句中，后面跟着一个查询。在例10.25中，期望的结果是完整的Reaches关系，我们也可以对Reaches作一些查询，例如从丹佛可以到达的城市集合。

```
1)  WITH RECURSIVE Reaches(frm, to) AS
2)          (SELECT frm, to FROM Flights)
3)      UNION
4)          (SELECT R1.frm, R2.to
5)           FROM Reaches R1, Reaches R2
6)           WHERE R1.to = R2.frm)
7)  SELECT * FROM Reaches;
```

图10-12 对可到达城市配对的递归SQL查询

图10-12说明了怎样用SQL查询来计算Reaches。第(1)行介绍Reaches的定义，对这个关系更确切的定义是在第(2)到第(6)行。

这个定义是两个查询的并，它们分别对应于例10.25中定义Reaches的两条规则。第(2)行 493
是并的第一项，对应第一个或基本规则。它表示对于Flight关系中的每个元组，第二和第三组元(frm和to组元)是Reaches的一个元组。

相互递归

有一种图论方法可以检查两个关系或谓词是否是相互递归的。建立一个其中节点与关系（如果我们在使用Datalog规则则是谓词）相对应的依赖图。如果关系B的定义直接依赖于关系A的定义，则从A到B画一条边。如果在使用Datalog，则指A出现在一个规则的体部而B出现在头部。在SQL中，A会出现在B定义中的某处，一般是在一条FROM子句中，但也可能作为并、交或差的一项。

如果有一个环包含节点R和S，那么R和S是相互递归的。最常见的情况是一个从R到R的循环，表明R递归地依赖它自己。

注意，依赖图近似于我们在10.3.3节介绍过的定义分层非的图。然而，那里需要区别肯定依赖和否定依赖，而这里则不需要进行区分。

⊖ 因为from在SQL中是关键词，所以我们把第二个属性名改成了frm。

第(4)到第(6)行对应Reaches定义中第二条即归纳规则。这两个Reaches子目标在FROM子句中由Reaches的两个别名R1和R2表示。R1的第一个组元对应规则(2)中的x，R2的第二个组元对应y。变量z由R1的第二个组元和R2的第一个组元表示。注意，这些组元在第(6)行中相等。

最后，第(7)行描述了由整个查询生成的关系。它是关系Reaches的一个拷贝。用另一种方法，我们可以把第(7)行替换成一个更为复杂的查询。例如：

```
7)  SELECT to FROM Reaches WHERE frm = 'DEN';
```

会产生所有从丹佛可到达的城市。 □

10.4.2 分层非

可以作为递归关系定义一个查询而不是任意的SQL查询。它们必须受到特定方式限制。一个最重要的要求就是相互递归关系的非必须是分层的，就像在10.3.3节讨论的那样。在10.4.3节
494 我们将看到分层的原理可扩展到其他在SQL中，但在Datalog中没有，如聚集。

例10.31 让我们重新考查例10.27，在该例中我们查找可以乘坐UA航空公司的航班（而不是AA的航班）从x飞到y的所有城市配对(x,y)。我们需要用递归来表达这样的概念，即乘坐一个航空公司的航班，通过一个无限的短程飞行序列来进行旅行的概念。然而，非的特征以分层方式出现：用递归计算出例10.27中的UAreaches和AAreaches两个关系后，我们得出了它们的差。

我们可以采用同样的策略来写SQL查询。但是，为了说明一个不同的解决方法，我们将先递归地定义一个关系Reaches(airline,frm,to)，它的三元组(a, f, t)表示只乘坐航空公司a的班机，允许经过若干次转机，从城市f飞到城市t。我们还要使用一个非递归关系Triples(airline,frm,to)，它是Flights向这三个相关组元的投影。这个查询如图10-13所示。

第(3)到第(9)行将关系Reaches定义为两项的并。基础项是第(4)行的关系Triples。归纳项是第(6)到第(9)行的查询，它生成Triples和Reaches自身的连接。这两项的作用是为Reaches加入所有(a,f,t)元组，以使得人们可以只乘坐航空公司a的班机，经过一次或多次转机，从城市f飞到城市t。

查询本身出现在第(10)到第(12)行。第(10)行给出乘坐UA飞机可到达的城市配对，第(12)行给出乘坐AA飞机可到达的城市配对。查询的结果是这两个关系的差。

```
1) WITH
2)     Triples AS SELECT airline, frm, to FROM Flights,

3)     RECURSIVE Reaches(airline, frm, to) AS
4)             (SELECT * FROM Triples)
5)         UNION
6)             (SELECT Triples.airline, Triples.frm, Reaches.to
7)              FROM Triples, Reaches
8)              WHERE Triples.to = Reaches.frm AND
9)                    Triples.airline = Reaches.airline)

10)    (SELECT frm, to FROM Reaches WHERE airline = 'UA')
11) EXCEPT
12)    (SELECT frm, to FROM Reaches WHERE airline = 'AA');
```

图10-13 分层查询两家航空公司中一家开通的航班可到达的城市

例10.32 在图10-13中，第(11)行EXCEPT所表示的非明显是分层的，因为它只是在第(3)行到第(9)行的递归结束之后才被使用。另一方面，我们认为在例10.28中所应用的非是非分层的非，必须将它转换成相互递归关系定义中的EXCEPT。图10-14给出了针对该例进行直接转换的SQL语句。这个查询只求出了P的值，尽管我们还可以求出Q或P和Q的一些函数。 495

```
 1)  WITH
 2)      RECURSIVE P(x) AS
 3)              (SELECT * FROM R)
 4)          EXCEPT
 5)              (SELECT * FROM Q),

 6)      RECURSIVE Q(x) AS
 7)              (SELECT * FROM R)
 8)          EXCEPT
 9)              (SELECT * FROM P)

10)  SELECT * FROM P;
```

图10-14 非分层查询，非法的SQL

在图10-14的第(4)和第(8)行中两次使用了EXCEPT，是非法的SQL，因为在任一情况下第二个参数都是与被定义的关系相互递归的关系。因此，这些非的使用不是分层非，是不允许的。事实上，这个问题没有SQL解法，也不会有，因为图10-14中的递归并没有为关系P和Q定义惟一的值。 □

10.4.3 有问题的递归SQL表达式

我们在例10.32中已经看到，借助EXCEPT定义一个递归关系时有可能违反SQL分层非的要求。而且，还有其他的不使用EXCEPT的不可接受的查询形式。例如，一个关系的非也可以表示为NOT IN。这样，图10-14中第(2)到第(5)行也可以写成：

```
RECURSIVE P(x) AS
    SELECT x FROM R WHERE x NOT IN Q
```

这个重写使得递归仍是非分层的，也是非法的。

另一方面，在WHERE子句中简单地使用NOT，如NOT x=y（或者写为x<>y）并不会自动违反分层非条件。那么，确定可以用来定义SQL递归关系的SQL查询的一般规则是什么呢？ 496

定义一个合法的SQL递归的原则是，递归关系R的定义只能用于相互递归关系S（S可以是R本身），其前提是S的使用是*单调的*。当添加任意元组到S中时，要么添加一个或多个元组到R，要么使得R不变，但不会使得R中任意元组被删除，此时称S的使用是单调的。

当考虑10.3.2节中描述的最小不动点计算时，这个规则就显得很有意义了。我们开始时递归定义的IDB关系为空，接着在后续的循环中不断向它们添加元组。如果在一个循环中添加一个元组会导致在下一循环必须删除一个元组，那么就会有抖动的危险，不动点计算有可能不能收敛。在下面的例子中，我们将会看到一些非单调构造，它们在SQL递归中是非法的。

例10.33 图10-14是对例10.28的非分层非的Datalog规则的实现。其中，规则允许两个不同的最小不动点。正如我们预期的，图10-14中P和Q的定义不是单调的。我们以第(2)到第(5)行的P的定义为例。P依赖于Q，两者相互递归，但添加一个元组到Q中会删除P中的一个元组。来看看为什么，假设R包含两个元组(a)和(b)，Q包含元组(a)和(c)。那么$P=\{(b)\}$。然而，如果添加(b)到Q，那么P为空。对Q添加一个元组会导致P中删除一个元组，所以我们可以说它是一

个非单调的非法构造。

当我们试图通过计算最小不动点来计算关系P和Q时，缺乏单调性会直接导致抖动行为的发生[⊖]。例如，假设R有两个元组$\{(a),(b)\}$。开始时，P和Q都为空。因此，在第一循环中，图10-14中的第(3)到第(5)行计算出包含值$\{(a),(b)\}$的P。第(7)到第(9)行计算出包含同样值的Q，因为在第(9)行中使用的是原有的为空值的P。

现在，R、P和Q都有值$\{(a),(b)\}$。因此，在下一循环中，在第(3)到第(5)行以及第(7)到第(9)行分别计算出P和Q的值为空。在第三循环，计算得到它们的值为$\{(a),(b)\}$。这个过程会不断持续下去，在偶数循环两个关系都为空，而奇数循环都为$\{(a),(b)\}$。因此，我们永远不能从图10-14的“定义”中得到两个关系P和Q的确定值。 □

例10.34 聚集也可能导致非单调性，尽管一开始连接可能会不明显。假设我们有一元（一个属性）关系P和Q，它们由以下两个条件定义：

497

1. P是Q与一个EDB关系R的并。
2. Q有一个元组，它是P的成员的总和。

我们可以用一个WITH语句表示这些条件，尽管这条语句违反了SQL的单调性要求。图10-15给出了对P的值的查询。

```
1)  WITH
2)      RECURSIVE P(x) AS
3)              (SELECT * FROM R)
4)          UNION
5)              (SELECT * FROM Q),

6)      RECURSIVE Q(x) AS
7)          SELECT SUM(x) FROM P

8)  SELECT * FROM P;
```

图10-15 涉及聚集的非单调非法SQL查询

假设R包含元组(12)和(34)，且P和Q的初始值都为空，因为这是不动点计算开始时的必然要求。图10-16一起给出了前六个循环中计算出的值。重温一下我们已经认可的策略，即所有关系都由前一循环的值在下一个循环中计算出来。于是，P在第一循环计算出的值是与R一样的，而Q为空，因为在第(7)行中使用了P原有的空值。

Round	P	Q
1)	{(12), (34)}	∅
2)	{(12), (34)}	{(46)}
3)	{(12), (34), (46)}	{(46)}
4)	{(12), (34), (46)}	{(92)}
5)	{(12), (34), (92)}	{(92)}
6)	{(12), (34), (92)}	{(138)}

498

图10-16 对非单调聚集不动点的迭代计算

在第二个循环，第(3)到第(5)行的并等于集合$R = \{(12),(34)\}$，所以它成为P的新值。P的原

⊖ 如果递归不是单调的，那么我们在WITH子句中对求解关系的顺序会影响最终结果，而当递归是单调时，结果是与顺序无关的。在本例和下一例中，我们假设在每一循环中，P和Q是“并行”求解的。即在每个循环中，都用每个关系的原有值来计算另一关系的值。可参考“在不动点计算中使用新值”框中内容。

值与新值一样，所以在第二循环Q = {(46)}。也就是说，46等于12和34的和。

在第三循环，我们从第(2)到第(5)行得到P = {(12),(34),(46)}，使用P的原值{(12),(34)}，第(6)到第(7)行再次将Q定义为{(46)}。

在不动点计算中使用新值

人们可能会问为什么在例10.33和10.34中我们用P的原值计算Q，而不是用P的新值。如果这些查询是合法的，而我们在每个循环中使用新值，那么查询结果可能会依赖我们在WITH子句中给出递归谓词定义的先后顺序。在例10.33中，P和Q会依赖于计算顺序而收敛到一个或两个可能的不动点。在例10.34中，P和Q还是不会收敛，事实上它们的值在每个循环中都改变，而不是每隔一个循环改变一次。

在第四循环，P有同样值{(12),(34),(46)}，但Q的值为{(92)}，因为12+34+46=92。注意，Q失去了元组(46)，尽管它得到了元组(92)。也就是说，添加一个元组(46)到P中会导致Q中一个元组（因为巧合是同样元组）被删除。这种行为是被SQL在递归定义中禁止的非单调性行为，印证了图10-15中的查询是非法的。一般来说，在第$2i$个循环，P包含元组(12)，(34)和(46i–46)，而Q仅包含元组(46i)。 □

10.4.4 习题

习题10.4.1 在例10.23中我们讨论了关系

```
SequelOf(movie, sequel)
```

它给出了一个电影紧接的续集。我们还定义了一个IDB关系FollowOn，其配对(x,y)表示电影y为x的续集、续集的续集或者不断延伸的续集。试完成下述练习：

a) 以SQL递归形式写出FollowOn的定义。

b) 写出一个递归SQL查询，返回配对(x,y)的集合，使得电影y是电影x的后续，但不是续集。

c) 写出一个递归SQL查询，返回配对(x,y)的集合，使得电影y是电影x的后续，但既不是续集，也不是续集的续集。

! d) 写出一个递归SQL查询，返回至少有两个后续的电影x的集合，注意两个后续可以都是续集，而不必一个是续集，另一个是续集的续集。

! e) 写出一个递归SQL查询，返回配对(x,y)的集合，使得电影y是电影x的后续，但y最多有一个后续。

499

习题10.4.2 在习题10.3.3中，我们介绍了关系

```
Rel(class, eclass, mult)
```

它描述了一个ODL类是如何与其他类关联的。具体地说，如果有一个从类c到类d的关系，那么这个关系拥有元组(c,d,m)。如果m= 'multi'，那么这个关系是多值的；如果m = 'single'，那么这个关系是单值的。我们在习题10.3.3中还建议可以把Rel看做对一张图的定义，图中的节点是类。当且仅当(c,d,m)是Rel的一个元组时，在图中有一条从c到d的边，标记为m。写出一个递归SQL查询，生成下列配对(c,d)的集合：

a) 在上述的图中有一条从类c到类d的路径。

* b) 有一条路径从c到d，该路径上的每条边都被标记为single。

*! c) 有一条路径从c到d，该路径上至少有一条边被标记为multi。

d) 有一条路径从c到d，但不存在所有边都被标记为single的路径。

! e) 有一条路径从c到d，该路径上的边交替标记为single和multi。

f) 有路径从c到d和从d到c，路径上每条边都被标记为single。

10.5 小结

- Datalog：这种逻辑形式允许我们在关系模型上编写查询。在Datalog中可以编写规则，规则的头部谓词或关系根据子目标组成的体部来定义。
- 原子：规则头部和子目标都是原子，原子由一个应用于若干个参数（可选为否定）的谓词组成。谓词可以表示关系或算术比较（如<）。
- IDB和EDB谓词：某些谓词对应于已存储的关系，被称为EDB（扩展数据库）谓词或称为关系。另一种谓词，称为IDB（内涵数据库）谓词，是由规则定义的。EDB谓词不出现在规则头部。
- 安全规则：一般我们说Datalog规则是安全的，是指规则中每个变量都出现在体部的一些非否定关系子目标中。安全规则保证：如果EDB关系是有限的，那么IDB关系也将是有限的。

500

- 关系代数和Datalog：所有关系代数可以表示的查询也可以用Datalog表示出来。如果规则是安全和非递归的，那么它们可以定义与关系代数完全一样的查询集合。
- 递归Datalog：Datalog规则可以为递归的，允许一个关系由它自身定义。不加否定的递归Datalog规则的含义是最小不动点：IDB关系的最小元组集合，它使得规则头部与规则体部的共同含义保持一致。
- 分层非：当一个递归包含非操作时，最小不动点可能不惟一，在某些情况下对于Datalog规则没有可接受的含义。因此，在一个递归内部使用非是必须被禁止的，这导致了对分层非的要求。对这种类型的规则，有一个（也许若干个）最小不动点，它是规则一般可以接受的含义。
- SQL递归查询：在SQL中可以定义暂时关系，它的使用类似于Datalog中的IDB关系。这些暂时关系可以被递归使用，以建立查询的结果。
- SQL的分层：在SQL递归中的非和聚集必须是单调的，这是对Datalog中分层非要求的概括。直观地说，就是一个关系不能直接或间接的由自身的非或聚集来定义。

10.6 参考文献

Codd在他的一篇关于关系模型的早期论文[4]中介绍了一种被称为关系演算的前序逻辑形式。关系演算是一种表达式语言，很像关系代数，事实上在表示能力上与关系代数是一样的，该事实已在[4]中证明。

看上去更像逻辑规则的Datalog曾受到过编程语言Prolog的启发。因为它允许递归，它在表达力上强于关系演算。书目[6]引发了查询语言逻辑的很大发展，而书目[2]把这些思想引入了数据库系统。

关于分层方法给出不动点的正确选择的想法来自于书目[3]，而使用该方法来计算Datalog规则是书目[1]，[8]，[10]中的意见。关于分层非、关于关系代数、Datalog和关系演算之间的关系，以及关于有、无否定时Datalog规则计算的讨论，都可以在书目[9]中找到。

501

[7]纵览了基于逻辑的查询语言，关于递归的SQL-99提案的来源为书目[5]。

1. Apt, K. R., H. Blair, and A. Walker, "Towards a theory of declarative knowledge," in *Foundations of Deductive Databases and Logic Programming* (J. Minker, ed.), pp. 89–148, Morgan-Kaufmann, San Francisco, 1988.

2. Bancilhon, F. and R. Ramakrishnan, "An amateur's introduction to recursive query-processing strategies," *ACM SIGMOD Intl. Conf. on Management of Data*, pp. 16–52, 1986.

3. Chandra, A. K. and D. Harel, "Structure and complexity of relational queries," *J. Computer and System Sciences* **25**:1, pp. 99–128.

4. Codd, E. F., "Relational completeness of database sublanguages," in *Database Systems* (R. Rustin, ed.), Prentice Hall, Engelwood Cliffs, NJ, 1972.

5. Finkelstein, S. J., N. Mattos, I. S. Mumick, and H. Pirahesh, "Expressing recursive queries in SQL," ISO WG3 report X3H2–96–075, March, 1996.

6. Gallaire, H. and J. Minker, *Logic and Databases*, Plenum Press, New York, 1978.

7. M. Liu, "Deductive database languages: problems and solutions," *Computing Surveys* **31**:1 (March, 1999), pp. 27–62.

8. Naqvi, S., "Negation as failure for first-order queries," *Proc. Fifth ACM Symp. on Principles of Database Systems*, pp. 114–122, 1986.

9. Ullman, J. D., *Principles of Database and Knowledge-Base Systems, Volume I*, Computer Science Press, New York, 1988.

10. Van Gelder, A., "Negation as failure using tight derivations for general logic programs," in *Foundations of Deductive Databases and Logic Programming* (J. Minker, ed.), pp. 149–176, Morgan-Kaufmann, San Francisco, 1988.

第11章 数据存储

这一章研究数据库管理系统的实现。首先讨论的是DBMS如何有效地处理非常大量的数据。研究分为两个部分：

1. 计算机系统如何存储和管理非常大量的数据？

2. 何种表示方式和数据结构是对有效处理这类数据的最佳支持？

本章先回答第一个问题，第二个问题留待第12～14章讨论。

本章研究用于存储大量信息的设备，特别是旋转式磁盘。同时介绍“存储器层次”，并考察数据在主存储器与辅助存储器（通常是磁盘）乃至“第三级存储器”（用于存储和访问大量的光盘或磁带的机器人设备）之间移动的模式如何影响涉及大量数据的算法的效率。一个特定的算法，即两阶段多路归并排序算法，会被用来作为这类有效地使用存储器层次算法的一个重要示例。

在11.5节还将讨论缩短从磁盘读写数据时间的若干技术。最后两节讨论提高磁盘可靠性的方法，所涉及的问题包括间断性读写错误，以及造成数据永不可读的“磁盘崩溃”。

作为本章的开端，先来讨论一个虚构的测试，看看如果不采用DBMS的特定实现方法会带来什么问题。

11.1 Megatron 2002数据库系统

如果你用过某个DBMS，你可能认为实现这样的一个系统并不困难。你心里可能会想到Megatron系统公司最近（虚构）提供的一个实现：Megatron 2002数据库管理系统。该系统在 503
UNIX和其他操作系统上运行，采用关系方法，支持SQL语言。

11.1.1 Megatron 2002实现细节

首先，Megatron 2002采用UNIX文件系统来存储它的关系。例如，关系Students(name,id,dept)会存储在文件/usr/db/Students中。对于该关系的每个元组，在文件Students中有一行。一个元组中各个分量的值存储成由特殊的标记字符#分开的字符串。例如，文件/usr/db/Students可能看起来像下面这样：

```
Smith#123#CS
Johnson#522#EE
...
```

存储数据库模式的特定文件的命名为/usr/db/schema。对于每一个关系，文件schema中有一个以该关系的名字起始的行，行中关系的属性名和属性类型交替出现。字符#用来分隔行中元素。例如，文件schema中可能包含如下的行：

```
Students#name#STR#id#INT#dept#STR
Depts#name#STR#office#STR
...
```

这里对关系Students(name,id,dept)进行了描述，即属性name和dept的类型是字符串，属性id的类型是整数。文件schema中同时还有对模式为Depts(name,office)的关系的

描述。

例11.1 下面是使用Megatron 2002 DBMS的一个会话实例。程序运行在一台称做dbhost的机器上，通过UNIX层的命令megatron 2002启动DBMS。

```
dbhost>megatron2002
```

产生如下响应

```
WELCOME TO MEGATRON 2002!
```

现在是与Megatron 2002的用户界面对话，对于Megatron提示符（&），可以键入SQL查询作为响应。查询用#结束。例如，

```
& SELECT * FROM Students #
```

产生下表作为回答

name	*id*	*dept*
Smith	123	CS
Johnson	522	EE

504

Megatron 2002还允许执行一个查询后把结果存在一个新的文件中，做法是用一条竖线和文件名来结束查询。例如：

```
& SELECT * FROM Students WHERE id >=500 | HighId #
```

查询创建了一个新的文件/usr/db/HighId，文件中只有一行数据：

```
Johnson #522 #EE
```

□

11.1.2 Megatron 2002如何执行查询

考虑SQL查询的一般形式：

```
SELECT * FROM R WHERE <Condition>
```

Megatron 2002将做下列事情：

1. 读文件schema，以确定关系 R 中有哪些属性以及它们的类型。
2. 检查<条件>对于关系R的语义合法性。
3. 显示每个属性的名字作为列的头，然后画一条线。
4. 读文件名为 R 的文件，对于每一行：
   a) 检查是否符合条件；
   b) 若符合条件，则显示该行为一个元组。

要执行

```
SELECT * FROM R WHERE <condition> | T
```

Megatron 2002将做以下事情：

1. 如上所述处理查询，但省略第(3)步，该步骤是产生每列的头部并画一条线来分隔开列头和元组。

2. 将结果写到一个新文件/usr/db/T中。

3. 往文件/usr/db/schema中添加一个表示 T 的新条目，该条目和关于 R 的条目一样，只是关系名为 T，而不是 R。也就是说，关系 T 的模式与关系 R 的模式相同。

例11.2 现在考虑一个更复杂的查询，查询涉及到两个作为范例的关系Students和Depts的连接：

505

```
SELECT office
```

```
FROM Students, Depts
WHERE Students.name = 'Smith' AND
      Students.dept = Depts.name #
```

这个查询要求Megatron 2002对关系Students和Depts进行“连接”。即系统必须逐个考虑从这两个关系中各取一个元组所组成的每一对元组，并检验是否满足下列条件：

a) 这两个元组表示相同的系。

b) 学生的名字是Smith。

该算法可以非形式化地描述如下：

```
FOR each tuple s in Students DO
    FOR each tuple d in Depts DO
        IF s and d satisfy the where-condition THEN
           display the office value from Depts;
```

□

11.1.3 Megatron 2002有什么问题

毫无疑问，DBMS与臆想的Megatron 2002的实现不同。对于涉及大量数据或多用户的应用来说，这里描述的实现在许多方面不合适。下面列出其中的部分问题：

- 元组在磁盘上的排列不够好，缺乏对数据库进行修改时所需的灵活性。例如，如果在一个Students元组中将EE改为ECON，整个文件都需要重写，因为后续的每一个字符都需要在文件中后移两个位置。
- 查找代价太高。即使查询给出一个值或一组值，使查找可以集中针对某个元组上（例如例11.2的查询），也总是需要读入整个关系。在例11.2中，即使所要的只是学生Smith的元组，也必须查看整个的Student关系。
- 查询处理是一种“蛮干”的方法，而实际上对于执行连接这样的操作，有更加巧妙的方法可用。例如，以后会看到对于像例11.2中那样的查询，即使没有指明某个学生（Smith），也不必查看取自两个关系的每一对元组。
- 无法在主存储器中缓存有用的数据，所有的数据都来自硬盘，任何时候都是这样。 506
- 没有并发控制。几个用户可以同时修改同一个文件，从而导致不可预期的结果。
- 没有可靠性。发生故障时可能丢失数据，或者会有半途而废的操作。

本书的剩余部分将向你介绍解决这些问题的技术，希望你对这个研究感兴趣。

11.2 存储器层次

一个典型的计算机系统包括几个可以存储数据的不同部件。这些部件的数据存储容量至少有7个数量级，其访问速度也有7个或超过7个数量级。这些部件的每个字节的费用也各不相同，但是变化的范围要小些，最便宜的存储器与最昂贵的存储器相比每个字节的费用也许要相差3个数量级。毫不奇怪，具有最小容量的设备提供最快访问速度，其每个字节的费用也最高。存储器的层次结构如图11-1所示。

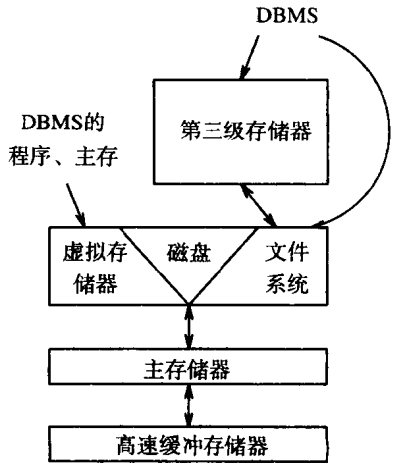


图11-1 存储器层次

11.2.1 高速缓冲存储器

存储器层次的最低层是高速缓冲存储器（cache）。单板高速缓存集成在微处理器芯片上，而附加的二级

507 高速缓存集成在另外的芯片上。高速缓存中的数据（包括机器指令）是主存储器中特定位置的数据的副本，主存储器位于存储器层次中高速缓存的上一层。有时，高速缓存中的数据值被改变了，而主存储器中相应的变化滞后于高速缓存的变化。尽管如此，某一时刻高速缓存中的每一个值还是与主存中某一位置的值相对应，高速缓存与主存之间的数据传输单位通常是少量字节。因此，可以认为，高速缓存中保留着单独的机器指令、整数、浮点数或短字符串。

当机器执行指令时，它在高速缓存中寻找指令以及这些指令要使用的数据。如果在高速缓存中找不到这些指令和数据，它就要到主存中去寻找，并将它们拷贝到高速缓存中去。由于高速缓存中只能保留有限数量的数据，通常必须将高速缓存中某些内容移出去，以便接纳新的数据。如果被移出高速缓存的内容自从它被复制到高速缓存以来一直没有改变，就不需要做任何事情。否则，如果数据已经被修改，那么新的值必须拷贝到它在主存中原先的位置。

当高速缓存中的数据被修改时，只有单个处理器的简单计算机不需要立即更新主存中相应位置的数据。然而，在某些多处理器系统中，允许多个处理器访问相同的内存，并且各处理器拥有各自私有的高速缓存。在这种情况下，直写（write through）对于高速缓存更新通常是必不可少的，即它立即修改主存中相应的位置。

在2001年，典型的高速缓存的容量达到了1兆字节。高速缓存与处理器之间数据的读写操作可以以处理器指令的速度执行，通常为几纳秒（1纳秒 = 10^{-9}秒）。另一方面，在高速缓存与主存之间移动一条指令或一个数据项的时间则要长得多，或许要100纳秒。

11.2.2 主存储器

计算机主存储器（简称主存，又称内存）是计算机的活动中心。可以认为，发生在计算机中的每一件事情，不论是指令的执行还是数据的操纵，都是作用于驻留在主存的信息上，尽管实际上使用的数据通常会转移到高速缓存中，正如在11.2.1节讨论过的那样。

在2001年，通常的机器配置的主存大约有100MB（10^8字节）。然而，配有更大容量主存的机器也能找到，比如10GB（10^{10}字节）或更大容量。

主存是随机访问的，这意味着在同一时间内可获得任何一个字节 ⊖。通常主存访问数据的
508 时间为10～100纳秒（10^{-8}～10^{-7}秒）。

计算机的量级是2的乘幂

通常在谈论计算机部件的大小或容量时就好像它们是10的方幂，例如：兆字节、千兆（吉）字节，等等。实际上，因为最有效的方法是将存储器芯片那样的部件设计成其容量恰好是2的乘幂的二进制数，所以事实上这些数字都是与之最接近的2的乘幂的简写。因为2^{10} = 1024，它接近于一千，于是常常假设2^{10} = 1000，并且在谈及2^{10}时使用“千”（前缀kilo）。2^{20}为“兆”（前缀mega），2^{30}为“吉”（前缀giga），2^{40}为“太”（前缀tera），2^{50}为“拍”（前缀peta）。尽管按照科学计数法，这些前缀依次指的是10^3、10^6、10^9、10^{12}和10^{15}。误差随着谈论的数量的增大而增长。1吉字节的值真正是1.074×10^9字节。

一般采用标准的缩略来表示这些数值，K、M、G、T和P分别依次表示千、兆、吉、太和拍。这样，16GB表示16吉字节，或者严格地说是2^{34}字节。由于有时想谈论传统的10的方幂的数值，所以将保留这些传统的数字单位来表示这些数值，而不是采用“千”、“兆”等前缀。例如，“一百万字节”就是1 000 000字节，而“一兆字节”则是1 048 576字节。

⊖ 尽管有些现代的并行计算机有一个主存储器被许多处理器以某种方式共享，这种方式使存储器的某一部分的访问时间不同，对于不同的处理器，可能相差3倍。

11.2.3 虚拟存储器

在写程序的时候，所使用的数据如程序变量、要读取的文件等都要占据一个*虚拟存储地址空间*（Virtual Memory Address Space），程序指令同样也占据一个它们自己的地址空间。许多机器使用32位字长的地址空间；也就是说有2^{32}个或者大约40亿个不同的地址。因为每个字节需要它自己的地址，一般可以认为虚拟存储器是4GB。

由于虚拟存储器空间比通常的内存大得多，一个完全被占用的虚拟存储器的大部分内容实际上是在硬盘上。在11.3节将讨论硬盘的通常操作，现在只需要意识到硬盘是被逻辑地分成多个块（block）。通常，硬盘上块的大小是在4KB～56KB。虚拟存储器以整个块为单位在硬盘和主存之间移动，在主存中的块常常被称做页(page)。机器硬件和操作系统允许虚存页进入主存的任何部分，并且虚拟地址能够正确地指向块中的每一个字节。

图11-1中有关虚拟存储器的路径代表传统的程序和应用的处理方法。它不代表数据库中管理数据的通用方式。然而，人们对*主存数据库系统*（main-memory Database system）的兴趣正在增加。主存数据库系统通过虚存来管理它们的数据，依靠操作系统，通过页面机制把所需要 509
的数据送入主存。类似于大多数应用系统，当数据量小到可以被保存在主存时最有用，因为这时不需要通过操作系统来交换数据。如果一台机器有32位字长的地址空间，那么主存数据库系统很适于那些不需要同时在主存中保留4千兆字节数据的应用（如果机器的实际主存比2^{32}字节小，那么数据量要更少些）。这个规模的空间对于许多应用来说是足够的，但是对于规模较大的DBMS应用来说就不够了。

因此，大规模数据库系统直接在硬盘上管理它们的数据。这些系统的大小仅受可以存储在计算机所有硬盘和其他可用存储设备的数据总量的限制。下一节将介绍这种操作模式。

11.2.4 二级存储器

每一台计算机基本上都有某种类型的*二级存储器*（secondary storage）。二级存储器是存储器的一种，它的速度要比内存慢得多，而存储容量则要比内存大得多，并且基本上是随机访问，访问不同数据项所需时间的差别相对较小（这些差别将在11.3节中讨论）。现代计算机系统使 510
用某种形式的磁盘作为二级存储器。通常，存储器以磁为介质，尽管有时候也采用光盘或者磁光盘。光盘或磁光盘比较便宜，但是许多不支持在盘上方便地写数据，或者根本就不支持写数据。因此，它们一般只用来存放不需要改变的归档数据。

摩尔定律

Gordon Moore在很多年以前就发现，集成电路在以多种方式改进时，其发展速度遵循着指数曲线，每18个月就要翻一番。遵循摩尔定律而变化的一些参数是：

1. 处理器的速度，即每秒钟执行的指令数，以及处理器的速度与费用之比。
2. 主存的每个二进制位的价格和可以置于一个芯片的二进制位数。
3. 磁盘每个二进制位的价格和最大的磁盘容量。

另一方面，还有一些不遵从摩尔定律的其他重要参数，如果它们确实在增长的话，它们增长得很慢。这些缓慢增长的参数包括对主存中数据的访问速度，或磁盘旋转的速度等。由于它们增长缓慢，延迟便逐渐地变得更大。也就是说，数据在存储器层次各级之间移动的时间与计算所花的时间相比变得越来越长。因此可以预计，在今后若干年内，主存储器将显得比高速缓存更加远离处理器，而磁盘上的数据将显得离处理器更远。实际上，这些显而易见的“距离”所产生的影响在2001年已经相当严重。

从图11-1中可见，磁盘被认为是既支持虚拟存储器，也支持文件系统。也就是说，在一些磁盘块保存一个应用程序的虚拟存储器页面的同时，其他磁盘块用于保存（部分）文件。在操作系统或数据库系统的控制下，文件以块的方式在磁盘与主存之间移动。将一个块从磁盘移动到主存是一次*磁盘读*，将一个块从主存移动到磁盘是一次*磁盘写*。这两种操作一般称之为一次*磁盘I/O*。主存的某些部分被用作*文件缓冲区*（buffer），即以块大小为单位保留这些文件的一部分。

例如：假定磁盘块的大小是4K字节，当你为了读而打开一个文件时，操作系统可能保留一个4K大小的主存块作为这个文件的缓冲区。开始，文件的第一个块被拷贝到缓冲区，当应用程序已经用过文件的那4K字节时，文件的下一个块就被送入缓冲区，置换原先的内容。这个过程如图11-2所示，将一直进行到读完整个文件者文件被关闭为止。

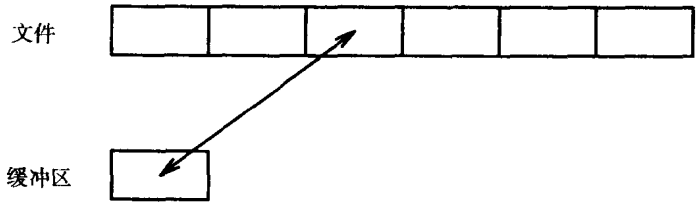


图11-2 文件及其主存缓冲区

DBMS自己管理磁盘块，而不是依赖操作系统的文件管理器在主存和二级存储器之间移动块。可是，不管是着眼于文件系统还是DBMS，管理中所涉及的问题本质上是一样的。在磁盘上读或写一个块大约要花10～30毫秒（0.01～0.03秒）。在这段时间内，一台普通的机器或许能执行数万条指令。这样，通常情况下，读或写一个磁盘块所花的时间，决定了对这个磁盘块的内容无论进行什么样的操作所花费的总时间。因此，至关重要的是尽可能地让含有需要访问的数据所在磁盘块已经在主存缓冲区内。这样就不必付出一次磁盘I/O的代价。在11.4节和11.5节中将进一步讨论这个问题，看一些处理在存储层次的不同级别之间移动数据所带来的高昂开销的例子。

在2001年，单个磁盘单元所拥有的容量已达到100GB或者更多。此外，机器可以使用多
511 个磁盘单元，于是拥有数百吉的外存的机器在目前也是有可能的。这样，二级存储器的速度将比一般的内存慢10^5倍，而容量则至少比一般内存大100倍。二级存储器显然也要比内存便宜得多。在2001年，磁盘单元的价格是每兆字节1～2美分，与此同时，主存的价格则是每兆字节1～2美元。

11.2.5 三级存储器

磁盘单元集合可以使数据容量相当大，但是有的数据库的数据量要比配置在单台机器甚至相当大的机器集群上的磁盘所能存储的容量还要大得多。例如，连锁零售商店保留着太字节数量级的有关他们经营情况的数据，而人造卫星每年要返回拍字节数量级的信息。

为了适应这样的需求，开发出*三级存储器*（tertiary storage），用以保存以太字节计数的数据容量。三级存储器的特点在于，与二级存储器相比，其读/写时间长，但是其容量比磁盘大，每个字节的费用也比磁盘少。主存储器为访问任何数据提供的访问时间是不变的；访问磁盘任何数据的时间差别仅取决于一个较小的因数；第三级存储器设备的访问时间是一个很宽的范围，这取决于数据与读/写点靠得有多近。下面是几种主要的三级存储器设备：

1. Ad-hoc*磁带存储器*（ad-hoc tape storage）。最简单的，而且过去几年中惟一一种使用三

级存储器的方式，是将数据存放在磁带卷或盒式带上，并将盒带存放在机架中。当想要从三级存储器中获取信息时，人工操作员要找到磁带并把磁带放置在磁带阅读器上。通过磁带的卷动找到信息的正确位置，然后把信息拷贝到二级存储器或主存储器上。为了向三级存储器写数据，首先要找磁带并将磁带定位到正确的位置，然后再将数据从磁盘复制到磁带。

2. *自动光盘机*（optical-disk juke box）。“自动光盘机”由若干个CD-ROM架所组成。CD是“compact disk”（袖珍盘）的缩写，ROM是“read-only memory”（只读存储器）的缩写。这种光盘通常用于分发软件。光盘上的二进制位用黑或白的小区域来表示，所以通过对这些点投射激光并看其是否被反射来读取这些数据位。自动光盘机中的自动机械臂能快速选取任意一个CD-ROM，并且将它送到CD阅读器，CD中的全部或部分内容就可被读入第二级存储器。

3. *磁带仓*（tape silo）。“磁带仓”是一个房间大小的设备，其内部有磁带机柜。通过自动机械臂将要访问的磁带送到多个磁带阅读器中的一个。因此磁带仓是早期ad-hoc磁带存储器的一种自动化方式。由于它使用计算机编目控制和自动磁带检索过程，因此它比人力系统至少快一个数量级。

512

在2001年盒式磁带的容量高达50吉字节，因此磁带仓可保存许多个太字节的数据。CD的标准容量大约是2/3吉字节，已形成的下一个标准规定大约为2.5吉字节（DVD），并且该标准将逐渐流行起来。容量在多个太字节范围内的CD-ROM自动光盘机也将面世。

从三级存储器设备访问数据所花费的时间从几秒钟到几分钟不等。自动光盘机或磁带仓中的自动机械臂可以在几秒钟内找到所要的CD-ROM或盒式带，而操作员很可能要花几分钟来定位和检索磁带。一旦被装入到阅读器中，便可以在零点几秒钟内访问CD的任何部分，但是要把磁带的正确区段移动到磁带阅读器的磁头下则另外还要花许多秒钟。

总的来说，访问第三级存储器比访问第二级存储器大约要慢1000倍（毫秒与秒的比值）。然而，单个的第三级存储器单元的容量可以比第二级存储器设备大1000倍（吉字节与太字节的比值）。图11-3按双对数度量显示了在已经研究过的4个级别存储器层次中，访问时间与容量之间的关系。Zip盘和软盘（塑料磁盘）是很普遍的存储设备，尽管它们不是通常用于数据库系统的第二级存储器，但在此也将它们包括进来。在图11-3中，横轴是用来度量秒的10的指数，例如，-3的意思是10^{-3}秒，或者说1毫秒。纵轴是用来度量字节的10的指数，例如，8代表100兆字节。

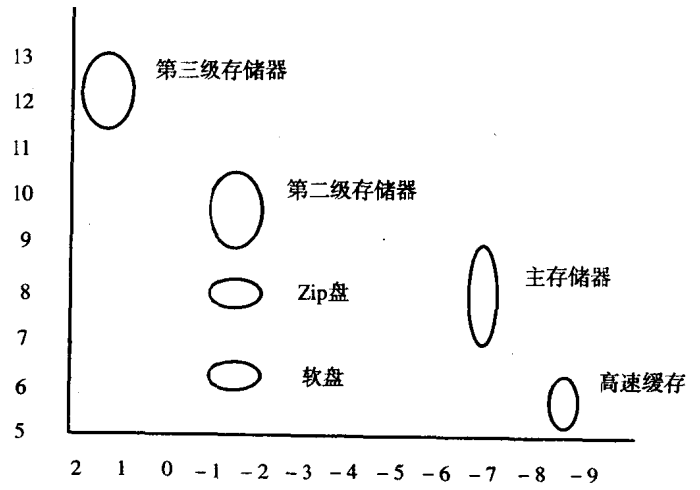


图11-3 存储器层次中各个级别的访问时间与容量关系

11.2.6 易失和非易失存储器

区别存储设备的另外一个特性是，存储在其上的数据是*易失的*还是*非易失的*。一个易失的设备，当切断电源时，会“忘记”它所存储的数据。但是，一个非易失的设备，当设备被关闭
513 或电源发生故障时，可期望保持它的内容完整无缺，甚至长期保存。易失性问题是一个很重要的问题，因为DBMS颇具特色的性能之一就是，即便在发生电源故障等错误情况下仍有能力保留其数据。

磁材料在没有电的情况下仍能保持其磁性，因此，磁盘、磁带这样的设备是非易失的。同样，光学设备如CD，即使在没有电的情况下，仍然保留在其上所刻录的黑白圆点。的确，对于许多设备来说，没有方法改变写在其表面的内容。因此，所有二级存储器和三级存储器设备在本质上都是非易失的。

另一方面，主存储器通常属于易失的存储设备。如果允许二进制位的值在一分钟后降低，那么存储器芯片就可以设计成比较简单的电路，这种简单性可降低芯片每一位的成本。实际的情况是，代表一个二进制位的电荷会缓慢地流出贡献给那个位的区域。结果，被称为*动态随机访问存储器*（DRAM）的芯片需要对整个内容周期性地读和写。如果切断电源，这种刷新就不会发生，芯片也就很快丢失它所存储的内容。

运行在配置了易失主存的机器上的数据库系统，在认定将要改变数据是数据库的一部分之前，必须先将改变的数据备份到磁盘上，否则就会有因电源故障而丢失信息的危险。因此，查询和修改数据库必然要涉及到大量的磁盘写操作。如果无需在任何时候都保留所有的信息，那就可以避免某些写磁盘操作。一个可供选择的方法是使用一种非易失的主存储器。一种被称做*快闪存储器*（flash memory）的新型存储器芯片就是非易失的，它的价格正在变得越来越便宜。另外一种可供选择的方法是用一般的存储器芯片建立一种RAM盘，用电池组作为它的主电源的后备。

11.2.7 习题

习题11.2.1 假定在2001年，普通的计算机有一个主频为1500MHz的处理器，有一个40GB硬盘以及容量为100MB的主存储器。假设摩尔定律（这些参数每18个月翻一番）无限期有效。

* a) 什么时候太字节磁盘将成为通用存储设备？

b) 什么时候吉字节主存将成为通用存储设备？

c) 什么时候太赫兹处理器将成为通用存储处理器？

d) 到2008年，什么样的配置（处理器、磁盘、主存）将成为典型的配置？

! **习题11.2.2** 来自24世纪Trek星球上的下一代机器人德特骄傲地宣布，他的处理器以“12太
514 次操作/秒”的速度运行。尽管一次操作与一个周期可以不同，但在此假定它们是相同的，并且摩尔定律在未来300年依然有效。如果是这样，德特的处理器的真正速度是多少？

11.3 磁盘

使用二级存储器是数据库管理系统的重要特性之一，而二级存储器几乎无例外的基于磁盘。这样，为了给出DBMS实现中采用的许多思想的理由，必须详细地研究磁盘操作。

11.3.1 磁盘结构

图11-4给出了磁盘驱动器的两个主要移动部件；一个是*磁盘组合*（disk assembly），另一个是*磁头组合*（head assembly）。磁盘组合由一个或多个圆形的*盘片*（platter）组成，它们绕着一

根中心主轴旋转。圆盘的上表面和下表面覆盖了一层薄薄的磁性材料，二进制位存储在这些磁性材料上。0是通过在一个方向上定向磁化的一个小区域表示，而1则是通过在相反方向上定向磁化的一个小区域表示。尽管直径从一英寸到几英尺的磁盘都已经制造出来了，但盘片的直径一般是3.5英寸。

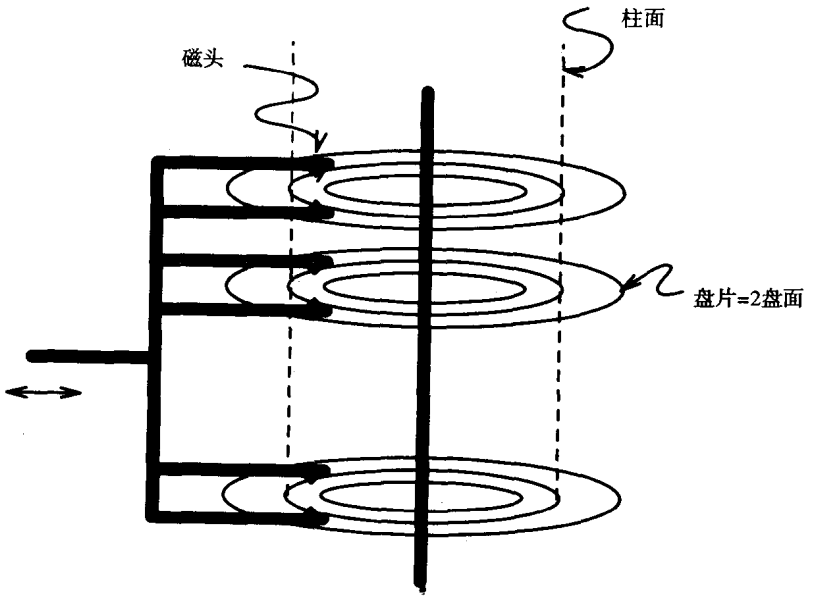


图11-4 一个典型的磁盘

将二进制位的存储位置规划成*磁道*（track），磁道是单个盘片上的同心圆。如同从图11-5的俯视图可以看到，除了最靠近主轴的区域外，磁道占据了大部分盘面。磁道由许多个点组成，每一个点代表一个由它的磁化方向决定的二进制位。

515

磁道被规划成扇区（sector）。扇区是被间隙（gap）分割的圆的片断，间隙在两个方向上均不被磁化[⊖]。对于读写磁盘来说，扇区是不可分割的单位。对于磁盘错误来说，扇区也是一个不可分割的单位。如果一部分磁化层被损坏，以致它不再能存储信息，那么包含这个部分的整个扇区也不能再使用。间隙大约占整个磁道的10%，用于帮助标识扇区的起点。在11.2.3节中所提到的“块”，是在磁盘与主存之间传输数据的逻辑单元，由一个或多个扇区组成。

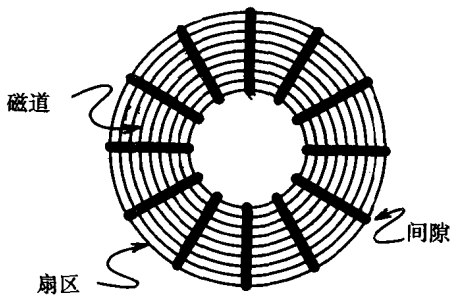


图11-5 磁盘的俯视图

图11-4中的第二个可移动部件是磁头组合，它承载着磁头。每一个盘面有一个磁头，它悬浮在盘面上，与盘面紧密相邻，但是绝对不与盘面接触（否则就要发生“磁头损毁”，盘片和存储在它上面的一切数据都被破坏）。磁头读出位于它下方的盘面的磁化方向，也能改变其磁化方向，以便在磁盘上写信息。每个磁头被固定在一个磁头臂上，所有盘面的磁头随着磁头臂一

⊖ 在图11-5中，用相同的扇区数显示每一个磁道。然而，正如将在例11.3中所讨论的，每个磁道的扇区数可以不同，靠外圈磁道的扇区数比靠内圈磁道的扇区数多。

同移进移出，磁头臂是固定的磁头组合的一部分。

11.3.2 磁盘控制器

一个或多个磁盘驱动器被一个*磁盘控制器*（disk controller）所控制，磁盘控制器是一个小处理器，具有如以下功能：

1. 控制移动磁头组合的机械传动装置，将磁头定位到一个特定的半径。在该半径位置，每个盘面都有一个磁道处于那个盘面的磁头之下，使该磁道可读可写。同一时刻位于磁头下的各个磁道构成了一个*柱面*（cylinder）。

2. 选择一个准备读写的盘面，并从位于
516 该盘面的磁头下的磁道上选择一个扇区。控制器还负责识别旋转主轴到达定点的时刻，在此定点位置，所寻扇区在磁头下开始移动。

3. 将从所寻扇区读取的二进制位传送到计算机的主存，或者将从主存储器写入的二进制位传送到所期望的扇区。

图11-6给出了一台简单的单处理器计算机的示意图。处理器经由数据总线与主存储器和磁盘控制器进行通信。一个磁盘控制器能够控制多个磁盘，在这个示意图中计算机有三个磁盘。

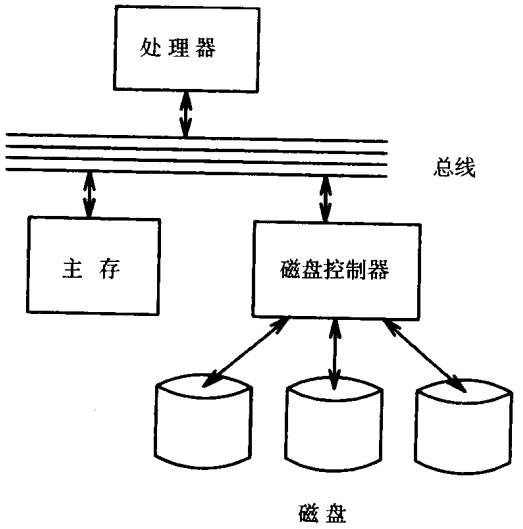


图11-6 一个简单计算机系统示意图

11.3.3 磁盘存储特性

磁盘技术处于不断变化之中，存储一个二进制位所需要的空间在迅速减少。在2001年，与磁盘相关的一些典型参数是：

- *磁盘组合的旋转速度*。5400 RPM（转/分钟），即每11毫秒旋转一周。这只是常见的转速，虽然再高一些或低一些的转速也都有。
- *每个磁盘的盘片数*。一般的磁盘驱动器大约有5个盘片，因此有10个盘面。然而普通的“软”盘或“Zip”盘只有一个盘片，两个面。带有多达30个盘面的磁盘驱动器已经出现。
- *每个盘面的磁道数*。一个盘面可以有多达20 000个磁道，而软盘的磁道数要少得多。参见例11.4。
- *每个磁道的字节数*。一般的磁盘存储器，每个磁道有大约有一百万个字节，但软盘磁道
517 所保存的字节数要少得多。如前已说明的，磁道被分成扇区。图11-5显示出每个磁道有12个扇区，但是实际上现在磁盘的每个磁道的扇区数可以多达500个。每个扇区可能有数千个字节。

扇区与块的比较

请记住，“扇区”是磁盘的物理单元，而“块”则是使用磁盘的软件系统——例如操作系统或者DBMS——所建立的逻辑单元。正如我们已经提到过的，目前一般的块至少有扇区那么大，块由一个或多个扇区组成。然而，没有任何理由来解释为什么块不能是一个扇区的一小部分，即不能将若干个块挤压到一个扇区中。实际上，一些较老的系统采用过这种存储策略。

例11.3 Megatron 747磁盘是一种2001年出品的较大型驱动器，它具有下列特性：

- 8个盘片，16个盘面。
- 每个盘面有2^{14}即16 384个磁道。
- 每个磁道平均有2^7 = 128个扇区。
- 每个扇区有2^{12} = 4096个字节。

整个磁盘容量是：16个盘面，乘以16 384磁道，乘以128扇区，再乘以4096字节的乘积，即2^{37}字节。这样，Megatron 747就是一块128GB的磁盘。一个磁道存放128 × 4096字节，或512KB。如果一个块的容量是2^{14}即16 384字节，那么一个块使用4个连续的扇区，一个磁道上有128/4 = 32个块。

Megatron 747盘面的直径是3.5英寸。磁道位于盘面靠外1英寸的区域，靠内的0.75英寸区域内没有磁道。径向位密度是每英寸16 384位，因为这就是磁道数。

环绕磁道的位密度要大得多。首先假定，每个磁道平均有128个扇区。假设间隙占据磁道的10%，这样每个磁道为512KB（或4M二进制位），占据磁道的90%。最外圈的磁道长度是3.5π即大约11英寸。这个距离的90%即大约9.9英寸，存放4兆位。因此，磁道占据部分的位密度大约是每英寸420 000位。

另一方面，最内圈磁道的直径仅为1.5英寸，在0.9 × 1.5 × π，即大约4.2英寸的长度上也要存放4兆位。这样，内圈磁道的位密度大约是每英寸1兆位。 518

如果扇区数和位数保持相同，磁道内圈密度与外圈密度就会相差得太远。与现在其他的磁盘驱动器一样，Megatron 747的外圈磁道比内圈磁道存储更多的扇区。例如，对于磁盘的中间1/3磁道部分，每个磁道存放128个扇区。在靠内的1/3磁道部分，每个磁道存放96个扇区。而靠外的1/3磁道部分，每个磁道存放160个扇区。如果这样做，那么在最内圈磁道到最外圈磁道上每英寸的位密度位于530 000 ~ 742 000之间。 □

例11.4 磁盘中的低端产品是标准的3.5英寸软盘。它有两个面，每面40个磁道，总计80个磁道。格式化成MAC格式或者PC格式后，这个磁盘大约能容下1.5MB数据，或者说每个磁道中有150 000个二进制位（18 750字节）。大约有1/4可用空间被间隙和其他的格式化（无论哪种格式化）开销所占据。 □

11.3.4 磁盘访问特性

在数据库管理系统的研究中，不仅要理解数据在磁盘中存储的方法，而且要理解其操纵的方法。由于所有计算都在主存或高速缓存中进行，关于磁盘，我们一般关心的惟一问题是如何在磁盘与主存之间移动数据块。正如11.3.2节所叙述的，在下述情况下进行块（或者说组成块的连续扇区）的读或写：

a) 磁头被定位到包含目标块的磁道所在的柱面。

b) 在整个磁盘组合转动时，组成该块的扇区移动到磁头下面。

从发出读块命令的时刻起，到块的内容出现在主存中，其间所花费的时间称为磁盘的*延迟时间*（latency）。这个时间可以细划分为如下几部分：

1. 处理器和磁盘控制器处理请求所花费的时间，通常是零点几毫秒，这个时间将忽略不计。同时还将忽略由于争用磁盘控制器（同一时刻其他进程可能正在读写磁盘）所花费的时间，并且忽略由于诸如争用总线等原因所引起的其他延时。

2. *寻道时间*（seek time）：将磁头组合定位到合适柱面所花费的时间。如果磁头恰巧是在

合适的柱面上，寻道时间就是0。如果不是，那么磁头就需要一些时间去开始移动和再次停下来，再加上与磁头移动距离大致成正比的一段附加时间。通常的最小寻道时间，从起动、移过一个磁道，再停住，大约是几毫秒。最长寻道时间，即
519 磁头要跨越所有的磁道，大约是在10～40毫秒的范围内。图11-7给出了寻道时间如何随距离而变化。该图表明，寻道时间开始于某个值x（跨越一个柱面距离的时间），并且显示，最大寻道时间是在$3x$～$20x$的范围内。平均寻道时间经常被用来作为标识磁盘速度特征的一种度量。例11.5将讨论如何计算该平均值。

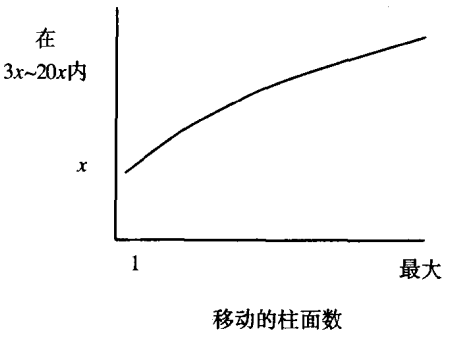


图11-7 寻道时间随行进距离而变化

3. 旋转延迟（rotational latency）：磁盘转动到组成该块的第一个扇区到达磁头时所需要的时间。常用的硬盘大约每10毫秒完整地转动一周。平均来说，当磁头到达目标扇区的柱面时，该扇区大约要转半周，所以平均转动延迟在5毫秒左右。图11-8图示了旋转延迟问题。

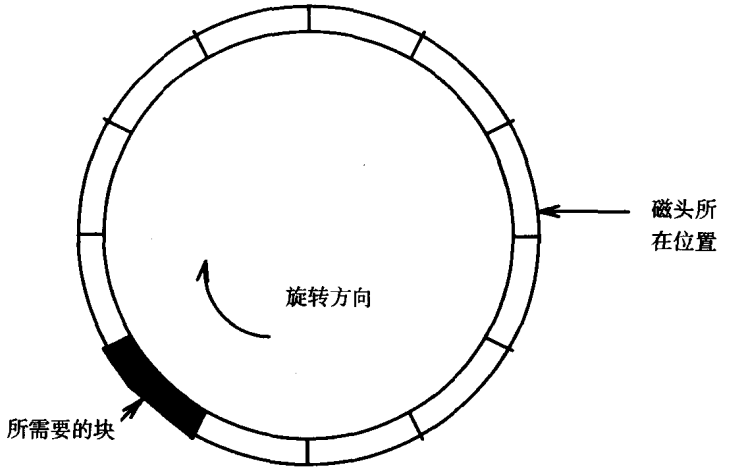


图11-8 旋转延迟的原因

4. 传输时间（transfer time）：块的扇区以及各扇区之间的间隙旋转通过磁头所需要的时间。如果磁盘每个磁道有大约250 000字节，大约在10毫秒内旋转一周，每秒钟大约能够从磁盘读
520 取25兆字节。16 384字节块的传输时间大约是2/3毫秒。

例11.5 考察从Megatron 747磁盘读取16 384字节所花费的时间。首先需要知道磁盘的时间属性：

- 磁盘以每分钟7200转的速度旋转，即8.33毫秒内旋转一周。
- 在柱面之间移动磁头组合，从起动到停止需要1毫秒，每移动1000个柱面另加1毫秒。这样，磁头在1.001毫秒内移动一个磁道，从最内圈移动到最外圈，移过16 383条磁道大约用时17.38毫秒。

计算一下读取16 384字节块的最小、最大和平均时间。由于忽略了因使用控制器而引起的额外开销和争用，所以最小时间就是传输时间。也就是说，磁头已经定位在块所在的磁道上，而且块的第一个扇区即将从磁头下面通过。

由于Megatron 747的每个扇区有4096字节（见例11.3关于磁盘的物理规格说明），所以该块要占用4个扇区。为此，磁头必须越过4个扇区和扇区之间的3个间隙。回忆一下，间隙占圆周的10%，而扇区占其余的90%。围绕着圆周有128个间隙和128个扇区。由于间隙合在一起覆盖36度圆弧，而扇区覆盖其余的324度圆弧，所以被3个间隙和4个扇区覆盖的圆弧的总度数为：

$$36 \times \frac{3}{128} + 324 \times \frac{4}{128} = 10.97$$

传输时间是（10.97/360）×0.008 33 = 0.000 253秒，即1/4毫秒。也就是说10.97/360是读取整个块所需的旋转时间，0.008 33秒是旋转360度所需的时间。

现在来看一下读该块的最大可能时间。在最坏的情况下，磁头被定位在最内圈柱面，而要读的块是在最外圈柱面上（或者相反）。这样，控制器必须做的第一件事就是移动磁头。正如前面已经知道的，移动Megatron 747磁头跨越全部柱面所花费的时间大约是17.38毫秒。这个数量就是读盘的寻道时间。

当磁头到达正确的柱面时，可能出现的不理想情形是，所需块的起点刚好从磁头下面越过。假定必须从块的起点开始读，那么实际上就必须等待完整一圈的时间，或者说8.33毫秒，使块的起点再次到达磁头下，还必须等待读整个块的0.25毫秒的传输时间。这样，最坏情况下的延迟时间是17.38 + 8.33 + 0.25 = 25.96毫秒。 521

最后来计算读一个块的平均时间。延迟时间的两部分很容易计算：传输时间总是0.25毫秒，平均旋转延迟时间是磁盘旋转半周所需要的时间，即4.17毫秒。可以假定，平均寻道时间正好是越过一半磁道所需要的时间。然而，这样计算并不很准确，因为通常磁头起初是位于中间位置附近的某个位置，因此平均来说，磁头到达所要求柱面需移动的距离小于跨越一半磁道的距离。

更为详细的估算磁头必须移动的平均磁道数可按如下方法获得。假设磁头起初以相同的概率被定位在16 384个柱面上的任一位置。如果是在柱面1或柱面16 384，那么要移动的平均磁道数是（1+2+…+16 383）/ 16 384，即大约8192道。如果是在柱面8192，即中间位置，则磁头移进或移出的可能性相同，而且无论移进还是移出，移动距离平均来说大约都是总磁道数的四分之一，即4096磁道。计算表明，当磁头的初始位置从柱面1到柱面8192变化时，磁头需要移动的平均距离从8192到4096按二次方曲线减少。同样，当磁头的初始位置从柱面8192变化到柱面16 384时，磁头需要移动的平均距离按二次方曲线回升到8192，如图11-9所示。

磁盘控制器结构的发展趋势

由于数字硬件价格的急剧下跌，磁盘控制器开始变得越来越像它们自己的计算机，拥有通用处理器和相当大的随机访问存储器。借助这样的附加硬件可以做许多事情，磁盘控制器可将磁盘的全部磁道读出并存入它们的本地存储器中，即使只从磁道中请求一个块也如此。当需要访问单一磁道的全部或大部分块时，这种能力大大减少了对块的平均访问时间。11.5.1节将讨论全磁道或全柱面的读写应用。

如果把图11-9中的参数对所有初始位置积分，就会发现，行进的平均距离是跨越磁盘路程的三分之一，或者说是5461个柱面。也就是说，平均寻道时间是一微秒加上跨越5461磁道的时间，即1 + 5461/1000 = 6.46毫秒[⊖]。测算的平均延迟时间为：6.46 + 4.17 + 0.25 = 10.88毫秒。

⊖ 这个计算忽略了完全不移动磁头的可能性，但是在假设的16 384次随机块请求中，那种情况只发生一次。另一方面，正如将在11.5 节中所见到的，随机块请求并不是一种必需的好的假设。

522 这三项依次代表平均寻道时间、平均旋转延迟时间和传输时间。 □

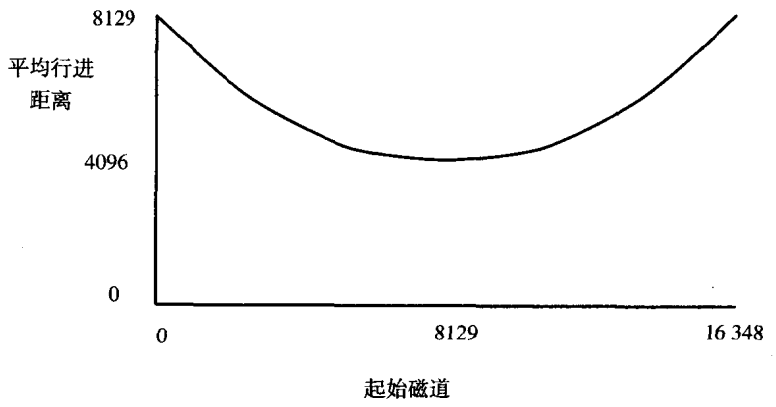


图11-9 平均行进距离是磁头初始位置的函数

11.3.5 块的写操作

用最简单的方式写一个块的过程与读一个块的过程非常类似。磁头被定位在合适的柱面上后，等待合适的扇区转到磁头下面。只不过这时是用磁头写入新数据，而不是读磁头下的数据。最大、最小和平均写入时间与相应的读出时间完全相同。

如果想要检测块的写入是否正确，情况将会变得复杂起来。因为这样做需要等待一次附加的旋转，并且要回读每一个扇区，以检查写入的数据是否确实存储在那里。11.6.2节中将讨论通过使用校验和来检测写入是否正确的一个简单方法。

11.3.6 块的修改

不可能直接修改磁盘上的一个块。而且，即使是只想修改少量字节（例如存储在块上的若干元组之一的一个分量），也必须做如下几项工作：

1. 将块读入主存储器。
2. 对主存储器中块的副本做出所要求的修改。
3. 将块的新内容回写到磁盘。
4. 如果合适，检查写操作是否被正确执行。

块修改的总时间是这样一些时间的和：读的时间、在主存储器中执行更新的时间（这个时间与读写磁盘的时间相比通常可以忽略不计）、写的时间，如果执行校验的话，还有磁盘的另
523 一次旋转时间[⊖]。

11.3.7 习题

习题11.3.1 Megatron 777磁盘具有以下特性：

1) 有10个盘面，每个盘面有10 000个磁道。
2) 磁道平均有1000个扇区，每个扇区是512字节。
3) 每个磁道的20%用于间隙。

⊖ 现在可能想知道写入刚刚读过的块的时间是否与执行一次“随机的”块写入的时间相同。如果磁头停留在它们所在的位置，那么已经知道，必须等待一次完整的旋转时间来写，但是寻道时间为零。然而，由于磁盘控制器不知道应用程序何时将完成写块的新值，在写块的新值的请求被执行之前，磁头可能已经移动到另一个磁道去执行另外一些磁盘I/O。

4) 磁盘转速是10 000转 / 分钟。

5) 磁头移动n个磁道所需要的时间是1+ 0.001n毫秒。

回答下列有关Megatron 777的问题：

* a) 磁盘的容量是多少？

b) 如果所有的磁道都拥有相同的扇区数，那么一个磁道扇区中的位密度是多少？

* c) 最长寻道时间是多少？

* d) 最长旋转延迟时间是多少？

e) 如果一个块是16 384字节（即32扇区），一个块的传输时间是多少？

! f) 平均寻道时间是多少？

g) 平均旋转延迟时间是多少？

! **习题11.3.2** 假设Megatron 747磁道的磁头位于磁道2048，即在跨越磁道行程的1/8处。并假设下一个请求是针对随机磁道上的一个块。计算读这个块的平均时间。

*!! **习题11.3.3** 在例11.5的末尾，计算了磁头从一个随机选取的磁道移动到另一个随机选取的磁道之间的平均距离，并且发现，该距离为总磁道数的1/3。假设每个磁道的扇区数与磁道的长度（或半径）成反比，那么所有磁道的位密度相同。另外又假设，需要将磁头从一个随机的扇区移动到另一个随机的扇区。由于扇区倾向于聚集在磁盘的外侧，所以可以期待磁头平均移动距离小于跨越磁道行程的1/3。就像Megatron 747一样，假设磁道占据从0.7 5 ~ 1.75英寸的半径范围，计算磁头在两个随机扇区之间移动时跨越的平均磁道数。 524

!! **习题11.3.4** 例11.3的末尾曾建议，如果将磁道分成三个区域，在每个区域设置不同的扇区数，就可以减小磁道的最大密度。如果三个区域之间的划分可以在任何半径上进行，每个区域的扇区数可以不同，而且惟一的约束条件是在一个盘面的16 384个磁道上的总字节数为8GB，如何选择这5个参数（区域之间两个划分的半径，以及三个区域中，每个区域的每个磁道的扇区数）使得任何磁道的最大密度为最小？

11.4 有效使用二级存储器

大多数算法研究中，人们都假设数据是在主存储器中，访问任何数据项所花费的时间一样。这种计算模式通常被称为计算的“RAM模型”，或称为随机访问计算模型。然而，当实现一个DBMS时，必须假设数据不在主存储器中。因此，所设计的有效的算法中，必须考虑使用第二级存储器，或许甚至要考虑使用第三级存储器。在二级存储器或三级存储器中处理极大量数据的最佳算法，通常与处理同样问题的最佳主存储器算法不同。

本节将主要考虑主存储器与二级存储器之间的相互作用。特别指出，设计算法时注意限制磁盘访问次数很有好处，即使这种算法作为主存算法并非最有效时也如此。一个相似的原理适用于存储器层次中的每一级。如果记住高速缓存的大小并设计算法，以便移动到高速缓存的数据能被多次使用，那么主存储器算法有时也能被改进。同样，使用三级存储的算法需要考虑在三级存储器与二级存储器之间移动的数据量，而且，即使是以存储器结构层次的较低层中的更多工作为代价，来最小化这个数量，也是明智之举。

11.4.1 计算的I/O模型

设想一台运行DBMS的简单计算机，正尝试着为那些以查询和修改数据库等不同方式访问数据库的众多用户提供服务。另外，假设该计算机有一个处理器、一个磁盘控制器和一个磁盘。数据库本身太大以至于主存储器中容纳不下。数据库的关键部分可以缓存在主存储器中，但是

通常，用户要访问的数据库的每一个片断，必须先从磁盘中检索。

由于存在着许多用户，每个用户频繁地发出磁盘I/O请求，磁盘控制器经常会有一个请求队列，先假定该请求队列以先到先服务（first-come-first-served）原理处理请求。该策略使得
525 一个指定用户给出的每一个请求看起来都像是随机请求（也就是说，在请求之前磁头处于一个随机的位置），即使该用户正在读属于单个关系的块，而且那个关系被存储在单个磁盘柱面上。在本节的后面部分，将讨论如何以不同的方法改善系统的性能。但是，下面定义的计算的I/O模型的规则继续有效：

I/ O开销的主导地位：如果一个块需要在磁盘与主存之间移动，那么执行读或写所花费的时间或许要比用于操纵主存中的数据所花费的时间长得多。这样，块访问（读和写）次数就是算法所需要的时间的很好的近似值，而且应该被最小化。

假定，磁盘是Megatron 747，块大小为16KB，时间属性由例11.5决定。特别是，读或写一个块的平均时间大约是11毫秒。

例11.6 假设数据库有关系 R，有一个对元组的查询请求，该元组有一个确定的键值 k。正如所看到的那样，最好为 R 创建一个索引，用于标识带有键值 k 的元组出现的磁盘块。而索引是否告知这个元组出现在块的什么位置通常并不重要。

理由是读这个16KB的块大约要花11毫秒。在11毫秒内，一个新式的处理器能够执行上百万条的指令。然而，一旦块进入主存，即使采用最笨的线性搜索，搜索键值 k 仅需执行成千条指令。因此在主存中执行搜索的附加时间将小于块访问时间的1%，忽略附加时间不会有任何问题。 □

11.4.2 二级存储器中的数据排序

下面用一个扩展的例子来讨论如何在计算开销的I/ O模型下改变算法。这个例子就是当数据量比主存容量大得多的情况下的排序问题。首先介绍一个特定的排序问题，并且给出执行排序的机器的若干细节。

例11.7 假定有一个由10 000 000个元组构成的大型关系。每个元组用拥有若干个字段的一个记录来表示，其中的一个字段为*排序键*（sort key）字段，如果不与其他类型的键发生混淆的话，或者就叫“键字段”。排序算法的目的是按排序键值的升序来对记录排序。

排序键可以是SQL通常意义上主键的“键”，即那种保证每个记录取惟一值的主键，也可以不是。如果允许排序键有重复值，那么具有等值排序键的记录的任意顺序都是可接受的。为
526 简单起见，假定排序键具有惟一性。

记录（元组）以每块16 384字节填到磁盘块中。假设每块有100条记录。也就是说，每条记录160字节。块中还需要存储一些额外的信息（在12.2节讨论过），100条这样大小的记录占据一块中的16 384个字节。于是，关系R占了100 000块，共16.4亿个字节。

执行排序操作的机器拥有一个 Megatron 747磁盘并且还有 100MB 的主存作为关系的块的缓冲区。实际的主存储器容量更大，但是其剩余部分被系统所用。在容量为 100MB 的主存（实际为 100×2^{20} 字节）中能容纳的块数是 $100 \times 2^{20}/2^{14}$，或 6400 块。 □

如果数据能全部装入主存，那么有很多众所周知的算法可以使用[⊖]。各种各样的“快速排序法”（Quicksort）通常被认为是最快的算法。选取的快速排序算法仅仅使用键字段排序，该

⊖ 见D.E.Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching, 2nd Edition*, Addison-Wesley, Reading MA,1998.

键字段还带有指向整个记录的指针。仅仅当键及其指针按顺序排定时，指针才能正确指向每一个记录所在的位置。

遗憾的是，当需要二级存储器保存数据时，这些想法并不十分奏效。当数据的大部分在二级存储器中时，更好的排序方法是那些使每一个块在主存与辅存之间仅需移动很少次数的算法。通常，这些算法进行少量的几趟操作，在一趟中每个记录被读入主存一次，写入磁盘一次。11.4.4将讨论这种算法。

11.4.3 归并排序

读者可能熟悉一种被称为“归并排序”（Merge-Sort）的排序算法，该算法将几个小的已排好序的表归并为一个较大的排好序的表。为了归并两个已排序的表，要反复比较每个表剩余部分中最小的键，把其中键值较小的记录移出，如此反复，直到一个表被取尽。这时候，按上述选取顺序输出的表，加上未取尽的表的剩余部分，就是已排好序的完整的记录集。

例11.8 假设有两个有序表，各有4个记录。为了使事情简化，我们以记录的键来表示记录，没有其他数据，还假设键是整数。一个有序表是（1，3，4，9），而另一个有序表是（2，5，7，8）。在图11-10中，可以看到归并处理的步骤。

步骤	表1	表2	输出
start	1,3,4,9	2,5,7,8	none
1)	3,4,9	2,5,7,8	1
2)	3,4,9	5,7,8	1,2
3)	4,9	5,7,8	1,2,3
4)	9	5,7,8	1,2,3,4
5)	9	7,8	1,2,3,4,5
6)	9	8	1,2,3,4,5,7
7)	9	none	1,2,3,4,5,7,8
8)	none	none	1,2,3,4,5,7,8,9

图11-10 归并两个有序表生成一个排序表

第一步，比较两个表的首元素1和2。由于1 < 2，1被从第一个表中移出，成为输出的第一个元素。第二步，两个表剩余部分的首元素是3和2，进行比较，2获胜并输出。归并继续进行，直到第7步，此时第二个表被取尽。这时，第一表的剩余部分恰巧只有一个元素，将其输出，归并完成。请注意，输出是有序的，而且必然是这样，这是由于每一步都是选取剩余元素中最小的那一个。 □ 527

在主存中归并的时间与两表长度的总和成线性关系。原因是，由于给定的表已经排序，只有两个表的首元素，才有可能是所有未被选择元素中的最小的元素，对它们进行比较的时间是恒定的。如果有 n 个元素要排序，传统的归并排序算法需要进行 $\log_2 n$ 趟递归排序。该算法描述如下：

基础：如果有一个单元素表要排序，由于顺序已经存在，所以不需要做任何事情。

归纳：如果要排序的表含有一个以上的元素，则将表分为两个同样长度的表。如果原始表的长度为奇数，则两者长度要尽可能地接近。递归排序两个子表。然后归并所得到的排序表为一个排序表。

这个算法的分析是众所周知的，而且在这里也不太重要。简而言之，表示 n 个元素排序的时间 $T(n)$，等于某常数乘以 n（将原表分开，并归并所得到的排序表），加上大小为 $n/2$ 的两个表的排序时间。也就是说，$T(n) = 2T(n/2) + an$，a为常数。这个递归公式的结果是 $T(n)$

$= O(n\log n)$, 即与 $n\log n$ 成正比。

11.4.4 两趟多路归并排序

下面介绍的*两趟多路归并排序*（Tow-Phase, Multiway Merge-Sort，经常简写为TPMMS）的归并排序法的一种变体，我们将在例11.7所描述的机器上用这种方法为该例的关系排序。在
528 许多数据库应用中，这是一个常被采用的排序算法。简单地说，该算法包括：

- *第一趟*：排序主存大小的数据片断，使得每个记录都是一个有序表的一部分，该表正好可以装在可用主存内。这样，可能有任意个这样的有序子表，我们在下一阶段归并这些子表。
- *第二趟*：归并所有的有序子表以形成单个排序表。

我们首先观察到的是，由于数据是在第二级存储器中，因此不要在一个记录或几个记录的基础上开始递归。理由是，当要排序的记录存放在主存中时，归并排序不如其他算法那么快。这样，通过把记录装入整个内存来开始递归，并且使用诸如快速排序这样的恰当的主存排序算法。根据需要多次重复如下处理：

1. 在所有可用的主存中存入来自原始关系的需要排序的块。
2. 对主存中的记录排序。
3. 将排序后的记录从主存储器写入二级存储器的新块中，并形成一个有序子表。

在第一趟结束时，原始关系的所有记录已经被读入主存一次，而且成为已经被写入二级存储器的主存大小的有序子表的一部分。

例11.9 考虑例11.7中描述的关系。将100 000个块中的6400个块装入主存储器。这样将装入存储器16次，在主存中为记录排序，并将有序子表写入磁盘。16个子表的最后一个要比其余的子表短，它仅占据4000 块，而其余15个子表占据6400块。

这个阶段要花费多长时间呢？这里要读100 000个块，每个块读一次，还要写100 000个新块。这样就有200 000次磁盘I/O。现在假定块在磁盘上的排序是任意的，正如在11.5节将看到的那样，这一假定能大大地加以改进。而在我们的随机性假定中，每一个块的读或写大约要花费11毫秒。这样，第一趟的I/O时间是2200秒，即37分钟，每个有序子表需要的时间超过2分钟。在处理器速度为每秒钟执行数千万条指令的条件下，可以毫无困难地推算出，1亿条记录构成有序子表的时间远小于I/O时间。这样，总时间约为37分钟。 □

现在，考虑如何通过归并有序子表来完成排序。像在经典的归并排序中那样，可以成对
529 地将它们归并，但是，如果有 n 个有序子表，将会导致所有的数据进出存储器$2\log_2 n$次。例如，例11.9的16个已排序子表要被读入辅存和从辅存读出一次以便归并成8个表。一次完整的读和写将有序表减少到4个，另外两次完整的读和写将把表减少到1个。这样，每一块进行8次磁盘读写。

一种较好的处理方法是将每个有序子表的第一块读入一个主存缓冲区。对于一些巨型的关系来说，来自第一趟的有序子表太多，以致即使每个子表只读取一个块，也无法将所有这些块放入主存中，这个问题将在11.4.5节处理。但是，对于诸如例11.7中的那些数据，由于表的数目相对较少，该例中是16个，使得来自每个表的一个块都可以顺利地装入主存。

也可以利用一个缓冲区存放输出块，输出块将容纳尽可能多的元素，这些元素位于完全排序表的前部。输出块开始为空。缓冲区的安排可以如图11-11所示。按如下步骤将有序子表归并成一个包含所有记录的有序表：

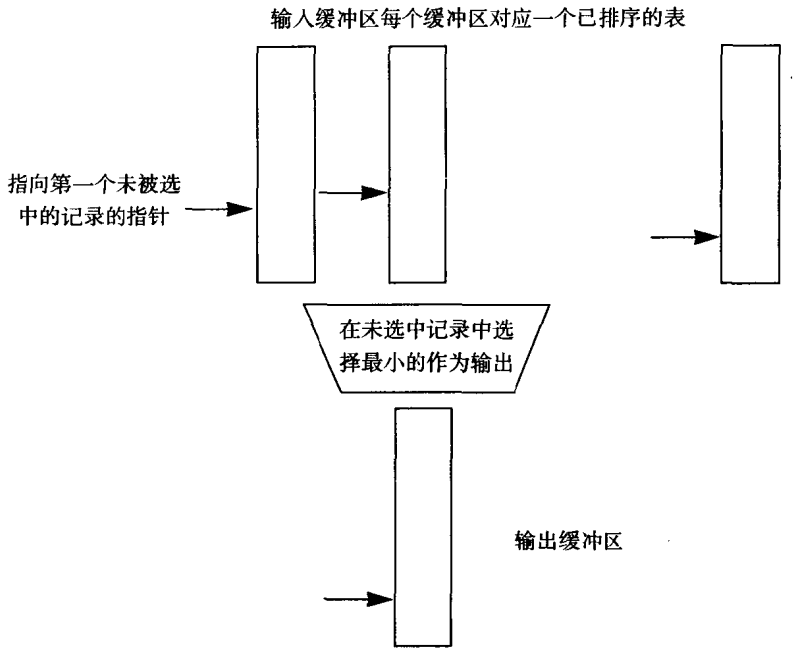


图11-11 多路归并法的主存组织

1. 在所有表的剩余元素的第一个元素中找出最小键。由于这种比较在主存中进行，是线性查找，要执行的机器指令数目正比于子表数。然而，如果愿意，可以利用一个基于“优先队列”[⊖]的方法去查找最小元素，优先队列所花费的时间与子表数的对数成正比。

2. 将最小元素移动到输出块的第一个可用位置。 530

3. 如果输出块已经填满，则将其写入磁盘，并且重新初始化主存中对应的缓冲区，以便存放下一个输出块。

4. 如果刚刚从中取出最小元素的块已经被取尽，则从同一有序子表中读取下一个块到同一缓冲区，即刚刚取尽记录的那个块所用的缓冲区。如果有序子表中没有任何块剩余，则听任缓冲区为空，而且在任何对最小剩余元素的选取中，都不再考虑来自那个表的元素。

第二趟与第一趟不同，因为不知道何时一个输入块会被取尽，所以块以一种不可预料的次序读出。然而，请注意，来自一个有序表记录的每一个块，都恰好从磁盘读出一次。这样，第二趟读取块的总次数是100 000，与第一趟一样多。同样，每个记录被放入输出块一次，而且每一个输出块均被写入磁盘。这样，第二趟写块的次数也是100 000。第二趟在主存中的计算量与I/O开销相比可以忽略，因此我们断定，第二趟又花费了37分钟，或者说整个排序花费74分钟。 531

11.4.5 更大型关系的多路归并

以上描述的两趟多路归并排序可用于一些非常大的记录集。为了看看这样的记录集有多大，假设：

1. 块大小是 B 字节。

2. 缓冲区块可用的主存是 M 字节。

3. 记录占用 R 字节。

⊖ 见Aho,A.V. and J.D. Ullman *Foundations of Computer, Science*, Computer Science Press,1992.

块应当有多大?

在使用Megatron747磁盘的算法分析中，设定了一个16KB大小的块。可是有人认为较大的块尺寸有利。在例11.5中，一个16KB块的传输时间大约是0.25毫秒，平均寻道时间和旋转延迟时间大约是10.63毫秒。如果使块大小加倍，那么对于像所描述的多路归并排序这样的算法，磁盘I/O数将减少一半。另一方面，访问一个块的时间的惟一变化是传输时间增加到0.50毫秒。这样，排序所花费的时间近似于减少一半。如果再把块大小加倍至512KB(Megatron 747 的一个磁道)，传输时间仅增加到8毫秒，在这种情况下，平均块访问时间将是20毫秒，此处仅需要访问12 500块，排序速度提高了14倍。

另一方面，保持相当小的块尺寸也有其相应理由。首先，无法有效地利用覆盖若干个磁道的那些块；其次，如果块的尺寸太大，小的关系只占据块的一小部分，这样要浪费磁盘上许多空间。还有针对二级存储器组织的某些数据结构，提出将数据分布到许多块当中，从而当块尺寸太大时，工作情况不好。实际上，在11.4.5节中将看到，块越大，能用本节所描述的两趟多路方法排序的记录越少。尽管如此，随着机器速度的加快，磁盘容量的加大，块加大是一种趋势。

这样，主存中可用的缓冲区数是M/B。在第二趟，一个缓冲区用于存放输出块，其余的缓冲区供有序子表使用。这样，第一趟可以创建的有序子表数是（M/B）-1。这个量也正是待排序记录填满主存的次数。每填满主存一次，可以排序M/R个记录。这样，可以排序的记录的总数是$(M/R)((M/B)-1)$，或者近似为M^2/RB个记录。

例11.10 如果采用例11.7中给出的参数，则M =104 857 600, B = 16 384, R = 160。这样，能够排序的记录数达到 M^2/RB =42亿，占据2/3TB。注意，这样大的关系，在Megatron 747磁盘上是装不下的。 □

如果需要排序更多的记录，可以增加第三趟，使用TPMMS为包含M^2/RB个记录的记录组排序，将它们变成有序子表。然后，在第三趟，在最终的多路归并中，归并多达（M / B）- 1个这样的子表。

第三趟大约能够排序 M^3/RB^2 个记录，占据 M^3/B^3 个块。对于例11.7的参数来说，这个总量大约是27万亿条记录，占据4.3拍（PB）。这样一个量至今未闻。由于即使对TPMMS的0.67 太字节限制，在二级存储器中都不大可能达到，因此建议，多路归并排序的两趟方式足以满足所有实际目的。

11.4.6 习题

习题11.4.1 使用两趟多路归并排序法对例11.7中的关系排序。如果用习题11.3.1中描述的Megatron 777磁盘代替Megatron 747磁盘，而其他所有机器特性和所保存的数据相同，则排序需要多长的时间?

习题11.4.2 假设在机器上采用两趟多路归并法和例11.7的关系R。如果关系R和/或机器特性作如下更改，请说明排序需要多少次磁盘I/O:

532

* a) 关系R的元组数加倍（其他任何参数仍保持不变）。

b) 元组长度加长到320字节（其他任何参数仍保持与例11.7相同）。

* c) 块的大小增加到32 768字节（同样地，完成本习题的全过程中其他所有参数不变）。

d) 可用主存储器的大小增加到200MB。

! **习题11.4.3** 假设例11.7中的关系 R 的元组数增加到与在该例中描述的机器上使用的两阶

段多路归并排序法所能排序的最大元组数相同。同时假设磁盘可以存放关系R，但是磁盘、机器和关系R的其他所有特性保持不变。对关系R排序需要花费多少时间？

* **习题11.4.4** 再次考虑例11.7的关系R，但是假设按照排序键（事实上，排序键是一般意义上的“键”，并且惟一地标识记录）的顺序存储。同时假设，关系R存储在一个位置已知的块序列中，因此对于任何一个i，可以用一个磁盘I / O来定位和检索R的第 i 块。给出一个键值 K，通过使用标准的二分法搜索技术，能够找到包含该键值的元组。请问找到包含键 K 的元组所需要的最大磁盘I/O数是多少？

!! **习题11.4.5** 假设有与习题11.4.4相同的情况，但是有10个键值要查找。请问找出全部10个元组所需的最大磁盘I/O数是多少？

* **习题11.4.6** 假设有一个关系，它有n个元组，每个元组有 R 字节。另外有一台机器，其主存为M，磁盘块大小为B，使用两趟多路归并排序法，它恰好能对 n 个元组排序。如果对参数按下列情况之一作改变：a) B加倍 b) R加倍 c) M加倍，n 的最大值的变化是多少？

! **习题11.4.7** 如果能够利用三趟多路排序法排序，重复计算习题11.4.6。

*! **习题11.4.8** 作为参数R、M 和B（同习题11.4.6）以及整数 k 的函数，利用 k 趟多路归并排序法，可对多少个记录排序？

11.5 加速二级存储的访问

11.4.4节的分析是假定数据存储在一个单独的磁盘上，并且从磁盘的可能位置上随机地选择块。这个假设对于并发执行大量小查询的系统是合理的。但是，如果系统所做的一切就是排序一个大的关系，那么通过明智地安置排序中所涉及的块的位置，充分利用磁盘的工作方式，就能够节省大量的时间。事实上，即便系统负载是来自访问磁盘上“随机”块的大量无关查询，也可以做一些工作使查询执行得更快，并且/或者允许系统同时执行更多的查询（“提高吞吐量”）。本节将要讨论的策略有： 533

- 将一起访问的块放置在同一柱面上，这样，通常可以避免寻道时间，并且有可能避免旋转延迟。
- 将数据划分到几个较小的磁盘上，而不是集中在一个大磁盘上。使用更多的可独立访问块的磁头组合，从而可以增加单位时间内访问块的数量。
- “镜像”磁盘：在单个磁盘上制作两份或更多的数据备份。这种策略，除了当其中一个磁盘出现故障时能保护数据外，还能像划分数据到几个磁盘上一样，增加可以同时访问的块数。
- 无论是在操作系统中，或是在DBMS中，以及在磁盘控制器中，使用磁盘调度算法来选择读写请求的块的顺序。
- 将预期要使用的块预取先取到主存储器中。

在讨论中，将强调改进的可能性，这些改进只是当系统用在（至少是暂时地）一项特殊任务（例如在例11.7中介绍的排序操作）时才是可能的。但是，至少还可以从其他两个方面来测试系统性能和二级存储器的使用：

当系统同时支持大量的进程时会发生什么情况？例如，一个航班订票系统将同时接收到来自许多代理商的关于班机和订票情况的查询。

如果对一个计算机系统已有一个固定的预算，或者必须在一个已经设计好且不易改动的系统上执行一系列查询时，该如何执行？

在研究可能的选择之后，将在11.5.6节中讨论这些问题。

11.5.1 按柱面组织数据

534 由于寻道时间占平均块访问时间的一半，因此在某些应用中，将一些可能被一起访问的数据（例如关系）存储在单个柱面上是一件有意义的事情。如果空间不够，可以利用相邻的几个柱面。

事实上，如果选择在单个磁道上或者在单个柱面上连续地读所有块，那么可以只考虑第一次寻道时间（移动到柱面上的时间）和第一次旋转延迟（等待第一个块移动到磁头下的时间），而忽略其他时间。这样，从磁盘上读写数据的速度接近于理论上的传输速率。

例11.11 回顾11.4.4节中描述的两趟多路归并排序法的性能。回忆一下在例11.5中，对于Megatron 747磁盘，我们确定了它的块平均传输时间、寻道时间和旋转延迟时间分别为0.25毫秒、6.46毫秒和4.17毫秒。同时还发现，一个占据1GB、有10 000 000条记录的排序操作，大约需要花费74分钟。在这段时间内需要执行四个操作，其中一个读操作，一个写操作。两趟算法的每趟都涉及一个读操作和一个写操作。

现在考虑按柱面组织数据是否可以缩短这些操作的时间。第一个操作是将原始数据读到主存储器中。回忆例11.9中，需要装载主存储器16次，每次6400块。

原始的100 000块数据可以被存储在连续的柱面上。Megatron 747磁盘的16 384个柱面中的每一个大约可存储8MB，共512块。从技术角度上看这是一个平均数，因为内侧磁道的存储量比外侧磁道小，但是为简单起见，可以假设所有磁道和柱面有同样的存储量。这样，就必须要在196个柱面上存储原始数据，并且每次将13个柱面读入主存储器。

可以利用单个寻道时间读取一个柱面。甚至不必等待柱面上的任何特殊块通过磁头，因为所读记录的顺序在这一阶段并不重要。必须移动12次磁头到相邻的柱面，但是回忆一下，根据例11.5中的参数，移动一个磁道只需要花费1毫秒。写主存储器的总时间是：

1. 一次平均寻道时间为6.46毫秒。
2. 12次单柱面的寻道时间为12毫秒。
3. 6400块的传输时间为1.60秒。

除了上述最后一项时间之外，其他各项都可以忽略不计。因为往主存储器中装载16次，所以第一趟读操作的总时间大约是26秒。当假设块是随机地分布在磁盘上时，这个数字应当与例11.9中第一趟中的读操作所花费的18分钟相比较。类似地，第一趟中的写操作也可以在邻近的柱面上存储记录的16个有序表。采用与读操作同样的磁头运动可以将这些记录写到另外
535 196个柱面上。对于16个表的每一个进行一次随机寻道和12次单柱面寻道。这样，第一趟的写操作时间大约是26秒，或者第一趟的全部操作时间为52秒，相比之下，当采用随机分布块时为37分钟。

另一方面，按柱面存储对排序的第二趟没有帮助。在第二趟中，块以一定的顺序从16个有序子表的头部读出，该顺序由数据和取尽其当前块的表所决定。同样，包含完整有序表的输出块一次写入一块，与块的读操作交替进行。这样，第二趟仍然要花费37分钟。这里排序时间几乎减掉一半，但是只去审慎地使用柱面并不能做得更好。 □

11.5.2 使用多个磁盘

如果用多个磁盘（每个磁盘都具独立磁头组）来替换一个磁盘（其多个磁头被锁定在一起），常常能够提高系统的速度。图11-6图示了这样的安排。该图中，是将三个磁盘连接到同一个控制器。只要磁盘控制器、总线和主存储器能以高速率处理数据传输，其效果近似于将读写磁盘

的所有时间被磁盘数除。说明这个差别的例子如下：

例11.12 Megatron 737磁盘具有例11.3和例11.5中的Megatron 747磁盘的所有特性，不过它只有两个盘片和四个盘面。这样，Megatron 737可保存32GB。假设用4个Megatron 737代替一个Megatron 747，考虑两趟多路归并排序法如何实施。

首先，可以把所给定的记录分配到4个磁盘上，这些数据将占据各个磁盘上196个相邻的柱面。当在第一趟期间准备从磁盘向主存装载数据时，将从4个磁盘的每一个中读取数据并填入1/4主存。此时仍能得到例11.11中获得的好处：寻道时间和旋转等待基本上趋于零。但是这样能够在400毫秒内从一个磁盘读取1600个块，并装入1/4主存。只要系统能处理来自4个磁盘的以此速率传输的数据，便能在0.4秒内将100MB装入主存。与此相比，使用一个磁盘时要用1.6秒。

同样，当在第一趟从主存写入磁盘时，可以将每个有序表分配到4个磁盘上，每个磁盘大约占据13个相邻的柱面。这样，第一趟的写操作也加速了4倍，整个阶段大约花费13秒。与此相比，仅用11.5.1节基于柱面的改进措施需要52秒，原始随机方法则需要37分钟。

现在，考虑TPMMS的第二趟。此时仍然必须以看似随机的且与数据相关的方式从各个表的前面读块。如果第二趟的核心算法—来自16个子表的剩余元素中的最小元素选择—需要将用 536
块表示的16个表完全装入主存，那么就不能从使用4个磁盘中得到好处。每次一个块被用完时，就必须等待，直到来自同一个表的一个新块被读出以取代原来的块。这样，在同一时刻只有一个磁盘可以使用。

然而，如果更仔细地编写代码，新块的第一个元素一出现在主存，就可以在16个最小元素之间重新开始比较[⊖]。如果这样，若干个表就可以同时将它们所在的块装入主存。只要它们是在不同的磁盘上，就能同时执行若干个块的读操作，而且第二趟读操作的速度有增加4倍的潜力。另外还必须遵循按随机顺序读块的限制。如果紧接着需要的两个块恰巧是在同一个磁盘上，那么其中的一个块就得等待另一个，至少在第二个块的开始部分到达主存之前，所有主存处理都被中止。

要提升第二趟写操作的速度比较容易。可以采用4个输出缓冲区，并依次填充各个缓冲区。每个缓冲区一旦装满，就写入一个特定的磁盘，按顺序装入柱面。这样可以在其他三个缓冲区写入磁盘的同时，填写另一个缓冲区。

尽管如此，把整个有序表写入磁盘的速度还是不可能比从16个中级表读数据的速度更快。正如前面所看到的，要保持全部4个磁盘在所有时间都做有用的工作是不可能的，第二趟的增速可能是在2～3倍的范围内。然而，即使是2倍也可以节省18分钟。通过按柱面组织数据并在4个磁盘保存数据，对于排序示例，可以将排序时间从两趟每趟37分钟，减少到第一趟13分钟和第二趟18分钟。 □

11.5.3 磁盘镜像

有时为两个或更多的磁盘保留同样的数据拷贝很有意义。这些磁盘被称做相互镜像(mirror)。磁盘镜像的一个重要动机是，无论哪一个磁盘头文件损坏都可使数据幸免于难，因为已损磁盘镜像上的数据仍然可读。为增强可靠性而设计的系统常使用磁盘对互为镜像。

然而，镜像磁盘也能提高访问数据的速度。回忆例11.12中第二趟的讨论。讨论中可见，如果非常仔细地安排时序，最多时可装入4个块。这些块来自4个不同的有序表，而且表的先前

⊖ 应当强调，这个方法要求极端准确地执行，而且只有在这样做有重大好处时才予以尝试。如果不小心，就会有极大的风险，那就是在一个记录实际达到主存之前就试图读这个记录。

块已经处理完毕。但是，装入时不能确定选择哪四个表会得到新块。这样，就可能不走运地发现头两个表是在同一个磁盘上，或者头三个表中的两个是在同一个磁盘上。

537 如果人们愿意购买和使用单个大磁盘的4个拷贝，那么就能保证系统总是能一次检索4个块。也就是说，不管需要哪四个块，都能把每个块分配到4个磁盘中的任何一个，并且从该磁盘读出那个块。

通常，如果做一个磁盘的n个拷贝，就能并行地对任何n个块进行读取。而且，如果一次要读的块数小于n，那么通过正确地选择从哪个磁盘读，就常常能获得速度提升。也就是说，可以选择其磁头最靠近要读的那个柱面的可用磁盘。

与使用单个磁盘相比，使用镜像磁盘既不提高写操作的速度，也不降低写操作的速度。也就是说，无论何时需要写一个块，都要在有拷贝的所有磁盘上写这个块。但由于写操作可以并行地执行，因此所花费的时间与单个磁盘大致相同。对于不同的镜像，磁盘的写操作时间可能有微小的差别，因为不能指望镜像磁盘会完全同步旋转。这样，当一个磁盘的磁头刚刚错过一个块时，另一个磁盘的磁头可能正准备越过相同块的相同位置。然而，这些旋转延迟上的差异最终会达到平均值，而且如果使用11.5.1节的基于柱面的策略，旋转延迟总是可以忽略。

11.5.4 磁盘调度和电梯算法

在某些情况下，另一个提高磁盘访问速度的有效方法是让磁盘控制器在若干个请求中选择谁先被执行。当系统需要按一定的顺序读或写磁盘块时，例如在例举的运行归并排序法的某些情况下，这个方法无法产生效用。可是，当系统支持只访问少量块的多个小进程时，通过选择首先要处理的进程请求，常常可提高吞吐量。

电梯算法（elevator algorithm）是调度大量块请求的一个简单有效的方法。此方法把磁头的移动看做是在做跨越磁盘的扫描，从柱面最内圈到最外圈，然后再返回来，就好像电梯在做垂直扫描，从建筑物的底层到顶层，然后再折返。当磁头通过柱面时，如果有一个或多个该柱面上块的请求，磁头就停下来。根据请求，所有这些块被读或写。然后磁头沿着其正在行进的同一方向继续移动，直至遇到下一个包含要访问块的柱面。当磁头到达其行进方向上的某一个位置，而该位置的前方不再有访问请求时，磁头就朝相反方向移动。

例11.13 假设现在正在调度一个Megatron 747磁盘，该磁盘的平均寻道时间、旋转延迟和传输时间分别为6.46、4.17和0.25（本例中，所有时间均以毫秒计算）。假设某一时刻存在着对柱面2000、6000和14 000的块访问请求。磁头目前位于柱面2000。此外，在晚些时候还有3个对块的访问请求。所有请求如图11-12所示。例如，从图中可见对柱面4000的块访问请求是在10毫秒时产生。

538

请求的柱面	到达时间
2000	0
6000	0
14000	0
4000	10
16000	20
10000	30

图11-12 6次块访问请求的到达时间

假定每个块访问需要0.25毫秒传输时间和4.17毫秒平均旋转延迟时间，即无论寻道时间是多少，都需要为每一次块访问加上4.42毫秒。寻道时间可通过例11.5给出的Megatron 747的规则计算：1加上磁道数被1000除。现在来看如果使用电梯算法调度会发生什么情况。对柱面2000的第一个请求不需要寻道，因为磁头已经定位在那里。这样，在4.42时刻完成第一次访问。对柱面4000的请求这时尚未到达，所以移动磁头到柱面6000，即磁道的最大数字方向扫描过程中的下一“站”。从柱面2000到6000花费的寻道时间是5毫秒，所以在9.42时刻到达，并在另一个4.42毫秒内完成访问。这样，第二次访问在13.84时刻完成。在这个时间之前，对柱面4000的请求已经到达，但是由于在7.42时刻已经过那个柱面，所以在下一次经过之前不会回到那个位置。

这样，磁头下次移动到柱面14 000，花费了9毫秒用于寻道，4.42毫秒用于旋转和传输。于是，第三次访问在27.26时刻完成。现在，对柱面16 000的请求已经到达，所以磁头继续向外圈行进。这时需要3毫秒的寻道时间，所以本次访问完成时间是在27.26+3+4.42=34.68时刻。

在这个时刻，对柱面10 000的请求已经产生，所以现在还剩有本次请求和柱面4000的请求。磁头将向内圈行进，处理这两次请求。图11-13总结了各个请求处理的时间。

现在用“先到先服务”这样更朴素的方法来比较电梯算法的性能。假设前三个请求的顺序是2000、6000和14 000，那么，这前三个请求的处理方式完全相同。然而，这时要回到柱面4000去，因为它是第四个到达的柱面请求。该请求的寻道时间是11毫秒，因为要从柱面14 000行进到4000，行程超过了磁盘的一半。第五次是对柱面16 000的请求，需要13毫秒的寻道时间。 539
最后一次是对柱面10 000的请求，其寻道时间是7毫秒。图11-14概括了用“先到先服务”调度法所产生的活动。两种算法相差14毫秒，看起来似乎不是很显著，但是请注意，在这个简单的例子中，请求数较少，而且假设在到达的六个请求中前四个请求的处理两种算法没有区别。□

请求的柱面	完成时间
2000	4.42
6000	13.84
14000	27.26
16000	34.68
10000	46.10
4000	57.52

图11-13 采用电梯算法的块访问完成时间

请求的柱面	完成时间
2000	4.42
6000	13.84
14000	27.26
4000	42.68
16000	60.10
10000	71.52

图11-14 采用先到先服务算法的块访问完成时间

如果等候磁盘的平均请求数增加，电梯算法会进一步提高吞吐量。例如，如果等候请求池等于柱面数，那么每次寻道将只跨越少量柱面，而平均寻道时间将接近于最小。如果请求池增长到超过柱面数，那么在每个柱面通常会有一个以上的请求。磁盘控制器可以围绕柱面来对请求排序，以减少平均旋转延迟以及平均寻道时间。可是，如果请求池增长到那么多，服务任何请求所花费的时间就会变得非常非常大。下面的例子将解释这种情况。

例11.14 假设再一次操作Megatron 747磁盘，它有16 384个柱面。想像有1000个磁盘访问 540
请求在等待。为简单起见，假定它们都是不同柱面上的块，每16个柱面一块。如果在磁盘的一端开始横跨磁盘扫描，1000个请求中的每一个请求的寻道时间为1毫秒多一点儿，旋转延迟时间是4.17毫秒，传输时间是0.25毫秒。这样每5.42毫秒处理一个请求，大约是随机块查询平均时间10.88毫秒的的一半。可是，全部1000次访问需要5.42秒，所以处理一个请求的平均延时是该值的一半，即2.71秒，这是一个相当显著的延时。

现在，假设请求池的大小是32 768，为简单起见，假设每个柱面恰好有两次访问。这种情况下，每次寻道时间是1毫秒(每次访问0.5毫秒)，当然传输时间是0.25毫秒。由于每个柱面上有两个块访问，平均来说，当磁头到达那个磁道时，磁头与两个块中较远的那个块的距离将是绕磁盘一周路程的2/3。这个估算的证明错综复杂，本文在“等待两个块中的后一个块”框中作了解释。

这样，两个块的平均延迟将是旋转一周时间的2/3的一半，即 $\frac{1}{2}\times\frac{2}{3}\times 8.33 = 2.78$ 毫秒。这样就把访问一个块的平均时间减少到0.5+0.25+2.78=3.53毫秒，大约是先到先服务调度算法平均时间的1/3。另一方面，32 768次访问花费的总时间为116秒，所以处理一次请求的平均延时大约是1分钟。 □

等待两个块中的后一个块

假设有两个块分布在一个柱面上的随机位置。设这两个位置为x_1和x_2，以圆周的分数标识，所以它们的取值范围是0～1。x_1和x_2都小于0～1之间某个y值的概率为y^2。这样，y的概率密度就是y^2的导数$2y$。也就是说，y取某个给定值的概率随着y从0增大到1而线性增大。y的平均值就是y乘以y的概率密度的定积分，即$\int_0^1 2y^2 = 2/3$。

11.5.5 预取和大规模缓冲

加速某些第二级存储器算法的最后一个建议称为预取（prefetching），有时也称做双缓冲（double buffering）。在某些应用中，能够预测从磁盘请求块的顺序。如果是这样的话，就能在需要这些块之前将它们装入主存。这样做的好处是能够较好地调度磁盘，通过采用诸如电梯算法等，减少访问块所需要的平均时间。例如，这样做能够在加速例11.14提出的块访问同时避
541 免该例处理请求所带来的很长的延时。

例11.15 作为采用双缓冲的一个例子，再次考虑TPMMS的第二趟。该算法通过从每个表取出一个块带入主存，归并了16个有序子表。如果有很多有序子表要归并，以至于从每个子表取出一个块就会填满主存，那么算法就不可能再有改进了。但是在例子中，还有很多主存没有使用。例如，在归并期间，可以为每个表分配两个块缓冲区，当从一个缓冲区选取记录归并的同时，可以把磁盘块写入另一个缓冲区。当一个缓冲区被取尽时，就切换到相同表的另一个缓冲区，而且没有任何延时。 □

然而，例11.15的方式仍要花费时间读有序子表的全部100 000块。如果下列事实成立，就能够把11.5.1节的基于柱面策略与预取结合起来：

1. 在整个连续柱面上存储有序子表，每个磁道上的块都是有序子表的连续块。
2. 每当需要一个给定表中的更多记录时，读整个磁道或整个的柱面。

例11.16 为了体会按磁道大小或按柱面大小读的优越性，再次考虑TPMMS的第二趟。主存储器为16个表中的每一个设置两个磁道大小的缓冲区。已知Megatron 747一个磁道的大小是0.5MB，所以需要的主存总空间大约是16MB。因为可以从任何一个扇区开始读一个磁道，所以读一个磁道的时间大体是平均寻道时间加上磁盘转动一周的时间，即6.46+8.33=14.79毫秒。由于要读16个有序子表，由于需要读196个柱面上的所有块，即3136个磁道，所以读全部数据的总时间大约是46秒。

如果每个有序子表拥有两个柱面大小的缓冲区，在一个缓冲区正在使用的同时填入另一个缓冲区，那么就能够做得更好。由于Megatron 747的一个柱面有16个磁道，因此使用32个4MB的缓冲区，即128MB。虽然在例中说主存中只有100MB可用于排序，但是128MB也是合理的。

采用柱面大小的缓冲区，每个柱面只需寻道一次。这样，寻道时间加上读一个柱面的所有16个磁道的时间是$6.46+16\times 8.33=140$毫秒。读全部196个柱面的时间是196乘以上述时间，即大约27秒。 □

写的思路与刚刚讨论过的读的思路类似。按照预取的思路，只要不需要立即再次使用缓冲区，就可以延迟该缓冲块的写。这个策略允许避免等候块被写入磁盘时的延时。

然而，更为强有力的策略是采用大的输出缓冲区（磁道大小或柱面大小）。如果应用允许
542 写入这样大的块，那么基本上能消除寻道时间和旋转延迟，并以磁盘的最大传输速率写盘。例如，如果修改了排序算法第二趟的写操作部分，具有两个容量各为4MB的输出缓冲区，那么就

能用有序记录填充一个缓冲区，并且在用下一个有序记录填充另一个缓冲区的同时，将填入第一个缓冲区的记录写入一个柱面。那么，像例11.16中的读时间一样，写时间将是27秒，这样第二趟总共花费将少于1分钟，刚好与例11.11改进的第一趟一样。大体说来，将按柱面组织数据、柱面大小的缓冲以及预取方法相结合，可以在少于2分钟内完成一个排序，而通过朴素的磁盘管理策略完成这个排序则需要花费74分钟。

11.5.6 对策略和折中的小结

本节讨论了能改进磁盘系统性能的5种不同的“手段”。它们是：

1. 按柱面组织数据。
2. 用若干个磁盘代替一个磁盘。
3. 镜像磁盘。
4. 通过电梯算法调度请求。
5. 按磁道大小或柱面大小预取数据。

另外还考虑了它们在两种情况下的效果，这两种情况代表了磁盘访问请求的两个极端：

a) 一种相当常规的情况，以两趟多路归并排序法（TPMMS）第一趟为例，在这种情况下，块可以按事先预测的次序读和写,并且只有一个使用磁盘的进程。

b) 像航班订票或银行账目改变那样的短进程的集合，它们并行地执行，共享同一磁盘，并且不能事先预测。两趟多路归并排序法的第二趟就具有这样一些特性。

下面总结前面提到的每一种方法对于这两种不同应用类型，以及介于这两者之间的应用类型的优点和缺点：

基于柱面的数据组织

- 优点：对于类型（a）的应用效果极佳。这里访问可事先预测，并且仅有一个进程正在使用磁盘。
- 缺点：对类型（b）的应用没有任何帮助。这里访问不可预测。 543

多磁盘

- 优点：对于这类应用，都能增加处理读/写请求的速率。
- 问题：对同一磁盘的多个读或写请求不能同时满足，所以加速系数可能小于磁盘数增加的系数。
- 缺点：若干小磁盘的费用超过具有相同总容量的单个磁盘的费用。

镜像

- 优点：对这两种类型的应用，均能增加处理读/写请求的速率，而且不存在多磁盘中遇到的访问冲突的问题。
- 优点：改善了所有应用的容错性。
- 缺点：必须付出两个或多个磁盘的代价，但只能得到一个磁盘的存储容量。

电梯算法

- 优点：当对块的访问不可预测时，减少读/写块的平均时间。
- 问题：在有许多磁盘访问请求等候的情况下，这种算法最有效，但也因此使得这些请求处理的平均延时较高。

预取/双缓冲

- 优点：当需要访问的块已知但请求的时间与数据相关时，可加速访问，如同TPMMS的第二趟。

• 缺点：需要额外的主存缓冲区。当访问是随机时没有任何帮助。

11.5.7 习题

习题11.5.1 假设为Megatron 747磁盘调度I/O请求，磁头的初始位置在磁道8000，访问请求如图11-15所示。问在下列两种情况下，每一种请求何时可以完全得到服务？

请求的柱面	请求到达时间
2000	0
12000	1
1000	10
10000	20

图11-15 6个块访问请求的到达时间

544

a) 采用电梯算法（开始移动任意方向）。

b) 采用先到先服务调度。

*! **习题11.5.2** 假设使用两台Megatron 747磁盘互相作为镜像。同时，采用相关技术使第一个磁盘的磁头位于柱面靠内的一半，第二个磁盘的磁头位于柱面靠外的一半，而不再允许从任一个磁盘读任何的块。假设读请求是针对随机的磁道，并且始终不必去写，那么问：

a) 系统读块的平均速率是多少？

b) 这个速率与无任何约束的镜像Megatron 747磁盘的平均速率相比如何？

c) 你认为该系统的缺点是什么？

! **习题11.5.3** 讨论请求的到达速率、电梯算法的吞吐量和请求的平均延时之间的关系。为使问题简化，作如下假设：

即使是没有对最两端柱面的请求，电梯算法的扫描总是从最内圈到最外圈，或者相反。

当扫描启动时，只有那些已经在等待的请求会被处理，而扫描正在前进时到达的请求不会被处理，即使磁头通过它们的柱面也如此[⊖]。

在一次扫描中决不会有同一柱面上的两个块的请求在等候。

令A是交互到达率，即块访问请求之间的时间间隔。假设系统处于稳定状态，即它已经进行了长时间的请求接收和回答。对于Megatron 747磁盘，计算下列值（表示为A的函数）：

545

* a) 执行一次扫描所花费的平均时间。

b) 在一次扫描中得到服务的请求数。

c) 一次请求等候服务的平均时间。

*!! **习题11.5.4** 从例11.12可以看到，怎样通过把要排序的数据分散到4个磁盘上，使得能够同时读取一个以上的块的方法。假设：在归并阶段产生对随机磁盘块的读请求，并且所有读请求都会被处理，除非它们要访问的磁盘目前正在为另外一个请求服务。问任何时刻正在为请求服务的磁盘平均数是多少？注意：有两个使问题简化的提示：

1. 一旦一个读块请求不能继续，那么归并必须停止，并且不再产生请求，因为产生这个未被满足的读块请求的子表已经没有任何数据。
2. 只要归并能继续进行，它就将产生一个读请求，因为与满足一个读请求的时间相比，主存归并基本上不花费时间。

! **习题11.5.5** 如果要从一个柱面上读k个随机选定的块，那么在经过所有的块之前，必须绕着柱面行走的平均距离有多远？

⊖ 这个假设的目的在于避免不得不涉及的这样一个事实：电梯算法的一次典型扫描开始进行得很快，因为刚刚经过的这些区域很少有等待着的请求；当磁头移入最近未曾经过的一个磁盘区域时，扫描速度提升。就其本身而言，对请求密度在扫描行进过程中变化的分析是一项有趣的实践。

11.6 磁盘故障

在本节和下一节，考虑磁盘可能发生故障的方式，以及采取什么措施来减轻这些故障。

1. 最常见的一种故障形式是*间断性故障*（intermittent failure），读或写一个扇区的某次尝试不成功，但是经过反复尝试，又能成功地读或写。

2. 更严重的一种故障形式是，一个或多个二进制位永久地损坏，不管尝试多少次都不能正确地读一个扇区，这种错误形式称为*介质损坏*。

3. 一种相关的错误类型是*写故障*（write failure）。在这种故障中，当试图写一个扇区时，既不能正确地写，也不能检索先前写入的扇区。该故障的一种可能的原因是在写扇区的过程中发生了供电中断。

4. 磁盘故障的最严重形式是*磁盘崩溃*，在这种故障中，整个磁盘突然变为永久不可读。 546

本节讨论一种简单的磁盘故障模型。奇偶校验将作为检测间断性故障的方法。另外，还将讨论“稳定的存储”这样一种组织磁盘的技术，这种技术使介质损坏或写故障不会导致数据永久性损失。在11.7节中讨论一般被称为“RAID”的解决磁盘崩溃问题的技术。

11.6.1 间断性故障

磁盘扇区通常伴随着一些冗余位一起存储，对此将在11.6.2节讨论。这些位的目的是能够识别正在从扇区读出的内容是否正确。通过它们同样能够判断写过的扇区是否已被正确地写入。

磁盘读的一种有用模式是读函数返回一对值（w,s），这里w代表被读扇区中的数据，s代表一个表示读是否成功的状态位，即能否相信w是该扇区的真实内容。在间断性故障中，可以多次得到“坏”状态，但是如果读函数被重复足够多次（100次是一般的限制），那么最终会返回一个状态“好”，而且还可相信随着这个状态一起返回的数据是磁盘扇区的正确内容。在11.6.2节中将看到，有一种被欺骗的可能：即虽然状态是“好”，但是返回的数据错误。然而，如果增加扇区的冗余度，就可以按照所希望的降低出现这种现象的概率。

扇区的写也能通过观察所写内容的状态获益。在11.3.5节中将会看到，可以在写扇区之后试着读各个扇区，并且确定写是否成功。执行检查的一个直截了当的方法是读这个扇区，并且与打算写的扇区进行比较。然而，与在磁盘控制器中进行完全比较相比，更简单的方法是读这个扇区然后看看其状态是否为“好”。如果是“好”状态，就假定写正确，而如果状态是“坏”，那么显然写不成功，并且必须重写。注意，正如读的情况一样，如果状态是“好”，但是实际上写不成功，那么则是被欺骗了。另外也正如读的情况一样，能够使出现这种情况的概率降低到希望的水平。

11.6.2 校验和

一个读操作如何能决定一个扇区的好/坏状态，乍看起来不可思议。然而，用于当前磁盘驱动器中的这项技术相当简单：每个扇区有若干个附加位，称为*校验和*（checksum），附加位的设置取决于存储在那个扇区的数据位的值。在读出时，如果发现校验和对该数据位不合适，那么就返回状态“坏”，否则就返回状态“好”。尽管除了不正确的位碰巧与正确位有相同的校验码（因此不正确的位将被赋予状态“好”）之外，还存在着数据位被错读的一种较小的可能 547
性。但是，通过使用足够多的校验和位，就能够将这种概率减少到所希望的低水平。

校验和的一种简单形式是基于扇区内所有位的*奇偶性*（parity）。如果在二进制的集合中有奇数个1，就称数据有*奇数奇偶性*，或者说它们的奇偶位是1。同样，如果在二进制位的集合中有偶数个1，就称数据位有*偶数奇偶性*，或者说它们的奇偶位是0。从而：

• 在二进制位的集合与它们的奇偶位中，1的个数总是偶数。

当写一个扇区时，磁盘控制器能计算出奇偶位，并将它附加到扇区中的二进制位的序列中。这样，每个扇区将有偶数奇偶性。

例11.17 如果扇区中二进制位序列是01101000，那么就有奇数个1，所以奇偶位是1。如果这个序列后面加上它的奇偶位，便有011010001。如果所给的数据位序列是11101110，有偶数个1，而奇偶位为0。序列后面加上它的奇偶位便是111011100。注意，加上一位奇偶位后构成的每一个9位序列有偶数奇偶性。 □

在读或写数据位及其奇偶位的过程中，任何一个位错误都会导致具有奇数奇偶性的位序列产生，也就是说序列中1的个数是奇数。计算1的个数，如果一个扇区有奇数奇偶性就判定存在一个错误，这对于磁盘控制器来说很容易。

当然，扇区可能有一个以上的位出错。如果这样，二进制位1的个数为偶数的概率是50%，此时检测不到错误。如果保持若干个奇偶校验位，就能增加检测出大量错误的机会。例如，保留8位奇偶校验位，其中一位用于检测每个字节的第一位，另一位用于检测每个字节的第二位，等等，直到奇偶位的第八位检测每个字节的最后一位。那么，关于大规模的错误，检测出任何一个奇偶位错误的概率是50%，8位都检测不出错误的机会仅仅是$1/2^8$，即1/256。一般地说，如果用n个独立位作为校验和，漏掉一个错误的机会仅为$1/2^n$。例如，用4字节作为校验和，那么大约在40亿次中仅有一次错误不能被检测出来。

11.6.3 稳定存储

尽管校验和的确几乎能正确地检测出介质故障或读写故障的存在，但是它不能帮助纠正错误。此外，写操作时还可能陷入一种困境：覆盖了一个扇区先前的内容，但是又不能读出新的
548 内容。这种现象在某种情况下很严重，例如，当正准备将一个小的增额加到账目余额中去时发生此类故障，原始余额和新的余额均已丢失。如果能确定扇区的内容是新的余额还是旧的余额，那么只需要判定写操作是否成功即可。

为了处理上述问题，可以在一个磁盘或多个磁盘上执行一个称为*稳定存储*（stable storage）的策略。总的思想是，扇区是成对的，每一对代表一个扇区内容 X。把代表 X 的扇区对分别称做“左”拷贝X_L和“右”拷贝X_R。进一步再假定这两个拷贝用足够多的奇偶校验位来写，以便考虑奇偶校验时，能排除看上去像是好扇区而实际上是坏扇区这种现象的可能性。于是假定，如果读函数对 X_L或 X_R中的任何一个返回（w，好），那么w是X的真值。稳定存储的写策略是：

1. 写 X 的值到 X_L。检查返回值的状态是否为“好”，即在写入拷贝中奇偶校验位正确。如果不是“好”，则反复写。如果在若干次写尝试之后，仍没有成功地将 X 写入X_L，则认为该扇区中有一个介质故障，必须采用以备用扇区代替 X_L一类的补救措施。

2. 对X_R重复步骤1。

稳定存储的读策略是：

1. 读X_L以获得X的值。如果返回状态是“坏”，则反复读若干次。如果最终返回了带有状态“好”的值，则取这个值为X。

2. 如果不能读X_L，则用X_R重复步骤1。

11.6.4 稳定存储的错误处理能力

11.6.3节描述的策略能够校正若干不同种类的错误。对此概述如下：

1. *介质故障*。在将X存入扇区X_L和X_R后，如果两者中有一个出现介质故障并且变为永久不可读，总是能从另一个扇区读取X。如果X_R有故障而X_L没有故障，那么读策略根本不考虑X_R就

能正确地读X_L，X_R故障将在下一次试图往X中写新值时被发现。如果仅仅是X_L有故障，那么在任何一次读X_L的尝试中都不能得到 X 的“好”状态。（前节中曾假设一个坏扇区将总是返回“坏”状态，即使实际上有一个返回“好”的极小机会时也如此，这是由于所有奇偶校验位碰巧都匹配造成的。）这样，就将进而执行读算法的步骤(2)，并正确地从 X_R 读取 X。注意，如果 X_L 和 X_R 两者都有故障，那么就不能读取 X，但是两者都发生故障的概率非常小。 549

2. 写故障。假设当写 X 的时候，有一个系统故障，例如电源断电，那么 X 在主存中可能丢失，同时写入的 X 的拷贝也将被篡改。例如，半个扇区可能已写入了 X 的部分新值，同时另一半扇区仍保留着原来的值。当系统变为可用时，通过测试 X_L 和 X_R 能够确定 X 的旧值或者新值。可能的情况有以下几种：

(a) 故障在写 X_L 的时候发生。那么将发现 X_L 的状态是“坏”。但是，由于还没有写X_R，它的状态将是“好”（除非在 X_R 中同时发生了介质故障，通常这种概率极小的情况排除在外）。由此就能够获得X的旧值。另外也可以将 X_R 拷贝到 X_L，修复 X_L 的故障。

(b) 故障在写 X_L 之后发生。那么，可以预计 X_L 将有“好”状态，并且可以从 X_L 读取X的新值。注意，X_R 可能有“坏”状态，如果那样的话，应该将 X_L 拷贝到 X_R。

11.6.5 习题

习题11.6.1 计算下列位序列的奇偶位：

* a) 00111011

b) 00000000

c) 10101101

习题11.6.2 如果在串末附加一个位作为该串各奇数位置的奇偶位，附加另一个位作为该串各偶数位置的奇偶位，就有了与一个串相关的两个奇偶位。按这种方法找出习题11.6.1中的每一对奇偶位。

11.7 从磁盘崩溃中恢复

本节将考虑最严重的磁盘故障——“磁头损坏”，其中数据被永久性地破坏。在这个事件中，如果数据没有备份到另一介质中，例如在11.5.3节中讨论的磁带备份系统，或者一个镜像磁盘，那么就根本不可能做任何事情来恢复数据。这对于重要的DBMS应用如银行和其他金融上的应用、航空公司或者其他预订数据库、库存管理系统以及其他许多应用来说是一场灾难。

已经有许多成功的开发方案用于减少因磁盘崩溃造成数据丢失带来的风险。它们通常涉及到冗余技术，这种技术扩展了11.6.2节中讨论的奇偶校验思想以及11.6.3节中讨论的复制扇区的 550
思想等。这类策略的一般术语是RAID（Redundant Arrays of Independent Disks，冗余独立磁盘阵列）[⊖]。这里，将初步讨论称为RAID 4级、5级和6级的三种方案。这些RAID方案也用于处理11.6节中讨论的故障模型：介质故障和由临时系统故障引起的单扇区数据丢失。

11.7.1 磁盘的故障模型

为了开始关于磁盘崩溃的讨论，首先需要考虑故障的统计数据。描述故障状态的最简单方法是使用平均故障时间（mean time to failure）参数。这个数据是时间长度，在这个时间内，一组磁盘的50% 将发生灾难性故障，也就是说，有磁头损坏，所以它们就不再是可读的。对于

⊖ 以前，缩写词RAID被解释为“Redundant Array of Inexpensive Disks”(冗余廉价磁盘阵列)，这个含意可能仍出现在文献中。

当今的磁盘，平均故障时间大约是10年。

使用这个数字的最简单方法是假设一年中有十分之一的磁盘发生故障。这样，磁盘生存的比率是一个指数函数。更实际情形是，磁盘生存的比率看起来更像图11-16。就大多数类型的电子设备而论，许多磁盘故障发生在生命周期的早期，这由磁盘制造中的微小缺陷造成。这些缺陷的大部分有望在出厂前发觉，但是有些缺陷并没有在数月内暴露出来。一个没有经历早期故障的磁盘有可能工作许多年。在生命周期的后期，像“磨损和破裂”这样的因素以及积聚的灰尘微粒的影响都增加了磁盘损坏的机会。

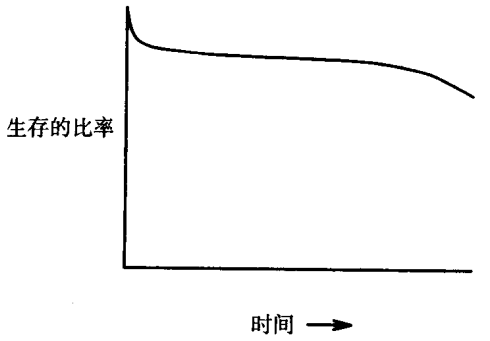


图11-16 磁盘的生存率曲线

然而，磁盘崩溃的平均时间与数据丢失的平均时间未必相同。原因是存在着若干可用的方案可以保证一个磁盘发生故障时，有另外的磁盘帮助恢复故障盘的数据。本节的剩余部分将研
551 究最常见的方案。

每一个方案初始时都有一个或多个保存数据的磁盘（这些盘被称做*数据盘*），再加上一个或多个保存信息的磁盘，这些信息完全由数据盘的内容所决定。后者被称做*冗余盘*。当一个数据盘或冗余盘发生磁盘崩溃时，可用其他的磁盘恢复故障磁盘，从而保证没有任何信息永久性丢失。

11.7.2 镜像冗余技术

正如11.5.3节讨论过的，最简单的方案是镜像各个磁盘。一个磁盘是数据盘，另一个是冗余盘。在本方案中，哪个作数据盘，哪个作冗余盘无关紧要。作为防止数据丢失的镜像，常常被称为RAID 1级。它给出一个存储器丢失的平均时间，正如下面的例子所示，这个时间比磁盘平均故障时间长得多。事实上，借助镜像和将讨论的其他冗余方案，数据丢失的惟一方式是，在修复第一个磁盘损坏的同时，第二个磁盘也损坏了。

例11.18 假设每个磁盘的平均故障时间为10年。通常认为一个磁盘出现故障的机会是每年10%。如果磁盘被镜像，当发生磁盘故障时，只需要用一个好盘代替它，并且将镜像磁盘拷贝到新磁盘上。最终，有了两个相互镜像的磁盘，并且系统被恢复到它的早先的状态。

可能造成错误的惟一事件是在拷贝期间镜像磁盘又出现故障。现在，至少有部分数据两个拷贝都已经丢失，而且没有任何办法恢复。

但是，这类事件序列发生的频繁程度如何呢？假设替换故障磁盘的过程花费3小时，这是一天的1/8，或者一年的1/2920。由于假定每个盘的平均故障时间是10年，拷贝过程中发生故障的可能性是(1/10) × (1/2920)，即1/29 200。如果一个磁盘每10年发生一次故障，那么两个磁盘之一平均5年发生一次故障。每29 200个这种故障中有一个导致数据丢失。换句话说，导致数据丢失的平均时间是5 × 29 200 = 146 000年。 □

11.7.3 奇偶块

尽管镜像磁盘是减少磁盘崩溃造成数据丢失的可能性的一种有效方法，但它所使用的冗余盘与它所拥有的数据盘一样多。另有一种被称为RAID 4级的方法，不管有多少个数据盘，仅使用一个冗余盘。假设磁盘相同，所以可以给每一个磁盘编号，从1到数字n。当然，所有磁盘上的所有块都有着相同的二进制位数。例如，在举例的Megatron 747磁盘中有16 384字节块，

$8 \times 16\,384 = 131\,072$位。在冗余盘中，第 i 块由所有数据盘的第 i 块奇偶校验位组成。也就是说，所有第 i 块的第 j 位，包括数据盘和冗余盘，在它们中间必须有偶数个1，于是总是选取冗余盘的位使这个条件为真。 552

从例11.17已知如何使得条件为真。如果有奇数个数据盘的第 j 位为1，在冗余盘中，选取位 j 为1。如果在数据盘中的第 j 位有偶数个1，就选取冗余盘的位 j 为0。关于这个计算的术语是模2和（modulo-2 sum）。也就是说，如果在若干个位当中有偶数个1，则这些位的模2和为0。如果有奇数个1，则模2和为1。

例11.19 举一个极简单的例子，假定块仅由一个字节——8位组成。令有三个数据盘，分别称为盘1、盘2和盘3，还有一个冗余盘，称为盘4。集中考虑所有这些盘的第一块。如果在数据盘的第一块中，有如下位序列：

盘1：11110000
盘2：10101010
盘3：00111000

那么冗余盘的第一块将有奇偶校验位：

盘4：01100010

注意：四个8位序列中的每一个位置上是如何分布着偶数个1的。在位置1、2、4、5和7有两个1，在位置3有四个1，而在位置6和8有零个1。 □

读操作

从一个数据盘读块与从任何一个磁盘读块没有什么差别。虽然能够从冗余盘读，但通常没有任何理由这样做。在某些情况下，实际上能够获得来自一个数据盘的两个同步读操作的效果，下面的例子解释了这种情况，尽管使用它的条件很罕见。

例11.20 假设正在读第一个数据盘的一个块，另一个请求进入要读同一磁盘的一个不同的块，比如说块1。通常，必须等待第一个请求完成。然而，如果其余的磁盘都不忙的，就能读这些磁盘的块1，并且通过取模2和计算出第一磁盘的块1。

具体来说，如果磁盘及其第一个块数据如例11.19所示，那么就能读第二、第三数据盘和冗余盘，得到下列块： 553

盘2：10101010
盘3：00111000
盘4：01100010

如果取每一列的那些位的模2和，就能得到

盘1：11110000

这与第一磁盘的块1相同。 □

写操作

写数据盘的一个新块时，不仅需要改变那个块，而且需要改变冗余盘的相应块，以便它能继续保持是所有数据盘相应块的奇偶校验。一个朴素的方法是读取 n 个数据盘的相应块，取它们的模2和，并重写冗余盘的块。这个方法要求执行不被重写的数据块的 $n-1$ 次读、重写数据块的一次写以及冗余盘块的一次写。这样总共有 $n+1$ 次磁盘I/O。

一种更好的方法是只关注正在被重写的数据块 i 的老版本和新版本。如果取它们的模2和，

就可以知道，所有磁盘上编号i的块中那个位置上的1的总数是否有变化。由于这些变化总是一种方式，将1的个数由任意一个偶数变成了一个奇数。这样如果改变冗余块的相同位置，那么每个位置的1的个数重新变为偶数。使用4个磁盘I/O，就可以执行下列计算：

1. 读要被改变的数据盘上的旧值。
2. 读冗余盘的相应块。
3. 写新数据块。
4. 重新计算并写冗余盘的块。

模2和代数

了解有关位向量的模2和运算的代数定律，对于理解奇偶校验所采用的一些策略可能很有帮助。用符号⊕表示这种操作。例如，1100⊕1010 = 0110。下面是关于⊕操作的一些有用的定律。

- 交换率：$x \oplus y = y \oplus x$
- 结合率：$x \oplus (y \oplus z) = (x \oplus y) \oplus z$
- 相应长度的全0向量(用$\bar{0}$表示)是⊕的恒等元素，即 $x \oplus \bar{0} = \bar{0} \oplus x = x$
- ⊕是其自身的逆：$x \oplus x = \bar{0}$。作为一个有用的结论，如果$x \oplus y = z$，则可以在等号两边“加”x，得到$y = x \oplus z$

例11.21 假设三个数据盘的第一块如例11.19所示，为：

盘1：11110000
盘2：10101010
盘3：00111000

另假设，第二个磁盘上的块由10101010变为11001100。如果求盘2上旧值与新值的模2和，
554 得到01100110。这个结果表明，必须改变冗余盘第一块位置2、3、6和7上的值。于是先读该块：01100010，并用通过改变其适当位置而得到的新块替换这个块。事实上，一般是以冗余盘自身与01100110的模2和替换冗余块，得到00000100。表示新冗余块的另一个方法是，它是正在被重写的块的旧版本与新版本以及冗余块旧值的模2和。在例中，四个盘（三个数据盘和一个冗余盘）的第一块，在盘2上写入块和对冗余块进行必要的重新计算之后已经变为：

盘1：11110000
盘2：11001100
盘3：00111000
盘4：00000100

注意，在上述块中，每一列依旧有偶数个1。

顺便指出，像所有采用前面所描述方案的数据块写一样，数据块的这次写要花去4个磁盘I/O。朴素的方案——读除重写块之外的全部块并直接重新计算冗余块——在本例中也要求4个I/O：两个用于从第一和第三数据盘读数据，两个用于写第三数据盘和冗余盘。但是，如果有三个以上的数据盘，朴素方案的I/O数将随数据盘的增加而线性上升，而这里提倡的方案依旧
555 只需要4个I/O请求。 □

故障恢复

现在考虑，如果磁盘之一崩溃了，应该做些什么。如果故障盘是冗余盘，就换进一个新磁

盘，并重新计算冗余块。如果故障盘是数据盘之一，那么需要换进一个好盘，并且根据其他盘重新计算它的数据。重新计算任何丢失数据的规则实际上很简单，并且不依赖于是数据盘故障，还是冗余盘故障。由于事先知道，所有磁盘中相应位的1的个数是偶数，它遵循如下规则：

• 任何位置的位是所有其他磁盘的相应位置所有位的模2 和。

如果谁不相信上述规则，就只需要考虑两种情况。如果所讨论的位是1，那么相应位1的个数必须是奇数，所以它们的模2和是1。如果所讨论的位是0，那么相应位中有偶数个1，并且它们的模2和是0。

例11.22 假设盘2发生故障。需要重新计算替换盘的各个块。依照例11.19，看看如何重新计算第二盘的第一个块。已经给出第一和第三数据盘以及冗余盘的相应块，所以总的情况如下：

盘1：11110000
盘2：????????
盘3：00111000
盘4：01100010

如果取每一列的模2和，可以推导出，丢失的块是10101010，正是例11.19一开始的情况。□

11.7.4 一种改进：RAID 5

除非两个磁盘几乎同时发生崩溃，11.7.3节描述的RAID 4级方法就能有效地保护数据。但是，重新分析写一个新数据块的过程，就可以看到它存在一个瓶颈。无论采用什么更新硬盘的方案，都需要读和写冗余盘的块。如果有n个数据盘，那么冗余盘的写次数，将是任何一个数据盘平均写次数的n倍。

然而，正如在例11.22中所看到的，对数据盘和冗余盘的恢复规则相同：取其他磁盘相应位的模2和。这样，就不必把一个盘作为冗余盘，而把其他盘作为数据盘。相反，可以把每个磁盘都作为某些块的冗余盘来处理。这种改进通常称为RAID 5级。 556

例如，如果有$n+1$个编号为$0 \sim n$的磁盘，并且j是当i被$n+1$除时的余数。则可以把盘j的第i个柱面看做冗余。

例11.23 在使用的例子中，$n=3$，就是有4个磁盘，第一个盘的编号为0，将作为编号4、8、12等柱面的冗余，因为当被4除时，这些柱面编号的余数是0。编号为1的盘将作为编号1、5、9等块的冗余，盘2是编号2、6、10等块的冗余，而盘3是编号3、7、11等块的冗余。

结果，每个盘的读负载和写负载相同。如果所有的块有同样的被写可能性，那么对于每次写，每个盘有要写的块的几率是1/4。如果不是这样，那么它有1/3的机会作那个块的冗余盘。这样，4个盘中的每一个被写的几率是$1/4+3/4 \times 1/3=1/2$。□

11.7.5 多个盘崩溃时的处理

如果使用足够多的冗余盘，运用纠错码原理可以处理多个磁盘（无论是数据盘还是冗余盘）崩溃。这个策略引出最高的RAID“级”——RAID 6级。这里仅给出一个简单的例子，例子中可纠正两个同时发生的崩溃。该策略是基于最简单的纠错码——*海明码*（Hamming code）。

在本文的描述中，关注带有7个磁盘的系统，磁盘编号是1～7。前4个盘是数据盘，盘5～盘7是冗余盘。数据盘和冗余盘之间的关系被概括为0和1组成的3×7矩阵，如图11-17所示。注意：

a) 除了全0列之外，0、1组成所有可能三位数列，都出现在图11-17的矩阵中。

b) 冗余盘对应的列只有一个1。

c) 数据盘对应各列至少各有两个1。

	数据盘				冗余盘		
盘号	1	2	3	4	5	6	7
	1	1	1	0	1	0	0
	1	1	0	1	0	1	0
	1	0	1	1	0	0	1

557 图11-17 能从两个同时发生的磁盘崩溃中恢复的系统的冗余模型

如果观察来自全部7个盘的相应的位，并且把注意力集中在该行有1的那些磁盘上，那么由0和1组成的三行的每一行的含义是这些位的模2和必须是0。换言之，矩阵中给定行内带有1的磁盘，可以看做是RAID 4级方案中的整个磁盘组合。这样，通过找出该磁盘有1的那一行，并且取同一行有1的其他磁盘的相应位的模2 和，就能计算出冗余盘之一的相应位。

对于图11-17的矩阵，这个规则意味着：

1. 盘5的位是盘1、2和3相应位的模2和。
2. 盘6的位是盘1、2和4相应位的模2和。
3. 盘7的位是盘1、3和4相应位的模2和。

从上例可以很快看出，矩阵中位的特定选择有一个简单的规则，通过该规则，人们能够将两个同时发生崩溃的磁盘恢复。

读操作

可以从任何一个数据盘中正常地读数据。可以忽略冗余盘。

写操作

写操作的想法类似于11.7.4节中概括描述的写策略，但是现在可能要涉及几个冗余盘。为了写某个数据盘的一个块，需要计算那个块的新版与旧版的模2和。这些位以模2和的方式加入到所有冗余盘的相应块中，条件是这些冗余盘在该数据所写盘中为1的某一行中同样为1。

例11.24 再一次假设块只有8位长，并且关注在RAID 6级举例中用到的7个磁盘的第一块。首先，假设数据盘和冗余盘的第一块内容如图11-18所示。注意，盘5的块是前3个盘的块的模2和，第6行是行1、2和4的模2 和，而最后一行是行1、3和4的模2和。

磁 盘	内 容
1)	11110000
2)	10101010
3)	00111000
4)	01000001
5)	01100010
6)	00011011
7)	10001001

图11-18 所有磁盘的第一块

假设要将盘2的第一块重写为00001111。如果计算这个序列与序列10101010（该块的旧值）的模2和，则得到10100101。如果观察图11-17中盘2的列，就会发现，该盘前两行中有1，但是第三行没有。由于冗余盘5和盘6分别在行1和行2有1，因此必须分别
558 对它们第一块的当前内容和刚刚算出的序列10100101执行求模2和操作。也就是说，对这两个块在位置1、3、6和8上的值求反。所有磁盘的第一块的结果内容如图11-19所示。注意，新内容依旧满足图11-17所包含的约束：图11-17矩阵中有1的特定行的相应块的模2和依然是全0。 □

磁 盘	内 容
1)	11110000
2)	00001111
3)	00111000
4)	01000001
5)	11000111
6)	10111110
7)	10001001

图11-19 在重写盘2并改变冗余盘之后所有盘的第一块

故障恢复

现在来看看前面概述的冗余方案如何能够用于纠正两个同时发生的磁盘崩溃。令故障盘为

a和b。由于图11-17矩阵的所有列都不同，所以必定能够找到某行r，在r中a和b对应的值不同。假设在行r，a有0，而b有1。

然后，通过来自除b之外的所有在行r有1的磁盘的相应位的模2和，就能够计算出正确的b。注意：a不在其中，所以它们都没有发生故障。这样做完后，必须用所有其他可用盘来重新计算a。由于图11-17矩阵的每一列都会在某一行里有一个1，因此能够使用这一行去重新计算磁盘a，办法是取该行中有1的其他磁盘的位的模2和。 559

例11.25 假设盘2和盘5几乎在同一时刻发生故障。参照图11-17的矩阵，可以发现，这两个盘的列在行2不同，盘2有1而盘5有0。这样，通过取盘1、4和6（行2带1的其他三个盘）的相应位的模2和，可以按原样修复盘2。注意，这三个盘都没有发生故障。例如，根据与图11-19中第一块有关的情况可以得出，在盘2和盘5出现故障后，最初有图11-20所示的可用数据。

如果取盘1、4和6块内容的模2和，可以发现，盘2的块为00001111。从图11-19可以验证这个块的正确性。结果如图11-21所示。

磁 盘	内 容
1)	11110000
2)	????????
3)	00111000
4)	01000001
5)	????????
6)	10111110
7)	10001001

图11-20 盘2和盘5发生故障后的状况

磁 盘	内 容
1)	11110000
2)	00001111
3)	00111000
4)	01000001
5)	????????
6)	10111110
7)	10001001

图11-21 盘2恢复之后的状况

现在，可以看到，图11-17中盘5的列在第一行有一个1。因此通过取与盘1、2和3（在第一行有1的其他三个盘）的相应位的模2和，可以重新计算盘5。对于块1，这个和是11000111。而且这个计算的正确性可通过图11-19确认。 □ 560

关于RAID 6级的附加评述

1. 通过根据块号或柱面号变换冗余盘的方式，可以将RAID 5级和6级的思想结合起来。这样做将避免写操作时的瓶颈。在11.7.5节给出的方案将在冗余盘产生瓶颈。

2. 在11.7.5节的方案不只限于4个数据盘。磁盘的数量可以是2的任意次方减1，即2^k-1。在这些盘中，有k个是冗余盘，而剩余的2^k-k-1个是数据盘，所以冗余盘差不多是按数据盘数目的对数增长。对于任意k，通过写k个0和1构成的所有可能的列（全0列除外），可以构造出与图11-7相对应的矩阵。有单个1的列是冗余盘，而有多个1的列是数据盘。

11.7.6 习题

习题11.7.1 假设使用例11.18中的镜像盘，每年故障率为4%，更换一个盘要花8小时。导致数据丢失的磁盘平均故障时间是多少？

*! **习题11.7.2** 假设磁盘每年的故障百分率为F，更换一个盘要花费H小时。

a) 如果使用镜像盘，作为F和H的函数，数据丢失的平均时间是多少？

b) 如果采用RAID 4级和5级方案，使用N个磁盘，数据丢失的平均时间是多少？

!! **习题11.7.3** 假设使用3个磁盘作为一个镜像组，即所有3个盘保存相同的数据。如果一个磁盘每年的故障率是F，恢复一个磁盘要花费H小时，数据丢失的平均时间是多少？

习题11.7.4 假设使用RAID 4级方案，有4个数据盘和一个冗余盘。与例11.19一样，假设

块为单字节。如果数据盘的相应块如下，给出冗余盘的块。

* a) 01010110，11000000，00111011和11111011。

561 b) 11110000，11111000，00111111和00000001。

习题11.7.5 采用与习题11.7.4相同的RAID 4级方案，假设数据盘1有故障。请在下列情况下恢复该磁盘的块：

* a) 盘2～盘4的内容为01010110、11000000和00111011，同时冗余盘的内容是11111011。

b) 盘2～盘4的内容为11110000、11111000和00111111，同时冗余盘的内容是00000001。

纠错码和RAID 6级

存在一个指导选择如图11-17那样的合适矩阵的原理，以确定冗余盘的内容。一个长度为n的代码是一组长度为n的位向量（称为码字）。两个码字之间的海明距离是这两个码字取值不同的位置的数量，而一个代码的最小距离是任何两个不同码字的最小海明距离。

如果C是长度为 n 的任意代码，可以要求 n 个磁盘的相应位是这样的一个序列，它是代码C的成员。举一个很简单的例子，如果正在使用一个磁盘和它的镜像，那么$n = 2$，就可以使用代码$C = \{00,11\}$。也就是说，两个磁盘的相应位必须相同。对于另一个例子，图11-17的矩阵定义了长度为7的16个位向量组成的代码，前4位有任意的值，剩余的3位由3个冗余盘规则决定。

如果代码的最小距离是d，则要求相应位是代码中的一个向量的磁盘，将能够承受$d-1$个磁盘同时发生崩溃。理由是，假如使一个码字的$d-1$个位置模糊不清，并且有两种不同的方法填写这些位置，以形成两个码字，那么这两个码字最多在这$d-1$个位置不同。这样，代码就不会有最小距离d。作为示例，图11-17矩阵实际上定义了著名的海明码，它的最小距离是3。这样，它能处理两个磁盘损坏。

习题11.7.6 假设习题11.7.4 第一个盘中的块被改为10101010。问其他盘上相应的块必须做什么样的改变？

习题11.7.7 如果使用例11.24的RAID 6级方案，4个数据盘的块分别为00111100、

562 11000111、01010101和10000100。

a) 冗余盘的相应块是什么？

b) 如果第三个盘的块被重写成10000000，必须采取哪些步骤以改变其他盘？

习题11.7.8 使用带有7个磁盘的RAID 6级方案，描述从下列故障中恢复所要采取的步骤：

* a) 盘1和盘7。

b) 盘1和盘4。

c) 盘3和盘6。

习题11.7.9 找出使用15个磁盘（其中4个盘是冗余盘）的RAID 6级方案。提示：广义化7个磁盘的海明矩阵。

习题11.7.10 列出长度为7的海明码的16个码字。也就是说，对于基于图11-7矩阵的拥有7个磁盘的RAID 6级方案，能够同它的相应位对应的16个位表是什么？

习题11.7.11 假设有4个磁盘，其中盘1和盘2是数据盘，盘3和盘4是冗余盘。盘3是盘1的

镜像。盘4保存着盘2和盘3 相应位的奇偶校验位。

a) 通过给出类似于图11-17的奇偶校验矩阵，表达上述情况。

!! b) 在某些情况（但不是所有情况）下，能够在两个磁盘同时发生故障时恢复数据。请确定，哪一对盘可能恢复，哪一对盘不可能恢复。

*! **习题11.7.12** 假设有8个数据盘，编号为1～8。有3个冗余盘，编号为9、10和11。盘9是关于盘1～盘4的奇偶校验盘，盘10是关于盘5～盘8的奇偶校验盘。如果所有磁盘同时发生故障的可能性相同，现在希望从两个磁盘同时发生故障中恢复的概率达到最大，那么盘11应该是哪些磁盘的奇偶校验盘？

!! **习题11.7.13** 找出一个具有10个磁盘的RAID 6级方案，要求能从任何三个磁盘的同时故障中恢复。你应该使用尽可能多的数据盘。

11.8 小结

- *存储器层次*：一个计算机系统使用多种存储部件，这些存储部件在速度、容量和每个二进制位的费用方面的范围涉及多个数量级。从最小/最贵到最大/最便宜，它们分别是：高速缓存、主存储器、二级存储器（磁盘）和三级存储器。 563
- *三级存储器*：三级存储器的基本设备是盒式磁带、磁带仓（管理磁带盒的机械设备）和自动光盘机（管理CD－ROM盘的机械设备）。这些存储设备可容纳许多个太字节，但是均属于最慢的可用存储设备。
- *磁盘/二级存储器*：二级存储器设备主要是具有多个吉字节容量的磁盘。磁盘设备有若干个磁性材料的圆盘，圆盘上有存储二进制位的同心圆磁道。圆盘围绕着中心轴旋转。距离圆盘中心的给定半径上的所有磁道形成一个柱面。
- *块和扇区*：磁道被分成扇区，扇区被非磁化间隙分隔。扇区是读盘和写盘的基本单位。块是DBMS这样的应用进行存储的逻辑单位。块通常由若干个扇区组成。
- *磁盘控制器*：磁盘控制器是控制一个或多个磁盘装置的处理器。它负责将磁头移动到合适的柱面，以便读写一个所要求的磁道。它还可以调度对磁盘访问的竞争请求，并且缓冲要读或要写的块。
- *磁盘访问时间*：磁盘延迟是指从发出一个读块或写块请求到该访问完成的时间。延迟大体上由三个因素引起：移动磁头到合适柱面的寻道时间，所要求的块转到磁头下的旋转延迟以及传输时间，即块在磁头下面移动进行读或写的时间。
- *摩尔定律*：从计算机处理器速度以及磁盘和主存储器容量等参数每18个月翻一番得出的恒定发展趋势。但是在同一个时期内，磁盘访问时间只减少一点点。一个重要的结论是，磁盘访问的（相对而言）费用在逐年增长。
- *使用二级存储器的算法*：当数据量大到主存装不下的时候，用于操纵数据的算法必须考虑到这样的事实：在磁盘与存储器之间读写磁盘块所花费的时间通常要比在内存中处理数据的时间长得多。因此，对二级存储器中数据的算法的评价集中在磁盘I/O的请求数上。
- *两趟、多路归并排序法*：这个排序算法仅仅用到各个数据的两次磁盘读和两次磁盘写，就能对磁盘上的极大量的数据进行排序。它是被大多数数据库应用所选择的排序方法。 564
- *磁盘访问的加速*：对于某些应用来说，有多种技术可以较快地访问磁盘块。这些技术包括：将数据分布到若干个磁盘当中（以允许并行访问），镜像磁盘（保存数据的多个拷贝，也是允许并行访问），组织数据使数据以多个磁道或柱面为一个整体的方式被访问，以及通过同时读或写整个磁道或柱面来预取或双缓冲数据。

- *电梯算法*：通过按一定顺序处理访问请求排队来提高访问速度，这种顺序将使得磁头能够作一次跨越整个磁盘的扫描。每当磁头到达的柱面包含等待中的访问请求所需要的一个或多个块时，磁头就停下来处理请求。
- *磁盘故障类型*：为了避免数据丢失，系统必须能处理错误。磁盘故障的基本类型包括：间断性故障（如果重复多次，读或写错误将不会再发生）、永久性故障（磁盘上的数据已损坏，并且不能被正确读出）以及磁盘崩溃（此时整个磁盘成为不可读）。
- *校验和*：通过增加奇偶校验（外加的二进制位，使得一个位串中1的个数为偶数）。借助校验和，间断性故障和永久性故障可以被检测出来，尽管不能纠正这些故障。
- *稳定存储*：通过制作所有数据的两个拷贝，并且注意写那些拷贝的顺序，单个磁盘可以用于防止单个扇区的几乎所有的永久性故障。
- RAID：通过使用一个或几个额外的磁盘，可以有若干个方案使数据幸免于磁盘崩溃。RAID1级是磁盘镜像。4级需另加一个磁盘，其内容是所有其他磁盘的相应位上的奇偶校验。5级用不同的磁盘保存奇偶位，以避免单个奇偶盘成为写操作的瓶颈。6级涉及纠错码的使用，可以使数据幸免于多个磁盘的同时崩溃。

11.9 参考文献

RAID 思想可以追溯到文献[6]中关于磁盘分条（striping）的内容。名称和纠错能力来自文献[5]。

11.6节中的磁盘故障类型出现在Lampson和Sturgis未发表的著作[4]中。

有几篇与本章相关的有用的综述。文献[2]讨论了磁盘存储器以及类似系统的发展趋势。RAID系统的研究参见[1]。[7]综述了适用于二级存储器计算模型（块模型）的算法。

[3]是一篇重要的论文，研究如何为执行特定任务，优化一个包括处理器、存储器和磁盘在
565 内的系统。

1. P. M. Chen et al., "RAID: high-performance, reliable secondary storage," *Computing Surveys* **26**:2 (1994), pp. 145–186.

2. G. A. Gibson et al., "Strategic directions in storage I/O issues in large-scale computing," *Computing Surveys* **28**:4 (1996), pp. 779–793.

3. J. N. Gray and F. Putzolo, "The five minute rule for trading memory for disk accesses and the 10 byte rule for trading memory for CPU time," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1987), pp. 395–398.

4. B. Lampson and H. Sturgis, "Crash recovery in a distributed data storage system," Technical report, Xerox Palo Alto Research Center, 1976.

5. D. A. Patterson, G. A. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks," *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 109–116, 1988.

6. K. Salem and H. Garcia-Molina, "Disk striping," *Proc. Second Intl. Conf. on Data Engineering*, pp. 336–342, 1986.

7. J. S. Vitter, "External memory algorithms," *Proc. Seventeenth Annual*
566 *ACM Symposium on Principles of Database Systems*, pp. 119–128, 1998.

第12章 数据元素的表示

本章将我们在11.4节所讲的第二级存储器的块模型与数据库管理系统的要求联系起来。首先，我们看一下关系或对象集在第二级存储器中的表示方法。

- 属性需要用定长或变长的字节序列表示，称做“字段”。
- 而后，字段被组装成定长或变长的集合，称为“记录”，记录对应于元组或对象。
- 记录需要存储在物理块中。多种不同的数据结构是有用的，特别是当修改数据库需要重新组织记录块时。
- 构成一个关系或类外延的记录集存储为块的集合，称为文件[⊖]，为支持对这些集合的有效查询和修改，我们在文件上添加许多索引结构中的一种；这些结构是第13章和第14章讨论的内容。

12.1 数据元素和字段

我们首先研究最基本的数据元素的表示，即关系或面向对象数据库系统中的属性值的表示。这是用“字段”来表示的。然后我们考察字段如何组装成存储系统中更大的元素：记录、块和文件。 567

12.1.1 关系数据库元素的表示

假设我们用图12-1所示的CREATE TABLE 语句在SQL系统中声明一个关系。DBMS负责表示和存储由这个定义描述的关系。既然关系是元组集合，元组与记录或“结构”（C或C++术语）相似，我们可以设想每一个元组在磁盘中作为一条记录来存储。记录会占据某个磁盘块（或其一部分），在记录内部，对应于关系的每一个属性有一个字段。

```
CREATE TABLE MovieStar(
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE
);
```

图12-1 一个SQL表的声明

尽管总的想法看起来简单，但“可怕的是实现细节”，我们必须讨论很多问题：

1. 如何将SQL数据类型表示成字段？
2. 如何将元组表示成记录？
3. 如何在存储块中表示记录或元组的集合？
4. 如何用块的集合表示和存储关系？

⊖ 数据库中“文件”的概念比操作系统中“文件”的含义更广泛一些。数据库文件可以是一个无结构的字节流，但更经常的情况是由块的一个集合构成。这些块按某种有用的方式组织在一起，具有索引或其他专门的访问方式。我们在第13章讨论这些结构。

5. 如果不同的元组可能具有不同的记录大小，或者记录大小不能整除块大小，或二者兼而有之，这时我们如何处理记录大小？
6. 如果因为修改一些字段而导致记录大小发生改变，那么将会发生什么情况？我们如何在记录所在的块内找到存储空间，特别是当记录增大时？

第一个问题是本节要讨论的，下边两个问题在12.2节讨论。我们将分别在12.4节和12.5节讨论最后两个问题。第四个问题，即如何表示关系以有效存取其元组，将在第13章中研究。

568 此外，我们需要考虑如何表示现代对象关系或面向对象系统中存在的某些数据类型，如对象标识（或其他指向记录的指针）和“blobs”(二进制的大对象，如2 GB的MPEG视频)。这些问题将在12.3节和12.4节讨论。

12.1.2 对象的表示

大致上讲，一个对象就是一个元组，而它的字段或“实例变量”就是属性。对象－关系系统中的元组与普通关系系统中的类似。尽管如此，除在12.1.1中讨论的情况外，还有两种重要的拓展：

1. 对象可以有与其相关的方法或专用的函数，这些函数代码是对象类模式的一部分。
2. 对象可以有对象标识（OID），它在某个全局地址空间中惟一地指代该对象的地址。另外，对象可以与其他对象有联系，这些联系由指针或指针序列表示。

方法通常和模式存储在一起，因为它们完全属于数据库中的一个整体，而不是任何特殊的对象。但是为了访问方法，一个对象的记录必须有一个可以表示该方法属于何种类型的域。

无论是对象的标识还是对其他对象的引用，其地址表示技术都将在12.3节中讨论。正如ODL所规定的，关联作为对象的一部分，在存储中也同样需要给予关注。因为我们不知道有多少个被关联的对象（至少不是多对多联系或多对一联系中的“多”的那一方面），我们必须用变长记录表示联系，这也是12.4节中讨论的主题。

12.1.3 数据元素的表示

我们首先考虑基本SQL数据类型如何表示成记录的字段。所有的数据最终被表示成一个

569 字节序列。例如，一个类型为 INTEGER的属性通常表示成两个或四个字节，一个类型为FLOAT的属性通常表示成四个或八个字节。整数和实数表示成字节串，由机器的硬件对字节串进行特定解释，从而在其上可执行通常的算术操作。

定长字符串

要表示的最简单的字符串类型是由SQL类型CHAR(*n*)描述的，它们是长度为*n*的定长字符串。对应于具有这种类型属性的字段是*n*字节的数组。假如这个属性的值是长度小于*n*的字符串，则字节数组用特定的填充符号填充，填充符号的8位编码不是SQL字符串的合法字符。

例12.1 如果一个属性*A*声明是类型CHAR(5)，则在所有的元组中与*A*对应的字段是一个5字符数组。如果在一个元组中，对应于属性*A*的成分是“cat”，则这个数组的值是：

cat⊥⊥

其中，⊥是“填充”字符，它占据这个数组的第四和第五个字节。注意，在SQL程序中指明字符串所必须使用的单引号并不与字符串的值一起存储。 □

变长字符串

有时，一个关系中某一列的值是长度变化很大的字符串。SQL类型VARCHAR（n）常被用作这样的列的类型。但是，对以这种方法声明的属性我们打算采用的实现方法是将n+1字节用于字符串值，而不管字符串值的长度。这样，SQL VARCHAR类型实际上表示定长字段，尽管它的值的长度可变。我们将在12.4节考查其表示长度可变的字符串。对VARCHAR字符串有两种常见的表示： 570

1. *长度加内容*。我们分配一个 n+1字节的数组。第一个字节存储字符串中字节数，是一个8位二进制整数。字符串不能超过 n字符，而 n 本身不能超过255，否则我们将不能在一个字节内表示长度[⊖]。第二个及以后的字节存储字符串中的字符。如果因为字符串比可能的最长值要短，数组中存在不被使用的字节，那么这些字节都被忽略。它们不可能被当作值的一部分，因为第一字节告诉我们字符串何时终止。

2. *空值-终止字符串*。我们再为字符串的值分配一个n+1字节的数组。用字符串的字符填充这个数组，其后跟一个空字符，它不是可以出现在字符串中的合法字符。与第一种方法类似，数组中没有使用的位置不可能被当作值的一部分；这里，空值终止符警告我们不能再继续往下找，同时也使VARCHAR字符串的表示与C中字符串的表示兼容。

对术语的注释

如果你使用过文件系统、传统编程语言如C、关系数据库语言（特别地，如SQL）或面向对象的语言（如，Smalltalk，C++，或面向对象的数据库语言OQL），根据你经验的不同，对一些实质上相同的概念你可能知道的术语也不一样。下面这个表总结了这些术语的对应关系，尽管它们有差异，例如，类可有方法，而关系不可以有。

	数据元素	记录	集合体
文件	字段	记录	文件
C	字段	结构	数组，文件
SQL	属性	元组	关系
OQL	属性，联系	对象	（一个类的）外延

例12.2 假设属性A声明为VARCHAR(10)。在每个元组的记录中，我们为A的值分配11个字符的一个数组。假设cat是要表示的字符串，则在方法1中，我们将会把3放入第一个字节中来表示字符串的长度，接下的3个字符是字符串本身。最后的7个位置是无关的。从而值看来是：

```
3cat
```

注意“3”是8位二进制整数3，即00000011，而不是字符“3”。

在第二种方法中，我们用字符串填充前3个位置，第四个位置是空字符（我们使用符号⊥，正如我们选择“填充”字符一样），余下的7个位置是无关的。从而，是cat的空值-终止字符串表示。 □

```
cat⊥
```

⊖ 当然我们可使用两个或更多字节专用于长度的模式。

日期和时间

正如在6.1.4节中提到的，日期通常表示为符合某种格式的定长字符串，因此我们可以像表示其他定长字符串一样表示日期。一个时间很容易表示为定长字符串。但是SQL标准中还允
571 许使用TIME类型的值以包含秒的小数。因为这种字符串是任意长的，所以我们有两种选择：

1. 系统可在时间的精确度上加以限制，然后把时间作为类型VARCHAR(*n*)存储，*n*是时间可以有的最大长度，即9加上秒中允许的小数位数。

2. 可把时间作为真正的变长值存储，如在12.4节中讨论的那样来处理它们。

二进制位

二进制位序列（在SQL中用类型BIT(*n*)描述的数据）可以按每8位构成一个字节的方式组装起来。如果*n*不能被8整除，则我们最好忽略最后一个字节中未用的二进制位。例如，二进制位序列010111110011表示成01011111为第一个字节，00110000为第二个字节，而最后4个“0”不是任何字段的一部分。布尔值可以作为这种情况的特例即单独的一位来表示，即10000000为“真”而用00000000表示“假”。但在某些情况下，如果我们使每一位都不同，例如用11111111表示“真”而00000000表示“假”，则布尔值的检测更容易一些。

枚举类型

有时让属性从一个小的、固定的值集中取值是有用的。这些值被赋予符号名称，而包含所有这些名称的类型是一个枚举类型。有关枚举类型的常见例子是一周中的各天，如{SUN, MON, TUE, WED, THU, FRI, SAT}，或颜色集，如{RED, GREEN, BLUE, YELLOW}。

我们可以使用整数编码表示一个枚举类型的值，使用的字节数恰好能满足所需。例如，我们可以用0表示RED，1表示GREEN，2表示BLUE，3表示YELLOW。这些整数都可以用两个二进制位表示，分别是00、01、10和11。但使用整个字节表示从一个小集合中选择的整数更方便，如YELLOW由整数3表示，即一个8位字节00000011。任何不大于256个值的枚举类型都可由单一字节表示，如果枚举类型有2^{16}个值，则两个字节的短整数就足够了。依此类推。

12.2 记录

现在我们开始讨论字段如何组装成记录，12.4节将继续讨论变长字段和记录。

总的来说，数据库系统使用的每一种记录类型必须有一个模式。模式由数据库存储，包括记录中字段的名称和数据类型，以及在记录内它们的偏移量。需要存取记录的组成部分时
572 将参考模式。

将字段组装成单一字节

人们可能想充分利用小的枚举类型或布尔型字段的特点，将几个这样的字段组装成单一字节。如，我们有三个字段，分别是一个布尔型、周中的某一天和四种颜色中的一种，那么我们可使用1个二进制位表示第一个字段，3个二进制位表示第二个字段，两个二进制位表示第三个字段。将这三个字段全放入一个字节中，这个字节还能剩下两个二进制位。尽管完全可以这样做，但是它使得检索其中一个字段的值和向其中一个字段写入新值变得比较复杂，而且容易出错。这种字段组装方法曾经比较重要，因为那时存储空间比较昂贵。现在，我们在一般情况下不主张使用它。

12.2.1 定长记录的构造

元组由记录表示，而记录由12.1.3节所讨论的各种字段组成。最简单的情况是记录的所有

字段均为定长，则我们可将字段连接成记录。

例12.3 考虑图12-1中关系MovieStar的声明，有下列四个字段：

1. name，一个30字节的字符串。
2. address，VARCHAR(255)类型。如果采用例12.2中讨论的方案，这个字段用256个字节表示。
3. gender，单一字节，我们认为它总是保存字符“F”或字符“M”。
4. birthdate，DATE类型。我们将假设用10字节长的SQL日期表示这个字段。

所以，MovieStar类型的记录占30+256+1+10=297字节，如图12-2所示。我们指出了每个字段的偏移量，即从记录开始到这个字段自身开始处的字节数。从图中可以看到，字段name在偏移量0处开始，字段address在偏移量30处开始，gender在偏移量286处开始，birthdate在偏移量287处开始。 □

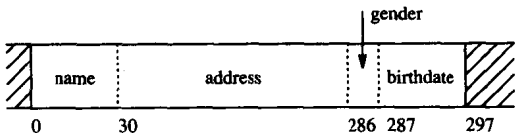


图12-2 一个MovieStar记录

有些机器可以对主存中地址为4的倍数（或8的倍数，如果机器有64位处理器）的字节处开始的数据进行更有效的读写。某些数据类型，如整数，也许要求必须从4的倍数的地址处开始，而其他的数据类型，如双精度实数，可能需要从8的倍数处开始。 573

当关系的元组存储在磁盘中而不是存储在主存中时，我们必须注意这个问题。原因在于当我们将一个块从磁盘读到主存中时，块的第一个字节肯定被放置在4的倍数的主存地址处，事实上将是2的某个高次幂的倍数处，例如，如果块和页的长度为4096=2^{12}，则为2^{12}的倍数。某些字段需装载到首字节地址为4或8倍数的主存处这一需求，因而转换成这些字段在块内的偏移量必须具有与此相同的因子。

为简便起见，我们假设对数据的惟一要求是字段从地址为4的倍数的主存字节处开始，那么有下面两条就足够：

a) 每一条记录在块内从4的倍数的字节处开始，而且

b) 记录中所有的字段都从与记录起始偏移量为4的倍数的字节处开始。

换一种说法，我们将所有的字段和记录长度进到下一个4的倍数。

例12.4 假设关系MovieStar的元组需要表示成每一个字段从4的倍数的字节处开始，则4个字段的偏移量是0、32、288和292，并且整条记录占304个字节，其格式如图12-3所示。

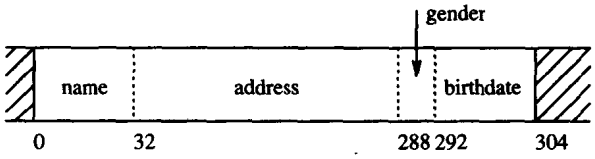


图12-3 MovieStar元组在要求字段从4字节的倍数处开始的格式

例如，第一个字段name占30个字节。对于第二个字段，我们只能从下一个4的倍数或偏移量32处开始，因而address在记录格式中的偏移量为32。第二个字段长度为256个字节，这意味着address后第一个可用的字节为288。虽然第三个字段gender只需一个字节，但是 574

最后一个字段必须从下一个4的倍数即292的字节处开始。第四个字段birthdate长度为10个字节，在字节301处结束，这样，记录的长度为302（注意第一个字节标号为0）。但如果所有记录的所有字段必须在4的倍数的字节处开始，标号为302和303的字节是无用的，因此记录实际占304个字节。我们将把字节302和303分配给字段birthdate，使得它们不会被偶然用于任何其他目的。 □

记录模式的必要性

我们可能想知道，既然目前我们只考虑定长记录，为什么还需要在记录自身中指明记录模式。例如，用于C或类似语言中的一个“结构”中的字段，当程序运行时不存储它们的偏移量，而偏移量被编译到访问该结构的应用程序中。

但有几个理由说明为什么必须存储记录模式，而且让DBMS能够访问它。原因之一，是关系模式（和表示其元组的记录模式）可能会改变。查询需要使用这些记录的当前模式，因而需要知道当前模式。在其他情况下，我们不可能仅仅从记录在存储系统中的位置立刻判断出它的类型。例如，一些存储组织允许不同关系的元组出现在同一存储块中。

12.2.2 记录首部

当我们设计记录的格式时，必然会引出另一个问题：通常在记录中需要保存一些信息，而这些信息不是任何字段的值。例如，我们可能想在记录中保存：

1. 记录模式，或更可能是指向DBMS中存储该类记录模式的位置的一个指针；
2. 记录长度；
3. 时间戳，指明记录最后一次被修改或被读的时间以及其他可能的信息。因此，许多记录格式包括一个由数目不多的字节组成的首部，以提供这种额外信息。

数据库系统维护模式信息，模式信息主要是出现在为那个关系所写的CREATE TABLE语句中的信息：

1. 关系的属性。

575

2. 属性类型。
3. 属性在元组中出现的顺序。
4. 属性或关系自身上的约束，如主键声明，或约束某个整数属性的值必须在某一范围内。

我们不必把所有这些信息都放入元组的记录首部，只需在记录首部设置一个指针，让它指向存储该元组所属关系信息的位置就足够了。那么在需要时所有这样的信息都可以获得。

再如，尽管元组长度可从它的模式中推断出来，但是在记录中存储长度信息会更方便些。例如，我们可能不希望细查记录内容，只想快速地找到下一条记录的开始，长度字段可让我们避免存取记录的模式，而存取记录模式可能要进行的磁盘I/O。

例12.5 让我们修改例12.4中的格式以包含一个12字节长的首部。第一个4字节为类型，实际上它是在存储所有关系模式的区域中的一个偏移量；第二个4字节是记录长度，它是一个4字节整数；第三个4字节是时间戳，指明元组插入或最后一次被修改的时间。时间戳也是一个4字节整数。所得到的格式如图12-4所示。现在记录的长度是316个字节。 □

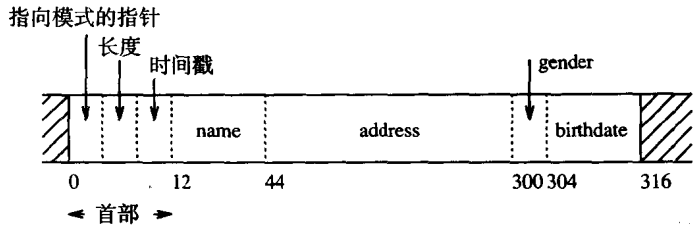


图12-4 给表示关系MovieStar的元组的记录加一些首部信息

12.2.3 定长记录在块中的放置

表示关系元组的记录存储在磁盘块中。当我们需要存取或修改记录时，记录（与整个块一起）就被移进主存。存储记录的块的格式如图12-5所示。

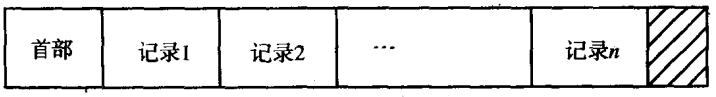


图12-5 一个典型的存储记录的块

它有一个可选的块首部，存储诸如以下各种信息：

1. 与一个或多个其他块的链接，这些块构成一个块的网络，例如在第13章中所描述的为一个关系的元组创建索引的块。

2. 关于这个块在这样一个网络中所扮演的角色的信息。

3. 关于这个块的元组属于哪个关系的信息。

4. 一个给出每一条记录在块内偏移量的“目录”。

5. 一个“块ID”，参见12.3节。

6. 指明块最后一次修改和/或存取时间的时间戳。

到目前为止，最简单的情况是块存储一个关系的元组，并且元组的记录有固定格式。在这种情况下，在块首部的后面，我们把尽可能多的记录装入块内，而留下剩余空间不用。

例12.6 假设我们要存储具有例12.5所示格式的记录，这些记录长度为316字节。我们还假设使用4096字节的块。其中，12个字节用于块首部，剩余4084字节由数据使用。在这个空间中，我们能装入12条给定的316字节格式的记录，每一块有292个字节是被浪费的的空间。 □

12.2.4 习题

* **习题12.2.1** 假设一条记录有如下所示顺序的字段：一个长度为15的字符串，一个2字节整数，一个SQL日期，一个SQL时间（无小数点）。如果

a) 字段可在任何字节处开始；

b) 字段必须在4的倍数的字节处开始；

c) 字段必须在8的倍数的字节处开始；

这条记录占用多少个字节？

习题12.2.2 对字段序列：一个8字节实数，一个长度为17的字符串，单独一个字节，一个SQL日期，重做习题12.2.1。

习题12.2.3 假设字段同习题12.2.1，但是记录有一个首部，它由两个4字节的指针和一个字符组成。对习题12.2.1中的(a)～(c)三种情况，计算记录长度。

习题12.2.4 如果记录包括一条记录首部，它由一个8字节的指针和10个两字节的整数组成，

576
577

重做习题12.2.2。

* **习题12.2.5** 假设记录同习题12.2.3，且我们希望在一个4096字节的块中装入尽可能多的记录，使用的块首部由10个4字节的整数组成。对习题12.2.1中的三种情况，我们各能将多少记录装入块中？

习题12.2.6 假设块长度为16 384字节，块首部由3个4字节的整数和一个对块中每一条记录有一个2字节整数的目录组成。对习题12.2.4的记录重做习题12.2.5。

12.3 块和记录地址的表示

在继续研究如何表示具有更复杂结构的记录之前，我们必须考虑如何表示记录和块的地址、指针或引用，因为这些指针通常构成复杂记录的一部分。有些其他的原因使我们还必须了解第二级存储器中的地址表示。我们在第13章中研究表示文件或关系的有效结构时，将看到块地址或记录地址的几个重要用途。

当块被加载到主存缓冲区时，块地址可作为其第一个字节的虚拟的存储器地址，且块内记录的地址是该记录第一个字节的虚拟的存储器地址。但是在第二级存储器中，块不是应用的虚拟的存储器地址空间的一部分，事实上，有一个字节序列描述块在DBMS可访问的整个数据系统中的地址：磁盘的设备ID，柱面号，等等。记录可通过它所在的块和其第一个字节在块内的偏移量来标识。

最近，“对象代理”成为一种趋势，它允许许多协作系统独立创建对象，这使得地址表示进一步复杂化。这些对象可由作为面向对象DBMS一部分的记录来表示，尽管我们可以将它们想像成关系的元组而不会失去主要思想。然而，对象或记录的独立创建给维护记录地址的机制增加了额外的压力。

在这一节中，我们将首先讨论地址空间，特别是与常用的DBMS“客户-服务器”体系结构有关的地址空间；然后，我们讨论表示地址的可用方法；最后看一下“指针混写”，这种方法
578 使我们能将数据服务器空间中的地址转换成客户端应用程序空间中的地址。

12.3.1 客户–服务器系统

通常，数据库包括一个服务器进程，它为一个或多个客户端进程提供第二级存储器数据，客户端进程是使用数据的应用程序。服务器和不同的客户端进程可以在一台机器上，也可分布在许多机器上。参见8.3.4节，这一节中首次介绍了上述概念。

客户端应用使用常规的“虚拟”地址空间，通常为32位即有大约40亿不同的地址。操作系统或DBMS决定地址空间的哪些部分目前在内存中，而硬件则将虚拟地址空间映射到主存的物理地址。我们不进一步考虑这个虚拟到物理的转换问题，而是将客户端地址空间，看做主存本身。

数据库的数据存在于*数据库地址空间*。这个空间的地址指向块或块内的偏移。这个地址空间寻址的方式可以描述为：

1. *物理地址*：物理地址是字节串，据此我们可确定第二级存储系统内块或记录的位置。下面各项都使用物理地址的一个或多个字节来指明：

(a) 存储所连接的主机（如果数据库存储在不只一台机器上）；

(b) 块所在的磁盘或其他设备的标识符；

(c) 磁盘的柱面号；

(d) 柱面内磁道号（如果磁盘有不只一个盘面）；

(e) 磁道内块号；

(f)（在某些情况下）记录起始在块内的偏移量。

2. *逻辑地址*：每一个块或记录有一个“逻辑地址”，这是具有某个固定长度的一个任意字节串。存储在磁盘上一个已知位置的映射表将逻辑地址与物理地址联系起来，如图12-6所示。

注意，物理地址很长，如果要包含所有列出的元素，8字节是我们能使用的最小长度，有些系统使用的地址长达16字节。例如，想像设计一个要用100年的对象数据库。将来数据库可能会增长到包含一百万台机器，而每一台机器可能会快到每一纳秒创建一个对象，这个系统将创建大约2^{77}个对象，它最小需要10个字节来表示地址。因为我们可能更倾向于预留一些字节来表示主机，另一些字节表示存储单元等，因此对如此规模的系统，一个合理的地址表示可能要使用到比10大得多的字节数。

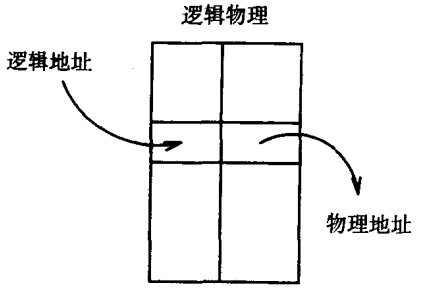


图12-6 映射表将逻辑地址转换成物理地址

579

12.3.2 逻辑地址和结构地址

人们可能想知道使用逻辑地址的目的是什么。虽然物理地址所需的信息都可在映射表中找到，而且跟踪指向记录的逻辑指针需要参考映射表，然后才能找到物理地址，但是与映射表有关的间接层次给我们提供了相当大的灵活性。例如，许多数据组织方式要求我们到处移动记录，或者在块内或者从一个块移到另一个块。如果我们使用映射表，则所有指向这条记录的指针参考这个映射表，当我们移动或删除记录时，必须做的是改变表中这条记录对应的表项。

逻辑地址和物理地址的多种组合也是可能的，得到的是*结构化的地址模式*。例如，人们可使用块的物理地址（而不是块内的偏移量）加上被访问的记录的码值。那么，为找到一个具有这种结构地址的记录，我们可使用物理地址部分找到包含那条记录的块，然后检查块内记录以找到具有合适的码的记录。

当然，为检查块内记录，我们需要有足够多的信息对它们进行定位。最简单的情况是，记录是具有已知的固定长度的类型，而码字段具有已知的偏移量。那么我们只需在块首部找到块内记录的条数，并且我们精确地知道哪里能找到可能与作为地址一部分的码值匹配的码字段。但是还有许多对块进行组织的其他方法可以使我们能检查块内记录，稍后我们将讨论这些方法。

一个相似且非常有用的物理和逻辑地址的组合是在每一个块内存储一个*偏移量表*，它保存块内记录的偏移量，如图12-7所示。注意表是从块的前端向后增长，而记录是从块的后端开始

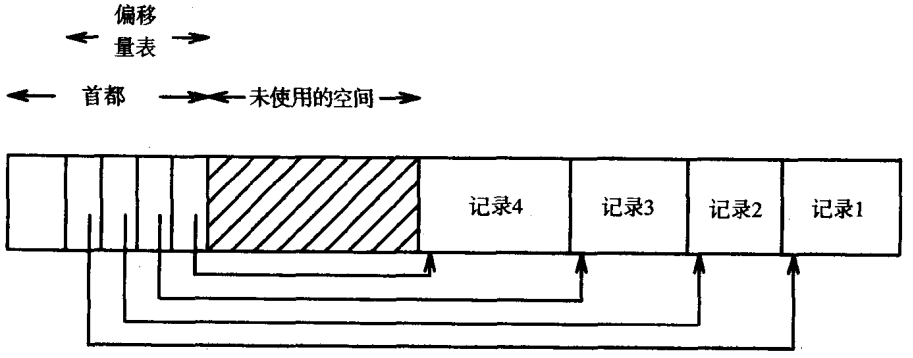


图12-7 一个有偏移量表的块，该表告诉我们块内每一条记录的位置

580 放置。当记录不等长时，这种策略很有用，因为我们事先不知道块能存储多少记录，而且我们不必一开始就给这个表分配固定大小的块首部。

现在，记录的地址是块的物理地址加上该记录在此块的偏移量表项中的偏移量。这种块内间接层次提供了逻辑地址的许多长处，而不需要一个全局映射表。

- 可以在块内移动记录，我们所要做的只是改变记录在偏移量表中的表项；指向这条记录的指针仍能找到它。
- 如果偏移量表项足够大，能存储这条记录的“转向地址”，我们甚至可以允许记录移到另一个块。
- 最后，如果记录被删除，我们可选择在它的偏移量表项中留下一删除标记，即一个标示记录被删除的特殊值。记录删除前，指向这条记录的指针可能已存储在数据库中的不同地方。记录删除后，沿指向这条记录的指针，找到删除标记，然后指针要么被一个空指针代替，要么修改数据结构以反映记录的删除。如果我们不留下删除标记，指针可能会指到一些新记录上，产生意外的错误结果。

12.3.3 指针混写

指针或地址经常是记录的一部分。尽管表示关系元组的记录不常遇到这种情况，表示对象的元组却常遇到这种情况。现代对象关系数据库系统也允许属性是指针类型（或引用），因此
581 即使是关系系统也要能在元组中表示指针。还有一点，索引结构由内部有指针的块组成。所以，必须研究块在主存储器和第二级存储器之间移动时的指针管理。我们在本节讨论这一点。

如前面所述，每一个块、记录、对象或其他可引用的数据项都有两种地址形式：

主存地址空间的属主

在本节中，我们讨论了一种在第二级存储器与主存储器之间的转换，其中每一个客户端拥有自己的主存地址空间，而数据库地址空间是共享的。这种模型在面向对象DBMS中是很普遍的。但是，关系系统经常将主存地址空间当成共享的，以支持我们将在第17章和第18章中讨论的恢复和并发。

一种有用的折中方法是在服务器端有一个共享的主存地址空间，而在客户端有此空间的一部分的拷贝。这种组织方法支持恢复和并发，同时也允许处理以“可扩展的”方式分布：客户端越多，能运用的处理器越多。

1. 它在服务器的数据库地址空间中的地址，通常是一个8字节左右的序列，指明数据项在系统的第二级存储器中的地址。我们把这种地址叫做*数据库地址*。

2. 虚拟内存中的地址（假如数据项正缓存在虚拟内存中），这些地址通常是4个字节。我们把这种地址称为数据项的*内存地址*。

当数据项位于第二级存储器中时，我们必然使用数据库地址。但是当数据项在主存中时，我们既能通过它的数据库地址，也能通过主存地址引用它。当数据项有指针时，使用内存地址更有效，因为可使用单一机器指令跟踪这样的指针。

相反，跟踪数据库地址要费时得多。我们需要一个表将目前在虚存中的所有数据库地址转换成它们的当前内存地址。这样的转换表如图12-8所示。它可能会使人联想到图12-6中逻辑和物理地址的映射表。但是，

a) 逻辑和物理地址都是数据库地址的表示，而转换表中内存地址用于相应对象在内存中的拷贝。

b) 数据库中所有可访问的数据项在映射表中都有表项，而转换表只记载当前在内存中的数据项。

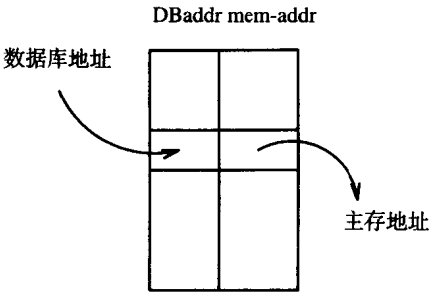


图12-8 转换表将数据库地址转换成内存中的相应地址

582

为避免将数据库地址重复转换成内存地址的开销，现已提出几种技术，统称为指针混写。总的思想是当我们把块从第二级存储器移到主存储器中时，块内指针可以“混写”，即从数据库地址空间转换为虚拟地址空间。因此，一个指针实际上包含：

1. 一个二进制位，指明指针目前是数据库地址还是（混写的）内存地址。

2. 数据库或内存指针，看哪个合适。无论当前使用哪一种地址形式，所用空间均相同。当然，若当前是内存地址，可能不使用整个空间，因为内存地址通常比数据库地址短。

例12.7 图12-9是一种简单情况，其中块1有一条记录，该条记录有两个指针，一个指针指向同一块中的第二条记录，另一个指针指向另一块中的记录。图中还表示了当块1被拷贝进内存中时，可能会发生的事情。第一个指针，即块1内的指针，可被混写以直接指向目标记录的内存地址。

但是，如果块2此时不在内存中，那么我们不能混写第二个指针；它必须不被混写，并指向目标的数据库地址。假如后来块2被放入内存，从理论上说混写块1的第二个指针就成为可能。根据使用的混写策略不同，内存中可能有也可能没有这样的指向块2的列表；如果有，我们可以选择在那个时刻混写指针。 □

583

我们可以使用几种策略来决定什么时候混写指针。

自动混写

块一旦被放入内存，我们就为它的所有指针和地址定位，并且如果这些指针和地址不在转换表中，我们将它们放入转换表。这些指针既包括来自块中记录并指向其他地方的指针，也包括块自身和/或其记录的地址，如果它们是可访问的数据项。我们需要一些为块内指针定位的机制。例如：

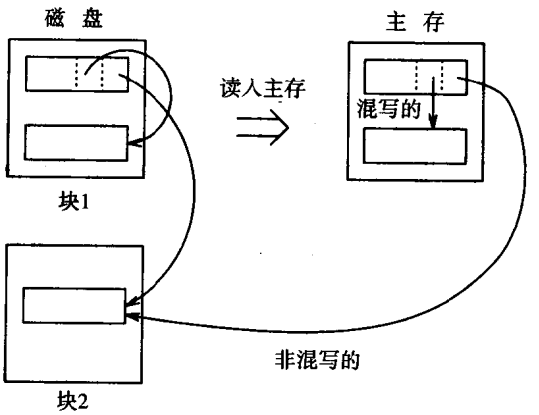


图12-9 当使用混写时一个指针的结构

1. 如果块存储的记录具有一个已知的模式，这个模式将告诉我们在记录的哪个地方可以找到指针。

2. 如果块被用于将在第13章讨论的索引结构的一种，则块在已知位置存储指针。

3. 我们可能在块首部存放指针位置的一个列表。

当我们将刚刚移入内存的块和/或某记录地址放入转换表中时，我们知道块缓存在内存的何处，从而可直接创建这些数据库地址的转换表表项。当我们向转换表添加其中一个数据库地址A时，可能会发现它已存在于表中，因为对应的块目前正在内存中。在这种情况下，我们用相应的内存地址代替刚移进内存中的块中的A，并将“混写”位设为真。另一方面，如果A还不在转换表中，说明它对应的块还未拷贝进内存，那么我们不能混写该指针，只能将它保留在
584 块中作为数据库指针。

如果我们试图跟踪块中的指针P，而且发现还未混写，即它具有数据库指针形式，那么我们要确保包含P指向的数据项的块B在内存中（否则为什么要跟踪那个指针？）。我们参考转换表，看数据库地址P目前是否有相应的内存地址，如果没有，我们将块B拷贝进内存缓冲区。一旦B在内存中，我们就可以通过用等价的内存形式代替它的数据库形式来混写P。

按需混写

另一种方法是当块第一次被移入内存时，所有指针都不混写。我们将它的地址以及其指针的地址与相应的内存地址一起放入转换表。当我们跟踪某个内存块中的指针P时，将它混写，使用的策略与使用自动混写策略时发现一个没有混写的指针所使用的策略相同。

按需混写与自动混写的区别是当块被装载进内存时，后者试图快速、有效地混写所有指针。我们必须在一次混写所有指针可能节省的时间与某些混写指针永远也用不上的可能性之间进行权衡。在那种情况下，花费在混写和不混写指针的时间将被浪费。

一个有趣的选择是通过安排使数据库指针看起来像无效的内存指针。这样，我们可以允许计算机跟踪任何指针，就像它是内存形式一样。如果指针凑巧没有被混写，那么内存引用必然产生一个硬件陷阱。如果DBMS提供一个函数，它由该陷阱触发，并用上面描述的方法对指针混写，则我们可在单一指令中跟踪混写的指针，并只有当指针未被混写时才需做一些较费时的工作。

不混写

当然，永远不混写指针也是可能的。我们仍然需要转换表，以使指针可以以它们未混写的形式被跟踪。这种方法确实提供了记录不被固定在内存的好处，如12.3.5节所讨论的那样，而且不需要决定当前使用的是哪种形式的指针。

混写的程序控制

在某些应用中，应用程序员可能会知道块中的指针是否被跟踪。该程序员可显式地指明装载进内存的块中指针将被混写，也可以只在需要时请求对指针进行混写。例如，如果程序员知
585 道一个块可能被大量存取，如B树（13.3节讨论）中的根块，那么指针将被混写。但是，若被装载进内存中的块只使用一次，然后就从内存中删除，则它不被混写。

12.3.4 块返回磁盘

当块被从主存移回到磁盘中时，块中的任何指针必须解除混写；即它们的内存地址必须由相应的数据库地址取代。转换表可用于将两种类型的指针进行双向联系，因此从原理上说，给定一个内存地址，可以找到与其对应的数据库地址。

但是，我们并不想每一次解除混写时均需搜索整个转换表。虽然我们还未讨论转换表的实现，但是我们可以想像图12-8所示的表有合适的索引。如果将转换表想像为一个关系，则寻找与数据库地址x相联系的内存地址的问题，可表述为如下的查询：

```
SELECT memAddr
```

```
FROM TranslationTable
WHERE dbAddr = x;
```

例如，使用数据库地址作为码的散列表可能适用于在dbAddr属性上建立的索引；第13章提出了许多可以采用的数据结构。

如果我们想支持反向查询：

```
SELECT dbAddr
FROM TranslationTable
WHERE memAddr = y;
```

则我们还需要在属性memAddr上建立索引。第13章也提出适用于这种索引的数据结构。12.3.5节还讨论了链表结构，在某些情况下，它可用于从一个内存地址到达所有指向它的指针。

12.3.5 被固定的记录和块

如果内存中一个块当前不能安全地被写回磁盘，则称它为被固定的。块首部可有一个指明块是否被固定的二进制位。块可能被固定的原因有很多，包括系统的恢复需要，如第17章所讨论的那样。指针混写是为什么某些块必须被固定的一个重要原因。

如果一个块 B_1 内有一个混写指针，指向块 B_2 中的某一数据项，那么在将块 B_2 移回磁盘并重用它的主存缓冲区时，我们需非常小心。原因是，假如我们跟踪 B_1 中的指针，它将把我们 586
指引到缓冲区，但缓冲区中已不保存 B_2；其实，指针已成为悬挂指针。因此一个像 B_2 那样被其他地方的混写指针引用的块是固定。

当我们将块写回磁盘时，不仅需要“解除混写”该块中的所有指针，而且需要保证它没有被固定。如果它被固定，我们必须要么使它不被固定，要么让它继续留在内存中，占用本可用于其他块的空间。为了一个由于存在外部混写指针而被固定的块不再被固定，我们必须“解除混写”指向它的所有指针。因此，对每一个有数据项在内存中的数据库地址，转换表必须记录指向那个数据项的混写指针内存中的位置。两种可采用的方法是：

1. 将对一个内存地址的引用列表保存为与在转换表中该地址的表项相关的链表。

2. 如果内存地址比数据库地址短得多，我们可以在指针自身空间中创建链表。即每一个用于数据库指针的空间被替换为：

   a) 被混写的指针；和

   b) 另一个指针，它是被混写的指针每一次出现所构成的链表的一部分。

图12-10表示了从转换表中数据库地址的表项x和它相应的内存地址y出发，如何将内存指针y的每一次出现链接起来。

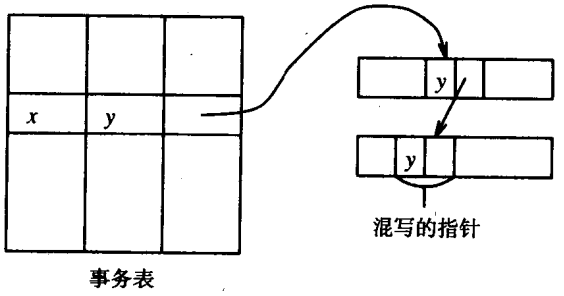


图12-10 一个混写指针出现的链表

12.3.6 习题

587

* **习题12.3.1** 如果我们为Megatron 747磁盘表示物理地址，给每一个柱面，柱面内每一个磁道，磁道内的每一个块各分配一个字节或多个字节，则需要多少字节？对每一个磁道内块的最大数做一个合理的假设。回想一下，Megatron 747的扇面/磁道数为一个可变的数目。

习题12.3.2 为习题11.3.1中描述的Megatron 777磁盘，重做习题12.3.1。

* **习题12.3.3** 如果我们希望既表示块地址，又表示记录地址，则需要额外的字节。假设我们需要如习题12.3.1所示的一个Megatron 747磁盘的地址，如果

* a) 将块内字节数作为物理地址的一部分；

b) 使用记录的结构地址，假设被存储的记录有4字节整数作为码；

则我们需要多少字节表示记录地址？

习题12.3.4 现在，IP地址有4个字节，假设一个全球范围的地址系统中的块地址由主机IP地址、1～1000之间的设备号以及各个设备上的块地址（假设为Megatron 747磁盘）组成。块地址需要多少字节？

习题12.3.5 将来，IP地址将使用16个字节。另外，我们可能不仅需要访问块，还需要访问记录，而记录可能在块内任何字节处开始。但是，设备将有它们自己的IP地址，因此不需要在主机内表示设备，而在习题12.3.4中我们认为这是必要的。仍假设设备为Megatron 747磁盘，在这种情况下，地址的表示需要多少字节？

! **习题12.3.6** 假设我们希望逻辑地表示Megatron 747磁盘上的块地址，即使用k字节（k为某个值）的标识符。我们还需要在磁盘上存储一个映射表，如图12-6所示的那样，它由逻辑地址和物理地址对组成。用于映射表的块不是数据库的一部分，因此在映射表中没有这些块的逻辑地址。假设物理地址使用物理地址所能使用的最少字节数（如习题12.3.1计算的那样），逻辑地址使用逻辑地址所能使用的最小字节数，磁盘的映射表占用多少个4096字节的块？

*! **习题12.3.7** 假设我们有4096字节块，块中存储100字节长的记录。块首部由一个偏移量表组成，如图12-7所示，它使用2字节长的指针指向块内记录。通常，每天向每一块插入两

588 条记录，删除一条记录。删除记录必须使用一个“删除标记”代替它的指针，因为可能会有悬挂指针指向它。更明确地说，假设任何一天删除记录总发生在插入之前。如果刚开始时块是空的，多少天之后，不再有插入记录的空间？

* **习题12.3.8** 假设每天平均插入1.1条记录，删除一条记录，重做习题12.3.7。

习题12.3.9 假设不删除记录，而是将它们移到另一个块，且必须在它们的偏移量表的表项中提供一个8字节的“转向”地址，若

! a) 给所有的偏移量表的表项分配一个表项所需要的最大字节数。

!! b) 允许偏移量表的表项长度发生变化，但可正确地找到和解释所有的表项。

分别重做习题12.3.7。

* **习题12.3.10** 假设我们自动混写所有指针，所用的总时间是单独混写每一个指针所用总时间的一半。如果主存中一个指针被至少跟踪一次的概率为 p，p为何值时自动混写比按需混写更有效？

! **习题12.3.11** 对习题12.3.10进行推广，将从不混写指针的可能性包括进来。假设几个重要动作占用以下时间(以某个时间单位计)：

a) 指针按需混写：30。

b) 指针自动混写：20/每个指针。

c) 跟踪一个混写指针：1。

d) 跟踪一个未混写指针：10。

假设内存中的指针要么不被跟踪（概率为$1-p$），要么被跟踪k次（概率为p），k和p为何值时，不混写、自动混写和按需混写各自都能提供最好的平均性能？

12.4 变长数据和记录

到目前为止，我们一直简单地假设每个数据项有固定长度，记录有固定模式，且模式是定长字段的列表。但是，实际情况很少会这么简单。我们可能希望表示： 589

1. 大小变化的数据项。例如，在图12-1中，我们考虑了一个关系MovieStar，它的地址字段最大可为255字节。虽然有些地址可能有那么长，但绝大多数地址可能为50个字节甚至更少。如果我们只为地址分配它所需的实际空间，则可能节省存储MovieStar元组所用空间的一半以上。

2. 重复字段。如果我们试图在表示对象的记录中描述多对多关系，就必须存储与给定对象所关联的所有对象的引用。

3. 可变格式记录。有时，我们事先不知道记录的字段是什么，或每一个字段出现多少次。例如，一些电影明星也执导影片，我们可能想给他们的记录添加字段，指明他们执导的电影。同样，一些影星出品电影，或以其他方式参加，而我们也可能希望把这样的信息放入他们的记录。但是，因为大多数影星既不是出品人也不是导演，我们不想在每个影星的记录中都为这种信息预留空间。

4. 极大的字段。现代DBMS支持属性值为非常大的数据项的属性。例如，我们可能想在电影明星记录中定义一个picture属性，它是明星的一幅GIF图像。一个电影记录可能有一个字段，它是电影的MPEG编码，大小为 2GB，还有更多的普通字段，如电影标题。这些字段如此之大，以致与我们的记录能存储在一个块内的直觉相矛盾。

12.4.1 具有变长字段的记录

如果记录的一个或多个字段是变长的，则记录必须包含足够多的信息以便我们能找到记录的任何字段。一个简单而有效的模式是将所有定长字段放在变长字段之前。我们在记录首部写入以下信息：

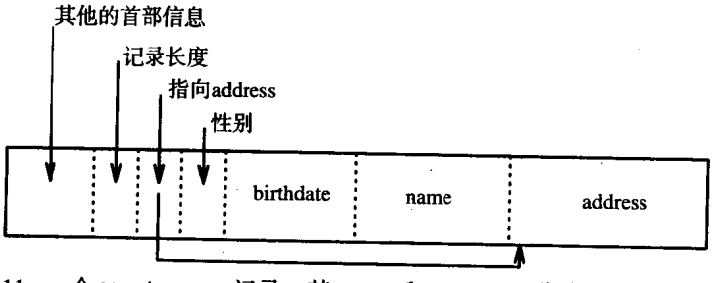


图12-11 一个 MovieStar 记录，其 name 和 address 作为变长字符串实现

1. 记录长度。

2. 指向所有变长字段起始处（即偏移量）的指针，然而，如果变长字段总是以同样的顺序出现，则第一个变长字段不需要指针；我们知道它就紧跟在定长字段之后。

例12.8 假设我们有电影明星的记录，其字段为姓名、住址、性别和出生日期。假设性别

和出生日期为定长字段，各占4和12个字节。但是，姓名和住址将由具有任意合适长度的字符
590 串表示。从图12-11可以看到一个典型的影星记录。我们总是将姓名放在住址前面，因此不需要有指针指向姓名的起始处；那个字段总是紧跟在记录的定长部分之后。 □

12.4.2 具有重复字段的记录

如果定长字段F出现的次数可变，类似的情况也会发生。将字段F的每次出现放在一起，在记录首部放一个指针，让它指向字段F出现的第一个位置，这就足够了。我们可用以下方法找到字段F出现的所有位置。令字段F的一次出现占用的字节数为L，然后在字段F的偏移量上加上L的所有整数倍数，从0开始而后L、$2L$、$3L$，依此类推。最后，我们到达F后面的字段的偏移量，至此停止。

例12.9 假设我们重新设计电影明星记录，只存储姓名、地址（为变长字符串）和指向明星主演的影片的指针。图12-12给出了这种记录类型是如何被表示的。首部包含指向地址字段
591 起始处的指针(我们假设姓名字段总是恰好在首部之后开始)和指向第一个电影指针的指针。记录长度告诉我们有多少电影指针。 □

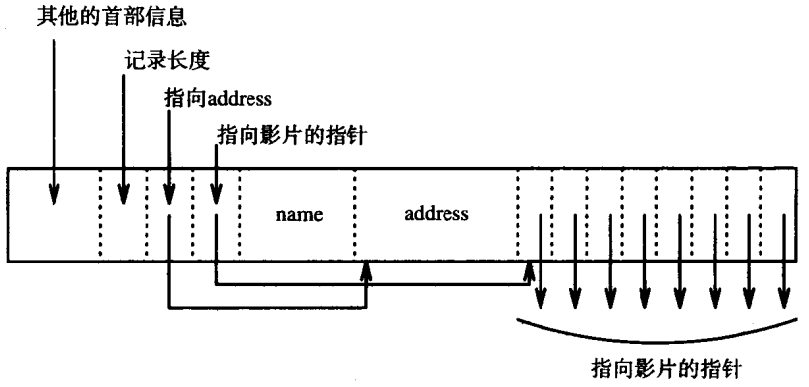


图12-12 具有对电影的一组重复引用的记录

另一种表示方法是保持记录定长，而将变长部分（无论它是变长字段，还是重复次数不确定的字段）放在另一个块上。在记录本身中我们存储

1. 指向每一个重复字段开始处的指针；和
2. 重复次数或者重复结束处。

图12-13表示例12.9中的问题的记录格式，但其变长字段`name`和`address`以及重复字段`starredIn`（影片引用集）存储在另外的一个或多个块上。

为记录的变长部分使用间接既有好处，也有缺点：

- 保持记录定长。可以更有效地对记录进行搜索，使块首部的开销最少，记录能很容易地在块内或块间移动。
- 另一方面，将变长部分存储在另一个块中增加了为检查一条记录的所有部分而进行的磁盘I/O数目。

一种折中方案是在记录的定长部分保留足够的空间，以存储以下信息：

1. 重复字段合理的出现次数。
2. 指向可以找到这个重复字段其他出现的地方的指针。
3. 其他出现的次数。

如果所需空间小于预留空间，则一些空间可能会无用。如果信息不能放入定长部分，则指向附加空间的指针将是非空的，且我们能通过跟踪这个指针找到重复字段另外的出现。 592

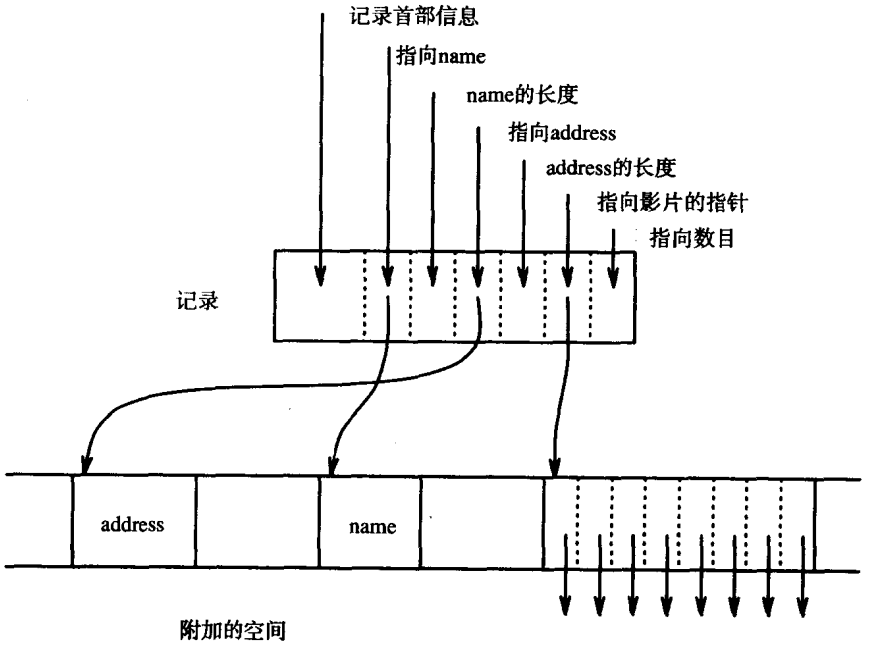


图12-13 将变长字段与记录分开存储

空值的表示

元组经常有可为NULL的字段。图12-11所示的记录格式提供了一种通常使用的表示NULL值的方法。如果像address这样一个字段为空，则我们在指向一个地址的指针空间处放一个空指针。这样，除地址指针外，我们不需要为地址分配空间。这种方式平均来说能节省空间，甚至在address为定长字段但经常为NULL值时也如此。

12.4.3 可变格式记录

如果记录没有固定的模式，情况会更复杂。所谓记录没有固定的模式，指字段或字段顺序不是完全由记录所表示元组或对象的关系或类决定。变格式记录最简单的表示是标记字段序列，每一个标记字段包含以下内容：

1. 关于这个字段角色的信息，如：
   (a) 属性或字段名；
   (b) 字段类型，如果它不能从字段名和一些可用的模式信息中明显推知，以及
   (c) 字段长度，如果它不能从类型明显推知。
2. 字段值。

至少有两个原因可以说明为什么标记字段有用：

1. *信息集成应用*。有时，一个关系是根据以前的几个数据源创建的，而这些数据源有不同种类的信息；参见20.1节中的讨论。例如，影片明星信息可来自几个数据源，一个数据源记载出生日期，而其他几个不记载；一些数据源提供地址，而其他不提供，等等。如果字段数不是很多，我们最好给未知字段值赋NULL。但是如果数据源很多，信息种类很多，则可能会有太 593

多的NULL。然而，如果我们使用标识，只列出非空字段，则可节省相当大的空间。

2. *具有非常灵活的模式的记录*。一条记录的许多字段会重复或根本不出现，那么即使我们知道模式，标识字段也可能是有用的。例如，医疗记录可能包含许多有关检验的信息，但是可做的检验数以千计，而每一个病人只有较少的检验结果。

例12.10 假设一些电影明星有诸如执导的影片、前配偶、所拥有的餐馆和许多其他的固定但不常用的信息。在图12-14中，我们看到一个使用标记字段的假想的电影明星记录的开头。我们假设各种可能的名称和类型都使用单字节编码。图12-14指出了所显示的两个字段的长度及其合适的编码，这两个字段凑巧都是字符串类型。 □

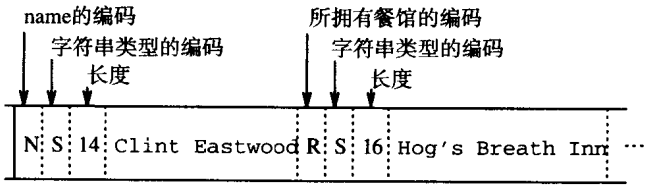


图12-14 一个具有标记字段的记录

12.4.4 不能装入一个块中的记录

下面，我们将讨论另一个问题。这个问题的重要性越来越大，因为DBMS更经常地用于管理大值数据类型，这是一些通常不能装入一个块中的值。典型的例子是视频或音频“片段”。这些大值经常是变长的，但即使这种类型的的属性值是定长的，我们也需要使用一些特殊技术来表示它们。本节中我们将考虑一种称做“跨块记录”的技术，它用于管理比块大的记录。对非常大的值（兆字节或吉字节）的管理在12.4.5节讨论。

当记录比块小，但是将整条记录装入块中将浪费大量空间时，跨块记录也是有用的。例如，例12.6中，空间仅浪费7%。但如果记录只是比块的一半稍大，则浪费率接近50%，原因是这时
594 我们在一个块中只能装一条记录。

因为这两个原因，我们有时希望将*记录*分开，存储在两个或多个块中。出现在一个块中的记录的一部分被称为*记录片段*。一个具有两个或多个片段的记录被称为是*跨块的*，而不跨越块边界的记录是*不跨块的*。

如果记录能跨块，则每一条记录和记录片段需要一些另外的首部信息：

1. 每一条记录或片段首部必须包含一个二进制位，指明它是否为一个片段。

2. 如果它是一个片段，则它需要几个二进制位，指明它是否为它所属的记录的第一个或最后一个片段。

3. 如果对同一条记录有下一个和/或前一个片段，则片段需要指向这样一些其他片段的指针。

例12.11 图12-15表示如何将3个大约为块的60%大小的记录存储在两个块中。记录片段2a

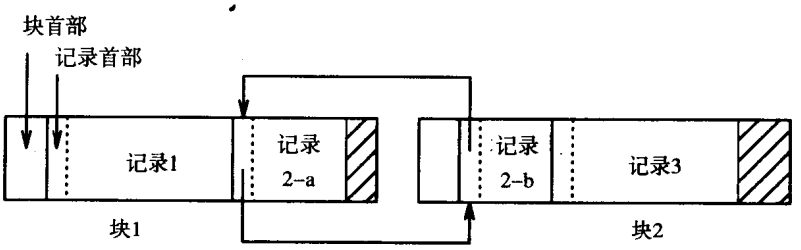


图12-15 跨多个块存储跨块记录

的首部包含一个指明它是片段的标记，一个指明它是记录的第一个片段的标记和一个指向下一个片段2*b*的指针。同样，2*b*首部指明它是记录的最后一个片段，且有一个指向前一个片段2*a*的反向指针。 □

12.4.5 BLOBS

现在，我们考虑真正大的记录值或记录字段值的表示。常见的例子包括各种格式的图像（如GIF或JEG），格式为MPEG等的电影，或各种信号：声音、雷达等。这样的值经常被叫做*二进制大对象*或BLOB。当一个字段的值为BLOB时，我们必须重新考虑至少两件事情。 595

BLOB的存储

BLOB必须存储在一系列块中，我们总是希望这些块在磁盘的一个或多个柱面上顺序分配，这样就能有效地检索BLOB。但是也有可能将BLOB存储在块的链表中。

另外，可能需要对BLOB进行很快地检索（如必须实时播放一部电影），以致如将其存储在一个磁盘上会使我们不能对其进行足够快的检索。那么，有必要将BLOB进行分割，存储在几个磁盘中，即在这些磁盘上交替存储BLOB的块。这样就可以同时检索BLOB的几个块，检索效率提高的倍数大约等于参与分割的磁盘数。

BLOB的检索

我们假设当客户端需要记录时，包含那条记录的块全部从数据库服务器传到客户端，这一假设可能不再成立。我们可能只想传送记录的“小”字段，同时允许客户端一次一个地请求BLOB的块，而与记录的其余部分无关。例如，如果BLOB是一部两小时的电影且客户端请求播放这部影片，那么可以一次向客户端传送电影的几块，其速率正好是播放电影必须的速率。

在许多应用中，客户端能请求BLOB内的部分，而不必接收整个BLOB，这也很重要。例如，有一个请求要观看一部电影的第45分钟，或一个音频片段的结束部分。如果DBMS要支持这些操作，那么它需要合适的索引结构，例如，在一个电影BLOB上通过秒进行索引。

12.4.6 习题

* **习题12.4.1** 一个病人记录包含以下定长字段：病人的出生日期、社会保险号码和病人ID，每一个字段都是10字节长。它还有下列变长字段：姓名、住址和病史。如果记录内一个指针需要4个字节，记录长度是一个4字节整数，不包括变长字段空间，这条记录需要多少字节？可以假设不需要对字段进行对齐。

* **习题12.4.2** 假设使用习题12.4.1的记录。变长字段姓名、住址和病史的长度都符合均匀分布。对姓名来说，其范围为10～50个字节；对住址来说，其范围是20～80个字节；对病史来说，范围是0～1000字节。一个病人记录的平均长度是多少？

习题12.4.3 假设在习题12.4.1的病人记录上添加另外的可重复字段，表示胆固醇化验，每一次胆固醇化验需要一个16字节的日期和化验的整数结果。如果 596

a) 重复化验保存在记录中。

b) 化验存储在另外一个块中，记录中存储指向化验的指针。

分别指出病人记录的格式。

习题12.4.4 假设在习题12.4.1的病人记录上，我们添加用于检验及检验结果的字段。每个检验包括检验名、日期和检验结果。假设每一个这样的检验需要40个字节。还假设对每一个病人和每一次检验，存储检验结果的概率为 p。

a) 假设指针和整数都需要4个字节，所有检验结果都作为变长字段存储在记录内，在病人记录中，检验结果平均需要多少字节？

b) 如果检验结果字段存储在其他地方，而记录内有指向检验结果字段的指针表示检验结果，重作(a)。

! c) 假设我们使用混合模式，k次检验结果存储在记录内，另外的检验结果存储在另一块（或块链中）中。这些另外的检验结果可通过跟踪指向存储它们的块（或块链）的指针而被找到。作为p的函数，当k为何值时，用于存储检验结果的空间最小？

!! d) 重复的检验结果字段所用的空间总量不是惟一问题。假设我们想最小化的一个重要指标是所使用的字节数。如果我们必须将一些结果存储在另一块中，从而对许多我们必须进行的检验结果的存取需要一次磁盘I/O，则加上10 000的损失。在这种假设下，k作为p的函数，其最佳值是多少？

*!! **习题12.4.5** 假设块有1000个字节可用于存储记录，我们希望在块上存储长度为r的定长记录，其中$500 < r \leqslant 1000$。r的值包括记录首部，但是一个记录片段需要另外的16个字节用做片段首部。r为何值时，我们能通过跨块记录提高空间使用率？

习题12.4.6 回想一下例11.5。一部MPEG影片每小时的播放大约使用1吉字节。如果我们能以最好的方式在一个Megatron 747磁盘上安排MPEG影片的块，且磁盘延时极短（100毫秒），从一个磁盘能传送几部影片？

597

12.5 记录的修改

记录的插入、删除和更新经常产生某些特别的问题。尽管当记录改变长度时，这些问题尤为严重，但即使记录和字段都是定长的，也会出现这些问题。

12.5.1 插入

首先，我们考虑将一条新记录插入到一个关系中（或等价地插入到类的当前外延中）。如果关系的记录没有特定的存储顺序，则我们只需找到一些有空闲空间的块，或当块没有空闲空间时就找一个新块，然后将记录放在那里。通常有一些用以找到存储某个给定关系的元组或类对象的所有块的机制，但我们将如何找到这些块的问题放到13.1节讨论。

若元组必须以某个固定次序存储，如按主键顺序存储，则会有更多的问题。保持记录有序性是有充分原因的，因为它便于回答某些查询，正如我们将在13.1节中要看到的那样。如果我们要插入一个新记录，首先要找到那条记录应放置的块。此块可能凑巧有空间存放这条新记录。因为记录必须保持有序，我们可能不得不在块中滑动记录以在合适的地点得到所需的空间。

如果需要滑动记录，则图12-7所示的块组织是很有用的，这里我们把它再现在图12-16中。回想在12.3.2节中的讨论，我们可以在每一个块的首部创建一个“偏移量表”，其中指针指向块中每一记录的位置。从块的外部指向记录的指针是一个“结构地址”，即块地址和偏移量表中该记录表项的位置。

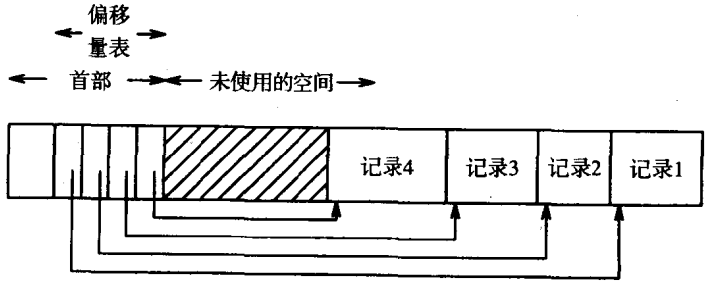


图12-16 偏移量表让我们在块中滑动记录以便为新记录腾出空间

如果能在当前块中为插入记录找到空间，则我们只是简单地在块中滑动记录，调整偏移量表中的表项。新记录被插入到块中，在此块的偏移量表中添加指向此记录的新指针。 598

但是，块中可能没有空间用于新记录的存储，在这种情况下，我们不得不在此块之外寻找到空间。解决这个问题主要有两种方法，这两种方法也可以结合使用。

1. 在“邻近块”中找空间。例如，若块B_1没有空间供需要以有序方式插入该块中的记录所用，则在块的有序排列中找到下一个块B_2。如果B_2有空间，将B_1的最高记录移到B_2，并移动两块中的记录。但是，如果有外部指针指向记录，则我们必须小心，要在B_1的偏移量表中留下一个转发地址（forwarding address），说明某一记录被移到B_2以及它的表项在B_2的偏移量表中哪个地方。允许使用转发地址通常会增加偏移量表表项所需空间的总量。

2. 创建一个溢出块。在这种模式下，每个块B的首部有一个位置，这个位置存放一个指向溢出块的指针，理论上属于B的多余的记录放入溢出块中。B的溢出块可以指向第二个溢出块，依此类推。图12-17指明了这种情况。我们将溢出块的指针表示为块上的小块，尽管实际上它是块首部的一部分。

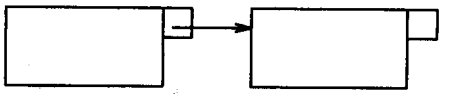


图12-17 块及其第一个溢出块

12.5.2 删除

在删除记录时，可以回收记录空间。如果使用图12-16所示的偏移量表，且记录可以在块内滑动，则我们能让块中空间紧凑，以使块中间总有一个未用的区域，如该图所示。

如不能滑动记录，则需要在块首部维护一个可用空间列表。当向块中插入一条新记录时，我们知道可用区域在哪里，它有多大。注意，块首部通常不必存储全部可用空间列表，只把链头放于块首部就足够了，然后使用可用区域自身存储列表中的链接，与图12-10很相似。

当删除记录时，我们或许能够去除溢出块。如果记录从块B中或从溢出链的任一块中删除，我们可以考虑那条块链的所有块中已用空间总量。如果记录能被较少的块容纳，且能安全地在 599
被链接的块之间移动记录，则可对整个链进行重新组织。

但是在删除记录时，还有另外一个复杂的因素，无论使用何种模式重新组织块，我们都必须记住它。如果有指向被删除记录的指针，而我们不想让这些指针成为悬挂指针或指向放于被删除记录地址处的新记录，通常采用的方法是在记录处放一个删除标志，我们在12.3.2节已指出了这种方法。这个删除标记是永久的；它必须一直保留，直到对整个数据库进行重构。

删除标记放在何处依赖于记录指针的特征。如果指针都指向固定的位置，在这些固定位置可以找到记录的位置，则我们可将删除标记放在该固定位置。这里是两个例子：

1. 在12.3.2节我们指出，如果使用图12-16的偏移量表模式，则删除标记可以是偏移量表中的空指针，因为指向记录的指针实际上是指向偏移量表表项的指针。

2. 如果使用图12-6所示的映射表将逻辑地址转换为物理地址，则删除标记可以是在物理地址处的空指针。

如果我们需要用删除标记代替记录，明智的做法是将记录首部起始处的一个二进制位用作删除标记；即如果记录没被删除，则这个二进制位为0；若它为1，则意味着记录已被删除。只有记录起始处的这个二进制位必须保留，而以后的字节可为另一条记录重用，如图12-18所示 ⊖。

⊖ 但是，12.2.1节讨论的字段对齐问题可能迫使我们留下4个或更多字节不用。

当我们跟踪指针到一个被删除的记录时，我们明白的第一件事是“删除标记”位告诉我们记录已被删除，然后我们就知道不要再看下面的字节了。

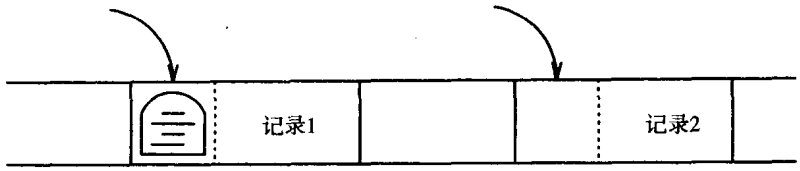


图12-18 记录1可被替代，但是删除标记保留；记录2没有删除标记，可以通过跟踪指向它的指针而看到它

600

12.5.3 更新

当一个定长记录被更新(update)时，对存储系统没有影响，因为我们知道它占用与更新前完全相同的空间。但是当一个变长记录被修改时，我们就会碰到与插入和删除有关的所有问题，只不过永远都不需要为记录的旧版本创建删除标记。

如果更新后的记录比其旧版本长，则我们可能需要在它所在的块中创建更多的空间。这个过程可能涉及记录的滑动，有时甚至要创建一个溢出块。如果记录的变长部分存储在另一个块中，如图12-13所示，则我们可能需要在那个块中移动元素，或创建一个新块用于存储变长字段。反过来，如果记录由于更新而变短，我们可以像删除记录时那样恢复、合并空间或去除溢出块。

12.5.4 习题

习题12.5.1 假设我们将记录块通过它们的排序键字段排序，并在块之间按序分割。每个块有一个可从外部获知的排序键范围（13.1.3节中的稀疏索引结构正是这种情况的一个例子）。没有从外部指向记录的指针，因此当我们需要时可在块间移动记录。用来处理插入和删除的一些方法如下：

1) 只要有溢出则分割块。当我们分割块时，调整块的排序键范围。
2) 使块的排序键范围固定，而在必要时使用溢出块。为每块和每个溢出块存储一个偏移量表，表项只包含该块中的记录。
3) 与(2)相同，但是在第一个块中（或偏移量表需要空间时在溢出块中），为块及其所有溢出块存储偏移量表。注意，如果偏移量表需要更多空间，我们可将记录从第一个块移到溢出块以腾出空间。
4) 同(2)，但偏移量表中存储排序键和指针。
5) 同(3)，但偏移量表中存储排序键和指针。

回答下列问题：

* a) 比较方法(1)和方法(2)在可能包含给定排序键的记录块（或有溢出块的块链的第一个块）被找到后，为检索记录而进行的磁盘I/O的平均数目。具有较少平均磁盘I/O的方法有什么缺点吗？

601

b) b为块链中块的数目，每一条记录检索所需的磁盘I/O的平均数目为b的函数。就这个平均数目，比较方法(2)和(3)。假设偏移量表占空间的10%，记录占其余90%。

! c) 在(b)部分的比较中，将方法(4)和(5)包括进去。假设排序键占记录的1/9。注意，如果排序键在偏移量表中，我们就不必在记录中重复存储它。这样，偏移量表实际上使用空间的20%，记录的其余部分使用空间的80%。

习题12.5.2 关系数据库系统总是倾向于尽可能使用定长元组，给出这种优先考虑的三

种理由。

12.6 小结

- 字段：字段是最基本的数据元素。在第二级存储器中，许多字段如整数或定长字符串被简单地分配予适当的字节数。变长字符串的编码要么使用包括一个结束标记的固定长度的字节序列，要么存储在变长串的区域中，其长度用记录开始处的一个整数或结束处的结束标记表明。
- 记录：记录由几个字段再加上一个记录首部组成。首部包括有关记录的信息，可能包括像时间戳、模式信息和记录长度这样的信息。
- 变长记录：如果记录包含一个或多个变长字段，或包含一个重复次数未知的，则需要附加的结构。记录首部的指针目录可用于定位记录内的变长字段。也可用（定长）指针来代替变长或重复字段，这个指针指向记录外部存储记录值的地方。
- 块：记录通常存储在块内。块首部是有关块的信息，占用块中的一些空间，其余空间由一条或多条记录占用。
- 跨块记录：通常，一条记录存储在一个块中。但是，如果记录比块大，或我们希望利用块的剩余空间，则可将记录分成两个或多个片段，不同片段存储在不同块上。这时就需要用片段首部将一条记录的片段链接起来。 602
- BLOBS：非常大的值（如图像和视频）被称做BLOBS（二进制大对象）。这些值必须跨多个块存储。根据存取需要，可能将BLOB存储在一个柱面上以减少BLOB的存取时间，或者有必要将BLOB分割，跨多个磁盘存储，以允许对其内容进行并行检索。
- 偏移量表：为支持记录的插入和删除，以及由于修改记录中的变长字段而引起的记录长度改变，我们可在记录首部放一个偏移量表，其中包含指向块中每一记录的指针。
- 溢出块：也用于支持记录插入和记录长度增加。块可以链接到溢出块或块链，溢出块或块链中存储一些逻辑上属于第一个块的记录。
- 数据库地址：由DBMS管理的数据存在于几个存储设备上，通常为磁盘上。为在存储系统中定位块和记录，我们可使用物理地址，它描述设备号、柱面、磁道和扇区，还可能包括扇区内字节。我们也可使用逻辑地址，它是任意的字符串，由映射表转换成物理地址。
- 结构地址：我们也可通过使用部分物理地址（如记录所在块的位置）加上另外的信息（如记录的码或块的偏移量表中用于定位该记录的位置）来定位记录。
- 指针混写：当磁盘块被放入主存时，如果要跟踪指针，则数据库地址需要转换成内存地址。这种转换被称为混写，且可于块被放入内存时自动进行，或当块第一次被跟踪时按需混写。
- 删除标记：当一条记录被删除时，可能会使指向它的指针成为悬挂指针。替代（部分）被删除记录的删除标记警告系统记录已不在那里。
- 被固定的块：因为各种原因，例如块可能包含被混写的指针，把一个块从内存拷贝回它所在磁盘的位置中也许是不能被接受的。这样的块称为被固定的块。如果块被固定是由于混写指针，则在将块返回磁盘之前，必须为这些指针解除混写。

12.7 参考文献

1968年撰写的有关数据结构的经典文章[2]最近已被修改。[4]提供了与本章和第13章有关 603

的关于结构的信息。

删除标记作为处理删除的技巧，来自[3]。[1]讨论了数据表示问题，如面向对象DBMS环境中的地址和混写。

1. R. G. G. Cattell, *Object Data Management*, Addison-Wesley, Reading MA. 1994.

2. D. E. Knuth, *The Art of Computer Programming, Vol. I, Fundamental Algorithms, Third Edition*, Addison-Wesley, Reading MA, 1997.

3. D. Lomet, "Scheme for invalidating free references," *IBM J. Research and Development* **19**:1 (1975), pp, 26–35.

4. G. Wiederhold, *File Organization for Database Design*, McGraw-Hill, New York, 1987.

604

第13章 索引结构

前面介绍了表示记录的几种可选方式，现在我们需要考虑整个关系或类外延如何表示。仅仅把关系中的元组或类外延中的对象随机分散到各存储块中是不够的。为了说明这一点，我们来看一看怎样回答哪怕是像 SELECT * FROM R这样最简单的查询。我们将不得不检索存储器中的每个存储块，并期望块首部中存在足够的信息来标明该块中记录从什么地方开始，并且记录首部中存在足够的信息来说明该记录属于哪个关系。

一个稍好的记录组织方式是为给定的关系预留一些块甚至几个完整的柱面。我们可以认为这些柱面上所有存储块中都存放表示该关系中元组的记录。现在，我们至少不需要扫描整个存储器就能找到该关系的所有元组。

然而，这种组织方式却无助于回答下面这个简单查询："找出给定主键值的元组"。6.6.6节介绍了在关系中建立索引的重要性如SELECT * FROM R WHERE a = 10。建立索引的目的是为了加快关系中那些在某个特定属性上存在特定值的元组的查找速度。如图13-1所示，索引是这样一种数据结构：它以记录的特征（通常是一个或多个字段的值）为输入，并能"快速地"找出具有该特征的记录。具体来说，索引使我们只需查看所有可能记录中的一小部分就能找到所需记录。建立索引的字段（组合）称为*查找键*，在索引不言而喻时也可称"键"。

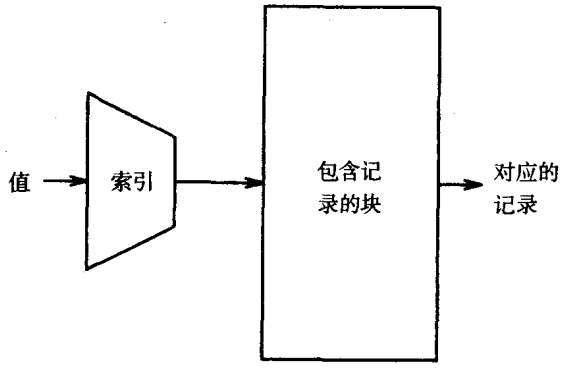


图13-1 索引以某个（或某些）字段值为输入并找出对应字段值符合要求的记录

目前已有多种不同数据结构可用做索引。在本章剩下的内容中，我们考虑下面几种设计和实现索引的方法：

1. 排序文件上的简单索引。
2. 非排序文件上的辅助索引。
3. B树，一种可在任何文件上建立索引的常用方法。
4. 散列表，另一种有用而重要的索引结构。

13.1 顺序文件上的索引

研究索引结构，我们首先来考虑最简单的一种：由一个称为*数据文件*的排序文件得到另一

个称为索引文件的文件，而这个索引文件由键-指针对组成。在索引文件中查找键K通过指针指向数据文件中查找键为 K 的记录。索引可以是“稠密的”，即数据文件中每个记录在索引文件中都设有一个索引项；索引也可以是“稀疏的”，即数据文件中只有某些记录在索引文件中表示出来，通常为每个数据块在索引文件中设一个索引项。

13.1.1 顺序文件

一种最简单的索引结构要求文件按索引的属性排序，这样的文件称为顺序文件。当查找键是关系的主键时这种结构特别有用，当然对关系的其他属性也可使用这种结构。图13-2所
606 示为一个用顺序文件表示的关系。

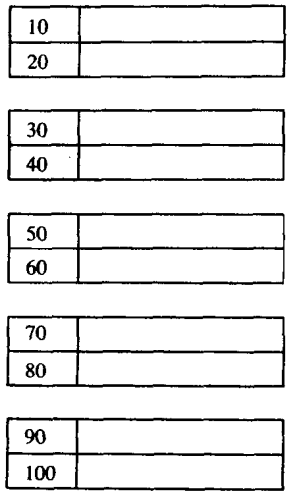


图13-2 顺序文件

键以及几种不同的键

术语“键”有几个涵义，本书在7.1.1节中用它表示一个关系的主键。在11.4.4节中我们学过“排序键”，文件的记录据此排序。现在来看“查找键”。已知在该属性(组)上的值，需要通过索引查找具有相应值的元组。当“键”的涵义不清楚时，尽量使用恰当的修饰词(“主”、“排序”或“查找”)。不过请注意，在13.1.2节和13.1.3节中，很多时候这三种键是同一涵义。

在这个文件中，元组按主键排序。这里假定主健是整数，我们只列出了主键字段。同时，我们还做了一个不代表典型情况的假定，即每个存储块中只可存放两个记录。例如，文件的第一个块中存放键值为10和20的两条记录。在这里和其他许多例子中，我们使用10的连续倍数来做键值，虽然实际中不会要求键值都是10的倍数或10的所有倍数都必须出现。

13.1.2 稠密索引

既然把记录排好了序，我们就可以在记录上建立稠密索引。它是这样的一系列存储块：块中只存放记录的键以及指向记录本身的指针，指针就是如12.3节讨论的那样的地址。我们说这里的索引是“稠密的”，因为数据文件中每个键在索引中都被表示出来。与此相比，将在13.1.3节中讨论的“稀疏”索引通常在索引中只为每个数据块存放一个键。

稠密索引文件中的索引块保持键的顺序与文件中的排序顺序一致。既然假定查找键和指针所占存储空间远小于记录本身，我们就可以认为存储索引文件比存储数据文件所需存储块要少得多。当内存容纳不下数据文件但能容纳下索引文件时，索引的优势尤为明显。这时，通过使用索引文件，我们每次查询只用一次I/O操作就能找到给定键值的记录。

例13.1 图13-3所示为一个建立在顺序文件上的稠密索引，该顺序文件的起始部分如图
607 13-2所示。为了方便起见，还是假定文件中记录的键值是10的倍数，虽然实际中我们不能指望找到这样一个有规律的键值模式。我们还假定每个索引存储块只可存放四个键-指针对。在实际中我们会发现每个存储块能存放更多的键-指针对，甚至是好几百个。

第一个索引块存放指向前四个记录的指针，第二个索引块存放指向接下来的四个记录的指针，依此类推。出于将在13.1.6节讨论的原因，实际中我们可能不希望装满所有的索引块。 □

稠密索引支持按给定键值查找相应记录的查询。给定一个键值 K，我们先在索引块中查找K。当找到 K 后，按照 K 所对应的指针到数据文件中寻找相应的记录。似乎在找到 K 之前我们需要检索索引文件的每个存储块，或平均一半的存储块。然而，由于有下面几个因素，基于索引的查找比它看起来更为有效：

1. 索引块数量通常比数据块数量少。

2. 由于键被排序，我们可以使用二分查找法来查找K。若有n个索引块，我们只需查找$\log_2 n$个块。

3. 索引文件可能足够小，以致可以永久地存放在主存缓冲区中。要是这样的话，查找键K时就只涉及主存访问而不需执行I/O操作。

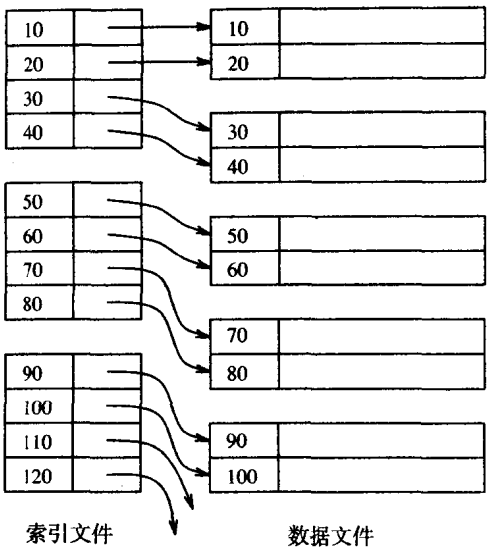


图13-3 顺序文件（右）上的稠密索引（左）

608

例13.2 假设一个关系有1 000 000个元组，大小为4096字节的存储块可存放10个这样的元组，那么这个关系所需的存储空间就要超过400 MB，因为太大而可能在主存中无法存放。然而，要是关系的键字段占30字节，指针占8字节，加上块头所需空间，那么我们可以在一个小大为4096字节的存储块中存放100个键-指针对。

这样，一个稠密索引就需要10 000个存储块，即40 MB。我们就有可能为这样一个索引文件分配主存缓冲区，不过要看主存的使用情况和主存的大小。进一步讲，$\log_2$（10 000）大约是13，因此采用二分查找法我们只需访问13~14个存储块就可以查找到给定键值。且由于二分查找法只检索所有存储块中一小部分（是这样一些存储块，它们位于1/4或3/4、1/8或3/8、5/8或7/8，等等），即使不能把整个索引文件存放到主存中，我们也可以把这些最重要的存储块放到主存中，这样，检索任何键值所需的I/O次数都远小于14。 □

索引块的定位

假设存在一些定位索引块的机制，根据索引可以查找到单个元组(若是稠密索引)或数据存储块(若是稀疏索引)。许多定位索引的机制可以使用。例如。假如索引较小，我们可以将它存放到主存或磁盘的预留存储区中。假如索引较大，则如我们在13.1.4节讲述的那样，我们可以在索引上再建一级索引，并将新的索引本身存放到某个固定的地方。对这一观点进行推广。最后就产生了13.3节中的B树，在B树中我们只需要知道根索引块的位置。

13.1.3 稀疏索引

要是稠密索引太大，我们可以使用一种称为稀疏索引的类似结构。它节省了存储空间，但查找给定值的记录需更多的时间。如图13-4所示，稀疏索引只为每个存储块设一个键-指针对。键值是每个数据块中第一个记录的对应值。

例13.3 同例13.1一样，假定数据文件已排序，且其键值为连续的10的倍数，直至某个较大的数。我们还继续假定每个存储块可存放四个键-指针对。这样，第一个索引存储块中为前四个数据存储块的第一个键值的索引项，它们分别是10、30、50和70。按照前面假定的键值模式，第二个索引存储块中为第5~第8数据存储块的第一个键值的索引项，它们分别是90、110、

130和150。图中还列出第三个索引存储块存放的键值，它们分别是假设的第9~第12个数据存储块的第一个键值。 □

609

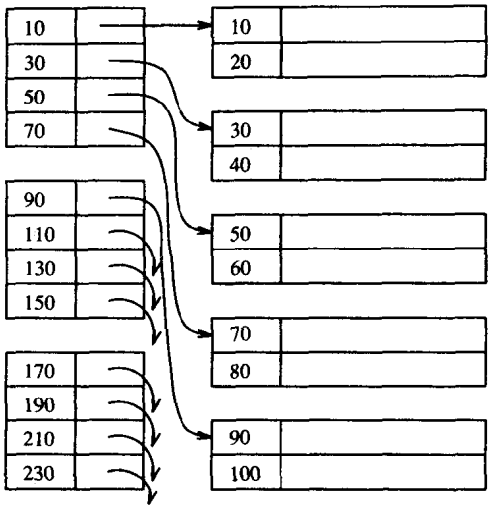


图13-4 顺序文件上的稀疏索引

例13.4 稀疏索引所需的索引存储块要比稠密索引少得多，以例13.2中更接近实际的参数为例。由于有100 000个数据存储块，且每个索引存储块可存放100个键-指针对，那么要是使用稀疏索引，我们就只需要1000个索引存储块。现在索引的大小只剩下4MB，这样大小的文件完全可能存到主存中。

另一方面，稠密索引不用去检索包含记录的块，就可以回答下面形式的查询："是否存在键值为 K 的记录？"。键 K 在索引中的存在足以保证数据文件中键值为 K 的记录的存在。当然如果使用稀疏索引，对于同样的查询，却需要执行I/O操作去检索可能存在键值为 K 的记录的块。 □

在已有稀疏索引的情况下，要找出键值为K的记录，我们得在索引中查找键值小于或等于K的最大键值。由于索引文件已按键排序，我们可以使用二分查找法来定位这个索引项，然后根据它的指针找到相应的数据块。现在我们必须搜索这个数据块以找到键值为K的记录。当然，数据块中必须有足够的格式化信息来标明其中的记录及记录内容。只要合适，可以采用12.2节和12.4节中的任何技术。

13.1.4 多级索引

索引文件本身可能占据多个存储块，正如我们在例13.2和例13.4中看到的那样。要是这些存储块不在我们知道能找到它们的某个地方，比如指定的磁盘柱面，那么，就可能需要另一种数据结构来找到这些索引存储块。即便能定位索引存储块，并且能使用二分查找法找到所需索引项，我们仍可能需要执行多次I/O操作才能得到我们所需的记录。

610

通过在索引上再建索引，我们能够使第一级索引更为有效。图13-5对图13-4进行了扩展，它是在图13-4的基础上增加二级索引得到的（和前面一样，我们假设使用10的连续倍数这一不常见的模式）。按照同样想法，我们可以在二级索引的基础上建立三级索引，等等。然而，这种做法有它的局限，与其建立多级索引，我们宁愿考虑使用在13.3节讲述的B树。

在这个例子中，一级索引是稀疏的，虽然我们也可以选择稠密索引来作为一级索引。但是，二级和更高级的索引必须是稀疏的，因为一个索引上的稠密索引将需要和其前一级索引同样多的键-指针对，因而也就需要同样的存储空间。因此，二级稠密索引只是增加额外的结构，而不会带来任何好处。

例13.5 继续以例13.4中假设的关系来分析，假定我们在一级稀疏索引上建立二级索引。由于一级稀疏索引占据1000个存储块，且我们可以在一个存储块中存放100个键-指针对，则只需要10个存储块来存放这个二级索引。

10个存储块完全可以一直缓存在主存中。若是这样，要查找给定键值 K 的记录，可以先查看二级索引以找到小于或等于键值 K 的最大键值。根据这个最大键值对应的指针找到一级

索引的某个存储块 B，通过这个存储块 B，我们就肯定能找到所需的记录。假若存储块 B 不在内存中，我们就把存储块 B 读进内存，这是我们所需的第一次I/O操作。在块 B 中查找小于或 611
等于键值 K 的最大键值，要是有键值为 K 的记录存在，则通过块 B 中这一键值对应的指针就可以找到包含键值为 K 的记录的数据块。这个数据块需要又一次的I/O操作。这样，我们只用两次I/O操作就完成了查询。 □

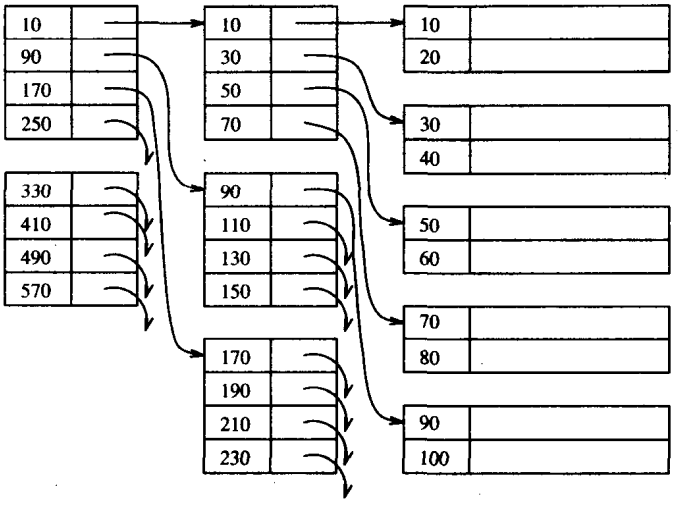


图13-5 增加一个第二级稀疏索引

13.1.5 重复查找键的索引

到目前为止，我们都假定作为建立索引基础的查找键是关系的键，所以对任何一个键值，关系中最多有一个记录存在。然而，索引经常用于非键属性，因此有可能一个给定的键对应于多个记录。假如按查找键对记录进行排序，而不管相同键值记录之间的次序，那么，我们可以采用前面介绍的方法来处理不是关系的键的查找键。

对前面方法最简单的扩充是为数据文件建立稠密索引：每一个具有键值 K 的记录设一索引项。也就是说，我们允许索引文件中出现重复的查找键。找出所有给定索引键 K 的记录因此就比较简单：在索引文件中找到具有键值 K 的第一个索引项，后面紧接着就是其他的具有键值 K 的索引项，找出它们，然后依据这些索引的指针找出所有具有键值 K 的记录。

更为有效的方法是为每个键值K只设一个索引项。该索引项的指针指向键值为 K 的第一个记录。为了找出其他键值为 K 的记录，需在数据文件中顺序向前查找；在排序的数据文件中，这些记录一定紧跟在所找到的记录后存放。图13-6举例说明了这一想法。

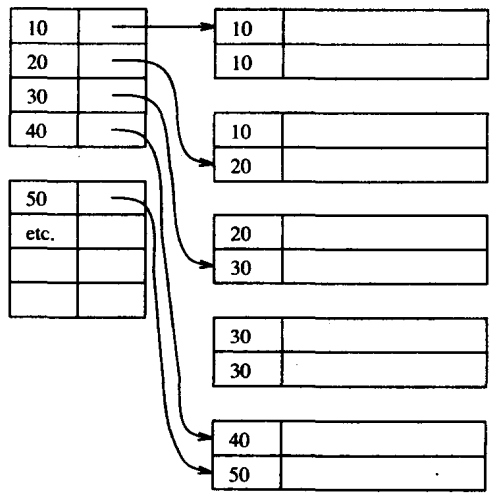


图13-6 允许重复键的稠密索引

例13.6 假如要找出图13-6中所有索引键值为20的记录。先在索引中找到键值为20的索引项并顺着它的指针找到第一个键值为20的记录。然后在数据文件中继续往前找，由于刚好 612

处于数据文件第二个存储块的最后一条记录，我们就到第三个存储块中去查找 ⊖。我们发现第三块中第一个记录的键值为20，但第二个记录的键值为30。因此，不用再往前查找；我们已经找到键值为20的两条记录。 □

图13-7为图13-6所示数据文件上的一个稀疏索引。这种稀疏索引是很常见的；它的键-指针对对应数据文件中每个块的第一个查找键。

为了在这种数据结构中找出所有索引键为 K 的记录，我们先找到索引中键值小于或等于 K 的最后一个索引项 E_1，然后往索引起始的方向找，直到碰到第一个索引项或碰到一个严格小于键值K的索引项E_2为止。从 E_2 到 E_1 的索引项（含 E_2 和 E_1）指向所有可能包含查找键为 K 的记录的数据块。

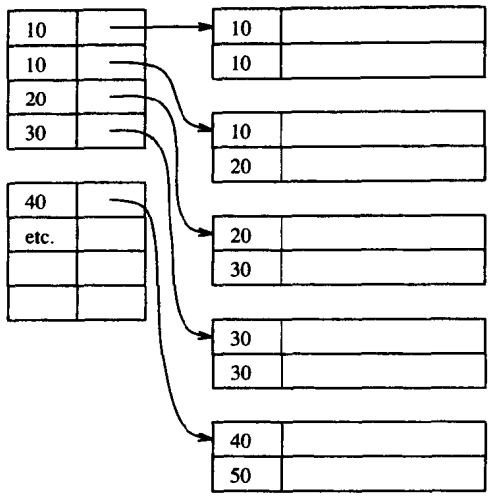


图13-7 指明了每个块中最小查找键的稀疏索引

例13.7 假定我们想在图13-7中查找键值为20的记录。第一个索引块的第三个索引项就是 E_1，它是符合键值小于等于20的最后一个索引项。我们向后查找，立即找到了键值小于20的索引项。因此，第一个索引块的第二个索引项就是 E_2。它们相应地指向第二、第三个数据块，正是在这两个数据块中我们找到查找键为20的所有记录。

再举一例。若 $K=10$，则 E_1 就是第一个索引块的第二个索引项，而E_2不存在，因为我们
613 在索引中找不到更小的键值。因此，按照第一个索引项到第二个索引项的指针，我们找到前两个数据块。在这两个数据块中我们就能找到所有键值为10的记录。 □

一种稍有不同的方式如图13-8所示。图中为每个数据块中新的、即未在前一存储块中出现过的最小查找键设一个索引项。要是在存储块中没有新键值出现，那么就为该块中惟一的键值设一个索引块。在这种方式下，我们查找键值为 K 的记录可以通过在索引中查找第一个键值满足如下条件之一的索引项。

a) 等于 K；或者

b) 小于 K，但下一个键值大于 K。

我们按照这个索引项的指针找到相应的数据块。要是在这个数据块中至少找到一个键值为K 的记录，那么我们就要继续查找其他数据块，直到找出所有查找键为 K 的记录。

例13.8 假定要在图13-8所示的结构中查找K=20的记录，上述规则指示出第一个索引块的第二个索引项，我们沿着这个索引项的指

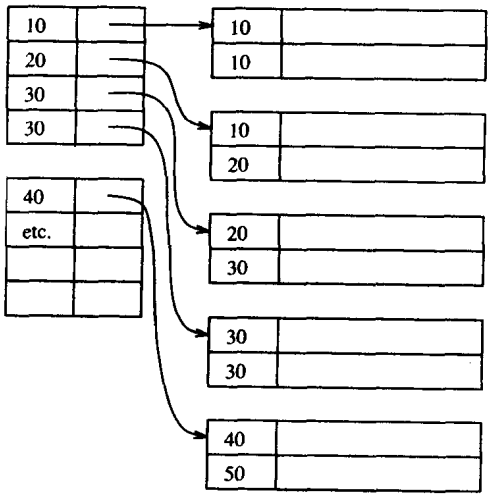


图13-8 指明每块中最小新查找键的稀疏索引

⊖ 为了能找到数据文件的下一个块，我们可以用链表的形式把所有块链接起来，换言之，给每个存储块一个指针指向下一个存储块。我们也可以返回索引，沿下一指针到达下一数据文件块。

针找到键值为20的第一个存储块。由于下一个存储块也有键值为20的记录，我们必须继续向前查找。 614

假若 $K = 30$，上述规则指示出第三个索引项。我们沿着它的指针找到第三个数据块，键值为30的记录在这个数据块中开始存放。最后，假若 $K = 25$，则上述选择规则的(b)部分指出第二个索引项。我们因此来到第二个数据块。如果存在查找键为25的记录，则至少有一个在该块上值为20的记录之后，因为我们知道第三个存储块中新出现的第一个键值是30。既然没有键值为25的记录，我们的查找就失败了。 □

13.1.6 数据修改期间的索引维护

到目前为止，我们都假定数据文件和索引文件由一些连续、装满某种类型的记录的存储块组成。由于随着时间的推移数据会发生变化，我们需对记录进行插入、删除和更新。这势必引起像顺序文件这样的文件组织发生变化，以至于曾经能容纳于块中的记录不再被容纳。我们可以使用12.5节讲述的技术来重新组织数据文件。我们来回忆一下那一部分的三个重要想法。

1. 当需要额外的存储空间时，创建溢出块；或当溢出块中记录被删除后不再需要该存储空间时删除溢出块。溢出块在稀疏索引中没有索引项而应该被看做是基本存储块的扩充。

2. 若不用溢出块，可以按序插入新的存储块。要是这样做，那么新的存储块就需要在稀疏索引中设索引项。在索引文件中索引项的变动会引起和数据文件的插入与删除同样的问题。要是我们创建新索引块，那么这些索引块必须能以某种方法定位，例如像13.1.4节中那样使用另一级索引。

3. 当块中没有空间可以插入元组时，有时可移动一些元组到相邻块；相反，当相邻块元组太少时，我们可以合并它们。

然而，当数据文件发生变化后，我们通常必须对索引进行调整以适应数据文件的变化。具体的方法要依赖于索引是稠密还是稀疏以及在上述三种方法中选择哪一个。不过，我们应记住一个一般的原则：

- 索引文件是顺序文件的一个例子，键-指针对可以看做是按查找键排序的记录。因此，数据文件修改过程中用来维护数据文件的那些策略同样适用于索引文件。

在图13-9中，我们总结了针对数据文件七种不同行为而对稠密索引或稀疏索引所需采取的措施。这七种行为包括：创建或删除空溢出块、创建或删除顺序文件的空块、插入、删除和移动记录。注意，假定空的存储块才能被创建或删除，具体来说，要是我们想删除一个有记录的块，需要先删除其中的记录或把记录移到其他的块。 615

行　为	稠密索引	稀疏索引
创建空溢出块	无	无
删除空溢出块	无	无
创建空顺序块	无	插入
删除空顺序块	无	删除
插入记录	插入	更新（？）
删除记录	删除	更新（？）
移动记录	更新	更新（？）

图13-9 顺序文件上的行为对索引文件的影响

在这个表中，我们注意到：

- 创建或删除一个空溢出块对两种索引均无影响。对稠密索引不产生影响是因为索引是针对记录；对稀疏索引不产生影响是因为索引是针对基本存储块而非溢出块。
- 创建或删除顺序文件的块对稠密索引无影响，这仍是因为索引是针对记录而非存储块；但它对稀疏索引会有影响，因为我们必须为创建或删除的块分别创建或删除一个索引项。
- 插入或删除记录导致稠密索引上的同一动作，因为记录的相应键-指针对要被插入或删除。然而，这对稀疏索引通常没有影响。例外的情况是当记录是存储块中第一个记录时，稀疏索引中对应块的键值必须被更新。因此，我们在图13-9中相应的更新操作后加注了一个问号，表示这个更新是可能的，但不确定。
- 类似地，移动一个记录，不论是在块内还是在块间都会引发稠密索引中相应索引项的更新；对稀疏索引而言，则仅当被移动记录是或变成了该块中的第一个记录时才会引发更新操作。

为数据变迁所做的准备

由于随着时间的推移，关系或类外延通常会增长。因而较为明智的做法是：不论数据块还是索引块，都为其保留一定的空闲空间。比如说，一开始在每块中只使用75%的空间。这样，在创建溢出块或在块之间移动记录之前，我们能运行一段时间。无溢出块或仅有少量的溢出块的优势在于访问每个记录平均I/O次数仅为1。溢出块数越多，则查找给定记录所需访问的平均存储块数也多。

我们将通过一系列的例子来阐明上述几点所隐含的一组算法。这些例子既包括稀疏索引又
616 包括稠密索引，既包括记录移动方法又包括溢出块方法。

例13.9 首先，让我们来考虑顺序文件的记录删除操作，该顺序文件上建有稠密索引。我们从图13-3所示的文件和索引开始。假设键值为30的记录被删除。图13-10所示为记录删除后的结果。

首先，键值为30的记录从顺序文件中删除。我们假定块外的指针可以指向块内的记录，因而我们不打算将剩余记录40在块中向前移动，而选择在记录30处留下一个删除标记。

在索引中，我们删去键值为30的键-指针对。假定不允许块外的指针指向索引记录，因而没有必要为该键-指针对留下删除标记。所以，我们采取合并索引块的方法，并把后面的索引记录前移。□

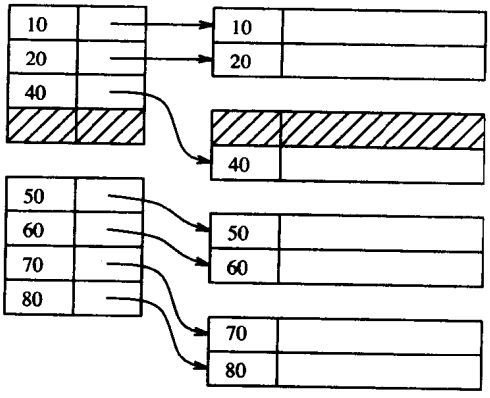


图13-10 稠密索引中删除查找键为30的记录

例13.10 现在，我们来考虑顺序文件上的
617 两个删除操作，该顺序文件上建有稀疏索引。从图13-4所示的文件和索引开始，同样假设键值为30的记录被删除。我们还假定块中记录可前后移动：要么块外没有指针指向记录，要么我们使用图12-16所示的偏移量表来支持这样的移动。

删除记录30后的情形如图13-11所示。记录30已被删除，后面的记录40向前移以使块的前部紧凑。由于40现在是第二个数据块的第一个键值，我们需要更新该块的索引记录。从图13-11中我们看到，与指向第二数据块指针相对应的键值已从30改为40。

现在，假定记录40也被删除，删除后的情形如图13-12所示。第二个数据块已经没有记录。如果顺序文件是存放在任意的存储块上（例如，不是柱面上连续的存储块），那么我们可以把该块链接到可用空间链表中。

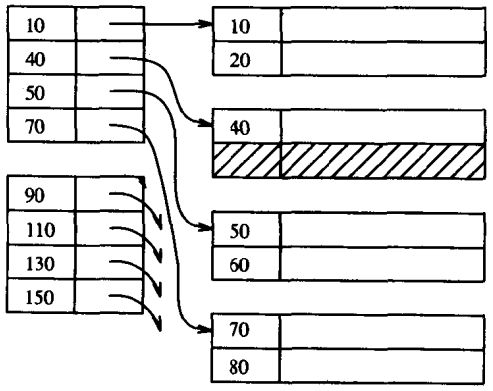


图13-11 稀疏索引中删除查找键为30的记录

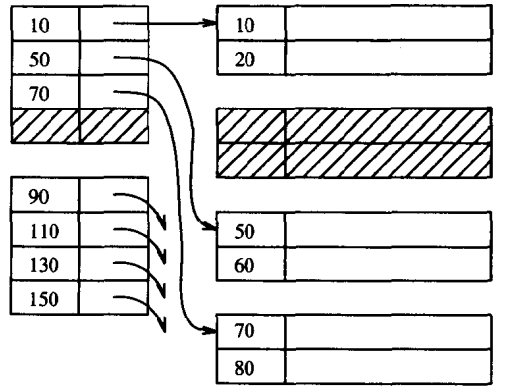


图13-12 稀疏索引中删除查找键为40的记录

要完成记录40的删除，我们还要调整索引。既然第二个数据块不再存在，我们就从索引中删除其索引项。在图13-12中通过把后面的索引项前移，使第一个索引块紧凑。这一步是可选的。 □

例13.11 现在，我们来考虑插入的影响，从图13-11开始，该图中我们刚刚从一个有稀疏索引的文件中删除了记录30，但记录40还保留着。我们现在插入一个键值为15的记录。通过查看稀疏索引，我们发现该记录属于第一个数据块。但是该数据块已满，它存放着记录10和记录20。

我们能做的一件事是在附近找有空闲空间的块，这样就找到第二个数据块。因此，我们就在文件中往后移动记录，以腾出空间来存储记录15。结果如图13-13所示。记录20从第一个数据块移到了第二个数据块，并且记录15存放到记录20的位置上。为了在第二个数据块中存放记录20和保持记录的顺序，将第二个数据块的记录40往后移并把记录20放在它的前面。

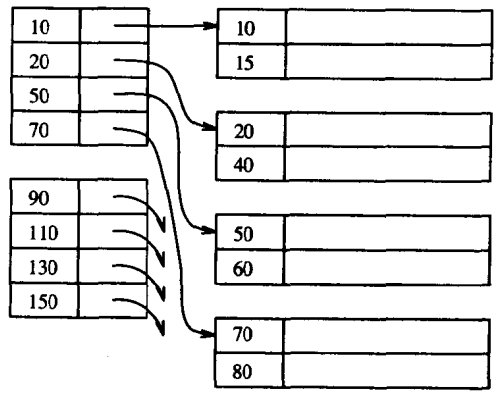


图13-13 在有稀疏索引的文件中插入，使用立即重组方法

618

最后一步是修改变动的数据块的索引项。我们可能需要修改第一个数据块的键-指针对的键值。不过这里不用，因为插入的记录不是该数据块的第一个记录。然而，我们需要修改第二个数据块的索引项的键值，因为该块的第一个记录的键值原来是40，现在变成了20。 □

例13.12 例13.11所示策略的问题在于，我们恰好在相邻存储块中找到了空闲空间。要是前面没有删除键值为30的记录，那么我们查找空闲空间的工作只是徒劳。原则上讲，我们不得不把从记录20到数据文件最后的所有记录全部后移，直到我们到达文件末尾并能创建额外的块。

因为有这样的风险，用溢出块为有太多记录的基本块补充空间的做法通常更为明智。图13-14所示为在图13-11所示结构中插入键值为15的记录后的情形。和例13.11一样，第一个数据块有太多记录。我们不是将记录移动到第二个数据块，而是为第一个数据块创建一个溢出块。

可以看到图13-14中每个数据存储都有一个小凸起，它表示块头中存放指向溢出块的指针的空间。任意多个溢出块之间可通过这些指针空间链接起来。

在我们这个例子中，记录15被插入到位于记录10之后的正确位置；记录20为腾出空间被移到溢出块中。因为数据块1的第一个记录没有改变，所以索引项也就不必改变。注意，索引中不存在该溢出块的索引项，因为该溢出块被看成是数据块1的扩充，它不是顺序文件本身的存
619 储块。 □

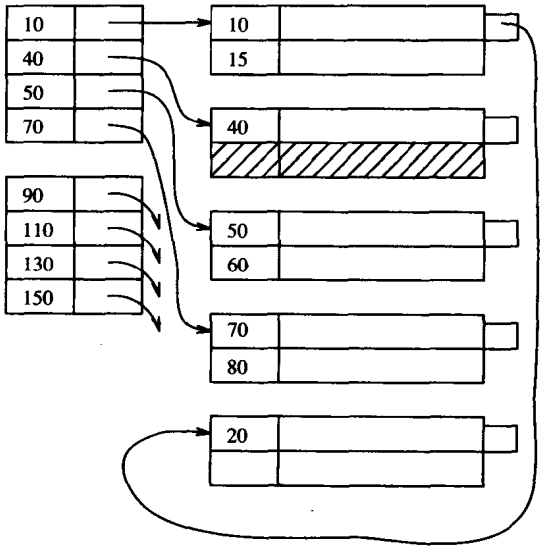


图13-14 使用溢出块技术在有稀疏索引的文件中插入

13.1.7 习题

* **习题13.1.1** 假定每个存储块可存入3个记录或10个键-指针对。设记录数为n，保存一个数据文件和

a) 一个稠密索引。

b) 一个稀疏索引。

各需要多少块（表示为 n 的函数）？

习题13.1.2 假定每个存储块可存放30个记录或200个键-指针对，但数据块和索引块充满度均不许超过80%。重做习题13.1.1。

! **习题13.1.3** 假定我们使用多级索引，级数恰好使最后一级索引只有一个存储块。重做习题13.1.1。

*!! **习题13.1.4** 假定每个存储块可存放3个记录或10个键-指针对，如习题13.1.1，但是可能出现重复键。说得具体点，数据库中1/3的键有一个记录，1/3的键有两个记录，1/3的键恰好有三个记录。假设我们有一个稠密索引，但它只为每个查找键值设一个索引项，指针指向该键值的第一个记录。最初没有存储块在内存中，试计算找到给定键值K的所有记录所需的平均磁盘I/O数。你可以认为包含键K的索引块地址已知，尽管它在磁盘上。

620 ! **习题13.1.5** 分别按下述情况重做习题13.1.4。

a) 稠密索引：每个记录设一个键-指针对，包括有重复键的记录。

b) 稀疏索引：如图13-7所示，索引中指明每个数据块的最小键值。

c) 稀疏索引：如图13-8所示，索引中指明每个数据块新出现的最小键值。

! **习题13.1.6** 假如在一个关系的主键属性上建立稠密索引，那么，我们就可以使指向元组（或表示这些元组的记录）的指针指向索引项而非记录本身。这两种方式各自的优点是什么？

习题13.1.7 在图13-13基础上继续做改变，假设接下来删除键值为60、70和80的记录，然后插入键值为21、22等直到29的所有记录。假定额外的空间可通过下述方式得到：

* a) 为数据文件或索引文件增加溢出块。

b) 记录尽量后移：必要时在数据文件和/或索引文件末尾增加存储块。

c) 可根据需要在这些文件中间插入新的数据块或索引块。

621

*! **习题13.1.8** 假定在处理一个具有n个记录的数据文件中的插入操作时，我们按照需要创建溢出块。同时还假定数据块当前平均只充满一半。假如随机地插入记录，试问得插入多少记录才能使我们查找到给定键值记录所需访问存储块（包括溢出块）的平均数达到2？假定在查找中，我们先查看索引指向的块，然后只按顺序查看溢出块，直到找到所需记录，该记录必然在链上的某一块中。

13.2 辅助索引

13.1节中所描述的数据结构为*主索引*，因为它们决定了被索引记录的位置。在13.1节中数据文件是按查找键的值排序，由此能决定记录的位置。在13.4节中将讨论另一种常用的主索引：散列表，其中查找键决定记录所属的“桶”。

然而，为了给各种各样的查询提供便利，经常需要在一个关系上建多个索引。例如，再考虑图12-1中定义的关系MovieStar。由于将name声明为该关系的主键，我们可以期望DBMS创建主索引结构来支持指定明星名字的查询。但是我们现在想在数据库中按出生日期查找指定明星的信息，因而执行如下查询：

```
SELECT name, address
FROM MovieStar
WHERE birthdate = DATE '1952-01-01';
```

我们需要在属性birthdate上建立*辅助索引*来帮助执行这个查询。在SQL系统中，可以通过如下显示的命令来建立这样一个索引：

```
CREATE INDEX BDIndex ON MovieStar(birthdate);
```

辅助索引可用于任何索引目的：这种数据结构有助于查找给定一个或多个字段值的记录。但是，辅助索引与主索引最大的差别在于辅助索引不决定数据文件中记录的存放位置。而仅能告诉我们记录的当前存放位置，这一位置可能是由建立在其他某个字段上的主索引确定的。辅助索引和主索引的这一差别有一个有趣的推论：

- 谈论一个稀疏的辅助索引是毫无意义的。因为辅助索引不影响记录的存储位置，我们也就不能根据它来预测键值不在索引中显式指明的任何记录的位置。
- 因此，辅助索引总是稠密索引。

622

13.2.1 辅助索引的设计

辅助索引是稠密索引，且通常有重复值。如前所述，索引项由键-指针对组成，这里的“键”是查找键且不要求惟一。为了便于找到给定键值的所有索引项，索引文件中索引项按键

值排序。要是我们在这种结构上建立二级索引，那么这个二级索引将是稀疏的，其原因在13.1.4节中已讨论过。

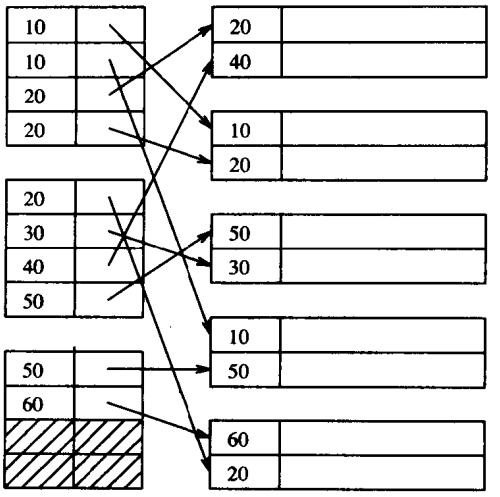


图13-15 辅助索引

例13.13 图13-15所示为一个典型的辅助索引。与我们前面图示的准则一样：数据文件中每块存放两个记录。记录只显示了各自的查找键；其值为整型，而且像前面一样我们给它们取值为10的倍数。要注意，与13.1.5节数据文件不同的是，这里的数据没有按查找键排序。

然而，索引文件中的键是排序的。这样就造成索引块中的指针并不是指向一个或少数几个连续存储块，而是指向许多不同的数据块。例如，为了检索键值为20的所有记录，不仅要查找两个索引块，而且还得访问指针指向的三个不同数据块。因此，查找同样数量的记录，使用辅助索引比使用主索引可能需要多得多的磁盘I/O。但是这个问题是无法解决的，我们无法控制数据块中的元组顺序，因为这些元组可能已按其他属性排序。

我们可以在图13-15所示的辅助索引上建立二级索引。这一级索引将是稀疏的，如13.1.4节
623 讨论的那样，对应于每个索引块的第一个键值或第一个新出现的键值有一个键-指针对。 □

13.2.2 辅助索引的应用

除了能在被组织成顺序文件的关系（或类外延）上建立附加索引外，辅助索引甚至还用做某些数据结构的主键索引。这些结构之一就是“堆”结构。在这种结构中，关系的记录之间没有特定的顺序。

第二种需要辅助索引的常见数据结构是聚簇文件。在这种结构中，两个或多个关系的元组被混在一起。下面的一个例子说明了这种组织结构在特定情况下存在的合理性。

例13.14 假设有两个关系，其数据模式可简要定义如下：

```
Movie(title, year, length, inColor, StudioName, ProducerC# )
Studio(name, address, presc# )
```

属性title和year一起组成关系Movie的键，而属性name是关系Studio的键。Movie中的属性studioName是参照Studio中的name的外键。进一步假定查询的常见形式如下：

```
SELECT title, year
FROM Movie, studio
WHERE studioName = zzz AND Movie, StudioName=Studio Name;
```

这里，*zzz*表示某一特定制片厂的名称，如，“Disney”。

如果我们确信关系Moive和Studio之间的基于制片厂名称的连接很常见，就可以采用一种聚簇文件结构来使这些连接效率更高。在这种结构中，关系Movie的元组和关系Studio的元组存放在相同的一系列块中。说得具体些，我们在每个Studio的元组后面存放关系Movie中该制片厂的所有电影元组。其结构如图13-16所示。

如果我们为Studio建立了一个查找键presC#的索引，那么不管zzz是什么值，我们都可

以快速地找到与zzz对应的制片厂。并且，在聚簇文件中，Studio的元组后面将跟随着Movie中所有studioName值与该制片厂的name值相匹配的元组。从而，我们可以通过尽可 624
能少的I/O来找到某个制片厂制作的全部影片，因为我们要查找的Movie元组已经被尽可能密集地存放在相邻的块中了。 □

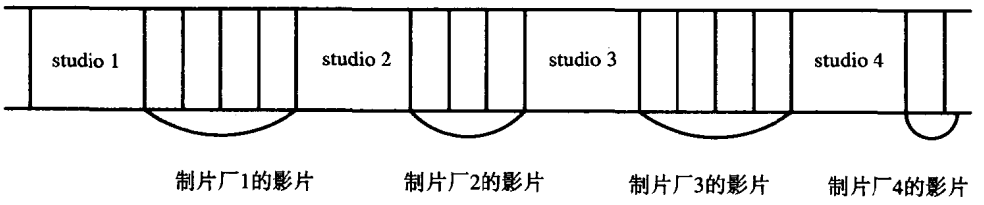


图13-16 将制片厂及其制作的影片聚簇在一起的聚簇文件

13.2.3 辅助索引的间接性

图13-15所示结构存在空间浪费，有时浪费很大。假如某个索引键值在数据文件中出现n次，那么这个键值在索引文件中就要写n次，如果我们只为指向该键值的所有指针存储一次键值，这样就会比较好。

避免键值重复的一种简便方法是使用一个称为桶的间接层，它介于辅助索引文件和数据文件之间。如图13-17所示，每个查找键K有一个键-指针对，指针指向一个桶文件，该文件中存放K的桶。从这个位置开始，直到索引指向的下一个位置，其间指针指向索引键值为K的所有记录。

例13.15 例如在图13-17的索引文件中，沿索引键为50的索引项指针找到中间“桶”文件。这一指针刚好将我们带到桶文件中第一个块的最后一个指针。继续向前查找，找到下一块的第一个指针。因为索引文件中键值为60的索引项指针刚好指向桶文件的第二个块的第二个指针，所以我们停止查找。 □ 625

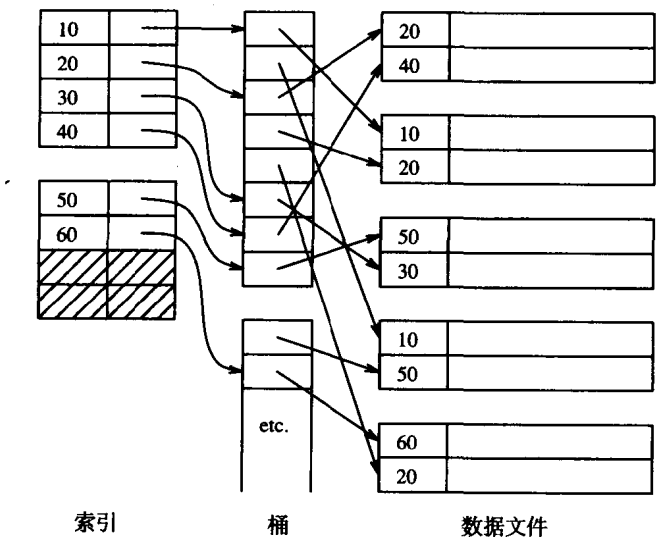


图13-17 在辅助索引中使用间接节省空间情况

在图13-17所示的方式中，只要查找键值的存储空间比指针大就可以节省空间。不过，即使在键值和指针大小相当的情况下，在辅助索引上使用间接也有一个重要的好处：我们通常可

以在不访问数据文件记录的前提下利用桶的指针来帮助回答一些查询。特别是，当查询有多个条件，而每个条件都有一个可用的辅助索引时，我们可以通过在主存中对指针集合求交来找到满足所有条件的指针，然后只需要检索交集中指针指向的记录。这样，我们就节省了检索满足部分条件而非所有条件的记录所需的I/O开销[⊖]。

例13.16 考虑常用的关系

```
Movie(title, year, length, inColor, studioName, ProducerC#)
```

假定我们在studioName和year上都建立了有间接桶的辅助索引，而且我们要执行如下查询：

```
SELECT title
FROM Movie
WHERE studioName = 'Disney' AND  year = 1995;
```

即：找出Disney在1995年制作的所有电影。

图13-18说明了如何使用索引来回答这个查询。通过studioName上的索引，我们找出了所有指向Disney制作的电影的指针。但是，我们并不把这些记录从磁盘上取到主存中，而是通过year上的索引，再找出所有指向1995年制作的电影的指针。然后我们求两个指针集的交集，正好得到1995年Disney制作的所有电影。现在我们到磁盘上去检索所有包含一部或几部这样的电影的块，这样只需检索尽可能少的数据块。 □

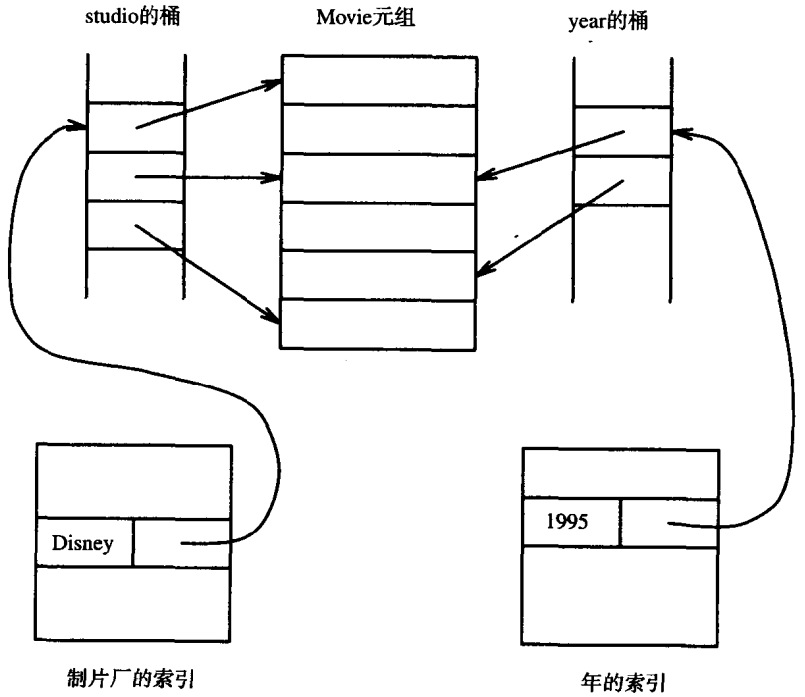


图13-18 在主存中求桶的交集

⊖ 假若我们直接从索引而非桶中取得指针，也可以使用这一指针求交技巧。不过，由于指针比键-指针对使用更少存储空间，因此使用桶通常能节省磁盘I/O。

13.2.4 文档检索和倒排索引

多年来，信息检索界都在处理文档的存储和按关键字集高效检索文档的问题。随着WWW的出现以及在线保存所有文档成为可能，基于关键字的文档检索已成为数据库最大的难题之一。尽管可用来找出相关的文档的查询有许多种，但最简单、最常见的形式可用关系的术语描述为： 626

- 一个文档可被看成是关系Doc的元组。这个关系有很多属性，每个属性对应于文档可能出现的一个词。每个属性都是布尔型——表明该词在该文档出现还是没有出现。因此，这一关系模式可以被看做：

```
Doc(hasCat, hasDog,…)
```

其中hasCat取值为真当且仅当该文档中至少出现一次“cat”这个词。

- 关系Doc的每个属性上都建有辅助索引。不过，我们不必费心为属性值为FALSE的元组建索引项；相反，索引只会将我们带到出现该词的那些文档，即，索引中只有查找键值为TRUE的索引项。

- 我们不是给每个属性（即每个词）建立一个单独的索引，而是把所有的索引合成一个，称为*倒排索引*。这个索引使用间接桶来提高空间利用率，正如13.2.3节中讨论的那样。

例13.17 图13-19所示为一个倒排索引。取代记录数据文件的是一个文档集合，每个文档可以被存放在一个或多个磁盘块上。倒排索引本身由一系列词-指针对组成；词实际上是索引 627
的查找键。正如到目前为止讨论的任何一种索引那样，倒排索引被存储在连续的块中。不过，在一些文档检索应用中，数据的变化不像典型数据库中那样频繁，因而通常也就不用提供应付块溢出或索引变化的措施。

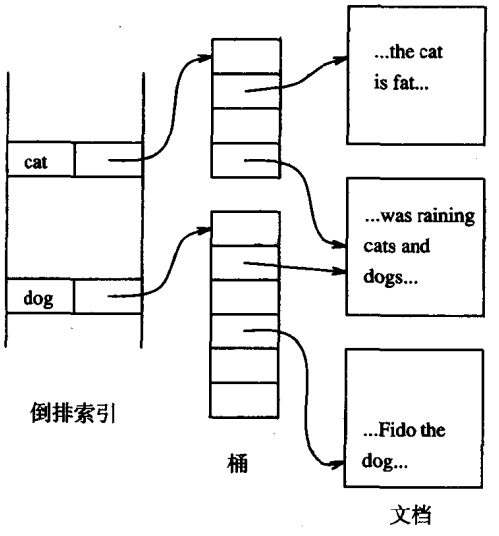


图13-19 文档的倒排索引

指针指向“桶”文件中的位置。例如，在图13-19中，“cat”一词有一个指针指向桶文件。该指针在桶文件中指明的位置之后是所有包含“cat”的文档的指针。图中给出了一些这样的指针。类似地，图中“dog”一词的指针指向一个指针列表，该列表中的指针指向包含“dog”的所有文档。 □

桶文件中的指针可以是：

1. 指向文档本身的指针。

2. 指向词的某一次出现的指针。在这种情况下，指针可以是由文档的第一个块和一个表示该词在文档中出现次数的整数构成的对。

一旦有了使用这种由指向词的每次出现的指针桶的想法，我们可能就会想扩展这个想法，
628 使桶数组包含更多有关词的出现的信息。这样，桶文件本身就成了有重要结构的记录集合。这种做法的早期应用是区分一个词出现在文档的题目、摘要，还是正文中。随着Web上文档的增长，尤其是使用HTML、XML或其他标记语言的文档的增长，我们也可以指明与词关联的标记。例如，我们不仅可以区分出现在题头、表或锚中的词，而且可以区分以不同字体和字号出现的词。

对信息检索的进一步讨论

有很多技术可用于改进基于关键字的文档检索效率。完整介绍这些技术已超出本书的范围，这里介绍两种有用的技术：

1. 抽取词干。在将单词的出现放入索引中之前。我们删除词的后缀以找出它的“词干”。例如，复数名词可被当作其单数形式处理。因此，例13.17中的倒排索引显然使用了抽取词干技术，因为搜索“dog”一词时我们不仅得到“dog”的文档，而且得到一个有单词“dogs”的文档。

2. 无用词。像“the”、“and”之类最常用的词称为无用词，通常不包含在倒排索引中。原因在于好几百个常用词出现在太多的文档中，以至于它根本无益于检索特定主题。去除无用词还可以明显地缩小索引。

例13.18 图13-20所示为一个标明HTML文档中词的出现情况的桶文件。如果有出现类型即标记的，就在第一列指明。第二、第三列一起构成指针指向词的出现。第三列指明文档，而第二列给出了该文档中该词出现的次数。

可以用这种数据结构来回答关于文档的各种查询，而且不用仔细查看文档。例如，假设我们想找出有关狗的，并将狗与猫做了比较的文档；没有深刻理解文档的内容，我们就无法准确地回答这个查询。但是，要是查找符合以下条件的文档，我们可以获得一个很好的提示：

a) 在标题(title)中提到狗；且

629 b) 在某个锚中提到猫——该锚可能是连到一个关于猫的文档的链接。

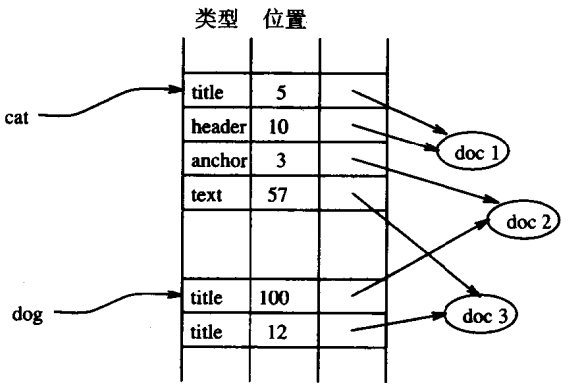


图13-20 在倒排索引中存放更多信息

我们可用通过对指针求交来回答这个查询。也就是说，按对应于“cat”的指针找到这一单词的所有出现。我们从桶文件中选择有“cat”出现且类型为“锚”的文档指针。接着，我们

找到“dog”的桶中项目，并从中选择类型为“标题”的文档指针。如果把这两个指针集相交，就得到符合在标题中提到“dog”且在某个锚中提到“cat”这一条件的文档。 □

桶中的插入与删除

在一些图如图13-19中，我们所给的桶是大小适中的紧凑数组。实际上，它们是单个字段(指针)的记录，且像其他任何记录集合一样存放在块中。因此在插入和删除指针时，我们可用到目前为止学过的任一种技术，例如为文件的扩充预留空闲空间、溢出块和可能的块内或块间记录移动。在后一种情况下，当我们移动倒排索引和桶中指针指向的记录时，必须小心地改变从倒排索引到桶文件中的相应指针。

13.2.5 习题

* **习题13.2.1** 随着数据文件的插入和删除，辅助索引也需要修改。请提出一些使辅助索引与数据文件的改变同步的方法。

! **习题13.2.2** 如习题13.1.1一样，假定我们的存储块能存放3个记录或10个键-指针对。假设 630
用这样的块来存放一个数据文件和查找键K上的辅助索引。对于文件中出现的每个K值v，可能有1个、2个或3个记录的K字段值为v。正好1/3的键值出现一次，1/3的键值出现2次，1/3的键值出现3次。再假定索引块和数据块都在磁盘上，但有在一种结构能使我们接受任意K值v，并获知包含一个或多个查找键值为K的记录的所有索引块指针（或许在主存中有一个二级索引）。计算检索所有查找键值为v的记录的平均磁盘I/O数？

*! **习题13.2.3** 考虑如图13-16所示的聚集文件组织结构，且假定每个存储块可以放10个制片厂记录或电影记录。再假定每个制片厂制作的电影数都在1～m之间均匀分布。如果表示成m的函数，则检索某个制片厂和它所制作的电影所需的平均磁盘I/O数是多少？如果电影记录随机分布在大量块中，这个平均磁盘I/O数又是多少？

习题13.2.4 假定一个存储块可存放3个记录、10个键值-指针对或50个指针。如果我们使用图13-17的间接桶：

* a) 如果平均每个查找键值出现在10个记录中，存放3000个记录和它的辅助索引共需要多少块？如果不使用桶又需要多少块？

! b) 如果给定键值的记录数没有限制，所需的最大和最小存储块数各为多少？

! **习题13.2.5** 在习题13.2.4(a)的假定下，在有桶结构和无桶结构时查找和检索具有给定键值的10个记录所需的平均磁盘I/O各为多少？假定开始时内存中没有任何存储块，但定位索引块或桶的块时，可以不引入额外的I/O，而只需要检索这些块并将其送入主存的I/O。

习题13.2.6 假定和习题13.2.4一样，一个存储块可存放3个记录，10个键-指针对或50个指针。和例13.16一样，设关系`movie`的属性`studioName`和属性`year`上建有辅助索引。假定有51部Disney制作的电影，101部1995年制作的电影，其中只有一部是Disney制作的。分别计算下列情况下回答例13.16的查询（找出1995年Disney制作的电影）所需的磁盘I/O数：

* a) 两个辅助索引都使用桶，从桶中检索指针，并在主存中求它们的交，然后只检索1995年Disney制作的那一部电影对应的记录。 631

b) 不使用桶，而使用`studioName`上的索引来取得指向Disney影片的指针，检索它们并选择1995年制作的那些电影记录。假定任意两个Disney影片的记录都不在同一个存储块中。

c) 同（b）一样，但从`year`上的索引开始。假定任意两个1995年制作的电影的记录都

不在同一个存储块中。

习题13.2.7 假定我们有一个1000个文档的库，且希望建立一个10 000个词的倒排索引。一个存储块能容纳10个词-指针对或50个指针，指针可以指向文档或文档的某个位置。词的分布为Zipfian分布（参见16.4.3节中的“Zipfian分布”框）；出现频率排名第i位的词出现的次数是$100\,000\sqrt{i}$，其中$i = 1, 2, \cdots, 10\,000$。

* a) 每个文档中平均有多少个词？

* b) 假定我们的倒排索引中只为每个词记录出现该词的所有文档。存放该倒排索引最多需要多少个存储块？

c) 假定我们的倒排索引保存指向每个词的每次出现的指针。存放该倒排索引需要多少存储块？

d) 如果常用的400个词（“无用”字）不包括在索引中，重做(b)。

e) 如果常用的400个字不包括在索引中，重做(c)。

习题13.2.8 如果我们使用一个扩充的倒排索引，如图13-20所示，就能执行许多其他类型的查询。说明如何使用这种索引去找到：

* a) “cat”和“dog”彼此相距不超过五个位置并且出现在同一类元素（如：标题、文本或锚）中的文档。

b) “cat”后刚好隔一个位置就跟（有）“dog”的文档。

c) 标题目中同时出现“dog”和“cat”的文档。

13.3 B树

虽然一级或两级索引通常有助于加快查询，但在商用系统中常使用一种更通用的结构。这一通用的数据结构簇称为B树，而最常使用的变体称为B+树。实质上：

632

- B树能自动地保持与数据文件大小相适应的索引层次。
- 对所使用的存储块空间进行管理，使每个块的充满程度在半满与全满之间。这样的索引不再需要溢出块。

在接下来的内容中，我们将讨论“B树”，但具体细节都针对B+树这一变体。其他类型的B树在习题中讨论。

13.3.1 B树的结构

正如其名称所暗示的那样，B树把它的存储块组织成一棵树。这棵树是平衡的，即从树根到树叶的所有路径都一样长。通常B树有三层：根、中间层和叶，但也可以是任意多层。为了对B树有一个直观的印象，可以先看一下图13-21、图13-22和图13-23，其中前两个图所示的为B树结点，而后一个图所示的为一个小而完整的B树。

对应于每个B树索引都有一个参数n，它决定了B树的所有存储块的布局。每个存储块存放n个查找键值和$n+1$个指针。在某种意义上讲，B树的存储块类似于13.1节讲述的索引块，只不过B树的块除了有n个键-指针对外，还有一个额外的指针。在存储块能容纳n个键和$n+1$个指针的前提下，我们把n取得尽可能大。

例13.19 假定我们的存储块大小为4096字节，且整数型键值占4字节，指针占8字节。要是不考虑存储块块头信息所占空间，那么我们希望找到满足$4n + 8(n + 1) \leqslant 4096$的最大整数值$n$。这个值是$n = 340$。 □

下面几个重要的规则限制B树的存储块中能出现的东西：

- 叶结点的键值是数据文件中键值的副本，这些键值从左向右有序地分布在叶结点上。
- 根结点中至少有两个指针被使用[⊖]。所有指针指向位于B树下一层的存储块。
- 叶结点中，最后一个指针指向它右边的下一个叶结点存储块，即指向下一个键值大于它的块。在叶块的其他n个指针当中，至少有$\lfloor (n+1)/2 \rfloor$个指针被使用且指向数据记录；未使用的指针可看做空指针且不指向任何地方。如果第i个指针被使用，则指向具有第i个键值的记录。
- 在内部结点中，所有的$n+1$个指针都可以用来指向B树中下一层的块。其中至少$\lceil (n+1)/2 \rceil$个指针被实际使用（但如果是根结点，则不管n多大都只要求至少两个指针被使用）。如果j个指针被使用，那该块中将有$j-1$个键，设为K_1，K_2，…，K_{j-1}。第一个指针指向B树的一部分，一些键值小于K_1的记录可在这一部分找到。第二个指针指向B树的另一部分，所有键值大小等于K_1且小于K_2的记录可在这一部分中，依此类推。最后，第j个指针指向B树的又一部分，一些键值大于等于K_{j-1}的记录可以在这一部分中找到。注意：某些键值远小于K_1或远大于K_{j-1}的记录可能根本无法通过该块到达，但可通过同一层的其他块到达。 633

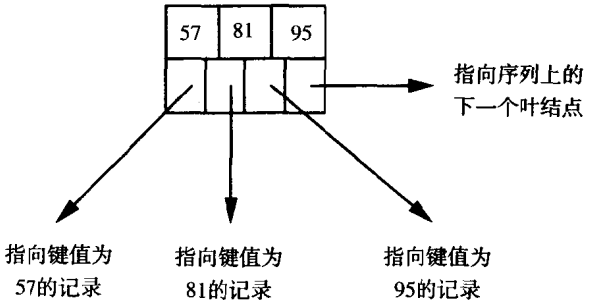


图13-21 典型的B树叶结点

例13.20 在这个例子和其他B树实例中，我们设$n=3$。也就是说，块中可存放3个键值和4个指针，这是一个不代表通常情况的小数字。键值为整数。图13-21所示为一个完全使用的叶结点。其中有三个键值57、81和95。前三个指针指向具有这些键值的记录。而最后一个指针，指向右边键值大于它的下一个叶结点，这正是叶结点中通常的情况。如果该叶结点是序列中的最后一个，则该指针为空。

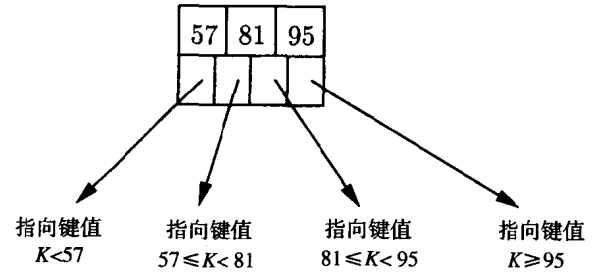


图13-22 典型的B树内部结点

⊖ 从技术上来讲，整个B树的块只有一个指针也是可能的，因为它可能是只有一个记录的数据文件的索引。在这种情况下，整个B树既是根块又是叶块，且这个块只有一个键值和一个指针。在下面的描述中我们忽略这种平凡的情况。

叶结点不必全部充满，但在我们这个例子中，$n = 3$，故叶结点至少要有两个键-指针对。也就是说，图13-21中的键值95和第三个指针可以没有，该指针标有“至键值为95的记录”。

图13-22所示为一个典型的内部结点。其中有三个键值，与我们在叶结点的例子中所选的一样：57、81和95[⊖]。该结点中还有四个指针。第一个指针指向B树的一部分，通过它我们只能到达键值小于第一个键值即57的那些记录。第二个指针通向键值介于该B树块第一个键值和第二个键值之间的那些记录，第三个指针对应键值介于该块第二个键值和第三个键值之间的那
634 些记录，第四个指针将我们引向键值大于该块中第三个键值的那些记录。

同叶结点的例子一样，内部结点的键和指针槽也没有必要全部占用。不过，当n=3时，一个内部结点至少要出现一个键和两个指针。元素缺失最极端的情形就是键值只有57，而指针也仅使用前两个，在这种情况下，第一个指针对应于小于57的键值，而第二个指针对应于大于等于57的键值。 □

例13.21 图13-23所示为一棵完整的三层B树[⊜]；其中使用例13.20中所描述的结点。我们假定数据文件的记录的键是2~47之间的所有素数。注意，这些值在叶结点中按顺序出现一次。所有叶结点都有两个或3个键-指针对，还有一个指向序列中下一叶结点的指针。当我们从左到右去看叶结点时，所有键都是排好序的。

根结点仅有两个指针，恰好是允许的最小数目，尽管至多可有4个指针。根结点中的某个键将通过第一个指针访问到的键值与通过第二个指针访问到的键值分隔开来。也就是说，不超
635 过12的键值可通过根结点的第一个子树找到；大于等于13的键值可通过第二个子树找到。

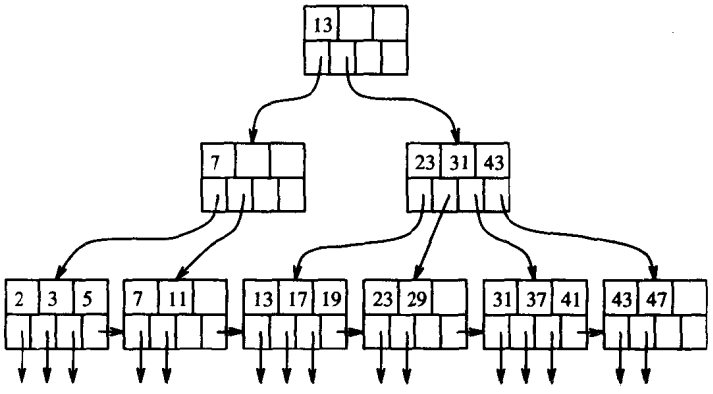


图13-23 B树

如果我们看根结点的第一个具有键值7的子结点，会发现它有两个指针，一个通向小于7的键，而另一个通向大于等于7的键。注意，该结点的第二个指针只能使我们找到键7和11，而非所有大于等于7的键，比如键13（虽然我们可以通过叶结点中指向下一个块的指针找到那些更大的键）。

最后，根结点的第二个子结点的4个指针槽都被使用。第一个指针将我们引向一些键值小于23的键，即13、17和19。第二个指针将我们引向键值大于等于23而小于31的所有键；第三个

⊖ 虽然键值一样，但图13-21所示的叶结点与图13-22所示的内部结点之间并没有什么联系。事实上，它们不可能出现在同一棵B树中。

⊜ 记住本节讨论的所有B树都是B+树，但在以后提到时我们将省略“+”号。

指针将我们引向键值大于等于31而小于43的所有键；而第四个指针将我们引向一些键值大于等于43的键（在这个例子中，是所有的键）。 □

13.3.2 B树的应用

B树是用来建立索引的一种强有力的工具。它的叶结点上指向记录的一系列指针可以起到我们在13.1节和13.2节学过的任何一种索引文件中指针序列的作用。下面是一些实例：

1. B树的查找键是数据文件的主键，且索引是稠密的。也就是说，叶结点中为数据文件的每一个记录设有一个键、指针对。该数据文件可以按主键排序，也可以不按主键排序。

2. 数据文件按主键排序，且B树是稀疏索引，在叶结点中为数据文件的每个块设有一个键、指针对。

3. 数据文件按非键属性排序，且该属性是B树的查找键。叶结点中为数据文件里出现的每个属性值K设有一个键、指针对，其中指针指向排序键值为K的记录中的第一个。

B树变体的另一些应用允许叶在结点上查找键重复出现[⊖]。图13-24所示即为这样一棵B树。这一扩展类似于我们在13.1.5节讨论的带重复键的索引。

如果我们确实允许查找键的重复出现，就需要稍微修改对内部结点中键的涵义的定义，我们曾在13.3.1节中讨论过这一定义。现在，假定一个内部结点的键为 $K_1, K_2, \cdots, K_n$，那么 K_i 将是从第i+1个指针所能访问的子树中出现的最小新键。这里的“新”，是指在树中第i+1个指针所指向的子树以左没有出现过K_i，但K_i在第i+1个指针指向的子树中至少出现一次。注意，在某些情况下可能不存在这样的键，这时K_i可以为空，但它对应的指针仍然需要，因为它指向树中碰巧只有一个键值的那个重要的部分。 636

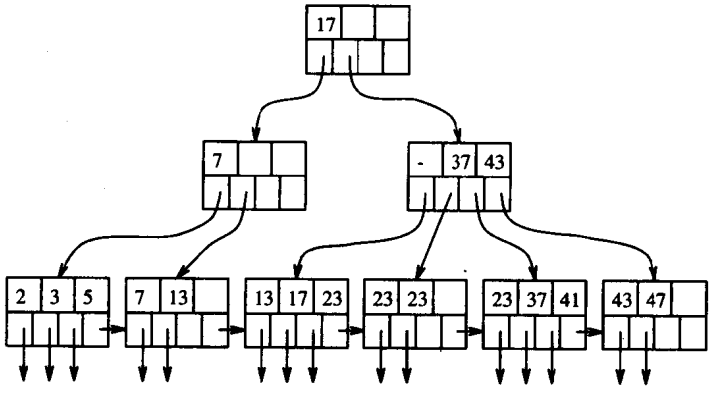


图13-24 一棵带重复键的B树

例13.22 图13-24所示的B树类似于图13-23，但有重复键值。具体来说：键11已被键13替换；键19、29和31全部被键23替换。这样就造成根结点的键是17而不是13。原因在于，虽然13是第二个子树根结点中最小的键，但它不是该子树的新键，因为它在根结点的第一个子树中出现过。

我们还需要对根结点的第二个子结点做些改变。第二个键改为37，因为它是第三个子结点（从左数第五个叶结点）的第一个新键。最有趣的是，第一个键现在为空，因为第二个子结点（第四个叶结点）根本就没有新键。换言之，如果查找某个键且到达根结点的第二个子结点，

⊖ 记住，如果“键”必须惟一，那么从这个意义上来说“查找键”不一定是“键”。

我们不会从该子结点的第二个子结点起开始查找。若是查找23或其他更小的值，我们应该从它的第一个子结点起开始查找，在那里我们将找到所需的记录（如果是17），或找到所需的记录的第一个（如果是23）。

注意：

- 查找13时我们不会到达根结点的第二个子结点，而是直接到第一个子结点中去查找。
- 如果查找介于24~36之间的某个键，我们会直接到第三个叶结点中查找。但当我们连一个所需键值都找不到时，我们就知道不必继续往右查找。举例来说，如果叶结点中存在键24，它要么在第四个叶结点上，这时根结点的第二个子结点中的空键将会被24替代；

637 要么在第五个叶结点上，这时根结点的第二个子结点的键37将被24替代。 □

13.3.3 B树中的查找

我们现在再回到最初的假定，即叶结点中没有重复键。同时我们还假设B 树是一个密集索引，因而数据文件中的每一个查找键也将出现在某个叶结点上。这个假定可以简化对B 树操作的讨论，但该假定对这些操作来说并非必不可少。特别地，稀疏索引的修改与我们在13.1.3节中介绍过的顺序文件上的索引修改相类似。

假设我们有一个B树索引并且想找出查找键值为 K 的记录。我们从根到叶递归查找，查找过程为：

基础：若处于叶结点，我们就在其键值中查找。若第 i 个键是 K，则第i个指针可让我们找到所需记录。

归纳：若处于某个内部结点，且它的键为$K_1, K_2, \cdots, K_n$，则依据在13.3.1节中给出的规则来决定下一步该对此结点的哪个子结点进行查找。也就是说，只有一个子结点可使我们找到具有键 K 的叶结点。如果 $K < K_1$，则为第一个子结点；如果 $K_1 \leqslant K < K_2$，则为第二个子结点，等等。在这一子结点上递归地运用查找过程。

例13.23 假定有一棵如图13-23所示的B树，且我们想找到查找键为40的记录。我们从根结点开始，其中有一个键13。因为13≤40，我们就沿着它的第二个指针来到包含键为23、31和43的第二层结点。

在这个结点中，我们发现31≤40<43，因而我们沿着第三个指针来到包含31、37和41的叶结点，如果数据文件中有键值为40的记录，我们就应该在这个叶结点中找到键40。既然我们没有发现键40，就可以断定在底层的数据块中没有键值为40的记录。

注意，要是查找键为37的记录，我们所做的决定都和上面一样，但当到达叶结点时，我们将找到键37。因为它是叶结点中的第二个键，因此我们沿着第二个指针可以找到键值为37的数据记录。 □

13.3.4 范围查询

B树不仅对搜寻单个查找键的查询很有用，而且对查找键值在某个范围内的查询也很有用。一般来说，范围查询在WHERE子句中有一个项，该项将查找键与单个值或多个值相比较，可用除“=”和“<>”之外的其他比较操作符。使用查找键属性K的范围查询例子如下：

```
SELECT *
FROM R
WHERE R.k > 40;
```

638

或者

```
SELECT *
FROM R
WHERE R.k >= 10 AND R.k <= 25;
```

如果想在B树叶结点上找出在范围 [a,b] 之间的所有键值，我们通过一次查找来找出键 a。不论它是否存在，我们都将到达可能出现 a 的叶结点，然后在该叶结点中查找键 a 或大于 a 的那些键。我们所找到的每个这样的键都有一个指针指向相应的记录，这些记录的键在所需的范围内。

如果我们没有发现大于 b 的键，我们就使用当前叶结点指向下一个叶结点的指针，并继续检查键和跟踪相应指针，直到我们

1. 找到一个大于 b 的键，这时我们停止查找；或者
2. 到了叶结点的末尾，在这种情况下，我们到达下一个叶结点且重复这个过程。

上面的查找算法当 b 为无穷时也有效；即项中只有一个下界而没有上界。在这种情况下，我们查找从键 a 可能出现的叶结点开始到最后一个叶结点的所有叶结点。如果 a 为 $-\infty$（即项中有一个上界而没下界），那么，在查找“负无穷”时，不论处于B树的哪个结点，我们总被引向该结点的第一个子结点，即最终将找到第一个叶结点。然后按上述过程查找，仅在超过键b时停止查找。

例13.24 假定我们有一棵如图13-23所示的B树，给定查找范围是（10，25）。我们查找键10，找到第二个叶结点，它的第一个键小于10，但第二个键11大于10。我们沿着它的相应指针找到键为11的记录。

因为第二个叶结点中已没有其他的键，我们沿着链找到第三个叶结点，其键为13，17和19。这些键都小于或等于25，因此我们沿着它们的相应指针检索具有这些键的记录。最后，移到第四个叶结点，在那里我们找到键23。而该叶结点的下一个键29超过了25，因而已完成我们的查找。这样，我们就检索出了键11到键23的五个记录。 □

13.3.5 B树的插入

当我们考虑如何插入一个新键到B树时，就会发现B树优于13.1.4节介绍的简单多级索引的一些地方。对应的记录可使用13.1节中介绍的任何方法插入到建有B树索引的数据文件中；这时，我们只考虑B树如何相应地修改。插入原则上是递归的： 639

- 设法在适当的叶结点中为新键找到空闲空间，如果有的话，就把键放在那里。
- 如果在适当的叶结点中没有空间，就把该叶结点分裂成两个，并且把其中的键分到这两个新结点中，使每个新结点有一半或刚好超过一半的键。
- 某一层的结点分裂在其上一层看来，相当于是要在这一较高的层次上插入一个新的键-指针对。因此，我们可以在这一较高层次上逆归地使用这个插入策略：如果有空间，则插入；如果没有，则分裂这个父结点且继续向树的高层推进。
- 例外的情况是，如果试图插入键到根结点中并且根结点没有空间，那么我们就分裂根结点成两个结点，且在更上一层创建一个新的根结点。这个新的根结点有两个刚分裂成的结点作为它的子结点。回想一下，不管n（结点中键的槽数）多大，根结点总是允许只有一个键和两个子结点。

当我们分裂结点并在其父结点中插入时，需要小心地处理键。首先，假定N是一个容量为n个键的叶结点，且我们正试图给它插入第（n+1）个键和它相应的指针。创建一个新结点M，该结点将成为 N 结点的兄弟，紧挨在N的右边。按键排序顺序的前$\lceil(n+1)/2\rceil$个键-指针对保留

在结点N中；而其他的键-指针对移到结点M中，注意，结点M和结点N中都有足够数量的键-指针对，即至少有$\lfloor(n+1)/2\rfloor$个这样的键-指针对。

现在，假定N是一个容量为n个键和$n+1$个指针的内部结点，并且由于下层结点的分裂，N正好又被分配给第（$n+2$）个指针。我们执行下列步骤：

1. 创建一个新结点M，它将是N结点的兄弟，且紧挨在N的右边。

2. 按排序顺序将前$\lceil(n+2)/2$个指针留在结点n中，而把剩下的$\lfloor(n+2)/2$个指针移到结点M中。

3. 前$\lceil n/2\rceil$个键保留在结点N中，而后$\lfloor n/2\rfloor$个键移到结点M中。注意，在中间的那个键总是被留出来，它既不在结点N中也不在结点M中。这一留出的键K指明通过M的第一个子结点可访问到最小键。尽管K不出现在N中也不出现在M中，但它代表通过M能到达的最小键值，从这种意义上来说它与M相关联。因此，K将会被结点N和M的父结点用来划分在这两个结点之间的查找。

例13.25 让我们在图13-23所示的B树中插入键40。根据13.3.3节中的查找过程找到插入的叶结点。如例13.23一样，我们找到第五个叶结点来插入。由于n=3，而该叶结点现在有四个
640 键-指针对：31、37、40和41，所以我们需要分裂这个叶结点。首先是创建一个新结点，并把两个最大的键40和41以及它们的指针移到新结点。图13-25表示了这个分裂。

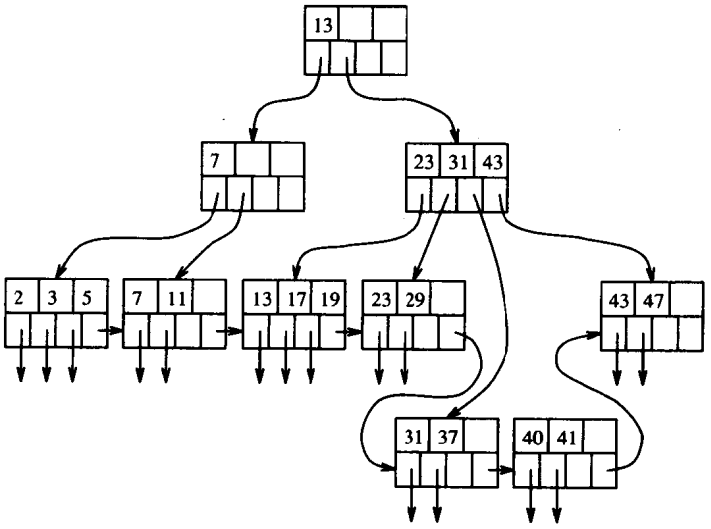


图13-25 键40插入之初

注意，虽然我们现在把这些结点显示在四排，但对树而言还是只有三层，而七个叶结点占据了图中的后两排。这些叶结点通过各自的最后一个指针链接起来，仍形成了一条从左到右的链。

我们现在必须插入一个指向新叶结点（具有键40和41的那个结点）的指针到它上面的那个结点（具有键23、31和43的那个结点），还必须把该指针与键40关联起来，因为键40是通过新叶结点可访问到的最小键。很不巧，分裂结点的父结点已满，它没有空间来存放别的键或指针。因此，它也必须分裂。

我们开始先找到指向后五个叶结点的指针和表示这些叶结点中后四个的最小键的键列表。也就是说，我们有指针P_1、P_2、P_3、P_4和P_5指向这些叶结点，它们的最小键分别是13、23、31、

40和43，并且我们用键序列23、31、40和43来分隔这些指针。前三个指针和前两个键保留在被分裂的内部结点中；而后两个指针和后一个键放到一个新结点中。剩下的键是40，表示通过新结点可访问到最小键。

插入键40后的结果如图13-26所示。根结点现在有三个子结点，最后两个是分裂的内部结点。注意，键40标志着通过分裂结点的第二个子结点可访问到的最小键，它被安置在根结点中，用来区分根结点的第二个子结点和第三个子结点的键。 □ 641

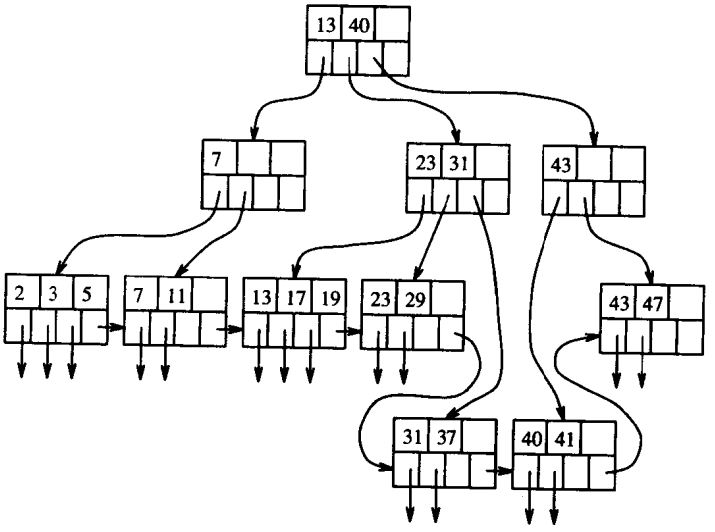


图13-26 键40插入完成后

13.3.6 B树的删除

如果我们要删除一个具有给定键 K 的记录，必须先定位该记录和它在B树叶结点中的键-指针对。正如13.3.3节所述，这部分删除过程主要是查找。然后我们删除记录本身并从B树中删除它的键-指针对。

如果发生删除的B树结点在删除后至少还有最小数目的键和指针，那就不需要再做什么[⊖]。但是，结点有可能在删除之前正好具有最小的充满度，因此在删除后，对键数目的约束就被违背了。这时，我们需要为这个键的数目仅次于最小数目的结点N做下面两件事之一，其中有一种情况需要沿着树往上递归地删除。

1. 如果与结点 N 相邻的兄弟中有一个键和指针超过最小数目，那么它的一个键-指针对可以移到结点 N 中并保持键的顺序。结点 N 的父结点的键可能需要调整以反映这个新的情况。例如，如果结点 N 的右边兄弟 M 可提供一个键和指针，那么从结点 M 移到结点 N 的键一定是 642
结点 M 的最小键。在结点 M 和结点 N 的父结点处有一个表示通过 M 可访问到的最小键，该键必须被提升。

2. 最困难的情况是当相邻的两个兄弟中没有一个能提供额外的键给结点N时。不过，在这种情况下，我们有结点N和它的一个兄弟结点 M，其中一个的键数少于最小数，而另一个的键数刚好为最小数。因此，它们合在一起也没有超过单个结点所允许的键和指针数（这就是为什

⊖ 如果具有叶结点中最小键的数据记录被删除，那么我们可选择在该叶结点的某个祖先上将某个合适的键增大，但不一定非这样做不可；所有的查找仍能找到正确的叶结点。

么B树结点最小允许的充满程度为一半的原因）。我们合并这两个结点，实际上就是删除它们中的一个。我们需要调整父结点的键，然后删除父结点的一个键和指针。如果父结点现在足够满，那我们就完成了删除操作，否则，我们需要在父结点上递归地运用这个删除算法。

例13.26 让我们从图13-23所示最初的B树开始，即在键40插入之前。假定我们删除键7。该键在第二个叶结点中被找到。我们删除该键、该键对应的指针以及指针指向的记录。

不巧的是，第二个叶结点现在只剩下一个键，而我们需要每个叶结点至少有两个键。但该结点左边的兄弟，即第一个叶结点，有一个额外的键-指针对，这就帮了我们的大忙。我们因此可以将它的最大键以及它的相应指针移到第二个叶结点。产生的B树如图13-27所示。注意，因为第二个叶结点的最小键现在是5，所以前两个叶结点的父结点的键从7改为5。

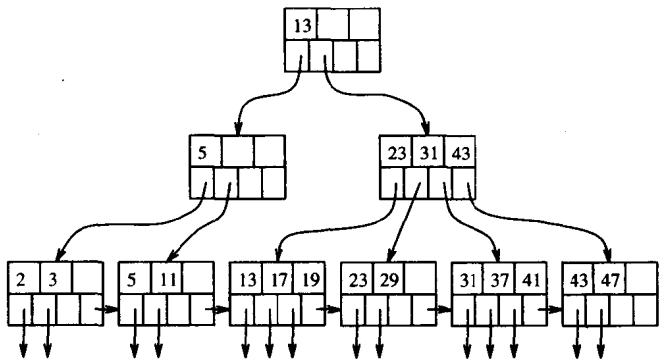


图13-27 键7的删除

下一步，假定我们删除键11。这个删除对第二个叶结点产生同样的影响；又一次把它的键
643 数减少到低于最小数。不过，这次我们不能从第一个叶结点借键，因为后者的键数也到了最小数。另外，它的右边没有兄弟，也就无处可借[⊖]。这样，我们需要合并第二个叶结点和它的兄弟，即第一个叶结点。

前两个叶结点剩下的三个键-指针对可以放在一个叶结点中，因此，我们把键5移到第一个叶结点并删除第二个叶结点。父结点中的键和指针需要进行调整，以反映它的子结点的新情况；具体地说，它的两个指针被换成一个指针（指向剩下的叶结点）且键5不再有用，也被删除。现在的情况如图13-28所示。

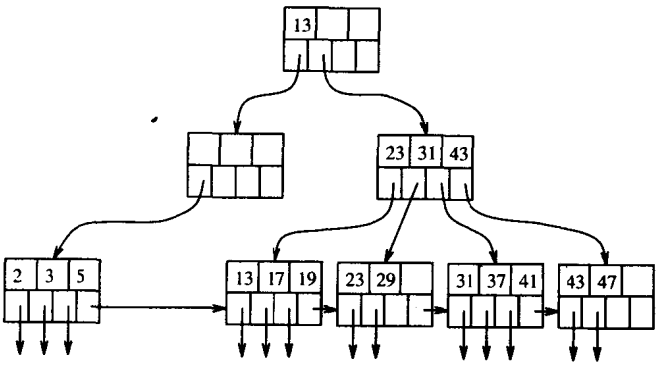


图13-28 键11删除之初

⊖ 注意：它右边的键为13、17和19的叶结点不是它的兄弟，因为它们有不同的父结点。不论怎样，我们还是可以从那个结点“借”键的，但那样的话，调整键的算法将涉及整个树，因而使算法变得更复杂。我们把这一改进留作习题。

但是，叶结点的删除给它的父结点即根结点的左子结点带来了负面的影响。正如我们在图13-28中所看到的那样，该结点现在没有键且只剩一个指针。因此，我们试图从与它相邻的兄弟那里获得一个额外的键和指针。这一次，我们碰到容易的情况，因为根结点的另一个子结点可以提供它的最小键和一个指针。

变化如图13-29所示。指向键为13、17和19的叶结点的指针从根结点的第二个结点移到了它的第一个子结点。我们还修改了内部结点的一些键。键13原来位于根结点且表示通过那个被转移的指针可访问到的最小键，现在需要放到根结点的第一个子结点中。另一方面，键23原来用来区分根结点第二个子结点的第一个和第二个子结点，现在表示通过根结点第二个子结点可访问到最小键，因此它被放到根结点中。 □ 644

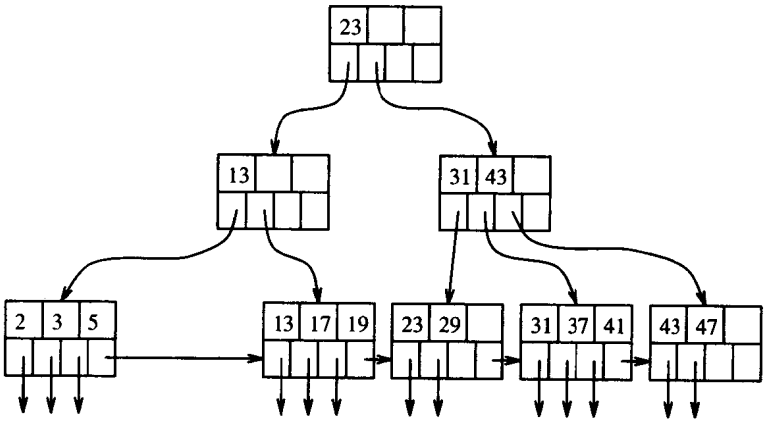


图13-29 键11删除完成后

13.3.7 B树的效率

B树使我们能实现记录的查找、插入和删除，而每个文件操作只需很少的磁盘I/O。首先我们注意到，如果每个块容纳的键数n相当大，比如10或更大，那么，分裂或合并块的情况将会很少。此外，这种操作必需时，绝大多数时候都被局限在叶结点，因此只有两个叶结点和它们的父结点受到影响。所以，我们基本上可以忽略B树重组的I/O开销。

然而，每次按给定查找键值查找记录都需要我们从根结点一直访问到叶结点以找到指向记录的指针。因为我们只读B树的块，所以磁盘I/O数将是B树的层数加上一次（对查找而言）或两次（对插入或删除而言）处理记录本身的磁盘I/O。我们肯定会这样问：B树到底有多少层？对于典型的键、指针和块大小来说，三层就足够了，除非数据库极大。因此，我们一般取3作为B树的层数。下面的例子说明了其原因。

例13.27 回忆一下我们在例13.19中的分析，我们当时确定每块可容纳示例数据的340个键-指针对。假若一般的块充满度介于最大和最小中间，即一般的块有255个指针。一个根结点，有255个子结点，有255^2=65 025个叶结点；在这些叶结点中，我们可以有255^3，即约1.66×10^7个指向记录的指针。也就是说，记录数小于等于1.66×10^7的文件都可以被3层的B树容纳。 □

不过，对于每次查找，我们甚至可以通过B树用比3次还少的磁盘I/O来实现。B树根结点块是永久地缓存在主存中的绝佳选择。如果这样，那么每次查找3层的B树只需两次磁盘读操作。实际上，在某些情况下，把B树的第二层结点块保存在缓冲区中也是合理的。这样，B树的查找就减少到一次磁盘I/O再加上处理数据文件本身所需的磁盘I/O。 645

我们是否该从B树中删除?

有一些B树的实现根本不对删除做修复。如果叶结点键和指针太少，这种情况也允许保留。其基本理由在于，大多数文件的发展比较平衡，尽管有时可能出现使键数刚好少于最小数删除操作。但该叶结点可能很快增长并且再次达到键-指针对的最小数。

此外，如果记录有来自B树索引外的指针，那么，需要用“删除标记”来替换记录，并且我们不想用任何方式删除B树中的指针。在某些情况下，如果可以保证所有对删除记录的访问都将通过B树，我们甚至可以在B树叶结点中指向记录的指针处留下删除标记。这样，该记录的空间就可以重新使用。

13.3.8 习题

习题13.3.1 假定存储块能放10个记录或者99个键和100个指针，再假定B树结点的平均充满程度为70%；即有69个键和70个指针。我们可以用B树作为几种不同结构的一部分。对下面描述的每种结构，确定：(1) 1 000 000个记录的文件所需的总块数；(2) 检索一个给定键值的记录所需的平均磁盘I/O数。可以假定最初在主存中不存在任何东西，并且查找键是记录的主键。

* a) 数据文件是按查找键排序的顺序文件，每块存放10个记录。B树为稠密索引。

b) 同(a)一样，但组成数据文件的记录没有特定顺序；每块存放10个记录。

c) 同(a)一样，但B树为稀疏索引。

! d) B树的叶结点中不放指向数据记录的指针，而是保存记录本身。每块可存放10个记录，但平均每个叶结点的充满度为70%，即每个叶结点存入7个记录。

646 * e) 数据文件是顺序文件，且B树是稀疏索引，但数据文件的每个基本块有一个溢出块。平均来讲，基本块是满的，而溢出块只半满。不过，记录在基本块和溢出块中没有特定的顺序。

习题13.3.2 假设查询是范围查询且匹配的记录有1000个，在这种情况下，重做习题13.3.1。

习题13.3.3 假定指针占4个字节，而键占12个字节，大小为16 384字节的块可存放多少个键和指针?

习题13.3.4 B树中(1)内结点和(2)叶结点的键和指针的最小数目在下列情况下分别是多少?

* a) $n = 10$；即每块可存放10个键和11个指针。

b) $n = 11$；即每块可存放11个键和12个指针。

习题13.3.5 在图13-23中执行下操作，描述那些引起树改变的操作所带来的变化。

a) 查找键值为41的记录。

b) 查找键值为40的记录。

c) 查找键值在20~30之间的所有记录。

d) 查找键值小于30的所有记录。

e) 查找键值大于30的所有记录。

f) 插入键值为1的记录。

g) 插入键值为14~16的所有记录。

h) 删除键值为23的记录。

i) 删除键值大于等于23的所有记录。

! **习题13.3.6** 我们提到图13-21所示的叶结点和图13-22所示的内部结点不可能出现在同一棵B树中。解释其原因。

习题13.3.7 当在B树中允许重复键值时，我们在这一节描述的查找、插入和删除的算法都要做一些必要的修改。给出下列情况所需进行的修改：

* a) 查找。

b) 插入。

c) 删除。

647

! **习题13.3.8** 在例13.26中我们提出，如果使用更复杂的维护内部结点键的算法，那么可以从右（或左）边的非兄弟结点中借键。描述一个合适的算法，它可以通过从同层相邻结点中借键来重新达到平衡，而不管这些相邻结点是否是键-指针对太多或太少的结点的兄弟结点。

习题13.3.9 如果我们使用这一节例子中的3个键和4个指针的结点，当数据文件中记录数如下时分别有多少不同的B树：

*! a) 6个记录。

!! b) 10个记录。

!! c) 15个记录。

*! **习题13.3.10** 假定我们的B树结点可存放3个键和4个指针，如同本节的例子一样。再假定当我们分裂叶结点时，把指针分成2和2。而当分裂内部结点时，前三个指针到第一个（左）结点，后两个指针到第二个（右）结点。我们从指向键分别为1、2和3的记录的指针所在叶结点开始，按序加入键值为4，5，6，…等的记录。在插入哪个键时B树将第一次达到四层？

!! **习题13.3.11** 考虑按B树组织的一个索引。叶结点一共包含指向N个记录的指针，且构成索引的每个块有m个指针。我们希望选择一个m值，使在具有下列特征的特定磁盘上查找时间最短：

1) 读一给定块到内存的时间大致为70+0.05m毫秒，其中70毫秒表示读操作的寻道和等待的时间，而0.05m毫秒是传输时间。也就是说，随着m的增大，块也将变大，因而把块读进内存也就需要更多的时间。

2) 一旦块在内存中，可用二分查找法来找到正确的指针，这样，在内存中处理一个块的时间为$a+b\log_2 m$毫秒，其中a、b为常量。

3) 主存时间常量a比磁盘的寻道和等待时间70毫秒小得多。

4) 索引是满的，因而每次查找需要检查块数为$\log_m N$。

回答下列问题：

a) 什么样的 m 值能最小化查找一个给定记录的时间。

b) 随着寻道和等待时间常量（70毫秒）的减少，会发生什么？例如，如果这个常量到一半，最优值m会怎样变化？

648

13.4 散列表

有许多涉及散列表的数据结构可用做索引。我们假定读者知道用作主存数据结构的散列表。在这种结构中有一个*散列函数*，它以查找键（我们可称之为散列键）为参数并计算出一个介于0到$B-1$的整数，其中B是桶的数目。桶数组，即一个序号从0到$B-1$的数组中包含B个链表的头，每一个对应于数组中的一个桶。如果记录的查找键为K，那么通过将该记录链接到桶号为

散列函数的选择

散列函数对键的“散列”应使得到的整数类似键的一个随机函数、因此。桶常常能分到相同数量的记录，正如我们将在13.4.4节中讨论的那样，这能改进访问一个记录的平均时间。另外，散列函数应该容易计算，因为我们要多次计算它。

- 当键为整数时，散列函数的一种常见选择是计算K/B的余数，其中K是键值，B是桶的数目。通常，B选为一个素数，尽管正如我们将从13.4.5节开始讨论的那样，将B选为2的幂也有其理由。
- 当键为字符串时，我们可以把每个字符看做一个整数来处理，把它们累加起来，并将总和除以B，然后取其余数。

$h(K)$的桶列表中来存储它，其中h是散列函数。

13.4.1 辅存散列表

有的散列表包含大量记录，记录如此之多，以至于它们主要存放在辅助存储器上，这样的散列表在一些细小而重要的方面与主存中的散列表存在区别。首先，桶数组由存储块组成而不是由链表头的指针组成。通过散列函数h散列到某个桶中的记录被放到该桶的存储块中。如果桶溢出，即它容纳不下所有属于它的记录，那么可以给该桶加一个溢出块链以存放更多的记录。

我们将假定，只要给一个i，桶i的第一个存储块的位置就可以找到。例如，主存中可以有一个指向存储块的指针数组，数组项以桶号为序号。另一种可能是把每个桶的第一个存储块存放到磁盘上某固定的、连续的位置，这样我们就可以根据整数 i 计算出桶 i 的位置。

例13.28 图13-30所示为一个散列表。为了使我们的图例易于处理，假定每个存储块只能存放两个记录，且 $B = 4$，即散列函数h
649 的返回值介于0～3之间。我们列出了一些位于散列表中的记录。在图13-30中，键值为字母 a～f。我们假定 $h(d) = 0$，$h(c) = h(e) = 1$，$h(b) = 2$且$h(a) = h(f) = 3$。因此，这六个记录在块中的分布如图所示。□

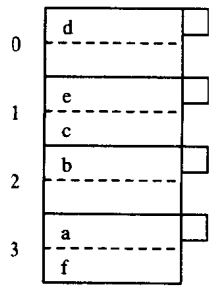


图13-30 散列表

注意，图13-30中所示每个存储块的右端都有一个小凸块，这个小凸块表示存储块块头中附加的信息。我们将用它来链接溢出块，并且从13.4.5节开始，我们将用它来保留存储块的其他重要信息。

13.4.2 散列表的插入

当一个查找键为K的新记录需要被插入时，我们计算$h(K)$。如果桶号为$h(K)$的桶还有空间，我们就把该记录存放到此桶的存储块中或在其存储块没有空间时存储到块链上的某个溢出块中。如果桶的所有存储块都没有空间，我们就增加一个新的溢出块到该桶的链上，并把新记录存入该块。

例13.29 假若我们给图13-30的散列表增加一个键值为g的记录，并且$h(g)=1$。那么，我们必须把记录加到桶号为1的桶中，也就是从上面数起的第二个桶。可是，该桶的块中已经有两个记录。因此，

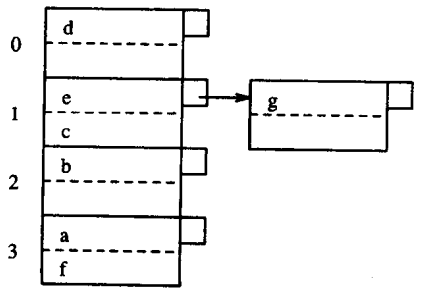


图13-31 为散列表的桶增加另外的一块

我们增加一个新块，并把它链到桶1的第一块上。键值为g的记录插入到这一块中，如图13-31所示。 □ 650

13.4.3 散列表的删除

删除查找键值为K的记录方式相同。我们找到桶号为 $h(K)$的桶且从中搜索查找键为K的记录，继而将找到的记录删除。如果我们可以将记录在块中移动，那么删除记录后，我们可选择合并同一链上的存储块[⊖]。

例13.30 图13-32所示为从图13-31的散列表中删除键值为c的记录后的结果。由前面可知，$h(c)$=1，因而我们到桶号为1的桶（即第二个桶）中去查看它的所有块，以找出键值为c的一条记录（或所有记录，当查找键不是主键时）。我们在桶1的链表的第一个存储块中找到了该记录。既然现在有可用空间，我们可以把键值为g的记录从链表的第二个存储块移到第一个存储块，并删除第二个存储块。

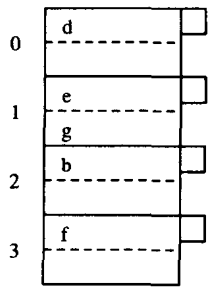


图13-32 散列表的删除结果

651

我们也做了删除键值为a的记录。对于这一键值，我们找到桶3，删除该记录，并把剩下的记录移到块的前部以使之紧凑。 □

13.4.4 散列表索引的效率

理想情况是存储器中有足够的桶，使绝大多数桶都只由单个块组成。如果这样，那么一般的查询只需一次磁盘I/O，且文件的插入和删除也只需两次磁盘I/O。这样的结果比直接用稀疏索引、稠密索引或B树好得多（尽管散列表不能像B树那样支持范围查询；参见13.3.4节）。

但是，如果文件不断增长，那么最终就会出现多数桶的链表中都有许多块的情况。如果这样，我们就需要在块的长链表中查找，每个块至少需要一次磁盘I/O。因此，我们就必须设法减少每个桶的块数。

到目前为止，我们学过的散列表都称为*静态散列表*，因为桶的数目B从不改变。但是，散列表中还有几种*动态散列表*，它们允许B改变，使B近似于记录总数除以块中能容纳的记录数所得到的商；也就是说，每个桶大约有一个存储块。我们将讨论两种这样的方法：

1. 13.4.5节的可扩展散列；和
2. 13.4.7节的线性散列。

第一种方法在认为B太小时即将其加倍，而第二种方法每当文件的统计数字表明B需要增加时即给B加1。

13.4.5 可扩展散列表

我们的第一种动态散列方法称为*可扩展散列表*。它在简单的静态散列表结构上主要增加了：

1. 为桶引入了一个间接层，即用一个指向块的指针数组来表示桶，而不是用数据块本身组成的数组来表示桶。

2. 指针数组能增长，它的长度总是2的幂，因而数组每增长一次，桶的数目就翻倍。

3. 不过，并非每个桶都有一个数据块；如果某些桶中的所有记录都可以放在一个块中，那么，这些桶可能共享一个块。

4. 散列函数 h 为每个键计算出一个 K 位二进制序列，该K值足够大，比如32。但是，无论 652

⊖ 合并一条链上的块随时都要冒风险，这是指当摇摆发生时，即当我们往一个桶里交替地插入或删除记录时，可能每一步都会导致块的创建或删除。

何时桶的数目都使用从序列第一位开始的若干位，此位数小于K，比如说是i位。也就是说，当i是使用的位数时，桶数组将有2^i个项。

例13.31 图13-33所示为一个小的可扩展散列表。为简单起见，我们假定K=4，即散列函数h只产生四位二进制序列。当前使用的只有其中一位，正如桶数组上方的框中i=1所标明的那样。因此，桶数组只有两个项，一个对应0，另一个对应1。

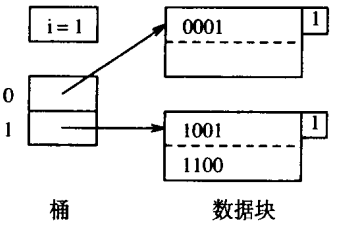


图13-33 可扩展散列表

桶数组项指向两个块。第一块存放当前所有查找键被散列成以0开头的二进制序列的记录；第二个块存放所有查找键被散列成以1开头的二进制序列的记录。为方便起见，我们显示的记录键是散列函数将这些键转换成的二进制位序列。因此，第一块有一个键被散列为0001的记录；而第二个块存放着键分别散列为1001和1100的记录。 □

我们应该注意到，图13-33中每个存储块的“小凸块”中都出现了数字1。这个数字其实出现在每个存储块的块头中，表明由散列函数得到的位序列中有多少位用于确定记录在该块中的成员资格。在例13.31的情况下，只用一个二进制位来确定所有的块和记录，但正如我们将要看到的那样，随着散列表的增长，不同块中需要考虑的位数可能不同。也就是说，桶数组的大小由我们当前正在使用的最大二进制位数来决定，但有些块可能使用较少的位数。

13.4.6 可扩展散列表的插入

可扩展散列表的插入开始时类似静态散列表的插入。为了插入键值为K的记录，我们计算出$h(K)$，取出这一二进制位序列的前i位，并找到桶数组中序号为这个i位的项。注意，因为i作
653 为散列数据结构的一部分保存，我们能确定i。

根据数据组中该项的指针找到某个存储块B。如果B中还有存放新记录的空间，我们就把新记录存入，而插入也就完成了。如果B中没有空间，那么视数字i的不同有两种可能，数字j表明散列值中有多少位于确定存储块B的成员资格（回忆一下，j的值可在图中每个存储块的“小凸块”中找到）。

1. 如果$j < i$，那么不必对桶数组做什么变化。我们：

a) 将块B分裂成两个存储块。

b) 根据记录散列值的第（j+1）位，将B中的记录分配到这两个存储块中，该位为0的记录保留在B中，而该位为1的记录则放入到新块中。

c) 把（j+1）存入这两个存储块的小凸块中，以标明用于确定成员资格的二进制位数。

d) 调整桶数组中的指针，使原来指向块B的项指向块B或新块，这由项的第（j+1）位决定。

注意，分裂块B可能解决不了问题，因为有可能块B中所有记录将分配到由B分裂成的两个存储块的其中一个中去。如果这样，我们需要对仍然太满的块用下一个更大的j值重复上述过程。

2. 如果$j = i$，那么我们必须先将i加1。我们使桶数组长度翻了一倍，因此数组中现在有2^{i+1}。假定w是以前的桶数组中作为某项序号的i位二进制位序列。在新桶数组中，序号为w0和w1（即分别用0和1扩展w所得到的数）的项都指向原w项指向的块。也就是说，这两个新项共享同一个存储块，而存储块本身没有变化。该块的成员资格仍然按原先的位数确定。最后，我们继续像第一种情况中那样分裂B。由于i现在大于j，所以满足第一种情况。

例13.32 假如在图13-33的表中插入一个键值散列为1010序列的记录。因为第一位是1，所以该记录属于第二个块。然而，该块已满，因此需要分裂。这时我们发现$j=i=1$，因此首先需要将桶数组加位，如图13-34所示。图中我们已将i设为2。

注意，以0开头的两个项都指向存放键值散列序列以0开头的记录的那个存储块，且该存储块的“小凸块”中数字仍然为1，这表明该块的成员资格只由位序列的第一位确定。但是，位序列以1开头的记录存储块需要分裂，因此我们把这一块中的记录分到以10开头和11开头的两个存储块中。在这两个存储块中的小凸块中有一个2，表示成员资格用位序列的前两位来确定。幸好，分裂是成功的；既然两个新块都至少有一个记录，我们就不用进行递归分裂。

现在，假定我们插入键值分别为0000和0111的记录。这两个记录都属于图13-34中第一个存储块，于是该块溢出。因为该块中只用一位来确定其成员资格，而$i=2$，所以我们就不用调整桶数组。我们只需分裂该块，让0000和0001留在该块，而将0111存放到新块中，桶数组中由01项改为指向新块。这一次我们又很幸运，所有记录没有全分配到一个块中，所以我们不必递归地分裂。

假若现在要插入一个键值为1000的记录。对应10的块溢出。由于它已经使用两位来确定其成员资格，这时需要再次分裂桶数组，并且把i设为3。图13-35给出了这时的数据结构。注意，图中对应10的块被分裂成100的块和101的块，而其他块仍只使用两位来确定成员资格。 □

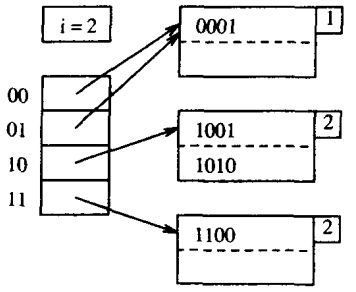
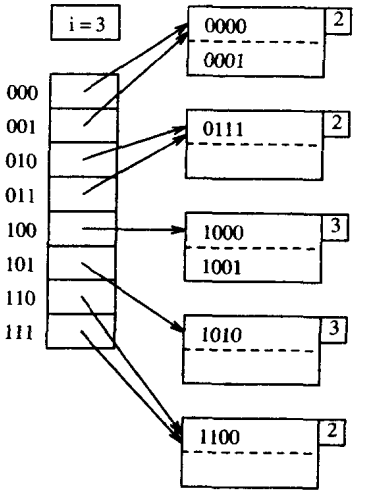


图13-34 使用两位散列函数值的散列表



654

655

图13-35 使用3位二进制序列的散列表

13.4.7 线性散列表

可扩展散列表有一些重要的好处。最大的好处在于，当查找一个记录时，我们总是只需要查找一个数据块。我们还需要查找到一个桶数组的项，但如果桶数组小到可以存放在主存中，那么访问桶数组就不需要进行磁盘I/O。然而，可扩展散列表也有一些缺点：

1. 当桶数组需要翻倍时，要做大量的工作（当i很大时）。这些工作会阻碍对数据文件的访问，或是使某些插入看来花费很长的时间。

2. 当桶数翻倍后，它在主存中可能就装不下了，或者把其他的一些我们需要保存在主存的数据挤出去。其结果是，一个运行良好的系统可能突然之间每个操作所需的磁盘I/O开始大增，并且出现明显的性能下降。

3. 如果每块的记录数很少，那么很有可能某一块的分裂比在逻辑上讲需要分裂的时间提前许多。例如，如果像我们使用的例子一样，块可存放两个记录，即使记录的总数远小于2^{20}，这也有可能出现三个记录的前20位二进制位序列。在这种情况下，我们将不得不使用$i=20$和一百万个桶数组项，尽管存有记录的块数远小于一百万。

另一种策略称为*线性散列*，其中桶的增长较为缓慢。在线性散列中我们发现的新要点为：

- 桶数n的选择总是使存储块的平均记录数保持与存储块所能容纳的记录总数成一个固定的

比例，如80%。

- 由于存储块并不总是可以分裂，所以允许有溢出块，尽管每个桶的平均溢出块数远小于1。
- 用来做桶数组项序号的二进制位数是$[\log_2 n]$，其中n是当前的桶数。这些位总是从散列函
656 数得到的位序列的右（低位）端开始取。
- 假定散列函数值的i位正在用来给桶数组项编号，且有一个键值为K的记录想要插入到编号为$a_1a_2\cdots a_i$的桶中；即$a_1a_2\cdots a_i$是$h(K)$的后i位。那么，把$a_1a_2\cdots a_i$当作二进制整数，设它为m。如果$m<n$，那么编号为m的桶存在并把记录存入该桶中。如果$n\leqslant m<2^i$，那么桶m还不存在，因此我们把记录存入桶$m-2^{i-1}$，也就是当我们把a_1（它肯定是1）改为0时对应的桶。

例题13.33 图13-36所示为一个$n=2$的线性散列表。我们目前只用散列值的一位来确定记录所属的桶。按照例13.31建立的模式，我们假定散列函数产生四位，并且用将散列函数作用到记录的查找键上所产生的值来表示记录。

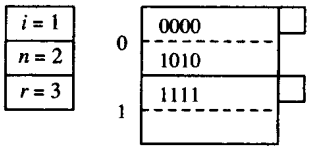


图13-36 线性散列表

我们在图13-36中看到两个桶，每个桶包含一个存储块，桶的编号为0和1。所有散列值以0结尾的记录存入第一个桶，而所有散列值以1结尾的记录存入第二个桶。

参数i（当前被使用的散列函数值的位数）、n（当前的桶数）和r（当前散列表中的记录总数）也是这一结构的一部分。比率r/n将受到限制，使一般的桶都只需要约一个磁盘存储块。在选择桶数n时，我们采用的策略是使数据文件中记录的个数不超过$1.7n$，即$r\leqslant 1.7n$。也就是说，由于每个存储块存放两个记录，桶的平均充满程度不会超过存储块容量的85%。 □

13.4.8 线性散列表的插入

当插入一个新记录时，我们通过在13.4.7节提出的算法来确定它所属的桶。也就是说，我们计算$h(K)$，其中K是记录的键，并确定$h(K)$序列后面用做桶号的正确位数。我们把记录或者放入该桶，或者（在桶号大于等于n时）放入把第一个二进制由1改为0后确定的桶中。如果桶中没有空间，那么我们创建一个溢出块，并把它链到那个桶上，并且记录就存入该溢出块中。

每次插入，我们都用当前的记录总数r的值跟阈值r/n相比，若比率太大，就增加下一个桶
657 到线性散列表中。注意，新增加的桶和发生插入的桶之间没有任何联系！如果新加入的桶号的二进制表示为$1a_2a_3\cdots a_i$，那么我们就分裂桶号为$0a_2a_3\cdots a_i$的桶中的记录，根据记录的后i位值分别存入这两个桶。注意，这些记录的散列值都以$a_2a_3\cdots a_i$结尾，并且只有从右数起的第i位不同。

最后一个重要的细节是当n超过2^i时的情况。这时，i递增1。从技术上来讲，所有桶的桶号都要在它们的位序前面增添一个0，但由于这些位序列被解释成整数，因而就不需要做任何物理上的变化，还是保持原样。

例13.34 我们继续例13.33，考虑插入键值散列为0101的记录时的情况。因为位序列以1结尾，记录属于图13-36中的第二个桶。桶中有空间，因而不用创建溢出块。

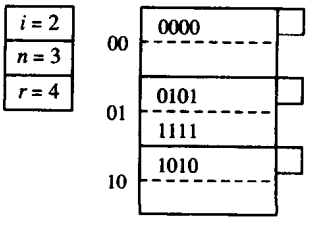


图13-37 增加第三个桶

但是，由于那里现在有四个记录在两个桶中，超过了1.7这一比率，因此我们必须把n提高到3。因为$\lceil \log_2 3 \rceil = 2$，我们应该开始考虑把桶0和1改成桶00和01，但不需要对数据结构做任何改变。我们增加下一个桶到散列表中，该桶编号为10，接着，分裂桶00，桶00的序号只有第一位与新

加的桶不同。在分裂桶时，键值散列为0000的记录保留在00桶，因为它以00结尾；而键值散列为1010的记录存入桶10，因为它以10结尾，所产生的散列表如图所示13-37。

下面，假定我们增加一个键值散列为0001的记录。记录最后两位为01，且01桶目前存在，我们把记录存入该桶。不巧的是，该桶的块已经装满，所以我们增加一个溢出块。这三个记录被分配在这个桶的两个块中；我们选择按散列键的数值顺序来保存它们，但这个顺序并不重要。由于该散列表中记录与桶的比率为5/3，小于1.7，故我们不需创建新桶。所产生的散列表如图13-38所示。

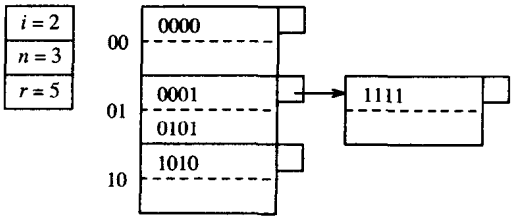


图13-38 必要时使用溢出块

最后，考虑插入键值散列为0111的记录。该记录最后两位为11，但桶11还不存在，因此我们把记录改为存入桶01，该桶号只是在第一位上与桶11不同，不是1而是0。新记录存入到该桶中的溢出块中。

658

但是，该散列表的记录与桶的比率已超过1.7，因此，我们必须创建一个编号为11的新桶，该桶碰巧是新记录所需的桶。我们分裂桶01中的四个记录，散列值为0001和0101的记录保留在桶01，而散列值为0111和1111的记录存入新桶。因为桶01现在只有两个记录，我们可以删除其溢出块。现在，散列表如图13-39所示。

注意，当下次插入记录到图13-39中时，我们将会使记录与桶的比率超过1.7。那时，我们将把n提到5，并且i变成3。 □

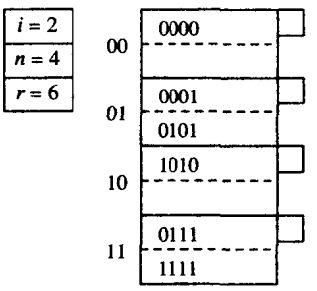


图13-39 增加第四个桶

例13.35 线性散列表的查询依照我们所描述的选择插入记录所属桶的过程。如果我们希望查找的记录不在该桶中，那么别的地方也不会有所需记录。举例来说，考虑图13-37中的情形，其中 $i = 2$且 $n = 3$。

首先，假定我们想查找键值散列为1010的记录。由于 $i = 2$，我们查看最后两位10，把它们解释为二进制整数，即 $m = 2$。因为 $m < n$，则编号为10的桶存在，因而我们到该桶中查找。注意，不能因为我们找到一个键值散列为1010的记录就以为找到了所需的记录，我们需要检查记录的整个键来确定是否所需记录。

接着，我们考虑查找键值散列为1011的记录。现在，我们必须查看编号为11的桶。由于该桶号作为二进制值整数 $m = 3$，并且 $m \geq n$，所以桶11不存在。我们通过把第一位1改为0后重新定位到桶01中。可是，桶01中不存在键值散列为1011的记录，因而我们所需查找的记录肯定不在散列表中。 □

659

13.4.9 习题

习题13.4.1 假若在图13-30的散列表中发生下列插入和删除，请说明将产生什么情况：

1) 记录g到记录j分别插入桶0到桶3。
2) 记录a和b记录被删除。
3) 记录k到n分别插入桶0到桶3。
4) 记录c和d被删除。

习题13.4.2 我们没有讨论在线性散列表或可扩展散列表中删除操作如何实现。定位被删除记录的机制应该是显而易见的。你认为应用什么方式实行删除操作？特别地，如果删除后表变小，允许压缩某些块，那么重构散列表有什么优缺点？

! **习题13.4.3** 本节内容假定索引键是惟一的。不过，只需对这些技术稍微做些修改就可用来处理有重复值的查找键。描述删除、查询和插入算法需做的修改，并说明当重复值出现在下列结构中时带来的主要问题：

* a) 简单散列表。

b) 可扩展散列表。

c) 线性散列表。

! **习题13.4.4** 实际中有些散列函数并不像理论上那样好。假定我们在整数键值i上定义一个散列函数$h(i)=i^2 \bmod B$

* a) 如果B=10，该散列函数会出现什么问题？

b) 如果B=16，该散列函数又有什么好处？

c) 该散列函数对哪些B值有用？

习题13.4.5 在每个存储块可存放n条记录的可扩展散列表中，何时会出现需要递归处理溢出块的情况；即块中的所有记录对应到分裂所产生的两个块中的同一块。

660

习题13.4.6 假定键值散列为4位序列，就像本节中可扩展散列表和线性散列表的例子一样。但是，假定块中可存放三个记录而非两个记录。如果开始时散列表中有两个空存储块（对应于0和1），请给出插入键值如下的记录后的结构：

* a) 0000，0001，…，1111，且散列方法是可扩展散列。

b) 0000，0001，…，1111，且散列方法是线性散列，其充满度阈值为100%。

c) 1111，1110，…，0000，且散列方法是可扩展散列。

d) 1111，1110，…，0000，且散列方法是线性散列，其充满度阈值为75%。

* **习题13.4.7** 假定我们使用可扩展散列表或线性散列表模式，但是有指向记录的外部指针。这些指针妨碍了记录在块之间的移动，而这些散列模式中有时需要如此。提出几种能修改结构的方法，从而允许外部指针。

!! **习题13.4.8** 线性散列模式中使用一个阈值常量c，使当前桶的数目n和当前记录总数r之间有 $r = ckn$ 的关系，其中k是每块可容纳的记录数。例如，在例13.33中，我们使用 $k = 2$和 $c = 0.85$，因而每个桶的记录数为1.7，即 $r = 1.7n$。

a) 为了方便起见，假定每个键恰好能按预期的次数出现[⊖]。作为c、k和n的函数，并包括溢出块在内，这种结构需要多少个存储块？

b) 键一般都不会平均分布，而给定键（或键后缀）的记录一般满足泊松分布。也就是说，如果λ是给定键后缀所期望的记录数，那么实际记录数为 i，其概率是 $e^{-\lambda}\lambda^i/i!$。在这种假定下，作为c、k和n的函数，期望使用的块数是多少？

*! **习题13.4.9** 假定有一个1 000 000条记录的文件，且我们想把它散列到一个有1000个桶的散列表中。每个存储块可存放100个记录，并且我们希望块尽可能满，但不允许两个桶共享一个块。存储这一散列表所需的最多和最少的存储块数各是多少？

661

⊖ 该假定并不意味着所有存储块都有数量相等的记录，因为有些桶所代表的键是其他桶的两倍。

13.5 小结

- *顺序文件*：几种简单的文件组织，其产生方式是将数据文件按某个查找键排序，并在该文件上建立索引。
- *稠密索引*：这种索引为数据文件的每个记录设一个键-指针对。这些键-指针对按它们的键值顺序存放。
- *稀疏索引*：这些索引为数据文件的每个存储块设一个键-指针对。与指针相对应的键为该指针所指向的存储块中第一个键值。
- *多级索引*：在索引文件上再建索引，在索引的索引上再建索引，等等，这在有时候是很有用的。高级索引必须是稀疏的。
- *文件的扩展*：随着数据文件和它的索引文件的增长，必须采取一些措施来为文件增加附加的存储块。为文件的基本块增加溢出块是一种可行的办法。在数据文件或索引文件的块序列表中插入更多的块，除非要求文件本身存放在连续的磁盘块中。
- *辅助索引*：即使数据文件没有按查找键K排序，我们也可在键K上建立索引。这样，索引必须是稠密的。
- *倒排索引*：文件与其包含的词之间的关系通常可通过一个词-指针对的索引结构来表示。指针指向“桶”文件的某个位置，该位置上有一个指向文件中词的出现的指针列表。
- *B树*：这些结构实质上是有着很好的扩充性能的多级索引。带有n个键和$n+1$个指针的存储块被组织成一棵树。叶结点指向记录。任何时候所有索引块都在半满与全满之间。
- *范围查询*：指查找键值在给定范围内的所有记录，有索引的顺序文件和B树索引可为这类查询提供便利，但散列表索引不能。
- *散列表*：同创建主存散列表一样，我们也可以基于辅存的存储块来建立散列表。散列函数将键值映射到桶，有效地将数据文件的记录分配到多个小组（桶）。桶用一个存储块和可能出现的溢出块表示。
- *动态索引*：如果一个桶中的记录太多，势必降低散列表的性能，因而随着时间推移，桶的数量可能需要增加。允许合理增加桶的两种重要方法是可扩展散列和线性散列。它们都首先将键值散列到一个长位串，然后使用其中若干位来决定记录所属的桶，位的数目 662
是可变的。
- *可扩展散列*：这种方法允许在存在记录数太多的桶时将桶的数目加倍。它使用指向块的指针数组来表示桶。为了避免块过多，几个桶可以用同一个块表示。
- *线性散列*：这种方法每当桶中的记录比例超出阈值时增加一个桶。由于单个桶的记录不会引起表的扩展，所以在某些情形下需要溢出块。

13.6 参考文献

B树最初是由Bayer和McCreight[2]提出来的。与本章里描述的B树不同，B树的内部结点和叶结点都有指向记录的指针。[3]是对B树变体的一个综述。

散列用做数据结构可追溯到Peterson[8]。可扩展散列在[4]中提出，而线性散列则来自[7]。Knuth写的书[6]包含了有关数据结构方面的许多内容，包括选择散列函数和设计散列表的技术以及关于B树变体的许多思想。B+树（内部结点没有键值）的明确陈述出现在[6]的1973版中。

[9]中讨论了辅助索引和文档检索的技术，而[1]和[5]是有关文本文档索引方法的综述。

1. R. Baeza-Yates, "Integrating contents and structure in text retrieval," *SIGMOD Record* **25**:1 (1996), pp. 67–79.

2. R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica* **1**:3 (1972), pp. 173–189.

3. D. Comer, "The ubiquitous B-tree," *Computing Surveys* **11**:2 (1979), pp. 121–137.

4. R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashing — a fast access method for dynamic files," *ACM Trans. on Database Systems* **4**:3 (1979), pp. 315–344.

5. C. Faloutsos, "Access methods for text," *Computing Surveys* **17**:1 (1985), pp. 49–74.

6. D. E. Knuth, *The Art of Computer Programming, Vol. III, Sorting and Searching, Third Edition*, Addison-Wesley, Reading MA, 1998.

7. W. Litwin, "Linear hashing: a new tool for file and table addressing," 663 *Proc. Intl. Conf. on Very Large Databases* (1980) pp. 212–223.

8. W. W. Peterson, "Addressing for random access storage," *IBM J. Research and Development* **1**:2 (1957), pp. 130–146.

9. G. Salton, *Introduction to Modern Information Retrieval*, McGraw-Hill, 664 New York, 1983.

第14章 多维索引和位图索引

到目前为止，我们讨论的所有索引都是一维的。也就是说，它们使用单个查找键，并且按给定的查找键值来检索记录。我们把查找键想像成单个属性或字段。不过，一个建立在由多个字段组成的查找键上的索引仍然可以被认为是一维的。如果想建立一个一维索引，它的查找键为字段（$F_1, F_2, \cdots, F_k$），那么我们可以把查找键值取成这些字段值的拼接：第一个为F_1，第二个为F_2，等等。我们可以使用特殊标记符号来分隔这些字段值，从而使查找键值和字段（$F_1, F_2, \cdots, F_k$）值的列表之间的联系不产生歧义。

例14.1 若字段F_1和F_2分别是字符串和整数，并且#不会在字符串中出现，那么F_1='abcd'和F_2=123的组合可以被表示成字符串'abcd#123'。 □

在第13章中，我们利用一维键空间的方式有几种：

- 顺序文件上的索引和B树都利用了使所有键具有单一、有序的顺序这一点。
- 散列表要求查找的查找键值是完全已知的。如果查找键由几个字段组成，那么即使只有一个字段不知道，我们也无法运用散列函数，而必须查找所有的桶。

许多应用要求我们将数据视为存在于二维或更高维的空间中。这样的应用中有一些能被传统的DBMS系统支持，但也有一些专为多维应用设计的系统。这些专用系统的不同之处的一个重要方面在于它们使用一些支持在普通SQL应用中不常见的某些种类的查询的数据结构。14.1节将介绍得益于支持多维数据和多维查询的索引的典型查询。然后，在14.2节和14.3节中我们讨论下列数据结构： 665

1. *网格文件*，一维散列表的一种多维扩展。
2. *分段散列函数*，另一种把散列表思想引入多维数据的方式。
3. *多键索引*，属性A上的索引将引向另一属性B上的索引，对属性A的每个可能的值都有一个对应的属性B的索引。
4. kd树，把B树推广到点集的一种方式。
5. *四叉树*，结点的每个子结点代表较大空间的一个象限的多元树。
6. R树，B树的一种推广，适用于区域的集合。

最后，14.4节讨论一种称为*位图索引*的索引结构。这样的索引是在给定字段上具有给定值的记录位置的简洁编码。目前，它们开始在主要的商用DBMS中出现，而且它们有时是一维索引的很好的选择。不过，它们也可以用作回答某些多维查询的有力的工具。

14.1 需要多维的应用

我们将考虑两大类多维应用。一类具有*地理的特点*，其中的数据是两维或三维世界中的元素。另一类涉及维的更加抽象的概念。大略地说，关系的每个属性都可以看成一维，而所有的元组就是在由这些维上定义的空间中的点。

另外，本节中将分析传统的索引（如B树）如何用来支持多维查询。尽管在有些情况下它们足以解决问题，但是在一些例子中它们显然比更专门的结构差。

14.1.1 地理信息系统

地理信息系统用一个（通常是）两维的空间存储对象，对象可能是点或形状。通常，这些数据库是地图，其中存储的对象可能表示房子、路、桥、管道或其他物理对象。图14-1所示就是这样的一幅地图。

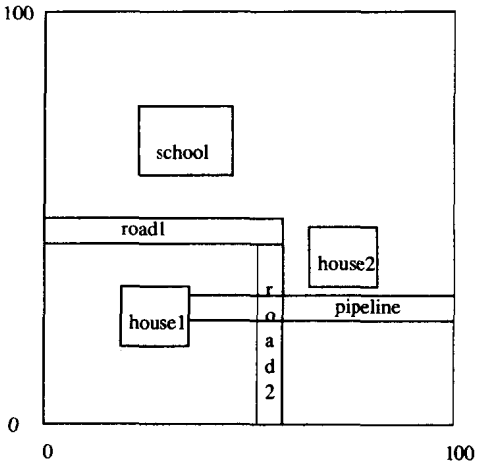


图14-1 二维空间中的对象

不过，地理信息系统也还有许多其他用途。例如，一个集成电路设计通常是由称为“图层”的特定材料的二维区域（常为矩形）图。同样，我们可以把屏幕上的窗口和图标看成是二维空间中的对象集合。

地理信息系统的查询并不是一般的SQL查询，尽管通过某些努力许多查询可以表达成SQL的形式。这类查询的例子如下：

1. 部分匹配查询。指定一维或多维上的值，并查找在这些维上匹配这些值的所有点。

2. 范围查询。给出一维或多维上的范围，并查找在这些范围内点的集合，或者在表示形状时查找出部分或全部形状在该范围内的形状的集合。这些查询推广了我们在13.3.4节所讨论的一维范围查询。

3. 最邻近查询。查找与给定点最近的点。例如，如果点表示城市，我们可能想找到一个与给定的小城市最近且人口超过100 000的城市。

4. Where-am-I查询。已知一个点，我们想知道该点所处的形状，若存在这样的形状的话。一个熟悉的例子是当你点击鼠标时，系统决定你点击的是哪个显示元素。

14.1.2 数据立方体

最近，一类称为数据立方体系统的DBMS得到发展，这类系统将数据视为存在于一个高维空间中。这些在20.5节会有更详细的讨论，但下面的例子表明了主要的思想。

许多公司为了决策支持应用而收集多维数据，在这些应用中，公司分析像销售等信息以便更好地了解公司的运作。例如，一个连锁店可能记录每一笔销售，包括：

1. 日期和时间。
2. 销售所在的商店。
3. 购买的商品。
4. 商品的颜色。
5. 商品的大小。

以及其他可能的销售特征。

通常，我们把这些数据看成一个关系，每个属性对应一个特征。这些属性可以被看做多维空间（“数据立方体”）的维，每个元组是该空间的一个点。然后，分析人员可以按照某些维对数据分组，并通过聚集汇总这些分组，一个典型的例子就是“按商店和月份给出1998年粉红色衬衫的销售情况”。

14.1.3 SQL多维查询

用传统的关系数据库来建立上面提到的应用并以SQL的方式实现上面提及的查询是可能的。下面是一些例子。

例14.2 假如我们希望回答关于二维空间点集的最邻近查询。可以把点表示成一个由实数对组成的关系：

```
Points(x, y)
```

即：有两个属性x和y，表示点的x坐标和y坐标。这里没有给出的关系Points的其他属性可能表示点的特征。

假定我们想找出离点（10.0，20.0）最近的点。图14-2所示的查询能找出最近的点或点集。该查询对于每一个点p，询问是否存在另一个离点（10.0，20.0）更近的点q。距离的比较是通过计算点（10.0，20.0）和点p、q之间的x坐标和y坐标的差的平方和来实现。注意，我们没有取和的平方根来得到实际距离，因为距离平方的比较同距离本身的比较是一样的。 □ 668

```
SELECT *
FROM POINTS p
WHERE NOT EXISTS(
    SELECT *
    FROM POINTS q
    WHERE (q.x-10.0)*(q.x-10.0) + (q.y-20.0)*(q.y-20.0) <
          (p.x-10.0)*(p.x-10.0) + (p.y-20.0)*(p.y-20.0)
);
```

图14-2 找出离点（10.0，20.0）最近的点

例14.3 矩形是地理系统中常用的形状。我们可以用多种方式表示矩形，最常见的一种是给出矩形的左下角和右上角的坐标。那么我们就可用关系Rectangles来表示矩形的集合，关系Rectangles由一个矩形标识ID和四个描述矩形的坐标以及其他我们希望记录的矩形的特征构成。在这个例子中，我们将使用关系

```
Rectangles(id,xll,yll,xur,yur)
```

其属性分别为：矩形标识ID，左下角的x坐标，左下角的y坐标和右上角的两个坐标。

图14-3所示查询是查找包含点（10.0，20.0）的矩形，WHERE子句中的条件是很明显的。对于一个包含点（10.0，20.0）的矩形，其左下角的x坐标必须位于10.0或10.0的左边，y坐标必须位于20.0或低于20.0；其右上角必须是$x \geqslant 10.0$和$y \leqslant 20.0$。 □

```
SELECT id
FROM Rectangles
WHERE xll <= 10.0 AND yll <= 20.0 AND
      xur >= 10.0 AND yur >= 20.0;
```

图14-3 找出包含给定点的矩形 669

例14.4 适合数据立方体系统的数据通常组织成一个*事实表*和若干*维表*，事实表给出被记录的基本元素（如每笔销售），而维表给出每一维的属性值的特征。例如，如果销售所属的商店是一维，则该维表可能包括地址、电话和商店经理的姓名。

在这个例子中，我们将只处理事实表，假定它包括在14.1.2节提到的维。即事实表是关系：

```
Sales (day, store, item, color, size)
```

查询“按日期和商店汇总粉红色衬衫的销售”如图14-4所示。它通过日期和商店这两维来分组组织销售，同时利用COUNT聚集操作符汇总其他维的情况。通过利用WHERE子句仅选择粉红衬

衫的元组，我们将注意力集中到所关心的点。 □

```
SELECT day, store, COUNT(*) AS totalSales
FROM Sales
WHERE item = 'shirt' AND
      color = 'pink'
GROUP BY day, store;
```

图14-4 汇总粉红色衬衫的销售

14.1.4 使用传统索引执行范围查询

现在，让我们来考虑第13章讨论的索引能在多大程度上帮助回答范围查询。为了简单起见，假定有两维。我们可以给x和y这两维中的每一维都建立一个辅助索引。为每一维都建立B树将会使获得每一维的某个范围内的值变得特别容易。

给定两维的范围，首先我们通过x的B树索引得到在x范围内的所有记录的指针。然后，我们通过y的B树索引得到在y范围内的所有记录的指针。最后，我们利用13.2.3节的思想求出这些指针的交集。如果主存能存放下这些指针，那么磁盘I/O的总数是每个B树中需要被检查的叶结点的数目加上沿着B树查找的少量I/O操作（见13.3.7节）。在此基础上，我们必须加上检索所有匹配记录所需的磁盘I/O，不论它们是多少。

670

例14.5 让我们考虑一个由1 000 000个点构成的假想点集，它们随机分布在一个x坐标和y坐标的范围都是0~1000的空间中。假定每个存储块可存放100个点，每个B树的叶结点大约有200个键-指针对（回想一下，并非在任何时候B树块的每个槽都必须占满）。我们将假定在x和y上都建有B树索引。

设想我们有一个范围查询，要求找出围绕该空间中心边长为100的正方形所包含的点。即$450 \leqslant x \leqslant 550$且$450 \leqslant y \leqslant 550$。利用$x$的B树索引，我们可以找到$x$值在该范围内的所有记录的指针，大约有100 000个指针，这个数量的指针应该可以在主存中存放下来。类似地，我们利用y的B树索引得到y值在该范围内的所有记录的指针，这大约也有100 000个。这两个指针集的交集大约有10 000个指针，而通过这10 000个指针找到的记录就构成了我们的答案。

现在，让我们来估计回答这个查询所需要的磁盘I/O数量。首先，正如我们在13.3.7节中提出的那样，把B树的根保存在主存是可行的。由于我们在每个B树中查找某个范围内的查找键值，且叶结点的指针按查找键排好了序，因此在访问每一维的100 000个指针时我们需要做的是检查一个中间层结点和所有包含所需指针的叶结点。因为假定叶结点包含约200个键-指针对，所以对每个B树我们需要查看约500个叶结点块。当再加上每个B树的一个中间结点时，我们需要的磁盘I/O总数为1002。

最后，我们需检索包含这10 000个所需记录的块。如果它们随机存放，我们必须预计它们将会在10 000个不同的块中。由于每一个块存放100个记录，整个100万条记录的文件假定被存储在10 000个块中，我们基本上需要查看数据文件的每个块。这样，至少在这个例子中，传统的索引在回答范围查询方面如果说有帮助也是很少。当然，如果范围很小，则两个指针集的交集的构造将使得我们把查找限制在数据文件块的一小部分中。 □

14.1.5 利用传统索引执行最邻近查询

我们使用的几乎所有的数据结构都允许我们来回答最邻近查询：在每一维上选定一个范围；执行范围查询；并且选择该范围内离目标最近的点。不巧的是，下面两件事情可能引起问题：

1. 所选定的范围内没有点。

2. 范围内最近的点可能总的来说不是最近的点。

让我们以例14.2的最邻近查询为背景，考虑每种情况，并使用例14.5假设的维x和y上的索 671
引。如果有理由相信与点（10.0, 20.0）距离为d的点存在，则我们可使用x的B树索引来得到所有x坐标在$10-d$和$10+d$之间的点记录的指针。然后我们使用y的B树索引来得到所有y坐标在$20-d$和$20+d$之间的点记录的指针。

如果在交集中有一个或多个点，并且记录了每个点的x坐标或y坐标（即索引的查找键），那么我们就有交集中所有点的坐标。我们因此可决定这些点中那个离点（10.0，20.0）最近并且只检索该记录。不巧的是，我们不能肯定在给定点的d范围内是否存在点，因此可能不得不用一个更大的d值来重复整个处理过程。

然而，即使在我们搜索的范围内存在点，也还存在该范围中最近的点离目标点（如本例中的（10.0, 20.0））的距离超过d的情况。这种情况如图14-5所示。如果情况是这样，那么必须扩大我们的范围并且再次搜索，从而确保没有更近的点存在。如果从目标点到目前已经发现的最近点的距离是 d'，且$d'>d$，则我们必须用 d' 代替 d 重新搜索。

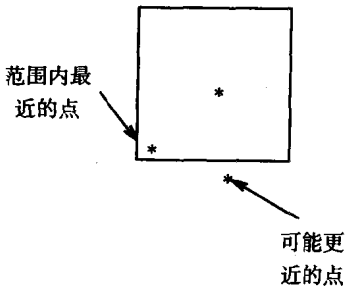


图14-5 范围内存在点，但可能在范围外存在着更近的点

例14.6 让我们考虑与例14.5同样的数据和索引。如果想找出与目标点P=（10.0，20.0）最邻近的点，我们可以选d=1。此外，平均每个面积单元有一个点，并且由于 $d=1$，我们找出围绕点P边长为2.0的正方形内的所有点，其预计的点数是4。

如果以范围查询$9.0\leqslant x\leqslant 11.0$检查 x 的B树，那么我们将找到大约2000个点，因而至少需要检索10个叶结点，也很有可能是11个（由于$x=9.0$的点很有可能不恰好在一个叶结点的开始）。像例14.5一样，我们可以把B树的根保存在内存中，因而对于中间的结点我们只需一次磁盘I/O；且对于叶结点，我们需要11次磁盘I/O 。另外还需要12次磁盘I/O来搜索y的B树索引中y坐 672
标在19.0~21.0之间的点。

如果我们在主存中对这将近4000个指针求交，我们将找到大约4个最邻近于点（10.0, 20.0）的候选记录。假定至少有一个，我们能从与指针相关的x坐标和y坐标决定哪一个是最邻近点。检索这个所需记录需要另一次磁盘I/O，这样完成整个查询总的磁盘I/O数为25。不过，如果d=1的正方形中没有点，或者最近点离目标点的距离大于1，那我们不得不用一个更大的d来重复搜索过程。 □

从例14.6得出的结论是，传统索引用于最邻近查询可能并不很糟，但它们使用的磁盘I/O比已知键值和该键上的B树索引时找出对应记录所需I/O数（可能只需2~3次磁盘I/O）多得多。本章中提出的方法一般都能提供更好的性能，并且在支持多维数据的专用DBMS系统中得以运用。

14.1.6 传统索引的其他限制

前面提到的结构为范围查询提供的好处并不比最邻近查询大。事实上，我们在例14.6中解决最邻近查询的方法是把它转换成每一维上的一个小范围的范围查询，并且希望范围足以包括至少一个点。因此，如果需要处理更大范围的范围查询，而数据结构是各维上的索引，那么检索每一维上候选记录的指针所需的磁盘I/O数将会比我们在例14.6中看到的大很多。

图14-4中的查询的多维聚集也不能得到很好的支持。如果在商品名称和颜色上建有索引，

我们就能够找出表示粉红色衬衫的所有销售记录并对它们求交，就像我们在例14.6中所做的那样。然而，指定基于商品名称和颜色之外的属性的查询则需要使用这些属性上的索引。

更糟的是，虽然能够按五个属性中的任何一个来排序数据文件，我们却不可能在两个属性上保持数据文件的有序，更不用说五个属性。因此，大多数如图14-4所示形式的查询都需要从所有或几乎所有的数据文件块中检索记录。如果数据是保存在辅存中，则这类查询的执行开销是极其昂贵的。

14.1.7 多维索引结构综述

大多数支持多维数据查询的数据结构归于以下两类之一：

673 1. 类散列表方法。

2. 类树方法。

对于其中的每一种，我们放弃第13章所讨论的一维结构的一些特性。

- 对于基于散列的方法（14.2节中的网格文件和分段散列函数），查询答案恰好就在一个桶中这一优势不再存在。不过，这类方法中的每一种都把我们的搜索限制到桶的子集中。
- 对于基于树的方法，我们至少放弃B树的下列重要特征之一：

1. 树的平衡，其中所有叶结点位于同一层。
2. 树结点和磁盘块的对应。
3. 数据修改执行的速度。

正如我们将在14.3节看到的那样，树经常是一部分比另一部分深；通常深的部分对应于点多的区域。我们也将看到，通常一个树结点所表示的信息远小于一个块所能存放的信息，因此有必要以某种有用的方式来把结点分组存到块中。

14.1.8 习题

习题14.1.1 使用例14.3中的关系

```
Rectangles(id, xll,yll,xur,yur)
```

写出回答下列问题的SQL查询：

* a) 找出与左下角为（10.0，20.0）、右上角为（40.0,30.0）的矩形相交的矩形集。

b) 找出相交的矩形对。

c) 找出完全包含（a）中提到的矩形的所有矩形。

d) 找出完全包含在（a）中提到的矩形中的所有矩形。

! e) 找出关系Rectangles中不是真的“矩形”，即它们实际上不可能存在。

674 对上述每一个查询，说明要是有索引的话，什么样的索引将会有助于检索所需元组。

习题14.1.2 使用例14.4中的关系

```
Sales (day, store, item, color, size)
```

用SQL写出下列查询：

* a) 列出所有销售量超过1000的衬衫颜色及其销售总量。

b) 按商店和颜色列出衬衫的销售。

c) 按商店和颜色列出所有商品的销售。

! d) 按商品和颜色列出销售量最大的商店及其销售量。

对于每一个查询，说明要是有的话，什么样的索引将会有助于检索所需元组。

习题14.1.3 假设范围查询是查询大小为$n \times n$的一个正方形，其中$1 \leq n \leq 1000$，重做例

14.5。需要多少次磁盘I/O？n 为何值时索引才会有帮助？

* **习题14.1.4** 如果记录的文件按 x 排序，重做习题14.1.3。

!! **习题14.1.5** 假设和例14.6一样，点随机分散在正方形内，并且我们想执行一个最邻近查询。选择一个距离d并找出中心在目标点、边长为$2d$ 的正方形中的所有点。如果我们在这个正方形中至少找到一个与目标点距离小于等于 d 的点，则搜索是成功的。

* a) 如果每个面积单元平均有一个点，给出 d 的一个函数来说明我们成功的概率。

b) 如果不成功，我们必须用一个更大的 d 来重复搜索。为简单起见，假设每次不成功时，我们都将 d 加倍且花费的开销也加倍。再次假定每个面积单元有一个点，d 的初始值为多大时我们预计的搜索开销最小？

14.2 多维数据的类散列结构

在本节中，我们将考虑由使用单键建立的散列表推广得到的两种数据结构。在每种情况下，点的桶是所有属性或维的函数。一种方法称为“网格文件”，它通常不是按维来“散列”值，而是通过排序该维的值来划分该维；另一种方法称为“分段散列”，它确实“散列”各维，且每一维都影响着桶号。

675

14.2.1 网格文件

在涉及维数据的查询中，通常比单维索引性能要好的最简单的数据结构之一是网格文件。想一想划分成网格的点空间。在每一维上网格线把空间分成条状，落在网格线上的点被认为是属于以该网格线为下边界的条。不同维的网格线的数目可以不同，并且相邻网格线之间可以有不同的间距，甚至在同一维的线之间也可有不同的间距。

例14.7 让我们为本章引入一个例子：问“谁买金首饰？”设想一个买金首饰的顾客数据库，它告诉我们有关各个顾客的许多信息——姓名、地址等。不过，过了使问题简化，我们假定相关的属性只有顾客的年龄和薪水。我们的示例数据库中有12个顾客。我们可以把它表示成下列的年龄-薪水对：

(25, 60) (45, 60) (50, 75) (50, 100)
(50, 120) (70, 110) (85, 140) (30, 260)
(25, 400) (45, 350) (50, 275) (60, 260)

在图14-6中我们看到，这12个点位于一个二维空间中。我们还在每一维上选择了一些网格线。对这个简单例子，我们在每一维上选择了两根网格线，把空间分成九个矩形，但在每一维

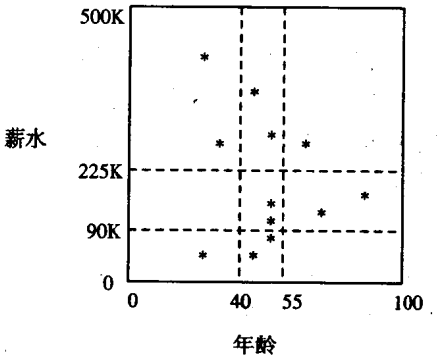


图14-6 网格文件

上选用同样数目的网格线并没有什么理由。我们还使线之间的间距大小不同。例如，在年龄维上，两条竖直线把空间分成宽度分别为40、15和45的三个不等的区域。

在这个例子中，没有点正好落在网格线上。但一般来说，一个矩形包括落在其左边界和下边界上的点，但不包括落在其右边界和上边界上的点。例如，图14-6中央的矩形表示的点的范围是40≤年龄<55和90≤薪水<225。 □

14.2.2 网格文件的查找

空间划分成的每一个区域可以被看成是散列表的一个桶，落入该区域的每个点的记录都存放在属于该桶的块中。如有必要，溢出块可以用来增加桶的大小。

与传统散列表中使用的一维桶数组不同，网格文件使用的数组的维数与数据文件的维数一样。为了正确定位一个点所属的桶，我们需要知道每一维网格线所在的系列值。因此，散列一个点与在它的分量值上运用散列函数多少有些不同。更确切地说，我们查找点的每一个分量并且确定该维上的点在网格中的位置。点
676 在每一维的位置一起决定点所属的桶。

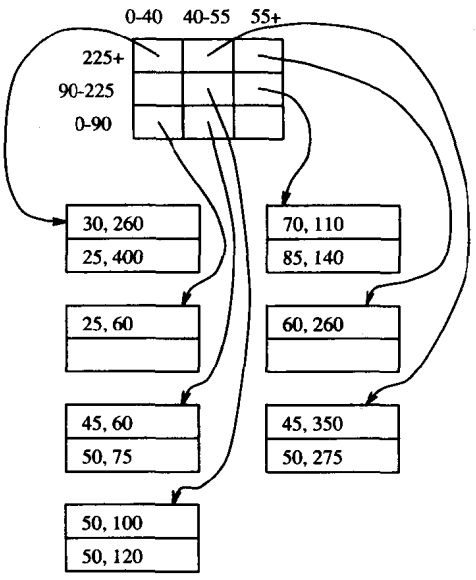


图14-7 表示图14-6中点的网格文件

例14.8 图14-7所示为图14-6中的数据在桶中的存放情况。由于在两维上网格都把空间分成了三个区域，所以桶数是一个3 × 3的矩阵。其中的两个桶

1. 薪水在$90K和$225K之间且年龄在0~40之间，以及

2. 薪水低于$90K且年龄超过55

是空的，且我们没有画出那两个桶的块。其他的桶全被画出，且人为地使每个桶的最大容量小到每个存储块最多容纳两个点。在这个例子中，没有桶超过两个成员，因而不需要溢出块。 □

14.2.3 网格文件的插入

当我们往网格文件中插入一个记录时，遵循查找记录的过程并把新记录放到查找到的桶中。如果在该桶中的块有空间，那就不需要做更多的事。如果在桶中没有空间，那问题就出来了。通常有两种方法解决这个问题：

1. 按需要给桶增加溢出块。只要桶中块的链不会变得太长，这种方法就会运行良好。否则，用来查找、插入或删除的磁盘I/O数最终将大得不可接受。

2. 通过增加或移动网格线来重组结构。这种方法类似于13.4节讨论的动态散列技术，但这
677 里还有别的问题，因为桶的内容在一维上是相互关联的。也就是说，增加一条网格线将分裂沿该线的所有桶。因此，要选择一条对所有桶都最优的网格线也许是不可能的。例如，要是一个桶太大，我们也许不能选择分裂的维或分裂的点且不会造成许多的空桶或使某些桶太满。

例14.9 假若有一个52岁且收入为$200K的人买了金首饰。该顾客属于图14-6中央的矩形。可是，现在该桶有3个记录。我们可以简单地为该桶增加一个溢出块。如果我们想分裂该桶，那么需要选择年龄维或者薪水维，且需要选择一个新网格线来进行划分。这里，引入网格线对

中央桶进行分裂并使两个点在一边而一个点在另外一边的方法只有三种，且是在这种情况下最可行的分裂。

1. 一条垂直线，比如年龄=51，它把两个50岁的顾客记录同52岁的分隔开来。这条线对上、下桶的分裂没有什么影响，因为上、下两个桶中的两个点都在线年龄=51的左边。

2. 一条水平线，它把中央桶中薪水=200的点同其他两个点分隔开来，我们不妨选择像130 678
这样的数字，它同样分裂右边的桶（年龄在55～100且薪水在90～225的桶）。

3. 一条水平线，它把薪水=100的点同其他两个点分隔开来，这次我们建议选择像115这样的数字，它也同样分裂其右边的桶。

选择(1)可能不太明智，因为它没有分裂其他桶，却给我们留下了更多的空桶，且没有减少任何被占用的桶的大小。选择(2)或(3)同样地好，虽然我们可能选(2)，因为它把水平线放在薪水=130，它比(3)更接近于下限90和上限225的中间值。划分桶的结果如图14-8所示。 □

访问网格文件的桶

虽然在一个如图14-7所示的3×3的网格中为点查找相应的坐标比较容易，但我们应该记住网格文件在每一维都可能有大量的条。如果这样，那我们必须为每一维创建索引。索引的查找键是对应维的划分值。

给定某坐标的一个值 v，我们搜索小于等于 v 的最大键值 w。在该索引中与 w 相关联的是v落入矩阵中的行或列。若每一维都给定值，我们就能够找到矩阵中指向桶的指针。然后可以直接检索指针指向的块。

在极端情况下，矩阵十分大；从而使得大多数桶都是空的，而我们又负担不起存储所有的空桶。这时，我们必须把矩阵当作一个关系来处理，它的属性是非空桶的顶点且最后有一个属性表示指向桶的指针。查找这个关系本身就是一个多维搜索，但它的大小比数据文件本身小得多。

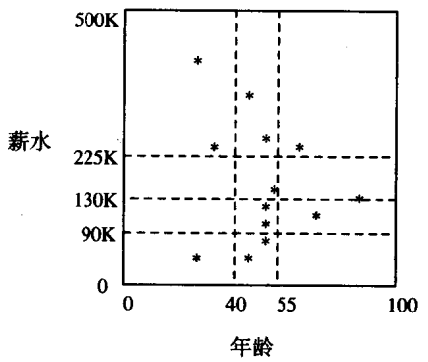


图14-8 点（52, 200）的插入及其后桶的分裂

14.2.4 网格文件的性能

让我们来考虑对各种类型的查询网格文件需要多少磁盘I/O。虽然网格文件可用任意数目的维，但我们一直只考虑二维的情况。对于高维的情况，一个主要的问题是随着维数的增加，桶的数目按指数级增长。如果空间的大部分是空的，那将会有许多空桶。即使在二维时我们也能觉察这个问题。假若在年龄和薪水之间有很大的相关性，那么在图14-6中不论我们把网格线 679
放在哪里，不在对角线上的桶都将会是空的。

不过，如果数据的分布性很好，且数据文件本身又不太大，那么可以选择网格线，使：

1. 桶足够少，这样我们能够将桶矩阵存入主存中，查阅桶矩阵或者当我们插入一条新网格线而给矩阵增加一行或一列时都不会引起磁盘I/O。

2. 我们也能够在主存中保存每一维上网格线的值的索引（请看“访问网格文件的桶”框），或者我们能够一并避免索引而使用主存的二分查找法来查找每一维上定义网格线的值。

3. 一般的桶只有少量的溢出块，因而当我们搜索桶时不会造成太多的磁盘I/O。

在这些假定下，下面是网格文件在一些重要查询类中的表现。

具体点的查找

我们直接找到适当的桶，因而惟一的磁盘I/O就是读入该桶所需的操作。如果我们进行插
680 入或删除，则还需要一个磁盘写操作。若插入需要创建溢出块，则需另一个写操作。

部分匹配查询

这类查询的例子包括“找出所有年龄为50岁的顾客”，或“找出所有薪水为$200K的顾客”。现在，我们需要查找桶矩阵某一行或某列的所有桶。如果在这些行或列上有许多桶，那么磁盘I/O的数量可能很大。

范围查询

范围查询定义网格的一个矩形区域，且在覆盖该区域的桶中找到的全部都是该查询的答案，除了搜索范围边界上的桶中的某些点外。例如，如果我们想找出所有年龄为35～45、薪水为50～100的顾客，那么需要在图14-6左下方的四个桶中查找。在这个例子中，所有的桶都处于边界上，因此我们可能查找到许多不是查询结果的点。不过，如果搜索区域包括大量的桶，那么大多数桶必定是内部的，且这些桶的所有点都是查询的结果。对于范围查询来说，由于我们需要检查许多的桶，因此磁盘I/O数可能会很大。不过，由于范围查询一般得到一个大的结果集，不论如何组织，我们检查的块数一般不会比结果所需的最小块数目多很多。

最邻近查询

给定一个点P，首先查找该点所属的桶。如果我们至少找到了一个点，则对最近邻点就有了一个候选点Q。不过，有可能在相邻桶中存在比Q离P更近的点，这种情形就像图14-5所示的那样。我们必须考虑P到该桶的边界的距离是否小于P到Q的距离。如果存在这样的边界，那么每个这样的边界的相邻桶也必须被搜索。事实上，如果桶是个很窄的矩形（它的一维比另一维长很多）。甚至还可能需要搜索与包含点P的桶不相邻的桶。

例14.10 假若我们在图14-6中查找点 P=（45，200）的最近点。我们发现点（50，120）是那个桶中最近的点，距离为80.2。下面三个桶中都不存在比这更近的点，因为它们的薪水成分最多为90，所以我们可以不去搜索它们。但是，其他5个桶必须被搜索，且我们找到两个等距离的点：（30，260）和（60，260），与P的距离为61.8。一般来说，最邻近查询可以被限制到一小部分桶，因此只需少量磁盘I/O。不过，由于最靠近P的桶可能为空，我们不可能轻易给
681 出查找所需开销的上界。 □

14.2.5 分段散列函数

散列函数能够接受一个属性值的列表作为参数，虽然它们一般只散列一个属性上的值。例如，如果a是一个整型值属性，而b是一个字符串型值属性，那么我们可以把a的值加上b的每一个字符的ASCII码值再除以桶数以后取余数。结果可用作属性对（a，b）上的索引的散列表的桶号。

不过，这样的散列表只可用在a、b值都被指定的查询。一个更好的选择是设计一个散列函数，使它产生若干个二进制位，比如说k个。这k位在几个属性中进行划分，从而我们为第i个属性产生k_i位散列值，其中i=1, 2, …, n，且$\sum_{i=1}^{n} k_i = k$。更精确地说，散列函数h实际上是一个散列函数（h_1, h_2, …, h_n）的列表，其中每个h_i运用到第 i 个属性上且产生 k_i 位二进制位序列。进行散列时，在这几个属性上值为（v_1, v_2,…, v_n）的元组所属的桶通过拼接二进制序列$h_1(v_1)h_2(v_2)\cdots h_n(v_n)$计算得到。

例14.11 如果我们有一个10位桶数目的散列表（1024个桶），可以把4位分给属性 a 而其他6位留给属性b。假若有属性 a 值为 A 且 b 值为 B 的一个元组，可能还有其他一些没有参与散列的属性。利用与属性 a 相应的散列函数h_a散列A得到4位二进制数，比如说为0101。然后我们利用散列函数 h_b 散列 B，可能得到111000这样6位二进制数。那么这个元组的桶号为0101111000，即两个二进制位序列的拼接。

通过采用分段散列函数这种方式，我们可从任何一个或多个参与散列的已知的属性值中得到一些便利。例如，如果我们已知属性 a 的值 A，且得到 $h_a(A)$=0101，那么我们知道只有包含属性 a 的值 A 的元组在64个桶中，它们的桶号形式为0101…，这里…表示任意的6位二进制数。类似地，如果给定一个元组的属性b的值B，我们可以分离可能存在元组的桶为16个桶，这些桶号以6位二进制数 $h_b(B)$结尾。 □

例14.12 假定有例14.7中的“金首饰”数据，我们想把它存放到一个有8个桶的分段散列表中（即3位桶数目）。我们像以前一样假设每块能存放两个记录。我们把一位给年龄属性，而其余两位给薪水属性。

对于年龄上的散列函数，我们取年龄模2；也就是说，年龄为偶数的记录将散列到桶号形式为0xy的桶中(有若干位x和y)；年龄为奇数的记录散列到桶号形式为 1xy 的桶中。对于薪水上的散列函数，用薪水（以千为单位）模4来得到。例如，若薪水为57K，模4后剩下余数1，则 682
该记录所属桶号的形式为z01(有若干位z)。

在图14-9中，我们了解到例14.7中数据在这种散列表中的存放情况。注意，因为我们使用的大多数年龄和薪水都可被10除尽，散列函数不能很好地分布这些点。8个桶中有两个桶存放了4个记录且需要溢出块，而其他3个桶为空。 □

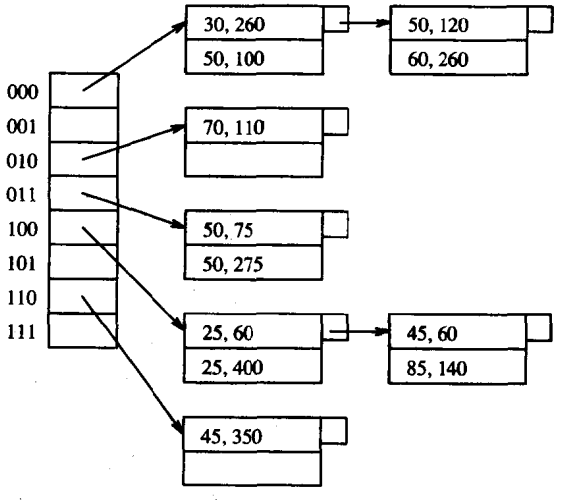


图14-9 分段散列表

14.2.6 网格文件和分段散列的比较

在本节中讨论的两种数据结构的性能很不相同，下面是比较的要点：

- 分段散列表对于最邻近查询或范围查询实际上没有什么用处，问题在于点之间的物理距离并没有通过桶号的接近反映出来。当然，我们可以在某属性a上设计散列函数，从而使最小值分配给第一个位串（全0），下一个值分配给下一个位串，（00…01），等等。如果这样做的话，那么我们又发明了一种网格文件。
- 选择一个好的散列函数将把点随机地散列到各个桶中，这样，这些桶将趋于被均匀地占
683 用。然而网格文件，特别是当维数很大时，易于留下许多空桶或几乎都是空桶。最直观的原因在于，当有许多属性时，至少有一些属性，它们之间有相关性是很可能的，于是空间的很大区域是空的。例如，我们在14.2.4节提到的年龄和薪水的相关性将导致图14-6中大多数点位于对角线附近，使大多数矩形为空。由于这个结果，我们用分段散列表实现会比网格文件实现使用更少的桶和/或更少的溢出块。

因此，如果我们只需要支持部分匹配查询——指定某属性的值而不指定其他属性，那么分段散列函数是可能会比网格文件好。相反，如果我们需要经常做最邻近查询或范围查询，那么我们宁愿选择使用网格文件。

14.2.7 习题

习题14.2.1 在图14-10中是12台PC机的规格说明，假若我们希望只在速度和硬盘大小上设计索引。

* a) 选择5条网格线（两维的总数）以便使任何桶中不超过两个点。

! b) 如果只使用4条网格线，每桶至多有两个点，你能分隔这些点吗？如果可能，则画出如何分隔；如果不可能，则解释为何不可能？

684 ! c) 提出一个分段散列函数，它能划分这些点到4个桶中且每桶不超过4个点。

型号	速度	内存	硬盘
1001	700	64	10
1002	1500	128	60
1003	866	128	20
1004	866	64	10
1005	1000	128	20
1006	1300	256	40
1007	1400	128	80
1008	700	64	30
1009	1200	128	80
1010	750	64	30
1011	1100	128	60
1013	733	256	60

图14-10 一些PC机和它们的特性

处理小桶

我们通常把桶看做包含有价值数据的一个块。可是，我们可能需要创建许多桶，以至于平均每个桶只利用块的一小部分来存放记录。例如，如果我们沿每一维都进行大量划分，高维数据将需要许多桶。因此，在本节的结构和14.3节基于树的方法中，我们可以选择把几个桶（或树的结点）存入一个块中。如果我们这样做，要记住下面的几个要点：

- 块必须在它的块头中保存关于每个记录在哪里和属于哪个桶的信息。

- 如果插入一个记录到桶中，我们可能在包含该桶的块中找不到空间。如果是这样，我们需要按某种方式来分裂块。我们必须决定哪些桶存入哪个块，找出每个桶的记录并且把它们存放到适当的块中，并且调整桶表以指向适当的块。

! **习题14.2.2** 假定我们希望把图14-10中的数据放到一个基于速度、内存和硬盘大小属性的三维网格文件中，给每一维提出一个划分从而使它能把数据划分得很好。

习题14.2.3 选择一个分段散列函数，且速度、内存和硬盘大小三属性各为一位二进制数，使它能很好地划分图14-10中的数据。

习题14.2.4 假定我们用一个只有速度和内存的二维网格文件来存放图14-10中的数据。速度维上的划分为720、950、1150、1350，内存维上的划分为100和200。还假定每桶只能存放两个点。如果插入下列点，试提出好的分裂：

* a) 速度=1000且 内存=192。

b) 速度 = 800且 内存 = 128，然后速度 = 833，内存 = 96。

习题14.2.5 假若我们用网格文件来存放关系 $R(x, y)$。两个属性的范围是0 ~ 1000。该网格文件的划分刚好是等区间：对于x维的划分单位为20，即20、40、60，等等；而对于y维的划分单位为50，即50、100，150，等等。 685

a) 为了回答下面的范围查询，我们需检查多少个桶？

```
SELECT *
FROM R
WHERE 310 < x AND x < 400 AND 520 < y AND y < 730;
```

*! b) 我们希望对点（110，205）执行一个最邻近查询。我们开始搜索左下角为（100，200）、右上角为（120，250）的桶，并且发现在这个桶中最近点为（115，220）。为了证实该点是最近点，还要搜索哪些桶？

! **习题14.2.6** 假定我们有一个网格文件，每一维上有三条网格线（即四个条）。可是，点(x, y)刚好有一个特殊的性质。计算可能有的最大非空桶数，如果：

* a) 点处于一条线上，即存在常量a和b，对于每一个点(x, y)满足$y=ax+b$。

b) 点具有二次相关性，即存在常量a、b和c，对于每一点(x, y)满足$y=ax^2+bx+c$。

习题14.2.7 假定我们用一个有1024个桶（即10位桶地址）的分段散列表来存放关系$R(x, y, z)$。每个关系R的查询恰好指定一个属性，且三个属性被指定的概率相同。如果散列函数基于x维产生5位二进制数，基于y维产生3位二进制和基于z维产生两位二进制数。那么回答一个查询所需要的平均查找的桶的数目是多少？

!! **习题14.2.8** 假定我们有一个桶号为$0 \sim 2^n - 1$的散列表，即桶地址为n位长。我们希望把有两个属性x和y的关系存放到这个表中。任何查询可指定一个x值或y值，但不能同时指定。若x被指定的概率为p：

a) 假定我们划分散列函数使m位给x维而其余$n - m$位给y维。对于回答任意一个查询，预计需要检查多少桶，给出一个m、n和p的函数？

b) 什么样的m（作为n和p的函数）值会使得预计桶数最小？不用担心m可能不是整数。

*! **习题14.2.9** 假若我们有一个随机分布的1 000 000个点的关系$R(x, y)$。x和y的范围是0~1000。我们可以在每个块中存放100个R的元组。我们决定使用每一维上等间距网格线的网格文件，且m为条的宽度。我们希望选择一个m，使得回答一个边长为50的正方形的 686

范围查询所需访问的所有桶的磁盘I/O数最小。你可以假定正方形的边与网格线从不在一直线上。如果我们选择的m太大，在每个桶中将会有许多溢出块，而且桶中的许多点不在查询的范围之内；如果我们选择的m太小，那么将会出现太多的桶，而且块不会装满数据。m的最佳值为多少？

14.3 多维数据的树形结构

现在，我们考虑对于多维数据的范围查询和最邻近查询都有用的另外四种结构。我们将依次考虑：

1. 多键索引。
2. kd树。
3. 四叉树。
4. R树。

前三种用于点集。R树通常用来表示区域的集合，也可用来表示点集。

14.3.1 多键索引

假若我们有几个属性表示数据点的维，并且我们想在这些点上支持范围查询或最邻近查询。一个用来访问这些点的简单的树形模式是索引的索引，或更一般的一棵树，它的每一层的结点都是一个属性的索引。

这种想法如图14-11所示，这是两个属性的情况。“树根”是两个属性中第一个属性的索引，它可以是任何类型的常规索引，如B树或散列表。该索引把每一个索引键值——即第一个属性的值——同指向另一个索引的指针相关联。如果V是第一个属性的一个值，那么通过键值V和它的指针找到的索引是一个指向这些点集的索引，这些点的第一个属性值是 V，而第二个属性为任意值。

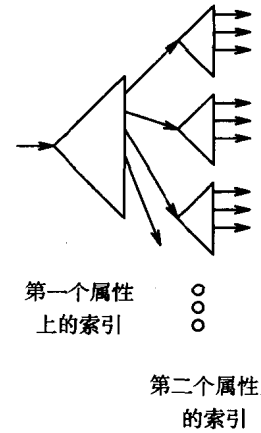


图14-11 在不同键上使用嵌套索引

例14.13 图14-12显示了一个我们一直在用的“金首饰”例子的多键索引，它的第一个属性为年龄，而第二个属性为薪水。关于年龄根索引，如图14-12左边所示。我们没有指出索引如何运作。例如，键-指针对形成那个索引的7行可能分散在B树的叶结点中。不过，最重要的
687 是不论年龄是一个还是多个数据点都只用一个键表示，而且索引使得查找与给定键值相关的指针很容易。

在图14-12的右边是提供访问点本身的7个索引。例如，如果根据根索引找到与年龄为50相关联的指针，我们得到一个以薪水为索引键的更小的索引，且索引中的4个索引键值是与年龄为50的点相关联的4个薪水值。我们再次没有在图中指出索引如何实现。仅指出它所关联的键-指针对。当我们跟踪与这些值（75、100、120和275）各自相关的指针时，我们就找到各自

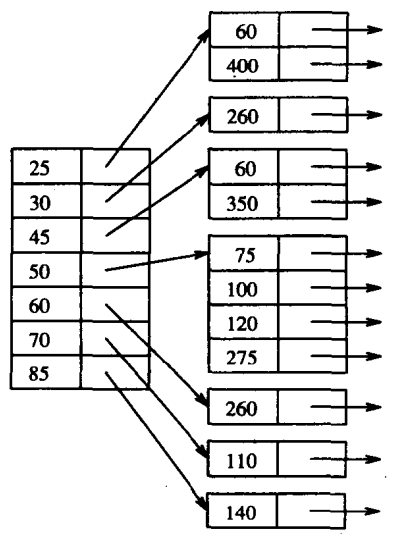


图14-12 对年龄/薪水数据的多级索引

表示的记录。例如，跟踪与100相关联的指针，我们找到了年龄为50、薪水为$100K的人。 □

在多键索引中，有些第二或更高级的索引可能会很小。例如，图14-12有4个只有一个键-指针对的第二级索引。因此，把几个简单表压缩到一个块中来实现这些索引可能是合适的，这种方法在14.2.5节的“处理小桶”框中提到过。

14.3.2 多键索引的性能

让我们考虑多键索引如何在各种不同的多维查询上实现。我们将集中在两个属性的情形，虽然推广到多于两个属性并不使人感到意外。

部分匹配查询

如果第一个属性被指定，那么访问是很有效的。我们使用根索引找到一个子索引，该子索引引导我们到想要的点上。例如，如果根是一个B树索引，那么我们将执行两三次磁盘I/O去找 688
到合适的子索引，然后使用相应的I/O来访问那个子索引和数据文件本身的点。另一方面，如果第一个属性没有给出一个指定值，那我们必须搜索每一个子索引，这是一个潜在的、耗时的处理过程。

范围查询

要是单个索引本身在它们各自的属性上支持范围查询（例如，若它们是B树索引），多键索引对范围查询就会运作得很好。为了回答一个范围查询，我们使用根索引和第一个属性的范围找出可能包含答案点的所有子索引。然后我们使用第二个属性的指定范围搜索每个子索引。

例14.14 假定我们有图14-12的多键索引并且要求作年龄在35～55之间、薪水在100～200之间的范围查询。当我们检查根索引时，发现键45和50是在年龄范围之内。按照相关联的指针找到有关薪水的两个子索引。年龄为45的子索引中没有在100～200之间的薪水，而年龄为50的子索引中有两个薪水：100和120。这样，在范围内仅有两个点，它们是（50，100）和（50，120）。 □ 689

最邻近查询

用多键索引回答最邻近查询使用同本章几乎所有的数据结构实现最邻近查询一样的策略。为了找出点（x_0, y_0）的最邻近的点，我们给出一个距离d，以便期望能找到与点（x_0，y_0）的距离小于等于d的几个点。然后我们再执行$x_0-d\leqslant x\leqslant x_0+d$和$y_0-d\leqslant y\leqslant y_0+d$的范围查询。如果在这个范围内证明是不存在点，或者如果存在点，但该点与（x_0，y_0）的距离大于d（像14.1.5节讨论的那样，在该范围之外可能有更近的点），那么我们必须增加范围，并且再次查找。不过，我们可以依次搜索以便最近的地方被先查找。

14.3.3 kd树

kd树（k维搜索树）是把二叉搜索树推广到多维数据的一种主存数据结构。我们将先介绍这种思想，然后讨论怎样使这种思想适合存储的块模型。kd树是一个二叉树，它的内部结点有一个相关联的属性a和一个值V，它将数据点分成两个部分：a值小于V的部分和a值大于等于V的部分。由于所有维的属性在层间循环，所以树的不同层上的属性是不同的。

在一般的kd树中，数据点被存放在结点内，就像在二叉搜索树中一样。不过我们在开始引入这种思想时做了两个修改，以便获得块模式的有限益处：

1. 内部结点只有一个属性，该属性的一个划分值和指向左、右子女的指针。
2. 叶结点是块，块空间中存放着尽可能多的记录。

例14.15 在图14-13中是一棵我们一直在用的金首饰示例库的12个点的kd树。为简单起见，

我们使用只能存放两个记录的块。这些块和它们中的内容显示在正方形的叶结点中。内部结点是有一个属性——年龄或薪水——和一个值的椭圆。例如，根用薪水属性来分裂：左子树中的所有记录的薪水小于$150K，而右子树的所有记录的薪水至少是$150K。

在第二层，用年龄属性来分裂：根的左子树以年龄为60来分裂，因此在它的左子树中的所有记录将是年龄小于60且薪水少于$150K，在它的右子树中的所有记录将是年龄至少为60且薪水少于$150K。图14-14表明了各个内部结点是如何分裂点空间到叶结点块中的。例如，薪水为150的水平线表示根结点上的分裂。线下方的空间在年龄为60处被垂直分裂，而上方的空间在
690 年龄为47处分裂，它对应于根结点右子女的结果。 □

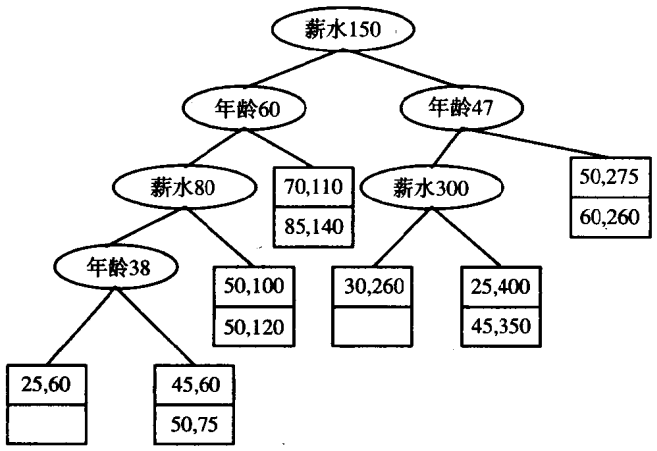


图14-13 kd树

14.3.4 kd树的操作

查找一个所有维都给定值的元组的处理如同在二叉树中一样。我们在每个内部结点上决定沿哪个走向，并且被引向我们所搜索的单个叶结点的块。

为了实现一个插入，我们先做一个查找。最后我们找到一个叶结点，如果叶结点的块中还有空间，就把新的数据点放在那里；要是没有空间，我们把块分裂成两个，并根据分裂叶结点所在层的相应属性划分叶结点中的内容。最后，我们创建一个新的内结点：它的子结点为分裂的两个新块，并且给该内结点一个与分裂相对应的划分值[⊖]。

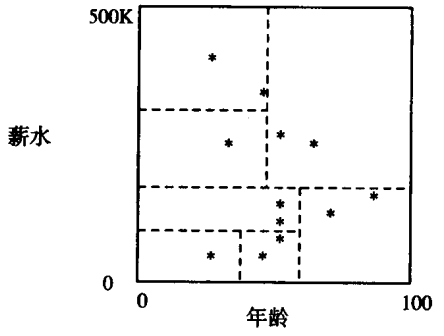


图14-14 图14-13中的树所隐含的划分

⊖ 可能会引起问题的情况是：对于给定维上存在着许多相同值的点且对于该维桶中只有一个值，因而无法被分裂。我们可以试图沿着其他维来分裂，或是使用溢出块。

例14.16 假若某个年龄为35且薪水为$500K的人买了金首饰。从根开始，我们已知薪水至少是$150K，因而往右边走；在该右结点处，拿年龄35跟年龄47比较，它使我们往左边；在第三层上，我们再次比较薪水，且我们的薪水大于划分值$300K。因此我们被引向包含点（25，400）和（45，350）的叶结点，新点（35，500）需插入该块。

在该块中存放不下三个记录，因而我们必须分裂该块。第四层是按年龄来分裂，所以得选择某个年龄来尽可能均匀地划分这些记录。中间值35是一个不错的选择，这样我们就用分裂值为35的内结点取代该叶结点。该内结点的左边是只有一个记录（25，400）的叶结点块，而右边是有另外两个记录的叶结点块，如图14-15所示。 □ 691

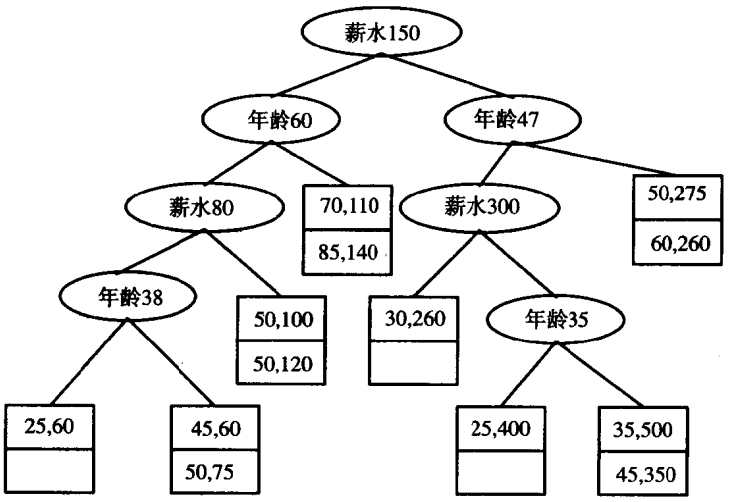


图14-15 插入点（35, 500）后的树

在本章中讨论的更为复杂的查询也可以被kd树支持。下面是算法的关键思想和梗概。

部分匹配查询

如果给定某些属性的值，那么当处于属性值已知的层的结点上时，我们可以往一个方向走；当处于属性值未知的结点上时，必须考察它的两个子结点。例如，如果我们找出图14-13的树中的所有年龄为50的点，必须考察根的两个子结点，因为根是按薪水来分裂的。可是，在根的左子结点上，我们只需要往左走；在根的右子结点上，我们也只需考察它的右子树。又如，假定树是完全平衡的，有大量的层并且有两维，其中有一维的值在搜索中是给定的，那么我们将不得不使用这两种方式来考察其余各层结点，直到达到叶结点数的平方根。

范围查询

有时一个范围允许我们移动到结点的惟一的一个子结点，但如果范围跨越了结点的划分值，那么我们就必须考察两个子结点。例如，给定一个年龄范围35～55和一个薪水范围$100K～$200K，考察图14-13的树如下：在根结点，薪水范围跨越了$150K，因此考察它的两个子结点：在左结点，范围完全属于左边，这样我们前进到薪水为$80K的结点。现在，范围又完全属于右边，这样我们就搜索到记录为（50，100）和（50，120）的叶结点，两个记录都符合查询范围。回到根的右结点，划分值年龄为47告诉我们查看两个子树：在薪水为$300K的结点上，我们只需往左前进，找到点（30，260），它实际上超出了范围；在年龄为47的右结点，我们找到两个点，它们都超出了范围。 692

无长久之事

本章讨论的每种数据结构应允许插入和删除。它们根据局部的判断来重组结构。多次数据库更新后，这些局部判断的结果使得结构在某些方面不平衡。例如，网格文件可能有太多的空桶，或kd树可能很不平衡。

经过一段时间后，任何数据库都需要重构是十分正常的。至少在这个时候，我们有机会创建索引结构且尽可能使得这个索引平衡和有效。这种重构的开销能够由大量导致不平衡的更新来摊还，使每个这样的更新的开销变小。可是，我们确实需要能够“关闭数据库”，即：使得数据库在被重装时不可用。这种情况可能是个问题也可能不是，这依赖于应用。例如，在晚间，当没有人访问它们时，许多数据库被关闭。

最邻近查询

使用在14.3.2节讨论的方法。把这个问题当作一个适当的范围查询来处理。如有必要，用一个更大的范围重做。

14.3.5 使kd树适合辅存

假定我们用一个有n个叶结点的kd树存放文件，那么从根到叶结点的路径的平均长度将大约是$\log_2 n$，如同任何二叉树一样。如果每块存放一个结点，那么当我们遍历一条路径时，必须为每个结点作一次磁盘I/O。例如，如n=1000，那么需要大约10次磁盘I/O，远远超过通常B树的2～3次磁盘I/O——即使是一个十分大的文件。另外，由于kd树内部结点的信息相对较少，块的大部分空间将被浪费掉。

我们无法彻底解决长路径和未用空间这一对孪生问题。不过，下面的两种方法将会在性能上做些改进。

内结点的多分枝

693 要是有许多键-指针对，kd树的内结点看起来更像B树结点。如果在结点上有n个键，我们能够把属性a的值分裂成n+1个范围。如果有n+1个指针，我们能够沿着适当的指针到只包含属性a值在那个范围内的点的子树。当我们为了保持分布和平衡——就像我们为B树所做的那样——设法重组结点时，问题就来了。例如，假若我们有一个年龄上分裂的结点且需要合并它的两个子结点，它们都是按薪水分裂。我们不能简单地把两个子结点的薪水范围弄到一个结点中，因为这些范围一般会有部分重叠。注意，如果（像在B树中那样）两个子结点进一步优化年龄范围，这会变得多么的容易。

聚集内结点到块

我们可以保持树的结点只有两个子结点的思想。我们能把多个内结点压缩到一个块中。为了减少遍历路径访问的块数量，我们最好远离一个结点一个块的方式而将若干层的所有子结点存入一个块。在这种方式下，一旦检索到该结点的块，我们必定会使用该块中的其他某些结点，这样节省了磁盘I/O。例如，假若把三个结点压缩到一块。那么在图14-13的树中，我们把根和它的两个子结点存入到一个块中。然后我们压缩薪水为80的结点和它的左子结点到一个块。剩下薪水为300的结点，把它放在一个单独的块中。或许它与后面两个结点共享一个块，虽然当树增长或收缩时，共享需要我们做相当的工作。因此，如果想查找记录（25，60），我们只需
694 要遍历两个块，尽管我们搜索了四个内结点。

14.3.6 四叉树

在一个四叉树中，每个内结点对应于二维空间中的一个正方形区域，或是k维空间的k维立

方体。像本章中的其他数据结构一样，我们将主要考虑二维的情形。如果一个正方形中的点数不比一个块中能存放的数多，那么我们就把这个正方形看做树的叶结点，并且由存放它的点的块表示；如果正方形中还有太多的点以至于一个块存放不下，那么我们就把这个正方形看做内结点，它的子结点对应于它的四个象限。

例14.17 图14-16显示了被组织成对应于四叉树结点区域的金首饰的数据点。为了计算的便利，我们限制了可用的空间：薪水范围在0~$400K之间，而不是像在本章中其他例子那样达到$500K。我们继续假定每个块只存放两个记录。

图14-17清楚地显示了这种树。对于象限和结点的子结点，我们使用指南针的标示法（例如，SW代表西南象限——在中心左下方的点）。子结点的顺序总是如同在根结点所标识的那样。每个内结点指出该区域的中心坐标。

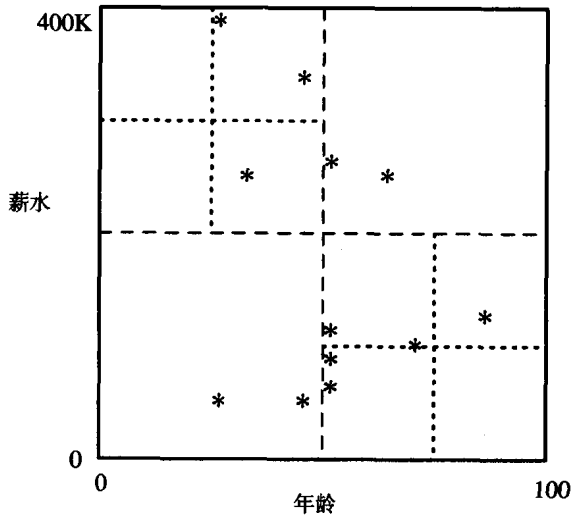


图14-16 组织成四叉树的数据

由于整个空间有12个点且一个块中只存放两个点，我们必须把空间分裂成象限，我们在图14-16中用短划线标示。两个结果象限——西南和东北——只有两个点。它们可以用叶结点来表示且不需要进一步分裂。

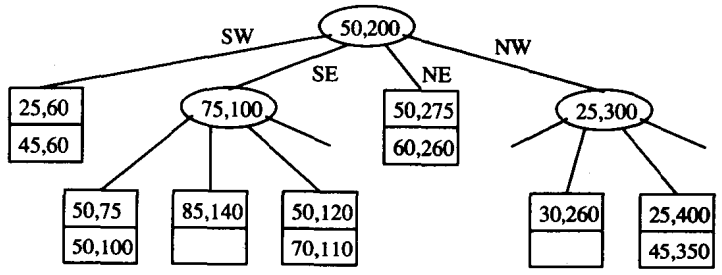


图14-17 四叉树

剩下的两个象限都有多于两个点。两个都被分裂成子象限，如图14-16中点线所示。每个结果象限都有两个或更少的点，因而不需更多的分裂。 □ 695

由于一个k维的四叉树的内结点有2^k个子结点，因而存在一个k的范围，使得结点能方便地

装入块中。例如，若128即2^7的指针能在一个块中存放下，则k=7是一个合适的维数。不过，对于二维的情形，情况并不比kd树好多少，一个内结点有4个子结点。此外，我们能够为kd树选择一个分裂点，可我们被约束在四叉树区域的中心，它可能能也可能不能把该区域的点均匀地划分。尤其当维数很大时，我们可能在内结点中会找到许多空指针（对应于空象限）。当然我们在高维结点如何表示方面可以聪明些，并且只保留非空指针和该指针表示的象限的标示，这样可节省相当的空间。

关于我们在14.3.4节讨论的kd树的标准操作，我们这里不再进行详细的讨论：四叉树的算法类似于kd树的算法。

14.3.7 R树

R树（区域树）是一种利用B树的某些本质特征来处理多维数据的数据结构。回想起B树的结点有一个键的集合，这些键把线分成片段，沿着那条线的点仅属于一个片段，如图14-18所示。B树因此使我们很容易地找到点。如果把沿线各处的点表示成B树结点，我们就能够确定点所属
696 的惟一子结点，在那里可以找到该点。

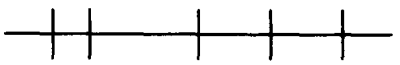


图14-18 B树结点沿线把键分成不相连片段

另一方面，R树表示由二维或更高维区域组成的数据，我们把它称为*数据区*。一个R树的内结点对应于某个*内部区域*，或称“区域”，它不是普通的数据区。原则上，区域可以是任何形状，虽然实际中它经常为矩形或其他简单形状。R树的结点用子区域替代键，子区域表示结点的子结点的内容。图14-19显示了一个与大矩形相关联的R树的结点。虚线表示的矩形表示与它的四个子结点相关联的子区域。注意，子区域没有覆盖整个区域，只要把位于大区域内的所有数据区都完全包含在某个小区域中就合乎要求。进一步说，子区域允许有部分重叠，尽管使部分重叠更小是所希望的。

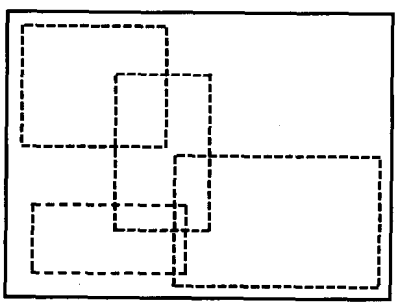


图14-19 R树结点区域和子结点子区域

14.3.8 R树的操作

对于“where-am-I”这类典型查询，R树是有用的。这类查询指定一个点P且询问该点所处的数据区域。

我们从根结点开始，它关联着整个区域。我们检查根上的子区域且确定根结点的哪个子结点对应的内部区域包含点P。注意到可能有0、1或几个这样的区域。

如果为0个区域，那我们就结束，P不存在任何数据区中。如果至少有一个内部区域包含点P，那么我们必须在与每个这样的区域对应的子结点上递归查找P。当查找到一个或多个叶结点时，我们将找到真正的数据区，要么为每个数据区找到一个完整的记录，要么找到一个指向
697 该记录的指针。

当插入一个新区域R到R树时，我们从根开始且设法找到一个适合R的子区域。如果有多于一个这样的区域，那么就选择一个，进到它相应的子结点，且在那里重复这个过程。如果不存在包含R的子区域，那么我们得扩大其中一个子区域。究竟选择哪一个是一个艰难的决定。直观上讲，我们希望扩大的区域尽可能小，因此我们可以看一下哪一个子结点的子区域将会使它们的面积增加得尽可能少，改变该区域的边界来包含R，且在它的相应子结点中递归地插入R。

最后，我们到达叶结点，在那里插入区域R。不过，如果在该叶结点上没有空间，那么我

们必须分裂叶结点。怎样分裂叶结点受某些选择的支配。通常我们希望两个子区域能尽可能小。不过在它们之间，它们必须覆盖原始叶结点的所有数据区。完成分裂后，我们用对应于两个新叶结点的区域和指针对替换原始叶结点的区域和指针。如果父结点中有空间，那么就结束。否则，就像在B树中一样，我们往上递归地分裂结点。

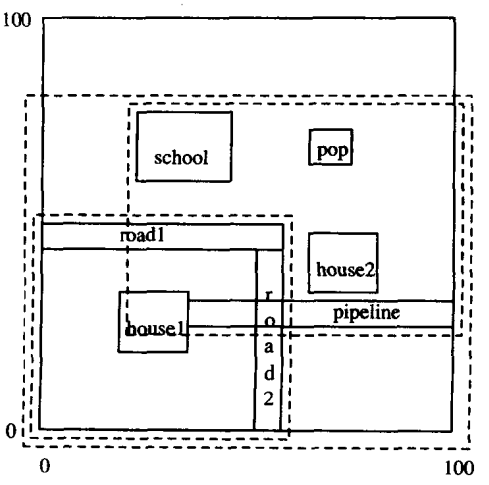


图14-20 分裂成对象集

例14.18 让我们考虑追加一个新区域到图14-1所示的地图中。假设叶结点可存放6个区域。再假设图14-1中的6个区域一起在一个叶结点上，该区域由图14-20中的一个外部（实）矩形表示。

现在，假设当地的电话公司在图14-20所示的位置增加一个POP（存在点，或天线）。由于7个数据区域不能存放在一个叶结点上，我们要分裂该叶结点，4个在一个叶结点上而其他3个在另一个叶结点上。我们有很多的选择。我们选择了图14-20中的划分（用内部的、虚线矩形表示）：它使重叠部分最小化，同时又尽可能均匀地分裂叶结点。

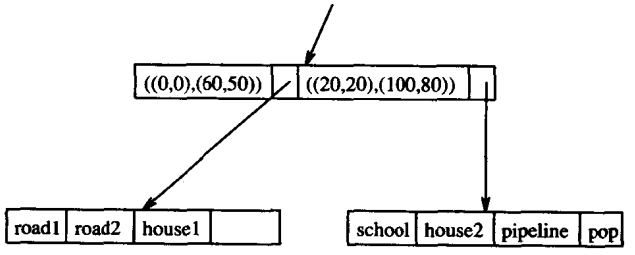


图14-21 R树

我们在图14-21中显示了两个新叶结点如何插入到R树中。两个结点的父结点有指针指向它们，且与指针相关联的是每个叶结点所覆盖的矩形区域的左下角和右上角坐标。 □

例14.19 假设我们在house2的下方插入另一个house，其左下角坐标为（70，5），右上角坐标为（80，15）。由于这个house没有被任何一个叶结点的区域完全包含，我们必须选择一个区域来扩大。如果扩大对应于图14-21中下部的子区域，那么我们给该区域增加了1000平方单位，因为往右延伸了20个单位；如果把另一个子区域的底边下移15个单位来扩大该子区域，那么我们增加了1200平方单位。我们最好选择第一个，且新区域改变成如图14-22所示。我们还必须把图14-21中顶层结点的区域描述从

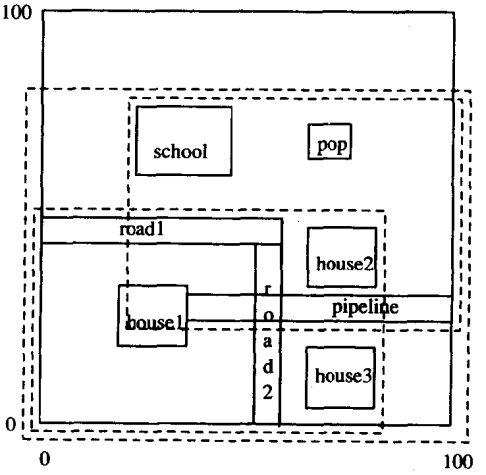


图14-22 扩大一个区域来容纳新数据

698

((0，0)，(60，50)) 改变成 ((0，0)，(80，50))。 □

14.3.9 习题

699 **习题14.3.1** 给图14-10的数据画出多级索引，如果索引建立在：

a) 速度，然后内存。

b) 内存，然后硬盘。

c) 速度，然后内存，最后硬盘。

习题14.3.2 把图14-10的数据放到一个kd树上。假定每块能存放两个记录，给每一层挑选一个使数据划分尽可能均匀的划分值。分裂属性的顺序选择：

a) 速度，然后内存，再交替。

b)速度，然后内存，最后硬盘，再交替。

c)不论什么属性，只要它在每个结点产生最均匀的分裂。

习题14.3.3 假设我们有一个关系R（x，y，z），其属性x和y一起形成键。属性x的范围是1～100，属性y的范围是1~1000。对于每一个x，有100个y值不同的记录；对于每个y，有10个x值不同的记录。注意R有10 000个这样的记录。我们希望使用一个多键索引来帮助回答如下形式的查询：

```
SELECT z
FROM R
WHERE x = C AND y = D;
```

其中C和D为常量。假定每块能够存放10个键-指针对，并且我们希望在每一级创建稠密索引。可能在它们之上有高层的稀疏索引，因此每个索引从单个块开始。还假定最初所有的索引和数据块都在磁盘上：

* a)如果第一个索引是建在x上，回答上述形式的查询需要多少次磁盘I/O?

b) 如果第一个索引是建立在y上，回答上述形式的查询需要多少次磁盘I/O?

! c) 假设你一直被允许有11个内存缓冲块，如果你想使额外需要的磁盘I/O数量最小化，将选择哪些块？你会生成x或y的第一个索引吗?

习题14.3.4 对于习题14.3.3(a)的结构，需要多少次磁盘I/O来回答$20 \leqslant x \leqslant 35$且$200 \leqslant y \leqslant 350$这个范围查询。假定数据分布均匀，即对于任何给定的范围，预计数量的点将会被找到。

700 **习题14.3.5** 在图14-13的树中，什么样的新点将会被插入到：

* a) 点（30，260）所在的块。

b) 点（50，100）和点（50，120）所在的块。

习题14.3.6 如果我们相继插入点（20，110）和（40，400），画出图14-15的树可能的变化。

! **习题14.3.7** 我们提到，如果kd树是完全平衡的，并且执行一个两属性中给定其中任意一个值的部分匹配查询，我们就结束查找需访问n个叶结点中的$\sqrt{n}$个。

a) 解释为什么。

b) 如果树在d维中交替地分裂，且指定这些维中的m个值，则我们预计要查找的叶结点所占比例是多少?

c) 把(b)的结果和分段散列表进行对比?

习题14.3.8 把图14-10的数据存入一个有速度和内存维的四叉树中。假定速度的范围是100～500，而内存的范围是0～256。

习题14.3.9 加上第三维硬盘，它的范围是0～32，重做习题14.3.8。

*! **习题14.3.10** 如果允许放入一个中心点到四叉树的任意象限中，我们总是能划分象限为等数目点的子象限（或者尽可能相等，如果象限的点数不能被4整除）吗？证明你的答案。

! **习题14.3.11** 假设我们有一个1 000 000个区域的数据库，这些区域可能部分重叠。R树的结点（块）能够容纳100个区域和指针。任何结点表示的区域有100个子区域，且这些区域部分重叠情况是这样的：100个子区域的总面积是该区域面积的150%。如果我们对一个给定点执行“where-am-I”查询，预计要检索多少个块？

! **习题14.3.12** 在图14-22表示的R树中，一个新区域可能进入包含学校的子区域或者包含房子3的子区域，描述我们更愿意把它放入包含学校的子区域中的新矩形区域（即选择子区域大小增加最少的子区域）。

701

14.4 位图索引

现在让我们转到一种十分不同于目前所见索引的索引上来。先设想文件的记录有一个永久的号：1, 2, …, n。此外，文件存在某种数据结构，对于任意的一个i它可使我们容易地找到第i个记录。

字段F的一个*位图索引*是一个长度为n的位向量的集合。每一个位向量对应于字段F中可能出现的值。如果第i个记录的字段F的值为v，那么对应于值v的位向量在位置i上的取值为1；否则该向量的位置i上的取值为0。

例14.20 假设文件由包括两个字段F和G的记录组成，字段分别为整数型和字符串型。当前文件有六个记录，编号为1~6，它们的值依次如下：(30, `foo`),(30, `bar`),(40, `baz`),(50, `foo`),(40, `bar`),(30, `baz`)。

第一个字段F的位图索引有三个位向量，每个长度为6。第一个对应于30，是110001，因为第一、第二和第六个记录都是F=30；另外两个分别对应于40和50，是001010和000100。

字段G的位图索引也有三个位向量，因为那里有三个不同的字符串出现。三个向量是：

值	向量
`foo`	100100
`bar`	010010
`baz`	001001

在每种情形下，1表示在那个记录中出现了相应的字符串。 □

14.4.1 位图索引的诱因

最初可能会出现位图索引需要太多太多的空间，尤其是当字段有许多不同值时，因为位的总数是记录和字段值数的乘积。例如，如果字段是键，且记录数为n，那么该字段的所有位向量需使用n^2个二进制位。不过，压缩能够做到使位的数量接近于n，而与不同字段值的数量无关，就像我们将在14.4.2节看到的那样。

你也可能怀疑位图索引的管理会出现问题。例如，它们依赖于自始至终保持同样的记录数。随着文件增加和删除记录，我们如何找到第i个记录？我们怎样有效地为一个值找到位图？这些和其他相关问题在14.4.4节讨论。

位图索引的补偿优势是它们允许我们在许多情形下高效地回答部分匹配查询。在某种意义上说，它们提供了我们在例13.16中讨论的桶的优势。在那里，我们通过指定几个属性值找到 702
`Movie`元组，而无须先检索在每一个属性中匹配的所有记录。下面的一个例子将说明这一点：

例14.21 回想例13.16，在那里，我们查询Movie关系

```
SELECT title
FROM Movie
WHERE studioName = 'Disney' AND
      year = 1995;
```

假设在属性studioName和year上建有位图索引，我们就能够相交对应于year=1995 和studioName='Disney'的两个位向量，即逐位求这些位向量的与，且得到一个向量，该向量的位置*i*取值为1当且仅当第*i*个Movie元组对应的电影是由Disney于1995年制作的。

如果能够按给定的元组号检索Movie元组，我们就只需要读包含一个或多个这样的元组的那些块，正如我们在例13.16所做的那样。为了相交位向量，我们必须把它们读进内存，这需要给这两个位向量之一所占据的每个块一个磁盘I/O。如上所述，我们稍后将讲述两个问题：14.4.4节的按给定记录号访问记录和14.4.2节的证实位向量并不占太多的空间。 □

位图索引也能够帮助回答范围查询。下面我们将考虑一个例子：它不仅阐述了它们在范围查询中的使用，而且用一些短的位向量详细地显示了位向量的逐位与逐位或如何能够被用来找到查询的答案，它只查找我们所要的记录而不会查找任何其他记录。

例14.22 先考虑例14.7的金首饰数据。假定例子中的12个点是如下编号从1到12的记录：

1:(25, 60) 2:(45, 60) 3:(50, 75) 4:(50, 100)
5:(50, 120) 6:(70, 110) 7:(85, 140) 8:(30, 260)
9:(25, 400) 10:(45, 350) 11:(50, 275) 12:(60, 260)

对第一个分量年龄，它有7个不同值，因此年龄的位图索引由下面7个向量组成：

25: 100000001000 30: 000000010000 45: 010000000100
50: 001110000010 60: 000000000001 70: 000001000000
85: 000000100000

703 对于薪水分量，它有10个不同值，因而薪水的位图索引有下面的10个向量：

60: 110000000000 75: 001000000000 100: 000100000000
110: 000001000000 120: 000010000000 140: 000000100000
260: 000000010001 275: 000000000010 350: 000000000100
400: 000000001000

假定我们要找出年龄范围在45~55且薪水范围在100 ~ 200的所有首饰购买者。先找出在这个范围内的年龄值向量：在这个例子中它们是010000000100和001110000010，分别对应于年龄45和50。如果取它们的逐位或，我们就得到一个新向量：当且仅当第*i*个记录的年龄在给定范围内时它的位置*i*上取值为1，且该向量为011110000110。

下一步，我们找出范围在100K~200K之间的薪水值向量：它们有4个，分别对应于薪水100、110、120和140，它们的逐位或结果是000111100000。

最后一步是取通过“或”运算得到的两个位向量的逐位与，即：

011110000110 AND 000111100000=000110000000

于是我们找到只有第四和第五个记录（50，100）和（50，120）是在所需范围内。 □

14.4.2 压缩位图

假定我们在一个有n个记录的文件的字段F上建有位图索引，且在文件中出现的字段F的不同值是m。那么该索引的所有位向量的二进制位数就是mn。如果块的大小为4096个字节，那么在一个块中我们可存放32 768位，因此所需块数是$mn/32\ 768$。这个数比存放文件本身所需的块数要小，但是随着m的变大，位图索引所需空间也就越多。

但是，如果m很大，那么位向量中的1将会很少。精确地说，任何一位出现1的概率为$1/m$。

二进制数不能用作分段长度编码

假设我们用整数i的二进制数来表示一个 i 个0后跟一个1的段。位向量000101分别由长度为3和1的两个段组成。这两个整数的二进制表示为11和1，这样000101的分段长度编码是111。可是，类似的计算表明位向量010001也被编码成111；而位向量010101是编码为111的第三个向量。因此，111不能够被惟一地解码成一个位向量。

如果1很少，那么我们就有机会编码位向量以便它们平均所占用的位比n少很多。一个常用的方法叫做*分段长度编码*，通过对整数i进行适当的二进制编码，得到一个由i个0且后跟一个1所组成的序列，这个序列表示一个段。我们把每个段的代码拼接在一起，则这个位序列就是整个位向量的编码。

我们可能会想到可以就把i表达成二进制数来表示整数i。不过，如此简单的方法是不行的，因为把编码序列分成多个部分后要惟一地确定各个段的长度是不可能的（请看“二进制数不能用作分段长度编码”框）。因此，表示段长度的整数i的编码一定比简单的二进制表示更复杂。 704

我们应当使用多种可选方法中的一种来编码。它们中有一些更好、更复杂的方法能够按2的倍数来提高压缩的量。但只有当一般的段很长时才行。在我们的方法中，首先需要确定i的二进制表示是多少位，数字 j 近似于$\log_2 i$，被表示成“一元”：$j-1$个1和单个0。然后，我们在它后面加上 i 的二进制数[⊖]。

例14.23 如果 $i = 13$，那么 $j = 4$。即我们需要4位二进制来表示i。因此，i 的编码开始部分为1110。我们把 i 的二进制数1101加上，这样，13的编码就是11101101。

$i = 1$的编码是01，而$i = 0$的编码是00。在每一种情况下，$j = 1$，因此我们以一个0开始且0后面为表示i的一位二进制数。 □

如果我们拼接了一个整数的编码序列，总能够恢复段的长度序列，且据此可以恢复原始位向量。假设已经扫描了一些编码位，且我们正处在某个整数 i 的编码位序列的开始位置上。我们向前扫描到第一个0并确定 j 的值。即，j 等于我们找到第一个0所扫描过的位数（在计算位时包括0本身）。一旦我们知道 j，就查找后 j 位，该 j 位用二进制表示的数就是 i。此外，一旦扫描过表示i的二进制位，我们便知道下一个整数编码的开始位置。所以我们可以重复这个过程。

例14.24 让我们来对序列11101101001011进行解码。从第一位开始，我们在第四位上找到第一个0，因而 $j = 4$，下面四位为1101，因而我们确定第一个整数是13。现在我们剩下001011要解码。

由于第一位是0，我们知道下一位表示整数本身，该整数为0。因此已经解码的序列为13和

⊖ 实际上，除了$j=1$（即 $i=0$或 $i=1$）的情况，我们能够肯定 i 的二进制表示都以1开头。这样，如果省去这个1，使用剩下的 $j-1$位，我们能够为每个数字节省一位。

705 0，我们必须解码剩下的序列1011。

我们在第二个位置上找到第一个0，于是下结论：最末两位表示最后的整数3。我们的整个分段长度序列是这样：13, 0, 3。从这些数字中，我们能够重新构造实际的位向量：0000000000000110001。 □

从技术上讲，每个这样解码的位向量都以1结尾，且任何的尾数0串都不会被恢复。由于我们可能知道文件中记录的数目，因此附加的0串能够被加上。不过，既然在位向量中0表示相应的记录不在所描述的集中，我们甚至不必知道记录的总数，并且可以忽略这个尾数0串。

例14.25 让我们把例14.23中的某些位向量转换成分段长度码。前三个年龄25、30和45的位向量分别为100000001000、000000010000和010000000100。第一个位向量的分段长度序列为（0，7）。0的编码是00，7的编码是110111。这样，年龄25的位向量变成了00110111。

类似地，年龄30的位向量只有一个7个0的段。因此，它的代码是110111。年龄45的位向量有两个段（1，7），由于1的编码是01，且我们确定的7的编码是110111，则第三个位向量的编码是01110111。 □

例14.25中的压缩并不大。不过，当记录数n很小时，我们看不到真正的好处。为了认识这种编码的价值，假定$m=n$，即建有位图索引的属性的值是惟一的。注意到长度为i的段大约有$2\log_2 i$位。如果每个位向量只有一个1，那么它是一个单一段，且那个段的长度不可能比n长。这样，在这种情况下，位向量编码的上限长度为$2\log_2 n$位。

由于在索引中有n个位向量（因为$m=n$），则表示这个索引的位总数最多为$2n\log_2 n$。注意，没有编码时需要n^2位。只要$n>4$，我们就有$2n\log_2 n < n^2$，且随着n的增大，$2n\log_2 n$变得越来越小于n^2。

14.4.3 游程长度编码位向量的操作

当我们需要在编码位向量上执行逐位与或逐位或操作时，除了解码它们并在原始的位向量上操作外，没有别的选择。不过，我们不必同时全部解码。我们描述的压缩方法可以一次解码一个游程，且能因此确定操作数位向量的下一个1在什么位置。如果进行或操作，就在输出的相应位置生成1；而如果进行与操作，当且仅当两个操作对象在相同的位置有下一个1时，我们才能生成1。涉及的算法很复杂，但下面的例子可能使这个思想清晰明了。

706 **例14.26** 考虑我们在例14.25中得到的年龄为25和30的编码位向量：00110111和110111。我们能够容易地解码它们的第一个游程：我们找出它们分别为0和7。也就是25的位向量的第1个出现在位置1，而30的位向量的第一个1出现在位置8。因此我们在位置1生成1。

下一步，我们必须解码年龄25的位向量的下一个游程，因为该向量可能在年龄30的位向量在位置8上生成1之前生成另外一个1。年龄25的位向量的下一个段是7，说明该位向量生成下一个1的位置为9。因此我们可生成一个6个0的串和一个位置8上的1，它是来自于年龄30的位向量。位置9上生成一个1，它是来自于年龄25的位向量。该向量也不会生成后续的1串。

我们得出结论，这两个位向量的或是100000011。对照12位的原始位向量：我们看到这几乎是正确的，除了尾部的三个0被省去外。如果我们知道文件中的记录数是12，就能够追加这些0。不过，我们追不追加都不重要，因为只有1才会导致记录被检索。在这个例子中，我们不论怎样也不会去检索10～12之间的任何一个记录。 □

14.4.4 位图索引的管理

我们已经描述了位图索引上的操作，但没有谈到三个重要问题：

1. 当我们想查找一个给定值的位向量，或者给定范围内的值对应的多个位向量时，我们如何有效地找到它们？

2. 当我们已经选择好回答查询的记录集时，如何有效地检索这些记录？

3. 当数据文件由于记录的插入或删除而发生改变时，我们如何调整给定字段上的位图索引？

查找位向量

第一个问题可以基于我们已经学过的技术来回答。把位向量看成记录，它们的键是对应于该位向量的字段值（虽然值本身不在“记录”中出现）。任何辅助索引技术都可以使我们有效地按值找到它们的位向量。例如，我们能够使用B树，它的叶结点包含键-指针对，指针指向该键值的位向量。B树通常是较好的选择，因为它很容易地支持范围查询。不过散列表和索引顺序文件是另外的可选对象。

我们也需要在某处存储位向量。最好是把它们看做可变长记录，因为随着数据文件中记录的增加，它们一般会增长。如果位向量或许处于压缩的形式，它一般会比块更短，这时可考虑把几个位向量存入一个块且按需要移动它们。如果位向量比块更长，应该考虑使用块的链表来 707
存放每个这样的位向量。在这时，12.4节的技术很有用。

查找记录

现在，让我们来考虑第二个问题：一旦确定了我们所需要的数据文件中的记录k，如何找到它。同样，可以采用我们已经学过的技术。把第k个记录看做索引键值为k（虽然该键实际上并不在记录中出现）。然后我们可在数据文件上创建辅助索引，它的索引键是记录号。

如果找不到用其他方式来组织文件的理由，我们甚至可以用记录号作为主索引的索引键，就像在13.1节讨论的那样。那么，数据文件的组织就特别简单，因为记录号从不改变（即使记录被删除也一样），并且我们只需追加新记录到数据文件的后面。因此，我们可以把数据文件的块完全装满，而不用为插入到文件中间的记录留出额外的空间——像我们在13.1.6节的索引序列文件的一般情况所必需的那样。

数据文件修改的处理

在位图索引中，有两个方面来反映数据文件修改的问题：

1. 一旦分配后，记录数必须保持一定。

2. 数据文件的改变需要位图索引也作相应改变。

第1项的结果是当我们删除记录i时，“隐去”其记录号是最容易的。它在数据文件中的空间用“删除标记”代替。位图索引也必须改变，因为在位置i上为1的位向量必须把这个1改成0。注意，因为在删除之前，我们知道记录i的值，所以我们能够找到适当的位向量。

接下来考虑新记录的插入。我们保留了下一个可用记录号，并且把它分派给新记录。然后，对于每个位图索引，我们必须确定新记录在相应字段的值，并在该值的位向量后面追加1。从技术上讲，在这个索引的其他位向量的末端都加上一个新0，但是，如果我们使用了像14.4.2节的压缩技术，则对于压缩值就不需要作任何改变。

作为一种特殊的情况，新记录有一个索引字段以前没有出现过的值。在这种情况下，我们需要给这个值一个新的位向量，且这个位向量和它的相应值需要被插入到辅助索引结构中，该结构被用来按给定值查找它的相应的位向量。 708

最后，让我们来考虑对数据文件中记录i的修改，即把一个有位图索引的字段的值从值v改为值w。我们必须找到v的位向量并把位置i上的1改为0。如果存在一个值w的位向量，那么我们

把它的位置i上的0改为1；如果仍不存在值w的位向量，那么我们就像前一段讨论这种情况的那样创建一个位向量。

14.4.5 习题

习题14.4.1 针对下列属性，分别用非压缩方式和14.4.2节介绍的压缩方式给图14-10中的数据画出位图索引：

* a) 速度

b) 内存

c) 硬盘

习题14.4.2 利用例14.22中的位图，找出年龄范围为20～40且薪水为0～100范围的首饰买主。

习题14.4.3 考虑一个有1 000 000个记录的文件，且字段F有m个不同值：

a) 作为m的一个函数，F的位图索引有多少个字节？

! b) 假定编号从1到1 000 000的记录的字段F的值按循环方式给出。因此，每个值每隔m个记录出现一次。使用压缩索引需要多少个字节？

!! **习题14.4.4** 我们在14.4.2节中建议：把编码数i占用的位数从我们在那一节里使用的$2\log_2 i$减少到接近$\log_2 i$是可能的。当i很大时，指出如何尽可能地接近该极限。*提示*：使用一元编码——用来对i的二进制编码的长度进行编码的方法。你能用二进制对码的长度编码吗？

习题14.4.5 使用14.4.2节的方法，编码下列位图：

* a) 0110000000100000100

b) 1000001000000010011101

709 c) 0001000000000010000010000

*! **习题14.4.6** 我们指出：对于n个记录的文件，压缩位图索引要耗费大约$2n\log_2 n$位。这个位数与B树索引耗费的位数相比如何？记住，B树索引的大小依赖于键和指针的大小，以及（小部分）块的大小。不过，在你的计算中，可对这些参数做些合理估计。为什么我们可能更愿意选择B树，即使它比压缩位图占用更多的空间？

14.5 小结

- *多维数据*：许多应用，诸如地理数据库或销售和仓库数据，可以被认为是在二维或多维空间中的点。
- *需要多维索引的查询*：在多维数据上需要被支持的查询种类包括：部分匹配（在维的子集上指定值的点集）、范围查询（在每一维的范围内的点集）、最邻近查询（离给定点最近的点）和where-am-I（包含一个给定点的区域或区域集）。
- *最邻近查询的执行*：许多数据结构允许通过执行一个围绕给定点的范围查询来执行最邻近查询。要是在该范围内不存在点，则扩大这个范围。因为在矩形范围内找到了点并不排除在矩形外有更近点的可能性，所以我们必须小心才是。
- *网格文件*：网格文件在每一维上切分点空间。网格线间的距离可以不同，且每一维上的网格线数目也可以不同。只要数据分布得相当均匀，网格文件就能很好地支持范围查询、部分匹配查询和最邻近查询。
- *分段散列表*：分段散列函数从每一维上构造桶号的一些二进制位。它们较好地支持部分匹配查询，且不依赖于数据的均匀分布。

- *多键索引*：一个简单的多维结构有一个根，根是某个属性的索引，它导入第二个属性上的索引集合，而第二个属性的索引又导入第三个属性的索引集合，等等。它们对于范围查询和最邻近查询有用。
- kd树：这些树像二叉搜索树，但它们按不同层次在不同属性上分枝。它们较好地支持部分匹配查询、范围查询和最邻近查询。为了使该结构适合二维辅存操作，需要把多个树 710
结点压缩到一个块。
- *四叉树*：四叉树划分多维立方体成四个象限，且若它们有太多的点，则递归地用同样的方式划分这些象限。它们支持部分匹配查询、范围查询和最邻近查询。
- R树：这种树的结构通常表示成区域的集合，且通过聚集它们成一个更大区域的层次结构。它对于where-am-I查询有帮助。如果原子区域实际上是点，它将同样支持在本章中研究的其他类型的查询。
- *位图索引*：这种索引结构支持多维查询。它排序点或记录，并且通过位向量表示记录的位置。这些索引支持范围查询、最邻近查询和部分匹配查询。
- *压缩位图*：为了节省由很少个1的向量组成的位图索引的空间，通过采用游程长度编码来对位图索引进行压缩。

14.6 参考文献

本节讨论的数据结构都是20世纪70年代或80年代早期的研究成果。kd树来自文献[2]。为适合辅存所作的修改出现在文献[3]和[13]。分段散列和它在部分匹配检索的使用来自文献[12]和[5]。不过，习题14.2.8的设计思想来自文献[14]。

网格文件首先出现在文献[9]中。四叉树出自文献[6]。R树来自文献[8]。且文献[15]和[1]中的两个扩展很有名。

位图索引有一段有趣的历史。有一个由Ted Glaser创立、名为Nucleus的公司，它申请了这个构想的专利，并开发了一个DBMS，在该系统中位图索引既是索引结构又是数据表示。该公司在20世纪80年代后期倒闭，但这种构想最近才被结合到几个主要的商用数据库系统中去。第一本关于这个主题的出版著作是文献[10]。文献[11]是这个构想的最新发展。

有许多关于多维存储结构的综述。文献[4]是最早的文献之一，最近的综述可在文献[16]和[7]中找到。前者还包括几个其他重要的数据库主题的综述。

1. N. Beekmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1990), pp. 322–331.

2. J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Comm. ACM* **18**:9 (1975), pp. 509–517. 711

3. J. L. Bentley, "Multidimensional binary search trees in database applications," *IEEE Trans. on Software Engineering* **SE-5**:4 (1979), pp. 333-340.

4. J. L. Bentley and J. H. Friedman, "Data structures for range searching," *Computing Surveys* **13**:3 (1979), pp. 397–409.

5. W. A. Burkhard, "Hashing and trie algorithms for partial match retrieval," *ACM Trans. on Database Systems* **1**:2 (1976), pp. 175–187.

6. R. A. Finkel and J. L. Bentley, "Quad trees, a data structure for retrieval on composite keys," *Acta Informatica* **4**:1 (1974), pp. 1–9.

7. V. Gaede and O. Gunther, "Multidimensional access methods," *Computing Surveys* **30**:2 (1998), pp. 170–231.

8. A. Guttman, "R-trees: a dynamic index structure for spatial searching," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1984), pp. 47–57.

9. J. Nievergelt, H. Hinterberger, and K. Sevcik, "The grid file: an adaptable, symmetric, multikey file structure," *ACM Trans. on Database Systems* **9**:1 (1984), pp. 38–71.

10. P. O'Neil, "Model 204 architecture and performance," *Proc. Second Intl. Workshop on High Performance Transaction Systems*, Springer-Verlag, Berlin, 1987.

11. P. O'Neil and D. Quass, "Improved query performance with variant indexes," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1997), pp. 38–49.

12. R. L. Rivest, "Partial match retrieval algorithms," *SIAM J. Computing* **5**:1 (1976), pp. 19–50.

13. J. T. Robinson, "The K-D-B-tree: a search structure for large multidimensional dynamic indexes," *Proc. ACM SIGMOD Intl. Conf. on Mamagement of Data* (1981), pp. 10–18.

14. J. B. Rothnie Jr. and T. Lozano, "Attribute based file organization in a paged memory environment, *Comm. ACM* **17**:2 (1974), pp. 63–69.

15. T. K. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-tree: a dynamic index for multidimensional objects," *Proc. Intl. Conf. on Very Large Databases* (1987), pp. 507–518.

16. C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari, *Advanced Database Systems*, Morgan-Kaufmann, San Francisco, 1997.

712

第15章 查询执行

前面几章给我们提供了一些数据结构，它们有助于支持基本的数据库操作，如根据查询关键字查找元组。我们现在可以利用这些结构来支持响应查询的有效算法。查询处理将在本章与第16章中讨论。查询处理器是DBMS中的一个部件集合，它能够将用户的查询和数据修改命令转变为数据库上的操作序列，并且执行这些操作。既然SQL允许我们在很高的层次上表达查询，那么查询处理器必须提供关于查询将被如何执行的大量细节。此外，查询的一个幼稚的执行策略可能导致采用的查询执行算法所用时间比必需的高出许多倍。

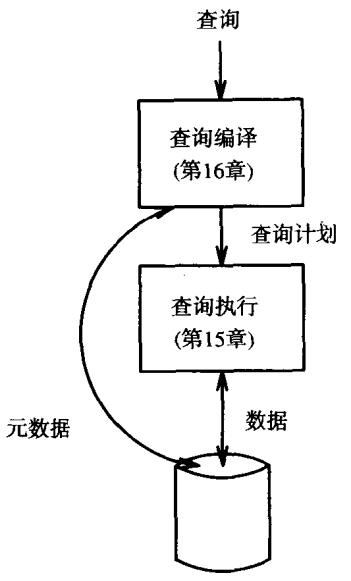


图15-1 查询处理器的主要部分

图15-1表明了第15章与第16章的主题的划分。在本章中，我们将集中在查询执行上，也就是操作数据库数据的算法。我们将集中讨论5.4节所述的扩展关系代数的操作，由于SQL使用包模型，因此我们也假设关系都是包，从而可使用5.3节中的关于包的操作符。

本章列出了执行关系代数运算的基本方法。这些方法的基本策略有所不同，扫描、散列、排序和索引是主要的方式。这些方法对可得到的主存容量上所做的假设也有所不同；有些算法假设可得到的主存至少能够容纳参加操作的一个关系。另一些算法假设操作对象太大以至于不能装在主存中，这些算法在代价和结构上有明显的差别。

查询编译预览

正如图15-2所描绘的，查询编译可以分为三个主要步骤：

a) 分析，在这个过程中构造分析树，用来表达查询和它的结构。

b) 查询重写，在这个过程中分析树被转化为初始查询计划，这种查询计划通常是查询的代数表达式。然后，初始查询计划被转化为一个预期所需执行时间较小的等价的计划。

c) 物理计划生成，在这一步，通过为(b)中抽象的查询计划，即通常所谓的逻辑查询计划的每一个操作符选择实现算法，并选择这些操作符的执行顺序，逻辑计划被转化为物理查询计划。与分析结果和逻辑计划一样，物理计划用表达式树来表示。物理计划还包含许多细节，如被查询的关系是怎样被访问的，以及一个关系何时或是否应当被排序。

(b)和(c)部分常被称做查询优化器，它们是查询编译的难点。第16章专门讨论查询优化，我们将在那里学习怎样选择一个占用时间尽可能少的“查询计划”。为了选择最好的查询计划，我们需要判断：

1. 查询的哪一个代数等价形式会为回答查询带来最有效的算法？
2. 对选中形式的每一个操作，我们应当使用什么算法来实现？
3. 数据如何从一个操作传到另一个操作，比如使用流水线方式、主存缓冲区还是通过磁盘？

714 这些选择中的每一个都依赖于关于数据库的元数据。查询优化可利用的典型的元数据包括：每个关系的大小；统计数据，如一个属性的不同值的近似数目和频率；某些索引的存在以及数据在磁盘上的分布。

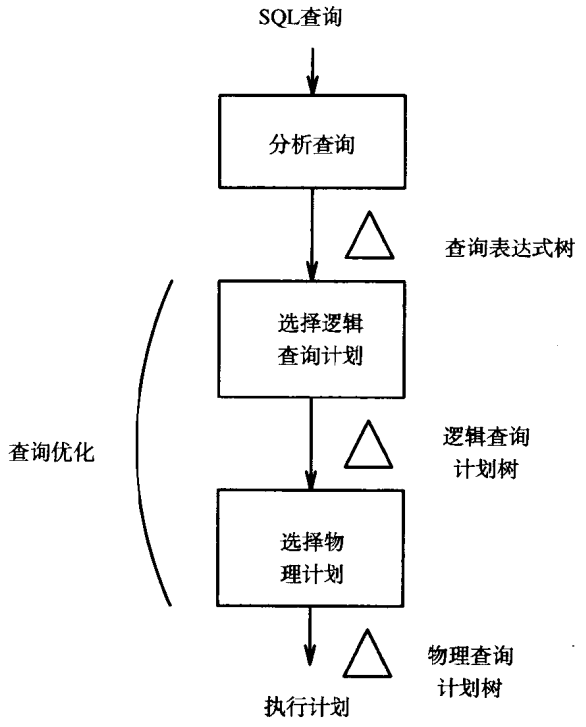


图15-2 查询编译概貌

15.1 物理查询计划操作符介绍

物理查询计划由操作符构造，每一个操作符实现计划中的一步。物理操作符常常是一个关系代数操作符的特定的实现。但是，我们也需要用物理操作符来完成另一些与关系代数操作符无关的任务。例如，我们经常需要“扫描”一个表，即将作为关系代数表达式的操作数的某个关系的每个元组调入内存。本节中我们将介绍物理查询计划的基本构造块，后面的章节中包括有效地实现关系代数操作符的更复杂的算法，这些算法也是物理查询计划的一个必不可少的部分。我们还在这里介绍“迭代器”的概念，它是使包含在一个物理查询计划中的操作符之间能
715 够传递对元组的请求以及结果的一个重要方法。

15.1.1 扫描表

可能我们在一个物理查询计划中可以做的最基本的事情是读一个关系R的整个内容。有时候，例如将R与另一个关系做并或连接时，这一步是必须的。这个操作符的一个变体包含一个简单的谓词，我们仅读出关系R中那些满足这个谓词的元组。定位关系R中的元组的基本方法有两种。

1. 在很多情况下，关系R存放在第二级存储器的某个区域中，它的元组排放在块中。系统知道包含R的元组的块，并且可以一个接一个地得到这些块。这个操作叫做*表-扫描*。

2. 如果R的任意一个属性上有索引，我们可以使用这个索引来得到R的所有元组。比如，R上的一个如13.1.3节中所讨论的那样的稀疏索引，可以用来引导我们得到所有包含R的块，即

使除此之外我们并不知道哪些是包含R的块。这个操作叫做*索引-扫描*。

我们将在15.6.2节讨论σ操作符的实现时再次考虑索引-扫描。然而，就目前来说，最重要的事实是我们不仅可以通过索引得到它索引的关系的所有元组，还可以通过索引得到在构成索引的属性或属性组合上具有特定值(或有时是一个特定值的范围）的那些元组。

15.1.2 扫描表时的排序

在读一个关系的元组时，有很多原因促使我们将关系排序。其中一个是查询可能包含一个ORDER BY子句，要求对关系做排序。另一个原因是关系代数运算的许多种算法要求一个或所有的操作对象是排序的关系。这些算法在15.4节和其他一些地方出现。

物理查询计划操作符*排序-扫描*接受关系R以及对作为排序依据的属性组的说明，并产生排好顺序的R。实现排序-扫描的方法有多种：

a) 如果我们想产生按照属性a排序的关系R，并且a上有一个B树索引，或R是作为按a排序的索引顺序文件来存储的，那么对索引进行扫描使我们得到具有所需顺序的R。

b) 如果我们想要排序的关系R很小，可以装进内存，那么可以使用表扫描或索引扫描来得到它的元组，再使用许多可供使用的有效的内存排序算法中的一种。内存排序在许多这方面的书中进行了讨论，这里我们不再考虑。 716

c) 如果R太大以至于不能装进内存，那么11.4.3节中的多路归并方法是一个较好的选择。但是，我们并不将最终好排序的R存回磁盘，而是根据对元组的需要一次产生R的一个排好序的块。

15.1.3 物理操作符计算模型

一个查询通常包括几个关系代数运算，相应的物理查询计划由几个物理操作符组成。一个物理操作符通常是一个代数操作符的实现，但正如我们在15.1.1节见到的那样，另外有一些物理计划操作符对应的操作(如扫描）可能是关系代数中见不到的。

既然明智地选择物理计划操作符是一个好的查询处理器所必不可少的，我们必须能够估价我们使用的每个操作符的“代价”。我们将使用磁盘I/O的数目作为衡量每个操作的代价的标准。这个衡量标准与我们的一个观点（见11.4.1节）是一致的，即从磁盘中得到数据的时间比对内存中的数据做任何有用操作花费的时间都长。主要的例外情况是在回答查询需要通过网络进行数据通信时。我们在15.9节和19.4.4节讨论分布式查询处理。

在比较相同操作的算法时，我们将做一个假设，它可能会使我们在开始时感到惊讶：

- 我们假设任何操作符的操作对象都位于磁盘上，但操作符的结果放在内存中。

如果操作符产生一个查询的最终结果，这个结果需要写到磁盘上，那么保存结果的代价仅仅依赖于结果的大小，而不依赖于结果是怎样被计算的。我们可以简单地将最后的回写代价加到这个查询的总代价上。但是，在许多应用中，结果根本不存放到磁盘上，而是打印或传送到某个格式化程序。于是，在输出上耗费的磁盘I/O或者是零，或者依赖于某个未知的程序对数据所做的操作。

同样，形成一个查询的部分(而非整个查询）操作符的结果通常也不写到磁盘上。在15.1.6节中我们将讨论“迭代器”，其中一个操作符的结果在内存中构造，每次可能是一小部分，并作为操作对象传递给另一个操作符。在这种情况下，我们不必将结果写到磁盘上，而且，还节省了使用这一结果作为操作对象的操作符从磁盘上读取该操作对象的开销。这一节省为查询优化器提供了极好的机会。

15.1.4 衡量代价的参数

现在，让我们引入用来表达一个操作符代价的参数。如果优化器想确定许多查询计划中的 717

哪一个执行最快，那么估计代价是必须的。16.5节将介绍这些代价估算的使用。

我们需要一个参数来表达操作符使用的内存大小，还需要其他参数来衡量它的操作对象的大小。假设内存被分成缓冲区，缓冲区的大小与磁盘块的大小相同。那么M表示一个特定的操作符执行时可以获得的内存缓冲区的数目。记住，当估算一个操作符的代价时，我们不考虑产生输出结果所用内存或磁盘I/O的代价，因而M仅包含容纳输入和操作符的所有中间结果使用的空间。

有时候，就像我们在11.4.4节中所做的那样，我们可以认为M是整个内存或内存的绝大部分。但是，我们也将看到几个操作共享内存的情况，这时M比整个内存要小得多。事实上，就像我们将在15.7节中讨论的那样，一个操作可得到的缓冲区的数目可能不是一个可以预计的常数，而可能在执行过程中根据同时执行的其他进程来决定。如果是这样，M实际上是对一个操作可得到的缓冲区数目的估计。如果估计错误，那么实际的执行时间将不同于优化器预计的时间。我们甚至可能发现，如果查询优化器知道执行时真正的缓冲区可用情况，那么选出的物理查询计划可能就会不同。

另外，让我们来考虑测量访问参与操作的关系所需代价的参数。这些测量关系中数据的多少和分布的参数经常被定期地计算，以便帮助查询优化器选择物理操作符。

我们将做一个简化的假设，即在磁盘上一次访问一个块的数据。实际上，如果我们能够一次读一个关系的许多块，这些块可能来自一个磁道上连续的块，那么11.5节中讨论的某项技术可能会提高算法的速度。有三类参数，B、T和V：

- 当描述一个关系R的大小时，绝大多数情况下，我们关心包含R的所有元组所需的块的数目。这个块的数目表示为$B(R)$，或者如果我们知道指的是关系R，就可以仅仅表示为B。通常，我们假设R是聚簇的，即R存储在B个块中或近似B个块中。正如在13.1.6节中讨论的那样，实际上我们可能希望在保存R的每个块中留出一小部分空闲空间，以备将来向R中插入数据。不过，对于为得到完整的R而需要从磁盘上读取的块数来说，B通常是一个足够好的近似。我们将使用B来作为统一的估计值。
- 有时候，我们也需要知道R中的元组的数目，我们将这个数表示为$T(R)$，或在我们知道所指关系为R时简记为T。如果需要一个块中能容纳的R的元组数，我们可以使用比率T/B。
718 此外，有一些情况下，一个关系分布地存储在若干块中，这些块同时还被其他关系的元组占用。如果这样，那么一个简化的假设是R的每个元组要求一个单独的磁盘读，我们将使用T作为这种环境下读R所需磁盘I/O数的估算。
- 最后，我们有时候希望参考出现在关系的列中的不同值的数目。如果R是一个关系，它的一个属性是a，那么$V(R,a)$是R中a对应列上不同值的数目。更一般地讲，如果$[a_1, a_2, ..., a_n]$是一个属性列表，那么$V(R,[a_1, a_2, \cdots, a_n])$是$R$中属性$a_1, a_2, \cdots, a_n$对应列不同的$n$元组的数目。换言之，它是$\delta(\pi_{a_1, a_2, \cdots, a_n}(R))$中的元组数目。

15.1.5 扫描操作符的I/O代价

作为前面介绍的参数的一个简单应用，我们可以表示迄今为止讨论过的每个表-扫描操作符的磁盘I/O数目。如果关系R是聚簇的，那么表-扫描操作符的磁盘I/O数目近似为B。同样，如果R能够装入全部主存，那么我们可以通过将R读入主存并做内排序，从而实现排序-扫描，所需磁盘I/O数仍是B。

如果R聚簇，但需要两阶段多路合并排序，那么，正如11.4.4节中讨论的那样，我们需要大约$3B$次磁盘I/O，它们相等地分布于在子表中读R、回写子表和重读子表的操作中。记住，我们

不将最终结果的写操作计入，也不计入积累的输出结果所占用的内存空间。而是假设每一个输出块立即被某个其他的操作消耗；有可能就是写回磁盘而已。

但是，如果R不是聚簇的，那么所需的磁盘I/O数通常较高。如果R分布在其他关系的元组之间，那么表-扫描所需读的块数可能与R的元组一样多；即I/O代价为T。类似地，如果我们想把R排序，但是R能被内存容纳，那么T就是将R的全部元组调入内存所需的磁盘I/O数。最后，如果R不是聚簇的，而且需要进行两阶段排序，那么最初读入子表需要花费T次磁盘I/O。但是，我们可以用聚簇的方式存储和重读子表，因此这些步骤仅需要$2B$次磁盘I/O。在一个大的、非聚簇的关系上执行排序-扫描操作的总代价是$T+2B$。

最后，让我们考虑索引-扫描的代价。通常，关系R的一个索引需要的块数比$B(R)$少许多。因此，扫描整个R(至少需B次磁盘I/O）比查看整个索引需要更多的I/O数目。这样，即便索引-扫描既需要检查关系又需要检查它的索引，

- 我们仍继续用B或T作为使用索引时访问整个聚簇或不聚簇的关系的代价估计。 719

但是，如果不只想要R的一部分，我们通常能够避免查看整个索引和整个R。我们将这些索引的使用推迟到15.6.2节中分析。

15.1.6 实现物理操作符的迭代器

许多物理操作符可以作为迭代器实现。迭代器是三个函数的集合，这三个函数允许物理操作符结果的使用者一次一个元组地得到这个结果。形成一个操作的迭代器的三个函数是：

为何使用迭代器?

我们将在16.7节中看到在查询计划中，迭代器如何组合起来以支持有效的执行。它们与物化策略相反，物化策略产生每个操作符的整个结果，或者将它存放在磁盘上，或者允许它在内存中占据空间。在使用迭代器时，同一时刻活跃的操作有许多。元组按照需要在操作符之间传递，这样就减少了存储要求。当然，正如我们将见到的那样，并非所有物理操作符对迭代方法或“流水线”的支持都是有意义的。在某些情况下，几乎所有的工作都需要Open函数来完成，这样就等效于物化方法了。

1. `Open`。这个函数启动获得元组的过程，但并不获得元组。它初始化执行操作所需的任何数据结构，并为操作的任何操作对象调用`Open`。

2. `GetNext`。这个函数返回结果中的下一个元组，并且对数据结构作必要的调整以得到后续元组。在获取结果的下个一元组时，它通常在操作对象上一次或多次调用`GetNext`。这个函数还设置一个表明是否一个元组已产生或已经没有元组可产生的信号。我们将用`Not Found`来作为一个布尔变量，它当且仅当新元组返回时为真。

3. `Close`。这个函数在所有的元组或使用者想得到的所有元组都获得后结束迭代。它通常为操作符的每个操作对象调用`Close`。

当描述迭代器和它的函数时，我们假设每一类的迭代器（即每一类作为迭代器来实现的物理操作符），都有一个“类”。这个类的实例支持Open，GetNext和Close方法。 720

例15.1 最简单的迭代器可能是表-扫描操作符的实现。假设我们想执行`TableScan(R)`，其中R是聚集在某个块序列中的关系，我们可以很方便地访问它。因此，我们假设“得到R的下一个块”这一想法由存储系统实现，不需要详细描述。此外，我们假设块内有一个记录(元组）的目录，这样可以容易地得到块中的下一个元组或者判断是否到达最后一个元组。

图15-3简略描述了这个迭代符的三个函数。我们设想有块指针b和指向块b中元组的元组指

针t。假设这两个指针分别可以“超出”最后一个块和最后一个元组，并且在这两种情况发生时能够指明。注意在这个例子中Close没做什么事。实际上，迭代符的Close函数可能以各种方式清除DBMS的内部结构。它可能通知缓冲区管理器某些缓冲区已不再需要，或者通知并发管理器关系的读操作已经完成。 □

```
Open() {
    b := the first block of R;
    t := the first tuple of block b;
}

GetNext() {
    IF (t is past the last tuple on block b) {
        increment b to the next block;
        IF (there is no next block)
            RETURN NotFound;
        ELSE /* b is a new block */
            t := first tuple on block b;
    }  /* now we are ready to return t and increment */
    oldt := t;
    increment t to the next tuple of b;
    RETURN oldt;
}

Close() {
}
```

图15-3 表-扫描操作符的一个迭代器

721 **例15.2** 现在，让我们考虑一个迭代器在它的Open函数中执行大部分工作的例子。操作符是排序-扫描，其中我们读一个关系R的元组，按照排好的顺序将它们返回。此外，让我们假设R很大，我们需要使用11.4.4节中所述的两阶段、多路归并排序。

在检查完R的每一个元组之前，我们甚至连第一个元组也不能返回。因此Open至少必须做下面的工作：

1. 将 R 的所有元组读入若干大小与主存相等的chunk中，将它们排序，并把它们存储到磁盘上。

2. 为第二阶段(归并）初始化数据结构，并将每个子表的第一块装入主存数据结构中。

然后，GetNext就可以让所有子表头部剩下的第一个元组进行竞争。如果来自获胜子表的块已经空了，GetNext就重新装载其缓冲区 □

例15.3 最后，我们考虑一个关于多个迭代器怎样通过调用其他迭代器而结合起来的简单例子。对于多个迭代器怎样能同时处于活跃状态来说，这并不是一个很好的例子，这个问题需要推迟到我们考虑过像选择或连接这样的物理操作符的算法后，这些算法能更好地开发迭代器的能力。

我们的操作是包的并$R \cup S$，这里我们首先产生R的所有元组，再产生S的所有元组，而不必考虑是否存在重复。我们假设有形成R的迭代器的函数R.Open、R.GetNext和 R.Close，

对于关系S也有类似的函数。如果R和S是存储的关系，那么这些函数可能是施加于R和S上的表-扫描函数，否则可能是调用其他迭代器的网络来计算R和S的迭代器。这一并操作的迭代器函数在图15-4中进行了描述。巧妙之处在于函数使用了一个共享变量CurRel，它是R还是S取决于当前正在读哪一个关系。 □

```
Open() {
    R.Open();
    CurRel := R;
}

GetNext() {
    IF (CurRel = R) {
        t := R.GetNext();
        IF (t <> NotFound) /* R is not exhausted */
            RETURN t;
        ELSE /* R is exhausted */ {
            S.Open();
            CurRel := S;
        }
    }
    /* here, we must read from S */
    RETURN S.GetNext();
    /* notice that if S is exhausted, S.GetNext()
       will return NotFound, which is the correct
       action for our GetNext as well */
}

Close() {
    R.Close();
    S.Close();
}
```

图15-4 用部件构建一个并迭代器

15.2 数据库操作的一趟算法

我们现在将开始学习查询优化中一个非常重要的问题：怎样执行逻辑查询计划中的每个单独的步骤(例如，连接或选择)？每一个操作符的算法的选择是将逻辑查询计划转变成物理查询计划过程中的一个必不可少的部分。关于操作已提出了很多算法，它们大体上分为三类：

1. 基于排序的方法。这些方法主要在15.4节中讲述。 722

2. 基于散列的方法。这些方法在15.5节、15.9节以及其他一些地方提到。

3. 基于索引的方法。这些方法在15.6节中着重提出。

另外，我们可以将操作符算法按照难度和代价分成三种“等级”：

a) 一些方法仅从磁盘读取一次数据，这就是一趟算法。它们是本节的主题。尽管存在例外，尤其是像15.2.1节讨论的选择和投影，但通常仅当操作的至少一个操作对象能装入内存时它们才有效。

b) 一些方法处理的数据量太大以至于不能装入内存，但又不是可想像的最大的数据集合。 723
这种算法的一个例子是11.4.4节中的两阶段、多路归并排序。这些两趟算法的特点是首先从磁盘读一遍数据，用某种方式处理，并将全部或绝大部分写回磁盘，然后在第二趟中为了进一步处理，再读一遍数据。我们将在15.4节和15.5节中见到这些算法。

c) 某些方法对处理的数据量没有限制。这些方法用三趟或更多趟来完成工作，它们是对两阶段算法的自然的递归的推广。我们将在15.8节学习多趟方法。

本节中，我们主要讨论一趟算法。然而，无论在本节中还是在后面，我们都将把操作符分为三大类：

1. 一次多元组，一元操作。这类操作(选择和投影）不需要一次在内存中装入整个关系，甚至也不需要关系的大部分。这样，我们一次可以读一个块，使用内存缓冲区，并产生我们的输出。

2. 整个关系，一元操作。这些单操作对象的操作需要一次从内存中看到所有或大部分元组，因此，一趟算法局限于大小约为M或更小的关系。这里我们考虑的属于这一类的操作是γ和δ。

3. 整个关系，二元操作。其他所有的操作可以归为这一类：并、交、差、连接和积的集合形式以及包形式。我们将发现，如果要用一趟算法，那么这类操作中的每一个都要求至少一个操作对象的大小限制在M以内。

15.2.1 一次多元组操作的一趟算法

无论关系R能否被内存容纳，一次多元组运算$\sigma(R)$和$\pi(R)$都有显而易见的算法。正如图15-5所示，我们一次读取R的一块到输入缓冲区，对每一个元组进行操作，并将选出的元组或投影得到的元组移至输出缓冲区。由于输出缓冲区可能是其他操作的输入缓冲区，或正在向用户或应用发送数据，因而我们不把输出缓冲区算在所需空间内。因此，不管B怎样，我们只要求输入缓冲区满足$M \geqslant 1$。

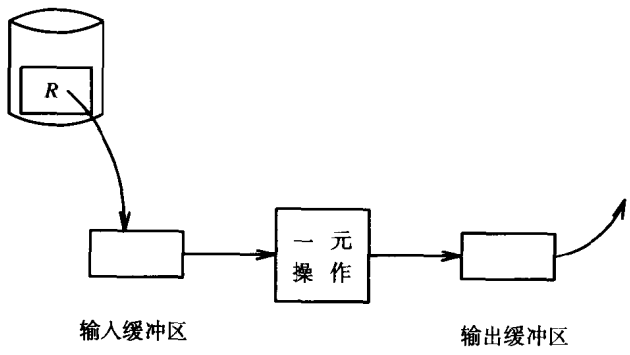


图15-5 在关系R上执行选择或投影运算

这一过程的磁盘I/O需求取决于作为操作对象的关系R是怎样提供的。如果R最初在磁盘上，那么代价就是执行一个表-扫描或索引-扫描所需的代价。这一代价已在15.1.5中讨论过；通常，如果R是聚集的，代价就是B；如果R不是聚集的，代价就是T。然而，我们需要再次提醒读者，当执行的操作是一个选择，其条件是比较一个常量和一个带索引的属性时，这是一个重要的例外。这种情况下，我们可以使用索引来检索R所在块的一个子集，这样通常会显著地提高执行效率。

724

额外的缓冲区可以加快操作

正如图15-5所指示的，尽管一次多元组操作只通过一个输入缓冲区和一个输出缓冲区可以实现，但如果分配更多的缓冲区可以加速处理过程。这个想法最初在11.5.1节中出现。如果R存储在柱面上连续的块中，那么我们可以读取完整的柱面到缓冲区，尽管为每一个柱面仅一个块的查询时间和旋转延迟付出了代价。类似地，如果操作的输出可以存储在全部的柱面上，我们在写入上几乎不浪费时间。

15.2.2 全关系的一元操作的一趟算法

现在让我们考虑施加于整个关系上而非施加于元组上的一元操作：消除重复(δ）和分组(γ)。

消除重复

为了消除重复，我们可以一次一个地读取R的每一块，但是对每一个元组，我们需要判断是否：

1. 这是我们第一次看到这个元组，这时我们将它复制到输出；或者 725

2. 我们从前见过这个元组，这时我们不必输出它。

为支持这个判定，我们需要为见过的每一个元组在内存中保存一个备份，如图15-6所示。一个内存缓冲区保存一个R的元组的块，其余的$M-1$个缓冲区可以用来保存到目前为止我们见过的每个元组的一个副本。

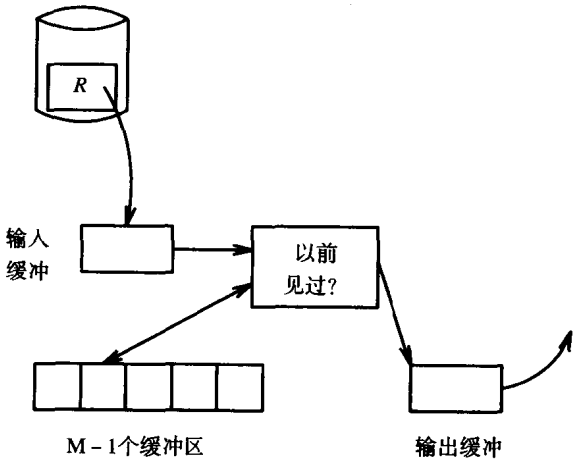


图15-6 一趟消除重复的内存管理

当存储已经见过的元组时，必须注意我们使用的内存数据结构。我们可以简单地列出我们见过的所有元组。当考虑R中的一个新元组时，我们将它与迄今为止看到的所有元组比较，如果它与这些元组当中的任何一个都不相等，就把它复制到输出，并将它加入到存在于内存中我们所看到的元组的列表中。

然而，如果内存中有 n 个元组，每一个新元组占用的处理器时间与 n 成比例。因此整个操作占用的处理器时间与 n^2 成比例。由于 n 可能会非常大，对于我们所作的只有磁盘I/O需要大量时间这一假设来说，这样的时间量将带来严重的问题。因此，我们需要一个主存结构，它允许下面的每一项操作：

1. 增加一个新元组，以及

2. 辨别一个给定的元组是否已经存在；

在接近于一个常量的时间内完成，而不依赖于目前我们在内存中拥有的元组数量n。已知的这样的结构有很多。例如，我们可以使用具有大量桶的散列表或某种形式的平衡二分查找树[⊖]。 726
每一种结构除了需要存储元组的空间外，还需要一些开销。例如，一个主存散列表需要一个桶数组和连接桶内元组的指针空间。然而，所需额外空间与存储元组所需空间相比一般较小。因此我们将做一个简化的假设，即不需要额外的空间，并把重点放在内存中存储元组的空间需求上。

基于这个假设，我们可以在主存的$M-1$个可用缓冲区存储与R的$M-1$个块所能容纳的一样多的元组。如果我们希望R的每个惟一的元组的一个副本能装在主存中，那么$B(\delta(R))$肯定不能超过$M-1$。因为我们预计M远远大于1，我们将经常用到的这个规则的一个简单近似是：

- $B(\delta(R)) \leqslant M$

注意，通常在没有计算出$\delta(R)$本身时，我们不能计算$\delta(R)$的大小。如果我们低估了这个值，因而$B(\delta(R))$实际上大于M，那么我们将为系统颠簸付出惨重的代价，因为保存R中惟一元组的块必须频繁地出入主存。

⊖ 关于合适的内存数据结构的讨论，请参见Aho、A.V. J. E. Hopcroft, 和J. D. Ullman 的《Data Structures and Algorithm》(Addison-Wesley, 1984)。特别地，散列平均花费$O(n)$的时间来处理n项，平衡树花费$O(n\log n)$的时间。这两者都能足够接近我们的线性目标。

分组

分组操作γ_L给我们零个或多个分组属性以及可能的一个或多个聚集属性。如果我们在主存中为每一个组（也就是为分组属性的每一个值）创建一个项，那么我们可以一次一块地扫描R的元组。每个组的项包括分组属性的值和每个聚集的一个或多个累计值。除了一种情况外，累计的值是显而易见的：

- 对MIN(a)或MAX(a)聚集来说，分别记录组内迄今为止见到的任意元组在属性a上的最小或最大值。每当见到组中的一个元组时，如果合适，就改变这个最小值或最大值。
- 对于任意COUNT聚集来说，为组中见到的每个元组加1。
- 对SUM(a)来说，在迄今为止扫描到的累加值上增加属性a的值。
- AVG(a)的情况复杂。我们必须保持两个累计：组内元组个数以及这些元组在a上的值的和。二者的计算分别与我们为COUNT和SUM聚集所做的一样。当R中所有元组都被扫描后，计算总和和个数的商以得到平均值。

727

当R的全部元组都已经读到输入缓冲区中，并且已用于各自分组中聚集的计算时，我们就可以通过为每个组写一个元组来产生输出。注意，直到扫描最后一个元组后，我们才开始为γ操作创建输出。这一情况的一个推论是，这种算法并不太适合迭代器结构；在第一个元组被GetNext检索到以前，全部的组必须用Open函数分好。

为了使每一个元组在内存的处理过程更有效，需要使用一个内存数据结构来帮助我们在已知分组属性值时找到各分组的项。就像前面讨论的δ操作那样，通常的内存数据结构，如散列表和平衡二分检索树能发挥很好的作用。然而我们应该记住，这种结构的查找关键字只能是分组属性。

这个一趟算法所需磁盘I/O数是B，与任何一元运算的一趟算法相同。尽管通常情况下M将小于B，所需内存缓冲区数M与B的关系不是任何一种简单的形式。问题在于组的项可能比R的元组长一些或短一些，并且组的数目可能是等于或小于R元组的数目的任意一种情况。然而大多数情况下，组的项不会比R的元组长，而且组的数目远小于元组数目。

非聚簇数据上的操作

记住，我们所有关于操作所需磁盘I/O数的计算都是在操作对象是聚簇的这一假设基础上进行的预测。在操作对象R没有聚簇的情况下(通常较罕见)，读取R的全部元组可能需要我们进行$T(R)$次而非$B(R)$次磁盘I/O。然而请注意，任何作为操作结果的关系总是可以被设想为聚簇的，因为我们没有理由以非聚簇的形式存储临时关系。

15.2.3 二元操作的一趟算法

现在我们开始讨论二元操作：并、交、差、积和连接。由于在某些情况下我们必须区分这些操作符的集合版本和包版本，我们用下标B和S来分别表示“包”和“集合”，例如，$\cup_B$是包的并，$-S$是集合的差。为了简化连接的讨论，我们将仅考虑自然连接。将属性适当地重命名后，等值连接可以按照相同方式实现，并且θ连接可以被认为是在积或等值连接后再跟上那些在等值连接中不能表达的条件。

包的并可以通过一种非常简单的一趟算法计算出来。为了计算$R \cup_B S$，我们复制元组R的每一个元组到输出，然后复制S的每一个元组，就像我们在例15.3中所做的那样。磁盘I/O数是$B(R) + B(S)$，正如操作对象R和S上的一趟算法所必需的那样，并且不管R和S多么大，$M = 1$就足够了。

其他的二元操作需要将R和S中较小的那个操作数读到内存中，并且建立一个合适的数据结构，就像15.2.2节中讨论的那样，使元组不仅可以被快速插入还可以被快速检索到。和前面一样，散列表或平衡树就可以满足要求。这种结构需要少量空间(除了元组本身的空间)，我们将忽略不计。因此，在关系R和S上用一趟执行一个二元操作的近似需求是：

- $\min(B(R),B(S)) \leqslant M$

728

这个规律假定一个缓冲区将被用来读取较大关系的块，而大约M个缓冲区用来容纳整个较小的关系和它的内存数据结构。

现在我们将给出各种操作的细节。在每一种情况下，我们假定R是两个关系中较大的一个，并且我们将把S放在内存中。

集合并

我们将S读到内存的$M-1$个缓冲区中并且建立一个查找结构，其查找关键字是整个元组。所有的这些元组也都复制到输出。然后我们一次一块地将R的每一块读到第M个缓冲区。对于R的每一个元组t，我们观察t是否在S中，如果不在，我们就将t复制到输出。如果t也在S中，我们就跳过t。

集合交

将S读到$M-1$个缓冲区中，并建立将整个元组作为查找关键字的查找结构。读取R的每一个块，并且对R的每个元组t，观察t是否也在S中。如果在，我们将t复制到输出；而如果不在，则忽略t。

集合差

既然差不是一种可交换的操作符，我们必须区别$R-_S S$和$S-_S R$，并继续假设R是较大的关系。在两种情况下，我们都将S读到$M-1$个缓冲区中，并建立将整个元组作为查找关键字的查找结构。

为了计算$R-_S S$，我们读取R的每一个块，并且检查块中的每一个元组t。如果t在S中，那么忽略t；如果t不在S中，则将t复制到输出。

为了计算$S-_S R$，我们读取R的每一个块，并依次检查每一个元组t。如果t在S中，那么我们从主存中S的副本里删掉t；而如果t不在S中，则我们不做任何处理。在考虑完R的每一个元组后，我们将S中剩余的那些元组复制到输出。

729

包交

我们将S读到$M-1$个缓冲区中，但是把每一个不同的元组与一个计数联系起来，其初值是该元组在S中出现的次数。元组t的多个副本并不分别存储。相反，我们存储t的一个副本并且将它与一个计数联系起来，计数值等于t出现的次数。

如果很少有重复的话，那么这种结构将占用比$B(S)$块稍大的空间，尽管结果经常是S被压缩。因此，我们将继续假设$B(S)\leqslant M$足以运行一趟算法，尽管这个条件只是一个近似。

接着，我们读取R的每一块，并且对于R的每一个元组t我们观察t是否在S中出现。如果不出现，那么我们忽略t；它不会出现在交中。然而，如果t在S中出现，并且与t对应的计数仍为正值，那么我们输出t并将计数减1。如果t在S中出现，但是它的计数器已经到0，那么我们不输出t；我们在输出中已经产生的t的副本和S中的一样多。

包差

为了计算$S-_B R$，将S的元组读到内存中，并且像我们在计算包交集时那样统计每一个不同的元组出现的次数。当读取R时，对每一个元组t，观察t是否在S中出现，如果是，那么我们将

与之对应的计数递减1。最后，将内存中计数是正数的每一个元组复制到输出，并且我们复制它的次数等于其计数。

为了计算$R -_B S$，我们也将S的元组读到内存中，并且统计每一个不同的元组出现的次数。当读取R的元组时，我们可以把具有计数是c的元组t看做是不将t复制到输出的c个理由。也就是说，当读取R的一个元组t时，我们观察t是否在S中出现。如果不出现，那么我们将t复制到输出。如果t确实在S中出现，那么我们看与t对应的计数c的当前值。如果c=0，那么我们将t复制到输出。如果c>0，那么不将t复制到输出，但是将c值减1。

乘积

将S读到主存的$M-1$个缓冲区中，不需要特殊的数据结构。然后读取R的每一个块，并且对R中的每一个元组t，将t与主存中S的每一个元组连接。在每一个连接而成的元组形成后即将其输出。

注意，对R的每一元组，这种算法都可能占用相当多的处理器时间，因为每一个这样的元组必须和装满元组的$M-1$个块相匹配。然而，输出所占空间也很大，因此每个输出的元组所用的时间很少。

如果不知道M会怎样?

尽管在我们对算法的介绍中可用内存块数M似乎是固定的并且是预知的，但是请记住，除了一些明显的限制例如机器的总的内存容量以外，可用块数M通常是未知的。因此，当查询优化器在一趟算法和两趟算法之间进行选择时，可能估计M值并且基于这种估计做出选择。如果优化器估计错误，其后果是造成缓冲区在磁盘和内存之间颠簸(如果对M的估计太高)，或者在低估M值时导致多余的趟数。

也有一些算法，在内存容量小于预期值时性能的降低不至于太唐突。例如，我们可以像一趟算法那样做，直到耗尽空间，然后再开始按两趟算法来做。15.5.6节和15.7.3节讨论一些这样的方法。

自然连接

730 在这一连接算法和其他连接算法中，我们沿袭惯例，即$R(X,Y)$与$S(Y,Z)$连接，Y表示R和S的所有公共属性，X是R的所有不在S的模式中的属性，并且Z是S的所有不在R的模式中的属性。我们继续假设S是较小的关系。要计算自然连接，请执行以下步骤：

1. 读取S的所有元组并且用它们构造一个以Y的属性为查找关键字的内存查找结构。和平常一样，散列表或平衡树是这种结构的很好的例子。将内存的$M-1$块用于这一目的。

2. 将R的每一块读到内存中剩下的那一个缓冲区中。对于R的每一个元组t，利用查找结构找到S中与t在Y的所有属性上相符合的元组。对于S中每一个匹配的元组，将它与t连接后形成一个元组，并且将结果元组移到输出。

和所有一趟的二元操作算法一样，这一算法读取操作对象需要使用$B(R)+B(S)$次磁盘I/O。只要$B(S)\leqslant M-1$或近似地$B(S)\leqslant M$，它就能正常工作。和我们学习过的其他算法一样的还有，内存的查找结构所需空间没有计入，但这可能需要一个小的、额外的内存空间。

我们不打算讨论自然连接以外的连接。记住，等值连接以与自然连接基本相同的方式执行，但是我们必须考虑两个关系的“相等”属性可能有不同的名字这一事实。不是等值连接的θ连
731 接可以用在等值连接或积之后加以选择来代替。

15.2.4 习题

习题15.2.1 对于下面的每一个操作，利用本节中描述的算法为其书写一个迭代器。

* a) 投影。
* b) 消除重复(δ)。
  c) 分组(γ_L)。
* d) 集合并。
  e) 集合交。
  f) 集合差。
  g) 包交。
  h) 包差。
  i) 乘积。
  j) 自然连接。

习题15.2.2 对于习题15.2.1中的每一个操作符，判别它是否是阻塞的，阻塞意味着直到所有的输入都读入以后才能产生第一个输出。换句话说，阻塞操作符惟一可行的迭代器是由Open完成所有重要的工作。

习题15.2.3 图15-9概括了本节和下一节中算法的内存和磁盘I/O需求。然而，它假设所有操作对象都是聚簇的。如果一个或所有操作对象不是聚簇的，图中的项将怎样变化?

! **习题15.2.4** 给出以下每一个类连接的一趟算法。

* a) $R \ltimes S$，假设R可装入内存。(参见习题5.2.10对半连接的定义)
* b) $R \ltimes S$，假设S可装入内存。
  c) $R \overline{\ltimes} S$，假设R可装入内存。(参见习题5.2.11对反半连接的定义)
  d) $R \overline{\ltimes} S$，假设S可装入内存。
* e) $R \overset{\circ}{\bowtie}_L S$，假设$R$可装入内存。(参见习题5.4.7节对外连接的定义)
  f) $R \overset{\circ}{\bowtie}_L S$，假设$S$可装入内存。
  g) $R \overset{\circ}{\bowtie}_R S$，假设$R$可装入内存。
  h) $R \overset{\circ}{\bowtie}_R S$，假设$S$可装入内存。
  i) $R \overset{\circ}{\bowtie} S$，假设$R$可装入内存。

732

15.3 嵌套循环连接

在讨论下一节中更为复杂的算法之前，我们将注意力转向一个称为“嵌套循环”连接的连接操作符算法系列。这些算法，在某种意义来说需要“一趟半”，因为在其中的各种算法中，两个操作对象中有一个的元组仅读取一次，而另一个操作对象将重复读取。嵌套循环连接可以用于任何大小的关系；没有必要要求一个关系必须能装入内存中。

15.3.1 基于元组的嵌套循环连接

我们将从嵌套循环系列中最简单的形式开始，其中循环包容了所涉及关系的各个元组。在这个我们称为*基于元组的嵌套循环连接*算法中，将计算连接

$$R(X,Y) \bowtie S(Y,Z)$$

如下：

```
FOR each tuple s in S DO
    FOR each tuple r in R DO
        IF r and s join to make a tuple t THEN
            output t;
```

如果我们不注意关系R和S的块的缓冲方法，那么这种算法需要的磁盘I/O可能多达$T(R)T(S)$。然而，在很多情况下，这种算法都可以作修改，使代价低得多。一种情况是当我们可以使用R的连接属性上的索引来查找与给定的S元组匹配的R元组时，这样的匹配不必读取整个关系R。我们在15.6.3节讨论基于索引的连接。第二种改进更加注重R和S的元组在各个块中的分布方式，并且在执行内层循环时，要尽可能多地使用内存，以减少磁盘I/O的数目。我们将在15.3.3节考虑基于块的嵌套循环连接形式。

15.3.2 基于元组的嵌套循环连接的迭代器

嵌套循环连接的一个优点是它非常适合用于迭代器结构，因此，就像我们将在16.7.3节中看到的那样，某些情况下它能使我们避免将中间关系存储到磁盘上。$R \bowtie S$的迭代器很容易用R和S的迭代器构造起来，我们用`R.Open()`等表示这些迭代器，就像15.1.6节中那样。嵌套循环
733 连接的三个迭代函数的代码如图15-7所示。它假定关系R和S都是非空的。

```
Open() {
    R.Open();
    S.Open();
    s := S.GetNext();
}

GetNext() {
    REPEAT {
        r := R.GetNext();
        IF (r = NotFound) { /* R is exhausted for
                the current s */
            R.Close();
            s := S.GetNext();
            IF (s = NotFound) RETURN NotFound;
                /* both R and S are exhausted */
            R.Open();
            r := R.GetNext();
        }
    }
    UNTIL(r and s join);
    RETURN the join of r and s;
}

Close() {
    R.Close();
    S.Close();
}
```

图15-7 基于元组的嵌套循环连接的迭代器函数

15.3.3 基于块的嵌套循环连接算法

如果按以下步骤计算$R \bowtie S$，我们可以改进15.3.1节中基于元组的嵌套循环连接：

1. 对作为操作对象的两个关系的访问均按块组织，并且
2. 使用尽可能多的内存来存储属于关系S的元组，S是外层循环中的关系。

第1点确保了当在内层循环中处理关系R的元组时，我们可以用尽可能少的磁盘I/O来读取R。

第2点使我们不是将读到的R的每一个元组与S的一个元组连接，而是与能装入内存的尽可能多的S元组连接。 734

像15.2.3节中一样，假设$B(S) \leq B(R)$，但是现在让我们再假定$B(S) > M$；也就是说，任何一个关系都不能完整地装入内存。我们重复地将$M-1$个块读到内存缓冲区中。为S在内存中的元组创建一个查找结构，它的查找关键字是R和S的公共属性。然后我们浏览R的所有块，依次读取每一块到内存的最后一块中。在录入R的块后，将块中所有元组与S在内存中的所有块的所有元组进行比较。对于那些能连接的元组，我们输出连接得到的元组。这种算法的嵌套循环结构可以在图15-8中看到，在那里我们更加正式地描述此算法。

```
FOR each chunk of M-1 blocks of S DO BEGIN
    read these blocks into main-memory buffers;
    organize their tuples into a search structure whose
        search key is the common attributes of R and S;
    FOR each block b of R DO BEGIN
        read b into main memory;
        FOR each tuple t of b DO BEGIN
            find the tuples of S in main memory that
                join with t;
            output the join of t with each of these tuples;
        END;
    END;
END;
```

图15-8 嵌套循环连接算法

图15-8的程序似乎有三重嵌套循环。然而，如果我们从正确的抽象层次上看代码，实际上仅有两重循环。第一重循环或外层循环是对S元组进行的，其他的两层循环对R的元组进行。然而，将此过程表达为两层循环是为了强调我们访问R的元组的顺序不是任意的。相反，我们需要一次一块地处理这些元组(第二层循环的作用)，并且在继续移动到下一个块之前，我们要处理当前块内的所有元组(第三层循环的作用)。

例15.4 假定$B(R)=1000$且$B(S)=500$，并令$M=101$。我们将使用100个内存块来按照大小为100块的chunk对S进行缓冲。因此图15-8中的外层循环需迭代5次。每一次迭代中，用100个磁盘I/O读取S的chunk，并且在第二层循环中必须用1000个磁盘I/O来完整地读取R。因此，磁盘I/O的总数量是5500。

注意，如果我们颠倒R和S的角色，算法使用的磁盘I/O要略多一些。我们将在外层循环中迭代10次，并且每一次迭代使用600次磁盘I/O，总共是6000次。一般来说，在外层循环中使用较小的关系略有优势。 □ 735

图15-8中的算法有时被称做“嵌套的块连接”。我们继续将其简单地称为*嵌套循环连接*，因为它是嵌套循环思想在实践中使用最广泛的实现形式。如果需要将它与15.3.1节的基于元组的嵌套循环连接区别开，我们可以称图15-8为“基于块的嵌套循环连接”。

15.3.4 嵌套循环连接的分析

例15.4的分析可以重复应用在任何$B(R)$、$B(S)$和M上。假设S是较小的关系，chunk数或外层循环的迭代次数是$B(S)/(M-1)$。每一次迭代时，我们读取S的$M-1$个块和R的$B(R)$个块。这样，磁盘I/O的数量是：

$$\frac{B(S)}{M-1}(M-1+B(R))$$

或

$$B(S)+\frac{B(S)B(R)}{M-1}$$

设想M、$B(S)$和$B(R)$都很大，但M是其中最小的，上面公式的一个近似值是$B(S)B(R)/M$。也就是说，代价与两个关系的大小的乘积再除以可用内存容量得到的商成比例。当两个关系都很大时，我们可以做得好得多，尽管我们应当注意对像例15.4中那样的相当小的实例来说，嵌套循环连接的代价并不比一趟连接的代价即1500次磁盘I/O大多少。实际上，如果$B(S)\leqslant M-1$，嵌套循环连接与15.2.3节中的一趟连接算法是一样的。

尽管嵌套循环连接通常并不是可能的连接算法中最有效的算法，我们应该注意在一些早期的关系DBMS中，它是惟一可用的方法。即使今天，某些情况下在更有效的连接算法中，仍然需要把它作为一个子程序，例如，当每个关系中的大量元组在连接属性上具有相同的值时。关于嵌套循环是必不可少的一个例子，参见15.4.5节。

15.3.5 迄今为止的算法小结

图15-9中给出了15.2节和15.3节中我们已经讨论过的算法的内存和磁盘I/O需求。γ 和 δ 的内存需求实际上比给出的更复杂，并且$M = B$ 仅是一个大致的近似。对于 γ，M 随组的数量增长；而对于δ，M 随不同的元组的数量增长。

操作符	大致需要的M	磁盘I/O	节
σ, π	1	B	15.2.1
γ, δ	B	B	15.2.2
$\cup$, $\cap$, $-$, $\times$, $\bowtie$	$\min(B(R),B(S))$	$B(R)+B(S)$	15.2.3
$\bowtie$	任意 $M\geqslant 2$	$B(R)B(S)/M$	15.3.3

图15-9 一趟算法和嵌套循环算法的内存以及磁盘I/O需求

15.3.6 习题

736

习题15.3.1 给出基于块的嵌套循环连接形式的三个迭代器函数。

* **习题15.3.2** 假设$B(R)=B(S)=10\,000$，并且$M=1000$。计算嵌套循环连接的磁盘I/O代价。

习题15.3.3 对于习题15.3.2中的关系，使用嵌套循环连接算法计算$R\bowtie S$时我们需要什么样的M值，磁盘I/O才不超过：

a)100 000

! b)25 000

! c)15 000

! **习题15.3.4** 如果R和S都是非聚簇的，似乎嵌套循环连接将需要大约$T(R)T(S)/M$次磁盘I/O时间。

a) 怎样做才能明显好于这个代价？

b) 如果R和S中只有一个是非聚簇的，你怎样执行嵌套循环连接？考虑两种情况：较大的关系是非聚簇的和较小的是非聚簇的。

! **习题15.3.5** 如果R或S是空的，图15-7的迭代器将无法正确地工作。重写这些函数,使得即使一个关系为空或两个关系都是空时，它们仍能工作。

15.4 基于排序的两趟算法

现在，我们开始学习在关系上执行关系代数操作的多趟算法，这里的关系大于15.2节的一

趟算法能够处理的关系。我们的重点在两趟算法上，其中来自于操作对象关系中的数据被读到内存，以某种方式处理，并再次写回到磁盘，然后重新读取磁盘以完成操作。我们可以自然地将这种想法扩展到任何趟数，其中数据被多次读取到内存。然而，我们将重点放在两趟算法上，这是因为：

a) 即使对于很大的关系，两趟通常也就足够了。

b) 将两趟算法推广到多趟并不难；我们将在15.8节讨论这些扩展。 737

本节中，我们考虑把排序作为实现关系操作的一种工具。基本思想如下。如果我们有一个较大的关系R，其$B(R)$大于我们可用的内存缓冲区数M，那么，我们可以重复地：

1. 将R的M个块读到内存。

2. 使用一种有效的排序算法，在内存中对这M个块排序。这样的算法将占用的处理器时间只比内存中的元组数的线性函数略大一点，因此我们预计排序的时间不会超过第(1)步所需的磁盘I/O时间。

3. 将排好序的列表写入磁盘的M个块中。我们将有R的一个排序子表来指代这些块的内容。

我们将要讨论的所有算法接下来在第二趟中以通过某种方式“归并”排序子表，以执行所期望的操作。

15.4.1 利用排序消除重复

为了用两趟执行 $\delta(R)$操作，我们按上面的描述在子表中将 R 的元组排序。然后，像在11.4.4节的多路归并排序中所做的那样，我们利用可得到的内存来容纳来自每一个排序子表的一个块。然而，我们并不将来自这些子表的元组排序，而是不断地复制元组到输出，并忽略与它相同的所有元组。过程如图15-10所示。

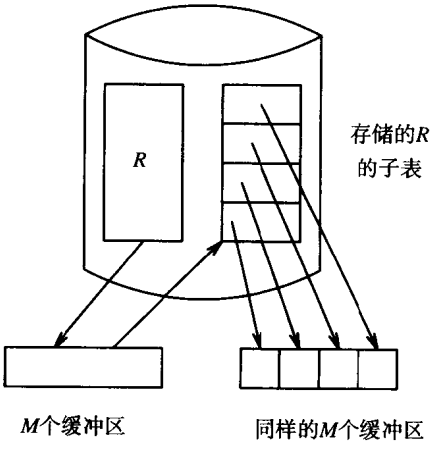


图15-10 消除重复值的一个两趟算法

更准确地讲，我们看来自每一块的第一个未考虑的元组，并且按排序的次序在它们中间找到第一个元组，例如 t。我们在输出中生成 t 的一个副本，而且从不同输入块的前端删除t所有的副本。如果一个块已经空了，我们就在它的缓冲区中装入同一子表的下一个块，而且如果那个块中有t，我们也要将其删除。 738

例15.5 为简明起见，我们假设元组是整数，而且一个块中仅有两个元组。我们还设 $M = 3$；即内存中有三个块。关系R包含17个元组：

2, 5, 2, 1, 2, 2, 4, 5, 4, 3, 4, 2, 1, 5, 2, 1, 3

我们读取前6个元组到内存中的三个块中，对它们排序，并且将它们作为子表 R_1 写出。同样，我们接着读进元组7~12，排序并作为子表 R_2 写出。最后的5个元组同样被排序并成为子表R_3。

第二趟开始时，我们可以把三个子表中的每一个的第一个块放入内存中。现在的情况是：

子表	内存中	磁盘上剩下的
R_1:	1 2	2 2, 2 5
R_2:	2 3	4 4, 4 5
R_3:	1 1	2 3, 5

观察内存中三个块的第一个元组，我们发现1是排序后的第一个元组。因此，我们在输出中产生1的一个副本，并且从内存的块中删除所有的1。当我们这样做后，子表R_3的块就处理完了，所以我们装入该子表中的下一个块，它具有元组2和3。如果在这个块上还有1的副本，我们就将它们去掉。现在的情况是：

子表	内存中	磁盘上剩下的
R_1:	2	2 2, 2 5
R_2:	2 3	4 4, 4 5
R_3:	2 3	5

现在，2是表的前部最小的元组，而且事实上它恰巧出现在每一个表中。我们将2的一个副本写到输出，并从内存块中去除2的副本。子表R_1的块处理完毕，并且该子表中的下一个块被装进内存。此块中有2的副本，于是去除它们，这样就再次处理完了R_1的块。将该子表的第三个块装进内存，并且删除其中的2。现在的情况是：

子表	内存中	磁盘上剩下的
R_1:	5	
R_2:	3	4 4, 4 5
R_3:	3	5

现在，3被选作最小的元组，3的一个副本写到输出，并且R_2和R_3的块都已处理完，并用磁
739 盘上的块替换，状态变为：

子表	内存中	磁盘上剩下的
R_1:	5	
R_2:	4 4	4 5
R_3:	5	

为结束这个例子，4是下一个要选定的，这将消耗掉子表R_2的大部分。在最后一步，每一个表碰巧都包含一个5，将它输出一次并从输入缓冲区中删除。 □

和平常一样忽略对输出的处理，执行这个算法的磁盘I/O数为：

1. $B(R)$用于在创建排序子表时读 R 的每一个块。
2. $B(R)$用于将各排序子表写到磁盘。
3. $B(R)$用于在适当的时候从子表中读每一个块。

因此，相对于15.2.2节一趟算法的代价$B(R)$，这个算法总的代价是$3B(R)$。

另一方面，我们可以用两趟算法处理比一趟算法所能处理的文件大得多的文件。假设内存中有M个块可用，我们创建若干大小为M块的排序子表。在第二趟中，我们需要内存中有每个子表的一个块，所以子表不能超过M个，每一个有M块。因此，与一趟算法的$B \leqslant M$相比，要使两趟算法可行，需要使$B \leqslant M^2$。换言之，用两趟算法计算$\delta(R)$仅需要$\sqrt{B(R)}$个内存块，而不是$B(R)$个内存块。

15.4.2 利用排序进行分组和聚集

$\gamma_L(R)$的两趟算法与15.4.1节中$\delta(R)$的算法非常相似。我们将它概括如下：

1. 将R的元组读到内存中，每一次读M块。用L的分组属性作为排序关键字，对每M块排序。将每一个排好序的子表写到磁盘。
2. 为每一个子表使用一个主存缓冲区，并且首先将每一个子表的第一个块装入其缓冲区。
3. 在缓冲区可以获得的第一个元组中反复查找排序关键字(分组属性）的最小值。这个最

小值v成为下一分组，我们为它：

a) 准备在这个分组的列表L上计算所有的聚集。就像15.2.2节中那样，使用计数与求和来代替求平均。

740

b) 检查每个排序关键字为v的元组，并且累计所需聚集。

c) 如果一个缓冲区空了，则用相同的子表中的下一个块替换它。

当不再有排序关键字为v的元组时，输出一个由L的分组属性和对应的我们已经为这个组计算出的聚集值构成的元组。

正如δ算法那样，γ的这种两趟算法使用$3B(R)$次磁盘I/O，而且只要$B(R) \leqslant M^2$就可以正常工作。

15.4.3 基于排序的并算法

当需要包的并时，15.2.3节中简单地复制两个关系的一趟算法就可以。这一算法的正常工作与操作对象的大小无关，因而我们不必考虑$\cup_B$的两趟算法。然而，只有当至少一个关系小于可用的内存时，$\cup_S$的一趟算法才起作用，因此，我们应该考虑集合并操作的两趟算法。正如我们将要在15.4.4节看到的那样，我们提出的方法对于集合和包的交和差也都适合。为计算$R \cup_S S$，我们做以下工作：

1. 重复地将R的M块装入内存中，对它们的元组排序，并且将产生的排序子表写回磁盘。
2. 为了创建关系S的排序子表，对S做相同的工作。
3. 为R和S的每个子表使用一个内存缓冲区，用对应子表的第一块初始化各缓冲区。
4. 重复地在所有缓冲区中查找剩余的第一个元组t。将t复制到输出，并且从缓冲区中删除t的所有副本(如果R和S都是集合，则至多有两个副本)。如果一个缓冲区变空了，则从它的子表中重新调入下一个块。

我们看到，R和S的每一个元组被两次读进内存，一次是当子表创建时，第二次是作为子表的一部分。元组还写回磁盘一次，作为新建子表的一部分。因此，磁盘I/O的代价是$3(B(R) + B(S))$。

因为对每一个子表，我们需要一个缓冲区，所以只要两个关系的子表总数不超过M，算法就能工作。既然每一个子表长度是M块，那么两个关系的大小不能超过M^2；即$B(R) + B(S) \leqslant M^2$。

741

15.4.4 基于排序的交和差算法

无论是要计算集合形式还是包形式，除了我们在处理排序子表前部的元组t的副本时有区别以外，算法基本上与15.4.3节中的算法相同。通常，我们为作为操作对象的关系R和S各自创建若干大小为M块的排序子表。我们为每一个子表使用一个内存缓冲区，并将每个子表初始化为相应子表的第一个块。

接下来，我们不断考虑所有缓冲区内剩余的元组中最小的元组t。我们统计R的元组中与t相同的元组数，并且也统计S的元组中与t相同的元组数。这需要我们为当前缓冲块已处理完的子表重新装载缓冲区。下面说明我们怎样判定t是否输出，以及如果输出时，应该输出多少次：

- 如果操作是集合交，并且如果t在R和S中都出现，就输出t。
- 如果操作是包交，则输出t的次数是它在R和S中出现的最小次数。注意，如果两个计数中有一个为0，就不输出t；也就是说，当t在一个或两个关系中未出现时就不输出它。
- 如果操作是集合差，$R -_S S$，当且仅当t出现在R中但不在S中时输出t。
- 如果操作是包差，$R -_B S$，输出t的次数是t在R中出现的次数减去在S中出现的次数。当然，

如果 t 在S中出现的次数至少等于在R中的出现次数，那么根本就不要输出t。

例15.6 让我们做和例15.5同样的假设：$M=3$，元组是整数，并且一个块能容纳两个元组。数据也几乎和那个例子中的相同。然而，这里需要两个操作对象，所以我们将假设R有12个元组，S有5个元组。因为内存可以容纳6个元组，第一趟中得到R的两个子表，我们称之为R_1和R_2，而S仅有一个排序子表，我们称之为S_1[⊖]。(在类似于例15.5中数据的未排序关系上）创建子表之后，状态为：

子表	内存中	磁盘上剩下的
R_1:	1 2	2 2, 2 5
R_2:	2 3	4 4, 4 5
S_1:	1 1	2 3, 5

假设我们想计算包的差 $R -_B S$。我们发现在内存缓冲区中最小的元组是1，所以在R的子表
742 和S的子表中统计1的数目。我们发现1在R中出现1次，在S中出现两次。因为1在R中出现的次数不比在S中的多，我们不输出元组1的任何副本。因为计算1的数目后S_1的第一个块就空了，我们装入S_1的下一个块，状态变为：

子表	内存中	磁盘上剩下的
R_1:	2	2 2, 2 5
R_2:	2 3	4 4, 4 5
S_1:	2 3	5

现在我们发现2是剩下的元组中最小的，所以我们统计它在R中出现的次数，为5次，并且我们统计它在S中出现的次数，为1次。因此我们将2输出4次。当我们执行统计时，必须重新装载R_1的缓冲区两次，这使状态变为：

子表	内存中	磁盘上剩下的
R_1:	5	
R_2:	3	44, 45
S_1:	3	5

接着，我们考虑元组3，而且发现它在R中出现1次并在S中出现1次。因此，我们不输出3，并且从缓冲区中删除它的副本，状态变为：

子表	内存中	磁盘上剩下的
R_1:	5	
R_2:	44	45
S_1:	5	

元组4在 R 中出现3次并且在S中根本没有出现，所以我们输出4的三个副本。最后，5在 R 中出现两次并且在 S 中出现1次，所以我们将5输出1次。完整的输出为：2，2，2，2，4，4，4，5。 □

这一类算法的分析和15.4.3节中对集合并算法所做分析相同：

- $3(B(R)+B(S))$ 次磁盘I/O。
- 为使算法能工作，近似地要求$B(R)+B(S) \leqslant M^2$。

15.4.5 基于排序的一个简单的连接算法

将排序用于连接大的关系的方法有多种。在讨论连接算法之前，我们来看一个在计算连接

⊖ 由于S能被主存所容纳，我们实际上可以使用15.2.3节中的一趟算法，但是为了说明问题我们将使用两趟方式。

时可能出现的问题，但这个问题不是迄今为止考虑过的二元操作的问题。在计算连接时，两个关系在连接属性上具有相同的值因而需要同时放入内存中的元组，可能超过内存所能容纳的数量。极端的例子是当连接属性仅有一个值时，这时一个关系中的每个元组与另一关系的每个元 743
组都能连接。这种情况下，除了对在连接属性上值相等的两个元组集合进行嵌套循环连接外，就真的没有其他选择了。

为了避免面对这种情形，我们可以尽量减少为算法中其他方面使用的内存，因而可以用大量缓冲区保存具有给定连接属性值的元组。本节中我们将要讨论一个算法，它可以为具有共同值的元组连接获取最大量的可用缓冲区。在15.4.7节中，我们考虑另一个使用较少的磁盘I/O并基于排序的算法，但该算法在大量的元组在连接属性上具有共同的值时可能会出现问题。

已知将要连接的关系$R(X,Y)$和$S(Y,Z)$，并且已知有M块内存用作缓冲区，我们做下面的事情：

1. 用Y作为排序关键字，使用两阶段、多路归并排序对R进行排序。

2. 类似地对S做排序。

3. 归并排好序的R和S。我们通常仅用两个缓冲区，一个给R的当前块，另一个给S的当前块。重复执行以下步骤：

(a) 在当前R和S的块的前端查找连接属性Y的最小值y。

(b) 如果y在另一个关系的前部没有出现，那么删除具有排序关键字y的元组。

(c) 否则，找出两个关系中具有排序关键字y的所有元组。如果需要，从排序的R和/或S中读取块，直到我们确定每一个关系中都不再有y的副本。最多可以用M个缓冲区来做这件事情。

(d) 输出通过连接R和S中具有共同的Y值y的元组所能形成的所有元组。

(e) 如果一个关系在内存中已没有未考虑的元组，就重新装载为那个关系而设的缓冲区。

例15.7 让我们考虑例15.4的关系R和S。回想一下，这两个关系分别占用1000个块和500个块，并且有M=101个内存缓冲区。当我们在一个关系上使用两阶段归并排序时，对每一个块我们使用4次磁盘I/O，每个阶段有两次。那么，对R和S排序我们要使用$4(B(R)+B(S))$次磁盘I/O，即6000次磁盘I/O。

当归并R和S以得到连接的元组时，我们用另外的1500次磁盘I/O来第五次读取R和S每一个块。这个归并中，我们通常仅需要101个内存块中的两个。然而，如果需要，我们可以使用所有101个块来容纳具有公共的Y值y的R和S的元组。因此，只要对于任意的y，R和S中Y值为y的元组占用的空间不超过101块，这就足够了。 744

注意，这个算法执行的磁盘I/O总数量是7500，而例15.4中嵌套循环连接的磁盘I/O数量是5500。然而，嵌套循环连接是一个固有的二次算法，占用的时间与$B(R)B(S)$成比例，而排序连接具有线性的I/O代价，占用的时间与$B(R)+B(S)$成比例。只有常数因子以及较小的示例关系(每一个关系只不过比一个能完全装入分配的缓冲区中的关系大5或10倍)，嵌套循环连接才更可取。此外，在15.4.7节中我们将看到，在$3(B(R)+B(S))$次磁盘I/O内执行一个排序连接通常是可能的，在这个例子中将是4500，它低于嵌套循环连接的代价。 □

如果具有某个Y值y的元组数量不能被M个缓冲所容纳，那么我们需要修改上面的算法。

1. 如果有一个关系(比方说R)中具有Y值y的元组能完全装入$M-1$个缓冲区中，那么将R的这些块装入到缓冲区，并且一次一个地将S的块中其值是y的元组读到剩余的缓冲区中。实际上，我们是只在具有Y值y的元组上做15.2.3节的一趟连接。

2. 如果两个关系中具有Y值y的元组都不够小，都不能完全装入$M-1$个缓冲区中，那么，使用M个缓冲区，在两个关系中具有Y值y的元组上执行嵌套循环连接。

注意，每一种情况都可能需要从一个关系读取块，然后又忽略它们，后来又需要读这些块。例如，在情况1中，我们可能首先读取S的块中具有Y值y的元组，并且发现这样的元组太多了以至于不能全部装入$M-1$个缓冲区。然而，如果我们接下来读取R中具有Y值y的元组，我们发现它们确实能装入$M-1$个缓冲区中。

15.4.6 简单排序连接的分析

就像我们在例15.7中注意到的那样，对于操作对象的每一个块，我们的算法执行5次磁盘I/O。如果具有一个公共的Y值y的元组非常多，在这些元组上我们需要做一种特殊的连接，这是一种例外情况。在这种情况下，额外的磁盘I/O数量依赖于是一个关系中还是两个关系中具有公共的Y值y的元组太多，因而单是这样的元组需要的缓冲区数就超过$M-1$。这里，我们将不再深入地研究所有的细节；习题中包括了一些例子，需要你去解决。

我们还需要考虑为了使简单排序连接能够运行，M需要多大。主要的限制在于我们必须能够在R和S上执行两阶段、多路归并排序。就像我们在11.4.4节看到的那样，为执行这些排序，我们需要$B(R)\leqslant M^2$和$B(S)\leqslant M^2$。一旦排序完成，我们将不再会有缓冲区不够的问题。尽管像前面讨论的那样，如果具有一个公共的Y值y的所有元组不能全部装入M个缓冲区中，我们可能不
745 得不偏离简单归并。总之，假设不必要发生这样的偏离是不必要的：

- 简单排序连接使用$5(B(R)+B(S))$次磁盘I/O。
- 为了能工作，它要求$B(R)\leqslant M^2$且$B(S)\leqslant M^2$。

15.4.7 一种更有效的基于排序的连接

如果不必担心具有连接属性公共值的元组太多，那么我们可以将排序的第二阶段和连接本身合并，这样对每个块而言可以节约两次磁盘I/O。我们称这个算法为*排序连接*；还有一些其他的名字如“归并连接”和“排序归并连接”也指这个算法。为了用M个缓冲区计算$R(X,Y)\bowtie S(Y,Z)$，我们：

1. 用Y作为排序关键字，为R和S创建大小为M的排序子表。
2. 将每一个子表的第一块调进缓冲区；我们假设总共不超过M个子表。
3. 重复地在所有子表的第一个可以得到的元组中查找最小的Y值y。识别两个关系中具有Y值y的所有元组。如果子表数少于M，可能使用M个缓冲区中的一部分来容纳这些元组。输出R和S中具有此公共Y值的所有元组的连接。如果一个子表的缓冲区处理完毕，则重新将磁盘上的块装入其中。

例15.8 让我们再次考虑例15.4中的问题：使用101个缓冲区连接关系R和S，它们分别有1000块和500块。我们将R分成10个子表，将S分成5个子表，每一个子表长度为100，并且对它们排序[⊖]。然后我们用15个缓冲区来容纳各子表的当前块。如果我们面临着许多元组都有某个固定Y值的情况，可以用剩下的86个缓冲区来存储这些元组；但是，如果元组比这还多，则我们必须使用一种如15.4.5节最后所讨论的那样的特殊算法。

假设不需要为具有相同Y值的大量元组的集合而修改算法，则我们为数据的每个块执行三次磁盘I/O。其中两次是为了创建排序的子表。然后，在多路归并过程中，每一个排序子表的

⊖ 从技术上来说，我们可以使每个子表的长度为101块，并且R的最后一个子表有91块，S的最后一个子表有96块，但代价完全相同。

每一块被再次读取到内存中。因此，总的磁盘I/O数是4500。 □

这个排序连接算法在能够使用时比15.4.5节中的算法更有效 。就像我们在例15.8中所看到的那样，磁盘I/O的数目是3(*B*(*R*)+*B*(*S*))。我们可以在几乎和前面算法中一样多的数据上执行这个算法。排序子表的长度是*M*块，而且两个关系总的子表数至多是*M*。因此，*B*(*R*)+*B*(*S*)≤*M*²就 746
足够了。

我们可能想知道，当有很多元组具有公共的*Y*值时，我们是否可以避免因此产生的问题。一些重要的考虑是：

1. 有时，我们可以确定问题不会发生。例如，如果*Y*是*R*的一个关键字，那么，一个给定的*Y*值y在*R*的子表的所有块中仅出现一次。当轮到y时，我们可以让*R*的相应元组保持不动，并且将它和*S*中所有匹配的元组连接。如果这一过程中*S*子表的块处理完毕，无论*S*中有多少元组有*Y*值 *y*，这些子表的缓冲区都可以再调入下一个块，而不需要额外的空间。当然，如果*Y*是*S*的一个关键字而不是*R*的，上述过程也同样适用，只不过将*R*和*S*交换一下。

2. 如果*B*(*R*)+*B*(*S*)远远小于*M*²，就像在例15.8中指出的那样，我们将有很多未用的缓冲区，可以用来存储具有某个公共的*Y*值的元组。

3. 如果其他的所有情况都不满足，我们可以只在具有某个公共*Y*值的元组上使用嵌套循环连接，这样做将使用额外的磁盘I/O，但能使工作顺利完成。这一可选方案将在15.4.5节讨论。

15.4.8 基于排序的算法小结

图15-11中的表格是对15.4节中我们讨论过的算法的分析。就像在15.4.5节和15.4.7节讨论过的那样，如果我们连接的两个关系有许多元组在连接属性上有相同的值，那么有必要对时间和内存需求进行修改。

操作符	大致需要的*M*	磁盘I/O	节
γ, δ	$\sqrt{B}$	$3B$	15.4.1, 15.4.2
$\cup, \cap, -$	$\sqrt{B(R)+B(S)}$	$3(B(R)+B(S))$	15.4.3, 15.4.4
$\bowtie$	$\sqrt{\max(B(R),B(S))}$	$5(B(R)+B(S))$	15.4.5
$\bowtie$	$\sqrt{B(R)+B(S)}$	$3(B(R)+B(S))$	15.4.7

图15-11 基于排序算法的内存和磁盘I/O需求 747

15.4.9 习题

习题15.4.1 使用例15.5中的假设(每个块两个元组，等等)，

a) 将序列0, 1, 2, 3, 4重复六遍，形成一个具有30个单属性元组的序列，说明在这个序列上执行消除重复的两趟算法的行为。

b) 说明计算关系γ_a，*AVG*(*b*) (*R*) 的两趟分组算法的行为。关系*R*(*a*,*b*)包括t_0~t_{29}这30个元组，元组t_i以*i*模5的结果作为它的分组成分*a*，并以*i*作为它的第二个成分*b*。

习题15.4.2 使用本节中描述的算法，为下面的每个操作编写迭代器。

* a) 消除重复(δ)。

b) 分组(γ_L)。

* c) 集合交。

d) 包差。

e) 自然连接。

习题15.4.3 如果$B(R)=B(S)=10\,000$，并且$M=1000$，以下情况中磁盘I/O的需求是多少：

a) 集合并。

* b) 简单的排序连接。

c) 15.4.7节中更有效的排序连接。

! **习题15.4.4** 假设本节中所描述算法的第二趟不需要所有的M个缓冲区，因为子表数小于M。我们怎样通过使用额外的缓冲区来节省磁盘I/O?

! **习题15.4.5** 例15.7中，我们讨论了两个关系R和S的连接，它们分别有1000块和500块，并且$M=101$。但是我们指出，如果具有给定值的元组太多，以至于任何一个关系的元组都不能全部装入内存，那么就会有额外的磁盘I/O。计算需要的磁盘I/O总数，如果：

* a) 仅有两个Y值，每一个出现在R的元组的一半和S的元组的一半中(回忆一下Y是一个或多个连接属性)。

748 b) 有5个Y值，每个Y值在每一个关系中出现的可能性相等。

c) 有10个Y值，在每一个关系中每一个很可能相等。

! **习题15.4.6** 用15.4.7节中更有效的排序连接重复习题15.4.5。

习题15.4.7 如果每个关系有10 000个块，并且使用基于排序的两趟算法，对于以下运算，我们各需要多少内存?

* a) δ。

b) γ。

c) 一个二元操作，比如连接或并。

习题15.4.8 对于习题15.2.4中5个类连接操作符中的每一个，描述一种基于排序的两趟算法。

! **习题15.4.9** 假设记录可以大于块，也就是说，我们可以有跨块记录。基于排序算法的内存需求将怎样变化?

!! **习题15.4.10** 有时，如果我们将最后一个子表放在内存中，可能会节省部分磁盘I/O。为利用这个效果而应使用少于M块的子表，这甚至也是有意义的。这种方法能节省多少磁盘I/O?

!! **习题15.4.11** OQL允许依据对象的任意的用户定义函数来对对象分组。例如，我们可以依据两个属性的和对元组分组。在一个对象集上，我们怎样执行这种类型的一个基于排序的分组操作?

15.5 基于散列的两趟算法

有一个基于散列的算法系列处理与15.4节中相同的问题。所有这些算法的基本思想如下。如果数据量太大以至于不能存入内存缓冲区中，就使用一个合适的散列关键字散列一个或多个操作对象的所有元组。对于所有通常的操作，有一种选择散列关键字的方法，它能使在我们执行该操作时需要一起考虑的所有元组具有相同的散列值。

然后，我们通过一次处理一个桶(或者在二元操作运算的情况下，通过一次处理具有相同散列值的一对桶) 的方式执行操作。实际上，我们已经减小了操作对象的大小，减小的比例等于桶的数目。如果有M个可用缓冲区，我们可以将M作为桶的数目，这样我们所能处理的关系大小就增大了M倍。注意，15.4节中基于排序的算法通过预处理也得到了一个因子M，尽管排

749 序和散列这两种方法达到这一相似比例的方法各不相同。

15.5.1 通过散列划分关系

首先我们回顾一下接受关系R并使用M个缓冲区将R划分成大小大致相等的$M-1$个桶的方式。我们将假设h是散列函数，并且h将R的整个元组作为参数(也就是说，R的所有属性都是散列关键字的一部分)。我们将每一个桶和一个缓冲区联系起来。最后一个缓冲区用来每次一块地装入R的块。块中的每个元组t被散列到桶$h(t)$并且被复制到适当的缓冲区中。如果缓冲区满了，我们就将它写到磁盘，并且为同一个桶初始化另一个块。最后，对于每个桶的最后一块，如果它不空的话，我们就把它写到磁盘。图15-12更详细地描述了这一算法。注意，虽然元组的长度可能是变化的，但是该算法假设元组不会大到不能装入一个空的缓冲区中。

```
initialize M-1 buckets using M-1 empty buffers;
FOR each block b of relation R DO BEGIN
    read block b into the Mth buffer;
    FOR each tuple t in b DO BEGIN
        IF the buffer for bucket h(t) has no room for t THEN
            BEGIN
                copy the buffer to disk;
                initialize a new empty block in that buffer;
            END;
        copy t to the buffer for bucket h(t);
    END;
END;
FOR each bucket DO
    IF the buffer for this bucket is not empty THEN
        write the buffer to disk;
```

图15-12 将一个关系划分到$M-1$个桶中

15.5.2 基于散列的消除重复算法

现在，对于各种可能需要两趟算法的关系代数操作，我们来考虑基于散列的算法的细节。首先考虑重复的消除，即操作$\delta(R)$。按照图15-12，我们将R散列到$M-1$个桶。注意，相同元组t的两个副本将散列到同一个桶中。因此，δ具有我们所需要的基本特性：我们可以一次检查一个桶，在该桶中独立地执行δ，并且把$\delta(R_i)$的并作为结果，其中R_i是R中散列到第i个桶的那一部分。15.2.2节中的一趟算法可以用来依次消除每个R_i中的重复，并将产生的惟一元组写回磁盘。 750

只要每一个R_i小到能装入内存因而允许使用一趟算法，这个方法就可行。因为我们假设散列函数h将R划分到大小相同的桶中，那么每一个R_i的近似大小为$B(R)/(M-1)$个块。如果这一块数小于等于M，即$B(R)\leqslant M(M-1)$，那么基于散列的两趟算法就可行。事实上，就像我们在15.2.2节讨论的那样，只要每个桶中不同元组的数量能被M个缓冲区容纳就可以，但是我们根本不能确定是否有重复。所以，一个保守的估计是$B(R)\leqslant M^2$，这是在认为M和$M-1$相同时得到的简单形式，和δ的基于排序的两趟算法一样。

磁盘I/O的数量也与基于排序的算法相似。当我们散列元组时，读取R的每个块一次，并且将每个桶的每个块写到磁盘上。然后在针对该桶的一趟算法中，我们再次读取每个桶的每一个块。因此，磁盘I/O的总数量是$3B(R)$。

15.5.3 基于散列的分组和聚集算法

为了执行$\gamma_L(R)$操作，我们也是首先将R的所有元组散列到$M-1$个桶中。然而，为了确保同一组的所有元组最终都在同一个桶内，我们选择的散列函数依赖于表L中的分组属性。

将R分到桶中以后，接下来，我们可以用15.2.2节中γ的一趟算法依次处理每一个桶。和我们在15.5.2节对δ的讨论一样，只要$B(R)\leqslant M^2$，我们就可以在内存中处理每一个桶。

然而，在第二趟中，我们在处理每一个桶时只需要每个分组中的一个记录。因此，即使桶的大小大于M，只要桶内各个分组的记录使用的缓冲区数不超过M，我们就可以在一趟中处理该桶。一般地说，一个分组的记录不会比R的一个元组大。如果是这样，那么$B(R)$的一个更好的上界是每个分组中平均元组数量的M^2倍。

因此，如果分组很少，那么我们实际上能处理的关系R可能比$B(R) \leqslant M^2$规则所指出的更大。另一方面，如果M超过了分组的数量，那么，我们不能填充所有的桶。所以，作为M的一个函数，R的大小的实际限制很复杂，但$B(R) \leqslant M^2$是一个保守的估计。最后，和δ一样，我们发现γ的磁盘I/O数是$3B(R)$。

15.5.4 基于散列的并、交、差算法

当操作是二元的时，我们必须保证使用相同的散列函数来散列两个操作对象的元组。例如，
751 为了计算$R \cup_S S$，我们将R各自散列到$M-1$个桶，例如$R_1, R_2, \cdots, R_{M-1}$和$S_1, S_2, \cdots, S_{M-1}$。然后，对于所有的$i$，我们计算$R_i$和$S_i$的集合并，并且输出结果。注意，如果一个元组$t$在$R$和$S$中都出现，那么对于某$i$，我们在$R_i$和$S_i$中都将发现$t$。这样，当我们计算这两个桶的并时，将仅输出$t$的一个副本，不可能将重复引入结果中。对于$\cup_B$而言，15.2.3节中简单的包-并算法胜于执行这一操作的任何其他方法。

为了计算R和S的交或差，我们像计算集合并时一样创建$2(M-1)$个桶，并且在每对对应的桶上运用适当的一趟算法。注意，所有这些算法需要$B(R)+B(S)$次磁盘I/O。在这个数目上，我们还必须加上每个块的两次磁盘I/O，这是用来散列两个关系中的元组，并将桶存储到磁盘上，一共是$3(B(R)+B(S))$次磁盘I/O。

为了使算法可行，我们必须能一趟计算R_i和S_i的并、交或差，R_i和S_i的大小分别约为$B(R)/M-1$和$B(S)/M-1$。回忆一下，这些操作的一趟算法要求较小的操作对象至多占用$M-1$个块。因此，基于散列的两趟算法近似地要求$\min(B(R), B(S)) \leqslant M^2$。

15.5.5 散列连接算法

为了使用基于散列的两趟算法计算$R(X,Y) \bowtie S(Y,Z)$，我们所要做的与15.5.4节中讨论的其他二元操作几乎一样。惟一的区别是我们必须用连接属性Y作散列关键字。这样我们就能确定，如果R和S的元组能连接，那么它们必然出现在具有某个i值的相应桶R_i和S_i中。所有对应桶对的一个一趟连接最后完成这个我们称为*散列连接*[⊖]的算法。

例15.9 让我们再次讨论例15.4中的两个关系R和S，它们的大小分别为1000块和500块，并且有101个内存缓冲区是可用的。我们可以将每一个关系散列到100个桶中，所以一个桶的平均大小对于R是10个块，对于S是5个块。因为较小的数5远远小于可得到的缓冲区的数量，我们预计在每一个桶对上，执行一趟连接不会有困难。

当散列到桶中时，读取R和S共需要1500次磁盘I/O，将所有的桶写到磁盘又需要1500次，执行对应桶的一趟连接时再次将每一个桶对读到内存需要第三个1500。因此，需要的磁盘I/O数是4500，和15.4.7节中有效的排序连接一样。 □

752 我们可以对例15.9进行推广而得到如下结论：

- 散列连接需要$3(B(R)+B(S))$次磁盘I/O来完成它的任务。
- 只要近似地有$\min(B(R),B(S)) \leqslant M^2$，两趟散列连接算法就是可行的。

⊖ 有时候，术语“散列-连接”专门指15.2.3节中一趟算法的一种变体，其中用散列表作为内存中的查找结构。这时，这里描述的两趟散列-连接算法就称做“划分散列-连接”。

后一点的操作对象与其他二元操作相同：每一个桶对中必须有一个能全部装入$M-1$个缓冲区中。

15.5.6 节省一些磁盘I/O

在第一趟时，如果可用内存比容纳每一个桶的一个块所需内存更多，那么，我们有可能节省磁盘I/O。一个选择是为每一个桶使用若干块，并且将它们整体写到磁盘中连续的块上。严格地说，这项技术不能节省磁盘I/O，但是它使得I/O执行得快一些，因为我们写磁盘时能减少寻道时间和旋转延迟。

然而，有一些技巧曾被用来避免将一些桶中内容写到磁盘并且接着再次读它们。其中最有效的技巧，称为*混合散列连接*，它的工作方式如下。一般地讲，假设我们确定连接$R \bowtie S$且S是较小的关系，我们需要建立k个桶，这里的k远远小于可用的内存M。当我们散列S时，可以选择将k个桶中的m个完全保留在内存中，而对于其他$k-m$个桶中的每一个，则仅保留一个块。只要内存中的桶的预期大小加上其他每个桶的一个块不超过M；即：

$$\frac{mB(S)}{k} + k - m \leqslant M \tag{15-1}$$

我们就能努力做到这一点。解释一下，一个桶的预期大小是$B(S)/k$，并且有m个桶在内存中。

现在，当读取另一个关系R的元组，以将这个关系散列到桶中时，我们在内存中保留：

1. S的m个从未写到磁盘的桶，以及
2. R的$k-m$个桶中每一个的一块，这$k-m$个桶对应的S桶被写到磁盘上。

如果R的一个元组t散列到最开始的m个桶的其中一个，那么我们立刻将它和相应的S桶的元组连接，就好像正在做一趟散列—连接那样。任何成功连接的结果都被立刻输出。和一趟散列-连接一样，为了加速这个连接，S在内存中的各个桶有必要组织成某种有效的查找结构。如果t散列到某个桶中，该桶相对应的S桶位于磁盘上，那么和基于散列的两趟连接一样，t被送到该桶在内存中的块中，而且最终将移到磁盘上。 753

在第二趟中，我们像通常一样连接R和S的相应的桶。然而，对留在内存中的S桶，没有必要连接相应的桶对，因为这些桶已经被连接而且结果也已输出。

对于留在内存的S的桶和相应的R桶中的每一个块，节省的磁盘I/O的数目等于2。因为在内存中的桶的比率为m/k，所以I/O节省为$2(m/k)(B(R)+B(S))$。因此，我们肯定会问，在满足等式(15-1)的约束前提下，如何最大限度地增加m/k。尽管这个问题的答案可以形式化地得到，但凭直觉可以得到一个令人惊奇但又正确的答案：m=1，而k尽可能小。

论证如下。内存缓冲区中除了$k-m$个以外的所有缓冲区都可以用来容纳S在内存中的元组，并且这样的元组越多，磁盘I/O的数量越少。因此，我们想使k，即桶的总数量最小。我们通过使每个桶在能被内存容纳的前提下尽可能大来做到这一点；也就是说，桶的大小为M，因此$k=B(S)/M$。如果是这样，那么，额外的内存中只有存放一个桶的空间，即 $m = 1$。

事实上，我们真正需要使桶稍小于 $B(S) / M$，否则，我们就没有足够的空间来同时容纳一个满的桶和其他$k-1$个桶各自的一块。为简化起见，假设 k 大约是 $B(S)/M$，$m = 1$，节省的磁盘I/O是：

$$\left(\frac{2M}{B(S)}\right)\Big(B(R) + B(S)\Big)$$

而且总的代价为：

$$\left(3 - \frac{2M}{B(S)}\right)\Big(B(R) + B(S)\Big)$$

例15.10 考虑例15.4的问题，在那里我们使用M=101连接关系R和S，其大小分别为1000块和500块。如果使用混合散列-连接，那么我们希望桶的数量k大约为500/101。假设我们选定k=5，那么，平均每个桶将有S的元组的100个块。如果试图在内存中容纳这些桶中的一个以及对应于其他4个桶的4个额外的块，那么需要104个内存块，而我们不能冒内存中的桶溢出内存这个险。

因此，我们建议选择k=6。现在，当在第一趟中散列S时，我们有对应于5个桶的5个缓冲区，而且，对于内存中的桶，我们已经高达有96个缓冲区，桶的预期大小是500/6即83。对于S，在第一趟中我们用来读取S的所有内容所使用的磁盘I/O的数目是500，并且500 - 83=417个I/O用来将5个桶写到磁盘。当我们在第一趟中处理R时，需要读取R的所有内容(1000次磁盘I/O)，并将它的6个桶中的5个写到磁盘上(833个磁盘I/O)。

在第二趟中，我们读取写到磁盘上的所有的桶，即再进行417+833=1250个磁盘I/O。总的磁盘I/O的数量就是1500个用于读取R和S，1250个用于将这些关系中的5/6写出，再有1250个用于再次读取那些元组，或者说总数是4000个磁盘I/O。这个数字可与直接的散列连接或排序连
754 接所需的4500个磁盘I/O比较。 □

15.5.7 基于散列的算法小结

图15-13给出了本节中讨论的每一个算法的内存需求和磁盘I/O的数量。就像其他类型的算法一样，我们应该注意到对于 γ 和 δ 的估计可能是保守的，因为它们实际上分别决定于副本和组的数量，而不是操作对象关系的元组的数量。

操作符	大致需要的M	磁盘I/O	节
γ, δ	$\sqrt{B}$	$3B$	15.5.2, 15.5.3
$\cup, \cap, -$	$\sqrt{B(S)}$	$3(B(R)+B(S))$	15.5.4
$\bowtie$	$\sqrt{B(S)}$	$3(B(R)+B(S))$	15.5.5
$\bowtie$	$\sqrt{B(S)}$	$(3-2M/B(S))(B(R)+B(S))$	15.5.6

图15-13 基于散列算法的内存和磁盘I/O需求；对于二元操作，假设$B(S) \leq B(R)$

请注意基于排序的算法和相应的基于散列的算法的需求几乎是相同的。两种方法明显的区别是：

1. 二元操作的基于散列的算法有一个大小的需求，它仅依赖于两个操作对象中较小的一个，而不是在基于排序的算法中的两个操作对象的和。

2. 基于排序的算法有时允许我们产生一个有序序列的结果，而且以后利用那个排序序列。结果可能在以后的另一个基于排序的算法中使用，或它可以作为一个需要以有序序列输出的查询的回答。

3. 基于散列的算法依赖于相等大小的桶。由于通常在大小上至少有小的差异，因此就不可能使用平均占用M块的桶；我们必须将它们限制在一个较小的数字。如果不同散列关键字的数量小的时候，这一现象尤其显著，例如，在一个关系上执行一个具有少数组的group-by或者执行一个在连接属性上有很少元组的连接。

4. 在基于排序的算法中，如果我们适当地组织磁盘，排序子表可能被写到磁盘上连续的块
755 中。因此，每个块的三个磁盘I/O中的一个可能需要较短的旋转延迟或寻道时间，所以可能比基于散列的算法中需要的I/O快得多。

5. 此外，如果M比排序子表的数量大得多，那么，我们可以从一个排序的子表一次读一些连续的块，再一次节省了一些延迟和寻道时间。

6. 另一个方面，在一个基于散列的算法中，如果能够选择桶的数量小于M，那么，我们可以一次写出一个桶的若干个块。因而在散列的写这一步上，可以得到与我们在(5)中看到的基于排序算法的第二次读相同的利益。类似地，我们可以组织磁盘使得一个桶处于磁道连续的块上。如果是这样，就如(4)中所看到的排序的子表被有效地写出一样，桶可以用较短的延迟或寻道时间来读取。

15.5.8 习题

习题15.5.1 在内存中存储一个桶的混合散列连接的思想也可以用于其他的操作。当执行下列的一个两趟的基于散列的算法时，说明怎样节省从每一个关系中存储和读取一个桶的代价？

* a) δ

b) γ

c) $\cap_B$

d) $-_S$

习题15.5.2 如果$B(S)=B(R)=10{,}000$，$M=1000$，则对于一个混合散列连接需要多少磁盘I/O？

习题15.5.3 对于(a)δ (b) γ (c) $\cap_B$ (d) $-S$ $(e)\bowtie$，写出实现它们的两趟的基于散列的算法的迭代器。

*! **习题15.5.4** 设想我们正在一个大小合适的关系R上执行一个两趟的、基于散列的分组操作。也就是说，$B(R)\leqslant M^2$。然而，组太少，并且一些组大于M；也就是说，它们不能一次装入内存中。对这里给定的算法，需要做怎样的修改？

! **习题15.5.5** 设想我们正使用一个磁盘，它将磁头移动到一个块的时间是100毫秒，而且它用1/2毫秒读取一个块。所以，一旦磁头定位，它用k/2毫秒去读取k个连续的块。假设我们想计算一个两趟的散列连接$R\bowtie S$，这里$B(R)=1000$，$B(S)=500$，而且$M=101$。为了加快连接，我们想使用尽量少的桶(假设元组最后都分配到桶中)，而且，将尽量多的块读和写到磁盘连续的位置上。一次随机的磁盘I/O计为100.5毫秒，从磁盘读或写到磁盘k个连续的块计为要100+k/2毫秒：

a) 磁盘I/O占用多少时间？

b) 如果我们使用例15.10中描述的混合散列连接，磁盘I/O占用多少时间？

c) 在相同的条件下，一个基于排序的连接将占用多少时间(假设我们将排序子表写到磁盘连续的块中)？

756

15.6 基于索引的算法

索引存在一个关系的一个或多个属性上使得一些没有索引就不可行的算法可行了。基于索引的算法对于选择操作尤其有用，但是，连接和其他二元操作符的算法也使用索引来获得好处。本节中，我们将介绍这些算法。我们也继续对15.1.1节中开始的对于访问一个带索引的表时索引扫描操作的讨论。为了理解问题，我们首先需要离开主题去考虑“聚簇”索引。

15.6.1 聚簇和非聚簇索引

回忆15.1.3节，如果一个关系的元组紧缩到能存储这些元组的尽可能少的块中，那么这个

关系就是“聚簇”的。我们迄今为止所做的所有的分析都假设关系是聚簇的。

我们也可能谈到在一个或多个属性上的聚簇索引，具有这个索引查询关键字的一个固定值的所有元组都出现在能容纳它们的尽可能少的块中。请注意一个非聚簇的关系不能够有一个聚簇索引[⊖]，但是一个聚簇的关系可以有非聚簇索引。

例15.11 一个关系$R(a, b)$按属性a排序并且按此序列存储，再装入到块中，肯定是聚簇的。因为对于一个给定的a值a_1，所有的具有那个a值的元组是连续存放的，所以在a上的索引是一个聚簇索引。因而除了如图15-14所示的包含a值a_1的第一块和最后一块以外，它们都出现在组装好的块中。然而，在b上的一个索引未必是聚簇的，因为有一个固定的b值的元组将分布
757 到文件中，除非a和b的值有很紧密的相互关系。 □

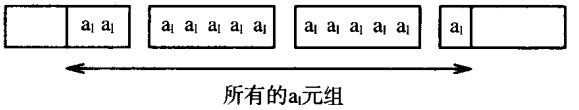


图15-14 一个聚簇索引中具有一个固定值的所有元组组装到最小可能数量的块中

15.6.2 基于索引的选择

15.1.1节中，我们讨论了通过读取关系R的所有元组执行一个选择$\sigma_C(R)$，看一下哪些元组满足条件C，而且输出它们。如果R上没有索引，那么，那就是我们所能做的最好的；或者，如果R不是一个聚簇的关系，操作所用的磁盘I/O的数目是$B(R)$，甚至是$T(R)$，即R中元组的数量[⊖]。然而，设想条件C是$a=v$的形式，这里的a是一个存在着索引的属性，v是一个值。于是就可以用v值来查找索引并且得到恰好指向R中具有a值v的那些元组。这些元组组成了$\sigma_C(R)$的结果，于是我们所需要的就只是取回它们。

如果$R.a$上的索引是聚簇的，那么取回集合$\sigma_{a=v}(R)$所需的磁盘I/O的数目将大约是$B(R)/V(R,a)$。实际的数目可能会高一些，因为：

1. 通常，索引并不是完全保存在内存中，因此需要一些磁盘I/O支持索引的查找。

2. 即使$a=v$的所有元组都可以装入b个块中，它们也可能分布到$b+1$个块中，因为它们不会在一个块的起始处开始。

3. 尽管索引是聚簇的，但$a=v$的元组可能会分布到几个额外的块中。为什么这种情况会发生，有两个原因：

聚簇的概念

我们已经看到了三个尽管相关但又不同的概念称为“聚簇”：

1. 在13.2.2节中，我们谈到“聚簇文件组织”，这里一个关系R的若干个元组与另外某个关系S的一个元组放在一起，它们共享一个公共值；例子就是若干个电影元组与拍摄这些电影的工作室的元组分为一个组。

2. 在15.1.3节中，我们谈到了一个“聚簇关系”，意思是关系元组存储在全部，或者至少是主要用来存储这个关系的块中。

⊖ 技术上讲，如果索引建在关系的一个关键字上，因此给定一个索引关键字值，只存在一个具有给定值的元组，那么即便这个关系不是聚簇的，这个索引也总是“聚簇”的。但是，如果每个索引关键字值都只有一个元组，那么聚簇并没有益处，并且对这样的索引的性能度量与假设把它当作非聚簇的是一样的。

⊖ 回忆15.1.3节我们建立的记号：R中元组的数目$T(R)$，和$\pi_L(R)$中不同元组的数目$V(R, L)$。

3. 这里，我们已经介绍了聚簇索引的概念——一个索引，其中要查找关键字上具有某个给定值的所有元组都出现在主要集中于存储具有那个查找关键字值的元组的块中。一般地讲，具有一个固定值的元组将被连续地存储，而且，仅在具有那个值的元组的第一块和最后一块中有另一个查找关键字值的元组。

聚簇文件组织是聚簇关系的一种组织的方式的例子，这种方式是，聚簇关系不是存放在全部用来存储这个关系的块中。假设关系S的一个元组和一个聚簇文件中许多R元组相联系。那么，R的元组没有存储到专门为R所占用的块中，但这些块是主要用于存放R的，并且我们称R是聚簇的。另一个方面，通常S不是一个聚簇的关系，因为它的元组通常存放在主要用于R元组的块中，而不是元组的块中。

(a) 我们不可能将R的块都尽可能地填满，因为正如13.1.6节中所讨论的，我们想为R的增长留下空间。

(b) R可能与一些不属于R的其他的元组存储在一起。例如在聚簇文件组织中就是这样。

此外，如果比例$B(R)/V(R,a)$不是一个整数，我们当然必须向上取整。最有意义的是，如果a是R的一个关键字，那么$V(R,a)=T(R)$，可以假定它一定比$B(R)$大得多，然而，我们确实需要一个磁盘I/O去检索具有关键字值v的元组，加上访问索引需要的磁盘I/O。 758

现在，让然我们考虑当$R.a$上的索引是非聚簇的时会发生什么。作为第一个近似，我们取回的每一个元组将在不同的块上，而且，我们必须访问$T(R)/V(R,a)$个元组。因此，$T(R)/V(R,a)$是我们估计所需要的磁盘I/O的数目。这个数字可能会更高，因为我们可能也需要从磁盘上读一些索引块；它也可能会低一些，因为存取的元组偶然地会出现在同一个块中，而且那个块留在内存缓冲区中。

例15.12 假设$B(R)=1000$，$T(R)=20\ 000$，也就是说，R有20 000个元组存放到20个块中。令a是R的一个属性，假设在a上有一个索引，并且考虑$\sigma_{a=0}(R)$操作。以下是一些可能的情形以及最坏情况下的磁盘I/O的数目。在所有情况中，我们将忽略访问索引块的代价。

1. 如果R是聚簇的，但是我们不使用索引，那么代价是1000个磁盘I/O。也就是说，我们必须检索R的每一个块。 759

2. 如果R不是聚簇的，而且我们不使用索引，那么代价是20 000个磁盘I/O。

3. 如果$V(R,a)=100$，并且索引是聚簇的，那么，基于索引的算法需要1000/100=10个磁盘I/O。

4. 如果$V(R,a)=10$，并且索引是非聚簇的，那么，基于索引的算法需要20 000/10=2000个磁盘I/O。请注意，如果R是聚簇的而索引不是的，这个代价将高于扫描整个关系R的代价。

5. 如果$V(R,a)=20\ 000$，也就是说，a是一个关键字，那么，基于索引的算法，不管索引是聚簇的或是非聚簇的，将需要1个磁盘I/O加上访问索引所需要的I/O。 □

索引-扫描作为一种访问的方法，对几个其他种类的选择操作也有所帮助。

a) 一个索引，比如一个B树，让我们在一个给定的范围内有效地访问查询关键字。如果关系R的属性a上的这样一个索引存在，那么对于比如$\sigma_{a\geq 10}(R)$或$\sigma_{a\geq 10 \text{ AND } a\leq 20}(R)$这样的选择，我们可以使用索引检索所需要的范围内的$R$元组。

b) 一个具有复杂条件C的选择，有时可以通过这种方法实现：在索引-扫描后，对索引-扫描检索到的元组进行另一个选择。如果C的形式是$a=v$ AND C'，其中C'可以是任何条件，那么，我们可以将选择分割成为两个选择的一个串联，第一个仅检查$a=v$，而第二个检查C'。第一个

很可能使用索引-扫描操作查询优化器在产生一个逻辑查询计划时可以做许多改进，这种选择操作的分割是其中的一种；这将在16.7.1节中专门讨论。

15.6.3 使用索引的连接

我们已经考虑过的所有的二元操作，以及γ和δ这两个一元的全关系操作，都可以使用某些索引，从而得益。我们将留下这些算法的大部分作为习题，而集中于关于连接的讨论。特别地，让我们考虑自然连接$R(X,Y) \bowtie S(Y,Z)$；回忆一下，X、Y和Z可以代表属性的集合，尽管把它们看做单个属性就够了。

对于我们的第一个基于索引的连接算法，假设S有一个属性Y上的索引。那么计算这个连接
760 的一种方式是检查R的每一个块，并在每一个块中考虑每一个元组t。令t_Y是t的对应于属性Y的部分。使用索引来找S中所有在Y部分上具有t_Y的元组。这些恰好是S中与R的元组t连接的元组，因此我们输出这些元组中每一个与t的连接。

磁盘I/O的数量依赖于几个因素。首先，假设R是聚簇的，我们将需要读取$B(R)$个块来得到R的所有元组。如果R是非聚簇的，那么可能会需要多达$T(R)$个磁盘I/O。

对R的每一个元组，我们必须平均读取S的$T(S)/V(S,Y)$个元组。如果S在Y上有一个非聚簇的索引，那么所需的磁盘I/O的数量是$T(R)T(S)/V(S,Y)$。但如果索引是聚簇的，那么仅$T(R)B(S)/V(S,Y)$个磁盘I/O就足够了[⊖]。对上述每一种情况，我们可能都需要为每个Y值增加几个磁盘I/O，用于读取索引本身。

不管R是否是聚簇的，访问S的元组的代价是占主导地位的，因此我们可以采用$T(R)T(S)/V(S,Y)$或者$T(R)(\max(1,B(S)/V(S,Y)))$分别作为$S$上非聚簇的和聚簇的情况下的连接方法的代价。

例15.13 让我们考虑一个运行的例子，$R(X,Y)$和$S(Y,Z)$分别是包括1000和500个块的关系。假设一个块可以容纳每个关系的10个元组，因而$T(R) = 10\ 000$，$T(S) = 5000$。同样，假设$V(S,Y) = 100$；也就是说，在S的元组中有100个不同的Y值。

假设R是聚簇的，并且S在Y上有一个聚簇索引。那么磁盘I/O的近似值，排除掉访问索引本身所需的后，是1000个磁盘I/O用来读取R的块(在上面的公式中忽略了)，再加上$10\ 000 \times 500/100 = 50\ 000$个磁盘I/O。对于前面讨论的同样的数据，这个数字明显超过了其他方法的代价。如果R是非聚簇的或S上的索引是非聚簇的，代价就会变得更高。 □

虽然例15.13使得索引-连接看起来像是一个很差的办法，但在另一些情况下用这种方法连接$R \bowtie S$就有意义得多。最常见的情况是，与S相比，R很小，而$V(S,Y)$很大。我们在习题15.6.5中讨论一个典型的查询，这个查询在连接前的选择使得R很小。在那种情况下，S的大部分将不再被算法检查，因为大多数Y值根本不出现在R中。但是，不论基于排序的还是基于散列的连接方法都将检查S的每一个元组至少一次。

15.6.4 使用有排序索引的连接

当索引是一个B树或者其他可以使我们容易地按照排序的序列提取关系的元组的结构时，我们可以有若干种其他的方法来利用该索引。可能最简单的情况是当我们想计算$R(X,Y) \bowtie$
761 $S(Y,Z)$时，对于R或S，我们在Y上有一个排序的索引。我们可以执行一个普通的排序-连接，但不必执行在Y上对其中一个关系排序的中间步骤。

作为一个极端的情况，不论是R还是S，如果都有在Y上的排序索引，那么我们仅需要执行

⊖ 但是记住，就像在15.6.2节中讨论的一样，如果$B(S)/V(S,Y)$很小，就被1替代。

15.4.5节中简单的基于排序的连接的最后一步。这个方法有时叫做zig-zag连接，因为我们在索引之间跳来跳去地查找它们共享的Y值。注意，R中具有不出现在S中的Y值的元组不需要检索，同样，S中具有不出现在R中的Y值的元组也不需要检索。

例15.14 假设我们有关系$R(X,Y)$和$S(Y,Z)$，两个关系都有在Y上的索引。在一个极小的例子中，假设R的元组的查询关键字的值(Y值)是有序的1、3、4、4、4、5、6，令S的元组的查询关键字的值是2、2、4、4、6、7。我们以R和S的第一个关键字开始，它们分别是1和2。因为1<2，我们跳过R的第一个关键字，看它的第二个关键字3。现在，S当前的关键字小于R当前的关键字，因此我们跳过S的两个2到达4。

在这一刻，R的关键字3小于S的关键字。因此我们跳过R的关键字。现在，两个关系的当前关键字值都是4。我们在两个关系中沿着所有关键字值4相关的指针，检索相应的元组，并将它们连接。注意，直到我们遇到共同的关键字4之前，没有检索关系的元组。

处理完4之后，我们走到R的关键字5和S的关键字6，由于5<6，我们跳到R的下一个关键字。现在关键字都是6，因而我们检索相应的元组并连接它们。既然R现在已经空了，我们知道两个关系中已经没有连接的元组了。 □

如果索引是B树，那么我们像图15-15那样，使用B树结构中从叶结点到叶结点的指针，按照顺序从左边开始扫描两个B树的叶结点。如果R和S是聚簇的，那么根据一个给定的关键字对所有元组的检索将带来一个与读取两个关系的片段成比例的磁盘I/O 数目。注意在极端情况下，即R和S有太多的元组以至于没有一个能装入可用的内存时，我们将不得不使用类似15.4.5节的修正。但是，在一般的情况下，使用公共的Y值来连接所有的元组这一步，具有与读它们同样多的磁盘I/O。

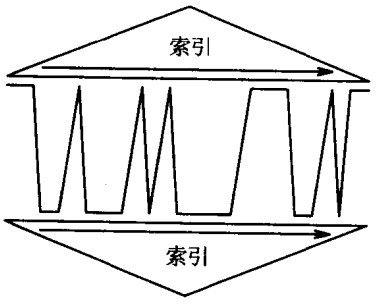


图15-15 使用两个索引的zig-zag连接

例15.15 让我们继续例15.13，看一看使用排序和索引相结合来做连接将怎样在这个数据上执行。首先，假设S有一个在Y上的索引，允许我们来检索按照Y排序的S的元组。在这个例子中，我们将假设关系和索引都是聚簇的。暂时，我们假设R上没有索引。

假设内存有101个可用的块，我们可以使用它们来为关系R的1000个块建立10个排序的子表。磁盘I/O的数量是2000，用来读和写R的全部内容。我们再使用内存的11个块——10个用于排序 762
的子表，一个用于经过索引检索到的S的元组的块。我们忽略操纵索引所需的磁盘I/O和内存缓冲区，但如果索引是B树，这些数目将非常小。在第二趟中，我们读R和S的所有的元组，使用总共1500个磁盘I/O，加上一次一块地读取索引块所需的少量磁盘I/O。这样我们估计总共的磁盘I/O在3500，这少于迄今为止考虑过的其他方法的代价。

现在，假设R和S都有Y上的索引，那么就不需要对任何一个关系排序。我们使用恰好1500个磁盘I/O来通过R和S的索引读它们的块。实际上，如果单从索引上确定R或S的一个大的片段不能与另一个关系的元组匹配，那么总的代价将远远低于1500次磁盘I/O。但是，不管哪一种情况，我们都应该增加读索引自身所需的少量磁盘I/O。 □

15.6.5 习题

习题15.6.1 假设属性$R.a$上有一个索引。描述怎样将这个索引用来改善下面操作的执行效

率。在什么情况下，基于索引的算法比基于排序或基于散列的算法更有效？

* a) $R \cup_S S$ (假设R和S没有重复，尽管它们可以有公共的元组)。

b) $R \cap_S S$ (同样，使用R和S集合)。

c) $\delta(R)$。

习题15.6.2 假设$B(R) = 10\ 000$，$T(R) = 500\ 000$。$R.a$上有一个索引，另外，对某个k有$V(R,a) = k$。在下面的情况下，给出$\sigma_{a=0}(R)$的代价，作为k的一个函数。可以忽略访问索引自身所需的磁盘I/O。

763

* a) 索引是聚簇的。

b) 索引是非聚簇的。

c) R是聚簇的，并且不使用索引。

习题15.6.3 如果操作是范围查询$\sigma_{C \leqslant a \text{ AND } a \leqslant D}(R)$，重复习题15.6.2。可以假设$C$和$D$是使得$k/10$的值在范围内的常量。

! **习题15.6.4** 如果R是聚簇的，但$R.a$上的索引是非聚簇的，那么依赖于k。我们可能愿意通过执行一个R的表扫描，或者愿意使用索引来实现一个查询。对于什么样的k值，我们更愿使用索引，如果关系和查询与：

a) 习题15.6.2相同。

b) 习题15.6.3相同。

* **习题15.6.5** 考虑SQL查询：

```
SELECT birthdate
FROM StarsIn, MovieStar
WHERE movieTitle = 'King Kong' AND starName = name;
```

这个查询使用“movie”关系：

```
StarsIn( movieTitle , movieYear , starName)
MovieStar(name, address, gender, birthdate)
```

如果我们把它翻译成关系代数，核心是一个在

$$\sigma_{movieTitle='King\ Kong'}(\texttt{StarsIn})$$

和MovieStar之间的等值连接，这可以像自然连接$R \bowtie S$一样来实现。既然仅有两部名为“King Kong”的电影，$T(R)$就非常小。假设S，即关系MovieStar，有一个在name上的索引。比较$R \bowtie S$的索引连接和基于排序或基于散列连接的代价。

! **习题15.6.6** 在例15.15中，我们讨论了连接$R \bowtie S$的磁盘I/O代价，其中R和S中的一个或两个都有在连接属性上的排序索引。但是，如果有太多的在连接属性上值相等的元组，那么在那个例子中描述的方法可能会失败。所描述的方法在什么限制条件下（用具有相同值的元组所占用的块数来刻画）将不再需要额外的磁盘I/O呢？

764

15.7 缓冲区管理

我们已经假设关系上的操作符可以得到某些数量的内存缓冲区，它们可以用来存储所需的数据。实际上，这些缓冲区很少预先分配给操作符，并且M的值将依赖于系统的条件而变化。为在数据库上查询那样的处理过程提供可用的内存缓冲区，这样一个中心任务就交给缓冲区管理器了。缓冲区管理器的职责是使处理过程得到它们所需的内存，并且尽可能缩小延迟和减少

不可满足的要求。缓冲区管理器的角色在图15-16中进行了说明。

15.7.1 缓冲区管理结构

有两个主要的缓冲区管理结构：

1. 在大多数关系型DBMS中，缓冲区管理器直接控制主存，或者

2. 缓冲区管理器在虚拟内存中分配缓冲区，允许操作系统来决定哪些缓冲区在任何时候都真正在主存中以及哪些缓冲区在操作系统管理的磁盘上的“交换空间”中。许多“主存”DBMS和“面向对象”的DBMS按这种方式操作。

不管DBMS使用哪种方法，都会引起同样的问题：缓冲区管理器应当限制使用的缓冲区数使得它们能够适合内存的容量。当缓冲区管理器直接控制主存，并且要求超过了可得到的空间时，就不得不通过将缓冲区的内容返回到磁盘上来清空缓冲区。如果缓冲的块没有改变，就简单地将它在主存中消除掉，但如果块已经发生改变了，它就必须回写到磁盘它自己的位置上。当缓冲区管理器在虚拟内存中分配空间时，它有机会来分配更多的可以超过内存容量的缓冲区。但是，如果这些缓冲区都真正使用，那就将会“颠簸”，这是一个操作系统常见的问题，即有许多块在磁盘交换空间移进移出。在这种情况下，系统花费大部分时间来交换块，而只能完成很少的有用工作。

765

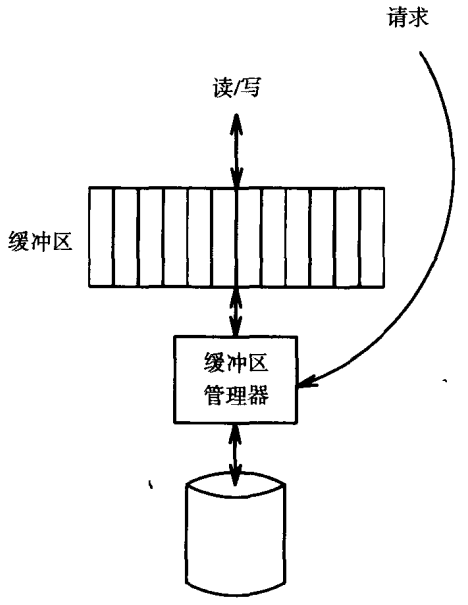


图15-16 缓冲区管理器响应内存访问磁盘块的要求

通常，当DBMS初始化时，缓冲区的数目是一个设置的参数。我们期望这个数目设置得使缓冲区占用可用的内存，而不管缓冲区是否被分配到内存或虚拟内存。在下面的讨论中，我们不关心采用哪种缓冲方式，只简单地假设有一个固定的缓冲池，即查询或其他数据库操作可用的缓冲区的集合。

查询处理的内存管理

我们假设缓冲区管理器为一个操作符分配M个缓冲区，M的大小依赖于系统条件（包括其他正在处理中的操作符和查询）并且可能动态地变化。一旦操作符获得了M个缓冲区，它可能使用其中的一些缓冲区来调入磁盘页，另一些用于索引页，还有一些用于排序处理或散列表。在某些DBMS中，内存并不是仅仅从一个内存池中进行分配，而是根据不同目的设置了多个独立的内存池，这些内存池拥有独立的缓冲管理器。例如，操作符可能从某个缓冲池中分配了D个缓冲区用于保存调入的磁盘页，从另一个独立的内存区分配了S个缓冲区用于排序，以及H个缓冲区用于建立散列表。这种技术为系统配置和“调整（tuning）”提供了更多的选择，但内存的使用可能不是全局最优的。

15.7.2 缓冲区管理策略

缓冲区管理必须做出的关键的选择是当一个新近要求的块需要一个缓冲区时，应该将什么块丢出缓冲池。从其他的诸如操作系统的调度策略中，你可能已经对通常使用的缓冲—替换策

766 略非常熟悉了。它们包括：

- 最近最少使用(LRU)。LRU规则是丢出最长时间没有读过或写过的块。这种方法要求缓冲区管理器保持一张表明每个缓冲区的块被访问的最后一次时间的表。它还要求每个数据库访问在这个表中生成一个表项，这样在保持这个信息的过程中有了一个有意义的成果。但是，LRU是一个有效的策略；直觉上，长时间没有使用的缓冲区比那些最近访问过的缓冲区有更小的最近访问的可能性。
- 先进先出(FIFO)。在FIFO策略中，当需要一个缓冲区时，被同一个块占用时间最长的缓冲区被清空，并用来装入新的块。在这种方法中，缓冲区管理器仅需要知道当前占用一个缓冲区的块装入缓冲区的时间。当块从磁盘读入内存的时候，可以生成表中的一个表项，且当块被访问时，不需要修改这个表。与LRU相比，FIFO需要较少的维护，但它会造成更多的错误。被重复使用的并称做B树索引的根块的块，将最终变成一个缓冲区中最旧的块。它将被写回到磁盘上，很快又被重新读入另一个缓冲区。
- “时钟”算法。这个算法是LRU的一个常用的、有效的近似。正如在图15-17中建议的，将缓冲区看做排好的一个环。一个“指针”指向这些缓冲区中的一个，如果想找到一个缓冲区来放置一个磁盘的块，就按顺时针旋转。每一个缓冲区有一个相应的“标志”，它或者是0，或者是1。带有0标志的缓冲区容易被选中，将其内容写回磁盘；具有1标志的缓冲区就不是。当一个块读进缓冲区时，它的标志就设为1。同样，当缓冲区的内容被访问过后，它的标志也设为1。当缓冲区管理器需要为一个新块分配缓冲区时，就按照顺时针旋转，查找能够找到的第一个0。如果它通过标志1的缓冲区，就将它设为0。这样，如果一个缓冲区从指针执行一个完全的旋转并将它的标志置为0，到接着再做另一个完整旋转来找没有改变的带标志0的缓冲区为止，都保持没有被访问，那么这个块被丢出缓冲区。举例来说，见图15-17，指针将把它左边的缓冲区由1置为0，再顺时针移动，找到带0的缓冲区，它的块将被替换，标志将被置为1。

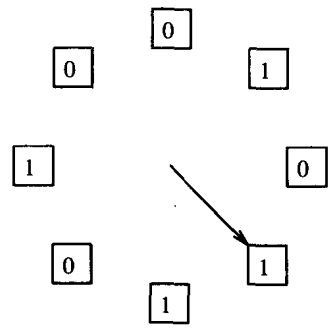


图15-17 时钟算法按轮转方式访问缓冲区，替换掉第一个具有0标志的缓冲区

使用时钟算法的进一步的技巧

选择要释放掉的缓冲区的“时钟”算法并不局限于第15.7.2节所描述的，标志取值为0和1的模式。例如，对于一个重要的页面，在开始时可以赋予它一个大于1的数字作为标志，然后每当“指针”经过这个页面时，标志的值就减1。事实上，被钉住的块的概念可以通过如下的方法来具体实现：赋予被钉住的块一个无限大的值作为标志，然后当系统在适当的时机要释放被钉住的块时，将它的标志置为0就可以了。

767

- 系统控制。查询优化器或者其他的DBMS部件可以给缓冲区管理器提供建议来避免像LRU、FIFO或者时钟这样的严格的策略引起的错误。回忆一下12.3.5节，有时候内存中的一些块如果不首先修改某些其他的指向它的块就不能移到磁盘上，这是某些技术原因造成的。这些块叫做“固定的”，任何缓冲区管理器都不得不修改缓冲替代策略来避免驱

除固定的块。这个事实给了我们机会，通过将其他的块定义为“固定的”来强迫它们保持在内存中，即便是没有不能写到磁盘上的技术原因。举例来说，对于上面提到的关于B树的根的问题的一个补救是“固定”根，强迫它在任何时候都保持在内存中。同样，对于像一趟散列—连接那样的算法，查询处理器可以“固定”较小的关系的块，确保在全部时间内它都留在内存中。

15.7.3 物理操作符选择和缓冲区管理的关系

查询优化器将最终选择一个用来执行给定查询的物理操作符的集合。物理操作符的选择可以假设能得到执行每一个操作符所需的一定数目的缓冲区M。但是，就像我们已经看到的，当 768
执行查询时，缓冲区管理器不打算或不能够保证它能得到这M个缓冲区。于是，关于物理操作符，有如下两个相关的问题要问：

1. 这个算法能够适应可得到的内存缓冲区数目M的值的变化吗？

2. 当不能够得到所期望的M个缓冲区，并且期望留在内存中的块实际上已被缓冲区管理器移到了磁盘上时，缓冲区管理器使用的缓冲区替换策略是怎样影响必须执行的额外的磁盘I/O数量的？

例15.16 作为这个问题的一个例子，让我们考虑图15-8中基于块的嵌套循环连接。基本的算法不真正依赖于M的值，尽管它的性能依赖于M。这样，在执行开始前找出M的值就足够了。

甚至于M有可能在不同的外循环迭代中将会发生改变。也就是说，每次将关系S(外循环的关系) 的一部分装入内存时，我们可以使用除了一个以外的所有可用的缓冲区；保留的缓冲区是留给R的块的，R是内循环中的关系。这样，我们执行外循环的次数依赖于每一次迭代可用的缓冲区个数的平均数。但是，只要平均有M个缓冲区可用，那么就15.3.4节分析的代价就成立。在极端情况下，我们可能有幸在第一次迭代就找到足够的可用缓冲区来装入S所有的内容，在这种情况下，嵌套循环连接就变成了15.2.3节的一趟连接。

如果在外循环的一次迭代上固定用于S的$M-1$个块，那么在运行中就不会失去它们的缓冲区。另一方面，在那次迭代中，可能会有更多的缓冲区变为可用。这些缓冲区允许R的多于一个的块同时保存在内存中，但除非我们很仔细，否则额外的缓冲区将不会改善嵌套循环连接的运行时间。

举个例子，假设我们使用LRU缓冲区替换策略，并且有k个可用的缓冲区来容纳R的块。当我们依次读R的块的时候，在外循环这次迭代的最后留在缓冲区中的块将是R的最后k个块。我们接着往S的$M-1$个缓冲区中装入S的新块，并且开始在外循环的下一次迭代中再次读R的块。但是，如果再次从R的头开始，R的k个缓冲区将需要替换，我们不能仅仅因为$k>1$就节省磁盘I/O数量。

嵌套循环连接的一个较好的实现将按照一个交替的序列来访问R的块：第一到最后，再最后到第一个。在这种方式下，如果R有k个可用的缓冲区，我们将在外循环的每次迭代中(第一次除外) 节省k次磁盘I/O数目。也就是说，第二次和后续的迭代对R仅需要$B(R)-k$次磁盘I/O。 769
注意，即便$k=1$(也就是说，对R没有额外的缓冲区)，我们也能每次迭代节省一次磁盘I/O。 □

其他的算法也受到M可以变化的事实和缓冲区管理器使用的缓冲区替换策略的影响。下面是一些有用的经验。

- 如果对一些操作符使用基于排序的算法，那么适应M的变化是可能的。如果M减少，我们可以改变子表的大小，因为我们讨论的基于排序的算法不依赖于同样大小的子表。主要的局限是当M减小时，我们被迫不得不建立许多子表，使得在合并过程中，我们不可

能为每一个子表分配一个缓冲区。

- 子表的内存排序可以由许多不同的算法来实现。由于像归并排序和快速排序等算法是递归的，所以大多数时间内都只占用内存很小的区域。这样，不论是LRU还是FIFO，基于排序的算法的这一部分都将运行得很好。
- 如果算法是基于散列的，且M减少，我们可以减少桶的数目，只要桶不会变得太大以至于它们不能装入分配的内存。但是，与基于排序的算法不同，当算法运行时，我们不能对M的变化做出反应。一旦桶的数目选定了，在整个第一趟中，它都保持固定的数目。如果缓冲区变为不可用了，则属于某些桶的块将不得不被交换出去。

15.7.4 习题

习题15.7.1 假设我们想执行连接$R \bowtie S$，可用的内存在M 和$M/2$之间变化。依据M、$B(R)$和$B(S)$，给出保证下面算法可以执行的条件：

* a) 一趟连接。

* b) 两趟的基于散列的连接。

c) 两趟的基于排序的连接。

! **习题15.7.2** 如果额外的缓冲区可用，并且是下面的缓冲区替代策略，那么嵌套循环连接的磁盘I/O数目将怎样改进：

a) 先进先出。

770 b) 时钟算法。

!! **习题15.7.3** 在例15.16中，我们表明了利用额外的缓冲区的可能，方法是缓冲存储R的不止一个块和在外部循环中偶数地迭代上按照反序访问R的块。但是，还可以为仅为R保持一个缓冲区并增加供S使用的缓冲区数目。哪一个策略产生最小的磁盘I/O数量？

15.8 使用超过两趟的算法

虽然两趟对于除了最大的关系外的所有关系上的操作已经足够了，但我们应当看出，15.4节和15.5节讨论的主要的技术，通过对算法进行推广，根据需要使用多趟，就可以处理任意大小的关系。在本节中，我们将考虑基于排序的和基于散列的方法的推广。

15.8.1 基于排序的多趟算法

在11.4.5节中，我们提到了怎样将两阶段多路归并排序扩展成三趟算法。事实上，有一个简单的递归排序方法，它允许我们不管关系多么大，都能完整地排序，或者如果我们愿意，对任意给定的n，可以建立n个排序的子表。

假设我们有M个可用的内存缓冲区来对关系R排序，并假设R是按照聚簇存储的。那么，按如下方式做：

基础：如果R可装入M个块中(也就是说，$B(R) \leqslant M$)，那么将R读入内存，使用你最喜爱的排序算法来把它排序，并将排好序的关系写到磁盘上。

归纳：如果R不能装入内存，将R的块分成M个组，称做$R_1, R_2, \cdots, R_M$。对每个$i=1,2,\cdots,M$，递归地将R_i排序。接着，像11.4.4节那样，将M个排序的子表合并。

如果不仅仅对R排序，而是要在R上执行一个一元操作，如γ或δ，那么修改上面的算法，使得在最后的归并中，我们在排序子表的前端的元组上执行操作。即，

- 对于δ，输出每一个不同元组的一个副本，并跳过这个元组的其他副本。
- 对于γ，仅在分组属性上排序，然后像在15.4.2节讨论的一样，将在分组属性上具有一个

给定值的那些元组以适当的方式结合。

当我们想执行一个二元操作的时候，比如交或连接，我们基本上使用同样的思想，所不同的只是这两个关系首先分成总数为M的子表。然后，每一个子表通过上面的递归算法排序。最后，我们读M个子表中的每一个，将每一个放入一个缓冲区，并且按照15.4节相应部分描述的 771
方式来执行操作。

我们可以按照我们的愿望将M个缓冲区在关系R和S之间分开。但是，为了使总的趟数最少，我们通常根据关系的块数，按比例地划分缓冲区。也就是说，R得到缓冲区的$M \times B(R)/(B(R)+B(S))$，$S$得到剩余部分。

15.8.2 基于排序的多趟算法的性能

现在，让我们探讨所需的磁盘I/O数目、被操作的关系大小和内存大小之间的关系。令$s(M,k)$是我们使用M个缓冲区和k趟能排序的最大的关系的大小。那么我们可以按照下面的方法计算$s(M, k)$：

基础：如果$k=1$，即允许进行一趟，那么我们有$B(R) \leqslant M$。换句话说，$s(M, 1)=M$。

归纳：假设$k>1$。那么我们将R分成M片，每一片必须是通过$k-1$趟可排序的。如果$B(R)=s(M, k)$，那么R的M个片的每一片的大小$s(M,k)/M$，不能超过$s(M, k-1)$。即：$s(M, k)=Ms(M, k-1)$。

如果我们展开上面的递归，会发现

$$s(M, k) = Ms(M, k-1) = M^2s(M, k-2) = \cdots = M^{k-1}s(M, 1)$$

既然$s(M, 1) = M$，我们得出结论$s(M, k)= M^k$。也就是说，如果$B(R) \leqslant s(M, k)$，即$B(R) \leqslant M^k$，那么通过k趟，我们可以将关系R排序。换句话说，如果我们想在k趟中将R排序，那么可以使用的缓冲区的最小数目是$M=(B(R))^{1/k}$。

排序算法的每一趟从磁盘上读取所有数据并再将它们写回。这样，一个k趟排序算法需要$2kB(R)$个磁盘I/O。

现在，让我们将多趟连接$R(X,Y) \bowtie S(Y, Z)$作为关系上二元操作的代表，考虑它的代价。令$j(M, k)$是在k趟中，使用M个缓冲区的最大的块数，即我们可以连接关系中的块的总数等于或小于$j(M, k)$。也就是说，如果$B(R)+B(S) \leqslant j(M, k)$，连接就可以实现。

最后一趟中，我们归并两个关系的M个排序的子表。每个子表是使用$k-1$趟排序的，所以它们中的每一个的大小都不会超过$s(M, k-1)=M^{k-1}$，或者总的大小是$Ms(M, K-1) =M^k$，即$B(R) +B(S)$不会超过M^k，或者换句话说，$j(M, k)= M^k$。颠倒参数的角色，我们也可以说K趟计算连接需要$(B(R)+B(S))^{1/k}$个缓冲区。

为了计算多趟算法所需的磁盘I/O数量，我们应当记住，不像排序那样，我们不统计为连接或其他关系操作将最终结果写到磁盘上的代价。这样，我们使用$2(k-1)(B(R)+B(S))$个磁盘I/O来将子表排序，另外，$B(R)+B(S)$个磁盘I/O在最后一趟中读取排序的子表。最后的结果是总 772
共$(2k-1)(B(R)+B(S))$个磁盘I/O。

15.8.3 基于散列的多趟算法

对于大关系上的操作，有一个递归地使用散列的方法。我们将一个或两个关系散列到$M-1$个桶中，M是可用的内存缓冲区的数目。对于一元操作，我们再将操作分别应用到每一个桶。如果操作是二元的，比如连接，我们将操作应用到每一对相应的桶上，就像它们是整个的关系。对于我们已经考虑的普通的关系操作——消除重复、分组、并、交、差、自然连接和等值连接——在整个关系上操作的结果将是桶上结果的并集。我们可以递归地将这个方法描述为：

基础：对于一元操作，如果关系能装到M个缓冲区中，就将它读入内存并执行操作。对于二元操作，如果有一个关系能装到$M-1$个缓冲区中，就将这个关系读入内存，再将第二个关系一次一块地装入第M个缓冲区，这样来执行操作。

归纳：如果没有一个关系能够装入内存，那么就像15.5.1节中所讨论的，将每个关系散列到$M-1$个桶中。在每个桶上或每个相应的桶对上递归地执行操作，并将每个桶或桶对的输出积累起来。

15.8.4 基于散列的多趟算法的性能

在下面，我们将做一个假设，即当散列一个关系时，要将元组尽可能平均地分到桶中。事实上，如果我们选择一个真正随机散列的函数，将会近似地符合这个假设，但在将元组分布到桶的过程中总会有某些不均衡。

首先，考虑一个一元操作，例如关系R上使用M个缓冲区的γ或δ。令$u(M, k)$是k趟散列算法能够处理的最大的关系的块数。我们可以通过如下方式递归地定义u：

基础：$u(M,1)=M$，因为关系R必须能装入到M个缓冲区中，也就是说，$B(R)\leqslant M$。

归纳：假设第一步将R分到$M-1$个大小相等的桶中。这样，我们可以按照如下方式计算$u(M,k)$。为下一趟准备的桶必须足够小，使得它们能够在$k-1$趟中处理；即，桶的大小是
773 $u(M, k-1)$。既然R被分到$M-1$个桶中，那么我们必定有$u(M, k)=(M-1)u(M, k-1)$。

如果展开上面的递归，我们会发现$u(M,k)=M(M-1)^{k-1}$，或者近似地，假设M很大，$u(M,k)=M^k$。换句话说，如果$M\leqslant(B(R))^{1/k}$，我们可以用M个缓冲区经过k趟来执行关系R上的一元关系操作。

我们可以为二元操作做一个简单的分析。就像在15.8.2节一样，让我们考虑连接。令$j(M, k)$是$R(X, Y)\bowtie S(Y, Z)$涉及的两个关系R和S中较小者的大小的上限。这里，就像以前一样，M是可用的缓冲区的数目，k是我们可以使用的趟数。

基础：$j(M,1)=M-1$；也就是说，像我们在15.2.3节讨论的，如果我们使用一趟算法来连接，那么或者R或者S一定能够装入$M-1$个块中。

归纳：$j(M, k)=(M-1)\,j(M, k-1)$；也就是说，在k趟的第一趟中，可以将每一个关系分到$M-1$个桶中，我们可能期望每个桶是整个关系的$1/(M-1)$，但我们必须能够在$M-1$趟中连接每一个相应的桶对。

通过展开$j(M,k)$的循环，我们得到结论$j(M,k)=(M-1)^k$。再次假设M很大，我们可以近似地说$j(M,k)=M^k$。也就是说，如果$M^k\geqslant\min(B(R),B(S))$，我们可以使用$k$趟和$M$个缓冲区来连接$R(X,Y)\bowtie S(Y,Z)$。

15.8.5 习题

习题15.8.1 设$B(R)=20\,000$，$B(S)=50\,000$，并且$M=101$。描述下面计算$R\bowtie S$的算法的执行。

* a) 基于排序的三趟算法。

b) 基于散列的三趟算法。

! **习题15.8.2** 有几个我们已经讨论过的用来提高两趟算法性能的“技巧”。辨别下面的技巧是否可以用于多趟算法，如果可以，怎样用？

a) 15.5.6节的混合散列-连接技巧。

b) 通过在磁盘上连续地存储块（15.5.7节）来改善基于排序的算法。

774 c) 通过在磁盘上连续地存储块（15.5.7节）来改善基于散列的算法。

15.9 关系操作的并行算法

数据库操作经常大量消耗时间并且涉及大量数据，因而通常采用并行处理。在本节中，我们将回顾并行机的基本结构。然后，我们再集中在“无共享”结构上，尽管它不比其他的并行应用更优越，但它表现出对数据库操作是最有效的。对于大多数关系数据库操作的标准算法的一个简单修改将能够极好地利用并行机制。也就是说，在一台p个处理器的机器上完成一项操作的时间大约是在一个单处理器机器上完成该操作的时间的$1/p$。

15.9.1 并行模型

所有并行机的核心都是一个处理器的集合。处理器的数量 p 经常是很大的，成百上千。假设每个处理器有它自己的局部高速缓存，这在图15-18中没有明确显示。在大多数的组织结构中，每个处理器还有局部内存，我们已经标明了。对数据库处理极其重要的是还有连同这些处理器一起的许多磁盘，可能每个处理器有一个或多个，或者在某些结构中，所有的处理器可以直接访问一个大的磁盘集合。

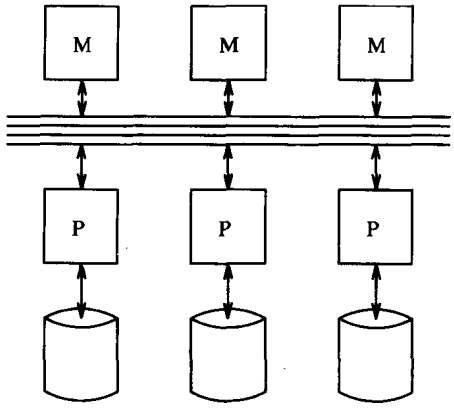


图15-18 一台共享内存的机器

另外，并行计算机都有某些在处理器之间传送信息的通信设备。在我们的图中，显示的通信方式好像对于所有的机器部件都有一个共享的总线。但是，实际上在最大的机器中，总线不能连接所有的处理器或其他部件，因此在许多结构中，互连系统是一个强大的交换机，辅之以在局部簇中连接处理器子集的总线。

775

并行机的三个最重要的类是：

1. *共享内存*。在图15-18描述的结构中，每一个处理器可以访问所有处理器的所有内存。也就是说，对整个机器，有一个单一的物理地址空间，而不是每个处理器一个地址空间。图15-18实际上太极端，它表明每个处理器根本没有私有的内存。每个处理器应当有某些局部的、能单独使用的内存。但是，当需要时，它可以直接访问其他处理器的内存。这一类的大型机是NUMA(nonuniform memory access) 类型，这意味着一个处理器访问“属于”其他处理器内存的数据所花的时间在一定程度上多于访问它“自己”的内存或它的局部簇中处理器内存的时间。但是，当前结构中内存访问时间的差异并不大。相反，不管数据在哪里，内存访问的时间都远远大于高速缓存的访问时间，因此关键的问题是，处理器所需的数据是否在它自己的高速缓存中。

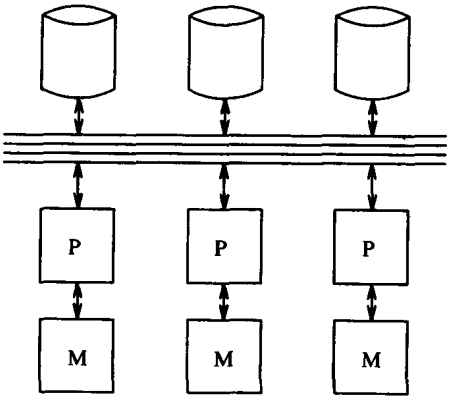


图15-19 一台共享磁盘的机器

2. *共享磁盘*。在图15-19所示的结构中，每一个处理器都有它自己的内存，其他的处理器不能直接地访问到。但是，磁盘可以由任何一个处理器通过通信网络访问到。磁盘控制器管理来自不同处理器的潜在的竞争需求。磁盘和处理器的数目不必像图15-19所示的那样是

相同的。

776 3. 无共享。如图15-20所示，所有的处理器都有它们自己的内存和一个或多个磁盘。所有的通信都经过从处理器到处理器的通信网络。举例来说，如果一个处理器*P*想从另一个处理器*Q*的磁盘上读元组，那么处理器*P*向*Q*发请求数据的消息。接着，*Q*从它的磁盘上获得元组，并用另一条消息把它们通过网络发送给*P*。

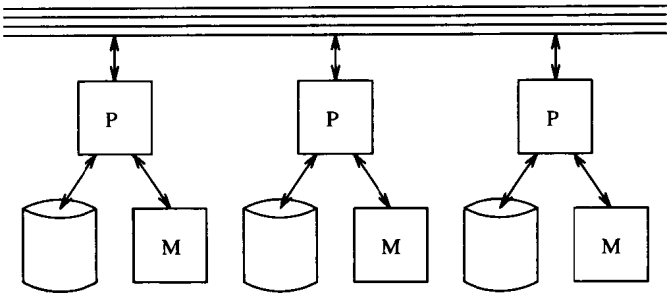


图15-20 一台无共享的机器

就像我们在本节引言中提到的，无共享的体系结构是“数据库机”最常用的结构。数据库机是为支持数据库而专门设计的并行计算机。无共享机器的建造是相对便宜的，但是我们为这些机器设计算法的时候，必须注意从一个处理器到另一个处理器发送数据的代价是很高的。

通常，数据必须在处理器之间的消息中发送，这需要可观的系统开销。两个处理器必须都执行一个支持消息传送的程序，这里会有一个与通信网络相关的竞争或延迟。消息的代价通常可以被分为一个大的固定的系统开销加上每个字节传送的少量的时间。这样，设计一个并行算法使得在处理器之间的通信包括大数据量的发送将很有益处。举例来说，我们可将处理器*P*中将要发送到处理器*Q*的数据的若干个块进行缓冲。如果*Q*不立即需要数据，那么等待直到我们在*P*有一个长消息，再将它发到*Q*，这将是非常有效的。

其他并行结构上的算法

就像无共享的机器一样，共享磁盘的机器也喜欢长消息。如果所有的通信都经过一个磁盘，那么我们需要移动块中的数据。如果我们可以组织将被移动的数据，即它们都在一个磁道或一个磁柱面上，那么就像11.5.1节中讨论的一样，我们可以节省许多延迟。

另一方面，共享内存的机器允许在两个处理器这间通过内存来进行通信。这里没有发送消息所需的广泛适用的软件，读或写内存的代价与包含的字节数成比例。这样，共享内存的机器将能够有效利用处理器之间所需的快速、顺畅和简洁的通信的算法。有趣的是，虽然这样的算法在其他领域已有了，但数据库处理看起来不需要这样的算法。

15.9.2 一次一个元组的并行操作

通过考虑选择操作符来开始无共享机的并行算法的讨论。首先，我们必须考虑怎样最好地存储数据。就像首先在11.5.2 节中建议的，将数据分布到尽可能多的磁盘上将是很有用的。为了方便，我们将假设一个处理器只有一个磁盘。那么如果有*P*个处理器，就将任意一个关系*R*
777 的元组平均地分到*p*个处理器的磁盘上。

假设我们想执行$\sigma_C(R)$。我们可以使用每一个处理器来检查分布在它自己的磁盘上的元组。每一个处理器找到满足*C*条件的元组，并将它们复制到输出。为了避免处理器之间的通信，我们将$\sigma_C(R)$中的元组*t*存储到那些磁盘上有*t*的处理器中。这样，关系$\sigma_C(R)$的结果就像*R*一样分

布在各个处理器上。

因为$\sigma_C(R)$可以是另一个操作的输入关系，并且我们希望使空闲时间最小在任何时间都让所有的处理器保持繁忙，因此我们希望将$\sigma_C(R)$平均地分到各个处理器中。如果我们是在做一个投影，而不是选择，那么每个处理器上$\pi_L(R)$ 中元组的数目与处理器上R的元组的数目相同。这样，如果R是平均分布的，那么它的投影也是平均分布的。但是，与R的分布相比，选择将大大改变结果中元组的分布。

例15.17 设选择是$\sigma_{a=10}(R)$，即，找出R中所有属性a (假设是R的一个属性) 的值是10的元组。还假设我们已经按照属性 a 的值将 R 进行了划分。那么所有具有a=10的元组在一个处理器上，并且整个的关系$\sigma_{a=10}(R)$就在一个处理器上。 □

为了避免例15.17表明的问题，我们需要仔细考虑在处理器之间划分存储关系的策略。可能我们能够做的最好的就是使用包含一个元组的所有成分的散列函数h，使得改变元组t的一个成分就可以将$h(t)$改变到任何可能的桶号[⊖]。举例来说，如果想要B个桶，我们可以将每一个成 778
分转化为0 ~ B - 1之间的一个整数，并将对应每一个成分的整数加起来，结果用B除，余数作为桶号。如果B也是处理器的数目，那么我们可以将每个处理器与一个桶关联，并把桶的内容传给那个处理器。

15.9.3 全关系操作的并行算法

首先，让我们考虑操作$\delta(R)$，这是一个不那么典型的全关系操作。如果我们按照15.9.2节的建议，使用一个散列函数分布R的元组，那么我们在同一个处理器上放置R的重复元组。如果是这样，那么通过在每一个处理器的R部分上使用一个标准的、单处理器算法（就像15.4.1节和15.5.2节的例子）来并行地产生$\delta(R)$。同样，如果使用同一个散列函数来分发所有R和S的元组，那么我们可以通过在每一个处理器的R和S的部分上并行工作来取得R和S的并、交或差。

但是，假设R和S不是使用同一个散列函数来分布，并且我们希望执行它们的并[⊜]。在这种情况下，我们必须生成R和S的所有元组的副本，并按照单一的一个散列函数 h[⊛]来分发它们。

我们使用散列函数h，并行地将R和S的元组散列到每一个处理器上。散列过程就像15.5.1节那样进行，但当一个处理器j上相应于桶i的缓冲区满了时，并不是在 j 上将它移动到磁盘，而是将缓冲区的内容输送到处理器i。如果我们有空间在内存中存放每个桶的几个块，那么在将它们输送到处理器i之前，我们可以等待用桶i的元组填满几个缓冲区。

这样，处理器i接收R和S的所有属于桶i的元组。在第二阶段，每个处理器对属于它的桶中的R和S的元组执行并操作。其结果是，关系$R \cup S$将分布在所有处理器上。如果散列函数h真正随机产生桶中元组的位置，那么我们可以期望$R \cup S$在每一个处理器上的元组的数目近似相等。

交和差的操作可以和并一样来执行；它并不管这些操作是集合还是包的形式。另外，

- 为了计算$R(X,Y) \bowtie S(Y,Z)$，我们将R和S的元组散列到与处理器数目相同的桶中。但是，我们使用的散列函数必定仅依赖于Y的属性，而不是全部的属性，这使得连接元组总是被 779
送到同一个桶中。就像并那样，我们将桶i的元组送到处理器i。那么我们就可以在每一个

⊖ 特别地，我们不希望使用分割散列函数（在14.2.5节中讨论），因为那会将具有一个给定的属性值如a=10的所有元组存放到桶的很小的子集中。

⊜ 在原理上，这个并可以是基于集合的，也可以是基于包的。但是15.2.3节的包-并技术，是复制所有的操作对象的元组，它是并行工作的，因此我们可能不希望使用这里描述的算法来执行包-并。

⊛ 如果用来分发R或S的元组的散列函数是已知的，那么我们可以使用该散列函数来分发另一个关系，而不是两个关系都分发。

处理器上使用在本章中已经讨论过的任何一个单处理器连接算法来执行连接了。

- 为了计算分组和聚集$\gamma_L(R)$，我们使用一个仅依赖于列表L中的分组属性的散列函数h来分发R的元组。如果每个处理器具有对应于h的一个桶的所有元组，那么可以使用任何一个单处理器γ算法来在这些元组上局部地执行γ_L操作。

15.9.4 并行算法的性能

现在，让我们比较在有p个处理器的机器上一个并行算法的运行时间与使用单处理器在同样数据同样操作上的算法的执行时间。总的工作量——磁盘I/O和处理器周期——并行机不可能比单处理器机更小。但是，因为有p个处理器与p个磁盘一起运转，我们可以期望多处理器耗费的时间比单处理器小得多。

就像我们在15.9.2节中建议的，如果关系R是均匀分布的，一元操作符，如$\sigma_C(R)$，可以在单处理器执行这个操作的时间的$1/p$内完成。磁盘I/O的数目与单处理器上的选择操作必然是一样的。惟一的差别是，平均每个处理器上有R的p个半满的块，而不是将R的所有元组存储在一个处理器的磁盘上，从而只有单个半满的块。

780 现在，考虑二元操作，如连接。我们在连接属性上使用一个散列函数，它将每一个元组传送到p个桶中的一个，其中p是处理器的数目。对于所有的i，为了将桶i的元组传送到处理器i，我们必须将每个元组从磁盘上读入内存，计算散列函数，并将所有这些元组，除去碰巧属于它自己的处理器上的桶的那p分之一的元组外，传送出去。如果我们正在计算$R(X,Y)\bowtie S(Y,Z)$，那么需要做$B(R)+B(S)$次磁盘I/O来读R和S的所有元组，并且确定它们的桶。

接着我们必须经过网络，将$\left(\frac{p-1}{p}\right)(B(R)+B(S))$个数据块传送到适当的处理器上；仅有$(1/p)$的已在正确处理器上的元组不需要传输。依赖于机器的体系结构，传送的代价可以大于或小于同样的磁盘I/O数目的代价。但是，我们将假设网络上的传送比磁盘和内存之间的数据移动的代价小得多，因为网络上的传送没有包含物理动作，但是磁盘I/O中包括。

原理上讲，我们可以假设接收处理器必须首先在它自己的磁盘上存储数据，然后在收到的元组上执行一个局部连接。举例来说，如果我们在每一个处理器上使用两趟的排序连接，朴素的并行算法将在每一个处理器上使用$3(B(R)+B(S))/p$次磁盘I/O，因为每一个桶中关系的大小近似为$B(R)/p$和$B(S)/p$，并且这种连接类型将对每个操作对象关系占用的桶花费三次磁盘I/O。对于这个代价，我们将在每个处理器上增加另外的$2(B(R)+B(S))/p$次磁盘I/O，用于每个元组的第一次读和在元组的散列和分布的过程中接收元组的处理器对每一个元组进行存储。我们应当还加上传送数据的代价，但实际上与同样数据的磁盘I/O代价相比，这个代价是可以忽略的。

上面的比较说明了多处理器的价值。当我们总共做了更多的磁盘I/O，每个数据块五次磁盘I/O，而不是三次时，所消耗的时间，以在每个处理器上执行磁盘I/O的时间来计，从$3(B(R)+B(S))$降到了$5(B(R)+B(S))/p$，对于大的p来说，这是一个有意义的胜利。

此外，有许多提高并行算法的速度的方法，使得总共的磁盘I/O的数量不多于单处理器算法所需的磁盘I/O。事实上，因为我们在每个处理器上对一个小关系操作，因此，可能使用一个对每个数据块使用较少的磁盘I/O的局部连接算法。举例来说，即便R和S很大，使得在单处理器上需要两趟算法，我们仍能够在$(1/p)$的数据上使用一趟算法。

如果将一个块传送到它的桶的处理器时，处理器能够立即使用这个块作为它的连接算法的一部分，我们就可以避免每个块的两次磁盘I/O。大多数已知的连接和其他关系操作符允许这种使用，在这种情况下，并行算法看起来就像15.8.3节中在第一趟时使用散列技术的多趟算法。

例15.18 考虑我们运行的例子$R(X,Y) \bowtie S(Y,Z)$，其中R和S各占用1000个和500个块。现在，设有一台10个处理器的机器，每个处理器有101个缓冲区。而且，假设R和S在这10个处理器上均匀分布的。

开始时，我们使用仅依赖于连接属性Y的散列函数h，将R和S的每一个元组散列到10个“桶”中的一个。这10个桶代表10个处理器，元组被传送到对应“桶”的处理器上。读R和S的

大 错 误

当使用基于散列的算法来在处理器之间分布关系并执行操作时，就像例15.18那样，我们必须当心不要过度使用一个散列函数。举例来说，假设为了计算R和S的连接，我们使用散列函数h在处理器之间将关系R和S的元组进行散列。我们可能想尝试使用h将S的元组局部地散列到桶中，就像我们在每个处理器上执行一趟的散列—连接那样。但如果我们这样做了，所有的那些元组都会跑到一个桶中，例15.18中建议的内存连接将极端低效。

元组的总的磁盘I/O的数目是1500，或者说每个处理器150个。每个处理器有大约15个块的数据散列到它自身，用于其他处理器数据的块，因此，它将135个块传送到其他的9个处理器上。这样，总的通信量是1350个块。

我们将安排处理器在传送R的元组之前传送S的元组。因为每一个处理器接收大约50个S的元组的块，它就能够使用它的101个缓冲区中的50个，在内存数据结构中存储这些元组。然后，当处理器开始发送R的元组时，将每一个元组与局部的S的元组比较，任何连接元组的结果都将作为输出。 781

在这种方式中，连接惟一的代价是1500次磁盘I/O，比本章中讨论的任何其他方法都小得多。而且，在每个处理器上操作所消耗的时间大致上是150次磁盘I/O，加上在处理器之间传送元组和执行内存计算所需的时间。注意到150次磁盘I/O少于在单处理器上执行同样算法时间的1/10。赢得这些并不仅仅因为我们有10个运行的处理器，而且在10个处理器中总共有1010个缓冲区为我们赢得了额外的效率。

当然，你可能会说，如果在一个单处理器上有1010个缓冲区，那么我们连接的例子可能会使用1500次磁盘I/O在一趟中完成。但是，因为多处理器具有的内存通常与处理器的个数成比例，所以我们同时利用了多处理器的两个益处来得到两个独立的提高：一个与处理器个数成比例，另一个是因为额外的内存允许我们使用更高效的算法。 □

15.9.5 习题

习题15.9.1 假设一次磁盘I/O占用100毫秒。令$B(R)=100$，所以在一个单处理器的机器上计算$\sigma_C(R)$的磁盘I/O将占用10秒。如果这个选择在一台有p个处理器的机器上执行，效率提高多少？

* a) $p=8$

b) $p=100$

c) $p=1000$

! **习题15.9.2** 在例15.18中，我们描述了一个算法，它通过首先将元组散列分布到多个处理器上，再在处理器上执行一趟连接算法来并行地计算$R \bowtie S$。按照$B(R)$和$B(S)$，即相关的关系的大小，p(处理器的数目) 和M(每一个处理器上内存的块数)，来给出算法可以成功执行的条件。 782

15.10 小结

- *查询处理*：查询被编译，其中涉及大量的优化，然后被执行。查询执行的研究包括了解与SQL功能匹配的扩充的关系代数上的执行操作的方法。
- *查询计划*：查询首先被编译为逻辑查询计划，通常就像关系代数表达式，然后如在第16章中将要讨论的，通过为每一个操作符选择一个实现、对连接排序和做出一些其他的决定，来将逻辑查询计划转化成物理查询计划。
- *表扫描*：为了访问关系的元组，有几个可行的物理操作符。表扫描操作符简单地读取存放关系的元组的块。索引扫描使用索引来找到元组，排序扫描产生排好序的元组。
- *物理操作符的代价度量*：通常，执行一个操作占用的磁盘I/O的数量是消耗时间的主要部分。在我们的模型中，只计磁盘I/O时间，并且计算读操作对象所需的时间和空间，而不管写出结果的代价。
- *迭代器*：如果我们把一个查询的执行看做是由迭代器来操作，那么一个查询执行涉及的几个操作可以很方便地混合起来。这个机制包含三个函数，用于打开关系的结构、得到关系的下一个元组和关闭这个结构。
- *一趟算法*：只要关系代数操作符的一个操作对象能够装入内存，我们就可以将小的关系读进内存，并一次一个块地读另一个操作对象来执行这个操作符。
- *嵌套循环连接*：这个简单的连接算法甚至在两个操作对象都不能装入内存时也能运转。它将较小的关系尽可能多地读进内存，并将它与整个的另一个作对象比较；这个过程重复执行直到较小的关系的所有元组都进过内存。
- *两趟算法*：除了嵌套循环连接，大多数对于不能装入内存的操作对象的算法，或者是基于排序的和基于散列的，或者是基于索引的。
- *基于排序的算法*：这些算法将它们的操作对象分割成内存大小的、排序的子表。然后排序的子表被适当地归并来产生所需的结果。

783

- *基于散列的算法*：这些算法使用一个散列函数将操作对象分割到桶中。然后操作被分别应用到桶(对一元操作) 和桶对(对二元操作)上。
- *散列与排序*：基于散列的算法常常优与基于排序的算法，因为它仅要求一个操作对象是“小的”。在另一方面，当有另外的原因需要保持数据排序时，基于排序的算法表现得很好。
- *基于索引的算法*：对于条件是索引属性等于常量的选择来说，使用索引是提高性能的一种极好的方式。当一个关系是小的，而另一个具有连接属性上的索引时，基于索引的连接也是很好的。
- *缓冲区管理器*：内存块的可用性是由缓冲区管理器来控制的。当内存中需要一个新的缓冲区时，缓冲区管理器使用一个读者所熟悉的替代策略，如最近最少使用，来决定哪一个缓冲区的内容返回到磁盘上。
- *处理缓冲区数目变化*：通常，用于一个操作的可用的内存缓冲区的数目是不可预测的。如果是这样，当可用的缓冲区数目减少时，用于实现操作的算法会大大降级。
- *多趟算法*：基于排序的或基于散列的两趟算法可以自然地递推到三趟或更多趟，用来运行更大的数据量。
- *并行机*：当前的并行机可以根据特点分为共享内存、共享磁盘和无共享。对于数据库应

用来说，无共享结构通常是最有效的。

- *并行算法*：关系代数的操作通常能够在并行机上获得一个接近于处理器数目的因子的提高速度。常用的算法是将数据散列到对应于处理器的桶，再将数据传送到适当的处理器上。接着每个处理器对它的局部数据执行操作。

15.11 参考文献

[6]和[2]是两个查询优化的综述。[8]是分布式查询优化的一个综述。[5]中有一个对连接方法早期的研究。[3]对缓冲区管理进行了分析、综述和改进。

基于排序技术的使用最早由[1]提出。[7]和[4]提出了基于散列的连接算法的优点，[4]是混合散列-连接的起源。在并行连接和其他操作符中使用散列已经提出多次了。我们知道的最早的资料是[9]。 784

1. M. W. Blasgen and K. P. Eswaran, "Storage access in relational databases," *IBM Systems J.* **16**:4 (1977), pp. 363–378.

2. S. Chaudhuri, "An overview of query optimization in relational systems," *Proc. Seventeenth Annual ACM Symposium on Principles of Database Systems*, pp. 34–43, June, 1998.

3. H.-T. Chou and D. J. DeWitt, "An evaluation of buffer management strategies for relational database systems," *Proc. Intl. Conf. on Very Large Databases* (1985), pp. 127–141.

4. D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. Wood, "Implementation techniques for main-memory database systems," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1984), pp. 1–8.

5. L. R. Gotlieb, "Computing joins of relations," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1975), pp. 55–63.

6. G. Graefe, "Query evaluation techniques for large databases," *Computing Surveys* **25**:2 (June, 1993), pp. 73–170.

7. M. Kitsuregawa, H. Tanaka, and T. Moto-oka, "Application of hash to data base machine and its architecture," *New Generation Computing* **1**:1 (1983), pp. 66–74.

8. D. Kossman, "The state of the art in distributed query processing," *Computing Surveys* **32**:4 (Dec., 2000), pp. 422–469.

9. D. E. Shaw, "Knowledge-based retrieval on a relational database machine," Ph. D. thesis, Dept. of CS, Stanford Univ. (1980). 785

第16章 查询编译器

在第15章中我们已经知道了执行物理查询计划操作符的基本算法，本章我们来讲解查询编译器及其优化器的体系结构。正如在图15-2中提到的那样，查询处理器必须采取三个主要步骤：

1. 对使用诸如SQL的某种语言书写的查询进行*语法分析*，亦即将查询语句转换成按某种有用方式表示查询语句结构的语法树。

2. 把语法分析树转换成关系代数表达式树（或某种类似标记），我们称之为*逻辑查询计划*。

3. 逻辑查询计划必须转换成*物理查询计划*，物理查询计划不仅指明了要执行的操作，而且也找出了这些操作执行的顺序、执行每步所用的算法、获得所存储数据的方式以及数据从一个操作传递给另一个操作的方式。

第一步，语法分析，是16.1节的主题。这一步的结果是查询语句的一棵语法分析树。另外两步涉及许多选择。在挑选一个逻辑查询计划时，我们有机会应用多个不同的代数操作，其目标是得到最佳的逻辑查询计划。16.2节在理论上讨论关系代数的代数定律。接着在16.3节讲述如何将语法分析树转换成初始的逻辑查询计划，并说明16.2节中的代数定律如何应用到改进初始逻辑查询计划的策略中去。

当从一个逻辑计划产生物理计划时，我们必须估计每个可选方案的预计代价。代价估计本身就是一门科学，我们在16.4节讨论。在16.5节我们讲述如何使用代价估计来评价一个计划。考虑多个关系的连接顺序时会引出许多特殊问题，这是16.6节的话题。最后，16.7节讨论了其他有关选择物理查询计划的各种问题与策略：算法选择以及采用流水线处理还是实体化方法。

16.1 语法分析

查询编译的开始几个阶段如图16-1所示。图中的四个方框对应于图15-2的开始两个阶段。我们在语法分析与转换成初始逻辑查询计划之间分离出“预处理”步骤，该步将在16.1.3节中讨论。

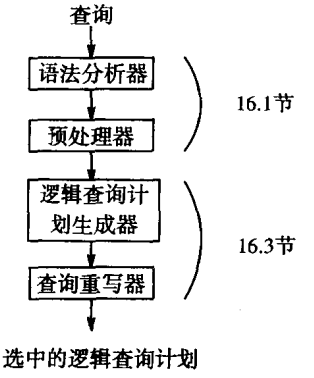


图16-1 从一个查询到一个逻辑查询计划

本节我们讨论SQL的语法分析，并给出可用于该语言的基本语法要素。16.2节我们暂时偏离查询编译这条主线，详尽考察可应用到关系代数表达式的各种定律或变换。在16.3节我们继续查询编译这个主题。首先，我们考察如何将一棵语法树转换成一个关系代数表达式，后者就是初步的逻辑查询计划。接着，我们考察如何应用16.2节中的某些特定变换公式改进查询计划，而不是简单地将一个计划变换成一个等价的却不一定更好的计划。

16.1.1 语法分析与语法分析树

语法分析器的工作是接收用类似SQL这样的语言编写的文本并将之转换成*语法分析树*，该树的结点对应于以下两者之一：

1. *原子*：它们是词法成分，如关键字（如SELECT）、关系或属性的名字、常数、括号、操作符如+或<，以及其他模式成分；或

788 2. *语法类*：即在一个查询中起相似作用的查询子成分所形成族的名称。我们用尖括号将描述性的名称括起来表示语法类。例如，<SFW>用于表示以常用的select-from-where形式的查询，而<Condition>将用于表示属于条件的任何表达式，如那些跟在SQL语句WHERE之后的表达式。

如果结点是一个原子，则该结点没有子女。然而，若该结点是一个语法类，则其子女通过该语言的语法规则之一进行描述。我们将通过例子来说明这些思想。关于如何设计一个语言的语法以及如何进行语法分析，如将一个程序或查询语句转换成语法分析树，这些细节当属编译课程的内容[⊖]。

16.1.2 SQL的一个简单子集的语法

通过给出可用于SQL子集的语言的某些规则，我们借此说明语法分析的过程。同时我们也包含了关于得到完整的SQL语法还需要哪些其他规则的评论。

查询

语法类<Query>用于表示所有正则SQL查询语句。它的一些语法规则是：

```
<Query> ::= <SFW>
<Query> ::= ( <Query> )
```

注意，我们按习惯用::=符号表示“可以表述为”的意思。第一个规则说的是一个查询语句可以是select-from-where的形式；下面我们将进述<SFW>的语法规则。第二个规则说的是一个查询可以是用括号括起的另一个查询。在完整的SQL语法中，我们还需要这样的规则，它允许一个查询是单个关系或是涉及多个关系与各种类型操作的表达式，如UNION或JOIN操作的表达式。

Select-From-Where形式

我们给出语法类<SFW>的一条规则：

789
```
<SFW> ::= SELECT <SelList> FROM <FromList> WHERE <Condition>
```

该规则接受限定形式的SQL查语询句，但不接受其他多种任选子句，如GROUP BY、HAVING或ORDER BY子句，也不接受SELECT之后的DISTINCT选项。这里提醒一下，一个真正的SQL语法包含复杂得多的查询结构，除了包含前面提到的select-from-where的变形外，还包含由UNION、NATURAL JOIN等操作符构造而得的SQL语句以及许多其他的语句。

注意我们的习惯，关键字用大写。语法类<SelList>与<FromList>表示可以分别跟在SELECT与FROM之后的列表。很快我们就会讲述这类列表的受限形式。语法类<Condition>表示SQL条件（那些要么为真要么为假的表达式）；后面我们将给出该语法类的简化规则。

Select列表

```
<SelList> ::= <Attribute> , <SelList>
<SelList> ::= <Attribute>
```

这两条规则说明一个选择列表可为任何由逗号分隔的属性列表：要么是单个属性要么是一个属性、一个逗号以及一个或多个属性的任意列表。注意，在完整的SQL语法中，我们在选择表中还需提供接纳表达式与聚集函数以及属性与表达式别名的规则。

⊖ 对这方面内容不熟悉的读者可以参阅A.V.Aho、R.Sethi和J.D. Ullman的《*Compilers: Principles, Techniques, and Tools*》，Addison-Wesley, Reading MA, 1986。当然16.1.2节的例子用于表示查询处理器中的语法分析是足够了。

From列表

```
<FromList> ::= <Relation> , <FromList>
<FromList> ::= <Relation>
```

这里的from列表可由任意用逗号分隔的关系列表组成。为简便起见，我们省略了from列表元素是表达式的可能性，如不会是R JOIN S，甚至一个select-from-where表达式。类似地，完整的SQL语法应当提供对from列表中的关系取别名的机制；这里我们不允许关系后跟上表示该关系的元组变量名。

条件

我们将使用的规则有：

```
<Condition> ::= <Condition> AND <Condition>
<Condition> ::= <Tuple> IN <Query>
<Condition> ::= <Attribute> = <Attribute>
<Condition> ::= <Attribute> LIKE <Pattern>
```

虽然我们在条件类中列出了比其他语法类更多的语法规则，但是这些规则对于各种形式的条件而言只不过触及皮毛而已。我们省略了涉及下列操作符的规则：OR、NOT、EXISTS、除了等号与LIKE之外的其他比较操作符、常量操作数以及许多其他结构，这些结构在完整的SQL语法中是需要的。此外，虽然一个元组可能有多种形式，但我们只为语法类<Tuple>引入一条规则，意为一个元组可为单个属性：

790

```
<Tuple> ::= <Attribute>
```

基本语法类

语法类<Attribute>、<Relation>和<Pattern>是较为特殊的，因为它们不是通过语法规则定义的，而是通过它们所代表的原子的规则来定义的。例如，在语法分析树中，<Attribute>的一个子女可以是任意的字符串，可解释为查询所针对的任意数据库模式中一个属性的名字。类似地，<Relation>可以被当前模式中作为关系而言的任何有意义的字符串所替代；<Pattern>可以用任何一个用引号括起的字符串替换，其中字符串是一个合法的SQL匹配模式。

例16.1 在语法分析与查询重写阶段，我们的研究主要围绕一个查询语句的两个版本来进行，该查询涉及在电影例子的几个关系：

```
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
```

该查询的两个版本都是询问那些其中至少有一个在1960年出生的影星的电影的名字。我们通过使用LIKE操作符来判定其出生日期(一个SQL字符串)是否以'1960'结尾，从而找出那些出生于1960年的影星。

发起该查询的一种方式是用子查询来构造那些出生于1960年的影星名字的集合，并查看每个StarsIn元组中的starName是否是该子查询返回的集合的一个成员。这个查询版本的SQL语句如图16-2所示。

```
SELECT movieTitle
FROM StarsIn
WHERE starName IN (
    SELECT name
    FROM MovieStar
    WHERE birthdate LIKE '%1960'
);
```

图16-2 找出有出生于1960年影星的电影

按照我们所描绘的语法，图16-2所示查询语句的语法分析树如图16-3所示。根是语法类<Query>，任何一个查询语句的语法树都必然是这

种情况。顺着树往下走，我们可知该查询语句是select-from-where的形式；选择列表仅由属性
791 title构成，from列表只有一个关系StarsIn。

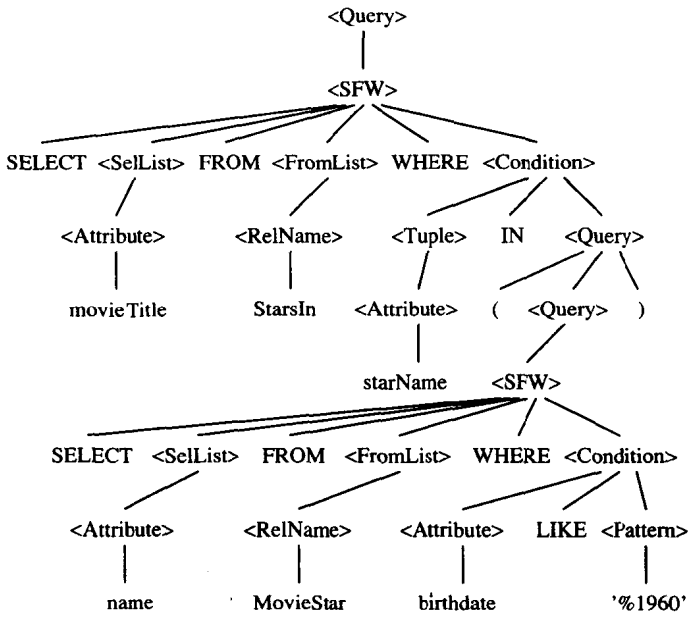


图16-3 图16-2的语法树

在WHERE子句外层的条件比较复杂。它属于“元组-IN-查询”的形式，其中查询是一个用括号括起的子查询，因为在SQL中所有子查询必须用括号括起。该子查询本身又是属于select-from-where形式，有其自身的单个select 列表与from 列表以及含有LIKE操作符的简单条件。□

例16.2 现在我们看看图16-2所示查询的另一个版本，这次我们不使用子查询，而是采用StarsIn与MovieStar两个关系的等值连接，利用条件starName = name来规定两个关系中提到的影星是相同的影星。注意，starName是关系StarsIn的属性，而name是MovieStar的一个属性。图16-2中所示查询的这个版本如图16-4所示[⊖]。

```
SELECT MovieTitle
FROM StarsIn, MovieStar
WHERE starName = name AND
    birthdate LIKE '%1960';
```

图16-4 查询有出生于1960年的影星的电影的另一种方法

对应于图16-4的语法树如图16-5所示。该语法树所使用的许多规则与图16-3相同。不过，请注意在多于一个关系时from 列表在树中是如何表示的，同时注意一个条件可以是用操作符连接起来的多个较小条件的组合，我们所举的例子是用AND
792 连接的。□

16.1.3 预处理器

在图16-1中我们称之为预处理器的部件有多个重要的功能。如果查询语句中用到的关系实际上是一个视图，则在from列表中有用到该关系的地方就必须用描述对应的视图的语法树来替换。这棵语法树由视图的定义得到，本质上就是一个查询语句。

⊖ 这两个查询有一点小小的区别。图16-4所示的查询在一部电影中有一个以上的影星出生于1960年时，会产生重复。严格地说，我们应当把DISTINCT加到图16-4中，但我们所举的语法例子作了简化，省去了该选项。

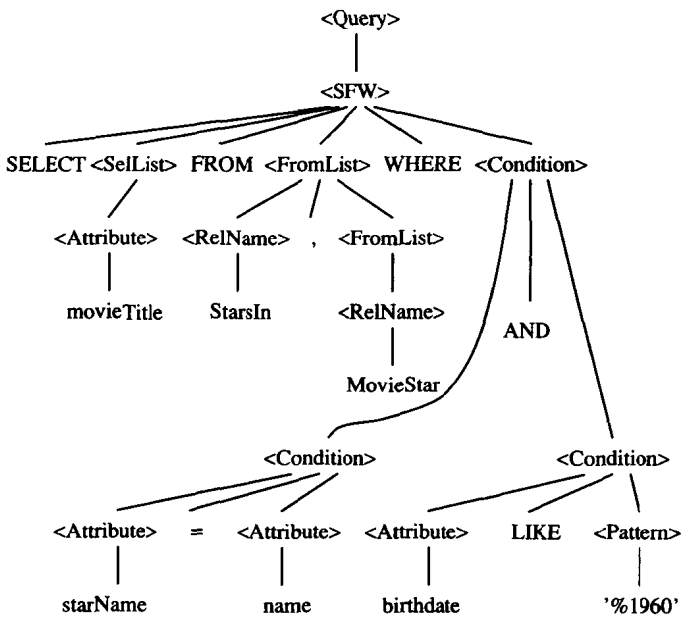


图16-5 图16-4的语法树

预处理器也负责语义检查。即使该查询语句语法上有效，它实际上也可能在名称使用上违反一条或多条语义规则。例如，预处理器必须：

1. 检查关系的使用。FROM子句中出现的关系必须是该查询所对应模式中的关系或视图。例如，预处理器对图16-3所示的语法树进行处理时，将检查from列表中出现的两个关系StarsIn与MovieStar是否在模式中是合法的关系。 793

2. 检查与解析属性的使用。在SELECT子句或WHERE子句中提到的每个属性必须是当前范围中某个关系的属性；如若不然，语法分析器必须报错。例如，在图16-3所示的第一个select列表中的属性title属于仅有的一个关系StarsIn的范围，因此预处理器核实了title属性的使用。如果在查询语句中没有把关系显式地附加到属性上(如，StarsIn.title)，通常查询处理器此时会通过给属性加上它所引用关系的信息来解析(resolve)每一属性。同时也检查二义性，如果某属性属于两个或多个关系，则报错。

3. 检查类型。所有属性的类型必须与其使用相适应。例如，图16-3中的birthdate被用于LIKE比较中，而这种比较要求birthdate是一个字符串或是可被强制转换成字符串的某种类型。由于birthdate是一个日期型，在SQL中日期型通常作字符串处理，因此该属性的使用受到了校验。类似地，要对操作符进行检查，确保它们作用到适当的且相兼容的类型的值上。

如果语法树通过了所有这些检查，它就被认为是合法的。该语法树在进行了视图扩展、属性作用解析后，被传递给逻辑查询计划生成器。如果语法树是非法的，则报告相应的诊断信息，不作进一步处理。

16.1.4 习题

习题16.1.1 增加或修改<SFW>的规则，使其包含以下各种SQL select-from-where表达式的特性的简单版本：

* a) 产生包含DISTINCT关键字的集合的能力。

b) GROUP BY子句和HAVING子句。

c) 用ORDER BY子句对结果排序。

d) 没有where子句的查询语句。

习题16.1.2 给<Condition>增加规则，使其包含以下SQL条件表达式的特征：

* a) 逻辑操作符OR和NOT。

b) 除了‘=’之外的比较操作。

794 c) 带括号的条件。

d) EXISTS表达式。

习题16.1.3 使用本节中所给出的简单的SQL语法，画出关于关系$R(a,b)$与$S(b,c)$查询语句的语法树：

a)
```
SELECT a, c
FROM R, S
WHERE R.b = S.b;
```

b)
```
SELECT a FROM R WHERE b IN
  ( SELECT a FROM R, S WHERE R.b = S.b);
```

16.2 用于改进查询计划的代数定律

我们将在16.3节继续查询编译器的讨论；在那里，首先把语法树转换成一个表达式，该表达式全部或大部分由在5.2到5.4节介绍的扩充关系代数操作符组成。在16.3节，我们将看到如何应用启发式规则，使用关系代数中多个代数定律中的一部分来改进查询语句的代数表达式。作为准备，本节列出一些代数定律，用于将一个表达式树转换成一个等价的表达式树，后者可能有更有效的物理查询计划。

应用这些代数变换式会生成逻辑查询计划，它是查询重写阶段的输出。逻辑查询计划接着被转换成物理查询计划，这个过程中查询优化器要对操作符的实现作一系列决策。物理查询计划的生成将在16.4节开始讲述。有一种方式(在实际中不大采用)是在查询重写阶段产生多个好的逻辑计划，对这些逻辑计划产生的物理计划进行考察，选择出总的最佳物理计划时。

16.2.1 交换律与结合律

用于简化所有类型表达式的最通用的代数定律是交换律和结合律。有关某个操作符的*交换律*是指提供给该操作符的参数的顺序是无关紧要的，其结果总是相同。例如，+与×是算术操作中可交换的操作符。更准确地说，对于任意的数x与y，$x+y=y+x$与$x \times y=y \times x$成立。但是，“-”不是一个可交换的操作符：$x-y \neq y-x$。

一个操作符的*结合律*说的是该操作符出现的两个地方既可以从左边进行组合也可以从右边进行组合。举例说，+与×是满足结合律的操作符，意味着$(x+y)+z=x+(y+z)$和$(x \times y) \times z=x \times (y \times z)$成立。相反，“-”操作符则不满足结合律：$(x-y)-z \neq x-(y-z)$。当一个操作符既满足结合律又满足交换律时，我们可以对用这个操作符连接起来的任意多个操作数进行随意组合与排

795 列，而不会改变结果。例如，$((w+x)+y)+z=(y+x)+(z+w)$。

关系代数的多个操作符同时满足结合律与交换律。尤其是：

- $R \times S = S \times R; (R \times S) \times T = R \times (S \times T)$
- $R \bowtie S = S \bowtie R; (R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
- $R \cup S = S \cup R; (R \cup S) \cup T = R \cup (S \cup T)$

- $R \cap S = S \cap R$; $(R \cap S) \cap T = R \cap (S \cap T)$

注意，以上定律中关于并和交的定律对于集合与包也是成立的。

我们不对每一个定律都一一加以证明，不过在下面给出了一个定律的证明。证明含有关系的代数定律总的方法是要证明左边的表达式产生的每个元组也可由右边的表达式可产生的，同时右边表达式产生的每个元组左边表达式也能产生。

例16.3 我们来证明$\bowtie$的交换律：$R \bowtie S = S \bowtie R$。

首先假设元组t在$R \bowtie S$的结果中，即在左边的表达式中，则必存在r属于R、s属于S，它们在t的公共属性上值相同。因此，当我们计算右边的表达式$S \bowtie R$时，元组s与r又会组合形成t。

我们可能会想像 t 的各分量在左边公式与右边公式中的顺序是不同的，但在形式上，关系代数的元组没有固定不变的属性次序。相反，我们可以自由地对元组属性重新排列，只要如3.1.5节所述，在列标题中有相应的属性名即可。

我们还没有证完。由于我们的关系代数是一个包的代数，而不是集合的代数，我们必须证明，如果t在左边出现了n次；则t在右边也出现n次，反过来，若t在右边出现n次，则t在左边也至少出现n次。假设t在左边出现了n次，那么与t相对应来自关系R的元组r必须出现某个值n_R次，与t相对应来自S的元组s出现n_S次，其中$n_R\ n_S = n$。当我们计算右边表达式$S \bowtie R$时，应有s出现n_S次，r出现n_R次，从而得到$n_R\ n_S$份t的拷贝，即有n个t元组拷贝。

我们仍没有证完。我们已完成了证明的一半，即左边出现的每个元组也出现在右边，但我们还必须证明出现在右边的每个元组也出现在左边。由于明显的对称性，论证本质上是一样的，在这里恕不赘述。 □

我们未把θ-连接包括在满足结合－交换律的操作符中。该操作符满足交换律：

- $R \overset{\bowtie}{_C} S = S \overset{\bowtie}{_C} R$

此外，如果条件所在位置是有意义的，则θ连接也满足结合律。但是，正如下面例子所指出的那样，在某些情况下，结合律不成立，因为条件不能作用到参与连接的属性上去。 796

包与集合的定律可能不同

在把我们所熟悉的有关集合的定律应用到属于包的关系上时，应当小心。例如，你可能已经学过集合论中的定律如$A \cap_S (B \cup_S C) = (A \cap_S B) \cup_S (A \cap_S C)$，它就是交对并的分配律。该定律对集合成立，而对包不成立。

举例来说，假设包 A、B、C 均为$\{x\}$，则$A \cap_B (B \cup_B C) = \{x\} \cap_B \{x, x\} = \{x\}$。然而$(A \cap_B B) \cup_B (A \cap_B C) = \{x\} \cup_b \{x\} = \{x,x\}$，与左边的结果$\{x\}$是不同的。

例16.4 假设有三个关系$R(a, b)$，$S(b, c)$，$T(c, d)$。表达式

$$(R \overset{\bowtie}{_{R.b>S.b}} S) \overset{\bowtie}{_{a<d}} T$$

可按假设的结合律转换成：

$$R \overset{\bowtie}{_{R.b>S.b}} (S \overset{\bowtie}{_{a<d}} T)$$

然而，我们不能使用条件$a<d$对S与T进行连接，因为a既不是S的属性也不是T的属性。所

以θ连接的结合律不可随意使用。 □

16.2.2 涉及选择的定律

从查询优化的观点来看，选择是一个关键的操作。由于选择可以明显地减少关系的大小，进行有效查询处理最重要的规则之一就是只要不改变表达式的结果，就把选择在语法树上尽可能地下移。确实，早期的查询优化器将这个变换的各种变种形式作为其选择良好逻辑计划的首要策略。正如我们稍后指出的那样，“在语法树上下推选择”这个变换并不十分通用，但“下推选择”的思想仍是查询优化的一个主要手段。

本节我们研究有关σ操作符的定律。作为开始，当选择条件较复杂（如涉及用AND或OR连接起来的条件）时，将条件分解为其组成部分是有帮助的。其动机是部分条件涉及的属性比整个条件少，可移到某个方便的地方，而整个条件则不一定能够移入。因此，有关σ 操作符的头两条定律是分解律：

- $\sigma_{C_1 \text{ AND } C_2}(R) = \sigma_{C_1}(\sigma_{C_2}(R))$

797

- $\sigma_{C_1 \text{ OR } C_2}(R) = (\sigma_{C_1}(R)) \cup_S (\sigma_{C_2}(R))$

不过，第二条定律只有在 *OR* 为集合时成立。注意，如果 R 为包，集合的并可能会错误地消除重复。

注意，C_1与C_2的顺序是灵活的。例如，我们还可将上面的第一条定律写成C_2作用在C_1之后，$\sigma_{C2}(\sigma_{C1}(R))$。事实上，更一般地，我们可以任意交换$\sigma$操作符的顺序：

- $\sigma_{C_1}(\sigma_{C_2}(R)) = \sigma_{C_2}(\sigma_{C_1}(R))$

例16.5 令$R(a,b,c)$是一关系，则$\sigma_{(a=1 \text{ OR } a=3) \text{ AND } b<c}(R)$可分解为$\sigma_{a=1 \text{ OR } a=3}(\sigma_{b<c}(R))$。接着我们又可将这个表达式在OR处分解为$\sigma_{a=1}(\sigma_{b<c}(R)) \cup \sigma_{a=3}(\sigma_{b<c}(R))$。在这种情形下，由于一个元组不可能同时满足 $a = 1$与 $a = 3$，不论 R 是否是一个集合，只要$\cup_B$用作并，该变换总是成立的。然而，总的来说，对OR的分解要求其参数是集合并且使用$\cup_S$。

另一种分解方法是把 $\sigma_{b<c}$ 作为外层操作，即$\sigma_{b<c}(\sigma_{a=1 \text{ OR } a=3}(R))$。然后对OR进行分解，得到$\sigma_{b<c}(\sigma_{a=1}(R) \cup \sigma_{a=3}(R))$，它与我们得到的第一个表达式等价，但略有不同。 □

涉及 σ 的另一类定律允许我们对二元操作符进行下推选择：积、并、交、差和连接。有三种类型的定律，这取决于下推选择到每个参数是可选的还是必需的：

1. 对于并，选择必须下推到两个参数中。
2. 对于差，选择必须下推到第一个参数，下推到第二个参数是可选的。
3. 对于其他操作符，只要求选择下推到其中一个参数。对于连接和积，将选择下推到两个参数是没有意义的，因为参数可能有也可能没有选择所要的属性。即使可以同时下推到两个参数，该做法也不一定能改进计划：参见习题16.2.1。

因此，对于并的定律是：

- $\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$

这里，将选择移入语法树的两个分枝是必须的。

对于差，定律可以写成：

- $\sigma_C(R - S) = \sigma_C(R) - S$

798

当然，把选择下推到两参数上也是允许的：

- $\sigma_C(R - S) = \sigma_C(R) - \sigma_C(S)$

下面这些定律允许将选择下推到一个或两个参数中。对于选择σ_C，我们只能将其下推到一个包含C中涉及的全部属性的关系中，如果此关系存在的话。假设关系R具有C中提及的全部属性，我们列出如下定律：

- $\sigma_C(R \times S) = \sigma_C(R) \times S$
- $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$
- $\sigma_C(R \overset{\bowtie}{_D} S) = \sigma_C(R) \overset{\bowtie}{_D} S$
- $\sigma_C(R \cap S) = \sigma_C(R) \cap S$

如果C只涉及S的属性，则有：

- $\sigma_C(R \times S) = R \times \sigma_C(S)$

其他三个操作符$\bowtie$、$\overset{\bowtie}{_D}$和$\cap$也类似。如果关系R与S恰好都包含了C的属性，则可使用诸如下面所列的定律：

- $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie \sigma_C(S)$

注意，如果操作符是$\times$或$\overset{\bowtie}{_D}$，则不能应用这个定律的变体，因为在这种情形下R与S没有公共的属性。但是，对于$\cap$而言，这个定律总是适用的，因为此时R与S的模式必须相同。

例16.6 考虑关系$R(a,b)$与$S(b,c)$以及表达式

$$\sigma_{(a=1 \text{ OR } a=3) \text{ AND } b<c}(R \bowtie S)$$

条件$b < c$只能用到S上，而条件$a = 1$与$a = 3$只能用到R上。这样我们可像例16.5的第一种可供选择的方法那样由分解两个条件的AND开始：

$$\sigma_{a=1 \text{ OR } a=3}\left(\sigma_{b<c}(R \bowtie S)\right)$$

接着，我们可把选择$\sigma_{b<c}$下推到S，得到表达式：

$$\sigma_{a=1 \text{ OR } a=3}\left(R \bowtie \sigma_{b<c}(S)\right)$$

最后，我们下推第一个条件到R，得到：$\sigma_{a=1 \text{ OR } a=3}(R) \bowtie \sigma_{b<c}(S)$。我们还可如例16.5那样对两个条件中的OR进行分解，但这样做可能有好处也可能没有好处。 □ 799

一些平凡的定律

我们不去陈述关系代数的每一个经过证明的定律。读者应当小心对待，尤其是对于那些针对极端情形的定律：空关系、条件总为真或总为假的选择或θ连接、包含全部属性的投影等。举例来说，下面是许多可能的特殊情形定律中的一部分：

- 任何对空关系的选择为空。
- 如果C是总为真的条件(如：x>10 OR $x \leqslant 10$应用到不允许 x=NULL的关系上)，则$\sigma_C(R)=R$。
- 如果R为空，则$R \cup S=S$。

16.2.3 下推选择

正如我们提到的那样，在表达式树中下推选择——即用16.2.2节所述的一条规则的右边表

达式替换其左边的表达式——是查询优化器最强有力的工具。很久以来一直假定，通过应用σ的定律我们可以在那个方向上进行优化。然而，当支持视图的系统变得普遍时，发现在某些情况下首先选择尽可能往树的上部移是很重要的，然后再把选择下推到所有可能的分枝。用一个例子来说明如何适当地移动选择。

例16.7 假设我们有如下关系：

```
StarsIn(title, year, starName)
Movie(title, year, length, inColor, studioNameProducerC#)
```

注意，为了便于进一步讨论本例，这里修改了Movie的前两个属性movieTitle和movieYear。定义视图：

```
CREATE VIEW MoviesOf1996 AS
    SELECT *
    FROM Movie
    WHERE movieYear = 1996;
```

800 我们可用SQL查询“在1996年有哪些影星为哪些电影制作公司工作？”：

```
SELECT starName, studioName
FROM MoviesOf1996 NATURAL JOIN StarsIn;
```

视图MoviesOf1996用关系代数表达式定义为：

$$\sigma_{year=1996}(\text{Movie})$$

因此，以上查询是这个表达式与StarsIn的自然连接，然后在属性starName与studioName上投影，它具有如图16-6所示的表达式或“逻辑查询计划”。

在这个表达式中，仅有的选择已经位于它可移入的树的最下面，因此已无法“在树中下推选择”了。然而，规则$\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$可以“反过来”用，把选择$\sigma_{year=1996}$放到图16-6的连接之上。然后，由于*year*是Movie与StarsIn两者的属性，我们可以把选择下推到连接结点的两个子结点。所得的逻辑查询计划如图16-7所示。它很可能得到了改进，因为在StarsIn与1996年的电影连接之前已减小了关系

801 StarsIn的大小。 □

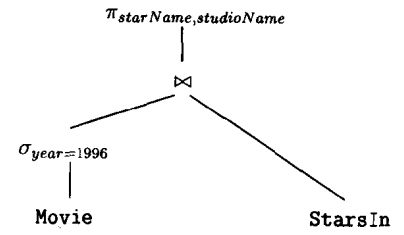


图16-6 由查询与视图的定义构造的逻辑查询计划

16.2.4 涉及投影的定律

投影可以像选择一样下推到多个其他操作符中。下推投影与下推选择不同，当下推投影时，投影通常留在原处。换句话说，“下推”投影确实可能需要在一个已存在的投影下的某个地方引入一个新的投影。

下推投影是有用的，但一般而言不如下推选择那么有用。原因是选择通常以较大的因子减小关系的大小，投影不改变元组数，只缩短元组的长度。事实上，5.4.5节的外投影操作实际上增加了元组的长度。

为描述外投影的转换，我们需要引入一些术语。考

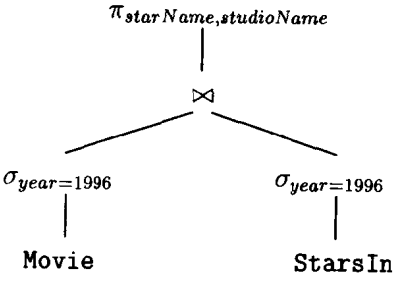


图16-7 通过在树中上移和下移选择来改进查询计划

虑投影列表中的项$E \to x$，其中E是一个属性或含有属性与常量的一个表达式。我们称E中提到的全部属性是投影的输入属性，x是一个输出属性。若一个项是单个属性的，则它既是输入属性又是输出属性。注意，不可能存在不是单个属性的没有箭头与重命名的表达式，因此我们已经覆盖所有的情形。

如果一个投影列表的属性构成不包含更名或不是单个属性的表达式，则我们称该投影是简单的。在严谨的关系代数中，所有投影都是简单的。

例16.8 投影$\pi_{a,b,c}(R)$是简单的；a、b和c既是其输入属性又是其输出属性。但是$\pi_{a+b\to x,c}(R)$不是简单的。其输入属性是a、b和c，而其输出属性是x和c。 □

投影定律之后隐藏的原理是：

- 我们可以在表达式树上的任何地方引入投影，只要它所消除的属性是其上的操作符从来不会用到的，并且也不在整个表达式的结果之中。

在这些定律的最基本形式中，所引入的投影总是简单的，虽然其他的一些投影可能不是简单的，如下面的人：

- $\pi_L(R \bowtie S) = \pi_L\big(\pi_M(R) \bowtie \pi_N(S)\big)$，其中$M$是$R$的所有属性列表，或者是连接属性(同属于$R$与$S$的模式中)，或者是$L$的输入属性；$N$是$S$的属性列表，属于连接属性或$L$的输入属性。
- $\pi_L(R \overset{\bowtie}{C} S) = \pi_L\big(\pi_M(R) \overset{\bowtie}{C} \pi_N(S)\big)$，其中$M$是$R$的属性列表，或者是连接属性(如在条件$C$中提到的)，或是$L$的输入属性；$N$是$S$的属性列表，要么是连接属性，要么是$L$的输入属性。 802
- $\pi_L(R \times S) = \pi_L\big(\pi_M(R) \times \pi_N(S)\big)$，其中$M$、$N$分别是$R$与$S$的全部属性列表，且是$L$的输入属性。

例16.9 令$R(a,b,c)$与$S(c,d,e)$是两个关系。考虑表达式$\pi_{a+e\to x,b\to y}(R \bowtie S)$。投影的输入属性是$a$、$b$和$e$，而$c$是仅有的连接属性。我们可以应用下推投影到连接之下的定律得到以下等价表达式：

$$\pi_{a+e\to x,\ b\to y}\big(\pi_{a,b,c}(R) \bowtie \pi_{c,e}(S)\big)$$

注意，投影$\pi_{a,b,c}(R)$是平凡的；它是对R全部属性的投影。我们可以消除这个投影，得到第三个等价表达式：$\pi_{a+e\to x,b\to y}(R \bowtie \pi_{c,e}(S))$。也就是说，与原始表达式相比，仅有的变化是我们在连接之前把属性d从S移走了。 □

此外，我们可在包并之前进行投影。即：

- $\pi_L(R \cup_B S) = \pi_L(R) \cup_B \pi_L(S)$

另一方面，投影也不能被推到集合并或集合、包的交或差之下。

例16.10 令$R(a,b)$由一个元组$\{(1,2)\}$组成，$S(a, b)$由一个元组$\{(1, 3)\}$组成。则$\pi_a(R \cap S) = \pi_a(\phi) = \phi$。然而，$\pi_a(R) \cap \pi_a(S) = \{(1)\} \cap \{(1)\} = \{(1)\}$。 □

如果投影涉及一些计算，并且投影列表中某一项的输入属性全部属于投影下面连接或积之中的某一参数，则我们可选择直接在那个参数上进行计算，尽管这不是必需的。下面的例子有助于说明这一点。

例16.11 再令$R(a, b, c)$与$S(c,d,e)$是关系，考虑连接与投影$\pi_{a+b\to x,\ d+e\to y}(R \bowtie S)$。我们可以把和$a+b$及其新名$x$直接移到关系$R$上，类似地，把和$d+e$移到$S$上。所得到的等价表达式

是$\pi_{x,y}(\pi_{a+b\to x,\ c}(R) \bowtie \pi_{d+e\to y,\ c}(S))$。

要处理的一种特别的情形是如果x或y是c。此时，我们不能将求和式更名为c，因为一个关系不能同时有两个同名为c的属性。因此，我们必须采用一个临时的名字，然后在连接上的投影中再做一次更名。例如，$\pi_{a+b\to c,\ d+e\to y}(R \bowtie S)$可变为$\pi_{z\to c,\ y}(\pi_{a+b\to z,\ c}(R) \bowtie \pi_{d+e\to y,\ c}(S))$。 □

803 下推投影到选择之下也是可能的。

- $\pi_L(\sigma_C(R)) = \pi_L(\sigma_C(\pi_M(R)))$，其中$M$是$L$的输入属性列表或条件$C$中提到的属性列表。

像在例16.11中一样，我们可以将在列表L上的计算移到列表M上去，只要条件C不需要那些包含在计算中的L的输入属性即可。

通常，我们希望把投影下推到表达式树中，即使在上面还必须保留另一个投影，因为投影会减小元组的大小从而减少中间关系所占用的块数。然而，这样做时，我们必须小心，因为有一些普通的例子说明下推投影是要花时间的。

例16.12 考虑询问那些在1996年工作过的影星的查询：

```
SELECT starName
FROM StarsIn
WHERE movieYear = 1996;
```

是关于关系StarsIn(movieTitle, movieYear, starName)的查询。由该查询直接翻译而得到的逻辑查询计划如图16-8所示。

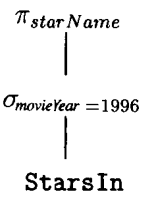


图16-8 例16.12查询的逻辑查询计划

我们可以在选择之下增加一个投影到下列属性上：

1. starName，因为该属性在结果中是需要的；和
2. movieYear，因为该属性在选择条件中被用到。

该结果如图16-9所示。

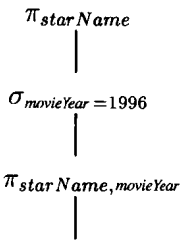


图16-9 引入投影后的结果

如果StarsIn不是一个被保存的关系，而是用另一个操作构造而得的关系，如连接操作，则图16-9所示的计划是有意义的。当连接产生元组时，可简单地抛弃无用的title属性，将投影作“流水线”处理（参见16.7.3节）。

但是，在本例情形中，StarsIn是一个被保存的关系。在图16-9中较低处的投影可能会浪费许多时间，尤其是当在movieYear上存在索引时，则基于图16-8的逻辑查询计划而产生的物理查询计划将首先使用索引获取那些movieYear
804 等于1996的StarsIn中的元组，可能只是一小部分元组。如果我们像图16-9那样先做投影，则必须从StarsIn中读取每一个元组并进行投影。

更糟的是，在被投影的关系$\pi_{starName,\ movieYear}$(StarsIn)上，movieYear上的索引很可能是没有用处的，因此选择需要对投影产生的全部元组进行扫描。 □

16.2.5 有关连接与积的定律

我们在16.2.1节中已看到许多有关连接与积的重要定律：交换律与结合律。当然还有另外一些定律直接来自连接的定义，如5.2.10节所述：

- $R \overset{\bowtie}{_C} S = \sigma_C(R \times S)$。
- $R \bowtie S = \pi_L(\sigma_C(R \times S))$，其中有相同名字的属性对进行等值比较，$L$是包含$R$与$S$中等值对的其中一个属性以及其余属性的列表。

在实际中，我们通常想从右到左使用这些规则。即我们把后面跟着选择的积认为是某个连接。这样做的原因是计算连接的算法通常比计算积之后跟着一个对（很大的）结果进行选择的算法要快得多。

16.2.6 有关消除重复的定律

操作符δ用于从包中消除重复，它可下推到多个操作符中，但不是全部操作符。一般而言，将δ移到树的下边减小了中间关系的大小，因而可能是有好处的。此外，我们有时会把δ移到一个可以完全消除的位置，因为它作用于一个不含重复元组的关系上。

- 若R没有重复，则$\delta(R) = R$。此种关系R会有以下几种重要情形： 805

a) 一个声明了主键的存储关系；以及

b) 属于γ操作结果的一个关系，因为分组创建一个没有重复的关系。

在其他操作符中“下推”δ的几个定律如下：

- $\delta(R \times S) = \delta(R) \times \delta(S)$
- $\delta(R \bowtie S) = \delta(R) \bowtie \delta(S)$
- $\delta(R \overset{\bowtie}{_C} S) = \delta(R) \overset{\bowtie}{_C} \delta(S)$
- $\delta(\sigma_C(R)) = \sigma_C(\delta(R))$

我们也可以把δ移到交操作中的一个或两个参数上：

- $\delta(R \cap_B S) = \delta(R) \cap_B S = R \cap_B \delta(S) = \delta(R) \cap_B \delta(S)$

还有，一般情况下，δ不能移入$\cup_B$，$-_B$或π等操作符中。

例16.13 令R有元组t的两个拷贝，S有t的一个拷贝，则$\delta(R \cup_B S)$有t的一个拷贝，而$\delta(R) \cup_B \delta(S)$有$t$的两份拷贝。同时，$\delta(R -_B S)$有$t$的一份拷贝，而$\delta(R) -_B \delta(S)$不含$t$的拷贝。

现在考虑关系$T(a,b)$，含有元组(1，2)与(1，3)各一份拷贝，此外不再有其他元组。则$\delta(\pi_a(T))$有元组(1)的一份拷贝，而$\pi_a(\delta(T))$有元组(1)的两份拷贝。 □

最后，注意δ与$\cup_S$、$\cap_S$或$-_S$交换没有意义。因为产生一个集合是保证没有重复元组的一种方法，从而可以去除δ。例如：

- $\delta(R \cup_S S) = R \cup_S S$

然而，注意$\cup_S$或其他集合操作符的实现涉及一个消除重复过程，这相当于应用δ。有关例子参看15.2.3节。

16.2.7 涉及分组与聚集的定律

考虑操作符γ时，我们发现很多变换的适用性取决于所用聚集操作符的细节。因此，我们不能像对待其他操作符定律那样陈述它的通用定律。其中的一个例外是在16.2.6节中提到的定律，即γ吸收δ。准确地说：

- $\delta(\gamma_L(R)) = \gamma_L(R)$ 806

另一个通用规则是：在应用γ操作符之前，只要我们需要，就可以利用投影除去参数中无用的属性。该定律可写为：

• $\gamma_L(R) = \gamma_L(\pi_M(R))$，其中 M 是在 L 中提到的那些 R 中的属性列表。

其他变换依赖于 γ 操作符中的聚集的原因是某些聚集——尤其是MIN与MAX——不受重复是否存在的影响。而另一些聚集——如SUM、COUNT和AVG——如果在计算聚集之前消除重复，一般会得到不同的值。

因此，我们称操作符γ_L是不受重复影响的，如果L中仅有的聚集是MIN与/或MAX。则有：

• $\gamma_L(R) = \gamma_L(\delta(R))$，成立的条件是 γ_L 不受重复影响。

例16.14 假设我们有如下关系

```
MovieStar(name, addr, gender, birthdate)
StarsIn(movieTitle, movieYear, starName)
```

我们想知道那些出品于给定年份的电影中最年轻影星的生日。我们将这个查询表达为：

```
SELECT movieYear, MAX(birthdate)
FROM MovieStar, StarsIn
WHERE name = starName
GROUP BY movieYear;
```

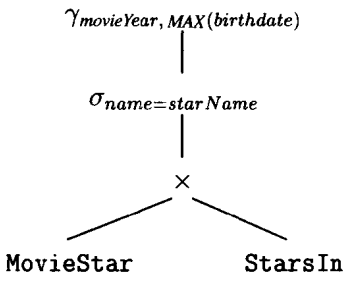


图16-10 例16.14查询的初步逻辑查询计划

直接由查询语句构造而得的初步逻辑查询计划如图16-10所示。FROM列表是一个积，WHERE子句是积之上的一个选择。分组与聚集是用在它们之上的γ操作符表示的。如
807 果我们愿意，可应用到图16-10之上的变换是：

1. 将选择与积组合成一个等值连接。
2. 在γ之下引入 δ，因为 γ是不受重复影响的。
3. 在γ与所引入的δ之间引入 π，投影到movieYear与birthdate上，即与 γ 相关的仅有的两个属性。

所得的计划如图16-11所示。

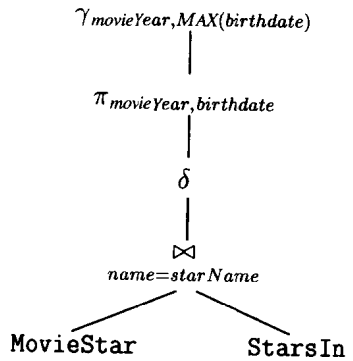


图16-11 例16.14查询语句的另一个查询计划

现在我们可以下推 δ 到⋈之下，并且如果需要，可在 δ 之下引入π。新得到的查询计划如
808 图16-12所示。如果name是MovieStar的码，则延伸到这个关系的分枝上的 δ 可以去除。 □

16.2.8 习题

* **习题16.2.1** 当可以下推选择到二元操作符的两个参数上时，我们需要决定是否这样做。其中某个参数存在索引时是否需要下推选择？例如，考虑表达式$\sigma_C(R \cap S)$，其中S上有一个索引。

习题16.2.2 给出例子证明：

* a) 投影不能下推到集合并之下。

b) 投影不能下推到集合差或包差之下。

c) 消除重复(δ)不能下推到投影之下。

d) 消除重复不能下推到包并或包差之下。

! **习题16.2.3** 证明我们总是可以下推投影到包并的两个分枝之下。

! **习题16.2.4** 某些集合的定律也适用于包；某些则不然。下面的每条定律对于集合都是正确的，判定对包是否也正确。对包是正确的定律给出证明，或给出一个反例。

* a) $R \cup R = R$ (并的幂等律)

b) $R \cap R = R$ (交的幂等律)

c) $R - R = \phi$

d)$R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$(并对交的分配律)

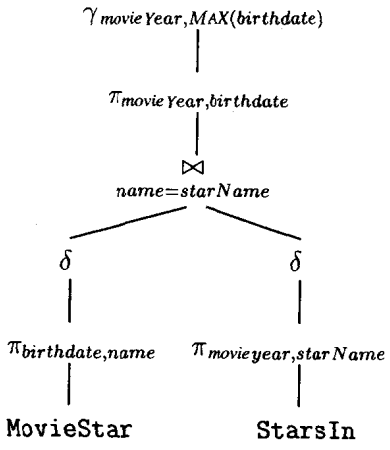


图16-12 例16.14的第三个查询计划

! **习题16.2.5** 我们可按以下方式定义包的$\subseteq$：$R \subseteq S$ 当且仅当对于每个元素x，x在R中出现的次数少于或等于它在S中出现的次数。判定以下陈述(对于集合全真)对于包是否全真；给出证明或反例：

a) 若 $R \subseteq S$，则$R \cup S = S$

b) 若 $R \subseteq S$，则$R \cap S = R$

c) 若 $R \subseteq S$ 且$S \subseteq R$，则$R = S$

习题16.2.6 针对表达式$\pi_L(R(a,b,c) \bowtie S(b,c,d,e))$，尽可能下推投影，其中$L$是：

* a) $b+c \rightarrow x$，$c+d \rightarrow y$

b) $a,b,a+d \rightarrow z$

809

! **习题16.2.7** 我们曾在例16.14中提到，我们所给出的计划没有一个是必然最佳的计划。你能想出一个较好的计划吗？

! **习题16.2.8** 下面是有关关系 $R(a,b)$ 操作的一些等式。判定它们是否为真；给出证明或举出反例。

a) $\gamma_{MIN(a)\rightarrow y,\ x}(\gamma_{a,\ SUM(b)\rightarrow x}(R)) = \gamma_{y,SUM(b)\rightarrow x}(\gamma_{MIN(a)\rightarrow y,\ b}(R))$

b) $\gamma_{MIN(a)\rightarrow y,\ x}(\gamma_{a,\ MAX(b)\rightarrow x}(R)) = \gamma_{y,MAX(b)\rightarrow x}(\gamma_{MIN(a)\rightarrow y,\ b}(R))$

!! **习题16.2.9** 习题15.2.4中类似连接的操作符服从某些熟知的定律，其他的操作符则不然。判定下面每一个式子是否成立。对于成立的定律给出证明，否则给出反例。

* a) $\sigma_C(R \ltimes S) = \sigma_C(R) \ltimes S$

* b) $\sigma_C(R \overset{\circ}{\bowtie} S) = \sigma_C(R) \overset{\circ}{\bowtie} S$

c) $\sigma_C(R \overset{\circ}{\bowtie}_L S) = \sigma_C(R) \overset{\circ}{\bowtie}_L S$，其中$C$只包含$R$的属性

d) $\sigma_C(R \overset{\circ}{\bowtie}_L S) = R \overset{\circ}{\bowtie}_L \sigma_C(S)$，其中$C$只包含$S$的属性

e) $\pi_L(R \overline{\ltimes} S) = \pi_L(R) \overline{\ltimes} S$

* f) $(R \overset{\circ}{\bowtie} S) \overset{\circ}{\bowtie} T = R \overset{\circ}{\bowtie} (S \overset{\circ}{\bowtie} T)$

g) $R \overset{\circ}{\bowtie} S = S \overset{\circ}{\bowtie} R$

h) $R \overset{\circ}{\bowtie}_L S = S \overset{\circ}{\bowtie}_L R$

i) $R \ltimes S = S \ltimes R$

16.3 从语法分析树到逻辑查询计划

我们现在继续查询编译器的讨论。在16.1节中已经构造了一个查询语句的一棵分析树，下一步我们需要把分析树转换成所希望的逻辑查询计划。这需要分两步，正如图16-1中所提示那样。

第一步，按适当的组用一个或多个关系代数操作符替换分析树上的结点与结构。我们将提示这些规则中的某一些，其余的一些留作练习。第二步，利用第一步中产生的关系代数表达式，
810 将其转换成我们所期望的一个表达式，它可转换成最有效的物理查询计划。

16.3.1 转换成关系代数

现在我们非正式地说明将SQL分析树转换成代数逻辑查询计划的一些规则。第一条规则，可能也是最重要的，使我们能够直接将所有“简单的”select-from-where结构转换成关系代数。规则可非正式地陈述为：

- 如果有一个属于<SFW>成分的<Query>，并且该成分中的<Condition>没有子查询，则可以用一个关系代数表达式来替换整个成分——select列表、from列表以及条件，其中代数表达式自底向上由以下内容组成：
  1. <FromList>中提及的全部关系的积是以下操作符的参数：
  2. 选择σ_C，其中C就是要被替换成份中的<Condition>表达式，同时，选择又是下面操作符的参数：
  3. 投影π_L，其中L是<SelList>中的属性列表。

例16.15 让我们考虑如图16-5所示的分析树。select-from-where变换应用到图16-5的整棵分析树上。我们取from列表中两个关系StarsIn与MovieStar的积，用根为<Condition>的子树中的条件作选择，并投影到select列表title上。所得关系代数表达式如图16-13所示。

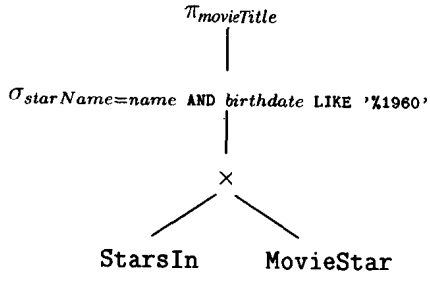


图16-13 把分析树转换成代数表达式树

同样的变换不能应用到图16-3所示的外层查询上。原因是条件中包含一个子查询。我们将在16.3.2节中讨论如何处理带有子查询的条件，你应当查看有关“选择条件的限制”的文本框以得到为什么对有子查询与无子查询的条件加以区别的解释。

然而，我们可以将 select-from-where 规则应用到图16-3所示的子查询上。我们从子查询得到的关系代数表达式是$\pi_{name}(\sigma_{birthdate \text{ LIKE } '\%1960'}$
811 (MovieStar))。 □

16.3.2 从条件中去除子查询

对于<Condition>中包含子查询的分析树，我们将引入操作符的中间形式，它介于分析树的分析类与作用到关系上的关系代数操作符之间。该操作符通常称为两参数选择。我们将用不带参数的标记为 σ 的结点表示经转换后的分析树中的两参数选择。该结点之下有一个左子结点，

它表示要对其做选择操作的关系R；以及一个右子结点，它表示作用到关系R的每个元组上的条件表达式。两个参数均可表示为分析树、表达式树或两者的混合。

例16.16 图16-14所示的是图16-3使用两参数选择对分析树的重写。由图16-3构造图16-14时进行了多种变换：

1. 图16-3中的子查询被一个关系代数表达式所替换，已在例16.15的末尾讨论过。

2. 外层查询也已经用16.3.1节的select-from-where表达式规则做了替换。不过，我们已将必需的选择表示为两参数选择，而不是常规的关系代数操作符 σ。因此，标有<Condition>的分析树的上层结点没有被替换，而仍旧作为选择的一个参数，其表达式的某部分按第1点被关系代数替换。 812

这棵树需要作进一步转换，在后面讨论。 □

我们需要能够用单参数选择与其他关系代数操作符来替换两参数选择的规则。每种条件形式需要其自身的规则。在通常情况中，去除两参数选择并得到纯关系代数表达式是可能的。然而，在一些极端情形下，两参数选择可任其原样保留，并认为是逻辑查询计划的一部分。

选择条件的限制

有人可能感到疑惑，为什么我们不允许选择操作符σ_C中的C包含子查询，在关系代数中，一个操作符的变元(argument)——不在下标中出现的成分——习惯上可以产生关系的表达式。另一方面，参数(parameter)——出现在下标中的成分——不同于关系，它具有类型。例如，σ_C中的参数C是布尔值条件，π_L中的参数L是属性列表或公式。

如果我们遵循这个习惯，则一个参数所隐含的无论是什么计算均可应用到关系自变量的每一个元组上。这种对参数使用的限制简化了查询优化。作为对比，假定我们可有像$\sigma_C(R)$的一个操作符，其中C包含一个子查询。则将C应用到R的每个元组涉及计算子查询。我们要为R的每个元组重新计算它吗？那将是不必要的巨大开销，除非该子查询是相关的，例如其值依赖于查询语句之外所定义的某些值，如图16-3的子查询依赖于starName的值。大部分情形下，即便是相关子查询也无需对每个元组重新计算，条件是我们正确地组织计算。

我们将举例给出处理图16-14中涉及IN操作符条件的规则。注意，这个条件中的子查询是不相关的，即该关系可以只计算一次，与要被检测的元组无关。消除这种条件的规则可做如下非正式陈述：

- 假设我们有一个两参数选择，其中第一个参数代表某个关系R，第二个参数是一个形如t IN S的<Condition>，其中表达式S是一个非相关子查询，t是R的(某些)属性组成的一个元组。我们按如下方式对树作变换：

a) 用S的表达式的树替换<Condition>。如果S有重复，则在S的表达式的根部有必要包

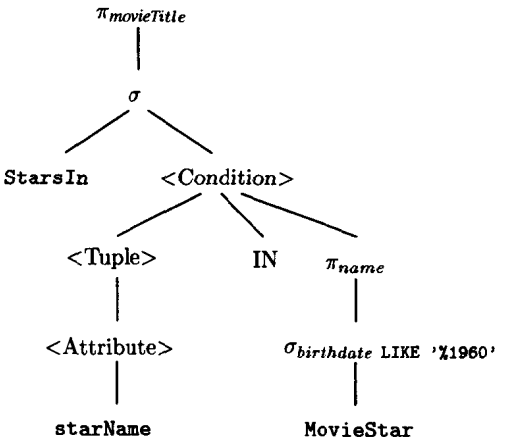


图16-14 使用两参数 σ 的表达式，介于分析树与关系代数之间

813 含一个δ操作，因此所形成的表达式所产生的元组拷贝不会多于原始查询所产生的。

b) 用一个单参数选择σ_C替换两参数选择，其中C是元组t的每个分量与关系S中相应的属性取等值的条件。

c) 给σ_C一个参数，它是R与S的积。

图16-15描绘了这个转换。

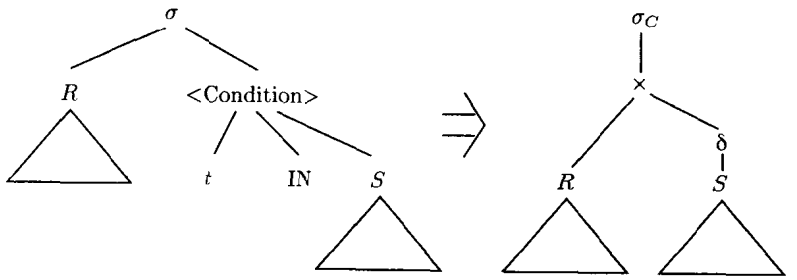


图16-15 本规则处理涉及IN条件的两参数选择

例16.17 考虑图16-14中的树，我们将上面所描述的用于IN条件的规则应用到该树上。在这个图中，关系R是StarsIn，关系S是由以π_{name}为根的子树所构成的关系代数表达式的结果。元组t有一个分量，即属性starName。

两参数选择被 $\sigma_{starName=name}$ 所替换；它的条件 C 是元组 t的一个分量与查询 S 的结果的属性取等值。σ 结点的子女是一个 × 结点，× 结点的参数是标有 StarsIn 的结点以及 S 的表达式的根。注意，由于 π_{name} 是 MovieStar 的码，没有必要在 S 的表达式中引入重复消除操作符 δ。新的表达式如图16-16所示。它完全是关系代数，且等价于图16-13中的表达式，尽管其结构很不相同。

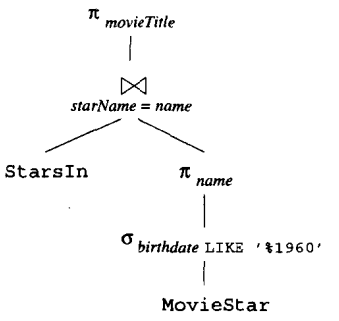


图16-16 应用IN条件规则

□

当子查询相关时，将子查询翻译成关系代数的策略更为复杂。由于相关子查询涉及在其之外定义的未知值，它们不能进行孤立的翻译。确切地讲，我们需要对子查询进行翻译，使得它能产生一个出现了某些特定的额外属性的关系，这些属性在以后将与外部定义的属性相比较。然后把从子查询到外部属性的相关属性的条件应用到这个关系上，不再需要的额外属性可以投影消除。在这个过程中，如果该查询在最后没有消除重复，我们必须小心不经意引入的重复元
814 组。下面这个例子说明了这项技术。

例16.18 考虑查询“找出那些在制作时影星的平均年龄至多为40的电影”。图16-17是该查询的SQL表示。为简化起见，将birthdate作为出生年，这样我们可以取其平均得到一个值，用于与StarsIn的属性movieYear相比较。我们所写查询中每一个对三个关系的引用都有其自身的元组变量，其目的是用于提醒我们各个属性来自何处。

```
SELECT DISTINCT m1.movieTitle, m1.movieYear
FROM StarsIn m1
WHERE m1.movieYear - 40 <= (
    SELECT AVG(birthdate)
    FROM StarsIn m2, MovieStar s
    WHERE m2.starName = s.name AND
        m1.movieTitle = m2.movieTitle AND
        m1.movieYear = m2.movieYear
    );
```

图16-17 找出年长影星出演的电影

图16-18显示了对查询进行语法分析以及部分翻译成关系代数后的结果。在这个初步翻译

过程中，我们把子查询的WHERE子句分解成两个，并使用该子句的一部分把关系的积转换成等号连接。我们在这棵树的结点上保留了别名m1、m2与s，目的是使每个属性的来源更清晰。另一种方法是，我们还可使用投影来更新属性名，从而避免属性名的冲突，但其结果将更加难以理解。

为了移去<Condition>结点并消除两参数σ，我们需要创建一个用于描述<Condition>右分枝上关系的表达式。然而，由于子查询是相关的，无法从子查询中所涉及的关系中获得属性 815
m1.movieTitle或m1.movieYear，这些关系是StarsIn(别名是m2)与MovieStar。因此，我们需要延迟选择$\sigma_{m2.movieTitle=m1.movieTitle \text{ AND } m2.movieYear=m1.movieYear}$，直到子查询中的关系与外层查询中StarsIn的拷贝(其别名为m1的拷贝)相结合之后。为按这种方式转换逻辑查询计划，我们需要修改γ按属性m2.movieTitle与m2.movieYear进行分组，所以当选择需要这些属性时，它们便可获得。最后的效果是我们为子查询计算一个由电影组成的关系，其每个元组由名字、年份以及该电影的平均影星出生年份组成。

修改后的group-by操作符出现在图16-19中；除了两个分组属性外，我们需要把（平均出生日期）平均值更名为abd以便后面引用。图16-19也显示了完全翻译后的关系代数。在γ之上，外层关系的StarsIn与子查询的结果相连接。子查询的选择接着作用到StarsIn与子查询结果的积上；我们把这个选择表示为 θ-连接，在正常应用代数定律后就会如此。在θ-连接之上是另一个选择，这个选择对应于外层查询的选择。在这个选择中，我们把电影的year与它的影星平均出生年份进行比较。代数表达式在分析树顶部结束，这类似于图16-18所示的表达式，投影到所期望的属性上并消除重复。 816

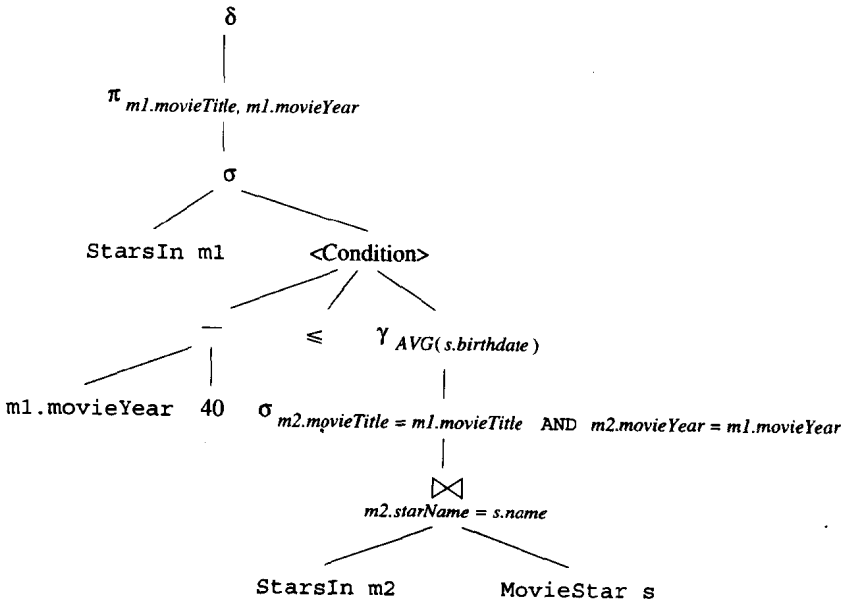


图16-18 图16-17经部分转换后的分析树

正如我们将在16.3.3节看到的那样，一个查询优化器改进查询计划还有许多可做的事情。这个特殊的例子满足了三个条件，使我们可以对查询计划作很大的改进。这些条件是：

1. 重复是最后消除的；
2. StarsIn m1中的影星名字被投影掉了；并且
3. StarsIn m1与剩余表达式之间的连接将StarsIn m1中的title、year属性与StarsIn

m2 中的属性取等值。

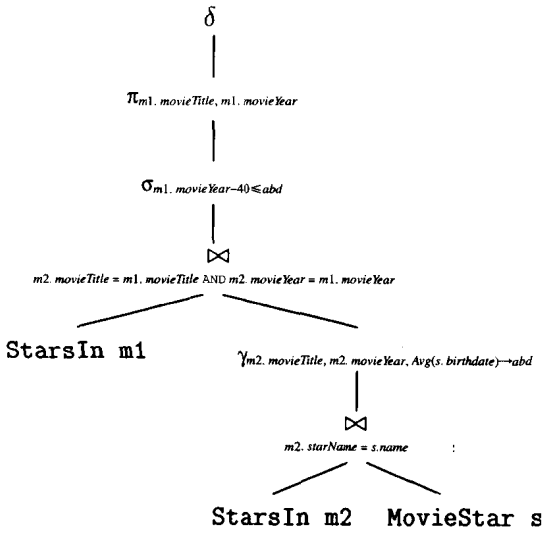


图16-19 把图16-18转换成逻辑查询计划

由于这些条件成立，我们可以分别用 m2.Title与m2.Year替换m1. Title与m1.Year。因此，图16-19中上部的连接是不必要的，参数StarsIn m1亦如此。该逻辑查询计划如图16-20所示。 □

16.3.3 逻辑查询计划的改进

当我们把查询语句转换为关系代数时获得了一个可能的逻辑查询计划。下一步是利用在16.2节中列出的代数定律重写计划。另外，我们可能产生多个逻辑计划，表示不同的操作符顺序或组合。但是在本书中我们假设查询重写模块选取它认为是“最佳”的单个逻辑查询计划，其含义是该计划很可能最终得到最便宜的物理计划。

817

然而，我们没有考虑称为“连接顺序”的问题，因此，涉及多个连接关系的一个逻辑查询计划可视为一簇计划，与不同的方法相对应，连接可被排序和分组。在16.6节我们讨论如何选择连接顺序。类似地，一个查询计划涉及三个或更多关系，这些关系作为其他满足结合律和交换律的操作符的参数，例如并，当我们把逻辑计划转换为物理计划时应当假定允许重新排序或重新分组。我们在16.4节中讨论有关排序与物理计划选择等问题。

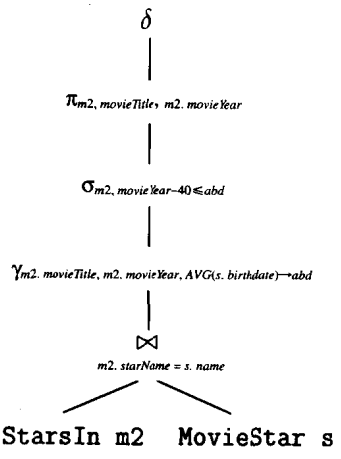


图16-20 图16-19的简化

16.2节中有许多代数定律有望改进逻辑查询计划。下列定律是优化器中最常用到的：

- 选择要尽可能深地下推入表达式树中。如果一个选择条件是多个条件的AND，则可以把该条件分解并分别将每个条件下推。这个策略很可能是最有效的改进技术，但我们应当记起16.2.3节中的讨论，在那里我们看到，在某些情形下首先在树中上推选择是必要的。
- 类似地，投影可被下推到树中，或新的投影可被加入。至于有选择时下推投影应当小心，如16.2.4节所述。
- 重复的消除有时可以消去，或移到树中更方便的位置，如16.2.6节所述。

• 某些选择可以与其下面的积相结合以便把操作对转换成等值连接。一般而言，计算等值连接比之于分别计算两个操作要有效得多。我们已在16.2.5节讨论过这些定律。 818

例16.19 考虑如图16-13所示的查询。首先，我们可以把选择的两部分分解成 $\sigma_{starName=name}$ 和 $\sigma_{birthdate\ LIKE\ '\%1960'}$。后者可以下推到树中，因为所涉及的仅有属性birthdate来自关系MovieStar。第一个条件涉及积两边的属性，但它们取等值，因此积与选择恰为一个等值连接。这个变换的结果如图16-21所示。 □

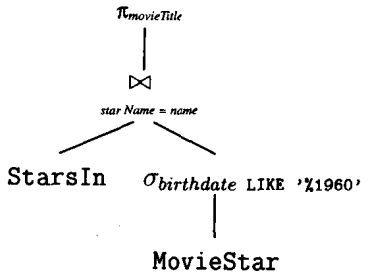


图16-21 查询重写的结果

16.3.4 结合/交换操作符的分组

常规语法分析器不能产生其结点可有数目不限的子女的树。因此，操作符仅以一元或二元形式出现是正常的。然而，满足结合律与交换律的操作符可视为具有任意多个操作数。此外，将诸如连接的操作符看做多路操作符有利于我们重新对操作数排序。这样，当连接作为一个二元连接序列执行时，它们所花的时间将比按语法分析树所隐含的连接顺序的执行时间少。我们在16.6节讨论多路连接的排序。

这样，我们在产生最终的逻辑查询计划之前进行最后一步：对于由相同结合性与交换性操作符结点组成的子树的每一部分，我们将具有这些操作符的结点组合成单个具有多子女的结点。回想一下，常用的满足结合律与交换律的操作符是自然连接、并与交。在特定情形下，自然连接与θ连接可以相互结合：

1. 我们必须用 θ 连接替换自然连接，将具有相同名字的属性取等值。
2. 我们必须增加一个投影以便消除已变为 θ 连接的自然连接中涉及重复属性。 819
3. 连接的条件必须具有结合性。回忆一下，如16.2.1节曾讨论的，存在θ连接不满足结合律的情形。

此外，乘积可被认为是自然连接的特例，并且，如果在树中与连接相邻则与连接相结合。图16-22说明了这种变换，该变换进行的情景是其逻辑查询计划有一两个并操作符聚成的簇以及由三个自然连接操作符聚成的簇。注意，字母 R 到 W 代表任意表达式，不一定是存储的关系。

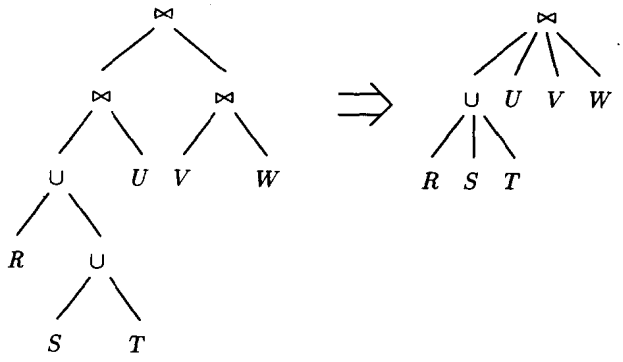


图16-22 产生逻辑查询计划的最后一步：组合结合律与交换律的操作符

16.3.5 习题

习题16.3.1 将下述表达式中的自然连接用等价的θ连接与投影替换。说明所得的θ连接是否形成了一个满足交换律与结合律的分组：

* a) $(R(a,b) \bowtie S(b,c)) \underset{S.c>T.c}{\bowtie} T(c,d)$.

b) $(R(a,b) \bowtie S(b,c)) \bowtie (T(c,d) \bowtie U(d,e))$.

c) $(R(a,b) \bowtie S(b,c)) \bowtie (T(c,d) \bowtie U(a,d))$.

习题16.3.2 将习题16.1.3(a)与(b)的语法树转换成关系代数。对于(b)，写出两参数选择形式以及最终转换成单参数(常规σ_C)选择的形式。

! **习题16.3.3** 给出一条规则用于把下面形式的每一<Condition>转换成关系代数。所有条件均可假定是(通过一个两参数选择)作用到一个关系R上。你可以假设子查询与R不相关。小心不要引入或消除有悖于SQL形式定义的重复。

820

* a) 形如EXISTS (<Query>)的条件。

b) 形如 a = ANY<Query>的条件，其中a是R的一个属性。

c) 形如 a = ALL<Query>的条件，其中a是R的一个属性。

!! **习题16.3.4** 重复习题16.3.3，但允许子查询与R相关。为简便起见，你可假定子查询具有本节所述的简单的select-from-where形式，没有进一步的子查询。

!! **习题16.3.5** 在图16-22右边的分组树可由多少种不同的表达式树得到？记住分组后子女结点的次序未必反映了原始表达式树的次序。

16.4 操作代价的估计

假设我们已对一个查询进行了语法分析并将之转换成一个逻辑查询计划。进一步假定我们已把所选择的变换应用于构造所期望的逻辑查询计划中。下一步我们必须把逻辑计划转换成物理计划。通常我们考察由逻辑计划派生而得的多个不同物理计划，并对每个物理计划进行评价，或估计实现这个转换的代价。经过这种评价，通常称为*基于代价的枚举*，我们选择具有最小估计代价的物理查询计划。当对由给定逻辑计划导出的可能的物理计划进行枚举时，我们为每个物理计划选择：

1. 满足结合律与分配律的操作如连接、并和交的次序与分组。
2. 在逻辑计划中每个操作符的算法，例如，决定使用嵌套循环连接或散列连接。
3. 其他操作符——扫描、排序等——是物理计划所需要的组，在逻辑计划中都不显式地存在。
4. 参数从一个操作符传送到下一个操作符的方式，例如，通过在磁盘上保存中间结果或者通过使用选代算子(iterator)，并每次传给参数一个元组一个主存缓冲区。

我们将逐个考虑这些问题。然而，为了回答与每项选择相关联的问题，我们需要知道各个

821

物理计划的代价是多少。在没有执行计划情况下，我们不能准确地知道其代价，对于一个查询我们当然不想执行多个计划。因此，在计划还没有执行的情况下就要对它的代价进行评估。

在讨论物理计划枚举之前，首先必须考虑如何准确估计这些计划的代价。这种估计是基于数据的参数(参见“概念评论”框)，这些参数要么精确地由数据计算而得，要么是由16.5.1节讨论的“统计量收集”过程来估计。在给定这些参数值的情况下，我们可以对关系大小作许多合理的估计，它们可用于估计完整物理计划的代价。

16.4.1 中间关系大小的估计

对物理计划作选择的目的是最小化执行查询的代价。不管使用什么方法来执行，也不管查询计划的代价如何估计，对计划的代价有重要影响的是中间关系的大小。

理想的情况是我们可得到估计中间关系中元组数的规则，这些规则应：

1. 给出准确的估计。

2. 易于计算。

3. 逻辑上一致；即，一个中间关系大小的估计不依赖于该关系的计算方式。例如，多个关系连接的大小估计不应当依赖于我们计算这些关系连接的次序。 822

但是不存在满足这三个条件一致认同的方法。我们将给出一些适合于大多数情况的简单规则。幸好，大小估计的目标不是准确估计大小，而是帮助选择一个物理查询计划。即便是一个不准确的大小估计方法也可以很好地达到这个目的，只要该方法始终如一地产生误差，也就是说，只要大小估计方法赋予最佳查询计划最小的代价，即使该计划的实际代价与所估计的代价不同。

记号复习

回忆一下15.1.3节中我们用来表示关系大小的惯用记法：

- $B(R)$是容纳关系R所有元组所需的块数。
- $T(R)$是关系R的元组数。
- $V(R,a)$是关系R的属性a的值计数，即，关系R的属性a上所具有的不同值的数目。并且，$V(R, [a_1,a_2,\cdots, a_n])$表示关系$R$的属性$a_1,a_2,\cdots, a_n$作为一起考虑时所出现的不同值的数目，即，$\delta(\pi_{a_1,a_2,\cdots,a_n}(R))$中的不同元组数。

16.4.2 投影大小的估计

投影不同于其他操作符，因为结果的大小是可计算的。由于投影为每个变元元组产生一结果元组，输出大小的仅有变化在于元组的长度。回忆一下，投影操作符是作为包操作符来使用的，不消除重复元组；如果想消除投影中产生的重复，我们必须在投影之后加一个δ操作符。

通常，投影时元组大小缩减，因为某些成分被消除。然而我们在5.4.5节所引入的一般形式投影允许产生新的成分，它们是已有属性的组合。因此存在这样的情形，π 操作符实际上增加了关系的大小。

例16.20 假设$R(a,b,c)$是一个关系，其中a、b是4字节长的整数，c是长为100字节的字符串。设元组头需要12字节。这样R的每个元组占120字节。设块为1024字节长，其中块头占24字节。因此每块可存放8个元组。假设$T(R)$=10 000；即R中有10 000个元组。则$B(R)$=1250。

考虑$S=\pi_{a+b,c}(R)$，即用a与b的和替代a和b。S的元组占116字节：头部占12字节，和占4字节，字符串占100字节。虽然S的元组比R的元组略小，但我们仍可在一个块中存放8个元组。因此，$T(S)$=10 000，$B(S)$=1250。

现在考虑 $U = \pi_{a,b}(R)$，其中去掉了字符串成分。U的元组仅占20字节长。$T(U)$ 仍为10 000。但是，我们现在可在一个块中放入50个元组，故$B(U)$ = 200。从而该投影缩减了关系约6成还多。 □

16.4.3 估计选择的大小

当我们执行选择时，一般而言是减少元组的数目，尽管元组的大小保持原样。最简单的选择情形，即一个属性等于某个常量，有一个比较容易的方法估计结果的大小，前提是我们知道或可估计该属性不同取值的数目。令$S=\sigma_{A=c}(R)$，其中A是R的属性，c是一个常量。则我们推荐如下的一个估计：

823

- $T(S) = T(R)/V(R, A)$

如果属性A的各值在数据库中均匀出现，以上规则必然成立。然而，正如在“Zipfian分布”文字框中讨论的那样，以上公式总的说来仍是最佳估计，即使属性A的诸值在数据库中不均匀分布。不过，我们要求A的所有值在说明A的值的各查询中出现的可能性是相等的。

当选择中涉及非等值比较时，大小估计更困难，如，$S=\sigma_{a<10}(R)$。有人可能会认为平均而言有一半元组满足比较条件而另一半不满足，因此$T(R)/2$是S大小的估计。然而，直觉上涉及非等值比较的查询倾向于产生更少量的可能的元组 ⊖。因此，我们提议承认这种倾向的规则，并假设常见的不等值比较将返回约三分之一的元组而不是一半元组。如果$S=\sigma_{a<c}(R)$，则我们对$T(S)$的估计是：

- $T(S) = T(R)/3$

“不等”比较的情形是比较少的。但是，如果遇到像$S=\sigma_{a\neq 10}(R)$这样的选择，我们建议假设所有的元组将满足这个条件，即取$T(S) = T(R)$作为估计。另外，我们也可用$T(S) = T(R)(V(R, a) - 1)/V(R, a)$作为估计，它比前一估计略小。这个方案承认约有$1/V(R,a)$个$R$的元组不满足条件，因为它们的$a$值确实等于该常量。

当选择条件C是多个等值与不等值比较的AND时，我们可把选择$\sigma_C(R)$作为多个简单选择的级联，其中的每一个选择只检查其中的一个条件。注意，我们安排这些选择的次序是没有关系的。其效果是结果的大小估计是原始关系的大小乘以每个案件的选中率因子。该因子对于任何不等值比较是1/3，对于≠是1，对于条件C中与一个常量相比较的任何属性A取$1/V(R, A)$。

Zipfian分布

当我们假设R的$V(R, a)$个元组之一满足像$a = 10$这样的条件时，似乎做了默认的假设：属性a的所有值出现在R的一个给定元组中的可能性相同。我们同时假定10是这些值之一，这是个合理的假设，因为大部分时间人们在数据库中查找确实存在的信息。然而，等值分布这个假设很少成立，即使近似服从也很少。

许多属性值的出现遵从Zipfian分布，其中第i个最公共值的出现频率正比于$1/\sqrt{i}$。例如，如果最公共值出现1000次，则第二最公共值将预期出现$1000/\sqrt{2}$次，即707次，而第三最公共值将出现约$1000/\sqrt{3}$次，即577次。原先该分布用于描述单词在英语句子中出现的相对频率，后来发现该分布又出现在许多种数据中。例如，在美国，州人口遵从一个近似的Zipfian分布，如人口数位居第二的纽约州拥有人口最多的加利弗尼亚州的70%的人口。因此，如果state是描述美国人口的关系的属性，比如说杂志订阅者清单，我们预期state的诸值按Zipfian分布而不是均匀分布。

只要选择条件的常量随机选定，所涉及属性诸值是否是均匀分布、Zipfian分布或其他分布是无关紧要的；匹配集合的平均大小仍将是$T(R)/V(R, a)$。然而，如果常量的选定也遵从Zipfian分布，则我们期望所选择集合的平均大小会比$T(R)/V(R, a)$稍大。

例16.21 令$R(a,b,c)$是一个关系，$S=\sigma_{a=10 \text{ AND } b<20}(R)$。同时，令$T(R)=10\ 000$，$V(R, a)=50$。则我们的最佳$T(S)$估计是$T(R)/(50\times 3)$，即67。即$R$的元组的1/50将满足$a=10$的过滤，其中1/3将

⊖ 例如，如果你拥有关于教职工工资的数据，你是更可能查询那些挣钱少于200 000美元的教职工还是那些挣钱多于200 000美元的教职工？

满足$b<20$过滤。

一个有意思的特别情形是当条件矛盾时，我们的分析失效。例如，考虑$S=\sigma_{a=10 \text{ AND } a>20}(R)$。根据规则，$T(S)=T(R)/3V(R,a)$，即67个元组。然而，很清楚，没有元组能同时满足$a=10$和$a>20$，因此正确答案是$T(S)=0$。当重写逻辑查询计划时，查询优化器可查找许多特殊情形规则的实例。在上面的例子中，优化器可应用那些选择条件逻辑上等值于FALSE的规则，并且用空集代替S表达式。 □ 824

当选择涉及OR条件时，比如说$S=\sigma_{C_1 \text{ OR } C_2}(R)$，则结果的大小更难确定。一个简单的假设是没有同时满足两个条件的元组，因此结果的大小是分别满足各条件的元组数之和。该度量一般是过高估计，并且事实上有时会引出荒谬的结论：S中的元组数比原始关系R中的还多。因此，另一个简单的方法是取关系R的大小与分别满足条件C_1和C_2的元组数之和这两者中的较小者。

一个稍复杂但可能更准确的对下式的大小估计是假设C_1与C_2相互独立：

$$S=\sigma_{C_1 \text{ OR } C_2}(R)$$

并且如果R有n个元组，其中有m_1个满足C_2，有m_2个满足C_2，则我们估计S中元组的数目为： 825

$$n\,(1-(1-m_1/n)(1-m_2/n))$$

解释如下，$1-m_1/n$是不满足C_1的那部分元组，$1-m_2/n$是不满足C_2的那部分元组。这两数之积是不在S中的那部分R的元组，1减去这个积就是属于S的那部分元组。

例16.22 假设$R(a,b)$有$T(R)=10\ 000$个元组，且$S=\sigma_{a=10 \text{ OR } b<20}(R)$。令$V(R,a)=50$，则满足$a=10$的元组数我们估计为200，即$T(R)/V(R,a)$。满足$b<20$的元组数我们估计为$T(R)/3$，即3333。

对S大小的最简单的估计是取该二数之和，即3533。基于条件$a=10$与$b<20$的独立性的更为复杂的估计是：

$$10000(1-(1-200/10000)(1-3333/10000))$$

即3466。在这两个估计之间没有多少差别，从而对两种估计的选择不可能改变我们对最佳物理查询计划的估计。 □

可能出现在选择条件中的最后一个操作符是NOT。如果关系R有n个元组，则满足条件NOT C的$T(R)$的元组的估计数是n减去满足C的元组估计数。

16.4.4 连接大小的估计

我们在此只考虑自然连接。其他连接可按以下纲要进行处理：

1. 等值连接结果中的元组数在考虑到变量名的变化后可按自然连接那样计算。例16.24将说明这一点。

2. 其他θ连接可看成积之后跟一个选择来作估计，并注意到以下几点：

   (a) 积中的元组数是所涉及关系的元组数之积。

   (b) 等值比较可用自然连接所用的技术来估计。 826

   (c) 两属性的非等值比较，如$R.a<S.b$，可按我们在16.4.3节中讨论的非等值比较形式$R.a<10$那样处理。即我们假定这个条件有选中率因子1/3(如果你相信那些询问是在查询很少出现的条件)或1/2(如果你不作那种假设)。

在开始我们的研究之前假定两个关系的自然连接只涉及两个属性的等值比较。即，我们研究连接$R(X,Y)\bowtie S(Y,Z)$，但初步假定Y是单个属性，当然X、Z可代表任何属性集。

问题在于我们不知道R中的Y值与S中的Y值是如何联系的。例如：

1. 两个关系可能有不相交的Y值集合，在这种情况下，连接是一个空集且$T(R \bowtie S)=0$。

2. Y可能是S的关键字并且是R的外关键字，因此R的每个元组正好与S中的一个元组连接，且$T(R \bowtie S)=T(R)$。

3. 几乎所有S与R的元组都具有相同的Y值，在这种情况下，$T(R \bowtie S)$约为$T(R)T(S)$。

针对最通用的情形，我们作两个简化的假设：

- *值集的包含*。如果Y是出现在多个关系中的一个属性，则每个关系从值y_1，y_2，y_3…的一个固定列表的前头选择其值，并且所有值以其作为前缀。因此，如果R与S是具有属性Y的两个关系，且$V(R, Y) \leqslant V(S, Y)$，则$R$的每个$Y$值将是$S$的一个$Y$值。
- *值集的保持*。如果我们把关系R与另一个关系连接，则不是连接属性的属性A(即两个关系中不同时拥有)不会在其可能的值集中丢失值。更准确地说，如果A是R的一个属性但不是S的属性，则$V(R \bowtie S, A)=V(R, A)$。注意，$R$与$S$的连接次序并不重要，因此我们还可以说$V(S \bowtie R, A)=V(R, A)$。

假设(1)，值集的包含，很显然会被违反，但当Y是S中的关键字且是R中的外关键字时，则是满足的。在许多其他情形下也近似为真，因为直觉上我们预期，如果S有许多Y值，则一个出现在R中的给定Y值有较大的可能出现在S中。

假设(2)，值集的保持，也可能被违反，但当$R \bowtie S$的连接属性是S的关键字且是R的外关键
827 字时它为真。事实上，(2)被违反只有当R中存在“悬挂元组”时，即R中不与S中的元组相连接的元组；并且即便R中存在“悬挂元组”，该假设仍然成立。

在这些假设下，我们可以对$R(X,Y) \bowtie S(Y,Z)$的大小作如下估计。令$V(R,Y) \leqslant V(S,Y)$。则R的每个元组t有$1/V(S,Y)$的机会与给定的一个S元组相连接。因为存在S中的$T(S)$个元组，与t连接的期望元组数是$T(S)/V(S,Y)$。由于有$T(R)$个R的元组，$R \bowtie S$的估计大小是$T(R)T(S)/V(S,Y)$。如果$V(R,Y) \geqslant V(S,Y)$，则一个对称参数给出的估计是$T(R \bowtie S)=T(R)T(S)/V(R,Y)$。一般而言，我们用$V(R,Y)$与$V(S,Y)$中的较大者去除。即：

- $T(R \bowtie S)=T(R)T(S)/\max(V(R,Y), V(S,Y))$

例16.23 让我们来考虑如下三个关系及其重要统计值：

$R(a,b)$	$S(b,c)$	$U(c,d)$
$T(R)=1000$	$T(S)=2000$	$T(U)=5000$
$V(R,b)=20$	$V(S,b)=50$	
	$V(S,c)=100$	$V(U,c)=500$

假定我们要计算自然连接$R \bowtie S \bowtie U$。方法之一是把R与S分为一组，即$(R \bowtie S) \bowtie U$。我们对$T(R \bowtie S)$的估计是$T(R)T(S)/\max(V(R,b),V(S,b))$，其值为$1000 \times 2000/50$，即40 000。

然后我们把$R \bowtie S$与U连接。我们对结果大小的估计是$T(R \bowtie S)T(U)/\max(V(R \bowtie S, c),V(U,c))$，根据值集保持的假设，$V(R \bowtie S, c)$与$V(S,c)$相同，即100；也就是说，当我们做连接时属性$c$的值不会消失。在这种情况下，我们对$R \bowtie S \bowtie U$中的元组数作估计为$40\ 000 \times 5000/\max(100\ 500)$,即400 000。

亦可以用S与U连接来开始。如果这样，我们得到估计$T(S \bowtie U)=T(S)T(U)/\max(V(S, c),V(U,c))=2000 \times 5000/500=20\ 000$。根据值集保持的假设，$V(S \bowtie U,b)=V(S,b)=50$，因此结果大小的估计是：

$$T(R)T(S \bowtie U)/\max(V(R,b),V(S \bowtie U,b))$$

即为1000 × 20 000/50，或400 000。 □

在例16.23中，不论我们由$R \bowtie S$开始连接还是由$S \bowtie U$开始连接，连接$R \bowtie S \bowtie U$的大小估计总是相同的，这不是巧合。回忆一下，16.4.1节中的需求之一就是表达式结果估计应当不依赖于计算的次序。可以证明我们所作的两个假设——值集的包含与保持——保证任何自然连接的估计是相同的，不管连接的次序如何。 828

16.4.5 多连接属性的自然连接

现在我们来看看当Y表示连接$R(X,Y) \bowtie S(Y,Z)$中的多个属性时会发生什么。举个具体的例子，假设我们要作连接$R(x,y_1,y_2) \bowtie S(y_1,y_2,z)$。考虑$R$中的一个元组$r$。$r$与一个给定的$S$的元组$s$连接的概率可按如下方式计算。

首先，r与s在属性y_1上相同的概率是多少？假设$V(R,y_1) \geq V(S,y_1)$。则s的y_1值必然是出现在R中的诸y_1值之一，这是根据值集包含假设而知的。因此，r具有与s相同y_1值的概率是$1/V(R,y_1)$。类似地，如果$V(R,y_1)<V(S, y_1)$，则r中的y_1值将出现在S中，r与s有相同y_1值的概率是$1/V(S,y_1)$。总的说来，我们得知在y_1值上相同的概率是$1/\max(V(R,y_1),V(S,y_1))$。

关于r与s在y_2上取相同值的概率的类似的论据告诉我们，这个概率是$1/\max(V(R,y_2),V(S, y_2))$。由于$y_1$与$y_2$的值是独立的，元组在$y_1$与$y_2$上相同的概率是这些因子的积。因此，源自$R$与$S$的$T(R)T(S)$对元组中，在$y_1$与$y_2$上匹配的期望的对数是：

$$\frac{T(R)T(S)}{\max\big(V(R,y_1),V(S,y_1)\big)\max\big(V(R,y_2),V(S,y_2)\big)}$$

一般而言，当两个关系间有任意数目的公共属性时，以下规则可用于估计自然连接的大小。

- $R \bowtie S$的大小估计是通过$T(R)$乘以$T(S)$，对于每一个R与S的公共属性y，除以$V(R,y)$与$V(S,y)$中的较大者来计算。

例16.24 下面这个例子使用上面的规则。它同时说明了我们一直在做的对自然连接的分析适用于任何等值连接。考虑连接：

$$R(a,b,c) \underset{R.b=S.d \text{ AND } R.c=S.e}{\bowtie} S(d,e,f)$$

假设我们有以下的大小参数：

$R(a,b,c)$	$S(d,e,f)$
$T(R) = 1000$	$T(S) = 2000$
$V(R,b) = 20$	$V(S,d) = 50$
$V(R,c) = 100$	$V(S,e) = 50$

仅考虑元组数还不够

虽然对关系大小的分析集中在结果的元组数上，我们还得考虑每个元组的大小。例如，关系连接后产生的元组比参与连接的任一关系的元组都长。举例来说，考虑两关系的连接$R \bowtie S$，每个关系有1000个元组，所产生的结果可能也有1000个元组。但是，结果占用的块数比R或S任一关系都多。

例16.24是一个恰当而有趣的例子。尽管我们可使用自然连接技术来估计θ连接的元组数，正如那个例子那样，θ连接中的元组比相应自然连接的元组具有更多的属性。具

体地说，θ连接 $R(a,b,c)$ $\underset{R.b=S.d \text{ AND } R.c=S.e}{\bowtie}$ $S(d,e,f)$ 产生六个属性的元组，依次是a到f，而自然连接$R(a,b,c)\bowtie S(b,c,d)$产生相同数目的元组，但每个元组只有四个属性。

如果把$R.b$与$S.d$看成相同的属性，同时也把$R.c$与$S.e$看成相同的属性，则我们可把这个连接看成自然连接。从而上面给出的规则告诉我们$R\bowtie S$的大小估计是积1000 × 2000除以20与50中较大的一个，再除以100与50中较大的数。因此，连接的大小估计是1000 × 2000/(50 × 100) = 400个元组。 □

例16.25 重新考虑例16.23，但是考虑第三种可能的连接次序，其中我们首先做$R(a, b)\bowtie U(c,d)$。这个连接实际上是一个积，结果中的元组数是$T(R)T(U)$=1000 × 5000=5 000 000。注意，积中不同b的数目是$V(R,b)$=20，不同c的数目是$V(U, c)$=500。

当我们将这个积与$S(b,c)$相连接时，乘上元组数并除以$\max(V(R,b),V(S,b))$与$\max(V(U,c),V(S,c))$。这个数是2000 × 5 000 000/(50 × 500)=400 000。注意，第三种连接方法得
829 到的结果大小估计与我们在例16.23中得到的相同。 □

16.4.6 多个关系的连接

最后，我们来考虑自然连接的一般情形：

$$S=R_1\bowtie R_2\bowtie\cdots\bowtie R_n$$

假设属性A出现在k个R_i中，在这k个关系中值的集合的个数(即$V(R_i, A)$的各个值，其中i = 1, 2, …, k)，是$v_1\leqslant v_2\leqslant\cdots\leqslant v_k$，次序从最小到最大。假设我们从每个关系中选一个元组。所选元组在属性A上相同的概率是多少？

为回答这个问题，考虑从具有最小数目的A值v_1的关系中选取的元组t_1。根据值集包含的假
830 设，这v_1个值中的每一个值是在其他具有属性A的关系中所发现的A值中。考虑属性A上有v_i个值的关系。它所选的元组t_i在属性A上与t_1相同的概率是$1/v_i$。由于这个结论对于所有i=2,3,…,k均为真，因此所有k个元组在A上相同的概率是积$1/v_2v_3\cdots v_k$。这个分析为我们估计任何连接的大小给出了一条规则。

- 从每个关系中元组数的积出发，然后，对于至少出现两次的属性A，除以除了$V(R,A)$中最小值之外的所有值。

类似地，我们可以对连接后属性A的值的数目作估计。根据值集保持假设，该值就是这些$V(R,A)$中最小的一个。

例16.26 考虑连接$R(a,b,c)\bowtie S(b,c,d)\bowtie U(b,e)$，假定重要的统计值如图16-23所示。为估计这个连接的大小，我们由各个关系大小的乘积开始，即1000 × 2000 × 5000。接着，查找那些出现多于两次的属性。它们是b出现三次，c出现两次。将积除以$V(R,b)$、$V(S, b)$和$V(U, b)$中较大的两个值，它们是50与200。最后，再除以$V(R,c)$与$V(S,c)$中较大的一个，即200。估计

$R(a,b,c)$	$S(b,c,d)$	$U(b,e)$
$T(R)=1000$	$T(S)=2000$	$T(U)=5000$
$V(R,a)=100$		
$V(R,b)=20$	$V(S,b)=50$	$V(U,b)=200$
$V(R,c)=200$	$V(S,c)=100$	
	$V(S,d)=400$	
		$V(U,e)=500$

图16-23 例16.26的参数

结果是：

$$1000 \times 2000 \times 5000/(50 \times 200 \times 200) = 5000$$

我们也可估计连接中每个属性的值的个数。每个估计值就是该属性所出现的所有关系中的最小值计数。对于a、b、c、d和e，它们各自的估计值是：100、20、100、400和500。 □

为何连接大小估计与次序无关?

这个结论的形式化证明通过对参与连接的关系的个数进行归纳来完成。我们不准备给出这个证明，但在这个文字框中给出直观描述。假设我们要对一些关系进行连接，最后一步是

$$(R_1 \bowtie \cdots \bowtie R_n) \bowtie (S_1 \bowtie \cdots \bowtie S_m)$$

我们可以假设不管对R关系如何连接，其连接的大小估计是各关系R的大小除以除了在R中出现多于一次的每个属性的最小值计数之外的所有值。对每个属性估计的值计数是R关系中最小的值计数。对各S关系可作类似的陈述。

当我们把用于估计两个关系连接大小的规则（见16.4.4节）应用到由R连接以及S连接所形成的两个关系上，其估计是两个估计的乘积除以每个出现在R与S关系中属性值计数中的较大者。因此，这个估计必然有一个因子，它是$R_1, \cdots, R_n, S_1, \cdots, S_m$中每一关系的大小。此外，对于每个不是该估计属性的最小值计数的属性值计数，该估计将拥有一个除数。或者这个除数已经在R或S中出现，或者它是在最后一步引入的，因为其属性A同时出现在R与S中，并且它是$V(R_i,A)$的最小值与$V(S_j,A)$的最小值中较大的一个值计数。

基于我们所作的两个假设——值集的包含与保持——我们得到上面给出的估计规则的一个令人惊讶且方便的特性：

- 不管我们如何对几个关系的自然连接中各项进行组合与排序，分别对每个连接应用估计规则所得到的结果大小是相同的。此外，这个估计与我们对几个关系作为一个整体来连接并应用估计规则所得到的结果大小估计是相同的。 831

例16.23与例16.25是对3个关系的3种组合方式应用规则的一个说明，其中包括了连接之一是积的组合情形。

16.4.7 其他操作的大小估计

我们已经知道两个操作，它们对元组数估计有一个准确的公式：

1. 投影不改变关系中的元组数。
2. 积所产生结果的元组数等于各变元关系的元组数的乘积。

其他两个操作——选择与连接——我们也得到了合理的估计技术。然而，对于其余的操作，结果大小却不易确定。我们来看看其他关系代数操作符，并给出一些如何进行估计的建议。 832

并

如果采用包的并，则其大小正好是变元大小之和。集合的并可以大至两变元大小之和，也可小至两变元之较大者。我们建议取其中间值，例如取和与较大者的平均值(等于较大者加上一半较小值)。

交

结果可以少至0个元组或多至两变元之较小者，无论采用集合的交还是包的交。方法之一是取两极端的平均值，即较小值的一半。

另一种方法是注意到交是自然连接的极端情况，采用16.4.4节的公式。当采用集合交时，

这个公式保证所产生的结果不大于两关系的较小者。然而，在包交的情形下，可能存在异常情况，估计值大于两变元的较大者。例如，考虑$R(a,b) \cap_B S(a,b)$，其中R由两份(0,1)元组的拷贝组成，S由同一元组的3份拷贝组成。则：

$$V(R, a) = V(s,a) = V(R,b) = V(S,b) = 1$$

$T(R) = 2$，$T(S) = 3$。估计是$2 \times 3/(\max(1, 1) \times \max(1, 1))=6$，这是根据连接规则得到的，但显然结果中不会多于$\min(T(R),T(S))=2$个元组。

差

当我们计算$R - S$时，结果中可具有$T(R)$至$T(R) - T(S)$个元组。我们建议估计值取其平均值：$T(R) - T(S)/2$。

消除重复

若$R(a_1,a_2,\cdots,a_n)$是一关系，则$V(R,[a_1,a_2,\cdots,a_n])$的大小为$\delta(R)$。然而，通常我们得不到这个统计值，因此必须取近似值。在极端情形下，$\delta(R)$的大小可与R的大小相同(R无重复元组)或为
833 1(R中全部元组相同)[⊖]。$\delta(R)$中元组数的另一个上限是可能存在的不同元组的最大数：$V(R,a_i)$之积，i=1,2,⋯,n。该数可能比$T(R)$的其他估计更小。有多条规则可用于估计$T(\delta(R))$。一个合理的估计是取$T(R)/2$与所有$V(R,a_i)$之积中较小的一个。

分组与聚集

假设有一个表达式$\gamma_L(R)$，要估计其结果的大小。若统计值$V(R,[g_1,g_2,\cdots,g_k])$已知，其中$g_i$是$L$中的分组属性，则它就是我们的答案。然而，这个统计值很可能得不到，因此我们需要寻找另一种方法估计$\gamma_L(R)$的大小。$\gamma_L(R)$中的元组数与分组数相同。结果中的数目可能只有一个分组，也可能有R中的元组数那么多的分组。与δ一样，我们也可据$V(R,A)$之积取分组数的上界，不过这里的属性A只取L中的分组属性。我们仍然建议该估计值取$T(R)/2$与这个积的较小者。

16.4.8 习题

习题16.4.1 下面是4个关系W、X、Y和Z的关键统计值：

$W(a,b)$	$X(b,c)$	$Y(c,d)$	$Z(d,e)$
$T(W) = 100$	$T(X) = 200$	$T(Y) = 300$	$T(Z) = 400$
$V(W,a) = 20$	$V(X,b) = 50$	$V(Y,c) = 50$	$V(Z,d) = 40$
$V(W,b) = 60$	$V(X,c) = 100$	$V(Y,d) = 50$	$V(Z,e) = 100$

估计下列表达式结果关系的大小：

* a) $W \bowtie X \bowtie Y \bowtie Z$

* b) $\sigma_{a=10}(W)$

c) $\sigma_{c=20}(Y)$

d) $\sigma_{c=20}(Y) \bowtie Z$

e) $W \times Y$.

f) $\sigma_{d>10}(Z)$

* g) $\sigma_{a=1 \text{ AND } b=2}(W)$

h) $\sigma_{a=1 \text{ AND } b>2}(W)$

i) $X \underset{X.c<Y.c}{\bowtie} Y$

⊖ 严格地说，若R为空，则R或$\delta(R)$中没有元组，因此下界为0。但是我们很少对这种特殊情形感兴趣。

* **习题16.4.2** 下面是4个关系E、F、G和H的统计值： 834

$E(a,b,c)$	$F(a,b,d)$	$G(a,c,d)$	$H(b,c,d)$
$T(E)=1000$	$T(F)=2000$	$T(G)=3000$	$T(H)=4000$
$V(E,a)=1000$	$V(F,a)=50$	$V(G,a)=50$	$V(H,b)=40$
$V(E,b)=50$	$V(F,b)=100$	$V(G,c)=300$	$V(H,c)=100$
$V(E,c)=20$	$V(F,d)=200$	$V(G,d)=500$	$V(H,d)=400$

利用本节中的估计技术，估计这些元组的连接会产生多少结果元组？

! **习题16.4.3** 如何估计一个半连接的大小？

!! **习题16.4.4** 假设我们计算$R(a,b)\bowtie S(a,c)$，其中R与S均有1000个元组。每个关系的a属性均有100个不同的值，并且它们是相同的100个值。如果值的分布是均匀的，如每个a值正好在每个关系的10个元组中出现，则连接结果中有10 000个元组。进而假设100个a值在每个关系中有相同的Zipfian分布。更准确地说，令这些值是$a_1,a_2,\cdots,a_{100}$。则R与S有a值为a_i的元组数正比于$1/\sqrt{i}$。在这种情形下，连接有多少个元组？你应当忽略给定a值的元组数不是一个整数的情况。

16.5 基于代价的计划选择介绍

无论是选取一个逻辑查询计划还是从一个逻辑计划构造一个物理计划，查询优化器都需要估计特定表达式的代价。在此我们对有关基于代价的计划选择问题加以研究，并在16.6节中我们详细探讨基于代价的计划选择中最重要也是最为困难的问题：多个关系的连接次序选择。

与以前一样，我们假设计算表达式的代价可用所执行的磁盘I/O数来加以近似。而磁盘I/O又受以下因素影响：

1. 所选取用于实现查询的特定逻辑操作符，这是在我们选择逻辑查询计划时所选定的。
2. 中间关系的大小，其估计我们已在16.4节中讨论过。
3. 用于实现逻辑操作符的物理操作符，例如，对一趟或两趟连接的选择，对给定关系是否加以排序的选择；这个问题在16.7节中讨论。
4. 相似操作的排序，尤其是在16.6节中讨论的连接。 835
5. 由一个物理操作符向下一个物理操作符传递的参数，这个问题也在16.7节中讨论。

为进行有效的基于代价的计划选择需要解决许多问题。在本节中，我们首先考虑如何从数据库中有效地获得大小参数，在16.4节中它们对估计关系大小是很关键的。然后我们重新回顾为找到所希望的逻辑查询计划而引入的代数定律。基于代价的分析证实了我们使用许多通用的启发式转换逻辑查询计划的合理性，诸如在树中下推选择等。最后，我们考虑可由所选逻辑计划导出的物理查询计划的各种方法。特别重要的是为减少所需评价的计划数的方法，而同时又保证很有可能考虑到最小代价的计划。

16.5.1 大小参数估计值的获取

16.4节的公式是以知道某些重要参数为基础的，特别是$T(R)$、关系R中的元组数以及$V(R,a)$，即关系R中属性a列的不同值的数目。现代DBMS一般允许用户或管理员显式地要求做统计信息的收集。这些统计值用到以后的查询优化中去估计操作的代价。由此后的数据库修改导致的统计值的变化仅在下一个统计量收集命令发出时才被更新。

通过对整个关系R的扫描，直接可以得到元组数$T(R)$的计数，并且找出每个属性A的不同值数目$V(R,a)$。R所占用的块数$B(R)$可通过实际所用的块数计数(若R是聚集存放)或通过$T(R)$除以一个元组的长度(或除以一个元组的平均长度，若R的元组按可变长格式存放)得到。注意，这

两个$B(R)$估计可能不同，但对于代价比较通常是“足够接近”的，只要我们一致地选取这种方式或那种方式。

此外，一个DBMS可以计算一个给定属性诸值的*直方图*。如果$V(R,A)$不是太大，则该直方图由具有属性A的每个值的元组的数目（或一部分）组成。如果这个属性存在大量不同值，则只有最常出现的值被单独记录，而其他值则分组统计。最常用的直方图类型是：

1. 等宽。选定宽度w以及常量v_0。提供值为v的元组数计数，v的范围是$v_0 \leqslant v < v_0+w$，$v_0+w \leqslant v < v_0+2w$，等等。值$v_0$是最小可能值或当前所知的最小值。在后一种情况下，若见到新的更
836 小的值，我们就把v_0减少w，并在直方图中增加一个计数。

2. 等高。它们是公共的“百分数”（percentile）。我们选择某个分数p，列出最小值、比最小值多p的值、比最小值多$2p$的值，等等，直到最大值。

3. 最频值(most-frequent-value)。我们列出最为公共的值以及它们的出现次数。这个信息可以连同所有其他值作为一组的出现次数计数一起提供，或除了其他值的等宽或等高直方图之外，我们再记录常出现的值。

使用直方图的一个优点是连接的大小估计比按16.4节中的简化方法估计更准确。尤其当连接属性的值显式地出现在参加连接的两个关系的直方图上时，我们可以准确地知道结果中有多少元组将具有该值。对于那些没有显式地出现在一个或两个关系直方图上的连接属性的值，我们按16.4节中的方法估计其连接后的结果。然而，如果使用等宽直方图，两个关系的连接属性有相同的带宽(band)，则我们可以估计相应带宽上连接的大小，并对这些估计值求和。这个结果是正确的，因为只有在相应带宽上的元组才能连接。下面的例子说明了如何进行基于直方图的估计；我们在后面不使用直方图作估计。

例16.27 考虑关于3个最频值及其计数以及对其余值进行分组的直方图。假设我们想要计算连接$R(a,b) \bowtie S(b,c)$。令$R.b$的直方图是：

1：200，0：150，5：100，其他值：550

即，R的1000个元组中，有200个b值为1，150个b值为0，100个b值为5。此外，有550个元组的b值不为0、1和5，且其余值中的任一值出现次数不超过100。

令$S.b$的直方图是：

0：100，1：80，2：70，其余值：250

并且假设$V(R,b)=14$，$V(S,b)=13$。即，R的未知b值的550个元组被分成11个值，每个值平均有50个元组，S的未知b值的250个元组被分成10个值，每个值平均有25个元组。

值0和1在两个直方图中显式地出现，因此我们可以计算R的150个$b=0$的元组与S的具有相同b值的100个元组的连接结果有15 000个元组。类似地，200个$b=1$的R元组与$b=1$的80个S元组的连接结果有16 000个元组。

837 对其余元组影响的估计则更为复杂。我们将继续作如下假设：在具有较小值集的关系中(本例是S)出现的每个值将在其他关系的值集中出现。因此，在S的其余11个b值中，我们可知其中值之一是2，并且假定另一个值是5，因为它是R中最常出现的值之一。我们估计2在R中出现50次，5在S中出现25次。这些估计是分别通过假设该值是它的关系的直方图中“其他”值中的一个而获得的。B值为2的额外元组数是$70 \times 50 = 3500$，b值为5的其余元组数是$100 \times 25 = 2500$。

最后，在两个关系中还有其他9个b值，我们估计它们每一个值在R中出现50次，在S中出

现25次。因此9个值中的每一个给结果中增加的元组数是50 × 25 = 1250。因而结果大小的估计是：

$$15000+16000+3500+2500+9 \times 1250$$

即48 250个元组。注意，16.4节中较为简单的估计是1000 × 500/14，即35 714，它是基于每个关系中的每个值出现次数均等的假设。□

例16.28 在这个例子中，我们假定等宽划分。我们还将说明在知道两个关系的值几乎不相交的情况下，如何影响对连接结果大小的估计。关系是：

```
Jan(day, temp)
July(day, temp)
```

查询是：

```
SELECT Jan.day, July.day
FROM Jan, July
WHERE Jan.temp = July.temp;
```

即，找出在1月份和7月份中有相同温度的日期对。这个查询计划是对Jan与July在温度上做等值连接，然后投影到每个day属性上。

假设关系Jan与July有关温度的直方图由如图16-24所示的表给定[⊖]。一般而言，如果两个连接属性有相同带宽集（可能关系之一的某些带宽为空）的等宽直方图，则我们可以通过估计相应带宽的每一对的连接的大小并求和来估计连接的大小。

如果相应带宽分别有T_1与T_2个元组，且在这一带宽中值的个数是V，则在那些带宽上连接的元组数估计是T_1T_2/V，遵从16.4.4节中制定的原则。对于图16-24的直方图，许多积是0，因为T_1与T_2之中的某一个为0。仅有的两个均不是0的带宽是40~49与50~59。因为V=10是带宽的宽度， 838
40~49的带宽产生10 × 5/10=5个元组，而50 ~ 59带宽则产生5 × 20/10=10个元组。

因此这个连接大小估计是5+10=15个元组。如果我们没有直方图，只知道每个关系有245个元组分布于0~99的100个值之中，则我们对连接大小的估计是245 × 245/100=600个元组。□

范围	Jan	July
0 ~ 9	40	0
10 ~ 19	60	0
20 ~ 29	80	0
30 ~ 39	50	0
40 ~ 49	10	5
50 ~ 59	5	20
60 ~ 69	0	50
70 ~ 79	0	100
80 ~ 89	0	60
90 ~ 99	0	10

图16-24 温度的直方图

⊖ 赤道南部的朋友应当将1月份与7月份的列对调。

不同的代价估计方法

前面已经提到了将磁盘I/O的估计作为预测查询实际代价的一个好方法。但有时候估计磁盘I/O可能太复杂并且容易出错，而且如果要估计的是逻辑计划而不是物理计划的代价，I/O估计可能行不通。在这种情况下，只考虑16.4节讨论的中间结果大小可能是一种代价估计的有效方法。记住，查询优化器只需要比较查询计划，而不需要预测实际的执行时间。另一方面，磁盘I/O作为代价预测有时可能过于粗略。一个更详细的代价估计应考虑CPU时间，进一步详细的话还应结合磁盘访问块可能的位置来考虑磁头的运动。

16.5.2 计算统计量

在大多数DBMS中，统计量的周期性计算是一种规范。这有几个原因。首先，统计量在短
839 时间内不会发生剧烈变化。其次，一旦统计量被一致性地用于所有计划，那么即使是部分不准确的统计量也是非常有用的。第三，选择保持统计量的最新状态可能使统计量自身成为数据库中的“热点”，因为统计量的读是很频繁的，我们也不愿频繁地去更新它们。

统计量的重新计算在一段时间或者若干更新之后会被自动触发。但是，数据库管理员应注意，如果查询优化器根据正常的原则选择了执行较差的查询计划，它可能会请求重新计算统计量以便改正这一问题。

计算整个关系R的统计量可能会非常昂贵，尤其当计算关系的每一个属性a的$V(R, a)$时（或者更为恶劣，计算每个a的直方图）。通常的方法是通过数据的部分采样来计算近似的统计量。例如，设想我们想以一小部分元组为样本来得到$V(R, a)$的一个估计值。可信的统计计算可能十分复杂，这取决于若干假设，例如属性a的值是否按Zipfan分布或者其他分布方式一致性分布。然而，直觉是这样的：如果观察R的一个小样本，比方说R的1%的元组，可以发现我们见到的a的大多数值是不同的，这意味着$V(R, a)$与$T(R)$可能很接近。如果发现样本中只有很少的不同的a值，则意味着我们已经看到了当前关系中a的大多数值了。

16.5.3 减少逻辑查询计划代价的启发式

有关查询或子查询的代价估计的一个重要用途是应用查询的启发式变换。在16.3.3节我们已经看到，某些应用中与代价估计无关的启发式可期望几乎肯定会改善逻辑查询计划的代价。在树中下推选择是这种转换的典型例子。

然而，在查询优化处理中还存在其他方法，依据一个转换之前与之后的代价估计，当该转换呈低代价时我们就应用这个转换，否则避免使用这个转换。尤其是当正在产生所想要的逻辑查询计划时，我们可能考虑许多可能的转换以及这些转换之前与之后的代价。举个例子来说明这些问题及其过程。

840 **例16.29** 考虑如图16-25所示的初步的逻辑查询计划，并令关系R与S的统计量如下所示：

$R(a,b)$	$S(b,c)$
$T(R)=5000$	$T(S)=2000$
$V(R,a)=50$	
$V(R,b)=100$	$V(S,b)=200$
	$V(S,c)=100$

为从图16-25产生最终的逻辑查询计划，我们坚持尽可能下推选择。不过，我们不能确信下推δ到连接下面是否有意义。因此，由图16-25产生了如图16-26所示的两个查询计划；它们的不同之处在于，是在连接之前还是在连接之后消除重复。注意，在计划(a)中，δ被下推到树

的两个分枝中。如果R与S已知无重复，则分枝中的δ可以去除。

在16.4.3节中，我们已经知道如何估计选择结果的大小；我们用$T(R)$除以$V(R,a)$得50。并且我们也知道如何估计连接的大小；将参数的大小相乘，并除以max $(V(R,b),V(S,b))$，即200。我们所不知道的是重复消除之后关系大小的估计。 841

结果大小的估计不必相同

注意，在图16-26中，两树根部的估计不同：一个是250，另一个是500。因为估计是一个不精确的事件，会发生这类异常。事实上，这是保证一致性的一个例外，如我们在16.4.6节中所做的那样。

直观上讲，计划(b)的估计较高是因为，如果R与S中有重复，这些重复在连接中会相乘。例如，对于R中出现3次、S中出现2次的元组，在连接$R \bowtie S$中会出现6次。我们用于估计δ结果大小的简单公式没有考虑重复的效果被原先的操作放大的可能性。

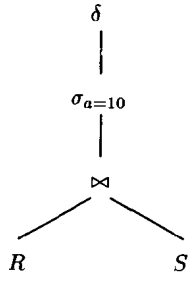


图16-25 例16.29的逻辑查询计划

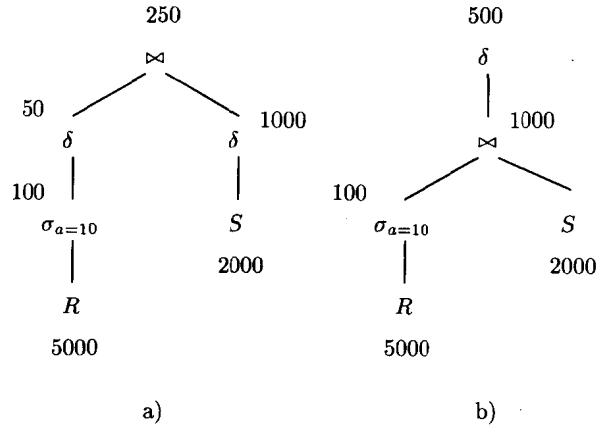


图16-26 最佳逻辑查询计划的两个候选计划

首先考虑$\delta(\sigma_{a=10}(R))$的大小估计。由于$\sigma_{a=10}(R)$只有一个a值却有100个b值，并且这个关系估计有100个元组，16.4.7节的规则告诉我们，每个属性的值计数之积不是一个限定因子。因此，我们估计δ结果大小是$\sigma_{a=10}(R)$元组数的一半。所以，图16-26(a)说明了$\delta(\sigma_{a=10}(R))$的元组数估计是50。

现在考虑图16-26(b)中δ结果的估计。连接有一个 a 值，b 值是 $\min(V(R,b), V(S,b))=100$，c值是$V(S,c) = 100$。因此值计数的乘积不限制 δ 的结果大小。我们估计这个结果是500个元组，或连接中一半的元组数。

为比较图16-26中的两个计划，我们把除了根结点与叶结点之外的所有结点的大小估计相加。我们排除根节点与叶结点，是因为这些结点的大小不依赖于计划的选取。对于计划a这个代价，即内部结点估计大小之和，是100+50+1000=1150；而计划b的和是100+1000=1100。因此，根据这个小差额我们推知，延迟重复消除到最后是一个较好的计划。我们可能得出相反的结论，比方说如果R或S只有少量b值。此时连接的大小会更大，使得计划b的代价更大。 □

16.5.4 物理计划的枚举技术

现在我们来考虑在逻辑查询计划到物理查询计划的转换中如何使用代价估计。基本的方法

称为*穷尽法*（exhaustive），它对16.4节开头中列出的各种问题中的每一个选择加以组合(连接的次序、操作符的物理实现，等等)。每个可能的物理计划被赋予一个估计的代价，并选择具有最小代价的一个计划。

842 不过，存在多种选择物理计划的方法。在本节中，我们将概述已被使用的各种方法，在16.6节讲述有关选择连接次序的重要问题中的主要思想。在继续讲解之前，先对搜索可能的物理计划间的两个主要方法加以说明：

- *自顶向下*：这里，从逻辑查询计划树的根部开始向下进行。对于根结点操作的每个可能的实现，考虑计算其参数的每种可能的方法，并计算每种组合的代价，取最优的一个[⊖]。
- *自底向上*：对于逻辑查询树的每个子表达式，计算用于计算该子表达式的所有可能方法的代价。子表达式E的可能性与代价是通过考虑子表达式E的各种选项并按所有可能的方式与E的根操作符的实现相结合来进行计算。

实际上，如果做最广泛的解释则两种方法之间没有太大的区别，因为每一方法都考虑了实现查询树中每个操作符各种方式的组合。当限制搜索时，自顶向下方法使我们可能去掉某些自底向上方法不能去除的选项。但是，有效限制选择的自底向上策略已被研究出来了，因此下面我们集中在自底向上的方法中。

实际上，你可能已经注意到自底向上方法已有明显的简化，其中我们在计算较大子表达式的计划时只考虑每个子表达式的*最佳计划*。这个方法，在下面列出，称为*动态规划*，不保证产生最优计划，尽管它常常可获得最优。后面也列出了称为Selinger风格(或*系统R风格*)的优化方法，它利用了一个子表达式计划中的某些计划所具有的特性，目的是为了从对于某些子表达式不是最优的计划中得到总体上最优的计划。

启发式选择

方法之一是使用通常用于选择逻辑计划的方法选择物理计划：基于启发式作一系列选择。在16.6.6节中，我们将讨论连接次序的“贪婪”启发式。在该方法中，我们由连接两个具有最小估计大小的关系开始，然后对这个连接结果以及参与连接的关系集合重复这个过程。有许多
843 可应用的启发式；下面是一些最常用的启发式：

1. 如果逻辑计划需要选择$\sigma_{A=c}(R)$，且保存的关系R在属性A上有索引，则执行一个索引扫描以获得A值等于c的R的元组。

2. 更一般地讲，如果选择涉及像上面$A=c$那样的一个条件以及其他条件，我们可以先进行一次索引扫描，然后对元组中作进一步选择来实现这个选择，我们将用物理操作符fiLter来表示。有关这个问题在16.7.1节中作进一步讨论。

3. 如果连接的一个参数在连接属性上有索引，则采用索引连接，其中该关系在内层循环中。

4. 如果连接的一个参数是排序的，则采用排序连接比用散列连接好，尽管未必比用索引连接好，如果可能的话。

5. 当计算三个或多个关系的并或交时，先对最小关系进行组合。

⊖ 记得在16.3.4节中讲到，逻辑查询计划树的单个结点可能表示单个可交换、可结合操作符的各种使用方法，如连接。因此，考虑单个结点的所有可能计划本身可能涉及非常多选择的枚举。

分枝界定计划枚举

这个方法在实际中经常使用，它通过使用启发式为整个逻辑查询计划找到一个好的物理计划开始。令这个计划的代价为C。然后当我们考虑这个子查询的其他计划时，可以去除那些代价大于C的子查询的计划，因为这个子查询的计划不可能参与到比我们已知计划更好的完整查询的计划中。类似地，如果我们构造出代价小于C的完整查询的一个计划，则在此后的物理查询计划空间搜索中用较好计划的代价替换C。

这种方法一个重要的好处在于我们可以判定何时中止搜索并获得到目前为止最优的计划。例如，如果代价C较小，则即便可以发现更好的计划，为找到这些计划所花费的时间可能超过C，因此继续搜索是没有意义的。但是，如果C较大，则花点时间希望找到一个更快的计划是明智的。

爬山法

这个方法从一个依据启发式选定的物理计划开始，实际上我们是在物理计划与代价的一个“峡谷”中进行搜索。接着我们可以对计划作小的修改，如用另一种方法替换一个操作符的一个方法，或通过使用结合律与/或交换律对连接重新排序，找到具有较低代价的“邻近”计划。如果我们找到一个计划，小的修改已不能产生代价更低的计划，则选择这个计划作为选定的物理查询计划。

844

动态规划

在这个一般性自底向上策略的变体策略中，对于每个子表达式，我们仅保留最小代价的计划。当自底向上对这棵树进行处理时，假定每个子表达式使用了最佳计划，我们对每个结点的可能实现加以考虑。我们在16.6节深入研究这个方法。

Selinger风格优化

这个方法改进了动态规划方法，不仅记录了每个子表达式的最小代价的计划，而且也记录了那些具有较高代价但所产生结果的顺序对表达式树中较高层很有用的计划。这类有趣的排序的例子是当子表达式的结果按以下属性排序：

1. 在排序(τ)操作符根结点中说明的属性。
2. 后分组操作符(γ)的分组属性。
3. 后连接的连接属性。

如果我们把一个计划的代价视为是中间关系的大小之和，则对一个变元进行排序似乎没有什么优点。然而，如果我们使用更准确的度量，如磁盘I/O数，作为代价，则当我们使用15.4节中基于排序的算法之一时，对某一变元排序的优点就变得清晰了，为已排序的参数节省了第一趟工作。

16.5.5 习题

习题16.5.1 使用$R.b$与$S.b$的直方图估计连接 $R(a,b) \bowtie S(b,c)$的大小。假设$V(R,b) = V(S,b) = 20$，两个属性的直方图给出了4个最公共值的频率。列表如下：

	0	1	2	3	4	其他
$R.b$	5	6	4	5		32
$S.b$	10	8	5		7	48

假定所有20个值等可能发生，其中$T(R) = 52$且$T(S) = 78$，这个估计与较简单的估计相比怎样？

* **习题16.5.2** 如果我们有如下直方图信息，估计连接$R(a,b) \bowtie S(b,c)$的大小：

	$b<0$	$b=0$	$b>0$
R	500	100	400
S	300	200	500

845

! **习题16.5.3** 在例16.29中，我们建议减少两个名为b的属性的值的数目可能使图16-26中的计划a比计划b更好。对于什么样的值：

* a) $V(R,b)$

b) $V(S,b)$

将使计划(a)比计划(b)的代价更低？

! **习题16.5.4** 考虑四个关系R、S、T和V。它们各自分别有200、300、400和500个元组，随机并相互独立地从1000个元组的同一池中选取。例如，在R中找到任意给定元组的概率是1/5，对于S是3/10。一元组同时在R与S中的概率是3/50。

* a) $R \cup S \cup T \cup V$所期望的大小是多少？

b) $R \cap S \cap T \cap V$所期望的大小是多少？

* c)什么样的并的顺序得到最小的代价(估计中间关系大小的总和)？

d) 什么样的交的顺序得到最小的代价(估计中间关系大小的总和)？

! **习题16.5.5** 如果四个关系有从1000个元组中随机抽取的500个[⊖]，重复习题16.5.4。

!! **习题16.5.6** 假设我们希望计算表达式

$$t_b(R(a,b) \bowtie S(b,c) \bowtie T(c,d))$$

即，我们把三个关系作连接得到在属性b上排序的结果。我们作简化的假设：

1. 我们不会先对R与T作连接，因为那是一个积。

2. 任何其他连接可用一个两遍排序连接或散列连接进行，但不采用其他的方式。

3. 任何关系或任何表达式的结果，可用两阶段、多路归并排序算法排序，但不采用任何其他算法。

4. 要连接的两个关系的结果将作为参数传送给最后的连接，一次一块，在磁盘上不作临时存放。

846

5. 每个关系占1000块，两个关系的任一连接结果占5000块。

基于这些假设回答以下问题：

* a) Selinger风格的优化将考虑所有的子表达式与顺序是什么？

b) 使用磁盘I/O数作为代价度量[⊖]，说明哪个查询计划得到最小的代价。

!! **习题16.5.7** 对于某些表达式E和F（你可以选择），给一个形如$E \bowtie F$的逻辑查询计划的例子，其中使用最佳计划计算E与F不允许为最后连接的选择算法，该算法用于最小化计算整个表达式的总的代价。对于可用的主存缓冲区数目与E、F中所提及的关系大小作你所希望的假设。

⊖ 这个习题的相应部分的答案没有在Web上发布。

⊖ 注意，由于作了使用连接方法某些特别的假设，我们可以估计磁盘I/O数，而不是依赖于更简单但更不正确的元组计数作为代价估计。

16.6 连接顺序的选择

在本节中，我们着眼基于代价的优化中的关键问题：为三个或三个以上关系的（自然）连接选择顺序。该思想同样适用于其他像并和交这样的二元操作，但是在实际应用中这些操作并不是那么重要，因为它们与连接相比，只需要很少的时间，并且很少以连续三个或三个以上的形式出现。

16.6.1 连接的左右变元的意义

当选择连接顺序时，我们应该记住在第15章中提到的连接方法中大多数是不对称的，也就是说，两个变元关系所代表的意义是不同的，这些连接的代价取决于哪种关系代表何种意义。可能最为重要的连接是在15.2.3节介绍的一趟连接，它将一个关系（较小的优先）读入主存，并形成一种结构，例如一个散列表，从而可以便利地匹配来自其他关系中的元组。然后该连接再读入其他关系，每次一块，并将关系中的元组和已存储在内存中的元组进行连接操作。

假设当我们选择一个物理计划时，决定用一趟连接。然后我们应该将连接的左变元作为（较小的）关系存储在主存的数据结构中（该关系称为“*构建关系*”），同时以每次一块读入连接的右变元并将它的元组与已存储的关系进行匹配（“*探针关系*”）。以变元作为区分的其他连接算法包括：

1. 嵌套循环连接，在这种连接中我们认为左变元是外层循环关系。
2. 索引连接，我们认为这种连接的右变元有索引。

847

16.6.2 连接树

当有两个关系的连接时，我们需要给变元排序。按照惯例，我们应该选择估计数值较小的变元作为左变元。应注意以上提到的算法——一趟连接、嵌套循环连接以及索引连接——如果以较小的作为左变元，则每个算法将会运行最佳。更为准确地说，一趟连接和嵌套循环连接都为较小的关系（构建关系，或者外层循环）指定特定的意义，而只有当一个关系是小的，并且另外一个关系有一个索引时，索引连接通常才是合理的选择。各个变元的大小具有重要的并且可辨别的差别，这是非常普遍的，因为一个涉及到连接的查询往往会涉及至少一个属性上的选择，并且这个选择使得一个关系的估计值大大减小。

例16.30 回忆图16-4中的以下查询

```
SELECT movieTitle
FROM StarsIn, MovieStar
WHERE starName = name AND
    birthdate LIKE '%1960';
```

我们将关系StartsIn与关系MovieStar的选择进行连接，从而可以推导出图16-21所示的首选逻辑查询计划。我们并没有给定关系StartsIn和MovieStar的估计值，但是我们可以认为在一年中出生的明星中进行选择将会在MovieStar中产生大约1/50的三元组。由于每一部电影通常有几个明星，我们假设StarsIn在开始将大于MovieStar，这样，连接的第二个变元，$\sigma_{\text{birthdate LIKE '\%1960'}}(\text{MovieStar})$，会远小于第一个变元StarsIn。我们得出的结论与图16-21中所示的变元的顺序应该是相反的，这样，就选择MovieStar为左变元。 □

当有两个关系时，对于一棵连接树只有两种选择——在两个关系中任选一个作为左变元。当连接有两个以上的关系时，可能的连接树的数量会迅速增长。比如，图16-27所示为有四种关系R、S、T和U进行连接时三种可能的树的形状。由于有变元顺序，并且对于n种情况将会有$n!$种方法对其进行排序，当考虑叶节点的各种可能的标识时，每棵树将会代表4!=24棵不同的树。

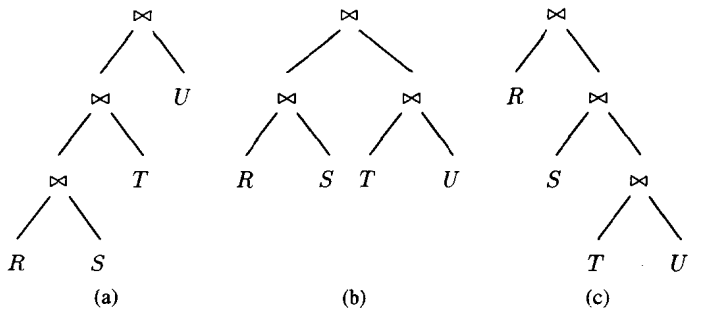


图16-27 对四种关系进行连接的方法

16.6.3 左深连接树

图16-27(a)是称之为*左深树*的一个例子。一般来说，一个二叉树如果所有右边的子女都是叶节点的话，它就是左深树。同样，如图16-27(c)所示的树，所有左边的子女都是叶节点，这
848 样的树称为*右深树*。如图16-27(b)所示的树既不是左深树也不是右深树，称之为*紧密树*。在下面我们将只考虑以左深树作为可能的连接顺序来讨论一个双重优点。

1. 对于给定叶节点的可能的左深树的数目是很大的，但几乎不会像所有树的数目那样大。因此，如果我们将搜索限制在左深树，查询计划的搜索将可以用于比较大的查询。

2. 用于连接的左深树可以和通用的连接算法很好地交互——尤其是嵌套循环连接和一趟连接。基于左深树和这些算法的查询计划将会比非左深树所用的同样的算法更有效。

实际上，一个左深连接树或右深连接树中的叶节点可以是带有除连接之外的操作符的内部节点。例如图16-21在技术上就是一个带有一个连接操作的左深连接树。将一个选择应用于连接的右操作数的事实并没有将树排除在左深树类之外。

对于给定关系数目的多路连接，左深树的数目的增长几乎不会像所有树的数目增长那样快。对于n个关系，我们只有一种左深树的形状可以以$n!$种方法来分配关系。对这n个关系有同样数目的右深树。然而，n个关系的树形状$T(n)$的总数目由以下的递归给出：

$$T(1) = 1$$
$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i)$$

对于第二个等式的解释是：我们可以设置根的左子树中的叶节点数目为介于$1 \sim n-1$之间的任意一个数i，对带有i个叶节点的树的排列将有$T(i)$种方法，用其中任意一种方法对这些叶节
849 点进行排列。同样，对右深树中的$n-i$个叶节点，用$T(n-i)$种方法中的任意一种对其进行排列。

$T(n)$的头几个值为$T(1)=1$、$T(2)=1$、$T(3)=2$、$T(4)=5$、$T(5)=14$和$T(6)=42$。我们将$T(n)$乘上$n!$，就得到当叶节点表示各种关系时所有树的数目。因此，带有6个叶节点且各叶节点均被标识的树的数目为$42 \times 6!$，即30 240，其中6!即720棵是左深树，另外720棵是右深树。

现在，让我们考虑上面曾经提到的左深连接树的第二个优点：即有利于形成有效的计划。我们将给出以下两个例子：

1. 如果用的是一趟连接，并且“建立关系”是在左边，则任何时候所需的内存都将比对同样关系用右深树或紧密树的情况要小。

2. 如果是用迭代的方法实现的嵌套循环连接，则可以避免多次构建任意中间关系。

例16.31 考虑如图16-27(a)所示的左深树，假设以左变元作为建立关系，也就是说左变元将会存储在主存中，对每三个⋈操作符进行一个简单的一趟连接。要计算$R \bowtie S$，需要在主存中保

留R，在计算$R \bowtie S$时，我们还要在主存中保留结果。这样，我们需要$B(R)+B(R \bowtie S)$的主存缓冲区。如果选择最小的关系作为R，并且有一个选择将使得R更小，则有可能很容易获得如此大容量的缓冲区。

算出$R \bowtie S$后，我们需要将该关系与T进行连接。然而，此时R所使用的缓冲区不再需要，可以用它来存储$(R \bowtie S) \bowtie T$的结果。同样，将该关系与U进行连接时，不再需要保留关系$R \bowtie S$，其缓冲区可以用于存储连接的最终结果。一般说来，以一趟连接所计算出的连接的左深树需要的存储空间，是用来存储在任何时候都存储在主存中的至多两个临时关系。

现在，考虑如图16-27(c)中右深树的实现。首先应将R载入主存缓冲区，因为左变元往往是“构建关系”。然后，构建$S \bowtie (T \bowtie U)$并将其用于根连接的探针关系。要计算$S \bowtie (T \bowtie U)$，我们需要将S放入缓冲区，然后计算出$T \bowtie U$作为S的探针关系。但是$T \bowtie U$需要我们先将T读入缓冲区。现在，内存中同时有了R、S和T。一般地讲，如果我们试图计算一个有n个叶节点的右深连接树，我们必须将$n-1$个关系同时读入内存。 850

缓冲区管理器的作用

读者也许注意到在一系列例子中介绍的方法有一个不同点，例如在例15.4和例15.7中，我们假设一个连接可以获得的主存缓冲区的个数有固定的限值。如果为了计算更加灵活，则假设可以获得足够多的缓冲区满足我们的需要，但我们尽量不要用“太多”缓冲区。回忆一下15.7节我们知道，缓冲区管理器可以非常灵活地为操作分配缓冲区。然而，如果一次分配太多的缓冲区，就会使所用算法的性能降低。

当然，整个大小$B(R)+B(S)+B(T)$可能会小于我们在对左深树进行计算的任意两个中间阶段中所需空间的数目，这两个数目分别为$B(R)+B(R \bowtie S)$和$B(R \bowtie S)+ B((R \bowtie S) \bowtie T)$[⊖]。尽管如此，正如在例16.30中指出的，几个连接的查询常常会有一个小的关系，我们可以在一个左深树中，首先以该关系作为最左边的变元。如果R是小的，我们可以预期$R \bowtie S$远小于S以及$(R \bowtie S) \bowtie T$小于T，从而更加证实了左深树的作用。 □

例16.32 现在，假设我们要通过嵌套循环连接来实现如图16-27所示的四路连接，并且对所涉及的三个连接中的每一个都使用一个迭代器（参见15.1.6节）。另外，为简便起见，假设关系R、S、T和U都是已存储的关系而不是表达式。如果我们采用如图16-27(a)所示的左深树，则在树根的循环会获得左变元$(R \bowtie S) \bowtie T$这样大小的主存块。它会将该块和全部U进行连接，只要U是已存储的关系。它只需要对U进行扫描，而不必构建它。一旦获得左变元的下一块，并将其放入内存，就可以再次对U进行读操作，但是如果两边的变元都是大的，嵌套循环连接就不可避免地需要反复进行。

同样，为了获得$(R \bowtie S) \bowtie T$的块，我们将$R \bowtie S$的块放入内存并对T进行扫描。对T进行几次扫描最终也许是必要的，而且也不可能避免。最后，要获得$R \bowtie S$的块需要读入R的块并将它与S进行比较，也许要进行几次。尽管如此，在所有这些过程中，只有已存储的关系要读入几次。当主存不足以保留整个关系时，这种反复读入会使得嵌套循环连接的工作方式出现人为的痕迹。

现在，将左深树的循环行为与如图16-27(c)所示的右深树的循环行为进行比较。树根的循环以读入R的块作为开始。然后它必须构建整个关系$S \bowtie (T \bowtie U)$，并将其与R进行比较。当我们

⊖ 注意在计算代价时我们通常不考虑表达式的存储代价。

将下一个R的块读入内存时，必须再次构建$S \bowtie (T \bowtie U)$。后续的每一个R的块同样需要构建这同一关系。

当然，我们可以一次构建$S \bowtie (T \bowtie U)$，并将其存储在内存或磁盘上。如果将其存储在磁盘，
851 与左深树计划相比，我们就要使用额外的磁盘I/O；如果存储在内存，我们就会遇到曾在例16.31中讨论的关于过度使用内存的问题。 □

16.6.4 通过动态规划来选择连接顺序和分组

要为许多关系选择连接顺序，我们有以下三个选择：

1. 考虑全部。
2. 考虑一个子集。
3. 采用启发式选取一个。

在这里我们应该考虑一个称为*动态规划*的方法来进行枚举。它可以用于或者考虑所有顺序，或者只考虑特定子集，例如限制于左深树的顺序。在16.6.6节我们为选择一个单一的顺序考虑一个合理的启发式。动态规划是一种通用算法范例[⊖]。动态规划的思想是：我们对一个代价表进行填充，只记住我们推出结论所需的最少的信息。

假设我们想对$R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$进行连接。在一个动态规划算法中，我们为包含$n$个关系中的一个或多个关系的每一个子集构建带有一个入口的表。在这个表中我们将记录：

1. 这些关系的连接的估计值。我们可以利用16.4.6节中介绍的公式。

2. 计算这些关系的连接的最小代价。在我们的例子中将使用中间关系大小的和（不包括R_i本身或与该表入口相关的所有关系集合的连接）。回想一下，中间关系大小是我们可以用来估计磁盘I/O、CPU的利用率或者其他因素的实际代价的最简单的量度。不过，如果我们愿意并且可以进行额外的计算，则可以使用其他复杂的估计，比如所有磁盘的I/O。如果使用磁盘的I/O或另一个关于运行时间的量度，则必须考虑连接所用的算法，因为不同的算法会有不同的代价。在已经掌握基本的动态规划技术后我们将讨论这些问题。

3. 给出最小代价的表达式。这个表达式对有待求解的关系集合进行分组连接。我们可以选
852 择性地只考虑左深表达式，在这种情况下表达式仅仅是关系的顺序。

这个表的构建是一个关于子集大小的归纳。有两个变量，依赖于我们是希望考虑所有可能的树的形状还是只考虑左深树。构建表的方法有所不同；当我们在下面讨论表构建的归纳步骤时会对这个不同进行解释。

基础：一个单一关系R的入口包括R的大小、值为0的代价以及R本身的表达式。一对关系$\{R_i, R_j\}$的入口也容易计算。代价为0，是因为没有涉及到中间关系，并且估计值由16.4.6节中介绍的规则给出；如果代价不为0，则它的值为R_i与R_j的乘积再除以R_i与R_j共享的属性的较大的值集大小。表达式要么是$R_i \bowtie R_j$，要么是$R_j \bowtie R_i$。按照在16.6.1节中介绍过的思想，我们挑选R_i与R_j中较小的一个作为左变元。

归纳：现在，我们可以建立表，并为大小是3、4等的所有子集计算入口，直到我们获得大小为n的子集的入口。这个入口告诉我们对所有关系的连接进行计算的最佳方法；它还告诉我们这个方法的估计代价，该值在计算以后的入口时会要用到。我们需要明白如何计算有k个关系的$\mathcal{R}$的入口。

⊖ 动态规划的一般处理请参阅：Aho、Hopcroft和Ullman写的《Data Structures and Algorithms》(Addison-Wesley出版社1984年出版)。

如果希望只考虑左深树，那么对$\mathcal{R}$中的k个关系R的每一个，通过首先计算$\mathcal{R}-\{R\}$的连接并将它与R再进行连接来考虑我们对$\mathcal{R}$进行连接计算的可能。对$\mathcal{R}$进行连接计算的代价等于$\mathcal{R}-\{R\}$的代价加上后一个连接。我们选取能生成最小代价的R。当$\mathcal{R}-\{R\}$作为最终连接的左变元，而R作为右变元时，对$\mathcal{R}$的表达式也就是对$\mathcal{R}-\{R\}$的最佳连接表达式。16.4.6节中的公式将给出$\mathcal{R}$的大小。

如果我们希望考虑所有的树，则关系集合$\mathcal{R}$的入口的计算会变得复杂。我们需要考虑所有的将$\mathcal{R}$分解成不相连接的集合$\mathcal{R}_1$和$\mathcal{R}_2$的方法。对每一个这样的子集，考虑以下两项的和：

1. $\mathcal{R}_1$和$\mathcal{R}_2$的最佳代价。
2. $\mathcal{R}_1$和$\mathcal{R}_2$的大小。

对给出最佳代价的划分，我们利用这个和值作为$\mathcal{R}$的代价，并且$\mathcal{R}$的表达式是$\mathcal{R}_1$和$\mathcal{R}_2$的最佳连接顺序的连接。

例16.33 考虑四个关系R、S、T和U的连接。为了简单起见，我们假设每个关系有1000个元组。它们的属性以及每个关系中属性的数值区间的估计大小如图16-28所示。

对于单独的集合，它的大小、代价以及最佳计划如图16-29所示。即，对每一个单独的关系，给定它们的大小为1000，代价为0，这是因为它们不需要间关系，并且最佳（惟一的）表达式就是关系本身。

$R(a,b)$	$S(b,c)$	$T(c,d)$	$U(d,a)$
$V(R,a)=100$			$V(U,a)=50$
$V(R,b)=200$	$V(S,b)=100$		
	$V(S,c)=500$	$V(T,c)=20$	
		$V(T,d)=50$	$V(U,d)=1000$

图16-28 例16.33的参数

	$\{R\}$	$\{S\}$	$\{T\}$	$\{U\}$
大小	1000	1000	1000	1000
代价	0	0	0	0
最佳计划	R	S	T	U

图16-29 单独集合的表

现在，考虑关系对。由于两个关系的连接中仍然没有中间关系，所以每个关系的代价为0。两个关系中的任意一个都可以作为左变元，因此有两种可能的计划，但由于每个关系偶然会相等，我们将没有根据来选择哪一个计划。在每一种情况下，我们按照字母的顺序选择在前面的。结果关系的大小由一般的公式算出。结果如图16-30所示。注意，1M代表1 000 000，这是由一个乘积得到的连接的大小。 853

	$\{R,S\}$	$\{R,T\}$	$\{R,U\}$	$\{S,T\}$	$\{S,U\}$	$\{T,U\}$
大小	5000	1M	10 000	2000	1M	1000
代价	0	0	0	0	0	0
最佳计划	$R\bowtie S$	$R\bowtie T$	$R\bowtie U$	$S\bowtie T$	$S\bowtie U$	$T\bowtie U$

图16-30 关系对的表

现在，考虑四个关系中的三个关系的连接的表。计算三个关系的一个连接的惟一方法是首先选择两个进行连接。结果的估计大小由标准公式计算出，在此我们省略计算的细节；记住，不管用什么方法计算连接，我们都会得到同样的大小。

关系的每个三元组的代价估计等于一个中间关系的大小——这个中间关系是最先选择的两

个关系的连接。由于我们希望这个代价尽可能小，因此考虑三个关系中的每一对并且提取最小的一对。

854 对于表达式，我们首先将被选中的两个关系组成一组，但是这些可以是左变元或者右变元。假设我们只对左深树感兴趣，则我们常常利用最先的两个关系的连接作为左变元。由于在所有的情况下四个关系中的两个的连接的估计大小至少为1000（相当于每单个关系的大小），如果允许非左深树的话，我们常常会选择一个关系作为这个例子中的左变元。三个关系为一组的汇总表如图16-31所示。

让我们考虑{R,S,T}作为计算的例子。我们必须依次考虑三对中的每一对。如果我们以$R \bowtie S$开始，则代价就是这个关系的大小，即5000，这可以从如图16-30所示的一对关系的表中得知。以$R \bowtie T$开始使得中间关系的代价为1 000 000，以$S \bowtie T$开始则代价为2000。由于后者是三个选择中最小的，我们选择了这个计划。该选择不仅反映在{R,S,T}列的代价入口中，也反映在最佳计划行，在这一行中，将S和T组成一组的计划首先出现。

	$\{R,S,T\}$	$\{R,S,U\}$	$\{R,T,U\}$	$\{S,T,U\}$
大小	10 000	50 000	10 000	2 000
代价	2 000	5 000	1 000	1 000
最佳计划	$(S \bowtie T) \bowtie R$	$(R \bowtie S) \bowtie U$	$(T \bowtie U) \bowtie R$	$(T \bowtie U) \bowtie S$

图16-31 三个关系为一组表

现在，我们必须考虑全部四个关系连接的情况。这个关系的大小估计值为100个元组，因此真正的代价从本质上说是全部位于中间关系的构建。然而，回忆一下，当比较计划时我们从不考虑结果的代价。

有两种通用的方法我们可以计算全部四个关系的连接：

1. 以可能的最佳方法选择三个进行连接，然后与第四个连接。

2. 将四个关系划分为两对，将每一对进行连接，再将两个结果行连接。

当然，如果我们只考虑左深树，则第二种计划被排除，因为它会生成紧密树。如图16-32所示的表在如图16-30和图16-31中的成组方法的基础上总结了7种可能的成组连接的方法。

例如，考虑如图16-32所示的第一个公式。它表示首先将R、S和T进行连接，再将所得结果与U进行连接。从图16-31我们知道，将R、S和T进行连接的最佳方法是先将S和T进行连接。我们曾经利用这个表达式的左深形式，并在右边连接U以继续使用左深形式。如果我们只考虑左深树，则这个表达式以及关系顺序就是惟一的选择。如果允许紧密树，则我们要在左边连接U，因为它将小于其他三种的连接。该连接的代价为12 000，等于代价值加上$(S \bowtie T) \bowtie R$的

855 大小，它们分别为2000和10 000。

分 组	代 价
$((S \bowtie T) \bowtie R) \bowtie U$	12 000
$((R \bowtie S) \bowtie U) \bowtie T$	55 000
$((T \bowtie U) \bowtie R) \bowtie S$	11 000
$((T \bowtie U) \bowtie S) \bowtie R$	3 000
$(T \bowtie U) \bowtie (R \bowtie S)$	6 000
$(R \bowtie T) \bowtie (S \bowtie U)$	2 000 000
$(S \bowtie T) \bowtie (R \bowtie U)$	12 000

图16-32 成组连接以及它们的代价

图16-32中的最后三个表达式表示如果我们包含紧密树时的额外选择。这些表达式首先是由两对关系的连接组成。例如，最后一行表示首先对$R \bowtie U$和$S \bowtie T$进行连接，然后再将两个结果进行连接。该表达式的代价等于两对关系的大小与代价之和。任意一对的代价都为0，这是必然的，它们的大小分别为10 000和2000。由于我们通常选择较小的关系作为左变元，故我们的表达式为$(S \bowtie T) \bowtie (R \bowtie U)$。

在这个例子中我们看到，所有代价中的最小值与第四个表达式$((T \bowtie U) \bowtie S) \bowtie R$相关。我们选择该表达式来计算连接；它的代价为3000。不管我们的动态规划策略是否考虑所有的计划

或者仅考虑左深计划，由于它是左深树，它就是被选择的逻辑查询计划。 □

16.6.5 带有更具体的代价函数的动态规划法

利用关系的大小作为代价的估计可以简化动态规划算法。然而，这个简化将会带来一个缺点，就是它在计算中没有考虑连接的实际代价。举一个极端的例子，如果有一个可能的连接$R(a,b) \bowtie S(b,c)$包括一个带有一个元组的关系R和另一个带有连接属性b的索引的关系S连接，则该连接几乎不花费任何时间。另一方面，如果S没有索引，则我们必须对它进行扫描，即使R是一个单一的关系，这也会花费$B(S)$次磁盘I/O。只考虑R、S以及$R \bowtie S$的代价度量不可能区分这两种情况，所以在组中利用$R \bowtie S$的代价要么会估计过高，要么会被估计不足。

然而，对动态规划算法进行修改以将连接算法考虑进去是不难的。首先，用磁盘I/O作为我们所采用的代价度量，或者我们更愿意用的任何运行时间单元。计算$\mathcal{R}_1 \bowtie \mathcal{R}_2$的代价时，我们将$\mathcal{R}_1$的代价、$\mathcal{R}_2$的代价以及利用可获得的最佳算法对这两个关系进行连接所需的最小代价相加。由于后一个代价一般依赖于$\mathcal{R}_1$和$\mathcal{R}_2$的大小，我们还必须计算这些大小的估计值，正 856
如我们在例16.33中所做的那样。

动态规划的一个功能更强大的版本是基于在16.5.4节中所提到的Selinger风格优化。现在，对每一个可能被连接的关系集合，我们保持的不只是一个代价，而是几个代价。Selinger风格优化不仅考虑算出连接结果的最小代价，还考虑生成按几个“感兴趣”的顺序中的任意一个排序的关系的最小代价。这些感兴趣的顺序包括任何对以后的排序连接有利或者能够生成以用户所期望的序列的全部查询的输出的顺序。当必须生成排序的关系时，必须考虑选择排序连接，使用一趟的或者多趟的，而当不考虑将一个结果进行排序的价值时，散列连接至少常常与相应的排序连接一样好。

16.6.6 选择连接顺序的贪婪算法

如例16.33所建议的，即使是动态规划的仔细限定范围的搜索也会导致计算量与被连接的关系数成指数关系。采用像动态规划或者分枝界定搜索这样的方法来寻找五个或六个关系的最佳连接顺序，是合理的。然而，当连接数超过范围，或者如果我们选择不想在搜索中花费必需的时间，则我们可以在查询优化中采用连接顺序试探。

试探的最普遍的选择是贪婪算法，在这个算法中，我们一次为连接的顺序做一个决定，并且从不返回，或者说一旦做出决定便不再重新考虑。我们将考虑只选择左深树的贪婪算法。“贪婪”是基于这样的思想——希望在树的每一级保持尽可能少的中间关系。

连接的选择率

一个查看诸如为选择一个左深连接树的贪婪算法的试探的有效方式是每个关系R与当前树进行连接时有一个选择率，它是以下的比率

$$\frac{\text{连接结果的大小}}{\text{当前树结果的大小}}$$

由于我们经常不能获得任意关系的确切大小，因此就像以前所做的来估计这些大小值。一个连接顺序的贪婪算法将选取有着最小选择率的那个关系。

例如，如果一个连接属性是R的一个关键字，则选择率几乎为1，这常常是有利的情况。注意，从如图16-28的统计可以判断，属性d是U的一个关键字，并且其他关系没有关键字，这表明了为什么将T和U进行连接是启动连接的最好方式。

基础：以估计连接大小是最小的关系对开始。这些关系的连接成为当前树。

归纳：在所有还没有包含在当前树中的关系中，寻找与当前树进行连接能生成估计大小最小的关系。以旧的当前树作为左变元，被选中的关系作为右变元来形成新的当前树。

例16.34 将贪婪算法应用于例16.33中的关系。基本的步骤是找出连接最小的一对关系。考虑图16-30，可以看出是连接$T \bowtie U$，其代价为1000。因此，$T \bowtie U$是“当前树”。

现在考虑下一步是否将R和S连接进入树。我们比较$(T \bowtie U) \bowtie R$和$(T \bowtie U) \bowtie S$的大小。图16-31告诉我们后者的大小为2000，好于前者，其代价为10 000。因此，我们将$(T \bowtie U) \bowtie S$作为当前树。

现在没有选择了；我们必须在最后一步连接R，这将使得总的代价为3000，相当于两个中
857 间关系大小的和。注意，由贪婪算法得到的树和在例16.33中用动态规划算法所得到的树是相同的。然而，也有贪婪算法寻找最佳结果失败的例子，而动态规划算法能保证找到最佳结果；参看习题16.6.4。

16.6.7 习题

习题16.6.1 对习题16.4.1中的关系，给出评价以下所有可能的连接顺序的动态规划表入口：

a) 只有左深树。

b) 所有树。

在每一种情况下什么是最佳选择？

习题16.6.2 除了以下的修改，其他与习题16.6.1相同：

a) Z的计划改为$Z(d,a)$。

b) $V(Z,a)=100$。

习题16.6.3 用习题16.4.2中的关系重复习题16.6.1。

* **习题16.6.4** 考虑关系$R(a,b)$、$S(b,c)$、$T(c,d)$以及$U(a,d)$的连接，其中R和U都有1000个元组，而S和T有100个元组。另外，对于属性c，有$V(S,c)=V(T,c)=10$，除此之外，所有关系的所有属性都有100个值。

a) 用贪婪算法选择什么样的顺序？其代价是多少？

858 b) 什么是最佳连接顺序？其代价是多少？

习题16.6.5 对以下连接有多少棵树：

* a)七个关系。

b)八个关系。

既不是左深树也不是右深树的有多少？

! **习题16.6.6** 假设我们希望对在如图16-27所示的一个树结构中的关系R、S、T以及U进行连接，并且希望在内存中保存所有的中间关系，直到不再需要它们为止。按照我们通常的假设，所有四个关系的连接的结果将会被产生这个结果的其他过程冲掉，因此那个关系不需要内存。依据被存储关系以及中间关系(例如$B(R)$或者$B(R \bowtie S)$)所需的块数，为以下的树给出所需内存的块数M的一个下限：

* a) 如图16-27(a)中的左深树。

b) 如图16-27(b)中的紧密树。

c) 如图16-27(c)中的右深树。

什么样的假设使我们得出这样的结论：一棵树一定比另一棵树使用较少的内存？

*! **习题16.6.7** 如果利用动态规划来对k个关系的连接选择顺序，我们必须填充多少个表的入

口?

16.7 物理查询计划选择的完成

我们已经分析了查询，将它转化为初步的逻辑查询计划，并采用16.3节中描述的变换来提高那个逻辑查询计划的性能。选择物理查询计划过程中的一部分是我们在16.5节中所讨论到的所有选项的枚举和代价估计。16.6节着眼于枚举问题、代价估计以及为几个关系的连接排序。经扩展，我们可以利用同样的技术为多个并、交或它们的任意结合/交换操作进行排序。

要将逻辑计划变成一个完整的物理查询计划仍需要几个步骤。我们在本节还必须包括的几个主要问题为：

1. 当算法选择没有作为某个开始步骤(如通过动态规划选择连接顺序)的一部分来执行时，实现查询计划操作的算法选择。

2. 关于何时中间结果将会被实体化（被整个创建并被存储在磁盘上）以及何时它们将会被流水线操作（只在主存中被创建，并且任何时候不需要被完整地保存）的决定。

3. 物理查询计划操作符的注释，它必须包括有关被存储关系的访问方法的细节以及相关代数操作的执行算法的细节。

859

我们不讨论全部操作符的算法选择。只选择性地讨论最重要的两个操作符：15.7.1节中的选择以及16.7.2节中的连接。然后，我们在16.7.3节到16.7.5节中将考虑流水线操作和物化的选择。16.7.6节将给出物理查询计划的注释。

16.7.1 选取选择方法

选取一个物理查询计划最重要的步骤之一是为每个选择操作符精选算法。在15.2.1节中我们提到$\sigma_C(R)$操作的执行，在这个操作中我们访问完整的关系R，并且看哪一个元组满足条件C。然后在15.6.2节中我们考虑了C是形式“属性等于常数”的概率，并且对那个属性我们有一个索引。如果是这样的话，我们可以找出满足条件C的元组，而不必查看R的全部。现在，让我们考虑这个问题的广义性，即我们有一个选择条件，它是几个条件的AND，其中一些条件是“属性等于常数”的形式或者是属性与常数之间的另外的比较，如<。

假设对几个属性并没有多维索引，则每个物理计划必须使用满足下列条件的一个或多个属性：

a) 有一个索引；并且

b) 与选择项之一的一个常量相比较。

然后我们利用这些索引来识别满足每一个条件的元组集。13.2.3节和14.1.5节讨论了在将元组从磁盘读出之前，我们如何利用通过这些索引得到的元组指针来找出满足所有条件的元组。

为简便起见，我们将不以这种方式来考虑利用几个索引。我们将讨论限制在以下的算法：

1. 利用形式$A\theta c$的一个比较，其中A是带有一个索引的一个属性，c是一个常量，θ是一个比较操作符例如“=”或者“<”。

2. 采用在15.1.1节中讨论的索引-扫描物理操作，对所有满足(1)中的比较的元组进行搜索。

3. 考虑在(2)中所选中的每个元组，看它是否满足剩下的选择条件。我们称执行这一步的物理操作符为Filter。它将用来选择元组的条件作为一个参数，很像相关代数学的σ操作符所做的那样。

除了这种形式的算法，我们还必须考虑一个算法，该算法没有利用任何索引，但它读取全部关系（采用表-扫描物理操作符）并且将每个元组传递给Filter操作符来检查选择条件是否满足。

860

我们通过估计读取每一个可能的选项的数据的代价，从诸多算法中决定用哪一个来执行给定的选择。为了比较不同算法的代价，我们不能继续使用经过简化的中间关系大小的代价估计。原因是我们现在正考虑逻辑查询计划的一个单一步骤的执行，并且中间关系与执行无关。

因此，我们将再次计算磁盘I/O，如在第15章中讨论算法及其代价一样。如前面一样，为简便起见，我们将只计算访问数据块的代价，而不考虑索引块。所需的索引块的数目一般远小于所需的数据块的数目，因此磁盘I/O代价的这个近似经常足够准确。

以下是对于不同的算法所估计的代价大小的概要。我们假设操作是$\sigma_C(R)$，其中条件C是一个或多个项目的AND。我们利用例子中的项目a=10和b<20来分别表示等条件和不等条件。

1. 表-扫描算法与一个过滤器步骤相结合的代价是：

(a) 如果R被聚簇，则为$B(R)$；

(b) 如果R没有被聚簇则为$T(R)$。

2. 选出一个相等项如a=10，该项存在关于属性a的一个索引，并且利用索引-扫描来找出匹配元组，然后将被检索的元组进行过滤来看它们是否满足全部条件C，这样的一个算法的代价是：

(a) 如果索引是聚簇的，则为$B(R)/V(R,a)$；

(b) 如果索引不是聚簇的，则为$T(R)/V(R,a)$。

3. 选出一个不等项如b<20，该项存在关于属性b的一个索引，并且利用索引-扫描来搜索匹配元组，然后将被检索的元组进行过滤来看它们是否满足全部条件C，这样的一个算法的代价是：

(a) 如果索引是聚簇的，则为$B(R)/3$[⊖]；

(b) 如果索引不是聚簇的，则为$T(R)/3$。

例16.35 考虑选择$\sigma_{x=1 \text{ AND } y=2 \text{ AND } z<5}(R)$，其中$R(x,y,z)$有以下的变元：$T(R)$=5000、$B(R)$=200、$V(R,x)$=100以及$V(R,y)$=500。另外，假设$R$是聚簇的，并且所有的$x$、$y$以及$z$上都有索引，但只
861 有z的索引是聚簇的。以下是执行这个选择的选项：

1. 表-扫描后进行过滤。其代价为$B(R)$，或者是200次磁盘I/O，因为R是聚簇的。

2. 使用x的索引以及索引-扫描来找出x=1的元组，然后利用Filter操作符来检测y=2以及z<5。由于有大约 $T(R)/V(R,x)$=50个元组的x=1，并且索引是非聚簇的，我们需要大约50次I/O。

3. 使用y的索引以及索引-扫描来找出y=2的元组，然后对这些元组进行过滤来检测x=1以及z<5。使用这个非聚簇索引的代价大约为 $T(R)/V(R,y)$，即10次磁盘I/O。

4. 使用z的索引以及索引-扫描来找出z<5的元组，然后对这些元组进行过滤来检测x=1以及y=2。磁盘I/O数大约为$B(R)/3$=67。

我们看到代价最小的算法是第三种，估计代价为10次磁盘I/O。因此，这个选择的最佳物理计划搜索所有y=2的元组，然后对另外两个条件进行过滤。 □

16.7.2 选取连接方法

在第15章中我们看到与各种连接算法相联系的代价。假设我们知道（或者可以估计）执行连接所需的缓冲区的容量，则我们可以对排序连接使用15.4.8节中的公式，对散列连接使用15.5.7节中的公式，以及对索引连接使用15.6.3节和15.6.4节中的公式。

然而，如果我们不能确定，或者不知道执行这个查询所需的缓冲区的容量（因为我们不知道DBMS同时在做什么其他的事情），或者如果我们没有重要的大小参数，比如$V(R,a)$的估计值，则仍有一些重要的原则使我们可以用来选择一个连接方法。同样的思想适用于其他二元操作例

⊖ 回想一下，我们假设典型的不等式仅检索1/3的元组，在16.4.3节中讨论了原因。

如并，以及完全关系、一元操作符、γ和δ。

- 一个方法是调用一趟连接，希望缓冲区管理器可以为连接分配足够的缓冲区，或者缓冲区管理器可以关闭，所以颠簸不是一个主要的代价。另一个方法（只用于连接，而不用于其他二元操作符）是选择一个嵌套循环连接，希望如果不能保证为左变元分配足够的缓冲区立刻装入内存，则变元将不会被分解成太多的片，并且结果连接将仍是合理的和有效的。
- 当以下两点中的任意一点成立时，一个排序连接就是一个好的选择：
  1. 一个或两个变元已经在它们的连接属性上排序；或者 862
  2. 对于同样的属性有两个或多个连接，例如

$$(R(a,b) \bowtie S(a,c)) \bowtie T(a,d)$$

其中在a上对R和S进行排序将会引起$R \bowtie S$的结果在a上被排序，并且在第二个排序连接中被直接使用。

- 如果有一个索引机会，例如一个连接$R(a,b) \bowtie S(b,c)$，其中R被期望是小的（也许是基于码的一个选择，它的结果是一个元组），并且有一个连接属性$S.b$的索引，则我们应该选择一个索引连接。
- 如果没有机会利用已经排序的关系或索引，并且需要一个多趟连接，则散列连接也许是最佳选择，因为它所需要的扫描次数取决于较小变元的大小而不是两个变元的大小。

16.7.3 流水线操作与实体化

我们将讨论与物理查询计划有关的最后一个主要的话题，即结果的流水线操作。执行一个查询计划的原始方法是对操作进行适当的排序（即直到位于一个操作下面的变元已经被执行

内存中的实体化

可以想像在流水性操作与实体化之间有一个中间方法，在该方法中，一个操作的整个结果在被传递给消费操作之前被存储在主存缓冲区（不是磁盘）。我们视操作的这种可能的模式为流水性操作，消费操作所做的第一件事情是在内存中对整个关系或者关系的大部分进行组织。这种行为的一个例子是一个这样的选择，该选择的结果是作为左变元被流水线操作送给几个连接算法中的一个，包括简单的一趟连接、多趟无序连接或者排序连接。

后操作才被执行），并且将每个操作的结果存储在磁盘上，直到它被另一个操作所需要。这个策略叫做实体化（materialization），因为每个中间关系在磁盘上被实体化。

执行一个查询计划的一个更巧妙、更有效的方法是一次同时进行几个操作。由一个操作产生的元组直接传递给使用它的操作，不需要将中间元组存储在磁盘上。这个方法叫做流水线操作，一般由一个迭代器网络（参见15.1.6节）执行，该网络的函数在适当的时候互相调用。由 863
于它节省了磁盘I/O，因此流水操作有一个明显的优点，但是也有相应的缺点。由于任何时候几个操作必须共享内存，就有可能必须选择有更高磁盘I/O要求的算法，或者将会发生颠簸，从而用掉所有由流水操作所节省的磁盘I/O，甚至可能更多。

16.7.4 一元流水线操作

一元操作——选择和投影——是流水线操作极好的候选对象。由于这些操作是一次一元组，我们从不需要有多个块的输入输出。图15-5给出了这种操作模式。

我们可以通过迭代器来执行一个一元流水线操作，正如15.1.6节中所讨论的。每次需要另一个元组时，流水线操作结果的消费者就调用GetNext()。在一个投影的情况下，只需要对元组源调用GetNext()一次，对那个元组进行适当的投影，并将结果返回给消费者。对于一个选择σ_C（从技术上是物理操作Filter(C)），也许需要对源调用GetNext()若干次，直到找到一个满足条件C的元组。图16-33给出了这个过程。

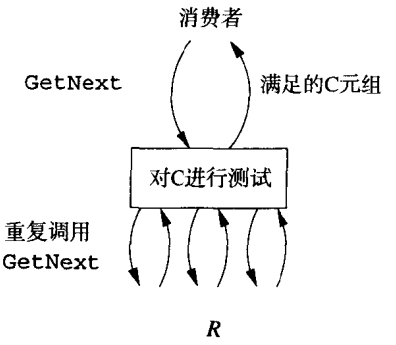


图16-33 使用迭代器执行一个选择的流水线操作

16.7.5 二元流水线操作

二元操作的结果也可以进行流水线操作。我们使用一个缓冲区将结果传递给消费者，一次一块。然而，计算结果和消费结果所需的其他缓冲区数目是不同的，它们取决于结果的大小以及查询中所包含的其他关系的大小。我们将使用一个扩展的例子来演示权衡和契机。

例16.36 考虑下列表达式的物理查询计划

864

$$(R(w,x) \bowtie S(x,y)) \bowtie U(y,z)$$

假设以下条件成立：

1. R占用5000块；S和U各占用10 000块。

2. 对某个k值，中间结果$R \bowtie S$占用k块。根据R和S中的x值的数目以及(w,x,y)元组与R的(w,x)元组和S的(x,y)元组的比值的大小，我们可以估计出k。然而，我们希望知道当k改变时会发生什么，所以我们令这个常数也可以变化。

3. 将两个连接作为散列连接来执行，或者是一趟连接或者是两趟连接，这取决于k。

4. 有101个可用的缓冲区。这个数目被人为地被设得较低。

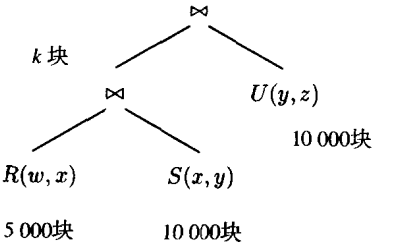


图16-34 例16.36的逻辑查询计划和变元

图16-34是带有关键参数的表达式的一个示意图。

首先，考虑连接$R \bowtie S$。在主存中没有任何关系存在，所以我们需要一个两趟散列连接。为了将较小关系R的桶限制为每个100块，我们至少需要50个桶[⊖]。如果我们正好使用50个桶，则散列连接$R \bowtie S$的第二趟将使用51个桶，其中的50个桶用于$R \bowtie S$的结果与U的连接。

现在，假设$k \leqslant 49$；即$R \bowtie S$的结果最多占用49块。那么我们可以将$R \bowtie S$的结果流水线操作进入49个缓冲区，将它们进行组织以便作为一个散列表来查看，并且利用一个缓冲区来依次读入U的每一块。我们可以将第二趟连接作为一趟连接来执行。磁盘I/O的总数为：

865

a) 45 000，执行R和S的两趟散列连接。

b) 10 000，在$(R \bowtie S) \bowtie U$的一趟散列连接中读入U。

总共为55 000次磁盘I/O。

⊖ 按照惯例，我们假设元组在装入所有桶时，每个桶都正好装满。如果有意外，肯定会有，那么偶尔需要一些额外的缓冲区，我们依靠缓冲区管理器通过可能是临时移动一些缓冲区到磁盘上的交换空间来分配。然而，不考虑交换的磁盘I/O的额外代价，因为我们认为这些代价在整个代价中只占一小部分。

现在，假设$k>49$，但是$k\leqslant 5000$。我们仍可以将$R\bowtie S$的结果进行流水线操作，但需要使用另一个策略，在这个策略中，该关系与U进行一个50个桶的两趟散列连接。

1. 在开始$R\bowtie S$之前，我们将U装进每个200块的50个桶。

2. 接下来，像前面一样用51个桶对R和S进行一个两趟散列连接，但是当$R\bowtie S$的每个元组产生时，我们将它放入用于形成50个桶来将$R\bowtie S$和U进行连接的50个剩余缓冲区中的一个。当这些缓冲区装满时就写入磁盘，对一个两趟散列连接来说，这是很正常的。

3. 最后，我们一桶一桶地将$R\bowtie S$和U进行连接。由于$k\leqslant 5000$，$R\bowtie S$的桶的大小最多为100块，因此这个连接是可行的。U的桶的大小为200块不是一个问题，因为在桶的一趟连接中，我们正在利用$R\bowtie S$的桶作为建立关系，而U的桶作为探针关系。

这个流水线操作连接的磁盘I/O数为：

a) 20 000，用于读取U并将它的元组写入桶。

b) 45 000，用于执行$R\bowtie S$的两趟散列连接。

c) k用于写出$R\bowtie S$的桶。

d) k+10 000用于读取最终连接中的$R\bowtie S$和U的桶。

全部代价为75 000+2k。注意，当k从49增长到50时有明显的不连续性，因为我们必须将最终连接从一趟改变为两趟。在实际应用中，该代价不会如此剧烈地变化，因为即使没有足够的缓冲区以及发生小量的抖动，我们也可以使用一趟连接。

k的范围	流水线或物化	最后的连接的算法	总的磁盘I/O数
$k\leqslant 49$	流水线	一趟	55 000
$50\leqslant k\leqslant 5000$	流水线	50个桶，两趟	75 000+2k
$5000<k$	物化	100个桶，两趟	75 000+4k

图16-35 作为$R\bowtie S$大小的函数的连接算法的代价

最后，让我们考虑当$k>5000$时会有什么情况发生。现在，如果$R\bowtie S$的结果被流水线操作，则我们不能在所得到的50个桶中执行一个两趟连接。我们可以使用一个三趟连接，但是那样的话，一个变元的每块需要额外的两次磁盘I/O，或者是20 000+2k个更多的磁盘I/O。如果改为不对$R\bowtie S$进行流水线操作，则我们可以做得更好。现在，连接计算的要点是这样的：

1. 用一个两趟散列连接计算$R\bowtie S$，并且将结果存储在磁盘。

2. 将$R\bowtie S$和U进行连接，仍然使用一个两趟散列连接。注意由于$B(U)$=10 000，我们可以用100个桶执行一个两趟散列连接，而不考虑k值的大小。从技术上说，如果我们决定将U作为散列连接的建立关系，那么U应该作为其在图16-34中的连接的左变元。 866

这个算法的磁盘I/O数为：

a) 45 000用于R和S的两趟连接。

b) k用于将$R\bowtie S$存储在磁盘上。

c) 30 000+3k用于U和$R\bowtie S$的两趟散列连接。

全部代价为75 000+4k，它要小于在最后一步进行一个三趟连接时的代价。图16-35所示的表总结了这三个完整的算法。 □

16.7.6 物理查询计划的符号

我们已经看到了许多可以用于形成一个物理查询计划的操作符的例子。一般来说，逻辑计划的每个操作符成为物理计划的一个或多个操作符，逻辑计划的叶子(存储的关系)成为物理计划中适用于那个关系的一个扫描操作符。另外，当实体化结果被其消费者访问时，实体化由一个Store操作符指示，该操作符应用于将要被物化的中间结果，后跟一个合适的扫描操作符(常为TableScan，因为中间关系没有索引，除非明确地创建一个)。然而，为简便起见，在物理查询计划树中我们将指出某一中间关系被一条双线实体化，该线与在那个关系与其消费者之间的边交叉。假定所有其他边表示元组的提供者和消费者之间的流水线操作。

我们现在把在物理查询计划中经常发现的各种不同的算法进行分类。不像关系代数，它的符号一般是标准的。对物理查询计划来说，每个DBMS将利用其本身的内部符号。

叶子的操作符

作为逻辑查询计划的一个叶子操作数的每个关系R将被一个扫描操作符替代。这些操作符
867 为：

1. TableScan(R)：以任意顺序读入所有存放R的元组的块。

2. SortScan(R, L)：按照顺序读入R的元组，并按列表L中的属性进行排列。

3. IndexScan(R,C)：这里，C是$A\theta c$的一个条件，其中A是R的一个属性，θ是一个比较操作符例如=或者<，c是一个常量。可以通过属性A上的一个索引来访问R的元组。如果比较操作符θ不是=，则索引必须支持范围查询，例如B树。

4. IndexScan(R,A)：这里A是R的一个属性。整个关系R通过$R.A$上的一个索引被检索。这个操作符看起来像TableScan，但是，如果R没有被聚簇以及/或者它的块不容易被找到，则在某些环境下该操作符也许会更有效。

选择的物理操作符

当R是一个存储关系时，关系R的访问方法常常与逻辑操作符$\sigma_C(R)$进行联合或部分联合。其他那些变元不是存储关系或者没有一个合适的索引的选择，将被相应的我们曾经称做Filter的物理操作符替代。我们在16.7.1节中讨论过对一个选择实现方法进行选取的策略。用于不同选择实现方法的注意事项有：

1. 我们可以简单地用操作符Filter(C)替代$\sigma_C(R)$。如果没有R的索引，或者没有一个条件C提到的属性的索引，则该选择是有意义的。如果R，即选择操作的变元，实际上是一个被流水线操作进入该选择的中间关系，则除了Filter以外，不需要其他的操作符。如果R是一个存储关系或实体化关系，则我们必须使用一个操作符，TableScan或者SortScan(R, L)，来访问R。如果$\sigma_C(R)$的结果以后将被传递给一个需要其变元被排序的操作符，则我们也许偏向于排序-扫描。

2. 对于某个其他条件D，如果条件C能够表示为$A\theta c$ AND D，并且有一个$R.A$的索引，则我们可以：

(a) 使用操作符IndexScan(R, $A\theta c$)来访问R；

(b) 使用Filter(D)替代选择$\sigma_C(R)$。

物理排序操作符

一个关系的排序可能发生在物理查询计划中任何一点。我们已经介绍了SortScan(R,L)操作符，该操作符读取一个存储关系R并且根据属性L的列表对它进行排序。当我们在诸如连
868 接或分组的操作中采用一个基于排序的算法时，有一个根据某个属性列表对变元进行排序的初

始阶段。一般使用一个显式的物理操作符Sort(L)来对没有存储的一个操作数关系进行这一排序。如果是由于在原始查询中的ORDER BY子句使得结果需要被排序时，则该操作符也可以用于物理查询计划的顶层，因此它与5.4.6节中的τ操作符有着同样的作用。

其他关系代数操作

所有其他操作被一个适当的物理操作符替代。可以给予这些操作符一些标识，该标识指明：

1. 被执行的操作，例如，连接或分组。
2. 必要的参数，例如，一个θ连接中的条件或一个分组中的元素列表。
3. 算法的一个通用策略：基于排序、散列或者在一些连接中基于索引。
4. 关于要用到的趟数的一个决定：一趟、两趟或者多趟（递归，对数据使用尽可能多趟）。该选择也可以保留直到运行时。
5. 操作所需的预期缓冲区数。

例16.37 图16-36所示为在例16.36中在$k > 5000$的情况下所生成的物理计划。在这个计划中，我们通过表扫描来访问三个关系中的每一个。我们对第一个连接使用一个两趟散列连接，对它进行实体化，并且对第二个连接使用一个两趟散列连接。实体化的双线符号的暗示顶部连接的左变元也可以通过一个表扫描获得，并且使用Store操作符来存储第一个连接的结果。 869

相反，如果$k \leqslant 49$，则在例16.36中所生成的物理计划如图16-37所示。注意第二个连接使用了一个不同的趟数，不同的缓冲区数以及一个被流水化而没有被实体化的左变元。 □

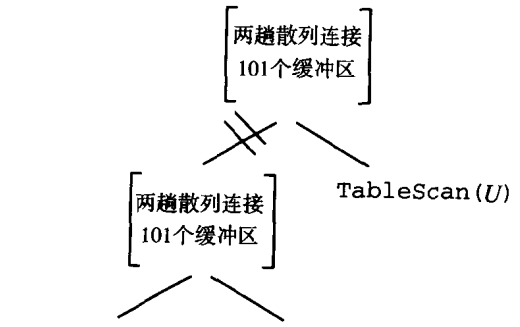


图16-36 例16.36的一个物理计划

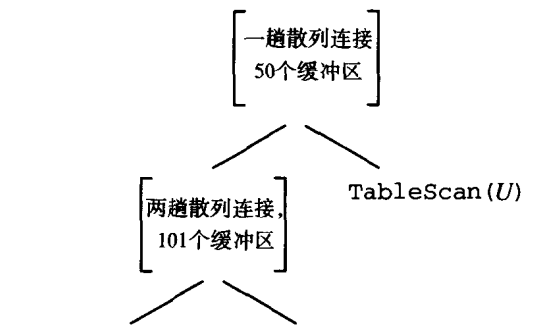


图16-37 在$R \bowtie S$非常小的情况下的另一个物理计划

例16.38 考虑例16.35中的选择操作，该例中我们决定最佳选择是使用y的索引来寻找$y=2$的那些元组，然后用其他条件$x=1$和$z<5$来检查这些元组。如图16-38所示为物理查询计划。叶子表示R将通过y的索引被访问，只找出$y=2$的元组。Filter操作符要求我们通过进一步选择所检索到的并且满足$x=1$以及$z<5$的元组来完成选择操作。

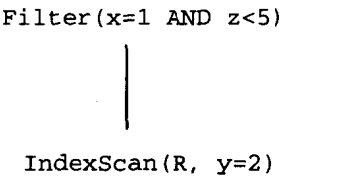


图16-38 使用最恰当索引的一个选择

□

16.7.7 物理操作的顺序

我们关于物理查询计划的最后一个主题是操作的顺序。通常，物理查询计划表示为一棵树，而树暗示了一些关于操作顺序的东西，因为数据必须沿着树流动。然而，由于紧密树可以有既

870 不是祖先节点也不是另一个点的后代节点的内部节点，内部节点估值的顺序可能并不总是清楚的。另外，由于可以使用迭代器来执行流操作方式的操作，有可能不同节点的执行时间会重叠，因此节点顺序的概念就没有意义了。

如果以明显的“先存后取”（store-and-later-retrieve）方式执行实体化，并且以迭代器执行流水线操作，则我们可以建立事件的固定序列，由此可以执行一个物理计划的每一个操作。以下的规则总结了在一个物理查询计划中隐式的事件排序：

1. 在表示物化的边上将树分解为子树。子树将一次一个地被执行。

2. 以从下到上、从左到右的顺序依次执行各子树。精确地说，执行整棵树的一个前序趟历。以前序趟历从子树中退出的顺序对子树进行排序。

3. 使用一个迭代器网络来执行每一棵子树的所有节点。因此，在一棵子树中所有节点被同时执行，并用`GetNext`调用它们中间的操作符来决定事件的确切顺序。

根据这个策略，查询优化器现在能够为查询生成可执行的代码，对于查询也许是一个函数调用序列。

16.7.8 习题

习题16.7.1 考虑一个关系$R(a,b,c,d)$，该关系有一个a的聚簇索引以及对每一个其他属性的非聚簇索引。相关变元为:$B(R)=1000$、$T(R)=5000$、$V(R,a)=20$、$V(R,b)=1000$、$V(R,c)=5000$以及$V(R,d)=500$。对于下列各项选择，给出最佳查询计划（索引扫描或者表扫描，然后进行一个过滤步骤）以及磁盘I/O开销：

* a) $\sigma_{a=1 \text{ AND } b=2 \text{ AND } d=3}(R)$

b) $\sigma_{a=1 \text{ AND } b=2 \text{ AND } c\geqslant 3}(R)$

c) $\sigma_{a=1 \text{ AND } b\leqslant 2 \text{ AND } c\geqslant 3}(R)$

! **习题16.7.2** 用$B(R)$、$T(R)$、$V(R,x)$和$V(R,y)$来表示以下关于R上执行一个选择操作的费用的条件：

* a) 通过x的一个非聚簇索引和使x等于一个常量的条件来使用索引扫描，要比通过y的一个非聚簇索引以及使y等于一个常量的条件来使用索引扫描好。

b) 通过x的一个非聚簇索引和使x等于一个常量的条件来使用索引扫描，要比通过y的
871 一个聚簇索引以及使y等于一个常量的条件来使用索引扫描好。

c) 通过x的一个非聚簇索引和使x等于一个常量的条件来使用索引扫描，要比通过y的一个聚簇索引以及对某些常量C使$y>C$的条件来使用索引扫描好。

习题16.7.3 如果关系R的大小不是5000块而是以下数值时，在例16.36中关于何时进行流水线操作的结论将会有何变化：

a) 2000块。

! b) 10 000块。

! c) 100块。

! **习题16.7.4** 假设我们想以给定的顺序计算$(R(a,b)\bowtie S(a,c))\bowtie T(a,d)$。我们有$M=101$的主内存缓冲区，并且$B(R)=B(S)=2000$。由于连接属性$a$对两个连接来说是相同的，因此我们决定通过一个两趟排序连接来执行第一个连接$R\bowtie S$，并且对第二个连接使用合适的趟数，先将T分为一些在a上排序的子列，再将它们与连接$R\bowtie S$的经过排序和流水操作的元组流进行复合。对于T与$R\bowtie S$连接，我们应该为$B(T)$选择什么值：

* a) 一个一趟连接；例如，我们将T读入内存，当其元组被生成时，将它们与$R\bowtie S$的元

组进行比较。

b) 一个两趟连接；例如，当生成$R \bowtie S$的元组时，我们为T创建经过排序的子列，并且为每一个排序子列在内存中保留一个缓冲区。

16.8 小结

- *查询的编译*：编译将一个查询变成一个物理查询计划，即一个能够由查询执行引擎执行的操作序列。查询编译的主要步骤是分析、语义检查、选择优先的逻辑查询计划（代数表达式）以及由此生成最佳物理计划。
- *分析器*：处理一个SQL查询过程中的第一个步骤是对其进行分析，就像任何编程语言中需要对源代码所做的一样。分析的结果是与SQL结构相应的带有节点的一棵分析树。
- *语义检查*：一个预处理器检查分析树，检查其属性、关系名以及有意义的类型，并且在当几个关系中有相同属性名时解决属性参照的问题。
- *转换为逻辑查询计划*：查询处理器必须将经过语义检查的分析树转换为一个代数表达式。转换为关系代数工作的绝大部分是直接的，但是子查询是一个问题。通常的方法是引进一个两变元的选择，将子查询放入选择条件中，然后使用恰当的覆盖常见的特殊情况的转换。 872
- *代数转换*：使用代数转换，有许多方法可用来将一个逻辑查询计划转换为一个更好的计划。16.2节罗列了主要的几种方法。
- *选择一个逻辑查询计划*：查询处理器必须选择最有可能成为一个有效物理计划的查询计划。除了使用代数转换，将结合和交换操作符，尤其是连接操作符，进行分组是有利的，所以物理查询计划可以选择最佳顺序并对这些操作进行分组。
- *估计关系的大小*：当选择最佳逻辑计划时，或者当对连接或其他可结合—交换操作进行排序时，我们使用中间关系的估计大小，代替最终选择的物理计划的真正运行时间或磁盘I/O开销。对于关系的大小（元组数）和每个关系的每个属性的不同值的数目，不管已知的或是估计的都帮助我们获得中间关系大小的较好估计。
- *直方图*：某些系统保持一个给定属性的值的直方图。该信息可以用来获得中间关系大小的估计，这个估计要好于本章所提到的简单方法。
- *基于代价的优化*：当选择最佳物理计划时，必须能够估计每一个可能计划的代价。使用不同的策略来生成所有或某些实现一个给定逻辑计划的可能的物理计划。
- *计划枚举策略*：检索最佳物理计划的通用方法包括动态规划（对给定逻辑计划的每一个子表达式，制作最佳计划表）、Selinger风格的动态规划（包括作为表的一部分的结果的排序，该表为每一个排序的以及未排序的结果给出最佳计划）、贪婪方法（做出一系列局部最优的决定，从已有的物理计划中选择），以及分枝和界定（只枚举那些还不能立即知道是否比现有的最佳计划更差的计划）。
- *左深连接树*：当为几个关系的连接选取一个成组和顺序时，通常将搜索限制在左深树，即在二叉树中只沿着左侧的边向下延伸，留下右侧的叶子。这种形式的连接表达式将有利于生成有效的计划，并且还大大限制了需要考虑的物理计划的数目。 873
- *选择的物理计划*：如果可能的话，一个选择应该被分解成要进行选择操作的关系的索引扫描（通常使用一个索引的属性等于一个常量的条件），接着进行一个过滤操作。过滤器检查通过索引扫描搜索到的元组，并且只传递那些满足部分选择条件的元组，而不是那

些基于索引扫描的元组。

- 流水操作与物化：理想的情况下，每个物理操作符的结果被另一个操作符消费，结果在主存中的两者之间传递（“流水线操作”），也许使用一个迭代器来控制数据从一个流向另一个。然而，有时存储（“物化”）一个操作符的结果是有利的，它节约了另一个操作符所需主存的空间。因此，物理计划的生成应该考虑中间关系的流水线操作以及实体化。

16.9 参考文献

在第15章的文献目录的注释中所提到的综述中包含与查询编译有关的资料。另外，我们推荐综述[1]，它包含与商业系统的查询优化有关的资料。

文献[4]、[5]和[3]是关于查询优化的三篇最早的研究论文。文献[6]是另一篇较早的研究论文，它把沿树下推选择的思想与用于连接顺序选择的贪婪算法结合在一起。文献[2]第一次介绍了“Selingr风格优化”并描述了系统R优化器，这是当今在查询优化方面作出最大努力的著作之一。

1. G. Graefe (ed.), *Data Engineering* **16**:4 (1993), special issue on query processing in commercial database management systems, IEEE.

2. P. Griffiths-Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, “Access path selection in a relational database system,” *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1979), pp. 23–34.

3. P. A. V. Hall, “Optimization of a single relational expression in a relational database system,” *IBM J. Research and Development* **20**:3 (1976), pp. 244–257.

4. F. P. Palermo, “A database search problem,” in: J. T. Tou (ed.) *Information Systems COINS IV*, Plenum, New York, 1974.

5. J. M. Smith and P. Y. Chang, “Optimizing the performance of a relational algebra database interface,” *Comm. ACM* **18**:10 (1975), pp. 568–579.

6. E. Wong and K. Youssefi, “Decomposition — a strategy for query processing,” *ACM Trans. on Database Systems* **1**:3 (1976), pp. 223–241.

874

第17章 系统故障对策

从本章开始，我们集中考虑数据库管理系统中控制数据访问的那些部分。我们要考虑的主要有两大问题：

1. 当故障发生时数据必须受到保护。本章讲述支持回复性的相应技术，回复性指系统发生某些故障时数据的完整性。

2. 数据不能仅仅因为几个无错查询或数据库的更新而受到破坏。这一问题在第18章和第19章讨论。

支持回复性的主要技术是日志，日志以一种安全的方式记录数据库变更的历史。我们将讨论三种不同类型的日志，分别称为“undo”、“redo”和“undo/redo”。我们还将讨论恢复，恢复是故障发生后使用日志重建数据库所做更新的过程。日志和恢复的一个重要方面是要避免需要追溯到很久以前的日志这一情形。因此，我们将要学习称为“检查点”的重要技术，它限制了恢复时必须检查的日志长度。

在最后一节中，我们讨论“备份”（archiving），它使得数据库不仅能经受暂时的系统故障，而且可以在整个数据库都丢失的情况下使数据库存留下来。在这种情况下，我们必须依赖于数据库的一个近期拷贝（备份）以及所有幸存的日志信息，将数据库恢复到最近某个时刻的状态。

17.1 可回复操作的问题和模型

我们从可能发生的各种问题及数据库管理系统针对这些问题能做什么、该怎么着手开始来讨论如何对付故障。首先我们主要讨论“系统故障”或“崩溃”，日志和恢复机制的设计正是 用来修复这几类错误。在17.1.4节我们还将引入缓冲区管理模型，它是对系统错误恢复进行的所有讨论的基础。下一章我们讨论几个事务并发访问数据库时仍需要这个模型。

17.1.1 故障模式

在查询和更新数据库时可能发生多种多样的问题。问题的范围从键盘输入错误数据到存储数据库的磁盘所在房间发生的爆炸。下面列出最重要的故障模式及DBMS对这些故障所能采取的行动。

错误数据输入

有的数据错误是不可能被检测到的。例如，如果某个职员把你的电话号码中的一位输错了，号码看起来可能仍然是你的。另一方面，如果职员遗漏了你的电话号码中的一位，数据就显然是错的，因为它不具有电话号码应有的格式。

现代的DBMS提供许多软件机制，以捕捉那些可被检测的输入错误。例如，SQL标准以及所有流行的SQL实现中都为数据库设计者提供了在数据库模式中引入约束的方法，如码约束、外码约束和值的约束（例如，电话号码长度必须是10位）。触发器是一旦某种类型的更新（如在关系R中插入一个元组）发生就执行的程序，用来检查刚刚进入的数据是否满足数据库设计者认为它应满足的约束。

介质故障

磁盘的局部故障，即只改变一位或少数几位的故障，通常能通过与磁盘扇区相关联的奇偶校验检测到，正如我们在11.3.5节讨论的那样。磁盘的主要故障，通常是磁头损坏，使整个磁盘都不再能被访问，这时通常用下述方法中的一种或两种来处理：

1. 采用11.7节讨论的某种RAID模式，这样,丢失的磁盘就可以被恢复。

2. 维护一个备份，即数据库在诸如磁带或光盘这样的介质上的一个拷贝。备份周期性地创建，可以是完全的或增量式的，存储在与数据库自身相距大于某个安全的距离的地方。我们将
876 在17.5节讨论备份。

3. 我们可以不采用备份，而是联机保存数据库的冗余拷贝，这些拷贝分布在几个节点上。维护这些拷贝一致性的机制我们将在19.6节讨论。

灾难性故障

这类故障包括容纳数据库的介质完全毁坏的多种情况，例如爆炸或大火以及对数据库站点的非法入侵，对此RAID也无能为力，因为所有的数据盘及其奇偶校验盘同时失去作用。但是，用于防止介质故障的其他方法（备份和冗余分布式拷贝）也可以用来防止灾难性故障。

系统故障

查询和修改数据库的进程称为事务。事务和其他任何程序一样执行一系列步骤；通常这些步骤中的一部分将修改数据库。每个事务有一个状态，代表该事务中到目前为止已发生了什么。状态包括所执行事务代码中的当前位置和所有以后将会需要的事务局部变量的值。

*系统故障*是导致事务状态丢失的问题。典型的系统故障包括掉电和软件错误。要明白诸如断电之类的问题为什么会导致状态的丢失，请注意事务和其他任何程序一样，它的各个步骤最初是在主存中进行的。和磁盘不同，内存是“易失性的”，正如我们在11.2.6节讨论的那样。也就是说，掉电会导致主存中的内容消失，而磁盘上的（非易失性的）数据完好无损。类似地，一个软件错误可能覆盖主存的一部分，也许会修改程序状态的值。

当主存丢失时，事务状态就丢失了；也就是说，我们不再能确定事务的哪些部分(包括它对数据库的修改)已经进行。重新运行事务不一定能解决问题。例如，如果事务必须将数据库中的某个值加1，我们并不知道到底是否应该再重复加1。解决由系统错误所引起问题的基本方法是在分离的、非易失性的日志中记录所有数据库更新，必要时加上恢复。但是，要保证这样的日志记载能以一种不受故障干扰的方式进行，其机制非常复杂，正如我们将在17.2节看到的那样。

17.1.2 关于事务的进一步讨论

8.6节中从SQL程序员的角度介绍了“事务”的概念。在我们进入对数据库回复性和故障恢
877 复的讨论之前，需要进一步更详细地讨论有关事务的基本概念。

事务是数据库操作的执行单位。例如，如果我们正在向一个SQL系统提交即席的命令，那么每一个查询或数据库更新语句就是一个事务。当我们使用嵌入式SQL接口时，事务的范围就由编程者控制，它既可以包括若干查询和更新，又可以包括在宿主语言中进行的操作。在典型的嵌入式SQL系统中，一旦对数据库进行的操作执行，事务就开始，而事务的结束使用显式的COMMIT或ROLLBACK（“终止”）命令。

正如我们将在17.1.3节中讨论的那样，事务必须原子地执行，即全做或全不做，并且似乎它是在某个时刻瞬间完成的。保证事务正确执行是*事务管理器*的工作，这一子系统完成的功能包括：

1. 给日志管理器（下面讲述）发信号，使必需的信息能以“日志记录”的形式存储在日志中。

2. 保证并发执行的事务不会相互干扰，否则将导致错误（“调度”；见18.1节）。

事务管理器及其相互作用如图17-1所示。事务管理器将关于事务动作的消息传给日志管理器，将关于何时可以或必须将缓冲区拷回磁盘的消息传给缓冲区管理器，并传消息给查询处理器使之能执行查询以及其他构成事务的数据库操作。

日志管理器维护日志。它必须同缓冲区管理器打交道，因为日志的空间最初存在于主存缓冲区中，在一定的时候这些缓冲区必须被拷贝到磁盘上。日志和数据一样占用磁盘上的空间，正如我们在图17-1中表示的那样。 878

最后我们在图17-1中给出恢复管理器的作用。当系统崩溃时，恢复管理器就被激活。它检查日志并在必要时利用日志恢复数据。和平常一样，对磁盘的访问是通过缓冲区管理器来进行的。

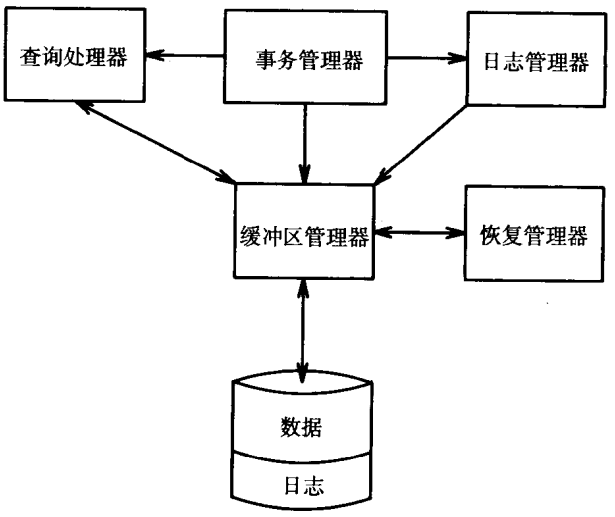


图17-1 日志管理器与事务管理器

17.1.3 事务的正确执行

在讨论如何纠正系统的错误之前，我们必须理解事务“正确”执行意味着什么。首先假设数据库由“元素”组成。我们并不打算精确定义什么是“元素”，而只是指出元素具有一个值并且能被事务访问或修改。不同的数据库系统使用不同的元素概念，但它们常常是从以下各项中选择一个或多个：

1. 关系，或其面向对象的等价概念：类外延。
2. 磁盘块或页。
3. 关系的单个元组，或其面向对象的等价概念：对象。

在接下来的例子中，读者可以将数据库元素设想为元组，在许多例子中甚至可以设想为整数。但是，实践中选择选项(2)（磁盘块或页）作为数据库元素有很多很好的理由。使用这种方式，缓冲区内容成为单个元素，使我们可以避开一些有关日志和事务的严重问题，这些问题在我们学习不同技术的过程中会不断发现。避免大于磁盘块的数据库元素还可以避开崩溃发生时元素的部分（但并非全部）已被放到非易失性存储中的情况。

数据库具有其状态，即对应于其各个元素的取值[⊖]。直觉上，我们认为某些状态是一致的，而另一些是不一致的。一致的状态满足数据库模式的所有约束，例如键码约束和值的约束。但

⊖ 我们不能混淆数据库状态与事务状态；后者是事务局部变量的取值，而不是数据库元素的取值。

是，一致的状态还必须满足数据库设计者心目中的隐式约束。隐式约束可能通过作为数据库模式一部分的触发器来维护，但也可能仅仅通过有关数据库的策略说明或与用于更新的用户界面相关的警告信息来维护。

关于事务的一个基本假设是：

• 正确性原则：如果事务在没有其他任何事务和系统错误的情况下执行，并且在它开始执行

正确性原则可信吗?

如果数据库事务可以是在某个终端上提交的即席修改命令，而提交命令的人并不知道数据库设计者心目中隐式的约束，那么所有事务都将数据库从一个一致的状态转换到另一个一致的状态，这一假设合理吗？显式的约束由数据库体现，所以任何违背这些约束的事务都会被系统拒绝，因而根本不会改变数据库。而对于隐式的约束，在任何情况下都没人能准确地刻画它们。我们认为正确性原则合理的理由是，如果一个人被赋予修改数据库的权利，那么他也有权决定隐式的约束是什么。

879 时数据库处于一致的状态，那么当事务结束时数据库仍然处于一致的状态。

正确性原则的反面表述构成了本章所讨论的日志技术以及第18章讨论的并发控制机制的动机。这一反面表述包括两点：

1. 事务是原子的；即事务必须作为整体执行或根本不执行。如果仅有事务的部分被执行，那么很有可能所产生的数据库状态是不一致的。

2. 事务的同时执行可能导致状态的不一致，除非我们设法控制它们之间的相互影响，正如我们将在第18章中所做的那样。

17.1.4 事务的原语操作

我们现在详细考虑事务如何同数据库交互。有三个地址空间，它们在重要的方面相互影响：

1. 保存数据库元素的磁盘块空间。

2. 缓冲区管理器所管理的虚存或主存地址空间。

3. 事务的局部地址空间。

事务要读取数据库元素，该元素首先必须被取到主存的一个或多个缓冲区中，除非它已经在那里。接下来，缓冲区中的内容可以被事务读到其局部地址空间中。事务为数据库元素写入一个新值要沿着与此相反的路径。事务首先在自己的局部空间中创建新值，然后该值被拷贝到
880 适当的缓冲区中。

缓冲区可能是也可能不是立即拷贝到磁盘；这一决定通常是缓冲区管理器的任务。正如我们即将看到的那样，使用日志来保证系统故障发生时的回复性，其最基本的步骤之一是在适当的时候强制缓冲区管理器将缓冲区中的块写回磁盘。但是，为了减少磁盘I/O次数，数据库系统可以允许并且将会允许更新只存在于易失性的主存中，至少在某些时间段中以及适当的条件组合下。

为了研究日志算法和其他事务管理算法的细节，我们需要一种记法来描述所有使数据在地址空间之间移动的操作。我们将使用的原语包括：

1. `INPUT(X)`：将包含数据库元素X的磁盘块拷贝到内存缓冲区。

2. `READ(X,t)`：将数据库元素X拷贝到事务的局部变量t。更准确地说，如果包含数据库元素X的块不在内存缓冲区中，则首先执行`INPUT(X)`。接着将X的值赋给局部变量t。

3. `WRITE(X,t)`：将局部变量t的值拷贝到内存缓冲区中的数据库元素X。更准确地说，如

果包含数据库元素X的块不在内存缓冲区中，则首先执行INPUT(X)。接着将t的值拷贝到缓冲区中的X。

4. OUTPUT(X)：将包含X的缓冲区拷贝回磁盘。

查询处理与事务中的缓冲区

如果你已经习惯于有关查询处理的章节中对缓冲区使用的分析，或许会发现在这里我们的视点有所改变。在第15章和第16章中，我们对缓冲区的兴趣主要在查询求值中用于计算临时关系时。这是缓冲区的一个重要用途，但由于临时值从不需要保存，因此这些缓冲区的值通常是不记录日志。另一方面，包含从数据库中检索出的数据的缓冲区需要保存其中的值，特别是事务更新这些值时。

只要数据库元素存在于单个磁盘块中，也就会存在于单个缓冲区中，上述操作就是有意义的。当数据库元素是块时正是这样的情况。只要关系模式不允许元组比一个磁盘块所能提供的空间大，那么当数据库元素是元组时也满足这样的情况。如果数据库元素占据多个块，那么我们把元素中每个具有块大小的部分本身看做一个元素。我们将使用的日志机制保证只有在写X是原子的时候事务才能完成；即要么X的所有块都被写到磁盘上，要么都没有写到磁盘上。因此，我们在对日志进行讨论的整个过程中将假定：

- 数据库元素的大小不超过一个块。

请注意发出这些命令的DBMS成分是不同的，这很重要。READ和WRITE由事务发出。INPUT和OUTPUT由缓冲区管理器发出，尽管在某些情况下OUTPUT也能由日志管理器发起，正如我们将看到的那样。 881

例17.1 为了明白上述原语操作如何同事务所要做的联系起来，我们来看具有两个元素A和B的数据库，这两个元素要满足的约束是在任何一致的状态中它们的值相等[⊖]。

事务T在逻辑上由下述两步构成：

```
A := A*2;
B := B*2;
```

注意，如果数据库惟一的一致性要求是$A = B$，而且T从一个正确的状态开始，在其完成动作的过程中没有其他事务以及系统故障的干扰，那么最终的状态也必然是一致的。也就是说，T将两个相等的元素加倍，得到两个新的相等的元素。

T的执行包括从磁盘读A和B，在T的局部地址空间中执行算术运算，以及将A和B的新值写入其缓冲区中。我们可以将T表述为六个相关步骤的序列：

```
READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t);
```

此外，缓冲区管理器最终将执行OUTPUT步骤，将这些缓冲区写回磁盘。图17-2给出了T的原语步骤，其后跟随着缓冲区管理器的两个OUTPUT命令。我们假设最初$A = B = 8$。图中给出了每一步中A和B的内存值、磁盘拷贝的值以及事务T地址空间中局部变量t的值。 882

第一步，T读A，如果A的块还不在缓冲区中，这将导致对缓冲区管理器的INPUT(A)命令。

⊖ 读者可能会问为什么我们要自寻烦恼地使用两个不同的元素而且约束它们必须相等，而不是只维护一个元素，这是很有道理的。但是，这个简单的数量约束抓住了很多更加现实的约束的核心，例如，一次航班售出的座位数多于飞机上的座位数不能超过10%，一个银行贷款余额总和必须等于该银行的总债务。

READ命令还将A的值拷贝到T地址空间中的局部变量t。第二步将t加倍，对A没有任何影响，不管是缓冲区中还是磁盘上。第三步将t写到缓冲区的A中；它不会影响磁盘上的A。接下来的三步对B做同样的事，而最后两步将A和B拷贝到磁盘。

不难发现，只要所有这些步骤都执行，数据库的一致性就能得到保持。如果在执行OUTPUT(A)前发生了系统故障，那么磁盘上存储的数据库不会受到任何影响，就仿佛T从未发生，因而一致性得到保持。但是，如果系统故障在OUTPUT(A)后而在OUTPUT(B)前发生，那么数据库就会处于不一致的状态。我们不能防止这种情况的发生，但可以安排当这种情况发生时对问题进行修复——或者A和B都被重置为8，或者二者都更新为16。 □

动 作	t	内存A	内存B	磁盘A	磁盘B
READ(A,t)	8	8		8	8
t := t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
READ(B,t)	8	16	8	8	8
t := t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

图17-2 一个事务的步骤及其对内存和磁盘的影响

17.1.5 习题

习题17.1.1 假设数据库上的一致性约束是$0 \leqslant A \leqslant B$。判断以下各事务是否保持一致性。

* a) `A := A+B; B := A+B;`

b) `B := A+B; A := A+B;`

c) `A := B+1; B := A+1;`

习题17.1.2 对习题17.1.1中的每个事务，在计算中加入读写动作，并给出各步骤对主存和磁盘产生的影响。假设最初$A = 5$且$B = 10$。此外，请说明当OUTPUT动作顺序恰当时，是否有可能即使在事务执行过程中发生了故障，一致性仍能得到保持。

883

17.2 undo日志

我们现在从日志作为保证事务原子性的一种方法方面开始研究日志，事务的原子性指从数据库来看，事务要么作为一个整体被执行，要么根本不执行。日志是日志记录的一个序列，每个日志记录记载有关某个事务已做的事的某些情况。几个事务的行为可以是“交错的”，因此可能是一个事务的某个步骤被执行并且其效果被记录到日志中，接着对另一事务的某个步骤做同样的事情，然后对第一个事务的下一步骤或第三个事务的某个步骤，依此类推。事务的交错执行使日志更复杂；仅在事务结束后记载事务的全过程是不够的。

如果系统崩溃，日志将被查阅，以重建崩溃发生时事务正在做的事情。日志还可以和备份一起用于某个未存储日志的磁盘发生介质故障时。通常，为了修复崩溃造成的影响，某些事务的工作将会重做，它们写到数据库中的新值要重写一次。而另一些事务的工作将会撤销，数据库被恢复，就仿佛这些事务未曾执行过。

我们的第一种日志类型称为undo日志，这类日志仅仅进行第二类修复。如果不能完全确定事务的影响已经完成并且已存储到磁盘上，那么事务对数据库所做的所有更新都将被撤销，数据库状态被恢复到事务发生以前的状态。

本节我们将介绍日志记录的基本思想，包括提交（事务的成功完成）动作及其对数据库状

态和日志的影响。我们还将考虑日志自身如何在内存中创建并被“刷新日志”操作拷贝到磁盘上。最后，我们要具体地看一看undo日志，懂得从崩溃中恢复时如何利用它。为了避免在恢复

为什么事务可能终止?

读者可能会疑惑为什么事务会终止而不是提交。事实上有几个原因。最简单的是在事务自身的代码中有某些错误情况，例如通过“撤销”事务来处理的除零企图。DBMS也可能由于一个或多个原因需要终止事务。例如，事务可能陷入死锁中，这时该事务和其他的一个或多个事务各自占有其他事务等待的某个资源（如，为某个数据库元素写入新值的权利）。我们将在19.3节看到，这种情况下系统必须强迫一个或多个事务终止。

时不得不检查整个日志，我们引入“检查点”这一想法，它使得我们可以抛弃日志中旧的部分。undo日志的检查点机制在本节中将会明确谈到。

17.2.1 日志记录

不妨将日志看做一个按只允许附加的方式打开的文件。当事务执行时，日志管理器负责在日志中记录每个重要的事件。每次日志的一个块被填满日志记录，每个日志记录对应于这些事件中的一个。日志块最初在主存中创建，和DBMS所需的其他任何块一样由缓冲区管理器分配。一旦合适，日志块就被写到磁盘的非易失性存储中；关于这个问题我们在17.2.2节中有进一步的讨论。

本章我们所讨论的各种日志类型用到的日志记录有几种形式，包括：

1. <START T>：这一记录表示事务T已开始。

884

2. <COMMIT T>：事务T已成功完成并且对数据库元素不会再有修改。T对数据库所做的任何更新都应反映到磁盘上。然而，由于我们不能控制缓冲区管理器何时决定将块从主存拷贝到磁盘，当我们看到<COMMIT T>日志记录时，通常不能确定磁盘已经进行了更新。如果我们坚持更新已经在磁盘上，这一要求必须由日志管理器来体现（正如undo日志的情况那样）。

3. <ABORT T>：事务T不能成功完成。如果事务T终止，它所做的更新都不能已经被拷贝到磁盘上，并且日志管理器有责任保证这样的更新永不出现在磁盘上或当此发生时消除对磁盘的影响。我们将在19.1.1节讨论消除中止事务的影响这一问题。

对undo日志而言，更新记录是我们惟一需要的其他日志记录类型，更新记录是一个三元组<T, X, v>。这一记录的含义是：事务T改变了数据库元素X，而X原来的值是v。更新记录所反映的改变通常发生在主存中而不是磁盘上；即日志记录是对WRITE动作做出的反应而不是对OUTPUT

一个更新记录有多大?

如果数据库元素是磁盘块，而更新记录包括数据库元素的旧值（或数据库元素的新值和旧值，正如17.4节undo/redo日志中我们将看到的那样），那么似乎一个日志记录的大小可能超过一个块。这并不构成问题，因为和任何传统文件一样，我们可以将日志看做磁盘块的一个序列，将字节存放到这些块时不必考虑块边界。但是，我们有多种压缩日志的方法。例如在某些情况下，我们可以仅记载更新，如某个元组中被事务改变的属性的名字及其旧值。关于在“逻辑日志”中记载更新这一问题将在19.1.7节讨论。

动作做出的反应（参阅17.1.4节去回想这些操作的区别）。此外还请注意undo日志不记录数据库元素的新值，而只记录旧值。正如我们将看到的那样，如果在使用undo日志的系统中需要进行恢复时，恢复管理器要做的惟一事情是通过恢复旧值消除事务可能在磁盘上造成的影响。

17.2.2 undo日志规则

885 要让undo日志能使我们从系统故障中恢复，事务必须遵循两条规则。这些规则影响到缓冲区管理器能做什么，并且要求一旦事务提交就要执行某些动作。我们在此对这些规则进行概括。

U_1：如果事务T改变了数据库元素X，那么形如<T, X, v>的日志记录必须在X的新值写到磁盘前写到磁盘。

U_2：如果事务提交，则其COMMIT日志记录必须在事务改变的所有数据库元素已写到磁盘后再写到磁盘，但应尽快。

简要概括规则U_1和U_2，与事务相关的内容必须按如下顺序写到磁盘：

a) 指明所改变数据库元素的日志记录。

b) 改变的数据库元素自身。

c) COMMIT日志记录。

但是，(a)和(b)的顺序是对各个数据库元素单独适用，而不是对事务的更新记录集合整个适用。

为了强制将日志记录写到磁盘上，日志记录需要一条刷新日志命令来告诉缓冲区管理器将以前没有拷贝到磁盘的日志记录或从上一次拷贝以来已发生修改的日志记录拷贝到磁盘。在动作的序列中，我们将显式地给出FLUSH LOG。事务管理器还需要以某种方式告诉缓冲
886 区管理器在某个数据库元素上执行OUTPUT动作。在事务步骤的序列中我们将继续给出OUTPUT动作。

例17.2 让我们按照undo日志来重新考虑例17.1中的事务。图17-3对图17-2进行了扩展，以给出必须和事务T的动作一起发生的日志项以及刷新日志动作。注意，我们将“A在主存缓冲区中的拷贝”简记为M-A，将“B在磁盘上的拷贝”简记为D-B，依此类推。

图17-3的第1行，事务T开始。发生的第一件事是<START T>记录被写到日志中。第2行表示T读A。第3行是在局部对t做修改，既不影响存储在磁盘上的数据库，也不影响主存缓冲区中数据库的任何部分。第2行与第3行都不需要日志项，因为它们对数据库没有影响。

第4行将A的新值写到缓冲区。对A的这一修改由日志项<T, A, 8>反映，它表示A被T修改，
887 其改前值为8。注意，新值16在undo日志中并没有提到。

其他日志方式预览

在“redo日志”（17.3节）中，恢复时我们重做所有已开始但未提交的事务。redo日志的规则保证我们不必重做一个事务，如果日志上有它的COMMIT记录。“undo/redo日志”（17.4节）在恢复时撤销所有未提交的事务，并重做已提交的事务。同样，关于日志和缓冲区管理的规则保证这些步骤能成功地修复数据库的任何损坏。

影响日志和缓冲区的后台活动

当我们看到图17-3中那样的一系列动作和日志项，很容易将这些动作看做是孤立发生的。然而，DBMS可能同时处理多个事务。因此，日志中事务T的四条日志记录可能和其他事务的记录相互交错。此外，如果这些事务中的一个刷新日志，那么T的日志记录出现在磁盘上可能比图17-3中刷新日志记录所隐含的要早。反映数据库改变的日志记录比所需的更早出现并没有坏处，并且我们要等到T的OUTPUT动作完成才写<COMMIT T>记录，因而保证修改后的值出现在磁盘上比COMMIT记录要早。

如果数据库元素A和B共用一个块，情况会更棘手。这时，将它们中的一个写到磁盘

也就将另一个写到磁盘。在最坏的情况下，我们可能由于将其中的一个元素过早地写到磁盘而违反规则U_1。为了能让undo日志发挥作用，我们可能有必要在事务上采用一些附加的约束。例如，我们可以像18.3节描述的那样，在数据库元素是磁盘块时使用封锁机制，以防止两个事务同时访问同一块。数据库元素是块的部分时出现的这一问题以及其他问题促使我们建议以块作为数据库元素。

第5行到第7行重复同样的三个步骤，不过是对B而非对A。这时，T已经完成并且必须提交。它希望修改后的A和B转到磁盘上，但为了遵循undo日志的两条规则，一系列固定的事件必须发生。

首先，A和B只有在关于修改的日志记录在先写到磁盘上后才能被拷贝到磁盘。因此，在第8步日志被刷新，以保证这些记录出现在磁盘上。接着，第9步和第10步将A和B拷贝到磁盘。事务管理器为了提交T，需要请求缓冲区管理器执行这些步骤。

步 骤	动 作	t	$M-A$	$M-B$	$D-A$	$D-B$	日 志
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<T, A, 8>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T, B, 8>
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)	OUTPUT(B)	16	16	16	16	16	
11)							<COMMIT T>
12)	FLUSH LOG						

图17-3 动作及其日志项

现在可以提交T了，于是<COMMIT T>记录被写到日志中，这就是第11步。最后在第12步，我们必须再次刷新日志，以保证<COMMIT T>记录出现在磁盘上。注意，如果这一记录没有写到磁盘上，我们就可能遇到这样的情况，即事务已经提交，但在很长一段时间内查看日志我们都看不出它已经提交。这种情况可能在崩溃发生时带来一些奇怪的行为，因为正如我们将在17.2.3节看到的那样，一个在用户看来已经提交并已将其修改写到磁盘上的事务这时将被撤销，因而在效果上等同于被终止。 □ 888

17.2.3 使用undo日志的恢复

现在假设系统故障发生了。有可能给定事务的某些数据库更新已经写到磁盘上，而同一事务的另一些更新尚未到达磁盘。如果这样，事务的执行就不是原子的，数据库状态就可能不一致。使用日志将数据库状态恢复到某个一致的状态是*恢复管理器*的任务。

本节我们只考虑恢复管理器最简单的形式，这样的恢复管理器不管日志有多长，都要查看整个日志，作为检查的结果它还会对数据库做一些改变。在17.2.4节我们考虑一种更合乎情理的方式，其中在日志上定期做“检查点”，以限制恢复管理器必须回溯的历史长度。

恢复管理器的第一个任务是将事务划分为已提交事务和未提交事务。如果有日志记录<COMMIT T>，那么根据undo规则U_2，事务T所做的全部改变在此之前已写到磁盘上。因此当系统故障发生时，T自己不可能使数据库处于不一致的状态。

然而，假设我们在日志上发现了<START T>记录但未发现<COMMIT T>记录，那么有可能在崩溃前T对数据库所做的某些修改已经写到磁盘上，而T的另一些修改甚至在主存缓冲区中都还没有进行，或者在缓冲区中进行了但还未拷贝到磁盘上。在这种情况下，T是一个未完成的事务因而必须被撤销。也就是说，T所做的任何改变都必须重置为其原有的值。幸运的是，规则U_1保证如果T在崩溃前改变了磁盘上的X，那么日志中会有<T, X, v>记录，并且该记录在崩溃发生前已被拷贝到磁盘。因此，恢复时我们必须为数据库元素X写入值v。注意，这一规则避开了X在数据库中值是否为v的问题；我们甚至不必检查。

由于日志中可能有一些未提交的事务，并且甚至可能有一些未提交的事务修改了X，所以我们在恢复值的顺序上必须是有规划的。因此，恢复管理器必须从尾部开始扫描日志（即从最近写的记录到最早写的记录）。在扫描过程中，恢复管理器记住所有有<COMMIT T>或<ABORT T>记录的事务T。同时在其向后扫描过程中，如果它看见记录<T, X, v>，则：

1. 如果T的COMMIT记录已被扫描到，则什么也不做。T已经提交，因而不需要撤销。

2. 否则，T是一个未完成的事务或一个终止的事务。恢复管理器必须将崩溃前已发生变化的数据库中的X的值改为v。

889 在做过这些改动后，恢复管理器必须为之前未终止的每个未完成事务T书写日志记录<ABORT T>，之后刷新日志。现在，数据库可以重新开始正常操作，新事务开始执行。

例17.3 让我们考虑图17-3与例17.2中的动作序列。系统崩溃可能发生的不同时机有几个；让我们来考虑每个有明显差别的时机。

1. 崩溃在第12步后发生。这种情况下，我们知道<COMMIT T>记录在崩溃前到达磁盘。当我们恢复时，没有必要撤销T的结果，所有与T相关的日志记录被恢复管理器忽略。

2. 崩溃发生在第11步和第12步之间。包含COMMIT的日志记录可能已被刷新到磁盘；例如，

> **恢复过程中的崩溃**
>
> 假设当我们从上一次崩溃中恢复时系统又一次崩溃。由于undo日志记录的设计方式，所给的是旧值而不是数据库元素值的改变，因此恢复步骤是幂等的（idempotent），即将它们重复多次与执行一次的效果完全相同。我们已经注意到，如果发现记录<T,X,v>,X的值是否已经为v无关紧要——不管怎样我们都可以为X写入v。类似地，如果我们不得不重复恢复过程，那么第一个未完成的恢复过程是否已恢复旧值是无关紧要的；我们只需再次恢复它们。顺便提一下，同样的推理对本章讨论的其他日志方式也成立。由于恢复操作是幂等的，因而我们再次进行恢复时不必考虑前一次恢复所做的更改。

缓冲区管理器可能需要把包含日志结尾的缓冲区提供给另一事务，或者另外的某个事务已请求刷新日志。如果这样，那么就T而言其恢复和情况1一样。但是，如果COMMIT记录尚未到达磁盘，那么恢复管理器认为T未完成。当它向后扫描日志时，首先遇到记录<T, B, 8>。于是它将B在磁盘上的值存为8。接着它遇到记录<T, A, 8>并将A在磁盘上的值置为8。最后，记录<ABORT T>被写到日志中且日志被刷新。

3. 崩溃发生在第10步和第11步之间。现在，COMMIT记录肯定没有写入，因此T未完成并且会像情况2中那样撤销。

4. 崩溃发生在第8步和第10步之间。和情况3一样，T也被撤销。惟一的区别是现在A和/或B的更新可能尚未到达磁盘。不管怎样，这些数据库元素中的每一个都被存为正确的值8。

5. 崩溃发生在第8步以前。现在，关于T的日志记录是否已到达磁盘上并不确定。然而这无

关紧要，因为根据规则U_1我们知道，如果对A和/或B所做更新到达磁盘，则相应的日志记录也到达磁盘，因而如果T在磁盘上改变了A和/或B，那么相应的日志记录将使恢复管理器撤销这些改变。 □

17.2.4 检查点

正如我们看到的那样，恢复原则上需要检查整个日志。当采用undo类型的日志时，一旦事务的COMMIT日志记录被写到磁盘上，该事务的日志记录在恢复时就不再需要。我们可以设想在COMMIT前删除日志，但有时却不能。原因在于，常常是很多事务同时在执行。如果我们在一个事务提交后将日志截断，关于另外的某个活跃事务T的日志记录就可能丢失因而不能在需要进行恢复时用来撤销T。 890

解决潜在的问题最简单的方法是周期性地对日志做检查点。在一个简单的检查点中，我们将：

1. 停止接受新的事务。
2. 等到所有当前活跃的事务提交或终止，并且在日志中写入了COMMIT或ABORT记录。
3. 将日志刷新到磁盘。
4. 写入日志记录<CKPT>，并再次刷新日志。
5. 重新开始接受事务。

所有在检查点前执行的事务将已经完成，并且根据规则U_2其更新将已经到达磁盘。因此，恢复时这些事务中的任何一个都不需要撤销。在恢复中，我们从日志尾部开始向后扫描，确定未完成的事务，就像17.2.3节中那样。但是，当发现<CKPT>记录时，我们知道已经看到了所有未完成的事务。由于只有在检查点结束后事务才能开始，我们必然已经看到了关于未完成事务的所有日志记录。因此没有必要扫描<CKPT>以前的部分，并且事实上将该点以前的日志删除或覆盖是安全的。 891

例17.4 假设日志开始是：

<START T_1>
<T_1, A, 5>
<START T_2>
<T_2, B, 10>

这时，我们决定做一个检查点。由于T_1和T_2是活跃的（未完成的）事务，我们将不得不等到它们完成后才能在日志中写入<CKPT>记录。

日志后续部分一种可能的情况如图17-4所示。假设这时发生崩溃。从尾部开始扫描日志，我们确定T_3是惟一的未完成事务，并且分别将E和F恢复到其改前值25和30。当到达<CKPT>记录时，我们知道没有必要再检查以前的日志记录，并且数据库状态的恢复已经完成。 □

<START T_1>
<$T_1, A, 5$>
<START T_2>
<$T_2, B, 10$>
<$T_2, C, 15$>
<$T_1, D, 20$>
<COMMIT T_1>
<COMMIT T_2>
<CKPT>
<START T_3>
<$T_3, E, 25$>
<$T_3, F, 30$>

图17-4 一个undo日志

17.2.5 非静止检查点

上一节所述检查点技术的一个问题是，我们在设置检查点时相当于关闭了系统。由于活跃事务可能需要很长时间来提交或终止，在用户看来系统似乎停止了。因此，一种称为*非静止检查点*的更复杂的技术通常更受欢迎，它在系统处于检查点时允许新事务进入。非静止检查点的步骤包括： 892

1. 写入日志记录<START CKPT(T_1, …, T_k)>并刷新日志。其中T_1, …, T_k是所有活跃事务（即尚未提交和将其改写到磁盘的事务）的名字或标识符。

2. 等待T_1, …, T_k中的每一个提交或中止，但允许其他事务开始。

3. 当T_1, …, T_k都已完成时，写入日志记录<END CKPT>并刷新日志。

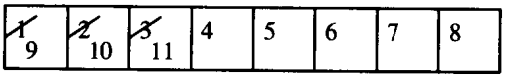
采用这种类型的日志，我们可以按照如下所述从系统故障中恢复。和通常一样，我们从尾部开始扫描日志，在进行过程中找到所有未完成的事务，并将这些事务所改变的数据库元素恢复为其旧值。根据在向后扫描时我们先遇到<END CKPT>记录还是<START CKPT(T_1, …, T_k)>记录，有两种情况。

- 如果先遇到<END CKPT>记录，那么我们知道所有未完成事务在前一<START CKPT (T_1, …, T_k)>记录后开始。因此我们可以向后扫描，直到下一个START CKPT记录，然后就停止；以前的日志没有用处，因而也是可以抛弃的。

最后一条日志记录的发现

日志本质上是一个文件，它的块中存放日志记录。块中从未填充过的空间可以被标注为“空”。如果记录永远不被覆盖，那么通过搜索第一个空记录，并将该记录的前一记录作为文件结尾，恢复管理器可以找到最后一条日志记录。

但是，如果覆盖旧的日志记录，那么我们需要为每条记录保存一个序列号，序列号递增，如下所示：



我们可以找到序列号比下一记录的大的那条记录；二者中靠后的记录将是日志当前的结尾，而通过将当前记录按照它们现在的序列号排序，整个日志就可以被找到。

实践中，一个大的日志可能由多个文件构成，其中有一个“顶层”文件，该文件的记录表明构成日志的文件。在恢复时，我们找到顶层文件的最后一条记录，到达它所表明的文件，并在那里找到最后一条记录。

893

- 如果先遇到<START CKPT(T_1, …, T_k)>记录，那么崩溃发生在检查点过程中。但是，未完成的事务只有在向后扫描过程中到达START CKPT前遇到的那些以及T_1, …, T_k中在发生崩溃前还没有完成的那些。因此，我们扫描到这些未完成事务中最早的那个事务的开始就不必再继续向后扫描。前一个START CKPT记录当然比这些事务的开始都早，但通常我们发现这些未完成事务的开始比到达上一检查点要早得多[⊖]。此外，如果用指针将属于同一事务的日志记录链在一起，那么我们不需要搜索整个日志来找到属于活跃事务的记录；只需要沿着它们的链接向后查看日志。

一个通常的规律是，一旦<END CKPT>记录写到了磁盘，我们就可以将上一个START CKPT记录前的日志删除。

例17.5 假设和例17.4中一样，日志开始是：

<START T_1>

<T_1, A, 5>

⊖ 但请注意，由于检查点是非静止的，因此未完成事务中可能有事务在上一检查点开始和结束之间开始。

<START T_2>

<T_2, B, 10>

现在，我们决定做一个非静止检查点。由于T_1和T_2在这时是活跃的（未完成的）事务，我们写入日志记录

<START CKPT (T_1, T_2)>

假设在等待T_1和T_2完成时，另一个事务T_3开始了。日志后续部分的一种可能情况如图17-5所示。

假设这时发生了系统崩溃。从尾部开始检查日志，我们发现T_3是未完成的事务，因而必须被撤销。最后一条日志记录告诉我们将数据库元素F恢复为值30。当发现<END CKPT>记录时，我们知道所有未完成事务在前一个START CKPT后开始。进一步向后扫描，我们发现记录<T_3, E, 25>，它告诉我们将E恢复为值25。在该记录与START CKPT之间没有其他已开始但尚未提交的事务，所以对数据库不再做进一步的改变。

现在我们来考虑崩溃发生在检查点过程中的情况。假设崩溃后日志的结尾如图17-6所示。向后扫描，我们先确定T_3然后又确定T_2是未完成的事务，并撤销它们所做的修改。当发现<START CKPT (T_1, T_2)>记录时，我们知道其他可能未完成的事务只有T_1。但是，已经扫描到了<COMMIT T_1>记录，所以我们知道T_1不是未完成的。我们也已经看到了<START T_3>记录。因此，我们只需要继续向后扫描直到T_2的START记录，并在此过程中将数据库元素B恢复为值10。 □ 894

<START T_1>
<T_1, A, 5>
<START T_2>
<T_2, B, 10>
<START CKPT (T_1, T_2)>
<T_2, C, 15>
<START T_3>
<T_1, D, 20>
<COMMIT T_1>
<T_3, E, 25>
<COMMIT T_2>
<END CKPT>
<T_3, F, 30>

图17-5 一个使用非静止检查点的undo日志

<START T_1>
<T_1, A, 5>
<START T_2>
<T_2, B, 10>
<START CKPT (T_1, T_2)>
<T_2, C, 15>
<START T_3>
<T_1, D, 20>
<COMMIT T_1>
<T_3, E, 25>

图17-6 检查点过程中系统崩溃时的undo日志

17.2.6 习题

习题17.2.1 给出习题17.1.1中各个事务（称其为T）的undo日志，假设开始时$A = 5$且$B = 10$。

习题17.2.2 对每个表示事务T的动作的日志记录系列，说明所有合乎undo日志规则的事件系列，其中我们关心的事件是将包含数据库元素的块以及包含更新和提交记录的日志块写到磁盘。可以假定日志记录按下面所示顺序写磁盘；即不可能在前一记录还没有写到磁盘时将一记录写到磁盘。

* a) <START T>; <T, A, 10>; <T, B, 20>; <COMMIT T>; 895

b) <START T>; <T, A, 10>; <T, B, 20>; <T, C, 30> <COMMIT T>;

! **习题17.2.3** 习题17.2.2引入的模式可以扩展到事务为n个数据库元素写入新值的情况。如果遵从undo日志规则，这样的一个事务有多少合法的事件系列？

习题17.2.4 下面是两个事务T和U的一系列日志记录：<START T>; <T, A, 10>; <START U>; <U, B, 20>; <T, C, 30>; <U, D, 40>; <COMMIT U>; <T, E, 50>; <COMMIT T>。描述恢

复管理器的行为，包括对磁盘和日志所做的改变，假设发生故障且出现在磁盘上的最后一条日志记录为：

a) <START *U*>

* b) <COMMIT *U*>

c) <*T*, *E*, 50>

d) <COMMIT *T*>

习题17.2.5 对于习题17.2.4描述的每种情况，*T*和*U*所写的哪些值必然出现在磁盘上？哪些值可能出现在磁盘上？

*! **习题17.2.6** 假设改变习题17.2.4中的事务*U*，使<*U*, *D*, 40>记录变为<*U*, *A*, 40>。如果在事件系列中的某个时刻发生故障，对*A*在磁盘上的值有什么影响？这个例子对undo日志自身在保持事务原子性能力方面说明了什么？

习题17.2.7 考虑如下日志记录系列：<START *S*>; <*S*, *A*, 60>; <COMMIT *S*>; <START *T*>; <*T*, *A*, 10>; <START *U*>; <*U*, *B*, 20>; <*T*, *C*, 30>; <START *V*>; <*U*, *D*, 40>; <*V*, *F*, 70>; <COMMIT *U*>; <*T*, *E*, 50>; <COMMIT *T*>; <*V*, *B*, 80>; <COMMIT *V*>。假设我们在（内存）写入如下的一条日志记录后立即开始一个非静止检查点：

a) <*S*, *A*, 60>

* b) <*T*, *A*, 10>

c) <*U*, *B*, 20>

d) <*U*, *D*, 40>

e) <*T*, *E*, 50>

对其中的每一个，说明：

1) 何时写入<END CKPT>记录；并且

2) 对于每一个可能发生故障的时刻，为了找到所有可能未完成的事务，我们需要在日志中回溯多远。

896

17.3 redo日志

尽管undo日志提供了一种维护日志和从系统故障中恢复的简单而自然的策略，但它并不是惟一可能的方法。undo日志一个潜在的问题是，我们在将事务改变的所有数据写到磁盘前不能提交该事务。有时，如果让数据库修改暂时只存在于主存中，我们可以节省磁盘I/O；只要在崩溃事件发生时有日志可以用来修复，这样做是安全的。

如果我们使用一种称为redo日志的日志机制，立即将数据库元素备份到磁盘的需要就可以被避免。redo日志和undo日志的主要区别是：

1. undo日志在恢复时取消未完成事务的影响并忽略已提交事务，而redo日志忽略未完成的事务并重复已提交事务所做的改变。

2. undo日志要求我们在COMMIT日志记录到达磁盘前将修改后的数据库元素写到磁盘，而redo日志要求COMMIT记录在任何修改后的值到达磁盘前出现在磁盘上。

3. 当遵循undo规则U_1和U_2时，发生改变的数据库元素的旧值才是恢复时我们所需要的；而使用redo日志恢复时，我们需要的是新值。因此，redo日志记录尽管和undo日志记录的形式一样，但含义不同。

17.3.1 redo日志规则

redo日志中日志记录<*T*, *X*, *v*>的含义是“事务*T*为数据库元素*X*写入新值*v*”。在这个记录中

没有指出X的旧值。每当一个事务T修改一个数据库元素X时，形如<T, X, v>的一条记录必须被写入日志中。

另外，数据和日志项到达磁盘的顺序可以用一条“redo规则”描述，这条规则称为*提前写日志规则*。

R_1：在修改磁盘上的任何数据库元素X以前，要保证所有与X的这一修改相关的日志记录，包括更新记录<T, X, v>及<COMMIT T>记录，都必须出现在磁盘上。

由于事务的COMMIT记录只有在事务结束后才能写入日志，因而提交记录必然在所有更新日志记录后，所以我们可以将规则R_1的效果概括为如下断言：当使用redo日志时，与事务相关的材料写到磁盘的顺序为：

1. 指出被修改元素的日志记录。
2. COMMIT日志。
3. 改变的数据库元素自身。

例17.6 让我们考虑与例17.2中相同的事务T。图17-7给出了该事务一个可能的事件系列。

步骤	动作	t	M-A	M-B	D-A	D-B	日志
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<T, A, 16>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T, B, 16>
8)							<COMMIT T>
9)	FLUSH LOG						
10)	OUTPUT(A)	16	16	16	16	8	
11)	OUTPUT(B)	16	16	16	16	16	

图17-7 动作及其在使用redo日志时的日志项

图17-7与图17-3的主要区别如下。首先，我们注意到在图17-7的第4和第7行，反映修改的日志记录具有A和B的新值，而不是旧值。其次，我们看到<COMMIT T>记录出现较早，在第8步。然后日志被刷新，因此所有与事务T的更新相关的日志记录出现在磁盘上。只有等到这时，A和B的新值才能写到磁盘。紧接着在第10和11步给出了立即写入的这些值，而实际中它们的发生可能比这要晚得多。 □

17.3.2 使用redo日志的恢复

redo规则R_1的一个重要推论是，只要日志没有<COMMIT T>记录，我们就知道事务T对数据库所做的更新都没有写到磁盘上。因此，恢复时对未完成事务的处理就可以像它们从未发生过似的。然而，提交的事务存在问题，因为我们不知道它们的哪些数据库改变已经写到磁盘。幸运的是，redo日志正好有我们需要的信息：新值，我们可以将新值写到磁盘而不管它们是否已经在磁盘上。在系统崩溃后使用redo日志恢复，我们需要做以下事情。

1. 确定提交的事务。
2. 从起始处扫描日志。对遇到的每一<T, X, v>记录：

(a) 如果T是未提交的事务，则什么也不做。

(b) 如果T是提交的事务，则为数据库元素X写入值v。

3. 对每个未完成的事务T，在日志中写入一个<ABORT T>记录并刷新日志。

Redo动作的顺序

由于一些已提交事务可能对数据库的同一个元素X写入新值，因此在用redo日志恢复过程中我们需按从最早到最晚的顺序扫描日志。这样X最终的值将是最后被写入的值，这是它应该的值。类似地，在undo恢复中，我们需从最晚到最早地扫描日志，这样X的值将是所有被撤消事务修改前的值。

然而，如果DBMS强制操作的原子性，那么在undo日志中我们不希望出现两个未提交的事务，它们修改了数据库的同一个元素。相反，在重做型日志机制中，我们关注的是那些已提交的事务，因为他们需要被重做。有两个已提交的事务，他们在不同的时候都改变了数据库中的同一个元素。这种情形是非常普遍的。因此重做操作的顺序一般是比较重要的，而当采用了恰当的并发控制机制时，撤消操作的顺序就可能不重要了。

例17.7 让我们考虑图17-7中的日志，看一看在动作序列的不同步骤之后发生故障时恢复如何进行。

1. 如果故障发生在第9步后的任何时候，那么<COMMIT T>记录已被刷新到磁盘。恢复系统认定T是一个提交的事务。当向前扫描日志时，日志记录<T, A, 16>和<T, B, 16>使恢复管理器为A和B写入值16。请注意，如果故障发生在第10和11步之间，那么写A是冗余的，而写B还未发生，因而将B改变为16是恢复数据库的一致状态所必需的。如果故障发生在第11步以后，
899 那么写A和写B都是冗余的，但也是无害的。

2. 如果故障发生在第8和9步之间，那么尽管<COMMIT T>记录写入了日志，但可能还没有到达磁盘（依赖于日志是否因其他某种原因而刷新）。如果该记录已到达磁盘，则恢复如同情况1那样进行；而如果该记录没能到达磁盘，那么恢复和下面的情况3一样。

3. 如果故障发生在第8步以前，那么<COMMIT T>记录肯定没有到达磁盘。因此，T被看做一个未完成的事务。磁盘上的A和B不为T做任何改变，而最后一条<ABORT T>记录被写到日志中。□

17.3.3 redo日志的检查点

我们可以在undo日志和redo日志中插入检查点。但是，redo日志提出了一个新问题：由于提交事务所做的修改拷贝到磁盘的时间可能比事务提交的时间晚得多，因此不能仅仅考虑在我们决定创建检查点时活跃的事务。不管检查点是静止的（不允许事务开始）还是非静止的，在检查点的开始和结束之间我们必须采取的一个关键动作是将已被提交事务修改但还未写到磁盘的所有数据库元素写到磁盘。要做到这样，需要缓冲区管理器跟踪哪些缓冲区是脏的，即它们已经被修改但还没有写到磁盘。还需要知道哪个事务修改了哪个缓冲区。

另一方面，我们不需要等待活跃事务提交或中止就能完成检查点，因为它们无论如何都不允许在那个时候将它们的页写到磁盘。执行redo日志的非静止检查点的步骤如下：

1. 写入日志记录<START CKPT(T_1, ···, T_k)>，其中T_1, ···, T_k是所有活跃（即未提交的）事务，并刷新日志。

2. 将START CKPT记录写入日志时所有提交事务已经写到缓冲区但还没有写到磁盘的数据库元素写到磁盘。

3. 写入日志记录<END CKPT>，并刷新日志。

例17.8 图17-8给出了一个可能的redo日志，其中发生了一个检查点。当我们开始检查点时，只有T_2是活跃的，但T_1所写的A值可能已经到达磁盘。如果没有，那么我们必须在检查点结束前将A拷贝到

<START T_1>
<T_1, A, 5>
<START T_2>
<COMMIT T_1>
<T_2, B, 10>
<START CKPT (T_2)>
<T_2, C, 15>
<START T_3>
<T_3, D, 20>
<END CKPT>
<COMMIT T_2>
<COMMIT T_3>

图17-8 一个redo日志

磁盘。检查点的结尾出现在几个其他的事件发生后：T_2为数据库元素C写入一个值，一个新事务T_3开始并为D写入一个值。在检查点结束后发生的惟一的事情是T_2和T_3提交。 □ 900

17.3.4 使用带检查点的redo日志的恢复

正如undo日志那样，插入表明检查点开始和结束的记录可以帮助我们缩小在需要恢复时检查日志的范围。根据最后一个检查点记录是START还是END，有两种不同的情况，这也和undo日志一样。

- 首先假设在崩溃发生前日志中的最后一条检查点记录是<END CKPT>。现在，我们知道在对应的<START CKPT (T_1, …, T_k)>前提交的事务已经将其修改写到了磁盘，因此我们不必关心如何恢复这些事务的影响。但是，T_i中的任一事务或检查点开始后启动的任一事务即使已经提交，都仍可能未将其所做修改转到磁盘上。因此，我们必须像17.3.2节描述的那样进行恢复，但可以只关心最后一个<START CKPT (T_1, …, T_k)>中提到的事务T_i与该日志记录在日志中出现后开始的事务。在搜索日志时，我们在找到最早的<START T_i>记录后就不必继续向后看。但请注意，这些START记录可能出现在任意多个检查点前。将一个事务的所有日志记录向后链接在一起可以帮助我们找到所需记录，正如undo日志中那样。
- 现在，假设日志中的最后一条检查点记录是<START CKPT(T_1, …, T_k)>。我们不能确定在此检查点开始前提交的事务是否已经将其修改写到磁盘上。因此，我们必须搜索到前 901
一<END CKPT>记录，找到与之匹配的<START CKPT(S_1, …, S_m)>记录[⊖]，并重做这些已经提交的、要么在START CKPT后开始要么在S_i中的事务。

例17.9 重新考虑图17-8中的日志。如果故障在结束时发生，我们向后搜索，找到<END CKPT>记录。于是我们知道将所有在写记录<START CKPT (T_2)>后开始的事务或者出现在该记录的列表中的事务（即T_2）作为重做的候选者就足够了。因此，候选集合是{ T_2, T_3}。我们找到了记录<COMMIT T_2>和<COMMIT T_3>，于是知道它们都必须重做。我们向后搜索日志直到<START T_2>记录，为提交的事务找到更新记录< T_2, B, 10>、< T_2, C, 15>和< T_3, D, 20>。由于我们不知道这些修改是否已到达磁盘，因此分别为B、C和D重新写入值10、15和20。

现在，假设崩溃在记录<COMMIT T_2>和<COMMIT T_3>之间发生。恢复与上述过程类似，只不过T_3不再是提交的事务。因此，其修改< T_3, D, 20>不能被重做，并且在恢复中不对D做任何改变，尽管此日志记录处于被检查的记录范围内。恢复后我们也要在日志中写入一条<ABORT T_3>记录。

最后，假设崩溃正好在<END CKPT>记录前发生。原则上，我们必须向后搜索到倒数第二个START CKPT记录，并得到其活跃事务列表。但是，这种情况下没有前一检查点，因而我们必须一直走到日志的开头。因此，我们确定提交的事务只有T_1，重做其动作< T_1, A, 5>，并在恢复后将记录<ABORT T_2>和<ABORT T_3>写入日志中。 □

由于事务可能在几个检查点是活跃的，在<START CKPT (T, …, T_k)>记录中不仅包括事务的名字并且包括指向事务在日志中开始的地方的指针会比较方便。这样做，我们就知道什么时候删除日志中较早的部分是安全的。当我们写入<END CKPT>记录时，就会知道不再需要向后查看到比所有活跃事务T_i中最早的<START T_i >记录更早的日志记录。因此，早于该START记录的所有记录都可以被删除。

17.3.5 习题

习题17.3.1 给出习题17.1.1中各个事务（称其为T）的redo日志记录，假设开始时A = 5且

⊖ 一个小的技术细节是，由于上次的故障，有的START CKPT记录可能没有匹配的<END CKPT>。这就是我们为什么不能只找前一START CKPT记录、而要先找<END CKPT>然后再找前一START CKPT的原因。

$B = 10$。

习题17.3.2 使用redo日志，重做习题17.2.2。

902 **习题17.3.3** 使用redo日志，重做习题17.2.4。

习题17.3.4 使用redo日志，重做习题17.2.5。

习题17.3.5 使用习题17.2.7的数据，对该习题中（a）到（e）的各项，回答：

1) 何时能写入<ENDCKPT>记录；并且

2) 对每一个可能发生故障的时刻，为了找到所有可能未完成的事务，我们需要在日志中向后看多远。请考虑<END CKPT>记录在崩溃发生以前写入和未写入的两种情况。

17.4 undo/redo日志

我们已经看到了两种不同的日志方式，它们的差别在于当数据库元素被修改时日志中保存旧值还是新值。它们各有其缺陷：

- undo日志要求数据在事务结束后立即写到磁盘，可能增加需要进行的磁盘I/O数。
- 另一方面，redo日志要求我们在事务提交和日志记录刷新以前将所有修改过的块保留在缓冲区中，可能增加事务需要的平均缓冲区数。
- 如果数据库元素不是完整的块或块集，在检查点过程中undo日志和redo日志在如何处理缓冲区方面都存在矛盾。例如，如果一个缓冲区中包含被提交的事务修改过的数据库元素A和同一缓冲区中被尚未将其COMMIT记录写到磁盘的事务修改过的数据库元素B，那么由于A我们需要将缓冲区拷贝到磁盘，但将规则R_1运用到B，又不能这样做。

现在要看一种称为undo/redo日志的日志类型，这一日志类型通过付出在日志中维护更多信息的代价，提供了动作顺序上的更大灵活性。

17.4.1 undo/redo规则

undo/redo日志与其他日志类型有相同种类的日志记录，只有一个例外。当数据库元素修改其值时我们写入的更新日志记录有四个组成部分。记录<T, X, v, w>的含义是，事务T改变了数据

903 库元素X的值；其改前值为v，新值为w。undo/redo日志系统必须遵循的约束可用如下规则概括：

UR_1：在由于某个事务T所做的改变而修改磁盘上的数据库元素X前，更新记录<T, X, v, w>必须已写到磁盘上。

undo/redo日志的规则UR_1因而只实施undo日志和redo日志都有的约束。具体地说，<COMMIT T>日志记录可以在磁盘上任何数据库元素的修改之前或之后。

例17.10 图17-9是我们最后在例17.6中看到的事务T的一个变体，其中与事务相关的动作

步骤	动作	t	M-A	M-B	D-A	D-B	日志
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<$T, A, 8, 16$>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<$T, B, 8, 16$>
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)							<COMMIT T>
11)	OUTPUT(B)	16	16	16	16	16	

图17-9 动作及使用undo/redo日志的日志项的一个可能序列

顺序发生了变化。请注意，更新日志记录中现在同时包括A和B的旧值和新值。在这个序列中，我们在将数据库元素A和B输出到磁盘之间写入日志记录<COMMIT T>。第10步也可以出现在第9步前或第11步后。 □

17.4.2 使用undo/redo日志的恢复

当需要用undo/redo日志恢复时，我们拥有的信息既允许通过恢复事务T所改变的数据库元素的旧值来撤销事务T，也允许通过重复T所做的改变来重做T。undo/redo日志的恢复策略是：

1. 按照从前往后的顺序，重做所有已提交的事务；并且
2. 按照从后往前的顺序，撤销所有未提交的事务。 904

请注意这两件事都做对我们来说是必要的。由于undo/redo日志在COMMIT日志记录与数据库修改本身拷贝到磁盘的相对顺序方面提供的灵活性，我们既可以让一个提交事务的部分或全部修改不在磁盘上，也可以让一个未提交事务的部分或全部修改在磁盘上。

例17.11 考虑图17-9中的动作序列。下面是假设崩溃发生在序列中不同的点时进行恢复的不同方式。

1. 假设崩溃发生在<COMMIT T>记录刷新到磁盘后。这时T被认为是提交的事务。我们为A和B往磁盘上写入值16。由于事件实际的顺序，A已经具有值16，而B可能没有，这决定于崩溃发生在第11步之前还是之后。

2. 如果崩溃在<COMMIT T>记录到达磁盘前发生，则T被作为未完成的事务。A和B原来的值被写到磁盘，两种情况下这个值都是8。如果崩溃发生在第9和10步之间，则A在磁盘上

推迟提交的一个问题

和undo日志一样，使用uvdo/redo日志的系统中可能出现这样的行为：事务在用户看来已经提交（例如，他们在网上预订了一个航班座位然后断开连接），但由于<COMMIT T>记录尚未刷新到磁盘，后来的一次崩溃使该事务被撤销而不是重做。如果这样的可能性是一个问题，我们建议为uvdo/redo日志使用一条附加的规则：

UR_2A<COMMIT T>记录一旦出现在日志中就必须被刷新到磁盘上。

例如，在图17-9中我们将在第10步后加入FLUSH LOG。

的值是16，将其恢复到值8是必要的。在这个例子中，B的值不需要撤销，而如果崩溃发生在第9步前，则A的值也不需要撤销。然而，通常我们不能确定恢复是否必要，因此一般执行撤销操作。 □

17.4.3 undo/redo日志的检查点

undo/redo日志的非静止检查点在某种程度上比其他日志方式简单一些。我们只需要做如下事情： 905

1. 写入日志记录<START CKPT($T_1, \cdots, T_k$)>，其中$T_1, \cdots, T_k$是所有的活跃事务，并刷新日志。
2. 将所有脏缓冲区写到磁盘，脏缓冲区即包含一个或多个修改过的数据库元素的缓冲区。和redo日志不同的是，我们刷新所有缓冲区，而不是仅刷新那些被提交事务写过的缓冲区。
3. 写入日志记录<END CKPT>并刷新日志。

关于第(2)点需要注意的是，由于undo/redo日志在数据何时到达磁盘方面提供的灵活性，我们可以容忍将未完成事务写入的数据写到磁盘。所以，我们能够容忍小于完整块的数据库元素，并因此可以共享缓冲区。我们必须对事务做出的惟一要求是：

• 事务在不确定其不会中止之前不能写入任何值（甚至连写到主存缓冲区也不允许）。

正如我们将在19.1节看到的那样，为了避免事务间不一致的相互影响，这一约束无论如何都几乎是必需的。请注意，在redo日志下，上面的条件并不充分，因为即使写入B的事务一定会

事务在恢复中的奇怪行为

读者可能已经注意到，我们并没有指明在使用undo/redo日志恢复时是先撤销还是先重做。事实上，不管先撤销还是先重做，我们都会面临如下情况：提交并被重做的事务T读取值X，该值是由某个未提交并被撤销的事务U写入的。问题不在于我们先重做，使X具有U以前的值；还是先撤销，使X具有由T写入的值。不管哪种方式，这种情况都没有意义，因为数据库的最终状态不对应于任何原子的事务序列的结果。

在实际中，DBMS必须要做的不仅仅是把改变记入日志中。它必须通过某些机制保证这样的情况不会出现。第18章中讨论隔离像T和U这样的事务的方法，使它们通过数据库元素X产生的相互影响不会发生。在19.1节中，我们明确地讨论防止T读“脏”值X（即尚未提交的值）这样的情况发生的方法。

提交，规则R_1也会要求事务的COMMIT记录在B写到磁盘以前写到磁盘。

906 **例17.12** 图17-10给出了类似于图17-8中redo日志的一个undo/redo日志。我们仅仅改变了更新记录，不仅给了它们新值，还给了旧值。为简单起见，我们假设各种情况下旧值比新值小1。

和例17.8中一样，T_2被确定为检查点开始时惟一的提交事务。由于这一日志是undo/redo日志，有可能T_2的新B值10已写到磁盘，而这在redo日志中是不可能的。但是，这一磁盘写是否已经发生却是无关紧要的。在检查点过程中，如果B的新值还不在磁盘上，则我们肯定会将B刷新到磁盘上，因为我们刷新所有的脏缓冲区。同样，如果由提交的事务T_1写入的A的新值还不在磁盘上，我们将刷新A。

<START T_1>
<$T_1, A, 4, 5$>
<START T_2>
<COMMIT T_1>
<$T_2, B, 9, 10$>
<START CKPT (T_2)>
<$T_2, C, 14, 15$>
<START T_3>
<$T_3, D, 19, 20$>
<END CKPT>
<COMMIT T_2>
<COMMIT T_3>

图17-10 一个undo/redo日志

如果崩溃在这一事件系列的末尾发生，则T_2和T_3被确定为提交的事务。事务T_1在检查点前。由于在日志中发现<END CKPT>记录，我们可以正确地假设T_1已经完成并已将其改变写到磁盘上。因此我们重做T_2和T_3，就像在例17.8中那样，并忽略T_1。但是，当重做像T_2这样的事务时，我们并不需要查看比<START CKPT (T_2)>还早的记录，即使T_2在那时是活跃的，因为我们知道T_2在检查点开始前的改变在检查点过程中已经被刷新到磁盘。

再举一例，假设崩溃正好在<COMMIT T_3>记录写到磁盘前发生。那么我们确定T_2是提交的而T_3是未提交的。通过将磁盘上的C设为15来重做T_2；没有必要将B设为10，因为我们知道这一改变在<END CKPT>前已到达磁盘。但是，和redo日志的情况不一样，我们还要撤销T_3；也就是说，我们将磁盘上的D设为19。如果T_3在检查点开始时是活跃的，我们将不得不查看比
907 START-CKPT记录更早的记录，以确定T_3是否有更多的动作已到达磁盘，因而需要撤销。 □

17.4.4 习题

习题17.4.1 给出习题17.1.1中各个事务（称其为T）的undo/redo日志记录，假设开始时$A = 5$且$B = 10$。

习题17.4.2 对每一个表示事务T动作的日志记录序列，说明所有合乎undo/redo日志规则的事件系列，其中我们关心的事件是将包含数据库元素的块以及包含更新和提交记录的日

志块写到磁盘。可以假定日志记录按下列所示顺序写磁盘；即不可能在前一记录还没有写到磁盘时将一记录写到磁盘。

* a) <START *T*>; <*T*, *A*, 10, 11>; <*T*, *B*, 20, 21>; <COMMIT *T*>;

b) <START *T*>; <*T*, *A*, 10, 21>; <*T*, *B*, 20, 21>; <*T*, *C*, 30, 31>; <COMMIT *T*>;

习题17.4.3 下面是两个事务*T*和*U*的一系列日志记录：<START *T*>; <*T*, *A*, 10, 11>; <START *U*>; <*U*, *B*, 20, 21>; <*T*, *C*, 30, 31>; <*U*, *D*, 40, 41>; <COMMIT *U*>; <*T*, *E*, 50, 51>; <COMMIT *T*>。描述恢复管理器的行为，包括对磁盘和日志所做的改变，假设故障发生，且出现在磁盘上的最后一条日志记录如下：

a) <START *U*>

* b) <COMMIT *U*>

c) <*T*, *E*, 50, 51>

d) <COMMIT *T*>

习题17.4.4 对于习题17.4.3描述的每种情况，*T*和*U*所写的哪些值必然出现在磁盘上？哪些值可能出现在磁盘上？

习题17.4.5 考虑如下日志记录序列：<START *S*>; <*S*, *A*, 60, 61>; <COMMIT *S*>; <START *T*>; <*T*, *A*, 61, 62>; <START *U*>; <*U*, *B*, 20, 21>; <*T*, *C*, 30, 31>; <START *V*>; <*U*, *D*, 40, 41>; <*V*, *F*, 70, 71>; <COMMIT *U*>; <*T*, *E*, 50, 51>; <COMMIT *T*>; <*V*, *B*, 21, 22>; <COMMIT *V*>。假设我们在（主存）写入如下的一条日志记录中后立即开始一个非静止检查点：

a) <*S*, *A*, 60, 61>

* b) <*T*, *A*, 61, 62>

c) <*U*, *B*, 20, 21>

d) <*U*, *D*, 40, 41>

e) <*T*, *E*, 50, 51>

908

对其中的每一个，说明：

1) 何时写入<END CKPT>记录，并且

2) 对每一个可能发生故障的时刻，为了找到所有可能未完成的事务，我们需要在日志中回溯多远。请考虑<END CKPT>记录在崩溃发生以前写入和未写入的两种情况。

17.5 防备介质故障

日志可以帮助我们防备系统故障，系统故障发生时磁盘上不会丢失任何东西，而主存中的临时数据会丢失。但是，正如我们在17.1.1节讨论的那样，更严重的故障包括一个或多个磁盘的丢失。原则上讲，如果以下条件成立，我们可以通过日志重建数据库。

a) 日志所在的磁盘不同于存放数据的磁盘；

b) 日志在检查点以后永远不会被丢弃；并且

c) 日志是redo或undo/redo类型，因而新值被存储在日志中。

但是，正如我们提到的那样，日志的增长通常比数据库快，所以永远保存日志是不现实的。

17.5.1 备份

为了防止介质故障，我们采取一种涉及备份（archive）的解决方法，即维护与数据库本身分离的一个数据库拷贝。如果有可能暂时关闭数据库，我们可以在某种存储介质（如磁带或光盘）上创建一个备份拷贝，并将它们存放在远离数据库的某个安全的地方。备份保存数据库在

该时刻的状态，而当介质故障发生时，数据库就可以恢复到该时刻的状态。

要前进到一个更近的状态，我们可以使用日志，前提是备份拷贝以来的日志得到保存并且日志自身在故障之后仍存在。为了防止日志的丢失，我们可以在日志几乎刚刚创建时就将它的一个拷贝传送到与备份一样的远程节点。那么，如果日志与数据都丢失，我们就可以使用备份
909 和远程存储的日志进行恢复，至少恢复到日志最后被传送到远程节点的那一时刻。

由于当数据库规模很大时建立备份是一个冗长的过程，我们通常尽量避免在每个备份步骤

为什么不是仅仅备份日志?

我们可能会对备份的必要性提出疑问，因为如果我们不想停滞在上次做备份时的数据库状态，那么无论如何我们都要在安全的地方备份日志。虽然不太明显，答案在于大型数据库变化的典型速率。尽管一天中数据库只有很少一部分会发生变化，但一年中的改变会比数据库自身大得多，而每个变化都需要在日志中记录。如果我们从不做备份，那么日志永远也不能被截短，而存储日志的开销很快就会超过存储数据库拷贝的开销。

中都要拷贝整个数据库。因此，区别两个级别的备份：

1. 完全转储，这时需要拷贝整个数据库。

2. 增量转储，这时只需要拷贝上一次完全转储或增量转储后改变的那些数据库元素。

也可以有多个级别的转储，其中完全转储被认为是“0级”转储，而“i级”转储拷贝上次在i级或更低级别转储以来改变过的所有东西。

我们可以用一个完全转储及其后续的增量转储来恢复数据库，其过程与redo或undo/redo日志用来修复因系统故障所带来的损害的方式非常相似。我们将完全转储拷贝回数据库，然后以从前往后的顺序，做后续增量转储所记录的改变。由于增量转储一般只包括上次转储以来改变的少量数据，它们占据的空间比完全转储少，做起来也更快。

17.5.2 非静止转储

有关17.5.1节中简单介绍的备份的问题在于，大多数数据库不能在做备份拷贝所需要的一段时间（可能是几个小时）内关闭。因此，我们需要考虑*非静止转储*，它与非静止检查点类似。请回忆一下，非静止检查点试图在磁盘上建立数据库在检查点开始时（近似）状态的一个拷贝。我们可以依赖于检查点附近一段时间内的很小一部分日志，弥补和该状态之间的任何偏差，而
910 偏差的存在是由于这样一个事实：在检查点过程中，新事务可能开始并写磁盘。

类似地，非静止转储试图建立转储开始时数据库的一个拷贝，而在转储进行的几分钟甚至几小时中数据库活动可能改变磁盘上的许多数据库元素。如果需要从备份中恢复数据库，在转储过程中记录的日志项可以用来整理数据，使数据库到达一个一致的状态。二者的类比如图17-11所示。

非静止转储按某种固定的顺序拷贝数据库元素，有可能正好在这些元素被执行中的事务改变时。其结果是，拷贝到备份中的数据库元素可能是也可能不是转储开始时的值。只要在转储持续过程中的日志得到保留，这样的差异可以通过日志来纠正。

例17.13 举一个非常简单的例子，假设我们的数据库由A、B、C和D四个元素构成，当转储开始时它们分别具有1～4这几个值。在转储过程中，A改变为5，C改变为6，而B改变为7。但是，数据库元素的拷贝是按顺序的，而发生的事件序列如图17-12所示。那么尽管数据库在转储开始时具有值（1，2，3，4），而数据库在转储结束时具有值（5，7，6，4），但是在备份中数据库拷贝具有值（1，2，6，4），这是在转储过程中任何时候都不存在的数据库状态。 □

更详细地讲，建立备份的过程可以划分为以下步骤。我们假设日志方式是redo或undo/redo日志；而undo日志不适合与备份一起使用。

1. 写入日志记录<START DUMP>。

2. 根据采用的日志方式执行一个适当的检查点。

3. 根据需要执行完全转储或增量转储，确定数据的拷贝已经到达安全的远程节点。

4. 确定足够的日志已经拷贝到安全的远程节点，至少保证第2项中的检查点以前及包括该检查点的日志在数据库介质故障后仍能存在。

5. 写入日志记录<END DUMP>。

在转储结束后，抛弃上述第2项所进行的检查点的前一个检查点开始以前的日志是安全的。

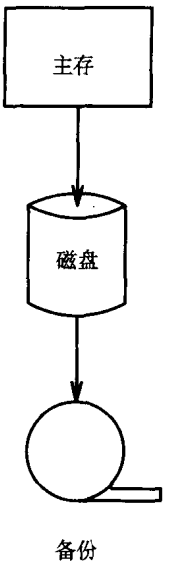


图17-11 检查点与转储的类比

911

例17.14 假设对例17.13中简单的数据库所做的改变由事务T_1（写A和B）和T_2（写C）引起，而它们在转储开始时是活跃的。图17-13给出了转储过程中事件的一个可能的undo/redo日志。

请注意，我们没有表示T_1提交。事务在转储进行的整个过程中保持活跃并不是通常的情况，但这并不影响我们下面要讨论的恢复机制的正确性。 □

912

磁 盘	备 份
A := 5	副本A
C := 6	副本B
B := 7	副本C
	副本D

图17-12 非静止转储中的事件

```
<START DUMP>
<START CKPT (T1,T2)>
<T1,A,1,5>
<T2,C,3,6>
<COMMIT T2>
<T1,B,2,7>
<END CKPT>
Dump completes
<END DUMP>
```

图17-13 转储中记载的日志

17.5.3 使用备份和日志的恢复

假设发生了介质故障，并且我们要通过此前已到达安全的远程节点、在崩溃中未丢失的日志和最近的备份重建数据库。我们执行下列步骤：

1. 根据备份恢复数据库。

(a) 找到最近的完全转储，并根据它来恢复数据库（即将备份拷贝到数据库）。

(b) 如果有后续的增量转储，按照从前往后的顺序，根据各个增量转储来修改数据库。

2. 用保留下来的日志修改数据库。使用适合于所用日志方式的恢复机制。

例17.15 假设在例17.14所示的转储完成后发生了介质故障，而图17-13所示日志得以保存。为了让过程更有意思一些，假设留下来的部分日志中尽管如该图中所示包括<COMMIT T_2>记录，但不包括<COMMIT T_1>记录。数据库首先恢复到备份中的值，即对数据库元素A、B、C和D来说，分别是（1，2，6，4）。

现在，我们必须查看日志。由于T_2已完成，我们重做将C置为6的步骤。在这个例子中，C已经具有值6，但有可能：

a) C的备份在T_2改变C以前产生；或者

b) 备份实际捕获的是C的一个更靠后的值，而该值可能是也可能不是由一个提交记录得以

保留的事务所写入。如果该事务提交，则后面的恢复中C将被恢复为在备份中找到的值。

由于假设T_1没有COMMIT记录，我们必须撤销T_1。使用T_1的日志记录，我们确定A必须被恢复为值1,而B必须被恢复为值2。在备份中它们碰巧具有这些值，但实际的备份值可能由于修改
913 后的A和/或B被包括在备份中而不同。 □

17.5.4 习题

习题17.5.1 如果在例17.14和例17.15中使用的是redo日志而不是undo/redo日志，那么：

a) 日志会是怎样的?

*! b) 如果我们需要使用备份以及这一日志恢复，T_1未提交的结果是什么?

c) 恢复后数据库的状态是什么?

17.6 小结

- *事务管理*：事务管理器的两个主要任务是通过日志保证数据库动作的可恢复性，以及通过调度器保证事务正确的并发行为（本章未讨论）。
- *数据库元素*：数据库划分为元素，通常是磁盘块，但也可以是元组、类外延或者其他的许多单位。数据库元素是日志和调度的单位。
- *日志*：事务的每个重要动作（开始、改变数据库元素、提交或中止）的一条记录被存储在日志中。日志在某个时候必须被备份到磁盘上，这一时刻同对应的数据库改变转移到磁盘上的时刻相关，但依赖于所采用的日志方式。
- *恢复*：当系统崩溃发生时，日志被用来修复数据库，将其恢复到一个一致的状态。
- *日志方式*：日志的三种重要方式是undo、redo和undo/redo，其命名根据恢复时它们可以进行恢复的方式。
- undo日志：每当数据库元素被修改时，这种方式只在日志中记录旧值。使用undo日志，数据库元素的新值必须在关于该改变的日志记录到达磁盘后、并且在做出这一改变的事务的提交记录到达磁盘前写到磁盘。恢复通过为每个未提交事务恢复旧值来完成。
- redo日志：这里，只有数据库元素的新值被记录在日志中。采用这种日志形式，数据库元素的值只有在这一改变的日志记录与其事务的提交记录都已到达磁盘后才能写到磁盘。
914 恢复要做的是为每个提交的事务重新写入新值。
- undo/redo日志：在这种方式中，旧值和新值都被记录在日志中。undo/redo日志比其他方式更灵活，因为它只要求关于改变的日志记录比改变自身先出现在磁盘上。对于提交记录何时出现并没有要求。恢复通过重做提交的事务并撤销未提交的事务来进行。
- *检查点*：由于当需要进行恢复时，原则上所有的方式都需要从头查看整个日志，DBMS必须偶尔对日志做检查点，以保证检查点以前的日志记录在恢复中不再需要。因此，旧的日志记录最终可以被丢弃，而其磁盘空间也可以被重用。
- *非静止检查点*：为了避免做检查点时关闭系统，与各种日志方式相关的技术使检查点可以在系统运作且数据库改变发生时进行。惟一的代价是恢复时非静止检查点前的某些日志记录可能需要检查。
- *备份*：日志防止仅涉及主存丢失的系统故障，而防止磁盘内容丢失的故障有必要使用备份。备份是在安全的地方存储的数据库拷贝。
- *增量备份*：与周期性地将整个数据库拷贝到备份中相反，一个完整备份之后可以跟着几个增量备份，其中只有改变的数据被拷贝到备份中。

- 非静止备份：在数据库运作中建立数据备份的技术是存在的。它们涉及记载备份开始和结束的日志记录，以及在备份时为日志执行一个检查点。
- 从介质故障中恢复：当磁盘丢失时可以通过如下过程恢复，首先用数据库的一个完整备份恢复，然后根据后续的增量备份进行修改，最后通过使用日志的一个备份拷贝恢复到某个一致的数据库状态。

17.7 参考文献

关于事务处理的各个方面，包括日志和恢复，主要的教材是Gray和Reuter写的[5]。这本书的部分材料来自Jim Gray[3]关于事务的一些非正规的、广为传播的笔记；后者以及[4]和[8]是许多日志和恢复技术的主要来源。

[2]是对事务处理技术一个更早、更简洁的描述。[7]是最近有关这一主题的论述。 915

两篇早期的综述，[1]和[6]，都描绘了关于恢复的大量基础性工作，并且将这一主题按照undo、redo和undo/redo三部分来组织，我们在这里也采用了这一组织方式。

1. P. A. Bernstein, N. Goodman, and V. Hadzilacos, "Recovery algorithms for database systems," *Proc. 1983 IFIP Congress*, North Holland, Amsterdam, pp. 799–807.

2. P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading MA, 1987.

3. J. N. Gray, "Notes on database operating systems," in *Operating Systems: an Advanced Course*, pp. 393–481, Springer-Verlag, 1978.

4. J. N. Gray, P. R. McJones, and M. Blasgen, "The recovery manager of the System R database manager," *Computing Surveys* **13**:2 (1981), pp. 223–242.

5. J. N. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan-Kaufmann, San Francisco, 1993.

6. T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery — a taxonomy," *Computing Surveys* **15**:4 (1983), pp. 287–317.

7. V. Kumar and M. Hsu, *Recovery Mechanisms in Database Systems*, Prentice-Hall, Englewood Cliffs NJ, 1998.

8. C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. on Database Systems* **17**:1 (1992), pp. 94–162.

916

第18章 并发控制

事务之间的相互影响可能导致数据库状态的不一致，即使各个事务能保持状态的正确性，而且也没有任何故障发生。因此，不同事务中各个步骤的执行顺序必须以某种方式进行规范。控制这些步骤的功能由DBMS的调度器部件完成，而保证并发执行的事务能保持一致性的整个过程称为*并发控制*。调度器的作用如图18-1所示。

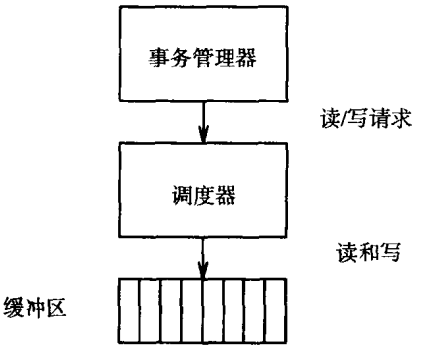


图18-1 调度器接受事务的读/写请求，或者在缓冲区中执行，或者将其推迟

当事务请求对数据库中的元素进行读写时，这些请求被传递给调度器。在大多数情况下，调度器将直接执行读写，如果所需数据库元素不在缓冲区中就首先调用缓冲区管理器。但是在某些情况下，立即执行请求是不安全的。调度器必须推迟请求的执行；有的并发控制技术中，调度器甚至可能终止提交请求的事务。

首先讨论如何保证并发执行的事务能保持数据库状态的正确性。抽象的要求称为*可串行性*，另外还有一个更强的、重要的条件称为*冲突可串行性*，它是大多数调度器所真正实现的。我们考虑实现调度器的最重要技术：封锁、时间戳和有效性确认。

917

对基于封锁的调度器的讨论包括“两阶段封锁”这一重要概念，这是广泛使用的一个保证可串行性的要求。我们还发现调度器能使用许多不同的封锁方式集合，它们各自的应用不同。在我们学习的封锁模式中包括用于嵌套的可封锁元素集合和树结构的可封锁元素集合的模式。

18.1 串行调度和可串行化调度

要展开对并发控制的讨论，必须看一看保证并发执行的一组事务能保持数据库状态一致性的条件。我们最基本的假设是在17.1.3节所说的“正确性原则”：每个事务如果在隔离的情况下执行（即没有其他任何事务与之同时执行），将把任何一致的状态转换到另一个一致的状态。但是，在实践中，事务通常和其他事务并发执行，因而正确性原则并非直接适用。所以，我们需要考虑能保证产生结果与一次执行一个事务所产生结果相同的动作调度。

18.1.1 调度

*调度*是一个或多个事务的重要操作按时间排序的一个序列。当研究并发控制时，重要的读写动作发生在主存缓冲区中，而不是磁盘上。也就是说，某个事务T放入缓冲区中的数据库元素A在该缓冲区中可能不仅被T还被其他访问A的事务读或写。回忆一下17.1.4节，如果数据库元素不在缓冲区中，则READ和WRITE动作先调用INPUT从磁盘上得到该元素；否则READ和WRITE动作直接访问缓冲区中的元素。因此，在考虑并发时只有READ和WRITE及其顺序是重要的，而我们将忽略INPUT和OUTPUT动作。

例18.1 让我们考虑两个事务以及它们的动作按某些顺序执行时的数据库的影响。T_1和T_2的重要动作如图18-2所示。变量t和s分别是T_1和T_2的局部变量；它们不是数据库元素。

918

我们将假设数据库上惟一的一致性约束是$A = B$。由于T_1给A和B都加上100，而T_2将A和B都乘2，我们知道这两个事务隔离运行时各自都能保持一致性。 □

T_1	T_2
READ(A,t)	READ(A,s)
t := t+100	s := s*2
WRITE(A,t)	WRITE(A,s)
READ(B,t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)

图18-2 两个事务

18.1.2 串行调度

如果一个调度的动作组成首先是一个事务的所有动作，然后是另一个事务的所有动作，依此类推，而没有动作的混合，那么我们说这一调度是串行的。更精确地讲，如果有任意两个事务T和T'，若T的某个动作在T'的某个动作前，则T'的所有动作在T'的所有动作前，那么调度S是串行的。

919 **例18.2** 对图18-2中的事务而言，有两个串行调度，一个是T_1在T_2前，而另一个是T_2在T_1前。图18-3给出了T_1在T_2前时的事件序列，初态为$A = B = 25$。遵照惯例，当我们竖直显示时，在页面中靠下的在时间上靠后。此外，所给A和B值指的是它们在主存缓冲区中的值，而不一定是它们在磁盘上的值。

接着，图18-4给出了T_2在T_1前的另一个串行调度，初态仍假设为$A = B = 25$。请注意，两个调度A和B最终的值是不同的；A和B在先做T_1时都是250，而在先做T_2时都是150。然而，只要一致性得以保持，最后的结果并不是核心问题。通常，我们不能期望数据库终态与事务顺序无关。 □

我们可以像图18-3和图18-4中那样，通过按发生顺序列出所有动作来表示串行调度。但是，由于串行调度中动作的顺序只依赖于事务本身的顺序，因此我们有时通过事务列表来表示串行调度。因此，图18-3中的调度表示为（T_1，T_2），而图18-4中调度表示为（T_2，T_1）。

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250

图18-3 T_1在T_2前的串行调度

T_1	T_2	A	B
		25	25
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	50	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		50
READ(A,t)			
t := t+100			
WRITE(A,t)		150	
READ(B,t)			
t := t+100			
WRITE(B,t)			150

图18-4 T_2在T_1前的串行调度

18.1.3 可串行化调度

事务的正确性原则告诉我们，每个串行调度都将保持数据库状态的一致性。但是还有其他能保证可保持一致性的调度吗？有，下面的例子可以说明。通常，如果不管数据库初态怎样，
920 一个调度对数据库状态的影响都和某个串行调度相同，我们就说这个调度是可串行化的。

例18.3 图18-5给出了例18.1中事务的一个调度，此调度是可串行化的，但不是串行的。在这个调度中，T_2在T_1作用于A后而在T_1作用于B前作用于A。但是，我们看到两个事务按这种方式调度，结果却和我们在图18-3中看到的串行调度（T_1，T_2）一样。为了说明这一陈述是正确的，我们必须不仅考虑像图18-5所示那样从数据库状态$A = B = 25$开始产生的结果，还要考虑从

任何一致的状态开始的情况。由于所有一致的数据库状态满足$A = B = c$，不难推断在图18-5的调度中，A和B得到的值都是$2(c+100)$，因此从任意一致的状态开始，一致性都能得到保持。

另一方面，考虑图18-6所示的调度。显然它不是串行的，而更重要的是，它不是可串行化的。我们之所以能确定它不是可串行化的，原因在于它从一致的状态$A = B = 25$开始，最后使数据库处于不一致的状态$A = 250$而$B = 150$。请注意，按照这个动作的顺序，即T_1先作用于A，而T_2先作用于B，我们实际上在A和B上实施了不同的运算，也就是说$A := 2(A+100)$，而$B := 2B+100$。图18-6所示的调度是并发控制机制必须避免的行为类型。 □

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250

图18-5 一个非串行的可串行化调度

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		50
READ(B,t)			
t := t+100			
WRITE(B,t)			150

图18-6 一个非可串行化的调度

18.1.4 事务语义的影响

在到目前为止我们对可串行性的学习中，我们详细地考虑了事务执行的操作，以确定一个调度是否可串行化的。事务细节确实是有关系的，正如我们将在下面的例子中看到的那样。 921

例18.4 考虑图18-7所示的调度，它和图18-6所示的调度惟一不同的地方在于T_2所执行的运算。也就是说，T_2并非将A和B乘2，而是乘1[⊖]。现在，在这一调度的结尾，A和B的值相等，而我们很容易验证，不管初态是什么，终态都将是一致的。事实上，终态就是串行调度（T_1，T_2）或（T_2，T_1）的终态。 □

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*1		
	WRITE(A,s)	125	
	READ(B,s)		
	s := s*1		
	WRITE(B,s)		25
READ(B,t)			
t := t+100			
WRITE(B,t)			125

图18-7 一个仅仅由于事务细节行为而可串行化的调度

⊖ 读者可能会问为什么事务要这样做。这样问是合理的，但为了这个例子，让我们忽略这个问题。事实上，我们可以用许多更合理的事务替代T_2，它们都保持A和B不变；例如，T_2可以仅仅读A和B，并打印其值。或者T_2可能请求用户输入某些数据，然后计算一个因数F来乘A和B，而对某些用户输入我们发现$F = 1$。

遗憾的是，调度器考虑事务所进行的计算的细节是不现实的。由于事务通常不仅包括SQL或其他高级语言的语句书写的代码，还包括通用编程语言编写的代码，有时很难回答诸如“事务是否将A乘上一个不等于1的常数”这样的问题。但是，调度器的确能看到来自事务的读写请求，于是能够知道每个事务读哪些数据库元素，以及它可能改变哪些元素。为了简化调度器的工作，通常假定：

- 事务T所写的任意数据库元素A被赋予一个值，该值以这样一种方式依赖于数据库状态，
922 即不会发生算术上的巧合。

换句话说，如果T对A做某件事情能使数据库状态不一致，则T将会做这件事。在18.2节当讨论保证可串行性的充分条件时，我们会使这一假设更精确一些。

18.1.5 事务和调度的一种记法

如果接受事务所进行的精确计算可以是任意的，那么我们不需要考虑像`t:=t+100`这样的局部计算步骤细节。只有事务执行的读和写需要考虑。因此，我们将用一种简写的记法来表示事务和调度，其中动作有$r_T(X)$和$w_T(X)$，分别表示事务T读和写数据库元素X。此外，由于我们经常将事务称为T_1，T_2，…，我们采用惯用记法$r_i(X)$和$w_i(X)$分别作为$r_{T_i}(X)$和$w_{T_i}(X)$的同义词。

例18.5 图18-2的事务可以写为：

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$;
T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$;

注意，在任何地方都没有提到局部变量t和s，并且关于A和B被读后发生了什么也没有任何说明。直观地说，在怎样修改数据库元素方面，我们将“做最坏的假设”。

923 举另一个例子，考虑图18-5中T_1和T_2的可串行化调度。这一调度写为：

$$r_1(A);\ w_1(A);\ r_2(A);\ w_2(A);\ r_1(B);\ w_1(B);\ r_2(B);\ w_2(B);$$

□

为使这一记法更精确：

1. 动作是形如$r_i(X)$或$w_i(X)$的表达式，分别表示事务T_i读或写数据库元素X。
2. 事务T_i是具有下标i的动作序列。
3. 事务集合T的调度S是一个动作序列，其中对T中的每个事务T_i，T_i中的动作在S中出现的顺序和其在T_i自身定义中出现的顺序一样。我们说S是组成它的事务动作的一个交错。

例如，例18.5的调度中，所有下标为1的动作出现的顺序和它们在T_1的定义中的顺序一样，而所有下标为2的动作出现的顺序和它们在T_2的定义中的顺序一样。

18.1.6 习题

* **习题18.1.1** 航班预订系统执行的一个事务T_1执行以下步骤：

1) 询问顾客希望的航班时间和城市。所需航班信息位于数据库元素（可能是磁盘块）A和B中，系统在磁盘上检索所需信息。
2) 告诉顾客供选择的选项，顾客选择一个航班，该航班的数据在B中，包括该航班的预订号。为该顾客预订该航班。
3) 顾客为该航班选择一个座位；该航班的座位信息位于数据库元素C中。
4) 系统获得顾客的信用卡号，并将该航班的账单附加到数据库元素D的账单列表上。
5) 顾客的电话和航班数据被加到数据库元素E上的另一个列表中，这是为了向顾客发确认航班的传真。

将事务T_i表示为r和w动作的一个序列。

*! **习题18.1.2** 如果两个事务分别有4个动作和6个动作，它们的交错有多少？ 924

18.2 冲突可串行性

我们现在将要提出一个足以保证调度可串行化的条件。当想要保证事务以一种可串行化的方式执行时，商用系统中的调度器通常保证这个我们称为“冲突可串行性”的较强的条件。它基于冲突这一概念：调度中一对连续的动作，它们满足：如果它们的顺序交换，那么涉及的事务中至少有一个的行为会改变。

18.2.1 冲突

首先我们要看一下，大多数的动作对按上面的理解并不冲突。在接下来的内容中，我们假设T_i和T_j是不同的事务，即$i \neq j$。

1. $r_i(X)$;$r_j(Y)$从不会是冲突，即使$X=Y$。原因是这些步骤都不改变任何值。

2. 如果$X \neq Y$，那么$r_i(X)$；$w_j(Y)$不会是冲突。原因是T_j如果在T_i读X以前写Y，X的值不会改变。而且T_i读X对T_j没有影响，因此它不会影响T_j为Y写的值。

3. 如果$X \neq Y$，那么$w_i(X)$；$r_j(Y)$不会是冲突，原因和（2）一样。

4. 类似的还有，只要$X \neq Y$，那么$w_i(X)$；$w_j(Y)$不会是冲突。

另一方面，在下列三种情况下我们不能交换动作的顺序：

a) 同一事务的两个动作冲突，如$r_i(X)$；$w_i(Y)$。原因在于单个事务的动作顺序是固定的，而且是不能被DBMS重新排列的。

b) 不同事务对同一数据库元素的写冲突。也就是说，$w_i(X)$；$w_j(X)$是一个冲突。原因在于，在被写入时，X的值在T_j计算出它是多少后就一直保持。如果我们交换顺序为$w_j(X)$；$w_i(X)$，那么最后使X具有T_i计算出的值。关于“没有巧合”的假设告诉我们，T_i和T_j写入值可能不同，因而对于某个数据库初态而言值将会不同。

c) 不同事务对同一数据库元素的读和写也冲突。也就是说，$r_i(X)$；$w_j(X)$是冲突，$w_i(X)$；$r_j(X)$也是。如果将$w_j(X)$移到$r_i(X)$前，那么T_i读到的X的值将是被T_j写入的值，而我们认为这个值不一定等于X原有的值。因此，交换$r_i(X)$和$w_j(X)$的顺序会影响T_i读到的X的值，而且可能因此影响T_i所做的事。 925

我们得到的结论是，不同事务的任何两个动作在顺序上可以交换，除非

1. 它们涉及同一数据库元素；并且

2. 至少有一个是写。

将这一想法进行扩展，我们可以接受任一调度，进行任意非冲突的交换，目标是将该调度转换为一个串行调度。如果我们能做到这一点，那么初始的调度是可串行化的，因为它对数据库状态的影响在我们做每一个非冲突交换时是不变的。

我们说两个调度是*冲突等价的*，如果通过一系列相邻动作的非冲突交换能将它们中的一个转换为另一个。如果一个调度冲突等价于一个串行调度，那么我们说该调度是*冲突可串行化的*。请注意，冲突可串行性是可串行性的一个充分条件；即冲突可串行化调度是可串行化调度。冲突可串行性对一个可串行化调度来说并不是必要的，但它是商用系统中的调度器在需要保证可串行性时通常使用的条件。

例18.6 考虑例18.5中的调度

$$r_1(A);\ w_1(A);\ r_2(A);\ w_2(A);\ r_1(B);\ w_1(B);\ r_2(B);\ w_2(B);$$

我们说这个调度是冲突可串行化的。图18-8给出了将这一调度转换为串行调度（T_1，T_2）的一系列交换，在此串行调度中，T_1的所有动作在T_2的所有动作之前。我们在每一步中要交换的相邻动作对上加了下划线。 □

$r_1(A); w_1(A); r_2(A); \underline{w_2(A)}; r_1(B); w_1(B); r_2(B); w_2(B);$
$r_1(A); w_1(A); \underline{r_2(A)}; \underline{r_1(B)}; w_2(A); w_1(B); r_2(B); w_2(B);$
$r_1(A); w_1(A); r_1(B); r_2(A); \underline{w_2(A)}; \underline{w_1(B)}; r_2(B); w_2(B);$
$r_1(A); w_1(A); r_1(B); \underline{r_2(A)}; \underline{w_1(B)}; \underline{w_2(A)}; r_2(B); w_2(B);$
$r_1(A); w_1(A); r_1(B); \underline{w_1(B)}; \underline{r_2(A)}; w_2(A); r_2(B); w_2(B);$

图18-8 通过交换相邻动作将冲突可串行化调度转换为串行调度

18.2.2 优先图及冲突可串行性判断

926 检查调度S并决定它是否冲突可串行化相对而言比较简单。其思路是，不管在S的什么地方出现了冲突动作，执行这些动作的事务在任何冲突等价的串行调度中出现的顺序必须和它们在S中出现的顺序一样。因此，冲突动作对给假定的、冲突等价的串行调度中事务的顺序加上了限制。如果这些限制不是相互矛盾的，那么我们就能找到一个冲突等价的串行调度。如果存在相互矛盾，我们就知道不存在这样的串行调度。

已知调度S，其中涉及事务T_1和T_2，可能还有其他事务，我们说T_1优先于T_2，写做$T_1 <_S T_2$，如果有T_1的动作A_1和T_2的动作A_2，满足：

1. 在S中A_1在A_2前；
2. A_1和A_2都涉及同一数据库元素；并且

927 3. A_1和A_2中至少有一个是写动作。

请注意，这正是我们不能交换A_1和A_2顺序的情况。因此，在任何冲突等价于S的调度中，A_1将出现在A_2前。所以，如果这些调度中有一个是串行调度，那么该调度必然使T_1在T_2前。

我们可以在优先图中概括这样的先后次序。优先图的结点是调度S中的事务。当这些事务是具有不同的i的T_i时，我们将仅用整数i来表示T_i的结点。如果$T_i <_S T_j$，则有一条从结点i到结点j的弧。

为什么冲突可串行性对可串行性来说不是必要的

在图18-7中我们已经看到了一例子。在那里我们看到了T_2所进行的具体计算如何使调度可串行化。然而，图18-7的调度不是冲突可串行化的，因为A先被T_1写而B先被T_2写。由于不管是A还是B的写都不能被重新排序，我们没有办法使T_1的所有动作位于T_2的所有动作前，或者反过来。

但是，不依赖于事务所执行的计算的可串行化但非冲突可串行化调度的例子是存在的。例如，考虑事务T_1、T_2和T_3，它们各为X写入一个值。T_1和T_2在为X写入值以前还都为Y写入值。一个可能的、恰好是串行的调度是：

$$S_1: w_1(Y); w_1(X); w_2(Y); w_2(X); w_3(X);$$

S_1最后使X具有T_3写入的值而Y具有T_2写入的值。而调度

$$S_1: w_1(Y); w_2(Y); w_2(X); w_1(X); w_3(X);$$

也如此。直观地说，T_1和T_2写入的X值是无效的，因为T_3覆盖了它们的值。因此S_1和S_2最后得到的X和Y值都相等。由于S_1是串行的，而对任何数据库状态而言，S_1具有和S_2一样的效果，所以我们知道是S_2可串行化的。然而，由于我们不能交换$w_1(Y)$和$w_2(Y)$，并且不能交

换$w_1(X)$和$w_2(X)$，因此我们不能通过交换将S_2转换为串行调度。也就是说，S_2是可串行化的，但不是冲突可串行化的。

例18.7 下面的调度S涉及三个事务T_1、T_2和T_3。

$$S: r_2(A);r_1(B);w_2(A);r_3(A);w_1(B);w_3(A);r_2(B);w_2(B);$$

如果看关于A的动作，我们可以找到$T_2<_S T_3$的多个原因。例如，在S中，$r_2(A)$在$w_3(A)$前，而$w_2(A)$既在$r_3(A)$前又在$w_3(A)$前。这三种情况中的任何一种就足以证明图18-9的优先图中从2到3的弧是正确的。

类似地，如果看关于B的动作，我们可以找到$T_1<_S T_2$的多个原因。例如，动作$r_1(B)$在$w_2(B)$前。因此，S的优先图中也有从1到2的弧。然而，我们能用调度S的动作顺序来证明其合理性的弧也就只有这些。 □

判断调度S是否是冲突可串行化有一条简单的规则：

• 构造S的优先图，并判断其中是否有环。

如果有，那么S不是冲突可串行化的。但如果该图是无环的，那么S是冲突可串行化的，而且结点的任一个拓扑顺序[⊖]都是一个冲突等价的串行顺序。

例18.8 图18-9是无环的，因此例18.7中的调度S是冲突可串行化的。与该图相符的结点顺序或事务顺序只有一个：(T_1，T_2，T_3)。注意，将S转换成这三个事务中每一个的所有动作都按这个顺序发生确实是可能的；这一串行顺序是：

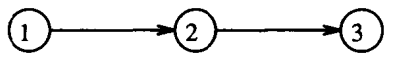


图18-9 例18.7中调度的优先图

$$S': r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B); r_3(A); w_3(A);$$

928

为了说明我们可以通过相邻元素的交换从S得到S'，首先请注意我们可以将$r_1(B)$无冲突地移到$r_2(A)$前。接着，通过三次交换，我们可以将$w_1(B)$移到紧随$r_1(B)$的地方，因为涉及的每个动作都是关于A而不是关于B的。然后我们可以将$r_2(B)$和$w_2(B)$移到紧随$w_2(A)$的位置，移动中涉及的动作都是关于A的；结果是S'。 □

例18.9 考虑调度

$$S_1: r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B);$$

它和S的区别仅仅在于动作$r_2(B)$被向前移动了三个位置。查看关于A的动作，我们仍然只能得到先后次序$T_2<_{S_1} T_3$。但是，当我们检查B时，不仅得到$T_1<_{S_1} T_2$(因为$r_1(B)$和$w_1(B)$出现在$w_2(B)$前)，还得到$T_2<_{S_1} T_1$(因为$r_2(B)$出现在$w_1(B)$前)。因此，我们得到图18-10中调度S_1的优先图。

该图中显然有环。我们断定S_1不是冲突可串行化的。直观地说，任何冲突等价串行调度都必须既使T_1在T_2前又使T_1在T_2后，因而这样的调度是不存在的。 □

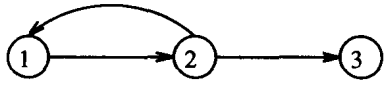


图18-10 一个有环的优先图；其调度不是冲突可串行化的

18.2.3 优先图测试发挥作用的原因

正如我们已经看到的那样，优先图中的环在假想的冲突等价串行调度中事务的顺序上加上了过多的限制。也就是说，如果有一个涉及n个事务的环$T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$，那么在假想的串

⊖ 无环图的拓扑顺序是满足如下条件的任何顺序：对每条弧$a \rightarrow b$，在拓扑顺序中都有结点a在结点b前。通过重复地去除在剩余结点中没有前驱的结点，我们可以为任何无环图找到一个拓扑顺序。

行调度中，T_1的动作必须位于T_2的动作前，而T_2的动作必须位于T_3的动作前，依此类推，一直到T_n。应该位于T_1的动作之后的T_n的动作却又因为存在弧$T_n \rightarrow T_1$而被要求位于T_1的动作之前。因此，我们的结论是，如果在优先图中存在环，则该调度不是冲突可串行化的。

反过来要稍微难一些。我们必须证明只要优先图中无环，就可以通过相邻动作的合法交换来改变调度中动作的顺序，直到调度成为一个串行调度。如果能做到这一点，那么我们就能证明任意一个具有无环优先图的调度都是冲突可串行化的。我们的证明是对调度所涉及的事务数进行归纳。

基础：如果 $n = 1$，即调度中只有一个事务，那么调度已经是串行的，因此也肯定是冲突
929 可串行化的。

归纳：设调度S由n个事务

$$T_1, T_2, \cdots, T_n$$

的动作构成。我们假设S有一个无环的优先图。如果一个有限图是无环的，那么至少有一个结点没有到达该结点的弧；设对应于事务T_i的结点i是这样的一个结点。由于没有弧到达结点i，因此S中不可能有这样的动作A：

1. 涉及T_i以外的某个事务T_j；
2. 位于T_i的某个动作前；并且
3. 与这个动作冲突。

因为如果存在这样的A，我们应该在优先图中加入从结点j到结点i的弧。

因此我们可以交换T_i的所有动作，保持它们的顺序，但将它们移到S的前部。该调度现在具有如下形式

（T_i的动作）（其他$n-1$个事务的动作）

我们现在考虑S的后半部分——T_i以外所有事务的动作。由于这些动作保持了与它们在S中相同的顺序，除了没有结点T_i以及从该结点出发的所有弧以外，后半部分的优先图和S的一样。

由于原始的优先图是无环的，删除结点和弧不可能使其成为有环的，我们断定后半部分的优先图无环。此外，由于后半部分涉及$n-1$个事务，归纳假设对它来说是适用的。因此，我们可以通过相邻动作的合法交换重新排列后半部分中动作的顺序来将其转换为串行调度。现在，S自身已经被转成了一个串行调度，其中首先是T_i的动作，然后是按照某种串行顺序的其他事务的动作。归纳证明完成，我们的结论是每个具有无环优先图的调度都是冲突可串行化的。

18.2.4 习题

习题18.2.1 下面是用对数据库元素A和B的影响来描述的两个事务，我们可以假设数据库元素A和B是整数。

T_1: `READ(A,t); t:=t+2; WRITE(A,t); READ(B,t); t:=t*3; WRITE(B,t);`

T_2: `READ(B,s); s:=s*2; WRITE(B,s); READ(A,s); s:=s+3; WRITE(A,s);`

我们假设，不管数据库上的一致性约束是什么，这些事务在隔离的情况下能够保持这些约
930 束。注意，$A = B$不是一致性约束。

a) 这两个串行顺序对数据库的影响是相同的，即（T_1，T_2）与（T_2，T_1）等价。通过给出任意数据库初态时这两个事务的结果，说明这一事实。

b) 给出上面12个动作的一个串行调度的例子和一个非串行调度的例子。

c) 这12个动作共有多少串行调度？

*!! d) 这12个动作共有多少可串行化调度?

习题18.2.2 用只给出读写动作的记法，习题18.2.1中的两个事务可以写作：

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$;

T_2: $r_2(B)$; $w_2(B)$; $r_2(A)$; $w_2(A)$;

回答以下问题：

*! a) 在上述8个动作可能的调度中，有多少冲突等价于串行顺序（T_1，T_2）?

b) 这8个动作的调度中，有多少等价于串行顺序（T_2，T_1）?

!! c) 这8个动作的调度中，有多少等价于（不一定是冲突等价于）串行调度（T_1，T_2），假设事务具有习题18.2.1中描述的对数据库的影响?

! d) 为什么上述(c)的答案与习题18.2.1(d)的答案不同?

! **习题18.2.3** 假设习题18.2.2中的事务改为：

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$;

T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$;

也就是说，事务保持它们在习题18.2.1中的语义，但T_2改为在处理B以前处理A。给出：

a) 冲突可串行化的调度数。

b) 可串行化的调度数，假设事务对数据库状态的影响同习题18.2.1。

习题18.2.4 对以下的每个调度：

* a) $r_1(A)$; $r_2(A)$; $r_3(B)$; $w_1(A)$; $r_2(C)$; $r_2(B)$; $w_2(B)$; $w_1(C)$; 931

b) $r_1(A)$; $w_1(B)$; $r_2(B)$: $w_2(C)$; $r_3(C)$; $w_3(A)$;

c) $w_3(A)$; $r_1(A)$; $w_1(B)$; $r_2(B)$: $w_2(C)$; $r_3(C)$;

d) $r_1(A)$; $r_2(A)$; $w_1(B)$; $w_2(B)$; $r_1(B)$; $r_2(B)$; $w_2(C)$; $w_1(D)$;

e) $r_1(A)$; $r_2(A)$; $r_1(B)$; $r_2(B)$; $r_3(A)$; $r_4(B)$; $w_1(A)$; $w_2(B)$;

回答如下问题：

1) 调度的优先图是什么?

2) 调度是冲突可串行化的吗? 如果是，等价的串行调度有哪些?

3) 是否有等价的调度（不管事务对数据做什么），但又不是冲突等价的?

!! **习题18.2.5** 如果调度S中事务T的每个动作都在事务U的所有动作之前，我们说事务T位于事务U前。注意，如果T和U是S中仅有的事务，那么说T位于U前等同于说S是串行调度（T，U）。但是，如果S还涉及T和U以外的事务，那么S可能不是可串行化的，而事实上，由于其他事务的影响，甚至可能不是冲突可串行化的。给出一个调度S的例子，满足：

1) 在S中，T_1位于T_2前；并且

2) S是冲突可串行化的；但是

3) 在每个冲突等价于S的串行调度中，T_2位于T_1前。

! **习题18.2.6** 解释对任意n>1怎样找到一个调度，其优先图中具有长度为n的环，但没有更小的环。

18.3 使用锁的可串行性实现

设想以一种不受约束的方式执行其动作的事务的一个集合。这些动作将形成一个调度，但

是这一调度不太可能是可串行化的。防止导致非可串行化调度的动作顺序是调度器的任务。在本节中，我们考虑调度器最常用的体系结构，这种结构在数据库元素上维护“锁”，以防止非可串行化的行为。直观地说，事务获得在它所访问的数据库元素上的锁，以防止其他事务几乎在同一时间访问这些元素并因而引入非可串行性的可能。

在本节中，我们用一个（过于）简单的封锁模式来介绍封锁的概念。这种模式中只有一种
932 锁，它是事务想要在数据库元素上执行任何操作时都必须在该数据库元素上获得的。在18.4节中，我们将学习更现实的封锁模式，这样的封锁模式使用多种锁，包括常用的分别对应于读权限和写权限的共享/排他锁。

18.3.1 锁

在图18-11中我们看到一个使用锁表来协助自己工作的调度器。回忆一下，调度器的责任是接受来自事务的请求，或者允许它们在数据库上操作，或者将它们推迟到允许它们执行是安全的。锁表用来指导这一决策，我们将详细讨论其方式。

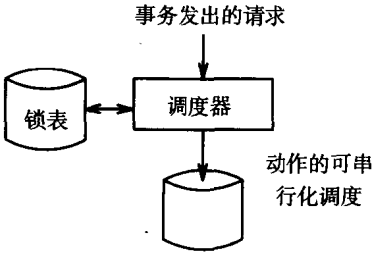


图18-11 使用锁表指导决策的调度器

理想地讲，调度器转发请求当且仅当该请求的执行不可能在所有活跃事务提交或终止后使数据库处于不一致的状态。但是在现实中这个问题太难而不能决定。因此，所有调度器都使用一种简单的测试，它能保证可串行性，但可能会禁止一些自身并不会导致不一致性的动作。封锁调度器像大多数调度器种类一样，事实上实现的是冲突可串行性，而我们已经知道这是一个比可串行性更苛刻的条件。

当调度器使用锁时，事务在读写数据库元素以外还必须申请和释放锁。锁的正确使用有两种意义：一种适用于事务的结构，而另一种适用于调度的结构。

- *事务的一致性*：动作和锁必须按预期的方式发生联系：

1. 事务只有以前已经在数据库元素上申请了锁并且还没有释放锁时才能读或写该数据库元素。

2. 如果事务封锁某个数据库元素，它以后必须为该元素解锁。

- *调度的合法性*：锁必须具有其预期的含义：任何两个事务都不能封锁同一元素，除非其
933 中一个事务已经先释放其锁。

扩展我们关于动作的记法，以加入封锁和解锁动作：

$l_i(X)$：事务T_i请求数据库元素X上的锁。

$u_i(X)$：事务T_i释放它在数据库元素X上的锁（解锁）。

因此，事务的一致性条件可以表述为：“只要事务T_i有动作$r_i(X)$或$w_i(X)$，那么前面必然有一个动作$l_i(X)$且二者之间没有$u_i(X)$，并且后面将会有一个$u_i(X)$”。调度的合法性表述为：“如果调度中在动作$l_i(X)$后有$l_j(X)$，那么这些动作之间的某个地方必然有一个动作$u_i(X)$”。

例18.10 让我们考虑在例18.1中介绍的两个事务T_1和T_2。回忆一下，T_1给数据库元素A和B加上100，而T_2将它们加倍。下面是这些事务的说明，我们在其中包含了锁的动作，还包含了算术动作，以帮助我们记起这些事务是做什么的[⊖]。

⊖ 请记住事务的实际计算在我们现在的记法中通常是不被表示出来的，因为调度器在决定是同意事务请求还是拒绝时并不考虑它们。

T_1: $l_1(A)$; $r_1(A)$; A := A+100; $w_1(A)$; $u_1(A)$; $l_1(B)$; $r_1(B)$; B := B+100; $w_1(B)$; $u_1(B)$;

T_2: $l_2(A)$; $r_2(A)$; A := A*2; $w_2(A)$; $u_2(A)$; $l_2(B)$; $r_2(B)$; B := B*2; $w_2(B)$; $u_2(B)$;

这些事务中每一个都是一致的。它们都释放它们在A和B上持有的锁。此外，它们对A和B的操作都只是在前面已经获得了该元素上的锁且尚未释放该锁的那些步骤中。

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$;			
A := A+100;			
$w_1(A)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$;		
	A := A*2;		
	$w_2(A)$; $u_2(A)$;	250	
	$l_2(B)$; $r_2(B)$;		
	B := B*2;		
	$w_2(B)$; $u_2(B)$;		50
$l_1(B)$; $r_1(B)$;			
B := B+100;			
$w_1(B)$; $u_1(B)$;			150

图18-12 一致事务的一个合法调度；但不幸的是，它不是可串行化的

图18-12给出了这两个事务的一个合法调度。为了节省空间，我们在一行上放了多个动作。这个调度是合法的，因为这两个事务从未同时在A上持有锁，并且对B也一样。具体地说，T_2一直等到T_1执行$u_1(A)$后才执行$l_2(A)$，而T_1一直等到T_2执行$u_2(B)$后才执行$l_1(B)$。正如我们在所计算值的序列中看到的那样，这个调度尽管是合法的，却不是可串行化的。我们将在18.3.3节中来看保证合法调度冲突可串行所需要附加的条件“两阶段封锁”。 □

18.3.2 封锁调度器

基于封锁的调度器的任务是当且仅当请求将产生合法调度时同意请求。为了帮助进行决策，调度器有一个锁表，对于每个数据库元素，如果其上有锁，那么锁表指明当前持有该锁的事务。 934
我们将在18.5.2节更详细地讨论锁表的结构。但是，当像我们到目前为止所假设的那样只有一种锁时，该表可以被看做是关系Locks(element, transaction)，由满足事务T当前具有数据库元素X上的锁的（X，T）对组成。调度器只需要用简单的INSERT和DELETE语句访问和修改这一关系。

例18.11 图18-12中的调度是合法的，正如我们提到的那样，所以封锁调度器将按照所示的请求到达的顺序同意每个请求。但是，有时不能同意请求。下面是来自例18.10的T_1和T_2，我们只做了简单的（但也是重要的，正如我们将在18.3.3节看到的那样）修改，其中T_1和T_2都在释放A上的锁以前封锁B。

T_1: $l_1(A)$; $r_1(A)$; A := A+100; $w_1(A)$; $l_1(B)$; $u_1(A)$; $r_1(B)$; B := B+100; $w_1(B)$; $u_1(B)$;

T_2: $l_2(A)$; $r_2(A)$; A := A*2; $w_2(A)$; $l_2(B)$; $u_2(A)$; $r_2(B)$; B := B*2; $w_2(B)$; $u_2(B)$;

在图18-13中，当T_2请求B上的锁时，调度器必须拒绝此锁，因为T_1仍持有B上的锁。因此，T_2停转，而接下来的动作是来自T_1的。最后，T_1执行$u_1(B)$，这将解锁B。现在，T_2可以获得它

在B上的锁，这是在下一步所执行的。请注意，由于T_2被迫等待，它推迟到T_1给B加100后再将
935 其乘2，因而得到一个一致的数据库状态。 □

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$;			
`A := A+100;`			
$w_1(A)$; $l_1(B)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$;		
	`A := A*2;`		
	$w_2(A)$;	250	
	$l_2(B)$ 被拒绝		
$r_1(B)$; `B := B+100;`			
$w_1(B)$; $u_1(B)$;			125
	$l_2(B)$; $u_2(A)$; $r_2(B)$;		
	`B := B*2;`		
	$w_2(B)$; $u_2(B)$;		250

图18-13 封锁调度器推迟将导致非法调度的请求

18.3.3 两阶段封锁

在一个令人吃惊的条件下，我们可以保证一致事务的合法调度是冲突可串行化的。这一条件称为两阶段封锁或2PL，在商用封锁系统中被广泛采用。2PL条件是：

• 在每个事务中，所有封锁请求先于所有解锁请求。

因此2PL中所指的“两阶段”是获得锁的第一阶段和放弃锁的第二阶段。两阶段封锁像一致性一样，是对一个事务中动作的顺序进行限制的条件。服从2PL条件的事务被称为两阶段封锁事务，或2PL事务。

例18.12 在例18.10中，事务不遵循两阶段封锁规则。例如，T_1在封锁B以前解锁A。但是，在例18.11中看到的事务版本确实遵从2PL条件。注意，T_1在前五个动作中封锁A和B，并且在接下来的五个动作中解锁；T_2的行为类似。如果比较图18-12和图18-13，我们可以看到两阶段封锁事务如何同调度器进行正确的交互以保证一致性，而非2PL事务允许不一致的（因而非冲突
936 可串行化）行为。 □

18.3.4 两阶段封锁发挥作用的原因

在例子中我们所看到的由2PL带来的好处通常都成立，这一点是正确的，却绝不是显而易见的。直观地说，每个两阶段封锁事务可以被认为是在其提出第一个解锁请求的瞬间完整执行，如图18-14所示。与2PL事务的调度S冲突等价的串行调度是事务顺序与其第一个解锁顺序相同的串行调度[⊖]。

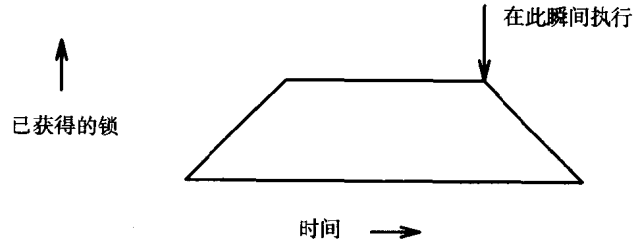


图18-14 每个两阶段封锁事务有一个可以认为它瞬间执行的时刻

⊖ 某些调度还存在其他的冲突等价串行调度。

我们将说明如何把由一致的两阶段封锁事务构成的任意合法调度S转换为冲突等价的串行调度。这一转换最好用S中事务数n上的归纳来描述。在下面的内容中，牢记冲突等价问题只针对读写动作是非常重要的。一旦将读和写串行排序，我们就可以根据不同事务的需要在它们周围加上封锁和解锁动作。由于每个事务在其结束前释放所有锁，因此我们知道该串行调度是合法的。

基础：如果$n = 1$，则没有什么需要做的；S已经是一个串行调度。

归纳：假设S涉及n个事务$T_1,T_2,\cdots,T_n$，并设T_i是在整个S中有第一个解锁动作（例如$u_i(X)$）的事务。我们断言，将T_i的所有动作不经过任何冲突动作而向前移动到调度的开始是可能的。

考虑T_i的某个动作，例如$w_i(Y)$。S中这一动作前可能有冲突的动作例如$w_j(Y)$吗？如果有，那么在调度S中，$u_j(Y)$和$l_i(Y)$必然交错出现在这样一个动作序列中

$$\cdots;\ w_j(Y);\ \cdots;\ u_j(Y);\ \cdots;\ l_i(Y);\ \cdots;\ w_i(Y);\ \cdots$$

既然T_i是第一个解锁的，S中$u_i(X)$必然在$u_j(Y)$前；也就是说，S可能形如：

$$\cdots;\ w_j(Y);\ \cdots;\ u_i(X);\ \cdots;\ u_j(Y);\ \cdots;\ l_i(Y);\ \cdots;\ w_i(Y);\ \cdots$$ 937

或$u_i(X)$甚至可能出现在$w_j(Y)$前。不管哪种情况，$u_i(X)$出现在$l_i(Y)$前，这意味着T_i不像我们假定的那样是两阶段封锁的。尽管我们只证明了写的冲突对不存在，同样的证明也适用于任意一对由来自T_i的一个动作和来自T_j的一个动作构成的可能冲突的动作。

我们的结论是，确实能够通过先使用非冲突的读写动作的交换，然后恢复T_i的封锁和解锁动作，将T_i的所有动作向前移动到S的开始。也就是说，S能被写作如下形式

（T_i的动作）（其他$n-1$个事务的动作）

由$n-1$个事务构成的后半部分仍然是一致的2PL事务的一个合法调度，因此归纳假设在其上适用。我们将后半部分转换为冲突等价串行调度，而我们现在已经证明整个S是冲突可串行化的。

18.3.5 习题

习题18.3.1 下面是两个事务，其中给出了封锁请求和事务的语义。回忆一下习题18.2.1中，这些事务具有特殊的性质，即它们被调度的方式可以是非冲突可串行化的，但由于其语义又是可串行化的。

T_1: $l_1(A)$; $r_1(A)$; `A := A+2`; $w_1(A)$; $u_1(A)$; $l_1(B)$; $r_1(B)$; `B := B*3`; $w_1(B)$; $u_1(B)$;

T_2: $l_2(B)$; $r_2(B)$; `B := B*2`; $w_2(B)$; $u_2(B)$; $l_2(A)$; $r_2(A)$; `A := A+3`; $w_2(A)$; $u_2(A)$;

在下面的问题中，只考虑读写动作的调度，而不要考虑封锁、解锁或赋值步骤。

* a) 给出被锁禁止的调度的一个例子。

! b) 在8个读写动作的$\binom{8}{4} = 70$种顺序中，有多少是合法的调度（即它们被锁所允许）？

! c) 在合法调度中，有多少是可串行化的（根据所给事务语义）？

! d) 在那些合法的可串行化调度中，有多少是冲突可串行化的？

!! e) 由于T_1和T_2不是两阶段封锁的，我们能够预见某些非可串行化行为可能发生。是否有非可串行化的合法调度？如果有，给出一个例子；如果没有，解释原因。 938

*! **习题18.3.2** 下面是习题18.3.1中的事务，但所有解锁都移到了末尾，从而使它们成为两阶段封锁的事务。

T_1: $l_1(A)$; $r_1(A)$; `A := A+2`; $w_1(A)$; $l_1(B)$; $r_1(B)$; `B := B*3`; $w_1(B)$; $u_1(A)$; $u_1(B)$;

T_2: $l_2(B)$; $r_2(B)$; `B := B*2`; $w_2(B)$; $l_2(A)$; $r_2(A)$; `A := A+3`; $w_2(A)$; $u_2(B)$; $u_2(A)$;

这些事务所有的读写动作的合法调度有多少?

939 **习题18.3.3** 对习题18.2.4中的每个调度，假设每个事务刚好在读或写每个数据库元素以前获得该元素上的锁，并且每个事务在最后一次访问一个元素后立即释放其锁。说明封锁调度器对这些调度中的每一个会怎么做；即哪些请求将被推迟，而什么时候它们又将被允许继续?

! **习题18.3.4** 对下面描述的每个事务，假设我们为每个被访问的数据库元素插入一个封锁动作和一个解锁动作。

* a) $r_1(A)$; $w_1(B)$

b) $r_2(A)$; $w_2(A)$; $w_2(B)$

说明如下几种情况下封锁、解锁、读和写动作各有多少顺序：

1) 一致的并且两阶段封锁的。

2) 一致的但不是两阶段封锁的。

3) 不一致的但是两阶段封锁的。

4) 既不是一致的也不是两阶段封锁的。

死锁的风险

两阶段封锁一个未解决的问题是死锁的可能性，即调度器迫使几个事务永远地等待另一个事务持有的锁。例如，考虑例18.11中的2PL事务，但将T_2改为先对B操作：

T_1: $l_1(A)$; $r_1(A)$; `A := A+100`; $w_1(A)$; $l_1(B)$; $u_1(A)$; $r_1(B)$; `B := B+100`; $w_1(B)$; $u_1(B)$;

T_2: $l_2(B)$; $r_2(B)$; `B := B*2`; $w_2(B)$; $l_2(A)$; $u_2(B)$; $r_2(A)$; `A := A*2`; $w_2(A)$; $u_2(A)$;

这些事务动作的一个可能的间隔为：

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$;			
	$l_2(B)$; $r_2(B)$;		
`A := A+100`;			
	`B := B*2`;		
$w_1(A)$;		125	
	$w_2(B)$;		50
$l_1(B)$ 被拒绝	$l_2(A)$ 被拒绝		

现在，两个事务都不能继续进行，而它们将永远等待。在19.3节，我们将讨论这种情况的补救方法。但是，请注意让两个事务都继续进行是不可能的，因为如果我们这样做，数据库的最终状态就不可能满足$A=B$。

18.4 用多种锁方式的封锁系统

18.3节的封锁模式阐明了在封锁背后的重要思路，但它过于简单，因而不是一个实用的模

式。主要的问题在于，事务T即使只想读数据库元素X而不写它，也必须获得X上的锁。我们不能避开锁的获得，因为如果不获得，当T活跃时另一个事务可能为X写入一个新值而导致非可串行化行为。另一方面，只要事务都不允许写X，那么不允许几个事务同时读X就是毫无理由的。

这促使我们介绍第一个也是最常用的封锁模式，这种模式中有两种不同的锁：一种用于读（称做“共享锁”或“读锁”），另一种用于写（称做“排他锁”或“写锁”）。接着我们要看一种改进的模式，其中事务允许获得共享锁，并在以后将其“升级”为排他锁。我们还将考虑“增量锁”，它专门处理对数据库元素进行增量的写操作；重要的区别在于增量操作可交换，而通常的写操作并非如此。这些例子把我们引到封锁模式的一种使用“相容性矩阵”的通用记法，“相容性矩阵”表明当数据库元素上存在其他锁时该数据库元素上可以被授予什么样的锁。

18.4.1 共享锁与排他锁

940

由于同一数据库元素上的两个读操作并不产生冲突，因而没有必要采用封锁或其他并发控制机制来强制读操作以某种特定的顺序发生。正如前面提到的，我们仍然需要封锁我们将要读取的元素，因为写该元素的事务是必须被禁止的。但是，在写时我们需要的锁比在读时需要的锁要“强”，因为它既禁止读又禁止写。

因此让我们考虑使用两种不同类型的锁的封锁调度器：*共享锁*和*排他锁*。直观地说，对任何数据库元素X，其上或者可以有一个排他锁，或者没有排他锁而有任意数目的共享锁。如果我们想要写X，则需要有X上的一个排他锁。可以推测，如果想要读X而不写它，那么我们倾向于只获得共享锁。

我们将使用$sl_i(X)$来表示“事务T_i申请数据库元素X上的一个共享锁”，而用$xl_i(X)$来表示“事务T_i申请数据库元素X上的一个排他锁”。我们继续用$u_i(X)$表示T_i解锁X；即它释放自己在X上持有的不管什么样的锁。

事务的一致性、事务的2PL和调度的合法性这三种要求中的每一种在共享/排他封锁系统中都有各自的对应项。这里我们将这些要求概括为：

1. *事务的一致性*：如果不是持有排他锁就不能写，并且如果不是持有某个锁就不能读。更精确地说，在任何事务T_i中，

   (a) 读动作$r_i(X)$之前必须有$sl_i(X)$或$xl_i(X)$，而且中间没有$u_i(X)$。

   (b) 写动作$w_i(X)$之前必须有$xl_i(X)$，而且中间没有$u_i(X)$。

   所有锁都必须跟一个相同元素的解锁。

2. *事务的两阶段封锁*：封锁必须在解锁之前。更精确地说，在任意一个两阶段封锁事务T_i中，任何$sl_i(X)$或$xl_i(X)$动作前不能有$u_i(X)$动作。

3. *调度的合法性*：一个元素或者可以被一个事务排他地封锁，或者可以被几个事务共享地封锁，但不能二者兼而有之。更精确地讲，

   (a) 如果$xl_i(X)$出现在调度中，那么对某个$j \neq i$，后面不能再有$xl_j(X)$或$sl_j(X)$，除非中间插入了$u_i(X)$。

   (b) 如果$sl_i(X)$出现在调度中，那么后面不能再有$xl_j(X)$，除非中间插入了$u_i(X)$。

941

请注意，我们允许一个事务在同一个元素上既申请并持有共享锁又申请并持有排他锁，只要它这样做不与其他事务的锁发生冲突。如果事务能预先知道自己对锁的需求，那么肯定只会请求排他锁。但如果锁的需求是不可预测的，那么可能事务在不同的时候申请共享锁和排他锁。

例18.13 让我们看一下使用共享锁和排他锁时，以下两个事务的一个可能的调度：

T_1: $sl_1(A)$; $r_1(A)$; $xl_1(B)$; $r_1(B)$; $w_1(B)$; $u_1(A)$; $u_1(B)$;

T_2: $sl_2(A)$; $r_2(A)$; $sl_2(B)$; $r_2(B)$; $u_2(A)$; $u_2(B)$;

T_1和T_2都读A和B，但只有T_1写B。二者都不写A。

图18-15所示为T_1和T_2动作的一个交错，其中T_1从获得A上的共享锁开始。接着，T_2跟随其后获得A和B上的锁。现在，T_1需要B上的一个排他锁，因为它既要读B又要写B。但是，它不能获得排他锁，因为T_2已经有B上的共享锁。因此，调度器迫使T_1等待。最终T_2释放B上的锁。这时，T_1才能得以完成。 □

T_1	T_2
$sl_1(A)$; $r_1(A)$;	
	$sl_2(A)$; $r_2(A)$;
	$sl_2(B)$; $r_2(B)$;
$xl_1(B)$ 被拒绝	
	$u_2(A)$; $u_2(B)$
$xl_1(B)$; $r_1(B)$; $w_1(B)$;	
$u_1(A)$; $u_1(B)$;	

图18-15 使用共享锁和排他锁的一个调度

请注意图18-15中产生的调度是冲突可串行化的。冲突等价的串行顺序是（T_2，T_1），尽管T_1先开始。尽管我们在这里没有证明，但在18.3.4节给出的证明表明一致的2PL事务的合法调度是冲突可串行化的，论证也适用于具有共享锁和排他锁的系统。在图18-15中，T_2在T_1前解锁，所以我们能够预期在串行顺序中T_2位于T_1前。同样，我们可以查看图18-15中的读写动作，并发现可以将$r_1(A)$交换到T_2的所有动作后，但不能将$w_1(B)$移到$r_2(B)$前，而如果在某个冲突等价的
942 串行调度中T_1位于T_2前，那么这是必要的。

18.4.2 相容性矩阵

如果我们使用几种封锁方式，那么调度器需要一个关于在已知同一数据库元素上可能已经持有的锁的情况下何时能同意封锁请求的策略。尽管共享/排他系统比较简单，我们将会看到一些复杂得多的封锁方式系统也在使用中。我们因此将在下面以简单的共享/排他系统为背景，介绍描述锁授予策略的记法。

相容性矩阵中对应每种封锁方式有一行和一列。行对应于数据库元素X上另一事务已经持有的锁，而列对应于X上申请的锁方式。使用相容性矩阵做出锁授予决定的规则是：

- 我们能够授予C方式的锁，当且仅当对于其他事务在X上已经有的每个R方式锁对应的每一行R，在C列上有一个“是”。

		申请的锁	
		S	X
持有锁的方式	S	是	否
	X	否	否

图18-16 共享锁和排他锁的相容性矩阵

例18.14 图18-16是共享锁（S）和排他锁（X）的相容性矩阵。关于S的列说明，如果一个数据库元素上当前被持有的锁只有共享锁，那么我们可以授予该元素上的共享锁。关于X的列说明，只有在当前其他任何锁都不被持有时，我们才能授予一个排他锁。请注意这些规则是如何反映这一封锁系统中调度合法性的定义的。 □

18.4.3 锁的升级

占有X上的共享锁的事务T对其他事务来说是“友好的”，因为在T被允许访问X的同时其他事务也被允许访问X。因此，我们可能很想知道，如果一个想要读X并写入新值的事务T首先获得X上的一个共享锁，而仅在后来当T准备好写入新值时将锁升级为排他的（即除了它在X上已经持有的共享锁外再申请X上的一个排他锁），这样是否更友好一些。没有理由阻止事务在同一
943 数据库元素上对不同方式的锁提出申请。我们沿袭$u_i(X)$释放X上事务T_i持有的所有锁的惯例，尽管在需要用到的时候我们可以引入与方式相关的解锁动作。

例18.15 在下面的例子中，事务T_1可以和T_2并发地执行计算，如果T_1最初在B上取得排他

锁,则这是不可能的。这两个事务是:

T_1: $sl_1(A)$; $r_1(A)$; $sl_1(B)$; $r_1(B)$; $xl_1(B)$; $w_1(B)$; $u_1(A)$; $u_1(B)$;

T_2: $sl_2(A)$; $r_2(A)$; $sl_2(B)$; $r_2(B)$; $u_2(A)$; $u_2(B)$;

这里，T_1读A和B，并对它们执行某种（可能很冗长的）计算，最终使用其结果来为B写入新值。请注意，T_1先获得B上的一个共享锁，而后来，当它完成涉及A和B的计算后，再申请B的一个排他锁。事务T_2只读A和B，但不写。

图18-17给出了这些动作一个可能的调度。T_2在T_1前获得B上的共享锁，但在第四行，T_1也能以排他方式封锁B。因此，T_1有了A和B，能够使用它们的值进行计算。只有等到T_1试图将其在B上的锁升级为排他的时，调度器才必须拒绝这一请求，并迫使T_1等待T_2释放它在B上的锁。至此，T_1获得它在B上的排他锁，然后完成。

T_1	T_2
$sl_1(A)$; $r_1(A)$;	
	$sl_2(A)$; $r_2(A)$;
	$sl_2(B)$; $r_2(B)$;
$sl_1(B)$; $r_1(B)$;	
$xl_1(B)$ 被拒绝	
	$u_2(A)$; $u_2(B)$
$xl_1(B)$; $w_1(B)$;	
$u_1(A)$; $u_2(B)$;	

图18-17 锁的升级允许更多的并发操作

请注意，如果T_1最初在读B前就请求B上的排他锁，那么这一请求将被拒绝，因为T_2已经有B上的共享锁。T_1在没有读到B的情况下不能执行其计算，因此在T_2释放其锁后T_1有更多的事情需要做。所以，只使用排他锁时T_1完成得比它使用升级策略时晚。 □

例18.16 但是，不加区别地使用升级将会引入新的并且可能更严重的死锁。假设T_1和T_2分别读数据库元素A，并为A写入新值。如果两个事务都使用升级方法，首先获得A上的共享锁然后将其升级为排他锁，那么只要 T_1 和 T_2 几乎同时开始，图18-18所示的事件序列就会发生。 944

T_1	T_2
$sl_1(A)$	
	$sl_2(A)$
$xl_1(A)$ 被拒绝	
	$xl_2(A)$ 被拒绝

图18-18 两个事务的升级可能导致死锁

T_1和T_2都能得到A上的共享锁。接着，它们都试图升级到排他锁，但是由于另一个事务在A上有共享锁，调度器迫使它们中的每一个都等待。因此，二者都不能取得进展，它们各自都会永远等待，或者等到系统发现存在死锁，并中止两个事务中的一个，及给另一个事务A上的排他锁。 □

18.4.4 更新锁

通过使用第三种称为*更新锁*的封锁方式，有办法避免例18.16中的死锁问题。更新锁$ul_i(X)$只给予事务T_i读X而不是写X的权限。但是，只有更新锁能在以后升级为写锁；读锁是不能升级的。当X上已经有共享锁时我们可以授予X上的更新锁，但一旦X上有了更新锁，我们就禁止在X上加其他任何种类（共享、更新或排他）的锁。其原因是，如果我们不拒绝这样的锁，那么更新者可能由于X上总有其他的锁而永远没有机会升级到排他锁。

这一规则导致一个不对称的相容性矩阵，因为更新锁（U）在我们申请它时看起来像共享锁，而当我们已经持有它时看起来像排他锁。因此，关于S和U的列相同，关于U和X的行相同。该矩阵如图18-19所示[⊖]。

例18.17 更新锁的使用对例18.15不会产生影响。作为其第三个动作，T_1将获得B上的更新

⊖ 但请记住，还有一个关于调度合法性的条件在这个矩阵中没有反映出来：尽管我们通常并不禁止一个事务在同一元素上持有多个锁，在元素X上持有共享锁而不是更新锁的事务不能被授予X上的排他锁。

锁，而不是共享锁。但是更新锁可以被授予，因为B上被持
945 有的只有共享锁，与图18-17中相同的动作序列将会发生。

但是，更新锁解决了例18.16中的问题。现在，T_1和T_2都首先申请A上的更新锁而只在后来获得排他锁。T_1和T_2可能的描述为：

T_1: $ul_1(A)$; $r_1(A)$; $xl_1(A)$; $w_1(A)$; $u_1(A)$;

T_2: $ul_2(A)$; $r_2(A)$; $xl_2(A)$; $w_2(A)$; $u_2(A)$;

	S	X	U
S	是	否	是
X	否	否	否
U	否	否	否

图18-19 共享锁、排他锁和更新锁的相容性矩阵

与图18-18对应的事件序列如图18-20所示。现在，请求A上的更新锁的第二个事务T_2被拒绝。T_1被允许完成，然后T_2可以进行。这一封锁系统有效地阻碍了T_1和T_2的并发执行，但在这个例子中，任何相当数量的并发执行或者会导致死锁，或者会导致不一致的数据库状态。 □

T_1	T_2
$ul_1(A)$; $r_1(A)$;	
	$ul_2(A)$ 被拒绝
$xl_1(A)$; $w_1(A)$; $u_1(A)$;	
	$ul_2(A)$; $r_2(A)$;
	$xl_2(A)$; $w_2(A)$; $u_2(A)$;

图18-20 使用更新锁的正确执行

18.4.5 增量锁

另一类有趣的、在某些情况下很有用的锁是“增量锁”。很多事务都只通过增加或减少存储的值来对数据库进行操作。例如：

1) 把钱从一个银行账户转到另一个账户的事务。

946 2) 出售飞机票并减少该航班上可获得的座位计数的事务。

增量动作一个有趣的性质是这些动作相互之间是可交换的，因为如果两个事务都给同一个数据库元素加上常数，谁先做是无关紧要的，正如图18-21所示的数据库状态转换图所表明的那样。另一方面，增量与读或写都不能交换；如果在A增加以前或以后读它，得到的值是不同的，而如果在其他事务为A写入新值以前或以后增加A，则在数据库中也会得到不同的A值。

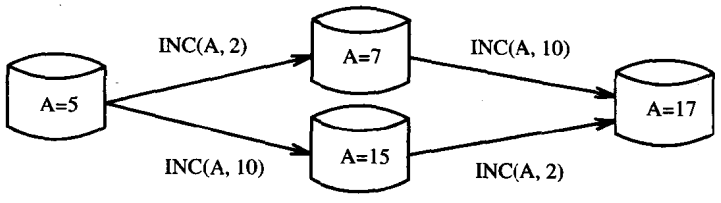


图18-21 两个增量动作可交换，因为最终数据库状态不依赖于哪个先做

让我们将增量动作作为事务中一种可能的动作引入，写作INC(A, c)。非形式地表述，这一动作将常数c加到数据库元素A上，我们假设A是单个数。请注意c可以是负的，这种情况下我们实际上是在减少A。在实践中，我们将INC施加到元组的一个组成成分上，元组自身而并非其成分之一是可封锁的元素。

更形式化地讲，使用INC(A, c)表示以下步骤的原子执行：READ(A,t);t:=t+c;WRITE(A, t);。我们不准备讨论用来使这一操作原子地执行的硬件和/或软件机制，但是我

们应该知道这种形式的原子性位于比通过封锁支持的事务的原子性更低的层次上。

与增量动作对应，我们需要一个增量锁。我们将用$il_i(X)$表示T_i请求X上的增量锁这一动作。对事务T_i在数据库元素X上增加某个常数的动作，我们还将使用简记法$inc_i(X)$；具体常数无关紧要。

增量动作和增量锁的存在需要我们对自己关于一致的事务、冲突和合法调度的定义做一些修改。这些修改包括：

a) 一致的事务只有在它持有X上的增量锁时才能在X上进行增量动作。但增量锁并不能赋予读或写动作的权力。

b) 在一个合法的调度中，任何时候都可以有任意多个事务在X上持有增量锁。但是，如果某个事务持有X上的增量锁，那么其他事务不能同时在X上既持有共享锁又持有排他锁。这些要求可以用图18-22所示的相容性矩阵表示，其中I表示一个增量方式的锁。

	S	X	I
S	是	否	否
X	否	否	否
I	否	否	是

图18-22 共享锁、排他锁和增量锁的相容性矩阵

947

c) 对$j \neq i$，$inc_i(X)$动作既与$r_j(X)$冲突又与$w_j(X)$冲突，但与$inc_j(X)$不冲突。

例18.18 考虑两个事务，每一个都读数据库元素A然后增加B。可能它们将A加到B上，或者它们在B上增加的常数以某种方式依赖于A。

T_1: $sl_1(A)$; $r_1(A)$; $il_1(B)$; $inc_1(B)$; $u_1(A)$; $u_1(B)$;

T_2: $sl_2(A)$; $r_2(A)$; $il_2(B)$; $inc_2(B)$; $u_2(A)$; $u_2(B)$;

请注意这些事务都是一致的，因为它们只在有增量锁时执行增量操作，而且只在有共享锁时执行读操作。图18-23给出T_1和T_2一种可能的交错。T_1首先读A，但接着T_2读A且增加B。但是，T_1在这时允许获得它在B上的增量锁并继续执行。

T_1	T_2
$sl_1(A)$; $r_1(A)$;	
	$sl_2(A)$; $r_2(A)$;
	$il_2(B)$; $inc_2(B)$;
$il_1(B)$; $inc_1(B)$;	
	$u_2(A)$; $u_2(B)$;
$u_1(A)$; $u_1(B)$;	

图18-23 有增量动作和增量锁的事务调度

请注意，在图18-23中，调度器不需要推迟任何请求。例如，假设T_1将B增加A，而T_2将B增加$2A$。它们可以按照两种顺序中的任何一种执行，因为A的值不变，因而增量动作也可以按照两种顺序中的任何一种执行。

换句话说，我们可以看一看图18-23中非封锁动作的序列；它们是：

$$S: r_1(A); r_2(A); inc_2(B); inc_1(B);$$

948

我们可以把最后一个动作$inc_1(B)$移到第二个位置，因为它与同一元素的另一个增量动作不冲突，而且肯定与另一个元素的读动作不冲突。这一系列交换表明S冲突等价于串行调度$r_1(A); inc_1(B); r_2(A); inc_2(B)$。类似地，我们可以通过交换把第一个动作$r_1(A)$移到第三个位置，得到一个$T_2$在$T_1$前的串行调度。 □

18.4.6 习题

习题18.4.1 对下面的事务T_1、T_2和T_3的每一个调度：

a) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(D)$;

b) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(A)$;

c) $r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(D)$; $w_1(C)$; $w_2(D)$; $w_3(E)$;

* d) $r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(D)$; $w_1(A)$; $w_2(B)$; $w_3(C)$;

e) $r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(A)$; $w_1(A)$; $w_2(B)$; $w_3(C)$;

做下列事情：

1) 插入共享锁和排他锁，并插入解锁动作。如果一个读动作后没有同一事务对同一元素的写动作，请在该读动作前紧靠它的地方放一个共享锁。在其他的每一个读动作或写动作前放一个排他锁。在每个事务的末尾放上必需的解锁。

2) 说明支持共享锁和排他锁的调度器运行每个调度时会发生什么。

3) 以一种允许升级的方式插入共享锁和排他锁。在每个读动作前放一个共享锁，在每一个写动作前放一个排他锁，并在事务的末尾放上必需的解锁。

4) 说明(3)中的支持共享锁、排他锁和升级的调度器运行每个调度时会发生什么。

5) 插入共享锁、排他锁和更新锁以及解锁动作。在每一个不会升级的读动作前放一个共享锁，在每一个将升级的读动作前放一个更新锁，并在每一个写动作前放一个排他锁。在事务的末尾照例放上解锁。

949 6) 说明(5)中的支持共享锁、排他锁和更新锁的调度器运行每个调度时会发生什么。

! **习题18.4.2** 考虑两个事务：

T_1: $r_1(A)$; $r_1(B)$; $inc_1(A)$; $inc_1(B)$;

T_2: $r_2(A)$; $r_2(B)$; $inc_2(A)$; $inc_2(B)$;

回答以下问题：

* a) 这些事务的交错中有多少是可串行化的？

b) 如果T_2中增量动作的顺序反过来［即$inc_2(B)$后面是$inc_2(A)$］，有多少可串行化的交错？

习题18.4.3 对下面的每个调度，在每个动作前插入适当的锁（读、写或增量），并在事务末尾插入解锁动作。然后说明该调度在一个支持这三类锁的调度器上运行时会发生什么。

a) $r_1(A)$; $r_2(B)$; $inc_1(B)$; $inc_2(C)$; $w_1(C)$; $w_2(D)$;

b) $r_1(A)$; $r_2(B)$; $inc_1(B)$; $inc_2(A)$; $w_1(C)$; $w_2(D)$;

c) $inc_1(A)$; $inc_2(B)$; $inc_1(B)$; $inc_2(C)$; $w_1(C)$; $w_2(D)$;

习题18.4.4 在习题18.1.1中，我们讨论了关于航班预订的一个假想事务。如果事务管理器可以获得的锁包括共享锁、排他锁、更新锁和增量锁，那么在事务的每一步你推荐使用什么锁？

习题18.4.5 乘上一个常数因子的动作可以用一个它自己的动作来建模。假设MC(X, c)表示以下步骤的原子执行：`READ(X,t);t:=c*t;WRITE(X,t);`。我们也可以引入只允许乘常数因子的一种封锁方式。

a) 给出读、写和乘常数锁的相容性矩阵。

! b) 给出读、写、增量和乘常数锁的相容性矩阵。

! **习题18.4.6** 为了便于讨论，假设数据库元素是二维向量。我们在向量上可以执行的操作有四个，每一个有它自己的锁类型。

1) 改变沿x轴的值（X锁）。
2) 改变沿y轴的值（Y锁）。
3) 改变向量的角度（A锁）。
4) 改变向量的大小（M锁）。

950

回答以下问题：

* a) 哪些操作对可交换？例如，如果我们先旋转向量使其角度为120°，然后将x坐标变为10，这和先将x坐标变为10，再将角度变为120°一样吗？

b) 根据你对(a)的回答，这四类锁的相容性矩阵是怎样的？

!! c) 假设我们改变这四个操作，不是把新的值赋给一个量，而是在量上增加（即，“在x坐标上加10”，或“顺时针将向量旋转30°”）。这时相容性矩阵又是怎样的？

! **习题18.4.7** 这里有一个丢失了一个动作的调度：

$$r_1(A);\ r_2(B);\ ???;\ w_1(C);\ w_2(A);$$

你的问题是要指出某种类型的什么操作可以替代???，并将使调度不可串行化。对以下的每类动作，说明所有可能的非可串行化替代：

* a) 读动作。
b) 写动作。
c) 更新动作。
d) 增量动作。

18.5 封锁调度器的一种体系结构

看过几种不同的封锁机制后，我们接下来需要考虑使用这些模式之一的调度器如何操作。在这里我们打算只考虑基于以下几个原则的一个简单调度器：

1. 事务自身不会申请封锁，或我们不能依赖于事务做这件事。在读、写以及其他访问数据的动作流中插入锁的动作是调度器的任务。

2. 事务不释放锁，而是调度器在事务管理器告诉它事务将提交或中止时释放锁。

18.5.1 插入锁动作的调度器

图18-24所示为一个由两部分构成的调度器，它接受来自事务的诸如读、写、提交以及中止这样的请求。调度器维护一个锁表，尽管在图中锁表是作为第二级存储器数据，但它可能部分地或全部位于主存中。通常，锁表使用的主存不是用于查询执行和日志的缓冲池的一部分。相反，锁表正是DBMS的另一组成部分，并且将像DBMS的其他代码和数据那样由操作系统为其分配空间。

951

事务请求的动作通常通过调度器传送并在数据库上执行。但是在某些情况下，事务等待一个锁而被推迟，其请求（暂时）不被传送到数据库。调度器的两个部分执行如下动作：

1. 第I部分接受事务产生的请求流，并在所有数据库访问操作如读、写、增量和更新前插入适当的锁动作。数据库访问操作接下来被传送到第II部分。不管调度器使用什么样的封锁方式集合，调度器的第I部分必须从其中选择适当的封锁方式。

2. 第II部分接受由第I部分传来的封锁和数据库访问动作序列，并正确地执行它们中的每一个。如果第II部分接收到一个封锁或数据库访问请求，那么它要决定提出请求的事务T是否由于某个锁不能被授予而被推迟。如果是，那么这个动作自身被推迟并被加入一个最终必须为事务T执行的动作列表中。如果T不被推迟（即前面它所申请的所有锁已经被授予），那么

(a) 如果动作是数据库访问，这一动作被传送到数据库并被执行。

(b) 如果第II部分接收到一个封锁动作，它将查看锁表以决定锁是否能被授予。

952

a) 如果是，则修改锁表，并将刚刚授予的锁包括进去。

b) 如果不是，那么锁表中必须加入一项以表明该锁已经被申请。调度器的第II部分接着推迟事务T进一步的动作，直到出现像锁被授予这样的时候。

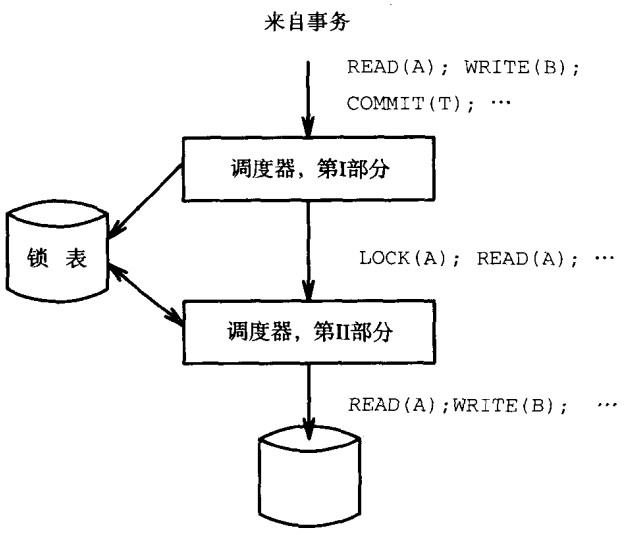


图18-24 在事务请求流中插入封锁请求的调度器

3. 当事务T提交或中止时，事务管理器将通知第I部分，于是第I部分释放T持有的所有的锁。如果有事务在等待这些锁中的任何一个，第I部分将通知第II部分。

4. 当第II部分被告知某个数据库元素X上的锁可以获得时，它决定接下来能获得X上的锁的一个或多个事务。获得锁的这个（或这些）事务被允许尽可能多地执行它们被推迟的动作，直到它（们）完成或到达另一个不能被授予的封锁请求。

例18.19 如果像18.3节中那样只有一种锁，那么调度器第I部分的工作很简单。只要它看见数据库元素X上的动作，并且它还没有为该事务插入X上的封锁请求，那么它就插入这样的一个请求。当事务提交或中止时，第I部分释放该事务的锁后就可以遗忘关于该事务的一切，所以第I部分所需要的主存不会无限增长。

当有几种锁时，调度器可能需要预先知道同一数据库元素上将发生什么动作。让我们用例18.15中的事务重新考虑共享—排他—更新锁的情况，下面我们写这些事务时没有给出其中的任何锁：

$$T_1: r_1(A);\ r_1(B);\ w_1(B);$$
$$T_2: r_2(A);\ r_2(B);$$

传给调度器第I部分的消息不仅必须包括读或写请求，还必须包括关于同一元素上将有动作的指示。特别地，当$r_1(B)$被送来时，调度器需要知道后面会有$w_1(B)$动作（或可能有这样的动作，如果T_1的代码中包含分支的话）。获得这一信息的方法有多种。例如，如果事务是一个查询，那么我们知道它不会写任何东西。如果事务是一个SQL数据库更新命令，那么查询处理器能预先确定既可能被读又可能被写的数据库元素。如果事务是一个采用嵌入式SQL的程序，那么编译器能访问所有的SQL语句（这是惟一能写数据库的语句），并且能确定可能被写的数据库元素。

在我们的例子中，假设事件按图18-17的顺序发生。那么T_1首先发出$r_1(A)$。由于将来不会将该锁升级，调度器在$r_1(A)$前插入$sl_1(A)$。下一步，来自T_2的请求（$r_2(A)$和$r_2(B)$）到达调度器。由于将来仍不会有升级，所以第I部分发出动作序列$sl_2(A)$; $r_2(A)$; $sl_2(B)$; $r_2(B)$。 953

接着，动作$r_1(B)$以及该锁可能升级的警示信息到达调度器。调度器第I部分因此发送$ul_1(B)$;$r_1(B)$给第II部分。后者查阅锁表，发现它可以把B上的更新锁授予T_1，因为B上只有共享锁。

当动作$w_1(B)$到达调度器时，第I部分发送$xl_1(B)$;$w_1(B)$。但是，第II部分不能同意$xl_1(B)$请求，因为B上有一个T_2的共享锁。来自T_1的这一动作以及后续动作被推迟，第II部分将它们存储起来等待将来执行。最后，T_2提交，第I部分释放T_2持有的A和B上的锁。这时发现，T_1在等待B上的锁。调度器第II部分被告知这一信息，并且它发现$xl_1(B)$锁现在可以获得了。它将此锁加入锁表中，并继续最大限度地执行存储的来自T_1的动作。在这个例子中，T_1完成。 □

18.5.2 锁表

抽象地说，锁表是将数据库元素与有关该元素的封锁信息联系起来的一个关系，如图18-25所示。举例来说，这个表可以用一个散列表来实现，使用数据库元素（地址）作为散列码。任何未被封锁的元素在表中不出现，因此表的大小只与被封锁元素的数目成正比，而不是与整个数据库的大小成正比。

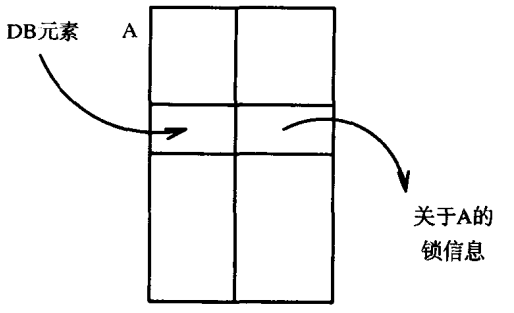


图18-25 锁表是从数据库元素到其封锁信息的映射

图18-26是我们在锁表项中所能找到信息种类的例子。这个示例结构假设调度器使用18.4.4节的共享—排他—更新锁模式。图中所示一个典型数据库元素A的表项是由以下成分构成的一个元组：

1. 组模式是对一个事务申请A上的一个新锁时所面临的最严格的条件的概括。我们并不是将封锁请求和同一元素上其他事务持有的锁一个个比较，而可以通过只比较请求与组模式来简化授予/拒绝决定[⊖]。在共享—排他—更新（SXU）的封锁方式中，规则很简单：组模式 954

(a) S表示被持有的只有共享锁。

(b) U表示有一个更新锁，而且可能有一个或多个共享锁。

(c) X表示有一个排他锁，并且没有其他的锁。

对于其他的封锁方式，总是存在使用组模式的一个适当的概括系统，我们把例子留作习题。

2. 等待位说明至少有一个事务等待A上的锁。

3. 一个列表描述所有或者在A上当前持有锁、或者在等待A上的锁的那些事务。每个列表项中的有用信息可能包括：

(a) 持有锁或等待锁的事务名。

(b) 该锁的模式。

⊖ 但是，封锁管理器必须处理发出请求的事务在同一元素上已经持有其他方式的锁的可能性。例如，在所讨论的SXU封锁系统中，如果发出请求的事务在同一元素上有U锁，封锁管理器或许可以同意其X锁请求。在不支持一个事务在同一元素上有多个锁的系统中，组模式总能为封锁管理器提供所需信息。

(c) 事务是持有锁还是等待锁。

在图18-26中，对每一项我们还给出了两个链接。一个将列表项自身链接起来，另一个将一个具体事务的所有项链接起来（图中的Tnext）。后一个链接在事务提交或中止时会用到，
955 以使得我们能比较容易地找到需要释放的所有锁。

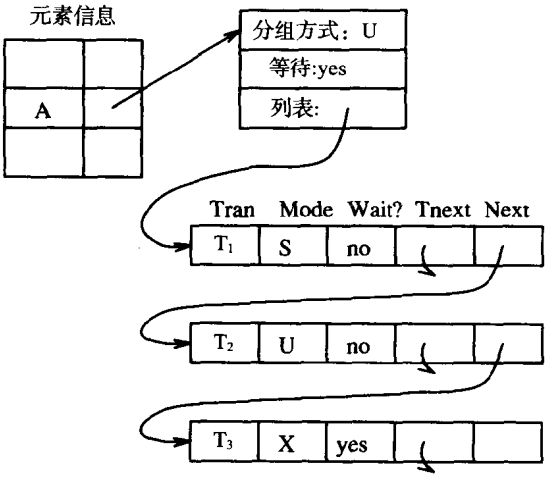


图18-26 锁表项的结构

封锁请求的处理

假设事务*T*请求*A*上的锁。如果没有*A*的锁表项，那么肯定在*A*上无锁，于是相应的表项被创建并且请求被同意。如果存在*A*的锁表项，就用它来指导我们做出有关封锁请求的决定。我们找到组模式，这在图18-26中是*U*，即“更新”。一旦元素上有更新锁，其他锁就不能被授予（除了*T*自己持有*U*锁和其他与*T*的请求相容的锁这一情况）。因此，*T*的这一请求被拒绝，而在列表中将加入表示*T*申请锁（其模式由*T*的申请而定）的一项，并且Wait? ='yes'。

如果组模式是*X*（排他的），则同样的事情将发生；但如果组模式是*S*（共享的），则另一个共享锁或更新锁可以被授予。在这种情况下，*T*在列表中的项将有Wait?='no'，并且如果新锁是更新锁，则组模式被改为*U*；否则组模式保持*S*。不管锁是否被授予，新的列表项通过Tnext和Next字段正确地链接起来。请注意，不管锁是否被授予，调度器可以从锁表得到所需信息而不必检查锁的列表。

解锁的处理

现在假设事务*T*解锁*A*。列表中*T*关于*A*的项被删除。如果*T*持有的锁与组模式不同（例如，*T*持有*S*锁，而组模式为*U*），则不需要改变组模式。另一方面，如果*T*的锁处于组模式，我们可能不得不检查整个列表以找出新的组模式。在图18-26的例子中，我们知道在一个元素上只能有一个更新锁，因此当该锁被释放时，新的组模式只能是*S*（如果还存在共享锁的话），或什么也没有（如果当前没有其他锁被持有）[⊖]。如果组模式是*X*，我们知道不会有其他锁；而如果组模式是*S*，则需要判定是否有其他共享锁。

如果Waiting的值为'yes'，我们需要授予申请锁列表中的一个或多个锁。有几种不同的方式，它们各有其优点：

⊖ 我们永远也不会看到组模式是“什么也没有”，因为如果元素上既没有锁也没有锁请求，那么锁表中就没有关于该元素的项。

1. 先来先服务：同意等待时间最长的封锁请求。这种策略保证不会饿死，即一个事务永远等待一个锁的情况。

2. 共享锁优先：首先授予所有等待的共享锁。接着，如果有等待的更新锁，则授予一个更新锁。只在没有其他锁等待时才授予排他锁。这一策略允许等待*U*或*X*锁的事务饿死。 956

3. 升级优先：如果有一个持有*U*锁的事务等待将其升级到*X*锁，则首先授予该锁。否则，采用已经提到的其他策略中的一个。

18.5.3 习题

习题18.5.1 锁表中合适的组模式是哪些，如果使用的锁模式为：

a) 共享锁与排他锁。

*! b) 共享锁、排他锁和增量锁。

!! c) 习题18.4.6中的封锁模式。

习题18.5.2 对习题18.2.4中的各个调度，说明本节中描述的封锁调度器将要执行的步骤。

18.6 数据库元素层次的管理

现在回到我们从18.4节开始的对于不同封锁方式的探索。特别地，我们将主要关注在我们的数据中存在树结构时出现的两个问题。

1. 我们遇到的第一类树结构是可封锁元素的层次结构。在这一节中，我们讨论如何才能既允许大的元素如关系上的锁，又允许包含于其中的较小元素（如容纳关系中几个元组的块，或单个元组）上的锁。

2. 并发控制系统中另一类重要的层次是本身就组织为一棵树的数据。一个重要的例子是B树索引。我们可以将B树的结点看做数据库元素，但如果这样做，那么就像我们将在18.7节中看到的那样，到目前为止所研究的封锁方式的性能就会很低，而我们需要使用一种新的方法。

18.6.1 多粒度的锁

回忆一下，我们故意不给出“数据库元素”这一术语的定义，因为不同的系统用不同大小的数据库元素封锁，例如元组、页或块，以及关系。有的应用受益于小的数据库元素如元组，而另一些使用大的元素时最好。

例18.20 考虑银行的数据库。如果我们将关系作为数据库元素，并因而对整个关系（如给出账户余额的关系）只有一个锁，那么系统只能允许极少的并发。由于大多数事务都将或正 957
或负地改变账户余额，因而大多数事务都需要帐户关系上的一个排他锁。因此，同一时间只有一个存款或取款能进行，不管我们有多少处理器可以用来执行这些事务。一种比较好的方法是给单独的页或数据块上锁。这样，对应元组位于两个不同块上的账户就可以同时被更新，同时提供了系统中几乎所有可能的并发。极端的情况是为每个元组提供一个锁，那么不管什么样的账户集合都可以同时更新，但这样细的锁粒度或许不值它需要付出的特别大的代价。

作为对照，考虑文档的一个数据库。这些文档可能不时地被编辑，但大多数事务将检索整个文档。明智的选择是将整个文档作为数据库元素。由于大多数事务是只读的（即它们不会执行任何写动作），封锁只是为了避免读一个正在编辑中的文档。如果我们使用粒度更小的锁，例如图、语句或单词，基本上不会带来好处而只会增加开销。较小粒度的锁可以支持的惟一活动是，当同一文档的其他部分正在被修改时可以读文档的某些部分。 □

一些应用既能使用大粒度锁又能使用小粒度锁。例如，例18.20讨论的银行数据库显然需要块级或元组级封锁，但也可能在某些时候为了审计账户（例如，检查账户的总和是否正确）

而需要整个账户关系上的锁。但是，为了计算账户关系上的某个聚集而获得该关系上的一个共享锁，而同时在单个的账户元组上有排他锁，这很可能导致非可串行化行为，因为聚集查询在读假设被冻住的关系拷贝时，此关系事实上正在改变。

18.6.2 警示锁

解决管理不同粒度锁这一问题的方法牵涉到一种称为“警示”的新的锁。这样的锁在数据库元素形成嵌套或层次结构时很有用，如图18-27所示。其中，我们可以看到三个级别的数据库元素：

1. 关系是最大的可封锁元素。

2. 每个关系由一个或多个块或页组成，每个块或页上存储了关系的元组。

3. 每个块包含一个或多个元组。

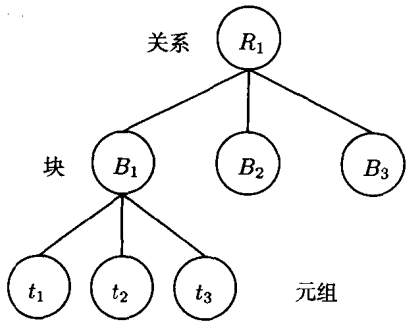


图18-27 层次组织的数据库元素

在数据库元素的层次上管理锁的规则由警示协议构成，它既包括“普通”锁又包括“警示”锁。我们将描述普通锁是*S*和*X*（共享和排他）时的封锁方式。警示锁将通过在普通锁前加前缀*I*（意为“打算”）表示；例如*IS*表示获得子元素上的一个共享锁的意向。警示协议的规则是：

958

1. 要在任何元素上加*S*或*X*锁，我们必须从层次结构的根开始。

2. 如果处于我们将要封锁的元素的位置，则不需要进一步查找。我们请求该元素上的*S*或*X*锁。

3. 如果我们希望封锁的元素在层次结构中更靠下，那么在这一结点上加一个警示锁；也就是说，如果我们想要获得其子元素上的共享锁，则请求该结点上的*IS*锁，如果我们想要获得其子元素上的排他锁，则请求该结点上的*IX*锁。当前结点上的锁被授予后，我们继续向适当的子结点（其子树包含我们希望上锁的结点的那一个）行进。接下来适当地重复步骤2）和步骤3），直到到达所需结点。

为了决定这些锁中的一个是否能授予，我们使用图18-28的相容性矩阵。为了明白为什么这个矩阵是有意义的，我们首先考虑*IS*列。当请求结点*N*上的*IS*锁时，我们打算读*N*的一个后裔。这个意向惟一会产生问题的是另外的某个事务已经声明了用*N*表示的整个数据库元素写入新拷贝的权利时；因此在关于*X*的行上看到“否”。请注意，如果另外的某个事务只打算写一个子元素，这通过*N*上的*IX*锁表明，那么我们能够承受在*N*授予*IS*锁。如果写意向与读意向恰好涉及共同的元素，则允许在较低的层次上解决冲突。

959

	IS	IX	S	X
IS	是	是	是	否
IX	是	是	否	否
S	是	否	是	否
X	否	否	否	否

图18-28 共享锁、排他锁和意向锁的相容性矩阵

现在考虑*IX*列。如果我们打算写结点*N*的一个子元素，那么必须防止对由*N*表示的整个元素的读和写。因此，我们在关于*S*和*X*的项中看到“否”。但是，按照我们对*IS*列的讨论，读或写一个子元素的另一个事务潜在的冲突可以在该子元素的层次上解决，所以*IX*与*N*上的另一个*IX*或*N*上的*IS*不冲突。

接下来，考虑*S*列。读对应于结点*N*的元素不会与*N*上的另一个读锁或*N*的某个子元素上的读锁发生冲突，后者由*N*处的*IS*表示。因此，我们在关于*S*和*IS*的行上都看到“是”。然而，*X*或*IX*都表示另外的某个事务至少会写由*N*表示的数据库元素的部分。因此，我们不能授予读整个

N的权利，这解释了S列上为“否”的项。

最后，X列上只有“否”。如果另外的某个事务已经有权利读写N或获得一个子元素上的读写权限，我们就不能允许写整个结点N。

例18.21 考虑关系

```
Movie (title, year, length, studioName)
```

意向锁的组模式

图18-28中的相容性矩阵展示了我们以前未见过的关于封锁模式能力的一种情况。在前面的封锁模式中，只要有可能同时将数据库元素以M或N模式封锁，其中一种模式必然在这样的意义上优于另一种模式：只要后者在行或列上有“否”，那么前者分别在对应行或列的位置上是“否”。例如，在图18-19中，我们看到U优于S，而X优于U和S。知道元素上总存在优势锁的好处是，我们可以用一个“组模式”概括多个锁的效果，正如18.5.2节讨论的那样。

正如我们从图18-28中看到的那样，S和IX模式相互不优于对方。此外，一个元素可能同时以S模式和IX模式封锁，只要这些锁是同一个事务申请的（回忆一下，相容性矩阵中的“否”项只适用于其他某个事务持有的锁）。事务可能申请这两个锁，如果它希望读整个元素，然后写其子元素的一个较小的子集。如果事务在元素上既有S有锁又有IX锁，那么它对其他事务的限制程度是其他的任一个锁都做不到的。也就是说，我们可以设想另一个封锁模式SIX，它的行和列除了关于IS的项以外全是“否”。如果一个事务有S和IX模式的锁而没有X模式的锁时，封锁模式SIX将充当组模式。

顺便说说，我们可以设想在图18-22 关于增量锁的矩阵中将会发生同样的情况。也就是说，一个事务可以既有S模式的锁又有I模式的锁。但是，这种情况等价于持有X模式的锁，于是在这种情况下我们可以使用X作为组模式。

假设有整个关系上的锁和单个元组上的锁。接着，由查询

```
SELECT *
FROM Movie
WHERE title = 'King Kong';
```

构成的事务T_1从获得整个关系上的IS锁开始。然后该事务转到单个的元组（有两部名字为*King Kong*的影片），并在它们中的每一个上获得S锁。

现在，假设当我们在执行第一个查询时，事务T_2开始，它改变一个元组中的年这一成分：

```
UPDATE Movie
SET year = 1939
WHERE title = 'Gone With the Wind';
```

T_2需要该关系上的一个IX锁，因为它打算为其中的一个元组写入新值。T_1在关系上的IS锁是相容的，因此锁被授予。当T_2来到关于*Gone With the Wind*的元组，发现那里没有锁，于是得到其X锁并重写元组。如果T_2试图在*King Kong*影片之一的元组中写入新值，它将必须等到T_1释放其S锁，因为S和X是不相容的。锁的集合如图18-29所示。 □

960

18.6.3 幻像与插入的正确处理

当事务创建可封锁元素的一个新的子元素时，有时候可能出错。问题在于我们只能封锁已

经存在的项；封锁并不存在但以后可能被插入的数据库元素没有简单的方法。下面的例子说明了这一点。

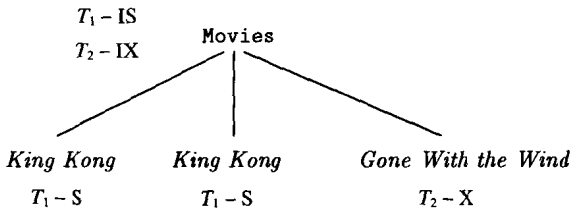


图18-29 访问Movie元组的两个事务被授予的锁

961 **例18.22** 假设我们有和例18.21中一样的关系Movie，而首先执行的事务是T_3，它是查询

```
SELECT SUM(length)
FROM Movie
WHERE studioName = 'Disney';
```

T_3需要读Disney影片的所有元组，所以它可能首先在该关系上获得*IS*锁，并在Disney影片的每个元组上获得*S*锁[⊖]。

现在，事务T_4出现并插入一个新的Disney影片。看起来似乎T_4不需要锁，但它已经使T_3的结果不正确。这一事实本身并不是并发的问题，因为串行顺序（T_3，T_4）与真正发生的等价。但是，也可能有其他某个T_3和T_4都写但T_4先写的元素X，因此在更复杂的事务中可能有不可串行化的行为。

为了更精确地说明，假设D_1和D_2是原来已经存在的Disney影片，而D_3是T_4插入的新Disney影片。设L是T_3计算出的Disney影片的长度之和，并假设数据库上的一致性约束是L应该等于最后一次计算L时存在的所有Disney影片的长度之和。那么下面是在警示协议下合法的一个事件序列：

$$r_3(D_1);\ r_3(D_2);\ w_4(D_3);\ w_4(X);\ w_3(L);\ w_3(X);$$

这里，我们用$w_4(D_3)$表示事务T_4创建D_3。上面的调度不是可串行化的。特别地，L的值不是D_1、D_2和D_3的长度之和，而它们是当前的Disney影片。此外，X具有由T_3写入而非T_4写入的值这一事务排除了在假定的等价串行调度中T_3位于T_4前的可能性。 □

例18.22中的问题是，新Disney影片有一个幻像元组，该元组应该被上锁却没有上锁，因为在获得锁时它还不存在。但是，有一种避免幻像发生的简单方法。我们必须将元组的插入或删除看做整个关系上的写操作。因此，例18.22中的事务T_4必须获得关系Movie上的X锁。由于T_3
962 已经以*IS*模式封锁了这个关系，而该模式与模式X不相容，T_4必须等到T_3完成。

18.6.4 习题

习题18.6.1 为了有多样性，我们考虑一个面向对象的数据库。类C的对象存在两个块B_1和B_2上。块B_1包含对象O_1和O_2，而块B_2包含对象O_3、O_4和O_5。类外延、块和对象构成可封锁数据库元素的一个层次结构。对下面的请求序列，说明封锁请求和基于警示协议的调度器的响应构成的序列。可以假设所有请求正好在它们被需要前发生，而所有解锁发生在事务末尾。

⊖ 但如果有很多Disney影片，则只在整个关系上获得一个*S*锁可能效率更高。

* a) $r_1(O_1)$; $w_2(O_2)$; $r_2(O_3)$; $w_1O_4)$;

b) $r_1(O_5)$; $w_2(O_5)$; $r_2(O_3)$; $w_1O_4)$;

c) $r_1(O_1)$; $r_1(O_3)$; $r_2(O_1)$; $w_2(O_4)$; $w_2(O_5)$;

d) $r_1(O_1)$; $r_2(O_2)$; $r_3(O_1)$; $w_1(O_3)$; $w_2(O_4)$; $w_3(O_5)$; $w_1(O_2)$;

习题18.6.2 修改例18.22中的动作序列，使$w_4(D_3)$成为T_4对整个Movie关系所做的一个写动作。接下来给出基于警示协议的调度器针对此请求序列的动作。

!! **习题18.6.3** 说明如何在基于警示协议的调度器中加入增量锁。

18.7 树协议

本节中我们考虑涉及元素树的另一个问题。18.6节中处理由数据库元素的嵌套结构形成的树，其中子结点是父结点的子部分。现在，我们处理由元素自身的链接方式形成的树结构。数据库元素是不相交的数据片段，但到达结点的惟一方式是通过其父结点；B树是这类数据的重要例子。知道我们必须沿着通向元素的特定路径，这给了我们以不同于到目前为止所看到的两阶段封锁方式来管理锁的重要自由。

18.7.1 基于树的封锁的动机

让我们考虑B树索引，这一系统中将单个结点（即块）看做可封锁的数据库元素。结点是正确的封锁粒度，因为将更小的片段看做元素不会带来好处，而将整个B树看做一个数据库元素阻止了在使用锁时通过构成18.7节主题的机制所能获得的那一类并发。

如果我们使用标准的诸如共享、排他和更新锁这样的封锁模式集合，并且使用两阶段封锁，那么B树的并发使用几乎是不可能的。原因在于，每个使用索引的事务必须从封锁B树的根结 963
点开始。如果事务是2PL的，那么在它获得B树结点和其他数据库元素上所有需要的锁以前不能解锁根结点[⊖]。此外，由于原则上任何插入或删除的事务可能最终重写B树的根，事务至少需要根结点上的一个更新锁，或更新锁不能获得时的一个排他锁。因此，任何时刻都只有一个非只读的事务能访问B树。

但是，在大多数情况下，我们几乎可以立即推断B树结点不会被重写，即使事务插入或删除元组。例如，如果事务插入一个元组，但我们所访问的根的子结点并不是全满的，那么我们知道插入不会向上传播到根。类似地，如果事务删除单独的一个元组，而我们所访问的根的子结点中码和指针的数目超过最小数目，那么可以肯定根不会改变。

因此，一旦事务移到根的子结点并观察到（非常常见的）不再重写根的情况，那么我们非常乐意释放根上的锁。这同样适用于B树中任何内部结点上的锁，尽管大多数并发的B树访问机会来自于提早释放根上的锁。不幸的是，提早释放根上的锁会违背2PL，因此我们不能确定访问B树的几个事务的调度是可串行化的。解决方法是为访问像B树这样的树结构数据的事务采用专门的协议。这一协议违背2PL，但使用访问元素必须沿树向下这样一个事实来保证可串行性。

18.7.2 访问树结构数据的规则

以下对锁的限制构成了树协议。我们假设只有一种锁，由形式为$l_i(X)$的封锁请求表示，但这一思路可以推广到其他任何封锁模式集合。我们假设事务是一致的，且调度必须是合法的（即，调度器通过只在申请的锁与结点上已有的锁不冲突时才授予锁来实施约束），但事务上没

⊖ 另外，事务将持有所有锁直至它准备提交，这有很好的理由；见19.1节。

有两阶段提交的要求。

1. 事务的第一个锁可以在树的任何结点上[⊖]。

2. 只有事务当前在父结点上持有锁时才能获得后续的锁。

964 3. 结点可以在任何时候解锁。

4. 事务不能对一个它已经解锁的结点重新上锁，即使它在该结点的父结点上仍持有锁。

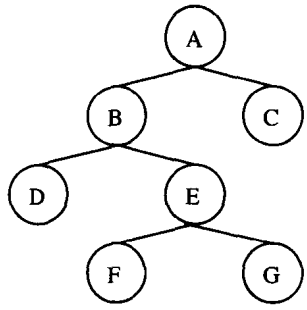


图18-30 可封锁元素的树

例18.23 图18-30给出了结点的一个层次结构，而图18-31说明三个事务在这一数据上的动作。T_1从根A开始，并继续向下进行到B、C和D。T_2从B开始，并试图移到E，但其移动一开始就由于T_3在E上的锁而被拒绝。事务T_3从E开始，并移到F和G。请注意，T_1不是2PL事务，因为A上的锁在D上的锁获得以前释放。类似地，T_3不是2PL事务，尽管T_2恰好是2PL的。 □

18.7.3 树协议发挥作用的原因

树协议在调度中涉及的事务上强制实现一个串行顺序。我们可以定义先后次序如下。如果在调度S中，事务T_i和T_j封锁一个共同的结点，而T_i先封锁该结点，我们就说$T_i <_s T_j$。

例18.24 在图18-31的调度S中，我们发现T_1和T_2都封锁B，而T_1先封锁。因此$T_1 <_s T_2$。

T_1	T_2	T_3
$l_1(A)$; $r_1(A)$;		
$l_1(B)$; $r_1(B)$;		
$l_1(C)$; $r_1(C)$;		
$w_1(A)$; $u_1(A)$;		
$l_1(D)$; $r_1(D)$;		
$w_1(B)$; $u_1(B)$;		
	$l_2(B)$; $r_2(B)$;	
		$l_3(E)$; $r_3(E)$;
$w_1(D)$; $u_1(D)$;		
$w_1(C)$; $u_1(C)$;		
	$l_2(E)$ 被拒绝	
		$l_3(F)$; $r_3(F)$;
		$w_3(F)$; $u_3(F)$;
		$l_3(G)$; $r_3(G)$
		$w_3(E)$; $u_3(E)$;
	$l_2(E)$; $r_2(E)$;	
		$w_3(G)$; $u_3(G)$
	$w_2(B)$; $u_2(B)$;	
	$w_2(E)$; $u_2(E)$;	

图18-31 三个遵循树协议的事务

我们还发现T_2和T_3都封锁E，而T_3先封锁。因此$T_3 <_s T_2$。但是，在T_1和T_3之间没有先后次序，因为它们不封锁共同的结点。所以，由这些先后次序关系派生的优先图如图18-32所示。 □

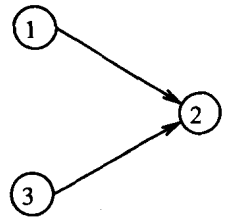


图18-32 图18-31的调度所派生的优先图

如果根据上面定义的先后次序关系画出的优先图中没有环，那么我们断言事务的任何拓扑顺序都是一

⊖ 在18.7.1节的B树例子中，第一个锁将总在根上。

个等价串行调度。例如，（T_1，T_3，T_2）或（T_3，T_1，T_2）都是图18-31的一个等价串行调度。原因在于这样一个串行调度中所有结点被触及的顺序与它们在原始调度中一样。

为了理解为什么上面描述的优先图必须总是无环的，让我们首先看一看下面的事实： 965

- 如果两个事务有几个二者都要封锁的元素，那么这些元素被封锁的顺序相同。

考虑两个事务T和U，它们封锁两个或更多共同的项。首先，注意每个事务封锁一个形成一棵树的元素集合，并且两棵树的交自身也是一棵树。因此，T和U都封锁的元素中有某个最高的元素X。假设T先锁X，但还有另外的某个元素Y是U在T前锁的。那么在元素的树中有一条从X到Y的路径，并且T和U都必须封锁沿着路径的每个元素，因为除非在结点的父结点上有锁，否则二者都不能封锁该结点。 966

考虑沿着这条路径上U先封锁的第一个元素，比如说是Z，如图18-33所示。那么T比U先锁Z的父结点P。但是这样当T封锁Z时它仍持有P上的锁，因此U在锁Z时还没有锁P。Z不可能是U与T共同封锁的第一个元素，因为它们都有锁祖先X（也可能是P，但不会是Z）。因此，U锁Z必须等到获得P上的锁后，这是在T锁Z后。我们的结论是，在每一个T和U共同封锁的结点上，T都在U前。

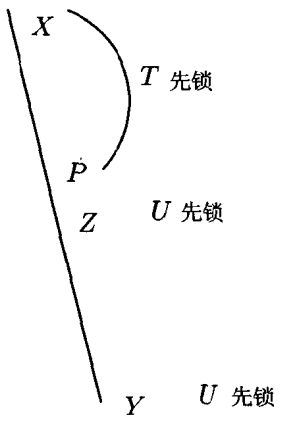


图18-33 两个事务封锁元素的一条路径

现在，我们考虑任意事务集合$T_1,T_2,\cdots$，T_n，它们遵循树协议，并且根据调度S封锁树中的某些结点。首先，封锁根的那些事务按照某个顺序来做这件事，并且按照我们刚刚发现的规则：

- 如果T_i在T_j前封锁根，那么T_i在T_j前封锁每一个T_i与T_j都要封锁的结点。也就是$T_i <_S T_j$，但不是$T_j <_S T_i$。

我们可以通过对树的结点数进行归纳来证明，对于整个事务集合上的调度S，必然有某个与之等价的串行调度。

基础：如果只有一个结点，即根，那么正如我们已经观察到的那样，事务封锁根的顺序就可以承担这样的角色。

归纳：如果树中不止一个结点，则对根的每一棵子树考虑在该子树中封锁一个或多个结点的事务集合。注意，封锁根的事务可能属于多棵子树，但不封锁根的事务只会属于一棵子树。例如，在图18-31的事务中，只有T_1封锁根，而它属于两棵子树——以B为根的树和以C为根的树。然而，T_2和T_3只属于以B为根的树。

根据归纳假设，封锁任一子树中的结点的所有事务有一个串行的顺序。我们只需要将不同子树的串行顺序混合起来。由于这些事务列表中共有的事务只是封锁根的那些事务，而我们已 967
经证明这些事务封锁每一个公共结点的顺序都与它们封锁根的顺序一样，封锁根的两个事务不可能在两个子列表中按不同的顺序出现。具体地说，如果T_i和T_j出现在根的某个子结点C的列表中，那么它们封锁C的顺序与它们封锁根的顺序一样，故在该列表中也以该顺序出现。因此，我们可以从封锁根的事务开始，建立所有事务的一个串行顺序。在将这些事务按照正确顺序排放后，我们把不封锁根的那些事务按照某种与其子树的串行顺序不冲突的顺序散布其中。

例18.25 假设有10个事务$T_1,T_2,\cdots$，T_{10}，并且在这些事务中T_1、T_2和T_3以此顺序封锁根。我们还假设根有两个子结点，第一个被T_1到T_7封锁，而第二个被T_2、T_3、T_8、T_9和T_{10}封锁。假设第一棵子树的串行顺序是（T_4，T_1，T_5，T_2，T_6，T_3，T_7）；注意这一顺序中必须按顺序包括T_1、T_2和T_3。还假设第二棵子树的串行顺序是（T_8，T_2，T_9，T_{10}，T_3）。封锁根的事务T_2和T_3在这个序列中的顺序必然和它们封锁根的顺序相同。

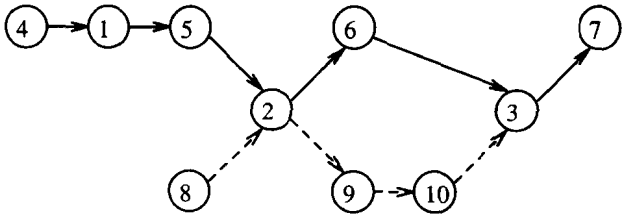


图18-34 将子树的串行顺序组合成所有事务的串行顺序

这些事务上的串行顺序受到的约束如图18-34所示。实线表示由根的第一个子结点导致的约束，而虚线表示第二个子结点中的顺序。（T_4，T_8，T_1，T_5，T_2，T_9，T_6，T_{10}，T_3，T_7）是该图的众多拓扑顺序中的一个。 □

18.7.4 习题

习题18.7.1 假设我们在图13-23所示的B树上执行下列动作。如果我们使用树协议，那么什么时候可以释放各个被搜索结点上的写锁？

* a) 插入10。

b) 插入20。

968 c) 删除5。

d) 删除23。

! **习题18.7.2** 考虑在图18-30的树上操作的如下事务。

T_1: $r_1(A)$; $r_1(B)$; $r_1(E)$;

T_2: $r_2(A)$; $r_2(C)$; $r_2(B)$;

T_3: $r_3(B)$; $r_3(E)$; $r_3(F)$;

回答以下问题：

* a) 如果T_1和T_2遵循树协议，它们可以以多少种方式进行交错？

b) 如果T_1和T_3遵循树协议，它们可以以多少种方式进行交错？

!! c) 如果这三个事务遵循树协议，它们可以以多少种方式进行交错？

! **习题18.7.3** 假设有八个事务$T_1, T_2, \cdots, T_8$，其中序号为奇数的事务T_1、T_3、T_5和T_7按此顺序封锁树的根。根有三个子结点，第一个被T_1、T_2、T_3和T_4按此顺序封锁，第二个被T_3、T_6和T_5按此顺序封锁，第三个被T_8和T_7按此顺序封锁。这些事务和以上所述一致的串行顺序有多少？

!! **习题18.7.4** 假设我们使用具有分别用于读和写的共享锁和排他锁的树协议。要求获得一个结点上的锁需要持有其父结点上的锁的规则（2）必须做出修改，以防止非可串行化行为。关于共享锁和排他锁的正确的规则（2）是什么？提示：父结点上的锁必须和其子结点上的锁类型相同吗？

18.8 使用时间戳的并发控制

接下来，我们将考虑封锁以外的、某些系统中用来保证事务可串行性的两种方法：

1. *时间戳*。我们给每个事务分配一个“时间戳”，记录最后读和写每个数据库元素的事务的时间戳，并比较这些值以保证按照事务的时间戳排序的串行调度等价于事务的实际调度。这

种方法是本节的主题。

2. 有效性确认。当事务将要提交时，我们检查事务的时间戳和数据库元素，这个过程称为事务的“有效性确认”。按照有效性确认时间对事务进行排序的串行调度必须等价于实际的调度。有效性确认方法在18.9节讨论。

这两种方法都假设没有非可串行化行为发生，并且只在违例很明显时做修复，在这个意义上它们是优化的。与此相反，所有封锁方法假设，如果不预防事务陷入非可串行化行为中，事情就会出错。优化的方法不同于封锁的地方在于，当确实发生问题时惟一的补救措施是中止并重启试图参与非可串行化行为的事务。与此相反，封锁调度器会推迟事务，但不中止它们[⊖]。通常，当很多事务为只读时优化的调度器比封锁好，因为这些事务自己永远不会导致非可串行化行为。 969

18.8.1 时间戳

为了使用时间戳来作为并发控制方式，调度器需要赋给每个事务T一个惟一的数值，即其时间戳TS(T)。时间戳必须在事务首次通知调度器自己将开始时按升序发出。产生时间戳的两种方法是：

a) 创建时间戳的一种可能的方法是使用系统时钟，只要调度器操作不会快到在一个时钟周期内给两个事务赋予时间戳。

b) 另一种方法是调度器维护一个计数器。每当一个事务开始时，计数器加1，而新的值便成为该事务的时间戳。在这种方法中，时间戳与“时间”无关，但它们具有任何时间戳产生系统都需要的重要性质：晚开始的事务比早开始的事务具有的时间戳要高。

不管使用什么时间戳产生方法，调度器都必须维护当前活跃事务及其时间戳的一张表。

为了使用时间戳来作为并发控制方式，我们需要将每个数据库元素X与两个时间戳以及一个附加的位联系起来：

1. RT(X)，X的读时间，它是读X的事务中最高的时间戳。

2. WT(X)，X的写时间，它是写X的事务中最高的时间戳。

3. C(X)，X的提交位，该位为真当且仅当最近写X的事务已经提交。这一位的目的是为了避免出现事务T读另一事务U所写的数据然后U中止这样的情况。T脏读“未提交数据”这一个问题肯定有可能会导致数据库状态变得不一致，而任何调度器都需要防止脏读的机制[⊜]。 970

18.8.2 物理上不可实现的行为

为了理解基于时间戳调度器的体系结构和规则，我们需要记住调度器假设事务的时间戳顺序也必须是它们看起来要执行的串行顺序。因此，调度器的任务除了分配时间戳和更新数据库元素的RT、WT和C外，还要核查在读写发生的任何时候，如果每个事务在对应其时间戳的那一刻瞬时执行的话，事实上发生的事就会发生。如果不是，我们说这一行为是物理上不可实现的。可能发生的问题有两类：

1. 过晚的读：事务T试图读数据库元素X，但X的写时间表明X现有的值是T理论上执行以后写入的；即TS(T) < WT(X)。图18-35说明这一问题。水平轴表示事件发生的实际时间。虚线将实际的事件与其理论上发生的时间（即执行事件的事务的时间戳）连起来。因此，我们看到事务U在事务T后开始，但在T读X前为X写入一个值。T应该不能读U写入的值，因为理论上U在T

⊖ 这并不是说使用封锁调度器的系统永远不会中止事务；例如，19.3节讨论中止事务以修复死锁。但是，封锁调度器从不简单地把中止事务当作封锁请求不能被同意时的回应。

⊜ 尽管商用系统通常给用户提供允许脏读的选择。

后执行。但是，T别无选择，因为U关于X的值是T现在所看到的。遇到这一问题时，解决办法是中止T。

2. 过晚的写：事务T试图写数据库元素X，但X的读时间表明另外的某个事务应该读到T写入的值却读到另外的某个值。也就是$WT(X) < TS(T) < RT(X)$。图18-36说明了这一问题。其中我们看到，事务U在事务T后开始，但在T有机会写X前读X。当T试图写X时，我们发现$RT(X) >$ 971 $TS(T)$，意味着X已经被一个理论上在T后执行的事务U所读。我们还发现$WT(X) < TS(T)$，意味着没有其他事务往X中写入能覆盖T所写值的值，因此取消T将其值写入X这一任务，使U能读它。

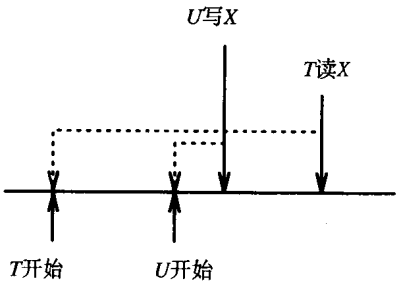


图18-35 事务T试图做过晚的读

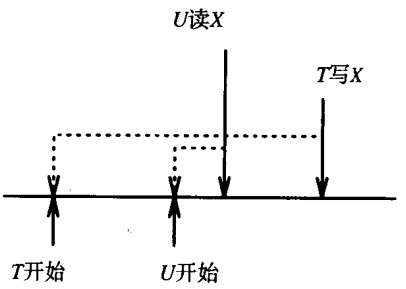


图18-36 事务T试图做过晚的写

18.8.3 脏数据的问题

提交位的设计是为了帮助解决一类问题。其中一个问题是“脏读”，如图18-37所示。其中，事务T读X，而X是最近被U写入的。U的时间戳小于T的时间戳，且在实际中T的读发生在U的写之后，因此这一事件看来在物理上是可实现的。但是，有可能在T读U写入的X值后，事务U中体；也可能是U在自己的数据中遇到了一个错误的情况，如除0，或像我们将在18.8.4节看到的那样，调度器由于事务U试图做物理上不可实现的事而将其中止。因此，尽管关于T读X并没有什么是物理上不可实现的，但最好将T的读推迟到U提交或中止后。我们可以断定U尚未提交，因为提交位$C(X)$为假。

另一个潜在的问题如图18-38所示。其中，时间戳比T晚的事务U先写X。当T试图写时，正 972 确的动作是什么也不做。显然不会有另一个事务V应该读T的X值却读到U的值，因为如果V试图读X，它将因为过晚的读而中止。以后对V执行的读将需要U的X值或一个更晚写入的X值，而不是T的。这一想法，即写操作在写时间更晚的写操作已发生时可以被跳过，称为Thomas写法则。

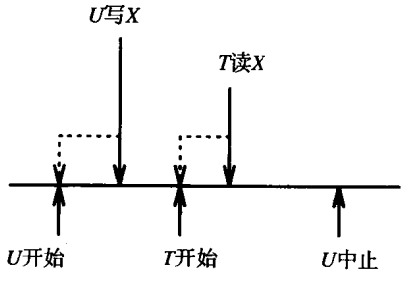


图18-37 如果T在如图所示时候读X,则可能进行的是脏读

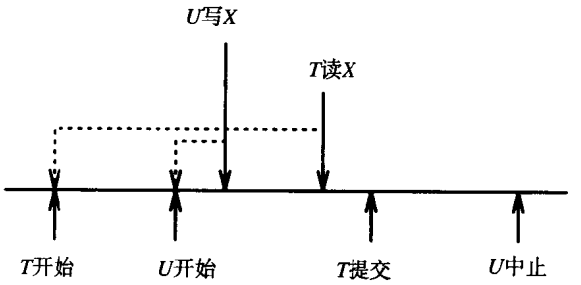


图18-38 写操作由于一个具有较晚时间戳的写操作的而被取消，但书写器终止

但是，Thomas写法则有一个潜在的问题。如果像图18-38所示的那样，U后来中止，那么它的X应该被删掉，并且前一个值和写时间应该被恢复。由于T已提交，看起来似乎T所写入的

X值应该是以后读到的值。但是，我们已经跳过T的写，而且已经来不及修复了。

尽管有多种处理上述问题的方法，但我们将采用一个相对简单的策略，它基于下面假设的基于时间戳的调度器所具有的能力。

- 当事务T写数据库元素X时，写是“尝试性的”且在T终止时可以被撤销。提交位$C(X)$被设为假，调度器保存X的旧值和原有$\mathrm{WT}(X)$的一个拷贝。

18.8.4 基于时间戳调度的规则

我们现在可以概括使用时间戳的调度器为了保证不会发生物理上不可实现的事所必须遵守的规则。作为对来自事务T的读写请求的响应，调度器可以有如下选择：

a) 同意请求，

b) 中止T（如果T违背事实）并重启具有新时间戳的T（中止再加上重启常称为回滚）；或

c) 推迟T并在以后决定是中止T还是同意请求（如果请求是读并且此读可能是脏的，正如18.8.3节那样）。

规则如下：

973

1. 假设调度器收到请求$r_T(X)$。
   (a) 如果$\mathrm{TS}(T) \geqslant \mathrm{WT}(X)$，此读是物理上可实现的。
      i. 如果$C(X)$为真，同意请求。如果$\mathrm{TS}(T) > \mathrm{RT}(X)$，置$\mathrm{RT}(X) := \mathrm{TS}(T)$；否则不改变$\mathrm{RT}(X)$。
      ii. 如果$C(X)$为假，推迟T直到$\mathrm{C}(X)$为真或写X的事务中止。
   (b) 如果$TS(T) < \mathrm{WT}(X)$，此读是物理上不可实现的。回滚T；即，中止T并以一个新的、更大的时间戳重启它。
2. 假设调度器收到请求$w_T(X)$。
   (a) 如果$\mathrm{TS}(T) \geqslant \mathrm{RT}(X)$并且$\mathrm{TS}(T) \geqslant \mathrm{WT}(X)$，此写是物理上可实现的并且必须执行。
      i. 为X写入新值；
      ii. 置$\mathrm{WT}(X) := \mathrm{TS}(T)$，并且
      iii. 置$C(X) :=$ false。
   (b) 如果$\mathrm{TS}(T) \geqslant \mathrm{RT}(X)$，但是$\mathrm{TS}(T) < \mathrm{WT}(X)$此写是物理上可实现的，但$X$中已经有一个更晚的值。如果$\mathrm{C}(X)$为真，那么前一个写$X$的已经提交，我们只要忽略$T$的写；我们允许$T$不对数据库做任何改变而继续进行下去。但是，如果$\mathrm{C}(X)$为假，那么我们必须像1(a)ii点中那样推迟$T$。
   (c) 如果$\mathrm{TS}(T) < \mathrm{RT}(X)$，那么此写是物理上不可实现的，而$T$必须被回滚。
3. 假设调度器收到提交T的请求。它必须（使用调度器维护的一个列表）找到T所写的所有数据库元素X，并置$\mathrm{C}(X) :=$ true。如果有任何等待X被提交的事务（从另一个调度器维护的列表中找到），那么这些事务被允许继续进行。
4. 假设调度器收到中止T的请求，或决定像在1b和2c中那样回滚T。那么任何等待T所写元素X的事务必须重新尝试读或写，看这一动作在取消已中止事务的写是否合法。

例18.26 图18-39给出了的三个事务T_1、T_2和T_3的一个调度，这三个事务访问三个数据库元素A、B和C。事件发生的实际时间照常随页面向下而增大。但是，我们还指明了事务的时间戳以及元素的读写时间。我们假设在开始时，每个数据库元素具有的读时间和写时间均为0。事务的时间戳是在它们通知调度器自己开始执行时获得的。请注意，尽管T_1执行第一个数据访问，但它并不具有最小的时间戳。假设T_2第一个通知调度器自己开始执行，接着是T_3，而T_1最

974

后开始。

T_1	T_2	T_3	A	B	C
200	150	175	RT=0	RT=0	RT=0
			WT=0	WT=0	WT=0
$r_1(B)$;				RT=200	
	$r_2(A)$;		RT=150		
		$r_3(C)$;			RT=175
$w_1(B)$;				WT=200	
$w_1(A)$;			WT=200		
	$w_2(C)$;				
	终止	$w_3(A)$;			

图18-39 三个事务在基于时间戳的调度器下执行

在第一个动作中，T_1读B。由于B的写时间小于T_1的时间戳，因而此读在物理上可实现并允许发生。B的读时间被置为T_1的时间戳200。第二和第三个读动作同样是合法的，导致各数据库元素的读时间被置为读它的事务的时间戳。

在第四步，T_1写B。由于B的写时间不大于T_1的时间戳，此写在物理上可实现。由于B的写时间不大于T_1的时间戳，我们必须真正执行此写。这样做时，B的写时间增加到200，即执行写操作的事务T_1的时间戳。

接下来，T_2试图写C。但是，C已经被事务T_3所读，T_3理论上的执行时间是175，而T_2将在时间150写它的值。因此，T_2试图做的事将导致物理上不可实现的行为，因而T_2必须回滚。

最后一步，T_3写A。由于A的读时间150小于T_3的时间戳175，因此写是合法的。但是，已经有一个较晚的A值存储在该数据库元素中，即由T_1理论上在时间200写入的值。因此，T_3不回滚，但它也不写入自己的值。 □

18.8.5 多版本时间戳

时间戳的一个重要变体除了维护数据库元素当前的、存储在数据库自身中的版本外，还维护数据库元素的旧版本。目的是允许其他情况下将导致事务T中止（由于T的当前版本应在T以
975 后写入）的读操作$r_T(X)$继续进行，这是通过让具有T时间戳的事务读适合它的X的版本来达到的。如果数据库元素是块或页，这种方法特别有用，因为这时所需做的只是让缓冲区管理器在主存中容纳对当前活跃的某个事务来说可能有用的某些块。

例18.27 考虑图18-40所示的访问数据库元素 A 的事务集合。这些事务在基于时间戳的普通调度器下操作，且当 T_3 试图读 A 时，它发现WT(A)比自己的时间戳大，因此必须中止。但是，A 有一个由 T_1 写入而被 T_2 覆写的旧值适合 T_3 去读；A 的这一版本的写时间为150，它小于 T_3 的时间戳175。如果 A 的这一旧值可以获得，T_3 就可以被允许去读它，尽管它不是 A 的“当前”值。 □

T_1	T_2	T_3	T_4	A
150	200	175	225	RT=0
				WT=0
$r_1(A)$				RT=150
$w_1(A)$				WT=150
	$r_2(A)$			RT=200
	$w_2(A)$			WT=200
		$r_3(A)$		
		终止	$r_4(A)$	RT=225

图18-40 T_3由于不能访问A的旧值而必须终止

多版本时间戳调度器与18.8.4节所述调度器的差别有以下几个方面：

1. 当新的写$w_T(X)$发生时，如果它合法，那么数据库元素X的一个新版本被创建。其写时间为$\mathrm{TS}(T)$，并且我们将用X_t来指代它，其中$t = \mathrm{TS}(T)$。

2. 当读$r_T(X)$发生时，调度器找到X的版本X_t，它满足$t \leqslant \mathrm{TS}(T)$，并且不存在满足$t < t' \leqslant \mathrm{TS}(T)$的版本$X_{t'}$。也就是说，恰好在$T$理论上执行前写入的$X$版本是$T$所读的版本。

3. 写时间与元素的版本相关，且永不改变。

4. 读时间也与版本相关。它们被用来拒绝某些写操作，例如时间小于原有版本读时间的写操作。图18-41表示了这一问题，其中X有版本X_{50}和X_{100}，前者在时间80被读，并发生了一个新的由具有时间戳60的事务T执行的写操作。这一写操作必然使T中止，因为如果T被允许执行，则它关于X的值应该被具有时间戳80的事务读到。

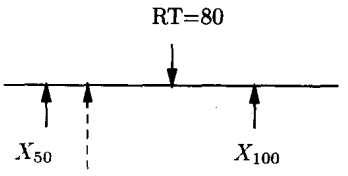


图18-41 事务试图写入使事件事实上不可实现的X版本

976

5. 当版本X_t的写时间t满足任何活跃事务的时间戳都不小于t时，我们就可以删除X的任何早于X_t的版本。

例18.28 让我们重新考虑图18-40中的动作在使用多版本时间戳时的情况。首先，A有三个版本：这些事务开始前存在的版本A_0，T_1写入的A_{150}，以及T_2写入的A_{200}。图18-42给出了事件序列、版本何时被创建以及它们何时被读取。特别注意T_3不必中止，因为它可以读A的一个较早的版本。 □

T_1	T_2	T_3	T_4	A_0	A_{150}	A_{200}
150	200	175	225			
$r_1(A)$						
$w_1(A)$				读		
	$r_2(A)$				创建	
	$w_2(A)$				读	
		$r_3(A)$				创建
			$r_4(A)$		读	读

图18-42 使用多版本并发控制的事务执行

977

18.8.6 时间戳与封锁

在大多数事务只读或并发事务极少试图读写同一元素的情况下，时间戳通常比较优越。在高冲突的情况下，封锁的性能比较好。对这一经验规律的论证为：

- 封锁在事务等待锁时会频繁地推迟事务，甚至可能导致死锁，这时多个事务等待一段很长的时间，然后其中一个不得不回滚。
- 但如果并发事务频繁读写公共元素，那么回滚就会很频繁，导致甚至比封锁系统中更多的延迟。

在几个商用系统中有一个折中方案。调度器将事务分为只读事务和读/写事务。读/写事务使用两阶段封锁执行，避免读/写事务相互间访问对方封锁的元素，以及避免只读事务访问它们封锁的元素。

只读事务使用多版本时间戳执行。当读/写事务创建数据库元素的新版本时，这些版本按18.8.5节所述进行管理。只读事务允许读适合于其时间戳的任何数据库元素版本。因此只读事务从不会被终止，只在极少时候被推迟。

18.8.7 习题

习题18.8.1 下面是几个事件序列，包括开始事件，其中st_i表示事务i开始。这些序列表示

真实时间，并且基于时间戳的调度器将按照事务开始的顺序为其分配时间戳。说明当各序列执行时将发生什么。

* a) st_1; st_2; $r_1(A)$; $r_2(B)$; $w_2(A)$; $w_1(B)$;

978 b) st_1; $r_1(A)$; st_2; $w_2(B)$; $r_2(A)$; $w_1(B)$;

c) st_1; st_2; st_3; $r_1(A)$; $r_2(B)$; $w_1(C)$; $r_3(B)$; $r_3(C)$; $w_2(B)$; $w_3(A)$;

d) st_1; st_3; st_2; $r_1(A)$; $r_2(B)$; $w_1(C)$; $r_3(B)$; $r_3(C)$; $w_2(B)$; $w_3(A)$;

习题18.8.2 说明下列事件序列过程中将发生什么，如果使用的是多版本的基于时间戳的调度器。而如果调度器不维护多个版本时，又将发生什么？

* a) st_1; st_2; st_3; st_4; $w_1(A)$; $w_2(A)$; $w_3(A)$; $r_2(A)$; $r_4(A)$;

b) st_1; st_2; st_3; st_4; $w_1(A)$; $w_3(A)$; $r_4(A)$; $r_2(A)$;

c) st_1; st_2; st_3; st_4; $w_1(A)$; $w_4(A)$; $r_3(A)$; $w_2(A)$;

!! **习题18.8.3** 在对基于锁的调度器的学习中，我们发现获得锁的事务发生死锁的原因有几个。使用提交位C(X)的基于时间戳的调度器可能有死锁吗？

18.9 使用有效性确认的并发控制

有效性确认是另一种优化的并发控制类型，其中我们允许事务不经封锁访问数据，并在适当的时候我们检查事务是否以一种可串行化的方式运转。有效性确认与时间戳的主要区别在于调度器维护关于活跃事务正在做什么的一个记录，而不是为所有数据库元素保存读时间和写时间。事务开始为数据库元素写入值前的一刹那，它经过一个“有效性确认阶段”，这时它已经读的和将写的元素集合被用来与其他活跃事务的写集合做比较。如果存在物理上不可实现的行为的风险，该事务就被回滚。

18.9.1 基于有效性确认的调度器的结构

当有效性确认被用作并发控制机制时，对每个事务T，调度器必须被告知T所读的数据库元素集合与T所写的元素集合。这些集合分别是读集合RS(T)和写集合WS(T)。事务分三个阶段来执行：

1. 读。在第一阶段，事务从数据库中读其读集合中的所有元素。事务还在其局部地址空间中计算它将要写的所有值。

2. 有效性确认。在第二阶段，调度器通过比较该事务与其他事务的读写集合来确认该事务的有效性。我们将在18.9.2节描述有效性确认过程。如果有效性确认失败，则事务回滚；否则它继续进入第三阶段。

3. 写。在第三阶段，事务往数据库中写入其写集合中元素的值。

直观地说，我们可以认为每个成功确认的事务是在其有效性确认的瞬间执行的。因此，基于有效性确认的调度器对事务的进行有一个假定的串行顺序，并且它根据事务行为是否与这一串行顺序一致来决定确认事务是否有效。

979 为了支持做出是否确认事务有效性的决定，调度器维护三个集合：

1. START，已经开始但尚未完成有效性确认的事务集合。对这个集合中的每个事务T，调度器维护START(T)，即事务T开始的时间。

2. VAL，已经确认有效性但尚未完成第3阶段写的事务。对这个集合中的每个事务T，调度器维护START(T)和VAL(T)，即T确认的时间。请注意，VAL(T)也是在假设的串行执行顺序中所设想的T的执行时间。

3. FIN，已经完成第3阶段的事务。对这样的事务T，调度器记录START(T)、VAL(T)和FIN(T)，即T完成的时间。原则上这个集合将增长，但正如我们将看到的，如果对任意活跃事务U（即对任何在*START*或*VAL*中的U），事务T满足FIN(T)<START(U)，这样的T我们不必记住。调度器因此可以周期性地清理*FIN*集合，以防止其增大到超过限度。

18.9.2 有效性确认规则

如果由调度器维护18.9.1节中的信息，这些信息足以使它能够监测出任何违反假设的事务串行顺序的潜在可能，假设的事务串行顺序即事务有效性确认的顺序。为了理解这些规则，让我们首先考虑当我们想要确认一个事务有效性时可能发生什么错误。

1. 假设存在事务U满足：

(a) U在*VAL*或*FIN*中；即U已经过有效性确认。

(b) FIN(U) > START(T)；即U在T开始前没有完成[⊖]。 980

(c) RS(T)∩WS(U)非空；特别地，设其包含数据库元素X。

那么U有可能在T读X后写X。事实上，U甚至可能还没有写X。U写了X但并不及时的一种情况如图18-43所示。为了解释这个图，请注意虚线将实时的事件与事务在其有效性确认瞬间执行时这些事件发生的时间联系起来。由于我们不知道T是否读到U的值，我们必须回滚T以避免T和U的动作与假设的串行顺序不一致的风险。

2. 假设存在事务U满足：

(a) U在*VAL*中；即U的有效性已经成功确认。

(b) FIN(U) > VAL(T)；即在T进入其有效性确认阶段以前没有完成。

(c) WS(T)∩WS(U)≠Φ；特别地，设X同时在两个写集合中。

这时潜在的问题如图18-44所示。T和U都必须写X的值，而如果我们确认T的有效性，它就可能在U前写X。由于我们不能确定，因此回滚T以保证它不会违反假设的T在U后的串行顺序。

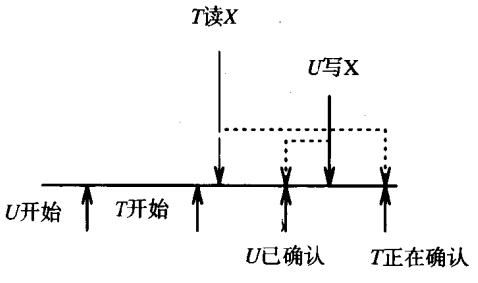


图18-43 如果一个较早的事务现在正在写入T应该读过的某些东西，则T的有效性不能确认

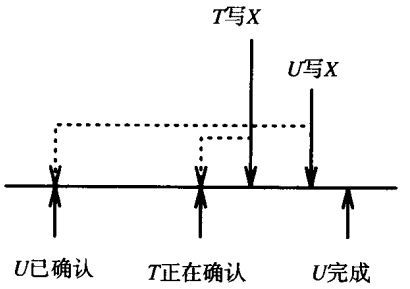


图18-44 如果T在有效性确认后并在一个较早的事务之前先写某些东西，则T不能确认

上面描述的两个问题是T的写可能在物理上不可实现的仅有的情形。在图18-43中，如果U在T开始前完成，那么T肯定应该读到U或某个更晚的事务所写的X值。在图18-44中，如果U在

⊖ 请注意，如果U在*VAL*中，那么当T确认时U尚未完成。在这种情况下，FIN(U)在技术上是未定义的。但是，我们知道这种情况下它必然大于START(T)。

T进行有效性确认前完成，那么U肯定在T前写X。因此我们可以用下面关于事务T进行有效性确认的规则来概括这些发现。

- 对于所有已经过有效性确认且在T开始前没有完成的U，即对于满足FIN(U) > START(T)
981 的U，比较RS(T)和WS(U)以检测是否满足RS(T)∩WS(U) = Φ。
- 对于所有已经过有效性确认且在T经过有效性确认前没有完成的U，即对于满足FIN(U) > VAL(T)的U，比较WS(T)和WS(U)以检测是否满足WS(T)∩WS(U)=Φ。

例18.29 图18-45给出了四个事务T、U、V和W试图执行并确认有效性的时间线。每个事务的读写集合都在图中标明。T先开始，尽管U最先进行有效性确认。

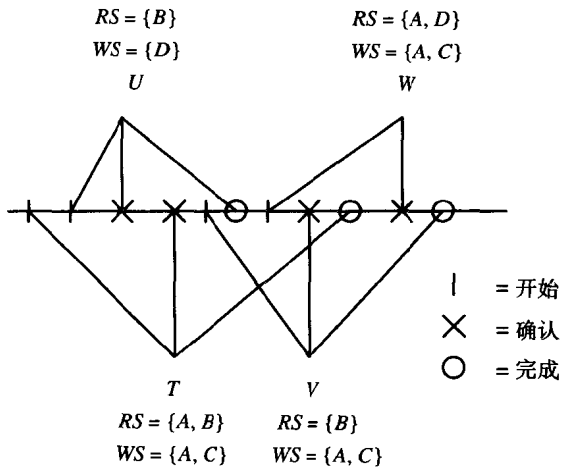


图18-45 四个事务及其有效性确认

1. U的有效性确认：当确认U的有效性时没有其他已确认事务，因而什么也不需要检测。U成功确认其有效性并为数据库元素D写入一个值。

2. T的有效性确认：当确认T的有效性时，U已确认但尚未完成。因此，我们必须检测T的读写集合是否满足与WS(U) = {D}没有公共元素。由于RS(T) = {A，B}，而WS(T) = {A，C}，两个检测都成功，因而确认T的有效性。

3. V的有效性确认：当确认V的有效性时，U已确认和完成，T已确认但尚未完成。并且V在U完成前开始。因此，我们必须用RS(V)与WS(V)来和WS(T)比较，但只需用RS(V)来和WS(U)比较。我们发现：

- RS(V) ∩ WS(T) = {B} ∩ {A,C} = Φ
- WS(V) ∩ WS(T) = {D,E} ∩ {A,C} = Φ
- RS(V) ∩ WS(U) = {B} ∩ {D} = Φ

982 因此，V的有效性也成功确认。

瞬时行为

你可能已经注意到确认这一动作在瞬间或不可分的时间内发生这一概念。例如，想像一下我们能够确定在我们开始确认事务T前事务U是否已经确认。U可能在我们正在确认T时完成确认吗？

如果在单处理器的系统上运行，并且具有一个调度器进程，那么我们确实可以将确

认或调度器的其他动作看做是在瞬间发生的。原因在于，如果调度器正在确认T，它就不可能也正在确认U，所以在T确认的整个过程中，U的确认状态不会改变。

如果我们运行在多处理器上，并且有多个调度器进程，那么就有可能一个在确认T，而另一个在确认U。如果这样，那么我们需要依赖多处理器系统所提供的同步机制来使确认成为一个原子的动作。

4. W的有效性确认：当确认W的有效性时，我们发现U在W开始前已完成，因此在W与U之间不执行任何比较。T在W确认前完成但未在W开始前完成，因此我们只比较RS(W)和$WS(T)$。V已确认但尚未完成，因此我们需要用RS(W)与WS(W)来和WS(T)比较。检测如下：

- $\text{RS}(W) \cap \text{WS}(T) = \{A, D\} \cap \{A, C\} = \{A\}$
- $\text{RS}(W) \cap \text{WS}(V) = \{A, D\} \cap \{D, E\} = \{D\}$
- $\text{WS}(W) \cap \text{WS}(V) = \{A, C\} \cap \{D, E\} = \Phi$

由于交集并非都为空，因此W的有效性不能确认。相反，W被回滚且不为A或C写入值。 □

18.9.3 三种并发控制机制的比较

我们已经考虑过的三种可串行性方法（封锁、时间戳和有效性确认）各有其优点。首先，可以在存储的利用率上进行比较：

- 封锁：锁表空间与被封锁元素个数成正比。 983
- 时间戳：在不成熟的实现中，每个数据库元素的读时间和写时间都需要空间，不管该元素当前是否被访问。但是，更精细的实现会将最早的活跃事务以前的所有时间戳看做“负无穷”，并且不记录它们。在这种情况下，我们可以类似锁表那样将读时间和写时间记录在一张表中，其中只给出那些最近已经被访问过的数据库元素。
- 有效性确认：对于每个当前活跃的事务以及少量几个在某当前活跃事务开始后完成的事务，空间用于时间戳和读/写集合。

因此，每种方法使用的空间数量大致正比于所有活跃事务访问的数据库元素之和。时间戳和有效性确认可能使用空间略微多一些，因为它们记录最近提交事务的某些访问，而这是锁表所不记录的。有效性确认的一个潜在问题是，事务的写集合必须在写发生以前（但在事务的局部计算已经完成后）知道 。

我们还可以在不推迟事务完成的能力方面比较这些方法的成效。这三种方法的性能依赖于事务间的相互影响（事务访问一个并发事务所访问元素的可能性）是高还是低。

- 封锁推迟事务但避免回滚，即使当相互影响较高时。时间戳和有效性确认不推迟事务，但会导致其回滚，而这是推迟的一种更严重的形式，并且也浪费资源。
- 如果相互影响较低，那么时间戳和有效性确认都不会导致太多的回滚，并且因为它们通常比封锁调度器开销小而更受欢迎。
- 当回滚是必要的时，时间戳比有效性确认更早地捕获某些问题，或者在考虑一个事务是否必须回滚前常常让其做完所有的内部工作。

18.9.4 习题

习题18.9.1 在下列时间序列中，我们用$Ri(X)$表示“事务T_i开始，且其读集合是数据库元素列表X”。此外，V_i表示“T_i试图确认有效性”，而$W_i(X)$表示“事务T_i完成，且其写集合是X”。说明当一个基于有效性确认的调度器处理每个序列时会发生什么。

984

* a) $R_1(A,B)$; $R_2(B,C)$; V_1; $R_3(C,D)$; V_3; $W_1(A)$; V_2; $W_2(A)$; $W_3(B)$;

b) $R_1(A,B)$; $R_2(B,C)$; V_1; $R_3(C,D)$; V_3; $W_1(A)$; V_2; $W_2(A)$; $W_3(D)$;

c) $R_1(A,B)$; $R_2(B,C)$; V_1; $R_3(C,D)$; V_3; $W_1(C)$; V_2; $W_2(A)$; $W_3(D)$;

d) $R_1(A,B)$; $R_2(B,C)$; $R_3(C)$; V_1; V_2; V_3; $W_1(A)$; $W_2(B)$; $W_3(C)$;

e) $R_1(A,B)$; $R_2(B,C)$; $R_3(C)$; V_1; V_2; V_3; $W_1(C)$; $W_2(B)$; $W_3(A)$;

f) $R_1(A,B)$; $R_2(B,C)$; $R_3(C)$; V_1; V_2; V_3; $W_1(A)$; $W_2(C)$; $W_3(B)$;

18.10 小结

- 一致的数据库状态：遵循设计者想要的所有隐含的或声明的约束的数据库状态被称为是一致的。数据库上的操作保持一致性是必要的，即将一个一致的数据库状态转换到另一个。
- 并发事务一致性：多个事务同时访问一个数据库是很正常的。隔离开来执行的事务是假定能保持数据库一致性的。保证并发操作的事务也保持数据库的一致性是调度器的任务。
- 调度：事务被划分为动作，主要是对数据库的读和写。来自一个或多个事务的这些动作的一个序列称为一个调度。
- 串行调度：如果一次执行一个事务，则该调度被称为是串行的。
- 可串行化调度：某个调度在对数据库的效果上等价于某个串行调度，则被称为是可串行化的。在一个可串行化的但非串行的调度中，来自几个事务的动作相互交错是可能的，但是对于我们允许的那些动作序列必须要小心，否则一个交错将使数据库处于不一致的状态。
- 冲突可串行性：可串行性的一个易于判断的充分条件是调度能通过一系列相邻动作的非冲突交换转变为串行的。这样的调度称为冲突可串行化调度。如果我们试图交换同一事务的两个动作，或者交换访问同一数据库元素的两个动作而其中至少一个动作是写，那么冲突将发生。
- 优先图：冲突可串行性的一种简单的测试方法是为调度构造一个优先图。结点对应于事务，而如果某个动作T在调度中与一个比较靠后的动作U冲突，那么就有一条弧$T \rightarrow U$。调

985

度是冲突可串行化的，当且仅当优先图无环。
- 封锁：保证可串行化调度最常用的方法是在访问数据库元素前先封锁，并在完成对该元素的访问后释放其锁。元素上的锁禁止其他事务访问该元素。
- 两阶段封锁：封锁自身并不能保证可串行性。但是，两阶段封锁能保证可串行性，在两阶段封锁中所有事务首先进入一个只申请锁的阶段，然后进入一个只释放锁的阶段。
- 封锁模式：为了避免不必要地将事务封锁在外，系统通常使用多种封锁模式，对每种模式什么时候可以授予锁有不同的规则。最常用的系统中包含用于只读访问的共享锁和用于包括写的访问的排他锁。
- 相容性矩阵：相容性矩阵是对已知同一元素上可能有另一模式或同一模式的锁情况下，授予某种模式的锁什么时候合法的一种有用的汇总。
- 更新锁：调度器允许一个先读再写一个元素的事务首先获得一个更新锁，而在以后将该锁升级为排他锁。更新锁在元素上已经有共享锁时可以被授予，但一旦元素上有了一个更新锁，它就禁止在该元素上授予其他锁。

- *增量锁*：对于事务只想在元素上增加或减少一个常数这一常见的情况，增量锁非常适合。同一元素上的增量锁相互不冲突，尽管它们同共享锁以及排他锁冲突。
- *有粒度层次元素的封锁*：当大的元素和小的元素（或许是关系、磁盘块和元组）都可能需要被封锁时，警示封锁系统保证可串行性。事务在大的元素上加意向锁，以警示其他事务自己打算访问该元素的一个或多个子元素。
- *组织成树的元素的封锁*：如果数据库元素只在沿着树向下时被访问，如同B树中那样，那么一个非两阶段封锁策略可以用来保证可串行性。其规则要求在子结点上上锁时需要持有父结点上的锁，尽管父结点上的锁可以随后释放，而且以后可以获得其他的锁。
- *优化并发控制*：调度器可以不使用封锁，而假设事务是可串行化的，并在看到某个潜在的非可串行化行为时将事务中止。这种被称为是优化的方法分为基于时间戳的调度和基于有效性确认的调度。 986
- *基于时间戳的调度器*：这类调度器在事务开始时为其赋予时间戳。数据库元素有相关的读写时间，它们是最近执行这些操作的事务的时间戳。如果一个不可能的情况被检测到，例如一个事务读到在该事务的将来写入的一个值，那么违例的事务被回滚，即中止并重启。
- *基于有效性确认的调度器*：这些调度器在事务已经读完所有它们所需要的东西以后，但在它们写以前确认事务有效性。事务如果已经读或将要写另外的某个事务正在写的元素，那么将产生有歧义的结果，所以该事务的有效性不能确认。确认失败的事务被回滚。
- *多版本时间戳*：实践中用于只读事务的一种常用技术是使用时间戳，但有多个版本，其中对一个元素的写不覆盖先前为该元素写入的值，直到所有可能需要先前的值的事务已经完成。写事务通过传统的封锁来调度。

18.11 参考文献

[6]是有关调度和封锁的重要资料来源。[3]是另一个重要的来源。[12]和[11]是近期关于并发控制的两篇综述。

关于两阶段封锁的[4]可能是事务处理领域中意义最大的文章。粒度层次的警示协议来自[5]。树的非两阶段封锁来自[10]。为了研究封锁模式的行为，[7]中引入了相容性矩阵。

时间戳作为一种并发控制方法出现在[2]和[1]中。使用有效性确认的调度来自[8]。[9]研究了多版本的使用。

1. P. A. Bernstein and N. Goodman, "Timestamp-based algorithms for concurrency control in distributed database systems," *Proc. Intl. Conf. on Very Large Databases* (1980), pp. 285–300.

2. P. A. Bernstein, N. Goodman, J. B. Rothnie, Jr., and C. H. Papadimitriou, "Analysis of serializability in SDD-1: a system of distributed databases (the fully redundant case)," *IEEE Trans. on Software Engineering* **SE-4**:3 (1978), pp. 154–168.

3. P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading MA, 1987.

4. K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Comm. ACM* **19**:11 (1976), pp. 624–633.

987

5. J. N. Gray, F. Putzolo, and I. L. Traiger, "Granularity of locks and degrees of consistency in a shared data base," in G. M. Nijssen (ed.), *Modeling in Data Base Management Systems*, North Holland, Amsterdam, 1976.

6. J. N. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan-Kaufmann, San Francisco, 1993.

7. H. F. Korth, "Locking primitives in a database system," *J. ACM* **30**:1 (1983), pp. 55–79.

8. H.-T. Kung and J. T. Robinson, "Optimistic concurrency control," *ACM Trans. on Database Systems* **6**:2 (1981), pp. 312–326.

9. C. H. Papadimitriou and P. C. Kanellakis, "On concurrency control by multiple versions," *ACM Trans. on Database Systems* **9**:1 (1984), pp. 89–99.

10. A. Silberschatz and Z. Kedem, "Consistency in hierarchical database systems," *J. ACM* **27**:1 (1980), pp. 72–80.

11. A. Thomasian, "Concurrency control: methods, performance, and analysis," *Computing Surveys* **30**:1 (1998), pp. 70–119.

12. B. Thuraisingham and H.-P. Ko, "Concurrency control in trusted database management systems: a survey," *SIGMOD Record* **22**:4 (1993), pp. 52–60.

988

第19章 再论事务管理

本章我们讨论第17章和第18章所没有谈到的几个关于事务管理的问题。首先我们要协调一下前两章中的观点：错误恢复、允许事务中止以及维护可串行性这三种需要如何相互影响？接着我们讨论事务间的死锁管理，死锁通常由几个事务引起，这些事务各自需要等待一个被另一事务占用的资源，例如锁。

本章中还包括了对分布式数据库的介绍。我们主要讨论如何封锁分布在几个节点中的元素，这些元素可能有多个副本。我们还要考虑当事务自身涉及几个节点上的动作时，事务提交或中止的决定怎样做出。

最后我们考虑由“长事务”引起的问题。某些应用中人和计算机进程需要进行交互，例如CAD系统和“工作流”系统，这样的交互可能长达数天。这些系统和短事务系统（如银行系统和机票预订系统）一样需要保持数据库状态的一致性。但是，第18章讨论的并发控制机制不能很好地工作，因为锁需要被占用数天，或者有效性确认需要基于几天以前发生的事件。

19.1 读未提交数据的事务

在第17章中我们讨论了日志的创建以及在系统崩溃时如何使用日志恢复数据库状态。我们引入了数据库计算的视图，即值在非易失性磁盘、易失性主存以及事务的局部地址空间之间移动。各种日志方式所给的保证是，当崩溃发生时，它能在数据库的磁盘拷贝中重建提交事务 989
（而且仅有提交事务）的动作。日志系统不试图支持可串行性；它盲目地重建数据库状态，即使该状态是由动作的非可串行化调度产生的。事实上，商用数据库系统不一定总是坚持可串行性，在有的系统中，仅在用户显式地要求时才实现可串行性。

另一方面，第18章只讨论可串行性。根据那一章的原则设计的调度器可能做出日志管理器所不能容忍的事。例如，可串行性定义不禁止封锁元素A的事务在提交前为A写入新值，因而违反日志策略的规则。更糟糕的是，即使没有发生系统崩溃，并且调度器在理论上维护可串行性，事务也可能在写数据库后中止，而这很容易导致数据库状态不一致。

19.1.1 脏数据问题

回忆一下8.6.5的内容，如果数据被未提交事务写入，那么该数据就是“脏的”。脏数据可能出现在缓冲区中、磁盘上或兼而有之；不管哪一种都可能导致问题。

例19.1 我们重新考虑图18-13中的可串行化调度，但假设在T_1读B后由于某个原因不得不中止。事件系列如图19-1所示。在T_1中止后，调度器释放T_1所获得的对B的锁；这一步骤很关 990
键，否则别的事务就永远也不能获得B上的锁了。

但是，T_2现在已读入了不代表数据库的一致状态的数据。也就是说，T_2读到的A值是T_1修改后的值，而它读到的B值是T_1采取动作以前的值。在这种情况下，T_1是否将A值125写到磁盘无关紧要；不管怎样，T_2都从缓冲区中得到该值。T_2读到不一致状态的后果是，它使（磁盘上的）数据库处于不一致的状态，其中$A \neq B$。

图19-1中的问题在于，T_1写入的A是脏数据，不管它在主存中还是在硬盘上。T_2读A，然后又在自己的计算中使用该值，这就使T_2的动作变得不可靠。正如我们将在19.1.2节看到的那样，

如果允许这样的情况发生，就有必要中止和回滚T_1与T_2。此外，如果允许T_1的A值到达磁盘，修复的代价就很高；我们必须用日志撤销相应的更改。因此，我们需要提出一些防止脏数据到达磁盘的规则。 □

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$;			
`A := A+100;`			
$w_1(A)$; $l_1(B)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$;		
	`A := A*2;`		
	$w_2(A)$;	250	
	$l_2(B)$ 被拒绝		
$r_1(B)$;			
中止；$u_1(B)$;			
	$l_2(B)$; $u_2(A)$; $r_2(B)$;		
	`B := B*2;`		
	$w_2(B)$; $u_2(B)$;		50

图19-1 T_1写入脏数据后中止

T_1	T_2	T_3	A	B	C
200	150	175	RT=0	RT=0	RT=0
			WT=0	WT=0	WT=0
	$w_2(B)$;			WT=150	
$r_1(B)$;					
	$r_2(A)$;		RT=150		
		$r_3(C)$;			RT=175
	$w_2(C)$;				
				WT=0	
	中止	$w_3(A)$;	WT=175		

图19-2 T_1从T_2读到脏数据，因而当T_2中止时T_1也必须终止

例19.2 现在考虑图19-2，它给出了18.8节中基于时间戳的调度器下的一个动作序列。但是，我们假设这一调度器不使用18.8.1节中所引入的提交位。回忆一下，提交位的目的是防止未提交事务写入的值被其他事务读取。因此，当T_1在第二步读取B时，没有提交位来告诉T_1它需要等待。T_1可以继续执行，甚至可以写磁盘并提交；我们没有进一步给出T_1的详细步骤。

最后，T_2试图以一种物理上不可实现的方式写C，T_2因而中止。T_2以前写B所产生的效果被撤销；B的值和写时间分别重置为T_2写以前的值和写时间。然而，T_1已被允许使用这一撤销的B值，并能用该值做任何事情，例如用它来计算A、B和/或C的新值，并将它们写回磁盘。因此，
991 T_1在读B的脏值后可能会导致不一致的数据库状态。请注意，如果记录并使用提交位，那么第2步的$r_1(B)$将被推迟，必须等到T_2中止，并且B的值已恢复为原（假设已提交的）值后，这一动作才允许发生。 □

19.1.2 级联回滚

正如我们在上面的例子中看到的那样，如果事务可以获得脏数据，那么有时候我们需要执行*级联回滚*。也就是说，当事务T中止时，我们必须确定哪些事务读了由T写入的数据，中止这些事务，然后递归地中止读了由被中止事务所写的数据的所有事务。换言之，我们必须找到每个读了T所写的脏数据的事务U，中止U，再找到每个读了U所写的脏数据的事务V，中止V，依此类推。如果日志是提供改前值的某类日志（undo日志或undo/redo日志），那么我们可以利用日志来撤销中止事务的影响。如果脏数据的影响还没有到达磁盘，我们也可以使用数据库的硬

盘拷贝来恢复数据。这些方式将在下一节中讨论。

正如我们已经提到的那样，使用提交位的基于时间戳的调度器禁止可能已读到脏数据的事务继续执行，因而使用这样的调度器时不可能产生级联回滚。基于有效性确认的调度器也能避免级联回滚，因为写数据库（甚至缓冲区）只有在确定事务将提交后才发生。

19.1.3 可恢复调度

为了使第17章中所讲的任何一种记录日志的方法都能用于恢复，那些被认为是已提交的事务集合在恢复后必须一致。也就是，若T_1在恢复后仍是已提交的事务，而且T_1使用了T_2所写的值，则T_2在恢复后必须也是提交的事务。因此，可给出如下定义：

- 若每个事务只在改变其所使用数据的事务提交后才提交，则称此调度是可恢复的（recoverable）。

例19.3 下面将给出相同操作的多种调度，在这些调度中，均含有读和写操作，其中C_i表示“事务T_i提交”的操作。可恢复调度：

$$S_1\text{: } w_1(A);\ w_1(B);\ w_2(A);\ r_1(B);\ c_1;\ c_2;$$

注意到 T_2 读取了 T_1 写的一个值（B），所以为了使该调度是可恢复的，T_2 必须在 T_1 提交后提交。

调度S_1显然是串行的（因此是可串的），并且是可恢复的。但是这两个概念是矛盾的。例如，下面给出对S_1改动后的结果S_2，这个调度仍是可恢复的，但是不可串行的。

$$S_2\text{: } w_2(A);\ w_1(B);\ w_1(A);\ r_2(B);\ c_1;\ c_2;$$

992

在S_2中，由于对A的写，所以串行序中T_2必须在T_1前，然而由于对B的读写又使得串行序中T_1必须在T_2之前。

最后，观察下面对S_1进行变动后的另一个调度，它是可串行的但不是可恢复的：

$$S_3\text{: } w_1(A);\ w_1(B);\ w_2(A);\ r_2(B);\ c_2;\ c_1;$$

在S_3中，T_1在T_2之前，但它们的提交操作顺序相反了。若在故障发生前，T_2的提交记录已写回磁盘，但T_1的提交记录仍未写回磁盘，则不管使用哪种记录日志法（撤销、重做或撤销/重做法），T_2将在恢复后提交，而T_1就不会。 □

为了使可恢复调度能根据任一种记录日志法都能真实地被恢复，必须给出对调度的一个附加的假设：

- 日志提交记录写回磁盘的顺序必须和它们写顺序一致。

考虑例19.3中的调度S_3，它的提交记录以错误的顺序写回磁盘，因此不可能会出现一致性恢复。19.1.6节中将进一步讨论这个规则。

19.1.4 避免级联回滚的调度

可恢复调度有时需要级联回滚。如例19.3中的调度S_1，若它的前四步操作完成后T_1不得不回滚，则T_2也要回滚。因此需要一个比可恢复更强的条件来消除级联回滚。这就是：

- 若调度中的事务仅仅读取已提交事务所写的值，则该调度避免了级联回滚（或称做“是*ACR*调度”）。

用另一种说法则是，ACR调度禁止读取脏数据。对于可恢复调度，假设“提交”意味着日志的提交记录已被写回磁盘。

例19.4 例19.3中的调度都不是ACR。在每种调度中，T_2均从未提交事务T_1中读取B。然而，考虑调度：

$$S_4\text{: } w_1(A);\ w_1(B);\ w_2(A);\ c_1;\ r_2(B);\ c_2;$$

此时，T_2在T_1（最后一个写B的事务）提交后才读取B，所以T_1已提交，并且它的日志已写回磁盘。因此调度S_4是ACR调度； □

注意，像T_2这样读取已提交事务T_1所写的值的事务，在T_1提交后T_2一定有显式提交或是夭折的结果。因此：

993 • 每个ACR调度是可恢复调度。

19.1.5 回滚的管理

将早先的讨论应用于任一种调度策略产生的调度。在基于锁的通常调度中，有一种简单的方法可以保证不产生级联回滚：

• *严格锁*（Strict Locking）：事务不释放任一排它锁（或其他锁，如增量锁，这种锁允许值的改变），除非这个事务已经提交或中止，而且它的提交或中止日志记录已经被写回磁盘。

满足严格锁规则的事务调度被称为严格调度（strict schedule）。这种调度的两种重要特性是：

• *每个严格调度是ACR调度*。原因是，事务T_2在T_1释放任一排他锁（或允许X被改变的相似锁）之前，不能读取由T_1所写的数据X。

• *每个严格调度是可串行的*。原因是严格调度等价于串行调度，该调度中的每个事务都在提交时瞬时运行。

根据上述两点观察，可以画出到目前为止所提到的不同调度间的关系，见图19-3。

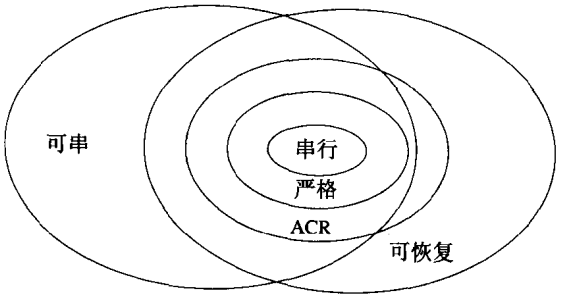


图19-3 调度类之间的包含与不包含

很明显，在严格调度中，因为未提交事务在把数据写入缓冲器时仍持有锁，而到提交时才释放锁，所以事务不可能会读脏数据。然而，存在的问题是：当事务中止时，因为要撤销这些改变带来的影响，所以要把数据固定在缓冲器中。把数据固定在缓冲器中的难度依赖于数据库
994 的元素是块还是更小的单元。下面将依次对它们进行讨论。

块的回滚

如果可封锁数据库元素是块，那么我们有一种不需要使用日志的回滚方法。假设事务T获得块A上的排他锁，在缓冲区中为A写入新值，然后不得不中止。由于A在T写入其值以来一直是被锁住的，其他事务都不会读A。如果遵循下面的规则，则恢复A的旧值就很简单：

• 未提交事务所写的块被钉在主存中；即不允许它们的缓冲区写到磁盘。

在这种情况下，我们通过让缓冲区管理器忽略A值来“回滚”T。也就是说，A占据的缓冲区不写到任何地方去，且该缓冲区被加入可用缓冲区池中。我们可以确定磁盘上的A值是已提交事务最近写入的值，这正是我们希望A具有的值。

如果使用18.8.5节和18.8.6节的多版本系统，那么我们还有另一种简单的回滚方法。我们必

须再次假设未提交事务所写的块被固定在主存中。那么，我们只要从A的可用值列表中去掉T所写入的值即可。注意，由于T是写事务，其A值从写入该值那一刻起到T中止这一段时间内是被锁住的（假设使用18.8.6节的时间戳/封锁方式）。

小的数据库元素的回滚

如果可封锁数据库元素是块的部分（例如元组或对象），那么恢复被已中止事务修改过的缓冲区的简单方法就行不通了。问题在于，一个缓冲区中可能包含两个甚至更多事务修改过的数据；如果其中之一中止，我们仍然必须保留其他事务所做的修改。当需要恢复被已中止事务修改过的一个小的数据库元素A的旧值时，我们可以有几种选择：

1. 我们可以从存储在磁盘上的数据库中读取A原来的值，并对缓冲区内容做适当的修改。

2. 如果日志是undo日志或undo/redo日志，那么我们可以从日志中获得改前值。用于故障恢复的代码同样可用于“自动”回滚。

3. 我们可以为每个事务所做的修改维护一个单独的主存日志，该日志仅在对应事务活跃时保留。旧值可以从这一“日志”中获得。

这些方法都不理想。第一种方法显然需要一次磁盘访问。第二种（查看日志）方法在相关 995
的日志部分仍在缓冲区中时可能不需要访问磁盘，但是，这种方法也可能需要查看磁盘上的大量日志，以找到给出正确改前值的更新记录。最后一种方法不需要访问磁盘，但主存“日志”可能消耗很大一部分主存。

19.1.6 成组提交

在某些情况下，即使不立即将日志中的提交记录刷新到磁盘，我们也可以避免读脏数据。只要按写日志记录的顺序刷新它们，我们就可以在提交记录写到位于缓冲区的日志中以后立即释放锁。

例19.5 假设事务T_1写X，完成，并将其COMMIT记录写入日志，但日志记录仍保留在缓

什么时候事务真正提交？

成组提交的微妙之处提醒我们，已经完成的事务在它完成工作到它真正“提交”之间可以有几种不同的状态，所谓真正提交是指在任何情况下（包括系统故障发生时）事务所产生的影响都不会丢失。正如我们在第17章提到的那样，事务可能已经完成工作，甚至已将COMMIT记录写到位于主存缓冲区的日志中，但当系统发生故障并且COMMIT记录尚未到达磁盘时，该事务所产生的影响仍可能丢失。此外，我们在17.5节中看到，即使COMMIT记录已在磁盘上，但如果还没有转储到备份中，那么介质故障仍可能导致事务被撤销且事务所产生的影响丢失。

在没有故障发生时，各事务必然都会从完成工作开始进一步推进，直到使自己产生的影响甚至在介质故障后也能得到保留，从这一意义上来说，所有的这些状态都是等价的。然而，当我们需要考虑故障和恢复时，识别这些在无故障情况下都可以被非正规地认为“已提交”的状态之间的差别是很重要的。

冲区中。虽然T_1的提交记录能否在崩溃后依然存在尚未知晓，且从这种意义上来说T_1还没有提交，但我们仍将释放T_1的锁。接下来，T_2读X并“提交”，但它的提交记录也仍然保留在缓冲区中，并位于T_1的提交记录之后。由于我们按写日志记录的顺序将它们刷新到磁盘，恢复管理器只有在认为T_1也已经提交（因为其提交记录已到达磁盘）时才会认为T_2已经提交。因此，恢复管理器可能遇到三种情况：

996 1. T_1和T_2的提交记录都还没有到达磁盘。那么，这两个事务都被恢复管理器中止，而T_2从未提交事务T_1读X这件事也是无关紧要的。

2. T_1提交而T_2未提交。这没什么问题，因为：T_2没有从未提交事务读X，而且不管怎样它中止了，因而它不会对数据库产生任何影响。

3. 两个事务都提交。那么T_2读X不是脏的。

另一方面，假设包含T_2提交记录的缓冲区已刷新到磁盘（例如，由于缓冲区管理器决定将该缓冲区用于别的目的），但包含T_1提交记录的缓冲区尚未刷新到磁盘。如果在这时发生崩溃，那么在恢复管理器看来T_1未提交而T_2已提交。T_2所产生的影响将在数据库中得到永久的反映，但这一影响建立在T_2对X的脏读上。 □

从例19.5可以得到这样的结论：我们可以在事务提交记录刷新到磁盘前释放锁。这一策略通常称为*成组提交*，它指的是：

- 在事务完成且提交日志记录至少出现在缓冲区中以前不能释放锁。
- 日志记录按创建的顺序刷新。

与19.1.3节所讨论的要求“可恢复调度”的策略一样，成组提交保证从不读脏数据。

19.1.7 逻辑日志

我们在19.1.5节中看到，如果封锁的单位是块或页，那么脏读的修复比较容易。但是，当数据库元素是块时至少有两个问题。

1. 所有日志方式都要求在日志中记录数据库元素的新值、旧值或二者都记录。如果块中变化较小，例如改写一个元组的某个属性或插入、删除一个元组，那么将有大量的冗余信息写入日志中。

2. 要求调度可恢复，即只有提交后才能释放锁，这可能严重抑制并发性。例如，回忆一下18.7.1节中对使用B树访问数据时提前释放锁的好处所作的讨论。如果我们要求事务占有锁直到提交，就得不到这一好处，并且实际上在任何时刻我们都只允许一个写事务访问B树。

这些因素推动了*逻辑日志*的使用。逻辑日志中只描述块中的变化。根据所发生变化的性质，
997 复杂程度也有所不同。

1. 数据库元素的少量字节改变，例如更新一个定长字段。这一情况可以用一种直截了当的方式来处理，即我们只记录改变的字节及其位置。例19.6将说明这一情况以及合适的更新记录形式。

2. 数据库元素的改变描述简单，易于恢复，但它产生的影响是改变了该数据库元素的大多数或全部字节。一种常见的情况是，记录的一个变长字段改变，该记录的大部分甚至其他记录需要在块内滑动。这种情况在例19.7中讨论。块的新值和旧值看来差别很大，除非我们发现并指明导致变化的简单原因。

3. 变化影响到数据库元素的许多字节，且进一步的改变可能使这一变化变得不可撤销。这种情况是真正的“逻辑”日志，因为我们不能将undo/redo过程视为发生在数据库元素上，而应视为发生在数据库元素所代表的高层“逻辑”结构上。在例19.8中，我们将以B树为例来说明逻辑日志的这一复杂形式，B树就是一种用磁盘块这样的数据库元素表示的逻辑结构。

例19.6 假设数据库元素是块，每块中包含某个关系的一个元组集合。我们可以用一个日志记录来表述一个属性的更新，这一日志记录说明“元组t的属性a的值从v_1变到v_2”。在块的空闲空间中插入一个新元组可以表述为“具有值（a_1，a_2，…，a_k）的元组被插入，其起始偏移量为p”。除非改变的属性或插入的元组与块大小相当，否则这些记录占据的空间将远远小于整

个块。此外，它们既能服务于undo操作，又能服务于redo操作。

注意，这些操作都是幂等的；在块上将这样的一个操作执行多次，其效果等同于执行一次。类似地，它们所隐含的逆操作，即将$t[a]$的值从v_2恢复成v_1以及删除元组t，也是幂等的。因此，这种类型的记录在恢复中就可以像整个第17章中的更新日志记录那样使用。 □

例19.7 再次假设数据库元素是存储元组的块，但元组中存在变长字段。如果像例19.6所描述的那样的改变发生，我们可能需要移动块中的大部分东西，以给变长后的字段腾出空间或在字段变短时维护空间。在极端的情况下，可能还必须创建溢出块（回忆一下12.5节）以容纳原块中的部分内容，或由于字段变短而使我们可以将两块的内容合为一块时，我们还可以删除溢出块。 998

只要块及其溢出块被看做是一个数据库元素，使用被改变字段的旧值和/或新值来撤销或重建修改就比较容易。但是，块及其溢出块必须看做是在一个“逻辑”的层次上容纳了某些元组。在undo和redo后，我们甚至不能将这些块中的字节恢复到原有状态，因为可能由于其他字段长度改变而导致块的重组。然而，如果我们认为表示某些元组的块的集合是数据库元素，那么redo和undo可以真正地恢复元素的逻辑“状态”。 □

但是，正如我们在例19.7中提到的那样，通过溢出块机制将块当作可扩展的来看待有时是不可能的。这样，我们只能在一个比块高的层次上执行undo和redo动作。下一个例子讨论B树索引中块的管理不允许溢出块这一重要情况，这时我们必须认为undo和redo发生在B树自身这一“逻辑”层次上而不是发生在块上。

例19.8 我们考虑为B树结点记录逻辑日志的问题。我们并不将整个结点（块）的新值和/或旧值写入日志中，而是写入一个描述变化的简短记录。这样的变化包括：

1. 插入和删除子结点的一个键-指针对。
2. 改变对应于指针的键值。
3. 分裂和合并结点。

这些变化中的每一个都可以用简短的日志记录来表示，即使是分裂操作也只需要指明分裂发生在哪里而新结点又在哪里。类似地，合并只需要指明涉及的结点，因为合并方式由所使用的B树管理算法决定。

在满足可恢复调度的要求时，使用这几类逻辑更新记录可以比不用的情况下更早地释放锁。原因在于，只要事务只利用B树来定位所需访问数据的位置，B树块的脏读对读这些块的事务来说就永远也不是问题。

例如，假设事务T读取叶结点N，但最后写入N的事务U后来却中止了，N上的某些改变（例如，由于U插入一个元组而导致N中插入一个新的键-指针对）需要撤销。如果T也向N中插入了键/指针对，那么N就不可能恢复到U修改它以前的状态。但是，U对于N的影响可以撤销；在这个例子中，我们将删除U插入的键-指针对。所得到的N与U执行操作前不一样；它已包含由T所进行的插入。然而，数据库并没有不一致，因为整个B树仍只反映提交事务所做的改变。 999 也就是说，我们已经在一个逻辑层次上而不是物理层次上恢复了B树。 □

19.1.8 根据逻辑日志恢复

若逻辑操作是等幂的（即它们可无损地进行多次重复），则使用逻辑日志进行恢复就比较容易。例如，例19.6中曾讨论过，怎样用元组和该元组在块中的位置在逻辑日志中表示一个元组的插入。若在同一个地方进行至少两次的写元组操作，则结果同一次写一样。因此，在进行

恢复时，若要重做一个插入元组的操作，则可以在适当的位置重复执行插入适当块的操作，而不用担心元组是否已被插入。

相反地，考虑例19.7和例19.8中的两种情况，即元组在同一个块中移动，以及元组在块间移动的情况。此时并不知道元组即将插入到哪个位置，能做的最好情形是在记录中写入如“元组t插入到块B的某个地方”这样的操作。若在恢复时需重做t的插入操作，就可以用块B中t的两份拷贝完成。较坏的情形是不知道块B（持有第一个t的拷贝）是否已把t的拷贝写回磁盘。例如，正往块B中的另一个数据库元素执行写操作的事务可能会使B的一个拷贝写回磁盘。

为了消除不同情形（如使用逻辑日志进行恢复时）之间的歧义，就要使用一种称为逻辑序列号（log sequence number）的技术。

- 每个日志记录的序列号比之前的日志记录序列号大[⊖]。因此，一个典型的逻辑记录的形式为<L, T, A, B>，其中：
  - L是日志的序列号，为整型。
  - T是相关事务。
  - A是T中的操作，如，“插入元组t”。
  - B是操作执行的块。
- 对于每种操作，都有一个进行逻辑undo操作的补偿操作（compensating action）。见例19.8，补偿操作不能把数据库恢复到操作从未发生过的状态S，但它可以把数据库恢复到逻辑等价于S的状态。例如，“插入元组t”的补偿操作是“删除元组t”。
- 若一个事务T中止了，则对于T在数据库上所做的每个操作都要进行一次补偿，事实上，这个操作也被记录在日志中。
- 每个块的头部都保存了影响该块的最后一个操作的序列号。

1000

假设在故障后，我们需要使用逻辑日志来进行恢复。下面给出恢复的步骤。

1. 第一步是重建发生故障时数据库的状态，包括当前值在缓冲器中但随后丢失了的块。此时要做：

(a) 在日志中寻找最近的检查点，由此确定当前活跃的事务集合。

(b) 对于每个日志项<L, T, A, B>，比较块B上的日志序列号N和当前日志记录的日志序列号L。若$N < L$，则重做操作A，此操作从未在块B上执行过。可是，若$N \geqslant L$，则不做任何事。此时，A已对B产生了影响。

(c) 每个日志项（记录了事务T开始执行、提交或中止等活动），根据这个记录来调整相应的活跃事务集合。

2. 当读到日志的尾部时，必须中止仍保持活跃的事务集合。此时要做：

(a) 再次扫描日志，此时是从尾部反向扫至前一个检查点。每次遇到声明事务T必须中止的记录<L, T, A, B>时，为A执行补偿操作，并在日志中进行记录。

(b) 若必须中止一个在最近检查点之前开始的事务（也就是，这个事务在检查点的活跃队列中），则继续反向扫描日志直至找到所有这样的事务起始记录。

(c) 为每个不得不中止的事务，在日志中写下中止记录。

19.1.9 习题

* **习题19.1.1** 考虑在如下动作序列中插入（像18.3节中那样的单一类型）锁的所有方式，

⊖ 最终日志序列号又会从0开始，但由于序列号数非常大，所以不会同时有两个具有相同号的日志记录。

使事务T_1满足：

$$r_1(A);r_1(B);w_1(A);w_1(B)$$ 1001

a) 两阶段封锁且是严格的。

b) 两阶段封锁但不是严格的。

习题19.1.2 假设下面的各个动作序列后面都跟着事务T_1的中止动作。说明哪些事务需要回滚。

* a) $r_1(A)$; $r_2(B)$; $w_1(B)$; $w_2(C)$; $r_3(B)$; $r_3(C)$; $w_3(D)$;

b) $r_1(A)$; $w_1(B)$; $r_2(B)$; $w_2(C)$; $r_3(C)$; $w_3(D)$;

c) $r_2(A)$; $r_3(A)$; $r_1(A)$; $w_1(B)$; $r_2(B)$; $r_3(B)$; $w_2(C)$; $r_3(C)$;

d) $r_2(A)$; $r_3(A)$; $r_1(A)$; $w_1(B)$; $r_3(B)$; $w_2(C)$; $r_3(C)$;

习题19.1.3 考虑习题19.1.2中的各个动作序列，但现在假设三个事务都提交，并在它们的最后一个动作后立即将提交记录写入日志中。但是，崩溃发生了，并且在崩溃发生以前日志尾部尚未写到磁盘因而丢失。根据丢失的日志尾部的起始点，说明：

a) 哪些事务可以认为是未提交的？

b) 在恢复过程中是否产生脏读？如果是，哪些事务需要回滚？

c) 如果丢失的日志不是尾部，而是中间的一部分，那么还可能产生哪些脏读？

! **习题19.1.4** 考虑下面两个事务：

$$T_1\text{: } w_1(A);\ w_1(B);\ r_1(C);\ c_1;$$

$$T_2\text{: } w_2(A);\ r_2(B);\ w_2(C);\ c_2;$$

* a) 有多少T_1和T_2的调度是可恢复的？

b) 在上述可恢复调度中，有多少ACR调度？

c) 有多少调度即是可恢复，又是可串行的？

d) 有多少调度既是ACR，又是可串行的？

习题19.1.5 给出一个ACR调度的例子，这个调度含有共享锁和排它锁，但不是严格的。 1002

19.2 视图可串行性

回忆一下，我们在18.1.4节中曾讨论过，我们设计调度器的真正目标是只允许可串行化的调度。我们也看到了事务对数据所采取操作的不同如何影响调度是否可串行化。在18.2节中，我们学习了通常保证“冲突可串行性”的调度器，这样的调度器不管事务对其数据做什么都能保证可串行性。

但是，有一些比冲突可串行性弱的条件也能保证可串行性。在本节中，我们将考虑这样的一个条件，它称为“视图可串行性”。直观地说，视图可串行性考虑事务T和U之间所有T写一个数据库元素而U又读该元素这样的联系。视图可串行性与冲突可串行性的关键区别体现在事务T写一个其他事务都不读的值A（因为另外的某个事务后来为A写入了自己的值）时。在这种情况下，动作$w_T(A)$可以放到调度中另外的某些（A类似地也永不会被读到的）地方，而这在冲突可串行性定义下是不允许的。本节中我们将讨论视图可串行性的精确定义，并给出判断它的一种标准。

19.2.1 视图等价性

假设我们有同一事务集合的两个调度S_1和S_2。假定调度中任何事务读到的每个数据库元素

都由一个假想的事务T_0写入初值，而另一个假想的事务T_f在每个调度结束后读取一个和多个事务写过的所有数据库元素。那么，对每个调度中的任一读动作$r_i(A)$，我们都可以找到在它之前最靠近它的写动作$w_j(A)$[⊖]。我们说T_j是读动作$r_i(A)$的源。注意，事务T_j可以是假想的初始事务T_0，而T_i可以是T_f。

对调度S_1和S_2来说，如果一个调度中每个读动作的源与另一调度中相同，那么我们说S_1和S_2是*视图等价的*。视图等价调度当然真正等价；在任何一个数据库状态上执行时，它们做的事情都一样。如果调度S视图等价于一个串行调度，那么我们说S是*视图可串行化的*。

例19.9 考虑调度S，其定义如下：

1003

T_1:			$r_1(A)$			$w_1(B)$				
T_2:	$r_2(B)$	$w_2(A)$					$w_2(B)$			
T_3:				$r_3(A)$				$w_3(B)$		

注意，为了更好地表明哪个事务做了什么，我们将各个事务的动作垂直分开；在读调度时仍然应像往常一样从左到右。

在S中，T_1和T_2写的B值都丢失了；只有T_3所写的B值保留到调度结束时，并被假想事务T_f“读”到。S不是冲突可串行化的。要明白为什么，首先请注意T_2在T_1读A前写A，因此在假想的冲突等价串行调度中，T_2必须位于T_1前。动作$w_1(B)$在$w_2(B)$前这一事实又要求在任何冲突等价的串行调度中T_1必须位于T_2前。实际上，$w_1(B)$和$w_2(B)$对数据库都不产生长期的影响。正是这种类型的无关写操作使得在确定等价串行调度上的真正约束时视图可串行性能够忽略。

更精确地讲，我们考虑S中所有读动作的源：

1. $r_2(B)$的源是T_0，因为它之前没有写过B。
2. $r_1(A)$的源是T_2，因为T_2在此读操作前最后写A。
3. 类似地，$r_3(A)$的源是T_2。
4. 假想的T_f读A动作的源是T_2。
5. 假想的T_f读B的源是最后写B的事务T_3。

当然，T_0在任何调度中都出现在所有真正的事务前，而T_f出现在所有事务后。如果我们将真正的事务排列为（T_2，T_1，T_3），那么所有读动作的源和调度S相同。也就是说，T_2读B且T_0显然是前一“写事务”。T_1读A，但T_2已写A，所以$r_1(A)$的源和S中一样是T_2。T_3也读A，但由于其前T_2写A，所以$r_3(A)$的源和S中一样是T_2。最后，假想事务T_f读A和B，而调度（T_2，T_1，T_3）中A和B最后分别由T_2和T_3写入，也和S中一样。我们的结论是，S是一个视图可串行化的调度，而由顺序（T_2，T_1，T_3）表示的调度是一个视图等价的调度。 □

19.2.2 多重图与视图可串行性的判断

在18.2.2节中，我们曾用优先图判断(test)冲突可串行性；优先图的一种推广形式能反映视图可串行性定义所需的所有先后次序约束。我们定义调度的*多重图*构成如下：

1. 每个事务对应一个结点再加上对应于假想事务T_0和T_f的结点。

1004

2. 对应每个源为T_j的动作，$r_i(X)$，有一条从T_j到T_i的弧。
3. 假设T_j是$r_i(X)$的源，而T_k是另一个写X的事务。T_k不允许插入T_j和T_i之间，所以它必须出现在T_j前或T_i后。我们用从T_k到T_j以及从T_i到T_k的弧对（用虚线表示）来表示这种情况。直观

⊖ 尽管我们在前面没有禁止一个事务两次写同一元素，但事务通常没有必要这样做，并且在这里的讨论中需要假设对给定元素一个事务只写一次。

地讲，弧对中的一个或另一个是“真实的”，但我们并不关心是哪个，并且当试图使多重图无环时，我们可以选择二者中有助于使多重图无环的任一条弧。但是，在一些重要的特殊情况下，弧对将变成单弧：

(a) 如果T_j是T_0，那么T_k不可能出现在T_j前，因此我们用弧$T_i \rightarrow T_k$代替该弧对。

(b) 如果T_i是T_f，那么T_k不可能出现在T_i后，因此我们用弧$T_k \rightarrow T_j$代替该弧对。

例19.10 考虑例19.9中的调度。我们在图19-4中给出了S的多重图开始时的情况，其中只有结点和根据规则2所能得到的弧。我们还指明了导致弧产生的数据库元素。也就是说，A从T_2传到T_1、T_3和T_f，而B从T_0传到T_2、T_3和T_f。

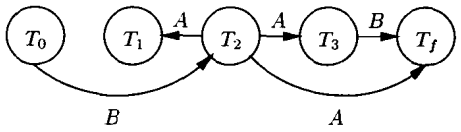


图19-4 例19.10的初始多重图

现在，我们必须考虑对这五个联系中的每一个而言，哪些事务可能由于在它们之间写同一元素而产生干扰。这些潜在的干扰用规则（3）的弧对排除，尽管正如我们将看到的那样，在这个例子中每个弧对都由于属于某种特殊情况而变成单弧。

考虑基于元素A的弧$T_2 \rightarrow T_1$。写A的事务只有T_0和T_2，而它们都不能到此弧中间来，因为T_0不能移动位置，而T_2已经是弧的一个端点，因此不需要附加其他的弧。通过类似的推断，我们可以知道，使A的写事务位于弧$T_2 \rightarrow T_3$和$T_2 \rightarrow T_f$外并不需要额外的弧。

现在考虑基于B的弧。注意，T_0、T_1、T_2和T_3都写B。首先考虑弧$T_0 \rightarrow T_2$。T_1和T_3是写B的其他事务；T_0和T_2也写B，但正如我们所看到的那样，弧端点不会带来干扰，所以不必考虑它们。由于我们不能把T_1置于T_0和T_2之间，原则上需要弧对（$T_1 \rightarrow T_0$，$T_2 \rightarrow T_1$）。然而，任何事务都不能位于T_0前，所以$T_1 \rightarrow T_0$这一选项是不可能的。在这种特殊情况下，我们可以只在多重图中加入弧$T_2 \rightarrow T_1$。但因为A，该弧已经存在，所以实际上要使T_1位于$T_0 \rightarrow T_2$外并不需要对多重图进行改变。 1005

我们也不能把T_3放在T_0和T_2之间。通过类似的推理，我们知道需要加入弧$T_2 \rightarrow T_3$而不是一个弧对。然而，这条弧也已经由于A而存在于多重图中，所以我们不做改变。

接下来考虑弧$T_3 \rightarrow T_f$。由于T_0、T_1和T_2是写B的其他事务，我们必须使它们都位于此弧外。T_0不可能移动到T_3和T_f之间，但T_1和T_2可以。由于二者都不能移到T_f后，所以我们必须限制T_1和T_2出现在T_3前。弧$T_2 \rightarrow T_3$已经存在，但我们必须在多重图中加入弧$T_1 \rightarrow T_3$。这是我们必须加入多重图中的惟一的弧；该多重图最终的弧集合如图19-5所示。 □

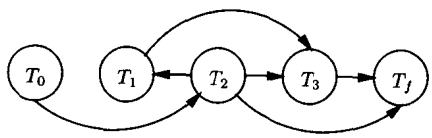


图19-5 例19.10的完整多重图

例19.11 在例19.10中，所有弧对都因为是特殊情况而变成单弧。图19-6是四个事务构成的一个调度，它是多重图中存在真正弧对的调度的例子。

图19-7所示的多重图中只给出了表示源—读事务联系的弧。和图19-4中一样，我们标出了导致弧产生的元素。接下来必须考虑可能添加弧的各个方面。正如在例19.10中看到的那样，

1006 我们可以做一些简化。在避免对弧$T_j \rightarrow T_i$的干扰时，所需考虑的事务T_k（不能出现在中间的事务）只有：

- 对导致弧$T_j \rightarrow T_i$的元素执行写操作的事务；
- 但不是T_0和T_f，它们都不可能是T_k；并且
- 不是T_i和T_j，它们是弧本身的端点。

T_1	T_2	T_3	T_4
	$r_2(A)$;		
$r_1(A)$; $w_1(C)$;			
		$r_3(C)$;	
$w_1(B)$;			
			$r_4(B)$;
		$w_3(A)$;	
			$r_4(C)$;
	$w_2(D)$; $r_2(B)$;		
			$w_4(A)$; $w_4(B)$;

图19-6 多重图中需要弧对的事务之例

记住这些规则，我们来考虑由于数据库元素A而产生的弧，该数据库元素被T_0、T_3和T_4所写。我们根本不需要考虑T_0。T_3不能介于$T_4 \rightarrow T_f$之间，因此我们加入弧$T_3 \rightarrow T_4$；不要忘了该弧对中的另一条弧$T_f \rightarrow T_3$不是一种选择。类似地，T_3不能介于$T_0 \rightarrow T_1$或$T_0 \rightarrow T_2$之间，这导致产生了弧$T_1 \rightarrow T_3$和$T_2 \rightarrow T_3$。

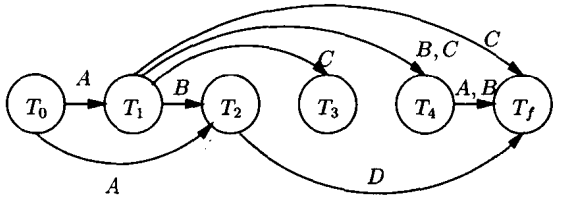


图19-7 例19.11的初始多重图

现在，考虑T_4由于A也不能介入弧中这一事实。T_4是弧$T_4 \rightarrow T_f$的一个端点，因此这条弧是无关的。T_4不能介于$T_0 \rightarrow T_1$或$T_0 \rightarrow T_2$之间，这导致弧$T_1 \rightarrow T_4$和$T_2 \rightarrow T_4$的产生。

我们接下来考虑由于B而产生的弧，该数据库元素被T_0、T_1和T_4所写。我们仍然不需要考虑T_0。由于B而产生的弧只有$T_1 \rightarrow T_2$、$T_1 \rightarrow T_4$和$T_4 \rightarrow T_f$。在前两个中，T_1都不可能介于其间，但第三个需要加入弧$T_1 \rightarrow T_4$。

T_4只可能干涉$T_1 \rightarrow T_2$。这条弧的两个端点都不是T_0或T_f，因此它真正需要弧对（$T_4 \rightarrow T_1$，$T_2 \rightarrow T_4$）。在图19-8中，我们给出了这一弧对以及加入的所有其他的弧。

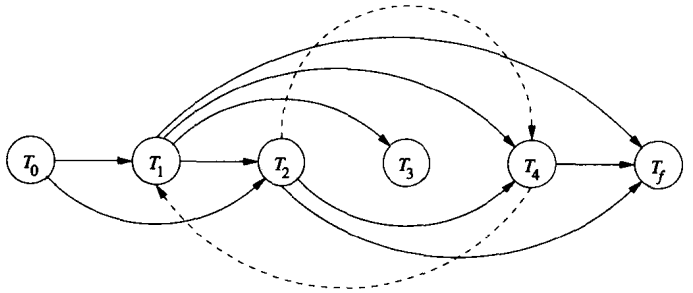


图19-8 例19.11的完整多重图

接下来，我们考虑写C的事务T_0和T_1。和前面一样，T_0不会带来问题。另外，T_1是每一条由C导致的弧的一部分，因此它不可能介于其间。类似地，D只被T_0和T_2所写，因此我们可以断定不再需要更多的弧。这样，最后得到的多重图如图19-8所示。 □

19.2.3 视图可串行性的判断

因为在每个弧对中我们必须选一条且只能选一条，我们能为调度S找到一个等价的串行顺序，当且仅当在每个弧对中选一个能使S的多重图变成无环图。要明白原因，请注意如果存在 1007
这样的一个无环图，那么该图的任一拓扑排序中所有写事务都不会介于读事务与其源之间，并且每个写事务都在它的读事务之前。因此，串行顺序中的读事务—源联系与S中完全一样；这两个调度是视图等价的，因此S是视图可串行化的。

反过来，如果S是视图可串行化的，那么就会有一个视图等价的串行顺序S'。对于S的多重图中的每一弧对（$T_k \rightarrow T_j$，$T_i \rightarrow T_k$），在S'中T_k只能出现在T_j前或T_i后；否则T_k的写将打破从T_j到T_i的联系，这意味着S和S'不是视图等价的。类似地，多重图中的每条弧在S'的事务顺序中都应得到遵循。我们的结论是，从每个弧对中选出一条弧，那么必然存在一种选择使串行调度S'与这种选择所产生的图中的所有弧都一致。因此，这个图是无环的。

例19.12 考虑图19-5中的多重图。它已经是一个图并且是无环的。惟一的拓扑顺序是（T_2，T_1，T_3），因此这就是例19.10中调度的一个视图等价串行顺序。

现在考虑图19-8中的多重图。我们必须考虑其中弧对的每一种选择。如果我们选择$T_4 \rightarrow T_1$，那么图中有环。但是，如果我们选择$T_2 \rightarrow T_4$，那么产生的将是一个无环图。该图惟一的拓扑顺序是（T_1，T_2，T_3，T_4）。这一顺序产生了一个视图等价的串行顺序，并且表明原来的调度是视图可串行化的。 □

19.2.4 习题

习题19.2.1 为下列调度画出多重图，并找出所有视图等价串行顺序：

* a) $r_1(A)$; $r_2(A)$; $r_3(A)$; $w_1(B)$; $w_2(B)$; $w_3(B)$;

b) $r_1(A)$; $r_2(A)$; $r_3(A)$; $r_4(A)$; $w_1(B)$; $w_2(B)$; $w_3(B)$; $w_4(B)$; 1008

c) $r_1(A)$; $r_3(D)$; $w_1(B)$; $r_2(B)$; $w_3(B)$; $r_4(B)$; $w_2(C)$; $r_5(C)$; $w_4(E)$; $r_5(E)$; $w_5(B)$;

d) $w_1(A)$; $r_2(A)$; $w_3(A)$; $r_4(A)$; $w_5(A)$; $r_6(A)$;

! **习题19.2.2** 下面是一些串行调度。说明有多少调度是：（i）冲突等价（ii）视图等价于这些串行调度。

* a) $r_1(A)$; $w_1(B)$; $r_2(A)$; $w_2(B)$; $r_3(A)$; $w_3(B)$; 即三个事务都读A，然后写B。

b) $r_1(A)$; $w_1(B)$; $w_1(C)$; $r_2(A)$; $w_2(B)$; $w_2(C)$; 即两个事务都读A，然后写B和C。

19.3 死锁处理

我们已经几次发现并发执行的事务由于竞争资源而到达一个存在死锁的状态：若干事务中的每一个都在等待被其他事务占有的资源，因而每个事务都不能取得进展。

- 即使是两阶段封锁事务的普通操作也可以导致死锁，在18.3.4节中我们看到了这是怎样发生的，其原因在于一个事务封锁了另一事务也需要封锁的东西。
- 将锁从共享升级为排他的能力可能导致死锁，在18.4.3节中我们看到了这是怎样发生的，

其原因在于每个事务在同一元素上持有共享锁，并且希望将锁升级。

处理死锁的方法大致分为两种。我们可以检测死锁并进行修复，也可以对事务进行管理，使死锁永远都不可能形成。

19.3.1 超时死锁检测

当存在死锁时，对该状态进行修复以使所有涉及的事务都能继续执行通常是不可能的。因此，至少一个事务必须回滚——终止并重新开始。

检测并解决死锁最简单的方法是利用*超时*。对事务活跃的时间做出限制，如果事务超过这个时间就将其回滚。例如，在一个典型事务执行时间为几毫秒的简单事务系统中，以一分钟为超时时间只会影响到陷于死锁中的事务。但是，如果有的事务比较复杂，我们可能希望超时时间间隔更长一些。

1009 注意，当死锁涉及的一个事务超时后，该事务将释放锁和其他资源。因此，死锁涉及的其他事务有可能在到达超时限制前完成。然而，由于死锁涉及的事务可能几乎在同一时间开始（否则，一个事务可能在另一事务开始前已经完成），不再陷于死锁中的事务假超时也是可能发生的。

19.3.2 等待图

由于事务等待另一事务持有的锁而导致的死锁问题可以用*等待图*来解决，等待图表明哪些事务在等待其他事务持有的锁。这种图可以用来在死锁形成后检测死锁，也可以用来预防死锁的形成。我们假设是后一种，在任何时候我们都需要维护等待图，并拒绝将在图中产生环的动作。

回忆一下，在18.5.2节中，锁表为每个数据库元素X维护等待X上的锁以及当前持有X上的锁的事务列表。等待图中对应当前持有锁和等待锁的每个事务有一个结点。对于结点（事务）T和结点U，如果存在某个数据库元素使：

1. U持有A上的一个锁；
2. T等待A上的一个锁；并且
3. 除非U先释放它在A上持有的锁，否则T不能获得所需封锁方式的锁；[⊖]

如果在等待图中无环，那么每个事务最终都能完成。至少有一个事务不在等待其他事务，该事务肯定能完成。这时，至少有另一个事务不在等待，这个事务又能完成；依此类推。

然而，如果图中有环，那么环中的任何事务都不能取得进展，因此存在死锁。这时，避免死锁的一种策略是回滚所提请求将导致等待图中出现环的任一事务。

例19.13 假设我们有下面的四个事务，每个事务都读一个元素和写另一元素：

1010
T_1: $l_1(A)$; $r_1(A)$; $l_1(B)$; $w_1(B)$; $u_1(A)$; $u_1(B)$;

T_2: $l_2(C)$; $r_2(C)$; $l_2(A)$; $w_2(A)$; $u_2(C)$; $u_2(A)$;

T_3: $l_3(B)$; $r_3(B)$; $l_3(C)$; $w_3(C)$; $u_3(B)$; $u_3(C)$;

T_4: $l_4(D)$; $r_4(D)$; $l_4(A)$; $w_4(A)$; $u_4(D)$; $u_4(A)$;

我们使用一个只有一种封锁方式的简单封锁系统，实际上，如果使用共享/排他系统，并用合适的方式封锁，即读时使用共享锁而写时使用排他锁，我们也会发现同样的效果。

⊖ 通常情况下，例如在共享锁和排他锁的情况下，每个等待的事务都必须等到当前的所有锁持有者释放其锁。但在某些封锁方式的系统中，事务可以在某个当前被持有的锁释放后就能获得其锁；见习题19.3.6。

图19-9中是这四个事务的一个调度最开始的部分。在前四步中，每个事务都获得了该事务要读的元素上的锁。在第5步，T_2试图封锁A，但由于T_1已持有A上的锁，所以这一请求被拒绝。因此，T_2等待T_1，我们画一条从T_2的结点到T_1的结点的弧。

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$;			
2)		$l_2(C)$; $r_2(C)$;		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(D)$; $r_4(D)$;
5)		$l_2(A)$; 被拒绝		
6)			$l_3(C)$; 被拒绝	
7)				$l_4(A)$; 被拒绝
8)	$l_1(B)$; 被拒绝			

图19-9 一个有死锁的调度的开始部分

类似地，在第6步，T_3对C的封锁请求由于T_2而被拒绝；在第7步，T_4对A的封锁请求由于T_1而被拒绝。这时的等待图如图19-10所示，此图中无环。

在第8步，T_1必须等待T_3在B上持有的锁。如果我们允许T_1等待，那么等待图中将有一个包含T_1、T_2和T_3的环，如图19-11所示。由于它们各自都等待另一事务完成，它们都不能取得进展，因此存在涉及这三个事务的死锁。T_4碰巧也不能完成，尽管它不在环中，但它的进展依赖于T_1所取得的进展。 1011

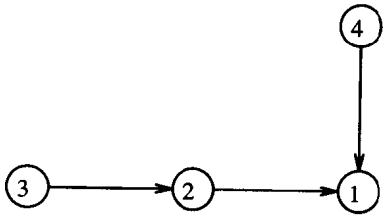


图19-10 图19-9中第7步后的等待图

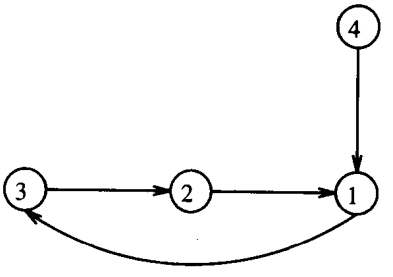


图19-11 有环的等待图，该环由图19-9中第8步导致

既然我们需要回滚导致环的任一事务，T_1必须回滚，所产生的等待图如图19-12所示。T_1放弃自己在A上锁，该锁可以给T_2或T_4。假设给T_2，那么T_2可以完成，然后T_2释放自己在A和C上的锁。现在需要封锁C的T_3和需要封锁A的T_4都可以完成。在某个时候，T_1重新开始，但在T_2、T_3和T_4完成以前它不能获得A和B上的锁。 □

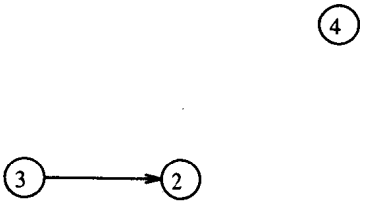


图19-12 T_1回滚后的等待图

19.3.3 通过元素排序预防死锁

现在，我们考虑预防死锁的几种其他的方法。第一种方法需要我们将数据库元素按某种任意但固定的顺序排列。例如，如果数据库元素是块，我们可以将它们按物理地址的字典顺序排列。回忆一下，在8.3.4节中，块的物理地址通常用描述块在存储系统中位置的一系列字节表示。

如果每个事务申请元素上的锁都必须按顺序（一个在大多数应用中都不太现实的条件），那么就不会由于事务等待锁而导致死锁。为了证明这一点，假设T_2等待T_1在A_1上持有的锁；T_3等待T_2在A_2上持有的锁；依此类推；而T_n在等待T_{n-1}在A_{n-1}上持有的锁；并且T_1在等待T_n在A_n上

1012 持有的锁。由于T_2持有A_2上的锁而又等待A_1，在元素的顺序中必然有$A_2 < A_1$。类似地，对$i = 3,4,\cdots,n$，有$A_i < A_{i-1}$。但是，由于T_1持有A_1上的锁而又等待A_n，这说明$A_1 < A_n$。我们现在有$A_1 < A_n < A_{n-1} < \cdots < A_2 < A_1$，这是不可能的，因为这蕴含着$A_1 < A_1$。

例19.14 我们假设元素按字母顺序排列。那么，如果例19.13中的四个事务要按字母顺序封锁元素，则T_2和T_4需要重写，把封锁元素的顺序反过来。因此，这四个事务现在是：

T_1: $l_1(A)$; $r_1(A)$; $l_1(B)$; $w_1(B)$; $u_1(A)$; $u_1(B)$;

T_2: $l_2(A)$; $l_2(C)$; $r_2(C)$; $w_2(A)$; $u_2(C)$; $u_2(A)$;

T_3: $l_3(B)$; $r_3(B)$; $l_3(C)$; $w_3(C)$; $u_3(B)$; $u_3(C)$;

T_4: $l_4(A)$; $l_4(D)$; $r_4(D)$; $w_4(A)$; $u_4(D)$; $u_4(A)$;

图19-13给出了执行时机与图19-9相同时这些事务的执行情况。T_1开始并获得A上的锁。T_2接着开始并试图获得A上的锁，但必须等待T_1。然后，T_3开始并获得B上的锁，但T_4不能开始，它也需要A上的锁，因此它必须等待。

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$;			
2)		$l_2(A)$; 被拒绝		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(A)$; 被拒绝
5)			$l_3(C)$; $w_3(C)$;	
6)			$u_3(B)$; $u_3(C)$;	
7)	$l_1(B)$; $w_1(B)$;			
8)	$u_1(A)$; $u_1(B)$;			
9)		$l_2(A)$; $l_2(C)$;		
10)		$r_2(C)$; $w_2(A)$;		
11)		$u_2(A)$; $u_2(C)$;		
12)				$l_4(A)$; $l_4(D)$;
13)				$r_4(D)$; $w_4(A)$;
14)				$u_4(A)$; $u_4(D)$;

图19-13 按字母顺序封锁元素可预防死锁

由于T_2停顿，不能继续执行下去，而按照图19-9中的顺序，接下来是T_3。T_3可以获得C上的
1013 锁，然后它在第6步完成。现在，由于T_3在B和C上的锁释放，T_1在第8步得以完成。这时，A上的锁成为可获得的，而我们假定按先来先服务的原则将此锁给T_2。那么，T_2获得它所需要的两个锁，并在第11步完成。最后，T_4获得其锁并完成。 □

19.3.4 时间戳死锁检测

正如在19.3.2节讨论的那样，我们可以通过维护等待图来检测死锁。然而，等待图可能很大，而每次事务需要等待锁时分析等待图看是否有环可能很耗时。维护等待图的另一种可选方案是将每个事务与一个时间戳关联起来。该时间戳：

- 只用于死锁检测；它和18.8节中用于并发控制的时间戳不同，即使使用基于时间戳的并发控制机制。
- 特别地，如果事务回滚，那么它以一个新的、较晚的并发时间戳重新开始，但其用于死锁检测的时间戳从不改变。

时间戳在事务T必须等待另一事务U持有的锁时使用。根据是T还是U更老一些（时间戳更早），可能发生两种不同的情况。两种不同的策略可以用于管理事务和检测死锁。

1. 等待-死亡方案：

(a) 如果T比U老（即T的时间戳比U的时间戳小），那么允许T等待U持有的锁。

(b) 如果U比T老，那么T“死亡”；T将回滚。

2. 伤害-等待方案：

(a) 如果T比U老，它将“伤害”U；这样的伤害通常是致命的：U必须回滚并放弃T需要从U得到的所有锁。一个例外是在“伤害”生效前U已经完成并释放自己的锁。在这种情况下，U得以存活并且不需要回滚。

(b) 如果U比T老，那么T等待U持有的锁。

例19.15 以例19.14中的事务为例，我们来考虑等待-死亡方案。我们将假设T_1、T_2、T_3、T_4是时间的顺序；即T_1是最老的事务。我们还假设当事务回滚时，该事务的重新启动不会很快，不会在其他事务完成前变得活跃。

图19-14给出了等待-死亡方案下一个可能的动作序列。T_1首先获得A上的锁。当T_2请求A上的锁时，T_2死亡，因为T_1比T_2老。在第3步，T_3获得B上的锁，但在第4步T_4请求A上的锁，由于 1014
A上锁的持有者T_1比T_4老，因而T_4死亡。接下来，T_3获得C上的锁并完成。当T_1继续时，它发现B上的锁可以获得，因而也在第8步完成。

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$;			
2)		$l_2(A)$; 死亡		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(A)$; 死亡
5)			$l_3(C)$; $w_3(C)$;	
6)			$u_3(B)$; $u_3(C)$;	
7)	$l_1(B)$; $w_1(B)$;			
8)	$u_1(A)$; $u_1(B)$;			
9)				$l_4(A)$; $l_4(D)$;
10)		$l_2(A)$; 等待		
11)				$r_4(D)$; $w_4(A)$;
12)				$u_4(A)$; $u_4(D)$;
13)		$l_2(A)$; $l_2(C)$;		
14)		$r_2(C)$; $w_2(A)$;		
15)		$u_2(A)$; $u_2(C)$;		

图19-14 用等待-死亡方案检测死锁的事务动作 1015

基于时间戳的死锁检测发挥作用的原因

我们断言在等待-死亡方案和伤害-等待方案下等待图中都不会出现环，因而不存在死锁。假设不是这样；即存在环，例如$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$。有一个事务最老，假设是$T_2$。

在等待-死亡方案中只会等待比较新的事务。因此，T_1不可能等待T_2，因为T_2肯定T_1老。在伤害-等待方案中只会等待比较老的事务。因此，T_2不可能等待比较新的T_3。我们断定环不可能存在，因此不存在死锁。

现在，回滚的两个事务T_2和T_4重新开始。就死锁而言，它们的时间戳不变；T_2仍然比T_4老。但是，我们假设在第9步T_4首先重启，因此在第10步当T_2请求A上的锁时，T_2被迫等待但不必中止。T_4在第12步完成，然后T_2得以运行至结束，正如最后三步所给出的那样。 □

例19.16 下面，我们考虑同样的事务在伤害-等待策略下的运行，如图19-15所示。和图19-14中一样，T_1首先封锁A。当第2步中T_2请求A上的锁时，它需要等待，因为T_1比T_2老。在第3

步T_3获得在B上的锁后，T_4也不得不等待A上的锁。

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$;			
2)		$l_2(A)$; 等待		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(A)$; 等待
5)	$l_1(B)$; $w_1(B)$;		被伤害	
6)	$u_1(A)$; $u_1(B)$;			
7)		$l_2(A)$; $l_2(C)$;		
8)		$r_2(C)$; $w_2(A)$;		
9)		$u_2(A)$; $u_2(C)$;		
10)				$l_4(A)$; $l_4(D)$;
11)				$r_4(D)$; $w_4(A)$;
12)				$u_4(A)$; $u_4(D)$;
13)			$l_3(B)$; $r_3(B)$;	
14)			$l_3(C)$; $w_3(C)$;	
15)			$u_3(B)$; $u_3(C)$;	

图19-15 用伤害-等待方案检测死锁的事务动作

接着，假设T_1继续执行并在第5步请求B上的锁。该锁已被T_3持有，但T_1比T_3老。因此，T_1“伤害”T_3。由于T_3尚未完成，这一伤害是致命的：T_3放弃自己的锁并回滚。因此，T_1得以完成。

当T_1使A上的锁可用后，假设该锁被T_2得到，这样T_2就能够继续执行。在T_2后，该锁被T_4获得，T_4继续执行至完成。最后，T_3重启并在不受干扰的情况下完成。 □

19.3.5 死锁管理方法的比较

在等待-死亡方案和伤害-等待方案中，较老的事务消灭较新的事务。由于事务以旧时间戳重启，最终每个事务都将变成系统中最老的事务而必然能完成。每个事务最终都能完成的这一
1016 保证称为*无饿死*。注意，本节描述的其他方法不一定能防止饿死；如果不采取额外的措施，事务可能不断重启，陷入死锁，然后回滚。参见习题19.3.7。

然而，等待-死亡方案和伤害-等待方案的行为有着细微的差别。在伤害-等待中，只要老事务请求较新的事务持有的锁，较新的事务就被杀死。如果我们假设事务在开始后较近的时间内获得锁，那么老事务抢夺新事务持有的锁的情况就很少发生。因此，我们可以预见在伤害-等待中回滚比较少见。

另一方面，当回滚发生时，等待-死亡回滚仍处于获得锁这一阶段的事务，这一阶段被假定是事务中最早的阶段。因此，尽管等待-死亡可能比伤害-等待回滚的事务多，这些事务常常只做了极少的工作。与此相比，当伤害-等待回滚事务时，该事务可能已经获得自己的锁，并且它的活动可能已经占用了大量的处理器时间。因此，根据处理事务的数量不同，两种方案都可能浪费更多的工作。

我们还应该比较等待-死亡和伤害-等待与直接构造和使用等待图的优缺点。最重要的几点为：

- 等待-死亡和伤害-等待的实现都比维护或周期性构造等待图更容易。当数据库是分布式数据库时，构造等待图的缺点更加显著，这时等待图必须通过各个节点上的锁表集合来构造。参见19.6节的讨论。
- 使用等待图能使由于死锁而必须中止事务的次数最少。除非真的存在死锁，否则我们绝不会中止事务。另一方面，等待-死亡和伤害-等待偶尔都会在并不存在死锁的情况下回滚事务，而这时如果该事务不被杀死，死锁也不会发生。

19.3.6 习题

习题19.3.1 在下面的每个动作序列中，假设共享锁恰好在每个读动作前申请，而排他锁恰好在每个写动作前申请。此外，解锁恰好发生在事务执行的最后一个动作后。说明哪些动作被拒绝以及是否有死锁发生，并说明在动作执行过程中等待图怎样演变。如果存在死锁，则选择一个事务并将其中止，说明动作序列将怎样继续下去。

* a) $r_1(A); r_2(B); w_1(C); r_3(D); r_4(E); w_3(B); w_2(C); w_4(A); w_1(D);$

b) $r_1(A); r_2(B); r_3(C); w_1(B); w_2(C); w_3(D);$ 1017

c) $r_1(A); r_2(B); r_3(C); w_1(B); w_2(C); w_3(A);$

d) $r_1(A); r_2(B); w_1(C); w_2(D); r_3(C); w_1(B); w_4(D); w_2(A);$

习题19.3.2 对习题19.3.1中的每个动作序列，说明在伤害-等待死锁避免系统下将会发生什么。假设死锁时间戳的顺序与事务下标相同，即T_1、T_2、T_3和T_4。还假设事务的重启按照这些事务回滚的顺序来进行。

习题19.3.3 对习题19.3.1中的每个动作序列，说明在等待-死亡死锁避免系统下将会发生什么。所做假设同习题19.3.2。

! **习题19.3.4** 对任意整数$n > 1$，是否存在这样的等待图，其中有一个长度为n的环却没有更小的环？当$n = 1$时即一个结点上的环时又怎样？

!! **习题19.3.5** 避免死锁的一种方法是要求每个事务在开始时声明自己需要的所有锁，然后或者授予该事务需要的所有锁，或者都不授予而让该事务等待。这种方式能避免由于封锁造成的死锁吗？如果能，解释原因；如果不能，举出一个可能发生死锁的例子。

! **习题19.3.6** 考虑18.6节的意向锁系统。描述如何为这种封锁方式的系统构造等待图。特别地，请考虑数据库元素A被不同事务以S和IX方式封锁这种可能性。如果一个对A封锁的请求需要等待，那么我们应该画什么弧？

*! **习题19.3.7** 在19.3.5节中，我们指出伤害-等待和等待-死亡以外的死锁检测方法不一定能防止饿死，即事务可能重复回滚而永远不能完成。举例说明，如果使用的策略是回滚任一可能导致环的事务，这将怎样导致饿死。要求事务以固定的顺序申请元素上的锁一定能防止饿死吗？超时这一死锁处理机制又怎样呢？

19.4 分布式数据库

我们现在来考虑分布式数据库系统的要素。在分布式系统中，许多相对自治的处理器可能参与数据库操作。分布式数据库能提供若干机会：

1. 由于在处理一个问题时可以使用许多机器，并行以及加快查询反应速度的可能性增大。 1018

2. 由于数据可以在多个节点上存在副本，系统可能不会仅仅由于一个节点或部件发生故障而不得不停止处理。

另一方面，分布式处理增加了数据库系统各个方面的复杂性，因此，即使是DBMS中最基本的组成部分的设计，我们也需要重新考虑。在许多分布式环境中，通信开销可能远大于处理开销，因此关键的问题是消息如何传送。这一节我们将引入最基本的问题，而后面几节主要讨论分布式数据库中出现的两大问题：分布式提交和分布式封锁。

19.4.1 数据的分布

数据分布的一个重要原因是由于组织机构自身分布在若干节点上，而每个节点都有主要与

该节点密切相关的数据。例如：

1. 一个银行可能有许多分行。每个分行（或给定城市中所有分行构成的一个组）将保存该分行（城市）维护的账户数据库。客户可以选择在任何一个分行存款，但常常会在“自己的”分行即保存该客户账户数据的分行存款。银行的中央机构也可能保存数据，如员工记录或像当前利率这样的政策。当然，各分行的记录都有备份，这个备份可能既不在分行机构，也不在中央机构。

2. 一个连锁店可能有许多单独的商店。每个商店（或给定城市所有商店构成的一个组）有该商店的销售和存货数据库。可能有一个中央机构，它保存员工数据、整个连锁店的存货数据、信用卡客户数据以及供应商的信息，例如货物尚未交付的订单和欠债情况。此外，所有商店销售数据的一个副本可能都存放在一个“数据仓库”中，可以用来通过分

影响通信开销的因素

由于带宽开销迅速减小，有人可能对设计分布式数据库系统时是否需要考虑通信开销感到疑惑。现在，某些类型的数据属于电子方式管理的大对象，因此即使在通信开销极小时，以太字节计的数据的传输开销也是不能忽略的。此外，通信开销通常不仅仅涉及数据的传送，还有为数据传送做准备的各层协议、在接收方重建数据以及通信的管理。这些协议各自都需要大量的计算。尽管计算开销也在减小，与关键数据库操作的传统单处理器执行相比，进行通信所需的计算可能仍然不能忽视。

析员的即席查询分析和预测销售情况；参见20.4节。

3. 一个数字图书馆可能由一些大学联合构成，每个大学都有一些在线书籍和其他文档。在任一节点上进行搜索都将查看所有节点文档的目录，并且在某个节点上有满足条件的文档时提交该文档的一个电子拷贝。

在某些情况下，逻辑上看做一个关系的数据可能在多个节点上进行划分。例如，我们可以认为连锁商店只有单独的一个关于销售的关系，如

1019

```
Sales (item, date, price, purchaser)
```

但是，这个关系并非物理地存在，而是一些模式相同的关系的并，其中的每个关系存在连锁店的一个商店中。这些局部的关系称为片段，而将这个逻辑关系划分为物理片段则称为关系Sales的水平分解。我们认为这一划分是“水平的”，是因为我们可以把这一划分看做是用若干水平线将单一的关系Sales的元组分隔成每个商店的元组集合。

在另一些情况下，分布式数据库似乎对关系做“垂直”划分，即将一个逻辑的关系分解为两个或更多关系，每个关系有原来的一个属性子集且位于不同节点。例如，如果我们想要找出Boston商店的哪些销售记录属于信用卡付款拖欠90天以上的客户，最好有一个关系（或视图）包含Sales中的货物、日期、购买者信息以及该购买者最后一次信用卡付款日期。但是，在我们所描述的场景中，这一关系是垂直分解的，并且需要在总部将Boston商店的Sales片段与信用卡客户关系进行连接。

19.4.2 分布式事务

数据分布的结果是一个事务可能涉及多个节点的处理，因此事务的模型必须修改。事务不再是单个节点上的单个处理器所执行的一段代码，该处理器只需与一个调度器和一个日志管理器进行通信。相反，事务由一些相互通信的事务成分构成，每个部件位于不同的节点并与局部的调度器和日志管理器通信。因此，两个需要重新考虑的重要问题是：

1020

1. 如何管理分布事务的提交/中止决定？如果事务的一个部件希望中止整个事务，而其他部件没有遇到任何问题因而希望提交事务，这时会发生什么？我们在19.5节讨论一种称为“两阶段提交”的技术；它能做出正确的决定，并且常常允许正常的节点在其他节点发生故障时仍能继续工作。

2. 我们怎样保证涉及多个节点上部件的事务的可串行性？在19.6节我们具体讨论封锁，看一看锁表怎样用来支持数据库元素的全局封锁，并因而支持分布式环境中事务的可串行性。

19.4.3 数据复制

分布式系统的一个重要好处是可以复制数据，即在不同的节点上建立数据的副本。这样做的一个动机是，如果一个节点发生故障，可能另一个节点可以提供与故障节点相同的数据。另一个作用是可以通过在提交查询的节点上建立所需数据的副本来提高回答查询的速度。例如：

1. 银行可以在每个分行建立当前利率政策的副本，这样关于利率的查询就不需要送到中央机构。

2. 连锁店可以在每个商店维护供应商信息副本，这样，对供应商信息的局部查询（例如，经理希望知道某个供应商的电话以查询送货情况）就可以在不向中央机构传送消息的前提下得以解决。

3. 在电子图书馆的例子中，如果某个学校的学生需要读某个指定的文档，该文档的一个副本可以暂时缓存在这个学校。

然而，数据复制带来了几个问题。

a) 我们怎样保持副本相互一致？实质上，对有副本的数据库元素的更新将变成更新所有副本的分布式事务。

b) 我们怎样确定维护多少副本以及在什么地方维护副本？副本越多，更新就越难，而查询就越容易。例如，一个极少更新的关系可以在各个地方都有副本，使效率最高；而一个频繁更新的关系或许就只有一两个副本。

c) 当网络通信发生故障时，同一数据的不同副本可能各自演化，因而在网络连接恢复时必须对各个副本进行协调，这种情况需要做什么？ 1021

19.4.4 分布式查询优化

分布式数据的存在还影响到物理查询计划（见16.7节）设计的复杂性和可选方案。在选择物理计划时必须考虑的问题包括：

1. 如果某个所需关系R有多个副本，那么我们应该从哪个副本中获得R的值？

2. 当我们在两个关系R和S上实施某个操作例如连接时，有多个可选方案而且必须选择其中之一。一些可能的选项如下：

(a) 我们可以将S复制到R所在节点，并在该节点执行计算。

(b) 我们可以将R复制到S所在节点，并在该节点执行计算。

(c) 我们可以将R和S都复制到二者各自所在节点之外的第三个节点，并在该节点执行计算。

哪种选择最好，这依赖于多个因素，其中包括哪个节点上有可用的处理时间以及操作结果是否需要与第三个节点上的数据相结合等。例如，如果我们计算$(R \bowtie S) \bowtie T$，那么可以将R和S都传送到T所在节点，并在该节点上执行两个连接操作。

如果关系R由分布在若干节点上的片段$R_1, R_2, \cdots, R_n$构成，那么在选择逻辑查询计划时，我们还应该考虑用

$$R_1 \cup R_2 \cup \cdots \cup R_n$$

替代查询中使用的R。替代后的查询或许能让我们很大程度地简化表达式。例如，如果每个R_i代表19.4.1节中讨论的Sales关系的一个片段，而每个片段都对应于一个商店，那么在关于Boston商店销售额的查询中我们可以去除集合并中除Boston的片段外的所有R_i。

19.4.5 习题

*!! **习题19.4.1** 下面的习题使你能体会到在决定数据复制策略时将会遇到的一些问题。假设有n个节点都要访问关系R。对于$i = 1, 2, \cdots, n$，第i个节点每秒钟都提出关于R的查询q_i和对R的更新u_i。如果在提出查询的节点上有R的副本，那么查询执行的开销为c，否则开销为$10c$。在提出更新的节点上，更新R的副本的开销为d，而更新其他节点上每个副本的开销为$10d$。对于一个很大的n，你如何根据这些参数选择在什么样的节点集合上复制R。

1022

19.5 分布式提交

在本节中，我们将讨论在多个节点上有组成成分的分布式事务如何原子地执行的问题。下一节讨论分布式事务的另一个重要性质，即执行的可串行性。我们将以一个例子来说明可能出现的问题。

例19.17 考虑19.4节我们提到的连锁店的例子。假设连锁店的某个经理想要查询所有商店，找出所有店中牙刷的仓储情况，然后指示将牙刷从一些商店运到另一些商店，以平衡仓储。该操作由一个全局的事务T来完成；事务T在第i个商店有组成成分T_i，并且在该经理所在的机构有组成成分T_0。T执行的动作序列概括如下：

1. 成分T_0在经理所在节点上创建。
2. T_0向所有商店发送消息，指示它们创建成分T_i。
3. 每个T_i在商店i执行一个查询，找出该商店库存的牙刷数目，然后将这一数目报告给T_0。
4. T_0接收这些数，并根据某种算法决定需要运输的牙刷数量，使用的算法我们在此不做讨论。然后，T_0发送“商店10应该运输500支牙刷到商店7”这样的消息给对应的商店（本例中为商店7和商店10）。
5. 接收到指示的商店更新存货清单并送货。 □

19.5.1 分布式原子性的支持

在例19.17中，很多地方都可能出错，而其中的许多错误都会导致T的原子性受到破坏。也就是说，组成T的某些动作得以执行，而另一些动作却没有执行。我们假定在各个节点上都应该有的日志和恢复机制保证每个T_i的执行都是原子的，但它们不能保证T本身的原子性。

例19.18 假设重新分配牙刷的算法中有一个错误，导致要求商店10运输的牙刷数量比该商店的库存量还多。在这种情况下，T_{10}将中止，牙刷不会从商店10运出，且商店10的存货清单也不会更改。然而，商店7的T_7没有发现任何错误并提交，它修改了该商店的存货清单以反映假设应运输的牙刷数量。现在，T不仅不能原子地执行（因为T_{10}永远也不能完成），还使分布式数据库处于不一致的状态；存货清单上的牙刷数量与实际的数量不等。 □

1023

导致问题的另一种情况是分布式事务执行过程中某个节点可能发生故障或与网络断连。

例19.19 假设在响应T_0的第一条消息时，T_{10}告诉T_0自己的牙刷存货量。但是，商店10的机器在此后崩溃，T_0发出的指示就再也不能被T_{10}收到。分布式事务T能提交吗？T_{10}在其所在节点恢复时应该做什么？ □

19.5.2 两阶段提交

为了避免19.5.1节中所提到的那些问题，分布式DBMS在决定是否提交一个分布式事务时使用很复杂的协议。在本节中，我们将描述这些协议所基于的基本思想，也就是两阶段提交[⊖]。通过做出关于提交的全局决定，事务的成分要么都提交，要么都不提交。和往常一样，我们假设每个节点上的原子性机制保证该节点上的局部事务成分要么提交，要么对该节点上的数据库状态不产生任何影响；也就是说，事务的各成分都是原子的。因此，通过实行分布式事务的所有成分要么都提交要么都不提交这一规则，我们可以保证分布式事务本身的原子性。

下面是关于两阶段提交的几个要点：

- 在两阶段提交中，我们假设每个节点记录该节点上动作的日志，但没有全局的日志。
- 我们还假设一个称为协调器的节点在决定分布式事务是否提交中扮演特殊的角色。例如，协调器可能是发起事务的节点，比方说19.5.1节的例子中的节点T_0。
- 两阶段提交协议涉及协调器和其他节点之间的消息发送。在发送每条消息时，发送节点都将把该消息记录到日志中，以便在需要恢复时提供帮助。

记住这些要点后，我们可以按照节点之间发送的消息来描述两阶段提交。

阶段I :

在两阶段提交的第一个阶段中，分布式事务*T*的协调器决定何时试图提交*T*。大概在位于协调器节点上的事务*T*的成分准备好提交后开始试图提交，但原则上即使协调器的成分希望中止， 1024
各个步骤仍然要执行（然而正如我们将看到的那样，这些步骤有了极大的简化）。协调器询问包含事务*T*的成分的所有节点，以确定这些节点希望提交还是中止。

1. 协调器在所在节点的日志中放入<Prepare *T*>日志记录。
2. 协调器向成分所在的各个节点（原则上包含它自己）发送消息prepare *T*。
3. 每个接收到消息prepare *T*的节点决定该节点上*T*的成分是提交还是中止。如果事务成分尚未完成其行动，节点可以推迟发送响应的消息，但终究是要发送的。
4. 如果节点希望提交其事务成分，就必须进入一个称为预提交的状态。一旦进入预提交状态，节点就不能中止其上的*T*的成分，除非协调器指示中止事务。进入预提交状态必须执行如下步骤：

   (a) 执行为保证*T*的局部成分再没有必要中止而必须的所有步骤，使即便该节点上发生故障然后又恢复的情况下此事务成分也不需中止。因此，不仅与局部*T*相关的所有动作都必须执行，还需要执行与日志相关的适当动作，以保证恢复时*T*将被重做而不是被撤销。所需动作依赖于日志方式，但显然与局部*T*的动作相关的日志记录必须刷新到磁盘上。

   (b) 在局部日志中放入记录<Ready *T*>,并将日志刷新到磁盘。

   (c) 向协调器发送ready *T*消息。

   但是，该节点在这时并未提交其上的*T*的成分；它必须等待第二个阶段。
5. 如果情况正好相反，节点希望中止其*T*的成分，那么它记载日志记录<Don't commit *T*>，并向协调器发送消息don't commit *T*。这时中止事务成分是安全的，因为即使只有一个节点希望中止，事务*T*也肯定会中止。

图19-16概括了第一阶段中的消息。

⊖ 不要混淆两阶段提交和两阶段封锁。它们是相互独立的概念，用来解决不同的问题。

阶段II

第二阶段从协调器收到来自各节点的ready或don't commit响应开始。然而，有可能某些节点不能进行响应；这样的节点可能是停机了，也可能已经与网络断连。在这种情况下，协调器会在某个合适的超时期间后将这样的节点视为发送了don't commit来处理。

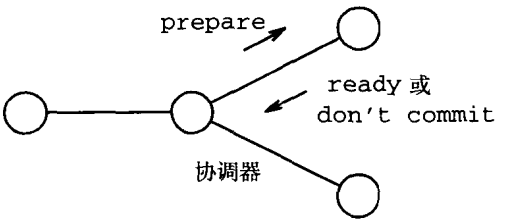


图19-16 两阶段提交的第一阶段中的消息

1. 如果协调器从*T*的所有成分接收到的都是ready *T*，那么它就决定提交*T*。协调器

(a) 在其节点中记载<Commit *T*>日志记录；并且

(b) 向参与*T* 的所有节点发送commit *T* 消息。

2. 如果协调器收到来自一个或多个节点的don't commit *T*消息，那么它

(a) 在其节点中记录<Abort *T*>日志记录；并且

(b) 向参与*T*的所有节点发送abort *T*消息。

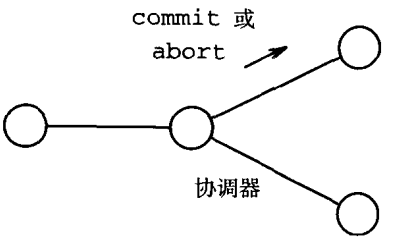


图19-17 两阶段提交的第二阶段中的消息

3. 如果节点收到commit *T*消息，该节点将提交其上的*T*的成分，并在此过程中记录日志<Commit *T*>。

4. 如果节点收到abort *T*消息，该节点将中止*T*，并写入日志记录<Abort *T*>。

图19-17概括了第二阶段中的消息。

19.5.3 分布式事务的恢复

在两阶段提交过程中的任何时候，节点都可能发生故障。我们需要保证在该故障节点恢复时发生的一切都和分布式事务*T*的全局决定一致。根据*T*最后一个日志项的不同，有几种情况需要考虑：

1. 如果*T*的最后一个日志记录是<Commit *T*>，那么协调器必然已经提交*T*。根据所使用的日志方式，在恢复节点上可能需要重做该节点上的*T*成分。

2. 如果最后一个日志记录是<Abort *T*>，那么类似地我们知道全局的决定是中止*T*。根据所使用的日志方式，在恢复节点上可能需要撤销该节点上的*T*的成分。

3. 如果最后一个日志记录是<Don't comit *T*>，那么该节点知道全局的决定必然是中止*T*。*T*对局部数据库的影响可能需要撤销。

4. 比较难的情况是*T*的最后一个日志记录是<Ready *T*>。现在，正在恢复的节点并不知道全局的决定是提交*T*还是中止*T*。该节点必须和至少一个其他的节点交流，以找出关于*T*的全局决定。如果协调器在工作，那么该节点可以询问协调器。如果这时协调器未工作，那么可以要求其他的某个节点查看日志以找出*T*的结局。在最坏的情况下，恢复节点与其他所有节点都联系不上，这时*T*的局部成分必须保持活跃，直到确定关于*T*的决定是提交还是中止。

5. 局部日志中也可能没有关于*T*在两阶段提交协议中的动作的记录。如果这样，那么恢复节点可以单方面地决定中止其*T*成分，这和所有日志方式都是一致的。有可能协调器已经检测到该故障节点超时，并决定中止*T*。如果故障持续时间很短，那么*T*在别的节点上仍可能是活跃的；但是，如果恢复节点决定中止其*T*成分，并稍后在第一阶段中被询问时用don't commit

T回答，不一致的情况肯定不会发生。

上面的分析中假设故障节点不是协调器。如果协调器在两阶段提交中发生故障，就会产生一些新的问题。首先，余留的参与节点要么等待协调器恢复，要么选举一个新的协调器。由于原协调器可能无限期地停止工作，因此最好选举一个新的领导，至少在一个较短的等待时间以观察协调器能否恢复后应重新选举。

*领导选举*这个问题本身就是分布式系统中一个极其复杂的问题，不在本书所要讨论的范围内。然而，有一种简单的方法在大多数情况下都可以使用。例如，我们可以假设所有参与节点都有用作惟一标识的编号；许多情况下可以使用IP地址。所有参与者都向其他节点发送消息，宣称自己可以作为领导，并给出自己的标识编号。在一段适当的时间后，所有参与者都将自己所听到的编号最低的节点作为新的协调器，并将关于这一结果的消息发送给其他的所有节点。如果所有节点接收到一致的消息，那么新协调器就有了惟一的选择，并且每个节点都知道这个节点是哪一个。如果有不一致，或者某个余留的节点未做出回答，这也会被所有节点知道，而 1027
选举再次开始。

现在，新的领导询问各个节点关于每个分布式事务T的信息。如果在节点的日志中有关于T的记录，那么节点就报告其中的最后一条。可能出现的情况有：

1. 某个节点的日志中有<Commit T>记录，那么原协调器必然已经试图向所有节点发送commit T消息，因而将T提交是安全的。

2. 类似地，如果某个节点的日志中有<Abort T>记录，那么原协调器必然已经决定中止T，因而新协调器下令中止T是安全的。

3. 假设现在所有节点上都没有<Commit T>或<Abort T>记录，但至少一个节点的日志中没有<Ready T>。那么，由于记录日志发生在发送对应消息之前，我们知道原协调器不可能已收到这一节点的ready T消息，因而不可能已经决定提交T。对新协调器来说，决定中止T是安全的。

4. 最困难的情况是找不到<Commit T>和<Abort T>，但每个余留的节点上都有<Ready T>。现在，我们不能确定原协调器是否已发现中止或不中止T的原因；例如，它可能由于自身所在节点上的动作而决定中止T，也可能由于接收到另一故障节点的don't commit T消息而决定中止T。原协调器也有可能已经决定提交T，并已经提交其T的局部成分。因此，新协调器不能决定是提交T还是中止T，而必须等待原协调器恢复。在实际的系统中，数据库管理员可以对此进行干涉，并手工强制所有等待中的事务成分完成。其结果是可能破坏原子性，但执行被阻塞事务的人将得到通知，可以采取一些适当的补救措施。

19.5.4 习题

! **习题19.5.1** 考虑在一台家用计算机上发起的一个事务T，该事务请求银行B从该行某个账户中的10 000美元转入银行C中的另一账户。

* a) 分布式事务T的成分有哪些？位于B和C的事务成分各需要做什么？

b) 如果B的账户中没有10 000美元，那么会出现什么错误？

c) 如果一个银行或两个银行的计算机崩溃或者网络断连，那么会发生什么错误？ 1028

d) 如果(c)中提出的问题有一个发生了，那么在计算机和网络正常工作后事务怎样正确地继续下去？

习题19.5.2 在本习题中，我们需要一种描述两阶段提交中可能发生的消息序列的记法。设（i，j，M）表示节点i向节点j发送消息M，其中M的值及其含义可以是P（预提交）、R

（准备好提交）、*D*（不要提交）、*C*（提交）和*A*（中止）。我们将讨论一种简单的情形，其中节点0是协调器，但除此外它不再担任事务中的任何工作；节点1和节点2是事务成分。例如，下面是在事务成功提交过程中可能发生的消息序列：

$$(0, 1, P), (0, 2, P), (2, 0, R), (1, 0, R), (0, 2, C), (0, 1, C)$$

* a) 举出节点1希望提交而节点2希望中止时可能发生的消息序列的一个例子。

*! b) 如果事务成功提交，像上面这样可能发生的消息序列有多少？

! c) 假设不发生故障，如果节点1希望提交而节点2不希望，那么有多少可能的消息序列？

! d) 如果节点1希望提交，但节点2停止工作且没有对消息做出响应，那么有多少可能的消息序列？

!! **习题19.5.3** 使用习题19.5.2中的记法，假设节点包括一个协调器和另外n个作为事务成分的节点。作为n的函数，事务成功提交时可能的消息序列有多少？

19.6 分布式封锁

在本节中，我们将讨论如何把封锁调度器扩展到事务是分布的且由若干节点上的成分构成时的情况。假设锁表由各个节点管理，并且各个节点上的事务成分只能申请该节点中数据元素上的锁。

当数据被复制时，我们必须设法使每个事务对同一元素*X*的所有副本的改变以同样的方式进行。这一要求引入了封锁逻辑数据库元素*X*与封锁*X*的一个或多个副本的差别。在本节中，我们将提供一种适用于有副本数据和无副本数据的分布式封锁算法代价模型。但是，在介绍这一模型前，我们先考虑解决分布式数据库中锁的维护问题的一个显而易见的（且有时能解决问题的）方法——集中封锁。

1029

19.6.1 集中封锁系统

最简单的方法可能是指定一个封锁节点来维护一张逻辑元素的锁表，而不管这些元素在该节点上是否有副本。当事务希望获得逻辑元素上的锁时，它向该封锁节点发送一个请求，而该节点根据实际情况授予或拒绝。由于获得*X*上的全局锁等同于在该封锁节点获得*X*上的局部锁，因此只要该封锁节点按照传统的方式来管理锁，我们就可以保证全局封锁行为的正确性。除了事务恰好在封锁阶段上运行外，一般情况下每个锁的开销为三次消息传递（请求、授予和释放）。

在某些情况下，使用单一封锁节点就足够了，但如果节点和并发事务很多，那么封锁节点可能成为瓶颈。此外，如果封锁节点崩溃，那么其他任何节点都不能获得锁。由于集中封锁存在这些问题，维护分布式封锁可以用一些别的方法，我们将在讨论如何估计封锁代价之后介绍这些方法。

19.6.2 分布式封锁算法的代价模型

假设每个数据元素正好在一个节点上存在（即不进行数据复制），并且每个节点上的封锁管理器存储该节点上的锁和锁请求。事务可以是分布式的，一个事务可能由一个或多个节点上的成分组成。

尽管与封锁管理相关的代价是多方面的，但其中有很多都是固定的，与事务在网络上请求封锁的方式无关。我们能够控制的一个代价因素是事务获得和释放锁时节点之间传送的消息数。因此，我们将在假设所有封锁请求都被同意的前提下计算各种封锁模式需要的消息数。当然，

封锁请求可能被拒绝，这将导致拒绝请求的一条额外消息以及后来授予锁时的一条消息。但是，由于我们不能预知拒绝封锁的频率，并且这一频率不是我们所能控制的东西，所以在比较中将忽略这一额外的消息需求。

例19.20 正如我们在19.6.1节提到的那样，在集中封锁方式下，一般的封锁请求需要三条消息：一条消息请求锁，一条来自中央节点的消息授予锁，而第三条消息释放锁。例外的情况是：

1. 当请求封锁的节点就是中央封锁节点时，这些消息是不必要的；并且
2. 当最初的封锁不能被同意时，需要发送额外的消息。

但是，我们假设这两种情况都较少见；即大多数封锁请求来自中央封锁节点以外的节点，并且大多数封锁请求都能被授予。因此，在中央封锁方式下，每个锁需要三条消息是一个比较好的估计。 □ 1030

现在考虑一种比集中封锁更灵活的情况，每个数据库元素在自己的节点上维护自己的锁。由于希望封锁X的事务在X所在节点上将有成分，看起来似乎不需要在节点之间传输消息。然而，如果分布式事务需要多个元素上的锁，例如X、Y和Z，那么事务只有在这三个元素上的锁都获得后才能完成。如果X、Y和Z位于不同节点上，那么这些节点上的事务成分至少需要交换同步消息以防止事务“超过自己”。

我们并不打算讨论所有可能的变体，而只讨论事务搜集锁的一种简单模型。我们假设每个事务的一个成分即*封锁协调器*负责搜集所有事务成分需要的所有锁。封锁协调器封锁自己所在节点上的元素不需消息，但封锁其他任何节点上的元素X需要三条消息：

1. 发向X所在节点以请求封锁的消息。
2. 授予锁的回答消息（回忆一下，我们假设所有锁都立即被授予；否则需要一条拒绝消息以及稍后的一条授予消息）。
3. 发向X所在节点以释放锁的消息。

由于我们只想对分布式封锁协议进行比较，而不打算给出这些协议各自平均消息数的绝对值，这样的简化可以解决我们的问题。

如果挑选事务需要的锁最多的节点作为封锁协调器，那么我们就能使对消息的需求极小化。所需消息数是其他节点上数据库元素的三倍。

19.6.3 封锁多副本的元素

当数据库元素X在多个节点上有副本时，我们必须小心处理X的封锁。

例19.21 假设数据库元素X有两个副本X_1和X_2，并假设事务T在X_1所在节点上获得了X_1上的共享锁，而事务U在X_2所在节点上获得了X_2上的排他锁。现在，U可以修改X_2，但不能修改X_1，这导致元素X的两个副本变得不同。此外，由于T和U还可能封锁其他元素，且这两个事务读写X的顺序不受它们在X的副本上的锁约束，因此T和U还可能产生不可串行化的行为。 □ 1031

例19.21说明的问题是，当数据被复制时，我们必须区分在逻辑元素X上获得共享锁或排他锁与在X的某个副本所在节点上获得该副本的局部锁。也就是说，为了保证可串行性，我们要求事务获得逻辑元素上的全局锁。但是，逻辑元素在物理上并不存在（只有其副本存在），并且没有全局的锁表。因此，事务获得X上的全局锁的惟一办法是，事务在X的一个或多个副本所在节点上获得这些副本的局部锁。我们现在考虑将局部锁转变为具有所需性质的全局锁的方法：

- 任意两个事务不能同时获得逻辑元素X上的全局排他锁。
- 如果一个事务在逻辑元素X上拥有全局排他锁，那么任何事务都不能获得X上的全局共

享锁。

• 只要任何事务都没有X上的全局排他锁，就可以有任意多个事务拥有X上的全局共享锁。

19.6.4 主副本封锁

集中封锁方式的一种改进是把封锁节点的功能分散，但每个逻辑元素的封锁只由一个节点负责这一原则仍保持不变。这样的分布封锁方式称为主副本方式。这一改变避免了中央封锁节点成为瓶颈的可能性，但保持了集中封锁方式的简单性。

在主副本封锁方式中，每个逻辑元素X的多个副本中有一个被指定为X的“主副本”。为了获得逻辑元素X上的锁，事务向X的主副本所在节点发送一个请求。主副本所在节点在其锁表中维护关于X的一项，并根据实际情况同意或拒绝这一请求。和前面相似，只要每个节点正确管理主副本的封锁，那么全局（逻辑）封锁就能得到正确管理。

和集中封锁节点类似的另一个地方是，除了事务与主副本位于同一节点的情况外，大多数封锁请求产生三条消息。但是，如果我们精明地选择主副本，那么可以认为事务和主副本经常位于同一节点。

例19.22 在连锁店的例子中，我们让每个商店存放该商店销售数据的主副本，而这一数据的其他副本（例如位于中央机构的副本或数据仓库中用于销售分析的副本）都不是主副本。
1032 通常的事务很可能在一个商店执行，并且只更新该商店的销售数据。这类事务获得锁时不需要消息。只有在事务检查或修改其他商店的数据时才需要传送与封锁相关的消息。 □

19.6.5 局部锁构成的全局锁

另一种方法是用局部锁的集合合成全局封锁。在这样的模式中，数据库元素X没有“主副本”；X的所有副本是对称的，在这些副本中的任一个上都可以申请共享锁或排他锁。成功的全局封锁模式的关键在于，事务只有获得了一定数量的X副本上的锁后才能假设自己获得了X上的全局锁。

假设数据库元素A有n个副本。我们选择两个数：

1. s是事务获得A上的全局共享锁时必须以共享方式封锁的A的副本数。
2. x是事务获得A上的排他锁时必须以排他方式封锁的A的副本数。

只要$2x>n$且$s+x>n$，我们就能获得所需的性质：A上只能有一个排他锁，A上不能既有一个排他锁又有一个共享锁。解释如下。由于$2x>n$，如果两个事务都有X上的全局排他锁，那么至少有一个副本为这两个事务授予了局部排他锁(因为所授予的局部排他锁数大于副本数)。但是如果这样的话，那么局部封锁方式就是错误的。类似地，由于$s+x>n$，如果一个事务在A上有全局共享锁而另一事务在A上有全局排他锁，那么必然有某个副本同时授予了局部共享锁和排他锁。

分布式死锁

在事务试图获得有副本的数据上的全局锁时，事务可能陷于死锁的情况很多。构造一个全局等待图并检测死锁的方法也很多。但是，在分布式环境中，使用超时通常更简单也更有效。所有在某个适当的时间内未完成的事务都被认为是已经死锁并被回滚。

一般情况下，获得全局共享锁所需的消息数为$3s$，获得全局排他锁所需的消息数为$3x$。与集中封锁方式平均每个锁需要3条甚至更少的消息相比，这个数字看起来似乎比较大。但是，
1033 正如下面两个针对某个（s，x）选择的例子所说明的那样，有的参数能起补偿作用。

• 读封锁一个；写封锁所有。这里，$s = 1$且$x = n$。获得全局排他锁代价很高，但全局共享

锁至多需要三条消息。此外，这一方式有一个胜过主副本方式的地方：尽管后者允许我们避免读主副本时的消息，但“读封锁一个”方式则只要事务在需要读的数据库元素的任一副本所在节点上就能够避免消息。因此，当大多数事务是只读事务但读元素X的事务在不同节点上发起时，这一方式比较优越。一个例子是，分布式数字图书馆在文档最经常访问的地方缓存相应文档的副本。

- 大多数封锁。这里，$s=x=\lceil (n+1)/2 \rceil$。看起来似乎不管事务在哪里，这一系统都需要大量的消息。但是，下面的几个因素使这一方式受到欢迎。首先，许多网络系统支持广播，这使事务可以发出请求元素X上局部锁的一条总的消息，而这条消息将被所有节点收到。类似地，锁的释放也可以用一条消息来实现。此外，s和x的这一选择能提供别的方式所不能提供的好处：即使在网络断连的情况下这一方式也能允许部分的操作。只要网络成分中包含X的大多数副本所在的节点，事务就有可能获得X上的锁。即使其他节点在断连时是活跃的，我们知道它们甚至不能获得X上的共享锁，因此运行在不同网络成分中的事务不可能产生不可串行化的行为。

19.6.6 习题

! **习题19.6.1** 我们给出了如何用局部的共享锁和排他锁分别创建全局的共享锁和排他锁。你怎样用相应类型的局部锁创建：

* a)全局的共享锁、排他锁以及增量锁。

b) 全局的共享锁、排他锁以及更新锁。

!! c) 全局的共享锁、排他锁以及每种类型的意向锁。

习题19.6.2 假设有五个节点，每个节点上有数据库元素X的一个副本。这些节点中有一个节点P对X而言占据统治地位，在主副本分布式封锁系统中将被用作X的主节点。关于对X的访问的统计数据为：

a) 所有访问中有50%是P发起的只读访问。 1034

b)其他四个节点各发起10%的访问，这些访问都是只读的。

c) 其余10%的访问需要排他的访问，而五个节点上发起这种访问的概率相等（即每个节点上发起2%）。

对于下列每种封锁方式，给出获得一个锁所需要的平均消息数。假设所有请求都被同意，因而不需要拒绝的消息。

* a)读封锁一个；写封锁所有。

b) 大多数封锁。

c) 主副本封锁，并设主副本位于P。

19.7 长事务

有这样一类应用，它们的数据可以用数据库管理系统来管理，但是作为数据库并发控制机制基础的多个短事务的模型对它们不适合。本节中我们将讨论一些这类应用的例子和所引起的问题。然后，我们将讨论一个基于“补偿事务”的解决方法，用来取消已经提交但不应该提交的事务的影响。

19.7.1 长事务的问题

大致说来，长事务是需要太长时间因而不允许它们保持其他事务所需要的锁的事务。根据环境，“太长”可以是若干秒、分或者小时。我们将假设“长”事务至少要几分钟，也可能几

小时。可能出现长事务的三大类应用为：

1. 传统的DBMS应用。尽管通常的数据库应用主要运行短事务，但许多应用偶尔需要长事务。例如，一个事务可能检查银行的所有账户以确定总的余额是正确的。另一个事务可能要求偶尔地重构索引以保持效率最高。

2. 设计系统。不管设计的东西是机械的（如汽车）、电的（如微处理器）还是软件系统，设计系统共同的一个要素是，设计被划分为一系列成分（例如，软件项目的文件），且不同的
1035 设计者同时工作在不同的成分上。我们不希望两个设计者各自获得文件的一个副本，通过编辑各自的副本来修改设计，然后将新的文件版本写回，因为这样将导致一组修改被另一组修改覆盖。因此，检出检入系统使用户能“检出”文件，并在修改文件后“检入”该文件，而这可能是在若干小时或若干天后。即使第一个设计者正在修改文件，其他设计者仍可能希望阅读该文件，以获得关于其内容的信息。如果检出操作相当于排他锁，那么一些合理的、切合实际的动作就可能被延迟，可能长达数天。

3. 工作流系统。这样的系统涉及过程集合，有的过程由软件单独执行，有的过程需要人的交互，而有的过程可能只涉及人的活动。我们马上要给出办公室中报销时所需文书工作的例子。这样的应用可能需要执行很多天，并且在此期间某些数据库元素可能会改变。如果系统为事务中涉及的数据授排他锁，那么其他事务在很多天内都被锁在外面。

例19.23 考虑职员报销差旅费的问题。该职员希望从账户A123中获得相应的补偿，付账的过程如图19-18所示。这一过程从动作A_1开始，即出差者的秘书在线地填写一张表格，描述旅途、付账的账户以及金额。我们假设在这个例子中账户为A123而金额为1000美元。

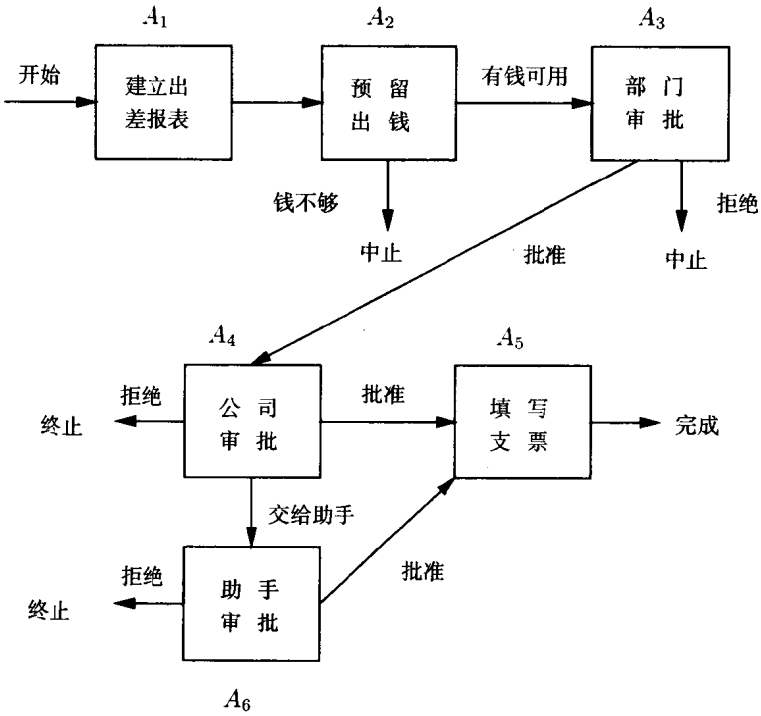


图19-18 出差者请求报销差旅费的工作流图

该职员的收据在现实中被送到部门的相应授权机构，而表格则在线传送到一个自动的动作A_2。这一过程检查付款账户A123中是否有足够的钱，并为这笔费用预留出钱来；也就是说，

这一过程尝试从账户中减去1000美元，但并不发出具有此金额的支票。如果该账户中没有足够的钱，则此事务中止，并假定它会在账户中金额足够或改变付款账户后重启[⊖]。

动作A_3由部门主管执行，部门主管检查收据和在线表格。这一动作可能发生在下一天。如果一切正常，部门主管就批准该表格，并将表格和实际的收据一起送给公司主管；否则事务中止。或许出差者需要对申请做某些修改并重新提交表格。

动作A_4可能发生在若干天以后。在这一动作中，公司主管或者批准该申请，或者拒绝，或者将表格交给一个助手去处理，而此助手将在动作A_5中作出决定。如果表格被拒绝，事务又需要终止，表格也需要重新提交。如果表格被批准，那么动作A_6中将书写支票并完成从账户A123中减去1000美元的操作。 1036

但是，假设我们只能用传统的封锁方法来实现这一工作流。特别地，由于账户A123的余额可能由整个事务修改，该账户必须在动作A_2时以排他方式封锁，并且要等到事务中止或动作A_6完成后才能释放。这个锁可能需要保持若干天，而在这段时间内只有负责批准报销的人能够查看相应的情况。如果这样，那么其他费用就不能使用账户A123，即使是尝试性的也不允许。另一方面，如果对账户A123的访问根本不加以控制，那么可能有多个事务同时从该账户中预留和减去一些金额，因而导致透支。因此，我们必须在严格的长期封锁和无控制这两个极端之间进行折中。 □

19.7.2 saga（系列记载）

*saga*是构成长“事务”的一系列动作，例如例19.23中的那些动作。也就是说，saga包括：

1. 一系列动作。

2. 一个图，其结点是动作或特殊的终止及完成结点，而弧连接结点对。不存在从两个特殊结点中发出的弧，这两个特殊结点称为终止结点。 1037

3. 关于动作从哪个结点开始的指示，这一结点称为*开始*结点。

图中从开始结点到终止结点之间的路径表示可能的动作序列。通向中止结点的路径表示导致整个事务回滚的动作序列，这些动作序列不应改变数据库。通向完成结点的路径表示成功的动作序列，这些动作对数据库所做的改变都将保留在数据库中。

例19.24 图19-18的图中通向中止结点的路径是$A_1\ A_2$、$A_1\ A_2\ A_3$、$A_1\ A_2\ A_3\ A_4$和$A_1\ A_2\ A_3\ A_4\ A_5$。通向完成结点的路径是$A_1\ A_2\ A_3\ A_4\ A_6$和$A_1\ A_2\ A_3\ A_4\ A_5\ A_6$。注意，在这个例子里，图中没有环，因此通向终止结点的路径数是有限的。但是，通常情况下图中可能有环，这种情况下路径数可能是无限的。 □

saga的并发控制通过两方面的能力来管理：

1. 可以认为每个动作自身是一个（短）事务，在执行时使用传统的并发控制机制，如封锁。例如，A_2可以实现为在账户A123上（短暂地）获得锁，减去差旅费单据上的金额，然后释放锁。这样的封锁可以防止两个事务同时为账户余额写入新值，并因而丢失第一个写操作的结果，使钱“魔术般地出现”。

2. 整个事务即任何通向终止结点的路径通过“补偿事务”机制来管理，“补偿事务”是saga在各个结点上的事务的逆。它们的工作是回滚已提交的动作，回滚方式不依赖于在该动作执行时刻和补偿事务执行时刻之间数据库上发生了什么。我们将在下一节讨论补偿事务。

⊖ 当然，出差者（肯定不在斯坦福工作）不会不恰当地使用另一个政府合同，而将使用某个恰当的经费来源。我们必须要提到这一点是因为，斯坦福中到处都有许多政府审计员，尽管他们并不懂大学应如何运作。

19.7.3 补偿事务

在saga中，每个动作A都有一个补偿事务，我们记为A^{-1}。直观地说，如果我们执行A，后来又执行A^{-1}，那么所产生的数据库状态同A和A^{-1}都未执行前一样。更形式化地表示：

- 如果D是任一数据库状态，$B_1B_2\cdots B_n$是动作和补偿事务的任一序列（不管是来自所讨论的saga还是来自在数据库上合法执行的任何其他的saga或事务），那么在数据库状态D

1038 上运行序列$B_1B_2\cdots B_n$和$AB_1B_2\cdots B_nA^{-1}$所产生的数据库状态一样。

数据库状态什么时候“一样”

在讨论补偿事务时，我们必须小心对待使数据库回复到和以前“一样”的状态的含义。当我们在例19.8中讨论B树的逻辑日志时已经领略过这个问题。在那里我们看到，如果我们“撤销”一个操作，B树的状态可能和执行该操纵以前不完全相同，但就B树上的访问操作而言，它们是等价的。更一般地讲，执行一个动作及其补偿事务或许不能将数据库恢复到与以前完全相同的状态，但它们的差异对数据库所支持的应用来说应该是不能察觉的。

如果saga的执行通向中止结点，那么我们通过为每个已执行的动作执行补偿事务来回滚该saga，补偿事务执行的顺序与对应动作执行的顺序相反。根据上面描述的补偿事务的性质，该saga的影响被消除，而数据库状态就和saga没有发生一样。19.7.4节中将解释为什么能保证消除影响。

例19.25 我们来考虑图19-18中的动作，看一看A_1~A_6的补偿事务各是什么。首先，A_1创建一个在线文档。如果该文档存储在数据库中，那么A_1^{-1}必须将其从数据库中删除。注意，这一补偿遵循补偿事务的基本性质：如果我们创建文档，执行任意动作序列α（如果我们愿意，其中也可以包括删除该文档），那么$A_1\,\alpha\,A_1^{-1}$的效果和α的效果一样。

A_2的实现必须非常小心。我们通过从账户中减去相应金额来“预留”钱。这些钱一直保持被消除的状态，除非补偿事务A_2^{-1}对其进行恢复。如果通常管理账户的规则得到遵循，那么我们就说该A_2^{-1}是正确的补偿事务。为了理解这一点，有必要考虑一个类似的事务，对这个事务来说，明显的补偿不能起到应有的作用；我们稍后在例19.26中考虑这样的一个例子。

动作A_3、A_4和A_5都包括在表格上添加批准意见，因此它们的补偿事务可以将批准意见删除⊖。

最后，动作A_6写支票，这个动作没有明显的补偿事务。实际上也不需要，因为一旦A_6被执

1039 行，这个saga就不能回滚了。但是，从技术上来说，A_6根本不影响数据库，因为支票金额由A_2减去。如果需要在更宽的范围内考虑“数据库”，这时像兑现支票这样的结果将影响数据库，那么我们必须将A_6^{-1}设计为首先取消支票，然后写封信给领款人要求归还钱；如果所有补救措施都失败，则恢复账户中的金额，并申报由于呆帐而造成的损失。 □

下面我们继续看例19.25中隐含的例子，其中对账户的改变不能通过一个逆向的改变来补偿。问题在于账户通常不允许为负。

例19.26 假设事务B为一个账户中增加1000美元，该账户原来有2000美元，而B^{-1}是消除同样数额的钱的补偿事务。此外，假设在试图从账户中消除钱而这将导致余额为负时事务失败是合理的。设C是从同一账户中消除2500美元的事务，那么$BCB^{-1}\neq C$。原因在于只执行C时会失

⊖ 在图19-18的saga中，补偿这些动作的惟一时机是当我们将删除表格时，但补偿事务的定义要求它们各自独立地工作，不管其他补偿事务是否可能使其所做的改变变得无关紧要。

败，账户中保持为2000美元，而如果先执行B再执行C，则账户余额为500美元，这时再执行B^{-1}就会失败。

我们的结论是，只使用补偿事务不能同时支持在账户间任意转账的saga和不允许账户余额为负的规则。系统必须进行某些修改，例如允许账户中出现负的余额。

19.7.4 补偿事务发挥作用的原因

如果两个动作序列将任意数据库状态D转换为同一状态，那么我们说这两个序列是等价的（$\equiv$）。补偿事务的基本假设可以表述为：

- 如果A是任一动作，α是合法动作和补偿事务任一序列，那么$A\alpha A^{-1}\equiv\alpha$。

现在，我们需要证明，如果执行$A_1A_2\cdots A_n$这一saga后，再以相反的次序执行它们的补偿事务$A_n^{-1}\cdots A_2^{-1}A_1^{-1}$，其间无论有什么样的动作介入，执行效果与动作和补偿事务都没有执行过的一样。证明的方法是对n进行归纳。

基础：如果$n=1$，那么A_1和其补偿事务A_1^{-1}之间的动作序列形如$A_1\,\alpha\,A_1^{-1}$。根据补偿事务的基本假设，$A_1\alpha A_1^{-1}\equiv\alpha$；即此saga对数据库状态没有影响。

归纳：假设这一陈述对长度不超过$n-1$个动作的路径成立，考虑一条n个动作的路径，其后跟随着反序的补偿事务，中间可以有任何事务介入。此序列的形式为

$$A_1\alpha_1A_2\alpha_2\cdots\alpha_{n-1}A_n\beta A_n^{-1}\gamma_{n-1}\cdots\gamma_2A_2^{-1}\gamma_1A_1^{-1} \qquad (19\text{-}1)$$

1040

其中所有希腊字母都表示零个或多个动作的序列。根据补偿事务的定义，$A_n\,\beta\,A_n^{-1}\equiv\beta$。因此（19-1）等价于

$$A_1\alpha_1A_2\alpha_2\cdots A_{n-1}\alpha_{n-1}\beta\gamma_{n-1}A_{n-1}^{-1}\gamma_{n-2}\cdots\gamma_2A_2^{-1}\gamma_1A_1^{-1} \qquad (19\text{-}2)$$

因为在（19-2）式中只有$n-1$个动作，根据归纳假设，（19-2）式等价于

$$\alpha_1\alpha_2\cdots\alpha_{n-1}\beta\gamma_{n-1}\cdots\gamma_2\gamma_1$$

也就是说，该saga及其补偿事务使数据库状态跟saga从未发生一样。

19.7.5 习题

*! **习题19.7.1** “卸载”软件的过程可以被认为是安装该软件的动作的补偿事务。在安装和卸载的一个简单模型中，假设动作包括将一个或多个文件从源（如CD-ROM）装载到机器的硬盘上。为了装载文件f，我们从CD-ROM上复制f，如果已经存在路径名相同的f，则取代原有文件。为了区别路径名相同的文件，我们可以假设每个文件有一个时间戳。

a) 装载文件f这一动作的补偿事务是什么？考虑两种情况，即不存在相同路径名的文件和存在相同路径名的文件f'这两种情况。

b) 解释为什么你在(a)中的答案能保证补偿。提示：仔细考虑用f取代f'后又用具有相同路径名的另一文件取代f的情况。

! **习题19.7.2** 用saga描述预订航班座位的过程。考虑顾客可能查询座位但并不预订。顾客可能预订座位，后来却又取消，或者在规定的期限内没有付款。顾客可能乘坐预订的航班也可能不乘坐。对于每个动作，描述相应的补偿事务。

19.8 小结

- 脏数据：由未提交事务写入主存缓冲区或硬盘上的数据称为“脏”数据。

- 级联回滚：允许事务读脏数据的日志和并发控制组合可能需要回滚那些从稍后中止的事务中读取了数据的事务。 1041
- 严格封锁：严格封锁策略要求事务一直持有锁（除共享锁外），直到事务提交且日志中的提交记录已刷新到磁盘。严格封锁保证事务不会读到脏数据，即使崩溃和恢复后回头再看时也如此。
- 成组提交：如果确信日志记录到达磁盘的顺序和写入顺序一样，那么我们可以放松严格封锁中要求提交记录到达磁盘这一条件。这时仍能保证不存在脏读，即使有崩溃和恢复发生。
- 在中止后恢复数据库状态：如果事务中止但已经往缓冲区中写入了值，那么我们可以用日志或磁盘上的数据库拷贝恢复原来的值。如果新值已到达磁盘，那么我们仍可以用日志来恢复旧值。
- 逻辑日志：对于大的数据库元素如磁盘块来说，如果在日志中增量地记录新值和旧值，即只表示变化，那么我们可以节省许多空间。在某些情况下，通过块中所含内容的抽象逻辑地记录改变，这使我们在事务中止后可以逻辑地恢复状态，即使数据库状态不能精确恢复。
- 视图可串行性：如果事务写入的值在没有被读取的情况下就被覆盖，那么对调度来说冲突可串行性这一条件过于严格。视图可串行性是一个较弱的条件，只要求在等价串行调度中每个事务读取的值的来源和原始调度中一样。
- 多重图：判断视图可串行性需要构造多重图，图中的弧表示值从写事务到读事务的传递，而弧对表示写事务不能介入写事务-读事务联系中。调度是视图可串行化的，当且仅当从每个弧对中选出一条弧能得到无环图。
- 死锁：当事务必须等待另一事务占有的资源例如锁时可能发生死锁。危险在于，如果没有适当的规划，就会发生循环等待，而环中的事务都不能取得进展。
- 等待图：为每个等待中的事务创建一个结点，结点上有一条指向该事务所等待的事务的弧。存在死锁等同于等待图中存在一个或多个环。如果我们维护等待图，并中止将导致死锁的等待事务中的任一个，那么可以避免死锁。
- 通过资源排序避免死锁：要求事务在获得资源时必须按照资源的某种字典顺序，这可以防止死锁的产生。 1042
- 基于时间戳的死锁避免：另一种方法是维护时间戳，并根据请求资源的事务比占有资源的事务新还是老来决定事务的中止/等待。在等待-死亡方案下，较老的请求事务等待，而较新的请求事务以原时间戳回滚。在伤害-等待方案下，较新的事务等待，而较老的事务迫使占有资源的事务回滚并释放资源。
- 分布式数据：在分布式数据库中，数据可以水平划分（一个关系的元组分散到若干节点上）或垂直划分（一个关系模式被分解为若干模式，这些模式对应的关系位于不同节点上）。数据还可以进行复制，因此一个关系可能有若干完全相同的副本分别存在于不同节点上。
- 分布式事务：在分布式数据库中，一个逻辑的事务可能由多个成分构成，每个成分在一个不同的节点上执行。为了维护一致性，这些成分在提交还是中止该逻辑事务上必须取得一致。
- 两阶段提交：这种方式支持事务成分关于提交或中止的一致决定，即使在面临系统崩溃

时也能产生决定。在第一阶段中，协调器成分轮询各成分是希望提交还是希望中止。在第二阶段中，当且仅当所有成分都表达了提交的愿望时，协调器通知所有成分提交。

- 分布式封锁：如果事务必须封锁存在于多个节点上的数据库元素，那么必须用某种方式来协调这样的锁。在集中节点方式中，一个节点维护所有元素的锁。在主副本方式下，元素的锁由其主节点维护。
- 封锁有副本的数据：当数据库元素在多个节点上有副本时，元素的全局锁必须通过一个或多个副本上的锁来获得。大多数封锁方式要求在大多数副本上获得读锁或写锁以获得全局锁。另一种方法是，我们允许通过在任一副本上获得读锁来获得全局的读锁，而只有在获得了所有副本上的写锁时才获得了全局的写锁。
- sagas：当事务中包括持续时间很长、可能需要几小时或几天的步骤时，传统封锁机制可能极大地限制并发。saga由一个动作网络构成，其中的每个动作可能通向一个或多个其他动作，可能使整个saga完成，也可能使saga需要中止。
- 补偿事务：要使saga有意义，每个动作都必须有一个补偿动作，该补偿动作能取消原动作对数据库状态的影响，并且不影响已完成的saga或运行中的saga所做的任何其他动作。 1043
  如果saga中止，那么对应的补偿动作序列将被执行。

19.9 参考文献

[2]、[1]和[10]是这里所讨论话题的一些有用的、全面的资料来源。关于逻辑日志的讨论参考[9]。视图可串行性和多重图判断法来自[11]。[7]对死锁预防进行了综述；等待图就来自该文献。等待-死亡和伤害-等待方式来自[12]。

两阶段提交协议在[8]中提出。三阶段提交（本章未讨论）是一种更强有力的方式，它来自[13]。[4]中对恢复的领导选举方面进行了讨论。

分布式封锁方法由[3]（集中封锁方式）、[14]（主副本）和[15]（以副本上的锁构造全局锁）提出。

长事务由[6]引入。[5]中描述了saga。

1. N. S. Barghouti and G. E. Kaiser, "Concurrency control in advanced database applications," *Computing Surveys* **23**:3 (Sept., 1991), pp. 269–318.

2. S. Ceri and G. Pelagatti, *Distributed Databases: Principles and Systems*, McGraw-Hill, New York, 1984.

3. H. Garcia-Molina, "Performance comparison of update algorithms for distributed databases," TR Nos. 143 and 146, Computer Systems Laboratory, Stanford Univ., 1979.

4. H. Garcia-Molina, "Elections in a distributed computer system," *IEEE Trans. on Computers* **C-31**:1 (1982), pp. 48–59.

5. H. Garcia-Molina and K. Salem, "Sagas," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1987), pp. 249–259.

6. J. N. Gray, "The transaction concept: virtues and limitations," *Proc. Intl. Conf. on Very Large Databases* (1981), pp. 144–154.

7. R. C. Holt, "Some deadlock properties of computer systems," *Computing Surveys* **4**:3 (1972), pp. 179–196.

8. B. Lampson and H. Sturgis, "Crash recovery in a distributed data storage system," Technical report, Xerox Palo Alto Research Center, 1976.

9. C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. on Database Systems* **17**:1 (1992), pp. 94–162.

10. M. T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems*, Prentice-Hall, Englewood Cliffs NJ, 1999.

11. C. H. Papadimitriou, "The serializability of concurrent updates," *J. ACM* **26**:4 (1979), pp. 631–653.

12. D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II, "System-level concurrency control for distributed database systems," *ACM Trans. on Database Systems* **3**:2 (1978), pp. 178–198.

13. D. Skeen, "Nonblocking commit protocols," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1981), pp. 133–142.

14. M. Stonebraker, "Retrospection on a database system," *ACM Trans. on Database Systems* **5**:2 (1980), pp. 225–240.

15. R. H. Thomas, "A majority consensus approach to concurrency control," *ACM Trans. on Database Systems* **4**:2 (1979), pp. 180–219.

第20章 信息集成

伴随着现代数据库系统的演化和发展，信息集成（information integration）的研究领域内涌现出大量的新应用。这些应用是把存贮在两个或者更多个数据库（信息源，information source）中的数据提取出来，建立一个包含所有这些信息源的信息的大数据库（该数据库可以是虚拟的）。于是，可以对这个大数据库中的数据进行整体查询。这里的信息源既可以是传统的数据库，也可以是其他类型的信息，比如web网页集合。

本章中将介绍信息集成的重要概念。先概括给出集成的基本方法：联邦方式，数据仓库方式和协调器方式。然后分析包装器，这是一个软件，它允许信息资源符合一些共享的模式。为了有效操作信息集成系统，需要有专门的查询优化技术。为此，将简单地研究基于能力的优化技术，这是一种在传统DBMS中不常见的重要技术。

在考虑集成信息的应用类型时，特别重要的是“OLAP”（联机分析处理，on-line analytic processing）查询和“数据挖掘”（data-mining）查询。这些查询类型是所有数据库查询中最复杂的类型。有一种专门的数据库体系结构，叫做“数据立方体”（data-cube），是在某些应用中被用于组织集成数据的一种方式，以便有助于支持OLAP和数据挖掘查询。

20.1 信息集成的方式

有几种使数据库或其他分布式信息源协同工作的方式。在本节中，我们将考虑三个最常用方法：

1. *联邦数据库*。数据源是独立的，但一个数据源可以访问其他数据源以提供信息。

2. *数据仓库*。来自几个数据源的数据副本存储在单一数据库中，称其为（*数据*）*仓库*。存储在数据仓库中的数据在存储之前可能要经过一些处理，例如，对数据进行筛选，将关系进行连接或聚集。数据仓库定期更新，可能需要整整一夜的时间。当从数据源拷贝数据时，可能需要以某种方式对其进行转换，以使所有的数据都符合数据仓库的模式。

3. Mediation。协调器是一种软件组件，它支持*虚拟数据库*，用户可以查询这个虚拟数据库，就像它已*物化*（materialized）（已实际创建，就如数据仓库一样）。协调器不存储任何自己的数据，而是将用户的查询翻译成一个或多个对数据源的查询。然后，协调器将那些数据源对用户查询的回答进行综合处理，将结果返回给用户。

我们将依次介绍这些方法。所有方法的一个关键问题是当数据从信息源中提取出来时使用的数据转换方法。在20.2节，我们讨论这种转换器的结构，称其为*包装器*（wrapper）或*提取器*(extractor)。20.1.1节首先介绍包装器要解决的一些问题。

20.1.1 信息集成的问题

无论我们选用何种集成结构，当试图给不同数据源中的原始数据赋予一些含义时，总会产生一些微妙问题。我们把涉及同一类数据但在处理方法上存在各种差异的数据源（集合）称作*异构数据源*。一个扩充的例子将有助于揭示这些问题。

例20.1 Aardvark汽车公司有1000位代理商，每一位代理商都维护其库存汽车数据库。

Aardvark想创建一个集成数据库，以包含所有1000个数据源[⊖]的数据。集成数据库能帮助代理商找到库存中没有的某一型号汽车。公司分析人员也可将其用于预测市场和调整产量以提供销量最好的型号。

但是，这1000位代理商并不使用相同的数据库模式。例如，一位代理商可能将有关汽车的信息存储在单一关系中，其形式如下：

```
Cars(serialNo, model, color, autoTrans, cdPlayer, ...)
```

对每一个可能的选项有一个相应的布尔值与其对应。另一位代理商可能使用另一个模式，在这
1048 个模式中，选项分开存储在第二个关系中，如

```
Autos(serial, model, color)
Options(serial, option)
```

注意，不仅模式存在差异，而且很显然，等价的名称也发生了改变：Cars成为Autos，serialNo 变成serial。

使事情变得更为糟糕的是，不同数据库中的数据虽有相同含义，但以不同方式表示。

1. *数据类型不同*。序列号可在一个数据源中用变长字符串表示，而在另一个数据源中用定长字符串表示。定长字符串的长度也可能不同，而且一些数据源可能使用整数表示序列号，而不用字符串。

2. *值不同*。同一概念在不同数据源中用不同常数表示。黑色在一个数据源中可能用一个整数代码表示，而在另一个数据源中用字符串BLACK表示，在第三个数据源中用BL表示。更糟糕的情况是，BL在另一个数据源中可能表示“蓝”。

3. *语义不同*。许多重要术语在不同数据源中有不同的解释。一位代理商可能在关系Cars中包含有关卡车的信息，而另一位代理商在关系Cars中只存储小汽车信息 。一位代理商可能区分车站货车与小型货车，而另一位代理商则不区分。

4. *数据丢失*。一个数据源可能不记录所有或大部分数据源提供的类型信息。例如，一位代理商根本不记录颜色。为了处理丢失的数据，有时我们可以使用NULL或默认值。但是，现在的趋势是使用“半结构化”数据模型（如在4.6节提到的那样）来表示可能不完全一致的集成数据。但符合得可能不会那么精确。

在创建集成数据库之前，对数据源之间的每一种不一致都需要进行一种变换。 □

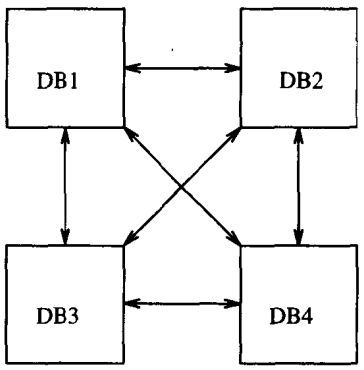


图20-1 四个数据库的联邦需要12个组件以相互翻译查询

20.1.2 联邦数据库系统

集成几个数据库的最简单结构可能是实现需要交互的所有数据库对之间的一对一连接。这些连接允许一个数据库系统D_1以另一个数据库系统D_2能理解的术语来查询D_2。这种结构的问题是，如果n个数据库中的每一个都需要与其他$n-1$个数据库进行交互，则我们必须写$n(n-1)$条代码以支持系统之间的查询。如图20-1所示。在这个图中，我们看到四个

⊖ 大多数真正的汽车公司都有适当的类似工具，且它们的发展史可能会与我们的例子有所不同。例如，可能会首先有一个集中数据库，而后代理商能够将相关数据下载到他们自己的数据库中。但是，这是目前很多产业公司试图采用的结构示例。

数据库形成了一个联邦。这四个数据库中每一个都需要三个组件，以存取其他三个数据库。

但是，在某些情况下，联邦系统可能是最容易建立的，特别是在数据库之间的通信本来就受限时。下面用一个例子说明翻译组件是如何工作的。 1049

例20.2 假设Aardvark汽车代理商想共享商品目录，但是每一位代理商只需要查询几个本地代理商的数据库，以查看他们是否有自己需要的汽车。更具体一些，考虑代理商1，他有一个关系

```
NeededCars(model, color, autoTrans)
```

这个关系的元组表示客户的汽车需求，客户通过型号、颜色和是否需要自动变速器来表达他们的这种需求。代理商2将目录存储在例20.1所讨论的有两个关系的模式中:

```
Autos(serial, model,color)
Options(serial, option)
```

代理商1写了一个应用程序，它远程查询代理商2的关系，以查找与NeededCars中描述的每一辆汽车匹配的汽车。图20-2 是这个程序的框图，嵌入式SQL语句查找所要汽车。这个程序的目标是用嵌入式SQL语句表示对代理商2数据库的远程查询，查询结果返回给代理商1。我们使用标准SQL规范，在一个变量前加一个冒号，表示从数据库中检索到的常数。

这些查询是针对代理商2的模式而写的。如果代理商1还想问代理商3相同问题，而代理商3使用的是例20.1中讨论的第一个模式，只有单一关系

```
Cars(serialNo, model, color, autoTrans,...)
```

查询将看起来大不相同。但是每一个查询适用于它所针对的数据库。 □ 1050

```
for(each tuple (:m, :c, :a) in NeededCars) {
    if(:a= TRUE) { /* automatic transmission wanted */
        SELECT serial
        FROM Autos, Options
        WHERE Autos.serial = Options.serial AND
              Options.option = 'autoTrans' AND
              Autos.model = :m AND
              Autos.color = :c;
    }
    else { /* automatic transmission not wanted */
        SELECT serial
        FROM Autos
        WHERE Autos.model = :m AND
              Autos.color = :c AND
              NOT EXISTS (
                  SELECT *
                  FROM Options
                  WHERE serial = Autos.serial AND
                        option = 'autoTrans'
              );
    }
}
```

图20-2 代理商1为需要的汽车查询代理商2

20.1.3 数据仓库

在*数据仓库*集成结构中，来自几个数据源的数据被抽取出来，合成一个*全局模式*。然后，数据存储在数据仓库中，这在用户看来与普通数据库无异。组织方式如图20-3所示，尽管数据

源可能多于图中所示的两个。

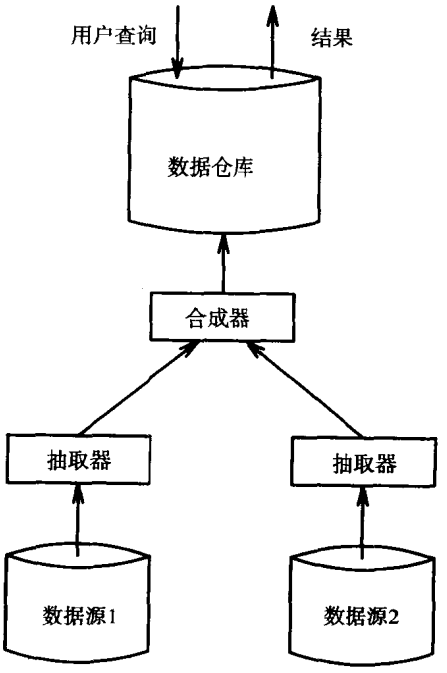


图20-3 数据仓库存储独立数据库中的集成信息

一旦数据存储在数据仓库中，用户就可以提出查询，正如他们向任何数据库提出查询一样。另一方面，通常不允许用户对数据仓库进行更新，因为这些更新不能反映在基本数据源中，并且可以导致数据仓库与数据源不一致。

数据仓库中数据的构造方法至少有三种：

1. 数据仓库根据数据源中的当前数据进行周期性地重建。这种方法是最常用的，数据重建每夜进行一次（当系统可以关闭时进行，所以在数据仓库重建时不能再查询），或间隔时间更
1051 长一些。这种方法的主要缺点在于需要关闭数据仓库，而且重建数据仓库需要的时间可能长于一“夜”。对某些应用来说，另一个缺点是数据仓库中的数据可能会非常过时。

2. 根据自上次数据仓库被更新以后对数据源所做的更新，对数据仓库中的数据进行周期性地更新（例如每个晚上）。这种方法可能只与少量数据有关，当数据仓库需要在很短的时间内进行更新，而数据仓库很大时（使用多个GB或TB的数据仓库），这很重要。缺点是计算数据仓库中的变化，即一种被称为*增量更新*的过程，与简单地重新构造数据仓库的算法相比，前者较复杂。

3. 对一个或多个数据源中的每一次变化或一组变化，数据仓库立即做出相应变化。这种方法需要太多的通信和处理，只适用于小的且底层数据源变化缓慢的数据仓库。但是，这是一种研究课题，而且如果这种数据仓库实现方法能成功，则将会有很多重要应用，例如在数据仓库中进行自动股票交易。

例20.3 为简单起见，假设在Aardvark系统中只有两位代理商，他们分别使用模式

1052
```
Cars(serialNo, model, color, autoTrans, cdPlayer,...)
```

和

```
Autos(serial, model, color)
Options(serial, option)
```

我们想建立具有以下模式的数据仓库

```
AutosWhse(serialNo, model, color, autoTrans, dealer)
```

即全局模式与第一位代理商使用的模式相似，但是我们只记录具有自动变速器的选项，并且我们包含一个属性表明哪一位代理商拥有这辆汽车。

从两位代理商的数据库中抽取数据并存入全局模式的软件可以用SQL查询来写。为第一位代理商写的查询很简单：

```
INSERT INTO AutosWhse(serialNo, model, color,
        autoTrans, dealer)
    SELECT serialNo, model, color, autoTrans, 'dealer1'
    FROM Cars;
```

为第二个代理商写的数据抽取器比较复杂，因为我们必须决定一个给定的小汽车是否有自动变速器。我们使用'yea' 和 'no' 作为属性autoTrans的值，含义很明白。这个抽取器的SQL代码如图20-4所示。

```
INSERT INTO AutosWhse(serialNo, model, color,
        autoTrans, dealer)
    SELECT serial, model, color, 'yes', 'dealer2'
    FROM Autos, Options
    WHERE Autos.serial = Options.serial AND
          option = 'autoTrans';

INSERT INTO AutosWhse(serialNo, model, color,
        autoTrans, dealer)
    SELECT serial, model, color, 'no', 'dealer2'
    FROM Autos
    WHERE NOT EXISTS (
        SELECT *
        FROM Options
        WHERE serial = Autos.serial AND
              option = 'autoTrans'
    );
```

图20-4 将代理商2的数据转换到数据仓库中的抽取器

在这个简单例子中，从数据源中抽取的数据的合成器为空。因为数据仓库是从每一个数据源中抽取的关系的并，所以我们显示了数据是直接加载到数据仓库中的。但是，许多数据仓库对它们从每一个数据源中抽取的关系进行操作。例如，将从两个数据源中抽取的关系进行连接后，再将结果放入数据仓库；或者我们可能将从几个数据源中抽取的关系进行并操作，而后对这个并的数据进行聚集。更普遍的是，从每一个数据源可以抽取几个关系，不同的关系以不同的方式组合。

20.1.4 协调器

协调器支持虚拟视图或视图集合，它集成几个数据源的方式与数据仓库中物化关系集成数据源的方式很相似。但是，因为协调器不存储任何数据，其机制与数据仓库机制大相径庭。图20-5表示一个协调器集成两个数据源。与数据仓库结构类似，数据源通常多于两个。首先，用户向协调器提出一个查询。因为协调器没有自己的数据，必须从它的数据源中得到相应数据，

并使用这些数据以形成对用户查询的回答。

1053 因而，我们在图20-5中看到，协调器向每一个包装器发送查询，包装器再依次向相应数据源发送查询。事实上，协调器可向一个包装器发送几个查询，还可不查询所有包装器。结果返回协调器进行组合。我们没有像在图20-3中的数据仓库图那样画出一个显式的合成器，因为使用协调器时，将来自数据源中的结果进行组合由协调器来完成。

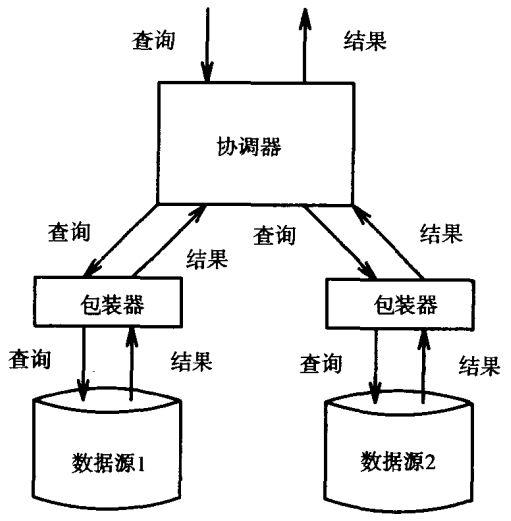


图20-5 协调器和包装器将查询翻译成的数据源的查询形式，并将应答进行组合

例20.4 我们考虑与例20.3相似的情况，但是使用协调器。即协调器将同样的两个汽车数据源集成为一个单一关系视图，其模式为：

```
AutosMed(serialNo, model, color, autoTrans, dealer)
```

假设用户提出以下查询，询问协调器有关红色汽车的信息：

```
SELECT serialNo, model
FROM AutosMed
WHERE color = 'red';
```

协调器对用户的这个查询作出反应，可能会将同样的查询转发到两个包装器。设计和实现处理类似查询的包装器的方法是20.2节的主题。对更复杂的情况，查询组件的翻译和分布是必

1054 要的，但是在这种情况下，翻译工作可由包装器独立完成。

代理商1的包装器将这个查询翻译成符合代理商模式的形式，该模式是：

```
Cars(serialNo, model, color, autoTrans, cdPlayer,...)
```

一种合适的翻译是：

```
SELECT serialNo, model
FROM Cars
WHERE color = 'red';
```

对这个查询的回答是serialNo-model对的集合，这个集合由第一个包装器返回到协调器。

同时，代理商2的包装器将同一个查询翻译成其模式，即

```
Autos(serial, model, color)
Options(serial, option)
```

适合经销商2的查询翻译几乎与前面的相同：

```
SELECT serial, model
FROM Autos
WHERE color = 'red';
```

与经销商1查询翻译不同的仅仅是被查询关系和一个结果属性的名字。第二个包装器将一个serial-model对的集合返回给协调器，并可能要将serial-model对的serial转变为集成模式中所用的serialNo-model。

协调器可使用这些集合的并，并将结果返回给用户。因为我们期望序列号是一个“全局键”，两辆汽车即使在不同的数据库中，也不存在相同序列号，所以我们可以采用包（bag）的并，假设不会有重复。 □ 1055

协调器用于回答查询的可选方法还有几种，在例20.4中未加说明。例如，协调器可以向一个数据源提出查询，查看结果，然后根据返回结果决定下一个或几个要提出的查询。如当用户查询查找是否有Aardvark“Gobi”型号的蓝色赛车时，就适用这种方法。第一个查询可询问代理商1，且只有当结果非空时，才向代理商2发出一个查询。

20.1.5 习题

! **习题20.1.1** 计算机公司*A*将它所卖的PC型号存储在以下的模式中：

```
Computers(number, proc, speed, memory, hd)
Monitors(number, screen, maxResX, maxResY)
```

例如，Computers中的元组（123，PIII，500，128，40）的意思是型号123有1000 MHz奔腾-III处理器，128M内存，40G硬盘。Monitors中的元组（456，19，1600，1200）意思是型号456有一19英寸的显示器，最大分辨率为1600 × 1200。

计算机公司*B*只卖完整的系统，包括一台计算机和一台显示器。它的模式是

```
Systems(id, processor, mem, disk, screenSize)
```

属性processor记录运行速度,以整数表示,不记录处理器类型（如奔腾-III）,也不记录显示器的最大分辨率。属性id、mem和disk与公司*A* 中的number、memory和hd类似，但是硬盘大小以MB而不是以GB来度量。

a) 如果公司*A*想把公司*B*中相关项目的信息插入它的关系中，应使用什么样的SQL语句?

* b) 如果公司*B*想向Systems插入尽可能多的、有关用*A*销售的计算机和显示器组装的系统的信息，用什么样的SQL语句最便于获得这样的信息?

*! **习题20.1.2** 提出一个全局模式，允许我们维护尽可能多的有关习题20.1.1中的公司*A*和*B*销售的产品的信息。

习题20.1.3 写SQL查询，从公司*A*和*B*的数据中搜集信息，并将这些信息放入习题20.1.2的全局模式中。如果你愿意，可以参考习题解答中给出的全局模式。 1056

习题20.1.4 假设习题20.1.2中的全局模式（或解答习题中的模式，如果你不喜欢自己的答

案）用作协调器，它如何处理查找具有1500 MHz处理器速度的任何计算机所能使用的硬盘最大容量这个查询？

! **习题20.1.5** 提出两个其他模式，计算机公司可能使用它们存储如习题20.1.1中的数据。如何将你的模式集成为习题20.1.2中的全局模式？

! **习题20.1.6** 在例20.3中我们说到了代理商1使用的一个关系Cars，它有一个属性autoTrans，这个属性只有值“yes”和“no”。因为这些值与全局模式中那个属性的值相同，因此关系Autos1的建立特别容易。假设属性Cars.autoTrans的值为整数，0意味着没有自动变速器，$i > 0$意味着小汽车有一个i速自动变速器。说明从Cars到Autos-Whse的翻译如何用SQL查询实现。

习题20.1.7 例20.4中的协调器如何翻译如下查询：

* a) 查找有自动变速器的汽车的序列号。

b) 查找没有自动变速器的汽车的序列号。

! c) 查找代理商1销售的蓝色汽车的序列号。

习题20.1.8 找几个在线书商的网页，看看你能找到多少有关这本书的信息。你将如何把这些信息组合成一个适用于数据仓库或协调器的全局模式？

20.2 基于协调器系统的包装器

在如图20-3所示的数据仓库系统中，数据源抽取器包括：

1. 一个或多个内置式查询，它们在数据源中执行，为数据仓库提供数据。
2. 合适的通信机制，使包装器能：
   (a) 向数据源传送特定查询；
   (b) 接收来自数据源的响应；
   (c) 向数据仓库传送信息。

1057

如果数据源是SQL数据库，如20.1节我们使用的例子，则内置的对数据源的查询可以是SQL查询。对于不是数据库系统的数据源，查询可以是以任何适用于数据源的语言书写的操作。例如，包装器可能填充一个网页中的在线表格，使用在线书目服务系统自身的特定语言发送一个查询，或使用其他各种符号来提出查询。

但是协调器系统需要比数据仓库系统更复杂的包装器。包装器必须能够从协调器接收各种查询，并将各个查询翻译成数据源的术语。当然，包装器必须将结果传送到协调器，就像是数据库系统中的包装器与数据仓库通信一样。在本节余下的部分我们将研究构造适用于协调器的灵活的包装器。

20.2.1 查询模式的模板

连接协调器与数据源的包装器的系统设计方法是将协调器可能要使用的查询分类，成为*模板*，它们是具有代表常数的参数的查询。协调器提供常数，包装器执行具有给定常数的查询。用一个例子来说明这种思想，在例子中，符号T => S表示模板T由包装器变成源查询S的思想。

例20.5 假设我们想从代理商1的数据源构造一个包装器，这个数据源的模式为

```
Cars(serialNo, model, color, autoTrans, cdPlayer, ....)
```

包装器由具有模式

```
AutosMed( serialNo, model, color, autoTrans, dealer)
```

的协调器使用。考虑协调器如何查询包装器以获得给定颜色的汽车。无论颜色是什么，如果能用参数$c代表表示那种颜色的代码，我们就能使用图20-6所示的模板。

```
SELECT *
FROM AutosMed
WHERE color = '$c';
    =>
SELECT serialNo, model, color, autoTrans, 'dealer1'
FROM Cars
WHERE color = '$c';
```

图20-6 描述对固定颜色汽车的查询的包装器模板

同样，这个包装器可能有另一个模板，它只规定参数$m表示一种型号，还可有另一个模板，它只指明是否需要一个自动变速器，等等。在这种情况下，如果允许查询指定三个属性中的任何一种：model、color和autoTrans，共有八种选择。总的来说，如果我们可指定n个属性[⊖]，则将有2^n种模板。如果有查找某一类型汽车总数的查询，或是否存在某一类型的汽车，则需要其他模板。尽管模板数目将会变得异常巨大，但是若设计包装器时使用更多的技巧，也有可能获得某些简化，如我们将在20.2.3节开始讨论的那样。 1058

20.2.2 包装器生成器

定义包装器的模板必须转变成包装器自己的代码。创建包装器的软件称为包装器生成器。从本质上说，它与分析器生成器（如YACC）类似；分析器生成器从高级规范说明中产生编译器的组件。生成过程如图20-7所示，当一个规范说明（即模板集合）输入包装器生成器时，这个过程就开始了。

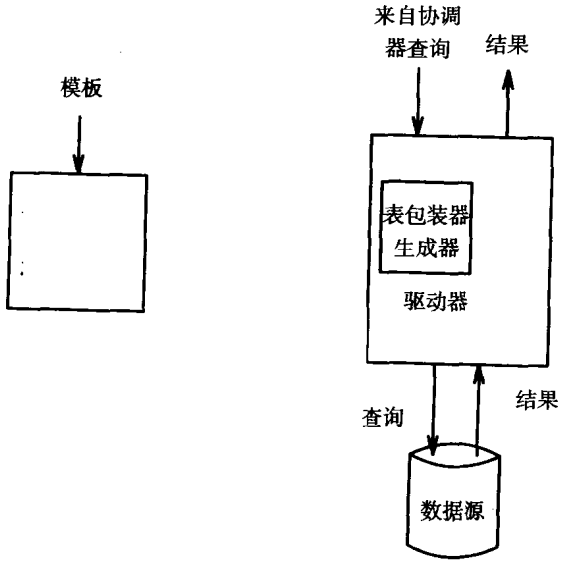


图20-7 包装器生成器为驱动器建表，驱动器和表组成包装器 1059

⊖ 如果数据源是一个能用SQL查询的数据库，如我们的例子，通过简单地将WHERE子句做成一个参数，你可以期望一个模板将任何数目的属性处理为常数。这种方法适用于SQL数据源和只将属性绑定到常数的查询，我们不必对任何数据源使用同样的思想，如网页只允许某些表格作为接口。在一般情况下，我们不能假设一个查询的翻译方式与所有类似查询的翻译都相同。

包装器生成器创建一个表，它存储模板中包含的各种查询模式和与每一个查询模式相关的源查询。每一个包装器都要用到一个驱动器；一般来说，对每一个生成的包装器，驱动器可以是相同的。驱动器的任务是：

1. 接收来自协调器的查询。通信机制可能是协调器专用的，并作为一个“plug-in”给予驱动器，从而同一驱动器可用于通信机制不同的系统。

2. 在所创建的表中查找匹配查询的模板。如果找到一个模板，则查询中的参数值用于实例化源查询。如果没有匹配的模板，包装器拒绝对协调器做出反应。

3. 源查询发送到数据源，又一次使用“plug-in”通信机制。数据源的答复由包装器收集。

4. 如果必要，数据源的答复由包装器处理，然后返回给协调器。下一节讨论包装器如何通过处理结果支持更大的查询类别。

20.2.3 过滤器

假设一位汽车代理商的数据库的一个包装器有如图20-6所示的模板，但是要让协调器查找某一型号和颜色的汽车。包装器可能被设计为一个具有如图20-8所示的更复杂模板，它处理既指明型号又指明颜色的查询。但是，正如我们在例20.5结束处所指出的那样，为每一个可能的查询都写一个模板是不现实的。

```
SELECT *
FROM AutosMed
WHERE model = '$m' AND color = '$c';
   =>
SELECT serialNo, model, color, autoTrans, 'dealer1'
FROM Cars
WHERE model = '$m' AND color = '$c';
```

图20-8 一个获得给定型号和颜色的包装器模板

支持更多查询的另一种方法是让包装器过滤向数据源所提查询的结果。只要包装器有一个模板(通过对参数的适当替代)，它返回查询所需结果的超集，则有可能在包装器中过滤返回的
1060 元组，只将所需元组传送到协调器。总的来说，决定一个协调器查询是否请求某个包装器模板的模式所返回的结果的子集是一个困难的问题，尽管在某些简单情况下，如我们所看到的例子中，理论已经很完善。参考文献包含了一些可以进一步研究的内容。

例20.6 假设我们所拥有的惟一模板如图20-6所示，它查找给定一种颜色的汽车。但是协调器要查找蓝色‘Gobi’型号的汽车，如使用查询

```
SELECT *
FROM AutosMed
WHERE color = 'blue' and model = 'Gobi';
```

一种可能的回答查询的方法是：

1) 使用图20-6的模板，用$c =’blue’查找所有蓝色汽车。

2) 将结果保存在下面的临时关系中：

```
TempAutos(serialNo, model, color, autoTrans, dealer)
```

3) 从TempAutos选择Gobis汽车，返回结果，如使用查询

```
SELECT *
FROM TempAutos
WHERE model = 'Gobi';
```

结果是所需汽车的集合。在实际中，TempAutos的元组将以流水线方式一次一个地产生，且一次一个地过滤，而不是在包装器中物化整个关系TempAutos，然后再过滤。 □ 1061

过滤组件的位置

在我们的例子中，假设过滤操作发生在包装器。也可能包装器将原始数据传送到协调器，由协调器过滤数据。但是，如果由模板返回的大部分数据不匹配协调器的查询，则最好在包装器过滤，避免传输不需要的元组的代价。

20.2.4 其他在包装器上进行的操作

只要我们确保模板的源查询部分将变换所需的所有数据返回到包装器，那么在包装器中以其他方式对数据进行变换也是可能的，例如，在将元组传输到协调器之前，对列进行投影。甚至可能在包装器进行聚集和连接，而后将结果传送到协调器。

例20.7 假设协调器想知道在各个代理商处的蓝色Gobi信息，但只查询序列号、代理商和是否有自动变速器，因为字段model和color的值从查询中显而易见。包装器可以像例20.6那样进行处理，但在最后一步，当结果要返回到协调器时，包装器在SELECT子句中执行投影，以及在WHERE子句中过滤以得到Gobi型号。查询

```
SELECT serialNo, autoTrans, dealer
FROM TempAutos
WHERE model = 'Gobi';
```

执行这种额外的过滤，尽管像在例20.6中那样，关系TempAutos可能会被以流水线方式送入投影操作符，而不是在包装器中进行物化。 □

例20.8 对一个更复杂的例子，假设要求一个协调器查找代理商和型号，条件是代理商有两辆同型号红色汽车，一辆有自动变速器，另一辆没有。假设对代理商1来说，惟一可用的模板是图20-6中有关颜色的模板，即协调器询问包装器，以查找对图20-9中查询的回答。注意，我们不必为A_1或A_2指明一位代理商，因为这个包装器只能存取属于代理商1的数据。协调器也将就同一个查询询问其他代理商的包装器。

```
SELECT A1.model A1.dealer
FROM AutosMed A1, AutosMed A2
WHERE A1.model = A2.model AND
      A1.color = 'red' AND
      A2.color = 'red' AND
      A1.autoTrans = 'no' AND
      A2.autoTrans = 'yes';
```

1062

图20-9 从协调器到包装器的查询

一个设计精巧的包装器能发现，有可能先从代理商1的数据源得到一个该代理商代理的所有红色汽车的关系，再回答协调器的查询：

```
RedAutos(serialNo, model, color, autoTrans, dealer)
```

为了得到这个关系，包装器使用图20-6得到的模板，这个模板处理只指明颜色的查询。结果是，包装器进行运行，就像给了它下面的查询：

```
SELECT *
FROM AutosMed
WHERE color = 'red';
```

而后，包装器通过使用图20-6的模板，令$c='red'，从代理商1的数据库构建关系RedAutos。下一步，包装器在RedAutos上执行自身连接，进行必要的选择，以得到图20-9

```
SELECT DISTINCT A1.model, A1.dealer
FROM RedAutos A1, RedAutos A2
WHERE A1.model = A2.model AND
      A1.autoTrans = 'no' AND
      A2.autoTrans = 'yes';
```

图20-10 为回答图20-9中的查询在包装器（或协调器）上执行的查询

中的查询所要求的关系。在这一步中，包装器所执行的工作[⊖]如图20-10所示。 □

20.2.5 习题

* **习题20.2.1** 在图20-6中，我们看到一个简单的包装器模板，它将来自协调器查找一种给定颜色汽车的查询翻译成针对具有关系Cars的代理商的查询。假设协调器的模式所使用的颜色代码不同于这个代理商所用的颜色代码，且存在一个关系GtoL (globalColor, localColor)实现这两套代码之间的翻译。重写模板，以产生正确的查询。

习题20.2.2 在习题20.1.1中，我们说到两个计算机公司*A*和*B*，使用不同模式描述其产品信息。假设我们有一个协调器，其模式为

1063 PCMed(manf, speed, mem, disk, screen)

直观的含义是，元组给出制造商（*A*或*B*），以及你要从那个公司购买的系统的处理器速度、主存大小、硬盘大小和显示器尺寸。为以下类型的查询书写包装器模板。注意你必须为每个查询写两个模板，每位制造商一个。

* a) 给定一个速度，查找具有那个速度的元组。

b) 给定一个显示器尺寸，查找具有那个尺寸的元组。

c) 给定内存和硬盘大小，查找匹配的元组。

习题20.2.3 假设对两个数据源(计算机制造商)的每一个，都可用习题20.2.2中所描述的包装器模板。协调器如何使用包装器的能力以回答下列查询？

* a) 查找具有1000 MHz速度和40 GB硬盘的所有系统的制造商、内存大小和显示器尺寸。

! b) 查找具有1500 MHz处理器的系统的硬盘最大容量。

c) 查找具有128 MB内存，显示器尺寸（以英寸计）大于硬盘大小（以GB计）的所有系统。

20.3 协调器基于能力的优化

16.5节介绍了基于成本优化的思想。典型的DBMS评估每个查询计划的代价，选出其中最好的一个。协调器要回答一个查询时，并不知道数据源要用多少时间来完成这些查询。甚至，很多数据源不是SQL数据库，通常它们只能回答协调器给出的查询类型中的一小部分。这样，协调器的查询优化不能单纯地依赖成本估量来选择查询计划。

协调器的优化通常采用简单的策略，如众所周知的基于能力的优化（capability-based optimization）。关键问题不是查询计划的成本，而是计划是否能被执行。仅评估那些被认为是可执行的(或是可行的)计划的成本。

本节将讨论数据源的能力为什么重要，接着描述表示能力的符号。最后考虑发现可行协调
1064 器查询计划的策略。

20.3.1 数据源能力有限的问题

现在，很多有用的数据源只有基于Web的界面，即使在其表像的后面是普通数据库时也如此。Web数据源通常只允许通过查询表格来进行查询，并不接受任意的SQL查询。这里，用户只要对指定的属性输入值，然后系统就接受响应给出其他属性的值。

例20.9 Amazon.com的界面允许用户以多种方式对书进行查询。可以指定一个作者以得到所有他写的书，可以指定书的标题以得到关于这本书的信息。可以指定关键字以得到和关键

⊖ 在某些信息集成结构中，该项工作可能实际上由协调器替代执行。

字匹配的书。但是答案中得到的某些信息是不可以指定的。例如，Amazon根据销售量对书分类，但是用户不能查询“给出排名前10的畅销书”。此外，它也不能查询太笼统的问题。如下面的查询：

```
SELECT * FROM Books ;
```

Amazon Web界面不能查询或者回答“给出系统所知的书的所有信息”一类问题，虽然如果能直接访问Amazon数据库的话，就可以透过表像来回答这类问题。 □

关于数据源为什么要限制查询的方式有很多其他原因：

1. *遗留数据源*（Legacy Sources）中保存的数据是陈旧的或是独特的。很多早期的数据库不使用DBMS，当然也就不是支持SQL查询的关系DBMS。这些系统很多被设计成只能以某种十分特殊的方式来查询。将这些数据移植到现代系统中几乎是不可能的，而且只能在这些遗留系统上运行的应用程序还在被使用。这种没人喜欢的被“锁定”在旧系统中的问题叫做遗留数据库问题（Legacy database problem），而且这种问题不可能很快被解决。

2. 从安全的角度考虑，数据源可以限制它接受的查询类型。Amazon不愿意回答“给出所有书的信息”就是一个例子，它防止了对手利用Amazon数据库。另外一个例子是医学数据库可以进行一般的查询，但是不可以(向未被授权的用户)透露某个病人病例的详细内容。

3. 大型数据库的索引可以使得某些类型的查询可行，而另外一些由于代价太高不可执行。例如，如果书库是关系型的，其中一个属性是作者，那么没有关于这个属性的索引的话，就不可能进行只指定一个作者的查询。每个这样的查询将要求查阅成千上万的元组[⊖]。 1065

20.3.2 描述数据源能力的符号

如果数据是关系型的，或是可以认为是关系型的，那么可以用装饰(adornment)[⊜]来描述查询的合法形式，装饰就是代表对关系属性要求的代码序列，该代码序列的次序是标准次序。本书将使用的装饰代码反映了数据源的最通常的能力。如：

1. f(自由)表示属性可以根据需要被指定或者不被指定。

2. b(受限)表示必需为属性指定一个值，但是允许任意值。

3. u(不指定)表示不允许为属性指定值。

4. c[S](从S集中选择)表示必须指定一个值，且这个值必须是属于有限集S。例如，这个选项对应于Web界面中下拉菜单中指定的值。

5. o[S](任选项，来自集合S)表示要么不指定一个值，要么指定有限集S中的一个值。

另外，如果属性不是查询输出的一部分，那么代码右上角加一标记（(如，f')。

数据源的能力规格说明（capabilities specification）是装饰的集合。意思是为了成功地对数据源进行查询，查询必须匹配能力规格说明中的一个装饰。注意，如果装饰有自由或任选的成分，那么不同属性集指定的查询可以与该装饰匹配。

例20.10 假定有两个与例20.4中那两个经销商相似的两个数据源。经销商1是具有如下格式的数据源：

```
Cars(serialNo, model, color, autoTrans, cdPlayer)
```

⊖ 必须意识到，不可以像操作关系数据库那样存取类似Amazon的产品信息。而且关于书的信息是以正文形式存储，具有例排索引。如13.2.4节中描述。此索引支持对书的任何类型查询，如作者、标题、标题中文字、正文中文字。

⊜ 此术语因实际情况中将关系的能力作为关系名的下标“修饰”而得来。

注意，原来的关系Cars还有表示可选的附加属性，但是本例为了简单起见，只考虑自动传送
1066 和CD唱片。经销商1有两种可能的方式来查询这些数据：

1. 用户指定序列号，关于那个序列号的汽车的所有信息(如，其他四个属性)作为输出。该查询形式的装饰是b'uuuu。也就是说，第一个属性serialNo必须被指定且不能作为输出的一部分。其他的属性不能被指定且是输出的一部分。

2. 用户指定模型和颜色，自动传送和CD唱片可有可无。所有匹配的汽车输出这个五个属性。一个合适的装饰是：

ubbo[yes，no]o[yes，no]

这个装饰说明不能指定序列号，而必须指定模型和颜色，但是允许其是任何可能的值。另外如果希望的话，可以指定是否需要自动传送和/或CD唱片，但是这些域的值仅仅只能用”yes”和”no”这两个值。

第二种方式的另一种可能的选择是，可以假定查询限制属性model 和/或color是有效值。也就是说，模型是从Aardvark实际上制造的模型中选择，颜色是从有效颜色中选择。如果是这样，那么uc[Gobi,...]c[red,blue....]bo[yes,no]o[yes,no]之类的装饰可能更加合适。 □

20.3.3 基于能力的查询计划选择

协调器给定一个查询，基于能力的查询优化首先考虑，对数据源端可以提出哪些查询将有助于回答该查询。设想一下所提出的和回答的那些查询，它是绑定了更多的属性，并且这些绑定使得在数据源端产生更多的查询成为可能。重复上面的过程直到遇到下面两种情况：

1. 在源端已经提出了足够多的查询来解决协调器查询的所有条件，因此完成这些查询。这样的策略称为可行的（feasible）。

2. 已不能再为源端构造更多有效形式的查询，但是协调器的查询仍然不能完成。这种情况下协调器必须放弃。因为协调器给出了一个不可能的查询。

需要应用以上策略的协调器查询的最简单形式是关系的连接，带有某种装饰的每个关系在一个或多个源处可用。如果这样的话，那么查询策略是试图得到连接中的每个关系的元组，方式是提供足够的参数绑定某个源，而该源允许那个关系响应询问和回答查询。下面的简单例子
1067 将说明这一点。

装饰保证什么?

如果支持查询的源匹配给定装饰，并且返回该查询所有可能的结果，那将是很奇妙的。然而，正常情况下源只拥有查询的可能结果的子集。例如，Amazon没有存储已经出版的每一本书，汽车例子中的两个经销商的数据库有不同的汽车集合。因此，装饰更准确的解释是：“以装饰描述的方式回答查询，给出的结果都是对的，但是不保证提供所有正确的结果。”

现在，考虑如果多个源提供相同的关系R会怎么样。协调器对于包含R的查询不只选择一个查询计划，相反，协调器选择的计划使用了R的每个源。如果源的装饰不同，那么计划可能不同。甚至，如果查询中涉及的关系有交错的源，那么不同计划的数目以这种关系的数目为基数成指数倍增长。

例20.11 假设有像例20.4中的经销商2关系的源如下：

```
Autos(serial, model, color)
Options(serial, option)
```

然而，假定Autos和Options是代表两个不同数据源中数据的关系[⊖]。假定ubf是Autos惟一的装饰，而Options有两个装饰，bu和uc[autoTrans,cdPlayer],表示在该源处要询问的两种不同的查询。令查询是“查找带有CD播放器的Gobi模型的序列号和颜色”。

这里协调器必须考虑的不同的查询计划有三种：

1. 指定模型是Gobi，查询Autos获取所有Gobis的序列号和颜色。接着，为每个这样的序列号对Options使用装饰 bu，找出那个汽车的选项并进行筛选以确保它拥有CD播放器。

2. 指定CD播放器选项，使用如下装饰查询Options：

uc[autoTrans,cdPlayer] 1068

获取带有CD播放器的汽车的所有序列号。接着查询Autos如(1)，得到Gobis的所有序列号和颜色，取两套序列号的交集。

3. 如(2)查询Options获取带有CD播放器的汽车的序列号。接着使用这些序列号来查询Autos看其中哪些汽车是Gobis。

前两种计划都是可行的。然而，第三种计划是这几个计划中不起作用的一个；系统没有能力执行这个计划，因为其第二部分——对Autos的查询——没有匹配的装饰。基于能力的优化器审查这些计划和所涉及关系的装饰，并消除上面第三种类型的不可行计划。 □

20.3.4 增加基于代价的优化

当审查源的能力时，协调器的查询优化并没有完成。因为已经找到了可行性计划，就必须从中选择。而要作出明智的、基于代价的优化要求协调器很清楚所涉及查询的代价。但是由于源通常都是独立于协调器，所以协调器很难评估代价。例如，源在少量载入时用的时间较少，但是载入时间是什么时候呢？即使是估计响应时间的长短，协调器也需要做长期的观察。

在例20.11中，可以简单地计算要发布的源的查询数目。计划(2)仅仅使用了两个源查询，而计划(1)使用了一个源查询，还要再加上Autos关系中找到的Gobis的数目。因此，计划（2）看起来比计划（1）代价低。另一方面，如果每个序列号对应的Options查询可以合并为一个查询的话，那么计划(1)将被认为是更好的选择。

20.3.5 习题

习题20.3.1 假定习题20.1.1的每个关系

```
Computers(number, proc, speed, memory, hd)
Monitors(number, screen, maxResX, maxResY)
```

是一个信息源。使用20.3.2节的符号，写出表达下面能力的一个或多个装饰：

* a) 能够查询给定处理器（如“P-IV”、“G4”或“Athlon”）、给定速度和(可选的)内存大小的计算机。

b) 能够查询指定硬盘大小和/或给定内存大小的计算机。 1069

c) 如果指定了显示器的数目、屏幕大小或两维的最大分辨率，则能够查询符合条件的显示器。

d) 如果指定了显示器屏幕大小，如15、17、19、或21英寸，能够查询符合条件的显示器，并且返回除显示器屏幕大小之外的所有属性。

⊖ 另外，我们可以假设经销商2的包装器支持协调器特性，允许它对使用基于能力技术发给经销商2的查询进行优化。

! e) 如果指定任意两种处理器类型、处理器速度、内存大小或硬盘大小，能够查询符合条件的计算机。

习题20.3.2 假定有习题20.3.1中的两个源，但是完整系统的属性是由两个关系的属性数组成。也就是说，该系统的一些属性在一个关系中而有一些属性在另外一个关系中。还假定对关系Computers访问的装饰描述是buuuu，ubbff，和uuuubb，而对Monitors的装饰描述是bfff和ubbb。找出下面查询计划中哪些是可行的（排除那些开销明显较大的计划）：

* a) 找出拥有128M内存、80G硬盘和19英寸显示器的系统。

b) 找出拥有Pentium-IV 2000MHz处理器、21英寸显示器且最大分辨率是1600*1200的系统。

! c) 找出拥有G4 750MHz的处理器、256M内存、40G硬盘和17英寸显示器的系统。

20.4 联机分析处理

现在，我们开始研究信息集成系统特别是数据仓库的一种重要应用。公司和组织创建一个数据仓库，拷贝大量可得到的数据，指定分析员对这个数据仓库进行查询以得到对这个组织非常重要的模式或趋势。这种称为OLAP（表示联机分析处理，读作“oh-lap”）的行为，通常包括使用一个或多个聚集的非常复杂的查询。这些查询经常被称作OLAP查询或决策支持查询。20.4.1节将给出一些例子，一个典型的例子是搜索总销售额增加或减少的产品。

数据仓库和OLAP

数据仓库之所以在OLAP应用中扮演重要角色的原因有几个。首先，数据仓库对于组织和集中公司相关的数据以支持OLAP查询是必要的，开始时数据可能分散在许多不同数据库中。但常常更重要的是，OLAP查询是复杂的，与许多数据有关，在一个事务处理系统中要使用大量的时间才能执行它，有很高的吞吐量。就19.7节来说，OLAP查询可被视为“长事务”。

锁住整修数据库的长事务会关闭普通OLTP操作（例如，如果有一个计算平均销售量的并发OLAP查询，则不能在新销售记录产生的同时对其进行录入。）一种常用方法是将原始数据拷贝在数据仓库中，在数据源执行OLTP查询和数据修改。在一般方案中，数据仓库只是在晚上进行更新，而分析人员则于白天在固定的数据副本上进行工作。数据仓库中的数据可能过时长达24小时，这限制了它对OLAP查询回答的及时性，但在许多决策支持应用中，这种延迟是可以承受的。

用于OLAP应用的决策支持查询通常要检查大量的数据，即使查询结果很小。相反，通常的数据库操作，如银行存款或航班预订，只与数据库中一小部分数据有关。后一种类型的操作通常被称作OLTP（在线事务处理，读作“oh-ell-tee-pee”）。

1070 最近，已开发出一些查询处理新技巧，对OLAP查询的执行有很好的效果。此外，因为某一类OLAP查询的本质特点，称作数据立方体系统的特殊形式的数据库处理系统，已经开发出来，并投入了市场，以支持OLAP应用。我们将在20.5节讨论这些系统。

20.4.1 OLAP应用

一个常见的OLAP应用使用销售数据的仓库。大型连锁店将积累几个TB的信息，表示每一个商店每一种物品的每一笔销售情况。将销售情况聚集成组且识别出具有特殊意义的组的查询，对公司预测未来问题或机会有极大用途。

例20.12 假设Aardvark 汽车公司建立了一个数据仓库，以分析它的汽车销售情况。数据仓库的模式可能是：

```
Sales(serialNo, date, dealer, price)
Autos(serialNo, model, color)
Dealers(name, city, state, phone)
```

一个典型的决策支持查询可能检查自2001年4月1日当天和以后的销售情况，以查看最近每一辆汽车的平均价格在各州有何不同。这样的查询如图20-11所示。

```
SELECT state, AVG(price)
FROM Sales, Dealers
WHERE Sales.dealer = Dealers.name AND
      date >= '2001-01-04'
GROUP BY state;
```

图20-11 按州查找平均销售价

1071

注意图20-11的查询是如何涉及数据库中大量数据的，因为它对每一个最近的Sales事实按销售它的代理商所在的州来分类。相反，常用的OLTP查询，如“查找序列号为123的汽车销售价格”，只涉及单一数据元组。 □

考虑另一个OLAP的例子，一个信用卡公司试图确定信用卡申请人是否可信。公司为它的所有顾客及还款记录建立了一个数据仓库。OLAP查询查找一些因素，如年龄、收入、是否有家和邮改编码，这些因素可帮助预测顾客是否能按时付账单。同样，医院可能使用一个病人数据仓库，关于他们的入院、化验管理、结果、诊断和治疗等，以分析风险和选择最好的治疗方法。

20.4.2 OLAP数据的多维视图

在典型的OLAP应用中，存在一个中心关系或数据集合，称作“事实表”。事实表代表感兴趣的事件或对象，如例20.12中的销售额。将事实表中的对象想像成在多维空间或“立方体”中排列是有帮助的。图20-12是三维数据的情况，数据由立方体内的点来表示。对应于先前汽车销售的例子，我们称其为汽车维、代理商维和时间维。从而，在图20-12中，我们将每一点想像为一辆汽车的一次销售记录，而维表示那次销售的属性。

图20-12的数据空间非正式地可看做“数据立方体”，或者更准确的说是原始数据立方体（raw-data cube），以便和20.5节中更复杂的“数据立方体”相区别。当有必要和原始数据区分开时，后者被看做是正式的数据立方体，它和原始数据立方体在以下两个方面有区别：

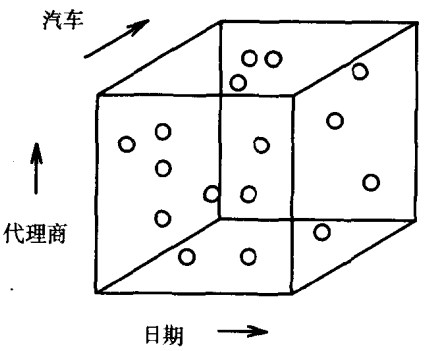


图20-12 组织成多维立方体的数据

1072

1. 它包括了所有维的子集数据的聚集，以及数据本身。

2. 正式数据立方体的点表示原始数据立方体中点的聚集。例如，取代表示每个单独汽车的“汽车”维（如在原始数据立方体中所表示的那样），该维只可以仅对车的模型来聚集，而正式数据立方体的点表示在给定日子中，给定的经销商所销售的给定模型的汽车的总销售量。

原始数据立方体和正式数据立方体两个大的方面的区别，被支持OLAP立方体结构数据专用系统所采用：

1. ROLAP 或关系OLAP。在这种方法中，数据存储在具有被称作“星型模式”的特殊结构的关系中。这些关系中的一个是事实表，它包含原始的或未聚集的数据。对查询语言和系统的

其他能力进行剪裁，以适用于数据是以这种方法组织的假设。我们将在20.4.3节讨论星型模式。

2. MOLAP或多维OLAP。这是一种特殊的结构，即上述的“数据立方体”，用于存储数据。如上所述，数据经常被部分聚集。系统可实现非关系型操作符，以支持对用这种结构形式表示的数据的查询。

20.4.3 星型模式

*星型模式*由事实表的模式组成，事实表与几个其他关系相连，它们被称作“维表”，下面会对其进行描述。事实表位于“星”的中心，它的点是维表。总的来说，事实表通常有几个表示维的属性和一个或多个依赖属性，这些依赖属性作为一个整体表示对该点有意义的一些特征。例如，销售数据的维可能包括销售日期、销售地点（商店）、所售物品类型、付款方式（如现金或信用卡），等等，而依赖属性可能是销售价格、物品成本或税。

1073 **例20.13** 例20.12中的Sales关系

```
Sales(serialNo, date, dealer, price)
```

是一个事实表。维是

1. serialNo，表示所售汽车，即在汽车空间中点的位置。
2. date，表示销售日，即在时间维中事件的位置。
3. dealer，表示在代理商空间中事件的位置。

一个依赖属性是price，它是向这个数据库所提出的OLAP查询在一个聚集中所通常要求的。但是，查找数目而不是总价或平均价格的查询也是有意义的，如“为每一位代理商列出2001年5月的销售总数”。 □

*维表*作为事实表的补充，它描述沿每一维的值。通常，事实表的每一维属性是一外码，引用相应维表的主码，如图20-13所示。维表的属性也描述在SQL GROUP BY查询中可能的分组。一个例子会使这种思想更清楚。

1074 **例20.14** 对例20.12中的汽车数据，两个维表是显然的：

```
Autos(serialNo, model, color)
Dealers(name, city, state, phone)
```

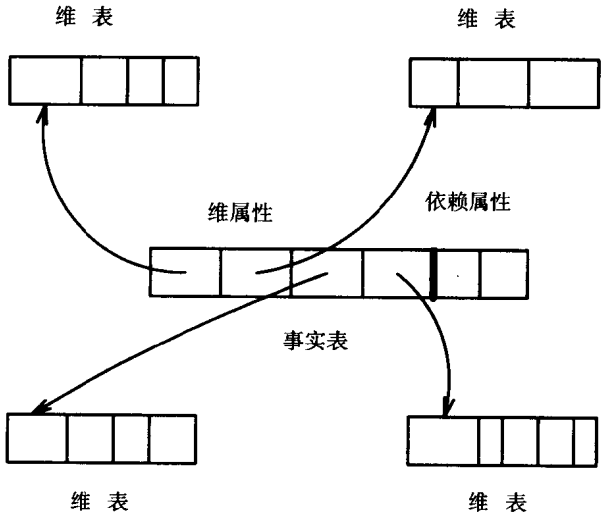


图20-13 事实表中的维属性引用维表的键

事实表Sales(serialNo, date, dealer, price)中的属性serialNo是一个外键，引用维表Autos的serialNo[⊖]。属性Autos.model和 Autos.color给出一汽车的特性。我们可以在这个关系中添加许多其他属性，如布尔属性，标明汽车是否有自动变速器或许多其他可能选项中的一个。如果我们将事实表Sales与维表Autos进行连接，则属性model和color可用于对销售以有意义的方式进行分组。例如，我们能要求将销售按颜色进行分解，或按月和代理商对销售进行分解。

同样，Sales的属性dealer是外键，引用维表Dealer的name。如果将Sales和Dealer进行连接，则我们可选择另外的数据分组方式，如可以将销售记录根据州或城市进行分解，也可以根据代理商进行分解。

人们可能想知道用于时间（Sales的date属性）的维表位于何处。时间是一个事实属性，在数据库中存储有关时间的事实没有意义，因为我们不能改变对诸如“2000年7月5日出现在哪一年？”此类问题的答案。但是，因为分析人员经常需要根据各种时间单位如周、月、季度和年分组，所以将时间概念加入数据库会有帮助，就如同存在一个时间维表，如

```
Days(day, week, month, year)
```

这个关系的一个典型元组是

```
(5, 27, 7, 2000)
```

表示2000年7月5日。解释为这天是2000年第7个月的第5天，它也是15为2000年的第27个整周。因为周可以从其他三个属性计算得到，所以产生了一定程度的冗余。但是，周并不与月精确相当，我们不能从按周分组得到按月分组，反之亦然。因而，在这个“维表”中，周和月都被表示是有意义的。 □

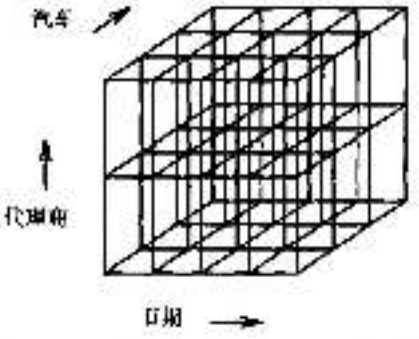


图20-14 通过沿每一维分割将立方体切块

20.4.4 切片和切块

我们可以将数据立方体的点想像为在某个粒度水平上沿每一维进行的分割。例如，在时间维上，我们可以根据日、周、月和年进行分割（SQL术语中的“group by”），或根本不做分割。对汽车维，我们可以根据型号、颜色以及型号和颜色进行分割，或不分割。对代理商，我们可以按代理商、城市以及州进行分割，或不分割。

对每一维进行分割的一种选择是对立方体切块，如图20-14所示。结果是立方体被分成更小的立方体，它们表示点的分组，通过一个在其GROUP BY语句中执行分割的查询对它们的统计信息进行聚集。通过WHERE语句，查询也可集中于沿一个或多个维（即在立方体中的特定“片”）的特定分割。结果是查询将在整个立方体的某个子空间中寻求值的聚集。

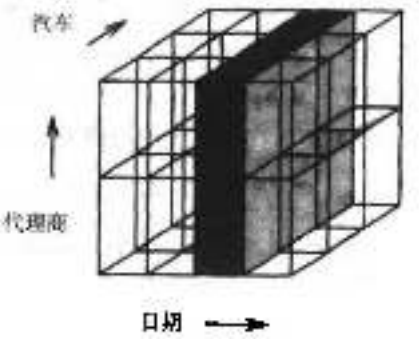


图20-15 选择切块立方体的一片

例20.15 图20-15是一个查询示例，在这个查询中我们要在某一维（日期）上切片，在其他两维（汽

⊖ serialNo属性也是Sales关系的主键，但是不必有一个属性，它既是事实表的键，也是某个维表的外键。

车和代理商）上切块。日期被分为四组，可以是积累数据的年数。图中的阴影说明我们只对这些年中的一个感兴趣。

汽车被分割为三组，轿车、SUV和可转换汽车，而代理商被分割为两组，西部和东部地区。查询结果是一个表，它给出感兴趣的那一年的六类中每一类的销售总额。 □

所谓“切片和切块”查询的一般形式为：

```
SELECT 分组属性和聚集
FROM 与零维或多维表连接的事实表
WHERE 等于常量的某些属性
GROUP BY 分组属性
```

1076 **例20.16** 让我们继续讨论汽车例子，但是包含用于在例20.14讨论过的时间的概念Days维表。如果Gobi的销售不如所想的那样好，我们可以试图查找哪一种颜色卖得不好。这个查询只用到Autos维表，可将SQL写为：

```
SELECT color, SUM(price)
FROM Sales NATURAL JOIN Autos
WHERE model = 'Gobi'
GROUP BY color;
```

这个查询根据颜色切块，然后根据型号切片，集中于一个特定型号Gobi，而忽略其他数据。

假设查询没有告诉我们很多信息，每一种颜色的收入大致相等。因为查询没有在时间上进行分割，我们只看到每一种颜色在所有时间上的销售总额。我们可以假设最近的趋势是一个或多个颜色的销售量在下降，因而可以发布一个查询，它按月分割时间，这个查询是：

```
SELECT color, month, SUM(price)
FROM (Sales NATURAL JOIN Autos) JOIN Days ON date = day
WHERE model = 'Gobi'
GROUP BY color, month;
```

重要的是要记住，Days关系并不是传统的存储关系，尽管我们可以把它处理成似乎它有以下模式：

```
Days(day, week, month, year)
```

使用这样的“关系”的能力表明一种方式：数据立方体系统是DBMS的一种特化。

我们可以发现红色Gobi最近销售得不怎么好。接下来我们可能会问，这个问题对所有的代理商都存在吗？还是仅有几个代理商的红色Gobi销得不好。这样，我们会进一步将查询集中在
1077 只看红色Gobi，并沿代理商维来分区。这样的查询为：

```
SELECT dealer, month, SUM(price)
FROM (Sales NATURAL JOIN Autos) JOIN Days ON date = day
WHERE model = 'Gobi' AND color = 'red'
GROUP BY month, dealer;
```

此时，我们发现红色Gobi的月销售量非常少，以致不能发现任何趋势，因此，我们认为按月分割是个错误。一个更好的主意是只按年分割，且只看过去的两年（在这个假想的例子中，是2001年和2002年）。最后查询如图20-16所示。 □

```
SELECT dealer, year, SUM(price)
FROM (Sales NATURAL JOIN Autos) JOIN Days ON date = day
WHERE model = 'Gobi' AND
      color = 'red' AND
      (year = 2001 OR year = 2002)
GROUP BY year, dealer;
```

图20-16 有关红色Gobi销售量的切片和切块

20.4.5 习题

* **习题20.4.1** 一个计算机在线销售商想维护有关定单的数据。顾客可以定购他们的PC，这些PC装有几种处理器之一、选定的主存容量、几种磁盘之一以及几种CD或DVD读取器之一。这样，一个数据库中的事实表可能是：

```
Orders(cust, date, proc, memory, hd , rd, quant, price)
```

我们应该明白，cust属性是一个ID，它是有关顾客的一个维表的外键，属性proc、hd和rd同样也是有关表的相应外键。例如，可以在一个维表中更详细地说明一个磁盘ID，给出磁盘制造商和几个磁盘特征。属性memory只是一个整数，即定购的内存兆数，属性quant表示这位顾客要定购的机器数量，price表示每一台定购机器的总价格。

a) 哪些是维属性，哪些是依赖属性？

b)对一些维属性，可能需要一个维表。给出这些维表的一种适当模式。 1078

! **习题20.4.2** 假设我们想检查习题20.4.1中的数据以查找销售趋势，从而预测公司应该更多地定购哪些组件。描述一系列的下钻（drill-down）和上卷（roll-up）查询，它们能产生“顾客开始更喜欢DVD驱动器而不是CD驱动器”的结论。

下钻和上卷

例20.16以切片和切块数据立方体的查询序列形式，说明了两种常用方式。

1. 下钻是更精细地分割和或集中在某些维上的特定值的过程。在例20.16中，除最后一步外其余各步都是下钻的例子。

2. 上卷是更粗一些的分割过程。在最后一步中，我们按年分组而不是按月来消除数据的随机性结果，是上卷的一个例子。

20.5 数据立方体

在这个小节里面，将考虑”正规”的数据立方体以及用这种形式表示的数据上的操作。回忆20.4.2小节中的正规数据立方体（在本节就叫做‘数据立方体’），用系统性方法预先计算了所有可能的聚集。令人吃惊的是，额外所需的存储空间一般可以忍受，而且只要数据仓库的数据没有改变，保持所有的聚集为最新版是可行的。

在数据立方体系统中，将事实表中的原始数据存入数据立方体存储系统之前，对其进行一些聚集是正常的。例如，在汽车例子中，我们认为是星型模式中序列号的维可能由汽车型号来代替。那么数据立方体中的每一个记录便描述型号、代理商和日期，以及该型号的汽车在那个日期由该代理商所销售的总额。我们将继续把数据立方体中的点称作“事实表”，即使对点的解释可能与星型模式中的事实表有所不同。

20.5.1 立方体操作符

给定一个事实表F，我们能定义一个扩充表CUBE（F），这个表向每一维添加一个附加值，

1079 以*表示。这里的*具有它的本意“任何”，且表示沿它出现的维的聚集。图20-17表示了为立方体的每一维添加一个边的过程，以此来表示它所代表的*值和聚集值。在这个图中，我们看到三维，颜色最浅的阴影表示在一维上进行的聚集，较深的阴影表示在二维上的聚集，在边角处颜色最深的立方体表示在所有三维上的聚集。注意，如果每一维上值的数目相当大，但还不足以使立方体中的大部分点都被未占用，则“边”只代表对立方体容积（即事实表中的元组数）的少量增加。在这种情况下，所存储数据的CUBE（F）大小并不比F自身的规模大多少。

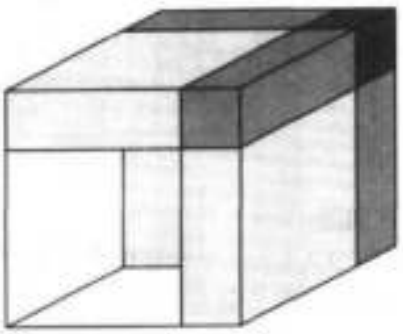


图20-17 立方体操作符对数据立方体进行扩充，使其具有在维的所有组合上进行聚集的边

在一个或多个维上具有*的表CUBE（F）中，对每一个依赖属性，元组将具有所有元组中那个属性值的总和（或另一个聚集函数），通过用真实的值替代*，我们获得这个总和。事实上，我们在数据中加入沿任何维集合的聚集。但是，注意CUBE操作符不支持基于维表数据的中间粒度级别上的聚集。例如，我们可能要么按日将数据进行分解（或对时间来说最细的粒度），要么完全聚集时间，但是我们不能单独使用CUBE操作符，按周、月或年来聚集。

例20.17 让我们根据CUBE操作符，重新考虑例20.12中的Aardvark数据库。回想那个例子中的事实表是

```
Sales(serialNo, date, dealer, price)
```

但是，serialNo代表的维不太适合用于立方体，因为序列号惟一标识了一部汽车，从而serialNo是Sales的一个码。所以，要对所有日期的价格求和，或对所有代理商求和，但保
1080 持固定序列号不起作用，我们仍将得到具有那个序列号的汽车的“总和”。一个更有用的数据立方体将用两个属性——型号和颜色，来代替序列号，通过维表Autos，序列号将Sales与这两个属性联系。注意，如果我们使用model和color代替serialNo，则立方体的维中不再有码，因此，立方体的数据项将有某一型号的某一颜色的汽车由某一位代理商在某一日期的销售总额。

还有一种改变对于Sales事实表的数据立方体实现有用。因为CUBE操作符通常对依赖属性求和，而我们可能想得到某种分类的平均销售价格，我们既需要每一个汽车分类的销售总额（某一位代理商在某一天所卖的某一型号的某一颜色汽车），也需要那个类别的销售总量。因而，我们使用CUBE操作符的关系Sales是

```
Sales(model, color, date, dealer, val, cnt)
```

属性val用于表示给定型号、颜色、日期和代理商的所有汽车的总价值，属性cnt是那个类别的汽车总量。注意，在这个数据立方体中，单辆汽车并不被识别，它们只影响其所属类别的总价和数量。

现在，让我们考虑关系CUBE（Sales）。既在Sales中又在CUBE（Sales）中的一个可能的元组为

```
('Gobi','red','2001-05-21','Friendly Fred', 45000, 2)
```

对它的解释是：在2001年5月21日，代理商Friendly Fred卖了两辆红色Gobi，总价值为$45 000。元组

```
('Gobi', *, '2001-05-21', 'Friendly Fred', 152000, 7)
```

表示在2001年5月21日，代理商Friendly Fred卖了7辆各色Gobi，总价值为$152 000。注意这个元组在CUBE（Sales）中，但不在Sales中。

关系CUBE（Sales）也包含表示在多于一个属性上的聚集的元组。例如，

('Gobi', *, '2001-05-21', *, 2348000, 100)

表示在2001年5月21日，所有代理商卖了100辆Gobi，总价值为$2 348 000。

('Gobi', *, *, *, 1339800000, 58000)

表示在所有时间内，所有代理商共卖了58 000辆各色Gobi，总价值为$1 339 800 000。

最后，元组

(*, *, *, *, 3521727000, 198000)

1081

告诉我们所有代理商在所有时间共卖了各种Aardvark型号汽车198 000辆，总价值$3 521 727 000。□

考虑如何回答这样一个查询，在这个查询中，我们在Sales关系的某几个属性上指明条件，并按其他一些属性进行分组，同时查询总和、数量或平均价格。在关系CUBE（Sales）中，我们查找具有以下特点的元组t：

1. 如果查询为属性a指明一个值v，则元组t在它表示a的分量中有v。
2. 如果查询根据一个属性a进行分组，则t在它表示a的分量中有任何非*值。
3. 如果查询既不根据属性a进行分组，也不为a指明一个值，则t在它表示a的分量中有*。

每一个元组t有期望得到的分组之一的总价值和数量。如果我们想要平均价格，则对每一个元组t，在总价值和数量分量上进行相除。

例20.18 对查询

```
SELECT color, AVG(price)
FROM Sales
WHERE model = 'Gobi'
GROUP BY color;
```

的回答是通过查找CUBE（Sales）中所有具有形式

('Gobi', c, *, *, v, n)

的元组，其中c是任意指定的颜色。在这个元组中，v是那个颜色的Gobi的销售总额，n是那个颜色的Gobi的销售量。这个查询所要求的元组是（c, v/n），也就是说平均价格，尽管它不直接是Sales或CUBE（Sales）的属性，但是可以通过用汽车数量去除总销售额而得到。对这个查询的回答是从('Gobi', c, *, *, v, n)得到的（c, v/n)对的集合。□

20.5.2 通过物化视图实现立方体

在图20-17中我们说明了给立方体添加聚集不会费太多的空间，而且会使常见的决策支持查询的执行节省大量时间。但是，我们的分析是基于查询选择在一维上要么完全聚集，要么不聚集这样的假设。对一些维来说，有许多粒度可用于选择作为在那个维上的分组。

我们已经提到时间情况，对时间来说，除按日进行全部或空的分组选择或在所有时间上进行聚集外，还存在许多选择，如按星期、月、季度和年进行聚集。另一个基于汽车数据库的例子是我们可能选择对代理商完全聚集或根本不聚集。但是，我们也可能选择按城市、州或其他地区以及大或小进行聚集。因而，对时间分组的选择至少有六种，对代理商进行分组的选择至少有四种。

1082

当沿每一维分组的可选方法数量增加时，存储按每一个可能的分组组合的聚集结果变得格外昂贵。不仅有许多这样的结果，而且它们不像全部-或-空的情况下那样容易组织成如图20-17所示的结构。因此，商品化的数据立方体系统可以帮助用户选择一些数据立方体的物化视图。物化视图是一些查询结果，我们将它存储在数据库中，而不是响应查询要求对其全部或部分进行重建。对数据立方体来说，我们选择来进行物化的视图是典型的对整个数据立方体的聚集。

分组所蕴含的划分越粗，物化视图所需空间就越少。另一方面，如果我们想使用视图去回答某一查询，则视图对任一维的划分不能比查询对这一维的划分更粗。从而，为了最大化物化视图的作用，我们通常想用一些大视图，它们将维分组成相当细的划分。另外，分析人员可能要提出的查询类型会极大地影响对要物化的视图的选择。用一个例子提出所涉及到的权衡问题。

例20.19 让我们回到例20.17中研究的数据立方体

```
Sales(model, color, date, dealer, val, cnt)
```

一个可能的物化视图按月将日期分组，按城市将代理商分组。我们称这个视图为SalesV1，由图20-18中的查询建立。这个查询不是一个严格的SQL，因为我们想像数据立方体系统能理解日期和它们的分组单元如月，而不需要告诉数据立方体系统将Sales与例20.14中讨论的表
1083 示日的假想关系进行连接。

```
INSERT INTO SalesV1
    SELECT model, color, month, city,
        SUM(val) AS val, SUM(cnt) AS cnt
    FROM Sales JOIN Dealers ON dealer = name
    GROUP BY model, color, month, city;
```

图20-18 物化视图SalesV1

另一个可能的物化视图对颜色进行完全聚集，将时间聚集成星期，根据州对代理商进行聚集。这个视图SalesV2由图20-19中的查询来定义。

```
INSERT INTO SalesV2
    SELECT model, week, state,
        SUM(val) AS val, SUM(cnt) AS cnt
    FROM Sales JOIN Dealers ON dealer = name
    GROUP BY model, week, state;
```

图20-19 另一个物化视图SalesV2

视图SalesV1或SalesV2可用来回答在二者的任一维上不做更细划分的查询。因此，查询

```
Q1: SELECT model, SUM(val)
    FROM Sales
    GROUP BY model;
```

可以用

```
SELECT model, SUM(val)
FROM SalesV1
GROUP BY model;
```

来回答，也可以用

```
SELECT model, SUM(val)
FROM SalesV2
GROUP BY model;
```

来回答。另一方面，查询

```
Q2: SELECT model, year, state, SUM(val)
    FROM Sales JOIN Dealers ON dealer = name
    GROUP BY model, year, state;
```

只能从SalesV1中得到回答：

```
SELECT model, year, state, SUM(val)
FROM SalesV1
GROUP BY model, year, state;
```

附带说明一下，上面这个查询，像聚集时间单元的查询一样，不是严格的SQL。即，state不是SalesV1的属性；只有city是SalesV1的属性。我们必须假设数据立方体系统知道如何执行将城市聚集成州，可能是通过访问为代理商建立的维表。 1084

我们不能从SalesV2回答$Q2$。尽管我们能将城市上卷成州（如将城市聚集成它们的州）以使用SalesV1，但不能将星期上卷成年，因为年不能被星期整除。比如以开始于2001年12月29日的某周作参考，对2001年和2002年内的数据做统计，我们就不能区分按星期聚集的数据。

最后，一个类似的查询：

```
Q3: SELECT model, color, date, SUM(val)
    FROM Sales
    GROUP BY model, color, date;
```

的查询既不能从SalesV1也不能从SalesV2得到回答。它不能从SalesV1得到回答，是因为它的按月来划分天数太粗，不能包含按日的销售额。它也不能从SalesV2得到回答，是因为这个视图不是按颜色进行分组的。我们将不得不直接从整个数据立方体回答这个查询。 □

20.5.3 视图的格

为了形式化例20.19中的现象，把对立方体的每一维进行的可能分组想像成格是有帮助的。分格的方法就是按其维表的一个或多个属性通过分组对这一维的值进行划分。我们称划分P_1在划分P_2之下，当且仅当P_1的每一个分组包含在P_2的某些分组内，记作$P_1 \leqslant P_2$。

例20.20 对时间划分的格可以选择图20-20中的格。从某个结点P_1向下到P_2的一条路径表示$P_1 \leqslant P_2$。这些不是惟一可能的时间单元，但它们可用于作为系统可能支持什么单元的示例。注意，日位于周和月的下面，但是周并不位于月的下面。原因是发生在一天内的一组事件肯定发生在一个周和一个月内，但是发生在一周内的一组事件不一定发生在一个月内。同样，一周分组不必包含在对应于季度和年的分组内。最上面的是我们称之为“全部”的划分，意思是事件被分成一个组，即我们不区分时间。 1085

图20-21表明了另一个格，这次是为汽车例子中的代理商维建立的。这个格比较简单，它表明按代理商划分销售额比按代理商所在城市进行划分更细，按代理商所在城市进行划分比按代理商所在州划分更细。格的上面是将所有的代理商放入一个分组的划分。 □

如果每一维都有一个格，现在我们可以为数据立方体的所有可能的物化视图定义一个格，它根据对每一维的一些划分通过分组而形成。如果说V_1和V_2是两个视图，它们通过为每一维选择一种划分而形成，则$V_1 \leqslant V_2$意味着在每一维上，在V_1中我们使用的划分P_1至少与V_2中我们为那个维使用的划分P_2一样细，即$P_1 \leqslant P_2$。

许多OLAP查询也能放在视图格中。事实上，OLAP查询与我们已讨论过的视图常常具有相同形式：查询为每一维指明一些划分（可能空的或全部）。其他OLAP查询如我们在图20-15

中所示的那样，包括相同的分组类型，而后对立方体进行“切片”，以集中于数据的某一子集。一般的规则是：

• 我们能用视图V回答一个查询Q当且仅当$V \leqslant Q$。

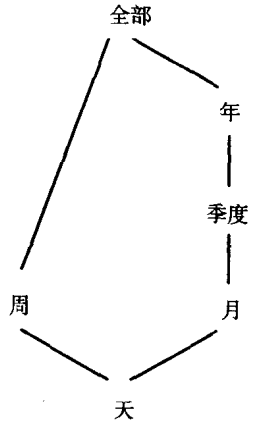


图20-20 对时间区间划分的格



图20-21 汽车代理商划分的格

例20.21 图20-22使用例20.19中的视图和查询，并将它们放在一个格中。注意Sales数据立方体本身从技术上说是一个视图，对应于沿每一维可能进行的最细划分。正如我们在原来的例子中所看到的那样，Q_1既可以从SalesV1得到回答，也可从SalesV2中得到回答；当然它也从整个数据立方体中得到回答，但是如果一个其他视图已物化，则没有这样做的任何理由。Q_2可以从SalesV1 或Sales中得到回答，而Q_3只能从Sales中得到回答。每一个这样的关系都用通过从查询到支持它们的视图的向下路径来表示，如图20-22所示。□

1086

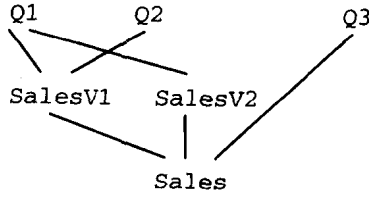


图20-22 例20.19中视图的格和查询

将查询放在视图格中可以帮助设计数据立方体数据库。一些最近开发的用于数据立方体的设计工具，从它们认为目前应用中非常典型的一个查询集合开始，然后选择一个视图集合来物化，使每一个查询至少在一个视图的上面，最好与它相同或非常接近（即查询和视图在大部分维中使用相同的分组）。

20.5.4 习题

习题20.5.1 如果事实表F具有下列的特点，则CUBE（F）的大小与F的大小的比率是多少？

* a) F有10个维属性，每一个维属性有10个不同的值。

b) F有10个维属性，每一个维属性有两个不同的值。

习题20.5.2 使用例20.17中的立方体CUBE（Sales），它是从下列关系中构建的

```
Sales(model, color, date, dealer, val, cnt)
```

说明我们使用立方体的哪些元组来回答下列查询：

* a) 为每一位代理商查找蓝色汽车的总销售额。

b) 查找代理商“Smilin Sally”所卖的绿色Gobi的总数量。

1087

c) 查找每一位代理商在2000年3月的每一天所卖的Gobi的平均数量。

*! **习题20.5.3** 在习题20.4.1中，我们提到将PC定货单数据组织成一个立方体。如果我们要使用CUBE操作符，可能会发现将几个维进行更细的分解会更方便。例如，我们不使用一个处理器维，而是对每一种型号都有一个维（如，AMD K-6或奔腾-IV），为速度建立另一个维。提出维和依赖属性的集合，我们将用它们获得多种聚集查询的回答。特别地，顾客扮演什么角色？另外，一台机器的价格参考习题20.4.1中的价格，且几个相同机器可在一个元组内进行定购。用什么作为依赖属性？

习题20.5.4 你将使用习题20.5.3中的什么元组来回答下列查询：

a) 对每一种处理器速度，查找2002年每一个月内定购的计算机的数量。

b) 为每一种硬盘类型（如SCSI或IDE）和每一种处理器类型，列出所定购的计算机的数量。

c) 查找自从2001年1月以来每一个具有1500 MHz处理器的计算机的平均价格。

! **习题20.5.5** 习题20.5.3的立方体所描述的计算机不包括显示器。你会建议使用什么维来表示显示器？可以假设显示器的价格包含在计算机的价格之内。

习题20.5.6 假设一个立方体有10个维，且每一维有5种聚集粒度的选择，包括“无聚集”和“全部聚集”。通过在每一维中选择一个粒度，我们能创建多少不同的视图？

习题20.5.7 说明如何将下列的时间单元加到图20-20的格中：小时、分、秒、两周、十年和世纪。

习题20.5.8 如何改变图20-21中的代理商格，使得包含“区域”，如果：

a) 区域是州的集合。

* b) 区域不与州匹配，但每一个城市在一个区域中。

c) 区域与地区代码类似，每一个区域都包含在一个州内，一些城市在两个或更多区域中，一些区域包括几个城市。

1088

! **习题20.5.9** 在习题20.5.3中，我们设计了一个立方体，适用CUBE操作符。但是，也可以给一些维赋予非平凡的格结构。特别地，处理器类型可以根据制造商（如SUN、Intel、AMD和Motorola）、系列号（如SUN UltraSparc、Intel Pentium或 Celeron、AMD K-系列，或 Motorola G-系列）和型号（如Pentium-IV或AMD K-6）进行组织。

a) 根据上面的描述设计处理器类型的格。

b) 定义一个视图，使其根据系列号对处理器分组，根据类型对硬盘进行分组，对根据速度CD进行分组，并对所有其他属性进行聚集。

c) 定义一个视图，使其根据制造商对处理器进行分组，对根据速度硬盘进行分组，对除主存以外的任何其他属性进行聚集。

d) 给出查询示例，使其分别只能从(b)的视图中得到回答，只能从(c)的视图中得到回答，或从二者都能得到回答，或从二者都得不到回答。

*!! **习题20.5.10** 如果我们对其使用CUBE操作符的事实表F是稀疏的（即F的元组比每一维可能的值的数目乘积小得多），则CUBE（F）的大小与F的大小的比率可能会非常大。这个比率能有多大？

20.6 数据挖掘

一系列被称为*数据挖掘*或*数据库中知识发现*的数据库应用已经引起了相当的重视，因为这些应用提供了从现有数据库中获取不寻常事实的条件。数据挖掘查询可被想像成决策支持查询

的扩展形式，虽然二者的区别是非形式的（参看“数据挖掘查询和决策支持查询”框）。数据挖掘着重于传统数据库系统的查询优化和数据管理部分，对数据库语言也提出一些重要扩充。在本节中，我们将考查数据挖掘应用的主要发展方向。而后，我们将集中于称作“频繁项目集”的问题，这种问题已从数据库的观点得到极大的关注。

数据挖掘查询和决策支持查询

虽然决策支持查询需要检查和聚集数据库中大部分数据，提出查询的分析人员通常会准确地告诉系统执行什么查询，即集中在数据的哪一部分。数据挖掘查询向前进了一步，引入系统以决定应集中于数据的哪一部分。例如，一个决策支持查询可能会问“根据颜色和年份对Aardvark汽车销售额进行聚集”，而一个数据挖掘查询可能会问“哪一个因素对Aardvark销售额影响最大？”。数据挖掘查询的朴素实现将引起许多决策支持查询的执行，且可能需很长时间才能完成，因而朴素方法是完全不可行的。

20.6.1 数据挖掘的应用

广泛地讲，数据挖掘查询查找数据的有用汇总，而经常不提供最好地获得这些汇总的参数值。因此，这种问题需要重新考虑将数据库系统用于提供对数据这样理解的方法。下面是一些
1089 数据量非常巨大的应用及其要解决的问题。因为在许多这些问题中如何最好地使用DBMS是未解决的，我们将不讨论对这些问题的解决方案，只说明它们为何如此困难。在20.6.2节中，我们将讨论已取得进展的那些问题，在那里我们将看到一种非平凡、面向数据库的解决方法。

决策树构造

数据用户想在数据中发现一种决定重要问题的方法。例如，例14.7为我们引入可作为一种有意义的数据挖掘问题的基础：“谁买金首饰？”。在那个例子中，我们只考虑顾客的两个特征：他们的年龄和收入，但是，现在的顾客数据库记录更多的顾客信息，或从合法数据源获得信息，而后将它们与顾客数据一起集成到数据仓库。这些属性有顾客的邮政编码、婚姻状况、有房或租房以及有关他或她最近购买的其他物品数目的信息，等等。

不像例14.7中的数据只包括已知的购买珠宝的人的信息，决策树是一种树，用于将数据分成两个集合，我们可将这两个集合想像为“接受”和“拒绝”。就珠宝例子来说，数据是有关人的信息，接受集合将是我们认为可能会买珠宝的那些人，而拒绝集合则是我们认为可能不买珠宝的那些人。如果我们的预测是可靠的，则我们拥有一个目标群体，可将珠宝广告直接邮寄给他们。

决策树自身与图14-13类似，但是在叶结点没有真实数据。即，每一个内部结点有一个属性和一个作为阈值的值，结点的子结点要么是其他内部结点，要么是一个决策：接受或拒绝。
1090 一个给定的表示要分类的数据的元组，它在树中向下传递，每一步根据此元组在那个结点所提及的属性的值，决定向左走或向右走，直到到达一个决策结点。

树是通过结果已知的元组训练集来构造的。在珠宝例子中，我们使用顾客数据库，包括有关哪些顾客已购珠宝哪些未购的信息。数据挖掘问题是从这些数据中设计决策树，它能最可靠地决定，一个新顾客，我们已知其属性（年龄，收入等），是否可能购买珠宝。即，我们必须决定放在树根的最佳属性A，以及那个属性的一个最佳阈值v。那么，当决策向左走时，我们为$A<v$的那些顾客找到最佳属性和阈值；对$A \geq v$的那些顾客也进行同样的操作。每一层都面临相同问题，直到我们不再能向树添加有用结点（因为到达一个给定结点的训练数据实例太少，我们不能做出有用的决策）。我们设计每一个结点时所问的查询与很多数据的聚集有关，因此我

们能决定哪一个属性-阈值对划分数据，从而“接受”的最大部分去一边，“拒绝”去另一边。

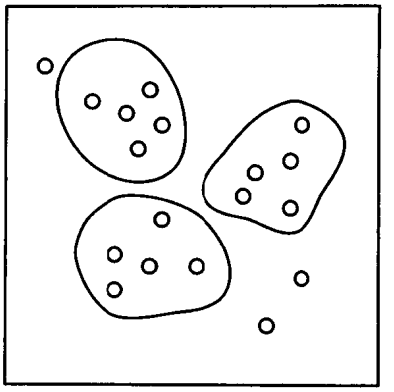


图20-23 二维数据的三个聚簇

聚簇

另一类数据挖掘问题与“聚簇”有关，我们试图将数据项分组，使其成为数目不多的几个组，而每一组都有一些重要的共性。图20-23表示了二维数据的聚簇，而实际中维的数目可能会很大。在这个图中，我们表示了最好的三个聚簇的近似轮廓，而一些点则离任何聚簇的中心都很远，可将这些点视为“孤立点”或与最近聚簇分组在一起。

以Web搜索引擎作为一种应用示例。Web搜索引擎经常找到几十万个匹配一个给定查询的文档。为帮助组织这些文档，搜索引擎可根据它们使用的单词将它们聚簇。如，搜索引擎可将文档放入一个空间中，对每一个可能用到的单词，在这个空间中都有一维与它对应，可能要排除最常用的单词（*中止词*），如“and”和“the”，这些词可能会出现在所有的 1091
文档中，并不告诉我们任何内容。根据任何特定词的出现频率（用小数表示）将文档放入空间。如，如果文档有1000个词出现，其中两个是“数据库”，则将文档放在与“数据库”对应的那一维的0.002坐标处。通过将文档在此空间聚簇，我们能得到谈论相同事情的文档组。例如，谈论数据库的文档可能会出现诸如“数据”、“锁”等词，而有关棒球的文档则不大可能出现这些词汇。

这里的数据挖掘问题是使用数据且选择聚簇的“均值”或“中心”。聚簇数目通常事先给定，尽管这个数目也可通过数据挖掘过程来选择。对这两种方法的任一种，如果使用朴素算法来选择中心，使一个点到离它最近的中心的距离最小，都包含许多查询，每一个这样的查询都要进行复杂的聚集。

20.6.2 寻找频繁项目集

现在，我们将看到一种有效地使用二级存储的算法的数据挖掘问题已经开发出来。这种寻找所谓“频繁项目集”的问题，就其主要应用是最易描述的：*对商场-购物筐数据的分析*。现在商场常在数据仓库中记录顾客同时买哪些物品，即顾客拎着盛满他或她所买物品的购物筐走向收款台，收银员将所有这些物品记录下来，作为单个事务的一部分。因此，即使我们对顾客一无所知，不能说出这位顾客是否会回来买其他物品，但我们确实知道一位顾客同时买的某些物品。

如果有些物品比预计的更经常同时出现在商场购物筐中，则商场可获得一些有关它的顾客可能如何在商场内寻找物品的信息。将这些物品放在商场中，以便顾客沿某些路径在商场内走动，并将有吸引力的物品沿这些路径放置。

例20.22 由几个人声明的一个著名例子是发现购买尿布的人总是要买啤酒。已经提出一些理论解释为什么这种关系是真实的，其中一种可能是购买尿布的人家里有婴儿，晚上不可能出去到酒吧，因此常常在家里喝啤酒。商场可以使用这个事实，即许多顾客会从商场卖尿布的地方走到卖啤酒的地方，或反向。聪明的商场人员会将啤酒和尿布临近放置，将土豆片放在中间。据说这三种物品的销量都有所增加。 □

我们可以用一个关系 1092

```
Baskets(basket, item)
```

表示商场购物筐数据，第一个属性是“购物筐ID”，即一个商场中的购物筐的惟一标识，第二个属性是在该购物筐内出现的某些物品的ID。注意，关系是否来自真正的商场购物筐这一点并不重要，它可以来自我们想发现关联项目的任何数据。例如，“购物筐”可以为文档，而物品为单词，在这种情况下，我们其实是在查找同时出现在许多文档中的单词。

频繁出现在商场购物筐数据中最简单的项目集形式是物品集合。物品集合$\{i_1, i_2, \cdots, i_n\}$的重要性可能相异。我们可能查找的最基本特性是所有物品都出现的购物筐数目是巨大的。对物品集合的支持度是其所包含的物品都出现的购物筐数目。找高支持度物品集合的问题是给定一个阈值s，找出所有那些物品集合，它们至少具有支持度s。

如果数据库中物品数目很多，即使我们将注意力局限在小集合中，如考虑物品对，计算所有物品对的支持度所需的时间也极长。因此，这种直接求解高支持度对的方法——对每一对物品i和j，计算支持度，如图20-24的查询所示——将是不可行的。这个查询包括将Baskets进行自连接，按结果元组中出现的两个物品对结果，去掉购物筐数低于阈值s的那些组。注意，WHERE子句中的条件I.item < J.item，是为防止对同一个物品对考虑两种顺序关系，或对同一个物品重复两次组成的对根本不考虑。

```
SELECT I.item, J.item, COUNT(I.basket)
FROM Baskets I, Baskets J
WHERE I.basket = J.basket AND
      I.item < J.item
GROUP BY I.item, J.item
HAVING COUNT(I.basket) >= s;
```

图20-24 寻找高支持度物品对的朴素方法

关联规则

一个更复杂的购物篮数据挖掘搜索是形式为$\{i_1,i_2,\cdots,i_n\} \Rightarrow j$的关联规则。对以这种形式存在的有用规则，我们所需要的两个属性可能是：

1. 可信度：在一个包含集合$\{i_1,i_2,\cdots,i_n\}$中所有物品的购物筐中找到物品j的概率大于某一阈值，如50%，即“至少50%买尿布的人买啤酒”。

2. 重要性：在包含集合$\{i_1,i_2,\cdots,i_n\}$中所有物品的购物筐中找到物品j的概率与在任一随机购物中找到物品j的概率相比，前者比后者高得多或低得多。用统计术语，即j与$\{i_1,i_2,\cdots,i_n\}$要么正相关，要么负相关。例20.22中的发现其实是规则{尿布}⇒啤酒有很大的重要性。

注意，即使一条规则有很高的可信度和重要性，若有关物品集合没有高支持度，它也是没有用的。原因在于如果支持度低，则规则实例数不大，这限制了使用这条规则的策略的益处。

20.6.3 A-Priori算法

当支持度阈值非常大，以至很少有对能达到时，对类似图20-24的查询，有一种优化方法可以显著降低其运行时间。将阈值设定为很高是合理的，因为成千上万的“对”列表不会有任何用处，我们想让数据挖掘查询将我们的注意力集中于数目很小的最佳候选对上。A-Priori算法基于以下观察：

1093

- 如果物品集合X的支持度为s，则X的任一子集的支持度必须至少为s。特别地，如果一对

物品，如{i, j}，出现在1000个购物筐中，则我们知道至少有1000个购物筐有物品i，至少有1000个购物筐有物品j。

上述规则的逆是，如果我们寻找具有支持度至少为s的物品对，可以将任何自身不出现在至少s个购物筐中的物品排除在外。A-Priori算法是这样回答与图20-24中相同的查询的：

1. 首先寻找“OK”物品集合，即那些本身出现在足够多的购物筐中的物品集合。

2. 只对OK集合中的每一个物品执行图20-24的查询。

图20-25中的两个SQL查询序列对A-Priori算法进行了概述：它首先计算candidates，即是Baskets关系的一个子集，这个子集中的物品本身有高支持度，而后对candidates执行自连接，如图20-24中的朴素算法。 1094

```
INSERT INTO OkBaskets
    SELECT *
    FROM Baskets
    WHERE item IN (
        SELECT item
        FROM Baskets
        GROUP BY item
        HAVING COUNT(*) >= s
    );

SELECT I.item, J.item, COUNT(I.basket)
FROM candidates I, candidates J
WHERE I.basket = J.basket AND
      I.item < J.item
GROUP BY I.item, J.item
HAVING COUNT(*) >= s;
```

图20-25 A-Priori算法在找高支持度对之前先找高支持度物品

例20.23 为说明A-Priori算法如何有效，考虑一个销售10 000种不同物品的超市。假设商场购物筐平均有20件物品，并假设数据库存储1 000 000个购物筐作为数据（与实际存储相比，这是个小数目）。

Baskets关系有20 000 000个元组，图20-24中的连接（朴素算法）有190 000 000对，这个数字表示100万个购物筐乘以$\binom{20}{2}$，即190个物品对，必须对所有这190 000 000个元组进行分组和计算。

但是，假设s是10 000，即购物筐的1%。多于20 000 000/10 000 = 2000件物品出现在至少10 000个购物筐中是不可能的，因为Baskets中只有20 000 000个元组，且出现在10 000个购物筐中的任何物品出现在至少10 000个元组中。因此，如果我们使用图20-25中的A-Priori算法，寻找高支持度物品的子查询不能产生多于2000件物品，产生的物品数可能比2000少得多。

我们不能确定Candidates有多大，因为在最坏的情况下，出现在购物筐中的所有物品将会出现在至少1%的购物筐中。但是实际上，如果s很大，Candidates将大大小于Baskets。为了讨论方便，假设Candidates中每个筐平均有10件物品，即为Baskets大小的一半，则在步骤2中，Candidates与自身的连接有1 000 000 × $\binom{10}{2}$=45 000 000个元组，比Baskets自连接的元组的1/4还少。从而，我们可以期望A-Priori算法的运行时间大约是朴素算法的运行时间的1/4。通常情况下，Candidates的元组数比Baskets元组数目的一半还少得多，并会有更大的性能提高，性能随参与连接的元组数目减少而成二次方地增长。 1095 □

20.6.4 习题

习题20.6.1 如图20-26，给定8个“购物篮”：

B_1 = {milk, coke, beer}
B_2 = {milk, pepsi, juice}
B_3 = {milk, beer}
B_4 = {coke, juice}
B_5 = {milk, Pepsi, beer}
B_6 = {milk, beer, juice, pepsi}
B_7 = {coke, beer, juice}
B_8 = {beer, pepsi}

图20-26 购物篮数据例子

* a) 集合{beer，juice}的支持度是多少？以篮子的百分比表示出来。

b) 集合{coke，pepsi}的支持度是多少？

* c) 给定了beer、milk的置信度是多少？（也就是说，关联规则{beer}⇒milk）

d) 给定了milk、juice的置信度是多少？

e) 给定beer和juice、coke的置信度是多少？

* f) 如果支持度的阈值是35%（也就是说，8个篮子中3个是必须的），哪一对项目是频繁的？

g) 如果支持度的阈值是50%，哪对项目是频繁的？

! **习题20.6.2** a-priori算法也可以用在寻找项目数大于2的频繁集合。回想规律:一个含有k个项目的集合X不会拥有至少s支持度，除非每个X的真子集有至少s支持度。特别地，X大小为$k-1$的子集必须都有至少s的支持度。所以，找到了大小为$k-1$的频繁项目集（至少是有s支持度的集合）后，可以定义大小为k的候选集合有k个项目，而且所有它的$k-1$大小的子集至少有s支持度。给定大小为$k-1$的项目集，请编写SQL查询，先计算出大小为k的候选集，然后计算大小为k的频繁项目集。

1096

习题20.6.3 使用习题20. 6. 1的篮子，回答下面问题:

a) 如果支持度的阈值是35%，候选三元集合是什么？

b) 如果支持度的阈值是35%，那些三元集合是频繁的？

20.7 小结

- *信息集成*：常常存在多种多样的数据库或其他包含相关信息的信息源。我们可以将这些数据源组合成一个。但是模式经常是异构的，这种不相容性表现在值的不同类型、代码或值的约定、概念的解释以及不同模式中表示的不同概念集合。
- *信息集成方法*：早期的方法涉及“联邦”，每一个数据库可以通过其他数据库能理解的术语来访问其他数据库。最近的方法涉及数据仓库，数据转换成全局模式，而后拷贝到数据仓库中。另一种可选用的方法是使用协调器，创建一个虚拟数据仓库，允许查询全局模式，然后将对全局模式的查询翻译为数据源的术语。
- *抽取器和包装器*：数据仓库和协调器要求在每一个数据源都有分别被称为抽取器和包装器的组件，其主要作用是在全局模式和数据源的局部模式之间翻译查询和查询结果。
- *包装器生成器*：设计包装器的一种方法是使用模板，模板描述如何将一个特定形式的查

询从全局模式翻译为局部模式。将这些模板制作成表，由引擎解释，引擎用来将查询与模板进行匹配。引擎也能将模板以各种方式进行组合、和/或执行额外工作如过滤以回答更复杂的查询。

- OLAP：数据仓库的一种重要应用是在数据源执行事务处理的同时，能够提出与所有或很多数据有关的复杂查询。这些查询通常涉及数据的聚集，被称作在线分析处理查询或OLAP查询。
- ROLAP和MOLAP：当建立用于OLAP的数据仓库时，将数据想像成立方体，维对应于要表示的数据的独立特征，这样做经常是很有用的。支持这样的数据视图的系统或者采用关系的观点（ROLAP，或关系OLAP系统），或者特化为数据立方体（MOLAP，或多维OLAP系统）。
- 星型模式：如果使用ROLAP方法，每一个数据元素（如一件物品的销售额）在一个称作事实表的关系中进行表示，而帮助解释沿每一维的值的信息（如哪种产品是物品1234？）存储在为每一维所建立的维表中。这种形式的数据库模式被称作星型模式，事实表是星的中心，而维表是点。
- 立方体操作符：当使用MOLAP方法时，一种沿维的所有子集对事实表进行预先聚集的操作符特别有用。它并不增加事实表所需空间，且极大地提高许多回答OLAP查询的速度。
- 维格和物化视图：在一些数据立方体实现中，一种比CUBE更强有力的方法是建立沿每一维所进行的聚集粒度的格（如，不同的时间单位，日、月和年）。然后，设计数据仓库时，将一些沿不同维以不同方法聚集的视图进行物化，并使用最接近的视图来回答一个给定的查询。
- 数据挖掘：数据仓库也可用于更广泛的问题，这些问题不仅涉及按命令进行聚集（如在OLAP查询中），而且搜索“正确的”聚集。数据挖掘的常见类型包括将数据聚簇成相似组，设计决策树来预测基于其他属性值的一个属性，寻找与同时出现的对或更大的值组合有关的关联规则。
- A-Priori算法：寻找关联规则的一种有效算法是使用A-Priori，这种技术使用这样一种事实，即如果一个集合经常发生，则它的任何子集也是如此。

20.8 参考文献

数据仓库和相关技术的最新综述在[9]、[3]和[7]中。[12]对联邦系统进行了综述。协调器的概念来自[14]。

协调器和包装器的实现，特别是包装器生成器包含在[5]中。

Modiator基于能力的优化在[11]、[15]中引入。

立方体操作符由[6]提出。通过物化视图实现立方体出现在[8]中。

[4]是数据挖掘技术的综述。[13]是数据挖掘的联机综述A-Priori算法来自[1]和[2]。

1. R. Agrawal, T. Imielinski, and A. Swami, “Mining association rules between sets of items in large databases,” *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1993), pp. 207–216.

2. R. Agrawal, and R. Srikant, “Fast algorithms for mining association rules,” *Proc. Intl. Conf. on Very Large Databases* (1994), pp. 487–499.

3. S. Chaudhuri and U. Dayal, "An overview of data warehousing and OLAP technology," *SIGMOD Record* **26**:1 (1997), pp. 65–74.

4. U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, *Advances in Knowledge Discovery and Data Mining*, AAAI Press, Menlo Park CA, 1996.

5. H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, V. Vassalos, J. D. Ullman, and J. Widom) The TSIMMIS approach to mediation: data models and languages, *J. Intelligent Information Systems* **8**:2 (1997), pp. 117–132.

6. J. N. Gray, A. Bosworth, A. Layman, and H. Pirahesh, "Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Proc. Intl. Conf. on Data Engineering* (1996), pp. 152–159.

7. A. Gupta and I. S. Mumick, *Materialized Views: Techniques, Implementations, and Applications*, MIT Press, Cambridge MA, 1999.

8. V. Harinarayan, A. Rajaraman, and J. D. Ullman, "Implementing data cubes efficiently," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1996), pp. 205–216.

9. D. Lomet and J. Widom (eds.), Special issue on materialized views and data warehouses, *IEEE Data Engineering Bulletin* **18**:2 (1995).

10. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom, "Object exchange across heterogeneous information sources," *Proc. Intl. Conf. on Data Engineering* (1995), pp. 251–260.

11. Y. Papakonstantinou, A. Gupta, and L. Haas, "Capabilities-base query rewriting in mediator systems," *Conference on Parallel and Distributed Information Systems* (1996). Available as:

    `http://dbpubs.stanford.edu/pub/1995-2`

12. A. P. Sheth and J. A. Larson, "Federated databases for managing distributed, heterogeneous, and autonomous databases," *Computing Surveys* **22**:3 (1990), pp. 183–236.

13. J. D. Ullman,

    `http://www-db.stanford.edu/~ullman/mining/mining.html`

14. G. Wiederhold, "Mediators in the architecture of future information systems," *IEEE Computer* **C-25**:1 (1992), pp. 38–49.

15. R. Yerneni, C. Li, H. Garcia-Molina, and J. D. Ullman) "Computing capabilities of mediators," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1999), pp. 443–454.

索　引[⊖]

A

Abiteboul，S. 21，187，1099
Abort（夭折），885，970，1017，1026
参见回滚（Rollback）
ABSOLUTE，361
Achilles，A. -C.，21
ACID properties（ACID性质），14
参见原子性（Atomicity），一致性（Consistency），持久性（Durability），孤立性（Isolation）
ACR schedule，调度
参见连锁回滚（cascading rollback）
Action（动作），340
ADA，350
ADD，294
Addition rule（附加规则），101
Address，地址
参见数据库地址（Database address），转存地址（Forwarding address），逻辑地址（Logical address），内存地址（Memory address），物理地址（Physical address），结构化地址（Structured address），虚拟内存（Virtual memory）
Address space（地址空间），509，582，880
Adornment（装饰），1066，1068
AFTER（AFTER），341～342
Aggregation（聚集），221～223，497～499
参见求平均（Average），计数（Count），分组（Group by），最大值（Maximum），最小值（Minimum），和（Sum）
Aggregation operator（聚集操作符），807
参见立方体操作符（Cube operator）
Agrawal，R.，1099
Aho，A. V.，474，530，726，789，852
Algebra（代数），192～193
参见关系代数（Relational algebra）
Algebraic law（代数定律），795～810，818～820
Alias，别名
参见AS
ALL，266，278，437
ALTER TABLE，294，334～335
Anomaly，异常
参见删除异常（Deletion anomoly），冗余（Redundancy），修改异常（Update anomaly）
Anonymous variable（匿名变量），466
ANSI，239
Antisemijoin（逆半连接），213
ANY，266
Application server（应用服务器），7
A-priori algorithm（A-priori算法），1093～1096
Apt，K.，502
Archive（备份），875～876，909～913
Arithmetic atom（算术原子），464
Armstrong，W. W.，129
Armstrong's axioms（Armstrong公理），99
参见增广律（Augmentation），自反律（Reflexivity），传递律（Transitive rule）
Array（数组），144，161，446
AS，242，428
ASC，251
Asilomar report（Asilomar报告），21
Assertion（断言），315，336～340
Assignment statement（赋值语句），*206*，444
Association rule（关联规则），1094
Associative law（结合律），220，555，795～796，819～820
Astrahan，M. M.，21，314，874
Atom（原子），463～464，788
Atomic type（原子类型），132，144
Atomicity（原子性），2，397，399～401，880，1024
Attribute（属性），25，31～32，62，136～138，

⊖ 索引中显示的页码与正文中边栏数字相呼应。

156 ~ 162, 166 ~ 167, 183 ~ 185, 255 ~ 256, 304, 456 ~ 458, 567, 575, 791, 794
参见依赖属性（Dependent attribute），维属性（Dimension attribute），输入属性（Input attribute），输出属性（Output attribute）
Attribute-based check（基于属性检查），327 ~ 330, 339
Augmentation（增广律），99, 101
Authorization（授权），383, 410 ~ 422
Authorization ID（授权 ID），410
Automatic swizzling（自动混写），584 ~ 585
Average（求平均），223, 279 ~ 280, 437, 727

B

Baeza-Yates, R.，663
Bag（包），144 ~ 145, 160 ~ 161, 166 ~ 167, 189, 192, 214 ~ 221, 446, 469 ~ 471, 728, 730, 796 ~ 798, 803
Bancilhon, F.，188, 502
Barghouti, N. S.，1044
Batini, Carlo, 60
Batini, Carol, 60
Bayer, R.，663
BCNF, 102, 105 ~ 112, 124 ~ 125
参见 Boyce-Codd 范式（Boyce-Codd normal form）
Beekmann, N.，711
Beeri, C.，129
BEFORE, 342
BEGIN, 368
Bentley, J. L.，711 ~ 712
Berenson, H.，424
Bernstein, P. A.，21, 129, 424, 916, 987
Binary large object（二进制大对象），595 ~ 596
Binary relationship（二元联系），25, 27 ~ 28, 32 ~ 33, 56
Binding columns（绑定列），390 ~ 392
Binding parameters（绑定参数），392 ~ 393
Bit（二进制位），572
Bit string（位串），246, 292
Bitmap index（位图索引），666, 702 ~ 710
Blair, R. H.，502
Blasgen, M. W.，785, 916
BLOB
参见二进制大对象（Binary large object）
Block（块）509
参见磁盘块（Disk block）
Block address，块地址
参见数据库地址（Database address）
Block header（块首部），576 ~ 577
Body（体）465
Boolean（布尔量），292
Bosworth, A.，1099
Bottom-up plan selection（自底向上的计划选择），843
Bound adornment，约束装饰
参见装饰（Adornment）
Branch and bound（分支界定），844
B-tree（B-树），16, 609, 611, 632 ~ 648, 652, 665, 670 ~ 671, 674, 762, 963 ~ 964, 999 ~ 1000
Bucket（桶），649, 652 ~ 653, 656, 676, 679, 685
参见间接桶（Indirect bucket）
Buffer（缓冲区），12 ~ 13, 506, 511, 725, 880, 882, 990
Buffer manager（缓冲区管理器），765 ~ 771, 850, 878 ~ 879
Buffer pool（缓冲池），766
Build relation（构建关系），847, 850
Buneman, P.，187
Burkhard, W. A.，712
Bushy tree（紧密树），848

C

Cache（高速缓存），507 ~ 508, 513
CALL（调用），366 ~ 367
Call-level interface，调用层界面
参见CLI
Candidate item（候选项），1094
Capabilities specification（能力规格说明），1066
Capability-based plan selection（基于能力的计划选择），1064 ~ 1070

Cartesian product，笛卡儿积
参见积（Product）
Cascade policy（连锁原则），321～322
Csacading rollback（级联回滚），992～994
Case insensitivity（大小写不敏感性），181，244
Catalog（目录），379～381
Cattell，R. G.，188，424，462，604
C/C++，133，350，385～386，443，570
CD-ROM
参见光盘（Optical disk）
Celko，J.，314
Centralized locking（集中封锁），1030
Ceri，S.，60，348，712，1044
Chamberlin，D. D.，314，874
Chandra，A. K.，502
Chang，P. Y.，874
Character set（字符集），382
Character string（字符串），569～571，650
参见 串（String）
Chaudhuri，S.，785，1099
CHECK
参见断言（Assertion），基于属性的检查（Attribute-based check），基于元组的检查（Tuple-based check）
Check-out-check-in（检出检入），1036
Checkpoint（检查点），875，890～895，912
Checksum（校验和），547～548
Chen，P. M.，566
Chen，P. P.，60
Chou，H.-T.，785
Class（类），132～133，135～136
CLI，349，385～393
Client（客户），7，382
Client-server system（客户-服务器系统），96～97，582
Clock algorithm（时钟算法），767～768
Close（关闭），720
Closure，of attributes（属性闭包），92～97，101
Closure，of sets of FD's（函数依赖(FD)集闭包），98
Cluster（簇），379～380
Clustered file（聚簇文件），624，759
Clustered relation（聚簇关系），717，728，759
Clustering（聚簇），1091～1092
Clustering index（聚簇索引），757～759，861～862
Cobol（Cobol语言），350
Cochrane，R. J.，348
CODASYL，4
Codd，E. F.，4，129～130，237，502，785
Code，编码
参见错误校正码（Error-correcting code）
Collation（核对），382
Collection（集合），570
Collection type（集合类型），133，145，444
参见数组（Array），包（Bag），字典（Dictionary），列表（List），集合（Set）
Combiner（合成器），1052～1053
Combining rule（组合规则），90～91
Comer，D.，663
Commit（提交），402，885～886，905，996，1023～1029
参见分组提交（Group commit），两阶段提交（Two-phase commit）
Commit bit（提交位），970
Communication cost（通信开销），1020
Commutative law（交换律），218，221，555，795～796，819～820
Compatibility matrix（相容性矩阵），943，946，948，959
Compensating transaction（补偿事务），1038～1041
Complementation rule（完备性规则），122
Complete name（完全名），383
Compressed bitmap（压缩位图），704～707
Concurrency（并发），880，888，917
参见锁机制（Locking），调度（Scheduler），可串行性（Serializability），时间戳（Timestamp），验证（Validation）
Concurrency control（并发控制），12～14，507
Condition（条件），340，371，374～376，790～791

参见选择（Selection），θ-连接（Theta-join），WHERE
Confidence（可信度），1094
Conflict（冲突），925～926
Conflict-serializability（冲突可串行性），918，926～930
Conjunct（合取），474
Connection（连接），382～383，393～394，412
Connection record（连接纪录），386～388
Consistency（一致性），879，933，941，947
Constraint（约束），47～54，231～236，315～340，576，876，879～880
参见依赖（Dependency）
Constructor function（构造函数），447
Containment of value sets（值集的包含），827
CONTINUE，375
Coordinator（协调器），1024，1031
Correctness principle（正确性原则），879～880，918
Correlated subquery（相关子查询），268～270，814～817
Cost-based enumeration（基于代价的枚举），821
参见连接排序（Join ordering）
Cost-based plan selection（基于代价的计划选择），835～837，1069
Count（计数），223，279～280，437
Crash，崩溃
参见介质故障（Media failure）
CREATE ASSERTION，337
CREATE INDEX，296～297，318～319
CREATE METHOD，451
CREATE ORDRERING，459
CREATE SCHEMA，380～381
CREATE TABLE，293～294，316
CREATE TRIGGER，341
CREATE TYPE，450
CREATE VIEW，302
Creating statement（创建语句），394
CROSS JOIN（交叉连接），271
Cross product，叉积
参见积（Product）
Cube operator（立方体操作符），1079～1082
CURRENT OF，358
Cursor（游标），355～361，370，396
Cycle（循环），928
Cylinder（同位磁道组），516，534～536，542～543，579

D

Dangling tuple（悬浮元组），228，323
Darwen，H.，314
Data cube（数据立方体），667，673，1047，1072～1073，1079～1089
Data disk（数据磁盘），552
Data file（数据文件），606
Data mining（数据挖掘），9，1047，1089～1097
Data source，数据源
参见源（Source）
Data structure（数据结构），503
Data type（数据类型），292
参见UDT
Data warehouse（数据仓库），9
参见仓库（Warehouse）
Database（数据库），2
Database address（数据库地址），579～580，582
Database administrator（数据库管理员），10
Database element（数据库元素），879，957
Database management system（数据库管理系统），1，9～10
Database programming（数据库程序设计），1，15，17
Database schema，数据库模式
参见关系数据库模式（Relational database schema）
Database state，数据库状态
参见数据库的状态（State, of a database）
Data-definition language（数据定义语言），10，292
参见ODL，模式（Schema）
Datalog，463～502
Data-manipulation language，数据操纵语言
参见查询语言（Query language）
DATA（数据），247，293，571～572

Date, C. J., 314
Dayal, U., 348, 1099
DB2, 492
DBMS
参见数据库管理系统（Database management system）
DDL
参见数据定义语言（Data-definition language）
Deadlock（死锁）, 14, 885, 939, 1009～1018, 1033
Decision tree（决策树）, 1090～1091
Decision -support query（决策支持查询）, 1070, 1089～1090
参见OLAP
DECLARE, 352～353, 356, 367
Decomposition（分解）, 102～105, 107～114, 123～124
Default value（默认值）, 295
Deferred constraint checking（延缓的约束检查）, 323～325
Deletion（删除）, 288～289, 410, 599～600, 615～619, 630, 642～646, 651～652, 708
参见修改（Modification）
Deletion anomaly, 删除异常
Delobel, C., 130, 188
DeMorgan's laws（摩根定律）, 331
Dense index（稠密索引）, 607～609, 611～612, 622, 636
Dependency, 依赖
参见约束（constraint）, 函数依赖（Functional dependency）, 多值依赖（Multivalued dependency）
Dependency graph（依赖图）, 494
Dependent attribute（依赖属性）, 1073
Dereferencing（参照）, 455～456
DESC, 251
Description record（描述纪录）, 386
Design（设计）, 15～16, 39～47, 70～71, 135
参见规范化（Normalization）
DeWitt, D. J., 785
Diaz, O., 348
Dicing（切块）, 1076～1078
Dictionary（字典）, 144, 161
Difference（差）, 192～194, 205, 215～216, 260～261, 278～279, 442, 472, 729～730, 737, 742～743, 747, 751～752, 755, 779, 798, 803, 833
参见EXCEPT
Difference rule（差规则）, 127
Digital versatile disk（数字易失磁盘）, 0
参见光盘（Optical disk）
Dimension attribute（维属性）, 1074
Dimension table（维表）, 1073～1075
Dirty buffer（脏缓冲区）, 900
Dirty data（脏数据）, 405～407, 970～973, 990～992
DISCONNECT, 383
Disk（磁盘）, 515～525
参见软盘（Floppy disk）
Disk access（磁盘存取）, 297～330
Disk assembly（磁盘配件）, 515～516
Disk block（磁盘块）, 12, 516, 531, 575～577, 579, 633, 694, 717, 733, 735～736, 765, 822, 879, 888
参见数据库地址（Database address）
Disk controller（磁盘控制器）, 517, 522
Disk Crash, 磁盘崩溃
参见介质故障（Media failure）
Disk failure（磁盘故障）, 546～563
参见磁盘崩溃（Disk crash）
Disk head（磁头）, 516
参见头配件（Head assembly）
Disk I/O（磁盘I/O）, 511, 519～523, 525～526, 717, 840, 852, 856
Disk scheduling（磁盘调度）, 538
Disk striping, 磁盘分割
参见RAID, 分割（RAID, Striping）
Diskette（软磁盘）, 519
参见软盘（Floppy disk）
DISTINCT, 277, 279, 429～430
Distributed database（分布式数据库）, 1018～1035
Distributive law（分配律）, 218, 221, 797

DML
参见数据操纵语言（Data-manipulation language）
Document retrieval（文档检索），626～630
Document type definition，文本类型定义
参见DTD
Domain（域），63，382
Domain constraint（域约束），47，234
Double-buffering（双缓冲），541～544
DRAM
参见动态随机存取存储器（Dynamic random-access memory）
Drill-down（下钻），1079
Driver（驱动器），393
DROP，294，297，307～308
DTD，178，180～185
Dump（转储），910
Duplicate elimination（消重复），221～222，225，278，725～727，737～740，747，750～751，755，771，773，779，805～806，818，833～834
参见DISTINCT
Duplicate-impervious grouping（不受重复影响的分组），807
Durability（持久性），2
DVD
参见数字易失磁盘（Digital versatile disk）
Dynamic hashing（动态散列），652
参见可扩展散列，线性散列（Extensible hashing，Linear hashing）
Dynamic programming（动态规划），845，852～857
Dynamic random-access memory（动态随机存取存储器），514
Dynamic SQL（动态 SQL），361～363

E

ELEMENT，444
Elevator algorithm（电梯算法），538～541，544
ELSE，368
ELSIF，368
Embedded SQL（嵌套SQL），349～365，384
END，368
End-checkpoint action（检查点结束操作），893
End-dump action（转储结束操作），912
Enity set（实体集），24～25，40～44，66～67，155
参见弱实体集（Weak entity set）
Entity/relationship model，实体/联系模型
参见E/R 模型（E/R model）
Enumeration（枚举），137～138，572
Environment（环境），379～380
Environment record（环境纪录），386～388
Equal-height histogram（等高直方图），837
Equal-width histogram（等宽直方图），836
Equijoin（等值连接），731，819，826
Equivalent sets of functional dependencies（等价函数依赖集），90
E/R diagram（E/R 图），25～26，50，53，57～58
E/R model（E/R 模型），16，23～60，65～82，173，189
Error-correcting code（错误校正码），557，562
Escape character（换码字符），248
Eswaran，K. P.，785，987
Event（事件），340
Event-condition-action rule，事件-条件-动作规则
参见触发（Trigger）
EXCEPT，260，442
Exception（例外处理），142，374～376
Exclusive lock（排他锁），940～942
EXEC SQL，352
EXECUTE，362，392，410～411
Executing queries/updates，in JDBC（在JDBC中执行查询/更新）394～395
Execution engine（执行引擎），10，15
EXISTS，266，437
EXIT，375
Expression tree（表达式树），202，308
Extended projection（外投影），222，226～227
Extensible hashing（可扩展散列），652～656
Extensible markup language，可扩展的标记语言
参见XML
Extensional predicate（扩展谓词），469

Extent（范围），151～152，170
Extractor，抽取器
参见包装器（Wrapper）

F

Fact table（事实表），670，1072～1075，1079
Fagin，R.，129～130，424，663
Faithfulness（忠实性），39
Faloutsos，C.，663，712
Faulstich，L. C.，188
Fayyad，U.，1099
FD
参见函数依赖（Functional dependency）
Federated databases（联邦数据库），1047，1049～1051
FETCH，356，361，389～390
Field（字段），132，567，570，573
FIFO
参见先进先出（First-in-first-out）
File（文件），504，506，567
参见顺序文件（Sequential file）
File system（文件系统），2
Filter（过滤器），844，860～862，868
Filter，for a wrapper（包装器过滤），1060～1061
Finkel，R. A.，712
Finkelstein，S. J.，502
First normal form（第一范式），116
First-come-first-served（先来先服务），956
First-in-first-out（先进先出），767～768
Fisher，M.，424
Flash memory（快闪存储器），514
Floating-point number，浮点数
参见实数（Real number）
Floppy disk（软盘），513
参见软磁盘（Diskette）
Flush log（刷新日志），886
FOR，372～375
For EACH ROW，341
Forign key（外键），319～322
Formal data cube（正式数据立方体），1072
参见数据立方体（Data cube）
Fortran，350
Forwarding address（转发地址），581，599
4NF，122～125
Fragment，of a relation（关系的片段），1020
Free adornment，自由装饰
参见装饰（Adornment）
Frequent itemset（频繁项目集），1092～1096
Friedman，J. H.，712
FROM，240，264，270，284，288，428，430，789～790
Full outerjoin，完全外连接
参见外连接（Outerjoin）
Function（函数），365～366，376～377
参见构造函数（Constructor function）
Functional dependency（函数依赖），82～117，125，231，233

G

Gaede，V.，712
Galliare，H.，502
Gap（间隙），516
Garcia-Molina，H.，188，566，1044，1099～1100
Generalized projection，广义投影
参见分组（Grouping）
Generator method（产生器方法），457～458
Generic SQL interface（类SQL界面），349～350，384
Geographic information system（地理信息系统），666～667
GetNext，720
Gibson，G. A.，566
Global lock（全局锁），1033
Global schema（全局模式），1051
Goodman，N.，916，987
Gotlieb，L. R.，785
Graefe，G.，785，874
Grammer（语法），789～791
Grant diagram（授权图），416～417
Grant option（授权选项），415～416

Granting privileges（授权优先权），414～416
Granularity，of locks（锁的粒度），957，958
Graph图
参见多图，先序图，等待图（Polygraph，Precedence graph，Waits-for graph）
Gray，J. N.，424，566，916，987～988，1044，1099
Greedy algorithm（贪婪算法），857～858
Grid file（网格文件），666，676～681，683～684
Griffths，P. P.，424
GROUP BY，277，280～284，438～441
Group commit（成组提交），996～997
Group mode（组模式），954～955，961
Grouping（分组），221～226，279，727～728，737，740～741，747，751，755，771，773，780，806～808，834
参见GROUP BY
Gulutzan，P.，314
Gunther，O.，712
Gupta，A.，237，785，1099
Guttman，A.，712

H

Haderle，D. J.，916，1044
Hadzilacos，V.，916，987
Haerder，T.，916
Hall，P. A. V.，874
Hamilton，G.，424
Hamming code（海明码），557，562
Hamming distance（海明距离），562
Handle（处理），386
Hapner，M.，424
Harel，D.，502
Harinarayan，V.，237，785，1099
Hash function（散列函数），649～650，652～653，656～657
参见分拆的散列函数（Partitioned hash function）
Hash join（散列连接），752～753，844，863
参见混合散列连接（Hybrid hash join）
Hash key（散列键），649
Hash table（散列表），649～661，665，749～757，770，773～774，779
参见动态散列法（Dynamic hashing）
HAVING，277，282～284，441
Head（磁头），465
Head assembly（磁头组合），515～516
Head crash，磁头损坏
参见介质故障（Media failure）
Header，首部
参见块首部，纪录首部（Block header，Record header）
Heap structure（堆结构），624
Held，G.，21
Hellerstein，J. M.，21
Heterogeneous sources（异构数据源），1048
Heuristic plan seletion（启发式计划选择），843～844
参见贪婪算法（Greedy algorithm）
Hill climbing（爬山法），844
Hinterberger，H.，712
Histogram（直方图），836～839
Holt，R. C.，1044
Hopcroft，H. E.，726，852
Horizontal decomposition（水平分解），1020
Host language（宿主语言），350～352
Howard，J. H.，129
Hsu，M.，916
HTML，629
Hull，R.，21
Hybrid hash join（混合散列连接），753～755

I

ID，183
Idempotence（幂等性），230，891，998
Identity（标识），555
IDREF，183
IF，368
Imielinski，T.，1099
Immediate constraint checking（立即约束检查），323～325
Immutable object（不变对象），133

Impedence mismatch（阻抗不匹配），350～351
IN，266～267，430
Inapplicable value（不可使用值），248
Incomplete transaction（未完成的事务），889，898
Increment lock（增量锁），946～949
Incremental dump（增量转储），910
Incremental update（增量更新），1052
Index（索引），12～13，16，295～300，318～319，605～606，757～764，1065
参见位图索引，B-树，簇索引，稠密索引，倒排索引，多维索引，二级索引，稀疏索引（Bitmap index，B-tree，Clustering index，Dense index，Inverted index，Multidimensionanl index，Sencondary index，Sparse index）
Index file（索引文件），606
Index join（索引连接），760～763，844，847，863
Index-scan（索引扫描），716，719～720，725，758～760，862，868
Indirect bucket（间接桶），625～630
Information integration（信息集成），8～9，19，173，175～177，1047～1049
参见联邦数据库，协调器，数据仓库（Federated databases，Mediator，Warehouse）
Information retieval，信息检索
参见文件检索（Document retrieval）
Information schema（信息模式），379
Information source，信息源
参见源（Source）
INGRESS，21
Inheritance（继承性），132，134～135
参见 Isa 联系，多重继承性，子类（Isa relationship，Multiple inheritance，Subclass）
Input action（输入动作），881，918
Input attribute（输入属性），802
Insensitive cursor（不敏感游标），360
Insertion（插入），286～288，410，598～599，615～620，630，639～642，650～651，653～660，677～679，691，697～698，708
参见更新（Modification）
Instance，of a relation（关系实例），64，66
Instance，of an entity set（实体集实例），27
Instance variable（实例变量），132
INSTEAD OF，344～345
Integer（整型），292～293，569，650
Intensional predicate（内涵谓词），469
Intention lock（意向锁），959
Interest（重要性），1094
Interesting order（有趣的排序），845
Interface（界面），152
Interior node（内部节点），174，633～635
Interleaving（交错），924
Intermediate collection（中间集），439
Intermittent failure（中间故障），546～547
Intersection（交运算），193～194，205，215～216，260～261，278～279，442，471～472，626，729～730，737，742～743，747，751～752，755，779，796～799，803，833
Intersection rule（交运算规则），127
INTO，355～356
Inverse（逆），555
Inverse relationship（逆关联），139～140
Inverted index（倒排索引），626～630
Isa relationship（Isa 联系），34，54，77
Isolation（孤立性），2
Isolation level（孤立层次），407～408
ISOWG3，313
Iterator（迭代符），720～723，728，733～734，871
参见流水线（Pipelining）

J

Java，393
Java database connectivity，Java数据库连接性
参见JDBC
JDBC，349，393～397
Join（连接），112～113，192～193，254，255，270～272，505～506
参见逆半连接，交叉连接，相等连接，自然连接，嵌套循环连接，外连接，连接选择性，半连接，θ-连接，zig-zag 连接（Antisemijoin，CROSS JOIN，Equijoin，Natural join，Nestedloop join，

Outerjoin, Selectivity, of a join, Semijoin, Theta-join, Zig-zag join)
Join, Ordering (连接顺序), 818, 847 ~ 859
Join tree (连接树), 848
Joined tuple (连接的元组), 198
Juke box (Juke 盒), 512

K

Kaiser, G. E., 1044
Kanellakis, P., 188
Kanellakis, P. C., 988
Katz, R. H., 566, 785
kd-tree (Kd-树), 666, 690 ~ 694
Kedem, Z., 988
Key (键), 47 ~ 51, 70, 84 ~ 88, 97, 152 ~ 154, 164, 316,
参见外键, 散列键, 主键, 查询键, 排序键 (Foreign key, Hash key, primary key, Search key, Sort key)
Kim, W., 188
Kitsuregawa, M., 785
Knowledge discovery in databases, 数据库中知识发现
参见数据挖掘 (Data mining)
Knuth, D. E., 604, 663
Ko, H. -P., 988
Korth, H. F., 988
Kossman, D., 785
Kreps, P., 21
Kriegel, H. -P., 711
Kumar, V., 916
Kung, H. -T., 988

L

Lampson, B., 566, 1044
Larson, J. A., 1099
Latency (延时), 519, 535
参见转动延时 (Rotational latency)
Lattice, of views (视图的格), 1085, 1087
Layman, A., 1099
Leader election (领导选举), 1027
Leaf (叶节点), 174, 633 ~ 634
Least fixedpoint (最小不动点), 481 ~ 486, 488, 499
Least-recently used (最近最少使用), 767 ~ 768
LEAVE, 371
Left outerjoin (左外连接), 228, 273
Left-deep join tree (左深度连接树), 848 ~ 849, 853
Left-recursion (左递归), 484
Legacy database (遗留数据库), 9, 175, 1065
Legality, of schedules (调度的合法性), 933 ~ 934, 941 ~ 942, 947
Lewis, P. M. II, 1045
Ley, M., 21
Li, C., 1100
LIKE, 246 ~ 248
Lindsay, B. G., 916, 1044
Linear hashing (线性散列法), 656 ~ 660
Linear recursion, 线性递归
参见非线性递归 (Nonlinear recursion)
Lisy (列表), 144 ~ 145, 161, 445 ~ 446
Literal (文字), 474
Litwin, W., 663
Liu, M., 502
Lock (锁), 400
参见全局锁 (Global lock)
Lock site (封锁节点), 1030
Lock table (锁表), 951, 954 ~ 957
Locking (封锁), 932 ~ 969, 978, 983 ~ 984, 1029 ~ 1035
参见排他锁, 渐增锁, 意向锁, 共享锁, 严格锁技术, 修改锁 (Exclusive lock, Increment lock, Intention lock, Shared lock, Strict locking, Update lock)
Log manager (日志管理器), 878, 884
Log record (日志记录), 884 ~ 885, 893
Logging (日志技术), 12 ~ 13, 875, 910, 913, 993, 996
参见逻辑日志技术, redo日志, undo日志, undo/redo日志 (Logical logging, Redo logging,

Undo logging，Undo/redo logging）
Logic，逻辑
参见Datalog，三值逻辑（Datalog，Three-valued logic）
Logcial address（逻辑地址），579～582
Logical logging（逻辑日志），997～1001
Logical query plan（逻辑查询计划），714～715，787～788，817～820，840～842
参见计划选择（Plan selection）
Lomet，D.，604，1099
Long-duration transaction（长事务），1035～1041，1071
Lookup（查找），609，613～614，638～639，659～660，676～677，680，691，707～708
Loop（循环），370～371
Lorie，R. A.，874，987
Lotus notes，175
Lozano，T.，712
LRU
参见最少最新使用的（Least-recently used）

M

Main memory（主存），508～509，513，525
Main-memory database system（主存数据库系统），510，765
Majority locking（大多数封锁），1034
Many-many relationship（多对多联系），28～29，140～141
Many-one relationship（多对一联系），27，29，56，140～141，154
Map table（映射表），579～580
Market-basket data（商场－购物筐数据），1092
参见关联规则（Association rule）
Materialization（实体化），859，863～867
Materialized view（物化视图），1083，1085～1087
Mattos，N.，348，502
Maximum（最大值），223，279，437
McCarthy，D. R.，348
McCreight，E. M.，663
McHugh，J.，187
McJones，P. R.，916
Mean time to failure（平均无故障时间），551
Media decay（介质损坏），546
Media failure（介质故障），546，549，876～877，909～913
Mediator（Mediator），1048，1053～1070
Megatron 2002 (imaginary DBMS)（Megatron 2002（虚构DBMS））），503～507
Megatron 737(imaginary disk)（Megatron 737（虚构磁盘））），536～537
Megatron 747(imaginary disk)（Megatron 747（虚构磁盘））），518～519，521～522
Megatron 777(imaginary disk)（Megatron 777（虚构磁盘））），524
Melkanoff，M. A.，130
Melton，J.，314，424
Memory address（存储器地址），582
Memory hierarchy（存储器层次），507～513
Memory size（内存大小），717，728，731
Merge-sort（归并排序），527～532
参见两阶段，多路归并排序（Two-phase，multiway merge-sort）
Merging nodes（归并节点），643～645
Metadata（元数据），13
Method（方法），133～134，141～143，156，167，171，45～452，569
参见产生器方法，Mutator方法（Generator method，Mutator method）
Minimum（最小值），223，279，437
Minker，J.，502
Mirror disk（镜像磁盘），534，537～538，544，552
Mode，of input or output parameters（输入或输出参数模式），142，365～366
Model，模型
参见E/R模型，面向对象模型，对象关系模型，关系模型，半结构化数据（E/Rmodel，Object-oriented model，Object-relational model，Relational model，Semistructured data）
Modification（更新），297，321～322，358～359

参见删除，插入，可修改视图，修改（Deletion，Insertion，Updatable view，Update）
Module（模块），38～385，412～413
参见PSM
Modulo-2 sum，模2和
参见奇偶校验位（Parity bit）
Mohan，C.，916，1044
MOLAP，1073
参见数据立方体（Data cube）
Monotonicity（单调性），497～499
Moore's law（摩尔定律），510
Moto-oka，T.，785
Multidimensional index（多维索引），665～666，673～674
参见栅格文件，多键索引，分拆散列函数，四叉树，R-树（Grid file，kd-tree，Multiple-key index，Partitioned hash function，Quard tree，R-tree）
Multidimensional OLAP（多维OLAP）
参见MOLAP
Multilevel index（多层索引），610～612
参见B-树（B-tree）
Multimedia data（多媒体数据），8
Multipass algorithm（多趟算法），771～774
Multiple disks（多磁盘），536～537，544
Multiple inheritance（多重继承），150～151
Multiple-key index（多键索引），666，687～690
Multiset，多集
参见Bag
Multi-tier architecture（多层体系结构），7
Multivalued dependency（多值依赖），118～127
Multiversion timestamp（多版本时间戳），975～977
Multiway merge-sort，多路归并排序
参见两阶段，多路归并排序（Two-phase，multiway merge-sort）
Multiway relationship（多路联系），28～30，32～33，148～149
Mumick，I. S.，502，1099
Mumps，350
Mutable object（易变对象），133
Mutator method（Mutator方法），457
Mutual recursion（互斥递归），494
MVD
参见多值依赖（Multivalued dependency）

N

Naqvi，S.，502
Natural join（自然连接），198～199，205，219，272，467～477，730～731，737，743～747，752～755，760～763，771～773，779～780，796，798～799，802，805，819，826～832，862～867
Navathe，S. B.，60
Nearest-neighbor query（最邻近查询），667～669，671～672，681，683，690，693
Negated subgoal（逆子目标），465，467
Nested relation（嵌套关系），167～169
Nested-loop join（嵌套循环连接），258，733～737，744，769～770，847，849～850
NEW ROW/TABLE，341～344
NEXT，361
Nicolas，J.-M.，237
Nievergelt，J.，663，712
Node（节点），174
Nonlinear recursion（非线性递归），484，492
Nonquiescent archieving（非静止转储），910～913
Nonquiescent checkpoint（非静止检查点），892～895，900～902，905～907
Nontrivial FD，非平凡函数依赖
参见平凡函数依赖（Trivial FD）
Nontrivial MVD，非平凡多值依赖
参见平凡多值依赖（Trivial MVD）
Nonvolatile storage，非易失性存储器
参见易失存储器（Volatile storage）
Normalization（规范化），16
Null character（空字符），571
Null value（空值），70，76，79～80，228，248～251，283，295，316，318，328，592～594，1049

参见设置空规则（Set-null policy）

O

Object（对象），78～79，133，135，170，569
Object broker（对象代理），578
Object definition language，对象定义语言
参见ODL
Object identifier（对象标识符），569
Object identify（对象标识），132～133，135，167，171
参见引用列（Reference column）
Object query language，对象查询语言
参见OQL
Object-oriented database（面向对象数据库），765
Object-oriented model（面向对象模型），132～135，170～171，173
参见对象关系模型，ODL，OQL（Object-relational model，ODL，OQL）
Object-relational model（对象关系模型），8，16，131，166～173，425，449～461
Observer method（观察方法），456～457
ODBC
参见CLI
ODL，16，135～166，172，569
ODMG，187
Offset（偏移量），572～573
Offset table（偏移量表），580～581，598
OID
参见对象表示符（Object identifier）
OLAP（联机分析处理），1047，1070～1089
参见MOLAP，ROLAP
OLD ROW/TABLE，341～344
Olken，F.，785
OLTP，1070
ON，271
On-demand swizzling（立即混写），585
O'Neil，E.，424
O'Neil，P.，424，712
One-one relationship（一对一联系），28～29，140～141
One-pass algorithm（一趟算法），722～733，850，862
On-line analytic processing，联机分析处理
参见OLAP
On-line transaction processing，联机事务处理
参见OLTP
Open，720
Operand（算子），192
Operator（操作符），192
Optical disk（光盘），512～513
Optimistic concurrency control，乐观并发控制
参见时间戳，验证（Timestamp，Validation）
Optimization，优化
参见查询优化（Query optimization）
OQL，425～449，570
ORDER BY，251～252，284
Ordering relationship，for UDT（UDT的有序联系），458～460
Outerjoin（外连接），222，228～230，272～274
Output action（输出动作），881，918
Output attribute（输出属性），802
Overflow block（溢出块），599，616～617，619，649，656
Overloaded method（过载方法），142
Ozsu，M. T.，1045

P

Pad character（填充符号），570
Page，509
参见磁盘块（Disk block）
Palermo，F. P.，874
Papadimitrion，C. H.，987，1044
Papakonstantinou，Y.，188，1099
Parallel computing（并行计算），6～7，775～782，983
Parameter（参数），392，396～397
Parity bit（奇偶校验位），548，552～553
Parse tree（语法分析树），788～789，810
Parser（分析器），713～715，788～795
Partial-match query（部分匹配查询），667，681，684，688～689，692
Partition attribute（分拆属性），438
Partitioned hash function（分段散列函数），666，

682 ~ 684
Pascal，350
Path expression（路径表达式），426 ~ 428
Paton，N. W.，348
Pattern（模板），791
Patterson，D. A.，566
PCDATA，180
Pelagatti，G.，1044
Pelzer，T.，314
Percentiles，百分数
参见等高直方图（Equal-height histogram）
Persistence（持久性），1，301
Persistent stored modules。永久存储模块
参见PSM
Peterson，W. W.，664
Phantom（幻像），961 ~ 962
Physical address（物理地址），579，582
Physical query plan（物理查询计划），714 ~ 715，787，821，842 ~ 845，859 ~ 872
Piateesky-Shapiro，G.，1099
Pinned block（被固定的块），586 ~ 587，768，995
Pipelining（流水线），859，863 ~ 867
参见迭代器（Iterator）
Pippenger，N.，663
Pirahesh，H.，348，502，916，1044，1099
Plan selection（计划选择），1022
参见算法选择，基于能力的计划选择，基于代价的枚举，基于代价的计划选择，启发式计划选择，物理查询计划，自顶向下计划选择（Algorithm selection，Capability-based plan selection，Cost-based enumeration，Cost-based plan selection，Heuristic plan selection，Physical query plan，Top-down plan selection）
Platter（母板），515，517
PL/I，350
Pointer swizzling，指针混写
参见混写（Swizzling）
Polygraph（多图），1004 ~ 1008
Precedence graph（优先图），926 ~ 930
Precommitted transaction（预提交事务），1025
Predicate（谓词），463 ~ 464
Prefetching，预提取
参见双缓冲技术（Double-buffering）
PREPARE，362，392
Prepared statement（准备语句），394 ~ 395
Preprocessor（预处理器），793 ~ 794
Preservation，of FD's（保持函数依赖），115 ~ 116，125
Preservation of value sets（值集的保持），827
Price，T. G.，874
Primary index（主索引），622
参见稠密索引，稀疏索引（Dense index，Sparse index）
Primary key（主键），48，316 ~ 317，319，576，606
Primary-copy locking（主副本封锁），1032 ~ 1033
PRIOR，361
Privilege（优先权），410 ~ 421
Probe relation（试探关系），847，850
Procedure（过程），365，376 ~ 377
Product（积），192 ~ 193，197 ~ 198，218，254 ~ 255，476，730，737，796，798 ~ 799，803，805，832
Projection（投影），112 ~ 113，192 ~ 193，195，205，216 ~ 217，242，245，473，724 ~ 725，737，802 ~ 805，823，832，864
参见扩展投影，下推投影（Extended projection，Pushing projection）
Projection，of FD's（函数依赖的投影），98 ~ 100
Prolog，501
Pseudotransitivity（伪传递性），101
PSM，349，365 ~ 378
PUBLIC，410
Pushing projections（下推投影），802 ~ 804，818
Pushing selections（下推选择），797，800 ~ 801，818
Putzolo，F.，566，988

Q

Quad tree（四叉树），666，695 ~ 696
Quantifier，量词

参见ALL，ANY，EXISTS
Quass，D.，187，237，712，785，1099
Query（查询），297，466，504-505
参见决策支持查询，查找，最邻近查询，部分匹配查询，范围查询，Where-am-I查询（Decision-support query，Lookup，Nearest-neighbor query，Partial-match query，Range query，Where-am-I query）
Query complier（查询编译器），10，14～15，713～715，787
参见查询优化（Query optimization）
Query execution（查询执行），713，870～871
Query language（查询语言），2，10
参见Datalog，OQL，关系代数，SQL（Datalog，OQL，Relational algebra，SQL）
Query optimization（查询优化），15，714～715
参见计划选择（Plan selection）
Query plan（查询计划），10，14
参见逻辑查询计划，物理查询计划，计划选择（Logical query plan，Physical query plan，Plan selection）
Query processing（查询处理），17～18，506
参见执行引擎，查询编译器（Execution engine，Query compiler）
Query processor，查询处理器
参见查询编译器，查询执行（Query compiler，Query execution）
Query rewriting（查询重写），714～715，788，810～821
参见代数定律（Algebraic law）
Quicksort（快速排序），527
Quotient（商），213

R

RAID，551～563，876～877
Rajaraman，A.，1099
RAM disk（RAM磁盘），514
Ramakrishnan，R.，502
Random-access memory（随机访问存储器），508
Range query（范围查询），638～639，652，667，673，681，689，692～693
Raw-data cube（原始数据立方体），1072
参见数据立方体，事实表（Data cube，Fact table）
Read action（读动作），881，918
READ COMMITTED，407～408
Read lock，读锁
参见共享锁（Shared lock）
Read set（读集合），979
Read time（读时间），970
READ UNCOMMITTED，407～408
Read-locks-one-write-locks-all（读封锁—个写封锁所有），1034
Read-only transaction（只读事务），403～404，958
Real number（实数），293，569
Record（记录），567，572～577，598～601
参见滑动记录，生成记录，特征字段，可变格式记录，变长记录（Sliding record，Spanned record，Tagged field，Variable-format record，Variable-length record）
Record address，记录地址
参见数据库地址（Database address）
Record fragment（记录片段），595
Record header（记录首部），575～576
Record structure，记录结构
参见 Struct
Recoverable schedule（可恢复调度），992～994
Recovery（恢复），12，875，889～890，898～902，904～905，913，990，1000～1001，1026～1028
Recovery manager（恢复管理器），879
Recursion（递归），463，480～500
Redo logging（redo日志），887，897～903
Redundancy（冗余），39～40，103，118～119，125
Redundant arrays of independent disks，独立磁盘的冗余阵列
参见RAID
Redundant disk（冗余磁盘），552
Reference（引用），133，167，169～171，452，455～456

Reference column（引用列），452～454
REFERENCES，320，410
REFERENCING，341
Referential integrity（引用完整性），47，51～53，232
参见外键（Foreign key）
Reflexivity（自反性），99
Relation（关系），61，303，463，791，793～794
参见创建关系，维表，事实表，试探关系，表，视图（Build relation，Dimension table，Fact table，Probe relation，Table，View）
Relation schema（关系模式），62，66，73，194，292～301
Relational algebra（关系代数），189～237，259～260，463，471～480，795～808，811
Relational atom（关系原子），464
Relational database schema（关系数据库模式），24，62，190～191，379～381，383
Relational model（关系模型），4～5，61～130，155～164，173
参见嵌套关系，对象关系模型（Nested relation，Object-relational model）
Relational OLAP，关系OLAP
参见ROLAP
Relationship（联系），25，31～32，40～44，67～70，138～141，162～163
参见二元联系，Isa联系，多对多联系，多对一联系，多路关系，一对一联系，支持联系（Binary relationship，Isa relationship，Many-many relationship，Many-one relationship，Multiway relationship，One-one relationship，Supporting relationship）
Relationship set（联系集），27
RELATIVE，361
Renaming（重命名），193，203～205，304～305
REPEAT，373
REPEATABLE READ，407～408
Repeating field（重复字段），590～593
Replicated data（多副本数据），1021，1031～1032
Resilience（回复性），875
RETURN，367
Reuter，A.，916，988
Revoking privileges（收权），417～421
Right outerjoin（右外连接），228，273
Right-deep join tree（右深度连接树），848
Right-recursion（右递归），484
Rivest，R. L.，712
Robinson，J. T.，712，988
ROLAP，1073
Role（角色），29～31
Rollback（回滚），402，404～405
参见夭折，连锁回滚（Abort，Cascading rollback）
Roll-up（上卷），1079
Root（根），174，633
Root tag（根标记），179
Rosenkrantz，D. J.，1045
Rotation，of disk（磁盘转动），517
Rotational latency（转动延时），520，540
参见延时（Latency）
Rothnie，J. B. Jr.，712，987
Roussopoulos，N.，712
Row-level trigger（行级触发），342
R-tree（R-树），666，696～699
Rule（规则），465～468
Run-length encoding（分段长度编码），704～707

S

Safe rule（安全规则），467，482
Saga（系列记载），1037～1040
Sagiv，Y.，1099
Salem，K.，566，1044
Salton，G.，664
Schedule（调度），918，923～924
参见串行调度，可串调度（Serial schedule，Serializable schedule）
Scheduler（调度管理程序），917，932，934～936，951～957，969，973～975，979～980
Schema（模式），49，85，167，173，504，572，575
参见数据库模式，全局模式，关系模式，关系数据库模式，星型模式（Database schema，Global schema，Relation schema，Relational

database schema，Star schema）
Schneider，R.，711
Schwarz，P.，916，1044
Scope of names（名字作用范围），269
Scrolling cursor（卷型游标），361
Search key（查找键），605～606，612，614，623，665
参见散列键（Hash key）
Second normal form（第二范式），116
Secondary index（辅助索引），622～625
参见倒排索引（Inverted index）
Secondary storage（二级存储器），6，510～513
参见磁盘，光盘（Disk，Optical disk）
Second-chance algorithm，second-chance 算法
参见时钟算法（Clock algorithm）
Sector（扇区），516，518
Seeger，B.，711
Seek time（查找时间），519～520，535，540
SELECT，240～243，284，410，428，431～432，789～790
参见单元组选择（Single-row select）
Selection（选择），192～193，196，205，217～218，221，241，243，245～246，473～475，724～725，737，758～760，770～779，797～801，805，818，823～826，844，860～862，864，868
参见过滤器，下推选择，二变量选择（Filter，Pushing selections，Two-argument selection）
Selectivity of a join（连接的选择），858
Self-describing data（自描述数据），175
Selinger，P. G.，874
参见Griffiths，P. P.
Selinger-style optimization（Selinger风格优化），845，857
Sellis，T. K.，712
Semantic analysis，语义分析
参见预处理器（Preprocessor）
Semijoin（半连接），213
Semistructured data（半结构化数据），16，131，173～178
Sequential file（顺序文件），606～607
Serial schedule（串行调度），919～920
Serializability（串行化），397～400，407，918，921~923，927，989～990
参见冲突串行化，视图串行化（Conflict-serializability，View-serializability）
Seriabile schedule（可串行化调度），920～921，994
Server（服务器），7，382
参见客户服务器系统（Client-server system）
Session（会话），384，413
SET，289，325，367～368，381，383～384，404，729，797～798，803
Set type（系型），144～145，158～160，166～167，217，446
Sethi，R.，789
Set-null policy（置空原则），322
Sevcik，K.，712
Shapiro，L. D.，785
Shared disk（共享磁盘），776，778
Shared lock（共享锁），940～942，956
Shared memory（共享存储器），775～776，778
Shared variable（共享变量），352～354
Shared-nothing machine（无共享机器），776～777
Shaw，D. E.，785
Sheth，A.，1099
Signature（签名），141～142
Silberschatz，A.，988
Silo（磁带仓），512
Simon，A. R.，314
Simple projection（简单投影），802
Simplicity（简化），40
Single-row select（单元组选择），354，370
Single-value constraint（单值约束），47，51
参见函数依赖，多对一联系（Functional dependency，Many-one relationship）
Size estimation（大小的估计），822～834，836～839
Size，of a relation（关系大小），717，822，840，842
Skeen，D.，1045
Slicing（切片），1076～1078

Sliding records（移动记录），616
Smalltalk，132
Smith，J. M.，874
Smyth，P.，1099
Snodgrass，R. T.，712
Sort join（排序连接），743～747，844，862～863
Sort key（分类键），526，606，636
Sorted file，排序文件
参见顺序文件（Sequential file）
Sorted sublist（分类的子列表），529，738，770
Sorting（排序），222，227～228，526～532，737～749，755～756，771～773，845
参见ORDER BY，UDT的有序联系（ORDER BY，Ordering relationship for UDT）
Sort-scan（排序扫描），716～717，719，721～722，868
Source（数据源），1047
Spanned record（跨块记录），594～595
Sparse index（稀疏索引），609～612，622，636
Splitting law（分解律），797～798
Splitting nodes（节点分裂），640～642，645，698～699
Splitting rule（分解规则），90～91
SQL，4～5，131，189，239～424，449～461，492～500，789～793
SQL agent（SQL代理），385
SQLSTATE，352～353，356，374
Strikant，R.，1099
Stable storage（稳定存储器），548～550
Star schema（星型模式），1073～1075
Start action（开始操作），884
START TRANSACTION，402
Start-checkpoint action（检查点开始操作），893
Start-dump action（转储开始操作），911
Starvation（饿死），1016～1017
State，of a database（数据库状态），879，1039
Statement record（语句记录），386～388
Statement -level trigger（语句级触发），342
Statistics（统计量），13，836，839～840
参见直方图（Histogram）
Stearns，R. E.，1045
Stemming（抽取词干），629
Stern，R. C.，210
Stonebraker，M.，21，785，1045
Stop word（无用词），629
Storage manager（存储管理器），12，17～18
参见缓冲区（Buffer）
Stratified negation（分层非），486～490，494～496
Strict locking（严格封锁），994
String（字符串），245～247，292
参见位串（Bit string）
Stripe（分割），676
Striping（分割），596
Strong，H. R.，663
Struct，132～133，137～138，144～145，157，166～167，431，446，568
Structured address（结构地址），580～581
Sturgis，H.，566，1044
Subclass（子类），33～36，76～80，149～151
Subgoal（子目标），465
Subquery（子查询），264～276，431～432，812～819
参见相关子查询（Correlated subquery）
Subrahmanian，V. S.，712
Suciu，D.，187，188，1099
Sum（求和），223，279，437
Superkey（超键），86，105
Support（支持度），1093
Supporting relationship（支持联系），56，72，74～75
Swami，A.，1099
Swizzling（混写），581～586
Syntactic category（语法类），788～789
Syntax analysis，语法分析
参见语法分析程序（parser）
System failure（系统故障），876～877
System R（系统R），21，314，874

T

Table（表），293，301，303
参见关系（Relation）
Table-scan（表扫描），716，719，721，861～862，

867 ~ 868
Tag（标记），178
Tagged field（标记字段），593
Tanaka，H.，785
Tape（磁带），512
Template（模板），1058 ~ 1059
Tertiary memory（三层主存储器），512 ~ 513
Tertiary storage（三层存储器），6
Thalheim，B.，60
THEN，368
Theta-join（θ-连接），199 ~ 201，205，220，477，731，796 ~ 799，802，805，819 ~ 820，826 ~ 827
Theta-outerjoin（θ-外连接），229
Third normal form，第三范式
参见3NF
Thomas，R. H.，1045
Thomasian，A.，988
Thrashing（颠簸），766
3NF，114 ~ 116，124 ~ 125
Three-valued logic（三值逻辑），249 ~ 251
Thuraisingham，B.，988
TIME，247 ~ 248，293，571 ~ 572
Timeout（超时），1009 ~ 1010
TIMESTAMP，248，575，577，969 ~ 979，984，1014 ~ 1017
Tombstone（删除标记），581，600
Top-down plan selection（自顶向下的计划选择），843
TPMMS
参见两阶段，多路归并排序（Two-phase，multiway mergesort）
Track（磁道），515 ~ 517，579
Traiger，I. L.，987 ~ 988
Training set（训练集），1091
Transaction（事务），1 ~ 2，12，17 ~ 19，397 ~ 409，877 ~ 883，923 ~ 924，1020 ~ 1021
参见不完全事务，长事务（Incomplete transaction，Long-duration transaction）
Transaction component（事务组件），1020
Transaction manager（事务管理器），878，917
Transaction processing，事务处理
参见并发，死锁，锁技术，日志技术，调度（Concurrency，Deadlock，Locking，Logging，Scheduling）
Transfer time（转移时间），520，535
Transitive rule（传递规则），96 ~ 97，121
Translation table（转换表），582 ~ 583
Tree，树
参见B-树，Bushy树，决策树，表达式树，连接树，kd-树，左深度连接树，语法分析树，四叉树，右深度连接树，R-树（B-tree，Bushy tree，Decision tree，Expression tree，Join tree，kd-tree，Left-deep join tree，Parse tree，Quad tree，Right-deep join tree，R-tree）
Tree protocol（树协议），963 ~ 969
Trigger（触发），315，336，340 ~ 345，410 ~ 411，876，879
Trivial FD（平凡函数依赖），92，105
Trivial MVD（平凡多值依赖），120 ~ 122，127
Tuple（元组），62 ~ 63，170
参见悬浮元组（Dangling tuple）
Tuple variable（元组变量），256 ~ 257
Tuple-based check（基于元组的检查），327，330 ~ 331，339
Turing-complete language（图灵完备语言），189
Two-argument selection（两参数选择），812 ~ 817
Two-pass algorithm（两趟算法），737 ~ 757
Two-phase commit（两阶段提交），1024 ~ 1028
Two-phase locking（两阶段封锁），936 ~ 938
Two-phase，multiway merge-sort（两阶段多路归并排序），0，528 ~ 532，536 ~ 537
Type（类型），794，1049
Type constructor（类型构造符），132
Type system（类型系统），132 ~ 133，144 ~ 146，171

U

UDT，449 ~ 452
Ullman，J. D.，21，130，474，502，530，726，789，852，1099 ~ 1100
UNDER，410 ~ 411
UNDO，375
Undo logging（undo日志），884 ~ 896

Undo/redo logging（undo/redo日志），887，903～909
Union（并），192～194，215～217，260～262，278，442，472，722～723，728～729，741，747，751～752，755，779，796～798，803，833
Union rule（并规则），127
UNIQUE，316～319
UNKNOWN，249～251
Unknown value（未知值），248
Unstratified negation，不分层非
参见分层非（Stratified negation）
Unswizzling（解除混写），586
Updatable view（不可更新视图），305～307
Update（修改），289～290，410，601，615～616，709，1052
参见更新（Modification）
Update anomaly（更新异常），103
Update lock（更新锁），945～946
Update record（更新记录），885～886，897，903
Upgrading locks（升级锁），943～945，957
参见更新锁（Update lock）
USAGE，410
User-defined type，用户定义类型
参见UDT
Uthurusamy，R.，1099

V

Valduriez，P.，1045
Valid XML（有效XML），178～179
Validation（有效性确认），969，979～985
Value count（值计数），719，822，840
VALUES，286
Van Gelder，A.，502
VARCHAR，292
Variable-format record（可变格式记录），590，593～594
Vatiable-length record（变长记录），570～571，589～594，998～999
Vassalos，V.，1099
Vertical decomposition（垂直分解），1020
Vianu，V.，21
View（视图），301～312，345，1052
参见实例化视图（Materialized view）
View-serializability（视图可串行性），1003～1009
Virtual memory（虚存），509～510，578
Vitter，J. S.，566
Volatile storage（易失存储器），513～514

W

Wade，B. W.，424
Wait-die（等待-死亡），1014～1017
Waiting bit（等待位），955
Waits-for graph（等待图），1010～1012
Walker，A.，502
Warehouse（数据仓库），1048，1051～1053，1071
Warning protocol（警示协议），958～961
Weak entity set（弱实体集），54～59，71～75，154
Weiner，J. L.，187
Well-formed XML（构造良好的XML），178～180
Westwood，J. N.，210
WHEN，340，342
WHERE，240～241，243～244，264，284，288，428～429，789
Where-am-I query（Where-am-I查询），667，697
WHILE，373
White，S.，424
Widom，J.，187～188，348，1099
Wiederhold，G.，604，1100
WITH，492～493
Wong，E.，21，874
Wood，D.，785
Workflow（工作流），1036
World-Wide-Web consortium（WWW国际财团），187
Wound-wait（伤害－等待），1014～1017
Wrapper（包装器），1048，1057～1064
Wrapper generator（包装器生成器），1059～1060
Write action（写动作），881，918
Write failure（写失败），546，550

参见系统故障（System failure）
Write lock，写锁
参见排他锁（Exclusive lock）
Write set（写集合），979
Write time（写时），970
Write-ahead logging rule（提前写日志规则），897
参见redo日志（Redo logging）
Write-through cache（直写），508

X

XML，16，131～132，173，178～186，629

Y

Yerneni，R.，1100
Youssefi，K.，874

Z

Zaniolo，C.，130，712
Zicari，R.，712
Zig-zag join（Zig-zag连接），762～763
Zip disk（Zip磁盘），513
Zipfian distribution（Zipfian 分布），632，825