

Database:
Principles, Programming, and Performance
(Second Edition)

数据库原理、
编程与性能

(原书第2版)

(美) Patrick O'Neil
Elizabeth O'Neil 著

周傲英 俞荣华 季文祯 钱卫宁 等译



机械工业出版社
China Machine Press



MORGAN
KAUFMANN

数据库原理、 编程与性能

(原书第2版)

**Database:
Principles, Programming,
and Performance**

(Second Edition)

ISBN 7-111-09310-0

9 787111 093107



华章图书

www.china-pub.com

北京市西城区百万庄南街1号 100037

购书热线: (010)68995259, 8006100280 (北京地区)

ISBN 7-111-09310-0/TP · 2094

定价: 55.00 元



北京华章图文信息有限公司

(原书第2版)

数据库原理、 编程与性能

*Database:
Principles, Programming,
and Performance* (Second Edition)

(美) Patrick O'Neil
Elizabeth O'Neil 著

周傲英 俞荣华 季文簇 钱卫宁 等译



机械工业出版社
China Machine Press

本书是在波士顿马萨诸塞大学数据库入门和提高等一系列教材的基础上写成的，从理论和实际两方面详细介绍了数据库的设计和实现。本书把重点放在对象-关系模型上，介绍了ORACLE、DB2和INFORMIX系统中普遍采用的新概念，并在结合数据库的基本原理和主要的商业数据库产品的基础上介绍了SQL-99。本书涵盖了关系数据库理论、SQL语言、数据库设计以及数据库完整性、视图、安全性、索引、事务管理等各个方面的内容。

本书是计算机专业学生的一本理想教材，对于数据库设计者和实现者也是一本优秀的参考书。

Patrick O'Neil and Elizabeth O'Neil: Database-Principles, Programming, and Performance Second Edition

Copyright ©1994, 2001 by Morgan Kaufmann Publishers, Inc.

Chinese edition published by arrangement with Morgan Kaufmann.

All rights reserved.

本书中文简体字版由美国Morgan Kaufmann公司授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

本书版权登记号：图字：01-2000-2808

图书在版编目（CIP）数据

数据库原理、编程与性能 / (美) 奥尼尔 (O'Neil, P.) , (美) 奥尼尔 (O'Neil, E.) 著；周傲英等译。—北京：机械工业出版社，2002.1
(国外经典教材)

书名原文：Database-Principles, Programming, and Performance Second Edition

ISBN 7-111-09310-0

I. 数… II. ①奥…②奥… ③周… III. 数据库系统 IV. TP311.13

中国版本图书馆CIP数据核字（2001）第067409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：杨海玲

北京市密云县印刷厂印刷·新华书店北京发行所发行

2002年1月第1版第1次印刷

787mm×1092mm 1/16 · 38.25印张

印数：0 001 ~ 5 000册

定价：55.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

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、研究及收藏。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：针对本科生的核心课程，剔抉外版菁华而成“国外经典教材”系列；对影印版的教材，则单独开辟出“经典原版书库”；定位在高级教程和专业参考的“计算机科学丛书”还将保持原来的风格，继续出版新的品种。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。

“国外经典教材”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本科教学

度身订造的。在广泛地征求并听取丛书的“专家指导委员会”的意见后，我们最终选定了这20多种篇幅内容适度、讲解鞭辟入里的教材，其中的大部分已经被M.I.T.、Stanford、U.C. Berkley、C.M.U.等世界名牌大学采用。丛书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) 68995265

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

(按姓名笔画顺序)

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

译 者 序

本书是一本出色的关于数据库原理与应用的教材。它是作者在波士顿马萨诸塞大学的数据库入门和提高这两门课程教材的基础上编纂而成的。和其他数据库系统教材不同，本书理论与实践并重，每一部分都从基本概念入手，并详细地阐述各个概念在真实的数据库系统中的实现与应用。同时，书中详细比较了各种数据库管理系统的技术差异，并在附录A中给出了两种数据库管理系统的使用方法。于是，学生可以在学习了数据库系统的基本原理之后快速地运用所学的知识。这种对实践的重视同样体现在习题的设计中。作为一本教材，本书在详细介绍了数据库经典技术的同时准确地把握了数据库系统的发展趋势，着重介绍对象-关系模型。于是，在阅读本书以后，读者可以很容易地充分利用现有主流数据库管理系统提供的主要功能。

本书的作者Patrick O'Neil和Elizabeth O'Neil教授都是数据库界知名的学者。他们在数据库领域有很多重要的研究成果，其中部分成果也被编入本书。作者波士顿马萨诸塞大学的计算机科学教授，他们在这本书中融入了多年教学经验；同时，由于和工业界有着紧密的联系，他们在撰写本书时非常注意理论和实践相结合，并准确地把握了数据库系统的发展趋势。组织严谨、注重实践、内容新颖是本书的三大特色。

本书共分11章和4个附录，依次为：第1章数据库概论，第2章关系模型，第3章基本SQL查询语言，第4章对象-关系SQL，第5章访问数据库的程序，第6章数据库设计，第7章完整性、视图、安全性和目录，第8章索引，第9章查询处理，第10章更新事务，第11章并行和分布式数据库，附录A教学指导，附录B编程细节，附录C SQL语法，附录D集合查询计数。最后，作者还给出了部分习题的解答。

参加本书翻译的人员有：周傲英、俞荣华、季文瀛、钱卫宁、贺奇、梁宇奇、王焱、郑仕辉、钱海雷、张龙、魏藜、范晔、胡江滔。全书由周傲英审校。

限于译者水平，若有疏漏，敬请读者批评指正。

译 者
复旦大学计算机系
2001年6月

译者简介

周傲英 计算机软件教授、博士生导师，复旦大学计算机系主任。1993年获博士学位。现任中国计算机学会理事、数据库专业委员会委员、上海计算机学会数据库专业委员会副主任。近年来，曾先后担任PAKDD、DAFAA等国际学术会议程序委员会委员和WAIM'2000的程序委员会主席。目前WAIM已成为在中国召开的一个系列的国际数据库学术会议。周傲英作为发起人之一担任会议顾问委员会成员。作为项目负责人或主要研究人员主持或参加了国家973计划、863计划、国家自然科学基金、国家博士点基金、国防科工委预研计划、上海市科技发展基金及对外合作项目的多项研究和开发工作，取得一系列成果，先后四次获上海市和国家教委科技进步奖。近五年来发表了一批高水平论文以及出版教材/著作五本。1995年入选上海市青年科技启明星计划，1997年获上海高校优秀青年教师荣誉称号，2000年获得国务院特殊津贴，2000年入选教育部“跨世纪优秀人才培养计划”。

俞荣华 硕士研究生。专业为计算机软件与理论，研究方向为数据质量和数据清洗。

季文贊 博士研究生。主要研究方向为数据库和Web数据管理。

钱卫宁 博士生。专业为计算机软件与理论，研究兴趣为数据挖掘和Web挖掘。

序

从1994年问世开始，O’Neil的数据库书籍就已经成为学习、设计或管理关系数据库的标准教材和参考书。本书从理论和实践两方面详细介绍了数据库的设计和实现，包括关系理论，数据库设计，数据库的实现以及性能优化等问题。本书在介绍时，先给出一般的概念，然后结合实际数据库系统中的具体例子加以说明。

第2版反映了自第1版发行后的六年里数据库领域取得的实质性进展。本书把重点放在对象-关系模型；介绍了在Oracle、DB2和Informix系统中普遍采用的新概念；更新了对隔离技术的表述方式。用最新的方法提出性能的问题。对象-关系的介绍是本书的重点：这被认为是SQL数据库语言自最初标准问世以来发生的最重要的变化。本书介绍了SQL-99的设计以及SQL-99与基本原理及主要的商用数据库的关系。

Pat教授和Elizabeth O’Neil教授对数据库设计问题有着广博而精深的见解，在过去的三十年中，他们在数据库领域做出了卓越的贡献。他们桃李满天下，发表了很多基础研究的论文，并与厂商合作开发了一些数据库产品。他们不断创新——在本书中，他们就试图用统一的眼光来看待数据库领域中许多完全不同的概念和趋势。第2版提出一种全新的见解。

《数据库原理、编程与性能》对初学者来说是一本极好的教材，对我们这些不注意数据库领域最新动态的人来说则是易于阅读的最新动态资料，而对需要接受适时教育的数据库设计者和实现者来说，这又是一本不可多得的参考书。

Jim Gray
Microsoft Research

前　　言

本书第1版介绍了数据库理论的基本原理,探讨了理论和商业应用之间的联系和裂缝。笔者认为有必要让读者了解最新的数据库应用情况,因为在过去的五年中,各种商业数据库产品都发生了巨大的变化。正是数据库领域的这些重要的变化促使我们出版了第2版,但力求保持对原书基本目标的追求。

在第1版中,我们介绍了最新的SQL和商业数据库系统中的应用程序。本书广泛适用于交互式SQL用户、应用软件程序员、数据库管理员和对数据库感兴趣的学生。在第2版中,我们进一步加强了实际应用与理论基础之间的联系。

一般的书往往只简单罗列了嵌入式SQL的特性,不能满足程序员编写复杂数据库应用软件的需要。厂商的手册能对SQL和数据库编程进行很好的介绍,但缺乏学生和专业人员适应数据库系统和语言变化所需的基础理论。在许多介绍数据库的书籍或手册中并未很好地介绍如下原理:死锁终止(事务需要重做)隐含的问题;实体-联系模型的建立和规范化(以及数据库设计到实际表的转化);在执行访问公共数据的事务时,用户的交互问题(事务活动时进行交互是危险的);索引和查询优化如何影响查询性能方面的考虑——一个数据库管理员应该重视的问题;等等。

本书可以看做是一本完整的技术导论,数据库管理员、应用软件程序员和资深的SQL用户都会需要它。虽然数据库管理员要求比应用软件程序员了解更多的东西,但是,如果后者对前者必需掌握的知识有一个总体的把握,那么他的工作会变得更有效。类似的情况也适用于那些认真的交互式SQL用户。任何与数据库相关的工作人员除了牢固掌握SQL外,还应该了解数据库的逻辑设计、数据的物理存储、索引、安全和性能价格比等方面的知识。

本书的作者都是计算机科学的教授,在与数据库公司合作和数据库应用方面有着丰富的经验。有兴趣的读者可以访问主页 www.cs.umb.edu/~poneil 和 www.cs.umb.edu/~eoneil。

本书的使用

本书是在波士顿马萨诸塞大学的数据库入门和提高这两门课程的基础上写成的。第一门课程介绍数据库的基本原理,本书的前六章取自该课程。第二门课程介绍更高深的数据库概念和性能价格比的问题,本书第7~第10章的内容取自该课程。

在学习本书的时候,读者可根据学习的经验和目的自己安排学习的顺序和重点。举例来说,有基础的读者可以根据自己的兴趣从后面的章节开始学习,只在需要的时候查阅前六章的内容(图1显示了各章之间的关系)。本书在读者掌握前面的概念的基础上才展开新的知识,所以有经验的读者可以根据需要从适当的地方学起。

本书既可用作专业人员的参考书和指南书,也可用作本科学生数据库课程一学期或两学期的入门教材。它涵盖了数据库领域从基础理论和基本概念到发展前沿的所有内容。本书将基本的SQL语言和关系数据库基础放在一起介绍。本书的例子取自ORACLE,INFORMIX和DB2等数据库,我们用它们解释概念,并通过比较这些成功系统中采用的不同方法阐明性能-价格

比问题。每章中的关键性问题通过编程示例和习题得以深入。书后的四个附录提供了补充性的背景介绍。为了方便自学,本书最后还提供了部分习题的答案。

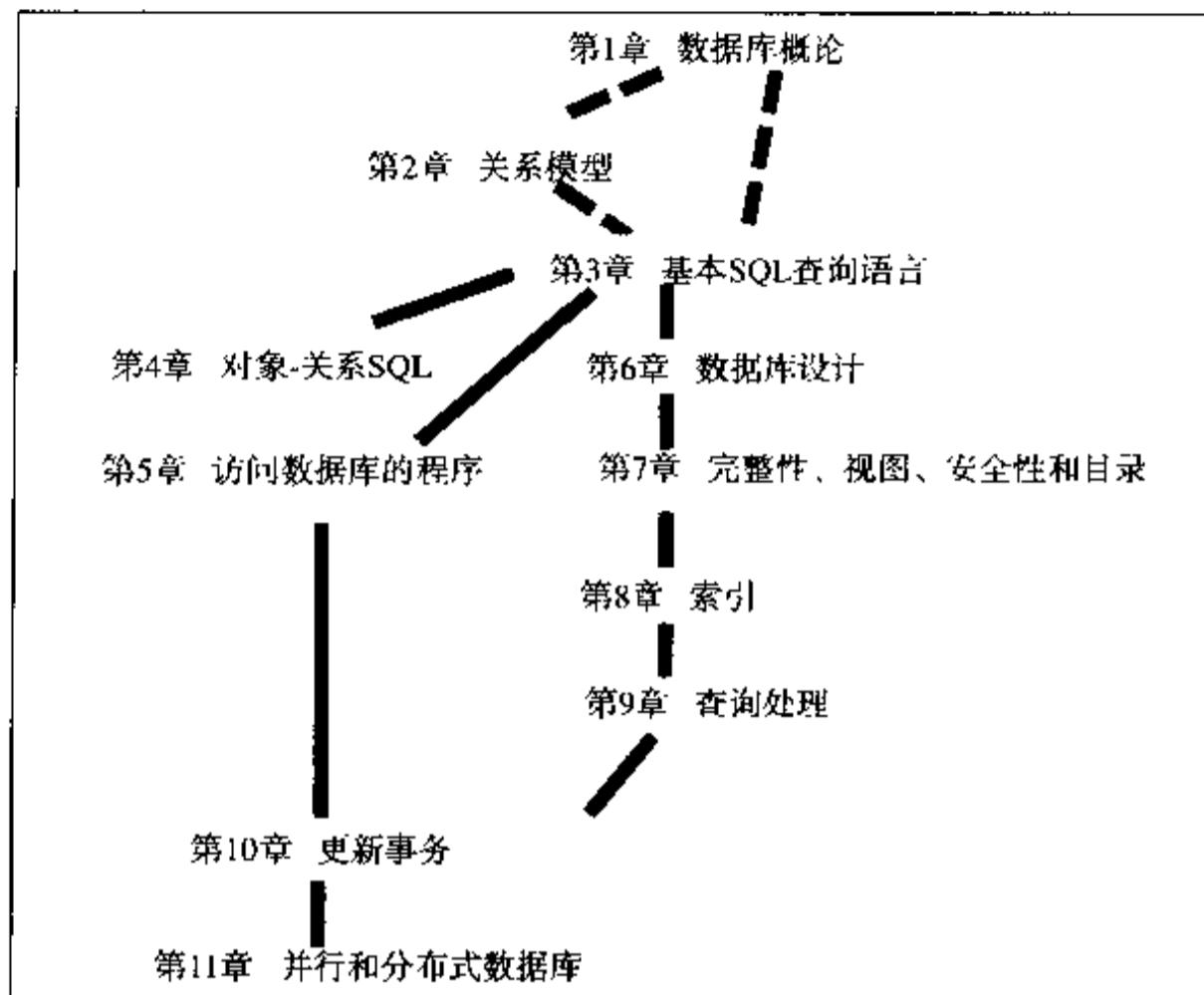


图1 各章之间的联系

新增内容讨论: 对象-关系模型

自从本书第1版出版以来,SQL语言增加了许多新的特性。第3章中新增的几节介绍了这些特性。从根本上说,对象-关系模型已经变成新的数据库标准。所有新的对象-关系数据库管理系统产品都向后兼容,以便支持关系模型,我们认为这一变化正在彻底改变数据库产业。数据表的设计以及利用SQL语句访问数据的方法将发生很大变化。

对象-关系模型成为主流的过程是值得研究的。INFORMIX公司从1996年收购Illustra产品起,就着力研究把对象-关系特性融入到INFORMIX产品中。(Illustra,后来被称做Montage,参见第1版3.11节;在第2版中,我们大大扩充了这方面的内容,独立编写成第4章。)1997年,ORACLE公司推出的ORACLE第8版,支持一整套对象-关系特性,该数据库被普遍认为是具有革命性意义的产品。从那时起,在数据库产业界,从关系模型到对象-关系模型的转变就已是大势所趋了。最近,一向以良好的对象-关系编程接口著称的IBM的DB2 UDB开始将对象-关系数据模型融入设计和交互层面^[2]。不幸的是,由于还没有一个统一的对象-关系SQL标准,各个产品在语法上差别很大,应用程序不可移植。所幸的是,经过多年的努力,ANSI SQL-3终于在1999年发布了对象-关系数据库的最终标准——SQL-99^[3]。

由于有两种完全不同的表述数据的模型,整个知识领域就被分割开来,这在教学上是个很大的挑战。现在,绝大部分的商业数据库软件使用纯关系模型。这意味着如果不仔细对对象-关系概念与关系概念加以区分,用户对前者会产生混淆。和第1版一样,在第2版中我们仍然在第2章中介绍关系模型,而仅在第3章中介绍最新的关系SQL的表现形式。同时,我们扩充了第1版3.11节中对象-关系领域的内容,单独设为第4章,在这一章中,我们分别介绍了

ORACLE和INFORMIX中对象-关系的标准。因为这两个产品间存在很大差异，我们把第4章中每一节分成两个部分，并行地介绍ORACLE和INFORMIX。我们没有介绍DB2 UDB关于对象-关系的语法规则，因为在编写第4章的时候，DB2对象-关系模型正处于开发阶段。

在详细介绍了对象-关系的概念和产品的使用方法之后，接下去的几章在内容上与第4章基本无关。这样做的原因是：迄今为止有些问题在对象-关系领域还没完全搞明白，如第6章数据库的逻辑设计；而有些问题看起来与对象-关系的概念（PL/SQL和SPL除外）无关，如第10章的更新事务。一个专业人员在仔细研读第4章之后，就能用对象-关系SQL的知识扩充大部分后续章节的内容；而那些目前不与对象-关系数据库打交道的读者，可能不想立即接触这些内容。从图1中可以看到，第4章与后续各章没有前后联系。各章的内容相对独立。（第7章涉及对象-关系模式下对象目录表的内容，但读者在学习时可以跳过这一段。）在下一版中，对象-关系的概念可能会遍布每个章节，因为那时对象-关系数据库将会广泛使用，不过现在还为时过早。

在编写第2版的时候，我们不得不做一些困难的选择，最难的莫过于选择合适的数据库产品。我们舍弃了面向对象数据库管理系统（OODBMS），因为虽然面向对象数据库的潜在商业价值被一致看好，但却从来没有真正实现过。现有的OODBMS产品只是存在而已。虽然Microsoft SQL Server正越来越引起大家的注意，我们还是舍弃了对它的介绍，因为我们的主要目的是介绍新的对象-关系数据库产品的特性，而SQL Server不支持这些特性。INGRES系统是第1版的重点内容，因为它是校园中使用最广泛的系统。在第2版中，我们删去了这部分内容，因为许多流行的数据库产品都提供了针对校园市场的低价版本。

每一章的变化

在下面对各章的描述中，我们简要介绍了在第2版中第一次引用的材料。

第1章 数据库概论。新增了对对象-关系模型的探讨。

第2章 关系模型。除了澄清第1版的若干细节外，本章基本上未作改动。我们在第2、3章中介绍关系模型，从第4章开始介绍对象-关系模型。

第3章 基本查询语言SQL。在本版中，我们介绍了很多SQL的新特性。3.6节介绍了一些现有数据库产品尚不支持的“高级SQL语法”。它包括INTERSECT[ALL]和EXCEPT[ALL]操作和新的

第4章 对象-关系SQL。这是全新的一章。我们列表说明各节和各小节的主要内容。每节的最后是对ORACLE和INFORMIX的每一个特性的分别说明和相互比较。

4.1 引言：对象-关系数据库的定义和历史。

4.2 ORACLE中的对象类型，INFORMIX中的行类型，用对象(行)类型定义表，对象嵌套，访问列的点标号，对象数据缺乏封装，ORACLE中REF的介绍，INFORMIX中的类型继承。

4.3 汇集(collection)类型。ORACLE中有两种汇集类型：嵌套表(表类型的列值)和VARRAY(数组类型的列值)。INFORMIX有三种汇集类型：集合、多重集合(无

序,但允许有重值)和列表(有序)。这两种数据库都允许即席查询,从汇集中检索数据,并插入和更新汇集。

4.4 用户自定义函数(UDF)和方法。ORACLE支持一种过程性SQL语言PL/SQL,而INFORMIX支持SPL。在这两个数据库中,UDF可以用过程性语言编写(或用一种嵌入式SQL语言,比如Java)。UDF也被称为SQL中的内置函数。方法是ORACLE支持的一种UDF,它被定义成对象类型的一部分。

4.5 外部函数和用户定义类型包(UDT)。我们概述了数据库系统中打包一组外部函数的用户定义类型的功能。外部函数用类C语言编写并在数据库服务器上执行的UDF。这样一种包在ORACLE中称为Cartridge,在INFORMIX中称为DataBlade,而在DB2 UDB中称为Extender。

以下各章的章号比第1版中的相应章号增加了1,因为在第2版中增加了新的第4章。

第5章 访问数据库的程序。在第2版中,我们提供了更多的编程实例,特别是关于事务方面的。我们改进了错误处理的内容。我们介绍了最新的ORACLE语法,特别是ORACLE动态SQL语法(包括SQLSTATE)。同时还将介绍DB2 UDB(代替第1版中的INGRES)。

第6章 数据库设计。这是指数据库的逻辑设计,包括E-R模型和规范化。本章阐述了很多定义和证明,增加了很多实例说明。

第7章 完整性、视图、安全性和目录。本章新介绍了大量Create Table和Alter Table语句的标准子句,重写了触发器部分,其中的例子来自于ORACLE和DB2 UDB。对可更新视图的限制的介绍变化较大,对查询视图不再有任何限制。系统目录部分稍有扩充,增添了一小节对象-关系目录的内容。

第8章 索引。磁盘速度、容量、价格和内存价格都已更新。还更新了ORACLE中Create Tablespace语句。对INGRES索引功能(ISAM、散列等)不再做专门介绍。新增的内容有ORACLE索引组织表和表聚簇(主要是散列聚簇)。为了处理新的ORACLE散列聚簇结构,对溢出链接做了一些修改。新增了DB2 UDB的索引结构。

第9章 查询处理。虽然在这一章的前几节中已经介绍了新的DB2产品的特性,在一些审阅者的坚持下,本书中我们保留了第1版介绍IBM大型机上DB2的查询特性的内容。正是这些特性导致了集合查询基准测试的测试结果。因为大部分DB2查询功能仍代表了当今的最高技术,而第1版中对查询过程的解释即使在现在看来仍是非常详尽的,所以这部分内容仍然用来向学生解释如何完成查询的重要概念。事实上,查询处理器的内容仍与当前大型机DB2产品(DB2 for OS/390)有关。

第10章 本章的一些定义和证明做了较大改动。在10.5节隔离层次定义中,新增了推荐读物[1]中的研究成果。

第11章 只做了一些细微的改动。

万维网上的支持

本书的主页 http://www.mkp.com/books_catalog/1-55860-438-3.asp 提供以下功能:链接作者的主页;提供为几种数据库产品编写的创建和载入数据库的脚本;提供本书最新的勘误表,示例程序和讲课幻灯片;为教师提供不带黑点标记(·)的习题的答案。读者的建议、意见和勘误

可通过发电子邮件到 poneil@cs.umb.edu或者eoneil@cs.umb.edu 与我们联系。

推荐读物

- [1] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. "A Critique of ANSI SQL Isolation Levels," *ACM SIGMOD Proceedings*, May 1995,pp.1-10.
- [2] Michael Carey, Don Chamberlin, et al. "O-O,What Have They Done to DB2?" *Proceedings of the 1999 VLDB Conference*.
- [3] Andrew Eisenberg and Jim Melton. "SQL:1999,formly known as SQL3." *SIGMOD Record*,vol.28,no.1,March 1999,pp.131-138.

目 录

出版者的话	3.10 Insert、Update和Delete语句	105
专家指导委员会	3.11 Select语句的能力	107
译者序	推荐读物	116
序	习题	116
前言		
第1章 数据库概论	第4章 对象-关系SQL	123
1.1 基本的数据库概念	4.1 引言	123
1.2 数据库用户	4.2 对象和表	125
1.3 关系数据库管理系统和对象-关系 数据库管理系统概述	4.2.1 ORACLE 中的对象类型	125
1.4 小结	4.2.2 INFORMIX 中的对象行类型	136
第2章 关系模型	4.2.3 对象和表小结	140
2.1 CAP 数据库	4.3 汇集类型	142
2.2 数据库各部分的命名	4.3.1 ORACLE中的汇集类型	142
2.3 关系规则	4.3.2 INFORMIX 中的汇集类型	153
2.4 键、超键和空值	4.3.3 汇集类型小结	158
2.5 关系代数	4.4 过程SQL、用户定义函数和方法	159
2.6 集合运算	4.4.1 ORACLE PL/SQL过程、用户定 义函数和方法	160
2.7 自然关系运算	4.4.2 INFORMIX中的用户定义函数	171
2.8 运算依赖	4.4.3 用户定义函数小结	179
2.9 综合例子	4.5 外部函数和打包的用户定义类型	180
2.10 其他关系运算 推荐读物	推荐读物	183
习题	习题	184
第3章 基本SQL查询语言	第5章 访问数据库的程序	186
3.1 引言	5.1 C 语言中嵌入式SQL 的介绍	188
3.2 创建数据库	5.2 条件处理	195
3.3 简单的Select语句	5.3 一些通用的嵌入式SQL语句	202
3.4 子查询	5.4 事务的编程	206
3.5 UNION运算符和FOR ALL 条件	5.5 过程性SQL程序的能力	218
3.6 高级SQL语法	5.6 动态SQL	221
3.7 SQL中的集合函数	5.7 一些高级的编程概念	229
3.8 SQL中行的分组	推荐读物	232
3.9 SQL Select 语句的完整描述	习题	233
	第6章 数据库设计	238
	6.1 E-R概念介绍	239

6.2 E-R模型的细节	245	9.7 连接表的方法	416
6.3 其他E-R概念	250	9.8 磁盘排序	426
6.4 案例学习	253	9.9 查询性能基准程序：样例研究	430
6.5 规范化：基础知识	255	9.10 查询性能测量	434
6.6 函数依赖	259	9.11 性能价格比评估	445
6.7 无损分解	272	推荐读物	448
6.8 范式	276	习题	448
6.9 其他设计考虑	288	第10章 更新事务	456
推荐读物	289	10.1 事务经历	459
习题	290	10.2 交错的读写操作	463
第7章 完整性、视图、安全性和目录	296	10.3 可串行化和前趋图	467
7.1 完整性约束	296	10.4 用来保证可串行性的锁机制	472
7.2 建立视图	313	10.5 隔离级别	476
7.3 安全性：SQL中的Grant语句	320	10.6 事务恢复	482
7.4 系统目录和模式	322	10.7 恢复细节：日志格式	484
推荐读物	329	10.8 检查点	489
习题	329	10.9 介质恢复	493
第8章 索引	335	10.10 性能：TPC-A基准测试	494
8.1 索引的概念	335	推荐读物	501
8.2 磁盘存储	338	习题	502
8.3 B树索引结构	348	第11章 并行和分布式数据库	505
8.4 聚簇索引和非聚簇索引	361	11.1 多CPU体系结构	505
8.5 散列主索引	367	11.2 CPU价格与性能曲线	508
8.6 向靶上的空位随机投掷飞镖问题	373	11.3 无共享式数据库体系结构	509
推荐读物	377	11.4 查询并行性	515
习题	378	推荐读物	517
第9章 查询处理	382	习题	517
9.1 基本概念	383	附录A 教学指导	518
9.2 表空间扫描和I/O代价	387	附录B 编程细节	529
9.3 DB2中的简单索引存取	391	附录C SQL语法	534
9.4 过滤因子和统计	399	附录D 集合查询计数	557
9.5 四配索引扫描、复合索引	402	习题解答	559
9.6 多重索引存取	409		

第1章 数据库概论

本章介绍本书的主要思想和定义。我们描述了数据库的基本概念及数据库系统的典型用户，然后概述与数据库管理系统相关的概念和特性。

1.1 基本的数据库概念

数据库管理系统——简称为数据库系统或DBMS——是一种软件产品，它把一个企业的数据以记录的形式在计算机中保存起来。举例来说，批发商往往用一个数据库管理系统保存销售记录（交易的操作数据）；大学可以用数据库管理系统保存学生的记录（学费、成绩等）；大部分大型图书馆利用数据库系统保存藏书清单和出借记录，提供主题、作者和题目等多种类型的索引；所有的航空公司都利用数据库系统管理航班和提供订票服务；州机动车管理部门利用数据库系统管理驾驶员执照、登记车辆。Tower唱片公司用数据库系统来管理库存，打印所有的磁带和CD，并为顾客提供查询唱片的功能。像这样为一个共同的目的而保存起来的所有数据的集合称为数据库。数据库中的记录通常保存在磁盘上（一种在断电时不丢失保存信息的低速存取介质），一般只在访问时才把记录从磁盘载入内存。

一个数据库管理系统能同时管理多个数据库。举例来说，一所大学可能拥有一个登记学生的数据库和一个图书馆数据库。两个数据库之间没有共享的数据（虽然可能有一部分重复信息，因为一部分读者是学生），不同的用户可以通过同一个数据库管理系统访问这两个数据库。

1. 数据库的历史

为了访问数据库中的信息，已经开发出许多方法。回顾历史，有两个产品为组织信息提供了两种截然不同的数据模型：1968年IBM发布的IMS和20世纪70年代Cullinet Software的IDMS。IMS提出了不同类型的记录通过层次结构相互联系的层次数据模型。例如，一个银行数据库系统可以把公司实体记录和诸如总部地址、电话号码这样的信息放在层次结构的顶部；接下来是银行的各个业务部门；在每个部门分支下，是该部门的出纳员和其他职员的记录。当要查询某个出纳员时，程序就会沿着各个分层导航。另一方面，在CODASYL委员会数据库任务组1971年发表的报告的基础上诞生的IDMS，被称为网状模型。网状模型是层次模型的一个推广，某一级的一个记录集合在上一级中可能对应两个不同的包含层次（containing hierarchy）。

当然，IMS和IDMS还有许多我们没有提到的特性。简单地说，层次模型把数据组织成一棵根在上、叶在下的有向树。网状模型把数据组织成无环有向图，使得网状模型更容易表达现实世界中的数据结构。这些产品的主要缺点是对数据的查询很难执行，一般需要熟悉复杂的数据导航结构的专业程序员编写相应程序。今天，仍然有相当多的公司在使用这两种数据库。IMS仍然是IBM重要的利润来源。但是，这些使用中的IMS和IDMS已经是“遗产系统”了，而且，很难把这些系统转化成现代的数据模型。虽然某些公司用原有的系统已经足够，但任何想安装新系统的公司都会选择一个支持更新的数据模型的数据库管理系统。

2. 关系模型和对象-关系模型

最近18年来，数据库系统产品使用最广泛的数据模型是关系模型。关系模型使用灵活，即使用户不是程序员，也可以快捷轻易地写出一般的查询语句。一个利用关系模型的数据库管理系统称为关系数据库管理系统（RDBMS）。最近几年，一种更新的数据模型——对象-关系模型在许多产品中正逐渐取代关系模型。利用对象-关系模型的数据库管理系统称为对象-关系数据库管理系统（ORDBMS）。因为对象-关系模型实际上是关系模型的扩展，对象-关系数据库管理系统也支持关系数据库管理系统中的数据。因此有些作者将这种产品作为RDBMS/ORDBMS类型，如果我们不愿区分它们，可笼统地称之为数据库管理系统。

在本书中，我们将学习一整套在数据库管理系统中建立、维护和使用数据库的概念和方法。虽然RDBMS和ORDBMS之间存在很大差异，本书全面介绍了几乎所有的相关技术。因为关系数据库是最流行的数据库，所以我们把重点放在这里。只在第4章中独立介绍了对象-关系数据库，而关系数据库的概念遍及本书的其余部分。两种模型中绝大多数的新概念都用相当多的不同种类的商业数据库管理产品和标准中的具体命令加以说明。很多针对复杂的ORDBMS特性的命令语法随系统的差异各不相同，但基本的RDBMS的功能是相同的。

3. 涉及的数据库系统

为了扩展知识覆盖面，本文涉及的商业数据库系统有：

- ORACLE Server，记作ORACLE。运行在几乎所有的UNIX、Windows NT和一些较早的操作系统上。读者可访问www.oracle.com获取相关信息。
- DB2 Universal Database，记作DB2 UDB。运行在大多数UNIX、Windows NT、OS/2和OS/390上。相关站点：www.ibm.com/db2。
- Informix Dynamic Server 2000，记作INFORMIX，运行在大多数UNIX、Windows NT上。相关站点：www.informix.com。
- DB2 for OS/390，记作DB2，特指IBM主机上的DB2系统。运行在OS/390上。相关站点：www.ibm.com/db2。

4. 一个关系数据库的例子

现在我们开始介绍一些基本的概念。图1-1a显示了一个关系数据库的例子，它的内容是大学里每个学生的注册记录。为了阐明概念，这个例子大大简化了。实际上，这样一个数据库会有好几万条记录，每条记录又包含很多字段。

关系数据库中，所有的信息被保存在带名字的表中，每个表又由若干条具有名字的列组成。拿上面的例子来说，名为students的表包含以下各列：sid代表每个学生唯一的标识号；lname和fname代表学生的姓和名；class代表年级，从1到4分别表示一年级到四年级；telephone是学生的住宅电话号码。在一个完整的数据库中，还应包括学生的家庭地址、学费、GPA等内容。students表中的每一行代表一个学生的记录。举例来说，第一行（列名下面的那一行）表示一个名为Jones的二年级学生，他的学号为1，电话号码为555-1234。虽然很多数据库专家强调表与磁盘文件间的重大区别，我们还是可以把一张表描述为一个由记录（每条记录代表一行）组成的磁盘文件。为了更直观地表述问题，以下的术语可互换使用：磁盘文件和表，行和记录，列和字段。

图1-1a中的courses表列出了所开的课程的集合。其中，cno给出课程号，cname给出课程名称，croom给出上课地点，time给出上课日期和时间。其中time列的值是编码：MW2表示星期一、星期三的第二节有课。enrollment表只有三列：sid代表学号，cno代表课

程号，`major`代表这门课是否是这个学生的主修课。例如，学号为1的学生有一门主修课是数学，学号为3的学生主修一门现代语言。图1-1a的三张表一起组成一个关系数据库。注意此处表和列的名字都是小写的，在很多其他的文章里则用大写。

students					enrollment		
sid	lname	fname	class	telephone	sid	cno	major
1	Jones	Allan	2	555-1234	1	101	No
2	Smith	John	3	555-4321	1	108	Yes
3	Brown	Harry	2	555-1122	2	105	No
5	White	Edward	3	555-3344	3	101	Yes

courses			
cno	cname	croom	time
101	French I	2-104	MW2
102	French II	2-113	MW3
105	Algebra	3-105	MW2
108	Calculus	2-113	MW4

图1-1a 关系模式下的学生注册数据库

很多以表的形式表述数据的概念将在第2章中介绍，但这里我们先提一个概念：第一范式规则。在关系模型里，表的每一列只有一个单一的非结构化的值。非结构化约束意味着，举例来说，我们不能把姓和名这两个可分别设置和检索的值放到同一列；也就是说，我们不能创建一个类似C中的结构或JAVA中的类的列（例如，`name`列由`name.lname`和`name.fname`组成）。当然，我们可以创建一个只包含字符串类型值的`name`列，并用分隔符连接姓和名，例如`Allan.Jones`，但这是另一回事了。规则要求每一列包含单一类型的值，我们就不能把`enrollment`表当作`students`表中的一个列，否则表的每一行包含一个(`cno, major`)对的集合，而`enrollment`表变得多余。这样的列的例子如图1-1b所示，这在关系数据库中是不允许的，但可以作为对象-关系数据库系统中的一个学生注册数据库。

5. 一个对象-关系数据库的例子

在对象-关系数据库中，信息仍然被表达成由带名字的列组成的带名字的表的形式。但此时，列值不再受第一范式规则的约束。在图1-1b中，我们给出了一个与图1-1a中的关系数据库等价的对象-关系数据库。关系模型有时被称做是“形式完整”、“位置与值一一对应”的模型。在对象-关系模型中，则没有“一一对应”约束。

在图1-1b中，`students`表中的`name`列用一种新的结构类型来包含多个可分别设置和检索的成分：`name.lname`和`name.fname`，两者作为`name`列的两个成分都出现在父列`name`下。对象-关系数据库还没有一个统一的标准，所以不同的对象-关系数据库产品支持不同的特性和命名规则。在ORACLE中，`name`列的结构类型称为对象类型，在INFORMIX中称为行类型，在DB2 UDB(和新的ANSI SQL-99标准)中称为用户定义类型(UDT)。

students						
sid	name		class	telephone	enrollment	
	lname	fname			cno	major
1	Jones	Allan	2	555-1234	101	No
					108	Yes
2	Smith	John	3	555-4321	105	No
3	Brown	Harry	2	555-1122	101	Yes
					108	No
4	White	Edward	3	555-3344	102	No
					105	No

courses			
cno	cname	croom	time
101	French I	2-104	MW2
102	French II	2-113	MW3
105	Algebra	3-105	MW2
108	Calculus	2-113	MW4

图1-1b 对象-关系模式下的学生注册数据库

图1-1b中students表的每一行的enrollment列值是行类型值的汇集。现在来看学生1(Allan Jones)所在的行，Jones选了两门课:非主修课程101(在Jones的major字段中为No)和主修课程108(在Jones的major字段中为Yes)。同样，在不同的产品中，像enrollment这样包含多个结构化数据的列有不同的名称。在SQL-99中，这样的列称做是江集类型的。由于students表中的enrollment列包含了图1-1a中enrollment表的所有信息，所以在图1-1b中就不再需要enrollment表。

1.2 数据库用户

DBMS的一个重要特性是缺乏经验的用户可以从数据库中检索数据，这种类型的用户称为“最终用户”。他们用键盘输入查询语句，要求数据库把答案输出到显示器或打印出来。例如，在Tower唱片公司，用户希望通过一系列交互式的菜单查询选项，找到需要的唱片。最后，唱片列表输出到显示屏上，同时打印机也输出相关内容来帮助用户找到所需唱片。

在最终用户可以方便地访问数据库的数据之前，专业人员还需做大量的工作。与其他的软件不同，数据库系统在计算机平台上运行后，并不能马上投入使用，相关的数据库还需要设计和载入数据(以后还要不断地更新数据)，然后必须编写应用程序为没有经验的最终用户提供简单的菜单界面。做这些工作的专业人员也可以被认为是数据库系统的用户。只有了解不同类型的用户，才能设计出界面友好的数据库管理系统。专家级用户有责任为高级用户提供一个良好的环境。不管哪类用户都希望有一个便于使用的界面。下面是对各种类型用户的描述:

- 最终用户——交互式用户。
- 临时用户——用SQL访问DBMS的用户。
- 初级用户——通过菜单访问DBMS的用户。
- 应用程序员——编写菜单程序的程序员。
- 数据库管理员——管理DBMS的专家。

最终用户 临时用户应该懂得关系数据库标准查询语言SQL的使用机制。在以后的章节中我们会提到，要学会使用SQL不是一件简单的事。这里的“临时”是指用户经常从一个会话到下一个会话改变查询的要求，此时为每个这样的使用编写基于菜单的应用程序是不经济的。在实现一些急迫的需求时配制的查询称为交互式查询或即席查询(ad hoc query)。（拉丁文ad hoc意为“为了特殊的目的”。）初级用户通过菜单使用数据库，而不必编写SQL语句。银行和航空公司订票系统的职员就是这种初级用户。这里用“初级”可能会产生误解。即使是精通SQL的数据库系统的实现者，在维护软件的时候，也会利用菜单功能以多种形式输出出错记录。这使程序员能集中精力处理真正的工作，而不必花费时间编写SQL语句。

应用程序员 这类用户直接与数据库管理系统打交道，为初级用户编写菜单程序。这种程序必须充分预见到客户的需求，在执行期间提交相应的SQL语句从数据库中检索所需的信息。程序员在处理复杂的概念和困难的语法问题方面经验丰富，把初级用户的要求转化为确切的查询语句是他们的职责所在。为了更便于使用，程序必须确保在查询中没有任何细微的错误。我们将会看到，越是复杂的SQL查询，在构造即席模式时出错的风险也越大。

数据库管理员 DBMS的最后一类用户是数据库管理员(DBA)。DBA是负责设计和维护数据库的计算机专家。一般DBA决定如何把数据分解到各个表中、如何创建数据库以及如何载入表，为了实现访问和更新数据时的各种策略，DBA还要做大量的幕后工作。这些策略包括安全性控制（例如用户访问数据的权限）和完整性约束（例如储蓄账户结余账目不得少于0元）。另外，DBA还要负责设计数据库在辅存（磁盘）中的物理实现和有效的索引结构来获得最好的性能。

在本书中，我们的目标是给出成为一个资深临时用户、一个好的应用程序员或者一个DBA所必须掌握的基本的技术基础。一位使用SQL的资深最终用户不一定要知道如何编程，但他可以从本书的其他许多内容中受益。一位顶级的应用程序员应该懂得所有DBA要用到的技术，这样才能设计出高效的程序，才能反馈适当的信息供DBA调试系统。有关数据库管理员职责的内容几乎贯穿全书，是下面要介绍的各章的主题。DBA处理一些DBMS中最高级的特性。他必须精通前台用户的需求，最好还能了解应用程序的技术细节。DBA所做的决定会在很多方面影响这些程序，包括SQL语句的格式和程序的性能。

注意还有一种与数据库有密切联系的专业人员，称为数据库系统的实现者。他们是编写实现系统核心功能程序的系统程序员。DBMS正是通过这些程序提供各种特性。一些高级数据库教科书着重介绍数据库的内核。由于数据库是一个很大的领域，本书着重介绍一个数据库系统是如何使用的，这是着手实现一个系统前关键的步骤。在任何情况下，一位优秀的DBA都要深入了解DBMS的工作细节。我们会看到，这样的细节在调试系统性能的时候是必不可少的。

1.3 关系数据库管理系统和对象-关系数据库管理系统概述

到现在为止，我们只讨论了关系数据库管理系统的一个主要特性——提交查询语句以便从

数据库中检索信息。通常还有很多其他的特性，其中有些专业性很强，如果没有一定的使用经验，要评价它们的价值是很困难的。尽管如此，在这里我们还是试图给出这些特性的一些思想以及它们的重要意义。这使读者在以后学习细节内容时有一个全局的观念。

下面，我们按顺序对每个主题给出简短的说明。读者会发现那些为个人电脑设计的廉价的小型数据库管理系统并不具备所有的特性。例如，PC数据库系统因为只有一个用户访问，所以通常不是多用户系统。

1. 第2章：关系模型

第2章一开始，介绍多年来支配数据表达方式的关系模型的概念和规则。一份更详细的定义有助于用户弄清楚哪些特性是用户所期望的，什么时候一个商业产品提供的特性不属于标准模型。我们已经说过，有一条规则规定表中的值不能是多值的。这样，`students`表的`hobby`列在同一行中不允许有几个值（如`chess`、`hiking`、`skeet shooting`）。这样的多值列在早期的数据模型中是允许的，称做“重复字段”，但在关系模型中是禁止的。我们已经知道这样的多值列在对象-关系模型中又变得合法了，这看来是下一步的发展方向。关系规则的一个重要作用是为早期关系模型的不同产品提供了统一的标准，这使所有的数据库设计变得一致。即使如此，在没有重大损害的情况下，有些规则经常被打破（不过在对象-关系模型之前，第一范式规则从没有被打破过）。

接下来，我们从关系代数固有的查询能力方面介绍关系模型的特性。关系代数包含一个基于表的操作集合，用这些操作可以产生新的表（就像实数的乘和加运算产生新的实数一样）。关系代数的概念在后面几章中有重要的价值，因为许多数据库的查询要求可以表示为关系代数表达式——但在关系模型中，查询总以表的形式出现。商业DBMS产品用SQL语言而不是关系代数产生计算机化的查询。但是与SQL相比，关系代数有一个好处：它的运算少，而且这些运算描述简单透彻。任何可用的查询语言，例如SQL，都必须包含所有关系代数的运算。学习关系代数的运算是很有意义的，因为你可以通过它理解查询操作，而且它的形式比SQL简单。遗憾的是，至今还没有类似的对象-关系代数。

2. 第3章：基本SQL查询语言

第3章深入介绍了工业标准关系查询语言SQL。因为工业标准SQL，Core SQL-99，还没有被所有的数据库产品采用。我们用“基本SQL”语法介绍被所有主要关系数据库产品普遍采用的特性，用“高级SQL”介绍还未普遍采用的特性。我们推迟到第4章介绍对象-关系模型。第3章中的所有语法只跟关系模型有关。图1-2是对图1-1a学生记录数据库进行关系SQL查询的例子，同时还以表的形式给出了查询结果。

通过察看图1-1a的`students`表中`class`列值等于2的学生姓名和学号可以验证图1-2中的结果。

图1-3展示了一个更复杂的查询。看图1-1a，为了得到查询结果，我们同时需要来自`students`和`enrollment`表的信息。如果单看`students`表，我们不知道学生上什么课；如果单看`enrollment`表，就不知道学生姓名。我们先从`enrollment`表中查出`cno`等于101的`sid`，然后依据这个`sid`值在`students`表中查出学生姓名。图1-3中的SQL语句表示了这种查询方法。

图1-2和图1-3的结果一样并不能说明这两个查询的含义相同。没有理由相信上French1(课程号为101)的所有学生恰好等于学校里所有大二(`class=2`)的学生，这仅仅是个巧合，只说明对这个例子来说，两个查询的结果恰好相等。对于另一个`enrollment`表，这两个查询很可能产生不同的结果。这说明了一个重要的概念：列由创建数据库的人设计，以后不再更改；而

对应于表中的数据记录的行希望经常更改而不产生警告。两个查询语义相同当且仅当对于表中所有可能的相关行值，查询结果都相同。

检索所有大二学生的sid和lname。						
SQL: select sid, lname from students where class = 2;						
answer						
<table border="1"> <thead> <tr><th>sid</th><th>lname</th></tr> </thead> <tbody> <tr><td>1</td><td>Jones</td></tr> <tr><td>3</td><td>Brown</td></tr> </tbody> </table>	sid	lname	1	Jones	3	Brown
sid	lname					
1	Jones					
3	Brown					

图1-2 学生记录数据库中的一个SQL查询

检索上101号课的学生的sid和lname。						
SQL: select students.sid, lname from students, enrollment where students.sid = enrollment.sid and enrollment.cno = 101;						
answer						
<table border="1"> <thead> <tr><th>sid</th><th>lname</th></tr> </thead> <tbody> <tr><td>1</td><td>Jones</td></tr> <tr><td>3</td><td>Brown</td></tr> </tbody> </table>	sid	lname	1	Jones	3	Brown
sid	lname					
1	Jones					
3	Brown					

图1-3 学生记录数据库中的另一个SQL查询

SQL语言还包括向表中插入新行的语句、删除现存行的语句、更新现存行的某些列值的语句。(参见图1-4)。把SQL语言简单地称为查询语言以及在SQL语句是实际的Update、Insert、Delete语句时将SQL语句统称为查询可能不准确，但术语“查询”的这种广义用法已经约定俗成。(有些作者把SQL更确切地称为数据操作语言或数据库语言。)为了方便起见，在不影响理解的情况下，本书把SQL Update、Insert、和Delete语句合称为更新语句，只有它们之间的区别特别重要时才单独指出。

在students 表中插入一行(6, Green, John, 1.555-1133)。
SQL: insert into students values (6, Green, John, 1, 555-1133);
在students 表中删除大三学生的记录。
SQL: delete from students where class = 3;
把courses表中房间号2-113 改为2-121。
SQL: update courses set croom = '2-121' where croom = '2-113';

图1-4 SQL中删除、插入和更新操作

3. 第4章：对象—关系模型

第4章中，我们详细介绍了对象—关系模型的特性。首先介绍用户自定义类型，这使用户可以创建如图1-1b中含有结构化和汇集值的表。如果用第3章介绍的SQL语言从这样的表中检索数据，要做若干语法扩充，我们在这里要介绍扩展的对象—关系SQL (ORSQL)。不幸的是，我们所学习的数据库产品ORACLE、INFOMIX和DB2 UDB中，扩展SQL的语法不一样。所以我们不

得不把教材分成两个部分，分别介绍前两个产品。经过几年的修改，于1999年发布的产业SQL标准——SQL-99在语法特性上确立了规范，我们希望不同产品的各种ORSQL最终会统一到这个标准上来。但对于我们这样的教材来说，事情发展得太快了。我们只能努力介绍有实用价值的语法。读者可以访问本教材的主页<http://www.cs.umb.edu/~poneil/dbppp.html>来获取最新的发展情况。

如果图1-1b中对象-关系表包含的列与关系表中的列相同，那么不用奇怪，对该表的查询与第3章中的SQL查询一样。例如，图1-2中的关系SQL查询语句，由于姓现在是name列的一部分而不是单独的一列，所以只需把lname改为name.lname就变成ORSQL语句了。但是图1-3中的查询，用到了关系SQL中的连接（与enrollment表）。而enrollment表在对象-关系数据库中变成students表的一个列，这时变化就很大了（参见图1-5）。

检索上101号课的学生的sid和lname。							
SQL: select sid, name.lname from students where 101 in (select cno from table(students.enrollment));							
answer							
<table border="1"><thead><tr><th>sid</th><th>name.lname</th></tr></thead><tbody><tr><td>1</td><td>Jones</td></tr><tr><td>3</td><td>Brown</td></tr></tbody></table>		sid	name.lname	1	Jones	3	Brown
sid	name.lname						
1	Jones						
3	Brown						

图1-5 学生记录数据库中的一个ORSQL查询

在图1-5中，括号中的Select子句利用enrollment汇集，把每个学生所上的所有课程的cno收集到一个行集合中，然后where 101 in语句检查101是否在某个学生的cno集合中，如果在此集合中，则返回TRUE给顶层Select语句，打印这个学生的sid和name.lname。

ORSQL同样产生插入、删除和更新操作。看图1-6的例子，这个简单的例子把汇集类型的数据当作一个整体处理，我们也可以遍历汇集，对其中每一行进行操作。

在students表中插入学生John Green 的记录。	
SQL: insert into students values (6, Green, John, 1, 555-1133, set(row(101,'No'),row(108,'Yes')));	
删除所有只上101号课，且101号课不是主课的学生记录。	
SQL: delete from students where students.enrollment = set(row(101,'No'));	
修改1号学生的enrollment属性，使他上105号非主修课和107号主修课	
SQL: update students set enrollment = set(row(105,'No'),row(107,'Yes')) where sid = 1;	

图1-6 删除、插入和更新的ORSQL语句

在第4章的最后几节中，我们学习怎样通过书写任意复杂的用户自定义函数（UDF）来增强对象-关系SQL的能力，UDF可以用到非过程SQL语句中实现任何程序化的操作。

4. 第5章：访问数据库的程序

在第4章的例子中(最后一节除外)，我们集中介绍了这样一个环境：用户交互地把SQL查

语句输入数据库系统（一个查询接口），然后系统把结果以表的形式输出到用户显示屏上。这的确是获得查询结果的一种方法，但不是唯一的一种。我们在第5章中将讨论，实际上大多数的查询都是通过一个应用程序界面提交的。这里，程序员用诸如C、Java等高级语言或DBMS提供的特殊的过程性SQL语言编写程序。程序执行时，一般以菜单的形式与监视器用户交互（可能有多个用户同时执行该程序）。用户选择菜单选项，然后程序可以通过完成一个或多个SQL查询来执行由用户的菜单选择提交的任务。所有的行为都由监视器用户操纵相关菜单的选项控制——实际上任何时候，监视器用户都不直接创建SQL查询。在图1-7中，我们给出菜单交互的例子。

在图1-7中，用户想查询Isaac Asimov写的所有关于“机器人学”(robotics)的书。为了使用方便，可以使用SQL语言提供的各种通配符。例如，在用户不清楚拼写的时候，可以用“Asim%”代替“Asimov”或“Asimoff”。在Subject栏的关键字的选择可能以字母表的次序列列出。这对于程序来说是额外的工作，但不这样用户就不清楚是使用“robotic”还是“robot”，亦或“robots”。

在这里要注意：虽然SQL查询（或其他的SQL语句）可以在程序内部执行，但C、Java等大多数高级语言不支持SQL查询，所以要用一种特殊的标记（如图1-8的C程序代码中开头的`exec sql`）指出SQL语句。程序通过预处理器运行，这些特殊的格式被转化为对数据库函数库中C函数的合法调用，然后由编译程序产生一个可执行程序。

```
Library Database Selector Menu
Author last name: Asimov
Author first name: Isaac
Title:
Subject: robotics
```

图1-7 一个菜单交互式数据库查询的例子

```
exec sql begin declare sections;          /* C variables known to SQL */
    char lname[18];                      /* author last name variable */
    char fname[14];                      /* author first name variable */
    char title[20];                      /* book title variable */
    char subject[24];                    /* book subject variable */
exec sql end declare section;
exec sql declare c1 cursor for          /* define cursor for query */
    select alname, afname, title, subject from library
        where alname = :lnamev and afname = :fnamev
        and title = :title and subject = :subject;
menu(lname,fname,title,subject);        /* function to display menu */
exec sql open c1;                      /* begin query, start cursor */
exec sql whenever not found goto alldone; /* set up for loop termination */
while (TRUE) {                          /* row by row printout */
    exec sql fetch c1 into :lname,:fname,:title,:subject; /* get row values */
    displayrow(lname,fname,title,subject); /* print them out */
}
alldone:                                /* reached when loop is done */
```

图1-8 一段(简化的)用于数据库访问的程序

把SQL语句放到高级语言程序中的做法称为嵌入式SQL编程。如果你有过这样的编程经历：把C结构写入磁盘文件，然后再读出；那么你会发现用SQL做这些事是多么的简单。特别是，嵌入式SQL使得程序员遇到的最困难的任务之一完成起来比较容易。我们会看到，DBMS利用很多基于磁盘的数据结构来有效地访问数据，但程序逻辑中不必知道这些结构，

其中仅引用表的结构和表中包含的列值。程序逻辑中也不必了解访问模式、文件中的位置、记录结构等，所有这些细节都由DBMS处理，这种特性称为程序-数据独立性。表的结构随着时间的流逝而发生变化时，程序-数据独立性的重要性就日益明显。表中可能会增加一些列，而行数的增加可能使原有的单个磁盘不能容纳整个表，从而分割存储到几个磁盘中，这些变化引起对新的数据结构的需求，以加快根据列值访问行（类似通过卡片目录查找图书馆中的书）的速度。在一个设计合理的数据库里，所有这些需求都不会影响应用程序的逻辑结构。程序-数据独立性非常重要，我们为它给出本节唯一一个正式定义：

定义1.3.1 程序-数据独立性是具有良好构造的数据库的一种特性，它使应用程序逻辑的存储结构和存取方法免于修改。

5. 第6章：数据库设计

数据库管理员（DBA）要处理数据库从创建开始的整个生存周期里各个方面的工作。一开始，DBA深入研究要用关系数据库来表示的企业。DBA研究现实世界中组成企业的数据对象的基本属性和相互关系，并为之建模。这样做的目的是为了提供一种把这样的对象表示成关系表中列的准确方法。作为这种表示法的一部分，DBA还制定规则，即完整性约束，来限制对数据可能的更新。DBA工作的前期阶段称为数据库逻辑设计，简称为数据库设计。数据库设计是一个极其复杂的研究领域，在这个领域中要求DBA识别各种反映现实世界的数据关系，是一种以实用为目的，但又类似抽象的哲学方法的数据分类学。在本书中，我们要学习两种基本的数据库设计方法。一种方法是凭直觉识别现实世界中的数据分类，称为实体-关系模型（E-R模型）；另一种方法，作为前一种的补充，称为数据库规范化(database normalization)。这些方法将在第6章中具体介绍。

分析一个诸如一所大学这种大型的对象时，我们要识别将在关系表中表示为列名的为数众多的各部分数据。很明显，我们需要一些指导规则决定如何把这些列名联系起来，哪些应放在同一个表中，哪些应放在不同的表中。这是逻辑设计的基本问题。首先，我们把对象区分为实体，有时称为实体集或实体类型，是一类真实存在、彼此之间可区分的对象。作为一个例子，我们考虑某所大学的全体学生组成的集合，并将其命名为实体Students。与每个实体相连的是描述和区别这些对象的属性集合。在图1-1a中，我们把Students实体具体化为一个关系表（命名为students），它包含一组描述学生的列名（属性）：sid, lname, fname, class和telephone。在大学里必须保存的另一类对象是Courses实体。在图1-1a中有一个courses表，包含描述不同课程的属性：cno, cname, croom和time。

图1-1a中的第三个表enrollment表示的并不是一个实体，而是实体间的联系，在图1-9的实体-联系图中，以动词形式命名为enrolls_in。每个学生要上多门课，每门课有多个学生上。在图1-9的E-R图中，矩形表示实体，椭圆表示属性，菱形表示实体间的联系，实体与相关属性以及实体与联系之间用线段连结。

在为一个大企业设计新的数据库时，实体-关系观点使数据库设计者对在设计过程中要遇到的数据项汇集产生一个大致的印象。当每个实体都被识别出来后，就能清楚地看到很多数据项仅能作为描述某个实体的属性，这样一个关系表和很多列就被识别出来。随着DBA深入研究这个实体，很多原本不清楚的属性也被识别出来。因为我们还没有介绍联系的很多重要属性，设计表示联系的关系表会更复杂一些。其实，有些联系并不转化为一个单独的表，而是表示为实体表的一个列，称为“外键”（foreign key）。在第6章中，我们再详细介绍利用E-R图设计数据库。

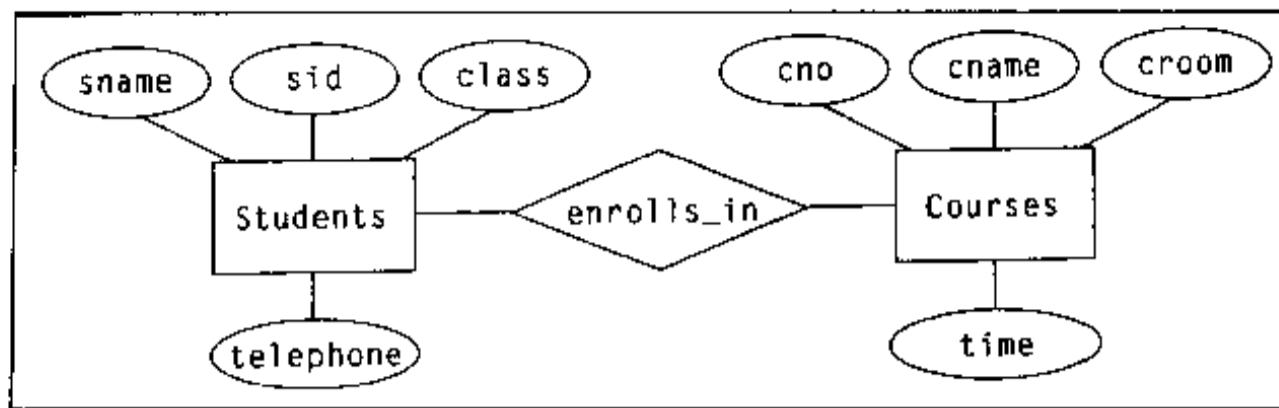


图1-9 学生记录数据库的E-R表

6. 第7章：完整性、视图、安全性和目录

DBA完成数据库的逻辑设计以后，还有很多物理层面的工作要做：把逻辑设计转换成用户可以访问的计算机化的表示法。我们首先说明许多DBA用来创建数据库表的SQL命令和其他的数据库实用程序，再把数据从操作系统输入文件中载入表，最后在不断变化的环境下维护数据库。大部分的命令都是很直观的。你可能希望一开始就能掌握一些简单的创建表的命令，以便练习SQL查询；然而，一开始创建的表都是简单的缺省模式，缺少后面几章介绍的诸如完整性约束、数据视图、安全性约束等性质，也没有支持高性能访问的索引结构。在下面的几节中，我们简单介绍某些性质的价值，具体内容在第7章中展开。

(1) 完整性约束

在逻辑设计过程中，DBA已经确定了很多数据完整性约束，这些都是数据库在任何时候都必须遵守的重要规则。例如，有规则规定：courses表的每一行都有唯一的cno值（一个课程号不能被两门不同的课程共用）。另一个例子是enrollment表中sid列的每一个值在students表中都有对应的行（引用完整性约束），或savings_accounts表中balance列的值不能少于\$25.00（最少结余）。很明显，如果一个表最初遵守完整性约束，后来却出错，那么，这可能是由某个SQL更新操作造成的一——例如，某个SQL操作把一行插入到courses表中，新插入的行与原表中的某一行有相同的cno值。

如果我们不允许即席的SQL更新，那么，通过在应用程序逻辑中添加周全的强制性检查规则，就可以保证所有这些完整性规则（“……在插入新的courses行时，程序员总得检查cno是否重复……”）。然而，把强制检查的工作留给程序员的做法并不是关系数据库管理系统采用的方式。相反，DBA创建完整性约束就像创建不同的实体，这种约束由DBMS自动执行，所以对于任何约束都不可能出现无意间被打破的情况。采用这种方式的好处是：即使是即席更新或由无经验的程序员编写拙劣的应用程序逻辑也不会损害数据的合法性。图1-10显示了ORACLE数据库中的一个例子：用SQL语言创建完整性约束的语法。

为students 表增加完整性约束，确保class 列的值只能在1和4 之间。注意这是ORACLE 数据库中合法的SQL 语句。

```

SQL: alter table students modify
      (class smallint check(class >= 1 and class <= 4));

```

图1-10 在ORACLE中创建新的完整性约束的命令

(2) 数据和数据库安全的集中控制

注意把信息放在两个不同的数据库中会产生严重的后果：一个查询不能同时从两个数据

库中取出数据。例如，前面提到的学生记录数据库和大学图书馆数据库。如果一个学生所借的书超期，对于图书馆而言打印催书信和邮寄标签的程序就不能使用学生记录数据库中的学生住址。这意味着大学图书馆数据库必须保持一份学生住址的自己的备份。这被称为数据冗余(data redundancy)（数据重复——可能是不必要的）。这不是一个很好的选择，因为当学生记录数据库里登记的学生地址被更改时，会造成图书馆数据库里的学生地址过期。可以采用另一种方法来获得最新的地址：频繁地给邮局写信询问地址的变化情况，然后由职员根据新的地址更新数据库中的记录。这样做的结果是大大增加了注册办公室的费用，而这都是由于在一个分离的数据库中构建表造成的。

这种讨论指出这样一个优势：DBA可以利用相关信息把数据库结合起来，集中控制数据以减少数据冗余。数据的集中控制是著名的数据库原理，但是，集中也有它的缺点。例如，我们也许不希望学生图书管理员能够更新(甚至是察看)学生记录中有关评分、学费和病史等内容。DBA可以利用一套安全性命令实现这些约束，并在创建视图时使用安全性约束。安全性命令的执行方式类似完整性约束，由DBMS在底层自动执行，这使用户不能利用SQL语句访问或更新某些表或表中的某些字段，除非用户明确地获得DBA的授权。安全性在SQL中统一用Grant语句声明，几乎所有的关系数据库产品都提供这个功能。Grant语句有完备和灵活的特性。举例来说，可能允许一组用户对一个表做读访问，但不允许更新。图1-11显示了一个用标准的SQL语法设置完整性约束的例子。

授权允许poneil用户在students 表中执行select、update和insert 操作，但不允许执行delete 操作。 SQL: grant select, update, insert on students to poneil;

图1-11 标准SQL Grant语句

(3) 数据库视图

集中控制存在一个隐含问题是连接带有相关信息的数据库时的逻辑复杂性。例如，一个简单地验证大学生保险金的应用程序可能要处理几十个表来获取需要的列信息，而其中很多表的名字和用途可能完全与健康保险无关。这使得培训程序员的工作变得很困难。而且，在一个大的机构里，一开始就把所有相关的数据库结合起来也是不可能的——我们应该允许数据库的分阶段实施。在这个过程中，集中控制的好处是在将新数据库结合起来的过程中逐步增长的。为了消除冗余，当加入新表时，我们不得不重写应用程序，用对新列的访问代替对旧列的访问。我们希望有一种方法能够确保表的数目的增加不会导致培训程序员的问题，并且能够保证在为消除数据冗余而重新安排旧表时不必重写原来的应用程序。

使人惊奇的是，这些问题都可以通过称为数据库视图的DBMS特征来解决。视图命令允许我们创建一个假想的或虚拟的视图表，视图表中的行是利用SQL中的Select语句作用于其他表来定义的。产生视图的表中也可以是视图表，但视图表中的行最终都基于以文件形式存储在磁盘上的基本表。在图1-12中，我们提供了一个用标准SQL语法创建视图表的例子。

视图的巨大功用（在理论上）源于这样一个事实：SQL中的Select语句可以访问一个视图表，就像访问一个真正的表。这样，DBA就可以把几个（新）表中的相关字段组成一个（旧）表的虚拟拷贝，供原有的应用程序频繁访问。在数据库中，DBA通过创建仅与特定应用有关的视图，可以简化培训程序员的数据库环境。视图还提供了天衣无缝的安全性措施，使得未被授权访问的字段和表的用户不能访问。很明显，视图扩充了程序 - 数据独立的概念：应用程序逻辑不仅不受物理存储结构的变化的影响，也不受基本表结构的逻辑变化的影响。

创建一个名为studentcourses 的视图表，列出学生名字和学号(sid) 以及这个学生所上课程的课程名和课程号(cno)。

```
SQL: create view studentcourses (lname, sid, cname, cno)
      as select s.lname, s.sid, c.cname, c.cno
      from students s, courses c, enrollment e
      where e.cno = c.cno and e.sid = s.sid;
```

图1-12 创建视图表的标准SQL命令

不幸的是，实际上按商业SQL标准实现的视图在某些方面与理论还有差距。虽然基于多个基本表可以创建一个相对概括的视图表，但对于访问这个视图表的SQL语句却有很多限制。具体来说，某些select语句和绝大多数的更新操作都是不允许的，虽然它们很有用。把用户对视图的更新转化为对基本表的更新是困难的，这表明视图上的某些更新语句在理论上还有一些局限性(包括Update, Insert和Delete语句)。当然我们可以不使用这些操作，但在商业标准中为了方便系统实现，又舍弃了其他很多有用的更新操作。将来我们期待能看到更少局限性的视图，但现在却不得不面对大量的人为限制。

7. 第8章：索引

在安装数据库的时候，DBA的一个主要职责是决定如何在磁盘上安排基本表，以及创建什么样的索引来优化应用程序和交互操作所请求数据的访问，这就是所谓的数据库的物理设计。第8章结合实际数据库系统研究如何选择设计方案，讨论数据存储与索引的各种方式。

索引和表的设计

我们从三个主要的RDBMS产品ORACLE、DB2 UDB和INFORMIX提供的有关功能着手研究数据库的物理设计，而且我们把注意力集中在索引上，当索引被设置成表的某一列的值时（例如图1-1a中students表的lname列），索引类似于图书馆里查书用的目录卡片。例如，系统可以通过lname索引执行查询语句。

```
select sid, class from students where lname = 'Smith';
```

lname的值‘Smith’在索引中被迅速找到，然后索引项给出lname值等于‘Smith’的行（可能不止一行）的位置。当然，对图1-1a中只有少数几行的students表来说，这样的索引并没有什么用处。但在实际应用中，一个表往往包含几千行，在这样的表中查询某些行的时间主要花在检查每一行是否符合查询条件上。这时，利用索引可以大大提高查询的效率。另一种方式是直接把行按某种索引的次序排列，称为聚簇(clustering)，这类似于在图书馆书架上按作者次序排列书。聚簇只能是单索引，不过这种附加的索引也是实现高效访问所必需的。

本书详细介绍了多种基于磁盘的用于行访问的结构(比如散列)和B树结构，和这些结构中作为一个DBA必须熟练掌握的许多特性(比如磁盘空间管理)。还介绍了一些折衷策略处理大容量表中使用大量索引的问题。例如在一个图书馆目录系统中，虽然使用大量的索引可以提供各种高效的查询，但当在表中插入新行或删除旧行时，就要对索引做大量的更新工作。

8. 第9章：查询处理

在第8章中详细介绍了索引概念以后，在本章中我们将探讨查询优化的概念。优化过程是由数据库系统完成的。对复杂的SQL查询来说，很难一步一步地决定使用什么样的访问策略来优化查询过程，因为查询过程可能涉及基于不同列值和不同表的多种索引以及在各个连续

的阶段，可以有选择地使用一批完全不同的访问技术。SQL查询语法具有一个重要的特性，称为非过程性。这意味着SQL查询语法允许用户只说明希望得到什么结果，而不必指出如何得到结果（也就是说，不必按部就班地指明查询的过程）。因为SQL查询有非过程性的特点，所以就有可能把决定访问策略的工作交给DBMS的查询优化模块去做。这仍然是前面提到的作为现代关系数据库管理系统的特征之一的程序-数据独立的另一个方面。书写SQL查询语句的程序员可以完全不必知道数据和表索引的物理结构。

本书几乎介绍了IBM OS/390系统上DB2数据库关于查询优化的所有内容。由于各个数据库产品在查询优化方面大同小异，所以我们只集中介绍其中的一个。不管怎么说，DB2在查询优化方面表现出色，涵盖了其他数据库产品在这方面绝大多数的特性。然后，我们通过具体解说集合查询基准程序的测试结果，给出一个扩充的例子说明查询效率的问题。基准程序是一种工业标准测试，用来判断为某一类特定用途结合在一起的不同的硬件-软件（DBMS）产品之间的相对的性能-价格比。集合查询基准程序用来测试产业界推出的普通的查询应用程序的性能价格比。提出性能价格比的概念是为了找到用来比较不同的硬件-软件平台的一般尺度。通常，一种平台（如IBM大型机上的DB2）的某些特性很难与另一种平台（如Sun Solaris计算机上的ORACLE）的某些特性相比较。每一个厂商联盟都宣称自己的产品的特性更优秀，这使得我们难以判断。这时，就可以用性能价格比的概念解决这个问题：整理在业界应用中有普遍用途的一类工作。为这个工作设计若干测试单元，比如查询单元（实际上是大量的各种类型查询的一个平均数）。我们测试不同平台为达到同一个标准工作比率（每分钟查询（QPM））而耗费的美元价格（\$COST）。最后，比值\$COST / QPM就可用作不同硬件-软件平台之间比较的尺度。

9. 第10章：更新事务

对数据库进行更新访问和进行查询访问在概念和技巧上存在很大差异。当对数据库做更新访问时，我们要确保所有的更新操作或者全部成功或者一个也没有执行。以下这种情况是不能接受的：在进行转账操作时，从第一个账户中成功扣除了一笔款额，但由于系统崩溃的原因，没有把这笔款额加到第二个账户中去。另外，在转账过程中，单纯的查询应用程序不能计算这两个账户的余额总数。因为此时可能刚好从第一个账户中扣除了款额，但还没有在另一个账户中加上款额。如果一个人在转账过程中计算资产总值，就可能得不到一个正确的答案。为了避免发生这种问题，提出了数据库事务的概念：事务把一系列对记录的读和更新操作组成一个不可分割的包。事务具有原子性，这意味着事务所涉及的行只能在事务执行前或事务结束后才能被其他操作访问。在事务执行过程中，决不允许有其他操作访问相关的行。

要理解如何在一个事务中打包一组操作，先得掌握其他很多复杂的概念。从讨论嵌入式SQL开始，我们初步介绍了三个不同的数据库产品提供给程序员的一系列SQL事务命令。包括如何开始和结束（或提交）一个事务。由于直到第10章才具体展开更新事务的详细内容，为了使编写的应用程序能处理多个更新操作同时进行的情况，我们提前介绍一些方面的内容。研究相互影响的更新操作是数据库系统中关于集中控制的一个重要领域，通常称为事务处理或在线事务处理（online transaction processing，OLTP）。若干事务的概念对于应用程序编写者来说包含了重要的结论——例如事务异常中止的概念。为了保证事务的原子性，有时DBMS不得不中止事务的执行，回退到现在为止所做的一切更新，并把该事件通知给应用程序。这时，应用程序应能处理这类事件，重做该事务。

在第10章中，我们探讨事务系统中并发概念的理论基础。某个事务在执行时，可能有其他用户的事务试图访问(称并发访问)正在处理的相同记录，这时，并发控制要保证事务的原子性。并发访问出于性能因素是清楚的，于是，由并发引起的冲突就是以最一般的表达形式和某种众所周知的方法避免所述的事情的发生在实际应用中，这就导致称为两段锁(two-phase locking)的方案：对被某个事务访问的记录加锁，以防止这个记录被其他并发用户访问，一般在这个事务完成时才对记录解锁。两段锁法与后面关于性能调节的概念密切相关。

在仔细研究了并发控制以后，本书考虑如何处理在系统崩溃破坏主存内容时已经写入磁盘的那部分更新操作。这时候，在应用程序中，系统不知道事务执行到了哪一步。这就是数据库恢复(database recovery)的问题。为了解决这个问题，系统会自动在磁盘上记录事务执行的步骤。这样当系统崩溃时，系统就可以撤销属于尚未完成的事务的那些更新操作，把系统恢复到事务执行前的状态。

接下来，我们考虑事务处理的性能问题。我们将介绍一些调节系统的数据库管理命令和由事务处理性能协会(Transaction Processing Performance Council)制定的工业标准OLTP基准程序TPC-A。很多事务处理的性能价格比问题与前面介绍的查询处理不同。在第10章中我们将详细解释这些新概念。

10. 第11章：并行和分布式数据库

最后，在第11章中，我们介绍并行和分布式数据库处理的概念。一个数据库可分解成若干部分，每一部分都可以存储在不同的计算机磁盘上。这样的数据库系统可以包含任意多台计算机。虽然被分成若干部分，我们仍希望能对数据库做查询和更新操作。这些操作需要同时访问若干台计算机上的数据。促使分布式数据库采用这种方法的一些概念如下：首先，我们希望如果各地的站点可以在当地存储它们的数据，而不必通过远距离通信线传输信息，那么系统的性能价格比就能提高。然而此时，数据的集中控制还是非常重要。例如，当本地的一个大公司库存耗尽时，就得对另一个城市中的仓库发相关命令。可靠性仍是一个重要的考虑因素。随着数据库系统包含的磁盘和处理器数目的增加，平均无故障时间(MTTF)也变短。提高可靠性的办法是把数据复制到不同的磁盘上，使得系统中独立的计算机都可以访问。以上很多因素都是第一次在关键任务应用程序中提到。

1.4 小结

在本章中我们介绍了本书要讨论的基本概念和定义。接下来的几章会深入讨论如何创建、维护和使用关系数据库。我们的目标是使读者理解数据库理论以及这些理论在现有的数据库标准和产品中的应用细节。

第2章 关系模型

一个数据库模型或者数据模型是一组描述如何用计算机化的信息表示现实世界中数据的定义。它同时也描述了访问和更新这些信息的操作类型。第1章中已经说过，本书将侧重点放在了关系模型和对象-关系模型上。从20世纪70年代末就开始使用的关系模型是最为经典的一种数据库模型。但是大多数的数据库系统厂商(比如ORACLE、DB2 UDB和INFORMIX)在它们最近发布的版本中已经开始支持对象-关系模型;也许在不久的将来，所有重要的数据库厂商都将这么做。在这两种模型中，数据库，也就是相关信息的集合，是用一组表来表示的。这些“表”有着非常严格的概念结构。表中的各种不同结构是如何命名的(如表的标题)，表所遵循的结构规则(如表中的两行元素在列上的值不能全部相同)以及这些规则用于数据访问时衍生的概念(如一个表的主键是表中能够唯一标识行的列的集合)都将是讨论的重点。本章的前四节将讲述关系模型的一些结构上需要考虑的东西，以及和对象-关系模型的区别。关于对象-关系模型的细节部分我们将在第4章做详细讨论。

从2.5节开始，我们将介绍关系代数。关系代数是由一系列最基本的关系操作组成，它可以用来从旧表中建立新表，就像在算术上用加法或者乘法就可以产生新的数一样。关系代数是一种抽象的语言，这意味着无法在一台具体的计算机上执行用关系代数形式化的查询。之所以在本章引入关系代数，是为了用最简单的形式来表达，所有关系数据库查询语言为了回答用英语所表达的对数据库中信息的查询而必须采用的操作的集合。所有这些基本概念对于在以后的章节中我们介绍关系数据库系统的标准查询语言SQL是非常有用的。

本章包含的大量的数学定义和定理，因此相对于后面的关于查询语言的部分章节来说，本章的组织用了非常抽象的方法。这一点是有很多原因的。首先，在介绍某个领域的最基本的概念的时候当然是尽可能做到准确；而更为重要的是，对于从事正规数据库研究的人员来说有时候需要能在不同类型的表达中相互变换。大多数的商业化的数据库产品，如DB2 UDB、ORACLE 和 INFORMIX都提供了一本极其精美的手册，用于说明实际数据库操作中可能出现的不同概念，然而，还是有一定数量的更深层次的概念是这些手册所无法包括的。为了掌握这些概念，数据库的使用者还是需要去查阅参考书或者原始的研究论文。(例如，可以参阅本章最后的“推荐读物”)。在这里，我们使您在阅读那些更为抽象的参考资料之前先掌握一些基本的知识。

2.1 CAP数据库

正如我们在1.1节中提到过的那样，数据库是为了某个特殊目的存储在一起的相关数据记录的集合。在以下的章节里将使用一个特殊的数据库例子，我们称之为CAP数据库(由表CUSTOMERS, AGENTS和PRODUCTS组成)。CAP数据库表显示在图2-2中，批发商用它来记录他们的顾客、商品和接受顾客订单的代理商的信息。

图2-2的CAP数据库中的表结构信息显示在图2-1中。表中所列的顾客是从批发商那里批发大量商品然后自己转销的零售商。CUSTOMERS中的顾客用cid(顾客标识符)属性作为唯一标

识。顾客向代理商(在表AGENTS中用aid属性唯一标识)要求购买商品(在表PRODUCTS中用pid唯一标识)。每次订货都会在表ORDERS中增加一条订单记录，用ordno唯一标识该记录。比如，在表ORDERS中ordno为1011的订单表示是在一月份(jan)由顾客c001，通过代理商a01订购了1000份商品p01，价值\$450。图2-1提供了CAP数据库中所有的表和列的定义，而这些表中的内容显示在图2-2中(这些内容可能随时间改变)。

CUSTOMERS	存放顾客信息的表
cid	唯一标识一个顾客/行——注意：表的内容中没有cid为‘c005’的相关顾客
cname	顾客的名称
city	顾客所在的城市
discnt	每个顾客可能会有的折扣
AGENTS	存放代理商信息的表
aid	唯一标识一个代理商/行
aname	代理商的名称
city	代理商所在的城市
percent	每笔交易代理所能获得的佣金百分比
PRODUCTS	存放商品信息的表
pid	唯一标识一件商品/行
pname	商品的名称
city	商品库存所在的城市
quantity	目前可销售的商品库存数量
price	每单位商品的批发价
请注意：在到目前为止已定义的所有三个表中，都有相同的列名city，这不是偶然的事情，三者的含义不同。	
ORDERS	存放订单信息的表
ordno	唯一标识一份订单
month	订单月份，假设所有的订单是从本年度的一月份开始的
cid	购买商品的顾客
aid	经由该代理商订货
pid	所订购的商品
qty	订购的商品数量
dollars	商品的总价

图2-1 CAP数据库中表和列的定义

注意，数据库管理员创建cid列为顾客的唯一标识符，而没有其他单独列能够满足该要求。比如，属性cname在不同的行上可能有相同的值(并且实际上“ACME”就是出现在不同的行上)。同样的，aid、pid和ordno属性在它们对应的表中定义为唯一的标识符。ORDERS表中的dollars值可以从qty和price两个值通过简单的相乘，然后计算给顾客所打的折扣(根据cid所对应的CUSTOMERS表中的discnt属性可得)而得到。我们也不需要再减掉给代理商的佣金，因为已经假定dollars代表了对顾客来说的所有开销。

这个CAP数据库的例子非常简单，并且人工的痕迹比较的明显，但是非常适合做本书的例子来说明我们的思想。如果是为了更加符合事实的话，那么我们就需要考虑更多、更大的表，首先我们需要更多的列。例如，AGENTS表中的aname应该还有名字；每个city列还必须

加上附属的街道、州和邮政编码等列;我们还需要记录所有的代理商在最近的几个月里的销售情况,就好像我们需要记录目前手头上还剩的商品的数目;我们需要记录每个顾客公司中的联系人的名字、掌管库存的人员的名字、住址、电话号码等等;我们可能还需要每个订单的更完整的日期和时间戳(只有月份是不够的);我们可能还需要记录人员的报酬;也有可能有不在代理商处工作的员工,我们还需要为他们另外记录工资、赋税等等的情况;我们也需要在表中包含更多的行。

CUSTOMERS			
cid	cname	city	discnt
c001	TipTop	Duluth	10.00
c002	Basics	Dallas	12.00
c003	Allied	Dallas	8.00
c004	ACME	Duluth	8.00
c006	ACME	Kyoto	0.00

PRODUCTS			
pid	pname	city	quantity
p01	comb	Dallas	111400
p02	brush	Newark	203000
p03	razor	Duluth	150600
p04	pen	Duluth	125300
p05	pencil	Dallas	221400
p06	folder	Dallas	123100
p07	case	Newark	100500

AGENTS			
aid	aname	city	percent
a01	Smith	New York	6
a02	Jones	Newark	6
a03	Brown	Tokyo	7
a04	Gray	New York	6
a05	Otasi	Duluth	5
a06	Smith	Dallas	5

ORDERS						
ordno	month	cid	aid	pid	qty	dollars
1011	jan	c001	a01	p01	1000	450.00
1012	jan	c001	a01	p01	1000	450.00
1019	feb	c001	a02	p02	400	180.00
1017	feb	c001	a06	p03	600	540.00
1018	feb	c001	a03	p04	600	540.00
1023	mar	c001	a04	p05	500	450.00
1022	mar	c001	a05	p06	400	720.00
1025	apr	c001	a05	p07	800	720.00
1013	jan	c002	a03	p03	1000	880.00
1026	may	c002	a05	p03	800	704.00
1015	jan	c003	a03	p05	1200	1104.00
1014	jan	c003	a03	p05	1200	1104.00
1021	feb	c004	a06	p01	1000	460.00
1016	jan	c006	a01	p01	1000	500.00
1020	feb	c006	a03	p07	600	600.00
1024	mar	c006	a06	p01	800	400.00

图2-2 CAP数据库(某一时刻的内容)

2.2 数据库各部分的命名

数据库中有两套标准术语。一套用的是表、列、行;而相应的另外一种就用关系(对应表)、元组(对应行)、属性(对应列)。我们将同时使用这两套术语,以便你很容易地识别。因此我们

会经常用表的列名指代该表的属性(如表CUSTOMERS的属性包括cid、cname、city和discnt)，用表的行来指代表中的元组。

数据库是表或者说是关系的集合。比如CAP数据库由下列表的集合组成：

```
CAP = {CUSTOMERS, AGENTS, PRODUCTS, ORDERS}
```

表的标题是表的列名集合，它出现在表的最上部，位于表的第一行(也就是元组)的上面。表T的标题用Head(T)标记。比如：

```
Head(CUSTOMERS) = {cid, cname, city, discnt}
```

在列出标题的列集合的时候，对于标记({x, y, ...})有一个约定的简化写法，可以简单地列出这些属性，用空格隔开。因此更通用的标题写法是：

```
Head(CUSTOMERS) = cid cname city discnt
```

表的标题也被称做关系模式，即组成关系的属性的集合。数据库的所有关系模式的集合构成了数据库模式。

表的行集合，也就是元组集合，称为表的内容，表的行数称为表的基数。比如，图2-2中的PRODUCTS表包含了7行数据。值得注意的是，表名以及表的标题是表长期存在的属性，而表的行集合在数量和详细值方面相对而言是时常变化的，这就是我们为什么在图2-2中要称表的内容是相对于“某一时刻”的原因。对于熟悉C和Java编程的读者来说，类似图2-2的表可以被看做储存记录或者结构的文件，表中的每一行数据一般说对应文件中的一条记录。记录中的字段的结构也可以由表的标题确定，然而在关系数据库依据程序-数据独立的原则应用程序编程人员与具体的存储是完全隔离的，数据库在理论上可以用各种完全不同的磁盘结构来储存表中的行。

1. 域和数据类型

表必须建立在一个计算机化的数据库系统之上(也就是说，空表的结构必须预先声明)，就像表所包含的记录(结构)需要用C或者Java声明。在目前的数据库产品(如ORACLE、DB2 UDB以及INFORMIX)中，表中的列必须有特定的类型。比如说，表CUSTOMERS中的discnt列的类型是real(实型——占四字节的浮点数)，并且city列的类型为char(13)(长度为13的字符串)。在理论性文献中，更一般地说，表T的属性A的取值范围是在集合D上，我们称D为域。域一般定义为可以被用作表的属性值的常数的集合，因此域在某些编程语言中通常对应于一些特定的枚举类型的概念。比如属性city的域(用CITY表示)可能是代表美国(假定我们的公司只和在美国的客户有业务往来)所有的城市的字符串的集合。在另一方面，属性discnt的域DISCNT可能被定义为从0.00到20.00的实数集合，并且可能这些实数必须精确到小数点后两位(它表示了我们的代理商们给他们的顾客所打的折扣的范围)。

大多数商业数据库系统都支持小数位数的显式精确度(比如DISCNT)，而不支持由枚举集合组成的类型(比如CITY)。我们通常不得不满足于这样的系统，也就是在city列中出现的值属于char(13)这样的原始类型。然而把类似于char(13)这样的域限制在出现在另外一个表的列中的枚举值的集合是可能的。通过使用约束类型也就是第7章中要讲述的引用完整性我们可以轻易地做到这一点。

给表的列加上一个特定的类型的意义在于，我们可以对声明在该类型上的两个不同的列的值进行比较。举个例子，如果CUSTOMERS表的city列的值声明为类型char(13)，而AGENTS表的city列的值声明为类型char(14)，那么对于要求将所有的在同一个城市的代理顾

客对打印输出的请求就无法肯定得到结果，因为理论上说数据库系统可能会拒绝在不同类型的列之间进行比较。

2. 表和关系

首先我们回顾一下笛卡儿积这个数学概念。

定义2.2.1 笛卡儿积 集合 S_1, S_2, \dots, S_k 的笛卡儿积 $S_1 \times S_2 \times \dots \times S_k$ 由所有的 k 元组 (e_1, e_2, \dots, e_k) 组成，这里 e_1 是 S_1 中的任意元素， e_2 是 S_2 中的任意元素， \dots ， e_k 是 S_k 中的任意元素。 ■

假设集合CID, CNAME, CITY和DISCNT分别代表了表CUSTOMERS中的属性cid, cname, city和discnt的域。属性A的域用Domain(A)表示，所以Domain(cid)=CID，如果c允许作为cid列的值，那么我们就有 $c \in CID$ 。现在我们来考虑集合CID, CNAME, CITY和DISCNT的笛卡儿积：

$$CP = CID \times CNAME \times CITY \times DISCNT$$

笛卡儿积CP由所有可能的四元组 $t=(c, n, t, d)$ 组成。这里c是CID中的顾客的标识号，n是CNAME(不一定对应于标识符为c的顾客)中的顾客名称，t是CITY集合中的某个美国城市名，d是DISCNT中的某个值。比如，笛卡儿积CP将包含下面的元组： $t=(c003, Allied, Dallas, 8.00)$ 和 $t'=(c001, Basics, Oshkosh, 18.20)$ 。

现在关系是一种数学结构，被定义为一个笛卡儿积的子集。比如，表CUSTOMERS是 $CP=CID \times CNAME \times CITY \times DISCNT$ 的一个子集。表CUSTOMERS中的所有行(也就是所有的4元组)都包含在笛卡儿积CP所表示的元组的集合当中了。对于上面那两个元组，t包含在CUSTOMERS中，而t'不在。因此，表CUSTOMERS是真子集——没有包含笛卡儿积中的所有元素的子集，因为并不是所有笛卡儿积的元素都是关系中的元素。我们称那些符合条件的笛卡儿积中的元素为相关的，并由此得到数学上的术语——关系。特别需要注意的是表中的每一行都与不同的列上的值相关联，所以表中的行的完整集合才被称为一个关系。

如图2-1中所示，每一个顾客都假定拥有唯一的cid值，因此包含所有笛卡儿积元素的CUSTOMERS表是没有任何意义的。只要CNAME, CITY和DISCNT中有一个域中包含超过一个值，笛卡儿积中必然包含两行在CID列上具有相同的值，但是CUSTOMERS表中CID列的值对于不同的行依旧保持唯一。

例2.2.1 使用刚引入的术语，如果我们定义一个表T的标题为

$$\text{Head}(T)=A_1 \cdots A_n$$

那么表T将是 $\text{Domain}(A_1), \dots, \text{Domain}(A_n)$ 的笛卡儿积的一个子集。 ■

前面提到过，表的标题是相对稳定的，它描述了表的行的结构。增加或者删除列都不是很常见的，我们也不希望在日常业务中这么做。在设计表的列布局的时候，我们必须根据数据库的需求做好全面的分析。第1章已经讲过，这一步称为数据库的逻辑设计。

另一方面，表的内容是经常变化的，新的行可以随时被增加或者删除。当我们新增一个客户的时候我们可以在数据库中添加一行数据；当我们失去一个顾客的时候，我们可以在数据库中删除对应的行。在大型的批发公司中跟踪这些数据的变化是不可能的因为有些表可能有上百万条记录；但是列的数量一般来说被限制在一定的范围之内。用户不需要特殊的知识，只要知道列的名字和特性就可以通过查询语言检索一个或者多个表的行的信息。如果你认为一个查询的准确结果和查询所涉及到的表的内容有关，那么你就错了。像CAP这样的小型数

据库是例外的:你可以利用CAP中的表的内容去检验查询结果是否符合要求。然而,查询正确性最终的测试要求对表内容的所有可能的改变能得到正确的结果。

2.3 关系规则

关系模型中一些众所周知的规则告诉我们在表结构中哪些变化是允许的,哪些检索操作是受限的。这些关系规则在各种不同的商业数据库产品的标准化工作方面起到了很大的作用,因此逻辑数据库设计(第6章将学习)的一些概念对于这些产品来说是完全一样的。很多商业化的数据库产品有目的地打破了某些关系规则,而新的对象-关系数据库模型更甚。正因为如此,读者应该对于下面将要讲述的概念保持充分的警觉,对你期望在某些商业数据库系统中遇到的变化应保持灵活。

规则1 第一范式规则 在定义的表中,关系模型坚持不允许含有多值属性(有时称为重复字段)和含有内部结构(比如记录类型)的列。遵守这样规则的表称为第一范式。 ■

作为一个例子,考虑图2-3中表EMPLOYEES,它包含对应公司雇员的行。表中含有唯一的雇员标识列eid、雇员的名字ename、雇员在公司的职位position以及一个多值字段(属性)列出了该雇员的家属。比如ID为e001的雇员John Smith有两个家属,Michael J.和Susan R.,被分列在不同的行上,而David Andrews有一个家属,Franklin Jones则有三个。

EMPLOYEES			
eid	ename	position	dependents
e001	Smith, John	Agent	Michael J.
			Susan R.
e002	Andrews, David	Superintendent	David M. Jr.
e003	Jones, Franklin	Agent	Andrew K.
			Mark W.
			Louisa M.

图2-3 具有多值属性列dependents的EMPLOYEES表

关系规则1表明图2-3中的重复字段dependents在关系表中是不允许的。这是在设计过程中的一个约束。如果我们要把家属名放入表EMPLOYEES(如图2-4所示)中的唯一的雇员行中,我们需要给表建立一定数目的家属列,这个数目要达到某一雇员可能有的最多的家属数,比如dependent1, dependent2, ..., dependent20。

EMPLOYEES					
eid	ename	position	dependent1	dependent2	...
e001	Smith, John	Agent	Michael J.	Susan R.	...
e002	Andrews, David	Superintendent	David M. Jr.
e003	Jones, Franklin	Agent	Andrew K.	Mark W.	...

图2-4 每个雇员行中具有多列家属名的EMPLOYEES表

但是这是不切实际的,因为它浪费空间并使得查询变得非常困难,所以一个有效的方法是将表EMPLOYEES分解成两部分,建立单独的表DEPENDENTS,该表包含两列,eid和

dependent(如图2-5所示)。

EMPLOYEES			DEPENDENTS	
eid	ename	position	eid	dependent
e001	Smith, John	Agent	e001	Michael J.
e002	Andrews, David	Superintendent	e001	Susan R.
e003	Jones, Franklin	Agent	e002	David M. Jr.
			e003	Andrew K.
			e003	Mark W.
			e003	Louisa L.

图2-5 表EMPLOYEES和相应的表DEPENDENTS

表DEPENDENTS每一行对应一个家属，给每个eid一个雇员-家属名字对。等一会我们将会看到，任何关系数据库的查询语言都允许我们将一个雇员行和拥有相同eid的家属行连接起来。

第一范式规则同时也要求任何列的值都必须是简单类型，不允许包含内部结构。比如图2-5中表EMPLOYEES的ename列是由简单的字符串组成的，通过一些分隔符来构成姓(逗号前的所有文本)。我们不允许建立一个带有含ename.fname、ename.lname和ename.mi(分别对应姓、名和姓名首字母三个组成部分的列的表，因为在第一范式中不允许某一列的值是结构类型。

我们曾经简单地提过，第一范式规则可以被对象-关系数据库系统中定义的表所打破。这些表是非第一范式(non-first normal form, NNFN)。在第4章我们会讨论对象-关系模型的细节。打破第一范式规则的途径是该模型允许表中对象列的值是包含一个复杂类型值的集合。比如，我们可以为person_name新定义一个类型包含数个字符串的成分fname, lname和mi，然后可以定义一个dependent_names类型为:setof(person_name)，这样声明为dependent_names类型的列就可以包含复杂名字的集合了。

规则2 只能基于内容存取行规则 关系模型的第二条规则说明我们只可以通过行的内容即每一行中所存在的属性值来检索行。 ■

就目前用户的查询而言，规则2说明了行是没有次序的。因此任何一个(纯)关系查询语言都不可以要求检索表ORDERS中的第三行，取而代之的应该是要求查询ordno列为1019的行，根据图2-1中ordno的定义，该行是唯一的。在某种意义上讲，图2-2中所显示的ORDERS表可能会误导读者。根据规则2，表ORDERS是没有第一行、第二行或者是第三行之分的，由表中行组成的表的内容是集合的无序元素。用更抽象的数学术语来说，可以简单地表述为关系是元组的集合。规则2的另外一个含义是你无法建立一个指向行的指针以便以后可以再次检索它，这种指针是不允许的。

有些商业数据库系统甚至连规则2都打破了，通过行标识(row identification, RID)提供一种用户检索表中的行的方法，通常把行标识称为ROWID(读作row-eye-dee)，或者称为元组标识(TID，读作tee-eye-dee)。在一般情况下，RID可以很容易在数据被导入数据库的时候通过行的数目计算出来。商业数据库系统常声称规则2是不合理的，需要一个新的标准。为了确保可以通过RID值来访问数据行，需要把数据库中的行以特定的实际次序存储在非易失性的介

质上，比如磁盘(非易失性指的是机器的电源供应中止的时候介质仍然可以保持数据)。数据存储的次序是可以预见的，在第7章可以看到，通过RID检索数据对于数据库管理员(DBA)来说是非常有用的。它可以帮助数据库管理员检查在什么地方表中的行记录没有按正确的方式存放。目前的磁盘存储技术显示数据行互相之间存放得越近，检索数据越快。我们可以在本书的后面部分检验其正确性。总而言之，RID的值是很重要的，我们不需要考虑行记录存储的实际次序。同时，大多数的用户是不会对RID的值进行查询的，因为RID的值当表在特定场合下更新以后随之发生变化。

对象-关系模型允许某一行上列值指向另外一行，这和关系规则是严重违背的。

值得注意的是，关系模型同样也要求在一个关系中的列是没有次序的。因此我们再一次要说明我们所提供的表模型是有误导作用的。标准的SQL语言打破了这项规则，我们在第3章可以认识到这一点。

规则3 行唯一性规则 关系模型的第三条规则要求关系中的任何两个元组(表中的行)的值在同一时刻不能是完全相同的。关系可以被看做是关系模型中的元组的集合，理所当然集合是不允许包含相同元素的，集合中的任意元组必须唯一。而且，由于纯关系查询语言只能通过它们的列值来区分行(规则2)，这也说明了必定有另一个方法可以区分任何一行数据和其他的行的值，这样查询语言语句就可以唯一的检索出它来。 ■

在商业数据库系统中，很大一部分工作要求保证在插入一条新记录的时候，原数据库中没有相同的记录存在。而行的唯一性规则也正是大多数情况下合理的目标。SQL中的Create Table语句如果没有特殊的限制(维护起来花销更大)的话并没有保证这一点。在本章讲述的关系模型当中，行的唯一性规则具有特殊的含义，因此在以下的讲述当中，如果没有特别指明，我们都假定所有的表都遵守规则3。这个假定到我们讲述商业数据库系统中的表为止。

关系规则的动机是什么呢？这些规则(以及其他)在关系模型的发明者E·F·Codd的一系列文章中就提出了。这些规则反映了特定的数学假定，对于关系结构的良好性状有重大意义。在随后的章节中我们将使用大量的例子。比如，规则3，反映了这样的一个数学思想：关系是元组的集合，集合不可以包含相同的元素。规则1，从图2-3到2-5可以看出，对于数据库表的设计具有相当大的意义。规则2，要求只能通过行的内容来访问行，保证了本章稍后提供的检索行的方法是唯一被允许的方法。因此，一些关系数据库以前的数据库产品不能通过简单地定义几个表来储存数据，保持它们原先的数据访问方式，也无法再称之为关系产品。就如前面所提到的一样，关系规则在各产品的标准化方面起到了很重要的作用，它保证了在不同的系统中的数据库设计的规则都相同。出现新的对象-关系模型是因为很多人认为处于数学动机上关系规则实在是太严格了，以至无法容纳一些有价值的思想，很多这样的思想第一次出现在对象-关系的编程语言中。通过允许新的思想，对象-关系模型提供了许多在商业方面很有应用价值的扩展性质。在本章的剩余部分，如果没有特殊指明，我们都假定关系模型都遵守所有的关系规则。

2.4 键、超键和空值

关系规则3说明了表中的任意两行数据在所有列上的值不可以都相同，也就是说两行不同的数据可以通过某些列上的值区分开来，或者以不同的视角看这个问题，所有的列的集合将区分任意两行数据。那么是否有可能列集合的子集也能区分出任意两行呢？是的！举个例子，

在表CUSTOMERS中，单一的列cid就可以唯一区分任意两行数据(回忆一下图2-1，我们定义cid为顾客的标识符，任意行上的cid值唯一)。我们称cid是表CUSTOMERS的一个超键。意思就是说表中的任意两行数据在该列集合上都有唯一的值。列cid实际上是表CUSTOMERS的一个键。键是一个更为严格的事物，意思是组成键的列的集合中再也没有子集也是表的超键。

现在一个很重要的问题是：pname是不是表PRODUCTS的超键？即使在图2-2中显示表中的每一行都包含不同的pname值，而答案仍然是否定的！这是因为当我们说超键是用来唯一区分表中的任意两行数据的列的子集的时候，我们是从数据库设计者(通常是DBA)的角度来看这个问题的。在我们的设计当中从来没有想过要把pname作为表PRODUCTS的唯一标识。我们建立列pid作为行的标识(如图2-1所示)，而图2-2中的pname恰好没有重复值这个事实仅仅是个巧合，因为那是在某个特定时刻的表的内容，可能在下个时刻就改变了。比如，我们新插入一条记录，pid为p08，pname为folder；现在我们将有两行数据的pname都为folder，通过它们的pid值为p06和p08我们可以区分它们，虽然这两行可能拥有不同的大小。

我们也可以将大小也嵌入到名字里去(如half-inch folder)，但是这可能是不够的，两个不同的folder可能有相同的大小、不同的颜色或者不同的扣子或者是由不同的材料做出来的。所有这些区别可以通过实际表中的其他列(颜色，尺寸，材料等等)来表达出来。但是很容易想象当表的结构固定下来后，一个新出现的商品可能和其他的商品在所有的列上的值都相同。为了区分这样的两个商品的行，我们需要一个pid列。表PRODUCTS中包含单个唯一区分行的列标识符是非常有效的；正是由于这个原因我们决定定义一个唯一标识符pid，然后只依赖于pid的值唯一区分行。

表中新添加行的时候需要维持表的键或者超键，也就是说表的键表达了数据库设计者的意图，不管将来面对什么样的情况，并不仅仅表达在某一时刻表的结构。接下来，我们要给出几个定义，把键和超键的概念表达得更详细一点。

给定表T，其属性用带下标的字母表示， $\text{Head}(T)=A_1 \cdots A_n$ 。表T中有一元组t。在 $\{A_1, \dots, A_n\}$ 的子集 $\{A_{i_1}, \dots, A_{i_k}\}$ 上的定义元组t的约束(记为 $t[A_{i_1}, \dots, A_{i_k}]$)定义为命名的列上的t值的k元组。比如，对于表CUSTOMERS中的元组

$$t=(c003, \text{Allied}, \text{Dallas}, 8.00)$$

的约束为由列cid和cname组成的集合，记为 $t[cid, cname]=(c003, \text{Allied})$ 。

利用这个标记，我们可以给出键的形式化的定义。

定义2.4.1 表的键 给定一个表T，标题 $\text{Head}(T)=A_1 \cdots A_n$ 。表T的一个键，有时也称为候选键，是具有以下两个特征的一组属性的集合K= $A_{i_1} \cdots A_{i_k}$ ：

1) 如果u，v是T中的两个不同的元组，根据设计者的意图 $u[K] \neq v[K]$ ；这就是说，集合K中必定存在至少一个列 A_{i_m} ，使得 $u[A_{i_m}] \neq v[A_{i_m}]$ 。

2) 没有K的真子集H具有特征1。 ■

条件1用数学方式说明了行u在属性集K上的值是唯一的。我们称满足条件1而不满足条件2的属性集为超键。条件2保证了键是满足条件1的最小的属性集，因此键同时也是一个超键，但是键另有附加的特征，即它没有真子集同时也是超键。满足条件1的单个属性总是最小的，因为属性的空集 \emptyset 无法唯一的区分两行数据：对于所有的u和v，总有 $u[\emptyset]=v[\emptyset]$ 成立。

在一个可计算机化的数据库系统中，数据库设计者，通常是DBA，使用本书后面所讲述的特定的语法可以定义表中的键。一旦这样的键被定义，如果对于数据库的普通更新操作(插

入新行或更新现存的某行的列值)使得键的唯一性条件受到破坏,那么这些操作是不允许的。注意到当我们说到“设计者的意图”时,这也许并不是由设计者率先提出。举个例子来说,DBA需要保证socsecno列是表EMPLOYEES的键,即使设计者已经建立了特定的eid属性作为雇员的标识。然而,社会保障号当然是唯一的,任何重复的值都意味着表中的错误。

表可能具有不只一个键,我们来看一下如下的例子。

例2.4.1 考虑给定的表T:

T			
A	B	C	D
a1	b1	c1	d1
a1	b2	c2	d1
a2	b2	c1	d1
a2	b1	c2	d1

正如我们已经说明的,表在某一时刻的内容无法告诉我们表中的键是什么,因为键跟表的设计者的意图有关。然而,为了说明我们需要从给定的内容中计算出T的键,我们强加了一个很少见的条件:表中的内容是由设计者设定,在整个数据库存活期都保持不变。有了这个前提条件,我们可以通过确定哪些列的组合可以使得行数据保持唯一来推导出T的键。

一开始我们注意到表T中没有任何一个单独的属性可以是键,因为在每列上都至少有两个相同的值;另一方面,没有任何一个包含D的属性的集合S可以是键,因为D对于区分T中的行数据没有任何帮助,那么S-D同样也可以区分T的所有行,所以S是T的一个超键,而不是一个键。下一步考虑所有T中不包含D的属性对:AB、AC和BC。表T的所有行在这些属性对上的值都保持唯一。任何包含D或者AB、AC或BC作为真子集的其他属性集合都是超键(可以列出所有其他的集合验证这一点)。因此,AB、AC、和BC是表T的全部键。 ■

关系规则3保证了所有列的集合可以唯一区分任意两行,因此关系中至少存在一个键,其证明可以很好地说明我们所给出的定义是怎么使用的。

定理2.4.2 每个表都至少有一个键。

证 给定一个表T,标题 $\text{Head}(T)=A_1 \cdots A_n$,我们考虑所有属性的集合 S_1 , S_1 必定是一个超键,因为T中没有两行记录 u 和 v 在 S_1 中的所有列上都具有相同的值。对于任意不相同的行 u 和 v ,在 S_1 中存在一属性(列) A_i 使得 $u[A_i] \neq v[A_i]$ 。我们假定已经知道表设计者的意图,并且可以识别出可以作为表的超键的属性集。现在, S_1 要么是键,要么另外存在一个 S_1 的子集 S_2 也是超键。继续这样的推导,我们可以得到这样的集合链: $S_1, S_2, \dots, S_i, S_{i+1}, \dots$ 。在这个链中的每个集合都是其前面集合的真子集,并且链上的所有集合都是超键。为了表明有键存在,我们只需要证明这样的集合链最终会有个终点:那么链尾集合 S_k (S_k 包含在链上所有的 S 集合中)没有具备作为超键所要求的属性的更小的子集。

我们可以用如下的方式说明集合链最终会有个终点。用 $\#S_i$ 来代表集合 S_i 中的属性的个数。 S_1 有 n 个属性在标题 $\text{Head}(T)=A_1 \cdots A_n$ 中,所以 $\#S_1 = n$ 。另外,我们有 $\#S_i > \#S_{i+1}$,因为在链中的所有后继集合都是前一集合的真子集。最后,每个集合中的属性个数都是为正的,因为一个负的数目是毫无意义的,而空集不可以是一个超键。因此上面给出的集合链是一个递减的序列: $n = \#S_1 > \#S_2 > \cdots > \#S_i > \#S_{i+1} > \cdots > 0$ 。这样的序列肯定存在一个最小值,也就是序列中的最后

一个数 $\#S_k$ ，所以 S_k 没有可以作为超键的真子集，从而 S_k 是表T的键。这样我们就可以找到一个由不同整数组成的有限集中的一个最小元素。但是一个数学家会要求我们通过数学推导来证明这一点。■

本书中并不经常需要严格的数学证明，然而，你应该明白为什么为了证明 S_k 的存在性而需要这些证明。这就像当我们推出一个循环程序来计算键的属性集，并且我们需要保证循环总是会终止的，这一点值得注意。

关系的各种键通常被称为候选键，这个定义暗示着候选键中的某个被指定为主键的选择过程。

定义2.4.3 表的主键 表T的主键是被数据库设计者选择出来作为表T中特定行的唯一性标识符的候选键。■

通常主键标识符被用来作为别的表中的引用。如图2-5所示，表DEPENDENTS包含eid列指向表EMPLOYEES中的键列eid。在CAP数据库中，表ORDERS用ordno属性来唯一区分每一行。一个扩展的PRODUCTS表可能除了pid外另有其他的候选键，由表示名字、大小、颜色以及零件的材料等的列的集合组成。不过用这样的属性集合作为行的标识显得笨拙了点，所以我们还是选择了pid作为主键。表AGENTS也可以有其他的候选键，比如socsecno(社会保障号)。但是看起来给所有的雇员一个ID(eid或者aid)作为主键使用起来更为方便，这正是目前大多数公司采用的，因为某些雇员可能在拥有社会保障号之前就开始上班了。

空值(NULL Value)

假定一个新的订书机产品要加到PRODUCTS表中，但是我们还没有决定已经存到仓库中的产品的quantity或者仓库所在的城市都没有确定。但是我们知道我们手头上已经有足够的订书机来应付对于该商品的订单。所以我们需要提前就开始这项业务而不需等到知道所有的细节(等到我们知道手头上的库存数目以后，按照订单送出货，因此库存数可以在我们送出货以后保持准确)。为了记录PRODUCTS表中的订书机信息，我们用特殊的值——空值来表示新行中的city和quantity属性。

pid	pname	city	quantity	price
p07	stapler	null	null	3.50

这里所用的空值应该被解释为未知的或者是尚未定义的，意思是当我们以后知道更多情况的时候会重新填写该值。这和当一个字段的值是不适用的时候用null填充的情况稍有不同。比如，如果我们在雇员表中为公司总裁填写经理姓名这一列(总裁上面没有经理)的时候，或者是给一个没有佣金的雇员填写佣金比例的时候，我们只好用空值来代替。特别值得注意的是空值和数字0(对于一个数值属性)或者是空串(对于字符串属性)的区别。举个例子，当我们要求计算手头上的商品的平均数量的时候，订书机的0值是要被计算到平均值中去的，而一个空值意味着特殊含义，在计算平均数量的时候就把订书机的数目从中剔除出去了。

表的主键是用来唯一区分表的单独行的。我们现在讨论另外一条要求主键不能包含空值的关系规则。比如，我们在表PRODUCTS中新增了一个商品名为stapler的新元组，虽然我们并不知道这个新商品的city和quantity的属性值，这样我们就可以开始在接受商品的订单。但是我们还没有给它分配一个pid值，因为我们需要在ORDERS表中列出所有已订购产品的

pid值，所以我们还不可以存储这行数据。因为pid值是行的指定标识符，在我们还没有确定标识符的值之前，不允许该行存储到表中去。

主键通常是特殊的指定候选键，可能包含多个列。我们在其上增加空值限制来说明实体完整性规则。

规则4 实体完整性规则 表T中的任意行在主键列的取值都不允许为空值。 ■

我们将在后面的章节中更详细地讨论空值的性质。

2.5 关系代数

关系代数由E.F.Codd在一系列文章中率先提出，最早出现在1970年，到了1972年基本达到当前的形式。关系代数的目的在于演示一个查询语言从关系数据库系统中检索信息的潜力。尽管另外一些查询语言可能在一般的使用上性能更优越。关系数据库系统中用表的形式存储的信息，用这样的形式来显示查询结果显得很自然。关系代数可以被看做是根据查询结果来生成新表的方法的集合，这些方法称为关系代数运算。

关系代数是一种抽象的语言，这意味着无法在一台实际的计算机上执行用关系代数形式化的查询。之所以在这里引入关系代数，是为了用最简单的形式来表达所有关系数据库查询语言必须完成的运算的集合。这些基本的运算对于理解第3章是非常有用的，在那里我们要解释在关系数据库系统的标准查询语言SQL查询是如何被执行的。

在本节的剩余部分，我们使用在2.1节中介绍的CUSTOMERS-AGENTS-PRODUCTS(CAP)数据库来介绍不同的运算。我们给CAP数据库增加一些补充表。

关系代数的基本运算

我们把关系代数的运算分成两种类型：集合论的运算，因为表实际上是行的集合；另一种是自然关系运算，主要是在行的结构上。给定两个表R和S，表R的标题Head(R)=A₁,...,A_n，在大多数情况下Head(S)与其相同。我们定义如下八种基本运算来从R和S中产生新的表。下面给出的键盘格式是用来在不包含指定的特殊符号的键盘上使用的。除了这些运算以外，我们还要定义一种存放中间结果的方法，就好比在C语言或者Java语言中的赋值语句。

集合运算

名称	符号	键盘格式	示例
并	U	UNION	R U S, 或 R UNION S
交	cap	INTERSECT	R cap S, 或 R INTERSECT S
差	-	- 或 MINUS	R - S, 或 R MINUS S
乘积	times	TIMES	R times S, 或 R TIMES S

自然关系运算

名称	符号	键盘格式	示例
投影	R []	R []	R [A _{i1} ...A _{ik}]
选择	R where C	R where C	R where A ₁ = 5
连接	bowtie	JOIN	R bowtie S, 或 R JOIN S
除	÷	DIVIDE BY	R ÷ S, 或 R DIVIDE BY S

2.6 集合运算

本节假定表是被定义为行的集合，根据关系规则1、规则2、规则3有：表中只有简单的列值，集合中没有重复的行，行无次序。

需要掌握的集合运算有：并、交、差、笛卡儿积(参见定义2.2.1)。这些是前四种关系代数运算所基于的集合运算。对于并、交、差运算所涉及到的行的集合要求具有相同的标题结构。举个例子来说，我们无法建立一个包含顾客信息表和商品信息表的“并”组成的新表。因为不可能在相同的标题结构下给所有的CUSTOMERS表中的行以及PRODUCTS表中的行一个合适的表结构，因此我们给出以下的定义。

定义2.6.1 兼容表 如果表R和S具有相同的标题，也就是说，如果 $\text{Head}(R)=\text{Head}(S)$ 而且属性是从相同的域中选择并具有相同的含义，则表R和S是兼容的。 ■

1. 并、交和差运算

仅当两个表是兼容表的时候才可以做并、交、差运算。

定义2.6.2 并、交、差 R和S是两个兼容表， $\text{Head}(R)=\text{Head}(S)=A_1 \cdots A_n$ 。R和S的并是 $R \cup S$ 所得的表，具有同样的标题，由属于R或者属于S或者属于两者的所有行组成；R和S的交是 $R \cap S$ 的表，由既属于R又属于S的行组成。R和S的差是 $R - S$ 的表，由属于R而不属于S的行组成。 ■

注意， $R \cup S = S \cup R$ ， $R \cap S = S \cap R$ ，但是 $R - S$ 一般来说和 $S - R$ 不同。所有的关系运算是可以递归的，所以定义2.6.2中的表R和S也可以是其他关系代数表达式的结果。我们可以把这些表达式外面加上圆括号而不改变所有表达式的含义。比如，我们可以用 $(R - S)$ 来代替 $R - S$ 。这对于说明计算次序是非常有用的。在一般的数学表达式中，嵌在圆括号内的子表达式先被计算，因此对于表达式 $R - (S - R)$ 意味着我们首先计算 $S - R$ ，得到结果为T，然后计算 $R - T$ 得到最终结果。

例2.6.1 考虑表R和S：

R		
A	B	C
a1	b1	c1
a1	b2	c3
a2	b1	c2

S		
A	B	C
a1	b1	c1
a1	b1	c2
a1	b2	c3
a3	b2	c3

$R \cup S$ 有五行：

R \cup S		
A	B	C
a1	b1	c1
a1	b1	c2
a1	b2	c3
a2	b1	c2
a3	b2	c3

注意到在R和S中同时出现的只有两行，所以 $R \cap S$ 有两行记录：

$R \cap S$		
A	B	C
a1	b1	c1
a1	b2	c3

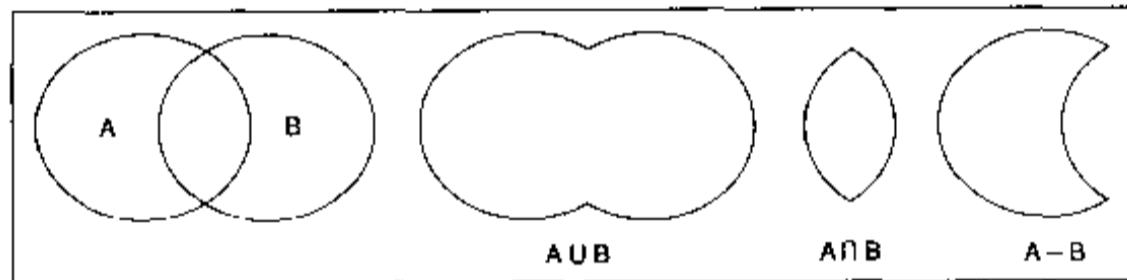
$R - S$ 只有一行记录：

$R - S$		
A	B	C
a2	b1	c2

而 $S - R$ 有两行：

$S - R$		
A	B	C
a1	b1	c2
a3	b2	c3

例2.6.2 交、并、差运算通常可以用文氏图(Venn Diagram)来示意，如下所示：



这种说明的格式有时也作为某些等价原理的演示，稍后可以看到。 ■

2. 赋值和别名

在关系代数表达式计算过程当中，有时候存储中间结果是很有用的。我们接下来要介绍一种符号使得关系代数具有该功能。

定义2.6.3 赋值、别名 R 是一个表， $\text{Head}(R)=A_1 \cdots A_n$ 。假定 B_1, \dots, B_n 是 n 个属性。对于所有的 $1 \leq i \leq n$ ，它们的域 $\text{Domain}(B_i)=\text{Domain}(A_i)$ 。通过赋值

$$S(B_1, \dots, B_n) := R(A_1, \dots, A_n)$$

我们定义一个新表 S ，且 $\text{Head}(S)=B_1 \cdots B_n$ 。新表 S 的内容恰好和旧表 R 的内容相同，也就是说行 u 在 S 中当且仅当在 R 中存在一个行 t ，对于所有的 $1 \leq i \leq n$ 有 $u[B_i]=t[A_i]$ 成立。在赋值中使用的符号 $:=$ 称为赋值运算符。

我们已经努力使得赋值运算允许对源表的标题里的属性名进行重定义，我们不久就可以体会到这一点的价值。现在我们需要指明，属性的重定义并不总是必要。如果两个表中所有的属性名都是一对一的，并且对于所有的 $1 \leq i \leq n$ 都有 $B_i=A_i$ 成立，我们可以简单地称 S 为表 R 的别名，简写为 $S=R$ 。 ■

注意到在公式右边的表 R 也可以是关系代数的表达式求值，因此，只要我们在编程语言中使用赋值。这就允许我们把表达式计算的中间结果“保存”下来。在公式左边的表 S 必须是一个命名的表，它不可以是一个表达式。

例2.6.3 考虑例2.6.1给出的两个表R和S。使用赋值运算我们定义一个新表:

$$T := (R \cup S) - (R \cap S)$$

表T具有如下的形式:

T		
A	B	C
a1	b1	c2
a2	b1	c2
a3	b2	c3

我们也可以通过先定义两个中间表来定义表T:

$$\begin{aligned} T1 &:= (R \cup S) \\ T2 &:= (R \cap S) \\ T &:= T1 - T2 \end{aligned}$$

■

例2.6.3中的两个中间表T1和T2不是必需的，因为关系代数表达式允许任意嵌套。如果我们给定的表达式expr1中包含了T1，而T1又通过赋值由包含其他表的表达式expr2表示，那么我们可以用expr2取代expr1中出现的所有T1而得到相同的结果。赋值运算的最主要目的是为了使接下来的章节中演示复杂表达式的中间结果的计算更容易理解。

3. 乘积运算

乘积运算是基于集合运算中的笛卡儿积的。假定R和S是两个表，并且 $\text{Head}(R)=A_1 \cdots A_n$, $\text{Head}(S)=B_1 \cdots B_m$ 。R和S的乘积可以允许我们建立一个包含两个表中行之间所有可能的相互联系的新表。 r 是R中的元组，值为 $(r(A_1), \dots, r(A_n))$ 。 s 是S中的元组，值为 $(s(B_1), \dots, s(B_m))$ ，笛卡儿积的定义说明R和S的乘积包含了所有的R和S的元组对，形如 $((r(A_1), \dots, r(A_n)), (s(B_1), \dots, s(B_m)))$ 。表R和表S的列的差别——它们的值分别落到了在笛卡儿积中的一个元组的不同列上——和我们的关系表的定义(要求不区分列)是相违背的，因此我们需要一个更面向关系的定义。

定义2.6.4 乘积 表R和S的乘积是标题 $\text{Head}(T)=R.A_1 \cdots R.A_n S.B_1 \cdots S.B_m$ 的表T。我们称t是T中的行当且仅当存在R上的一个行u, S上的一个行v, 而t是u和v串接, 即 $u||v$ 。我们称t是T上的行当且仅当存在R上的行u, S上的行v并且对所有的 $1 \leq i \leq n$, 有 $t(R.A_i)=u(A_i)$ 成立, 对所有的 $1 \leq j \leq m$, 有 $t(S.B_j)=v(B_j)$ 成立。R和S的积T用 $R \times S$ 表示。

■

我们用W.A的形式来表示一个属性的名字(W是相乘表中的表名)，并称之为限定属性名，或限定属性。乘积表的标题由限定属性组成，并且当我们需要重点指出属性是从哪个表中取出来的时，可以用这些限定属性，因为在多个乘积表中同时出现相同的列名，有可能造成混乱。如果属性名只出现在一个表中，我们可以直接用它的非限定属性。

例2.6.4 考虑表R和S以及它们的积如下所示:

R		
A	B	C
a1	b1	c1
a1	b2	c3
a2	b1	c2

S		
B	C	D
b1	c1	d1
b1	c1	d3
b2	c2	d2
b1	c2	d4

$R \times S$					
R.A	R.B	R.C	S.B	S.C	S.D
a1	b1	c1	b1	c1	d1
a1	b1	c1	b1	c1	d3
a1	b1	c1	b2	c2	d2
a1	b1	c1	b1	c2	d4
a1	b2	c3	b1	c1	d1
a1	b2	c3	b1	c1	d3
a1	b2	c3	b2	c2	d2
a1	b2	c3	b1	c2	d4
a2	b1	c2	b1	c1	d1
a2	b1	c2	b1	c1	d3
a2	b1	c2	b2	c2	d2
a2	b1	c2	b1	c2	d4

在表 $R \times S$ 中，属性A和D都是唯一的，因此可以不用限定就可以引用；而属性R.B和S.B就必须要加上限定符名(简称限定符)R和S才能区分开来，在上表的表 $R \times S$ 中，所有列名都是限定的。 ■

这个例子说明我们可以在用乘积运算来由相对较小的表构造大表。举个例子，对于两个具有1000行的表的积可以有100万行。

当我们试图计算表它与其本身的乘积的时候，即考虑 $R \times R$ 这种情况，那么定义2.6.4暗示我们限定符名会存在问题。因为 $R \times R$ 具有标题 $R.A_1 \cdots R.A_n, R.A_1 \cdots R.A_n$ ，根据这些名字无法分辨出单一属性是属于哪一部分，因此乘积 $R \times R$ 是不被允许的。这种情况可以通过表R的别名来避免，使用赋值 $S := R$ ，然后计算R和它的别名S的乘积： $R \times S$ 。

2.7 自然关系运算

我们定义了四个自然运算用来处理表的关系结构。它们是投影、选择、连接和相除。像定义乘积运算一样，我们处理表R和S，其标题分别为 $\text{Head}(R) = A_1 \cdots A_n$ 和 $\text{Head}(S) = B_1 \cdots B_m$ 。

1. 投影运算

投影运算作用在一个表上，删除了表上的某些列，包括标题以及表内容上对应的列上的值。我们说把表投影到没有被删除的列的集合上。注意到表R上不同的行在投影到列的子集上后可能会相同，因为用于区别两个值的列已经被删除掉了。当发生这种情况的时候，投影运算符同样删除重复的行，只在结果集中留下重复行的一个拷贝。下面是一个更为严格的定义。

定义2.7.1 投影 R在属性 $A_{n_1}, \dots, A_{n_k}(\{A_{n_1}, \dots, A_{n_k}\} \subseteq \{A_1, \dots, A_n\})$ 上的投影是标题为 $\text{Head}(T) = A_{n_1} \cdots A_{n_k}$ 的表T，包含如下内容。对于所有的R上的行r，在T上也存在一个唯一的行t，对于所有的 $A_{n_j} \in \{A_{n_1}, \dots, A_{n_k}\}$ 有 $r[A_{n_j}] = t[A_{n_j}]$ 成立。R在 A_{n_1}, \dots, A_{n_k} 上的投影用 $R[A_{n_1}, \dots, A_{n_k}]$ 表示。 ■

投影运算把表中没有在方括号中的属性列表中命名的列给清除掉了。

例2.7.1 假设我们需要把图2-2中表CUSTOMERS的顾客名全部打印出来，但是不需要包

含他们的标识符号码、城市和折扣量，这可以在关系代数中实现，通过写

$CN := CUSTOMERS[cname]$

结果表CN是表CUSTOMERS中列cname上的内容。

CN
cname
Tiptop
Basics
Allied
ACME

注意到图2-2中显示两个重复的ACME值经过投影后变成了单一的行。 ■

这是我们第一次用关系代数表达式来回答用自然语言表达的对数据的查询要求，我们称这样的表达式为关系代数查询，简称为查询。

2. 选择运算

下一个要定义的是选择运算，从给定的表中选择出满足特定准则的行来构成新的表。指定这些准则的条件的具体形式由下面给出的定义来表达。

定义2.7.2 选择 给定一个表S, $\text{Head}(S)=A_1 \cdots A_n$, 选择运算定义一个新表，用

$S \text{ where } C$

表示，具有相同的属性集合，包含了S中满足选择条件(简称条件，记为C)的元组。根据是否满足条件来决定对于每个给定的表的元组是否应该被选择出来保留在行集合中。条件C的形式可以通过如下的递归定义表示：

1) C可以是任何形如 $A_i \alpha A_j$ 或者 $A_i \alpha a$ 的比较，这里 A_i 和 A_j 都是S中具有相同域的属性， a 是 $\text{Domain}(A_i)$ 中的一个常数， α 是比较符号，可以是 $<$, $>$, $=$, \leq , \geq 和 \neq (比较符 \neq 在两个被比较值不等时为真)等。表 $S \text{ where } A_i \alpha A_j$ 包含所有满足条件 $t[A_i] \alpha t[A_j]$ 的行 t 。而表 $S \text{ where } A_i \alpha a$ 包含所有满足条件 $t[A_i] \alpha a$ 的行。

用图2-2中的CUSTOMERS表举例，条件可以写为 $\text{city}='Dallas'$ 和 $\text{discnt} \geq 8.00$ 。注意到条件中的字符常量是放在引号中间的。在CUSTOMERS表中没有属性对具有相同的值，但是对于比如 $\text{city} > \text{cname}$ 这样的一个条件，这里操作符“ $>$ ”要求第一个操作数中的字符常量在字母表中的位置位于第二个操作数之后。

2) C和C'都是条件，那么新的条件可以是 $C \text{ AND } C'$, $C \text{ OR } C'$ 或者是 $\text{NOT } C$ ，可能需要把新生成的条件放入圆括号中去。如果 $U := S \text{ where } C_1$, $V := S \text{ where } C_2$ 那么我们有：

- AND连接符: $S \text{ where } C_1 \text{ and } C_2$ 意思如同 $U \cap V$ 。
- OR连接符: $S \text{ where } C_1 \text{ or } C_2$ 意思如同 $U \cup V$ 。
- NOT连接符: $S \text{ where not } C_1$ 意思如同 $S - U$ 。

表“ $S \text{ where } C$ ”包含了S中所有满足条件C的行。条件的计算是通过检查定义2.7.2中的第一部分的子表达式的比较，然后再检查复杂的逻辑语句是否正确。 ■

例2.7.2 为了找到所有在Kyoto的顾客，我们需要如下的选择：

$\text{CUSTOMERS where city} = 'Kyoto'$

这个查询的结果根据图2-2如下表：

cid	cname	city	discnt
c006	ACME	Kyoto	0.00

另外一个例子，假设我们需要找到所有存放在Dallas的价格超过\$0.50的商品，我们可以用如下的选择：

```
PRODUCTS where city = 'Dallas' and price > 0.50
```

该查询的结果如下表：

pid	pname	city	quantity	price
p05	pencil	Dallas	221400	1.00
p06	folder	Dallas	123100	2.00

目前所提到的运算结合起来可以解决更为复杂的查询。

例2.7.3 很容易我们就可以写出一个关系代数表达式来检索所有佣金比例超过6%的代理商。我们使用选择运算并定义表L：

```
L := AGENTS where percent >= 6
```

这个查询所得的结果表如图2-6。

```
M := AGENTS where percent >= 6
```

aid	aname	city	percent
a01	Smith	New York	6
a02	Jones	Newark	6
a03	Brown	Tokyo	7
a04	Gray	New York	6

图2-6 L:=AGENTS where percent>=6

现在假设我们需要检索所有的代理对，它们都具有超过6%的佣金比例，并且都在同一个城市。我们通过使用乘积运算符可以做到。但是我们需要小心，不要使用L × L，原因我们已经在2.6节解释过了，所以我们从另外的一个别名定义开始。

现在我们可以给出解决问题的表达式：

```
PAIRS := (L × M) where L.city = M.city
```

有了表M，它和L的区别仅仅在于具有不同的名字，我们就可以在乘积L × M上定义一个条件，来检索满足具有相同城市的记录。基于图2-2的PAIRS表达式的结果如下：

PAIRS

L.aid	L.aname	L.city	L.percent	M.aid	M.aname	M.city	M.percent
a01	Smith	New York	6	a01	Smith	New York	6
a01	Smith	New York	6	a04	Gray	New York	6
a02	Jones	Newark	6	a02	Jones	Newark	6
a03	Brown	Tokyo	7	a03	Brown	Tokyo	7
a04	Gray	New York	6	a01	Smith	New York	6
a04	Gray	New York	6	a04	Gray	New York	6

这个表包含了大量的冗余信息。实际上仅一个aid对(a01, a04)就可以满足我们所要求的，但是PAIRS的关系代数表达式同时把逆序的aid对(a04, a01)以及相同的代理对比如(a01, a01)，也表示出来所以需要建立一个更为严格的选择条件使得一次就可以取出唯一的代理组合来。

PAIRS2 := (L X M) where L.city = M.city and L.aid < M.aid

这个表达式的结果如下：

PAIRS2

L.aid	L.aname	L.city	L.percent	M.aid	M.aname	M.city	M.percent
a01	Smith	New York	6	a04	Gray	New York	6

■

例2.7.4 我们注意到在最后的答案中，例2.7.3中的PAIRS2，只有一行。再看看图2-2我们就知道这是为什么了：只有两个代理商是在同一个城市的，这个城市是New York。现在我们来回答下面这个问题，我们是不是可以用一个稍有不同的查询PAIRS3，直接指定城市为New York而不指定同一个城市来取代查询PAIRS2呢？也就是说我们用

PAIRS2 := (L X M) where L.city = M.city and L.aid < M.aid

来代替

PAIRS3 := (L X M) where L.city = 'New York' and M.city = 'New York' and L.aid < M.aid

如果不可以，为什么呢？虽然这两个查询给出了相同的结果。这个问题说明了前面所提到的一个常犯的错误，所以需要在这里特别指出。实际上，即使它们似乎给出了相同的答案，两个查询是完全不同的。这两个查询的结果仅在某一时刻的表的内容上是相同的。回想我们在图2-2中特别强调这些表的内容会经常发生变化，我们的查询应该在不知道表内容的情况下给出正确的结果。如果我们突然决定在图2-2中的表AGENTS加入一个新的代理商元组(a07, Green, Newark, 7)，那么我们的PAIRS2的结果就会有变化，因为现在有另外一对代理商(a02, a07)也具有至少6%的佣金比例，且在同一个城市(Newark)。这正是我们所要求查询的，这个查询具有自适应能力。然而，PAIRS3的结果不会改变，因为我们错误地用了一个依赖AGENTS表内容的条件(L.city='New York' and M.city='New York')代替了所希望表达的条件(L.city=M.city)。在建立我们自己的关系代数查询表达式的时候，要小心地避免这样的错误，我们称之为内容依赖。对于某个特定时刻表的内容，两个查询结果相同还不足以保证这两个查询表达式是等价的；它们必须要求对于所有可能的表的内容的都给出相同的结果。

3. 关系代数优先级

例2.7.4中定义的表达式PAIRS2和PAIRS3，L和M的乘积是在where子句前包含在两个圆括号内的，这是因为表达式(L X M) where C是先做笛卡儿积，然后才执行where子句所表达的选择运算；而表达式L X M where C代表的先建立选择M where C然后与L做笛卡儿积运算，因此L X M where C和L X (M where C)具有相同的含义。实际上，如果我们用第二种方式的话PAIRS2是不确定的，因为L X (M where L.city=M.city and L.cid<M.cid)中有个where子句指向一个不在表M中的列，但它却是选择的对象之一。

L X M where C和L X (M where C)具有相同含义，这是因为关系代数运算具有缺省的优先级。或者说是绑定强度，决定了在一个没有圆括号的表达式中哪一个运算最先被执行。表达

式中的圆括号覆盖了缺省的优先级关系，因此在圆括号中的子表达式总是先被计算。在普通(数值)代数中的一个例子是表达式 $5 \times 3 + 4$ ，很容易就可以计算出这个表达式的值为19。因为我们直觉地辨认出乘号(\times)具有比加号($+$)更高的优先级，写成 $5 \times 3 + 4$ 和写成 $(5 \times 3) + 4$ 是一样的，但是我们可以用圆括号来覆盖掉原来的优先级关系： $5 \times 3 + 4 = 19$ 而 $5 \times (3 + 4) = 35$ 。同样的道理，关系代数运算也具有自己的优先级关系。现在我们已经介绍了大多数的运算，准备把它们结合起来构成更复杂的表达式。在此之前我们需要把优先级的概念表达清楚。

定义2.7.3 关系运算的优先级 关系运算符的优先关系在图2-7中给出，这个表格包括了关系运算连接和除，我们稍后介绍。 ■

优先级	运算	符号
高	投影	$R []$
	选择	$R \text{ where } C$
	乘积	\times
	连接、除法	$\bowtie, +$
	交	\cap
低	并、差	$\cup, -$

图2-7 关系运算优先级

例2.7.5 假设，我们需要找出满足条件的城市，这个城市要求有顾客折扣率低于10%或者有一家佣金百分率不低于6%的代理商。这个查询可以用以下的关系代数表达式表示：

$(\text{CUSTOMERS} \text{ where } \text{discnt} < 10) [\text{city}] \cup (\text{AGENTS} \text{ where } \text{percent} \geq 6) [\text{city}]$

注意到并操作在此处是必需的。我们无法用定义2.7.2中的选择条件中的OR连接符来取代。因为结果中的城市是从完全不同的表中得出，需要有不同的检索条件才能构成最终结果。 ■

4. 连接运算

连接运算的目的在于，通过某个特定同名列上的等值关系把两个给定的表关联起来，构成新表。比如，我们连接表ORDERS和CUSTOMERS，它们具有共同的cid列，结果表将包含ORDERS表中的订单内容以及从CUSTOMERS表中取出该订单的顾客的辅助信息。这里所定义的连接运算也称为等值连接或者自然连接。

定义2.7.4 连接 考虑表R和S，其标题分别为： $\text{Head}(R)=A_1 \cdots A_n B_1 \cdots B_k$ 和 $\text{Head}(S)=B_1 \cdots B_k C_1 \cdots C_m$ 。这里 n, k, m 分别是属性类A, B, C的数目。我们定义 $B_1 \cdots B_k$ 是两个表共享的完全属性子集，该子集在 $k=0$ 时可能为空。同样 $A_1 \cdots A_n$ 是R中不在S里的属性， $C_1 \cdots C_m$ 是S中不在R里的属性，这两个集合也同样可以是空，此时 $n=0$ 或者 $m=0$ 。表R和S的连接是新表 $R \bowtie S$ ，标题 $\text{Head}(R \bowtie S)=A_1 \cdots A_n B_1 \cdots B_k C_1 \cdots C_m$ 。行 t 在表 $R \bowtie S$ 中，当且仅当存在两个行 u 属于R, v 属于S，对于所有的 $1 \leq i \leq k$ 有 $u[B_i]=v[B_i]$ 成立。T中的属性列上的值为：对于所有 $1 \leq i \leq n$ 有 $t[A_i]=u[A_i]$ 成立，对于所有 $1 \leq i \leq k$ 有 $t[B_i]=u[B_i]=v[B_i]$ 成立，对于所有 $1 \leq i \leq m$ 有 $t[C_i]=v[C_i]$ 成立。如果R中的 u 行和S中的 v 行构成了T中的行 t ，我们称这两行是可连接的。 ■

我们称 $B_1 \cdots B_k$ 是表R和表S连接的属性。关系规则之一（我们仅在前面的关系规则2中提到）说明表中的列的次序是无关紧要的。特别地，并不需要 $B_1 \cdots B_k$ 正好按次序出现在表R的末尾表S的开头，它们其实可以出现在两个表中的任何位置。

例2.7.6 考虑如下的两个表R和S:

R			S		
A	B1	B2	B1	B2	C
a1	b1	b1	b1	b1	c1
a1	b2	b1	b1	b1	c2
a2	b1	b2	b1	b2	c3
			b2	b2	c4

我们可以看到表R中的第二行和S中的任何行在列B₁和B₂上都不同步匹配，所以它无法在R \bowtie S中构成新行。另一方面，表R中的第一行可以和S中的两个记录做连接，因此，它在表R \bowtie S中可以生成两行。表R中的第三行同样可以和表S中的一行做连接。我们也注意到了表S中的第四行没有和表R中的任何行是可做连接的。因此在R \bowtie S中没有构成新的记录。新表R \bowtie S如下：

R \bowtie S			
A	B1	B2	C
a1	b1	b1	c1
a1	b1	b1	c2
a2	b1	b2	c3

如果表R和S没有共同的属性做连接运算，那么连接的结果和两个表做笛卡儿积操作的结果相同，这也就是说，如果B₁…B_t为空，即k=0，那么R \bowtie S=R \times S；另一方面，如果R和S具有相同的属性集，那么两个表的连接运算和两个表的交集相同，这就是说，如果A₁…A_n和C₁…C_m都为空集，即n=m=0，那么R \bowtie S=R \cap S。两者的等价关系在本章后面的习题可以验证，但是你可以对它们提出有助于理解连接定义的某种见解。

我们现在来考虑CAP数据库中的一些例子。

例2.7.7 我们要求找出所有订购商品p01的顾客的名字。要明白我们需要两个表中的信息才能回答这个问题，因为订单信息是存放在表ORDERS中，而表ORDERS中并不包含顾客的名字，为了得到这个值(cname)我们需要查询表CUSTOMERS。在关系代数中完成这一查询的一个很自然的方法是在表ORDERS和表CUSTOMERS上进行连接，因此，我们先来看看这个连接的结果。如果定义CUSTORDS:=CUSTOMER \bowtie ORDERS，对应于图2-2的内容。图2-8显示了连接的结果。

表CUSTOMERS和表ORDERS的唯一的共同属性是cid，而cid列是表CUSTOMERS的键，也就是说cid的值在所有的行上是唯一的。同时，表ORDERS中每一行数据上也有一个特殊的cid值列出了订购商品订单的顾客(两个不同行具有相同的cid值当然是允许的)。连接两个表的作用在于扩展表ORDERS中的每一条记录，使得它们包含了表CUSTOMERS中的对应的cid所在的行信息。连接运算显然是一个重要的操作，因为我们需要把表ORDERS中的有关cid信息和表CUSTOMERS中的对应的cid信息相连起来。我们也可以在pid和aid中做同样的事情。通过把表ORDERS和表PRODUCTS或表AGENTS做连接运算，可以扩展表ORDERS中记录的对应的商品或者代理商信息。

CUSTORDS

cname	city	discnt	ordno	month	cid	aid	pid	qty	dollars
TipTop	Duluth	10.00	1011	jan	c001	a01	p01	1000	450.00
TipTop	Duluth	10.00	1012	jan	c001	a01	p01	1000	450.00
TipTop	Duluth	10.00	1019	feb	c001	a02	p02	400	180.00
TipTop	Duluth	10.00	1018	feb	c001	a03	p04	600	540.00
TipTop	Duluth	10.00	1023	mar	c001	a04	p05	500	450.00
TipTop	Duluth	10.00	1022	mar	c001	a05	p06	400	720.00
TipTop	Duluth	10.00	1025	apr	c001	a05	p07	800	720.00
TipTop	Duluth	10.00	1017	feb	c001	a06	p03	600	540.00
Basics	Dallas	12.00	1013	jan	c002	a03	p03	1000	880.00
Basics	Dallas	12.00	1026	may	c002	a05	p03	800	704.00
Allied	Dallas	8.00	1015	jan	c003	a03	p05	1200	1104.00
Allied	Dallas	8.00	1014	jan	c003	a03	p05	1200	1104.00
ACME	Duluth	8.00	1021	feb	c004	a06	p01	1000	460.00
ACME	Kyoto	0.00	1016	jan	c006	a01	p01	1000	500.00
ACME	Kyoto	0.00	1020	feb	c006	a03	p07	600	600.00
ACME	Kyoto	0.00	1024	mar	c006	a06	p01	800	400.00

图2-8 CUSTORDS:=CUSTOMERS \bowtie ORDERS

对于要求找出所有订购商品p01的顾客的名字这个问题的解决方法用关系代数查询表达如下：

```
CNP01 := (CUSTOMERS  $\bowtie$  ORDERS where pid = 'p01') [cname]
```

注意，根据关系代数的优先规则，在ORDERS where pid='p01'两边隐含了一个圆括号，这个表达式给出表ORDERS中与p01产品相关的行，然后与表CUSTOMERS做连接运算扩展表ORDERS中包含p01商品的记录，最后在cname上的投影使得只有cname列出现在结果中，最终结果CNP01如下：

CNP01

cname
TipTop
ACME

■

例2.7.8 现在我们想要查询所有订购了至少一种价值为\$0.50的商品的顾客名字。我们首先列出：

```
CHEAPS := (PRODUCTS where price = 0.50) [pid]
```

这个表达式取出了所有价值为\$0.50的商品的pid，然后我们计算

```
((ORDERS  $\bowtie$  CHEAPS)  $\bowtie$  CUSTOMERS) [cname]
```

从(ORDERS \bowtie CHEAPS)中得出了包含价值\$0.50的商品的订单，然后把这些中间结果的

订单信息和表CUSTOMERS做连接运算可以找到顾客的名字。我们也可以通过把上面两步合并来避免赋值语句：

$$((ORDERS \bowtie (PRODUCTS \text{ where } price = 0.50) [pid]) \bowtie CUSTOMERS) [\text{cname}]$$

很重要的一点是在我们建立CHEAPS表的时候我们必须把价值\$0.50的商品信息投影到pid上。表达式

$$((ORDERS \bowtie PRODUCTS \text{ where } price = 0.50) \bowtie CUSTOMERS) [\text{cname}]$$

可能无法正确解决这个问题，因为在第二次连接操作中有一些无法预计的额外列对于两个表来说是共有的：这个表达式要求商品的city和订购该商品的顾客所在的city也必须相同。这个练习之一就是要求你显示出后面这个查询的结果，并获得原先正确的关系代数表达式查询的结果。这个比较的意义在于告诉我们，在做连接的时候必须使用投影来避免一些连接上的不确定影响。

乘积和连接运算都满足结合率。也就是说，如果R, S, T是三个表，那么我们有：

$$\begin{aligned} (R \times S) \times T &= R \times (S \times T) \\ (R \bowtie S) \bowtie T &= R \bowtie (S \bowtie T). \end{aligned}$$

它们同时还满足交换率。也就是说，对于所有的R和S有 $R \times S = S \times R$ 和 $R \bowtie S = S \bowtie R$ 成立。我们在习题中讨论连接的其他性质。

5. 除运算

除运算是我们介绍的最后一个自然关系运算，考虑两个表R和S，这里S的标题是R的标题的子集。特别地，假定 $\text{Head}(R)=A_1 \cdots A_n B_1 \cdots B_m$ 并且 $\text{Head}(S)=B_1 \cdots B_m$ 。

定义2.7.5 除 表T为 $R \div S$ (读作R除以S)的结果，如果 $\text{Head}(T)=A_1 \cdots A_n$ 并且T中包含的恰好是这样的行t：对于S中每一个行s， $t \bowtie s$ (t和s串接的结果的行可以在表R中找到。(关于t和s串接的含义参见定义2.6.4))。

表S($\text{Head}(S)=B_1 \cdots B_m$)和表T= $R \div S$ ($\text{Head}(T)=A_1 \cdots A_n$)之间没有共同的列，但是因为 $\text{Head}(R)=A_1 \cdots A_n B_1 \cdots B_m$ ，我们可以看到S和T具有不相交的标题，所以有可能 $T \times S = R$ (或者 $T \bowtie S = R$ ，因为如果T和S之间没有共同的属性列的时候，连接和乘积是等价的)。这恰好是我们需要定义的除运算，也就是乘积的逆运算。

定理2.7.6 给定两个表T和S， $\text{Head}(T)=A_1 \cdots A_n$ 且 $\text{Head}(S)=B_1 \cdots B_m$ 。如果表R是通过 $R=T \times S$ 定义的，那么有 $T=R \div S$ 成立。

证 因为 $R=T \times S$ ，所以有 $\text{Head}(R)=A_1 \cdots A_n B_1 \cdots B_m$ 。用W表示 $R \div S$ 的结果表，那么根据定义2.7.5有 $\text{Head}(W)=A_1 \cdots A_n$ ，且W具有和T相同的标题。现在根据乘积的定义，如果元组t存在于T中，那么对于所有的S中的行s， $t \bowtie s$ (t和s串接)存在于表 $R=T \times S$ 中，而在定义2.7.5中这正说明了行t存在于 $W=R \div S$ 中，因此我们得到了 $T \subseteq W$ 。同理可证 $W \subseteq T$ 。所以有 $T=W$ 成立。

不幸的是，有这样的可能性存在：我们从表R和S的内容入手，因此并不是对于所有可能的T都有 $R = T \times S$ 成立。然而T的定义表明了，当 $T=R \div S$ 时表T包含了满足 $T \times S \subseteq R$ 的最大可能的记录集合。这一点和C或者Java中的整数除法 $x=y/z$ 类似，x是满足 $x \times z=y$ 的最大的数字。我们在后面的习题中可以看到更多相关的信息。

例2.7.9 给定表R如下：

R	A	B	C
a1	b1	c1	
a2	b1	c1	
a1	b2	c1	
a1	b2	c2	
a2	b1	c2	
a1	b2	c3	
a1	b2	c4	
a1	b1	c5	

我们列出一个可能的表S和结果T： $= R \div S$ 。

S	T
C	A B
c1	a1 b1
c1	a2 b1
c2	a1 b2

在上面的例子中，注意到所有的行 $S \times T$ 都在R中。在T中没有更大的行的集合也满足这个条件，因为在表R中只有3个行在列C上具有值c1。

S	T
C	A B
c1	a1 b2
c2	a2 b1

再次注意， $S \times T$ 中所有的行都在R中，而且在T中没有更大的行的集合也满足这个条件，我们可以根据R上的列C上的值c2得到这个结论。

S	T
C	A B
c1	a1 b2
c2	
c3	
c4	

$S \times R$ 中的所有行都在R中，根据表R上列C上的值c3和c4，我们可以看出T为什么是具有该性质的最大集合。

S	T
B C	A
b1 c1	a1
	a2

这里我们看到的例子是有三个列的表R被一个具有两个列的表S除，得到的结果表T只有单

-列。R中具有B=b1和C=c1的行只有两个，这两个行在A上的值分别为a1和a2；因此T是最大的。

S	T
B	A
b1	a1
b2	c1

这里我们很容易看出为什么T是本例中最大集合。 ■

接下来我们要给出例子来说明除运算在实际查询中的应用。

例2.7.10 假定我们要求取出由顾客c006订购的商品的号码。这个结果存放在表PC6中。

PC6
pid
p01
p07

我们可以用以下的查询得到该结果：

```
PC6 := (ORDERS where cid = 'c006')[pid]
```

我们更感兴趣的是找出所有的订购了所有这些商品的顾客。通过CP := ORDERS[cid, pid]，我们可以从表ORDERS中提取出顾客号码以及他们所订购的商品。对于我们需要的查询，要求找出所有订购了PC6中全部商品的顾客(用cid值唯一标识)，可以应用除法运算得到答案。结果表CP ÷ PC6如下：

CP + PC6
cid
c001
c006

从本例中可以了解到，当检索要求中包含“所有”字样的时候，我们就可能需要使用除运算。 ■

在例2.7.10中在被PC6除之前把ORDERS投影到列cid和pid上而不是在除运算以后再投影到pid上是完全必要的。否则，我们可能被要求表ORDERS上额外的列值在和PC6上的两个pid值做串接的时候保持不变。为了说明这一点，我们来看下面两个表R和S。

R	S		
A	B	C	C
a1	b1	c1	c1
a2	b1	c1	c2
a3	b2	c1	
a1	b1	c2	
a2	b2	c2	
a3	b3	c2	

根据除法的定义，R ÷ S的标题含列A和B。具有唯一行(a1, b1)。因此(R ÷ S)[A]具有唯

一行a1。另一方面，如果我们想把表R投影到属性列A和C上， $R[A, C]$ 具有六个不同的行，表 $R[A, C] \div S$ 具有唯一的列A，三行值分别为a1, a2和a3。你应该经常问自己作除运算的表中哪些列，和被除数的所有行串接的时候应该是保持不变的。

例2.7.11 我们可以用下面的式子来回答取出所有订购了全部商品的顾客的名字这一检索要求：

$$((ORDERS[cid, pid] + PRODUCTS[pid]) \bowtie CUSTOMERS[cname]).$$

再强调一下，先把表ORDERS投影到属性[cid, pid]上，以便结果表中只有一个属性cid是很必要的。只有一个顾客c001满足上述查询要求，订购了所有的商品，c001就正好是结果表的内容。 ■

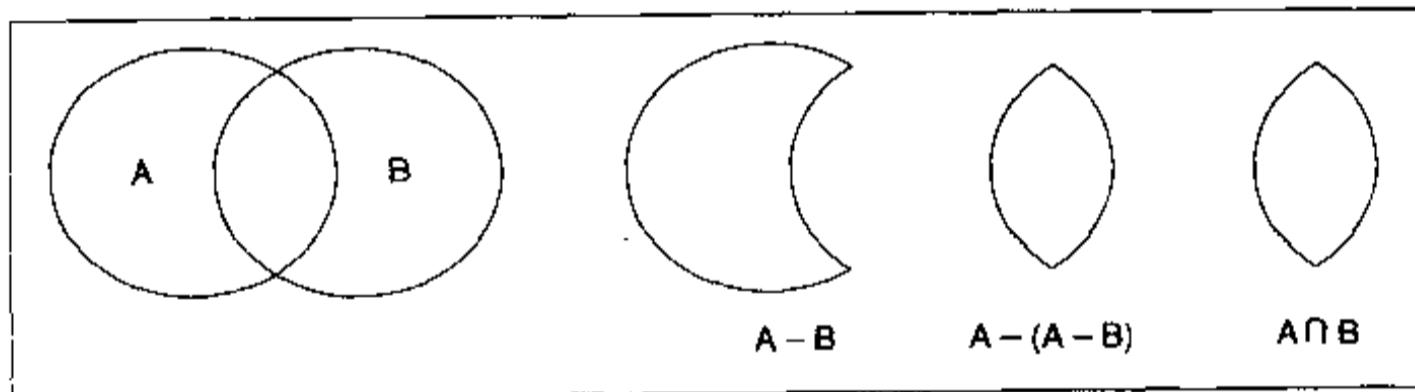
2.8 运算依赖

2.7节定义的某些关系运算只是为了应用上的方便才定义的。在某种意义上说，关系代数的全部功能可以由较小的运算子集来完成。我们称基本运算的最小集合由并、差、乘积、选择和投影组成，加上允许重定义属性名字的赋值算子。为了验证这个命题，我们需要说明余下的运算——交、连接和除——可以如何用基本关系运算来表示。交和连接运算很容易说明。

定理2.8.1 R和S是两个兼容表， $\text{Head}(R)=\text{Head}(S)=A_1 \cdots A_n$ 。交运算可以被减运算重定义为： $A \cap B = A - (A - B)$ 。

证 我们用文氏图说明：

这个说明可以很容易用自然语言表达，如果需要的话。



定理2.8.2 给定两个表R和S， $\text{Head}(R)=A_1 \cdots A_n, B_1 \cdots B_k$, $\text{Head}(S)=B_1 \cdots B_k, C_1 \cdots C_m$ ，其中 $n, k, m \geq 0$ ，R和S的连接可以用积、选择、投影运算以及赋值算子重写。

证 为了证明这一点，考虑表

$$T := (R \times S) \text{ where } R.B_1 = S.B_1 \text{ and } \cdots \text{ and } R.B_k = S.B_k$$

表T由积和选择操作构成，这和R和S的连接类似。现在我们只要简单地用投影运算把不需要的重复列剔除以及利用赋值算子重定义列名。

$$T_1 : T[R.A_1, \dots, R.A_n, R.B_1, \dots, R.B_k, S.C_1, \dots, S.C_m]$$

$$T_2(A_1, \dots, A_n, B_1, \dots, B_k, C_1, \dots, C_m) := T_1$$

表T2结果和 $R \bowtie S$ 相同。 ■

要说明除运算可以用基本关系运算代替并不容易掌握，证明是带星号的，这就意味着你

无法从中对本教材加深多少了解。这个证明是为那些具有非常出色的数学背景的好奇的读者提出的。

定理2.8.3* [难] 除法可以用投影、乘积和差表达。

证 考虑两个表R和S, $\text{Head}(R)=A_1 \cdots A_n B_1 \cdots B_m$ 且 $\text{Head}(S)=B_1 \cdots B_m$ 。我们可以证明:

$$R \div S = R[A_1, \dots, A_n] - ((R[A_1, \dots, A_n] \times S) - R)[A_1, \dots, A_n]。$$

假设 u 是表 $R[A_1, \dots, A_n] - ((R[A_1, \dots, A_n] \times S) - R)[A_1, \dots, A_n]$ 的一个元组。这意味着 u 在表 $R[A_1, \dots, A_n]$ 中, 且 u 不在表 $((R[A_1, \dots, A_n] \times S) - R)[A_1, \dots, A_n]$ 中。再假设 S 中存在元组 s , 使得 u 和 s 的串接不在 R 中。由于 u 在表 $R[A_1, \dots, A_n]$ 中, 这就说明了 u 是表 $((R[A_1, \dots, A_n] \times S) - R)[A_1, \dots, A_n]$ 的成员, 这就与假设相矛盾。所以对所有的 S 中的元组 s , u 和 s 的连接都在 R 中, 这意味着 u 属于表 $R \div S$ 。

反过来, 如果 u 是 $R \div S$ 中的一个元组, 显然 u 在表 $R[A_1, \dots, A_n]$ 中。另一方面, u 不在表 $((R[A_1, \dots, A_n] \times S) - R)[A_1, \dots, A_n]$ 中, 因此在 S 中存在一个元组 s 使得 u 和 s 的串接在 $R[A_1, \dots, A_n] \times S$ 中而不在 R 中。因此 u 属于表 $R[A_1, \dots, A_n] - ((R[A_1, \dots, A_n] \times S) - R)[A_1, \dots, A_n]$, 这就推出了我们的命题。你也可以从特定的表中用刚给出的除运算示例中推出结果。

很自然, 有人会怀疑上面说的基本关系运算集(并、差、乘积、选择、投影)是否就是最小的集合。换句话说, 我们是否可以找到一个更小的运算集合(在上述五种运算中), 使得这些运算可以用来表达所有其余的运算?事实上这五个关系运算确实构成了最小集合, 习惯数学推导的读者可能会考虑怎么来证明这一点。

关系查询语言(如关系代数或者SQL)的表达能力就在于如何在一系列现存的表上为了回答某个查询而建立新表。在E·F·Codd 1972年写的论文中, 他把关系代数和基于符号逻辑的元组演算做了比较并证明两者在能力上是等价的。这就成为测试关系语言的标准, 即必须具有和关系代数相当的表达能力。通过该测试的查询语言称为完备的, 或者是关系完备的, 并且所有的流行的查询语言(如SQL)都具有该能力。某些学派认为关系代数的表达能力是不够的, 我们将在本章的后续部分进行讨论。

2.9 综合例子

关系代数强大的功能给了我们数据查询的方法, 我们可以从任意数量的表中取出任意列的集合(通过投影), 这里不同的表上的行被串接成一个单一的行(乘积), 然后根据不同列上的条件做限定(选择)。可以不考虑并和差运算, 这两个运算增加了技术的能力, 这个粗略的表述在我们考虑匹配用英语表达的查询的时候给了我们许多能力。本节将讲述这些能力。

本节我们要推导出一个不用别名建立的中间结果表的最终查询表达式。一般地说, 用单一的自含的表达式来表达一个最终查询结果效果更好;我们在后面练习中, 对于不一定需要别名的练习, 要求用单一的表达式解答。注意, 下面的一组别名定义仅仅是对CAP表的缩写。

C := CUSTOMERS, A := AGENTS, P := PRODUCTS, O := ORDERS.

例2.9.1 我们要求查询所有订购了至少一个价值为\$0.50的商品的顾客的名字。对于这样的查询通常的做法是从里到外来构造查询语句。首先, 从价值\$0.50商品入手。

P where price = 0.50

我们怎么样才可以找到订购该商品的顾客呢? 和ORDERS表做连接!

```
(P where price = 0.50) ⚡ O
```

这个结果给了我们表ORDERS包含适合的商品的一个子集，并且扩展了商品的信息。不幸的是，我们还无法从中得到顾客的名字，只有订购这些商品的顾客的cid值。自然的想法是把这个结果和CUSTOMERS表做连接运算，但是我们首先必须剔除掉这个结果中的city属性。最好的方法是退回来开始选择PRODUCTS的时候就对pid做投影，然后我们把结果和CUSTOMERS表做连接，最后投影到最终要求的cname列上。最终答案如下：

```
((P where price = 0.50)[pid] ⚡ O) ⚡ C[cname]
```

注意，表达式(*P* where *price* = 0.50)在对[pid]做投影之前被放入圆括号中去了，否则，根据图2-7中的优先级表，投影运算会先于选择运算执行，这样对*price*的选择条件是毫无意义的。在和O做连接以后，结果表达式(*P* where *price* = 0.50)[pid] ⚡ O也再次被放入圆括号中，这是因为连接是个二元运算，我们并没有定义X ⚡ Y ⚡ Z，因此我们必须写成(X ⚡ Y) ⚡ Z。有一道习题要求你说明连接运算符合结合律:(X ⚡ Y) ⚡ Z = X ⚡ (Y ⚡ Z)，因此我们可以用表达式X ⚡ Y ⚡ Z来代替其中的任何一个。最后，表达式((*P* where *price*=0.50)[pid] ⚡ O) ⚡ C在投影到 cname之前也被圆括号包括；否则，表达式((*P* where *price*=0.50)[pid] ⚡ O) ⚡ C[cname]会在做连接运算之前把C投影到cname上(参见图2-7)，这样我们就会损失连接列cid。 ■

例2.9.2 为了找出全部没有在代理商a03处订购商品的顾客的cid值，我们可以从a03处订购了商品的顾客入手：

```
(ORDERS where aid = 'a03') [cid]
```

那么答案显然可以用

```
ORDERS[cid] - (ORDERS where aid = 'a03') [cid]
```

或者

```
CUSTOMERS[cid] - (ORDERS where aid = 'a03') [cid],
```

来表示。

根据我们最初所需要的，第一个查询给出的是至少在别处订购了一个商品的顾客的cid值；而在第二个查询中把那些没有订购任何商品的顾客的cid也查出来了。我们要问自己是否需要把这些“无效”的顾客放入我们的结果表中呢？在缺乏足够的提示的情况下(如果查询要求是另外一个人做出，我们无法征求他的意见)，后者可能比较合适，因为它包含了更多的信息。 ■

例2.9.3 现在我们来考虑那些只在代理商a03处订购商品的顾客。这个查询可以从曾经从某一代理(非a03)订购商品的顾客入手，然后从通过a03订购商品的顾客列表中剔除上述顾客：

```
ORDERS[cid] - (ORDERS where aid <> 'a03')[cid].
```

例2.9.4 现在我们需要找出没有被任何一个在New York的顾客通过在Boston的代理商订购的所有商品。我们很容易找出不符合这个条件的商品，即在New York的顾客通过在Boston的代理商订购的商品：

```
((C where city = 'New York')[cid] ⚡ ORDERS)
  ⚡ A where city = 'Boston' [pid]
```

和前面一样，C选择在和A选择做连接之前被投影到了cid列上，所以连接不要求在

CUSTOMERS和AGENTS两个表中的city值匹配。现在我们需要做的就是从所有商品列表中剔除这些商品：

```
PRODUCTS[pid] ~
(((C where city = 'New York')[cid] ⋈ ORDERS)
 ⋈ A where city = 'Boston') [pid]
```

强调一次，我们需要判断是从PRODUCTS[pid]入手，还是从ORDERS[pid]入手。前者可能包括了没有被订购的“不流动”的商品，而后者列出了那些至少被订购一次的商品。和例2.9.2一样，我们选择给出更多信息的答案，而正确的现实做法是考虑用户的实际需要。■

例2.9.5 取出订购了所有那些价格为\$0.50的商品的顾客的名字。要得到这些顾客的cid值，根据线索“谁订购了所有的商品”我们可以考虑用除运算来得到这些顾客的cid值：

```
O[cid, pid] + (P where price = 0.50)[pid]
```

要得到这些顾客的名字，我们还需要把结果和表CUSTOMERS做连接然后投影到cname上。

```
((O[cid, pid] + (P where price = 0.50)[pid]) ⋈ C)[cname]
```

■

例2.9.6 取出订购了所有被任何人订购过一次的商品的顾客的cid值。再一次用到了除运算。除数，即“所有”商品的列表，需要从ORDERS表而不是PRODUCTS表投影而来。

```
O[cid, pid] + O[pid]
```

如果在除数中用P[pid]代替的话，结果可能包括一些最近被列入商品列表中，而还没有被任何顾客订购的商品，那么结果表中的cid集合就会是空的。■

例2.9.7 取出所有接到至少顾客c004订购的商品集合（可能会更多）的订单的代理的aid值。这个看起来很难的查询要求，可能只需重新表示。试用这样一个要求：取出接到所有c004订购的商品的订单的代理的aid值。

```
O[aid, pid] + (O where cid = 'c004')[pid]
```

■

例2.9.8 取出订购了p01和p07这两种商品的顾客的cid值。要注意下面的关系代数表达式没有正确地回答这个要求。

```
O where pid = 'p01' and pid = 'p07' **错误结果**
```

问题在于ORDERS表中的每一行在pid上只有一个值，而这个值要么等于p01要么等于p07，但不同时等于两者。正确的答案如下：

```
(O where pid = 'p01')[cid] ∩ (O where pid = 'p07')[cid]
```

使用交运算来取出同时订购了p01和p07的顾客。■

例2.9.9 取出从至少一个接受订购商品p03订单的代理商处订购过商品的顾客的cid值。如图2-9，要求取出的顾客在最右边的圆中表示，它只和中间圆所表示的代理相关。特别要注意的是，要求检索的cid本身没有订购商品p03。在图2-2的ORDERS表中，我们可以看到第四行中的c001通过代理商a06订购了商品p03；同样在倒数第四行中，顾客c004通过代理商a06订购了商品p01，因此c004应该被包含在我们的结果表中，因为它从至少一个接受订购商品p03订单的代理商(a06)处订购了商品。然而，我们可以检查到c004没有订购商品p03。

对于这个查询的正确的做法是从里到外；从图2-9的中间的圆圈开始，我们检索出接受了商品p03订单的代理：

```
(0 where pid = 'p03')[aid]
```

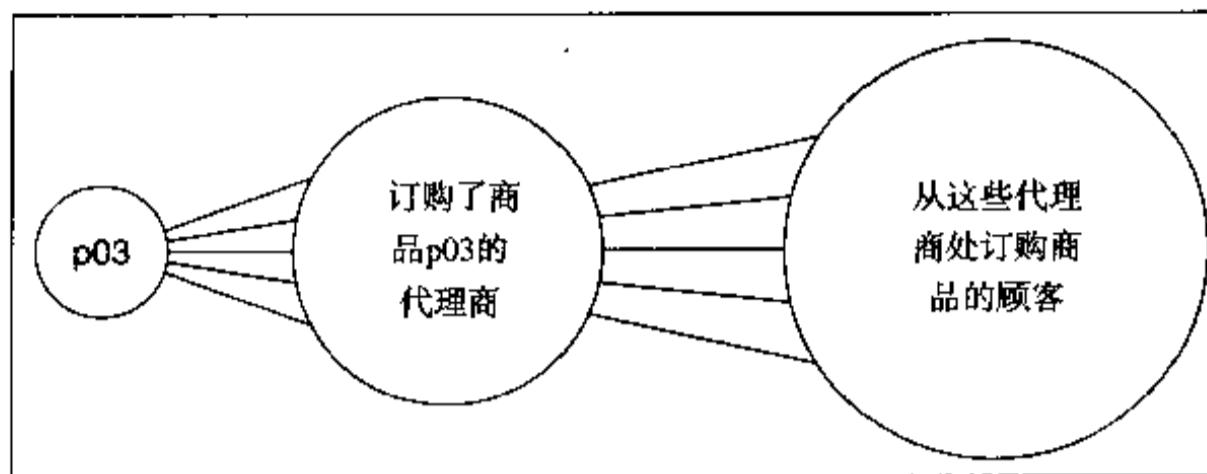


图2-9 例2.9.9中的对象集合和它们之间的连接关系

我们所需要的结果如图2-9中最右的圆圈表示，需要检索出从这些代理商处订购商品的顾客的cid值。这表明需要再次使用到表ORDERS，为了安全起见，我们使用一个不同的别名，X := ORDERS，这样在表达式中的列名就不会混淆，即使两次使用同一个表：

```
(X ⋈ (0 where pid = 'p03')[aid])[cid]
```

一般来说，如果表达式中没有乘积运算，而对同一个表使用不同的别名有点矫枉过正，但是我们这里使用了这种方式使得问题的解决得到了简化。 ■

例2.9.10 取出所有具有和在Dallas或者Boston的顾客相同的折扣率的顾客的cid值。这个例子说明：如果查询中包含了“所有”这个词不一定要使用除运算。实际上这个查询中的所有一词具有误导作用。查询要求可以重新表达为如下我们所熟悉的查询要求：取出那些具有和在Dallas或者Boston的顾客相同折扣率的顾客的cid值。关键在于当我们要求取出满足相同条件的顾客的cid值的时候，自然需要检索出全部符合要求的顾客；但是这并没有要求必须使用除运算。要回答这个查询，我们先找出在Dallas或者Boston的顾客的所有折扣率。

```
(C where city = 'Dallas' or city = 'Boston')[discnt]
```

下一步，要找出具有上面的折扣率的顾客的cid值，我们使用连接运算。首先我们建立一个别名，D := CUSTOMERS，然后给出表达式：

```
((C where city = 'Dallas' or city = 'Boston')[discnt] ⋈ 0)[cid]
```

这些从表D和所有的折扣率集合相连接所得的cid值就是我们所需要的答案。 ■

例2.9.11 现在我们有一个非常复杂的查询要求。我们需要列出通过符合下列条件的代理商订购的商品的pid列表。这些代理商从可能不同的顾客处接受订单，而这些顾客从一个接受过顾客c001订单的代理商处订购了至少一个商品。要建立这样的一个查询表达式的秘诀就是安下心来从里到外解决问题。

我们先直接给出接受过顾客c001订单的代理商：

```
(0 where cid = 'c001')[aid]
```

接下来，从这些代理商处订购过商品的顾客可以用下面的式子表示：

```
(X ⋈ (0 where cid = 'c001')[aid])[cid]
```

这里 $X := ORDERS$ 。再给定 $Y := ORDERS$ ，从这些顾客处接受过订单的代理商可以通过下面表达式列出来：

$$(Y \bowtie (X \bowtie (O \text{ where } cid = 'c001')[aid])[cid])[aid]$$

最后令 $Z := ORDERS$ ，通过上述代理商订购的商品的 pid 是：

$$(Z \bowtie (Y \bowtie (X \bowtie (O \text{ where } cid = 'c001')[aid])[cid])[aid])[pid]$$

这就是我们最初所需要的结果。 ■

例2.9.12 取出没有被生活在以D开头的城市的顾客所订购的商品的 pid 值。怎么找出生活在以D开头的城市的顾客呢？回想字符串在字母表中的顺序不等式，下面的表达式给出了答案。

$$C \text{ where } city >= 'D' \text{ and } city < 'E'$$

而没有被这些顾客所订购的商品可以从所有的商品中减去这些顾客所订购的商品而得到：

$$P[pid] - (O \bowtie (C \text{ where } city >= 'D' \text{ and } city < 'E'))[pid]$$

■

2.10 其他关系运算

到目前为止我们讨论的关系运算基础集合是为了用一个较小的集合来表达关系代数中对查询要求的处理。在理论上可以用基本运算集合来表示所有的查询。然而，一些附加的运算可以使得表达起来更为简单，比如我们可以用一系列的基本运算来表示除运算。

下面我们要讨论两个附加的很有用的关系运算，外连接和θ连接。这些运算的作用显示在下面的表中。

名称	符号	键盘格式	示例
外连接	\bowtie_0	OUTERJ	$R \bowtie_0 S$ 或 $R \text{ OUTERJ } S$
左外连接	\bowtie_{L0}	LOUTERJ	$R \bowtie_{L0} S$ 或 $R \text{ LOUTERJ } S$
右外连接	\bowtie_{R0}	ROUTERJ	$R \bowtie_{R0} S$ 或 $R \text{ ROUTERJ } S$
θ连接	$\bowtie_{A_i \theta B_j}$	$JN(A_i \theta B_j)$	$R \bowtie_{A_1 \theta B_2} S$ $R JN(A_2 \theta B_2) S$

外连接在商业数据库中对于简化查询具有特殊作用。对于这样的连接运算在原先的许多查询语言中都被遗漏了，而现在又全补上。这两种运算都可以用五种基本运算来表达，但是对外连接的表达在实际上并不容易使用。

1. 外连接

介绍外连接我们需要用到两个表：表AGENTS和一个新表SALES。前者我们已经接触过了，后者包含两列：出现在表ORDERS中的代理商的 aid 和一个total列给出了每一个代理商的营业额。如果代理商没有收到任何订单，我们就假定该代理商的 aid 值不出现在SALES表中。现在我们需要打印一张提供了所有的代理商名字、他们的 aid 以及营业额的表格(作为销售经理的报表)。销售经理熟悉所有的代理商名字，用 aid 仅仅是为了区分相同的代理商名字。你可能会认为这只要简单地写出关系代数表达式 $(AGENTS \bowtie SALES)[\text{aname}, aid, total]$ 。但是这个结果遗漏了某些代理商的名字：如果这个代理商没有收到任何订单，那么根据假定，他的 cid 将不会出现在SALES表中，当然在结果的报表中也不会出现。(SALES表中没有 aid 值可以和AGENTS表中的 aid 做连接)。销售经理当然不希望有代理商的名字遗漏在结果报表

外，即使他没有收到任何订单。实际上，这可能需要人工干涉。我们也可能遇到这样的情况：一个新的名叫Beowulf的代理商，aid为a07。在这个信息没有更新到AGENTS表之前，他收到了订单，然后在SALES表中插入记录，那么我们再一次遇到这样的问题——表达式 $(AGENTS \bowtie SALES) [aname, aid, total]$ 无法给出销售经理满意的答案。因为SALES中的aid列无法和AGENTS中的aid相匹配。外连接运算，用 \bowtie 表示，可以帮助我们解决这个问题。我们给出的答案是：

hired and has taken an order that shows up in the SALES table before any information on

这样就可以得到了两个表中的所有值了。在代理商没有收到订单的情况下，我们检查到SALES表中没有一个营业额的值，那么在结果表中对应的营业额的值被填上了null。同样对于收到了订单而在AGENTS表中没有名字的情况也是如此处理。我们可以看到如下的结果：

OJRESULT		
aname	aid	total
Smith, J	a01	850.00
Jones	a02	400.00
Brown	a03	3900.00
Gray	a04	null
Otasi	a05	2400.00
Smith, P.	a06	900.00
null	a07	650.00

所有的没有匹配的列的值都显示在了外连接的结果中了。代理商Gray, a04, 没有收到任何订单，因此他的total列上的值是null。代理商Beowulf, a07, 在aname列上的值为null。

为了定义外连接运算，考虑表R和S， $Head(R)=A_1 \cdots A_n B_1 \cdots B_k$ 且 $Head(S)=B_1 \cdots B_k C_1 \cdots C_m$ ，这里 $n, k, m \geq 0$ 。

定义2.10.1 外连接 表R和S的外连接 $R \bowtie_0 S$ ，其标题 $Head(R \bowtie_0 S)=A_1 \cdots A_n B_1 \cdots B_k C_1 \cdots C_m$ 。行 t 属于表 $R \bowtie_0 S$ ，如果下列情况之一发生：

1) 两个可连接的行 u, v 分别在R和S中，对于所有的 $i, 0 \leq i \leq k$ ，有 $u[B_i]=v[B_i]$ 成立。在这种情况下我们通过以下方式定义 t ：对于所有的 $1 \leq l \leq n$ ， $t[A_l]=u[A_l]$ ；对于所有 $1 \leq i \leq k$ ， $t[B_i]=u[B_i]$ ，对于所有 $1 \leq j \leq m$ ， $t[C_j]=v[C_j]$ 。

2) 表R中的一个行 u 使得S中没有一个可以于之连接的行 v ，在这种情况下， $t[D]=u[D]$ 对于所有的标题 $Head(R)$ 中的D。而对于所有的在 $\{C_1, \dots, C_m\}$ 中的D， $t[D]$ 上的值为null。

3) 表S中的一个行 v ，而R中没有一个可以于之连接的行 u ，在这种情况下，对于所有的标题 $Head(S)$ 中的D。 $t[D]=v[D]$ 。而对于所有的在 $\{A_1, \dots, A_n\}$ 中的D， $t[D]$ 上的值为null。 ■

我们说外连接保留了未匹配的行，也就是说在外连接一端的表上的行，即使在另一端上的表中没有与之相匹配的连接列值也会出现在外连接的结果中。而左外连接和右外连接运算只是因为我们需要在某一边上保留未匹配的行而已。左外连接保留了在操作符左边的未匹配行，而右外连接正好相反，保留在右边出现的外连接。比如说，我们有个表SPECIAL_AGENTS和AGENTS兼容，只不过包含的是特殊的代理商的行(如a01, a04, a06等)。现在对这个表我们想看到和上面OJRESULT同样的有关代理的报表。如果我们使用表达式：

$(SPECIAL_AGENTS \bowtie SALES)[aname, aid, total]$.

我们看不到有关代理商a04的为null营业额值。而如果我们使用表达式：

$(SPECIAL_AGENTS \bowtie_0 SALES)[aname, aid, total]$.

我们可以看到结果中包含了所有的代理商，不仅仅是a01, a04和a06。而在SPECIAL_AGENTS表上没有的aname值的代理商的营业额为null。

OJRESULT2

aname	aid	total
Smith, J.	a01	850.00
null	a02	400.00
null	a03	3900.00
Gray	a04	null
null	a05	2400.00
Smith, P.	a06	900.00
null	a07	650.00

这是用SPECIAL_AGENTS代替AGENTS表做外连接所得的结果。但是我们知道的仅仅是在SPECIAL_AGENTS表中存在的代理的报表。一个公司有数千个代理商，得到所有的并不需要的额外信息是一个很严重的问题。为了正确得到我们想要的，我们可以用左外连接运算，表达式为：

$(SPECIAL_AGENTS \bowtie_{L0} SALES)[aname, aid, total]$.

结果如下：

LOResult

aname	aid	total
Smith, J.	a01	850.00
Gray	a04	null
Smith, P.	a06	900.00

这就是左外连接表的结果，显示的正是我们所需要的信息。作为右外连接的例子我们可以通过如下表达式得到同样的结果：

$(SALES \bowtie_{R0} SPECIAL_AGENTS)[aname, aid, total]$

我们把这个作为练习，要求你说明怎么修改定义2.10.1(外连接运算)来给左外连接和右外连接下一个正确的定义。

2. θ连接

当连接两个表而表中的同名属性并不是需要等值的时候，我们可以使用θ连接。符号θ代表在θ连接中的不同的表中的相关属性之间的比较运算符，它可以是>,<,>=,<=,=,<>。

定义2.10.2 θ连接 R和S是两个表， $\text{Head}(R)=A_1 \cdots A_n$ 且 $\text{Head}(S)=B_1 \cdots B_m$ 。我们允许 $\text{Head}(R)$ 中的任何属性 A_i 可以和 $\text{Head}(S)$ 中的任何属性 B_j 同名，但是不需要指定任何属性是共用的。假设属性 A_i 和 B_j 具有相同的域，而关系运算符θ是集合{>,<,>=,<=,=,<>}中的一个。如果 A_i 和 B_j 具有不同的名字，那么R和S的θ连接是表T := $R \bowtie A_i \theta B_j S$ ，这里 $\text{Head}(R \bowtie A_i \theta B_j S)=A_1 \cdots A_n B_1 \cdots B_m$ 。

否则，如果A_i和B_j具有相同的名字，我们必须限定表R和S的θ连接，T := R ⋈ R.A_iθS.B_j S。下面我们将假定使用未受限的名字。

新表的行具有格式t=(a₁, …, a_n, b₁, …, b_m)，这里，A_ib_j，(a₁, …, a_n)在R中，(b₁, …, b_m)在S中，很容易看出θ连接可以用基本关系代数操作符表达。 ■

例2.10.1 找出所有商品数目超过了当前手头上拥有的商品的数目的订单的ordno值。这可以用θ连接解决：

```
(ORDERS ⋈ ORDERS.qty > PRODUCTS.quantity PRODUCTS)[ordno]
```

这等价于关系代数表达式：

```
((ORDERS × PRODUCTS) where ORDERS.qty > PRODUCTS.quantity)[ordno] ■
```

条件 ORDERS.qty > PRODUCTS.quantity 称为θ条件。更普遍地称本例中的θ连接为大于连接。而普通的连接运算符称为等值连接，或者为自然连接。

推荐读物

关系模型还有许多内容在本章中没有涉及到。David Maier给出了相当先进并且优秀的理论^[2]。而E·F·Codd的关系代数的基础文献可以在推荐读物[1]中的1.2小节中找到。这些读物中的其他文献也是很有用的。

- [1] E.F. Codd." A Relational Model of Data for Large Shared Data Banks." In *Readings in Database Systems*, 3rd ed. Michael Stonebraker and Joseph M. Hellerstein, editors. San Francisco: Morgan Kaufmann, 1998.
- [2] David Maier. *The Theory of Relational Databases*. New York: Computer Science Press, 1983.

习题

下面部分习题在本书后面的“习题解答”中给出了答案，这些习题用•标出。

2.1 假设本题所用的表与例2.4.1中的表相似，通过观察各行的内容，我们可以仅根据表的内容来判断设计者考虑该表键的意图。

T1			
A	B	C	D
a1	b1	c1	d1
a2	b3	c1	d2
a3	b4	c2	d2
a4	b2	c2	d1

- (a)• 找出上面的表T1的三个候选键，其中一个候选键包含两个列。
- (b) 找出下面的表T2的两个候选键。

T2				
A	B	C	D	E
a1	b1	c1	d1	e1
a2	b1	c1	d1	e2
a3	b1	c2	d1	e1
a4	b2	c1	d1	e1

(c)• 给出像习题(a)那样的一个表的例子，要求有四列且只有四行，但是只有一个候选键由前三个列组成。要注意，前三列，不能两两唯一标识所有的行。

(d) 给出像习题(b)那样的一个表的例子，要求有五列A、B、C、D、E以及四行，但是只有一个候选键由前四个列组成。并要求说明为什么该表中没有其他的列的集合构成键。

2.2 考虑一个电话公司的数据库系统，该数据库中包含了表SUBSCRIBERS，其标题是：

name ssn address city zip information-no

假设information-no是提供给用户的唯一的10位的电话号码，包括区号。尽管一个用户可能有多个电话号码，但是其他的备选号码存放在一个单独的表中；当前的表中对于每一个用户都有唯一的行(丈夫和妻子可能同时申请，可能会在表中占据两行，但是具有唯一的信息码information-no)。DBA已经建立了该表的以下规则来反映设计者对数据的意图：

- ◆ 没有两个用户(在不同的行上)具有相同的社会保障号。
- ◆ 两个不同的用户可以共享相同的信息号码(比如丈夫和妻子的例子)，他们在SUBSCRIBERS表中占据不同的两行。然而，同名用户不能共享同一个地址、城市、邮编而且不能共享同一个信息号码。比如，Diane Isaacs和David Isaacs住在一起，具有相同的信息码，如果他们想要在数据库中都以D.Issacs存放的话，需要区分他们的名字，也就是说，至少要求他们中的某一个以全名存储。

(a)• 根据上面的假定，指出表SUBSCRIBERS的所有的候选键。这样的候选键有两个，其中一个包含information-no属性，另一个包含zip属性。

(b) 你会选择哪一个候选键作为表的主键？为什么？

2.3 在2.2节中的“表和关系”部分，我们定义了笛卡儿积： $CP = CID \times CNAME \times CITY \times DISCNT$ 。在例2.2.1之前的段落中，我们指出没有必要在表CUSTOMERS中包含所有的笛卡儿积的行，因为如果CNAME、CITY和DISCNT的域中任何一个包含了多个值，那么在笛卡儿积中将至少有两行具有相同的CID值。举例说明这一点，并阐述只要CP中存在这些行，那么这个结论总是正确的。

2.4 用关系代数来解决有关CAP数据库的查询。这些习题是很基本的，是后面系统的基础。

(a)• 找出订单总价大于或者等于\$1000的(ordno, pid)对。

(b) 找出所有价格在\$0.50和\$1.00之间的商品的名字，包括边界价格。

(c)• 找出订单价格低于\$500的(ordno, cname)对。使用一次连接。

(d) 找出所有三月份接受的订单的(ordno, aname)对。使用一次连接。

(e)• 找出所有三月份接受的订单的(ordno, cname, aname)三元组。使用两次连接。

(f) 找出所有位于New York的代理商，并且要求这些代理商所接受的单个订单价格少于\$500。

(g)• 找出所有三月份订购的Duluth的商品的名字。

2.5 用关系代数来解决有关CAP数据库的查询。(这些查询要求在下一章中用于阐述基于计算机的查询语言SQL的功能。) 注意，对于下面的所有查询，需要用单个的自含式关系代数表达式来解决，若无必要不能依赖任何通过使用别名得到的中间结果。

- (a) • 找出所有顾客、代理商和商品都在同一个城市的三元组(cid, aid, pid)。本题不涉及订单信息。
- (b) 找出所有顾客、代理商和商品不都在同一个城市(可能有两个在同一城市)的三元组(cid, aid, pid)。
- (c) • 找出所有顾客、代理商和商品两两不在同一个城市的三元组(cid, aid, pid)。
- (d) 取出接受顾客c002订单的代理商所在的城市。
- (e) • 取出至少被一个在Dallas的顾客通过位于Tokyo的代理商订购的商品的名字。
- (f) 取出曾经收到Kyoto的顾客订单的代理商所销售的所有商品的pid值。注意，本题和要求取出所有曾经被Kyoto的顾客订购的商品不同。
- (g) • 列出所有在同一个城市的代理商的aid对。
- (h) 找出没有通过代理商a03订购过商品的顾客的cid值。
- (i) • 找出折扣率最大和最小的顾客的cid值。注意，用关系代数提供的运算来表示本题比较困难。
- (j) 找出订购了所有商品的顾客的cid值。
- (k) • 找出通过代理商a03而不通过代理a06订购的商品的pid值。
- (l) 取出商品的pname和pid值，要求这些商品所在的城市和某个销售过该商品的代理商所在的城市相同。
- (m) • 取出名字是以字母N开头的代理商的aid和aname值，并且这些代理没有销售过任何Newark生产的商品。
- (n) 取出同时订购了商品p01和p07的顾客的cid值。
- (o) • 取出销售过所有曾被顾客c002订购过的商品的代理商的名字。
- (p) 取出销售过所有曾经被某些顾客订购过的商品的代理商的名字。
- (q) • 取出所有的三元组(cid, aid, pid)，要求对应的顾客，代理商和商品中至少有两者是位于同一座城市。(本题的要求和习题(a), (b), (c)相同么？)
- (r) 取出所有曾在代理商a03处订购商品的顾客订购过的商品的pid值。
- (s) • 取出接受过Kyoto的顾客一笔总额超过\$500的订单的代理商的aid值。
- (t) 给出所有的(cname, cname)对，要求对应的顾客曾经在对应的代理商处订购过商品。
- (u) • [难]取出只从一家代理商处订购过商品的顾客的cid值。

2.6 (a) 给定如例2.6.1中的表R和S，要求列出表RJOINS的内容。

- (b) • 证明：如果 $\text{Head}(R) \cap \text{Head}(S) = \emptyset$ (没有共同的列属性)，那么表 $R \times S$ 等于表 $R \bowtie S$ 。在这里，如果说两个表相等指的是两个表具有相同的标题和内容，表中行的次序不改变表的内容。如果你无法从数学上证明该命题，请阐述为什么这里表 $R \times S$ 等于表 $R \bowtie S$ 。

2.7• R和S是没有共同属性的两个关系，同习题2.6请说明 $R \bowtie S \div S = R$ 。

2.8 根据定理2.8.2，表达式 $R \bowtie S$ 可以用乘积、选择和投影运算来表达。

- (a) • 请说明外连接也可以用其他的运算代替。你可以使用一般连接、并、乘积、选择和投影。
- (b) 请说明θ连接也可以用乘积、选择和投影表示。

2.9 解释为什么对于所有的表R、S和T，下面的等式总成立。

$$(a) \bullet (R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

$$(b) R \bowtie S = S \bowtie R$$

2.10 证明： $R \bowtie S = R \cap S$ ，当且仅当R和S是兼容表。

注意：下面的习题需要某些数学方面的训练。

2.11 • 说明对于表R上的任何条件C1和C2，下面的查询是等效的：

$$(R \text{ where } C1) \text{ where } C2$$

$$(R \text{ where } C2) \text{ where } C1$$

$$R \text{ where } (C1 \text{ and } C2)$$

2.12 证明：对于任意表R，表R上的属性A，以及A的域上的某个值a，必定存在表S，使得选择($R \text{ where } A = a$)可以表达成 $R \bowtie S$ 。这里的表S独立于表R。

2.13 • R和S是两个兼容关系，如果把R和S看成是元组的集合，那么，如果R中的每一个元组都存在于S中我们可以使用集合的符号来表示包含关系，记作 $R \subseteq S$ 。证明：如果 $R \subseteq S$ ，那么 $R \bowtie T \subseteq S \bowtie T$ ；如果U和V是具有相似标题的两个表，那么 $U \div R \supseteq U \div S$ ， $R \div V \supseteq S \div V$ ；如果C是一个条件，那么 $(R \text{ where } C) \subseteq (S \text{ where } C)$ 。

2.14 R和S是两个关系， $\text{Head}(R)=H$ 且 $\text{Head}(S)=K$ 。证明下面的等式成立：

$$(R \bowtie S)[H] = R[H] \bowtie S[H \cap K] = R \bowtie S[H \cap K]。$$

这些关系的值称为R和S的半连接，记作 $R \ltimes S$ 。

注意， $R \ltimes S$ 与 $S \ltimes R$ 不等价。半连接 \ltimes 在分布式数据库中具有相当重要的作用。

2.15 R和S是两个兼容关系，H是两者的共同标题的子集的属性的集合。证明下面的等式或者举出反例。

$$(a) (R \cap S)[H] = R[H] \cap S[H]$$

$$(b) \bullet (R \cup S)[H] = R[H] \cup S[H]$$

$$(c) (R - S)[H] = R[H] - S[H]$$

如果你能举出反例，请说明等式的一边包含在另外一边。

第3章 基本SQL查询语言

本章介绍数据库语言SQL的基本形式。SQL是一种允许我们对计算机化的关系数据库进行查询和操纵的语言。自从20世纪80年代早期以来，SQL就一直是关系数据库管理系统(RDBMS)的语言，而且它对本书列出的许多概念都有着重要的意义。现在，SQL语言正从关系形式(ANSI SQL-92标准)转向对象-关系形式(ANSI SQL-99标准，1999年颁布)。SQL-99应被视为SQL-92的扩展，它并没有改变任何有效的早期语言。本章介绍的SQL基本形式(或基本SQL)包含了在绝大多数主要数据库产品中使用的SQL-92和SQL-99标准的所有关系性能。在一般情况下，我们定义的基本SQL与ANSI SQL标准的基本子集最为接近，分别被称为Entry SQL-92和Core SQL-99。我们也将介绍Entry SQL-92和Core SQL-99之外一些已被广泛实现的特征。在定义基本SQL时我们的试金石是要提供一种绝大多数主要RDBMS产品完全支持的语法。第4章将涉及更加新的对象-关系语法，它将在新的方向上对基本SQL进行扩展。在第3、4章里，我们将给出在常用数据库产品中频繁出现的例子。

3.1 引言

我们首先从总体上介绍SQL的特性，然后说明多种SQL标准和产品语言以及我们在具体的叙述中将如何处理这些标准和产品语言。

1. SQL特性

在第2章中我们看到了如何构造关系代数查询来回答对数据库信息的查询请求。在第3章里我们将学习如何使用Select语句这样的形式来构造相应的SQL查询。我们将会看到，在构造查询时SQL的Select语句在许多方面都要比关系代数更为灵活。然而，和关系代数相比，就能力而言，SQL并没有根本的改进。对于一些经过精心考虑的扩展语法来说，其中没有一种是关系代数所不能实现的。正因为如此，在关系代数查询方面的经验可以成为用SQL来实现查询的良好借鉴。同时，SQL和关系代数在许多方面的概念模型又有着相当大的差异，而在熟悉关系代数方法的过程中获得的认识能增强对SQL能力的理解。

你将遇到的最重要的SQL特征是它在计算机环境下交互地构造查询的能力。SQL的Select语句比相对简单的关系代数更为复杂也更难掌握，但只要有机会上机，只需几个实验就能打消你对使用SQL所持的怀疑态度，你也就不会再感到疑惑或动摇了。本章讨论的交互式环境允许你在显示屏上键入一个查询并且立即得到答案。这种交互式查询有时被称做即席查询。这个术语涉及到这样一个事实，即一个SQL Select语句是在键入一些输入行之后立即形成的，并且对一个用户来说该语句不依赖于前而的任何交互操作。这种不依赖于用户会话中前面交互操作的性质也被称做非过程性。SQL在这方面也不同于关系代数，因为在关系代数中为了表示一个表与自身的笛卡儿积，前而可能需要一个定义表别名的语句。SQL与诸如Java或C之类的过程性语言之间的差别是深刻的：你不需要为了实现一个SQL查询所完成的功能而编写一个程序，你只需要输入相对较短的、独立的查询文本然后提交。

当然，一个SQL查询可能会相当复杂。在图3-1中我们列出了基本SQL Select语句的完整

形式。这个完整形式的一个有限的部分被称为子查询，对它的定义是递归的，而完整的SQL Select语句形式有一个子句。不管怎么说，你不应被SQL语句的复杂性吓倒。Select语句的非过程性意味着它与使用菜单的应用有许多共同之处。在使用菜单的应用中，用户需要从菜单中填写一组选项然后按下回车键来立即执行它们。Select语句的各种子句对应于菜单选项：你偶尔会需要所有的子句，但并不需要在每次查询的时候都用上所有的子句。

```

subquery ::=

    SELECT [ALL | DISTINCT] { * | expr [[AS] c_alias] {, expr [[AS] c_alias]...} }

        FROM tableref {, tableref...}

        [WHERE search_condition]

        [GROUP BY colname {, colname...}]

        [HAVING search_condition];

    | subquery {UNION [ALL] | INTERSECT [ALL] | EXCEPT [ALL]} subquery

select statement ::=

    subquery [ORDER BY result_column [ASC|DESC] {, result_column [ASC|DESC]...}]

```

图3-1 基本SQL子查询和Select语句的通用形式

除了需要有SQL Select语句构造查询的能力以外，一个关系数据库系统还必须提供执行一系列其他运算和内务处理任务的方法。在这一章的开始部分，我们会学习如何执行某些简单的数据定义语句，比如我们在CAP数据库中建立表格时所需的SQL Create Table语句的有限形式。这类内务处理任务通常由数据库管理员(DBA)来完成。我们将大部分由数据库管理员用作数据定义的其他SQL语句推迟到第7章讲述，比如创建数据视图的语句以及对数据修改实施完整性约束所需的更复杂的Create Table语句形式。对复杂Select语句的研究占据了本章大部分的篇幅，而我们仅在3.10节中介绍SQL的其他数据操纵语句(data manipulation statement)。有了这些新的语句，我们就能对数据库中的表进行所有必要的修改操作：对一个表插入新的行，从表中删除现有的行，以及改变现有行的某些列的值。整个第3章都用来介绍能在我们已描述的交互式环境中使用的SQL语句。在第5章里，我们将考虑如何编写程序来执行这些以及许多其他SQL语句。一种在程序中使用的SQL形式被称做嵌入式SQL，它在某种程度上不同于交互式SQL。

2. SQL的历史——标准和产品语言

最早的SQL原型是由IBM的研究人员在20世纪70年代开发的，该原型被命名为SEQUEL(由Structured English QUery Language的首字母缩写组成)。许多人仍然将在这个原型之后出台的SQL语言发音为“sequel”，尽管根据ANSI SQL委员会的规定，正式的发音应为：“ess cue ell”。20世纪80年代早期颁布了几个使用SQL产品语言的关系数据库系统产品，而从那时起SQL开始成为国际标准的数据库语言(尽管我们将看到“标准”的SQL是不确定的)。在SQL成为标准的同时，一些老的语言也由于支持它们的商用数据库系统厂商纷纷引进自己的SQL产品语言而逐渐被废弃。尽管绝大多数产品的SQL有着极大的相似性，但它们之间的一系列差别可能会使初学者们感到迷惑。正因为这样，本书集中介绍我们自己的“标准”SQL，我们一直称之为基本SQL(从现有的多种标准中提取当前的目标语言)，然后跟踪一些主要数据

库系统产品与该标准之间的重要差异：ORACLE颁布了版本8.1.5，INFORMIX颁布了版本9.2，DB2 Universal Database颁布了版本6（我们将这个版本称做DB2 UDB，以区别于别的版本，比如仅在OS/390上运行的DB2系统）。我们偶尔也会涉及到其他重要的产品，如SYBASE和MICROSOFT SQL Server。与数据库系统打交道的读者很可能正在使用这些产品中的一个。

现在，让我们来进一步讲述SQL标准。目前，被数据库产品和教材引用的有好几种SQL“标准”。由美国国家标准局（ANSI）在1989年采纳的规范被称为SQL-89，它被用来取代更早的1986标准，并且此规范也被国际标准化组织（ISO）采纳。SQL-86和SQL-89都失去了许多重要的DBA能力并形成了一种最小的标准集——一个绝大多数关系数据库产品在几年前就已经能符合的普通标准。这意味着对新的产品特征缺乏相应的指导，在开发的过程中就互不兼容并缺少互操作性（也就是说，用SQL写一个能在不同数据库产品上运行的程序已变得很困难）。X/Open标准是由一个最初包含了UNIX SQL生产厂商的开放系统协会提出的，它被用来开发ANSI/ISO标准扩展语法的公共集合并允许SQL应用程序在不同产品之间有更大的可移植性。

与此同时，ANSI和ISO的委员会正在采取新的措施。它们采用的方法是跳过现有的版本并为满足将来的需求而制定一种新的SQL标准。这样一来，生产厂商在开发新的产品功能时就能不为以前发生的不兼容性问题所困扰。在发展这种方法的过程中，建议在将来实现的性能被分配到两个不同的修订本中：SQL2和SQL3。在1992年，ANSI/ISO颁布了SQL2版本，标准的名称为SQL-92。SQL-92分成几个顺序级别：从代表SQL-89最小扩展集的“Entry”到“Intermediate”和“Full”。然而，主要的几个数据库生产厂商可能永远不会遵守SQL-92的高层部分，这已经是显而易见的事实。所以，一些功能在实际情况下将永远不会被采纳。完成于1999年的SQL-99修订本具有更加高级的特征（包括对象-关系特性）。一些数据库产品甚至在SQL-99定稿以前就已经实现了符合该标准的特征。SQL-99摒弃了“顺序级别”的概念而是创建了“Core”级别，这个级别集中了大家公认为重要的特征以及生产厂商为了某些特殊应用而提供的其他功能集合。

正如我们所能见到的那样，SQL标准是不确定的，而“标准”一词几乎失去了它的意义（尽管ANSI SQL标准委员会打算用SQL-99来取代SQL-92并仔细考虑了SQL-92到SQL-99的向上兼容性）。在这一章，我们将集中介绍SQL-99，在相关的地方还会引用X/Open标准。我们前而已说过，庞大的SQL-99（以及更老的Full SQL-92）标准的一些特征将永远不会被一些生产厂商实现，而X/Open标准却竭力强调系统之间的可移植性，因而不会包含不被成员厂商认可的特征。在这一章里，我们将介绍在某些产品中已实现的关系SQL-99特征，但我们将其中的对象-关系特性放到第4章来讨论。本章介绍基本SQL，它包含了SQL-92和SQL-99中已经被我们将要学到的所有主要生产厂商实现了的功能。第4章将覆盖目前一些数据库产品所支持的对象-关系特征等内容。

现在，我们来回顾一下本章所涉及的商用数据库产品的简短历史。尽管绝大多数最新的产品版本都支持对象-关系模型，但目前流行的仍然是关系数据库。这种情况在将来可能会有所改变。

目前销售量最大的数据库产品是ORACLE。该产品以其在不同的平台之间的可移植性而著称，但最突出的应用可能还是在UNIX和Windows NT系统上。在1997年的后期，ORACLE颁布的版本8是最早提供对象-关系特性的版本。

DB2产品最初由IBM在1983年颁布，用在IBM大型机的MVS操作系统上。IBM对关系模

型的执着是关系模型胜过诸如IMS和IDMS之类的老的关系产品的推动力。(IMS仍然被频繁地使用着，因为有太多的公司把它作为假定的系统结构来编写它们的操作型数据处理系统。)在1997年，IBM发布了一个新的对象-关系版本：DB2 Universal Database，版本5(DB2 UDB)。它可以在PC机和UNIX平台上运行。现在它也在OS/390上运行。

INFORMIX是具有对象-关系特性的第三大数据库生产厂商，它在1996年购买了Illustra产品。Illustra产品是为推进对象-关系标准而特地开发的，此产品非常之成功。

据历史记载，INGRES产品是加州大学伯克莱分校在DEC VAX和UNIX系统下开发的，并且被世界各地的计算机科学系广泛地采用。本书的第一版包含了INGRES的某些细节，但大多数大学已转向使用其他的数据库系统，所以目前的版本不再包含这部分内容。

我们刚才提到的每个产品都有许多版本，因为大约每年都会出现一个新的版本。而且，从目前的情况来看一个版本和下一个版本之间的差别还是相当明显的。在本书中我们将涉及ORACLE版本8.1.5(对象-关系版本)、INFORMIX Dynamic Server(INFORMIX)版本9.2以及DB2 Universal Database(DB2 UDB)版本6。一般说来，所有数据库产品的版本都具有“向上兼容性”，这意味着后来的版本可能提供新的特征但应该还支持早期版本的所有特征。这样，本书的内容对上述所有产品后来的版本应该也是适用的。

从所有这些对新数据库模型和未颁布标准的讨论以及主要数据库产品从一个版本到另一个版本的重大变化来看，你可能会认为数据库系统往往处于一种变化不定的状态。这的确是事实。尽管在从创始到现在的近40年里数据库领域里取得了巨大的进步，但我们目前却处在一个自17世纪科学方法论的首次提出以来还未曾在某个更加成熟的科学学科中出现过的发展阶段上。当前数据库系统的许多重要原理必将长久地保持其有效性，但我们仍然有希望看到在接下来几年里发生的重大变化。在本书中，我们强调基本的原理并指出它们在当前软件实践中的局限性。

3.2 创建数据库

在这一节的末尾，练习3.2.1给出了一个以本节和附录A介绍的SQL数据定义语句为基础的上机作业。该练习会让你创建一些数据库表来复制图2-2的顾客-代理商-产品数据库(CAP数据库)。以后的上机作业会假定这个数据库已存在并让你用SQL数据操纵语句来检索和修改这些表中的数据。

首先，你需要得到计算机系统的一个账号——登录到计算机的用户ID和密码——以及能让你进入数据库系统的数据库系统自身的一个独立账号。你的指导老师会告诉你如何得到这两个账号。一旦得到了这些账号，附录A的教程会教你一些在ORACLE和INFORMIX数据库系统中所需的基本技能。这些教程涉及到的技能包括如何从操作系统进入数据库系统，如何在数据库中建立表或删除表，如何将在操作系统文件中的数据导入一个已定义的数据库表中，以及如何交互地执行SQL命令。为进入ORACLE而键入sqlplus(所有的都是小写字母)或者为进入INFORMIX而键入dbaccess都会使你进入一个交互环境，在这个环境中你可以用许多命令来编写SQL语句，对它们进行编辑，将它们保存到操作系统的文件中并在以后把它们从文件中读回，将它们提交给数据库系统执行；等等。

为了在ORACLE系统中建立属于CAP数据库的CUSTOMERS表(我们采用小写的表名以区分SQL和关系代数)，你需要进入交互环境并键入以下SQL命令：

```
[3.2.1] SQL > create table customers (cid char(4) not null,
2   cname varchar(13), city varchar(20), discnt real,
3   primary key(cid));
```

注意，在第一行输入回车后，ORACLE SQL* Plus 环境会为第二行打印出一个提示符2。第三行的提示符是3，依此类推。可键入的行的数目是没有限制的。在用一个分号(;)结束一行以前，系统不会试图解释你已编写的代码。所有的SQL语句都用分号来表示结束。

此Create Table语句将导致一个名为customers的空表的建立。表的列名分别为cid, cname, city和discnt。主键只包含了单独的一列：cid。在每个指定的属性名后面紧跟着每个属性的类型(type)或数据类型(datatype)(即第2章中讨论的属性的域(domain))。因此，cname的类型为varchar(13)(一个最大长度为13个字符的可变长字符串)，而discnt的类型为real(ANSI数据类型，代表4个字节的实数，相当于C语言里的float型)。附录A的A.3节包含了在定义列时可以使用的许多其他可能的数据类型。在cid列的名字和类型之后的not null子句意味着在该列上不能出现空值：任何会导致该列出现空值的SQL数据操纵语句都会失败，并发给用户一个警告信息。回顾2.4节，我们知道一个空值代表一个未定义的或不适用的值。但是由于DBA决定将cid列作为每一行唯一的标识(表的键)，所以在这一列上不允许出现空值。与NOT NULL子句类似，Create Table语句的许多其他子句可以被用来确保附加的数据完整性。不过，在讲下一章以前我们还不需要Create Table语句的全部功能。现在为了定义CAP数据库中的表，我们在图3-2中给出Create Table命令的有限形式。

```
CREATE TABLE tablename (colname datatype [NOT NULL]
[, colname datatype [NOT NULL] ...]
[, PRIMARY KEY (colname [, colname ...])]);
```

图3-2 Create Table语句的有限形式

1. 标准格式约定

注意图3-1和3-2的SQL语句形式。其中大写的项必须完全按照图中所写的那样键入(除非在特定的例子里大写的项被正常地转化成小写。例如，基于图3-2形式的语句[3.2.1])。这些语句形式中的小写项，比如图3-2中的tablename，代表了由用户选择和命名的项。这样，以图3-2这种形式出现的每条命令都以create table打头，而tablename并不被逐字照搬地输入而是应该填入用户希望创建的表名，比如customers。

还有许多SQL语句不会用到的特殊字符，包括左右花括号{}、左右方括号[]以及竖杠|。由于这些字符从不出现在SQL语句中，所以我们可以使用它们，与省略号…(在同一行上的三个句号)一起来说明SQL的语法特征。

回顾出现在语句形式中的在上述字符序列之外的所有特殊字符，特别是逗号和圆括号，它们必须完全按照语句形式中所写的那样键入并且没有书写格式上的特殊含义。这样在图3-2中，在tablename后面的是一个左圆括号，之后紧跟着用逗号分隔的列定义，然后是一个关闭的右圆括号。一个表中可被定义的列的确切数目有一个视系统而定的上限，但我们在这些SQL语句形式中并不考虑这项限制。

当一个短语出现在方括号[]里面的时候，该短语是可选的。因此，我们发现图3-2中的短语NOTNULL在列名和数据类型描述之后可以出现也可以不出现。同样，对主键的定义也是可选的，而且一个主键可以由一列或多列组成。语句[3.2.1]可以视为图3-2语句形式的一个例子。

竖杠(|)被用来表示在许多可能的短语之间的一种选择。当其中有且只有一个短语必须出现时，该选择可以是强制的。在那种情况下，语法形式为（假定有三种选择）：选择1|选择2|选择3或者{选择1|选择2|选择3}（用花括号来对选择“分组”）。在第二种写法中，花括号被用来指明选择从哪里开始以及到哪里结束。例如，形式

```
term1 term2 | term3
```

是有歧义的。我们是给出了一个在term1 term2和term3之间的选择吗？选项之间的选择是从哪里开始的？注意当我们换一种写法时此问题是怎样消失的：

```
term1 {term2 | term3}
```

这种形式意味着我们可以先写term1，然后“分组”形式中的花括号提供了term2和term3之间的一个选择。因此，term1 term3就是可能出现的短语。

闭合花括号的这种“分组”用法不是图3-2所用的形式。在那里我们使用了约定俗成的写法：将短语放入花括号并以省略号结尾。比如，图3-2的最后一行{, colname...}说明该短语（初始的逗号和一些选用的列名）在实际应用中可能出现零次或多次，这意味着图3-2的语法要求一个表有一个或多个列定义，因为我们看到短语colname datatype [NOT NULL]先出现了一次，然后又在花括号里出现了一次，并且以作为分隔符的逗号打头，后面还跟着省略号。

竖杠也能在方括号内使用，此时可选择由竖杠分隔的短语中的一个，但整个短语是可选的（也就是说可能没有短语出现）。如果在这种情况下其中的一个短语还被加了下划线，这意味着缺省的选择为加下划线的短语。作为一个例子，我们看一下图3-1,它的开头一行为：

```
SELECT [ALL | DISTINCT] select_list
```

这里的ALL说明查询得到的所有行都将被放入结果表（即使有重复的行），而DISTINCT说明重复的行不会被放入结果表。如果在[ALL | DISTINCT]中没有做出选择，缺省的选择是ALL，即允许出现重复行。

2. 实践练习

如果你所用的数据库产品是ORACLE或INFORMIX，那可以读一下附录A，那里有一个关于建立CAP数据库所需技能的教程。即使你正在使用一个不同的产品，附录A也会提供一些必要的技能以及相对标准的Create Table命令。（作者以后可能会在本书的主页上提供其他产品的教程，网址为<http://www.cs.umb.edu/~poneil/dbppp.html>。）

练习3.2.1 在读完附录A对ORACLE和INFORMIX的使用说明后，如果还没有人为你建立一个数据库，那就用登录的ID的前六个字符(如poneil)作为dbname来创建一个数据库。接着，为该数据库建立如图2-2所示的表—customers, agents, products和orders。你应使用屏幕命令建立命令过程文件以便做尽可能多的工作。比如：看一下附录A.1 对ORACLE的说明中给出的cust.ctl的形式，并试着用文件名agents.ctl, prods.ctl和orders.ctl重复这一过程以建立其他的CAP表。然后在数据文件—custs.dat, agents.dat, prods.dat和orders.dat—中输入数据以便在导入数据时使用，接着再给出建立和导入表的命令。在建立了文件并将它们导入数据库之后，用适当的数据库命令来显示所有的表在数据库中的布局，然后用Select语句来显示每个表的内容。

3.3 简单的Select语句

SQL是用Select语句来完成查询的。Select语句的通用形式相当复杂，在接下来的几节里

我们要对该语句的特征做出说明。这一节将介绍SQL查询的一些最基本的特征，这些特征对应于关系代数的选择、投影和笛卡儿积运算。在关系代数运算的情况下，Select语句查询的结果本身就是一张表。我们也会简要地介绍一下一个简单的Select语句是如何被数据库系统当作访问计划来实现的。访问计划是一个对用户透明的循环程序，它访问对计算机上的数据库进行查询时所需的数据并产生期望的结果。

例3.3.1 找出住在纽约的代理商的aid值和名字。

在关系代数中这个查询可表示成：

```
(AGENTS where city = 'New York')[aid, aname]
```

为了在SQL中解决同样的问题并显示被检索的行，我们用语句：

```
select aid, aname from agents where city = 'New York';
```

该语句中的保留字（即在所有SQL Select语句中都会出现的字）是SELECT、FROM和WHERE。为了执行这个Select语句，SQL解释器从跟在FROM后面的表开始，首先执行WHERE后面指定的选择条件，然后提取由SELECT后面的字段名定义的投影（我们称之为选择列表）。注意，表的名字是用小写字符来指定的（因为它们在Create Table命令中是那样被定义的），而常量'New York'则要用单引号围起来。（在一些数据库产品中也可以用双引号，但是单引号总是可接受的而且是被推荐的标准的一部分。） ■

例3.3.2 在附录A中有一个Select语句的例子，它显示CUSTOMERS表中每一行的所有字段值以检查数据是否被正确地导入了表中：

```
select * from customers;
```

选择列表中的符号“*”是一个表示检索所有字段的简写符号。也就是说，在这种情况下没有对字段的有限集合进行投影。而且，在这种情况下所有的行都会被打印出来，因为没有选择条件的限制（WHERE子句没有出现）。 ■

Select语句不会自动删除结果中的重复行；如果要求结果中不出现重复行，必须明确地要求这项服务。

例3.3.3 检索订货记录中所有零件的pid值。如果我们提交以下的查询：

```
select pid from orders;
```

结果中会有大量重复的pid值，每个值分别对应于图2-2中orders表中相应的pid所在的行。例如，p01会出现五次，因为它分别出现在orders表中ordno值为1011，1012，1021，1016，1024的行中。为了确保结果中的每一行的唯一性，我们需要给出以下的查询：

```
select distinct pid from orders;
```

注意，保留字distinct实际上确保了检索后的每一行是唯一的。这种唯一性是相对其他行而言的；也就是说，如果提交如下查询：

```
select distinct aid, pid from orders;
```

仅确保了(pid, aid)这一对值的唯一性而没有保证pid值的唯一性。这样，尽管(a01, p01)会出现在orders表中ordno值为1011, 1012, 1016的行中，在结果中却只会出现一次。 ■

Select语句在关键词DISTINCT没有出现的时候允许出现重复的结果行；这意味着缺省的情况是不遵守关系规则3——行唯一性规则的。毕竟，去除重复的行需要额外的开销而且会导致实际信息的丢失，因为在例3.3.3的第一个Select语句中pid值p01出现五次可能正是用户感

兴趣的信息。

如果我们想要强调所有的结果行都要打印出来，即重复的行不要被去掉，我们可以通过在选择列表前使用保留字ALL来表达与DISTINCT相反的意思。

```
select all pid from orders;
```

由于ALL是缺省的选择，所以我们不需要特地输入，除非是为了更好的可读性。注意，这种标准的SQL行为是怎样提供一种不符合2.3节的关系规则3的情况的，因为在结果中出现的行可能是不唯一的。

例3.3.4 写一个关系代数表达式来检索所有满足以下条件的顾客-代理商姓名对(cname, fname)，其中的顾客cname通过代理商fname订了货。我们要涉及的表为CUSTOMERS、ORDERS和AGENTS，因为列cname在CUSTOMERS表中，列fname在AGENTS表中，而ORDERS表则跟踪所有的订货记录。在关系代数中我们可以用下列表达式来解决这个查询问题：

$$((CUSTOMERS[cid, cname] \bowtie ORDERS) \bowtie AGENTS)[cname, fname]$$

在关系代数中很重要的一点是要把CUSTOMERS表投影到列[cid, cname]上以去掉city这一列，因为我们并不要求(CUSTOMERS \bowtie ORDERS)和AGENTS之间要连接的行具有相同的city值。另一种方法是，我们用关系代数的笛卡儿积运算而不用连接运算：

$$(((CUSTOMERS \times ORDERS) \times AGENTS) \text{ where } CUSTOMERS.cid = ORDERS.cid \\ \text{and } ORDERS.aid = AGENTS.aid)[cname, fname]$$

在这种做笛卡儿积的方法中，我们通过明确指定值必须相等的列来限定要选择的行，这在连接中是自动发生的。在SQL中执行该查询的正规方法为：

```
select distinct customers.cname, agents.fname  
from customers, orders, agents  
where customers.cid = orders.cid and orders.aid = agents.aid;
```

SQL Select语句的语法与上述两个关系代数解法中的笛卡儿积形式是等价的。Select语句导致了以下几个（概念性的）步骤：

- 1) 计算出现在FROM后面的表之间的笛卡儿积（这被认为是Select语句中的FROM子句）。
- 2) 实施由WHERE后面的条件规定的选择操作（WHERE子句）。
- 3) 将出现在选择列表中的属性投影成结果表。在这里，我们看到当涉及到多个表的时候，SQL也具有使用限定的属性名的能力，比如CUSTOMERS.cid。然而，SQL标准认为在诸如此类的关系查询中，当列名仅在其中的一个表中出现时它就不必受到限定。这样，取代上面第一行的写法

```
select distinct customers.cname, agents.fname . . .
```

我们可以写成

```
select distinct cname, fname . . .
```



SQL-99和Full SQL-92标准规定了在SQL Select语句的FROM子句中执行的是连接运算（参见3.6节），但这种特性在大多数商用产品中还未得到实现。在没有这种功能的情况下，就必须通过进行笛卡儿积运算（在FROM子句中写一个由逗号分隔的参与笛卡儿积运算的表序列）并且在WHERE子句中包含表示参与连接的列的值相等的特定条件来模拟连接运算。

我们还可以写一个能完成算术运算的Select查询语句并且打印出选择列表中各列的结果值。

例3.3.5 在orders表的基础上生成含有列ordno, cid, aid, pid和profit的“表”，其中的profit是由quantity和售出商品的price计算而得，方法是全部销售收人减去60%的销售收人、顾客的折扣以及代理商的酬金百分率。

```
select ordno, x.cid, x.aid, x.pid,
       .40*(x.qty*p.price) - .01*(c.discnt + a.percent) * (x.qty*p.price)
  from orders as x, customers as c, agents as a, products as p
 where c.cid = x.cid and a.aid = x.aid and p.pid = x.pid;
```

注意字母x, c, a和p分别与orders, customers, agents和products表相对应，因为它们在FROM子句中被定义为表别名。相类似的，在关系代数中别名是通过书写`x:=orders`, `c:=customers`, `a:=agents`, `p:=products`来建立的。表别名在Select语句中是通过在表名后面紧跟可选关键词AS以及所希望的别名（中间不加逗号）来定义的。逗号被用来指明后而跟的是一个新的表名。注意如果关键词AS被省略了，SQL仍能识别出表别名，因为在表名和表别名这两个标识符中间没有逗号分隔。一个表别名仅在定义它的Select语句的上下文中与该表相联系；因此，相同的别名可以因不同的用途而在不同的Select语句中被定义。在该查询中使用表别名是为了为列名建立一个较短的限定形式。为了便于理解，我们在这个Select语句中使用了限定的列名：`x.qty`, `p.price`, `c.discnt`以及`a.percent`，尽管这些独一无二的列名并不严格地需要写成限定的形式。

此查询的选择列表说明了一个重要的特征，即我们能计算由常数项（比如.01）或来自当前连接行（比如`p.price`）的不同列值组成的表达式的值。SQL支持标准运算符(+, -, *, /)，我们也可以使用标量函数，比如`mod(n, b)`（n被b除时得到的余数），以及诸如`lower(c)`（它返回由所有从c中字符转化而来的小写字符组成的char型变量）之类的字符串函数都在ORACLE, DB2 UDB和INFORMIX中有所规定，但这些并不是标准的一部分（除了Full SQL-99支持`abs()`和`mod()`）。有关这类表达式的详细介绍在3.9节给出。注意这里有隐式类型转换，所以`c.discnt`（float型）和`a.percent`（integer型）相加能得到float型的结果。计算后的表达式的值没有一个明确定义的列名，而不同的产品在为需计算的列命名时使用的方法是不同的。在ORACLE中，此查询得到如下结果。

ordno	cid	aid	pid	.40*(x.qty*p.price) - .01*(c.discnt + a.percent)* (x.qty*p.price)
1011	c001	a01	p01	120.00
1012	c001	a01	p01	120.00
1013	c002	a03	p03	210.00
1014	c003	a03	p05	300.00
1015	c003	a03	p05	300.00
1016	c006	a01	p01	170.00
1017	c001	a06	p03	150.00
1018	c001	a03	p04	138.00
1019	c001	a02	p02	48.00
1020	c006	a03	p07	198.00
1021	c004	a06	p01	135.00
1022	c001	a05	p06	200.00
1023	c001	a04	p05	120.00
1024	c006	a06	p01	140.00
1025	c001	a05	p07	200.00
1026	c002	a05	p03	184.00

对于例3.3.5使用的表别名x, c, a和p, 不同的商用数据库产品使用了众多不同的称呼。ORACLE和INFORMIX使用的术语是别名或表别名, 该术语出现在这些产品的所有帮助文件中。DB2 UDB和ANSI SQL标准使用术语相关名。范围变量或元组变量偶尔也会被使用, 原因在后面解释。如果你想轻松地阅览各种商用数据库产品提供的指南, 这类术语是很重要的。例如, 在《*ORACLE Sever SQL Language Reference Manual*》(《ORACLE服务器SQL语言参考手册》)中, 术语别名、表被列在索引中, 而术语相关名就找不到; 在《*INFORMIX Guide To SQL Syntax*》(《INFORMIX SQL语法指南》)中, 别名是标准的说法; 而《*DB2 Universal Database SQL Reference*》(《DB2 Universal Database SQL参考》)使用的则是相关名这个术语, 而术语别名, 即便是出现, 指的也是另一个语法元素。

关于例3.3.5中代表利润的列表达式的列名显示问题, 不同的商用数据库产品采用了众多不同的方法。在这种情况下, 不存在可从表中直接得来的简单列名。不同的商用数据库系统产品在这一问题上存在着差别是因为早期的标准没有定义统一的方法。在ORACLE中, 符号表达式的完整文本将被复制并被用作列名, 这就像我们在例3.3.5的结果表中见到的那样。从SQL-92标准开始, 用户就可以为Select语句中的表达式列提供一个即席名称。在ANSI SQL标准文档中, 它被称做列名字(表明它将成为结果的列名); 在ORACLE中它被称为列别名, 以区别于我们已见过的表别名; 在INFORMIX中它被称做显示标签; 而在DB2 UDB中, 它被简单地称做“在Select语句中的一个列名”(基本上是遵从ANSI SQL标准)。在ORACLE里, 我们可以通过书写

```
select . . . expr [as] c_alias], . . .
```

在选择列表中给出表达式expr所代表的列的名字c_alias。目前, 所有的主要数据库产品都允许为这样一个经过计算而得来的列重新命名。比如在ORACLE中, 我们可以将例3.3.5的查询写成:

```
select distinct x.ordno, x.cid, x.aid, x.pid, .40*(x.qty*p.price)
    -.01*(c.discnt + a.percent)*(x.qty*p.price) as profit           -- column alias
  from orders x, customers c, agents a, products p
 where c.cid = x.cid and a.aid = x.aid and p.pid = x.pid;
```

以建立一个有如下表头的表。

x.ordno	x.cid	x.aid	x.pid	profit
---------	-------	-------	-------	--------

注意, 在任一SQL语句行中, 我们都能使用两个连续的连字符以表明跟在连字符后面的文本是注释。

例3.3.6 求出住在同一城市的所有顾客对。在关系代数中, 为解出该查询的结果, 我们会从计算CUSTOMERS表的两个副本的笛卡儿积开始, 然后使用一个适当的选择条件:

```
- C1 := CUSTOMERS, C2 := CUSTOMERS
(C1 x C2) where C1.city = C2.city and C1.cid < C2.cid
```

我们需要用两个不同的名字来标识CUSTOMERS表: C1和C2。这样, 笛卡儿积中的列将受到唯一的限定。正如我们在例2.7.3中见到的那样, 限制条件C1.cid < C2.cid使我们去掉了冗余的顾客对(即由同一顾客组成的对, 或是由不同顾客组成的等价对出现了两次)。在SQL中, 该查询可用如下Select语句来执行:

```
select c1.cid, c2.cid
```

```
from customers c1, customers c2
where c1.city = c2.city and c1.cid < c2.cid;
```

在这个SQL查询的例子里，为了考虑由同一表中的行组成的行对，表别名是必需的。如果没有这样的别名，我们就不能对笛卡儿积里名字相同的列进行限定。注意大写的表别名C1和C2在SQL中也完全能被接受；但在一般的情况下我们仍使用小写的名字。在SQL标准中名字C1和c1是等价的（大小写无关）。 ■

正如例3.3.4中提到的那样，求解例3.3.6的Select语句的概念性步骤如下：首先，FROM子句将表c1和c2等价于customers表并做它们的笛卡儿积；然后，WHERE子句限定了要从笛卡儿积中选取的行集；最后，从这个行集中获得的值被投影到选择列表上。这为我们展示了一个正确的概念性求解顺序，但如果这就是数据库系统在回答我们的查询时所实际遵循的过程，那就有些惊人了。为了引出一个被称做查询优化的研究领域，我们介绍一种被称为访问计划的不同算法，这是数据库系统在实际检索查询数据时会采用的一种方法。此方法也说明了一种不同的有助于我们描述求解过程的概念性观点。我们从表别名c1和c2开始考虑，因为它们是出现在FROM子句中的范围变量，即载有customers表的行位置值的变量。在嵌套循环中，两个范围变量c1和c2互不依赖地在customers表的行上变化而我们可以认为上述查询的结果是由图3-3的伪码产生的。

```
FOR c1 FROM ROW 1 TO LAST OF customers
  FOR c2 FROM ROW 1 TO LAST OF customers
    IF (c1.city = c2.city and c1.cid < c2.cid)
      THEN PRINT OUT SELECT_LIST VALUES: c1.cid, c2.cid
    END FOR c2
  END FOR c1
```

图3-3 例3.3.6的Select语句的嵌套循环访问计划伪码

图3-3检索到的值与我们期望得到的例3.3.6 Select语句的值是相同的。这种嵌套循环(nested-loop)算法产生了与我们实际建立的笛卡儿积 $c1 \times c2$ 相同的行对。实际上，这种嵌套的循环是我们能产生笛卡儿积行对的唯一途径。在图3-3的算法中，我们没有将这些行对放入一个临时表中，而是通过执行Select语句的下一个概念性步骤并实施WHERE子句的选择条件来缩短处理过程。结果，嵌套循环方法在空间利用率上比第一步实际建立笛卡儿积的方法要高得多。

嵌套循环算法仅仅是涉及到笛卡儿积操作的Select语句可选做访问计划的众多算法中的一个。还有别的算法存在，例如，可以利用能极大地改进性能的索引查找功能。但不管怎么说，嵌套循环算法代表了另一种有助于我们理解Select语句的概念性观点。我们明白了为什么范围变量是诸如c1和c2之类的变量的共同术语，因为它们在嵌套循环中是代表行位置的循环变量；每一个由行位置组成的对对应于笛卡儿积表中的一行。很明显我们可以将这种方法推广到在不同的或相同的表上定义的任意数目的范围变量上去。注意，如果指定的表没有别名，表名本身也可以被认为是一个载有特定行值的范围变量。让我们重新思考例3.3.4的查询过程：

```
select distinct customers.cname, agents.ename
  from customers, orders, agents
 where customers.cid = orders.cid and orders.aid = agents.aid;
```

在这里，我们认为该Select语句有三个独立的范围变量：customers，agents和orders。由这三个独立的变量在一个三重嵌套循环中占据它们各自表中的行位置而得到的值实际上就是会出现在三个表的关系代数笛卡儿积中的值。WHERE子句基本上选出了能由三个表进行连接而得到的行，只是这里不要求customers和agents的city列相等。

例3.3.7 找出至少被两个顾客订购的产品的pid值。我们可用表别名来解决这个问题：

```
select distinct x1.pid
  from orders x1, orders x2
 where x1.pid = x2.pid and x1.cid < x2.cid;
```

考虑此查询的最佳方法是x1和x2各自在orders表的行上变化（有时候它们代表同一行）。条件 $x1.cid < x2.cid$ 保证了这两行分别代表由不同的顾客订购的产品而且同样的行对只出现一次，而条件 $x1.pid = x2.pid$ 保证了两个顾客订购的是同一产品。我们打印出所有具有性质——被两个不同的顾客订购的产品。 ■

上述分析中很重要的一点是要确保被位于orders表的不同行x, y, z上的三个顾客订购了的产品在结果中不出现三次：x和y的组合一次，x和z的组合一次以及y和z的组合一次。为了去除重复的行我们需要使用例3.3.3中介绍的保留字DISTINCT。关键词DISTINCT往往加重执行查询的过程的工作量。访问计划通常要求结果行根据被检索到的列值排好序并且要测试连续行的相等性以避免重复。如果用户知道没有重复的可能性，那就有可能改进整个执行过程以避免这种额外的工作。然而，知道重复什么时候可能出现在连接查询以及其他一些形式中并不是一件简单的事。我们会在本章结尾部分的习题中继续讨论这一问题。

例3.3.8 查询那些订购了某个被代理商a06订购过的产品的顾客的cid值。设计这一查询所需要的重要思想我们以前在关系代数中已介绍过了（见例2.9.9）。我们必须认识到此请求并不是想要通过a06订货的顾客的cid值，而是比那间接得多。见图3-4，我们希望检索的行集被包含在最右面的那个圈中。

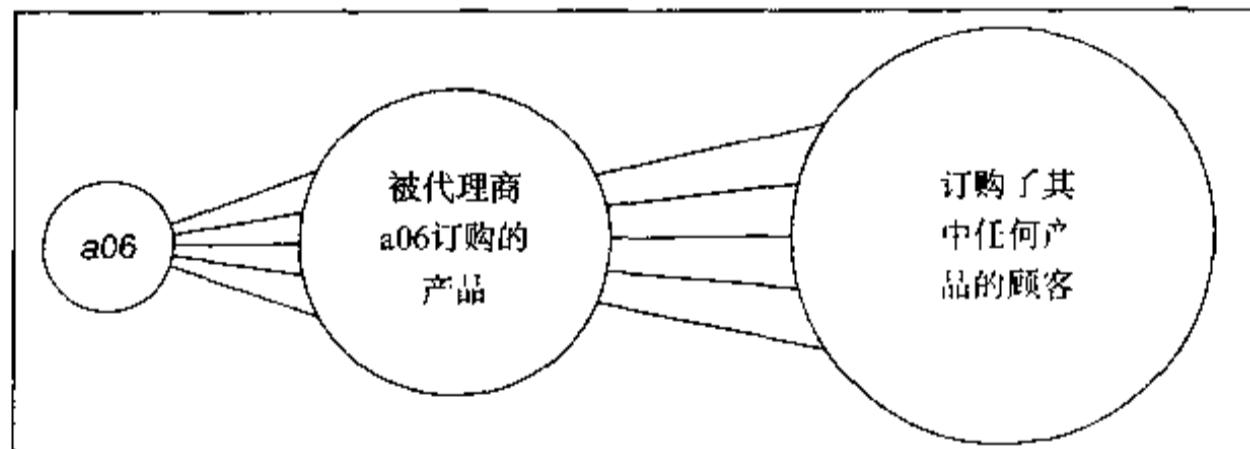


图3-4 例3.3.8中的对象集合以及它们之间的联系

首先，考虑被代理商a06订购的产品，即图3-4中中间那个圈包含的行集。

```
select x.pid from orders x where x.aid = 'a06';
```

被该查询从图2-2的orders表中检索到的产品ID集合为{p03, p01}。现在我们想要检索订购了其中的至少一个产品的顾客。我们需要考虑orders表中所有不同的行以列出这样的顾客。例如，订单号1017所在的行表明p03是通过代理商a06订购的，同时该行还显示顾客

c001订购了零件p03。然而，订单号为1013的行显示c002也订购了p03，尽管他不是通过代理商a06订购的。这两行我们都需要考虑以显示c002也在上述请求的选择序列中。我们将上述思想都集中到下面的查询中：

```
select distinct y.cid
  from orders x, orders y
 where y.pid = x.pid and x.aid = 'a06';
```

在这里我们看到，当范围变量x对应orders表中ordno=1017的行（由于x.aid=a06且x.pid=p03）而范围变量y对应orders表中ordno=1013的行（由于y.pid=x.pid且y.cid=c002）时，检索到的是cid为c002。此查询返回的是下面的表：

cid
c001
c002
c004
c006

回顾上述过程，该查询形式体现了一种极其重要的思想，即同时需要orders表上两个不同的范围变量以便将检索到的cid值与aid值为a06的代理商联系起来。 ■

在这一节里，我们介绍了图3-5列出的Select语句的有限形式。形式[ALL | DISTINCT]中的竖杠意味着用户能在ALL和DISTINCT这两项中选一个；该选择被包含在方括号([])里说明整个短语是可选的，而ALL被加了下划线则意味着如果该短语被忽略那它就是缺省的有效选项。WHERE子句中的搜索条件对象(search-condition)本身就显得相当复杂，它是本章许多节的讨论主题。

<pre>SELECT [ALL DISTINCT] { * expr [[AS] c_alias] [, expr [[AS] c alias]...]} FROM tablename [[AS] corr_name] [, tablename [[AS] corr_name]...] [WHERE search_condition];</pre>

图3-5 3.3节所介绍的Select语句语法

3.4 子查询

每个Select查询都会生成一张表，但是我们不能像在第2章里嵌入关系代数表达式那样任意地将一个Select语句嵌入另一个Select语句。最明显的一个限制是在Select语句的FROM子句中出现的表本身不能是任意Select语句的结果（Full SQL-92和SQL-99标准已去掉了这项限制，但这项特性在现有的数据库产品中并未得到完全实现）。这是SQL与关系代数的一个很重要的不同点，因为这意味着失去了关系代数所具备的某些嵌套功能的Select语句必须通过增加别的功能来作为补偿。我们在WHERE子句的搜索条件中能发现这些新增加的功能。我们将会看到，在搜索条件下嵌入某些形式的Select语句是完全可能的。

出现在另一个Select语句之内的Select语句形式被称为子查询（subquery）。子查询的通用形式缺乏完全Select语句所具备的某些语法元素，但我们还要过一段时间才会接触到这些语法元素，而图3-5中的形式就可以被当作是子查询。目前我们可以认为Select语句和子查询这两种形式是等同的。一个子查询能以众多方式出现在另一个Select语句的WHERE子句搜索条件

中。在这一节里，我们要学习许多谓词（predicate）和具有TRUE-FALSE值的逻辑条件，它们允许我们对子查询进行有意义的测试。我们首先考虑IN谓词，它测试一个集合的成员资格。

1. IN 谓词

假定我们希望求出通过住在Duluth或Dallas的代理商订了货的顾客的cid值，一种可能的解法是在SQL中对几个表做笛卡儿积，但这里关心的是如何举例说明IN谓词的用法以及新的子查询语法。

例3.4.1 求出通过住在Duluth或Dallas的代理商订了货的顾客的cid值。我们从找出所有住在Duluth或Dallas的代理商开始。此问题可用下面的查询来解决：

```
select aid from agents
  where city = 'Duluth' or city = 'Dallas';
```

该Select语句可被当作是一个值集（子查询）而放入一个更大的解决原问题的Select语句：

```
select distinct cid from orders
  where aid in (select aid from agents
    where city = 'Duluth' or city = 'Dallas');
```

从概念上来讲，我们把子查询（由括号包围的内层Select语句）视为向外层Select语句的WHERE子句返回的一个值集：“...where aid in (set)”。如果外层Select语句中的orders表当前行的aid值是内层Select语句返回的集合的一个元素（aid值在集合中），外层Select语句的WHERE子句就被认为是真(true)。特别地，对于图2-2的数据库内容，由子查询返回的aid值集合为a05和a06。接着，在外层的Select语句中，要选择的行被限定为orders表中那些aid值是该集合的成员的行。最后，结果行的cid值被打印出来。此查询的答案为

cid
c001
c002
c004
c006

成员资格的测试不仅能在由子查询提供的集合上进行，还能在显式定义的集合上进行，我们将在下面的例子中看到这一点。

例3.4.2 检索有关住在Duluth或Dallas的代理商的所有信息（与前一例子中的子查询非常相似）。这可以通过下面的查询来实现：

```
select * from agents
  where city in ('Duluth', 'Dallas');
```

我们看到，一个集合可以由置于括号中的被逗号分隔的常数序列构造而得。此查询返回如下的表：

aid	aname	city	percent
a05	Otasi	Duluth	5
a06	Smith	Dallas	5

上述查询与使用逻辑或(OR)的查询有异曲同工之妙：

```
select * from agents
where city = 'Duluth' or city = 'Dallas';
```

我们会发现，在下面的例子里出现了子查询的多层次嵌套。

例3.4.3 求出通过住在Duluth或Dallas的代理商订货的所有顾客的姓名和折扣。我们使用下面的语句形式：

```
select cname, discont from customers
where cid in (select cid from orders where aid in
(select aid from agents where city in ('Duluth', 'Dallas')));
```

注意，最内层子查询中的city列显然与agents表相关，而不是与最外层Select语句中的customers表相关。agents表是city这个列名最为局部的“作用域”。此查询返回下表：

cname	discont
TipTop	10.000
Basics	12.000
ACME	8.000
ACME	0.000

到目前为止的例子所涉及的子查询都是在没有接受任何输入数据的情况下向外层的Select语句传递一个行集。这些子查询被认为是非相关子查询，因为内层的子查询完全独立于外层的Select语句。更一般的情况是，我们能向内层子查询提供来自外层Select语句的数据。

例3.4.4 找出订购了产品p05的顾客的名字。当然，我们可用最直接的Select语句来解决这个问题：

```
select distinct cname from customers, orders
where customers.cid = orders.cid and orders.pid = 'p05';
```

但是，这里感兴趣的是子查询从外层Select语句中接收数据的解法：

```
select distinct cname from customers where 'p05' in
(select pid from orders where cid = customers.cid);
```

该查询返回下表：

cname
TipTop
Allied

如果这些Select语句没有包含关键词DISTINCT，那它们将在两个不同的行上检索到cname值Allied。在这章结尾的习题中，你需要验证许多有助于预测不带关键词DISTINCT的Select语句什么时候会产生重复行的规则。要特别注意的是，与上面第二个Select语句的子查询所引用的未限定的cid相关的应是orders表。当一个子查询包含单独一个表时，它就不需要以限定形式来访问局部列名。但在上例中，涉及customers.cid的子查询要访问外层

Select语句中的表，所以在那种情况下限定符是必需的。

一个要使用外层Select语句所提供的数据的子查询被认为是相关子查询。此类子查询通常具有这样的性质，即在外层Select语句开始执行以前它不能被一次性地彻底解出，而这意味着对执行过程的优化将很难进行。例如，在例3.4.4的子查询形式中，我们在确定customers.cid的值之前就求不出被内层子查询检索到的pid值集合，而customers.cid的值对于外层Select的每一行来说都是不同的。

尽管内层子查询能使用来自外层Select语句的变量，但反之则不然。我们可以将这视为一种作用域规则，这相当于我们在诸如C之类的众多编程语言中遇到的作用域局部性规则。在SQL中，一个子查询的局部变量在进入该子查询以前是不被定义的。

例3.4.5 如果我们想得到从代理商a03处订购了产品p07的顾客的名字，下面的查询形式是不合法的：

```
select cname from customers
  where orders.aid = 'a03' and 'p07' in -- ** ILLEGAL SQL SYNTAX
        (select pid from orders where cid = customers.cid);
```

上面的语句之所以不合法是因为对orders.aid的引用发生在它的正确作用域之外。 ■

IN谓词表达式的通用形式到目前为止已有两种不同的形式：

```
expr in (Subquery) | expr in (val [, val...])
```

expr记号代表我们在例3.3.5的选择列表中见到的那种字符串表达式或数值表达式，最常见的情况是一个简单的列名或常数；当然，expr也可能是如我们在下面例3.4.6中所见的一个“表达式序列”，即一个由逗号分隔的表达式序列。val记号代表一个常数，比如17或'Duluth'或者也可能是由逗号分隔的常数序列。

例3.4.6 检索由住在Duluth的顾客和住在New York的代理商组成的所有订货记录的ordno值。下面这种测试一对值是否包含于子查询的形式是基本SQL所支持的。

```
select ordno from orders
  where (cid, aid) in
        (select cid, aid from customers c, agents a
         where c.city = 'Duluth' and a.city = 'New York');
```

Full SQL-92和SQL-99允许对由逗号分隔的表达式序列、变量对或更长的元组进行针对子查询的成员资格测试，只要它们能如上所示被放置在括号中。也许你正在使用的产品还不具备这项扩展特性。（在写这句话的时候，该特性已在ORACLE和DB2中出现，但还未在INFORMIX中出现。）适用于这些产品的另一种标准SQL形式是这样的：

```
select ordno from orders x where exists
  (select cid, aid from customers c, agents a
   where c.cid = x.cid and a.aid = x.aid and c.city = 'Duluth' and a.city = 'New York'); ■
```

隐藏在expr IN (Subquery)格式后面的思想——通过求解子查询返回一个值集然后由外层Select语句来进行成员资格测试——是相当吸引人的。但这里很重要的一点是要能再次认识到这仅仅是一个概念性的过程；在很多情况下这并不是系统在建立访问计划时所采用的方法。我们在这一章的后面会看到，SQL Select语句是非过程性的，这意味着发出查询的用户

并不指定系统在回答这个查询时应使用的算法。我们已暗示过，系统中被称为查询优化器(query optimizer)的那个功能模块会将查询转化成众多不同的形式，然后从中选出能最高效地产生结果的那种形式。本章结尾部分的一道习题表明：到目前为止已介绍的所有涉及在子查询上使用IN谓词的查询都能被转化成不用子查询而对FROM子句中的表做笛卡儿积的另一种形式。这是实际执行查询时系统可能会采用的一种的转化方式。

除了IN谓词以外，还有NOT IN谓词。例如，子句

```
expr NOT IN (Subquery)
```

为真当且仅当求解出的expr值不在由子查询返回的集合中。NOT IN谓词也适用于在常数集上进行成员资格测试的形式。图3-6显示的是IN谓词的通用形式，可选的NOT被放置在关键词IN的前面。

expr [NOT] IN (Subquery) | expr [NOT] IN (val [, val...])

图3-6 IN谓词表达式的通用形式

2. 量化比较谓词

量化谓词将一个简单的表达式值和子查询的结果进行比较。它的通用形式显示于图3-7。

expr θ(SOME|ANY|ALL) (Subquery)
where θ is some operator in the set {<, <=, =, >, >=}

图3-7 量化谓词的通用形式

有十八种谓词满足这种通用形式。该形式中的SOME和ANY含义相同，而SOME则是绝大多数数据库产品的最新版本所推崇的形式。

现给出量化谓词的定义：对于某个属于集合{<, <=, =, >, >=}的比较运算符θ，两个等价的谓词exprθsome(Subquery)和exprθany(Subquery)为真，当且仅当至少存在一个由子查询返回的元素s，exprθs为真；谓词exprθall(Subquery)为真当且仅当对每一个由子查询返回的元素s，exprθs为真。下面我们给出说明这些谓词用法的例子。

例3.4.7 找出佣金百分率最小的代理商的aid值。这可以通过下面的查询来实现：

```
select aid from agents where percent <=all (select percent from agents);
```

子查询返回agents表中的所有percent值，然后量化比较谓词<=ALL要求被选择行的percent值小于或等于由子查询返回的所有值；因此，被选中的行的percent值将是同类中的最小值。 ■

例 3.4.8 找出与住在Dallas或Boston的顾客拥有相同折扣的所有顾客。(注意，在图2-2的例子中没有顾客住在Boston，但满足用户请求的正确查询形式应该不依赖于属于某个特定时刻的“偶然的”表内容。) 我们使用查询：

```
select cid, cname from customers  
where discnt =some (select discnt from customers  
where city = 'Dallas' or city = 'Boston');
```

被选择行的discnt值必须与被子查询检索到的至少一个discnt值相等，也就是说被选择的行拥有与某个住在Dallas或Boston的顾客相同的discnt值。注意谓词=SOME ...与谓词IN ...

具有完全相同的效果。这是一个相对令人吃惊的事实，我们会在本节的后面部分讨论这一问题。 ■

注意，虽然谓词 $\text{expr}=\text{SOME } (\text{Subquery})$ 与 $\text{expr } \text{IN}(\text{Subquery})$ 含义相同， $\text{expr } \text{NOT } \text{IN} (\text{Subquery})$ 却不等价于 $\text{expr } <>\text{SOME } (\text{Subquery})$ 。事实上，与 $\text{expr } \text{NOT } \text{IN} (\text{Subquery})$ 等价的是 $\text{expr } <>\text{ALL } (\text{Subquery})$ ，这也是你应该验证的。

ANSI SQL标准对ANY和SOME的定义是相同的，并且支持将谓词 $\theta \text{ ANY}$ 替换成 $\theta \text{ SOME}$ 的用法。这种替换之所以能成功是因为“any”在英语中的用法本身就非常容易引起歧义。

例3.4.9 求出所有满足以下条件的顾客的cid值：该顾客的discnt值小于任一(any)住在Duluth的顾客的discnt值。如果在用户提出这一请求的时候我们正好感到有些困惑，那很可能会有下面的Select语句，而它并没有实现我们所期望的结果：

```
select cid from customers
  where discnt < any -- ** WRONG EFFECT
        (select discnt from customers where city = 'Duluth');
```

将该Select语句应用于图2-2的customers表，上述子查询返回discnt值集合{10.00, 8.00}，其中的两个元素分别对应于cid值为c001和c004的顾客。根据定义，discnt值8.00会导致谓词 $\text{discnt} < \text{any}\{10.00, 8.00\}$ 为真。这是因为 $8.00 < 10.00$ (也就是说，子查询中至少一个元素使得 $\text{expr} < s$ 为真)。因此，整个Select语句将返回cid值集合{c003, c004, c006}，但这并不是我们实际想要的结果。原题中的请求可以改写为求出所有满足以下条件的顾客的cid值：该顾客的discnt值小于所有(all)住在Duluth的顾客的discnt值。在原来的表述中，词语any的英文含义起了误导作用。实现目标的正确Select语句为：

```
select cid from customers
  where discnt < all (select discnt from customers
                      where city = 'Duluth');
```

如果我们所用的谓词没有包含任一(any)，而仅涉及到某些(some)，那我们掉入这类陷阱的可能性就会小得多。

除了某些产品可能还不认同这种使用SOME的新形式以外，所有的商用数据库产品都支持上面提到的量化谓词集合。我们采用来自SQL-92和SQL-99标准的命名协议；参见本章结尾部分的推荐读物中由Jim Melton和Alan R · Simon编写的《*Understanding the New SQL*》(《理解新的SQL》)，有关这个主题的内容被编在该书的索引条目“量化比较谓词”下。《ORACLE Server SQL Language Reference Manual》(《ORACLE服务器SQL语言参考手册》)将该主题分作三个独立的部分来讨论：“ALL比较运算符”，“ANY比较运算符”以及“SOME比较运算符”。《ORACLE SQL Manual》(《ORACLE SQL手册》)没有包含“量化谓词”甚至是“谓词”的索引条目；取而代之的是“条件”。《INFORMIX Dynamics Server Guide To SQL Syntax》(《INFORMIX动态服务器SQL语法指南》)也把“谓词”称做“条件”。包括《DB2 Universal Database SQL Reference》(《DB2 Universal Database SQL参考》)在内的大多数其他参考文献则把该主题编在索引条目“量化比较谓词”下。

3. EXISTS谓词

EXISTS谓词测试被子查询检索到的行集是否为非空。图3-8显示了该谓词的通用形式。

[NOT] EXISTS (Subquery)

图3-8 Exists谓词的通用形式

谓词EXISTS (Subquery)为真当且仅当子查询返回一个非空的集合（如果集合中存在元素）。谓词NOT EXISTS (Subquery)为真当且仅当返回的集合为空。

例3.4.10 检索通过代理商a05订货的所有顾客的名字。此查询的基本思想是看orders表中是否存在一个连接cname和代理商a05的行。

```
select distinct c.cname from customers c
  where exists (select * from orders x
    where c.cid = x.cid and x.aid = 'a05');
```

注意子查询的选择序列为*，而不是某个单独的属性；这是测试子查询的结果是否为空的最简方法。从图2-2的表中检索到的是分别对应于cid值c001和c002的顾客名字：TipTop和Basics。注意上面查询中的相关变量x不是必需的，这里包含它仅仅是为了便于和下面的另一种解法做比较：

```
select distinct c.cname from customers c, orders x  
      where c.cid = x.cid and x.aid = 'a05';
```

此查询也解决了提出的问题。注意，如果上面的两个查询都没有包含关键词DISTINCT，那么customers表中拥有相同顾客名字的不同行各自对cid列进行限定会导致名字的重复显示。 ■

我们看到，第二种查询形式与使用了EXISTS谓词的第一种查询形式非常相似。为了对两者加以比较，需要考虑以下几点。上例中的第一个查询明确要求orders表中存在这样的行x：它把customers表中的行c和代理商a05联系到一起；而第二个查询则利用了这样的行x，但没有对它进行任何约束：x仅出现在WHERE子句的搜索条件中。这种出现在WHERE子句中但未被选择列表使用的范围变量类似于数理逻辑中的自由变量(unbound variable)。我们有理由问：在这种情况下，必须产生什么样的事件才能使WHERE子句为真？从直观上理解，我们一直认为答案是：如果存在使得WHERE子句为真的自由变量值（行），那WHERE子句就为真。也就是说，在上例中必须存在满足 $c.cid = x.cid$ 且 $x.aid = 'a05'$ 的行x。这样，自由变量x就好像也受到了谓词exists(select * from orders x . . .)的约束。

上述全部讨论表明，EXISTS谓词的正向形式（即不加NOT的形式）不是构造查询所必需的；使用一个自由变量也能完成查询任务。在求解习题的时候要尽可能地避免使用Select语句的复杂形式；特别是，不到万不得已的时候不要使用EXISTS。

例3.4.11 回顾例2.9.8中需要进行关系代数交运算的请求：求出既订购了产品p01又订购了产品p07的顾客的cid值。当时使用的查询语句是：

(0 where pid = 'p01')[cid] ∪ (0 where pid = 'p07')[cid]

在SQL中，我们很自然地会使用EXISTS谓词的正向形式来实现此查询：

```
select distinct cid from orders x
    where pid = 'p01' and exists (select * from orders
        where cid = x.cid and pid = 'p02'));
```

然而，我们也可以使用不带子查询的查询形式来实现此查询：

```
select distinct x.cid from orders x, orders y
  where x.pid = 'p01' and x.cid = y.cid and y.pid = 'p07'
```

另一方面，NOT EXISTS谓词也确实为我们带来了一些新的功能。

例3.4.12 检索没有通过代理商a05订货的所有顾客的名字。此查询正好检索那些没有被例3.4.10的查询检索到的顾客名字。注意，将例3.4.10的第二个Select语句的搜索条件简单地取反会产生错误的结果。

```
select c.cname from customers c, orders x
  where not (c.cid = x.cid and x.aid = 'a05'); -- ** WRONG EFFECT
```

在customers表中，什么样的行c能使条件not(c.cid = x.cid and x.aid = 'a05')为真呢？考虑满足c.cid = 'c001'的顾客c以及orders表的第一行（它满足x.cid = 'c001'且x.aid = 'a01'）。对于这些c和x，x.aid = 'a05'为假，因此条件not(c.cid = x.cid and x.aid = 'a05')为真。即使TipTop确实通过代理商a05订了货，将这一行的cname值TipTop当作结果来返回也是不正确的。上面作用在自由变量x上的隐含EXISTS位于以not打头的子句之外，其效果相当于当存在满足not(c.cid = x.cid and x.aid = 'a05')的行x时该条件为真；也就是说要存在满足c.cid <> x.cid或x.aid <> 'a05'的x。这两个条件都很容易实现（我们刚才找到的x满足c.cid = x.cid但x.aid <> 'a05'）。但我们实际想要做的却是当orders表中不存在满足条件(c.cid = x.cid and x.aid = 'a05')的行x时返回cname值。因此，我们必须显式地写出EXISTS谓词：

```
select distinct c.cname from customers c
  where not exists (select * from orders x
    where c.cid = x.cid and x.aid = 'a05');
```

此查询正确地解决了原问题。

例3.4.12 所使用的NOT EXISTS谓词似乎提供了一种表达查询的新能力。但事实证明，使用NOT IN 谓词和等价的<>ALL谓词，我们对许多查询已经具备了这种能力。

例3.4.13 我们重新考虑例3.4.12的查询：检索所有没有通过代理商a05订货的顾客的名字。这里要用两个等价的谓词——NOT IN和<>ALL来代替NOT EXISTS。两种查询形式如下：

```
select distinct c.cname from customers c
  where c.cid not in (select cid from orders where aid = 'a05');
```

以及

```
select c.cname from customers c
  where c.cid <> all (select cid from orders where aid = 'a05');
```

这里很自然地产生了一个问题，即NOT EXISTS谓词是否具有NOT IN 和<>ALL谓词所不具备的能力。我们将在本章结尾部分的习题中探讨这一问题。

NOT EXISTS谓词能被用来实现关系代数的MINUS运算。

例3.4.14 在例2.9.2中，我们为找出没有通过代理商a03订货的顾客的cid值这一请求设

计了一个关系代数表达式。当时有两种可能的解法。第一种解法仅检索到已订过货的顾客：

```
ORDERS[cid] - (ORDERS where aid = 'a03')[cid]
```

对应该表达式，我们有下面的SQL语句：

```
select distinct cid from orders x
  where not exists (select * from orders
    where cid = x.cid and aid = 'a03');
```

此查询的结果表如下：

cid
c004

例2.9.2中另一种包含了没有订货的顾客的解法是：

```
CUSTOMERS[cid] - (ORDERS where aid = 'a03')[cid]
```

我们将它转化成等价的SQL：

```
select cid from customers c
  where not exists (select * from orders
    where cid = c.cid and aid = 'a03');
```

在检索图2-2的数据库内容时这两种解法是没有区别的，因为那里不存在没有订货的顾客。■

注意，在高级SQL中有一个新的运算符（将在3.6节中讲述），这个被称做EXCEPT的运算符直接复制MINUS运算符的结果。然而，我们正在学习的许多产品还不具备该运算符。我们将在本章的3.6节介绍新的SQL运算符。

一般说来，如果R和S是两个兼容的表（即Head(R) = Head(S) = A₁ ... A_n），差R-S就可以通过下面的SQL语句来计算：

```
select A1 ... An from R
  where not exists (select * from S
    where S.A1 = R.A1 and ... and S.An = R.An);
```

4. SQL的缺点：过多的等价形式

下面，我们来探讨SQL语言有争议的原因之一：对于同一查询，经常会有众多不同的构造方法。

例3.4.15 考虑检索订购了产品p01的顾客所在的city名这一请求。有四种主要的Select语句表达方式：

```
select distinct city from customers where cid in
  (select cid from orders where pid = 'p01');

select distinct city from customers where cid = any
  (select cid from orders where pid = 'p01');

select distinct city from customers c where exists
  (select * from orders where cid = c.cid and pid = 'p01');

select distinct city from customers c, orders x
  where x.cid = c.cid and x.pid = 'p01';
```

此外，还有许多不那么显眼的方法，比如：

```
select distinct city from customers c where 'p01' in
  (select pid from orders where cid = c.cid)
```

■

由于谓词IN等同于=ANY，我们似乎有理由对两者的同时存在提出质疑。比如，去掉谓词IN可能就是个好主意，因为许多人发现它有大量令人疑惑的替换形式。事实上，只要能保留谓词[NOT] EXISTS，没有IN谓词和所有的量化谓词（any或all）也是完全可行的。在20世纪70年代早期，IBM的一个小组竭力提倡谓词的多样性，也就是要对用户更加友好而不只存在EXISTS谓词。这当然是有争议的。存在这些不同形式的一个重要缺点是：在开始构造查询的时候，我们很难单凭想象来对所有可能的方法进行彻底的搜索。在关系代数中，寻找一种构造一个新查询的方法所需的一般思维技巧是说一些诸如“噢，我知道要得到答案就必须对这三张表进行连接”之类的话。但既然有了子查询，我们就不得不为有什么新的功能可以避免简单的连接而操心。事实上，SQL语言的其他一些方面还是很强大的，因为它提供了如此多的形式而且其中的许多形式还有待于进一步地探讨。为了帮助大家解决这些问题，本章后面有一节专门列出了完整的Select语句通用形式，以便你对SQL所允许的所有形式进行彻底的搜索。而且，我们在正文和习题中都指明了一种新的SQL形式什么时候为该语言增加了额外的描述能力以及什么时候没有。幸运的是，我们需要学习的概念集合不是无限的，尽管在最开始的时候它似乎大得出奇。

3.5 UNION运算符和FOR ALL条件

3.3节介绍了SQL Select语句实现关系代数投影、选择和笛卡儿积运算的方法。3.4节详述了搜索条件的功能，给出了许多用于创建包含子查询的搜索条件的新谓词，并将模拟关系运算符差和交的方法加进了Select语句的技巧库。在这一节里，我们将看到如何用SQL来实现并(union)和除(division)运算（这些是SQL的新特性）。这样，SQL就完成了对整个关系运算符集合的模拟，所以我们现在似乎已具备了用SQL形式来表达任何关系代数查询的能力。尽管如此，下一节仍然会介绍一些执行交、差以及特殊连接运算的新SQL运算符。

1. UNION运算符

为了提供关系代数的 \cup (UNION运算符)运算，SQL需要一种新的Select句法。图3-9的UNION句法可被任意数目的子查询（见图3-5所列的语法）反复使用，只要它们所产生的表都是可兼容的。

Subquery UNION [ALL] Subquery

图3-9 子查询的UNION形式

在选用UNION运算符的时候，我们必须三思而后行。当前的许多产品都允许在完整的Select语句中使用UNION，但不允许在包含子查询的谓词中使用。也就是说，在图3-6、3-7和3-8的IN谓词、量化比较谓词以及EXISTS谓词所包含的子查询里，我们不能使用子查询的UNION形式。Core SQL-99允许在任何定义了子查询的地方使用UNION。DB2 UDB现在也允许这么做，而其他产品可能也将紧随其后。

例3.5.1 建立一个包含了顾客所在的或者代理商所在的或者两者皆在的城市的名单。这可以通过下面的Select语句来实现：

```
select city from customers
union select city from agents;
```

可以想象：如果一个城市同时满足了上述两个条件，我们就希望它在结果中显示两次（这样的结果包含了更多的信息）。在那种情况下，我们使用下列语句：

```
select city from customers
union all select city from agents;
```

注意，从customers和agents两表的笛卡儿积中检索单个city列的查询方法在这里是行不通的。

同以往一样，我们可以对任何表达式加括号，而且当涉及到的子查询数目大于或等于3时，括号可被用来判别UNION和UNION ALL的运算顺序。比如，考虑查询

```
(select city from customers
union select city from agents)
union all select city from products;
```

此时，如果Chicago同时出现于customers表和products表但不出现在agents表中，它将在结果中显示两次。但如果以如下方法移动Select语句中的括号位置，它就不再显示两次。

```
select city from customers
union (select city from agents
union all select city from products);
```

■

2. 除法：SQL “FOR ALL …” 条件

假定我们需要找出通过住在New York的所有代理商订了货的顾客的cid值。在关系代数中此查询可用下面包含了除法的表达式来解出：

```
ORDERS[cid, aid] DIVIDE BY (AGENTS where city = 'New York')[aid]
```

不幸的是，在SQL中没有等价的DIVIDE BY运算符，而且我们也不打算像处理UNION那样立即引入一种特殊的新SQL语法。注意此DIVIDE BY查询不能用诸如<ALL, <=ALL, =ALL之类的量化谓词形式来构造，因为在住在New York的代理商的属性上根本就没有出现任何比较操作。因此，我们不得不使用一种全新的SQL方法来实现该查询。这种方法由于其内在的难度，所以值得我们花较大的篇幅来详细介绍。

此方法以数理逻辑和数学证明概念为基础。我们的介绍从这样的问题开始：如何证明或反驳所有住在New York的代理商都为处在某个范围变量c所指定的行上的特定顾客c.cid订了货？显然，我们可以通过找出反例来反驳，即有一个住在New York的代理商没有为c.cid订货。如果我们把该代理商命名为a.aid，我们可以将此反例表示成SQL的搜索条件（为了便于引用我们将它标注为cond1，尽管这不是规范的SQL语法）：

```
cond1: a.city = 'New York' and
      not exists (select * from orders x
                  where x.cid = c.cid and x.aid = a.aid)
```

该条件说明a.aid所代表的代理商住在New York，但在orders表中却没有连接c.cid和a.aid的行；也就是说，c.cid没有通过a.aid订货。

现在来证明所有住在New York的代理商确实都为c.cid所代表的特定顾客订了货，我们必须写出能保证刚才构造的那种反例不存在的条件。也就是说，我们要确保没有代理商a.aid能使cond1为真。该条件也可以被表示成搜索条件，这里称之为cond2：

```
cond2: not exists (select * from agents a where cond1)
```

或者写成完整的形式：

```
cond2: not exists (select * from agents a where a.city = 'New York'  
and not exists (select * from orders x  
where x.cid = c.cid and x.aid = a.aid))
```

这是一个非常难掌握的条件，所以我们需要再花时间回顾一下cond2，它的内在逻辑是：不存在这样一个住在New York的代理商a.aid，他没有为c.cid订货（c.cid中的范围变量c在这里还没有被定义）。当然，这就意味着住在New York的所有代理商都为c.cid订了货。如果你也同意cond2有这样的含义，那我们就已经达到目的了，因为现在所要做的只是检索满足cond2性质的所有cid值。下面的例子综合了上面所有的论述。

例3.5.2 求出通过住在New York的所有代理商订了货的顾客的cid值。通过上述讨论，该请求的答案为：

```
select cid from customers where cond2;
```

或者写成完整的形式：

```
select c.cid from customers c where          -- select c.cid if...  
not exists (select * from agents a          -- ...there is no agent a.aid  
where a.city = 'New York' and              -- ...living in New York  
not exists (select * from orders x          -- ...where no order row  
where x.cid = c.cid  
and x.aid = a.aid));                      -- ...connects c.cid and a.aid
```

此查询应返回表：

cid
c001



我们非常能理解那些第一次遇到如此复杂的符号逻辑结构的读者的心情。当然，这是SQL查询中最难的概念。要想熟练地运用这种方法就必须采取有组织的循序渐进的步骤，掌握每一步的原理直到其中所包含的每一个概念都深深地扎根于脑海。

如果所面临的查询要求被检索的对象集合必须符合某个带有“所有”这类关键词的条件，我们就按照下列步骤来执行：

- 1) 为要检索的对象命名并考虑如何用英文来表述要检索的候选对象的一个反例。在该反例中，前面提到的“所有”对象中有至少一个对象不符合规定的条件。
- 2) 建立Select语句的搜索条件以选出步骤1所创建的所有反例。（步骤1和2必定会引用来自外部Select语句的对象，所以我们要在如何用这些外部对象所在的表来引用它们这一问题上表现出一定的灵活性。）
- 3) 建立包含步骤2所创建的语句的搜索条件，说明不存在上面定义的那种反例。这里将涉及到NOT EXISTS谓词。

4) 用步骤3的搜索条件来建立最终的Select语句，检索所期望的对象。

例3.5.3 求出住在New York或Duluth并订购了价格超过一美元的所有产品的代理的aid值。为了能进行上述步骤1，我们用a.aid来代表满足题意的代理商（但为了对范围变量保持灵活性，这里把它表示成?.aid以允许包含除agents以外的其他表）并构造反例：

"There is a product costing over a dollar that is not ordered by ?.aid."

接着（步骤2），表述收集?.aid所有反例的Select语句的搜索条件：

```
cond1: select pid from products p where price > 1.00 and not exists
       (select * from orders x where x.pid = p.pid and x.aid = ?.aid)
```

按照步骤3，建立表示这类反例不存在的条件：

```
cond2: not exists (select pid from products p where
       price > 1.00 and not exists (select * from orders x
       where x.pid = p.pid and x.aid = ?.aid))
```

最后，按照步骤4，建立最终的Select条件。注意，这里与例3.5.2不同的是步骤3仅建立了要检索的代理商所必须满足的条件之……：

```
select aid from agents a
  where (a.city = 'New York' or a.city = 'Duluth')
    and not exists (select p.pid from products p
      where p.price > 1.00 and not exists (select * from orders x
      where x.pid = p.pid and x.aid = a.aid));
```

该Select语句的结果如下表所示：

aid
a05

上面最先出现的条件(a.city = 'New York' or a.city = 'Duluth')被放入括号是因为AND连接符比OR连接符具有更高的优先级，而自然的优先级顺序将产生违背我们意愿的分组，即 ...where a.city = 'New York' or (a.city = 'Duluth' and not exists ...)。这样得到的答案为：

aid
a01
a04
a05

注意上面介绍的步骤序列通常会产生如下所示的嵌套子查询对：

```
select ... where not exists (select ... where not exists (select ...));
```

如果SQL的设计者们能像引入EXISTS谓词那样引入FOR ALL谓词，那就会方便许多。但不幸的是他们并没有那么做。[⊖]这意味着我们需要使用已存在的其他条件运算符来创建等价

[⊖] FOR ALL量词曾被建议用在标准ANSI SQL中，但该建议未被采纳。注意在数理逻辑中，FOR ALL运算符(\forall)和EXISTS运算符(\exists)被认为是量词而不是谓词。我们使用谓词这个术语是为了遵从目前的ANSI SQL标准协定。

的谓词。熟悉数理逻辑的读者都知道，下列重言式是使用两次NOT EXISTS谓词这种方法的理论依据：

$$\text{[3.5.1]} \quad \forall z (\exists y p(z, y)) \leftrightarrow \neg \exists z (\neg \exists y p(z, y))$$

如果用文字重述[3.5.1]，下面两句话是等价的：(1) 对所有的z，存在y使得依赖于z和y的语句p(z, y)为真，(2) 对于某个z，使得不存在使p(z, y)为真的y，这一命题为假。在例3.5.2中，z是agents表中city值为New York的行；y是orders表中的行；而p(z, y)则表示位于行z上的代理商通过行y所代表的订货记录连接到顾客c上（位于该子查询之外）。考虑语句[3.5.1]的最佳方法是：左边的形式是我们所希望创建的FOR ALL谓词，而右边的形式则如例3.5.2阐述的那样表示反例的不存在。注意重言式[3.5.1]右面的表达式包含了两个嵌套的NOT EXISTS谓词。

例3.5.4 找出订购了产品p01且价格超过一美元的所有产品的代理商的aid值。注意对在练习3.2.1中被导入示例数据库的图2-2的CAP表而言，该查询的结果是一个空表。但我们仍有必要创建正确的Select语句，因为一旦CAP数据库的内容有了变化，结果很可能就会变为非空。我们对例3.5.1的查询语句略作改动便能得到符合题意的查询，因为在上例中我们检索的是住在New York或Duluth并订购了价格超过一美元的所有产品的代理商。这里只需要定义原题所规定的第一个条件，而第二个条件则可以原封不动地照搬上例解法中的FOR ALL条件：

```
select a.aid from agents a where a.aid in
  (select aid from orders where pid = 'p01')
  and not exists (select p.pid from products p
    where p.price > 1.00 and not exists (select * from orders x
      where x.pid = p.pid and x.aid = a.aid));
```

用下列语句来表示第一个条件会显得更加自然：

```
select y.aid from orders y where y.pid = 'p01' and ...
```

这意味着FOR ALL条件内部所引用的范围变量a必须被修改：

```
select y.aid from orders y where y.pid = 'p01' and
  not exists (select p.pid from products p
    where p.price > 1.00 and not exists (select * from orders x
      where x.pid = p.pid and x.aid = y.aid));
```

回顾建立FOR ALL条件的四步过程中的步骤2，我们说过该步骤所创建的条件“必定会引用来自外部Select语句的对象，所以我们要在如何用这些外部对象所在的表来引用它们这一问题上表现出一定的灵活性。”这里a.aid必须变成y.aid就是这种情况的一个例子。 ■

例3.5.5 找出具有下列性质的顾客的cid值：如果顾客c006订购了某种产品，那要检索的顾客也订购了该产品。如何建立一个Select语句来满足该请求不是一目了然的，而且如果不是在讲述如何构造FOR ALL条件的章节中出现该问题，我们会感到更加困惑。此问题必须按照改写以后的语句来思考。原题中的请求可被改写成：找出订购了所有被顾客c006订购的产品的顾客的cid值。我们仍按上面介绍的步骤来求解。根据步骤1，我们称c.cid是符合题意的顾客并用英文构造反例：

“There is a product ordered by customer c006 that is not ordered by c.cid.”

现在我们把它表述成搜索条件，这是步骤2。我们将被c006订购的产品命名为p.pid，但仍保持其灵活性：

```
cond1: p.pid in (select pid from orders x where x.cid = 'c006')
       and not exists (select * from orders y
                         where y.pid = p.pid and y.cid = c.cid)
```

遵照步骤3，建立表示这种反例不存在的条件：

```
cond2: not exists (select p.pid from products p
                     where p.pid in (select pid from orders x
                                      where x.cid = 'c006') and
                           not exists (select * from orders y
                                         where y.pid = p.pid and y.cid = c.cid))
```

最后，根据步骤4，建立最终的Select条件：

```
select cid from customers c
  where not exists (select p.pid from products p
                     where p.pid in (select pid from orders x
                                      where x.cid = 'c006') and
                           not exists (select * from orders y
                                         where y.pid = p.pid and y.cid = c.cid));
```

该Select语句的一个明显的变体是

```
select cid from customers c
  where not exists (select z.pid from orders z
                     where z.cid = 'c006' and
                           not exists (select * from orders y
                                         where y.pid = z.pid and y.cid = c.cid));
```

结果如下表所示：

cid
c001
c006

经过充分的练习，在一般情况下将FOR ALL条件表述成反例不存在并由此立即建立所需的SQL语句已成为可能。我们通常可以认为最内层子查询所要做的就是在orders表中选出连接外层Select和子查询的行。

例3.5.6 找出被所有住在Duluth的顾客订购的产品的pid值。我们要表达的意思是不存在没有订购我们要检索的pid并住在Duluth的顾客。

```
select pid from products p          -- Retrieve product p.pid if
  where not exists
    (select c.cid from customers c      -- ... there is no customer
     where c.city = 'Duluth'           -- ... in Duluth
     and not exists
       (select * from orders x        -- ... where no row in orders
        where x.pid = p.pid           -- ... connects p.pid
        and x.cid = c.cid));          -- ... and c.cid
```

现在，我们能得出结论：SQL有能力计算关系代数所能计算的一切。这一结论的公认术

语是：SQL是关系完备的（relationally complete）。我们在下面几节里将看到，SQL在功能上超过了关系代数。现在是一个停下来完成本章末尾开头一批习题(3.1~3.7)的好时机，这些习题反映了到目前为止已介绍的所有Select语句特征。

3.6 高级SQL语法

从3.1到3.5节，我们介绍了实现所有关系代数查询所必需的SQL功能。这几节所介绍的基本SQL语法，除极个别的一些以外，都在几个主要的数据库产品中得到了实现。这一节将介绍一些高级SQL运算符，它们并不统一适用于所有的数据库系统而且不是Entry SQL-92的一部分。但另一方面，这些运算符又几乎都在Core SQL-99中，所以很可能会出现在未来的数据库产品中。本节将介绍的所有语法都不属于基本SQL。你必须仔细阅读以便了解哪些功能是你正在使用的数据库产品所支持的。大多数的高级语法只不过是我们已见过的某些关系查询提供了一种新的解法。尽管如此，还是有一些新增加的特性极大地增强了SQL的能力，比如在FROM子句中实现查询的特性。

我们首先说明模拟关系运算交和差的SQL运算符，以后还会介绍一种针对出现在FROM子句里的Select语句的新语法，包括多种连接运算符。

1. 高级SQL中的INTERSECT和EXCEPT运算符

为了模拟关系代数的 \cap (INTERSECT)和 $-$ (DIFFERENCE)操作，Full SQL-92和SQL-99提供了两个新的运算符：INTERSECT和EXCEPT。这些运算符的使用方法与图3-9所示的UNION用法是一样的。事实上，图3-9所显示的是Entry SQL-92语法。图3-10则显示了一个更大的Full SQL-92语法的子集，包含了当前一些产品所支持的集合运算UNION、INTERSECT和EXCEPT的用法。在所有这些运算中，Core SQL-99对UNION[ALL]和EXCEPT做出了规定。

```
subquery {UNION [ALL] | INTERSECT [ALL] | EXCEPT [ALL] subquery}
```

图3-10 包含UNION、INTERSECT和EXCEPT的高级SQL子查询形式

图3-10说明两个子查询能通过UNION、INTERSECT和EXCEPT运算符来连接以生成一个新的子查询（它既可以被看做一个完整的Select语句也可以被当成子查询而放在某些谓词中）。这个结果子查询随后又可以递归地替换图3-10形式中的任何一个子查询，这样，任意数目的子查询都能通过这些运算符连接起来。同以往一样，这里可以用括号来保持运算次序并消除二义性。如果把表看做行的集合，那INTERSECT运算符的含义是不言自明的，而EXCEPT运算符则模拟关系代数的DIFFERENCE运算。

我们已从对结果包含一个公共行的两个子查询进行UNION ALL操作的例子中看到：最终的UNION结果将包含两个相同的行。这从另一个侧面说明：关键词ALL会促使UNION考虑重复行的数目。对于同样的两个子查询，两次UNION ALL运算的结果会包含三个重复行，依此类推。考虑对两个子查询Q1和Q2执行INTERSECT ALL运算：

```
Q1 INTERSECT ALL Q2
```

由于查询Q1和Q2本身可能都是任意次UNION ALL运算的结果，所以Q1中可能有三个重复行而Q2中可能有两个与之相匹配的重复行。此时，INTERSECT ALL的结果会同时考虑两边的重复行数目。在这种情况下，最终的结果将包含两个这样的重复行（同一重复行在两个操作数中的最小数目——这与我们希望那些不出现在INTERSECT两边任一子查询中的行在最终结果

中的行数为零的想法是一致的)。另一方面,如果关键词ALL被省略了,那么我们仅认为上面提到的那一行同时在Q1和Q2中(行的数目只可能为0或1),所以这一行会出现在结果中。

同样,EXCEPT ALL运算符也会考虑重复行的数目:

```
Q1 EXCEPT ALL Q2
```

如果Q1的结果包含了三个相同的行且Q2的结果也包含了两个与之相匹配的行,那EXCEPT ALL子查询的结果将包含一个这样的行。也就是说,在EXCEPT ALL的结果中,同一行的出现次数将是该行在Q1中的出现次数减去它在Q2中的出现次数(但最小值应为零)。如果关键词ALL被省略了,那上面提到的重复行将被看做同时在Q1和Q2中出现了一次,所以它不会出现在结果中。注意Core SQL-99只包含了简单的EXCEPT而没有包含EXCEPT ALL。

如果两个由子查询计算而得的表具有相同的列数而且它们在Create Table语句中从左到右的列数据类型是兼容的,那么它们就能成为UNION、INTERSECT或EXCEPT运算的操作数。比如,类型为char(5)的列可以和类型为char(10)的列进行比较,该列在最后的结果中将拥有更通用的数据类型char(10)。

INTERSECT和EXCEPT运算符并没有增强Select语句的能力(我们已经能用NOT IN和NOT EXISTS谓词来实现这些操作)。但关键词ALL确实增强了Select语句的能力,尽管它可能有用也可能没用。

例3.6.1 回顾例3.4.11中的请求:求出既订购了产品p01又订购了产品p07的顾客的cid值。有好几种查询形式可以解决该问题:

```
select distinct cid from orders x
  where pid = 'p01' and exists (select * from orders
    where cid = x.cid and pid = 'p07');
```

以及

```
select distinct cid from orders x
  where pid = 'p01' and cid in (select cid from orders
    where pid = 'p07')
```

然而,不使用子查询也可以达到这一目的。

```
select distinct x.cid from orders x, orders y
  where x.pid = 'p01' and x.cid = y.cid and y.pid = 'p07';
```

新介绍的INTERSECT运算符为我们提供了另一种解法:

```
select cid from orders where pid = 'p01'
  intersect select cid from orders where pid = 'p07';
```



下面,我们用一个例子来说明EXCEPT谓词本身并没有给Select语句增加新的能力。

例3.6.2 检索没有通过代理商a05订货的所有顾客的名字。此查询可以用新引入的EXCEPT谓词来完成:

```
select c.cname from customers c
  except
    select c.cname from customers c, orders x
      where c.cid = x.cid and x.aid = 'a05';
```

但我们在例3.4.12中已给出了一种不同的解法。在这种解法中，NOT EXISTS谓词可被替换成NOT IN 和 \neq ALL：

```
select c.cname from customers c
  where not exists (select * from orders x
    where c.cid = x.cid and x.aid = 'a05');
```

还有许多数据库生产厂商没有实现INTERSECT和EXCEPT。X/Open标准支持UNION[ALL]但不支持INTERSECT[ALL]或EXCEPT[ALL]。（参见本章结尾部分“推荐读物”中的X/Open标准以及其他文献），ORACLE提供了UNION、UNION ALL、INTERSECT和MINUS（EXCEPT的别名），但不支持INTERSECT ALL或MINUS ALL。DB2 UDB版本5实现了所有这些形式：{UNION|INTERSECT|EXCEPT} [ALL]。INFORMIX版本9.2支持UNION [ALL]。

2. 高级SQL中的连接形式

在ANSI SQL-89标准中，FROM子句是非常基本的，它的形式如下所示：

```
FROM tablename [corr_name] {, tablename [corr_name]...}
```

Entry SQL-92沿用了这一形式。除ORACLE以外的所有主要产品都扩展了该形式以允许关键词AS出现在相关名前面，这种扩展形式已显示在图3-5中并被当作基本SQL来使用。

```
FROM tablename [[AS] corr_name] {, tablename [[AS] corr_name]...}
```

但如图3-11所示，Full SQL-92和SQL-99又进一步扩展了这种语法。图3-11从定义一个被称为tableref的通用形式开始，然后根据它来定义FROM子句（只要在FROM后面紧跟由逗号分隔的tableref序列即可）。我们要再一次提醒你：大多数商用DBMS产品还不支持图3-11所显示的语法。我们相信图3-11所描述的语法将来会在绝大多数数据库产品中得到实现，但所有的数据库系统都不可能完全遵循这种语法。仔细阅读本文以明确哪些产品具体支持图3-11中的哪些特征；或者查阅附录C，那里列出了针对特定DBMS产品的SQL语法。

<pre>tableref ::= tablename [[AS] corr_name] [(colname [, colname ...])] -- simple form (subquery) [AS] corr_name [(colname [, colname ...])] -- subquery as table tableref1 [INNER {LEFT RIGHT FULL} [OUTER]] JOIN tableref2 -- join forms ON search condition USING (colname [, colname ...])} FROM Clause ::= FROM tableref {, tableref...}</pre>

图3-11 SQL-99 对tableref的递归定义以及使用tableref形式的FROM子句

注意图3-11对tableref的定义是递归的，按照该定义形成的tableref对象随后可被递归地用于该形式中任何tableref对象所在的位置。图3-11中的任何FROM子句形式都可以被子查询（或Select语句）使用。在JOIN形式的外围可以使用括号。

图3-11所提供的第一项SQL新特性体现在tableref定义的第一行，即允许用带括号的序列[(colname){, colname...}]来为从一个表（FROM子句中的表）中检索到的所有列重新命名。如果使用了这样的序列，那么FROM子句所在的Select语句的选择列表也必须使用这些列名。在第二行，tableref可以是一个子查询，这使得我们能够在一个子查询或Select语句中自由地检索由另一个子查询提供的数据。与第一行不同的是，第二行中的相关名是必不可少的，因为我们需要一个表名以便引用由子查询生成的表。再强调一下，第二行中的列名集是可选的，它提供了一种为子查询选择列表所包含的每个元素指定一个列别名的方法。

尽管Select语句的结果本身就是一张表，但只有Full SQL-92和SQL-99（在扩展特性中）

允许一个子查询出现在另一个子查询的FROM子句中。图3-11的FROM后面能反复出现任意次tableref，这意味着我们能按顺序列出多个不同种类的tableref并且像以往对待一般的表那样对所有这些tableref做笛卡儿积。我们所涉及的一些数据库系统目前还不允许在FROM子句中出现子查询，比如，INFORMIX DS版本9.2就不支持这种用法，而ORACLE版本8和DB2 UDB版本5则允许这样做。然而，目前的ORACLE既不支持tableref形式中corr_name之后的[(colname[, colname...])]也不支持corr_name之前的关键词AS。作为已命名的扩展特性，FROM后面的子查询是Full SQL-99的一部分；也就是说，它不是Core SQL-99的一部分。

例3.6.3 检索对同一产品至少订购了两次的所有顾客的名字。有很多种方法可以实现此查询，这里使用把一个子查询当作表来检索的方法：

```
select cname from (select o.cid as spcid from orders o, orders x where o.cid = x.cid
    and o.pid = x.pid and o.ordno <> x.ordno) y, customers c
    where y.spcid = c.cid;
```

注意在该查询中我们为子查询的结果指定了一个别名y(这样做是必要的)，并为子查询所检索的列提供了一个别名(spcid)。我们要用这些别名来设定条件y.spcid = c.cid，从而选出表y和表customers的笛卡儿积中符合条件的行。 ■

下面介绍图3-11的第三行和第四行所显示的各种连接运算符。大多数数据库产品还没有全部实现这些连接运算符。

在图3-11的第三行中出现了关键词INNER，而且在关键词JOIN(不可省略)之前，在INNER与{LEFT | RIGHT | FULL} [OUTER]的选择之间，INNER是缺省的选项。事实上，一个INNER JOIN就是我们原先在关系代数中学到的那种连接类型，它与(FULL) OUTER JOIN的含义相反。我们先来看一下有关INNER JOIN的一些例子，但这些例子中的Select语句都不会刻意地包含关键词INNER，因为那显然是多此一举。

如果我们使用了图3-11第三行的关键词JOIN，那JOIN两边的两个tableref都需要一个关于它们如何做连接的说明，也就是有关哪些列要参与连接的说明。在使用关键词JOIN的时候，我们必须使用关键词ON或USING中的一个来指定要参加连接的列。ON形式比USING形式更为通用，但USING形式要相对简洁一些。ON形式还能处理一个连接列在两个参与连接的表中具有不同的列名这种情况。

当两个或更多个任意指定的列要在ON子句中进行比较时，我们称之为条件连接(condition join)。比如，如果有一个包含列cityname, latitude和longitude的cities表，我们就可以写：

```
select cname, city, latitude, longitude
    from customers c join cities x on c.city = x.cityname;
```

例3.6.4 检索至少订购了一件价格低于\$0.50的商品的所有顾客的姓名。(注意图2-2的CAP数据库没有包含满足该查询的行，但这显然与查询本身无关。)下面的解法是不正确的：

```
((ORDERS ⋈ PRODUCTS where price < 0.50) ⋈ CUSTOMERS) [cname]
```

这是因为这三个表的连接要求前两个表的笛卡儿积与customers表在city列上相匹配，而不是我们所希望的要求。正因为这样，我们要在表达式中给第一个连接加上限制，

如下所示：

```
((ORDERS ⋈ (PRODUCTS where price < 0.50) [pid]) ⋈ CUSTOMERS) [cname]
```

但在SQL中，我们必须显式地指定要参加连接的列，而且有两种方法可供选择。一种是用ON search_condition形式来指定所有要参加连接的列，从而实现条件连接：

```
select distinct cname from (orders o join products p on o.pid = p.pid)
    join customers c on o.cid = c.cid where p.price < 0.50;
```

或者我们可以通过列名连接(Column Name Join)来限制要参加连接的列，即用USING子句来指定两表中要参加连接的列名集合：

```
select distinct cname from (orders join products using (pid))
    join customers using(cid) where price < 0.50;
```

显然，在这些特性中存在着冗余现象。比如，假如我们已经有了ON子句，那USING子句显然是多余的——它只能处理对名字相同的列进行连接操作这类特殊情况。当然，你可能会问本节所介绍的任何特性是否真的为SQL增加了新能力。我们不是说过3.5节所列举的SQL功能已提供了我们所需的一切关系运算能力吗？显然，ANSI SQL-92委员会增加新的语法是为了使查询变得“更简单”。然而，对于那些必须学着记住所有这些语法形式的初学者来说，事情并不见得就变简单了。ANSI SQL标准委员会正朝着这个目标继续努力：现在，除关键词FULL以外的所有JOIN形式都已成为Core SQL-99的一部分。

3. OUTER JOIN

下面讨论图3-11中确实为SQL增加了新能力的一些语法，因为它们提供了经典关系代数所不具备的运算，即各种OUTER JOIN形式。

```
((LEFT | RIGHT | FULL) [OUTER]) JOIN
```

我们对OUTER JOIN（我们这里称之为FULL OUTER JOIN）的最早介绍出现在2.10节，即关系代数的结尾部分。同INNER JOIN一样，在任何类型的OUTER JOIN中都必须出现关键词ON或USING中的一个以决定哪几列将参加连接。

作为第一个例子，假定有两个表：S和T，表的内容如下：

S		T	
C	A	A	B
c1	a1	a1	b1
c3	a3	a2	b2
c4	a4	a3	b3

S和T的FULL OUTER JOIN将产生下面的结果：

```
select * from
S full outer join t using (A);
```

C	A	B
c1	a1	b1
c3	a3	b3
c4	a4	null
null	a2	b2

FULL OUTER JOIN的结果将包含自然INNER JOIN的全部结果行外加表S和T中互不匹

配的行，而且在不匹配的列上将出现空值。即使你所使用的系统不支持关键词FULL（Core SQL-99对它不做要求），你仍然可以通过如下形式的Core SQL-99子查询来实现FULL OUTER JOIN运算：

```
select * from S left join T using (A) union select * from S right join T using (A);
```

注意JOIN运算符作用在tableref的内部（见图3-11），而UNION则作用在子查询上（见图3-10）。所以，在用UNION将行集合并起来以前，需要建立一个子查询以选出连接结果表中的所有行。

作为另一个例子，假定（像我们在2.10节里所做的那样）有一个名为sales的表，该表包含两列：在orders表中出现的代理的aid以及表示每个代理商订货总金额的total列。注意，如果一个aid值没出现在orders表中，那我们就认为它也不会出现在sales表中。现在要做的就是列出所有的代理商以及他们相应的销售业绩，这基本上就是sales表中的aid列和total列，但为了便于理解，我们需要加入与每个aid值相对应的代理商名字。这似乎意味着需要进行如下形式的连接运算：

```
select fname, aid, total from sales join agents using (aid);
```

这种自然连接在下列两种情况下会产生问题：当agents表中有一个代理商在orders表或sales表中没有相应行的时候（因为该代理商没有订任何货），以及当有一个代理商已有了一定的销售业绩但在agents表中却没有相应的aid值的时候（这可能是由数据项中的错误或延迟造成的）。为了解决这一问题，我们应使用FULL OUTER JOIN运算符，因为该运算即使当一个表在另一个表中没有匹配行时也会在结果中保留参加连接的两表所包含的所有行，而且在所有的不匹配列上都会显示空值：

```
select fname, aid, total from sales full outer join agents using (aid);
```

注意，FULL OUTER JOIN这三个词没必要全部出现；根据SQL-99标准（图3-11），写成FULL JOIN就可以了，但要注意不要写成OUTER JOIN。同样，可以将LEFT OUTER JOIN写成LEFT JOIN，将RIGHT OUTER JOIN写成RIGHT JOIN。该查询的结果可能如下表。

aname	aid	total
Smith, J.	a01	850.00
Jones	a02	400.00
Brown	a03	3900.00
Gray	a04	null
Otasi	a05	2400.00
Smith, P.	a06	900.00
null	a07	650.00

在该表中，代理商a04（Gray）没有任何销售业绩，而代理商a07有销售业绩，但在agents表中却没有aname值（在agents表中可能也没有aid值——这种情况必须得到控制）。

LEFT OUTER JOIN只会在结果中保留运算符左边表中的行，即使当该行不能与右边表中的任何一行相匹配时也是如此。但右边表中的不匹配行就不能被保留下。

```
select fname, aid, total from sales left outer join agents using (aid);
```

在这种情况下，行(null, a07, 650.00)将出现在结果中，而(Gray, a04, null)则不会。

当运算符右边表中的行没能和左边表中的任何一行做成连接时，RIGHT OUTER JOIN只会在结果中保留右边表中的行。

```
select fname, aid, total from sales right outer join agents using (aid);
```

这里，行(Gray, a04, null)会出现在结果中，而(null, a07, 650.00)则不会出现。

4. 数据库系统所实现的连接形式

大多数数据库生产厂商还没有全部实现这些连接运算。DB2 UDB实现了我们已讲述的除(冗余的)USING子句以外的所有形式。ORACLE仅提供了左外连接和右外连接，并采用了与ANSI SQL标准不同的语法。在ORACLE中，一个外连接要用WHERE子句中的特殊条件来表示，例如：

```
SELECT . . . FROM T1, T2 WHERE {T1.c1 [(+)] = T2.c2 | T1.c1 = T2.c2 [(+)]}  
AND search cond;
```

T1和T2两表中仅有一表能在后面跟上带括号的加号，而这意味着参加连接的另一表将保留所有的行，甚至包括那些不满足连接条件的行。注意，如果两表之间有两个或更多的连接条件，那么无论在哪种情况下符号(+)都必须统一地作用在同一个表上，比如，T1.c1 = T2.c2 (+)和T1.c3 = T2.c4 (+)。还应注意的是，在N个表的连接中，可能会有N-1个表带有(+)。有意思的是我们能将表中的一列和一个后面紧跟符号(+)的常数作比较(select T1.c1 where T1.c2 = const1(+))，这会促使ORACLE系统返回在被比较的列上具有特定常数值或空值的所有行。

INFORMIX版本9.2同ORACLE一样，也只提供了左外连接和右外连接，但符号表示却略有不同。它没有往WHERE子句中的表名后面放置(+)，而是在FROM子句中将关键词OUTER置于第二个表名之前。用INFORMIX的术语来说，这就使得该表成为从表(subservient table)，并且在进行连接的时候使另一个表即主表(dominant table)中的全部行得以保留下来。在FROM子句中，关键词OUTER被放置在右边那个表的前面，通过交换左右两边的表我们可以实现分别等价于LEFT OUTER JOIN和RIGHT OUTER JOIN的运算。同ORACLE一样，在INFORMIX中参与连接操作的两表中只有一个表能具有这样的性质。

3.7 SQL中的集合函数

下面介绍已在所有的主要数据库产品中得到了全面实现的功能。SQL提供了五个作用在列值集合上的内置函数：COUNT、MAX、MIN、SUM和AVG。除COUNT外，这些集合函数都必须作用在由简单值组成的集合上，也就是，数字集合或字符串集合，而不是拥有多个列值的行集。

例3.7.1 求出所有订货交易的总金额。此查询可以写成

```
select sum(dollars) as totaldollars from orders;
```

把该查询用在实例数据库上将得到一个只含一行的表

totaldollars
9802.00

同数据库领域里的其他术语一样，集合函数的术语也有一些变体。X/Open标准和ANSI SQL标准称它们为五个集合函数(set function)；C.J.Date的文章以及INFORMIX文档都称之为聚集函数(aggregate function)(聚集的意思是“将许多不同的元素一起放入单个容器中，或合计”)；ORACLE称之为组函数(group function)；而IBM的产品，DB2 Universal Database，则使用术语列函数(column function)。如果你在同熟悉其他产品的人交谈时使用了集合函数这一术语，他们应该在理解上不存在什么困难。图3-12描述了基本SQL和Core SQL-99所规定的集合函数语法。

名称	参数类型	结果类型	描述
COUNT	任意(可以是*)	数值型	出现次数
SUM	数值型	数值型	参数的和
AVG	数值型	数值型	参数平均值
MAX	字符型或数值型	同参数内容一样	最大值
MIN	字符型或数值型	同参数内容一样	最小值

图3-12 SQL的集合函数

要特别注意的是，当我们把MAX和MIN这两个函数应用在字符型的参数集合上时，它们分别返回所有参数中按字母顺序排列的最大值和最小值。此外，某个数值集合的AVG与用COUNT除SUM所得的结果是一样的(只要这两种方法所处理的都是经过正确定义的数值并且不存在浮点溢出问题)。

例3.7.2 为了求出产品p03的订购总量，我们可以使用函数SUM并把它作用在满足相应条件的行集上：

```
select sum(qty) as TOTAL
      from orders where pid = 'p03';
```

答案是一个只含一行的表。

TOTAL
2400

600, 1000和800的和。 ■

不要把集合函数与诸如upper和substr之类的标量函数混淆起来，尽管标量函数也能出现在Select语句选择序列的表达式中。这些内置标量函数以单个行值为参数并为每一行返回一个与之对应的单值，而集合函数则把一个表中所有符合特定条件的值组合起来并返回一个单值。这样，Select语句

```
select sum(dollars) as TOTAL
      from orders where pid = 'p03';
```

将返回一个单值，而语句

```
select upper(cname) as UPCNAME from customers
      where discnt >= 10;
```

将返回一列值，这些值与customers表中满足discnt>=10的所有行一一对应。我们将在3.9节中给出内置函数列表。

例3.7.3 求出顾客总数的查询应使用COUNT函数而且不应局限于对简单值集合应用COUNT。下列两种形式均正确：

```
select count(cid)
  from customers;
```

或者

```
select count(*)
  from customers;
```

第一个Select语句计算在cid列下出现的值的数目。要特别注意的是，它对列中的空值是忽略不计的。此例中的两个语句会给出相同的答案，因为Create Table语句不允许在customers表的cid列上出现空值。 ■

我们可以要求集合函数仅对满足特定条件的不同 (distinct) 值起作用。

例 3.7.4 求出有顾客居住的城市的数目。Select语句

```
select count(distinct city)
  from customers;
```

将产生有顾客居住的不同城市的数目；同样，这里没把空值计算在内。由于在图2-2的CAP数据库中有两个顾客住在Dallas以及两个住在Duluth，上述语句的查询结果是3，这与没加关键词DISTINCT的同一查询的结果相差甚远：

```
select count(city)
  from customers;
```

该查询的结果是5。注意，求出有顾客居住的城市的数目这一请求一般都会被解释成不同城市的数目，所以第二种形式从某种意义上来说是不对的。 ■

查询的选择列表引用集合函数的基本SQL通用形式为：

```
SET_FUNCTION_NAME([ALL | DISTINCT] colname) | COUNT(*)
```

ALL或DISTINCT都可以出现在colname的前面，但缺省值为ALL。

将关键词DISTINCT和函数MAX或MIN放在一起使用是没有价值的，因为这两个函数都只从集合中选出一个值而且在任何情况下都不考虑重复行。此外，如下格式的查询：

```
select sum(distinct dollars) from orders where . . .
```

所针对的请求比较特殊，它强调相加的金额必须互不相同。在通常情况下使用函数SUM和AVG都不需要这样考虑。

这里有一个重要的约束，即在WHERE子句的比较操作中不允许出现集合函数，除非它们是出现在子查询的选择列表中。

例3.7.5 列出折扣值小于最大折扣值的所有顾客的cid值。下面的方法是不正确的：

```
select cid from customers
  where discnt < max(discnt); -- ** INVALID SQL SYNTAX
```

上述判断的基本依据是：该Select语句只含有一个范围变量（变量名为customers），它依次指向customers表中的每一行；但要使max(discnt)的值有意义，必须预先生成一个遍历customers表中所有discnt值的循环。然而，基本的Select思想却无法实现这样一个独立的循环。用户可以用一种不同的方法来检索所需的信息：

```
select cid from customers
```

```
where discnt < (select max(discnt) from customers);
```

注意，这里有两个名为customers但完全不同的范围变量，首先计算子查询以提供外层Select所需的值。由于子查询返回了一个单元素集合，所以这里能使用包含单个小于号(<)的比较运算。我们也可以使用比较运算符<ANY或<ALL，因为对返回值为单元素集合的子查询来说，这类比较运算与一般的小于运算是等价的。■

例3.7.6 回顾例3.3.7查询：找出被至少两个顾客订购的所有产品。我们现在能用另一种方法来解决该问题，把这种方法推广到多于两个顾客的情况下应该是易如反掌。

```
select p.pid from products p
  where 2 <= (select count(distinct cid) from orders
    where pid = p.pid);
```

此查询返回表：

pid
p01
p03
p05
p07

空值的处理

空值的概念最早出现在2.4节，但在介绍集合函数的定义以前，我们还无法讲述此概念所涉及的一些思想细节。空值是特殊的标量常数（在数值型或字符串型的列上有意义），它代表了未定义的（不适用的）或者有意义但在目前还处于未知状态的值。比如，当我们往employees表中插入新的一行时，由于薪水还未确定，我们可能希望该行在salary列上的值为空；另一方面，如果一个员工的工作类别是“图书管理员”，那他的percent（佣金百分率）可能为空值，因为从事这类工作的员工都是没有销售佣金的（这样的列值就是未定义的，或者说不适用的）。当前的大多数商用系统所实现的空值概念都还不区分这两种情况（尽管已经有人建议这么做了）。

注意 一些老的数据库系统没能很好地实现空值。在这些系统里，字符型的列用空白符来代替空值，而数值型的列则用零来代替空值；我们将看到这在某些情况下会产生严重的后果。

尽管我们暂时还不研究SQL Insert语句的完整语法，但为了说明一个表中怎样才会出现空值，下面的例子将涉及Insert语句的一些简单用法。

例3.7.7 往customers表中加入在cid, cname和city列上分别等于某些特定值的一行。我们从图2-2中看到：在customers表中还有一列：discnt，但这里假定在插入新行时该值还处于未知状态——也就是说，顾客的折扣值还没定。

```
insert into customers (cid, cname, city)
  values ('c007', 'WinDix', 'Dallas');
```

由于此Insert语句的列名序列或值序列都没有提到discnt列，新插入行的discnt值就缺省为空值。大多数数据库产品都允许在Insert的Values子句中用NULL来明确地指定一个空值；此

特性已成为SQL-92标准的一部分。 ■

出现在一个表中的空值具有许多重要的性质。首先，在搜索条件的任何一个一般比较谓词中出现的空值都会使整个谓词等于一个既不是TRUE也不是FALSE的特殊布尔值：UNKNOWN。对一个要被Select语句选取的行来说，它必须使WHERE子句中的复合谓词等于TRUE，所以UNKNOWN值在大多数情况下实际上与FALSE具有相同的效果。

例3.7.8 在例3.7.7往customers表中加入行(c007, Windix, Dallas, null)之后，下面的Select语句仍然检索不到这一行。

```
select * from customers where discnt <= 10 or discnt > 10;
```

这真令人惊讶！尽管此处的WHERE子句看上去覆盖了所有的情况，但如果在小于10、等于10或大于10的比较中出现了空值，那整个WHERE子句将等于UNKNOWN。如果要用包含discnt的谓词来检索discnt值为空的行，那唯一办法就是使用特殊谓词IS NULL：

```
select * from customers where discnt is null;
```

此外，SQL还特别提供了测试discnt值非空的谓词：discnt is not null，但是谓词discnt is null的一些变体形式，比如discnt = null，虽然也符合基本SQL语法，其效果却是错误的。如果SQL在检查语法错误的时候就能否决这样的语句，那我们就更容易避免这类错误。但与任何其他编程环境一样，SQL系统有时也会给出违背你意愿的结果。 ■

在这里，我们要重申上面提到的一个重要观点：如果一个空值出现在任何一个一般比较谓词中，那么该谓词等于UNKNOWN（我们将在3.9节中详细讨论这个特殊布尔值）。此规则甚至对形如col1 = col2且同一行上的col1列和col2列均为空值的相等谓词也成立；也就是说，在这种情况下，该谓词仍然等于UNKNOWN。此规则仅对特殊谓词IS NULL不适用。

值得注意的是，诸如0之类的常规值或者由两个连续单引号('')表示的空串都不具备例3.7.8所显示的特性。空串小于'a'（按字母顺序），零小于10，而被赋予了真实值（非空值）的行总能被某个谓词检索到。但用常规值或空串来表示空值有时会产生问题：一个薪水为空的新员工并不一定会成为扶贫计划的候选人，但如果他的薪水为零的话，显然他就会成为候选人。这里还产生了另一个问题：如果我们正试着计算一个部门的平均薪水，将新员工的薪水计为零来求平均是不合适的。一种更好的方法是把新员工完全排除在考虑范围之外，而这正是集合函数处理空值的方法。

例3.7.9 在例3.7.7把行(c007, Windix, Dallas, null)插入customers表之后，求出所有顾客的折扣平均值。

```
select avg(discont) from customers;
```

在该SQL语句中，空值在计算平均值以前就已被丢弃。 ■

有关空值处理的类似想法同样也适用于其他一些集合函数。正如我们在紧接着图3-12的讨论中提到的那样， $\text{avg}() = \text{sum}() / \text{count}()$ ，所以很显然sum()和count()函数也必定不考虑空值。这里还有一个有趣的问题：作用在空集（没有相关的行存在）上的集合函数将返回什么值？答案视函数而定：count()对空集返回零，而sum()、avg()、max()和min()都返回一个空值。

3.8 SQL中行的分组

SQL允许Select语句提供一种自然的“报表”功能：根据某些列值的共性把一个表所包含

的全部行分成若干个子集，然后对每个子集执行集合函数。比如，考虑查询

[3.8.1] `select pid, sum(qty) as total from orders
group by pid;`

Select语句的GROUP BY子句将产生一个行集，其效果好比是执行了下面由循环控制的查询：

```
FOR EACH DISTINCT VALUE v OF pid IN orders:  
    select pid, sum(qty) as total from orders where pid = v;  
END FOR;
```

Select语句[3.8.1]中的GROUP BY子句的结果为表：

pid	total
p01	4800
p02	400
p03	2400
p04	600
p05	2900
p06	400
p07	1400

出现在选择列表里的集合函数分别将每一组里的行聚集起来并为每个组创建一个值。这里很重要的一点是：选择列表中的所有属性都必须以单个原子值来对应由GROUP BY分得的每一组。比如，下面的Select语句是无效的：

```
select pid, cid, sum(qty) from orders  
group by pid; -- ** INVALID SQL SYNTAX
```

我们无法在与每个pid值相对应的单独一行上打印出多个不同的cid值；比如，由产品p01标识的第一组对应于一组cid值{c001, c004, c006}。但是，Select语句的GROUP BY子句本身却能包含多个列名。比如，我们可以对orders表中的两个ID属性进行GROUP BY操作并且在选择列表中检索它们。

例3.8.1 创建一个计算每样产品被每个代理商订购的总量的查询。在下面的Select语句中，我们按orders表中的pid和aid来分组：

```
select pid, aid, sum(qty) as TOTAL from orders  
group by pid, aid;
```

作为该查询的结果，我们得到下面的表：

pid	aid	TOTAL
p01	a01	3000
p01	a06	1800
p02	a02	400
p03	a03	1000
p03	a05	800
p03	a06	600
p04	a01	600
p05	a03	2400
p05	a04	500
p06	a05	400
p07	a03	600
p07	a04	800

如果同时使用WHERE子句和GROUP BY子句，我们可以对表的笛卡儿积进行分组。

例3.8.2 打印出代理商的名字和标识号、产品的名字和标识号以及每个代理商为顾客c002和c003订购该产品的总量。

```
select a.name, a.aid, p.name, p.pid, sum(qty)
  from orders x, products p, agents a
 where x.pid = p.pid and x.aid = a.aid and x.cid in ('c002', 'c003')
 group by a.aid, a.name, p.pid, p.name;
```

对于该查询，ORACLE返回下表：

aname	aid	pname	pid	sum(qty)
Brown	a03	pencil	p05	2400
Brown	a03	razor	p03	1000
Otasi	a05	razor	p03	800

由于a.aname和p.pname在选择列表中，所以GROUP BY子句就有必要同时包含a.aname和a.aid、p.pname和p.pid以向系统保证选择列表中的每一列都以单值来对应每一组。事实上，a.aid是agents表中行的唯一标识，而p.pid是products表中行的唯一标识，所以在GROUP BY列表中包含a.aname和p.pname不会引起组的进一步细分。然而，大多数数据库系统都还没有注意到这一事实，并且如果GROUP BY子句没有包含a.aname和p.pname列，他们就会发出抱怨。 ■

注意GROUP BY子句要紧跟在WHERE子句的后面。求解子查询（没有UNION、INTERSECT或EXCEPT操作）的概念性步骤如下：

- 首先，对FROM子句中的所有表做笛卡儿积。
- 接着，删除不满足WHERE子句的行。
- 然后，根据GROUP BY子句对剩余的行进行分组。
- 最后，求出选择列表中的表达式的值。

正如我们在例3.3.6和例3.4.6后面的讨论中提到的那样，上述求解子查询的概念性步骤可能与一个数据库产品执行Select语句的实际步骤大相径庭。回顾前面对空值的讨论：集合函数忽略了所有的空值；空值不能通过搜索条件中的所有相等或不等测试——甚至对一个空值等于另一个空值这样的测试，也是如此。但是，出现在作为GROUP BY对象的列上的空值确实会导致相应的行被分在同一组里。我们将在本章结尾部分的习题中进一步探讨这一问题。

如果要从包含GROUP BY子句的Select语句的结果中去掉某些行，比如，当选择列表中诸如sum(qty)之类的合计值太小时就去除相应的结果行，我们不可能通过在WHERE子句设置约束条件来实现这一目标。

```
select pid, sum(qty) from orders -- ** INVALID SQL SYNTAX
  where sum(qty) > 1000
  group by pid;
```

首先，一个集合函数不能出现在WHERE子句中，除非它是在子查询的选择列表中（参见例3.7.5）。更重要的是，我们刚说过在GROUP BY子句执行分组操作以前WHERE子句在理论上已删除了所有不满足搜索条件的行，这意味着WHERE子句中的条件不可能顾及在选择列表

中的合计值，因为这些合计值取决于实际的分组情况。为了能创建基于分组情况的条件，大多数标准都为SQL Select语句提供了一种新的约束子句——HAVING子句，它的求解过程发生在GROUP BY操作之后。

例3.8.3 当某个代理商所订购的某样产品的总量超过了1000时，打印出所有满足条件的产品和代理商的ID以及这个总量。

```
select pid, aid, sum(qty) as TOTAL from orders
  group by pid, aid
  having sum(qty) > 1000;
```

注意，HAVING子句的操作紧跟在GROUP BY子句的操作之后但先于对选择列表中的表达式的计算。该查询打印出查询[3.8.1]的结果表中最右边那一列超过1000的所有行：

pid	aid	TOTAL
p01	a01	3000
p01	a06	1800
p05	a03	2400

■

在Select语句中，HAVING子句只能对与各个分组相对应的单值（也就是能合法地出现在选择序列中的值）进行测试。图3-13所列的子查询通用形式是子查询的最终形式；也就是说，在子查询中不再会出现新定义的子句。然而，要实现完整的Select语句，我们还需要定义一个新的子句。

<pre>subquery ::=* SELECT [ALL DISTINCT] { * expr [[AS] c_alias] {, expr [[AS] c_alias]...} } FROM tableref {, tableref...} [WHERE search_condition] [GROUP BY colname {, colname...}] [HAVING search_condition]; subquery UNION [ALL] subquery</pre>

图3-13 基本SQL子查询的通用形式（作为Select语句也是可以的）

在上面的基本SQL子查询形式中，FROM子句中的tableref专指3.6节中图3-11第一行的简单形式：

```
tableref ::= tablename [[AS] corr_name]
```

我们的基本SQL通用形式没有引用图3-11所列的tableref语法的通用形式。此外，还要注意基本SQL只包含了UNION [ALL]运算符而没有包含INTERSECT和EXCEPT。尽管例3.3.7和3.7.6已经用两种完全不同的Select语句解决了同一问题，这里的HAVING子句又为我们提供了一种解决该问题的新方法。

例3.8.4 求出被至少两个顾客订购的所有产品的pid值。

```
select pid from orders
  group by pid
  having count(distinct cid) >= 2;
```

在上面的Select语句中，引用cid是一项危险的操作，因为与根据pid值分得的每个组相对应的cid值不是单值。HAVING子句必须作用在能出现在选择列表中的值上，因为这些值必定是在分组过程中形成的单值。然而，对根据pid值分得的每个组来说，集合函数值COUNT(cid)是单值的，而且由于它能出现在选择列表中，所以它也可以出现在HAVING子句中。该查询的结果在下面的表中：

pid
p01
p03
p05
p07

注意，在一般情况下我们不会使用HAVING子句，除非有GROUP BY子句出现；如果GROUP BY子句被省略了，那么HAVING子句将把整个结果当成一个组来使用。这样，在例3.4.8中，如果没有group by pid语句但HAVING子句仍然存在，那么打印出来的结果将是一张空表，除非在orders表中至少有两个不同的cid值。

我们所介绍的基本SQL Select语句的通用形式不允许集合函数的嵌套，比如，我们不能把AVG函数应用在由MAX元素组成的集合上。尽管如此，3.6节所介绍的通用语法已提供了一种实现嵌套的方法。

例3.8.5 构造一个查询来求出所有代理商的最大销售额平均值。在基本SQL中，把一个集合函数放入另一个集合函数的内部是行不通的，如下所示：

```
select avg(select max(dollars) from orders -- ** INVALID SQL SYNTAX
           group by aid);
```

该SQL语句之所以无效是因为基本SQL既不允许集合函数内部包含子查询也不允许FROM子句包含子查询。但如果使用了图3-11的扩展语法(参见3.6节)，我们就能将子查询放在FROM子句中(这无法用基本SQL来实现)。通过对表重新命名并对集合函数生成的列命名，我们能如愿以偿地实现上述查询，如下所示：

```
select avg(t.x) from (select aid, max(dollars) as x -- ** ADVANCED SQL
                      from orders group by aid) t;
```

这里的FROM子句将子查询的结果表重命名为t(这也是必要的)，并在选择列表中将检索到的聚集列命名为x。这样，外层的Select语句就能对t.x求平均。该查询在ORACLE和DB2 UDB中是可行的。上述子查询的结果是下面的表：

aid	max(dollars)
a01	500.00
a02	180.00
a03	1104.00
a04	450.00
a05	720.00
a06	540.00

外层的Select语句将返回这些值的平均值582.33。 ■

3.9 SQL Select语句的完整描述

图3-14列出了基本SQL Select语句的通用形式，其中的新语法元素都会在后面的段落中得到定义。所有的SQL语法都将在本节结束以前介绍完毕。

```

subquery ::=

    SELECT [ALL | DISTINCT] ( * | expr [{AS} c_alias] [, expr [{AS} c_alias]...] )
        FROM tableref [, tableref...]
        [WHERE search condition]
        [GROUP BY colname [, colname...]]
        [HAVING search_condition]
    | subquery UNION [ALL] subquery

Select statement ::=

    Subquery [ORDER BY result column [ASC | DESC] [, result column [ASC | DESC]...]]
```

图3-14 基本SQL子查询和Select语句的通用形式

如图3-14所示，**ORDER BY**子句不能出现在子查询形式中，而只能出现在完整的Select语句中。**ORDER BY**是新加入的子句，它允许我们将最后的结果行根据出现在选择列表中的一个或多个结果列(result_columns)(列名或列别名)排序。当选择列表所指定的结果列多于一个时，结果行首先根据排在最前面的结果列来排序，仅当根据最先的j个结果列排序所得的行序相同时，我们才需要考虑位于第j+1个位置上的结果列。为了提供更大的灵活性，SQL允许在**ORDER BY**子句中用列号1到n来指定结果列，其中，n是出现在选择列表中的列的数目。列号选项是针对结果列没有有效列名这类情况而设立的。比如，计算一个表达式之后没有为其创建列别名就会导致这种情形。产生这种情况的另一种可能性是Select语句是许多不同子查询的并，因为我们不能假定在不同子查询的选择列表中彼此对应的列都具有相同的限定名。在[ASC | DESC]选项中(升序或降序)，ASC是缺省的选择，这意味着值越小的行将越早地出现在结果表中。

注意，如果出现在作为**ORDER BY**对象的列上的值为空，那相应的结果行将被放到相同的排序位置上，或大(“高”)于或小(“低”)于在该列上具有非空值的所有行。空值的确切排序位置视产品而定，因为各种标准都没有对此作出规定；所以，在ORACLE和DB2 UDB中，空值被排在“高”处，但包括INFORMIX在内的其他一些产品都把空值排在“低”处。当然，在**WHERE**子句中对某个列应用**IS NOT NULL**谓词可以避免对该列值为空的行进行检索。

- ◆ 首先，对**FROM**子句中的所有表做关系乘积。
- ◆ 接着，删除不满足**WHERE**子句的行。
- ◆ 根据**GROUP BY**子句对剩余的行进行分组。
- ◆ 然后删除不满足**HAVING**子句的组。
- ◆ 求出**SELECT**子句选择列表中的表达式的值。
- ◆ 若有关键词**DISTINCT**存在，则删除重复的行。
- ◆ 求解子查询的**UNION**、**INTERSECT**和**EXCEPT**。
- ◆ 最后，若有**ORDER BY**子句存在，则对所有选出来的行进行排序。

图3-15 求解Select语句的概念性步骤

在图3-14所列的通用形式中，各子句在Select语句中的排列顺序可被看做求解过程的概念性顺序，我们可以从图3-15中看出这一点。新加入的步骤以这种顺序出现是非常合理的，因为，求解ORDER BY子句，即根据某些列值将结果行按照一定的顺序排列，显然是显示结果前的最后一步。我们要再一次提醒读者：这种求解过程的概念性步骤可能与查询优化器所选择的实际步骤有相当大的差别。

例3.9.1 列出所有的顾客、代理商以及与每个顾客-代理商对相对应的销售总额，并将结果按销售总额从大到小的顺序排列，最后仅保留那些销售总额至少有900.00的顾客-代理商对。

```
select c.cname, c.cid, a.aname, a.aid, sum(o.dollars) as casales
  from customers c, orders o, agents a
 where c.cid = o.cid and o.aid = a.aid
 group by c.cname, c.cid, a.aname, a.aid
 having sum(o.dollars) >= 900.00
 order by casales desc;
```

在ORACLE中，该查询返回下面的表：

cname	cid	aname	aid	casales
Allied	c003	Brown	a03	2208.00
TipTop	c001	Otasi	a05	1440.00
TipTop	c001	Smith	a01	900.00

注意，该Select语句结尾部分的句法：order by casales desc。这里使用关键词DESC是因为我们想把具有最大销售总额的结果行排在最前面。注意，在上述Select语句中使用形如order by sum(o.dollars)的ORDER BY子句在所有的数据库中都是行不通的。要使该Select语句在所有的数据库中都有效，ORDER BY子句的结果列必须由列号或者选择列表所包含的列名或别名来指定。 ■

现在是对构成搜索条件的基本语法对象进行更精确的定义的好时机。下面，我们将回顾前面已介绍过的一些概念，但这里要从更严格的角度来阐述。

1. 标识符

一般的SQL标识符是大小写无关的。实际上，SQL在使用标识符的时候都要把它们转化成大写形式。所以，键入select PID from Customers或者select PiD from CuStOmErS所得的结果是相同的。一个标识符必须以一个字母打头，之后则可以包含字母、数字或下划线。Entry SQL-92和Core SQL-99将一个标识符的字节数限制在18个以内。SQL在解释关键词（select、from等）以前也会把它们转化成大写形式。

如果要声明符合某种模式的实例或者使用某些特殊符号，那可以用另一种被称做定界标识符（delimited identifier）的标识符格式，Entry SQL-92和Core SQL-99都支持这种格式。这类标识符被双引号包围着，而双引号说明标识符名本身就是它所代表的内容，也就是说，必须按名字的字面意义来使用这类标识符。尽管一般的SQL标识符不能包含空格或其他特殊字符（除了下划线），但定界标识符却可以。本文不使用这种格式，但有一个地方使用它会显得特别方便，那就是在显示结果时用它来定义列别名，如下所示：

```
select sum(qty) as "Total quantity of product p01" from orders where pid = 'p01';
```

2. 表达式、谓词和搜索条件

搜索条件是WHERE子句中用于删除某些行以及HAVING子句中用于删除某些组的条件：就图3-15所列的求解步骤而言，一行能在步骤2得以保留或者一组能在步骤4得以保留当且仅当该行或该组使得相应的搜索条件等于TRUE(真)。

首先介绍被称做表达式(expr)的语法对象：数值表达式、串值表达式以及诸如日期值表达式之类的其他类型的表达式（如果某个产品支持的话），等等。我们称为expr ::= numexpr | strvexpr | dateexpr | ...。数值表达式和串值表达式是出现频率最高的表达式并且是基本SQL的一部分。表达式通常出现在搜索条件中。例如，在形如x.dollars > 100的属性值（某一行的）和常数的比较中，x.dollars和100都是简单表达式。下面定义的表达式也能出现在Select语句的选择列表中。numexpr是一个由常数、表属性、算术运算符、内置算术函数以及集合函数所组成的算术表达式。常数也被称做文字(literal)。图3-16a给出了数值表达式的递归定义。

数值表达式	例子
1. Val: 常量或变量	6, 7.00, :percent
2. 列名	dollars, price, percent
3. 限定符.列名	orders.dollars, p.price
4. 数值表达式算术运算符数值表达式	7.00 + p.price
5. (数值表达式)	(7.00 + p.price)
6. 数值函数(表达式)	sqrt(7 + a.percent), char_length(str)
7. 集合函数(数值表达式)	sum(p.price)
8. (返回单个值的子查询)	(select max(percent) from agents)
9. cast表达式	cast(substring(cid from 2 for 3) as integer)
10. case表达式	case when price > 1.00 then price else 0 end

图3-16a 数值表达式(numexpr)的递归定义

我们在图3-16a中看到了一些新的语法项。出现在后面三个由线分隔的框中的语法项还未在一些重要的产品中得到实现而且不是我们所指的基本SQL的一部分（这意味着它们是高级SQL的一部分）。不过，由于这些语法项是Core SQL-99的一部分，所以在不久的将来它们很可能出现在几个主要的数据库产品中。在图3-16a的第一行，: percent代表嵌入式SQL语句（在某个程序中运行的SQL语句）所能使用的一个程序变量。在变量名前面加冒号是为了告诉SQL的预编译程序该名字代表了一个程序变量。

图3-16a的第6行列出了两个数值函数（返回值是数值型的）：sqrt和char_length。我们还在后面列出了各种返回其他类型值的函数。使用数值型或字符型参数的函数并没有完全符合SQL-99标准，这类函数有时被称做标量函数以区别于集合函数。在Core SQL-99中，与串有关的函数是char_length、substring等等；这些函数在Entry SQL-92中是不做要求的。数据库产品通常把char_length称做length并称substring为substr。接收并返回数值的函数通常被称为数学函数，它们的特性在很大程度上与具体产品有关；在所有这些产品中，只有abs()和mod()被写进了SQL-99标准，而且只属于扩展功能。参见图3-17中的例子。

只有当图3-16a第8行括号中的子查询返回数值型的单值时，整个形式才算得上是数值表达式。初级SQL-92不要求这种表达式形式，而某些产品可能也不是在所有能使用表达式的地方都支持它。所以，它不在基本SQL中。我们将在3.19节之后的“作为表达式的标量子查询”部分进一步讨论这一问题，那里会列出WHERE子句形式的某些重要细节。

图3-16a第9行的cast表达式以一种类型的值为参数并把它显式地转化为另一种类型。这在C和Java中是通过写`(cast) value`来实现的，这里的`cast`是类型名。

除了产生的结果是单值以外，case表达式都使人联想到C语言或Java中的switch语句。它的通用形式如下：

```
CASE
  WHEN search_condition1 THEN result1
  WHEN search_condition2 THEN result2
  ...
  WHEN search_conditionN THEN resultN
  ELSE result(N+1)
END
```

CAST和CASE都还不是基本SQL的一部分，但它们已变得相当普及并且是Core SQL-99的一部分。注意，值NULL并不能算表达式，但在一些使用表达式的地方也可以使用值NULL。同图3-16a对numexpr定义一样，图3-16b定义了一个串值表达式strvexpr。

串值表达式	例子
1. Val: 常量或变量	'Boston', 'TipTop', :cityname
2. 列名	cid, a.name
3. 限定符.列名	orders.cid, a.city
4. 串值表达式 运算符 串值表达式	a.cid 'Boston' (concatenate two strings with) (o.cid 'Boston')
5. (串值表达式)	substring(o.cid 'Boston' from 7 for 4) (= 'ston')
6. 串值函数(表达式)	max(o.cid)
7. 集合函数(串值表达式)	(select max(city) from agents)
8. (返回单个值的子查询)	cast(o.qty as char(10))
9. cast 表达式	case when city > 'Atl' then city else 'Atlanta' end
10. case 表达式	

图3-16b 串值表达式的递归定义 (strvexpr)

图3-16b第4行的串接运算符（||）在Full SQL-92和Core SQL-99以前还不属于任何标准，但它已在数据库产品中得到了广泛的实现。图3-17列出了由本书所涉及的数据库产品提供的一些数学函数。参见本章结尾部分列出的特定产品文档以获得更完整的函数列表（通常被编录在标题“函数”下）。

除了图3-17所列的函数以外，ORACLE、INFORMIX和DB2 UDB还实现了三角函数、指数函数、对数函数、幂函数和round(n)。在其他函数中，ORACLE和DB2 UDB实现了ceil(n)、floor(n)和sign(n)。图3-18分别列出了标准的以及与特定产品有关的串处理函数。

名称	描述	结果的数据类型
abs(n)	n 的绝对值, n 是数值型的	所有数值类型
mod(n, b)	n 被b 除以后得到的余数, n 和b 均为整数	整型
sqrt(n)	n 的平方根, n是整数或浮点数	浮点型

图3-17 ORACLE、INFORMIX和DB2 UDB中的一些数学函数

Core SQL-99 中的描述和形式	ORACLE	DB2 UDB	INFORMIX
返回串的长度(整数个字符) CHAR_LENGTH(str)	length(str)	length(str)	length(str), char_length(str)
返回串str的子串, 从第m个字符到结束(或者长度n), SUBSTRING(str FROM m FOR n)	substr(str,m [,n])	substr(str,m [,n])	substr(str,m [,n]); substring(str from m for n)
返回将str去掉左端或右端之后得到的包含空格(或者任意的字符集)的串, TRIM(['LEADING' '.TRAILING' BOTH '[set]FROM]str)	trim([leading trailing both] [set] from str), ltrim(str [,set]), rtrim(str [,set])	ltrim(str), rtrim(str)	trim([leading trailing both] [set] from str)
返回串str2在str1中出现的起始位置(整数), 对str1从头开始搜索或者如果指定了位置n, 从第n个位置开始搜索, POSITION(str1 IN str2)	instr (str1,str2 [,n])	posstr(str1, str2 [,n])	
返回由小写字母组成的串, LOWER(str)	lower(str)	lcase(str)	lower(str)
返回由大写字母组成的串, UPPER(str)	upper(str)	ucase(str)	upper(str)

图3-18 一些标准的以及与特定产品有关的串处理函数

图3-18中的串处理函数在不同的产品中具有不同的名字但实现的功能却大体相同。可能在不久的将来, Core SQL-99 的形式将出现在大多数产品中。在所有这些函数中, 有两个函数将返回整数, 这意味着它们是数值函数(图3-16a中的数值函数); 剩余的函数都是串值函数(图3-16b中的串值函数)。INFORMIX提供了一种被称做subscripting的求子串运算以及一个求子串函数。例如, 为了得到由c.cid的最后三个(从位置2到位置4)字符组成的子串, 我们使用c.cid[2,4]。除了表3-18所列的函数以外, ORACLE、DB2 UDB和INFORMIX还实现了在串中替换子串以及为子串填充空白符或其他字符的函数。

我们的基本SQL标准(以及Core SQL-99标准)包含了七种谓词, 它们是逻辑子句的最简形式。图3-19列出了这些谓词的形式以及相应的例子。在GROUP BY或HAVING子句中, 它们表现出来的值为TRUE(T)、FALSE(F)或UNKNOWN(U)。我们将简要地说明引入UNKNOWN值的动因。

谓词	形式	例子
比较谓词	expr1 θ {expr2 (Subquery)}	p.price > (Subquery)
BETWEEN谓词	expr1 [NOT] BETWEEN expr2 and expr3	c.discount between 10.0 and 12.0
量化谓词	expr θ [ALL ANY] (Subquery)	c.discount >= all (Subquery)
IN谓词	expr [NOT] IN (Subquery) (参见例3.4.6中的变体形式)	pid in (select pid from orders) (cid, aid) in (select cid, aid...)
	expr [NOT] IN (val {, val...})	city in ('New York', 'Duluth')
EXISTS谓词	[NOT] EXISTS (Subquery)	exists (select * ...)
IS NULL谓词	colname IS [NOT] NULL	c.discount is null
LIKE谓词	colname [NOT] LIKE val [ESCAPE val]	cname like 'A%'

图3-19 对所有产品（也就是对基本SQL）都有效的标准谓词

这些谓词中的大多数我们已在以前遇到过，新的谓词会在后面得到介绍。基本SQL允许在图3-19中任何出现expr的地方使用NULL。但如例3.7.8所示，NULL往往会产生一些意想不到的后果。前面我们说过，NULL本身并不算表达式。有关表达式（数值类型和串值类型）的定义参见图3-16a和3-16b。这两张图的第一行都对图3-19中的val做了定义，即常数或者来自嵌入式程序（包含了嵌入式SQL语句的程序）的变量（第5章的内容）。

在某些情况下，什么才算表达式取决于数据库系统是否实现了图3-16a和3-16b中的第8、9和10行所示的新功能，即返回单值的子查询、CAST以及CASE。只需做一个实验或查阅一下SQL手册就可以确定数据库系统是否支持CAST或CASE。比较难检验的是子查询。

3. 作为表达式的标量子查询：高级SQL

标量子查询是返回值为单值而不是由多行或多列组成的集合的子查询。如图3-16a和3-16b所示，高级SQL（以及Core SQL-99，但还不是基本SQL）把子查询当作一种正式的表达式构造部件，我们正期待着能以多种方式来使用这种新的功能。

- 目前，DB2 UDB、INFORMIX和ORACLE允许将标量子查询用作选择列表中的表达式，例如：

```
select cid, (select max(qty) from orders o where o.cid = c.cid)
  from customers c;
```

- 基本SQL不允许将标量子查询用作WHERE子句中的表达式，例如：

```
select cid from customers
  where (select max(qty) from orders o where o.cid = c.cid) > 100;
```

注意，图3-19中其余六个出现expr地方，它们的用法是类似的。目前的DB2 UDB和ORACLE都允许上面的第二种形式。最后要注意，将这里的高级SQL表达式语法与图3-11中的高级SQL tableref语法作比较之后，我们会发现：尽管tableref语法也涉及到了（Subquery）这种形式，但那里的子查询可以返回完整的行集，“导出表”。

有了图3-19的谓词以后，我们就能递归地定义搜索条件了，如图3-20所示。

搜索条件	例子
谓词	<code>o.pid = 'p01'. exists (Subquery)</code>
搜索条件	<code>(o.pid = 'p01')</code>
NOT 搜索条件	<code>not exists (Subquery)</code>
搜索条件 AND 搜索条件	<code>not (o.pid = 'p01') and o.cid = 'c001'</code>
搜索条件 OR 搜索条件	<code>not (o.pid = 'p01') or o.cid = 'c001'</code>

图3-20 搜索条件(search_condition)的递归定义

到目前为止，我们已讲述了SQL的所有谓词以及以这些谓词为构造部件的逻辑搜索条件。待我们完全解释了这些谓词的含义之后，创建任何可能的搜索条件应该不再是一件难事。

4. 基本SQL与高级SQL：总结

在这一章里，我们定义了基本SQL——一种出现在绝大多数重要DBMS产品（包括ORACLE、DB2 UDB和INFORMIX）中的SQL语法。基本SQL与Entry SQL-92和X/Open版本2标准的关系相当密切（参见本章结尾部分的“推荐读物”）。Core SQL-99标准将Entry SQL-92扩展到了略微超过我们所定义的基本SQL的层次上，这为数据库生产厂商将来要实现的功能提供了指南，但我们的指导原则是当前的可用性。基本SQL包括了本章所介绍的除高级SQL功能以外的所有SQL语法；这些特征是SQL-99标准的一部分，它们可能不在Core SQL-99中，但已被一些生产厂商采用。我们已在3.6节“高级SQL语法”、图3-16a和3.16b中的被横线分隔的方框以及上一小节“作为表达式的标量子查询：高级SQL”中介绍了高级SQL语法。如果要查阅有关语法扩展的更多信息，可参见附录C。基本SQL在产品之间有相当大的可移植性：唯一需要我们查阅产品文档的语法变化只有以不同方式命名的串操作函数（图3-18）以及空值的排放位置（大多数产品将空值排在高位而某些产品则将它排放在低位）。用于描述语法元素的术语在不同产品之间的差异，如“条件”与“谓词”等，比这些产品的实际语法之间的差别要明显得多。

5. 关于谓词的讨论

(1) 比较谓词

比较谓词的形式为：

`expr1 θ (expr2 | (Subquery))`

其中的 θ 是集合 $\{=, <, >, >=, <, <=\}$ 中的一个元素。注意，在绝大多数数据库系统产品中，不等($<$)也可用($!=$)或($^=$)来表示，但我们相信($<$)是最通用的形式。 θ 的右边可以出现子查询形式仅当该子查询检索到的结果是单值或空集。Core SQL-99把这种形式推广成`expr1θ expr2`，其中的两个表达式都可以包含由括号包围的子查询（除子查询外可能还会有其他一些表达式元素）。不过，在写这本书的时候还很少有数据库产品实现了这种推广形式。图3-19所列的其他子查询形式在SQL-99中没有得到推广。

例3.9.2 回顾例3.7.5：列出了折扣值小于最大折扣值的所有顾客的cid值。我们可以使用查询：

```
select cid from customers
where discnt < (select max(discnt) from customers);
```

因为子查询只检索到一个值。我们也可以使用谓词`<ANY`或`<ALL`。

如果比较谓词右边的子查询的结果是一个空集，那么以`expr1 θ(Subquery)`这种形式出现的比较谓词将等于`UNKNOWN(U)`。如果在比较谓词的任何一边（或者说两边）出现了空值，那么也会导致`UNKNOWN`的结果，我们已在例3.7.8的Select语句中看到了这一点：

```
select * from customers where discnt <= 10 or discnt > 10;
```

在其中的`customers`表中，有一行在`discnt`列上的值为空。我们将在下一节里解释引入布尔值`UNKNOWN`的动因。

(2) 真值：`TRUE(T)`、`FALSE(F)`和`UNKNOWN(U)`

在`WHERE`子句中进行测试的特定行（或在`HAVING`子句中的组，但后面都假定在`WHERE`子句中）可能使谓词等于`UNKNOWN`，这基本上意味着在测试该行的时候产生了空值或空的子查询。所以，如果该谓词构成了整个搜索条件，那么说明提交该查询的人可能不希望这一行被检索到。比如：如果有一个Select语句：

```
select * from customers where discnt < (Subquery);
```

而且其中的子查询将返回一个空集或者单个空值，那么我们不可能检索到任何行。由于一行能被检索到当且仅当它使得搜索条件等于`TRUE`，所以，从这个意义上来说，新引入的真值`UNKNOWN`就等价于`FALSE`。

然而，`UNKNOWN`不可能在所有的情况下都等价于`FALSE`。考虑由值为`UNKNOWN`的谓词的逻辑混合运算构成的搜索条件。例如，如果我们把上面提到的Select语句改成：

```
select * from customers where not (discnt < (Subquery));
```

并且谓词`discnt < (Subquery)`的结果仍然为`UNKNOWN`，想一想该语句会有什么结果？我们认为`not (discnt < (Subquery))`等价于`discnt >= (Subquery)`，而`discnt >= (Subquery)`理所当然应该等于`UNKNOWN`，因为`discnt < (Subquery)`等于`UNKNOWN`。这正说明了为什么需要`UNKNOWN`值：因为如果当子查询返回一个空集时谓词`discnt < (Subquery)`等于`FALSE`，那么根据正常的逻辑规则，`not (discnt < (Subquery))`将等于`TRUE`！这不是我们所希望的结果。于是，一个新的真值`UNKNOWN`便诞生了，并且具有性质`not(UNKNOWN) = UNKNOWN`。图3-21列出了在逻辑运算中的三个真值——`TRUE(T)`、`FALSE(F)`和`UNKNOWN(U)`的完整运算规则。

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

NOT	
T	F
F	T
U	U

图3-21 `UNKNOWN`值在逻辑运算中的表现

在这一节里，我们说明了在计算各种谓词的过程中产生真值`UNKNOWN`的情况。这在某些Select语句中会有相当惊人的效果，我们将在本章结尾部分的习题中探讨这一问题。

(3) BETWEEN谓词

`BETWEEN`谓词测试一个值是否在由另外两个值限定的范围内。它的形式为：

```
expr1 [NOT] BETWEEN expr2 and expr3
```

其含义（不考虑NOT）实际上就好比我们写了：

```
expr2 <= expr1 and expr1 <= expr3
```

如果包含了关键词NOT，那当原先的值不在限定的范围内时谓词的结果值为TRUE。提供这种形式最初是因为使用BETWEEN 谓词的表达式比它的等价形式——由AND连接的两个比较谓词具有更高的计算效率。这条规则已不再适用于所有的产品，但每当必须声明一个具有两个端点的约束范围并且发生了从一个产品到另一个产品的可移植性问题时，出于性能的原因，我们应假定该规则成立。

(4) 量化比较谓词

以如下形式出现的量化谓词的含义已在3.4节中得到了介绍，但我们仍需对特殊的UNKNOWN值计算作出定义。

```
expr θ [SOME | ANY | ALL] (Subquery)
```

$\text{expr } \theta \text{ ALL } (\text{Subquery})$ 的结果为FALSE当且仅当比较的结果对子查询所返回的至少一个值为FALSE。由此可推得：整个谓词的值等于TRUE当且仅当比较 θ 的结果对检索到的每一个值均为TRUE或者子查询的结果是一个空集。然而，如果左边表达式的值为空或者子查询至少返回了一个空值，那最后的结果将是UNKNOWN。

当比较 θ 对检索到的至少一个值为TRUE时， $\text{expr } \theta \text{ ANY } (\text{Subquery})$ 的结果为TRUE；当子查询的结果是一个空集或者比较的结果对返回的每一个值均为FALSE时，该谓词的结果为FALSE。然而，如果左边表达式的值为空，或者如果在子查询所返回的值中有一个为空值并且比较的结果对其他所有值均为FALSE，那整个谓词的结果将是UNKNOWN。

下面，我们举例说明：当子查询的结果是一个空集时，为什么谓词 $\text{expr } \theta \text{ ALL } (\text{Subquery})$ 应等于TRUE。

例3.9.3 检索所有顾客的最大折扣值。显然，我们可以用下面的查询语句来回答这一问题：

```
select max(discnt) from customers;
```

但我们想要举例说明有关 $\theta \text{ ALL}$ 谓词的特点，所以使用如下语句。

```
select distinct discnt from customers c
  where discnt >= all (select discnt from customers d
    where d.cid <> c.cid);
```

用语言来表述就是：我们所要检索的行（在左边）的discnt值大于或等于customers表中其他所有行的discnt值（一个相关子查询）。如果customers表中只有一行，那么子查询将检索到一个空集。显然，我们想检索的正是这个唯一的discnt值，因为它就是最大的，这就意味着在子查询检索到空集的情况下搜索条件的谓词应等于TRUE。 ■

(5) IN 谓词

IN谓词具有如下形式：

```
expr [NOT] IN ((Subquery)|(val [, val...]))
```

我们在3.4节中介绍了它的用法。IN谓词等同于=ANY谓词。在例3.4.6中，这里的expr被推广成一个包含了多列的行值。val的定义在图3-16a和3-16b的第一行中。

(6) EXISTS谓词

EXISTS谓词具有如下形式：

[NOT] EXISTS (Subquery)

它只有当子查询的结果不是空集时才等于TRUE。3.4节介绍了该谓词的用法。在任何条件下该谓词都不会等于UNKNOWN。

(7) IS NULL谓词

例3.7.8所引入的IS NULL谓词具有如下形式：

```
colname IS [NOT] NULL
```

在任何条件下该谓词都不会等于UNKNOWN。

(8) LIKE谓词

LIKE是新引入的谓词。它的通用形式如下：

```
colname [NOT] LIKE val [ESCAPE val]
```

第一个val代表模式串，它通常由一般字符和特殊字符组成并且被引号包围着。（根据图3-16a和图3-16b中的val定义，它也可以是一个程序变量，但我们还没涉及到这类程序。）模式串为满足某种特征的字符串值构造了一个模板。下面列出了包括通配符在内的所有可用在模式串中的特殊字符：

模式串中的字符	含义
下划线(_)	任意单个字符的通配符
百分号(%)	包含零个或多个字符的任意序列的通配符
转义字符	用在需要按字面含义引用的字符之前(在下面说明)
所有其他字符	代表它们自己

例3.9.4 检索cname值以字母“A”打头的顾客的所有信息。我们输入

```
select * from customers where cname like 'A%';
```

它返回如下表：

cid	cname	city	discnt
c003	Allied	Dallas	8.00
c004	ACME	Duluth	8.00
c006	ACME	Kyoto	0.00

如果我们要在模式串中按照字面含义来引用特殊模式字符%和_，那该模式串必须包含一个转义（Escape）字符。使用LIKE谓词通用形式中的Escape子句，我们可以为每个Select语句定义不同的转义字符。当一个模式包含了转义字符时，紧随其后的字符就要按它的字面含义来使用。

例3.9.5 检索cname值的第三个字母不等于“%”的顾客的cid值。

```
select cid from customers where cname not like '_A%' escape '\';
```

注意最后的百分号表示允许出现由零个或多个任意字符组成的序列。

例3.9.6 检索cname值以“Tip_”打头并且后面跟着任意个字符的顾客的cid值。

```
select cid from customers where cname like 'TIP\_%' escape '\';
```

注意最后的百分号表示允许出现由零个或多个任意字符组成的序列。 ■

在LIKE谓词的形式中，如果左边的colname呈现出空值，那对当前行来说该谓词的结果是UNKNOWN。

注意，在一些数据库产品中，某个转义字符，比如+，是由系统来选定的。对所有的转义字符都适用的规则是：如果希望某个转义字符以其字面含义出现在模式串中，那么我们应在同一行里使用两次转义字符。这样，如果+是转义字符，那么_ _ ++就表示在任意两个字符后面紧跟着字符+的模式串。

例3.9.7 检索cname值以序列“ab\”打头的顾客的cid值。我们可以输入

```
select cid from customers where cname like 'ab\\%\'' escape '\\';
```

或者，我们可以选择一种不同的转义字符或根本不使用转义字符：

```
select cid from customers where cname like 'ab\%\'';
```

注意最后的百分号表示允许出现由零个或多个任意字符组成的序列。 ■

3.10 Insert、Update和Delete语句

我们用三种SQL语句——Insert、Update和Delete——来对已存在的表执行修改操作。Insert语句插入新的行，Update语句改变现有行的信息，而Delete语句则删除表中现有的行。这三种语句通常被统称为更新语句（update statement），因为它们都起到了更新表的作用。这里有混淆概念的危险，因为Update是其中一种语句的特定名称，而我们需要在可能引起混淆的地方小心地区分两个update所指的具体概念。比如：要清醒地认识到Update语句仅仅是更新语句集合中的一员。为了在一个特定的表上执行Update语句，当前的用户必须是表的创建者或是已被授予了update特权（update privilege）的用户。我们将在后面的章节中介绍授权的步骤。

1. Insert语句

SQL的Insert语句的作用是往现有的表中插入新的行。它具有图3-22所显示的通用形式。

Insert语句将新的行插入指定的表中。我们必须使用两种形式中的一种（用“或运算符”，|，来表示）：要么是VALUES形式，即插入具有特定值的行；要么是子查询形式，即插入由子查询（可能涉及多个不同的表）返回的所有行。

<pre>INSERT INTO tablename [(colname [, colname...])] (VALUES (expr NULL [, expr NULL...]) Subquery)</pre>
--

图3-22 基本SQL Insert语句的通用形式

例3.10.1 往orders表中加入具有特定值的行，将qty和dollars列设成空值。有两种方法可以实现这个插入操作。

```
insert into orders (ordno, month, cid, aid, pid)  
values (1107, 'aug', 'c006', 'a04', 'p01');
```

就该insert语句而言，在执行插入操作的时候，由于还不知道qty和dollars的值，所以我们既没有在列名中也没有在值列表中提到这两列，它们将缺省为空值。

```
insert into orders (ordno, month, cid, aid, pid, qty, dollars)
values (1107, 'aug', 'c006', 'a04', 'p01', null, null);
```

在该语句中，我们显式地给出qty和dollars的空值。 ■

例3.10.2 创建一个名为swcusts的表，它包含住在西南部的所有顾客，并往该表中插入所有来自Dallas或Austin的顾客。

```
create table swcusts (cid char(4) not null, cname varchar(13),
city varchar(20), discnt real); -- same as customers

insert into swcusts
select * from customers
where city in ('Dallas', 'Austin');
```

这个例子说明Insert语句不一定要包含接收新值的特定列的名字。省略了列名就相当于指定表中所有的列并且列的顺序与在Create Table语句中定义的顺序相同。如果像例3.10.1那样指定列名，那它们就不一定要按照那种顺序出现。注意，例3.10.2中的Insert语句，把跟在表名swcusts后面的子查询放入括号是一种错误的做法（在Entry SQL-92和许多产品中）。

这种由子查询来为Insert语句创建输入数据的方法大大增强了SQL的能力。在Insert语句中，只有一个表能接收新的行，但一个子查询却可以作用在任意数目的表上，只要它能产生列数以及每一列的类型均正确的数据来作为要插入的新行。

2. Update语句

SQL的Update语句用于改变表中现有行的信息。它具有图3-23所显示的基本SQL通用形式（这是Entry SQL-92形式的扩展）。

Update语句将表中所有满足搜索条件的行在指定列上的值替换成特定表达式的值。这里的表达式只能引用在表的某个特定行上当前正被修改的列值。

<pre>UPDATE tablename SET colname = {expr NULL (subquery)} [, colname = {expr NULL (subquery)}...] [WHERE search_condition];</pre>
--

图3-23 基本SQL Update语句的通用形式

例3.10.3 将所有住在New York的代理商从每次订货中赚得的佣金百分率提高10%。

```
update agents set percent = 1.1 * percent where city = 'New York';
```

利用其他表中的数据来判定当前表中哪些列需要修改可通过在搜索条件下使用子查询来实现。

例3.10.4 将所有订货总额超过\$1000的顾客所收到的discnt值增加10%。

```
update customers set discnt = 1.1 * discnt where cid in
(select cid from orders group by cid having sum(dollars) > 1000);
```

注意只有一个表可以作为Update语句的对象。有些数据库系统不允许在SET子句中使用限定属性名：

```
update agents set agents.percent = ... -- ** MAY BE INVALID
```

Entry SQL-92中的Update语句形式不提供SET子句里的子查询。这样，在Insert语句中可采用的一种方法——引用其他表来导出更新当前表所需的数据在这里是行不通的。然而，Full SQL-92标准、Core SQL-99标准以及ORACLE、INFORMIX和DB2 UDB这些产品都有针对该问题的Update语句扩展句法，所以我们将这项特性包括在我们的基本SQL通用形式之内。事实上，由于Core SQL-99将标量子查询归为一种通用的表达式形式，所以，把图3-23中的I(subquery)选项去掉之后所得的形式就是该语句在Core SQL-99中的描述形式。下面是一个使用该语法的例子。

例3.10.5 用customers表中最新的discnt值来更新例3.10.2所创建的swcuds表中各行的discnt值。

```
update swcuds set discnt = (select discnt from customers where cid = swcuds.cid);
```

3. Delete语句

SQL Delete语句将现有的行从表中删去。它具有图3-24所显示的基本SQL通用形式。

```
DELETE FROM tablename  
[WHERE search_condition];
```

图3-24 基本SQL Delete语句的通用形式

例3.10.6 删除住在New York的所有代理商。

```
delete from agents where city = 'New York';
```

同前面一样，Search_condition也可以包含子查询，这就允许我们利用其他表中的数据来判定哪些行要从当前表中删去。

例3.10.7 删除总订货金额小于\$600的所有代理商。

```
delete from agents where aid in  
(select aid from orders group by aid having sum(dollars) < 600);
```

3.11 Select语句的能力

过程性语言（procedural language）是指用该语言编写的程序应写明完成某项任务的有序指令序列。语句的顺序是很重要的，因为我们可以认为排在前面的语句具有长远的影响，即它是后面语句正确执行的先决条件。比如，如果希望建立一个循环来对一串输入的变量值V求和，我们首先会设定SUM = 0，然后当循环不断进行的时候语句SUM = SUM + V会逐渐地增加这个总和。这里至关重要的一点是把语句SUM = 0放在循环开始以前而不是循环中间的某个地方。为了获得正确的结果而必须按特定的顺序来使用一系列语句的特性可被认为是证明了语言的过程性（procedurality）。

相反，非过程性语言（non-procedural language）是指用该语言编写的程序直接描述了所期望的结果。这意味着我们必须指明期望做什么而不是怎么做才能实现目标，而且程序的各语句之间也不存在着需要程序员来考虑的隐含顺序。这种非过程性语言的一个典型例子是一个由屏幕菜单组成的界面，在这个界面上，用户通过选择一些选项来完成某项任务（选择的

顺序是无关紧要的)，然后按下执行(Execute)键。一些拥护关系模型的早期书作者将非过程性视为一个至关重要的目标。在那以前的数据库语言通常需要数据导航。作为一个简单的例子，考虑上面提到的对一串输入值求和的程序。如果将这些值视为来自一个文件，那当执行一个循环来求和时，在从头到尾读取数据的同时我们始终保持着一种游标——我们知道哪些值已被计入总和而哪些值还没有——所以每个程序语句都有一个预先存在的上下文环境。关系模型的拥护者们指出：许多潜在的数据库用户（比如金融专家、房产经纪人和律师）可能无法胜任通过编写循环程序来获取信息这项工作。如果预先编制的针对他们需求的某个程序没能提供他们想要的信息，那他们就只能自认倒霉了，更何况他们想要执行的许多查询很可能无法提前预知。这样的查询属于即席查询，即它们源于紧急的需求并且不可能用预先编好的程序来解决。

上面所说的目标就是要提供一种能处理即席请求的查询语言，提供给用户一种提出与数据有关的单个问题的能力。这个单个问题可能会相当复杂，但即便如此，查询语言也应避免这种需要程序员来处理的过程复杂性。符号逻辑的研究似乎与这一目标（设计数据库语言）相关，我们在第2章里提到，E·F·Codd指出：建立在一阶谓词演算（first-order predicate calculus）（符号逻辑的一种类型）基础上的关系代数具备了一种语言所拥有的全部能力。从这一结论出发，在确定一种标准的数据库语言的过程中产生了许多问题。其中的一个问题是该语言具有什么程度的非过程性——没有一种复杂的语言在这方面是完美无缺的，但总有一些语言会比另一些强一点。还有一个问题就是该语言具有多大的能力。基于Codd的结论，具备关系代数全部能力的语言应被定义为完备的或者关系完备的，而这也暗示了一种衡量语言的重要标准。尽管如此，我们仍会介绍关系完备的SQL语言所不具备的一些特性，而且有迹象表明非过程性的方法事实上削弱了该语言现有的能力。

1. 非过程性的Select语句

让我们从验证两种语言的相对非过程性开始。

例3.11.1 回顾例2.7.8推导出的关系代数查询，它能检索出所有订购了价格为\$0.50的产品的顾客的名字：

```
((ORDER ⚈ (PRODUCTS where price = 0.50)[pid]) ⚈ CUSTOMERS)[cname]
```



这种方法只用一个语句就完成了查询，因而比使用赋值语句(:=)来保存中间结果的方法似乎更具非过程性。然而，这一优点在某种程度上是一种错觉。该表达式在一个很重要的方面体现出了过程性，因为我们必须首先计算括号中的子表达式，甚至在没加括号的子表达式中，图2-6的优先权规则也限定了计算的先后次序。而且，用户显然也希望按这种顺序来计算。比如，(PRODUCTS where price = 0.50)在 [pid] 上的投影运算必须先于它与 CUSTOMERS 表的连接运算，因为我们并不要求两者在 city 列上的值也必须相等。

关系代数中有许多等价性规则，比如交换律和结合律，这些规则允许我们将一个表达式转化成另一种等价形式。因此，一个特定的严格加括号的关系代数查询不会妨碍查询优化器找出一种效率更高的变体形式。然而，毫无疑问，例3.11.1中的表达式隐含着精确的计算顺序，而且查询优化器将发现：只要查询是用加括号的形式来表达的，那从更高的层次上来考虑如何高效地实现查询目标将是一件很困难的事。含有嵌套括号的表达式与一个使用赋值语句来建立中间结果的语句序列几乎没有任何差别。换句话说，关系代数的过程性相当强。想一想该表达式与等价的SQL语句之间有什么差别。

例3.11.2 下面用SQL语句来解决同一问题——检索订购了价格为\$0.50的产品的顾客的名字：

```
select cname from customers c, orders x, products p
  where c.cid = x.cid and x.pid = p.pid and p.price = 0.50;
```

回顾图3-15中给出的求解Select语句的概念性步骤（在3.9节的开头部分），我们可以认为此例中的Select语句是按照下列步骤来执行的：步骤1生成FROM子句中的customers、orders和products表的关系乘积；然后，步骤2删除不满足WHERE子句搜索条件的行；最后，求解选择列表中的表达式（单个的cname值）并输出结果。

当然，这种概念性步骤仅仅是想象计算过程的一种方法，并不一定是数据库查询优化器实际决定采用的执行方法。特别地，查询优化器可以随意合并概念性步骤中的连续两个或多个步骤。比如，如前所述，计算机的访问策略可能不会一开始就为FROM子句中的三个表的乘积创建一张新表。相反，该访问策略可能会通过执行一个三重循环来读取分别来自三张表的三个独立的行并对每个新的三行组合测试WHERE谓词。这样，我们就把步骤1和2合并起来：执行创建关系乘积的循环；在存储一行以前先对该行测试WHERE子句搜索条件的真假性以除去不满足条件的行。当然，查询优化器也可能采用一种完全不同的策略来合并步骤1和2：一个一个地访问orders表中的所有行；利用cid和pid值的唯一性，根据连接条件的要求在某种高效的索引结构中“查找”customers表中在cid值上唯一与之匹配的行以及products表中在pid值上唯一与之匹配的行。

上面全部论述的关键在于在写一个SQL语句的时候，我们并没有对精确的计算顺序作出任何假定！一个Select语句仅仅是对要检索什么数据的一种说明，而不是描述如何检索数据的一个过程。图3-15所提供的概念性计算步骤仅仅是一种用户可以想象的而且总能奏效的步骤：对FROM子句中的表做乘积先于删除不满足WHERE子句搜索条件的行是因为搜索条件在乘积以前一般是没有意义的（参见例3.11.2中的谓词）；根据搜索条件来删除不符合条件的行必须发生在计算选择列表的结果之前。查询优化过程中的一项重要技术是：以一种不同于概念性步骤的方法来计算Select语句，在提高效率而不改变结果的前提下，合并甚至颠倒某些步骤。比如，在上述三重循环的方法中，如果对行的删除以连续行的形式进行，那么乘积的全部结果就会推后，所以这两步已被合并了起来。与这种方法相联系的一个重要想法是：一旦产生的行多得足以满足当前的需求，三重循环的过程能够暂停。比如，屏幕在显示了一整屏的输出行之后就暂停显示，要使后面的输出行继续显示出来，用户必须提出请求。这一过程体现了三个步骤的合并，而且如果用户决定在输出别的结果以前就终止显示，那还能节省大量的空间。在使用索引的方法中，我们根据orders表中的cid值来查找customers表中的一行，这就使得WHERE子句的删除操作先于FROM子句的乘积操作而且消除了访问customers表中每一行的需要。

我们刚才看到了非过程性的Select语句的一项重要优点。当提供了要检索什么数据的一种说明而不是描述如何操作的一个过程时，我们把决定具体访问策略的任务留给数据库系统。在理论上，数据库查询优化器完全有能力作出究竟如何进行数据导航以检索所需的信息这类复杂的决定。该决定通常能优于程序员所能作出的决定，这是因为程序员在某个特定的时刻写了一个查询之后便不再关注它（程序员的时间是相当宝贵的），所以由程序员所决定的访问策略不可能随时跟踪表在大小和索引等方面最新发展。另一方面，如果查询优化器能注意

到某些条件上的明显变化，一个未经修改的Select语句以后还可以被重新编译并且能生成一个不同的访问策略。我们将在后面有关索引和性能的章节中讨论更多关于查询优化的问题。

2. 图灵能力

学过“自动机理论”（用在正规的语言中）的读者可能遇到过图灵论题（Turing's thesis）这一概念。数学家图灵研究了一种概念性的机器，除了能用无限的磁带（或内存）来保存中间结果以外，在别的特性上它与现代计算机很相似。图灵使几乎每个人都相信这种能执行有限长度的简单过程性程序的机器具有执行任何可用特定（算法的）术语来描述的计算过程所必需的所有能力。所以，图灵机能计算出下一盘完美的国际象棋的全部走法（可能会花很长的时间）或者能在它访问过的数据上创建任何类型的报表，只要需要该报表的人能在有限的步骤里准确地描绘出创建该报表的方法即可。用特定的语言来实现这类计算的能力被称做计算的完备性（computational completeness），或者有时也被称做图灵能力（Turing power）。如果我们假定所有诸如C之类的现代编程语言都能使用任意大（无限）的寻址空间来存储中间结果，那它们都具有图灵能力。因此，图灵能力是测试语言能力的基准。

没有一种非过程性语言可能具备图灵能力。设想一下在菜单上选择选项的模型，由于可选择的选项是全部选项的任意子集，所以我们的选择最终只能局限于数量有限的几种可能组合中。有些选项还可能提供参数以扩大我们的选择范围（比如在ORDER BY子句中指定列名），但是除非该模型允许将任意长的顺序程序编入参数（这将背离我们的目标），否则，它不可能产生图灵能力。

缺乏图灵能力并不一定是一个很大的缺点。比如，人类的大脑并不真的具有图灵能力，因为它可能只有有限的储存中间结果的能力。但是，图灵能力是考虑SQL在查询数据库时能做什么以及当使用能访问同样数据的编程语言（比如C）时我们能做什么的出发点。在介绍嵌入式SQL的第4章里，在学习如何在一个程序中调用SQL的时候，我们会发现我们能写出具有这种特性的程序，而这是使用非过程性的SQL所无法实现的。如今，似乎只有极个别的数据库专业人员会对此持有异议。关系模型的早期拥护者们以此为据认为应该使非程序员获得通过交互式界而完成一切操作的能力。此建议看似合理，实践起来却困难重重——现在，几乎所有的数据访问都还是通过数据库应用程序来实现的。目前的公认看法是：SQL语句的价值在于它的灵活性和通用性，而这种灵活性和通用性的受益者与其说是终端用户还不如说是程序员。

3. 基本SQL Select语句的有限能力

下面，我们通过列举一些基本SQL所不能实现的查询例子来研究本章所介绍的基本SQL Select语句的局限性。这些例子之所以有价值是因为它们大都推动了新的对象-关系SQL（在第4章介绍）的发展。（同样，为了实现其中的一些查询，3.6节的一些高级SQL语法也对基本SQL的功能进行了扩展）

(1) SQL和非过程性集合函数

我们曾经说明了关系代数具有一阶谓词逻辑的全部能力。可能当我们意识到没有一个SQL集合函数属于关系代数时，这一点似乎就没有那么可怕了。（关系代数原本可以引入这些集合函数，但最终还是没有。）所以，在关系代数中我们无法回答这样的问题：金额超过\$500的订货记录有多少个？我们已为SQL增加了集合函数：sum、avg、max、min和count。还能想起别的来吗？

例3.11.3 中值 (median) 是一种统计平均值, 对某些统计数据 (比如每月的房价) 而言, 它通常是一种比SQL集合函数avg()所提供的均值更为稳定的度量。给定一个n个数的序列, a_1, a_2, \dots, a_n , 且对 $i = 1, \dots, n - 1$, $a_i \leq a_{i+1}$, median被定义成序列中的 a_k 值, 其中 $k = \text{FLOOR}((n+1)/2)$ 。由于median不属于SQL的集合函数, 所以我们不能写出下面这种Select语句:

```
select median(dollars) from orders; -- ** INVALID SQL SYNTAX
```

为了说明median函数的运算过程, 我们将图2-2中orders表的所有行的dollars值排成一个非递减的序列, 从而得到 {180.00, 450.00, 450.00, 450.00, 460.00, 500.00, 540.00, 540.00, 600.00, 704.00, 720.00, 720.00, 800.00, 880.00, 1104.00, 1104.00}。这里共有16个数据项, 所以median值就是编号为FLOOR((16 + 1)/2)的数据项, 即第8项, 也就是540.00。

事实上, 我们可以巧妙地运用允许子查询结果作为FROM表的高级SQL扩展语法来计算中值。参见习题3.18。 ■

其他常用的集合函数有mode和variance。一个数集的模 (mode) 是在该集合中最常出现的数, 而方差 (variance) 则是一种重要的统计度量, 即单个观察值与均值 (集合函数avg() 的结果) 差的平方均值。事实上, ORACLE产品确实提供了集合函数variance()。统计学家们通常将观察值的平方和视为二阶矩 (second moment)。我们也可以用二阶矩以及被称做一阶矩 (first moment) 的均值来表示方差。但是, 实际应用对集合函数的需求是无止境的, 因为三阶矩 (三次方的均值)、四阶矩 (四次方的均值) 以及更高的阶矩都在实际应用中有一定的用武之地。值得注意的是, 所有这些集合函数都很容易用能访问数据的程序来实现, 只需用一个循环来读取所有的值并执行适当的聚集操作, 比如平方和等。然而, 在非过程性的SQL模型中, 如果系统没有提供某个集合函数, 那么我们就不能求得该函数的结果。因此, 提供一种被称做用户定义函数的工具使用户能在SQL中创建新的集合函数这一想法就成为推动第4章要介绍的对象-关系SQL发展的动因之一。当然, 该特性在目前的大多数产品中还未得到实现。

我们的基本SQL所不具备的另一项特性是集合函数的嵌套。不过, 我们在例3.8.5中已指出该特性是3.6节图3-11所列的扩展语法的一部分并且已在ORACLE和DB2 UDB中得到了实现。

例3.11.4 构造一个查询以求出所有代理商每人销售总额的平均值。一种看似自然的方法 (即集合函数的嵌套) 如下所示:

```
select avg(select sum(dollars) from orders -- ** INVALID SQL SYNTAX
           group by aid);
```

不幸的是, 此SQL语句是无效的, 因为SQL不允许子查询出现在一个集合函数的内部。如果数据库系统支持将某个查询的结果用作可供检索的表的这种高级SQL特性, 那么我们可以用下面的语句来实现此查询:

```
select avg(totdollars) from
  (select sum(dollars) as totdollars from orders -- ** ADVANCED SQL SYNTAX
   group by aid) sumtab;
```

把上述语句中的子查询作用到图2-2的orders表上, 它产生如下表:

sum(dollars)
1400.00
180.00
4228.00
450.00
2144.00
1400.00

这些值的平均值为1633.67。 ■

在基本SQL中，确实有一种方法能求出所有代理商每人总销售额的平均值。由于此平均值等于代理商每人总销售额之和除以代理商的数目，并且由于代理商每人总销售额之和就是全部销售额之和，所以我们可用下面的语句来实现此查询：

```
select sum(dollars)/count(distinct aid) from orders;
```

然而，基本SQL不允许集合函数嵌套这一事实依旧存在。正因为这样，我们可以列举出大量无法由任何基本SQL Select语句实现的请求。比如，SQL无法求出代理商每人平均销售额的总和或是代理商中每人总销售额的最大值。我们将在下一章里看到如何用以过程性方式访问SQL数据的程序来实现这类查询。

(2) 非过程性报表

关系代数所不具备的另一个概念——GROUP BY子句，为SQL提供了创建报表的功能。但我们可以轻而易举地找出基本SQL所不能创建的报表。正如求中值的例子，高级SQL扩展语法确实提供了构造下例中的查询所需的能力。

例3.11.5 按下列要求创建一个报表：将全部的销售额根据金额大小分类。类别（我们称之为范围）划分如下：0.0~499.99, 500.00~999.99，依此类推。我们可能会写出如下所示的语句

```
select range-start, sum(dollars) from orders -- ** INVALID SQL SYNTAX
group by dollars in ranges from 0.00 in blocks of 500.00;
```

创建这样的报表是为了体现每个销售额类别对总销售额的贡献。销售经理可能希望以这种方式来确定哪一范围内的销售额最能体现基本的销售额水平。对图2-2中的orders表来说，该查询的结果为：

range-start	sum(dollars)
0.00	2390.00
500.00	5204.00
1000.00	2208.00

以范围0.00~499.99为例，落在此范围内的dollars值为450.00, 450.00, 180.00, 450.00, 460.00和400.00，合计2390.00。像这样的报表不可能由一个合法的基本SQL语句产生。我们可以用两种高级SQL功能来实现此查询：CAST表达式以及FROM子句包含子查询的能力。参见习题3.18和习题3.19。 ■

显然，这类报表可以由过程性的编程语言产生。事实上，我们可以想象出无数个无法用

SQL非过程性地创建但可以用程序来产生的报表形式，除了编写可访问数据库的过程性程序以外似乎就没有别的方法可以解决这类问题了。我们将在第5章里介绍具体的操作方法。

(3) 传递闭包

一阶谓词演算的一个显著缺点是它不能实现传递闭包(transitive closure)，考虑定义在节点集合{a, b, c, ...}上的有向图G；当有一条从a指向b的边时，我们称 $a \rightarrow b$ 。该图是传递的当且仅当下面的性质成立：对任何 $a \rightarrow b$ 且 $b \rightarrow c$ ，边 $a \rightarrow c$ 也存在。对于一个非传递的图G，我们可以添加边使得传递性对所有的节点三元组(a,b,c)都成立：若 $a \rightarrow b$ 且 $b \rightarrow c$ ，则添加边 $a \rightarrow c$ （如果它还不存在）。在做完这些以后，我们基本上可以说从a到c的任意条路径中必有一条是从a到c的边。这种添加边的过程所产生的结果被称做图G的传递闭包。这种想法有时会体现在真实的查询中。

例3.11.6 让我们用下面的Create Table语句来创建一个名为employees的表：

```
create table employees (eid char(5) not null, ename varchar(16),
mgrid char(5));
```

属性eid代表员工号，它是每行所代表的员工的唯一标识，而属性mgrid则代表该员工的顶头上司的员工号。假定我们已经导入employees表：

employees		
eid	ename	mgrid
e001	Jacqueline	null
e002	Frances	e001
e003	Jose	e001
e004	Deborah	e001
e005	Craig	e002
e006	Mark	e002
e007	Suzanne	e003
e008	Frank	e003
e009	Victor	e004
e010	Chumley	e007

这里要注意员工e010, Chumley是e007, Suzanne（经理）的直接下属，Suzanne是e003, Jose的直接下属，而Jose则是e001, Jacqueline的直接下属。Jacqueline的mgrid列上是一个空值因为她是总裁，没有上司。我们可以很轻松地写一个Select语句来列出任一特定员工（假设是一个经理，否则结果列表为空）的所有直接下属。比如，为了检索Jacqueline（员工号为e001）的所有直接下属，我们键入以下Select语句：

```
select e.eid from employees e where e.mgrid = 'e001';
```

该语句将列出员工e002, e003和e004。现在，我们也可以检索这些员工的所有直接下属：

```
select e.eid from employees e where e.mgrid in
(select f.eid from employees f where f.mgrid = 'e001');
```

同前面一样，该语句将列出满足下列条件的所有员工：该员工是子查询所返回的某个员工的直接下属。因此，我们将得到e005, e006, e007, e008和e009。员工e010不在其中是因为她是e007的直接下属而不是子查询所返回的任一员工的直接下属。单个SQL表达式都不能保证

完整地列出e001手下所有不同级别的员工（甚至3.6节的高级SQL也不能保证）。这样的查询应返回除e001以外的所有员工。 ■

如果把employees表表示成一张图：每一行对应一个节点以及一条从mgrid指向eid的边，那么在上例中我们可以用一个Select语句列出从任一特定eid（比如e001）经过一条边所能到达的所有员工；同样，我们可以检索出从任一特定eid经过恰好两条边所能到达的所有员工。我们正试图构造的查询将返回从一个特定eid经过任意条边所能到达的所有员工。如果该图具有传递闭包的性质，那么所有有路径可达的员工也必定可以由一条边被到达。在查询语言中构造这样一个查询的能力通常被称做传递闭包。

ORACLE产品已提供了两个可构造传递闭包查询的非标准Select子句——CONNECT BY和START WITH。检索以e001为上司的全体员工这一查询在ORACLE中具有如下形式：

```
select eid from employees
  connect by prior eid = mgrid
  start with eid = 'e001';
```

SQL-99也包含了支持传递查询的功能，本书对此不作介绍。

(4) 布尔条件的有限能力

在信息检索的某些领域里存在着一个普遍共识：SQL语言存在着严重的缺陷是因为用户在构造查询时受到了布尔条件的限制。Gerard Salton——文本检索领域里的创始人之一列举了大量的例子以说明布尔条件不能提供解决某些重大问题所需的能力。（参见[6]，《*Extended Boolean Retrieval Model, Fuzzy Set Extentions*》，10.4节。）

例3.11.7 假定有一个名为documents的表，每一行代表一个文本文档（比如期刊文章），主键为docid，每个文档都用一百个关键词（keyword值）来标识文章的主题（比如，magnetic resonance, superconductor, gallium arsenide）。正如我们在2.3节对第一范式的说明中所见到的，要实现这种联系还需要一张包含两个属性：keyword和docid的keywords表。图3-25给出了一个满足上述条件的简单例子。

如图3-25所示，关键词'integer'和'integral'出现在'Intro Math'文档中，关键词'SQL'出现在'Intro DB'文档中，而关键词'relation'则同时出现在这两个文档中。我们通常会有几十万个文档和几千个关键词，而且在这些文件中存在着大量的关键词重用现象。为了检索文档d12293的所有关键词，我们会用如下查询：

```
select k.keyword from keywords k where k.docid = 'd12293';
```

为了检索拥有关键词'Xyz1'的所有文档，我们会用如下查询：

```
select * from documents d where d.docid in
  (select k.docid from keywords k where k.keyword = 'Xyz1');
```

检索文件的一种常用方法是找出一串与我们想要的内容相关的关键词。检索包含特定集合{Xyz1, Xyz2, …, Xyz6}中的全部六个关键词的所有文件这一查询可用3.5节所介绍的FOR ALL查询类型来表达，形式如下：

```
select * from documents d          -- retrieve documents d
  where not exists
    (select * from keywords k      -- ... where no keyword
     where k.keyword in            -- ... in our list
```

```

('Xyz1','Xyz2','Xyz3',
 'Xyz4','Xyz5','Xyz6')
and not exists
(select * from keywords m      -- ... fails to equal
 where m.keyword = k.keyword   -- ... a keyword
 and m.docid = d.docid));    -- ... in the document d

```

documents			keywords	
docid	docname	docauthor	keyword	docid
d12272	Intro Math	Thomas	integer	d12272
d23753	Intro DB	Gray	integral	d12272
...	integrity	d23753
		
			relation	d12272
			relation	d23753
			SQL	d23753
		

图3-25 文档和关键字数据库

总之，我们选出任何满足下列条件的文档d：不存在这样一个关键词，它在我们的关键词列表中，但又不是文档d的关键词。对这样一个简单的思想来说，这是一个相对复杂（而且效率低下）的查询，但这还不是最糟糕的情况。更重要的是，如果没有一个文档包含了全部六个关键词，那情况会是怎样的呢？我们怎样才能构造一个查询来检索含有其中任意五个关键词的文档？或者任意四个？在SQL中没有这样的语法。实际上，检索含有其中任意五个关键词的所有文档需要六个查询语句，而检索含有其中任意四个关键词的所有文档则需要 $(6 \times 5)/2 = 15$ 个查询语句。

这里应注意的是，检索在一系列属性上符合用户期望的所有行这类请求对应一个模糊集合（fuzzy set）概念，这只是文档检索领域里的一项基础工作而已。不同的关键词经常根据其重要性而带上不同的权值，查询条件可能也会被加权，而不同的k维距离测量方法则被用来检索匹配程度最高的一组文档。用户有权请求特定数量的文档并且要求返回的文档按照匹配程度从高到低的顺序排列。此外，有人提议并初步开发了为用户带来极大方便的界面，它替用户提交查询并让用户对已返回的文档与他们期望之间的接近程度做出评价，系统对用户的评价进行统计分析从而形成一定的反馈信息并根据它来修改查询中的权值。布尔型的搜索条件由于其自身的局限性因而不能表现出上述的大多数特性。我们仅在熟悉的CAP数据库环境下考察其中的一个特性：检索特定数量的文档并让它们按照匹配程度的降序排列。

例 3.11.8 在标准的CAP数据库中，假定代理商的数目非常庞大，而我们想打印出总销售额排名前20位的代理商名单以便奖励他们去夏威夷度假。这一请求无法用SQL语句来实现。当然，按照总销售额从高到低的顺序来列出所有的代理商还是可能的。

```

select aid, sum(dollars) from orders x group by aid
order by 2 desc;

```

对于该查询的输出结果，我们可以在得到最初的20行之后便停止检索，这样就可以把结

果限制在我们所关心的代理商中。当然，这暗示着一个过程的执行。 ■

第5章将介绍如何创建程序来解决本节所引入的任何问题，当然，这里指的是我们能想出算法的那些问题。尽管如此，例3.11.7的模糊集合问题仍然颇具挑战性，因为关系模型实在不适合关键词检索。

推荐读物

Jim Melton（编订了SQL-92标准）和Alan R. Simon的文章[4]介绍了SQL-92标准。对X/Open标准的介绍在[8]中。对DB2 UDB SQL的介绍在推荐读物[1]和推荐读物[2]中。对INFORMIX SQL的介绍在推荐读物[3]中。对ORACLE 8 SQL的介绍在[5]中。有关文本检索的最佳参考读物是由Gerald Salton创作的推荐读物[6]。信息检索（包括文本检索）领域里的研究人员通常拒绝使用SQL语言，因为相对他们的需求而言SQL语言具有太多的限制而且效率低下。有关数据模型和原型数据库系统的许多奠基性的论文在推荐读物[7]中给出。

- [1] Don Chamberlin. *A Complete Guide to DB2 Universal Database*. San Francisco: Morgan Kaufmann, 1998.
- [2] *DB2 Universal Database SQL Reference Manual*, Version 6. IBM, 1999. Available at <http://www.ibm.com/db2>.
- [3] *INFORMIX Guide to SQL Syntax*. Version 9.2. Menlo Park, CA: Informix Press, 1999. <http://www.informix.com>.
- [4] Jim Melton and Alan R. Simon. *Understanding the New SQL: A Complete Guide*. San Francisco: Morgan Kaufmann, 1993.
- [5] *ORACLE 8 Server SQL Reference*. Volumes 1 and 2. Redwood Shores, CA: Oracle. <http://www.oracle.com>.
- [6] Gerald Salton. *Automatic Text Processing*. Reading, MA: Addison-Wesley, 1988.
- [7] Michael Stonebraker and Joseph M. Hellerstein, editors. *Readings in Database Systems*, 3rd ed. San Francisco: Morgan Kaufmann, 1998.
- [8] *Data Management: Structured Query Language (SQL)*. Version 2. Berkshire, UK: X/Open Company, Ltd., March 1996. Email: xospecs@xopen.co.uk.

习题

在本书后面“习题解答”中有答案的习题都用符号•来标记。

学生作业：我们建议学生们用如下方式来提交可执行的SQL语句：在适当的目录下创建SQL命令文件，文件的名字从Q1A到Q1U，依此类推。这些命令文件应包含正确的（测试过的）查询。然后，在脚本模式下显示并运行这些查询，该模式会把运行的结果存入一个可上交的文件中。注意前面的几个查询将返回数量庞大的结果行，在那种情况下，在上交结果文件以前，请删除其中的大部分结果行。

注意 习题3.1至3.7仅涉及到3.5节所介绍的SQL特征，这些特征提供了第2章的关系代数所具备的能力。

3.1 用SQL查询来回答第2章结尾部分的习题2.5 (a)到(u)的请求。

3.2 在下面的(a)和(b)中使用All谓词或Any谓词。

- (a)• 检索佣金百分率大于最小佣金百分率的代理商的aid值(列名: percent)。
- (b) 检索获得了最高佣金百分率的代理商的aid值。
- (c)• 解释为什么下面的查询无法回答上述请求(a), 尽管它从图2-1的agents表中检索到的结果行正好是(a)的正确答案: select aid from agents where a.percent > 5;

3.3 谓词IN和谓词=SOME具有相同的效果(在例3.4.8中有说明)。

- (a)• 假定上述结论成立, 解释为什么谓词NOT IN 和<>ANY(不等于任何一个)具有不同的效果, 相反, 与NOT IN具有相同效果的是<>ALL。
- (b) 运行例3.4.13中使用谓词NOT IN和<>ALL的两个查询。然后把<>ALL换成<>ANY, 运行替换后的查询并用语言表述检索到的结果是什么。
- (c)• 运行例3.4.7的查询, 它使用了谓词<=ALL。要恰好检索到使用<=ALL没有检索到的行, 你认为应该用哪个量化比较谓词来替换<>ALL? 通过运行来证明替换后的查询返回了正确的行。
- (d) 参见谓词<ANY和<ALL的定义并解释为什么这些谓词与子查询返回单个元素情况下的条件expr < (Subquery)中的<具有相同的含义。

3.4 (a)• 写出一个不含子查询的SQL语句来解决例3.4.1的问题。(FROM子句应引用与该查询相关的所有表格。)

(b) 我们总能像(a)那样避免子查询吗? 假定有一个具有属性 A_1, \dots, A_n 的表S和具有属性 B_1, \dots, B_m 的表T, 以及常数c和k, 当 $i = 1, 2, 3$ 时, A_i 和 B_i 来自同一个域, c、k与 A_2 (B_2) 来自同一个域。考虑查询

```
select A1, ..., An from S where A2 = k and
    A1 in (select B1 from T where B2 = c);
```

重写此Select语句, 要求得到相同的结果但不要用子查询。不要忘记必要时应对属性名进行限定。

(c)• 重复(b), 重写下面的查询, 但不要用子查询。

```
select A1, ..., An from S where A2 = k and
    A1 in (select B1 from T where B2 = c and B3 = S.A3);
```

3.5 [难] 求出满足条件的所有(cid, aid)对: 其中的顾客cid没有通过代理商aid订货。这可以由下面的Select语句来实现:

```
select cid, aid from customers c, agents a
    where not exists (select * from orders x
        where x.cid = c.cid and x.aid = a.aid);
```

用NOT IN谓词和一个子查询联用的形式来代替NOT EXISTS谓词形式可以实现此查询吗? 使用NOT IN谓词和多个子查询联用的形式可以吗? 解释你的答案并通过运行来展示所有的等价形式。

3.6 再看一下图3-3的伪码。

- (a)• 写出相应的伪码说明在不用嵌套循环的情况下如何求解习题3.4(b)的查询。你应该有两个独立的循环(而不是一个在另一个的内部)——第一个循环对应于子

查询，它将结果(A , values)放入值列表 L 中，而第二个循环则用一个谓词来测试值 A_i 是否在 L 中。我们不对伪码的形式作出严格的定义，但要尽量把意思表达清楚（参照图3-3的伪码）。

- (b) 解释为什么在用伪码表示习题3.4(c)的查询时不能避免嵌套循环。试着从该查询包含了相关子查询这一事实中寻找原因。

3.7 怎样才能说明SQL Select语句确实提供了关系代数所具备的全部能力？回顾2.8节的内容：关系代数的所有八种运算（列在2.5节的末尾）都可以用五种基本运算——UNION、DIFFERENCE、PRODUCT、PROJECT和SELECT来表示。假定在SQL数据库中已存在两个由Create Table语句创建的被称做基本表的表R和S。

解释如何用SQL Select语句来检索下列任一结果表：

- (a)• RUNIONS
- (b) 关系代数中的 $R \text{ MINUS } S$, $R \text{ TIMES } S$, $R[\text{subset of columns}]$, $R \text{ WHERE } <\text{condition}>$ 。对子查询只能使用NOT EXISTS谓词。假定R和S在必要的时候具有一致的表头。
- (c)• [最难] 到现在为止，SQL Select语句的能力还没有全部展现出来，因为在关系代数中可能出现递归表达式，而在SQL Select语句中则不可能。比如，假定R、S和T是兼容的表并且都具有表头 $A_1 \cdots A_n$ ，解释怎样才能用SQL Select语句来表示关系代数表达式 $(R \text{ UNION } S) \text{ MINUS } T$ 。
- (d) [非常难；需要数学基础] 证明如果U和V代表由SQL Select语句在基本表上实现的任意深度的关系代数递归表达式，那么我们也能用SQL Select语句来实现更深的递归：

$$U \text{ UNION } V, U \text{ MINUS } V, U \text{ TIMES } V, U[\text{subset of columns}], U \text{ WHERE } <\text{condition}>$$
- (e)• [非常难] 回顾定理2.8.3： $R \text{ DIVIDEBY } S$ 可以由投影、差以及乘积来表示。令R代表SQL语句“select cid, aid from orders;”，S代表“select aid from agents where city = 'New York';”。这样， $R \text{ DIVIDEBY } S$ 将给出与例3.5.2相同的答案。利用公式2.8.3并用一个SQL Select语句来表示 $R \text{ DIVIDEBY } S$ ，然后运行该语句以确认它给出了正确的答案。

注意 下面三道习题需要3.6节所介绍的SQL特征，这些特征对关系代数（第2章的内容）所提供的能力进行了扩展但还没有得到大多数数据库产品的支持。一般说来，你只能用ORACLE版本8或更新版本来在线地执行这些习题，或者还可以用DB2 UDB。关于高级SQL的更多应用参见习题3.18和3.19。

- 3.8 (a)• 写一个不含WHERE子句的Select语句来检索所有顾客的cid值以及每个顾客在每样产品上的最大花费。将结果表中的列记为：cid, MAXSPENT。
 (b) 写一个查询来检索查询(a)中的MAXSPENT的平均值AVERAGE（对所有的顾客而言）。
 (c) 可以不用3.6节所介绍的新特性来解决(b)吗？

- 3.9 解释为什么引用3.6节所介绍的特性会使习题3.7的(d)的证明变得更容易（与递归有关）。(附加题：如果可以的话，证明该结论。)

- 3.10 (a)• 假定有一个由搜索条件WHERE C确定的customers表的小子集C（500 000个顾客记录中的10行）。另外，还有一个包含顾客订货记录的sporders表，其中的顾客与子集C大致（但可能不完全）相同。我们想打印一份报表：cid, cname, TOTDOLL。其中，TOTDOLL是sporders表中的每个顾客的总订货金额SUM(dollars)。该报表要反映出下面这种情况：在sporders表中出现的顾客不在子集C中（在cname列上显示一个空值即可）；同时，它不能包含在sporders表中没有订货记录的顾客（不管他是否在C中）（也就是说，要避免显示大约500 000 - 10个在TOTDOLL上为空值的顾客）。什么样的SQL语句能实现上述请求？
- (b) 列出子集C所包含的在sporders表中没有订货记录（TOTDOLL的值为空）的所有顾客。什么样的SQL语句可以实现该请求？

注意 后面的许多习题都需要在3.6节之后介绍的SQL功能，这些功能对关系代数（第2章的内容）所提供的能力进行了扩展并且在所有的主要数据库产品中都得到了广泛的应用。

- 3.11 用我们所提供的基本SQL标准（在3.6节之外）来编写能完成下列任务的SQL语句，然后运行这些语句。同时，想一想如何用3.6节的扩展功能来编写相应的SQL语句（不作硬性要求）。

- (a)• 为每个有订货记录的代理商列出他所订购的每样产品的pid值以及所有通过该代理商订购该产品的顾客们所订购的总量。
- (b) 如果A是orders表中所有满足cid = x且pid = y的行的avg(qty)，那我们称顾客x以平均量A订购了一个产品y。可以用一个SQL语句求出以至少300的平均量订购了他们所收到的每样产品的顾客的cid值吗？
- (c)• 求出没有为任何住在Duluth的顾客订购任何在Dallas生产的产品的代理商的aid值。
- (d) 求出为住在Duluth或Kyoto的所有顾客订购了至少一样公共产品的代理商的aid值。
- (e)• 求出只通过代理商a03或a05订货的顾客的cid值。
- (f) 求出被所有住在Dallas的顾客都订购了的产品的pid值。
- (g)• 使用集合函数max来找出拥有最高percent（佣金百分率）的代理商。
- (h) 在agents表中，删除名为Gray的代理商所在的行，以完整的形式打印出结果表，然后用Insert语句将Gray所在的行放回原表。
- (i)• 用Update语句把Gray的percent值改成11。然后再改回来。
- (j) 用一个Update语句把存放在Duluth或Dallas的所有产品的价格提高10%。然后，重新运行最初用于创建products表并导入数据的过程以恢复表中的原值。
- (k)• 写一个SQL查询来检索订购了至少一样产品但所有的产品都只通过代理商a04来订购的顾客的cid值。该查询还应在每个cid值所在的同一行上列出每个顾客的总订货金额。
- (l) 写一个SQL查询来求出为住在Duluth的所有顾客订货的代理商的aid和percent值。按照percent值从大到小的顺序排列结果中的aid值。（注意，

如果选择列表中没有包含percent，那就不能根据该列的值来排序。)

- (m)• 写一个SQL查询来求出满足下列条件的产品的pid值：该产品被至少一个顾客订购；每个顾客与为他订购该产品的代理商住在同一城市。

3.12 本题要求你在上机验证查询结果以前写下你认为SQL会给出的答案。这里的关键在于理解SQL得出这个答案的过程。

- (a)• 说明下列查询生成结果的过程，即列出所创建的每个子查询元素集合，然后写出你预想的最终答案。

```
select city from customers where discnt >=all
  (select discnt from customers where city = 'Duluth')
union select city from agents where percent >any
  (select percent from agents where city like 'N%');
```

- (b) 用文字说明SQL是怎样得出下列查询的结果的，然后写出你预想的答案。

```
select cid, pid, sum(qty) from orders
  where dollars >= 500.00
  group by cid, pid having count(qty) > 1;
```

3.13 本习题所研究的问题是：什么样的SQL查询在不使用关键词DISTINCT的情况下必定能返回没有重复行的结果表？这可能是一个很重要的问题，因为在不必要的时候使用关键词DISTINCT会导致资源的浪费。我们在下面列出不返回重复行的查询所遵循的一系列规则，并要求你做到：(i) 解释为什么每条规则都能奏效；(ii)对每条规则，用一个例子说明当违反该规则时查询可能返回重复行。在(a)和(b)中，我们仅考虑既不包含子查询也不带有GROUP BY子句的查询。

- (a)• 在FROM子句只包含一个表的查询中，如果选择列表中的列构成了该表的超键，那么返回的结果表就不包含重复行。
- (b) 在FROM子句包含多个表的查询中，如果对其中的每一个表，总能在选择列表中找出某个构成该表超键的列集，那么返回的结果表就不包含重复行。
- (c)• 没有一个包含GROUP BY子句的查询会返回重复行这一命题为真吗？如果为真，解释原因；否则，给出一个反例。
- (d) 在WHERE子句含有子查询的Select语句中，至少在类似上述(a)和(b)的情况下该查询能保证返回的行都是独一无二的。然而，习题3.4(b)中的形式转换（从子查询形式转换成不用子查询的连接形式）可能会导致重复行，除非在选择列表前面使用关键词DISTINCT。给出一个本来不返回重复行但在这种转换中产生了重复行的查询例子。

3.14 (a)• 重写例3.9.4的查询：不要用任何部分匹配模式的形式而仅用比较谓词。

- (b) 你能找出一个不能被其他谓词形式替代的模式吗？

3.15 回顾2.10节的外连接定义。考虑（一般的）连接查询：

```
select a.aname, a.aid, sum(x.dollars)
  from agents a,orders x
  where a.aid = x.aid group by a.aid, a.aname;
```

- (a)• 用基本SQL重写该查询以实现一个外连接（这需要对三个子查询进行并操作）。

即使一个代理商没有订购任何产品，他的`aname`和`aid`也应该出现；即使一个代理商在`agents`表中没有相对应的`aid`，他在`orders`表中的有关信息（根据`aid`值分了组）也应该出现；当列上不存在合适的值时，将`aname`或`sum(x.dollars)`的列值设成常数`null`（如果系统不允许使用空值，也可以使用空格和零）。

- (b) 为了检验(a)的答案，往`orders`表中插入一个新的行：(1027,'jun', 'c001', 'a07', 'p01', 1000, 450.00)；往`agents`表中插入一个新的行：('a08', 'Beowulf', 'London', 8)，然后运行该查询。当你对结果感到满意的时候，再删除刚才插入的那两行。

3.16 下面的习题说明了空值的一些特殊性质。为了完成该习题，我们往`agents`表中插入一些在`percent`列上为空值的新行（并在习题结束以后把它们从表中删去）。我们执行两个`Insert`语句：

```
insert into agents (aid, aname, city)
    values ('a07', 'Green', 'Geneva');
insert into agents (aid, aname, city)
    values ('a08', 'White', 'Newark');
```

- (a) 预测下列语句的结果，然后运行该语句验证你的答案。这里的问题是：在`GROUP BY`子句中，空值是如何运作的？

```
select percent from agents group by percent;
```

- (b) 预测下列语句的结果，然后运行该语句。这里的问题是：当与别的值一起排序时，空值是如何运作的？

```
select aid, percent from agents order by percent;
```

- (c) 考虑下面的查询：

```
select distinct aname from agents where percent >=all
    (select percent from agents where city = 'Geneva');
```

(i) 用文字来表述该查询：“获得……代理商的名字”

(ii) 注意这里的子查询将返回单个空值。在这种情况下，对`agents`表中的所有`percent`值，`>ALL`谓词的结果均为未知。这样，你认为该查询的结果会是什么？运行该查询以验证你的想法。

- (d) 以同样的方式考虑下面的查询，并完成类似于(c)中(i)和(ii)的练习。

```
select distinct aname from agents a where not exists
    (select * from agents b
        where b.city = 'Geneva'
        and a.percent <= b.percent);
```

你是否认为该查询的效果等同于问题(c)的效果？但这里有一个惊人之处：由于唯一满足`b.city = 'Geneva'`的行具有性质`b.percent = null`，而所有其他`percent`值与空值的比较结果为`UNKNOWN`，所以这里的子查询将返回一个空集并且对所有代理商`a`，`NOT EXISTS`的结果均为`TRUE`。这显然不是(c)的结果。

3.17 仔细体会“`ALL`意味着没有反例”这句话。假设我们对例3.5.2进行了修改：把`New York`改成了`Los Angeles`。

(a) 现在的答案是什么?

(b) 解释原因。

3.18 在例3.11.5中, 我们看到规定的报表不可能由一个基本SQL语句生成。

(a) 说明如何用多个基本SQL语句(前面的语句创建后面语句所需的表)来生成规定的报表。首先, 创建宽度为500.00的块: 舍去dollars/500.00的小数部分从而得到一个整型的“桶号”。比如, $750.00/500.00 = 1.5$, 舍去小数部分后得到的桶号为1。接下来, 用Create Table语句创建一个存放桶号的表buckets, 它有一个名为bucket的整数列。以三种最常用的数据库产品为例: 在ORACLE中, 可以用floor()函数来对任一数字的小数部分进行四舍五入从而得到一个整数; 在DB2 UDB和INFORMIX系统中, 可以用一个CAST表达式来把美元值转换成桶号。由于这些桶号在buckets中被具体化了, 所以你最终可以通过连接orders表和btab表来实现必要的分组。(注意有些系统在实现CAST的时候可能采用了四舍五入而不是下舍入。在那种情况下, 需要使用表达式: cast((dollars+250.005)/500.00-1 as integer), 额外的0.005确保了0美元被转换成桶号0; 500.00被转换成1; 依此类推。)

(b) 利用3.6节所介绍的高级SQL特征(在ORACLE和DB2 UDB中得到了实现)来写一个生成此报表的查询。这里的下舍入运算可以由floor()或CAST表达式来完成。你所需要的特殊的高级SQL特征是把一个子查询放在FROM子句中。

3.19 在例3.11.3中, 我们无法用一个基本SQL语句来完成中值(median)的计算。说明如何分别用多个基本SQL语句以及一个(很长的)高级SQL语句来计算中值。

(a) 使用多个基本SQL语句。创建表relhist(relative frequency histogram, 相对频率直方图), 它包含了dollars值以及这些值各自在orders表中的出现次数。例如, dollars = 720.00的行的出现次数为2。创建另一个表cumhist(cumulative frequency histogram, 累积频率直方图), 它包含了小于或等于每个特定dollars值的所有dollars值的出现次数。例如, dollars = 400.00的行的累积次数为2, 其中一次是对值400.00, 另一次是对值180.00。查询第二个表以求出中值。

(b) 利用高级SQL特征来写一个产生该报表的查询。

3.20 如前所述, ORACLE有能力执行传递闭包型的查询。创建例3.11.6所示的employees表, 然后运行该例后正文中的CONNECT BY查询。

第4章 对象-关系SQL

本章介绍数据库管理系统的对象-关系模型(ORDBMS)，并将其和我们前边的章节中介绍的老的关系DBMS (RDBMS) 模型相比较。ANSI/ISO SQL-99 标准提出了支持ORDBMS模型的SQL语言标准，我们介绍过的三个主要的数据库产品DB2 UDB, ORACLE和INFORMIX都支持数据库ORDBMS SQL语言（然而，这三个产品在一些基本语法元素上有些差异）。尽管数据库厂商MICROSOFT和SYBASE在本书撰写时尚未发布支持ORDBMS的产品，其产品却有朝这一方向发展的趋势。我们将在本章中对ORDBMS进行详细的介绍。由于向上兼容性，ORDBMS产品同样支持RDBMS SQL，因此，我们有时也将这些产品称为RDBMS/ORDBMS类的数据库产品。

尽管IBM的DB2 UDB已经具有强大的对象-关系编程界面，它对交互式的ORDBMS SQL特征的支持是在我们开始撰写本章之后不久，稍晚于其他数据库产品。鉴于此，在当前的章节中，我们仅涵盖ORACLE 和INFORMIX的ORDBMS特征。由于ORDBMS是一个全新的模型，在本书新版本发布之前，在这一领域内可能存在其他方面的变化，因此，我们推荐读者访问我们正文的互联网主页以获取关于ORDBMS和其他主题最新资料。主页的URL为<http://www.cs.umb.edu/~poneil/dbppp.html>。

4.1 引言

目前存在着几种不同的ORDBMS SQL “方言”。每个产品都有其自己独特的方言，甚至是SQL-99模型和语法在被采纳之前的最后几年也经历了巨大的变化。在我们的讨论中，为采用一般性的术语，我们将大多数ORDBMS SQL方言公用的概念表示为ORSQL。因此，我们使用的ORSQL术语未必对所有的ORDBMS产品都准确。

我们从对ORSQL的能力的总的看法来开始本节，包括多个ORSQL标准和方言以及我们在表述中如何处理这些ORSQL。

1. ORSQL的能力

ORSQL的对象-关系模型支持用户对象定义——我们习惯将ORSQL中定义的对象看做行，即一组可以存储在一个表中的某类型的数值。我们将ORACLE中的对象类型、INFORMIX中的行类型及DB2 UDB中的用户定义类型 (UTD) 称为行 (或对象) 及其类型的属性的格式。由于用户定义类型同样适用于SQL-99标准，一般我们也将在ORSQL中采用这一术语。这样，一个表可定义为包含用户定义类型的多个行 (或对象)。此外，表内的一列可定义为包含一个用户定义类型的值 (即一个列可以有像行一样的值)。

(1) 汇集类型(COLLECTION TYPE)

另外，对象-关系模型允许一个行包括一个行值汇集 (或一些其他汇集类型，如数组)。在一些系统中，单个列自身可拥有一个表，称为嵌套的表。在其他情况下，这样的对象汇集可转化为一个表，从而使SQL检索的概念也可作用于汇集。所有这些都违反了关系模型的第一范式规则。我们将2.3节中描述的第一范式规则重述如下。

规则 1 第一范式 在表的定义中，关系模型不允许列中出现形如多值域（有时称为重复域）或任何内部的数据结构（如记录）。满足这样的条件的表即是第一范式的表。

对象-关系模型允许包含多个值或结构化数据值，违反了第一范式规则。图4-1给出了一个对象-关系employees表的例子，其中，列dependents包含结构（两个属性dep_name和dep_age）并且每个行中包含多个值（如雇员001, John Smith有两个家属, Michael J. 和 Susan R.）。

employees				
eid	ename	position	dependents	
			dep_name	dep_age
e001	Smith, John	Agent	Michael J.	9
			Susan R.	7
e002	Andrews, David	Superintendent	David M. Jr.	10
e003	Jones, Franklin	Agent	Andrew K.	11
			Mark W.	9
			Louisa M.	4

图4-1 带有多个从属者属性值的雇员表

(2) 方法和UDF

在诸如Java的面向对象语言中，在一对对象中的私有数据仅能通过对象方法访问；这些方法是可被调用作用于特定对象的函数。通常地，对象的方法集合将处理对象能被孤立对待的所有特性。例如，对一个简单的银行账户对象，我们可能有方法get_balance()，make_deposit()和make_withdrawal()及创立和删除对象的方法。一些方法，如get_balance()，仅读取对象数据并返回信息给调用者。其他方法，如make_deposit()和make_withdrawal()，则修改对象中的数据。在纯面向对象的环境中（其中所有的数据都是私有的），由于没有其他方式可以操纵对象，一个对象类的方法的汇集刻画了该类（的行为）。

我们在本章中研究的对象总允许ORSQL直接访问对象的各个部分，因此，在ORDBMS中，所有的对象被看做是公共的而非私有的。在稍后的一些章节中，我们通过通用的ORSQL访问来处理对象。然后解释如何写对象方法，这些方法是“依附”于对象的函数，再说明如何编写更通用的用户定义函数（UDF），UDF函数不完全的“依附”于对象，但可以将多个不同的对象当作同等重要的参数。

2. 本章的表述形式

在本章中，我们将以两种商用产品ORACLE和INFORMIX为例介绍对象-关系的概念。这两个产品在设计和命名系统上有较大的区别，这种显著的差异使我们感到有必要对这两种产品分别加以介绍。鉴于此，我们打算在不同的小节中分别对这两种产品逐一介绍。例如，在4.2节“对象和表”中，我们给出了大量的基本定义和例子：4.2.1节覆盖“ORACLE对象类型”，4.2.2节则覆盖“INFORMIX对象行类型”。

一个非常重要的观点：我们建议读者通读本书包括两种产品的连续的章节，无论你是否对两种产品都感兴趣。因为在不同的小节中介绍的不同的产品的概念对正确理解对象-关系模型是十分重要的。由于对ORACLE的介绍覆盖了每一节，读者可以在不失连续性的情况下有意地跳过关于INFORMIX的小节，但仅研究一种产品将在一定程度上牺牲对对象-关系模型理解的广度。

3. 对象-关系的历史

有许多原因导致了对象-关系模型的形成和发展。在初期，对诸如Smalltalk和C++等面向对象语言的兴趣使得研究者们试图开发可以和这些语言自然地交互的数据库系统。80年代中期，大量的面向对象数据库系统（OODBS）开始出现。这些OODBS产品通过提供持久变量（这里的持久这个词与持久存储器有相同的含义）扩展了面向对象程序语言。其思想是在诸如C++的语言中，一个变量J首先被声明为“持久的”，然后可以在语言中以一种自然的方式被更新，如通过写一行诸如 $J=J+1$ 的程序代码。下层的OODBS将按如下方式处理变量J：当更新J的程序完成后，J的新值将被自动地储存在持久储存器中，并且，当这样的程序再次执行时，它将访问保存过的J值。在OODBS模型中，SQL语句不是必须的，数组和类（或封装的结构）等复杂对象可用于保存数据。这样的复杂对象尤其适用于诸如机器部件设计等应用场合，在机器部件设计中一些机器部件有带有数千个子部件的极为详细的装配部件。在关系数据库中构造这样的设计模型需要大量的不同类型的关系表间的连接运算，在编码时是很困难的并且执行这样的运算的效率将十分低下，而C++允许使用更复杂的结构，可以很自然地构造这样的模型。

尽管OODBS可能对于某些应用是最好的选择，但早期的OODBS产品没有“表”的概念，且查询也是十分简单的。尽管后来的OODBS产品对此做了很多的改进，但许多年来一直习惯于“填表”、“对一个空间返回一个答案”方式的关系模型的商业客户发现关系模型已经能够很好地满足他们的要求，不愿意再用对象模型重复实现他们的应用。结果使得OODBS产品（如来自Object Design、Versant、Gemstone和Objectivity等公司的产品）只占有很小的一部分市场份额，其销售额仅是RDBMS/ORDBMS产品的一部分。鉴于此，我们在本书中将集中介绍ORDBMS模型。

ORDBMS模型的发展满足了许多应用的需要。首先，一范式规则被认为是不必要的限制，大量的支持“嵌套关系”的非一范式的模型的大量的论文被发表，IBM当时支持对这种模型的商业研究，同时，许多研究人员开始开发这样的原型系统。商业用户最终将愿意接受这种对关系表的消除“对一个空间返回一个答案”的限制的革命性变化。早期最重要的对象-关系原型系统产品是加州大学伯克利分校的Michael Stonebraker提出POSTGRES系统。（关于POSTGRES系统和Michael Stonebraker对对象关系模型的论述请参阅本章末的“参考读物”。）POSTGRES系统后来被发展成为商业产品（由于版权的原因，这个系统加上了以Illustra结尾的产品系列名）。最后，在1996年INFORMIX购买了Illustra产品，并在此基础上开发了他们的ORDBMS产品。ORACLE公司则在1997年推出了ORACLE的对象-关系版，即版本8。ANSI/ISO SQL-99组织发布的SQL-99（官方称为SQL:1999）标准是一个对象-关系的标准。实际上，ANSI/ISO SQL-99组织在1994年就开始了对该标准的制定，但该标准的多次改动花了很多的时间，结果具有对象-关系特征的数据库产品早在1999年该标准发布之前就已经发布了。

4.2 对象和表

在ORACLE对象关系数据模型中，我们有一种新的用户定义类型，即对象类型。我们在4.2.1节中介绍新对象类型的使用。在INFORMIX中，ORACLE的对象类型被称为行类型，我们将在4.2.2节中介绍行类型。

4.2.1 ORACLE中的对象类型

我们首先说明如何创建一个对象类型。我们说一个对象类型有多个类型属性，与一个表

的列相似。ORACLE中的一个对象类型由Create Object Type(对象类型创建)语句生成，创建对象类型的例子如例4.2.1所示。

例4.2.1 创建一个对象类型name_t表示一个人的名字，名字中包括名，姓，中间名。下边是用ORACLE的交互SQL环境SQL*Plus写的一个例子^Θ。

```
create type name_t as object -- We call this a "Create Object Type" statement
(
    lname  varchar(30), -- last name
    fname  varchar(30), -- first name
    mi     char(1)      -- middle initial
);
/
-- for Sql*Plus, not part of SQL syntax
```

该SQL语句执行后(/字符被输入以后)，系统已经记录了一个新的数据库类型，即对象类型。name_t的属性为lname, fname和mi。在每一行双短线--后的部分为该属性的注释。 ■

例4.2.1以和C中的结构定义声明一个“结构类型”相同的方式定义了一个ORACLE对象类型name_t以表示人名，但是这种类型的对象仍未生成。对此，我们必须创建一个其中的行或列能够包含对象的表。例4.2.2中给出了一个其中的列能够包含对象的表。

例4.2.2 创建名为teachers的表，其中包含三列：tid(教师的整型标识，整型)，tname(教师名，name_t类型)，room(教师住房的编号，整型)。

```
create table teachers
(
    tid      int,
    tname    name_t,
    room     int
);
```

这是一个空表，因此我们仍然没有创建name_t类型的任何对象。但是，若我们现在执行插入语句插入行到teachers表中，在tname中的列值将使用name_t类型的对象值。这样一个插入语句的例子如下：

```
insert into teachers values (1234, name_t('Einstein', 'Albert', 'E'), 120);
```

在Insert语句中使用的VALUES关键字按照3.10节图3-22的Insert语句的一般形式的语法定义。被插入的第二列使用name_t(...的形式将name_t, lname, fname和mi等属性的值放在一起构成tname列的对象类型name_t的值。这种object_type_name()的形式从它的属性值构造对象，被称为对象构造器（相关的细节包含在下边的图4-5中讨论）。显然，在teachers表的tname列的定义破坏了关系规则1，因为规则1禁止表的列中出现结构值。

我们使用一种“点号”的形式来访问例4.2.2中定义的teachers表中的tname对象列。下边先给出一个仅处理teachers的列的规则SQL语句：

```
select t.tid from teachers t where t.room = 123;
```

^Θ 当例4.2.1的对象类型创建语句在SQL*Plus环境下执行时，终止的分号();后必须跟着一个包含/的行。若一个包含/的行出现在我们先前所介绍的语句中，则该语句将被执行两次。

现在，我们给出一个处理name_t 类型列中的属性的SQL语句；

```
select t.tid, t.tname.fname, t.tname.lname from teachers t where t.room = 123;
```

在t.tname.lname中使用了两个点号，第一个点号表示tname是表t (teachers的别名) 的列，第二个点号表示lname是tname对象值的一个属性。尽管在第3章中我们无须使用表的别名（且当FROM语句中只有一个表时，可以不用表名限定列名），但在访问属性的Select语句中却不行。我们将在后边对此给予详细的说明，在此之前，你必须包括表的别名并用别名限定Select语句中的所有列，除非你能够确定可以不必如此。特别地，下边是刚才提及的Select语句，其中的变量缺少对tname访问的表名限定，该语句无法实现。

```
select tid, tname.fname, tname.lname from teachers where room = 123; ** DOESN'T WORK
```

一个对象类型可用另一个对象类型的属性来定义。例如在例4.2.3中，我们用一个名为pname的name_t属性来定义一个描述人的对象类型person_t。

例4.2.3 构造一个分别包含人的社会保障号、姓名和年龄的对象类型person_t。构造之前先对person_t使用Drop Type语句清除所有已经存在同名的类型。

```
drop type person_t;
create type person_t as object (
    ssno      int,
    pname     name_t,
    age       int
);
```

和例4.2.1一样，要使类型创建语句在SQL*Plus中执行我们需要在其后跟一个带/的行。 ■

为了使person_t的定义语句能够被正确执行，name_t必须是一个已定义的类型，这就是所说的依赖性，即person_t依赖于name_t。试图删除一个为另一个类型所依赖的类型将产生错误，删除该类型之前必须删出所有的依赖于它的类型。例如，若我们试图在例4.2.3的person_t类型定义之后删除name_t类型，我们必须先删除person_t类型及可能依赖于name_t其他所有类型。类型间相互依赖的信息可以从系统维护的目录表中得到（见第7章的最后一节）。

若ORACLE中的一个表中的行包含对象类型，则该表被称为对象表，即每一行由该类型的一个对象组成。例4.2.4构造了一个由person_t类型行组成的表。

例4.2.4 构造一个包含person_t对象（行）的名为people的对象表。使用社会保障号作为表的主键^Θ。

```
create table people of person_t
(
    primary key(ssno)
);
```

我们用4行来写该语句，以便于阅读，当然，空白不影响其含义，我们可以将该语句写为：

^Θ 带PRIMARY KEY约束的列被隐含地定义为非空，尽管NOT NULL语句和PRIMARY KEY语句可能同时出现。实际上，许多老的数据库产品要求被定义为PRIMARY KEY的列同时必须是NOT NULL。由于仅有新的数据库产品具有面向对象的扩展，我们在本章中使用同一列中不带NOT NULL的PRIMARY KEY定义语句。

```
create table people of person_t(primary key(ssno));
```

生成的people表的一个例子如图4-2所示。

people					
包含行对象的无名顶层列					
命名列(顶层属性)	ssno	pname			age
pname中的属性		pname.lname	pname.fname	pname.mi	
第1行	123550123	March	Jacqueline	E	23
第2行	245882134	Delaney	Patrick	X	59
第3行	023894455	Sanchez	Jose	F	30

图4-2 某一给定时刻的例4.2.4的ORACLE 对象表people

关系Create Table语句的形式（第3章图3-2）在定义中描述了所有的列名和该列的数据类型。但在对象表中，对象属性提供了表的列。因此，在例4.2.4 中ORACLE的Create Object Type语句中，只需要描述对象类型（person_t）和一个可选的主键；person_t的属性ssno, pname和age变成了people的列名，在people中，这些属性被称为列或顶层属性。由于ssno被当作一列，在对象类型创建语句中，它可以被描述为主键。

在后边的部分中，我们将用于构造关系表的Create Table语句称为Create Relational Table(关系表创建)语句，将用于构造对象表的构造语句称为Create Object Table(对象表创建)语句，在无须区别的情况下，我们将着两种语句统称为Create Table(表创建)语句。

例4.2.4的people表涉及两个对象类型。person_t对象被称为行对象，因为它处于表的行中。由于象列一样的pname属性自身又是一个对象类型，pname值被称为列对象。在people表中的pname的属性可通过点号来访问，如p.pname.lname，其中，p是people的别名，但是，p被当作属性而不是列。在图4-2中的“无名顶层列”表示我们可以用VALUE()关键字的形式引用整个people行对象(注意不要和Insert语句中的VALEUS关键字混淆)，在非对象的表中这是不可能的。一般地，我们在对象表的图中将不会提到无名顶层行。

由于对象表顶层行属性扮演列的角色，我们可以以常规的方式写简单的查询语句：

```
select p.age from people p where p.ssno = 123550123;
```

我们可以以同样的方式显示列对象pname，例如，

```
select p.pname from people p where p.age > 25;
```

在pname列中的name_t对象将按形如name_t('Sanchez', 'Jose', 'F')的方式被打印出来，稍后我们说明其细节。和普通的表一样，在Select语句

```
select * from people p where p.age > 25;
```

的选择列表中的*形式将显示所有被选中的行；就对象表而言，将显示所有的顶层属性，如图4-3所示。

在图4-3中，非对象列按照我们所期望的方式显示。pname列有一个显示属性名的标题，其值包含在对象构造器name_t()中（在一些系统中，ORACLE将每一行的每一列用一个新的行来显示，一个值在另一个值的下面。这种情况在一个或多个行表示对象时出现，但显示

的形式将随新的ORACLE版本的变化而变化)。

ssno	pname(lname, fname, mi)	age
245882134	name_t('Delaney', 'Patrick', 'X')	59
023894455	name_t('Sanchez', 'Jose', 'F')	30

图4-3 ORACLE的Select * 查询的输出形式，显示所有列

可以使用另一种方式将同样的数据显示为全行对象。这是图4-2的顶层无名列所隐含的方式，即我们可以以单个列值的方式检索诸如people的对象表的所有列。

```
select value(p) from people p where age > 25;
```

再次提醒读者不要将value(p)形式和插入语句中的VALUES关键字相混淆，该查询的结果如图4-4所示。

value(p)	(ssno,	pname(lname, fname, mi),	age)
person_t	(245882134,	name_t('Delaney', 'Patrick', 'X'),	59)
person_t	(023894455,	name_t('Sanchez', 'Jose', 'F'),	30)

图4-4 ORACLE的Select value(p) 查询的输出形式，显示完整的列

在图4-4的输出中，我们看到在选中列中的对象构造器person_t()和它的各个成分一起出现。这在概念上与图4-3的Select * 的查询结果截然不同，在后边的例子中我们将看到这种从一个对象中检索单个对象值的能力如何提供新的功能。

图4-4演示了ORACLE如何显示列对象和行对象：

```
name_t('Delaney', 'Patrick', 'X')
person_t(023894455, name_t('Sanchez', 'Jose', 'F'), 30)
```

这种显示格式和例4.2.2构造一个插入teachers表的列对象name_t的形式相似。ORACLE提供了这种将对象构造器用于从属性值生成对象和显示一个给定类型的对象的一般形式，如图4-5所示。

typename(argument [(,argument...)])

图4-5 ORACLE对象构造器的一般形式

例4.2.5给出了如何使用对象构造器的另一个例子。

例4.2.5 对Jose F. Sanchez显示person_t对象的所有部分，包括社会保障号和年龄。

```
select value(p) from people p
where p.pname = name_t('Sanchez', 'Jose', 'F');
```

注意，这里的对象构造器是name_t()而非value()形式，value()仅能被用于检索一个表的当前行，而不能用于构造一个新的对象。 ■

在例4.2.6中，我们举例说明使用点号访问一个列对象的一个属性的查询，列对象自身不能作为一个列限定。

例4.2.6 发现姓氏以“Pat”开始、年龄大于50的所有人的姓名和年龄。

```
select p.pname, p.age from people p
  where p.pname.fname like 'Pat%' and p.age > 50;
```

和前面的例子一样，在查询中要求用表的别名p。我们将在下边对此给予说明。

在任意的ORACLE SQL语句中，可以用嵌套的点号访问一个对象的属性，该对象又是另一个对象的属性……另一个对象列x的属性，直至某个有限但非常大的层次的嵌套。我们用x.attr1访问对象列x的属性attr1；若attr1是带有属性attr2的一个对象，则我们可以用x.attr1.attr2访问属性attr2，依次类推。

尽管第3章中的SQL语句不要求对某个列引用使用限定符，访问对象属性的SQL语句却有不同的规则。正式的情况下，ORACLE要求所有访问属性的表达式均用别名限定，即这样的表达式如同例4.2.6那样，必须以一个表别名（而非表名）开始，后边跟着带点号的属性嵌套序列。非正式的情况下，ORACLE似乎支持例外的情况，即允许一个对象表（其中的行为对象）的顶层属性被不加限定地使用。例如，考虑对顶层属性pname和fname的访问：

```
select p.pname from people p;    -- table alias used, officially correct, works
select pname from people;        -- no table alias, not officially correct, but works
```

规则对顶层以下需要用嵌套的点号的属性是严格实施的。

```
select pname.fname from people;      -- not a top level attribute: doesn't work
select people.pname.fname from people; -- also doesn't work, since must use alias
```

下边的两个例子说明，Insert和Update语句同样可以使用对象构造器为新行指定值，但带有相当严格的限制。

例4.2.7 建立一个对象类型name_t的名为scientists的表，并为Albert Einstein 在表中插入一行对象。

```
create table scientists of name_t;
insert into scientists values ('Einstein', 'Albert', 'E');
```

由于我们可以用对象构造器name_t('Einstein', 'Albert', 'E')生成一个name_t对象，这似乎意味着下边的Insert语句将有效。

```
insert into scientists name_t('Einstein', 'Albert', 'E'); ** DOESN'T WORK
```

然而，该语句却是无效的。一个简单的原因是该语句不符合图3-22的Insert语句的一般形式，该形式要求表名后面必须跟随一个VALUES关键字或一个子查询。然而，这种将整个scientists行对象用一个对象构造的值替换的更新是有效的。

```
update scientists s set s = name_t('Eisenstein', 'Andrew', 'F')
  where value(s) = name_t('Einstein', 'Albert', 'E');
```

我们可以使用VALUES关键字和一个对象构造器将Einstein姓名对象插入people中。

```
insert into people values (123441998, name_t('Einstein', 'Albert', 'E'), 100);
```

我们同样可以将在people中发现的所有name_t对象插入scientists表中。

```
insert into scientists select p.pname from people p;
```

下边，我们将看到，一个空值同样可以被赋予一个对象。

例4.2.8 将ssno为321341223，pname和age为空值的行到people中。我们可以以两种

方式完成插入。

```
insert into people values (321341223, null, null);
(该插入等价于下边的语句)
insert into people (ssno) values (321341223);
```

由于Insert语句的语法不支持表的别名，顶层属性(如在第2个例子中的ssno)在列表中被不加限定地引用。然而，Update语句却允许表的别名，因此需使用属性的完全限定的表达式。

用Ben Gould替换上一个插入行中值为空的姓名属性。我们不知道它的mi。

```
update people p set p.pname = name_t('Gould', 'Ben', null) where ssno = 321341223;
```

Update语句可用于替换一个复杂对象列的某一部分，例如，单个的两层属性值。下边的语句用C替换刚插入的行的空mi值。

```
update people p set p.pname.mi = 'C' where ssno = 321341223;
```

我们曾提到，可以使用行对象构造器以一个Update语句替换整个行。在执行更新时，我们甚至可以修改主键ssno。

```
update people p set p = person_t(332341223, name_t('Gould', 'Glen', 'A'), 55)
where ssno = 321341223; ■
```

图4-6列出了我们至此所介绍的对象类型和对象表创建和删除的一般形式。所有属性的数据类型均在Create Object Type语句中描述，唯有在Create Object Type语句中命名的属性是那些被指定为NOT NULL或包含在PRIMARY KEY说明中的属性。

```
CREATE TYPE typename AS OBJECT
  (attrname datatype [, attrname datatype ...]);
CREATE TABLE tablename OF typename
  ( [attrname NOT NULL] [, attrname NOT NULL ...]
    [. PRIMARY KEY (attrname [, attrname ...])]);
DROP TYPE typename;
DROP TABLE tablename;
```

图4-6 (迄今为止涉及的) 创建和删除对象类型和对象表的ORACLE语句

REF对象引用的定义

出现在对象表中的对象称为行对象，而在表的列中出现的对象（或其他对象中的属性）称为列对象。ORACLE为每个行对象（但没有列对象）提供一个唯一标识，称为对象标识符，同时，一个表的列（或属性）可被定义为一个称为REF的内部的数据类型，允许它“指向”一个（可能不同的）对象表中行对象。指向某个行对象的REF和该对象自身被认为具有不同的数据类型。与行对象不同，列对象被看做是分组相关属性的一种方便的形式，其自身并不重要；它们不应得到自己的REF。

在图2-2的CAP数据库中有4个表：customers、agents、products和orders。在例4.2.9中，我们重新将这些表转换为对象表并用REF从order行对象依次指向customer、agent和product行对象。我们将看到我们可以使用REF来避免orders和其他表间的低效率的连接。我们可以用3个对象REF来代替orders表中的cid、aid和pid列。但是若没有这些列，我们不可能设置REF的值。这说明最好保存cid、aid和pid列的原来的值，而仅增加3个对象REF列。和前边一样，这里的cid、aid和pid是它们各自所在的表的主键。

例4.2.9 对顾客、代理商、商品和订单使用对象表生成CAP数据库，在每个表中保留原来的列，并依次增加从orders对象表中的每个订单到该订单涉及到的customer、agent和product行对象的REF。

```

create type customer_t as object (...); -- attributes same as columns for customers
create type agent_t as object (...);    -- attributes same as columns for agents
create type product_t as object (...);   -- attributes same as columns for products
create type order_t as object
(
    ordno      int,
    month      char(3),
    cid        char(4),
    aid        char(3),
    pid        char(3),
    qty        int,
    dollars    double precision,
    ordcust    ref customer_t,          -- added column points to customers row object
    ordagent   ref agent_t,            -- added column points to agents row object
    ordprod    ref product_t         -- added column points to products row object
);
create table customers of customer_t (primary key (cid));
create table products of product_t (primary key (pid));
create table agents of agent_t (primary key (aid));
create table orders of order_t
(
    primary key (ordno),
    scope for (ordcust) is customers,
    scope for (ordagent) is agents,
    vscope for (ordprod) is products
);

```

由于order_t属性cid，每个orders行对象有一个cid列（和其他的customers列）；像这样的列经常被用作外码，因为它经常被用在一个连接中以确定与cid主键对应的customers行对象。每个orders行对象有一个ordcust列，一个指向具有相同cid的customers行对象的REF。类似地，每个orders行对象有aid和ordagent、pid和ordprod列。（当然，这里我们假定上边所有的表都已装载，且ordcust、ordagent和ordprod REF值都已适当地设置；其过程可以参考本节的最后一小节“装载带REF的表”。）出现在例4.2.9中orders的Create Table语句中的SCOPE FOR子句用于确保新定义REF的列值总是由SCOPE描述范围内的表中对象；当然，在SCOPE中描述的表必须是一个对象表。这个范围决定了给REF赋值的内部格式；对不在范围内的表的REF引用需要更长的格式。如果一个被引用的对象被删除或不存在，则REF引用是无效的。在初始时，我们假定对REF引用总有相应用对象的。

我们可以在一个REF之后使用点号。我们将其称为对REF的间接引用。下边给出了一个检索所有超过200.00美元的订单号和订购的客户名的查询：

```
select o.ordno, o.ordcust.cname from orders o where o.dollars > 200.00;
```

我们已经使用一个非常简单的REF语句来代替连接查询，其结果不仅使查询变得更加简单，

而且使 cname 值更加便于访问（至少在这种情况下如此）。以下是关于这种简化的更多例子。

例4.2.10 和例3.3.4一样，检索所有的顾客-代理商名对(cname, aname)，其中，顾客通过该代理商发出订单。

```
select distinct o.ordcust.cname, o.ordagent.aname from orders o;
```

原来例3.3.4的查询要求三次连接。 ■

并非所有的涉及订单的查询都如此简单。要求和订单自身做连接的查询就不可能简化，如下例所示。

例4.2.11 和例3.3.7一样找出所有被至少两个客户订购的产品的pid值。

```
select distinct x1.pid
  from orders x1, orders x2
 where x1.pid = x2.pid and x1.ordcust < x2.ordcust;
```

在该查询中，我们检测所引用的订单的 ordcust 值以避免出现重复的结果。例3.7.6给出了执行该查询的另一种方式。 ■

在例4.2.9中的REF允许我们简化基于cid的customers和orders间的、基于aid的agents和orders间的和基于pid的products和orders间的连接。这是在实际中常用的连接，因此是有价值的简化。在一些情况下，通过REF访问比通过连接访问的效率更高，但是，我们只有了解查询计划的前提下才能判定何时会更有效。现在我们可以说通过按REF处理的连接的方式来查询orders中与customers, agents和products相关的信息比直接通过这些表间的主键、外键的连接更有效。

ORACLE提供了一个用于获得出现在其他连接的SQL语句中的REF对象值的REF()函数，下边是该函数的一个例子。

例4.2.12 和例3.4.12一样，查询所有没有通过代理商a05订购商品的顾客名。

```
select c.cname from customers c
  where not exists (select * from orders x
    where x.ordcust = ref(c) and x.aid = 'a05');
```

这里我们通过使用REF值来确定客户以避免显式地引用cid。我们希望这比根据cid搜索更快，但是这只有在对该例中使用的查询计划有所了解的情况下才能确定。至少应该清楚子查询如何依赖于外部的Select语句：这是一个相关的子查询。 ■

在3.5节的FOR ALL查询中，我们看到了由于REF的使用面对复杂度有了很大的改进。这可以减少由标识产生的混乱，而使对象间的基本连接问题变的更加清晰。

例4.2.13 和例3.5.2一样，取得所有通过New York的代理商发订单的顾客的cid值。像先前一样，我们得到“找到无反例存在的客户c，也就是说，对于顾客c找不到一个居住在New York且没有订单x将a与c相连”。这里我们可以将一个相连的订单刻画为对a和c的引用。

```
select c.cid from customers c where          -- select c if
  not exists (select * from agents a        -- there is no agent
    where a.city = 'New York' and            -- living in N.Y.
      not exists (select * from orders x   -- with no order connecting a and c
        where x.ordcust = ref(c) and x.ordagent = ref(a)));
```

到目前为止，我们遇到的查询中，我们假定所有的REF值列均非空且引用正确的对象表中的正确的行。SCOPE子句保证了所有的非空引用在创建时指向正确的表，此外，我们假定REF值在创建后被适当地设置，但是，当被引用的表中的行被删除时，连接到这些行的REF将变成挂起的REF。我们可以用一个新的IS DANGLEING谓词来查询这种异常。

例4.2.14 取得orders中挂起的REF的顾客的cid值。

```
select o.cid from orders o where o.ordcust is dangling;
```

类似地，我们可以使用谓词IS NULL发现空的REF。这与IS DANGLEING不同，因为一个挂起引用可以有一个不正确的值。发现这种异常的另一种方式是使用orders.cid列作为我们将要决定的合适的ordcust值。

```
select o.cid from orders o where o.ordcust <>
  (select ref(c) from customers c where c.cid = o.cid);
```

挂起的REF是非空而无用的。若o.ordcust是空的或挂起的，则o.ordcust.cname像被引用的其他不存在的customers表行的属性一样是空的。这样一个坏的引用不会像C语言中的坏的指针一样地死亡，因为我们可以间接引用它而不导致使程序终止的异常。然而，这可能与检索一个不期望的空值相混淆。

一个对象类型不能嵌套地包含一个与其类型相同的成分，但可以包含一个对具有同样类型的其他对象的引用。考虑下边的police-officers表的定义，其中假定每个警官有一个搭档。

例4.2.15 创建一个带有通过对另一警官对象的REF表示的搭档属性的警官类型。为这些警官创建一个对象表。

```
create type police_officer_t as object
(
  pol_person      person_t,
  badge_number    integer,
  partner         ref police_officer_t
);
create table police_officers of police_officer_t
(
  primary key (badge_number),
  scope for (partner) is police_officers
);
```

例4.2.16 检索其搭档的年龄超过60岁的所有警官的姓。

```
select pol_person.lname from police_officers p
  where p.partner.pol_person.age > 60;
```

ORACLE提供了一个DEREF函数允许 SQL语法检索一个由给定的REF引用的整个对象。下边是一个检索所有警官及其搭档的信息的查询。

```
select value(p), deref(p.partner) from police_officers p;
```

上边查询中的最后一个value(p)属性是partner，一个REF值，其结果将以16进制数的方式打印出来。

(1) REF依赖

一个表集合可以具有一个用REF表示的复杂关系集。例如，雇员具有部门，部门具有经理，经理又是一个雇员。我们可以在employees表中创建一个REF列指向departments行对象，以及一个departments表中的REF列指向employees表中的manager employee列。这里涉及到回路的依赖(部门有一个指向雇员的REF，而且雇员有一个指向部门的REF)，似乎不可能创建一个类型，因为每个Create Type语句指向其他类型。然而，ORACLE支持不完全类型定义技术。你可以用下面的语句部分地创建employee_t类型(为了输出到SQL*Plus在句尾用/代替分号)：

```
create type employee_t
/
           -- note no semicolon used in partial create
```

然后你可以用一个指向employee_t的REF完整地创建department_t，并用一个指向department_t的REF最终创建完整的employee_t类型。当我们试图删除另一个这样相互引用的表或类型集合时，我们必须在删除类型之前删除表。但是，由于类型间存在相互引用，当我们试图删除两个类型的时候，我们将发现有错误返回。为了避免这种情形的发生，不是

```
DROP TYPE typename;
```

简单地使用：

```
DROP TYPE typename FORCE;
```

我们必须对在多个表间的REF环中涉及的所有类型使用：

(2) 装载带REF的表

我们需要单独考虑装载带REF列的表。在orders表的例子中，REF列完全由orders表中的外码cid, aid或pid列值决定，因为这些列是它们自身表的主键。我们可以从用同样的数据使用同样的装载过程开始，像附录A中描述的关系情形那样。然后，我们可以使用如下一个Update语句设定所有的引用。

例4.2.17 用带有对customers、agents和products的正确的引用替换orders中的任意当前的REF值。

```
update orders o set
    ordcust = (select ref(c) from customers c where c.cid = o.cid),
    ordagent = (select ref(a) from agents a where a.aid = o.aid),
    ordprod = (select ref(p) from products p where p.pid = o.pid);
```

这里我们使用一个标量子查询来检索REF值，ORACLE允许的并且通用的Update语句形式，如图3-23所示。 ■

例4.2.18 从people表中选择一个ssno为033224445，徽章号为1000的警官插入到例4.2.15的表中，新警官的搭档是另一个在police_officers中已经存在的徽章号为990的警官，因此，新警官的搭档可以被立即确定。

```
insert into police_officers
    select value(p), 1000, ref(p0) from people p, police officers p0
        where p.ssno = 033224445 and p0.badge_number = 990;
    update police_officers set p.partner = (select ref(p0) from police_officers p0
        where p0.badge_number = 1000) where badge_number = 990;
```

在上边的Insert语句中，我们使用一个子查询从people中挑出正确的行对象并且使右边

的REF指向police_officers。这里需要一个Update语句来设定badge_number为990的警官的REF值。

4.2.2 INFORMIX中的对象行类型

INFORMIX行类型与我们在ORACLE中讨论的对象类型密切相关。INFORMIX行类型由Create Row Type(行类型创建)语句定义，如例4.2.19所示。注意在ORACLE中被称为属性的成分在INFORMIX中被称为字段。

例4.2.19 创建一个带有两个类型为varchar(30)的字段lname和fname和一个char(1)字段mi的行对象。该行类型与ORACLE中的例4.2.1相似。

```
create row type name_t
(
    lname      varchar(30),
    fname      varchar(30),
    mi         char(1)
);
```

现在，我们可以将行类型实例称为对象或行对象，因为这些对象与ORACLE中的行对象相似，同时它们也满足对象的某些特性。例如，它们具有一个用户定义类型，该用户定义类型具有值的内部成员结构，但在许多方面像内部类型。然而INFORMIX不使用对象这一术语，而将它们称为行类型的行或实例。

像ORACLE一样，例4.2.19中的name_t行(对象)类型定义并不创建该类型的任何实例。为了创建这样的实例，我们需要定义一个其行或列可以包含这样的实例的表，如后边的例4.2.21所示。

一个行类型可以被用来定义另一个行类型的字段。在例4.2.20中，我们定义了一个INFORMIX行类型person_t，描述一个带有称为pname的name_t字段(简记为person_name)和其他字段。例4.2.20与ORACLE的例4.2.3相似。

例4.2.20 创建一个分别包含人的社会保障号、姓名和年龄的person_t行类型，在此之前先删除已经存在的同名行类型。

```
drop row type person_t restrict;
create row type person_t
(
    ssno      int,
    pname     name_t,
    age       int
);
```

在所有的INFORMIX Drop Row Type语句中必须有“可选的”RESTRICT关键字。尽管在所有的Drop Row Type语句仅要求一个选项显得有点奇怪，下边将对此进行说明。

由于person_t 的定义中使用了name_t，我们说person_t依赖于name_t。像ORACLE 中一样，我们不能在person_t存在时删除name_t行类型。为了删除这两个行类

型，我们必须按“引用者优先”的次序来删除它们：先删除person_t，再删除name_t^Θ。类型和表间相互引用的信息可以从数据库系统中的目录表中得到。见第7章的最后一节。

在INFORMIX中，若一个表中的行是已命名的行类型，则该表被称为类型表。一个INFORMIX类型表对应一个ORACLE对象表。例4.2.21创建了一个与ORACLE中的例4.2.4相似的带有person_t类型行的表。

例4.2.21 创建一个称为people的person_t对象的表，使用社会保障号作为主键。

```
create table people of type person_t
(
    primary key (ssno)
);
```

■

后边我们将例4.2.21中的CREATE TABLE table_name OF TYPE typename ... 语句称为Create Typed Table(类型表创建)语句，将用列名列表创建关系表的旧语法称为Create Relational Table(关系表创建)语句。当我们对此不加区别时，我们将二者都称为Create Table(表创建)语句。创建和删除行类型和类型表的一般形式见图4-10。特别地，你将注意到INFORMIX不像ORACLE，不在Create Type语句而是在Create Typed Table语句中定义NOT NULL字段。

与在ORACLE对象表中处理顶层属性的方式相对应，用于定义类型表的行类型的顶层字段将像表中的列一样起作用，这样，例4.2.21中的people表将像图4-2描述的模型那样具有ssno，pname和age列。类型表可以被看做是带有一个行类型对象列的表或像ORACLE中带有命名列序列的表。这样，我们可以使用顶层字段名ssno，pname和age来查询INFORMIX类型表，如下的Select语句可以完成这样的功能：

```
select age from people where ssno = 123550123;
select pname from people where age > 25;
```

或者使用等价的限定列引用

```
select q.pname from people q where q.age > 25;
```

注意，在INFORMIX Select语句中不必像ORACLE中那样使用列的限定别名。这里的name_t对象将按row('Sanchez', 'Jose', 'F')的格式被打印出来。下面我们会对此做进一步的讨论。选择列表的*格式：

```
select * from people where age > 25;
```

将显示所有的列。和ORACLE的图4-3相比，在图4-7中的列对象pname的显示格式将有所不同，特别是用row()构造器的形式代替name_t()对象构造器的形式的用法。

ssno	pname	age
245882134	row('Delaney', 'Patrick', 'X')	59
023894455	row('Sanchez', 'Jose', 'F')	30

图4-7 INFORMIX SELECT *查询输出，显示所有的列

^Θ 在SQL-99标准中除RESTRICT以外还描述了另一个称为CASCADE的选项，用来删除一个用户定义类型及所有依赖于它的类型。RESTRICT和CASCADE选项中的一个必须在所有这样的Drop语句中描述，但是RESTRICT是SQL-99核心标准的一部分，而CASCADE不是，因此INFORMIX仅使用RESTRICT一个选项是合理的。ORACLE提供了不使用RESTRICT的非标准的能力，并提供了一个不同于CASCADE的FORCE选项。

为了显示“整个”行对象（而非行对象中列的集合），我们可以使用表的列名自身作为选择列表中的唯一元素。考虑如下查询和图4-8中的显示格式（在ORACLE中我们需要在例4.2.5的选择列表中使用VALUE(p)）。

```
select p from people p where age > 25;
```

p
row(245882134, row('Delaney', 'Patrick', 'X'), 59)
row(023894455, row('Sanchez', 'Jose', 'F'), 30)

图4-8 INFORMIX SELECT table_alias查询的输出形式，显示全部的行对象

行类型对象可以使用行构造器 ROW()构造，再转换成适当的行类型。图4-9给出了类型转换表达式CAST(...的语法及其使用方式（注意在ORACLE的例4.2.5和例4.2.8中我们使用形如name_t(...和person_t(...)的对象构造器来实现同样的功能）。

ROW(expression {, expression...}).
CAST(ROW(expression {, expression...})) AS rowtype)

图4-9 INFORMIX 非类型和类型行对象的一般形式

```
cast(row('Delaney', 'Patrick', 'X') as name_t)
cast(row(023894455, cast(row('Sanchez', 'Jose', 'F') as name_t), 30) as person_t)
```

例4.2.22 对Jose Sanchez显示整个person_t对象，并为其姓名使用name_t对象。

```
select p from people p
  where p.pname = cast(row('Sanchez', 'Jose', 'F') as name_t);
```

这里有一种将INFORMIX的ROW()函数表达式转换为适当命名类型的快捷方式，row_expression::typename。这样，我们可以将先前的Select语句写为：

```
select value(p) from people p
  where p.pname = row('Sanchez', 'Jose', 'F')::name_t;
```

像ORACLE中访问属性中的属性一样，INFORMIX中的点号可用于访问字段中的字段。二者唯一的差别在于INFORMIX中不必在这些表达式中使用表限定符；如果在Select语句中仅出现一个表，则我们可以选择使用列.字段，别名.列或表.列.字段的方式来访问列中的字段。

例4.2.23 找出名字以“Pat”开始、年龄超过50岁的所有人的姓名和年龄。

```
select pname, age from people
  where pname.fname like 'Pat%' and age > 50;
```

这里的pname是行对象的顶层字段，pname.fname是pname的fname字段。注意这种形式不能用于ORACLE中，因为这里没有表的别名（参见例4.2.6）。■

插入和更新也可以使用行构造器。像ORACLE一样，INFORMIX中对象也可以为空，因为空值是一个受过全面考验的数据项值。

例4.2.24 先将Albert Einstein的行对象插入到people表中，该对象的社会保障号为

123441998，年龄为100。然后插入一个社会保障号有意义而pname和age为空的对象。

```
insert into people values (123441998, cast(row('Einstein', 'Albert', 'E') as name_t), 100)
insert into people values (321341223, null, null);
```

(或者使用等价的方式)

```
insert into people (ssno) values (321341223);
```

在最后一个插入的行中将空名字替换为“Ben Gould”。

```
update people set pname = cast(row('Gould', 'Ben', null) as name_t) where ssno = 321341224;
```

用“c”替换空间中间名，保留已指定的名字的其余部分。

```
update people set pname = cast(row(pname.lname, pname.fname, 'C') as name_t)
where ssno = 321341224;
```

这里的行构造器ROW()从非常量表达式(pname.lname和pname.fname)构造一个行。现在，我们生成一个name_t行类型表scientists并将people中的名字插入其中。

```
create table scientists of type name_t;
insert into scientists
select pname.fname, pname.minitial, pname.lname from people;
```

图4-10给出了创建和删除行类型和类型表的一般形式。

图4-10还给出了Create Typed Table语句的一般形式（创建关系表语句的形式已在图3-2中给出，由于行类型是受过全面考验的类型，所以我们同样可以用行类型列创建关系表）。一个类型表的所有列均建立在Create Table语句中定义的字段的基础上，若合适的话，可以带有NOT NULL说明，而Create Typed Table语句的括号中的PRIMARY KEY说明是唯一的可选项。注意，ORACLE将NOT NULL描述放在Create Object Table语句而非Create Type语句中。Drop Table说明不会产生例外，但Drop Row Type需要在末端使用RESTRICT子句。

```

CREATE ROW TYPE rowtype
  (fieldname datatype [NOT NULL], fieldname datatype [NOT NULL] ...);
DROP ROW TYPE rowtype RESTRICT;
CREATE TABLE tablename OF TYPE rowtype
  ([ PRIMARY KEY (column [,column ...])]);
DROP TABLE tablename;

```

图4-10 INFORMIX创建和删除行类型和类型表的一般形式

1. INFORMIX中缺乏REF引用

当前的INFORMIX版本与ORACLE的一个重要不同在于缺乏REF。将来的INFORMIX版本可能会加入REF。

2. INFORMIX中的类型继承

INFORMIX具有一种ORACLE不具备的能力，即类型层次中的类型继承。如果我们需要用创建一个新的employee类型作为person类型的子类型的方式来定义层次的思想，其中的子类型继承了父类型中的数据项和方法，并将其中的一部分作为自己的数据项和方法。回顾一下person_t类型的定义：

```
create row type person_t
(
  ssno      int,
  pname     name_t,
  age       int
);
```

我们可以定义一个名为employee_t的子类型如下：

```
create type employee_t
(
    eid      int,
    salary   double precision,
    mgrid   int
)
under person_t;      -- says employee_t is subtype of person_t
```

employee_t类型的顶层字段现在将由一个已在类型和子类型定义中命名的字段的联合构成：eid, salary, mgrid, ssno, pname和age。一个称为manger_t的新类型可以被定义为employee_t类型的子类型。

```
create type manager_t
(
    budget   double precision,
    groupname  varchar(30)  -- name of group managed
)
under employee_t;
```

manager_t类型对象继承了为employee_t定义的所有字段。现在我们可以直接创建一个包含像manager_t那样的子类型对象的类型表。

```
create table managers of type manager_t;
```

但是，为了获得实现继承的某些行为，我们需从一个最高层类型表开始定义子表：

```
create table people of type person_t;
```

```
create table employees of type employee_t
under people;           -- employees table is subtable of people
```

```
create table managers of type manager_t
under employees;        -- managers table is subtable of employees
```

现在，我们可以看到具有表继承能力的SQL行为的一些重要方面是：当一个行被插入到managers表中时，它也将自动成为employees表中的一行。因此查询

```
select pname.fname, pname.lname, eid from employees
where salary > 50000;
```

将不仅只从已在employees表中插入的限定雇员检索信息，而且将从在managers表中插入的限定雇员检索信息。同时，managers表是一个独立的实体，因此，若我们写查询

```
select pname.fname, pname.lname, eid from managers
where salary > 50000;
```

则我们将只检索到已插入managers中的行。最后，如我们在上边的查询只想检索那些不是经理的雇员，则可以将查询重写为：

```
select pname.fname, pname.lname, eid from only (employees)
where salary > 50000;
```

这种方式标志着创建一个单独的manager_t类型的manager表与将其创建为employees表的子表的差异。

4.2.3 对象和表小结

我们已经介绍了如何创建对象（行）类型和包含这些类型的对象的表。在两个主要的数

据库产品中，这些对象可以作为行值也可以作为列值。除了一些关键字和术语的不同外，它们是非常相似的。

(1) ORACLE和INFORMIX 对象间的相似性

两个产品都具备以下特征：

- 允许程序员在数据库类型之外建立复杂的复合对象（行）类型。
- 允许用户提供新的对象数据类型，如用于普通的表的列的数据类型。
- 允许对象嵌套。
- 在查询中使用点号来访问属性，并允许点号多重嵌套。
- 允许对象填满表的一整行，在这种情况下允许顶层属性作为表的列使用。
- 不支持对象数据的封装（即数据未对外部视图隐藏，见下边“面向对象”部分的解释）。

(2) ORACLE和INFORMIX 对象间的差别

- ORACLE提供使用REF的能力，而INFORMIX没有。
- INFORMIX提供类型层次和继承功能，而ORACLE没有。

面向对象

将对象模型引入数据库类型系统的一个主要的目的是希望支持面向对象的编程语言（如Java和C++）的概念。1989年，6位在该领域颇有名望的专业人士起草了一份题为“面向对象的数据库系统宣言”的论文。该论文试图将面向对象数据库系统的各种特性加以优化（参见推荐读物[1]）。该宣言列出了面向对象数据库系统必备的13条特征（或称为“金规玉律”）。面向关系数据库系统只采用了其中的一部分。这里我们列出ORDBMS的一些相关特性。

在面向对象术语中，对象是数据结构和作用于这些数据结构的函数（在ORSQL中称为用户定义函数）概念的组合。一个涉及对象的用户定义函数如果是该对象定义中隐含的部分，则也称为对象的方法。在面向对象术语中，如果用户不能通过除对象提供的方法以外的方式访问对象中的数据，则该对象被称为是封装了数据和方法。然而，ORSQL对象不支持真正的封装，因为方法只是访问数据的一种方式：直接用SQL查询和更新数据也是允许的（也可以说不支持任意SQL访问的封装违背了数据库查询语言的目标）。

带有特定的结构和方法集合的对象在OODBS中被称为属于某个OODBS中的对象类（对象属于一个类就像编程语言中的变量被声明属于一个给定的类型）。ORDBMS优先使用类型这一术语而非OODBS术语类。

最后，OODBS模型允许创建新类来扩展已有的类的描述，这被称为类型层次，这样的新子类型被称为继承了它被定义的类型的数据和方法。

面向对象特征	ORACLE对象类型	INFORMIX行类型	SQL-99用户定义类型
封装	不支持	不支持	不支持
用户定义函数	支持，包括方法	支持	支持，包括方法
对象引用	支持	计划支持	支持
继承	不支持	支持	支持

正如我们所见到的那样，INFORMIX允许行类型在继承层次创建。一个经理是一个带有额外信息的雇员，`manager_t`行类型的定义只需要显式地列出这些额外的字段加上`under employee_t`下的短语。INFORMIX版本9.2尚不支持对象引用即REF，但计划在版本9.3中

支持它。

正如将在4.4节介绍的那样，ORACLE允许为对象类型定义方法函数。INFORMIX中有不受限于单个对象或对象层次的用户定义函数，该函数具有方法一样的功能。ORACLE允许一个行对象被一个来自其他对象属性或表列的REF(对象引用，其他数据类型)指向该行对象。

新的SQL-99标准具有和ORACLE对象类型或INFORMIX行类型相似的用户定义类型(UDT)。然而，在SQL-99的基本核心中没有包括该部分。UDT具有像ORACLE的方法和引用及像INFORMIX的继承功能，但像这两种产品一样没有封装的特性。缺乏封装可能使标准体不能将他们称为对象，并终止于中性的无色彩的名字用户定义类型。

尽管ORACLE不支持对象继承并且INFORMIX版本9.2不支持对象方法，两者都不会对一般的应用构成障碍。

4.3 汇集类型

汇集类型允许我们在一个行的某一列中放入多个值(值的汇集)，这显然违反了关系规则1，该规则不允许多值属性。例如，汇集类型允许我们表示EMPLOYEES表一个列中雇员的家属集，如图4-11(重复第2章的图2-3)所示。

EMPLOYEES			
eid	ename	position	dependents
e001	Smith, John	Agent	Michael J. Susan R.
e002	Andrews, David	Superintendent	David M. Jr.
e003	Jones, Franklin	Agent	Andrew K. Mark W. Louisa M.

图4-11 带多值dependents属性的EMPLOYEES表

由于在不同的数据库产品对聚集类型的处理并不完全一致。我们将在4.3.1节中介绍ORACLE的嵌套表和数组，在4.3.2节中介绍INFORMIX的集合、列表和多重集，然后在4.3.3节中对两者进行一般讨论性总结。

4.3.1 ORACLE 中的汇集类型

ORACLE中有两种汇集类型：表类型和数组类型。每个汇集类型描述包含所有相同类型的项，即元素类型。一个元素类型可以是内部类型或对象类型，但ORACLE中的元素类型不能为汇集类型。汇集类型是数据库系统中的成熟的数据类型，经过适当的转换，可以在查询中解释为表。我们先介绍表类型，再说明数组类型。这里针对ORACLE 8.1.5版本或更高版本进行介绍，ORACLE 8.0支持汇集类型，但从8.0到8.1.5版本在语法上有较大的变动，这里列举的一些查询在8.0版本中将不能正常工作。

1. 表类型和嵌套表

我们定义可被用以实现多值属性的表类型。

例4.3.1 创建一个名为dependents_t的表类型包含person_t对象的表。

```
create type dependents_t as table of person_t;
```

现在我们创建一个带有`eid`, `person`, `dependents`列的表, 其中`dependents`可带有多个值。我们假定`eid`是主键。

```
create table employees
(
    eid          int,
    eperson      person_t,
    dependents   dependents_t,
    primary key  (eid)
) nested table dependents store as dependents_tab;
```

注意, 当以适当的列首次创建一个`employee_t`类型时, 我们可以创建`employees`表的一个对象版本。 ■

例4.3.1创建了一个标准的关系表, 除了它包含一个表类型列`dependents`和一个形如
NESTED TABLE colname STORE AS tablename

的NESTED TABLE子句。其中, 大写字母是关键字。在例4.3.1中, 该语句描述了一个名为`dependents_tab`的表, 该表包含多个可出现在`employees`表的`dependents`列中的`person_t`对象。这样, 两个数据库表将被一个Create Table语句生成, 表`employees`作为顶层的表, 表`dependents_tab`包括所有雇员的家属的`person_t`对象。图4-12给出了例4.3.1的`employees`表, 该表带有两个雇员行, 一个雇员带有两个家属, 而另一个仅带有一个。

employees		
eid	eperson	dependents
101	person_t(123897766,name_t('Smith', 'Michael', 'J'),8) ('Smith', 'John', 'P'),45)	person_t(322456776,name_t('Smith', 'Michael', 'J'),8)
		person_t(123822332,name_t('Smith', 'Susan', 'R'),12)
102	person_t(432112233,name_t('Andrews', 'David', 'S'),32)	person_t(565534555,name_t('Shaw', 'David', 'M'),3)

图4-12 带有嵌套表类型的`dependents`列的ORACLE `employees`表

我们注意到雇员101的两个家属可以被当作是在概念上构成一个小的对象表, 如同雇员102的一个家属那样, 尽管事实上所有的`dependents`都在单个的`dependents_tab`表中。然而, 这个概念上的表在`employees`的单个行中作为列值出现被称为嵌套的表。

图4-13显示了用例4.3.1的NESTED TABLE语句创建关系表的语法。到现在为止, 我们仅看到一个带一个嵌套表列的关系表例子, 构造带有多个嵌套表列的表并不难, 那样我们需要为每个这样的列声明一个NESTED TABLE子句。

```
CREATE TABLE tablename (columnname datatype [NOT NULL]
    {, columnname datatype [NOT NULL]...}
    [, PRIMARY KEY (columnname [, columnname...]})
[NESTED TABLE columnname STORE AS tablename
    {, NESTED TABLE columnname STORE AS tablename...}];
```

图4-13 ORACLE中带有嵌套表存储描述的创建关系表的语法

注意, 图4-6 Create Object Table语句形式可以像图4-13那样扩展, 即在表对象类型的某

个嵌套层上为每个表类型属性增加一个NESTED TABLE子句。例如，在例4.3.1中，定义了一个带有dependent_t类型的dependents属性的一个employees_t类型，然后定义employees_t类型的一个employees对象表。然而，在ORACLE中一个表类型不能包含一个表类型属性，多层次规范的对象类型嵌套以完全包含一个表类型属性将十分完美。这样，我们能够创建一个包含一个人当前的公司名和雇员信息（在类型employees_t的属性内）的属性的对象类型workhistory_t；然后，对出现在第二层嵌套中的dependents_t属性使用NESTED TABLE子句，我们可以创建一个名为workhist的workhistory_t类型的对象表。

ORACLE嵌套表可以被交互地查询，但这种查询带有某些限制。我们已看到嵌套的表被包含于它自身的数据库表中（我们将称之为子表）。独立于嵌套它的表（称为父表）而存在。但是一个子表的数据仅能通过父表访问。在例4.3.1的表的Create Table语句之后，在FROM子句中的带dependents_tab的Select语句将无效，错误信息显示用户必须访问父表。

下边，我们提供一些访问嵌套表的例子。

例4.3.2 检索雇员101的所有家属的嵌套表。

```
select dependents from employees where eid = 101;
```

这将显示一个雇员行，其中dependents列为一个标量嵌套表值，如图4-14所示。在这个查询中，虽然我们可以为employees提供一个别名e，写作e.dependents，但我们没有使用限定符。只有对对象属性的引用限定符才是必需的，而dependents是一个关系表的非对象列。这里检索到的嵌套的标量表值表示包含两个不同行的表——但其本身并不是包含两行的表。正如我们将看到的那样，这种限制有重要的含义。

<pre>dependents(ssno, pname(fname, minitial, lname), age) ----- dependents_t(person_t(322456776, name_t('Smith', 'Michael', 'J'), 8), person_t(123822332, name_t('Smith', 'Susan', 'R'), 12))</pre>

图4-14 例4.3.2的两个家属的ORACLE输出形式

在图4-14中，嵌套表的输出格式与图4-3中的对象类型的显示格式相似，即类型名之后跟着括号括起来的各个部分的列表。 ■

例4.3.3 检索所有employees的所有dependents。

```
select dependents from employees;
```

这将为每个雇员显示一行，其中，dependents按图4-14所示的格式作为一个标量嵌套表值列出。 ■

例4.3.4 检索有6个以上家属的雇员的eid。

```
select eid from employees e
where 6 < (select count(*) from table(e.dependents));
```

这是我们第一次看到出现在FROM子句中的嵌套表，这里我们必须使用TABLE()形式将标量嵌套表值e.dependents转换成一个表，目的是完成其上Select语句。若没有这样的转换，检索到的e.dependents标量值将不能返回查询所需的person_t对象。在这里我们不必将

子查询放在比较谓词的右边：ORACLE支持允许子查询在任何一边的高级SQL比较谓词。

为了发现一个带有某一家属集合的employees行，我们可能尝试如下查询：

```
select eid from employees e where e.dependents = . . .
```

在右边使用集合构造的形式描述期望的集合。ORACLE不支持嵌套表之间的等值匹配谓词，但支持同一对象类型的简单对象或内部类型之间的匹配。下面，我们将简单对象定义为在其属性内无任何层次的聚集类型对象的对象。

为发现带有某一确定家属的雇员，我们可以使用谓词IN。

例4.3.5 列出带有社会保障号为3451112222的家属的雇员的eid。

```
select eid from employees e
where 3451112222 in
  (select d.ssno from table(e.dependents) d);
```

由于ORACLE支持简单对象的等值匹配，我们一样可以检索带有由name_t ('Lukas', 'David', 'E') 指定的名字的家属的雇员的eid。

```
select eid from employees e
where name_t('Lukas', 'David', 'E') in
  (select d.pname from table(e.dependents) d); ■
```

在例4.3.5的子查询的FROM子句中，我们又一次使用了TABLE()形式。图4-15给出了TABLE()形式的语法。

FROM TABLE(collection_expr)

图4-15 ORACLE中TABLE()的形式

注意，图4-15的语法要求可以指定汇集列值的指定的行。在下列情况下，我们不能对TABLE()决定指定的行，因此，TABLE()形式不能用于提供最外层Select语句的FROM子句的表。

```
select ssno from table(employees.dependents); -- Invalid: employees row undetermined
```

将嵌套表引入查询的另一种方式在例4.3.6中给出。

例4.3.6 假如我们要检索雇员101的家属数目，则下边是一种可能合理的方法：

```
select count(*) from
  (select e.dependents from employees e where e.eid = 101); -- ** UNEXPECTED RESULT
```

然而，不幸的是，这里的子查询检索一个标量嵌套表值e.dependents，而非为每个家属带有person_t对象的表，因此，计算的结果将是1（如果期望的雇员有空的e.dependents值，则是0）。为此，我们需要将e.dependents值转换为一个表，但我们不能用选择列表中的TABLE()格式来完成此功能，将子查询变换为：

```
(select table(e.dependents) from employees e where e.eid = 101); -- ** BAD SYNTAX
```

由于子查询的工作是为单个行传递标量值，而不是为整个表。为了弥补这一点，在它已经由子查询传递之后我们需要将e.dependents值转换成一个表，并且这可以通过如下的TABLE()形式做到这一点：

```
select count(*) from
  table(select e.dependents from employees e where e.eid = 101);
```

使用这种语法，我们可以得到单个雇员的实际家属数。首先，子查询选择eid为101的雇员并检索e.dependents作为一个标量嵌套表值；然后，TABLE()将值返回到person_t对象的行值集合中，COUNT(*)在外层Select语句中计算行数。

一般地，TABLE关键字可以被看做从一个子查询或集合表达式产生的嵌套结果表值删除“集合容器”，表现其中的所有行。这种分解子查询的集合值为其内容的行为被称为消除嵌套集合。

在例4.3.6中，我们已经尝试了另外一种不同的方法：

```
select count(e.dependents) from employees e where e.eid = 101;
```

这里，我们希望用COUNT()计算这一行中的家属数。但是，在Select语句中的e.dependents再次表示该行的单个嵌套表值，因此，我们将仅计算从employees表的适当的行中选择的零或非零，空或非空的e.dependents标量值。但事实上，ORACLE将返回一个错误“不一致的数据类型”，因为ORACLE规定COUNT()不能采用聚集值作为参数。这样的规定可以避免混淆。但是，正如我们将在下一节中见到的那样，INFORMIX提供了一种在这种情况下代替COUNT()的新形式，即CARDINALITY()，这种方式非常有用。

例4.3.7 代替例4.3.6那样计算eid为101的雇员的家属数，我们可以从同样的家属嵌套表中显示所有他们的社会保障号，如下：

```
select d.ssno from
  table(select e.dependents from employees e where e.eid = 101) d;
```

这里，和例4.3.6一样，子查询对雇员101选择dependents嵌套表，然后TABLE()操作符将其消除嵌套到一个person_t对象表中。在这种情况下，对dependents使用别名d。在选择列表中的d.ssno选择要显示的表中的ssno列。

2. 从表中表检索的两种技术

从所有雇员的所有家属中检索ssno列的值不是一个简单的问题。我们需要从多个嵌套表(每个雇员一个)中选择，并立即报告所有的行，而不仅仅检索我们已经做过的嵌套表的行。我们不再需要从表中表选择ssno列值，因为每个嵌套表不再是聚集值，而是已经变成了person_t行对象的表。在例4.3.3中，我们实现了以标量值的方式在dependents列中检索所有数据，但将dependents值作为其自身的表对待则更加困难。

为了讨论处理表中表的问题，ORACLE的设计者们提出了两种不同的机制：两层表的表乘积和嵌套游标。

3. 利用表的乘积消除嵌套

如下雇员和家属表的乘积可被用于消除家属表的嵌套，显示所有雇员的标识和家属ssno值。

```
select e.eid, d.ssno from employees e, table(e.dependents) d;
```

这将显示一个带有eid和ssno两列的表，该表的每行是每个eid和ssno的组合。无家属的雇员将不会出现在该表中。为了看到无家属的雇员行(带有空ssno)，可以在家属的边上加一个(+)号，形式如下：

```
select e.eid, d.ssno from employees e, table(e.dependents) (+) d;
```

(+)号来自于3.6节中介绍的ORACLE的外部连接。在这两种情况下，(+)都标志着空值将被填入的另外一个表的保留行的那一侧。

回忆一下从3.9节开始到3.15节结束的查询执行的一般模式。首先形成FORM表的关系乘积。对上边涉及到的table(e.dependents)的表的乘积，我们有一个修改过的关系乘积运算。标准的乘积将第一个表中的每一行和第二个表的每一行进行匹配。在包含消除嵌套的表乘积中，第一个表的每一行和它自己的聚集列值(被转换为一个表)进行匹配。

这里我们有一个带有类型行x和表的别名为e的employees表。行x有一个带有类型行y的名为dependents的聚集列表，别名为d。这里的关系乘积为对所有的x和y产生形如x||y的行的表。也就是说，一个典型结果行是被其某个家属行填充的雇员行。每个这样的复合行保留它的某个雇员和该雇员的某个家属的信息。这样，我们可以在查询中对雇员和家属的任意列分别使用e.colname和d.colname而不会产生混淆。随后的执行步骤(如while子句的处理)根据这种无二义值的方式继续进行。特别地，e.dependents汇集列值自身有一个明确的含义，在我们需要进一步查询一个特定的雇员的家属集合的时候，可将它当作TABLE()表的主体，正如我们将在例4.3.9中看到的那样。

4. 嵌套游标

显示表中表的另一种方式是使用嵌套游标。游标像一个在执行查询时移过行集合的循环控制变量。顶层的Select提供一个在FROM表上像游标一样的循环，以产生待检索的行。然后，当在外层Select循环中遇到时，嵌套的游标语法CURSOR()允许我们增加一个扫描每一个行中嵌套表的第二层循环(如果是多个嵌套可以是更多的循环)。

例4.3.8 显示雇员表中的所有行的eid值和该雇员的年龄小于16岁的任意家属的社会保障号。

由于必须从多重嵌套的表中显示多行，我们需要通过表乘积或游标的形式来消除嵌套。我们先看一个使用表乘积形式消除家属嵌套表的查询：

```
select e.eid, d.ssno as dep_ssno from employees e, table(e.dependents) d
  where d.age < 16;
```

该查询的输出如图4-16所示。

EID	DEP_SSNO
101	322456776
101	123822332
102	565534555

图4-16 例4.3.8的表乘积查询的ORACLE SQL*Plus输出

然后，我们使用嵌套游标来检索同样的信息：

```
select e.eid, cursor(select d.ssno as dep_ssno
  from table(e.dependents) d where d.age < 16) dep_tab
  from employees e;
```

这里的SELECT设定一个在employees表上的循环，CURSOR设定在被检索到的每个employees行的dependents嵌套表上的第二级循环。图4-17给出了图4-12表示的雇员表的显示形式：外层循环发现eid 101，内层循环为该雇员行发现两个ssno，然后，外层循环发现eid 102，内层循环再发现一个ssno。

```

EID DEP_TAB
-----
101 CURSOR STATEMENT : 2
CURSOR STATEMENT : 2
    DEP_SSNO
    322456776
    123822332
102 CURSOR STATEMENT : 2
CURSOR STATEMENT : 2
    DEP_SSNO
    565534555

```

图4-17 例4.3.8的嵌套游标查询的ORACLE SQL*Plus输出

注意，图4-17的显示不是关系的形式，因为它包含了两种不同的格式行，并带有相同的标题。嵌套游标在外层Select扫描父表时产生对嵌套表的扫描。外层扫描首先显示EID和它下边的101，接着DEP_TAB作为游标表达式(CURSOR STATEMENT: 2)的别名，其下的内层CURSOR()扫描的元素输出一个标记为dep_ssno(列的别名在查询过程中产生)的表和eid 101的两个家属的ssno值。诸如dep_ssno的列别名对嵌套游标检索的可读性是很重要的。前边的步骤完成后外层扫描显示另一个eid 102和另一个dep_ssno表，这次仅带有一个家属的ssno。

CURSOR()表达式的一般形式如图4-18所示。子查询可能返回如例4.3.8所示的单个嵌套表的列值或正常的关系行集合。这两种子查询类型均可用于CURSOR()，在使用关系行的情况下，CURSOR()将移过子查询的所有行。

```
cursor expression ::= cursor(Subquery)
```

图4-18 ORACLE中CURSOR()表达式的一般形式

在图4-18中，CURSOR()被划分为表达式，但是，它仅能用于顶层选择列表中，也就是说作为下列表达式之一：

```
select expression, expression, .... from ... ;
```

此外，如果存在一个嵌套表或在外层扫描中遇到行集合，则一个CURSOR()可被用于另一个中。我们注意到游标是产生一种对嵌套表查询的显示非常有效的表中表数据的新方式。

到目前为止，所有的Select语句都是以关系的形式检索数据，即每个被显示的行具有同样的数据类型模式(如两个字串和一个数字)。一旦我们允许聚集对象值作为表单元可接受的值，甚至是在本节中介绍CURSOR()之前所涉及的聚集查询的结果也是纯关系的表，但是，CURSOR()的结果不是关系表，因为一些被显示的行描述外层循环的过程和一些内层循环过程，在这种情况下，其结果带有不同数量和类型的数据值。

另一方面，如图4-16所示，表乘积查询按关系的方式检索数据。如果你对单个雇员有多个行不介意的话，则这种显示方式更加简单直观。

例4.3.9 按最老的家属的名字列出雇员名和eid。

```
select eid, cursor(select d1.pname.fname from table(employees.dependents) d1
  where d1.age = (select max(d2.age) from table(employees.dependents) d2))
  from employees;
```

图4-15给出的TABLE()的语法及其后的讨论意味着如下语句：

```
select fname from table(employees.dependents) d1
```

可能产生问题。因为TABLE()必须以单层嵌套表的列值作为参数，外层Select语句中的employees.dependents对单个行没有约束。然而，在当前的例子中，employees.dependents落入CURSOR()表达式的子查询中并反向引用外层Select循环的employees限定符。它在CURSOR()开始查询行的嵌套表之前总是稳定在employees的单个行上。

若不用嵌套游标检索，我们要使用两次TABLE()形式，一次形成表积，另一次产生可以在其上对某个雇员计算家属的最大年龄的表。

```
select eid, d.pname.fname from employees e, table(e.dependents) d1
  where d1.age = (select max(d2.age) from table(e.dependents) d2));
```

例4.3.10 在例4.3.6中，我们计算了某个雇员家属数。我们也可以检索计算分别属于每个雇员的家属的多行结果：

```
select eid, cursor(select count(*) from table(e.dependents))
  from employees e;
```

嵌套游标扫描外层Select的每个雇员行的嵌套表并计算dependents表的行数。在输出时没有显示内层的表（如图4-17中的dep_ssno），因为内层表中没有每个家属的数据可显示。

作为选择，我们可以删除这里的CURSOR()并在选择列表中使用结果的标量子查询，一种高级SQL特征。

```
select eid, (select count(*) from table(e.dependents))
  from employees e;
```

最后，我们可以用表乘积消除嵌套。

```
select eid, count(*) from employees e, table(e.dependents)
  group by eid;
```

例4.3.10给出了查询每个雇员的家属数的不同方式。为了计算雇员的所有家属数，我们必须使用表乘积以消除嵌套，因为这是能够使我们无缝地跨越多个雇员行集合的唯一可能的形式。我们可以写：

```
select count(*) from employees e, table(e.dependents);
```

5. VARRAY数组类型

ORACLE的另一种汇集类型即数组类型，以VARRAY声明，代表“变长数组”。每个数组类型声明有一个名字，一个元素类型，一个VARRAY对象可以包含的最大元素个数（和varchar类型一样，VARRAY的实际大小不是正比于这个最大数，而是正比于其所含的实际元素个数）。不像嵌套表类型，VARRAY类型的元素有特定的次序。

例4.3.11 建立一个简单的电话号码本：每个人有4个以整数表示的电话扩展号的VARRAY。拨号人应该先尝试第一个扩展号，然后第二个扩展号，依次拨号。由于在第一个号后的扩展号应属于合作者，这种按次序拨号的规则是很重要的，以便合作者免受不必要的打扰。

```

create type extensions_t as varray(4) of int;
create table phonebook
(
    phperson person_t,
    extensions extensions_t
);

```

与嵌套表不同，VARRAY数据通常直接包含在表的行中。因此，在Create Table语句中不像例4.3.1那样需要专门的存储子句。在某些情况下，VARRAY数据可以存放在外部，但对此我们不做深入的讨论。同样，我们可以看到，在这里没有使用主键子句，因此在该表中同一个人可能对应两行。或者我们可以通过将下边的语句加入到Create Table语句中，在phperson列的ssno属性的基础上建立主键：

```
primary key (phperson.ssno)
```

图4-19给出了一个对John Smith有两个扩展号和对David Andrews有一个扩展号的小phonebook表。

phonebook	
phperson	extensions
person_t(123897766, name_t('Smith', 'John', 'P'), 45)	extensions_t(345, 989)
person_t(432112233, name_t('Andrews', 'David', 'S'), 32)	extensions_t(123)

图4-19 带整数extensions的ORACLE VARRAY列的phonebook表

例4.3.12 检索社会保障号为123897766的人的名字和VARRAY extensions。

```

select pb.phperson.fname, pb.extensions from phonebook pb
where pb.phperson.ssno = 123897766;

```

这将对每个人显示一行，带有如下标量VARRAY值的extensions，结果如图4-20所示。

phperson.fname	extensions
John	extension_t(345, 989)

图4-20 ORACLE对例4.3.12的查询的VARRAY列值的输出

注意，SQL语句实际上无法访问一个以VARRAY定义的对象的下标元素，如例4.3.12中的选择列表中出现pb.extensions[1]（你可以将VARRAY转换为嵌套表并访问单个行，但是行的相对次序无法保证）。除了显示整个聚集值让读者从左到右去计算以外，我们不能访问第一个扩展或其他任何编号的下标元素。当我们在第5章中考虑SQL程序语言接口时，从一行中检索到的VARRAY将被映射到一个编程语言数组中，只有那时才能访问带下标的VARRAY元素。

在一个查询中，TABLE()表同样可以应用于VARRAY值。

例4.3.13 在phonebook表中寻找社会保障号为123440011的人所拥有的扩展号。

```

select (select count(*) from table(pb.extensions)) from phonebook pb
where pb.phperson.ssno = 123440011;

```

或者使用下边的语句：

```
select count(*) from phonebook pb, table(pb.extensions)
  where pb.phperson.ssno = 123440011;
```

例4.3.14 列出具有扩展号104的phonebook表中的人。

```
select phperson from phonebook pb
  where 104 in (select * from table(pb.extensions));
```

有时我们需要将一个聚集类型转换为另一个类型或从一个行集合转换为汇集类型。例如，我们可能有一个person_t嵌套表作为一个列值并使用它去更新person_t的一个VARRAY类型列，或者将一个子查询中的关系数据填充一个汇集值。图4-21给出了汇集类型的CAST表达式的一般形式。下边的collection_type必须使用一个嵌套表（如dependents_t）或VARRAY类型（如extensions_t）替换。CAST表达式允许嵌套表或VARRAY的表达式或子查询被转换为另一个嵌套表或VARRAY类型值。这种使用CAST实现从一个VARRAY类型到嵌套表类型或从嵌套表类型到VARRAY类型的转换是非常有用的。

```
collection_cast_expression ::= -
  CAST(collection_valued_expr AS collection_type)
  |CAST((collection_valued_subquery) AS collection_type)
  |CAST(MULTISET(rowset_valued_subquery) AS collection_type)
```

图4-21 ORACLE中的CAST表达式的一般形式

在图4-21的最后一行中的MULTISET形式被用于包含一个返回我们希望转换为VARRAY或嵌套表的行集合的子查询。由于在INFORMIX中的多重集合(multiset)有不同的含义，我们将后边出现的传统关系查询返回的行集合被称为行集。

下边给出一个MULTISET的例子，该例子将所有家属的社会保障号收集到一个汇集类型中。

```
create type ssno_set as table of int;
select e.eid, cast(multiSet(select d.ssno from table(e.dependents) d
  where d.age < 16) as ssno_set) as dep_ssno
  from employees e;
```

这个查询将输出每个雇员eid和一个ssno_set()嵌套表值。可以看到，MULTISET()与TABLE()相反。MULTISET将一个行集变为汇集值，而TABLE()将汇集值变为行集。上边的查询消除家属嵌套，从中取出ssno值，并将它们嵌套为汇集值。

6. ORACLE中汇集的SQL语法

RDBMS厂商和SQL标准制定者希望将汇集类型包括在ORDBMS SQL的语法中，然而，他们面临着比最初想象更难解决的问题。他们的目标是通过扩展关系查询的语法来查询嵌套表的内容。显然，如图3-11所示的SQL-99扩展，在FROM子句中，跟在FROM后面的tableref子句就是一个推广语法以包括嵌套表的地方，ORACLE采用了这种方式。

图4-22的tableref语法定义包含两种关系形式（图3-11中的前两种方式：基本tablename形式和rowset_valued_subquery）和一个新的汇集形式。注意到tableref之后总跟着一个FROM，所以新的查询格式为FROM TABLE()。

```

tableref ::= {tablename | (rowset_valued_subquery)
              | TABLE (collection_expr)           -- collection-valued subquery, etc.
              [corr_name]                      -- "AS" keyword not allowed in ORACLE

```

图4-22 ORACLE中的tableref子查询形式的语法（与图3-11相比）

回顾图4-14的例4.3.2的显示格式：

```

dependents_t(person_t(322456776, name_t('Smith', 'Michael', 'J'), 8),
             person_t(123822332, name_t('Smith', 'Susan', 'R'), 12))

```

这样的语法也可以被用作嵌套表构造器从适当的类型表达式建立一个特定的嵌套表。类似地，

```
extensions_t(395,667)
```

是一个VARRAY构造器，通过它我们可以建立一个extensions_t类型的VARRAY。VARRAY构造器和嵌套表构造器都镜像了图4-5显示的ORACLE对象构造器的语法。这些汇集构造器以和对象构造器一样的方式用于插入和更新。

注意，根据图4-22，FROM TABLE(…)可把一个汇集表达式转换为一个行集。汇集表达式可以是以下指定一个嵌套表或VARRAY类型值的任意一种方式：

- 汇集值子查询：返回单个向量汇集值的子查询。
- 例如(select e.dependents from employees e where eid = 101)。
- 列或属性表达式。例如e.dependents。
- 产生一个聚集值的CAST表达式
- 返回汇集值的函数

7. ORACLE中的插入和更新

简单的插入和更新使用汇集构造器构成的新行和行的新列值。

例4.3.15 在employees表中插入一个John Smith雇员行，如图4-12所示。

```

Insert into employees values (101, person_t(123897766, name_t('Smith', 'John', 'P'), 45),
                               dependents_t(person_t(322456776, name_t('Smith', 'Michael', 'J'), 8),
                                             person_t(123822332, name_t('Smith', 'Susan', 'R'), 12)));

```

在phonebook表中为John Smith插入一行，如图4-19所示。

```

Insert into phonebook values (person_t(123897766, name_t('Smith', 'John', 'P'), 45),
                               extensions_t(345,989));

```

将phonebook表中的John Smith的扩展列表更新为345和999。

```
update phonebook set extensions = extensions_t(345,999) where eid = 101;
```

在ORACLE中，通过使用语法形式TABLE(nested_table_subquery)返回一个标量嵌套表值到一个普通表中，我们可以插入或更新一个已经存在的嵌套表中的元素。

例4.3.16 为雇员John Smith插入一个新的家属。

```

Insert into table(select e.dependents from employees e
                  where e.eperson.ssno = 123897766)
values (344559112, name_t('Smith', 'Diane', null), 0);

```

注意在嵌套表中插入新家属的这种能力是令人吃惊的。这里的Insert语句将添加到子查询

指定的某个雇员John Smith的dependents嵌套表的行，再将这些行的集合包装在一个嵌套表汇集容器中以完成插入操作。这种操纵嵌套表的能力是非常有用的。

接下来，我们将显示如何更新某个雇员的所有家属。这个例子将所有雇员的（小写字母表示的）姓改用大写字母表示。

```
update table(select e.dependents from employees e where e.eid = 101) d
  set d.pname.lname = upper(d.pname.lname); -- convert last name to uppercase
```

这里我们不能仅用一个Update语句来完成对所有雇员的家属的更新。游标只能用于Select语句，所以，我们不能在单个Update语句中使用游标来遍历所有人的所有dependents。■

尽管对插入或更新可以使用TABLE(subquery)来指定嵌套表，却不能用来指定VARRAY类型。因此，我们不能以例4.3.16的方式来插入或更新VARRAY中的单个元素。在例4.3.15中，我们已经看到如何用Update语句来替换单个phonebook行的扩展的整个VARRAY。

在下边的例子中，我们将看到在从另一个表汇集中选择家属集时如何针对一个特定的雇员的dependents汇集使用子查询。

例4.3.17 将people表的所有行插入John Smith的dependents表中。

```
insert into table(select e.dependents from employees e
  where e.eperson.ssno = 123897766)
    select * from people;
```

4.3.2 INFORMIX中的汇集类型

INFORMIX中有三种汇集类型：集合，多重集合和列表。给定任意一个类型 x_t ，我们可以将其变为汇集类型中的一个元素类型：SET(x_t)、MULTISET(x_t)和LIST(x_t)。INFORMIX允许集合的集合和列表的集合。我们将在本节中讨论集合和列表。多重集合与集合一样，在多重集合中集合是无序的元素汇集，但与集合不同，多重集合还允许重复元素。注意不要将INFORMIX中的多重集合汇集类型和图4-21中介绍的ORACLE中的MULTISET()形式相混淆。

在数学中，集合是一个无序的元素汇集；这些元素必须类型相同，且元素间无重复。SET类型可以以类型或表定义的形式出现在任意内部类型可以出现的地方。例如，一个表的单个列可能是一个集合类型。正如我们后面将要说明的那样，集合大致对应于ORACLE中的嵌套表。

列表是一个允许重复的有序元素汇集。例如，地址在不同的环境中有不同的行号，因此，一个字符串列表是保留它们的一种方便的方法。列表对应于ORACLE 中的VARRAY类型，但列表对元素的最大数目没有限定。

1. INFORMIX中的集合

为了与我们在ORACLE中使用的例子保持一致，我们将对employees表中的dependents列使用SET(person_t)类型，对phonebook表中的电话扩展号使用LIST(int)类型。同样，我们将对employees和phonebook使用传统关系表（不用类型表），但可以用类型表替代关系表。

例4.3.18 为雇员创建一个关系表，带有一个int类型的eid列、person_t行类型的eperson列和包含person_t元素集合的dependents列。

```

create table employees
(
    eid      int,
    eperson  person_t,
    dependents set(person_t not null)
);

```

在ORACLE中，我们首先为家属嵌套表定义一个dependents_t类型，再在Create Table语句中使用它。在INFORMIX中，我们不必命名汇集类型；我们只需要在Create Table语句中简单地使用SET来定义一个集合类型。注意在SET类型说明SET(person_t NOT NULL)中，关键字NOT NULL是必须的。

图4-23显示了例4.3.18中的employees表，其内容为两个雇员行，一个雇员带有两个家属，另一个仅带一个家属。

employees		
eid	eperson	dependents
101	row(123897766, row('Smith', 'John', 'P'), 45))	set{row(322456776, row('Smith', 'Michael', 'J'), 8), row(123822332, row('Smith', 'Susan', 'R'), 12)}
102	row(432112233, row('Andrews', 'David', 'S'), 32)	set{row(565534555, row('Shaw', 'David', 'M'), 3)}

图4-23 带行类型person_t集合列dependents的INFORMIX employees表

例4.3.19 检索eid为101的雇员的所有家属的集合。

```
select dependents from employees where eid = 101;
```

这将对该雇员显示一行，带有如图4-24所示的汇集值的形式列出的家属。类似地，查询

```
select * from employees;
```

将对每个雇员显示这样的家属的集合和其他的列。

<pre> dependents set{row(322456776, row('Smith', 'Michael', 'J'), 8), row(123822332, row('Smith', 'Susan', 'R'), 12)} </pre>
--

图4-24 例4.3.19查询带集合值的列INFORMIX输出

INFORMIX同样允许我们在一个集合中使用CARDINALITY()格式计算元素数目。

例4.3.20 计算雇员101的家属数。

```
select cardinality(dependents) from employees where eid = 101;
```

检索带有6个以上家属的雇员的eid。

```
select eid from employees
where cardinality(employees.dependents) > 6;
```

与ORACLE不同，INFORMIX支持集合是否相等的测试，下面是一个例子。

例4.3.21 找出所有带有相同dependents集合的雇员，每次列出一个雇员对。

```
select e1.eid, e2.eid from employees e1, employees e2
  where e1.eid < e2.eid and e1.dependents = e2.dependents;
```

■

在INFORMIX中的IN谓词被用于测试一个汇集中的某个特定的元素。dependents集合中的一个特定的元素是一个person_t行类型对象。我们可以使用行构造器表达式ROW(...)并将结果转换为一个适当的类型来构造这样的对象，如4.2.2节所述。

例4.3.22 通过对person_t对象的匹配，列出以David Shaw为家属的雇员的eid。

```
select eid from employees
  where cast(row(565534555,cast(row('Shaw', 'David','M') as name_t),3) as person_t)
    in employees.dependents;
```

■

从9.2版本开始，INFORMIX使用汇集驱动表的形式TABLE()，其工作方式与ORACLE中的TABLE()相同，如图4-22所示。INFORMIX的TABLE()格式作用于由一个表达式指定的参数汇集值，或作用于返回单个汇集值的一个子查询，并将其转换到一个由汇集元素组成的行表中。下面是一些例子。

例4.3.23 列出家属社会保障号为322456776的雇员的eid。

```
select eid from employees e
  where 322456776 in
    (select d.ssno from table(e.dependents) d);
```

■

例4.3.24 显示eid为101的雇员的家属的所有社会保障号（和ORACLE中的例4.3.7相比较）。

```
select d.ssno from
  table (select e.dependents from employees e where e.eid = 101) d;
```

■

这里的子查询选择雇员101的dependents嵌套表，然后TABLE()形式将其打平到person_t对象表中。在这个例子中，该表被赋予别名d，选择列表中的d.ssno选择要显示的该表的ssno列。

例4.3.25 显示employees表中的所有行的eid值，以及对于每个eid，该雇员年龄小于16岁的家属的数目。

```
select e.eid, (select count(*) from table(e.dependents) d
  where d.age < 16)
  from employees e;
```

这里的SELECT在employees表上设定一层循环，子查询在某个雇员的家属的嵌套表上设定第二层循环。

■

注意，INFORMIX和ORACLE一样允许在选择列表中返回单个值的子查询。这种能力已被用于INFORMIX的例4.3.25中和ORACLE的例4.3.10中。然而，通过使用ORACLE的CURSOR()语法或例4.3.8的专门的表乘积，我们可以列出所有雇员的年龄小于16岁的家属的社会保障号。这种表中表类型的扫描不能在社会保障号放在输出不同行的单个INFORMIX SQL语句中执行。这可以通过搜集每个雇员的家属所有社会保障号到一个汇集对象中，然后再显示这个汇集对象，如图4-21中的ORACLE查询所示。

```

select e.eid, multiset(select item ssno from table(e.dependents)
    where age < 16)
from employees e;

```

另外，我们也可以使用INFORMIX的用户定义函数和其他形式的过程化程序来实现上述查询。我们将在4.4.2节讨论INFORMIX的用户定义函数。

2. INFORMIX列表

在INFORMIX汇集类型中，列表和多重集合很相似，它们都有操纵元素的IN和CARDINALITY，并使用TABLE()形式将汇集转换为可以被查询的表。因此，所有的汇集类型在查询中的使用也非常相似。列表与集合的差异在于列表中的元素是有序的，且像ORACLE中的VARRAY一样允许元素重复。但是，当一个列表被TABLE()转换时，其次序丢失。

例4.3.26 像ORACLE例4.3.11一样，建立一个phonebook表。其中每个人有一个整数电话扩展号列表，次序是重要的：第一个号是主扩展号，紧跟着是第二扩展号，依此类推。

```

create table phonebook
(
    phperson person_t,
    extensions list(int not null)
);

```

注意这里没有主键定义子句，所以在表中同一个人可能对应两行。在ORACLE的例子中，我们可以将phperson.ssno用作主键，但是INFORMIX不支持那样的定义。当然，如果有必要，我们可以增加一个顶层ssno列并将其定义为主键。 ■

图4-25像图4-19一样描述了对于John Smith有两个扩展号和对于David Andrewa有一个扩展号的小phonebook表，一个扩展345是Smith的主扩展号而898是次扩展号。

phonebook	
phperson	extensions
row(123897766, row('Smith', 'John', 'P'), 45)	list{345,989}
row(432112233, row('Andrews', 'David', 'S'), 32)	list{123}

图4-25 带整数extensions列表值列的INFORMIX phonebook表

例4.3.27 检索年龄大于30的所有人的名和扩展号列表。

```

select phperson.phname.fname, extensions from phonebook
    where phperson.age > 30;

```

这将为每个人显示一行，带有扩展列表作为单个汇集值。参见图4-26，其中的列表构造器用大括号{}而非ORACLE的VARRAY构造器使用的括号格式。

fname	extensions
John	list{345,989}
David	list{123}

图4-26 例4.2.27的INFORMIX查询的带列表值的列的输出

我们可以广泛地对列表使用包括相等测试在内的可作用于集合的操作，但我们必须注意次序问题。

例4.3.28 找出电话本中的社会保障号为123897766的人的扩展号的数目。

```
select cardinality(extensions) from phonebook
  where phperson.ssno = 123897766;
```

列出扩展号为123的人：

```
select phperson from phonebook
  where 123 in extensions;
```

列出具有相同的扩展列表的人对：

```
select pb1.phperson, pb2.phperson from phonebook pb1, phonebook pb2
  where pb1.extensions = pb2.extensions and pb1.phperson <> pb2.phperson;
```

由于我们仅要求phperson值行类型不等，这里的输出对每个对显示两次。如果我们知道社会保障号是非空的，则我们可以使用pb1.phperson.ssno <> pb2.phperson.ssno代替查询中的不等条件来消除重复。 ■

例4.3.29 检索扩展号为123且无其他扩展号的人。

```
select phperson from phonebook
  where phperson.extensions = list{123};
```

这里，我们使用列表构造器来定义包含单个整数常量的列表。从INFORMIX 9.2起，“list(123)”两边的双引号可以省略。 ■

3. INFORMIX中的汇集的SQL语法

INFORMIX中的汇集值表达式具有图4-27所示的各种形式。这里没有在表达式中实现汇集合并的操作，比较图4-27和图3-16的数字表达式的定义。

汇集表达式	例子
1. val: 常数或变量	"set('a', 'b')", "list{1,2}", :citylist
2. 汇集构造器表达式	set('a', 'b' 'c'), list{1+5, 2*age}
3. 列名	dependents
4. 点分表达式	ph.extensions, a.b.mycollection
5. (子查询)返回一个汇集	(select dependents from employees where eid = 101)
6. case 表达式	case when cardinality(extensions) > 0 extensions else list {1000}

图4-27 INFORMIX的汇集值表达式 (collvexpr) 定义

在INFORMIX查询中，任意汇集值表达式均可用于选择列表中，如等值比较谓词或IN谓词。图4-28给出了对SQL的相关语法的扩展。注意CARDINALITY()不能以一般的集合值表达式为参数，它要求汇集类型的列名。

例4.3.30 我们曾提到若在表的创建时缺乏主键子句意味着对应同一个人可能有重复行出现。下边的语句检索重复的人。

```

select pb.phperson from phonebook pb
  group by pb.phperson
  having count(*) > 1; -- remove unduplicated people

  CARDINALITY(colname)           -- an extension to numexpr
  IN collvexpr                   -- an extension to the IN predicate
  TABLE(collvexpr)              -- in place of a tablename in FROM ...

```

图4-28 INFORMIX SQL汇集的一般形式

这个查询可以作用于整个person对象，但不能将相应的查询用于pb.phperson.ssno；INFORMIX在9.2版中的GROUP BY语句不完全支持对字段的操作。

4. INFORMIX中的插入和更新

插入和更新需使用汇集构造器将汇集值放置到表中。在dependents表的例子中，我们有一个person_t行类型。

例4.3.31 在phonebook表中为图4-25的John Smith插入一行。

```

insert into phonebook values
(cast(row(123897766,cast(row('Smith','John','P') as name_t),45) as person_t),
list{345,989});

```

在employees表中为图4-25的John Smith插入一个雇员行。

```

insert into employees values (1,
cast(row(123897766, cast(row('Smith','John','P') as name_t),45) as person_t),
set{row(322456776, row('Smith','Michael','J'),8), --NOTE NO CASTS HERE
row(123822332, row('Smith','Susan','R'),12)} );

```

在版本9.2中，集合构造器SET()中的行构造器ROW(...)可被转换为适当的行类型name_t和person_t。

将phonebook表中John Smith的扩展列表中的扩展号更新为345和999。

```
update phonebook set extensions = list{345,999} where eid = 101;
```



在例4.3.16中谈到ORACLE语法时，我们对雇员John Smith插入一个新家属，并将家属的所有姓氏变为大写。在交互式的INFORMIX中，我们不能对集合元素执行这样的操作；而必须写一段过程代码来完成上述功能。我们将在4.4.2节讨论这一问题。

4.3.3 汇集类型小结

汇集类型和数据库产品 我们看到ORACLE和INFORMIX的汇集类型比4.2节中的用户定义类型有更多的差异。两种产品都支持真正的汇集类型，即允许违反关系规则I并且允许一个行中的某一列上出现任意类型的多重数据对象。二者都允许对转换为表的形式的标量汇集值的查询。除此以外，其他都是不同的。ORACLE在嵌套表中有一个一级汇集类型并且要求在被转换为嵌套表前有一个二级的VARRAY汇集类型。INFORMIX以一种更统一的方式来处理集合、列表和多重集合，并且它们都可以以TABLE()格式来查询。ORACLE不允许嵌套表中出现嵌套表，而INFORMIX允许集合中的集合及其他方式的嵌套。ORACLE期望用户命名所有的汇集类型，INFORMIX却不对汇集类型命名，而是使用诸如SET(elementtype)的

通用形式作为集合类型描述符。ORACLE通过特殊的CURSOR()语法来显示一个表中的每一行的汇集中的所有元素，而INFORMIX不具有这样的功能。

汇集和数据库设计 不同产品间的差异意味着如果你在数据库设计中使用汇集，则将不得不费力地把你的应用程序从一个产品移植到另一个产品。此外，在决定对某种数据使用汇集类型时必须很小心。如在我们研究过的employees表中，每一行有一个dependents标量的汇集值，对于任何一个单独的行都可以查询它们。然而，所有dependents的集合被分解为许多孤立的表，不易操作。在ORACLE中，我们可以使用表乘积来扩展一个表以使其包含每个集合元素的行，但INFORMIX中没有相应的工具。在两种情况下，我们都必须通过父表行来访问汇集元素。这样，对好的数据库设计来说，不把重要的对象归入存储在汇集类型中是很重要的。只有父表的属性而没有独立的属性存储在汇集中。ORACLE允许包含REF的集合，INFORMIX在将来的版本中也将允许REF的出现。这样，将来实际的对象将可以出现在一个独立的对象表中并被其他对象经过REF在汇集值中引用。

汇集和SQL-99 SQL-99工作组在汇集语法上摇摆不定。每个新的版本在这方面都有很大的变化，在最终的版本中支持的汇集类型只有数组，准备向其他汇集类型扩展。在SQL-99最新版中的表引用有一种新的格式UNNEST()，其作用像ORACLE和INFORMIX中的TABLE()形式。然而，数组并不是Core SQL-99的一部分。

4.4 过程SQL、用户定义函数和方法

自3.11节起，过程性语言被当作一组完成某项任务的有序指令代码构成的程序，然而，非过程性语言，如交互式的SQL语言，要求期望的任务被一次性地描述。非过程性在SQL被引入时被当作SQL的一个比易用性更重要的特征来介绍。但是，正如我们在3.11节中指出的那样，非过程性语言缺乏图灵机的表现力（也称为计算完整性），这意味着非过程性语言可能无法完成相当简单的任务（如按照大量可能格式中的某一种方式打印客户报表）。

这个问题在1978年Sybase SQL Server产品引入了过程性SQL语言，称为T-SQL（事务SQL）时讨论到。从那时起，其他多数的其他厂商都在其产品中引入了各自的过程性SQL语言，如INFORMIX的SPL和ORACLE的PL/SQL。过程性SQL支持内存驻留变量、条件和循环结构（IF ... THEN ... ELSE, WHILE和FOR）、过程和函数及在一个程序中执行SQL语句的能力等。在过程性SQL中写的函数可以以内部函数的方式用在非过程性SQL语句中（如选择列表表达式）。

过程性SQL在客户机-服务器应用程序中具有改进性能的潜力。这种应用程序通常用C或Java语言编写，并分别运行在通常具有不同的处理器的客户机和服务器上。SQL语句在服务器上执行，性能改进的事实来源于过程性SQL可以像普通的SQL一样在服务器上执行，而不需要经常和客户交互，因此减少了客户和服务器间经常交互的时间开销。一个用过程性SQL编写的函数在服务器上执行时被称为存储过程，由Create Function语句记录在服务器上的数据库目录表中，即使是函数代码也记录在目录表中。因此，我们可以说函数被存储在服务器上。过程性SQL语言在SQL-99中被标准化为SQL-99/PSM（Persistent Stored Module）。

在第7章中，我们将考虑一种称为触发器的机制，触发器是可以在任意事件（如插入行到一个表中）发生时执行的过程性SQL语句块。触发器用于实现定制的约束或缺省行为。

用户定义函数(UDF)是用过程性SQL（或用C或Java）写的函数，可以像交互式SQL中的内部函数或数据库系统的过程性SQL语言一样被调用。一些用户定义函数不是对象-关系

的，就像计算数字公式一样的简单。而其他可能以对象或汇集作为参数，或从函数中返回对象或汇集给调用者。在本节中，我们将讨论如何用PL/SQL (ORACLE) 和SPL (INFORMIX) 实现UDF。

方法是与用户定义类型一起定义的用户定义函数。由于具有定义方法的能力，ORACLE 对象比较接近于面向对象的编程语言中提供的某种对象。然而，我们将看到，无方法的对象也是有用的。INFORMIX行对象没有方法，但使用行类型参数的用户定义函数具有同样普遍的能力（实际上更灵活）。我们将介绍如何使用INFORMIX中的UDF提供具有与相应的ORACLE中提供的相同对象操作的行类型。

4.4.1 ORACLE PL/SQL过程、用户定义函数和方法

我们将在下一小节中简单介绍ORACLE PL/SQL，然后演示如何使用它来实现UDF和方法所需的程序逻辑。希望全面了解程序语法的读者可以参考《*ORACLE8 PL/SQL User's Guide and Reference*》(《ORACLE PL/SQL用户指南》)和《*ORACLE Server Application Developer's Guide*》(《ORACLE服务器应用开发指南》)，参见本章最后的推荐读物部分。在后面的章节中，我们假定读者已经熟悉C语言的一些概念。

1. PL/SQL: ORACLE 的过程性SQL语言

PL/SQL是交互式SQL的过程化扩展。它提供了一种执行程序逻辑、声明内存变量和执行循环条件，且同时可以在程序中的任何地方使用SQL访问数据库的方式。然而，这里的SQL语句，特别是Select语句因查询结果可能必须作为内存变量存储而变得复杂了：一个检索多行的Select语句不可能仅仅向用户显示结果。PL/SQL内存变量同样可被用于SQL的search_conditions中出现的常量的位置。

一个PL/SQL程序块由三部分组成：定义内存变量的DECLARE部分，紧跟着出现可执行语句的BEGIN-END部分，在所有的执行语句之后和END之前可出现第三部分EXCEPTION部分。我们将从一个执行简单的计算并将答案保存在名为result的表中的程序示例开始。

例4.4.1 写一段PL/SQL程序将从1到100的整数相加。

```

declare
    i      integer;                      -- local variable without an initial value
    total  integer:= 0;                   -- local variable with an initial value
begin
    for i in 1..100 loop
        total := total + 1;
    end loop;
    insert into result(rvalue) values (total); -- insert answer into database table
end;

```

你可能输入这段代码并在SQL*Plus下执行它，像Create Type语句一样，必须在其后加一个只带/的最后一行。注意这段程序将执行并不输出结果给用户，但是，该程序的成功执行将输出信息“PL/SQL procedure successfully completed.”。我们假定已经建立了带有单个整型列rvalue的result表。

例4.4.1显示了PL/SQL的几个重要特征。这段程序块中出现了DECLARE关键字和BEGIN、END关键字，定义相应的程序块的部分。注释与SQL的方式一样，以双短线(--)开始。赋值操作是:=，而等号(=)用作比较，这与标准的SQL相同。简单的循环可以用如下的结构建立：

```

FOR counter IN [REVERSE] lower_bound..higher_bound LOOP
    -- sequence of statements
END LOOP;

```

在FOR LOOP和END LOOP之间是一到多条语句构成循环体；每条语句像C和Java一样，以分号结束。在例4.4.1中仅出现了一条赋值语句。递归过程从按从下界到上界的次序进行，除非出现表示按相反次序执行的REVERSE关键字。

循环也可以因某种选定条件而结束，而无需到达计数限制才结束。这样的循环格式如下：

```

LOOP
    -- sequence of statements
    -- exit loop at any point with: EXIT WHEN condition;
END LOOP;
-- flow of control resumes here after EXIT WHEN occurs within loop

```

例如，我们可以重写例4.4.1的程序如下：

```

declare
    i      integer:= 0; -- local variable, with an initial value
    total  integer:= 0; -- local variable, with an initial value
begin
    loop
        i := i + 1;
        total := total + 1;
        exit when i >= 100;
    end loop;
    insert into result(rvalue) values (total); -- insert answer into database table
end;

```

在执行上边的程序后，用户可以从result表中选择来显示的答案。这些程序将只计算从1到100的整数。一种更灵活的方式是将逻辑放在PL/SQL函数中。PL/SQL函数可以像sqrt()和substr()一样作为内部函数使用。

例4.4.2 写一个PL/SQL函数对任意的整数n，从1到n的整数求和。并从SQL调用它。

```

create function sum_n(n integer) return integer is
    i          integer;
    total      integer:= 0;
begin
    for i in 1..n loop
        total := total + i;
    end loop;
    return total;           -- return result to SQL or PL/SQL caller
end;

```

这个函数程序段的DECLARE部分在Create Function之后，没有出现关键字DECLARE。为了在SQL*Plus中运行必须在其后加一个只带有/的新行。如果定义成功，则你将看到确认“Function created.”，否则，你可以使用SQL*Plus命令：

```
show errors
```

ORACLE将显示在编译时出现的错误和出现错误的行号。 ■

在形如例4.4.2的Create Function语句执行后，函数将像表或视图一样作为长期对象存在。可被带任意整数表达式的参数的SQL语句调用，例如：

```
select sum_n(10) from orders;
```

这个选择语句很奇怪。它实际上不使用orders表——在这里使用该表仅仅是为了满足一个Select语句应在FROM子句中有一个表的语法要求。然而，这个Select语句将以列表的方式返回多个值。这样，orders表的每一行中将对应一个值，这是一个意外的结果！为了讨论这个问题，ORACLE提供了一个仅有一行一列的专门的表dual。我们可以这样写：

```
select sum n(10) from dual; -- use Oracle's provided table (not actually accessed)
```

当然，我们可以在放入函数sum_n中的表达式中使用实际的列值：

```
select qty, sum n(qty), 2*sum_n(qty*qty) from orders where aid = 'a04';
```

这里的sum_n()是一个标量函数，不能与诸如sum()和count()的SQL集合函数相比。我们定义的函数sum_n()从每一行中取一个值，并返回一个值，而集合函数sum()扫描一组值，返回一个值。上边最后一个Select语句将对代理a04的每个订单返回一行。

例4.4.2介绍了Create Function语句和RETURN语句。PL/SQL像SQL中的命名列一样（见3.9节中的“表达式、谓词和搜索条件”）对内存变量使用同样的数据类型语法，并像访问相同的内部函数，如sqrt()和substr()。附录A.3中给出的任意ORACLE列类型都可以当作任一程序DECLARE部分的内存变量类型使用。

从SQL调用的一个函数被假定不对数据库作任何变动；而仅仅计算一个值。在例4.4.1中，我们执行了一个表插入操作。如果我们在例4.4.2中有这样的插入，而我们试图从SQL语句调用函数，则我们将看到运行错误“Function sum_n does not guarantee not to update database.”。当然，我们可以用如下代码从程序块执行函数：

```
x := sum_n(100);
```

Select语句不改变任何数据库中的数据，所以它可以用于SQL的函数调用语句中，如下面的例子所示。

例4.4.3 写一个查找给定cid的customers表中的cname和city，并以诸如“Tiptop (Duluth)”的格式返回信息。

```
create function namecity(custid char) return char is
    custname char(20); -- maximum size, gets trimmed back below custcity char(20);
begin
    select cname, city into custname, custcity from customers where cid = custid;
    return rtrim(custname)||" ("||rtrim(custcity)||')'; -- trim extra space at right
end;
```

注意在第一行(custid char和return char)中为一个字符串类型赋长度是错误的，因为系统将调整参数并返回任意长度的值。可以从SQL调用namecity函数。 ■

例4.4.3使用了返回单个行的选择。通过游标，PL/SQL同样可以处理多行的选择。但这不属于本文的讨论范围。

在许多语言中(如C语言)，在函数代码内可以安全地改变参数变量，因为这些语言具有传值调用的方式，即在被调用函数中的参数仅仅是在调用语句中使用的参数的一个副本：参数的初值实际上不会改变。然而，在PL/SQL中，参数不是这样的副本，而是函数调用中使用的实际的参数值，编译程序拒绝试图改变它的任何代码。如果你试图改变在函数中代表参数的变量，Create Function语句将失效。这样，必须学会在函数中不用参数变量作为存储中间计算结果的地方。例如，考虑下边递增形参x的函数：

```

create function increment(x int) return int is
begin
    x := x + 1;      -- change x, a parameter variable **ERROR - WON'T COMPILE**
    return x;        -- (we should have written "return x+1")
end;

```

这个函数导致ORACLE错误“Function create with compilation error.”，如果试图使用它，则将看到错误“Package or function INCREMENT is in an invalid state.”。

有一种可以使参数变量是可更新的，我们可以在Create Function语句中用increment(x in out int)代替increment(x int)。但是，不能在任意SQL语句中使用这一函数，因为SQL表达式(函数是表达式的部分)并不支持改变数据。因此，如果在ORACLE的SQL语句中使用这样的函数，系统将可更新的参数解释为可能改变它的参数而返回一个不同的错误信息。与可更新参数相对的参数可用于Insert和Update语句的值的表达式和查询表达式中。我们对用户定义函数要习惯仅用只读的参数，以保证该函数对交互式SQL和PL/SQL均可用。

由于参数变量必须是只读的，我们需要定义局部变量来保存中间结果和最终结果。我们可以使用对象构造器创建局部对象，如例4.4.4中的局部变量nper：

例4.4.4 写一个将person_t对象的age增加1的函数。具有参数x的age增加1后的返回值回到局部person_t对象类型变量中。

```

create function inc_age(person_t x) return person_t is
    nper person_t:= person_t(x.ssno, x.pname, x.age); --clone x to local object nper
begin
    nper.age := x.age + 1; --change age in LOCAL VARIABLE, not PARAMETER
    return nper;
end;

```

这里需要使用对象构造器person_t()来创建用于更新的局部对象。我们通过使用x对象的属性(它的ssno, pname和age)来描述person_t()的所有参数，以使新对象是x的一个克隆。我们不能仅仅在右边使用对象构造器person_t(x)；我们需要分别指定这些属性。赋值nper := x是可行的，但是，在此前必须调用构造器person_t()来创建nper变量。

为使用这个函数，我们显示更新过年龄的people表中的所有person_t对象：

```
select inc_age(value(p)) from people p;
```

这里的表数据不会被更新；增加操作仅影响显示值。为获得增加了的年龄到表中，我们需要使用下边的Update语句：

```
update people p set p = inc_age(value(p)) where p.age < 40;
```

当我们在任何表达式(该Update语句的SET P=后面的子句)中计算行对象时，必须使用VALUE()的格式：value(p)。但是在等号前的Update语句中，我们使用表的别名来代表需要更新的行对象。

我们介绍的两种FOR循环类型如图4-29所示。在计算循环时，循环变量是一个整型变量(在代码的开始声明)，它以lower_bound为初值，每次循环增加1，直至带有值higher_bound的循环通过终止。在图4-29中，我们也介绍了新的形式：IF-THEN-ELSE，该形式可用于PL/SQL中。

2. ORACLE中用PL/SQL实现方法

方法是属于某种对象类型的函数。对方法的每次调用是可以引用该类型的对象的一次调

用。当程序执行到方法代码时，被引用的对象简单地称为self，它是一个对象类型变量。

```

FOR counter IN [REVERSE] lower_bound..higher_bound LOOP
    -- sequence of statements
END LOOP

LOOP
    -- sequence of statements
    -- exit loop at any point with: EXIT WHEN condition;
END LOOP
    -- flow of control resumes here after EXIT WHEN occurs within loop

IF condition THEN
    statements
[ELSE
    statements...]
END IF

```

图4-29 ORACLE PL/SQL中的Loop语句和IF语句格式

例4.4.5 为点和矩形分别定义对象类型point_t和rectangle_t。为rectangle_t定义方法area()和inside(point_t)。area()方法返回矩形对象的面积(不需要参数)，inside(point_t)方法用于判断一个点是否在矩形对象内或是在矩形对象的边缘上，是则返回1，否则返回0。

```

create type point_t as object
(
    x int,                      -- horizontal coordinate of the point
    y int                         -- vertical coordinate of the point
);

create type rectangle_t as object
(
    pt1 point_t,                 -- lower left-hand corner of rectangle
    pt2 point_t,                 -- upper right-hand corner of rectangle
    member function inside(p point_t) -- method to test if point is inside rectangle
        return int;                -- return 1 if point inside, else 0
    member function area           -- method for area of rectangle
        return int;                -- area value will always be an integer
);

```

例4.4.5中的rectangle_t对象包含内两个方法area和inside，它们是该对象类型的部分，在意义上与pt1和pt2属性相同。方法也称为成员函数(Member Function)，这是因为它们定义的语法，方法还没有提供任何代码执行必要的逻辑，例如，将矩形的长度乘以宽度来计算面积。Create Type语句自身仅包含可以调用方法的代码所需的信息，这就像C原型或Java接口定义一样。代码出现在另外的独立的语句中，即例4.4.8中才出现的Create Type Body语句。现在，我们看到的rectangle_t类型的Create Type语句中每个方法包含以下声明元素：

- 方法名：area和inside。
- 方法的返回类型：在两个例子中都为整型。
- 方法的参数：area没有，inside有一个命名为P的point_t对象。

如果你不熟悉对象编程，你可能想知道area函数没有参数是如何能够存在的。原因是所有的方法都属于一个特定的对象类型，且仅可以使用该类型的对象来调用。这里的area方法属于rectangle_t类型，只能被实际的rectangle_t对象调用。我们可以想象每一个rectangle_t对象仅有一个面积值，就像方法计算的数据属性一样。发现方法area就像找对象pt1的属性。这样，如果rect1是rectangle_t类型的对象，rect1.pt1的值将代表rect1的pt1属性，而rect1.area()的值将代表rect1的面积。当然，rect1.area()的值通过计算其他属性值的方法得到，但是，计算过程对调用者来说是不可见的：area()就像对象的属性。另外，inside方法用于处理已知一个point_t对象和rectangle_t对象（如pt1）并且想检验点是否在矩形中的情况。因此，矩形的inside方法必须传递一个点作为参数，它将以rect inside(pt1)的格式执行。

例4.4.6 创建point_t和rectangle_t类型的对象表，并插入一些值。

```
create table points of point_t (primary key (x, y));
create table rects of rectangle_t (primary key (pt1.x, pt1.y, pt2.x, pt2.y));
insert into rects values (point_t(1,2),point_t(3,4));
insert into rects values (point_t(1,1),point_t(6,6));
insert into points values (2,3);
insert into points values (1,4);
insert into points values (4,4);
```

■

我们看到例4.4.6中方法的增加对4.2.1节介绍的创建对象的能力并没有任何改变。和前边一样，我们可以在类型point_t和rectangle_t之外构造表对象并在Insert语句的VALUES的形式中使用对象构造器point_t()来插入建立在常量以外的矩形或点。类似地，我们也可以在Select语句中引用各种属性。但是，如下例所示，我们可以通过使用方法area和inside提供的更多功能。

例4.4.7 我们从检索rects表中所有矩形的面积和判断点(4, 2)是否在每个矩形中的选择查询开始。

```
select value(x), x.area() from rects x;
select value(x), x.inside(point_t(4,2)) from rects x;
```

在图4-4中，选择列表中的value(x)以对象构造器的方式检索作用于rects的别名x的整个对象（行）。在第一个查询中，x.area()调用area方法，并为rects中的每一行返回值，正如若x.pt1在选择列表中代替x.area()，则它将被返回。在第二个查询中，x.inside(point_t(4,2))将以坐标为4和2的point_t参数调用inside方法来判断是否该点在别名为x的rects的每一行中。

下边，我们写一个查询来列出points表中在rects表的某个矩形内的所有的点。

```
select distinct p.x, p.y from points p, rect r
where r.inside(value(p)) > 0;
```

这就像写select distinct value(p) ...一样，但是，ORACLE不支持对整个对

象的DISTINCT 处理、GROUP BY和ORDER BY语句，除非使用专门的排序方法^Θ。当我们需要比较整个对象时，我们要避免这种问题。

有一个点p在所有的矩形中意味着没有矩形r满足r.inside(p)=0，如下面的查询所示。

```
select value(p) from points p where not exists
  (select * from rects r where r.inside(p) = 0);
```

最后，为了解释Update语句，我们对所有面积小于24的矩形，将x和y值都加1来改变pt2。

```
update rects r set pt2 = point_t(r.pt2.x+1, r.pt2.y+1)
  where r.area() < 24
```

这里必须使用表的别名，原因是属性必须通过从表的别名开始的点号来引用的规则。 ■

迄今为止所涉及的Create Type语句的语法细节参见图4-30。我们已经在4.2.1节中见过Create Type语句，但现在增加了MEMBER FUNCTION语句。

```
CREATE TYPE typename AS OBJECT
  (attrname datatype [, attrname datatype ...])
  MEMBER FUNCTION methodname [(param type [, param type ...])]
    RETURN datatype;
  [, MEMBER FUNCTION methodname [(param type [, param type ...])]]
    RETURN datatype ...);
```

图4-30 迄今为止所涉及的Create Objct Type语句的ORACLE格式

例4.4.8 为实现rectangle_t对象类型的area()和inside(point_t)方法，我们使用一个新的类型语句Create Type Body语句。

```
create type body rectangle_t as
  member function area return int is      -- all logic is in PL/SQL
  begin
    return (self.pt2.x-self.pt1.x)*(self.pt2.y-self.pt1.y); -- The calculated area
  end;
  member function inside(p in point_t) return int is
  begin
    if (p.x >= self.pt1.x) and (p.x <= self.pt2.x) and  -- inside on x coordinate
      (p.y >= self.pt1.y) and (p.y <= self.pt2.y) then -- inside on y coordinate
      return 1;  -- p is inside rectangle (including the boundary)
    else
      return 0;  -- p is outside rectangle
    end if;
  end;
```

这个Create Type Body语句和前面同种类型的Create Type语句相关。rectangle_t（参见例4.4.5）提供了这里定义的方法的实现代码。注意这里的内部变量self是如何被用于代表方法所操作的rectangle_t类型的对象的，而inside(p in point_t)则声明一个由名p

^Θ 也就是说，我们需要定义一种“排序”方法来告知系统如何比较两个对象。参见《ORACLE8 Server Application Developer's Guide》。

所引用的inside方法的类型参数。

self对象作为成员函数的不可见参数而存在。如前所述，参数是只读的，self也一样。当我们需要以对象来存储结果时，我们应该创建局部对象，别忘了创建新的局部对象要调用对象构造器，关于对象构造器的用法参见例4.4.4。

图4-31给出了Create Type Body语句的语法细节。注意可选的OR REPLACE语法的使用，带有这个选项在再次创建一个类型之前可以不用Drop Type Body。

```

CREATE [OR REPLACE] TYPE BODY type [AS|IS]
  MEMBER FUNCTION methodname [(param type [{, param type, ...}])]
    RETURN type IS
      BEGIN                      -- BEGIN PL/SQL statements
        implementation statements
      END;                      -- END PL/SQL statements
    {MEMBER FUNCTION methodname [(param type [{, param type, ...}])]
      RETURN type IS
      BEGIN                      -- BEGIN PL/SQL statements
        statements
      END;                      -- END PL/SQL statements
      ...
    }
  END;

```

图4-31 迄今为止所涉及的Create Type Body语句的ORACLE格式

假定我们一直在讨论的矩形表示地球表面的区域。我们已经有了大量这样的矩形，而想要找到覆盖某个给定的点的集合(也就是表)的面积最小的矩形，如图4-32所示。

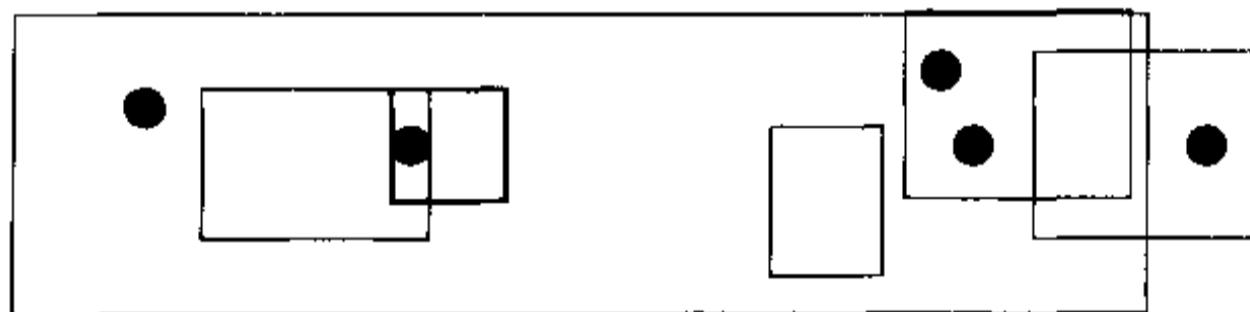


图4-32 例4.4.9点和矩形

例4.4.9 如果我们只想找到覆盖每个点的矩形的面积（而不需显示这些矩形），则只需使用下边的语句：

```

select p.x, p.y, min(r.area()) from rects r, points p
  where r.inside(value(p)) > 0
  group by p.x, p.y;

```

注意，这里的GROUP BY子句的每个组中正好一行。因为坐标(x, y)构成主键。然而，我们仍然需要GROUP BY子句来保证最小集合函数min()每次运行一个点而不是整个点的集合。

我们同样可以使用相关子查询对每个点计算最小面积的矩形并在外层Select语句中使用它。

```

select value(p), value(r), r.area() from rects r, points p
  where r.area() =
    (select min(r1.area()) from rects r1 -- minimal area rectangle...
     where r1.inside(value(p)) > 0)   -- ...containing the point p

```

```

        and r.inside(value(p))
        order by p.x, p.y;

```

外层Select可能对某些点发现多于一个相同最小面积的矩形。ORDER BY语句将这些矩形收集到一起，并以一种有用的次序输出所有结果。 ■

一个纯对象论者可能会说不直接引用对象属性而只使用方法来访问所有对象的Select语句“正好是面向对象的”参见4.2.3节对面向对象的讨论。在这种意义上，例4.4.9的最后的查询正好是面向对象的，因为仅在最后的ORDER BY语句中有对对象属性的直接引用，并且实际上不用这一子句，我们也可以得到所有数据^Θ。在面向对象术语中，属性被称为是提供了对象的内在表示，不能从对象外直接访问。

然而，在ORSQL中，我们通常不会限制自己通过方法来访问表，因为我们目前所用的查询的自然模型是知道表的列名，并使用这些列来搜索所需的数据。由于没有等价的模型可以创建并使用户清楚可以以通用的形式访问表所必需的所有方法，在必要的时候我们还是继续存取内部的列。当然，我们可以创建访问这些列的方法并提供适当的面向对象的查询，但是，这种直接由列值到方法的映射没有什么作用。

在本节结束之前的一个练习将显示例4.4.9的查询可以不用方法，而用交互式的SQL来实现，但是，这样的查询十分冗长而复杂。我们为rectangle_t引入的方法实际上没有扩展查询的功能的原因是方法的逻辑太简单，以至于可以用非过程性SQL直接来表达。如果我们在一个交互式SQL不能复制的方法中执行循环语句，则这一点是不正确的。

例4.4.10 我们展示如何创建一个对象类型wordset_t，它表示词的嵌套表（如一个关键字集合）。例4.4.11的documents表中的该类型的一个属性将使我们可以以集合中的任意关键字的子集合来查询文档。

```

create type wordtbl_t as table of varchar(30); -- nested table type of [key]words
create type wordset_t as object {
    words      wordtbl_t;                      -- [key]word set object type with methods
                                                -- attribute is nested table of [key]words
    member function wordcount return int; -- return count of words in self object
    member function subset(x in wordset_t) return int; -- is self a subset of x?
    member function superset(x in wordset_t) return int -- is self a superset of x?
};
create or replace type body wordset_t as
    member function wordcount return integer is
        begin
            return self.words.count;           -- built-in collection method counts words in
                                                -- wordset_t
        end;
    member function subset(x in wordset_t) return int is
        matches int; i int; j int;
        begin
            for i in 1..self.words.count loop -- loop through all words in self.words
                matches := 0;
                for j in 1..x.words.count loop -- loop through x.words; nested table..
                    if self.words(i) = x.words(j) then -- rows found by index no.
                        -- in PL/SQL
                            matches := matches + 1; -- found match. count it
                    end if;
            end loop;
        end;

```

^Θ 正如前面的一个注释中提到的，通过定义rectangle_t的“排序”方法我们也可以在ORDER BY子句中使用value(p)。

```

    end loop;                                -- end for loop on x.words
    if matches = 0 then
        return 0;                            -- self.words not subset of x.words
    end if;
    end loop;
    return 1;                                -- we found all self.words in x.words
end;
member function superset.          -- similar to subset (reverse x and self)
end;

```

由于篇幅的限制，我们没有给出superset()方法的定义。其代码与我们所见到的subset()方法的代码相似。 ■

在例4.4.10中，wordcount的成员函数（方法）的定义通过在表达式self.words.count中使用ORACLE内部的方法count来确定嵌套表对象self.words的元素个数。注意右边的“（）”对内部的方法来说不是必须的。但不幸的是，这种内部的方法不能用在交互式的SQL中。

在subset方法定义中，我们需要操作wordset_t对象中的嵌套表中的元素。在嵌套表中的整个元素集合将被读入内存，并以类似数组的结构提供给PL/SQL使用。元素被编址为self.word(1), self.word(2)，等等。我们使用一个对词的外循环和一个内循环来计算我们的词与其他wordset_t对象x的词的匹配的数目。如果在第一遍匹配时就退出内循环，则这里的代码的效率将更高。

在第3章的例3.11.7中，我们考虑了如何检索含有许多指定的关键词的所有文档的问题。我们可以再次利用该问题来说明对象关系的能力。首先，我们不再像关系数据库中那样将文档分解为两个表，而是定义一个包含每个文档的关键字集的嵌套表的documents表。

例4.4.11 创建一个对象类型document_t，document_t含有标识属性docid、名字docname、作者person_t和wordset_t类型的关键字集合对象属性keywords。创建一个对象表documents，主键为docid，包含document_t类型的对象。

```

drop table documents;                      -- drop old table (depends on type below)
create or replace type document_t as object-- and old.type, before re-creation
(
    docidint,                           -- unique id of document
    docnamevarchar(40),                -- document title
    docauthorperson_t,                 -- document author
    keywordswordset_t                  -- keyword set for document
);
create table documents of document_t       -- this is the object table of documents
(
    primary key(docid)
) nested table keywords store as keywords_tab; -- nested table of keywords for
                                                -- documents

```

下面我们描述一个如何使用适当的关键词来从documents表中检索行的例子。

例4.4.12 在上载documents中的行之后找出包含关键词boat的文档。

```

select docid from documents d
where 'boat' in (select * from table(d.keywords.words));

```

第一个查询不需要使用我们在例4.4.10中创建的方法。但如果我们要查询所有文档中的关键词总数，我们可以不用wordcount()方法，但是在ORACLE的交互式SQL中不使用wordcount()方式实现这一任务是不可能的。

```
select sum(d.keywords.wordcount()) from documents d;
```

例4.4.13 检索含有三个关键词'boat', 'Atlantic'和'trip'的文档。这需要使用例4.4.10中的其他方法。

```
select docid, docname from documents d
  where d.keywords.superset(wordset_t(wordtbl_t('boat', 'Atlantic', 'trip')))>0;
```

比较上面的查询和例3.11.7中的关系查询。上面的表达式

```
wordtbl_t('boat', 'Atlantic', 'trip')
```

是图4-22之后介绍的嵌套表构造器。这个表被传给对象构造器wordset_t()。最后将被构造的对象与Superset方法相比来查看document对象中的关键词是否在它的超集中。

下面我们将显示如何检索正好包含三个关键词'boat', 'Atlantic'和'trip'的文档。我们可能试图使用where d.keywords=wordset_t(wordtbl_t('boat', 'Atlantic', 'trip'))，但是，虽然d.keywords是一个对象，简单的对象间可以比较，而带集合的对象不能比较。

```
select docid, docname from documents d
  where d.keywords.superset(wordset_t(wordtbl_t('boat', 'Atlantic', 'trip')))>0
    and d.keywords.subset(wordset_t(wordtbl_t('boat', 'Atlantic', 'trip')))>0;
```

3. 更新方法

迄今为止我们所讨论的方法仅限于只读的方法。但是尽管只读方法很重要，完成更新对象的方法可能比只读的方法更重要。写更新方法的程序员具有防止这样的更新方法的负面影响的时间和经验，然而，因为不经意的疏忽无计划地修改专用特性可能会导致数据不一致。本节讨论如何创建更新对象的方法。

让我们使用点和矩形的例子来解释更新方法。假定我们想要通过方法允许改变矩形的pt2值来放大或缩小一个矩形。我们可以考虑对例4.4.5的rectangle_t对象类型增加一个方法：

```
member function resize(npt2 point_t) return...
```

这里的npt2是我们想要为矩形设定的新pt2值。注意这种改变可能导致一个无效的矩形，即它的右上角可能低于或在左下角的左部。因此，我们必须在方法的代码中防止出现这种可能性。

在例4.4.3后面的讨论中，我们指出如果一个函数可被用于SQL语句中，那么该函数不能修改它的任何参数。由于同样的原因，方法也不能改变它们的self的对象。resize()方法附加于rectangle_t对象，并且不能改变self对象意味着在返回结果给调用者的通信过程中我们需要返回一个rectangle_t类型的对象。我们将在例4.4.14显示这一过程。

例4.4.14 实现上面讨论的resize方法。下面是rectangle_t的Create 或Replace Type Body语句的部分。

```
member function resize(p point_t) return rectangle_t is
```

```

newrect rectangle_t := self;           -- clone self to local object
begin
  if p.x > self.pt1.x and p.y > self.pt1.y then -- check new rectangle is valid
    newrect.pt2 := p;                         -- OK, update pt2 for return rectangle
  end if;
  return newrect;                        -- no change if pt2 was illegal
end;

```

■

注意在例4.4.14中我们是如何克隆一个self矩形到局部变量newrect中的。这个局部变量可以被改变，然后再返回给想知道新矩形的调用者。如果带有pt2等于p的新矩形不是有效的矩形，则原来的矩形值将被返回。

例4.4.15 若改变有效，则显示将rects中pt2改变为新坐标(10, 10)的所有矩形，假设这是有效的矩形形式；否则，保留矩形的当前形式。

```
select r.resize(point_t(10,10)) from rect r; -- display resized rectangles
```

■

在例4.4.15中，rects中实际上没有矩形被改变，因为我们仅仅显示由resize返回的值。但是，我们当然是要使用resize()来改变数据库中的值，那么需将改变后的矩形放在Update语句中的SET子句中。

例4.4.16 对rects中的所有pt1在(2, 1)位置的矩形应用重定义大小的操作将pt2.x和pt2.y都增加1。

```

update rect r
set r = r.resize(point_t(r.pt2.x + 1,r.pt2.y + 1))
where r.pt1 = point_t(2,1);

```

■

4.4.2 INFORMIX中的用户定义函数

INFORMIX没有对象方法，属于对象类型的函数被自动定义作用于该类型的对象，但是用户定义函数有同样的表现力且灵活性要大一点。我们将在INFORMIX的过程性语言SPL中观察用户定义函数，SPL与ORACLE的PL/SQL非常相似。我们将在后面介绍SPL的语法。本节后面的图4-35总结了SPL和PL/SQL的基本语法成分。关于语言的更多信息可以参考《INFORMIX Guide to SQL Tutorial》(参见本章末尾的“推荐读物”)。

1. SPL: INFORMIX的过程性SQL语言

在过程性含义上讲，SPL是交互式SQL的扩展，它提供了在函数或过程中实现过程逻辑的方式。像ORACLE的PL/SQL一样，SPL编程包括声明内存变量和实现循环和条件的执行；在任意时刻SQL可被用来访问数据库。

在SPL中，所有的代码包含在一个函数（或过程）中。SPL不支持ORACLE例4.4.1那样的独立的代码块。每个函数以CREATE FUNCTION开始以END FUNCTION结束并且由代码块组成。一个SPL代码块从声明内存变量的一系列DEFINE开始，紧跟着一系列可执行的语句。DEFINE语句不能出现在任何可执行语句的后面。整个代码块被CREATE FUNCTION/END FUNCTION或BEGIN/END对包围。在一个块内定义的变量不能被这个块外的程序访问。也就是说，整个函数定义像一个隐含的代码块一样，但我们可以其中创建语句块来定义内存变量。我们从一个完成简单运算的函数开始对此进行说明。

例4.4.17 用SPL写一个累加1到n的整数的函数，n为任意整数。

```
create function sum_n(n int) returning int;
  define i int;
  define total int;
  let total = 0;
  for i = 1 to n
    let total := total + i;
  end for;
  return total; -- return result to SQL or SPL caller
end function;
```

如定义有错，则系统将自动显示带行号的编译错误。我们可以使用下边的INFORMIX SQL语句来执行上面的函数：

```
execute function sum_n(10);
```

我们也可以在任何SQL语句中使用这个函数：

```
select sum_n(10) from orders;
```

然而，这个Select语句将以列的形式返回多个值！对应orders中的每一行将有一个值。显然在INFORMIX中这不是预期的结果。当我们并不想返回多个值的时候，我们仅仅使用上边的Execute语句。当然，我们可以在函数sum_n中使用实际的列值：

```
select qty, sum_n(qty), 2*sum_n(qty*qty) from orders where aid = 'a04';
```

注意sum_n()是一个标量函数，不能与诸如sum()和count()这样的SQL集合函数相比。 ■

例4.4.17显示了SPL的几个重要的特点。我们看到关键字CREATE FUNCTION和END FUNCTION限制了程序块的界限。注释以双短线(--)开始。赋值语句形如let target=expression，不同于PL/SQL中的target := expression。SPL对内存变量使用与SQL对命名列所用的同样的数据类型语法（参见3.9节中的讨论），并访问同样的内部函数如sqrt()和substring()。

在一个像例4.4.17那样的Create Function语句执行后，函数将成为如表或视图那样的一个长期存在的模式对象。Create Function语句的一般形式如图4-33所示。

<pre>CREATE FUNCTION functionname ([param [, param ...]]) RETURNING datatype; statements end function; param ::= paramname datatype</pre>

图4-33 INFORMIX用户函数的SPL语法（到现在所涉及的）

一个简单的循环可以用如下语法设定：

```
FOR counter = lower_bound TO higher_bound [STEP increment]
  -- sequence of statements
  -- exit loop at any point with EXIT FOR condition
END FOR;
-- flow of control resumes here after EXIT FOR occurs within loop
```

在FOR和END FOR之间是一到多条语句构成的循环体；每个语句像C和Java一样以分号结束。在例4.4.17中仅出现了一条语句。递归过程从按从小到大的次序从第一个界限到后一个界

限，即便STEP递增值为负时也如此。如for 10 to 1 step -1。

循环也可以因任意选定条件而结束，而无需到达计数限制，一种不同的循环形式如下：

```
WHILE condition
  -- sequence of statements
  -- exit loop at any point with: EXIT WHILE condition;
END WHILE;
- flow of control resumes here after EXIT WHILE occurs within loop
```

例如，我们可以重写例4.4.1的程序如下：

```
create function sum_n2(n int) returning int;
  define i integer;
  define total integer;
  let i = 0;
  let total = 0;
  while TRUE                         -- could instead write while i <= n
    let total := total + i;
    let i := i + 1;
    exit while i > n;                -- exit while clause
  end while;
  delete from result;                -- delete all old rows from database table
  insert into result (rvalue) values (total); -- insert answer into a database
                                              -- table
end function;
```

在执行上面的函数后，用户可以从result表中选择来决定结果或使用Execute语句显示结果。本节末尾的图4-35给出了INFORMIX和ORACLE的各种编程结构，包括循环语句语法。

上面定义的第二个函数sum_n2()不能被SQL成功调用，尽管使用Execute Function可以很好地运行。相反，从SQL使用它将产生错误“Illegal SQL statement in stored procedure.”。这里的非法语句是指Insert语句。和ORACLE一样，在INFORMIX中，由SQL调用的函数不能改变数据库中的数据，因为SQL中的函数调用被当作表达式（仅仅从其他值来计算一个值）。

由于Select语句不改变数据库中的数据，因此可以用在一个SQL可调用的函数中，如下例所示。

例 4.4.18 写一个查找给定cid的customers表中的cname和city并以诸如“Tiptop(Duluth)”的格式字符串返回信息的函数。

```
create function namecity(custid char(10)) returning char(20)
  custname char(20);   -- make local string big enough (it gets trimmed below)
  custcity char(20);
begin
  select cname, city into custname, custcity from customers where cid = custid;
  return trim(custname) || '(' || trim(custcity) || ')'; -- trim extra spaces off
end;
```

这个函数可以从SQL以sum_n()同样的方式调用。 ■

例4.4.18中使用了返回单个行的Select。通过游标，PL/SQL同样可以处理多行的选择，本文不做介绍。

在诸如C的许多语言中，在函数代码内可以安全地改变任何参数变量，因为这些语言具有

传值调用的方式，即在被调用函数中的参数仅仅是在调用语句中使用的参数的一个副本：任何参数变量的初值实际上不会改变。尽管SPL使用传值调用，但具有模糊的关于什么时候参数变量可以用于RETURN语句中的代码规则。最安全的方式是像PL/SQL一样用SPL中使用引用调用的方式。这意味着我们应该在计算返回值时避免改变参数变量并创建新的局部变量，这样可以防止打破SPL中关于返回值的特殊代码规则。这里有一个改变参数来打破我们的规则的例子（尽管SPL对违反该规则的惩罚仅是返回更新后的值）。

例4.4.19 写一个更新某个参数的函数。

```
create function x_out(s varchar(80)) returning varchar(80);
  let s[1,3] = 'XXX'; -- change the parameter **DON'T DO THIS!!**
  return s;           -- return updated parameter variable
end;
```

注意，这里使用的INFORMIX子串的语法是3.9节的图3-18后面提到的。这里`s[1,3]`表示从位置1到3的子串。这个函数将被成功地编译并可用于Execute Function或更复杂的SQL调用。

```
select x_out(cname) from customers;
```

然而，`x_out()`函数更新了参数变量，违反了我们的代码规则，因此我们应将其重写为使用局部变量的更新语句：

```
create function x_out(s varchar(80)) returning varchar(80);
  define newstr varchar(80);
  let newstr = s;           -- clone parameter string to local variable newstr
  let newstr[1,3] = 'XXX'; -- change newstr's first three chars to 'XXX'
  return newstr;           -- return updated LOCAL string
end;
```

注意，SPL编译程序并不报告违反了返回参数的代码规则，但是结果函数可能有时正确，有时不正确。为了保证安全性，我们将只改变局部变量，就像我们在PL/SQL中所做的那样。

我们经常需要定义局部变量来保存中间和最终结果。我们可以使用行构造器生成一个局部行类型，如同我们在例4.4.20中看到的局部变量`nper`，类似的例子见例4.4.4。

例4.4.20 写一个将`person_t`对象的`age`增加1的函数。具有参数`x`的`age`增加1后的返回值返回到局部`person_t`对象类型变量中。

```
create function inc_age(x person_t) returning person_t;
  define nper person_t;           -- local variable nper
  let nper = cast(row(x.ssno, x.pname, x.age) as person_t); -- clone x to nper
  let nper.age = age + 1;         -- change age
  return nper;                   -- return local nper
end;
```

这里需要创建一个局部行类型变量`nper`来用于逻辑的改变。我们通过在第二行中使用`x`对象的字段（它的`ssno`，`pname`和`age`字段）来指定行构造器的所有参数。因此新的局部变量是`x`的克隆。不幸的是，`row(x)`并不起作用；我们需要单独描述字段。赋值`nper := x`是可行的，但是，预先调用构造器`row()`来生成变量`nper`仍是必需的。整个函数体十分简单，可以仅用

一行表示如下：

```
return cast(row(x.ssno, x.pname, x.age+1) as person_t); -- construct and return
```

为使用这个函数，我们显示更新过年龄的人中的所有person_t对象：

```
select inc_age(p) from people p;
```

这里没有表的数据被更新；增加操作仅影响显示值。为了更新表中数据，我们需要使用Update语句。然而，在INFORMIX中，我们不能一次性地更新整行对象，因此，我们不能给出使用inc_age()来更新people表的简单例子；而是要更新其作为列的每一个字段(ssno, pname, age)。 ■

2. 在INFORMIX中使用SPL实现UDF

像ORACLE例4.4.5一样来实现点和矩形的用户定义函数area()和inside()。由于这些不是方法，我们需要用显式的矩形参数来调用他们。对一个矩形r和点p的参数调用是area(r)和inside(r, p)。

例4.4.21 我们创建INFORMIX行类型point_t和rectangle_t，再实现用户定义函数area(r)和inside(r, p)。

```
drop function area;
drop function inside;
drop row type rectangle_t restrict; -- Note we must drop objects depending...
drop row type point_t restrict; -- ...on these types before dropping the types
create row type point_t
(
    x int,                      -- horizontal coordinate of point
    y int                        -- vertical coordinate
);
create row type rectangle_t
(
    pt1 point_t,                -- point at lower left-hand corner of rectangle
    pt2 point_t                 -- point at upper right-hand corner of rectangle
);

create function area (r rectangle_t) returning int;
    return (r.pt2.x-r.pt1.x)*(r.pt2.y-r.pt1.y); -- area of rectangle r
end function;

create function inside (r rectangle_t, p point_t) returning boolean;
    if (p.x >= r.pt1.x) and (p.x <= r.pt2.x) and (p.y >= r.pt1.y) and (p.y <= r.pt2.y)
        then return 't';           -- return True if p inside r
    else
        return 'f';             -- return False if p not inside r
    end if;
end function;
```

可以看到，SPL创建UDF的语法与ORACLE PL/SQL定义方法的Create Type Body语句的语法十分接近，该语句定义方法代码。INFORMIX支持inside返回值为布尔类型，这使得查询更好一些。 ■

例4.4.22 为点和矩形创建类型表，并插入一些值。

```
insert into rects values (cast(row(1,2) as point_t),cast(row(3,4) as point_t));
```

```

insert into rects values (cast(row(1,1) as point_t),cast(row(6,6) as point_t));
insert into points values (2,3);
insert into points values (1,4);
insert into points values (4,4);

```

为验证所做的工作，书写简单的查询来计算每个矩形的面积并检查点(4, 2)是否在每个矩形中。

```

select area(x) from rects x;
select inside(x,cast(row(4,2) as point_t)) from rects x;

```

结果证明是对的。 ■

INFORMIX Create Function语法如图4-33所示，本节末尾的图4-35给出了我们在INFORMIX和ORACLE中使用的各种编程构造器。

现在，我们来考虑ORACLE中例4.4.10中介绍的关键字和文档的例子。这个例子很重要，因为INFORMIX缺乏ORACLE中的交互式SQL使用游标来跨越一个表的多行的汇集或插入更新汇集中的单个元素的能力。但是，通过使用SPL函数，我们可以逐个元素地搜索多行汇集并改变其中的元素。

像例4.4.10的方法一样，我们通过创建实现子集操作的用户定义函数开始。这里，我们不需要词计数函数，因为INFORMIX提供了CARDINALITY()作为集合的内部函数。我们也不需要超集(superset)操作，因为我们的UDF没有隐含由被self引用的一或两个参数决定的次序。

例4.4.23 对varchar(30)元素集合写一个用户定义函数subset(wdset1, wdset2)，若wdset1是wdset2的一个子集，则返回真(TRUE)。

```

create function subset(wdset1 set(varchar(30) not null),
                      wdset2 set(varchar(30) not null)) returning boolean;
define matchcnt int;          -- local variable for count of word matchcnt
define wd1 varchar(30);       -- local variable for word from wdset1
define wd2 varchar(30);       -- local variable for word from wdset2
foreach cursor1 for          -- loop through elements of wdset1; see Fig. 4.34
    select * into wd1 from table(wdset1) -- select 1 element at a time from cursor1
    let matchcnt = 0;                  -- initialize count of matchcnt for wd1 in wdset2
    foreach cursor2 for            -- loop through elements of wdset2
        select * into wd2 from table(wdset2)
        if (wd2 == wd1) then
            let matchcnt = matchcnt + 1; -- count match of wd1 in wdset2
        end if;
    end foreach;
    if (matchcnt==0) then
        return 'f';                -- this wd1 not in wdset2. fail
    end if;
end foreach;
return 't';                   -- all wdset1 words were found in wdset2
end function;

```

■

图4-34显示了用在例4.4.23中的汇集循环的语法。为使用它，我们必须定义汇集元素型参数的局部变量（在图4-34中被称为element_var）并在将collection_var参数转换为表的选择语句Select Into中使用它。在对汇集元素的每一次循环中，Select Into将来自下一个汇集元素填充到元素变量中。

```

FOREACH cursorname FOR
  SELECT * INTO element_var FROM TABLE(collection_var)
  statements
END FOREACH;

```

图4-34 汇集变量的游标循环的INFORMIX SPL语法

在例4.4.11中，我们定义了在检索所有具有许多指定关键字的文档的ORACLE查询中所需的documents表。在例4.4.24中，我们类似地定义了一个对每个文档包含关键字集合的名为documents的INFORMIX表。

例4.4.24 创建一个行类型document_t，document_t含有标识符docid、文档名docname、作者person_t和keywords集合varchar(30)。创建一个类型为document_t的类型表documents。

```

drop table documents;
drop row type document_t restrict; -- all objects dependent on type must be dropped
create row type document_t
(
  docid      int,                      -- unique id of document
  docname    varchar(40),                -- document title
  docauthor  person_t,                 -- document author
  keywords   set (varchar(30) not null) -- keyword set for document
);
create table documents of document_t
(
  primary key(docid)
);

```

例4.4.25 上载一些行后，找出带关键词'boat'的文档。

```

select docid from documents
  where 'boat' in keywords; -- note IN doesn't require a Subquery in INFORMIX

```

和例4.4.12一样，这个查询根本不需要任何函数。我们的下一个目标是检索所有文档的关键词总数。

```

select sum(cardinality(keywords)) from documents;

```

例4.4.26 检索带所有关键词'boat'，'Atlantic'和'trip'的文档。

```

select docid, docname from documents
  where subset(set{'boat','Atlantic','trip'},keywords);

```

下边，我们查询正好带有三个关键词'boat'，'Atlantic'和'trip'的文档。对此，在ORACLE中，我们可能需要检查子集和超集，因为PL/SQL不支持对嵌套表的等价测试。但是，INFORMIX支持对集合的等价测试，因此，我们有如下的简单查询：

```

select docid, docname from documents d
  where d.keywords = set{'boat','Atlantic','trip'};

```

假定我们创建一个requests表，其中的每一行有一个类型为SET of varchar(30)名为wds的列，包含一个要在一些文档中查找的关键词集合和一个整型的ID reqid。下面的Select语句将文档与请求相匹配：

```
select d.docid, r.reqid from documents d, requests r
  where subset(r.wds, d.keywords);
```

一个含有许多关键词的文档可能被许多请求发现。为了更具有选择性，我们可以对每个查询要求报告具有所有的关键词的文档和最小关键词总数的文档。

```
select d.docid, r.reqid, cardinality(d.keywords) from documents d, requests r
  where cardinality(d.keywords) =
    (select min(cardinality(d1.keywords)) from documents d1
      where subset(r.wds, d1.keywords)) and subset(r.wds, d.keywords);
```

3. 更新函数

用户定义函数提供了在INFORMIX中改变汇集中的单个元素的唯一方式。和ORACLE一样，我们可以创建返回更新后的汇集的函数，也可以使用它们改变该数据类型的任一列。

例4.4.27 写一个增加一个词到varchar(30)类型的词集合的函数addword，该函数返回一个增加了词的新集合。

```
create function addword(wdset set(varchar(30) not null), wd varchar(30))
  returning set(varchar(30) not null);
define newset set(varchar(30) not null);
let newset = wdset;           -- clone parameter set for return value
insert into table(newset) values (wd);  -- add new element
return newset;
end function;
```

例4.4.28 使用addword增加'bird'到documents表中docid为200的文档的关键词列表中。

```
update documents set keywords = addword(keywords, 'bird')
  where docid = 200;
```

类似地，我们可以写一个将一个词集合加入到某些文档的关键词集合中的函数。然后，我们需要一个扫描插入集合的游标，可以将每个元素插入到扩展集合中。

例4.4.29 写一个删除varchar(30)类型的词集合中的指定词的函数delword，该函数返回一个减少了的词的新集合。

```
create function delword(wdset set(varchar(30) not null), wd varchar(30))
  returning set(varchar(30) not null);
define wd1 varchar(30);          -- for Select into use in cursor
define newset set(varchar(30) not null);
let newset = wdset;           -- clone set for return value
foreach cursor1 for           -- scan set, looking for wd to delete
  select * into wd1 from table(newset)
  if wd1 = wd then           -- found wd, delete it
    delete from table(newset) where current of cursor1;
  exit foreach;             -- only one to delete, so leave loop now
end if;
```

```

end foreach;
return newset;                                -- return set, possibly one element smaller
end function;

```

注意这里的SPL语句格式：

```
delete from table(wdset) where current of cursor1;
```

delword()函数和addword()的方式类似。

4.4.3 用户定义函数小结

我们已经研究了ORACLE的方法和INFORMIX的用户定义函数即UDF。实际上，ORACLE的方法是一种用户定义函数UDF类型，并且ORACLE支持用PL/SQL或C这样的语言写的非方法的UDF。这样，我们在ORACLE中可以将subset()作为一个单独的UDF函数来实现，并像INFORMIX一样用subset(s1, s2)来调用它。

PL/SQL和SPL具有如下相似性（两种语言元素的比较如图4-35）：

- 数据类型在Create Type或Create Row Type语句中指定。
- 算术运算与交互式SQL一样。
- 点号被用于访问属性/字段。
- 对象/行类型的局部变量在使用前需用对象/行构造器(或一条Select Into语句)初始化。
- RETURN表达式可以出现在函数体中的任何地方。

语言元素	ORACLE PL/SQL	INFORMIX SPL
局部变量声明	varname datatype [:= initval];	DEFINE varname datatype;
赋值	varname := expr;	LET varname = expr;
布尔等	-	-
IF语句	IF boolean_expr THEN statements [ELSE statements] END IF;	IF boolean_expr THEN statements [ELSE statements] END IF;
FOR LOOP语句	FOR var IN first..last LOOP statements END LOOP;	FOR var IN (first TO last) statements END FOR;
RETURN语句	RETURN expr;	RETURN expr;
汇集x的循环	i int; --local variable; ... FOR i in 1..x.count LOOP statements using element x(i) END LOOP;	DEFINE e elementtype;--local var ... FOREACH cursorname FOR SELECT * INTO e FROM TABLE(x) statements using element e END FOREACH;
对象/行类型构造器	objecttype(expr....)	row(expr....)
SELECT INTO语句	SELECT expr INTO varname FROM ...	SELECT expr INTO varname FROM ...

图4-35 过程性SQL语句成分：ORACLE PL/SQL和INFORMIX SPL

4.5 外部函数和打包的用户定义类型

在本章中，我们已经研究了允许我们直接并相对容易地实现的对象和汇集类型。在本节中，我们将研究对象-关系扩展，即用户定义类型(UDT)及其函数，它们的实现更具有挑战性，通常用比执行复杂计算的过程化的SQL更有效的C或Java代码。但是一旦实现后，新的UDT像前边讨论过的一样易于使用。显然，这种情况为工具构造者提供了写通用的UDT并将它们的卖给数据库商的机会。数据库厂商已经认识到这一点的积极性并为此提供了便于安装的UDT“打包”机制。这些UDT包在INFORMIX中被称为DataBlade(数据刀片)，在ORACLE中称为Cartridge、DB2 UDB则称为Extender(扩展器)。

例如，文本检索包使得书写涉及到在文档中发现字模版的应用程序更加方便。当这样的包安装后，新UDT出现准备使用。我们仍然将其称为UDT，因为对数据库厂商来说它们是添加上去的，即便是数据库商自己销售的包。那些不是作为数据库系统一部分提供的语言（如C或Java）写的函数被称为外部函数。

打包的UDT一般支持“多媒体”数据，也就是说混和了非传统数据类型。涉及大量数据：图像、文本文档、声音文件、视频等。为了能够处理这些大的数据对象，数据库需要支持存储这些数据的二进制大对象(Binary Large OBject, BLOB)。BLOB将在后面进一步讨论。一种常用的实现技术是由BLOB和少量的内部数据类型来构成一个UDT。例如，一个图像可能有一个包含像素和整型的高度和宽度数据的BLOB，高度或宽度可能是BLOB的一部分或以独立的表值存在(依赖于数据库)。

为理解这一主题，我们需要先介绍几个在这种包中使用的概念：二进制数据和BLOB、外部函数、封装和特殊类型。

1. 二进制数据和BLOB

存储“自然数据”的能力，对数据库而言即长的位串，其实质是为了存储图像、声音和其他大数据对象。ORACLE、INFORMIX 和DB2 UDB都支持BLOB。对一个2MB的BLOB，你可以在INFORMIX和ORACLE中使用BLOB数据类型或在DB2 UDB中使用BLOB(2M)。在上边的三个产品中，BLOB可以根据需要任意增长，直至所需的资源耗尽。在ORACLE中BLOB的最大长度以字节为单位为4GB - 1，在INFORMIX中可达4TB(terabyte)，在DB2 UDB为2GB - 1。

SQL-99用命名扩展特征“基本LOB数据类型”来描述二进制字符串类型BINARY LARGE OBJECT即BLOB。例如，BLOB(100)提供100字节的存储空间，BLOB(10G) 提供10G存储空间(假设数据库支持这样大的存储空间)。类似地，K和M可以用来表示千字节和兆字节。这些BLOB可以赋予带有十六进制串的字母值，如X'104ABF'。根据SQL-99的另外一种命名特征“扩展LOB支持”，BLOB中的某部分可以在SQL中用二进制串函数SUBSTRING来访问，且在查询中可以用POSITION或LIKE子句来进行模式匹配。然而，BLOB的完整的功能是通过外部定义函数来实现的。外部函数可以从文件中装载它们，对数据执行专门的计算，并返回重要结果给SQL调用者。

2. 外部函数

外部函数是用数据库没有保存但可以从SQL调用的语言实现的函数。在函数实现后，已编译的代码必须放入一个数据库服务器可以访问的文件中。然后，外部函数通过服务器用Create Function语句登记。这个语句指定函数名、参数、返回值的类型和编译代码的位置。需注意外

部函数和已存储的过程（代码存储在数据库中）都用Create Function语句注册，但二者的参数不同。

ORACLE、INFORMIX和DB2 UDB都支持带BLOB、标准数字、字符串或其他类型参数的外部函数。在这三种产品中，BLOB的部分可以被外部函数的代码访问。它们都使用Create Function语句来注册外部函数，尽管这些产品在语法细节上有些差别。

Core SQL-99描述了外部函数（SQL-92则没有），并称之为SQL调用函数，Full SQL-99还描述了外部方法，即附属于用户定义类型的外部函数。这些也都用Create Function语句注册。

3. 封装

封装用于使程序不能通过除与对象关联的方法和用户定义函数以外的其他途径来访问对象中的数据的情形。封装的一个例子如INFORMIX的不透明类型。将封装作为面向对象概念的讨论参见4.2.3节。

4. 特殊类型

可以从任一其他内部或非内部的类型使用特殊类型(*distinct type*)的方式复制现有类型为新类型。特殊类型一旦被定义，就会被当作单独的类型处理，这样，如果类型U是基于类型T的特殊类型，不经过类型转换(一种显示改变类型的方法)就不能比较这两种类型的对象。特殊类型的思想是提供类型检查来捕获诸如将价格与百分数或将美元价格与日元价格比较的错误。

5. BLOB对象

一般地，如果一个数据库支持BLOB和传递BLOB及其他多数有用类型的外部函数，则该数据库也支持函数可访问的UDT，即封装的对象。要支持在插入和更新中使用BLOB对象的更新函数在很多情形下是很困难的。现在考虑在例4.4.20和例4.4.4的inc_age()函数中我们是如何必须复制一个局部变量返回给调用者的情形。一般地，我们不可能复制一个局部BLOB，因为它可能超过内存容量。这样，除非数据库厂商确保一直直接引用返回的BLOB的方法（像DB2 UDB DB一样），然而，返回一个BLOB值可能是不可能的。如果数据库厂商不提供这种能力，则你需要用C语句（或其他过程性语言）写一个能够知道哪些表的哪些列需要转载或更新的专门的BLOB装载器或更新器。在很多情况下，这应该不是严重的问题。

对BLOB对象应用，考虑我们在4.4节中使用的矩形的例子。我们可以使用4个整数以16字节长的BLOB来存储定义矩形的两个点的坐标。我们可以写一个外部函数area()来接受一个BLOB，获得在它之外的4个数字来计算面积，用C或Java实现。类似地，我们可以实现inside()函数。然后，在SQL中，我们可以建立一个带有BLOB列的表，并按照4.4.2节中我们处理用户定义函数的方式来使用这些函数。我们可以需要用专门的程序将矩形装入BLOB列中。

尽管我们可能想命名这种rectangle_t类型，在一些产品中，我们可能会使用BLOB作保留类型名。如果数据库支持特殊类型，这种能力可以使我们将BLOB类型重命名为rectangle_t类型。但是，即使没有特殊的类型名，BLOB及其函数也具有对象类型的重要特点，我们可以将以这种方式实现的对象称为BLOB对象。

当然，为了更实现，我们应该考虑比保存在BLOB的矩形更复杂的情形。实际上，我们应该在需要用到BLOB的强大能力的时候使用它。注意，尽管在SQL中BLOB的设置各种产品间非常类似，由C实现代码使用的BLOB程序接口会有相当多的变化。

尽管通常对BLOB的讨论很有用，但我们需要注意伴随在其中的许多的特定产品的细节，

同时一些BLOB对象的竞争者也值得注意。让我们来看一下这三种不同的产品。

6. 打包的UDT和其他封装的UDT

INFORMIX最早支持UDT，有最多的UDT包。UDT包被称为DataBlade（数据刀片），可以被DBA安装以支持特定数据类型的访问，如地理数据和图像。数据库程序员可以用数据库建模类型写他们自己的DataBlade。这些类型通常是基于内部BLOB类型的。有专门的DataBlade管理器来处理DataBlade的函数和类型登记。

被称为不透明类型的单个UDT更为简单而易于实现并且对数据库程序员完全文档化。不透明类型是基于lvarchar的属性类型。它们必须小得足以装入程序内存，于是它们通常被限制在1GB以内。不透明类型由Create Opaque Type语句定义并完全封装。它们可以包含嵌入式BLOB引用（称为locator），由不透明类型的函数管理。不透明类型的输入函数可以被SQL Insert语句使用，函数可以用于比较、批量装载及其他数据库操作。

不透明类型的函数必须用C语言书写，因此它是外部函数。每个函数由数据库使用Create Function语句按图4-33给出的语法注册，对外部函数的情形，Create Function语句的SPL函数体(图4-33中的statements行)部分被一个指定复杂代码文件的EXTERNAL NAME子句和指定为C语言的LANGUAGE C子句替换，

ORACLE中与INFORMIX的DataBlade对应的打包UDT为“Cartridge”。例如，Virage公司提供可以同时用作INFORMIX DataBlade和ORACLE Cartridge的图像检索软件。数据库程序员可以写PL/SQL或用C写访问BLOB的外部函数来设立一个BLOB对象。在PL/SQL或C中，BLOB可以通过字串函数或其他只读函数访问（只读维持我们从SQL调用它的能力）。任一函数类型用Create Function语句在数据库中登记，像4.4.2节介绍的成员函数一样。对外部函数的情况，PL/SQL函数体（AS或IS后的部分）被一个描述编译后代码的位置EXTERNAL LIBRARY或各种可选的语句替换。尽管BLOB可以传递给外部函数，但对象和汇集却不能。这一点并不是严重的问题，因为我们可以有带BLOB属性的对象。你可以把BLOB加上任何其他所需的属性作为参数传递给外部函数。为此，你将需要写一个专用的BLOB上载程序。

从版本5.2开始，DB2 UDB具有像ORACLE的对象和INFORMIX的行类型那样的结构化的用户定义类型。像INFORMIX一样，这些类型可能被安排在继承层次，并且像ORACLE一样，它们可由REF指向。然而，UDT并没有被完全集成到数据库的类型系统中。例如，不能有UDT类型列，尽管一个列中可能包含指向UDT的REF。DB2 UDB在很早以前就具有外部函数和实现BLOB对象的BLOB能力。使用DB2 UDB的特殊类型能力，每个对象可有一个用户指定的名字。这些UDT的包称为Extender，操纵UDT的外部函数可用C或Java（或C++及其他语言）实现。

DB2 UDB支持用户定义函数，即允许从用户函数以表的形式一个接一个地返回行给系统的编程接口。这样，使用用户定义的表函数可以使任意一种表格式的数据看起来像一个表一样，这是一种方便的扩展数据库的方式。另外，在撰写本文时，DB2 UDB还不支持汇集类型。

7. 小结

显然，前面任一种数据库都支持多媒体数据。图4-36给出了这些数据库逐个特征的比较。每个数据库都有各自的书写访问（只读）数据的外部函数的方式，这些外部函数都可以由SQL调用。我们可以购买INFORMIX的DataBlade、ORACLE的Cartridge、DB2 UDB的Extender中的某一种产品，使用这些产品提供的UDT和函数来访问数据。这些包的数目正在增长，你可

以访问数据库厂商的Web站点了解当前的产品状况。这三种产品的URL地址分别为：www.oracle.com、www.informix.com和www.ibm.com/db2。

	ORACLE	INFORMIX	DB2 UDB
SQL中的用户定义类型(UDT)	有, 结构化的“对象类型”	有, 结构化的“行类型” 独立类型	有, 结构化的UDT和 独立类型
继承	无	有	有
UDT引用	有	无, 但计划有	有
SQL中的集合类型	有, 嵌套表VARRAY	有, 集合, 多集合, 列表	无
用户定义函数(UDF)	有, 包括PL/SQL, C, Java写的对象类型方法	有, 用SPL(过程化SQL), C, Java写	有用C, Java写
封装的UDT(未打包)	BLOB对象	BLOB对象, 不透明对象	BLOB对象
打包UDT	Cartridge	DataBlad	Extender
BLOB的最大尺寸	4G-1	4T	2G-1
BLOB子串	从UDF访问	从UDF访问作为不透明类型的一部分来访问	使用SQL或从UDF访问

图4-36 三种产品的对象-关系和关系特征总结

推荐读物

Stonbraker和Hellerstein的《Readings》(推荐读物[9])给出了大量关于代替关系系统的数据模型和原型系统的基础性的文章。关于面向对象和对象-关系的更完整的论文集可以在Zdonik和Maier的[10]中找到。推荐读物[1]集中介绍了一个特定的面向对象数据库(OODBMS, 即O₂), 在其第1章中包括了“面向对象数据库系统宣言”。其他读物则是关于具有对象-关系能力的特定的产品DB2 UDB ([2]和[3]), INFORMIX ([4]和[5]) 和ORACLE ([6]、[7]和[8]) 的。

- [1] Francois Bancilhon, Claude Delobel, and Paris Kanellakis, editors. *Building an Object-Oriented Database System, The Story of O₂*. San Mateo, CA: Morgan Kaufmann, 1992.
- [2] Don Chamberlin. *A complete Guide to DB2 Universal Database*. San Francisco: Morgan Kaufmann, 1998.
- [3] *Db2 Universal Database SQL Reference Manual*. Version 6. IBM, 1999. Available at <http://www.ibm.com/db2>.
- [4] *INFORMIX Guide to SQL Syntax*, Version 9.2. Menlo Park, CA: Informix Press, 1999. (Includes SPL syntax).
- [5] *INFORMIX Guide to SQL: Tutorial*. Version 9.2 Menlo Park, CA: Informix Press, 1999.
- [6] *ORACLE8 PL/SQL User's Guide and Reference*. Redwood City, CA: Oracle.<http://www.oracle.com>.
- [7] *ORACLE8 Server Application Developer's Guide*. Redwood City, CA: Oracle.<http://www.oracle.com>.

- [8] *ORACLE8 Server SQL Language Reference Manual. Volume 1 and 2.*
<http://www.oracle.com>.
- [9] Michael Stonbraker and Joseph M. Hellerstein, editors. *Readings in Database Systems*, 3rd ed. San Francisco: Morgan Kaufmann, 1998.
- [10] Stanley B. Zdonik and David Maier, editors. *Readings in Object-Oriented Database Systems*. San Mateo, CA: Morgan Kaufmann, 1998.

习题

下面的部分习题的解答在本书最后的“习题解答”中给出，这些有答案的习题用•号标注。除非特别声明，否则习题可以用ORACLE或INFORMIX来完成。

4.1 下面是一些对people表的查询。

- (a)• 查找教名（中间名）为空的人的ssno。
- (b) 查找名字为空的人的ssno。
- (c)• 使用对象等价检测人名的方式查找名字相同的人的ssno对。
- (d) 查找关于年龄最大的人的一切信息，并以person对象的形式显示。

4.2 假定student_t类型中有一个person_t类型以及一个整型的学号ID、一个电子邮件名字和预期的毕业年份（整数）。其中person_t不能为空。

- (a)• 定义student_t类型和一个student表（ORACLE中的对象表或INFORMIX中的类型表），以学生ID作为主键。
- (b) 给出插入一个学生的语句，该学生名为Susan A. Morris，年龄为25岁，社会保障号和学号都为231458888，电子邮件名为“samorris”，预计于2001年毕业。

4.3 （仅用ORACLE）考虑习题2.5的查询。确定哪一个可以用REF完成，并将其写出。可以使用3.6节描述的ORACLE支持的SQL-92的扩展。注意带点号的问题本章和第2章一样都有答案。

4.4 汇集上的查询：

- (a)• 查找仅为雇员的家属的家属。
- (b) 检查employees中的任一雇员是否被列为他/她自己的家属，如果是，则显示出该雇员的eid。并使用对象间的比较。
- (c)• 计算phonebook中的电话扩展号为100的人数。

4.5 对汇集的高级查询：

- (a)• 查找和一个或多个家属同名的雇员的eid。
- (b) 查找家属年龄比他/她自己大的雇员。
- (c)• 查找并显示100雇员的最老的家属。
- (d) 显示phonebook中的扩展号大于1000的人的ssno。
- (e)• 显示phonebook中的扩展号在800到900之间的人的ssno。

4.6 用户定义函数和方法：

- (a)• 查找在三个以上的矩形中出现的点，显示这样的点和覆盖这些点的矩形数，并按数量从高到低排序。
- (b) 查找覆盖的点最多的矩形。

- (c)* 查找被面积最大的矩形所覆盖的点。
 - (d) 按面积由大到小显示矩形。
- 4.7* (仅用ORACLE) 尽管我们希望例4.4.9的查询能显示一些对象方法的思想和功能,但是,方法对查询来说并不是必须的。用交互式SQL写出完成同样功能的查询。
- 4.8 (仅用ORACLE) 考虑在点和矩形间的一个“above”操作:若一个点的y值大于($>=$)矩形点的y值且在矩形的x值之间,则这个点在矩形上。将这个操作转换为方法。
- 4.9 建立一个requests表, requests表的每一行有一个名为wds的以varchar(30)定义的包含匹配的词集合的SET类型列, 和一个整数请求的ID reqid。
- (a) 构造并检证一个匹配请求文档的Select查询, 即查找包含请求中所有关键词的文档。
 - (b) 实现相交记数函数或成员, 并构造一个匹配部分关键词的查询, 特别地, 再构造一个至少匹配一半的请求关键词的查询。

第5章 访问数据库的程序

对SQL语言的改进目的之一是为了使最终用户能够使用SQL对数据库进行特定的查询。以前的查询功能依赖于应用程序，并且大多数数据处理中心存在积压工作，以至于新的应用程序要等上数月的时间。回忆一下，在第1章中，我们曾把一个通过交互式SQL访问数据库的最终用户称为临时用户。由于一些原因，事实证明直接通过SQL访问数据的最终用户数量并不如预想中那么多，相反，大多数用户仍然通过应用程序接口访问数据。通常把通过一个菜单应用程序访问数据库的用户称之为初级用户，这一菜单应用程序在显示器上提供了各种表格选择，从而避免了使用SQL中Select语句的语法。程序员正是出于这一目的而书写的程序称为菜单应用程序，或表单应用程序。从程序员的观点来看，应用程序逻辑通过一个表格界面与最终用户进行交互，确定用户的要求，然后通过执行SQL语句完成这些要求。大多数与数据打交道的人，诸如银行职员、机票预定代理人，都是使用表单应用程序来完成他们的工作的。

嵌入式SQL是指在程序内执行的SQL语句，它强调这些SQL语句是“嵌入”在编程语言（或宿主语言，例如C，Java，COBOL，Fortran）的常规语句中的。由于程序员发现嵌入式SQL提供了一个比在关系模型存在之前访问数据更加容易的平台，使用这种形式可以实现许多SQL语句早期的功能。Java中标准的嵌入式SQL称为SQLJ，可以在ORACLE、INFORMIX、DB2 UDB中使用，也可以通过JDBC在Java中使用SQL，JDBC是标准Java的一部分，用来连接到任何数据库或数据库的一部分。在本章中，我们将重点放在C语言编写的嵌入式SQL程序。如果你不是完全熟悉C语言，只要你有其他语言的编程经验，这应该是一个不难克服的障碍，因为我们考虑的程序是相当简单的。由于只有个别的格式上的不同，如输入名字或输出格式，因此只要从本书中（或从在线的例子中）拷贝程序格式，就足够你完成大部分的任务了。同时，你应当注意本章中提到的新概念，特别是如果你缺少C语言背景知识的话，你可能需要特别努力地掌握细节。有一些很好的C语言书可供参考，特别推荐一本导论性的书《*The C Programming Language*》，由Kernighan和Ritchie著。

现在让我们来看一下大多数一般的数据库任务都选择表单应用程序而不是交互式SQL的原因。首先，使用交互式SQL，必须要清楚地知道所有表、列名并且能够写出符合语法的SQL语句，这一点分散了专家在实际工作上的注意力。设想一个机票预订代理人想要写即席的SQL语句来预订班机、安排座位、安置顾客进入等待队列、处理不同的付款方式；等等。即使是最有经验的数据库系统程序员通常也利用表单应用程序执行一些支持工作，诸如追踪出错报告；与写即席的SQL语句相比，这是对技能要求较低的一个接口。

倾向于应用程序而不用即席SQL的第二个原因，在于许多我们认为简单的任务并不能仅仅通过非过程的SQL语句来完成，而是经常要使用一些程序循环。例如，假设你需要得到某一列值的中值。所谓一组值的中值是指这样的值，这组值中至少有一半的值小于等于它，至少有一半的值大于等于它。因为不存在内部函数MEDIAN，所以通过一条Select语句是不能得到这一中值的。在本章中，我们将看到怎样通过使用嵌入式SQL的简单程序来完成这样的任

务。另外，一些任务需要写一组SQL语句来读取不同的行，然后根据所读到的内容来更新其中的某些行，这时必须保证其他用户所做的修改不会破坏结果。我们需要将这一组SQL语句捆绑在特定的不可分割的称之为事务的包内。在这一方面，嵌入式SQL的过程性功能显得尤为重要。

第三点，倾向于使用表单应用程序还因为它使我们摆脱了书写SQL语句时遇到的一些复杂的语法（例如，用于实现FOR ALL条件的形式），以及在书写过程中时常会出现错误的情况。在许多商业领域，不正确的查询（更糟的是，不正确的更新语句）会导致可怕的结果。显然，我们希望让专业程序员来处理这些语法问题，他们既有时间也有能力写出正确的SQL语句。同时既然这些语句在同一领域被反复使用，我们可以获得某种一致性，这种一致性在多数商业领域极为重要。

而另一方面，在有些特殊情况下，却倾向于使用交互式SQL。某些特别的用户需要使用不寻常的、复杂的查询，而一般的编程接口不能提供这些查询。例如，图书管理员需要为借阅者查找他们所需要的书，显然手工查找已被证明效率极低而不再被使用。另外由于上文提到的应用程序的积压，在表单应用程序尚未书写完成的特殊情况下，交互式SQL将为用户提供工具。然而，在这种情况下，主要考虑可能产生错误结果的风险，因而实现能够处理这些特殊情况的应用程序应该一直是我们的一个目标。

在介绍嵌入式SQL的具体功能之前，先让我们来看一下想要得到的是什么。本章的目标是提供给你在嵌入式SQL程序中实现任何算法所需的技术。在我们向这一目标迈进的过程中，需要不断地考虑一些复杂的技术。为了避免在这些具体细节中迷失方向，你必须记住这些技术本身并不是我们的目标，能够知道嵌入式SQL语句中每一个新功能在实现访问数据库的应用程序中所起的作用，这才是重要的。在5.1节中我们介绍了在程序中使用Select语句访问数据的基本语法；为在一个循环中访问多条记录而对Select所作的微小改动（通过声明一个游标抽取多条记录）。在5.2节中我们将学习如何处理各种不同的出错返回，同时学习如何识别其他条件类型，例如通过游标检索多条记录时的循环结束条件，或者是当读取的记录中某列出现空值时的条件。在5.3节中，我们详细地给出了一些嵌入式SQL语句的一般格式，包括新的Update语句的格式。

在5.4节，我们考虑了前面提到的有关事务的一些细节。5.5节给出了用程序实现的在非过程式SQL中不能完成的一些任务的例子。

5.6节提到了动态SQL。在前面有关静态SQL的章节中，所有的嵌入式SQL语句必须在编译之前就在程序源文件中以文字语句的形式存在。而动态SQL提供了更大的灵活性，允许嵌入式SQL语句中有字符串变量，以便在程序编译时具体内容是不能预知的SQL语句能够根据用户需求的改变而动态地建立并执行。动态SQL的确比较复杂，这正是我们将它放到后面章节的原因。

本章中，你将会在学习过程中碰到两种数据库系统间的某些区别，这两种数据库系统分别是ORACLE和DB2 UDB。所有其他主要的DBMS产品也支持嵌入式SQL。如果我们的描述应用于某一特定的系统时，我们将会指出；如果对一个嵌入式SQL功能描述时没有这样的说明，就意味着这一功能是基本SQL的一部分（可参见第3章第一段）。本章中所涉及到的基本嵌入式SQL语句与X/Open、SQL-92和SQL-99中的很接近，唯一的区别在于使用游标时出现的WITH HOLD子句，这在5.4节的最后讨论。在附录C中按字母顺序给出了不同的SQL语句

格式，可作参考，而不必依赖于以后章节中的叙述顺序。在不断学习嵌入式SQL的过程中，应该不时地参考附录C。附录B中涉及的在ORACLE和DB2 UDB中编写嵌入式程序时与产品有关的问题。

5.1 C语言中嵌入式SQL的介绍

当程序员在宿主语言源文件中写入嵌入式SQL语句时，在SQL语句之前必须有exec sql短语；EXEC SQL语句可以扩展到该文件的多行。例如，以下是C语言中嵌入式SQL语句的一个例子：

```
[5.1.1] exec sql select count(*) into :host_var
      from customers;
```

注意到C语言的源文件通常是小写的，所以尽管在许多文本中使用大写的SQL语句，我们仍然遵循C语言的这一习惯，使用小写。语句格式自由、可以跨过多行、以分号作为结束，这是C语言语句的标准。而在COBOL中，语句通常并不像C语言中那样跨过多行、格式自由，并且它的嵌入式SQL语句是以end-exec作为结束的。

接下来，我们将描述一些实现一个非常简单的嵌入式SQL程序、执行一条Select语句然后打印出结果所需要的特征。

1. 使用嵌入式SQL的简单程序

首先我们介绍一些基本的语言特征，为讲述例5.1.1中的程序作准备。必须注意到C语言编译程序通常是不能识别嵌入式SQL中的EXEC SQL语句的，因而通常源文件需要首先通过预编译程序，将这些嵌入式语句转换为对数据库引擎的C语言函数调用。

(1) 声明部分

在嵌入式SQL程序中，程序逻辑可以使用普通的程序变量，即宿主变量，来表示SQL语句中的一些语法元素。例如，我们可能需要声明一些程序变量来存放查询得到的值，比如说cname（使用变量cust_name，具体结构将在后面解释）、discnt（使用变量cust_discount），以及用户在WHERE子句中提供的值cid（使用变量cust_id）。这样我们就可以通过嵌入式SQL语句来完成检索。

```
[5.1.2] exec sql select cname, discnt into :cust_name, :cust_discount
      from customers where cid = :cust_id;
```

注意到这一Select语句中新的INTO子句，以及在嵌入式SQL语句中使用宿主变量时必须在它们之前加冒号(:)。执行这一语句，如果字符串变量cust_id已在这-C程序之前被设置为c001，从图2-2中的customers表可以看到变量cust_name和cust_discount分别被赋予了值“TipTop”和10.000。

要在嵌入式SQL语句中使用这些宿主变量，必须先声明它们，包括对预编译程序的一个声明（需要特殊的对待）。我们在C语言程序中一个特殊的嵌入式SQL声明部分声明这些变量，所有这样的声明都在那里完成。下面给出了声明在[5.1.2]的Select语句中所需要的变量的例子。

```
[5.1.3] exec sql begin declare section;
      char cust_id[5] = "c001"; /* declare and initialize cust_id */
      char cust_name[14];
```

```

float cust_discnt;
exec sql end declare section;

```

这三个变量的声明是标准的C语言的声明，出现在Begin Declare语句和End Declare语句之间，这是预编译程序和C语言编译程序都能够理解的相同格式。嵌入式SQL语句中使用的宿主变量的数据类型必须能被数据库系统识别，这一点很重要。有些情况下可能需要对得到的数据库值进行类型转换，转换为宿主变量的数据类型。例如，如果我们改变[5.1.3]中对cust_discnt的声明，使它只能接受整型数据。

```
int cust_discnt;
```

然后执行[5.1.2]中的Select语句，这时变量cust_discnt的值将是10，而不是10.00。然而，这会丧失一些精确性，比如，如果客户c009的discnt值为11.3，而通过[5.1.2]中的Select语句得到的cust_discnt结果为11。这些我们都已相当熟悉了，但是对于在嵌入式SQL语句中接受字符串的C变量格式还需要许多说明，我们在看过第一个程序的例子之后才提到这些说明。

(2) 在SQL中建立连接和释放连接

在一个嵌入式SQL程序的开始，程序逻辑面临着和交互式用户同样的问题：怎样与SQL数据库管理系统和正确的数据库建立连接。接下来，我们将给出一些在ORACLE和DB2 UDB中使用SQL-99的例子。在完全SQL-99中连接到SQL数据库的语法如下：

```
EXEC SQL CONNECT TO target-server [AS connect-name] [USER username];
```

或

```
EXEC SQL CONNECT TO DEFAULT;
```

“target-server”是由数据库管理员提供的一个字符串名，“connect-name”是你为这次数据库的连接所起的一个名字，以后将引用到它（一次有不只一个连接打开是可能的），“username”是鉴别你是否为数据库用户的一个字符串。当你不需要在程序中引用到connect-name时，你只有一个连接打开，那么就可以使用CONNECT TO DEFAULT格式。因为不同平台在用户识别、权限要求方面的可变性，Connect语句并不是Entry SQL-92和Core SQL-99的一部分。然而，SQL-99的连接管理特征提供了Connect语句的标准的使用方法，正如我们在后面所见到的。

在ORACLE和DB2 UDB中，字符串常量通常不能用作Connect语句的参数。我们需要如下的声明：

```

exec sql begin declare section;
      . . .
      /* unaliased declarations */ 
      char user_name[10], user_pwd[10];
exec sql end declare section;

```

假设口令为“XXXX”，我们将以如下语句对上述变量初始化：

```

strcpy(user_name, "poneilsql"); /* set user_name string */
strcpy(user_pwd, "XXXX");       /* set user password string */

```

那么在ORACLE中嵌入式SQL的Connect语句为：

```
exec sql connect :user_name identified by :user_pwd; -- ORACLE form of Connect
```

对于同样的声明，在兼容SQL-99连接管理特征的数据库系统，诸如DB2 UDB中，若要连接到数据库mydb，应该书写如下语句：

```
exec sql connect to mydb user :user_name using :user_pwd;
```

数据库的名字同样也可以通过一个声明的字符串变量给出。当用户名或口令不需要时，DB2 UDB也允许一个没有用户名或口令的较为简短的格式。注意，在这一SQL语句中使用宿主变量作为参数时所用到的冒号。断开与数据库连接的基本SQL语句SQL Disconnect的格式如下：

```
exec sql disconnect connect-name;
```

或

```
exec sql disconnect current;
```

然而，在使用Disconnect语句之前，必须对成功的事务执行Commit语句，或对不成功的事务执行Rollback语句撤销不成功事务中已做的部分工作。否则，断开连接操作会失败。我们将在5.4节中详细介绍Commit和Rollback语句。这里我们只是简单地提一下，对于成功的任务结束Commit与Disconnect之间的重要组合。

```
exec sql commit work;
exec sql disconnect current;
```

以及对于失败的任务结束，Rollback与Disconnect连接之间的重要组合。

```
exec sql rollback work;
exec sql disconnect current;
```

ORACLE中有一条语句同时执行一条Commit和一条Disconnect：

```
exec sql commit release; -- ORACLE form of success Disconnect
```

在ORACLE版本7之后就必须使用这种格式的Disconnect语句。如果Disconnect是在一个严重错误之后，我们就使用下面的格式释放连接（Rollback 和 Disconnect）：

```
exec sql rollback release; -- ORACLE form of failure Disconnect
```

DB2 UDB既有SQL-99格式的释放连接语句，也有自己如下的释放连接语句格式：

```
exec sql connect reset; -- DB2 UDB form of success Disconnect
```

例5.1.1给出了目前为止介绍的嵌入式SQL特征和一些新的概念。为简单起见，在这个例子以及大多数例子中，我们都会忽略由于程序错误无效的数据库信息或不正确的用户输入而导致运行中可能出现的错误。在例5.1.1的程序中，我们对执行嵌入式SQL语句时可能产生的错误用两条语句作了处理：exec sql include sqlca和exec sql whenever sqlerror goto report_error。在程序开始，exec sql include sqlca语句为特定的错误和统计信息分配空间，它同数据库系统监测之间进行通信，称之为通信区(communication area)。对于真正支持SQL-99的系统来说，它是不需要的，然而目前的系统通常都需要它。exec sql whenever sqlerror goto report_error语句建立了一个错误中断条件，使程序在发生这一错误时跳转到显示数据库出错信息的代码处。5.2节介绍了SQL出错处理。

例5.1.1 考虑图5-1所示的嵌入式SQL程序。程序不断提示用户输入一个顾客的cid，显示顾客的名字和应有的折扣作为回答，当用户输入一个空字符串时程序终止。

图5-1中的大部分语法已经介绍过了。字符串变量的格式将在下一小节介绍；与用户交互的函数prompt()也将在后面的小节中介绍。注意到main()函数的循环语句最后，即在从表中进行了一系列的读操作之后，而在任何一个与用户交互的动作(如显示结果)之前，执行语句

exec sql commit work。在本章的后面将会说明，在某一条记录数据被读时，必须对这一条记录加锁来防止由于其他用户更新数据库而导致提供给读取数据库的用户不一致的数据视图。我们需要在循环中找到某处释放这些读锁，以免它们无限制的累积从而导致其他用户不能够更新那些很早就已被读取了的记录。释放锁的一个好时机就是在与用户交互之前。

```
#include <stdio.h>
/* header for prompted-input function */
#include "prompt.h"
/* header for database system's SQLCA structure */
exec sql include sqlca;
char cid_prompt[] = "Please enter customer id: " /* declaration unknown to SQL */

int main( )
{
    exec sql begin declare section; /* declare SQL host variables */
        char cust_id[5], cust_name[14]; /* character strings */
        float cust_discnt; /* host var for discnt value */
        char user_name[20], user_pwd[20]; /* for Connect */
    exec sql end declare section;

    exec sql whenever sqlerror goto report_error; /* error trap condition */
    exec sql whenever not found goto notfound; /* not found condition */
    strcpy(user_name, "poneilsql");
    strcpy(user_pwd, "XXXX");
    exec sql connect :user_name
        identified by :user_pwd; /* ORACLE Connect */
    while((prompt(cid_prompt, 1, cust_id, 4)) >= 0){ /* main loop: input cid */
        exec sql select cname, discnt
            into :cust_name, :cust_discnt /* retrieve cname, discnt */
            from customers where cid = :cust_id;
        exec sql commit work; /* release read lock on row */
        printf("Customer's name is %s and discount is %5.1f\n", /* print values */
            cust_name, cust_discnt); /* NOTE: no initial colons */
        continue; /* back to top of loop */
    notfound: printf("Can't find customer %s, continuing\n", cust_id); /* loop again */
    }
    exec sql commit release; /* ORACLE Disconnect from database */
    return 0; /* indicate success of program */
report_error:
    print_dberror(); /* print out error message */
    exec sql rollback release; /* ORACLE Disconnect in error */
    return 1; /* indicate failure of program */
}
```

图5-1 ORACLE嵌入式SQL程序

print_dberror()函数显示数据库系统的错误消息，如ORACLE中的“ORA-00942:table or view does not exist”（表或视图不存在）。在附录B.2中给出了ORACLE和DB2 UDB中的print_dberror()函数的代码。

注意到只有那些以ORACLE开头的注释行所对应的语句才是使用了ORACLE特定的语法，即Connect和Disconnect语句。其他所有的exec sql语句行都是在各个数据库系统之间可移植的。对于DB2 UDB和数据库abc，应将Connect语句替换成如下形式：

```
exec sql connect to abc user :user_name using :user_pwd;      -- DB2 UDB Connect
```

或者如果你的数据库使用操作系统所提供的用户授权，那么就使用较为简单的格式：

```
exec sql connect to abc;      -- DB2 UDB Connect, simple form
```

将ORACLE的成功的Disconnect替换成如下形式：

```
exec sql connect reset;      -- DB2 UDB success Disconnect
```

或者SQL-99的格式：

```
exec sql commit work;
exec sql disconnect current;
```

最后只需将这两行中的commit替换成rollback就完成了事务失败情况下的Disconnect功能。

(3) 数据库系统和C语言中的字符串

这里我们通过图5-1的程序给出如何处理字符串。由于C语言和数据库对字符串的处理是有区别的，因而这是一项复杂的工作。

C语言字符串是由一个字符类型值的序列构成的，一个8位的整数，即一个字节代表一个单独的字符编码，字符串以一个空字节（所有8位都为0）作为结束。正是由于这个原因，C语言中的字符串有时被称为以空字符结束的字符串。在C中大多数的字符编码值是用由单引号括起来的字符所代表的。如，字符代码'c'代表了ASCⅡ字符编码中的整数99，而字符代码'0'代表了整数值48。空字符串用'\0'表示，整数值为0。（不要混淆空字符'\0'和SQL中的空值。）

按照惯例，所有C语言库的字符串函数都要求使用以空字符结束的字符串。在[5.1.3]的初始化语句，声明和初始化了一个5个字符的数组cust_id[5]，字符串的值为“c001”，产生了如下的字符数组：

cust_id:	'c'	'0'	'0'	'1'	'\0'
----------	-----	-----	-----	-----	------

一个字符串常量，例如“c001”，包含在双引号内，代表了一个指向由C编译程序建立的以空字符结束的字符串，同样一个数组名如user_name是指向它内容的一个指针。一般情况下，C语言中的字符串是通过它们的字符串指针来处理的。如在图5-1程序中使用的strcpy()函数：

```
strcpy(user_name, "pone1sql");
```

将右面字符串参数中的所有字符依次复制到左边的字符串参数（一个未被初始化的字符数组）中，包括终止符'\0'。如果函数调用拷贝或显示一个没有空字符结尾的字符串，它的处理将尝试由字符串指针所指的位置开始，直到遇到某一8位都是0的字节（空字符，即'\0'）。

数据库系统通常在Create Table语句中以char(n)或varchar(n)的类型声明的列中使用完全不同的格式代表字符串。例如，在表customers中，列cid的类型定义为char(4)，占用了4个字符，没有空结束符。如果我们在customers中插入新的一行，其中cid的值设置为“c9”（本章后面我们将介绍如何实现这一操作），那么在cid中实际存储的字符为‘c’，‘9’，‘ ’，‘ ’，最后两个字符用空格填充。因为数据库事先知道字符串的长度，所以这里并不需要空字符结束符来标记字符串的结束。数据库中的列若是变长字符串类型，如表customers中的cname

列的字符串长度不是一个常量；由一个2字节长度的整数计数器来从字符串头开始记录字符个数。由于有了这个计数器，同样也不需要空字符结束符标记字符串结束。

目前当用嵌入式SQL语句将字符串列中得到的值读到C数组时，大多数的数据库系统会把列值转换成以空字符结束的字符串，这样C语言就能够正常地处理这一字符串。这意味着要从SQL中接受字符串列值的C数组变量必须能有足够的数组空间容纳相应表列值的每一个字符，包括'\0'。在第3章的Create Table语句[3.2.1]中定义了表customers的cid列类型为char(4)，所以在[5.1.3]的Declare Section语句中对cust_id的声明如下：

```
char cust_id[5]
```

一些数据库系统不能完成这一转换是很可能的。如果你发现从数据库中得到的字符串打印出来带有“垃圾”，就应该查看一下所使用的数据库系统的正确转换规则。

(4) 提示用户交互

我们将解释在例[5.1.1]中提到的用于提示用户进行交互的prompt()函数（这一函数的C代码在附录B.1中给出）。在例5.1.1中循环的开始是这样的：

```
while((prompt(cid_prompt, I, cust_id, 4)) >= 0)
```

函数prompt()将向用户显示在cid_prompt数组中的消息，这一数据已在程序前面被初始化过了：

```
Please enter customer id:
```

用户键入的对这一提示进行响应的任一行都被解释成用空白符分隔开的连续标记(token)。空白符包括一个或多个空格、跳格或是换行符，在C中分别以' '、'\t'、'\n'来表示。标记是内部没有嵌入空白符的字符串。在输入的字符串中，多个标记是用空白符(通常是空格)来分隔的。在例5.1.1中，函数prompt()期望得到一个标记(因为函数的第二个参数为1)，然后将它读到cust_id数组中(第3个参数)，数组可容纳4个字符(第4个参数)加上1个空字符结束符。因此，如果用户在看到提示信息后键入

```
c003
```

cust_id将被填充成一个C字符串“c003”。如果用户键入

```
c003 wxyzabcdef
```

cust_id会得到同样的值。在这里，prompt()函数仅仅查找一个标记，当看到第一个时，就将它返回。现在如果用户在提示行上键入一个简单的回车，或者一个过分长而不能全部放入数组内的字符串(多于4个字符)，那么返回结果会小于0，并且循环中的While条件将会是FALSE，循环将终止。这里，在提示行键入回车是推荐的一种结束提示循环的方法。

函数prompt()也可接受用户输入的多个标记。例如，图5-13的程序提示输入两个账号字符串和一个美元金额。要想了解函数prompt()更复杂的使用方法和它的源代码可以参考附录B.1。

(5) 预编译和编译过程

我们在前面提到过，C语言编译程序不能识别嵌入式exec sql语句的语法，所以源程序必须先通过预编译程序将这些嵌入式语句转换成C中的正确语句。若想获得更多的信息，可参考附录B.3。

2. 使用游标选择多行

只有当Select语句保证每次最多读一行到一个列变量的集合时，才能使用例5.1.1中的嵌入式Select语句格式。我们如何在程序中检索多行呢？使用交互式查询从表中检索多行显示到屏

幕上是一件比较容易的事，但是在程序中我们必须明确指定要将每一行的每一列放置在哪里，这样我们就可以通过名字来访问。可以想到，我们可以通过声明数组然后检索所有数据行到这些数组，但这不是一个好方法——我们一般不可能预先知道这些数组应该有多大，甚至也不知道检索出来的数据是否能一次全部放入一个基于内存的数组中。实际上，你应该把这一点作为嵌入式SQL编程的原则。

一次一行原则 在嵌入式SQL程序中，每当检索未知数量的行时，程序员应该假设这些记录不能一次全部放入任一声明过的数组中，并且程序的设计也应该反映出这一假设。

如何处理未知个数的行这一问题让我们想到了从文件中处理未知个数的输入值的方法，我们通常是通过一个循环每次处理一个值。由于程序是以顺序的、每次一步的方式执行的，很显然，从一条Select语句中我们总是每次查看一行。当我们在未知个数的行上执行Select语句时，每次从记录集中检索一行，需要一个游标记录当前位置。

对一个由用户交互提供的cid所唯一确定的特定客户，我们想要检索列出为该客户提供订货的代理商aid的一行，同时列出每个代理商所提供给客户的订单金额总值。下面我们声明一个名为agent_dollars的游标来取代简单的Select语句：

```
exec sql declare agent_dollars cursor for
    select aid, sum(dollars) from orders
    where cid = :cust_id group by aid;
```

Declare Cursor语句通常意义上是一个声明，它一般放于程序中较前面的地方。在游标被声明之后，它仍然不是活跃状态。在程序开始检索信息之前，必须执行一条SQL语句来打开这个游标。游标被打开之后，程序就可以使用取(fetch)操作从游标处每次检索一行；当程序完成了检索之后，应该立即关闭游标。首先，看一下打开游标的语句：

```
exec sql open agent_dollars;
```

在程序运行过程中碰到Open Cursor语句时，向数据库系统监测器发出一个调用，以准备执行agent_dollars游标的Select语句。如果这一查询依赖于一些宿主变量（在本例中是cust_id），那么这些宿主变量在这时被赋值。有一点很重要必须认识到，以后改变cust_id的值将对检索得到的一系列行不产生任何影响。一旦游标被打开了，我们就可以使用如下的Fetch语句开始检索连续的记录行：

```
exec sql fetch agent_dollars into :agent_id, :dollar_sum;
```

Select语句中的INTO子句检索单独的一行。在检索多行的情况下，INTO子句是加入到Fetch语句中去的，而不是作为游标的一部分被声明的。这带来了最大程度的灵活性，因为可以想到我们可能会在不同的情况下将从一个游标取行值到多于一个的变量集合中。很明显INTO子句属于Fetch语句，而Fetch语句正是从行中检索值。由于对于一次打开的游标数量有限制，因此当一个游标所有的Fetch语句都执行之后，应该执行Close Cursor语句。当程序结束时，所有打开的游标会被关闭。

游标通常可以被看做是指向最近刚刚被检索到的行，或者在游标第一次被打开的情况下，恰恰指向位于第一条记录之前的位置。在随后的提取调用中，游标的位置首先加1，然后这条记录中的值被检索到指定的宿主变量中。有可能出现这样的情况，一个新的提取操作将游标加1，发现没有行可被检索了，这一事件的状态被返回给程序。这一动作与许多程序从文件检索任意数量的值的情况完全类似。例如，在C中，函数getchar()从标准输入文件中检索字符，

就像磁盘文件那样工作，只是这里是从键盘读取输入。如果标准输入文件被重定向到从普通磁盘文件输入，那么当函数getchar()在读取文件时发现已读完所有字符时，它返回一个特殊的EOF值。程序员的责任在于检测每个返回值，当遇到EOF时，开始新的处理。在SQL的Fetch语句中，标明游标已到最后的状态不是通过函数返回值的形式来表示的，而是通过特别的条件处理方法，我们将在下一节介绍。

SQL通信区：SQLCA

SQL通信区，或称为SQLCA，是一个已被声明过的内存结构（C中的结构），它的成员变量用来在数据库系统监视器与程序之间进行信息通信。虽然早在20世纪90年代初，SQL标准就不赞成使用SQLCA结构，但是这一结构仍然在商业数据库产品中广泛应用（如ORACLE和DB2 UDB都支持SQLCA），所以在我们的程序前部也包括了如下语句。SQLCA在嵌入式的C程序中声明，通常写在其他的外部声明语句前面，使用如下的Include语句：

```
exec sql include sqlca;
```

接下来的例子和程序都将解释其中一些概念。

例5.1.2 图5-2给出了一个检索多行的程序，GROUP BY Select语句列出了代理商的ID值，以及由这些代理商提供给某一用户指出的顾客的订单金额总值。与例5.1.1不同的是，这一程序是用DB2 UDB的格式来写的。很容易将它改成ORACLE的格式，只要改变Connect和Disconnect语句。 ■

5.2 条件处理

Wheneve语句使我们在遇到出错和其他的情况（如数据已被读完）时，控制程序的运行。通常的格式如下：

```
EXEC SQL WHENEVER condition action;
```

在例5.1.1和5.1.2中使用到的这一语句的例子如下：

[5.2.1] exec sql whenever sqlerror goto report error;

Wheneve语句的作用是设置一个“条件陷阱”，这样会对所有后面由EXEC SQL语句所引起的对数据库系统的调用自动检查它是否满足出错条件。如果存在这样的出错条件，就必须采取相应的动作；在[5.2.1]中，就是GOTO动作。

以下是对在Wheneve语句中出现的条件和动作的说明。

(1) 条件

- SQLERROR 在由Wheneve语句所覆盖的每一条EXEC SQL...调用之后检测是否有出错情况。这些错误通常是由编程错误所引起的。错误代码的意义依赖于某一特定的DBMS。ORACLE中的错误代码可在推荐读物[6]中找到。例如，执行exec sql connect可能产生的出错返回为“-02019:database(link) does not exist.”^Θ。
- NOT FOUND 执行某一条SQL语句，如Fetch、Insert、Update或者Delete，检测是否没有记录受到这一SQL语句的影响。这一条件出现在Entry SQL-92和Core SQL-99中。
- SQLWARNING 除了上述的NOT FOUND条件检测之外，它还检测一个不是错误但应

^Θ 这个条件曾经出现在Entry SQL-92中，但不在SQL-99中，它已由SQL EXCEPTION代替（虽然这不是Core SQL-99的内容）。这是表明这些标准缺乏向上兼容能力的极其罕见的情况之一。

引起注意的条件。注意，SQLWARNING并不是Full SQL-92的一部分，并且在X/Open SQL标准中也不是必需的，但它被定义为SQL-99的一个扩展特性；而且，被广泛地实现于各种数据库产品中，包括ORACLE、DB2 UDB和INFORMIX。要使用SQLWARNING可能在你的程序中需要exec sql include sqlca（对于SQLERROR，这通常是不需要的）。SQL WARNING的例子有“在集合函数求值时至少删去了一个空值”以及“在Select语句中一个列值被截取后赋给了一个宿主变量”。

```
#define TRUE 1
#include <stdio.h>
#include "prompt.h"
exec sql include sqlca;
exec sql begin declare section;
    char cust_id[5], agent_id[4];
    double dollar_sum; /* double float variable */
exec sql end declare section;

int main( )
{
    char cid_prompt[] = "Please enter customer ID: ";
    exec sql declare agent_dollars cursor for
        select aid, sum(dollars) from orders /* cursor for select */
        where cid = :cust_id group by aid; /* note: depends on cust_id */

    exec sql whenever sqlerror goto report_error; /* error trap condition */
    exec sql connect to testdb; /* DB2 UDB Connect, simple form */
    exec sql whenever not found goto finish; /* end of cursor trap condition */
    while (prompt(cid_prompt, 1, cust_id, 4) >= 0) { /* main loop. get cid from user */
        exec sql open agent_dollars; /* cust_id value used in open cursor */
        while (TRUE) { /* loop to fetch rows */
            exec sql fetch agent_dollars /* fetch next row and ... */
            into :agent_id, :dollar_sum; /* ... set these variables */
            printf("%s %11.2f\n",
                   agent_id, dollar_sum); /* print out latest values */
        }
    }
    finish: exec sql close agent_dollars; /* close cursor when done */
    exec sql commit work; /* end locks on fetched rows */
}
exec sql disconnect current; /* end of main loop */
return 0; /* Disconnect from database */
report_error:
    print_dberror(); /* print out error message */
    exec sql rollback work; /* failing, undo work, end locks */
    exec sql disconnect current; /* Disconnect from database */
    return 1;
}
```

图5-2 检索多行的DB2 UDB程序（例5.1.2的图解）

(2) 动作

- CONTINUE 这意味着对于相应的条件不采取任何动作，程序按正常流程继续。

- GOTO 标号 它的效果同C语言中的“goto 标号”语句。必须注意到，这一标号必须在所有后续的EXEC SQL语句到又遇到另外一条拥有同样条件的Whenever语句的范围内。
- STOP 这一动作结束程序的运行，撤消当前的事务，并且断开与数据库的连接。它并不显示出错信息，所以退出程序看上去很神秘。这一动作在ORACLE和INFORMIX中都有定义，但在DB2 UDB 和基本SQL中没有。
- DO函数 (ORACLE), CALL函数 (INFORMIX) 条件引发一个对已命名的C函数的调用。这一函数返回之后，程序的控制流程从引发这一条件的EXEC SQL语句之后的那一条语句继续下去。这一动作在DB2 UDB和基本SQL中都没有定义。

1. Whenever 语句：控制的作用域和流程

我们来描绘一下预编译程序如何通过在后续的运行时数据库系统调用后插入检测来实现带有不同于Continue的任意动作的Whenever语句的条件陷阱功能。这些检测将SQLCODE（与特定产品相关）或SQLSTATE（可移植，已被标准化）与各种不同的值作比较。在遇到一条Whenever语句之后，预编译程序在顺序扫描源文件代码中的每一行时执行一个简单的逐句搜索和插入，只有当遇到一条后续的Whenever语句覆盖了前面的Whenever语句时，才可以改变程序的动作。特别注意的是，如果当插入的Whenever语句覆盖了原来的命令时，程序的动作并不跟随控制流程。

例5.2.1 考虑一个程序文件中的下述语句序列。

```
main()
{
exec sql whenever sqlerror stop;          /* first whenever statement */
.
.
.
goto s1
.

exec sql whenever sqlerror continue;      /* overrides first whenever */
s1: exec sql update agents set percent = percent + 1;
.

}
```

尽管标识S1处的Update语句是在第一条Whenever语句的作用域内由Goto所转移到的，此处对于条件sqlerror所应采取的动作是continue（第二条Whenever语句的作用）。这是因为预编译程序是在第二条Whenever语句的作用下设置的检测；控制程序的因素在于文件中语句的位置，而不是控制流程，预编译程序是不知道控制流程的。 ■

注意到在没有Whenever语句的情况下缺省的动作是Continue，意味着不需要采取任何特殊动作，按照正常的控制流程继续。此时没有必要通过对数据库调用的检测来得到这一动作。可以通过覆盖带不同动作的Whenever语句来重建这一缺省动作，如例5.2.1所示：

```
exec sql whenever sqlerror continue.
```

我们在使用Whenever语句时必须注意要避免无限循环。

例5.2.2 避免无限循环。以下的代码嵌在一个函数中，该函数被调用来在嵌入式SQL中创建一张表：

```
exec sql whenever sqlerror goto handle_error;
exec sql create table customers
```

```
(cid char(4) not null, cname varchar(13), . . .);
```

如果由于某个错误（例如，磁盘空间不够）而导致建表失败，程序的控制流程转到了如下标号的程序段：

```
handle_error:
    exec sql whenever sqlerror continue;
    exec sql drop customers;
    exec sql disconnect;
    fprintf(stderr, "Could not create customers table\n");
    return -1;                                /* return error condition */
```

这里的exec sql whenever sqlerror continue语句很重要，因为删除表customers可能会引起错误——实际上，如果我们没有成功地创建表的话，这是很有可能发生的。如果原来的Whenever语句还在起作用，那么程序又将跳转到handle_error标号处，重复尝试Drop语句，不停地循环。新的Whenever语句覆盖了Goto动作，所以不会发生循环。

注意，在C语言函数中当错误产生时，正确地退出处理是很重要的。在上文的Disconnect语句之后，return -1语句返回了出错的信息，返回0则表明函数已成功地完成。在收到返回值-1以后由程序员采取相应的操作。一般在Create Table语句失败之后不会有其他可选的动作，所以在这种情况下，一个带警告值的正常返回可以指出有必要采取这一动作。 ■

2. 显式的错误检测

Whenever机制的基础是一个提供了更多与条件有关的信息的错误编码系统。不幸的是，没有可以完全移植的方法使用这些代码。支持SQLCA的产品要填写sqlca.sqlcode（SQLCA结构的一个成员），或者一个与特定产品的错误码相关的SQLCODE变量。正如上文提到的，SQL-92、SQL-99和X/OpenSQL等标准一直都反对使用SQLCA，而要将一种新的报告错误的方法，称为SQLSTATE，设置为标准，但是一些产品还未使用它，或者在缺省设置时不支持它。特别在ORACLE中，它要求预处理程序运行时必须设置选项MODE=ANST才能支持SQLSTATE。在例5.2.3中我们看到，DB2 UDB提供SQLSTATE作为SQLCA的一个成员。在标准和所有支持它的产品中，SQLSTATE由一个5个字符的字符串编码而成，带有一个2个字符长的“类别编码”和一个3个字符长的“子类编码”。例如，“00000”代表没有错误，“82100”表示“内存溢出”，对于所有的产品这一编码都是相同的。你可以查阅你正在使用的数据库系统产品的嵌入式SQL参考指南来得到你所使用的数据库系统的SQLSTATE、SQLCA和SQLCODE，来确定所有主要的出错条件，以及它们是如何被报告的。

如果Whenever语句中所采取的动作不是CONTINUE，那么要在执行中某处显式地识别出出错状态，这通常是不可能的。在例5.2.3中我们可以看到原因。

例5.2.3 显式的错误检测。考虑下面的代码段：

```
exec sql begin declare section;
    char SQLSTATE[6];                      /* 5-char SQLSTATE needs 6-char C array */
exec sql end declare section;
exec sql whenever sqlerror goto handle_error; /* this is in force below */      */
. .
exec sql create table custs
    (cid char(4) not null, cname varchar(13), . . .);
if (strcmp(SQLSTATE,"82100") == 0)           /* ORACLE, out of disk space? */      */
    <call procedure to handle this condition> /* ** NEVER REACH THIS */
```

注意，预处理程序在语句Create Table之后立即设置了sqlerror检测（例如，`if(errcode<0)goto handle_error`来实现Whenever语句。因此很明显strcmp检测将永远不会执行到跟在这一比较语句后面的行。Whenever条件抢在我们企图要作的检测之前。这说明了如果我们想要识别某一明确的出错条件，就需要丢弃先前为SQLERROR所采取的GOTO动作，将上述的程序修改成如下，变动处用斜体表示。

```
exec sql whenever sqlerror goto handle_error;      /* this is in force below */
. . .
exec sql whenever sqlerror continue;                /* but this overrides it */
exec sql create table custs
  (cid char(4) not null, cname varchar(13),...); /* as above */
  if (strcmp(SQLSTATE,"82100")!=0)                  /* ORACLE, out of disk space? */
    <call procedure to handle this condition>        /* now this works */
  else if (strcmp(SQLSTATE,"00000")!=0) {            /* another error? */
    goto handle_error;                                /* yes, handle it */
  }
  exec sql whenever sqlerror goto handle_error;      /* again in force below */

```

在DB2 UDB中，假设程序的`exec sql include sqlca`语句生效，那么SQLSTATE的值可从`sqlca.sqlstate`的SQLCA处得到。对此我们并不需要一个声明部分，但是在比较时必须小心，因为SQLCA中的字符数组只有5个字符的位置。我们需要使用`strncpy`而不是`strcmp`使字符串在比较了5个字符之后停止：

```
if (strncpy(sqlca.sqlstate,"82100",5)==0)          /* DB2 UDB, out of disk space? */
  <call procedure to handle this condition>        /* now this works */
else if (strncpy(sqlca.sqlstate,"00000",5)!=0) {    /* another error? */
  goto handle_error;                                /* yes, handle it */
}
```

Whenever语句的优点

Whenever语句的主要价值在于一个条件陷阱能够大大减少处理错误返回的代码行数，另外一个优点在于Whenever 的语法可以在各个不同的数据库系统之间进行最大限度地移植。例如，通常Fetch循环和Update语句所作操作的结果是不确定的，因此可以利用检测条件NOT FOUND，但是这一条件在不同的数据库产品中有不同的`sqlca.sqlcode`值（由于一些历史原因，这一值通常为100）。Whenever语句将总是在适当的情况下被触发。

另一方面，对于特定的数据库产品又可能在所有的数据库调用之后通过对SQLSTATE、SQLCODE或`sqlca.sqlcode`值仔细地插入检测来检查任何标准的Whenever条件。的确，这些检测在识别特殊条件，和指定所应采取的动作时提供了更大的灵活性。

3. 处理错误：从数据库中获取错误信息

为了调试程序，经常需要访问与错误返回值相联系的由系统产生的错误信息。我们将扩充例5.2.2的handle_error代码段说明这一点。抽取错误信息的确切代码顺序依赖于你所使用的数据库系统。在SQL-92中曾试图将这些方法标准化（被复制到SQL-99），然而指定的复杂方法从未被主要的数据库厂商所采用。每一产品都应该存在一个可与下面的ORACLE代码序列比拟的方法，我们已在例子的print_dberror()函数中使用到了这一方法。可参考附录B.2得到ORACLE和DB2 UDB中print_dberror()的代码。

例5.2.4 显示ORACLE中的错误信息。首先我们需要一些声明：

```
#define ERRLEN 512           /* maximum length of an ORACLE error message      */
int errlength = ERRLEN;     /* size of buffer                                     */
int errsize;                /* to contain actual message length                 */
char errbuf[ERRLEN];        /* buffer to receive message                         */
```

ORACLE调用sqlglm抽取错误信息，并且将实际的信息长度返回到errlength。注意到错误信息不是以空字符结尾的。在printf中使用了*特性用第二个参数值来填充格式长度。

```
char errbuf[ERRLEN];          /* buffer to receive message                         */
sqlglm(errbuf, &errlength, &errsize);
printf("%.*s\n", errsize, errbuf); /* print just errlength chars from string */
```

4. 指示器变量

回忆第3章中表的列值可以为空值（除非在Create Table语句中该列被声明为not null），并且空值在列类型所取的正常值的范围以外。例3.7.8中，我们指出语句

```
select * from customers where discnt <= 10 or discnt > 10;
```

不会从表customers中检索列discnt值为空的行。这看起来可能比较奇怪，由于可以这么认为所有的值要么小于、要么等于、要么大于10，但是事实上，这确实是正确处理空值的做法。回忆空值代表的是未知的或未定义的量。如果还没有为一个顾客的discnt赋值，我们又不想把discnt的值认为是<=10或>10的。现在假设我们从表customers中检索行的discnt值到一个声明为int cust_discnt的宿主变量（可能其他列值检索到其他的变量中），以如下嵌入式Select语句：

```
exec sql select discnt, . . . into :cust_discnt, . . .
from customers where cid = :cust_id;
```

想象一下如果在选取的行中的discnt值为空，会发生什么样的情况。多数现代的数据库产品不会将空值读取到变量cust_discnt中，但是如果允许读取空值，那么C变量cust_discnt将得到某种位模式。现在考虑如下检测：

```
if (cust_discnt <= 10 || cust_discnt > 10) . . .
```

(||操作符是C语言中的逻辑运算OR)。如果cust_discnt满足两个条件中的任意一个，那么这个检测中的if语句为TRUE。然而如果cust_discnt的值为空，这一if语句就会有问题，因为这时值是不确定的，既不是小于、等于也不是大于10。认识到这一问题后，现在多数的数据库(ORACLE, DB2 UDB, INFORMIX给编译命令提供了-icheck选项)系统遵循SQL-99的标准，当空值读入普通的宿主变量时产生错误(例如，:cust_discnt，它没有指示器变量，将在后面加以说明)。

我们总想在C中尽可能地模仿SQL的逻辑，但是这里出现了一个问题。为了尽可能处理好空列值，我们需要能够辨别返回的变量值是否为空，我们可以写如下语句：

```
if (not-null(cust_discnt) && (cust_discnt <= 10) . . . /* INVALID **/
```

(&&运算是C语言中的逻辑运算AND)。然而，实际上并不存在名为not-null()的函数。我们的意图很明确，就是希望有一种检测，在cust_discnt中检索到空值时，检测不应该返回TRUE。不幸的是，变量cust_discnt没有有效的信息来告诉我们它得到的值是否为空，所有有效信息都被用来表示从属性discnt的非空值中读取的整数值。因而我们需要其他方法记

求`cust_discnt`得到了一个空值，在SQL中使用的标准方法是随着变量`cust_discnt`声明一个指示器变量。我们将这一指示器变量命名为`cd_ind`。这里我们用了如下的声明：

```
exec sql begin declare section;
  float cust_discnt;
  short int cd_ind;
  .
  .
  .
exec sql end declare section;
```

现在在Select语句中，我们将`cust_discnt`的引用替换成`cust_discnt:cd_ind`对：

```
exec sql select discnt into :cust_discnt:cd_ind
  from customers where cid = :cust_id;
```

基本SQL允许在以冒号为前缀的名字前加一个可选的关键字`indicator`来强调发生了什么，所以另一种可选的格式如下：

```
exec sql select discnt into :cust_discnt indicator:cd_ind
  from customers where cid = :cust_id;
```

如果不使用关键字`indicator`，ORACLE要求这两个宿主变量写在一起，中间没有空隔（在DB2 UDB中没有如此要求），写法如下：

```
exec sql select discnt into :cust_discnt:cd_ind      /* ORACLE form      */
  from customers where cid = :cust_id;
```

很明显多数可移植的语法使用了关键字`indicator`，在下面的例子中我们也这样做。在检索之后，如果变量`cd_ind`的值为-1，意味着变量`cust_discnt`得到了一个空值，若是0则意味着`cust_discnt`中的值为一个正常的整数值。现在，我们在前面提到的逻辑测试，即对`cust_discnt`的非空值测试，就可以通过对这一指示器变量值的测试来进行。我们可以写成：

```
if ((cd_ind >= 0) && cust_discnt <= 10) . . .
```

仅当从属性`discnt`中抽取到变量对`:cust_discnt indicator:cd_ind`中的值为非空值时，对`c_ind`的测试结果值才是TRUE。

不管是读取数据库中的数据还是更新数据库，我们都可以用指示器变量的-1值代表空值。例如，要将表`customers`的某一行的`discnt`值设置为空值，我们可以这样写：

```
cd_ind = -1;
exec sql update customers
  set discnt = :cust_discnt indicator:cd_ind where cid = :cust_id;
```

指示器变量还有另外一种标准用法。如果与之相关的宿主变量存储的是一个字符串，并且当从数据库中检索这一字符串时为了使它能在宿主变量中放下截取了字符串的长度，那么数据库系统会将指示器变量设置为一个正数值(`ind > 0`)。比较典型的是，在截取之后这一正数指示器变量的值是这一被截取的数据库字符串的长度。因而，在这种标准用法中，指示器变量的可能值如下：

- =0 数据库值(非空值)被赋值给宿主变量。
- >0 将一个被截取的数据库字符串赋值给一个宿主变量。

= -1 数据库值为空值，宿主变量值是一个没有意义的值。

显然可以进一步扩展指示器变量的值。如，DB2 UDB的产品中，如果指示器变量的值为-2，意味着是由于一个错误而不是数据库中存储的值而使检索到的值为空值（然而，必须执行一些特别的步骤来达到这一对指示器变量的扩展）。

5.3 一些通用的嵌入式SQL语句

这一节我们将给出各种数据操作的嵌入式SQL语句的一般格式，我们从一个对嵌入式Select语句的完整介绍开始。

1. Select语句

回忆一下可以不使用游标在程序中执行Select语句，但是这只能在最多检索出一行记录的情况下（允许0或1行）。如果这一Select语句抽取到了多于一行的记录，系统将返回一个运行期错误。嵌入式SQL的Select语句语法在图5-3中给出。可以参见第3章得到对于语法中元素的介绍，包括对有效的查找条件(search_condition)的定义。注意到图5-3中的宿主变量(host_variable)可以包括指示器变量作为它的一部分。

```
EXEC SQL SELECT [ALL|DISTINCT] expression [, expression...]
    INTO host-variable [, host-variable...]
    FROM tableref [corr_name] [, tableref [corr_name]...]
    [WHERE search_condition]
```

图5-3 基本的嵌入式SQL的Select语句格式（选取单行记录）

由于使用嵌入式Select语句只能抽取到一行记录，交互式的SQL GROUP BY、HAVING、UNION和ORDER BY子句并不包括在这一般格式中。在INTO子句中命名的变量必须与所检索的表达式是一对一的对应关系；由于自动转换并不总是做所有你认为是合理的事情，所以它们必须有正确的类型。在ORACLE中，当数据库中的字符串值'1234'被检索到类型为整型或短整型的C变量时，它会被数据库系统转换成数字格式。同样，在ORACLE的交互式SQL或嵌入式SQL中是可以有qty = '1000'这样的谓词的，（这里qty是一个正数类型的列，而'1000'是一个字符串常量），这是因为ORACLE会自动将字符串转换成整数。然而所有的数据库产品并没有对这种字符串和数字类型之间的转换做标准化。如果要依赖这些转换，你必须熟悉你所使用的数据库系统所支持的操作。当图5-3中Select语句检索到0条记录时，SQLSTATE和SQLCODE的值将被设置成一个合适的警告值，并且将触发一个Whenever Not Found动作。

在Select语句和下面将介绍的其他语句中，在EXEC SQL DECLARE...部分所声明的宿主变量可以使用到INTO子句中来获取检索到的目标列表变量(target list variable)。图5-4给出了一部分在Create Table命令中定义的列数据类型和C语言中变量的数据类型之间的对应关系。有关SQL数据类型的更大的一张表在附录A.3中可以看到。注意，float类型在C和SQL中是不同的；SQL中的float对应于C中的double类型。同样，虽然在ORACLE中，SQL的int与C中的int类型是兼容的，然而在DB2 UDB中，需要使用C中的long int来匹配SQL中的int类型，从而确保它足够大。

在EXEC SQL DECLARE部分所声明的宿主变量也可以在嵌入式SQL中的任何search_condition语法中使用。然而，在search_condition中的宿主变量只能用来表

示表达式中的数字或字符串常量。特别是，宿主变量不能容纳字符串来表示更复杂的语法元素，例如列名、表名或表达式中的逻辑条件。在设置了宿主变量之后，编译程序还要关注这些变量。使用宿主变量字符串来表示这些元素甚至整条语句的功能将在5.6节“动态SQL”中介绍。

Basic SQL 类型	ORACLE 类型	DB2 UDB 类型	C 数据类型
char(n)	char(n)	char(n)	char arr[n+1]
varchar(n)	varchar(n)	varchar(n)	char array[n+1]
smallint	smallint	smallint	short int
integer, int	integer, int, number(10)	integer, int	int (DB2 UDB) 要求 long int
real	real	real	float
double precision, float	double precision, number, float	double precision, double, float	double

图5-4 类型的对应关系：基本SQL, ORACLE, DB2 UDB和C

2. Declare Cursor语句

图5-5给出了基本SQL Declare Cursor的核心语法。这一语法的大部分与图3-13中的交互式Select语法相同。子查询格式允许使用子句GROUP BY和HAVING，也可以包含宿主变量表达式。然而，大多数的数据库系统产品不为在目标列表中检索到的表达式提供列别名；其实这一功能并没有多大的价值，因为一般在嵌入式语句中不用显示缺省表，虽然有时需要在ORDER BY子句中提供命名的列。然而，通常在result_column中允许使用整数，用一列在结果表中的位置来指明该列。

```
EXEC SQL DECLARE cursor_name CURSOR FOR
  Subquery
  [ORDER BY result_column [ASC | DESC] [, result_column [ASC | DESC]]...]
  [FOR {READ ONLY | UPDATE [OF columnname [, columnname...]}]];
```

图5-5 嵌入式SQL的Declare Cursor语法

图5-5最后一行的可选格式用于指明游标是READ ONLY还是FOR UPDATE的。如果两者都没有被指定，那么在缺省情况下，如果使用了ORDER BY那么游标是READ ONLY，否则游标为FOR UPDATE。在DB2 UDB(和Core SQL-99)中，至多需要选择最后两个子句中的一条，ORDER BY或者FOR UPDATE。而ORACLE允许同时使用这两个格式。(然而，在ORACLE中，使用子句WITH READONLY在ORDER BY子句前面，而不像FOR READ ONLY子句在ORDER BY子句后面)。如果程序的逻辑想要通过游标更新或删除一条记录，在游标的声明中必须包括FOR UPDATE子句(将在后面解释)。如果在使用FOR UPDATE时没有指定列，所有被选到的列都被认为是可更新的。当允许数据库优化查询时应该使用READ ONLY子句。如果你使用了READ ONLY，逻辑上就认为你将不会通过游标来更新或者删除记录。ORACLE和DB2 UDB允许在ORDER BY子句中使用表达式，而不仅仅是一个列名。

注意到前面定义的游标只能在一个行集合中向前移动。在SQL-99中消除了这一限制，它新增了一个可滚动游标特征，这将在5.7中介绍。然而，ORACLE或DB2 UDB并不支持这一功

能，这意味着要第二次抽取某行记录你不得不先关闭然后再打开游标。同一游标可能在一个程序中被连续的打开和关闭多次。通常情况下，游标在被再打开之前必需关闭（ORACLE中对这一规则有例外的处理，将在下文中介绍）。

3. Delete语句

有两种格式的Delete语句。一种是定位删除(Positioned Delete)，它删除了游标所在的当前记录（最近被抽取的记录）；另外一种格式是查找删除(Searched Delete)，它的格式同我们在3.9节中看到的交互式SQL中的格式相同。图5-6给出了这两种删除格式的语法。

```
EXEC SQL DELETE FROM tablename [corr_name]
[WHERE search_condition | WHERE CURRENT OF cursor_name];
```

图5-6 嵌入式的基本SQL Delete语法

定位删除使用了一种特殊的语法CURRENT OF cursor_name。两个WHERE只能用一个；如果两个都没有用到，那就删除表的所有行。corr_name用在search_condition中，并且在某些产品中如果它与定位删除一起使用，就可能引起运行期错误。在使用SQLCA的系统中，若选择了查找删除，那么长整形变量sqlca.sqlerrd[2]就包含了删除操作所影响到的行数。如果没有行被影响到，那么Whenever语句中的条件NOT FOUND就成立。

在定位删除中，当删除操作被执行后，游标指向一个新的位置，如果删除后还有记录的话，它就在紧跟着被删除行后面的那行之前。这同刚刚打开一个游标时的情况一样，游标恰指向Select语句所得到的第一行之前的位置；选择这种做法是为了能够更好地处理循环（就像我们在例5.3.1中看到的）。如果执行删除操作时，游标没有指向某一行（也就是说，如果游标恰指向某一条记录之前的位置，或者Fetch已经返回了NOT FOUND，并且游标指向所有行之后的某个位置），系统将返回一个运行期错误。要使定位删除正常工作，在删除中约定的游标必须已经打开并且指向一个实际的行，删除中的FROM子句必须同选择游标时的FROM子句中所指的是同一张表。另外，这时使用的游标必须是一个可更新的游标，即它不可以被声明为READ ONLY。

例5.3.1 从customers表中删除所有住在Duluth并且没有订单的（在表orders中没有出现）顾客记录。我们可以使用查找删除非常简单地完成这一操作，使用如下语句：

```
exec sql delete from customers c where c.city = 'Duluth' and
    not exists (select * from orders o where o.cid = c.cid);
```

我们也可以使用游标来完成这一删除：

```
exec sql declare delcust cursor for
select cid from customers c where c.city = 'Duluth'
    and not exists (select * from orders o where o.cid = c.cid)
    for update of cid; /* must declare cursor for update */
whenever not found goto skip; /* Label "skip" lies later in code */
exec sql open delcust;
while (TRUE) { /* TRUE has value 1: loop forever */
    exec sql fetch delcust into :cust_id; /* if not found, goto skip: out of loop */
    exec sql delete from customers
        where current of delcust; /* delete row under cursor */
}
...

```

注意，在while(TRUE)循环中的Delete语句之后，游标指向随后一行之前的位置。不管删除操作是否被执行了，都需要循环开始处的Fetch语句将游标通过选出的行的每一行向前推进。在由于某种复杂的逻辑条件删除操作可能执行也可能没有执行时，这是有价值的缺省操作，因为我们总是依赖取操作在循环顶部这一要求。 ■

可以修改例5.3.1中的程序，创建一个游标检索所有下过订单的顾客的city和cid，然后在程序逻辑中完成测试if(strcmp(city, "Duluth") == 0)以判断这行是否应该被删除。然而，由于性能上的原因这种方法可能是错误的。将选择的准则交给SQL去处理总是最好的，这是因为，SQL在执行一个检测时能够不带有额外的开销（从表中检索值然后把它们送到程序中，这通常是很费时间的）。的确，通常我们都倾向于尽可能将所要做的操作留给SQL。在例5.3.1刚开始处的查找删除比通过游标使用定位删除来完成同样的任务更有效。

4. Update语句

与Delete一样，有两种形式的Update语句。一种是查找更新(Searched Update)，类似于交互式中的多行更新语句；另一种是定位更新(Positioned Update)，通过游标来完成更新。查找更新(参见图5-7)实质上等同于在3.9节中介绍过的交互式更新的形式，并且这种形式中没有出现我们目前还未介绍的元素。

```
EXEC SQL UPDATE tablename [corr_name]
  SET columnname = expr [, columnname = expr...]
  [WHERE search_condition];
```

图5-7 嵌入式基本SQL的查找更新语法

在查找更新中，长整形变量sqlca.sqlerrd[2]包含了更新所影响到的行数。如果没有行被影响到，那么Whenever语句中的NOT FOUND条件就成立。

同Delete一样，要使定位更新语句(参见图5-8)正常工作，游标必须打开，在Update中指明的表必须和Declare Cursor语句中指明的相同，并且游标必须指向有效的记录行，而不是指向某些行的前而或后而的位置。另外，该游标必须是可更新的游标，在所有被Update语句所改变的列上声明为FOR UPDATE。如果这些条件中的任何一条不能满足，就会在定位更新语句执行时返回运行期错误。

```
EXEC SQL UPDATE tablename
  SET columnname = expr [, columnname = expr...]
  WHERE CURRENT OF cursor_name;
```

图5-8 嵌入式基本SQL的定位更新语法

5. Insert语句

Insert语句只有一种单一的格式，它同在3.9节中介绍的交互式版本的Insert语句相同。在图5-9中给出了Insert语句的语法。不过，有两种插入形式，一种将指定的值插入某一单个行中，另一种可能将从一条普通的子查询语句中得到的值插入到多行中。在通过子查询插入的格式中，长整形变量sqlca.sqlerrd[2]包含了被影响到的行数。如果没有记录被影响到（例如，子查询返回0行），那么Whenever语句中的NOT FOUND条件就成立。

```
EXEC SQL INSERT INTO tablename [(columnname [, columnname...])]  
    {VALUES (expr [, expr]) | Subquery};
```

图5-9 嵌入式基本SQL的Insert语法

让我们考虑一下为什么没有定位插入，如`INSERT...WHERE CURRENT OF cursor_name;`。正如我们将在下面的章节中看到，新插入的一行通常不能被放在表中指定的位置，而是放在由表的磁盘结构所决定的位置。因而短语`WHERE CURRENT OF cursor_name`就会处理应由数据库系统决定的将新插入的记录存放在哪里的问题，这是不合适的。

6. 游标的打开、抽取、关闭

打开一个先前定义过的游标语句格式如图5-10。当Open语句被执行时，数据库系统计算定义游标`cursor_name`中`WHERE`子句的表达式值，并且识别一个活动的行集合，为后面的Fetch操作使用。

游标定义时所用到的任何宿主变量在Open语句执行时被赋值。如果这些值后来有所改变，这也不会影响活动的行集。对于许多数据库产品，如果游标已经被打开了，就不能再一次打开它。ORACLE对这一规则却有例外的处理，即一个已经被打开的游标可以被再次打开，并且重新定义活动的行集。

从一个打开的游标中抽取活动行集的一行的语句格式如图5-11所示。执行Open Cursor语句将游标`cursor_name`定位到活动行集中第一行之前的位置。随后的每一条Fetch语句重新定位

`cursor_name`到活动行集中的下一行，并且将行（在Declare Cursor语句中命名）中所得到的列的值赋给Fetch语句中所命名的宿主变量。为了使Fetch语句工作，`cursor_name`必须已经被打开，并且宿主变量的个数必须与游标定义中的目标列表的列的个数相同。当Fetch语句执行时遇到了一个空的活动行集或者游标位于最后一个活动行后面时，Whenever语句的NOT FOUND条件成立。

关闭一个游标的语句格式如图5-12。这一语句关闭游标，从而使得活动行集再也不能被访问。关闭一个没有被打开的游标会导致错误。

7. 其他嵌入式SQL操作

在接下来的部分将介绍一些其他的嵌入式SQL语句。我们已经介绍过的语句如下：

这些语句和其他一些语句的语法都可以在附录C中找到。大部分语句的语法都非常简单，然而Create Table语句有许多语法元素现在还未介绍到，需要更为详尽的解释，这些将在第6章中介绍。

```
exec sql create table  
exec sql drop table  
exec sql connect  
exec sql disconnect
```

```
EXEC SQL OPEN cursor_name;
```

图5-10 嵌入式基本SQL的Open Cursor语法

```
EXEC SQL FETCH cursor_name  
    INTO host-variable [, host-variable];
```

图5-11 嵌入式基本SQL的Fetch语法

```
EXEC SQL CLOSE cursor-name;
```

图5-12 嵌入式基本SQL的Close Cursor语法

5.4 事务的编程

在第3章中，我们只处理了即席交互式SQL语句。很自然我们会考虑创建一些程序，依次

使用几条SQL语句来完成单独的一个任务。在这一部分，我们将介绍数据库事务的概念。我们将看到有时需要将几条SQL语句集合成一个不可分割的要么全有要么全无的事务包，并且这一概念使编写程序的方法有很重要的分支。

1. 事务的概念

大多数的数据库系统允许多个用户同时执行和访问同一数据库中的表。你可能看到过许多用户在不同的监视器上与同一台计算机交互（学生在一个与一台分时计算机相连的终端房间，银行职员在他们的监视器上，航班订票代理等）。计算机操作系统的一个重要任务是跟踪这些用户，所有正在运行的不同工作流称为用户进程。数据库系统也使用用户进程的概念，通过允许多个进程同时访问数据。这称为并发访问(concurrent access)，或并发(concurrency)。

事实证明若没有有效地对并发访问进行控制，用户进程就可能看到一组永远不应该同时存在的数据元素的集合。

例5.4.1 数据的不一致性 假设一个存款人在表A中有两个银行账号，在表中用不同的账号ID值aid，分别为A1和A2，来代表不同的行。为简单起见，我们用An.balance来表示A.balance where A.aid = An（必须注意，这不是基本SQL的符号）。这里我们假设开始时A1.balance为\$900.00、A2.balance为\$100.00。现在假设进程P1想要从账号A1中取出\$400.00到A2，来平衡两个账户的存款。为了完成这一任务必须对两行进行更新，首先从A1减去\$400.00，然后将这\$400.00加到A2。因而我们在这一过程中有三个“状态”。

S1 A1.balance == \$900.00, A2.balance == \$100.00

更新之前的值。

S2 A1.balance == \$500.00, A2.balance == \$100.00

从账号A1中减去\$400.00后的值。

S3 A1.balance == \$500.00, A2.balance == \$500.00

加\$400.00到账号A2后的值。

现在让我们看另外一个进程P2，它与进程P1同时运行，对这一存款人的存款额进行检查，如果他的所有存款大于等于\$900.00那么允许存款人取出信用卡。为了完成这一任务，进程P2必须分别读取两行，它们的aid值分别为A1和A2，然后将这两个账号上的存款值相加。当进程P2在状态S2执行时，存款总额为\$600.00，这时对这一存款人信用卡的检测就失败；如果这两个账号上的存款总额为\$1000.00，那么他将通过这一检查。这里我们给出了一个操作的调度，它将导致刚才所述的进程P2造成的存款总额视图：

进程P1	进程P2
<pre>Update A set balance = balance - \$400.00 where A.aid = 'A1'; (现在balance = \$500.00)</pre>	<pre>int bal, sum = 0.00; select A.balance into :bal from A where A.aid = 'A1'; sum = sum + bal; (现在sum = \$500.00) select A.balance into :bal from A where A.aid = 'A2'; sum = sum + bal; (现在sum = \$600.00) (信用卡被拒绝)</pre>
<pre>Update A set balance = balance + \$400.00 where A.aid = 'A2'; (现在balance = \$500.00 转账完成)</pre>	

但是进程P2在状态S2时所看到的存款余额总值\$600.00实际上是一个假象。存款人开始时有余额\$1000.00，并且进程P1并不是想要取走钱，仅仅是转账而已。在状态S2下显示出来的存款额为\$600.00的情况称为“不一致视图”。我们假定有特定的必须遵守的一致性规则（如进程P1不能产生或者消除存款，所以即使执行了进程1，存款人的余额总值不应该改变），然而为了得到这一正确的最后状态，在进程P1的逻辑在执行过程中必须经过不一致的状态（如状态S2）。我们更愿意使用一些方法，阻止其他进程看到这些暂时的不一致状态。■

为了避免访问这些不一致状态和由于并发访问带来的一些其他的困难，数据库系统提供了一个特征，叫做事务。每一个进程可以将一系列的数据库操作结合到一起，这些操作构成了一个状态的一致性改变（如进程P1中的两个更新操作）或者是在一致视图下的一些尝试（如进程P2中的两次读操作）；这种数据库操作包称为事务。注意，许多事务要求更新必须是在数据处于一致性状态的基础上进行；例如，进程P1的两次更新操作，每一个都可以认为是在读操作之后进行了写操作，并且这两次读必须是一致的，否则在最后的余额数量上就会出现错误。在后面我们将进一步讨论这一点。（然而，必须注意到虽然每一次更新都可以被认为是读了之后再写，在DBMS中应用于一行上的Update语句本身是不可分的。任何操作都不可能干预这一更新操作时的读和写。从这种意义上说，对应于一行的Update语句已经是事务上不可分的。）

支持事务的数据库系统给程序员提供一些事务特性的保证以方便他们编写程序。对于目前的讨论，最重要的一个保证是隔离。隔离保证了即使事务T1（可能是进程P1两个更新操作的包）与事务T2（可能是进程P2两个读操作的包）同时执行，对于每个事务而言，都好像是不受对方事务的影响而与其他事务相互隔离地运行在数据库上。当我们说隔离时，指的是看上去要么是T2在T1开始对数据作改动前运行，要么是T2在T1完成了所有操作之后执行；事务T2看到的是例5.4.1的状态S1或状态S3，不会看到S2，因而不可能导致不一致的视图。

2. 如何在程序中指定事务

成功连接到一个数据库之后，在进程中并没有活动的事务。没有活动事务的程序可以通过执行任何对数据库行进行操作的SQL语句（如Select, Open Cursor, Update, Insert或者Delete）来产生一个活动的事务。在运行这些可执行语句的过程中，事务保持活动状态，直到程序通过执行如下两个嵌入式SQL语句之一来终止事务：

- EXEC SQL COMMIT [WORK]；这一语句导致事务成功终止；所有在事务中被更新的行将永久地保留在数据库中，并且这些行对其他并发用户来说又成为可见的。事务中读到的所有行可重新被其他并发用户更新。
- EXEC SQL ROLLBACK [WORK]；这一语句导致事务失败终止；所有在事务中对行所作的更新被撤销，恢复这些行原来的值，并且这些行对其他并发用户来说又成为可见的。事务中读到的所有行可重新被其他并发用户更新。

虽然关键字WORK从SQL-92开始是可选的，但在SQL-89中它是必需的，因此为了对较老的数据库系统的可移植性，最好使用这一关键字。在事务以Commit或Rollback结束之后，下一条操作在数据库行上的SQL语句开始一个新的事务，这一事务将保持活动状态直到下一条Commit或Rollback语句。

如果在程序结束前对于一个事务既没有执行Commit也没有执行Rollback语句，那么将执行一个缺省的动作，这一动作（Commit或Rollback）与产品有关。在这一章开头的例5.1.1和5.1.2中，我们看到了这两个例子中使用的Commit语句将在事务中读到的行所加的锁释放掉。

这些锁用来保证事务的隔离，我们一会儿就讨论这一点。一个事务通常由一个在两条Commit语句之间的数据库操作集（Select语句或Update语句，或两者都有）所构成，Rollback语句较为少见。应用程序的一个典型做法是，每个并发用户进程在大量语句之间循环，交替地要求用户输入，然后按照用户的意图执行一组数据库操作。我们将看到，在跨越用户的交互行为时仍保持事务的活动性并不是一个好主意，所以通常在要求用户输入之前执行Commit语句。然而，有可能每一次用户交互有多于一个的事务，因此将事务从一个用户输入持续到下一个用户输入是不安全的做法，尽管这是最典型的做法。

很明显向系统给出事务的结束是程序员的责任——系统自己不能够猜出来。一个涉及到数据库更新（包括Insert和Delete）的事务是一个更新的集合，这些更新作为一个整体，要么都成功，要么都失败。例5.4.1的情况可以扩展为在许多存款账号之间转账，最后账平了，余额既不能多也不能少。但是系统并不知道什么时候达到了这种平衡（它并不遵循数学规则），除非程序员在提交语句中说明了这一点。在Commit之后，所有在事务中对行所作的更新将永久保存下来，并且这些行对其他并发用户成为可见的。在Rollback之后，所有在事务中对行所作的更新被撤销，这些行原来的值成为可见的。必须保证只涉及到数据库读操作的事务T2不能看到在事务T1的一系列更新操作过程中暂时的不一致的数据库状态——这一问题在例5.4.1中可能发生，因为事务T2（代表进程P2）在事务T1（代表进程P1）更新之前读了一些账号的值，而在事务T1更新之后又读了另外一些账号的值。一个只读的事务T2必须冻结它所接触的所有数据，这构成了对数据库进行更新的事务的一个约束。因此，当我们完成任务时需要结束这一只读事务，同样这里系统还是不能猜出来什么时候应该终止这一事务。程序员需要使用Rollback或Commit指明一个事务的结束。在只读事务的情况下，Rollback和Commit的效果是没有分别的，因为没有对数据库做更新。

对于一个程序员要书写更新事务Rollback语句是极其方便的，考虑上一段中在许多存款账户之间转账的例子。假设从前面N - 1个账号中提款，然后这些提款的总额被加到最后的账号中。完成这一功能最简单的编程方法是读取每一条新的账号行，检查这一余额是否能够满足所要提取的钱款数目，假设答案是肯定的，那么从这行的余额中减去这一提款数目，然后继续处理下一行，得到了所有前面提款数目的总和之后更新最后一行，然后提交。其他的事务不能够看到这些中间的更新过程，直到这一更新事务提交了，所以不存在当我们还未确定事务是否成功时其他用户看到所作更新的危险性。但是现在假设在这一系列更新中碰到某一账号的余额没有足够的资金，那么我们就不能完成这一事务。程序只要简单的执行Rollback语句就可以了，而不是往回读已经被更新过的每行，恢复原来的值（在这样做的过程中很有可能发生错误）。回退（Rollback）的一个普遍应用的同义词是事务的异常中止（abort）。执行了Rollback之后，到目前为止所作的所有更新都被系统自动撤回了，就好像事务中的更新从来就没有发生过一样。

数据库系统能够回退失败事务中所有更新操作的功能显示了系统所提供的另一个保证（就像隔离），这一保证被称为原子性。原子性给程序员提供了这样的保证，在一个事务中所执行的行的更新操作集是一个“原子”，即它是不可分割的；要么所有的更新操作都发生，要么都不发生。为了支持原子性，必须知道当事务异常中止时应该撤回数据库中所更新的行，数据库系统应该有某种方法记录这些行的更新，并且把这些记录存起来以备后用。这种“要么全有要么全无”的更新保证甚至在系统崩溃时仍然成立，系统崩溃时内存丢失并且不可能记住程序在做什么、已经做了什么更新或者为什么做了这些更新。

成功提交的事务对数据库所作修改在因系统崩溃时仍能恢复原状的性质叫做持久性，这是数据库系统提供给程序员的第三个保证，另外的两个保证是前面所述的隔离性和原子性。为了支持持久性，数据库系统必须作额外的记录来提醒它自己的意图并且保证它们被存放到磁盘上，以使得它能从内存丢失中恢复过来。我们注意到关于事务的一些保证有些隐含的东西，但是在这里不讨论许多较为复杂的内容，我们在后面章节中介绍了更新事务之后再来讨论这些问题。在本节中，我们讨论更新时只停留在能把读操作时的一致性视图讨论清楚的深度上；在这一节中的剩余部分我们将着重于只读事务和隔离保证。

3. 事务的例子

图5-13举了一个银行出纳员要求在账号之间转账的银行事务程序的例子。回忆一下一个事务是以第一条对数据库行进行操作的SQL语句开始的，所以前面的声明部分、Connect语句和在图5-13中的程序中包含提示的循环都不是任何事务的一部分。对用户的提示要求用户输入指定账号，分别命名为from和to，它们指明了转账所涉及的两个账号以及转账的金额。sscanf()函数用来将金额的文本格式（存放在数组dollarstr[]中）转换成金额的double(float)格式（存放在变量dollars中）。要理解这一部分代码，你应该阅读附录B.1。其中，设置事务语句SET TRANSACTION ISOLATION LEVEL SERIALIZABLE是一个样板语句，你现在没有必要理解，我们将在10.5节中进一步讨论事务时解释它们。

操作在数据库行上的事务语句是两条Update语句。如果执行这些语句没有返回错误，事务就被提交，程序结束。如果有错误，控制会跳转到do_rollback，并且事务回退。在这种情况下，如果第一条更新已经发生了，数据库系统将撤回所有的改动，将数据库恢复到这一事务从未运行时的状态。

一般而言，这种程序会在一个函数调用中执行，在程序前面的循环中让出纳员选择所要做的操作，如从一个账号到另一账号转账金额、从一个账号提款、存钱到一个账号、查询账号余额；等等。在图5-13中，程序执行事务的部分被放在一个循环中进行重复，并且强调了在每个转账过程的最后使用Commit或Rollback语句的重要性；否则当另外一个不同客户的事务开始时，旧的事务将仍然有效，并且最后会导致所有的账号行被其他的出纳员锁住而不能使用。

注意到可以同时在一个程序中运行不同的进程（代表不同的出纳员），并且一个进程下的事务可能是将存款从账号A1转移到账号A2，然而另一事务将存款从账号A2转移到账号A1。令人惊讶的是，如果在这一程序中的事务中都没有包含一对Update语句，那么这些事务并不会相互影响：即使Update语句是交织的，同样的转账依然可以成功完成。这是因为对单个行单独执行的各个的Update语句是不可分的，不管目前的账号余额为多少，都完成要求的任务，加或减去一个美元数额。但是对于包含两条Update语句的事务，就不能成功完成如图5-13的转账，而需要一个事务保证与它并发的事务能从两个账号中返回正确的余额总值。

注意到在图5-13的程序中，用户交互（prompt()循环）是完全在任一事务的作用域之外执行的。这是因为这样做可以使得事务能够较快地被执行，以使为保证隔离而被这一事务占用的锁或其他资源不会对其他事务有长期的负面影响。这将在后面进一步讨论。

4. 事务隔离的保证和加锁

隔离属性保证了当事务T1和T2同时执行时，对于每个事务而言它对数据库的操作不受另外一个事务的影响。更准确地说，T2所作的所有数据库操作看上去要么都是在T1开始之前要么就是在T1完成之后。另外一种说法是任一并发执行的事务集合，它的执行结果就好像这些事务是按照某种串行的顺序来执行的，即对于这一集合中的任意两个事务，它们中的一个必须在

另一个事务开始之前完成它所有的操作。事务好像是按照某种串行的顺序执行的这一事务特性，也称为可串行性，我们将在下一章深入讨论这一性质。在这里可串行性与隔离是等价的。

```
#include <stdio.h>
#include "prompt.h"

int main()      /* note no #include SQLCA, since SQL-99 deprecates this */
{
    exec sql begin declare section;
    char acctfrom[11], acctto[11];
    double dollars;
    exec sql end declare section;
    char dollarstr[20];

    exec sql connect to default;           /* (not part of transaction) */
    exec sql set transaction isolation level serializable;
    while (1) {               /* loop forever: simplified teller-program loop */
        while ((prompt("Enter from, to accounts and dollars for transfer:\n",
            3, acctfrom, 10, acctto, 10, dollarstr, 10)) < 0) ||
            (sscanf(dollarstr,"%lf",&dollars) != 1)) { /* convert to double */
            printf("Invalid input. Input example: 345633 445623 100.45\n");
        } /* Above, print out error msg until input is acceptable */
        exec sql whenever sqlerror goto do_rollback;

        /* transaction starts here-- */
        exec sql update accounts set balance = balance - :dollars
            where acct = :acctfrom;
        exec sql update accounts set balance = balance + :dollars
            where acct = :acctto;
        exec sql commit work;           /* transaction ends here... */
        printf("Transfer complete\n");
        continue;
    do_rollback:
        exec sql rollback work;          /* ... or here */
        printf("Transfer failed\n");
    }
    exec sql disconnect current;
    return 0;
}
```

图5-13 在账号之间转账的简单程序(SQL-99)

如果我们将T1看做是一个同一个更新事务集并发执行的只读事务，它最后一段的定义意味着在这些并发执行的更新事务的某些(可能为0)已经完成之后事务T1只看到数据库的一致性状态。数据库系统实现隔离最普遍使用的方法是对数据库行加锁。简化了的加锁规则在图5-14中给出，这些规则有助于支持事务的隔离。

更加普遍的加锁方法将会在第10章中介绍，这种方法对读访问的锁(R锁)和更新(或者说是写)访问锁(W锁)区别对待；但是现在我们只讲在所有情况下使用的简化了的排他锁的方法。

- 1) 当事务访问R行时，它必须以排它模式先对该行加锁。
- 2) 直到事务结束(提交或撤回)才将事务中得到的所有的锁释放。
- 3) 如果T1已加锁了一行R 并且另一个事务T2 企图访问该行，那么在访问它并被拒绝之前T2也企图锁定R，因此T2 的锁与T1 保持的锁冲突。
- 4) 对于系统而言解决这一冲突通常采用的方法是等到T1 提交或异常中止并释放了对R的锁之后，T2 的加锁要求才能被满足，T2 继续执行。

图5-14 简化的保证隔离的数据库行加锁规则

例5.4.2 加锁和不一致视图 考虑图5-15中事件的顺序，这是例5.4.1的重述，开始时账号A1有balance \$900.00，A2有\$100.00：

事务T1更新事务	事务T2只读事务
<pre>Update A set balance = balance - \$400.00 where A.aid = 'A1'; (现在 balance = \$500.00)</pre>	<pre>int bal, sum = 0.00; select A.balance into :bal from A where A.aid = 'A1'; sum = sum + bal; (现在 sum = \$500.00) select A.balance into :bal from A where A = 'A2'; sum = sum + bal; (现在 sum = \$600.00)</pre>
<pre>Update A set balance = balance + \$400.00 where A.aid = 'A2'; (现在 balance = \$500.00)</pre>	<pre>commit work;</pre>

图5-15 没有加锁的事务：不一致数据视图

例5.4.1的事件顺序允许进程P2得到有关账号A1和A2的balance的不一致视图，这一事件序列与例5.4.1相对应。在这一调度中，事务T1代表进程P1，事务T2代表进程P2；图中行的次序就是访问的次序，所以这两个事务的访问是交织在一起的。开始时，账号A1 (A.aid = 'A1') 的balance为\$900.00，A2的balance为\$100.00。

在图5-15中，事务T2看到了A1和A2两行的不一致视图，这一视图与例5.4.1中的状态S2有关联。但是现在我们考虑如何使用前面所述的方法对这些行加锁。在图5-16中，可以看到T1在更新行A1之前必须先对其加锁。此后，事务T2在通过Select语句读取行A1时想要对它加锁，这一加锁将不被准许，它必须等到事务T1完成了它的操作并且提交了之后才能对A1加锁。这将使事务T2的所有操作都被延迟，直到T1的两个更新操作都完成，这时T2能够执行它的读操作并看到一致性状态S3。图5-16给出了这些事件的顺序。

自然，这仅仅是一个例子，并不能证明在所有情况下都可以通过行加锁来确保隔离性。实际上要获得这一保证有一些其他的细节要谈论（特别是当使用Open Cursor语句得到一个活动的行集合时），但是行加锁是大多数数据库系统用以保证隔离的基本方法，以后我们假设这

些加锁的动作在所有的数据访问中都发生。

事务T1(更新事务)	事务T2(只读事务)
	int bal, sum = 0.00;
Update A set balance = balance + \$400.00 where A.aid = 'A1'; (这一行现在被锁定balance = \$500.00)	
	select A.balance into :bal from A where A.aid = 'A1'; (相同的行被下锁定，必须等待)
Update A set balance = balance + \$400.00 where A.aid = 'A2'; (This row now locked) (现在balance=\$500.00；转账完成)	
commit work; (释放锁)	(前一个Select现在可以获得需要的锁) select A.balance into :bal from A where A = 'A1'; sum = sum + bal; (现在 sum=\$500.00)
	select A.balance into :bal from A where A = 'A2'; sum = sum + bal; (现在 sum = \$1000.00)
	commit work;

图5-16 对事务加锁改正了图5-15中的不一致视图

5. 事务中需特别考虑的事项

在保证隔离时存在一个问题：当两个事务的访问导致了死锁怎么办？我们也将再下文介绍为什么通常情况下有必要在事务中避免用户交互。

(1) 死锁

伴随用数据库行加锁来保证隔离所带来的一个潜在的问题是有可能会出现死锁。当两个事务（或更多）等待其他事务释放资源并且如果没有一个事务放弃（rollback或abort）资源使得没有事务可以进行，那么将导致死锁。

例5.4.3 事务死锁 让我们重述一下例5.4.2，但是这里事务T2的访问数据顺序有所不同，见图5-17的调度表。

在这个例子中，事务T1在更新行A1之前先锁住了它。然后事务T2在用Select语句读取行A2之前成功对A2加锁。但是现在事务T2想要对行A1加锁，而A1已被T1锁住了，所以T2加锁失败。系统让T2等待直到T1提交并释放它所拥有的锁。接下来，事务T1想要对记录A2加锁，而A2已被T2锁住了，所以T1加锁失败。系统让T1等待直到T2提交并释放它所拥有的锁，但在这种情况下系统需要能够识别出这一死锁状态。T1不能继续执行直到T2提交并释放了锁，但是由于T2又在等待T1完成并释放锁，所以T1将永远不能继续下去。两个事务直到另外一个释放了它所拥有的锁才能继续执行。在这种情况下，如果系统不想让这些事务永远挂在那里，它就要让其中的一个能够继续下去。做法是从这两个事务中选择一个回退。这被称为死锁中止。所有由最后被中止的事务所做的更新被取消，运行这一事务的进程收到一条由最近的数据访问语句所返回的出错信息，进程的逻辑由这一点继续。运行这一被中止事务的进程可以

采取的方法是重新运行事务（即试图重新执行这一事务）。注意到对被中止的事务不仅所有的行更新被取消，在其中读取的所有行值也被认为是不可信的（它们当中的一些可能在锁释放后被另外一个更新事务改变），所以唯一安全的方法是重新执行这一事务。

事务T1	事务T2
	int bal, sum = 0.00;
Update A set balance = balance - \$400.00 where A = 'A1'; (得到锁)	
	select A.balance into :bal from A where A = 'A2'; sum = sum + bal; (得到锁)
	select A.balance into :bal from A where A = 'A1'; (同T1冲突，等待)
	...
Update A set balance = balance + \$400.00 where A = 'A2'; (同T2冲突，等待)	...
...	...
...	...

图5-17 事务的死锁

对于嵌入式SQL的应用程序编程，死锁中止条件隐藏着重要的含义。如果系统异常中止完成了一部分逻辑的事务来解除死锁，程序就需要检测出这一状态并且尝试重新执行这一事务，即重试。重试的方法在例5.4.3中提到，但实际上通常应由程序逻辑来实现，并且可能会有许多复杂性。尝试重试是正确的做法，因为前面事务执行中产生的失败并不是逻辑上的问题，而仅仅是在时间调度上的问题，所以我们有理由希望再执行一遍事务将成功完成。

在开始了一个事务之后，程序必须从访问数据的SQL语句返回的sqlca.sqlcode中寻找“deadlock abort”的出错信息。ORACLE和DB2 UDB产品对于这一状态的sqlca.sqlcode错误编码显示在图5-18中。我们也给出了ORACLE在ANSI模式下支持的SQLSTATE的错误返回信息。

变量		值	描述
Basic SQL	SQLSTATE	'40001'	Serialization Failure
ORACLE	sqlca.sqlcode	-60	
DB2 UDB	sqlca.sqlcode	-911	

图5-18 死锁中止错误(ANSI模式下的ORACLE和DB2 UDB也支持SQLSTATE)

死锁中止的错误返回几乎会出现在一条对数据库中数据进行读或更新的语句的任何时候。对于一个死锁中止，程序应该反复尝试多次运行事务（即重试事务）如果死锁继续出现，那么可能潜藏着更严重的问题，程序应当建议用户请教系统管理员（终端用户通常缺少对事务细节的理解）。注意到如果你在检测出死锁中止的语句之前建立了一个条件句处理程序，如 whenever sqlerror stop，那么这一死锁中止的错误将不会被检测到。就像在4.2节中

看到的那样，`Whenever`语句在对一错误显式检测之前已对错误做出了反应（不幸的是，没有与死锁中止相关的`Whenever`条件）。因此捕捉死锁中止的错误之前在适当的位置为错误返回设置缺省的`CONTINUE`动作是非常重要的。

例5.4.4 死锁中止检测 在接下来的程序段中，我们写了一个使用查找更新语句对多行进行更新的事务，并且当死锁发生时重试事务。用的是ORACLE中死锁中止错误信息返回的符号常量。这一编码的另一个可选的更具潜在移植性的版本将使用SQLSTATE值。

```

...
#define DEADABORT -60
#define TRUE 1
exec sql whenever sqlerror continue;
int count = 0;
while (TRUE) {                                /* loop over deadlock-abort retries */
    exec sql update customers
    set discnt = 1.1*discont where city = 'New York';
    if (sqlca.sqlcode == DEADABORT) {
        count++;                                /* count up deadlock aborts */
        if (count < 4) {                         /* retry up to four times */
            exec sql rollback work;
            /* here, call operating system to wait for a second */
        } else break;                            /* too many retries */
    } else if (sqlca.sqlcode < 0)
        break;                                  /* non-deadlock error */
    }
    if (sqlca.sqlcode < 0) {                    /* over-retried deadlock or other error */
        print_dberror();
        /* print error message */
        exec sql rollback work;
        return -1;                             /* return error */
    } else return 0;                            /* return success */
}

```

注意到在事务运行中可能存在其他错误，例如在申请锁时超时，但在死锁中止的情况下重试事务看来是最合理的做法。 ■

重试事务还有另一个需要考虑的重要事项。有可能在事务死锁中止之前的执行过程中程序重新设置了内存中的一些局部程序变量。这些局部变量可能代表被订购产品的标志，例如，当数据库中相应的产品被取出时，标志被清除；还有一种可能是一个局部变量代表了某个用户所订购的总金额。在死锁中止发生之后，所有的数据库信息被回退到程序逻辑第一次看到这一信息时的那个位置。然而，内存中的局部程序变量是不受数据库控制的，所以程序员应当注意使这些变量也要返回到它原来的初始值。如果原来的状态难以恢复，程序员应当在开始事务之前复制所有的在事务执行过程中可能改变的状态变量，当死锁中止发生时再将这些原始值复制回去。

(2) 事务执行期间不进行用户交互

数据库行加锁最重要的特征之一是只要事务保持活动状态，事务就一直占有它所获得的锁，也就是说，直到程序执行了Commit或Rollback语句。这限制了事务合适的持续时间。一个事务锁住了一行之后，这一事务执行期间的每一秒都阻止了想要访问这一行的其他用户。最重要的一条准则是当事务在执行时不应发起任何用户交互（当重要的数据仍不能被访问时，用户可能想与人谈话，或者溜跶出去喝杯咖啡）。然而，在许多情况下程序可能要在事务执行当中与用户交互。

例5.4.5 事务中的用户交互 考虑图5-19的程序段，我们从代理商那里接受订单，更新products表中的quantity列，quantity代表了在仓库中可得到的产品数量。这一程序段与代理商交互，检测订单是否被填写，并且它不能防止所有可能的错误。

```
/* Interaction with agent taking product order for pid given by req_pid */
exec sql select price, quantity into :price, :qoh
  from products where pid = :req_pid; /* get info for later use */
while (TRUE) { /* loop until we get a quantord that fits in qoh */
  printf("We have %d units on hand at a price of %d each\n", qoh, price);
  if (prompt("How many units?\n", 1, quantordstr, QUANTORDLEN) < 0 || 
      sscanf(quantordstr, "%d", &quantord) != 1) /* get int from string */
    printf("Please enter a decimal number.\n");
  continue; /* start over pass in loop */
}
if (quantord <= qoh)
  break; /* break out of loop if quantord fits in qoh */
else printf("There are not enough units to fill your order.\n");
} /* end get-quantord loop
exec sql update products /* reflect new order
  set quantity = quantity - :quantord /* now know this fits
  where pid = :req_pid;
/* now insert new order in orders table
...
exec sql commit work; /* transaction complete
...
```

图5-19 拙劣安排用户输入位置的程序段(例5.4.5的图解)

注意到当While循环中的用户交互发生时，通过Select语句已经执行了对products行的读操作：已经开始了一个事务并且products中的一行已被加锁。当代理商在与客户就所需订单量进行协商的过程中，其他代理商不能订购这一产品。在行被锁住时执行用户交互不是数据库系统的运行期错误——数据库系统并不知道上述例子中的用户交互——但是在实现时，如果程序中锁住的行有可能被另一用户访问，还是应该避免用户交互。 ■

当事务不在用户交互之间跨越时会产生一个令人吃惊的问题。

例5.4.6 没有封闭事务的用户交互 考虑图5-20的程序段，它是图5-19的重写，这里避免了事务执行中的用户交互。

它看上去像是对例5.4.5逻辑的一个直接的修改，避免了在用户请求时保留锁。然而，如果我们仔细地看一下就会注意到一些奇怪的事情发生。检索选中产品的price和quantity之后，我们在向代理商想要多少数量的订单之前提交了这一事务。当代理商响应时我们已不再对这一行加锁，但是有可能在第一次读取它到根据代理商的回答更新这行的时间内有另外的事务改变了这行中的quantity。因此我们显示给代理商的qoh值可能已经无效了。因此必须在Update语句本身中检测所订购的产品数量(quantord)，确定仅当减去quantord后不会使quantity小于0时才对products.quantity作更新。如果执行更新后没有行被影响到(NOT FOUND条件满足)，那么由于所要求的数量过多，这一订单被拒绝。令人惊讶的是，即使要求订购的数量比我们之前显示出来的手头上拥有的产品数量少，这一订单也可能被拒

绝。例如，我们可能已经显示出来目前的产品数量有500，然而却拒绝了一个要求订购数量为400的订单，并告诉用户“对不起，目前没有足够的数量满足你的订单。”更糟的是，如果我们再执行一遍While循环，将新的quantity值读到qoh中，显示这一值，告知目前有300个产品，然而若现在客户要求订购数量为300的产品，我们可能会又一次拒绝这一订单！

```

/* Interaction with agent for product req_pid-Version 2 */
while (TRUE) {                                /* loop until update succeeds */
    exec sql select price, quantity into :price, :qoh
        from products where pid = :req_pid;      /* get info for later use */
    /* shouldn't hold lock during user interaction, so need to Commit */
    exec sql commit work;
    printf("We have %d units on hand at a price of %d each\n", qoh, price);
    if (prompt("How many units?\n", 1, quantordstr, QUANTORDLEN) < 0) ||
        (sscanf(quantordstr,"%d",&quantord) != 1)) { /* get int from string */
        printf("Please enter a decimal number\n");
        continue;                                /* start over pass in loop */
    }
    exec sql whenever not found goto unitsgone;
    exec sql update products                  /* new type of update with... */
        set quantity = quantity - :quantord
        where pid = :req_pid
        and quantity - :quantord >= 0;           /* ...new test that quantord fits*/
        /* insert new order in orders table */
    exec sql commit work;                      /* transaction complete */
    break;                                     /* break out of loop */
unitsgone:                                     /* if no row selected, quantord didn't fit */
    printf("There are not enough units to fill your order.\n");
    exec sql rollback work;                   /* unneeded: no lock held */
} /* try-update loop */
...

```

图5-20 程序段（例5.4.6的图解）



这一类不一致性是很少见的，这是在写事务系统时我们通常需要接受的一种情况。你可能在预订航班座位的时候遇到过类似的情况；机票代理开始时说：“有一个靠窗的位子；我会为你预订它”，过一会儿之后又说：“噢，对不起，别人已经预订了！”

(3) 游标和事务

正常情况下用Commit或Rollback语句结束一个事务时，所有打开的游标被关闭。这一缺省的动作反映了编程的一种普通模式，在这一模式下，游标在一个作为原子的事务单元所执行的循环中做一些细节性的工作。然而，有时程序需要在一个循环中提交多次。例如，假设程序循环完成将所有的雇员工资涨4%。如果在提交之前执行整个循环，我们最后会锁住这张表的所有行，使它们都不能够被读取。为了减少锁定的行的数量，我们通常在对一定数量的雇员（假如说是100个）做了更新后就提交。这样我们能够在循环中进行足够多的处理，以使提交的开销不会比逻辑处理的开销多10倍。但是如果我们在每次提交时丢失了游标所在的位置，就要花很大的开销把这一位置重新找到。对于这种情况，Core SQL-99在Declare Cursor语句中指定了一个可选的WITH HOLD子句：

```

EXEC SQL DECLARE cursor name [WITH HOLD] CURSOR FOR
Subquery

```

```
[ORDER BY result_column [ASC | DESC] [, result_column [ASC | DESC]...])
[FOR {READ ONLY | UPDATE [OF columnname [, columnname ...]}]];
```

当这一选项有效时，跨越提交时游标仍保持打开的状态，因而它也记住了在表employees中的位置。当然在搜索的过程中不在目前被锁定范围内的其他记录可以改变，这是件好事！如果我们规定salary列每次只能被一个应用程序更新，并且在这一进程执行过程中不插入新的不应加工资的雇员的记录，满足了这些，更新才有效。我们可以使用一张只有一行的特殊的表规定这样的一条规则，即所有这样的更新和插入必须在进行其他工作之前用更新锁。

WITH HOLD特性在SQL标准中是没有的，但是它出现在Core SQL-99中意味着很快就可以在一些重要的数据库产品中使用，DB2 UDB就从版本5（1997）开始提供这一特性。ORACLE没有这一子句，并且在MODE=ORACLE（缺省）的情况下，当Commit或Rollback时关闭所有CURRENT OF子句中提到的游标；按照SQL标准没有**WITH HOLD**子句时，在MODE=ANSI的情况下，它也将关闭所有属于当前连接的游标。

从对ORACLE行为的这种描述中我们发现可以通过避免使用CURRENT OF动作来防止提交的时候丢失游标位置。在Fetch循环中抽取伪列ROWID来保存游标位置，然后使用它执行查找更新where rowid = :row_id而不是where current of cursor来完成任一所需的更新。通过ROWID访问提供了快速访问。要得到关于ROWID更为详尽的信息请参阅第8章中的8.2节，并且关于访问RID的更多信息可参阅推荐读物[7]《*ORACLE8 Programmer's Guide to the Pro*C/C++ Precompiler*》。

5.5 过程性SQL程序的能力

在这一节中，我们将说明在3.11节中非过程的基本SQL语句不能完成的任务是如何通过过程性的方法完成的。（注意到3.6节中一些高级SQL语法也能够扩展基本SQL的功能来完成这类任务。）

定制的集合函数

例3.10.3中指出了SQL中缺少求中值的函数。我们现在通过程序克服这一限制。

例5.5.1 模拟一个求中值的函数。图5-21的程序反复要求用户输入cid，如果我们可以书写如下的SQL语句，就可以显示出这一中值：

```
select median(dollars) from orders where cid = :cid;
```

图5-21中得到中值的方法是对orders表中选到的行的dollars值的个数进行计数，然后在这个订单行计数值ocount的一半处抽取中值。最后抽取到的行号为(ocount+1)/2，例如，行号1代表有1条或2条记录、行号2代表有3条或4条记录，以此类推。如果行ocount是奇数，将给出确切的中值位置；如果ocount是偶数，检索到的值将是两个处于同样有效的中值位置的行中的一行，如行号2代表4行。注意到有一点很重要，count()函数必须与Open Cursor和Fetch循环在同一个事务中运行，以保证不会出现因为并发事务插入所作的改变导致中值计算无效。对于dollars的空值情况必须认真对待，因为在计算中值时我们不将空值考虑在内。注意到函数count(dollars)不会对空值计数，并且在定义dollars_cursor时的子句where...dollars is not null也将忽略空值。

另外在查询条件：...where cid = :cid中我们看到存放cid返回值的变量名正是它

自身cid。这并没有什么错，因为冒号(:)清晰地将宿主变量和列名区分开了。

```
#include <stdio.h>
#include "prompt.h"
exec sql include sqlca;
char custprompt[] = "Please enter a customer ID: ";

int main( )
{
    exec sql begin declare section;
    char cid[5], user_name[20], user_pwd[10];
    double dollars; int ocount;
    exec sql end declare section;
    exec sql declare dollars cursor cursor for      /* to calculate median */
        select dollars from orders where cid = :cid and dollars is not null
        order by dollars;
    int l;
    exec sql whenever sqlerror goto report_error;
    strcpy(user_name, "poneilsql");
    strcpy(user_pwd, "XXXX");
    exec sql connect :user_name identified by :user_pwd; /* ORACLE Connect */

    while (prompt(custprompt, l, cid, 4) >= 0) {      /* main loop: get cid */
        exec sql select count(dollars) into :ocount /* count orders by cid */
            from orders where cid = :cid;
        if (ocount == 0) {
            printf("No orders retrieved for cid value %s\n", cid);
            continue;                                /* do outer loop again */
        }
        /* open cursor and loop until midpoint of ordered sequence */
        exec sql open dollars_cursor;
        for (l = 0; l < (ocount+1)/2; l++)          /* fetch at least once */
            exec sql fetch dollars_cursor into :dollars;
        exec sql close dollars_cursor;
        exec sql commit work;                      /* release locks */
        printf ("Median dollar amount = %f\n", dollars);
    }
    exec sql commit release;                    /* ORACLE Disconnect */
    return 0;
report_error:
    print_dberror();
    exec sql rollback release;                 /* ORACLE Disconnect in error*/
    return 1;
}
```

图5-21 检索中值的ORACLE程序（例5.5.1的图解）

例5.5.2 传递闭包 给出例3.10.6中的表employees，我们想要提示用户输入eid，然后检索最终带有指定的eid的雇员领导（通过一些中间的管理人员）的所有雇员。对嵌入式SQL来说要完成这一任务的难度是相当惊人的。我们可以建立一个游标列出直接归这指定的eid的雇员领导的所有雇员，然后对于每一位接受直接领导的雇员再建立一个游标作为第二层领导，以此类推。但是我们不知道领导的层次总数，并且由于需要在一开始就声明所有会同时使用的游标，如果每一层都需要一个游标，这将是一个很严重的问题。我们通过执行一个略带技巧的对雇员树进行宽度优先搜索的双重循环来回避这一限制。

注意，所有对数据库数据进行的访问都在函数breadth_srch中进行，这正是Whenever SQLERROR语句放置的位置。如果我们在两个函数中都访问数据库，那么就需要在每个函数中为GOTO放置一个类似“report_error”的标记，否则程序将不能通过编译，因为GOTO转移的目标必须与每条访问数据的SQL语句在同一个函数中。

```
#include <stdio.h>
#include "prompt.h"
exec sql include sqlca;
int breadth_srch(char *start_eid, int cursor_open);
int main()
{
    char eid_prompt[] = "Enter Employee ID to see all lower level reports: ";
    char start_eid[6];
    exec sql connect to testdb;          /* DB2 UDB Connect, simple form */
    while(prompt(eid_prompt, 1, start_eid, 5) >= 0) { /* loop for eids */
        breadth_srch(start_eid, 0);      /* recurse: retrieve subtree */
        exec sql commit work;
    }
    exec sql disconnect current;
    return 0;
} /* end main */ */

exec sql begin declare section;
    static char eid[6], ename[17];
exec sql end declare section;
exec sql declare dirrept cursor for select eid, /* cursor over direct reports */
    ename from employees where mgrid = :eid; /* ... of given employee eid */

int breadth_srch(char *start_eid, int cursor_open)
{
    char save_eid[6];                  /* saved eid for recursion */
    exec sql whenever sqlerror goto report_error;
    if (!cursor_open) {                /* reached new report subtree */
        strcpy(eid, start_eid);        /* eid is at head of subtree */
        exec sql open dirrept;         /* get direct reports of eid */
    }
    exec sql whenever not found goto srch_done; /* return from here when done */
    exec sql fetch dirrept into :eid, :ename; /* next direct report */
    printf ("%s %s\n", eid, ename);       /* print current ename, eid */
    strcpy(save_eid, eid);              /* save this eid */
    if (breadth_srch(eid, 1)<0) return -1; /* get other emps, this level */
    exec sql close dirrept;            /* this level is exhausted */
    return breadth_srch(save_eid, 0);    /* recurse: retrieve subtree */
srch_done: return 0;
report_error: print_dberror();
    exec sql rollback work;
    return -1;
}
```

图5-22 执行传递闭包的DB2 UDB程序

在图5-22的程序中，每一次调用到函数breadth_srch时参数cursor_open==1，这意味着我们

在当前游标所在的雇员领导层次上由左至右的移动，做宽度优先搜索。因此breadth_srch抽取这一层的下一个雇员，显示eid和ename，将eid值保存到一个局部变量中，然后通过嵌套调用自身来得到同一游标中的下一个雇员信息（往右移动）。嵌套调用返回的值暗示所有子树已被搜索完毕；函数breadth_srch重建局部值eid，再一次打开一个游标向下一层搜索，递归地搜索这些eid的子树。当当前子树的所有低层雇员被搜索完之后，嵌套的breadth_srch返回，继续搜索这一层剩余eid的子树，或者当这一层的eid已被搜索完之后，它返回到开始调用这一层breadth_srch的地方。最后我们返回到主函数，提交所有工作，并且断开与数据库的连接。 ■

5.6 动态SQL

回忆在对嵌入式Select语句的语法进行讨论的5.3.1节中，我们提到在搜索条件中，声明部分命名的宿主变量只能被用作常量；该变量不能包含代表更为复杂的表达式的字符串，即这些语句的某些部分需要编译程序进行语法分析。在这一节中，我们将学习一种新的方法，称为动态SQL，它允许在宿主变量中构造一个字符串作为SQL语句。目前为止我们所看到的嵌入式SQL语句都称为静态SQL。动态SQL允许我们构造新的在程序编译时不能被具体预知的SQL语句，并且随着用户需求的改变动态地执行它们。

1. 立即执行

在程序中动态地对新的语句进行语法分析有许多好处；但是在具体介绍这些优点之前，我们需要先讲一个例子。

例5.6.1 立即执行 图5-23的程序中，字符数组sqltext[]存储代表一条SQL语句的字符串，然后对它进行语法分析，并使用一条新的嵌入式SQL语句Execute Immediate执行它。这一例子将sqltext[]的内容设置为一个字符串常量，但是通过菜单交互根据用户的意愿来设置这一字符串是更为普遍的用法。这里给出的例子描绘了一条Delete语句的立即执行，对于其他语句，如Update和Insert，也可以这样做的。然而，语句Select不能用这种语法执行，它需要使用另一种方法，稍后将作解释。

```
#include <stdio.h>
exec sql include sqlca;

exec sql begin declare section;
char user_name[] = "scott"; char user_pwd[] = "tiger";
char sqltext[] = "delete from customers where cid = '\c006\'";
exec sql end declare section;
int main( )
{
    exec sql whenever sqlerror goto report_error;
    exec sql connect :user_name identified by :user_pwd; /* ORACLE Connect */
    exec sql execute immediate :sqltext;           /* execute immediate */
    exec sql commit release;                      /* ORACLE Disconnect */
    return 0;
report_error:
    print_dberror( );
    exec sql rollback release;                  /* ORACLE Disconnect in error */
    return 1;
}
```

图5-23 立即执行 SQL语句的ORACLE程序（例5.6.1的图解） ■

基本SQL的Execute Immediate语句的一般格式非常简单：

```
EXEC SQL EXECUTE IMMEDIATE :host_variable;
```

宿主变量数组的字符串内容必须代表一条有效的SQL语句，目前为止我们遇到的类型有：Create Table, Delete(查找删除或者定位删除), Drop Table, Insert, Update(查找更新或者定位更新)。

考虑一个菜单交互，当行符合某一用户定义的条件集合时，它允许用户删除CAP数据库中customers表的行。这些条件可以通过菜单界面用平常的语言表达，如图5-24所示。

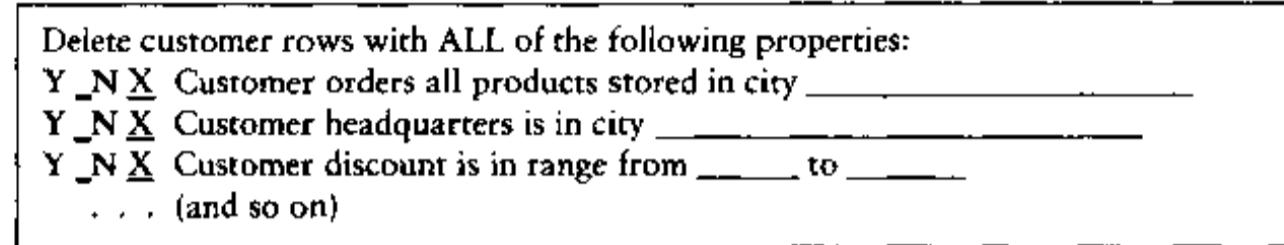


图5-24 从customers表中删除行的示例菜单

菜单用户可以在Y之后填写X（意味着“yes”，覆盖缺省的“no”），然后在右面填写相应的参数。程序应该接受这些选择，构造正确的sqltext[]来完成所需的Delete语句，然后立即执行这一语句。例如，假设我们指定了图5-24中的一行如下：

```
Y X N _ Customer headquarters is in city _Duluth_____
```

这里选择了Y，并且在城市名中填入了“Duluth”。假设程序已将这一城市名填入到数组cname2[]中。程序也必须初始化数组sqltext[]来包括Delete语句的开始部分，当然它必须留有足够的空间：

```
char sqltext[256] = "delete from customers where ";
```

程序现在就可以来填充sqltext[]中语句的剩余部分，使用库函数strcat()将新的字符串拼接到sqltext[]中：

```
strcat(sqltext, "city = '"); /* concatenate string: city = ' */  
strcat(sqltext, cname2); /* concatenate cityname: (Duluth) */  
strcat(sqltext, "'"); /* concatenate quote character: ' */
```

注意到转义字符(\)必须在C语言字符串中的单引号之前使用。以上操作的结果与我们书写如下语句的结果相同。

```
char sqltext[256] = "delete from customers where city = '\'Duluth\'';
```

为了确保数组sqltext[]没有溢出，可以使用strncat()代替strcat()。

显然这一字符串Execute Immediate的SQL语句就会完成用户的要求。当然我们可能想要进行一些预处理和用户交互以确定我们要做的正是用户所需求的，特别是在进行更新操作时。例如，可能应该对所要删去的行数计数，然后询问用户删去这些数目的行是否正确。为简单起见，下文我们忽略这些确认。

那么我们究竟为什么需要Execute Immediate语句呢？我们已经会使用查找删除语句执行删除，它以如下的形式嵌入在程序中：

```
exec sql delete from customers where city = :cname2;
```

Execute Immediate语句究竟是如何提高灵活性的呢？我们看图5-24就可得到答案，图5-24

中列出了可供用户选择的三个不同的选项，通过最后一行“... (and so on)”暗示我们还可以有更多的选择。很明显如果用户在第3行中填写了新的内容，那么我们就需要一条完全不同的删除格式，例如：

```
exec sql delete from customers
  where discnt between :lowval3 and :hival3;
```

这里的`lowval3`和`hival3`由第3行中的两个填充格输入。在嵌入式SQL程序中为菜单的每一行声明一条不同的Delete语句，看起来是可行的，但这并不够！设想如果第2行和第3行都被选中，将会怎样？这就需要在程序中构造另外一条嵌入式Delete语句。由于在这样一个菜单中可能会有15个选项，那么我们最多允许的选项子集将为 2^{15} ，显然使用静态的SQL语句来预见所有用户可能的需求是不现实的。在源程序中我们拒绝使用 2^{15} 个不同的Select语句，取而代之的是需要能够使用动态SQL在需要时构造语句，灵活地反映出用户的需求。

2. Prepare、Execute和Using

除了使用立即执行语句之外，也可以使用两条动态SQL语句、Prepare和Execute，来完成同样的任务。这一方法允许在SQL语句中将宿主变量作为参数使用（通过Execute语句的USING子句完成），并且通过在运行之前执行编译来提高以某种形式反复执行的语句的性能。

例5.6.2 Prepare、Execute和Using 图5-25给出了一个程序，在主程序函数的第5行中的Prepare语句有这样的作用：它分析`sqlText[]`中字符串的语法，将它变成一个被编译过的

```
#include <stdio.h>
exec sql include sqlda;

exec sql begin declare section;
  char cust_id[5], sqltext[256];
exec sql end declare section;
char cidprompt[] = "Name customer cid to be deleted: ";

int main()
{
  /* The ? in the following line marks the dynamic parameter */
  strcpy(sqltext, "delete from customers where cid = ?");
  exec sql whenever sqlerror goto report_error;
  exec sql connect to testdb;

  exec sql prepare delcust from :sqltext;          /* prepare for loop */
  while((prompt(cidprompt, 1, cust_id.arr, 4)) >= 0) { /* loop for cid */
    exec sql execute delcust using :cust_id;        /* using clause ... */
                                                       /* ...replaces "?" above */
    exec sql commit work;                          /* commit the delete */
  }
  exec sql disconnect current;
  return 0;
report_error:
  print_dberror();
  exec sql rollback work; exec sql disconnect current;
  return 1;
}
```

图5-25 使用Prepare和Execute语句的DB2 UDB程序（例5.6.2的图解）

形式，并命名为`delcust`。接下来`Execute`语句运行`delcust`。注意到变量`sqltext`已经被一条包含动态参数`(:dcid)`的文本SQL语句初始化。`Prepare`语句中随后的动态参数可以由`Execute`的`USING`子句所指定的宿主变量值填写。

这一程序的`Connect`和`Disconnect`语句都使用了DB2 UDB/SQL-99的语法，动态参数的格式“`?` ”也是DB2 UDB/SQL-99的语法。在ORACLE中，除了需要将`Connect`和`Disconnect`语句做适当的修改外，还需要改变标记动态变量的“`?`”，将之变为以冒号为前缀的标识符，如，`:dcid`，即改为如下形式：

```
strcpy(sqltext, "delete from customers where cid = :dcid"); /* ORACLE form */
```

由于这些参数是使用`Execute`语句的`USING`子句按固定位置顺序替换，其实并没有必要使用`:dcid`这样的特定名字，标准的“`?`”更为可取。 ■

`Execute Immediate`方法提供了许多与使用`Prepare-Execute`方法相同的功能。`Prepare`可以接受的SQL语句与`Execute Immediate`语句所能使用的SQL语句是相同的，所以在这里我们得不到任何灵活性。在`Execute Immediate`中不能使用`USING`子句，但因为变量的值通常可以被转换成文本，并且可以被拼接成立即执行的文本语句，所以这不是一个严重的问题。如果存在几个非文本的变量值，`Prepare-Execute`方法可能会更简单些；和将这些变量转换并且连接成文本相比，将这些变量放置到`USING`语句中会更加简单。一个倾向于使用`Prepare-Execute`方法的更为普遍的原因是性能，在这种情况下语句必须被动态产生但要重复执行。编译`Prepare`语句会消耗一些重要的资源，因此最好避开编译形式的重复执行。`Execute Immediate`格式必须每次重新编译一条新的语句。

3. 动态选择：`Describe`语句和`SQLDA`

我们还没有介绍动态建立和执行`Select`语句。动态建立`Select`语句的问题在于在编译之前不知道要检索的列值的个数。因此一般在静态SQL中简单的`INTO`子句语法不能工作。考虑语句：

```
exec sql select cname, discnt into :cust_name, :cust_discnt
  from customers where cid = :cust_id;
```

在声明了两个知道类型的宿主变量`cust_name`和`cust_discnt`来从检索到的两列中得到值的情况下，这一语句将工作得很好。但是为了提供动态的灵活性，我们允许检索表中列的任意集合，可能是4列或者12列，并且是任意的类型。为了处理这些任意检索的集合，我们需要考虑一种新的结构类型，称为SQL描述符区域(SQL Descriptor Area, SQLDA)。我们首先介绍ORACLE的数据库产品中的动态`Select`语句，随后将考虑一些`SQLDA`的细节。然后我们会涉及DB2 UDB产品中相应的内容。由于ORACLE和DB2 UDB中`SQLDA`的数据结构完全不同，我们不能考虑将`SQLDA`的任何用法作为基本SQL的一部分。

例5.6.3 考虑图5-26中的ORACLE程序。靠近图5-26顶部的`exec sql include sqllda`语句包含了一个定义`SQLDA`结构的头文件（头文件本身在图5-28）。在`Connect`语句后的第一条语句初始化了程序的`SQLDA`指针，通过调用ORACLE的库函数`sqlald()`使其指向已初始化过的`SQLDA`结构。（这一程序使用的都是ORACLE特有的结构，因此我们放弃了在每一行中以ORACLE作为开头的注释，而是简单地称之为ORACLE程序。）

一个带有SQL`Select`语句的常量文本字符串`sqltext[]`被构造。类似的常量字符串通常不是动态执行的：由于我们预先知道完整的语法，所以使用嵌入式`Select`语句会更加简单。然而，这里我们介绍这一简单的情况是为了提供给大家一个较容易理解的具体例子。

当执行`Prepare`语句来准备`sqltext[] Select`语句时，编译过程计算`Select`语句所检索的列值

的个数和类型（这里Select语句列出了cname和city，在Create Table语句中它们分别被定义为varchar(13)和varchar(20)）。为了让程序知道这一信息，它执行语句Describe，这一语句将检索到的所有列的信息放到SQLDA变量结构中。注意，执行Describe之前，程序将sqlda->N设置为MAX_COLS，即允许列的最大个数；实际上在这里是没有必要的，因为sqlda->N已被sqlald()函数正确初始化了，然而在一个重复使用SQLDA结构的循环中，这一步是至关重要的。在Describe语句执行完之后，程序应当将sqlda->N设置为由Describe所返回的动态列实际个数sqlda->F。

Describe语句将每一列的类型说明赋予了数组sqlda->T[]。然而，这些类型说明每一个都有一个空值标记（表示此列是否允许有空值），需要在T[]每一个元素上调用ORACLE的sqlnul()函数来清除这一标志，留下一个纯类型值。这一函数也返回给调用者空值标志；如果这一列允许有空值则返回TRUE。

```
#define MAX_COLS 100
#define MAX_NAME 20
#include <stdio.h>
#include <malloc.h>
exec sql include sqlca;
exec sql include sqlda;
SQLDA *sqlald(int, int, int);
exec sql begin declare section;
char sqltext[256];
char user_name[20], user_pwd[10];
exec sql end declare section;
int main()
{
    int i, null_ok; SQLDA *sqlda;
    exec sql whenever sqlerror goto report_error;
    strcpy(user_name, "poneilsql"); strcpy(user_pwd, "XXXX");
    exec sql connect :user_name identified by :user_pwd;

    sqlda = sqlald(MAX_COLS, MAX_NAME, 0); /* allocate sqlda var */
    strcpy(sqltext,
        "select cname, city from customers where cid = 'c003'");
    exec sql prepare stmt from :sqltext;
    sqlda->N = MAX_COLS; /* before describe, set to max */
    exec sql describe stmt into sqlda;
    sqlda->N = sqlda->F; /* reset N to described #cols */

    for (i = 0; i < sqlda->N; i++) { /* loop through 2 cols */
        /* clear out null flag from T[i], return it in null ok variable */
        sqlnul(&(sqlda->T[i]), &(sqlda->T[i]), &null_ok);
        /* allocate space for the column value plus a null terminator */
        sqlda->V[i] = malloc((int)sqlda->L[i]+1);
        sqlda->I[i] = malloc(sizeof(short)); /* alloc space for indicator */
    }
    exec sql declare crs cursor for stmt;
    exec sql open crs;
    exec sql fetch crs using descriptor sqlda; /* only 1 row: no loop */
    printrow(sqlda); /* print out row (Fig 5.27) */
    exec sql close crs;
    for (i = 0; i < sqlda->N; i++) {
        free(sqlda->V[i]); /* free col data space */
        free(sqlda->I[i]); /* free indicator space */
    }
    exec sql commit release;
    return 0;
report_error:
    print dberror(); /* See Appendix B for code */
    exec sql rollback release;
    return 1;
}
```

图5-26 执行动态Select的ORACLE程序

每一动态列需要内存区域存放列值和指示器值。这些内存区域通过在SQLDA中的V[]和I[]数组内设置指向它们的指针来指定。这里我们使用C语言库函数malloc()分配内存。注意，malloc()方法相对而言比较耗费CPU，考虑到这一点，程序员应该尽可能一次性为一组列值分配空间。

最后，声明了一个命名为crs的游标。虽然我们知道在这一例子中只会检索到一行，但对于动态选择游标总是必须使用的。

图5-27中的函数printrow()是显示任意数量的字符串类型的列变量的普通函数。将这一函数简单地扩展就可以使之显示多个不同的列类型。为了做到这一点，我们需要从sqlda->T[i]中读取每一列的数据类型，然后执行一条C语言的转换语句处理每一种类型。如果需要，可以通过使用1代表字符，3代表整形等等替换数据库类型T[i]，来完成类型的强制转换。几乎所有的类型都能够被强制转换成字符数据。很自然，在这种情况下V[i]必须指向足够大的可以容纳转换后表示形式的内存区域。另外，L[i]也要更新以反映现在内存区域的大小。

```
/* printrow( ): print two varchar cols in sqlda, for ORACLE */
#include <stdio.h>
exec sql include sqlda;
void printrow(SQLDA *sqlda)
{
    char *s;
    int i, length;

    for (i = 0; i < sqlda->N; i++) {           /* loop through columns */
        if ((*sqlda->I[i])) {                  /* if actually null ... */
            printf("null\n");
            /* ... display null */
        } else {
            s = (char *)sqlda->V[i];           /* point to column string value */
            length = sqlda->L[i];
            printf("%*s\n", length, s);         /* print non-null string in column */
        }
    }
}
```

图5-27 处理多个字符串变量列的ORACLE函数Printrow

例5.6.3使用了动态SQL从数据库检索数据到程序变量。在查找条件中如果使用任意数量的宿主变量，那么由于这些需要填入常数值的子句个数事先未知，因此你的程序也需要一种动态的方法来描述这些任意数量的宿主变量。需要使用语句Describe Bind Variables将这些宿主变量绑定到查询条件。具体细节请参阅本章结尾处的推荐读物[7]《*ORACLE Programmer's Guide to the Pro*C/C++ Precompiler*》。

图5-28提供了SQLDA头文件中由以下ORACLE语句读取的内容。

```
exec sql include sqlda;
```

ORACLE中的SQLDA在多个数组的数组项中保存了列的有关信息，这些数组包括例5.6.3中用到的V[]，L[]，T[]，I[]。注意，数组T[]所指向的不同数据类型的代码不在这一头文件中被定义，而是在使用到它们的源文件中被定义。

DB2 UDB在动态SQL方面与ORACLE有十分类似的功能。SQLDA结构的部分名字是不同

的，Include文件需要改变，还有一些其他的小改动，包括通常的Connect和Disconnect语句的标准化。就像在SQL标准（Full SQL-92和SQL-99，但不在Core SQL-99）里声明的，语句Prepare，Describe，Declare Cursor，Open，Fetch和Close的顺序也是一样的。

```
struct sqlda
{
    long      N;          /* maximum # of columns handled by this SQLDA        */
    char     **V;         /* pointer to array of pointers to col values        */
    long      *L;         /* pointer to array of lengths of column values      */
    short     *T;         /* pointer to array of types of columns                */
    short     **I;         /* pointer to array of ptrs to ind variables          */
    long      F;          /* actual number of columns in this SQLDA            */
    char     **S;         /* pointer to array of pointers to column names       */
    short     *M;         /* pointer to array of max lengths of col names      */
    short     *C;         /* pointer to array of actual lengths of col names    */
    char     **X;         /* pointer to array of addresses of ind var names     */
    short     *Y;         /* pointer to array of max lengths of ind var names   */
    short     *Z;         /* pointer to array of actual lengths of ind var names */
};
```

图5-28 ORACLE中通过exec sql include sqlda所读到的sqlda结构

```
struct sqlvar           /* Variable Description        */
{
    short      sqltype;    /* Variable data type          */
    short      sqllen;     /* Variable data length        */
    char     *sqldata;    /* Pointer to variable data value */
    short     *sqlind;    /* Pointer to Null indicator   */
    struct     sqlname;   /* Variable name               */
};

struct sqlda
{
    char      sqldaid[8];  /* Eye catcher ~ 'SQLDA'        */
    long      sqldabc;    /* SQLDA size in bytes=16+44*SOLN */
    short     sqln;       /* Number of SOLVAR elements   */
    short     sqld;       /* # of columns or host vars.   */
    struct sqlvar  sqlvar[1]; /* first SQLVAR element        */
};

/* macro for allocating SQLDA
#define  SQLDASIZE(n) (sizeof(struct sqlda) + \
                        ((long) n-1) * sizeof(struct sqlvar))
```

图5-29 DB2 UDB中sqlda.h的一部分

SQL标准没有指明SQLDA数据结构，所以这一部分在不同的数据库产品中会有所不同。ORACLE的SQLDA包含用来描述列和值的数字和指针的数组，而DB2 UDB的SQLDA有一个结构数组存放相同的信息，为每一个从数据库转移到程序（或反之）的值准备了一个sqlva结构。所以，在ORACLE中，指向第*i*个数据值的指针通过sqlda->V[i]引用，它是SQLDA中V指针数组的第*i*个元素，而在DB2 UDB中，它通过sqlda->sqlvar[i].sqldata引用，这是

SQLDA中sqlvar数组第*i*个sqlvar结构的sqldata成员。参见图5-29得到DB2 UDB声明头文件sqlda.h最重要的部分。SQL-92和SQL-99标准用更多的SQL关键字和语法代替了产品中使用的SQLDA，试图使它们独立于嵌入的语言。然而，这一方法目前看来还没有影响到数据库产品的发展。

例5.6.4给出了与例5.6.3的ORACLE产品的动态SQL程序相同功能的代码和详细的说明。

例5.6.4 考虑图5-30的DB2 UDB程序，对应于图5-26的ORACLE程序。图5-30顶部的包含sqlenv.h通过它自身包含sqlda.h定义了SQLDA结构(sqlda.h文件的一部分在图5-29中)。

```
#define MAX_COLS 100
#include <sqlenv.h>
#include <stdio.h>
exec sql include sqlca;                                /* communication area */
void printrow(struct sqlda * );
exec sql begin declare section;
char sqltext[256];
exec sql end declare section;
int main( )
{
    int i; struct sqlda *sqlda;
    exec sql whenever sqlerror goto report_error;
    exec sql connect to testdb;                         /* DB2 UDB Connect, simple form */

    sqlda = (struct sqlda *)malloc(SQLDASIZE(MAX_COLS));/* allocate sqlda area */
    strcpy(sqltext, "select cname, city from customers where cid = 'c003'");
    exec sql prepare stmt from :sqltext;                /* compile select statement */
    sqlda->sqln = MAX_COLS;                            /* set max # dynamic columns */
    exec sql describe stmt into :*sqlda;                /* describe this statement */

    for (i = 0; i < sqlda->sqld; i++) {                  /* loop through 2 described cols */
        /* allocate space for the column value plus a null terminator */
        sqlda->sqlvar[i].sqldata = malloc(sqlda->sqlvar[i].sqllen + 1);
        sqlda->sqlvar[i].sqlind = malloc(sizeof(short)); /* and for indicator */
    }

    exec sql declare crs cursor for stmt;
    exec sql open crs;                                  /* open cursor */
    exec sql fetch crs using descriptor :*sqlda;       /* only 1 row: no loop */
    printrow(sqlda);                                    /* print out row (Fig 5.30) */
    exec sql close crs;

    for (i = 0; i < sqlda->sqld; i++) {
        free(sqlda->sqlvar[i].sqldata);                /* free col data space */
        free(sqlda->sqlvar[i].sqlind);                 /* free indicator space */
    }

    exec sql commit work; exec sql disconnect current; /* or connect reset here */
    return 0;

report_error:
    print_dberror( );                                /* See Appendix B for code */
    exec sql rollback work; exec sql disconnect current;
    return 1;
}
```

图5-30 执行动态Select的DB2 UDB程序

Connect语句之后的第一条语句初始化了程序的SQLDA指针，令它指向通过调用C语言库函数malloc()得到的一个未经初始化的SQLDA结构。

当执行Prepare语句来准备sqltext[]的Select语句时，编译过程计算Select语句所检索到的列值的个数和类型（这里Select语句列出了cname和city，在Create Table语句中它们分别被定义为varchar(13)和varchar(20)）。为了让程序知道这一信息，执行语句Describe，这一语句将检索到的所有列的信息放到SQLDA变量结构中。注意，执行Describe之前，程序将sqlda->sqln设置为MAX_COLS，即允许列的最大个数。执行了Describe语句之后，从sqlda->sqlid中可得到由Describe返回的动态列的实际个数。

Describe语句将每一列的sqlvar结构赋给了数组sqlda->sqlvar。实际上，在Describe之后，sqlda->sqlvar[0].sqllen给出了第0列值的长度，并且sqlda->sqlvar[1].sqlname是描述第一列值列名的一个结构。

每一动态列需要内存区域存放列值和指示器值。这些内存区域通过在SQLDA中的sqlvar结构内设置指向它们的指针来指定。这里我们使用C语言库函数malloc()分配内存。注意，malloc()方法相对而言比较耗费CPU，考虑到这一点，程序员应该尽可能一次性为一组列值分配空间。

最后，声明了一个命名为crs的游标。虽然我们知道在这一例子中只会检索到一行，但对于动态选择游标总是必须使用的。

图5-31给出了这一程序的printrow函数，它可以在一个单独的文件中。这一函数与ORACLE中相应函数唯一的差别在于数据结构部分的名字。同样它也可以通过转换由Describe所返回的类型信息来扩展处理各种不同的数据类型。

```
/* print two varchar cols currently in sqlda, for DB2 UDB */  
#include <stdio.h>  
#include <sqlenv.h>  
void printrow(struct sqlda *sqlda)  
{  
    char *s; int i, length;  
  
    for (i = 0; i < s->sqlid; i++) { /* loop through columns */  
        if (*(sqlda->sqlvar[i].sqlind)) { /* if actually null . . . */  
            printf("null\n"); /* . . . display null */  
        } else {  
            s = (char *)sqlda->sqlvar[i].sqldata + 2; /* point to string val */  
            length = sqlda->sqlvar[i].sqllen; /* and find its length */  
            printf("%*s\n", length, s); /* display non-null value */  
        }  
    }  
}
```

图5-31 处理多个字符串变量列的DB2 UDB函数Printrow

5.7 一些高级的编程概念

本节包括对可滚动游标、游标敏感性和数据库编程环境的简单介绍。

1. 可滚动游标

在图5-5之后的讨论中我们提到了使用标准游标的Fetch语句的一个限制，就是游标只能在一个记录集中向前移动。这意味着如果要再次检索某一行，就必须关闭并且重新打开一个游标。在Full SQL-92和Full SQL-99中提出了对Declare Cursor和Fetch语句的泛化。新的Declare Cursor语句在图5-32中给出。

```
EXEC SQL DECLARE cursor_name [INSENSITIVE] [SCROLL] CURSOR [WITH HOLD] FOR
  Subquery
  {UNION Subquery}
  [ORDER BY result_column [ASC | DESC]
   {, result_column [ASC | DESC]}...]
  [FOR READ ONLY | FOR UPDATE OF columnname {, columnname}...];
```

图5-32 高级SQL的Declare Cursor语句

两个高级SQL语法元素是关键字INSENSITIVE和SCROLL。这些不包括在Core SQL-99中，但是图5-32中其余的语法在Core SQL-99中都有。我们稍后就会谈到游标敏感性。当在游标定义中使用关键字SCROLL时，这一游标就被称为是可滚动的，并且就可以使用SQL-99中Fetch语句的高级功能。这一高级SQL的Fetch语句在图5-33中。

```
EXEC SQL FETCH
  [{NEXT | PRIOR | FIRST | LAST          -- advanced
  |{ABSOLUTE | RELATIVE} value_spec} FROM ]
  cursor_name INTO host-variable {, host-variable...};
```

图5-33 高级SQL的Fetch语句

位置移动的说明（NEXT、PRIOR、…）被称为定向。现在标准的缺省动作是NEXT，意味着“从游标现在的位置起顺序检索下一行”。PRIOR定向意味着“检索游标当前所在位置之前的那一行”。FIRST定向和LAST定向检索游标所在行集的第一行或最后一行。ABSOLUTE定向检索由值value_spec指定的位置的行，它可以从1到游标行集中所有行的个数；ABSOLUTE 1等同于检索第一行。负值也可以使用，从 -1 到 -n；ABSOLUTE -1 等同于检索最后一行。RELATIVE意味着检索离当前位置整数个数距离的那行。因此RELATIVE -1 同PRIOR一样、RELATIVE 1 同NEXT、RELATIVE 0再一次检索当前检索的这行。

由于ORACLE和DB2 UDB在它们的server SQL中都不直接支持这一高级语法，我们不能在基本SQL中考虑它们，也许你会觉得这并不值得烦恼；然而，ODBC是支持它的，ODBC指开放数据库连接性(Open Database Connectivity)，ODBC API是一个重要的C语言程序接口，它提供了产品之间的协同工作。要更详细地了解ODBC，参阅本章结束部分的推荐读物 [11]《*The ODBC Solution*》。

ODBC直接导致了Java的数据库连接性包JDBC(Java Database Connectivity API)的产生。JDBC版本2.0包含在Java版本1.2中，它有可滚动的结果集，相当于可滚动游标，它也支持敏感性概念，这一概念将在后面加以介绍。可以浏览www.sun.java.com得到更多关于JDBC的信息。一些数据库产品，包括ORACLE和DB2 UDB也提供了SQLJ(Java的嵌入式SQL)，对于单个数据库上的简单的应用进行处理时它是比较容易的。

2. 游标敏感性

回忆一下，事务这一概念的产生是为了隔离某一用户执行的一系列逻辑，使其不受其他并发更新的影响。我们将在第10章中学习更多关于事务的内容，但是这种隔离意味着一旦在一个事务中打开了某一游标，这一游标中的行就不能被任何其他并发用户更新。这样做很好，但是在同一个事务中如果一些对游标中行所作的更新会对这一事务中一些其他的更新产生副作用，那么该怎么办呢？

例5.7.1 假设我们编写一个称为ord_ship的应用程序，实际上是装运已经订购的货物。orders表中的订单还没有被装运，ord_ship逻辑找出orders表中在this_mo月份订购的所有订单，调用将这一货物运给客户的子程序，然后删除表orders中的当前行，并且将其放在与表orders有相同列的表shipped_ords中。访问表orders中所有相关行的游标如下：

```
declare cursor ship_em cursor for
    select * from orders for update of ordno
    where month = :this_mo;
```

有时装运某一订单商品的程序会返回值指出由于商品脱销没法装运。这种情况下通常的做法是建立一张后备的订单，意味着当有存货时就装运这一订单上的商品。最简单的做法是回溯订单，将next_mo的值设为this_mo的下一个月，并且用此替换相关订单的month列的值。假设我们以如下语句来实现这一点：

```
update orders set month = :next_mo
    where pid = :ord_rec.pid;
```

这里ord_rec.pid就是我们发现的脱销商品的pid值。现在假设在游标ship_em中有许多其他订单行的pid正是我们刚才遇到的那一pid值。那么这些行还在这一游标中存在吗？对于游标ship_em的orders行的定义属性是列month的值为:this_mo，而现在这一month值改变了，那么这些行在使用Fetch循环时到底是否被检索到呢？可能用户会记得如果我们在打开了一个游标之后改变了this_mo的值，这并不影响选到的行；但是这里的情况不同，因为我们实际上在改变用于限定游标所应占有行的数据。如果这个答案太直接了，那么考虑一下，如果不是仅仅更新脱销商品订单行中的month，而是将这些订单行放到一个不同的表中，并且从orders表中删除订购这一商品的行，会怎样？

```
delete orders where pid = :ord_rec.pid;
```

现在当我们从游标ship_em中检索行时会发生什么情况？这一行根本不存在，所以我们不能检索到。并且Fetch没有办法检索不存在的行，所以我们不能够将其指向它原来所在的位置；可能我们需要跳过这一由先前的行留下的空位置。当然有可能当游标最初被打开时系统会有一个对数据的“快照”，这样我们就有这些已被删除的行的拷贝可以提供给Fetch。 ■

对例5.7.1问题的回答是在SQL-92之前没有标准，并且不同的产品有不同的做法。问题的答案可能依赖于这一游标是只读的还是其他类型。然而，在Full SQL-92中，这一选择由Declare Cursor语法所决定。再看一下图5-32的一般格式，注意关键字INSENSITIVE。当这一关键字出现时，游标被称为是不敏感的，这意味着作为查找更新或查找删除的结果，在这一游标中的行是不会改变的。这一效果就好像当游标打开时系统对这一游标中的行做了快照（然而，这一方法效率不高）。现在程序可以看到并不在其中的行；但是如果执行的是定位更新或定位删除（使用语法WHERE CURRENT OF CURSOR），会产生错误告知行不存在。如

果在遵照基本SQL功能的Select语句中省去关键字INSENSITIVE，那么在游标外面的更新和删除会立即反映到这一游标检索到的行集中。在可滚动游标的情况下，这意味着使用ABSOLUTE 23当一行被检索两次时可能得到不同的结果。

3. 数据库编程的其他开发环境

一些产品，如ORACLE、INFORMIX和SYBASE，用过程性语言来扩展SQL功能。有关ORACLE(PL/SQL)和INFORMIX(SPL)的过程性语言的某些内容参见4.4节。这些面向特定产品的语言通常有局部的内存变量、If-then-else类型的语句和建立、调用函数的功能。多数情况下，也存在一种方法建立较为友好的可视用户界面，如快速建立的表单、菜单、通过点击控制在检索到的游标上移动的能力，以及其他种种功能，使得开发应用程序更加简单。DB2 UDB现在正使用SQLJ(Java中的嵌入式SQL)或者Java的JDBC连接性包试图使Java成为新的更安全的过程性语言（比它现在所支持的C/C++和类似的语言更安全）。

使用过程性语言接口PL/SQL和SPL的唯一困难在于它们在细节上有较多的差异：每种产品都有自己的特点，并且当前仅仅在SQL-99/PSM（Persistent Stored Modules，持久的存储模块，SQL-99的一部分）中有一些可遵循的标准。

推荐读物

Kernighan和Ritchie编写的《*The C Programming Language*》[4]是一本优秀的指南。第3章和第4章中列出的各种SQL标准和具体产品的SQL参考手册仍然是有用的读物。用于嵌入式SQL的新参考手册，包括一般的嵌入式SQL语句结构以及在C语言中如何使用SQL的相关指南。专业人员同时也需要了解各个数据库产品的错误代码参考手册。

- [1] Don Chamberlin. *A Complete Guide to DB2 Universal Database*. San Francisco: Morgan Kaufmann, 1998.
- [2] *DB2 Universal Database Application Development Guide*. Version 6. IBM, 1999. Available at <http://www.ibm.com/db2>.
- [3] *DB2 Universal Database SQL Reference Manual*. Version 6. IBM, 1999. Available at <http://www.ibm.com/db2>.
- [4] Brian W.Kernighan and Dennis M.Ritchie. *The C Programming Language*. 2nd ed. Englewood Cliffs, NJ:Prentice Hall, 1988.
- [5] Jim Melton and Alan R.Simon. *Understanding the New SQL: A Complete Guide*. San Francisco:Morgan Kaufmann, 1993.
- [6] *ORACLE8 Error Messages*. Release 8.1.5. Redwood Shores, CA :Oracle. <http://www.oracle.com>.
- [7] *ORACLE8 Programmer's Guide to the Pro*C/C++ Precompiler*. Redwood Shores, CA: Oracle. <http://www.oracle.com>.
- [8] *ORACLE8 Server Concepts*. Volumes 1 and 2. Redwood Shores, CA: Oracle. <http://www.oracle.com>.
- [9] *ORACLE8 Server SQL Reference*. Volumes 1 and 2. Redwood Shores, CA: Oracle. <http://www.oracle.com>.
- [10] *Data Management: Structured Query Language(SQL) Version 2*. Berkshire, UK:X/Open

Company,Ltd.,1996.Email:xospecs@xopen.co.uk.

- [11] Robert Signore, John Creamer, and Michael Stegman. *The ODBC Solution*. New York:McGraw Hill,1995.

习题

在本书后的“习题解答”中有答案的习题用•标记。在编写这些习题的程序的时候，你应该知道可执行文件可能相当大。这是因为数据库代码库很大，而且有时它们会被捆绑到单独的应用程序中去。你要注意限制你的目录下的可执行文件的数目，以节省磁盘空间。

注意 习题5.1和5.2只涉及5.1节中包含的嵌入式SQL的特性。它们无需做指定以外的数据安全考虑。

5.1 输入例5.1.2的程序，并针对你的计算机上的数据库产品对它进行必要的修改，然后执行之。

5.2 • 写一个在循环中提示用户输入一个客户ID (cid) 和一个产品ID (pid)(各占一行) (使用例5.1.1后介绍的prompt()函数) 的程序。该程序应该逐行显示每一个提供pid给cid的代理商的aid和由每个代理商提供的qty总数的列表。如果提供的cid或pid的值在Customers表或Products表中不存在，则程序应该不返回任何行。当用户输入一个空行后，程序应该停止。

5.3 写一个在循环中提示用户输入一个代理ID (aid) 和一个产品ID (pid)(各占一行) (使用例5.1.1后介绍的prompt()函数) 的程序。该程序应该逐行显示通过aid订购pid的customers的cid和该顾客从该代理商处订购的每种产品的美元金额总数。如果用户提供的aid或pid值在agents或products表中不存在，则程序应该不返回任何行。当用户键入一个空行后，程序应该停止。

5.4 修改习题5.2的程序使它在输入的cid或pid不存在于customers或products表时通知用户并要求用户重新输入相应的cid或者pid。特别是，程序要能保持有效的pid，而要求重新输入错误的cid；或者保持有效的cid，而要求重新输入错误的pid——换句话说，就是尽量减少用户的击键次数。注意，不要修改prompt()函数。如果prompt()为用户输入的cid和pid返回一个负值，就假设输入的是一个空行。

5.5 这里要求写一个大型的应用程序。

- (a) 写一个主程序调用menu()函数。menu()函数为用户提供三个选项：(1) City Agent and Customers List, (2) Agent Performance, (3) Exit This Program。menu()函数应该返回一个值给主程序。主程序然后根据用户的选择，调用函数city()或perform()或退出。在选择(1)或者(2)以后，主程序应该循环调用menu()函数以获得进一步的输入。所有这些程序都应该有格式整齐的I/O声明，以使用户知道到底发生了什么。

函数city()应该要求输入某个城市的名称（例如Duluth），然后返回居住于那个城市的顾客的cid列表和代理商的aid列表。同时，两张列表必须满足以下性质：(1) 两张列表必须用明确的标签被区分开；(2) 列表中只要列出那些通过同一个城市的某个代理商至少订购了一次的商品的顾客的cid；同样列表中只要列出那些至少给一个同一城市的顾客提供商品的代理商的aid；(3) 在两张列表中aid和cid都分别只出现一次。如果不小心，在把一些行插入到orders表中去的时候

就会引起重复。如果不能保证(3)，那么在一张很大的orders表中就可能出现极多的重复行。注意，你必须在你的程序中创建两个游标以保证此性质——一条SQL语句多半不能达到此目的。可以把city()函数看做被来访的批发公司的行政官用来联系顾客和他们的代理商，邀请他们相互见面共进晚餐的工具。

函数perform()应该要求用户输入一个代理商的aname。当且仅当在agents表中存在相同的aname的时候，程序通过列出名为aname的每个代理商的city和aid的值，并要求用户选择输入的aid之一，来保证输入的代理商的唯一性。然后，perform()函数应该计算该代理商的销售总额、各代理商的销售总额的最大值以及该代理商的销售总额与最大值的比率。

- (b) 这里是区分同名代理商中到底需要哪个的另一种办法。取出代理商和他们所在的城市的列表，标以数字1, 2, 3, ...并输出，然后要求用户选择一个数字。在要求用户输入前先进行提交。接着使用同一游标声明重新取出行，通过指定的行号计数——这样就得到了需要找的代理商。解释为什么这种方法在还有其他用户在并发地更新数据库时不保证有效。(a)中的方法在这种情况下可靠吗？请给出说明。

注意 很多下面的编程作业显式地或隐式地和事务相关。在下面，除非显式地提到死锁发生的可能，你假设不需要编写错误处理程序来处理事务死锁。

5.6 (a) 写一个程序由一个代理商输入一张订单，并把它加入到数据库中去。主程序应该在循环中重复地要求输入订货的顾客的cid、供货的代理商的aid、订购的商品的pid、订购商品的数量quantord以及订单的月份。然后程序调用do_transaction函数以执行一个事务来把订单放入数据库。do_transaction函数的参数就是用户输入的值。注意，用户输入的值(cid, pid等)在main()中只是普通的程序变量，所以它们需要被复制到do_transaction中的SQL声明区域中。首先，事务验证pid在products表中、aid在agents表中以及cid在customers表中，而且products.quantity减去quantord不会使products.quantity小于0。然后，事务减去quantord个该商品，并利用products.price计算这些商品的价值。接着，事务减去客户的折扣以调整价格。如果pid、cid、aid中的任何一个不存在(为空)或者任何其他条件不满足，程序应该中止事务，接着输出合适的消息并从do_transaction函数中返回。如果所有的条件都满足，事务在orders表中插入一行，并把计算出来的价格放在dollars列中。ordno的值应该通过一个特殊的调用来得到，此调用应保证每次执行都返回一个新的值。对于此习题，写一个使用一个初始值为1027的静态变量的函数intfake_get_ordno()，每次调用该函数时，变量自增，并将值返回。最后，事务应该提交。该程序应该注意事务死锁异常中止。在这种异常中止发生以后，事务应该重试。重试的次数最多4次。

- (b) 考虑如何实现get_ordno()。这是一个即使在几个程序同时使用它时也能保证返回给每个调用者一个新的ordno的函数。很明显，我们需要利用数据库本身来保证一致的行为。建立只有一行的表ordno，列curordno包含了当前的ordno。写short int get_ordno(char *errmsg)来访问并更新此表。它返回一个新的ordno。如果执行失败，就返回-1，此时，errmsg中存放相关的错误字符串。

用一个简单的驱动程序检测此函数。假设get_ratio函数将在放入订单的事务中被调用，于是，它不应该有自己的提交或回滚操作。请说明原因。

- (c) 注意(b)中的ordno表将成为多用户数据库的一个“热点”。这是因为每一个放入订单项的事务都要访问表中的一行。解释为什么所有的并发事务在得到ordno以进行自己的工作之前都必须等待持有当前ordno的事务的提交或者回滚。提高性能的一种办法是把获得ordno从订单项事务中分离开来。把它放在放入订单事务之前，作为一个单独的事务。这样，get_ordno可以有自己的提交、回滚和死锁重试逻辑。解释为什么在orders表中的结果ordnos存在跳跃值，例如，ordnos 1100,1101,1102,1104,…，用(b)中的方法，这个问题会出现吗？
- (d) •解释为什么在cid、aid、pid插入检测、是否有足够数量的产品满足要求的检测（事务的读部分）之后、以及Update和Insert语句（事务的写部分）之前不应该放置Commit语句。在最后一个读写动作（insert）之前有放置提交的合适地方吗？

5.7 考虑图5-13中包含两个Update语句事务的转账程序。

- (a) 设想在此程序中没有任何事务包含这两个单行Update语句。那么考虑几个对应于这个改过的程序的进程并发地运行，且没有其他程序访问accounts表的情况。请说明“如果每个对单独行的单个更新不可分的话，事务仍然能够按照所预想的运行”的原因。
- (b) 现在假设另一个进程正在运行给两个账号增加结余以作决策的程序。解释为什么现在转账程序必须遵守可串行化规范以获得预期的结果。请参照例5.4.1回答此问题。
- (c) 需要一个核对作为转账结果的账号的余额不为负的修改过的转账程序（注意，我们假设dollars变量中的转账总额可以为负）。那么应该如何修改程序？注意类似的代码在图5-20中。
- (d) (a)中的结论对于(c)仍然有效吗？即运行(c)中程序的不包含检测和Update语句的事务进程仍然能够正确执行吗？如果发生更新，单行的更新仍然不可分，但是，更新可能根本就不会发生。（提示：转账总额可以为负对于考虑此问题至关重要。）

5.8 写一个程序，它重复要求用户输入顾客ID，然后打印出代理商名、代理商ID以及由该顾客的不同代理商在他们的美金总额中间值中放置的订单美金额。其结果就像我们执行了如下语句：

```
select fname, lname, median(dollars) /* INVALID SYNTAX */
  from orders
 where cid = :cid group by fname, lname;
```

5.9 写一个程序打印所有代理商销售的平均美元数额的总数。并给出一个例子说明这和所有代理销售商的总美元数的平均值不同。

5.10 •写一个程序打印出例3.11.5中说明的报表。

5.11 按照例3.11.6的方法创建employees表。编写一个程序，它重复要求用户输入eid。程序按照在管理层次中从下往上地顺序输出接受eid汇报的经理的序列。注意程序应该正确地中止。

注意 把下面的问题当作可能在考试中出现的简答题来回答。

5.12 (a) • 编写一段由exec sql begin declare section开始的C语言代码，声明一个名为city的数组用来存放agents表中声明为varchar(20)的city名称。

(b) • 声明一个游标cc。该游标对应于检索存在提供低于1美元商品的代理商所在的城市的城市名（每个只出现一次）的SQL查询。

(c) • 编写一个循环连续地把城市信息放到数组city中去，并打印出它们。当没有城市信息可以检索的时候，循环应该中止。

5.13 编写可以放置于可执行SQL语句后以代替以下Whenever语句的等价的测试（对sqlcode进行测试）。

(a) • exec sql whenever not found go to handle_error;

(b) • exec sql whenever sqlerror go to handle_error;

5.14 考虑如下的SQL语句：

```
select c.city, cid, aid, pid
  from customers c, agents a, products p
 where c.city = a.city and a.city = p.city;
```

首先要保证同一个城市中(c.city)的所有(cid, aid, pid)三元组按组出现（一行接一行）。

(a) • 如何修改上面的SQL语句以保证有相同的c.city的三元组一行接一行地出现？（提示：使用一个非过程性的Select语句。）

(b) • 如果在customers表中有30行的c.cid='New York'，agents表中有10行的a.city='New York'，products表中有20行的p.city='New York'。那么上面的SQL语句的结果中有多少c.city='New York'的行？

(c) • 为了限制在(b)这种情况下看到的信息数量，要写一个报表使得每个City只出现一次（作为标题），下面跟着三个具有该city值的cid、aid、pid值列表。给出伪代码来表明你是如何在嵌入式SQL中做到这一点的。（要注意游标的声明。）

5.15 假设在products表中有50个不同的pid值，而且所有这些值都在orders表中出现。现在要列出其中10个最畅销商品的信息，即orders表中销售总额最高的商品的pid、pname以及销售总额。

(a) • 用非过程性SQL不可能做到这一点。这是因为没有办法在获得10个最大销售额的商品以后就停止。所以我们要写一个嵌入式SQL程序，它从游标中得到10个最高销售额的商品。请给出如何声明非过程性SQL的游标，以使得所需要的信息在获取行时能尽快被得到。

(b) • 在(a)中，当在循环中从游标提取行的时候，注意如果开始得到dollars为空值的时候程序可能会碰到问题。（在某些数据库产品中，空值在排序时大于非空值；而在另一些产品中，空值在排序时小于非空值。）如何从(a)中的标游编写Fetch循环，以跳过提取到的行的初始的总额为空的行？

5.16 考虑如下的代码段，其中的两条语句代表一个事务。

```
exec sql whenever sqlerror stop;
```

```

begintx:
    exec sql update orders
        set dollars = dollars - :delta
        where aid = :agent1 and pid = :prod1 and cid = cust1;
    exec sql update orders
        set dollars = dollars + :delta
        where aid = :agent2 and pid = :prod1 and cid = cust1;

```

重写此代码段，修改并增加语句，以检测在事务执行过程中可能碰到的死锁中止错误（参见图5-18），并让程序在发生死锁时重试这两条语句。注意，你要确保你的检测的确有效！重写以后的代码在这两条语句成功执行后应该提交整个事务；如果遇到其他错误，程序逻辑应该跳转到handle_err。

5.17 在动态SQL中，假设有一个像例4.6.3中程序开始部分那样的ORACLE SQLDA声明。变量sqlda是一个指向类型为SQLDA的结构的指针。

- (a) • 在Prepare语句后，如何增加代码来提供输出数据的列的标题？如果使用ORACLE，可以利用列名数组和sqlda→M[]以及最长列名字符串长度数组。类似地，对于DB2 UDB，使用sqlda→sqlvar[i].sqlname.data以获得列名字符串，使用sqlda→sqlvar[i].sqlname→length以获得short int型的字符串长度。
- (b) • 在从动态游标中抽取数据以后（在主程序中），如何编写C代码测试是否抽取的第二列是空值？（如果测试到空值，调用handle_null()函数。）

5.18 编写一个程序，进行如下循环：输出SQL>提示符，然后接受用户的一行输入作为动态SQL语句执行。如果输入为空行，则退出程序。这个程序有点像SQL监控器（如ORACLE的SQL*Plus）的核心循环。但是，这个简单的监控器不能执行Select语句。（考虑如何扩展此程序，使之能够执行Select语句。）

5.19 假设你需要修改products表中的某些商品名称，而且你有一个文件包含了每个要改名的产品的pid和新的pname字符串（用空格分开）。

- (a) 编写一个程序从标准输入（stdin）中一行一行地读文件，然后执行所需要的更新。接着利用文本文件重定向执行此程序。你可以使用带空提示字符串的prompt()函数从标准输入中读取文件的每一行。对于所需执行的Update语句，使用Prepare和Execute。
- (b) 作为另一个选择，使用载入工具把文件中各行装入到表pchanges（包含pid和pname列）的行中。请只使用一条SQL语句利用pchanges表更新products表。

5.20 我们希望从orders表中检索一些列，其中被检索的列由用户指定。编写一个程序提示用户逐个输入到底要检索哪些列（输入列名），直到用户输入一个空行为止。然后输出orders表中所有行的这些列。列的顺序按照用户输入的次序。你需要像例5.6.3（对于ORACLE用户而言）或例5.6.4（对于DB2 UDB用户而言）那样用到动态SQL的所有功能。在ORACLE中，对于qty和dollars列，你可以通过设置T[i]为1，并把L[i]（以及内存区）设得足够大使得有足够的空间来存放值的文本形式。在DB2 UDB中，如果需要，你可以使用SQL中的CAST。

第6章 数据库设计

到目前为止，我们已经处理了由很多不同的表构成的数据库，但还没有涉及到这些表和组成它们的列最初是如何生成的。数据库逻辑设计（Logical database design），也被简单地称做数据库设计（Database design）或者数据库建模（Database modeling），研究数据项的基本属性以及它们之间的相互关系。它的目标是用数据库的基本数据结构表示现实世界中这些数据项。具有不同数据模型的数据库，它们表示数据的数据结构是不同的。在关系数据库中，表示数据的数据结构就是我们所说的关系表。我们将在本章中集中叙述关系数据库，因为对象-关系模型的设计方法的研究目前仍处于初级阶段。希望在这本书将来的版本中，我们能够对对象-关系数据库的设计讨论得更多一些。

逻辑数据库设计是数据库管理员（DBA）的责任。他使用一种方法将数据库中相关联的数据项分配到表的不同列上去，这种方法应当能够保持所期望的性质。对于逻辑数据库设计的最重要的衡量标准是：表和属性如实地反映现实世界对象的相互关系，并且将来在对数据库做任何可能的更新后，依然保持这一点。

数据库管理员首先要研究一些现实世界的实体，比如说一个批发订购商店、一个公司人事部门或者一所大学的登记管理部门，这些单位的运作都需要计算机数据库系统的支持。数据库管理员通常与某个对该组织细节有丰富经验的人员共同工作，逐渐提出一个包含数据项和潜在数据对象的列表（例如，对于一所大学的登记管理部门，这张列表中可能含有 student_names, courses, course_sections, class_rooms, class_periods 等等），同时还要提出一些与这些数据项相互关系相关的规则，或称为约束。下面是典型的学生登记规则：

- 每一个登记在册的学生有一个唯一的学号（我们定义为sid）。
- 一个学生在一个特定的课时时间段只能选择一门课程。
- 一间教室在一个特定的课时时间段只能提供给一门课程使用。

当然还有其他一些规则。根据这些数据项和约束，数据库管理员将进行数据库的逻辑设计。在这一章中介绍了数据库设计的两种一般技术。第一种是实体-联系方法（entity-relationship approach），或称为E-R方法；第二种是规范化方法。E-R方法试图提供一种数据项的分类方法，使数据库管理员能够直观地识别出数据类别对象的不同类型（实体、弱实体、属性、关系等等），从而将列出的数据项及它们的关系分类。在创建表现这些对象的E-R图之后，数据库管理员可以通过一个直观的过程将设计转换成为数据库系统的关系表和完整性约束。规范化方法看起来和E-R方法完全不同，而且也许较少依赖直觉：列出所有数据项，然后标识出所有的相互关系规则（是可识别的，称为依赖）。设计开始时，假设所有的数据项被放在一个大的表中，然后把这个表分裂成许多较小的表。在最终生成的表集合中检索原始数据需要做连接操作。只有当数据库管理员对现实世界数据关系以及关系最终被模型化的方法有了丰富的直觉的认识后，E-R方法和标准化方法才能被最好地运用。这两种方法趋向于生成相同的关系表设计。事实上它们相互提供对方所需的直觉，使对方进行得更好。我们不准备区

别两种方法中那种方法更适用。

逻辑数据库设计的一个主要特征是它强调数据项相互关系的规则。没有经验的用户经常把一个关系表看做是由描述性的列的集合组成，列和列非常相像。但这并不准确，因为还有很多规则限制着列中值之间的可能关系。回忆在2.2节中，我们指出在customers表作为一个关系，是四个域的笛卡儿积的一个子集， $CP = CID \times CNAME \times CITY \times DISCNT$ 。然而，我们也指出在任何合法的customers表中，不能有两行的cid列具有相同的值。这是因为在图2-1的定义中cid被描述为customers中行的唯一标识。这里有一个非常好的例子来说明我们希望在逻辑数据库设计中考虑的这种规则。一个如实反映现实世界的表的表示通过指定cid列是customers表的候选键或主键来强加这一要求。回想一下，候选键是表中一个指定的列集合，以使表中任意两行在所有这些列上的值不相同，并且不存在该关键列的一个更小子集（真子集）具有这一性质。主键是被数据库管理员选用的在从其他表引用该表时可以唯一标识表中行的一个候选键。

在计算机数据库中对候选键或主键的如实表述可以在用SQL的Create Table语句建立一个表时提供，我们将在第6章中给出这个语句更完整的语法，但现在可以先看一看在图6-1的声明中给出的语法以对它的语法有一个初步的印象。

```
create table customers (cid char(4) not null, ssn integer not null unique,
    cname varchar(13), city varchar(20), discnt real, primary key (cid));
```

图6-1 表customers的SQL声明，主键cid，候选键ssn

在Create Table语句中ssn列被声明为非空且唯一，就是说在customers表的任何被允许的内容中，不能有两行的ssn列含有相同的值，所以它是一个候选键。在Create Table语句中把cid声明为主键使得cid成为customers表中行的标识，这个标识可以被其他表使用。按照图6-1的表定义，将来任何使customers表的两行在cid列或ssn列具有相同值的SQL的Insert或Update语句都是不合法的并且无效。由此可见，表的键的正确描述是由数据库系统维护的。另外，Create Table语句的许多其他子句起着限制表内容的作用，我们称之为表的完整性约束。必须深刻理解关系表中列之间的相互关系，才能正确理解约束。虽然不是所有逻辑设计的概念都可以用今天的SQL来确切表述，但是SQL正向着使越来越多这些概念模式化的方向发展。无论如何，逻辑设计的许多观点作为系统数据库设计的辅助工具是非常有用的，即使没有直接的系统支持也是如此。

在以下各节中，我们首先介绍一些E-R模型中的定义。在有了些E-R的直观认识后，将介绍规范化进程。

6.1 E-R概念介绍

实体-关系方法试图定义许多数据分类对象；然后数据库设计人员就可以通过直观的识别将数据项归类到已知的类别中去。在本节中，我们将介绍三种基本的数据分类对象：实体、属性和关系。

1. 实体、属性和简单E-R图

我们从实体这一概念的定义开始。

定义6.1.1 实体 (entity) 实体就是具有公共性质的可区别的现实世界对象的集合。 ■

例如，在大学的注册数据库中我们可能有以下的实体：Students, Instructors, Class_rooms, Courses, Course_sections, Class_periods等等。（注意实体名字的首字母大写。）很明显，大学的教室组成的集合符合实体的定义：实体Class_rooms中的教室是可区别的（通过位置，即房间号码），同时还具有其他一些公共的属性，如座位数目（并不是说具有相同的值，而是一个公共的性质）。Class_periods是一个有点令人惊奇的实体——“下午2: 00到3: 00”是一个现实世界的对象吗？——但是，这里的登记过程把这些课程时间段当作对象一般对待，在学生时间表中分配时间段就像分配教室一样。在我们已经作了很多工作的CAP数据库中，可以给出以下实体的例子：Customers, Agents和Products（Orders也是实体，但在这里可能会引起混淆，所以我们将在稍候讨论它）。这里我们有一种感觉就是实体将被映射成关系表。一个实体，如Customers，通常会被映射成一个准确的表，表的每一行对应于一个可区别的现实世界对象（这些对象组成了实体），称为实体实例（entity instance），有时，也称为实体事件（entity occurrence）。

注意，我们通过一些性质区别不同的实体实例，对比列的值来区别关系表中的行，但是我们还没有给这些性质取一个名字。现在，简单地说实体实例是可区别的，就好像我们认为一所大学中的教室是可区别的，而不必知道所使用的房间标牌格式。以后，我们总是以一个大写字母开头来写一个实体名字，但是在SQL中当实体映射为关系表的时候，名字将变成全小写字母形式。

可以看到，我们使用了复数形式的实体名称：Students, Instructors, Class_rooms等等。更为标准的形式是使用单数形式命名实体：Student, Instructor和Class_room。我们使用复数形式是为了强调每个实体实际上代表现实世界的一个对象集合，在其中通常包含多个元素。再来看看我们的复数形式表名称，复数形式是为了强调这些表中通常含有很多的行。在E-R图中实体用矩形框表示，你可以从图6-2中看到这一点。

注意，一些作者使用实体集或者实体类型来表示我们所称的实体，那么对于这些作者而言，实体就是我们所称呼的实体实例。我们也注意到在某个作者的文章中偶尔会出现含糊：有时候表示一个实体，有时候表示一个实体集合。我们假定在一个E-R图中用矩形表示的对象是一个实体——现实世界对象的集合——按照同样方式看待这些矩形的作者与我们有同样的定义方式。这种含糊的存在是不幸的，但我们的概念在后面始终保持一致。

在数学方式的讨论中，为了定义时方便，我们通常用一个大写字母代表一个实体，如果有多个实体存在，可以使用下标。例如，E, E₁, E₂, …。一个实体E由一个现实世界对象的集合构成，我们使用小写字母加下标表示这些对象：E={e₁, e₂, …, e_n}。如上面提到的，实体E的每一个不同实例e_i被称为实体实例或者实体事件。

定义6.1.2 属性 (attribute) 属性是描述实体或者关系（将在下面定义）的性质的数据项。 ■

在实体的定义中说，属于一个实体的所有实体实例具有共同性质。在E-R模型中，这些性质就是属性。我们将看到，E-R模型中的属性和关系模型中的属性或列名并没有什么令人困惑的地方，因为当E-R设计转换成为关系中术语时，它们是相对应的。我们说，实体的一个特定实例具有描述实体的所有属性的属性值（可以是空值）。必须在头脑中记住，当我们列出某个实体E的所有实体实例{e₁, e₂, …, e_n}时，如果引用属性值就无法准确地区别不同的实例。

每个实体有一个标识符，也就是一个属性或者是一个属性集合，每个实体实例在这些属

性上具有不同的值。这类似于关系中的候选键概念。举例而言，我们为实体Customers的标识符定义为客户标识符cid。一个实体可以定义多个标识符，当数据库管理员选定一个单键属性作为整个数据库中实体实例的标识方法时，它就被称为实体的主标识符。其他属性，比如Customers的city属性，不是标识符而是描述性属性，被称为描述符。如同我们在关系模型中看到的，大多数属性的值是取自某个域的简单值，但是复合属性是一组共同描述一个性质的简单属性。举例来说，实体Students的属性student_names可以由简单属性lname、fname以及midinitial组成。注意，实体的标识符可以包含复合属性。最后，我们定义多值属性，它是在一个实体实例中可以取多个值的属性。例如，实体Employees可以附加一个称为hobbies的多值属性，取为雇员列出的自己的多个业余爱好或兴趣。一个雇员可以有多个业余爱好，所以这是一个多值属性。

前面提到，E-R图用矩形来代表实体。图6-2显示了两个简单的E-R图。简单的单值属性用椭圆形表示，并用直线连接到实体。复合属性同样用椭圆形表示并连接到实体，同时组成复合属性的简单属性连接到复合属性上。多值属性用双线连接到它所描述的实体上，而不是用单线。主标识符属性加下划线表示。

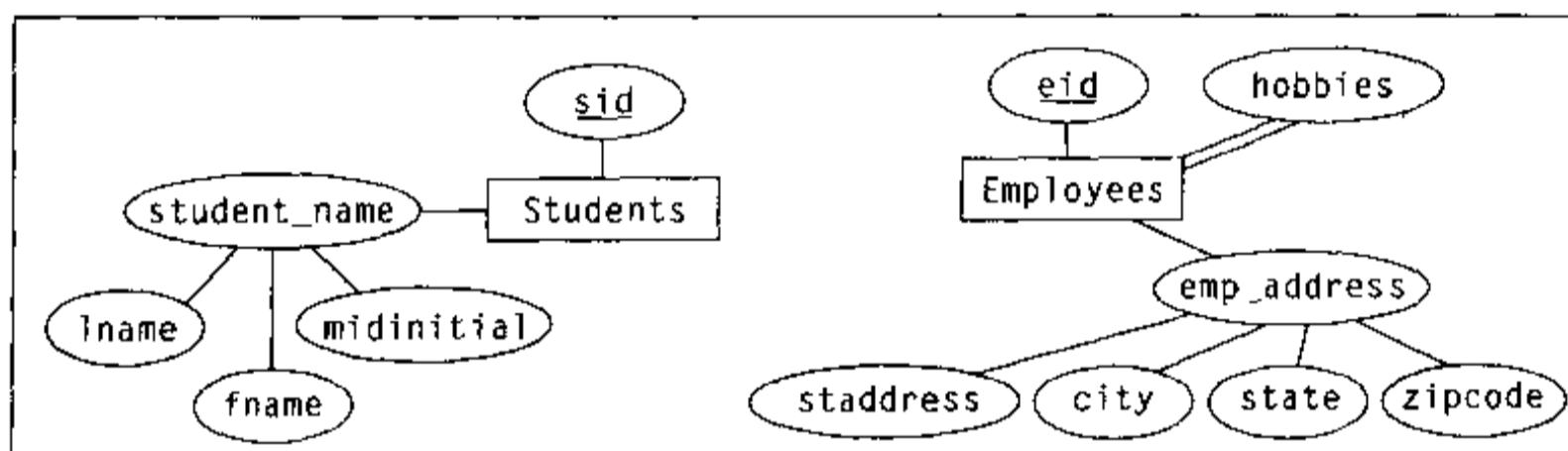


图6-2 具有实体和属性的E-R图例

2. 转换实体和属性到关系

我们的最终目标是从E-R设计转换到一组数据库中关系表的定义。我们通过一些转换规则实现这一目标。

转换规则1 E-R图中的每一个实体映射到关系数据库中的一个表，并用实体名来命名这个表。表的列代表了连接到实体的所有简单单值属性（可能是通过复合属性连接到实体的，但复合属性本身并不变成表的列）。实体的标识符映射为该表的候选键，这将在例6.1.1中说明，实体的主标识符映射为主键。注意到实体的主标识符可以是一个复合属性，所以它将变成为关系表中的一个属性集合。实体实例映射为该表中的行。 ■

例6.1.1 这里有两个表，是从图6-2中E-R图的实体Students和Employees映射而来的，作为例子，我们在每个表中填写了一行。主键加下划线表示。

转换规则2 给定一个实体E，主标识是p。一个多值属性a在E-R图中连接到E，那么a映射成自身的一个表，该表按照复数形式的多值属性名命名。这个新表的列用p和a命名（p或a都可能由几个属性组成），表的行对应(p, a)值对，表示与E的实体实例关联的a的属性值对。这个表的主键属性是p和a中列的集合。

例6.1.2 这里是一个数据库例子，两个表反映了图6-2中实体Employees以及与之连接的多值属性hobbies的E-R图。

employees					hobbies	
<u>eid</u>	staddress	city	state	zipcode	<u>eid</u>	<u>hobby</u>
197	7 Beacon St	Boston	MA	02102	197	chess
221	19 Brighton St	Boston	MA	02103	197	painting
303	153 Mass Ave	Cambridge	MA	02123	197	science fiction
...	221	reading

3. 实体间联系

定义6.1.3 联系 (relationship) 给定m个实体的有序列表：E₁、E₂、…、E_m（列表中同一个实体可以出现多于一次），一个联系R定义了这些实体实例之间的对应规则。特别地，R代表了一个m元组的集合，它是笛卡儿积E₁ × E₂ × … × E_m的子集。

对应于实体实例(e₁, e₂, …, e_n)的一个元组，其中e_i是定义中有序列表中E_i的一个实例，联系的一个实例被称为联系事件 (relationship occurrence) 或者联系实例 (relationship instance)。定义列表中实体数目m称为联系的度 (degree)。两个实体间的联系称为二元联系 (binary relationship)。例如，我们把Instructors和Course_sections之间的联系定义为二元联系teaches。我们把它的一个实例表述为一个特定的教师教授(teaches)一门特定的课程。另一个联系的例子是works_on, 定义为大公司中实体Employees和Projects之间的联系：(Employees works_on Projects)。

联系也可以有附加的属性。联系works_on可以有属性percent，表示某个雇员的每个工作周中有百分之多少的时间被分配到一项特定的工程（参见图6-3）。注意，如果附加到联系works_on上的这个percent属性改为附加到实体Employees或Projects上的话，它将是多值的。属性percent只有描述一个特定的雇员-工程对的时候才是有意义的，所以它是二元联系works_on的自然属性。

联系一个实体到这个实体自身的二元联系 (E_i × E_i的子集) 叫做环 (ring)，有时也叫做递归联系 (recursive relationship)。例如，实体Employees通过联系manages与其自身联系，我们说一个雇员管理 (manages) 另一个雇员。联系在E-R图中用菱形框表示，并用连接线连接到它们所联系的实体上。至于环的情况，通常在连接线上写上所涉及的实体实例在这个联系中扮演的角色名称。在图6-3中两个角色是manager_of和reports_to。

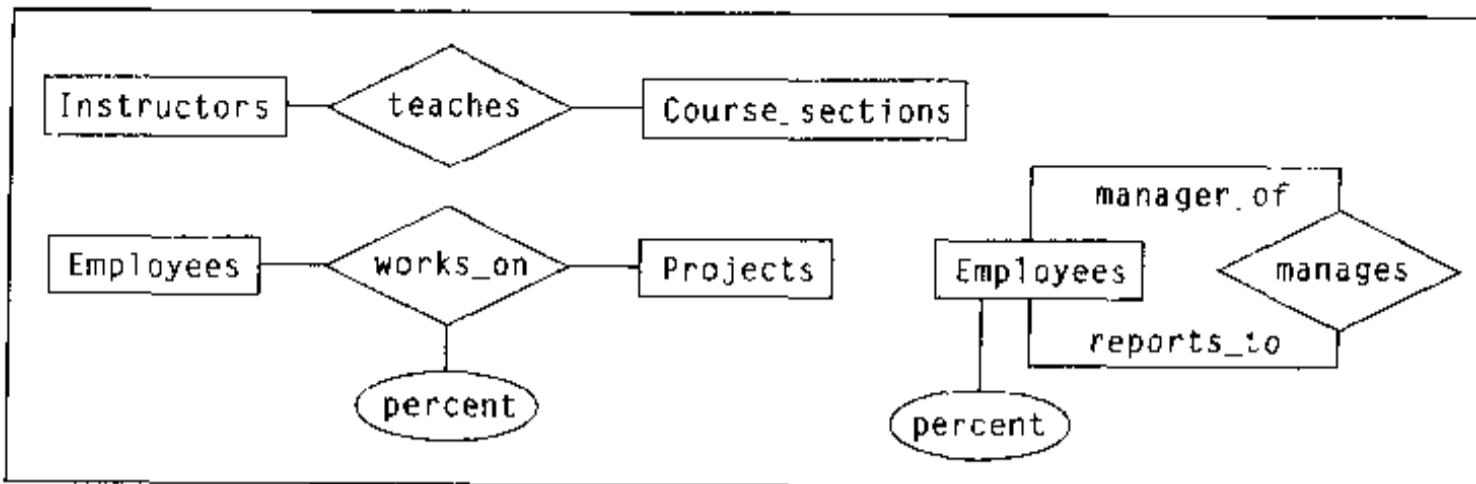


图6-3 具有联系的E-R图例子

注意，我们经常先不考虑E-R图中的属性，以集中精力处理实体间的联系，使我们不被额外的细节分散注意力。

例6.1.3 CAP中的表orders并不代表一个联系 根据定义6.1.3，CAP数据库中的表orders不是Customers、Agents和Products之间的联系。这是因为表orders行中的三元组（cid, aid, pid）不能像要求的那样确定笛卡儿积Customers × Agents × Products的一个子集。相反的，在图2-2的例子中，三元组（cid, aid, pid）的一些值出现不止一次，这显然是设计者的意图，因为同一个顾客可以两次从同一个代理商处订购同一种产品。作为对联系的替代，orders表本身代表了一个实体，ordno是标识符。这非常有意义，因为我们可能因为某种原因不得不查询orders表中的行，却不涉及实体Customers、Agents和Products的实例。例如，我们需要检查是否为先前的一份订单发出了帐单和产品（假设在图2-2的orders表中没有这些属性）。那么，Orders的实例将被按照它们自身表示的对象单独处理。当然，实体Orders与实体Customers、Agents和Products相联系，这将在本章的练习中进一步探讨。 ■

虽然表orders不直接与一个联系对应，但是非常清楚，按照Customers、Agents和Products之间的表orders，我们可以定义任何数目的可能联系

例6.1.4 假设我们要进行一项调查，在调查中需要知道当年所有交易的总额，而这要从customers、agents和products生成的orders表中得到。 例如，我们可能要调查agents和customers之间的交易量关系，也可能调查customers和products之间的交易量关系，以及那些关系是如何被地理因素（city的值）影响的。然而，开始设计时，我们发现不断地由表orders计算总量以获得这个基本数据十分低效，所以我们决定建一个新表yearlies。用下面的SQL命令定义这个新表：

```

create table yearlies (cid char(4), aid char(3), pid char(3),
    totqty integer, totdoll float);
insert into yearlies
    select cid, aid, pid, sum(qty), sum(dollars) from orders
    group by cid, aid, pid;
  
```

一旦我们有了新表yearlies，总额可以在应用逻辑中更新：当新的订单输入后，相关的yearlies行也应当被更新。现在，yearlies表是一个联系，因为表中行的三元组（cid, aid, pid）可以确定笛卡儿积Customers × Agents × Products的一个子集。也就

是说，现在在表yearlies中没有重复的三元组了。既然这些元组是唯一的，那么(*cid*, *aid*, *pid*)便构成了表yearlies的主键。

两个以上实体之间的联系称为N元联系 (*N*-ary relationship)。一个不同实体上的联系叫做三元联系 (ternary relationship)，如联系yearlies。N元联系 (*N*>2) 在E-R图中通常用多个不同的二元联系代替。如果这个替换表达了这个系统中真实的二元联系，那么进行替换是一个好主意。二元联系是多数程序员熟悉的，并且几乎在所有应用中，使用二元联系就足够了。然而，有些时候，三元联系不能够分解为二元联系。例6.1.4中联系yearlies表达了一年的顾客-代理商-产品订购模式——一个不能分解为二元联系的三元联系。我们将在本章后面的习题6.4和习题6.21中进一步探究三元联系。

在将E-R设计转换成关系设计时，联系有时被转换成一个关系表，有时则不是。我们将在下一节对此进一步说明。例如，联系yearlies(一个三元联系)转换成一个关系表yearlies。但是Employees和Employees之间的联系manages，如图6-3所示，并没有转换成为自身的一张表，相反，通常这个联系转换成表employees中的一列，用来指明雇员报告的对象*mgrid*，如同我们在例3.11.6中看到的。在图6-4中我们将这个表重写了一遍。

注意，虽然在表employees中*mgrid*作为一列，但是它并没有被看成实体Employees的一个属性。*mgrid*列是关系模型中所说的外键，它事实上对应于图6-3中E-R图的联系manages。下一节当我们有机会考虑联系的某些性质时，对此做更多讨论。作为这一节的总结，图6-5a和图6-5b列出了到现在为止已经介绍的概念。

employees		
<i>eid</i>	<i>ename</i>	<i>mgrid</i>
e001	Jacqueline	null
e002	Frances	e001
e003	Jose	e001
e004	Deborah	e001
e005	Craig	e002
e006	Mark	e002
e007	Suzanne	e003
e008	Frank	e003
e009	Victor	e004
e010	Chumley	e007

图6-4 表示实体Employees的表，其中包含一个环(递归联系)

概念	描述	例子
实体	现实世界中具有共同性质的可识别对象的集合	Customers, Agents, Products, Employees
属性	描述实体或者联系的某个性质的数据项	见下面
标识符(属性的集合)	唯一地标识一个实体或者联系实例	顾客标识符 <i>cid</i> , 雇员标识符 <i>eid</i>
描述符	非键属性，描述一个实体或者联系	city (Customers中), capacity (Class_rooms中)
复合属性	共同描述一个对象的某个性质的一组简单属性	<i>emp_address</i> (参见图6-2)
多值属性	对于一个实体实例可以取多个值的实体属性	<i>hobbies</i> (参见图6-2)

图6-5a 基本E-R概念：实体和属性

概念	描述	例子
关系	命名的m元组集合，标识笛卡儿积 $E_1 \times E_2 \times \dots \times E_m$ 的子集	
二元关系	两个不同实体上的联系	teaches, works_on (参见图6-3)
环、递归关系	联系一个实体到自身的联系	manages (参见图6-3)
三元关系	三个不同实体上的联系	yearlies (参见例6.1.4)

图6-5b 基本E-R概念：联系

6.2 E-R模型的细节

我们已经定义了一些基本概念，现在让我们来讨论数据库设计E-R方式中联系具有一些性质。

1. 联系中实体的基数

图6-6展示了参与联系的一个实体的最大基数和最小基数概念。图6-6中(a)、(b)和(c)，分别用左边和右边的集合表示实体E和F。当联系R联结两个实体的实例时，就用一条直线连结两个集合中的相应元素。那么，这些连结线本身就代表了联系R的实例。注意，图6-6中的图并非我们所说的E-R图。

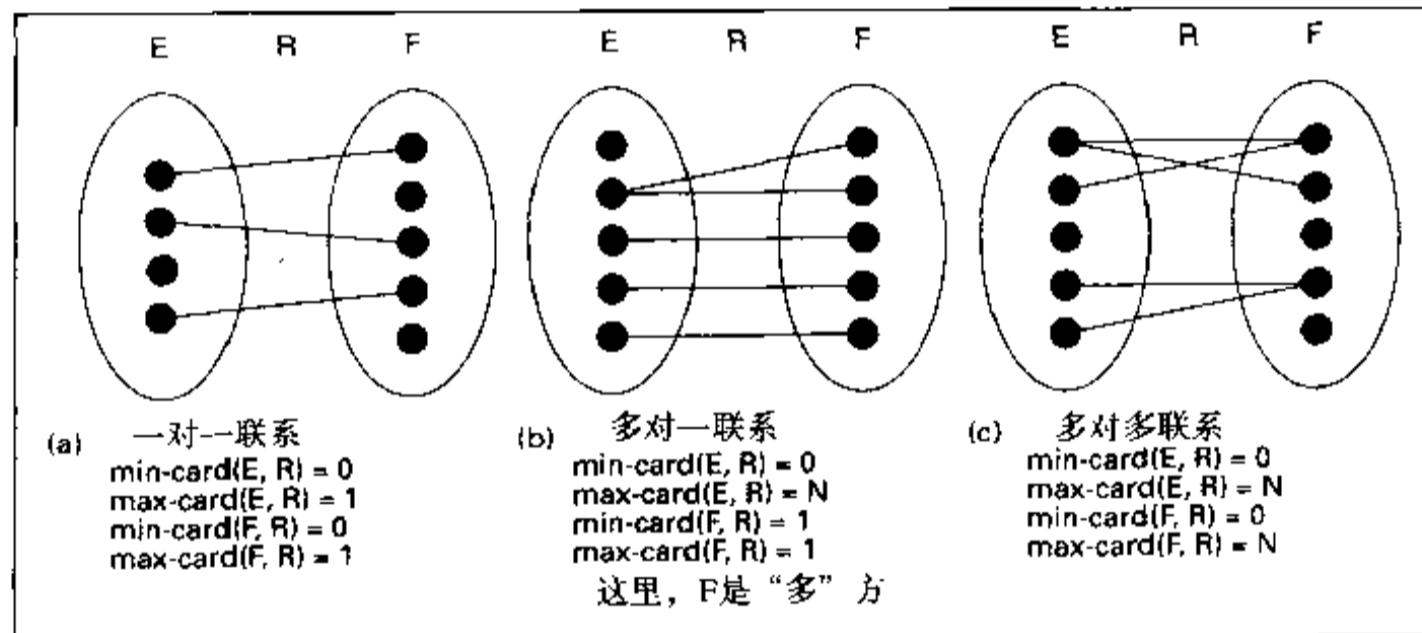


图6-6 两个实体E和F之间联系R的例子

一个实体所参与的联系的最小基数是数据库管理员所允许的可以连结到每个实体实例的最小连结线数目。注意，图6-6只给出了某个特定时刻联系的例子，连接线可能变化，正象一个表中行的内容会变化一样，直到一些实体实例由不同数量的连结线连结。另一方面，一个实体的最小和最大基数性质代表了数据库管理员制订的必须时刻遵守的规则，这些规则不能被影响联系的数据库更新操作所打破。在图6-6的(a)中，可以非常清楚地看到，数据库管理员允许实体E和F在参与联系R时具有为零的最小基数；也就是说，数据库管理员不要求每个实体实例都被连结线连结，因为其中两个集合的一些元素没有被连结线连结，我们用符号表示为 $\text{min-card}(E, R) = 0$ 和 $\text{min-card}(F, R) = 0$ 。然而，E和F在联系R中的最大基数从图6-6的(a)中不能明显地看出来。没有哪个实体实例被一条以上的连结线连结，我们不能保证将来连结线不会改变以使某些实体实例有多条连接线连结到它。然而，为了说明简单，我们将假设图6-6中的(a)确切地表达了数据库管理员想要设定的基数。所以，既然(a)中没有哪个E和F的实体实例具有一条以上的连结线，我们就把这一事实表达为 $\text{max-card}(E, R) = 1$ 和 $\text{max-card}(F, R) = 1$ 。

在图6-6的(b)中，同样假设连接线代表了设计者的意图。因为不是所有的E的元素都有连接线与之连接，所以我们可以写为 $\text{min-card}(E,R)=0$ ；又因为F中的每一个元素都至少有一条连接线与之相连，所以我们写为 $\text{min-card}(F,R)=1$ 。我们的假设意味着这两个值不会改变。我们也有 $\text{max}(E,R)=N$ ，N表示“多于一个”，它表示设计者不想把连接到E的实例的连接线数目限制为1。然而，我们有 $\text{max-card}(F,R)=1$ ，因为F的每一个元素恰有一条线由它发出。现在，最小基数使用了两个值0和1（实际上0根本不是限制，但1表示限制条件“至少有一个”），最大基数使用了两个值1和N（事实上N不是限制，但1表示限制条件“不多于一个”）。我们不讨论“零”、“一”和“多”以外的其他数值。因为 $\text{max-card}(E,R)=N$ ，所以在这个联系中有多个F的实体实例连接到一个E的实体实例。因此，在这个多对一的联系中，F被称为“多”方，而E被称为“一”方。（我们将在定义6.2.2中给出多对一联系的严格定义。）

特别注意 多对一联系中的“多”方是 max-card 值取1的一方。在图6-6的(b)中，实体F就是多对一联系中的“多”方，尽管 $\text{min-card}(F,R)=\text{max-card}(F,R)=1$ 。按照刚才的解释，多对一联系的“一”方是实体实例能够参与多个联系实例的一方，这些实体实例“射出多条连接线”连结到“多”方的多个实体实例！用这种方式解释这个术语很有意义，但是这种方式很容易被忘记，而且忘记它容易导致严重的混淆。

在图6-6的(c)中，我们有 $\text{min-card}(E,R)=0$, $\text{max-card}(F,R)=0$, $\text{max-card}(E,R)=N$, $\text{max-card}(F,R)=N$ 。从定义6.2.2开始，我们将解释用于图6-6中的三个术语的含义：一对联系、多对一联系和多对多联系。

例6.2.1 在图6-3的联系teaches中，教师讲授课程(*Instructors teaches Course_sections*)。数据库管理员可能希望通过写 $\text{min-card}(\text{Course_sections}, \text{teaches})=1$ 设定一条规则使一个课程至少有一个教师教授。我们在制定规则时需要十分小心，因为它意味着在没有决定哪位教师来讲授的情况下，我们不能创建一个新的课程、将它放入数据库、给它分配教室和课时时间段以及允许学生选该门课程。数据库管理员也可以通过写 $\text{max-card}(\text{Course_sections}, \text{teaches})=1$ 制定规则来规定至多分配一个教师来教授一门课程。另一方面，如果允许有一个以上的教师分担一门课程的讲授任务，则数据库管理员可写 $\text{max-card}(\text{Course_sections}, \text{teaches})=N$ 。很明显这是一个重要区别。我们可能并不想制定规则要求每个教师都要教授一些课程(写 $\text{min-card}(\text{Instructors}, \text{teaches})=1$)，因为教师可能休假，所以我们写 $\text{min-card}(\text{Instructors}, \text{teaches})=0$ 。在大多数的大学中，每个学期教师的平均授课数目是大于1的，所以我们写 $\text{max-card}(\text{Instructors}, \text{teaches})=N$ 。 ■

定义6.2.1 一个实体E参与联系R，并且 $\text{min-card}(E,R)=x$ (x 为0或1)， $\text{max-card}(E,R)=y$ (y 为1或N)，那么在E-R图中，E和R之间的连接线可以用有序对(x, y)来标记。我们使用一个新的标记来表示这个最小最大有序对(x, y): $\text{card}(E,R)=(x, y)$ 。 ■

按照定义6.2.1和例6.2.1的分配，连接实体Course_sections到联系teaches的边应该用(1,1)来标记。在图6-7中，我们重复图6-3中的E-R图，并在连接线上附加了有序对(x,y)，以显示所有实体-联系对的最小和最大基数。*Instructors teaches Course_sections*图的基数对遵从例6.2.1中的讨论，其他一些图用合理的基数对标记。我们做出了一系列的决定，达成了下面这些规则：每一个雇员必须为至少一个工程工作（可以为多个工程工作）；一个工程在某些时期可以没有雇员被分配给它（正等待人员分配），当然一些工程将有庞大的

雇员队伍为其工作；一个雇员如果充当manager_of的角色（看下面的讨论），那么在一个特定的时间他可以不管理任何雇员，但仍然被称为管理者；一个雇员向至多一个管理者报告工作，但是可以不向任何人报告（这种可能性是存在的，因为总是存在一个最高层的雇员，没有其他管理者管理他）。

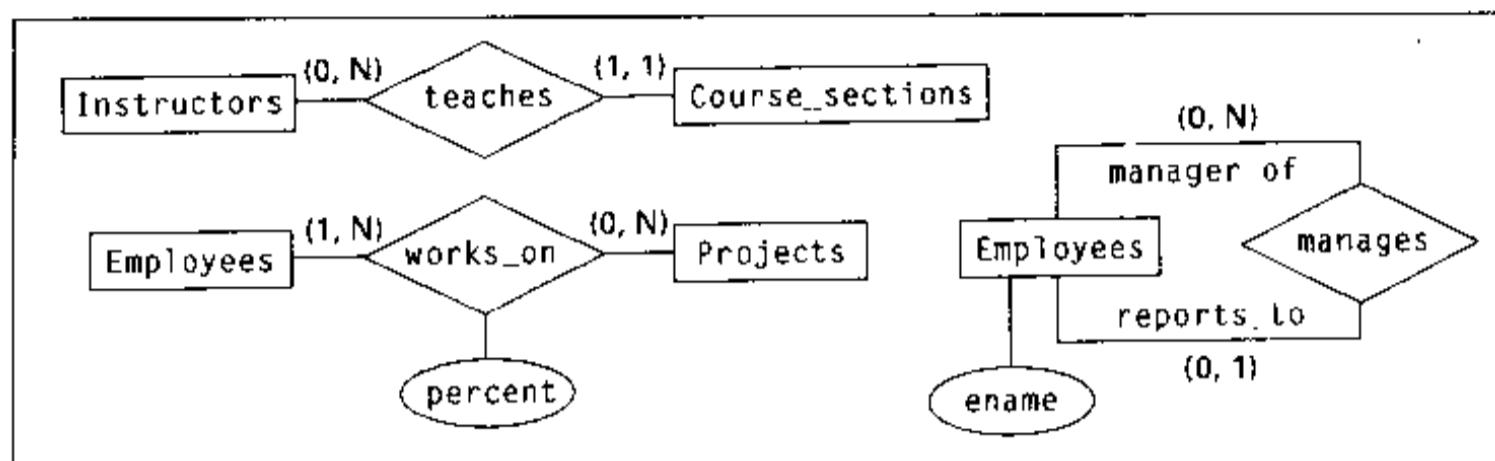


图6-7 E-R图：实体联系间的连接用(x,y)标示

在图6-7的Employees-manages图中，按照正常的标记， $\text{card}(\text{Employees}, \text{manages})$ 将是不确定的。我们说，实体Employees在这个联系中扮演了两种不同的角色：manager_of和reports_to角色。联系manages的每一个实例把一个被管理的雇员（扮演reports_to角色的Employees实例）联系到一个管理者雇员（扮演manager_of角色的Employees实例）。我们在实体后面加上一个括号，其中填写该实体的角色，并使用这种实体的基数概念来消除不确定性。如 $\text{card}(\text{Employees } (\text{reports_to}), \text{manages}) = (0, 1)$ 和 $\text{card}(\text{Employees } (\text{manager_of}), \text{manages}) = (0, N)$ 。

从这些基数中，我们看到一个雇员如果扮演了manager_of的角色，那他可能在某个时间并不管理任何其他雇员，但他仍然被叫做管理者。同样，一个雇员最多向一个管理者做报告，但是可能并不向任何人报告（因为最高层的雇员没有管理他的管理者——如果不是为了这个特殊的雇员，我们可以在Employees-manages边的reports_to分支上标注(1,1)）。

2. 一对一联系、多对多联系和多对一联系

定义6.2.2 当一个实体E在联系R中具有 $\text{max-card}(E, R)=1$ 时，E被称为在联系R中是单值的。如果 $\text{max-card}(E, R)=N$ ，那么E称为在联系中是多值的。如果实体E和F在联系中都是多值的，则E和F之间的二元联系R称为是多对多的或N-N的。如果E和F都是单值的，那么这个联系称为是一对一的或1-1的。如果E是单值的而F是多值的或者相反，那么这个关系称为是多对一的或N-1的（我们通常不区别1-N联系和N-1联系）。■

前面说到在一个多对一联系中的“多”方是具有单值的一方。可以通过考虑图6-7中的联系Instructors teaches Course_sections来更好地理解这一点，这里 $\text{card}(\text{Course_sections}, \text{teaches})=(1,1)$ ，实体Course_sections代表了联系的“多”方。这是因为一个教师可以教授“多”门课程，反之不然。

在定义6.2.2中，我们看到 $\text{max-card}(E, R)$ 和 $\text{max-card}(F, R)$ 决定了一个二元联系是否为多对多、多对一或者一对一。另一方面， $\text{min-card}(E, R)$ 和 $\text{min-card}(F, R)$ 没有被提到，我们说它们与这些特点无关。特别是在图6-6的(b)中 $\text{min-card}(F, R)=1$ ，与图6-6中的(b)代表一个多对一联系的事实无关。如果在F中还有额外的元素没有连接到E中的元素（现有的连接保持不变），这将意味着 $\text{min-card}(F, R)=0$ ，但是这个变化不会影响R是一个多对一联系的事实。我们仍将

看到E中的一个元素（从上数第二个）连接到了F中的两个元素，在这里，实体F是联系的“多”方。

虽然min-card(E,R)和min-card(F,R)并不左右一个联系R是多对多、多对一还是一对一的，但是在联系中，参与关系的另一个实体特点却是由这些量决定的。

定义6.2.3 当一个联系R中的实体E具有 $\text{min-card}(E,R)=1$ 时，E称为强制参与（mandatory participation）R，或者简单地称为在R中是强制的。一个实体E在R中不是强制的，则称为可选的，或者称为具有可选参与（optional participation）。

3. 由二元联系到关系

我们现在准备给出从一个二元多对多联系的转换规则。

转换规则3 N-N联系 当两个实体E和F参与一个多对多二元联系R时，在相关的关系数据库设计中，联系映射成一个表T。这个表包括从实体E和F转化而来的两个表的主键的所有属性，这些列构成了表T的主键。T还包含了连结到联系的所有属性的列。联系实例用表的行表示，相关联的实体实例可以通过这些行的主键值唯一地标识出。

例6.2.2 在图6-7中，works_on是实体Employees和Projects之间的一个多对多联系。图6-8中的关系设计按照转换规则1给出了实体Employees对应的关系表（如同例6.1.2中说明的）和实体Projects对应的关系表；它也遵循转换规则3为联系works_on给出了一个关系表。

employees					works_on		
<u>eid</u>	straddr	city	state	zipcode	<u>eid</u>	<u>prid</u>	percent
197	7 Beacon St	Boston	MA	02102	197	p11	50
221	19 Brighton St	Boston	MA	02103	197	p13	25
303	153 Mass Ave	Cambridge	MA	02123	197	p21	25
...	221	p21	100

projects		
<u>prid</u>	proj_name	due_date
p11	Phoenix	3/31/99
p13	Excelsior	9/31/99
p21	White Mouse	6/30/00
...

图6-8 图6-7中Employees works_on Projects的关系设计

我们一般假设表employees的eid列和表projects的prid列不能取空值，因为它们是各自表的主键，由它们的唯一值来区别所有的行。同样，表works_on中列组成的对(eid,prid)不能取空值，因为每一行必须唯一地标识相关的雇员-工程对。对于此观察，

我们在2.4节末尾的关系规则4（实体完整性规则）中作了总结，它表明一个关系表主键的列不能取空值。虽然我们将它称为实体完整性规则，但是它同样适用于从E-R模型的联系转换而成的关系表。3.2节介绍的SQL Create Table语句提供了在表上强制完整性约束的语法，可以确保该规则不被打破，即不会出现空值。例如SQL语句`create table projects (prid char(3) not null ...)`；确保了projects表的prid列在以后的插入、删除或者更新操作中不会取空值。还有其他一些约束可以起到相同的作用。

转换规则4 N-1联系 当两个实体E和F参与一个多对一的二元联系R时，这个联系在关系数据库设计中不能被映射自身的一个表。相反，如果我们假设实体F具有 $\text{max-card}(F,R)=1$ ，并表示联系中的“多”方，那么从实体F转化成的关系表T中应当包括从实体E转换出的关系表的主键属性列，这被称为T的外键。因为 $\text{max-card}(F,R)=1$ ，T的每一行都通过一个外键值联系到实体E的一个实例。如果F在R中是强制参与的，那么它必须恰恰与E的一个实例相联系，这意味着T的上述外键不能取空值。如果F在R中是选择参与的，那么T中不与E的实例相联系的行在外键所有列可以取空值。

例6.2.3 图6-9显示出一个图6-7中的Instructors teaches Course_sections E-R图的关系转换。我们制定过一条规则，表明一个教师可以教授多门课程，但是一门课程只能由一位教师来教授。表course_sections的列insid是一个外键，它将course_sections的实例（行）联系到唯一一个instructors实例（行）。

instructors				course_sections				
insid	lname	office_no	ext	secid	insid	course	room	period
309	O'Neil	S-3-223	78543	120	309	CS240	M-1-213	MW6
123	Bergen	S-3-547	78413	940	309	CS630	M-1-214	MW7:30
113	Smith	S-3-115	78455	453	123	CS632	M-2-614	TTH6
...

图6-9 图6-7中Instructors teaches Course_sections的关系设计

SQL的Create Table命令可以要求一列不能取空值，这就有可能保证如实描述多对一联系中“多”方实体的强制参与。我们可以建立表course_sections，在insid列上不能为空值。我们所说的“如实”是指用户不可能通过一个不加考虑的更新操作毁掉数据，因为SQL不会允许一个course_sections行的insid列有空值。我们将在下一章中看到，SQL也可以强制约束表course_sections中的外键insid的值恰恰是在表instructors的主键列insid中出现过的值。这个约束称为参照完整性（referential integrity）。

不幸的是，在标准SQL中不可能保证多对一联系中“一”方的强制参与，或者多对多联系中任何一方的强制参与。所以在例6.2.3中，没有办法在SQL表定义中给出一个如实描述来确保每一个教师至少教授一门课程。

注意，对于联系的一些E-R转换规则不同的文章中有不同的观点。推荐读物[4]给出了N-1联系转换规则4的等价表述，但是推荐读物[1]提供了一个不同的替代转换，那里如果联系中“多”方实体是可选参与，联系被映射为自身对应的一个表。这样做的原因是为了避免在外键中大量使用空值（例6.2.3 course_sections中的insid），但是使用空值看起来没有什么

问题，所以我们遵循推荐读物[4]中的转换方法。

转换规则5 1-1联系，可选参与 给定两个实体E和F，他们参与一对一二元联系R，二者的参与都是可选的。我们希望将这一情形转换为关系设计。为此，我们首先按照转换规则1建立表S来表示实体E；同样建立表T表示实体F。然后我们向表T中添加一组列（作为外键），这些列在表S构成主键。如果愿意，还可以在表S中加入一组外键列（表T中主键的列）。对于R的任何联系实例，都有唯一一个E的实例联系到唯一一个F的实例——在S和T的对应行中，外键列填写的值引用另一张表相应行，这一联系是R的实例确定的 ■

转换规则6 1-1联系，两边均强制参与 对于一个双方都是强制参与的一对一联系，最好将两个实体对应的两个表合并成为一个表，这样可以避免使用外键。 ■

我们没有给出所有可能的N元联系 ($N > 2$) 的转换规则。通常这些N元联系都转换成自己对应的表。但是如果除去一个实体外其余参与联系的所有实体都带有 $\text{max-card}=1$ ，那么可以在那个具有较大基数的参与实体对应的表中，加入N-1个外键来表示这个联系。

6.3 其他E-R概念

在这一节中，我们介绍E-R模型中其他一些非常有用的概念。

1. 属性的基数

首先，我们注意到 $\text{min-card}/\text{max-card}$ 概念可以用来描述隶属于实体的属性的基数。

定义6.3.1 给定一个实体E和隶属于它的属性A，我们用 $\text{min-card}(A, E)=0$ 来表示属性A是可选的，用 $\text{min-card}(A, E)=1$ 来表示属性A是强制的。一个强制的属性应当与一个列相对应，这个列在实体E对应的表中声明且不能取空值。我们用 $\text{max-card}(A, E)=1$ 来表示属性是单值的， $\text{max-card}(A, E)=N$ 来表示属性是多值的。一个属性A，当 $\text{min-card}(A, E)=x$ 且 $\text{max-card}(A, E)=y$ 时，就写为 $\text{card}(A, E)=(x, y)$ 。 (x, y) 可以用来标记E-R图中属性-实体间的连接，以显示这个属性的基数。 ■

在E-R图中用没有标记的连接线连接的属性如果是描述符属性，则可以假设具有基数(0,1)；如果是标识符属性，则可以假设具有基数(1,1)。图6-10概要地重述了图6-2，并在属性-实体连接线上作了标记。(注意，这些并不是在图6-2中我们所期望的缺省基数，没有标记基数是因为那时还没有介绍这个概念。)

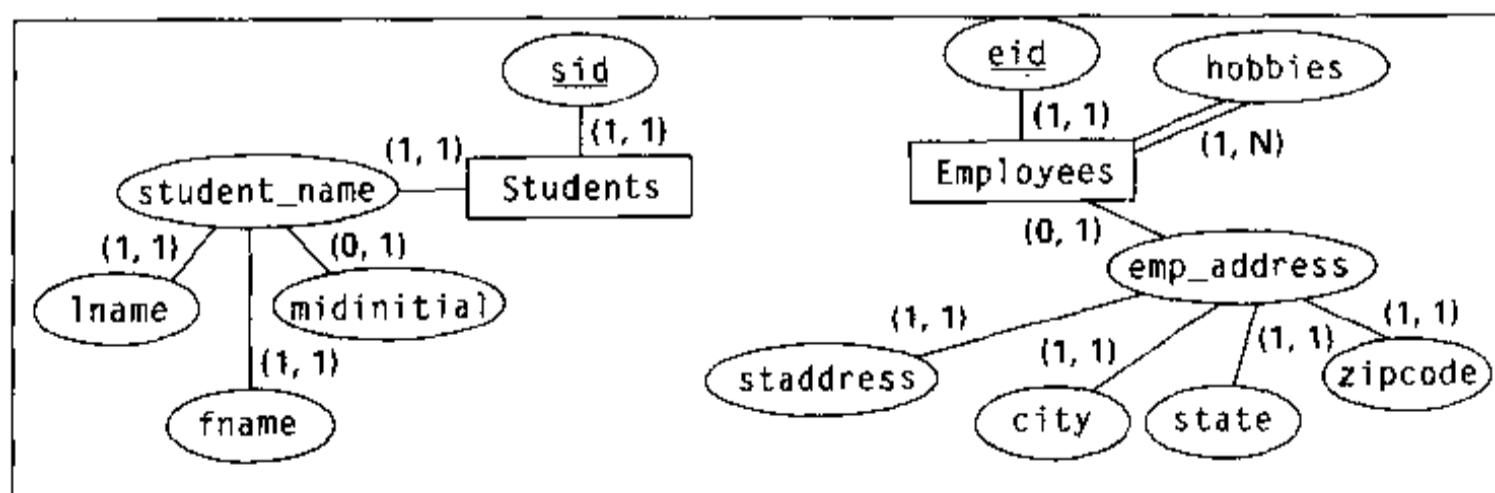


图6-10 标注了属性和实体间连接线的E-R图

在图6-10中，我们看到属性midinitial是可选的（有些人没有中间名）。Students的

复合属性student_names是强制的，但是Employees的属性emp_address是可选的。然而，如果emp_address存在，那么所有组成地址的四个简单属性都是强制的。sid和eid的都具有基数(1,1)；对于实体标识符而言总是如此。多值属性hobbies的max-card值为N，我们可以看到图中它通过一条双线连接到它的实体，min-card(hobbies, Employees)=1，这可能有些令人惊讶，它说明数据库中记录的雇员必须有至少一个业余爱好。

2. 弱实体

定义6.3.2 如果实体的所有实例都通过一个联系R依赖于另一个实体的实例而存在，那么这个实体就称为弱实体，而另一个实体称为强实体。 ■

作为一个例子，我们在CAP设计中已经假设，一个订单详细说明了顾客、代理商、产品、数量和金额。通常的设计变种若允许一次订购多个产品，那么将创建表orders来连接customers和agents的行，同样表line_items包含了每个产品的购买记录；表line_items中的多行对应于一个orders实例。在图6-11中给出了这个设计的E-R模型。

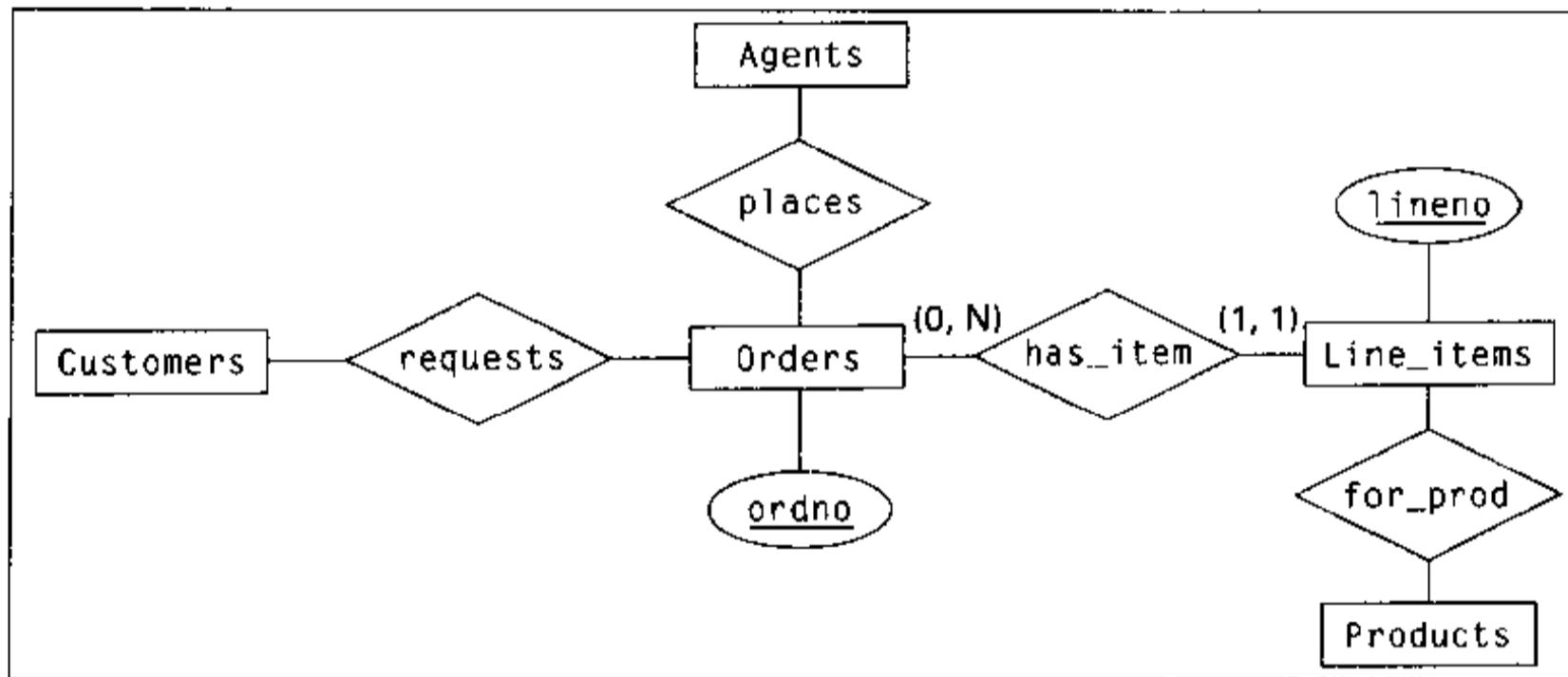


图6-11 一个弱实体Line_items，依赖于实体Orders

如我们所见，实体Orders在与Line_items的联系中是可选的，因为每个订单开始时总是空的。Line_items在联系中是强制的，因为一个订购条目行不可能脱离包含它来为该订单指定顾客和代理商的订单而存在。如果Orders的实例不存在了（顾客取消了它），那么弱实体Line_items的所有实例同样也将消失。弱实体的消失说明Line_items的主标识符（lineno）只有存在于某个订单中时才是有意义的。实际上，这意味着弱实体Line_items的主标识符必须包括实体Orders主标识符中的属性。像Line_items这样的属性叫做外标识符（external identifier）属性。

当弱实体Line_items映射为一个关系表line_items时，按照转换规则4，一个名为ordno的列被加入来表示N-1联系has_item；所以表line_items的主键由外属性ordno和弱实体标识符lineno组成。注意，有时候区分弱实体和多值属性是较困难的。例如，在例6.1.2中的hobbies可以看做一个弱实体Hobbies，标识符是hobby_name。然而，图6-11显然意味着Line_items是一个弱实体而不是一个多值属性，因为Line_items还单独联系到另一个实体Products。

3. 泛化层次

最后, 我们介绍泛化层次(generalization hierarchy)或泛化联系(generalization relationship), 一个对应于对象-关系继承特性(在4.2.3节之前介绍)的E-R概念。其思想是多个有公共属性的实体可以泛化为一个更高层次的超类型实体(supertype entity), 或者相反, 一个一般化实体可以分解成低层次的子类型实体(subtype entity)。目的是让属性隶属于适当的层次, 以避免使用一个每一个实例需要使用大量空值的公共实体的属性。例如, 假设我们把Managers和Non_managers区别为超类型实体Employees的子类型实体(见图6-12)。那么像expenseno(开销报告)这样的属性可以只隶属于实体Managers。而像Union_no这样的非管理员属性可以隶属于实体Non_managers。Consultants可以形成另一个实体类型, 与Employees有许多共同性质, 我们可以创建一个新的超类型实体Persons来包含二者。一个显示泛化层次的E-R图用箭头(不命名)从子类型实体指向超类型实体。

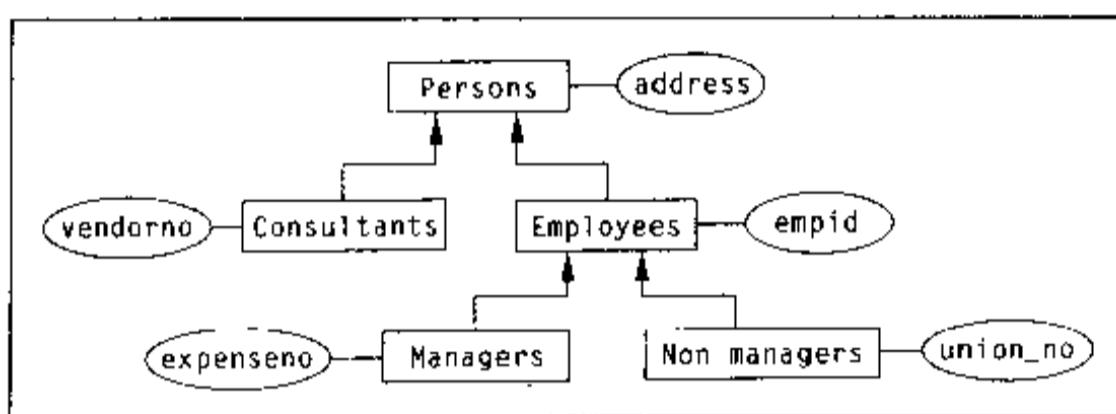


图6-12 附带属性的泛化层次例子

子类型实体和超类型实体之间的箭头联系经常称为ISA联系, 因为顾问(consultant)是一个人(person), 管理人员(manager)是一个雇员(employee), 等等。如同我们在第4章中看到的, 对象-关系数据库系统用类型继承(type inheritance)来表达这些概念, 其中一个子类型对象(行)包含特定属性但从它们的超类型那里继承所有属性。INFORMIX和SQL-99支持对象类型的继承。更多的信息参见4.2.3节。

关系模型没有为泛化层次概念提供支持, 所以有必要将这样一个设计因素重新配置成简单的概念。这个工作可以在转换到关系表之前完成, 也可以作为转换的一部分。在转换成关系表示之前我们给出一种如何在E-R模型阶段实现这一重新配置的方法, 我们每次考虑泛化层次的一层, 给出两个可选择的方法。

1) 我们可以将一个单层的子类型和超类型实体间泛化层次转成单个实体, 这通过把子类型实体的属性加入到超类型实体中来实现。一个附加的属性必须加入到这个单独实体中, 它用来区别不同的类型。作为一个例子, 图6-12中的实体Employees可以被放大, 同时表示管理人员和非管理人员, 这只要在实体Employees中加入属性union_no、expenseno和emptype。现在, 当emptype取值Manager时, 属性union_no将取空值; 类似地, 当emptype取值Non-Manager时, 属性expenseno将取空值。属性emptype也可以指明超类型的情况, 当这个超类型的实例不符合已命名的任何一个子类型时, 这个属性是一个重要的替代。

2) 我们可以保留超类型实体和所有子类型实体作为完全的实体, 并创建显式命名的联系来表示这个ISA联系。

当各种子类型实体和超类型实体的属性差异很大并且被应用逻辑按照不同方式处理时, 第二个方法尤其有用。

这里，我们没有对实体-关系模型的所有概念做完全深入的研究。在本章的末尾列出了全面研究E-R模型和数据库逻辑设计的参考文献，有兴趣的读者可以看一看。

6.4 案例学习

让我们从头实现一个E-R设计，以生成最终的一组关系表为结束。考虑一个简单的机场订票数据库系统，这里只处理从一条航线终点飞出的航班。我们需要明了旅客、航班、登机口和座位分配。现实中的情况可能是任意复杂的，因为航班实际上包括航班乘务组和飞机以及地勤人员，这些要素被组合安排进一张有规律的起飞时刻表中，对应不同的日期，并且分配给这些组合不同的航班编号。为简单起见，我们假设可以用一个实体Flights来代表这些航班，flightno是它的主标识符（具有唯一值，即使在以后的日期中也不会再次出现），depart_time是一个描述属性（实际上由日期和时间构成）；其他一些细节我们都不考虑了。旅客用另一个实体Passengers表示，ticketno是主标识符；旅客的其他属性我们都不关心。同时，我们要明了一次航班的座位情况。假设在一次航班中，一个座位就是一个实体实例，对应的实体是Seats。用座位编号seatno区别不同实例，它只在一个特定的航班中有效（不同的航班可能有不同的飞机座位布局，也就有不同的座位号码集合）。因此，我们把座位分配看做是Passengers和Seats之间的联系，命名为seat-assign。

现在来考虑一下这个需求说明。实体Passengers容易描述，Flights也是。Flights的属性depart_time是一个复合属性，它由简单属性dtime和ddate组成。我们可以加入另一个实体Gates，主标识符是gateno。我们已经定义了一个实体Seats，但它看起来有些奇怪：Seats的主标识符seatno只有与实体Flights的一个实例相联系才有意义。这就是上一节所说的弱实体，因此Flights和Seats之间必须有一个联系，我们定义为has_seat。Seats的标识符中一部分是外部的，其中包含班机的标识符。

我们还有什么其他联系呢？如果我们画出到现在为止已经命名的对象的E-R图，可以注意到实体Gates自身是不工作的。但是非常清楚，旅客要走到一个登机口登机。我们把这个模型成两个二元联系而不是一个三元联系：每个旅客与一个特定的航班通过联系Passengers travels_on Flights联系起来；而登机门通常作为多个航班的编组点（在不同的时间），这通过联系Gates marshals Flights表示。图6-13显示了这个E-R图。从seatno到flightno的箭头表示了Seats的主标识符中包括主实体Flights的标识符。

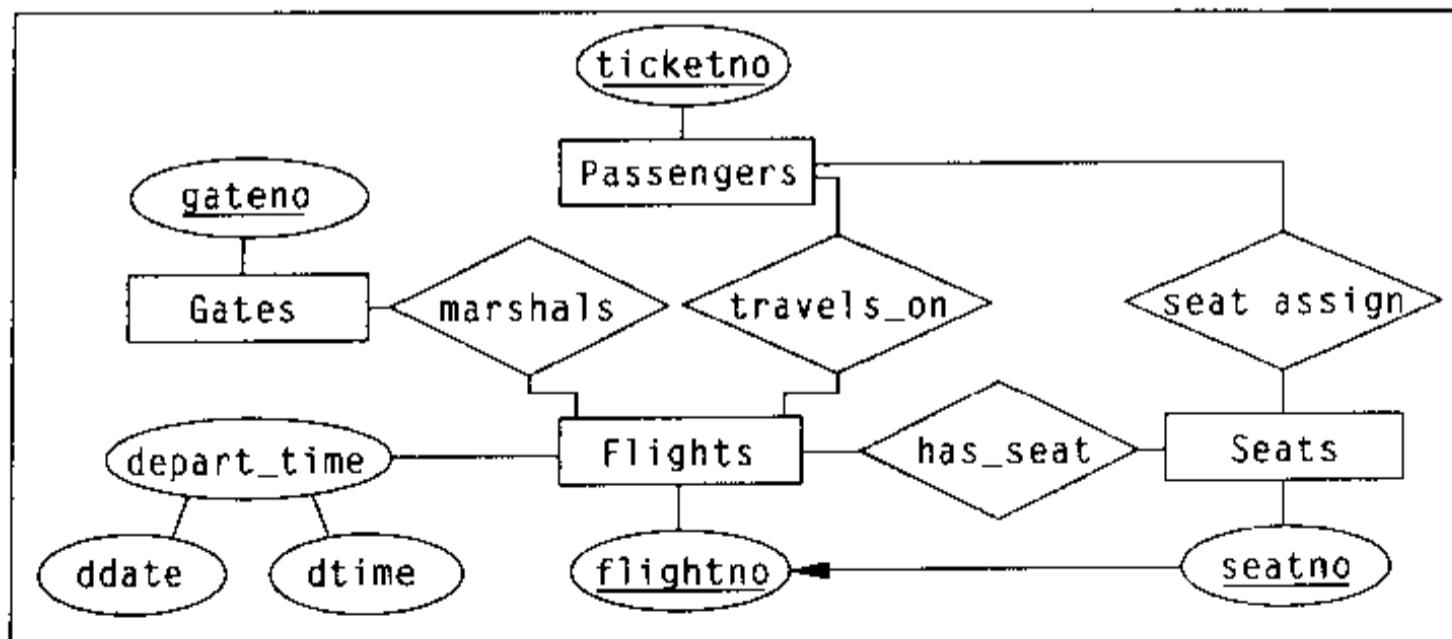


图6-13 简单航空订票系统数据库的早期E-R设计

现在我们需要算出联系中不同参与实体的基数。首先考虑联系marshals，显然，每一个航班只有一个登机门，所以 $\text{card}(\text{Flights}, \text{marshals})=(1,1)$ 。一个登机门可以被多个航班在不同的时间使用，但是没有规定一个登机门必须被使用，所以 $\text{card}(\text{Gates}, \text{marshals})=(0,N)$ 。现在每一个旅客必须只选择一个航班，所以 $\text{card}(\text{Passengers}, \text{travels_on})=(1,1)$ 。一个航班必然有多个旅客才起飞（如果旅客太少，那么这个航班将被取消，登机门被重新分配），但是数据库需要掌握从没有旅客开始的信息，所以我们将最小值设为0，而 $\text{card}(\text{Flights}, \text{travels_on})=(0,N)$ 。一个航班必须有很多的座位提供给旅客，所以 $\text{card}(\text{Flights}, \text{has_seat})=(1,N)$ ；并且每一个座位必须在唯一一个航班中，所以 $\text{card}(\text{Seats}, \text{has_seat})=(1,1)$ 。每一个旅客必须有且只有一个座位，所以 $\text{card}(\text{Passengers}, \text{seat_assign})=(1,1)$ ；座位最多被一个旅客使用但可以空着，所以 $\text{card}(\text{Seats}, \text{seat_assign})=(0,1)$ 。标有这些基数对的E-R图显示在图6-14中。

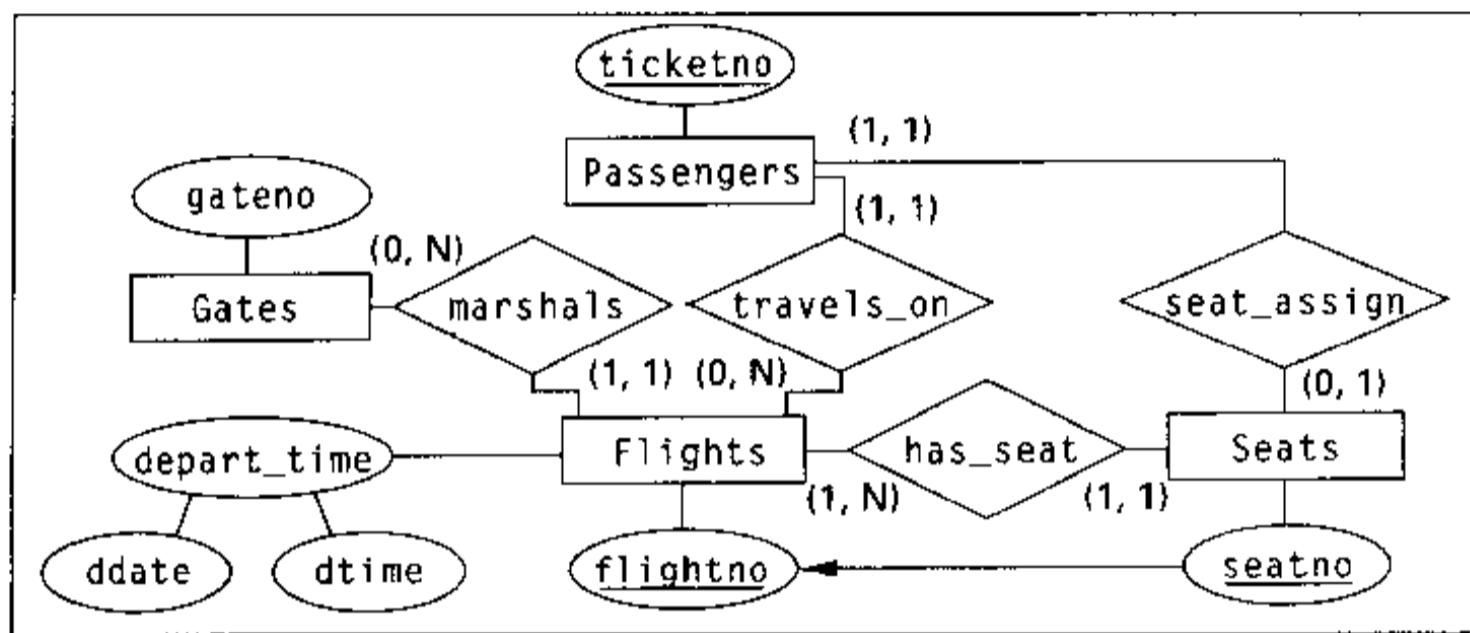


图6-14 带有基数的简单航空订票数据库的E-R图

现在E-R设计完成了，我们需要将它转换成关系表。我们可以从创建由实体映射而成的表开始，尽管这意味着可能忽略一些表示联系外键所需要的属性。在我们考虑联系的时候，再来向这些表中加入属性。我们首先注意到实体Flights在关系表中没有多值属性，所以按照转换规则1，我们加入ddate和dtime列到表flights中。除了表seats，映射其他表都很容易。seats只是简单的一列seatno，尽管这不是该表的一个完整的键。所创建的表如下：

passengers	gates	flights	seats
<u>ticketno</u>	<u>gateno</u>	<u>flightno</u> ddate dtime	<u>seatno</u>
...			

现在考虑联系has_seat，在图6-14中它是N-1的，Seats是“多”方。按照转换规则4，表seats的一个外键把每一个seats的行联系到适当的flights的行。这构成了表seats的主键，它代表了一个弱实体，所以需要一个外键来标识每一行。

passengers	gates	flights	seats
<u>ticketno</u>	<u>gateno</u>	<u>flightno</u> ddate dtime	<u>seatno</u> <u>flightno</u>
...			

passengers			gates
<u>ticketno</u>	<u>seatno</u>	<u>flightno</u>	<u>gateno</u>
...			

flights			seats	
<u>flightno</u>	<u>ddate</u>	<u>dtime</u>	<u>seatno</u>	<u>flightno</u>

联系seat_assign是1-1的，Seats是可选参与方。按照转换规则5，我们可以通过在表passengers中加入seats的一个外键（需要添加两列）来表示这一点。我们不期望将来为了一个给定座位还要查看对应的旅客，所以没有在表Seats中附加外键。结果表定义如下。

现在考虑联系marshals，这是一个N-1联系，Flights是“多”方，所以按照规则4，表flights中的一个外键gateno将连接每一个flights行到相应的gates行：

passengers			gates
<u>ticketno</u>	<u>seatno</u>	<u>flightno</u>	<u>gateno</u>
...			

flights				seats	
<u>flightno</u>	<u>gateno</u>	<u>ddate</u>	<u>dtime</u>	<u>seatno</u>	<u>flightno</u>

类似地，联系travels_on是N-1的，Passengers是“多”方。所以按照转换规则4表passengers的一个外键flightno将连接每一个passengers行到相应的flights行。然而，这个列已经在passengers表中存在了，所以关系表设计到此大功告成。

6.5 规范化：基础知识

规范化是关系数据库逻辑设计的另一种方法，它和E-R模型不太相似，但是，一个基于规范化的关系设计和一个由E-R设计转换成的关系设计几乎得到相同的结果。而且事实上，两种方法具有互补性。在规范化方法中，设计者从一个将被建模的现实世界的情形出发，列出将成为关系表中列名的候选数据项，以及这些数据项之间关系的规则列表。这样做的目标是将所有这些数据项表示成为符合限制条件的表的属性，这些限制条件与我们所称的范式相关。这些范式定义限制着表可以接受的形式，使它具有所期望的特定性质，并且避免各种各样的异常行为。目前存在一系列的范式定义，它们一个比一个限制得更严格。本章所讲述的范式有第一范式（1NF）、第二范式（2NF）、第三范式（3NF）和Boyce-Codd范式（BCNF）。其他类型的范式，4NF和5NF，没有在本书中详述。

首先，符合第一范式（1NF）的表就是没有多值字段的表，也就是一个遵循2.3节中关系规则1（第一范式规则）的表。我们在第2章和第3章中学习的关系模型和SQL语言把这一规则

作为基本规则，但是第4章中介绍的对象-关系系统在两个方面有所不同：允许复合类型和汇集类型的值出现在列中。以后除非特别指明，我们总假设表是符合1NF的。第二范式（2NF）看起来像是过时的东西，因为没有哪个明智的设计人员会把数据库设计成2NF的，他总是会继续规范化直到符合更严格的3NF。从一个将所有数据项包含在一个表（有时称为全局表）中的初始数据库以及这些数据项的相关性规则出发，可以通过一定的过程创建一个具有多个表的等价数据库，这些表均符合第三范式。（这是我们所说一个3NF数据库的含义——数据库中所有表都具有3NF的形式。）当我们进行到本章结束时，就会发现任何不符合第三范式的表都能够分解为几个不同的表，并具有以下性质：第一，每一个分解出的表都是一个有效的第三范式表，第二，所有分解出的表的联接恰恰包含原始表的信息。从原始全局表分解出的3NF表集合被称为数据库的一个3NF无损分解。

我们还可以在3NF分解中提供第三个性质。当向3NF分解中的一个表添加一行（或者更新一行）时，可能有一个错误的更新打破数据项相关性规则，这些规则是先前数据库设计时输入的一部分。我们希望对插入和更新操作施加一个限制，以使这些错误不会破坏数据。那么，在分解中的第三个重要性质是当对一个表做插入或者更新操作时，通过验证被作用的表中数据项的有效性，可以检查是否破坏了相关性规则，而不必进行表的连接操作。具有刚提到的三个期望性质的3NF分解通常被认为是可以接受的数据库设计。可以看出进一步将3NF分解为更严格的BCNF通常是没有必要的（现实中的很多3NF数据库同时也是BCNF的）。但是如果进行了进一步的分解，那么分解结果中将不再保证第三个性质成立，因此很多数据库设计人员决定使用3NF设计。

在能够恰当地处理这些想法之前，我们需要对规范化方法的细节做更多深入了解。让我们用一个例子开始说明。

1. 一个实际运行中的例子：雇员信息

我们需要用一个例子来澄清以后一些数据库设计涉及的定义。考虑图6-15中列出的数据项，他们代表了一个大公司人事部门必须建模的雇员信息。

以emp_开头的数据项表示我们在E-R方法中所说的实体Employees的属性。存在于图6-15中其他数据项下的实体包括Departments（表示雇员在公司中所属的部门）和Skills（表示各种雇员为进行他们的工作所需具备的技能）。在规范化方法中，我们没有给实体概念命名，而在数据项相关性规则中反映出它，这些相关性规则就是所谓的函数依赖，我们在稍后解释。建立数据项emp_id用来唯一标识雇员。每一个雇员为公司里某个部门工作，以dept_开头的数据项描述了不同的部门，数据项dept_name唯一标识部门，每一个部门通常有唯一一个的经理（也是一个雇员），它的名字在dept_mgrname中给出。最后，我们假设各种雇员各自拥有一些技能，例如打字或者文件编排。以skill_开头的数据项描述了技能，公司根据这些技能来考核雇员、分配工作和划定薪水。数据项skill_id唯一标识了一种技能，该技能还有一个名字skill_name。对于一个拥有特定技能的雇员，skill_date描述了这个技能最后一次被考核的日期，

emp_id
emp_name
emp_phone
dept_name
dept_phone
dept_mgrname
skill_id
skill_name
skill_date
skill_lv1

图6-15 没有规范化的雇员信息数据项

skill_lvl描述了雇员在考核中获得的技能等级。

图6-16提供了一个通用表emp_info，它包含了图6-15中雇员信息的所有数据项。按照第一范式，表中的每一个每一行的每一列中只能有原子值。这造成一个难题，因为每个雇员可能有许多技能。如我们在2.3节中讨论的，设计一个表，其中的行由emp_id唯一决定，并为每一种技能信息设计一列，这样做是不恰当的——因为我们甚至不知道一个雇员技能的最大数目，所以我们不知道需要用多少列来表示所有技能。在单独一个（通用）表中解决这个问题的唯一可行方法是，放弃一个雇员对应唯一一行的方法，而复制雇员信息，在不同的行中将该雇员和不同技能进行配对。

emp_info						
emp_id	emp_name	...	skill_id	skill_name	skill_date	skill_lvl
09112	Jones	...	44	librarian	03-15-99	12
09112	Jones	...	26	PC-admin	06-30-98	10
09112	Jones	...	89	word-proc	01-15-00	12
12231	Smith	...	26	PC-admin	04-15-99	5
12231	Smith	...	39	bookkeeping	07-30-97	7
13597	Brown	...	27	statistics	09-15-99	6
14131	Blake	...	26	PC-admin	05-30-98	9
14131	Blake	...	89	word-proc	09-30-99	10
...

图6-16 单个雇员信息表emp_info，满足第一范式

对于图6-16中emp_info表，数据库设计人员的设计意图是公司中存在的每一个雇员-技能对都在表中占有一行。从这一点可以清楚地看到，不存在某两行在属性emp_id和skill_id上具有相同的值。我们在定义2.4.1中定义表的键时已经介绍了描述这一情形的术语。我们声明表emp_info有一个（候选）键，即属性emp_id skill_id组成的集合。回忆前面表示一个属性集的简单方法：将属性列出，中间用空格分离。通过确认在表中这些属性取的值可以区别任意两行（也就是说，对任何行u和v，要么(emp_id)≠v(emp_id)，要么(skill_id)≠v(skill_id)），我们可以确信这些属性构成了一个键，并且不存在该属性集合的子集具有相同的性质（即能够有两行u和v满足(emp_id)=v(emp_id)，并且有两行r和s也满足(skill_id)=s(skill_id)）。后面，我们假设emp_id skill_id是表emp_info的主键。

图6-16的数据库设计看起来很糟，因为在使用数据控制语句更新这个表时，它存在某些会毁坏数据的异常。

2. 不良数据库设计中的异常

在图6-16中表emp_info可能会出现问题，因为雇员数据在不同的行中重复。按照我们目前已有的经验，使每一个不同的雇员有唯一一行似乎更自然。有什么理由来支持我们的感觉呢？让我们看看应用SQL更新语句时，这个表的行为。

假如某个雇员将获得一个新电话号码，我们就不得不更新很多行（那个雇员的所有包含不同技能的行）以使用统一的方法更改emp_phone值。如果我们只更新一行中的电话号码，使得该雇员的一些行具有和其他行不同的电话号码，那么就可能毁坏数据。这就是通常所说

的更新异常。它是因为数据冗余而产生的，即emp_info中雇员电话号码和其他一些雇员属性在多个行上重复。把它叫做“异常”，暗示着它是更新时的不规则行为。可能这有点极端，因为SQL语言具备一次性更新多行的能力，这只要使用如下的一个查找更新语句就可以完成：

```
update emp_info set emp_phone = :newphone where emp_id = :eidval;
```

事实上，多行将被更新的考虑在这个语法中并不明显，如果每一个emp_id值都只有唯一一个行与之对应的话，将使用相同的查找更新语句。然而，由于在不同行中电话号码的重复，在一个定位更新语句中仍然会发生问题。如果我们从一个为了完全不同的目的创建的游标中提取表emp_info的行。程序可能执行下面的语句来允许用户更正一个无效的电话号码：

```
update emp_info set emp_phone = :newphone
where current of cursor_name;
```

这将成为一个编程错误，因为一个有经验的程序员会意识到改变一个雇员的电话号码需要更新多行。但这仍然是一种在实践中非常容易出现的错误，我们希望能够在表上建立一个约束使这样的错误更新不会发生。看起来提供这一约束的最好方法是重新分配数据项到不同的表中，以消除信息的重复存放。这恰是规范化过程将实现的任务。我们用一个定义总结更新异常，定义中参考了我们对于E-R模型的直观理解。

定义6.5.1 更新异常（Update Anomaly） 如果更改表所对应的某个实体实例或者关系实例的单个属性时，需要将多行更新，那么就说这个表存在更新异常。 ■

另一种问题，称为删除异常，用下面定义反映。

定义6.5.2 删除异常（Delete Anomaly）、插入异常（Insert Anomaly） 如果删除表的某一行来反映某个实体实例或者关系实例消失时，会导致丢失另一个不同实体实例或者关系实例的信息，而这是我们不希望丢失的，那么就说这个表存在删除异常。插入异常是这个问题在插入方面的表现，其中我们无法在缺少另一个实体实例或关系实例的情况下，表示某个实体或者实例的信息。 ■

例如，假设一个雇员拥有的一个技能必须在五年后重新被考核。如果雇员没能够通过技能再测试（skill_date列被更新），那么技能将从emp_info列表中除去（一个自动过程删除具有这一emp_id和skill_id的行）。现在考虑如果具有图6-16中那些列的emp_info表中，某个雇员的技能数目减少为零，会发生的事情：没有任何有关这个雇员的行会剩下来！因为这个删除操作，我们丢失了这个雇员的电话号码和所在部门！这显然是不恰当的设计。定义6.5.2的另一个异常，插入异常，在表emp_info表中存在，因为除非这个雇员已经获得某种技能，否则我们不能输入一个新的雇员到表中，因此雇佣一个实习生是不可能的。很明显，这是删除异常的另一面，在删除异常中，当一个雇员失去他（或她）的最后一种技能时，关于这个雇员的信息便会丢失。

现在跳到对付到现在为止提出的问题的解决方法。我们简单地将表emp_info分解成两个表，表emps和表skills，表的列名在图6-17中列出。注意，表emps对每一个emp_id（emp_id是这个表的键）有唯一的行，同时表skills对每一个emp_id skill_id属性对有唯一的行，这个属性对形成了该表的键。因为与每一个雇员相关的技能有多个，所以加入到表skills中的emp_id列扮演了外键的角色，它将技能联系同employees。当我们计算这两个表的自然连接时，结果恰恰是开始时的表emp_info。（我们将在后面论证这一事实，

但是现在你可以确信这一点。) 现在, 删除异常不再是一个问题。如果我们删除与任何一个雇员的技能相关的所有行, 仅仅会删除表skills中的行; 表emps中仍然包含我们想保留的该雇员的信息, 例如emp_phone、dept_name等等。

在后面的小节中, 我们将学习怎样进行规范化, 来分解表以克服所有的异常。图6-17中的表还没有达到这一要求, 稍后我们将看到在其中还有很多异常存在。在我们能够适当处理基本规范化概念之前, 需要对规范化方法的细节仔细研究。下面, 我们给出关于数据库规范化的一些必要的基本数学概念。因为并不总是能够找到一个现实生活中的实例来阐明所有这些概念, 所以我们请读者耐心一点。到最后, 这些概念的价值就会体现出来。

6.6 函数依赖

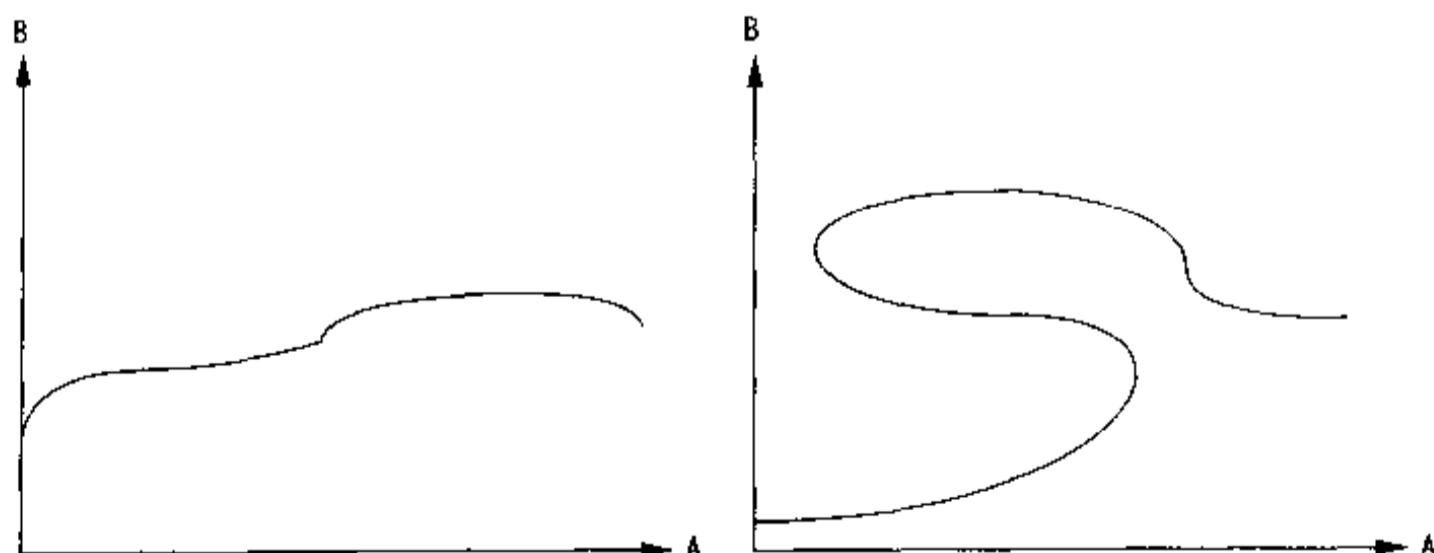
函数依赖 (FD) 定义了数据库系统中数据项之间相关性性质中最常见的类型。我们通常只需考虑单个关系表的属性列之间的相关性, 定义中反映了这一点。我们用 r_1, r_2, \dots 来表示表T的行, 同时按照标准惯例使用表的属性这个术语, 而不是列。一个表的单个属性将用A, B, …来表示, 用字母X, Y, …来表示属性的子集。我们按照第2章中的概念, 用 $r_i(A)$ 表示行 r_i 的属性A的值。

定义6.6.1 给定一个包含至少两个属性A和B的表T, 我们说 $A \rightarrow B$ (读做“ A 函数决定 B ”或者“ B 函数依赖于 A ”), 当且仅当设计者希望对于表T的任何可能的内容(行集合), T中的两行不能在A上取相同值而在B上取不同值。一个更形式化的说法是: 给定T的两行 r_1 和 r_2 , 如果 $r_1(A)=r_2(A)$, 那么 $r_1(B)=r_2(B)$ 。后面一般使用前一种表述。 ■

从数学角度看, 定义6.6.1可以和函数的定义相比: 对于属性A的每一个元素(它将出现在某行中), B中有唯一一个元素(出现在某一行中)与之对应。参见图6-18中函数依赖概念的图示。

emps	skills
emp_id	emp_id
emp_name	skill_id
emp_phone	skill_name
dept_name	skill_date
dept_phone	skill_lv1
dept_mngrname	

图6-17 具有两个表的emp_info数据库



A函数决定B, A的每个值对应B的唯一一个值。

A不函数决定B, A的某些值与B的多个值相对应。

图6-18 函数依赖的图形描述

例6.6.1 在图6-16的表emp_info中，有下面一些函数依赖成立：

$\text{emp_id} \rightarrow \text{emp_name}$
 $\text{emp_id} \rightarrow \text{emp_phone}$
 $\text{emp_id} \rightarrow \text{dept_name}$

在E-R概念中，我们知道这些是成立的，因为 emp_id 是实体Employee的一个标识符，其他数据项简单地表示了实体的其他属性。一旦一个实体被确定，它的所有其他属性也就确定了。但是我们也可以凭直觉辨识这一事实。如果我们看到在图6-16的表emp_info的设计中，有两行具有相同的 emp_id 值和不同的 emp_phone 值，我们就认为这些数据被破坏了（假设每一个雇员有唯一一个电话），但是如果看到两行有相同的 emp_phone 值和不同的 emp_id 值，我们的第一个想法是它们代表了不同的雇员，这两个雇员共用一个电话。但是这两个情形是对称的，这完全是由于我们对于数据的简单理解导致我们认为第一种情形是被破坏的数据。我们指望用 emp_id 打破这一束缚，来唯一地标识雇员。注意我们所说的暗示着 emp_id 函数决定 emp_phone 而 emp_phone 并没有函数决定 emp_id 。有时候用下面形式表示后面这一点：

$\text{emp_phone} \not\rightarrow \text{emp_id}$ ■

例6.6.2 这里用三个表来研究属性间函数依赖（注意，有的表违反了关系规则3，也就是行唯一规则，但是为了演示我们认为它们是有效的表）。在这些表中我们假设设计者的意图是有且仅有这些行出现在表中——不会发生对表的更改。所以我们可以检查数据来发现存在什么函数依赖。通常我们通过体会数据项和企业运作的规则来决定函数依赖（例如，一个雇员只有一个电话号码，雇员们可以共用一个电话，等等），如例6.6.1中所示。这些规则在任何数据被加入到表中之前就已存在。

T1		T2		T3		
行#	A	B	A	B	A	
1	x1	y1	x1	y1	x1	y1
2	x2	y2	x2	y4	x2	y4
3	x3	y1	x1	y1	x1	y1
4	x4	y1	x3	y2	x3	y2
5	x5	y2	x2	y4	x2	y4
6	x6	y2	x4	y3	x4	y4

在表T1中，我们很容易看到 $A \rightarrow B$ ；我们仅仅需要检查每一个行 r_1 和 r_2 组成的对，如果 $r_1(A)=r_2(A)$ 那么 $r_1(B)=r_2(B)$ 。然而，在表T1中没有行对在列A上具有相同的值，所以条件被平凡地满足。同时在表T1中， $B \not\rightarrow A$ （读做：“列B不函数决定列A”），因为，如果 r_1 是行1， r_2 是行3，那么 $r_1(B)=r_2(B)=y1$ ，但是 $r_1(A)=x1 \neq r_2(A)=x3$ 。在表T2中，我们有 $A \rightarrow B$ （我们只需要检查行1和行3，它们具有相同的A值也有相同的B值，同样检查行2和行5）和 $B \rightarrow A$ 。最后，在表T3中， $A \rightarrow B$ 但是 $B \not\rightarrow A$ （如果 r_1 是行2， r_2 是行6，那么 $r_1(B)=r_2(B)=y4$ ，但是 $r_1(A)=x2 \neq r_2(A)=x4$ ）。 ■

如何将函数依赖的定义扩展成完整的一般形式以处理属性集是显而易见的。

定义6.6.2 给定表T和两个属性集，分别为 $X=A_1A_2\cdots A_n$ 和 $Y=B_1B_2\cdots B_m$ ，X的一些属性可以与Y的一些属性重叠。我们说 $X \rightarrow Y$ （读做“X函数决定Y”或者“Y函数依赖于X”），当且仅当设计者的意图是对于表中可能存在的任何行的集合，T中两行不能在X的属性上相同，而同时在Y的属性取不同值。注意到两行在X的属性上相同是指它们在X的所有属性上取相同值，而它们在Y的属性上不相同是指在Y的任意一个属性上取值不同。更为形式化的表述是，给定T的两行 r_1 和 r_2 ，如果 $r_1(A_i)=r_2(A_i)$ 对每一个X中的 A_i 成立，那么 $r_1(B_j)\neq r_2(B_j)$ 对于Y中的每一个 B_j 成立。 ■

例6.6.3 我们在此列出图6-16中为表emp_info定义的所有函数依赖(FD)（没有图6-15中的属性）。根据这个FD列表，规范化过程需要的所有信息就有了。

- (1) $\text{emp_id} \rightarrow \text{emp_name } \text{emp_phone } \text{dept_name}$
- (2) $\text{dept_name} \rightarrow \text{dept_phone } \text{dept_mgrname}$
- (3) $\text{skill_id} \rightarrow \text{skill_name}$
- (4) $\text{emp_id skill_id} \rightarrow \text{skill_date skill_lvl}$

你应当能够解释每一个函数依赖，看看你是否同意它们。例如，FD(1)说明如果我们知道 emp_id ，那么 emp_name 、 emp_phone 和 dept_name 就被确定下来了。注意FD(1)恰恰是例6.6.1中函数依赖的另一种表达方法，也就是说如果我们知道下面的函数依赖成立：

$\text{emp_id} \rightarrow \text{emp_name}$, $\text{emp_id} \rightarrow \text{emp_phone}$, and $\text{emp_id} \rightarrow \text{dept_name}$,

那么就可以说FD(1)成立。

换一种说法，例6.6.1的三个FD共同导出了FD(1)。同样，从FD(1)我们可以得出结论：例6.6.1的三个FD成立。可以使用一个简单的FD推导规则得出这些结论，这个规则是基于函数依赖定义的，我们将在稍后学习更多有关这种规则的知识。

因为FD(1)到(4)全是表emp_info的函数依赖，所以我们可以推断，设计者并不希望skill_name对于特定技能是唯一的。因为skill_id是技能的唯一标识符。如果认为skill_name是唯一的，那么就意味着 $\text{skill_name} \rightarrow \text{skill_id}$ ，即FD(3)的反向。然而，这个FD在集合中并不存在，更不用说它是可以被推导出的。（看它是否可被推导的一个快速测试是，注意skill_name没有在任何集合中的FD的左边出现。）我们也注意到没有函数依赖 $\text{dept_mgrname} \rightarrow \text{dept_name}$ ，这条函数依赖意味着虽然每一个部门有唯一的经理，但是一个经理可以同时管理一个以上的部门。最后注意，skill_lvl和skill_date只有作为实体Employee和Skill之间联系的属性时才是有意义的。如果我们说一个给定的雇员的某个技能的水平是9，那就有必要问：“指的是什么技能？”；如果说知道有一个“打字”的技能水平是9，我们就可能想知道：“指的是哪类雇员？”所以我们需要命名emp_id和skill_id来决定这些属性。 ■

1. 函数依赖的逻辑蕴涵

在例6.6.3中许多结论的得出都依赖于理解函数依赖之间的蕴涵关系。接下来，我们将从定义6.6.2中直接得出一些函数依赖中的蕴涵规则。读者需要在严格的和直觉的两个层次上理解很多这种规则，才能够正确理解稍后章节中介绍的一些规范化技术。我们从一个非常基本的规则开始。

定理6.6.3 包含规则 (Inclusion Rule) 给定表T，其标题是Head(T)（是一个属性集，

如同2.2节中定义的)。如果X和Y是Head(T)中属性的集合，并且 $Y \subseteq X$ ，那么 $X \rightarrow Y$ 。

证 我们运用定义6.6.2。为了说明 $X \rightarrow Y$ ，只需要证明没有两行 u 和 v ，它们在X的属性上取相同值，而同时在Y的属性上取不同值。这是显而易见的，因为两行不可能在X的属性上取相同值而同时在这些属性的某个子集上取不同值。 ■

这个包含规则为我们提供了非常多的函数依赖，这些函数依赖对于任何属性构成的表都是成立的，而不论打算放入其中的内容是什么。

定义6.6.4 平凡依赖 (Trivial Dependency) 表T中的一个平凡依赖是一个形如 $X \rightarrow Y$ 的函数依赖，其中 $X \cap Y \subseteq \text{Head}(T)$ ，且对于表T的任何可能的内容都成立。 ■

我们可以证明平凡依赖是从包含规则产生的。

定理6.6.5 给定表T的一个平凡依赖 $X \rightarrow Y$ ，必然有 $Y \subseteq X$ 。

证 给定表T，其标题行包含 $X \cup Y$ 中的属性，考虑属性集 $Y - X$ (存在于Y而在X中的属性)。因为 $X \rightarrow Y$ 是一个平凡依赖，所以它对表T的任何内容必然都成立。我们假设 $Y - X$ 非空，以推出一个矛盾。如果集合 $Y - X$ 非空，令A是 $Y - X$ 中的一个属性。因为 $A \notin X$ ，那么可以在表T中构造两行 u 和 v ，它们在X的属性上取值相同，但是在A上取值不相同。然而表T有这样两行，函数依赖 $X \rightarrow Y$ 便不再成立，因为行 u 和 v 在X的属性上取相同值而在Y的属性上取不同值($A \in Y$)。既然一个平凡依赖必须对表T的任何内容都成立，那么我们就得到了一个矛盾。从这一点我们可以下结论： $Y - X$ 不含任何属性，即 $Y \subseteq X$ 。 ■

2. 阿姆斯特朗公理

包含规则是蕴涵规则之一，通过它可以产生对于所有可能表都确保成立的函数依赖。看起来从很小一组蕴涵规则可以推导出所有其他的规则。我们在这里列出三个基本规则，并称其为阿姆斯特朗(Armstrong)公理(参见图6-19)。(其他规则集合可以同样简单地给出，见本章末的习题。)

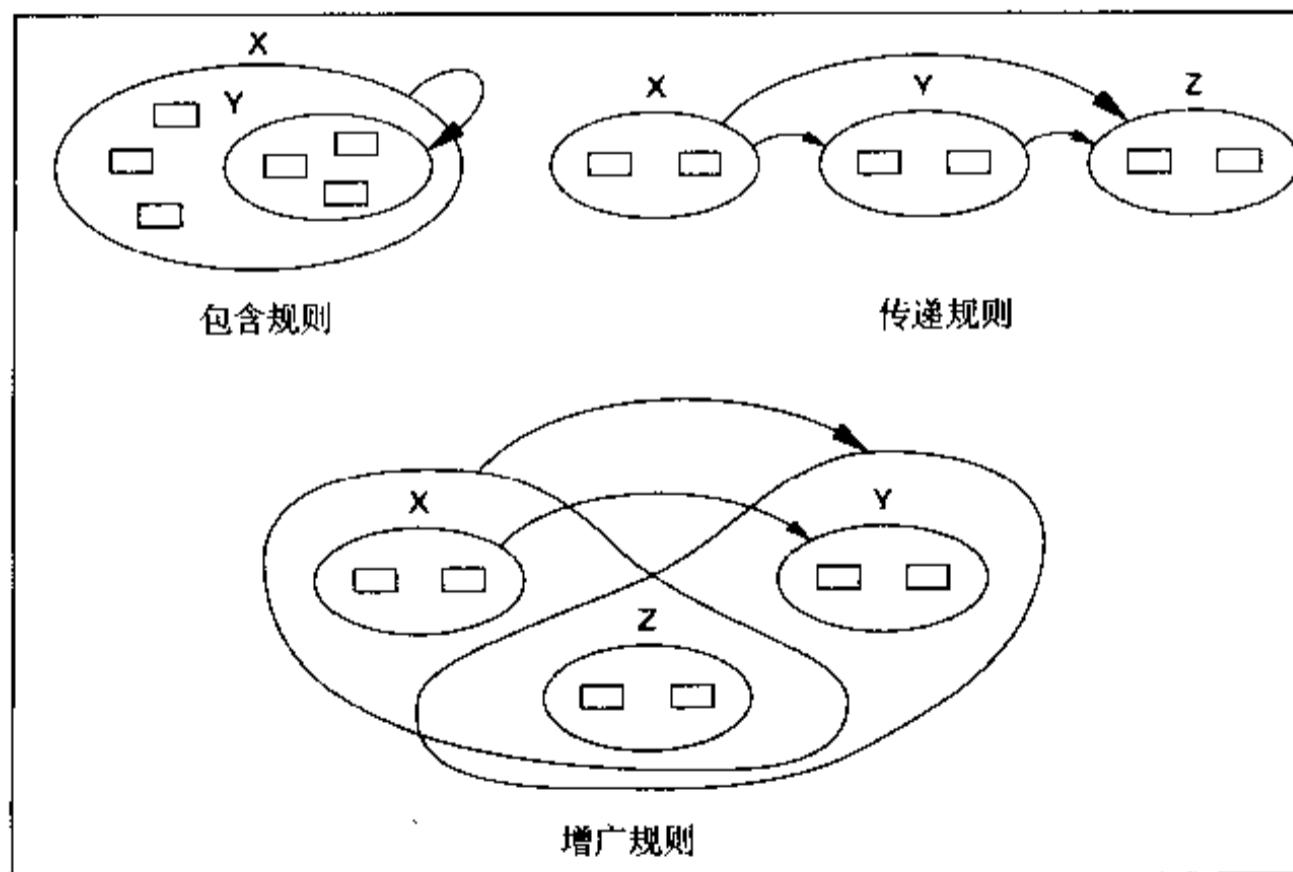


图6-19 阿姆斯特朗公理

定义6.6.6 阿姆斯特朗公理 假设给定一个表T，以及 $\text{Head}(T)$ 的子集X、Y、Z，那么我

们有下面的蕴涵规则：

- 1) 包含规则 (Inclusion rule) : 如果 $Y \subset X$, 那么 $X \rightarrow Y$ 。
- 2) 传递规则 (Transitivity rule) : 如果 $X \rightarrow Y$ 且 $Y \rightarrow Z$, 那么 $X \rightarrow Z$ 。
- 3) 增广规则 (Augmentation rule) : 如果 $X \rightarrow Y$, 那么 $XZ \rightarrow YZ$ 。

像我们在一个函数依赖中列出属性并在它们之间加入空格来表示包含这些属性的集合一样, 两个顺序排列的属性的集合表示一个并运算。所以增广规则可以重写为: 如果 $X \rightarrow Y$, 那么 $X \cup Z \rightarrow Y \cup Z$ 。 ■

我们已经在定理6.6.3中证明了包含规则。让我们来证明增广规则, 并把传递规则的证明作为本章末的习题。

定理6.6.7 增广规则 (Augmentation) 我们的目标是证明如果 $X \rightarrow Y$, 那么 $XZ \rightarrow YZ$ 。假设 $X \rightarrow Y$, 考虑表T的任何两行 u 和 v , 它们在 XZ (也就是 $X \cup Z$) 的属性上值相同。我们仅仅需要证明 u 和 v 在 YZ 的属性上不能取不同值。但是因为 u 和 v 在 XZ 的所有属性上值相同, 所以它们当然在 X 的所有属性上取相同值; 同时因为我们假设 $X \rightarrow Y$, 那么 u 和 v 必须在 Y 的所有属性上值相同。同样, 因为 u 和 v 在 XZ 的属性上值相同, 它们必然在 Z 的所有属性上值相同。因此, u 和 v 在 Y 的所有属性以及 Z 的所有属性上值相同, 证明结束。 ■

从阿姆斯特朗公理, 我们可以证明很多其他函数依赖中的蕴涵规则。进一步讲, 我们可以只使用公理本身而不再使用函数依赖定义来做到这一点。

定理6.6.8 阿姆斯特朗公理的一些蕴涵 我们再一次假设下面所有属性集合 W 、 X 、 Y 和 Z 包含在表T的标题行中。

- [1] 合并规则 (Union rule) : 如果 $X \rightarrow Y$ 且 $X \rightarrow Z$, 那么 $X \rightarrow YZ$ 。
- [2] 分解规则 (Decomposition rule) : 如果 $X \rightarrow YZ$ 那么 $X \rightarrow Y$ 且 $X \rightarrow Z$ 。
- [3] 伪传递规则 (Pseudotransitivity rule) : 如果 $X \rightarrow Y$ 且 $WY \rightarrow Z$, 那么 $XW \rightarrow Z$ 。
- [4] 集合累积规则 (Set accumulation rule) : 如果 $X \rightarrow YZ$ 且 $Z \rightarrow W$, 那么 $X \rightarrow YZW$ 。

证 我们只证明规则2和规则4, 其他留作练习。对于规则2, 注意到 $YZ = Y \cup Z$, 那么根据包含规则 (公理1) 有 $YZ \rightarrow Y$ 。根据传递规则 (公理2), 由 $X \rightarrow YZ$ 和 $YZ \rightarrow Y$ 可以推出 $X \rightarrow Y$ 。同样, 我们可以证明 $X \rightarrow Z$, 至此分解规则2得到证明。对于规则4, 给定 (a) $X \rightarrow YZ$ 和 (b) $Z \rightarrow W$ 。利用公理3, 我们用 YZ 增广 (b) 得到 $YZZ \rightarrow YZW$ 。因为 $ZZ = Z$, 所以我们有 (c) $YZ \rightarrow YZW$ 。最后, 由传递规则, 使用 (a) 和 (c), 我们得到 $X \rightarrow YZW$, 集合累积规则4得到证明。 ■

我们不证明而表述一个非常令人惊奇的结论: 所有函数依赖中有效的蕴涵规则都可以从阿姆斯特朗公理推导出来。事实上, 如果 F 是任何一个函数依赖集合, $X \rightarrow Y$ 是一个不能由阿姆斯特朗公理由 F 推导出的函数依赖, 那么一定有一个表T, 所有 F 中的函数依赖都成立, 但 $X \rightarrow Y$ 不成立。因此, 阿姆斯特朗公理通常被称为是完全的, 意思是没有其他蕴涵规则需要加入其中来增加它们的效力。

回忆例6.6.3中, 我们指出来自例6.6.1中的三个函数依赖:

$\text{emp_id} \rightarrow \text{emp_name}$, $\text{emp_id} \rightarrow \text{emp_phone}$, and $\text{emp_id} \rightarrow \text{dept_name}$

可以得出函数依赖(1)成立的结论:

(1) $\text{emp_id} \rightarrow \text{emp_name emp_phone dept_name}$

两次运用定理6.6.8的合并规则可以得出这个事实。而反方向的蕴涵, 即函数依赖(1)蕴涵

上面三个数据依赖，可以两次运用分解规则得到。无论什么时候有某个属性集X出现在一组函数依赖的左边，我们就可以计算这些函数依赖右边所有属性构成的集合的并，然后将这些函数依赖组合成为一个。例如，假设表T的标题行有属性A B C D E F G，并且我们知道下面这些函数依赖成立：

$$BD \rightarrow A, BD \rightarrow C, BD \rightarrow E, BD \rightarrow F, \text{ and } BD \rightarrow G$$

那么我们只要一次运用合并规则就可以把这些函数依赖组合成一个：

$$\text{[6.6.1]} \quad BD \rightarrow ACEFG$$

事实上，我们可以加入平凡函数依赖 $BD \rightarrow BD$ ，得到

$$BD \rightarrow ABCDEFG$$

如果能够从一组更为基本的函数依赖通过阿姆斯特朗公理推导出一个函数依赖集合，那么我们通常不将推导出的依赖加入进来。我们可能希望回到函数依赖[6.6.1]的形式。注意如果我们有另一个属性H在表T的标题中，但没有在任何函数依赖中被引用，那么除了[6.6.1]中的函数依赖我们可以得到下面的函数依赖：

$$BDH \rightarrow ACEFGH$$

但是因为这个函数依赖可以从函数依赖[6.6.1]通过增广规则推导出来，所以我们更趋向于较短一些的函数依赖[6.6.1]。

例6.6.4 列出下面表T满足的函数依赖的最小集，假设设计人员恰恰允许图中这些行在这个表中存在。我们再一次强调通常并不根据表的内容得出函数依赖。一般，我们通过理解数据项和该企业的规则来决定函数依赖。注意我们还没有一个严格的定义来描述函数依赖的最小集，所以我们只是简单地靠直觉来试图得到最小集。

行#	T			
	A	B	C	D
1	a1	b1	c1	d1
2	a1	b1	c2	d2
3	a2	b1	c1	d3
4	a2	b1	c3	d4

分析：让我们从左边只有一个属性的函数依赖开始考虑。很清楚，我们总是有平凡函数依赖， $A \rightarrow A$, $B \rightarrow B$, $C \rightarrow C$ 和 $D \rightarrow D$ 。但是因为要得到依赖的最小集，所以不打算将它们列出来。从这个特定的表，我们可以得到下面结论：(a) 所有属性B的值相同，因此对任何其他属性P（也就是说P是A、C或者D）都不会有 $r_1(P)=r_2(P)$ 而 $r_1(B) \neq r_2(B)$ 。所以我们得到 $A \rightarrow B$, $C \rightarrow B$ 和 $D \rightarrow B$ 。同时没有其他属性P函数依赖于B。这是因为这些P都至少有两个不同的值，所以总有两行 r_1 和 r_2 满足 $r_1(P) \neq r_2(P)$ 而 $r_1(B)=r_2(B)$ 。所以 $B \nrightarrow A$, $B \nrightarrow C$ 且 $B \nrightarrow D$ 。(b) 因为所有的D值都不相同，所以除了(a)部分的 $D \rightarrow B$ 以外，我们还有 $D \rightarrow A$ 和 $D \rightarrow C$ ；同时因为所有其他属性都至少有两个重复的值，所以D不函数依赖于任何其他属性。因此除了(a)部分的 $B \nrightarrow D$ 外，我们还有 $A \nrightarrow D$ 和 $C \nrightarrow D$ 。(c) 我们有 $A \nrightarrow C$ （因为行1和2）和 $C \nrightarrow A$ （因为行1和3）。我们可以列出所有左边只有一个属性的函数依赖（和非函数依赖）。(括号中的字母表示了它们从上面哪一部分中得来。)

- (a) $A \rightarrow B$ (a) $B \nrightarrow A$ (c) $C \nrightarrow A$ (b) $D \rightarrow A$
- (c) $A \nrightarrow C$ (a) $B \nrightarrow C$ (a) $C \rightarrow B$ (a) $D \rightarrow B$

(b) $A \nrightarrow D$ (a) $B \nrightarrow D$ (b) $C \nrightarrow D$ (b) $D \rightarrow C$

由合并规则，无论何时一个单独出现在左边的属性函数决定几个其他的属性，我们都可以合并右边的属性。如上面的D，可以得到： $D \rightarrow A B C$ 。根据到此为止的分析，我们得出以下函数依赖集（我们认为是最小的）：

(1) $A \rightarrow B$, (2) $C \rightarrow B$, (3) $D \rightarrow A B C$

现在考虑左边有成对属性的函数依赖。(d) 通过上面的函数依赖(3)和增广规则，可知任何包含D的属性对都决定所有其他属性，所以不存在还没有被蕴涵的左边包含D的新函数依赖。(e) 属性B若与任意其他属性P联合出现在左边，那么仍然将仅仅函数决定已经被P决定了的属性，如同我们在下面讨论中看到的。如果 $P \nrightarrow Q$ ，就意味着存在行 r_1 和 r_2 使得 $r_1(Q) \neq r_2(Q)$ 而 $r_1(P)=r_2(P)$ 。但是因为B在所有行上取相同值，并且我们知道 $r_1(BP)=r_2(BP)$ 也成立，所以 $BP \nrightarrow Q$ 。因此我们不会得到左边包含B的新函数依赖。

(f) 现在唯一左边不包含B或者D的属性对是A C。因为A C在每一行上有不同的值（请再一次检查表T），所以我们知道 $A C \rightarrow A B C D$ ，这是新的函数依赖。我们可以通过推理规则说明这一点：根据包含规则， $A C \rightarrow A$ 和 $A C \rightarrow C$ 是平凡的。并且我们已经知道 $A \rightarrow B$ ，所以很容易得到 $A C \rightarrow B$ 。因此我们从 $A C \rightarrow A B C D$ 得到的唯一新函数依赖是 $A C \rightarrow D$ 。因为我们在寻找函数依赖的一个最小集，所以我们在下面列表中把它当作函数依赖(4)加入。如果我们现在考虑左边含有三个属性的函数依赖，我们看到从已经有的函数依赖可以推出：任何三个属性函数决定所有其他属性。任何含有D的三元属性对显然是这样，唯一不含有D的三元属性对是A B C，其中只要A C就可以函数决定所有其他属性。显然，对于左边含有四个属性的情况也是如此。

所以，表T的完整函数依赖集如下：

(1) $A \rightarrow B$, (2) $C \rightarrow B$, (3) $D \rightarrow A B C$, (4) $A C \rightarrow D$

前三个函数依赖来自于先前那个左边只有一个属性的函数依赖列表，而最后一个函数依赖， $A C \rightarrow D$ ，是从左边有两个属性的情况得来的。后面我们将发现尽管为推导最小集花了很多力气，但这个函数依赖集还不是完全最小的。在定义了函数依赖最小集的含义之后，我们就会发现这一点。 ■

3. 闭包、覆盖和最小覆盖

从阿姆斯特朗公理推导出的函数依赖的蕴涵规则意味着无论什么时候给定一个函数依赖集F，都可以推导出一个更大的依赖集。

定义6.6.9 函数依赖集的闭包 给定一个函数依赖集F，它作用在表T的属性上。我们定义F的闭包为可以从F推导出的所有函数依赖的集合，记为 F^+ 。 ■

例6.6.5 考虑给定的函数依赖集合F：

$$F = \{A \rightarrow B, B \rightarrow C, C \rightarrow D, D \rightarrow E, E \rightarrow F, F \rightarrow G, G \rightarrow H\}$$

由传递规则， $A \rightarrow B$ 和 $B \rightarrow C$ 可以推导出 $A \rightarrow C$ ，它一定包含在 F^+ 中。同样， $B \rightarrow C$ 和 $C \rightarrow D$ 推导出 $B \rightarrow D$ 。实际上，如果在序列A B C D E F G H中一个属性出现在最后一个前面，那么它可以通过传递规则来函数决定序列中出现在它右边的每一个属性。我们也有平凡函数依赖如 $A \rightarrow A$ 。下一步，使用合并规则，我们可以生成其他函数依赖，例如 $A \rightarrow A B C D E F G H$ 。事实上，通过在不同的组合中使用合并规则，我们可以得到 $A \rightarrow$ （任何A B C D E F G H的非空子集）。有 $2^8 - 1 = 255$ 个这样的非空子集。上面推导出的所有函数依赖都包含在 F^+ 中。 ■

函数依赖通常在从常理规则建立数据库的时候产生。按照E-R概念的术语，可清楚地知道对应于实体标识符的数据项函数决定了那个实体的所有其他属性。类似地，联系的属性被参与这个联系的实体的标识符唯一决定。在我们的设计中，我们通常期望从一个可操作的函数依赖集F开始。但例6.6.5显示，由F推导出的函数依赖集可能以指数级的速度增长。在下面，我们试图寻找一种方式来说明如果没有这种指数集膨胀，函数依赖集F可以推导出什么。我们的目标是找到一种方法来求出等价于F的函数依赖的一个最小集。我们还将提供在合理时间内推导这个最小集的算法。

定义6.6.10 函数依赖集的覆盖 表T上的两个函数依赖集合F和G，如果函数依赖集G可以从F用蕴涵规则推导出来，或者换句话说，如果 $G \subset F^+$ ，那么我们说F覆盖G。如果F覆盖G且G覆盖F，那么这两个函数依赖集称为等价的，写作 $F \equiv G$ 。 ■

例6.6.6 考虑属性A B C D E组成的集合上的两个函数依赖集：

$$F = \{B \rightarrow CD, AD \rightarrow E, B \rightarrow A\}$$

和

$$G = \{B \rightarrow CDE, B \rightarrow ABC, AD \rightarrow E\}$$

我们将证明F覆盖G，方法是说明G的所有函数依赖可以由F的函数依赖推导出来。下面，我们使用定义6.6.6和定理6.6.8中的各种推导规则来得到F中函数依赖的蕴涵式。因为在F中有(a) $B \rightarrow CD$ 和 (b) $AD \rightarrow E$ ，由合并规则，我们有(c) $B \rightarrow ACD$ ，平凡依赖 $B \rightarrow B$ 显然成立，它和(c) 合并，得到(d) $B \rightarrow ABCD$ 。由分解规则， $B \rightarrow ABCD$ 推导出(e) $B \rightarrow AD$ ，同时因为(f) $AD \rightarrow E$ 在F中，由传递规则我们得到(g) $B \rightarrow E$ 。它再与(d) 合并，得到 $B \rightarrow ABCDE$ 。根据分解规则我们可以推导出G的前两个函数依赖，而第三个已经存在于F中。由此证明F覆盖G。 ■

在例6.6.1中使用了一种技术，在函数依赖集F下找出所有被属性B函数决定的属性。(这里恰是所有属性。)通常我们可以对函数依赖左边的任何属性集合X进行这个操作，找到所有由X函数决定的属性。

定义6.6.11 属性集的闭包 给定表T的函数依赖集F和T中包含的属性集X，我们定义X的闭包 X^+ ，作为由X函数决定的最大属性集合Y，则最大集合Y满足 $X \rightarrow Y$ 存在于 F^+ 中。注意，根据包含规则，集合Y一定包含了X的所有属性，但可以不包含其他任何属性。 ■

这里是一个求任意属性集X的闭包的算法。

算法6.6.12 集合闭包 这个算法求出在给定的函数依赖集F下，一个给定属性集X的闭包 X^+ 。 ■

```

I = 0; X[0] = X;
REPEAT
    I = I + 1;
    X[I] = X[I-1];
    FOR ALL Z → W in F
        IF Z ⊂ X[I]
            THEN X[I] = X[I] ∪ W;
    END FOR
UNTIL X[I] = X[I-1];
    /* integer I, attribute set X[0] */
    /* loop to find larger X[I] */
    /* new I */
    /* initialize new X[I] */
    /* loop on all FDs Z → W in F */
    /* if Z contained in X[I] */
    /* add attributes in W to X[I] */
    /* end loop on FDs */
    /* loop until no new attributes */

```

```
RETURN X+ = X[I];           /* return closure of X */
```

注意，算法6.6.12中将属性加入到X[I]中的那一步是基于集合累积规则的，它已在定理6.6.8中证明：如果 $X \rightarrow Y Z$ 且 $Z \rightarrow W$ ，那么 $X \rightarrow Y Z W$ 。

在算法中，因为 $X \rightarrow X[I]$ （我们的归纳假设）并且在F中由 $Z \subseteq X[I]$ 得到 $Z \rightarrow W$ 后， $X[I]$ 可以用 $Y Z$ ($Y=X[I]-Z$) 来表示。所以我们可以把 $X \rightarrow X[I]$ 写成 $X \rightarrow Y Z$ 。现在既然F包含 $Z \rightarrow W$ ，那么我们可以根据集合累积规则推出 $X \rightarrow Y Z W$ ，或者换句话说， $X \rightarrow X[I] \cup W$ ，同时维持了我们的归纳假设。

集合闭包是我们所进行开发中的一个重要里程碑。它给我们提供了一般方法来确定是否一个给定的函数依赖可以从函数依赖集F推导出来，而不用担心在例6.6.5中所说的计算 F^* 时可能出现的指数级膨胀。例如，假使我们需要知道函数依赖 $X \rightarrow A$ 是否可以由函数依赖集F推导出来，我们只需简单地在F下使用集合闭包算法6.6.12计算出 X^* ，然后看它是否包含A，如果包含，那么 $X \rightarrow A$ 在 F^* 中，即它可以由F推导出来。

我们将看到表的键恰恰是这个表中可以函数决定所有属性的最小属性集合。为判定X是否是一个键，我们只需要在该表的属性的函数依赖集F下计算 X^* ，看它是否包括了表中全部属性，然后确认没有X的某个子集也满足这一点。

例6.6.7 集合闭包和它的一个紧凑推导表示 在例6.6.6中，我们给定一个函数依赖集合F，编号如下：

F: (1) $B \rightarrow C D$, (2) $A D \rightarrow E$, (3) $B \rightarrow A$

给定 $X=B$ ，我们得到 $X^*=A B C D E$ 。我们从 $X[0]=B$ 开始使用算法6.6.12。那么 $X[1]=B$ ，然后我们开始在函数依赖集中循环。因为(1) $B \rightarrow C D$ ，我们得到 $X[1]=B C D$ 。为显示C和D是因为函数依赖(1)而在B之后加入进来的，我们用一种紧凑表示法表示成 $B C D (1)$ 。下一个函数依赖(2) $A D \rightarrow E$ ，在这个时候不起作用，因为 $A D$ 不是 $X[1]$ 的一个子集。下面，由(3) $B \rightarrow A$ ，我们得到 $X[1]=A B C D$ （或者用反映推导顺序的表示法，写为 $B C D (1) A (3)$ ）。现在 $X[0]$ 严格包含在 $X[1]$ 中（也就是说， $X[I-1] \subseteq X[I]$ ），所以 $X[1] \neq X[0]$ 。我们在这次循环中取得了进展，需要继续进行下一次循环，设置 $X[2]=X[1]=A B C D$ （也就是 $B C D (1) A (3)$ ）。再次在函数依赖中循环时，我们看到所有函数依赖都可以被使用（但是我们跳过那些已经使用过的，因为它们不会产生新作用），从仅剩的函数依赖(2) $A D \rightarrow E$ ，得到 $X[2]=A B C D E$ ，或者使用推导表示法，写作 $B C D (2) A (3) E (2)$ 。在这次循环的最后，算法发现 $X[1] \subset X[2]$ 。因为有新进展，所以我们继续计算 $X[3]$ ，在函数依赖中继续循环一次，结束时得到 $X[3]=X[2]$ 。如果注意到所有函数依赖都已经被使用了，就可以省略此次循环。注意，F中函数依赖的排列顺序不同将改变算法执行的细节。在后面练习中，要求给出推导表示法来说明进行了正确的推导，那么顺序是至关重要的。例如，上面的推导产生紧凑表示：

$B C D (1) A (3) E (2)$

而不是

$B C D (1) E (2) A (3)$ 。

给定表T的一个函数依赖集F，我们用下面的算法来计算覆盖F的最小依赖集M。集合M最小是指没有M中的函数依赖可以从整体中删除或者通过去除这个依赖左边的属性来使之改变，而不丢失它覆盖F的性质。

算法6.6.13 最小覆盖 这个算法构造最小函数依赖集M，它覆盖一个给定的函数依赖集

F 。 M 就是 F 的最小覆盖，或者，在有些文章中，称为 F 的规范覆盖（canonical cover）。

第一步 从函数依赖集 F ，我们创建函数依赖一个等价集 H ，它的函数依赖的右边只有单个属性。

```

H = Ø;                                /* initialize H to null set      */
FOR ALL X → Y in F                    /* loop on FDs in F            */
    FOR ALL A IN Y                   /* loop on attributes in Y    */
        H = H ∪ {X → A};           /* add FD to H                */
    END FOR                         /* end loop on attributes in Y */
END FOR                           /* end loop on FDs in F        */

```

因为第一步顺序使用了分解规则来推导出 H ，并且 F 可以从 H 通过顺序的联合规则来重构，所以显然 $F = H$ 。

第二步 从函数依赖集 H ，顺次去掉在 H 中非关键的单个函数依赖。一个函数依赖 $X \rightarrow Y$ 在一个函数依赖集 H 中是非关键的，是指如果 $X \rightarrow Y$ 从 H 中去掉，得到结果 J ，仍然有 $H^+ = J^+$ ，或者 $H = J$ 。就是说，从 H 中去除这个函数依赖对于 H^+ 没有任何影响。图6-20给出了非关键函数依赖的例子。

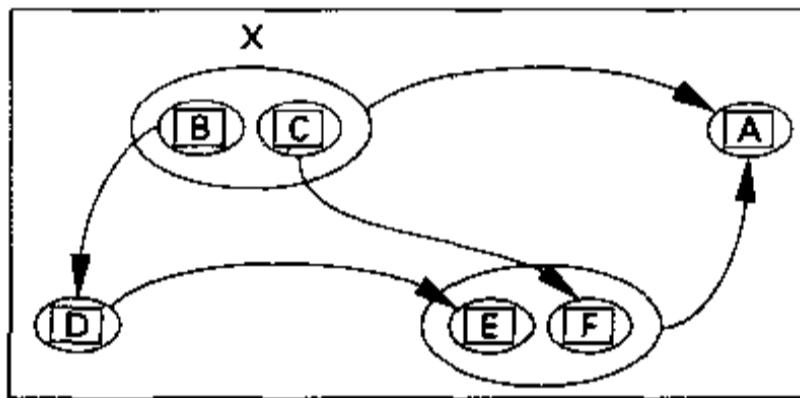


图6-20 非关键函数依赖 $X \rightarrow A$ 的例子

```

FOR ALL X → A in H                  /* loop on FDs in H          */
    J = H - {X → A};                /* try removing this FD      */
    DETERMINE X+ UNDER J;          /* set closure algorithm 6.6.12 */
    IF A ∈ X+                      /* X → A is still implied by J */
        H = H - {X → A};           /* ...so it is inessential in H */
    END FOR                         /* end loop on FDs in H      */

```

每次一个函数依赖按照第二步从 H 中去除时，结果集合等价于先前集合——更大的集合 H 。从这一点可以非常清楚，最终集合 H 是等价于初始集合的。但可能很多函数依赖已经被删除了。

第三步 从函数依赖集 H ，顺次用左边具有更少属性的函数依赖替换原来的函数依赖，只要不会导致 H^+ 改变。图6-21给出了可以按照这种方式简化的函数依赖的例子。

```

H0 = H                                /* save original H          */
FOR ALL X → A in H with #X > 1       /* loop on FDs with multiple attribute lhs */
    FOR ALL B ∈ X                     /* loop on attributes in X   */
        Y = X - {B};                  /* try removing one attribute */
        J = (H - {X → A}) ∪ {Y → A};  /* left-reduced FD          */
        GENERATE Y+ UNDER J, Y+ UNDER H; /* set closure algorithm 6.6.12 */
        IF Y+ UNDER H = Y+ UNDER J    /* if Y+ is unchanged       */
            UPDATE CURRENT X → A in H /* this is X → A in outer loop */
            SET X = Y;                /* change X, continue outer loop */

```

```

    END FOR          /* end loop of attributes in X      */
    END FOR          /* end loop on FDs in H           */
    IF H <> H0        /* if FD set changed in Step 3   */
    REPEAT STEP 2 AND THEN GOTO STEP 4 /* retest: some FDs may be inessential now*/

```

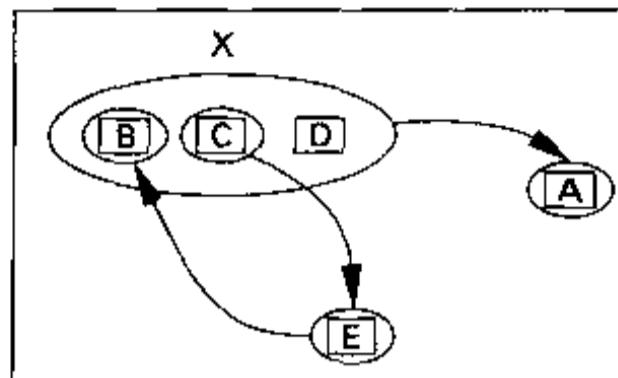


图6-21 一个函数依赖 $X \rightarrow A$ 的例子，其中B可以从左边删除

第四步 从剩下的函数依赖集中收集所有左边相同的函数依赖，使用联合规则创建一个等价函数依赖集M，它的所有函数依赖的左边是唯一的。

```

M = Ø;                      /* initialize M to null set      */
FOR ALL X → A in H          /* loop on FDs in H            */
  IF THIS FD IS FLAGGED, CONTINUE; /* if already dealt with, loop */
  FLAG CURRENT FD;             /* deal with FDs with X on left */
  Y = {A};                     /* start with right-hand side A */
  FOR ALL SUCCESSIVE X → B in H /* nested loop                 */
    FLAG CURRENT FD;           /* deal with all FDs, X on left */
    Y = Y ∪ {B};              /* gather attributes on right   */
  END FOR                      /* gathering complete           */
  M = M ∪ {X → Y};            /* combine right sides of X → ? */
END FOR                      /* end outer loop on FDs in H   */

```

我们不加证明地说明这个算法运算时间只按照n的多项式增长，其中n是F的函数依赖中属性的数目（包含重复）。第三步开销最大，因为我们需要对H中某个函数依赖左边的每一个属性进行一次集合闭包算法。如果我们在第二步之前进行第三步，那么根据上面第三步的最后，我们不需要返回来重复第二步；然而如果没有第二步先来做清除工作，那么费时的第三步将做更多的工作。 ■

例6.6.8 构造函数依赖集F的最小覆盖M，F如下：

F: (1) A B D → A C, (2) C → B E, (3) A D → B F, (4) B → E

注意当你在函数依赖已经改变的地方开始新的一个步骤时，重写函数依赖集很重要，这使你可以在下一步中方便地引用它们。

第一步 我们对F中的函数依赖运用分解规则，来创建一个等价集合，该等价集合的所有函数依赖的右边只有单个属性：

H = (1) A B D → A, (2) A B D → C, (3) C → B, (4) C → E, (5) A D → B, (6) A D → F, (7) B → E.

第二步 我们分别考虑这七个编号函数依赖各自的情形。

(1) A B D → A是平凡的，所以显然不关键（因为A B D*包含了A），它可以被除去。保留在H中的函数依赖是(2) A B D → C, (3) C → B, (4) C → E, (5) A D → B, (6) A D → F和(7) B → E。

(2) $A B D \rightarrow C$ 不能够从H中的其他函数依赖根据集合闭包算法(算法6.6.12)推导出来，因为没有其他函数依赖的右边包含C。 $(A B D^*$ ，除去函数依赖(2)，将不包含C。我们也可以使用算法6.6.12中的步骤来证明这个事实。见下面子步骤(6)。)

(3) $C \rightarrow B$ 是不关键的吗？如果 $\{(2) A B D \rightarrow C, (4) C \rightarrow E, (5) A D \rightarrow B, (6) A D \rightarrow F, (7) B \rightarrow E\}$ 被拿出来，它可以从这些函数依赖推导出来吗？为了知道 $C \rightarrow B$ 是否是关键的，我们在上面这个小一些的函数依赖集下生成 C^* 。(我们使用集合闭包算法6.6.12，在下面生成 X^* ，并使用例6.6.7中介绍的推导表示法。)从 $C^*=C$ 开始，由函数依赖(4)得到 $C^*=C E$ 。为了指明使用了函数依赖(4)，我们记为 $C^*=C E(4)$ 。现在在函数依赖(3)去除以后，没有其他函数依赖的左边包含在集合CE中，所以我们得到了属性C的完全闭包。既然C不包含B，那么(3) $C \rightarrow B$ 是关键的并保留在H中。

(4) $C \rightarrow E$ 是不关键的，这可以通过将函数依赖(4)去掉而计算C的集合闭包看出来：我们得到 $C^*=C B (3) E (7)$ 。所以既然在函数依赖(4)去掉后E在 C^* 中，那么我们可以除去函数依赖(4)。留在H中的函数依赖是(2) $A B D \rightarrow C$, (3) $C \rightarrow B$, (5) $A D \rightarrow B$, (6) $A D \rightarrow F$ 和(7) $B \rightarrow E$ 。

(5) 去掉函数依赖(5)之后剩下的函数依赖集是 $\{(2) A B D \rightarrow C, (3) C \rightarrow B, (6) A D \rightarrow F, (7) B \rightarrow E\}$ ，在这个函数依赖集下， $A D \rightarrow B$ 是不关键的吗？在集合闭包算法中， $A D^*=A D F (6)$ 而没有其他属性。所以函数依赖(5)是关键的，不能被除去。

(6) 除去函数依赖(6)之后的函数依赖集 $\{(2) A B D \rightarrow C, (3) C \rightarrow B, (5) A D \rightarrow B, (7) B \rightarrow E\}$ 下， $A D \rightarrow F$ 是不关键的吗？显然由这个函数依赖集合，我们可以推导出 $A D^*$ 包含 $A D B (3) C (2) E (7)$ ，除了F之外的所有在右边出现的其他属性都被包含了，所以没有函数依赖(6)，我们就无法推导出 $A D \rightarrow F$ 。另一种说法是，如果把函数依赖(6)除去，没有哪个函数依赖右边包含F，所以 $A D \rightarrow F$ 不能被推导出来。

(7) 在除去函数依赖(7)之后的函数依赖集 $\{(2) A B D \rightarrow C, (3) C \rightarrow B, (5) A D \rightarrow B, (6) A D \rightarrow F\}$ 下， $B \rightarrow E$ 是不关键的吗？回答是“并非不关键”，因为由这个函数依赖集推导 B^* 只能够得到B。

我们结束第二步，得到集合 $H=\{(2) A B D \rightarrow C, (3) C \rightarrow B, (5) A D \rightarrow B, (6) A D \rightarrow F, (7) B \rightarrow E\}$ ，为了在第三步中引用方便，我们重新编号如下：

$$H = \{1) A B D \rightarrow C, 2) C \rightarrow B, 3) A D \rightarrow B, 4) A D \rightarrow F, 5) B \rightarrow E\}$$

第三步 我们从函数依赖(1)开始。注意到左边有多个属性；我们把它左边的属性集合记为 $X=A B D$ 。那么我们需要尝试每次去除X中的一个属性来缩减X，并创建一个新的函数依赖集J。

去除A？我们试图从函数依赖(1)中去掉属性A，那么新的集合J有：(1) $B D \rightarrow C$, (2) $C \rightarrow B$, (3) $A D \rightarrow B$, (4) $A D \rightarrow F$, (5) $B \rightarrow E$ 。为了说明缩减产生一个等价函数依赖集合，我们需要说明 $B D^*$ (在H下)与 $B D^*$ (在J下)是相同的。这里的风险在于J下的 $B D^*$ 将比H下 $B D^*$ 函数决定更多的属性，因为J有一个函数依赖，其左边只有 $B D$ ，而H没有。我们声称两个函数依赖集合是等价的，当且仅当 $B D^*(H)$ 和 $B D^*(J)$ 是相同的。所以我们计算H下的 $B D^*$ 得到 $B D E (5)$ ，这是全部的。在J下， $B D^*$ 是 $B D C (1) E (5)$ 。既然它们不相同，那么我们不能够用(1) $B D \rightarrow C$ 来替换(1) $A B D \rightarrow C$ 。

去除B？我们重复这个方法。现在J包含(1) $A D \rightarrow C$, (2) $C \rightarrow B$, (3) $A D \rightarrow B$, (4) $A D \rightarrow F$, (5) $B \rightarrow E$ ，J下的 $A D^*$ 是 $A D C (1) B (2) F (4) E (5)$ 。但是在H下， $A D^*=A D B (3) F (4) E (5) C (1)$ 。

它们是相同的集合，但是产生顺序不相同。你需要使用具有正确顺序的推导表示形式来说明集合闭包算法运用到了正确的函数依赖集上。在H下，函数依赖(3)第一个扩展A D^{*}闭包，在第二轮中函数依赖(1)起作用。在J下，当我们在第一轮中按顺序处理函数依赖时，我们可以使用每一个函数依赖。无论如何，既然H下的A D^{*}和J下的A D^{*}是相同的，那么我们可以缩减函数依赖(1)，使其左边只有A D。函数依赖集H现在成为：

$$H = \{1\} A D \rightarrow C, \{2\} C \rightarrow B, \{3\} A D \rightarrow B, \{4\} A D \rightarrow F, \{5\} B \rightarrow E$$

去除D？ 我们已经考虑过从函数依赖(1) A B D→C的左边去除A，既然B已经除去，那么我们不需要再考虑B。但是我们必须考虑去除D。现在J将包含：(1) A→C, (2) C→B, (3) A D→B, (4) A D→F和(5) B→E。我们需要考虑在H下($A^+ = A$)和J下($A^+ = A C (1) B (2) E (5)$)得到 A^+ 。由于它们不相同，那么我们不能从函数依赖(1)中去除D。

我们注意到函数依赖(2) C→B不能在左边做任何缩减，(3) A D→B同样不能够在左边做任何缩减，因为在H下A^{*}和D^{*}将只包含这些属性，而在对应J下的闭包中包含有B。(4) A D→F不能缩减的讨论与此类似。

现在，因为H的函数依赖集在第二步的执行中被改变，我们需要返回到第二步。在我们到达函数依赖(3)并考虑是否去除(3) A D→B时，我们现在发现在{ (1) A D→C, (2) C→B, (4) A D→F, (5) B→E }下A D^{*}为A D C (1) B (2)。既然A D^{*}在不考虑函数依赖(3)时，包含B，那么这个函数依赖是不关键的，可以去除。（重复第二步非常重要！）第三步完成后的最终结果是：

$$H = \{1\} A D \rightarrow C, \{2\} C \rightarrow B, \{3\} A D \rightarrow F, \{4\} B \rightarrow E$$

这个集合是最小的。如果你愿意，你可以再执行一次第二步和第三步使自己确信不会再有其他改变。

最后，由第四步得到最终的函数依赖集合：

$$H = \{1\} A D \rightarrow C F, \{2\} C \rightarrow B, \{3\} B \rightarrow E$$

■

为了理解我们所取得的结果，你可以做一下例6.6.8，考虑在函数依赖集中做的每一次变动，然后试试使用阿姆斯特朗公理来证明实际进行的每一次变动将导致同样的函数依赖闭包。（不要机械地重复对集合闭包的讨论，请找出直接的证明表明变动是合法的。）

例6.6.9 我们在例6.6.4中推导出的函数依赖看起来不是最小的，尽管我们的目标是在那个例子中创建一个最小的集合。对此的证明留做练习。 ■

例6.6.10 在例6.6.3中为emp_info数据库表述的函数依赖集如下，

- (1) emp_id → emp_name emp_phone dept_name
- (2) dept_name → dept_phone dept_mgrname
- (3) skill_id → skill_name
- (4) emp_id skill_id → skill_date skill_lv1

它们已经构成了一个最小集。就是说，最小覆盖算法6.6.13不会进一步缩减它。我们把这个推导留作练习。 ■

在后面章节中使用规范化方法进行正确设计的算法中，寻找函数依赖集F的一个最小覆盖的算法是至关重要的。

6.7 无损分解

规范化过程依赖于将一个表分解成为两个或者更多较小的表，在这种方法中，我们可以把分解出的表连接起来重新获得到原始表的详细信息。

定义6.7.1 无损分解 对于任何表T以及它的一个函数依赖集F，T的一个分解是一个表的集合 $\{T_1, T_2, \dots, T_k\}$ ，该集合具有两个性质：(1)对于这个集合中的每一个表 T_i ， $\text{Head}(T_i)$ 是 $\text{Head}(T)$ 的一个子集；(2) $\text{Head}(T) = \text{Head}(T_1) \cup \text{Head}(T_2) \cup \dots \cup \text{Head}(T_k)$ 。给定表T的特定内容，即T的行被投影到每个 T_i 的列上作为分解的结果。如果对于表T的任何内容，F中的函数依赖都保证如下关系成立则称表T的一个分解相对于函数依赖集F是一个无损分解，有时候也称为一个无损连接分解即满足：

$$T = T_1 \bowtie T_2 \bowtie \dots \bowtie T_k$$

■

有时候，表T被分解后，不能通过将分解出的表连接起来而恢复原始表的所有信息。这不是因为我们没有得到所有以前存在的行，而是因为得到了原先没有的行。

例6.7.1 一个有损分解 考虑下面表ABC：

ABC		
A	B	C
a1	100	c1
a2	200	c2
a3	300	c3
a4	200	c4

如果我们把它分解成两部分，AB和BC，那么我们得到下面表：

AB		BC	
A	B	B	C
a1	100	100	c1
a2	200	200	c2
a3	300	300	c3
a4	200	200	c4

然而，连接这两个表得到的结果如下：

AB JOIN BC		
A	B	C
a1	100	c1
a2	200	c2
a2	200	c4
a3	300	c3
a4	200	c2
a4	200	c4

这不是ABC表的原始内容！注意，如果处理的表是ABCX，即ABCY和ABCZ中的任意一个（见下面），那么同样的分解表AB和BC将产生与AB JOIN BC内容相同的结果。

ABCY		
A	B	C
a1	100	c1
a2	200	c2
a2	200	c4
a3	300	c3
a4	200	c4

ABCZ		
A	B	C
a1	100	c1
a2	200	c2
a3	300	c3
a4	200	c2
a4	200	c4

因为我们不能够确定开始时的表内容到底是什么，所以信息在这个分解以及其后的连接操作中被丢失了，这称为有损分解，或者有时候称为有损连接分解。 ■

在例6.7.1的分解中，我们丢失信息的原因是，属性B在被分解的表的不同行中具有重复值（200）（在表AB中行a2和a4，在表BC中的行c2和c4）。当这些分解出的表连接时，我们得到交叉结果行在原始表中不存在（或可能不存在）：

a2	200	c4
----	-----	----

和

a4	200	c2
----	-----	----

例6.7.2 表ABC的不同内容 现在让我们从表ABC的一个不同的内容开始再进行一次，此时列B中没有重复值。

ABC		
A	B	C
a1	100	c1
a2	200	c2
a3	300	c3

问题是这样的：如果我们把表ABC分解成为两个表AB和BC，如同例6.7.1中那样，那么分解结果是不是无损的呢？答案是：不是。因为无损分解的定义要求分解出的表的连接能得到原始表的信息，而这应当对原始表将来任何可能的内容成立。但是我们可以通过插入一行到刚才给出的表ABC中而得到例6.7.1的表。似乎没有任何规则可以阻止这种情况的发生 ■

我们需要什么样的规则来限制表ABC将来所有可能的内容，以使分解成表AB和BC是无损的呢？理所当然函数依赖跃入脑海，因为它们确定了控制表将来内容的规则。注意到在定义6.7.1无损分解中，一个函数依赖集F被认为是表T定义的一部分。我们扩展第2章中数据库模式的定义。

定义6.7.2 一个数据库模式是数据库中所有表的标题的集合，以及设计者希望在那些表的连接上成立的所有函数依赖的集合。 ■

例6.7.3 具有一个函数依赖的表ABC 假设表ABC被定义了，它遵循函数依赖B→C。现在例6.7.2中的表内容是完全合法的：

ABC		
A	B	C
a1	100	c1
a2	200	c2
a3	300	c3

但是如果我们试图插入第四行以得到例6.7.1中的表内容，即插入如下行：

a4	200	c4
----	-----	----

这个插入将失败，因为它将破坏函数依赖 $B \rightarrow C$ 。一个新的行在B上取重复值也必然使C具有重复值，以维持 $B \rightarrow C$ 成立：

a4	200	c2
----	-----	----

那么，表ABC的这个新内容一定能够无损地分解然后重新连接吗？答案是肯定的。从下表开始：

ABC		
A	B	C
a1	100	c1
a2	200	c2
a3	300	c3
a4	200	c2

如果我们把它分解成为AB和BC两部分，我们得到下面的内容：

AB		BC	
A	B	B	C
a1	100	100	c1
a2	200	200	c2
a3	300	300	c3
a4	200		

注意到因为有重复值，在表BC中原来的四行被投影成为三行。现在当这两个表再次连接时，可以得到原始表以及它的函数依赖 $B \rightarrow C$ 。 ■

因为例6.7.3表ABC的函数依赖 $B \rightarrow C$ ，ABC在BC上的投影将总是在属性B有唯一值。回忆定义2.4.1，这意味着属性B是表BC的一个键。ABC分解成AB和BC是无损的，原因在于当它们连接时没有交叉会发生：虽然列B的重复值在表AB中会出现，但是表AB的每一行都与表BC的唯一一行连接（假设这个B值在表BC中存在，因为从ABC投影行的初始分解中这总是成立的）。回忆在CAP数据库中，将ORDERS和CUSTOMERS连接时所发生的情况。我们简单地扩展ORDERS的行，加入更多关于顾客的信息。重复值会在表ORDERS的cid列中存在，而cid值在表CUSTOMERS中是唯一的，所以ORDERS中每一行恰恰与CUSTOMERS的一行连接。

我们将上面的讨论一般化，以处理属性集合。

定理6.7.3 给定表T和属性集 $X \subseteq \text{Head}(T)$ ，下面的两个陈述是等价的：(1) X是T的一个超键；(2) $X \rightarrow \text{Head}(T)$ ，即属性集X函数决定了T中所有属性。等价表述为： $X^+ = \text{Head}(T)$ 。

证 (1) 推出 (2)。如果 X 是表 T 的一个超键，那么由定义 2.4.1，对表 T 的任何内容，两个不同的行必须在 X 上不同，即不同的行不能在 X 的所有属性上取相同值。但是由此，很明显两行 u 和 v 不能在 X 上相同而在 $\text{Head}(T)$ 的其他列上不同（因为如果两行在 X 上相同，那么它们必然代表同一行），同时这意味着 $X \rightarrow \text{Head}(T)$ 。(2) 推出 (1)。类似地，如果 $X \rightarrow \text{Head}(T)$ ，那么对表 T 的任何可能内容， T 的两行不能在 X 上具有相同值，而同时在 $\text{Head}(T)$ 的属性上取不同值。但是如果两行 u 和 v 在 $\text{Head}(T)$ 的任何属性都相同，那么根据关系规则 3，它们必须是同一行。这说明两个不同的行不能在 X 上取相同值，所以 X 是 T 的一个超键。 ■

我们已经可以给出这种无损分解的一般规则了，后面在进行规范化时将使用这个规则。

定理 6.7.4 给定表 T 和它的有效函数依赖集 F ，一个把 T 分为两个表 $\{T_1, T_2\}$ 的分解是 T 的一个无损分解，当且仅当 $\text{Head}(T_1)$ 和 $\text{Head}(T_2)$ 都是 $\text{Head}(T)$ 的真子集， $\text{Head}(T) = \text{Head}(T_1) \cup \text{Head}(T_2)$ （也就是说， T 的所有属性在 T_1 或者 T_2 中重复），同时如下函数依赖之一可以通过 F 推导出来：

$$(1) \text{Head}(T_1) \cap \text{Head}(T_2) \rightarrow \text{Head}(T_1)$$

或者

$$(2) \text{Head}(T_1) \cap \text{Head}(T_2) \rightarrow \text{Head}(T_2)。 ■$$

证 我们考虑给定表 T ，它分解成为 T_1 和 T_2 ，以及函数依赖 (1) $\text{Head}(T_1) \cap \text{Head}(T_2) \rightarrow \text{Head}(T_1)$ 。（对函数依赖(2)的证明是相同的。）在后面，我们用 X 表示属性集 $\text{Head}(T_1) \cap (\text{Head}(T_2))$ ； Y 是属性集 $\text{Head}(T_1) - \text{Head}(T_2)$ ， Z 是属性集 $\text{Head}(T_2) - \text{Head}(T_1)$ 。首先，我们分解的定义（定义 6.7.1），即 T_1 和 T_2 是 T 的投影，则 $\text{Head}(T_1) \cup \text{Head}(T_2) = \text{Head}(T)$ 。我们可以证明 $T_1 \subseteq T_1 \bowtie T_2$ 。 T 的每一列都在 $T_1 \bowtie T_2$ 中出现，如果 u 是 T 的一行，我们说 u 在 $\text{Head}(T_1)$ 上的投影是 y_1x_1 ，它是属性值的连接，其中 y_1 代表在 Y 中属性的值，而 x_1 代表在 X 中属性的值；类似地， x_2z_2 是 u 在 $\text{Head}(T_2)$ 上的投影。显然，若 $X = \text{Head}(T_1) \cap \text{Head}(T_2)$ ，那么 u 在 $\text{Head}(T_1)$ 上的投影与在 $\text{Head}(T_2)$ 上的投影在属于 X 的属性上具有相同的值。同时由连接的定义（定义 2.7.4），行 u （一个连接 $y_1x_1z_2$ ）将在 $T_1 \bowtie T_2$ 中出现。

现在我们在假设 $T_1 \bowtie T_2 \subseteq T$ 情况下证明。假设从 T 中的行 u ，我们通过投影得到 T_1 中的一行 y_1x_1 ，同上。类似地，假设从 T 中的行 v ，我们得到 T_2 中的行 x_2z_2 ，其中 x_2 代表 X 中属性的值。现在假设 T_1 和 T_2 中的行 y_1x_1 和 x_2z_2 可以连接，那么 x_1 所有属性与 x_2 的相同，而且 $y_1x_1z_2$ 在 $T_1 \bowtie T_2$ 中，这是 $T_1 \bowtie T_2$ 中行的一般形式，我们只要说明这行也在 T 中。我们用 z_1 表示 T 中行 u 在 y_1x_1 上投影后，剩下的属性值。所以 $u = y_1x_1z_1$ ，同时 $z_1 = z_2$ 。这是因为行 u 在 X 的属性上与 v 的值相同，而 $X \rightarrow \text{Head}(T_2)$ ，所以， $X \rightarrow \text{Head}(T_2) - \text{Head}(T_1) = Z$ 。又因为 u 和 v 在 X 上值相同，所以它们必然在 Z 的属性上取相同值。因此 $z_1 = z_2$ ，行 $y_1x_1z_1$ 在 $T_1 \bowtie T_2$ 中，且与 T 中的行 $y_1x_1z_2$ 相同。

我们将在练习中证明，如果 $\text{Head}(T_1)$ 和 $\text{Head}(T_2)$ 都是 $\text{Head}(T)$ 的真子集， $\text{Head}(T) = \text{Head}(T_1) \cup \text{Head}(T_2)$ ， $\text{Head}(T_1) \cap \text{Head}(T_2)$ 不函数决定 $\text{Head}(T_1)$ 或者 $\text{Head}(T_2)$ ，那么把 T 分解为 T_1 和 T_2 是无损的。 ■

例 6.7.4 在例 6.7.3 中，我们演示了表 T （标题是 $A B C$ ，有函数依赖 $B \rightarrow C$ ）分解成表 T_1 和 T_2 ， $\text{Head}(T_1) = A B$ ， $\text{Head}(T_2) = B C$ 。如果我们运用定理 6.7.4，我们有 $\text{Head}(T_1) \cap \text{Head}(T_2) \rightarrow \text{Head}(T_2)$ ，也就是 $A B \cap B C \rightarrow B C$ ，即 $B \rightarrow B C$ ，这从 $B \rightarrow C$ 是显而易见的。 ■

例 6.7.5 考虑例 2.7.7 中通过 CUSTOMERS 和 ORDERS 的连接得到的表 CUSTORDS。显然，

ordno 是CUSTORDS的一个键，因为它具有唯一值，读者也可以推导出函数依赖 $\text{cid} \rightarrow \text{Head}(\text{CUSTOMERS})$ 。现在我们注意到 $\text{Head}(\text{CUSTOMERS}) \cap \text{Head}(\text{ORDERS}) = \text{cid}$ ，这是CUSTOMERS的键，所以 $\text{Head}(\text{CUSTOMERS}) \cap \text{Head}(\text{ORDERS}) \rightarrow \text{Head}(\text{CUSTOMERS})$ 。根据定理6.7.4，CUSTORDS分解成CUSTS和ORDS是无损的，它们分别与CUSTOMERS和ORDERS具有相同的标题（我们需要验证从CUSTORDS投影到CUSTS和ORDS的行与我们在CUSTOMERS和ORDERS中使用的行相同）。这个分解看起来是直观的，原因在于CUSTOMERS和ORDERS连接后，我们扩展表ORDERS的每一行，扩展的内容来自与这行中唯一 cid 值相关联的customers的列。因此很显然，我们没有在分解成CUSTOMERS和ORDERS的标题时丢失任何信息。当然如果有一些顾客没有任何订单，那么当我们最初创建表CUSTORDS时可能已经丢失了一些信息。但是无损分解是从表CUSTORDS开始的，并确保在分解时没有信息丢失。 ■

定理6.7.4显示了如何证明表T分解成 $\{T_1, T_2\}$ 是一个无损分解。如果分解成三个或更多表， $\{T_1, T_2, \dots, T_k\}$, $k \leq 3$ ，我们可以利用两个表时的结果递归地证明无损性。

例6.7.6 多个表的无损连接分解 假设给定表T, $\text{Head}(T) = A B C D E F$, 以及函数依赖(1) $A B \rightarrow C$, (2) $A \rightarrow D$, (3) $B \rightarrow E$ 。注意没有关于属性F的函数依赖，但是A B构成了A B C D E的一个键，因为它的闭包包含了所有这些属性。因为键必须函数决定 $\text{Head}(T)$ 中的所有内容，所以表T的键一定是A B F。T的一个可以接受的无损分解是 $\{T_1, T_2, T_3, T_4\}$, 其中 $\text{Head}(T_1) = \underline{A} \underline{B} C$ (这些表的键用下划线标注)、 $\text{Head}(T_2) = \underline{A} D$ 、 $\text{Head}(T_3) = \underline{B} E$ 和 $\text{Head}(T_4) = \underline{A} \underline{B} E$ 。这些表的联合包含了T中的所有属性，所以我们仅需要证明无损性。注意，如果我们用下面的顺序一对一对地连接表，则可以根据定理6.7.4保证每一个括起来的表连接与下一个将和它连接的表共同构成一个无损分解。

$$(T_1 \bowtie T_2) \bowtie T_3 \bowtie T_4$$

我们注意到 $\text{Head}(T_1) = \underline{A} \underline{B} C$, $\text{Head}(T_2) = \underline{A} D$, $\text{Head}(T_1 \bowtie T_2) = \underline{A} B C D$, $\text{Head}(T_3) = \underline{B} E$, $\text{Head}((T_1 \bowtie T_2) \bowtie T_3) = \underline{A} B C D E$, 所以下面的函数依赖保证多个表连接所期望的无损性。

$\text{Head}(T_1) \cap \text{Head}(T_2) = \underline{A} \rightarrow \text{Head}(T_2) = \underline{A} D$, 因为(2) $A \rightarrow D$ 。

$\text{Head}(T_1 \bowtie T_2) \cap \text{Head}(T_3) = \underline{B} \rightarrow \text{Head}(T_3) = \underline{B} E$, 因为(3) $B \rightarrow E$ 。

$\text{Head}((T_1 \bowtie T_2) \bowtie T_3) \cap \text{Head}(T_4) = \underline{A} B \rightarrow \text{Head}(T_4) = \underline{A} B C$, 因为(1) $A B \rightarrow C$ 。

因为连接运算有结合性，所以无损性不要求连接的特定顺序，我们可以在表达式 $((T_1 \bowtie T_2) \bowtie T_3) \bowtie T_4$ 中去掉括号。 ■

在前面几节中，我们已经有了计算给定集合F的最小函数依赖集的算法，并定义了无损分解的含义。在后面一节中，我们将学习函数依赖的最小集如何帮助我们创建数据库的一个适宜的范式分解。

6.8 范式

让我们现在回到6.5节不良数据库设计的例子上，它引发出了其后两节中的数学细节。我们希望创建基于图6-15中数据项集合的一个数据库，并考虑例6.6.3中函数依赖集描述相关性规则。图6-22中我们重述一次。

我们从第一范式表emp_info开始，表中包含了所有这些数据项（参见图6-16），并且注意许多设计问题，也就是异常。在下面，我们进行一系列无损的表分解，来消除雇员信息数据库中的冗余情况。

emp_id	dept_name	skill_id
emp_name	dept_phone	skill_name
emp_phone	dept_mgrname	skill_date
$(1) \text{ emp_id} \rightarrow \text{emp_name emp_phone dept_name}$		
$(2) \text{ dept_name} \rightarrow \text{dept_phone dept_mgrname}$		
$(3) \text{ skill_id} \rightarrow \text{skill_name}$		
$(4) \text{ emp_id skill_id} \rightarrow \text{skill_date skill_lvl}$		

图6-22 雇员信息数据库的数据项和函数依赖

正如在定义6.7.2中说明的，数据库模式是数据库中所有表的标题的集合以及设计者施加的所有函数依赖。在图6-23中的表emp_info以及给出的函数依赖构成了这样一个数据库模式。

emp_info						
emp_id	emp_name	...	skill_id	skill_name	skill_date	skill_lvl
09112	Jones	...	44	librarian	03-15-99	12
09112	Jones	...	26	PC-admin	06-30-98	10
09112	Jones	...	89	word-proc	01-15-97	12
14131	Blake	...	26	PC-admin	05-30-98	9
14131	Blake	...	89	word-proc	09-30-99	10
...

(1) $\text{emp_id} \rightarrow \text{emp_name emp_phone dept_name}$
(2) $\text{dept_name} \rightarrow \text{dept_phone dept_mgrname}$
(3) $\text{skill_id} \rightarrow \text{skill_name}$
(4) $\text{emp_id skill_id} \rightarrow \text{skill_date skill_lvl}$

图6-23 具有单一表emp_info的雇员信息模式

1. 一个成功消除异常的分解

在图6-23中出现的一个异常是，如果表emp_info的某个雇员的技能数目变为零，那么没有任何有关这个雇员的行会留下来。删除这个技能，致使我们丢失了这个雇员的电话号码和所在部门。在6.5节的末尾，我们提出通过使表emp_info分解成两个表的办法来消除这个异常，也就是分解成表emps和skills，它们的列名在图6-17中给出，图6-24中我们重复一遍。

emps
emp_id
emp_name
emp_phone
dept_name
dept_phone
dept_mgrname

skills
emp_id
skill_id
skill_name
skill_date
skill_lvl

(1) $\text{emp_id} \rightarrow \text{emp_name emp_phone dept_name}$
(2) $\text{dept_name} \rightarrow \text{dept_phone dept_mgrname}$
(3) $\text{skill_id} \rightarrow \text{skill_name}$
(4) $\text{emp_id skill_id} \rightarrow \text{skill_date skill_lvl}$

图6-24 具有两个表emps和skills的雇员信息模式

当表 emps 和 skills 最初被定义时，我们分析了这个分解的很多特点但未加以证明。现在我们可以证明它们了。

命题6.8.1 表 emp_info 的键是属性集 emp_id skill_id ，这也是表 skills 的键，但是表 emps 的键只含有一个属性 emp_id 。

证 根据定理6.7.3我们可以通过找出一个属性集合 X , $X \subseteq \text{Head}(T)$, 以使 $X \rightarrow \text{Head}(T)$, 来决定表 T 的一个超键。那么，为了说明集合 X 是一个键，我们仅仅需要说明没有 X 的哪个子集 Y 具有这个性质。图6-23中任何一个函数依赖（重述如下）的左边属性集合设为 X ，我们从查找所有这些 X 的集合闭包开始搜索。

- (1) $\text{emp_id} \rightarrow \text{emp_name emp_phone dept_name}$
- (2) $\text{dept_name} \rightarrow \text{dept_phone dept_mgrname}$
- (3) $\text{skill_id} \rightarrow \text{skill_name}$
- (4) $\text{emp_id skill_id} \rightarrow \text{skill_date skill_lvl}$

从 $X=\text{emp_id skill_id}$ 开始（上面函数依赖(4)的左边），我们使用算法6.6.12和给定的函数依赖集合 F 来计算 X^* 。从 $X^*=\text{emp_id skill_id}$, 运用函数依赖(4)，我们得到 $X^*=\text{emp_id skill_id skill_date skill_lvl}$ 。下面，使用函数依赖(4)，因为 skill_id 在 X^* 中，我们把 skill_name 加入到 X^* 中。使用函数依赖(1)，因为 emp_id 在 X^* 中，我们把函数依赖(1)的右边加入，得到 $X^*=\text{emp_id skill_id skill_date skill_lvl skill_name emp_name emp_phone dept_name}$ 。最后我们运用函数依赖(2)，因为 dept_name 现在在 X^* 中，我们加入函数依赖(2)的右边，得到 $X^*=\text{emp_id skill_id skill_date skill_lvl skill_name emp_name emp_phone dept_name dept_phone dept_mgrname}$ 。这个最终结果包含了 emp_info 中的所有属性，也就是 $\text{Head}(\text{emp_info})$ 。根据 X^* 的定义，这意味着

[6.8.1] $\text{emp_id skill_id} \rightarrow \text{Head}(\text{emp_info})$

那么根据定理6.7.3， emp_id skill_id 是 emp_info 的一个超键。

为了说明 emp_id skill_id 事实上是 emp_info 的一个键，我们只需要说明没有它的子集（单独 emp_id 或者 skill_id ）函数决定所有这些属性。让我们由 emp_id 的闭包来看哪些属性可以被函数决定。我们可以立即运用函数依赖(1)来得到 $\text{emp_id} \rightarrow \text{emp_id emp_name emp_phone dept_name}$, 然后可以运用函数依赖(2), 推导出：

[6.8.2] $\text{emp_id} \rightarrow \text{emp_id emp_name emp_phone dept_name dept_phone dept_mgrname}$

因为 skill_id 不在[6.8.2]的右边集合中，且没有其他函数依赖可以使用，所以这是 emp_id 函数决定的最大右边属性集合。

最后单独由集合 X 中的 skill_id 计算闭包，只有函数依赖(3)可以使用。我们看到 skill_id 可以函数决定的最大右边属性集合如下：

[6.8.3] $\text{skill_id} \rightarrow \text{skill_id skill_name}$

[6.8.2]和[6.8.3]都没有包含 emp_info 的所有属性，所以我们可以从[6.8.1]得到结论

[6.8.4] emp_id skill_id 是表 emp_info 的一个键

另外，我们从[6.8.2]注意到 emp_id 函数决定了图6-24中表 emps 的所有属性，因为单元素集合没有子集可以出现在一个函数依赖的左边，所以：

[6.8.5] emp_id 是表 emps 的一个键

最后，我们注意到表skills有一些属性没有被emp_id或者skill_id函数决定，且skill_lv1在[6.8.2]和[6.8.3]的右边都没有出现，所以表skills唯一可能的键是emp_id skill_id。 ■

[6.8.6] emp_id skill_id是表skills的一个键

命题6.8.2 将表emp_info划分成表emps和skills的分解是一个真正的无损分解。

证 为了说明这是一个有效的分解，我们注意到 $\text{Head}(\text{emps}) \cup \text{Head}(\text{skills}) = \text{Head}(\text{emp_info})$ 。进一步， $\text{Head}(\text{emps}) \cap \text{Head}(\text{skills}) = \text{emp_id}$ ，因为函数依赖[6.8.2]显示 $\text{emp_id} \rightarrow \text{Head}(\text{emps})$ ，由定理6.7.4可知这个分解是无损的。 ■

从命题6.8.2可以看到从图6-23的表emp_info到图6-24中表emps和skills的分解总是允许我们重新得到emp_info的原始内容，只要连接两个分解出来的表，但是这个分解的真正动机是处理早些时候提到的种种异常。

在6.5节中提到的图6-23中表emp_info的删除异常是如何产生的呢？基本的原因是属性对emp_id skill_id形成了那个表的键，但是我们希望知道哪些属性被这两个属性中的emp_id函数决定。如果我们删除对于某个emp_id的最后一个skill_id值，我们将不再有与这个emp_id对应的(emp_id skill_id)对，但是我们仍然有只决定于emp_id的信息，那些是我们不希望丢失的！用E-R模型来说，employees是真实的实体，它的属性是我们想知晓的（所以雇员标识符emp_id在一个函数依赖的左边出现）。在图6-24的分解中，我们从表emp_info分解出表emps以使我们在这种方法中不会丢失信息。根据这个新的模式，我们可以在表emps中为一个给定的雇员保留一行，即使这个雇员没有任何技能。回忆前面提到插入异常是删除异常的相反面，所以插入一个新的没有技能的雇员（一个实习生）到表emp_info中变得不可能。正如前面所述，这个问题通过分解出表emps来解决，因为一个新的行可以插入到emps中，该行与表skills中的任何一行没有连接。至于更新异常，这个问题在表emp_info中再次出现，因为仅仅依赖于emp_id的属性存在于键是emp_id skill_id的表中；所以，我们可能在这个表中有带有相同的雇员电话号码的多行，必须全部同时更新。再强调一次，分解出表emps解决了这个问题，因为每一个雇员现在由单一行代表了。

现在的问题是：在图6-24的数据库模式中，还有更多异常存在吗？回答是肯定的，这可能并不令人惊讶。还存在另一种异常，如同我们刚刚在表skills中分析的。这个表有一个主键(skill_id emp_id)，我们回忆图6-22的函数依赖(3)：

[6.8.7] skill_id \rightarrow skill_name

这个函数依赖似乎在说，skills本身是一个实体，skill_id是这个实体的一个标识符，skill_name是一个描述符。（可能存在两个不同的技能具有不同的skill_id值但是有相同的skill_name，因为skill_name \rightarrow skill_id不能从我们的函数依赖列表推导出。）但是，我们已经确定表skills的键是emp_id skill_id。这似乎与我们从emp_info中分解出emps的情况是对称的。我们能够构造（例如）一个导致这一步的删除异常吗？回答是可以，因为如果假设某种技能很少见且不易掌握，同时我们突然失去了最后一个掌握这种技能的雇员，那么我们将根本不再有这种技能的任何信息了，既没有skill_id也没有skill_name。那么，我们需要分解出一个表来解决这个异常，在图6-25中可以看到结果。

通过检查图6-25中新的表emp_skills和表skills，可以清楚地看到这两个表构成了图

6-24中表skills的一个无损分解。事实上，图6-25中的三个表构成了开始时图6-23中表emp_info的一个无损分解。最为重要的是，我们已经处理了那些因为要保留实体skills的属性而产生的异常，这个skills实体在一个表中描述，该表的键具有两个属性。按照E-R模型的概念，我们刚刚所做的就是从两个实体Emps和Skills中分解出联系emp_skills。

现在考虑图6-25的三个表。在表emps的每一项，如同在命题6.8.1中显示的，被单一属性emp_id函数决定；表skills有类似的形式，如同我们在[6.8.7]的函数依赖中看到的；在表emp_skills中，看一下[6.8.2]和[6.8.3]可以弄清楚在这个表中不再剩下什么属性依赖于键(emp_id skill_id)的某个子集。那么我们问在这些表中是否还存在进一步的异常？再一次回答存在！为了看出这是为什么，考虑如果我们的公司中进行一次大的改编将发生什么事情：在一个部门中的每一个雇员将调动到其他部门中（甚至经理也将被调动，假设以后在这个刚被腾空的部门中将会有其他雇员来代替他们的职位）。现在注意到当最后一个雇员被去除时，在表emps中将不再有任何行包含关于这个部门的信息：我们甚至已经丢失了部门电话号码和它的名字！这个问题的解决办法显而易见：我们必须为部门分解出一个单独的表。这将导致图6-26的emp_info数据库；这个数据库满足第三范式（3NF）。在这个例子中，同时等价地满足Boyce-Codd范式（BCNF），稍后我们将给出这些范式的定义。

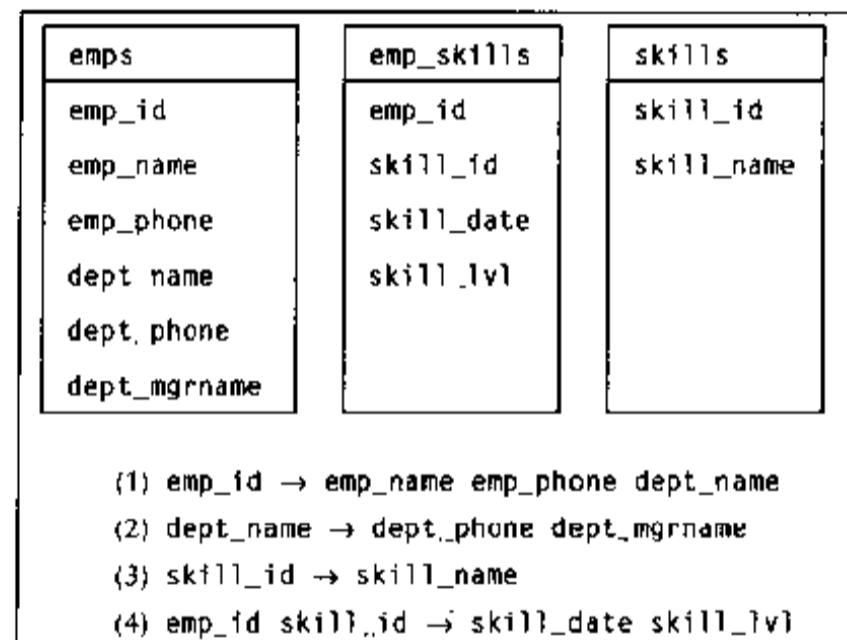


图6-25 具有三个表的雇员信息模式

得到图6-26中表depts的分解后，与部门信息相关联的更新异常将不再困扰我们。用E-R模型术语来说，我们所做的工作就是区分两个实体Emps和Depts以及两者之间的一个多对一联系（表现为表emps中的外键dept_name）。

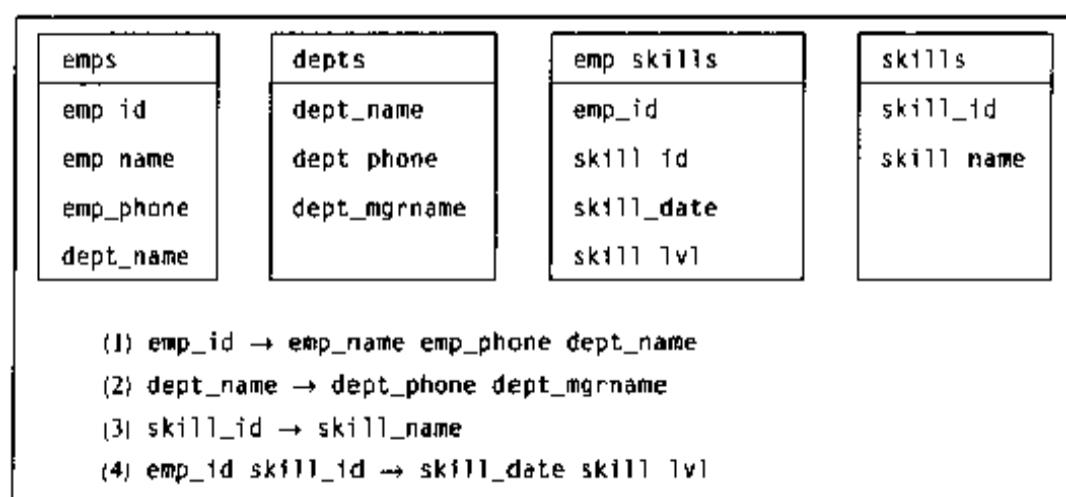


图6-26 符合3NF（也符合BCNF）的雇员信息数据库模式

在这里，我们宣称图6-26的数据库模式在某种意义上是一个最终结果——不再有异常困扰我们。为了明白验证这个陈述的基本原理，我们看看四个在数据库中必须保持的函数依赖，在下面表示成函数依赖集F。先前模式中我们已经提出的种种异常情况，其根本原因已经呈现出来：F中的一个函数依赖的左边的某个属性（在不同的模式中它可能是属性集合）在它出现的表中可能多次重复出现（也可能出现零次）。解决办法是创建一个单独的表，把这个函数依赖左边的属性以及所有处于右边的所有属性放入其中，同时把右边的属性从它们先前出现的表中除去。仔细地顺次观察图6-23到图6-26的分解，可以看出这是对已做工作的一个正确描述。因为这个函数依赖左边的属性在老的和新的表中都出现，而且决定所有新表中的其他属性，所以这个分解是无损的。因此，函数依赖(1)产生了表emp，函数依赖(2)产生了表dept，函数依赖(3)产生了表skills，函数依赖(4)产生了表emp_skills。因为在F中不再有其他函数依赖，所以我们保证没有其他异常会产生，不再需要进一步分解。这样，我们得到了最终形式。

2. 范式：BCNF、3NF和2NF

在图6-26的最终模式中，每一个表有唯一的候选键，我们把它看做这些表的主键。为什么不需要做进一步的分解来表达这些表中异常呢？一个刻画该问题的方法是阐明所有只涉及这个模式中单个表的属性的函数依赖只源自该表的键。下面提供定义来使这个方法精确化。

定义6.8.3 给定一个数据库模式和一个通用表T以及函数依赖集F，令 $\{T_1, T_2, \dots, T_n\}$ 是T的一个无损分解，那么对于F的一个函数依赖 $X \rightarrow Y$ ，如果对分解中的某一个表 T_i ，有 $X \cup Y \subseteq \text{Head}(T_i)$ ，则称该函数依赖在T的分解中被保持，或者说，T的分解保持了函数依赖 $X \rightarrow Y$ 。此时，我们也说函数依赖 $X \rightarrow Y$ “在 T_i 中被保持”，或者说“存在于 T_i 中”或者“在 T_i 中”。 ■

例6.8.1 对于图6-23中的通用表以及函数依赖集F构成的雇员信息模式，我们已经顺次推导出许多分解：两个表的分解（图6-24）三个表的分解（图6-25）和四个表的分解（图6-26）。每一种分解都保持F中的所有函数依赖。例如，在图6-26四个表的分解中，函数依赖(1)存在于表emp中，函数依赖(2)存在于表dept中，函数依赖(3)存在于表skills中，函数依赖(4)存在于表emp_skills中。 ■

因为F中的每一个函数依赖在图6-26四个表中的一个中保持，所以无论何时模式中的一个表被更新，都可以验证任何被这个更新影响的函数依赖仍然有效。可以通过在那个表中检验它的有效性来验证，而不需要连接操作。这是一个分解中为保持函数依赖而进行探求的动机。

定义6.8.4 Boyce-Codd范式（修正的第三范式） 当下面性质成立时，一个数据库模式中的表T及函数依赖集F被称为符合Boyce-Codd范式（BCNF）：任何F可推导出的函数依赖 $X \rightarrow A$ 都在T中，这里A是不在X中的单一属性，X必须是T的一个超键。当一个数据库模式包含的所有表都符合BCNF时，这个数据库被称为符合BCNF。 ■

考虑表T，令 $X \rightarrow A$ 是T的一个函数依赖。如果BCNF性质在此成立，那么X是一个超键，所以对于表示T的键的某个集合K，有 $K \supseteq X$ 。（注意，可能有很多不同的集合 K_1, K_2, \dots 都是T的候选键，如同我们在下面例6.8.4中考虑的。）如果BCNF性质不成立，那么X不包含键属性的集合K，对所有K， $K - X$ 是非空的。这里可能存在两种情况：或者（1） $X - K$ 对某个K是空的，即 $X \subseteq K$ 。我们说T的一些属性被键K的一个合适的子集X函数决定；或者（2）对所有K， $X - K$ 是非空的。那么一些属性被集合X决定，对于每一个K，X都至少有部分属性不在K

中。在第二个情况中，我们说T的一些属性被一个不同的属性集合（它不包含任何键集合，也不被任何键集合包含）函数决定。

例6.8.2 在图6-26的表emp_skills中，唯一的键是 emp_id skill_id ，我们可以通过讨论集合闭包很容易地证明这一点：任何函数决定表emp_skills的所有属性的属性集合必然包含这两个属性。我们宣称这个表符合BCNF并将在下一个例子中证明。正如我们刚指出的，定义6.8.4的BCNF性质意味着这个表没有属性被这个键集合的任何子集或者任何不包含这个键集合的不同属性集合函数决定。

在图6-24的表skills中，唯一的键是 emp_id skill_id ，而函数依赖 $\text{skill_id} \rightarrow \text{skill_name}$ 也存在于这个表中。显然，这个函数依赖的左边是键 emp_id skill_id 的一个子集。因此，BCNF性质对这个表不成立（我们曾指出一旦一个异常产生，我们就进行进一步分解）。

在图6-24的表emps中（与图6-25的表emps相同），唯一的键包含属性 emp_id ，而函数依赖 $\text{dept_name} \rightarrow \text{dept_phone}$ 是由F的函数依赖(2)推导的且在这个表中存在。既然这个函数依赖的左边与键集合不同（即不是子集也不是超集），那么不满足BCNF性质，需要进一步分解。注意，通过这个方法，一个包含emps的dept_phone以外所有其他属性的表emps2仍然不遵守BCNF性质。尽管函数依赖 $\text{dept_name} \rightarrow \text{dept_phone}$ 不存在于表emps2中，但是同样由函数依赖(2)推导出的函数依赖 $\text{dept_name} \rightarrow \text{dept_mgrname}$ 却存在于emps2中。■

例6.8.3 我们宣称图6-26的数据库模式是符合BCNF的。我们需要说明如果任意由F推导出的函数依赖 $X \rightarrow A$ 存在于图6-26的一个表中（这里A是一个不在X中的属性），那么X包含该表的一个键。我们已经在例6.8.1中说明对于图6-26的表集合，F的一个函数依赖存在于一个表中，而且这个函数依赖的左边是这个表的键。然而这并未彻底解决问题，因为我们还需要考虑所有被F蕴涵的函数依赖，那就是所有在这个模式中为真的函数依赖。对于命题6.8.1中的函数依赖[6.8.1]，[6.8.2]和[6.8.3]，我们计算了F的这三个函数依赖左边属性集合的闭包，说明这三个集合构成了三个表的键。对于第四个表，我们只需要计算dept_name的闭包，得到 $\text{dept_name dept_phone dept_mgrname}$ ，即 Head(depts) ：

[6.8.8] $\text{dept_name} \rightarrow \text{Head(depts)}$

现在我们宣称不包含这些X集合（F中一个函数依赖的左边，同时是图6-26中某个表的键）中任一个的所有属性集Z，必然有平凡闭包 $Z^+ = Z$ 。因为没有 $X \rightarrow Y$ 形式的函数依赖存在 $X \subseteq Z$ ，同时由算法6.6.12，没有属性将被加入到Z的集合闭包中，所以可以得出上面结论。

由此，我们可以容易地看到图6-26中的所有表符合BCNF，因为如果 $X \rightarrow A$ 成立，A是一个不被包含在属性集X中的属性，那么 $X \rightarrow AX$ ，所以 X^+ 不同于X。但是我们已经说明任何不包含表键的属性集有一个平凡闭包，这一定意味着X包含某个键K。在那个表中，我们同样已经包含了被K函数决定的所有属性，所以A也在那个表中。■

例6.8.4 假设我们改变雇员信息数据库中的规则，使dept_mgrname是实体Departments的第二个标识符，与dept_name的作用相当。这将在集合F中加入一个新的函数依赖 $\text{dept_mgrname} \rightarrow \text{dept_name}$ ；根据传递性，因为dept_name是图6-26表depts的一个键，所以dept_mgrname也是一个键。现在问题是表depts是否仍然是BCNF的。回答是肯定的，因为BCNF性质并不要求表有唯一一个键。在表depts中改变的唯一内容是现在有两个键，但是这个表中任何 $X \rightarrow Y$ 形式的函数依赖具有必需的性质，即X包含dept_mgrname

或者X包含dept_name。

F中的每一个函数依赖在图6-26的一个表中存在，所以无论何时模式中的一个表被更新都可以验证被涉及的函数依赖仍然成立，这可以通过只检查这一个表的数据项来确定。我们希望总可以从一个通用表开始进行无损分解到满足BCNF来保证函数依赖的保持。不幸的是，这不成立，因为一个表的BCNF标准太严格了。

例6.8.5 我们希望加入很多属性到图6-22的雇员信息数据库中，以知晓所有雇员的完整地址（假设生活在美国）：

emp_cityst, emp_straddr, emp_zip

这里，emp_cityst反映了城市和州，emp_zip是邮政编码，emp_straddr是街道名、号码和寓所。我们发现当到达图6-26的分解时，表emps包含了所有这些属性（除了已有的属性），参见图6-27。

我们假设每个雇员被要求提供一个地址，显然emp_id函数决定了所有这些新的属性，函数依赖(1)相应修改成：

(1) $\text{emp_id} \rightarrow \text{emp_name } \text{emp_phone } \text{dept_name } \text{emp_straddr } \text{emp_cityst } \text{emp_zip}$

图6-26中任何表的键都没有被影响，表emps的键仍然是emp_id。

emps
emp_id
emp_name
emp_phone
dept_name
emp_cityst
emp_straddr
emp_zip

图6-27 被扩展的包含雇员地址的emps表

emps
emp_id
emp_name
emp_phone
dept_name
emp_cityst
emp_straddr

empadds
emp_cityst
emp_straddr
emp_zip

图6-28 图6-27的一个3NF分解

但是邮局已经给城市的区域分配了邮政编码（由街道地址决定）并且不交叉城市边界，所以我们也有下面这些新函数依赖加入到集合F中：

(5) $\text{emp_cityst } \text{emp_straddr} \rightarrow \text{emp_zip}$ 城市区域决定邮政编码

(6) $\text{emp_zip} \rightarrow \text{emp_cityst}$ 邮政编码始终不交叉城市边界

因为函数依赖(5)的左边emp_cityst emp_straddr不是emps表的一个超键，所以我们需要进一步分解以获得BCNF性质。如果我们没有做这一步并且删除了在某一个邮政编码中的最后一个雇员，我们将丢失关于那个邮政编码的信息，也就是与它相关联的城市和州。按照在图6-26后面讨论中解释的命题，我们将函数依赖(5)左边的属性以及这个函数依赖右边的所有属性添加到一个单独的表empadds中，而把右边的属性从原先它们所在的表(emps)中除去。在图6-28中显示结果。这是先前表的一个完全合理的无损分解（无损是根据定理6.7.4得到的，因为emp_cityst emp_straddr是empadds的一个键，同时也是两个表标题的交集）。我们也注意到emp_zip emp_straddr是表empadds的另一个候选键，因为计算这个集合的闭包时我们由函数依赖(6)得到了emp_cityst。所以我们推导出新的函数依赖(7)。

很容易从闭包中看出empadds没有其他候选键存在。

(7) $\text{emp_zip } \text{emp_straddr} \rightarrow \text{emp_cityst}$ 由(5)和(6)推导出的函数依赖

图6-28的表emps现在满足BCNF，因为函数依赖(5)和(6)都不存在于emps中，同时唯一剩下的函数依赖(1)要求对于这个表的任何超键都包含emp_id。这个分解也保持函数依赖(5)和(6)，它们都完全存在于表empadds中。然而，这里函数依赖(6)强迫我们对表empadds进行进一步的分解以获得BCNF性质，因为 $\text{emp_zip} \rightarrow \text{emp_citystr}$ ，而且 emp_zip 不包含empadds的两候选键之一。明显，这个新的分解在每一个表中一定最多包含两个属性，我们要求（图6-29的zipcit）一个具有标题 $\text{emp_zip } \text{emp_cityst}$ 的表来包含函数依赖(6)。另一个表只有两个可能的属性对作为标题，即函数依赖(5)或者函数依赖(7)的左边，二者都是表empadds的键。选择函数依赖(5)的左边， $\text{emp_cityst } \text{emp_straddr}$ 将不能够得到一个无损分解，因为它与zipcit唯一共有的属性是 emp_cityst ，而这不包含两表中任何一个的键，所以我们选择图6-29中的BCNF分解。

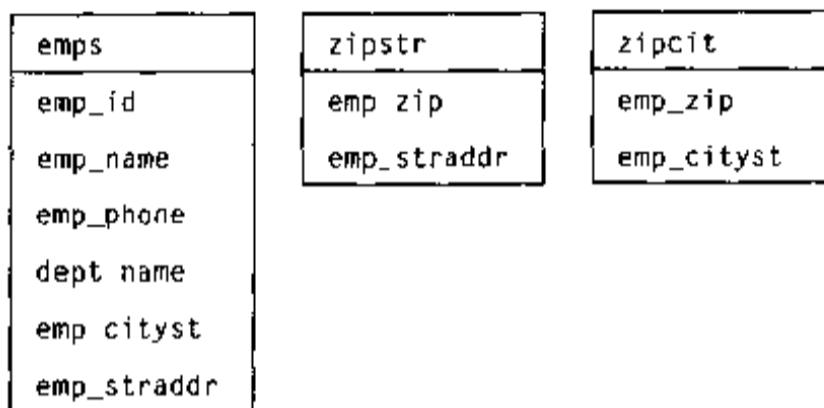


图6-29 图6-28的一个BCNF分解

图6-29的分解是无损的，原因如下： emp_zip 是表zipcit的键，是Head(zipstr)和Head(zipcit)的交集，所以这个连接是无损的；zipstr和zipcit标题的并集包含表empadds的所有属性，所以zipstr和zipcit连接后构成图6-28中表empadds；表empadds和emps构成一个无损连接，所以这三个表的连接是无损的。进一步，图6-29的两个新表都满足BCNF范式。表zipcit的唯一函数依赖是函数依赖(6)，而且 emp_zip 是键。表zipstr中没有函数依赖，所以唯一的键包含两个属性 $\text{emp_zip } \text{emp_straddr}$ ，它也是图6-28中表empadds的一个候选键。

但是图6-29的分解不保持扩展的集合F的依赖，因为函数依赖(5)不在两表之一中存在，这会产生一个不幸的结果：我们必须进行程序性检查以确保输入的给定街道地址、城市、州和邮政编码与邮局的分配相符合。■

如果我们是想保持函数依赖，那么我们似乎在分解上已经走的太远了。我们所期望的是对范式的一个定义，以使我们在图6-28停止而不继续到图6-29。为此，我们必须提出范式的一个新定义（也就是3NF）。

定义6.8.5 主属性 (Prime Attribute) 在表T中，一个属性A称为是主属性当且仅当属性A存在于这个表的某个键K中。■

定义6.8.6 第三范式 (Third Normal Form) 当数据库模式的表T以及函数依赖集F满足下面条件时，被称为符合第三范式 (3NF)：对任何由F推导并存在于表T中的函数依赖 $X \rightarrow$

A (这里A是单个属性且不在X中), 下面两个性质之一必须成立, 或者(1) X是T的一个超键, 或者(2) A是T的一个主属性。当一个数据库模式包含的所有表符合3NF时, 这个数据库模式称为符合3NF。 ■

例6.8.6 考虑图6-29的数据库模式。其中的每一个表都符合BCNF, 所以这个模式也符合3NF。BCNF要求一个表具有3NF定义的性质(1), 而不允许性质(2)的“逃脱子句”。所以任何BCNF的表也是符合3NF的, 但反之不然。 ■

例6.8.7 考虑图6-28的表empadds, 这个表是3NF但不是BCNF的。我们需要对这个表进一步分解的原因是图6-28的表empadds有一个键emp_cityst emp_stradd, 同时函数依赖(6) $\text{emp_zip} \rightarrow \text{emp_cityst}$ 存在于这个表中。这个函数依赖的左边不包含empadds的键, 所以它不满足BCNF性质。然而, 我们注意到这个函数依赖的右边的属性存在于某个键中, 所以它是主要属性。因此这个函数依赖满足3NF定义的性质(2)。 ■

例6.8.8 给定表T, $\text{Head}(T)=A B C D$, 函数依赖集F如下:

$$F: (1) A B \rightarrow C D, (2) D \rightarrow B$$

显然, A B是T的一个候选键, 我们由闭包看到A D是另一个候选键: $A D^+ = A D B$ (2) C (1)。很容易确认没有其他候选键。现在表T已经是3NF的, 因为由F推导的不包含A B在左边(所以不是平凡的)的唯一函数依赖决定于函数依赖(2) $D \rightarrow B$, 且因为B是一个主属性, 所以表T满足性质(2)的“逃脱子句”, 它是符合3NF的。

如果我们确实希望把T无损地分解成BCNF范式, 我们从投影到包含函数依赖(2)的表开始, 也就是表 T_2 , $\text{Head}(T_2)=B D$ 。那么我们想使表 T_1 包含T的一个候选键和属性C。但是如果我们将创建表 T_1 , 使 $\text{Head}(T_1)=A B C$, 那么 T_1 和 T_2 的标题的交集将不包含D, 所以连接不是无损的。因此, 我们必须创建表 T_1 , 使 $\text{Head}(T_1)=A D C$, 那么我们就有了BCNF分解 $\{A D C, B D\}$ 。 ■

在分解具有给定函数依赖集F的数据库以获得范式的时候, BCNF和3NF范式通常是相同的, 如同我们在例6.8.6中看到的。只有当存在两个F推导的非平凡函数依赖 $X \rightarrow Y$ 和 $Z \rightarrow B$, 其中 $Z \subset X \cup Y$ 且 $B \in X$ 时, 它们才不相同。在例6.8.8中, 函数依赖(1) $A B \rightarrow C D$ 和函数依赖(2) $D \rightarrow B$, 其中 $D \subset C D$ 且 $B \in AB$, 为取得BCNF而进行的进一步分解将导致依赖不再被保持。很多数据库设计者以保持依赖的3NF设计为目标。

表性质的另一个定义, 第二范式(second normal form, 2NF)比3NF弱。它是过时的东西, 因为在不达到3NF时停止没有什么好处。从定义6.8.6我们看到当一个表不能成为3NF时, 它一定含有一个有效的非平凡函数依赖 $X \rightarrow A$, 这里A是非主属性且X不是T的一个超键。回忆在定义6.8.4后面对BCNF的讨论, 如果X不是T的一个超键, 那么可能有两种情况: 或者对某个K有 $X \subset K$, 我们说T的一些属性被键K的某个适当子集X函数决定; 或者 $X - K$ 对T中所有键K都是非空的, 我们说T的一些属性函数决定于一个不同的属性集合, 这个集合不包含任何键集合也不被任何键集合包含。后一种情况也被称为传递依赖, 因为对任何键K我们有 $K \rightarrow X$, 且给定 $X \rightarrow A$, 所以函数依赖 $K \rightarrow A$ 可以由传递性推导出来。一个2NF的表不允许有被键K的真子集函数决定的属性, 但可能仍然有传递依赖。

定义6.8.7 第二范式 数据库模式中的表T以及函数依赖集F被称为满足第二范式(2NF), 需要满足下面条件: 对存在于T中由F推导的任何函数依赖 $X \rightarrow A$ (这里A是一个不在X中的单一属性而且是非主属性), X不是T的任何键K的真子集。当一个数据库模式包含的所有表都符

合2NF时，这个数据库模式称为符合2NF。

例6.8.9 图6-25的数据库模式是2NF的。它的证明留作本章末尾的习题。

3. 获得良好3NF分解的一个算法

因为很多技术原因，看起来顺次分解来获得保持函数依赖的3NF无损连接分解的方法是不被很多实践人员信任的。这是我们见到的是唯一方法，在图6-23到6-26中使用。因为在顺次分解中使用的函数依赖集F没有被认真地定义，所以问题会发生，如同我们在6.6节中看到的，可能有很多的等价集合F_c。算法6.8.8提供了一种直观的方法来创建所需要的分解。

算法6.8.8 给定一个通用表T和函数依赖集F，这个算法产生T的一个符合第三范式且保持F中函数依赖的无损连接分解。算法输出最终数据库模式中表的标题（属性集合）的一个集合。

```

REPLACE F WITH MINIMAL COVER OF F;          /* use algorithm 6.6.13      */
S = Ø;                                         /* initialize S to null set   */
FOR ALL X → Y in F                           /* loop on FDs found in F    */
  IF, FOR ALL Z ∈ S, X ∪ Y ⊥ Z              /* no table contains X → Y   */
    THEN S = S ∪ Heading(X ∪ Y);            /* add new table Heading to S */
  END FOR                                      /* end loop on FDs           */
  IF, FOR ALL CANDIDATE KEYS K FOR T         /* if no candidate Keys of T */
    FOR ALL Z ∈ S, K ⊥ Z                      /* are contained in any table */
    THEN CHOOSE A CANDIDATE KEY K AND        /* choose a candidate key     */
      SET S = S ∪ Heading(K);                 /* and add new table to S    */

```

注意到函数Heading(K)生成一个集合，包含属性集合K，它可以被加入到集合S中，S是一个属性集合的集合。

例6.8.10 为说明为什么在算法6.8.8中有时候需要选择候选键，考虑下面一个小型的学校数据库。给定一个通用表，其标题如下：

Head(T) = instructor class_no class room text

和函数依赖集F

F = {class_no → class room text}

在E-R概念中，有一个实体Classes，用class_no标识，且实际的班级在具有唯一内容的同一教室中举行它的所有会议。是否有实体Class_rooms（标识符是class_room）是可选择的。因为没有哪一个函数依赖的左边是class_room，所以这样一个实体将没有描述符属性，并且在关系模型中没有与它对应的表；因此如果我们愿意，可以把class_room认为是一个描述符属性。对Head(T)中的属性text可以用同样的方式讨论。但是Head(T)中的属性instructor的情况不同，因为属性instructor不被class_no函数决定，所以一个班级可能有几个教师，同时因为instructor不函数决定class_no，这意味着一个教师可以教多个班级。由此很清楚教师不依赖于班级而存在，事实上表示了一个实体Instructors。表T实际上包含了Instructors和Classes之间的一个联系。

根据标准BCNF/3NF规范化方法，因为属性class_room和text在表T中依赖于class_no，所以我们需要把T分解成为两个表，T₁和T₂，如下

Head(T₁) = class_no class room text

Head(T₂) = instructor class_no

但是在算法6.8.8中，因为属性instructor不存在于F的任何函数依赖中只有表T₁将在函数依赖的首次循环中被创建。然而从标准集合闭包方法中可以很明显看到，T的唯一候选键是class_no instructor，因此，算法6.8.8中候选键上的循环在为集合S创建表T₂时是需要的。 ■

通常我们说规范化方法和E-R方法互相补充，例6.8.10给出了这方面的例子。不考虑函数依赖，就不会清楚为什么数据项instructor一定代表一个实体而数据项class_room不是，另一方面，为什么在算法6.8.8在候选键上循环对创建表T₂是需要的？E-R方法解释了这样做的动机，我们需要用表T₂来表示实体Instructors和Classes之间的联系。

4. 规范化的回顾

在数据库设计的规范化方法中，我们从一个数据项集合以及一个函数依赖集F开始，设计者希望这个函数依赖集在数据库将来任何内容中都被数据库系统维持。数据项全放置在单个通用表T中，集合F用一个等价最小覆盖替换掉；然后设计者决定这个表的一个分解，将表分解成更小的表构成的一个集合{T₁, T₂, …, T_n}，它具有的很多优良性质如下。

- 1) 分解是无损的，所以T=T₁ ⊗ T₂ ⊗ … ⊗ T_n。
- 2) 在可能的最大限度内，表T_i中存在唯一函数依赖X→Y是由X包含T_i的某个键K而产生的，这是BCNF/3NF定义强加的。
- 3) F中所有形式是X→Y的函数依赖在分解出的表中被保持。

性质2的价值是我们可以避免在6.5节中定义的各种异常。同样重要的是，根据这些范式，我们可以保证函数依赖不会被破坏，只要我们保证一个表的所有键的唯一性。在第7章的开始，我们将看到SQL的Create Table语句给我们提供了一条途径来定义一个表的这些键K，而这些键的唯一性将被系统保证，对于其后所有SQL的表更新语句都将成立（一个打破这种唯一性约束的更新操作会产生一个错误）。如同我们稍后将看到的，如果在所涉及的键列上建立索引，那么唯一性条件很容易检查；然而表T_i的一般函数依赖X→Y较为困难，其中存在有相同X值的多行。标准SQL没有提供一个约束来保证这样的一般函数依赖不产生更新错误。

性质3的意义同样明显，因为我们想确保设计者提供的所有函数依赖对数据库的任何可能内容都成立。性质3意味着在最终的数据模式中函数依赖不能跨越表，所以如果一个表的更新发生，只有那张表的函数依赖需要被系统测试。另一方面，我们提供的分解的确导致一定数量的连接测试，因为标准无损连接分解为表T₁和T₂导致其中一个表的键产生，而这个键的属性存在于两个表中，也就是一个由(Head(T₁) ∩ Head(T₂))构成的键。标准SQL提供了一个约束，称为参照完整性，可以在Create Table语句中施加来确保属性值在它们连接的两个表之间依然有意义，这个约束也被称为外键条件。

总之，标准3NF分解消除了大多数的异常，使得数据库在更新时有效地验证所期望的函数依赖是否依然保持成为可能。

还有其他一些范式，4NF和5NF，这里不作介绍。特别地，第四范式，即4NF，基于一个完全新型的依赖，这种依赖称为多值依赖。你可以从推荐读物[4]或者[5]得到它们的详细描述。

在此，我们提一下过度规范化（overnormalization）。当以3NF做为目标时，它导致一个数据库分解成比所需要的还要多的表，过度规范化被认为是不良实践。例如，如果我们把表depts分解成两个表，一个包含dept_name和dept_phone，另一个包含dept_name和

`dept_mgrname`, 我们当然也得到了一个3NF的数据库, 但是在分解上我们已经走的比需要的远了, 在检索所有部门信息时将出现不必要的低效现象, 因为此时需要进行连接操作。

6.9 其他设计考虑

E-R方法和规范化方法都有各自缺点。通常表现为, E-R方法极端地依赖直觉, 但是如果直觉是错误的, 几乎没有任何反馈。如同我们在例6.8.10中看到, 单单依靠直觉来决定是否一个数据项代表了一个实体很困难。这使得我们由规范化形成了函数依赖的概念。规范化更多基于数学方法, 而且在应用时更为机械, 但是想在逻辑数据库设计的第一步就写出一个完整的函数依赖集经常是妄想的, 过后可能就会发现一些函数依赖被遗漏了。运用直觉来发现实体和联系以及弱实体等等有助于设计者发现可能被遗漏的函数依赖。

另一个影响规范化方法的因素是, 可能需要一定量的判断来决定是否特定的函数依赖应当在最终的设计中反映出来。考虑CAP数据库模式, 它的表在图2-2中列出。看起来所有对数据库成立的函数依赖都是表中键依赖的反映, 所以所有表都是BCNF的。然而, 有一个函数依赖未被预料到, 它具有如下形式:

[6.9.1] `qty price discnt → dollars`

对每一个订单, 从订购数量、产品价格、顾客折扣我们可以计算出这个订单的金额——这个联系在2.1节中提到, 并用SQL的Insert语句[6.9.2]表达。现在的问题是, 这个函数依赖使得图2-2中的表集合变成一个不良设计吗? 显然这个函数依赖是跨越多个表的, 所以分解不保持依赖。注意到我们可以创建另一个表`ddollars`, 包含函数依赖[6.9.1]两边的所有属性`qty price discnt dollars`, 同时从`orders`中去除属性`dollars`。结果是CAP的一个含五个表的模式, 其中包括表`ddollars`。这是一个可以由算法6.8.8得到的3NF设计。表`ddollars`的唯一键是`qty price discnt`, 唯一的函数依赖在[6.9.1]中给出。然而这个设计存在一个问题。无论何时我们想检索一个订单的金额, 对于给定的`qty`、`price`和`discnt`, 我们不得不连接三个表: 从`products`得到`price`, 从`customers`得到`discnt`, 从`ddollars`得到`dollars`值。所有这些是必须的吗? 最初图2-2中的设计从这一点来说似乎更好。

如果考虑分解的原始动机, 我们可以找出两点: 消除异常和使所有函数依赖在任何时候数据变化时依然有效。但是我们真的想使用范式中一个唯一键约束来使这个函数依赖有效吗? 设想当一个新的订单被插入时, 程序逻辑计算出金额总量并存贮, 就像如下操作:

```
[6.9.2] exec sql insert into orders
    values (:ordno, :month, :cid, :aid, :pid, :qty,
           qty*:price - .01*:discnt*:qty*:price;
```

由这个Insert命令我们可以保证函数依赖[6.9.1], 并且还保证了用函数依赖不能表示的确切数值联系。表`ddollars`能够提供的唯一保证是: 如果存在先前的一行有给定的`qty`、`price`和`discnt`, 那么计算出的`dollars`值将是相同的。这看起来是一个非常奇怪的有效性, 因为有很多产品和顾客, 订购量差别很大且在订购条目的数量上有某种限制, 那么我们希望在第一次加入很多(`qty`, `price`, `discnt`)元组, 所以这样的唯一键约束没有具有实际价值的验证功能: 没有具有相同键的旧行可以与之比较。你将宁愿依赖于Insert语句[6.9.2]来进行正确的计算。基于此, 在一个检验过的函数中提供这个插入当然是有意义的, 这个函数在所有的逻辑插入新行的操作中必须使用。

现在删除和插入异常共同说明我们不想丢失有关任何(*qty*, *price*, *discnt*)元组的信息，但是如果考虑到我们并不真正评价这个有效性方法，这便是一个有问题的命题。至于更新异常，对于给定的(*qty*, *price*, *discnt*) 我们考虑需要立即更新所有金额值的情况。如果需要为先前输入的订单改变*price*或者*discnt*值，需要更正，这便可能发生。但是这个改变如此不寻常并对一个批发业务将产生重大影响，假设一个不熟练的程序员能够写出代码来更正*orders*中错误输入的一行是不合情理的。事实上，很多设计者会把*dollars*列设计成只能做插入操作的量，根本不能被更新（期望改正输入错误），所以我们愿意放弃对更新异常的防范。

我们在这里已经用例子详细说明了一种在商业应用中频繁发生的情况，即一种通过反向规范化来提高性能的方式。大多数设计实践者会承认经常需要做这种事情。

数据库设计工具

很多商业产品以提供支持数据库管理员进行数据库设计的环境为目标。这些环境作为数据库设计工具提供，有时候则作为计算机辅助软件工程（CASE）工具的一部分，CASE是一种更一般化的产品。这种工具通常有很多组件，它们由下面各种类型组件中的部分组成。单个产品提供所有这些组件是很少见的。

E-R设计编辑器 一个通用组件是设计者可以在其中构造E-R图的界面，可以通过图形的拖放方式来编辑和修改E-R图，拖放方式在Apple的Macintosh和Microsoft的Windows中很容易实现。

E-R到关系设计转换器 这类工具的另一个通用组件是可以自动将E-R设计转换到关系表定义集合的转换器。它按照6.3节中概述的步骤进行，在6.4节中举了例子。

有了数据库设计工具，开发流程通常从E-R设计开始，直到关系表定义结束。然而，很多产品处理函数依赖。有一个工具用来装载一个小型通用表，并从这些数据中提取可能成立的函数依赖，然后自动生成这个函数依赖集合到BCNF/3NF的转换。

函数依赖到E-R设计转换器 有时候也提供另一种类型的组件，它从一个数据库的函数依赖集合生成有效的E-R图来反映数据的规则。

如同在前一节中说明的，一个理论上完美的设计可能效率十分低下。所以好的设计工具试图分析一个设计的性能表现，并接受设计者的决定来进行特定种类的反向规范化以提高性能。另外，工具必须“原谅”函数依赖和实体分类的错误和缺漏，以产生设计的某种最佳推测，设计者在做更正时可以描绘出这个设计。由此引出了另一种类型的标准工具组件。

设计分析器 这些组件分析目前阶段的设计并给出报告，它可以帮助数据库管理员改正各种各样的错误。

若想更好地了解数据库设计工具，你可以阅读“推荐读物”中的推荐读物[1]的最后一章。

推荐读物

在逻辑数据库设计领域中，术语存在很多差异是普遍的。E-R方法有时候被称为语义建模（semantic modeling）。我们把现实世界的对象称为实体实例，而其他的文字中常常称为实体；我们称实体是一个实体实例的类，而他们称为实体类型，正如我们在第4章中的对象和对象类型。属性有时候也被称为性质（property）。

让我们试着解释语义建模意味着什么。在一种程序设计语言中，语言的语法说明了语句

如何由基本的文字元素构成，然而语法不关联任何语句的含义。一个对程序设计语言语句在所有条件下如何作用的说明，即这些语句起什么作用，被称为语言的语义（semantics）。术语“语义建模”暗示着在E-R方法中我们将涉及数据项实际上表示了什么，以便能够按照数据库结构（如关系表）的概念来模型化它们的行为。

推荐读物[1]、[2]和[4]中介绍了逻辑数据库设计。推荐读物[1]的最后章节中还有David Reiner写的一篇关于数据库设计的商业产品即数据库设计工具的文章。推荐读物[3]和[5]对本章中没有介绍的很多规范化概念有很好的描述。推荐读物[3]尤其先进，代表了这一领域的最新成果。推荐读物[6]与本章处于同一个层次，介绍了实体联系和规范化。

- [1] C. Batini, S. Ceri, and S. B. Navathe. *Conceptual Database Design*. Redwood City, CA: Benjamin-Cummings, 1992.
- [2] C. J. Date. *An Introduction to Database Systems*. 6th ed. Reading, MA: Addison-Wesley, 1995.
- [3] David Maier. *The Theory of Relational Database*. New York: Computer Science Press, 1983.
- [4] Toby J. Teorey. *Database Modeling and Design: The Fundamental Principles*. 2nd ed. San Francisco: Morgan Kaufmann, 1994.
- [5] Jeffrey D. Ullman. *Database and Knowledge-Base Systems*. Volum 1. Rockville, MD: Computer Science Press, 1988.
- [6] Jeffrey D. Ullman and Jennifer Widom. *A First Course in Database Systems*. Englewood Cliffs, NJ: Prentice Hall, 1997.

习题

在本书最后的“习题解答”中给出答案的习题用•号标出。

6.1 • 在图6-6中，如果我们不假设三个图中R的连接线的数目完全表示了设计者的意图，而假设这个数目是偶然的但在设计者意图的限制之内。在一些情况下我们仍然可以得出与R关联的E和F的min-card和max-card值。列出图6-6中三个图的这些值。

6.2 • 如同在例6.1.3中指出的，表orders不表示一个联系，而更像一个实体Orders。实体Orders自身被一个二元联系连接到三个实体Customers, Agents和Products。这些联系如下：Customers要求（requests）Orders, Agents设置（places）Orders和Orders装运（ships）Products。画出所有这些实体和联系的E-R图并附加所有相关的属性，标明主键和基数。注意图6-11非常不同，一个实体Orders由多个Line_items构成。

6.3 模仿6.4节的例子，创建E-R设计并由此生成数据库的关系表设计来表示一项银行业务。在数据库中，我们需要知晓在这家银行的分行中有账户的顾客。每一个账号在一个特定分行中保存，但是一个顾客可能有多个账户并且一个账户可能有多个相关联的顾客。我们用acctid以及附加的属性acc_type（存款，结算等等）和acct_bal（账户余额）来标识账户。每一个分行有一个标识符bno和一个属性bcity。顾客用ssn（社会保险号）来标识并有属性cname，cname由clname、cfname和cmidinit构成。

在这个E-R设计中，你应当考虑如何表示顾客-账户-分行的组合。也许所有这三个全是实体，并且它们之间有一个三元联系，或者只有两个实体Customers和Branches，has_account是它们之间的一个联系，该联系具有自己的属性，联系的实例表示一个账户。

可能有不止一个正确的解决方案，但是在这些可选方案中你至少应当能够做出一个。你可以考察这三个设计，决定哪一个你更欣赏，并至少证明它们中一个的正确性。在证明你的决定时，考虑下面的问题：是不是所有这些设计都允许几个顾客共同拥有同一个账户？

6.4 在例6.6.4中，假设由内容推导出的函数依赖是正确的，但是现在新的行可以被加入（必须仍然遵守这些函数依赖）。下面哪些行可以合法地加入到已经存在的行中去？如果不能加入，说明例子中使加入操作不合法的函数依赖编号。

- (a)*

a5	b6	c7	d8
----	----	----	----

- (b)

a2	b2	c1	d8
----	----	----	----

- (c)*

a3	b1	c4	d3
----	----	----	----

- (d)

a1	b1	c2	d5
----	----	----	----

6.5 •像例6.6.4那样，列出下面表T满足的所有函数依赖，这里我们假设设计者的意图是只有这些行的集合可以存在于表中。

T				
行#	A	B	C	D
1	a1	b1	c1	d1
2	a1	b1	c2	d2
3	a1	b2	c3	d1
4	a1	b2	c4	d4

6.6 以如下表T重复前一个练习。

T				
行#	A	B	C	D
1	a1	b2	c1	d1
2	a1	b1	c2	d2
3	a2	b2	c1	d3
4	a2	b1	c2	d4
5	a2	b3	c4	d5

6.7 [难]我们扩展例6.6.4中的想法，表中有固定的内容，设计者的意图是只有那些行可以存在于表中，所以函数依赖可以通过测试来决定。注意到如果给定一个表，它具有给定的标题（属性集合），但是包含或者零行或者一行，测试看起来允许所有可能的函数依赖成立。为了由定义6.6.2说明一个函数依赖不满足，在表中必须至少有两行使得在某些列上匹配但在其他列上不匹配。一个阿姆斯特朗表（Armstrong table）T是包含最少行的表，从它的内容可以使一个给定的函数依赖集F成立（当然也使所有F*中的函数依赖成立），而所有不在F*中的函数依赖将不成立。所以如果X→A不是在F*中的函数依赖，T中一定有两行满足u[X]=v[X]且u[A]<>v[A]。

- (a) • 在例6.6.4中的表是阿姆斯特朗表吗？请说明。你能给出一个表，它具有更少的行但决定了相同的函数依赖集吗？（可能没有。）
- (b) 习题6.6中给出的表是一个阿姆斯特朗表吗？按照(a)来说明你的回答。
- (c) 创建一个阿姆斯特朗表来表示下面在属性A, B, C, D(没有其他属性)上的函数

依赖集。如果还存在其他函数依赖，则回答失败。试创建一个具有最少行集合的表，它具有这一性质。

(1) $A \rightarrow B$, (2) $B \rightarrow C$

(d) [很难]给出一个算法，它可以由一个给定的属性集上的函数依赖集生成一个阿姆斯特朗表。这个算法说明阿姆斯特朗表总是存在的。

6.8 给出定义6.6.6阿姆斯特朗公理的传递规则的证明，按照定理6.6.3中包含规则以及定理6.6.7的增广规则的证明形式。

6.9 • 使用定义6.6.6中阿姆斯特朗公理来推导定理6.6.8中列出的没有给出证明的规则。

6.10 考虑下面一组函数依赖推导规则，其中， X ， Y ， Z 和 W 是属性集合， B 是单一属性。

1) 自反规则 (Reflexivity rule) : $X \rightarrow X$ 永真。

2) 投影规则 (Projectivity rule) : 如果 $X \rightarrow Y$ ，那么 $X \rightarrow Y$ 。

3) 累积规则 (Accumulation rule) : 如果 $X \rightarrow Y$ 且 $Z \rightarrow B$ ，那么 $X \rightarrow YZB$ 。

说明如何用这些规则推导定义6.6.6中的三个推导规则，即阿姆斯特朗公理，以及这些规则如何从阿姆斯特朗公理推导。因为所有规则可以从阿姆斯特朗公理推导（因为完备性），所以这三个规则构成了一个替代的规则完备集。

6.11 [难]给定属性集 S 和函数依赖集 F ，我们说包含在 S 中的属性集 X 在 F 下具有非平凡集合闭包，只要 $X^+ - X$ 是非空的。让 F 是依赖的最小集，就是说， F 的最小覆盖是 F 。通过构造所有 F 中函数依赖左边的属性集合 X_i 的集合闭包，我们可以创建一个非平凡函数依赖基础 B ，然后创建 B 中的函数依赖 $FD_i: X_i \rightarrow (X_i^+ - X_i)$ 。

(a) • 说明如果 X 是不包含 F 中的某个函数依赖左边 X_i 的属性的集合，那么在 F 中函数依赖的闭包下 $X^+ = X$ 。

(b) 下面是一个看起来很有道理的假设：所有在 F 中的函数依赖 $W \rightarrow Z$ （这里 W 和 Z 没有共同属性）直接由非平凡函数依赖基础 B 产生，以至于如果属性 $A \in Z$ ，那么 $A \in \{X_i^+ - X_i\}$ 对 B 中某个具有形式 $X_i \rightarrow (X_i^+ - X_i)$ 的 FD_i 成立，其中 $X_i \subset W$ 。然而，这个单纯的设想是错误的。构造属性集 S 和函数依赖集 F 形成一个反例。在 F 中只需要两个函数依赖。

6.12 使用阿姆斯特朗公理和定理6.6.8的结果以及下面来自例6.6.4的函数依赖集：

(1) $A \rightarrow B$, (2) $C \rightarrow B$, (3) $D \rightarrow A B C$, (4) $A C \rightarrow D$

推导下面标号为(a)到(c)的函数依赖。使用一步接一步的方式来推导，每一步都用上面公理中的规则标记。

(a) • $D \rightarrow A B C D$

(b) $A C \rightarrow B D$

(c) $A C \rightarrow A B C D$

6.13 在例6.6.6中，函数依赖集 F 覆盖集合 G 。试说明相反的情况即集合 G 覆盖集合 F 。

6.14 (a) • 证明例6.6.9中的语句：例6.6.4中推导的函数依赖不是最小的。这可以通过寻找一个最小覆盖的步骤完成。

(b) 在(a)中，只有一个从例6.6.4推导的函数依赖需要改变以生成一个最小覆盖。使用推导函数蕴涵的一个简单应用来解释这个更改的必要性。

6.15• 在算法6.6.13中第二步，说明了用来决定 Y^+ 在从函数依赖集合 H 变到集合 J 时不变的测试同样暗示着 H^+ 和 J^+ 相同，如同在算法中定义的。

6.16 (a)复习例6.6.8并考虑在函数依赖集中做的每一个更改，然后试着使用阿姆斯特朗公理来说明每一个改变将生成相同的函数依赖闭包。

(b) 在例6.6.8，我们去除B之后的第三步中，证明如果我们再次考虑去除A，那么仍然是不恰当的。

(c) 你能够找到论据来说明为什么在第三步中考虑过该问题一次并除去了一个不同属性后再次考虑去除一个左边的属性是不必要的吗？

6.17 证明例6.6.10的陈述，即例6.6.3给定的数据库emp_info的函数依赖（集合F）构成了一个最小集。通过寻找最小覆盖的步骤证明。（使用在例6.6.7中介绍的推导表示。）

6.18 (a)假设给定了三个表T、T₁和T₂，且 $T=T_1 \bowtie T_2$, $\text{Head}(T_1) \cap \text{Head}(T_2) \rightarrow \text{Head}(T_2)$ 。

那么如果T被分解到（投影到）S₁和S₂，其中 $\text{Head}(S_1)=\text{Head}(T_1)$ 且 $\text{Head}(S_2)=\text{Head}(T_2)$ ，这确保S₁的行集合是T₁的行集合的一个子集，类似地，S₂的行集合是T₂中行集合的子集。给出一个例子说明为什么我们不能推导出S₁=T₁和S₂=T₂。

(b) 在(a)中将连接替换成外连接，找一个方法来扩展定义6.7.1使之处理无损外连接分解（Lossless Outer-Join Decomposition），并说明在(a)末尾的S₁=T₁和S₂=T₂。

6.19 (a)考虑下面给出的表T。首先说明这个内容分解成为两个表，满足 $\text{Head}(T_1)=A\ B$ 和 $\text{Head}(T_2)=B\ C$ 。当重新连接时，得到初始表。然而，可以通过从T中去除任何一行然后看分解结果来说明这个分解是无损的。

A	B	C
a1	b1	c1
a1	b1	c2
a2	b1	c1
a2	b1	c2

(b)• 如果T上没有函数依赖并且仍然如此分解的话，T的内容是如何说明定理6.7.4的“仅当”部分的？请加以说明。

(c) 证明定理6.7.4没有证明的“仅当”部分，即如果 $\text{Head}(T_1)$ 和 $\text{Head}(T_2)$ 都是 $\text{Head}(T)$ 的真子集， $\text{Head}(T)=\text{Head}(T_1) \cup \text{Head}(T_2)$ 且 $\text{Head}(T_1) \cap \text{Head}(T_2)$ 不函数决定 $\text{Head}(T_1)$ 或者 $\text{Head}(T_2)$ ，那么将T分解成T₁或T₂将不是无损的。

6.20 假设我们希望从一个数据项集合{A, B, C, D, E, F, G}（它们将成为表中的属性）构造一个数据库，且给定函数依赖集F如下：

$$F = \{(1) B C D \rightarrow A, (2) B C \rightarrow E, (3) A \rightarrow F, (4) F \rightarrow G, (5) C \rightarrow D, (6) A \rightarrow G\}$$

(a) 找出这个函数依赖集的最小覆盖，并命名这个集合为G。（使用例6.6.7中介绍的推导表示。）

(b)• 由包含所有这些属性的表T开始，进行一个无损分解，分解成两个表T₁和T₂，它们构成一个2NF分解。仔细列出每个表（T, T₁和T₂）的键以及每个表中存在的函数依赖。

(c) 继续分解成为3NF数据库。这个分解是BCNF的吗？

(d)• 使用算法6.8.8和函数依赖集G获得一个保持G中函数依赖的无损3NF分解。这个分解与(c)中的分解相同吗？

6.21 假设我们希望由一个数据项集合 { A, B, C, D, E, F, G, H } (它们将成为表中的属性) 构造一个数据库, 且给定函数依赖集 F 如下:

$$(1) A \rightarrow BC, (2) AB \rightarrow CDGH, (3) C \rightarrow GD, (4) D \rightarrow G, (5) E \rightarrow F$$

- (a) 找出这个函数依赖集的最小覆盖, 命名为 G。(使用例 6.6.7 中介绍的推导表示。)
- (b) 由包含所有这些属性的表 T 开始进行一个到 2NF 但不是 3NF 模式的无损分解。仔细列出每一个表 (T, T_1 和 T_2) 的键, 以及每一个表中存在的函数依赖。验证这个分解是无损的。说明为什么是 2NF 但不是 3NF 的。
- (c) 继续分解到 3NF 数据库。这个分解是 BCNF 的吗?
- (d) 使用算法 6.8.8 和 (a) 中的函数依赖集 G 构造一个保持 G 中函数依赖的无损 3NF 分解。这个分解和 (c) 中的分解相同吗?

6.22 • 重复习题 6.3 中的银行数据库关系设计, 但这次使用规范化方法。这需要你给出函数依赖集的真子集, 且应当确保你的回答有意义。比较所得的结果和 E-R 方式的结果。

6.23 假设我们希望由一个数据项集合 { A, B, C, D, E, F, G, H, J } (它们将成为表中的属性), 和给定了函数依赖集 F 构造一个数据库 F 构成一个最小覆盖。

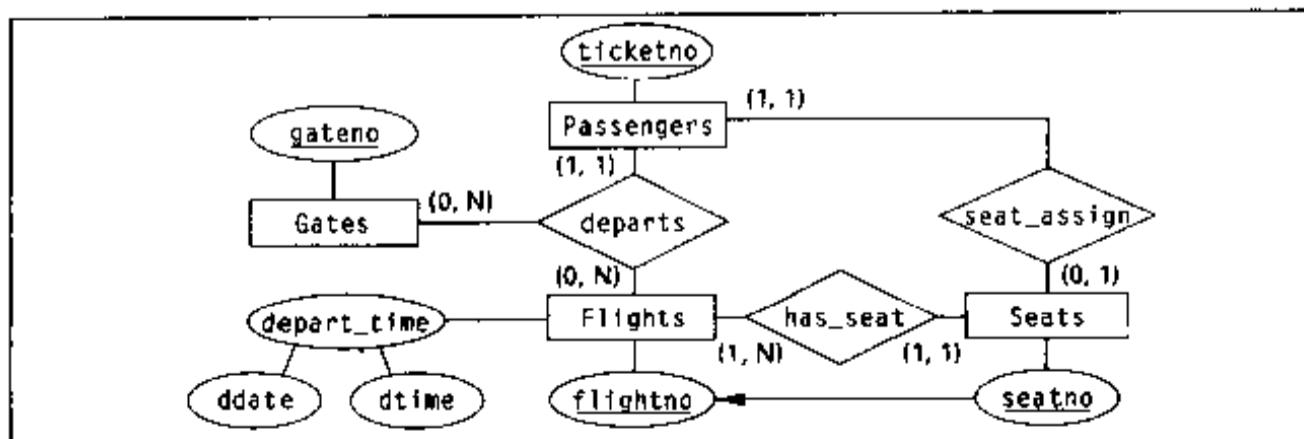
$$(1) AB \rightarrow CD, (2) A \rightarrow E, (3) B \rightarrow FH, (4) C \rightarrow G, (5) D \rightarrow B, (6) G \rightarrow C, (7) H \rightarrow I.$$

- (a) 创建表 T 的一个保持函数依赖的 3NF 无损连接分解。使用在文中提供的算法来决定分解出的表的标题, 并在每个表的一个键加上下划线。
- (b) 习题 (a) 中的一个或多个表有两个候选键, 找出它们并证明它们是键。
- (c) 证明这个分解是无损的。
- (d) 你所创建的分解是 BCNF 分解吗? 说明是或者不是的原因。

6.24 再次考虑 6.4 节中的飞机订票数据库。注意到旅客在登机时会集中在登机口, 看起来我们可以用联系 Passengers、Gates 和 Flights 的一个三元联系 departs 来替换图 6-14 中的两个二元联系 marshals 和 travels_on。

我们注意到这个联系是 1-N-N 的, 因为一个登机口有多个航班和很多旅客, 它的实体在联系里有 max-card=1。如同在 6.2 节末尾提到的, 这意味着我们可以在一个实体表中使用外键来表示这个联系。特别地, 因为实体 Passengers 以 max-card=1 参与, 所以我们为登机口和航班附加外键来标识对一个顾客而言唯一的登机口和航班。

- (a) 把这个具有三元联系 departs 的 E-R 设计转化成关系表。
- (b) 这个关系设计与 6.4 节末尾的设计有什么不同? 说出哪一个更好并证明你的回答。(你可以试图提出和练习 6.3 末尾类似的问题。)
- (c) 创建飞机订票数据库的函数依赖集, 然后按照算法 6.8.8 进行规范化, 达到一个关系表设计。这对 (b) 中的问题有什么启发吗?



- (d) 如果你创建了正确的函数依赖，你应当发现对于登机口没有独立的表。把gateno作为另一个表的属性所带来的问题是删除异常：如果最后一次航班从某个登机口起飞，那么将没有那个登机口的任何记录保留下来。但是似乎规范化过程所说的是似乎没有任何理由来记住这个登机口存在。我们可以想象，加入一个新的属性gatecap到设计中，它表示提供给在一个登机口等候的顾客座位容量。现在，规范化设计将试图为登机口另外创建一个分离的表。解释为什么。
- (e) 注意(d)中对登机口和容量的设计，我们可以很容易用程序逻辑来为一次航班分配一个登机口，这个登记口在起飞前的那个小时内还没有被分配给航班，并且有座位来容纳旅客。但假设所有登机口有相同的座位容量，我们仍然想为航班分配登机口，现在(d)中提到的删除异常成为实际的问题。这对规范化方法意味着什么？

第7章 完整性、视图、安全性和目录

第6章中，我们已经学习了如何通过设计一系列关系表和函数依赖为一个公司建立数据模型。本章介绍DBA将设计转换成物理形式所采用的步骤的细节。第4章中我们已经详尽介绍了对象-关系表，所以本章的重点是关系表。如果想了解有关对象关系模型的更详细内容，特别是从Create Table到Create Object Table的扩展，可参阅附录C，那里给出了SQL语句的一般语法。

本章第一节着重介绍如何创建物理（关系）表，以及如何给列加上完整性约束以保证其和现实模型相一致。完整性约束可以保证第6章中的逻辑设计阶段定义的数据相关性不被错误的SQL更新语句破坏，也就是说，我们希望在出现更新错误时保持数据的完整性。本章的后面几节将详细介绍DBA需要对大型的、访问频繁的数据库做的几项工作，以使终端用户能够简单有效地获得所需的信息。首先，DBA需要定义表、约束，并装入数据。另外，本章还会提到一种提供数据视图的服务，将物理表格重新组织成不同的表形式（这些表格物理上并不存在），以简化对数据的访问。DBA还需提供安全性，保证只有授权用户才能读取或更新某些保密的数据。视图的结构和数据库的其他对象是通过一系列系统定义的表提供给DBA的。这些表叫做系统目录，在本章的最后介绍。DBA同时还应该负责数据库的性能，如提供索引来加速对表的访问及其他类型的调谐工作，这些将在第10章讨论。

对企业来说，数据库是至关重要的，因此DBA的责任非常重大，他必须了解所有用户的需求，包括终端用户和应用程序员。本文的目的是给出DBA所需的一些基本概念。本章介绍了DBA进行工作时所需的命令和特征。通篇我们集中在从操作性的观点来看待数据库概念，这意味着重点放在使DBA能够做出明智、正确的决定，而不是放在系统程序员设计实现数据库系统软件产品的内部细节上。操作性的方法并不代表我们定义的概念不够严格，只是说我们的重点放在效果上，例如，我们介绍的是B树的结构，而不是编制B树程序的细节。实事求是地说，操作性应该比系统的其他问题优先考虑。一个编写数据库程序的程序员如果对这种考虑对于DBA的重要性没有正确的认识，将会遇到严重的障碍。

我们曾经了解了DBA创建表、装入数据时用的一些命令。在第3章中，我们介绍过SQL语句

```
create table customers (cid char(4) not null, cname varchar(13),
    city varchar(20), discnt real, primary key(cid));
```

我们将会看到，这条语句只给出了整个语法的一小部分。

7.1 完整性约束

完整性约束是表的创建者出于第6章中提到的各种考虑设计的一个规则，所有的SQL更新语句都必须满足这个规则。举例来说，如果我们规定数据库CAP中表customers的cid列的值不能重复（作为候选键或主键），那么若有一行的cid列的值与已有的cid列的值重复，该行就不能插入表customers中。这即是所谓的现实世界的忠实的反映。实际上，根据我们将要介绍的完整性约束，所有违背完整性约束的更新操作——Insert、Update或Delete——都是不

能执行的。我们在第6章中看到，一些完整性约束是在设计阶段产生的，例如，某些作为表的键的一部分的列不能为空值，而且键必须是唯一值。在讨论完整性约束的优缺点之前，我们先看看它们在商用数据库系统中是怎样实现的。我们先从SQL标准开始，从中我们可以看出完整性约束在Create Table语句中是怎样实现的。

1. Create Table语句中的完整性约束

最基本的Create Table语句如图7-1所示。记得在第3章中，我们把基本SQL作为一个通用的标准，加上了一些公共的标准，如X/Open和SQL-99。基本SQL的语法仅限为大部分数据库产品能够实现的部分。特别地，我们提供一些在ORACLE、DB2 UDB和INFORMIX三种产品中非常普遍的特征。值得一提的是创建表的用户（通常为DBA）即为表的所有者，这对我们后面讨论的数据库安全性来说有特殊的意义。

```
CREATE TABLE [schema.]tablename
  ((columnname datatype [DEFAULT (default_constant|NULL)] [col_constr {col constr...}])
   | table_constr)-- choice of either columnname-definition or table_constr
  (, (columnname datatype [DEFAULT (default_constant|NULL)] [col_constr {col constr...}])
   | table_constr)
  ...);                                -- zero or more additional columnname-def or table.constr
```

约束单个列的Col_constr形式如下：

```
{NOT NULL |                                -- this is the first of a set of choices
[CONSTRAINT constraintname] -- if later choices used, optionally name constraint
    UNIQUE                                -- the rest of the choices start here
    | PRIMARY KEY
    | CHECK (search_cond)
    | REFERENCES tablename [(columnname)]
        [ON DELETE CASCADE]}
```

约束多个列的table_constr形式如下：

```
[CONSTRAINT constraintname]
  (UNIQUE (columnname [, columnname...]))      -- choose one of these clauses
  | PRIMARY KEY (columnname [, columnname...])
  | CHECK (search_condition)
  | FOREIGN KEY (columnname [, columnname...])  -- following is all one clause
    REFERENCES tablename [(columnname [, columnname...])]
        [ON DELETE CASCADE]
```

图7-1 基本SQL的Create Table语法

从图7-1可以看出，在表名前可出现一个可选的语法元素[schema.]。7.4节中对模式(schemas)进行了介绍，它被用来区分数据库中同名对象。用户在数据库中为表或其他对象命名时一般会加上模式名(与其用户名相同)。例如，用户eoneil可能有一个叫做customers的表，加上模式后的全名为eoneil.customers，同时在数据库中还有叫做poneil.customers的表。当用户eoneil用customers这个名字时，指的是表eoneil.customers，用户poneil也是一样，这样就不会混淆了。如果用户eoneil有相应的权限，她可以使用poneil.customers访问不属于她的模式的表。

下面我们先定义图7-1的Create Table语句中的一些子句，再给出一个CAP数据库的例子。

然后，我们说明在ORACLE和DB2 UDB中的一些特殊的语法扩展。

定义7.1.1 Create Table命令的子句 图7-1的Create Table命令先要给表命名，包括表名和可选的模式名。如果模式名没有给出，将缺省地把执行Create Table命令的用户名作为模式名。表名之后的圆括号中是用逗号分隔的列定义或表约束的序列。列定义和表约束可以以任何顺序出现。每个列定义包括列的名字、数据类型和可选的DEFAULT子句。当SQL的Insert语句没有给出该列的值时，就用这个子句的缺省值。（注意，装入大量数据的命令，如Load，没有包含在SQL标准中，所以不要求给出缺省值。尽管有些命令给出了，但这不是必要的。）提供的缺省值可以是正确数据类型的常值，由defaultvalue或NULL给出。

每个列定义同样允许有列约束，在图7-1中用col_constr表示，col_constr包含一些可选的子句序列，具体解释如下。 ■

定义7.1.2 列约束 图7-1中Create Table语句中出现的col_constr子句是可选的，对应于单个的列。

NOT NULL的情况已经解释过了，它表示列中不能出现空值。如果col_constr为NOT NULL，那么DEFAULT子句不能指定为NULL。如果在列定义中既没有DEFAULT子句，也没有NOT NULL子句，那么缺省地使用DEFAULT NULL。

CONSTRAINT constraintname子句可以为除NOT NULL以外的列约束命名，这样，我们可以用后面要介绍的Alter Table语句来去掉某些约束，而不用重建整张表。

即使没有NOT NULL，也可以使用UNIQUE约束子句，这时，该列中所有的非空值必须是唯一的，但允许多个空值同时存在。如果一列既被声明为NOT NULL又被声明为UNIQUE，我们把它叫做候选键。

PRIMARY KEY子句指定一列为主键——被REFERENCES子句中的另一个表缺省引用的候选键。有PRIMARY KEY约束的列被隐式地定义为NOT NULL和UNIQUE。在任何Create Table语句中至多只能有一个PRIMARY KEY子句。UNIQUE子句和PRIMARY KEY子句不能用在同一列上，但PRIMARY KEY子句和NOT NULL子句可以一起用。实际上，一些老的数据产品要求定义为PRIMARY KEY的列必须是NOT NULL的。现在的标准已经不这么要求了，但为了保证老产品的健壮性，我们还是常常照这个习惯做。

如果出现CHECK子句，那么每一行该列必须包含满足特定的搜索条件的值。（在X/Open标准和DB2 UDB中，搜索条件只允许包含常数值和当前行特定列的值的引用，其他列的引用或集合函数都是不允许的。ORACLE允许引用同一行中的其他列。）

如果一列用REFERENCES tablename [(columnname)]子句定义，该列中的值或者为空或者是在被引用表的一列中出现的值；被引用的表中的列或者是该表的单列主键或者是在REFERENCES子句中可选的列名处指定的列。被引用表中指定的列的值必须是唯一的，否则包含REFERENCES子句的Create Table语句将失败（但空值是允许的）。如果REFERENCES子句后用的是ON DELETE CASCADE子句，那么当被引用表中一行被删除时，被引用表（有REFERENCES子句的表）中的行引用的行都要被删掉然后引用表中的这些行被删除。如果用的是NO DELETE子句，那么被引用的行不允许被删除。 ■

例7.1.1 下面是一个可能的customers表的Create Table语句。

```
create table customers (cid char(4) not null unique, cname
```

```

    varchar(13), city varchar(20),
    discnt real constraint discnt_max check (discnt <= 15.0));

```

这里，`cid`被定义为表`customers`的候选键，因为它既有`NOT NULL`子句，又有`UNIQUE`子句；尽管`cid`要求每行有唯一的值，这并不代表它就是主键；我们很快就会知道其含义。这个`Create Table`语句同时限定`customers`表中每一行的`discnt`值不能超过15.0，这个约束被命名为`discnt_max`，以便以后可以用`Alter Table`语句去掉这个约束。但我们不能再把这个约束加回来，而对命名了的表约束就可以。我们将会看到，可以把这个约束改成更普遍的表约束形式。 ■

定义7.1.3 表约束 下面列出了图7-1的`Create Table`语句中的`table_constr`子句中除了指定列的`NOT NULL`形式外，`col-constr`形式仅是`table_constr`的特例。

`UNIQUE`子句和`col-constr`中的`UNIQUE`含义是相同的，但在表中可以规定多个列的组合值唯一。因此这是为表格指定多列候选键的一种方法（虽然我们可以只指定一列，再用同样方法以表约束代替列约束）。

`PRIMARY KEY`子句指定一组非空的列作为主键——被`REFERENCES`子句中的另一个表缺省引用的候选键。`PRIMARY KEY`子句中的每个列都必须是`NOT NULL`的。在任何`Create Table`语句中，至多只能有一个`PRIMARY KEY`子句，不管该子句是列约束还是表约束。

如果表格中的几列被定义为`UNIQUE`，而且其中的每一列都单个定义为`NOT NULL`，我们称它们为候选键。

表约束中的`CHECK`子句与列约束中的`CHECK`子句类似，用来限制一组列的值；搜索条件可以取同一表中正在进行`Update`或`Insert`操作的同一行的任何列的值。与在列约束中讨论过的一样，在X/Open标准中，子查询或集合函数都是不允许的。扩展的SQL-99标准的一个显著特征就是允许在`CHECK`子句中包含子查询，并且这种扩展能力是很强的。但大多数的数据库产品，包括ORACLE和DB2 UDB，都不支持这种扩展。

`FOREIGN KEY ...REFERENCES...[ON DELETE...]`子句是一个具有这个序列中的关键字的单个子句。`FOREIGN KEY`列名列表指定正在创建的表中的列序列，如果表中某行的这几个列的值都非空，那么这些列的值被约束为与由`REFERENCES`子句指定的另一个表中的某行的对应列的值相等。当被引用表中的列构成了主键时，`REFERENCES`子句不必给出列的列表。如果`FOREIGN KEY`列名列表中的列值中的一个在某行的值为空，那么对那一行的该列值没有限制。

可选的`ON DELETE CASCADE`子句指出，当被引用表的行被删除时，导致引用表中所有引用它的行也要被删除。这个行为看上去有点太严格了，但只有这样才能满足约束。如果没有这个子句，被引用的行是不能被删除的。

在本节的稍后部分有关于主键、外键和参照完整性等概念的详细描述。 ■

下面是一个说明这些概念的例子。

例7.1.2 我们给出在CAP数据库中创建`customers`和`orders`表的命令。表定义基于图7-2所示的E-R图，这个E-R图是第6章后习题6.2的答案。前面讲过，属性A和实体E之间连线上的(x, y)表示x = min-card(A, E), y = max-card(A, E)。特别地，x = 0表示这一列中允许空值出现，x = 1表示该列强制参与，在`Create Table`语句中，该列必须用`NOT NULL`子句定义。每个实体的标识符属性都在E-R图中用下划线标出了，这通常对应于`Create Table`语句中

的PRIMARY KEY子句。在Create Table语句中，我们还加上了一些CHECK子句，这在E-R图中没有对应的部分。表orders中的FOREIGN KEY...REFERENCES子句，我们在本节的后面谈到参照完整性时再讨论。

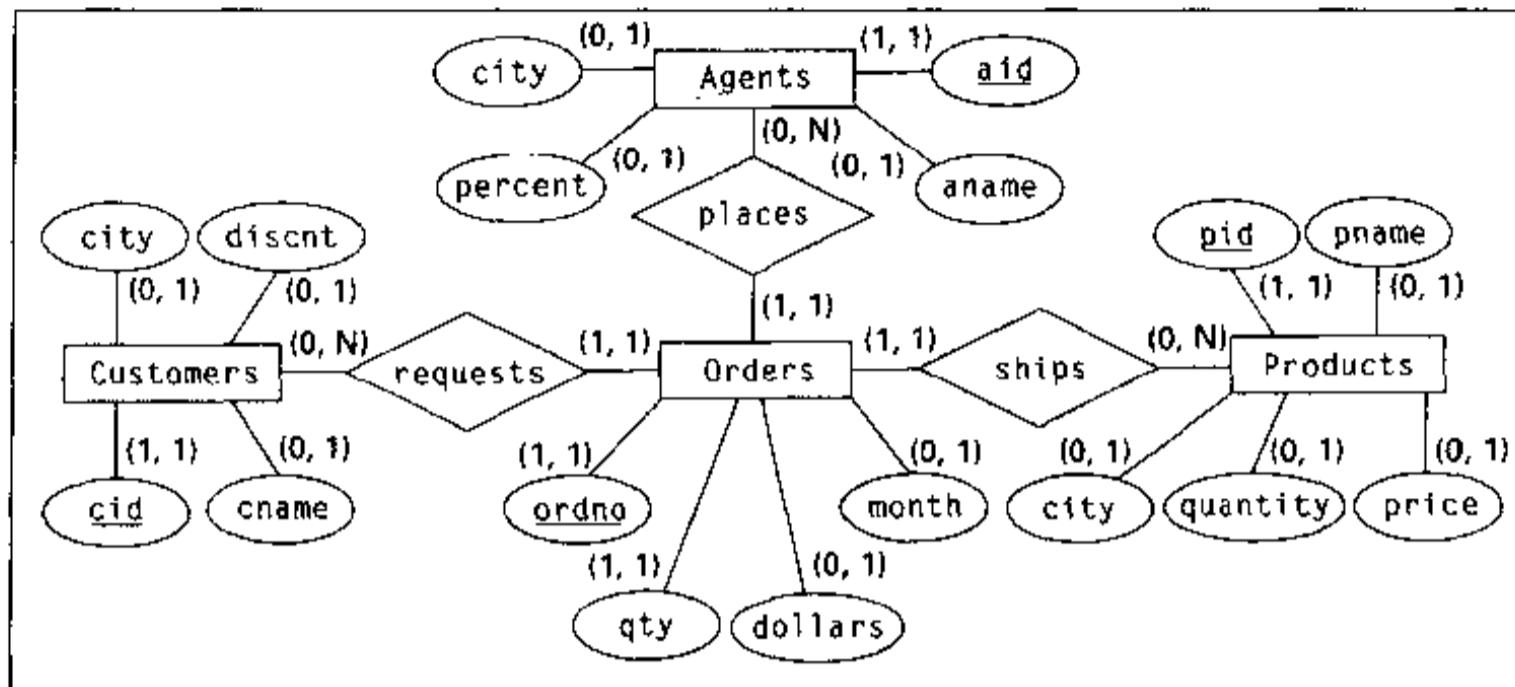


图7-2 CAP数据库的E-R图

```

create table customers (cid char(4) not null, cname varchar(13),
    city varchar(20),
    disct real constraint disct_max check(disct <= 15.0),
    primary key (cid));

create table orders (ordno integer not null, month char(3),
    cid char(4) not null, aid char(3) not null, pid char(3) not null,
    qty integer not null constraint qtyck check(qty >= 0),
    dollars float default 0.0 constraint dollarsck check(dollars >= 0.0),
    primary key (ordno),
    constraint cidref foreign key (cid) references customers,
    constraint aidref foreign key (aid) references agents,
    constraint pidref foreign key (pid) references products);

```

表orders的约束，将在例7.1.3中讲到如何从E-R图中产生参照完整性时再解释。 ■

在下面的小节中，我们讨论在数据库产品ORACLE和DB2 UDB中如何加入约束，这与基本SQL标准略有不同。

(1) ORACLE的Create Table语句

ORACLE的Create Table语句除了支持如图7-1所示的基本SQL语法中的所有完整性约束子句(col_constr和table_constr子句)外，还加入了如图7-3所示的ENABLE和DISABLE子句。ORACLE还有一点与基本SQL语法不同，它允许在NOT NULL之前出现可选的约束名，这样列约束NOT NULL也可以用Alter Table语句去掉。此外，ORACLE中除了有NOT NULL外，还有NULL约束。

注意到图7-3中Create Table语句有两种形式。每种形式都有disk storage and other clauses(磁盘存储及其他子句)，定义表的每一行在磁盘上如何存储。这些子句将在第8章中讨论。

图7-3 Create Table的第二种形式中，有一个AS子查询语句，用户可以用对其他表的查询

结果来创建表。新表的列名和数据类型可以从子查询的目标列表中继承得到（假定列的别名可以在表达式中出现），这样Create Table命令中的列名和数据类型就可以省略了。注意列约束和表约束不能被继承，即使是单个表的子查询也不行。如果用户想定义这样的约束作为Create Table语句的一部分，必须指定列名（但数据类型还是可以省略的）。

```

CREATE TABLE [schema.]tablename
  (... as in Basic SQL, except with [NOT] NULL)
  [disk storage and other clauses (not covered, or deferred)]
| CREATE TABLE tablename
  (... as in Basic SQL, columnnames and datatypes can be left out if no
    constraints are specified)
  [disk storage and other clauses (not covered, or deferred)]
  AS subquery;

```

约束单个列值的Col_Constr形式如下：

```

... as in Basic SQL
[ENABLE and DISABLE clauses for constraints];
[storage and transaction specifications];

```

一次约束多个列的table_Constr形式如下：

```

... as in Basic SQL
[ENABLE and DISABLE clauses for constraints];
[storage and transaction specifications];

```

图7-3 ORACLE关系Create Table语法(无对象关系扩展)

注意，ENABLE和DISABLE子句允许用户定义一个约束，然后在创建表前使它无效；如果没有一个子句出现DISABLE约束，那么缺省值为ENABLE。用Alter Table命令可以使约束重新有效。这种ENABLE/DISABLE能力的唯一问题是，它不适用于其他产品，我们只有在Alter Table命令中使用别的方法才能近似地实现这一功能。想了解ORACLE中Create Table语句的更多细节，请参阅附录C或《ORACLE8 Server SQL Reference Manual》(《ORACLE8服务SQL参考手册》)(参见本章最后的“推荐读物”[13])。

(2) INFORMIX的Create Table语句

INFORMIX的Create Table语句支持如图7-1所示的基本SQL语法中的所有子句，只有一点不同：INFORMIX要求所有的列定义在表约束定义之前出现，而图7-1允许两者以任意顺序出现。如图7-3所示，INFORMIX的Create Table语句中也有对针对产品的“disk storage and other clauses”。它有与ORACLE形式相同的对约束的ENABLE/DISABLE，在这里我们没有把它们列出。想了解INFORMIX的Create Table语句的更多细节，请参阅附录C或《INFORMIX Guide to SQL》(《INFORMIX SQL指南》)(见本章最后的“推荐读物”[8])。

(3) DB2 UDB的Create Table语句

DB2 UDB的Create Table语句支持基本SQL语法中的所有子句。它同样有对针对产品的“disk storage and other clauses”，我们将在第8章中介绍。表约束的FOREIGN KEY REFERENCES子句有更多的项可供选择(如后面的图7-6所示)。想了解DB2 UDB中Create Table语句的更多细节，请参阅附录C或《DB2 Universal Database SQL Reference Manual》(《DB2 UDB通用数据库SQL参考手册》)(见本章最后的“推荐读物”的推荐读物[4])。

2. 主键、外键和参照完整性

这一部分，我们将解释图7-1中的FOREIGN KEY...REFERENCES子句，以及隐含的完整性约束（称为参照完整性约束）是如何在更新语句上强制实施的。我们用CAP数据库中的一个例子来引出定义7.1.4中的参照完整性的定义。

例7.1.3 参照完整性 在如图7-2所示的CAP数据库的E-R图中，所有的关系requests, places和ships都是多对一的，Orders实体与多个实体有联系（参见定义6.2.2）。一个顾客可以有多个订单，每个代理商也有多个订单，每个产品可能在多个订单中出现，但一个订单只能对应一个确定的顾客、代理商和产品。因此在6.2节的转换规则4中，orders表应该包含一个外键的列来表示与Orders实例相关的每个Customers, Agents和Products实例。例如，例7.1.2中Create Table语句的orders表包含一个外键cid，说明Customers实例需要这行特定的订单来表示。特别注意到orders表中的列cid并不表示Orders实体的属性，而是表示requests关系的一个实例。

这是一个非常重要的概念，所以我们重复了好几次，每次略有不同：orders表每一行的每个特定的cid值都对应着一个联系着Orders实体和Customers实体的联系实体。参照完整性约束要求我们避免外键指向其他表中不存在的主键这种“悬挂”引用，这就保证了表orders中每一行的联系实例都是有意义的。所以orders表中的cid值也必须在customers中存在。例7.1.2中，这个约束是在创建orders表时，用orders的Create Table语句中的foreign key (cid) references customers子句保证的。

cid还有一个约束是基于图7-2中Orders实体必须在requests联系中强制参与（因为连接Orders和requests的连线上的标识的min-card等于1）。因此，orders中的每一行的cid值必须非空（这与参照完整性有明显不同，参照完整性要求每个非空的cid值必须引用Customers中一个真实存在的cid值）。例7.1.2中的强制参与约束是在orders的Create Table的cid定义后的NOT NULL子句的定义实现的。 ■

再重复一下：参照完整性定义了一条规则，即CAP数据库中表orders没有哪个行包含cid, aid或pid值，除非它指表customers, agents或products表中存在。这条规则基本的意思是当我们往orders中插入行时，应该避免出现如下的申明“尽管这条命令中引用的是一个不存在的代理商（或产品、顾客），但一有机会我就会在agents表中插入一行，使这个引用变得有效”。我们强调必须先在agents中插入行，再在orders中对它进行引用。我们使用完整性约束是出于逻辑设计的基本考虑，但在实际应用中非常有效。如果有人试图在orders中加入一个不存在的引用值，这可能意味着某人在往表orders中输入数据时有错误。例如，考虑将如下所示的一行插入到表orders中。

ordno	month	cid	aid	pid	qty	dollars
1011	jan	c001	a0@	p01	1000	450.00

在agents的主键中，没有aid值等于“a0@”的；这可能意味着在输入orders的aid列时出错了（输2的时候误按了Shift键，所以得到了“@”）。似乎并没有理由要求cid, aid或pid在orders中第一次出现时就是正确的；但我们有理由规定一个新的顾客、代理商或产品必须先在相应的被引用表中出现，这样有利于我们发现重要的错误。当然参照完整性约束会降低性能：加上像这样的三个参照完整性规则后，在表orders中执行插入操作所用的资源

将大受影响，因为每一次插入操作都需要查找三个表的主键。

现在让我们澄清思路来下一个定义：参照完整性约束要求数据库某个表格中的外键值必须与另一个表格中的主键值相匹配（也可能引用的是同一个表，例如表employees的主键是eid，通过列mgrid引用其他一些雇员的eid）。对于空值我们怎么办呢？我们知道空值是不能在主键中存在的（参见2.4节的RULE 4 实体完整性规则，以及定义7.1.2最后的PRIMARY KEY子句）。对于外键中的空值，我们应该采用什么样的规则呢？在DB2 UDB（如图7-6所示）的ON DELETE表约束子句中有一个动作是，当主键的值被删掉时，将一行中的外键值置为空。也就是说，数据库产品允许空值存在于外键中，但我们必须仔细地给出一个定义。

定义7.1.4 外键，参照完整性 如果表T1中任意行的F值的组合，至少包含一个空值或与被引用表T2中为候选键或主键的列集合P的组合值匹配，则列集合F被定义为外键。换句话说，如果表T1的每一行中F的列都满足（1）至少有一列为空值（如果该列允许为空），（2）如果不包含空值，就必须与表T2中某行的相应P的组合值相等，则参照完整性约束是有效的。 ■

注意，外键中的某些列可能还参与构成这个表的候选键或主键，因此不能为空。也就是说它们出现在Create Table语句的NOT NULL 或PRIMARY KEY子句中。但是，定义7.1.4暗示，如果外键支持的关系是可选参与，则外键中至少有一列的值必须为可空的，以表示关系实例的存在；如果由于候选键上的SET NULL动作删除了主键，使联系实例不再有效，那么这个可以为空的列中出现空值意味着整个外键是无效的，即为空。

例7.1.4 在2.2节中我们曾经说过一个属性的域通常被看做是将所有可能值枚举出来得到的集合。但实际上大部分数据库都不支持这种枚举类型。例如，customers，agents和products中的city列只是简单地被约束为字符串型，因为要列出所有正确的城市名是不可能的。但如果要创建一张叫做cities的新表，我们还是可以提供枚举类型的功能，列出所有可以接受的城市名。（在不同的州或国家中，同一个城市名可能出现多次，假定我们先不考虑这些；为简单起见，在原先那个例子的表中，我们不考虑州和国家，这样，如何建立城市 - 州的对应的问题就非常明了了。）

```
create table cities (city varchar(20) not null,
    primary key (city));
create table customers (cid char(4) not null, cname varchar(13),
    city varchar(20), discnt real check(discnt <= 15.0),
    primary key (cid),
    foreign key (city) references cities);
```

表cities中的city值是唯一的，可以作为主键，而表customers中的city值是一个从键，必须与表cities中的某个（主键）值匹配。这个约束保证了表customers中的城市名都是在表cities中列出来的，这样就有了枚举域的效果。 ■

完整性约束保证了在执行写得不好的Update语句时数据的完整性。外键约束对行更新施加了哪些限制呢？应该做哪些测试以保证一致性呢？在图7-1的ON DELETE子句中，动作CASCADE指定，如果被引用的主键表中有一行被删除，那么其他表中引用该行作为外键的所有行都必须被删除，但可能发生的情况不止这一种。考虑用于行更新的三种SQL语句——Insert、Delete和Update，以及可能被改变的两列——一个表中的主键和另一个表的引用外键。用于检查是否违背了完整性约束的必要测试如图7-4的表所示。

	Insert	Delete	Update
主键(被外键引用)	no test	test-1	test-2
外键(引用主键)	test-3	no test	test-4

图7-4 参照完整性测试的表格

表中标识为“no test”的单元表示不会破坏完整性约束的情况。例如，如果外键的所有值组合与主键的值组合对应，那么插入包含一个新的主键值的行就不会破坏这一特性。相反，表中标识为“test-1”至“test-4”的单元表示，在这种情况下，系统需要进行测试以保证这类更新操作不会破坏参照完整性。例如，当从主键中删除一行时(test-1)，我们可能删掉了一个正被其他表的某些行作为外键引用的主键值(这由我们已经讨论过的ON DELETE子句解决)。对主键的更新同样可能使正在被引用的主键值改变，DB2 UDB中的ON DELETE子句(后面将讨论，如图7-6所示)就是被扩展来处理这种情况的。Full SQL-99标准中有一个ON UPDATE子句用来控制这种情况，DB2 UDB中同样提供了这样的ON UPDATE子句。

显然，当向外键表中插入一行时(test-3)，我们必须验证它是否引用了一个合法的主键，同样，对外键所在列进行修改时(test-4)也需要验证。当插入一行时，要执行这些测试是可以理解的。如果未通过测试，缺省的动作是NO ACTION，即不允许执行那些可能产生非法外键的插入和修改操作。从执行的测试类型来看，只对很少的一些情况应用参照完整性可能是出于效率的考虑。例如，当在例7.1.4的表customers中插入一行时，系统附带的重要责任是查表cities的city值，该表可能包含几千条记录。这确实是DBA需要的功能，但必须慎重考虑，因为为此付出的代价是在一定的硬件配置下更新能力的下降。图7-5列出了我们已经讨论过的SQL语句和它们与测试、动作之间的关系。

	Insert	Delete	Update
主键(被外键引用)	无须测试	ON DELETE... 或 缺省 NO ACTION	ON DELETE... 或 ON UPDATE... 或 缺省 NO ACTION
外键	必要时语句失败	无须测试	必要时语句失败

图7-5 给出特定动作的参照完整性测试和SQL子句

3. 不同产品中的外键约束

大部分的数据库产品都支持如图7-1所示的基本SQL的 FOREIGN KEY子句：

```
| FOREIGN KEY (columnname [, columnname...])
  REFERENCES tablename [(columnname [, columnname...])]
  [ON DELETE CASCADE]
```

对应于多列的FOREIGN KEY子句在被引用表中只能特指一列。在上面的讨论中，我们注意到当ON DELETE子句没有出现时，缺省的动作是NO ACTION，这意味着当违反约束的删除操作失败时，动作不能发生。图7-6显示X/Open、Full SQL-99和DB2 UDB允许约束中出现显式的NO ACTION说明，而ORACLE和INFORMIX中只有无ON DELETE子句时才提供。

ORACLE, INFORMIX		DB2 UDB	X/Open, Full SQL-99
ON DELETE...	CASCADE SET NULL	<u>NO ACTION</u> CASCADE SET NULL RESTRICT	<u>NO ACTION</u> CASCADE SET NULL SET DEFAULT RESTRICT (SQL-99 only)
无 ON DELETE	NO ACTION 作用	NO ACTION 作用	NO ACTION 作用

图7-6 各产品和标准中外键约束可用的ON DELETE动作

DB2 UDB支持除SET DEFAULT外完全SQL-99定义的所有ON DELETE语法。例如，表orders的外键cid引用的是表customers的主键。当执行Delete语句将表customers的一行删除，而该行的cid值正被表orders的某些行引用时，应该相应地做些什么动作呢？DB2 UDB允许的四个动作是NO ACTION、CASCADE、SET NULL和RESTRICT。如果用CASCADE，当表customers中包含某个给定cid值的行被删除时，表orders中所有引用该cid值的行都必须被删除，与这些行有级联的外键依赖关系的行也要删除。如果用NO ACTION或RESTRICT，将不允许删除表customers中的行（这两者之间有细微的差别）；如果没有特指，NO ACTION将作为缺省动作。如果用SET NULL，那么当表customers的行被删除后，表orders中引用这一行的cid值将被置为空。（表orders中的列cid不能用NOT NULL或PRIMARY KEY说明。）如果FOREIGN KEY定义中不包含ON DELETE子句，则缺省使用ON DELETE NO ACTION选项。在SQL标准（还有DB2 UDB）中，ON DELETE NO ACTION也是未给出ON DELETE子句时的缺省动作。

4. Alter Table语句

Alter Table语句允许DBA改变原先在Create Table语句中定义的表的结构，加入或改变列、加入或删除各种约束等。执行Alter Table语句的必须是表的所有者（在有些产品中，也可以是有修改权限的用户），可能还需要有其他权限。我们将在本章的后面一节介绍这类权限。

在老的标准中没有定义Alter Table语句。因此，ORACLE、DB2 UDB和INFORMIX提供的一些特性在语法上略有不同。鉴于Alter Table语句在实际应用中不断完善，没有一个标准能对它进行很好的描述，我们也不给出它的基本SQL标准的形式，而是给出厂商提供的形式。

可以看到在图7-7、7-8和7-9的Alter Table语句中，约束的名字的用处。如果表约束被命名，表所有者可以用这个名字DROP该约束。用ADD子句可以加上一个新定义的表约束。ADD子句的操作对象不能为列约束，不过列约束实际上是表约束的特例，所以可以用这种方式加入大多数这样的约束（除NOT NULL外）。在ORACLE中，可以用ENABLE和DISABLE子句启用和禁用一条约束，即打开和关闭约束，只要它在数据库中有定义。在这里，我们不研究这个特征，因为它不可在数据库产品之间移植。一种可移植的方法是用DROP去掉一个有名字的约束，以后再用ADD把它加回来。

ORACLE、DB2 UDB和INFORMIX的功能基本相同，但语法上有些差异。例如，图7-7所示的ORACLE的ALTER TABLE语句，要求ADD和MODIFY子句中的“columnname datatype”元素用大括号括起来；图7-9的INFORMIX形式既允许用括号将一列由逗号分隔的数据类型括起来，也允许单个元素不用括号括起来；而图7-8的DB2 UDB不需要括号。ORACLE和INFORMIX都有删除一列的能力（这是现今的所有标准都支持的），但DB2 UDB却不提供。

ORACLE有修改一列的能力。(DB2 UDB同样有一种方法通过改变列的最大长度来改变列的属性，但图7-8中没有包含这条语句。)

```

ALTER TABLE tablename
  [ADD ({columnname datatype
        [DEFAULT {default_constant|NULL}] [col_constr {col_constr...}] | table_constr}
        [, {columnname datatype
        [DEFAULT {default_constant|NULL}] [col_constr {col_constr...}] | table_constr
        ...}])
      -- zero or more added columnname-def or table_constr]
  [DROP {COLUMN columnname | (columnname {, columnname...})}]
  [MODIFY {columnname data-type
        [DEFAULT {default_constant|NULL}] [[NOT] NULL]
        [, columnname data-type
        [DEFAULT {default_constant|NULL}] [[NOT] NULL]
        ...}])
  [DROP CONSTRAINT constr_name]
  [DROP PRIMARY KEY]
  [disk storage and other clauses (not covered, or deferred)]
  [any clause above can be repeated, in any order]
  [ENABLE and DISABLE clauses for constraints];

```

图7-7 ORACLE中Alter Table语句的形式

```

ALTER TABLE tablename
  [ADD [COLUMN] columnname datatype
    [DEFAULT {default constant|NULL}] [col_constr {col constr...}]]
  [ADD table_constr]
  [DROP CONSTRAINT constr_name]
  [DROP PRIMARY KEY]
  [repeated ADD or DROP clauses, in any order]
  [disk storage and other clauses (not covered, or deferred)];

```

图7-8 DB2 UDB中Alter Table语法的形式

```

ALTER TABLE tablename
  [ADD new_col | (new_col {, new_col...})]
  [DROP columnname | (columnname {, columnname...})]
  [ADD CONSTRAINT table_constr | (table_constr {, table_constr...})]
  [DROP CONSTRAINT constraintname | (constraintname {, constraintname...})]
  [repeated ADD or DROP clauses, in any order]
  [disk storage and other clauses (not covered, or deferred)];
约束单个列值的新_col形式如下:
  columnname datatype
    [DEFAULT {default_constant|NULL}] [col_constr {col_constr...}]

```

图7-9 INFORMIX中Alter Table语法的形式

加入一个原先不存在的约束，如为表格设置主键或将原来包含空值的一列置为非空(ORACLE中这只能在列的MODIFY子句中出现)，通常结果是什么都不做(NO ACTION)即

该语句不被执行。DROP PRIMARY KEY子句在ORACLE和DB2 UDB中有，在INFORMIX中没有。但是，如果事先为这个约束命了名，在任何一个产品中都可以去除它。

如果要了解所用产品的Alter Table语句的确切形式，请读者查阅相应的SQL参考手册。

注意，如果用Alter Table语句加入一个新列，必须先置好所有的空值或缺省值，因此这一列不能被定义为NOT NULL，除非已经设置了缺省值。向表中加入新的列将影响行的物理存储，因为行大小的膨胀可能使它们不能再存储在磁盘的当前位置。大多数的产品都允许DBA修改表结构，在新列中置空值，而不须将表的所有行移到新的磁盘位置，对比较大的表来说，这项工作将占用巨大的计算机资源。不同的数据库产品在处理物理存储的问题上有不同的方法。

5. 非过程性和过程性完整性约束：触发器

SQL完整性子句的设计者显然应尽量预见在逻辑数据库设计中可能用到的所有重要规则，这样，这些规则就能作为非过程性约束实现。该约束是不能被打破的，因为在执行SQL更新语句的过程中它们始终是有效的，这样能防止错误的更新破坏数据的完整性。在本节的其余部分，我们将对完整性约束的各种方面进行讨论，并与一种叫做触发器的生成动态规则的过程性方法做比较。如我们在3.11节中对非过程性SQL命令的讨论，非过程性约束比过程性约束的功能少很多，但这有一个表达复杂性的问题，如果我们要加上这些功能，就需要付出代价。我们先简单介绍一下触发器是怎样实现的，这样才能知道如何根据其利弊进行选择。

(1) 过程性约束和触发器

相对有效的触发过程在SYBASE商用数据库产品中早就存在，现在ORACLE、DB2 UDB、INFORMIX和MICROSOFT SQL SERVER中也都有。SQL-92中没有提供有关触发器的标准，Melton和Simon写到“标准委员会的猜想错误：他们没有意识到触发器的需求和实现的发展会如此迅速……但大多数销售商直接越过SQL-92标准，将SQL-99中触发器的规范作为实现的基础”。我们将学习Full SQL-99中的Create Trigger语法，如图7-10所示。

```

CREATE TRIGGER trigger_name {BEFORE | AFTER}
  ({INSERT | DELETE | UPDATE [OF columnname [, columnname...]]})
  ON tablename [REFERENCING corr_name_def [, corr_name_def...]]
  [{FOR EACH ROW | FOR EACH STATEMENT}]
  [{WHEN (search_condition)}
    {statement
      -- action (single statement)
    }
    | BEGIN ATOMIC statement; { statement;... } END -- action (multiple statements)
  ]
  定义相关名的corr_name_def如下：
  {OLD [ROW] [AS] old_row_corr_name
  | NEW [ROW] [AS] new_row_corr_name
  | OLD TABLE [AS] old_table_corr_name
  | NEW TABLE [AS] new_table_corr_name}

```

图7-10 高级SQL的Create Trigger语法

Create Trigger语句创建一个名为trigger_name的触发器数据库对象，这个名字用在错误信息和以后从数据库中删除该触发器的请求中：

```
DROP TRIGGER trigger_name;
```

触发器在指定表上发生所列事件（INSERT、DELETE或UPDATE，可任意限制在一组合

了名的列上)之前(BEFORE)或之后(AFTER)被触发(执行)。如果触发器被触发,将执行可选的搜索条件(如果说有的话),以决定当前受影响的一行或多行是否引起进一步的动作。如果是,被触发的动作,即BEGIN ATOMIC和END中包含的一个或多个语句将被执行。这些语句一般是以分号结尾的可执行的SQL语句(打开连接或开始一个会话或事务的语句是不允许的),包括SQL-99中用来调用SQL子程序的CALL语句,如内部存储着的过程(在SQL-99过程性SQL语言SQL/PSM,或产品自定义的过程性语言中)或C、Java语言的一个函数。动作的粒度是FOR EACH ROW或FOR EACH STATEMENT(缺省的选项),这决定了动作执行的频度——对每个受影响的行执行一次或是在语句结束时执行一次。(显然,我们可能执行多行的Update、Delete或Insert语句,在此将产生很大的不同。)

在被触发的UPDATE事件中,动作程序中的语句序列可能在更新一行之前或之后,使用该行的值。为达到这个目的,我们可以用REFERENCING子句为被更新的一行或多行、甚至整个表定义“相关名”。表的相关名(old_table_corr_name或new_table_corr_name)在两种粒度下都可以使用,而行的相关名(old_row_corr_name或new_row_corr_name)只能用在FOR EACH ROW的粒度中。行的相关名可用于WHEN子句或触发器动作语句,限定其表达式中的列名,还可以用于确定列的值到底是来自旧行(在更新之前)还是新行(在更新之后)。整个表的相关名像表名一样使用,类似地,用于确定所指的是哪个版本。

因为触发器能引起一个动作的执行,所以它可以用来实现过程性约束。DBA可以写一些语句作为实现约束的过程性动作(比较普遍的是用一些相对简单的过程)。大部分主要的数据产品,如SYBASE、ORACLE和INFORMIX都提供包含常驻内存的变量、循环控制和if-then-else逻辑的过程性SQL语言的扩展。在第4章中,我们简单介绍过ORACLE(PL/SQL)和INFORMIX(SPL)的过程性语言。DB2 UDB的过程性扩展少一些,而且只提供给触发器。比较复杂的过程性动作都是用如C等语言实现、通过触发器调用的。

因为在定义触发器动作时,各产品用自己特定的过程性语言,所以想给出一个Create Trigger的“基本SQL”版本是不可能的。但不同产品的Create Trigger形式中,非动作部分很相似,都要能够直接地将简单应用触发器从一个产品移植到另一个产品。(有些应用程序要用到产品特定的时序规则,要由一个更新事件同时触发多个触发器,这样的程序较难移植。)

```

CREATE TRIGGER trigger_name [NO CASCADE BEFORE | AFTER]
  ([INSERT | DELETE | UPDATE [OF columnname [, columnname...]]])
  ON tablename [REFERENCING corr_name_def [, corr_name_def...]]
  [FOR EACH ROW | FOR EACH STATEMENT] MODE DB2SQL
    [WHEN (search_condition)]
    { statement
      | BEGIN ATOMIC statement; {statement...} END};
  定义表或行的相关名字的corr_name_def如下:
    {OLD [AS] old_row_corr_name
     | NEW [AS] new_row_corr_name
     | OLD_TABLE [AS] old_table_corr_name
     | NEW_TABLE [AS] new_table_corr_name}

```

图7-11 DB2 UDB的Create Trigger语法

图7-11是DB2 UDB的Create Trigger语法。它与SQL-99的语法在一些次要的方面略有不同。

首先，DB2 UDB不需要在BEFORE触发器中加上关键字NO CASCADE来保证它不会触发其他触发器。其次，附加的关键字MODE DB2SQL必须出现，这样以后DB2 UDB就能够提供一个新的MODE关键字来定义新的功能，而不破坏现有代码。最后，可选择的REFERENCES子句中的关键字与SQL-99有些不同，必须特别指定FOR EACH子句中的一个（不能使用缺省值）。DB2 UDB仅对AFTER触发器支持FOR EACH STATEMENT。一般而言，FOR EACH ROW触发器的实现比FOR EACH STATEMENT更普遍。

ORACLE的Create Trigger语法如图7-12所示。它与SQL-99的一个很大的不同是，它的触发器语句完全是用PL/SQL描述的（可能包括CALL语句，调用已经存储了的过程或一些语言（如C）的函数。）

```

CREATE [OR REPLACE] TRIGGER trigger_name {BEFORE | AFTER | INSTEAD OF}
  {INSERT | DELETE | UPDATE [OF columnname [, columnname...]]}
  ON tablename [REFERENCING corr_name_def [, corr_name_def...]]
  {FOR EACH ROW | FOR EACH STATEMENT}
  [WHEN (search_condition)]
  BEGIN statement {statement;...} END;
  定义行的相关名字的corr_name_def如下：
  {OLD old_row_corr_name
  |NEW new_row_corr_name}

```

图7-12 ORACLE的Create Trigger语法

重要注意事项 在ORACLE中，可选的WHEN搜索条件不能包含任何子查询或调用用户定义的函数；但是，这样的条件代码可以在触发器体内实现。

INSTEAD OF子句将触发触发器，但不执行INSERT、UPDATE或DELETE等动作。而且，因为不存在old_table_corr_name，所以在一个AFTER触发器中，我们无法访问整个旧表，只能用正常的名字访问新表（但并不总是这样，参见《ORACLE8 Server Application Developer's Guide》（《ORACLE8服务应用程序开发指南》），本章最后的“推荐读物”[14]）。

下面我们将使用ORACLE触发器语句的语法，并简要说明等价的DB2 UDB形式。我们先用触发器实现几个原来用Create Table语句实现的完整性约束来研究触发器的特性。

例7.1.5 使用触发器来检查，新的customers行中值不超过15.0的discnt。

```

create trigger discnt_max after insert on customers
referencing new as x
for each row when (x.discnt > 15.0)
begin
  raise_application_error(-20003, 'invalid discount attempted on insert');
end;

```

每进行一次表customers的插入，这个触发器就会测试WHEN子句中的搜索条件 $discnt > 15.0$ ，看它是否对每一行都满足。如果发现有不满足的情况，触发器将执行RAISE_APPLICATION_ERROR语句，撤回触发器的所有作用，执行该插入命令的应用程序中将出现SQLERROR情况，新插入的行都被删除，SQLCODE返回-20003，SQLSTATE置为72000，表示SQL执行出错。如果应用程序是用PL/SQL写的，可以为特定的应用程序错误（这里是-20003）命名，像正常命名的例外一样处理。

在执行Update语句后，触发器也可以保证表customers中的discnt值不大于15.0；实际上，在ORACLE中的一条Create Trigger语句可以处理两个事件，只要把AFTER子句的第一行替换为

```
...after insert or update...
```

例7.1.5中，触发器动作中只有一个语句，ORACLE的语句形式：

```
RAISE_APPLICATION_ERROR(error_number, error_message_string);
```

其中的error_number必须在-20000到-20999之间，-20000表示非特定的应用程序错误，-20001到-20009表示程序员定义的特定的应用程序错误。执行这条语句会导致触发器的应用程序中出现SQLERROR情况。SQLCODE变量允许用户自定义错误码，而SQLSTATE只有一个通用的值72000表示SQL执行出错。注意，SQL-99不提供类似RAISE_APPLICATION_ERROR()的产生例外语句，可以用厂商特定的SQLCODE变量定义用户的例外号。另一方面，SQLSTATE的部分空间是为实现时定义的代码保留的，所以厂商采用这种机制也是合适的，如DB2 UDB中所用的。ORACLE的库函数sqlglm()得到（用附录B中的print_dberror()函数打印）包含用户定义的错误信息error_message_string的错误信息串，作为RAISE_APPLICATION_ERROR()的第二个参数，还有诸如被触发的触发器名等其他信息。（如果应用程序是用PL/SQL写的，可以为特定的应用程序错误环境命名，像正常命名的例外一样处理。）

DB2 UDB中，相应的触发器语句如下：

```
SIGNAL SQLSTATE_string (SQLCA_error_message_string);
```

如果用DB2 UDB来实现例7.1.5，代码如下：

```
create trigger discont_max after insert on customers
referencing new as x
for each row when (x.discont > 15.0)
SIGNAL '70003' ('invalid discount attempted on insert'); -- DB2 UDB STATEMENT
```

这条语句将导致语句在执行过程中异常终止，错误码是一个5个字符的SQLSTATE_string，如例7.1.5中的70003。括号中的SQLCA_error_message_string，'invalid discount attempted on insert'，放在SQLCA的SQLERRMC字段中，可以用附录B的print_dberror()函数打印出来（从嵌入的C程序中）。想了解关于SQLSTATE码的更多信息，请参阅5.2节中的“显式的错误检测”小节。根据厂商的习惯，所有以‘7’开头的SQLSTATE码被留作特殊用途（这与SQL-99标准是一致的，SQL-99标准中所有以数字5到9或字母I到Z开头的代码是在实现时定义的）。

为了得到一个能实现简单的FOREIGN KEY和REFERENCES子句的触发器，触发器还需要涉及一张表。

例7.1.6 下面是如何在X/Open和Full SQL-99标准中实现叫做ON DELETE SET NULL的策略，但ORACLE中只有ON DELETE选项，没有ON DELETE SET NULL选项。将这项策略用于表customers和orders，删除表customers的一行时，我们把orders中对cid的引用都置为空值，否则它们将成为非法值。

```
create trigger foreigncid after delete on customers
referencing old as ocust
for each row
```

```

begin
    update orders set cid = null where cid = :ocust.cid;
end;

```

注意，在DB2 UDB中我们不需要这个触发器，因为它已经有ON DELETE SET NULL选项了。上述的触发器与BEFORE触发器一样工作，但根据ORACLE的说法，总体来说AFTER触发器运行起来更快。 ■

要过程性地实现CASCADE策略是相当困难的（但在ORACLE和DB2 UDB中用ON DELETE子句可以实现），出于这个原因和我们将简单讨论的其他原因，SQL-99委员会决定，尽管加入了实现过程性能力，仍推荐使用非过程性动作（约束）。

(2) 过程性与非过程性约束的优缺点

我们考虑的所有约束都是为了防止错误的SQL更新语句意外地破坏数据完整性。这表明很多错误是由于用户疏忽大意地进行SQL更新而产生，完整性约束概念的出现，就是在一段时间内，这样粗心的用户很多。现今多数计算机专业人士都认为不应允许粗心的用户对重要的表进行交互式的更新。粗心的用户很容易错误地把一些行中某几列的值改成看似正确（满足所有能想到的约束）实际上完全错误的值。只允许用户通过一个程序界面来更新表会比较好，因为程序经过仔细的设计，不太可能出现错误。如果是这样，为什么我们不去掉即席SQL中的更新语句，用程序逻辑来实现约束呢？这种逻辑灵活性很大，而且理想情况下可以实现DBA需要的所有完整性约束。

它的危险之处在于它过于灵活。一个对规则不是很清楚的新程序员，甚至是一个有经验的程序员，稍有疏忽就可能破坏程序要维护的规则。为避免这种情况，有必要建立一个程序层，所有的数据库更新操作在这一层上进行，这样就可以减少甚至避免上述错误，也就是说，更新表时，应用程序员不能直接用嵌入式SQL语句，必须调用层函数update_tblname()。但是要提供一个既能很好地保证完整性又灵活高效的程序层并不是一个简单的设计任务。最初，触发器特征就是用过程性形式完成同样的功能，对程序员来说，这些都是透明的。

显然，Create Table语句中的非过程性约束的种类是有限的（CHECK约束、UNIQUE值约束、引用外键FOREIGN KEY-REFERENCES约束等），这限制了它的能力，但也使它比可任意改变的、复杂的过程性约束更容易理解。我们使用的有限的非过程性约束集可以像系统目录表中的数据值那样排列，系统目录表是系统支持的表集，列出数据库中定义了的所有对象（将在本章的后面介绍），而且约束可以根据有DBA权限的用户的要求进行缩减或更新。这是一个值得考虑的重要因素。系统所有的规则自动集中在一个地方，用易于理解的类似于存储数据的方法存储，便于DBA将它们作为一个整体进行检查。

非过程性完整性约束优于过程性触发器的另一个理由是，非过程性约束直接为系统所知，因此检查起来比过程性的逻辑要有效得多。多数情况下如此，但并不总是如此。为了实现同等的约束，系统经常做的工作和使用过程性逻辑实现可兼容的约束差不多，因此使用一个容易控制的系统，并不意味着能大幅度地提高性能。

驳斥非过程性系统约束的一个理由是，过程性约束的作用更大。在下面的两个领域中，非过程性约束目前还显得有所欠缺。

给出约束不满足时的相应动作 非过程性约束的一个主要缺点是，它们是事先实现的，因此无法知道当约束不满足时，程序应该做些什么。在一个管理员工的数据库中，我们可能希望创建一个完整性约束，保证所有员工的年龄在18到70岁之间。如果有一个自动例程将一

个员工的年龄改为71，除了指出更新失败外，我们肯定还需要做一些别的动作——不希望该员工继续在公司工作，将其年龄值保持为70，尽管实际年龄已经增加了。尽管约束能自动地拒绝错误的更新，针对该情况应执行什么动作却必须由应用程序员提供。这种拒绝是以带有某个错误码的SQL例外的形式出现的，在触发器中，应用程序对每个例外都要检测一次错误码，而不是对所有例外只检测一次。

再举一个例子，我们可能需要一条规则，只接受来自订货员的合法的名字和地址——例如，须保证邮编与州对应。（邮编不能跨州——这条约束可以用参照完整性实现。）如果检查不通过，我们不希望只是简单地拒收数据。可能需要调用一个程序，要求订货员重新输入信息，而且如果错误重复出现，应先将它暂时储存下来，可以存在一张单独的表中。毕竟，不完全正确的数据对一个企业也是有价值的，通过分析可以纠正错误。在这两个例子中，为非过程性约束指定的测试是这项工作中最简单的部分，难的是指定当测试不通过时应采取的动作，显然，这就需要一个过程性方法了。

保证事务的一致性 另一个重要的约束是事务的一致性，用来保证更新事务的正确性。例如，在商业事务中，把钱从一个账户转到另一个账户，必须要有一致性规则保证钱既不会多出来，也不会少掉。只要程序逻辑本身保证了这条规则，这可以由并发的程序保证。但程序逻辑中肯定会有问题，而约束可以在运行时检测到非常细小的、破坏一致性的错误，这对那些需要对更新事务操作负责的应用程序管理员来说可谓是一大福音。但是，大多数商用数据库都不提供在事务结束前用于处理两个或多个更新操作之间交互作用的约束。

使用触发器，我们可以检查多个更新的作用，或使一个更新操作在另一个执行之后自动发生。例如，假定对表orders中的每一个订单，需要统计与该订单有关的表line_items中行的个数。我们可以用一个触发器，当line_item插入时触发，将表orders相应行上的记数值加1。这样计数工作可由触发器完成，而不需要靠应用程序逻辑来实现。

例7.1.7 下面使用触发器为每个订单实现line_item计数。假定订单已存在，且n_items值初始化为0。

```
create trigger incordernitems after insert on lineitems
referencing old row as oldli
for each row
begin          -- for ORACLE, leave out for DB2 UDB
    update orders set n_items = n_items + 1 where ordno = :oldli.ordno;
end;           -- for ORACLE

create trigger decordernitems after delete on lineitems
referencing old row as oldli
for each row
begin
    update orders set n_items = n_items - 1 where ordno = :oldli.ordno;
end;
```

■

再举一个例子，假定两个账户是连在一起的，为保证余额为正，一个可以向另一个借钱。当借这个动作会引起大范围的程序动作时，我们可以用一个触发器来实现它。如果第二个账户的资金也不足，则产生一个例外。

因为在事务逻辑中不具有指定替代动作及测试一致性规则的约束能力，我们很自然地会

提出以下问题：如果让程序员保证程序中复杂的规则，并在程序逻辑中单独处理各个替代动作，为什么还要对每种情况单独给出一个约束呢？原因可能是，我们想尽量限制能预料到的错误，希望以此提高数据库应用程序的可靠性。

过程性约束的一个主要优点是便于实现一些新想法。尽管一个商用产品最后可能在系统中加入某个有特定作用的约束，但用户通常要等很长时间才能等到它在下一个版本中实现。在长达几年的时间里，很多数据库销售商都不提供参照完整性，直到它被证明有很多好处。约束的规格说明还处于发展阶段，本章提出的很多问题可能要等到更统一的系统出现后才能解答。也许在将来的数据库系统中，提供给DBA的约束中。一些是非常普通的非过程性约束，一些是用来处理复杂情况的过程性约束。

(3) 列出已定义的触发器

列出所有已定义的触发器的标准方法是用Select语句从系统目录表中检索触发器名，系统目录表是系统支持的表集，列出数据库中定义了的所有对象（将在本章后面介绍）。不同数据库产品的系统目录表的名字和结构也不同。在ORACLE中，要找到触发器和应用它们的表，应使用下列语句：

```
select trigger_name, table_name from user_triggers;
```

如果要得到某个特定触发器的信息，需要用大写字母的名字引用它：

```
select when_clause, trigger_body from user_triggers
  where trigger_name = 'DISCNT_MAX';
```

如果查询结果的显示被截断了，用户可以用SQL*Plus命令重新设置这类字段的显示的长度，缺省值为80，例如：

```
set long 1000
```

在DB2 UDB中，要列出用户eoneil的触发器，我们应该写以下语句：

```
select trigname, tabname, text from syscat.triggers where definer = 'EONEIL';
```

7.2 建立视图

下文中，我们将把用Create Table语句定义的表称为基本表。基本表中的行实际上是存储在磁盘上的，通常是以物理记录的形式，不同数据类型的字段连续存放，如Create Table语句中指出的那样。现在关系模型的一个重要属性是，由SQL的Select语句得到的数据仍然以表形式出现，由此引出了视图表的定义。视图表（有时简称为视图）是一个由子查询产生的表，但它可以有自己的名字，在很多方面都类似于基本表。视图表是一个从数据库的基本表中选取出来的数据组成的逻辑窗，基本表的名字用Select语句的FROM子句指定，在经过谨慎的限制后，它甚至可以是可以更新的。

例7.2.1 创建一个叫agentorders的视图表，它扩展了表orders的行，以包含订了货的代理商的所有信息。用SQL的Create View语句实现：

```
create view agentorders (ordno, month, cid, aid, pid, qty, charge, aname, acity, percent)
  as select o.ordno, o.month, o.cid, o.aid, o.pid, o.qty, o.dollars,
         a.aname, a.city, a.percent
    from orders o, agents a where o.aid = a.aid;
```

注意到我们将检索出的`o.dollars`列重命名为`charge`。除`o.aid`外，Select的目标列表中的所有列的名字对于它们的包含表是唯一的，因此不要求在目标列表中再进行限定：`o.ordno`, `a.city`等。

执行Create View语句时，没有数据被检索或存储。但是，视图的定义应该作为数据库中一个独立的对象存放在系统目录中，以备其他的查询或Update语句的FROM子句中以这一视图的名字对其进行检索。记得图3-11的高级SQL语法允许Select语句的FROM子句中包含子查询；这里看来，视图正是给子查询一个名字。视图表的概念早已被标准化了，因此大部分数据库产品都支持视图，而图3-11的高级SQL语法就不是普遍支持的。对数据库系统来说，修改视图的查询或Update语句是很容易的——视图的定义与这些语句的内容有紧密联系——因此修改的查询或更新实际上完成对基本表上的访问。这种方法称为查询修改。

例7.2.2 用例7.2.1中的agentorders视图计算Toledo的代理商的订单金额总数。下面是用户应该给出的查询：

```
select sum(charge) from agentorders where a.city = 'Toledo';
```

数据库（至少是概念上地）对这一查询进行修改来考虑例7.2.1中视图agentorders的定义，得到的查询如下（被替代的视图中定义的元素用下划线标出）：

```
select sum(o.dollars) from orders o, agents a
  where o.aid = a.aid and a.city = 'Toledo';
```

认识到视图只是基本表数据的一个窗，这是非常重要的。创建视图时，我们得到的并不是数据的快照，面只是存储一些定义，每执行一个新的查询都必须重新解释这些定义；所以作用于视图的查询能及时地反映基本表中数据的变化。

本文中用到的基本SQL标准的Create View语句的完整叙述如图7-13所示。（它与X/Open和Core SQL-99标准是相同的。）

<pre>CREATE VIEW viewname [(columnname [, columnname...])] AS subquery [WITH CHECK OPTION];</pre>

图7-13 基本SQL的Create View语法

记得图3-13中定义的子查询只比完整的Select语句少一个ORDER BY子句，对视图的定义基本没有限制。Create View命令可被用作嵌入式SQL语句，但视图的子查询中不能包含宿主变量或动态参数。创建视图的用户即是视图的所有者，有更新视图的权限，前提是视图是可更新的（解释见下），而且该用户对定义视图的基本表也有更新权限（只能有一个这样的基本表）。

可选的WITH CHECK OPTION子句规定，通过视图完成的Insert和Update操作改变基本表的内容，如果导致对视图的子查询是不可见的行，它们将不被允许；说明见例7.2.4。如果图7-13中可选的列名列表没有给出，新创建的视图将继承子查询中目标列表的名字。但如果视图的列在目标列表中表示表达式则必须给出名字。（ORACLE和DB2 UDB允许子查询的表达式用列的别名，但这并不普遍，目前INFORMIX就不支持这种用法。）同时要注意，在通过继承得到的视图的目标列表中不需要限定词保证列名唯一，但如果目标列表中不用限定词，如果原来的列名重复，则要专门为视图中的列命名。这一点在例7.2.3中阐述。

例7.2.3 创建一张称为cacities的视图，列出表customers和表agents中所有配对的城市，满足这个城市的代理商为另一个城市的顾客下了一份订单。下面的语句是错误的，因为得到的视图中两个city列的名字相同。

```
create view cacities as
  select c.city, a.city
  from customers c, orders o, agents a
  where c.cid = o.cid and o.aid = a.aid;
                                         ** ILLEGAL VIEW DEFINITION
```

这样的语句运行时会产生错误。我们需要为视图给出不同的city名：

```
create view cacities (ccity, acity) as
  select c.city, a.city
  from customers c, orders o, agents a
  where c.cid = o.cid and o.aid = a.aid;
```

■

假定有一个数据库系统的Create Table语句中不支持CHECK选项，我们可以用视图来克服这种限制。

例7.2.4 下面我们看看如何用视图中的WITH CHECK OPTION子句实现Create Table语句中CHECK子句的功能，所有对表的更新都要通过视图来实现。假定表customers已经用例7.1.2的方法建立，但没有CHECK子句保证discnt <= 15.0。我们创建如下的视图：

```
create view custs as select * from customers
  where discnt <= 15.0 with check option;
```

现在，所有使表customers的discnt > 15.0的更新都将失败，因为被改变的行对视图是不可见的。考虑如下更新：

```
update custs set discnt = discnt + 4.0;
```

根据图2-2的数据，对顾客c002的更新将失败，因为discnt的值将变成16.0，而且对视图是不可见的。如果对每一行的更新CHECK选项都不成立，那么最后会报错，而且表不会被改变。

我们将简要地说明，不是所有的视图都能接受更新并将更新实施到基本表上。可以被更新的视图称为可更新视图。注意，要在-一个视图的基础上建立另一个视图，即建立嵌套视图定义，是完全可能的。

例7.2.5 创建一个叫acorders的视图，给出所有订单信息以及订单中涉及到的代理商和顾客的名字：

```
create view acorders (ordno, month, cid, aid, pid, qty,
  dollars, fname, lname)
as select ao.ordno, ao.month, ao.cid, ao.aid, ao.pid, ao.qty,
  ao.charge, ao.fname, c.lname
from agentorders ao, customers c where ao.cid = c.cid;
```

可以看到，例7.2.1中的视图agentorders用在视图acorders定义的FROM列表中。 ■

(4) 列出已定义的视图

列出所有数据库中定义了的视图的标准方法是用Select语句从系统目录表中检索视图名，(细节参见7.4节)。例如，根据X/Open标准，要列出所有视图应该用下面的语句：

```
select tablename from info_schema.tables where table_type = 'VIEW';
```

不同数据库产品的系统目录表的名字和结构也不同。在ORACLE中，我们应该写

```
select view_name from user_views;
```

对每个这样的视图，我们可以用Describe语句获得任何表或视图（以及数据库中的其他对象）的更进一步的信息，例如：

```
describe agentorders;
```

在DB2 UDB中，要列出用户eoneil的视图，应该写

```
select viewname from syscat.views where definer = 'EONEIL';
```

DB2 UDB描述命令只能用于表，不能用于视图。但是，有关列的信息可以从syscat.columns中得到。参见本章7.4节有关的系统目录的内容。

(5) 删除表和视图

用于从系统目录表中删除视图定义的标准SQL语句与删除表的Drop语句基本相同，删除表语句的有些特征我们还没有谈到。下面先介绍X/Open和Full SQL-99中完整的Drop Table和Drop View语句（Core SQL-99中没有CASCADE选项）。

```
DROP {TABLE tablename | VIEW viewname} {CASCADE|RESTRICT};
```

这种形式要求指定CASCADE或RESTRICT，而大部分产品都给出了缺省值。在Drop Table语句中，{CASCADE | RESTRICT}指出，如果FOREIGN KEY约束或视图表引用了被删除的表，应该如何处理：CASCADE表示所有的外键约束和视图都同时被删除，RESTRICT表示在这种情况下，Drop Table语句失败。类似地，Drop View语句的情况下，CASCADE和RESTRICT指出，如果有任何一个视图是在这个即将被删除的视图的基础上建立的，该如何处理。当基本表被删除时，它的所有行都被自动删除，但视图实际上是虚的，没有实在的行存在，因此删除视图时，其实什么也没有被删掉。

只有表或视图的所有者（通常是创建者）有权删除它们。许多产品也用Drop语句从系统目录中删除其他类型的数据库对象，如表空间、索引等。我们将在第8章中介绍这些数据库对象。

以ORACLE为例解释图7-14的含义，ORACLE中的写法应如下：

```
DROP TABLE tablename [CASCADE CONSTRAINTS];
DROP VIEW viewname;
```

ORACLE删除一个表或视图时，基于它的视图和触发器将变成无效的。如果再以相同的列名重新建立一个表，这些视图和触发器可以再次变成有效的。在Drop-Table语句中，CASCADE CONSTRAINTS子句要求所有引用该表的约束同时被删除。如果没有这个子句，若任何约束引用该表则该语句将失败。

	ORACLE	DB2 UDB	INFORMIX	X/Open, SQL-99
DROP TABLE tablename...	[CASCADE CONSTRAINTS]		[CASCADE [RESTRICT]]	[CASCADE [RESTRICT]]
DROP VIEW viewname...			[CASCADE [RESTRICT]]	[CASCADE [RESTRICT]]

图7-14 各种产品和标准中的Drop Table和Drop View语句

如图7-14所示，DB2 UDB中的简单形式如下：

```
DROP {TABLE tablename | VIEW viewname};
```

DB2 UDB删除一个表或视图时，基于它的视图和触发器都将被变成无效的。它们的定义仍然保留在系统目录中，但只有当表被重建后，它们才是有效的。引用该表的约束同时被删除。

INFORMIX删除一个表或视图时，基于它的触发器变成无效的。如果用CASCADE子句，将删除引用该表的视图和引用约束，而用RESTRICT子句时，不允许删除被视图或引用约束引用的表。

1. 可更新视图和只读视图

视图并不是各个方面都像基本表一样灵活，有些视图不能被更新。不允许更新的最充分理由是，有时不知道如何将对视图的更新转化成对其基本表的更新，以反映视图的创建者和提出更新要求的用户的意图，但限制视图更新的一些规则解决了这个问题，能够将对视图的更新转化成对其基本表的更新。特别是，在很多数据库产品中，由连接操作得到的视图都不能被更新，虽然对大多数列的更新都能简单地转化成相应的对基本表的更新。本章最后的习题中有关于这方面的内容。

在基本SQL中，视图是可更新的或只读的。可更新视图允许Insert、Update和Delete操作，而只读视图则不允许（注意，所有的基本表都被认为是可更新的）。图7-15给出了在X/Open中更新视图时必须遵循的标准规则。

若视图的子查询语句满足以下条件，则称它为可更新的。

- 1) 子查询的FROM子句只包括一个表，而且如果这个表是视图表，则一定是可更新的视图。
- 2) 不出现GROUP BY和HAVING子句。
- 3) 没有使用DISTINCT关键字。
- 4) WHERE子句中，没有子查询直接或间接地通过视图引用FROM子句的表。
- 5) 所有由子查询得到的列名都是简单的(即没有类似于avg(qty)或qty+100的算术表达式)，在不同的结果列中也没有列名重复出现。

图7-15 X/Open对可更新视图的子查询语句的限制

我们将用例子说明有这些限制的原因。

例7.2.6 下面是一个称为colocated的视图的定义：

```
create view colocated as select cid, cname, aid, fname, a.city as acity
    from customers c, agents a where c.city = a.city;
```

这个视图列出了在同一个城市里的顾客和代理商，这便于将他们邀请出来，介绍他们互相认识。假定在图2-2的CAP数据库中，视图有两行分别是

c002 Basics a06 Smith Dallas

和

c003 Allied a06 Smith Dallas

但视图的定义违反了上面的第1条规则，因为FROM子句包括不止一个表，因此这个视图被限定为只读的，而不是可更新的。让我们试着通过观察一些应用到视图的Update语句来理解这一限制。先提一个问题：如果我们要将上面第二行（即cid = ‘c003’，aid = ‘a06’

的那一行) 中的 `aname` 从 `Smith` 改为 `Franklin`, 相应地要对基本表做哪些修改? 显然, 如果 `a06` 代理商将其名字改为 `Franklin`, 上面的第一行也要改, 所以对视图中一行的修改会对其他行带来不可预料的副作用。我们不希望在基本表中出现这种情况。

如果删除第二行, 情况又略有不同。这时应该对基本表进行哪些修改呢? 是删掉表 `customers` 中 `c003` 的一整行, 还是删掉表 `agents` 中 `a06` 的一整行, 还是改变其中一个表的 `city` 值 (改成什么呢?), 或是做一些别的什么修改? 类似地, 如果我们想插入一个新行:

```
c003      Allied      a12      Watanabe      Dallas
```

似乎是在表 `agents` 中新建了一行, `a12 Watanabe Dallas, percent` 字段为空。我们同样必须在视图中插入一行, 将顾客 `c002` 与代理商 `a12` 联系起来, 这同样可能有不可预料的副作用。

为避免这类复杂的情况, 标准规定 `FROM` 子句中只能有一个表。少数采取了复杂措施的数据库允许在连接视图上进行有限的修改, 我们将在练习中进行这方面的考虑。 ■

下面一个例子说明图 7-15 中的第 2 条规则。

例 7.2.7 我们定义一个称为 `agentsales` 的视图, 包含所有下过订单的代理商的 `aid` 值以及他们的销售总额:

```
create view agentsales (aid, totsales) as select aid, sum(dollars)
  from orders group by aid;
```

根据上面的第 2 条规则, 视图有 `GROUP BY` 子句, 因此是不可更新的。给出这条约束的理由同样是因为无法将对视图上的更新直接转换成对基本表的更新。例如, 假设我们想通过表 `agentsales` 将代理商 `a01` 的销售总额增加 \$1000.00。

```
update agentsales
  set totsales = totsales + 1000.00;          /* ILLEGAL SYNTAX */
```

这样更新的问题是, 我们不知道应加入什么信息才能让更新在基本表 `orders` 中体现出来。对于代理商 `a01` 加入到表 `orders` 中的订单的 `cid` 和 `pid` 值是什么呢? 到底是增加一个订单, 还是几个订单呢? 或者表 `orders` 的一行或多行的 `dollars` 都变成更大的值? 这些我们都不得而知。 ■

再重复一次, 如果我们不确切地知道对视图的更新应该怎样反映到基本表上, 就不能允许其发生。因此, 第 3 条规则要求可更新的视图中没有关键词 `distinct` 也是很容易理解的。如果视图中的一行在基本表中存在同样的好几行, 当要求更新指定行的一个属性时, 应该怎么办呢? 是更新基本表中的一行, 还是所有这样的行?

有些由连接得到的视图可以被安全地更新。ORACLE 允许对连接视图进行更新, 如果连接是 N-1 (多对一) 的, 而且 N 方有主键 (这样得到的是无损连接)。例如, 例 7.2.1 中的 `orders` 视图就具备这些特性, 所以在 ORACLE 中视图 `agentorders` 是可更新的。但是我们只能更新与表 `orders` 一对一的列, 不能更新列 `aname`, `acity` 和 `percent`, 也不能修改决定连接的列 `aid`。为列出这个视图中可以更新的行, 我们可以对 ORACLE 的数据字典视图 `user_updatable_columns` 执行下面的 Select 语句:

```
select column_name, updatable from user_updatable_columns
  where table_name = 'AGENTORDERS';
```

2. 视图的值

第1章中提到了集中控制的数据会产生的一些问题。第一是逻辑复杂性。例如，一个验证大学学生保险费的应用程序为获得所需的列信息，要访问很多表，而且这些表中很多表的名字、用途与健康保险是完全没有关系的。结果，要培训一个新的程序员在这么多表中导航是非常困难的。集中控制的第二个目标是分段实现，每当引入新的数据库时，集中控制的好处就增加一点。当出现新表，需要用新的列名访问旧的列名时，我们不希望为了消除数据冗余而重写应用程序。理想的情况是，我们希望有一些办法确保表的增加不给培训带来问题，并且当为了消除数据冗余而重新组织老的表，或用新的表取代它们时，不需要修改原来应用程序的代码。视图一般可用来解决这两个问题以及其他问题。

1) 视图提供了一种方法，使复杂的、经常用到的查询写起来更容易。这一点我们可以从例7.2.2中看出来，我们可以写查询

```
select sum(charge) from agentorders where city = 'Toledo';
```

而不用写

```
select sum(o.dollars) from orders o, agents a
  where o.aid = a.aid and a.city = 'Toledo';
```

以前，几乎每个查询都要用到表的连接和复杂的搜索条件，现在，视图可以隐藏这种复杂性，通过对系统不可见的子查询的修改来实现复杂的查询。（在实际应用中，视图通常隐藏了很多复杂性。）

2) 视图允许过时的表和引用过时表的程序不被重新组织。试想将表orders重新组成一个新表ords：

```
create table ords (ordno integer not null, month char(3),
  custid char(4) not null, agentid char(3) not null,
  prodid char(3) not null,
  quantity integer default null check(quantity >= 0),
  primary key (ordno),
  foreign key (cid) references customers,
  foreign key (aid) references agents,
  foreign key (pid) references products);
```

我们已经对cid, aid和pid列重新命了名，并从表orders中去除了列dollars；现在我们想用quantity值和被订货物的单价相乘来得到表orders中每一行的dollars总额。如果有很多程序访问老的orders表，我们可以为新组织的ords表和products表建立一个视图，让这些程序使用。

```
create view orders (ordno, month, cid, aid, pid, qty, dollars) as
  select ordno, month, custid, agentid, prodid,
    m.quantity, m.quantity*price
  from products p, ords m where m.prodid = p.pid;
```

通过这种方法，可以使程序对数据的访问独立于其物理结构的改变，这一特性被称为程序数据独立性，我们曾在第1章中提到过（定义1.3.1）。注意到独立性并不完全，因为视图orders的子查询中包含两个表，所以不能通过视图orders来更新数据；这意味着输入新订单的程序逻辑要重写。

3) 视图加入了一种安全措施，让不同的用户以不同的方式看相同的数据。集中控制的数

据支持数据管理的一条重要原则：任何对企业至关重要的信息都只能有一个副本。其危险在于，如果副本多于一个，各版本可能不能保持同步（在一个经纪业务中，同一个股票可能被列出两种不同的价格），这样就可能造成错误的决定。而且，实际上也不能让每个人都有访问所有信息的权利。员工的办公室地点、号码等可以让其他同事知道，而薪水只有经理、人力资源部和财务处才能得到，而平时表现只有经理和人力资源部能得到。

我们可以对同一个employees表建立不同的视图，对访问这些视图的不同用户采用不同的安全措施：见下一节授予安全权限的Grant语句。在授权一类用户访问某个视图和限制他们访问基本表时，自动地为视图中没有命名的字段提供安全性。

7.3 安全性：SQL中的Grant语句

Grant语句是表（基本表或视图）所有者授予一个或一类用户访问表的各种权利的SQL命令。这是访问表的安全性的一种形式，访问列的安全性也可以通过视图实现。其他用户首先必须能够连上包含表的数据库，这项权限是由DBA授予的。

例7.3.1 表customers的所有者只给用“eoneil”账户登录的用户Select权限。

```
grant select on customers to eoneil;
```

基本SQL的Grant语句形式（与X/Open等价）如图7-16所示。

<pre>GRANT {ALL PRIVILEGES privilege [, privilege...]} ON [TABLE] tablename viewname TO {PUBLIC user-name [, user-name...]} [WITH GRANT OPTION]</pre>

图7-16 基本SQL的Grant语句形式

Grant命令可以授予下面列出的所有访问权限，也可以给出用逗号隔开的一个权限序列。

```
SELECT
DELETE
INSERT
UPDATE [columnname [, columnname...]]
REFERENCES [columnname [, columnname...]]
```

形如（SELECT, DELETE, …）的权限给予现在、将来的用户（在PUBLIC中）或其他指定的用户名列表使用带有表名/视图名作为对象的相应SQL语句的权利；REFERENCES权限允许用户在引用该表的表上建立外键约束。如果UPDATE权限中没有指定列名列表，表中现有或将会有的所有列都加上这一权限。可选的WITH GRANT OPTION子句允许有这个权限的用户授予其他用户同样的权限。基本SQL中所有的Grant语句的语法ORACLE、DB2 UDB和INFORMIX也支持。Grant语句也可以用在嵌入式SQL程序中。

表的所有者自动拥有所有权限，而且不能被取消。要授予其他用户与访问视图相关的权利，授权者必须拥有该视图表（而且在生成视图的所有表上有必要的权限）或已经通过WITH GRANT OPTION子句被授予了这些权限。若要在视图上授予插入、删除或更新权限，视图必须是可更新的。

例7.3.2 授予用户eoneil对表orders的选取、更新或插入权限，但不给删除权限，然后

给予eoneil在表products上的所有权限。

```
grant select, update, insert on orders to eoneil;
grant all privileges on products to eoneil;
```

我们可以在创建视图时使用Grant语句，提供字段安全性。对视图使用Grant语句，而不对支持它的视图或基本表使用，同样可以达到预期效果。

例7.3.3 允许用户eoneil在表customers中插入或删除任意行，但只能更新cname和city列，可以选取除discnt外的所有列。因为没有与选取权限相关的字段说明，表所有者先创建一个视图custview：

```
create view custview as select cid, cname, city from customers;
```

现在所有者可以在custview视图上提供需要的授权：

```
grant select, delete, insert, update (cname, city) on custview
to eoneil;
```

因为在基本表customers中，eoneil没有被授予任何权限，所以他不能选取列discnt的值。

用视图还可以对从表中被选取出来的行的子集进行授权。

例7.3.4 允许用户eoneil对表agents中所有percent大于5的行进行所有访问。

```
create view agentview as select * from agents where percent > 5;
grant all privileges on agentview to eoneil;
```

X/Open SQL中撤消对一个表的权限的SQL语句的一般形式如下（基本SQL目前不支持{CASCADE | RESTRICT}子句）。

```
REVOKE {ALL PRIVILEGES | privilege [, privilege...]} |
ON tablename | viewname
FROM {PUBLIC | user-name [, user-name...]} |
{CASCADE | RESTRICT};           -- one of these is required in X/Open
```

Revoke语句可以取消以前授予用户的一些权限。与Grant语句不同，在撤消更新权限时不能指定列名。前面说过，表的所有者自动拥有所有权限，它们是不能被撤消的。Revoke语句可以用在嵌入式SQL中，如果试图撤消一个原来没有授权的权限，将会导致SQLWARNING情况，而不是SQLERROR，后者不为标准所提倡。

X/Open中，CASCADE选项的作用是删除与当前被撤消的权限有依赖关系的视图（要建立视图，必须在基本表上有SELECT权限）或删除依赖于REFERENCES权限的FOREIGN KEY约束。RESTRICT选项与CASCADE不同，如果有这样的依赖关系，将不允许执行Revoke语句。

注意，ORACLE和DB2 UDB中都没有实现这样的必选的CASCADE | RESTRICT子句，INFORMIX中将其作为可选子句，缺省值为CASCADE。ORACLE中语法的相应位置是一个可选的CASCADE CONSTRAINTS子句，这将使系统只删除与被撤消的REFERENCES权限有关的参照完整性约束。DB2 UDB中没有语法控制这种动作，当一个权限被撤消时，三个产品中与其有关的视图都缺省地变成无效的。

数据库产品间的不同

ORACLE、DB2 UDB和INFORMIX产品都支持X/Open SQL标准的Grant和Revoke语句的语法，只是在刚才说到的CASCADE | RESTRICT子句等细节方面略有不同，但三种产品又都有很多附加的权限。例如，ORACLE有一种叫做DBA的权限，有了它用户才能执行本章介绍的大部分SQL命令。另外，ORACLE中的连接(connect)权限允许用户进入数据库，资源(resource)权限允许用户创建表、索引等占用磁盘空间的数据库对象。想了解ORACLE有关这方面的更多细节，请参阅本章最后的“推荐阅读”的推荐读物[13]《ORACLE8 Server SQL Reference Manual》(《ORACLE8服务SQL参考手册》)和推荐读物[12]《ORACLE8 Server Concepts》(《ORACLE8服务概念》)。

DB2 UDB中也有很多附加的权限。它有DBADM权限，使用户可以访问和修改数据库中的所有表、视图等，CONNECT权限有允许用户创建表的CREATETAB权限，还有许多表一级的附加权限。参见推荐读物[1]《A Complete Guide to DB2 Universal Database》(《DB2通用数据库完全指南》)。

7.4 系统目录和模式

所有的关系数据库系统都有系统目录，即由系统维护的，包含数据库中定义的对象的信息的表(或视图)。这些对象包括用Create Table语句定义的表、列、索引、视图、权限、约束等。例如，在X/Open标准中，目录表INFO_SCHEM.TABLES的一行就是一个已定义的表的信息。下面是X/Open的INFO_SCHEM.TABLES系统视图中的几个列。

INFO_SCHEM.TABLES

列名	描述
TABLE_SCHEM	表的模式名(通常是表所有者的用户名)
TABLE_NAME	表名
TABLE_TYPE	'BASE_TABLE' 或 'VIEW'

其他地方的DBA需要引用这样的目录表以获取本地数据库设计的一些情况。DBA用普通的SQL Select语句来获取这些信息。例如，`select TABLE_NAME from TABLES`。执行动态SQL的应用程序为了作出某个决定可能需要访问目录表。例如，要知道某个表中列的数目和名称。数据库系统本身也根据目录表来转化视图上的查询，对运行期的更新语句加上约束。(但是，系统用的肯定是比较有效的方法，而不是直接用Select语句访问目录表的列，所以提到这一点时应该十分小心。目录表可以看做是程序的源代码，为使执行效率更高可以进行某种编译。例如，通过检索所有可以实施的约束，让系统回应已经提交的动态Update语句。)

每个商用数据库都有一套不同结构的目录表名，而且每个数据库用不同的术语来指代它们。DB2 UDB中叫做目录表；INFORMIX中叫做系统目录（其实也是表格）；ORACLE中叫做数据字典（是一些由系统维护的视图）；而X/Open标准中叫做系统视图。目录表是在建立数据库时建立的，目录的基本表只能由系统根据数据定义语句和其他类型的语句进行更新。通常，用户不能直接更新目录表，因为这可能破坏数据完整性（例如，用户删除TABLES中的一行，而该行对应的表对象还没有被删除）。很多产品只允许用户通过只读视图（即只授予PUBLIC SELECT权限）访问目录表以保证不发生更新。但有些产品（特别是DB2 UDB）允许DBA用修改目录表的数据，使查询优化器选择更好的查询方案。

目录表中的信息有时称为元数据，意思是“关于数据的数据”。元数据甚至是自描述的，因为目录表TABLES中有一行是描述表TABLES的（TABLES也可能是视图类型）。下面，我

们将用大写字母表示目录表名。

注意，ORACLE和DB2 UDB 的目录表中的对象标识符都是大写的，而INFORMIX中是小写的。^Θ

当一个用户用Create Table语句定义表orders时，TABLES中将输入表名ORDERS。同样，当用户用大小写混合的名字——orders或OrdErs定义表时，目录中的对象应写作ORDERS。当用户在Select语句的WHERE子句中使用对象名时，这是非常重要的。

1. 模式

Core SQL-99中模式是表、索引及其他与数据库有关的对象的集合，这些对象通常是为单个用户设计的。数据库目录中表的完整名是模式名.表格名（模式名通常是一个用户名），其他数据库对象的命名也是如此。这样的命名系统允许不同的用户（如一个班级里的所有学生）使用同名的表，而不会在整个数据库中造成混淆。有些数据库中，除每个用户一个模式外，系统还使用附加模式来容纳表和其他对象的相关集，如DB2 UDB中的SYSCAT模式和ORACLE中的PUBLIC模式。

在Core SQL-99和当前大多数产品中，当用户的数据库账户建立时，其模式在用户名之后给出，他们不能再建立其他模式。SQL-99的一个扩展特性是允许用户建立附加模式，由用户给定模式名和对象所有权。下面是Create Schema语句的语法。

```
[EXEC SQL] CREATE SCHEMA
  [schema_name | AUTHORIZATION authorization_id]
  [{schema_element schema_element ...}];
```

X/Open的语法与其相同，只是还有一个给定字符集的附加可选子句。Core SQL-99和X/Open中，可选的schema_element是下列SQL语句：Create Table，Create View，Create Index和Grant。它们以正确的所有权来为新模式提供初始内容。用户只须给出模式名，AUTHORIZATION和schema_element选项可以跳过，接着在新模式中创建表和其他对象。在我们介绍的三种产品中，目前只有DB2 UDB允许用户建立附加模式。其他产品的文档中，“模式名”用“所有者名”或“用户名”代替。

2. 数据库产品间目录的不同

因为每个数据库产品对目录表的约定都不同，所谓的标准只能给出一个什么表和列必须存在的概念。本小节对ORACLE、DB2 UDB和INFORMIX中的目录表进行简要的描述。描述不是很完全的，要了解具体细节，请查阅相关的产品指南。

(1) ORACLE数据字典

ORACLE数据字典由视图组成，根据前缀可分为三种不同的形式：

前缀	用途
USER_	用户视图（用户所有的对象，在用户模式中）
ALL_	扩展的用户视图（用户可以访问的对象）
DBA_	DBA视图（所有用户都可以访问的DBA对象的子集）

^Θ 实际上，SQL标识符是不分大小写的。解释见例3.9.1后面的小节“标识符”。当对目录的查询中的表名为小写字母时，将被查询处理器作为字符串数据处理，即按照字符串的字面意思解释，而不去匹配目录表中的大写字母。INFORMIX目录中的表名不是大写字母，是小写字母。

例如，字典中有USER_TABLES、ALL_TABLES和DBA_TABLES视图。ALL_TABLES和DBA_TABLES视图有很多列，其中很多都是与磁盘存储和Create Table语句中的更新事务子句有关的，前面我们没有给出。下面是我们感兴趣的一些列。

ALL_TABLES(或DBA_TABLES)

列名	描述
OWNER	表的所有者
TABLE_NAME	表名
(其他列)	磁盘存储和更新事务信息

与ALL_TABLES不同，USER_TABLES视图没有OWNER列，因为根据定义，表的所有者就是当前用户。（实际上，USER_TABLES视图是根据ALL_TABLES视图定义的，从ALL_TABLES中选出OWNER = 当前用户的行，但结果中不出现OWNER列。）

用户可以访问的所有表和视图（还有聚簇，ORACLE中定义的一种类似于表的结构）的列的信息都存放在ACCESSIBLE_COLUMNS视图和同义的ALL_TAB_COLUMNS视图中。DBA_TAB_COLUMNS有相同的结构，USER_TAB_COLUMNS除了没有OWNER列之外也相同。

ALL_TAB_COLUMNS(或ACCESSIBLE_COLUMNS)

列名	描述
OWNER	表、视图或聚簇的所有者
TABLE_NAME	包含该列的表、视图或聚簇的名字
COLUMN_NAME	列名
DATA_TYPE	列的数据类型
DATA_LENGTH	列的长度，以字节为单位
(其他列)	其他属性：空值？缺省值？等

ALL_TAB_COLUMNS的主键显然包括TABLE_NAME和COLUMN_NAME，因为不同的表可能有相同的列名，不同的用户也可能使用相同的表名。为了在SQL中区分出这些名字，表可以用用户名（又称为模式名）来限定：用户名.表名(username.tablename)。这样，访问由poneil创建的orders表中的列（当前用户不是poneil，但有访问orders表的权限），可以用Select语句：

```
select column_name from all_tab_columns
  where owner = 'PONEIL' and table_name = 'ORDERS';
```

总是有比较简单的方法来描述用户已经创建的表中的列，例如用SQL *Plus的命令：

```
describe orders;
```

Describe命令也可用于用户定义的视图和系统定义的表和视图，但ORACLE 8i版本（8.1.5或更早的版本）的用户必须用一个特殊的前缀来修饰系统定义的对象名，如下：

```
describe "PUBLIC".USER_TABLES;
```

注意，前缀“PUBLIC”必须用双引号括起来，而且要用大写字母（见例3.9.1后的“标识符”小节中对用双引号括起来的字符串的解释）。ORACLE中还有称为ALL_TAB_COMMENTS和ALL_COL_COMMENTS的目录视图（以及相关的USER_和DBA_

变量)来包含描述被某些假定会来访问的DBA使用的表和列的描述。ORACLE语言有一种非标准的SQL语句: COMMENT ON tablename | tablename.columnname IS 'text'。用这条语句, 可以对拥有的任何一个表或列加文字注释。

另外, ORACLE有相对标准的目录视图TABLE_PRIVILEGES(或ALL_TAB_GRANTS)和COLUMN_PRIVILEGES(或ALL_COL_GRANTS), 列出可访问表上的权限, 还有CONSTRAINT_DEFS(或ALL_CONSTRAINTS), 列出可访问表上的约束。视图DICTIONARY列出所有的数据字典表和视图的名字(TABLE_NAMES)及描述性注释(COMMENTS), 视图DICT_COLUMNS列出这些字典列和视图的所有列(TABLE_NAME、COLUMN_NAME作为主键)和描述性注释(COMMENTS列)。我们没有提到的其他对象, 也在目录中列出, 如索引并根据需要对它进行说明。有关用户和性能的统计信息也记录在数据字典中, 我们将在第9章中介绍性能统计信息。

用户可以用如下的SQL*Plus序列创建系统定义的视图的列表:

```
spool view.names           -- or whatever file name you want to use
select view_name from all_views where owner = 'SYS'
    and view_name like 'ALL_%' or view_name like 'USER_%';
```

要知道一个视图, 如agentorders, 最初在ORACLE中是怎样定义的, 可以用如下语句:

```
select text from user views where view name = 'AGENTORDERS';
```

在7.2节之前的触发器定义中, 我们曾提到过设置文本的显示长度。用SQL*Plus的set long命令可以增加文本显示长度, 其缺省值为80个字符:

```
set long 1000
```

另一方面, 如果一列的文本过长, 可以用SQL*Plus的Column命令进行调整; 为了在本次会话的其他部分, 视图名view_name能够在20列中显示, 键入如下命令:

```
column view_name format a20;
```

要了解有关ORACLE数据字典的详细信息, 请参阅推荐读物[10]《ORACLE8 Server Administrator's Guide》(《ORACLE8服务器管理员指南》)。

(2) DB2 UDB系统目录表

DB2 UDB产品有很多目录表, 每个表又有很多列。用户可以用SYSCAT限定符(这是一个模式名)来访问这些表, 或者给表名加上SYS前缀并使用SYSIBM限定符。下面是SYSCAT.TABLES或同等的SYSIBM.SYSTABLES中的一些列。

SYSCAT.TABLES(或SYSIBM.SYSTABLES)

列名	描述
DEFINER	表或视图的创建者
TABNAME	表或视图的名字
TYPE	'T'=TABLE, 'V'=VIEW
(其他列)	磁盘存储和更新事务信息

DB2 UDB中有关于表的目录表(TABLES)、关于表内列的目录表(COLUMNS)、关于视图的目录表(VIEWS), 还有关于表、视图上权限的目录表(TABAUTH)。不同类型的约束记录在不同的表中, 例如, 表KEYCOLUSE包含每个外键、主键或值唯一的列。表

SYSCAT.COLUMNS中的列如下。

SYSCAT.COLUMNS (或SYSIBM.SYSCOLUMNS)

列名	描述
TABNAME	包含该列的表或视图的名字
COLNAME	列名
TYPENAME	列的数据类型
LENGTH	列的最大长度，以字节为单位
(其他列)	其他属性：空值？缺省值？等等

例如，要列出由poneil创建的表orders中的所有列，应该写：

```
select colname from syscat.columns
where definer = 'PONEIL' and tablename = 'ORDERS';
```

要了解有关DB2 UDB系统目录的详细信息，请参阅本章结尾处的“推荐读物”[1]列出的(《A Complete Guide to DB2 Universal Database》)《DB2 通用数据库完全指南》。

(3) INFORMIX系统目录

INFORMIX中有关于表的目录表(SYSTABLES)、关于表内列的目录表(SYSCOLUMNS)、关于视图的目录表(SYSVIEWS)，还有关于表和视图上权限的目录表(SYSTABAUTH)。SYSREFERENCES列出引用约束，SYSCONSTRAINTS列出列约束。SYSDEPEND列出表和视图之间的依赖关系。当对这些表进行Select时，不需要限定符。下面是SYSTABLES中的一些列。

SYSCAT.TABLES (或SYSIBM.SYSTABLES)

列名	描述
owner	表或视图的创建者
tabname	表或视图的名字
tabid	表的唯一标识符
tabtype	'T'=TABLE, 'V'=VIEW
type_xid	如果是类型表，则为表类型的扩展ID或为0

表SYSCOLUMNS有以下比较重要的列。

INFORMIX SYSCOLUMNS

列名	描述
colname	列的名字
tabid	表标识符，如在SYSTABLES中
coltype	0=char, 1=smallint, ..., 19=set, ..., 4118=自定义的行类型
extended_id	列类型的唯一标识符

extended_id区分不同的“自定义的行类型”(在面向对象的INFORMIX中用户定义的类型)；coltype为4118是表示所有自定义类型的通用类型，而每个自定义的类型都是互不相同的。注意SYSCOLUMNS和SYSTABLES中都有列tabid。要找出表orders的所有列的名字，我们可以通过连接tabid的语句来实现。

```
select colname from syscolumns c, systables t
```

```
where c.tabid = t.tabid and t.tabname = 'orders';
```

与ORACLE和DB2 UDB不同，我们用小写字母表示orders，因为INFORMIX把所有的名字都变成小写的，而不是大写的。要了解有关目录表的详细信息，请参阅本章结尾处的“推荐读物”中列出的推荐读物[8]《INFORMIX Guide to SQL》(《INFORMIX SQL指南》)。

3. 面向对象的目录表：ORACLE和INFORMIX

当用户在数据库中创建用户定义类型和用户定义函数时，不光要记录这些类型和函数的存在，还要记录它们的相互依赖关系，因此要更新很多目录表。在使用UDT和UDF时，很快会发现，处理这些依赖关系是非常麻烦的。如果用户定义的类型A中用到另一个用户定义的类型B，而这时要改变B的定义的很小的一个地方，则不得不去除A原来的定义，重新定义。事实上，不光是类型A，所有用到类型A和B的表格等都要作废。

有一种简化工作的办法是建立并维护一个SQL程序，来去除所有的内容，然后从头再建。在一个数据库系统中，如果用户有删除和建立数据库的权限，就不必费力地一个一个地删除，可以直接删掉整个数据库再重建。

即便使用如此谨慎小心实现的程序，还有可能发现某个地方的类型没有被删除，因为有些依赖关系没有被发现。这说明必须参考记录这些相互依赖关系的目录表。

ORACLE中，目录表给出了每个对象类型的顶层描述。表对象在USER_OBJECT_TABLES(或ALL_或DBA_)中而不是USER_TABLES中描述。所幸的是，我们要考察的依赖关系都在表USER_DEPENDENCIES中。这些表中最重要的列如下所示。

ORACLE USER_TYPES

列名	描述
TYPE_NAME	类型名
ATTRIBUTES	类型的属性个数
METHODS	类型的方法个数
INCOMPLETE	'YES'或'NO'：类型是否完整

ORACLE USER_DEPENDENCIES

列名	描述
NAME	依赖模式对象的名字
TYPE	对象的分类：‘TABLE’，‘TYPE’等
REFERENCED_NAME	父对象(该对象依赖的对象)的名字
REFERENCED_TYPE	父对象的分类：‘TABLE’，‘TYPE’等
DEPENDENCY_TYPE	‘HARD’(正常)或‘REF’(通过REF)

例如，在例4.2.3和例4.2.4中，我们建立了对象类型person_t，它的属性类型依赖于对象类型name_t。然后，我们创建了表people，一个类型为person_t的对象表，因此它依赖于person_t，而通过person_t又依赖于name_t。为得到表people的依赖关系，可以用如下语句：

```
select referenced_name, referenced_type from user_dependencies
  where name = 'PEOPLE';
```

输出如下：

referenced_name	referenced_type
STANDARD	PACKAGE
NAME_T	TYPE
PERSON_T	TYPE

从输出我们可以看到，表USER_DEPENDENCIES甚至包含了表people通过类型person_t依赖于类型name_t的间接依赖关系，它还指出了表people属于“标准”包，因为用户没有为它建立用户命名的ORACLE包。

同样方式的查询能和表一样有效地找出一个类型的所有依赖关系。因为汇集类型也是命名的类型，它们也适用于这种依赖跟踪系统。例4.2.9中orders与customers的REF可以通过ORDERS上这样的查询找到：要把REF依赖与HARD依赖区分开（表或类型的依赖关系），应该在查询中包括列dependency_type。（参见上面对ORACLE USER_DEPENDENCIES列的描述。）

INFORMIX中用附加表来记录类型和表之间的依赖关系，因为依赖性并不是在一张表中的。而且，用数值标识符表示表之间的连接列，而不是简单地使用字符串名。记录用户定义类型的顶层表是SYSXTDTYPES。我们需要表SYSCOLUMNS和SYSTABLES（本节前面介绍过）将数值标识符和表名、列名及其类型匹配。最后，我们需要表SYSATTRTYPES将字段类型与它们封装的行类型联系起来。下面是记录类型（行类型和汇集类型）、行类型的属性、汇集类型的元素类型的目录表中比较重要的列：

INFORMIX SYSXTDTYPES

列名	描述
name	类型名(如果是命名的行类型)
extended_id	类型的唯一标识符
type	与coltype类似；0=char, …, 19=set, 20=multiset, 21=list, 22=未命名的行类型, 4118=已命名的行类型

INFORMIX SYSATTRTYPES

列名	描述
extended_id	包含该字段的类型的唯一标识符
fieldname	字段名
xtd_type_id	字段类型的唯一标识符

要列出所有类型表及其类型，需要连接表SYSTABLES和SYSXTDTYPES，并匹配表类型的唯一标识符，这个标识符在SYSTABLES中称为type_xid，在SYSXTDTYPES中称为extended_id。这里，我们将列出表people的表类型：

```
select t.tabname, typ.name from systables t, sysxtdtypes typ
where t.type_xid = typ.extended_id and tabname = 'people';
```

下面我们举一个与ORACLE相同例子，找出与例4.2.4中的表people有依赖关系的类型。刚才的查询将会为表people输出类型名person_t。要找出被表people使用的行类型（或列类型），需要与SYSCOLUMNS做进一步的连接，见习题7.20中的讨论。然而，我们仍然会漏掉列类型的属性的属性。不幸的是，对依赖性进行完全的搜索，需要根据依赖链，而这

是一个传递闭包计算。传递闭包在第3章的3.11节讨论过。ORACLE把传递闭包的结果放在目录表USER_DEPENDENCIES中，从而为用户解决了这个问题。

在INFORMIX中，汇集类型没有特定的类型名，所以上面的查询将输出空的类型名。但它们有特定的扩展标识符，因此数字可以被打印输出并用于跟踪该类型。

推荐读物

第3章和第5章中提到过的各种SQL标准和SQL产品手册仍然是很有用的。

- [1] Don Chamberlin. *A Complete Guide to DB2 Universal Database*. San Francisco: Morgan Kaufmann, 1998.
- [2] *DB2 Universal Database Message Reference*. IBM. Available at <http://www.ibm.com/db2>.
- [3] *DB2 Universal Database Administration Guide*. IBM. Available at <http://www.ibm.com/db2>.
- [4] *DB2 Universal Database SQL Reference Manual*. IBM. Available at <http://www.ibm.com/db2>.
- [5] *Data Management Structured Query Language (SQL)*. Version 2. Berkshire, UK: X/Open Company, Ltd. Email: xospccs@xopen.co.uk.
- [6] Jim Melton and Alan R.Simon. *Understanding the New SQL: A Complete Guide*. San Francisco: Morgan Kaufmann, 1993.
- [7] *INFORMIX Error Messages*. Version 9.2. Menlo Park, CA: Informix Press, 1999. <http://www.informix.com>.
- [8] *INFORMIX Guide to SQL: Reference*. Version 9.2. Menlo Park, CA: Informix Press, 1999. <http://www.informix.com>.
- [9] *INFORMIX Guide to SQL Syntax*. Version 9.2. Menlo Park, CA: Informix Press, 1999. <http://www.informix.com>.
- [10] *ORACLE8 Server Administrator's Guide*. Redwood Shores, CA: Oracle. <http://www.oracle.com>.
- [11] *ORACLE8 Server Error Messages*. Redwood Shores, CA: Oracle. <http://www.oracle.com>.
- [12] *ORACLE8 Server Concepts*. Volumes 1 and 2. Redwood Shores, CA: Oracle. <http://www.oracle.com>.
- [13] *ORACLE8 Server SQL Reference Manual*. Volumes 1 and 2. Redwood Shores, CA: Oracle. <http://www.oracle.com>.
- [14] *ORACLE8 Server Application Developer's Guide*. Redwood Shores, CA: Oracle. <http://www.oracle.com>.

习题

有*标记的习题在书后的“习题解答”中给出了答案。

在下面的习题中，如没有特别说明，所有SQL语句都是基本SQL形式，若不存在基本SQL

形式，可采用某种适当产品的语法。

- 7.1 (a) *根据例7.1.2中customers和orders的定义，写出带有完整性约束的创建表agents和products的Create Table语句。注意，percent应在0和10之间，quantity和 price大于0（假定0是有效的数据类型的形式）。
- (b) 假定DBA要求customers可能的discnt值在0.00和10.00之间，而且数值之间的差距只能为0.02，所以可接受的值为0.00, 0.02, 0.04, ..., 9.96, 9.98, 10.00。请用适当的Create Table语句实现这样的约束。注意，因为可能的值很多，所以用CHECK子句是不合适的；需要另外定义一个表来实现这一约束。
- 7.2 写出定义6.4节末尾的表passengers, gates, flights和seats的Create Table语句。使表正确地反映图6-14中实体-联系中参与方的基数。
- 7.3 *将图6-11中实体-联系的基数补充完整，然后设计关系表，最后写出能正确反映你的设计的Create Table语句。
- 7.4 用Create Table语句中提供的约束，下面(a)、(b)、(c)三种情况中，哪一种可以正确表示联系中实体的强制参与？
- (a) *当实体是N-1联系的“1”方。
 - (b) 当实体是N-1联系的“N”方。
 - (c) *当实体是N-N联系的任何方。
 - (d) 使用参照完整性，可以保证一个表的外键（可能代表一个联系实例）引用另一个表中确实存在的主键（可能代表实体参与了那个联系）。然而，参照完整性不能保证实体在联系中强制参与。你能否描述一种（假想的）完整性来实现这种约束？
- 7.5 (a) 根据X/Open中更新视图的约束，下面哪一条SQL语句是合法的？（括号中说明了视图是在哪个例子中创建的。）
- (i) update agentorders set month = 'jun'; (例7.2.1)
 - (ii) update acorders set month = 'jun' where pid = 'c001'; (例7.2.5)
 - (iii) update agentsales set aid = 'axx' where aid = 'a03'; (例7.2.7)
- (b) *假定我们没有在习题7.1的(a)中的Create Table语句中用完整性约束来限制列percent的值。在表agents上建立一个可更新的视图agentview，并防止任何更新agentview的用户，将percent变成小于0或大于10的值。
- (c) 同样假定我们没有对表agents的列percent用完整性约束进行限制，在ORACLE, DB2 UDB 和INFORMIX中都可以为表加上完整性约束而无须重建表。给出某些特定产品中的相应命令。
- (d) *我们希望用两个语句来授予用户Beowulf一些权限，使他能够访问表products的pid, pname, city和quantity（不是price）列，并且能更新city和quant列（但不能更新其他列）。写出实现这些功能的语句。
- 7.6 判断下面的说法正确与否，并给出说明。最好引用本文的内容（定义、例子、图或某一页的讨论）来支持你的观点。
- (a) *表的某行中允许有构成主键的列中有一个为空。对还是错？
 - (b) 表的某行中允许有构成外键的列中有一个为空。对还是错？

(c) 在基本SQL的Create Table语法中，可以用包含特定子查询的CHECK子句实现 FOREIGN KEY ... REFERENCES约束。对还是错？

(d) 尽管基本SQL中可以用约束来定义主键，但不能定义表的其他候选键。对还是错？

7.7 所有的数据库产品都允许DBA在完整性约束中指定一些供选择的动作，当从一个表中删去一行，而该行的主键值可能正被另一个表作为外键引用时，采用这些动作。其中的一种产品有多种可能的动作。

(a) 给出所有产品和某个特定产品中可能的动作名。

(b) 给出SQL产品中缺省执行的动作名。

(c) 参照图2-2中CAP数据库的内容，说说如果删除顾客c001，而(b)中列出的动作起作用，会发生什么结果。

7.8 假定表customers以例7.1.2的形式定义。我们想创建customers的精确副本customers1，包括customers的所有行和约束。

(a) 写出在ORACLE中创建customers1的语句，customers1要包括customers的所有行，要求用一条语句完成。

(b) 在基本SQL中用两条语句完成同样的任务（不用AS子查询子句）。

(c) 说明为什么在ORACLE中执行下面的Create Table语句时会出错，并改正错误。

```
create table customers1(cid primary key, cname, city,
discnt check (discnt <= 15.0)
as select c.* from customers c, orders o where c.cid = o.cid
and o.qty > 1000;
```

7.9 假定有一个表employees，它有eid, ename和mgrid列，还有两个列salary1和salary2为浮点数分别表示两种不同的工资（可能是对不同的工作项目）。注意，在SQL中，FLOAT是DOUBLE PRECISION的另外一个名字。现在假定用Create View语句创建一个视图emps：

```
create view emps (eid, ename, mgrid, totsal) as
select eid, ename, mgrid, salary1 + salary2 from employees;
```

这是一个不可更新的视图。它违反了什么规则？给出一个对视图emps进行更新的例子，而这种更新很难转化成对基本表employees的修改（即不清楚应该对基本表employees进行什么样的修改），由此说明这条规则存在的道理。

7.10 [难]例7.2.6显示了一个视图colocated，当我们想要删除一行或修改一列时，两个表customers和agents的city列连接将会引起无法预料的后果。如果我们通过连接表agents和orders的aid列定义视图agentords（外键与主键的自然连接），又会出现什么情况呢？考虑到要对得到的视图进行更新、删除和插入，我们想允许对这种自然视图进行的更新。说明当对视图进行这些操作时，基本表会发生什么变化。注意，各人的想法可能有些不同。

(a) 插入操作中，如果插入的是一个新的aid值会怎样？如果插入的是一个已经存在的aid值但aname值不相同又会怎样？（根据函数依赖，你认为会发生怎样的情况？）

(b) 应该如何处理删除操作？

(c) 更新操作中，分别考虑表agents和表orders中的不是键的那些列和是键的列

ordno, aid, cid和pid。

上机作业 为下面的习题中编写过程或程序。

7.11 关于触发器的练习。

- (a) 创建一个叫做agent_city的触发器，保证新插入表agents的行的城市值必须是表customers中的城市之一。注意，为了引用新插入的agent_city，触发器必须被定义为AFTER INSERT的。尽管是在行插入后对表进行检查的，如果被触发的触发器调用ORACLE的raise_application_error或DB2 UDB的信号函数，插入将失败。对你的触发器进行测试——如果它是正确的，可以使插入失败，如果不正确则不能——尝试两种不同的Insert语句。
- (b) 创建一个触发器，当向表orders中插入一个新订单时被触发，自动地更新表products的quantity列。触发器必须把在orders中指定的qty从products相应行的quantity中减去。
- (c) 假定非过程性约束将导致插入一行的失败。在这种情况下，BEFORE INSERT触发器可以触发吗？设计一个简单的实验来证明你的结论。

7.12 在你班连接到CAP的数据库系统的交互式监视器中，用一个命令列出所有的视图。这可能需要查询系统目录。先在表customers的基础上定义视图custview，选出discnt < 12.0的行，然后在视图定义中加上with check option子句。再列出所有视图，看看custview有没有加进来。执行Select语句select * from custview。下面试着修改custview，使customers中cid等于c001的行的discnt值等于13.0。可以进行修改吗？执行select * from customers语句，可以看到修改没有成功。现在不写with check option子句，重新定义视图custview（需要先删掉原来那个），再做一次更新。可以看到这次成功了。暂时让表customers处于被修改状态。

7.13 如果你正在使用ORACLE数据库系统，请在连接视图上做以下习题。

- (a) 首先要确定你的所有CAP表格中都定义了主键。然后创建一个视图prodords，选出products和orders的连接表中的所有列，子查询中只对pid一个列给出别名ppid。接下来执行语句

```
select column_name, updatable from user_updatable_columns
where table_name = 'PRODORDS';
```

修改一个查询结果中UPD列为YES的视图，并检查更新是否会影响基本表，以确定你是否可以更新这些视图。同样，检验一下，UPD为NO的视图是否不能被更新。

- (b) 对例7.2.6的视图colocated重复(a)中的步骤。
 - (c) 对视图apords重复(a)中的步骤。apords通过对表agents、products和orders的连接得到，只保留一个aid列和一个pid列，将它们分别重命名为aaid和ppid，将a.city和p.city分别重命名为acity和pcity。然后重复(a)中的步骤。
 - (d) 删除表products，再重建一个没有主键的表products（只要求对列pid使用not null unique）；然后重新装入表格。再对products重复(a)中的步骤。当完成了这一习题后，重新装入所有修改过的CAP表。
- 7.14•** 在你的SQL监视器中，用一条命令列出所有完整性约束。可能需要查询系统目录。

接上一习题，我们假定cid等于c001的行的discnt值等于13.0。在表customers上建立一个完整性约束（如果你的系统支持命名的约束，请给出命名的约束），要求 $\text{discnt} \leq 12.0$ 。这样能行吗？会出现什么情况？修改表customers，使cid等于c001的行的discnt值为10.0，再尝试建立完整性约束。接下来修改customers，使cid等于c001的行的discnt值为16.0。这样的修改可以实现吗？通过执行`select * from customers`说明修改没有成功。现在去掉约束，再进行修改，可以看到这次修改成功了。最后，将表customers回复原值并重新装入。

7.15 (a)•在表orders中插入一行：1031, jul, c001, a01, p01, 1000, 450.00。基于表orders上建立一个视图returns，包含列ordno, month, cid, aid, pid, qty, dollars以及discnt（来自表customers），percent（来自表agents）和price（来自表products）。用SQL语句查询相应的目录表，以确认视图returns确实存在。同时确定各列的名称。

(b) 执行语句`select ordno, qty, dollars, discnt, percent, price from returns where cid = 'c001'`。然后将基本表customers中c001的discnt值改为13.0，再对returns执行Select语句。可以看到，对基本表的修改在视图中反映出来了。最后，将c001的discnt值回复为10.0（或直接重新装入customers）。

(c) •注意，表returns中的列dollars应该等于qty乘以price再减去该金额的discnt值。测试一下你的能力，选择视图returns中所有行的ordno, qty, dollars, discnt, percent和price，并在此基础上写表达式，用Select语句中的一个附加列来计算表达式的值。这个附加列的值应该与dollars相等。

(d) 在视图returns的基础上建立一个新的视图profits，包括列ordno, cid, aid, pid和profit，其中profit列已经在Create View语句中计算过，等于qty乘以相应pid的price，若为批发，应减去该值的60%，再减去给某个特定代理商的percent和对当前顾客的discnt。确保表达式的正确性。执行`select * from profits`，看看结果如何。

7.16 在班级内指定一个同伴，与你的同伴相互交换“权限”。授予你同伴的数据库账号完成选取（只能选取）表customers中 $\text{discnt} < 10$ 的行的权限。需要建立一个视图，取名为custview，来达到这个目的。使用相应的命令来显示视图和权限的存在。保持授权的有效性，演示你如何访问同伴的数据库。（你需要先登录数据库，再执行语句`select * from custview`。）

7.17 说明如何通过系统目录表检索你的数据库中所有的视图名及基本表名。从视图returns检索所有的列名。能通过目录检索出视图returns的定义吗？

7.18 编写一个嵌入式SQL程序修改customers，将cid为c001的行的discnt值加1。在程序中包括检查是否没有行被修改而且给出了用户警告的测试。在customers上设置一个约束，使 $\text{discnt} \leq 15.0$ ，再执行程序直到出现警告。

7.19 在7.1节的最后一小节中，本文建议开发一个具有程序功能的函数层来执行所有的数据库更新操作。层本身要保证必要的约束，还要完成其他功能，如在约束被打破时执行供选择的动作。为解决这个问题，需要建立一个新表ords，它包括表orders的所有列，数据

类型也相同，但不包括orders的约束，甚至不要求ordno为NOT NULL的。表ords还另外有一列，称为constrok，为integer类型。下面编写一个C函数insertords()，其声明如下

```
int insertords(int *ordnop, char *monthp, char *cidp,
    char *aidp, char *pidp, int *qtyp, double *dollarsp);
```

若程序要往表ords中插入一行，就必须调用insertords，并用指针指向要插入的各相应列的值。空指针表明没有指定值并插入一个空值。函数insertords必须对例7.1.2中给出的表orders的所有11种约束进行测试。如果欲插入的行通过了测试，函数可以将它插入到ords中，并置constrok列值为1，返回0。如果有任何约束不满足，函数将给用户提示信息（如“Error: null ordno, continue? Enter Y or N”）。如果用户给出了Y和N之外的答案，回显示“Y or N, Please”信息，提示用户重新输入。如果键入N，函数insertords不插入该行，并返回-1。如果键入Y，函数insertords将该行照原样（诸如不满足约束等）插入，置标志constrok为0，并返回1。教师将提供一个驱动程序driveord.c来测试这个函数，并给出驱动程序的输入。注意，这种方法并不是要取代约束，只是要比较一下两者的方便性，同时说明如果数据库产品不支持你需要的约束时该怎么办。

7.20 • (面向对象的INFORMIX) 写一个对目录表SYSTABLES, SYSCOLUMNS和SYSXTDTYPES的查询，找出被例4.2.4的表people用作列类型的行类型。这个查询将找到name_t，因为它被用做了列类型，而person_t则不是，它是整个行的类型（已经在本文的查询中找到）。当然，我们可以把这两种查询UNION起来。但这样还是不能找到一个为行类型列的属性的类型。为了把这些也找出来，写一个同时还使用表SYSATTRTYPES的查询（总共用4个表）。

第8章 索引

当一个SQL查询提交给数据库系统的时候，一个称为查询优化器的软件模块将对查询中的非过程命令进行分析，然后决定一个能一步一步地访问到所需数据的步骤。这种执行查询的步骤被人们称为访问计划或者查询计划。在第9章中，我们将详细讨论查询优化器是怎么制定出访问计划的；在本章中，我们通过阐述查询如何利用数据库索引来提高访问表中数据的效率来奠定一个基础。

8.1 索引的概念

数据库索引，简称为索引，与读者以前学习过的驻留内存的数据结构有些类似。它的目的是提高对表中行的数据查找的效率。在学习本章前，我们假设读者对支持这类查找的驻留内存的数据结构已经是非常熟悉的了，这些数据结构包括二叉树、2-3树和散列表（有很多书本都包括这些数据结构的讨论，参见本章末尾的“推荐读物”）。数据库索引区别于驻留内存的数据结构的地方在于，数据库索引包含的数据量比一次能调入内存的数据量大。因此，数据库索引的数据是存放在磁盘上的，只有被访问的时候才会被部分地调入内存。这样做的另外一个优点是，当计算机关闭的时候，驻留内存的数据将丢失，但是数据库索引中的数据像表中的记录一样会永久存在。

索引是由一系列存储在磁盘上的索引项组成。一个索引项对应于索引中的一行，当行发生更新的时候，索引也将做出响应。这意味着如果一个行按某一列中的值访问，索引将提供这种访问；如果行中该列的值被修改了，索引也将做相应的变化。索引项有点儿像一个由两列组成的表：第一列是索引键，由行中某些列（通常是一列）中的值串接而成；第二列是行指针，指向行所在的磁盘位置。索引项存储磁盘上，通常是按照索引键排序的（尽管有的时候可能是按散列访问），这样就可以提高特定的SELECT语句的查询速度。典型的通过索引查询是给定一个键值或者键值的范围，找到索引项，然后根据行指针找到相应的行。数据库索引通常是存放在磁盘上的，但是和访问内存相比，磁盘访问是相当慢的。这个事实对于我们将来要学习的数据库索引结构（例如B树）有重要影响。数据库索引设计的最重要的目标就是要减少读数据所需的磁盘访问的次数。

读者可以把数据库的索引设想成图书馆中旧式的目录卡片，这些目录卡片对书架上的书目按类别做了索引。目录册中的一些卡片是按作者的字母顺序放置的，另外一些是按标题排序的，还有一些是按主题名排序的。目录册中的每一张卡片都记录了该书的书架号码，所以一旦读者找到了《*The Three Laws of Robotics*》一书的目录项，就可以立刻在书架上找到这本书。现在，我们再返回到关于索引的讨论中来，让我们看看处理下列SQL查询的时候，数据库系统都做了些什么。

```
[8.1.1] select * from customers where city = 'Boston'  
        and discount between 12 and 14;
```

查询优化器将要决定怎样从customers表中访问到上述查询需要的行。一个可选择的方

法是执行一次表扫描，将对表中所有行进行连续访问，把那些不满足WHERE子句中的两个And的谓词的行剔除。（在这种方法中，我们是直接检查行中数据的。）如果customers表中的行数小于10（如图2-2所示），那么扫描整张表可能是最好的策略；如果表中的行数不大的话，直接检查所有行将是非常快速的。类似地，如果书架上的书只有不到10本，那么在目录卡片中查询书并不比直接查询有效。但是，如果表中有一百万行，通过索引结构来查询记录将是非常有效的。

如果对一个有大量记录的customers表执行[8.1.1]中的查询，而customers表中discont属性又没有索引，只有city属性上有索引，正确的查询过程是什么呢？在这种情况下，首先系统将决定需要查询在Boston的顾客，然后通过上面讲述的行指针来访问所有在Boston的顾客。将查询限制在Boston中，将大大减少访问的行数。然后我们只要在挑选出的Boston的顾客中，选出discont属性又在12至14之间的记录就可以了。

最终用户或者提出查询的程序员不需要知道提高访问速度索引的存在。当一个应用程序执行[8.1.1]中的查询的时候，查询优化器将根据各个因素（例如是否有索引存在）来决定如何满足该要求，它可能和最初的查询已经有了很大的变化。事实上，关系模型的一个重要特征就是程序员可以不管索引结构是否存在，而编写出不依赖于特定索引的程序（因为总是可以使用扫描表的方法）。这也是第1章中定义1.3.1讲述过的程序-数据独立性的一个方面。在这种情况下，数据库管理员负责创建和删除索引。但是，这只是理论上的情况。在实际情况下，用户是可以注意到使用正确索引后执行某个查询只需要2秒钟而不使用索引时执行该查询需要两分钟之间的差别的。如果资源消耗严重影响到系统的总性能，可能就需要删除这些索引结构了。

在考虑索引的很多方面的时候，我们将进入SQL标准很少涉及的内容。为了能更好地说明，图8-1给出了X/Open标准的SQL语句。

```
CREATE [UNIQUE] INDEX indexname
    ON tablename (columnname [ASC | DESC] [, columnname [ASC | DESC]]);
```

图8-1 Create Index语句的语法

Create Index语句中的表名必须是发出该语句时已经存在了的基本表，指定的列名必须是没有被限定的。图8-1中的语句将在磁盘上创建一系列的索引项，一条索引项对应于表中的一行。上面我们已经讲过，索引项类似于一个含有两列的表：一个是索引键，由Create Index语句中指定的列的值串接组成；另外一个是指向磁盘位置的指针。当指定了UNIQUE子句后，索引中存放的索引键值必须是唯一的，除非有一个串接到键值的列是空值；在这种情况下重複索引键值是允许的。索引项将按索引键值的顺序存放在磁盘上的（在标准SQL中，散列访问是不被支持的），索引键值的顺序可以是升序，也可以是降序，像SELECT语句中的ORDER BY子句一样。对于给定的键值或者键值的范围，系统将在索引中找到相应的索引项，然后根据行指针找到相应的行。值得注意的是，当执行完Create Index语句后，索引中的内容将自动随表中内容的变化而变化。如果表中插入了一行，索引中也会创建一条新的索引项，然后把该索引项放在索引结构中正确的位置上。类似地，如果行中索引键的值发生了变化，索引项也会发生变化：旧的索引项将会被删除，然后在正确的位置插入一条新的索引项。

例8.1.1 对customers表的city列创建索引：

```
create index citiesx on customers (city);
```

值得注意的是，`citiesx`索引中的每一个索引键值（例如`city`名）都对应于`customers`表中的很多行。索引创建后，如果需要删除索引(`citiesx`)，可以用下面的标准SQL语句：

```
drop index citiesx;
```

值得注意的是，索引键这个术语和表键的关系概念（主键和候选键）之间有不同的含义。索引键是根据一些列按特定的顺序创建的，而关系键中列是没有顺序的。另外更基本的是，索引键是为了能更有效地查找数据，如果Create Index语句中不使用UNIQUE子句，那么同一个索引键值可以对应于表中的多个行。在例子8.1.1中，我们就看到了这样的重复的索引键值。另一方面，关系键并不和提高查找效率有关，键（主键或者候选键）是通过Create Table语句创建的，可以使用NOT NULL、UNIQUE或者PRIMARY KEY子句。但它只是反映了对于每一行都应该存在一个单一的键值。如果有可能引起理解混乱的话，我们将使用术语：索引键和表键（或者其他表示表键的术语，例如主键和候选键）来区别这两者。

当然也可以在表的某些列上使用唯一索引，以达到完成唯一性约束的目的。当UNIQUE和PRIMARY KEY成为Create Table语句的标准组成部分前，在一些数据库产品中，就是使用唯一索引来保证后选键的唯一性约束。在唯一索引中仍然可能出现空值，但是可以对候选键中所有的列包括NOT NULL，就像现在Create Table语句中使用了UNIQUE约束一样。

例8.1.2 假设我们没有声明`customers`表中的`cid`列是UNIQUE或者PRIMARY KEY。可以创建一个索引保证每一行都有唯一的`cid`值。

```
create unique index cidx on customers (cid);
```

当索引`cidx`第一次创建的时候，系统将测试`customers`表中是否存在重复的`cid`值，如果存在重复值（重复的空值除外），那么索引将不会被创建。索引创建完毕之后，对该表的所有更新操作将会对该索引产生影响。任何可能造成重复索引键值的插入或者更新操作都会被拒绝执行。

现在有了Create Table UNIQUE和PRIMARY KEY两种约束，使用这两种约束比使用通过索引的唯一性约束要好：查询优化器可以利用这一点，特别是当知道表的主键是什么的时候。但是，这些Create Table的唯一性约束通常是通过一个由系统创建的唯一索引完成的，因为插入新的记录或者影响索引键值的更新操作都需要快速的查找操作。

图8-1中的Create Index语句的X/Open标准，和一些主要的数据库系统产品的Create Index语句相比要简单得多。但是即使如此，它仍然不被SQL-99标准支持。Create Index没有标准的原因是，索引策略需要考虑数据库系统的内部体系结构，包括显式地设计访问磁盘上记录的方法，而现在对于如何优化设计还没有一致的观点。事实上，有多种类型的索引结构，而且如果不了解磁盘访问的某些细节，是不可能了解这些类型的优点的。举个例子，我们将学习一种称为二分查找法的算法，对于查询内存中已经排序了的列表中的键值而言，它几乎是最有效的方法；但是对于基于磁盘上的结构，它就不是最优的方法。所有商业数据库系统都采用不同的——可以称完全不同的——方法来存放行的实际的数据结构、分配引用的指针并完成其他在下面要讨论的很多细节方面。同时，虽然不同的产品之间有很多不同点，但是体系结构上也有很多相同点，我们将着重讨论ORACLE和DB2 UDB中的索引细节。我们不讨论INFORMIX的索引细节是因为有INFORMIX的不同版本INFORMIX XPS，它的索引结构很复

杂。有计划想让两个INFORMIX版本结合，到时，我们会在我们的主页`www://www.cs.umb.edu/~poneil/dbppp.html`上提供有关INFORMIX索引的细节。

8.2 磁盘存储

我们已经讲过，基本表中的行和创建的索引是存储在磁盘上的。当它们被访问的时候，将被读入到内存中。在第1章中我们说过，数据库必须要保证持久性，所以磁盘是最适合的存储介质。我们称计算机内存是易失性存储器，因为虽然可以快速访问内存，但是如果计算机电源关闭或者其他引起计算机系统故障发生的话，内存中的数据是无法保留下来的。磁盘被称为非易失性存储器，或者有时候被称为持久性存储器，意思是即使电源关闭上面的数据仍然不会丢失——正如每一个人都知道的可以把磁盘上的数据从一台计算机复制到另外一台计算机。在本书中，读者可以学到其他许多软件方法，这些软件方法可以帮助系统从系统故障甚至磁盘故障中恢复。磁盘很便宜，因此在许多系统中使用，但是磁盘的访问速度很慢。我们将要看到，由于磁盘访问速度很慢，所以需要使用数据库的索引结构。

1. 磁盘访问速度是极慢的

几乎每一个人知道，近几十年来，计算机变得越来越快，也变得越来越便宜。据说，如果从1950年开始，汽车工业的发展速度和计算机工业发展一样快的话，读者现在就可以花九美元买一辆汽车，只消耗一加仑的汽油，花一小时就可以从地球开到月球。而且计算机工业的发展速度还将继续保持。现在一台不算很贵的计算机的CPU每秒可以执行十亿条指令(100MIPS)。当然，执行不同的指令花费的时间是不同的，但是，这个速度是对相对较快的计算机上的一个标准指令集测出的大致速度。

磁盘访问速度是影响数据库系统性能的重要方面。磁盘访问速度的提高并不和CPU执行速度的提高同步。磁盘是一种旋转的磁性记录介质，通常是一些盘片堆在一起组成。现在一个中等性能磁盘的标准旋转速度是每秒120转，另外还有一个磁盘臂用于读取数据，就象老式的唱机一样。(磁盘的旋转速度和磁盘臂的移动速度在最近五年内翻了一番，而且随着新产品的问世，将不断提高)。磁盘臂的一头是一些位于磁盘表面的读写磁头。当磁盘臂移动到指定位置时，磁头也同时移动以寻找同心柱面上的数据(这些同心柱面是由许多盘片表面上的磁道组成)。一个磁道又分成若干个扇区，每个扇区通常包含512个字节。现在，一个普通的磁盘可以包含10GB的数据，每一个磁道包含200个扇区(各个产品实际包含的扇区数可能有些变化)，每个扇区包含512个字节。磁盘通常有10个磁盘表面(因此一个柱面大约包含1MB的数据)和大约10 000个磁盘柱而。为了读写磁盘上的数据，磁盘臂必须先快速地移动到正确的柱面位置，然后等待磁盘旋转，直到磁头正好指向指定的扇区。这样磁头从某一个旋转介质的盘片表面上的一系列的扇区中读取数据，这一系列的扇区被称为磁盘页面，或者简称为页面。一次磁盘页面访问通常由三个过程，这三个过程的度量标准如下：

寻道时间： 磁盘臂移动到指定柱面的时间。

旋转延时： 磁盘旋转到指定的扇区的时间。

传输时间： 磁盘臂读写相应表面上的磁盘页面的时间。

因为磁盘访问需要一个物理操作，所以在磁盘上随机读取一个页面数据需要的时间比运行一条计算机指令所需的时间要多得多：一次磁盘访问需要大致0.0125秒，即1/80秒。这0.0125秒是大致按下列时间分配的：

寻道时间： 0.008秒。

旋转延时： 0.004秒。

传输时间： 0.0005秒（大致传输几千个字节）。

寻道时间的变化范围很大，主要依赖于磁盘臂起始位置和要移动到正确柱面的距离。如果两个要连续读取的数据块在磁盘上是紧挨着的，那么寻道时间是很短的；如果这两个数据块在同一柱面上，那么寻道时间为零。0.008秒的平均寻道时间，是假设在这两个数据块在任意柱面上的概率是相等的基础上计算出来的。这里的旋转延时和转速为120转／每秒的磁盘旋转半圈所需的时间是相等的，因为我们假设需要读取的扇区可能处于磁道上的任何位置。

在一台CPU速度为100MIPS的计算机上，一旦数据被读入内存，该数据可以被一条指令在0.000 000 01秒（0.01μs）内访问到。磁盘访问速度和内存访问速度之间的差别是巨大的：在读写一个磁盘页面的时间里，我们可以执行1 250 000条指令。我们做个类比，假设你是美国革命时期伏尔泰的秘书，负责复制伏尔泰的信。我们假设你需要一秒钟来复制一个单词，我们可以把写信比作执行某个程序中的几百条指令。现在，你发现有一个单词因为伏尔泰字迹潦草而无法辨别。不幸的是，伏尔泰本人现在在圣彼得堡，你需要写一封信给他本人来询问这个单词的拼写。（类似地，当数据库系统需要在磁盘上读取一些信息，系统需要执行一系列的指令来完成该读取操作）你给伏尔泰写了信，回到办公室等上六个星期，直到伏尔泰回信。这和系统等0.0125秒直到磁盘上的数据被读到内存中很类似。很明显，我们希望尽量避免执行磁盘读取操作，除非是必须的。

需要说明的是，一旦磁盘臂移动到磁盘上的正确位置，你不要只读取一两个字节。读取一两个字节的传输时间是很短的，磁盘访问时间主要是移动磁盘臂到指定的位置所需的时间，一旦移动完毕，传输数据的速率是每秒几百万个字节。所以，我们可以发现所有的磁盘访问都是“面向页面的”。对于很多产品，一个页面的大小是2KB例如ORACLE，而另外一些产品一个页面的大小是4KB例如DB2 UDB（DB2 UDB和其他一些产品还支持更大的页面，例如8KB、16KB和32KB，但是对于大多数用户而言，标准页面大小还是4KB）。每次读取2KB或者4KB的数据到内存中，就好象给伏尔泰写一封长信询问大量的问题，秘书只不过多等几个小时而已。很明显，这样做很值得，因为从磁盘上读取一个页面，并把它保留在内存中，这样下一次访问这一页面的时候就不用再从磁盘上读取了。图8-2说明了数据库系统是怎么样对磁盘页面在内存中做缓冲的。

在图8-2中，我们看到数据库系统按照给定的磁盘页面地址（用符号dp表示）读取磁盘页面，并把这些磁盘页面存放到内存中被称为缓冲区的地方（这些缓冲区是在数据库系统初始化的时候建立的）。磁盘页面地址(dp)可以是连续的整数(1,2,3,...)，也可以由设备号、柱面号、磁盘表面号和开始扇区地址组成。每一个读入的页面在一个称为散列后备表的结构里都有一个对应的项，该项指向了该页面在缓冲区中的位置。每一次读取页面的时候，我们首先对页面地址做散列操作，然后在散列后备表中查询该页面是否已经在缓冲区中。如果该页面已经在缓冲区中，那么我们将忽略磁盘访问这一步骤；否则，我们将在缓冲区中找到一个页面，让新读入的页面替换它。我们将替换长时间不使用的页面，以便缓冲区中的页面总是经常被使用的页面。

既然我们要从磁盘上读取整个页面，我们将对访问要求进行组织，以便所有需要的信息都在同一个页面上。我们将要看到，这对索引结构有重要的影响。一个索引结构的最重要的特性是最小化访问磁盘操作的数目。

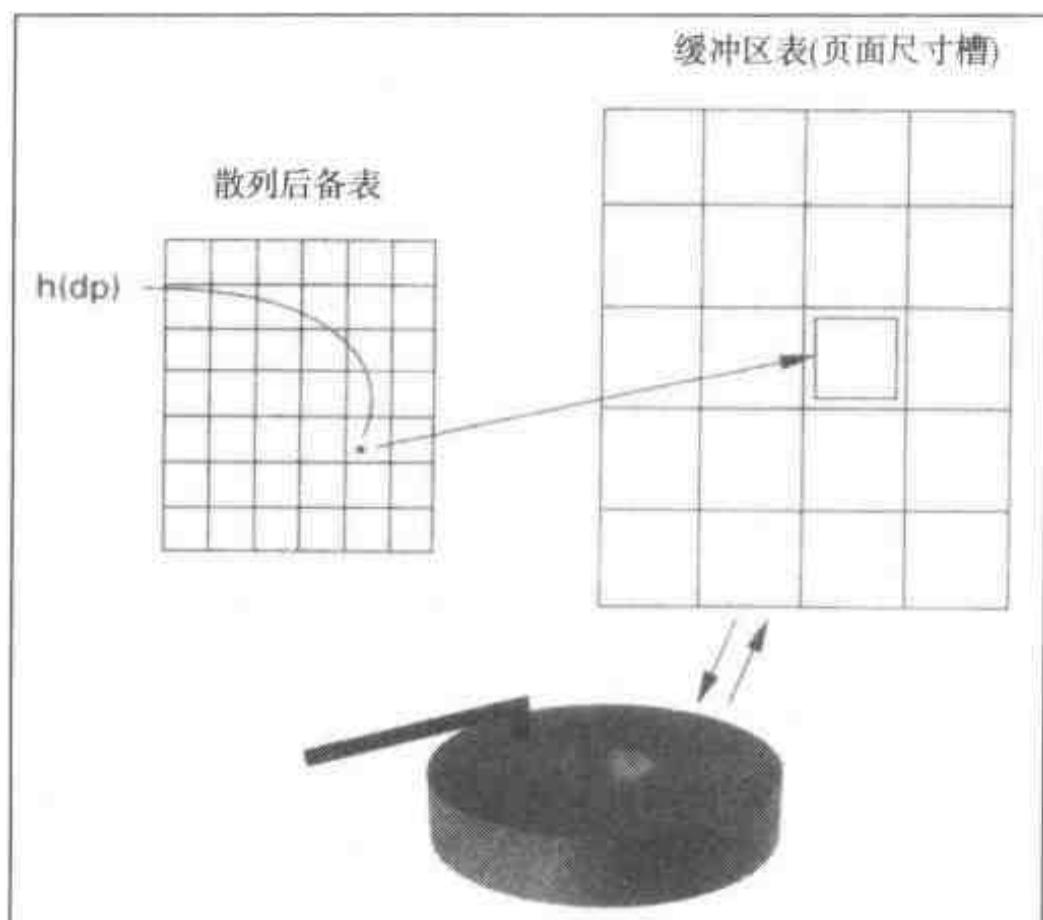


图8-2 磁盘页面缓冲及后备

既然磁盘访问比内存访问要慢很多，一个很自然的问题是：为什么不完全使用内存来进行数据访问呢？当然，我们已经提到过，内存是不稳定的，也就是说当计算机因为某种原因崩溃的时候，内存中修改过的数据将会丢失。但是具有事务处理特性的数据库的一个主要目标是，当计算机崩溃的时候，能保证内存中更新过的数据不会丢失。这一特性被称为事务性恢复，我们将在第10章中进行详细阐述。即便如此，事务性恢复需要两个基本条件，一是数据被备份到非易失介质上，二是恢复日志被记录到稳定的介质上。现在，所有主要的数据库系统都把磁盘作为非易失性介质，并并行地写入两个独立的磁盘来模拟稳定介质，这样能避免因为其中一个磁盘的故障而导致丢失日志信息。

所以在数据库系统中，我们不能没有磁盘，但是这样就提出了在CPU速度这么快的情况下，如何使用低速磁盘的问题。为什么我们不能把所有可能访问的数据都先放在一个很大的缓冲区(如我们在图8-2中的那种)中，这样来保持所有的访问操作都在内存中进行呢？事实上，很多数据库是这么想的。实现缓冲区所涉及的算法是很有效的，所以当缓冲区中的页面个数增加的时候，所需要的CPU时间不会增加。很多大型数据库的用户购买了越来越大的内存来提高访问速度。但是我们还没有意识到这种方法的潜力。在不久以前，对于大多数企业来说，内存的价格还是很高的，他们不太可能购买几吉字节的内存来装载数据，所以磁盘是他们唯一的选择。和个人计算机相比，服务器的内存价格还是很高的，但是价格不再是决定购买磁盘还是内存的主要因素。下面是对内存和磁盘价格的估计：

内存价格：大约4000美元/GB。

磁盘价格：大约100美元/GB。

花同样多的钱，我们可以购买40倍数量的磁盘。考虑到一家小公司不太可能使用超过1GB的数据库（可能需要更多的存储器作为备份或者其他用处，而磁盘可以完成这一任务）。如果性能非常重要的话，4000美元的价格不太可能会阻止用户购买内存。另外一方面，如果性能并不是很重要的话，4000美元的开销似乎有些浪费。使用小数据库的小公司可能每天可

能不需要做多少事务操作，他们认为性能是第二位的。需要高性能的大公司可能需要超过10GB的数据库，他们就需要在内存中使用缓冲区。但是计算机设计者需要扩展内存地址的大小，现在内存地址是32位，在大多数计算机上大概能访问4GB的内存地址。一些新的并行计算机已经采用了64位的体系结构，IBM也提供了不同32位地址空间切换的方法好几年了。这一切都表明不久的将来，我们可能可以在花费15美分的汽车中花费几分钟就到达火星了。

2. ORACLE中的DBA和磁盘资源分配

在第7章中，我们没有考虑表的磁盘空间的分配，但其实这是DBA最重要的任务之一。不幸的是，这个问题和特定的数据库产品有关，因为不同的商业数据库系统的体系结构有很大的不同，所以使得任何标准的SQL方法都失效了。但是，大多数数据库产品对于处理数据分配有相同的一般方法，所不同的是，它们在细节问题的处理上有很大的不同。下面我们将说明在ORACLE中怎么完成磁盘资源分配。我们不会再详细地描述其他数据库产品，读者如果想详细了解的话，可以参考本章结尾的“推荐读物”。

在创建数据库前，DBA首先需要在磁盘上分配一些操作系统文件，假设这些文件名是fname1,fname2,…。这些文件和用户编辑的文本文件或者C语言源文件是同一种类型的文件。许多操作系统允许DBA指定文件的大小和文件所在的磁盘设备。如果扇区是紧挨着的——扇区在同一柱面上是连续分布的——我们称磁盘存储器是连续的。让磁盘空间连续分布可以使寻道时间最小，大多数系统在分配文件空间的时候会使文件分配在一大块连续的空间内。同时，大多数操作系统文件没有跨越磁盘设备的灵活性。对于这些操作系统文件，DBA可能使用如下的ORACLE命令：

```
create tablespace tspace1 datafile 'fname1', 'fname2';
```

表空间是ORACLE数据库基本的分配介质，所有请求磁盘空间的表、索引和其他对象都在表空间中收到分配给它们的空间。表空间对应于一个或多个操作系统文件，也可以跨越多个磁盘设备。在大多数操作系统中，ORACLE可以创建或扩展操作系统文件，虽然管理员可能会损失一些精确度。另外，表空间也可以创建在“原始磁盘分区”上，即该磁盘设备不是操作系统文件系统的一部分，而ORACLE可以不通过文件系统服务来使用它。（虽然在本书中我们不会讨论使用原始磁盘分区，但是读者应该知道，如果在性能是第一位的情况下这种原始磁盘分区是很重要的。）所有的数据库产品都有类似于表空间的结构来隔离用户和操作系统：DB2 UDB称之为表空间，INFORMIX称之为数据库空间。在各种情况下，它代表了一块可以使用的磁盘空间，这块磁盘空间可能会跨越多个磁盘设备。

下面是一个Create Tablespace语句的例子，它使ORACLE创建两个文件fname3,fname4：

```
create tablespace tspace2 datafile 'fname3' size 200M,
    datafile 'fname4' size 300M;
```

关键字SIZE后的整数代表了字节数。如果整数后面跟着K，则代表千字节数；如果整数后面跟着是M，则代表兆字节数。值得注意的是1KB实际是 $2^{10} = 1024$ 字节，1MB实际是 $2^{20} = 1\,048\,576$ 字节。许多ORACLE数据库包含多个表空间，其中一个表空间的名字是SYSTEM，这个表空间是在执行Create Database命令的时候由系统自动建立的。SYSTEM表空间包含了数据库的数据字典（我们称为系统目录表），也可以为用户定义的表和索引以及其他数据库对象提供表空间。由DBA决定是否可以创建多个命名的表空间。在大型系统上创建多个表空间有两个优点：(1)可以让不同的磁盘设备作不同的用处，(2)如果卸下某个磁盘，不会影响到整个

数据库。

当DBA或者具有CREATE TABLE权限的用户创建表或者索引的时候，可以使用Create Table或Create Index语句的可选子句TABLESPACE。这个子句可以让用户指定表空间的名字和表空间所在的磁盘（参见图8-5中Create Table语句的语法）。如果没有创建任何命名的表空间，表将创建在用户缺省的表空间内，该表空间将在用户第一次被授予表空间资源的时候设置。创建表的时候，它的表空间分配是以叫做数据段的对象标识的；创建索引的时候，它是以叫做索引段的对象标识的。还有一些其他类型的段，它们共同划分了一个表空间。参见图8-3。

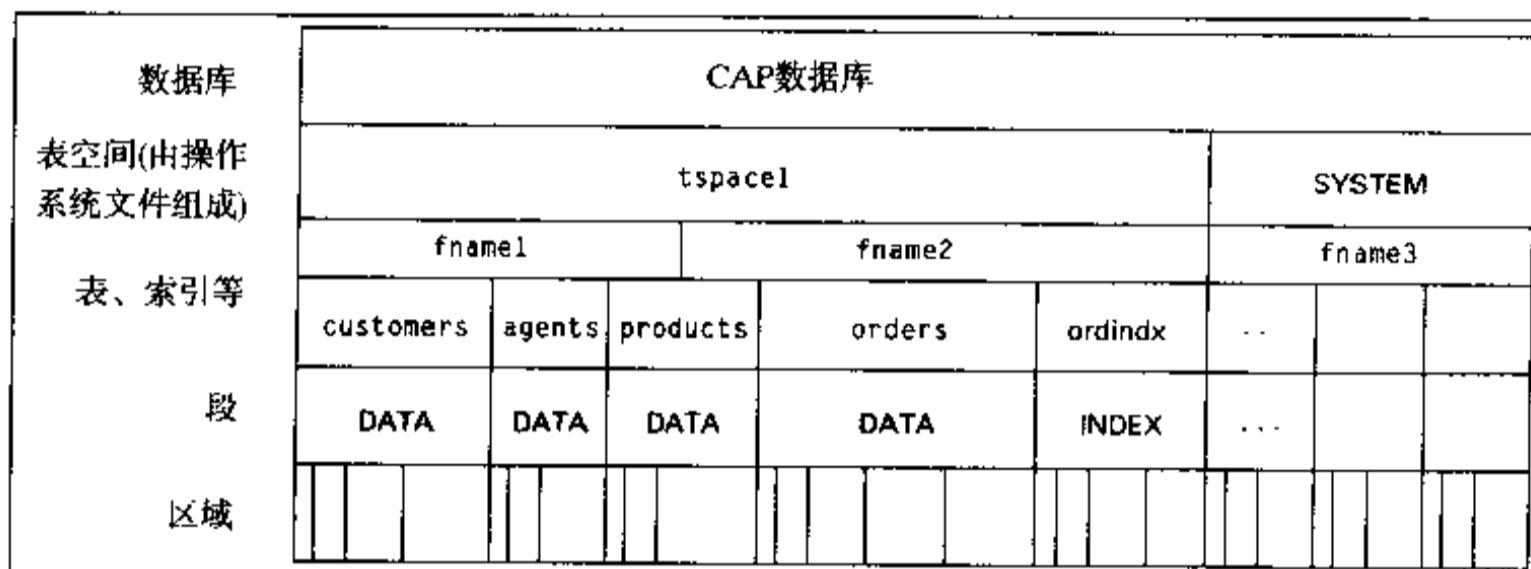


图8-3 数据库存储结构

当创建数据段或者索引段的时候，将从表空间中分配一个初始的磁盘空间，称为初始区域。初始区域的缺省大小是10KB。每一次当数据段的大小快超过指定空间大小的时候，都会再分配一块区域，被称做下一区域，以1开始的整数编号。图8-3是一张表明上述逻辑结构的示意图。值得注意的是，每一块区域都在一个数据文件上(在图8-3中很难表示这一点)。

一个区域必须构成表空间的单个文件中的连续磁盘空间组成，但是一个段可以是由来自多个文件的区域组成。可以在创建表空间的时候指定一些参数来处理新的区域的分配。该表空间所有的段都按这些参数设定。图8-4是Create Tablespace语句的部分语法。

```

CREATE TABLESPACE tb1spacename
  DATAFILE 'filename' [SIZE n [K|M]] [REUSE] [AUTOEXTEND OFF
    | AUTOEXTEND ON [NEXT n [K|M]] [MAXSIZE {UNLIMITED |n [K|M]}]
    |, 'filename' (repeat SIZE, REUSE, and AUTOEXTEND options) . . .]
  -- the following optional clauses can come in any order
  [ONLINE | OFFLINE]
  [DEFAULT STORAGE ([INITIAL n [K|M]] [NEXT n [K|M]] [MINEXTENTS n] [MAXEXTENTS {n|UNLIMITED}]
    [PCTINCREASE n]) (additional DEFAULT STORAGE options not covered)]
  [MINIMUM EXTENT n [K|M]]
  [other optional clauses not covered];

```

图8-4 ORACLE的Create Tablespace语法

我们已经说过：表空间是由多个在DATAFILE子句中指定的操作系统文件组成。如果没有使用关键字SIZE，数据文件必须已经存在，而且ORACLE以它的名字使用该文件。如果使用了关键字SIZE，ORACLE会按指定的文件名创建一个新文件，如果该名字的文件已经存在，则删除以前所有使用该名字的文件。只有当关键字SIZE出现的时候，REUSE才有意义。

REUSE关键字允许ORACLE重新使用一个已经存在的文件(ORACLE会首先检查文件大小是否正确),这个文件原来的内容将会被破坏。

AUTOEXTEND子句决定ORACLE是否可以自动扩展数据文件的大小。如果使用的是AUTOEXTEND ON,那么NEXT n[K|M]将指定每次新申请的区域的大小。值得注意的是,文件扩展的大小可能比需要的区域大小大很多。MAXSIZE子句决定了文件最大可以扩展的大小,可以是有限的字节数,也可以是UNLIMITED。读者应该注意,即使使用的是UNLIMITED参数,文件系统本身也会做一些限制。

缺省情况下,表空间是联机(ONLINE)创建的,意思是可以立即被数据库系统使用。如果表空间是脱机(OFFLINE)创建的,那么它不能被立即使用。脱机创建的表空间可以被DBA使用Alter Tablespace语句改成联机方式(Alter Tablespace语句在本书中没有讲述)。SYSTEM表空间始终是用Create Database语句联机创建的。

Create Tablespace语句的DEFAULT STORAGE子句可以让表空间创建者指定缺省参数来处理在这个表空间中定义的段中磁盘区域分配。下面是这些参数的含义(这些参数可以按任意顺序出现):

INITIAL n[K M]	n指定了初始区域的大小,缺省值是五个数据块中的字节数,通常是10KB。
NEXT n[K M]	n指定了下一区域的大小,第一个下一区域的缺省大小是5个数据块,最小值是1个数据块(2KB)。如果PCTINCREASE指定一个正值,后面的下一区域大小可以增加(但是不可以减少)。
MAXEXTENTS n	n指定了可以分配的区域的最大个数,其中包含了初始区域,当然也可以指定UNLIMITED参数。
MINEXTENTS n	n指定了创建段的时候初始分配的区域个数。因为区域必须是连续分配的,这个参数允许一个大的初始区域分配,即使可用磁盘空间是不连续的时候。缺省值是1。
PCTINCREASE n	每一次后续的下一区域比前一个区域大的百分比。如果n是零,则不增加。缺省值是50,意思是后续的下一区域比前一个区域大50%。

所有的区域大小都上入到一个数据块(ORACLE对磁盘页面的叫法)中字节数的整数倍。一个区域大小的最小值是2048个字节,最大的值是4095MB。MINIMUM EXTENT子句可以保证Create Table语句在不使用Create Tablespace缺省区域大小的时候,不会选择可能引起过多碎片的区域大小。MINIMUM EXTENT n[K|M]将保证每次分配的区域大小的最小值是n个字节。

在图7-3中,我们学习的是ORACLE的CREATE TABLE语句的简单形式。在图8-5中,我们提供了一个更复杂的CREATE TABLE语句语法。

Create Table语句的新的子句从可选的ORGANIZATION HEAP(这是缺省值)和ORGANIZATION INDEX开始。一个堆组织的文件是新的行存放在任何方便的位置上(通常是在连续区域的连续页面的连续位置上)。索引组织的文件是新的行存放在索引里,按照主键值排序。我们会在8.4节中讨论主索引的时候进行详细阐述,但是当前ORACLE的索引组织文件有一些限制,所以并不向我们期望的那样有用。

```

CREATE TABLE [schema.]tablename
  ([columnname datatype [DEFAULT {default_constant|NULL}]] [col_constr {col_constr. . .}]
   | table_constr) -- choice of either columnname-definition or table_constr
  [, {columnname (repeat DEFAULT clause and col_constr list) | table_constr} . . .])
  [ORGANIZATION HEAP | ORGANIZATION INDEX (this has clauses not covered)]
  [TABLESPACE tblspacename]
  [STORAGE [[INITIAL n [K|M]] [NEXT n [K|M]] [MINEXTENTS n] [MAXEXTENTS {n|UNLIMITED}]
             [PCTINCREASE n]] (additional STORAGE options not covered)]
   [PCTFREE n] [PCTUSED n]
   [disk storage and other clauses (not covered, or deferred)]
   [AS subquery]

```

图8-5 ORACLE中的Create Table的部分语法

Create Table语句中可选的STORAGE子句可以让表的创建者不顾表空间中关于区域分配的参数设定。PCTFREE n子句决定了每个磁盘页面上可以有多少空间用于行的插入（参见图8-6中磁盘页面分布的示意图）n的变化范围是0到99，缺省值是10。n的值为10表明如果当页面上有90%空间已经被使用了，那么新的插入操作将被停止。如果n的值为0意思是除非页面上没有可用空间，否则新的插入操作将一直进行。n如果取比较大的值，就可以在页面上多留一些可用空间，这些可用空间可以被varchar类型的数据使用，或者在Alter Table语句中用于插入新的列。PCTUSED n子句指定了一个条件，当已用空间降到某个百分比下的时候，新的行可以继续插入。这个值必须是从1到99，缺省值是40。PCTFREE和PCTUSED的和必须小于100，并且根据最后的百分比值共同决定了磁盘页面空间大小稳定的范围。

举个例子，缺省设置的意思是当页面空间使用率为90%的时候，页面会被标记为满，将不能插入新的记录。只有当页面空间使用率降到40%的时候，才可以插入新的行。因此，页面是不时地插入一些新的行的。如果插入操作频繁，这些对单个页面的连续插入操作可以避免对单独页面的读写，因为所需的页面会一直存放在缓冲区中的。如果我们设定PCTFREE为30，PCTUSED为60，那么插入操作只能使页面保持60%到70%的使用率。如果我们设定PCTFREE = 20，PCTUSED = 90（两者的和是110，这种情况是不应该发生的），那么当页面的使用率为80%的时候，该页面就会被标记为满，但是它又会被认为可以立刻进行插入操作，因为页面使用率小于PCTUSED的值。这是一种异常情况。

3. 数据存储页面和行指针：ORACLE和DB2 UDB

一旦创建了表并分配了初始区域，我们就可以向表中装载或者插入行。在一张堆组织的表中，记录总是一个接一个地插在第一个区域的第一页面里，直到第一个页面的空间使用完毕，然后使用第二个页面。当初始区域空间使用完毕的时候，系统会分配一个新的区域给这张表。这一过程将一直继续，直到到了区域数目到达了最大的数目或者没有页面可以被分配了。每一个行是由包含了列值的连续的字节组成的；每一个行与页面开始位置的偏移量是已知的。在某些体系结构里，可能某些长的行的大小会超过单个页面的大小，这种情况我们会在后面介绍。

图8-6给出了一个典型的数据存储页面布局情况，在图8-6中该页面上有N个行。头信息部分（我们通常称为页面头）可能包含了显示页面地址（对页面的磁盘访问标识符）的字段、段的类型（索引或者表）等信息。每一行都是由连续的字节组成，行的开始位置都是从页面

中特定的偏移量开始的（要记得由于有varchar(n)类型，每行的长度是不一致的）。行目录里的实体给每个行编号，并存放每行在页面中的偏移量。在下面的部分里，我们通常称页面中的行目录号为槽号。在图8-6所示的体系结构里，假设新插入的记录是从右至左存放的，行目录是从左至右存放的，而中间的可用空间可以用于插入新的记录。这意味着如果在磁盘页面上删除一行并回收它所用的空间，剩下的行会向右移动，而行目录（通常不会删除）会向左移动，这样可用空间仍然在中间部分。ORACLE和DB2 UDB可以不同于这种移动操作，上面讲述的只是大多数系统的简单结构。在行目录区域里，可以存放其他类型的信息。例如，ORACLE允许不同表中的行出现在同一个磁盘页面上（在ORACLE中磁盘页面被称为数据块，或者简称为页面），在这种情况下，头信息中必须要有一个表目录来区别不同表中的行。

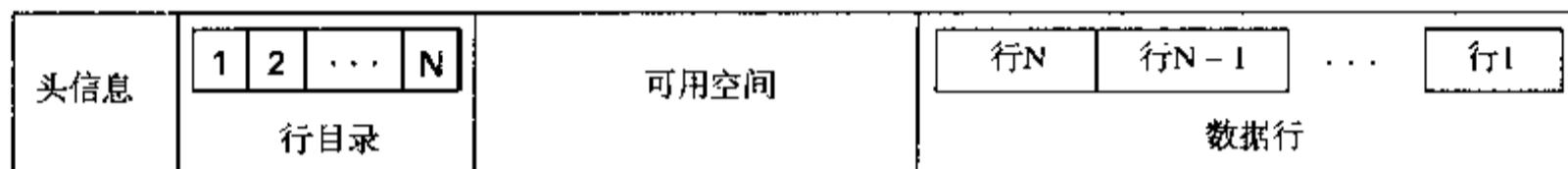


图8-6 磁盘页面中的行布局

数据库表中的行可以被行出现的磁盘页面地址和该页面中行的槽号唯一标识。这就是我们以前提到过的行指针，该行指针在索引结构中使用，用于按给定的索引键值来指向某个行。为了方便，我们使用页面中的逻辑槽号来指向行，这样就可以隐藏不必要的信息。因为如果我们使用行在页面中的偏移量，那么如果在某个页面重组织过程中（例如在某些varchar类型的列的宽度可以取更大的值，或者可用空间碎片的移动）记录发生了移动，那么索引中的行指针值就要修改。而如果使用逻辑槽号的话，即使页面中行位置发生变化，索引中指向行的指针值仍然保持不变，只要行目录实体仍然存在。

行指针在不同的系统中有不同的名字。在ORACLE中称为ROWID，在DB2 UDB中称为RID，通常我们统称为ROWID。读者需要知道的是，行指针的概念不存在于任何标准中，但是这里解释的概念是非常普遍的。为了给出RID的思想，我们再来看看本书第1版中提到的INGRES中的TID（元组ID）的概念。INGRES是最早的关系数据库产品，但是我们并没有说我们对于TID的定义已经过时了。在INGRES中，数据库表可以被分配连续的磁盘页面，这些页面都分配了一个磁盘页面号P，P的取值可以从0到 $2^{23} - 1$ ，最多可以有8 328 608个页面。这种结构允许在一个2KB的页面上最多有512个行，所以槽号的范围是0到511，可以表示为9位。在INGRES中行指针（TID）可以由磁盘页面号P和槽号S计算得出：

$$TID = 512 * P + S$$

例如，如果某个记录的S = 4, P = 2，那么它的TID的值为 $2 \times 512 + 4 = 1028$ 。某页面上的槽号不能超过511，所以不同页面上记录的TID的值是不同的。槽号是由9个比特表示，页面号由23个比特表示，所以TID的值是由32个比特即是一个4字节的无符号整数表示的。除了有一个指针指向行外，也可以把TID看成是一个“虚”列，然后使用一条Select语句得到TID的值，如下例所示。

例8.2.1 假设INGRES数据库的DBA刚刚装载了一个表名为employees的10 000行的表，而且每行的长度都长达200个字节，但是每行之间的长度差别不大。即一个大小为2048个字节的页面大约可以包含10行（头信息部分和行目录部分也需要占用磁盘空间，对于不同的列也需要一些磁盘空间，但是我们假设这些都已经包含在200字节的长度里了。为了免去具体的计

算，我们这里假设超过2000个字节的空间可以被数据所使用，这样的话，一个2048字节的页面上可以包含10个长度为200字节的行是大致正确的）。为了检查一个页面是否包含了10行，DBA可以利用下面的SQL语句来检查：

```
select tid from employees where tid <= 1024;
```

DBA希望得到的结果是0,1,2,3,4,5,6,7,8,9,512, 513, 514, 515, 516, 517,518, 519, 520, 521,1024。其他结果则说明每个页面上的记录数不为10。 ■

DB2 UDB的行指针RID也把表的页面号和槽号编成一个4字节的整数。但是DB2 UDB的RID不能看成是表的列而在Select语句中使用。另外DB2 UDB的RID的内部结构是不公开的，而且IBM警告说在今后任意时刻在没有任何警告的情况下可能会修改这一结构。

在ORACLE中，ROWID是一个6字节长的整数，有两种可能的形式：限制的ROWID和扩展的ROWID。限制的ROWID指定了行所在的块号（磁盘页面号）、记录号（我们称为槽号）和操作系统的文件号。ROWID是由三部分表示的，如下（每个字母都代表一个16进制数）：

BBBBBBBB.RRRR,FFFF

这里BBBBBBBB代表了文件中的块号，RRRR代表了行号，FFFF代表了文件号。例如，0000000F.0003.0002说明该行处在第2号文件的第15个块的第3个槽中。限制的ROWID占用了6个字节的空间，并且出现在索引项中。可以把ROWID看成是一个“虚”列，并可以在Select语句中使用，如下所示：

```
select cname, rowid from customers where city = 'Dallas';
```

可能得到的结果是（假设是限制的ROWID）：

CNAME	ROWID
Basics	00000EF3.0000.0001
Allied	00000EF3.0001.0001

而扩展的ROWID表示为一个由4部分组成的字符串，如下所示（每一个字母表示为一个基于64的编码）：

OOOOOOFFFBBBBBRRR

这里OOOOOO是数据对象号，代表了一个数据库的段（例如表）。FFF、BBBBB、RRR分别代表了文件号、块号和行号（槽号）。这里的编码是基于64的编码，除了有64个数字以外，与十六进制表示法兼容，该编码和可打印的字符对应关系如下：

DIGITS	CHARACTERS
0 to 25	A to Z (capital letters)
26 to 51	a to z (lowercase letters)
52 to 61	0 to 9 (decimal digits)
62 and 63	+ and / respectively

例如，AAAAAm5AABAAAAEtMAAB代表了对象号是AAAAm5= $38 \times 64 + 57$ ，文件号是AAB = 1，块号是AAEtM = $4 \times 642 + 45 \times 64 + 13$ ，槽号是1。而查询：

```
select cname, rowid from customers where city = 'Dallas';
```

可能返回的结果是：

CNAME	ROWID
Basics	AAAAm5AABAAAEtMAAB
Allied	AAAAm5AABAAAEtMAAC

因为一个索引是和一张表关联的，所以索引项中的ROWID的值没有必要包括唯一表段的对象号。但是，Select语句通常显示扩展形式。我们以后将使用ROWID来统称任一数据库中的行指针。

例8.2.2 考虑下面的嵌入式SQL语句，它的作用是从表中读出一行，然后调用一个复杂的决策过程，完成把taxcode列置为1的行更新。

```
exec sql select * into :emprec
  from employee where eid = :empidval;      /* unique row */
decisionproc(&emprec, &yesno);                /* call decision procedure function */
if (yesno)                                     /* if flag was set to yes */
  exec sql update employee set taxcode = 1    /* perform update */
    where eid = :empidval;
exec sql commit;
```

值得注意的是，为了更新所需的employee的列，将taxcode设置为1的Update语句需要完成第二次查找满足条件的行，可能是通过一个唯一的empid索引。我们可以通过记下第一次Select语句中的ROWID的值来避免这次查找，并改善性能。上面的Select语句可以修改如下(e.ROWID代表的是ORACLE中的虚列)：

```
exec sql select e.*, e.ROWID into :emprec, :emprid
  from employee e where eid = :empidval;
```

变量emprid声明为一个18字节长的字符串(一个19字节长的字符数组，包括了空结束符)。这样，上面的Update语句可以修改如下：

```
exec sql update employee set taxcode = 1
  where ROWID = :emprid;                      /* perform update */
```



在2.3节中的关系模式的规则2中，我们规定记录只能按内容来检索。在SQL语句中使用ROWID的值来检索很明显违反了这条规则，但是它被很多数据库产品支持，甚至是在对象关系语法中填加了REF能力之前。从表中取得ROWID的值是很有用的，其中至少有两个原因：

- ROWID的值可以用于查看表中的行是怎么存储在磁盘上的。
- 通过ROWID来访问某个行是最快的方法。

然而，ROWID并不能代替的主键，因为如果是很久以前存储下来的ROWID的值，用它来访问数据可能就是不合法的。例如，许多类型的数据库重组织可能会引起行移动到另外一个页面，因此使得原来的ROWID的值不能再被使用。这时，原来的ROWID的指针可能会指向一个并不存在的行，或者指向了占用了原来页面地址的新行。由此引起的错误可能是毁灭性的。为了安全起见，程序员应该在一个事务内部使用ROWID的值。在一个事务内部可以合法地使用ROWID的值，因为系统会对行加锁，因此行不会被删除或者进行重组织。值得注意的是，这种情况也可能影响对象关系中的REF值：在表进行重组织后，REF的值可能也会变成非法的。

另外一个ROWID不能代替表的主键的原因是，不同系统之间ROWID的格式是不同的，而且不同系统对ROWID的支持也是不同的。

行是否可以跨越多个页面与系统有关

不同的数据库产品对于行是否可以跨越多个页面有不同的处理规则。在DB2 UDB中，行的最大长度不能超过一个页面的大小。（对于一个4KB的磁盘页面，最大的行长度是4005个字节，两者之间的差用于一些必须的磁盘开销。）如果一个行不能在某个页面中存放得下，那么DB2 UDB会把行移到一个新的页面，然后在原来的页面中插入一条溢出记录（RID指针）。然而，ORACLE却允许行跨越多个页面。如果某个页面上的行变长而无法在该页面中存放时，那么该行将被分割成片段。行的ROWID值保持不变，在原来页面的原有槽中仍然有一个行片段，但是该片段的末尾指向了后续片段。后续片段和行一样，在一个新的页面上存放在某个槽中，也具有一个ROWID值。但是，延续片段的ROWID的值是不能被任何外部方法访问的，它只是一个用于访问跨越多个页面行的内部属性。如果发生了多次分割，那么原始的槽中将被写成指向新的位置，而不是存在一个指向的链表。

很明显，如果可能的话，我们会尽量避免分段的行的出现。分段的行就好像给伏尔泰写信的时候不列出所有的问题，而是说当你收到回信的时候，会把后面的问题在接下来的信中提出。在ORACLE中，可以使用Create Table语句中的PCTFREE子句在页面上空出尽量多的空间来处理行的增大，从而使行片段的出现概率减小。但是，由于ORACLE允许行跨越多个页面，因此在页面上多留可用空间的方法并不是一个一般的解决方法。而在DB2 UDB中，当行太大时，它会被整个移动到一个新的页面，由于RID指针必须保留在原始的页面中，所以仍然会引起片段。（DB2 UDB这样做的原因是，避免修改所有指向原始行位址的索引项指针）。减少片段是需要进行数据库重组织的原因之一。

8.3 B树索引结构

B树是一种键控索引结构，它和其他内存中的索引结构类似，例如平衡二叉树、AVL树、2-3树。它们的区别在于B树是存储在磁盘上的，只有需要访问时，某些项才会被调入内存中。另外一些基于磁盘的索引结构，例如散列表（HASH），也能在特定的应用程序中提供高效的访问速度，我们在本章中讨论其中的一部分。但是B树是当今数据库中使用最广泛的索引结构^Θ。它是DB2 UDB中唯一的索引结构；在ORACLE中，直到版本7，也是系统提供的唯一的索引结构。在ORACLE 7中，新增加了一种称为散列聚簇(Hash Cluster)的索引结构。DB2 UDB认为B树提供了不同类型索引访问的灵活性，在DB2 UDB中许多特性（例如顺序预取I/O操作，将在下一章中阐述）使得DB2对许多应用都有很好的性能。

下面的章节给出了关于了B树的索引结构的进一步介绍。在定义8.3.1中，列出了B树的重要特性。图8-7列出了ORACLE中Create Index的语法，该索引是采取B树实现的。在后面的章节将介绍ORACLE提供的散列访问以及Create Cluster语句。

后面，我们还会讨论关键字BITMAP；在下面的讨论中，我们假设BITMAP关键字不存在。在图8-1中的X/Open语法中，第二行中的圆括号中的列名列表指定了组成特定表上的索引键的

^Θ 确切地说，我们本文所说的B树应该是B+树，它是B树的一个变形。所有的商业产品都采用B+树，但是，通常这种索引结构被简称为B树。

列值的串接。关键字ASC | DESC实际上没有什么效果，因为在索引中的两个方向上很容易安排。执行ORACLE的Create Index语句的时候，会在命名的表空间中先创建一片初始区域。然后对应于表中的每一行都会插入一条(keyval, ROWID)形式的索引项。这些索引项将按键值排序存放在磁盘上。

```
(keyval1, ROWID1), (keyval2, ROWID2), . . . , (keyvalN, ROWIDN),
```

```
CREATE [UNIQUE | BITMAP] INDEX [schema.]indexname ON [schema.]tablename
  (columnname [ASC | DESC] [, columnname [ASC | DESC]])
  [TABLESPACE tblespacename]
  [STORAGE ([INITIAL n [K|M]] [NEXT n [K|M]] [MINEXTENTS n] [MAXEXTENTS n]
            [PCTINCREASE n] )]
  [PCTFREE n]
  [other disk storage and transactional clauses not covered or deferred]
  [NOSORT]
```

图8-7 ORACLE 的 Create Index语句的语法

所以在排序后keyval1 ≤ keyval2 ≤ ⋯ ≤ keyvalN。在第9章中，我们将讨论磁盘上索引项的排序算法；在排序阶段，需要大量磁盘空间、内存空间并且可以要求CPU和磁盘臂的使用。TABLESPACE和STORAGE子句以同对表一样的方式决定了索引的磁盘分配情况。在了解了B树索引节点分布情况后，我们再讨论PCTFREE选项。图8-7中的NOSORT选项表明表中的行已经按索引的键值排序并按此顺序存放在磁盘上了。因此可以通过连续访问表中行得到升序的索引项，所以排序步骤花费的时间就节省下来了。ORACLE将检查键值是否真的按升序排列，如果不是则将返回一个错误代码。

例8.3.1 内存数组的特殊二分查找法 在以前的数据结构课程中，读者可能已经学习了二分查找法。但是这里给出的特殊算法是针对已经排序的列表中含有多个重复值的情况的。在非唯一索引中，这一算法是很重要的。

在图8-8的程序中，假设我们有一个7行的表（可以推广为N）。有一个类似于索引的内存中的排序数组arr[7]，因此(keyvalK, ROWIDK)的值是通过arr[K-1].keyval和arr[K-1].ROWID给定的（记住下标的范围是0到N-1）。如果arr[]数组是通过keyval来排序的，即arr[0].keyval ≤ arr[1].keyval ≤ ⋯ ≤ arr[6].keyval，图8-8中的binsearch函数实现了查询最左边的keyval = x的下标，其中x可能有重复键值。

假设给出的keyval值序列是{1, 7, 7, 8, 9, 9, 10}，下标的范围是0到6。如果x=1，那么二分查找法将依次检测下标3、1、0，最后算法的返回值是0。如果x=7，那么算法将依次检测下标3、1、0，然后因为arr[0].keyval不等于7，所以我们检测arr[1].keyval，发现它等于7，所以算法最后的返回结果是1。如果x=8，那么算法将依次检测3、1、2，返回值是3。如果x=9，那么算法将依次检测3、5、4，并返回4。如果x=10，那么算法将依次检测3、5、6，返回值是6。如果在数组中有重复值，那么binsearch函数将返回使得x==arr[k].keyval的最小下标k。（本章结尾的习题中有一道题是要求读者证明这一点）。要使得binsearch函数具有普遍性，只需要：把7替换成N，而m的值要满足 $2^{m-1} \leq N < 2^m$ ，而probe的初始值为 $2^{m-1} - 1$ ，diff的初始值是 2^{m-2} ，这样检测 $\text{probe} \leq 6$ 或者 $\text{probe} + 1 \leq 6$ 就变成了检测 $\text{probe} \leq N - 1$ 和 $\text{probe} + 1 \leq N - 1$ 。

```

int binsearch(int x)           /* return K so that arr[K].keyval == x, or else -1 */
/* if no match: arr assumed to be external, dimension 7 is wired in      */
{
    int probe = 3;             /* start subscript K = 3          */
    diff = 2;                 /* difference to 2nd probe       */

    while (diff > 0) {         /* loop until K to return      */
        if (probe <= 6 && x > arr[probe].keyval) /* if probe too low            */
            probe = probe + diff;           /* raise the probe position   */
        else
            probe = probe - diff;           /* lower the probe position   */
        diff = diff/2;                  /* home in on answer, K       */
    }
    if (probe <= 6 && x == arr[probe].keyval) /* have we found x?          */
        return probe;                /* if so, return K             */
    else if (probe + 1 <= 6
        && x == arr[probe+1].keyval) /* might have undershot       */
        return probe + 1;           /* then return this K          */
    else return -1;               /* else, return failure        */
}

```

图8-8 包含7个元素的二分查找法

如果列表中元素的个数为N,那么该算法的循环次数是 $m - 1$,其中m是满足 $2^m > N$ 的最小值(每个对应一个diff值, diff的取值是从 2^{m-2} 到 2^0)。然后将检测最后的probe值,也许还要检测probe+1的值。另一种方法是循环次数是m或者m-1,其中m = CEIL(log₂(N))(CEIL(X)函数是取大于X的最小整数)。虽然我们可以证明它几乎是最有效的查找算法,但是对于基于磁盘的查找来说却不是最优的。可以证明,如果我们对磁盘上的已经排序的列表进行二分查找可能需要更多的I/O操作。

例8.3.2 一百万个索引项的二分查找法 假设表中有一百万行,因此在有序数组arr[]中有一百万项。假设ROWID需要占用4个字节,键值也要占用4个字节(这是对于整型键值而言,如果对于字符串索引键可能需要更多的空间)。因此数组中的每一项是8个字节,一个2KB的磁盘页面最多可以存放 $2048/8 = 256$ 个有序项(忽略额外开销)。上述的二分查找法首先检查 $2^{19}-1 = 524\ 287$ 那一项中的keyval的值,与x进行比较,接着根据比较的结果向左或者向右移动 $2^{18} = 262\ 144$ 项。每一次后续检测移动的距离是上一次的一半。图8-9显示了后续检测的模式,即每一次从前一次检测移动距离的情况。

检测次数	与前一次检测值的差	检测次数	与前一次检测值的差
1	无前一次检测	8	4096
2	262 144	9	2048
3	131 072	10	1024
4	65 536	11	512
5	32 768	12	256
6	16 384	13	128
7	8192

图8-9 1 000 000行的二分查找的probe的收敛距离

直到第13次，前一次的和当前的probe所指的项才有可能在同一个页面上。因为每一个页面上最多只有256项，所以前12次中，该算法都是不停地在不同页面之间来回切换。如果我们假设访问过的页面不会驻留在内存中，那么这就意味着我们至少要作12次I/O操作！ ■

我们可以使用B树来对二分法查找加以改进。

例8.3.3 一百万索引项的B树结构 如果我们假设每一页上有256个索引项，那么一共有 $\text{CEIL}(1\ 000\ 000/256) = 3907$ 个页面包含索引项。这些包含(keyval, ROWID)形式索引项的页面，是B树结构中的叶节点。现在给出了我们希望定位的带有任意键值x的项，让我们创建访问正确叶节点最有效的目录项。我们所需要的是指向每个叶节点的指针（一个页面号，我们称为节点指针np）和一个在这些指针之间的分隔符键值。图8-10给出了这种结构的一个例子。注意，叶节点的ROWID的值没有画出。

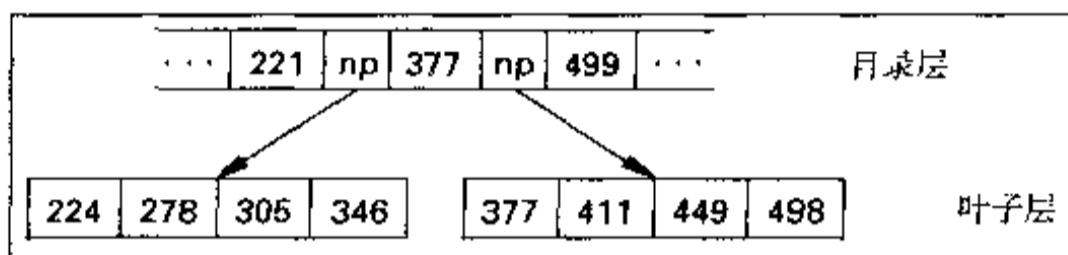


图8-10 指向叶子层节点的目录结构

在这样的目录层上查找一个指向包含键值x(例如305)的叶节点指针，我们只需要找到使得 $x \leq S_1$ 的最左边的目录分隔符键值 S_1 。在这个例子中，分隔符值377满足这个条件，所以我们可以按这个分隔符键值的左边的指针找到叶子层。在叶子层，我们可以使用一般的查找法来找到该键值的准确位置。因为在叶子层有3907个页面，因此上一层的目录节点只需要3907个节点指针和用来区分包含在所有叶节点中键值范围的3906个分隔符键值。我们假设在每一个目录项，类似于叶子层的索引项，包含了一个指针和一个分隔符键值(np,sepkeyval)，因为目录项所需要的空间和叶子项所需要的空间差不多，因此3907个索引项就大概需要 $\text{CEIL}(3907/256)=16$ 个页面，我们称其为B树的索引节点或者目录节点。下一步，我们需要创建高层目录层来引导我们到达我们建立在叶子层之上的目录的正确索引节点。我们可以使用同样的方法，有一个键值和一个指向子节点的指针，在这一层节点中我们只需要16个索引项。这个数目保证在一个页面中就可以装得下，我们称它为B树的根节点。图8-11表示了一个三层B树的结构。

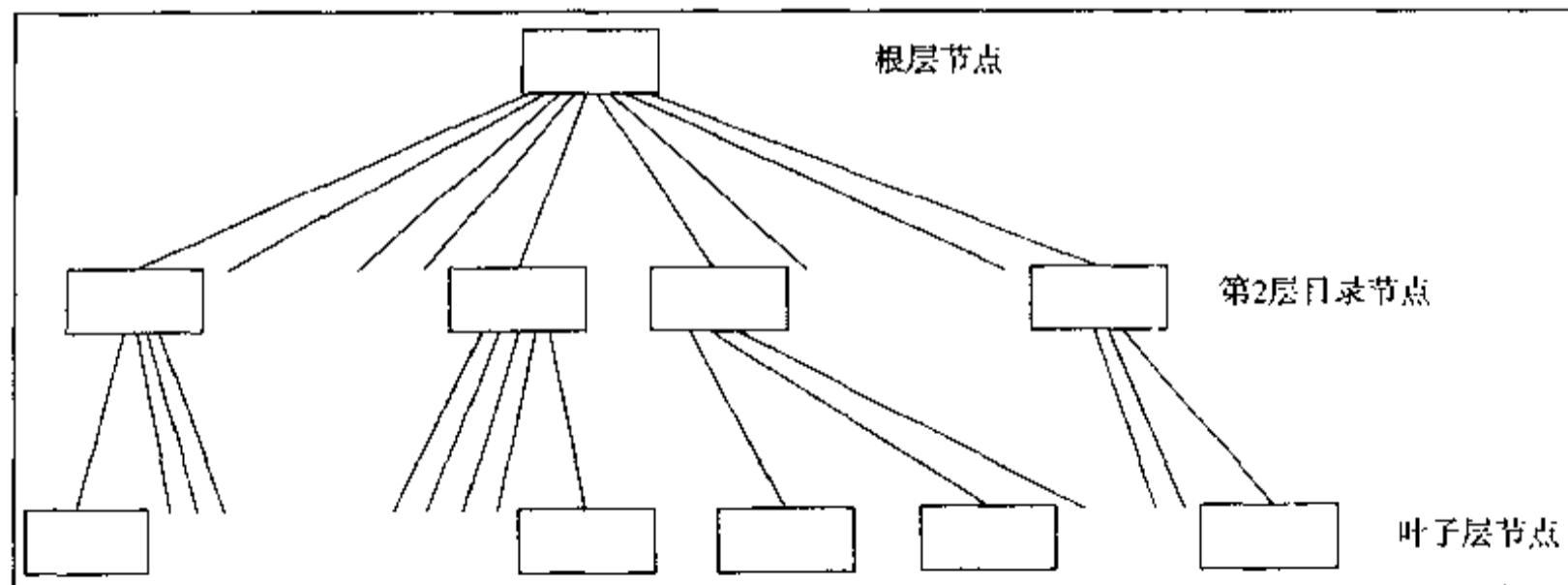


图8-11 三层B树

值得注意的是，B树是倒立的树，根在上面，叶子树在下面。我们把所有叶子层以上的节点，包括根节点，都统称为目录节点或者索引节点。根节点以下的目录节点有时候又称为树的内节点。根节点是B树的第一层，下面节点的层号依次增大，而叶节点的层号最大，在图8-11中，叶节点是第三层。叶节点的层号称为B树的深度。为了找到任一个叶节点，我们首先读入根节点，然后依次找到下面各层的目录页面，最后找到我们所需要的叶节点。然后在叶节点中查找各键值。在图8-11中，我们只需要进行三次I/O操作就可以访问到我们所需要的项，比上面需要13次I/O操作的一百万项的二分查找要少得多。

使用B树查找比二分查找要快得多的原因在于B树可以得到每个读出磁盘页面（B树节点）的大部分内容。一个目录节点页面上的所有项都包括了指向下一-层节点的指针，一共可以指向256个低一层节点。而在二分查找中，只能两选一，结果只有两个节点被访问。因此如果我们注意一下B树是稠密的，而二分法查找树是稀疏的。我们选择B树是因为一个三层的B树可以访问到 256^3 个叶子层索引项。B树有256个扇出，而二分查找仅有两个扇出。而树的叶节点的扇出数也较大，每一个叶节点都有256个ROWID值指向被索引的行。表中的行可以被看成是位于B树叶节点下面的一层。

如果我们假设某B树带有f个扇出，我们可以检测 $\text{CEIL}(\log_f(N))$ 次来访问叶子层的N项。因此如果扇出数为256，我们可以通过 $\text{CEIL}(\log_{256}(1\ 000\ 000))=3$ 次检测访问到1 000 000个叶子项。而对于二分查找，需要 $\text{CEIL}(\log_2(1\ 000\ 000))=20$ 次检测。当然达到某一点以后，二分查找不需要页面I/O操作。而且，在B树的情况下，对于每一个索引节点都需要多次检测才可以得到正确的分隔符键值来访问下一层。但是因为我们要最小化的是I/O操作的次数，所以在某一节点页面内部的多次检测是不计算在内的。实际上，对于一个经常使用的B树而言，上层的索引节点是经常被访问的，而且在所有的数据库产品中都有如图8-2所示的内存缓冲区，所以上层页面很有可能就驻留在内存缓冲区中！。在例8.3.3中，上两层一共才17个页面，和叶节点有3907个页面相比毕竟不多。把所有索引节点放在缓冲区中可以把I/O次数减少到一次。在二分查找中，我们通过保留31个驻留内存的页面($31=1+2+4+8+16$)来裁剪12次检测中的前5次检测。

1. B树的动态修改

上面我们讲述的是如何在叶子层通过多层的B树目录来访问索引项。我们还需要知道如何进行B树的插入操作，因为当插入新的索引项时，我们声明B树是一种有效的自修改结构。当一个新的项插入到磁盘上的有序列表中时，为了给新插入的索引项空一个位置需要把后面的索引项都依次向后移动，这样需要平均移动一半该列表所存储的页面。如果应用程序需要进行频繁的插入操作，那么这种方法对于一个大索引效率就太低了。在例8.3.3中，有3907个叶子页面，我们需要移动这些页面的一半——近1953个页面读，近1953个页面写——如果我们不覆盖所需的数据的话这种情况必须单独发生。如果每秒可以进行80次I/O操作（假设大多数页面不存在于内存缓冲区中），那么需要大约 $4000/80=50$ 秒钟（值得注意的是，必须进行4000次的I/O操作，速度为每秒80次I/O。结果的单位是“I/O次数”除以“I/O次数/秒”，所以最后的单位是秒。在后面的章节中，这种单位之间的计算是很有用的，将会被经常使用到这种运算）。

基于图8-12中的例子，我们将介绍一种方法，通过这种方法当在叶子层上插入新的项时能保证B树保持平衡，不需要更新过多的页面。B树的每一层上节点都不能是“满”的（在前面我们假设节点都是满的，每一个页面上有256项）。相反，任意一层上都留了一些空间，这

样当进行插入操作的时候就不需要申请新的空间了。当要插入新的项的时候，我们将根据目录结构到达要插入的项所在的叶子页面。当插入完毕之后，目录结构将引导我们到达新项键值的同一叶节点。新的项的插入操作总是发生在叶子层，但是某些情况下，叶节点已经太满以致于不能接受新项。这时，我们需要把该叶节点分裂到两子叶子页面（叶节点里的索引项是排了序的，分裂后的左边的叶节点包含的是较小的键值，右边的叶节点包含的是较大的键值）。这意味着上一层的索引节点必须修改以插入新的分隔符键值，另外还要插入一个指针以指向由于该分裂产生的新的页面（另外一个叶节点使用原有的页面）。有时候，插入新的分隔符键值和指向索引的下一更高一层的指针会引起索引节点空间不够，这时，我们需要把索引节点分裂，就象在叶子层分裂叶节点一样，并修改更高一层的节点。根节点层也可以插入新的节点，如果根节点也需要分裂的话，就创建一个更高层的节点，该节点的指针指向原来根节点分裂出来的那两个节点。

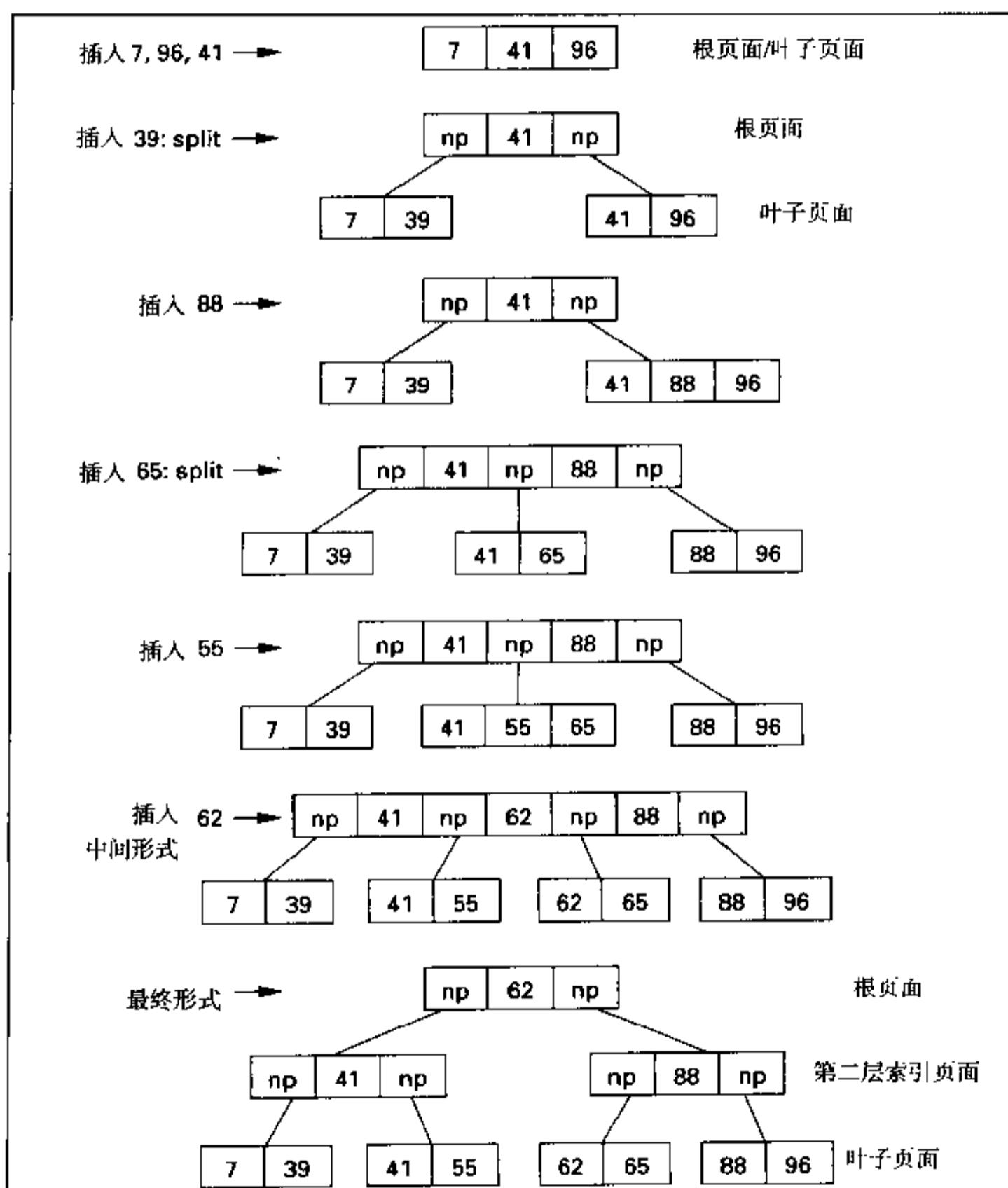


图8-12 B树的生长（叶子项的ROWID没有画出）

现在来看看图8-12。我们假设B树的叶子页面可以包含三个索引项，而不是四个（这比实际中的要少得多，但是我们只是为了说明方便）。我们也假设高层索引节点可以包含三个节点指针np，而不是四个（三个节点指针应该只有两个分隔符键值，但是这里我们假设节点指针个数和索引项个数相等）。这种结构与2-3树基本相同，是一种驻留内存的平衡树，在很多数据结构课程中都讲述过。

开始的时候，B树是空的。在前三次插入中，我们得到的是一个简单的结构：一个叶节点，同时也是一个根节点。也就是说，不需要高层的目录项，因为所有的项都在一个页面中。值得注意的是，我们没有把ROWID的值显示在叶子项中——假设每一个插到叶子层的键值都有一个相应的ROWID值。高层的节点项都有节点指针np指向低层的节点。这时，我们在B树中插入键值39，根节点就需要分裂，并创建一个新的根节点，该根节点有两个指针分别指向这两个低层的叶子层页面。然后，插入88和65，又产生一次分裂，并在根节点中插入一个新的分隔符键值。最后插入62，导致两次分裂。第一次是叶子节点的分裂，使得高一层的根节点中插入一个新的分隔符键值，从而导致该根节点也分裂，并创建一个新的根节点。

B树的深度只有当分裂根节点的时候才会增加。根节点分裂后，所有的叶节点的深度都会增加1。很明显不可能存在有两个叶节点的深度不相等的情况，所以B树仍然是平衡树。

B树中的项被删除来响应被索引的表中被删除的行或者发生修改列值的时候，B树是怎么样变化的呢？如果我们考虑图8-12中的最终形成，假设在叶子层中删除键值为62的项，很明显有一个叶节点中只有一个键值65。我们发现该节点与其右边的节点一共只有三个索引项，可以把这两个节点合并成一个带有三个项65、88和96的节点。合并之后，上层的目录层节点就不需要两个指针，而只需要一个指针就可以了。我们也不需要分隔符键值65，根节点处的分隔符键值55就可以了。现在考虑一下，我们就可以知道在目录层节点怎样进行合并（只有一个节点指针，而没有分隔符键值）。节点处的分隔符键值必须向下移动到合并的节点，以区别这两个节点指针。这就不需要根节点的存在，整个B树就减少了一层，而且和图8-12中插入键值62前的情况类似了（虽然不是完全一样）。

有一个算法可以执行上面描述的动作：保持B树是平衡的而且根以下的节点都至少是半满的。但是很少有商业数据库产品使用该算法。合并节点的思想有点复杂，在删除的时候需要额外的I/O操作。大多数的数据库系统都允许在不需要自动地重新平衡的前提下，减少节点的个数。如果树中的节点变空时，该节点就会被释放，以便一个右边生长左边删除的索引（例如一个日期/时间日历所使用的索引）在使用新节点的时候释放旧的节点（但是，从DB2 UDB版本5开始，当空余空间的比例有利于合并的时候，DB2 UDB就会支持B树节点的合并。我们将在讨论图8-15中的Create Index语句时进行详细讨论）。

不使用稀疏索引的主要原因不是因为磁盘空间（我们已经说过，磁盘是很便宜的），而是对分布在大量叶节点上的范围查询引起的磁盘I/O。为了解决这个效率的问题，在许多数据库产品中都提供了对B树进行重组的实用程序。这种实用程序能复制原始索引的建立过程，然后产生一个新而“干净”的B树。在下面的章节中，我们会介绍一些其他的磁盘存储需要考虑的事项，例如可用空间和压缩。

2. B树的属性

定义8.3.1描述了我们已经讨论的B树结构。这个定义假设键值可以是变长的，因为索引键可以是变长的列值。许多定义假设的是固定长度项，但是在实践中是不现实的。这个定义还

假设当分裂节点的时候，分裂的左右节点的长度是一样的，但是，某些产品允许不均匀的分裂来优化记录按键值升序插入的情形。最后，该定义假设当有项被删除的时候，会对B树进行重新平衡（通常不使用，这在前面已经解释过了），或者删除操作被插入操作压制，所以所有的B树节点中包含的项的数目都会增加。

定义8.3.1 B树(B⁺树)结构的属性 B树具有从根到叶多个扇出的树形结构。它具有下列属性：

- 1) 每一个节点都是和磁盘页面大小一致，并存放在磁盘上的某个合适位置。
- 2) 叶子层以上的节点包含目录项，有 $n - 1$ 个分隔符键值和 n 个指向下一层B树节点的磁盘指针。
- 3) 叶子层上的节点包含形式为(keyval, RID)的项，指向被索引的行。
- 4) 根节点以下的，所有节点都是至少半满的（进行多次删除之后，可能会不满足这一条件，除了DB2 UDB外）。
- 5) 根节点至少有两个项（除非只有一行录被索引并且根节点是包含一个(keyval, RID)对的叶节点）。 ■

3. 索引节点布局和可用空间

一个普通的B树节点页面有很简单的结构。图8-13表示的是一种具有唯一键值的叶子层节点的可能布局。具有重复索引键值的索引通常会对重复键值进行压缩，这样就能在一列内列出所有能在一个叶子页面中存得下的、具有同样键值的RID值（如图8-17所示）。值得注意的是，具有可变长度键值的索引也对应于可变长度的项，所以某些形式的“项目录”会在图8-13中的节点中存在，就和图8-6中的行目录槽一致。项目录也提供了在节点中进行二分查找的能力，即使键值是可变长度的。

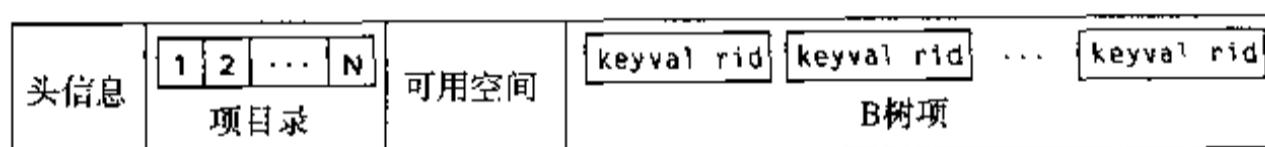


图8-13 具有唯一键值的B树叶节点的布局

因为在B树的生长过程中可能会发生分裂。所以随机插入后，B树节点可以在半满到满之间随机变化（根节点例外，它可能只有两个下层节点指针项）。正在生长的B树中的根层以下的节点的平均充满程度是70.7%，而不是我们直觉认为的75%。这个结果可以从数学分析中求得，而且和实验结果相符合。这个百分率对除根节点以外的所有节点成立，这个结果意味着如果对B树进行大量的随机插入操作后，先前对于100万行而估计出的叶子页面值过低。

例8.3.4 修改后的一百万个索引项的B树结构 和例8.3.3一样，我们再次假设一个索引项的大小是8个字节，一个节点页面的大小是2048个字节。但是，现在我们假设有48个字节的头信息，根层以下的节点大约是70%满的。每个节点上有1400个字节用于索引项，即有 $1400/8 = 175$ 项。在叶子层有1 000 000项，这就意味着需要 $\text{CEIL}(1\ 000\ 000/175)=5715$ 个叶节点页面，所以在更高一层有 $\text{CEIL}(5715/175)=33$ 个页面节点。因此在根层有33个目录项，该B树的深度也为3。在本章和下一章的习题中，我们使用的是相当粗略的计算。48个字节大小的头信息可能对于特定的产品是不对的，但是这只是一个粗略的计算而已。 ■

大家要记住的是B树索引的目的是最小化定位一个具有给定键值的行所需的磁盘I/O操作

的数目。B树的深度就近似于要访问到叶节点的I/O操作的次数。就像我们前面说过的一样，通常我们估计每层的扇出数为n，这里n是每个节点中出现的项数目。假设根节点和下面各层节点的目录项数目是n，那么第二层中项的数目是 n^2 ，第三层中项的数目是 n^3 ，依次类推。如果B树的深度是K，那么在根节点分裂之前，叶子层中项的总数为 n^K 。反过来，如果想创建一棵包含M行的B树，那么该B树的层数k是：

$$K = \text{CEIL}(\log_n(M))$$

因此，树的深度和行的数目是对数的关系。而树的深度K就是查找叶子层键值所需要的I/O操作的次数。但是，事实证明这两个说法容易令人误解。因为有内存缓冲区，那些经常使用的B树的上面几层节点都存放在缓冲区中（我们不关心那些不经常使用的B树的访问效率）。通常，深度为3的B树使所有叶子层以上的节点都在内存缓冲区中。但是很少有叶子层节点是存放在内存缓冲区中的，因此查找叶子层键值的实际的I/O操作的次数应该是1。同时，树的深度和行的数目是对数关系这一说法也容易令人误解，因为树的扇出很大，将不能凭直觉得到。在例8.3.4中假设，每个节点中项的平均数目是175，也就是n的值。忽略根节点处项的数目可能会大一些这一事实，这就意味着一棵有两层的B树要进行根节点分裂的时候，在叶子层包含了 $175^2=30\,625$ 项。如果一棵深度为3的B树直到有 $175^3=5\,359\,375$ 项时才会有根节点分裂。这就提供了足够索引500万行表的项，所以很少使用超过三层的B树。而175的值达到了937 890 625，所以表中行表的项数目要达到近十亿才需要五层的B树。（当然如果键值的长度很长，每一页中包含的项的数目会少一些，所以较少一些的行就可能要五层的B树）。

在大多数数据库产品中的表上创建一个索引时，最高效的方法是首先装载带有一些初始的行的表，然后再创建索引。这样做的优点在于Create Index首先取出索引项，然后按键值排序，最后把排序后的项加到B树的叶子层中的过程是非常高效的。B树中的所有节点是按从左到右装载的，所以后续的插入操作通常发生在同一个叶节点，这样就能保证在内存缓冲区中是连续分布的。当叶子节点分裂的时候，后续的叶节点是从下一个磁盘页面上分配的。每一层的节点在这种控制方式下分裂，而且允许我们在每一个页面上都留有正确的可用空间。另一方面，在Create Index语句后插入行的时候，通常会引起B树在叶子层随机插入，这样就需要更多的I/O操作次数（因为叶节点不存放在内存缓冲区中）和随机的节点分裂。因此先装载记录、然后在上面创建索引比先创建索引、然后在装载数据的效率要高。

4. ORACLE和DB2 UDB的Create Index语句

一旦创建后，B树很有可能保持不变。这种情况发生在表上的索引保证了没有新的行插入或者影响被索引的列的更新。如果不会对B树进行修改，我们可以让B树所有的节点都装满。但是，如果有可能会插入了新的项，让所有的节点都装满显然是一个愚蠢的想法。插入少量的新行就会引起大量的初始节点分裂（需要大量的I/O和CPU时间），结果是节点的个数迅速翻番，但是最后每个节点都几乎只有半满，从而导致资源的大量浪费。在大多数数据库系统中，我们可以控制B树开始创建的时候节点满的程度。图8-14重复了图8-7中表述的语法，在图8-14中通过ORACLE的Create Index语句中的PCTFREE子句说明了这一点。

PCTFREE子句中n的值可以从0到99，这个数字决定了当索引第一次创建的时候，每个B树节点页面（在ORACLE中称为块）将不被填充的百分比。这个空间用于当插入新行的时候那些新的索引项的插入。PCTFREE的缺省值是10，该值越大允许在节点分裂前能插入的行越多。在图8-15中，我们可以看到DB2 UDB中使用的一个类似参数。事实上，ORACLE一直

都保持和DB2 UDB语法的兼容性，因此我们可以认为磁盘存储结构没有什么区别的时候，命名约定是一致的。

```
CREATE [UNIQUE | BITMAP] INDEX [schema.]indexname ON [schema.]tablename
  (columnname [ASC | DESC] [, columnname [ASC | DESC]])
  [TABLESPACE tb]spacename]
  [STORAGE ([INITIAL n [K|M]] [NEXT n [K|M]] [MINEXTENTS n] [MAXEXTENTS n]
            [PCTINCREASE n] )
            [PCTFREE n]
  [other disk storage and transactional clauses not covered or deferred]
  [NOSORT]
```

图8-14 ORACLE的Create Index语句的语法（含有PCTFREE子句）

```
CREATE [UNIQUE] INDEX indexname ON tablename
  (columnname [ASC | DESC] [, columnname [ASC | DESC]])
  [INCLUDE (columnname [ASC | DESC] [, columnname [ASC | DESC]])]
  [CLUSTER]
  [PCTFREE n]
  [MINPCTUSED n]
  [ALLOW REVERSE SCANS];
```

图8-15 DB2 UDB的Create Index语句的语法

图8-15中的DB2 UDB的Create Index语句缺少了ORACLE中的TABLESPACE子句，因为在DB2 UDB中索引使用的表空间和表所使用的表空间是一样的。（DB2 UDB的Create Table语句在语法上和我们上面讨论的基本SQL形式一致）。INCLUDE子句可以和UNIQUE索引一起用来指定除了索引键以外的、索引中可以存储的列。这些列可以通过INDEXONLY查询来很快地访问到，这种查询不需要访问任何表的行。关键词CLUSTER将和聚簇索引一起说明。DB2 UDB中的PCTFREE子句和ORACLE中的含义是一样的。MINPCTUSED子句设置了一个阈值，当删除的行以后达到这个阈值，页面就会合并。ALLOW REVERSE SCANS子句（在DB2 UDB版本6中新引进的特征）创建了一个双向叶子指针，能让查询优化器在两个方向上都能进行扫描。当做指定范围检索的时候，DB2 UDB能非常有效地使用连续磁盘存储，所以尽可能让叶子节点在磁盘上连续存储是非常重要的。我们会在8.4节和第9章中详细阐述这一点。

5. 索引中的重复键值

我们已经讨论了很多关于B树的问题，我们并没有假设键值必须是唯一的。在例8.3.1中讨论二分查找法的时候，我们考虑了重复键值的可能性，并提供了在重复集合中找到最左键值的算法。在图8-12中，依次插入B树的键值是7,96,41,39,88,65,55,62，这些键值没有重复值，但是看看这张图最后B树形式，我们可以很简单地想象怎么样插入一个重复键值88。首先找到第一个满足 $88 \leq S_i$ ，然后按左边的np指针达到含有62和65的叶子节点，然后把88插到这个叶节点中。现在两个键值88的项的ROWID有不同的ROWID指针。键值和ROWID指针一起用于区别不同的索引项。

如果现在我们再插入键值88的另一行将发生什么情况呢？这行对应的新的项将插入到与前一个相同的叶子页面中，使得这个叶节点中有四项：62, 65和两个88。因为叶子层节点不能包含超过三个项，所以该节点分裂成两个：左边的那个包含62和65，右边的那个包含两个

88。这就意味着在上层节点中要新增加一个分隔符键值88，如图8-16所示。

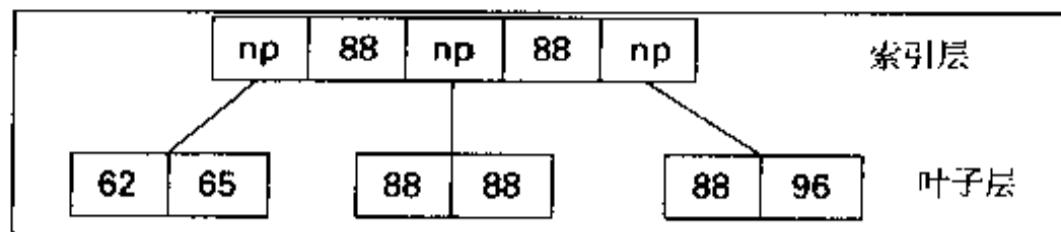


图8-16 有多个重复键值的B树结构

使用两个同样的分隔符键值88似乎有些奇怪，而且这个键值并不能区别下层最右边的两个叶节点（这两个节点中都包含键值88）。因此系统在处理查询的时候必须考虑到这一点，而实际上也的确考虑到了这一点。特别地，当要检索所有键值等于88的行的时候，系统在索引层查找分隔符键值大于等于88的最左边的那一项。当然我们找到了分隔符键值88，而且因为我们允许有重复键值，所以沿该键值左边的指针np向下的那个叶节点中，也有可能有键值88。在这种情况下没有（如果我们有唯一索引的话，我们可以查找分隔符键值大于88的最左边的那一项，如果该键值存在的话，总能找到正确的节点）。当我们到达了叶子层，我们读出所有含有键值88的项，从一个叶节点到下一个叶节点。这种检索重复键值的方法很容易推广为给定范围的检索（例如找出键值在88和120之间的行）。为了使得这种在叶子层的顺序查找能方便地进行，叶节点通常有指向它的后续兄弟节点的指针（也有可能存在指向前而兄弟节点的指针，用于那些降序查找的Select语句中）。

现在我们考虑一下如果当重复键值88的个数到达一定数目，从而需要在目录层以上包含两个分隔符键值的情况。只要我们的搜索算法保证能正确地在叶子层找到最左边的那一项就可以了，即总是沿最左边的目录指针向下查找。我们仍然可以使用从最左边开始、依次检索出具有相同键值后续项的方法。但是，当我们要删除具有键值88的行的时候，就出现了一个很有趣的情形。为了正确删除该行，它所对应的B树的叶层相关项也要删除，否则就会有一个索引项指向一个并不存在的行。因为该项没有被其键值唯一标识，所以系统可能要读出所有具有该键值的项（可能需要读多个页而）来检索具有将被删除的行的ROWID值并且键值等于88的项。可以使得同一键值的项按ROWID的值进行排序，并使得目录分隔符值中也包含ROWID的信息。因为分隔符中包含有唯一的ROWID前缀值，所以分隔符是唯一的，对于含有键值88和指定的ROWID值的行，可以通过B树很快地检索到。但是，这种方法并没有在某些商业数据库产品中实现。删除一个对应于多个重复键值的行，可能需要在B树的叶子层上进行一次较长的查找过程。

索引RID列表

对于重复键值，我们可以在叶节点使用合适的方法来节省空间。因为对于大量的行同一个键值可能重复多次，对于一个长的ROWID值列表我们可以一次只列出唯一一个键值。因为键值有时候是很长的字符串，而ROWID的长度却较短。这就可以在检索时节省大量的I/O时间。在DB2 UDB中，ROWID被称为RID，B树索引的叶节点有不同的形式，依赖于键值是唯一的还是有多个重复值。如果键值是唯一的，那么叶节点的布局和图8-13一致。如果有重复键值，那么叶节点和图8-17一致。

我们可以看到，如果有重复键值，DB2 UDB中键值只出现一次，后跟一个RID值的列表（在后面我将其称为RID列表或ROWID列表）。在DB2 UDB中，在不同的RID列表块的最前面有一个2个字节的前缀，在图8-17中表示为Prx。这个块前缀包含了逻辑块的长度，这样重

复键值的个数就可以确定了。每一个块中的RID的值最多可以有255个，但是可以创建一个具有相同键值的后续块。如果我们假设键值是10字节的字符串，每一个唯一键占用14个字节(keyval+RID的长度)，那么100个唯一键就需要占用1400个字节。另一方面，100个重复项需要占用 $2 + 10 + 100 \times 4 = 412$ 个字节(Prx的长度+keyval的长度+100个RID的长度)，从而节省了大量空间。很明显，当重复个数增加的时候，叶子层索引需要的空间主要就是RID值所需要的空间。

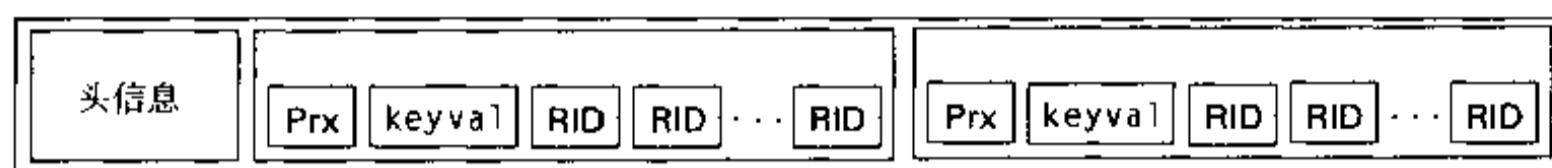


图8-17 在不同行有重复键值的DB2 UDB的叶节点的布局

6. ORACLE的位图索引

我们记得在图8-7中的ORACLE的Create Index索引有一个关键词BITMAP。如果有多个行有重复键值的时候，就可以使用位图索引(bitmap index)了。例如，我们假设定义了含有100 000个雇员的表：

```
create table emps (eid char(5) not null primary key, ename varchar(16),
    mgrid char(5) references emps, gender char(1), salarycat smallint,
    dept char(5));
```

这个表的主键列eid自动其有唯一索引。因为每一行都有一个唯一的eid键值，所以在该列上使用位图索引是不合适的。事实上，图8-7中的语法允许使用可选的UNIQUE或者BITMAP，但是两者不能同时使用。另一方面，列gender(值为'M'或者'F')、salarycat(值从1到10)和dept(有12个值，从'ACCNT'到'SWDEV')的值的个数都很少(这被称为这些列的集势较小)。因此，我们希望这些列的特定的键值有大量不同的ROWID值，这些列适合创建位图索引：

```
create bitmap index genderx on emps(gender);
create bitmap index salx on emps(salarycat);
create bitmap index deptx on emps(dept);
```

现在我们来介绍位图索引的思想。有N个行的表中每一行都有一个原始编号，依次是0, 1, 2, …, N - 1。连续编号的行，不论在什么情况下，在磁盘上都是物理连续的，而且一定有一个函数能进行原始编号到ROWID的转换。而位图是一个有N位的序列(每一位的值是0或者1，8位是一个字节)，位图能表示任何一个ROWID的列表L：如果原始编号为k的ROWID值在列表L中，那么N位位图的第k位被设置为1，否则就被设置为0。例如，位图

1001010001101101...

表示的是原始编号为0, 3, 5, 9, 10, 12, 13, 15, …的ROWID的列表L。每一个位图都代表了在B树中一个键值所对应的排序了的ROWID列表。因此，某一列的位图索引有该列上每个值的一个位图。例如表emps的dept列上的位图索引是一个普通的B树，该索引的每一个键值有一个相应的ROWID列表，只是这里我们使用的是位图(如图8-18所示)，而不是图8-17中使用的ROWID列表。值得注意的是，就像DB2 UDB把图8-17中的RID列表分割成逻辑块一样，ORACLE把位图分割成可以在磁盘页面上存储的段。很明显，一个有100 000行的表上的位图

将有100 000个位，也就是 $100\ 000/8=12\ 500$ 个字节，这就需要占用至少7个2KB大小的页面。

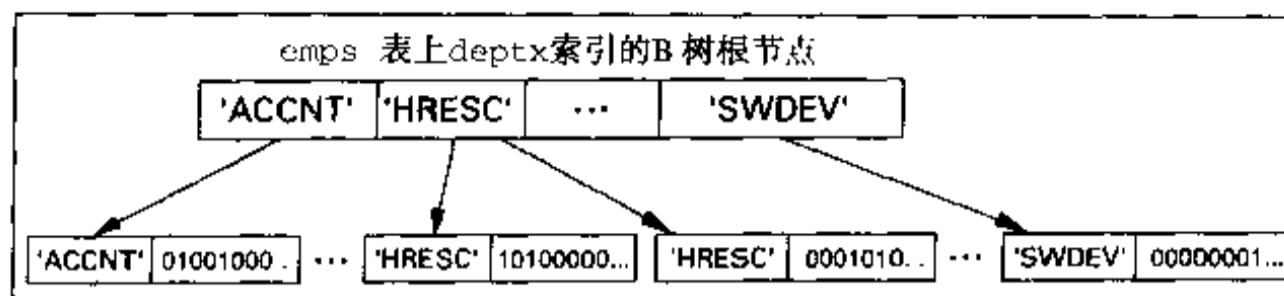


图8-18 emps 表的dept列上的位图索引

(1) 位图的密度

如果位图中1的比率较大，那我们就称该位图是稠密的。位图的密度定义为位图中为1的位的个数除以位图的所有位的个数N。如果某列上有32个值，那么每个值的平均密度就是1/32（这是原始编号为k的记录在该列上只有一个值，所以在位图索引的其中一个位图的第k个位置为1）。因为如果有32个键值，位图索引所需空间就和DB2 UDB的RID列表索引一样，对每个RID都需要32位，即4个字节（需要一些额外的空间用于存储键值，在位图索引中也需要这些空间）。但是，如果列上有320个值，那么对应的位图个数就是32个值所对应位图个数的10倍；每一个位图所需的大小和32个值时是一样的。这时，每一个位图的平均密度就降到1/320，那么位图索引的大小就和列值的个数成比例。但是对于任何数目键值的RID列表索引，它总是有相同个数的RID。如果RID的个数比键值的个数要多，它的大小仍然保持不变，所以键值的长度只是RID长度的一小部分。

如果位图的密度较小，那么我们称位图是稀疏的。当有大量键值的时候，由于这种稀疏索引位图索引所占磁盘空间的大小就比RID列表索引所占磁盘空间的大小要大得多。但是ORACLE用一个特别的算法来压缩位图索引中的稀疏位图，从而减小位图索引的大小。例如，通过类似于ROWID列表所提供原始编号列表的方法，稀疏列表可以被压缩。很明显在密度是1/320的位图中，这样，可以节省大量空间。ORACLE有这一算法更有效的压缩算法。为了说明压缩的有效性，我们举个例子：如果每一个键值都只有两行（一个相当稀疏的位图！），压缩后的位图索引比RID列表索引还要小。键值个数越小，压缩的位图索引的优点就越明显，直到位图的密度足够大了而不需要对位图索引进行压缩。在下面的讨论中，我们称未被压缩的位图为“逐字逐句的位图”(Verbatim Bitmap)，以区别于压缩了的位图。

(2) 位图索引的优缺点

和传统的RID列表索引相比，使用位图索引至少有两个性能上的优点。第一，位图索引有很好的I/O性能，因为压缩后的位图比ROWID列表索引需要更少的空间。第二，位图索引在执行表示在位图索引上的两个谓词之间的布尔操作（例如与和或）的时候有很好的性能。举个例子，假设我们要回答如下这一查询：

```
select eid from emps where salarycat = '10' and dept = 'ADMIN';
```

如果WHERE子句中的两个列属性都有一个位图索引，那么两个相等谓词将每一个都对应于位图索引中的一个位图。采用这种方式进行两个谓词之间的AND操作要比在传统的ROWID列表索引上进行要有效得多。而且，这个优点对于压缩位图索引一样存在，WHERE子句中出现的谓词越多就越有效。另外预先定义的特殊类型的位图索引在执行大多数普通类型的连接操作以及外键到主键的连接操作的时候很有效。在进行大量复杂查询的时候，位图索引经常被使用。

另一方面，如果对表进行大量的影响位图索引的更新操作的时候，位图索引就有缺点了。位图索引可以适应表上的更新、插入和删除操作，但是修改操作并不能很容易执行。特别是当位图索引被压缩以后，因为位图必须先解压缩，然后再进行修改。因为这个缺点，如果表上要进行大量的更新操作的时候，通常不使用位图索引。

8.4 聚簇索引和非聚簇索引

在图8-5后面的讨论中，我们看到通常连续插入到表中的行在磁盘上是连续存放的，使用的是堆结构（几乎所有的数据库系统都采用这种方法，ORACLE的缺省堆组织就是一个例子）。在图书馆中，这就像按照书籍购买的顺序把书本放在书架上——一种不常使用的方法，但是只要有许多目录卡片允许我们根据不同的分类法来查找所有的书籍，这还是挺方便的。但是更通常的做法是，根据杜威十进制编码（非小说类书籍）或作者名（小说类书籍）来安排书架上书籍的存放。（实际上，小说的存放顺序通常是根据作者的姓、作者的名、作者中间名和主题来存放的）这样的存放方法是很方便的。如果我们想要找到所有查尔斯·狄更斯的小说，我们可以在书目目录中查找任何一本狄更斯的小说，然后到达那个所有狄更斯的小说所在的书架，所以很快就可以找到所有他写的小说了，这样节省了好多步骤。类似地，如果我们想要找到所有关于造桥方面的书籍，可以根据杜威分类法（非小说类书籍主题分类的分类法）找到所有桥梁方面书籍所在的书架。

根据索引键值把书放在书架上或者把记录行放在磁盘上被称为聚簇(clustering)。所引用的行和键值顺序一样的索引称为聚簇索引 (clustered index，有时在DB2 UDB中称为 clustering index)。

主索引(primary index)是比聚簇索引更一般的概念，它指定了记录行在磁盘上的存放位置。在B树主索引（主索引的一个例子）中，行实际是存放在B树的叶节点上的，代替了我们前面讨论过的ROWID存放的索引项。包含指向行ROWID指针的索引现在被称为辅助索引(secondary index)，以区别于主索引。一张表中只能有一个主索引，而且很多数据库产品(例如DB2 UDB)根本就不支持主索引的概念，而另外一些数据库产品(例如ORACLE)提供的主索引也有很多的限制。值得注意的是B树主索引肯定是聚簇索引，因为行的顺序必须和索引中键值的顺序一致。但是，另外一种类型的主索引使用散列存储，在散列主索引中，行以索引键的散列值的顺序存放在磁盘上。因为散列值是随机的，因此连续的键值之间没有相关性。所以这种索引不是聚簇索引，具有连续键值的行在磁盘上的位置可能相隔很远。

数据库中的聚簇索引的优点在于：当所需要的行彼此间很靠近的时候，这些查询可以更有效地执行。因为查询需要的平均行只是一个页面中的很小一部分，所以如果行是聚簇在一起的，那么我们只要读出含有一个所需行的那个页面，其他所需的行就可能位于同一页面上。这样，访问其他行的时候，我们可能就不需要进行I/O操作。即使我们访问数据库的方法是很初级的并且根据行的ROWID值每次只访问一行，我们会发现第二行以及后续行已经在内存缓冲区中的一个页面上。内存缓冲区（参见图8-2）是为了节省系统处理I/O操作而设计的。另一方面，非聚簇索引就没有这一优点。非聚簇索引的后续行可能处于磁盘上的很远的地方，所以就不能节省处理I/O操作的时间。这就象是我们需要走遍整个书架来得到所有狄更斯的小说一样。图8-19说明了聚簇索引和非聚簇索引的比较。

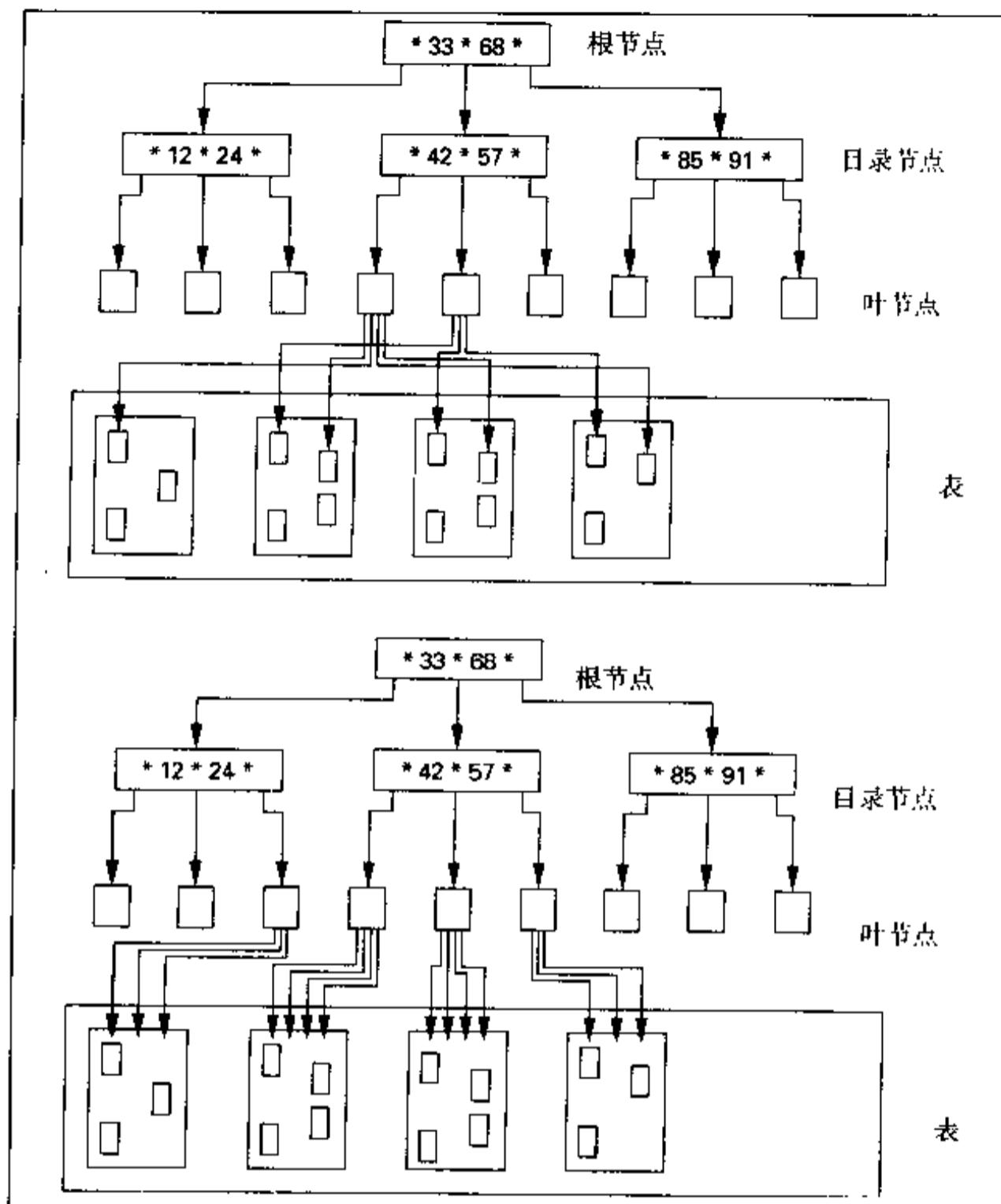


图8-19 非聚簇索引(上)和聚簇索引(下)的比较

例8.4.1 一个大的商店在美国一共有上百家的分店，并有1000万个顾客。该商店创建了一张customers表，其中的列名分别为straddr, cityst, zipcode, age, incomeclass（值从1到10，表明家庭收入的等级，从低到高），hobby(50个不同的值，从飞行学到动物学)，major1dept, major2dept（顾客消费最多的两个部门），还有一些其他的列属性。customers表的主要用处是找出应该给哪些顾客投递邮件。通常情况下，这种情况发生在某一个特定的地区，甚至是某个特定的分店（因为不同地区在购买什么样的商品方面有不同的要求，我们也没有足够多的商品可以在某个地区的所有商店里出售）。因此我们需要通过下面的Select语句给Boston地区的喜爱运动的顾客投递运动设备标签：

```
select name, address, city, state, zip from customers
where city = 'Boston' and age between 18 and 50 and hobby in ('racket sports',
'jogging', 'hiking', 'bodybuilding', 'outdoor sports');
```

当然，这些字段应该通过一个应用程序格式化为投递标签，而且查询需要通过游标在嵌入式SQL语句中执行，但是在这个即席的Select语句中I/O操作的性能的考虑是一样的。

现在考虑针对这个查询最好的建立索引的方法。首先，我们在 zipcode列上建立一个聚簇索引，这样Boston地区的所有行或者实际上任意一个地理区域的行都在磁盘上的一片连续空间里。通过统计信息，我们可以发现Boston地区的顾客占了整个美国的1000万顾客的1/50，即200 000。如果我们使用的数据库的页面是2KB大小的，每个顾客行需要100字节，那么每一个页面可以包含20行，一共需要10 000个页面。如果上面的查询要求选择1/5的Boston的顾客，那我们会发现40 000行仍然会占用上述10 000个页面中的大多数。但是，如果我们没有创建聚簇索引，那么Boston的顾客记录可能会存在于500 000个磁盘页面上（1000万行除以20行/页面），而且几乎每一个40 000条记录都会需要一次单独的I/O操作。因此聚簇索引使得需要访问的页面数从40 000减少到10 000。如果每秒执行80次I/O操作，那么这就是500秒和125秒的差别（虽然不同的磁盘臂可以同时执行多个磁盘访问动作，但是如果这些查询中只有一个为资源而竞争，那么该查询消耗的时间就接近于磁盘臂使用的时间）。

如果我们对WHERE子句再加一些限制，例如只寄给那些在体育商品部门进行大宗交易的顾客（major1dept = 'sports' 或者 major2dept = 'sports'）和家庭收入较高的顾客（incomeclass为9或10）那么我们只需要给2000个顾客邮寄。最多只需要访问2000个磁盘页面，假设我们可以同时估计出所有这些WHERE子句中的谓词。我们考虑的主要数据库系统都可以做到这一点，假设在所有其他属性上都建有辅助索引。如果我们没有这样的辅助索引，我们需要访问10 000个页面上的所有Boston的顾客，这样才能知道哪些行满足其他的谓词限制。

在例8.4.1中我们做了很多假设，但是关于这些假设我们要到下一章中才会详细讨论。现在，我们只是简单地认为这些假设在大多数的商业数据库中是成立的，而聚簇索引和非聚簇索引之间巨大的性能差别使得聚簇索引的价值更加明显。我们还需说明的是，我们并不需要有大量的重复键值来说明聚簇索引和非聚簇索引之间巨大的性能差异。使用BETWEEN谓词 BETWEEN keyval1 and keyval2 的查询也可以从聚簇索引中获得巨大的好处，即使这些键值是唯一的。不幸的是，一次我们只能通过一个索引聚簇一个表。很明显，我们不能把非小说的书籍同时按杜威十进制分类法和作者名存放在书架上，我们只能选择其中的一种方法。用于聚簇表的行的索引，主要依赖于数据库上经常使用的查询类型。DBA通常和应用程序管理员和程序员（他们能知道最经常使用的查询类型）进行商量来决定应该怎样使用聚簇索引。

1. DB2 UDB中的聚簇索引

当然，接下来的问题就是我们怎么样创建聚簇索引。不同的数据库产品创建聚簇索引的方法是不同的。在图8-20中，我们可以看到DB2 UDB的Create Index语句有一个CLUSTER子句。

```
CREATE [UNIQUE] INDEX indexname ON tablename
  (columnname [ASC | DESC] [, columnname [ASC | DESC]])
  [INCLUDE (columnname [ASC | DESC] [, columnname [ASC | DESC]])]
  [CLUSTER]
  [PCTFREE n]
  [MINPCTUSED n]
  [ALLOW REVERSE SCANS];
```

图8-20 具有CLUSTER子句的DB2 UDB的Create Index语句

DB2 UDB的CLUSTER子句没有任何参数，一张表只能有一个索引被标识为聚簇索引。当空表上的一个索引被标识为聚簇索引时，在装载新行（使用DB2 UDB的LOAD命令）前，用

户必须按CLUSTER键值对操作系统文件中的行排序。如果事先排序记录不方便的话，可以按任何顺序装载行并调用REORGANIZE命令。结果和图8-19中下面的图类似，B树中索引的顺序和页面上行的顺序是一致的。实际上，整个结构像一个扩充了一层的B树，按键值排序的行在新的叶子层，而原来的叶子层则作为高一层的目录层。这是我们期望的B树主索引结构。

不幸的是，这种类比有点不准确，因为当新行被插入的时候，这种结构的行为和我们希望的主索引是不一致的。DB2 UDB建议表的创建者在页面上留有一些可用空间（使用Create Table语句的PCTFREE子句），这样当插入新行的时候，DB2 UDB就可以把行插到正确页面的可用槽里，从而能保持聚簇顺序。但是，如果页面上没有可用槽了，那么B树就不会分裂该叶节点页面，如果可能的话则在当前区域内分配一个新页面，若当前区域(extent)满了，则在表的最后的另一个区域中分配一个新页面。我们插入的新行越长，聚簇性就越小，因为后面的行并不按聚簇顺序插入。最后，索引可能会丢失大多数的聚簇属性，用户被建议使用REORGANIZE操作，从而保证行按正确的顺序排列，并重新在每个页面上留有一定的可用空间。

我们将会看到，DB2 UDB使用聚簇索引检索将会减少大量的I/O操作。在例8.4.1中，Boston地区的记录可能需要从10 000个连续页面中取得，DB2 UDB实际上是读取多个页面（被称为预取I/O），这样就增加了每秒访问磁盘页面的个数。在8.2节中，我们说过I/O时间主要是消耗在磁盘臂移动到正确的位置，一旦磁头开始读取数据传输数据所消耗的时间很少。如果我们在磁盘臂定位在某一位置之后多读一些页面的话，我们就会提高I/O操作的速度，在某些情况下可以提高10倍，在其他DBMS产品中，这么高的多页面读写性能是不匹配的。我们会在第9章中谈到这个问题。

2. ORACLE中特殊的索引特征

ORACLE中有一些特殊的索引表的访问结构，包括索引组织表、索引聚簇和散列聚簇，下面我们就对此进行介绍。

(1) ORACLE的索引组织表

图8-5中的ORACLE的Create Table语句有一个ORGANIZATION INDEX子句：

```
CREATE TABLE [schema.]tablename
  ([columnname datatype . . .
  ORGANIZATION HEAP | ORGANIZATION INDEX (this has clauses not covered)])
```

如果出现了这一条子句，被建立的表中的行被放在真正的B树主索引中，按B树的叶子层的键值顺序排列。当插入新行的时候，它们会以正确的顺序插到B树的叶子层中，当叶节点没有可用空间时，该叶节点会分裂，而且分裂后的两个叶节点里行的顺序还是正确的。因此，ORACLE的索引组织表中的所有行是按键值聚簇的，当插入新行的时候，仍然保持聚簇。这似乎是一个理想状态，实际上的索引组织表上会有一些限制，这个限制就是ORGANIZATION INDEX结构中的B树键值必须是表的主键。

让我们再来看看例8.4.1。在这个例子中，我们建立了索引，能从指定的地区中按hobbies或者incomeclass来检索顾客。在列zipcode上创建了一个聚簇索引，在同一地区的行都处在连续的磁盘页面上。同时，列上的其他辅助索引，例如hobby和incomeclass，主要是限制从磁盘上检索出的行。

在ORACLE中，我们不能在列zipcode上定义索引组织表，因为zipcode不是该表的主键（它也不是后选键，因为同一个zipcode有多个行对应）。我们使用表customers的主键

custid，但是这样就不清楚怎么分配顾客号以保护地理位置。为了能提供按地理位置信息排序的唯一主键，我们需要使用至少两个列作为主键，例如zipcode和custid。这样我们可以在hobby和incomeclass上定义辅助索引。（索引组织表在ORACLE 8中已经可以使用，但是直到ORACLE 8i发布时，没有ROWID可以在这种结构中使用，因此不能定义辅助索引。但是在ORACLE 8i中，逻辑ROWID这一新特征可以为辅助索引提供支持。）

(2) ORACLE的表聚簇

ORACLE中的聚簇是包含了在某些特定列上进行连接的多张表中的行的结构，通常是以外键和主键的方式。这些进行连接的列通常定义为聚簇键，因此两张表中具有相同聚簇键值的行在磁盘上是存放在一起的。例如，表employees和表departments都有一个列deptno，这个列主要用于两张表的连接运算。deptno列在包含两张表的聚簇里作为聚簇键使用。两张表中具有同一个键值的所有行在磁盘上被存储在一起。例如所有表employees中deptno等于10的行和所有表departments中deptno等于10的行在磁盘上都存储在一起。Create Cluster的语法参见图8-21。

```

CREATE CLUSTER [schema.]clusternname
  (columnname datatype {, columnname datatype ...} -- this is the cluster key
   [cluster_clause { cluster_clause ...}]);

聚簇子句 指定了下列聚簇特性：
  [PCTFREE n] [PCTUSED n]
  [STORAGE ([INITIAL n [K|M]] [NEXT n [K|M]] [MINEXTENTS n] [MAXEXTENTS n]
            [PCTINCREASE n])]
  [SIZE n [K|M]]                                -- defaults to one disk block
  [TABLESPACE tblespacename]
  [INDEX | [SINGLE TABLE] HASHKEYS n [HASH is expr]]
  [other clauses not covered];

使用DROP CLUSTER语句来删除一个已有的聚簇：
DROP CLUSTER [schema.]clusternname [INCLUDING TABLES [CASCADE CONSTRAINTS]];

```

图8-21 ORACLE的Create Cluster语句的语法

图8-21中的Create Cluster语句中的许多子句都在Create Table语句中出现过，但是它们之间还是有一些差别的。首先，clusternname后的columnname datatype并没有定义包含在该聚簇中的表的所有列，而只是列出了聚簇键中的那些列。表的定义是定义了所有列和约束，并指定哪些列是和聚簇键中的列匹配的。图8-22说明了这个匹配是怎样指定的。和以前一样，如果使用了AS subquery子句，Create Table的列定义就不需要指定数据类型了。

```

CREATE TABLE [schema.]tablename
  (column definitions as Basic SQL, see Figure 7.1 or Figure 8.5)
  CLUSTER clustername (columnname {, columnname...}) -- table columns map to cluster key
  [AS subquery];

```

图8-22 包含有聚簇的Create Table语句

Create Cluster语句中的PCTFREE和PCTUSED关键词用于指定在聚簇磁盘块中插入新行的时候，磁盘块满的程度。图8-5的CREATE TABLE语句语法下面的讨论中，我们已经说明过了PCTFREE、PCTUSED和STORAGE的含义。图8-21中的SIZE参数表示的是为对应于一个聚簇

键值的行所保留的字节数。我们称一个聚簇键值所对应的行所占空间为聚簇槽，或者简称为槽。（这个术语是我们为了方便称呼而不是ORACLE中的标准）。因此SIZE参数指定一个槽的大小，ORACLE用这个参数来确定一个磁盘块上能存放的槽的最大个数：

一个磁盘块中槽的个数 = CEIL (一个磁盘块中可用字节数/SIZE)

当SIZE参数超过了-一个磁盘块中可用字节数的时候，每个磁盘块上槽的个数就为1。如果一个聚簇键值所对应的行所占用的空间超过了SIZE的估计值，就使用该磁盘块的其他槽中的空间，直到该磁盘块满了。如果该磁盘块满了（溢出），则分配一个新的磁盘块（在堆分配方式下，是从所在的表空间中分配的），而对应于一个键值的行会用链表从一个页面指向另外一个页面。但是这种溢出将会导致低效的磁盘访问，因此保证SIZE值足够大是非常重要的。另一方面，如果槽中的行的平均使用空间小于SIZE值，那么聚簇中所有磁盘块上都会有空间浪费的情况出现。如果某个磁盘块已经有了最大值，那么ORACLE将不会在该磁盘块上附加一个聚簇键值。

图8-21中的[INDEX | HASHKEYS n [HASH is expr]]子句表明了两类聚簇，索引聚簇和散列聚簇，缺省值是索引聚簇。不论是哪种聚簇，一个聚簇键值的所有行在磁盘上都存储在一个槽中。在索引聚簇中，聚簇上还要创建一个附加索引，这样就可以按聚簇索引键值访问行了（散列聚簇不需要这样的附加索引）。在索引聚簇上创建附加索引的语法和图8-14中的创建索引的语法基本上是一致的，只不过需要使用一个ON子句来指定聚簇名，而不是表名：

```
CREATE [UNIQUE] INDEX [schema.]indexname ON CLUSTER [schema.]clusternam
```

和我们在表上创建索引不同，索引聚簇上的索引对于访问聚簇表中的数据是非常重要的。虽然在任何时候索引都能被删除并重新创建，但是在这期间索引聚簇中的数据是不可被访问的。

在索引聚簇上创建了索引后，聚簇的表中可以插入行。值得注意的是，聚簇上定义的索引不是主索引。当聚簇中定义的表中的每一行以新的聚簇键值第一次存储在磁盘上的时候，会在该聚簇的一个磁盘块上为带有该聚簇键值的行分配一个新槽。这些槽是按堆组织的，先来先服务，直到槽的个数到了每个页面上允许的最大个数。因此，如果第一张表中要插入聚簇表的行是按聚簇键排序的，那么磁盘块中的槽和所有包含的行也是按聚簇顺序存放的。但是如果按新的聚簇键值插入的行，那么新的槽会按堆组织分配的原则，在表的最右边的磁盘块上分配。

在下一节中，我们会讨论散列聚簇。在散列聚簇中，当插入新的聚簇键值所对应的第一行的时候，一个散列函数将按键值计算聚簇槽的磁盘页面位置。按某个给定聚簇键值检索行的查询也将使用同一个散列函数来查找该行，这样就能节省B树查找的I/O次数。在本节结束前，我们举一些索引聚簇的例子。

例8.4.2 一个典型的索引聚簇的例子是对一家公司中的部门和职员进行聚簇。我们假设一个公共的查找任务是在给定一个部门中访问其中所有职员的信息。我们首先创建一个聚簇deptemp：

```
create cluster deptemp
  (deptno int)
  size 2000;
```

因为在Create Cluster语句中缺省的是索引聚簇并且我们没有使用HASHKEYS关键字，所以上面的聚簇是一个索引聚簇。现在我们创建聚簇中的表：

```
create table emps
( empno      int          primary key,
  ename       varchar(10)   not null,
  mgr         int          references emps,
  deptno     int          not null)
cluster deptemp (deptno);

create table depts
( deptno      int          primary key,
  dname       varchar(9),
  address     varchar(20))
cluster deptemp (deptno);
```

现在我们在聚簇上为聚簇键创建一个索引：

```
create index deptempx on cluster deptemp;
```

我们注意到SIZE参数是设成了2000个字节，这就意味着每一个页面上只能分配一个聚簇槽。如果我们假设一个磁盘块有2000个可用字节，depts的每一行占用40个字节，emps的每一行占用28个字节，因此在一个磁盘块上，我们可以存放带有公共deptno值的一个depts行和 $(2000 - 40) / 28 = 70$ 个emps行。如果我们假设有一个大的部门里有1000个职员，因此这个槽会占用很多连续的磁盘块。我们注意到每一个连续的页面上有 $2000 / 28 = 71$ emps行，因此包含所有的行需要 $(1000 - 70) / 71 = 14$ 个额外的磁盘块。虽然这看起来需要访问大量的磁盘页面，但是如果与depts表做连接的时候一下子访问所有emps行相比，这还是可以接受的。实际上，在这种情况下我们将记录存放在尽可能少的页面上。而且，如果我们希望通过其他属性访问emps行的话，例如empno或者ename，可以在emps表上为这些属性创建一个辅助索引。（注意，在这种情况下，empno是主键，已经有一个已建立的索引，所以只需要在ename上创建一个索引。）

```
create index enameix on emps (ename);
```

■

8.5 散列主索引

在图8-21中的Create Cluster语句中，我们看到cluster_clause子句指定了聚簇的特性：

```
[PCTFREE n] [PCTUSED n]
[STORAGE ([INITIAL n [K|M]] [NEXT n [K|M]] [MINEXTENTS n] [MAXEXTENTS n]
          [PCTINCREASE n])]
[SIZE n [K|M]] -- defaults to one disk block
[TABLESPACE tb1spacename]
[INDEX | [SINGLE TABLE] HASHKEYS n [HASH is expr]]
```

如果在Create Cluster语句中使用了关键词HASHKEYS，该聚簇就会成为一个散列聚簇，该散列聚簇包含了至少HASHKEYS个不同的槽。然后ORACLE会创建一个初始区域来包含大小为SIZE × HASHKEYS个字节的散列聚簇（需要注意的是，如果有STORAGE子句而且INITIAL的值大于SIZE × HASHKEYS，那么会使用较大一点儿的初始区域。使用初始区域来保证初始的磁盘分配尽可能地连续的是一个很好的方法）。在一个散列聚簇中可以包含多张表，

但是从ORACLE 8i开始，可选的SINGLE TABLE子句允许只使用一张表，并对它进行优化。散列聚簇中的表的所有行都存放在某个磁盘块的某个槽中的散列聚簇中。该槽的槽号是根据基于该行键值列的散列函数计算得出的。ORACLE提供了一个内置的散列函数，但是DBA可以通过使用HASH IS expr子句来提供一个用户自定义的散列函数，其中expr是计算数字值的表达式。通常用户不需要考虑使用自己创建的散列函数，除非在此之前已经经过了大量的分析。但是，为了完整性，我们提供一个使用用户自定义散列函数的散列聚簇的例子。

例8.5.1 我们创建了一个散列聚簇acctclust，只包含一张表accounts，在表accounts中有50 000行，这些行是检查银行客户的账号信息的记录（需要注意的是，散列聚簇和索引聚簇一样，可以包含多张表，用于进行高效率的连接操作。但是这里我们只使用一个具有唯一键值的单个表）

```
create cluster acctclust (acctid int)
  size 80
  single table
  hashkeys 100000 hash is mod(acctid, 100003);
```

值得注意的是，当HASHKEYS指定为n的时候，ORACLE会创建一个散列聚簇，它的实际的槽的个数S等于大于n的最小素数（ORACLE把实际的槽的个数S仍称为HASHKEYS）。在这种情况下，如果HASHKEYS最先指定的是100 000，那么实际创建的槽的个数是100 003；因此我们创建了一个散列函数mod(acctid,100003)，这个函数将基于acctid值计算出相应的槽值。实际创建的槽的个数（这里是100 003）可以通过创建一个带有HASHKEYS 100000的测试散列聚簇来确定，然后执行如下查询：

```
select hashkeys from user_clusters
  where cluster_name = 'ACCTCLUST';
```

现在我们创建表accounts，假设每行最大占用80个字节：

```
create table accounts
  (
    acctid      integer      primary key,
    balance     integer      not null,
    alname      varchar(12)   not null,
    afname      varchar(12)   not null,
    ami         char(1)       not null,
    apasswd     varchar(12)   not null,
    straddr     varchar(12),
    city        varchar(10),
    state       char(2),
    zipcode     int
  )
  cluster acctclust(acctid);
```

注意，不要在散列聚簇键值上建立索引。 ■

注意，在例8.5.1的表accounts中有一个主键acctid。这就说明在该列上已经创建了一个B树索引以避免有重复键值的出现。但是创建了散列聚簇后，就可以不用通过B树来访问accounts表中的行。在散列聚簇中包含该表，已经创建了该表的一个散列主索引。在表accounts中插入一行时，首先对acctid键值使用散列函数，然后产生一个伪随机号来决定槽号。从这个槽号，系统可以计算磁盘块号，然后可以通过一次I/O操作访问到那个磁盘块，

并把行存放到该槽中（在散列聚簇中，一个槽中有多个行）。在这种访问方式下，磁盘块上面没有键值目录，但是在辅助B树索引中就有。这里说的散列聚簇，并不需要在B树中进行多次I/O操作来查找指定acctid的行对应的ROWID值，然后再执行一次I/O操作来访问该行所在的磁盘块。在大型应用程序中，这种不需要目录查找直接到达正确页面的方式是很有价值的，例如在每秒钟需要进行几百次账号访问的大型银行中的出纳程序。

虽然散列的方法可以很有效地访问具有单个键值的行（银行出纳员通过acctid来访问客户的账户），但是在这种结构中无法按键值对行进行排序以提供给定范围内的有效检索。两个连续键值所对应的两行，可能存放在毫不相关的两个页面中，这依赖于伪随机散列函数。因此我们称散列访问结构形成了一个主索引（行在磁盘上的位置是由键值决定的，并提供了根据指定键值的有效检索），而不是聚簇索引（聚簇索引中行是按键值的顺序存放在磁盘上的）。结果，具有范围检索的SQL语句

```
select acctid, fname, ami, lname from accounts
  where acctid between :low and :high;
```

或者

```
select acctid, fname, ami, lname from accounts
  where acctid >= :low and acctid <= :high;
```

可能需要对表中的所有行进行扫描（在例8.5.1中，acctid上的唯一索引可以用于决定范围内的行）。

1. 在散列聚簇中调整HASHKEYS和SIZE的值

让我们回顾一下ORACLE是怎样决定散列聚簇中槽和数据页面的个数的。当执行例8.5.1中的Create Cluster语句的时候，会在一系列的页面上指定S个槽，用于存放表中的行。ORACLE会设置S为第一个大于等于HASHKEYS值的素数，我们设PRIMECEIL(HASHKEYS)函数的功能就是得到第一个大于等于HASHKEYS值的素数（在实际的ORACLE和其他数据库产品中并没有PRIMECEIL函数）。

$S = \text{槽的总数} = \text{PRIMECEIL}(\text{HASHKEYS})$

从图8-22后的讨论中，我们可以计算每一个磁盘块中的槽的个数B：

$B = \text{每一个磁盘块中的槽的个数}$

$= \text{MAX}(\text{FLOOR}(\text{每一个磁盘块上可用字节数}/\text{SIZE}), 1)$

不同的设备间，每一个磁盘块上可用字节数是不同的，其中在Sun Solaris 2.6上是1962个字节。在下面的例子和习题中，我们假设该值是2000个字节。给出了B和S的值，我们就可以计算最初分配给散列聚簇的磁盘块的点数D。

$D = \text{CEIL}(S/B)$

因此最初分配的磁盘块总数D等于槽的总数S除以每个磁盘块的槽的个数B，并且是上舍入的。散列聚簇上分配的槽的总数S是不会改变的，但是磁盘块的总数是会改变的。D是初始分配的磁盘块的个数，如果插入了足够多的行而产生溢出后，磁盘块的个数就可能会超过D。为了避免在磁盘上出现碎片，我们建议DBA先计算D的值，然后使用Create Cluster语句中的STORAGE子句来创建一个至少包含D个页面的初始区域（如果空间有可能发生溢出的话，需要 $1.1 \times D$ ，或者更多）。现在我们需要讨论的是散列溢出是怎样发生的。

散列聚簇中的每一行都存放在一个由散列函数h决定的槽中。键值对应的槽有一个槽号sn，

$0 \leq sn \leq S - 1$ 。具体一点，系统通过以下形式来决定给定的键值的槽号sn1：

```
sn1 = h(keyval1)
```

每个初始分配的磁盘块都包含固定数目的每个磁盘块中的槽的个数B，所以ORACLE可以通过下列公式计算出槽sn1对应的磁盘块号b和页面中的槽号s：

```
b = sn1/B
```

```
s = MOD(sn1, B)
```

这就是我们计算包含行的块号b的方法。给定b，访问散列聚簇中的一行只需要一次I/O操作，初始磁盘块发生了溢出时除外。

例8.5.2 在例8.5.1中，我们假设每一页面上有2000个字节，我们可以得到 $B = \text{FLOOR}(2000/80)=25$ 个槽/块， $D = \text{CEIL}(100003/25)=4001$ 个块。我们可以使用STORAGE INITIAL在磁盘上指定8002K或者更多的空间来确保初始的散列聚簇在磁盘上是连续的。该散列聚簇可能从第31244块开始，使用了第31245、31246等块，而在散列聚簇中的块号是 $b=0,1,2,\dots$ 。对于某一个acctid，例如acctid = 2345678，我们有 $h(2345678)=\text{MOD}(2345678,100003)=45609$ 。因此账号2345678将散列到槽45609。这个槽对应的相对块号 $b=45609/25=1824$ ，在初始散列聚簇的4001块当中，而实际块号则是 $31244 + 1824 = 33068$ 。这一行将会存放在该块中的第 $s=\text{MOD}(45609,25)=9$ 个槽中，在这个块的20个块当中。 ■

散列函数的思想是把每一个不同的键值散列到不同的槽号，但是因为只有S个不同的槽号sn，所以如果实际不同的键值数超过了S，那么上面这个要求是做不到的。实际上，即使不是这个原因，也有可能出现两个不同的键值散列到同一个槽号。假设我们在一个班上询问学生的生日（一个从1到365的伪随机数）。我们假设使用散列函数把学生散列到365个槽中，当学生的数据达到某个特定值（低于365，或者只有 $365/2$ ），就有可能出现两个学生有相同的生日的情况。这种现象被称为“生日问题”，我们会在习题中看到更多这样的例子。当两个不同的键值散列到同一个槽时，我们称为散列冲突。

在图8-23中，我们可以看到一个散列聚簇表有7个行稀疏分布在8个数据页的48个槽中。而每一个页面上有6个槽，一个页面在图8-23中表示为有6个槽的一行。（实际上，ORACLE中的槽的个数必须是素数，但是这里我们忽略这一点。）散列函数h()用于对键值进行散列操作。在图8-23中我们只显示了存储在页面上行的键值。在图8-23中，我们可以看到键值55和39散列到同一个槽中，从而引起了散列冲突。

h(55) = 槽 33						
0-5						
6-11	88			62		
12-17						
18-23		96				
24-29						
30-35	41		39	55		
36-41						
42-47	7					
冲突						
pg 1 pg 2 pg 3 pg 4 pg 5 pg 6 pg 7 pg 8						

图8-23 散列聚簇中的冲突，发生在插入键值55时

如果插入一个新的键值所对应的行的时候，发现它所散列到的槽已经被另外一个键值所使用，那么ORACLE会扩展该槽的空间以插入新行。值得注意的是，散列聚簇和索引聚簇一样，可能会有分配给它的多张表。一个非唯一的聚簇键值和一张表对应。因此，即使没有散列冲突，同一个槽中也可能存有多个行。散列到同一个槽的行在磁盘上可能是不连续的，它们是通过链接的方式连在一起的。ORACLE将试着为某一新插入的行扩展某个槽以保证所有的行都在同一个页面上，这样就可以最小化在链表中查找所需的I/O操作的次数。只有当第一个磁盘块中的空间已经用完的时候，才会在下一个页面上分配空间（新的页面会按堆组织表的方式来分配）。当溢出页面的个数不断增加的时候，插入行所需的时间也会越来越长，因为必须搜索整个链表，浏览多个磁盘块。

为了在散列聚簇中获得较高的性能，有必要提供较大的HASHKEYS和SIZE参数值，这样出现页面之间溢出的情况就会相当少了。当然，如果我们像例8.4.2一样把表depts和emps做聚簇，其中deptno是聚簇键，每一个deptno值对应了儿百个emps表的行，那么对应的每个槽不可避免地会出现一个长的溢出链表占用多个磁盘块的情况。但是，这不是散列聚簇创建的目的。当散列聚簇发生溢出的可能性很小的时候，查询计划将使用散列来访问聚簇中的行；但是，如果溢出经常发生，那么查询优化器将放弃使用散列访问，而使用辅助的B树索引访问。

考虑一下例8.5.1中的情况，50 000个行放在聚簇acctclust中的100 003个槽中。在这种情况下，磁盘块平均是半满的，所以溢出是很少发生的（我们将在下一节中讨论溢出出现的次数）。但是，如果我们把500 000个行放在100 003个槽中，那么就会有很多溢出出现。平均5行会散列到同一个槽中，这些磁盘块会被最先的100 000条记录填满（假设SIZE参数的值足够大，能装得下一个accounts行），从那以后就会发生溢出。由于有大量的溢出页面，散列就不再是访问行的有效方式。查询优化器将改为使用在accid上的辅助（唯一）索引来访问行，因为acctid是表的主键，所以该辅助索引是存在的。

调整散列聚簇一个通常的规则是，Create Cluster语句中的HASHKEYS参数设为使用的聚簇键最大可能个数的两倍，SIZE参数应该比磁盘块小，以提供更有效的访问（虽然散列聚簇可以支持较大的SIZE值，但是此时效率较低），但是SIZE的值应该能足够容纳一个聚簇键值所对应的所有的记录。因此，例8.4.2中emps-depts聚簇是不应该使用的，除非我们知道每个部门职员个数的上限保证了将填满一个磁盘块。一个散列聚簇中只有一个具有单一聚簇键值的表是非常普通的。

2. 所用槽的个数不会增加

前面我们说过，一旦散列表中的槽的个数S指定以后，S的值就不能再增加了。这个限制的原因是：散列函数可以被认为有两个步骤组成，在第一步里，先产生一个和键值有关的伪随机数 $x = r(keyvalue)$ ， x 是(0,1)之间均匀分布的浮点数。第二步，槽号 $h(keyvalue)$ 是散列函数根据下列公式计算出来的：

$$h(keyvalue) = \text{INTEGER_PART_OF}(S * r(keyvalue))$$

该公式的结果是0,1,...,S - 1之中的随机槽号。大多数的散列函数都采用这两个步骤，因为这种方法里对于任意给定的S值，通用函数r都可以得到0和S - 1之间整数的均匀分布。但是，如果槽的总数发生了变化，例如变为S'，我们可以发觉散列函数

$$h'(keyvalue) = \text{INTEGER_PART_OF}(S' * r(keyvalue))$$

将不会得到与前面计算出来的所有槽位置相同的槽号。例如，如果 $r(keyvalue)=0.33334$ ，而且 $S=2$ ，那么

$$h(keyvalue) = \text{INTEGER_PART_OF}(2 \times 0.33334) = 0$$

但是如果 $s = 3$ ，那么

$$h'(keyvalue) = \text{INTEGER_PART_OF}(3 \times 0.33334) = 1$$

这就是为什么在散列组织表中不能增加用于存储数据的页面的个数的原因。

因为这些方面的考虑，如果散列键值的个数超过了初始定义的HASHKEYS值而导致页面经常溢出的时候，用户需要创建一个有较大HASHKEYS值的新散列聚簇，然后把行从原来的聚簇移动到这个新的聚簇里。一个简单的方法是，创建一个有所需HASHKEYS值的新散列聚簇，然后使用CREATE TABLE...AS把原来聚簇表中的行移动到新的聚簇中。

3. 散列主索引的优缺点

为了让一张表或者多张表能够存储在散列聚簇里，用户必须要了解一些特性：

- 在大多数查询中，能使用某一个查找键。使用该查找键的某个键值能检索出唯一一行或者较小的行集合（这意味着不经常使用范围检索）。
- 给定一个唯一键值而检索出的较小的行集合，能简单地存放在一个磁盘页面上。
- 聚簇需要使用的HASHKEYS能很容易地事先计算出来。而且一旦表装载后，在短时间内不会有较大的变化。

大量的数据库方面的文章分析表明，在谓词的精确匹配时散列主索引比B树索引好，它能减少查询的I/O操作次数。这一点我们已经讨论过了，因为通过B树结构需要通过一些高层的目录节点才可以检索出需要的行。对于那些需要进行有效地根据键值查询的大表而言，通常需要三层目录节点，有时候需要四层B树的目录节点。然而我们应该注意，这些访问中的大多数和I/O操作是没有关系的——内存缓冲区能保证这些目录节点中的一部分是驻留在内存中的，对于随机行访问而言，通常只需要一次真正的访问目录节点的I/O操作。但是，和B树需要两次I/O操作才能访问行相比，散列索引只需要一次访问行的I/O操作的优点还是很重要的。但是，某些情况下的另外一些因素能使B树比用户想象的更有效。考虑下面两张表：

autodeposit		employees			
	eid	eid	bank	acctid	weeksal

第一张表autodeposit是一个包含职员标识eid的单列表，是为那些希望每周自动存款的顾客创建的。第二张表employees有下列几列：eid、银行号、账号和每个职员每周的薪水。

我们现在说明在一个从这两张表中读取行的应用程序中，使用B树能比使用散列聚簇更有效。一个典型的每周执行一次的程序，将从autodeposit表中一条一条地读取记录行，然后根据读出的eid的值访问employees表中行，并在正确的账号中自动存款。对于B树索引而言，是在eid列建立B树索引来访问employees表的行。我们可能会想到在表employees的eid列上创建一个主散列索引，因为这样可以给我们访问这些行的更有效的I/O操作。但是，事实证明这个想法是错误的。

另外一个可选的方法是，先根据eid按顺序把表employees装载到一个索引组织的表中，这样就可以提供一个聚簇组织（另外一个方法是先按eid的值排序，然后把行装载到一个普通的散列组织表中，然后在eid列上创建一个索引。这样做的优点是能让我们在表employees的

其他列创建辅助索引。但是新插入表中的行并不是按`eid`排序的。我们还需要进行重组织才能保持聚簇的效率)。现在在每周运行这个程序的时候, 我们发现`autodeposit`表中的行的顺序和`employees`表中行的顺序是一致的。我们可以发现当对`autodeposit`表中的行进行循环的时候, 应用程序是按`eid`的值对`employees`表中的行进行访问的。通常, 对`employees`表中的后续行进行访问, 是通过一些已经在缓冲区中的目录节点向下访问, 最后访问一个也已经在缓冲区中的行。

I/O操作中访问的表`employees`中的页面总数和表中页面的总个数是相等的(我们假设大部分职员需要自动存款, 因此需要涉及到大多数的`employees`表的数据页面)。另一方面, 对于散列聚簇而言, 每一个新访问的行都可能存放在一个和前一个页面毫不相关的页面中, 即使它正被后续的键所访问。如果每一个散列页面中平均有10个职员希望直接存款, 散列主索引需要的I/O操作的次数将是B树聚簇索引的20倍。这里我们假设散列磁盘块过多而不能都驻留在缓冲区中, 而且和数据页面相比, B树页面很少。这两个假设都是很普通的。

因此我们看到了一个散列结构对行的访问并不像其他有序结构那么有效的例子。当然, 我们可以把行堆起来, 因为`eid`引用实际上并不是随机的。另外一方面, 这是个很普通的情况, 读者应该意识到适当安排表中行的顺序能节省访问表所需的大量资源。

8.6 向靶上的空位随机投掷飞镖问题

现在我们来解决一个在数据库评估方面经常遇到的概率方面的问题。这个问题是: 有M个随机的空位, 现在向上面投掷N个飞镖。假设我们有N个飞镖, 我们向远处的一个靶投掷飞镖, 因此虽然我们想击中某个空位, 但是靶上的每个空位被投中的概率是相等的。我们可以对这个问题加一些限制, 然后提一些关于靶上的飞镖最终分配的问题。例如, 我们可以问: 靶上有多少空位里有飞镖?

因为某些空位里多个飞镖, 这个问题的答案就不是简单的N。它完全依赖于空位的容纳能力限制。

1. 不限制空位的容纳能力: 有多少个空位里有飞镖?

让我们首先解决最标准的问题: 有M个空位和N个飞镖, 而且每个空位里飞镖的个数没有限制。投掷完N个飞镖后, 有多少空位里有飞镖? 我们用S表示某次投掷完毕后, 被击中的空位的个数, 而用E(S)表示对于给定N和M被击中的空位个数的数学期望值。

描述这个问题的最好的方法是把它分成下列三种情况:

- 第一种情况: 飞镖的个数远远小于空位的个数, 即 $N \ll M$ 。
- 第二种情况: 飞镖的个数和空位的个数相近, 即 $N \approx M$ 。
- 第三种情况: 飞镖的个数大于空位的个数, 即 $N > M$ 。

在第一种情况下, 飞镖的个数远远小于空位的个数, 所以出现有很多空位中有两个或多个飞镖情况的概率是很小的。因此有飞镖的空位个数接近于飞镖的个数, 即 $S \approx N$ 。(大多数情况下, S略小于N, 因为有些空位里有多个飞镖。)

在第三种情况下, 飞镖的个数远远大于空位的个数, 比如飞镖的个数是空位个数的三倍。这样大多数的空位里都有飞镖(每个空位里含有飞镖的平均个数是三)。但是如果有很多空位的话, 仍然有一些空位里没有飞镖。因此这种情况下 $M - S \ll M$, 但是 $M - S \neq 0$ 。

现在我们考虑一下第二种情况: 假设飞镖的个数和空位的个数很接近(为了准确起见,

我们假设 $N=M$ ，有一些空位里有两个飞镖，还有一些空位里有三个或者多个飞镖。这就意味着有相当个数的空位里没有飞镖。（ $M-S$ 占了 M 的一部分，）但是直觉告诉我们试图估计没有飞镖的空位个数是很困难的。等一会我们可以看到当 $N=M$ 的时候， $E(S) = M(1 - e^{-1})$ ， e 是自然对数的底2.1828...。

为了计算给定 N 和 M 的 $E(S)$ ，我们首先考虑一下投掷某个飞镖 d 时，某个特定空位 s 没有被击中的概率 P 。因为有 M 个空位，而飞镖击中每个空位的概率是一样的，所以某个飞镖击中空位 s 的概率是 $1/M$ ，而空位 s 没有被击中的概率是：

$$\Pr(\text{空位 } s \text{ 没有被飞镖 } d \text{ 击中}) = (1 - 1/M).$$

如果空位数 M 的值较大，那么这个概率值是相当大的。但是现在我们计算一下投掷 N 个飞镖后，某个空位没有被飞镖击中的概率是多大。投掷每个飞镖都是一个独立事件，所以某个空位 s 没有被飞镖击中的概率是下列事件的与 s 没有被第一个飞镖击中且 s 也没有被第二个飞镖击中，...，且 s 也没有被第 N 个飞镖击中。

独立事件与的概率是各个事件概率的乘积，所以有

$$\Pr(\text{空位 } s \text{ 没有被 } N \text{ 个飞镖击中}) = (1 - 1/M)^N$$

因为有 M 个空位，每个飞镖都有相同的概率，所以没有被飞镖击中的空位的数学期望等于空位的个数乘以某个空位没有被击中的概率：

$$[8.6.1] E(\text{没有被飞镖击中的空位的个数}) = M(1 - 1/M)^N$$

因此，被飞镖击中的空位的个数 $E(S)$ ，等于 M 减去上面计算出的值：

$$[8.6.2] E(S) = M(1 - (1 - 1/M)^N)$$

在微积分中，我们知道由 e 表示的数定义为如下的极限形式：

$$e = \lim_{x \rightarrow 0} (1 + x)^{1/x}$$

如果我们假设 M 很大，我们可以用 $-1/M$ 来代替 x ，然后得到 e 的近似值：

$$e \approx (1 - 1/M)^{-M}$$

或者

$$[8.6.3] e^{-1} \approx (1 - 1/M)^M$$

从[8.6.2]中，我们看到：

$$E(S) = M(1 - ((1 - 1/M)^N)) = M(1 - ((1 - 1/M)^M)^{N/M})$$

代入公式[8.6.3]我们得：

$$[8.6.4] E(S) = M(1 - e^{-N/M})$$

因此，被飞镖击中的空位的期望值就近似为 $M(1 - e^{-N/M})$ ，而没有被飞镖击中的空位的期望值就近似为：

$$[8.6.5] E(\text{没有被击中的空位}) \approx Me^{-N/M}$$

在本章的例子中，我们会提供一些使用这些公式的例子。

2. 每一个空位只能占用一次：重试的次数(重散列链表)

在某些数据库系统中，把行散列到槽时，不允许在槽中存放多行。如果插入行的时候发

现对应的槽中已经包含了某个行，那么必须要找另外一个槽来插入该行。系统采用的方法是，把键值重新散列到某个槽中，如果该槽还是非空的，则继续散列直到某个槽是空的。分配给某一特定的键值的槽总是连续的，尽管不同键值之间会在某个槽上发生冲突。如果发生冲突的时候，重新散列总是试图散列到同一页面上的槽中，但是如果在原来的页面上没有空槽时，那么会试图映射到后续页面中的槽中。读者应该注意的是，这个算法只能创建相对有限的行，因为不能创建多于槽个数S个行。我们想要知道的问题是，在找到一个空槽前，需要重新尝试多少次？在上面的飞镖的问题中，我们可以把这个问题描述成下面的简写形式：

靶上有M个空位，有N支飞镖。每个空位只能插一支飞镖。重试直到所有飞镖都在靶上为止。

“每个空位只能插一支飞镖”的意思是，靶上每个空位的空间只能插一支飞镖。如果第二支飞镖也想插到这个空位里时，它会弹回。而“重试直到所有飞镖都在靶上为止”的意思是每次飞镖弹回的时候，我们拿起它，再重新投掷该飞镖直到它击中靶为止。最后当所有飞镖都插在靶上的时候，我们有 $N \leq M$ ，每个飞镖只能在一个空位里。在这种情况下，问题“有多少个空位里有飞镖”的答案就是N。但是我们的问题是另外一个：

投掷完第N支飞镖后，重新尝试次数的数学期望是多少？

答案和已经成功投掷的飞镖数 $N - 1$ 有关，下面的分析给出了一个很好的近似解。我们假设M很大，那么当我们试图投掷第N支飞镖的时候，该飞镖打中已经投掷了的 $N - 1$ 支飞镖的概率是 $P = (N - 1)/M$ ，也就是说该空位没有被飞镖占用的概率是 $(1 - P) = (1 - (N - 1)/M)$ ，这也是我们把最后一行成功插入第一个找到的位置的概率。我们称这个冲突链表的长度是L（只需要一次检测）并将其写成：

$$\Pr(\text{冲突链表的长度为 } 1) = (1 - P)$$

另外一方面，为了保证冲突链表长度为2，散列的第一个位置必须是满的（概率是P），第二个位置必须是空的（概率是 $(1 - P)$ ）。利用多个独立事件同时发生的概率是单个事件发生概率的乘积，我们有：

$$\Pr(\text{冲突链表的长度为 } 2) = (1 - P)P$$

现在计算冲突链表为3的情况，第一个和第二个位置都必须是满的（概率是 $P \cdot P = P^2$ ），第三个位置是空的（概率是 $1 - P$ ）。简单地扩展这一过程可以得到后面的情况：

$$\Pr(\text{冲突链表的长度为 } 3) = (1 - P)P^2$$

$$\Pr(\text{冲突链表的长度为 } 4) = (1 - P)P^3$$

...

$$\Pr(\text{冲突链表的长度为 } K) = (1 - P)P^{K-1}$$

现在计算冲突链表的期望长度E(L)，即所有上面的概率乘以相应长度的和：

$$E(L) = (1 - P) + 2(1 - P)P + 3(1 - P)P^2 + 4(1 - P)P^3 + \dots$$

或者写成

$$[8.6.6] \quad E(L) = (1 - P)(1 + 2P + 3P^2 + 4P^3 + \dots)$$

如果冲突的最大数目较大，该和趋于一个较大的值。现在我们给出求这个和的一个简单的公式。首先考虑函数f(x)：

$$f(x) = x + x^2 + x^3 + x^4 + \dots$$

这是有名的无限几何级数 $a + ar + ar^2 + ar^3 + \dots$ 中 $a = r = x$ 的情况，该几何级数的和为 $a/(1-r)$ ，所以我们得到：

$$[8.6.7] f(x) = x + x^2 + x^3 + x^4 + \dots = x/(1-x)$$

对上式求导可得：

$$[8.6.8] f'(x) = 1 + 2x + 3x^2 + 4x^3 + \dots = 1/(1-x)^2$$

这样我们把公式[8.6.6]改写一下：

$$[8.6.9] E(L) = (1-P)(1 + 2P + 3P^2 + 4P^3 + \dots) = (1-P)f'(P)$$

根据公式[8.6.8]，我们把 $f(P)$ 换成 $1/(1-P)^2$ ：

$$[8.6.10] E(L) = (1-P)f'(P) = (1-P)(1/(1-P))^2 = 1/(1-P)$$

因此冲突链表长度的数学期望是 $(1-P)$ 的倒数。在前面的讨论中，我们说过 $(1-P) = (1 - (N-1)/M)$ ，因此我们有：

$$[8.6.11] E(L) = 1/(1 - (N-1)/M) = M/(M - N + 1)$$

现在我们考虑几个例子。如果散列结构是50%满的 ($(N-1)/M = 0.5$)，那么 $P=0.5$ ，而 $E(L) = 1/0.5 = 2$ 。如果表是90%满的，那么 $P=0.9$ ，而 $E(L) = 1/0.1 = 10$ 。图8-24表示了这种关系。当 $(N-1)/M$ 趋近于1的时候，冲突链表长度趋于无穷大。

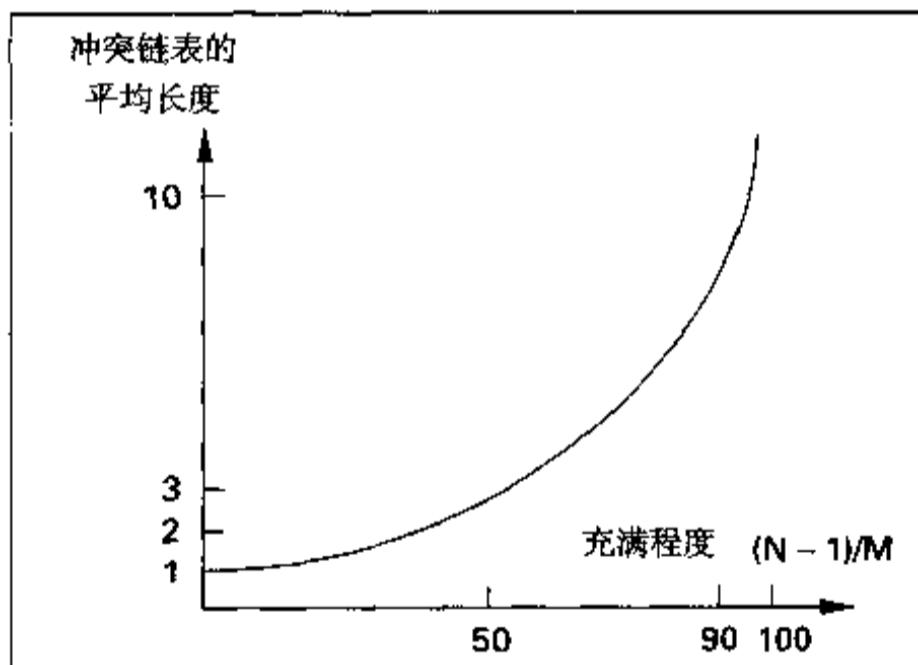


图8-24 $E(L)$ 与充满程度 $(N-1)/M$ 的关系

和我们想象的一样，表越满冲突链表也越长。令人惊奇的是，当充满程度接近于100的时候，冲突链表的平均长度增加得很快。如果每一个页面上可以存放20行，而把表装载到半满，平均长度为2的冲突链表的长度不太可能会增加到21。但是如果表满的程度达到95%，冲突链表的平均长度则为20。所以半数的冲突链表会需要第二个页面。这就说明避免让表太满是多么重要。平均来说，我们可以遍历冲突链表的一半长度来找到与某个唯一键值相关的行，因此如果有大量的输入项被散列到冲突链的第21位或者更后面的时候，开销就会随之增大。散列表如果含有重复键值，就需要更长的冲突链表，因为具有相同键值的行肯定会发生冲突。在我们的推导中，我们假设不同的散列项有独立的随机位置，当重复键值存在的时候这一点是不存在的。

3. 散列页面什么时候会填满

在计算冲突链表将到新的一个页面的概率的时候，多个行是否可以在一个页面中存储得下并不是实质性的问题。前面两小节中使用的散列算法在冲突链表需要使用新的页面前，需要使用完原来页面上的所有空间。现在问题是：如果我们只把表装载到部分满，散列链表什么时候会需要使用新的页面？学过统计学的读者可以看看下面粗略的讨论。

假设一张有100万行的表，每个磁盘页面上有20行。如果我们把每个磁盘页面都装载到半满，即每个页面上只存储10行，我们需要200万个散列槽来保存一行（在ORACLE中，这意味着HASHKEYS的值为2 000 000，SIZE的值是一行的长度）。因此我们将需要使用100 000个数据页面。但是占用率是随机的，我们需要正态分布函数来估计任意一个页面上行的个数。对于表中的任意一个页面，某一行被散列到该页面的概率是 $p = 1/100 000 = 0.00001$ 。我们用 $q = (1 - p) = 0.99999$ 来表示槽是空的概率（这个值和1非常接近，以至在下面的计算中，这个区别并不是很最重要）。所以散列到该页面的行数的数学期望值 $E(r)$ 是 $N \times p$ ，其中N是散列的行的总数：

$$E(r) = 0.00001 \times 1 000 000 = 10$$

平均每一页只包含了10条记录。这个概率分布的标准方差 σ 可由下式得到：

$$\sigma = \sqrt{Npq} = \sqrt{1000000 \times 0.00001 \times 0.99999} \approx \sqrt{10} = 3.162$$

因此超过 $E(r)+\sigma=13.162$ 个行在一个页面上是 $1 - \Sigma$ 事件，它的概率为 $\Phi(1.0)=0.158$ ； $10 + 2 * 3.162 = 16.324$ 个行在一个页面上是 $2 - \Sigma$ 事件，概率为 $\Phi(2.0)=0.228$ 。超过20个行散列到该页面上的概率是 $\Phi(10/3.162) = 0.000831$ 。因为表中有100 000个页面所以其中有83个页面溢出。

有一点儿统计学知识的读者在访问一个正态分布表时，可以运用这些估计值。

推荐读物

如果读者想了解驻留内存的数据结构，例如散列表和2-3树，可以参考Aho、Hopcroft和Ullman的书[1]。Knuth撰写的书[5]是这个领域的一本经典的书籍，该书是《The Art of Computer Programming》系列中的一本。在前面几章中我们讲述过的很多产品的SQL参考手册，对于理解如何在表上建立索引也是很有帮助的。对理解不同产品的索引能力有帮助的参考手册，我们用黑体字表示。

- [1] Alfred V.Aho, John E.Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Reading,MA: Addison-Wesley, 1987.
- [2] Don Chamberlin. *A Complete Guide to DB2 Universal Database*. San Francisco: Morgan Kaufmann, 1998.
- [3] *DB2 Universal Database Administration Guide*. IBM. Available at <http://www.ibm.com/db2>.
- [4] *DB2 Universal Database SQL Reference Manual*. Version 6. IBM. Available at <http://www.ibm.com/db2>.
- [5] Donald E.Knuth. *The Art of Computer Programming: Sorting and Searching*. Volume 3. Reading, MA: Addison-Wesley,1997.
- [6] *INFORMIX Guide to SQL: Reference*. Version 9.2. Menlo Park,CA:Informix Press,1999.

<http://www.informix.com>.

[7] *INFORMIX Guide to SQL Syntax*. Version 9.2. Menlo Park, CA: Informix Press, 1999.

<http://www.informix.com>.

[8] *INFORMIX Performance Guide*. Version 9.2. Menlo Park, CA: Informix Press, 1999.

<http://www.informix.com>.

[9] *ORACLE8 Server Administrator's Guide*. Redwood Shores, CA: Oracle.

<http://www.oracle.com>.

[10] *ORACLE8 Server Concepts*. Volumes 1 and 2. Redwood Shores, CA: Oracle.

<http://www.oracle.com>.

[11] *ORACLE8 Server SQL Reference manual*. Volumes 1 and 2. Redwood Shores, CA:

Oracle. <http://www.oracle.com>.

[12] *ORACLE8 Server Application Developer's Guide*. Redwood Shores, CA: Oracle.

<http://www.oracle.com>.

习题

在本书最后的“习题解答”中有答案的习题用•标记。

如果没有特别声明，所有问题中的磁盘页面大小都是2048字节。

8.1 假设某个DBA在ORACLE中执行如下Create Table语句：

```
create table customers (cid . . .)
storage (initial 20480, next 20480,
maxextents 8, minextents 3, pctincrease 0);
```

(a)• 当这个文件第一次创建的时候，会分配多少字节的磁盘空间？

(b)• 该表最多可以容纳多大的空间？

(c) 如果pctincrease 0改成pctincrease 100，那么(b)的答案又是什么？

请给出计算过程。

8.2 考虑一下例8.2.1前给出的关于INGRES的TID的定义。TID定义了4字节的无符号整数，值从0到 $2^{32} - 1$ 。TID允许在每个大小为2048字节的磁盘页面上有512个槽，虽然实际存放的数目可能会少一些。对于页面上的每一行，在行目录里都保持一个2字节的偏移量。

(a)• 如果实际上每一个大小为2048字节的页面上有512条固定长度的行，行的最大长度可以是多少？(别忘了记录目录里的偏移量。)

(b) 大小为2048字节的页面上可以存放多少条长度为50字节的行？

(c)• TID中可以表示的磁盘页面的最大数目的精确值是多少？

(d) 在INGRES的表中，可以存放的长度为50字节的行的最大数目是多少？

(e) 另外给一个TID的定义，它能增加INGRES表中能存放的行的最大数目。每个表的TID的定义依赖于表中行的最小长度m(从Create Table语句中计算得出)，而且可以使用一个更现实的计算一个页面中可以存放的行的最大数目的估算方法。

8.3 考虑例8.3.1中的二分查找法。推广它使它能处理任一长度为N的有序数组并允许它查找，大于但又最接近x的值，形式如下：

```
int binsearch(int x, int N)
```

对于给定的 x 和 N ,返回满足 $\text{arr}[K].\text{keyval} \geq x$ 的最小 k 值,否则,如果 x 大于任何一个键值则返回-1。

8.4 假设有一张雇员表emp中有200 000行,每行的长度是100字节。用下列语句建立该表:

```
create table emp (eid integer not null, . . . ) pctfree 25;
```

这里PCTFREE n子句指的是每个页面上都必须要留有n%的可用空间。

- (a)• 假设页面中可以使用2000个字节,行目录中的偏移量是2字节,而且这2个字节已经包含在100个字节中了。估算表emp中行所需的页面的个数。

假设在ORACLE中使用下面的命令在表emp的eid列创建一个唯一索引:

```
create unique index eidx on emp (eid) pctfree 20;
```

- (b) ORACLE中的ROWID占用6个字节(如果是非唯一索引,则需要占用7个字节),另外每个键的列还需要附加一个字节的空间。假设eid列需要占用4个字节。估算一下eidx B树索引中,每个叶子页面的索引项的个数及叶子层页面的总数。注意Create Index中的PCTFREE子句。
- (c) 估算一下B树中每一个目录层的节点页面的个数。假设B树中目录节点的每一项(sepkeyval, np)的长度和叶子层中项的长度相等。仍然需要注意PCTFREE子句。
- (d)• 执行查询语句select * from emp where eid between 10,000 and 20,000时所需的I/O次数是多少?假设eid的值是从1到200 000的连续值,而且假设表emp中记录并不按eid聚簇。假设磁盘页面没有被缓存,因此每读一次磁盘页面都需要一次I/O操作。假设I/O操作的速度是80次/秒(忽略CPU时间),那么执行该查询所需的时间是多少?

- (e) 假设表emp中的记录按eid聚簇,重新计算(d)。执行这个查询所需的I/O时间是多少?

- (f)• 执行查询select count(*) from emp where eid is between 10,000 and 20,000所需的I/O操作的次数(提示:查询优化器不会从表中取行,除非它需要这么做)。执行该查询的时间又是多少?

8.5 重做例8.3.4,假设节点是70%满的,但是对于IBM机页面的大小是4096字节。而且假设图8-17的DB2 UDB索引压缩RID的长度是4字节。在索引中假设有100万个索引项,索引键是city,它的数据类型是char(16)。最终一共有200个城市分布在该表的行。假设在B树的目录层没有使用压缩。

8.6 (a)下列语句是否正确,请验证你的答案:

- (i)• DB2 UDB允许一行跨越多个数据页面存储。
- (ii)• ORACLE的ROWID的长度比DB2 UDB的RID的要长。
- (iii)• 对于一个有序列表而言,在磁盘I/O方面,二分查找法不比B树有效。
- (iv)• 对于一个含有100万个不同键值的列表,二分查找法需要测试10次才可以找到一个键值。
- (v)• 假设内存缓冲区中可以存放1000个页面,我们从100万条记录中随机检索行,在足够长的时间后,我们有1/1000的概率可以不需要执行磁盘I/O操作就可以

检索一行。

- (b) 写出ORACLE的Create Table语句的STORAGE子句(只写出子句),使得表的磁盘存储区大小是10KB,在第二次增长后,每一次增长的容量是前面一次的两倍,当总容量大于5120KB时停止(即 $10 + 10 + 20 + 40 + \dots + 2560$)。写出计算过程。

8.7 在ORACLE中,有一张表students有400 000个行,每行的长度是200字节。假设已经执行了下列命令来装载students表:

```
create table students (stid char(7) not null primary
key, . . . <other columns>
pctfree 20;
```

- (a) students中的行需要占用多少个页面?假设每个页面有2000个字节,忽略行目录的偏移量和页面头信息写出计算过程。
- (b) 假设stid上有一个PCTFREE = 25的索引。假设ROWID占用6个字节,对于单列的键还需要1字节的额外开销,计算项的大小。然后计算B树中叶子层页面的数目。
- (c) 计算B树中各个上层中页面的个数。
- (d) 执行如下查询需要多少次I/O操作?

```
select * from students
where stid between 'e000001' and 'e020000';
```

假设:有1/20的学生的eid值在这个范围内, stid上的不是聚簇索引,而且没有内存缓冲区。

- (e) 假设stid上的索引是聚簇索引,重新计算(d)。
- (f) 在(e)的假设条件下,执行下列查询所需的I/O操作的次数是多少?

```
select count(*) from students
where stid between 'e000001' and 'e020000';
```

8.8 假设有一个散列组织表,每一个页面上可以存放20行。

- (a) 散列表中有100 000个页面,槽的个数是多少?给出计算过程
- (b) 假设有100 000个页面,另有100万行散列到这张表中,计算散列冲突链表长度为21的概率。给出计算过程。
- (c) 如果一行散列到某个页面,但是该页面没有可用空间,该行需要存放到另外一个页面,那么我们称该页面“溢出”。
- (i) 如果某个页面上存在长度为21的散列冲突链表,是否就意味着该页面溢出?
- (ii) 某页面溢出,是否就意味着该页面上有长度大于或等于21的散列冲突链表。

8.9 在8.6节中,我们假设M个空位和N个飞镖,而且空位中可以容纳的飞镖的个数没有限制。

- (a) 有10 000个空位和128支飞镖,计算被击中的空位的个数。使用公式[8.6.2]和[8.6.4]。注意128是2的幂,所以公式[8.6.2]可以通过连续自乘($1 - 1/M$)来计算,读者也应该使用这种方法。在这两种情况下,被击中的空位的个数是否都接近于128?
- (b) 假设有16 384个空位和16 384支飞镖,重新计算(a)。注意16 384是2的幂,所以

也可以采取自乘的方法来计算。两个结果接近吗？空位被击中的概率是否接近 $(1-e^{-1})$ ？

(c) 当有10 000个空位和30 000支飞镖的时候，利用公式[8.6.5]计算没有被击中的空位的个数。

8.10 (a) • 当我们向靶上只投掷一支飞镖的时候，某个空位里不可能有两支飞镖。当飞镖的个数远远小于空位的个数的时候，某个空位中有两支飞镖的概率也很小。但是，如果给定空位的个数是 M ，一直投掷飞镖，会在某一点出现两支飞镖在同一个空位的概率大于0.5。计算当 M 等于365的时候，出现上述情况时，已投掷飞镖的个数。（这就是所谓的“生日问题”。在一间屋子里，要求两个人同一天生日所需要的人的总数是很小的。）

(b) [难]如果空位个数是 M ，再计算(a)。

8.11 (a) 在CAP的orders表的month列上创建一个位图索引。写出这个脚本。

(b) 该索引的密度是多少？

(c) 保存每个位图需要多少字节？

(d) 假设行是按图2-2中行的顺序被给定索引，画出'feb'的位图。

8.12 (a) 图6-11给出了一个数据库设计的例子，订单是多项的（和我们的CAP的orders表不同，orders表中订单是单项的）。有一个弱实体Line_items通过has_item联系连接到Orders实体。再看看习题7.3及它的答案。数据库CAP没有line_items表。要求删除数据库CAP中的表，然后写一脚本创建一张新表orders2，该表没有列pid、qty和dollars，但是包含了其他所有的列。该脚本还应该创建customers、agents和product表。另外还有一张列为lineno, ordno, pid, qty和dollars的表line_items，该表的主键为(ordno, lineno)。接下来，复制loadcap文件（复制为loadcap2，这样以后能够返回到该CAP数据库的初始状态）。然后编辑loadcap包含的文件，为这些表增加若干行使得大多数的订货行的line-item值为1，但是对包含1000个p04订货的订单1011增加line_item为2。写出一个脚本来显示如何完成上述工作。

(b) (a)中的工作完成之后，删除刚才创建和装载的表，然后在索引表聚簇orderdata中重新创建表orders和line_item。（表customers和agents还是保留原来的方式）。写出创建该聚簇中表的脚本，并说明你如何算出指定的SIZE值。

(c) 在orders2表的month列上创建一个(辅助)位图索引。

8.13 继续习题8.12，将表创建为一个含有1000个散列键的散列聚簇并完成装载。给出ORACLE创建的散列表的大小。并在month列上增加一个位图索引。

第9章 查询处理

回顾一下，从第8章的开始就提到的查询[8.1.1]：

```
select * from customers  
where city = 'Boston' and disent between 12 and 14;
```

当一个数据库系统接收到这样一个查询的时候，它先通过一系列的查询编译步骤，然后再开始执行。在第一个阶段，即语法检查阶段，系统分析查询同时检查它是否符合语法规则，然后把查询语法中的对象同视图、表和在系统表中的列匹配，接着进行适当的查询重写。在这个阶段，系统确认用户有正确的权限并且查询没有违反任何相关的完整性约束。然后就进入查询优化阶段。在这个阶段，系统要用到表和列的统计信息，例如，在表中有多少行以及通过它们各自适当的统计信息找出相关的索引。这个阶段是一个复杂的过程，我们把它看做是“决定做什么”，这个过程的结果是生成一个用来执行查询的程序化的存取方案。然后进入执行阶段，在这个阶段，存取方案被执行，其中系统通过存取索引和表从数据中得出查询的结果。

本章的目的是解释查询处理的一些基本原理，尤其强调整查询优化的思想。对于执行一个给定的查询，通常有许多不相上下的存取方案，犹如在下国际象棋时为了赢（至少不输）会有许多种下法。系统查询优化器竭力选择那些在减少各类资源使用（例如，磁盘I/O的数目，CPU时间，等等）的同时减少运行时间的存取方案。对于一个复杂查询，查询优化器可能并不选择最好的可行方案，与国际象棋手下一盘完美的比赛相比较，它更进一步，它的目标是在优化过程中花足够的努力来确保一个合理的好的选择。查询优化的基本思想是如何利用索引、如何利用内存来积累信息和执行一些中间步骤，例如，排序、如何确定连接执行的顺序等。通过本章的学习，对于一个特定的查询，读者应当能够列出并且分析由数据库系统选择的一个存取方案。读者也应当理解是什么组成了一个“好的”或者“坏的”存取方案。最后，读者就可以更好地理解DBA（数据库管理员）应当采取的“调谐”步骤以提高查询性能。例如加入索引、索引行的不同聚簇、表的分解等。

本章还介绍了对于一个给定的SQL查询如何选择存取方案的许多考虑，但是仅局限于介绍对于多种可能的方案为什么应当选择一个特殊的存取方案，而不是介绍查询优化器怎样选择。通常，查询优化器会产生很多可能的策略，并且从中选取一个，而这是一个相当复杂的问题。因为一个计算机程序是没有“直觉”的，对于编写查询优化器程序的编程者而言，创建一个高效的决策程序并从中选取合适的存取方案是一项很困难的工作。这种类型的算法过程（例如动态规划技术）超出了本文讨论的范围。我们将主要依赖于直观感受来说明为什么一个存取方案比另外一个好，同时假定查询优化器通过算法搜索可以得到相同的结果。这种方法通常用于解释编程语言语句的效果上，而不考虑编译程序完成该效果的细节。

OS/390的DB2和DB2 UDB 直到现在，我们已经分别从很多不同的数据库产品中提及了它们的特性，但是在本章的余下部分，我们主要集中于IBM的DB2产品。在先前的讨论中，我们已经论述过DB2 UDB，它是运行于UNIX、Windows NT上的“通用数据库”，最近又在

OS/390的IBM大型机上应用。在查询优化的讨论中，我们把注意力集中于“老大哥”产品DB2 for OS/390，在本章中，简称为DB2，仅仅运行于IBM大型机的IBM OS/390操作系统上。我们相信不同产品的结构差异对于成功表达查询优化的细节而言并不重要。因此在本章中，我们在集中叙述DB2的同时只是偶然提及DB2 UDB和ORACLE。

对于查询优化，DB2已经有一种很复杂的方法，它提供了很多先进的执行特性，它们为查询提供了高性能，这并不是说其他产品没有自己的重要特性。查询优化是一个新的领域，起源于20世纪70年代中叶，没有一个产品对于好的思想具有垄断性。事实上，最近的很多研究结果提出了新的性能特性，而这些还没有完全被集成到商业产品中。本章仅仅集中于已经在商业产品中应用的特性。本章的最后三节提供了一个查询的基准测试程序（benchmark），它是一个测试查询性能的工业标准，它出现在本书第1版关于DB2 for OS/390的前一个版本MVS DB2的介绍。基准程序的测试以及相应的具体查询存取方案的讨论对于增进读者对前面几节查询性能原理的理解具有很大的帮助。

9.1 基本概念

我们先介绍一些基本概念作为我们讨论查询优化的基础。首先考虑的是在优化一个查询时尽量使系统利用的资源最小化。最终这个问题归结到减少在计算机设备上的投资和用户的时间（也可以看做是老板的开支问题）。我们也会考虑DBA提出的调谐数据库系统的特殊要求和理解由查询优化器产生的方案输出。

1. 查询资源利用

查询优化器通过选取一系列可供选择的查询存取方案中最好的一个来竭力减少对特定资源的利用。要考虑的主要资源是CPU时间和需要的I/O数量。尽管计算机内存也是一个重要资源，但是不同目的的内存容量通常在系统初始化时就已经确定。例如，在内存中用来保存公共磁盘页面的缓冲区的数目由DBA预先设定。因为查询优化器对此无能为力，所以，它通常通过在各种内存阈值上选择查询方案中的不同行为，以一种相对简单的方式来反映。我们将在本章的后面看到这样的例子。

相反，在不同的存取方案中执行查询所使用的CPU和I/O资源是必须要考虑的。对于每一个可供选择的存取方案都有一个相关的CPU代价(用符号COST_{CPU}(PLAN)表示)以及I/O代价(用COST_{I/O}(PLAN)表示)。无论何时，如果有两个不能比较的代价时，那么很有可能两个查询计划PLAN₁和PLAN₂在资源使用上也是不可比较的，如图9-1所示。

显然，在较小的CPU代价方面PLAN₂优于PLAN₁，但是在较小的I/O代价方面PLAN₁优于PLAN₂。为了提供一个使模糊性最小的可供计算的简单方法，DB2查询优化器定义了存取方案的总代价的概念，以COST(PLAN)来表示，它是I/O和CPU代价的加权和。

$$\text{【9.1.1】 } \text{COST}(\text{PLAN}) = W_1 \cdot \text{COST}_{\text{I/O}}(\text{PLAN}) + W_2 \cdot \text{COST}_{\text{CPU}}(\text{PLAN})$$

这里，W₁和W₂都是正数，体现了在总代价中两种代价的相对重要性。优化器的工作就是在处理一个查询的所有可能的方案中选择代价值COST (PLAN) 最小的。在下面几节中，我

	COST _{CPU} (PLAN)	COST _{I/O} (PLAN)
PLAN ₁	9.2 CPU秒	103 次读
PLAN ₂	1.7 CPU秒	890 次读

图9-1 带有不可比较的I/O和CPU代价的两个查询方案

们讨论怎样分析不同的查询方案，并相对精确地从中导出相应的I/O代价。单纯从理论的角度导出CPU的代价并不是一件容易的事，因为这依赖于CPU指令集的细节和数据库系统实现的效率。（当然，对于一个特定数据库系统的查询优化器而言是可以估计一个方案的CPU代价的，这需要用到内部函数所用的CPU时间的表格，但是这是一个繁琐的无关紧要的细节，因此，在本章中我们不涉及这个问题。）通常说来，执行一个查询对于不同的存取方案CPU代价的变化比I/O代价变化要小得多，除此之外，与每一个I/O代价相关的CPU代价常常是一个重要的因素，因此，将I/O代价减到最小的同时也会将CPU代价减到最小。这意味着图9-1的不可比较的CPU和I/O代价是不通用的。下面几节主要集中于I/O代价的定量估算，只有当CPU代价可能产生很重要的变化时，再以定性的方式来讨论CPU代价。

系统的工作负荷

我们把一个系统的工作负荷定义为查询和由系统用户提出这些查询的频率的结合体。例如，我们有一个查询系统，它用来帮助5000个保险调解员做他们的日常工作。也许保险调解员在他们的工作中会提出两种查询Q1和Q2。

Q1检索来自于一张使用索赔号的意外索赔清单。

Q2检索所有索赔号，由被保险人的(lastname, firstname)作为索引。

当它们被提交时，这些查询对于索赔号和被保险人的姓名有不同的值，但是，对于这些参数的任意特定的值，回答这些查询所需的CPU和I/O资源几乎是相同的。观察了调解员工作高峰期的情形，我们注意到，平均说来小组的成员每秒40次Q1类查询，20次Q2类查询。这样的频率可用图9-2反映。当然，这只是工作负荷的一个简单的例子；在实际生活中，可能有更多类型的查询和更少的具有精确数字的提交率。

对于一个系统上给定的工作负荷以及由查询优化器产生的针对工作负荷中的查询生成的查询执行方案，可以计算每秒需要的CPU和I/O资源。从这一点以及一个设备费用的列表中我们可以把需求转换为看上去更为直接的度量：为了支持工作负荷需要的系统的美元费用。例如，考虑一个工作负荷，它有一个出现频度很高的查询，用到大约1000次I/O但是很少有CPU。开始，我们注意到我们有相对长的响应时间（每秒只能执行80个随机的I/O，在一个查询方案中，后续的I/O必须等待以前的I/O完成）。在这样的工作负荷下，尽管我们可以购买相对便宜的CPU，我们也许需要购买大量磁盘来提供需要的I/O存取速率。在选取大一些或小一些的CPU系统方面我们通常有很多自由度。例如，IBM大型机数年来一直保持着CPU计算能力和产品价格的线性增长关系（当然其中有些是人为的因素）。

所有这些要购买的东西都需要事先规划好，这就是为什么DBA在一个应用实现之前就竭力要得到工作负荷估算的原因。工作负荷中的每一个查询都根据其峰值频率和资源使用被换算为计算机硬件上的费用，也就是所用设备的“合理租金”。很长的响应时间也是一种花销，主要是因为公司需要雇佣更多的雇员（雇员用来等待响应的时间通常是被浪费的），还因为过长的等待会使雇员在沮丧中辞职从而增加人员流动率。以这种方式来看，如果我们可以提高查询优化器的性能，使它可以在多种存取方案中选取一个更好的，那么马上就会节省费用。对于DBA而言，这是一个需要研究的至关重要的领域。

查询类型	每秒钟的提交率
Q1	40.0
Q2	20.0

图9-2 具有两个查询的简单工作负荷

2. 收集统计信息

任何精致的查询优化器都需要得到它处理的不同的表、列和索引的统计信息。例如，调用如下查询：

```
select * from customers
  where city = 'Boston' and discnt between 12 and 14;
```

如果customers表在一简单的数据页上仅包含3列，那么查询优化器应当忽略任何现存的索引而进行一次表扫描——对于列的直接搜索——来使这两个谓词的条件得到满足。另一方面，如果在10 000数据页面上有100 000列，那么查询优化器将通过使用在city或discnt上的索引来节省资源，如果这些索引存在的话。为了计算资源的费用，查询优化器需要进行估算，例如，当city='Boston'时有多少行。如果对于city列（值是'Boston'）只有一个值，那么所有的列都是这个数值，查询优化器就知道这个索引没有任何用处。我们将在下面几节考虑具体的统计信息。

当一个表装入内存和创建索引的时候，统计信息不是被自动地收集的，必须由DBA给出特定的命令，这些命令就是收集需要的统计信息并把结果放入系统表的命令实用程序。在表上的更新会导致在统计信息上没有反映出来的变化，统计信息会因此过时，如果没有足够频繁收集统计信息的命令的确保，也许会导致查询优化器不恰当的决定。

DB2使用RUNSTATS命令（语法如图9-3所示）来把统计信息放入目录表中。RUNSTATS命令的最简单的形式（以CAP数据库为例）如下：

```
runstats on table poneil.customers;
```

实际上能够检索到我们需要的绝大多数统计信息。详细细节将在下面几节中叙述。

RUNSTATS ON TABLE username.tablename [WITH DISTRIBUTION [AND DETAILED] {INDEXES ALL INDEX indexname}] [other clauses not covered or deferred]

图9-3 DB2 的RUNSTATS 命令语法

ORACLE使用Analyze(分析)语句收集统计信息，并将其放入数据字典中。分析语句的语法如图9-4所示。

ANALYZE {INDEX TABLE CLUSTER} [schema.] {indexname tablename clustername} {COMPUTE STATISTICS other alternatives not covered} {FOR TABLE FOR ALL [INDEXED] COLUMNS [SIZE n] other alternatives not covered}

图9-4 ORACLE的Analyze语句语法

3. 检索查询方案

查询优化器建立一个存取方案包含一系列过程存取步骤，或者简称过程步骤或存取步骤。这些过程步骤依赖于特定的数据库系统，并像在对象程序中由编译程序产生的用以执行高层次程序（在这里是非过程性SQL查询）逻辑的一系列指令一样放在一起。在下面几节中，我们将讨论DB2中的存取步骤，其中包括：

- 表空间扫描（在DB2中或者在DB2 UDB中的表扫描）。
- 索引扫描。

- 等价唯一索引查找。
- 非簇簇匹配索引扫描。
- 簇簇匹配索引扫描。
- 仅使用索引的扫描。

大多数数据库系统都有一个类似的过程步骤。然而，即使是具有类似名字的步骤对于不同的数据库系统也有不同的影响，这样就需要不同的过程流程，它十分类似于不同处理器的机器指令。尽管DB2不提供位图索引和散列存取，但是这个产品具有完善的过程步骤的集合并且支持在其他系统中拥有的绝大多数存取概念，同时拥有许多有价值的高性能的能力，这种能力在范围检索方面尤其有价值。

对于一个给定的查询，DBA使用由数据库系统提供的命令来产生查询方案。在DB2 UDB中，DBA使用如下形式的特殊形式的SQL语句：

```
EXPLAIN PLAN [SET QUERYNO = n] [SET QUERTYTAG = 'string']
FOR explainable-sql-statement;
```

这个语句可以在（大型机器的）DB2和DB2 UDB上正确执行。在DB2的情况下，它把行插入到用户创建的名为plan_table的DB2表中，其中每一行代表为由explainable_sql_statement创建的计划中的每一个独立的存取步骤。在DB2 UDB情况下，它插入被称为explain_tables(解释表)的用户创建表的集合中。我们在下面的讨论中将避免讨论相对复杂的DB2 UDB解释表，除非提到我们将介绍的所有DB2的简单能力在DB2 UDB（和ORACLE）中有反例。在Explain Plan(解释方案)语句指定的queryno是在DB2中表plan_table的一列，因此它可以作为从表plan_table中查询带有规定值n的检索。这样DBA就可以执行下列语句从而产生我们已经作为例子的查询的查询方案：

```
explain plan set queryno = 1000 for
select * from customers
where city = 'Boston' and discnt between 12 and 14;
```

然后从DB2的表plan_table中检索与此查询（在本情况中只有一个）有关的所有行，然后，DBA（或者拥有产生计划的感兴趣的使用者）可以执行语句：

```
select * from plan_table where queryno = 1000;
```

表plan_table（有时我们简称为方案表）含有大量的列，事实上在第一次遇到时列的数量是惊人的。当这些列与论述的内容相关时，我们将这些列的重要性，但是作为介绍，我们只涉及到一个名为ACCESSTYPE的重要列。在DB2方案中，一次表空间扫描步骤将扫描存储该查询的单个表的行的一个表空间（也可能是出现在同一页面上的来自于其他表的行，就像在ORACLE的簇簇中那样），同时确认与那张表相关的WHERE子句的搜索条件。当这个时候，我们可以看到在表plan_table的ACCESSTYPE列上有一个'R'，在下面几节中我们用ACCESSTYPE=R来表示。表plan_table的另外一列将给出扫描执行的表的名称。

需要注意的是，原始的Select语句可以用在From子句中视图表名上，而存取方案中用到的只是物理的基本表。我们在先前介绍的大多数方案包含有单个步骤，这样在方案表中就只有一行出现。（每一个单次存取方案步骤仅仅指单个表，因此在FROM子句中涉及到多于一个表的情况时，就需要多步方案，或者因为连接运算或者因为一系列子查询。）另外一个ACCESSTYPE对应于一个方案步骤，其中一个单个的索引用于限定检索表的行。例如，一个

city索引用于检索所有满足谓词city='Boston'的谓词的行，同时，在WHERE子句中不被那个索引（如discnt在10和20之间）限定的任何谓词在限定的行被存取后被确认。对于这种类型的方案我们可以在方案表中看ACCESTYPE=I。

在ORACLE中，对于一个特定的查询的方案可以放在缺省的plan_table中或者任何其他的带有相同列的命名表中，用一个带有下列语法的语句：

```
EXPLAIN PLAN [SET STATEMENT ID = 'text-identifier'] [INTO [schema.]tablename]
FOR explainable-sql-statement;
```

Explain Plan语句出现在《ORACLE8 Server SQL Reference》(推荐读物[9])中。要获得关于怎样解释一个执行方案的信息，参见《ORACLE8 Server Tuning Guide》(推荐读物[8])的第21章。

9.2 表空间扫描和I/O代价

作为我们的过程存取步骤研究的开始，我们先考虑在DB2中的表空间扫描。回顾一下，在代表一个表空间扫描的方案表中的行在ACCESTYPE列上有字母R，我们写为ACCESTYPE=R。

例9.2.1 表空间扫描步骤 DB2中的表空间扫描步骤是一个算法步骤，其中，一个表（以及相应的表空间的数据页面）中的所有数据被扫描，同时，表中的行被在WHERE子句中搜索条件的相关谓词限定。在DB2框架中，来自于不同表的行可以混合在一个表空间的共同区域上，这就是为什么我们把它称为表空间扫描而不是表扫描的原因。然而，在下面的内容中，我们假定被一次表空间扫描引用的所有页仅仅包含来自于一个表的行。

假定我们有一张有200 000行的表employees，每行200字节，同时数据页面装载了70%（在DB2中，确定全部数据页面怎样被缺省地装载的PCTFREE说明在Create Tablespace语句中给出，也可以在Create Table语句中被重写）。我们假定每一个4KB页面在头部大约用96个字节，留下4000字节，装载70%，即我们有2800字节可以使用，因此每个页面可以装载14行。这样，对于200 000行，需要的数据页面的总数是 $\text{CEIL}(200\ 000/14)=14\ 286$ 页。现在考虑下面的查询：

```
select eid, ename from employees where socsecno = 113353179;
```

这个查询将要检索一个具有给定社会保障号的一个雇员。如果我们没有一个关于socsecno的索引，那么我们能做的就是用一次表空间扫描来扫描整个表，寻找符合WHERE子句描述的所有行。（我们假定满足条件的只有一行，但是，依赖于搜集的统计信息查询优化器也许不知道这个情况。）注意，数据扫描步骤事实上是存取方案的全部，因为这个步骤的结果完全回答了查询。在本表中有14 286个数据页面，因此本方案的I/O代价， $\text{COST}_{\text{io}}(\text{PLAN})$ 为14 286次读，也就是14 286次随机的I/O。我们曾经说过，我们不必估算 $\text{COST}_{\text{cpu}}(\text{PLAN})$ ，但是可以假设整个的代价同I/O的代价是成正比的。 ■

表空间扫描在其他产品中会有其他的名称。例如，直接搜索、数据扫描或者表扫描。（事实上，在DB2 UDB中被称为表扫描，因为每一个页面仅包含来自于一个表的行。）例如，我们在前面讨论过，在例8.4.1中，有关于执行大量随机的I/O所需要的时间。现在，我们可以检查一些我们用来计算执行时间的假设了，同时介绍一些新类型的I/O，它们被称为顺序预取I/O和列表预取I/O。

I/O代价的估算

对于例9.2.1的查询方案，我们可以从14 286次读的I/O代价中得到什么暗示呢？曾经讲过在一个普通的磁盘上一次随机的I/O执行的时间大约为0.0125秒（一秒钟的1/80），就像我们在8.2节见到的那样。但是这并不意味着顺序的14 286次随机I/O需要的时间为 $14\ 286/80=178.6$ 秒。很有可能的情况是，这14 286个数据页面可以被分布放在10个不同的磁片上，系统可以使十个活动臂上的磁头同时移动，因此每一个磁片只是将1/10的页面装载入内存（每磁片1429个页面），这只需要原来时间的1/10（17.9秒）。这种使多个活动臂的磁头在一个查询中同时行动的方法叫做I/O并行性（I/O parallelism）。因为在查询方案中，CPU时间远远小于磁盘I/O请求的时间，我们期望CPU能够跟得上检索来自于数据页面中相应行的速度，这个速度同系统把页面从磁盘读入内存缓冲区的速度相同，它以大量并行移动的磁盘活动臂并发地占用CPU时间。当然，这样的并行读并不需要额外的CPU，但是它很明显地降低了查询的时间。

(1) I/O并行性和磁盘分条

I/O并行性是好几个数据库系统都提供了的特征。例如，一些系统提供了把数据页面在10个不同的磁盘上进行分条的能力，第1个页面在磁盘1上，第2个页面在磁盘2上，……，第10个页面在磁盘10上，然后第11个页面在磁盘1上，第12个页面在磁盘2上，……，第N个页面在磁盘 $((N - 1)\%10)+1$ 上。（“X%Y”就是“X MOD Y”的C语言表达式，参见图9-5）。当我们以这种方式把页进行分条时，从一个表中读取连续页面的系统可以把多次I/O读请求放到以后，这样就可以保持所有的磁盘臂在大多数的时间都忙碌。因为当执行一次表空间扫描时我们可以很容易地预测将来的表空间页请求，我们仅仅需要一个支持分条的结构把逻辑页面地址翻译成在多个磁盘上的物理设备地址，同时把以后的I/O请求传到适当的设备。这件事根本不困难，因为在这样一个分条结构下，被分配的区域必须以一种很好定义的方式跨过多个磁盘。

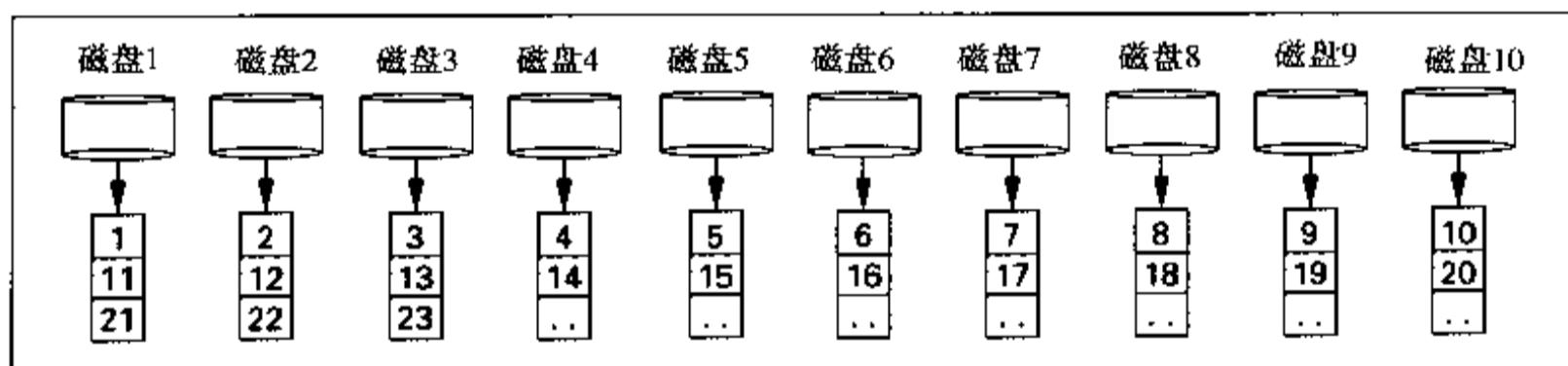


图9-5 在分条的磁盘上连续的磁盘页面

有很多原因可以说明为什么对并行I/O进行分条没有被广泛采用。当然，它需要DBA增加额外的工作，他必须在多个磁盘上分配等量的空间。平衡磁盘臂负载以获得最大的并行性也是一个复杂的问题：如果在工作负荷中，十个磁盘中有一个进行I/O读取时极端忙碌，那么那个磁盘就会成为并行存取的一个瓶颈，也许会导致在分条的表空间中的其他磁盘不能充分利用。再有，许多系统没有足够的磁盘设备或者一个足够大的应用程序来要求并行扫描对于这种情况进行协调。

另外一点就是磁盘分条实际上并没有节省任何计算机资源花费。即使工作负荷被分裂到几个不同的磁盘设备中，我们还是要求相同数量的随机I/O。如果我们通过对磁盘设备在运行期间收取合理租金的方式来估算例9.2.1中14 286次随机I/O的I/O代价，那么在这个费用上，并行性不起作用；我们只能用更多的设备来换取更少的时间，同时付相同的费用。对于查询节省下来的时间可以应付先前提到的雇员的费用、减少浪费的时间以及因为等待查询结果的

沮丧而造成雇员的辞职。因为I/O并行性相对说来很少，我们在下面的讨论中假定我们总是面临串行I/O这样一种选择，即第二个请求只有在第一个I/O请求被完成以后才能放入一个方案中，除非我们另外规定。然而，对于串行I/O依然有一个未公布的技巧，它实际上在执行一个表空间扫描时可以节省大量的资源。

(2) DB2中的顺序预取I/O

我们已经隐含地假定当一个接着一个执行14 286次读（随机I/O）时将需要单个随机的I/O时间的14 286倍。在8.2节，我们推出一个事实，单个随机的I/O时间平均为0.0125秒，在图9-6中是这段时间的分解。（我们对于8.2节的分析中的DB2的4KB页面没做任何修改）。

寻道时间	0.0080秒
旋转延时	0.0040秒
传输时间	0.0005秒(一次传输4KB)
总共	0.0125秒

图9-6 DB2的随机I/O时间的分解

现在让我们回到一次表空间扫描中执行14 286次I/O时的考虑。由于区域在磁盘介质上被分配的方式，我们通常会发现表中连续的页面在磁盘上也是连续的，连续的页面处于相同的磁道。这意味着，对于来自于一个表中连续的数据页面的I/O代价中，它通常不含有磁盘臂从一个柱面移动到另外一个柱面的寻道时间。让我们回忆在8.2节开始时的讨论，在磁盘臂的寻道时间中，大约0.016秒为平均的寻道时间，其中假定连续存放的页面被随机存放在两个端的柱面位置上。在我们现在的情况中，通常没有要移动的距离，我们可以认为寻道时间完全消失。事实上，乍看起来好像旋转的延迟时间也一并消失，因为通常我们是按顺序一个磁盘页面一个磁盘页面进行检索。这样，看上去似乎把连续的数据页面放入内存缓冲区需要的正常时间不能以完全传输率进行的说法没有理由，这个速度对于4KB的页面大约为0.00125秒。在本书的第1版中，对于这样的分析有很多争议，因为前一页被返回以后，对于磁盘上下一页的请求将由于时间过长，以至于磁头不经过磁盘上完全旋转的这段时间就不能够检索到下一页，因此必须加上大约为8ms的旋转延时。但是，最近在数据库系统并行性方面有了新的发展，它使得控制器可以不等待先前的请求被填满就执行连续的请求，这些发展使得在大多数情况下进行这种高速的顺序I/O成为可能。

事实上，磁盘控制器通常支持的一种通用的缓冲区技术加强了这种分析。通过这样一个控制器，无论什么时候请求一个磁盘页面的I/O，I/O总要执行于含有被要求的磁盘页面的整个磁道上。来自于磁道的数据被缓冲到由磁盘控制器拥有的内存中去，被请求的单个页面返回给数据库服务器。然而，如果以后对于页面的请求还是在相同的磁道上（就像它们在一个表空间扫描的情况下），那么数据从磁盘控制器的缓冲区中读出并立即被返回到数据库的服务器上，而不必对磁盘进行存取。所以，在第一次寻道之后I/O的速度几乎是瞬时的。磁盘控制器每次检索一个磁道，这样就把一个顺序I/O变为一个比随机I/O几乎快十倍的I/O速度！在图9-7中我们显示这种加速后的顺序I/O。

当单个磁盘页面被请求时，关于读入整个磁道并且放入缓冲区的磁盘控制器特性的一个问题是：这一点总要发生，即使在它达不到目的的时候。因此读入整个磁道的0.008ms的时间总是被加到由数据库执行的每一个I/O中去，甚至也加到随机的I/O中来拾取那些对于存取该磁道上的下一个页面无用的行。

寻道时间	0.00050秒(对于一个磁道的缓冲寻道时间)
旋转延时	0.00025秒(仍然需要等待要求的、被缓冲的第一页)
传输时间	0.00050秒(一次传输4KB)
总共	0.00125秒

图9-7 平均的顺序I/O时间的分解

DB2系统提供了一个易于及时回答的I/O请求，被称为顺序预取I/O。顺序预取I/O的思想就是系统指定了按序从磁盘上读入大量数据页，通常是32页，但是仅仅在需要的时候。就像在我们稍后看到的那样，DB2也花费了很多努力来确定可能的时候怎样按序放置I/O请求。结果，DB2可以在磁盘的全速旋转传输速率中按序执行32个页面读，即使在磁盘控制器磁道缓存能力被关闭的时候也能如此。这些都归结为：顺序的大量槽的检索在速度上大大超过随机I/O，就像我们在图9-8中所描述的那样。

随机I/O (以秒为单位)	顺序预取I/O (以秒为单位)
每个页面总共消耗的时间 0.0125	0.00125

图9-8 顺序预取和随机I/O的时间的比较

下面我们用简单的经验规则来说明，在可以应用顺序预取的场合，它在速度上十倍于随机I/O，它的速度是每秒800次I/O，也就是说，每次I/O为0.00125秒（即1.25ms）。

例9.2.2 带有顺序预取的表空间扫描 回忆例9.2.1的表空间扫描，其中给定的Select语句具有14 286次R的I/O代价（R代表随机I/O）。在标准的随机I/O假设下（无并行性），我们计算这个查询执行的时间，期间一个磁盘臂被完全使用，那就是 $14\ 286/80=178.6$ 秒。事实上，因为在一次表空间扫描中的磁盘页面请求能够以连续方式被完成，I/O将被执行得更快。我们写为 $COST_{io}(PLAN) = 14\ 286S$ 。被读入的页面数量末端的字母S指出页面是连续的并且在DB2中采用顺序预取。这个查询的时间是 $14\ 286/800=17.86$ 秒。这是一个重要的I/O费用的节省，因为花在磁盘臂上的时间实际上已经被减小了十倍，因此在设备上的租金也同时减小了相同的因子。从另外一个角度看，在相同的磁盘上我们用顺序预取的方法可以执行十倍于随机I/O的查询。 ■

一个查询也许会同时用到顺序预取I/O和随机I/O。我们把总的I/O耗时作为最好的通用衡量标准，因为在磁盘臂上的租金同磁盘臂在使用中的耗时直接成正比，所以在这个重要的意义上，顺序预取I/O比随机I/O效率提高十倍。

现在是介绍DB2方案表中另外一个列的好时机，这个列为PREFETCH列。当对于一个给定的存取步骤选择了顺序预取时候，我们会看到那个方案表的行上的PREFETCH列有字母S，我们把这种情况写为PREFETCH=S。这样，例9.2.2中的表空间扫描步骤在DB2方案表中写为ACCESSTYPE=R、PREFETCH=S。仅列有这两列的方案表在图9-9中给出。

(3) 列表预取

ACCESSTYPE	PREFETCH
R	S

图9-9 例9.2.2的方案表

在DB2中，还有一种预取方式称为列表预取，在这种预取方式中，一系列需要读入内存缓冲区的页面（通常为32页）被提前提供给磁盘控制器，但是页面不必像顺序预取那样具有连续的顺序。使用列表预取，通常磁盘控制器执行最有效的可能的移动序列，它可以确保执行连续的读（使用称为电梯算法的方法）。这样的I/O比随机I/O请求高效得多。对于这样一个列表预取执行的时间不遵循简单的通用规则，然而，在图9-8中，一个顺序预取执行所需的0.00125秒时间对于列表预取而言通常是一个不可达到的最优值。列表预取比随机I/O更为高效（我们认为是每秒80次I/O），同时又比顺序预取I/O低效一些（我们认为是每秒800次I/O）。实际的速度由磁盘上的数据页面的间隔来确定，但是，根据经验规则，在下面的问题中，作为经验规则我们假定列表预取以每秒200个I/O的速度来进行。在下面的内容中，我们将大量的使用这些随机I/O、顺序预取I/O、列表预取I/O近似速度值，因此，图9-10以表格形式列出刚才提到的经验规则。尽管这些是相当粗略的数据，但是它们对于实践中的查询通常可以给出一个相当好的估计。

随机I/O	顺序预取I/O	列表预取I/O
80页/秒	800页/秒	200页/秒

图9-10 I/O速度的经验规则

在使用列表预取计划表中的一个存取步骤将用PREFETCH=L来指明，也就是在PREFETCH一栏中的字母为L，类似于当顺序预取被使用时的字母S。一个既不使用顺序预取也不使用列表预取的存取方案将在PREFETCH列上有一个空白值。

(4) 有关于预取的其他内容

在下面的章节中，我们将举出许多例子来强化这些概念。但是，我们在9.6节讨论DB2中列表预取的前提条件以前，我们不考虑列表预取。在那节以前，我们总是认为表中的行通过随机I/O或顺序预取I/O从磁盘读入内存。

有很多涉及到预取I/O的特殊情况。当从磁盘中读入的页面数小于32时，预取I/O依然是可能的；实际上，预取在只读入4个页面时也可以执行。自然，因为我们减少了用于缓冲寻道时间和旋转延迟的页面的数目，这样小的传输率会影响到在图9-10中给出的I/O速度，但是在接下来的计算中，我们通常不考虑这种影响。然而，任何当3页或者小于3页的页面要从磁盘上读时，即便在顺序预取或列表预取的前提下满足的情况下，你应当考虑使用随机I/O。

很明显，顺序预取和列表预取是很有效的。在DB2中关于顺序预取的一个重要的事情是系统具有选择是用随机I/O读单个页面或者用一个预取I/O读32个页面。对于所有一次进行8个页面读的折衷方案都将使它在帮助一些应用程序的同时又使另外一些应用程序处于不利地位。例如，当所有想要的信息已经被放单个页面节点上时，在一棵B树节点周围读入8个页面就会得不到任何值。实际上，这样一种浪费内存缓冲区空间的行为使得每一个人都处于不利地位，内存缓冲区空间应当用于保持在缓冲区中有用的信息。

9.3 DB2中的简单索引存取

就像我们在第8章论述的那样，在一个Select语句中被引用的表中的列可以参与索引，对于Select语句的一个查询方案有时可以利用那些索引来限定被选中的行，这样可以使得查询更为高效。在下面几节，我们考虑基本的索引存取步骤的能力。一开始，我们只考虑如下情况：

只涉及到一个表的Select语句，而表名在FROM子句中命名，在WHERE子句的搜索条件的子查询中没有新的表名，这样就可以避免连接过程。除非另外声明，否则我们把注意力集中在DB2的查询优化器上，同时，我们只是在括号中指明ORACLE产品的变化。在下面的例子中，表以分类的名字T1, T2, …来命名，这些表中的列以C1, C2, C3, C4, …来命名。

例9.3.1 匹配索引扫描、单列索引 假定在表T1的C1列存在一个索引C1X（在DB2中总是B树索引），发出下面的查询：

```
select * from T1 where C1 = 10;
```

这个查询在DB2中通过一个被称为匹配索引扫描的存取步骤来完成。在执行阶段，列C1上的B树索引被遍历到值为10的最左边索引项的叶子层（回忆这样的练习，它是利用例8.3.1的二分查找算法的变型来找出最左边的项）。然后，被那个项指向的源于表T1的行被检索到答案集合。紧接着，连续的项从索引的叶子层被检索到，同时索引值依然保持10，在项中从左边到右边传递，从一个叶节点传到下一个叶节点，必要时用到叶子的兄弟指针。对于每一个项，在T1中被指向的相应的行被检索放入答案集合。当索引值第一次超过10，匹配索引扫描就完成了。尤其需要注意的是，在B树上遍历到叶子层，只需要一次。在这次扫描中检索到的行可能被以列C1为序的聚簇索引找出，也可能不能；我们没有假设对于C1是否有聚簇索引。 ■

匹配索引扫描是一个单步骤的查询方案，同表空间扫描共享此特征：当步骤完成时，已经回答了查询。在方案表中，图9-11是其中的三列，这是一个匹配索引扫描，就像在例9.3.1中看到的那样，ACCESSTYPE=I（这代表一个索引扫描），ACCESSNAME=C1X（被扫描的索引的名字），MATCHCOLS=1。MATCHCOLS给出在索引中匹配列的数目；对于一个在单列上的索引的一次索引扫描，这个值至多为1。

现在考虑下面的索引：

```
[9.3.1] select * from T1
      where C1 = 10 and C2 between 100 and 200 and C3 like 'A%';
```

回顾一下，在3.9节的图3-19中列出的由许多逻辑谓词组成的WHERE子句的搜索条件，它们由在图3-20中定义的逻辑运算符，如AND、OR和NOT来连接。上述[9.3.1]中的select语句包含三个由逻辑符AND连接的谓词：(1)涉及到列C1的比较谓词（这是一种特殊的比较谓词，被称为等价匹配谓词），(2)涉及到列C2的BETWEEN谓词，(3)涉及到列C3的LIKE谓词。在这些情况的每一种，在相关列上的索引在DB2查询方案中都能够有效地用来限制检索到的行。这些谓词在DB2中被称为是可索引的。然而，不是所有的谓词都是可索引的，这意味着尽管用一个索引限制检索到的行是可能的，但是索引将不能够被很有效地使用。例如，谓词C1<>10就是不可索引的，尽管它可以在C1X的索引上用索引扫描，但是查询

```
select * from T1 where C1 <> 10;
```

更可能使用一个表空间扫描，在低效率运行时它总可以得到。在本章的后面，我们将给出关于可索引和不可索引谓词的一张详细列表。现在，我们想要研究[9.3.1]的Select语句的WHERE子句中的多重谓词也许被同时使用来限定检索到的行。

例9.3.2 匹配索引扫描、单列索引、额外确认的谓词 就像前面一样，假定在表T1的列C1上存在一个索引C1X，但是这次[9.3.1]的查询是这样提出的：

ACCESS TYPE	ACCESS NAME	MATCH COLS
I	C1X	1

图9-11 例9.3.1的方案表

```
select * from T1
  where C1 = 10 and C2 between 100 and 200 and C3 like 'A%';
```

我们假定在列C2和C3上没有索引。就像在前面所做的那样，这个查询在DB2中是通过一次匹配索引扫描存取步骤来完成的。在列C1上的B树索引被遍历到具有值为10的最左边索引项的叶子层。值为10的叶子层的项被从左到右遍历。然后，系统存取这些被项指向的来自于表T1的连续的行，在行上执行测试以确认其他的两个谓词：C2 between 100 and 200和C3 like 'A%'。满足这些测试的行被检索入答案集合。 ■

例9.3.1和例9.3.2中介绍的匹配索引扫描步骤被定义为：它是这样一个步骤，其中单个索引被用来检索所有的来自于满足WHERE子句中一些谓词（或谓词集合）条件的表中的行。在一个单列上有索引的情况下，匹配索引扫描通常代表一个列值的连续范围，相应于一个谓词，比如C1=10（在列C1上的索引C1X），C2 between 100 and 200（在C2上的索引C2X）或者C3 like 'A%'（在C3上索引C3X）。列值的一个不连续集合也可以用匹配索引扫描的一种特殊情况来检索，被称为IN-LIST索引扫描，用在如C4 in (1, 2, 3)这样的谓词上（列C4上有一个C4X的索引）。通过在一次匹配索引扫描中的单个索引一次满足几个谓词也是可能的。考虑索引C123X，用下面的Create Index语句来建立：

```
create index C123X on T1 (C1, C2, C3) . . . ;
```

例9.3.2的Select语句中，仅仅可利用C1X索引，我们注意到两个后面的谓词并不减少从T1存取的行的数目。我们仅仅用第一个谓词C1=10来限制行的数目，然后从表中存取每一行来满足其他的谓词。但是如果可以得到C123X索引，那么我们就可以通过依靠这个单个索引一次限制所有三个谓词（C1=10, C2 between 100 and 200, C3 like 'A%'）。作为一个类比，我们在图书馆目录中寻找名为James（C1=10），姓以字母H到K开头（C2 between 100 and 200），题目以字母A开头（C3 like 'A%'）。显而易见，我们可以通过单独检查卡片很有效地来限制检索到的书的数量。

在第二种类型的情况下，分离的索引存在于下面讨论的每一列上。这样，我们也许在C1上有索引C1X、在C2上有索引C2X、在C3上有索引C3X。我们利用三种不同的索引来回答例9.3.2的查询，但是这种方法也许不是显而易见的。通过类比，查询的过程是这样的：它先从每一个满足一个给定谓词的相关目录中抽取出一列目录卡片，再通过调用序数（RID）对每一列卡片依序排序，然后对三列卡片进行合并一交操作，最后，我们可以得到满足三个谓词条件的书。这个多步骤方法被称为多索引存取；在我们叙述一些更为基本的关于索引存取的概念之后，我们会更详细地讨论它。

1. 相等的唯一匹配索引存取

经验表明：当第一次介绍用索引来检索时，许多读者倾向于使用有唯一值的列。实际上，这相对来说并不是很有趣的情况，我们还是给出如下的例子。

例9.3.3 索引扫描步骤、唯一匹配 考虑例9.2.1介绍的表employees，它在列eid上有一个唯一的索引edix。我们给出下面一个SQL查询：

```
select ename from employees where eid = '12901A';
```

我们假定有一个PCTFREE=30的DB2的索引（参见图8-15 DB2 UDB的Create Index，对于DB2也一样），对于索引项可得到的索引的每一页面上的2800个字节。假定列eid需要6字节。因为我们知道RID需要4字节，我们可以假定每个项为10字节。这样在每一个索引页面上有

$2800/10 = 280$ 个项，有 $200\ 000$ 行时，在叶子层我们需要 $\text{CEIL}(200\ 000/280)=715$ 页，在目录层上需要 $\text{CEIL}(715/280)=3$ 页，再往上，我们只需要一个根页面。存取带有一个给定 eid 值的唯一的行需要存取三层B树的节点，然后，存取14 286数据页面中的一页并从指定行上检索 $ename$ 值。所以，看上去Select语句的I/O代价为4R。但是，考虑到我们把常用的磁盘页面放在内存缓冲区来减少实际磁盘I/O的数目。由查询优化器计算的 $\text{COST}_{io}(\text{PLAN})$ 数值以实际的I/O代价来评估。已经在内存中的页面仅仅增加了少量的CPU代价。就像我们将要在第10章详细介绍的那样（参见例10.10.2），购买足够的内存来保存在几分钟内被再次存取的所有的磁盘页面（我们假定是120秒，这个是在IBM DB2上指定的），这样的选择在经济上是正确的。这就是我们说要保留常用页面的缓冲区驻留的含义。

现在，我们需要关心 $eidx$ 上的B树索引被存取的频度。如果每秒有一个随机的 eid 值，我们应当期望找到树的根节点和所有下一个目录层中已经在内存缓冲的三个节点。另外一方面，存取叶节点的期望时间是715秒，多于120秒的阈值，因此我们不会期望在缓冲中找到B树的叶节点。显而易见，我们也不会期望在页面缓冲驻留区找到14 286数据页面。这样，在合理的缓冲假设下，对于这个查询方案的正确的I/O代价应当是2R，它代表读磁盘的B树的叶节点和数据页面的I/O代价。注意，所有刚才的分析都是基于一个完美的假设——已经购买了足够数量的内存以及缓冲区模式可把所有B树目录节点放入内存缓冲区中。

对应于这个唯一匹配索引扫描的方案表的行是： $\text{ACCESSTYPE}=1$ （隐含一个索引扫描）， $\text{ACCESSNAME}=eidx$ （被扫描的索引的名字）， $\text{MATCHCOLS}=1$ 。这个方案没有提到应当有一个唯一的匹配；通过指名列 eid 被定义为唯一的，我们就可以预测到这样一个性质。 ■

聚簇与非聚簇索引

回忆在8.4节，一个聚簇索引通常有一个B树索引目录结构，它们在磁盘的一个区域，引用以主键顺序存储在连续或近似连续的磁盘页面上的行，它们也可能在磁盘的不同区域。

在下面的例子中，我们引入一个命名为prospects的表。表prospects建模于由执行定向邮件应用服务的组织所拥有的表上。表prospects有 $50\ 000\ 000$ 行，它们对应于在美国有可能成为新产品的潜在顾客的市民。候选人的信息通常来自于担保问卷。（假定填写担保问卷的人期望他们的统计数据可以保留在记录上用于将来可能的邮件）。表prospects的行在DB2中通过一个索引addrx被聚簇，它代表候选人的地址：

```
create index addrx on prospects (zipcode, city, straddr) . . .
cluster . . .;
```

回顾一下，一个邮编号码确定了一个州（因为邮编号码不会跨越州界），并且通常对应于一个大城市的一个区。许多直接的邮件，比如区域性商店的销售通知，包含有一些邮编代码区域的小的地理上的区域，这样，我们就可以明白通过addrx的聚集可以得到极为高效的查询。

我们假设，表prospects和所有的索引 $\text{PCTFREE}=0$ ，因为我们并不期望在表prospects的生命期中有任何插入、甚至更新的操作。我们很少会把新的家庭加到表prospects中去。典型的，我们会每周重新装载表来确保最佳的查询效率，这时新的候选者会被加入。假设表prospects的每一行包含有400个字节，我们可以在每个4KB的页面上放10行，那么一个5千万行的表需要5百万个页面。

在addrx索引B树结构的叶子层，我们假设列zipcode可以用4个字节的整型数来表示，

city需要12个字节，straddr为20个字节，当然对于每一个项的RID需要4个字节。这样在addrx索引上的一个项需要40个字节。（我们简单地假设没有重复的索引值，也就是说，没有两个或多个候选人有相同的地址。这样，我们就可以避免考虑在图8-17中涉及到的DB2/DB2 UDB中的索引压缩类型。）在100%满的4KB页面中，我们可以放入100个大小为40字节的项。对于总共5千万的项，这意味着在树的叶子层有50万节点页面。目录层含有5千个节点，在这层之上含有50个节点页面，再往上就是根的页面。这是一个4层的B树。

表prospects也有大量的非聚簇索引。考虑一种可能的索引，比如在属性hobby上，它是由候选人在问卷上提供的个人主要的爱好。

```
create index hobbyx on prospects (hobby);
```

假设在候选人填写的表格的这一栏上有100种不同的爱好（例如，扑克牌、国际象棋、收集货币……）。我们说hobby列的基数，也就是具有不同属性值的数目，是100：

```
CARD(hobby) = 100
```

这是在计算一个查询方案的I/O代价时由DB2使用的一种统计，它也给了我们一个在hobbyx索引中出现的索引压缩数目的思想。同样的，我们可以假设在addrx索引中，它有100 000个不同的值，从00000到99999。（这实际上是过多估计了，只是为了简单）。基本上爱好的种类少意味着我们可以假设hobby列的长度同叶子层索引项是无关的，因为从图8-17的DB2的索引压缩我们明白索引关键字对于几百个RID只出现一次。这样，叶子层的项可以有4字节长度，每页面有1000项。因为对于每一种爱好我们大约有500 000个候选者与其对应，DB2的索引压缩可以用于它的最有效的形式，缓冲到每一个主键值有255个RID。在叶子层有5千万个项，且在一个页面上有1000个项，我们就有50 000叶子页面。在紧接着的上一层，索引关键字分隔符随着每一个项而出现（没有压缩）。如果我们假设12字节的项，那么一页有333个项，这样我们有 $50\ 000/333=151$ 个节点。再往上一层就是根。这些计算归纳在图9-12中。我们现在看更多的演示例子。

prospects表	addrx索引	hobbyx索引
50 000 000 行	500 000 叶子页面	50 000 叶子页面
5 000 000 数据页面	5 000 个第3层的节点	151 个第2层 节点
	50 个第2层的节点	1 个根节点
	1 个根节点	
	CARD(zipcode) =100 000	CARD(hobby) =100

图9-12 表prospects的一些统计信息

例9.3.4 匹配索引扫描步骤、非聚簇的匹配 考虑下面的SQL查询：

```
select name, straddr from prospects where hobby = 'chess';
```

为了回答这样一个查询，查询优化器在索引hobbyx上考虑一个索引扫描步骤。这个可以比作例9.3.1的匹配索引扫描步骤，只是这次我们多了做I/O估算。首先，hobbyx的B树的根节点被读出来，紧接着是沿着正确路径到下一个节点，一直到相应于带有值为'chess'的hobbyx项的最左边的叶节点被读出为止。下一步就是从叶子层检索连续的项，一直遇到具有hobbyx的值为'chess'为止。就像在先前指明的那样，对于hobbyx只有100个不同的值。如果

我们假设每一个值都是平均分布的，那么对应于值为'chess'的项有500 000个。遍历每个叶节点带有1000个项的500 000个项需要500次叶子页面的I/O代价，这些I/O可以用顺序预取I/O来执行。现在，对于这500 000个'chess'索引项的每一个，我们必须执行一个随机页面I/O来存取表prospects的合适的行，并且返回name和straddr值。既然有了这5 000 000数据页面和500 000行，我们期望每一行在分离的页面上（这实际是“N支飞镖在M个空位中”问题的一个例子，这在8.6节中已讨论过，将作为本章最后的习题的一个主题）。在任何情况下，我们都还没有理由认为检索到的行在数据页面上是彼此相邻的，因为hobbyx是一个非聚簇索引。这样每一个行检索需要它自己的随机I/O，并且执行这个查询需要的I/O的总数是：对于索引目录节点2R，索引叶节点500S，对于数据页面是500 000R。

用顺序预取以每秒800页的速度读500页的执行时间是 $500/800 = 0.625$ 秒；用随机I/O读500 000页以每秒80次I/O的速度的执行时间是 $500\ 000/80 = 6250$ 秒，或者说略小于2小时。显而易见，在hobbyx的B树上往下找的I/O，甚至于在叶子层遍历项的I/O与数据页面的I/O相比都是无关紧要的。 ■

在涉及到I/O执行时间的习题中，通常你应当假设索引目录页面是缓冲区驻留的，而索引叶子页面不是，除非是特别指明的。当这些数目小于总和的5%时，你就可以在答案中忽略索引I/O。然而，你应当很小心地包括索引I/O计算以及说明为什么你认为它们是不重要的。

如果例9.3.4的查询是以一个空间表扫描来执行的，我们可以看到用顺序预取I/O检索5 000 000页，这需要 $5\ 000\ 000/800 = 6250$ 秒——几乎同索引扫描是相同的时间。这是一个令人相当惊奇的结果，这就体现了在比较查询方案的代价时一个查询优化器的价值。基本上，在相同的时间内，用顺序预取I/O与随机I/O相比大约可以读十倍的数据页面。因为这个例子中用非聚簇索引扫描，它读出5 000 000页来检索相同数目的带有一个hobby索引的prospects，而使用预取I/O数据扫描，它也读出5 000 000页，两种方法在执行时间上和I/O磁盘臂上的费用近似相同。如果对于我们的表prospects有更多的爱好，那么每一个索引值会对应更少的数目，应当使用非聚簇索引扫描，然而，对于更少的爱好使用数据扫描也许更为合适。

在例9.3.4中，我们用CARD(hobby)=100这样的事实来推断谓词hobby='chess'检索到表prospects行的大约1/100。这个分数就是谓词的过滤因子；在下面的内容中，我们将经常提及这个概念。

例9.3.5 匹配索引扫描步骤，聚簇匹配 考虑下面的SQL查询：

```
select name, straddr from prospects
  where zipcode between 02159 and 03158;
```

回顾一下我们的假设，有100 000个不同的zipcode列值组成了addrx索引的第一个部分，数据行在addrx索引上被聚簇。这里的索引扫描步骤通过读到与addrx对应的B树的最左边的具有zipcode值为02159的项来执行它的搜索（组合的关键词值的其他值没有被具体化），然后从左到右检查项直到在范围以外的第一个zipcode值被搜索到为止。我们看到当最后一个例子的hobbyx被处理时，几乎有相同数目的叶子层索引项被处理。hobby='chess'的过滤因子是1/100，但是这里我们正检索带有100 000个不同值的zipcode索引中的1000个值。zipcode=02159的过滤因子是1/100 000，具有1000个值的范围谓词的因子是 $1000/100\ 000 = 1/100$ 。如果以每个项有40个字节，每页有100个项计算，那么被读入的带有500 000个项的

addrx的叶子层的页面的数目是5 000；因为可以用顺序预取，这意味着5000S的I/O代价。（在这之上的一个简单的检查是在addrx的叶子层的页面的数目，像在图9-12中那样，是500 000，我们将处理这些页面的1/100）。

现在我们将从表prospects中检索相同数目的行，或者说是500 000行，就像我们在例9.3.4中所做的那样。然而，因为在prospects中的行是以zipcode聚簇的，也就是说，它们不是以hobbyx聚簇的，在zipcode中连续检索范围的500 000行在表数据页面中被放在一起，一个页面中有10行。因此，我们可以假设我们仅仅需要检索表中的50 000页，我们用顺序预取I/O来完成这个任务，所以I/O代价是50 000S。（注意，我们要求5 000S来检索一个相同数目的addrx项，长度是40字节，这些行在长度上是400字节，因此这就是对我们工作的一个检查）。对应5000S+50 000S的总执行时间是 $55\ 000/800=66.8$ 秒，略大于1分钟，请与前一个例子的几乎2小时相比。这就是用聚簇优点的一个最显著的例子。 ■

来看例9.3.5，我们注意到查询优化器在检索被索引的行之前，选择检查在范围中的每一个索引项，即便这些行已经被索引值聚簇。一旦行范围内的左边末端被定位后，忽略索引项似乎就可以节省I/O时间，所以只需看选择范围的数据页面序列的行。然而，在DB2中不是这样做的。所以项总是被检查的一个重要原因是许多新行的插入填满了数据页面后聚簇的性质破坏了。当这种情况发生后，以后插入的在范围内的值就被完全放置在局部数据页面序列之外，在值的范围内检索到这样的行的唯一方法便是检查索引项。尽管我们在表被重新组织以前不向表prospects插入新的行，但是DB2不知道这些，所以在检索行以前总是要检查索引项。

2. 仅使用索引的检索

例9.3.5的另外一个有趣的发现是在要求的范围内被检索的索引项增加了5000S的I/O代价，被检索的数据行增加了50 000S的I/O代价。因为涉及的数据项在长度上是40字节，数据行是400字节，I/O代价乘以10是自然的。这个发现以及我们看到的其他一些例子使得我们以下列特性来总结出在数据行上的索引项的检索优点：

- 1) 索引具有这允许我们有效地检索值范围的目录结构。
- 2) 索引项总是按照值在连续的磁盘页面上按序放置，这样系统就可以用顺序预取I/O来获得较低的检索代价。
- 3) 索引项比数据行短，因此需要相应比例的更少的时间。

我们可以有意在聚簇的行上放置一个目录结构，其中在B树的叶子层行取代项，这样在聚簇的行结构上可得到性质1和性质2。当然，我们可以仅仅通过值的一个集合来聚簇行，同时我们可以对于不同的索引值创建大量的索引。我们可以在下面的一个例子中看到性质3的一个独特的优点。

例9.3.6 串接索引，仅使用索引的扫描 再一次考虑例9.3.5的SQL Select语句：

```
select name, straddr from prospects
  where zipcode between 02159 and 03158;
```

回忆一下，addrx索引被用于例9.3.5以存取表prospects中指定zipcode范围的行，addrx用如下语句创建：

```
create index addrx on prospects (zipcode, city, straddr) . .
  cluster . . . ;
```

但是，现在让我们假设——仅对于现在这个例子——一个替代的索引naddrx已经用如下语句创建：

```
create index naddrx on prospects (zipcode, city, straddr, name)
  . . . cluster . . .;
```

正像我们将看到的，索引naddrx是表prospects的一个聚簇索引这一事实对于这个例子不是很重要，因为回答这个例子一开始的Select语句只要通过引用naddrx中的值的分量而无须对表prospects的任何引用。naddrx索引的串接键值具有概念形式：

```
naddrx key value: zipcodeval.cityval.straddrval.nameval
```

因此每一个分量的值可以在指定范围内检索到的索引项被读出，它被检索到select语句的选择列表中。为了回答我们开始谈到的Select语句，在zipcode范围中的naddrx索引项被存取，就像它们在例9.3.5中那样；但是代替由项RID指向的被引用的相关行，在Select语句的选择列表中的name和straddr的值从索引naddrx键值的第4和第3个分量被检索。这种类型的索引扫描就是仅使用索引的扫描。

这种检索的I/O代价是多少？假设在naddrx上的索引项是60字节，比40字节addrx项长50%，这是因为额外的name列。在例9.3.5中，addrx索引项检索的代价是5000S，因此我们期望对于naddrx是7500S。因为在仅使用索引检索中没有其他I/O代价，我们期望查询时间是 $7500/800 = 9.4$ 秒。这可以与例9.3.5中的相同查询的66.8秒进行比较。 ■

仅使用索引的扫描发生在下面的情况下，它相等于一个Select语句，而且当在选择列表中被检索的元素可以从使用的索引项被产生，并且不需要回溯到数据行。尤其应该注意，只要一个索引扫描可以完全解决WHERE子句中的搜索条件，那么来自于单个表的任何Select语句可以用仅使用索引的检索被执行，其中count(*)是选择列表的单一元素；检索到的项仅仅需要被叠加以提供解答。当在方案表中用EXPLAIN来产生查询方案步骤时，一个仅使用索引的扫描步骤由方案的列ACCESTYPE=I来注明，同时通过一个我们第一次提到的新列INDEXONLY=Y来注明。在前面的例子中，我们有INDEXONLY=N。

例9.3.6显示了在使用串接索引中允许仅使用索引的检索的一个有价值的优点（甚至比非聚簇索引检索更有优势）。然而，创建一个串接索引的集合来预测查询需要的索引经常是很困难的。例如，考虑例9.3.6的查询，如果我们希望检索一些其他的prospects属性，并且把它们放入选择列表中，这样会发生什么情况呢？

```
select name, straddr, age from prospects
  where zipcode between 02159 and 03159;
```

这样，串接索引naddrx对于回答一个仅使用索引扫描的查询是不够的，毕竟它对于读入表中的行是必须的，除了现在被使用的naddrx索引需要比我们一开始就涉及的单个addrx索引更多的I/O以外。如果我们在一个简单的串接索引中预见到所有的需求，索引项会变得越来越长，直到我们发觉比直接在行中读的优点还少为止（至少在聚簇的情况下）。更进一步讲，创建多重索引需要有一些耗费。一方面，会增加磁盘介质的费用，尽管通常而言这不是一个很重要的费用。另外一方面是新的行插入到表中，因为插入需要耗费更多的资源来更新索引。即使在只读数据库中，在初始化的装载时还是有额外的花销，对于像prospects这样的大表这是一个不容忽视的因素。你仍然应当清楚，有很多只读表的例子，例如prospects表，表中所有的单独列都有一个相应的索引。

9.4 过滤因子和统计

就像在例9.3.3中指出的那样，在一个带有单个值的列上进行索引检索实际上并不是一件有趣的事情。事实上，在诸如SQL这样的关系查询语言中介绍的一个极其重要的而在早期数据库模型中又被忽略的特性是它的基于复合（通常是AND）谓词条件的检索信息。明显地，如果一个诸如 $C1=10$ 的谓词是确定表中的一行，考虑诸如 $C1=10$ and $C2 \text{ between } 100$ and 200 的复合谓词没有意义。如果第一个谓词确定了一行，第二个谓词仅仅排除那一行而且返回空值。在下面的内容中，我们通常考虑单个的谓词限制，例如 $C1=10$ ，导致从表中行集合的一个大的子集，同时第二个谓词 $C2 \text{ between } 100$ and 200 过滤那个集合，把对于行的检索缩小到一个很小的数量。我们说一个谓词P的一个过滤因子 $FF(P)$ 就是从谓词限制P得到的行同表的总行数的比值。我们通常通过做一个统计上的假设来估算谓词的过滤因子，包括单个列值的均匀分布和来自于任何两个不相关列的独立连接分布。我们将在后而讨论这些假设的一些变化。

例如，我们说不同的zipcode值的数目被认为有100 000个，用 $CARD(zipcode)=100 000$ 来表示。假设在表prospects中所有zipcode的值被表示为相等（均匀分布假设），我们可以估算一个等价匹配谓词的过滤因子 $zipcode=\text{const}$ ，如下：

$$FF(zipcode=\text{const}) = 1/100\ 000 = 0.00001$$

相同的假设使得我们可以估算例9.3.5的BETWEEN谓词的过滤因子：

$$FF(zipcode \text{ between } 02159 \text{ and } 03158) = 1000(1/100\ 000) = 1/100 = 0.01$$

类似地，列hobby的基数由 $CARD(hobby)=100$ 给出，也用均匀分布假设，我们得到：

$$FF(hobby = 'chess') = 1/100 = 0.01$$

两个不关联列的值的联合分布是独立的，意味着对于复合AND谓词的两个过滤因子相乘，即：

$$FF(hobby = 'chess' \text{ and zipcode between } 02159 \text{ and } 02658) = (1/100)(500/100\ 000) = 0.00005$$

一个谓词集合，每个带有一个相对没有限制的过滤因子，可能在AND联合中有很重要的效果。

例9.4.1 过滤因子计算 考虑一个可能在警察局用到的一个查询，它追踪一辆涉及到一起严重撞车事件的车：找出所有的拥有红色的1997年由Olds制造的型号为88并在Ohio注册的驾驶员：

```
select * from autos where license = 'Ohio' and color = 'red'
    and year = 1997 and make = 'Olds' and model = '88';
```

在相当合理的假设下，我们期望这个子句的谓词将对具有4千万行的表autos进行过滤，最后，留下更易于管理的几百行。这是可能的，所有汽车的1/50在Ohio注册，1/10是红色的，1/8是1997年的，1/6由Olds制造，1/8是88型的。除了相关的Olds-88对，我们假设这些属性都是独立的。例如，驾驶一辆Olds88并不排斥他同时买红色的车。我们把每一个谓词的过滤因子都乘起来以得到整个在WHERE子句的查询条件中的过滤因子：

$$FF(search_cond) = (1/50)(1/10)(1/8)(1/6)(1/8) = 1/192\ 000$$

通过Select语句检索到的行的数目近似地统计为 $(1/192\ 000)(40\ 000\ 000) = 208.33$ ，即大约为208行。一旦给定这种大小的一个解的集合，通过同车主谈话和检查车的物理损伤标记以做

进一步的调查都是合情合理的。

过滤因子的术语来自于DB2，它基于它的查询优化器对于从RUNSTATS实用程序收集的统计信息的估算。

1. DB2统计

图9-13列出了由RUNSTATS实用程序收集的统计信息，它们被用于DB2查询优化器中以进行存取方案的决策。对于每一个统计信息我们列出：(1)它出现的DB2的目录表的名字，(2)统计信息名字，它是一个列名，在它之下，统计信息出现在规定的目录表中。每一个这样的统计信息在RUNSTATS没有运行时都有一个缺省值。注意，许多索引统计信息假设几个分量的一个串接索引，我们假设它为列的组合(C1,C2,C3)。

目录名称	统计信息名称	缺省值	描述
SYSTABLES	CARD NPAGES	10 000 CEIL(1+CARD/20)	表中的行 含有行的连续数据页面的页数
SYSCOLUMNNS	COLCARD HIGH2KEY LOW2KEY	25 N.A. N.A.	在这个列中不同值的数目 在这个列中第二大的值 在这个列中第二小的值
SYSINDEXES	NLEVELS NLEAF FIRSTKEY-CARD FULLKEY-CARD CLUSTER-RATIO	0 CARD/300 25 25 如果CLUSTERED='n'则为0% 如果CLUSTERED='Y'则为95%	B树索引的层数 B树索引的叶子页面的数目 这个关键字中的第一列C1中的不同值的数目 在全部关键字中不同值的数目，全部分量。 例如，C1, C2, C3 被这些索引值聚簇的表中的行的百分数

图9-13 由RUNSTATS收集的作为存取方案决策的一些统计信息

回到图9-12中的表prospects的统计。在SYSTABLES中，在RUNSTATS执行后，我们看到表prospects的一行，如下。

SYSTABLES

NAME	CARD	NPAGES
...
prospects	50 000 000	5 000 000
...

在SYSCOLUMNNS中，在图9-12前描述的hobby和zipcode列对应的行将在NAME列上有列值'hobby'和'zipcode'。

SYSCOLUMNNS

NAME	TBNAME	COLCARD	HIGH2KEY	LOW2KEY
...
hobby	prospects	100	Wines	Bicycling
zipcode	prospects	100 000	99 998	00001
...

最后，在下面的SYSINDEXES表中，我们看到在图9-12中列出的两个索引。

SYSINDEXES

NAME	TBNAME	NLEVELS	NLEAF	FIRSTKEY CARD	FULLKEY CARD	CLUSTER RATIO
...
addrx	prospects	4	500 000	100 000	50 000 000	100
hobbyx	prospects	3	50 000	100	100	0

对于组成了addrx索引第一列 zipcode，我们假定值从00000到99999，因此 COLCARD=100 000，LOW2KEY（第二小的值）是00001，HIGH2KEY = 99998。（回忆一下，这是不准确的，只是为了简单做的假设）。对于addrx索引，FIRSTKEYCARD=100 000 并且 FULLKEYCARD = 50 000 000，因为我们在图9-12前面的段落中假设addrx键值有唯一的RID。

上述表SYSINDEXES中的最右列CLUSTERRATIO是一个量度，它衡量对于表中的行的聚簇的性质相等于一个给定索引的符合程度。最符合时为100%，通过索引检索到更多的行发生在图8-19的下面的图形上。回顾一下，在更新造成了许多行被移动或被插入到在数据页面上的聚簇顺序以外的情况下，用CLUSTER子句生成的一个索引也许不再有一个高的CLUSTERRATIO值。对于查询优化器而言这个统计信息的重要性在于：当CLUSTERRATIO值大于80%或更高时，一次索引扫描将使用顺序预取的方法通过一个索引来检索数据页面。

图9-14包含一个谓词类型和由查询优化器执行的由过滤因子计算相应公式的列表。注意，对于涉及到子查询的谓词在图9-14没有给出过滤因子的计算。事实上，涉及到非相关子查询的谓词可以用于索引检索，但是它们的过滤因子不能够由一个简单的公式来预测。我们将在本章的稍后部分讨论子查询，同时我们讨论连接。

谓词类型	过滤因子	注释
Col = const	1/COLCARD	Col <> const同not(Col=const)相同
Col < const	插补公式	“<”是任何除了等号的比较谓词，例子如下
Col < const 或者 Col <= const	$\frac{(\text{const} - \text{LOW2KEY})}{(\text{HIGH2KEY} - \text{LOW2KEY})}$	LOW2KEY和HIGH2KEY是对于Col值范围的两个端点的估计
Col between const1 and const2	$\frac{(\text{const2} - \text{const1})}{(\text{HIGH2KEY} - \text{LOW2KEY})}$	Col not between const1 and const2 与 not (Col between const1 and const2) 相同
Col in list	(List size)/COLCARD	Col not in list与not(Col in list)相同
Col is null	1/COLCARD	Col is not null与not(Col is null)相同
Col like 'pattern'	插补公式	基本字母表
Pred1 and Pred2	FF(Pred1) * FF(Pred2)	基于概率
Pred1 or Pred2	$\text{FF}(\text{Pred1}) + \text{FF}(\text{Pred2}) - \text{FF}(\text{Pred1}) * \text{FF}(\text{Pred2})$	基于概率
Not Pred1	$1 - \text{FF}(\text{Pred1})$	基于概率

图9-14 对于不同谓词类型的过滤因子公式

2. DB2中的过滤因子

一般地，先前提出的均匀分布假设不总是对的。例如，考虑一个表上的gender列，这个表含有男童学校的人员。尽管偶然有人员gender='F'，例如工作人员和教师，但是显然，以公式(CARD(gender))=1/2来计算过滤因子为1/2会引起误导，使用这个假设的查询优化器会很正常

地做出错误的决定。出于这个原因，DB2和其他很多数据库系统，例如ORACLE，提供了严重偏离均匀假设的各个列值的统计信息。然而，为了简单起见，我们不深入讨论这些统计信息，同时假设在下面的内容中均匀分布假设通常是可用的。

9.5 匹配索引扫描、复合索引

假设对于表prospects用以下语句创建了一个名为mailx的DB2索引：

```
create index mailx on prospects (zipcode, hobby, incomeclass, age);
```

这是一个我们用来有效处理通常类型的邮件请求的索引（我们一会儿就会明白这是什么意思），我们假设这不是一个聚簇索引。这里，incomeclass是一个短整型变量，定义了收入范围，含有从1到10的10个数字，它在prospects行上几乎是相等分布（这样相等匹配谓词有一个值为1/10的过滤因子），age是一个短整型变量，值有50个，从16到65。一旦给定了每列的CARD，我们就可以为这个索引计算键值的潜在的数目：

$$\text{CARD(zipcode)} \times \text{CARD(hobby)} \times \text{CARD(incomeclass)} \times \text{CARD(age)} = 100\,000 \times 100 \times 10 \times 50 = 5\,000\,000\,000$$

因为我们有50 000 000行，我们可以把从行到mailx的赋值看做是把50 000 000支飞镖放入5 000 000 000个空位中去的问题。只有极少数的空位有2支飞镖，因此，我们就可以假设唯一键值的数目(FULLKEYCARD=50 000 000)，在图8-17显示的在叶子层进行处理时没有重复键值。现在索引mailx中的项具有长度4(整型 zipcode)+8(hobby属性)+2(incomeclass)+2(age)+4(RID)=20字节。我们可以计算在每个页面上有FLOOR(4000/20)=200项。索引的叶子层有50 000 000/200=250 000叶子页面(NLEAF=250 000)。往上的上一层有1250个节点，再往上有7个，最上面是根(NLEVELS=4)。对于mailx索引在SYSINDEXES行中相关列的值在下面的表中显示。

SYSINDEXES

NAME	TBNAME	NLEVELS	NLEAF	FIRSTKEY CARD	FULLKEY CARD	CLUSTER RATIO
...
mailx	prospects	4	250 000	100 000	50 000 000	0
...

例9.5.1 串接索引，匹配索引扫描 考虑SQL查询：

```
select name, straddr from prospects
  where zipcode = 02159 and hobby = 'chess' and incomeclass = 10;
```

尽管串接索引mailx没有使我们解决在只有索引情况下的查询问题，就像在例9.3.6中那样，它仍然提供了一个重要的优点，在WHERE子句中的三个谓词zipcode=02159、hobby='chess'和incomeclass=10可以在这个简单的索引中被解决（也就是，需要的列都可以从索引中被检索到）。三个相等的匹配谓词的过滤因子每一个以1/COLCARD计算，并且给出1/100 000=0.00001(对于zipcode=02159)，1/100=0.01(对于hobby='chess')，1/10=0.1(对于incomeclass=10)。三个AND属性联合给出一个过滤因子(1/100 000)(1/100)(1/10)=(1/100 000 000)，因此我们期望有(1/100, 000 000)50 000 000=0.5行被选中。

(例如，在一半的时间返回一行，另一半时间不返回结果)。我们估算对于数据存取的I/O代价是 $0.5R$ ，执行的时间为 $0.5/80=0.006$ 秒。

索引扫描步骤通过读到B树中满足 $\text{zipcode} = 02159 \text{ and } \text{hobby} = \text{'chess'} \text{ and } \text{incomeclass} = 10$ 最左项来完成它的搜索。然后它从左到右读出项，必要时紧接着叶子的兄弟指针，直到这些值的最后项被处理。你需要确定所有要求的项(甚至会多于一个)都在mailx索引上的叶子层的一个连续扫描中。我们仅期望0.5个这种项，因此我们期望所有项都在一个单个的索引叶子页面上。假设索引的最上两层被保留在缓冲区中，对于第三层索引页面和叶子页面，我们估算索引扫描的I/O代价是 $2R$ 。■

例9.5.1的查询是在一个复合索引上执行一次匹配索引扫描。匹配意味着在WHERE子句中的谓词匹配索引中的初始属性。当我们处理在初始属性上相等匹配谓词时，在索引中所选的RID值被找出的部分是整个索引叶子层的一个连续的子区间。类似地，假设在New York的电话目录上，电话用户通过姓，名，区或城市，最后是街道以字母顺序排列。这样，对例9.5.1的查询就像通过姓和名来查询电话号码，而不必知道街、区或者城市。

例9.5.2 串接索引，匹配索引扫描 考虑SQL查询：

```
select name, straddr from prospects
  where zipcode between 02159 and 04158
    and hobby = 'chess' and incomeclass = 10;
```

再一次说明，在WHERE子句中的所有三个谓词 $\text{zipcode between } 02159 \text{ and } 04158$ 、 $\text{hobby} = \text{'chess'}$ 和 $\text{incomeclass} = 10$ 可以在索引mailx中被解决。通过使用一个插补公式(参见图9-14的对于BETWEEN谓词的过滤因子)，我们计算对于谓词 $\text{zipcode between } 02159 \text{ and } 04158$ 的过滤因子：

$$(04158 - 02159)/(HIGH2KEY - LOW2KEY) = 1999/99,998$$

大约为 $2/100=0.02$ 。另外两个谓词就像以前一样用 $1/\text{COLCARD}$ 来计算，并且给出 $1/100=0.01$ (对于 $\text{hobby} = \text{'chess'}$)和 $1/10=0.1$ (对于 $\text{incomeclass} = 10$)。三个AND属性一同给出过滤因子 $(2/100)(1/100)(1/10)=(1/50\ 000)=0.00002$ 。这样平均会有 $(2/100\ 000)50\ 000\ 000=1000$ 行会被选中。因为索引不是聚簇的，连续的行会在磁盘上分得很远，对于数据存取的I/O代价为 $1000R$ ，需要执行时间为 $1000/80=12.5$ 秒。(列表预取不能在这里使用的原因将在例9.6.5中说明，基于没有提供具体化的数据。)

索引扫描步骤通过读到B树中满足 $\text{zipcode} = 02159 \text{ and } \text{hobby} = \text{'chess'} \text{ and } \text{incomeclass} = 10$ 最左边项来执行它的搜索，然后在叶子层从左到右跟着兄弟指针读出，直到最后一个满足WHERE子句条件的项被处理。然而，在这种情况下，要求所有的项在mailx索引的叶子层的一个连续扫描中是不正确的。存在带有其他hobby值的项，例如，插入在 $\text{zipcode} = 02159 \text{ and } \text{hobby} = \text{'chess'}$ 和 $\text{zipcode} = 02160 \text{ and } \text{hobby} = \text{'chess'}$ 之间的索引项中。这也是一个索引匹配扫描，但是从串接索引匹配的唯一的索引分量是 zipcode 的范围 $\text{zipcode between } 02159 \text{ and } 04158$ 。这意味着我们需要读出mailx索引的叶子层中的 $2000/100\ 000=1/50$ ，即 $(1/50)(250\ 000)=5000$ 页。我们可以用顺序预取以5000S的代价在 $5000/800=6.25$ 秒的时间来做这件事情。对于该查询，整个的执行时间为 6.25 秒(用于索引扫描)+ 12.5 秒(用于数据页检索)= 18.75 秒。■

例9.5.2用我们先前用过的类似的方法在New York电话目录中，查询每一个姓以“Sm”开头、名以“John”开头的人。有大量的项被扫描以检索电话用户的集合。但是情况会变得更坏。如果在WHERE子句中没有包含索引的初始属性，我们就需要考虑一个非匹配索引扫描。例子如下，我们要从New York电话目录中找出每一个名为“John”住在Bronx区的人，而不知道他的姓，这是一个困难的任务。我们不能使用目录的字母顺序来得到任何效果，因为我们根本不知道在哪一个的姓下名会显示出“John”，这样我们不得不看完整个目录。仍然，对于这样一个查询，使用电话目录搜索（相当于一个非匹配索引扫描）比走出门挨家挨户去找一个拥有电话并且名字为“John”的人要好（一个表扫描，类比的扩展）。

例9.5.3 串接索引，非匹配索引扫描 再一次，我们假设mailx索引已经存在。考虑SQL查询：

```
select name, straddr from prospects
  where hobby = 'chess' and incomeclass = 10 and age = 40;
```

回顾一下，hobby列的基数是100，incomeclass列是10，age列是50，因此我们计算复合选择谓词的过滤因子为 $(1/100)(1/10)(1/50) = (1/50\ 000) = 0.00002$ 。这同我们在例9.5.2中计算的过滤因子是相同的，我们仍然需要以1000R的I/O代价从表prospects中检索1000行，总共12.5秒的时间。但是索引扫描的情况完全不同。因为这是一个非匹配索引扫描，我们需要读入250 000个mailx叶子页面的每一个页面，I/O代价就是250 000s，执行时间为 $250\ 000/800 = 312.5$ 秒，大于5分钟。 ■

1. 匹配索引扫描的定义

我们在研究单列索引时已经谈到可索引谓词的思想，现在正是给出执行一次匹配索引扫描的意义的详细描述的时候。

定义9.5.1 一次匹配索引扫描就是把许多谓词匹配到单个索引的列分量上以后检索来自于一个表的行。至少一个可索引的谓词必须引用索引的初始列，这被称为匹配谓词。可索引的谓词的集合可以匹配复合索引的一系列初始列，它们都被称为匹配谓词。 ■

例如，考虑在表T上的索引C1234X，它是一个在列(C1, C2, C3, C4)上的复合索引。下面的复合谓词匹配所有C1234X索引上的列：

C1 = 10 and C2 = 5 and C3 = 20 and C4 = 25

复合谓词

C2 = 5 and C3 = 20 and C1 = 10

匹配C1234X的前三列（注意谓词并不需要列具有相同的顺序）。这与我们在例9.5.1中见到的匹配索引扫描类似，都是在四列复合索引mailx的前三列有三个AND相等匹配谓词。
复合谓词

C2 = 5 and C5 = 22 and C1 = 10 and C6 = 35

在C1234X的前2列有两个匹配谓词。列C5和C6不是索引的一部分，因此不能被匹配。复合谓词

C2 = 5 and C3 = 20 and C4 = 25

不是一个匹配索引扫描，因为它没有匹配谓词。这类似于我们在例9.5.3中见到的情况，在一个四列的复合索引的第二、三、四列是三个以AND连接的相等匹配谓词。

注意，在例9.5.3中没有匹配谓词并不意味着非匹配谓词的过滤因子无效。就像我们在例

9.5.3中见到的，谓词依然具有把50 000 000行过滤到1000行答案集的作用。因为这是一个非匹配索引扫描，然而，整个索引需要检查以执行这个过滤。作为对比，在一个匹配索引扫描中，我们通常以在需要考虑的索引中的一个小的连续范围的叶子层项来结束。(就像我们将看到的，当涉及到IN-LIST谓词时有一个例外，但是这是很特殊的。)

2. 谓词筛选和筛选谓词

当DB2使用非匹配谓词的过滤因子时，就像在例9.5.3中一样，在它存取任何数据行以前，它遍历了大量索引的叶子层项，同时排除了不符合谓词的项。这件工作被称为是谓词筛选，涉及到的非匹配谓词称为筛选谓词。在谓词筛选过程中，在没有涉及到匹配谓词但仍对应被计算的谓词的串接索引中的列值被用来确定保留或删除项。所有对应于项而在筛选后被保留的行是将要被检索的候选行，但是在这类筛选发生以后这样的行的数目通常会变得很小。

就像我们在例9.5.3中看到的那样，谓词筛选的应用可以导致执行磁盘I/O读索引项的时间远大于把行读入答案集的时间的情况。筛选谓词和匹配谓词的不同点是一个很重要的因素，我们将会看到，在索引存取中有谓词筛选不是由DB2执行的情况。然而，匹配谓词总是被使用。

匹配索引扫描的整个概念是相当复杂的，因此现在让我们回顾我们已经介绍过的内容。假设在列(C1, C2, C3, C4)上有一个复合索引C1234X，还有一个涉及到谓词P1, P2, …, Pk的一个复合谓词。

定义9.5.2 匹配谓词的基本规则

1) 一个匹配谓词必须是一个可索引的谓词。

图9-15提供了一个可索引谓词的列表：相等匹配谓词和比较谓词、BETWEEN谓词、LIKE谓词、IN-LIST谓词、IS NULL谓词都是可索引的。通常，把NOT逻辑运算符附加到一个可索引的谓词上会给出一个非可索引的结果。

2) 匹配谓词必须匹配一个索引的连续列C1, C2, …。

这是一个确定它们的过程。从左到右看索引列。对于每一列，如果在该列上至少有一个可索引的谓词，我们已经找到了一个匹配列和一个匹配谓词。如果对于一个列没有找到一个匹配谓词就结束过程。(还有其他规则，将在定义9.5.4中说明。)

3) 在一个索引中一个列上的一个非匹配谓词可以仍然是一个筛选谓词。

回答一个查询的一个索引上可能存在一个匹配索引扫描，同时在一个带有更好的更有效的过滤因子的一个不同索引上可能存在一个非匹配索引扫描。查询优化器必须考虑所有的到达存取方案的可能性。 ■

在一个索引(例如C123X)开始的K列上的一个匹配索引扫描的情况下，EXPLAIN命令创建方案表的一个行，其中，ACCESSTYPE=I, ACCESSNAME=C1234X, MATHCHCOLS=K。尤其是对应一个非匹配索引扫描，我们可以见到MATHCOLS=0。

从定义9.5.2的第一点，我们看到匹配谓词必须是一个可索引的谓词。事实上，术语“可索引的谓词”恰恰就是指这个意思。

定义9.5.3 一个可索引的谓词被定义为一个可以用来在一次匹配索引扫描中匹配一列的谓词，现在就是图9-15的谓词集合。 ■

谓词类型	可索引	注释
Col ∞ const	Y	∞ 代表 $>$, \geq , $=$, \leq , $<$ 但是 $\not\in$ 不是可索引的
Col between const1 and const2	Y	在匹配系列中必须是最后的(参见匹配规则)
Col in list	Y	仅对一个匹配列(参见匹配规则)
Col is null	Y	
Col like 'pattern'	Y	在Pattern中没有先导的%
Col like '%xyz'	N	带有先导的%
Col1 ∞ Col2	N	col1和col2来自同一个表
Col ∞ Expression	N	例如, C1(C1+2)/2
Pred1 and Pred2	Y	Pred1和Pred2都是可索引的, 指相同索引的列
Pred1 or Pred2	N	除了(C1=3或C1=5)以外, 它可以被认为C1在(3,5)中
Not Pred1	N	或者任何等价式: not between, not in list notlike pattern等

图9-15 单个表上的可索引谓词

术语“可索引的谓词”令人相当迷惑，因为它看上去似乎隐含着以下含义，非可索引的谓词就不能够用在一个过滤被检索行的索引中。但是这是不正确的：非可索引的谓词仍然可以用于筛选一次索引扫描中的谓词——它们仅仅不能用于匹配，就像我们在定义9.5.1中正式定义匹配谓词以后我们详细描述的那样。替换“可索引的谓词”的一个更好的术语是“可匹配谓词”，但是不幸的是，在这一点上，更早的术语已经被牢牢地嵌在数据库的词汇中了。

在一个单列索引的情况下，功能是筛选谓词的非可索引的谓词是很少见的，但不是不可能的。这种可能性依赖于过滤因子是否使得查询优化器认为它是有价值的。因为一个普通的非可索引的谓词是Col $\not\in$ const，带有一个过滤因子(1 - 1/COLCARD)，该过滤因子接近于1，并且对于减少检索到的行数以弥补读取很大部分的索引方面似乎没有代价上的节省。

当我们面对不是相等匹配谓词的可索引谓词时，在匹配列方面事情变得更为复杂。我们重复定义9.5.2中的第二点的规定，同时加入一些其他的规则。

定义9.5.4 匹配谓词的高级规则

- 1) 从左到右看串接索引的索引列。对于每一列，如果对于该列至少找到了一个可索引的谓词，它就是一个带有匹配谓词的匹配列。
- 2) 如果对于一列没有找到匹配谓词就结束搜索。然而，由于一些原因搜索也许会更早结束。
- 3) 当对一列使用了匹配范围谓词（比较形式如 $<$, \leq , $>$, \geq , LIKE谓词或者BETWEEN谓词）时，搜索在那一点结束。
- 4) 在一个匹配谓词集合中至多可以用一个IN-LIST谓词。一个IN-LIST中的第二个匹配列将在涉及到的列作为匹配列的一部分之前导致搜索结束。 ■

如果我们假设匹配索引扫描的思想是用一个叶子层的索引项的连续范围来作为结束，很

容易看到，为什么一个范围谓词结束了匹配谓词的搜索。回顾一下，在例9.5.2中一个查询的三个AND谓词引用mailx索引的头三行。然而，在第一列上的谓词 zipcode between 02159 and 04158是一个范围谓词，这样，剩下的谓词就不是匹配索引扫描的一部分。原因在于，一旦我们限制了范围谓词的索引扫描，满足剩下两个谓词的项就会出现在2000个分离的区间上，它们对应于2000个zipcode的值（分别为02159, 02160, 02161, …）的每一个并且满足hobby='chess' and incomeclass=10。对于索引扫描而言，扫描所有遵守范围谓词和把其余的两个谓词作为筛选谓词的所有项是最简单的。这样我们有一个相对较大的索引扫描要执行，即使用顺序预取也需要12.5秒的耗时，当用随机I/O时，可以把它比作一个相对较小的行使用随机I/O时耗时为25秒的I/O代价。

IN-LIST谓词是唯一一个打破要求叶子层索引项的最后范围是连续的这一规则的谓词。下面给出C1234X索引和复合谓词。

[9.5.1] C1 in (6, 8, 10) and C2 = 5 and C3 = 20

我们在所有三列上都有一个匹配谓词，尽管我们不以一个叶子层项的单个连续范围来结束。相反，有三个连续的范围，一个对应于第一个列选择C1=6，一个对应于C1=8，一个对应于C1=10。你可以以这种方式描绘出一次带有一个IN-LIST谓词的匹配索引扫描，就像带有相等的匹配谓词来代替IN-LIST的一系列扫描一样。但是DB2会因为缺少允许第二个IN-LIST谓词进入匹配扫描而停止。这样复合谓词：

[9.5.2] C1 in (6, 8, 10) and C2 = 5 and C3 in (20, 30, 40) and C4 = 25

将只有两个匹配谓词。在第二个C3的IN-LIST谓词出现之前匹配停止。当一个IN-LIST谓词用在一个匹配索引扫描中，方案表的行含有一个特殊的值ACCESTYPE=N。我们对于上述的第一个这样的复合谓词有MATCHCOLS=3，同时对于第二个则有MATCHCOLS=2。

例9.5.4 查询优化和复合索引扫描 假设我们给定一个带有列C1, C2, …的表和在(C1, C2, C3, C4)上的索引C1234X；在(C5, C6)上的索引C56X；还有一个在键列C7上的唯一索引C7X。考虑下面的查询：

1) select C1, C5, C8 from T where C1 = 5 and C2 = 7 and C3 < 9;

这个结果是在两个列C1和C2上的一个匹配索引扫描。谓词C3是不可索引的（但是它可以被用为筛选谓词）。在方案表中，我看到ACCESTYPE=1, ACCESSNAME=C1234X, MATCHCOLS=2。

2) select C1, C5, C8 from T where C1 = 5 and C2 >= 7 and C3 = 9;

我们可以看到一个在两列C1和C2上的匹配扫描。尽管第三个谓词是可索引的我们也停止，因为在C2上的谓词是一个范围谓词。方案表同1)相同。

3) select C1, C5, C8 from T where C1 = 5 and C2 = 7 and C5 = 8 and C6 = 13;

这是一个我们还未见过的多重索引使用的一种类型，其中我们可以把从多于一个索引的谓词的过滤因子组合起来，这种方法将在这里使用。如果那种替代方法不存在，查询优化器将考虑在以下两种方案之间进行选择：一个是在C1234X上两列使用匹配索引扫描，一个是在C56X上两列使用一个匹配索引扫描。我们从方案表中将知道发生了什么，ACCESTYPE=1, ACCESSNAME=C56X, MATCHCOLS=2。

4) select C1, C4 from T where C1 = 10 and C2 in (5, 6) and (C3 = 10 or C4 = 11);

这是在前面两列的一个匹配索引扫描。ACCESSTYPE=N（因为有IN-LIST谓词），ACCESSNAME=C1234X，还有MATCHCOLS=2。第三个谓词(C3=10 or C4=11)不是可索引的，但是在扫描中它将被用作为一个筛选谓词。（我们没有见到在方案表中涉及的筛选谓词，但是所有查询的谓词必须被用来过滤，涉及到被选择索引的列的谓词当然被用作筛选。）扫描也有INDEXONLY=Y，在确定代价时它是一个很重要的因素。

5) select C1, C5, C8 from T where C1 = 5 and C2 = 7 and C7 = 10;

因为C7X索引是唯一的，查询优化器将选择ACCESSTYPE=I，ACCESSNAME=C7X。方案表没有揭示这种查询是唯一的（返回0行或1行）。

6) select C1, C5, C8 from T where C2 = 7 and C3 = 10 and C4 = 12 and C5 = 15;

这个查询可以被处理，或者通过在C1234X上的一个非匹配索引扫描，列C2, C3, C4，或者通过在C56X上的一个匹配索引扫描，列C5。（为了简单起见，在上述3)中涉及的多重索引使用不是一个可替代的。）我们也许可以在方案表中看到下面的结果：ACCESSTYPE=I，ACCESSNAME=C1234X, MATCHCOLS=0。

3. 可索引谓词及其性能

模式匹配搜索 一个模式匹配搜索`C1 like 'pattern'`，在模式中有一个前导的统配符“%”，可以比作一个串接索引的一次非匹配扫描。谓词可以有一个小的过滤因子，但是这种搜索同使用一本普通的字典来查找所有以“action”字符串结尾的所有单词是类似的。在列C1上的一个索引必须被完全扫描以检索指向适当行的RID的集合。存在特殊的字典，字母表是倒排的，如果一个DBA发现他的工作负荷中有许多带有前导%统配符的模式匹配搜索，那么他应当考虑创建带有倒排拼写的索引列。这样如果列C2被创建以包含列C1倒排的文本，那么在C1上的规定%action成为在C2上no itca%，一个更简单的搜索。

表达式 非索引谓词，`Col < Expression`（图9-15给出），仅仅是一类非可索引的谓词的一个例子。基本上，任何涉及到一个表达式的比较都是非可索引的。例如，考虑如下查询：

```
select * from T where 2 * C1 <= 56;
```

查询优化器不能够使用一个索引来解决这个谓词。然而，你可以重新诠释这个谓词为：（两边都除以2），现在就可以使用索引了。

```
select * from T where C1 <= 28;
```

取一次访问 一定类别的查询在DB2中尤其有效，它提供了所谓的取一次（索引）访问，在方案表中ACCESSTYPE=I1。这样一个查询的一个例子如下：

```
select min(C1) from T;
```

其中一个索引同开始的列C1共存。很明显在这种情况下查询优化器可以简单地搜索到索引的叶子层的最左边的项并且检索C1值，这就是为什么以“取一次存取”命名的原因。在更为通常的情况下使用这个原则也是可能的。例如，下面的查询就可以用一个取一次存取的步骤来回答：

```
select min(C1) from T where C1 > 5;
```

（注意`C1>5`并不必然意味着对于C1的最小值是6。在这里使用索引是很重要的。）另外一个带有取一次存取的例子如下：

```

select min(C1) from T where C1 between 6 and 10;
select max(C2) from T where C1 = 5;
select max(C2) from T where C1 = 5 and C2 < 30;
select min(C3) from T where C1 = 6 and C2 = 20 and C3 between 9 and 14;

```

每一个连续的匹配等价的匹配谓词减小了对于索引键值的范围。在上述的最后一个例子中，DB2一直走到C12D3X索引来找出第一个项键值 $\geq 6.20.9$ 。

9.6 多重索引存取

假设下面的索引是定义在一个表T上的唯一一个索引：C1X在(C1)上，C2X在(C2)上，C345X在(C3, C4, C5)上。现在考虑下面的查询：

[9.6.1] select * from T where C1 = 20 and C2 = 5 and C3 = 11;

应用到现在为止我们已经学到的匹配索引扫描，查询优化器将选择三个索引中的一个，索引中的每一个仅仅匹配查询[9.6.1]中的三个谓词中的一个。这样，我们将从仅使用三个谓词过滤因子中的一个就可以在检索数据行之前抽取出RID值当中获益，同时查询方案必须检测剩余两个谓词的真实性以限制检索到的行。

但是，这是一个庞大的无效率的过程。如果我们假设谓词中的每一个有一个值为1/100的过滤因子并且表T含有10亿行，那么单个谓词仅仅把检索到的行的数目减少到1百万。三个谓词联合起来就有一个组合的过滤因子1/1 000 000，把检索到的行数减少到100行。所以，我们问是否在检索被选择的行之前有一种把匹配不同索引的谓词的过滤因子组合起来的方法。

事实上，有一种方法可以做到这一点，它提供我们多重步骤方案的第一个例子。基本上，这种方法从每一个索引抽取出满足匹配谓词的RID列表，然后，对于不同的索引的RID列表相交（AND），因此最后的RID列表对应于满足所有被索引谓词的行。来源于DB2方案表的一系列步骤也许就是图9-16中显示的方案，其中，方案表是查询[9.6.1]的一个EXPLAIN的结果。

TNAME	ACCESTYPE	MATCHCOLS	ACCESSNAME	PREFETCH	MIXOPSEQ
T	M	0		L	0
T	MX	1	C1X	S	1
T	MX	1	C2X	S	2
T	MX	1	C345X	S	3
T	MI	0			4
T	MI	0			5

图9-16 对于查询[9.6.1]的一个多重索引存取方案的方案表的行

- MIXOPSEQ=0 这个带有ACCESTYPE=M的方案行表明多重索引存取过程将要在表TNAME=T上开始。PREFETCH=L意味着最后被计算的RID列表产生以后列表预取I/O（我们在9.2节末尾讲到过）被用来检索表T中的行。
- MIXOPSEQ=1 这个带有ACCESTYPE=MX的方案行表明，满足查询的匹配谓词的带有ACCESSNAME=C1X的索引的项将被扫描，使用顺序预取I/O。在这样的情况下，MATCHCOLS=1，同时在查询[9.6.1]中的匹配谓词为C1=20。因为遇到C1X的项，RID被抽取出来并放入一个被称为是RID池的内存区域中称为RID候选列表的地方。在所有RID被从这个C1X存取步骤抽取出来之后的某一点上，RID候选列表以有序的形式放置使得以后的相交操作步骤容易执行。

- **MIXOPSEQ=2** 和 **MIXOPSEQ=3** 这些步骤在索引 C2X 和 C345X 上执行与在 MIXOPSEQ=1 中相同的 MX 函数，对于这些索引的匹配谓词产生它们自己的 RID 候选列表。我们可以把连续产生的 RID 候选列表看做被推入到一个栈中，就像在一个逆波兰计算器中一样：最近产生的列表接近于栈的顶部，并且由后面的计算器运算首先执行。
- **MIXOPSEQ=4** ACESSTYPE=MI 表明一个 RID 候选列表相交操作 (AND) 将发生。两个最近产生的 RID 列表从栈顶弹出（先是从 C345X 产生的列表，然后是从 C2X 产生的列表）；它们相交以提供一个新的 RID 列表（一个 DB2 命名为 IR1 的中间结果），同时这个列表被压入到栈中。因为两个列表的 RID 都以排序的形式存在，相交操作可以通过建立两个指向在每个列表中初始的 RID 的游标很容易地执行，然后使指向更低值的 RID 的游标不断前进。任何一个匹配状态出现，我们就发现了一个相交的元素。我们把这个 RID 值放入到 IR1 列表中，然后继续把两个游标中的一个往前继续移动以寻找下一个相交的元素。正在前进的游标的一个离开了列表的底部时，过程就结束了。
- **MIXOPSEQ=5** 最后一个步骤也有 ACESSTYPE=MI，这个步骤把栈顶的两个 RID 列表从栈中弹出，即 IR1 和从 C1X 产生的列表。列表被相交以形成一个名为 IR2 的新的 RID 列表，它被压入到栈中，这是产生的最终的 RID 列表。最终的列表被用来检索表 T 中的行，使用列表预取 I/O，就像在方案的初始 M 步提到的那样。

还有一种用于多重索引存取的存取步骤。带有 ACESSTYPE=MU 的一行表明一个将弹出在栈中的两个 RID 列表的步骤的行执行一个两个列表的 RID 候选列表的并 (OR 操作)，并且把新的中间结果压入栈中。例如，用在上面看到的相同的表和索引假设，应用于 [9.6.2] 给出的查询的 EXPLAIN 命令会导致图 9-17 所示的方案表的行。

[9.6.2] select * from T where C1 = 20 and (C2 = 5 or C3 = 11);

TNAME	ACESSTYPE	MATCHCOLS	ACCESSNAME	PREFETCH	MIXOPSEQ
T	M	0		L	0
T	MX	1	C1X	S	1
T	MX	1	C2X	S	2
T	MX	1	C345X	S	3
T	MU	0			4
T	MI	0			5

图 9-17 对于查询 [9.6.2] 的一个多重索引存取方案的方案表的行

图 9-17 与图 9-16 的不同点仅在于 MIXOPSEQ=4 的那一行，其中执行一个 MU 步骤来对谓词 C2=5 和 C3=11 产生的 RID 列表做并 (OR) 运算以代替相交 (AND) 运算。在最后一步 MIXOPSEQ=5 中，这个并同 C1=20 产生的列表作相交操作 (AND)。

注意，在图 9-17 和图 9-16 中，多重索引抽取入 RID 列表的连续步骤被产生以跟随查询中谓词的物理顺序。自然计算的顺序同查询的语法是独立的。实际上，由查询优化器产生的多重索引存取步骤是以一种更有效地使用 RID 池的顺序进行的，通常这意味着在任何时候，总有最小数目的 RID 列表存在；也就是，组合 RID 列表的操作应尽早执行以减小内存使用。对于查询 [9.6.2] 它的含义是把图 9-17 中的行重排为如下方案。

TNAME	ACCESSTYPE	MATCHCOLS	ACCESSNAME	PREFETCH	MIXOPSEQ
T	M	0		L	0
T	MX	1	C2X	S	1
T	MX	1	C345X	S	2
T	MU	0			3
T	MX	1	C1X	S	4
T	MI	0			5

这个新的方案同在图9-17中的方案具有相同的效果，但是它具有不会同时有多于两个RID列表出现的性质。

例9.6.1 多重索引存取 考虑表prospects和带有在图9-12中列出的统计信息的hobbyx和addrx索引，同时假设可以在prospects上得到的其他索引只有在列age上的agex索引和在列incomeclass上的incomex索引。一个小想法会使你信服，这些索引同hobbyx索引有几乎相同的统计信息，这是因为在叶子层有如此多的索引压缩以至于在4字节(NLEAF也是相同的)的多重RID项控制的项的长度中索引键的长度是多余的；在键长度更加相关的更高层，没有足够的变化去改变SYSINDEXES中的NLEVELS统计信息(参见图9-13)。现在考虑我们在例9.5.1中处理的查询：

```
select name, straddr from prospects
  where zipcode = 02159 and hobby = 'chess' and incomeclass = 10;
```

在那个例子中，我们有一个串接索引mailx，在其上一次索引扫描被执行。依据上面提到的单个索引假设，这个查询可以用图9-18中描述的多重索引存取方案来执行。

TNAME	ACCESSTYPE	MATCHCOLS	ACCESSNAME	PREFETCH	MIXOPSEQ
prospects	M	0		L	0
prospects	MX	1	hobbyx	S	1
prospects	MX	1	addrx	S	2
prospects	MI	0			3
prospects	MX	1	incomex	S	4
prospects	MI	0			5

图9-18 多重索引存取方案的方案表的行(在例9.6.1中说明)

让我们计算MIXOPSEQ=1的步骤的I/O代价。这个步骤扫描对于谓词hobby='chess'的hobbyx索引。因为FF(hobby='chess')=1/100，对于hobbyx NLEAF为50 000，这需要一次50 000叶子层页(忽略在目录间的I/O)的1/100的扫描，即500S的I/O代价。这些叶子层项的RID列表被抽取出来，并压入栈中。

对于MIXOPSEQ=2，我们扫描addrx索引来解决谓词zipcode=02159并且抽取出RID值放入一个列表中。因为FF(zipcode=02159)=1/100 000，同时对于addrx NLEAF=500 000，从左到右以解决这个谓词的扫描的叶子页面数是(1/100 000)(500 000)=5S。对于MIXOPSEQ=2，谓词incomeclass=10有一个过滤因子1/10，这样在50 000个叶子层页面

情况下抽取RID的I/O代价为5000S。

现在，带有MIXOPSEQ=3和5的ACCESSTYPE=MI列表相交步骤并不要求I/O，因为所有的RID已经驻留在内存中。就像我们在例9.5.1中做的那样，把三个谓词的过滤因子乘起来，我们可以再一次看到我们期望从表中仅检索到0.5行。当然，这只是一个概率上的估算。

然而，许多行被检索到了，它们很可能在不同的磁盘页面上，系统用列表预取来检索它们（就像我们在MIXOPSEQ=0看到的其中PREFETCH=L）。通过列表预取对于检索0.5页的I/O代价由0.5L来指明。就像我们在图9-10中指明的I/O速度的经验规则，以每秒200次I/O速度进行列表预取。一个0.5页的列表预取低于这条经验规则近似值的低限（4个页面），但是在任何情况下涉及到获得一个期望平行的时间是不重要的。

对于查询的整个的基于I/O代价的执行时间计算如下： $500S + 5S + 5000S + 0.5R$ ，即 $5505/800 + 0.5/80$ ，即大约为6.9秒。

例9.6.2 多重索引存取 在例9.6.1的索引假设下，我们检查前面的例9.5.2中的查询：

```
select name, straddr from prospects
  where zipcode between 02159 and 04158
    and hobby = 'chess' and incomeclass = 10;
```

对于这个查询，zipcode上的BETWEEN谓词引起一次addrx上的匹配索引扫描，它带有过滤因子($2000/100\ 000=1/50$)。因为对于addrx而言NLEAF=500 000，对于这个谓词抽出RID的扫描需要代价10 000S。解决hobby和incomeclass上的谓词的代价同例9.6.1没有改变，为500S和5000S。

被这个查询检索到的总的行数以例9.5.2中的相同方式被计算： $(1/50)(1/100)(1/10)(50\ 000\ 000)=1000$ 行，很有可能在分离的页上。然而，因为排序的RID列表被相交以达到这个行的序列，RID是有序的。这样列表预取将节省1000个磁盘页面的I/O代价。这个查询的整个I/O代价由 $10\ 000S + 500S + 5000S + 1000L$ 给出，执行的时间是 $15\ 500/800 + 1000/200 = 19.4 + 5 = 24.5$ 秒。这个结果可以同我们在例9.5.2得出的结果进行比较，其中，我们假定1000个数据页面I/O用随机I/O(1000R)来检索，所以用了12.5秒，而不是列表预取I/O(1000L)，列表预取I/O耗时5秒。 ■

1. 列表预取和RID池

如果列表预取比随机I/O更有效，为什么我们要执行随机I/O？答案就像我们在9.2节末尾提示的那样，在列表预取在DB2中发生以前需要有相当特殊的前提条件，条件出现在“RID列表使用规则”中。然而，在继续讲述以前，为防止可能的误解而做一个申明是很恰当的。

直到现在，我们已经介绍的大多数查询存取原则已经是相当通用的原则了，尽管我们的例子是关于DB2特性的，我们期望现在或在不久的将来在大多数关系数据库中看到等价的特性。例如，在教育界或工业界你有可能碰到的所有关系数据库都有查询优化器，它们利用数据统计信息计算和比较过滤因子，并产生包含诸如表扫描和各种类型的索引辅助扫描的查询存取方案。在复合索引上的匹配扫描的思想和可以被匹配或用于筛选的特定类型的谓词在大多数非DB2产品上通常以一种不是很复杂的方式完成。（匹配扫描实际上是DB2特定的命名法。）对于多重索引存取特性（通过位图索引有时会更有效）和对于RID列表驻留内存（或者最后的结果的位图）也都是相同的。这种思想在许多（还不是所有）关系产品上实现，但是通常灵活性少一些。本部分的主要内容，列表预取及其对于RID列表规则的依赖性涉及到从通

用规则到产品规定的标准等方面。这些概念对于正确理解DB2查询优化是很重要的。但是你应当警觉它们不是基本的；事实上，许多规则有些武断，并且其他数据库产品很有可能选择一个不同的方法（也许是更高超的！）来确定什么样的RID列表（或者位图）在内存中具体化。

回到当前的主题，是什么限制了在DB2中列表预取的使用？规则就是列表预取仅在从数据页面检索行时当索引允许查询估测器预测需要被存取的行时被执行。对于可能的列表预取，被检索到的行的RID必须已经被从一个扫描索引中抽取出来并且放入一个被称为RID池的内存存储区域然后按照页号的升序来排序（意味着RID按照升序）。列表预取机制需要这样一个RID列表使得它可以要求磁盘控制器检索块，一次点共32页，通过在一个磁盘的小区域中的最有效的预先决定的运动。正如我们在例9.6.1中论述的，我们假设对于列表预取的一个经验规则是每秒200次I/O，尽管这个速度实际上依赖于取出页面的近似值。

列表预取总是被用来存取多重索引存取中的数据页面（有时我们简称为MX存取），因为排序的RID列表必须在方案期间存在。在复合索引扫描中，例如在例9.5.2的扫描中1000行被检索，似乎我们倾向于使用列表预取来加速数据页面的检索。选择列表预取中的一个限制因素通常是在RID池中的空间问题，因为每一个RID的长度是4字节，1000个RID的一个列表将占用4000字节，大约为内存缓冲区中一个磁盘页面的大小。但是对于RID的使用的其他限制将在定义9.6.1中讨论。

RID池内存区域的尺寸基于由DBA选择的缓冲池的大小（在DB2中实际上有多个缓冲池，为了简单，我们仅指一个缓冲池，并假设其他的不用）。RID池的缺省尺寸是缓冲池大小的一半，除了在版本2中它不能超过200MB。这个限制在当前的版本5已经上升到1000MB，但是我们仍然使用老版本的值来对应于在以后章节出现的基准测试数据。（注意，尺寸的讨论并不意味着RID池是缓冲池的一部分；两个缓冲池是不同的内存区域。）对于RID池实际上只是在需要时才被分配，并且以16MB的大小递增直到系统设置的限制。RID池并发地被许多不同的执行查询的进程使用，它相当节俭地少量分配。通过下面一些限制性规则来做出每一种努力以使池空间的使用最小。

定义9.6.1 RID列表使用的规则 下面一些规则通过查询优化器控制RID列表的使用。

1) 当查询优化器建立了一个对于涉及到RID列表产生的查询方案（被称为绑定时间）时，在方案中任何时间活跃的可预测的RID数目不能多于RID池空间的50%。如果的确需要，那么会产生一个替代的方案，它不要求RID列表的建立。在RID列表抽取已经开始以后，如果RID的使用被证实是错误的，以致于RID列表的产生是不合适的，该方案被中止，并且用另外一个存取方法来回答这一查询。

2) 在一个抽取RID列表的一个索引扫描中不能使用筛选谓词。

3) 在一个抽取RID列表的一个索引扫描中不能使用一个IN-LIST谓词。

上述规则对于DB2的2.3版本都是正确的，与本章中后面引用的性能数据相对应。在DB2的版本5中，规则2和3是在文档中规定的，而1却没有，也许是因为它已作为内部的考虑了。 ■

例9.6.3 RID列表的大小限制 再一次考虑带有例9.6.1索引（即addrx, hobbyx, incomex）的表prospects。我们在prospects的一个名为gender的列上（它的值有'M'和'F'两个），添加一个新的索引genderx，其中，我们假设两个列值以等频率出现。考虑如下查询：

```
select name, straddr from prospects
  where zipcode between 02159 and 04158
```

```
and incomeclass = 10 and gender = 'F';
```

因为索引压缩，我们看到genderx索引的叶子层必须包含大约50 000个页面，与hobbyx索引的数目相同。为了回顾这个推理，在表prospects中有50 000 000行，并且对于RID平均的叶子层项（因为重复关键字值的压缩）占用4字节。所以在一个叶子页面上有1000个项，5千万项需要50 000叶子页面。现在考虑我们将从索引genderx抽取出的RID列表的大小以满足匹配谓词，其中条件为 $FF(gender = 'F') = 1/2$ 。这个扫描将遍历50 000页叶子层的一半，并且抽出所有RID，一个25 000页的RID列表，这是因为在叶子层上的几乎所有空间都是由RID构成的。但是根据定义9.6.1的规则1，我们将不能够建立一个25 000页的RID列表，除非我们有一个50 000页的RID池，即200MB，一个绝对的限制（在DB2版本2.3中）。假设在相交的同时从已有的方案中必须有另外一个RID列表，我们看到谓词 $gender = 'F'$ 也许不能使它的RID列表被抽取出。因为我们假设 $gender = 'F'$ 对于表prospects的5千万行的一半抽出RID，很明显对于这样一个谓词我们不能有一个MX步骤。多重索引方案必须凑合着用其他两个谓词和它们的过滤因子，同时从那里开始继续检索数据行。 ■

即使在简单的匹配索引扫描中，列表预取存取不总是一个被放弃的结论。

例9.6.4 RID列表大小的限制 再一次考虑在例9.3.4中的带有索引hobbyx的表prospects和评价的查询：

```
select name, straddr from prospects where hobby = 'chess';
```

因为 $FF(hobby = 'chess') = 1/100$ ，例9.3.4的主要I/O代价是从表prospects中以非聚簇顺序检索 $(1/100)(50 000 000) = 500 000$ 行。500 000R的时间代价是 $500 000/80 = 6250$ 秒，即大约2小时。显然列表预取将是最优先的，因为500 000L将以 $500 000/200 = 2500$ 秒即42分钟来执行。但是为了使列表预取发生，对于这个谓词的RID列表必须抽取到RID池中。有50 000个项被扫描，生成500 000个RID，在RID池中占有2MB。这意味着根据定义9.6.1的规则1必须有4MB的RID池，因为任何方案仅仅使用RID池的一半。因为RID池是缓冲池的一半大小，所以必须提供一个8MB的磁盘缓冲（含有2000个4KB的磁盘页面）。如果缓冲池比这个尺寸小，对于这个查询就不能够应用列表预取。但是今天在系统中使用这样小的缓冲池是不大可能的。 ■

RID池规则使用相当粗糙的试探法来设定资源范围。系统中正在活跃的使用者未考虑在内，如果我们有一个用户以一个大小为100MB的缓冲池执行例9.6.4的这个查询，那么限制我们仅仅用50MBRID池的一半是不合适的，并且导致在没有其他用户竞争RID空间时不能执行列表预取。另外一方面，如果在这种类型的查询中有几个用户都活跃，我们注意到查询并不会导致要求缓冲的共用页的一个很大的集合，这样我们可以把一些缓冲空间转化为RID空间来支持更多的列表预取。

在RID列表上的一个很重要的限制是规则3，它声明生成RID列表索引筛选不能执行。尤其这是一个非匹配索引扫描不会导致在数据行检索阶段中的列表预取。更进一步讲，因为MX处理要求RID列表，MX处理不能利用非匹配谓词。我们将在一个例子之后再进一步讨论它。

例9.6.5 列表预取和索引筛选 回顾一下例9.5.2的查询，其中prospects上只有mailx索引(zipcode, hobby, incomeclass, age)，并回忆一下在带有不同索引的例9.6.2中这个查询是怎样被重复的。

```
select name, straddr from prospects
where zipcode between 02159 and 04158
```

```
and hobby = 'chess' and incomeclass = 10;
```

使用索引mailx，索引扫描在列zipcode上匹配，而不是后面的hobby和incomeclass列上匹配，因此hobby='chess'和incomeclass=10被用作筛选谓词。可以计算出，mailx的索引扫描将遍历索引的 $(1/50)(250\ 000)=5000$ 个叶子页面，以5000S的代价且耗时为 $5000/800=6.25$ 秒，同时1000个数据页面将被以1000R的代价即耗时 $1000/80=12.5$ 秒被检索。因为筛选谓词用来获得这样小的复合过滤因子，我们不能从这些1000个页面中抽取出一个RID列表（因为规则3）。用列表预取执行这1000数据页面的读是不可能的。所以，以5秒的时间完成的一个1000L的数据页面的I/O，在使用串接索引时是不可能的就像我们在例9.6.2中看到的。 ■

过滤谓词和IN-LIST谓词不能用在RID列表的抽取中在理论上似乎没有任何重要的原因，就像在规则2和规则3陈述的那样，除了过滤在一个索引页面的大的集合上执行时RID列表将在一段有些扩展的阶段使用以外。大概设计者正在寻找RID列表抽取上的限制来为其他更有价值的应用保留RID池的空间，并偶然发现了这些规则。然而，查询优化的更多细节会从一个版本到另一个版本的过程发生变化。

2. 多重索引存取中减少返回的点

在多重索引存取中的另外一条关于RID列表抽取的规则不像一条优化规则有那么多的限制。一个带有对于叶子页面遍历的I/O代价的索引上的扫描只有当它通过更大量地减少数据页面检索来补偿自己时才会被执行。为了确定在一个多重索引存取方案中带有一个MX步骤的被扫描的索引，查询优化器采用某些下面定义中的步骤。

定义9.6.2 在MX存取中确定减少返回点的步骤

- 1) 列出查询的WHERE子句中带有匹配谓词的索引。为了简单起见，在下面我们假设每一个索引有一个匹配谓词的不相关集合。
- 2) 按照匹配扫描的过滤因子值递增的顺序放置索引。我们将选择在MX步骤上被执行的索引的一个初始序列，首先是带有最小过滤因子的。这意味着，我们从具有更小的索引I/O代价和在节省数据页面I/O有更大效果的谓词开始，这种方法可以被证明具有优化的结果。
- 3) 对于列出的连续的索引，只有对于抽取出RID列表的索引扫描的I/O代价将以一个对于最后的行检索数据页面的一个减小的代价补偿自己时，才会执行MX步骤。这个规则的一个最简单的例子是，一旦我们到达了一些行，我们不必为了得到行的数目而读出一个新索引的几百页！ ■

事实上，事情有时候比这更为复杂。例如，在第3步考虑的第一个索引与一个表空间扫描相比也许并不能补偿它自己在I/O上节省的时间，因此过程也许不得不考虑在一个节省是明显的之前使用两个索引。例如，如果在一个页面上有20行，第一个索引的1/20的过滤因子并没有节省很多I/O。然而，过滤因子为1/15的第二个索引可以节省大量的I/O。

例9.6.6 我们再次引用带有索引addrx, hobbyx, agex和incomex的表prospects。考虑查询：

```
select name, straddr from prospects
  where zipcode between 02159 and 02658
    and age = 40 and hobby = 'chess' and incomeclass = 10;
```

我们假设使用多重索引存取，并且尽力算出哪一个谓词可以补偿自己。这些子句的过滤因子如下（以升序）：

- 1) $FF(\text{zipcode} \text{ between } 02159 \text{ and } 02658) = 500/100,000 = 1/200$
- 2) $FF(\text{hobby} = \text{'chess'}) = 1/100$
- 3) $FF(\text{age} = 40) = 1/50$
- 4) $FF(\text{incomeclass} = 10) = 1/10$

把对于谓词(1)的过滤因子应用到5千万行上，我们得到检索到的 $(1/200)(50\,000\,000) = 250\,000$ 行，当然是非聚簇的，这样很可能是在5百万不同的数据页面上，并且用列表预取来检索。对于250 000L的执行时间为1250秒。我们忽略索引代价。

在谓词(1)之后应用谓词(2)，对于谓词 $\text{hobby} = \text{'chess'}$ 的索引 hobbyx 的扫描需要许多叶子页面I/O，计算为 $(1/100)(50\,000) = 500$ ，因此代价是500S，需要时间为 $500/800 = 0.625$ 秒。结果我们减少扫描数据页面的数目，从250 000（结果来自于前面的步骤）到 $(1/100)(250\,000) = 2500$ ，并且2500L耗时 $2500/200 = 12.5$ 秒。通过 hobbyx 索引扫描的0.625秒的时间，使得我们从2500秒的一次数据扫描到12.5秒的数据扫描，很明显是值得的。

在谓词(1),(2)以后应用谓词(3)，对于 $\text{age} = 40$ 的 agex 索引的扫描（在例9.6.1中讨论）需要许多叶子页面的I/O，计算为 $(1/50)(50\,000) = 1000$ ，因此代价为1000S，执行时间为 $1000/80 = 1.25$ 秒。结果，我们把扫描页的数量减少到 $(1/50)(2500) = 50$ ，并且50L执行时间为0.25秒。通过索引 agex 的1.25秒的投资可以得到数据扫描从12.5秒降到0.25秒的结果，很明显是值得的。

在谓词(1),(2),(3)以后应用谓词(4)，对于 $\text{incomeclass} = 10$ 的索引 incomex 的扫描需要大量叶子页面的I/O，计算为 $(1/10)(50\,000) = 5000$ ，因此代价为5000S，耗时 $5000/800 = 6.25$ 秒。结果，我们把扫描数据页的数目减少到 $(1/10)(50) = 5$ ，并且5L耗时0.025秒（近似的）。对于 agex 索引扫描投资6.25秒，数据扫描由0.25秒变为0.025秒。这是不值得的，在多重索引存取方案中的MX步骤 incomex 索引将不被扫描。■

9.7 连接表的方法

在本节中，我们研究当前在DB2中使用的三种连接两个表的算法。这些算法分别称为：嵌套循环连接法（nested-loop join），归并扫描连接法（merge scan join），混合连接法（hybrid join）。（DB2 UDB现在实现了嵌套循环连接法和归并扫描连接法，仅有DB2 for OS/390有混合连接法）。在执行一种连接时，每一种连接法在一定的情况下都有性能上的优势。还有其他一些方法未被DB2采用但是仍然在特定的环境下提供了性能上的优点。例如，众所周知的散列连接法（hash join）。但是我们将注意力集中在由DB2提供的连接法上。用来描述DB2连接方法的术语是相当通用的，这些概念中的一些在大多数数据库产品中都已被实现。

我们把两个表的连接定义为是一个过程，在这个过程我们把一个表的列同另外一个表的列组合起来用来回答一个查询。根据这个定义，一个连接发生在一个Select语句的FROM子句中出现两个或多个表的时候。甚至于我们对于来自于两个表的行做一个简单的笛卡儿积（一个表的积），我们称它为一个连接。就像我们看到的那样，一个带有FROM子句和一个含有来自不同表的子查询的WHERE子句的单个表的Select语句，总是由查询优化器转化成为一个等价的连接表的查询语句。首先，我们仅考虑只有两个表出现在FROM子句中的情况。

在DB2中两个表的连接通常以两步发生。第一步，仅仅存取一个表，这个表称为外表 (outer table)。在第二步，第二个表即内表 (inner table) 中的行同外表的行结合起来。其他与没有通过一个索引被检索的两个表中的列有关的谓词，当它们被检索时用以限制行。所有这些的结果是，产生了一个合成表 (composite table)，它包含该连接的所有符合条件的行。如果同第三个表的连接是必须的，那么合成表就成为下一步连接的外表。否则合成表的指定列便提供查询的答案。尽管在一个磁盘工作文件中考虑被完全具体化的一个连接的合成表的结果是最简单的，但是认识到我们也许可以避免这样一个浪费时间的具体化也是很重要的。例如，如果一个用户只是想看结果输出的前面20或者30行，具体化一个百万行的合成表就是极端没有效率的。这样在嵌入式SQL中，当一个连接查询上的游标首先被打开并且第一行被检索时，我们尽量避免具体化表。

1. 嵌套循环连接法

考虑下面的查询：

```
[9.7.1] select T1.C1, T1.C2, T2.C3, T2.C4 from T1, T2
      where T1.C1 = 5 and T1.C2 = T2.C3;
```

在嵌套循环连接中，在循环的嵌套对中被称为外表的表对应于循环的嵌套对中的“外循环”，就像我们在图9-19中见到的伪代码。假设[9.7.1]的Select语句中的表T1是外表，嵌套连接的第一步确定在T1中并且满足在T1上相应谓词的行，在本例中为T1.C1=5。如果我们假设在表T1的列C1上存在一个索引C1X，那么连接的第一步将给出方案表中的一行，它具有下列相关列的值。

PLAN NO	METHOD	TAB NO	ACCESS TYPE	MATCH COLS	ACCESS NAME	PREFETCH	SORTN_JOIN
1	0	1	I	1	C1X	L	N

```
R1: FIND ALL ROWS T1.* IN THE OUTER TABLE T1 WHERE C1 = 5;
    FOR EACH ROW T1.* FOUND IN THE OUTER TABLE;
R2:   FIND ALL ROWS T2.* IN THE INNER TABLE WHERE T1.C2 = T2.C3;
    FOR EACH ROW T2.* FOUND IN THE INNER TABLE
        RETURN ANSWER: T1.C1, T1.C2, T2.C3, T2.C4;
    END FOR;
END FOR;
```

图9-19 对于嵌套循环连接的伪代码（说明查询[9.7.1]）

方案表的行表示如下含义：我们正使用一个多步方案 (PLANNO=1) 的第一步；还没有使用连接的方法 (METHOD=0)；我们从连接的第一个表 (TABNO=1) 中抽取出行，同时使用一个在索引C1X上带有一个匹配列的索引扫描步骤；我们可以用列表预取检索表T1中的行。

既然外表的行已经被确定了（它们实际上还没有抽取出来），一个循环被执行以检索出这些行的每一行。对于外表中符合条件的行，在内表T2上又执行一次检索，所有满足连接两个表的连接谓词T1.C2=T2.C3的T2的行被检索到。注意，因为对于这次检索T1的行是固定的，我们可以把T1.C2的值看做似乎是一个常数K。那么，从T2检索的行恰恰就是满足具有T2.C3=K形式的谓词的那些行，同时表T2的列C3上的索引C3X将使得这个索引更为有效。嵌

套循环连接的第二步使用C3X索引在方案表中具有下列行：

PLAN NO	METHOD	TAB NO	ACCESS TYPE	MATCH COLS	ACCESS NAME	PREFETCH	SORTN_ JOIN
2	1	2	I	1	C3X	L	N

这个方案表的这行表明是它是一个多步计划的第二步（PLANNO=2）；被使用的连接方法是嵌套循环连接法（METHOD = 1）；通过使用一个索引扫描步骤（它在索引C3X上有一个匹配列），我们从连接的第二个表中抽取出行（TABNO = 2），事实上第二个表是T2，我们使用列表预取从这个表中检索行。图9-19包含刚才介绍的两步方法的过程伪代码。图9-20通过使用指定的表T1和T2阐明了对于查询[9.7.1]的嵌套循环连接的方法。

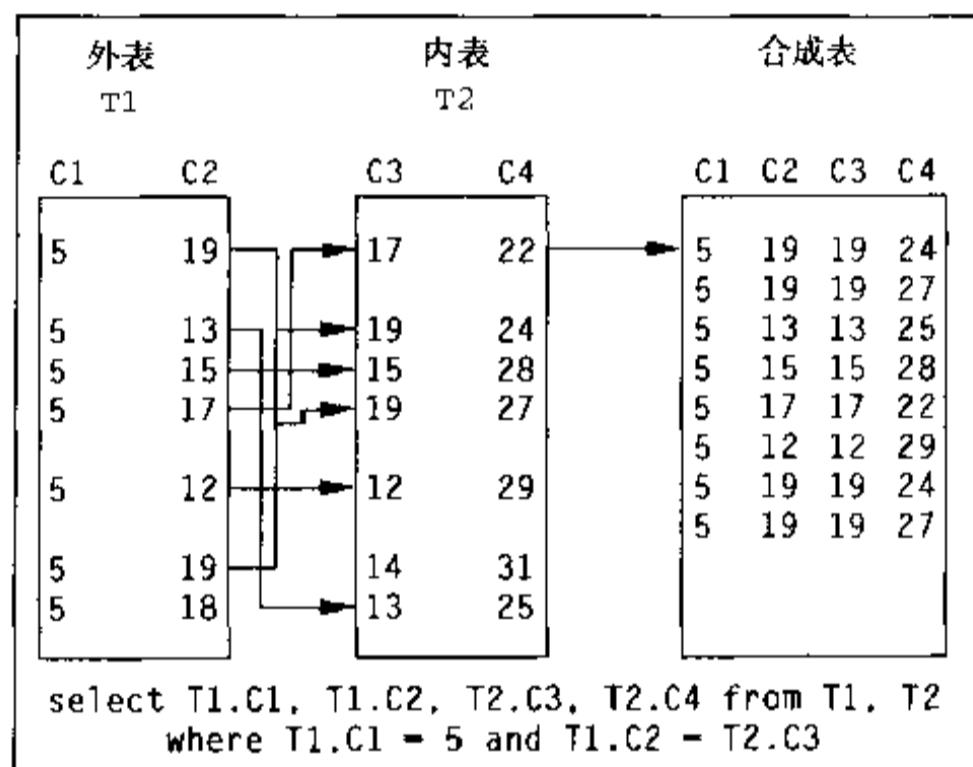


图9-20 嵌套循环连接（阐明查询[9.7.1]）

注意，在图9-19中的标签R1和R2指明了在连接过程中的检索。限制任意一个表中的行的额外的谓词被加到相应的检索中。任一个检索都可以用一个索引扫描（我们在上面假设的）或一次表扫描执行。外表仅有一个检索，而内表有许多检索，它们的数目同在外表中符合条件的行的数目相同。连接的I/O代价通过下面的公式给出：

$$\text{COST}_{\text{io}}(\text{嵌套循环连接}) = \text{COST}_{\text{io}}(\text{外表检索}) +$$

$$\text{外表中符合条件的行的数量} \times \text{COST}_{\text{io}}(\text{内表检索})$$

嵌套循环连接成为连接大的表的一个合适的算法，我们通常期望看到在内表的匹配列上的一个索引来确保高效的检索。在限制谓词被应用后，当来自于外表的符合条件的行的数目很小时或者当内表足够小以至于所有的索引和数据磁盘页面可以在内存缓冲区中常驻时，嵌套循环连接尤其高效。

例9.7.1 假设我们给出两张表：含有列C1和列C2的表TABL1，含有列C3和C4的表TABL2，每个表有1百万行，每行200字节。我们希望估算用嵌套循环连接执行下列查询的I/O代价：

```
select T1.C1, T1.C2, T2.C3, T2.C4 from TABL1 T1, TABL2 T2
where T1.C1 = 5 and T2.C4 = 6 and T1.C2 = T2.C3;
```

我们假设：在TABL1的列C1上存在一个非聚簇的索引C1X，在TABL2上有列C3上的索引C3X和列C4上的索引C4X。假设这些谓词的过滤因子以如下方式给出：FF(C1=const)=FF(C4=const)=1/100; FF(C2=const)=1/250 000; FF(C3=const)=1/500 000。我们可以这样一个问题开始：这个查询检索到多少行？在我们回答问题之前，最好有一个特定的连接方法在头脑中，就像我们在下面的I/O代价分析中看到的那样。

对于回答这个查询的一个可能的嵌套连接方案我们将以执行时间计算I/O代价，其中，外表为T1（TABL1更短的别名），内表为T2（TABL2的别名）。我们考虑的方案包含下列步骤：(1)使用C1X索引，检索来自于T1的所有满足T1.C1=5的行；(2)对于每一从外表T1检索到的行，认为T1.C2是一个常数，重命名为K。使用索引C3X，检索内表T2中的所有使得T2.C3=k的行。当从这个索引扫描的行被检索时，通过确认谓词T2.C4=6来进一步限制行。（注意，这里并不使用索引C4X，我们在下面会解释，我们仅仅用T2.C3=K子句检索两行，C4X索引的进一步使用将命中减小返回的点）；(3)显示出外表行T1.C1和T1.C2，以及对于符合条件的内表行的T2.C3和T2.C4。

下面的表格显示出这个策略的方案表中的行。

PLAN NO	METHOD	TAB NO	ACCESS TYPE	MATCH COLS	ACCESS NAME	PREFETCH	SORTN_JOIN
1	0	1	I	1	C1X	L	N
2	1	2	I	1	C3X	L	N

使用过滤因子和T1中行的数目，在步骤(1)中从T1中检索到的行的数目大约为 $(1/100)(1\ 000\ 000)=10\ 000$ 行，有可能都在不同的页面上。对于PLANNO=1的行告诉我们使用列表预取。假设对于这个索引的I/O代价相对于数据页面I/O代价而言是不重要的。所以，我们假设COST_{io}=（外表索引）=10 000L，执行时间为 $10\ 000/20=50$ 秒。

对于每一个符合条件的外表的行，我们假设值T1.C2(重命名为K)，在列T2.C3的值范围以内。因为FF(C3=const)=1/500 000，我们期望在1百万行的表中检索出2行。对于每一个新的T2.C3的值要求一个到索引C3X的叶子层的随机I/O（假设上层目录节点在内存缓冲区内），然后，两个I/O（平均）检索出包含有两行的两个页面。因此，对于10 000次不同内部循环步骤的I/O代价为 $10\ 000 \times (1R+2R)$ 。在这种情况下，我们将正常地执行一次列表I/O以检索数据页面，但是回忆一下，把两页的列表预取认为发生在 $2/100$ 秒，这会令人误解的，因为有太少的页面被检索到以分摊磁盘臂的寻道时间和旋转延迟。更为合理的是把这个检索考虑为等价于 $2R$ ，所以对子内层循环的执行时间被计算为 $10\ 000 \times (3R)$ ，即 $30\ 000/80=375$ 秒。总共的执行时间为 $50 + 375 = 425$ 秒。

现在确定在这个查询中有多少行，我们看到从表T1中检索到10 000行，对于在T1中的每一行有两行（平均）从T2中同它相连接。所以，在这个点上大约可检索到20 000行，从这以后，一个条件测试发生了，它测试检索到的行是否满足T2.C4=6。用过滤因子1/100，检索到的行的最终数目为 $(1/100)20\ 000=200$ 。

2. 归并连接

归并连接在其他文章中也被称为归并扫描连接（merge scan join）或排序归并连接（sort merge join）。再一次考虑例9.7.1的查询，使用相同的索引假设。

[9.7.2] select T1.C1, T2.C2, T1.C3, T2.C4 from TABL1 T1, TABL2 T2
 where T1.C1 = 5 and T2.C4 = 6 and T1.C2 = T2.C3;

归并连接方法对于两个表仅扫描一次，以它们连接的列的顺序进行。在[9.7.2]的Select语句中，DB2以应用非连接的谓词和创建中间表开始，我们首先计算查询 select C1, C2 from T1 where T1.C1=5 order by C2, 把结果行放入一个含有列C1和C2的中间表IT1中，以C2为序存放行。然后我们计算查询select C3, C4 from T2 where C4=6 order by C3, 以得到含有列C3和C4的一个中间表IT2，以C3为序存放行。注意，通常这些中间表IT1和IT2都很大，所以不能放入内存。它们作为临时表写入磁盘工作文件，行的排序是基于磁盘的排序，稍后对此进行说明。

现在我们准备执行归并的过程，正是这个过程使归并算法得到它的名称。要在IT1和IT2执行一次归并连接，我们使一个指针同每个中间表相关联，初始指向每一个表的第一行。当过程进行中两个指针都向前移动，使得对于两个表中的行任何匹配的C2/C3值都被检测到。除了在IT1中多重相等的值C2和IT2中多重相等的值C3匹配这种情况以外，两个指针都通过表的行一直往前移，并且检测出所有存在的匹配值IT1.C2=IT2.C3。在图9-21的伪代码中，被指针P1指向的表IT1中行的C2值用P1→C2表示，在表IT2中也类似地表示为P2→C3。

```

CREATE TABLE IT1 AS: SELECT C1, C2 FROM T1 WHERE C1 = 5 ORDER BY C2;
CREATE TABLE IT2 AS: SELECT C3, C4 FROM T2 WHERE C4 = 6 ORDER BY C3;
SET P1 POINTER TO FIRST ROW OF IT1; /* OUTER TABLE */
SET P2 POINTER TO FIRST ROW OF IT2; /* INNER TABLE */
MJ: WHILE (TRUE) {
    WHILE (P1 → C2 > P2 → C3) { /* IF P2 NEEDS TO ADVANCE */
        ADVANCE P2 TO NEXT ROW IN IT2; /* ADVANCE IT */
        IF (P2 PAST LAST ROW) EXIT MJ LOOP; /* OUT OF ROWS, EXIT */
    }
    WHILE (P1 → C2 < P2 → C3) { /* IF P1 NEEDS TO ADVANCE */
        ADVANCE P1 TO NEXT ROW IN IT1; /* ADVANCE IT */
        IF (P1 PAST LAST ROW) EXIT MJ LOOP; /* OUT OF ROWS, EXIT */
    }
    IF (P1 → C2 = P2→ C3) { /* FOUND MATCH ON JOIN */
        MEMP = P2; /* REMEMBER P2 START POINT */
        WHILE (P1 → C2 = P2 → C3) { /* LOOP */
            RETURN ANSWER: IT1.C1, IT1.C2, IT2.C3, IT2.C4;
            ADVANCE P2 TO NEXT ROW IN IT2; /* ADVANCE P2 */
        } /* LOOP CONTINUES IF P2 → C3 IS UNCHANGED */
        /* DONE WITH JOIN MATCH */
    }
    /* SINCE FELL THROUGH, P2 → C3 IS NEW OR BEYOND END OF TABLE */
    ADVANCE P1 TO NEXT ROW IN IT1; /* ADVANCE P1 */
    IF (P1 PAST LAST ROW) EXIT MJ LOOP; /* OUT OF ROWS, EXIT */
    IF (P1 → C2 = MEMP → C3) /* IF NEXT P1-> C2 IS SAME */
        P2 = MEMP; /* START OVER WITH P2 */
}
/* END OF MJ LOOP */

```

图9-21 归并连接的伪代码（说明查询[9.7.2]）

图9-22说明了对于查询[9.7.1]的归并连接的方法，它使用指定的表T1和T2。一旦在图9-21的伪代码中发现一个匹配，我们保持P1固定，然后使P2前进，通过所有的重复值。然后我们

使P1前进；如果我们发现一个重复值，这是我们把指针往后移的唯一一种情况，这时，我们设置P2=MEMP，再一次在P2的重复值中搜索。很明显，如果有许多C2和C3的值相同的情况，那么就会有许多行被连接，这样内层循环就有巨大的结果。然而，通常说来，在一个表中间只有很小一部分行会匹配另一个表中的一个以上的行，这是因为我们通常不会在具有大量重复值的列上做连接操作（参见6.7节的关于缺损和无损分解的内容）。在任何情况下，查询优化器使用现存的统计信息就可以确定重复值的可能的数目，并且用大多数的计算机资源来找出任何匹配是很可能的。

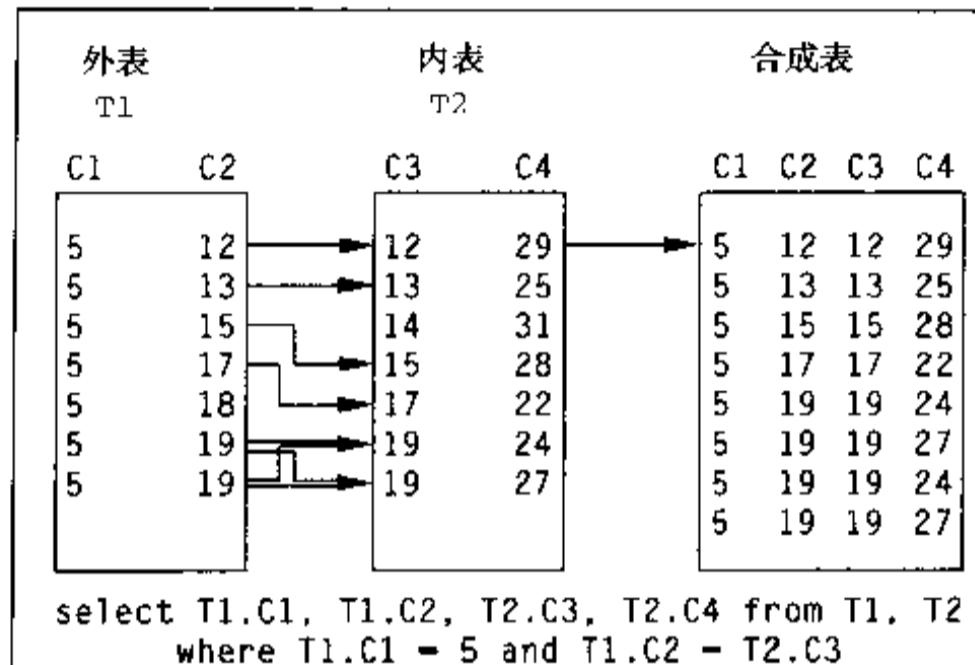


图9-22 归并连接（说明查询[9.7.1]）

例9.7.2 我们用一个归并连接来重复例9.7.1的连接查询：

```
select T1.C1, T1.C2, T2.C3, T2.C4 from TABL1 T1, TABL2 T2
where T1.C1 = 5 and T2.C4 = 6 and T1.C2 = T2.C3;
```

我们都做相同的假设：非聚簇索引依然在TABL1的列C1上存在，还有TABL2中列C3和C4上的索引C3X和C4X。两个表含有一百万行，每一行有200字节，我们有过滤因子FF(C1=const)=FF(C4=const)=1/100, FF(C2=const)=1/250 000, FF(C3=const)=1/500 000。在例9.7.1的嵌套连接中，我们使用索引C1X和C3X。

回答带有一个归并连接查询[9.7.2]的策略包括下列步骤：(1) 使用索引C1X，从T1中检索所有满足T1.C1=5的行（有10 000行），输出列C1和C2的值到IT1中，根据C2的值对结果进行排序；(2) 使用索引C4X，检索所有来自于T2并且满足T2.C4=6的行（也是10 000行），输出结果列C3和C4的值到IT2，并且对它们进行排序。

然后，采用上述的伪代码执行归并连接步骤。归并连接方案在下表中说明。注意，SORTN_JOIN列表明获得IT1和IT2是需要排序的。

PLAN NO	METHOD	TAB NO	ACCESS TYPE	MATCH COLS	ACCESS NAME	PREFETCH	SORTN JOIN
1	0	1	I	1	C1X	L	Y
2	2	2	I	1	C4X	L	Y

正如以前论述的，步骤(1)需要从T1.C1叶子层索引读取10 000项，这我们认为是不重要的，紧接者是10 000L的I/O代价用来检索来自于数据页面的索引行。如果我们假设每一个

被抽取的C1和C2的值需要10个字节，那么具体化IT1需要200 000字节，大约50页。考虑这种排序完全在内存中进行是合适的，因此步骤（1）的I/O代价为10 000L。相同的考虑可以应用于步骤（2）。对于该方案总的代价为20 000L，耗时100秒，比例9.7.1的嵌套连接有所提高，例9.7.1需要425秒。优点来自于在归并连接中使用索引C4X以获得从TABL2中更为有效的批量检索。 ■

注意，从表T1中抽取出行放入中间表IT1中并不总是必要的。如果在T1上有允许我们用谓词T1.C1=5限制T1的行的索引，并且仍然以T1.C2为序存取行，DB2当然采用那种选择。例如，如果T1在（C1, C2）上有索引C12X，对于带有给定谓词的该索引的匹配扫描将提供以C2排序的T1的行，这是可能的。相同的考虑对于T2也成立。

归并连接不总是一个好的策略。

例9.7.3 考虑下列连接查询：

```
select T1.C5, T1.C2, T2.* from TABL1 T1, TABL2 T2
  where T1.C5 = 5 and T1.C2 = T2.C3;
```

我们假定索引C2X, C3X和C5X在表T1和T2的相应列上存在，具有例9.7.1和例9.7.2中的过滤因子 $FF(C2=\text{const})=1/250\ 000$, $FF(C3=\text{const})=1/500\ 000$ 和一个新的过滤因子 $FF(T1, C5=\text{const})=1/1000$ 。可以用下述的方法来计算对子T1的嵌套循环连接的I/O代价。在T1中查找满足 $T1.C5=5$ 的1 000行，我们有一个索引查找的代价1R（对于索引）+1000L（对于数据页面）。对于外表T1的每一行，我们设置 $T1.C2=K$ ，然后在内表中寻找满足 $T2.C3=K$ 的行，大约有两行，对于索引页项查询需要1R，对于行检索需要2R。内层循环的I/O代价计算为 $1000 \times (3R) = 3000R$ 。这样整个嵌套循环连接的代价是 $3001R + 1000L$ ，耗时 $3001/80 + 1000/200$ ，即大约为42.5秒。

对于归并连接，我们按如下方法计算代价。我们可以很容易地计算IT1的抽取需要10秒的I/O，但是我们可以看到，与消耗在循环连接上时间相比，这是不重要的。因为在T2上没有像我们在例9.7.2中的独立的限制谓词（ $T2.C4=6$ ），我们可以有两种选择：一种是通过索引C3X整齐地存取T2的行来执行归并，另外一种是具体化并且对整个表T2排序生成中间表IT2。在第一种情况下，我们将通过非聚簇索引以一个1 000 000R的数据页面存取代价存取T2所有的行，显然，这不是一个好的策略。（这里我们不会使用预取I/O，因为我们不想以RID为序检索行。）在第二种情况下，我们需要以一百万行每行200字节具体化表IT2（注意，T2中的所有列都在选择列表中），同时按照C3对结果行排序。我们推迟考虑磁盘排序，但是需要指出，这些行的磁盘排序也许需要两遍扫描磁盘页面，写出第一遍扫描的结果，然后在为第二遍扫描读入这些结果，I/O代价大于 $100\ 000S$ ，耗时 $100\ 000/800 = 125$ 秒。显然，嵌套循环策略是较好的。 ■

3. 混合连接

混合连接方法(在IBM分类中METHOE=4)与嵌套循环连接和归并连接相比很少使用，为了简单起见，我们只对使用的算法做文字上的描述。

混合连接算法的描述 一个两个表之间的混合连接有一个外表和一个内表，就像嵌套循环连接和归并连接算法中的一样。第一步同外表的归并连接。以连接列的顺序对表进行一次扫描，或者通过一个索引，或者抽取由一些谓词限制的行的集合到中间表IT1以后。当外表的行以连接列的顺序被扫描时，内表的匹配连接列值通过在连接列上的一个索引被扫描。然而，内表的行还是没有存取；相反，来自于外表的行，它有一个给出在内表中每一个匹配连接行

的RID值的额外列，被写入一个中间表IT2。然后，IT2的行以RID的顺序排序，同时列表预取用来拾取来自于内表的列以同外表的行连接。

例9.7.4 再一次考虑例9.7.3的连接查询：

```
select T1.C5, T1.C2, T2.* from TABL1 T1, TABL2 T2
  where T1.C5 = 5 and T1.C2 = T2.C3;
```

就像在例9.7.3中那样，我们假定在表T1和T2的对应列上有索引C2X, C3X和C5X，它具有在例9.7.1和例9.7.2中的过滤因子 $FF(C2=const)=1/250\ 000$, $FF(C3=const)=1/500\ 000$, $FF(T1.C5=const)=1/1000$ 。T1作为外部表的一个混合连接的I/O代价计算如下。在T1中满足 $T1.C5=5$ 条件的有1000行需要查找，我们知道一个索引查找的代价为1R（对于索引）+1000L（对于数据页面）。对于这1000行的每一行我们仅仅需要从表T1中抽取出C2和C5来写入IT1，大约为8000字节，因此创建IT1和紧接着的C2的排序没有I/O代价。对于在外表IT1中的每一行，我们设置 $T1.C2=K$ ，然后在对于 $T2.C3=K$ 的C3X索引中查找索引项，对于每一个项需要1R的索引叶I/O，代价为1000R。

当我们执行这个查找时，我们在中间表IT2中创建具有($T1.C2, T1.C5, RID$)形式的行。这个表将包含2000行，每行12字节，大约24 000字节即六个磁盘页面的缓冲空间，因此我们再一次假设对于创建IT2和根据RID值的排序不需要I/O。最后，我们扫描IT2的行，使用在IT2中排序的RID值作为RID列表的一种来检索来自于内表T2的行，同时用T2的所有列匹配T1的C5和C2的值来产生目标列表行。我们从T2中检索的2000行极有可能都是在不同的页面上，I/O代价为2000L。所以，对于此方法总的I/O代价为1000L（从T1中抽取的行）+1000R（C3X的索引项）+2000L（从T2中抽取的行）。执行时间为 $(1000+2000)/200+1000/80=5+10+12.5=27.5$ 秒，比例9.7.3中计算的嵌套循环连接有所改善，而嵌套循环连接方法优于归并连接。■

嵌套循环连接的优势来自于这样一个事实：来自于内表的所有行的检索可以用带有大数块的列表预取I/O来执行。

4. 多重表连接

在DB2和其他大多数数据库系统中，三个或多个表的连接通过同时连接两个表被执行；前两个被连接的表合成的结果被写到一个中间表中去，然后同第三个表相连接。其结果也许同第四个表相连接，依次类推。连接的顺序不是由非过程性的SQL语句决定的，而是留给查询优化器处理。正确的选择是很重要的。

考虑三个表连接的形式：

```
[9.7.3] select T1.C1, T1.C2, T2.C3, T2.C4, T2.C5, T3.C6, T3.C7
      from T1, T2, T3
     where T1.C1 = 20 and T1.C2 = T2.C3 and T2.C4 = 40
       and T2.C5 = T3.C6 and T3.C7 = 60;
```

对于这样一个连接的查询方案有许多选择。我们可以通过执行两个连接 $T1 \bowtie T2$ 或 $T2 \bowtie T3$ 中的一个开始查询方案。假设我们以 $T1 \bowtie T2$ 开始，我们可以用一个嵌套循环连接或者一个以T1或者T2作为外表的混合连接或者一个归并连接（在归并连接中确定是内表还是外表并不重要）。一旦表 $T4:= T1 \bowtie T2$ 被具体化了（至少在概念上），我们需要执行连接 $T3 \bowtie T4$ ，可以再使用嵌套连接、混合连接或者归并连接。查询优化器需要考虑所有这样的方案以发现最有效

的，在这里对于查询优化器来说有效算法成为最重要的。对于涉及到多表的连接，查询优化器可以需要大量的计算上的努力。

注意，在刚才提到的方案中，连接T1和T2产生T4，然后连接T4到T3，如果查询优化器决定通过一个嵌套循环算法执行连接 $T4 \bowtie T3$ ，中间表 $T4 := T1 \bowtie T2$ 在开始最后连接步骤前不需要被具体化。因为T4的每一行来源于 $T1 \bowtie T2$ ， $T4 \bowtie T3$ 的嵌套连接的下一个循环可以立即执行。这种从一个存取方案的一步的连续的输出行可以作为方案的下一步的输入的技术称为流水线技术（pipelining）。流水线技术使得具体化所需的物理磁盘空间最小化。更为重要的是，从查询中只需要一小部分初始的行（或许因为用户在看了信息的几屏以后，停止滚动光标），通过流水线技术的最小的具体化总是可以节省大量的努力。然而，流水线技术不总是可能的：如果查询优化器选择一个归并连接来计算 $T4 = T1 \bowtie T2$ ，然后另外一个归并连接来计算 $T4 \bowtie T3$ ，其中， $T4 \bowtie T3$ 连接列指定了一个与 $T1 \bowtie T2$ 输出的连接列不同的排序顺序，那么在下一个连接步骤的初始排序被执行前，中间表T4必须完全具体化。

5. 把嵌套循环变换为连接

把大多数嵌套查询转换到仅涉及表连接的等价查询是可能的。对于查询优化器而言，这是一种很重要的技术。

例9.7.5 考虑查询

```
select * from T1 where C1 = 5
    and C2 in (select C3 from T2 where C4 = 6);
```

从概念上讲，我们考虑这个查询可以分两步进行执行。首先，计算子查询，对于C3的选择列表抽取出值的集合到一个中间表IT1中，然后计算外层查询`select * from T1 where C1=5 and C2 in IT1`，在列C2上的条件是它在刚产生的IT1列表中有一个值。做这件事的最有效的方法，如果在IT1上没有索引创建，可能是一个归并连接：`select * from T1 where C1=5`被抽取到一个中间表IT2，行以T1.C2的值进行排序，然后，图9-21的归并连接过程在IT2和IT1之间执行。可能的重复行现在必须被去除，这将在下面解释。这个过程使我们想起下面一个连接形式：

```
select distinct T1.* from T1, T2
    where T1.C1 = 5 and T2.C4 = 6 and T1.C2 = T2.C3;
```

在开始的子查询和这个连接查询给出相同的结果，并且等价的连接形式允许查询优化器使用在嵌套形式中不是显而易见的其他算法。例如，执行一个以T2为外表的嵌套循环连接是可能的。

在嵌套查询转换到连接形式的过程中，DISTINCT关键字的需要来源于下面的观察：如果T1的一行满足连接查询的条件，那么r1.C1=5，同时在T2中必然有一行r2使得r2.C4=6并且r1.C2=r2.C3。但是没有谈及列C3和C4形成了T2的一个关系的关键字，因此完全有可能在T2中还有第二行r3对于C3和C4有同r2相同的值。那么在没有DISTINCT关键字的连接查询的选择列表中，行r1将出现两次。在该查询的最初的嵌套形式中，这一点明显地不会发生，因为T1的每一单行在概念上认为仅仅是一次，被谓词C1=5和C2在IT1中限定或者不被限定。那样，在连接形式中的DISTINCT关键字去除了不会在嵌套形式中出现的重复行。 ■

像这样的一个变换的价值在于，它减小了查询优化器需要考虑的不同类型的谓词的数目以获得优化的效率。一旦例9.7.5的嵌套查询已经被重写为一个等价的连接查询，它就会演化

为一个以前已经解决的问题，并且查询优化器可以使用任何我们已经介绍过的连接方法。下一个例子考虑相关子查询的情况。

例9.7.6 考虑查询

```
select * from T1
  where C1 = 5
    and C2 in (select C4 from T2 where C5 = 6 and C6 > T1.C3);
```

因为这个嵌套形式包含了一个相关子查询，在固定外积的行以使计算T1.C3以前不可能计算子查询。从这方面考虑，唯一正确的方法似乎是：对于T1中的行进行循环取值，T1中的每一行找出通过C4上的索引对应的所有T2中的行，其中T2.C4等于外层的T1.C2，然后再解决条件T2.C6>T1.C3。现在注意到上述的嵌套查询给出与下述连接查询相同的结果：

```
select distinct T1.* from T1, T2
  where T1.C1 = 5 and T1.C2 = T2.C4 and T2.C5 = 6 and T2.C6 > T1.C3;
```

对于这种形式的查询，描绘它执行一次合并连接的策略是很容易的。从T1中抽取出满足条件T1.C1=5的行写入中间表IT1中，并且按T1.C2排序。然后按照列C4对T2进行排序，结果放入IT2中。现在就可以在T1.C2和T2.C4的匹配值上归并连接IT1和IT2（排除重复值），并且使行满足谓词条件T2.C6>T1.C3。这种策略也许比嵌套循环连接要好，尽管后者对于以嵌套形式提出的查询似乎是最自然的。 ■

定理9.7.1给出了在上述情况中的通用形式。

定理9.7.1 下列两种形式的查询给出等价的结果：

```
select T1.C1 from T1 where [Set A of predicates on T1 and] T1.C2
  is in (select T2.C3 from T2 [where Set B of predicates on T2, T1]);
```

和

```
select distinct T1.C1 from T1, T2
  where T1.C2 = T2.C3 [and Set A of predicates on T1]
    [and Set B of conditions on T2, T1];
```

注意到条件中的Set A可以为空，Set B也一样。请注意，如果来自于嵌套查询的Set B中没有涉及到外层查询中的表T1，那么整个嵌套查询是不相关的。上述给出的形式可以推广到在外层查询和内层查询中的多个表的情况。 ■

DB2只有在一定的条件下才执行这种转换，条件如下：

- 1) 子查询的选择列表是一个单个的列，这一点由一个唯一索引具有唯一的值来保证。
- 2) 连接外层查询到子查询的比较操作符或者是IN或者是=ANY（相同含义）。

因此，所有涉及到NOT EXISTS谓词的嵌套查询都不会转换为连接谓词。但是，绝大多数嵌套查询具有等价的连接形式，如果以连接形式提出的查询DB2查询优化器通常会找出一个更为有效的执行方案。这是真实的，尽管在DB2的转换规则下转换成为一个连接查询一般是不会发生的，同时也暗示写查询的人应当尽可能地建立连接形式的查询而不是等价的嵌套形式查询。一旦使用了嵌套形式的查询，可以从EXPLAIN命令的输出来判断是否已经执行了到连接形式的转换。一个连接可以由METHOD列的1、2或4的值来说明。

9.8 磁盘排序

在查询处理中有许多种情况，在这些情况下，对象的集合太大以至于不能一次全部都放入内存进行排序。回顾一下，RID列表被认为全部放入RID池的内存中。结果，用于对RID进行排序的方法可以是你在数据结构编程课程中所学的有效内存排序算法的任何一种，比如归并排序。然而，当排序的对象集合不能完全放入内存时，问题就会困难得多。数据库系统需要使用使磁盘I/O代价最小的方法，而这个问题在以前的编程课程中从来没有提及到。我们称数据(其中有一些必须是常驻磁盘的)的一次排序为磁盘排序。类似于关键字查找，对于驻留内存数据的最有效的查找结构是平衡二叉树或者2-3树，而对于磁盘驻留的数据最有效的结构是B树。

DB2在归并连接和混合连接过程中必然利用一个高效的磁盘排序，其中中间表IT1和IT2通过连接的列或者通过RID来排序。(除非一个索引用来存取已经有序的原始表，否则排序是必须执行的。)这些表足够的小可以放入内存，但是DB2不会只依靠这些，因此要用到磁盘排序。在归并连接和混合连接过程中使用的磁盘排序在这些两步连接计划的第二步发生(PLANNO=2, METHOD=2或者4)，排序通过方案表中的两个新的列值来反映，即SORTN_JOIN=Y和SORTC_JOIN=Y。还有许多其他的方案表的列用来表明什么时候磁盘排序正在被执行，用于何种目的。这些方案列包括SORTC_ORDERBY(通常在Select语句中使用了ORDER BY子句时使用)、SORTC_GROUPBY(对应于GROUP BY子句把项聚集起来)还有SORTC_UNIQ(通常当在选择列表中用一个DISTINCT关键字时使用)。当排序正在执行时，这些列包含值'Y'，其他情况下为'N'。注意到在方案表中有三个列，在DB2目前的版本中从不使用，所以它的值总是'N'；这些列是SORTN_GROUPBY, SORTNJ_UNIQ和SORTN_ORDERBY。

N路归并磁盘排序算法

下面，我们谈论由某个排序键(sortkey)列进行排序的行组成的一个表。注意我们不是必须指一个关系表，而是任何数据库系统不能全部放入内存的记录结构对象的列表。一个磁盘排序以一系列步骤产生。在每一步，该算法都向着在磁盘上产生一个有序的结果而努力，这个过程就是尽给出与被排序的表的整个空间相比较的内存空间的限制。如果偶然地，被排序的表和所有临时排序信息在排序过程中可以完全放入内存，那么一步就可以产生完整的有序列表。否则，第一趟将表的最大可能部分按照升序进行排序，并且作为一个有序块写到磁盘，然后重复这个过程直到表中的所有行都被以一系列有序块的形式被写到磁盘。每一个相邻把N个块的连续归并到新的、更长的含有有序行序列的块中，直到所有的行都被排序为止。N路归并排序是在数据结构课程中驻留内存的两路归并排序的推广以与B树是二叉树的推广一样的方式。通过增加N，我们可以减少这样趟的次数，其中每一趟必须从磁盘中读出表中所有的信息，再把它们写回磁盘。这样，I/O代价需要最小化。

更具体一些，假设我们有一个由需要排序的数据的短行组成的表，它占有D个磁盘页而(我们假设在磁盘上磁盘页面是连续的，这样就可以用顺序预取I/O)，同时我们有内存中的M+1个磁盘缓冲页面，其中 $M \leq D$ ，它就是我们在排序中可以允许使用的空间。我们希望演示一个N路归并排序的过程。正像我们将看到的，如果 $M \geq D$ ，我们可以在内存中完成整个排序。可以证明，行的精确长度不是很重要的，只要行总是完全在一个磁盘页面中(行不可能跨越页面)。为了有一些数字可以达到说明的目的，让我们假设 $D=10\,000$ 页， $M=2$ 页，下面，我们

将执行一个M路归并排序，也就是说，因为 $M=2$ ，这是两路归并排序。下面的所有计算都可以推广为D和M的任意值。

首先，我们说，要排序的行最初存储在称为A1区域的10 000个连续磁盘页面中，为简单起见，假设我们有第二个含有10 000个连续页面的磁盘区域，名为A2。在归并排序的连续步骤中，把页面从A1移到A2，再反过来。实际上，我们并不需要这么多的空间，但是用这两个区域算法更容易说明。开始我们假定行没有顺序，接着描述M路归并的第一趟。注意，在描述M路归并的每一趟时，我们并不考虑任何的顺序预取I/O的优化。最后，当考虑顺序预取I/O时，我们将描述更为通用的N路归并排序，其中 $N < M$ 。

第1趟 排序模块读出要排序的表的M页的第一块，用1, 2, ..., M表示，把它们从磁盘区域A1读入到内存；通过在一个后面要描述的方法在缓冲区内对行进行排序（我们假设在排序后有序的行可以再放置在M页上，以使在页面的边界上没有缝隙——如果所有行都有相同的长度这是很简单的）；把这M页作为一个有序的块写到区域A2的相应位置，页1, 2, ..., M。这个过程用M页的连续块重复（对于最后一块，如果M不能整除D也许会少一些）。模块从区域A1中读出编号为 $M \times i+1, M \times i+2, \dots, M \times i+M$ 的页面，对这个块中的行进行排序，并且把结果页面写入到区域A2，块 $B_{i,1}$ 。因为已经固定了值 $M=2$ （一个不大现实的小数字，只用于说明的目的），我们可以用下图来勾画出第1趟写入区域A2的结果：

页面	1 2	3 4	5 6	7 8	9 10	11 12	13 14	15	D - 1 D
块	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7		$B_{(D-1)/2}$

页被标号为1, 2, 3, ..., D - 1, D，它们用块 B_0, B_1, \dots 来编成块（两页为一块），意味着在每一块（包含两页）中间所有的行都是处于有序状态。我们这里假定D是偶数，在区域A2的最后两页D - 1和D分别是奇数和偶数。如果最后的磁盘页面D是一个奇数，那么最后一个块 $B_{(D-1)/2}$ 在长度上是单个的磁盘页面。在第1趟中，我们初始化了磁盘排序过程，但是后续的步骤是实际归并发生的地方。

后面的步骤 每一个归并步从最近完成的M块的组中读入最初的页面，同时把这些块归并到一个更大的块，一次把一页写入到不包含输入的不同区域的一个新块上。作为一个例子，上述 $M=2$ 的例子第2把在A2区域上的第1趟的输出作为输入，把两个块归并，每个块含有2页，同时把行的有序的结果写入到在区域A1上的长度为4页的块上。

页面	1 2 3 4	5 6 7 8	9 10 11 12	13 14 15 D - 1 D
块	B_1	B_2	B_3	B_4		$B_{(D-1)/4}$

对于这样的归并仔细考虑它的I/O代价。归并步骤以读入 B_1 块的第1页和 B_2 块的第1页作为开始。 B_1 和 B_2 都是按照排序的顺序，因此显而易见我们只需要看每一块的第一页以找出包含在两个块中最小的行。事实上，一旦这些初始的块的页面被读入内存，我们可以做的就是在每一个这样的页面上寻找最初的行，比较两个排序键值找出按照排序顺序排列的最小行。现在我们可以把两个块归并到一个更大的输出块，把归并的行移动到一个有序的在内存中的输出页面上，并且进行必要的磁盘读和写。

在通常情况下，我们从先前的趟中读入来自于M块中的每一块的第1页，在每一页上放置指向最初行的游标。然后，我们确定在任何游标下的最小行，并且把它放置于缓冲页面的开始，

开辟缓冲页面是用来放置排序输出的。(回顾一下，有M+1个缓冲页面可用，在需要归并的每一块上的每一页面上都需要一页，输出需要一页。)在这一点上，必须要前移游标，它指向的行刚刚被归并到那一块的下一行。如果在当前页没有行了，我们从剩下的块中读入下一页。如果在那一块中没有页了，我们删除这个游标并继续在剩下的游标中归并直到所有的行都被归并。当排序的输出页满时，我们把它写入输出块的下一页上。在这个过程的最后，在M个输入块中的所有行，它们的长度为b页，已经被归并成一个新的含有 $M \times b$ 页的输出块上排序序列。从先前的趟中，我们继续归并M块，直到所有行已经被归并成新的更长的块，这样全部步骤结束。我们一直执行后续的步骤，直到某一趟它的输出块的尺寸超过D；在这一点上，排序结束。

例9.8.1 两路归并排序的例子 假定有一个2000字节行的序列，在每一个磁盘页面上有2行，按照下面一系列的键值的顺序进行排序，序列代表了行的初始顺序。

57 22 99 64 12 29 46 7 91 58 69 17 65 36 33 28 77 6 54 63 88 95 38 44

经过两路归并排序的第1趟，我们有（每个两页的块含有4行）：

22 57 64 99	7 12 29 46	17 58 69 91	28 33 36 65	6 54 63 77	38 44 88 95
-------------	------------	-------------	-------------	------------	-------------

第2趟后，我们有（四页的块）：

7 12 22 29 46 57 64 99	17 28 33 36 58 65 69 91	6 38 44 54 63 77 88 95
------------------------	-------------------------	------------------------

第3趟后，我们有（八页的块，最后一个块没有对来归并）：

7 12 17 22 28 29 33 36 46 57 58 64 65 69 91 99	6 38 44 54 63 77 88 95
--	------------------------

最后，经过第4趟，我们就有了有序序列：

6 7 12 17 22 28 29 33 36 38 44 46 54 57 58 63 64 65 69 77 88 91 95 99

■

显然，在M路归并排序的每一趟所有的非中止的块都以M因子增长。来自于先前趟的块的每一页只要读入一次，在产生这些长块的过程中，等量数量的页面被写回磁盘。那样，第1趟需要2D页的I/O，每一个后续的归并步也需要2D页。图9-23给出了输出块尺寸和两路归并算法的对于后续步骤需要的I/O数，假定数据有10 000页。

归并的趟数	块的大小	每一趟中的I/O数
1	2	20 000
2	4	20 000
3	8	20 000
4	16	20 000
5	32	20 000
6	64	20 000
7	128	20 000
8	256	20 000
9	512	20 000
10	1024	20 000
11	2048	20 000
12	4096	20 000
13	8192	20 000
14	16 384	20 000

图9-23 用两路归并排序对于10 000页的磁盘排序步骤的序列

回顾一下，当块的尺寸超出了在表中页面的数目，表可以成功地被排序。这意味着14趟表可以被成功地排序，用了 $14 \times 20\ 000 = 280\ 000$ 次I/O。更为普遍的，经过K趟，我们可以得到一个块的尺寸为 2^k ，当 $2^k \geq D$ 时，或者换句话说，当 $K = \text{CEIL}(\log_2(D))$ 时，我们就完成了我们的排序。每一趟仍然需要 $2D$ 页的I/O代价，因此对于一个D页的M路归并排序要求的I/O总数是 $2\text{CEIL}(\log_M(D)) \times D$ 。

作为一个必须的I/O的因子，从 $\log_2(D)$ 到 $\log_M(D)$ 的变化是相当重要的。图9-24给出了输出块的尺寸和四路归并算法连续趟间需要的I/O，假设数据有10 000页。图9-25给出了一个100路归并排序对应的数据。

归并的趟数	块的大小	每一趟中的I/O数
1	4	20 000
2	16	20 000
3	64	20 000
4	256	20 000
5	1024	20 000
6	4096	20 000
7	16 384	20 000

图9-24 用四路归并排序对10 000页磁盘排序步骤的顺序

归并的趟数	块的大小	每一趟中的I/O数
1	100	20 000
2	10 000	20 000

图9-25 用100路归并排序对于10,000页磁盘排序的步骤

例9.8.2 一个三路归并排序的例子 给出与前一个例子相同的序列，演示一个三路归并排序。我们从如下初始顺序开始：

57 22 99 64 12 29 46 7 91 58 69 17 65 36 33 28 77 6 54 63 88 95 38 44

经过第1趟，我们有3页的块：

12 22 29 57 64 99	7 17 46 58 69 91	6 28 33 36 65 77	38 44 54 63 88 95
-------------------	------------------	------------------	-------------------

经过第2趟，我们有：

6 7 12 17 22 28 29 33 36 46 57 58 64 65 69 77 91 99	38 44 54 63 88 95
---	-------------------

经过第3趟，我们得到同两路归并排序相同的有序顺序：

6 7 12 17 22 28 29 33 36 38 44 46 54 57 58 63 64 65 69 77 88 91 95 99

例9.8.3 两步方案请求排序 回顾例9.3.6，其中我们对于prospects表使用由以下命令创建的naddrx索引：

```
create index naddrx on prospects (zipcode, city, straddr, name) . . . ;
```

同时对于这样的查询需要找出一个方案（为得到一个排序在这里有少量的修改）：

```

select name, straddr from prospects
  where zipcode between 02159 and 04158
  order by name, straddr;

```

查询方案执行一个INDEXONLY步骤，其中值从02159到04158的zipcode的叶子层项的范围使用顺序预取来存取，name和straddr的值被抽取出来。假定FF (zipcode between 02159 and 04158) 大约是2/100，抽取这些值的索引扫描步骤需要存取1百万项，每项60字节，15 000磁盘页面，因此I/O代价大约为15 000S，需要 $15\ 000/800=18.75$ 秒。

必须执行一个排序以把输出 (name, straddr) 行放入到正确顺序的输出上，因此我们从寻找用于放置这些行的临时表IT1的长度入手。我们假定name和straddr列各有20字节，因此行的长度为40字节，一页可以容纳100行。我们需要具体化一个含有1 000 000行的表，需要10 000磁盘页面。

假定我们有11块内存缓冲区页面用来执行排序（一个不现实的很小的数目，只是为了说明方便）。所以我们执行10路归并排序。第1趟以后，我们有尺寸为10的块；第2趟以后，尺寸为100；第4趟以后，尺寸为10 000，这样我们便完成了。我们在执行块归并的I/O中没有机会在I/O中做顺序预取I/O，认识到这一点很重要。所以，从第1趟到第4趟每一趟都需要5000页读和写，总共 $4 \times 2 \times 5000=40\ 000$ R。最后一趟不需要磁盘写，因为行现在以正确的顺序返回给用户。我们将具体化在缓冲页面中的最后一趟的行，并且尽快地把它们输出给用户，然后重新分配页面，使得最后的一次写不必要。所以最后一趟的I/O代价是5000R，对于排序的整个I/O代价是45 000，需要 $45\ 000/80=562.5$ 秒。

注意到排序步骤的最后一趟从来没有写页的代价，除非后续步骤需要一个具体化的表作为输入。更为通常的情况是，忽略从一个步骤的输出到另外一个步骤的输入的，不具体化其中的行。在一个排序的最后一趟，你应当对这个“结束游戏”高度敏感。 ■

在现代数据库系统中，对于一次排序可获得的缓冲页面的数目足够大，使得两路排序可能是任何人可看到的最长的一种。我们可以做一个粗略的估计，100页排序全部在内存中，所有其他的排序在两趟间发生。

9.9 查询性能基准程序：样例研究

软件基准程序是对于测量方法集合的一个规定，用于评估某种软件能力，通常是性能。一个好的基准程序使得管理员可以完全基于性能价格比来对软件/硬件平台做出购买的决定。另外，如果购买基于其他考虑，比如现存系统的兼容性，这时管理员可以用基准程序来确认性能价格比不会严重到使他改变购买决定。对于软件开发人员而言，基准程序也提供了一个好的质量确保测试，使得他们计划和实现他们的新产品来在关键方面提高性能。

在下面几节提出的集合查询基准程序(Set Query benchmark)对于很大范围的查询测量数据库系统的性能。查询以SQL的形式定义，也可以以其他查询语言实现，同时意味着可以移植到尽可能多的平台。例如，IBM OS/390大型机的DB2、Sun Solaris Unix系统的ORACLE或者Windows PC上的FoxPro。这里报告的测试结果来源于运行于小型IBM系列/390上的DB2版本2.3上的一个基准程序。

我们希望，这些具体的结果和由它们产生的对于查询方案的讨论可以使得你对于DB2执行查询的方法的评估有一个具体的认识，而这些仅通过单独的理论论述是无法获得的。然而，

在下面的几节中将描述很多细节，而对于这些细节的掌握不是很容易的。为了避免混淆，你可以细心地继续下去，一次只看一部分细节，同时做本章后的相应习题来对你的理解做测试。我们已经提出了所有用于讨论的基本原理，但是为了支持这些细节而不断地引用前面的章节是必须的。

重要提示 在本节中使用旧的磁盘I/O速度 在本节中，我们将使用在20世纪90年代初的磁盘I/O速率，这是因为数据是在那时收集的。这些速度值恰好是图9-10引用速度值的一半，也就是对于随机I/O为40页/秒，对于列表预取为100页/秒，对于顺序I/O为400页/秒。考虑实际的执行时间，磁盘在页面的存取速度方面没有很大进展。在同样的时期，它们的典型的磁盘存储能力已经从500MB上升到10GB，倍数是20。

同许多不同框架的数据库性能相比较，集合查询基准程序具有一种不同的任务，它通过运用一种简单统一的标准来达到这种比较，标准就是对每一个平台用一个简单的数据来评估：每分钟每一个查询所花费的美元（\$/QPM）。系统的花费代表了软件、硬件、还有5年的维护开销。（我们总是以每月或每年来计算软件和维护费用，而硬件是一次性购买的。到了5年结束时，通常认为系统已经完全折旧，而且需要用一个新的、更快的、更便宜的来替代它。）在系统的估算当中，每分钟的查询（QPM）是对于一个特定查询工作负荷(参见9.1节)吞吐量的一种度量，它接收相同负荷的基准程序的所有查询。尽管看上去这有相当的局限性，一个更为复杂的方法也是可能的，其中对于一个任意的工作负荷的QPM的估算可以来自于已公布的详细的计时，对于每一个单独的被测量的查询而言是基准程序要求的。这是一种高度的扩展能力，理论上它允许DBA对于其负责的工作负荷在不同的平台上的性能进行比较。

在第10章中，我们将讨论另外一个不同的基准程序，被称为TPC-A基准程序，它测量数据库更新事务的性能；但是现在我们集中于DB2的查询性能来完成我们查询优化的内容。

1. BENCH表

集合查询基准程序在一个简单表上完成它所有的查询，这个表被称为BENCH(基准)表，它被很详细的规定使得每一个执行这个基准程序的人都得到相同的结果。缺省的BENCH表包含1 000 000行，每行200个字节，同时含有行的磁盘页面应当被装载95%。（在规范的框架内一个多于1 000 000行的BENCH表也是可能的，但是在接下来的内容中不考虑这种情况。）为了提供带有过滤因子范围的索引谓词，BENCH表具有13个具有整形值的索引列，假定每一个值的长度为4字节。首先，我们有KSEQ列，它是一个键列，值为1, 2, …, 1 000 000，与装载行的顺序相同。如果KSEQ是一个完全的簇索引的列，那么我们会看到最好的性能，意味着行按照KSEQ的顺序，并且行所在的页面在磁盘上以连续的顺序聚集成块。这通常可以完成，甚至对于不支持簇索引的数据库产品，它是通过使用一个新近初始化过的磁盘保存数据然后在装载入行以前以KSEQ的顺序对行进行排序来实现的。所有其他的列含有自然数值，它们的值以它们的名字为基数。例如，列K2只有值1和2，K4具有值1, 2, 3和4。图9-26给出了在BENCH表中所有索引列的一个列表。

注意出现在一个列名最后的字母K，代表1000的倍数。这样K40K代表从1到40 000的值。在BENCH表中除了KSEQ列的所有其他索引列在图9-26规定的合适范围内随机产生整数值。把值赋给列的随机数产生函数在基准程序的规范中提供以确保对于执行基准程序的不同地方得到相同的结果。当然，数字是伪随机的，因为它们是由一个简单函数产生的。然而，在效果

上每一列在适合的范围都有随机的值。参见图9-27对于BENCH表的前十行的索引列的一张表。这些列的随机特性使我们可以对于许多查询的结果给出一个相对精确的统计预测。例如，考虑查询

```
select count(*) from BENCH where K5 = 2;
```

列名	包含在该列中的值的范围
KSEQ	依次的1,2,⋯,1 000 000
K2	随机的1,2,
K4	随机的1,2,3,4
K5	随机的1,2,3,4,5
K10	随机的1,2,⋯,10
K25	随机的1,2,⋯,25
K100	随机的1,2,⋯,100
K1K	随机的1,2,⋯,1000
K10K	随机的1,2,⋯,10 000
K40K	随机的1,2,⋯,40 000
K100K	随机的1,2,⋯,100 000
K250K	随机的1,2,⋯,250 000
K500K	随机的1,2,⋯,500 000

图9-26 基准表的索引列

因为对于K5有5个不同的值，对于谓词K5=2的过滤因子是 $1/5=0.20$ ，在BENCH表的1 000 000行中，我们期望可以搜索到接近200 000行的数据（实际搜索到的数目是200 637）。

KSEQ	K500K	K250K	K100K	K40K	K10K	K1K	K100	K25	K10	K5	K4	K2
1	16808	225250	50074	23659	8931	273	45	4	4	5	1	2
2	484493	243043	7988	2504	2328	730	41	13	4	5	2	2
3	129561	70934	93100	279	1817	336	98	2	3	3	3	2
4	80980	129150	36580	38822	1968	673	94	12	6	1	1	2
5	140195	186358	35002	1154	6709	945	69	16	5	2	3	2
6	227723	204667	28550	38025	7802	854	78	9	9	4	3	2
7	28636	158014	23866	29815	9064	537	26	20	6	5	2	2
8	46518	184196	30106	10405	9452	299	89	24	6	3	1	1
9	436717	130338	54439	13145	1502	898	72	4	8	4	2	2
10	222295	227095	21610	26232	9746	176	36	24	3	5	1	1

图9-27 BENCH表开始10行的索引列的值

13个索引列的总长度 $13 \times 4 = 52$ 字节。为了弥补200字节行的另外148字节BENCH表有8个在检索中从来不用的字符列：S1（8个字符），S2到S8（每个20个字符）。当考虑到一次表空间扫描的资源代价时，避免非索引列的查询的决定是合理的，这一点我们一会儿就会看到。

行含有200个字节的用户信息，但是由于必要的开销信息实际上会长一些。对于恰好200

字节长的行，我们可以把每20行装载入一页4KB的磁盘页面中。对于1 000 000数据行，这需要 $1\ 000\ 000/20 = 50\ 000$ 页。因为额外的行长度和页只有95%填满，所以在DB2（版本2.3）中只有18行/页，数据页面的数目为 $\text{CEIL}(1\ 000\ 000/18) = 55\ 556$ 。参见图9-28中关于基准程序的DB2统计信息。

基准表	基数(CARD)	页数(NPAGES)
	COLCARD	INDEX NLEAF
KSEQ	1000000	55 556
K500K	432419	2168
K250K	245497	1682
K100K	99996	1303
K40K	40000	1147
K10K	10000	1069
K1K	1000	1053
K100	100	1051
K25	25	1051
K10	10	1051
K5	5	1051
K4	4	1051
K2	2	1051

图9-28 在DB2版本2.8中的BENCH表的统计信息

所有索引也是95%填满，每个索引对应于一个索引列（没有串接不同列的索引），索引名基于相应的列名，同时加上词缀X；这样，列K100具有索引K100X。具有极大数目重复值的索引，例如K2到K100，具有长度为4字节的索引项，这是因为DB2执行了索引压缩。这样我们在每一叶子页面上可以放置1000个项（如果页是100%填满的），大约需要1000个索引叶子页面来包含所有项。在图9-28中我们对于每一个索引的磁盘页面的数目（NLEAF）提供了实际的估算值。对于具有很少重复值的索引，压缩是不必要的，有效长度会更大，因此需要更多的磁盘页面。对于KSEQ也许没有压缩，同时压缩对于K500K也毫无帮助，因此对于这些索引NLEAF是最大的。

注意在图9-28中，对于每一列的COLCARD值就是所有小基数列中期望的值，而对于大基数列的值它是有缺陷的。例如，K500K的COLCARD值只有432 419，比500 000这个值小许多。如果我们考虑列值是随机产生的，并且想像一下把1 000 000支标枪投向500 000个空位时的情景，那么对于上述情况产生的原因就会显而易见了。根据公式[8.6.4]，击中空位的期望值是：

$$M(1 - e^{-NM}) = 50\ 000(1 - e^{-2})$$

用计算器来计算这个公式，我们可以得到432 332，接近于观察到的实际值。

2. 装载时间测量

下面介绍的测量值是运行于IBM 9221型170主机上的DB2版本2.3的结果，它有1200个4KB内存缓冲区页面（基准程序要求的缓冲空间），还有两个3390磁盘驱动器（可以允许任何数目，但是最有效的是一个用于数据页面而另一个用于索引）。

图9-29是装载BENCH表和执行RUNSTATS以收集统计信息的执行时间和CPU时间，还有容纳表和索引需要的磁盘存储空间。

	执行时间(秒)	CPU时间(秒)	磁盘空间(MB)
装载	10.170	3186	
RUNSTATS	5082	1535	
所用的磁盘空间			296

图9-29 基准程序装载时间测量, DB2版本2.3, IBM9221型170主机

9.10 查询性能测量

在这一节, 我们通过运行标注为Q1, Q2A, Q2B, Q3A, Q3B, Q4A, Q4B, Q5, Q6A, 和Q6B的一系列查询来提供测试结果。我们再一次提醒你应当细心, 一次只看一段细节。每组查询通常包括单个SQL查询形式的一套测试, 通过谓词的变化来使用不同的索引列和提供不同的过滤因子。(请看例子, 图9-30是查询Q1的结果。) 像Q2A和Q2B这样一对查询指近似但又略有不同的SQL查询形式。

KN	执行时间 (秒)	CPU时 间(秒)	获取页面 请求	随机I/O 次数	顺序预取 I/O次数	列表预取 I/O次数
KSEQ	1.33	0.07	3	3		
K100K	0.59	0.08	3	3		
K10K	0.89	0.07	3	3		
K1K	0.67	0.09	4	4		
K100	0.83	0.19	14	2	2	
K25	1.73	0.58	44	25	3	
K10	2.20	1.34	107	54	5	
K5	3.47	2.55	214	46	9	
K4	5.09	3.13	265	57	10	
K2	7.73	6.17	528	48	18	

图9-30 Q1测量

对于每一个被测量的查询, 我们都提供以秒为单位的执行时间(那就是, 等待响应的挂壁钟的时间)、以秒为单位的CPU时间、请求的页的总数, 分解为随机I/O、顺序预取I/O还有列表预取I/O的次数。(这些I/O中的任意两个可以为0。)这些数据可以从DB2PM产品(DB2性能监视器)中的一个标准的MVS日志程序中获得, 在9.11节中会分别讨论它的用户许可和价格。表中列出的随机I/O在DB2PM报告中标明为同步(sync) I/O。注意, 这里的所有查询都是以嵌入式SQL语句在程序中提交, 这样你就会看到执行时间和CPU的时间都要略小于直接从标准用户界面(SPUFI)提交的即席查询。为了使I/O测量保持一致性, 我们在每一个查询组执行前都会刷新内存, 使得存取的磁盘页面不可能已经常驻内存。刷新内存的过程就是执行一个不同的很长的查询, 尽管在实际使用中很小的重叠也是可能的。我们将一边叙述一边依据来自于DB2的EXPLAIN结果说明每一个查询的结果。

1. Q1查询

Q1查询具有如下形式：

```
For each KN ∈ {KSEQ, K100K, . . . , K4, K2}
    select count(*) from BENCH where KN = 2;
```

就像在Select语句前的集合符号表明的那样，符号KN代表任何一个索引列KSEQ, K100K, . . . , K4, K2。这种情况中的每一个都代表了一个查询，它们的测量结果在图9-30中，它们共同组成了Q1查询组。

注意，Q1是在文本检索的早期应用中一个典型的查询。（参见例3.10.7）。一个用户要查询在摘要中出现特定关键字的所有已出版的期刊中的文章，他也许首先查询有多少文章包含这个关键字，然后再进一步精炼查询直到文章的数目足够少，使他可以检索每一篇文章的全部摘要。显然，关键字的过滤因子也许变化剧烈，而且从一个关键字到另外一个也许不可预料（如关键字='experiment%'同关键字='ruthenium'），这个动机对于不同的列的基本数有KN范围。

应用于Q1组中的所有查询的EXPLAIN命令表明：我们可以使用合适的KN索引（例如，在KN代表K100的情况下，ACCESSTYPE=I, ACCESSNAME=K100X），同时查询可以在不依赖于数据（INDEXONLY=Y）的情况下完全解决。那么在图9-30的最上面一行，其中KN代表KSEQ（我们在下面以KN=KSEQ来表示），我们看到存取KSEQX索引的三层直到在叶子层的唯一一项需要执行三次随机I/O。回顾一下内存已被刷新，因此通常执行最大数目的I/O次数。

在具有重复值的情况下，谓词KN=2在索引中是这样解决的：先在适合的索引叶子层中走到最左边的值为2的地方，然后从左到右前进直到没有项值为2为止。对于谓词K100K=2，这意味着要大约遍历10个项；对于K10K，是100个项，等等。（在附录D中给出了在集合查询基准程序中大多数查询检索到的行的精确数目。）因为对于每一叶子页面而言有大约1000个项，对于刚才谈到的两个谓词我们不期望有多于3次的I/O（叶子层的遍历也许不必跨越页面），对于K1K=2也许需要4次，其中有1000个项（遍历实际上确保从一叶子页到另外一页）。对于谓词K2=2，我们期望在叶子层大约遍历500 000个项，这意味着有526叶子页面，或者对于K2X索引是NLEAF值的一半（参见图9-28）。我们从获取页（Get Page）请求的数目可以看到G=528，就是存取了526个叶子页面和2页额外的索引页面。我们也可以看到执行了18次顺序预取I/O和48次随机I/O。我们可以把通过预取存取的页面的总数的计算为 $(528 - 48) = 480$ ，每一次预取读入的页面的平均数目为 $(480) / 18 = 26.7$ 。注意到DB2对于每26.7页使用18次顺序预取I/O，而不是最大的32页的预取次数。这一点的原因主要是技术上的，并且对此兴趣不大，但是要注意到这种类型的短预取的I/O出现在Q1测量的几行上。

对于更长的查询大多数的执行时间消耗在CPU上面，而不是I/O的等待时间。用我们的经验规则，我们可以大致计算出I/O的执行时间。在K2的情况下，在每秒400S的速度下480S需要1.2秒，以每秒40R的速度48R需要1.2秒，因此总的执行时间为2.40秒。加上CPU的6.17秒，我们得到8.57秒的执行时间。因为执行时间只有7.73秒，我们推断部分I/O的时间同CPU的时间是重叠的，我们在接下来的查询中将会看到这是一个完全合理的结果。CPU时间的测量可以由一个遍历叶子层项数目一个线性函数来预测。如果预测的CPU时间表示为T，遍历的项的数目为N，那么我们可以写为：

$$T = 0.0000122N + 0.07$$

这个线性形式在测量的整个范围内最大的误差为0.05秒。这里，常数0.07秒代表一个相对稳定的起始代价，常数0.0000122秒代表处理每一个叶子层项必须的时间，其中没有计算I/O的开销和其他不重要的操作的开销。如果有一个6.5MIPS的CPU，对于每一个项执行的实际的指令数是 $(6\ 500\ 000)(0.0000122)=79$ ，相对于以前的版本有了很大提高，但是当我们考虑到所有正在被做的是计算一个长的列表中连续的项时，这个时间仍然有些大。存在着一个更为有效的计算方法；最初的集合查询论文测试了一个称为MODEL 204的数据库产品的性能，它可以以20至30倍的高效来执行Q1查询。但是，大多数数据库产品在CPU方面都不会像DB2这么高效。

2. Q2A查询

Q2A查询具有如下形式：

```
For each KN ∈ {KSEQ, K100K, . . . , K4}
    select count(*) from BENCH where K2 = 2 and KN = 3;
```

测量结果如图9-31a所示。

KN	执行时间 (秒)	CPU时 间(秒)	获取页面 请求	随机I/O 次数	顺序预取 I/O次数	列表预取 I/O次数
KSEQ	0.38	0.08	4	4		
K100K	1.27	0.09	14	4		1
K10K	2.27	0.14	100	4		4
K1K	12.46	0.39	969	7		31
K100	71.33	2.29	9167	17	3	287
K25	108.41	7.59	28 935	45	4	903
K10	125.83	14.35	47 248	47	5	1474
K5	135.19	21.05	54 792	61	8	1706
K4	133.27	26.90	55 559	56	1737	

图9-31a Q2A测量

Q2A是另外一个在文本检索应用中的典型查询，它是带有两个特性的杂志文章的搜索，其中一个特性是低的过滤因子（日期>'65/01/31' 关键字='plutonium'）。Q2A查询也可以应用于市场调查来估算具有一对给定性质的候选者的数目（如gender='M', hobby='tennis'）。这种的查询类型包括在基准程序之中的原因之一是：可以用它来测量从两个不相关的索引中合并RID列表时数据库产品的性能如何。就像我们已经学过的，DB2产品具有通过利用在9.6节中的MX类型方案来合并这样的RID列表的能力。合并了这两个索引以后，没有必要再存取数据行，因为选择列表仅包含一个count()函数，而这个函数可以通过计算最终的RID列表来完成。

然而，对于这组查询的EXPLAIN命令表明：从来都不会用一个MX方案，原因是很明显的。回顾一下，单个RID列表仅仅用了RID池的50%，而RID池的空间是磁盘缓冲池的一半。因为基准程序允许的磁盘缓冲池中有1200页，RID池含有600页空间，即大约2400KB。现在考虑谓词K2=2。我们希望找到满足这个谓词的500 000行，并且500 000个RID占用2000KB，每一个占4个字节——多于RID池的2400KB的50%。所以在一个MX类型的方案中不能得到谓词K2=2的结果，因此必须使用其他方案。

对于Q2A组的EXPLAIN命令显示DB2对于不同的查询采用两种不同的方案。在 $KN \neq K4$ 时，DB2在一个匹配索引扫描中（ACCESSTYPE=1）使用带有更小的过滤因子的谓词（ $KN=3$ ）来检索数据中的行，然后测试每一行以解决其余的谓词，计算满足两个条件的行。然而，在 $KN=K4$ 情况下，我们需要使用一次表空间扫描（ACCESSTYPE=R）来解决查询。在图9-31a的最后一行中 $KN=K4$ ，我们可以看到执行了1737次顺序预取I/O和56次随机I/O。读的页面的总数为55 559页（注意在图9-28中，NPAGES=55 556）。顺序预取I/O读 $(55\ 559 - 56) = 55\ 503$ 页，我们可以计算出每一顺序预取为 $55\ 508 / 1737 = 32$ 页。我们的经验规则表明，对于55 503S所需要的执行时间是 $55\ 503 / 400 = 138.76$ 秒，56R读加入相对不重要的1.4秒。显然，与测试的133.27秒时间相比，这个结果稍微超过了5%，即 $138.76 + 1.4 = 140.16$ 秒。我们认为在这个受I/O限制严格的查询中CPU的26.90秒与I/O的时间完全重叠。

现在考虑其他随机产生的又在Q2A查询组中被引用的列上的谓词 $KN=3$ ，其中， $KN \in \{K100K, K10K, K1K, K100, K25, K10, K5\}$ 。图9-31a相应行的随机I/O和顺序预取I/O的测量对应于扫描适宜索引的资源，而且同Q1的INDEXONLY测量很相似。（例如，在图9-31a中计算谓词 $K5=3$ 的资源是8次顺序预取I/O和61次随机I/O。在图9-30中，计算谓词 $K5=2$ 的索引I/O资源是9次顺序预取I/O和46次随机I/O。）在图9-31a中出现的列表预取I/O单独地用作存取需要的行。在 $K100K=3$ 的情况下，大约有10行被检索到——有可能在分散的磁盘页面上——只有一次列表预取I/O。一次列表预取I/O可以存取最多32页，但是在这种情况下，它只能存取10页。类似地，在 $K10K=3$ 的情况下，大约有100行被检索到——可能在100个不同的磁盘页面上——需要4次列表预取I/O。考虑在 $K100=3$ 的情况。检索到的行数大约为10 000。为了看到检索到的磁盘页面的数量，我们可以将其看做10 000支飞镖随机投入55 556个磁盘页面中的问题，同时问有多少页面被击中了。公式[8.6.4]给出：

$$E(\text{被击中的页面的个数}) = M(1 - e^{-N/M}) = 55\ 556(1 - e^{-(10\ 000/55\ 556)}) = 9152$$

现在计算要拾取9 196磁盘页面时的列表预取I/O次数，我们得到 $\text{CEIL}(9\ 152 / 32) = 286$ ，很接近于执行287次列表预取I/O。

MX类型方案不能用于这个查询组，这个事实是一个严重的问题。我们可以组合索引不必到达数据，对于Q2A最大的执行时间也许可以与Q1的对于K2的时间加上在Q1中取出 KN 的RID列表的时间进行比较，最大不会超过20秒。但是，我们看到的却是很长的执行时间。注意在大多数情况下，DB2有足够的RID池空间来处理Q2A查询（最后一种情况也许不适合），但是条不灵活的规则使得它无法利用这种优势。这不是我们从一个查询优化器期望的行为。DB2仍然有很多灵活性，从这里我们可以得到的一个教训是：DBA应当总是计划一个尽可能大的缓冲池，而且，如果有必要订购大一些的内存。

3. Q2B查询

Q2B查询具有如下形式：

```
For each KN ∈ {KSEQ, K100K, . . . , K4}
    select count(*) from BENCH where K2 = 2 and not KN = 3;
```

对于这个查询组的测量，参见图9-31b。

Q2B是Q2A的一个变型，而它是一群商业用户最初对集合查询基准程序进行测试时提出的。它代表了一种查询类型，而这种类型可以找出在它被高效执行的情况下用法（204型用户通常可以在小于1秒的情况下得到对于这个查询的一个响应）。在DB2的情况下，很多问题

来自于高效的索引的使用。我们想要看到查询优化器执行一次非匹配索引扫描以得到对于谓词 $\text{not } \text{KN}=3$ 的一个RID列表，然后用谓词 $\text{K2}=2$ 执行MX过程，这样就可以避免存取所有的表中的行。但是，我们不能够抽取对于 $\text{K2}=2$ 的RID列表，就像我们在Q2A中讨论的那样，这是因为RID池大小的限制，即使在RID池上没有限制，另外一个因素排除了MX处理，因为DB2不会执行一个在 KN 上的非匹配索引扫描来抽取一个RID列表（定义9.6.1的RID池规则2）。排除了 KN 索引，在谓词 $\text{K2}=2$ 上的一个简单索引扫描没有足够的过滤力量来除去考虑中的任何磁盘页面（每页有17行），因此查询优化器对于Q2B组的所有查询选择一次表空间扫描。测量结果可以同Q2A的K4行进行比较。

KN	执行时间 (秒)	CPU时 间(秒)	获取页面 请求	随机I/O 次数	顺序预取 I/O次数	列表预取 I/O次数
KSEQ	142.73	30.07	55 569	56	1737	
K100K	137.83	30.24	55 568	63	1737	
K10K	137.45	30.12	55 569	47	1737	
K1K	140.33	30.42	55 569	59	1737	
K100	141.73	30.17	55 569	59	1737	
K25	144.65	30.25	55 569	60	737	
K10	139.72	29.82	55 569	53	1737	
K5	142.73	30.07	55 569	56	1737	
K4	135.29	29.13	55 562	48	1737	

图9-31b Q2B测量

4. Q3A查询

Q3A查询具有如下形式：

```
For each KN ∈ {K100K, K10K, K100, K25, K10, K5, K4}
  select sum(K1K) from BENCH
    where KSEQ between 400000 and 500000 and KN = 3;
```

图9-32a包含了这个查询组的测量。

KN	执行时间 (秒)	CPU时 间(秒)	获取页面 请求	随机I/O 次数	顺序预取 I/O次数	列表预取 I/O次数
K100K	1.84	0.09	24	8		1
K10K	2.06	0.14	110	8		4
K100	14.43	3.12	5778	200	184	
K25	14.89	3.37	5778	202	184	
K10	15.35	3.73	5778	199	184	
K5	15.07	4.17	5778	208	184	
K4	15.01	4.24	5778	197	184	

图9-32a Q3A测量

Q3A查询创建来为一个直接的市场类型的应用建模，prospects表中在邮政编码范围内（在一个给定的地理区域）并且具有其他某些特性（`hobby='tennis'`）的行必须被很详细地检查。直到现在为止，所有考虑的查询都是仅仅通过检查索引信息来取得满足条件的结果。（尽管DB2中的规则使得在Q2A和Q2B中这样做不可能，204型可以只用索引满足这些查询，同时响应的时间更快）。然而，在Q3A的情况下，集合函数`sum(K1K)`不能够在索引中满足，因为`K1K`的值不会出现于在WHERE子句中的任何索引中。（注意，图9-32a对于`KN=K1K`没有项，因此INDEXONLY情况被避免了。）所以，为了把`K1K`的值加起来，我们必须存取选择的行。尽管Q3A查询仅仅检索一个集合函数的值，但是它做了一个从每一行检索一个列值的查询的大部分工作。

`EXPLAIN`命令表明：在执行这个查询组时有两种不同类型的查询方案。对于`KN=K100K`和`KN=K10K`，匹配索引扫描是分别在`K100K`和`K10K`上执行的，并且对于检索到的行`KSEQ`上谓词被测试。图9-32a中，随机I/O用来检索索引项，列表预取I/O用来检索行，在`K100K`情况下一次预取大约为10行，在`K10K`情况下4个预取为100行。已经证明，使用随机I/O可以最多检索到100行，执行多重索引存取没有获得什么好处，从`KSEQ`抽取一个RID列表以存储一些存取的行——我们已经到达了9.6节中看到的减少返回点。

对于小基数的列，`KN ∈ {K100, K25, K10, K5, K4}`，`KSEQ`上的`BETWEEN`谓词用在一次匹配索引扫描中，对于检索到的行测试`KN`上的谓词。对于这些情况的每一种，图9-32a表明读入了5778页。这些页面中大部分代表了索引叶子页面的大约10%和计算谓词`between 400000 and 500000`的数据页面的10%，这样，我们可以从图9-28的统计信息计算为 $NLEAF/10+NPAGES/10=208+5556=5764$ 页。我们看到，用184个顺序预取I/O来检索 $5778-200=5578$ 页（在`KN=K100`情况下，对于小基数列的相对典型的I/O使用），这意味着读入顺序预取I/O的页面的平均数目为30.3页。

在这个查询组中DB2的性能极好，比204型和其他当前的产品都要好出许多，这是因为已成为DB2技巧包的一部分的独特的预取I/O能力，这由智能磁盘控制器辅助。

5. Q3B查询

Q3B查询具有如下形式：

```
For each KN ∈ {K100K, K10K, K100, K25, K10, K5, K4}
  select sum(K1K) from BENCH
    where (KSEQ between 400000 and 410000
      or KSEQ between 420000 and 430000
      or KSEQ between 440000 and 450000
      or KSEQ between 460000 and 470000
      or KSEQ between 480000 and 500000)
        and KN = 3;
```

图9-32b列出了这种查询的测量。

Q3B查询是Q3A查询的一个变型，主要是基于直接市场应用的经验表明在一个给定的地理区域中候选者不必落在单个的邮编范围中，而可能是不同范围的并。除此之外，这种查询评估查询优化器的高级性能的一种很好的查询。注意，在这个查询中由OR来连接的`BETWEEN`谓词一起构成了在Q3A中覆盖的单个范围的60%（`KSEQ`范围和聚簇数据范围的总数的6%）。

KN	执行时间 (秒)	CPU时 间(秒)	获取页面 请求	随机I/O 次数	顺序预取 I/O次数	列表预取 I/O次数
K100K	2.01	0.09	17	5		1
K10K	2.23	0.14	109	7		4
K100	7.57	2.48	714	34	12	18
K25	10.66	3.66	1934	57	7	55
K10	14.37	5.93	3068	79	14	88
K5	57.76	47.33	5430	250	173	
K4	57.77	47.58	5430	235	173	

图9-32b Q3B测量

EXPLAIN命令显示，执行Q3B查询组使用了三种不同的查询方案。就像在Q3A中一样，当 $KN=K100K$ 或 $KN=K10K$ 时，一次匹配索引扫描分别在K100K和K10K上执行，并且对于检索到的行KSEQ上的谓词被进行测试以确定这些行中的一个是否满足要求。对于这些情况的测量十分类似于Q3A的情况。对于两个最小基数的列 $KN=K5$ 和 $KN=K4$ ，EXPLAIN表明了带有ACCESSTYPE=I，ACCESSNAME=KSEQX和MATCHCOLS=0的一个方案；换句话说，在KSEQ上的一次非匹配索引扫描。接着要发生的是检查从400 000到500 000范围内的KSEQ索引值，对于每一个值KSEQ谓词被测试以确定它们中的一个满足条件。对于结果RID，来源于基准(BENCH)表的行被读入(使用顺序预取I/O，因为数据由KSEQ聚簇)，剩下的 $KN=3$ 谓词被测试。我们可以看到在本查询中使用比Q3A结果更为少的顺序预取I/O的数目，这是因为检索到更少的行。在Q3B中减少的数目比Q3A超过了60%，这个事实没有被完全解释。注意，在这两种情况下，所用的CPU资源比在Q3A中使用的CPU资源大许多。这似乎来源于在所有非匹配索引扫描中的索引中获得的值上测试许多BETWEEN谓词的必要性，同时表示了一个严重的资源代价。

对于中等基数的列， $KN \in \{K100, K25, K10\}$ ，MX处理对于来源于在KSEQ上的不同

ACCESSTYPE	MATCHCOLS	ACCESSNAME	PREFETCH	MIXOPSEQ
M	0		L	0
MX	1	K100X	S	1
MX	1	KSEQX	S	2
MX	1	KSEQX	S	3
MU	0			4
MX	1	KSEQX	S	5
MU	0			6
MX	1	KSEQX	S	7
MU	0			8
MX	1	KSEQX	S	9
MU	0			10
MI	0			11

图9-33 对于Q3B的EXPLAN方案，在 $KN=K100$ 情况下

BETWEEN谓词的RID列表作并操作，然后同来源于谓词KN=3的RID列表的结果做相交操作。例如，图9-33给出了KN=K100时的计划。在图9-32b中，你应当假设顺序预取被用来存取KN和KSEQ的索引值，同时列表预取用来存取数据页面。对于这个查询组，在含有最小基数的列的两种情况下使用非匹配的索引扫描，这也许是DB2查询优化器犯了一个错误，这个可以根据这些情况中CPU使用中的大步跳跃进行判断。然而，它的性能同其他的大多数产品相比是相当好的。

6. Q4A和Q4B查询

两种变型查询组Q4A和Q4B具有如下形式：

```
select KSEQ, K500K from BENCH
  where <constraint with 3 (Q4A) or 5 (Q4B) ANDed predicates>;
```

在Q4A组的查询中有三个谓词，Q4B系列的查询中有五个谓词，它们都从图9-34的十个谓词的序列中进行选择。其中右边的一列是每个谓词的过滤因子。

序号	谓词	FF	序号	谓词	FF
1	K2 = 1	1/2	6	K4 = 3	1/4
2	K100 > 80	1/5	7	K100 < 41	2/5
3	K10K between 2000 and 3000	1/10	8	K1K between 850 and 950	1/10
4	K5 = 3	1/5	9	K10 = 7	1/10
5	(K25 = 11 or K25 = 19)	2/25	10	K25 between 3 and 4	2/25

图9-34 对于Q4组，带有过滤因子FF的谓词序列

查询中的谓词按照1~10的顺序依序选择，给定一个起始点后，按照顺序往下，到达末端后，由谓词10回到谓词1。例如，一个带有三个谓词的Q4A的查询，它起始点是5，那么这个查询拥有谓词的范围在5~7（也就是{5, 6, 7}），查询具有如下形式：

```
select KSEQ, K500K from BENCH
  where (K25 = 11 or K25 = 19) and K4 = 3 and K100 < 41;
```

带有五个谓词并且起始点为7的Q4B组中的一个查询，它的谓词在范围7~1中（也就是{7, 8, 9, 10, 1}），具有如下的查询形式：

```
select KSEQ, K500K from BENCH
  where K100 < 41 and K1K between 850 and 950 and K10 = 7
    and K25 between 3 and 4 and K2 = 1;
```

在Q4A和Q4B中测量的这类查询是具有典型性的查询，它们来源于文档检索（例如，通过关键字、它们所在的期刊、出版周期来限制查询的文章）或者直接的电子邮件应用（例如，在一个给定的薪水范围内、性别、爱好、地理位置等中检索候选人的名字和地址）的最终形式查询。图9-35a和图9-35b分别给出了Q4A和Q4B查询组的测量结果。

EXPLAIN告诉我们Q4A包含的所有查询使用索引扫描或者MX处理。图9-34中的某些谓词不能用来索引，比如谓词(1)FF=1/2，谓词(7)FF=1/5和要求非匹配索引扫描的谓词(5)，在RID的处理中不支持这么做，否则一个MU步骤能获得FF=2/25。此外谓词(6)(FF=1/4)只在包含谓词5~7的查询中使用，因而没有可以索引的谓词（因为谓词5和7已经被排除在外）。如同在9.6节中解释的那样，为了减少返回的行数，未使用过的谓词将不被使用。所有的查询计划使用那些

在给定范围内能够使用索引的谓词，这么做简化了那些包含谓词1~3的查询些在谓词2和3上MX处理(谓词(1)从不使用)；包含谓词5~7的查询通过在谓词(6)上的一个匹配索引扫描得以解决，而这些查询也是谓词(6)唯一被使用的地方，谓词5和7已经被排除在外；最后，包含谓词8~10的查询在所有这三个谓词上执行MX处理，因为其中没有任何一个谓词被排除在外。

谓词序取范围	执行时间(秒)	CPU时间(秒)	获取页面请求	随机I/O次数	顺序预取I/O次数	列表预取I/O次数
1~3	109.04	32.29	17 321	102	14	531
2~4	62.10	26.91	4409	155	22	121
3~5	107.29	16.50	17 112	111	14	525
4~6	134.55	27.02	54 802	72	8	1706
5~7	156.67	39.47	55 834	126	1747	
6~8	127.00	30.42	47 378	73	5	1477
7~9	83.07	14.94	9393	115	11	287
8~10	22.74	10.69	1097	154	16	25

图9-35a Q4A测量

谓词序取范围	执行时间(秒)	CPU时间(秒)	获取页面请求	随机I/O次数	顺序预取I/O次数	列表预取I/O次数
1~5	63.17	21.06	4416	127	22	121
2~6	62.06	20.90	4416	165	22	121
3~7	109.02	15.72	17 377	160	24	525
4~8	109.60	15.82	17 613	180	23	532
5~9	84.30	10.48	9658	175	21	287
6~10	22.64	9.59	1092	161	16	25
7~1	22.96	9.71	1090	157	16	25

图9-35b Q4B测量

掌握了所用的谓词后，就有可能估计出表中要访问的行数并和报告的I/O次数进行印证。比如，以包含谓词8~10的查询为例，过滤因子和BENCH表中的行数目的乘积为 $(1/10)(1/10)(2/25)(1\ 000\ 000)=800$ 。因为没有其他谓词存在，因此这也是获取的行总数，从附录D中我们可以看到确切的数目中785。顺序预取通常用来访问索引数据，列表预取通常用来访问数据页面。因为将要获取的785个行位于将近785个数据页面上，因此它们可以通过 $\text{CEIL}(7850/32)=25$ 个列表预取放入内存中，具体数字参见图9-35a的8~10。

更多谓词可以从Q4B的5个谓词中衍生出来，因此我们期望稳定的MX处理吞吐量和较小的应答集，EXPLAIN告诉我们Q4B包含的查询都在三个谓词上使用MX处理，除了谓词(5)外，这些谓词都具有较小的过滤因子，而谓词(5)要求非匹配索引扫描或者MU处理，因此从未使用。比如，包含谓词4~8的查询和谓词(4), (6), (7), (8)四个谓词中的三个上执行MX处理，它们的过滤因子分别为 $1/5$, $1/4$, $2/5$ 和 $1/10$ ，选择其中最小的三个为(4), (6), (8)。再举一个例子，包含谓词7~1的查询有(7), (8), (9), (10), (1)几个谓词供选择。选择其中三个具有较小的过滤因

子的谓词为(8).(9)和(10)，这和包含谓词8~10的Q4A相同。因此查询Q4A和Q4B的I/O开销近似相等。然而，Q4B的查询结果包含的行数目要比Q4A(785个元组)少，这是额外的两上谓词(1)和(7)的过滤因子带来的影响，结果大约是 $(1/2)(2/5)(785)=157$ 。在附录D中，我们可以看到确切的数目是152。

7. Q5查询

Q5查询具有如下形式：

```
For each pair (KN1, KN2) ∈ { (K2, K100), (K4, K25), (K10, K25) }
    select KN1, KN2, count(*) from BENCH
        group by KN1, KN2;
```

图9-36给出Q5查询组的测量结果。

KN1, KN2	执行时间 (秒)	CPU时间 间(秒)	获取页面 请求	随机I/O 次数	顺序预取 I/O次数	列表预取 I/O次数
K2, K100	242	219	55 896	81	1737	
K4, K25	230	208	55 802	34	1737	
K10, K25	248	223	55 903	87	1737	

图9-36 Q5测量

这个查询组是一种称为交叉表(crosstab)应用的代表，它用在决策支持系统中来计算在人群中一种因素对于另外一种因素的影响。例如，一个百货商店有一张顾客表(customers)，其中的列含有诸如爱好(hobby)和收入阶层(incomeclass)，还有在各种商品的购买水平(1~10，像收入阶层一样)。例如，在运动装店的sportsw_purchases。对于Q5系列的交叉表分析可以用来观察在顾客的爱好和他购买商品的种类之间是否存在某种关联：

```
select count(*) from customers
    group by hobby, sportsw_purchases;
```

如果某些爱好(如航行)表明在高层次的商品方面有很高的注意力，那么给这群人发信告诉它们即将到来的运动装店的销售就是一个很好的选择。

用EXPLAIN命令可以看出在Q5组中的查询以两步方案完成。第一步(PLANNO=1)，一次表空间扫描(ACCESSTYPE=R)扫描所有数据页面，然后抽取出列对(KN1, KN2)放入一个临时表。图9-36没有表明完成写入临时表的I/O的数目，因为它们没有被报告。(页面的写通常不会作为方案的一部分而发生，而是发生在缓冲池的空间用完之后，必须把改变的页面写出到磁盘以腾出空间。然而，这个数据的页面仅仅是临时使用，也许会避免完全被写入到磁盘。)在第二步(PLANNO=2)，在列KN1和KN2上执行这个临时表上的排序步骤(METHOD=3, SORTC_GROUPBY=Y)。所有相同的结果被收集在一起，来自于这个排序的输出中，每一个相同类型的(KN1, KN2)对的数目被累加，并且放入答案集中。我们可以在图9-36中看到一次表空间扫描的标记，1735次顺序预取I/O，这就是我们在以前看到的。CPU时间构成了这些查询耗时的一个很大的比例。

8. Q6A查询

Q6A和Q6B查询组测量两个表的连接。Q6A具有如下形式：

```

For each KN ∈ {K100K, K40K, K10K, K1K, K100}
    select count(*) from BENCH B1, BENCH B2
        where B1.KN = 49 and B1.K250K = B2.K500K;

```

图9-37a给出了Q6A查询组的测量。

KN	执行时间 (秒)	CPU时 间(秒)	获取页面 请求	随机I/O 次数	顺序预取 I/O次数	列表预取 I/O次数
K100K	0.95	0.10	65	12		1
K40K	1.00	0.14	115	7		1
K10K	3.75	0.29	393	22		4
K1K	16.05	1.68	2313	15	36	31
K100	83.31	13.35	111 510	45	48	290

图9-37a Q6A测量

尽管在这个查询中BENCH表扮演了两个角色B1和B2，对于存在两个不同表的情况，并不能得到什么优势，只在缓冲区驻留略有提高。EXPLAIN告诉我们Q6A组的所有查询都以执行一个嵌套循环连接的两步方案来完成。在第一步（PLANNO=1）我们在KN上使用一次匹配索引扫描以抽取出行的RID列表，而行是从外表抽取出来的（TABNO=1，也就是满足B1.KN=49的行）。对于在外循环从B1中检索到的行，B1.K250K有一个固定的常数值，我们用K来代表。在第二步嵌套循环连接中（METHOD=1），假设来自外表B1的一固定行，检索在B2中满足B2.K500K=K的行的数目。对于这一点要使用在K500K上的一次匹配索引扫描，而且因为只有数目被检索，这一步具有INDEXONLY=Y。注意，对于在外表中找到的每一行，我们希望在内表中可以发现两行，因为条件B2.K500K=K对于B2中大约两行是正确的，只要K存在于1, 2, ..., 500 000的范围内；又因为K从B1.K250K中获取，当然就是这种情况。因此，对于KN=K100，我们希望在外表中发现10 000行，在匹配的列连接之后大约找到20 000行，这些列决定了来自于内表在最终答案中的行的总数。从附录D我们可以看到检索到的精确的数字是19 948。

在这种情况下，KN=K100，在外循环必须要存取来自于B1的大约10 000行。就像我们先前在分析Q2A时计算的那样，大约有9152个数据页面，读入的页面由290次列表预取I/O完成，剩下的I/O中的大多数都是为了执行最后的计数而检索从K500K索引上的叶子层的项。

9. Q6B查询

Q6B查询具有如下形式：

```

For each KN ∈ {K40K, K10K, K1K, K100}
    select B1.KSEQ, B2.KSEQ from BENCH B1, BENCH B2
        where B1.KN = 99 and B1.K250K = B2.K500K and B2.K25 = 19;

```

对于查询Q6B组的测量参见图9-37b。

除了对表B2有新的限制（B2.K25=19）以外，查询Q6B和Q6A类似，事实上，从两个表中的列检索的是数据而不是计数值。EXPLAIN命令显示嵌套循环连接（METHOD=1）被用于Q6B组的KN ∈ {K40K, K10K, K1K}的查询中，而当KN=K100时应用归并连接（METHOD=2）。

KN	执行时间 (秒)	CPU时 间(秒)	获取页面 请求	随机I/O 次数	顺序预取 I/O次数	列表预取 I/O次数
K40K	2.66	0.27	161	97		1
K10K	11.03	0.91	720	418		4
K1K	77.27	6.68	5164	2842	36	31
K100	191.59	27.59	38 646	86	67	1190

图9-37b Q6B测量

我们以嵌套连接查询 (METHOD=1) 过程作为讨论的开始，该方案十分类似于用于Q6A组中的所有查询的方案。在KN上的一次匹配索引扫描用来取出从外表中得到的行，其中B1.KN=99。对于检索自B1的每一行，我们用K来表示B1.K250K这个固定的常数值。在内循环中，不从K500K索引上检索一个计数值，而是在B2中检索满足条件B2.K500K=K的行。通过测试条件B2.K25=19来对这些行进行限制，结果行同来自于外表B1的行连接。注意，对于在外表中找到的每一行，我们希望在内表中找到满足条件B2.K500K=K的行，它们大约为两行，就像在Q6A中讨论的那样。然而，这次只有当B2.K25=19时结果行才会存在于答案集合中，因此在连接中的整个行的数目缩减到原先的1/25。在KN=K1K的情况下，我们期望在外表B1中检索到1000行并且在内表B2中检索到2000行，但是在最后的连接中仅有 $(1/25) \times (2000) = 80$ 行。就像我们在附录D中见到的那样，精确的数字是81行。注意到，在图9-37b中，这种情况的I/O的数目同在Q6A中发生的I/O数目几乎是相等的，只是多了2827次随机I/O。我们期望通过谓词B2.K500K=19在表B2中存取行，对于在外表中找到的1000行中的每一行需要三次I/O (一次是索引页面，两次是数据页而)。毫无疑问，比数目2827小一些是因为缓冲命中了K500K索引叶子页面。

在Q6B情况下，其中KN=K100，发生了一次归并连接。在第一步，B1中满足条件B1.K100=99的所有行被在K100上的一次索引扫描选择，并且投影到两列的行上(B1.KSEQ, B1.K250K)，放置于一个临时的中间表IT1中。类似地，在B2中满足条件B2.K25=19的行被在K25上一次索引扫描选择，并且投影到行(B2.KSEQ, B2.K500)，放置于临时表IT2中。表IT1和IT2被排序，IT1按B1.K250K来排序，IT2按B2.K500K来排序。使用图9-21的向前移动的游标过程，IT1和IT2在B1.K250K=B2.K500K上进行连接。考虑I/O资源，我们可以看出在B1.K100上的索引扫描要求一次顺序预取I/O来检索相应的索引，然后用许多列表预取I/O检索大约10 000行。这10 000行存在于很多页面上，这些页面由Q2A中的讨论来计算， $58\ 824(1 - e^{(-10\ 000/58\ 824)}) = 9196$ 。这样需要用来存取这些行的列表预取I/O为CEIL(9196/32)=288。在B2.K25上的索引扫描需要大约两次顺序预取I/O以检索相应的索引，并需要许多列表预取I/O用来检索大约40 000行。检索这些行大致需要 $55\ 556(1 - e^{(-40\ 000/55\ 556)}) = 28\ 514$ 页。这意味着我们将需要CEIL(28 514/32)=892次列表预取I/O。对于列表预取的总数目，我们可以计算为 $288 + 892 = 1180$ ，很接近于测量的值1190。

9.11 性能价格比评估

集合查询基准程序以每分钟每个查询的美元代价(\$/QPM)为硬/软件平台提供了一种估计方式。因为所有的平台都提供相同的数据和对于相同的查询答案作出响应，而提供这些查

询结果的美元代价被认为是一个相当好的量度。一个平台的QPM代价是很容易计算的。在基准程序中有69个查询，所有都具有相同的权重。如果我们测试基准程序的所有查询的执行时间为T分钟，那么我们以每分钟 $69/T$ 个查询的速度在运行。加上在9.10节中的对于所有查询的执行时间，我们得到4492.24秒，即74.87分钟，因此QPM的值是每分钟 $69/74.87 = 0.9216$ 个查询。

系统的美元代价代表了5年期限内的硬件、软件和维护的费用。9.10节测量的查询的平台是IBM 9221/170型，运行于MVS XA2.2操作系统之上的DB2版本2.3。对于这个系统的5年价格是基于在图9-38中给出的从1993年十月起的零售价格。DASD（IBM的磁盘名称）的价格是一个例外，代表过去的价格，因为这样的设备IBM已不再销售。注意，每个月的维护费用在拥有硬件的第一年内是免费的，因此只有48个月需要付费。软件通常是不能够被购买的并且前2个月是免费的，因此要按月为软件使用许可付费（对于MVS操作系统有一个少量的初始费用）。

经过测试，DB2系统的总的美元代价为 $\$296\ 705 + \$912\ 300 = \$1\ 209\ 000$ 。这支持了QPM的值为0.9216，同时我们通过美元代价被QPM数值除计算最后的比率。对于集合查询基准程序的在这个平台上的最后比率为 $\$1\ 209\ 005/0.9216 = 1\ 311\ 854/\text{QPM}$ 。

标价的项目	出售价格	每月花费(5年)		
		费用	每月	总计
IBM 9221 170型	\$252 350	\$775	48	\$37200
辅助硬件				
1通道组	\$35 450	无		
密封框架	\$3820	\$4	48	\$192
DASD控制器3880-E23	\$3000	\$158	48	\$7584
DASD设备3380-AE2	\$1500	\$256	48	\$12 288
MVS许可	\$585	\$10 662	58	\$618 396
DB2许可	无	\$3675	58	\$213 150
DB2PM许可	无	\$405	58	\$23 490
总共	\$296 705			\$912 300

图9-38 对于DB2系统的价格计算

1. 执行时间同CPU时间比率比较

我们在上面的估算中测试了执行时间。然而，这并不能得到多用户系统时的正确评估。假设我们需要10分钟在某个平台上运行所有的集合查询基准程序的查询，但是做这件事仅仅使用1分钟的CPU时间——更长的执行时间在于对于I/O的等待。如果我们有许多用户在等待查询的结果，难道我们不可以通过对CPU进行时间共享使一些用户重叠，从而可以在每分钟内得到更多的查询吗？答案是肯定的。如果我们有几个磁盘来保留数据，那么磁盘I/O就不会成为瓶颈，同时在CPU上重叠的查询能够发现重叠的磁盘以检索需要的数据。一旦给出了这些，在估计费用方面，使用CPU时间比执行时间显得更有意义。

采用这步碰到的唯一问题是集合查询基准程序已经以一个单用户基准程序运行了。如果我们在系统的多个磁盘上同时运行多个用户，我们将以这种方式进行概括，而且更为正确，

但是我们不能绝对相信在并发用户查询之间没有某种系统干扰，这就使得这种多用户的情况成为一种幻想。对于这种干扰没有任何理论上依据——在这里，没有在第10章中更新事务时碰到的行上的排它的锁和其他复杂情况。但是，当我们要得出在系统中使用所有的CPU时间这个结论时，我们必须要小心。一方面，CPU资源的利用几乎接近于100%，用户形成越来越长的队列来等待CPU服务，这对响应时间有一个负面的影响。在一个合理的平衡的产品系统中我们希望CPU的利用率为90%。注意，如果我们在DB2平台上计算集合查询基准程序的69个查询的执行时间，我们可以得到74.87分钟，但是我们计算整个CPU时间，我们可以得到24.77分钟。这是三比一的差距，还没有大到会引起很大的关注。如果并发查询可以在一个多用户系统上没有干扰地执行（我们从其他的测试中可以有理由相信这里是DB2测试的情况），那么多用户代价也许可以由1 311 854\$/QPM降到只有500 000\$/QPM。在某一点上，考虑一个多用户测试是适合的，看查询是否在CPU上有效地重叠也是适合的。

2. 定制比率

重新理解集合查询基准程序的测试来在一个特定的查询负荷上来估算测试平台的行为是可能的。基准程序查询组已经被选择来跨越查询工作的大多数类型，对于一个来自于基准程序的一个特定的查询，给出一个定制的工作负荷是可能的。如果被估算的工作负荷包含了在集合查询系列中没有包含的查询形式，就需要一些复杂的灵活性。带有冲突数据存取的并发更新事务的一个工作负荷不能够以这种方式建模。我们将会遇到更新事务的一个特定的工作负荷的基准程序，即第10章中的TPC-A基准程序。

假定我们要确定一系列的权值， W_i 来代表基准查询 Q_i 的相对工作负荷频率，其中*i*从1到69，同时假定所有的权值都是非负的（有些可能是0）并且相加为1.0。集合查询基准程序的规则说所有查询代价必须详细地报告，因此我们总是可以得到执行时间、CPU时间和对于每一个查询的各类I/O。现在，我们假定查询 Q_i 需要 T_i 执行时间来完成，我们可以通过如下公式来计算加权的执行时间：

$$E = \text{一个加权查询的执行时间} = \sum_i W_i T_i$$

权重 W_i 之和为1.0，因此数值E可以被看做是在基准程序中所有查询效果的总和所对应的一个简单查询所需要的以分钟为单位的执行时间。所以，每分钟的查询数目 Q_m 对于这样的权以 $1/E$ 给出。如果C是在基准程序运行的平台上的美元代价，那么以\$/QPM为单位的定制比率通过C被 Q_m 除而得到，或者等价地通过C被E乘得到。

$$\text{最终的权值的比率} = C \times E = \sum_i W_i T_i \quad (\text{以$/QPM为单位})$$

为了测试这是否有意义，注意一个较大的比率代表较低的性能，这是因为代价上升或QPM值下降。可以确定地说，如果在最终的权值的比率中的平台价格C上升，就是说，因为卖主提高了平台的价格，一个更大的费用反映了一个更低的性能价格比。如果含有一个正的权值的 T_i 值中的一个上升了，表明某个查询占用更长时间。

注意，在最近的子节中讨论的用CPU时间代替执行时间的所有考虑，在定制负荷上仍然有效。执行一个单独的测量集的优点在于：通过在一个工作负荷中使用不同的查询权值对于查询进行定制是可能的。在一个多用户测试中，通常的过程是选择一个特定的工作负荷，并且测试整个的执行时间。对于不同查询的单独测试是不能获得的，所有定制的比率得不到。然而，如果在并发查询中的非干扰假设成立，我们可以在限定的环境中得到一个近似的定制多用户的比率。

3. DB2和ORACLE之间索引用法的不同

对于所有数据库产品的每一个最新版本，在索引的使用上彼此有不同的限制。就像在版本7中，ORACLE除了等价匹配外不能组合任何谓词，同时对于谓词`c1 is null`也不能使用一个索引。通常，DB2的2.3版本在那时具有在查询优化方面的最先进的技巧包，这种说法是公正的，但是最近的发展可能会改变这一点。

推荐读物

大多数数据库产品的标准SQL参考手册在理解查询优化方面提供的引导很少。然而，《DB2 Administration Guide》(推荐读物[1])和《DB2 Applications Programming and SQL Guide》(推荐读物[2])对于在本章中提到的内容有相当好的介绍。DB2 UDB在推荐读物[3]和推荐读物[4]中有相应的介绍。ORACLE在《Server Administrator's Guide》(推荐读物[7])和《Server Tuning》(推荐读物[8])中都有相应的内容。在《The Benchmark Handbook》(推荐读物[6])一书中提供了集合查询基准程序的更为详细的细节。Goetz Graefe的关于查询评估技术的论文(推荐读物[5])解释了几种我们这里没有提到的基本原理，尽管在现在的商业数据库系统中这些原理尚未实现，但是在不久的将来可能会出现。

- [1] *DB2 for OS/390 V5 Administration Guide*. See Chapter 2, "Designing a Database," and Chapter 5, "Performance Monitoring and Tuning." Document no. SC26-8957-02. IBM. Available at www.ibm.com/db2.
- [2] *DB2 for OS/390 V5 Applications Programming and SQL Guide*. In particular, see Chapter 6, Section 4, "Using EXPLAIN to Improve SQL Performance." Document no. SC-26-8958-02. IBM. Available at www.ibm.com/db2.
- [3] *DB2 Universal Database Administration Guide: Design and Implementation*. V6. See Document no. SC092-2840-00. IBM. Available at www.ibm.com/db2.
- [4] *DB2 Universal Database Administration Guide: Performance*. V6. In particular, see Chapter 6 "SQL Explain Facility." Document no. SC09-2840-00. IBM. Available at www.ibm.com/db2.
- [5] Goetz Graefe. "Query Evaluation Techniques for Large Databases." *ACM Computing Surveys* 25(2), June 1993, pp.73-170.
- [6] Jim Gray, editor. *The Benchmark Handbook for Database and Transaction Processing Systems*, 2nd ed. San Mateo, CA: Morgan Kaufmann, 1993. Available at <http://www.benchmarkresources.com/handbook/>.
- [7] *ORACLE8 Server Administrator's Guide*. Redwood Shores, CA: Oracle. <http://www.oracle.com>.
- [8] *ORACLE8 Server Tuning*. See Chapter 21, "The EXPLAIN PLAN Command." Redwood Shores, CA: Oracle. <http://www.oracle.com>.
- [9] *ORACLE8 Server SQL Reference Manual*. Volumes 1 and 2. Redwood Shores, CA: Oracle. <http://www.oracle.com>.

习题

在书后“习题解答”中有答案的习题用符号•标注。

在涉及到对于I/O的执行时间的习题中，通常你应当假设索引目录页面是常驻内存的，而索引叶子页面却不是，除非特别说明。当索引I/O加起来不到总数的10%时，在你的答案中忽略它。然而，你应当很小心地包含对于索引I/O的计算，说明为什么你认为它们是不重要的。下面的习题假定DB2体系结构(有特别说明的例外)，也就是说4KB的页面、在一棵B树的叶子层压缩等等。

9.1 DB2被设计成为在一次索引扫描的一个部分使用顺序预取I/O（就像从索引中读入的情况）而不是另外一种（在例子9.3.4中的非簇聚的情况下读入数据页面，它是基于索引中找出的RID的情况下）。在本题中，我们将对磁盘控制器不把回答以后的页面请求的磁道放入内存以便可以利用DB2的灵活性的情况和缓冲区一次读入一条完整的磁道（我们认为是32磁盘页面，每页4KB）的情况相比较。

- (a) •对于组成一次I/O的成分利用图9-6的值来计算对于一次32页顺序I/O需要的时间（假设依次读入32页需要一次寻道时间和一次旋转等待并且扩展了传输时间）。在顺序I/O假设下，写出每页多少秒的结果。这就是当使用DB2顺序预取或者当磁盘控制器从一个新道读入单个磁盘页面时发生的事情。
- (b) 计算例9.3.3的唯一匹配查询执行的时间：
 - (i) 假设对于每一检索的页面，磁盘驱动器把一条完整的磁道读入磁盘缓冲区。再假设从一个唯一的匹配到另外一个没有缓冲节省，因为对于这个缓冲区有太多的不同的页面以至于没有任何帮助。
 - (ii) 假设是单页的随机I/O（对于每一I/O重复图9-6中的所有成分）。
- (c) •对于例9.3.5再一次计算花费的时间。
 - (i) 在8页预取的假设下。
 - (ii) 在单页随机I/O的假设下。

在这种情况下，8页顺序I/O可以提高执行时间的性能。
- (d) 考虑在(b) (类型 (b)) 中提到的那个查询的一些数目W1的一个工作负荷，和在(c) (类型 (c)) 中提到的查询的W2。当需要预取I/O时，类型(b)的查询有一个运行时间的缺点，而当需要随机I/O时，类型(c)查询有一个缺点。
 - (i) 选择权值W1和W2，使得对于两种选择（随机I/O或者8页预取），在工作负荷中的I/O的总的执行时间是相同的。我们假设类型(b)的查询比类型(c)的查询出现得更频繁。
 - (ii) 现在，当我们对于不同的查询在预取I/O和随机I/O的选择上具有灵活性时，计算每一个查询的总的执行时间。

9.2 这个问题重复8.6节给出的数学推导。如果学过基本概率课程，你应当毫无困难地得出结果。你可以通过编写一个带有随机数发生器的程序模拟来产生你的结果。

- (a) •考虑例9.3.3的唯一匹配索引，它带有一个employees表，这个表有100 000行，每行400字节。假设数据页面和e_idx索引（10字节的项）具有PCTFREE=0，因此，有10 000个数据页面，并且有一个深度为2的含有250叶子页面的B树索引。

现在假设我们有一系列内存缓冲区，通过一个算法来确定内存中的页面，当一页面在125秒内没有被引用，那么把这页面从缓冲区中删除。假设我们的工作负荷每秒执行例

9.3.3中的那类查询的两个，并且使用一个随机选择的 eid 值，这是工作负荷中引用`employees`表或者`eidx`索引的唯一的查询类型。每一次执行这些查询中的一个，并引用索引`eidx`的B树的一个根节点，因此根节点总在内存中。注意，如果被引用的 eid 值被正确地选择来通过`eidx`叶子节点计算，那么125秒后250个叶子节点的每一个都将被引用到，它们都将驻留在缓冲区中。那种情况是不可能发生的，因为 eid 的值是随机被选择的。

- (b) • (i) 一个给定的叶子节点在任何给定的查询上被引用的概率 $P(leaf)$ 是多少？
- (ii) 提供一个不被一个给定查询引用的叶子节点的概率的公式。
- (iii) 对于一个给定的叶子节点不被在一行中查询的某个数字 N 引用的概率，给出它的公式。(提示：这里要涉及到幂)。
- (iv) 使用一个计算器，给出索引`eidx`的B树的叶子节点没有被最近的250个查询，即125秒内引用的概率，因此它不会常驻缓冲区。
- (v) 在缓冲区中出现叶子节点的期望数字是多少？
- (vi) 以 N 支飞镖投向 M 个随机空位的术语来陈述这个问题。就像在8.6节中讨论的一样。执行那一节指数形式的计算。
- (c) (i) 对于表`employees`的数据页面，重复(b)的所有步骤。
- (ii) 在最后125秒内，包括索引根页面和其他页面在内，有多少实际的数据库页面被引用？对索引叶子页面的引用有多少？对数据页面的引用有多少？
- (iii) 在那段时间，大约有多少索引叶子页面被引用？有多少数据页面被引用？(一些页面已经被引用了两次或多次，但是统计时只算一次。)
- (iv) • 说明为什么保留在内存中的叶子页面的数量要小于数据页面的数量。
- (d) 为了保留在最后125秒内所有引用的页面都在缓冲区常驻，需要出现在缓冲区中的页面的大致数量 B 是多少（包括B树和数据页面）？
- (e) 最普遍的形式LRU缓冲算法，使用某种良好定义的P缓冲页面池：LRU把每一新引用的页面放入池，同时把长期没有被引用的页面删除以腾出空间。为了保证每一被引用的页面有大约125秒的生命周期，给出一个目前必须出现的我们到目前为止处理的查询的页面数量的估算。说明你的答案。
- (f) • 假设在缓冲区中的页面被引用时导致了一次0代价的I/O，而在缓冲区外的页面的代价为1（我们不得不把它从磁盘上取回，这里是一个单元）。对于在这里假设的查询工作负荷，它的每秒I/O代价是什么？（你可以从(b)和(c)部分的结果来计算。）
- (g) 假设我们把在缓冲区中的页面的生命周期从125秒扩展到250秒。计算一下，这将如何影响本习题(b)的(iv)和(v)，(c)的(i)，(d)，(e)和(f)。我们可以做的就是通过购买额外的内存来保留更多的缓冲页面，从而减少I/O（磁盘臂）的代价。
- (h) • 如果我们恰好有251个缓冲页面，并且在缓冲区中保留`eidx`根页面和所有`eidx`的叶子页面，同时删除所有数据页面，那么每秒的I/O代价将是多少？由此可以得出这样的结论：在一个固定长度的时间内保留所有页面的想法不是最优的。

9.3 用C语言写嵌入式的SQL程序来模仿习题9.2的缓存的情形。

- (a) 输入`eidx`叶子页面的数目 N 和`employees`数据页面的数目 M 。(我们将在习题

9.2的情形中用 $N=250$ 和 $M=10\ 000$ 。) 我们用 K 代表一页在被从缓冲区删除前必须经过的未被引用的秒数。(注意, 这不是一个LRU方法。) 假设一秒钟执行两个查询, 引用某个叶子页面和数据页面。定义一个被每个查询调用的函数, 返回查询后仍然保留在内存中的不同叶子页面和数据页面的数目。使用一个随机数发生器来产生对于查询的一个叶子页面和一个数据页面的随机引用的数目。如果你对很长序列的查询结果进行平均(在一行中调用函数100次左右, 一旦系统被唤醒, 页面就被删除), 你应当得到习题9.2(b)(v)和9.2(c)中相同的效果。(注意, 写这样的程序最困难的部分是执行有效的查找看是否有一新的被引用的页面已经在缓冲区中, 如果不是, 需要确定哪一页应当从缓冲区中删除。然而, 对于这个习题编写高效的代码是没有必要的。最简单的方法就是在两个大的数组中对于所有 $M+N$ 个页面保持引用次数。) 你应当考虑用数据库中的表来存放这些数据。

- (b) 使用(a)中的 N 和 M 的值, 编写一个LRU缓存的模拟程序, 它以值为B的输入给出一个输入的固定缓冲区大小。查询函数应当返回在每个新查询的缓冲区中没有发现的页面数量(值0, 1或2), 还有从缓冲区被删除的页面的存在时间。当系统被唤醒时, 查询函数的调用者应当显示出页面存在时间开始的序列。通过看存在时间的这个列表检查习题9.2(e)的结果。把一个长系列的返回值进行平均以估算一个查询的平均I/O代价, 同时检查习题9.2的(f)和(g)的计算结果。

9.4 假设有一个含有十亿万行的表T, 每一个行为200字节, 装载时90%满, 但是对于索引是100%满。假设我们在T的(C_1, C_2, C_3, C_4)上有索引C1234X, 表T由这个索引进行簇聚。我们在T上的(C_5, C_6, C_7, C_8)上有索引C5678X, 运行EXPLAIN后, 我们发现C5678X的CLUSTERRATIO是20。假设每列—— $C_1, C_2, C_3, C_4, C_5, C_6, C_7$ 和 C_8 ——都是4字节长。

$CARD(C_1) = 100, CARD(C_2) = 200, CARD(C_3) = 50, CARD(C_4) = 10,$
 $CARD(C_5) = 10, CARD(C_6) = 1000, CARD(C_7) = 20, CARD(C_8) = 5000$

假设对于每列 C_i , 列的值从1到 $CARD(C_i)$ 均匀分布。

- (a) 计算下列的统计信息。解释你的推理。
- (i) 计算表T的CARD和NPAGES。
 - (ii) 计算 C_2, C_3, C_6 和 C_7 的COLCARD, LOW2KEY和HIGH2KEY。
 - (iii) 对于C1234X和C5678X的每一个计算FIRSTKEYCARD和对于FULLKEYCARD的最好估算。
 - (iv) 利用FULLKEYCARD和关于DB2索引压缩的合理假设(重要), 计算每一个索引的NLEAF, 然后计算每一个的NLEVELS。请表明计算的过程。

- (b) 考虑下列查询:

```
select C10 from T where C1 <= 10 and C2 between 100
    and 110 and C3 = 4;
```

- (i) 在这个查询的EXPLAIN后, 给出将在方案表中看到的相关列。
- (ii) 为了回答(b)中的查询, 必须检索的索引项的范围是什么。在其中有多少叶子页面? 我们可以使用顺序预取吗? 对于这步, 近似的执行时间是多少?
- (iii) 计算对于(b)中的查询的复合谓词的过滤因子。

(iv) 检索到多少行？解释为什么这些行不是连续的，为什么不能使用顺序预取。这个数据页面存取步骤的执行时间是多少？

(v) 指定这个索引步骤（根据(ii)和(iv)）总的I/O和执行时间。

(c) 对于两种方案中的每一个重复在(b)中相同的步骤，这两种方案在T的两个索引上提出。对于查询：

```
select * from T where C7 = 3 and C1 = 99 and C2 = 55;
```

这些方案中哪一个的I/O执行时间代价较低？

9.5 再一次考虑表T和习题9.4中的相关索引和列。

(a) 指出下列搜索条件的匹配列，同时说明如果所有谓词不匹配时，匹配会中止于它所在的位置的原因。

(i)* `select * from T where C1 = 7 and C2 >= 101 and C3 in (1,5) and C4 in (2,4,7);`

(ii) `select * from T where C1 in (1,3,5) and C2 = 6 and C4 = 7;`

(iii)* `select * from T where C1 <> 6 and C2 in (1,3,5) and C3 = 5;`

(iv) `select * from T where C1 in (1,3,5) and C2 = 7 and C3 = 6 and C4 in (3,6,8);`

(v)* `select * from T where C1 = 7 and C2 in (1,5) and C3 like '%abc' and C4 in (2,4,7);`

(vi) `select * from T where C1 in (1,3,5) and C4 > 3 and C3 = 7 and C2 = 6;`

(vii)* `select * from T where C1 = C4 + 6 and C2 in (1,3,5) and C3 = 5;`

(b) 在给出下列问题答案时，写出你的计算过程。

(i)* (a)部分中(i)查询的匹配谓词的复合过滤因子是多少？

```
select * from T where C1 = 7 and C2 >= 101 and C3 in (1,5) and C4 in (2,4,7);
```

(ii) 整个查询条件的过滤因子是什么？

(iii)* 这个查询索引查找存取的页面的数目是多少？

(iv) 假设我们利用筛选谓词，存取的数据页面的数目是多少？I/O的种类（R, S或者L）？

(v)* 执行这些I/O的总的执行时间？

9.6 在这个习题中，假设prospects表具有在图9-12和例9.6.1中定义的索引addrx, hobbyx, agex和incomex。

(a)* 说明在什么样的环境下，谓词incomeclasse=10在多重索引扫描中的一个MX步骤中可以是一个匹配谓词。

(b) 在多重索引扫描的一个MX步骤中，使用谓词age=40是否可能？说明原因。

(c)* 在多重索引扫描的一个MX步骤中，使用谓词age between 20 and 39是否可能？

(d) 在多重索引扫描的一个MX步骤中，使用谓词age between 40 and 44是否可能？

(e)* 说明为什么在一个多重索引扫描的单个MX步骤中，使用谓词age in

- (40, 43, 45) 是不可能的。
- (f) 有没有一种方法，创建一个复合谓词具有与(e)中的谓词有相同的效果而又可以用于一个MX步骤序列中。

9.7 对于下面的练习，把在9.5节中定义的索引mailx加入到习题9.6考虑的索引上去。考虑搜索条件 zipcode between 02139 and 03138 and incomeclass=10 and age=40。

- (a) • 对于这个复合谓词，考虑mailx上的一次匹配索引扫描，并在方案表中指出相应的列值。尤其，MATCHCOLS是什么？
- (b) mailx上的扫描产生的过滤因子是什么？
 - (i) 如果没有使用一个RID列表？
 - (ii) 如果使用一个RID列表说明为什么有这样的区别。
- (c) • 在(b)的(ii)中，我们可以把这个mailx的扫描看做是一个多重索引存取中的一次MX扫描。在那种情况下，我们也可以使用其他索引。
 - (i) 在incomeclass=10上的我们可以执行一个MX步骤吗？
 - (ii) 表明我们可以使用的多重索引步骤的方案表。
 - (iii) 在本情况中，整个搜索条件的过滤因子是什么？
- (d) 对于(b)的(i)中表现出来的全部方案，计算I/O执行时间。（你需要计算索引页面和数据页面的I/O时间）。确信你使用了正确的I/O类型——随机I/O、列表预取I/O或者顺序预取I/O）。
- (e) • 对于(c)的(ii)中出来的全部方案计算它的I/O执行时间。

9.8 假设prospects表有索引addrx, hobbyx, incomex, genderx（例9.6.3中），agex和mailx。

- (a) • 在多重索引存取方案中的MX步骤中，具有谓词gender='F'是否可能？
- (b) 考虑查询：


```
select * from prospects where zipcode between 02139
      and 07138 and hobby = 'chess' and age = 20;
```

 - (i) • 用一次mailx的索引扫描解决这个问题是否可能。指出上面的匹配谓词和筛选谓词。
 - (ii) 执行多重索引存取也是可能的。按照降序的过滤因子放置谓词，然后给出用作MX型方案的方案表，其中最小的过滤因子先来，我们一次尽量保留尽可能少的RID列表。
 - (iii) • 在(i)和(ii)部分检索到的行数是多少？涉及到多少数据页面？使用了哪一种I/O（R, L或者S）？在两种情况下，数据页面存取的执行时间是多少？请写出计算过程。

9.9 对于下列查询假设索引mailx, hobbyx, incomex, agex。考虑查询：

```
select * from prospects
  where zipcode between 02159 and 04158 and
    (age = 40 or age = 44) and hobby = 'tennis'
    and incomeclass = 7;
```

- (a) • 说明为什么我们不能够在mailx上简单地使用一次匹配索引扫描来解决所有这

些谓词。然而，我们可以用筛选谓词执行一次匹配扫描，筛选谓词在其中扮演了过滤因子的角色。分析这样一个方案的I/O执行时间，包括了索引I/O、检索到的行数以及它们的I/O类型。给出全部的执行时间。

- (b) 现在确定多重索引存取方案以匹配尽可能多的看起来合理的查询谓词。首先按照过滤因子增加的顺序排列谓词。计算每个谓词的RID获取的索引的I/O代价，还要计算使用每一个谓词后的过滤因子，以及谓词是否自己偿付代价。紧接着例9.6.6的模式。在这个计算的末尾，写出像图9-18那样的多步索引的方案列，并且按照你已经计算的顺序和任何一个时间使栈中现存的RID列表的数目最小。这个多步方案的总的执行时间是多少？
- (c) •(a)和(b)，哪一个占优？作为一个通常的规则，当我们有一个复合索引，它只在第一列有一个搜索条件匹配，而在复合索引的后几列是一组其他的单列索引，复合索引扫描仍然是胜利者。证明这个规则。

9.10 考虑例9.7.1、例9.7.2和例9.7.3。我们有表TABL1和TABL2，但是我们使用一些新的列和索引。有一百万行，每行200字节，每列4个字节。我们又假设在索引页面和数据页面上没有浪费的空间。我们有如下查询：

```
select * from TABL1 T1, TABL2 T2
  where T1.C6 = 5 and T1.C7 = T2.C8
    and T2.C9 = 6;
```

假定我们有索引C6X, C7X, C8X和C9X； $FF(T1.C6=\text{常数}) = 1/20$ ； $FF(T2.C9=\text{常数}) = 1/400$ ；列T1.C7的值从1到200 000均匀分布；T2.C8的值也是从1到200 000均匀分布；T1.C7和T2.C8的值没有通过行数或其他什么有任何关联。它们是互相独立的随机的。

- (a) •推导从这个查询得到的行的期望数值。
- (b) 考虑这个查询，有三种可能的方案：(i) 嵌套循环连接，其中T1是外表，(ii) 嵌套循环连接，其中T2是外表（T1同T2是不同的）(iii)归并连接。对于每一种方案计算出总的I/O时间，并且得出哪一种方案更好。在归并连接的情况下，其中排序是必要的，假定不多于50页的一次排序不需要I/O，但是多于50页时，每一页必须要写出然后再读入，使用顺序I/O写但是用随机I/O读入。（这样做的原因是由于磁盘排序算法决定的。）

9.11 在例9.8.1中，假设2000字节长的行的序列，两行占据一页，通过下列值依次排序，下面的值代表了行的最初顺序。

67 12 45 84 58 29 76 7 91 81 39 22 65 96 33 28 77 4 54 13 41 32 1 59

- (a)• 就像在例9.8.1中那样，在这些数字上执行一个两路归并排序。显示出所有的中间结果。
- (b) 重复(a)部分的排序，现在使用一个三路归并排序。
- (c) 现在假定给出的行长度是4000字节，所以一行占据一页。在这些数字上执行四路归并排序。

9.12 回答下面的有关集合查询基准程序的问题，在所有情况下都是指DB2测量。对于下面的每一个WHERE子句你期望可以从BENCH表中检索到多少行？

- (a) • K2=2 AND K10=7。

- (b) KSEQ BETWEEN 400000 AND 410000 OR KSEQ BETWEEN 480000 AND 500000。
- (c) • 图9-34中的谓词列表序列的1到4。
- (d) (i) 在Q1中，假设所有的DB2预取I/O都是顺序预取，在K2情况下，I/O花费多长时间？在K10的情况下呢？
(ii) 论证在K2情况下，DB2是与CPU相关的。
(iii) 在K10情况下，有多少页面被读入内存？给出执行该查询方案时有多少页面必须被看到的详细情况。
- (e) • 考虑查询Q2A。在32页的块中，一些情况很明显地使用顺序预取。指出一种这样的情况。顺序预取I/O要用多长时间。该种情况如何同我们已经使用的经验规则协调起来？
- (f) 考虑查询Q4B，详细细节在图9-35b中。
(i) 在条件序列5~9中使用了什么索引？
(ii) 在组合这些索引后，给出你期望从表中检索到行数。
(iii) 使用飞镖投向空位的公式来计算在(ii)中行上的页面数，把这个数目同在图9-35b中报道的放入内存的页面数做比较。
- (g) • 在查询Q6B中，在K100情况下，执行一次计算，解释你的推理，估算检索到行（被连接的）的总数。怎样把这个数目同实际检索到的数目进行比较？（你可以在什么地方发现那些数目？）

9.13 考虑下列在集合查询基准程序的BENCH表上的查询。

```
select B1.KSEQ, B2.KSEQ from BENCH B1, BENCH B2
  where B1.K100 = 22 and B1.K250K = B2.K100K and B2.K25 = 19;
```

- (a) • 计算你期望看到的检索的行数。给出你的计算过程。
(b) • 计算在两种不同的可能的嵌套循环连接上的执行时间，并说明哪种方案更好。

9.14 考虑下列在集合查询基准程序的BENCH表上的查询。

```
select B1.KSEQ, B2.KSEQ from BENCH B1, BENCH B2
  where B1.K100 = 22 and B1.K250K = B2.K250K and B2.K100 = 19;
```

- (a) 计算你期望看到的检索的行数。给出计算过程。
(b) 计算用嵌套循环连接得出的I/O的执行时间（两种嵌套循环连接的时间是一样的）和归并连接得出的I/O的执行时间。给出计算过程，并说明哪种方案更好。

9.15 考虑BENCH表和它的索引。在下面的习题中，假设表BENCH和所有的索引被以100%满的页面装载。在紧接着的计算中使用DB2的标准假设，并给出计算过程。

- (a) • 运行RUNSTATS之后，给出一个Select语句来从相应的DB2系统目录表中检索BENCH表上的行所在的数据页面的数目。
- (b) 计算你期望寻找的数据页面的数目，假设页面是100%装载满的。
- (c) • 在页面是100%满的假设下，计算你期望在K2X索引中找到的叶子页面的数目。不要忘记叶子层索引压缩。
- (d) 假设我们删除在BENCH中KSEQ=300 000的行。为了删除该行，我们可以直接进入在K2X中的索引项，这是否正确？（也就是，我们是否有一种可以在目录结构中寻找项的方法？）针对你的答案给出原因（如果可能的话，引用在文章中的文献）。

第10章 更新事务

在5.4节我们介绍了事务的基本概念。现在我们通过一些定义开始回顾一下前面的内容。

定义10.1 事务是数据库提供的一种手段，通过这一手段，应用程序员将一系列的数据库操作组合在一起作为一个整体以便数据库系统提供一组保证，也就是事务的ACID性质(本节稍后会讲到)。当组成事务的操作同时包括读操作和更新操作时，表明应用程序员希望对数据进行一致的状态转换。如果这些操作仅仅包括读操作，表明程序员只希望得到当前数据的一致视图。 ■

在标准SQL中没有Begin Transaction这样的语句；事实上当系统处理进程中没有活动的事务时，一个事务就可以开始，并且访问数据的SQL语句(如Select, Update, Insert, Delete等)被完成。而当事仍处于执行状态时，它所作的任何更新对并行用户而言都是不可见的，并且数据读不能被更新。在标准SQL中关于事务的执行有两条语句。第一条语句是Commit语句：

```
exec sql commit work;
```

程序员可以使用这条语句通知系统当前事务已经成功完成；事务所做的所有更新在数据库中将永久地保存下来并对并行用户可见。

第二条和事务执行有关的语句是Rollback语句：

```
exec sql rollback work;
```

这条语句指出当前事务执行失败；事务所做的所有修改将被撤销，被修改的数据将恢复到修改之前的版本并对并行用户再次可见。无论是系统发出的还是程序发出的回滚操作都将终止一个事务，我们通常称之为异常中止。

在5.4节中我们只是简单介绍了事务的概念，以引出在应用程序中引入Commit和Rollback语句的必要性，以及当事务发生死锁时异常中止事务的必要性。本章中我们将更深入地了解关于事务的这些概念，即便如此，本章也只是对这一复杂而重要的领域（事务）的一个介绍而已。当前，许多数据库系统的软硬件提供商就数据库的事务处理系统专门成立开发小组，甚至在不同的地点进行开发。在第8章和第9章提到的，诸如以支持有效查询处理为目标的数据库系统的那些典型特性，与支持事务处理相比还存在很大的差距，所以你在本章要做好学习更多知识的准备。

从20世纪50年代开始，人们开始用事务的概念来解决开发早期的大型数据库应用过程中系统设计所面临的一系列问题。银行的日常业务就是这样的一个例子，必须允许许多出纳员同时处理多个顾客的读取和更新。在处理这样的应用的过程中，早期的开发人员必须面对以下的问题。

(1) 产生不一致的结果

当出现以下情况时我们将如何处理？一个应用程序正在从一个账户向另一个账户转一笔钱时（这两个账户是两条不同的记录，通常处于磁盘的不同页面上）。当第一个账户已经将这笔钱从它的余额中扣除，并且已经记录到磁盘上，这时由于电源故障使得系统崩溃。（必须注

意的是两次磁盘更新操作必定有一个操作是先做的，崩溃通常发生在执行第二次磁盘更新操作之前。) 当我们试图进行系统恢复时，我们的应用程序再也不能获得当时的执行逻辑(所有的内存内容已经丢失，包括程序变量和进行流程控制的寄存器)。唯一永久的存储器在磁盘上，而应用程序只是修改了一个账户中的金额，因此破坏了两个账户的总额平衡。

(2) 并发执行的错误

如果不对并发事务的读写记录进行控制，并发事务的执行可能以很多方式互相干扰。这些干扰中的一种方式就是不一致分析。假定出纳员1正在从一个顾客的账户A向该顾客的另外一个账户B转一笔钱，而出纳员2试图将该顾客的这两个账户的余额相加进行一次信用测试。如果代表出纳员1的应用将转账的金额从账户A中扣除了，而账户B还没有加上这笔钱。这时，代表出纳员2的应用将该顾客的两个账户的金额进行了相加操作，那么出纳员2看到的将是比这位顾客的实际存款要少的金额，由此可能导致这位顾客不能通过信用测试。

(3) 关于何时更新的数据会变为持久化的不确定性

回想一下，为了减少磁盘的访问次数(I/O)，我们通常将常用的页面存放在内存中。这意味着对于一条经常要用到的记录会在内存中放置很长一段时间，比如保存银行某个分行的余额的记录。就像我们在问题1中提到的那样，当崩溃发生时我们只能找到已经写到磁盘上的内容。因此看上去似乎我们有以下两种选择，一种方法是一旦有修改操作发生就将缓冲的记录全部写回到磁盘上去(如果这样做，我们根本不能通过缓冲机制来节省磁盘I/O操作)，另外一种方式是发生修改操作时不做频繁的写回(在这种方式下，当用户要从账户中取款时会感到紧张，因为系统可能在取款信息被记录下来以前崩溃)。对于每个记录改变没有完成磁盘写操作，我们能够相信更新被记录下来了吗？

为了解决这些问题，系统分析员提出了事务的概念(尽管这些想法直到20世纪70年代才被规范化)。针对以上这些具体的问题，系统分析员将事务定义为对数据库的一串读写操作，这些操作组成一个逻辑单位。比如多个账户之间的转账操作(包括读，也包括写)，再如包括多个账户的信用检查操作(只包含一系列的读操作)。程序员决定怎样将一些读取和更新操作逻辑地组成一个事务，接着数据库系统通过提供以下四个保证来解决上面所列的各种问题。这四个保证我们称之为ACID保证或ACID性质。ACID是四个属性的缩写：Atomicity(原子性)，Consistency(一致性)，Isolation(隔离性)和Durability(持久性)。

原子性 事务的这一性质保证事务包含的一组更新操作是原子不可分的(取原子这一单词的原来意思，原子衰变被发现之前)。也就是说，这些更新操作对于数据库而言要么全做要么全不做，不能部分地完成。这一性质即便在系统崩溃之后仍能得到保证(参考下面要提到的持久性)。一个称之为数据库恢复的过程将在系统崩溃后执行，用来恢复或撤销系统崩溃时处于活动状态的事务对数据库产生的影响，从而保证事务的原子性。因此问题1——产生不一致的结果，就可以得到解决。系统在对磁盘上的任何实际数据做出修改之前都会先将关于修改操作本身的信息记录到磁盘上。当发生崩溃后，系统就能根据这些操作记录掌握当时该事务处于何种状态，以此决定是撤销该事务所做出的所有修改操作(从而撤销对数据库的影响)还是将那些未在数据库中做出真正修改的操作重新执行(这样事务就成功完成)。

一致性 这是数据库系统提供的另外一个事务属性，我们将在稍后定义。对于一致性，我们不会像ACID的其他几个性质那样突出地介绍，因为一致性在逻辑上不是独立的。事实上，一致性由事务的另外一个更基本的性质——隔离性表示。

隔离性 事务是隔离的意味着仅当两个事务不处于并发状态(即它们的操作不是交错的)时,其中一个事务对另一个事务才有影响。这一性质的另外一种称法为可串行性,也就是说系统允许的任何交错操作调度等价于某一个串行调度。串行调度的意思是每次调度一个事务,在一个事务的所有操作没有结束之前,另外事务的操作不能开始。由于性能原因,我们需要进行交错操作的调度,但我们也希望这些交错操作的调度的效果和某一个串行调度是一致的。数据库对事务提供的这一保证就能很好地解决问题2——并发执行导致的错误。就像我们在5.4节中提到的那样,商业数据库系统中对这一性质的实现是通过对事务的数据访问对象加适当的锁从而排斥其他事务的对同一数据对象的并发操作。

持久性 系统提供的这一保证要求:当事务发出提交语句后系统返回到程序逻辑时,必须保证该事务是可恢复的。例如,自动柜员机(ATM)在向客户支付一笔钱时,就不用担心丢失客户的取款记录了。也就是说第3个问题可以得到解决了。我们称数据库提供的这一保证为持久性,即事务产生的影响是持久的,哪怕系统崩溃。正如我们在讲述原子性时提到过的那样,系统通过做记录来提供这一保证。同时,在修改大量记录的情况下,为了保证事务的这一性质可以减少大量的磁盘I/O次数。系统可以在一次磁盘写操作中包括多条关于记录更新操作的记录信息,而这些记录更新操作本身可能需要对不同的磁盘页面进行操作,因而需要多次的磁盘操作才能将各条记录写到各自的磁盘页面上去。事实上,即使这些磁盘页面都已经写回到磁盘上,关于这些操作的日志信息仍然是必要的,这一点我们会在稍后做出解释。

显然,原子性和持久性对于只读事务而言是平凡满足的(由于没有更新操作)。在这种情形下,唯一值得关注的事务性质是隔离性,也就是必须保证只读事务读到的行不包含未提交事务修改的数据。如果允许读取未提交事务修改过的数据,那么就有可能产生不一致的问题,比如,当出纳员2试图对账户A(此时代表出纳员1的未提交事务已经对账户A进行了修改)和账户B(代表出纳员1的未提交事务正在对该账户进行更新操作)进行求和时,就会产生不一致问题。事务的以上三项保证(性质)——原子性、持久性和隔离性——需要进行深入的学习,因为它们之间的联系比较紧密、复杂。在10.1节我们学习了隔离性(通常也称之为可串行性)。在后面的章节我们将学习原子性和持久性。

隐含在隔离性中的事务的另外一个性质就是一致性。我们曾经在问题2中谈到不一致问题时提到过。一致性是一种以一致性规则为基础的逻辑属性,比如“在转账过程中,钱既不能多,也不能少”。这样的一条规则对于程序员来说就是一个强制的规定,也就是说执行转账任务的事务结束时不能造成账户金额的不平衡,这样程序的执行逻辑才是符合要求的(我们总是假定在满足隔离性时程序的逻辑是服从一致性规则的)。事务的一致性属性要求在事务并发执行的情况下事务的一致性仍然是满足的。然而,事实证明事务的隔离性已经足以保证这一点,因为事务的隔离性能够保证并发执行的事务仅仅当操作互不交错时才能互相影响。很明显,如果在隔离情况下程序逻辑是符合一致性要求的,那么在给定了隔离性的保证后,在事务并发执行的情况下,一致性仍将得到保证。因此在接下去的内容中我们将不打算专门讲述数据库对事务提供的这一性质。值得注意的是,对数据库应用的一致性测试也是对该应用的隔离性实现情况的很好测试。我们将在10.10节的TCP-A基准测试中看到这一点。

在接下来的章节中,我们将用定义和定理的形式、以非常严格的方法来重新考察事务的各个概念。第2章介绍关系模型的概念时以及第5章学习数据库设计时,我们都已经采用了同样的方法。这样做的原因是,这些概念有一定的难度,而这种严格的方法恰恰是最为明确、

可行的交流思想的教学方法。这样做的另外一个原因是历史原因：自从20世纪50年代提出事务的概念以来，关于事务研究的一些文章都是在严格的计算机科学学报上发表的，尤其是20世纪70年代开始。

人们曾经期望在20世纪50年代开始计划开发的事务系统给他们带来数十亿美元的软硬件销售额（一份与银行以及数百万被无数账本压得喘不过起来的银行雇员的合同）。现在每年工业界花在事务系统上的资金在六十亿美元左右。自然开发这些系统也为国家和公司带来了可观的收益。当计算机公司投资开发这些系统时，他们自然希望程序实现过程用到的一些基本的概念到时不会引起令人尴尬的错误。公司的高层管理人员仔细聆听着系统的设计人员和开发人员的介绍，他们想知道如何才能保证这些投资数百万美元的项目最终能产生一个如其所宣称的那样的工作系统。任何事情都将围绕这一问题展开。我们是否已经了解了所有可能出现的问题（是否仅仅会出现以上所列的三个问题呢）？这些方法能够解决我们知道的所有问题吗？是否有别的问题我们还没有考虑到？我们如何确信这一点？

严格的证明可以用来回答这些问题，至少在这一领域这些问题可以简单地公理化。当然先出现的并非是学报的文章。一组早期的粗糙但实用的原理形成之后，第一代的事务系统就建立起来了。由于这些原理仍处于发展阶段，这样在接下去的一段时间里一批有影响的文章也相继发表，这些文章的作者往往是以前很少在学报上发表文章的编程人员。这些文章构成了事务研究领域的基础，并规范了各种假设和问题，证明了解决方法中应用的原理。这些原理的大多数仍被现在的商业系统所采用。

10.1 事务经历

当允许两个或多个用户对数据库同时执行读写的交错操作时就会出现事务隔离性的要求。在此处，当我们谈到读写操作时，我们是指最原始意义上的操作；而面向集合的SQL语句正是由这些操作组成的。“读”是指访问一个数据项，比如数据库表中的某一行或者数据库的某个索引项，“写”是指对数据库中某个数据项的改变。我们通常称这两个操作为读和写，而不是读和更新，因为“更新”一词在数据库SQL语句中有丰富的隐含意义（通常是指读操作之后，又根据所读的内容进行一次写操作）。

1. 数据库中基本的原子读写操作

我们使用记号 $R_i(A)$ 表示事务对数据项A的一次读操作，其中 T_i 表示该事务， i 是数据库系统给出的标识号。现在我们来考虑拥有两个属性`uniqueid`（该属性唯一标识一个数据项）和`val`（事务中要读写的值）的一张表T1。那么 $R_i(A)$ 可以表示为事务 T_i 执行以下的SQL语句：

```
select val into :pgmval1 from T1 where uniqueid = A;
```

我们还使用记号 $W_i(B)$ 表示事务 T_i 对数据项B的一次写操作；操作 $W_i(B)$ 也可以表示为事务 T_i 执行以下的SQL语句：

```
update T1 set val = :pgmval2 where uniqueid = B;
```

当然这些例子有些简单，上面的Select和Update语句可能包含对多个数据项的操作。在两条SQL语句中都使用了WHERE子句谓词，其作用是根据给出的`uniqueid`的值来确定要操作的T1中的行。我们可以将读取这个谓词信息（无论是否通过索引）看做读取它右边的一个数据项。

在多个事务读写同一个数据项的过程中必须保证一点，那就是一个事务对某个数据项的读操作不能在另一个事务对同一数据项正在进行写操作时进行，因为这时该数据项可能是不一致的（比如一个长的字符串）。同样，两个写操作也不能相互干扰产生不一致。我们称R_i(A)和W_j(B)是原子操作，是指我们可以将它们看做是在任意两个数据库的操作之间瞬间完成的。这和我们在前面提到的事务的原子性是完全不同的概念。

需要注意的是我们在记号R_i(A)和W_j(B)中并没有指出实际值；但是如果需要，我们可以将这两个记号扩展为R_i(A,val1)和W_j(B,val2)。读写的值作为第二个参数（与val1和val2相关的列名不用给出）。通常第二个参数可以定义为常量，比如，R_i(A,50)和W_j(B,80)。

2. 谓词读操作

在前面我们提供的Select和Update语句的例子可能是最简单的。事实上数据库读写操作可以表现得非常复杂。比如，读写操作可以包括一行中的多个列，而不仅仅是只对唯一标识行的单个列的操作（尽管表T1确实只有一个相关的列）。再比如事务T_i可以在一组行上执行相应的SQL语句，而不只是单行：

```
[10.1.1] update tbl set val = 1.15*val
      where uniqueid between :low and :high;
```

在这个例子中这条语句将保证大量不同的读和写操作。事务T_i首先执行谓词读操作——R_i(PREDICATE)，读出那些满足WHERE子句中给出的条件的行，即行的uniqueid的值介于程序变量:low和:high之间的行。当然在检查这一条件时需要采取一些必要的方法，比如通过索引查找。[10.1.1]的Update语句在事务T_i中可以如下表示：

```
Ri(predicate: uniqueid between :low and :high)
```

获得满足条件的行的列表后，[10.1.1]中的Update语句就可以执行一系列的读写操作，先是R_i(uniqueid_k,value_k)操作，然后是W_i(uniqueid_k,1.15*value_k)，其中uniqueid_k介于:low和:high之间。

3. 事务的读写经历

在接下去的内容中，我们假定每个用户进程都在事务中执行原子读写操作。稍后我们也会考虑谓词读问题。作为一个执行这些读写操作的应用，当进程中的前一个事务结束时，数据库系统的事务管理器层（参见图10-1）解释下一个事务第一个读或写操作，并赋以事务号i。现在，我们假定任何事务包含的数据库活动都可以视为一系列的读操作（R_i(A)）写操作（W_j(B)）以及提交语句发出的提交操作（C_i）和回滚语句或由于死锁（我们将在10.4节阐述）发出的回滚操作（A_j）。此处我们忽略整行的插入和删除。在我们的模型中，读操作代表事务所有的信息获取操作；而写操作代表事务所有的修改行为。

两个事务T₁和T₂执行的交错读写操作可能如下所示：

```
[10.1.2] ... R2(A) W2(A) R1(A) R1(B) R2(B) W2(B) C1 C2 ...
```

类似以上的一个操作序列称为事务经历，或者称之为一个调度。在经历[10.1.2]中，我们可以看到事务T₂先对A执行读操作，然后T₂将一个新的值写回到A中。随后T₁对取A的新值，最后C₁和C₂分别代表事务T₁和T₂的提交操作。这样一个经历是应用程序级别上同时执行的两个事务所发出的调用的结果（参见图10-1），最终转变为数据库调度器层次上看到的如[10.1.2]的形式。

为了给后面几节将要讲到的内容做一些铺垫，我们将向你展示数据库调度器（参见图10-

1) 是如何处理像我们在经历[10.1.2]中所看到的交错操作序列的，从而使得调度器产生的操作序列在执行效果上等价于一个串行调度。在这种方式下，我们能保证每个事务与其他事务都是隔离的，也就是说我们提供了隔离性保证。当调度器发现某些操作在当前顺序中会破坏隔离性时就会延迟该操作的执行，以此来保证隔离性；在某些特殊情况下，调度器发现死锁（我们已经在第5章介绍过），这时需通过将某个事务强行异常中止来解除死锁。事实上，从可串行化的角度来看经历[10.1.2]表示一个不合法的操作序列，数据库调度器是不允许出现这种形式的。

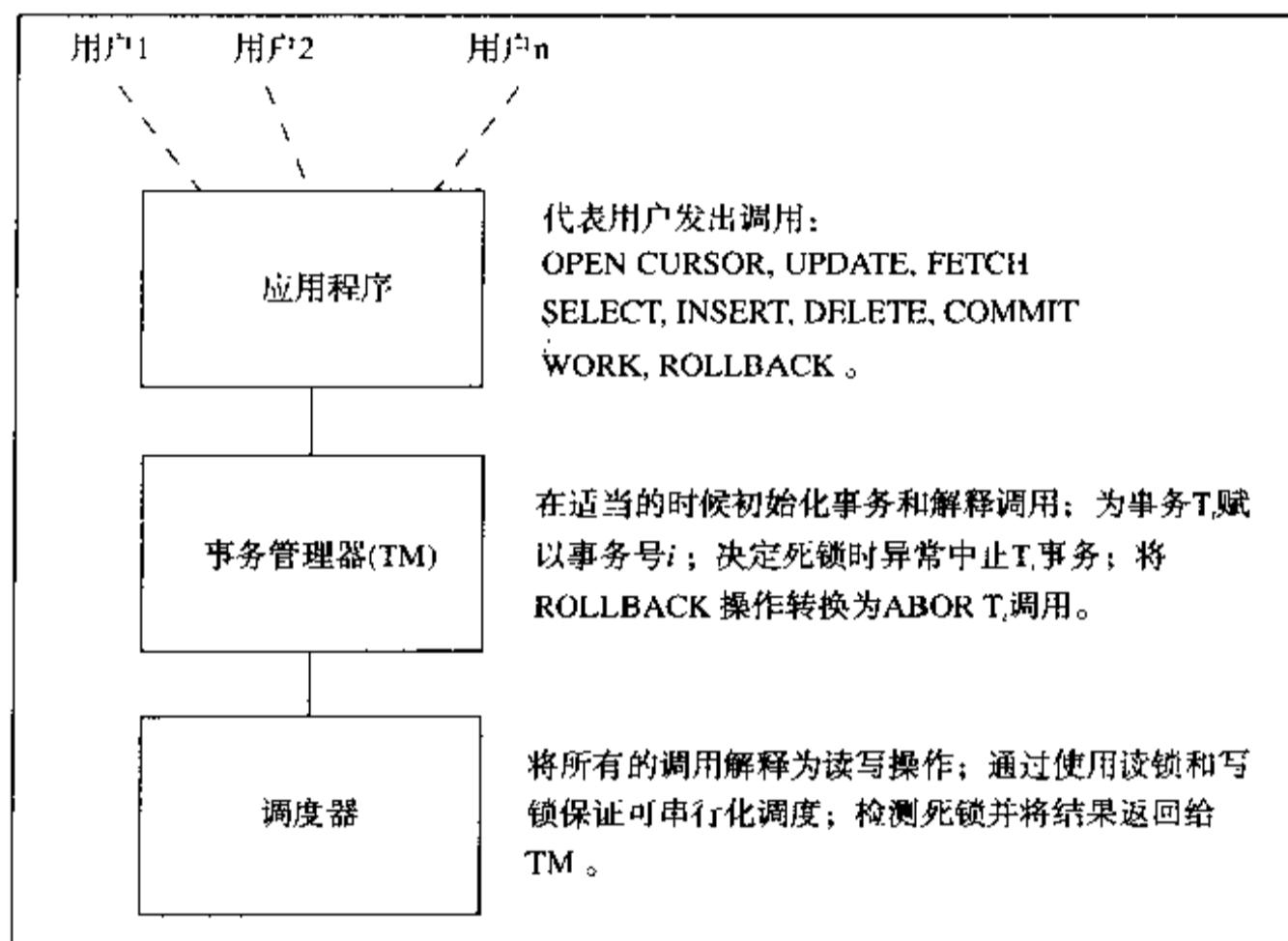


图10-1 事务系统的层次结构图

之所以说这个经历是不合法的是因为这一操作序列所给出的操作顺序在某些情形下可能导致不一致的结果。我们称之为该经历的一个解释。

例10.1.1 我们将给出经历[10.1.2]的一个解释，可以看到如何导致不一致的结果。为达到这一目的，对将要读取的数据项我们给以特定的值，并假定事务对该数据项最后写入的值和从该数据项读到的值相关，然后定义一个一致性规则，我们可以看到该规则将被这些操作破坏。假设数据项A和B是某个银行客户所拥有的两个账户，它们的初始值为A=50和B=50。类似例子5.4.1和5.4.2中为了引出事务隔离性的要求所给出的不一致分析，事务T₁只是简单地对这两个账户的余额相加计算该客户的净资产，从而对该客户进行信用评测。事务T₂执行从一个账户到另一个账户的转账操作，将数额为30的资金从账户A转入账户B。两个事务都必须满足这样的一致性要求，也就是两个账户的资金总额既不能多也不能少。然而，由于这两个事务的操作以调度器所不允许的顺序交错执行，不一致的情况就有可能发生。以下是给出了具体值之后的经历[10.1.2]：

[10.1.3] ... R₂(A, 50) W₂(A, 20) R₁(A, 20) R₁(B, 50) R₂(B, 50) W₂(B, 80) C₁ C₂ ...

我们可以看到，在事务T₂对A执行了写操作之后（这时A的值变为20），事务T₁读取了A的值，而且在事务T₂对B的值进行改变之前T₁读取了B的值（这时读到的是50）。这样事务T₁计算

得到的值是70，这个值无论是在 T_2 执行前或者在执行后计算都是一个不正确的值。这样的不一致视图当然会导致错误的信用评测结果。导致不一致的原因是事务 T_1 使用了事务 T_2 的部分结果。这两个事务没有很好的相互隔离。

这种部分结果的视图在事务的操作不相互交错时是不会发生的。考虑以下两个串行经历，在一个事务的所有操作完成之前，不执行其他事务的任何操作，如图10-2所示。

$\dots R_1(A, 50) R_1(B, 50) C_1 R_2(B, 50) W_2(B, 20) R_2(A, 50) W_2(A, 80) C_2 \dots$
$\dots R_2(B, 50) W_2(B, 20) R_2(A, 50) W_2(A, 80) C_2 R_1(A, 80) R_1(B, 20) C_1 \dots$

图10-2 [10.1.2]的两个串行经历

可以看到这两个经历都能得到数据A和B的一致视图。在第一种情形下， T_1 计算50和50的和为100，在第二种情形下计算80和20的和为100，符合一致性要求。很明显，如果所有事务各自都满足该一致性规则（资金总额平衡），则这些事务的任何串行调度也是满足一致性要求的：系统的资金总额平衡。然而，如果执行的是转账操作，那么在事务操作序列的某些中间点资金总额是不守恒的，因为我们不可能将从一行中扣钱和往另一行上加钱在一个写操作内完成。修改事务的部分结果视图问题在于数据的不一致状态可能被其他事务所看到。但是如果事务只是以串行的方式执行，则不会出现这些问题，因为在一个事务的所有操作结束之前，其他事务的操作不能开始，自然也不会访问到数据的不一致状态，因此例[10.1.3]中的不一致问题就不会发生了。

接下来的内容中，我们使用串行执行的效果作为事务调度正确性的标准。我们说一个多个事务的操作相互交错的经历是可串行化的，如果该经历的执行效果与这些事务的某个串行经历的执行效果相同。我们称这两个经历是等价的。但我们如何来判断一个未被解释的经历其执行效果等价于一个串行经历呢？我们再回忆一下例10.1.1，当我们定义了一个一致性规则（资金总额必须平衡）并假定了一个事务的目标，从而导致经历H中一系列对具体值的读写操作时（比如[10.1.3]），我们说我们创建了经历的一个解释H（比如例子[10.1.2]）。如果可以证明经历的一个解释H违反了一致性规则（比如 T_1 看到了一个不可能存在的总余额），那么这个经历就不可能和任何一个串行经历等价。接下来我们将看到许多这样的例子。

现在我们可以来回顾一下在本章开始时向大家介绍的事务的四个性质，即ACID性质。我们说事务具有以下性质：

原子性 事务的更新操作必须作为一个整体，要么全部成功完成，要么全部失败。

一致性 事务的成功完成将数据库从一个一致状态转变到另一个一致状态。比如，如果一致性要求资金总额既不能凭空多出来，也不能无端地消失，那么成功执行的前后状态必须具有相同的总余额。

隔离性 即使多个事务并发执行，看上去要像每个成功事务按串行调度执行一样。（某些事务必须强行异常中止以保证隔离性，比如在死锁情况下，这些异常中止的事务可以在稍后重新执行。就像例子5.4.4向我们演示的那样）。

持久性 一旦事务提交，那么它对数据所作的修改将是持久的，无论发生何种机器和系统故障。

事务的原子性、隔离性、持久性由数据库系统加以保证，从而使得编程人员不必关心这些问题。一致性是事务的一个逻辑属性，一般要求程序员在编写程序中予以保证，即在隔离

环境下安排合理的逻辑。通过保证事务的隔离性，系统同样也保证了程序中定义的一致性，即使在交错操作和系统崩溃的情况下。

10.2 交错的读写操作

我们已经知道对于一个串行经历而言，不会产生由于不同事务的读写操作交错执行而导致的不一致的问题，因为在串行经历中在一个事务的所有操作完成之前，其他任何事务的操作都不能执行。那么为什么我们要交错执行来自不同事务的读写操作呢？为什么不将所有的事务都按严格地串行执行呢？下面这个方法可以作为调度器用在操作上实现事务的串行执行的规则：一旦事务 T_i 执行了一个数据访问操作（ R_i 或 W_i ）开始该事务，我们说该事务 T_i 是活动的，或者说在执行中。此时如果代表另外一个用户的事务 T_j 向调度器发出了一个初始化该事务的操作（ R_j 或 W_j ），调度器将使事务 T_j 处于等待状态直到 T_i 完成——也就是直到 T_i 异常中止（ A_i ）或提交（ C_i ）。如果有多个事务被强制等待，那么当事务调度器发现某个事务结束时，将使用先来先服务的策略启动下一个事务的执行（执行该事务的第一个操作）。

大多数据库系统不强制事务严格串行执行的原因很简单：不同事务的操作的交错执行大大地提高了系统的性能。在处理过程中允许多个事务并发执行意味着当一个事务进行I/O操作时，另外一个事务可以使用CPU，这样就提高了整个系统的吞吐率，吞吐率以给定时间内完成的事务数量来衡量。因为I/O操作较费时间，因此当有多个磁盘并允许同时有多个活动事务时，将大大提高系统的吞吐率。

例10.2.1 假定有大量用户在等待事务型应用的服务，每个事务按照以下的顺序使用CPU和I/O资源：（使用CPU） $R_i(*)$ （使用CPU） $W_i(*)C_i$ ， $R_i(*)$ 和 $W_i(*)$ 代表I/O操作。我们假设系统只有一个CPU，提交操作不使用任何资源，CPU使用的时间片为5ms（0.005s），每个I/O操作（无论是 $R_i(*)$ 还是 $W_i(*)$ ）需要等待50ms。（我们可以看到，一个I/O操作立即得到服务时只需要12.5ms，然而等待队列使得平均服务时间变慢。在下面的讨论中我们以单个事务的线程作为开始，这是讨论多个用户并发情况的基础，在多用户并发时等待就是经常的事了。）我们可以使用图10-3所示的调度图来描述一组严格串行调度的事务所产生的操作的事件序列。

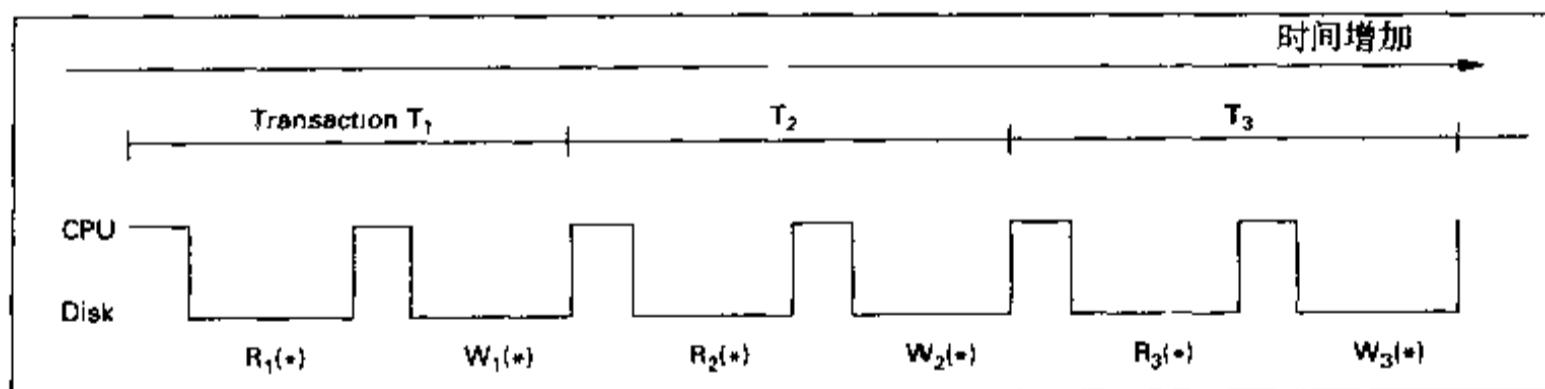


图10-3 串行事务调度中的事件序列

在图10-3中资源使用情况用阶跃功能图来表示，以一定的间隔来表示使用CPU和磁盘I/O所用的时间。每个磁盘操作以将要执行的事务的存取操作作为标记，比如 $R_i(*)$ 和 $W_i(*)$ 。我们可以看到每个事务开始先占用5ms的CPU，然后是 $R_i(*)$ 操作占用50ms的I/O时间，接下去又是占用5ms的CPU时间，最后的 $W_i(*)$ 操作占用50ms的I/O时间。提交操作 C_i 不占用任何资源，因此时间为0，于是下一个事务可以立即开始。因此每个成功完成的事务总共需要占用110ms的

时间，也就是说每隔110ms完成一个事务，所以吞吐率为每秒9.09个事务（9.09TPS）。在这种情况下，CPU未被充分使用，因为在110ms的时间中只有10ms在使用CPU，也就是说使用率只有9.09%，但是磁盘却得到了很好的使用率，110ms中有100ms在使用磁盘，如果有其他进程在使用磁盘这一比例甚至会更高。

图10-4显示了两个事务的操作交错执行的情况，但系统仍然只有一个磁盘。为了能够描绘出两个并发事务进程（或者事务线程）的资源使用情况，我们在图的左边对CPU和磁盘分别使用了两个标记，尽管如此两个标记表示的是同一个CPU和磁盘。

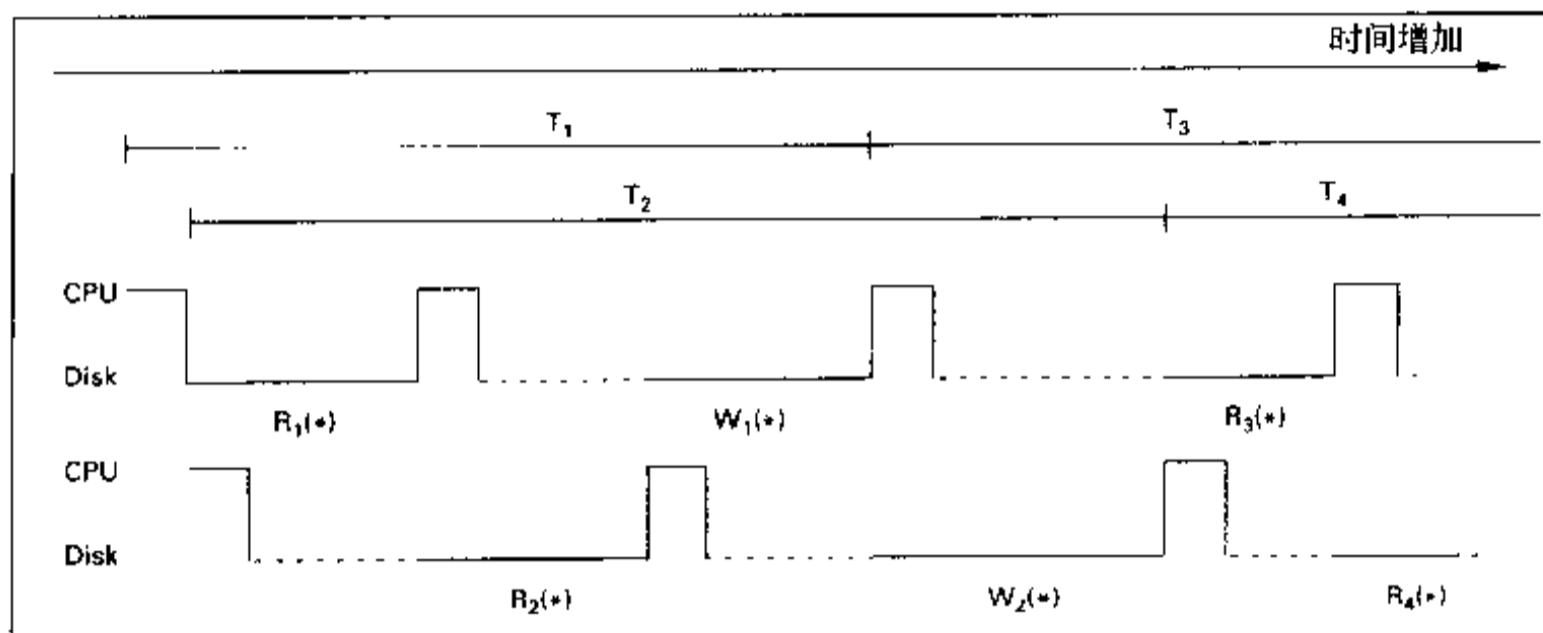


图10-4 一个磁盘上两个交错事务的事件序列

在图10-4中我们可以看到两个事务线程大部分时间在等待对方完成I/O操作，以便进行自身的I/O操作。然而我们可以看到这种调度成功地将CPU使用时间和磁盘I/O操作重叠起来。每个事务要求100ms的磁盘访问，这样磁盘被充分地利用起来，而没有像以前那样每110ms中有10ms的空闲时间。从长远来看，因为I/O操作花的时间是关键因素，因此在这种调度方式下我们得到的吞吐率是每100ms完成一个事务，也就是10TPS。一个具体的调度用图来表示也许不是很清楚，因此我们在下面详细叙述。在例子开始时，我们看到事务T₁占用了5ms的CPU，然后是R₁(*)操作占用50ms的I/O时间，接下去又是占用5ms的CPU时间，接着是花费45ms等待R₂(*)完成I/O操作，最后的W₁(*)操作占用50ms的I/O时间，整个事务完成。总共花的时间是：5+50+5+45+50=155ms。这个时间通常是比较快的，因为T₁是最先使用资源的。然而，其他的后继事务T₂花费5ms使用CPU，花费45ms等待磁盘，花费50ms进行R₂(*)操作，接着又是花费5ms使用CPU，45ms等待磁盘，50ms进行W₂(*)操作；因此总共花的时间是：5+45+50+5+45+50=200ms。因此总地来看，每个线程运行一个事务需要200ms时间，两个事务一起执行时每个是事务占100ms，即10TPS。

在图10-5中我们可以看到两个事务线程交错操作，奇数号的线程在磁盘1上进行I/O操作，而偶数号的线程在磁盘2上进行I/O操作（当然在现实的系统中没有这么理想，这一点我们将在稍后讨论）。我们可以看到两个线程执行过程中没有任何等待（除了最初的5ms等待，当时T₁正在使用CPU，因此T₂必须等待）。在后面的CPU使用中，最初的等待产生的时间偏移使得奇数号的线程和偶数号的线程的不会重叠（这对一个实际系统而言是完全不现实的）。就像在图10-3中看到的，每个事务线程每110ms执行一个事务，即9.09TPS，当两个事务线程同时执行时将得到两倍的吞吐率，即18.18TPS。在这种情形下，每55ms中CPU使用10ms，利用率为18.18%。

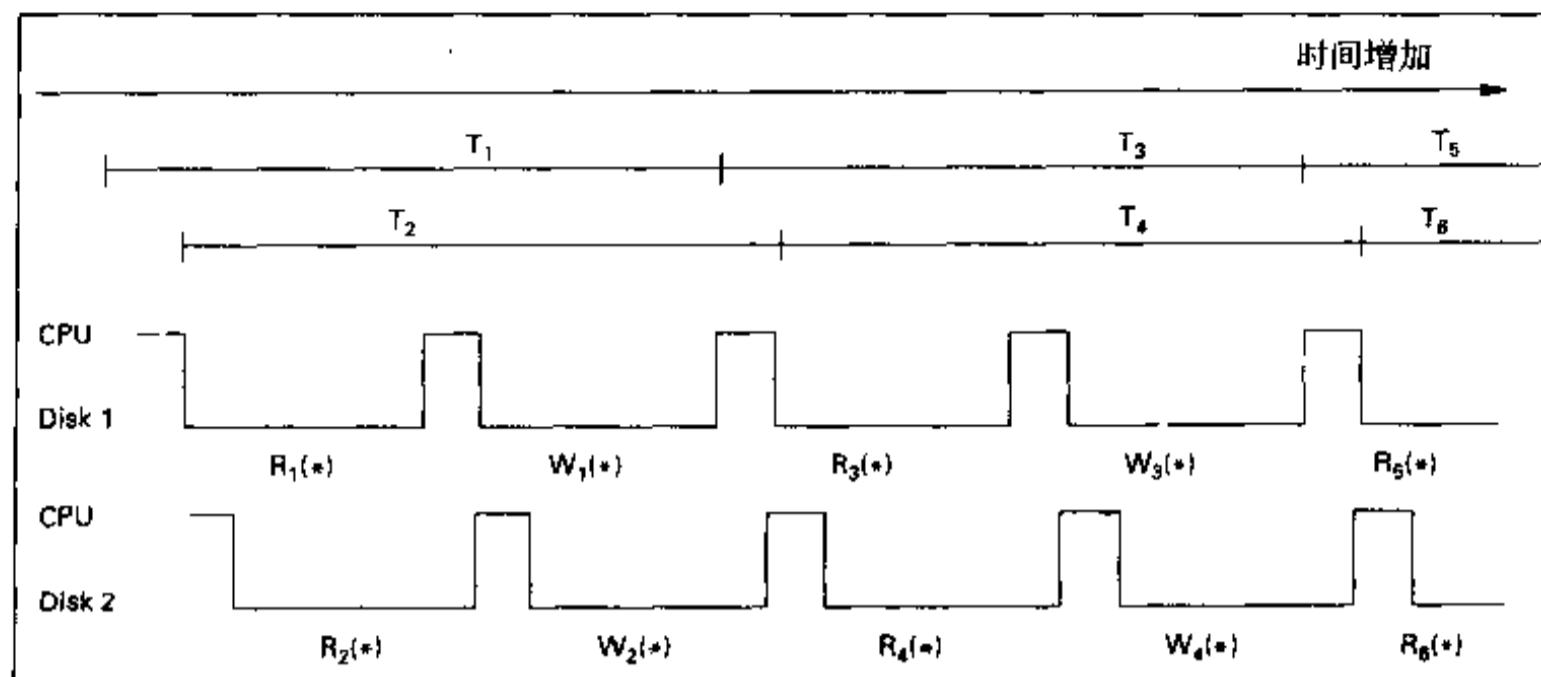


图10-5 两个磁盘上两个交错事务的事件序列

为了进一步提高CPU的利用率，我们可以增加更多的磁盘、运行更多的事务线程，从而获得更大的操作重叠。图10-6给出了具有11个磁盘和11个事务线程情况下的示意图，我们假定资源重叠使用情况是理想的。在这种情况下所有11个事务线程都是全速运行的。每个事务仍然是花110ms时间，但是11个事务线程是同时运行的，因此在110ms中我们可以完成11个事务，即100TPS。由于假定资源使用具有非常好的重叠性，CPU的使用率为100%。 ■

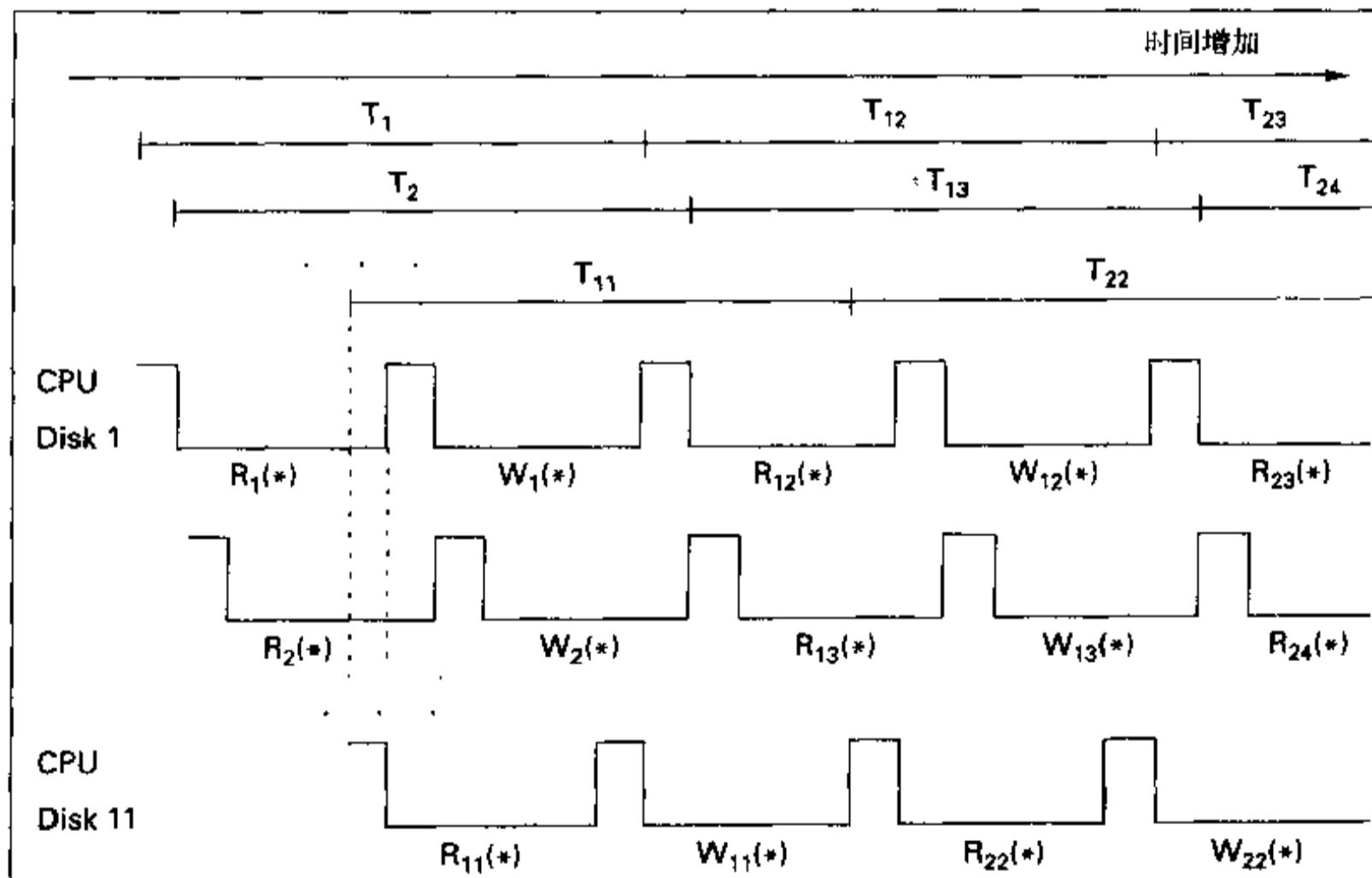


图10-6 11个磁盘上11个交错事务的事件序列

对于一个事务系统而言，5ms的CPU时间与50ms的I/O操作时间之比是很正常的。由于计算机的速度变得越来越快，这一比例会变得越来越大，因为CPU速度的提高要比磁盘I/O操作的速度提高要快得多。由于这个原因许多事务型系统往往配置了大量的磁盘来缓解两者的差异。

当然对于并发执行的事务，并不能保证它们像图10-6所描述的那样，能够访问不同的磁盘从而很理想地将CPU时间和I/O时间重叠。对于例10.2.1中的工作（10ms的CPU时间和50ms的I/O时间），让我们看一下DBA通常采取的配置方法。（在下面的分析中我们将考虑用实际移动磁盘臂过程中占用的时间25ms代替原来笼统的50ms的I/O时间，因为这一决定会影响实际的磁盘等待时间）。如果我们很好地表达这一问题，它实际上是数学中的排队理论。遗憾的是，如果没有一定的数学背景，很难将这个原理讲清楚。但我们可以试着使你有一些基本的了解。对于I/O与CPU时间5比1这样一个比例比较明智的配置是：磁盘数要大大超过5个，比如10个磁盘；然后同时运行10个以上的事务线程，比如20个。这些事务要读写的数据应该均匀地分布在这些磁盘上，这样任意一个事务线程的任意一个R_i(*)或W_i(*)操作都有可能访问这些磁盘中的任何一个。很明显随着我们增加磁盘的个数，对于任何一个事务线程，当它完成CPU使用而要到磁盘上进行数据存取时，它会发现数据所在磁盘往往是可用的，并没有被其他事务线程所占据。因此不会产生多个事务在队列中等待的现象。在此处我们所做的只是通过增加磁盘个数来缩短平均磁盘访问时间和CPU使用时间的比例。尽管在以上讨论的例子中我们指定磁盘访问时间为50ms，但对于给定的随机磁盘访问速度，实际的磁盘访问时间依赖于可用的磁盘数，后者可以减小等待队列的长度。在下面的讨论中我们假定一个更低的限制为25ms。

现在DBA同样需要降低购买硬件的费用，这就要求购买的磁盘个数不能超过实际充分使用CPU所必需的磁盘个数。这意味着仍然可能存在大量的磁盘访问冲突，即在相近的时间不同的事务线程向同一个目的磁盘提出了I/O操作请求，这样我们肯定不可能获得像图10-6那样的完美的重叠执行。如果我们只是运行11个事务线程，而平均的I/O访问时间为50ms，这样就会产生非常严重的问题，因为在这种情形下只有当这些I/O请求能很好地交错时才能保证CPU被充分的利用。但是如果我们将20个并发的线程，那么就可以有令人信服的同样多的20个并行运行事务。任何一个时刻，任何一个事务必定处于以下几种情况之一：一些事务正在等待CPU变得可用或者正在使用CPU（我们称这些事务处在CPU队列中），一些事务正在等待某个磁盘变得可用或者正在使用某个磁盘（我们称这些事务处在某个磁盘的队列中）。为了获得较高的CPU利用率，我们只需要保证CPU队列几乎不为空就行了（如果是空的，CPU就没有事务可执行），因此如果我们能够保证一个平均较长的CPU队列，就能达到目标。很清楚，如果我们能够保证在任何时刻10个磁盘中有大约5个磁盘在被使用，那样我们就达到目标了。根据排队理论的一个基本原理，如果一个请求者（某个事务）释放某个系统资源（比如磁盘）到做出另外一个资源请求（比如CPU）要快于该请求者获得服务的时间，那么超负荷的资源提供者的队列将无限制加长。（对于单个CPU而言，由于磁盘对CPU的服务时间比是25ms比5ms，因此当我们平均有超过5个磁盘在同时提供服务时，CPU就能够满负荷运行。）

通过允许大量的线程同时运行（即使在某些磁盘上发生I/O请求冲突时）我们为系统提供更多要做的事情。尽管一些线程处于等待I/O服务的状态，CPU可以运行其他的事务线程从而产生更多的I/O请求。有了足够的I/O请求之后，这些请求肯定会分布到不仅仅是10个磁盘中的5个磁盘，这样就能保证CPU被高效地使用。应该注意的是我们不需要仅仅因为我们可以同时运行20个线程而去并发执行20个事务，事实上到目前为止我们所解释的每一件事都是一个反馈过程的一部分。当一个新的事务进入系统时，系统将其分配给一个单独的线程，以便20个事务能够并发地执行。然而，对于仅有20个并发事务而言，事务的完成速度可能非常快——更大的并发事务数目意味着更高的吞吐率，通常我们不需要最大的并发线程数目。同样

我们也不需要100%的CPU利用率，也许90%或者95%就差不多了。另一方面，如果CPU利用率太低，这样处理工作负载的吞吐率就达不到要求。这时更多的事务会进入系统并发执行，直到达到最大数20，于是磁盘和CPU又被充分利用起来。（在所有20个线程都被占用以后，其他事务就必须在另一个队列中等待，直到分配到线程。）

因此当有更多的磁盘和线程加入到系统中时，我们就必须依赖统计手段来提高CPU的使用率了。对这一方法的具体实现有兴趣的读者可以去阅读一篇关于排队理论标准的介绍性读物。建议读者参考本章后面的“推荐读物”。

10.3 可串行化和前趋图

在本节中我们将推导出一个识别给定的经历（比如[10.1.2]）是否可串行化的标准。这意味着一个经历等价于（等价的定义将在下面的内容中给出）某个串行经历（在一个经历的所有操作完成之前其他经历的操作不能执行）。一个可串行化的经历能够被一个支持事务隔离性的系统调度器所接受。在定义事务的等价之前我们先来讨论一下什么时候一个串行经历中两个事务的交错操作是冲突操作。如果两个不同事务发出的读写操作访问的是同一个数据对象，并且其中至少有一个是写操作（另外一个可以是写也可以是读），我们说这两个操作是冲突的。这个概念在定义10.3.1中会有更准确地表述。大家将看到经历中不同事务的两个冲突操作在经历中出现的顺序将是很重要的一个因素。对于两个包含相同事务性操作的经历，如果所有的冲突操作对在两个经历中具有相同的次序，即使其他一些操作在次序上可能有些差异，这两个经历仍然可以说是等价的。有了这样的背景，经历的可串行化就可以定义为：如果一个经历等价于一个串行经历，这个经历就是可串行化的。在定理10.3.4中我们给出了一个确定什么情况下一个经历是可串行化的标准。

为了更好地了解一些概念，假定我们有一个类似[10.1.2]中的事务性操作经历H。假定在经历H中，一个事务读取了数据项A的值，稍候另一个事务对A进行了更新（写）操作。因此如[10.3.1]所示的这两个操作出现在经历H中，其中它们之间的省略号(…表示经历中在这两个操作之间的任何操作。

[10.3.1] ... R₁(A) ... W₂(A) ...

当我们说经历中执行了更新操作W₂(A)时，意味着一个新值已经覆盖了数据项A中原来的值。我们没必要认为该更新操作隐含的是由SQL中的Update语句完成的，因为此处我们并没有指出这个写入的新值和数据项A原来的值有什么必然的联系。（当然我们也没有否定可能会先读A原来的值。）如果该操作隐含着先要进行一次读操作，我们可以用…R₂(A)…W₂(A)…来代替…W₂(A)…。

现在我们要尽力提出一个和经历H等价的串行经历，H中包括了[10.3.1]中的两个操作并具有相同的次序。与之等价的串行经历可能存在多个，但是我们不妨用S(H)来代表它们中的任何一个。对于任何一个等价的串行经历，我们可以肯定事务T₂的任何操作都在事务T₁之后。这是因为T₁在经历H中对A的读操作（假定为50）在事务T₂对A的写操作之前；这一写操作有可能改变A的值（假定为20）。我们可以构建这个经历的一个解释。因为在所有的等价串行经历中，所有的事务和经历H中一样读取的是相同的数据项，这是毫无疑问的，因此很明显在两种情形下事务T₂都必须在事务T₁之后。我们使用如下的标记：

R₁(A) <<_H W₂(A)

表示在经历H中 $R_i(A)$ 在 $W_j(A)$ 之前，而且在等价经历 $S(H)$ （也许不只这两个操作）中这两个操作也必须保证同样的先后顺序。我们可以如下表示：

$$R_i(A) \ll_{S(H)} W_j(A)$$

更一般的，由于在串行经历中事务 T_i 的所有操作都是在一起执行的，因此我们可以用事务 T_i 本身来表示该事务所有的操作，所以可以表示成如下的形式：

$$T_1 \ll_{S(H)} T_2$$

现在我们需要讨论的是如果事务 T_1 在事务 T_2 对某个数据项进行写操作之前读了该数据项的值。那么在任何一个等价的串行经历中这两个操作的先后顺序不能颠倒。反之也同样成立，即如果经历H中事务 T_2 在事务 T_1 对某个数据项B进行读操作之前修改了该数据项的值。在等价串行经历中两者先后顺序也不能倒过来。

$$W_2(B) \ll_H R_1(B)$$

由此我们可以断言在任何串行经历 $S(H)$ 中事务 T_2 都必须在事务 T_1 之前，即：

$$T_2 \ll_{S(H)} T_1$$

到目前为止我们可以确定的东西可以表述如下：如果经历H中分别属于两个事务 T_i 和 T_j 的两个操作 $X_i(A)$ 和 $Y_j(A)$ 是冲突操作，当其中一个操作在另一个操作之前执行时，那么在等价的串行经历中相应的事务也应该保持同样的先后顺序。无论 $R_i(A)$ 和 $W_j(A)$ 在经历中的先后顺序如何，它们都是冲突操作。

为了放宽条件，我们已经确定对同一个数据的读操作和写操作是冲突的。在只有两种操作类型的情况下只可能有四种组合类型，我们现在需要确定的是读操作和读操作之间是否可能冲突，即 $R_i(A)$ 和 $R_j(A)$ 冲突吗？我们认为它们是不冲突的。如果在经历中我们有如下的操作序列： $\cdots R_i(A) \cdots W_k(A) \cdots R_j(A) \cdots$ ，那么由于两对读写冲突操作我们可以获得以下结论： $T_i \ll_{S(H)} T_k$ 且 $T_k \ll_{S(H)} T_j$ ，因此由传递性我们可以进一步的到 $T_i \ll_{S(H)} T_j$ 。然而我们应该注意到的是，这一结果并不是由于分别属于事务 T_i 和 T_j 的两个读操作的缘故。经历H有操作 $R_i(A)R_j(A)$ ，经历H'中有操作 $R_j(A)R_i(A)$ 可以产生同样的结果，只要这两个操作之间没有任何其他操作介入。

另一方面，由 $W_i(A)$ 和 $W_j(A)$ 构成的操作对是冲突的。我们应该意识到这两个操作之间的前后顺序是关键，因为由于两个操作对数据对象A写入的值不一样，因此数据对象A的值将由最后的写操作决定。显然最后给出的结果不一样的两个经历肯定是不会等价的。图10-7给出了我们已经发现的三对冲突操作。

- | |
|---|
| (1) $R_i(A) \rightarrow W_j(A)$ (意味着，在一个经历中，操作 $W_j(A)$ 在操作 $R_i(A)$ 之后。) |
| (2) $W_i(A) \rightarrow R_j(A)$ |
| (3) $W_i(A) \rightarrow W_j(A)$ |

图10-7 冲突操作的三种类型

图10-7所示的这三对操作和我们已经得出结论不是冲突操作的操作对 $R_i(A)$ 和 $R_j(A)$ 一起，构成了两个事务 T_i 和 T_j 对同一个数据对象的所有可能的操作组合。相比之下，在不同数据对象上的任何操作都是不冲突的：比如 $R_i(A)$ 和 $W_j(B)$ 是不会产生冲突的，因为这两个操作的先后顺序根本无关紧要。如果两个不同的事务对不同的数据对象进行操作，那么这两个事务执行的先后顺序是完全不相关的。让我们再次回到 $R_i(A)$ 和 $R_j(A)$ 的例子，我们可以想像一个事务 T_k ，该事务的操作和 T_i 的 $R_i(A)$ 以及 T_j 的 $W_j(B)$ 相冲突，因此由于传递性导致事务 T_i 和 T_j 必须保持某

种前后顺序；但是我们必须申明的是操作 $R_i(A)$ 和 $W_j(B)$ 本身并不是冲突的。作为对前面讨论内容的总结，我们给出下面的定义。

定义10.3.1 我们说经历H中两个操作 $X_i(A)$ 和 $Y_j(B)$ 是冲突的当且仅当以下三个条件成立：
(1) $A=B$ 。不同数据对象上的操作是不会冲突的。(2) $i \neq j$ 。不同事务发出的两个操作才可能冲突。(3) 两个操作X和Y中有一个是写操作W。另一个操作可以使读操作也可以是写操作。■

关于经历中冲突操作的定义看起来像是数学上的定义：我们想知道它可以有哪些应用。为了看如何来处理这个概念，我们需要使用以下一些例子。让我们回忆一下在[10.1.2]中给出的经历，我们不妨把它记为经历H1。

$$H1 = R_2(A) W_2(A) R_1(A) R_1(B) R_2(B) W_2(B) C_1 C_2$$

我们应该仍然记得在例10.1.1中我们曾经给出了这个经历的一个解释，我们已证明这个经历是不可串行化的。

定义10.3.2 经历H的任何一个解释有以下三个部分组成：(1) 经历中关于事务的逻辑目的的描述，应该足以证明事务对数据写入的值和事务对同一数据对象读取的值有关系。(2) 经历中读写操作的值要给出具体的说明。(3) 一致性规则。具有隔离性的事务执行过程中必须保持的某种逻辑属性，这些事务具有第1点中所陈述的逻辑属性。在具有交错存取操作的不可串行化的经历中，我们向大家指出经历中的某些事务违反了一致性规则，而这在任何串行执行中是显然不可能发生的。■

例10.3.1 重新考察例10.1.1，由下面给出的经历H1的一个解释可得出的结论是：经历H1是不可串行化的。我们首先来考察定义10.3.2中的条件(2)，即读写操作中包含的具体的数据是：

$$H1 = R_2(A, 50) W_2(A, 20) R_1(A, 20) R_1(B, 50) R_2(B, 50) W_2(B, 80) C_1 C_2$$

再来考察定义中的条件(1)，各事务的逻辑目的是： T_1 做的是客户的信用检查，即对可用的两个账户A和B的余额进行加操作。 T_2 做的是从账户A到账户B的转账操作（我们可以看到要转的这笔钱是30）。最后我们来考察定义中的条件(3)，即两个事务都没有无缘无故地增加钱的数目也没有毫无根据地减少钱的数目。但是这个调度是不可串行化的，因为事务 T_1 看到了不一致的结果，得到的两个账户A和B的余额和是70，而不是我们在串行经历中可以看到的正确值100。

冲突操作概念的引入可以使我们用更直接的方法来检测经历的可串行性。可以注意到在经历H1中的第二个和第三个操作是一对冲突操作： $W_2(A) \ll_{H1} R_1(A)$ 。根据我们关于冲突操作的讨论，我们知道任何与经历H1等价的串行经历 $S(H1)$ 中应该有 $T_2 \ll_{S(H1)} T_1$ 。同时，经历H1中的第四和第六个操作也是冲突操作： $R_1(B) \ll_{H1} W_2(B)$ ，我们可以得到： $T_1 \ll_{S(H1)} T_2$ 。很明显我们得到的两个结论是矛盾的，因为我们不可能创建同时满足 $T_2 \ll_{S(H1)} T_1$ 和 $T_1 \ll_{S(H1)} T_2$ 的一个串行经历。由此我们可以得出结论：经历H1不存在等价的串行经历。■

如果考虑导出这一结论的两个事实，则会看到例10.3.1恰恰为我们前面的说法提供了足够的论据，即任意一个经历中冲突操作的先后顺序在该经历的等价串行经历 $S(H)$ 中必须保持。产生不一致的原因在于：(1) T_1 在 T_2 对A进行了写操作之后对A进行读操作。(2) T_1 在 T_2 对B进行写操作之前对B进行读操作，也就是说两对冲突操作具有相反的方向。这才是真正的原因，

即为什么经历H1中给出了确切的值之后，事务T1计算出了一个任何该逻辑的串行执行中都不可能计算出来的总额值：在任何正确的等价串行经历中，其中一个冲突操作对的次序必须倒过来。

例10.3.2 考虑下面的经历H2：

$$H2 = R_1(A) \ R_2(A) \ W_1(A) \ W_2(A) \ C_1 \ C_2$$

这是事务不一致问题中一个典型的丢失更新问题，有时候也称之为“脏”写（dirty write）。其中每个事务都是先读某个数据项的值，然后再向这个数据项中写入新的值，却没有意识到另外一个事务也在对同一个数据项进行同样的操作。就像前面给出的经历的解释一样，假定A是一个银行账户并拥有初始的余额为100，事务T₁试图向该账户增加40，同时事务T₂试图向该账户增加50。结果将变成：

$$R_1(A, 100) \ R_2(A, 100) \ W_1(A, 140) \ W_2(A, 150) \ C_1 \ C_2$$

A中最后得到的值为150。然而，按照这两个事务的初始意图，在任何调度之后得到的结果应该是190。因此这个调度是不可串行化的。事实上，我们可以看到该经历中的第一个操作R₁(A)和第四个操作W₂(A)是一对冲突操作，给出了R₁(A)<<_{H2}W₂(A)，因此有T₁<<_{S(H2)}T₂。同样，该经历中的第二个操作R₂(A)和第三个操作W₁(A)是一对冲突操作，给出了R₂(A)<<_{H2}W₁(A)，因此有T₂<<_{S(H2)}T₁。于是类似例10.3.1，这个经历没有等价的串行经历存在。这个例子展示了图10-7中类型（1）的两对冲突操作，即R_i(A)<<_HW_j(A)和R_j(A)<<_HW_i(A)，从而如解释中看到的那样使得它们所在的经历变得不可串行化。■

例10.3.3 考虑下面的经历H3：

$$H3 = W_1(A) \ W_2(A) \ W_2(B) \ W_1(B) \ C_1 \ C_2$$

这个经历向我们展示的是第（3）种类型的冲突操作：W-W冲突对。这同样会导致经历变得不可串行化。很明显，因为没有读操作，不涉及第（1）和第（2）两种类型的冲突操作。由于前两个操作给出了W₁(A)<<_{H3}W₂(A)，因此有T₁<<_{S(H3)}T₂。同样，该经历中的第三个操作W₂(B)和第四个操作W₁(B)是一对冲突操作，给出了W₂(B)<<_{H3}W₁(B)，因此有T₂<<_{S(H3)}T₁。因此类似前两个例子，这个经历也不存在等价串行经历。那么我们如何来构造一个解释来说明这一事实呢？假定存在两个账户A和B，初始这两个账户的总的余额是90，在这个经历中我们只处理一种类型的事务，即向这两个账户中加钱，使两个账户的余额总和为100。我们给出的一致性规则是两个账户的余额总和不得超过100。需要注意向这两个账户中加钱的操作是不需要读取账户原来的余额值的——可以简单地向账户A和B进行盲写（不需要读操作就进行写）只要两个账户的总额为100即可。现在我们看下面给出的对经历H3的一个赋值的解释：

$$W_1(A, 50) \ W_2(A, 80) \ W_2(B, 20) \ W_1(B, 50) \ C_1 \ C_2$$

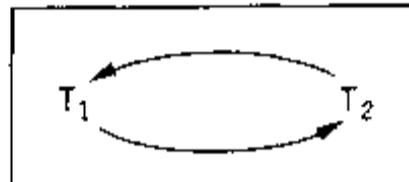
两个事务均违反了一致性规则（每个事务在单独执行时都将使两个账户A和B的总额为100），但是此交错调度产生的最终结果为130，违反了一致性规则。■

前趋图

为了概括我们在以上三个例子中看到的关于可串行化的讨论，某些冲突操作的构造在串行经历中是不可能出现的，为此我们定义了前趋图。前趋图是根据经历H构造的一种结构，它记录了隐含在经历的冲突操作对中的事务之间的先后顺序。

定义10.3.3 前趋图 经历H的前趋图是一个记为PG(H)的有向图。前趋图的顶点代表经历H中已提交的事务，也就是说那些在经历H中存在C_i操作的事务T_i。图中任意一条边T_i→T_j代表在经历中存在一对冲突操作X_i和Y_j也保持同样的先后次序。边T_i→T_j应该理解为在任何等价的串行经历S(H)中T_i在T_j之前执行，即T_i<<_{S(H)}T_j。

经历H中的任何一对冲突操作都对应着前趋图中的一条有向弧。对于例10.3.1、例10.3.2、例10.3.3中的经历对应的前趋图都是一样的，如下所示。



我们可以看到这张图中形成一个环，由于这个原因我们断定该经历没有等价的串行经历存在。任何串行经历都必须将事务T_i或者T_j放在另一个事务的前面，不妨假设T_i<<_{S(H)}T_j，那么根据该前趋图至少有一条返回边，也就是从事务T_j指向T_i。现在由于两个事务中存在冲突操作，因此前趋图PG(H)中存在一条边，这条边的指向就是冲突操作在经历中的先后次序。串行经历对应的PG(H)有一条返回边意味着我们将原来经历H中的一对冲突操作倒过来了，因此串行经历中的先后次序和原来经历H中的先后次序不一样。很明显，如果前趋图PG(H)中存在环，那么对应的经历H不存在等价的串行经历，因为前趋图PG(H)中对于任何串行事务而言至少有一条边是向回指向的。（该结论的证明留为本章后面的习题。）在接下来的内容中，我们将证明如果前趋图PG(H)中不存在环，那么存在一个事务的串行执行等价于该经历H。

定理10.3.4 可串行性定理 一个经历H存在等价的串行经历S(H)当且仅当对应的前趋图PG(H)中不存在环。

证 我们将“仅当”部分的证明留作本章后面的习题。此处我们证明，如果前趋图PG(H)中没有环，那么存在事务的一个串行顺序，使得前趋图PG(H)中不存在一条边对该串行顺序而言是从后面的事务指向前面的事务的。假定经历中总共有m个相关事务，如果有必要重新标记PG(H)中的事务，分别记为T₁, T₂, …, T_m。我们将对这些事务重新排序以找到满足我们要求的串行经历S(H)=T_{i(1)}, T_{i(2)}, …, T_{i(m)}，其中i(1), i(2), …, i(m)是对整数1, 2, …, m的重新排序。我们先假定一个引理（稍后再证明）：任何无环的有向图G中至少存在一个顶点，该顶点不存在入边（箭头指向该顶点）。这意味着前趋图PG(H)至少有一个顶点T_i没有人边。于是我们可以选择这个事务T_i作为串行经历S(H)中的第一个事务T_{i(1)}。这样做隐含着一个重要的限定属性：在PG(H)剩下的顶点中不存在一个其他的事务T_m（将被放置在T_{i(1)}的右面），在PG(H)中有一条指向T_{i(1)}的边。现在我们从PG(H)中删除该顶点T_i以及从T_i出发的所有边，我们将剩下的图不妨称之为PG'(H)，其中角标1表示已经有一个顶点从原来的图中删去了。我们应该注意到PG'(H)也具有无环的特性（因为没有向PG'(H)中加入边，因此自然也不会产生环）。根据我们的引理，在PG'(H)中同样也至少存在一个没有人边的顶点T_{i'}。我们选择该顶点作为串行经历S(H)的第二个事务T_{i(2)}。我们要注意到也许PG(H)中存在T_{i(1)}到T_{i(2)}的边，但是在PG'(H)中T_{i(2)}不存在入边。这意味着在将要确定的串行调度（由PG'(H)进一步给出）中不存在从右向左指向T_{i(2)}的边。依次逐步推导，假定PG'^{r-1}(H)中删除顶点T_{i'}之后得到PG'(H)，如果PG'(H)不空，则继续选取没有人边的顶点T_{i'}，将T_{i'}作为串行经历S(H)的第r+1个事务T_{i(r+1)}。根据我们的构造方

法， $PG(H)$ 中不可能存在从 $S(H)$ 中右边的事务指向 $S(H)$ 中左边的事务的边，即从 $T_{i(m)}$ 指向 $T_{i(n)}$ 且 $m > n$ 。得到这个序列的算法我们称之为拓扑排序，在大量的算法书中都有相关的表述。■

为了完成定理10.3.4的证明，我们必须证明上面提到的引理。

引理10.3.5 在任何有限的无环有向图G中，必然有一个顶点v不存在入边。

证 选择G中选择任意一个顶点 v_i ，要么没有入边，要么存在从顶点 v_i 来的人边。对于 v_1 ，要么没有入边，要么存在顶点 v_1 来的人边。以此类推，该序列要么在某个顶点 v_n 终止，即 v_n 没有入边，或者该序列继续下去。但如果继续这么推导下去，必然会使某些顶点相关，因为图G只有有限个顶点。假定当我们标记顶点 v_i 时，我们第一次发现该顶点和前面某个标记过的顶点 v_i 相同，且 $i < n$ 。但是这样我们就得到一个环： $v_n \rightarrow v_{n-1} \rightarrow v_{n-2} \rightarrow \dots \rightarrow v_{i+1} \rightarrow v_i = v_n$ 。我们已经假定图G是没有环的，因此产生矛盾，所以是不可能的。这意味着我们定义的序列推导不可能无限制地继续下去，必然存在某个顶点 v_m ，该顶点没有入边，这就是我们所要证明的。■

10.4 用来保证可串行性的锁机制

考察图10-8，这是几乎和图10-1一样的图，我们来做一番讨论。

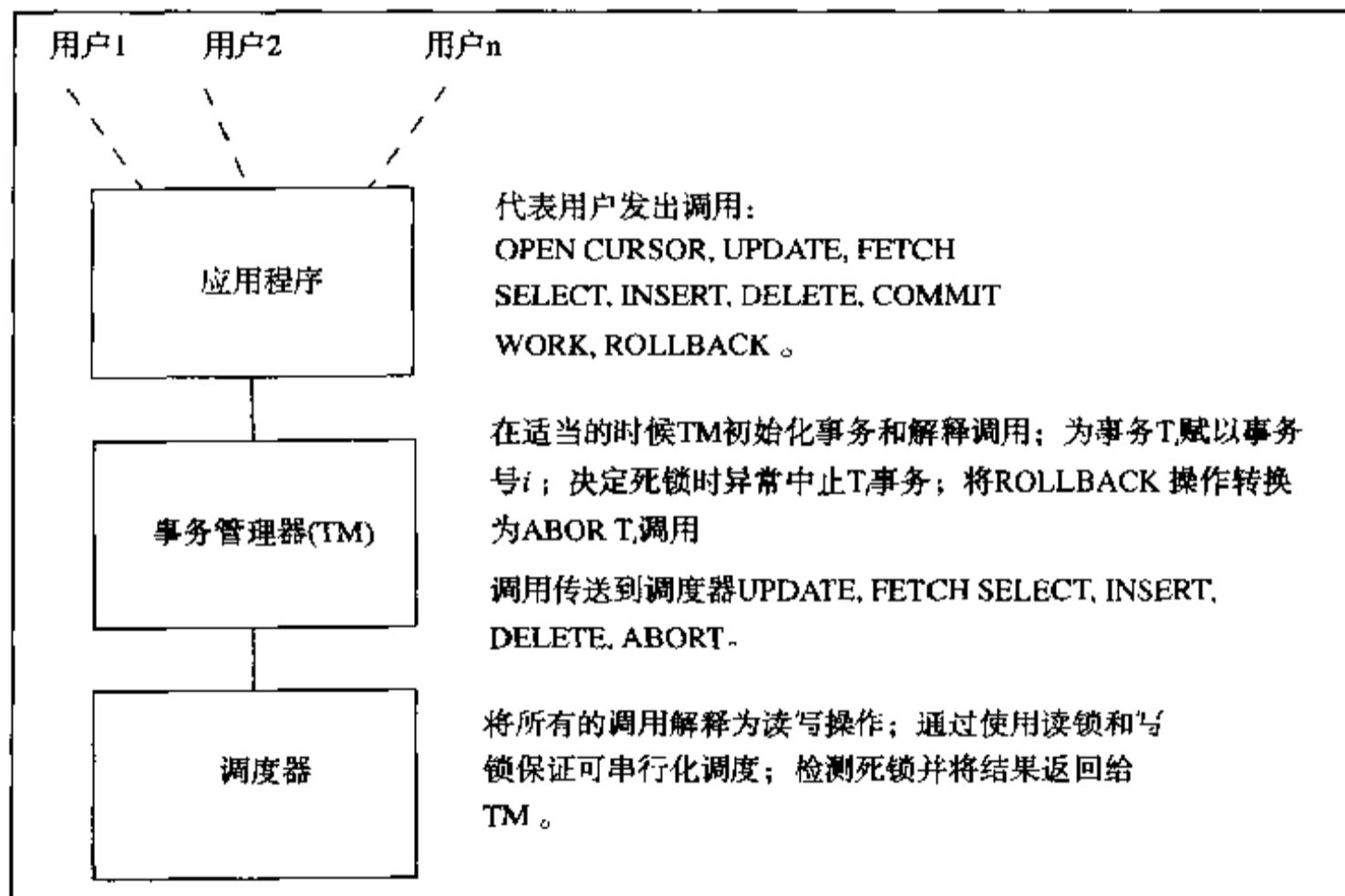


图10-8 事务系统的层次结构图

在事务型数据库系统中，通常有大量的用户在终端上完成他们的工作——用户不会注意到事务或代表他们的数据库调用。通常的执行过程中，应用程序发出一个代表某个用户的操作，数据库收到这个操作之后创建一个处理该操作的事务，并给该事务编号*i*。做这项工作的数据库系统模块称为事务管理器，或者缩写为TM，如图10-8所示。事务管理器TM将诸如UPDATE、FETCH、SELECT、INSERT和DELETE这样的操作传递给调度器。

调度器的工作是保证“具有交错存取操作的经历H”中所有事务的操作是可串行化的。调度器通过使某些事务操作等待，让另外一些事务操作先执行来达到这一目标，使最终产生的

经历H是可串行化的。当某个事务的操作被要求等待时，相应的用户也必须处于等待状态，但是由于时间片只通常是很短的一段时间，用户根本觉察不到。

我们已经提到过的一个简单的方法可以被调度器用在调度事务型操作上——产生严格的串行经历，即保证一个事务的所有操作（包括最后的提交或者回退）在任何后续事务的操作执行之前全部完成。就像我们在10.2节中看到的那样，从系统性能的角度出发，严格的串行经历是非常不可取的。在本节中我们将解释调度器是如何通过加锁的机制来保证一个拥有大量交错存取操作的调度的可串行化的。

在商业数据库系统中用来保证事务一致性的一个非常著名的锁机制称为两段锁协议，缩写为2PL，它被大多数商业数据库系统所采用。

定义10.4.1 两段锁协议，即2PL 以下三条规则用来规定在2PL中是如何申请和释放锁的。

1) 当事务T_i试图读取某个数据项时，即发出操作R_i(A)，调度器解释该操作并代表该操作第一次向该数据项加读锁RL_i(A)。同样，当事务T_i试图对某个数据项进行写（更新）操作时，即发出操作W_i(A)，调度器解代表该操作第一次向该数据项加写锁WL_i(A)。^Θ

2) 在获得某个数据项的锁之前，调度器要求请求事务处于等待状态，直到该数据项上没有冲突锁存在为止。（冲突锁是由于我们下面将要定义的冲突操作引起的。例10.4.2将向我们展示这样的一个等待操作是如何影响经历执行的。）

同一数据项上的两个锁被称为冲突的当且仅当它们被不同事务要求并且这两个锁至少一个为写锁。

3) 加锁有两个阶段：申请阶段，在该阶段事务获得所要求的锁；释放阶段，该阶段事务释放拥有的锁。调度器必须保证一旦释放阶段开始，不允许该事务再申请新的锁。也就是说不允许一个事务释放了某个锁之后又去申请别的锁。 ■

两段锁理论的规则允许锁在事务提交之前被事务释放，只要是释放之后不再申请新的锁。然而，在大多数商业数据库中往往在提交阶段（COMMIT）的最后一刻才释放所有的锁。我们假定在后面的讨论中就采取这个方法除非在提交语句（COMMIT）之前明确执行解锁操作（UNLOCK）。同一个事务所拥有的锁是不会冲突的——特别是，在一个数据项上拥有读锁的事务可以进一步申请写锁，只要该数据项上没有其他事物的读锁存在。在一个数据项上拥有写锁的事务是不必要申请读锁的。这一点上，我们说读锁比写锁要弱一点，当某个事务向一个已经拥有写锁的数据项上提交申请读锁的操作RL_i(A)时马上会取得成功。

冲突锁的定义很明显是要保证执行冲突操作的两个事务是可串行化的，以便对应的前趋图中不会产生环。当两个事务在某个数据项上有冲突操作时，先访问该数据的事务就先取得相应的锁先执行，而在该经历对应的任何一个等价串行经历中另一个事务就必须在前一个事务的后面了。如果第二个事务拥有第一个事务在稍后的执行中需要的一个锁，那么就会导致死锁，这样其中一个事务就必须异常中止。异常中止的事务将从经历中删除（在前趋图中也将删除相应的顶点），以便结果是可串行化的。因此两段锁原理保证了一个可串行化的经历。在我们严格地展示这一点之前我们需要处理死锁问题。

^Θ 尽管读锁和写锁在逻辑上已经充分，但在一些数据库系统中往往还引入了许多其他类型的锁，称之为粒度锁或多粒度锁或意向锁，这些都超出了本书所讲述的范围。可以参考本章后面的推荐读物[1]和[2]，专门对这些锁进行了讨论。

首先，我们想说明定义10.4.1中第3条规则的重要性，该规则要求事务加锁时申请锁阶段后面跟一个释放锁阶段。对于锁机制而言，这个“两段规则”对于保证经历的可串行化有着重要的意义。

例10.4.1 让我们回想一下在例10.3.1中向大家展示的那个不可串行化的经历H1。下面是对经历H1中的操作进行重新排序之后的得到经历H1'，这个经历也是不可串行化的。

$H1' = R_1(A) R_2(B) W_2(B) R_2(A) W_2(A) R_1(B) C_1 C_2$

现在我们要证明如果允许该经历违反“两段规则”，那么我们可以使得该经历在执行锁机制时满足其余的规则，而且这个不可串行化的经历H1'仍能得到执行。为了说明锁操作，我们在扩展的经历H1'中加入了一些关于锁的操作。操作RL代表获取读锁，操作WL代表获取写锁，操作RU代表释放一个读锁，操作WU代表释放一个写锁。因此RL_i(A)代表事务T_i对数据项A加读锁，WU_i(A)代表事务T_i释放数据项A上的写锁。通常，我们假定一个事务拥有的所有的锁在该事务的提交阶段一起释放。下面是扩展的经历H1'：

$H1': RL_1(A) R_1(A) RU_1(A) RL_2(B) R_2(B) WL_2(B) W_2(B) WU_2(B) RL_2(A) R_2(A) WL_2(A) W_2(A)$
 $RL_1(B) R_1(B) C_1 C_2$

在这个扩展的经历中每个事务在进行读写操作前都执行了相应的加锁操作，RL或WL。因此定义10.4.1中的规则1得到满足。此外规则2也是满足的，因为在向一个事务授予某个锁时，该数据项上不存在相冲突的已经授予其他事务的锁。经历H1'中的第3个操作RU₁(A)释放了A上的读锁，以免和该经历的第11个操作WL₂(A)相冲突。同样地，第8个操作WU₂(B)释放了B上的写锁，以免和该经历的第13个操作RL₁(B)相冲突。在该扩展经历中唯一违反的规则就是：“两段规则”，因为两个事务都在释放了某个锁之后又申请了别的锁。这说明了“两段规则”对于通过锁机制来保证经历的可串行化有着重要的作用。 ■

等待图

等待图是由许多事务型锁调度器维护的一个有向图，顶点由当前所有活动事务组成，顶点T_i到T_j的边T_i→T_j代表事务T_i正在等待对某个数据项加锁，而该数据项当前被事务T_j锁定着。当下列事件发生时，调度器对等待图进行更新：(1)一个新的事务产生（新的顶点将被加入到等待图中）。(2)一个事务被迫进入等待状态（一条新的边加入到等待图中）。(3)一个老的事务提交（对应的顶点将从等待图中删除，该事务拥有的锁将全部释放，每个数据项的等待队列中的第一个事务将获得相应的锁，这可能将导致其他事务等待这个新的事务释放锁，但该算法是相对有效的）。当等待图中出现回路时，就意味着产生死锁。

调度器会在等待图发生变化后的一定时间内检测等待图中是否有回路（有时也称为环）存在。如果有回路存在，调度器就会选择某些事务作为牺牲品，让其强行异常中止。选择牺牲品的标准可以是各种各样的：中止最年轻的事务、中止已经完成的操作最少的事务等等。

例10.4.2 我们再次引用例10.4.1中不可串行化的经历H1'。当应用2PL原则时会发生什么情况呢？

$H1' = R_1(A) R_2(B) W_2(B) R_2(A) W_2(A) R_1(B) C_1 C_2$

应用2PL原则后，经历H1'引起以下一系列的事件：

$RL_1(A) R_1(A) RL_2(B) R_2(B) WL_2(B) W_2(B) RL_2(A) R_2(A) WL_2(A)$ （与前面的RL₁(A)相冲突，因此T₂必须等待） $RL_1(B)$ （与前面的WL₂(B)相冲突，因此T₁必须等待，这样等待图中就出现

了回路，于是我们必须异常中止某个事务——比如选择 T_1 作为牺牲品） A_1 （ T_1 拥有的锁全部释放，于是导致 T_2 等待的操作 $WL_2(A)$ 现在可以执行了） $W_2(A) C_2$ （假定事务 T_1 的工作重新开始，并标记为事务 T_3 ） $RL_3(A) R_3(A) RL_3(B) R_3(B) C_{3,0}$ 。

调度器通过强行等待和异常中止一个事务并在稍后重新启动该事务的方法，使得两个事务最终都能成功运行。 ■

定理10.4.2 加锁定理 遵守2PL原则的事务型操作经历必定是可串行化的。 ■

很明显，这是一个重要的定理，因为大多数商业数据库系统中对事务隔离性的保证依赖于这个定理的结论。在证明之前，我们先来证明一个重要的引理。

引理10.4.3 假定 H 是一个遵守2PL协议的事务操作经历（我们称之为2PL经历）。如果在经历 H 的前趋图 $PG(H)$ 中存在一条边 $T_i \rightarrow T_j$ ，那么必然存在某个数据项 D ，经历 H 中存在该数据项的两个冲突操作 $X_i[D]$ 和 $Y_j[D]$ ，使得 $X_i[D] \ll_h Y_j[D]$ 。

证 由于存在边 $T_i \rightarrow T_j$ ，根据前趋图和冲突操作的定义，必然存在两个冲突操作 $X_i[D]$ 和 $Y_j[D]$ ，使得 $X_i[D] \ll_h Y_j[D]$ 。根据两段锁的定义，其中任何一个操作必然存在加锁和解锁的操作，如 $XL_i[D] \ll_h X_i[D] \ll_h XU_i[D]$ 和 $YL_j[D] \ll_h Y_j[D] \ll_h YU_j[D]$ 。（解锁操作也许隐含在事务的提交操作中，但是我们选择了在经历 H 中明确表示出来，使它们出现在提交操作的前面。）

现在考虑对于出现在 $XL_i[D] \ll_h X_i[D] \ll_h XU_i[D]$ 中的操作，数据项 D 上的X锁被事务 T_i 拥有；对于出现在 $YL_j[D] \ll_h Y_j[D] \ll_h YU_j[D]$ 中的操作，数据项 D 上的Y锁被事务 T_j 拥有。因为操作X和Y是冲突的，因此相应的锁也是冲突的，因此这两个锁的存在时间不能重叠。但是根据 $X_i[D] \ll_h Y_j[D]$ ，这两个操作的加锁必须按以下次序发生：

$$XL_i[D] \ll_h X_i[D] \ll_h XU_i[D] \ll_h YL_j[D] \ll_h Y_j[D] \ll_h YU_j[D]$$

于是引理所阐述的内容得到证明。 ■

有了这个引理我们可以证明定理10.4.2。

定理10.4.2证 我们所要证明的是每个2PL经历都是可串行化的。假定这个结论不成立，那么必然存在一个2PL经历 H ，该经历对应的前趋图 $PG(H)$ 存在环 $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_1$ 。根据引理10.4.3，对于前趋图 $PG(H)$ 中的每一对事务 $T_k \rightarrow T_{k+1}$ ，必然存在某个数据项 D_k ，使得 $XU_k[D_k] \ll_h YL_{k+1}[D_k]$ 。对于所有的 k ，我们可以将该序列按以下形式写出来：

1. $XU_1[D_1] \ll_h YL_2[D_1]$
2. $XU_2[D_2] \ll_h YL_3[D_2]$
- ...
- $n-1.$ $XU_{n-1}[D_{n-1}] \ll_h YL_n[D_{n-1}]$
- $n.$ $XU_n[D_n] \ll_h YL_1[D_n]$

根据两段锁协议，任何事务 T_i 的所有加锁必须在解锁之前完成。根据第1行和第2行，事务 T_2 分别在数据项 D_1 和 D_2 上进行加锁和解锁，根据两段锁性质我们可以断定 $YL_2[D_1] \ll_h XU_2[D_2]$ 。据此，我们可以进一步由第1行和第2行的事实以及传递性得到：

$$XU_1[D_1] \ll_h YL_3[D_2]$$

换句话说，在事务 T_i 对某个数据项加锁之前，事务 T_i 对另外某个数据项执行了解锁。以此类推直到第 n 行，我们可以进一步得出结论：

$$XU_1[D_1] \ll_h YL_1[D_n]$$

但是这一结论违反了事务T的所有加锁必须在解锁之前完成的两段锁性质。因为我们假定这是一个2PL经历，导致矛盾，所以前趋图PG(H)中不可能存在环。因此经历H必定是可串行化的。 ■

10.5 隔离级别

定理10.4.2告诉我们一个调度器通过应用两段锁协议可以保证产生可串行化的经历。这意味着对任何数据的读写操作，调度器都将代表该事务先申请对该数据的读锁或写锁，并且调度器将一直保留这些获得的锁直到能够将它们安全释放为止。2PL协议的第3条性质确保一个事务不能在释放一个锁之后又去申请不同的锁，因此我们必须保留所有的已经获得的锁直到没有新的数据访问操作产生为止。事实上，我们假定一个事务保留所有属于该事务的锁直到该事务提交或者异常中止。

然而，保证很好的串行性的调度对性能有较大的影响。当我们加入更多的线程以及并发事务时，特别是这些事务处理大量的常用数据时，对数据的存取就会产生大量的重叠，锁冲突意味着越来越多的事务处于等待状态。随着并发性的增加死锁的次数也随之增加，异常中止某些事务并在随后重新启动这些事务在很大程度上是对计算机资源的浪费。但事实上更严重的问题是，伴随着大量的并发活动事务，越来越多的事务会由于锁的冲突进入等待状态。在某些情况下这一效果很快会得到证明，如果我们为了事务的并发增加线程，我们事实上减少了处于等待状态的活动的并发事务的数量。性能问题不是因为异常中止某些事务而又再次重新启动所浪费的CPU资源，而是因为CPU从来没有被充分地使用，这是由于我们没有获得足够有效的并发度来保证活动事务的事务逻辑通过I/O的重叠而使CPU保持繁忙状态。

由于这个原因，多年前人们提出数据库系统的事务调度器应该降低两段锁协议的要求，从而降低事务间由于访问冲突而产生的等待。先假定这种弱化可以增加有效的并发。当然通过较弱的规则要求，我们将不能对事务的隔离性提供强有力的保证，这样就有可能使那些具有冲突操作的事务相互影响产生不一致的结果，而这在严格串行执行时是不会发生的。但是，现在各种弱化的形式却已经是不争的事实，ANSI SQL-99标准支持一种称为隔离级别的特性。（注意，SQL-99标准定义的隔离级别与早期定义的隔离度(0°, 1°, 2°, 3°)有很大的不同，隔离度在一些产品中仍然在以不同的名称使用。参见Gray和Reuter的文章(推荐读物[2])。）

在SQL-99的语法中允许在启动事务（访问任何数据）的SQL语句之前设置隔离级别，使用的语句通常为Set Transaction语句。以下是SQL-99提供的四个隔离级别，向程序员依次提供更高的隔离保证级别和限制：

- 1) Read Uncommitted (有时也称为“脏”读)。
- 2) Read Committed (比DB2的“游标稳定”隔离级别要弱一些)。
- 3) Repeatable Read。
- 4) Serializable。

最后一个隔离级别4)Serializable等价于我们在定义10.3.4和10.4.2中定义的可串行性，然而前面三个隔离级别1)、2)、3)要显得更为宽松，因此也提供了较低的隔离保证。ANSI SQL-99标准的Set Transaction语句格式如下：

```
SET TRANSACTION {READ ONLY | READ WRITE}
    ISOLATION LEVEL {READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ |
        SERIALIZABLE};
```

一个只读事务在它的执行过程中不能有任何更新操作（如果有更新操作会导致错误信息），然而读写事务是允许更新的。事务的隔离级别和执行更新操作的能力通常是分别定义的，只有在隔离级别READ UNCOMMITTED下更新操作是不允许的。下面我们将逐一讨论这些隔离级别，看看它们之间到底有哪些不同之处。

在接下来讲述的内容中我们需要理解短期锁（short-term lock）的概念。实际上，数据项上的短期锁只是在执行相关存取操作的一段足够长的时间内保留。例如事务将完成RL,(A) R,(A) RU,(A)序列。对于这种锁，仅仅当拥有该锁的事务正在执行该操作时才会阻塞其他试图访问该数据的事务，当然即使是申请短期锁的事务，也必须等待该锁直到被授予为止（这时，该数据项上已经没有其他事务的冲突操作存在了）。由于大多数事务会访问多个数据，因此申请短期锁的事务往往不满足两段锁协议的第3条规则，所以有可能产生不可串行化的行为。当然这是低隔离级别的极端情况；我们往往假定在弱隔离保证不至于产生负面影响的情况下才使用较低的隔离级别。相对于短期锁的另一种锁是长期锁（long-term lock），这种锁将一直保留直到事务提交为止。

图10-9列出了SQL-99各个隔离级别下哪些锁应该作为长期锁。（不作为长期锁的以短期锁对待）。在图10-9中，我们将对整个数据库表上的记录加锁和对满足某个谓词的记录（比如city='Boston'的记录，这些谓词往往出现在WHERE子句中，用来确定要对哪些记录进行读和修改）加锁这两种操作区别对待。我们要指出的是为了保证真正的可串行化我们需要在谓词上加锁，就像在数据项上加锁一样；我们将在下面涉及可串行化的隔离级别时讨论这一点。作为将要讨论的内容的一个简单预览，我们将会发现我们需要谓词锁来避免特定的诸如“幽灵问题”这样的不可串行化问题。

	记录上的写 锁为长期锁	记录上的读 锁为长期锁	谓词上的读锁和写 锁为长期锁
Read Uncommitted (dirty reads)	否(但它是只读的)	根本没有读锁	根本没有谓词锁
Read Committed	是	否	短期读谓词锁 长期写谓词锁
Repeatable Read	是	是	短期读谓词锁 长期写谓词锁
Serializable	是	是	长期读谓词锁 长期写谓词锁

图10-9 SQL-99 隔离级别^② 中的长期锁行为

对于事务调度器而言，在同一个事务执行环境中支持具有不同隔离级别的不同事务并发执行是完全可能的。这也是SQL-99标准之所以允许程序员在程序中使用Set Transaction语句并在后继事务开始执行之前重新设置事务隔离级别的原因。不同的进程可以以不同的隔离级别

② 需要注意的是SQL-99标准并没有利用锁的行为来定义隔离级别。取而代之的是以许多英语中的现象来表达事务的不规则，并根据各个隔离级别中可能出现的这些现象来描述这些隔离级别。然而，这样定义隔离级别是有缺陷的（参考本章后面的推荐读物[5]），在此处我们只提供从锁的角度对隔离级别的定义。但这并不意味着锁机制是实现各种隔离级别的唯一方法。锁机制是大多数商业数据库采用的方式，但不是唯一的方式。

执行各自的代码，但是一切都会按照我们所期望的那样运行。

1. Read Uncommitted隔离级别

从图10-9的第一行，我们可以看到以Read Uncommitted隔离级别执行的事务无论访问什么数据都不加锁。看上去这似乎会导致脏写的现象（参见例10.3.2）。对那个例子中的经历H2，我们重新来考察各个数据项的值，一个事务对数据项A的写操作的结果被另外一个事务对A的写操作覆盖，而这两个事务在写操作之前读到的是相同的A值。

$H2 = R_1(A, 100) R_2(A, 100) W_1(A, 140) W_2(A, 150) C_1 C_2$

这样两个事务对一个账户余额的更新产生相互干扰，导致一个事务增加40的操作结果被另一个事务增加50的操作覆盖，总共应该增加90的结果丢失。然而这个问题在SQL-99标准中的Read Uncommitted隔离级别下不会发生，因为在此隔离级别下更新操作是不允许的。如果执行了以下的Set Transaction语句：

```
set transaction isolation level read uncommitted;
```

那么事务中的更新操作缺省地变成只读的（注意，在其他隔离级别下更新操作具有读写属性）。此外，如果Set Transaction语句如下所示：

```
set transaction read write isolation level read uncommitted;
```

就会导致错误。一个Read Uncommitted事务是不允许更新数据的。因此经历H2中的脏写是不会发生的。

然而，读脏的确有可能发生。银行里读取账户余额数并进行求和操作的事务，会忽略任何其他事务在这些数据上加的锁，因此有可能读到未被提交的数据。（我们可以注意到在图10-9中Read Uncommitted隔离级别下不需要申请任何锁就可以访问数据，碰到其他事务的写操作也无须等待，而只是简单地忽略这些锁。因此定义10.4.1中的属性[1][2][3]都被忽略了。）结果是，一个Read Uncommitted的事务计算账户总余额的操作可能导致一个无效的数据，因为它可能读到了无效的中间结果。当然这个结果是不会让一个希望得到确切结果的核算人员满意的。但也许对于银行经理来说没什么大的关系，因为他只是每个星期过问一下银行的总余额。

2. Read Committed隔离级别和游标稳定性

在图10-9的第二行，我们可以看到Read Committed隔离级别下写锁作为长期锁，而读锁只是短期锁。这使得我们在图10-7中看到的并发事务之间产生的三种冲突操作中的两种不会发生：(2) $W_i(A) \rightarrow R_j(A)$ 和 (3) $W_i(A) \rightarrow W_j(A)$ 。在这一隔离级别下一个事务不能读或者写一个已经被其他事务写过的数据，直到那个事务提交为止。特别是不读未提交的脏数据，只能读取已经提交的事务写过的数据。这也说明了该隔离级别的名字“Read Committed”，同样前一隔离级别的名字“Read Uncommitted”也可以按这种思路理解，读取未提交的更新数据是可能的。在Read Committed隔离级别下一条记录上可能产生的唯一的冲突操作是(1) $R_i(A) \rightarrow W_j(A)$ 。那么什么样的异常情况会在该隔离级别下发生，而不会在一个串行经历中发生呢？下面两个例子解释了两种这样的异常。

例10.5.1 不可重复读异常 考虑一个事务，它在一组行上打开了一个游标并依次读取后续行的值，然后再次在这组行上打开游标第二次读取这些行的值。我们会发现在Read Committed隔离级别下两次遍历游标读到的数据可能不完全一样。在Read Committed级别下，

一个事务只是在读取数据的过程中持有读锁，一旦读操作完成就会释放相应的锁。因此极有可能其他事务在该事务的两次打开游标操作之间更新了数据。下面通过一个带值的扩展经历来说明这个问题，这个经历中事务T₁只读一个数据项A：

H4=R₁(A,50)W₂(A, 80)C₂ R₁(A, 80)C₁

这一异常，通常称之为非可重复读。

不可重复读对于大多数数据库应用来说可能并不是什么问题，因为对于一个应用来说很少需要对一个数据读两次。我们给出这个例子的一个主要原因是因为下一个更强的隔离级别是可重复读Repeatable Read，许多教科书使用该例子中的经历H4来说明这两个隔离级别的区别。然而，在Read Committed级别下可能会发生更为严重的错误，与经历H2中的丢失更新有得一比。

例10.5.2 学究丢失更新异常 回忆一下在经历H2中说明的丢失更新异常：

H2 = R₁(A, 100) R₂(A, 100) W₁(A, 140) W₂(A, 150) C₁ C₂

在Read Committed级别下，这类异常是不会发生的，不是因为读写之间的任何冲突，而是由于写锁是长期锁，因此操作W₂(A,150)不可能覆盖事务T₁对A设置的140。然而我们再来考虑如下的经历H5：

H5 = R₁(A, 100) R₂(A, 100) W₁(A, 140) C₁ W₂(A, 150) C₂

因为在事务T₁试图覆盖A之前T₁已经提交，再也没有写锁来阻止丢失更新异常的发生了。H5是应用中一种非常典型的错误，因此说明了Read Committed级别下一个非常严重的错误。事实上，如果进入锁调度器中的序列如经历H2，那么最终得到的序列可能是H5，因为当W₂(A,150)出现在调度器中时，调度器会让事务T₂等待直到事务T₁提交为止，结果是产生如同H5这样的经历。我们称这种异常为“学究丢失更新”，从而有别于经历H2中的丢失更新。H5并没有和H2有什么多大的不同，因此“学究”一词只是戏称而已，就象棋赛中的“学究对手”一词。

DB2和其他许多数据库产品提供了称为游标稳定性(cursor stability)的隔离级别，该级别比Read Committed要稍微严格一些。就像Read Committed一样游标稳定性将写锁作为长期锁，而读锁不是长期锁；但是它能阻止例10.5.2中的学究丢失更新，只要所有的数据更新操作是通过游标进行的（即，UPDATE...WHERE CURRENT OF CURSOR...）。名字游标稳定性来源于一种特别类型的锁，这种锁比读锁要强一点，加在通过游标读取的数据上，即使数据已经被读取仍然保留该锁直到游标移到下一行。（当然，如果在游标移到下一行之前要对当前行进行更新，那么在当前行保留写锁。）这一加锁并没有阻止非可重复读的异常。然而，在图10-10中的给SFBay支行的每个人增加\$10.00利息的代码不会出现丢失更新的异常，因为对游标的当前行始终保留相应的锁。

我们也可以采用图10-11的Read Committed（甚至在Read Uncommitted）事务隔离级别下的单条语句更新来避免丢失更新异常，因为Update语句在更新过程中会保留每行上的锁：读操作和写操作作为一个原子操作对逐个执行。

然而，图10-12的更新单行的代码仍然可能产生丢失更新异常，即使采取游标稳定性也不能幸免。因为在第一条语句读了该数据后（不通过游标读），在第二条语句对其更新之前，没有对该数据进行加锁。

```

exec sql declare cursor deposit for select balance from accounts
    where branch_id = 'SFBay' for update of balance;
exec sql open c;
(now loop through rows in cursor, and for each pass do)
    exec sql fetch c into :balance;
    balance = balance + 10.00;
    exec sql update account set balance = :balance
        where current of deposit;
(end of loop)
exec sql close deposit;
exec sql commit work;

```

图10-10 在游标稳定性隔离级别下无丢失更新的逻辑

```

exec sql update accounts set balance = balance + 10.00
    where branch id = 'SFBay';
exec sql commit work;

```

图10-11 在Read Uncommitted隔离级别下无丢失更新的逻辑

```

exec sql select balance into :balance from accounts
    where acct_id = 'A1234';
balance = balance + 10.00;
exec sql update accounts set balance = :balance
    where acct_id = 'A1234';
exec sql commit work;

```

图10-12 在游标稳定性隔离级别下可能丢失更新的逻辑

似乎并非所有使用Read Committed隔离级别的程序员都会意识到可能发生异常，因为游标稳定性的优点似乎被ANSI SQL委员会过高估计了。使用Read Committed隔离级别的数据库没有必要提供游标稳定性。对于一个DBA而言可以对正在使用的任何数据库进行测试，看使用了游标之后例10.5.2是否会发生异常，这实在是个好主意。

3. Repeatable Read隔离级别

图10-9的第三行显示Repeatable Read隔离级别支持对单个数据项（比如行）进行两段加锁的所有要求；它对所有的数据项提供长期锁，只对一种称为读谓词锁的类型提供短期锁。事实上这一隔离级别解决了到目前为止我们提到的所有异常问题。特别是，它解决了例10.5.1中出现的一个事务中对某行不可重复的异常。这是因为读事务对数据保留长期的读锁，因此没有任何其他事务能够更新该数据直到读事务提交为止。因此将这一隔离级别命名为“Repeatable Read”。

真正的可串行性是由图10-9第四行的隔离级别提供的，它提供了Repeatable Read这一级别所能提供的所有保证并且使读谓词锁变成长期锁，后者我们还没有讨论。事实证明，谓词锁能够使交错执行的事务避免一种称为幽灵（或者称为幽灵更新）的异常。这是一个相当精细的问题，我们将在下一小节讨论这个问题。

4. 可串行性和幽灵更新

我们提供一个在Repeatable Read隔离级别下的事务操作过程中可能产生的异常。

例10.5.3 幽灵更新异常 考虑一个在Repeatable Read隔离级别下的事务，该事务打开了一个包括银行某个支行的accounts表中一组行的游标，提取后续行并对它们的余额进行求和，然后测试该总额是否和表branch_totals中具有相同的branch_id的行中的总额相等。如果不相等，则打印错误信息。这个任务也许可以用以下的逻辑来实现：

```
exec sql select sum(balance) into :total from accounts
  where branch_id = :bid;
exec sql select balance into :bbal from branch_totals
  where branch_id = :bid;
if (bbal <> total)
  (print out serious error);
exec sql commit work;
```

如果将该事务逻辑要读的数据或者其他并发事务要更新的数据都加上长期锁，看上去我们似乎在Repeatable Read隔离级别下已经做了所有的努力。然而，一种不同的异常仍然可能发生，因为直到现在我们都没有考虑如下这种操作的影响：插入一个新行的操作和在老行上进行更新的操作将改变第一条Select语句的WHERE子句返回的行集合的成员。必须注意到这个计算一组行的总和的Select语句不可能同时访问该组中的所有行。它必须一边走一边判断该组中到底有哪些行，也许是通过索引每次读入一定范围中的行或者是读入整张accounts表中的行，然后逐行判断是否是符合要求的行。由于这需要花费一定的时间，因此当事务进程读入一串信息并对它们进行确认时，可能同时会有其他的事务插入一个新的行或者更新了老的行使得符合要求的行集变大，但前者并没有注意到这一点。新的行可能被忽视，因为该行所在的数据页面也许已经被负责求和的Select语句的WHERE子句检查过了，或者该行入口位置所在的范围已经被传递给WHERE子句去检查了。这时就会导致异常，接下去我们将会看到这一点。

我们使用术语I_i(A)来表示事务T_i插入数据项A的操作。当我们指定插入的列值时，我们可以使用以下的标记：I_i(A, branch = SFBay, balance = 50)。（A用来标记给出该行的唯一标识符的字段值，比如表accounts中的acct_id。）现在考虑下面一个经历：

R₁(谓词:branch_id = 'SFBay') R₁(A1,100.00) R₁(A2,100.00) R₁(A3,100.00) (我们假定这些都是最初在这个支行中找到的行) I₂(A4, branch=SFBay, balance = 100.00) (这个新的行不会被T₁注意到) R₂(branch_totals, branch_id = 'SFBay', 300) W₂(branch_totals, branch_id = 'SFBay', 400) C₂(现在事务T₂已经修改了这一支行的branch_totals行，而T₁还没有注意到这一点) R₁(branch_totals, branch_id = 'SFBay', 400) (打印出错信息) C₁。

概括地说，以上的经历说明了这样一件事：事务T₁先确定表accounts中哪些行的branch_id是'SFBay'，然后再去找acct_id值分别为A1、A2、A3的行。然后读出这三个行的余额相加，得到的值为\$300.00。在这时，另外一个事务T₂在SFBay支行创建了一个新的行A4（这时显然需要重新统计，因为开始只有三行）。T₂接着更新了值为SFBay的branch_totals从而反映出为该支行增加的账号金额。之后，事务T₁读到了这个被T₂更新过的值，它看到的是\$400.00，然而三行的总额只有\$300.00。 ■

期间没有任何措施来阻止T₂加入账号A₄，然而这一更新恰恰违反了SFBay支行的余额必须等于该支行所有账户的余额值总和这个一致性规则。这种逻辑并没有忽视任何行上的锁，但

是异常仍然发生了。原因就是这个新行A4插入表中时，事务T₁已经将谓词branch_id = 'SFBay'的检测完成了。如果事务T₁能够注意到行A4，当然会在行A4上加锁，但是行A4加入的太晚了。

这是否意味着前面所说的串行性理论存在问题了呢？是否两段锁协议2PL并非包含着真正的可串行性呢？当然不是，这只是意味着我们没有在足够的对象上加锁，我们没有对表中的某些行加锁。可串行性要求事务访问的所有数据对象都出现在10.3.3中定义的前趋图中。当事务读到谓词branch_id = 'SFBay'时，比较简单的谓词锁公式认为可以在相应的信息上加读锁，这些信息由包括余额值的符合要求的相应行组成，即该支行的三行{A1,A2,A3}。当事务T₂向该支行插入一个新的行A4时，这就隐含地在该谓词上加了写锁，因为当计算谓词时这一变动将改变可用的信息（结果行集）。因此这两个操作在这个谓词上是一对冲突操作；因此当有长期读谓词锁存在时，这样的经历是不允许的。扩展例10.5.3中的经历，使其包含合适的读锁和写锁，我们可以得到如下的经历：

RL₁(谓词branch_id = 'SFBay') R₁(谓词branch_id = 'SFBay') RL₁(A1,100.00)
R₁(A1,100.00)RL₁(A2,100.00) R₁(A2,100.00)RL₁(A3,100.00) R₁(A3,100.00)WL₂(A4, branch = 'SFBay') (准备执行插入I₂(A4, branch_id='SFBay', balance=100.00)，但是这和前面的该谓词上的RL₁冲突，因此T₂必须等待) R₁(branch_totals, branch_id = 'SFBay', 300) (总和是正确的，没有出错) C₁ (现在所有的锁都释放了) I₂(A4, branch = 'SFBay', balance = 100.00)R₂(branch_totals, branch_id = 'SFBay', 300) W₂(branch_totals, branch_id = 'SFBay', 400)C₂。

现在这个经历是完全可串行化的了。如图10-9所示，SQL-99标准中可Serializable这个隔离级别对谓词使用长期的读锁和写锁。因此例10.5.3的幽灵更新问题可以避免了，使用这个加锁协议的经历是完全可串行化的。然而读者不应把在谓词上加锁看得太学术化。任何加锁类型只要能够保证当一个事务在向一个基于某个谓词锁的行集合添加行时，其他事务不能访问该集合，就能去除幽灵更新问题。一些数据库系统通过加锁整个表来达到这个目的，这些不必要的太粗的加锁粒度会降低并发性，但却通过简单的系统逻辑保证了可Serializable隔离级别。目前最常用的谓词锁技术是键值范围的加锁(key-range locking)，该内容的细节已经超出了本书的范围。关于这一技术的介绍，可以参考本章末尾推荐读物中Gray和Reuter的文章[2]。

我们将提一下SQL-99的另一条规则：如果一个数据库系统希望遵循该标准，但却没有提供某一个隔离级别，那么它必须至少提供Set Transaction语句中的安全性。比如，如果数据库系统没有Read Committed这一隔离级别，但确实含有可重复读(Repeatable Read)的特性，那么当以下的语句执行之后：

```
exec sql set transaction isolation level read committed;
```

系统隐含地就有了可Repeatable Read(重复读)这一级别。

10.6 事务恢复

我们前面已经提到过，数据库中的数据是存放在磁盘上的，磁盘是一种非易失性存储媒介。此处的非易失性可以通过下面这个例子来说明：当系统突然断电时，磁盘上的数据不会因此而丢失。相比之下，存放在内存中的数据一旦断电就会丢失，因此通常称内存为易失性的存储媒介。还有其他一些原因也会导致内存的不可靠性，比如由于操作系统或者数据库系

统本身的不完善导致系统崩溃。许多系统设计人员认为这样的错误是不可避免的，这些错误导致我们不能确信内存中的内容会在一段较长的时间（比如几个月或者几年）内保持稳定。如果可以，我们将在任何时候将数据存放到磁盘上去而不必担心这些数据会丢失。

然而，还能记得我们在9.2节中讨论的，相对访问内存而言访问磁盘的速度是很慢的。我们需要将磁盘中的数据页面读到内存缓冲区中，以便为计算机指令提供高速的随机访问。一旦一页数据读入内存缓冲区中，我们将设法保存该页面以便在不久该页面中的数据被再次引用，从而减少系统磁盘I/O开销。为了支持缓冲机制我们使用了一种称为后备缓冲的方法，该方法允许系统收到一个磁盘页面的请求时，先去试着散列到后备表中该页面的项。如果在后备表中有这一项，说明对应的磁盘页面早已读入内存缓冲区中；否则需要从磁盘中读入该页面。自然，我们希望保存在内存中的磁盘页面是那些最为常用的数据页面，这一信息可以使用一种称为LRU缓冲策略获得（LRU是Least Recently Used的缩写）。该策略的基本想法是：读到内存中的数据页面会一直保留在内存中，直到一个新读入的磁盘页面需要空间并且所有的内存缓冲都被占满的时候为止。这时，必须释放某些缓冲区来腾出一定的空间，这些将要释放的缓冲区是那些没有被经常使用的页面——最近最少使用的页面。

正如我们希望将最常用的数据页面保留在内存中从而可以被不同的事务一次又一次的使用，我们同样不希望每次数据页面被某个事务更新之后都要做写回磁盘的操作。如果这是经常要被更新的页面，比如银行账户行，我们可以允许成百上千次的更新而不做写回操作。对于同类事务应用而言这是最重要的一种优化。在此重复一点：我们不希望一旦事务对某些页面做了更新并提交之后就写回这些数据页面。取而代之，我们通常将这些常用的数据页面保留在内存中直到它们变得不再常用从而成为LRU的数据页面被别的数据页面替换出缓冲（注意更新过的数据页面在它们腾出缓冲之前必须写回到磁盘上），否则我们在经过一定周期之后强制它们写回磁盘。（我们会在稍后讨论强制写回磁盘页面的问题）。大多数情况下，我们不会这么做，而是依靠LRU策略。一个数据页面被称为是脏的，如果从上一次写回磁盘之后被某些事务做了更新操作。我们通常会将脏的数据页面保留在内存中直到更新该数据页面的事务提交。（我们会在习题中看到在LRU策略下读写缓冲区的一些问题。）

但现在我们有一个问题。假定我们突然遇到断电或者系统崩溃。这样磁盘上的某些数据页面将得不到更新，因为它们被频繁使用以致在经过成百上千次更新之后仍然没有被写回磁盘。但是这些更新的结果都存在于内存中，这样看上去所有的更新都将丢失。我们将如何处理这个问题呢？如何将丢失的更新恢复过来，而不使用前面提到过的每次更新之后都进行写回的方式？

问题的答案是对每个更新操作系统给自己做一个记录，称为日志项，该记录保存在一个称为日志缓冲区的内存区域中。日志项包含相应更新操作的足够信息，使得系统能够知道如何来重做该更新操作或者在事务异常中止的情况下撤销该更新操作。在合适的时刻日志缓冲区将被写回到磁盘上一个称为日志文件的顺序文件中，该文件包括了在过去的某个时间段中所有的日志项。通过这种方式，在某一时刻一旦内存中的内容丢失，恢复进程就能够通过日志文件对磁盘的相应数据进行更新操作。这个日志方法比每次更新操作都写回的方式要好的其中一个原因是：系统只需要以一个不太频繁的间隔写回日志缓冲区就行了，而该操作可以和其他大量数据页面更新操作一起进行，从而节省了I/O开销。

我们将在稍后看到系统需要这些日志记录来完成恢复操作。即使所有磁盘页面的更新操

作都写回到磁盘（只要这些操作不怎么占系统资源）对于恢复操作而言也不是充分的。如果一个事务进行过程中发生系统崩溃，就会给数据库留下不一致的状态（比如由于补偿更新操作没有执行，从而使得钱多出来或者消失了），因为我们丢失了内存中的内容，我们同样也丢失了当时程序逻辑所处的位置，而这一信息是我们在执行补偿更新操作时所需要的。我们不仅要恢复对磁盘上每一页的更新，同样要恢复崩溃之前所有提交的事务的一致状态，这一点很重要。为了充分实现这一目标，日志文件是不可缺少的。

以上对数据库恢复的简单介绍覆盖了本章前面提到的事务ACID性质中的持久性的大部分内容。（另外一个关于处理存储在磁盘这样的稳定介质上的数据的丢失问题将在10.9节做简单介绍。）我们现在可以进行关于日志文件的讨论了，通常数据库用它来处理内存数据丢失问题，我们将看到系统是如何在系统崩溃之后利用这个文件进行恢复操作的。

10.7 恢复细节：日志格式

考虑下面调度器将看到的经历H5的操作：

[10.7.1] H5 = R₁(A, 50) W₁(A, 20) R₂(C, 100) W₂(C, 50) C2 R₁(B, 50) W₁(B, 80) C₁ ...

由于我们前面介绍过的LRU缓冲模式的原因，我们知道数据项A、B和C的更新后的值不一定会按它们在经历中发生的顺序写回到磁盘上去。如果在将来某个时刻发生系统崩溃，我们也许会在磁盘上看到这样的值：A = 50（由于A在一个常用的数据页面上，因此更新W₁(A, 20)没有被写回到磁盘上），C = 100（数据C的更新还没有写回到磁盘上），B = 80（B的更新已经写回到磁盘上了）。我们必须很清楚地知道LRU缓冲模式是以减少磁盘I/O次数为目标来进行的，因此我们不能期望磁盘能和LRU缓冲保持一致的结果。事务T₂从银行账户中提出了50美元，虽然事务T₂提交了但这一结果却没有被写回到磁盘上，假定用户已经将钱拿走了。同时客户从账户A向账户B转账30美元的交易中却意外地获得了30美元。（银行是不会喜欢这种行为的。）

然而，就像我们在前面提到过的那样，即使系统在每个更新操作后都将结果写回到磁盘上也不能解决问题。（即使系统在事务提交时执行所有写回操作也不能解决问题。）假定我们实行这样的策略：一旦缓冲区的数据页面被更新，该页就立刻写回到磁盘上去；假定经历H5中C₂之后系统发生崩溃。这样事务T₁的第二个更新就不能写回到磁盘上去了。当系统崩溃重新启动之后，我们会在磁盘上发现这样的值：A = 20, C = 50, B = 50。A和B的总和应该是100，因为T₁并不希望增加或减少这两个账户的余额总和。在事务提交时执行所有的写回操作仍然会发生同样的问题：当系统在写回两个数据页面的中间崩溃时导致的问题就和上面的例子一样了，因此这两个方法都不能解决问题。

问题的难点在于我们要保证在本章开始时提到的事务性质中的两条性质：原子性和持久性。回想一下原子性的定义，它是组成事务的一系列数据项更新是一个不可分的，这些操作要么全做要么全不做。这一性质即使在系统崩溃后仍将得到保证，这就涉及到事务的持久性。事务的持久性要求一旦事务提交之后，事务对数据库所做的更新就应该保存下来，即使系统崩溃之后也能将该事务所做的更新恢复过来。我们前面已经提到过，当系统使用日志文件时就能保证这一性质。一个称为数据库恢复的过程将在系统崩溃后执行，它使用崩溃之前记录的日志项将数据库所在磁盘恢复到系统在崩溃这一刻应该有的一致状态，保证一个事务的操作要么全做要么全不做。

必须注意我们在前一节中提到的关于数据库恢复的一个重要假设是：在系统崩溃重启之后，事务系统永远都不可能记得在内存数据丢失时正在运行的事务逻辑的意图。这样的阐述显得太过晦涩而难于掌握（尽管当前正在努力改变这一点）。当前，所有的数据库恢复过程都依赖于稳定存储器上的信息将数据库恢复到一个一致状态。那么系统崩溃时一致状态是怎么样的呢？比如经历H5中C₂之后发生崩溃。这个状态应该是这样的：因为事务T₁执行了提交操作，我们应该能够恢复该事务所产生的最终结果；而事务T₁没有能够执行其中的一个更新操作，当然也没有提交，因此它在前面所做的任何更新产生的影响都应该消除。因此我们得到的结果应该是：A = 50 , C = 50, B = 50。这是一个一致状态。

我们现在来重写经历H5，如图10-13所示。在一些操作旁边列出了存放在内存缓冲区中的恢复时使用的该操作对应的日志项。（我们强调这些日志项是放在内存中的，将日志缓冲区的内容写回到磁盘上的日志文件这一操作在图10-13中作为一个特殊的事件。）要提醒读者的是，在此处所陈述的细节并不是基于某个特定的商业数据库系统的。我们使用了简化的易于说明恢复机制的方式，从而能方便地掌握大多数数据库中恢复过程的实质精神。相比而言，商业数据库中的实际的数据库恢复体系结构要复杂得多。

操作	日志项	说 明
R ₁ (A, 50)	(S, 1)	事务T ₁ 的日志项开始，读操作无须做日志，但该操作标志着事务的开始
W ₁ (A, 20)	(W, 1, A, 50, 20)	T ₁ 为A.balance 的更新写日志。50是更新之前的值，20是更新之后的值
R ₂ (C, 100)	(S, 2)	事务T ₂ 的日志项开始
W ₂ (C, 50)	(W, 2, C, 100, 50)	写的日志项
C ₂	(C, 2)	提交事务T ₂ 日志项(将日志缓冲区写回日志文件)
R ₁ (B, 50)	没有日志项	
W ₁ (B, 80)	(W, 1, B, 50, 80)	
C ₁	(C, 1)	事务T ₁ 提交(将日志缓冲区写回日志文件)

图10-13 经历H5的操作以及相关的日志项

我们来看一下图10-13中的第二个操作W₁(A,20)，这是事务T₁执行的一个更新操作，将表accounts中acct_id = A的行的余额列设为20。同样，图10-13中写入的日志项(W,1,A,50,20)，其中50是数据项A在更新之前的值，20是数据项A在更新之后的值。在更为复杂的日志项中，也许还会用到唯一的行标识（比如ROWID），还有多列更新前和更新后的值。在此图中出现的所有日志项的类型都是针对单行的。对于插入和删除操作而言同样也存在日志项，为了简化我们的讨论我们在此将忽略它们。在图10-13中我们可以看到两个事件将日志缓冲区写回到磁盘上的日志文件中。在我们的简单模式中，只有在两种情况下我们才会将日志缓冲写回到日志文件中：（1）当一些事务提交时；（2）当日志缓冲区已满而没有多余的空间容纳更多的日志项时。（因此为了不影响其他的工作，我们希望系统有“磁盘写的双缓冲机制”，这意味着系统可以向第二个日志缓冲区写日志项，而同时将第一个已经满了的日志缓冲区写回到磁盘上去。）

现在我们将证明日志项中已经包含了足够的数据，可以让系统在崩溃之后恢复到一个一致状态。我们可以看到我们会碰到两种不同的问题。首先，我们已经将一些没有完成的事务所做的更新写回到磁盘上去了。我们将撤销这些更新对磁盘造成的影响（我们称之为UNDO），

由于这个原因我们在日志项中记录了更新前的值。其次，我们会发现一些已经提交了的事务没有将它们更新过的数据写回到磁盘上。我们将重新执行这些操作，将新的值写回到磁盘上（我们称之为REDO操作），出于这个原因我们在日志中记录了更新后的值。

现在假定在图10-13的操作序列中 $W_1(B, 80)$ 完成之后系统崩溃。这意味着日志项 $(W, 1, B, 50, 80)$ 已经写到日志缓冲区中，但是写到磁盘上的最后的日志项是 $(C, 2)$ ，这是我们在恢复过程中能够找到的最后的日志项。这时，事务 T_2 已经提交而 T_1 还没有提交，我们必须确信事务 T_2 所做的更新全部存到了磁盘上而事务 T_1 在磁盘上做的更新都被撤销。如同在图10-13前面所解释的那样，恢复之后最后得到的结果应该是： $A = 50, C = 50, B = 50$ 。

在系统崩溃重启之后系统操作员给出启动恢复的命令。（这通常是一条RESTART命令，我们通常称恢复过程为RESTART。）恢复过程可以分为ROLLBACK和ROLL FORWARD两个阶段。在ROLLBACK阶段，日志文件中的日志项从后向前扫描，直到日志文件头（所有数据存取操作的开始位置）。（我们假定此处系统是从经历H5的 $R_1(A, 50)$ 操作开始的）。在ROLL FORWARD阶段，日志文件是从前向后扫描的。在ROLLBACK阶段，恢复操作执行UNDO操作撤销那些不应该发生的更新，因为执行这些更新的事务没有提交，与此同时生成一张提交事务的列表。在ROLL FORWARD阶段，恢复程序执行REDO操作完成那些应该发生的更新，因为这些更新是由提交事务发出的。为了便于说明我们只是给出了一个简单的恢复过程的体系结构，其中我们假定ROLLBACK操作先做，然后进行ROLL FORWARD操作。这是IBM的System R（DB2的前身）系统使用的顺序，但是你必须知道的是许多现代的数据库系统执行的是相反的顺序。但这对我们说明性的介绍并不是太重要。

图10-14和图10-15列出了在恢复的两个阶段中遇到的所有日志项以及恢复过程所采取的相应行动。注意每一行的左边列的数字是恢复过程中的步骤。

日志项	执行ROLLBACK操作
1. $(C, 2)$	事务 T_2 放入提交列表
2. $(W, 2, C, 100, 50)$	因为 T_2 在提交列表中，所以不做任何事
3. $(S, 2)$	标记 T_2 不再是活动的了
4. $(W, 1, A, 50, 20)$	T_1 没有提交，因此系统对该数据项执行UNDO操作，将A的值设成修改之前的50。将 T_1 放入未提交事务列表
5. $(S, 1)$	标记 T_1 不再是活动的了

图10-14 经历H5的ROLLBACK过程（系统在 $W_1(B, 80)$ 后崩溃）

日志项	执行ROLLFORWARD操作
6. $(S, 1)$	无须任何操作
7. $(W, 1, A, 50, 20)$	T_1 是未提交事务，所以不做任何事
8. $(S, 2)$	无须任何操作
9. $(W, 2, C, 100, 50)$	T_2 在提交列表中，因此系统对该数据项执行UNDO操作，将C的值设成修改之后的50
10. $(C, 2)$	无须任何操作
11.	标记整个日志文件的ROLL FORWARD操作完成 恢复过程结束

图10-15 ROLL FORWARD过程（在图10-14的ROLLBACK操作之后）

我们需要对图10-14和图10-15中所采取的各个步骤进行一下说明。在ROLLBACK阶段系统从后向前扫描日志文件产生提交事务和未提交事务的列表。实现这一点是比较简单的，因为记录在日志文件中的任何事务的最后一个操作要么是提交操作要么是其他操作（说明该事务还没有提交），系统逆向扫描日志文件时首先遇到每个事务最后发出的操作。提交事务列表将在恢复过程的下一个阶段ROLL FORWARD中使用，而未提交事务列表将在这个阶段用来确定何时进行UNDO操作。因为当逆向扫描日志时（总是先遇到每个事务最后发出的操作），系统很容易判断一个事务是否是未提交事务。对未提交事务的写操作日志项可以立即进行UNDO操作。当然在恢复过程中UNDO或者REDO操作需要相应的磁盘缓冲区。我们可以在图10-14中的步骤4中看到对一个写操作执行UNDO操作的例子。由于发出写操作的事务没有提交，因此该写操作不应该反映到磁盘上。可能这些写操作的结果当时的确没有写到磁盘上，但是可以肯定的是将操作对象的值恢复到写操作之前的肯定是无害的——因为我们最终将该数据项恢复到了未提交事务试图对它进行更新之前的状态。在ROLLBACK过程中对记录事务开始的日志项的关注并不是必要的，因为我们能够保证当我们到达日志文件的开始处时（系统启动的时刻），没有任何事务处于活动状态。在更为普遍的情况下，ROLLBACK要求知道哪些事务仍然处于活动事务列表中以决定何时结束。

在图10-5的ROLL FORWARD示例中，系统简单地使用了在ROLLBACK过程收集的提交事务列表，对这些提交事务执行REDO操作。（参见步骤9的REDO操作的例子。）

在恢复操作结束之后，我们在磁盘上应该看到的是正确的值。所有提交事务发出的更新都反映到了磁盘上，而所有未提交事务发出的更新都被回滚。的确是这样，我们可以注意到在ROLLBACK的步骤4中我们将50写入数据A中，在ROLL FORWARD的步骤9中我们将50写入数据B中。我们应该记得崩溃是在经历H5的W_i(B,80)之后发生的。因为该操作的日志项没有被记录到磁盘上（在图10-13中我们可以看到这一点），因此在恢复过程中我们无法将找到B原来的值。但我们需要依赖这样一个事实，即对B的更新并没有真正写到磁盘上。基本上我们可以肯定这样一个事实，如果一个未提交事务的发出的写操作的值写到了磁盘上，那么记录该操作的日志项也必然写到了磁盘上。我们将在下一节中来讨论这个问题。假定我们可以相信这样一个事实，即B的新值80并没有写到磁盘上，那么经历H5中涉及的三个值：A=50，B=50（初始值），C=50（恢复过程将写入的正确值）。

保证需要的日志项都在磁盘上

我们如何能够保证正确的恢复过程所需要的日志项都在磁盘上呢？首先，很重要的一点是操作系统能够保证写到磁盘的操作能够正确地完成，这是和事务提供的ACID性质是对称的。写回到磁盘的操作是原子操作，这意味着即使是在写磁盘的过程中系统崩溃，那么当下次读到写过的这块数据时，也能立即判断出写操作是否成功执行了；如果成功执行了，那么可以认为该数据块是符合数据库的正确逻辑的。当执行写磁盘的操作时，I/O子系统通常会通过写完后再读一次来测试结果磁盘映像，如果有错则再次执行写操作。如果在经过多次重试之后仍然不能成功写入，则操作系统会得到一个严重的错误警告。因此如果发生写错误，那么写操作成功之后的正常的逻辑就不能继续。比如，当事务提交触发写回日志缓冲区时，如果发生写错误那么提交操作将失败。如果在写读操作过程中系统崩溃，那么将产生错误的块，RESTART之后对该数据块的读操作将检测到这种错误的存在。因此我们能够保证没有

不可检测的错误存在；敏感的“错误检测编码”数据存放方式将支持这一保证。“先写后读”的磁盘写操作协议内置于磁盘头中。所有的这些错误检测在I/O子系统的一个非常底层的级别上进行，甚至对操作系统是透明的。

没有不可检测的磁盘写错误在系统崩溃时也是可保证的。当系统得到写操作成功完成的通知时，就保证了将来对该数据块的读操作会取得成功（除非整个磁盘丢失，像10.9节中说明的那样）。因此当数据库系统将日志缓冲区写回到磁盘上的日志文件后（出于严格的恢复考虑，有时这样的写回操作是同时对系统中的两个磁盘进行的，这样可以获得更高的可靠性），我们可以安全地认为触发这次写回操作的事务提交获得了成功。

现在假定我们可以相信日志文件写回操作，那么我们仍然需要确保恢复的逻辑是正确的，也就是说我们始终能从日志文件中找到执行正确的恢复操作所需要的日志项。已经知道我们可以将恢复工作分成两种类型：UNDO和REDO。先来考虑对提交事务发出的更新操作执行的REDO操作。我们知道事务的提交操作是触发日志缓冲区写回磁盘的原因，在将日志缓冲区写回到磁盘之前该事务不能被认为是成功完成的。（比如，该事务是一个银行取款的任务，在日志缓冲区记录到磁盘之前我们不能将钱支出去。）因为提交操作的日志项必须写回到磁盘上去，因此在这之前的该事务的写操作的日志项都被写回到了磁盘上（我们必须注意到日志项在日志缓冲区中是按序放置的，因此关于提交操作的日志项必然在该事务所有其他日志项的后面），所以我们能够保证REDO任务能够正确地执行。

恢复过程的第二个任务是对日志文件中所有的未提交事务更新执行UNDO操作。我们未得到任何保证，未提交事务已经发出的更新操作的日志项都写回到了磁盘上。比如，图10-13到图10-15中的恢复例子中日志缓冲区的最后写操作日志项是(W,1,B,50,80)，但崩溃发生时该日志项并没有写回到磁盘上（我们没有执行写回日志缓冲区的操作，直到(C,1)记入日志缓冲区）。我们应该自问这样的结果是否会导致什么问题。对于这些没有写回到磁盘上的日志项，相关的UNDO操作是否就可以不做了呢？很明显仅当更新过的数据页面由于LRU的缓冲机制被写回到磁盘上时，才有可能使更新过的数据在日志缓冲区写回之前写到磁盘上。（如果该数据页面没有被替换出去，那么UNDO操作在恢复过程中将是不必要的，就像我们在前面看到的关于数据项B的情况一样。）看上去数据页面似乎会在相应的日志项之前写回到磁盘上去，比如，在一个写操作之后进行了大量的读入新的数据页面的操作，其间没有任何提交操作来触发写回日志缓冲区。这些读操作最终将迫使缓冲机制重新使用已被更新的行所在的数据页面。

这是一个非常合理的例子，因此我们必须提供一些特殊的措施来保证这种现象不会发生。其中一种可能的措施是在一定程度上修改LRU页面替换策略，使得更新过的数据也不会被移到磁盘直到相应的事务提交。这样的缓冲策略事实上已经被提出过了，所有被某个事务更新过的数据页面都将保留在缓冲区中直到该事务提交为止。在这种策略下，事实上在恢复过程中根本不需要UNDO操作，因此也无需在日志文件中记录它们原来的值。然而当一个事务对大量的数据页面进行更新时，我们就不能保证保留所有这些缓冲区直到该事务提交为止（我们可能会耗费所有的内存）。一个更为复杂的策略是允许事务在提交之前申明它对某个缓冲区页面的使用已经完毕，无需在内存中保留该缓冲区页面。比如，当对某个游标的某个数据页面执行了FETCH指令后，我们可以说该事务已经完成了对该数据页面的使用，该数据页面已经无须保留在内存中了。（当然，在Serializable隔离级别的事务必须保持它在每个读写过的

行上的锁，无论该行所在的数据页面是否在缓存中。)

因此我们又回到了前面提到过的情况：一个更新过的数据页面由于LRU替换策略有可能在相应的日志缓冲区写回磁盘之前写回到磁盘上，这样在恢复过程的UNDO操作中就会出现问题。为了防止这种情况，系统必须保证相关的日志必须在相应的脏（修改过的）数据页面之前写回到磁盘上去。通常称之为先写日志原则(WAL)。实现这一原则，数据库系统通常创建称为日志序列号(LSN)的一个日志项属性，该序列号随着写到日志缓冲区中日志项的增加而增长。我们跟踪最近一次日志缓冲区被写回到磁盘后被建立的日志缓冲区中的最小序列号，记为LSN_BUFFMIN。（这对数据库系统而言是一个全局值，如果我们假定我们只有一个系统缓冲区。）此外对于每个数据缓冲页面，我们记录该页最近一次更新操作的日志序列号，记为LSN_PGMAX。最后我们定义这样一个原则，即一个磁盘页面不能被LRU缓冲区写到磁盘，除非该缓冲页面的LSN_PGMAX比全局的LSN_BUFFMIN小。这就保证了在有关的日志项写回到磁盘上之前，相关的数据缓冲页面不会被写回到磁盘上去，从而解决了UNDO过程中可能遇到的问题。根据上面的陈述我们可以有下面的定理，在此我们不提供证明。

定理10.7.1 简单恢复 给定这些日志文件写到磁盘的保证，恢复过程能够为任何事务经历在任何时刻的崩溃提供有效的恢复。崩溃之前提交的事务所做的所有更新将切实地反映到磁盘上，而未提交事务所做的那部分更新操作将被回滚。 ■

10.8 检查点

在10.7节所提供的恢复例子中，我们执行ROLLBACK过程直到系统启动的那一刻。这是一个明显的选择，因为那时数据库处于完全一致的状态，系统缓冲区中不存在脏的缓冲页面，事实上所有的事务数据更新操作还没有开始。我们可以将ROLLBACK操作认为是将数据库恢复到这样一个质朴状态的过程（只有提交事务发出的更新操作将记录到磁盘上），ROLL FORWARD操作的任务是从那一刻起重做在崩溃之前提交事务发出的所有更新操作。

但是恢复过程所需要的时间和ROLLBACK和ROLL FORWARD过程中要读取的日志文件的长度成正比。在每一阶段我们都必须从磁盘上读取目标行进行更新，最坏情况下这样的过程需要的时间也许和当时应用程序执行的时间一样长。如果我们假定大多数事务的周期都是相对较短的（几秒钟），那么ROLLBACK阶段将变得快捷而有效。因为它只要对崩溃时仍然处于活动状态的事务已经发出的更新操作进行UNDO操作即可。然而ROLL FORWARD操作却和日志文件的长度有很大的关系，因为该过程必须REDO所有提交事务发出的更新操作（假定大多数事务已经提交）。因此执行ROLL FORWARD操作的时间甚至会比原始应用执行这些更新操作所花费的时间还要多。我们希望做的是记录系统启动之后经过一段合理的时间之后的一个一致状态，这样恢复过程就不需要从日志文件最初的位置开始执行ROLL FORWARD操作，而只需要从记录的这一点（我们通常称之为检查点）开始执行该操作即可。当然，我们希望在这种方式下恢复操作能达到预期的目的，即能够忽略所有在检查点之前发生的事务和数据更新操作。

有三种创建数据库的恢复检查点的不同方法。根据三种方法的复杂性排序，它们分别是：提交一致性检查点，高速缓存一致性检查点，模糊一致性检查点。使用较复杂的检查点方法，系统能够维护一个平稳的事务输出，因此作为用户而言当然希望有更为复杂的检查点策略，如果系统设计者能够实现这些策略的话。没有任何系统需要同时支持两种不同的方法。我们

先来讲述在普通事务系统中采用的较为简单的检查点策略——提交一致性检查点。

1. 提交一致性检查点

首先，系统决定开始记录一个检查点（也许由于上次作检查点后日志记录项的数量已经超出了一定的限制）时，系统进入“执行检查点状态”。

定义10.8.1 提交一致性检查点的过程步骤 在进入执行检查点状态之后，我们会有以下一些规则：

- 1) 新的事务不能开始直到检查点完成。
- 2) 现有的事务继续执行直到提交并且相关的日志都写到磁盘上。
- 3) 当前的日志缓冲区都写回到日志文件中，这样系统就能保证缓冲中所有的脏数据页面都写回到了磁盘上。
- 4) 步骤1~3完成之后，系统写入一个特殊的日志文件项(CKPT)到磁盘上，检查点的操作完成。 ■

定义10.8.1中的步骤1~3同样定义了我们在关闭系统时所要做的精确步骤，这样我们在下次启动时就不需要进行恢复了。最终的状态等价于系统启动时日志文件中还没有日志项的状态。结果很显然我们可以修改前一节中提到的恢复过程，这样回滚操作只需执行到遇到该形式的日志项(CKPT)即可，而不是整个日志文件。我们可以保证没有一个事务在检查点这一刻仍处于运行状态，前面所有的事务都已经提交并且它们的更新都已经写到磁盘上，这样这些操作的日志项在日志文件中就显得不再那么重要。通过执行检查点，我们可以大大减少进行恢复操作所需要的时间。

2. 其他检查点类型的动机

像我们提到的那样，定义10.8.1中定义的检查点的过程和系统关闭时所进行的操作顺序是相同的。没有新的事务可以开始直到系统中所有的活动事务结束并且所有的脏数据页面都已经写回到磁盘上。这一过程也许需要较长的时间。在此，我们认为事务的周期是非常短的，它们也许在几秒内就能结束。但是这一假设并非在所有的计算机上都是成立的。在大多数数据库系统中事务很有可能会持续几分钟甚至几小时，在这过程中都会有读和写的操作。然而这样长的操作在提交一致性检查点时几乎是不可能的，除非我们愿意在这段时间内关闭系统。（就像我们在步骤1中提到的那样，检查点过程中处理新的事务不能开始。）对于写回缓冲区的操作，可以考虑如果我们有200MB的内存空间用作磁盘缓冲区（每页4KB），其中一半的数据页面是脏的，因此必须被写回到磁盘上。于是我们将有 $100M/4K=25\ 000$ 个页面需要写回磁盘。因为当使用单个磁盘上，每写回一个页面大约需要 $1/40$ 秒，总共的时间将是 $25\ 000/40=625$ 秒。随着内存价格的下降，这样的情况将变得很普通。

为了避免这类问题在检查点过程阻塞系统操作，需要采用更为复杂的检查点策略。我们在定义10.8.1中提到的检查点类型称为提交一致性检查点，因为所有的事物都必须提交，并且脏数据页面都写回到磁盘上。下一个级别的检查点策略称为高速缓存一致性检查点。使用该策略，允许事务的活动周期跨检查点，我们只需要保证缓冲区中所有的脏数据页面都写回到磁盘上（包括相关的日志项）。内存中的磁盘缓冲区通常可以称为磁盘高速缓存，这也解释了这一检查点策略的名称。然而在做快存一致性检查点的过程中，仍然处于活动状态的事务必须处于等待状态；特别我们不能发出新的I/O操作。但是，这仍然要比提交一致性检查点的要求要宽松一些，后者在有较长事务存在的情况下会耗费大量的时间。我们在下一步将列出做

快存一致性的检查点所需要的步骤。此处我们简单的提一下，仍然存在更为复杂的检查点策略，即模糊检查点，这类检查点策略允许在脏数据页不写回磁盘的情况下作检查点。（这类检查点被认为是不完全的，除非所有的缓冲页写回到磁盘；但是它允许有些重要的工作在这过程中继续进行。）

3. 高速缓存一致性检查点

以下是系统在做高速缓存一致性检查点时需要的过程。

定义10.8.2 高速缓存一致性检查点的过程步骤

- 1) 新的事务不能开始。
- 2) 存在的事务不允许发出新的操作。
- 3) 当前的日志缓冲区写回到磁盘上，之后系统保证高速缓存中的所有脏数据页面都写回到磁盘上。
- 4) 最后将一个特殊的日志项(CKPT, List)写入磁盘，到此我们完成了检查点。List中保存的是检查点过程中仍然处于活动状态的事务的列表。 ■

使用高速缓存一致性检查点的恢复过程和使用提交一致性检查点的恢复过程在方法上有很大的不同。

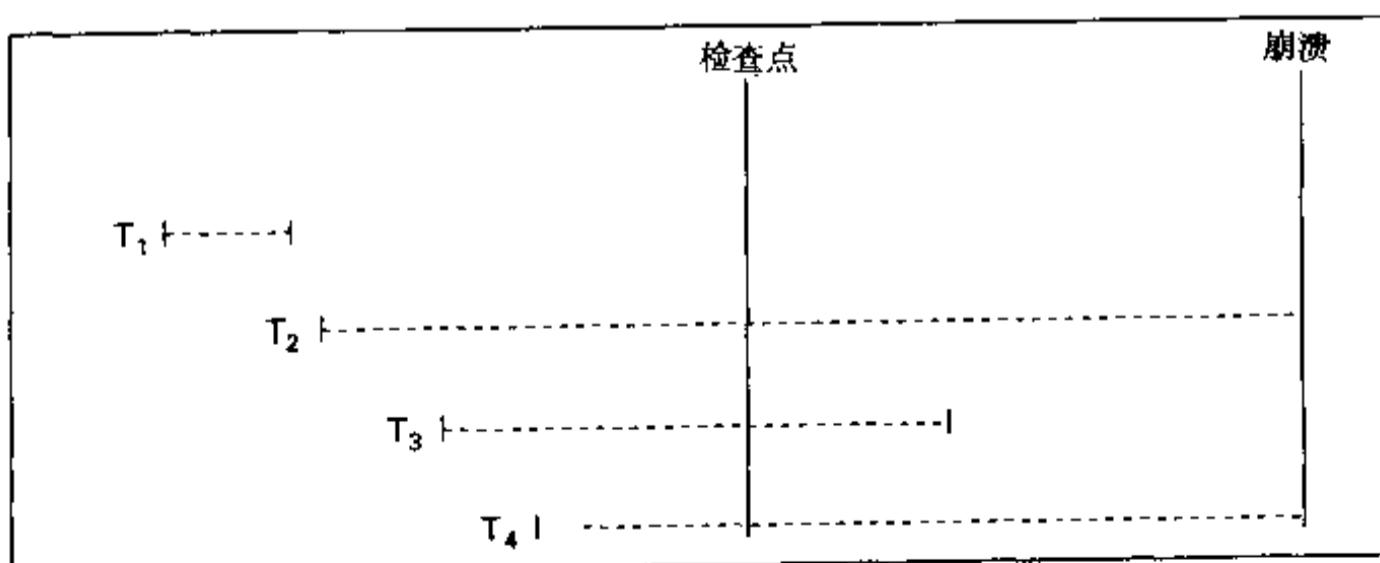
例10.8.1 使用高速缓存一致性检查点的日志文件和恢复 这里是一个新的经历用来说明高速缓存一致性检查点的特性。我们会列出日志缓冲区中的日志项以及该经历中崩溃之后进行的恢复过程。首先该经历如下：

```
R1(A, 10) W1(A, 1) C1 R2(A, 1) R3(B, 2) W2(A, 3) R4(C, 5) CKPT W3(B, 4) C3
R4(B, 4) W4(C, 6) C4 CRASH
```

这是该经历的日志项事件序列。操作C₄在崩溃之前没有完成，因此写回到磁盘上的最后一个日志项是(C,3)。

```
(S, 1) (W, 1, A, 10, 1) (C, 1) (S, 2) (S, 3) (W, 2, A, 1, 3) (S, 4)(CKPT,
(LIST = T2,T3,T4)) (W, 3, B, 2, 4) (C, 3) (W, 4, C, 5, 6)
```

在我们做高速缓存一致性检查点时，我们将以下一些值写回到磁盘上去：A=3,B=2,C=5。（包含数据A的脏数据页面在作检查点时写回到磁盘上。）我们假定没有其他的更新操作在崩溃之前写回到磁盘上，因此磁盘上数据项的值保持不变。在下面的图中说明了各种事件的发生过程。事务T_k以日志项(S,k)开始，以(C,k)结束。



现在我们列出在恢复过程中采取的动作，以ROLLBACK操作开始：

ROLLBACK

- | | |
|------------------------------------|--------------------------------|
| 1.(C,3) | 注意事务 T_3 是活动事务列表中已提交事务。 |
| 2. (W,3,B,2,4) | 已提交事务，等待ROLL FORWARD。 |
| 3. (CKPT,(LIST= T_2, T_3, T_4)) | 注意事务 T_2, T_4 是活动事务列表中已提交事务。 |
| 4.(S,4) | 活动事务列表现在变为 $\{T_2\}$ 。 |
| 5. (W,2,A,1,3) | 未提交事务。UNDO: A=1。 |
| 6.(S,3) | 已提交事务。 |
| 7.(S,2) | 活动事务列表为空，结束ROLLBACK。 |

我们可以看到在使用高速缓存一致性检查点的情况下，当ROLLBACK碰到CKPT日志项时，系统记下活动事务列表，尽管我们还没有看到日志文件中的任何操作。根据活动事务列表，删除已经提交的事务，则列表中剩下的事务发出的操作需要进行UNDO。我们继续ROLLBACK阶段，直到完成所有的UNDO操作。我们可以确定何时结束UNDO，因为当碰到(S,k)日志项时，就将该事务从未提交事务列表中去除。当所有这样的 T_k 被去除之后，ROLLBACK就可以结束了，尽管日志文件中仍然还有别的日志项。

ROLL FORWARD

- | | |
|------------------------------------|-------------------------------|
| 8. (CKPT,(LIST= T_2, T_3, T_4)) | 忽略在这之前的日志项。已提交事务= $\{T_3\}$ 。 |
| 9. (W,3,B,2,4) | ROLL FORWARD: B=4。 |
| 10.(C,3) | 没动作。最后一个日志项，于是ROLL FORWARD结束。 |

在ROLL FORWARD的开始阶段，我们只需REDO已提交事务发出的那些可能没有写回到磁盘上的更新操作即可。我们可以跳到检查点之后的第一个操作，因为我们知道检查点之前的所有更新操作都已经写回到磁盘上去了ROLL FORWARD继续到缓冲区文件的末尾。在系统崩溃时磁盘上的值为：A=3,B=2,C=5。恢复过程结束之后，我们有A=1（步骤5）和B=4（步骤9），C的值仍然是5。根据前面的时序图（事实上(C,4)日志项从来没有被写过），我们可以看到我们希望事务 T_1 和事务 T_2 以及 T_4 的更新操作写回到磁盘上。没有一个 T_3 完成的更新操作写回到磁盘上。因此我们得到恢复需要的结果是：A=1,B=4,C=5。 ■

4. 模糊检查点

模糊检查点的目标是减少完成检查点的绝对时间。高速缓存一致性检查点只需要将所有的脏的缓冲页面写回到磁盘上即可，相对于提交一致性检查点而言是一个提高，后者要求所有的活动事务提交。然而写回40MB的缓冲数据对于一个每秒钟只能进行80次写操作的磁盘而言仍然需要花费几分钟的时间。习惯于事务操作在几秒之内做出反应的用户对于这样的等待将是非常敏感的，在这过程中没有任何事务的动作可以得到执行。

为了去除这一限制，模糊检查点恢复策略使用两个检查点事件，即最近在日志文件中记录的两个用CKPT标记的检查点。这个策略的方法是系统每次做检查点CKPT_n时，记录自从上次做检查点CKPT_{n-1}以来缓冲区中的脏的数据页面集合。这样做的用途时，这些脏的数据页面将在下次做检查点CKPT_{n+1}时写回到磁盘上。在两个检查点之间的这段时间中这些脏的数据页面有很好的机会由正常的缓冲区管理操作写回到磁盘上。当下一次做检查点的事件临时时，可以启用后台进程来写回频繁更新的脏的数据页而，这样在下次做检查点时我们可以做到以

下保证：在做检查点 $CKPT_{N-1}$ 时记录的所有脏的数据页面在做检查点 $CKPT_N$ 之后都已经写回到了磁盘上。这样可以使得我们在做检查点 $CKPT_N$ 时避免写回缓冲区的操作。我们只需要对前面讲到的高速缓存一致性检查点下使用的恢复过程稍加改动，从最后的倒数第二个检查点 $CKPT_{N-1}$ （如果 $CKPT_N$ 是最后一个检查点）处开始执行ROLL FORWARD操作。

定义10.8.3 模糊检查点的过程步骤

- 1) 在做检查点之前，前一个检查点遗留的脏的数据页面将写回到磁盘上（同时应该留出一定的I/O带宽支持并发事务的正常操作的需要；因为这种写回操作并不显得太迫切）。
- 2) 在做检查点的开始阶段，没有新的事务可以开始。存在的事务不能发出新的操作。
- 3) 当前的日志缓冲区写回到磁盘上，并添加一个日志项（ $CKPT_N$, List），这和高速缓存一致性检查点中所做的一样。
- 4) 标记从上次检查点 $CKPT_{N-1}$ 到现在这段时间里变脏的数据缓冲区页面。这一过程也许可以通过在缓冲区目录中做标记来实现。这些信息不需要保留到磁盘上，因为这些信息只是在下次做检查点时使用，而不是在恢复时使用。 ■

就像上面解释的那样，使用模糊检查点的恢复过程和使用高速缓存一致性检查点的恢复过程不同之处只在于ROLL FORWARD必须从日志文件的倒数第二个检查点后的第一个日志项开始，参见后面的习题。

10.9 介质恢复

到目前为止我们只是假设内存中的数据会由于掉电或者系统崩溃而丢失，但是写到磁盘上的数据是可靠的，即假定崩溃之后磁盘上的数据始终是可用的。但这并不总是如此，一个磁盘上的数据也有可能全部丢失，比如读写数据的磁头损坏就会导致这种问题，就像老式留声机的碟片被磁针损坏一样。但这样的事件发生时，就需要特殊的恢复措施，我们称之为介质恢复。我们应该将此考虑到我们的通用恢复策略中。在这一节中，我们将指出解决这类问题的方法。

对于不包含介质故障的系统崩溃而言，我们可以采取10.7和10.8节中的恢复策略，这些措施中我们总是假设磁盘上的数据是正确的。如果一个磁盘在事务操作过程中损坏，那么系统将崩溃，并提醒操作员磁盘已经损坏。这时必须进行介质恢复。简单的介质恢复过程是这样的：在系统启动之前，事务系统要用到的磁盘都将做成批的拷贝。这些拷贝来自冗余的磁盘或者廉价的磁带介质，我们称这些数据为在线磁盘的备份。当一个磁盘在系统崩溃中损坏之后，我们就用备份的磁盘来替换（这时需要新的备份以防新问题的出现），然后进行正常的恢复过程。然而在这类恢复过程中，我们需要将ROLLBACK执行到系统开始时刻为止，因为我们不能保证做检查点时写回磁盘（已损坏）的数据在备份磁盘上也得到了更新。接着我们从系统崩溃的那一刻起执行ROLL FORWARD操作。我们可以假设新的磁盘就是系统崩溃时的那个磁盘，只是该磁盘上的数据自从系统启动后一直都没有更新过。根据这样的理解，很明显正常的恢复过程可以帮助我们恢复在该备份磁盘上的所有更新操作。

稳定的存储介质

现在我们已经意识到了磁盘介质也有可能损坏，由此可能产生下面的问题：如果磁盘上的日志文件也损坏了怎么办？答案是我们需要预见到这一点，并通过同时往两个不同的设备（通常是磁带）上写日志文件解决这一问题。这种技术通常称为日志镜像。我们的目的是通过

保证两个镜像设备独立存放日志来达到更高的可靠性。我们努力确保某一偶然事件可能导致其中一个设备上数据的丢失，但是却不影响另一个设备的数据。比如，两个设备是用不同的供电系统（或者至少一个设备有备份电源），它们是分开存放的，因此物理上的震动不会同时影响两个设备。存储的数据在两个设备上是重复的（无论是否用来存放日志数据），这样的存储方案相对于磁盘上单拷贝的非易失性存储和内存这样的易失性存储而言称为稳定的存储。

10.10 性能：TPC-A基准测试

我们来回顾一下到目前为止和第10章讨论的内容相关的性能问题。10.2节的内容告诉我们高级并发操作可以通过将不同事务的操作重叠执行使得多个设备同时工作。因此我们能够通过事务的并发来提高CPU的利用率和事务的吞吐量。在10.5节我们阐述了为什么当有较多的事务进入等待状态时，较高的事务可串行化要求会降低系统的并发性并进一步损害系统的吞吐量。由此我们引出了SQL-99标准中的隔离级别的概念。在10.6节中我们开始讨论内存中的磁盘缓冲区页面引发的数据持久性问题；当然性能问题是作为磁盘缓冲从而降低I/O开销和节省磁盘臂资源需求的最终动机。

然而，与此同时我们避开了对许多复杂的实现机制的讨论，而这些问题对事务的性能有着重要的影响。事务系统为了保证事务的ACID性质需要大量的CPU开销和I/O资源，就这一点而言拿事务系统和简单的磁盘访问文件系统相比是毫无意义的，因为后者不提供这些保证。为了对事务运行的开销有一个感性的认识，考虑下面一个事务执行的例子。

假定一个事务以一个SQL调用开始，通过索引访问一行。一串I/O请求将被提交给事务管理器（参见图10-8）。事务管理器给该线程建立一个事务编号，然后将调用传递给调度器。调度器必须对索引查找的谓词进行加锁，包括检查该谓词是否和已有的锁冲突。对于任何的I/O调用都必须在进行实际的磁盘访问之前执行这一检查过程。在磁盘页面和存取的行上同样需要使用短期锁和长期锁。如果请求的锁发生冲突，有时会进行一些测试确定是否发生了死锁。如果没有死锁发生，调度器将把调用线程设置为等待状态，系统将通过进程交换将别的进程调入执行。当检测到死锁发生时，必须选出牺牲事务异常中止执行，并将牺牲事务所做的所有更新操作回滚。当然在正常的事务执行过程中每次更新操作都会记录日志，在日志项中记录了该操作的目标数据更新前和更新后的值，日志项存放在日志缓冲区中。当事务提交的时候（我们考虑简单的系统），日志缓冲区的内容将被写回到磁盘上的日志文件中（两份拷贝存放在两个稳定的具有独立失效模型的设备上）。

我们可以看到为了保证事务的ACID性质，为了完成这项工作需要做大量的设计选择。历史上事务系统中的许多性能瓶颈是由最初的设计特性造成的。随着时间的推移，高级事务系统的实现在关于如何提高性能方面将变得更加复杂。人们开发了大量的事务基准来使用户对不同供应商提供的事务系统进行比较。这些基准测试中最著名的一个称为TPC-A基准测试。这一基准测试是由事务处理性能委员会（TPC）定义并标准化的，TPC是由工业界专门经营事务型数据库软件和硬件的供应商组成的一个委员会。标准制定后，没过几年大多数供应商不会将一个没有TPC-A（或者较简单的TPC-B）测试结果的事务型数据库产品推向市场，因为用户需要这方面的信息。现在TCP-A基准测试已经不再是供应商采用的一个正式的基准。尽管如此，在关于事务系统的性能瓶颈问题上TCP-A基准测试仍然能够给我们一些好的思路。大多数供应商内部仍然使用该基准测试来跟踪一些可能的性能问题。下面将给出该基准测试

的一个简单描述。

之所以将该基准测试定义得如此详尽的原因是能够保证使用该基准测试的所有供应商能够得到相同的测试，规则上不存在松散性使得只有某个供应商适应该规则，从而得到不公平的事务吞吐量。

1. TPC-A基准说明

TPC-A基准模仿银行业务的应用，要求定义四张表，如图10-16所示。（注意，用于测试的事务系统并非一定要是关系型的，并且我们也可以用包含同样记录数的文件来代替具有给定数量行的表。然而ACID性质是必须的，所以我们不能在一个简单的文件系统上进行这个基准测试。）

表明	行数	行大小	主键
Account	10,000,000	100 bytes	Account_ID
Teller	1000	100 bytes	Teller_ID
Branch	100	100 bytes	Branch_ID
History	Varies	50 bytes	(Account_ID, Time_Stamp)

图10-16 TPC-A基准测试需要的表（Rating ≤ 100TPS）

在图10-16的前三张表（Account, Teller, Branch）中定义了该表的行数，该行数是基于基准测试程序的事务吞吐量不超过100TPS的假设。这些表所需要的行数和TPS(比率)成线性关系。比如，在200TPS的事务吞吐率下，Account表需要两千万行，Teller表需要两千行，Branch表需要两百行。对于事务吞吐量较小的测试也可以使用较大的数据量，但是测试员可能希望通过减小表的大小看是否能够提高性能（因为这样I/O驻留在内存中的数据比例会更大）。

一般来说，在基准测试期间应尽可能使更多的事务并发，每个事务来自于模拟终端（模拟用户在一个简单的监视器前敲入命令）。这些事务都是一样的，向由某一特定的账户（Account.Account_ID = Aid）、出纳员（Tid）、支行（Bid）的值标识的大量的行的余额字段里面加钱或者取钱（此时Delta为负值）。在这过程中一条历史行写到相应的History文件中。图10-17提供了描述事务执行逻辑的伪代码。这些事务只有在Aid, Tid, Bid和Delta等参数上有一些不同。

```

(Read 100 bytes from the terminal, including Aid, Tid, Bid, and Delta)
(BEGIN TRANSACTION)
    Update Account where Account.Account_ID = Aid
        set Account.Balance = Account.Balance + Delta;
    Insert to History (50 bytes, include column values: Aid, Tid,
                      Bid, Delta, Time_Stamp)
    Update Teller where Teller.Teller_ID = Tid
        set Teller.Balance = Teller.Balance + Delta;
    Update Branch where Branch.Branch_ID = Bid
        set Branch.Balance = Branch.Balance + Delta
(COMMIT TRANSACTION)
Write 200 byte message to Terminal, including Aid, Tid,
      Bid, Delta, Account.Balance

```

图10-17 TPC-A基准测试的事务逻辑

我们可以看到每个提款事务对三行进行更新操作，三元组的记录形式防止错误的发生。基准测试过程中事务执行所需要的Aid、Tid、Bid三个参数的值是在运行前以这样一种方式随机产生的，这种方式能够保证Account表中的所有行被每个事务更新的机会是平等的，Teller表和Branch表也一样。注意，在运行期中没有测试能保证增加到Account表行的余额中的Delta并没有减少余额值到零以下（因为对于取款操作而言，Delta值是负的），但是这个决定是由TPC委员会做出的，对任何一家供应商都是一样的。

一个详尽的基准测试说明提供了大量的测试旨在确定系统是否提供了事务的ACID性质，即原子性、一致性、隔离性和持久性。这些测试不作为基准测试间隔的一部分。图10-18中给出了在TPC-A基准测试说明的段落2.4.2.1中列出的隔离性测试，这种测试使用了图10-17中的事物执行逻辑。然而，展示任意的事物组合都能满足SQL-99标准下的完全可串行化隔离级别是提供商的责任，而不仅仅是TPC-A所要做的。持久性测试包含导致运行系统中的内存失效的原因并表明恢复操作可以捕捉完成的事务提交的更新操作，将系统恢复到一个一致点稳定介质的失效恢复同样也可以测试。需要指出的一点是History表中的内容不能作为日志文件在恢复过程中使用，尽管History行看上去包含了执行恢复操作所需要的所有信息，但它是由应用过程控制，它们可能随时被删除。

事务1开始。(图10-17的逻辑) 在COMMIT之前事务1立即停止。
事务2开始 事务2试图更新和事务1相同的记录。 验证事务2处于等待状态。 允许事务1完成。这样事务2可以完成。 验证账户余额反映了两次更新操作。

图10-18 事务的隔离性测试（常规加锁模式）

我们在前面提到过我们模拟的系统在基准测试过程中从不同的终端接受大量的事务请求。为了说明这些终端发出请求的频率和次数，我们需要以下定义。

定义10.10.1 从单个终端发出的两个相继的事务请求间隔的时间称为周期，它由两部分组成：响应时间（系统的延迟）和思考时间（大致等于终端用户花在思考上的时间）。 ■

为了执行有效的TCP-A基准测试，两次终端请求之间的思考时间必须在基准测试程序运行之前随机生成，以便最后的平均周期时间最少为10秒。（这是一个略显复杂的问题，因为在执行基准测试之前我们不知道准确的系统响应时间，因此我们不知道需要多长的思考时间以便获得可能的10秒平均周期。我们只能通过一系列的近似来取得思考时间。）要求有10秒左右的周期的出发点是一个能够有效地服务1000个终端的事务系统至少要有100TPS的吞吐量。在基准测试过程中同样存在限制响应时间的条件。当只有一个用户时，大多被测试的系统会给出小于一秒的响应时间，但是随着越来越多的用户加入，CPU和I/O资源就会形成等待队列，从而导致响应变慢。测试TPS的正规方法基本上是这样的：我们以给定数量的终端开始，不断增加终端的数量，每10秒执行一个事务，直到10%的事务的响应时间超过两秒。这样做能够保证进行TPC-A基准测试的不同供应商的产品获得大致相同的资源利用率。

定义10.10.2 TPC-A基准测试的响应时间标准 在基准测试过程中，90%的事务的响应时间必须不超过2秒。 ■

为了运行测试程序，需要一定的时间使得并发事务流趋于稳定，直到系统进入稳定状态（具有持续的TPS）。这个时间段不能少于15分钟，但不能超过1小时，在这过程中必须作检查点。我们假定在基准测试过程中提供了足够的磁盘数量，这样在响应时间标准下期望的资源队列下CPU能够保持90%的利用率。使用多少磁盘取决于测试工程师，但是要尽可能地节约资源。因为除了TPS这样一个衡量指标之外，还有一个重要指标就是每个TPS的5年系统开销（\$COST/TPS）。

对于Set Query标准（早期的TPC-A基准中用来测量\$COST/QPM指标）而言，开销（\$COST）计算包括硬件开销（包括终端和网络）、加上5年的软件和维护费用。（这只是一个简单的相加值，不是净值的计算。）为了给大家一个事务处理速度的概念，在图10-19中给出了随机选择的几个商业系统的指标。对于一个事务系统而言每秒几百个事务的TPC-A TPS指标是完全可能的。一般来说，随着TPS的增加，\$COST/TPS也会增加。图10-19中的UNIFY 2000系统比较特别。

数据库系统	硬件	TPS	\$COST/TPS	日期
ORACLE7	VAXcluster 4x 6000-560	425.70	\$16 326	5/12/92
UNIFY 2000	Pyramid MIServer 12S/12	468.45	\$5971	3/4/92
INFORMIX 4.10	IBM RISC 6000/970	110.32	\$2789	7/16/92
SYBASE 4.8.1.1	Symmetry 2000/250	173.11	\$2770	3/30/92

图10-19 一些商业系统的历史TPC-A指标

2. TPC-A 基准测试的经验教训

如果事务系统使用初级的设计特性，那么在很多方面可能造成性能的瓶颈。如果系统中确实存在这些瓶颈，那么TPC-A基准测试的确能够很好地暴露这些问题导致的性能问题。起初，我们认为TPC-A事务可以用简单的交错CPU任务实现，理论上只需要几千条CPU指令和若干I/O请求以及在提交前一些相对较少的更新操作即可。然而并非如此，简单事务逻辑在三个方面造成瓶颈：(1) 数据项加锁，(2) 写到日志文件，(3) 缓冲。

(1) 数据项加锁

我们可以注意到在Branch表中的100行以及它们在History表中最近的行在事务执行过程中被频繁使用。在图10-16中我们可以看到在100TPS的处理能力下，每秒钟有一个事务访问该表的一行，这意味着Branch表每秒有一行被加锁，因此在很多情况下肯定会有许多重叠请求，因此事务会被迫处于等待状态。在这种情况下，许多早期的系统在切换进程环境时效率非常低，因此事务的吞吐量变小。前面我们提到过在测试过程中，10%的事务的响应时间会超过2秒（响应时间标准），因此为了达到事务处理的要求以每秒一个事务的速率访问Branch表的行时会存在一些问题。可以把这视为一个附加的要求，即对Branch表行的加锁时间不能太长，防止访问该表的行成为瓶颈。

由于历史的原因，很多数据库产品不是在表的行上加锁，而是在包括该行的磁盘数据页面上加锁。（DB2直到最近才成为这样的产品，没有官方的关于TPC-A基准测试的测试数据。）对数据页面加锁简化了加锁机制的逻辑，因为对数据页面的加锁在很多种情况下是必需的，比如调整页面中行的位置。很明显，如果一个事务封锁了包含某行的数据页面，那么另外一个企图访问该页面其他行的事务就会与它产生冲突，从而不得不进入等待状态。这种冲突类似于在对行进行加锁时对同一行的加锁冲突。唯一的问题就是对页面的加锁排斥了对该页面

中其他的行的并发访问。

在TPC-A基准测试中有两张表采用页面的加锁机制时有可能导致吞吐量的问题：Branch表和History表。因为没有新的行插入到Account表和Teller表中，我们希望在TPC-A基准测试过程中能够将这两张表全部装载到内存中。对于Branch表中的100字节的行而言，一个DB2的数据页面能够存放40行。每个事务在执行过程中会向History表中顺序插入行，但是我们希望将这些行尽可能放置得更为紧凑些。因此每个DB2数据页面能够存放80个50字节的行。因此对行的加锁对于TPC-A保证足够的并发性方面显得尤为重要。如果使用对页面加锁的机制，对Branch表而言其余39行也将被锁住，这将造成严重的瓶颈。同样，对于History表而言，每个事务对该表的插入行操作是顺序进行的，这意味着新的事务总是在该表的最后一个数据页面中插入行。我们可以假设该页面经常使用，因此放在内存缓冲中。如果使用对页面的加锁机制，将排斥其他很多并发事务对该页面进行插入History行的操作。事务将一直持有该锁直到事务提交，系统将日志缓冲区写到日志文件中，我们可以预见该过程至少要花1/80秒（根据我们对随机I/O的了解）。因此如果采用对页面加锁的机制，事务系统的最大事务吞吐量只能达到80TPS。

为了解决页面加锁协议导致的瓶颈问题，数据库管理员DBA可以做一些工作，比如将Branch表和History表配置成每行存放在单独一个页面中。这样做会浪费大量的空间，但是至少在使用页面加锁协议时，只会对单行进行加锁，因此可以认为前面提到的瓶颈问题得到解决。那么采用这种策略在磁盘空间和缓冲区上我们付出的代价是什么呢？对于Branch表而言，在1TPS的处理能力下只需要一行，即一页磁盘空间。因此在100TPS的处理能力下需要100个磁盘页面。这对于磁盘空间而言并不算什么，即使所有这些磁盘页面都放在内存缓冲区中，对于内存开销而言也不算太大。根据我们在7.2节中提到的费用经验规则，100个4KB的磁盘页面对于每G磁盘空间1000美元的价格来算，只需要0.40美元。100个内存缓冲页面对于\$50/MB的内存价格，需要20美元。

对于History表而言，我们只需要在内存中保存部分的数据页面，事务将在最近使用的磁盘页面中进行顺序地插入，因此内存开销可以更小。但是磁盘开销有些不同。TPC-A基准测试说明（10.2.3.1）中开销的计算要求包括在规定的事务处理能力下存放90天（每天8小时）的历史行所需在线磁盘存储空间的价格。那么在1TPS的处理能力下：有 $90 \times 8 \times 60 = 2\ 592\ 000$ 个历史行。假定每个磁盘页面为4KB，那么100TPS的处理能力必须要有 $252\ 200\ 000$ 个磁盘页面即大约1009GB的磁盘空间，这些空间大约需要2百万美元。而如果用每页存80行的方法，那么只需要25 000美元。这样庞大的历史行存储开销将严重影响系统的\$COST/TPS指标，这是TPC-A基准测试对原始的页面加锁机制的一种惩罚。可能值得欣慰的是，90天History表的在线存储是一个相对比较实际的要求，因此在商业系统中可能被采用。

(2) 写到日志文件

在磁盘上的日志文件通常是一种顺序文件的形式存在的，因此恢复过程必须按放置顺序读入日志记录项。当连续的事务提交和写回日志缓冲区的操作发生时，在一个磁盘写操作开始到下一个磁盘写操作之间有一段较“艰难”的时间。根据随机I/O的最小速率每秒80次I/O操作可以用来合理地估计写日志文件的速度。根据我们前面的讨论，在简单的机制下当事务提交时系统必须将日志缓冲区写回到磁盘上，这意味着我们每秒钟最多只能处理80个事务，

即最大速率80TPS。TPC-A基准测试程序使用非常简单的事务逻辑，可以发现这类瓶颈问题。就像我们在图10-19中看到的那样，每秒几百个事务的处理速度相对而言是非常普通的。在给定的写日志文件的限制下，是如何达到这样的速度的呢？

获得比日志文件I/O速率更高的事务处理TPS的一个可能的方法是采用一种称为组提交的复杂设计。这种方法的基本思路是：改变每个事务提交时都进行写日志缓冲区到日志文件的做法，允许在写回日志缓冲区之前让一组事务提交。当然事务提交操作必须等到日志缓冲区写回到磁盘上时才算完成（直到取款操作的行被存储到稳定存储器上银行才会付钱）。这意味着发出提交操作的事务必须处于等待状态，直到它的提交日志项写到磁盘上，即使写回操作不是立即执行。但是没必要担心等待的时间会过长，因为这样做的目的只是获得更高的TPS值。如果一组事务的提交完成时间过长，我们可以不必等待写到日志文件的完成。我们只是不得不忍受较慢的写回速度——每秒80次，也就是说我们不能摆脱磁盘I/O速率的束缚。基本上采用组提交策略的事务系统在下面两个事件发生时写日志缓冲区到磁盘：

- 1) 内存中的日志缓冲区已满。
- 2) 自从上次写到日志文件之后，已经过了1/80秒。

典型日志缓冲大小为16KB或者更大一些，因此对于较高的TPS能力，在写日志缓冲区之前该缓冲区应该能够存放足够的日志项。

(3) 缓冲策略和5分钟规则

执行图10-17中的事务逻辑时到底需要多少实际的磁盘I/O（当所要的数据页面不在内存中时）？（此处，写日志缓冲区的操作不计算在内。）很明显，这依赖于TPC-A基准测试间隔内进入稳定状态之后，内存中存放的是哪些数据和索引。我们也许可以购买足够的内存来扩大我们的缓冲区，使得所有的数据都能够存放在内存中，但是这会增加系统的\$COST/TPS指标。测试证明将Branch表和Teller表中的行存放在内存中能够获得较好的效果，但是要想将Account表中所有的行都存放在内存中是不现实的。

我们确定哪些数据放在内存中能较大地改善系统的性能，我们使用Jim Gray和Franco Putzolu在1987年提出的5分钟规则(参见推荐读物[3])。他们指出在内存中存放磁盘数据的目的是进行I/O操作时磁盘读写臂的移动时间尽可能小。更多的内存可以使我们减少需要购买磁盘的数量（我们假定我们可以将相同的数据放在多个磁盘上）。考虑到这一点，我们愿意花更多的钱购买足够的内存作为缓冲区。作为花钱购买更多的内存的折衷措施，我们可以将更多的磁盘数据放在内存中，这样可以减少购买磁盘上的投资。在给定的工作负载、内存价格和磁盘价格条件下，存在一个合理的临界点使得我们对内存的投资效果达到最好。磁盘页面的使用情况用两次数据访问之间的时间来衡量，访问的频率越高说明越常用。Gray和Putzolu指出两次访问之间的时间间隔决定了什么时候我们可以停止购买更多的内存作为缓冲区，这个时间大约是5分钟——因此我们称为5分钟规则。但是在稍微不同的前提假设下我们得到的时间间隔超出了我们预计的值，我们将推导我们的结论。

例10.10.1 5分钟规则 简单起见，假定我们为一个五年内工作负载不变的系统购买磁盘和内存，因此我们不必担心短时间的租率。购买更多的磁盘我们可以获得更高的磁盘访问速度，因为对于每个数据页面的我们可以重复存放在多个磁盘上，每个磁盘可以同时读取该数据页面的不同部分的数据，从而提高整体的访问速率。（有些磁盘有多个磁盘臂，可以同时在不同的区域读取数据。）当然买更多的磁盘需要更多的投资。比如，在200TPS的处理能力下，

按照TPC-A基准测试Account表包含2亿行，每行有100个字节，也就是说需要2GB的空间（我们现在忽略索引所需要的空间）。商业上，现有的磁盘容量为每个磁盘臂管理10GB，最大的I/O访问速度为每秒80个I/O操作。我们假定10GB的磁盘价格为500美元。在200TPS的处理能力下，每秒要对Account表执行200次读操作和200次写操作。对于每个磁盘臂每秒50次I/O操作的正常速度（比最大速度每秒80次I/O操作要小一些，因为要达到100%的磁盘利用率不太可能），这意味着至少需要8个磁盘臂同时对Account表进行操作。因为Account表包含2GB的数据，而80GB的磁盘空间需要8个磁盘臂。我们可以看到空间利用率只有大约2.5%。存放Account表的8个磁盘大约需要4000美元。如果我们可以将这些数据全部放在内存中而不是存放在磁盘上，我们可以减少所需磁盘臂的数量。整个Account表只需要放在一个磁盘上，可以节省大约3500美元。

将2GB数据放在内存中需要8000美元的代价(每GB 4000美元)，因此将整个Account表放在内存中来节省3500美元的磁盘开销是不适合的。但是如果内存的价格降到足够低，比如每GB 1000美元，而磁盘的价格仍然维持不变，那么将整个Account表存放在内存中是合适的，因为此时只需要2000美元，就可以节省3500美元的磁盘开销。从这个例子中我们看出，到底用多大的内存作为缓冲区是一个内存价格和磁盘价格的折衷方案。 ■

例10.10.2 关于5分钟规则更多的内容 在与例10.10.1相同的假设条件下，我们来推导磁盘页而的引用频率的临界值。当磁盘访问的频率处在该临界值时，将数据存放在内存中和存放在磁盘上具有相同的花费。假定这个值为X，单位是每秒钟内对一个数据页面（4KB数据）的访问次数。假定内存价格为\$4000/GB，则在内存中保留4KB数据的花费为\$4000/256 000 (1GB/4KB = 256 000)，即：

$$\text{驻留内存的花费} = \$4/256 = \$0.015625$$

另一方面，我们可以使用由X/50给出的磁盘臂来完成每秒X次存取，假定当没有等待队列时每个磁盘臂平均每秒处理50个I/O操作。因为每个磁盘的价格为1000美元（根据图8-2的数字），这意味着当该页而在磁盘上时提供每秒X次存取，要求的花费是：

$$\text{驻留磁盘的花费} = \$1000 \times (X/50) = X \times \$20$$

在临界值时两个花费应该相等，因此我们可以通过求解下面这个简单的方程来获得临界值。

$$\$0.015625 = X \times \$20$$

求解这个方程，我们可以得到 $X=0.00078125$ ，这是每秒的存取次数，或者说每隔 $1/0.00078125=1280$ 秒才会发生一次磁盘访问，这是访问间隔的临界值。经常用到的数据页面可能不需要1280秒这么长的间隔就会发生一次磁盘访问，也就是说有相对较大的X值。我们可以看到随着X的增大，驻留磁盘的解决方案花费增大，而驻留内存的解决方案花费不变。因此那些经常要访问的数据页而适合放在内存中。反过来对于不经常访问的数据页面（平均访问间隔大于1280秒的数据页面）则适合放在磁盘上。 ■

注意，1280秒大约比21分钟多一些，因此我们可以看到随着内存价格快速下降（比磁盘价格的下降速度要快得多），5分钟规则变成21分钟规则。现在，内存的价格仍然比磁盘的价格要降得快，因此随着内存变得越来越便宜，这个临界访问间隔会变得越来越长。再过20年，这个临界访问间隔将变得非常长。

有趣的是TPC-A基准测试的缓冲机制并不特别关心响应时间。事务操作过程可能花去2秒，而额外的I/O操作只花去50毫秒，因此实际上我们可以忽略后者。我们关心的好缓冲策略是减少系统的费用\$COST。在TPC-A基准测试过程中较好的事务显示对每个事务而言实际的I/O次数为每张表2次多一点。如果Account表的索引使用散列算法可能会有更好的结果，因为使用B树结构的话有可能使叶节点不能放到缓冲区中，从而使得I/O的次数可能超过每秒3次。Teller表和Branch表的访问频率比Account表的访问频率要高得多，因此在内存中的驻留时间也会更长一些，许多事务要对它们进行更新操作。很明显，当我们对某个缓冲页面进行更新时不应该立即将这些数据页面写回到磁盘上。History表只需要每80个事务写回一次，假定该表的行尽可能紧凑地存放在该页面上。

频繁访问的数据我们称为拥有热度。当数据有足够的访问频率时就可以放在内存缓冲区了，这些数据称为热数据。Branch表和Teller表就是这样的例子。当数据不再需要存放在缓冲区中时，为了提供更快的磁盘臂服务这些数据就要以尽可能快的方式写回到磁盘上，我们称这些数据为暖数据。我们可以在Account表中看到这种操作。事务系统通常处理的是暖数据。另一方面，对于查询系统而言经常有这样的情况：具有非常大的存储需求且相对较低的并行使用需求，因此存储能力就成为影响性能的一个重要因素，我们称这些数据为冷数据。History表就代表冷数据，但它需要在内存中保留一个缓冲区接受新插入的数据，因为当这些记录写之后没有实时的引用（至少在TPC-A基准测试程序中没有）。

推荐读物

在这里，我们要推荐两本关于事务的教科书。第一本是由Bernstein、Hadzilacos和Goodman写的[1]。该书已经不再印刷，但是可以从其他渠道获得该书，可以从<http://research.microsoft.com/pubs/ccontrol/default.htm>下载该书的PDF格式的文件。这本书主要证明了本章中提到的各种方法的正确性。而且该书还涉及到了模糊检查点的一些实现技术。本章中没有提到的很多的并发和恢复方法在该书中有详细的表述，比如多版本并发和屏蔽技术。（这些方法在现在的商业系统中很少使用，但是这种情况在将来可能会改变）。

第二本书是Gray和Reuter写的[2]。该书从另一方面向读者展现了我们在本章中讨论的数据库特征的具体实现细节，还有其他很多内容。证明已经给出，但是主要的焦点是如何让数据库工作起来。这是一项基本的工作，没有它任何事务系统都不可能获得成功。

- [1] P.A.Bernstein, V.Hadzilacos, and N.Goodman. *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison-Wesley, 1987
- [2] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. San Mateo, CA: Morgan Kaufmann, 1993.
- [3] Jim Gray and Franco Putzolu. “The 5-Minute Rule for Trading Memory for Disc Accesses and the 10-Byte Rule for Trading Memory for CPU Time.” *Proceedings of the 1987 ACM SIGMOD Conference*, pp.395-398.
- [4] K.Kant. *Introduction to Computer System Performance Evaluation*. New York: McGraw-Hill, 1992.
- [5] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O’ Neil, and Patrick O’ Neil. “A Critique of ANSI SQL Isolation Levels.” *Proceedings of the 1995 ACM*

SIGMOD Conference, pp. 1-10.

习题

用•标记的习题在书后的“习题解答”中给出了答案。

10.1 这个习题中，假定每个事务每次使用CPU时间片为10ms，每次I/O操作为50ms。和例10.2.1中的讨论相似，只是参数有些不同，考虑下面的问题：

- (a)• 使用图10-6中给出的方法，画一张等价的资源利用图，要求使用最多的磁盘数和最多的模拟事务线程，通过理想的资源重叠获得最优的吞吐量。假定每个事务要求两次I/O操作和两个CPU时间片。求最终的事务吞吐量，以TPS为单位。
- (b) 考虑更为实际的情况，我们采用统计学的资源分配方法。在10个磁盘上进行数据的随机分配。假定在某个时刻有20个并发事务，队列中5个事务使用CPU，另外15个在使用磁盘I/O。
 - (i) 以类似于向10个磁盘上投掷15支飞镖的模式。计算磁盘数目的期望值，要求不必移动磁盘臂。并给出这种情况下磁盘利用率。不要使用近似的方法。
 - (ii) 在队列中有5个事务使用CPU的模式下（类似于在散列中平均队列长度为5），计算CPU的利用率。

10.2 绘制下面两个经历的前趋图

- (a)• $W_1(A) R_1(A) W_1(Z) R_2(B) W_2(Y) W_3(B) C_1 C_2 C_3$
- (b) $W_3(D) R_1(D) W_1(F) W_2(F) R_2(G) W_3(G) C_1 C_2 C_3$

10.3• 考虑下面的经历：

$R_1(A) W_1(A) R_2(A) R_2(B) C_2 R_1(B) W_1(B) C_1$

该经历的前趋图中有一个回路，然而我们可以证明如果我们依照该经历在数据库中单独执行一遍，不会发生任何不一致的情况。事务 T_2 确实看到了事务 T_1 的部分结果，但是它没有采取任何操作，因此没有影响磁盘数据。另一方面，事务 T_1 看到的是一致的数据，因为 T_2 没有做任何更新。因此有人也许会问：为什么我们必须说这样的经历是不可串行化的，既然唯一的不可串行化的数据视图对最终结果并没有什么影响。（这一说法的缺陷在于如果它看到了一致性数据视图 T_2 有可能采取其他的操作。）说明并给出该经历的一个解释，其中这种不一致性可能产生问题。

10.4 证明如果PG(H)图中有回路，那么不存在不具有PG(H)图的自右向左的边的事务序列。（这是可串行性定理的一部分，在此留作习题）

10.5• 说明锁调度器会如何处理下面的操作序列，使用类似于例10.4.2的方法。

$R_1(A) R_2(A) W_1(A) W_2(A) C_1 C_2$

10.6 像前面的习题那样，说明锁调度器会如何处理下面的事务操作请求序列。画出死锁发生时的等待图。

- (a)• $W_1(A) R_1(A) W_1(Z) R_2(B) W_2(Y) W_3(B) C_1 C_2 C_3$
- (b) $W_3(D) W_1(F) R_1(D) W_2(F) R_2(G) W_3(G) C_1 C_2 C_3$

10.7 在下面的假设条件下考虑内存的磁盘页面高速缓冲机制：首先，高速缓存磁盘页面区为空，高速缓存中能保留的最大页面数目为4个页面（这是一个不切实际的数字，此处作为理解练习使用）。假定经历中要访问的数据项A, B, C, D, E, F位于不同的页面。

$H = R_1(A,1) R_2(B,2) W_1(A,3) R_3(C,4) W_2(B,5) C_2 W_3(C,6) R_4(D,7)$

$R_5(E,8) W_5(E,9) R_6(B,5) R_6(A,3) R_3(F,10) W_3(F,11) W_4(D,12)$

- (a) 指出第一个操作，当执行该操作时为了读入新的数据页面，缓冲区中的某个数据页面必须被替换出去。
- (b) 如果缓冲区中的数据页面在缓冲区中被更新之后，没有写回到它们所在的磁盘上，我们称这个缓冲区中的页面是脏的。指出在执行(a)中的操作时，内存中有哪几个脏的数据页面。
- (c) 假定我们使用LRU模式并且只有在对页面进行读写操作时我们认为该页面处于使用状态。指出在执行(a)中的操作时，哪个页面将被替换出去。
- (d) 我们是否可以简单地将(c)中指出的页面丢弃，不管它的值，或者必须采取其他某些措施？为什么？
- (e) 当在上面的经历执行 C_2 时，该操作是否会导致带有事务 T_2 修改的数据项的页面被强制移出？
- (f) 列出上面所有的操作以及执行每个操作后存放在缓冲区中的数据项页面以及它们当时的值，如果当时缓冲区中的值和磁盘上的值不一样，那么也列出该数据项在磁盘上的值。下面是你应该使用的格式的一个例子。

操作	缓冲区数据项的值	磁盘上的不同值
$R_1(A,1)$	$A = 1$	
$R_2(B,2)$	$A = 1, B = 2$	
$W_1(A,3)$	$A = 3, B = 2$	$A = 1$

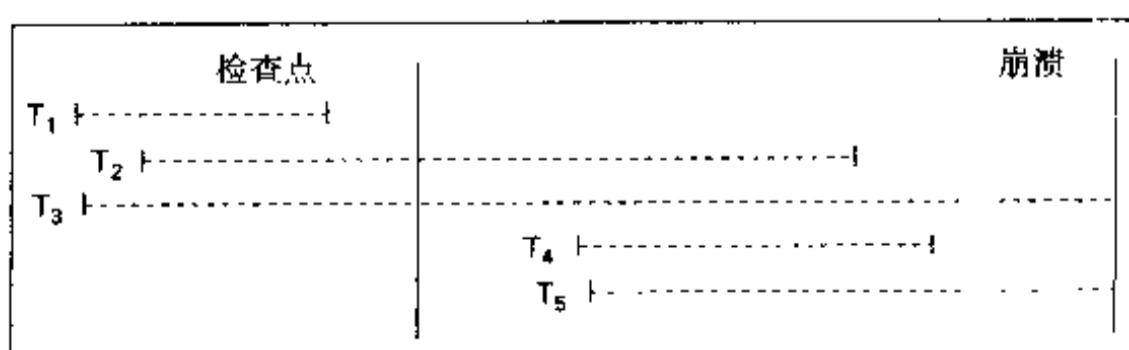
10.8 在下面的经历中，我们假定在该经历开始之前没有任何事务存在，每个事务以下面列出的操作作为开始的第一个操作，类似于例10.8.1。对于每个经历，提供下列问题的解答：

- (i) 假定使用高速缓存一致性检查点，写出这些操作产生的日志记录项。
- (ii) 从磁盘上的日志文件的最后一个日志项开始，写出高速缓存安全的ROLLBACK和ROLL FORWARD操作序列。
- (iii) 像例10.8.1那样画出事务持续的时序图。

(a) $H1 = R_2(A, 1) R_1(B, 2) W_1(B, 3) R_3(C, 4) W_2(A, 5) W_3(C, 6) CKPT R_4(D, 7) R_2(E, 8) W_2(E, 9) C_2 R_4(E, 9) W_4(D, 11) Crash$

(b) $H2 = R_1(A, 1) W_1(A, 3) R_2(D, 2) R_3(B, 4) C_1 W_2(D, 5) W_3(B, 6) CKPT R_3(C, 7) R_2(E, 8) W_2(E, 9) C_2 R_3(E, 6) W_3(C, 11) Crash$

10.9 在下面给出的时序图中，在做了一个高速缓存一致性检查点之后系统崩溃。下面各个事务的持续时间只是示意性的。证明为什么这是正确的，然后考虑每种情况并解释为什么我们在前面讲述的恢复过程能够在该情况下正确工作。你要做的不仅是向自己也要向别人证明该算法是正确的。



10.10 使用高速缓存一致性检查点时，是否有可能某个事务的活动期跨过不止一个检查点？如果不可能，说明原因。如果有可能，解释恢复措施仍然能够起作用的原因。

10.11• 给出一个至少包含三个事务 (T_1, T_2, T_3) 的经历 H ，该经历具有下面的属性：(1) 事务 T_1 的每个操作都在事务 T_2 的任何一个操作之前，而事务 T_2 的每个操作都在事务 T_3 的任何一个操作之前；(2) 在串行图 $SG(H)$ 中，我们有以下关系 $T_3 \rightarrow T_2$ 和 $T_2 \rightarrow T_1$ ，写出等价的串行经历。(提示：你需要另外两个事务 T_4 和 T_5 ，并且 $SG(H)$ 中的序列是遵循传递性的。) 指出一点，我们在此处没有定义锁机制。因此你可以自由地设计可串行化经历，独立于锁机制带来的时序上的限制。

10.12 让我们回想一下在例 10.5.3 中提到的幽灵问题。幽灵问题的一个典型的例子是：一个事务通过插入操作向某个集合中添加了一个新的行，而另外一个事务正在这些行上执行某些集合操作，从而使后者得到不一致的结果。请解释一下谓词锁是如何解决这个问题的。假设事务 T_1 要统计数学系的雇员人数，而与此同时事务 T_2 需要向该系添加一条新的雇员记录。

- (a) 说明在不使用谓词锁的情况下，幽灵问题是如何发生的。
- (b) 使用谓词锁的情况下， T_1 锁的是什么数据，是什么模式的锁？
- (c) T_2 锁的是什么数据，是什么模式的锁？
- (d) 该方法是如何解决幽灵问题的？

第11章 并行和分布式数据库

在前几章中我们讨论了传统数据库的大部分概念，但这些概念都局限于旧模式下单一CPU数据库系统框架的计算机环境。在这种环境下，一般是多个终端连接到一个单一CPU的主机上，而这个主机的功能很强，一般有多个磁盘驱动器。这种结构的模式如图11-1所示。这里不打算在这种框架上做过多的讨论，只是假设单一CPU进行的是分时计算，即按时间片的方式运行多个不同终端的过程（或线程），执行独立应用逻辑流。20年以前这种标准的数据库环境还是很符合实际的，但是现在大多数商业应用中新的数据库系统体系结构已经取代了它们。出于种种原因，现在的大部分系统中都使用多个CPU以并行方式提供数据库服务。

本章主要讨论一些支持多CPU的数据库系统体系结构。在这些体系结构中，有些假设各CPU承担相同的数据库服务责任，它们往往物理上放在一起，如在同一栋大楼中，而且相互间可以高速通信；另一些则假设CPU是地理上分开的，如可能在不同的城市，并且相互间通过电话线以相对较慢的速度进行通信。物理上在一起的多CPU系统一般叫并行系统或并行体系结构，地理上分开的多CPU系统一般叫分布式系统。这两种体系结构在许多方面都有差别，而且许多数据库操作的基本概念都要依赖于这些体系结构的差别。对所有不同的体系结构进行讨论已经超出了本书的范围。在以下几节中只简单介绍一些最基本的概念，并列出一些更深入处理这些体系结构的参考资料。

11.1 多CPU体系结构

本节将描述三种多CPU数据库体系结构，在这些体系结构中各CPU提供相同的数据库服务；并且描述一种称为客户机-服务器结构，在这种体系结构中各CPU承担不同的职责。新的系统类型正在不断地出现，而且这里所描述的也并不全面，但是讲述了大多数最基本的原则。

正如前面所说，具有并行体系结构的数据库系统是多个物理上连在一起的CPU，而分布式系统是多个地理上分开的CPU。这两种体系结构其实都是为了满足不同的需要。并行系统主要是为了构造一个既快又廉价的集中式计算机，同时又可以避免使用一个高速CPU。正如在下一节中看到的，用多个低档CPU来代替一个高档CPU会更经济一些。另一方面，分布式系统是为了提供在地理上分开的部门（如大公司的子部门）提供局部数据库自治的能力。分布式数据库方法使不同的分布式系统之间的通信成为可能，这样数据在不同地理位置的机器上可以进行有效地访问。这里将从并行体系结构中的紧密耦合的CPU开始，进一步介绍松散耦合的分布式系统。

第一个并行数据库系统类型是共享内存式多处理器，它就是一个计算机上同时有多个活

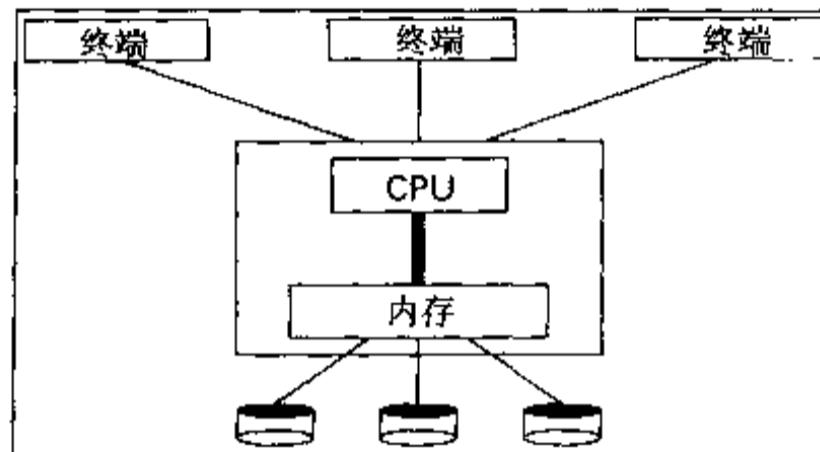


图11-1 单CPU数据库系统体系结构

动的CPU并且它们共享单个内存和一个公共磁盘接口。共享内存式处理器体系结构的模式图如图11-2所示。这种并行体系结构最接近传统的单CPU处理器结构，其设计主要挑战是用N个CPU来得到N倍单CPU的性能。但是，因为不同的CPU对公共内存的访问是平等的，这样就可能一个CPU正在访问的数据被另一个CPU修改，所以这种设计要有一些特殊的处理。然而由于

内存访问采用的是一种高速机制，这种机制很难在进行内存划分时不损失效率，所以这些共享内存访问的问题会随着CPU个数的增加而变得更加难以解决。目前最大的IBM OS/390也只有12个并行CPU（另一种称为大规模并行体系结构，它允许有几百个CPU同时访问公共内存，但需要一些特殊的设计，这里就不介绍了）。

第二种系统类型是无共享式并行体系结构，也就是说一个计算机上同时有多个活动的CPU并且它们都有自己的内存和磁盘。在不产生混淆的情况下，也称它为并行数据库系统。各个承担数据库服务责任的CPU划分它们自身数据，并且通过划分任务以及通过每秒兆位级的高速网络通信完成事务和查询。在图11-3中这些高速网络用粗线表示，用来联接各系统节点(简称节点)。这种高速网络规模的大小是有限的，这就要求并行体系结构具有物理上在一起的CPU。这种网络也就是通常所说的局域网，即LAN。

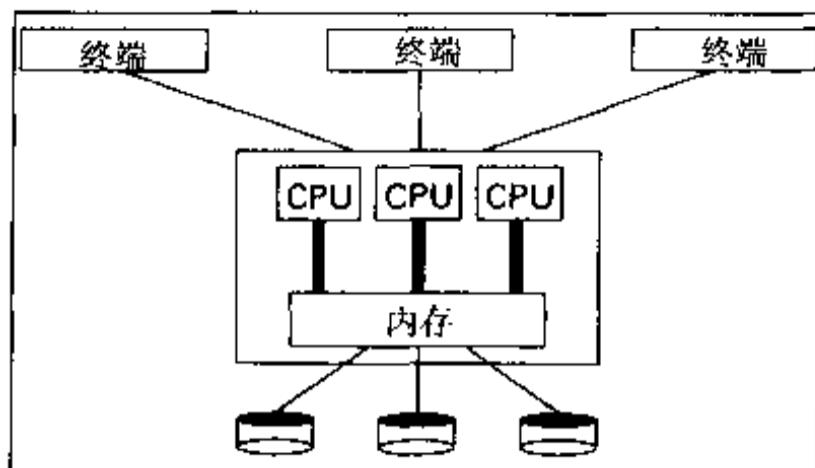


图11-2 共享内存式多处理器体系结构

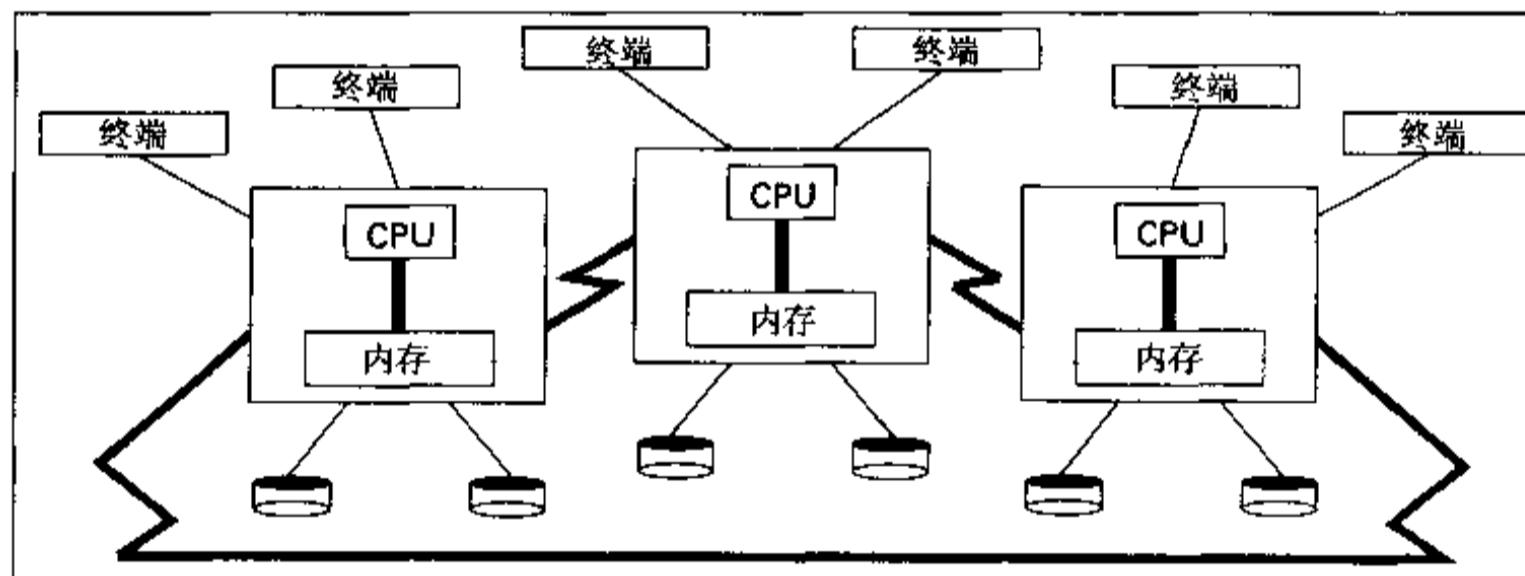


图11-3 并行无共享式体系结构

第三种系统类型是分布式数据库系统，由于内存不能在相隔很远的CPU之间进行共享，所以这也是一种无共享型结构。同样，系统数据库也是划分在不同的自治站点上，这样就要求查询和其他数据操作语句能在不同的站点上独立地执行，并且部分结果可能需要在一些相关的CPU之间进行通信。但是，在不同城市的CPU之间的通信与每秒兆位级的网络相比是很慢的。在图11-4中各CPU之间的联系用细线表示。与图11-3对比，可以发现除了这个差别以外，它们是完全相同的。

并行数据库系统往往是从无到有提供最好的性能价格比，并且各个站点可能都采用统一的体系结构。而另一方面，分布式数据库系统则往往是把以前在不同场所已存在的系统联系起来，结果，在不同站点的机器往往是异构的，带有各自不同的体系结构，如一个站点上可

能是Sun Solaris UNIX系统上的ORACLE系统，另一个站点上可能是OS/390机上的DB2系统，第三个站点可能是NT机上的Microsoft SQL服务器。这些机器往往有不同数据表示的数制（如浮点数），也可能有不同的SQL语法（在不同站点之间的查询通信要使用通用的X/Open语法）。在并行系统中不同站点机器的合作可通过数据库系统的事务管理器模块来实现，但在混合了带有不同数据库系统的异构站点的体系结构中这一般是不可能的。正是出于这个原因，出现了一种叫TP监视器的软件来连接这些站点。TP监视器在不同站点的单独数据库之上，并使用它们来提供局部所需的服务。TP监视器提供线程来运行用户程序和远程过程调用来实现对远程站点的请求调用。在这种调用的通信参数中监视器提供了所需的数据表示的转换。TP监视器是一个非常复杂的系统，已经超出了本书讨论的范围，详细内容可以参考Gray和Reuter所写的文章(推荐读物[5])。

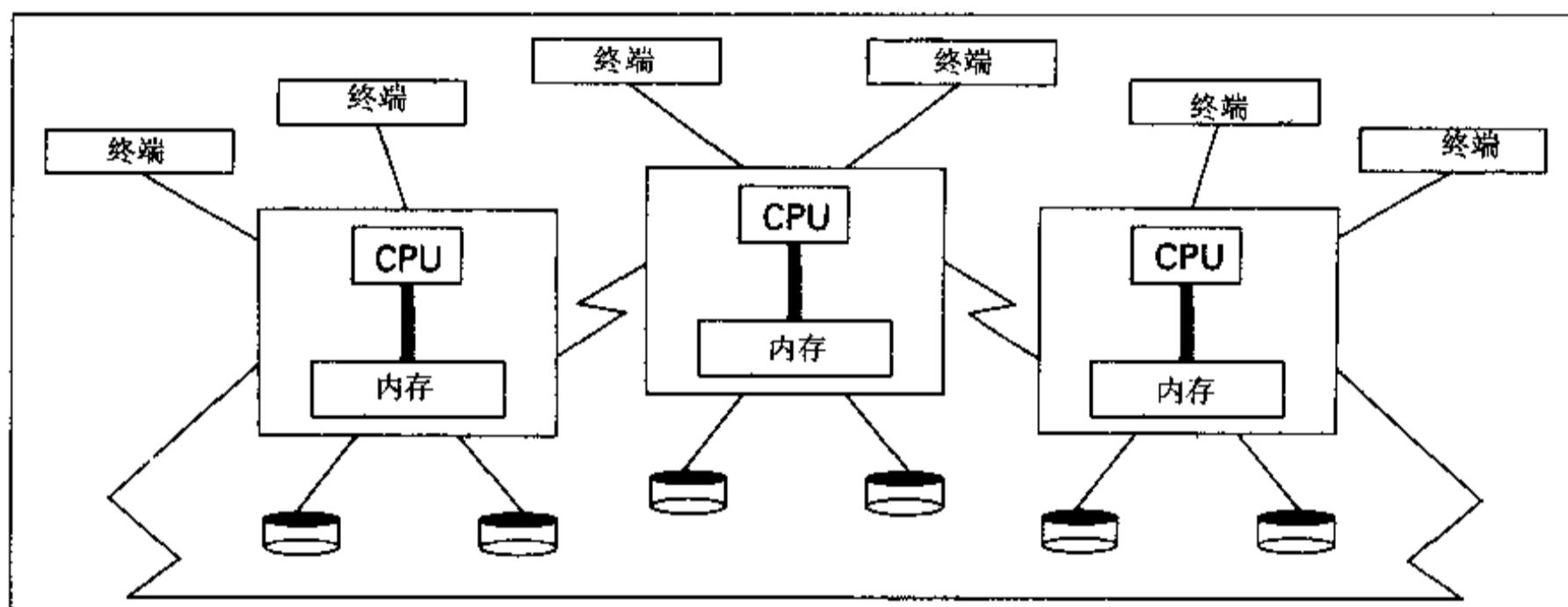


图11-4 分布式（无共享）体系结构

在图11-2、11-3、11-4中，所有的CPU对系统提交的数据库服务承担相同的责任。在无共享式体系结构下，数据被划分在不同站点的磁盘上，引用了多个站点数据的查询需要这些CPU之间的相互合作，并把结果返回给输入查询的用户终端。显然这些体系结构都对数据库设计有很重要的含义。当数据库中的表划分在不同城市间的磁盘上时如何进行查询优化？当一个CPU不能在一个更新事务中更新它所控制的数据时整个事务该怎么办？数据库实现者们正对这些重要问题进行不断的探索。

客户机-服务器体系结构

如图11-2和11-3所示，并行体系结构的动机是用多个小的CPU来取代一个大的CPU，这样可以更经济一些。但是对这种所有CPU承担相同职责的结构来说，还存在另一种体系结构类型可以选择。使用客户机-服务器体系结构，小的客户机CPU（往往是PC机）只要负责与用户交互功能、提供数据表示功能和决定需要回答用户请求的数据。客户机可以在本地磁盘上没有用户所需的大部分数据，它只要向集中式服务器发送高层数据请求（SQL级请求或程序的远端过程调用）就可以了，而服务器一般为共享内存式多处理器。客户机也可能与更复杂的并行服务器或甚至与多个服务器打交道。客户机-服务器系统的主要特点是客户机CPU与服务器CPU之间的职责的划分，客户机CPU主要负责数据表示服务，而服务器CPU主要负责数据库服务（如图11-5）。

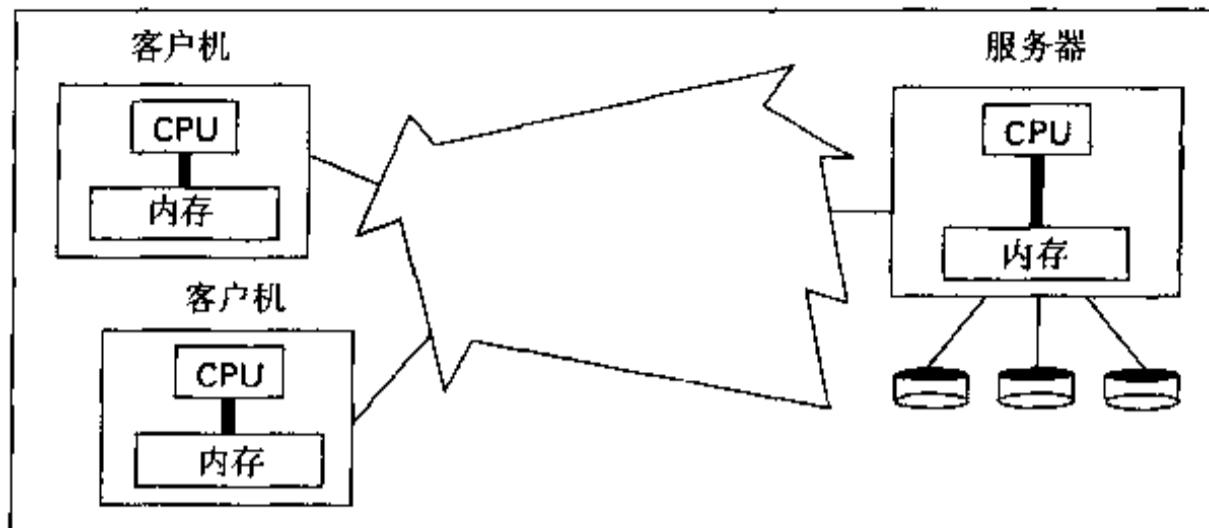


图11-5 客户机-服务器体系结构

11.2 CPU价格与性能曲线

前面已经提到过图11-2和11-3所示的并行数据库结构主要是为了获得更高的性能价格比。现在就这一点展开详细讨论。

用指定CPU的\$COST来表示购买一个CPU所花费的批发价。要注意的是大多数超过\$1000的PC机，它们的CPU往往低于\$100；其余的开销主要花在磁盘、软件等方面。CPU的性能可用每秒百万条指令来表示，也可简记为MIPS。但是，因为CPU可以完成许多种工作（科学计算、图形显示、各种不同类型的商用数据库应用程序），所以这种评定CPU性能的方法是有问题的。这就要求基准测试程序要能准确地测量出各种情况。一旦基准测试程序确定以后，它们就成为供应商们所追求的性能目标了。对某些著名的基准测试程序，计算机设计者采取了特殊操作和编译调整以在基准测试程序中达到更高的性能。在许多情况下，在基准测试程序逻辑上做一很小的调整会引起许多指标急剧下降。同样在很多情况下，人们衡量机器性能的指标与出售计算机的厂商一样，这样在性能评估上就有可能会有一些偏差。因此，一般50MIPS的PC机往往要在性能上大大低于IBM大型机的50MIPS计算机。也正因为上面所提到的这些困难，IBM在其他产品上有时并不支持MIPS指标。

由于大家都喜欢一个简单的答案，因此下面给出不同CPU的MIPS值。这个MIPS值用来反映在所有数据库系统应用程序中CPU的性能。那么在今后大约15年中计算机体系结构的曲线将如图11-6所示，它表示价格与性能之间的关系。图11-6中具体的数字不一定准确（因为CPU价格和技术一直在变化），但大体形状是正确的。

从图11-6中可以看出\$COST与性能的关系是超线性的。这也就是说，如果单个50MIPS的CPU价格为\$100，则单个100MIPS的CPU的价格就要超过\$200了。

目前，一个20倍50MIPS CPU性能的1000MIPS CPU要\$10 000 000，是一个50MIPS CPU价格的10 000倍。这主要是因为CPU速度越高，则越接近这项技术的上限，就不得不为采取更多

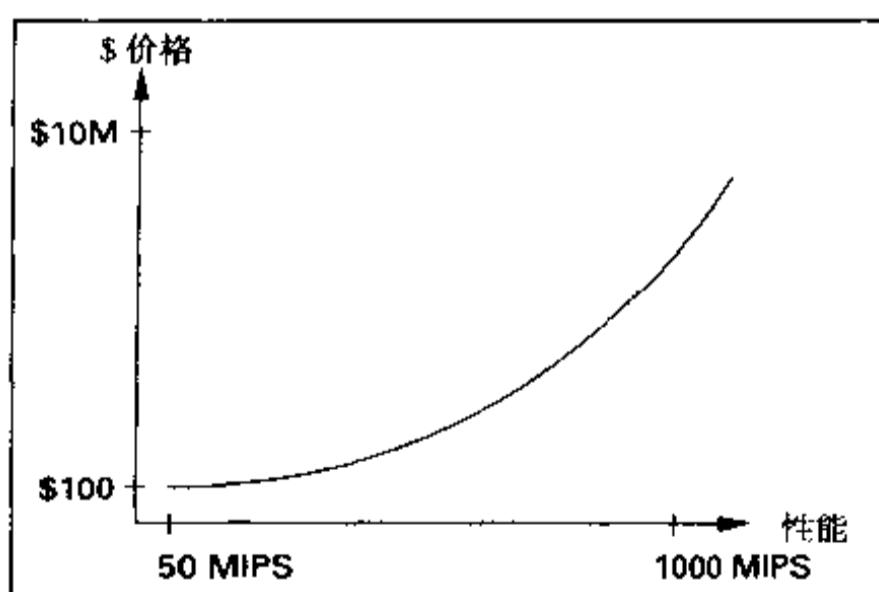


图11-6 CPU的价格与性能的关系

的复杂技术来实现的更快速度付更多的钱。就像要寄一封信到欧洲，可以寄普通信件很便宜但需要2天；也可以派专人乘飞机直接送过去，这当然很快，但花费也要大得多。

数据库行业中的规律是什么？非常简单：多个小的CPU比一个大的CPU更经济。如果一家商业公司拥有一个老式体系结构的主机，而它的大部分时间主要花在为一组终端用户提供交互服务的分时模式上，则该公司可以为每个终端用户买一台PC机以提供其所需的交互服务，然后把所有PC机作为客户机连结到一个集中式服务器上，又因为这个集中式服务器不需为许多终端做大量的数据表示服务，所以容量较小的计算机就可以胜任。在这种设计中，服务器只要去访问在大量磁盘中的数据并给用户返回简单结果就可以很好地做大容量机器所做的工作。可以从图11-6中看出价格/性能曲线促使企业采用客户机-服务器体系结构。确实，所有以前提到过的各种多CPU数据库体系结构都说明：应该学会如何用几个小的CPU来完成一个大的CPU的工作。

并行处理某项自然可分的工作是很简单的。比如一组技术人员进行文档处理，他们除了把打印好的文本给别人检查以外无需进行额外的相互通信。在这种情况下，一个合理的解决方法是给所有用户配置PC机和一些单用户的字处理软件。这种工作环境就可以取得所希望的并行性，而又无需用一个高档的分时计算机进行字处理。这样做往往更经济一些。

但是一些数据库应用程序中需要大量的相互间通信，这使应用程序在一组不同CPU上运行的必要性不是很明显，这些CPU通过一个高速网络通信时，更是如此。一个好的思路是进入支持数据库应用程序的系统体系结构中，不考虑这些应用程序的通信开销可能会抵消使用小的CPU所节省的花销。此外，一个数据库系统在数据访问设备上投资往往较大，如磁盘驱动器（现在的工作站系统花销的50%），并且一个粗心的存储方法也会使体系结构非常不经济。这样可以看出CPU价格并不是分布式数据库系统唯一的价格因素。寻找最佳的并行数据库系统体系结构的研究一直在持续，在后面章节中会对出现的问题进行一些讨论。

11.3 无共享式数据库体系结构

图11-3和11-4中的无共享式数据库体系结构的一般概念如下所述。考虑到性能价格比或地理分布，把所有的数据库服务放在一台大型计算机上往往是不可行的，因此往往把数据分开放在许多不同的带有独立内存的低成本计算机上。这些不同的系统（称其为站点或站点机器）必须能相互合作进行数据库操作，这就要求站点之间必须能进行相互通信。

例11.3.1 TPC-A基准测试程序。10.10节中的TPC-A基准测试程序没有提到在多CPU无共享式数据库系统环境下应用这种基准测试程序的可能性。但是，该基准测试程序的设计者早就预见到这一点，采取某些规则就可以确保这项测试是现实的。要在无共享式系统中运行TPC-A，需要把数据分开放到多个机器上。假设有10台站点计算机并用100-TPS指标进行测试，则在Branch表中有100行。此外每一出纳员和账户（Teller和Account表中的行）都与一个特定的部门有关系，设每一个部门有10个出纳员和100 000个账户。为了用分布式数据库系统中的每个站点机器来支持TPC-A中工作量，需要将表在10个站点中进行均分。标准做法是每个站点都有自己的Branch、Teller和Account表，则每个站点上的Branch表中只有10行（因为Branch表的100行在10个站点上进行了划分）。Teller和Account表也在10个站点上进行了划分，并存入相应的站点中。

现在简单描述一下TPC-A的事务逻辑，每一事务代表一个账户持有者（Aid）去某个银行

部门 (Bid) 的某个出纳员 (Tid) (随机的) 处从自己的存款账户中取款 (Delta)，此次提款同样也反映在Branch表和Teller表中。是否需要在站点之间进行通信呢？如果账户持有者在本地银行部门提款，则无需访问其他站点的数据。但是还存在其他的可能性。随机地产生 (Aid, Tid, Bid 和 Delta) 信息，其中 85% 表示 Aid 属于本地的 Bid 中，但是这些信息的另外 15% 可能性是 Aid 属于外地的部门。这种情况显然需要进行通信，由于每个站点有 10 个部门，大约有 90% 需要与其他的站点进行通信。要注意的是事务需要给本地站点 Branch 账目和本地站点 Teller 账目加上 Delta，同时在 Account 账目上也要加上 Delta。这就暗示更新不同站点上的 Account 表是更新本地的 Branch 表和 Teller 表事务的一部分。显然，这就意味着我们需要重新观察 ACID 特性，以确保分布式事务被正确处理。■

首先例 11.3.1 中通信开销不是很严重，似乎只要在 15% 时间中发一个消息并在外地站点上做一下更新就可以了。问题是在一个事务中要更新的不同行出现在不同的内存中，被不同的处理器加锁，因此也不能像传统集中式数据库体系结构中那样同步事务提交。假设需要启动另一站点上的一个事务做一更新，然后两个事务协同提交。虽然从整体上来说它似乎是一个非常简单的要求，但是实际上它是一个非常不明确的要求。进一步研究就会发现在执行分布式事务时会出现许多复杂的问题。下面几小节对其中几个问题进行探讨。

1. 两阶段提交

现在考察前面提到的情况：一个本地事务突然访问远地站点的数据而需要启动异地另一事务。比如，在无共享式系统中的 TPC-A 中考虑如下情况：一个账号持有者在站点 2 上访问站点 1 的银行部门信息。从图 10-17 的事务逻辑中我们可以看出第一个数据访问是在站点 2，而出纳员与终端的交互是在站点 1。因此分布式事务 T_b 在站点 1 上初始化并协同运行，而本地事务 T_a 则运行在站点 2 上。第二次去访问 History 表是在站点 1 的本地机器上，因此分布式事务 T_b 现在也在站点 1 上有一本地事务 T_1 。当 TPC-A 事务逻辑提交时，为成功完成分布式事务 T_b ，事务 T_1 和 T_a 必须协同提交。

在大多数带有异构站点的无共享式数据库系统中，访问远程数据对应用程序来说是透明的，它往往由数据库系统的事务管理器 (TM) 来提供，它具有在不同站点机器上如何划分数据库的全局图。参见图 10-8 以及其后的有关 TM 的语义含义和它在事务系统中地位的讨论。在无共享式体系结构中，TM 决定哪个站点启动本地事务、推进访问和操作合适站点的数据的进度程序和等待回答，并负责分布式事务提交。当 TM 最后决定提交（或回退）时，事务可能已经读取或更新几个不同站点上的不同数据项目（TPC-A 事务太简单了以至于涉及到多于两个站点的情况）。

如果分布式事务的各站点事务都能保证一起提交或终止，则称它们是协同的。假定相互协同条件，分布式事务就可以从各站点事务继承 ACID 特性。如果分布式事务在任何时候访问任意站点上的数据都采用了两阶段加锁，则可以实现作为包访问的分布数据的隔离（加锁定理 10.4.2 在分布式事务中仍然成立）。正如前面所说的，隔离性就意味着一致性。只要提到 ACID 性质中的持久性时，就要考虑一个站点崩溃时会发生什么样的情况。一个已提交的分布式事务会在所有本地站点中写入提交日志（要记住现在假设所有参加的站点事务都一起提交或中止），因此当崩溃站点恢复以后，系统可以进行重做恢复。当然一个站点可能会多次崩溃，这就需要一些新的恢复方法来处理分布式事务的恢复，但这并不难。

这其中有一个关键的假设：分布式事务的所有站点事务都可以是协同的，即可以一起全

部提交或中止。这也就保证了分布式事务的原子性，即事务的更新要么全执行要么全不执行。但是目前所遇到的本地事务类型都不能协同不同站点的事务，即所谓的基本事务行为。正如我们已经描述的，一个由调度程序初始化的基本事务只有三个状态，事务只能以这三种状态存在，如下所述：

激活 (Active) 态 当某一程序访问数据时，作为响应，调度程序就会启动一个事务，从而使事务进入激活态。一个事务可以从激活态转变到提交态或中止态。

在系统崩溃及其后的恢复事件中事务就会从激活态进入到中止态中。

提交 (Committed) 态 应用线程提交请求的结果是激活态的事务进入提交态。没有离开提交态的边。

中止 (Aborted) 态 应用线程请求回退或因为系统的其他原因(如事务死锁或系统崩溃和恢复)都会导致事务由激活态进入中止态。也没有离开中止态的边。

图11-7给出了基本事务状态和它们之间可能的事务转化。要注意的是虽然激活态事务可以转换到提交态或中止态，但是一旦它进入任何一种状态就不能再改变了。当一个事务提交后，它没有能力恢复在激活态所做的数据更新了。这是因为提交以后就会释放锁，这样其他事务就可以看到所做的数据更新。没有了隔离性，就再也不能进行回退了。同样，出于某个原因而引起的中止也会导致锁的释放。最后很重要的点是系统不能保证任何其他站点事务都会进入提交态。如果某激活态事务的站点崩溃了，即使是马上恢复，激活态事务也会对它所做的更新进行UNDO操作，并进入中止态。这种情况下进入中止态的主要原因是系统一般不能恢复足够的上下文来继续成功提交的事务。只有原始的应用程序才可以通过重做 (Redo) 做到这一点。最后要注意的是在我们的模型中没有保证避免在给定站点的崩溃。

基本事务的特性组合使事务间的协同几乎是不可能的。为了说明这一点，假定启动不同站点上的两个基本事务，来完成例11.3.1的分布式TPC-A逻辑。记 T_1 为本地部门站点1的事务，在这里给Branch和Teller账目加上Delta， T_2 为远端事务，那里要给外地部门站点2的账户账目加上Delta。现在假设站点1上的事务管理器充当协调程序，从而调度两个事务以便它们都提交或中止。协调程序会如何工作呢？

对于协调程序一种可能的做法是先提交本地事务 T_1 ，成功发送一个消息来提交远端事务 T_2 。但是如果在收到这一消息时站点2崩溃，则在恢复时会中止事务 T_2 。图11-8演示了这一系列事件。

在站点2恢复之后，提交 T_2 的请求不成功，因为 T_2 已经进入了中止态。由于 T_1 已经提交而 T_2 中止掉，所以银行记录中的钱就会损失掉（因为钱已经通过 T_1 从部门和出纳员那取出但通过 T_2 并没有加入到持有者账号上）。

协调程序第二种可能的做法是先给站点2发一个消息来提交事务 T_2 ，在收到成功返回信息时，再提交 T_1 。这种方法也会出错。在站点2提交可能会成功，但是站点1可能会在收到成功返回信息前就崩溃了。在恢复之后， T_1 中止了而 T_2 提交了，导致外地部门账号账目已经取出而本地部门和出纳员的账目却没有任何变化（假设账号持有者没收到任何现金）。最后协调程序

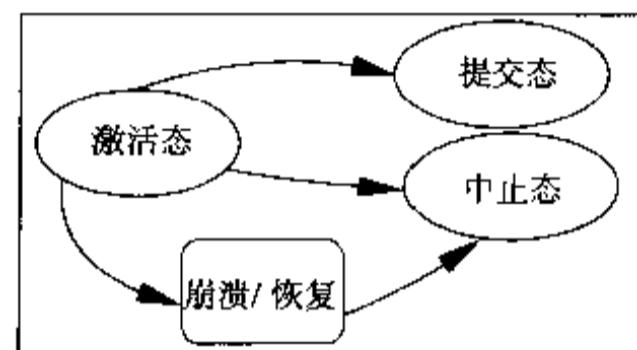


图11-7 基本事务状态转移图

可能会不等待成功信息就尝试同时提交 T_1 和 T_2 ，但是显然还是有可能两个事务中的一个提交成功而另一个因崩溃而进入中止态。

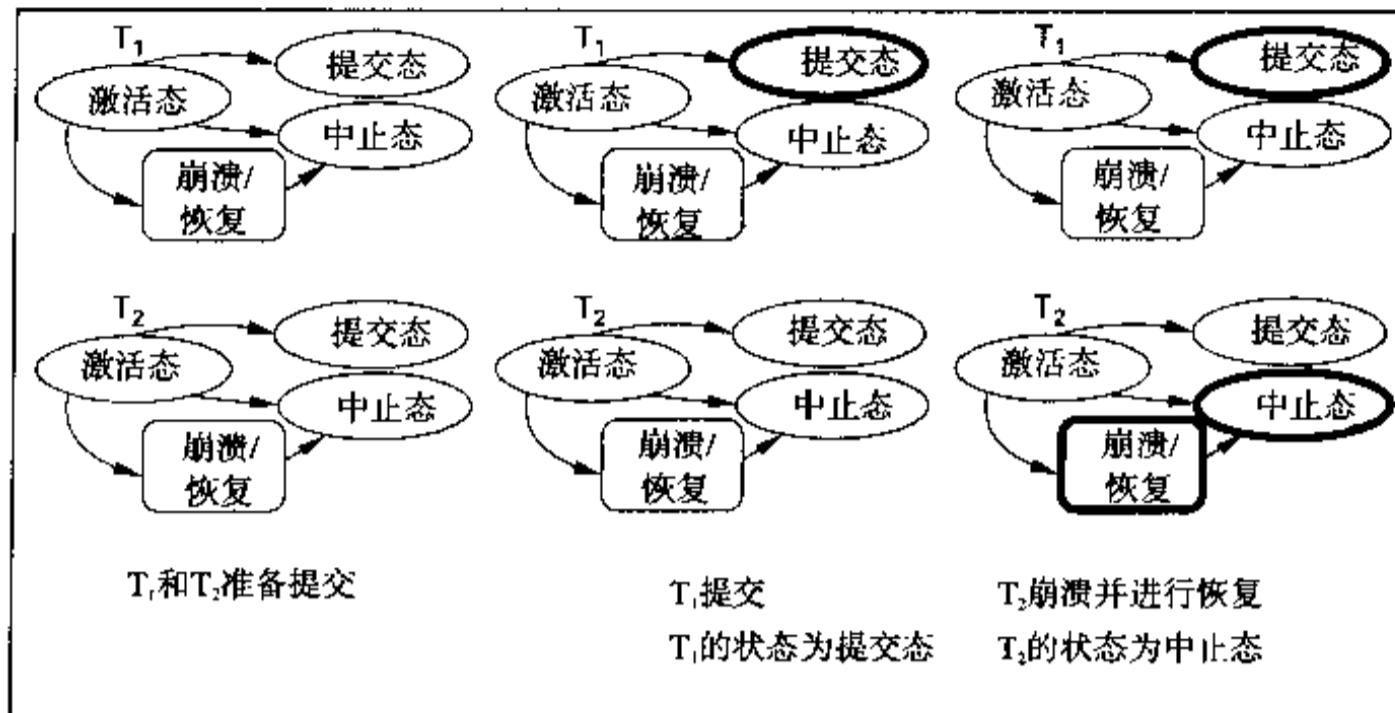


图11-8 分布式事务失败的协调程序

对分布式事务中的任意一个站点都没有办法确定它会在什么时候崩溃。协调程序所面临的主要问题是：一个站点的事务中止而另一个成功提交了，都进入了不可能转移的状态，而这些状态又是相互冲突的。目前情况下进入中止态的主要原因是活动事务的站点崩溃了，在系统恢复时事务就进入了中止态。现在引入准备请求使站点事务可以进入被称为准备态的状态。准备请求可以由协调程序产生。准备态的定义如下：

准备（Prepared）态 分布式事务协调程序发出一个准备请求以后事务就会进入准备态。

事务可以从准备态进入提交态或中止态。在崩溃及其后的恢复中，准备态的事务还会返回准备态。

当一个站点的活动事务收到准备请求时，它就会把当前活动事务的状态保存下来。注意，在开始准备之前事务要做的所有动作都已经做完。现在把准备日志写入日志缓冲区，并把日志缓冲区的内容写入到日志文件中。做完这些以后，我们就称事务进入了准备态。在发生崩溃并进行相应恢复中，事务的当前状态可以重建。从恢复的准备态仍然可以进行提交或中止操作（见图11-9）。为了维护这种灵活性，在站点崩溃以后，需要重做事务的所有更新并要维护完成恢复以后在写入日志文件之前的全部信息。在恢复以后也要重建活动事务保持的所有数据项的锁。

有了新的准备状态，分布式事务协调程序就可以克服参加站点的不可预见的中止问题。在分布式TPC-A事务（记为 T_n ）的例子中，站点1事务为 T_1 ，站点2事务为 T_2 ，站点1的协调程序可以完成如图11-10所示的逻辑。这种方法就称两阶段提交（2PC）协议，这是一种商业数据库系统实现分布式事务协同提交的标准方法。请注意图11-10中PREPARE（ T_2 ）可能出于种种原因而会失败。站点上的事务也可能由于某种原因而中止，如为了打破死锁，并且由站点2

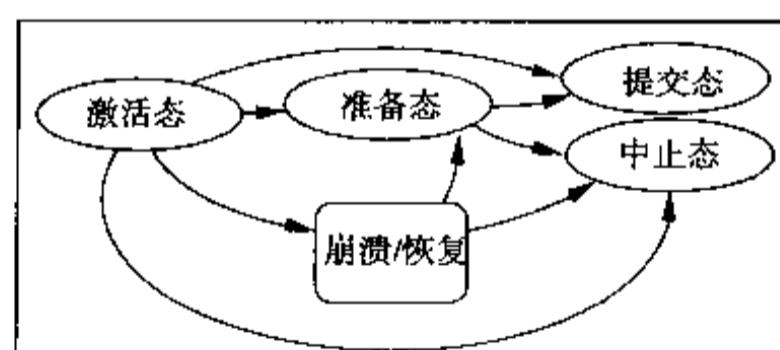


图11-9 带准备态的状态转移图

返回不成功消息。此外站点还可能崩溃或与网络断开连接，这些问题可以与返回不成功信息一样看待。在上述任一状态下假设 T_2 会中止。由于恢复可能会中止活动事务，显然如果系统崩溃，这是很合适的原因。如果网络发生失效，则超时以后就会中止事务。另一方面，如果在站点或网络失效之前准备请求到达了站点2，则可以有一个更持久的状态。由于协调程序没有收到响应，则它会中止 T_1 ，并把ABORT T_0 写入到本地日志文件中，并给站点2发送相应的消息。假设站点2恢复以后可以收到这个消息，则它就会中止 T_2 ，它是 T_0 的一部分。

```

BEGIN
  REQUEST PREPARE( $T_2$ )          /* requires message to site 2 */
  IF RESPONSE IS "UNSUCCESSFUL"  /* site 2 crashed, network failed */
    ROLLBACK  $T_1$ , WRITE "ABORT  $T_0$ "  /* abort distributed transaction */
  ELSE IF ( $T_1$  is "ACTIVE")
    COMMIT( $T_1$ ,  $T_0$ )             /*  $T_1$  hasn't aborted while waiting */
    REQUEST COMMIT( $T_2$ )          /* no going back now */
  ELSE
    WRITE "ABORT  $T_0$ "          /* will succeed eventually */
    REQUEST ABORT( $T_2$ )          /*  $T_1$  has aborted */
  END

```

图11-10 两站点分布式事务的两阶段提交

另一方面，如果站点2返回成功消息，则协调程序就会将本地事务 T_1 和分布式事务 T_0 一起协同提交，并写入到日志文件中。这往往是一般分布式事务协调的弱点。在一个节点提交成功后另一个崩溃，并且在恢复时另一个基本事务已被中止。但是因为 T_2 已经准备过，所以这里不会发生这种情况。如果站点2崩溃，则这只意味着一个延迟，在恢复之后 T_2 仍然可以返回准备状态。在 T_0 与 T_1 提交之后，协调程序会发COMMIT消息到站点2上，在站点2上处理准备态的 T_2 就可以提交成功。如果在 T_0 提交之后站点1崩溃了，恢复之后会给所有相关的站点重发一个COMMIT消息（重复消息没有害处）。图11-11说明了这一系列的事件。

两阶段提交协议同样适用于多个站点。多个站点的2PC协议的框架如下所示。

阶段1：协调程序给分布式事务中所有的所有站点事务发准备请求消息。如果有一个站点返回“不成功”，则协调程序回退本地事务，并发消息给其他站点中止事务执行。

阶段2：如果所有的准备请求结果都是“成功”，则协调程序提交分布式事务 T_0 和本地事务，并给其他站点发提交事务消息。这些消息迟早都会到达的。

2. 无共享式体系结构的进一步问题

两阶段提交协议不是很容易实现的（DB2在实现它时就花了好几年的时间），其中会出现许多难题，现列举一些如下。

分布式死锁 假设分布式事务 T_{01} 更新站点1上的数据项A并试图更新站点2的数据项B，但是发现B已经被加锁了，所以 T_{01} 进入等待状态。现在假设在站点2上给数据项B加锁的分布式事务为 T_{02} ，它也试图更新站点1上的数据项A。显然这是一个死锁。但是如何检测这种死锁呢？请回忆一下现在所面对的站点CPU是无公共内存或不知道外部站点计算机上加锁表中的任何数据项的。没有一个站点可以跟踪涉及到几个站点的等待环。这个问题的最简单的解决方法是放弃死锁检测而使用超时中止方法。当分布式事务在等待外部站点的数据项时，就有可能进入死锁，这样超过特定时间以后，事务中止并重试。很重要的一点是不同的站点选用

不同的超时时间间隔。这种方法可以避免它们一起放弃又同时重试。

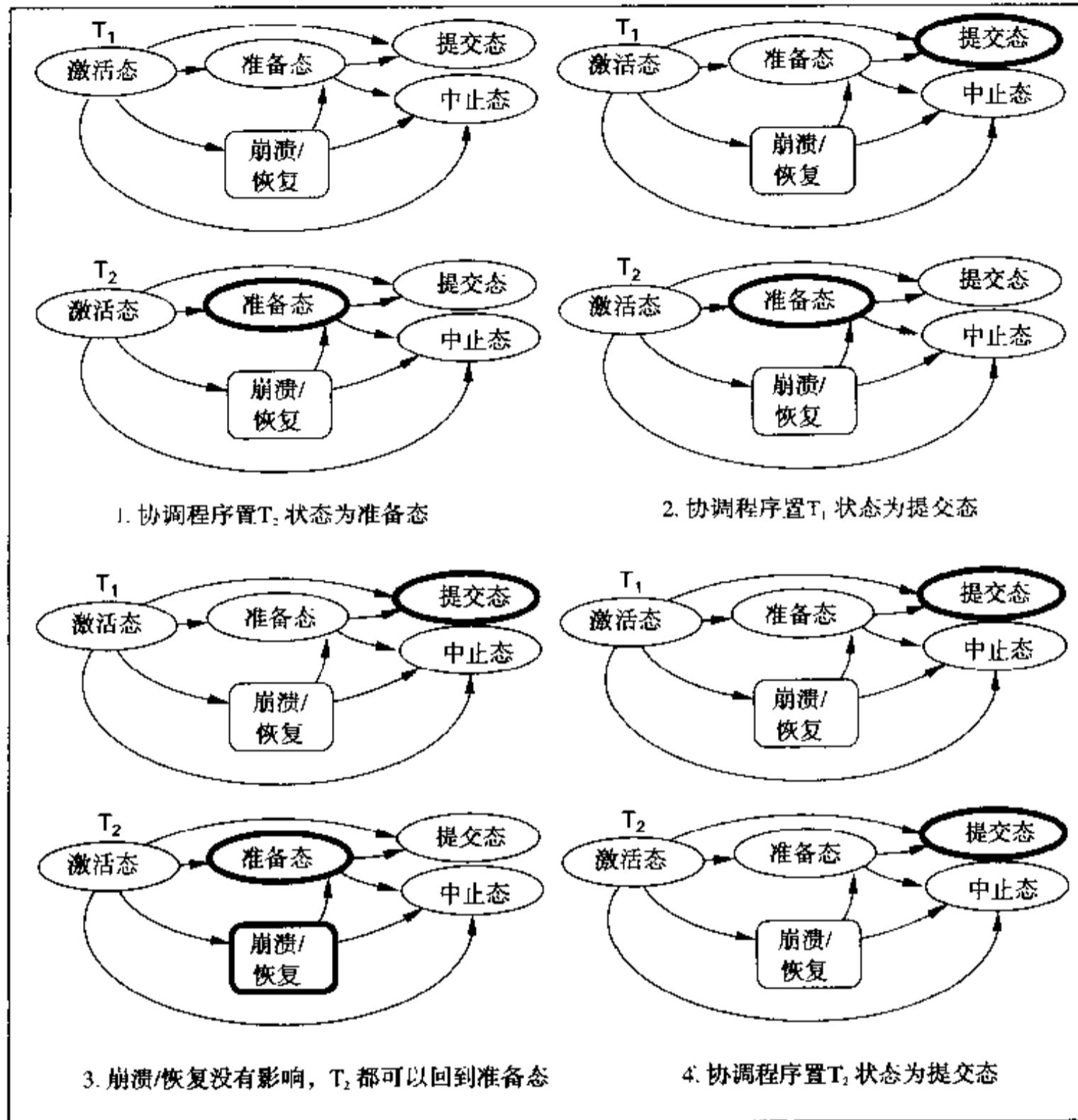


图11-11 两阶段提交的事件顺序

事务阻塞 如果在处理事务之前必须等待其从一个长时间失效中恢复则称事务是阻塞的。由于阻塞的事务可能会给大批数据加锁，所以它会引起很严重的问题。在以下的情况下两阶段提交协议就可能引起死锁。假设站点1的协调程序在给站点2的事务发准备请求以后，而在发最后的提交消息（或由于某种原因中止本地事务而发中止消息）以前就崩溃了。假定任一种结果都是可能的，正因为如此站点2的准备态事务 T_2 在这个协议下就不能自己做出选择。它必须一直等到协调程序恢复以后做出决定，但是这可能要花不少的时间，如果它在PEPSI_12_OZ行上加锁，而如果这数据可能会在一秒内被50个订单所访问，则这样的开销是无法忍受的，目前对此还没有很好的解决方法。另一个叫三阶段提交协议的设计用来在大多数情况下避免事务阻塞，但是为了达到提交它涉及到另外一条消息，所以目前它还没有商业化（请记住通信的开销会抵消使用小的CPU的好处）。

重复数据 著名的研究员Leslie Lamport曾经说过：“我还没有听说过在分布式系统中有某

些计算机失效以后还可以继续完成我的工作。”显然，这问题是对数据进行了划分，如果许多事务需要从多个计算机上获取数据（这不是TPC-A事务的问题，因为它们太简单了），则更容易引起这种错误。如果有一组每月只失效一次的CPU（假设每天8个小时，每周5天工作日），则把100台这样的计算机组成分布式数据系统时，则它们中每1.7个小时就会有一个失效（与大型CPU相比还要考虑的是低档CPU的内部平均出错时间）。为了克服这个问题，标准的方法是复制数据库中的数据，这样数据不是进行简单的划分，而是至少保存在两台机器上。这种做法的缺点是在数据更新时至少要与两台计算机通信。同样有通信的问题并增加了分布式数据库系统中的其他开销。

虽然更新事务有这么多的复杂问题，但是无共享式数据库系统即将到来。经济性和易用性是设计新数据库系统时考虑的重点，目前新的方法还在不断地尝试中。

11.4 查询并行性

直到现在我们主要集中在如何用多CPU体系结构来支持事务更新，并且目前所遇到的大部分困难都围绕着执行更新的需要。但是现在有越来越多的应用系统主要用于查询，很少有在线更新，新旧数据的合并主要也是离线操作的。比如，许多大百货公司一般通过研究过去顾客的购买习惯来决定放置商品的位置和需要对哪些顾客邮件通知。几乎所有的商人都会用已有的数据来决定以后的订单。重新订购主要通过简单地插入操作来实现，而订单的内容主要通过查询来实现。由于市场专家并不会只是简单地重新订购以前订过的东西，所以这里的查询可能会很复杂。仔细分析来决定应订购哪些产品、哪些季节性商品要贮存或如何改变商店格局使商品更加明显。这类系统就叫决策支持系统（DSS）。许多大公司（如药品批发商）把这种分析作为增值服务提供给他们的顾客。主要用于查询的系统还有许多，如图书馆、书店、音像店、警察局许可证查询，等等。

在只用于查询的分布式系统中，显然没有两阶段提交、事务阻塞（没有写锁）或分布式死锁的问题。如果为了避免某些站点失效，可以在几个站点上复制数据，也不用考虑同时更新多个站点的问题。现在体系结构所面临的主要挑战是如何分解一个查询，使其可以在多个不同站点上并行操作，这就是查询内并行性。

查询内并行性

下面给出一个典型的无共享式查询系统。与先前提到的TPC-A不同，要查询的Account表被划分到K个不同的站点上，Account表被划分成：Acct 1在站点1上，Acct 2在站点2上，……，Acct K在节点K上。这些站点机器不共享内存或磁盘，但它们可以通过通信网络发消息来联系。现在假设在站点1要查询老龄客户的账号信息：

```
[11.4.1] select Account.name, Account.phone from Account
          where Account.age >= 65;
```

在现在的体系结构中没有哪个站点有完整的Account表，因此最直接的方法就是把查询[11.4.1]分解为K个不同的查询Q_J，它的形式如下：

```
select AcctJ.name, AcctJ.phone from AcctJ
      where AcctJ.age >= 65;
```

这里J的取值范围为1到K。查询协调程序需要给站点J发查询Q_J请求，各个站点根据本站点的数据返回结果给查询协调程序，这里就是站点1。而站点1收集这些站点的返回信息，拼

凑起来形成结果。

值得注意的是，有时查询协调程序的工作可能很复杂，比如，如果在查询[11.4.1]中附带 ORDER BY 子句，如... order by Account.name。这样各站点上的查询Q都有这个子句，它们结果也是按合适的次序返回。但是查询协调程序仍需要把所有结果重新排序。从先前的查询可以看出，如果只是对早期返回的结果的行集合有一点疑问的话，就有理由限制结果的具体化。这种特征会使协调程序更加复杂。当修改查询[11.4.1]如下时，又会有新的问题出现：

```
[11.4.2] select count(*) from Account where Account.age >= 65;
```

这个查询也要分解为站点部分的查询，然后由协调程序把不同站点的查询结果加起来得到最后的结果。

最难的问题是需要不同站点表连接的查询结果。现在假设个人可以在不同的银行部门有多个账号，但是有一个是主账号。在账号表中有一个字段acct_type来记录是主账号还是从账号。是从账号的数据中有一个列值为Pracct_ID，记录相应的主账号的Account_ID的值（主账号行的Pracct_ID为空）。现在考虑一下多个账号持有者的多个账号的查询：

```
[11.4.3] select A1.Account_ID, A1.balance, A2.Account_ID, A2.balance
      from Account A1, Account A2
     where A2.acct_type = 'secondary'
       and A2.pracct_id = A1.Account_ID;
```

现在考虑一下这个查询如何在我们讨论的多个站点数据库中执行。我们可以将查询对应站点J分段成不同的查询Q_J，J从I到K，这里从账号被定位：

```
[11.4.4] select A1.Account_ID, A1.balance, A2.Account_ID, A2.balance
      from Account A1, AcctJ A2
     where A2.acct_type = 'secondary'
       and A2.Pracct_ID = A1.Account_ID;
```

但是我们注意到主账号的Account表还是需要多站点间连接，在这种环境下跨站点间连接是不可避免的。站点J的查询[11.4.4]的一种可能执行计划是收集AcctJ中所有的Account_ID到集合X中：

```
[11.4.5] select A1.Pracct_ID into X
      from AcctJ A2
     where A2.acct_type = 'secondary';
```

现在把站点J的集合X划分为X₁, X₂, ..., X_K，其中集合X_M包含有与站点M的Accout_ID匹配的Pracct_ID值（系统知道Account表的不同行按主键值贮存在哪里）。现在站点J给不同的站点M发送消息请求查询的结果：

```
[11.4.6] select A1.Account_ID, A2.balance
      from AccM A1
     where A1.Account_ID in XM;
```

当不同站点M都把查询结果返回给站点J后，连接一下就是查询[11.4.4]的结果。所有形如[11.4.4]的不同查询的结果返回后就是查询[11.4.3]的结果。

多站点无共享式数据库的查询在涉及到多层站点内消息时就会变得非常复杂。显然由于

这种方法涉及到了不同站点间的数据流，所以在广域网上只有低速通信能力的分布式节点集合是有很大缺陷的。无共享式查询的一个详细信息可参考 DeWitt和Gray的论文推荐读物(推荐读物[2])。

推荐读物

分布式数据库系统设计所涉及到的不同问题在Ozsu和Valduriez 的文章中有很好的表述(推荐读物[7])。Bernstein、Hadzilacos和Goodman的文章(推荐读物[1])对分布式事务做了一个很严格的介绍，而Gray和Reuter在设计与实现方面做了很好的简介(推荐读物[5])，他们的文章对TP监视器也做了很好的介绍。Khosafian、Chan、Wong和Wong的文章(推荐读物[6])介绍了多个系统下用SQL编写客户机-服务器应用程序的一些基本概念。并行查询问题在DeWitt和Gray的论文(推荐读物[2])中进行了讨论，推荐读物[4]《*The Benchmark Handbook*》主要处理了Wisconsin测试。Stonebraker的一些论文(推荐读物[8])也是有关分布式数据库系统的。

- [1] P.A.Bernstein, V.Hadzilacos, and N.Goodman. *Concurrency Control and Recovery in Database Systems*. Reading, MA:Addison-Wesley, 1987.
- [2] D.J. DeWitt and J.Gray. “Parallel Database Systems: The Future of High-Performance Database Systems.” *Comm. of the ACM*, 35(2), June 1992, p.85.
- [3] Goetz Graefe. “Query Evaluation Techniques for Large Databases.” *ACM Computer Surveys*, 25(2), June 1993, pp.73-170.
- [4] Jim Gray, editor. *The Benchmark Handbook for Database and Transaction Processing Systems*. 2nd ed. San Mateo, CA: Morgan Kaufmann, 1993. See Section 4.3, “Benchmarking Parallel Database Systems Using the Wisconsin Benchmark.”
- [5] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. San Mateo, CA:Morgan Kaufmann, 1993.
- [6] Setrag Khosafian, Arvola Chan, Anna Wong, and Harry K.T.Wong. *A Guide to Developing Client/Server SQL Applications*. San Mateo, CA: Morgan Kaufmann, 1992.
- [7] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Englewood Cliffs, NJ: Prentice Hall, 1991.
- [8] Michael Stonebraker, editor, *Readings in Database Systems*. 2nd ed. San Francisco: Morgan Kaufmann, 1994. See original research papers: Chapter 7, “Distributed Database Systems” and Chapter 8, “Parallelism in Database Systems.”

习题

- 11.1 对以下问题回答对或错，并给出理由。
- (a) 在共享内存式多处理器系统中为保证协同事务，两阶段提交是必须的。
 - (b) 在只处理查询的无共享式DSS系统中分布式死锁是不可能的。
 - (c) 在客户机-服务器体系结构下服务器不能是无共享式并行数据库系统。
 - (d) 根据图11-6，五个60MIPS的CPU比一个300MIPS的CPU便宜。
 - (e) 一个准备态的事务在它所在的站点机器崩溃而后又恢复之后仍然可以提交。

附录A 教学指导

A.1 在ORACLE中安装CAP数据库

本节解释完成第3章中练习3.2.1所需要的技能。练习SQL数据定义语句来创建数据库表，以复制图2-2Customers-agents-products (CAP) 数据库的上机作业。通过本部分的学习，你将明白如何从操作系统进入数据库、如何把数据从操作系统的文件中装入数据库以及如何操作SQL*Plus交互式环境。SQL*Plus有很多命令可以让你编写SQL语句、编辑它们、把它们保存到操作系统的文件中去、再把它们从文件中读入等。像在3.2节中介绍的那样，你首先需要获得一个操作系统的账号——一个用来登录到计算机的操作系统用户ID和口令——和一个用来进入数据库的ORACLE名和口令。在你安装时，你的指导老师或DBA应该可以给你提供指导。

在ORACLE中，一般都只有一个属于数据库管理员的数据库。和在INFORMIX中不一样，ORACLE的单个的用户一般并不创建他们自己的数据库。为了保证不同用户的数据能够彼此区分，数据库管理员授予每个用户访问一个数据库中的私有表空间 (tablespace) 的权利。一个表空间就像一个包含表集合的目录。我们仍将非正式地称一组相关的表为一个“数据库”，例如图2-2中的CAP数据库。

1. 创建CAP数据库

为了进入ORACLE，输入如下操作系统命令：

[A.1.1] sqlplus

ORACLE会提示你输入数据库用户名和口令（此时，DBA应该已经给了你数据库用户名和口令）。如果你的用户名和口令被接受了，那么你将看到一个新的提示符：

SQL>

这意味着你现在正在SQL*Plus交互式环境下。此环境下的很多命令（主要是编辑命令）将在下一节介绍。现在，我们集中讨论你将要创建的SQL语句。这些SQL语句是真正对ORACLE系统的指令（而不是针对交互式环境的，交互式环境仅仅是用来编写这些语句的）。它们使你可以创建表、载入表、进行查询等。按照在3.2节中的介绍，在ORACLE中创建CAP数据库的CUSTOMERS表，使用小写的表名customers。你可以使用以下语句来创建表：

```
SQL> create table customers (cid char(4) not null, cname varchar(13),
2 city varchar(20), discnt real, primary key (cid));
```

注意，在第一行按了Enter键以后，系统显示一个第二行的提示符：2。第三行的提示符是3，以此类推。你可以输入的行的数目没有限制。在你用分号（;）作为某行的结束之前，系统不会试图解释你的输入。

Create Table语句的结果是创建了一张带列名（或称为属性）cid, cname, city和discnt的空customers表。每个属性的类型（又称为属性的域）在每个属性名后给定。于是，cname的类型是varchar(13)（意味着长度最多为13个字符的变长字符串），而discnt的类型

是real。请参考3.2节以获得更多的内容，并请参考A.3节“数据类型”以得到ORACLE数据类型的完整列表。ORACLE支持ANSI SQL数据类型名integer（整数）、real（实数）、double precision（双精度）等。这些数据类型和它自己的数据类型（number(n)等）一起都满足最基本的范围和精度要求。我们将使用ANSI SQL数据类型名来代替ORACLE专用的名称来使SQL语句尽可能易于移植。

现在，我们已经有了一张表，我们可以向其中装入数据。为了做到这一点，我们通过在SQL提示符下键入exit退出SQL*Plus，然后运行SQL*Loader工具。

2. 使用SQL*Loader

SQL*Loader是从操作系统文本文件读数据并把内容转化为表中的字段的工具程序。为了做到这一点，它必须被告知要装入的外部数据的格式。此描述存放于控制文件中，控制文件的文件名后缀一般为“.ctl”。例如，控制文件custs.ctl是这样的：

```
load data
replace
into table customers
fields terminated by ","
(cid, cname, city, discnt)
```

数据文件一般后缀为.dat。它包含要装入到表中去的数据。它的格式一定要和控制文件中的描述一致。例如，custs.dat是这样的：

```
c001,Tiptop,Duluth,10.0
c002,Basics,Dallas,12.0
c003,Allied,Dallas,8.00
...
```

为了运行载入工具，输入以下操作系统命令：

```
sqlload control=custs.ctl
```

接着SQL*Loader就会提示你输入ORACLE用户名和口令。如果你愿意，你可以在命令行中输入用户名和口令：

```
sqlload username/password control=custs.ctl
```

注意，此命令假设在输入此命令时custs.ctl在当前目录下。否则就需要输入更完整的路径名。例如，在UNIX中，文件custs.ctl在poneil的用户目录的oracle子目录下，那么，在Copy命令中，将用/usr/poneil/oracle/custs.ctl来表示该文件。不管是上述哪种情况，SQL*Loader都要有对包含控制文件的目录的写权限，这样它才能建立日志文件（log file）和出错文件（bad file）。日志文件的扩展名为.log，它包含了装入过程的详细报告。出错文件的扩展名为.bad，它包含了不能被正确读入的记录。在你使用custs.ctl和custs.dat前，你需要将它们从指导老师的主页或者本书的主页拷贝到你自己的目录中去；如果企图在指导老师或者DBA的目录中创建日志文件和出错文件中，sqlload将失效，这是因为你对于该目录只有读权限。

你现在已经知道如何创建表并装入数据了，你还需要知道如何“销毁”表（就像在目录中删除文件那样，把表从数据库中删除）。像3.2节介绍的那样，为此我们使用命令：

[A.1.2] drop table tablename;

所有的表可以有一个包含语句的文件来创建，这里我们假设它叫做create.sql。我们把它提供给SQL*Plus，如下：

```
sqlplus username/password @create.sql
```

文件create.sql有以下内容。（注意，如果在数据库中有以前版本的表，在创建表之前要删除它们。）双连字号表示该行剩下的内容是注释，为不执行的内容。ORACLE也允许C语言风格的注释语法/*...*/，但这不是ANSI标准，不能移植到其他数据库系统上去。

```
-- create.sql: SQL script file for table creation

drop table customers;
create table customers (cid char(4) not null, cname varchar(13),
    city varchar(20), discnt real,
    primary key (cid));

drop table agents;
create table agents (aid char(3) not null, aname varchar(13),
    city varchar(20), percent smallint,
    primary key (aid));

drop table products;
create table products (pid char(3) not null, pname varchar(13),
    city varchar(20), quantity integer, price double precision,
    primary key (pid));

drop table orders;
create table orders (ordno integer not null, month char(3),
    cid char(4), aid char(3), pid char(3),
    qty integer, dollars double precision,
    primary key (ordno));
```

这里，我们使用的是SQL-92的数据类型，而不是由于历史原因更广泛地被使用ORACLE所特有的类型。一个受过ORACLE训练的DBA会为整型的数量属性使用number(6)或者number(10)，而为货币属性使用number(10,2)，并为双精度属性使用float。而在这里，我们为货币属性使用double precision。但是numeric(10,2)和decimal(10,2)这些SQL标准数据类型也常常被用来表示货币属性。这样做的好处是正好可以得到“分”，而以“元”为单位。而如果使用double precision，只有当最小单位是“分”的时候，才能准确到“分”。这是因为1/100不是二进制有限分数。

在上面的命令执行以后，那些表就可以在操作系统命令中使用sqlload装入了：

```
sqlload username/password control=custs.ctl
sqlload username/password control=agents.ctl
sqlload username/password control=prods.ctl
sqlload username/password control=orders.ctl
```

为了让该工作全自动地完成，把sqlplus命令和sqlload命令放在一个命令文件中(例如，Windows NT或其他系统中的loadcap.cmd，或者UNIX中的loadcap)于是，在命令行中运行loadcap就可以完成所有的工作了。

3. 使用SQL*Plus

SQL*Plus就是你在操作系统下输入sqlplus后进入的环境。一开始，你会看到提示符：

SQL> _

你可以输入你想输入的任何SQL语句。如果该语句在一行中放不下，你可以继续使用下一行，SQL*Plus会在每个新行的头上显示行号作为提示符。例如：

```
SQL> select * from customers
      2 where cname = 'Tiptop' _
```

SQL*Plus不会试图翻译此语句，除非你输入分号作为结束：

```
SQL> select * from customers
      2 where cname = 'Tiptop'; _
```

在第二行中按了Enter键以后，SQL*Plus显示出和查询相匹配的行，并返回给你SQL>提示符。也可以输入一条命令而不执行它，该特性的用途将稍候再介绍。为了中止一条命令而不执行它，你只要输入一个空行，然后，你就回到SQL>提示符下。命令的内容（无论是否执行了）存放在缓冲区中。通过输入命令l或者list，我们可以看到缓冲区的内容。

```
SQL> l
      1 select * from customers
      2* where cname='Tiptop'
SQL> _
```

处理缓冲的SQL*Plus命令，例如l，并不存放于缓冲区中；否则它将严重地限制该命令的用途。

在上面的例子中，2后面的星号表示缓冲中的当前行，它将会被多条SQL*Plus命令使用。要改变当前行，只要在SQL提示符下输入行号就可以了。

```
SQL> 1
      1* select * from customers
SQL> _
```

为了修改当前行中的一小部分，你可以用命令c或者change，这样就不需要重新输入整个命令了。该命令用新的文字序列替换掉旧的序列。例如：

```
SQL> c /customers/agents
      1* select * from agents
SQL> _
```

其中，斜杠 (/) 被称为分隔字符。你可以使用任何非字母数字的字符作为分隔字符。例如，命令c &customers&agents和上述命令是等价的。如果要删除当前行的一个字符串，只要简单地用一个空串代替就可以了。

```
SQL> c /agents/
      1* select * from
SQL> _
```

如果要替换整行的内容，只要输入行号，然后输入新的内容就可以了。

```
SQL> 1 select * from customers
```

要重新执行（可能修改过的）缓冲区中的内容，可以使用命令r；也可以使用命令l列出

缓冲区的内容，并执行它们。

你可以使用命令`del`删除缓冲区中的当前行。你还可以用命令`i`在当前行后插入一行。此命令有两种使用方式，你可以插入在命令`i`后所跟的单行内容：

```
SQL> i where cid = 'c001'
SQL> _
```

或者使用不带参数的`i`命令来输入多行：

```
SQL> i
  2 where cid
  3 = 'c001';
```

在这种方式下，SQL*Plus的表现就像你在前面的命令后继续输入一样（或是在一个很大的缓冲区的某一行后插入）。如这个例子所示，结束的分号使得整个缓冲区中的内容被执行。

现在，我们使用这些命令来修改一个典型的拼写错误。假设输入了：

```
SQL> select * from customers
  2 where cname = 'Tiptop' _
```

而后意识到拼写错误，你可以通过结束命令（输入一个空行）、选择要修改的行、用正确的单词代替拼错的单词并执行缓冲区内容来纠正它。

```
SQL> select * from customers
  2 where cname = 'Tiptop'
  3
SQL> 1
  1* select * from customers
SQL> c /sel1/sel
  1* select * from customers
SQL> /
```

这样做的效果和重新输入命令是一样的，但是这样做击键次数较少。

使用`edit`命令可以调用你的系统编辑器。如果不用参数，它将让编辑器编辑缓冲区中的内容。你也可以指定要编辑的文件名称。

为了要在SQL*Plus环境下执行SQL命令文件，需要使用带命令文件名的`@`命令。如果仅仅要把命令文件装入缓冲区，可以使用带文件名的`get`命令。如果要把缓冲区中的内容存入文件，可以使用带文件名的`sav`或`save`命令。

作为总结，这里给出SQL*Plus环境下可用的命令列表：

<code>c /old/new</code>	在当前行中把第一次出现的旧字符串修改为新字符串。
<code>l</code>	列出缓冲区中的内容。
<code>del</code>	从缓冲区中删除当前行。
<code>i new line</code>	在缓冲区中当前行后插入一新行。
<code>i</code>	开始在缓冲区中当前行后交互式地插入新行。
<code>/</code>	执行缓冲区中的语句。
<code>r</code>	列出并执行缓冲区中的语句。
<code>edit filename</code>	使用缺省的操作系统编辑器编辑文件。
<code>edit</code>	使用缺省的操作系统编辑器编辑缓冲区。

@filename	执行在filename文件中存放的SQL语句。
get filename	把filename文件中的内容装入缓冲区。
save filename	把缓冲区存放到指定的文件中去(覆盖掉原来的文件)。
5 new line	替换第5行。
5	设置当前行为第5行。
host command	执行指定的操作系统命令。
spool filename	开始把用户交互(用户击键和输出)存入指定的文件。
spool off	结束上一个spool命令开始的存放用户交互行为，并关闭文件。
exit	退出SQL*Plus环境。

如果edit命令无效，那么你就需要执行SQL*Plus命令define_editor='editor_name'。可以使用自动的启动文件来完成此设置，但是那已经超出了此总结的范围了。

A.2 在INFORMIX中安装CAP数据库

本节介绍完成第3章中练习3.2.1所需要的技能。练习SQL数据定义语句来创建数据库表，以复制图2-2中CAP数据库的上机作业。通过本部分的学习，你将明白如何从操作系统进入数据库、如何把数据从操作系统的文件中装入数据库以及如何执行SQL语句、编辑它们、把它们保存到操作系统的文件中去、把它们从文件中读入等。像在3.2节中介绍的那样，你首先需要获得一个操作系统的账号——一个用来登录到计算机的操作系统用户ID和口令。安装时，你的指导老师或DBA应该可以给你提供指导。此外还需要设置一些环境变量，例如对UNIX用户需要设置INFORMIXDIR。注意下面所说的INFORMIX交互式菜单环境只适用于UNIX操作系统。Windows操作系统使用的是另一种不同的界面：sqleditor工具。我们准备将来对Windows界面编写另外一个教程，把它放在网页<http://www.cs.umb.edu/~poneil/dbppp.html>上。

INFORMIX和ORACLE不一样，用户一般都可以创建自己的数据库。创建CAP数据库包括建立自己的数据库，然后在数据库中创建CAP的表。如果你在使用UNIX系统，为了在你创建数据库前进入INFORMIX交互式环境，输入操作系统命令dbaccess。

1. 创建CAP数据库(在UNIX中)

在UNIX中执行dbaccess命令以后，你将看到只有一行的菜单：

[A.2.1] DBACCESS: Query-language Connection Database Table Session Exit

通过键入d来选择Database选项，或者使用左箭头使Database高亮显示，然后按Return键。下一个菜单是：

[A.2.2] DATABASE: Select Create Info Drop Close Exit

通过键入c来选择Create选项，然后根据要求输入数据库名称，此名称可以根据你的用户名决定，或者由你的指导老师决定，例如eoneildb。接着，当要求决定Dbspace的时候，键入e以选择缺省值退出。会有一个缺省选择为“Create-new-database”的菜单要求你确认创建数据库的行为，按Enter键选择缺省值，然后键入e以退出Database菜单。如果你还想退出Dbaccess，那么就再键入一个e。如果你在任何时候忘了自己到底选择了什么选项或在菜单层次中陷入循环的话，你可以通过按Ctrl-W来获得帮助。连续地按e退出并不一定能退出循环，尽管这应该是你首先尝试的。

现在，你已经有了一个数据库了。下次执行dbaccess，你应该在命令行中提供数据库名，即dbaccess eoneildb，你将会看到[A.2.1]中那样的单行菜单。通过按Enter键选择第一个加亮的选项Query-language就进入了下一个菜单：

[A.2.3] SQL: New Run Modify Use-editor Output Choose Save Info Drop Exit

用相同办法选择New来开始一个新的SQL Editor会话以编写SQL语句。在此环境下，你可以为CAP数据库创建表。首先，输入SQL语句：

```
create table test1(coll int);
```

屏幕的顶端显示如下：

```
NEW:    ESC - Done editing      CTRl-A - Typeover/Insert      CNTL-R - Redraw
       CNTL-X - Delete character   CTRl-D - Delete rest of line
-----eoneildb@dbserver-----Press CTRl-W for Help-----
```

```
create table test1(coll int);
```

按Esc键从编辑状态回到高层菜单[A.2.3]，然后选择Run执行语句。你应该在屏幕底部看到“Table created”或者类似的输出。

如果你的Run命令失败了，你就需要再编辑你的SQL语句然后重试。选择Query-language菜单中的Modify，你就可以回到和选择New所进入的一样的SQL Editor环境。但是现在，你原来输入的文字还在屏幕的底部。你可以向前面一样编辑文字，并使用Esc和Enter来执行语句。

为了删除测试用的表，选择New，然后键入SQL语句：

```
drop table test1;
```

然后用前面介绍的办法来执行该语句。

现在，你已经知道如何在UNIX中使用dbaccess执行SQL语句了（或者已经掌握了在Windows中使用sqleditor来完成相应的工作）。下面我们把UNIX和Windows环境的内容一起讨论。像在3.2节中介绍的那样，在CAP数据库中创建customers表，你可以执行以下SQL语句：

```
create table customers (cid char(4) not null, cname varchar(13),
city varchar(20), discnt real, primary key(cid));
```

Create Table语句的执行结果是创建了一个带列名（或叫做属性）cid、cname、city和discnt的空customers表。每个属性的类型（又叫做属性的域）在各个属性名后指定。这样，cname的类型是varchar(13)（意思是最大长度是13个字符的变长字符串），而discnt的类型是real。请参考3.2节以获得更进一步的说明，并请参考A.3节“数据类型”以获得INFORMIX数据类型的列表。我们将使用SQL标准中的数据类型的名称，INFORMIX完全支持这些类型。

现在我们已经创建了一个表，我们可以把数据文件中的数据装入表中。数据文件的扩展名一般为.dat。例如，客户数据文件custs.dat的内容如下：

```
c001,Tiptop,Duluth,10.0
c002,Basics,Dallas,12.0
c003,Allied,Dallas,8.00
...
```

为了载入，我们可以使用INFORMIX加入到其SQL语言中去的Load语句。例如，为了把

custs.dat 中的数据装入数据库表 *customers* 中去，我们可以使用如下 SQL 语句（使用和执行 *Create Table* 语句相同的方法在 SQL Editor 中输入语句）：

```
load from 'custs.dat' delimiter '.' insert into customers;
```

注意，此命令假设 *custs.dat* 在输入命令时在当前目录下。否则，就需要提供更完整的目录路径名。例如，在 UNIX 中，文件 *custs.dat* 在 *poneil* 的用户目录的 *oracle* 子目录下，那么在 Load 命令中，就要被表示成 /usr/poneil/oracle/custs.dat。

现在，你已经知道如何创建表以及如何装入数据，你还需要知道如何“销毁”表（就像在目录中删除文件那样把表从数据库中删除）。正如在 3.2 节中介绍的那样，为此目的，我们可以使用我们已经碰到过的如下命令：

[A.2.4] drop table tablename;

所有的表都可以使用一个包含 SQL 语句的文件来创建。例如，我们将文件 *create.sql* 作为 UNIX 下 dbaccess（或 Windows 下的 SQL Editor）的输入以及 Windows 下的 SQL Editor 如下：

```
dbaccess eoneildb create.sql    (UNIX systems, using local server)
sqleditor /s csdb /d eoneildb create.sql  (Windows systems, using server csdb)
```

文件 *create.sql* 包含以下内容。（注意我们在创建它们之前先删除了表，以防止在数据库中有以前版本的表存在。）双连字号表示该行的内容是注释，即不需执行的文字。INFORMIX 还允许用大括号括起来的注释：{...}，但是这不是 ANSI 标准，所以不能移植到其他数据库系统上去。下面的 SQL 语句是标准的 SQL（SQL-92 和 X/Open）。

```
-- create.sql: SQL script file for table creation
drop table customers;
create table customers (cid char(4) not null, cname varchar(13),
    city varchar(20), discnt real,
    primary key (cid));

drop table agents;
create table agents (aid char(3) not null, aname varchar(13),
    city varchar(20), percent smallint,
    primary key (aid));

drop table products;
create table products (pid char(3) not null, pname varchar(13),
    city varchar(20), quantity integer, price double precision,
    primary key (pid));

drop table orders;
create table orders (ordno integer not null, month char(3),
    cid char(4), aid char(3), pid char(3),
    qty integer, dollars double precision,
    primary key (ordno));
```

在执行以上的命令之后，就可以使用以下名为 *infload.sql*（“Informix load”的简称）的使用产品特有的扩展的 SQL 脚本来将表装入。

```
-- load rows from text files into tables already created by create.sql
-- Note that the "load" statement is an INFORMIX extension to SQL
load from 'custs.dat' delimiter ',' insert into customers;
load from 'agents.dat' delimiter ',' insert into agents;
load from 'prods.dat' delimiter ',' insert into products;
load from 'orders.dat' delimiter ',' insert into orders;
```

然后用运行create.sql的相同办法运行此SQL脚本。为了使此过程自动化，你可以在命令文件（如Windows NT中的loadcap.cmd文件或UNIX中的loadcap文件）中放入两个脚本文件执行命令，然后简单的命令行Loadcap就可以完成所有的工作。你甚至还可以把Create Database语句也放在脚本文件中。此时，你需要在dbaccess后用一个“-”（连字符）来代替数据库名：dbaccess - createall.sql。如果你希望看到命令执行的回应，可以使用-e选项，即dbaccess-e-createall.sql。

2. 使用DB-Access (UNIX系统)

DB-Access是在操作系统环境下输入dbaccess dbname后进入的环境。进入该环境后，你将看到一个称为主菜单的单行菜单：

```
DBACCESS: Query-language Connection Database Table Session Exit
```

我们已经知道了如何使用Database菜单项来创建数据库。一旦数据库创建成功，Query-language菜单项就成为到目前为止最重要的一项了。因为这一项在列表中的第一个，所以只要按Enter键就可以进入Query-language子菜单了。（也可以使用命令dbaccess dbname -q跳过主菜单直接进入Query-language子菜单。）然后你将看到Query-language菜单屏幕，其中的选项如下：

```
SOL: New Run Modify Use-editor Output Choose Save Info Drop Exit
```

在此菜单中选择New以开始一个新的编辑会话（SQL Editor）来编写SQL查询。同时，你也可以使用Use-editor来选择系统编辑器（vi、emacs等）。

在SQL Editor中，你可以输入任何SQL语句。如果语句超过一行，你可以在到达右端前的任何地方敲Enter以开始新的一行。注意，SQL Editor并不会自动换行。如果输错了，你可以使用Enter和方向键重定位光标，然后覆盖错误的内容。你也可以使用Ctrl+A切换到“插入模式”。此时，后面的内容会被“推开”，以容纳新输入的字符。再按一次Ctrl+A会使系统回到“改写模式”。输入两行Select语句以后的屏幕如下：

```
NEW:   ESC  - Done editing      CTRL-A = Typeover/Insert      CNTL-R = Redraw
          CNTL-X = Delete character    CTRL-D = Delete rest of line
-----eoneildb@dbserver-----Press CTRl-W for Help-----
select * from customers
where cname = 'Tiptop';
```

现在，按Esc键以回到Query-language菜单，在这个菜单中你可以使用Run来执行这条语句，或者使用Modify回到编辑会话，或者切换为使用vi这样的系统编辑器。你可以选择Save并输入文件名来保存输入的文本。如果你的输入为mytest，那么文件系统中的文件名为mytest.sql，这是因为dbaccess会自动加上所预期的.sql扩展名。

对于大量的SQL语句，使用系统编辑器比使用dbaccess中的内嵌的简单的编辑器更合适。一旦使用Save或者其他编辑器得到了一个SQL语句的文件，比如说mytest.sql，它就

可以在UNIX命令行下运行了。

```
dbaccess eoneildb mytest.sql
```

或者，你可以在Query-language菜单中使用Choose把mytest.sql的内容装入dbaccess的内存，在那儿你可以执行、修改或用其他编辑器编辑它，最后重新保存它。

作为总结，以下列出了Query-language菜单下的菜单命令：

New	使用SQL Editor输入新的SQL语句。
Run	运行当前的SQL语句。
Modify	使用SQL Editor修改当前的SQL语句。
Use-editor	使用系统编辑器修改当前的SQL语句。
Output	把当前SQL语句的执行结果发送到打印机、文件或者管道。
Choose	选择一个包含SQL语句的文件，并使这些语句成为当前语句。
Save	把当前的SQL语句保存为文件，以便以后可以再次使用它们。
Info	显示当前数据库中表的信息。
Drop	删除一个包含SQL语句的文件。
Exit	回到DB-Access主菜单。

A.3 数据类型

Create Table语句中每一列可用的数据类型如图A-1所示。

SQL-99	ORACLE	INFORMIX	DB2 UDB	范围	C
char(n), 没有指定n 的限制	char(n), n <= 4000	char(n), n <= 32 767	char(n), n <= 254	1 ≤ n ≤ 254, X/Open	char array[n+1]
varchar(n), 没有指定n 的限制	varchar(n) varchar2(n), n <= 4000	varchar(n), n <= 255	varchar(n), n <= 32 672	1 ≤ n ≤ 254, X/Open	char array[n+1]
numeric(6, 0), decimal(6, 0)	numeric(6, 0), number(6)	numeric(6, 0), decimal(6, 0)	numeric(6, 0), decimal(6, 0)	-10 ⁶ < x ≤ 10 ⁶ - 1	short int, (大约)
numeric(p, 2), decimal(p, 2)	numeric(p, 2), decimal(p, 2), number(p, 2)	numeric(p, 2), decimal(p, 2), money(p, 2)	numeric(p, 2), decimal(p, 2)	p位数字，小数点 后2位	无
smallint	smallint	smallint	smallint	-2 ¹⁵ ≤ x ≤ 2 ¹⁵ - 1	short int
integer	integer	integer	integer	-2 ³¹ ≤ x ≤ 2 ³¹ - 1	int, long int
real	real	real, smallfloat	real	-10 ⁻³⁸ ≤ x ≤ 10 ³⁸ 7位精度	float
double precision, float, float(bp)	double precision, number, float, float(bp)	double precision, float	double precision, double, float, float(bp)	-10 ⁻³⁸ ≤ x ≤ 10 ³⁸ 15位精度，或者 bp位精度	double

图A-1 一些重要的SQL数据类型

char(n)数据类型是包含了定长为n个字符的字符串。于是，在[3.2.1]中cid列的数据类型被定义为char(4)表示“c001”这样的字符串正好可以放入，而“c1”（在我们的表中并没有这样的值）的最后将会用空格填充，即c1ΔΔ(其中Δ代表空格字符)。对于串长度变化很大的列，

使用数据类型varchar(n)可以节省存储空间。例如，city属性的数据类型为varchar(20)。它允许最大长度为20个字符的字符串。这样，短字符串就不需要很长的空格串作为结束。于是，像“Troy”、“Austin”这样的短城市名占用比“Oklahoma City”或“San Francisco”更少的存储空间。我们使用smallint类型而不使用integer类型也是出于同样的考虑：当整数值的绝对值不超过 $2^{15}=32\,767$ 的时候会节约存储空间。

SQL-92、SQL-99和各个数据库产品都有各自在图A-1所列内容以外的数据类型。SQL-92有data、time、timestamp、(time) interval、national character strings等类型。SQL-99（在其核心以外的命名特性中）有LARGE OBJECT或简称为LOB（包括BLOB，二进制大对象）和Boolean类型，以及对象关系、用户定义、结构类型和数组。Core SQL-99有distinct类型，即用户为内部类型指定的同义词。在INFORMIX中Ivarchar类型允许最大长度为32 767，而ORACLE的long类型支持最大长度为2GB的字符串。ORACLE使用动态大小的number(38)和number（即双精度（double precision））来实现更小的SQL数据类型smallint和real。以上的符点数类型的范围和精度是简化了的，要知道详细的内容请参考手册。

附录B 编程细节

B.1 prompt() 函数

在此附录中，我们将解释在第5章的很多例子中都用到的prompt()函数，此函数用来提示与用户的交互。请参考图5-1后的“提示用户交互”小节以获得在用户只输入一项的最简单的情况下该函数的使用方法。一般情况下，此提示函数输出作为第一个参数给定的提示字符串，然后从用户处输入一行文本，即用户输入的直到行尾（C语言中的'\n'）（包括行尾）的所有内容。该函数解析该行文本为由空白符分隔的标记。空白符和标记的定义如下。

空白符包括一个或者多个连续的空格、跳格符或换行符。在C语言中，它们一般被相应地表示成：' '、'\t'以及'\n'。标记就是不包含空白符的字符串。一个字符串中的多个标记由空白符分隔。输入的每一行之间由换行符分隔。于是，输入的一行中的多个标记是由不包括换行符的空白符分隔的。由于允许用户在输入换行符前编辑其输入的行的内容，所以对于像prompt()函数这样的提示输入函数，输入行作为处理单元是很自然的。

prompt()函数能够提示并从用户的输入中读入在一行中的指定数目的标记。提示函数的声明在图B-1的头文件prompt.h中。它使用了带省略号（...）的特殊形式来支持不定数目的参数。你仍然可以把它看成如下形式的函数：

```
int prompt(char prompt_str[], int N, buf1, len1, buf2, len2, ..., bufN, lenN);
```

第一个参数prompt_str是显示给用户的提示，第二个参数N是要从用户输入的行中读入的标记的个数。对接下来的个数不定的参数对——buf1, len1, buf2, len2, ..., bufN, lenN——解释如下：bufK（K从1到N）是存放第K个标记的字符数组，而lenK是第K个标记对应的字符数组所允许的最大长度。注意，所有的标记都会被prompt()函数读入字符数组。

```
/* prompt.h: prompted input of one or more tokens on one line of input */
int prompt(char prompt[], int ntokens, ...);
```

图B-1 提示头文件prompt.h

如果有些prompt()函数读入的字符串变量应该被解释为int或float型的数字，那么可以使用ANSI的标准C库函数sscanf()把串数组转换为合适的类型。在转换时，int型用%d，long int型用%ld，float型用%f，double型用%lf。下面是一些从图5-13中得到的输入两个字符串和一个数字（为C语言中为类型为double的变量dollars，对应于数据库中的double precision类型的列值）的代码：

```
char acctfrom[11], acctto[11];
double dollars;
char dollarstr[11];

while ((prompt("Enter from, to accounts and dollars for transfer:\n",
    acctfrom, acctto, &dollars, dollarstr) == 3) &&
```

```

    3. acctfrom, 10, acctto, 10, dollarstr, 10) < 0) ||
    (sscanf(dollarstr, "%lf", &dollars) != 1) { /* convert to double */
        printf("Invalid input. Input example: 345633 445623 100.45\n");
    }
}

```

在sscanf()函数中dollars变量前的&符号用来创建作为参数的指向该变量的指针。这样，被调用的函数就可以修改变量的值了。这是C语言中利用非数组类型变量的函数参数给调用者返回值的方法。

在代码例子中使用数字常数

我们的程序有一个特别的地方。注意，在上面的例子中对prompt()的调用包含为常量的参数，如图5-1中对prompt()函数调用时所使用的4：

```
while((prompt(cid_prompt, 1, cust_id, 4)) >= 0) { /* main loop, get cid */
```

正如上面所说明的那样，4代表了能在数组cust_id中放置的字符串的最大长度。一般C编程惯例要求在此使用符号常量，而不是直接常量4。于是，在源文件头，我们需要加上：

```
#define CIDLEN 4
```

然后把对prompt()的调用改为：

```
while((prompt(cid_prompt, 1, cust_id, CIDLEN)) >= 0) { /* . . . */
```

```

/*
 * prompt.c: prompted input of one or more tokens on one line of input
 */
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include "prompt.h"
#define LINELEN 1000
int prompt(char prompt_str[], int ntokens, . . .)
{
    va_list ap; /* type from stdarg.h */
    char line[LINELEN]; /* buffer for input line */
    char *token; /* token from user input */
    char *tbuffer; /* caller's buffer for token */
    int maxlen, i;

    va_start(ap, ntokens); /* start at last named arg */
    printf("%s", prompt_str); /* output prompt to user */
    fgets(line, LINELEN, stdin); /* get line from user */
    for (i=0; i<ntokens; i++) {
        /* strtok takes line the first time it's called, 0 after that */
        if ((token = strtok(i?0:line, "\t\n")) == 0)
            return -1;
        tbuffer = va_arg(ap, char *); /* caller's buffer for string */
        maxlen = va_arg(ap, int); /* and length of that buffer */
        if (strlen(token) > maxlen) { /* check length of user token */
            va_end(ap);
            return -1; /* fail: user token too long */
        }
        strcpy(tbuffer, token); /* return token to caller */
    }
    va_end(ap);
    return 0; /* success */
}

```

图B-2 prompt()函数的源代码，文件prompt.c

对于使用大量常量CIDLEN的应用程序，在include文件中定义常量更合适。这是因为当cid列的字符数由于表定义改变而改变时（例如长度从4变为6），相应的定义也能改变。于

是我们就很容易修改程序，而不需要搜索每个文件中的4，然后再根据上下文决定是否需要把某个4改为6。在知道了这点的前提下，我们在本书的例子中并不遵守此习惯。因为这样能够使在上下文中某个常量的值更显而易见。这并不等于说你可以不遵守好的编程惯例。

`prompt()`函数的代码在图B-2中。它利用了ANSI C处理不定个数变量的功能，使用在`stdarg.h`中定义的`va_list`类型。请参考Brian Kernighan和Dennis Ritchie的《The C Programming Language》第2版(Englewood Cliff, NJ: Prentice Hall, 1988)的7.3节。

B.2 print_dberror()函数

为了获得与SQLERROR条件相匹配的ORACLE的错误消息，我们可以调用ORACLE库函数`sqlglm()`。注意，`sqlglm()`提供的错误消息并不是以空字符结束的。为了使它以空字符结束，我们可以利用在`errsize`中返回的长度。然后把错误消息用`printf`打印出来，如图B-3所示。

```
/* print_dberror()--print database error message, ORACLE case */
#define ERRLEN 512                                /* max length of an ORACLE error message */
void print_dberror()
{
    int errlength = ERRLEN;                      /* size of buffer */
    int errsize;                                  /* to contain actual message length */
    extern sqlglm();

    char errbuf[ERRLEN+1];                         /* buffer to receive message */
    sqlglm(errbuf, &errlength, &errsize); /* get error message from ORACLE */
    errbuf[errsize] = '\0';                        /* make sure it is null terminated */
    printf("%s\n",errsize,errbuff);                /* print it out */
}
```

图B-3 针对ORACLE的print_dberror()函数

```
/* print_dberror()--print database error message, DB2 UDB case */
#include <sql.h>
#define MAXERRLEN 512                            /* max length of an error message */
#define LINWIDTH 72                               /* max characters desired on a line */
extern struct sqlca *sqlca;                   /* reference global SQLCA struct */
void print_dberror()
{
    char errbuf[ERRLEN];                         /* buffer to receive message */
    int errlen;
    if ((errlen=sqlaintp(errbuf, MAXERRLEN, LINWIDTH, &sqlca)) > 0)
        printf("%*s\n",errlen,errbuff);
    else
        printf("No error message provided. SQLCODE = %d, SQLSTATE = %s\n",
               sqlca.sqlcode, sqlca.sqlstate);
}
```

图B-4 针对DB2 UDB的print_dberror()函数

DB2 UDB提供了类似的工具。请参见图B-4的代码。此外，你需要根据例5.1.1的解释修改`Connect`语句和`Disconnect`语句为SQL-92的形式。可以参见附录C以得到`Connect`语句和`Disconnect`语句的完整语法。

B.3 编译嵌入式C程序

如我们在第5章中所说的那样，C语言编译程序并不认识嵌在文件中的EXEC SQL语句的语法，所以，源文件一般首先要经过预编译器的处理。预编译器把嵌入的语句转换成相应的C语言语句。对于不同的系统，预编译/编译过程是不同的，下面是两个例子。

1. ORACLE/UNIX环境下的预编译/编译过程

在UNIX环境下ORACLE版本7和版本8中，程序员应首先创建名为main.pc的源文件，它包含程序中的C语言语句和嵌入式SQL（EXEC SQL）语句。后缀.pc表示该文件是带嵌入式SQL结构的源文件。ORACLE程序员通过执行以下命令来调用预编译器：

```
proc i name=main.pc
```

此命令将会生成一个新的文件main.c。在此文件中，所有的EXEC SQL语句都会被相应的纯C语句替换。考虑例5.1.1中的Select语句：

```
exec sql select cname, discnt into :cust_name, :cust_discnt
      from customers where cid = :cust_id;
```

它将被一系列调用ORACLE运行期库函数替换。在创建了main.c以后，使用下面的语句就可以编译该文件了：

```
cc -c main.c
```

它将创建一个名为main.o的新目标文件。使用类似的命令可以编译前面提到的prompt.c文件并创建新文件prompt.o。为了连接main.o和prompt.o为一个可执行文件main，需要执行：

```
cc -o main main.o prompt.o $ORACLE_HOME/lib/libsql.a ...<many libraries>
```

实际上，专家一般都是创建一个包含每一步骤的make程序的描述文件(makefile)。有了make程序的描述文件(makefile)，整个连编过程就可以只用一个命令来完成了，例如make E=main。最后产生的可执行文件可能要占用非常大的磁盘空间，所以限制目录下的可执行文件数目是一个不错的习惯。

注意，数据库系统一般都带有例程以及相应的连编过程。它使得在目录下程序可以和运行数据库系统所需要的可执行文件一起工作。

2. DB2 UDB/UNIX环境下的预编译/编译过程

如我们在第5章中所说的那样，C语言编译程序并不认识嵌在文件中的EXEC SQL语句。所以，源文件一般首先要经过预编译器的处理。预编译器把嵌入的语句转换成相应的C语言语句。对于不同的系统，预编译/编译过程是不同的。

在UNIX环境下DB2 UDB版本5或者版本6中，程序员应首先创建名为main.sqc的源文件。它包含程序中的C语言语句和嵌入式SQL（EXEC SQL）语句。后缀.sqc表示该文件是带嵌入式SQL结构的源文件。首先使用预编译器prep，它以main.sqc为输入main.c为输出。此时所有的EXEC SQL语句都会被相应的纯C语句替换。考虑例5.1.1中的Select语句：

```
exec sql select cname, discnt into :cust_name, :cust_discnt
      from customers where cid = :cust_id;
```

它将被一系列调用DB2运行期库的函数替换。prep还创建一个为数据库在较高层次描述程序的二进制文件。可以通过使用另一个DB2命令bind把此二进制文件提供给数据库。通过运行DB2脚本embprep就可以完成这两个步骤以及连接/断开连接数据库的动作。该脚本可以

在文件系统的samples/c子目录下找到。程序员只要提供正确的参数就可以使用此脚本了。

```
embprep main [mydbname [username password]]
```

缺省的数据库名为sample，即DB2提供的示例数据库。你可能不需要提供用户名和口令，这是因为有些系统使用操作系统的权限来进行识别用户。

此时，程序还是以.c的形式存在的。可以使用普通的C编译程序来把它编译成目标文件main.o。方法如下：

```
cc -c main.c
```

使用类似的命令可以编译前面提到的prompt.c文件，并创建一个新文件prompt.o。为了把main.o和prompt.o连接为一个可执行文件main，我们需要使用如下命令：

```
cc -o main main.o prompt.o -L$(DB2INSTANCEPATH)/sqllib/lib -R ... -ldb2
```

实际上，专家一般都为一个节点创建一个包含每一步骤的make程序的描述文件。在sample/c目录下有一个make程序的描述文件可以参考。有了make程序的描述文件，整个连编过程就可以只用一条命令来完成了，例如make E=main。

注意，数据库系统一般都带有例程以及相应的连编过程。它负责在目录中检查运行数据库系统所需可执行文件。

你可以直接读embprep脚本的内容，看到它是通过UNIX命令db2 prep main.sqc bindfile来运行prep的。这意味着prep是叫做db2的UNIX程序的参数，而不是一个独立的程序。奇怪的是，在运行db2程序之后，对数据库的连接仍然是打开着的。所以，你可以先使用db2 connect mydb，然后回到UNIX提示符下运行某些UNIX命令，然后再使用db2 select pid from products来察看显示出来的pid。最后，你可以使用db2 connect reset来断开连接。embprep脚本就利用了这种能力。如果你只使用db2，那么你可以先进入db2环境（该环境有自己的提示符），直到你使用quit退出该环境为止。

附录C SQL 语 法

本附录收集了本书中出现的标准的关系SQL语句和对象-关系SQL语句的语法（偶尔也会提到用关系SQL语句处理对象-关系功能）。这两种语句分别列在图C-1和C-2，但是在文中是按字母顺序交叉出现的。此外针对标准SQL语句，这里还给出了前文中没提到的一些不同产品的SQL语句细节。

通常主要讨论我们所介绍的语句如何适应于通用SQL的标准。SQL-92、Core SQL-99和X/Open中经常用到的基本SQL和高级SQL的语法主要在第3章和第5章。

图C-1中所有SQL语句都可以内嵌于程序中。只有Select有额外的作为交互语句的功能：可以显示多行给用户。在嵌入式SQL中要处理多行记录，必须使用5.1节中提到的游标。有一些语句一般不作为交互SQL使用，如Commit语句。这些语句一般用EXEC SQL开头，而不像其他语句可以省略[EXEC SQL]。

Alter Table	在已有表上增加或删除列或约束
Close Cursor	关闭游标
Commit	成功提交事务
Connect	连接到数据库
Create Index	在基本表上创建索引
Create Schema	创建模式来保存表、视图等
Create Table	创建基本表(可使用于关系和对象-关系)
Create Tablespace	创建表空间(在ORACLE和DB2 UDB)
Create Trigger	创建触发器
Create View	创建视图
Declare Cursor	定义游标
Delete	删除表中行
Describe	提取有关动态准备列的信息
Disconnect	断开与数据库连接
Drop	删除表、视图、模式、触发器或索引
Execute	执行动态准备的语句
Execute Immediate	执行宿主变量字符串中的SQL语句
Fetch	从当前行中提取值并移动游标
Grant	授予表特权
Insert	向表中插入记录
Open Cursor	打开先前声明的游标
Prepare	准备要执行的动态SQL语句
Revoke	回收表特权
Rollback	使事务到达一个不成功结论
Select	从表中检索数据 (同时适用于关系和对象-关系)
Update	更新表中数据

图C-1 本附录给出的关系SQL语句

图C-2主要介绍了对象-关系和有关的用户定义函数语句。其中，没有涉及到C中嵌入式编程，如读入对象值到程序中合适的结构变量中的过程。这样在介绍这些语句时就省略了前缀

[EXEC SQL]。在ORACLE中有一个叫对象类型翻译程序(OTT)，可以把Create Type语句转化为相应的C结构声明。通过OCI(ORACLE Call Interface)提供的数组ADT可以访问C中的嵌套表。INFORMIX也有API可以访问汇集中的数据。

Create Function
Create Row Type
Create Table (同时适应于关系和对象-关系情况)
Create Type
Drop Function
Drop (Row) Type

图C-2 本附录给出的对象-关系和用户自定义函数语句

要注意的是这里省略了一些产品的特殊语句。参考产品的SQL使用手册可获得全部细节。第3章最后列出了ORACLE、DB2 UDB和INFORMIX的使用手册。关于游标和动态SQL语句的使用可参考第5章末尾列出的嵌入式SQL使用手册。

C.1 Alter Table语句

7.1节已经介绍了Alter Table语句，ORACLE、DB2 UDB和INFORMIX有不同的形式，分别如图7-7、7-8和7-9所示。这些格式在图C-3、C-4和C-5重新描述了一下。Alter Table允许DBA修改在Create Table语句中定义的结构，增加或删除表中的列，同样许多产品增加或删除约束。这条语句会影响已经存在的表中列，这样会给数据存储带来许多新的问题。Entry SQL-92中没有提供Alter Table语句，而在Core SQL-99中只提供增加新列的功能，不能删除旧的列或增加、删除约束。Full SQL-99提供所有这些功能。X/Open的Alter Table语句只提供增加和删除列功能。所有这些标准中增加列的语法都与DB2 UDB所使用的一样。由于这方面缺少标准约定，各产品之间差别很大。

[EXEC SQL] ALTER TABLE [schema.]tablename
[ADD ({columnname datatype [DEFAULT {default_constant NULL}]} [col_constr {col_constr...}]
table_constr) -- choice of columnname-def. or table_constr
{, {columnname datatype
[DEFAULT {default_constant NULL}]} [col_constr {col_constr...}] table_constr)
...))]) -- zero or more added columnname-def or table_constr
[DROP COLUMN columnname (columnname [, columnname...])]
[MODIFY (columnname data-type
[DEFAULT {default constant NULL}] [[NOT] NULL]
{, columnname data-type
[DEFAULT {default_constant NULL}] [[NOT] NULL]
...))]
[DROP CONSTRAINT constr_name]
[DROP PRIMARY KEY]
[disk storage and other clauses (not covered)]
[any clause above can be repeated, in any order]
[ENABLE and DISABLE clauses for constraints];

图C-3 ORACLE中Alter Table语法

```

ALTER TABLE [schema.]tablename
  [ADD [COLUMN] columnname datatype
   [DEFAULT {default_constant | NULL}]] [col_constr {col_constr...}]
  [ADD table_constr]
  [DROP CONSTRAINT constr_name]
  [DROP PRIMARY KEY]
  [repeated ADD or DROP clauses, in any order]
  [disk storage and other clauses (not covered)];

```

图C-4 DB2 UDB 的Alter Table语句语法

```

[EXEC SQL] ALTER TABLE [schema.]tablename
  [ADD new_col | (new_col {, new_col...})]
  [DROP columnname | (columnname {, columnname...})]
  [ADD CONSTRAINT table_constr | (table_constr {, table_constr...})]
  [DROP CONSTRAINT constraintname | (constraintname {, constraintname...})]
  [repeated ADD or DROP clauses, in any order]
  [disk storage and other clauses (not covered)];

约束单个列值的new_col形式如下：

  columnname datatype
  [DEFAULT {default_constant|NULL}]] [col_constr {col_constr...}]

```

图C-5 INFORMIX的Alter Table语句语法

ORACLE、DB2 UDB和INFORMIX的Alter Table 语句

ORACLE产品提供了许多特征，包括修改已存在的列的能力。查找Alter Table语句的准确语法可参考产品的SQL参考手册。

C.2 Close Cursor语句

在所提到的所有标准和产品中，Close Cursor语句都有如下形式：

```
EXEC SQL CLOSE cursor-name;
```

这条语句主要作用是关闭游标，以便不能再访问活动记录。由于游标没有用在专门的SQL中，所以它总是在程序中完成。详细信息参见5.4节中的图5-12。

C.3 Commit Work语句

在所提到的所有标准和产品中，提交事务的语句都有如下形式：

```
EXEC SQL COMMIT [WORK];
```

这条语句就是成功完成事务，把事务中所做的所有行的更新永久记录到数据库中，并让所有用户可以看到这个结果，同数据特权一致。详细信息参见5.4节。

C.4 Connect语句

Connect语句主要用在嵌入式SQL中和次与数据库建立连接的时候。在交互式SQL中，由用户界面来为用户会话完成与数据库的连接。在Entry SQL-92或Core SQL-99中都没有Connect和Disconnect语句；这样允许生产厂商自由指定需要什么：数据库服务器、数据库、

用户名、口令、授权服务器等等。Full SQL-99中的语法如下：

```
EXEC SQL CONNECT TO target-server [AS connect-name] [USER username];
```

或

```
EXEC SQL CONNECT TO DEFAULT;
```

这里target-server是由实现时决定的（不是由SQL-99定义，而是由生产厂商定义）。connect-name是将来引用这一连接的标识符，特别是这一连接是你所拥有的几个连接中的一个的时候。

在X/Open标准中还有一个附加的USING子句：

```
EXEC SQL CONNECT TO target-server [AS connect-name]
    [USER username [USING authentication]];
```

或

```
EXEC SQL CONNECT TO DEFAULT;
```

在ORACLE中连接的基本嵌入式SQL语句是：

```
EXEC SQL CONNECT :user_name IDENTIFIED BY :user_pwd;
```

注意，在SQL语句中的冒号表示宿主变量。DB2 UDB使用Full SQL-99标准，并且允许在用OS用户名来标识用户时第一种形式中不带USER子句。INFORMIX使用X/Open形式。详细信息参见5.1节。

C.5 Create Function语句 (UDF)

只有SQL-99中指定SQL中用户定义函数的调用。其中Create Function语句给出了许多选项，图C-6给出了最主要的选择。它与Create Procedure语句很相似，只是过程没有返回值。

```
CREATE FUNCTION [schema.]functionname ([param [, param ...]])
    RETURNS datatype
    [LANGUAGE SQL|C|FORTRAN|COBOL] -- or some others, but not Java (yet)
    [NO SQL|CONTAINS SQL] READS SQL DATA | MODIFIES SQL DATA
    {executable_SQL_statement           -- for SQL/PSM function
     |EXTERNAL NAME external_function_name} -- for external function
    param ::= [IN|OUT|INOUT] paramname datatype
```

图C-6 Core SQL-99的Create Function语句语法（部分）

目前有两种UDF，一种是存储过程，主要用可在数据库服务器上安全运行的语言开发（一般是Procedural SQL），另一种是外部函数，可用C或其他语言开发。

在Create Function语句选择LANGUAGE SQL定义该函数为SQL函数，连同它的代码，存储在数据库服务器上，也就成为了存储过程。SQL函数中可以有可执行的SQL语句。这些可执行语句(executable_SQL_statement)可以是SQL/PSM (SQL-99的过程式SQL语言中永久性存储模块) 中的形如BEGIN ... END的复合语句。因此可用PSM的代码块(PSM_Code_block)来取代可执行SQL语句(executable_SQL_statement)。关于SQL函数的例子参见4.4节，那里的过程性SQL语言对于ORACLE使用PL/SQL而INFORMIX使用SPL。

另一方面，在Create Function语句中如选用其他语言，如LANGUAGE C，则可以定义外部函数，当然它需要EXTERNAL NAME子句来指明代码放在哪个文件（在服务器以外）中。

注意，对于通过SQL访问或避免访问数据库数据的外部函数要进行保护。详细信息参见4.5节。

ORACLE的语法同图C-6的Core SQL-99很类似，并有图C-7中的重要子句。在某些库文件中这种形式可以定义PL/SQL函数或外部C函数。关于前者的例子参见4.4.1节。

```

CREATE [OR REPLACE] FUNCTION [schema.]functionname
  ([oparam [, oparm ...]])
  RETURN datatype IS
  {PL/SQL_code_block}                                -- for PL/SQL function
  |EXTERNAL LIBRARY libname [NAME ext_functionname]; -- for external UDF

oparm ::= paramname [IN|OUT|IN OUT] datatype

```

图C-7 ORACLE的Create Function语句语法（部分）

INFORMIX用来定义SPL函数或外部C函数的语法略有不同，如图C-8所示。在ORACLE和INFORMIX产品中，外部函数可使用C语言代码来访问数据库。这两种产品的外部函数中也可以使用Java语言。

```

CREATE [OR REPLACE] FUNCTION [schema.]functionname
  ([iparam [, iparam ...]])
  RETURN datatype IS
  {PL/SQL_code_block}                                -- for SPL function
  |EXTERNAL LIBRARY libraryname [NAME ext_functionname]; -- for external UDF

iparam ::= paramname [IN|OUT|IN OUT] datatype

```

图C-8 INFORMIX的Create Function语句语法（部分）

DB2 UDB中没有过程性语言。它用Create Function来声明外部C或Java函数。它的语法与SQL-99的语法非常接近。因为外部函数中不能使用SQL语句（这是一个很严重的限制），所以带有No SQL。外部函数主要用于单个列值的操作。此外，用RETURNS TABLE替代“RETURNS datatype”的Create Function可以定义表函数。表函数是用C或Java实现的，用来把非数据库中数据导入SQL可引用的虚表中。

C.6 Create Index语句

第8章介绍的Create Index语句有许多不同的形式，如图8-1(X/Open)、图8-7和8-14(ORACLE)、图8-15和8-20(DB2 UDB)。这些产品的Create Index语法规整理在图C-9和图C-10中。

下面是X/Open中关于Create Index语句的标准说明(SQL-99中没有提到Create Index语句)：

```

[EXEC SQL] CREATE [UNIQUE] INDEX [schema.]indexname
  ON tablename (columnname [ASC | DESC] [, columnname [ASC | DESC] ...]);

```

ORACLE、DB2 UDB和INFORMIX都支持这种标准。每个产品都对语法进行了扩充以指定其他信息，如索引存储在什么地方、创建什么样的索引以及如何递增存储索引。

1. ORACLE的Create Index语句

ORACLE的Create Index语句语法如图C-9所示。TABLESPACE子句用来指定创建的索引要

存储的表空间。ORACLE数据存储结构示意图参见图8-3。图8-4后面的文字介绍了STORAGE子句的作用。NOSORT选项的介绍在图8-7后面，而PCTFREE子句的介绍在图8-14后面。

```
[EXEC SQL] CREATE [UNIQUE | BITMAP] INDEX [schema.]indexname ON [schema.]tablename
  (columnname [ASC | DESC] {, columnname [ASC | DESC]}...)
  [TABLESPACE tb1spacename]
  [STORAGE ([INITIAL n [K|M]] [NEXT n [K|M]] [MINEXTENTS n] [MAXEXTENTS n]
            [PCTINCREASE n] ) ]
  [PCTFREE n]
  [other disk storage and transactional clauses not covered]
  [NOSORT]
```

图C-9 ORACLE的Create Index语句语法

```
[EXEC SQL] CREATE [UNIQUE] INDEX indexname ON [schema.]tablename
  (columnname [ASC | DESC] {, columnname [ASC | DESC]...})
  [INCLUDE (columnname [ASC | DESC] {, columnname [ASC | DESC]...})]
  [CLUSTER]
  [PCTFREE n]
  [MINPCTUSED n]
  [ALLOW REVERSE SCANS];
```

图C-10 DB2 UDB的Create Index语句语法

2. DB2 UDB的Create Index语句

DB2 UDB的Create Index语句语法如图C-10所示。图8-15后面的讨论详细介绍了图C-10的各个选项。

和DB2 UDB一样，INFORMIX支持CLUSTER索引。和ORACLE一样，它也允许索引的位置在自己的磁盘区域内（在INFORMIX中称为库空间，而在ORACLE中称为表空间）。此外INFORMIX带有DataBlades提供的访问方法的索引，如地理数据的二维索引。

3. 对象-关系的考虑

ORACLE允许索引使用对象的任何属性，但是要求用以表名开始的带点表达式来指明。使用Create Index中STORE AS子句指定的表名也允许嵌套表的索引。INFORMIX允许使用行类型列或列的函数值作为索引列。但是，INFORMIX不允许行类型列中的字段单独出现作为索引中列名。因为例4.2.23中WHERE子句的字段名是例4.2.20中行类型定义的字段中的字段，所以它不能作为列被索引。因为主键一定是要建索引的，所以表列中的字段不能为主键。

C.7 Create Row Type语句

用户自定义结构类型，在ORACLE中称为对象类型而在INFORMIX中称为行类型，可用如下语句创建：

```
CREATE ROW TYPE rowtype
  (fieldname datatype [NOT NULL]{, fieldname datatype [NOT NULL] ...})
  [UNDER supertype];
```

可在本附录中查看SQL-99中相应的Create Type语法。关于行类型的更多信息参见4.2.2节。

C.8 Create Schema语句

Core SQL-99中的模式主要是针对单个用户来讲的，它就是指表、索引和一些其他相关数据库对象的集合。数据库目录中表的全名是模式名.表名 (schemaname.tablename, 模式名往往是用户名)，其他数据库对象也是如此。更多信息参见7.4节。

以下是SQL-99的Create Schema语句的主要语法。在Core SQL-99中可以为新的模式名指定明确的名字。DBA可用AUTHORIZATION子句为用户设置一个模式，让模式名缺省为 authorization_id, 通常为用户名字。

```
[EXEC SQL] CREATE SCHEMA
[schemaname]                                -- to specify explicitly; not Core SQL-99
| AUTHORIZATION authorization_id           -- default schemaname is authorization_id
[({schema_element schema_element ...})];
```

X/Open有相同的语法，并有额外一个选项用来指明字符集。在Core SQL-99和X/Open中可选的模式元素(schema_elements)是用来创建数据库对象的SQL语句：Create Table, Create View, Create Index和Grant。它们一般给新的模式提供初始内容。这里所涉及到的三个产品中，只有DB2 UDB允许用户用显式指定的模式名创建附加的命令模式。这三个产品都支持Core SQL-99。

C.9 Create Table语句

7.1节介绍了Create Table语句的基本SQL形式，使用了ORACLE、DB2 UDB和INFORMIX的通用语法，如图C-11所示。

```
[EXEC SQL] CREATE TABLE [schema.]tablename
(({columnname datatype [DEFAULT {default_constant|NULL}]} [col_constr {col_constr...}])
| table_constr) -- choice of either columnname-definition or table_constr
[, ({columnname datatype [DEFAULT {default_constant|NULL}]} [col_constr {col_constr...}])
| table_constr]
...));          -- zero or more additional columnname-def or table_constr
```

约束单个列值的Col_constr形式如下：

```
{NOT NULL |                      -- this is the first of a set of choices
[CONSTRAINT constraintname]    -- if later choices used, optionally name constraint
      UNIQUE                   -- the rest of the choices start here
      | PRIMARY KEY
      | CHECK {search_cond}
      | REFERENCES tablename [(columnname)]
          [ON DELETE CASCADE]]
```

约束多个列值的table_constr形式如下：

```
[CONSTRAINT constraintname]
  (UNIQUE {columnname [, columnname...]})      -- choose one of these clauses
  | PRIMARY KEY (columnname [, columnname...])
  | CHECK {search_condition}
  | FOREIGN KEY (columnname [, columnname...]) -- following is all one clause
      REFERENCES tablename [(columnname [, columnname...])]
      [ON DELETE CASCADE]]
```

图C-11 Create Table语句的基本SQL语法

ORACLE、DB2 UDB和INFORMIX都接受基本SQL语法，除了在INFORMIX中需要所有的列定义在表约束定义之前，而基本SQL语法中是允许列定义和表约束定义以任何次序出现。这些子句的讨论参见7.1节。

1. ORACLE 的Create Table 语句

ORACLE的Create Table语法如图C-12所示，在前面图8-5(除了CLUSTER子句)和图8-22(CLUSTER子句)中也已经提到过。它的前四行语法与基本SQL语法一样。

```
[EXEC SQL] CREATE TABLE [schema.]tablename
  ({columnname datatype [DEFAULT {default_constant|NULL}]} [col_constr {col_constr...}]
   | table_constr)          -- choice of either columnname-definition or table_constr
  (, {columnname (repeat DEFAULT clause and col_constr list) | table_constr}...)
  [CLUSTER clustername {columnname [, columnname ...]}] --for table in cluster
  | [[ORGANIZATION HEAP | ORGANIZATION INDEX (this has clauses not covered)]]
  [TABLESPACE tb1spacename]
  [STORAGE ([INITIAL n [K|M]] [NEXT n [K|M]] [MINEXTENTS n] [MAXEXTENTS (n | UNLIMITED)]
            [PCTINCREASE n]) (additional STORAGE options not covered)]
  [PCTFREE n] [PCTUSED n]
  [disk storage and other clauses (not covered)] )
  [AS subquery]
```

图C-12 ORACLE的Create Table语句语法

注意最后一行的AS subquery，这个标准的扩展允许Create Table语句指定插入到新表中的行。STORAGE子句在图8-4后面说明，而PCTFREE和PCTUSED子句在图8-5后面说明。

2. DB2 UDB中Create Table语句

DB2 UDB的Create Table语句（图C-13）与ORACLE形式相比有较详细的存储说明和ORACLE一样，在命名的表空间内指定表的位置，而表空间可以通过存储说明来建立，这方面信息参见本附录的Create Tablespace语句。

```
[EXEC SQL] CREATE TABLE [schema.]tablename
  ({columnname datatype [DEFAULT {default_constant|NULL}]}
   [col_constr {col_constr...}] | table_constr)
  (, {columnname datatype [DEFAULT {default_constant|NULL}]}
   [col_constr {col_constr...}] | table_constr)...
  [IN tb1spacename [INDEX IN tblespacename]]
  [NOT LOGGED INITIALLY]
  [disk storage and other clauses (not covered)]
```

图C-13 DB2 UDB的Create Table 语句语法

INFORMIX的Create Table语句有一个IN dbspace子句而ORACLE和DB2 UDB有IN tablespace子句来指定磁盘存储的特定单元中表的位置。它也可以指定第一个及其后的扩展的大小，也就是磁盘分配单元的大小。

3. 对象-关系语法

带有对象-关系扩展的Create Table语句语法如图C-14所示。如果表名后面有OF typename，则这表就是对象表，否则它就是关系表。所有关系表的存储子句同样也适用对象表。表的聚簇与索引结构中的成员的特殊选项只局限于关系表中。在column_def语法表明数据类型是可选的，但是在其他部分都没有指定数据类型即对于关系表没有AS子查询子句的情况下，却必须指明数

据类型。如果数据类型已用对象类型或子查询指明了，则可在column_def中忽略数据类型。

```
[EXEC SQL] CREATE TABLE [schema.]tablename
    [OF [schema.]typename]                                -- for object tables
    (column_def | table_constr | table_ref_clause
     {, column_def | table_constr | table_ref_clause ...})
    (CLUSTER clustername (columnname [, columnname ...])) -- not for object tables
    {[ORGANIZATION HEAP | ORGANIZATION INDEX]          -- not for object tables
     [TABLESPACE tb1spacename]
     [STORAGE ([INITIAL n [K|M]] [NEXT n [K|M]] [MINEXTENTS n]
               [MAXEXTENTS {n | UNLIMITED}] [PCTINCREASE n])
      [PCTFREE n] [PCTUSED n]
     [NESTED TABLE columnname STORE AS tablename
      (. NESTED TABLE columnname STORE AS tablename...)]
     [AS subquery];

column_def ::=
    columnname [datatype] [DEFAULT {default_constant|NULL}] -- see discussion
                [col_constr {col_constr...}]

table_ref_clause ::=
    SCOPE FOR (columnname | dotted_expr) IS [schema.]tablename
```

图C-14 ORACLE中带有对象-关系扩展的Create Table语句的语法

在INFORMIX中Create Typed Table语句可以不用嵌套表子句或范围子句，而且它还可以处理类型定义的空列。其他列的约束可当做表约束处理，并不提供列的缺省值。它也提供一些存储说明，来弥补对于类型表语法过于简单的不足，如图C-15所示。

```
[EXEC SQL] CREATE TABLE [schema.]tablename
    OF TYPE row_typename ( table_constr {, table_constr ...})
    [IN dbspace] [EXTENT SIZE n_KB] [NEXT size n_KB]
    [UNDER supertablename];
```

图C-15 INFORMIX的Create Typed Table语句语法

C.10 ORACLE和DB2 UDB的Create Tablespace语句

Create Tablespace不出现在任何标准中。表空间为用户提供一种修改数据库的缺省设置以及充分使用磁盘资源的方法。ORACLE的Create Tablespace语句语法如图C-16（或图8-4）所示。关于ORACLE中存储结构元素的更完整的描述参见8.2节，其结构图在图8-3中。

```
[EXEC SQL] CREATE TABLESPACE tb1spacename
    DATAFILE 'filename' [SIZE n [K|M]] [REUSE] [AUTOEXTEND OFF
    | AUTOEXTEND ON [NEXT n [K|M]] [MAXSIZE {UNLIMITED |n [K|M]}]
    {, 'filename' (repeat SIZE, REUSE, and AUTOEXTEND options) . . .}
    -- the following optional clauses can come in any order
    [ONLINE | OFFLINE]
    [DEFAULT STORAGE ([INITIAL n] [NEXT n] [MINEXTENTS n] [MAXEXTENTS {n|UNLIMITED}]
                  [PCTINCREASE n]) (additional DEFAULT STORAGE options not covered)]
    [MINIMUM EXTENT n [K|M]]
    [other optional clauses not covered];
```

图C-16 ORACLE的Create Tablespace语句语法

DB2 UDB的Create Tablespace语句语法在图C-17。DB2 UDB支持两种表空间：系统管理表空间和数据库管理表空间。系统管理表空间由操作系统的文件系统来管理，而数据库管理表空间不用文件系统管理而由数据库系统直接管理磁盘。对于小型数据库，可用系统管理表空间来简化创建和管理操作，而对高性能应用程序，则经常需要数据库管理表空间的额外工作。

```
[EXEC SQL] CREATE TABLESPACE tb1spacename
  MANAGED BY
    {SYSTEM USING 'filename' npages           -- system-managed space
     |DATABASE USING {FILE|DEVICE} 'containernname'}   -- or database managed space
    [EXTENTSIZE npages] [OVERHEAD nmilliseconds] [TRANSFERRATE nmilliseconds];
```

图C-17 DB2 UDB的Create Tablespace语句语法

在ORACLE中EXTENTSIZE用来指定磁盘分配单元的大小。可选参数OVERHEAD和TRANSFERRATE用来向查询优化器提供磁盘速度信息。OVERHEAD是以微秒为单位的开始I/O操作的时间，而TRANSFERRATE是读一个4KB页面到内存所花的时间的估计。

ORACLE和DB2 UDB都提供Alter Tablespace语句用来修改已创建的表空间，主要是加入更多的磁盘存储器，并可用Drop Tablespace删除表空间。因为表空间主要是指定如何实现数据库，而不是一个服务，所以我们所提到的标准中没有一个标准规定表空间的命令。

INFORMIX没有Create Tablespace语句。取而代之，它使用一种类似的磁盘空间单元也就是库空间。可用图形化管理程序ON-monitor或系统程序“onspaces”创建一个库空间。

C.11 Create Trigger 语句

触发器就是存储在数据库上(即存储在数据库模式中)并当某表上发生某些事件如插入、删除或更新时运行的代码序列。SQL-92中还没有定义触发器的标准，而SQL-99的Create Trigger语句语法如图C-18所示。虽然Core SQL-99中也没有触发器的标准，但它是称为基本触发器功能的附加特征，它只少了FOR EACH STATEMENT子句，它被给了另外一个特征标识为扩展触发器功能。由于几种主要的数据库产品都按标准实现了触发器(从1990年开始)，所以下面就分别对它们进行介绍(来自图7-10)。

```
[EXEC SQL] CREATE TRIGGER [schema.]trigger_name {BEFORE | AFTER}
  {INSERT | DELETE | UPDATE [OF columnname [, columnname...]]}
  ON tablename [REFERENCING corr_name_def [, corr_name_def...]]
  [FOR EACH ROW | FOR EACH STATEMENT]
  [WHEN (search_condition)]
  {statement                                -- action (single statement)
   | BEGIN ATOMIC statement; { statement;... } END}; -- action (multiple statements)

  定义相关名的Corr_name_def的形式如下：

  {OLD [ROW] [AS] old_row_corr_name
   | NEW [ROW] [AS] new_row_corr_name
   | OLD TABLE [AS] old_table_corr_name
   | NEW TABLE [AS] new_table_corr_name}
```

图C-18 SQL-99的Create Trigger语句语法

Create Trigger语句创建一个命名为trigger_name的对象，这个名字可出现在出错消息中并可在以后删除触发器(Drop trigger_name)时使用。在通过表名给定表上指定事件(某组字段集的INSERT、UPDATE或DELETE)发生之前或之后触发器被触发。

DB2 UDB的Create Trigger语句如图C-19所示，它与SQL-99标准非常接近。

```
[EXEC SQL] CREATE TRIGGER [schema.]trigger_name
  {NO CASCADE BEFORE | AFTER}
  {INSERT | DELETE | UPDATE [OF columnname [, columnname...]]}
  ON tablename [REFERENCING corr_name_def [, corr_name_def...]]
  {FOR EACH ROW | FOR EACH STATEMENT} MODE DB2SQL
  [WHEN (search_condition)]
  { statement
  | BEGIN ATOMIC statement; {statement;...} END};
```

图C-19 DB2 UDB的Create Trigger语句语法

由于ORACLE的Create Trigger语句是在标准制订以前完成的，所以它与标准有较大的差别。它的语法如图C-20所示。

```
[EXEC SQL] CREATE [OR REPLACE] TRIGGER trigger_name
  {BEFORE | AFTER | INSTEAD OF}
  {INSERT | DELETE | UPDATE [OF columnname [, columnname...]]}
  ON tablename [REFERENCING corr_name_def [, corr_name_def...]]
  {FOR EACH ROW | FOR EACH STATEMENT}
  [WHEN (search_condition)]
  BEGIN statement {statement;...} END;
  提供行相关名的Corr_name_def如下:
  {OLD old_row_corr_name
  |NEW new_row_corr_name}
```

图C-20 ORACLE的Create Trigger语句语法

INFORMIX的Create Trigger语句语法与SQL-99也非常相近。

C.12 Create Type语句

在SQL-99中Create Type语句用于创建用户自定义类型，目前有两种类型：结构类型和特殊(distinct)类型。特殊类型只是给一种类型起了另一个名字而已，其中源类型可以是任何内部类型，如数字、字符串或BLOB。但是特殊类型的项目如不经过转换则不能与源类型的项目比较。Core SQL-99创建一个特殊类型的最简单语句如下：

```
CREATE TYPE distinct_typename AS source_typename FINAL;
```

在DB2 UDB和INFORMIX中，特殊类型可用以下语句创建：

```
CREATE DISTINCT TYPE distinct_typename AS source_typename;
```

DB2 UDB允许使用任何内部类型作为源类型，INFORMIX则允许任何内部类型、行类型或另一个特殊类型，而在ORACLE中则没有特殊类型。注意特殊类型可很自然地用在纯关系数据库设计中。

结构用户定义类型是SQL-99中一个命名特征的扩展，用于定义一个UDT重要子句的语法如图C-21所示。

```

CREATE TYPE [schema.]typename [UNDER] typename AS
  (attrname datatype [, attrname datatype ...]) -- attribute list
  {FINAL|NOT FINAL}
  INSTANCE METHOD methodname [(paramname type [, paramname type ...])]
    RETURNS datatype
  [, INSTANCE METHOD methodname [(paramname type [, paramname type ...])]
    RETURNS datatype ...];

```

图C-21 SQL-99的创建用户自定义类型语句语法(结构)

这里UNDER可选项可用于定义继承层次中子类型。结尾子句用于定义这种类型在类型层次中是否是一个超类。

在ORACLE中，Create Type语句创建的结构类型，称为对象类型，如图C-22所示。

```

CREATE [OR REPLACE] TYPE [schema.]typename AS OBJECT
  (attrname datatype [, attrname datatype ...])
  MEMBER FUNCTION methodname [(param type [, param type ...])]
    RETURN datatype,
  (MEMBER FUNCTION methodname [(param type [, param type ...])]
    RETURN datatype, ...)
  PRAGMA RESTRICT_REFERENCES (DEFAULT, RNDS, WNDS, RNPS, WNPS)
);
CREATE [OR REPLACE] TYPE [schema.]typename          -- incomplete type
CREATE TYPE typename AS TABLE OF datatype;         -- nested table
CREATE TYPE typename AS VARRAY (max_elements) OF datatype; -- VARRAY

```

图C-22 ORACLE的Create Type语句语法

关于成员函数实现的更多信息参见4.4.1节。在INFORMIX中用户自定义的结构类型称为行类型，可用Create Row Type语句来创建，该语句也被列在本附录中。

DB2 UDB也提供结构的UDT，但是不能用于一般的列值类型中。数据库的所有UDT值都作为完整的行出现在对象表中，并且指向它们的REF可能出现在列中。Create Type语句的语法与SQL-99一样，只是它不允许定义方法。

C.13 Create View语句

Create View语句完整的基本语法如图7-13所示。

```

[EXEC SQL] CREATE VIEW [schema.]viewname [(columnname [, columnname...])]
  AS subquery [WITH CHECK OPTION];

```

这里的子查询可以包括一个可选的UNION子句。在程序中Create View语句作为嵌入式SQL语句是合法的，但视图的子查询中不能有宿主变量或任何动态参数。WITH CHECK OPTION可选项的具体含义参见图7-13。

ORACLE、DB2 UDB、INFORMIX、Entry SQL-92、Core SQL-99和X/Open都支持基本的Create View语法。此外ORACLE还提供额外的WITH READ ONLY用来防止对视图的更新。

对象—关系考虑

ORACLE、DB2 UDB和INFORMIX都允许视图表与对象表一样操作，即使子查询是建立在关系数据之上的。在这种情况下，在Create View语句中可用对象类型或行类型取替列列表。

在ORACLE中，最简单语法如下：

```
CREATE VIEW [schema.]viewname
  {[columnname [, columnname...]}];          -- relational syntax
  | OF [schema.]objtypename WITH OBJECT OID DEFAULT; -- simplest object view form
  AS subquery [WITH {READ ONLY | CHECK OPTION}];      -- where the data comes from
```

INFORMIX用更简单的语法OF [schema.] typename来指明对象形式（不需要OBJECT OID DEFAULT子句），而DB2 UDB的语法则更复杂一些。Full SQL-99中在列列表之后有可选项OF [schema.] typename，这样可方便地使用用户自定义的类型定义更有效的列别名。子查询需要为与类型匹配的每个顶层的ORACLE、DB2 UDB的属性或INFORMIX的字段提供一列。同样使用列列表和对象表的子查询来把对象表转化为关系表。

C.14 Declare Cursor语句

X/Open (Entry SQL-92) 的Declare Cursor语句的语法如图C-23所示，与5.3节图5-5基本一样。在图C-23中定义的子查询语法就是如图3-11所示的一般Select语句语法并且在图C-27中重复。

```
EXEC SQL DECLARE cursor_name CURSOR FOR
  Subquery
  [ORDER BY result_column [ASC | DESC] [, result_column [ASC | DESC]]...]
  [|FOR {READ ONLY | UPDATE [OF columnname [, columnname...]}]];
```

图C-23 X/Open嵌入式SQL的Declare Cursor语法

注意Fetch语句用于从游标中提取所选行，但是游标在行集合中只能向前移动。这个限制在SQL-99的可滚动游标扩展功能中得以克服。

SQL-99的Declare Cursor语句

Full SQL-99提供Declare Cursor和Fetch语句的一些重要特征。首先，Declare Cursor语法如图C-24所示。

```
EXEC SQL DECLARE cursor_name [{INSENSITIVE} {SCROLL}] CURSOR [{WITH HOLD}] FOR
  Subquery
  [{UNION Subquery}]
  [ORDER BY result_column [ASC | DESC] [, result column [ASC | DESC]]...]
  [|FOR READ ONLY | FOR UPDATE OF columnname [, OF columnname]];
```

图C-24 SQL-99嵌入式SQL的Declare Cursor语法

三个新的语法元素是关键字INTENSIVE、SCROLL和WITH HOLD；其中只有WITH HOLD在Core SQL-99中出现。如在游标定义时使用SCROLL选项，则游标就可前后滚动，并且Fetch语句的相应功能也可实现。详细信息参见5.7节。

C.15 Delete语句

目前有两种删除语句形式：一种是定位删除，即只删除游标中当前行；另一种是查找删除，与3.10节介绍的交互式SQL Delete一样。下面是这两种形式的语法：

```
[EXEC SQL] DELETE FROM [schema.]tablename
    [WHERE search_condition];                                -- searched delete
EXEC SQL DELETE FROM [schema.]tablename
    [WHERE CURRENT OF cursor_name];                         -- positioned delete
```

定位删除使用了一种特殊语法：CURRENT OF cursor_name。在这两种形式中，如不使用WHERE子句，则会把整个表都删除掉。详细信息参见图5-6。

X/Open、SQL-92、Entry SQL-99和Full SQL-99都对以上形式做出了规定。只是Full SQL-99在对象-关系上做了一些扩展。所有的主要数据库产品都支持以上形式。此外，ORACLE和DB2 UDB还可以在表名后面指定相关名，用在WHERE子句中。

对象-关系考虑

在有对象层次类型的表中做删除操作时，为限制只删除一个表中行，可用ONLY [schema.] tablename来替代[schema.] tablename。在Full SQL-99中有它的标准并得到INFORMIX和DB2 UDB的支持。

C.16 Describe语句

当为准备sqltext[]中的动态选择语句而调用Prepare语句时，编译进程就会计算要收回的列值个数和类型。为了使程序可以得到这些信息，必须调用动态SQL Describe语句，用来将检索到行的信息放到SQLDA变量结构中。Describe语句的语法如下所示：

```
EXEC SQL DESCRIBE statement_identifier INTO sqldavar pointer;
```

动态Select语句的例子在5.6节中。注意INFORMIX与ORACLE的SQLDA结果相差很大，但INFORMIX的与DB2 UDB的很接近。X/Open与SQL-99不使用SQLDA结构而是使用更多的SQL语法，但是DB2 UDB和ORACLE都没有实现这种方法。而INFORMIX则对SQLDA和X/Open语法都支持。

C.17 Disconnect语句

用在嵌入式SQL程序中的Disconnect语句用于断开与数据库的连接。与Connect语句一样，它不是Entry SQL-92或Core SQL-99标准的一部分，而且各生产厂家之间语法相差很大。Full SQL-99的Disconnect语句如下所示：

```
EXEC SQL DISCONNECT {connectname | CURRENT};
```

在X/Open中的Disconnect语句是：

```
EXEC SQL DISCONNECT {connectname|ALL|CURRENT|DEFAULT};
```

从ORACLE 7开始，ORACLE用不同的语句断开连接：

```
EXEC SQL COMMIT RELEASE;
```

DB2 UDB则遵循SQL-99标准，在Disconnect语句之前必须有Commit或Rollback语句。可在一条语句中执行提交和断开与数据库的连接：

```
EXEC SQL COMMIT RESET;
```

详细信息参见5.1节。

C.18 Drop Function语句

只有SQL-99定义了用户自定义函数。Drop Function语法如下：

```
DROP {FUNCTION [schema.]functionname [(paramtype [, paramtype ...])]  
|SPECIFIC FUNCTION [schema.]functionname}  
CASCADE|RESTRICT;
```

在Core SQL-99中没有CASCADE选项，而且必须显示地指定RESTRICT选项。有参数类型列表的形式可以删除多个过载函数中相应的一个，这些函数有相同的名字但它们的参数列表不同。

INFORMIX和DB2 UDB都遵守SQL-99标准，而ORACLE则有更简单的形式：

```
DROP FUNCTION [schema.]functionname;
```

因为ORACLE不支持“独立”函数（用Create Function创建，用Drop Function删除，并只在模式中不出现在任何包中的函数）过载，所以在ORACLE中不需要带有参数列表的形式。成员函数和包内函数可以过载。成员函数随它的对象类型一起删除。包中是ORACLE专有的关系数据库对象的封装集合，这已超出本书范围。

C.19 Drop Index语句

只有X/Open对Drop Index语句制订了标准，如下所示：

```
[EXEC SQL] DROP INDEX [schema.]indexname;
```

所有主要数据库产品都支持这种形式。更多信息参见8.1节。

C.20 Drop Trigger语句

只有SQL-99标准提到了触发器，它提供的语法如下：

```
[EXEC SQL] DROP TRIGGER [schema.]triggername;
```

几乎在所有提供触发器的产品中都支持这种语法，包括ORACLE、DB2 UDB和INFORMIX。详细信息参见7.1节。

C.21 Drop(Row) Type语句

只有SQL-99标准涉及到用户自定义类型，它提供的语法如下：

```
DROP TYPE [schema.]typename CASCADE|RESTRICT;
```

在Core SQL-99中没有CASCADE选项，而且必须显式地指明RESTRICT选项。因为用户自定义结构类型出现在SQL-99扩展功能中，所以在Core SQL-99中它只删除特殊类型。关于特殊类型与用户自定义结构类型的信息可参考本附录的Create Type部分。

在ORACLE中，用户自定义结构类型又称为对象类型，它的删除方法如下：

```
DROP TYPE [schema.]typename [FORCE];
```

这里的FORCE选项用于删除有REF依赖的类型。关于REF依赖的信息参见4.2.1节的末尾部分。

在INFORMIX中，用户自定义结构类型又称为行类型，它的删除方法如下：

```
DROP ROW TYPE [schema.]typename RESTRICT;
```

关于行类型的更多信息参见4.2.2节。INFORMIX也支持特殊类型，并可用Core SQL-99的语法删除它们。

在DB2 UDB中，用户自定义结构类型和特殊类型的删除可用同一种方法，如下：

```
DROP [DISTINCT] TYPE [schema.]
```

C.22 Drop {Schema | Table | View}语句

Drop语句用于从系统目录中删除模式、表或视图定义。如是表（或模式），则释放所有分配给它的空间。X/Open中Drop语句的标准的完整描述如下：

```
[EXEC SQL] DROP {SCHEMA schemaname | TABLE tablename| VIEW viewname}
    (CASCADE|RESTRICT);
```

其中，表名(tablename)或视图名(viewname)可用模式名(schemaname)作前缀。关于CASCADE和RESTRICT的讨论参见图7-14。奇怪的是，在Entry SQL-92中没有提到Drop Table语句。Full SQL-92与X/Open和Full SQL-99的一样。Core SQL-99中不需要CASCADE选项但仍需RESTRICT选项。各产品都与标准略有差别，如图C-25所示。

ORACLE	DB2 UDB	INFORMIX	X/Open, Full SQL-92, -99
DROP TABLE tablename...	[CASCADE CONSTRAINTS]		[CASCADE [RESTRICT]]
DROP VIEW viewname...			[CASCADE [RESTRICT]]
DROP SCHEMA schemaname...	没有用户定义模式	RESTRICT	没有用户定义模式

图C-25 Drop Table和Drop View语句在各产品和标准中的不同选项

当带有RESTRICT选项删除模式时，除非模式是空，也就是它不含有表或其他数据，否则不能成功。当使用CASCADE选项删除模式时，则模式中所有相关的数据都将被删除。正如Entry SQL-92和Core SQL-99所示，用户可以使用与用户名相同的名字作为模式名，这样当他们账户创建时，就不能创建其他模式。在这种情况下，ORACLE和INFORMIX的Drop Schema不是很有用。但在DB2 UDB中，用户可将与账户名相同的名字给原始模式，而可用Create Schema创建任何其他名字的模式，并且这些模式也可用Drop Schema来删除。

C.23 Execute语句

当调用Prepare语句来准备来自sqltext[]字符串的动态SQL之后，Execute语句用来把字符串中的动态参数与所需参数值关联起来，并执行准备好的语句。下面就是我们介绍过的所有标准及产品的Execute语句的语法：

```
EXEC SQL EXECUTE statement_identifier
    USING :host_variable {, :host_variable...};
```

Execute语句的例子参见5.6节。

C.24 Execute Immediate语句

因为Execute Immediate语句可执行含有宿主变量字符串的动态SQL语句，所以不需要前面的Prepare语句。我们介绍过的所有标准及产品的Execute Immediate语句一般模式如下：

```
EXEC SQL EXECUTE IMMEDIATE :host_string;
```

注意，因为Execute Immediate语句是同一类的，所以不需要任何如Prepare和Execute语句

处理的动态参数。host_string所包含的字符串内容必须是合法的SQL语句，可包括如下类型：Alter Table, Create Table, Delete, Drop (Table, View或Index), Grant, Insert, Revoke和Update。在特定产品中其他语句也可能允许。详细信息参见5.6节。

C.25 Fetch语句

这语句可从当前打开的游标中提取一行，在我们介绍过的产品中该语句都有如下形式：

```
EXEC SQL FETCH [FROM] cursor_name
    INTO :host_variable {, :host_variable ...}
    | USING DESCRIPTOR :sqldavar_pointer;
```

SQLDA变量指针(sqldavar_pointer)由动态游标使用，而宿主变量(host_variable)则用于接收由Fetch检索到的值。静态的情形参见5.3节，而动态的情形参见5.6节。在标准中使用描述名而不是指针。

Fetch语句在SQL-99中可滚动游标的应用

SQL-99中非动态游标的Fetch语句语法如图C-26所示。详细信息参见5.7节。目前我们所介绍的产品中只有INFORMIX产品支持这种嵌入式SQL特征。

```
EXEC SQL FETCH
    [{NEXT | PRIOR | FIRST | LAST
    |{ABSOLUTE | RELATIVE} value_spec} FROM ]
    cursor_name INTO host-variable {, host-variable};
```

图C-26 SQL-99中Fetch语句语法

C.26 Grant语句

表（基本表或视图表）的所有者通过Grant语句把各种对表的访问权限（选择、更新、删除或插入）赋给其他用户。它是一种表访问安全性的形式，但通过视图也可以实现列的访问。其他用户必须先要有数据库管理者授权可以进入含有表的数据库。

X/Open和Entry SQL-92标准的Grant命令如下：

```
[EXEC SQL] GRANT {ALL PRIVILEGES | privilege {, privilege ...})
    ON tablename | viewname
    TO {PUBLIC | user-name {, user-name ...}} [WITH GRANT OPTION]
```

ORACLE、DB2 UDB和INFORMIX都支持这种语法。X/Open和Full SQL-92也定义在整理、字符集和转换上的Grant命令，但本附录中并未对此进行介绍。Full SQL-99支持所有X/Open和SQL-92所定义的语法，并增加了在用户自定义类型和函数设计器上的Grant命令。Core SQL-99支持上面的表/视图和在非结构化的用户自定义类型及SQL调用的函数设计器上的授权。

在X/Open和Entry SQL-92中，Grant可以授予所有类型的访问特权(ALL PRIVILEGES)或一些下面列出来的特权：

```
SELECT
DELETE
INSERT
UPDATE [(columnname {, columnname . . . })]
```

```
REFERENCES [(columnname [, columnname . . . ])]
```

而在Core SQL-99则增加了在用户自定义函数和过程上的EXECUTE授权。ORACLE、DB2 UDB和INFORMIX都支持所有这些特权，并加上表的ALTER特权及一些与相关产品的附加特权。详细信息参见7.3节。

在赋予用户DBA特权或其他类似的特权时，不同的产品有不同的Grant语法。如INFORMIX使用下面语法：

```
[EXEC SQL] GRANT {CONNECT|RESOURCE|DBA}
    TO user-name [, user-name ...];
```

ORACLE使用相同的语法，并增加50多个不同的系统特权，包括DBA的SYSDBA特权。DB2 UDB的Grant有好几种形式，其中授予DBA特权的是Grant Database Authoritics语句。

C.27 Insert语句

正如3.10和5.3节所述，Insert语句在嵌入式和交互式SQL中都是一样的。Insert语句有两种形式，一种是插入具体值的一行记录，另一种是使用前面提到的子查询形式插入多行记录。X/Open和Entry SQL-92标准关于Insert语句的语法如下：

```
[EXEC SQL] INSERT INTO [schema.]tablename [(columnname [, columnname . . . ])]
    {VALUES (expr | NULL [, expr | NULL...]) | subquery}
```

新插入的行的位置不能预先设定，它的位置是由表的磁盘结构所决定的。

X/Open与Entry SQL-92的区别主要在于如何明确定义列缺省值的使用。Core SQL-99与X/Open和Full SQL-92一样，允许使用DEFAULT来代替VALUES子句中任何列值的expr | NULL值。但是，在没有指明具体值时也使用了缺省值，所以对Insert来说DEFAULT关键字对于安装缺省值并不是很重要的。

所有主要的数据库产品都支持上面的Insert基本SQL语法。在ORACLE中有插入到嵌套表的附加形式，参见4.3.1节。类似地，INFORMIX也有插入到汇集表的特殊形式。另一个产品与标准之间的差别是所允许的查询的形式不同。Full SQL-92和DB2 UDB都支持子查询如图3-10所示的高级形式，包括INTERSECT和EXCEPT操作符。ORACLE与这很接近，只是用MINUS替代EXCEPT。X/Open和INFORMIX都不允许在子查询中使用INTERSECT和EXCEPT，X/Open甚至不允许使用UNION，但INFORMIX允许。

对象-关系语法

ORACLE中允许用TABLE(subquery)来取替tablename，这样允许将行插入嵌套表中。

C.28 Open Cursor语句

这语句主要用于打开先前声明的非动态游标，我们所介绍的所有标准和产品的形式如下：

```
EXEC SQL OPEN cursor_name;
```

详细信息参见5.1节。

如用它打开动态游标（使用SQLDA结构），则在我们介绍的所有产品中使用如下语法：

```
EXEC SQL OPEN cursor_name
  USING :host_variable [, :host_variable...]
  | USING DESCRIPTOR :sqlvar_pointer;
```

可用宿主变量或描述符值替代与游标相关的准备好的SQL Select语句的WHERE子句中的动态参数。详细信息参见5.6节。在标准中使用描述符名而不是指针。

C.29 Prepare语句

在嵌入式SQL中Prepare语句用于准备含有宿主变量字符串动态SQL语句（参见Execute语句和Execute Immediate语句，后者结合了Prepare语句和Execute语句的作用）。在所介绍过的所有标准和产品中Prepare语句有如下语法：

```
EXEC SQL PREPARE statement_identifier FROM :host_string;
```

host_string的字符串内容必须是合法的SQL语句，可包括如下类型：Alter Table, Create Table, Delete, Drop (Table, View或Index), Grant, Insert, Revoke和Update。在特定产品中其他语句也可能允许。详细信息参见5.6节。

C.30 Revoke语句

在X/Open和Core SQL-99中Revoke语句用于取消表的特权：

```
[EXEC SQL] REVOKE {ALL PRIVILEGES | privilege [, privilege ...] }
  ON tablename | viewname
  FROM {PUBLIC | user-name [, user-name ...] }
  CASCADE | RESTRICT;
```

Revoke语句可以取消当前用户先前赋予一特定用户的一组特权。与Grant语句不同，Revoke语句在取消更新特权时，不能指定列名。表的拥有者自动拥有所有特权，并且它们是不能取消的。

ORACLE和DB2 UDB都没有实现CASCADE | RESTRICT子句，但INFORMIX实现了，并且它们是可选的。详细信息参见7.3节。

C.31 Rollback语句

在嵌入式SQL程序中Rollback语句用于必须中止活动事务的时候，也就是异常中止。在我们所介绍的所有标准和产品中都有以下语法：

```
[EXEC SQL] ROLLBACK [WORK];
```

当执行Rollback语句时，所有事务中所做的更新将被撤销，把先前的值放回原来位置，并让其他用户也可见。如果一个程序中止前没有执行Commit或Rollback语句，则不同的产品会采取不同的缺省操作。详细信息参见5.4节。

C.32 Select语句

交互式Select语句的所有语法如图C-27所示。语法中提到的子查询就是SQL-92和X/Open中的查询表达式。ORACLE也称其为子查询。交互式Select语法的不同子句和语法元素在第3章进行了详细的介绍。Full SQL-92和Core SQL-99都允许用EXCEPT和UNION[ALL]来连接子

查询。Full SQL-92和SQL-99的一个扩展特征则允许INTERSECT[ALL]。所有这些信息参见3.6节。

1. 通用语法元素

图C-27中子查询的表引用形式在不同的标准中是不一样的。参见图C-28。在Core SQL-99或X/Open中连接表的形式是可选的并被放在圆括号中。

```

Subquery ::=

    SELECT [ALL | DISTINCT] { * | expr [[AS] c_alias] {, expr [[AS] c_alias]...} }

        FROM tableref {, tableref...}

        [WHERE search_condition]

        [GROUP BY colname {, colname...}]

        [HAVING search_condition]

    | subquery UNION [ALL] subquery

Select statement ::=

    Subquery [ORDER BY result_column [ASC | DESC] {, result_column [ASC | DESC]...}]

```

图C-27 基本子查询和基本Select语句一般形式

```

Entry SQL-92:
tableref ::= tablename [corr_name]

Basic SQL:
tableref ::= tablename [[AS] corr_name]
X/Open: 如果使用JOIN，则NATURAL, ON和USING中的一个必须使用
tableref ::= [schema.]tablename [[AS] corr_name]                                -- simple form
    | tableref1 [NATURAL | {INNER | {LEFT|RIGHT} [OUTER]}] JOIN tableref2      -- join form
        {ON search_condition | USING (colname {, colname ...})}

SQL-99 & Advanced SQL:
tableref ::= [schema.]tablename [[AS] corr_name][(colname {, colname ...})] -- simple form
    | (subquery) [AS] corr_name [(colname {, colname...})]                      -- non-Core subquery form
    | tableref1 [INNER | {LEFT|RIGHT} [OUTER]] JOIN tableref2                  -- join form
        (ON search_condition | USING (colname {, colname ...}))
```

图C-28 表引用(tableref)的不同形式

X/Open标准中有与基本SQL和Entry SQL-92相同的谓词，如图C-29所示。在Full SQL-92中增加了OVERLAPS谓词，用于两个时间段之间是否有重叠。其取值可以是CURRENT_DATE、CURRENT_TIME或CURRENT_TIMESTAMP。这些有用的构造在x/Open中被认为是伪代码(pseudo_literals)，但在SQL 92标准中是更为严格一些的非Entry的日期时间值函数，在SQL-92中，图C-29中的子查询是不允许有UNION的。在Core SQL-99中，比较谓词一般形式是：expr1 op expr2，其中表达式expr中可有(subquery)形式。其他使用(subquery)的形式与Core SQL-99一样，只是允许在子查询中使用UNION、EXCEPT和JOIN。Core SQL-99没有额外的谓词。谓词可以组合成如图C-30所示的搜索条件。表达式的语法和其他相关信息参见3.9节。

2. 嵌入式SQL的Select语句

只有在检索结果不多于一行(0或1行都允许)时才可以在嵌入式SQL程序中执行Select语句，否则使用游标。如果Select语句在嵌入式SQL中返回多于1行，则会返回运行期错误。该语句

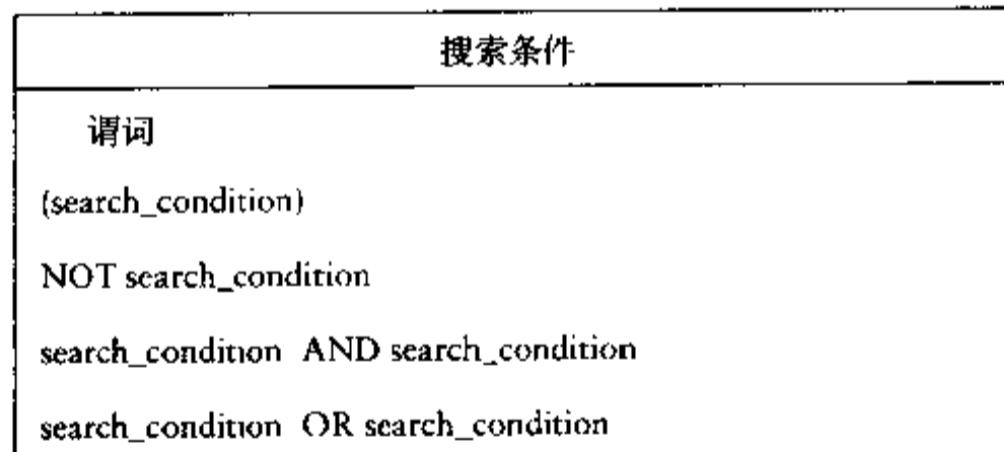
在嵌入式SQL中语法如图C-31所示。

产品间差别

这里涉及到的三种产品都支持如图C-27、C-29和C-31所示的语法，它们都使用如图C-28所示的Entry SQL-92中的表引用形式。如基本SQL中表引用形式所示，INFORMIX和DB2 UDB(但ORACLE不)都允许使用可选关键字AS，它们还支持在图C-28的SQL-99表引用形式中标记为Simple form的语法，也就是说，支持为查询中的FROM表中的列分配新名的功能。DB2 UDB和ORACLE都支持非Core SQL子查询形式，但是ORACLE不允许在加括号的子查询后加列列表，也不支持AS关键词，因此它的子查询形式非常简单为 (subquery) corr_name。在这三个产品中只有DB2 UDB支持如图C-28所示的SQL-99连接形式，但它也不支持其中的USING子句。ORACLE和INFORMIX对左右内连接都有特殊形式。

谓词	形式
比较谓词	expr1 θ {expr2 (Subquery)}
BETWEEN谓词	expr1 [NOT] BETWEEN expr2 and expr3
量化谓词	expr θ [ALL SOME ANY] (Subquery)
IN谓词	expr [NOT] IN (Subquery) expr [NOT] IN (value {, value})
EXISTS谓词	[NOT] EXISTS (Subquery)
IS NULL谓词	colname IS [NOT] NULL
LIKE谓词	colname [NOT] LIKE pattern-value [ESCAPE escape-char-value]

图C-29 基本SQL谓词



图C-30 Entry SQL-92,X/Open和Core SQL-99中搜索条件(search_condition)的递归定义

```

EXEC SQL SELECT [ALL | DISTINCT] expression {, expression}
    INTO host-variable {, host-variable}
    FROM tableref {, tableref ...}
    [WHERE search_condition];
    
```

图C-31 Select语句基本嵌入式SQL语法

3. 对象-关系语法

对象-关系扩展使用一种新的表达式语法——带点标识方法来访问结构类型的部分，创建

这些类型新的实例及引用整个行对象（在ORACLE中是VALUE(t_alias)）。在ORACLE中对象的引用使用REF()和DEREF()。关于ORACLE使用IS DANGLING来检测悬挂对象REF的信息参见4.2.1节。因为简单对象及引用主要对列所允许的类型进行了扩展，而它们一般不能直接出现在Select语句中（除非在CAST表达式中），所以Select语句所受影响较小，较大的影响来源于汇集，因为汇集提供了新的表源，即查询的主要材料。

如图C-32所示，表引用形式增加了可以从汇集中选择的功能。这里，把4.3.1节和4.3.2节的图4-22和图4-28结合在一起，此外增加了4.2.2节的ONLY关键字。ONLY关键字可以限制只从所指定的表中检索行，而不包括其继承层次上子表的行（这方面信息，参见4.2.2节的Select语句）。与INFORMIX一样，Full SQL-99有对象-关系表引用语法，它们的区别只是在INFORMIX用UNNEST替代TABLE以及一个附加的可选子句。

```

ORACLE object-relational tableref
tableref ::= [[schema. ]tablename | (rowset_valued_subquery)
             | TABLE(collection_expression)]
             [corr_name]                                -- "AS" keyword not allowed

INFORMIX object-relational tableref
tableref ::= [schema. ]tablename                -- no relational (subquery) form
             | TABLE(collection_expression)
             | ONLY ([schema. ]tablename)           -- restriction in table hierarchy
             [[AS] corr_name [(columnname, {columnname...})]]

```

图C-32 包括对象-关系扩展的ORACLE和INFORMIX表引用语法

INFORMIX扩展IN谓词为IN collection_expression的形式并提供了CARDINALITY(collection_columnname)(用于计算汇集列值中元素的个数)。ORACLE则提供用于交互选择的CURSOR()表达式，它允许显示外表中每一行的嵌套表的行，并提供新的CAST形式转化汇集类型。

C.33 Update语句

与Delete语句一样，Update语句也有两种形式：一种是查找更新，可在交互或嵌入式SQL中执行的可一次更新多行记录的Update语句；另一种是定位更新，它只能通过游标操作并只在嵌入式SQL中有效。基本的查找Update语法如图C-33所示，而定位Update语法则在图C-34中。详细信息参见5.3节。

```

[EXEC SQL] UPDATE [schema. ]tablename
    SET columnname = expr | NULL | (subquery)
    [, columnname = expr | NULL | (subquery)...]
    [WHERE search_condition];

```

图C-33 搜索Update语句基本语法

几乎所有主要的数据库产品都支持Update语句的基本语法，它们也同时遵守Full SQL-92和Core SQL-99。但是，在X/Open和Entry SQL-92标准中并没有新的列值指定的(subquery)选项。由于它们也不允许(subquery)量化为表达式，所以这个选项是不符合的。而Full SQL-92和Core SQL-99则扩展了表达式的含义使其包括(subquery)。在这些情况下，可以删除单独的(subquery)选项，或者注意到它是多余的。

```
EXEC SQL UPDATE [schema.]tablename  
    SET columnname = expr | NULL | (subquery)  
        [, columnname = expr | NULL | (subquery)...]  
    WHERE CURRENT OF cursor_name;
```

图C-34 嵌入式定位Update语句基本语法

对象-关系语法

ORACLE允许用TABLE(subquery)来替代表名，这里子查询返回嵌套表的值，这样允许更新嵌套表的行。在INFORMIX中可用ONLY ([schema.] tablename)来替代表名，用来限制更新只能在指定的表上进行，而不更新表层次中子表的行。

附录D 集合查询计数

下面列出了在集合查询基准程序中从查询Q1到Q6B的返回的行数。该列表对实践人员会有所帮助。在任何平台上如果正确产生和加载数据，这些数据应该都是一样的。

查询	实例	行数	总结
Q1	KSEQ	1	
Q1	K100K	8	
Q1	K10K	98	
Q1	K1K	1003	
Q1	K100	10 091	
Q1	K25	39 845	
Q1	K10	99 902	
Q1	K5	200 637	
Q1	K4	249 431	
Q1	K2	499 424	
Q2A	KSEQ	1	
Q2A	K100K	5	
Q2A	K10K	58	
Q2A	K1K	487	
Q2A	K100	5009	
Q2A	K25	19 876	
Q2A	K10	49 939	
Q2A	K5	100 081	
Q2A	K4	125 262	
Q2B	KSEQ	499 423	
Q2B	K100K	499 419	
Q2B	K10K	499 366	
Q2B	K1K	498 937	
Q2B	K100	494 415	
Q2B	K25	479 548	
Q2B	K10	449 485	
Q2B	K5	399 343	
Q2B	K4	374 162	
Q3A	K100K	1	434
Q3A	K10K	9	5513
Q3A	K100	991	496 684
Q3A	K25	3989	1 978 118
Q3A	K10	9920	4 950 698
Q3A	K5	20 116	10 027 345
Q3A	K4	24 998	12 499 521
Q3B	K100K	1	434
Q3B	K10K	6	3300

(续)

查询	实例	行数	总结
Q3B	K100	597	299 039
Q3B	K25	2423	1 209 973
Q3B	K10	5959	2 967 225
Q3B	K5	12 000	5 980 617
Q3B	K4	15 031	7 496 733
Q4A	1-3	10 059	
Q4A	2-4	4027	
Q4A	3-5	1637	
Q4A	4-6	4021	
Q4A	5-7	7924	
Q4A	6-8	10 294	
Q4A	7-9	4006	
Q4A	8-10	785	
Q4B	1-5	161	
Q4B	2-6	86	
Q4B	3-7	142	
Q4B	4-8	172	
Q4B	5-9	77	
Q4B	6-10	76	
Q4B	7-1	152	
Q5	K2=K100=1	4962	
Q5	K4=K25=1	9970	
Q5	K10=K25=1	4049	
Q6A	K100K	23	
Q6A	K40K	55	
Q6A	K10K	239	
Q6A	K1K	2014	
Q6A	K100	19 948	
Q6B	K40K	3	
Q6B	K10K	4	
Q6B	K1K	81	
Q6B	K100	804	

习题解答

第2章习题解答

2.1 (a) A和B都是键，因为作为单个属性它们在每一行的值都不相同。没有其他键可以包含A或者B。C或者D都不能把所有行都区别开，但是把它们放在一起可以做到这一点。所以，T1的三个候选键是A、B以及CD。

(c)

A	B	C	D
a1	b1	c1	d1
a2	b1	c1	d1
a1	b2	c1	d1
a1	b1	c2	d1

很明显，D不是任何键的一部分，且ABC是一个超键。任何单独的列：A、B或C都不是键。这是因为每一列都有三个相同的项。列对AB、BC、或CA也都不是键，这是因为它们都有重复的项。所以，ABC是唯一的键。

2.2 (a) 三个候选键是：ssn, name information-no和name address city zip。

2.4 (a) `(ORDERS where qty >= 1000)[ordno,pid]`.

圆括号是必须的，这是因为投影的优先权较高。

(c) `(ORDERS where dollars < 500 JOIN CUSTOMERS) [ordno,cname]`.

在ORDERS表和CUSTOMERS表间只有一个属性是相同的，所以连接运算匹配这一相同的列，并扩展ORDERS表以包含客户信息。然后投影到ordno和cname列。如果我们能够在连接前用投影去掉无用的ORDERS表中的列，结果同样正确：

`((ORDERS where dollars < 500)[ordno,cid] JOIN CUSTOMERS)
[ordno,cname]`

(e) `((ORDERS where month = 'mar' JOIN CUSTOMERS)[ordno,cname,aid]
JOIN AGENTS)[ordno,cname,aname]`

注意，与前两个练习不同，在这里，第一个投影是防止根据属性city连接而发生不必要的匹配所必须的。这是因为属性city既出现在`(ORDERS where month = 'mar' JOIN CUSTOMERS)`中，又出现在AGENTS中。如果不使用第一个投影以去除city属性，那么只有能匹配客户的city和代理商的city的行才会出现在结果中。

(g) `(PRODUCTS where city = 'Duluth' JOIN ORDERS
where month = 'mar')[pname]`.

同样，在连接之前的投影不会影响结果，除非它去掉了pid。

2.5 (a) `((CUSTOMERS TIMES AGENTS) TIMES PRODUCTS)
where CUSTOMERS.city = AGENTS.city`

```
and AGENTS.city = PRODUCTS.city)[cid,aid,pid]
```

由于笛卡儿积满足结合律，所以也可以使用不指明三个表的乘积哪个乘积先执行的方法解答：

```
((CUSTOMERS TIMES AGENTS TIMES PRODUCTS)
  where CUSTOMERS.city = AGENTS.city
  and AGENTS.city = PRODUCTS.city)[cid,aid,pid]
```

(c) ((CUSTOMERS TIMES AGENTS TIMES PRODUCTS)
 where CUSTOMERS.city <> AGENTS.city
 and AGENTS.city <> PRODUCTS.city
 and PRODUCTS.city <> CUSTOMERS.city)[cid,aid,pid]

你知道为什么需要第三个条件吗？ $x \neq y$ 且 $y \neq z$ 并不意味着 $x \neq z$ 。例如： $x=c, y=k, z=c$ 。

(e) (ORDERS JOIN (CUSTOMERS where city = 'Dallas')[cid] JOIN
 (AGENTS where city = 'Tokyo')[aid] JOIN PRODUCTS)[pname]

表CUSTOMERS和AGENTS的投影是用来去掉city列的。这样，在连接时city列就不会被考虑了。同样，我们没有用括号指定四个表的连接顺序，这是因为和笛卡儿积一样，JOIN也满足结合律。

(g) A1 := AGENTS. A2 := AGENTS
 ((A1 X A2) where A1.city = A2.city and A1.aid < A2.aid)
 [A1.aid, A2.aid]

这和正文中我们只希望每对不同的代理只出现一次是类似的。

(i) 如果以前并不知道该诀窍，此题是比较难的。让：

```
C1 := CUSTOMERS. C2 := CUSTOMERS
X(cid1, cid) :=
((C1 TIMES C2) where C1.discount >= C2.discount) [C1.cid, C2.cid].
```

现在，得到的答案正好是cid1的值（在左）和所有cid的值（在右）所组成的对，即包含了discount和所有小于等于它的discount的值——所以最终答案是：

X DIVIDE BY C[cid].

为了找到具有最少折扣的客户，只要把上面的 \geq 改成 \leq 就可以了。

(k) (ORDERS where ORDERS.aid = 'a03')[pid]
 -(ORDERS where ORDERS.aid = 'a06')[pid]

(m) 首先，我们可以找到不订购任何Newark的产品的代理T：

```
T := all agents - agents who order products in Newark
= AGENTS[aid] - (O JOIN PRODUCTS where city = 'Newark')[aid]
```

现在，(T JOIN AGENTS)[aid,aname]加上了aname的信息。于是，我们可以选择这些aname：

```
(T JOIN AGENTS)[aid,aname] where aname >= 'N' and aname < 'O'
```

最后的结果是通过代入法得到的。

(o) 设P1等于用户c002订购的所有产品，即P1 := (ORDERS where cid = 'c002')[pid]。接着，我们需要得到为所有订购P1的代理，他们的

aid为：

$\text{ORDERS}[\text{aid}, \text{pid}] \text{ DIVIDE BY } \text{P1}$

所以，通过代入得到的答案是：

$((\text{ORDERS}[\text{aid}, \text{pid}] \text{ DIVIDE BY } (\text{ORDERS where cid = 'c002'})[\text{pid}])$
 $\text{JOIN AGENTS})[\text{aname}]$

这里，两次使用ORDERS的用法似乎是可疑的，但是，事实上它不会引起任何问题：只有当两个表的列名在计算中要一起被用到时才需要使用别名。

- (q) 是的，这和(b)是一样的。使用JOIN：

$(\text{CUSTOMERS TIMES AGENTS TIMES PRODUCTS})[\text{cid}, \text{aid}, \text{pid}] \sim (\text{CUSTOMERS JOIN AGENTS}$
 $\text{JOIN PRODUCTS})[\text{cid}, \text{aid}, \text{pid}]$

- (s) $((\text{ORDERS where dollars} > 500)[\text{aid}, \text{cid}] \text{ JOIN CUSTOMERS}$
 $\text{where city = 'Kyoto'})[\text{aid}]$

- (u) 设 $OY := \text{ORDERS}$ ，且 $O := \text{ORDERS}$ 。于是答案是：

$O[\text{cid}] \sim ((O \times OY) \text{ where } O.\text{aid} \neq OY.\text{aid} \text{ and } O.\text{cid} = OY.\text{cid})[O.\text{cid}]$

在剩下的习题解答里（2.6到2.15）包括很多键盘形式的关系运算符，以示范它们的用法。

- 2.6 (b) 在定义2.7.4中，如果没有相同的列，那么在两个表中 B_1 到 B_k 就不存在 ($k=0$)。于是，在定义何时行 t 在表 R JOIN S 时，我们应该忽略对列 B_i 的引用。于是，对于 R TIMES S 的乘积使用合适的列名代换，则剩下的定义就退化 B_i 为定义2.6.4。另一个论据在定理2.8.2中，使用其他运算符来表示JOIN。如果我们去掉对列 B_i 的引用，那么定义的第一部分就退化为 $T := R$ TIMES S 。定义 T_2 的第二部分仅仅是重命名列以去除限定符。

- 2.7 通过练习2.6，我们知道，当 R 和 S 没有公共的属性（列）时， R JOIN S 和 R TIMES S 是完全相同的。设 r 是 R 中的一行。显然 r 会和 S 中所有的行相配以构成 R TIMES S 中的某些行。在这种情况下，如果被 S 除，我们就又能得到 r 了。从中我们可以看到，每一个 R 中的行都在 $(R$ JOIN $S)$ DIVIDE BY S 中。但是，如果另一行 u 在 $(R$ JOIN $S)$ DIVIDE BY S 中出现，这意味着 u 在 $(R$ JOIN $S)$ 中肯定和 S 的所有行相配，即 u 在 $(R$ TIMES $S)$ 中。但是，根据TIMES的定义，这样的行 u 一定在 R 中。这是因为只有这样的行才会和 S 中的行相配并被放入 R TIMES S 中。要完整、严密地证明需要形式化一些意义不明确的表述，例如用定义2.6.4的形式表示“ r 和 S 中所有的行相配”：“对于 R 和 S 中的相应的每一对行 u 和 v ，在 R TIMES S 中存在一行 t ，满足 $t(R.Ai) = u(Ai)$ ，且……”

- 2.8 (a) 设 N_A 是一个常数表，它只有一行，所有值为null，标题是由在 A 中但不在 B 中的属性组成的。类似地， N_B 是在 B 中但不在 A 中的属性为null的表。于是：

$$A \bowtie_Q B = A \bowtie B \cup N_A \times (B - (A \bowtie B)[\text{Head}(B)]) \cup (A - (A \bowtie B)[\text{Head}(A)]) \times N_B$$

- 2.9 (a) 对于三个表 R 、 S 和 T ，在表和表之间，属性共用有很多种可能：可能是 R 和 S 、 S 和 T 或者 T 和 S 之间有共用的属性。而且有些属性可能是三个表所共有的。 R JOIN S 可以得到在 R 和 S 的共有属性上相同的行。而 $(R$ JOIN $S)$ JOIN T 可以得到在 T 和 R JOIN S 的共有属性——既是 T 和 R 的共有属性，又是 T 和 S 的共有属性——上相同的行。显然，结果在 $R - S$ 、 $S - T$ 和 $T - S$ 三个属性集上都要一致。类似地， R JOIN $(S$ JOIN $T)$ 也有相同的要求。在任何一种情况下，结果行都是把 $R \times S \times T$ 中的行进行投影以去掉重复（在两个表中重复或者在三个表中都重复）的列以后，进行选择而得到的。

2.11 表S where C包含了S中所有满足条件C的行。于是：

r在(S where C)中，当且仅当r在S中，而且对于r，C为真。

r在((R where C1) where C2)中当且仅当 r在(R where C1)中，而且对于r，C2为真。

(R where C1)是R中使C1为真的行的集合。

于是，r在((R where C1) where C2)中当且仅当 r在R中，且C1和C2为真。

类似地，其他表达式也可以被简化为同样的基本形式。

2.13 通过前一个练习，可以知道兼容表的连接就是表的交。考虑在R INTERSECT T中的任意x。因为R是S的子集，所以R INTERSECT T是S INTERSECT T的子集。

考虑U DIVIDEBY S中的任意x。这意味着对于所有的在S中的s，(x,s)在U中。接着可以知道，对于所有的在R中的s，(x, s)在U中。这是因为R是S的子集。所以，x在U DIVIDEBY R中。

考虑R DIVIDEBY V中的任意x。这意味着对于所有的在V中的v，(x,v)在R中。接着可以知道，对于所有的在V中的v，(x,v)在S中。这是因为R是S的子集。所以，x在S DIVIDEBY V中。

考虑(R where C)中的任意x。x在R中，且对于x，C为真。于是，x在S中，而且对于x，C为真。所以，x在(S where C)中。

2.15 (b) 对于在(R ∪ S)[H]中的任意x，在R ∪ S中存在某个(x,x')，所以(x,x')在R中或者S中。当它们被单独投影的时候，x或者在R[H]中，或者在S[H]中。所以它在它们的并集中。类似地，任何R[H] ∪ S[H]中的x在R[H]中或S[H]中。所以存在R或者S中（即在R ∪ S中）的某个(x,x')投影为x。于是，x在(R ∪ S)[H]中。

第3章习题解答

3.1 (a)

```
select c.cid, a.aid, p.pid
      from customers c, agents a, products p
     where c.city = a.city and a.city = p.city;
```

(c)

```
select c.cid, a.aid, p.pid
      from customers c, agents a, products p
     where c.city <> a.city and a.city <> p.city
       and c.city <> p.city;
```

注意，即使其中两个条件为真，第三个条件仍然可能为假。所以必须使用三个条件的与 (AND)。

(e)

```
select pname from products where pid in
    (select pid from orders where
      cid in (select cid from customers where city = 'Dallas') and
      aid in (select aid from agents where city = 'Tokyo'));
```

从中可以看到，我们在同一个WHERE子句中使用了两个IN条件。我们还可以使用连接来达到相同的目的：

```
select distinct pname
      from products p, orders o, customers c, agents a
     where p.pid = o.pid and o.cid = c.cid and c.city = 'Dallas'
       and o.aid = a.aid and a.city = 'Tokyo';
```

- (g) select a1.aid, a2.aid from agents a1, agents a2
 where a1.city = a2.city and a1.aid < a2.aid;
- (i) select cid from customers
 where discnt >= all
 (select discnt from customers);

 select cid from customers
 where discnt <= all
 (select discnt from customers);
- (k) select distinct pid from orders where aid = 'a03'
 and pid not in (select pid from orders where aid = 'a06');

(m) 表面上看，似乎使用如下的方法就足够了：

```
select distinct a.aid, a.aname from agents a, orders o
  where a.aname like 'N%' and a.aid = o.aid
  and o.pid not in (select pid from products
    where city = 'Newark');
```

但是，事实上它仅仅列出了提供订单的代理商。完整的解答应该是：

- ```
select a.aid, a.aname from agents a where a.aname like 'N%'
 and a.aid not in
 (select o.aid from orders o where o.pid in
 (select pid from products where city = 'Newark'));
```
- (o)    select distinct a.name from agents a where not exists
       (select \* from orders x where x.cid = 'c002' and not exists
           (select \* from orders y where y.aid = a.aid
             and x.pid = y.pid));
- (q)    select cid, aid, pid from customers c, agents a, products p
       where c.city <> a.city or c.city <> p.city
       or a.city <> p.city;
- (s)    select distinct aid from customers c, orders o
       where c.cid = o.cid
       and o.dollars > 500 and c.city = 'Kyoto';

- 3.2 (a)    select aid from agents where percent > any
           (select percent from agents);

(c) 只有在最小佣金为5的时候，引用的查询返回和(a)相同的结果。此后，结论就可能不成立了。

这就是为什么我们在图2-2中说：“某个时刻的内容”。它也通过内容依赖——两个查询对于同一个表的内容有相同的结果并不足以保证两个查询是等价的。解释了例2.7.6。

- 3.3 (a) 我们知道'x in y'和'x = any y'是一样的（其中y是由子查询返回的一组值，而x是一个简单值）。但是，'x not in y'和'not (x in y)'是一样的，而'not (x = any y)'和'x not in y'是不等价的。这是因为如果在y中有一个值不等于x，则'x <> any y'即为真，但是只有当x不等于y中的任何值的时候，'not (x = any y)'才为真。此条件在SQL中用谓词'x <> all y'表示。于是，NOT ALL和<>ALL是等价的，但是和<>ANY是不同的。
- (c) 正好检索那些例3.4.7的查询不检索的行的查询是：

```
select aid from agents where percent >any
(select percent from agents);
```

3.4 (a) select distinct cid from orders, agents  
       where orders.aid = agents.aid  
       and (agents.city = 'Duluth' or agents.city = 'Dallas');

(c) select A<sub>1</sub>,...,A<sub>n</sub> from S, T where S.A<sub>2</sub> = k and  
       S.A<sub>1</sub> = T.B<sub>1</sub> and T.B<sub>2</sub> = c and T.B<sub>3</sub> = S.A<sub>3</sub>;

3.6 (a) INITIALIZE LIST L to EMPTY  
       FOR T FROM ROW 1 TO LAST OF TABLE T (No alias)  
           IF (T.B<sub>2</sub> = c)  
               Place B<sub>1</sub> on LIST L;  
       END FOR T;  
       FOR S FROM ROW 1 TO ROW LAST OF TABLE S  
           IF (A<sub>2</sub> = k and A<sub>1</sub> an element of LIST L)  
               Place (A<sub>1</sub>,...,A<sub>n</sub>) on SOLUTION LIST M  
       END FOR S  
       PRINT OUT UNIQUE ROWS ON SOLUTION LIST M

3.7 (a) select \* from R union select \* from S;

(c) 用CROSS JOIN替换每一个PRODUCT，并用括号把UNIONS和DIFFERENCES中的子表达式括起来，让它变成一个tableref(如图3-10定义的那样)。于是，整个表达式就成为了一个tableref T。构造SELECT语句select \* from T。  
(R UNION S) MINUS T在Full SQL-92表示成为：

```
select * from ((R UNION S) EXCEPT T).
```

(R UNION S) MINUS T表示成基本的SQL：考虑(R UNION S) MINUS T中的一行 w。它在R或S中，但是不在T中。假设它在R中。于是，它在R中但不在T中。类似地，如果它在S中，那么它在S中但不在T中。这样，它在R - T中或S - T中，即它在(R - T) UNION (S - T)中。此表达式可以用习题3.7(a)的结果写成SQL形式：

```
(select * from R where not exists (select * from T
 where R.A1 = T.A1 and R.A2 = T.A2... and R.An = T.An))
UNION
(select * from S where not exists (select * from T
 where S.A1 = T.A1 and S.A2 = T.A2... and S.An = T.An))
```

(e) 根据定理2.8.3，

```
R DIVIDEBY S = R[A1,...,An] - ((R[A1,...,An] TIMES S)
 - R)[A1,...,An]
```

其中，R为select cid, aid from orders，而S为select aid from agents where city = 'New York'。HEAD(R) = A<sub>1</sub>,...,A<sub>n</sub>，B<sub>1</sub>,...,B<sub>m</sub>=cid aid，而S=B<sub>1</sub>,...,B<sub>m</sub>=aid。所以我们可以知道，n=1，A<sub>1</sub>=cid，m=1，B<sub>1</sub>=aid，而且R[cid] TIMES S(设为V)的定义如下：

```
V = select o.cid,a.aid from orders o, agents a
 where city = 'New York'
```

我们需要的最后结果是 $R[cid] - (V - R)[cid]$ 。根据习题3.7(d)中的MINUS，

```
(V-R) = select o.cid, a.aid from orders o, agents a
 where city = 'New York' and not exists
 (select * from orders x
 where x.aid = a.aid and x.cid = o.cid)
```

$(V - R)[cid]$ 也类似可以得到，只要在选择列表中去掉 $a.aid$ 就可以了。最后，通过比习题3.7(d)多使用一次MINUS，可以得到结果：

```
select cid from orders y where not exists
 (select o.cid from orders o, agents a
 where city = 'New York'
 and not exists
 (select * from orders x
 where x.cid = o.cid and x.aid = a.aid)
 and y.cid = o.cid);
```

事实上，这样返回的结果包含了重复了多次的内容，所以要求使用`select distinct`。

3.8 (a) `select cid, max(dollars) as maxspent from orders
group by cid;`

3.10 (a) 显然，这是一个外连接。我们需要使用WHERE C来从10个客户中导出一张表，假定为ctab。然后把ctab和sporders右外连接。然后根据cid把订单求和。下面是具体的步骤：

1) 创建导出表ctab：

```
(select cid, cname from customers where C) ctab
```

2) 把ctab和sporders右外连接，产生如下的表引用：

```
(select cid, cname from customers where C) ctab
 right join sporders on ctab.cid = sporders.cid
```

3) 在最终的查询中使用表引用：

```
select cid, cname, sum(dollars) as totdoll from
 ((select cid, cname from customers where C) ctab
 right join sporders on ctab.cid = sporders.cid)
 group by cid, cname;
```

注意，在GROUP BY列表中，我们需要使用cname以使它可以出现在选择列表中。有时，它可能为null，但是那没有关系。不会因为有为null的cname而引起一些意外的情况——不会存在一个既为null又为非null的cname！在ORACLE中，我们可以使用导出表ctab作为表引用来进行单向连接。我们使用(+)来表示不保留的一边。在此，我们要保留在sporders这一边的行，所以我们把(+)放在ctab一边。

```
select cid, cname, sum(dollars) as totdoll from
 (select cid, cname from customers where C) ctab, sporders
 where ctab.cid(+) = sporders.cid
 group by cid, cname
```

我们可以用内连接和额外行的并来模拟外连接。

1) 构造内连接：

```
select cid, cname, sum(dollars) as totdoll from customers c, sporders
```

```

 where C and ctab.cid = sporders.cid
 group by cid, cname;

```

2) 表示外连接中在sporders中但不在ctab中的额外的行:

```

select cid, null, sum(dollars) as totdoll from orders
 where cid not in (select cid from customers where C) ctab

```

3) 把第一步和第二步的结果并 (UNION) 起来。

3.11 (a) select aid, pid, sum(qty) from orders group by aid, pid;

(c) select a.aid from agents a where not exists
 (select o.\* from orders o, customers c, products p
 where o.aid = a.aid and o.cid = c.cid and o.pid = p.pid
 and c.city = 'Duluth' and p.city = 'Dallas');

(e) select distinct cid from orders o
 where aid in ('a03', 'a05') and not exists
 (select \* from orders x where
 o.cid = x.cid and x.aid not in ('a03', 'a05'));

(g) select aid from agents
 where percent = (select max(percent) from agents);

(i) update agents set percent = 11 where fname = 'Gray';
 select \* from agents;
 update agents set percent = 6 where fname = 'Gray';
 select \* from agents;

(k) select cid, sum(dollars) from orders where aid = 'a04'
 and cid not in (select cid from orders
 where aid <> 'a04') group by cid;

(m) select o.pid from orders o, customers c, agents a
 where o.cid = c.cid and o.aid = a.aid and c.city = a.city;

3.12 (a) select discnt from customers where city = 'Duluth':

```

discnt

10
8

(2 rows)

select percent from agents where city like 'N%';

```

```

percent

6
6
6

(3 rows)

select city from customers where discnt >=all
 (select discnt from customers where city = 'Duluth');

```

city

```

Duluth
Dallas

(2 rows)

select city from agents where percent >any
 (select percent from agents where city like 'N%');

city

Tokyo

(1 row)

select city from customers where discnt >=all
 (select discnt from customers where city = 'Duluth')
union
select city from agents where percent >any
 (select percent from agents where city like 'N%');

city

Dallas
Duluth
Tokyo

(3 rows)

```

- 3.13 (a) 通过定义可以知道，超键是值可以唯一地确定每一行的列的集合：在这些列上，没有两行的值是一样的。这样，如果我们从表中选择包含超键的列，那么，结果中任意两行都是不同的。

- (c) 这一命题不为真。如下查询

```
select count(*) from table group by <key>;
```

(其中<key>为包含表的任何一个超键的列的列表（用逗号分隔）) 总是产生一张和原表一样长度的表。该表只有一列，且每行的值都为1。

- 3.14 (a) `select * from customers where cname >= 'A' and cname < 'B';`

- 3.15 (a) 在外连接中，对于每个没有订单的代理需要有一行，对于每个在agents表中没有对应aname的aid的订单需要有一行：

```

select a.aname, a.aid, sum(x.dollars) from agents a, orders x
 where a.aid = x.aid group by a.aid, a.aname
union
select aname, aid, 0 from agents
 where aid not in (select aid from orders)
union
select 'null', aid, sum(x.dollars) from orders x
 where aid not in (select aid from agents) group by aid;

```

- 3.16 (a) 答：它们构成了单独的组。输出为：

```
percent
```

6

7

(4 rows)

- (c) (i) “获得佣金等于或高于所有Geneva的代理的佣金的代理商的名字”。(ii)和null的比较结果为UNKNOWN，所以返回0行。在Geneva根本没有代理商的情况下，结果将列出所有的代理商（见习题3.14）。所以，空值的存在将会严重影响结果。

3.17 (a) 该查询返回所有的顾客ID。

3.18 (a) 使用ORACLE的floor()函数：

```
create table buckets (bucket int, primary key(bucket));
insert into buckets select distinct floor(dollars/500) from orders; -- ORACLE
select bucket*500 as "range-start", sum(dollars) from orders o, buckets b
 where floor(o.dollars/500) = b.bucket
 group by b.bucket;
drop table buckets; -- only needed for query
```

对于使用CAST的系统，可以使用CAST(dollars/500 as int)来代替floor()。

3.19 (b) 中值可以利用产生数据的累积直方图的能力来直接计算。也就是说对于每个值，计算小于该值（或者等于它）的值的个数。对于orders表中的dollars，累积直方图可以用如下的高级SQL查询来产生：

```
select h2.dollars, sum(ct) from
 (select dollars, count(dollars) ct from orders group by dollars) h1,
 (select dollars from orders group by dollars) h2
 where h1.dollars <= h2.dollars -- over all h1 values <= h2 value
 group by h2.dollars
 order by h2.dollars; -- just for reporting
```

内部的两个子查询把重复的dollars数值精简为值的计数，以避免连接时重复行所引起的乘法效应。现在，我们把得到的渐增直方图表用符号F表示。它的列为dollars和ct。在这里，ct是在orders表中小于等于某dollars值的计数。对于中值，我们需要得到ct的中间点，即orders表中 $ct=\max(ct)/2=\text{count}(dollars)/2$ 的dollars值。但是，我们不太可能得到准确的匹配。所以，我们要得到的是大于等于 $\text{count}(dollars)/2$ 的ct值中最小点所对应的dollars：

```
select dollars from F
 where ct = (select min(ct) from F
 where ct >= (select count(*)/2 from orders))
```

把两次出现的F替换掉，我们就得到了完整的查询。它包含两层的FROM（子查询）：

```
(select dollars from
 (select h2.dollars, sum(h1.ct) from
 (select dollars, count(dollars) ct from orders group by dollars) h1,
 (select dollars from orders group by dollars) h2
 where h1.dollars <= h2.dollars group by h2.dollars) F
 where ct = (select max(ct) from
 (select dollars, count(dollars) ct1 from orders group by dollars) h1,
 (select dollars from orders group by dollars) h2
 where h1.dollars <= h2.dollars group by h2.dollars) F
 where ct1 >= (select count(*)/2 from orders)):
```

## 第4章习题解答

- 4.1 (a) select ssno from people p where p.pname.mi is null;  
 (c) select p1.ssno, p2.ssno from people p, people p1  
       where p1.pname = p2.pname and p1.ssno < p2.ssno;

- 4.2 (a) **ORACLE:**

```
create type student_t as object (
 sperson person_t,
 stuid int,
 emailname varchar(30),
 gradyear int
);
create table students of student_t (sperson not null, primary key(stuid));
```

**INFORMIX:**

```
create row type student_t (
 sperson person_t not null,
 stuid int,
 emailname varchar(30),
 gradyear int
);
create table students of type student_t (primary key (stuid));
```

- 4.3 (a, c, g, i, q) 它们与orders表无关，所以不能使用REF。这是因为所有的REF都在表orders中。

- (e) select distinct o.ordprod.pname from orders o where o.ordcust.city = 'Dallas'  
     and o.ordagent.city = 'Kyoto';

(g, i) 与orders表无关。

(k) 只和orders表有关，不需要REF。

- (m) select aid, fname from agents a where a.fname like 'N%'  
     and a.aid not in  
         (select o.aid from orders o where o.ordprod.city = 'Newark');

(o) 为了去掉没有任何订单的代理商，我们应该在第一行中使用from orders:

```
select distinct o.ordagent.fname from orders o where not exist
 (select * from orders x where x.cid = 'c002' and not exists
 (select * from orders y where y.ordagent = ref(a)
 and y.ordprod = x.ordprod))
```

- (s) select aid from orders o where dollars > 500.00 and o.ordcust.city = 'Kyoto';

- (u) select cid from (select distinct cid, aid from orders) t  
     group by cid having count(\*) = 1;

- 4.4 (a) **ORACLE:** select e.dependents from employees e  
                   where l = (select count(\*) from table(e.dependents));

**INFORMIX:** select e.dependents from employees e  
                   where cardinality(e.dependents) = 1;

(c) **ORACLE:** select count(\*) from phonebook pb  
where 100 in (select \* from table(pb.extensions));

**INFORMIX:** select count(\*) from phonebook pb  
where 100 in pb.extensions;

4.5 (a) **ORACLE 或 INFORMIX:** select eid from employees e where e.fname in  
(select t.fname from table(e.dependents) t);

(c) **ORACLE:** select value(d) from table  
(select e.dependents from employees e where e.eid = 100) d  
where d.age =  
(select max(age) from table(select e.dependents from  
employees e where e.eid = 100));

**INFORMIX:** select d from table  
(select e.dependents from employees e where e.eid = 100) d  
where d.age =  
(select max(age) from table (select e.dependents from  
employees e where e.eid = 100));

(e) 我们可以在类似于例4.3.13的查询中使用<SOME和>SOME。但是一种更普通的方法是在内部扫描extensions表时使用条件“where extension is between 800 and 900”。但是，在两种产品中，我们都会碰到一个问题：对内部extensions表的那一列并没有指定列名。在两种产品中解决这个问题的办法是完全不同的。在ORACLE中，当没有指定列名时，可以使用通用名column\_value。INFORMIX支持SQL-99的语法，它允许像在tableref形式（图3-11第一行）中的表那样指定新的列名。所以，我们可以在查询中提供合适的列名（例如下面的ext）。

**ORACLE:** select pb.phperson.ssno from phonebook pb  
where exists (select \* from table(pb.extensions)  
where column\_value between 800 and 900);

**INFORMIX:** select pb.phperson.ssno from phonebook pb  
where exists(select \* from table(pb.extensions)  
as x(ext) -- column name "ext" added here  
where x.ext between 800 and 900);

4.6 (a) **ORACLE:**

```
select p.x, p.y, count(*) as ct from points p, rects r
 where r.inside(value(p)) > 0
 group by p.x, p.y
 having count(*)>3
 order by count(*);
```

**INFORMIX:**

```
select p, count(*) as ct from points p, rects r
 where inside(r,p)
 group by p
 having count(*)>3
 order by count(*);
```

## (c) ORACLE:

```
select value(p) from points p, rect r
 where r.inside(value(p))>0 and r.area()-
 -(select max(r.area()) from rect r);
```

## INFORMIX:

```
select p from points p, rect r
 where inside(r,p) and area(r)-
 (select max(area(r)) from rect r);
```

4.7 select value(p), value(r), (r.pt2.x-r.pt1.x)\*(r.pt2.y-r.pt1.y) from  
 rect r, points p  
 where (r.pt2.x-r.pt1.x)\*(r.pt2.y-r.pt1.y) =  
 (select min((rl.pt2.x-rl.pt1.x)\*(rl.pt2.y-rl.pt1.y)) from rectangles rl  
 where p.x >= rl.pt1.x and p.x <= rl.pt2.x and  
 p.y >= rl.pt1.y and p.y <= rl.pt2.y)  
 and p.x >= r.pt1.x and p.x <= r.pt2.x and p.y >= r.pt1.y and p.y <= r.pt2.y  
 order by p.x, p.y;

## 第5章习题解答

**注意** 大多数程序都使用文件prompt.c以及相应的头文件prompt.h。请参考附录B“编程细节”以获得更深入的讨论以及函数prompt()和函数print\_dberror()的代码。

## 5.2

```
/* ORACLE version: program to report on product quantities by agent for cid and pid */
#define TRUE 1
#include <stdio.h>
#include <string.h>
#include "prompt.h"
exec sql include sqlca;
main()
{
 exec sql begin declare section;
 char username[20], password[20];
 char cust_id[5], product_id[4], agent_id[4];
 long total_qty; /* sum of shorts may need long */
 exec sql end declare section;
 char promptcidpid[] = "PLEASE ENTER CUSTOMER ID AND PRODUCT ID: ";
 exec sql declare agent_qty cursor for /* declare cursor */
 select aid, sum(qty) from orders
 where cid = :cust_id and pid = :product_id
 group by aid;
 strcpy(username, "scott"); /* set up for */ */
 strcpy(password, "tiger"); /* ORACLE login */ */
 exec sql connect :username identified by :password; /* ORACLE connect */ */
 while (prompt(promptcidpid, 2, cust_id.arr, 4, product_id.arr, 3) >= 0) {
 exec sql whenever sqlerror goto report_error; /* error trap condition */ */
 exec sql open agent_qty; /* open cursor */ */
 exec sql whenever not found goto fetchdone;
```

```

 while (TRUE) { /* loop to fetch rows */
 exec sql fetch agent_qty into :agent_id, :total_qty;
 printf("%s %d\n", agent_id, total_qty); /* display row */
 }
 fetchdone:
 exec sql close agent_qty; /* close cursor when fetches done */
 exec sql commit work; /* release read locks */
 }
 exec sql commit release; /* end of prompt loop: repeat */
 return 0;
report_error:
 print_dberror(); /* print out error message */
 exec sql rollback release; /* ORACLE failure disconnect */
 return 1;
}

```

- 5.6 (d) 如果在读以后进行提交，那么这些读的内容在更新之前有可能发生变动，从而引起检查所要防止的问题。例如，保持存货总数为非负的。所以，在其中没有其他地方可以加入额外的Commit。

### 5.10

```

/* 4.10, DB2 UDB version: Histogram of dollar amounts of orders */
#define TRUE 1
#define RANGE 500.0
#include <stdio.h>
exec sql include sqlda;
main()
{
 exec sql begin declare section;
 double dollars;
 exec sql end declare section;
 double range_start = 0;
 double range_end = range_start + RANGE;
 double range_sum = 0.0;
 exec sql declare dollars_cursor cursor for
 select dollars from orders
 order by dollars;
 exec sql whenever sqlerror goto report_error;
 exec sql connect to testdb; /* DB2 UDB connect to database */
 exec sql open dollars_cursor;
 exec sql whenever not found goto fetchdone;
 printf("%-17s %-8s\n", "range", "total orders");
 while (TRUE) { /* loop over orders table */
 exec sql fetch dollars_cursor into :dollars;
 if (dollars <= range_end) {
 range_sum += dollars; /* point in same old range */
 } else { /* point beyond end of this range */
 while(dollars > range_end) { /* -- and possibly several ranges */
 printf("%8.2f-%8.2f %8.2f\n", range_start, range_end, range_sum);
 range_sum = 0;
 range_start = range_end;
 range_end += RANGE;
 }
 }
 }
 report_error:
 exec sql rollback release;
}

```

```

 range_start = range_end; /* step range */
 range_end += RANGE;
 }
 range_sum += dollars; /* found new range that fits point */
}
/* end while loop on aid values */

fetchdone:
if (range_sum)
 printf("%8.2f-%8.2f %8.2f\n", range_start, range_end, range_sum);
exec sql close dollars_cursor;
exec sql commit work; /* release locks */
exec sql disconnect current;
return 0;
report_error:
print_dberror(); /* print out error message */
exec sql rollback work; /* failing, undo work, end locks */
exec sql disconnect current; /* Disconnect from database */
return 1;
}

```

5.12 (a) exec sql begin declare section;  
       char city[21];  
       exec sql end declare sections;

(b) exec sql declare cc cursor for  
       select distinct a.city from agents a, orders x, products p  
           where a.aid = x.aid and x.pid = p.pid and p.price < 1.00;

当然也可以使用子查询。使用条件x.dollars < 1.00列出products是一个常见的错误。使用该条件根本不能显示产品列表。因为它不是问题的解答。

(c) while (1){ /\* or while TRUE, assume TRUE = 1 \*/
       exec sql whenever not found goto fetchdone;
       exec sql fetch cc into :city;
       printf ("%s", city);
}
fetchdone: ...

5.13 (a) if(sqlca.sqlcode == 100) goto error\_handle;

(b) if(sqlca.sqlcode < 0) goto error\_handle;

5.14 (a) select c.city, cid, aid, pid from customers c, agents a,  
       products p  
           where c.city = a.city and a.city = p.city order by c.city;

写成group by c.city是一个常见的错误。因为目标列表中的其他元素对于不同的c.city值并不是单值的这种写法当然是不合法的，根据c.city排序保证了来自同一城市的结果行只出现一次。

(b)  $30 \times 10 \times 20 = 6000$

(c) 基本的思想是避免打印New York的6000行。我们希望打印“New York”，然后是三个city = New York的customers、agents和products的cid、aid、

pid的值的列表（不重复）。我们希望对每一个城市都如此，最后结果一个接一个地输出。我们不要求写严格的C程序（伪代码就可以了），但是必须要能够显示如何使用嵌入式SQL来完成此任务。从声明需要的游标开始；我们在伪代码中省略exec sql。

```

declare cities cursor for select distinct c.city
 from customers c, agents a, products p
 where c.city = a.city and a.city = p.city;
declare cids cursor for select cid from customers
 where city = :city;
declare aids cursor for select aid from agents
 where city = :city;
declare pids cursor for select pid from products
 where city = :city;

open cities;
while (rows remain in cities) { /* loop to fetch citynames */
 fetch cities into :city; /* city value to open other cursors */
 print cityname header;
 open cids;
 loop to print out cid values of cursor cids;
 open aids;
 loop to print out aid values of cursor aids;
 open pids;
 loop to print out pid values of cursor pids;
}
/* loop while cities remain */

```

这是只使用一个游标常见的错误，例如：

```

declare ccs cursor for select c.city, cid, aid, pid
 from customers c, agents a, products p
 where c.city = a.city and p.city = a.city order by c.city;

```

然后，使用如下的程序逻辑：

```

open ccs;
fetch first row of ccs into :city, :cid, :aid, :pid;
remember city name as oldcity;
print city name header;
start printing columns of cid, aid, pid values
while (rows from ccs remain) {
 if latest city does not match oldcity{ /* start new oldcity */
 remember new city name as oldcity;
 print new city name header;
 start printing columns of cid, aid, pid values;
 }
 else { /* same oldcity name */
 print out new values of cid, aid, pid in columns;
 }
}
/* next row of ccs */

```

但是这种方法并不能达到限制信息数量（行数）的目的。因为仍然将有6000行被打印。（问题是New York对应的cid值列表中有很多重复值。）

5.15 (a) exec sql declare topsales cursor for  
           select pid, sum(dollars) totdollars from orders  
           group by pid  
           order by totdollars desc; /\* or "order by 2 desc" in some older systems\*/

```
(b) int count = 0; /* count 10 rows to print out */
while (1){
 whenever not found goto fetchdone;
 fetch topsales into :pid, :doltot: indicator :dolind;
 if (dolind >= 0){ /* not a null value, so count it */
 if (++count >= 10)
 break; /* have already printed 10 rows */
 printf("pid value is %s, total dollar sales is %s",
 pid, dolind);
 } /* end if */
} /* end loop */
fetchdone:
```

5.16 我们不需要在第一次访问之后测试死锁。这是因为不可能发生死锁（虽然如果测试不会有什么害处）。最重要的是在碰到死锁的时候释放所有的锁。一个常见的错误是仅仅循环来试着进行第二次访问而不进行回退。

```
#define DEADABORT ... /* fill in for your system */
exec sql whenever sqlerror continue; /* we're taking over error handling */
int count = 0;
while (1) {
 exec sql update orders
 set dollars = dollars - :delta
 where aid = :agentl and pid = :prod1 and cid = :custl;
 if (sqlca.sqlcode < 0)
 goto handle_err; /* can't be deadlock yet */
 exec sql update orders
 set dollars = dollars + :delta
 where aid = :agentl and pid = :prod1 and cid = :custl;
 if (sqlca.sqlcode == DEADABORT){
 if (count++ < 2) {
 exec sql rollback; /* need to drop locks held */
 /* here, call OS to wait for a second, sleep(1) on UNIX */
 continue; /* try again */
 } else {
 exec sql rollback;
 break; /* give up entirely */
 }
 if (sqlca.sqlcode < 0) goto handle_err; /* other error? */
 exec sql commit work; /* if not, commit */
 } /* end of loop of trials */
}
```

5.17 (a) **ORACLE:** for (i=0;i<sqlda->N;i++)
printf("%\*s ",sqlda->M[i], sqlda->S[i])
printf("\n")

**DB2 UDB:** for (i=0;i<sqlda->sqld;i++)
printf("%\*s ", sqlvar[i].sqlname.length,
 sqlvar[i].sqlname.length);

(b) **ORACLE:** if(\*sqlda->I[1])
handle\_null();

**DB2 UDB:** if (\*sqlda->sqlvar[1].sqlind))
handle\_null();

注意，检测空值的条件是指示器sqlind是否是TRUE，即在C中测试其是否为一个非0值。

## 第6章习题答案

6.1 在四种可能性( $\text{min-card}=0$ 或 $1$ ,  $\text{max-card}=1$ 或 $N$ )中只有两种是真正的约束:  $\text{min-card}=1$  (不允许为0) 和  $\text{max-card}=1$  (不允许大于1)。如果在某种情况下打破了约束, 那么我们就知道该约束不是普遍遵守的了。于是, 在图6-6的(a)中, 因为它属于这种情况下, 我们可以确定  $\text{min-card}(E,R)=0$ , 而且  $\text{min-card}(F,R)=0$ 。但是, 我们不能确定  $\text{max-card}$  的情况。在(b)中, 我们可以确定  $\text{min-card}(E,R)=0$  和  $\text{max-card}(E,R)=N$ 。但是我们不能确定  $\text{card}(F,R)$  的情况, 这是因为没有实际的约束被打破。在(c)中, 我们可以看到  $\text{min-card}(E,R)=0$ ,  $\text{max-card}(E,R)=N$ ,  $\text{min-card}(F,R)=0$ , 以及  $\text{max-card}(F,R)=N$ 。即没有实际的约束成立, 这是因为所有可能的约束都被打破了。

6.2 一个可以接受的E-R图参见图7-2。注意, 属性所标注的基数的  $\text{min-card}$  值基本都是任意的, 除非是主键, 主键的基数必须为(1,1)。我们试着把和F通过动词关系相联系的实体E放到实体F的左侧或者上方。这样, Orders ships Products 和 Orders 在左边, 而 Agents places Orders 和 Agents 在上面。

6.4 (a) 合法的。

(c) 违反了函数依赖(3)。

6.5 我们先考虑只有一个属性在左侧的FD。除了同一性函数依赖以外, 还有如下几点:  
 (a) A列所有的值都相同, 所以对于任何列X, 永远不会发生  $r_1(X)=r_2(X)$ , 而  $r_1(A)\neq r_2(A)$  的情况。于是我们可以看到  $B\rightarrow A$ ,  $C\rightarrow A$  而且  $D\rightarrow A$ 。同时, 不存在其他列X函数依赖于A, 这是因为它们都至少有两个不同的值, 以至于总有两行  $r_1$  和  $r_2$  满足当  $r_1(A)=r_2(A)$  时  $r_1(X)\neq r_2(X)$ 。于是,  $A\nrightarrow B$ ,  $A\nrightarrow C$ , 且  $A\nrightarrow D$ 。  
 (b) 因为所有的C的值都是互不相同的, 所以除了上面知道的  $C\rightarrow A$  以外, 还有  $C\rightarrow B$  且  $C\rightarrow D$ 。同时, C并不函数依赖于其他列, 这是因为所有其他列都至少有两个相同值。所以,  $B\nrightarrow C$  和  $D\nrightarrow C$  是新得到的FD。  
 (c) 我们有  $B\nrightarrow D$  (因为第一、二行) 和  $D\nrightarrow B$  (因为第一、三行)。所以, 我们可以列出所有有一个属性在左侧的FD (前面括号中的字母表示其是从上面第几项得到的)。

- |                       |                       |                      |                       |
|-----------------------|-----------------------|----------------------|-----------------------|
| (a) $A\nrightarrow B$ | (a) $B\rightarrow A$  | (a) $C\rightarrow A$ | (a) $D\rightarrow A$  |
| (a) $A\nrightarrow C$ | (b) $B\nrightarrow C$ | (b) $C\rightarrow B$ | (c) $D\nrightarrow B$ |
| (a) $A\nrightarrow D$ | (c) $B\nrightarrow D$ | (b) $C\rightarrow D$ | (b) $D\nrightarrow C$ |

这样, 我们得到了非平凡的FD: (1)  $C\rightarrow A B D$ , (2)  $B\rightarrow A$  以及 (3)  $D\rightarrow A$ 。现在, 我们考虑在左侧有两列的情况。(d) 因为C确定所有其他的列, 显然包含C的任何列对都能确定所有其他列, 但是这些都是平凡的结果。(e) 列A和任何其他列X在左侧相结合仍然只能函数决定X所能决定的列 (因为加上A以后, 没有其他新的不同行对产生)。现在, 仅有的不包含A或C的列对是BD, 而且因为BD在每一行的值都是不同的 (请再看一下表T), 所以我们知道  $BD\rightarrow$  任意列。我们已经从非平凡FD中知道(2)  $B\rightarrow A$ , 而且  $BD\rightarrow BD$  是平凡的, 所以能够从中得到的仅有的新的FD是(4)  $BD\rightarrow C$ 。如果我们现在考虑三列的情况, 显然, 任何不包括C的三列 (所以函数决定所有其他列) 必定包括BD (所以函数决定所有列)。所有的非平凡FD为:

(1)  $C\rightarrow A B D$ , (2)  $B\rightarrow A$ , (3)  $D\rightarrow A$ , (4)  $BD\rightarrow C$

6.7 (a) 不是阿姆斯特朗表, 最后一行在讨论中没有出现, 可以被删掉了。

6.9 合并规则: 如果  $X\rightarrow Y$  且  $X\rightarrow Z$ , 那么  $X\rightarrow YZ$ 。通过  $X\rightarrow Y$  和增广律, 可以得到  $XY\rightarrow$

$YX$ 。但是， $XX=X$ ，所以 $X \rightarrow XY$ 。类似地，在 $X \rightarrow Z$ 中加 $X$ 可以得到 $X \rightarrow XZ$ 。

接着，通过加 $Y$ ，获得 $XY \rightarrow XYZ$ 。接着 $X \rightarrow XY$ 和 $XY \rightarrow XYZ$ 意味着 $X \rightarrow XYZ$ 。但是 $YZ$ 是 $XYZ$ 的子集，所以通过包含律， $XY \rightarrow YZ$ ，而根据传递性， $X \rightarrow YZ$ 。

伪传递性：如果 $X \rightarrow Y$ 而且 $WY \rightarrow Z$ ，那么， $XW \rightarrow Z$ 。根据增广律 $X \rightarrow Y$ 可以得到 $XW \rightarrow YW$ ，而且根据传递性以及 $WY \rightarrow Z$ ，可以得到 $XW \rightarrow Z$ 。

6.11 (a) 假设 $X^+$ 不等于 $X$ ，那么在 $X^+$ 中有一个属性 $A$ 不在 $X$ 中，使得 $X \rightarrow A$ （否则我们使用FD就得不到和 $X$ 不同的 $X^+$ ）。此FD被F所覆盖，所以 $A$ 在F中某个元素的右侧，而左侧为 $X$ 。因为 $X$ 并不包含任何左侧的内容，所以，必定有某个非平凡的 $X$ 中的属性集B不在 $X$ 中。于是， $X=BX$ ， $BX \rightarrow A$ 且 $X \rightarrow A$ 。但是，这意味着 $X$ 并不是最小的。特别地，在最小覆盖算法中的第三步，B可以被删掉。得出矛盾。

6.12 下面是证明所期望的结果所要用到的阿姆斯特朗公理和定理6.6.8的结果。设 $W$ 、 $X$ 、 $Y$ 和 $Z$ 是任意列的集合。

- 1) 包含律：如果 $Y$ 是 $X$ 的子集，那么 $X \rightarrow Y$ 。
- 2) 增广律：如果 $X \rightarrow Y$ ，那么 $WX \rightarrow WY$ 。
- 3) 传递律：如果 $X \rightarrow Y$ 而且 $Y \rightarrow Z$ ，那么 $X \rightarrow Z$ 。
- 4) 合并规则：如果 $X \rightarrow Y$ 而且 $X \rightarrow Z$ ，那么 $X \rightarrow YZ$ 。
- 5) 分解律：如果 $X \rightarrow YZ$ ，那么 $X \rightarrow Y$ 而且 $X \rightarrow Z$ 。
- 6) 伪传递律：如果 $X \rightarrow Y$ 而且 $WY \rightarrow Z$ ，那么 $XW \rightarrow Z$ 。

#### (a) $D \rightarrow ABCD$

步骤如下：(1) 根据自反律，因为 $D$ 是 $D$ 的子集（都是单元素集）， $D \rightarrow D$ 。(2) 因为 $D \rightarrow ABC$ （由FD(3)给定）和 $D \rightarrow D$ ，通过合并规则，得 $D \rightarrow ABCD$ 。

6.14 (a) 步骤1： $H=\{A \rightarrow B, C \rightarrow B, D \rightarrow A, D \rightarrow B, D \rightarrow C, AC \rightarrow D\}$ 。

步骤2：

- 1)  $A \rightarrow B$ 。 $J=H - \{A \rightarrow B\}$ ，在 $J$ 下 $X^+$ ： $X[0]=A$ ,  $X[1]=A$ ,  $X^+=A$ ，不包含 $B$ ，所以保留 $A \rightarrow B$ ，即 $H$ 不变。
- 2)  $C \rightarrow B$ 。 $J=H - \{C \rightarrow B\}$ ， $X[0]=C$ ,  $X[1]=C$ ，所以 $X^+=C$ ，不包含 $B$ ，所以 $H$ 不变。
- 3)  $D \rightarrow A$ 。 $J=H - \{D \rightarrow A\}$ ， $X[0]=D$ ,  $X[1]=DBX$ ,  $X[2]=DBX$ ，不包含 $A$ ，所以 $H$ 不变。
- 4)  $D \rightarrow B$ 。 $J=H - \{D \rightarrow B\}$ ,  $X[0]=D, \dots, X^+=DABC$ ，包含 $B$ ，所以删除此FD。

新的 $H=\{A \rightarrow B, C \rightarrow B, D \rightarrow A, D \rightarrow C, AC \rightarrow D\}$ 。

- 5)  $D \rightarrow C$ 。 $J=H - \{D \rightarrow C\}$ ,  $X^+=DAB$ ，不包含 $D$ ，所以 $H$ 不变。

结果 $H=\{A \rightarrow B, C \rightarrow B, D \rightarrow A, D \rightarrow C, AC \rightarrow D\}$ 。

步骤3：只有 $AC \rightarrow D$ 可能在左侧可以化简。在其左侧 $B$ 上循环：

- 1)  $B=A$ ,  $Y=C$ ,  $J=\{A \rightarrow B, C \rightarrow B, D \rightarrow A, D \rightarrow C, C \rightarrow D\}$ 。 $J$ 下 $Y^+=CBDA$ ,  $H$ 下 $Y^+=CB$ ，不同，所以保持 $H$ 不变。
- 2)  $B=C$ ,  $Y=A$ ,  $J=\{A \rightarrow B, C \rightarrow B, D \rightarrow A, D \rightarrow C, A \rightarrow D\}$ 。 $J$ 下 $Y^+=ABDC$ ,  $H$ 下 $Y^+=AB$ ，不同，所以保持 $H$ 不变。

注意，被提议（但是被拒绝）的修改使得太多的属性被包含在闭包中了。

步骤4： $M=\{A \rightarrow B, C \rightarrow B, D \rightarrow AC, AC \rightarrow D\}$ 。

6.15 令 $Y^+_H$ 代表 $H$ 决定下的 $Y^+$ ，那么我们需要说明 $Y^+_H=Y^+$ 暗示了 $H^+=J^+$ 。在这里， $J=(H - \{x \rightarrow A\}) \text{ UNION } \{Y \rightarrow A\}$ ，而且 $Y=X - \{B\}$ 。除了一个在左侧少一个属性的FD以外， $J$ 和 $H$ 的FD是相同的，这使它蕴含了多于它替换的那个，所以显然 $H^+$ 是 $J^+$ 的子集。我们声明如果它们的确是不同的，那么显然在 $J^+$ 中的 $Y \rightarrow A$ 不在 $H^+$ 中，因为如果 $Y \rightarrow A$ 在 $H^+$ 中，那么根据定义，所有的 $J$ 中的FD都在 $H^+$ 中，以使得 $H^+$ 覆盖 $J$ ，进而覆盖 $J^+$ ，所以 $J^+$ 是 $H^+$ 的子集。但是我们已经知道 $H^+$ 是 $J^+$ 的一个子集，所以 $H^+=J^+$ 。这与假设矛盾。

到目前为止，我们说明了如果 $H^+$ 和 $J^+$ 是不同的，那么 $Y \rightarrow A$ 不在 $H^+$ 中。但是如果 $Y \rightarrow A$ 不在 $H^+$ 中，那么 $A$ 不在 $Y^+_H$ 中，而且我们知道 $Y \rightarrow A$ 在 $J^+$ 中，即 $A$ 在 $Y^+_J$ 中，所以集合 $Y^+_H$ 和 $Y^+_J$ 是不同的。它的逆否命题即是所要的结果。

6.19 (b) 由于对于T没有函数依赖， $B \rightarrow A$ 且 $B \rightarrow C$ ，所以 $\text{Head}(T1) \cap \text{Head}(T2)$ 并不函数决定 $\text{Head}(T1)$ 或者 $\text{Head}(T2)$ 。在这种情况下，定理的“仅当”部分可以确定分解不是无损的，即存在某些内容，有 $T < T1 \bowtie T2$ 。我们已经说明过这些内容。

6.20 (b) 在需要用到(a)部分的结论以前，我们首先独立于(a)部分进行讨论。从一开始就直接使用(a)的结论可能可以使整个过程更简洁。从表 $T=(A, B, C, D, E, F, G)$ 开始，该表的主键是什么？ $B$ 和 $D$ 不在任何函数依赖的右部，所以必定被包含在任意键中。这是由于一个键函数决定所有其他列。有了 $BC$ ，我们得到了 $A$ （通过(1)和(5)）； $E$ （通过(2)）；又由于有了 $A$ ，通过传递律，得到了 $F$ （通过(3)）。于是(4)提供了 $G$ ，而(5)提供了 $D$ ，所以我们得到了（通过从左侧积累） $BC, A, E, F, G$ 和 $D$ ，即所有的列。这样， $BC$ 就是 $T$ 的一个超键，又因为 $B$ 和 $C$ 必须被包含在任意键中，所以 $BC$ 是键。所有给定的FD在同一张表中。

现在， $D$ 并不函数依赖于 $BC$ ，而仅依赖于 $C$ ，则 $C \rightarrow D$ 是一个固有的键真子集FD。于是即使仅仅为了2NF，我们也需要分解出一张表，于是我们有了两张表 $T_1=(B, C, A, E, F, G)$ 和 $T_2=(C, D)$ ，相应的键为 $BC$ 和 $C$ 。 $T_1$ 包含了除(5)和(1)外所有给定的FD，而(5)在 $T_2$ 中，(1)跨两张表。但是，如果我们采用在(a)部分计算的FD的最小覆盖，我们看到只有一个完全在 $T_1$ 中，一个完全在 $T_2$ 中。函数依赖 $A \rightarrow F$ 和 $F \rightarrow G$ 是传递依赖。因此， $T_1$ 和 $T_2$ 构成了2NF分解。

(d) 对 $F=\{BC \rightarrow AE, A \rightarrow F, F \rightarrow G, C \rightarrow D\}$ 使用算法6.8.8。

$S=$ 空集。对 $F$ 中的所有FD进行循环：

$BC \rightarrow AE$ 。 $S$ 中什么也没有，所以没有函数依赖可以包含 $BCAE$ ，所以把它加入到 $S$ 中去得 $S=\{\{BCAE\}\}$ 。

$A \rightarrow F$ 。 $AF$ 不包含在 $BCAE$ 中，所以把它加入到 $S$ 中去得 $S=\{\{BCAE\}, \{AF\}\}$ 。

$F \rightarrow G$ 。 $FG$ 目前不包含在任何集合中，所以把它加入 $S$ 得 $S=\{\{BCAE\}, \{AF\}, \{FG\}\}$ 。

$C \rightarrow D$ 。 $CD$ 不包含在目前的任何集合中，所以把它加入： $S=\{\{BCAE\}, \{AF\}, \{FG\}, \{CD\}\}$ 。

这里，唯一的候选键是 $BC$ ，它被包含在 $BCAE$ 中。所以第二次循环并没有向 $S$ 中加入任何元素。得到的表的设计结果和以前得到的结果是一样的。

6.22 银行的FD问题：每个账户有唯一的类型、余额和支行，所以我们有 $\text{acct\_id} \rightarrow \text{acct\_type}$   $\text{acct\_bal}$   $\text{bno}$ 。每一个支行有一个城市，每一个客户有一个名字，所以集合为：

```

acctid -> acct_type acct_bal bno
bno -> bcity
ssn -> cname fname cmidinit

```

$T=(\text{acctid}, \text{acct\_type}, \text{acct\_bal}, \text{bno}, \text{bcity}, \text{ssn}, \text{cname}, \text{fname}, \text{cmidinit})$ 。由于 $\text{ssn}$ 和 $\text{acctid}$ 仅在FD的左侧出现，所以它们必定被包含在所有键中，而且事实上所有其他属性都被包含在 $\{\text{ssn}, \text{acctid}\}$ 的闭包中，所以它其实也就是 $T$ 的键。现在，第一和第三个FD是键子集依赖，而第二个FD是传递依赖，但是由于分解要求的范式是3NF，所以我们并不关心它们的区别，而仅仅进行三次分解，例如使用如下方法：

1.  $T_1 = (\text{acctid}, \text{acct\_type}, \text{acct\_bal}, \text{bno}, \text{bcity}), T_2 = (\text{acctid}, \text{ssn}, \text{cname}, \text{fname}, \text{cmidinit})$ .
2.  $T_1 = (\text{acctid}, \text{acct\_type}, \text{acct\_bal}, \text{bno}), T_2 = (\text{bno}, \text{bcity}), T_3 = (\text{acctid}, \text{ssn}, \text{cname}, \text{fname}, \text{cmidinit})$ .
3.  $T_1 = (\text{acctid}, \text{acct\_type}, \text{acct\_bal}, \text{bno}), T_2 = (\text{bno}, \text{bcity}), T_3 = (\text{ssn}, \text{cname}, \text{fname}, \text{cmidinit}), T_4 = (\text{acctid}, \text{ssn})$  Final result, in 3NF form.

注意，在过程中我们已经从 $T$ 抽取了所有的非键属性，而只留下 $T_4=(\text{acctid}, \text{ssn})$ 作为初始的通用表的框架。但是事实上它代表了该设计的一个重要事实：在accounts和customers之间存在着一个二元联系。该设计和习题6.3的第二个设计是一样的。习题6.3的第一个设计没有完全分解。请看结果：

```

accounts = (acctid, acct_type, acct_bal)
branches = (bno, bcity)
customers = (ssn, cname, fname, cmidinit)
has_account_at = (ssn, acctid, bno)

```

我们已经声明了 $\text{acctid} \rightarrow \text{bno}$ ，而这意味着最后一个表有一个键子集依赖，所以它甚至连2NF也不满足。

## 第7章习题解答

```

7.1 (a) create table agents (aid char(3) not null,
 aname varchar(13), city varchar(20),
 percent integer check(percent >= 0 and percent <= 10),
 primary key (aid));

 create table products (pid char(3) not null,
 pname varchar(13), city varchar(20),
 quantity integer check(quantity > 0),
 price real check(price > 0.0)), primary key (pid);

```

7.3 大多数内容参见图7-2中。对于每行，应该正好有一个产品，但是对于一个产品可能有多行，或者没有行相对应。所以for\_prod和Products之间的连接应被标上(0, N)，而for\_prod和Line\_items之间的连接应该被标上(1, 1)。

像在6.4节中那样，我们首先从作为实体的表开始，然后再加上外键和联系表来完成相关的设计。

为了简便，开始的实体表和实体ID就使用在图6-11中所见的内容：

```

customers = (cid)
orders = (ordno)

```

```

agents = (aid)
line_items = (ordno, lineno)
products = (pid)

```

联系requests是N-1的，因此它通过外键实现：

```
orders = (ordno, cid)
```

类似地，places和has\_item也是N-1的，所以orders为：

```
orders = (ordno, cid, aid, lineno)
```

最后，for\_prod也是N-1的，所以line\_items成为：

```

line_items = (ordno, lineno, pid);
create table customers (cid char(4) not null, primary key (cid));
create table orders (ordno integer not null, cid char(3) not null references
customers, aid char(3) not null references agents, lineno integer not null
references line_items, primary key (ordno));

create table agents (aid char(3) not null, primary key (aid));
create table products (pid char(3) not null, primary key (pid));
create table line_items (ordno integer references orders, lineno integer not null,
pid char(3) not null references products, primary key (ordno, lineno));

```

7.4 (a) 不可能。

(c) 不可能。

7.5 (b) 

```
create view agentview (aid, fname, city, percent)
as select aid, fname, city, percent from agents
where percent >= 0 and percent <= 10 with check option;
```

(d) 

```
create view vproducts as
select pid, pname, city, quantity from products;
grant select, update (city, quantity) on vproducts to beowulf;
```

7.6 (a) 错。根据定义7.1.3的主键子句定义。

(c) 错。在X/Open中，CHECK子句不允许包含子查询。

7.7 (a) 对于所有三种产品的CASCADE或缺省的NO ACTION都适用，对于DB2 UDB，  
RESTRICT、CASCADE、SET NULL和作为缺省的NO ACTION都适用。

(c) Create Table语句会返回一个错误，这是因为违反了主键约束（对于主键cid）。  
解决办法是使用Select UNIQUE子查询。

7.8 (d) 

```
create table customers1 (cid not null, cname, city,
discnt constraint discnt_max check(discnt <= 15.0)),
primary key (cid) as select * from customers;
```

7.10 (a) 引用agentords的视图仅仅在名称上不同于例7.2.1中的视图：

```

create view agentords (ordno, month, cid, aid, pid, qty,
charge, fname, city, percent)
as select ordno, month, cid, a.aid, pid, qty, dollars,
fname, city, percent
from orders o, agents a where o.aid = a.aid;

```

根据图7-15中的规则，视图agentords在X/Open中是不能更新的（类似地，在SQL-92中也不能更新，但是ORACLE允许在更多的情况下更新视图）。一般情况下，对于视图

agentords的行的插入、更新、删除操作可以被认为是对表orders的行的相同的操作。这是由于agentords中的行和orders中的行之间有一对一的对应关系。agents中的行一般对应agentords中的多行，但是对于agents表的更新如果是有意义的，那么允许进行该操作从道理上讲也是说得通的。

以一个新的aid插入。这看上去像是同时插入了一个新的agents行和一个新的orders行：一个新的订单引用了一个新的代理。我们允许这样操作。当然，仍必须遵守所有的已存在的完整性约束。例如，订单必须引用一个已经存在的产品pid，ordno不能和已经存在的ordno重复等。

以一个已经存在的aid插入。这就是插入一个新的引用了已经存在的aid的订单。可能出错的情况是：插入操作指定了一个已有的aid，但是给了一个错误的agents其他列的值，例如aname。由于aid被认为是代理的主键，aid函数决定所有其他列，而给定一个其他的aname值会破坏完整性约束，于是插入操作会失败。在插入时允许不指定agents的依赖列而使用函数依赖的值作为缺省值是比较合理的。

### 上机作业

#### 7.14 ORACLE的情况：

```
SQL> select * from user_constraints;
... many lines of output
SQL> alter table customers add constraint discntmax check (discnt <= 12.0);
ERROR at line 1:
ORA-02293: cannot enable ... check constraint violated
SQL> update customers set discnt = 10.0 where cid = 'c001';
1 row updated.
SQL> alter table customers add constraint discntmax check (discnt <= 12.0);
Table altered.
SQL> update customers set discnt = 16.0 where cid = 'c001';
ORA-02290: check constraint ... violated
SQL> select * from customers;
...see discnt = 10.0 for c001.
SQL> alter table drop constraint discntmax;
Table altered.
SQL> update customers set discnt = 16.0 where cid = 'c001';
1 row updated.
SQL> select * from customers;
...see discnt = 16.0 for c001.
SQL> exit
Now run loadcap as explained in Appendix A.
```

#### 7.15 (a) ORACLE的情况

```
SQL> insert into orders values (1031, 'jul', 'c001', 'a01', 'p01', 1000, 450.0);
1 row inserted.

SQL> create view returns (ordno, month, cid, aid, pid, qty, dollars,
2 discnt, percent, price)
3 as select ordno, month, o.cid, o.aid, o.pid, qty, dollars,
4 discnt, percent, price
5 from orders o, customers c, agents a, products p
6 where c.cid = o.cid and a.aid = o.aid and p.pid = o.pid;

SQL> select view_name from user_views;
view_name

```

```

returns
SQL> select column_name from user_tab_columns where table_name = 'RETURNS';
column_name

ORDNO
MONTH
CID
PIO
QTY
DOLLARS
DISCNT
PERCENT
PRICE

(c) select ordno, qty, dollars, discnt, percent, price,
 qty * price - (discnt / 100) * price * qty from returns;

```

7.20 为了查找表people所使用的行类型（作为列的数据类型），需要进行如下的三元连接：

```

select typ.name from syscolumns c, systables t, sysxtypes typ
 where t.tabid = c.tabid and t.tabname = 'people'
 and c.extended_id = typ.extended_id;
select typ.name from syscolumns c, systables t, sysattrtyps a, sysxtypes typ
 where t.tabid = c.tabid and t.tabname = 'people'
 and c.extended_id = a.extended_id
 and a.xtd_type_id = typ.extended_id;

```

## 第8章习题解答

- 8.1 (a) 由于最小区域(minextents)是3，所以 $3 \times 20\ 480 = 61\ 440$ 字节。  
      (b) 由于最大区域(maxextents)是8，所以 $8 \times 20\ 480 = 163\ 840$ 字节。
- 8.2 (a) 在一个页面中有2048个字节，对于512个槽行目录需要 $512 \times 2 = 1024$ 个字节。于是剩下 $2048 - 1024 = 1024$ 个字节用于存放行，于是对于每行有 $1024/512 = 2$ 个字节。  
      (c) 我们允许： $2^{23} = 8\ 388\ 608$ 页面/表。
- 8.4 (a) 每行需要大约100字节（包括两个字节的目录偏移），且我们每页有大约 $0.75 \times 2000 = 1500$ 个可用字节。那么 $1500 \text{字节/页} / 100 \text{字节/行} = 15 \text{行/页}$ 。 $200\ 000 \text{行} / 15 \text{行/页} = 13\ 334 \text{页}$ 。  
      (d) 仍然需要对索引的根进行一次磁盘访问，并对下一级目录进行一次磁盘访问。我们要在10 000个叶子层项中进行搜索，在148项/页的情况下，这意味着对叶节点页面有大约 $\text{CELL}(10\ 000/148) = 68$ 次访问。由于行并没有被聚簇，而且假设缓冲机制被忽略，获取10 000行需要10 000次磁盘I/O。所以磁盘I/O次数为 $(1+1+68+10\ 000) = 10\ 070$ 次磁盘访问。在每秒40次磁盘访问的情况下，我们需要大约252秒。在每秒80次I/O的情况下，我们仅需要126秒。  
      (f) 为了计算行的数目，查询优化器仅需要计算入口的数目，而不需要更深入地去取每一行。于是，磁盘访问次数为 $(1+1+68) = 70$ 次。在每秒40次I/O的情况下，这需要1.75秒。

- 8.6 (a) (i) 对DB2 UDB是错的（但是DB2 UDB v6允许32K字节的页面）。
- (iii) 对。参见例8.3.2。
- (v) 对。在1 000 000个页面中有1000个页面在缓冲区中，所以1000个页面中有1个页面存在于缓冲区中。
- 8.7 (b) 2000的75%是1500，所以页面中有1500字节可用。一项包含：rowid 6字节；keyval 在此为7字节，另有1字节的额外开销，总共14个字节。于是 $1500/14=107$ 项/页， $400\ 000 \text{项}/(107 \text{项/页})=3739$ 个叶子层页面。
- (e) 一个聚簇索引使用同样多的索引页面，但是20 000个数据行在磁盘上是邻近的，它们存在于 $20\ 000/8=2500$ 个页面中。于是共要2689次磁盘页I/O。
- 8.9 (a) 在此 $N=128$ ,  $M=10\ 000$ 。 $E(S)=10\,000*(1 - 0.9999^{128})$ ，其中 $0.9999^{128}=0.9999^{2\times 2\times 2\times 2\times 2} = (((((0.9999^2)^2)^2)^2)^2)=0.9873$ ，则 $E(S)=10\,000 \times 0.0127=127$ 。使用[8.6.4]， $10\,000 \times (1 - \exp(-128/10\,000))=10\,000(1 - 0.9873)=127$ 。是的，两种方法都表明期望的冲突次数为一次。
- 8.10 (a) 所有飞镖在不同的槽中的概率对于2支飞镖为 $364/365$ ，这是因为第一支飞镖可以到任何槽中，而第二支飞镖必须在365个槽中剩下的364槽中；而对于3支飞镖，概率为 $(364/365) \times (363/365)$ ；对于所期望的N支飞镖，概率为 $(364 \times 363 \times \dots \times 365 - N+1)/365^{N-1}=0.5$ 。 $(1 - 1/365)(1 - 2/365)\dots(1 - (N-1)/365)=0.5$ 。展开乘法，并忽略在分母中重复出现的因子365，约为 $1 - (1+2+3+\dots+(N-1))/365=0.5$ 或 $1+2+\dots+(N-1)=365/2$ 。其中的和为 $(N-1)^2/2$ ，所以 $(N-1)^2=365$ ，即近似得 $N=20$ 。仔细地计算得 $N=23$ 时得到的结果0.493最接近于0.5，所以忽略的项对结果仍有一些影响。

## 第9章习题解答

- 9.1 (a) 寻道时间0.008秒+旋转延时0.004秒+传输时间0.004秒（8乘以0.0005）=0.016秒。
- (c) (i) 在这种情况下，每次顺序预取8页是有帮助的。为了将5000个索引页面读取到P8块中，计算得 $5000/8=625$ 块，所以大约需要 $625 \times 0.016$ 秒（即大约10秒）。（在此没有经验规则，就和我们所说的以每秒800次 I/O 的速率处理顺序预取的情况一样，计算时需要格外小心。）我们还有50 000个聚簇数据页面，且 $50\ 000/8=6250$ ， $6250 \times 0.016=100$ 秒。所以总共执行时间为 $10+100=110$ 秒。
- (ii) 在这种情况下，使用了随机I/O， $50\ 000R+5000R$ 需要 $55\ 000/80$ 秒，即687.5秒。
- 9.2 (a) PCTFREE=0时，我们可以把每10个长为400字节的行放入4KB的磁盘页面中去。于是， $100\ 000$ 行就需要 $100\ 000/10=10\,000$ 个磁盘页面。eidx索引有10字节的项，在一个叶子页面可以放400个项。于是， $100\ 000$ 个叶子层项需要 $100\ 000/400=250$ 个叶子页面。在上一层，项的数目是相同的，每页400项，显然250项可以放入一个页面中，即根。
- (b) (i) $P(\text{leaf})=1/250$ 。
- (ii)  $(1 - P(\text{leaf}))=249/250$ 。
- (iii) 调用事件NOT-leaf-N。我们需要N个独立事件的连接，其中每个事件的概率为 $249/250$ 。 $P(\text{NOT-leaf-N})=(249/250)^N$ (N次幂)。

- (iv)  $(240/250)^{250} = 0.367$ 。（有趣的是，这是 $\exp(-1)$ 即 0.368的一个近似值，这是由于计算所用的公式为 $(1 - 1/x)^x$ ，对于一个很大的x，当x趋于无穷大时，该式子趋于 $\exp(-1)$ 。）
- (v) 有0.367部分会在缓冲区外，这是因为它们在最后125秒中没有被引用，所以保留在缓冲区中的数目为 $(1 - 0.367) \times 250 = 158.25$ 。（你可以根据“期望”的定义重新计算。对于某一个特定的叶子， $P(\text{叶子在缓冲区中}) = 1 - 0.367 = 0.633$ 。在缓冲区中的叶的期望的值=叶数目  $\times 0.633 = 158.25$ 。）
- (vi) 在此我们有 $N=250$ 个飞镖和 $M=250$ 个空位，所以空位命中率的期望约为 $E(S)=M(1 - e^{-NM})=250(1 - e^{-1})=158$ 。
- (c) (iv) 从(b)部分中我们看到在长为125秒的时间段内引用的250个数据页面中有超过247页最终在缓冲区中。于是，忽略两次重复引用，仅有三次引用命中已经引用过的页面，所以仅有三页被重复引用。另一方面，在这段时间内引用的250个索引页面中仅有158页最终在缓冲区中，所以 $250 - 158 = 92$ 个索引叶子页面被重复引用。由于叶子页面比数据页面少，所以它们被重新引用得更频繁。于是在这段时间内更少的不同的索引叶子页面被引用，且它们都在缓冲区中。
- (f) 在此我们每秒有两个查询。不在缓冲区中的数据页面的比例为 $(10\ 000 - 247)/10\ 000 = 0.975$ 。所以I/O代价为 $2 \times 0.975$ ，即为1.950。根页面从不会在缓冲区以外。一个叶子页面有0.367的概率不在缓冲区中（根据(b)），所以对于两个查询都增加了I/O开销0.734。于是，总的I/O开销为每秒 $1.950 + 0.734 = 2.684$ 。
- (h) 对于数据页面，I/O代价将为2，显然这是一个比LRU更高级的策略。这是因为我们使用了更小的内存和更低的I/O代价。
- 9.4 (b) (i) ACCESSTYPE = I, ACCESSNAME = C1234X, MATCHCOLS = 1, (INDEXONLY = N)。
- (ii) 必须检索所有C1值从1到10的索引项。这是C1234X整个范围的1/10，C1234X的NLEAF=150 151，所以需要获取的叶子页面数目为15 015。使用顺序预取，所以需要时间为 $15\ 015/800 = 18.8$ 秒。
- (iii) 该复合谓词的过滤因子为 $(1/10)(1/20)(1/50) = 1/10\ 000$ 。
- (iv) 检索的行的数目为 $(1/10\ 000)(100\ 000\ 000) = 10\ 000$ 。T中的行在索引C1234X中都是连续的。在此选择了C1的10个值，每一个又被分为C2值的200个范围。从中我们选取10个邻接的区域。但是这10个的每一个又被进一步分为不同C3值的50个范围，从中我们选取一个。这样，就使用了索引中分离的100个区域，而将它们分离开的区域并没有被使用。10 000行分布在100个区域中，所以在一个区域中大约有100个行是连续的，但是整个集合中的行并不是相互连续的。而DB2并不能利用这些小的连续的区域，因为它将在处理筛选谓词时处理它们。（列表预取也不会被使用，因为没有过滤可以对RID列表进行操作。）所以执行时间为 $10\ 000/80 = 125$ 秒。
- (v)  $15\ 015S + 10\ 000R$ ，执行时间为 $18.8 + 125 = 143.8$ 秒。
- 9.5 (a) (i) C1和C2是匹配列。匹配在C2停止，这是根据定义9.5.4的第3部分得出的结论。

- (iii) 没有匹配的列，这是因为 $C1 < 6$ 不是一个可索引的谓词。
- (v)  $C1$ 和 $C2$ 是匹配列。在 $C3$ 前匹配停止，因为 $C3 \text{ like } \%abc\%$ 不是可索引的。
- (vii) 没有匹配的列，这是因为 $C1 = \text{Expression}$ 不是可索引的。
- (b) (i)  $\text{FF}(C1=7)=1/100=0.01$ ,  $\text{FF}(C2>=101)=100/200=0.5$ , 所以合成的 $\text{FF}=0.01 \times 0.5=0.005$ 。
- (iii)  $0.005 \times 100\ 000\ 000=500\ 000$ 个索引项。请注意所有的列长度都为4个字节。  
 对于 $C1234X$ , 键值长度为16字节。在叶子层有大量的压缩，每个重复块有1个键值（16字节）、4个字节的块偏移以及10个长度为4个字节的RID。这意味着10个项共60字节，或者说每项6字节。我们可以在每个页面中存放 $\text{FLOOR}(4000/6)=666$ 项。这样就有 $\text{CEIL}(500\ 000/666)=751$ 个页面存在于叶子层。它们可以被顺序预取(S)。在叶子层以上，索引还有一层，但是其中只有一页在第一次访问的查找中被访问。该页面要求随机访问(R)。
- (v)  $751S+6000R=751/800+6000/80=75.95$ 秒。
- 9.6 (a) 在此，我们有五千万行且 $\text{FF}(\text{incomeclass}=10)=1/10$ , 所以从 $\text{incomex}$ 取得的RID列表将包含五百万个RID。这低于绝对最大值一千六百万。五百万个RID需要两千万个字节，这一定小于RID池的50%，所以RID池至少有四千万个字节，由于RID池是缓冲池的一半，这意味着缓冲池至少需要八千万字节。这样，系统必须有足够的内存来容纳一亿二千万字节的缓冲池和RID池。
- (c)  $\text{FF}(\text{AGE between } 20 \text{ and } 39)=2/5$ : 将有 $0.4 \times 5\text{千万}=2\text{千万agex索引项}$ ，超出了RID列表的绝对最大值，所以不可能。
- (e) 根据定义9.6.1的规则3，这不可能，因为排除了IN\_LIST谓词。
- 9.7 (a) ACCESSTYPE=I, ACCESSNAME=mailx, MATCHCOLS=1。我们停止从左到右在zipcode部分匹配的原因有两个：(1)mailx中下一列为hobby，它并不匹配；(2)当我们遇到BETWEEN谓词时总是停止匹配。
- (c) (i)是的，在习题9.6的(a)所要求的内存条件下，我们可以在 $\text{incomeclass}=10$ 中利用 $\text{incomex}$ 索引使用步骤MX。
- (ii)

| ACCESS TYPE | MATCH COLS | ACCESS NAME | PRE FETCH | MIXOP SEQ |
|-------------|------------|-------------|-----------|-----------|
| M           | 0          |             | L         | 1         |
| MX          | 1          | mailx       | S         | 2         |
| MX          | 1          | agex        | S         | 3         |
| MI          | 0          |             |           | 4         |
| MX          | 1          | incomex     | S         | 5         |
| MI          | 0          |             |           | 6         |

注意，谓词按照过滤因子从小到大的顺序排列，一旦可能就进行相交操作，以避免在假想的栈中存在超过一个的RID列表。其他顺序也是可用的。

- (iii) 1/50 000。
- (e) 像在(d)中那样，从扫描mailx索引开始为zipcode谓词抽取RID列表需要3.12秒。然后从NLEAF=50 000的agex为age谓词抽取RID列表，于是有(1/50)(50 000)，I/O代价为1000S，需要时间为 $1000/800=1.25$ 秒。最后从NLEAF=50 000的incomex中为income谓词抽取RID列表，于是有(1/10)(50 000)，I/O代价为5000S，需要时间为6.25秒。最后在1000各数据页面上有1000个数据行，但是此时我们可以使用列表预取1000L，需要时间为 $1000/200=5$ 秒。总共需要时间为 $3.12+1.25+6.25+5=15.62$ 秒。只有在偶然情况下它们才会同时发生。
- 9.8 (a) 不可能。它将有25 000 000个RID，于是超过了绝对最大值。
- (b) (i) zipcode between 02139 and 07138是匹配谓词，其余部分是筛选谓词。
- (ii) 对于任何一种方法，我们希望 $FF \times 50 000 000 = 0.01 \times 0.02 \times 0.05 \times 50 000 000 = 500$ 行，且并不聚簇。对于方法(i)，RID在筛选时被找到，于是必须使用随机I/O，需要 $500R = 500/80 = 6.25$ 秒。对于方法(ii)，我们获得了RID列表，于是可以使用列表预取，需要 $500L = 500/200 = 2.5$ 秒。
- 9.9 (a) 仅有的匹配列为zipcode，这是因为根据定义9.5.4的第3部分，搜索匹配谓词的操作在碰到BETWEEN谓词后停止。为zipcode的 BETWEEN谓词所进行的索引I/O将扫描 $(2000/100 000)(250 000\text{叶子页面}) = 5000\text{叶子页面}$ ，使用顺序预取5000S需要执行时间为 $5000/800 = 6.25$ 秒。和所有谓词相结合的过滤因子(包括筛选谓词)按照谓词子句的出现顺序得： $(2000/100 000)(1/50 + 1/50 - (1/50)(1/50))(1/100)(1/10) = (1/50)(1/25)(1/100)(1/10) = 1/1 250 000$ 。在prospects中有50 000 000行，这意味着我们期望抽取 $(1/1 250 000)(50 000 000) = 40$ 行。使用随机I/O(我们不可以使用列表预取，这是因为由于我们使用了筛选谓词，我们并没有抽取RID列表)，40R的执行时间为 $40/80 = 0.5$ 秒。总的执行时间为6.75秒。
- (c) (a)花费6.75秒，优于(b)的11.37秒。这条普遍的规律之所以成立是因为在使用了筛选谓词的条件下，对于第一列的匹配扫描和在MX抽取时和其他索引扫描一起进行的扫描是重复的，所以MX抽取时花费了更多的索引I/O。MX的该项亏损在最后可能被弥补，因为MX方法在抽取数据页面时允许列表预取。
- 9.10 (a) 考虑T1.C6=5，我们限制T1的1 000 000行到1/20，即50 000行。对于T1.C7中的每个值，我们希望匹配T2.C8的值，在T1中选取的50 000行中，T1.C7有一个常数值K。我们问在T2中有多少行与之匹配，即T2.C8=K的选择性为多少？显然，答案为 $1/200 000$ ，有5行(平均)匹配。所以，T1的50 000行中的每一行匹配T2中的5行，而且迄今为止在连接中有250 000行。现在，我们必须限制T2.C9=6，其筛选因子为 $1/400$ ，连接的结果中行数目为 $(1/400)(250 000) = 625$ 。
- 9.11 (a) 第一遍以后：
- ```
12 45 67 84 | 7 29 58 76 | 22 39 81 91 | 28 33 65 96 | 4 13 54 77 | 1 32 41 59
```
- 第二遍以后：
- ```
7 12 29 45 58 67 76 84 | 22 28 33 39 65 81 91 96 | 1 4 13 32 41 54 59 77
```
- 第三遍以后：
- ```
7 12 22 28 29 33 39 45 58 65 67 76 81 84 91 96 | 1 4 13 32 41 54 59 77
```

第四遍以后：

1 4 7 12 13 22 28 29 32 33 39 41 45 54 58 59 65 67 76 77 81 84 91 96

注意 在以下部分，我们使用以前的磁盘速度S=1/400, L=1/100, R=1/40秒。

- 9.12 (a) $(1/2)(1/10)(1\ 000\ 000)=50\ 000$ 。
- (c) 对于 $K_{100} > 80$, 1/5; 对于 K_{10K} between 2000 and 3000, $1000/10000 = 1/10$; 对于 $K_5 = 3$, 1/5; 所以 $(1/5)(1/10)(1/5)(1\ 000\ 000) = (1/250)(1\ 000\ 000) = 4000$ 。
- (e) 如同在正文中介绍的那样，在 K_4 的情况下可以进行表空间扫描，且1737次预取I/O提供了 $1737 \times 32 = 55\ 584$ 页。注意BENCH表有55 556页，所以非常接近。我们的经验规则意味着55 584S将花费 $55\ 584/400 = 138.96$ 秒。事实上，我们看到花费了133.27秒，所以读速度比计算的稍快。注意，在开始计I/O时间前，26.90秒的CPU时间并没有从133.27秒中减去：I/O和CPU是重叠的，所以所有的执行时间中都有I/O操作在进行。
- (g) 如同在习题9.4(a)中那样，我们使用嵌套循环连接作为计算的基础。通过 $B1.K_{100}=99$ 限制 $B1$ ，所以检索了10 000行。对于其中的每一行， $B1.K_{250}$ 是一个常数 K 。我们问 $B2$ 中有多少行满足 $B2.K_{500}=K$? 平均为2。这样，现在我们在连接中有20 000行。现在用 $B2.K_{25}=19$ 进行限制，FF为1/25，且 $(1/25)(20\ 000)=800$ 。实际上检索到的行的数目（附录D）为804。
- 9.13 (a) $B1.K_{100}=22$ 选择了 $B1$ 中的10 000行。对于其中的每一行， $B1.K_{250}$ 是一个1到250之间的常数，所以如果该常数大于100K，则 $B1.K_{250}=B2.K_{100}$ 不选择任何内容，否则从 $B2$ 中选择10行。后者可能发生为该时间的40%。10行中仅有1/25平均行即0.4被 $B2.K_{25}=19$ 选择。这样，在连接中的期望值为 $0.4 \times 0.4 \times 10\ 000 = 1600$ 行。
- (b) $B1$ 是外表：在此，10 000个值—— K_{100} 索引的22索引项被读取到RID列表中，允许对 $B1$ 的10 000行进行列表预取需要决定 $B1.K_{100}$ 和 $B1.K_{SEQ}$ ，该数据页面访问的I/O代价为10 000L。索引项表示了 K_{100} 索引的1%，或(使用图9-28)，1051页中的1%，即大约11页。它们可以使用顺序预取，I/O代价为11S。于是对于10 000行中的每一行，内循环中或者使用列表预取访问 $B2$ 的10行(40%的可能)或者访问0行(60%的可能)。这由 $B1.K_{250}$ 是否小于100 000决定。10行被检索出来以后又被谓词 $B2.K_{25}=19$ 过滤，得到在连接中的0.4行。但是这仅在10 000次的40%中发生，即4000次，所以I/O代价为 $4000 \times 10L = 40\ 000L$ 。于是，总的I/O代价为 $10S + 50\ 000L = 50\ 000/100 = 500$ 秒。
- $B2$ 是外表：在此，40 000个值—— K_{25} 索引的19索引项被读取到RID列表中。允许对 $B2$ 的40 000行进行列表预取需要决定 $B2.K_{100}$ 和 $B2.K_{SEQ}$ ，该数据页面访问的I/O代价为40 000L。所有的索引项都是相同的，所以它们每个占用4字节，共160 000字节，即40页，通过顺序预取的I/O代价为40S。然后对于40 000行中的每一行，内循环通过匹配 $B1.K_{250}=B2.K_{100}$ 访问 $B1$ 中大约4行(所有情况下都是如此，这是因为 $B2.K_{100}$ 的值总是在 K_{250} 的范围之内)。匹配还涉及到一个使用 K_{100} 索引、每行I/O代价为1R(总共代价为40 000R)。

的查找过程。4行对于列表预取似乎太小了点，但是我们仍然允许进行列表预取，所以我们计算为 $40\ 000 \times 40L = 160\ 000L$ 。这些行进一步被谓词 $B1, K100=22$ 筛选。总的I/O代价为 $40S + 200\ 000L + 40\ 000R = 3000$ 秒。

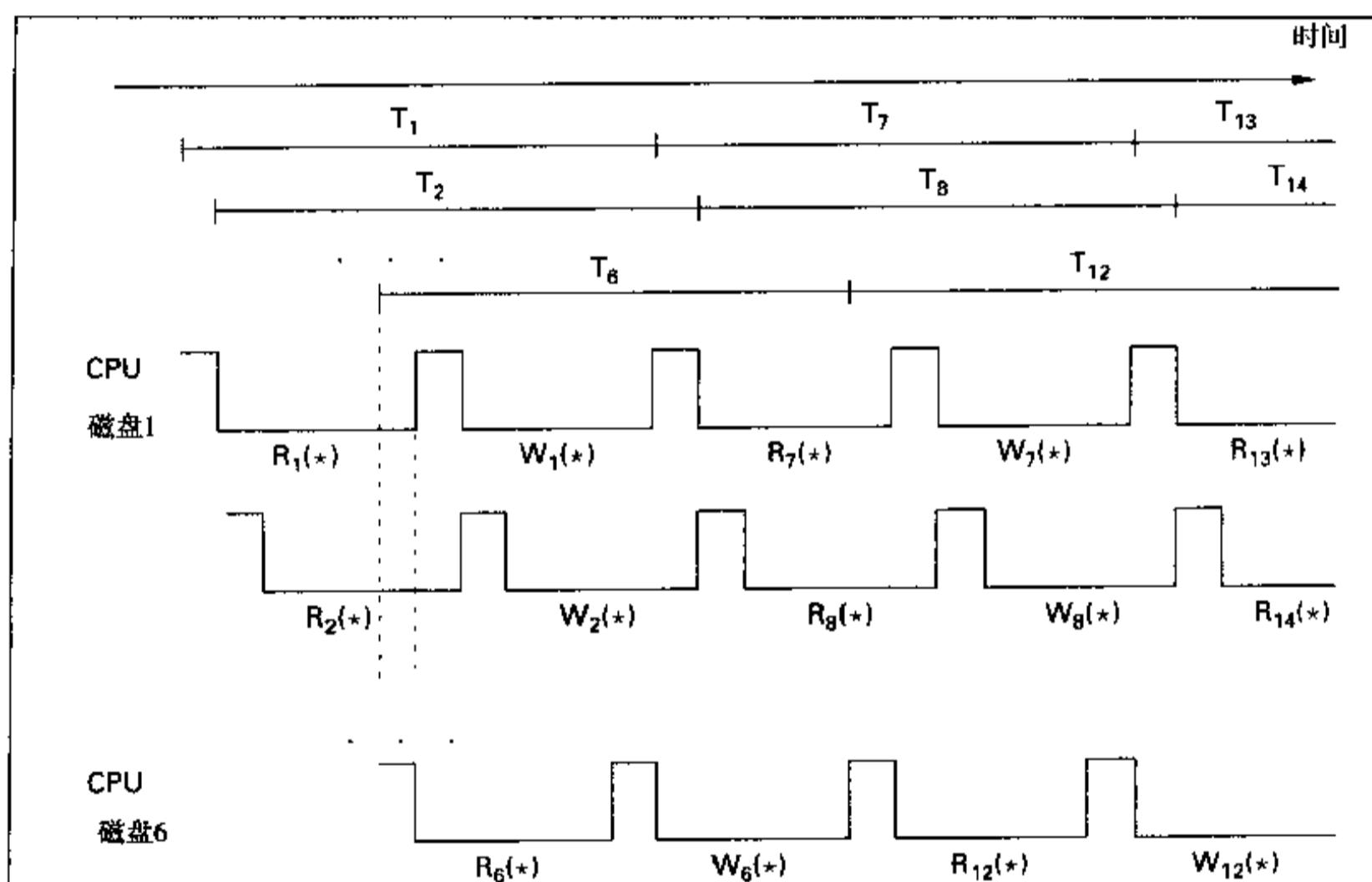
9.15 (a) select npages from systables where name = 'BENCH' ;

(参见图9-13)

(c) K2X索引几乎完全一样，所以有4字节/项，即大约1000项/页。在叶子层有1 000 000项，因此是1000页。

第10章习题解答

10.1 (a) 在120ms (2个I/O和2个CPU块) (即50TPS) 中有六个事务。



10.2 (a) $T_2 \rightarrow T_3 \rightarrow T_1$

10.3 操作号	R ₁ (A)	W ₁ (A)	R ₂ (A)	R ₂ (B)	C ₂	R ₁ (B)	W ₁ (B)	C ₁
	1	2	3	4	5	6		

操作2和3给出了 $T_1 \rightarrow T_2$ ，而操作4和6给出了 $T_2 \rightarrow T_1$ 。前趋图为：



为了回答问题的第二部分，假设A和B是一般只能看到存款的账户，而 T_1 的任务是保证账户A不超过100——如果A超过了100，那么 T_1 的任务就是把A的值降到50，并移走超额的部分到账户B。我们假设 T_2 的任务是检查账户A和B的总额，如果总额超过200，就将A和B的总额

减到150，并将超额部分移到第三方余额：C。现在假设开始时A=120，B=140。由于T₁并没有改变A和B的总额，我们可以看到任何一种串行化执行中T₂都将从A和B中减去部分加到C中。但是，我们真正看到的是：

R₁(A, 120) W₁(A, 50) R₂(A, 50) R₂(B, 140) C₂ R₁(B, 140) W₁(B, 210) C₁

我们注意到T₂看到A+B至多为50+140=190。由于该值小于200，T₂不执行任何动作。在任何一种串行调度中，这种情况都不会出现。

10.5 这是一个“脏写”的例子。在此，每一个事务回写新的A值都没能考虑到其他事务的更新操作。锁调度器将它们翻译为如下事件序列：

RL₁(A) R₁(A) RL₂(A) R₂(A) WL₁(A) (和前面的RL₂(A)冲突——T₁必须等待WL₂(A) (和前面的RL₁(A)冲突——在等待图中循环——选择T₂作为牺牲者) A₁ (WL₁(A)现在成功了) W₁(A) C₁ (T₂现在以事务T₃的形式重做) RL₃(A) R₃(A) WL₃(A) W₃(A) C₃

我们看到T₁和T₂（被重命名为T₃）的操作以串行的顺序发生。

10.6 (a) WL₃(A) W₃(A) RL₁(A) (和前面的WL₃(A)冲突——T₁必须等待T₃) RL₂(B) R₂(B) WL₂(Y) W₂(Y) WL₃(B) (和前面的RL₂(B)冲突——T₃必须等待T₂) C₂ (释放对B的锁，所以WL₃(B)就成功了) W₃(B) C₁ (释放对A的锁，所以RL₁(A)成功了) R₁(A) WL₁(Z) W₁(Z) C₁

W ₃ (A)	R ₁ (A)	W ₁ (Z)	R ₂ (B)	W ₂ (Y)	W ₃ (B)	C ₁ C ₂ C ₃
操作号	1	2	3	4	5	6

注意，从操作1和2中可以看到在等待图中T₁→T₂。接着从4和6中可以得到T₃→T₂。此外没有其他操作对是对相同的元素进行处理的。所以没有其他的冲突对，且等待图中没有环，于是不存在死锁。在操作6以后的等待图为：

T₁→T₄→T₂

- 10.7 (a) R₃(E, 8)。这是因为在此之前有A、B、C、D的页面都已经被访问过并填满了缓冲区中的四个页面。
 (c) 有数据项A的页面将被LRU模式丢弃。这是因为它具有最长的引用间隔。
 (e) 不会，它仅仅强制写出该事物的日志数据。而真正的脏的页面可能继续存在于缓冲区中，直到某个合适的时间才被写出。

- 10.8 (a) (i) 下面是作为该经历的结果被写的一系列日志项：

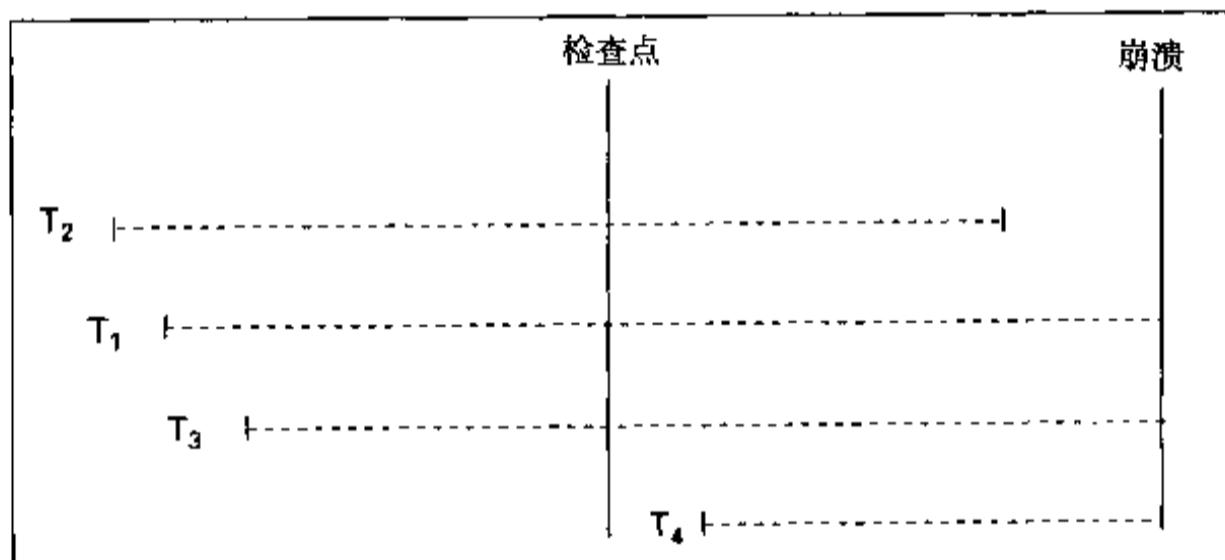
(ii) 最后的写操作将不写到日志文件中，这是由于没有Commit强制写它。现在我们略述恢复时所进行的动作。

Rollback

- | | |
|------------------------------|-------------------------------|
| 1. (C, 2) | 将T2作为已经提交的事务放入活动列表。 |
| 2. (W, 2, E, 8, 9) | 无动作。 |
| 3. (S, 4) | 无动作。 |
| 4. (CKPT, LIST = T1, T2, T3) | 活动列表=(T1(NC), T2(C), T3(NC))。 |
| 5. (W, 3, C, 4, 6) | UNDO: C=4。 |
| 6. (W, 3, A, 1, 4) | UNDO: A=1。 |
| 7. (S, 3) | 活动列表=(T1(NC), T2(C))。 |

- 8.(W, 1, B, 2, 3) UNDO: B=2。
 9.(S, 1) 活动列表=(T2(C)), 只剩下已经提交的, ROLLBACK结束。
- Roll forward(跳到检查点后的第一个日志项)**
- 10.(S, 4) 无动作。活动的已提交事务T2。
 11.(W, 2, E, 8, 9) REDO: E=9。
 12.(C, 2) 无动作。没有其他活动的已提交事务。ROLL FORWARD结束。

(iii)



10.9 首先我们说明为什么这些都是事务持续可能出现的情况。每一个事务必须开始和结束, 开始必须在结束之前, 且开始的时间有三种可能: (1)在最后的检查点发生之前 (在下一个习题中, 我们将说明如果在两个检查点之前开始也不会有任何区别); (2)在最后的检查点之后, 但是在崩溃之前; (3)在崩溃时 (在此时刻至少可能发生一个停止动作)。所以, 看上去所有可能的开始—停止时刻集合似乎为(1-1)、(1-2)、(1-3)、(2-2)、(2-3)和(3-3)。但是事实上(3-3)是不可能的, 而每一种其他情况按序都被事务T₁到T₄之一所覆盖。

现在我们说明每一种情况都能被具有高速缓存一致性检查点的恢复过程恢复。

(1-1)事务的开始和结束都在最后的检查点发生之前发生。在恢复时, 我们并不对这些数据项进行任何UNDO和REDO。这是因为事务在进行检查点时已经不是活动的了, 所以在检查点之后没有发生写操作。事实上, 该事务的所有更新操作都在最后的检查点之前发生, 而检查点强制所有的“脏”数据项写入磁盘。这样, 该事务的所有更新都已经被换出到磁盘上, 于是我们在恢复时不用对它们做任何修改。

(1-2)事务在最后一个检查点之前开始, 但是在它之后结束。注意在ROLLBACK时, 我们首先看到提交日志项, 所以在ROLL FORWARD之前对写日志不进行任何动作。在ROLL FORWARD时, 从检查点之后开始, 我们REDO该事务的所有对在磁盘上的数据项的写日志项。在检查点之前发生的更新都已经被写出到磁盘。

(1-3)事务在最后一个检查点之前开始, 并由于崩溃而结束。由于事务被异常终止, 我们应该UNDO它执行的所有写操作。注意, 在ROLLBACK时, 我们一开始或者先看到一个写日志 (且不是Commit), 或者先看到最后一个检查点的时间, 且此时该事务仍然是活动的 (且由于我们没有看到Commit日志, 该事务仍未被提交)。我们必须UNDO该事务的所有写项。如果有一个写日志项在最后一个检查点之后发生, 我们马上知道这是一个未提交的事务, 而

且能够UNDO该日志项的动作。如果该事务在检查点之后没有留下任何写日志项，我们仍然知道在检查点时它是活动的，所以可以继续ROLLBACK直到我们看到事务的Start日志为止，将对所有的写日志进行UNDO，直到回到那一点。

(2-2)事务在最后一个检查点之后开始，并在它之后结束。注意，在ROLLBACK时，我们先看見Commit日志项，所以在ROLL FORWARD之前对写日志不进行任何动作。在ROLL FORWARD时，从检查点之后开始，我们REDO该事务对在磁盘上的数据项的所有的写日志项。这正是所期望的。

(2-3)事务在最后一个检查点之后开始，并由于崩溃而结束。由于事务异常终止，我们必须UNDO它所进行的所有写操作。注意在ROLLBACK时，我们首先会看到一个写日志（且不是Commit），又由于我们没有看到Commit日志，所以该事务没有被提交。当我们看到一个写日志项的时候，我们马上知道这是一个未提交的事务，并可以UNDO该日志项以及所有其他写日志项的动作直到Start日志为止。

最后还有一点要考虑。我们是否会对同一个数据项进行不同的动作UNDO-UNDO、UNDO-REDO或REDO-REDO，从而相互影响，导致最后存在错误的数据项？我们首先考虑进行UNDO操作，我们知道该活动是由于事务对数据项的更新没有完成，必须异常终止，从而在回滚时进行的。通过锁机制，如果我们假设我们持有所有锁直到提交为止，我们知道在其后的某个时间点没有其他事务对该数据项进行了更新。其他UNDO操作只可能是由同一个事务引起的，并最终会把值设为该事务最早所见的值。将来对该数据项的REDO操作（如果有的话）必定发生得更早，而且显然最后的REDO动作（如果有的话）将设置和最早的UNDO操作所设的相同的值。所以所有已提交的事务已经完成了它们对该数据项的更新（仅有一个），而任何未提交事务对它们没有任何作用。如果我们考虑首先进行REDO动作，通过锁机制我们知道不会有更早的UNDO动作，所以最终的REDO操作设置了最终的值。我们根据锁原则下的经历的可串行性知道这是正确的值。

$$10.11 \quad H = W_4(A) R_1(A) C_1 W_2(B) R_4(B) W_5(C) R_2(C) C_2 W_3(D) R_5(D) C_3 C_4 C_5 \\ T_4 \rightarrow T_1 \quad T_2 \rightarrow T_4 \quad T_5 \rightarrow T_2 \quad T_3 \rightarrow T_5$$

$$PG(H) = T_3 \rightarrow T_5 \rightarrow T_2 \rightarrow T_4 \rightarrow T_1$$

因此， $PG(H)$ 不包含环，根据定理10.3.4可知 H 是可串行化的。等价的串行经历 $S(H)$ 为：

$$W_3(D) C_3 W_5(D) C_5 W_2(B) R_2(C) C_2 W_4(A) R_4(B) C_4 R_1(A) C_1$$

(注意，锁机制将把 T_1 ，然后是 T_4 、 T_2 以及 T_5 转入等待状态，所以在 C_1 时，仅有 T_3 是活动的，它将释放 T_1 ，依此类推。)

第11章习题解答

- 11.1 (a) 错。主锁表和主事务表可以在共享内存中，所以锁和事务状态是系统范围同步的，并且不需要两阶段提交。
- (b) 对。假设查询仅需要读锁。其他只读的进程就不需要等待这些锁。
- (c) 错。服务器可以通过到不同处理器的消息来实现一个服务。
- (d) 对。一个60MIPS的CPU将对应于图11-6中的一个点，而300MIPS的CPU将对应于水平轴方向五倍远的一个点。五个60MIPS的CPU将成为在原点到第一个点的射线上的一个点，该射线扩展到300MIPS的标记为止。由于该曲线越来越向上弯，300MIPS的CPU将更高一些。
- (e) 对。参见图11-9，崩溃状态反馈进入准备状态。

计算机科学丛书

《编译原理及实践》	Louden 著/冯博琴 等译/39.00元
《UNIX环境高级编程》	Stevens 著/尤晋元 等译/55.00元
《分布式操作系统：概念与实践》	Galli 著/尤晋元 等译/
《分布式系统设计》	Jie Wu 著/高传善 等译/30.00元
《现代操作系统》	Tanenbaum 著/陈向群 等译/40.00元
《UNIX操作系统设计》	Bach 著/陈葆珏 等译/33.00元
《UNIX编程环境》	Kernighan 等著/陈向群 等译/24.00元
《C语言解析》(原书第4版)	Pohl 著/麻志毅 等译/
《C程序接口与实现》	Hanson 著/
《C程序设计教程》(原书第2版)	Deitel 等著/薛万鹏 等译/33.00元
《C++程序设计教程》(原书第2版)	Deitel 等著/薛万鹏 等译/22.00元
《编码的奥秘》	Petzold 著/陆丽娜 等译/24.00元
《编码的奥秘》(原书第2版)	Petzold 著/陆丽娜 等译/
《C++核心：软件工程方法》	Shtern 著/李师贤 等译/
《Java程序设计导引》	Liang 著/王镁 等译/
《Java语言解析》	Pohl 著/
《Java程序设计教程》(原书第3版)	Deitel 著/袁兆山 等译/
《大型C++软件设计》	Lakos 著/李师贤 等译/
《C++语言的设计和演化》	Stroustrup 著/裘宗燕 译/
《C++编程思想》	Eckel 著/刘宗田 等译/39.00元
《C++编程思想》(原书第2版)	Eckel 著/刘宗田 等译/
《Java编程思想》	Eckel 著/京京工作室 译/39.00元
《Java编程思想》(原书第2版)	Eckel 著/京京工作室 译/
《电子商务》	Schneider 等著/成栋 译/28.00元
《多媒体应用程序的面向对象设计》	Guzdial 著/
《计算机文化》(原书第3版)	Parsons 著/朱海滨 等译/50.00元
《图论导引》(原书第2版)	West 著/
《专家系统原理与编程》(原书第3版)	Giarratano 著/印鉴 等译/49.00元
《神经网络》	Haykin 著/史忠植 等译/
《人机接口》	Raskin 著/
《设计模式：可复用面向对象软件的基础》	Gamma 等著/吕建 等译/35.00元
《软件工程：JAVA语言实现》(原书第4版)	Schach 著/袁兆山 等译/38.00元
《软件工程：实践者的研究方法》(原书第4版)	Pressman 著/黄柏素、梅宏 译/48.00元
《软件需求》	Wiegers 著/陆丽娜 等译/19.00元
《数据库系统导论》(原书第7版)	Date 著/孟小峰 等译/66.00元
《数据仓库》(原书第2版)	Inmon 著/王志海 等译/25.00元
《数据挖掘：概念与技术》	Jiawei Han等著/范明 等译/39.00元
《数据库系统概念》(原书第3版)	Silberschatz 等著/杨冬青 等译/49.00元
《数据库系统实现》	Garcia-Molina 等著/杨冬青 等译/45.00元

《数据库设计》	Stephens 等著/何玉洁 等译/35.00元
《数据库管理系统基础》	Pratt 等著/陆洪毅 等译/20.00元
《事务处理：概念与技术》	Gary 等著/孟小峰 等译/
《数字逻辑：应用与设计》	Yarbrough 著/朱海滨 等译/49.00元
《嵌入式计算机系统设计原理》	Wolf 著/孙玉芳 等译/
《并行程序设计》	Wilkinson 著/陆鑫达 等译/
《最新网络技术基础》	Palmer 著/严伟 译/20.00元
《计算机网络与因特网》(原书第2版)	Comer 著/徐贤良 等译/40.00元
《计算机网络》(原书第2版)	Peterson 等著/叶新铭 等译/49.00元
《光纤通信技术》	Mynbaev 等著/吴时霖 等译/40.00元
《高性能通信网络》(原书第2版)	Walrand 等著/史美林 等译/
《数据通信与网络》	Forouzan 等著/潘允 等译，吴时霖 校/48.00元
《ISDN、B-ISDN与帧中继和ATM》(原书第4版)	Stallings 著/程时端 等译/48.00元
《计算机网络实用教程》	Dean 著/陶华敏 等译/65.00元
《计算机网络实用教程实验手册》	Dean 著/陶华敏 等译/15.00元
《网络》(原书第2版)	Ramteke 著/侯春萍 等译/
《TCP/IP详解 卷1：协议》	Stevens 著/范建华 等译，谢希仁 校/ 45.00元
《TCP/IP详解 卷2：实现》	Wright/Stevens著/陆雪莹 等译，谢希仁 校/78.00元
《TCP/IP详解 卷3：TCP事务协议、HTTP、NNTP和UNIX域协议》	Stevens 著/胡谷雨 等译，谢希仁 校/ 35.00元
《Internet技术基础》(原书第2版)	Comer 著/袁兆山 等译/18.00元
《数据通信与网络教程》(原书第2版)	Shay 著/高传善 等译/40.00元
《密码学导引》	Garrete 著/
《分布计算的信息安全》	Bruce 著/
《计算机信息处理》(原书第7版)	Mandell 等著/尤晓东 等译/38.00元
《信息系统原理》(原书第3版)	Stair 等著/张靖 等译/42.00元

国外经典教材

《编译原理》	Aho 著/李建中 等译/
《Linux操作系统内核实习》	Nutt 著/陆丽娜 等译/
《操作系统原理与实践》(原书第2版)	Nutt著/尤晋元 等译/
《现代操作系统》(原书第2版)	Tanenbaum 著/陈向群 等译/
《C++程序设计语言》(原书第3版，特别版)	Stroustrup 著/裘宗燕 译/
《C程序设计语言》(原书第2版)	Kernighan/Ritchie 著/徐宝文 等译/28.00元
《程序设计实践》	Kernighan/Pike 著/裘宗燕 译/20.00元
《程序设计语言：概念与结构》(原书第2版)	Sethi 著/裘宗燕 等译/
《计算理论导引》	Sipser 著/张立昂 等译/30.00元
《离散数学及其应用》(原书第4版)	Rosen 著/袁崇义 等译/
《组合数学》(原书第3版)	Brualdi 著/冯舜玺 等译/

《人工智能》	Nilsson 著/郑扣根 等译/30.00元
《神经网络设计》	Hagan 等著/戴葵 等译/
《软件工程：实践者的研究方法》(原书第5版)	Pressman 著/梅宏 等译/
《数据结构、算法与应用：C++语言描述》	Sahni 著/戴葵 等译/49.00元
《数据结构与算法分析：C语言描述》(原书第2版)	Weiss 著/冯舜玺 等译/
《数据库系统原理》(原书第4版)	Siblerschatz 等著/杨冬青 等译/
《数据库原理、编程与性能》(原书第2版)	O' Neil 等著/周傲英 等译/
《并行计算机体系结构》	Culler/Singh/Gupta 著/李晓明 等译/
《结构化计算机组成》	Tanenbaum 著/刘卫东 等译/46.00元
《可扩展并行计算：技术、结构与编程》	Hwang/Xu 著/陆鑫达 等译/49.00元
《计算机图形学的算法基础》(原书第2版)	Rogers 著/石教英 等译/
《数据通信与网络》(原书第2版)	Forouzan 等著/吴时霖 等译/

经典原版书库

《UNIX环境高级编程》(英文版)	Stevens 著/
《现代操作系统》(英文版·第2版)	Tanenbaum 著/
《程序设计语言：概念与结构》(英文版·第2版)	Sethi 著/
《C程序设计语言》(英文版·第2版)	Kernighan/Ritchie 著/
《C++语言的设计和演化》(英文版)	Stroupstrup 著/
《程序设计实践》(英文版)	Kernighan/Pike 著/
《C++编程思想》(英文版·第2版)	Eckel 著/
《Java编程思想》(英文版·第2版)	Eckel 著/
《离散数学及其应用》(英文版·第4版)	Rosen 著/59.00元
《组合数学》(英文版·第3版)	Brualdi 著/
《人工智能》(英文版)	Nilsson 著/45.00元
《设计模式：可复用面向对象软件的基础》(英文版)	Gamma/Helm/Johnson/Vlissides 著/
《软件工程：Java语言实现》(英文版·第4版)	Schach 著/51.00元
《软件工程：实践者的研究方法》(英文版·第4版)	Pressman 著/68.00元
《系统分析与设计》(英文版)	Satzinger/Jackson 著/60.00元
《数据结构、算法与应用——C++语言描述》(英文版)	Sahni 著/66.00元
《数据库系统导论》(英文版·第7版)	Date 著/
《数据库系统概念》(英文版·第3版)	Silberschatz/Korth/Sudarshan 著/65.00元
《数据库系统实现》(英文版)	Molina/Ullman/Widom 著/
《高级计算机体系结构》(英文版)	Hwang 著/59.00元
《计算机体系结构：量化研究方法》(英文版·第2版)	Patterson/Hennessy 著/88.00元
《计算机组织和设计：硬件/软件方法》(英文版·第2版)	Hennessy/Patterson 著/80.00元
《并行计算机体系结构》(英文版·第2版)	Culler/Singh/Gupta 著/88.00元
《可扩展并行计算：技术、结构与编程》(英文版)	Hwang/Xu 著/69.00元
《结构化计算机组成》(英文版·第4版)	Tanenbaum 著/

《计算机图形学的算法基础》(英文版·第2版)	Rogers 等著/
《通信网络基础》(英文版·第2版)	Walrand 著/32.00元
《计算机网络》(英文版·第2版)	Peterson/Davie 著/65.00元
《高性能通信网络》(英文版·第2版)	Walrand/Varaiya 著/64.00元
《网络互连：网桥·路由器·交换机和互连协议》 (英文版·第2版)	Perlman 著/
《数据通信与网络》(英文版)	Forouzan 等著/59.00元
《ISDN、B-ISDN与帧中继和ATM》(英文版·第4版)	Stallings 著/
《TCP/IP详解 卷1：协议》(英文版)	Stevens 著/
《TCP/IP详解 卷2：实现》(英文版)	Wright/Stevens 著/
《TCP/IP详解 卷3：TCP事务协议、HTTP、 NNTP和UNIX域协议》(英文版)	Stevens 著/
《Internet技术基础》(英文版·第3版)	Comer 著/

[G e n e r a l I n f o r m a t i o n]

书名 = 数据库原理、编程与性能

作者 =

页数 = 591

S S 号 = 10440574

出版日期 =

[封面](#)
[书名](#)
[版权](#)
[前言](#)
[目录](#)
[正文](#)
[封底](#)