计 算 机 科 学 丛 书

原书第6版

Education

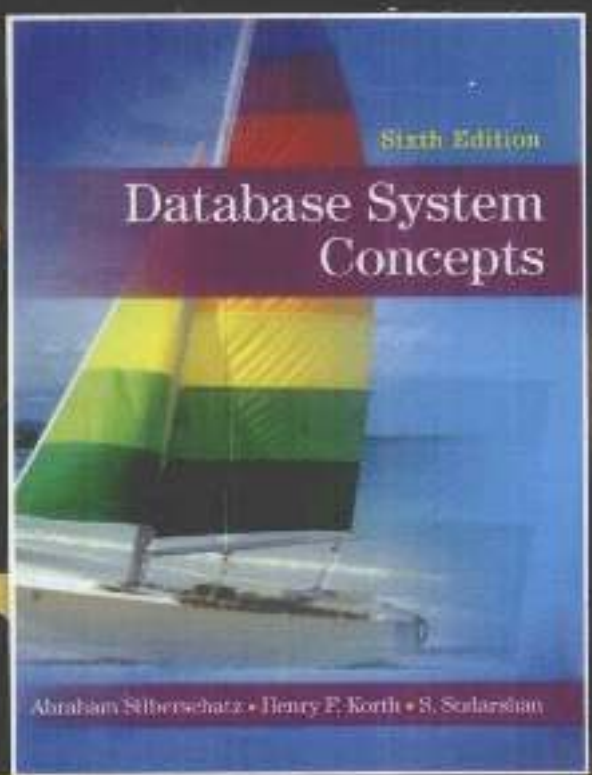
数据库系统概念

Abraham Silberschatz 耶鲁大学 Henry F. Korth 利哈伊大学 S. Sudarshan 印度理工学院 著

杨冬青 李红燕 唐世渭 北京大学 等译

Database System Concepts

Sixth Edition



机械工业出版社
China Machine Press

数据库系统概念（原书第6版）

Database System Concepts Sixth Edition

数据库领域的殿堂级作品

夯实数据库理论基础，增强数据库技术内功的必备之选

对深入理解数据库、深入研究数据库、深入操作数据库都具有极强的指导作用！

本书是数据库系统方面的经典教材之一，其内容由浅入深，既包含数据库系统基本概念，又反映数据库技术新进展。它被国际上许多著名大学所采用，包括斯坦福大学、耶鲁大学、得克萨斯大学、康奈尔大学、伊利诺伊大学等。我国也有多所大学采用本书作为本科生和研究生数据库课程的教材和主要教学参考书，收到了良好的效果。

第6版保持了前5版的总体风格，同时对内容进行了扩充，对结构进行了调整，以更好地符合数据库教学的需求和反映数据库设计、管理与使用方式的发展和变化。具体更新内容如下：

- 调整了内容组织结构，将SQL内容提前，并集中进行介绍。
- 采用一个新的模式（基于大学的数据）作为贯穿全书的运行实例。
- 修订和更新了对数据存储、索引和查询优化以及分布式数据库的涵盖。
- 修订了E-R模型、关系设计和事务管理等内容。
- 扩充了关于应用开发和安全性的素材。

本书配套网站（http://www.db-book.com）提供的教辅资源包括：

- 书中各章的教学课件。
- 实践练习的答案。
- 未放入纸版书中的四个附录（高级关系数据库设计、其他关系查询语言、网状模型、层次模型）。
- 实验素材（包括大学模式和习题中用到的其他关系的SQL DDL和样例数据，以及关于建立和使用各种数据库系统和工具的说明书）。
- 最新勘误表。

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

Education
http://www.mheducation.com

ISBN 978-7-111-37529-6

9 787111 375296

定价：99.00元

计 算 机 科 学 丛 书

原书第6版

Education

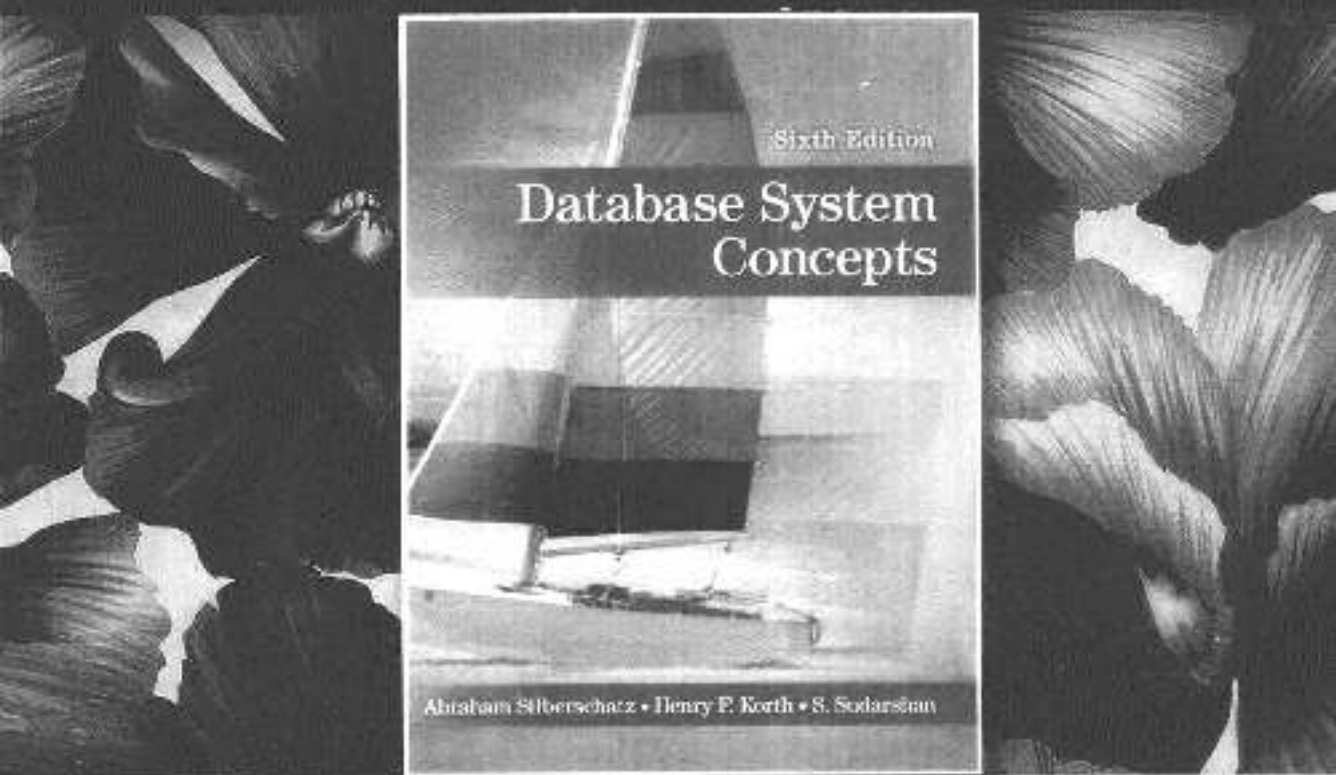
数据库系统概念

Abraham Silberschatz 耶鲁大学 Henry F. Korth 利哈伊大学 S. Sudarshan 印度理工学院 著

杨冬青 李红燕 唐世渭 北京大学 等译

Database System Concepts

Sixth Edition



机械工业出版社
China Machine Press

本书是经典的数据库系统教科书《Database System Concepts》的最新修订版，全面介绍数据库系统的各种知识，透彻阐释数据库管理的基本概念。本书内容丰富，不仅讨论了关系数据模型和关系语言、数据库设计过程、关系数据库理论、数据库应用设计和开发、数据存储结构、数据存取技术、查询优化方法、事务处理系统和并发控制、故障恢复技术、数据仓库和数据挖掘，而且对性能调整、性能评测标准、数据库应用测试和标准化、空间和地理数据、时间数据、多媒体数据、移动和个人数据库管理以及事务处理监控器、事务工作流、电子商务、高性能事务系统、实时事务系统和持续长时间的事务等高级应用主题进行了广泛讨论。

本书既可作为高年级本科生或低年级研究生的数据库课程教材，也可供数据库领域的技术人员参考。

Abraham Silberschatz, Henry F. Korth, S. Sudarshan: Database System Concepts, Sixth Edition (ISBN 978-0-07-352332-3).

Copyright © 2011 by The McGraw-Hill Companies, Inc.

All Rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including without limitation photocopying, recording, taping, or any database, information or retrieval system, without the prior written permission of the publisher.

This authorized Chinese translation edition is jointly published by McGraw-Hill Education (Asia) and China Machine Press. This edition is authorized for sale in the People's Republic of China only, excluding Hong Kong, Macao SAR and Taiwan.

Copyright © 2012 by McGraw-Hill Education (Asia), a division of the Singapore Branch of The McGraw-Hill Companies, Inc. and China Machine Press.

版权所有。未经出版人事先书面许可，对本出版物的任何部分不得以任何方式或途径复制或传播，包括但不限于复印、录制、录音，或通过任何数据库、信息或可检索的系统。

本授权中文简体字翻译版由麦格劳－希尔（亚洲）教育出版公司和机械工业出版社合作出版。此版本经授权仅限在中华人民共和国境内（不包括香港特别行政区、澳门特别行政区和台湾）销售。

版权 © 2012 由麦格劳－希尔（亚洲）教育出版公司与机械工业出版社所有。

本书封面贴有 McGraw-Hill 公司防伪标签，无标签者不得销售。

封底无防伪标均为盗版

版权所有，侵权必究

本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2010-3824

图书在版编目（CIP）数据

数据库系统概念（原书第6版）/（美）西尔伯沙茨（Silberschatz, A.）等著；杨冬青等译. —北京：机械工业出版社，2012.3

（计算机科学丛书）

书名原文：Database System Concepts, Sixth Edition

ISBN 978-7-111-37529-6

Ⅰ.数… Ⅱ.①西… ②杨… Ⅲ.数据库系统－高等学校－教材 Ⅳ.TP311.13

中国版本图书馆 CIP 数据核字（2012）第028157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：李　荣　　谢晓芳

北京诚信伟业印刷有限公司印刷

2012年3月第1版第1次印刷

185mm×260mm · 52.25印张

标准书号：ISBN 978-7-111-37529-6

定价：99.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991；88361066

购书热线：（010）68326294；88379649；68995259

投稿热线：（010）88379604

读者信箱：hzjsj@hzbook.com

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突出性的优势；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，不仅奠划了研究的范畴，还揭示了学术的源流，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和发展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到积极的推动作用，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公司较早意识到“出版要为教育服务”。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennie Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson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、研究及珍藏。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，有的还专门为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，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。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

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。随着计算机科学与技术专业学科建设的不断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将步入一个新的阶段，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。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：

华章网站：www.hzbook.com
电子邮件：hzjsj@hzbook.com
联系电话：（010）88379604
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

华章教育
华章科技图书出版中心

译者序

Database System Concepts, 6E

数据库系统是对数据进行存储、管理、处理和维护的软件系统，是现代计算环境中的一个核心成分。随着计算机硬件、软件技术的飞速发展和计算机系统在各行各业的广泛应用，数据库技术的发展尤其迅速，引人注目。有关数据库系统的理论和技术是计算机科学技术教育中必不可少的部分。《数据库系统概念》是一本经典的、备受赞扬的数据库系统教科书。其内容由浅入深，既包含数据库系统的基本概念，又反映数据库技术的新进展。本书被国际上许多著名大学所采用，并多次再版。

我们先后将本书的第3版、第4版和第5版译成中文，由机械工业出版社分别于2000年、2003年和2006年出版发行。国内许多大学采用《数据库系统概念》作为本科生和研究生数据库课程的教材或主要教学参考书，收到了良好的效果。现在，我们又翻译了该书第6版。第6版保持了第5版的总体风格，同时对内容进行了扩充，对结构进行了调整，以更好地符合数据库教学的需求，反映数据库设计、管理和使用方式的发展和变化。第6版的内容大体上可以分为五个部分。

第1～9章讲述数据库系统的基本概念，包括对数据库系统的作用和目标的综述，对关系数据模型和关系语言的介绍，对数据库设计过程、关系数据库理论以及数据库应用设计和开发（包括基于Web的界面构建数据库应用和应用安全性问题等）的详细讨论。

第10～19章主要讨论数据库系统实现技术，包括数据存储结构、数据存取技术、查询优化方法、事务处理系统的基本概念和并发控制、故障恢复技术，还包括在并行数据库系统和分布式数据库系统中所采用的一些主要策略和技术。

第20～23章主要讨论数据管理与应用的深入话题，包括对数据仓库和数据挖掘概念与技术的较详细的介绍，以及对用于查询文本数据的信息检索技术（包括在Web搜索引擎中使用的处理超链接的技术）的介绍。这一部分还介绍了新型的数据库系统，包括对象－关系数据库模型、数据表示的XML标准，以及XML的查询语言。

第24～26章是一些高级话题，内容包括应用开发中的诸如性能调整、性能评测标准、数据库应用测试和标准化等高级话题，以及空间和地理数据、时间数据、多媒体数据、移动和个人数据库管理中的问题。这一部分还讨论了事务处理监控器、事务工作流、电子商务、高性能事务系统、实时事务系统和持续长时间的事务等高级事务处理问题。

第27～30章对PostgreSQL、Oracle、IBM DB2和Microsoft SQL Server这四个领先的数据库系统进行实例研究，结合这几个具体系统来讨论前面各部分描述的各种实现技术是如何使用到实际系统中的。

上述五大部分中第一部分的主要内容，以及第二、第三、第四部分的部分内容可以作为本科生数据库概论课程的教材或主要参考资料，第二、第三和第四部分的其余内容可以用于研究生的数据库课程教学，第五部分可以作为帮助学生了解实际系统的补充材料。

杨冬青、李红燕、唐世渭组织并参加了本书的翻译和审校工作；参加翻译的还有包红杰、程学、苗高彬、李晓、张颖、王婧、王林青、孟必平。

限于译者水平，译文中疏漏和错误难免，欢迎批评指正。

译者

于北京大学

数据库管理已经从一种专门的计算机应用发展为现代计算环境中的一个重要成分，因此，有关数据库系统的知识已成为计算机科学教育中的一个核心的部分。在本书中，我们讲述数据库管理的基本概念。这些概念包括数据库设计、数据库语言、数据库系统实现等多个方面。

本书可作为本科生三年级或四年级数据库入门课程的教科书，也可作为研究生一年级的教科书。除了作为入门课程的基本内容外，本书还包括了可作为课件补充或作为高级课程介绍性材料的高级内容。

我们仅要求读者熟悉基本的数据结构、计算机组织结构和一种高级程序设计语言，例如 Java、C 或 Pascal。书中的概念都以直观的方式加以描述，其中的许多概念都基于我们大学运行的例子加以解释。本书中包括重要的理论结果，但省略了形式化证明，取而代之的是用图表和例子来说明为什么结论是正确的。对于形式化描述和研究结果的证明，读者可以参考文献注解中列出的研究论文和高级教材。

本书中所包括的基本概念和算法通常是基于当今的商品化或试验性的数据库系统中采用的概念和算法。我们的目标是在一个通常环境下描述这些概念和算法，而没有与某个特定的数据库系统绑定。特定的数据库系统的细节将在第九部分“实例研究”中讨论。

在本书第 6 版中，我们保持了前面版本的总体风格，同时对内容和结构进行了扩展来反映数据库设计、管理和使用的方式所发生的变化。我们还考虑了数据库概念的教学方面的趋势，并在适当的地方做出了推动这种趋势的修改。

本书的组织

本书组织成十个主要部分：

- **综述**（第 1 章）。第 1 章对数据库系统的性质和目标进行了一般性综述。我们解释了数据库系统的概念是如何发展的，各数据库系统的共同特性是什么，数据库系统能为用户做什么，以及数据库系统如何与操作系统交互。我们还引入了一个数据库应用的例子：包括多个系、教员、学生和课程的一个大学机构。这个应用作为贯穿全书的运行实例。这一章本质上是描述性、历史性和解释性的。
- **第一部分：关系数据库**（第 2 章至第 6 章）。第 2 章介绍了数据的关系模型，包括基本概念，诸如关系数据库的结构、数据库模式、码、模式图、关系查询语言和关系操作等。第 3 ~ 5 章主要介绍最具影响力的面向用户的关系语言——SQL。第 6 章介绍形式化的关系查询语言，包括关系代数、元组关系演算和域关系演算。

  这部分描述了数据操纵，包括查询、修改、插入和删除（假设已有一个模式设计）。关于模式设计的问题延迟到第二部分讲述。
- **第二部分：数据库设计**（第 7 章至第 9 章）。第 7 章给出了数据库设计过程的概要介绍，主要

侧重于用实体－联系数据模型来进行数据库设计。实体－联系模型为数据库设计问题，以及我们在数据模型的约束下捕获现实应用的语义时所遇到的问题提供了一个高层视图。UML 类图表示也在这一章中讲述。

第 8 章介绍关系数据库设计理论。这一章讲述函数依赖和规范化，重点强调提出各种范式的动机，以及它们的直观含义。这一章以关系设计的概览开始，依赖于对函数依赖的逻辑蕴涵的直观理解。这使得规范化的概念可以在函数依赖理论的完整内容之前先作介绍。函数依赖理论将在本章中稍后部分讨论。教师可以只选用 8.1 节至 8.3 节这些较前面的章节，而不会丢失连贯性。不过，完整地讲授这一章将有利于学生对规范化概念形成较好的理解，从而诱导出函数依赖理论中一些较艰深的概念。

第 9 章讲述应用设计和开发。这一章侧重于用基于 Web 的界面构建数据库应用。另外，这一章还讲述了应用安全性。

- **第三部分：数据存储和查询**（第 10 章至第 13 章）。第 10 章讨论存储设备、文件和数据存储结构。在第 11 章中介绍多种数据存取技术，包括 B+ 树索引和散列。第 12 章和第 13 章简述查询执行算法和查询优化。这两章使用户能更好地理解数据库的存储和检索的内部机制。
- **第四部分：事务管理**（第 14 章至第 16 章）。第 14 章首先介绍事务处理系统的基本概念，包括原子性、一致性、隔离性和持久性。它还提供了用于保证这些特性的方法的一个概述，包括封锁和快照隔离性。

  第 15 章重点讲述并发控制，并介绍保证可串行化的几种技术，包括封锁、时间戳和乐观（有效性检查）技术。在这一章中还讨论死锁问题，并介绍保证可串行化的其他方法，特别是详细讨论广泛使用的快照隔离方法。

  第 16 章讨论在系统崩溃和存储器故障情况下保证事务正确执行的主要技术。这些技术包括日志、检查点和数据库转储。被广泛使用的 ARIES 算法也在这里做了介绍。
- **第五部分：系统体系结构**（第 17 章至第 19 章）。第 17 章介绍计算机系统体系结构，并描述了作为基础的计算机系统对于数据库系统的影响。在这一章中讨论了集中式系统、客户－服务器系统、并行和分布式体系结构。

  在第 18 章关于并行数据库的讨论中，我们探讨了各种并行技术，包括 I/O 并行、查询间并行和查询内并行，以及操作间并行和操作内并行。这一章中还讨论了并行系统设计。

  第 19 章讨论分布式数据库系统。在分布式数据库系统的环境下重新讨论数据库设计、事务管理、查询执行和优化问题。这一章还包括了故障时的系统可用性问题，并介绍了异构分布式数据库、基于云的数据库和分布式目录系统。
- **第六部分：数据仓库、数据挖掘与信息检索**（第 20 章和第 21 章）。第 20 章介绍数据仓库和数据挖掘的概念。第 21 章描述用于查询文本数据的信息检索技术，包括在 Web 搜索引擎中使用的基于超链接的技术。

  第六部分使用了第一部分和第二部分的模型和语言概念，但并不依赖于第三部分、第四部分或第五部分。因此它可以很容易地结合到侧重于 SQL 和数据库设计的课程中。
- **第七部分：特种数据库**（第 22 章和第 23 章）。第 22 章介绍基于对象的数据库。该章讲述了对象－关系数据模型，该模型扩展了关系数据模型以支持复杂数据类型、类型继承和对象标识。该章还描述了用面向对象的编程语言来访问数据库。

  第 23 章介绍数据表示的 XML 标准，它正日益广泛地应用于复杂数据交换和存储。这一章

还描述了 XML 的查询语言。

- **第八部分：高级主题**（第 24 章至第 26 章）。第 24 章讨论应用开发中的高级话题，包括性能调整、性能评测标准、数据库应用测试和标准化。

  第 25 章介绍空间和地理数据、时间数据、多媒体数据以及移动和个人数据库管理一列问题。

  最后，第 26 章讨论高级事务处理。这一章的内容包括事务处理监控器、事务工作流、电子商务、高性能事务系统、实时事务系统和持续长时间的事务。

- **第九部分：实例研究**（第 27 章至第 30 章）。在这一部分我们对四个领先的数据库系统进行实例研究，包括 PostgreSQL、Oracle、IBM DB2 和 Microsoft SQL Server。这几章中列举了上述每一种系统的独有特性，描述了它们的内部结构，提供了关于各个产品的丰富的有用信息，帮助读者了解前面各部分描述的各种实现技术是如何使用到实际系统中的。这几章中还包括实际系统设计中的几个有趣的方面。

- **第十部分：附录**（附录 A - 附录 E）。我们提供 5 个附录，包括一些历史性的和高级的内容。这些附录只在本书的 Web 站点（http://www.db-book.com）中联机提供。只有附录 A（详细的大学模式）例外，它给出了我们的大学模式的细节，包括完整的模式、DDL 和所有的表。这个附录也出现在纸质版本中。

  附录 B（高级关系数据库设计）描述了高级关系数据库设计，包括多值依赖理论、连接依赖、按影连接和域 - 码范式。这个附录是为希望更详细地研究关系数据库设计理论的读者，以及希望在课程中这样做的教师准备的。这个附录同样只是联机提供，就在本书的网站上。

  附录 C（其他关系查询语言）描述其他的关系查询语言，包括 QBE Microsoft Access 和 Datalog。

  虽然大多数新的数据库应用系统使用关系模型或对象 - 关系模型，但网状的和层次的数据模型在一些遗留应用中也仍然在使用。为了满足希望了解这些数据模型的读者的需要，我们给出了描述网状和层次的数据模型的附录，分别为附录 D（网状模型）和附录 E（层次模型）。

第 6 版

对于本书第 6 版的产生起指导作用的包括我们收到的关于前面几版的许多意见和建议，我们在耶鲁大学、利哈伊大学、孟买印度理工学院讲授本课程的体会，以及我们对于数据库技术发展方向的分析。

我们用一个大学的例子替换了原先的银行企业的运行例子。这个例子与学生相关，不仅有助于他们记住这个例子，而且更重要的是，使他们能够更深地理解所需进行的各种设计决策。

我们重新组织了这本书，将我们关于 SQL 的介绍都集中在一起，放在本书的开头部分。第 3 章、第 4 章和第 5 章对 SQL 进行完整的介绍。第 3 章给出该语言的基础，而将更高级的特性放在了第 4 章。在第 5 章中，我们介绍从通用的程序设计语言访问 SQL 的 JDBC 和其他方法。我们介绍触发器和视图，最后讨论联机分析处理（OLAP）。入门性的课程可以选择只包括第 5 章的某些节，或者将一些开始还到讲过了数据库设计之后再讲，这样也不失连续性。

除了这两个主要的改变外，我们修订了每一章中的素材，对旧的素材进行更新，增加对于数据库技术的新的发展的讨论，并且改进学生难于理解的那些内容的描述。我们还增加了新的练习并更新了参考文献。具体的改变如下：

- **较早介绍了 SQL。**许多讲课教师使用 SQL 作为课程实习的核心成分（访问我们的网站 www.db-

book.com 以获取示例课程实习）。为了给学生充足的时间做课程实习，尤其对于那些四学期制的大学和学院来说，尽早教授 SQL 非常重要。有了这个认识，我们在内容组织方面做了几个改动：

- 简短的一章（第 2 章）介绍关系模型，放在 SQL 之前，打下概念基础，而不使学生迷失在关系代数的细节中。
- 第 3 章、第 4 章和第 5 章详细介绍 SQL。这几章涵盖了绝大部分数据库系统支持的变体。从而学生在实际的数据库系统上执行查询时会少遇到一些问题。这几章涵盖了 SQL 的所有方面，包括查询、数据定义、约束类型、OLAP 和从各种语言（包括 Java/JDBC）内部使用 SQL。
- 形式语言（第 6 章）推迟到 SQL 之后，而且可以整章不讲，而不会影响其他章的顺序。只有在第 13 章关于查询优化的讨论依赖于第 6 章介绍的关系代数。

- **新的数据库模式。**我们采用了一个新的模式作为贯穿全书的运行实例，基于大学的数据。对于学生来说，这个模式比以前的银行模式更加直观和有意义，而且在数据库设计的章节中展示了更复杂的设计权衡。
- **对学生的亲手实践提供了更多的支持。**为了推动学生跟着我们的运行实例进行实践，我们在附录 A 中列出了大学数据库的数据库模式和样例的关系实例，并在用到大学实例的各个章中进行了说明。另外，在我们的 Web 站点 http://www.db-book.com 上，提供了对于整个例子的 SQL 数据定义语句，以及建立我们的样例关系实例的 SQL 语句。这鼓励了学生直接在一个实际的数据库系统中去运行举例查询，并且试着去修改这些查询。
- **修订了对 E-R 模型的涵盖。**我们修改了第 7 章中 E-R 图的符号表示，以使得它与 UML 更兼容。这一章还充分利用了新的大学数据库模式来描述更复杂的设计权衡。
- **修订了对关系设计的涵盖。**第 8 章的风格现在更具可读性，它在描述函数依赖理论之非专家提供了对函数依赖和规范化的一个直观的理解，其结果是理论的动机更清晰了。
- **扩充了关于应用开发和安全性的素材。**第 9 章包括了关于应用开发的新的素材，以反映该领域的快速变化。特别地，扩展了关于安全性的介绍，包括它在当今紧密互关的世界中的重要意义，并重点强调了抽象和概念外的实际问题。
- **修订和更新了对数据存储、索引和查询优化的涵盖。**对第 10 章进行了新技术方面的更新，包括对入存的更多的讨论。

  更新了第 11 章对 B+树的讨论，以反映实际的实现，包括对批量载入的介绍，并且对陈述方式也进行了改进。第 11 章中 B+树的例子修改成了 $n=4$，以避免（不切实际的）$n=3$ 时发生的空结点的特殊情况。

  第 13 章中有关于高级查询值和技术的新的素材。
- **修订了对事务管理的涵盖。**第 14 章全面涵盖了入门级课程中的基础知识，而高级的细节在第 15 章和第 16 章中介绍。我们扩展了第 14 章，以涵盖数据库用户和数据库应用开发人员所面对的事务管理的实际问题。这一章还包括了对第 15 章和第 16 章所讨论问题的扩展的综述，以便即使跳过第 15 章和第 16 章，学生仍能对并发控制和恢复的概念有基本的了解。

  第 14 章和第 15 章现在包括了对当今被广泛支持和使用的类型隔离性的清晰介绍，对于使用它的潜在风险也做了介绍。

  第 16 章现在对基本的基于日志的恢复进行了简化的描述，进一步介绍了 ARIES 算法。

- **修订和扩充了对分布式数据库的涵盖。**我们现在涵盖了云数据存储，它在商业应用中正日益受到极大的重视。云存储为企业提供了改进成本管理和增加存储可扩充性的机遇，特别是对于基于 Web 的应用。我们分析了这些优点，也指出了它们的缺点和风险。

  我们原来在高级事务处理这一章中讨论多数据库，现在对它更具体进行了介绍，作为分布式数据库这一章中的一部分。

- **推迟了对对象数据库和 XML 的介绍。**虽然面向对象的概念和 XML 在数据库的范围之外被广泛使用，但在数据库中的实用仍然有限，这使得它们对于高级读者或作为入门性课程的补充材料更适合一些。因此这些话题在本书中向后移了，放到了第 22 章和第 23 章。

- **QBE、Microsoft Access 和 Datalog 放在联机附录中。**这些话题原来是“其他关系语言”这一章中的一部分，现在放在联机附录 C 中了。

  上面没有列出的所有话题都在第 5 版的基础上做了更新，虽然相对地它们的总体组织没有改变。

回顾材料和习题

每一章都有一个大话回顾表，还有一个总结，这可以帮助读者回顾这一章中涵盖的关键话题。

练习划分成两部分：**实践习题**（practice exercise）和**习题**（exercise）。实践习题的解答在本书的 Web 站点公开发布。我们鼓励学生独立解决这些实践习题，然后用 Web 站点上的解答来检查自己的答案。其他练习题的解答只有讲课教师能得到（参看下面的“讲课教师注意事项”以获取如何得到解答的信息）。

许多章的末尾有一个“工具”节，它提供与该章的话题相关的软件工具的信息。这些工具中的一些可以用于实验室练习。大学数据库和习题中用到的其他关系的 SQL DDL 和样例数据在本书的 Web 站点可以得到，也可以用于实验室练习。

讲课教师注意事项

本书包括基本内容和高级内容，在一个学期内也许不能讲授所有这些内容。我们以符号“*”把某些节标记为高级内容，如果愿意的话可以省略这些节，而仍能保持内容的连续性。较难的（可以忽略的）练习同样用符号“*”标记出来了。

可以用本书各章的不同子集来设计课程。某些章也可以用不同于本书中的顺序来讲授。我们列出一些可能的安排如下：

- 第 5 章可以跳过或推迟到后面讲，而不失连续性。我们期望大多数的课程至少 5.1.1 节较早地讲授，因为 JDBC 很可能会是学生实习的一个有用工具。
- 第 6 章可以在第 2 章之后、讲 SQL 之前讲。只可以把这一章从入门性的课程中省略掉。

  我们建议如果课程中讲查询处理的话，那么把 6.1 节也包括在课程中。但是，如果学生在课程中不使用关系演算的话，那么 6.2 节和 6.3 节可以省略。

- 如果教师愿意的话，第 7 章可以在第 3 章、第 4 章和第 5 章之前讲。因为第 7 章完全不依赖于 SQL。
- 第 13 章可以从入门性的课程中省略掉，而对于其他任何章的讲授都没有影响。
- 我们对事务处理的讨论（第 14 章至第 16 章）和对系统体系结构的讨论（第 17 章至第 19 章）都包括一章综述（分别为第 14 章和第 17 章）和后续的两章详细讨论。如果计划把后面几章排

这门高级课程中会讲授，可以选择使用第 14 章和第 17 章，省略第 15 章、第 16 章、第 18 章和第 19 章。

- 第 20 章和第 21 章介绍数据仓库、数据挖掘和信息检索，它们可以用作自学材料或从入门性课程中删去。
- 对于入门性课程来说，可以不讲第 22 章和第 23 章。
- 第 24 章至第 26 章涵盖高级应用开发、空间、时间和移动数据，以及高级事务处理，更适合于高级课程或学生自学。
- 第 27 章至第 30 章的实例研究适合学生自学，或者，这几章可以用作课堂上讲的前面那些章的概念的实例。

基于本书的样例课程提纲可以在本书的 Web 站点找到。

Web 站点和教学补充材料：

本书的 Web 站点的网址是：

http://www.db-book.com

该站点包括以下内容：

- 本书所有各章的幻灯片
- 实践习题的答案
- 五个附录
- 最新勘误表
- 实验素材，包括大学模式和习题中用到的其他关系的 SQL DDL 和样例数据，以及关于建立和使用各种数据库系统和工具的说明书

下列附加材料仅有教师可以获得：

- 包括书中所有习题的答案的教师手册
- 包括额外习题的习题库

关于如何得到教师手册和习题库的进一步信息，请发电子邮件到 customer.service@mcgraw-hill.com，McGraw-Hill 关于本书的网页是

http://www.mhhe.com/silberschatz

与我们联系

我们已尽了最大的努力来避免在本书中出现排版错误、内容失误等。然而，与发布新软件类似，错误在所难免；在本书的 Web 站点中有一个最新勘误表。如果你能指出尚未包含在最新勘误表中的本书的疏漏之处，我们将十分感激。

我们很高兴能收到你对改进本书的建议，我们也很欢迎你对本书 Web 站点做出对其他读者有用的贡献，例如程序设计练习、课程实习建议、联机实验室和指南以及讲课要点等。

请将电子邮件发到 db-book-authors@cs.yale.edu，其他来信请寄到 Avi Silberschatz, Department of Computer Science, Yale University, 51 Prospect Street, P. O. Box 208285, New Haven, CT 06520-8285 USA。

致谢

许多人对于第 6 版以及第 6 版所源自的前 5 版提供了帮助。

第 6 版

- Anastassia Ailamaki, Sailesh Krishnamurthy, Spiros Papadimitriou 和 Bianca Schroeder（Carnegie Mellon University），他们撰写了第 27 章，描述 PostgreSQL 数据库系统。
- Hakan Jakobsson（Oracle），他撰写了第 28 章，描述 Oracle 数据库系统。
- Sriram Padmanabhan（IBM），他撰写了第 29 章，描述 IBM DB2 数据库系统。
- Sameet Agarwal, José A. Blakeley, Thierry D'Hers, Gerald Hinson, Dirk Myers, Vaqar Pirzada, Bill Ramos, Balaji Rathakrishnan, Michael Rys, Florian Waas 和 Michael Zwilling（全部来自 Microsoft），他们撰写了第 30 章，描述 Microsoft SQL Server 数据库系统。特别地，José Blakeley 对这一章进行了协调和编辑；César Galindo-Legaria, Goetz Graefe, Kalen Delaney 和 Thomas Casey（全部来自 Microsoft）对于前一版中的 Microsoft SQL Server 这一章做出了贡献。
- Daniel Abadi 审阅了第 5 版的目录表，并协助进行了新的组织。
- Steve Dolins（University of Florida），Rolando Fernanez（George Washington University），Frantisek Franek（McMaster University），Latifur Khan（University of Texas-Dallas），Sanjay Madria（University of Missouri-Rolla），Aris Ouksel（University of Illinois）和 Richard Snodgrass（University of Waterloo），他们作为本书的审阅人，他们的意见对于我们形成第 6 版帮助很大。
- Judi Paige，她帮助形成了插图和演示幻灯片。
- Mark Wogahn，他保证了生成本书文本的软件正常工作，包括 LaTeX 宏和字体。
- N. L. Sarda，他的反馈帮助我们改进了好几章，特别是第 11 章；Vikram Pudi，他促使我们替换掉原来的转行模式；Shetal Shah，他对几个章节给出了反馈。
- 耶鲁大学和印度孟买理工学院的学生，他们对第 5 版，以及第 6 版的预印本给出了意见。

前面各版

- Chen Li, Sharad Mehrotra，他们为第 5 版提供了关于 JDBC 和安全性的素材。
- Marilyn Turnamian 和 Nandprasad Joshi 为第 5 版担任秘书工作。Marilyn 还为第 5 版准备了一个封面设计的早期版本。
- Lyn Dupré 对第 3 版进行了审查，Sara Strandtman 对第 3 版进行了文字编辑。
- Nilesh Dalvi, Sumit Sanghai, Gaurav Bhalotia, Arvind Hulgeri K. V. Raghavan, Prateek Kapadia, Sara Strandtman, Greg Speegle 和 Dawn Bezviner 协助准备了前面各版的教师素材。
- 用船作为封面概念的一部分的想法最初是 Bruce Stephan 向我们建议的。
- 以下人士指出了第 5 版中的错误之处：Alex Coman, Ravindra Guravannavar, Arvind Hulgeri, Rohit Kulshreshtha, Sang-Won Lee, Joe H. C. Lu, Alex N. Napitupulu, H. K. Park, Jian Pei, Fernando Saenz Perez, Donnie Pinkston, Yma Pinto, Rajarshi Rakshit, Sandeep Satpal, Amon Seagull, Barry Soroka, Praveen Ranjan Srivastava, Hans Svensson, Moritz Wiese 和 Eyob Delele Yirdaw。
- 以下人士对本书第 5 版和前面各版提出了建议和意见：B. B. Abhyankar, Hani Abu-Salem, Jamal R. Alsabbagh, Raj Ashar, Don Batory, Phil Bernhard, Christian Breimann, Gavin M. Bierman, Janek Bogucki, Haran Boral, Paul Bourgeois, Phil Bohannon, Robert Brazile, Yuri Breitbart, Ramzi Bualuan, Michael Carey, Soumen Chakrabarti, Tom Chappell, Zhengxin Chen, Y. C. Chin, Jan Chomicki, Laurens Damen, Prasanna Dhandapani, Qin Ding, Valentin Dinu, J. Edwards, Christos Faloutsos, Homma Farian, Alan Fekete, Frantisek Franek, Shashi Gadia, Hector Garcia-Molina, Goetz Graefe, Jim Gray, Le Gruenwald, Eitan M. Gurari, William Hankley, Bruce Hillyer, Ron Hitchens,

Chad Hogg, Arvind Hulgeri, Yannis Ioannidis, Zheng Jisping, Randy M. Kaplan, Graham J. L. Kemp, Rami Khouri, Hyoung-Joo Kim, Won Kim, Henry Korth (Henry F. 的父亲), Carol Kroll, Hae Choon Lee, Sang-Won Lee, Irwin Levinstein, Mark Llewellyn, Gary Lindstrom, Ling Liu, Dave Maier, Keith Marzullo, Marty Maskarinec, Fletcher Mattox, Sharad Mehrotra, Jim Melton, Alberto Mendelzon, Ami Motro, Bhagirath Narahari, Yiu-Kai Dennis Ng, Thanh-Duy Nguyen, Anil Nigam, Cyril Orji, Meral Ozsoyoglu, D. B. Phatak, Juan Altmayer Pizzorno, Bruce Porter, Sunil Prabhakar, Jim Peterson, K. V. Raghavan, Nahid Rahman, Rajarshi Rakshit, Krithi Ramamritham, Mike Reiter, Greg Riccardi, Odinaldo Rodriguez, Mark Roth, Marek Rusinkiewicz, Michael Rys, Sunita Sarawagi, N. L. Sarda, Patrick Schmid, Nikhil Sethi, S. Seshadri, Stewart Shen, Shashi Shekhar, Amit Sheth, Max Smolens, Nandit Soparkar, Greg Speegle, Jeff Storey, Dilys Thomas, Prem Thomas, Tim Wahls, Anita Whitehall, Christopher Wilson, Marianne Winslett, Weining Zhang 和 Liu Zhenming。

本书的制作出版

出版人是 Raghu Srinivasan，策划编辑是 Melinda D. Bilecki，项目经理是 Melissa Leick，市场经理是 Curt Reynolds，作品监督人是 Laura Fuller，设计者是 Brenda Rolwes，封面设计是 Studio Montage, St. Louis, Missouri，文字编辑是 George Watson，校对是 Kevin Campbell，特约索引制作者是 Tobiah Waldron，Aptara team 主 Raman Arora 和 Sudeshna Nandy 组成。

个人注记

Sudarshan 感谢他的妻子 Sita 的爱和支持，感谢他的孩子 Madhur 和 Advaith 的爱和乐观精神。Hank 感谢他的妻子 Joan，孩子 Abby 和 Joe 的爱和理解。Avi 感谢 Valerie 在本书修订期间的爱，耐心和支持。

A. S.

H. F. K.

S. S.

出版者的话
译者序
前 言

第1章 引言 …… 1
1.1 数据库系统的应用 …… 1
1.2 数据库系统的目标 …… 2
1.3 数据视图 …… 4
1.3.1 数据抽象 …… 4
1.3.2 实例和模式 …… 5
1.3.3 数据模型 …… 5
1.4 数据库语言 …… 6
1.4.1 数据操纵语言 …… 6
1.4.2 数据定义语言 …… 6
1.5 关系数据库 …… 7
1.5.1 表 …… 7
1.5.2 数据操纵语言 …… 8
1.5.3 数据定义语言 …… 8
1.5.4 来自应用程序的数据库访问 …… 8
1.6 数据库设计 …… 9
1.6.1 设计过程 …… 9
1.6.2 大学机构的数据库设计 …… 9
1.6.3 实体-联系模型 …… 10
1.6.4 规范化 …… 11
1.7 数据存储和查询 …… 11
1.7.1 存储管理器 …… 12
1.7.2 查询处理器 …… 12
1.8 事务管理 …… 12
1.9 数据库体系结构 …… 13
1.10 数据挖掘与信息检索 …… 15
1.11 特种数据库 …… 15
1.11.1 基于对象的数据模型 …… 15
1.11.2 半结构化数据模型 …… 16
1.12 数据库用户和管理员 …… 16
1.12.1 数据库用户和用户界面 …… 16
1.12.2 数据库管理员 …… 16
1.13 数据库系统的历史 …… 17
1.14 总结 …… 18
术语回顾 …… 19
实践习题 …… 19
习题 …… 19
工具 …… 20
文献注解 …… 20

第一部分 关系数据库

第2章 关系模型介绍 …… 22
2.1 关系数据库的结构 …… 22
2.2 数据库模式 …… 23
2.3 码 …… 24
2.4 模式图 …… 25
2.5 关系查询语言 …… 26
2.6 关系运算 …… 27
2.7 总结 …… 28
术语回顾 …… 29
实践习题 …… 29
习题 …… 30
文献注解 …… 30

第3章 SQL …… 31
3.1 SQL 查询语言概览 …… 31

3.2 SQL 数据定义 …… 32
3.2.1 基本类型 …… 32
3.2.2 基本模式定义 …… 32
3.3 SQL 查询的基本结构 …… 34
3.3.1 单关系查询 …… 34
3.3.2 多关系查询 …… 35
3.3.3 自然连接 …… 38
3.4 附加的基本运算 …… 40
3.4.1 更名运算 …… 40
3.4.2 字符串运算 …… 41
3.4.3 select 子句中的属性说明 …… 42
3.4.4 排列元组的显示次序 …… 42
3.4.5 where 子句谓词 …… 42
3.5 集合运算 …… 43
3.5.1 并运算 …… 43
3.5.2 交运算 …… 44
3.5.3 差运算 …… 44
3.6 空值 …… 45
3.7 聚集函数 …… 46
3.7.1 基本聚集 …… 46
3.7.2 分组聚集 …… 46
3.7.3 having 子句 …… 47
3.7.4 对空值和布尔值的聚集 …… 48
3.8 嵌套子查询 …… 48
3.8.1 集合成员资格 …… 49
3.8.2 集合的比较 …… 49
3.8.3 空关系测试 …… 50
3.8.4 重复元组存在性测试 …… 51
3.8.5 from 子句中的子查询 …… 52
3.8.6 with 子句 …… 53
3.8.7 标量子查询 …… 53
3.9 数据库的修改 …… 54
3.9.1 删除 …… 54
3.9.2 插入 …… 55
3.9.3 更新 …… 56
3.10 总结 …… 57
术语回顾 …… 58
实践习题 …… 58
习题 …… 60
工具 …… 61
文献注释 …… 62

第 4 章 中级 SQL …… 63

4.1 连接表达式 …… 63
4.1.1 连接条件 …… 63
4.1.2 外连接 …… 64
4.1.3 连接类型和条件 …… 67
4.2 视图 …… 67
4.2.1 视图定义 …… 68
4.2.2 SQL 查询中使用视图 …… 68
4.2.3 物化视图 …… 69
4.2.4 视图更新 …… 69
4.3 事务 …… 71
4.4 完整性约束 …… 71
4.4.1 单个关系上的约束 …… 72
4.4.2 not null 约束 …… 72
4.4.3 unique 约束 …… 72
4.4.4 check 子句 …… 72
4.4.5 参照完整性 …… 73
4.4.6 事务中对完整性约束的违反 …… 75
4.4.7 复杂 check 条件与断言 …… 75
4.5 SQL 的数据类型与模式 …… 76
4.5.1 SQL 中的日期和时间类型 …… 76
4.5.2 默认值 …… 77
4.5.3 创建索引 …… 77
4.5.4 大对象类型 …… 77
4.5.5 用户定义的类型 …… 78
4.5.6 create table 的扩展 …… 79
4.5.7 模式、目录与环境 …… 80
4.6 授权 …… 80
4.6.1 权限的授予与收回 …… 81
4.6.2 角色 …… 82
4.6.3 视图的授权 …… 82

4.6.4 模式的授权 …… 83
4.6.5 权限的转移 …… 83
4.6.6 权限的收回 …… 84
4.7 总结 …… 85
术语回顾 …… 85
实践习题 …… 85
习题 …… 87
文献注解 …… 87

第5章 高级SQL …… 88

5.1 使用程序设计语言访问数据库 …… 88
5.1.1 JDBC …… 89
5.1.2 ODBC …… 93
5.1.3 嵌入式SQL …… 95
5.2 函数和过程 …… 97
5.2.1 声明和调用SQL函数和过程 …… 98
5.2.2 支持过程和函数的语言构造 …… 99
5.2.3 外部语言过程 …… 101
5.3 触发器 …… 102
5.3.1 对触发器的需求 …… 102
5.3.2 SQL中的触发器 …… 102
5.3.3 何时不用触发器 …… 105
5.4 递归查询** …… 106
5.4.1 用迭代来计算传递闭包 …… 106
5.4.2 SQL中的递归 …… 107
5.5 高级聚集特性** …… 109
5.5.1 排名 …… 109
5.5.2 分窗 …… 110
5.6 OLAP** …… 112
5.6.1 联机分析处理 …… 112
5.6.2 交叉表与关系表 …… 114
5.6.3 SQL中的OLAP …… 116
5.7 总结 …… 118
术语回顾 …… 118
实践习题 …… 119
习题 …… 120
工具 …… 121
文献注解 …… 122

第6章 形式化关系查询语言 …… 123

6.1 关系代数 …… 123
6.1.1 基本运算 …… 123
6.1.2 关系代数的形式化定义 …… 128
6.1.3 附加的关系代数运算 …… 129
6.1.4 扩展的关系代数运算 …… 132
6.2 元组关系演算 …… 135
6.2.1 查询示例 …… 135
6.2.2 形式化定义 …… 137
6.2.3 表达式的安全性 …… 137
6.2.4 语言的表达能力 …… 138
6.3 域关系演算 …… 138
6.3.1 形式化定义 …… 138
6.3.2 查询的例子 …… 138
6.3.3 表达式的安全性 …… 139
6.3.4 语言的表达能力 …… 140
6.4 总结 …… 140
术语回顾 …… 140
实践习题 …… 140
习题 …… 142
文献注解 …… 143

第二部分 数据库设计

第7章 数据库设计和E-R模型 …… 146

7.1 设计过程概览 …… 146
7.1.1 设计阶段 …… 146
7.1.2 设计选择 …… 147
7.2 实体-联系模型 …… 148
7.2.1 实体集 …… 148
7.2.2 联系集 …… 148
7.2.3 属性 …… 150

7.3 约束 …… 151
7.3.1 映射基数 …… 152
7.3.2 参与约束 …… 152
7.3.3 码 …… 153
7.4 从实体集中删除冗余属性 …… 153
7.5 实体-联系图 …… 155
7.5.1 基本结构 …… 155
7.5.2 映射基数 …… 156
7.5.3 复杂的属性 …… 156
7.5.4 角色 …… 157
7.5.5 非二元的联系集 …… 157
7.5.6 弱实体集 …… 157
7.5.7 大学的 E-R 图 …… 159
7.6 转换为关系模式 …… 159
7.6.1 具有简单属性的强实体集的表示 …… 160
7.6.2 具有复杂属性的强实体集的表示 …… 160
7.6.3 弱实体集的表示 …… 161
7.6.4 联系集的表示 …… 161
7.7 实体-联系设计问题 …… 163
7.7.1 用实体集还是用属性 …… 163
7.7.2 用实体集还是用联系集 …… 164
7.7.3 二元还是 n 元联系集 …… 165
7.7.4 联系属性的布局 …… 165
7.8 扩展的 E-R 特性 …… 166
7.8.1 特化 …… 166
7.8.2 概化 …… 167
7.8.3 属性继承 …… 168
7.8.4 概化上的约束 …… 168
7.8.5 聚集 …… 169
7.8.6 转换为关系模式 …… 170
7.9 数据建模的其他表示法 …… 171
7.9.1 E-R 图的其他表示法 …… 171
7.9.2 统一建模语言 UML …… 173
7.10 数据库设计的其他方面 …… 174
7.10.1 数据约束和关系数据库设计 …… 174
7.10.2 使用需求：查询、性能 …… 175
7.10.3 授权需求 …… 175
7.10.4 数据流、工作流 …… 175
7.10.5 数据库设计的其他问题 …… 176
7.11 总结 …… 176
术语回顾 …… 177
实践习题 …… 177
习题 …… 179
工具 …… 180
文献注解 …… 180

第 8 章 关系数据库设计 …… 181

8.1 好的关系设计的特点 …… 181
8.1.1 设计选择：更大的模式 …… 181
8.1.2 设计选择：更小的模式 …… 182
8.2 原子域和第一范式 …… 183
8.3 使用函数依赖进行分解 …… 184
8.3.1 码和函数依赖 …… 185
8.3.2 Boyce-Codd 范式 …… 186
8.3.3 BCNF 和保持依赖 …… 187
8.3.4 第三范式 …… 188
8.3.5 更高的范式 …… 189
8.4 函数依赖理论 …… 189
8.4.1 函数依赖集的闭包 …… 189
8.4.2 属性集的闭包 …… 191
8.4.3 正则覆盖 …… 191
8.4.4 无损分解 …… 193
8.4.5 保持依赖 …… 194
8.5 分解算法 …… 195
8.5.1 BCNF 分解 …… 195
8.5.2 3NF 分解 …… 197
8.5.3 3NF 算法的正确性 …… 198
8.5.4 BCNF 和 3NF 的比较 …… 198
8.6 使用多值依赖的分解 …… 199

8.6.1 多值依赖 …… 199
8.6.2 第四范式 …… 201
8.6.3 4NF 分解 …… 201
8.7 更多的范式 …… 202
8.8 数据库设计过程 …… 202
8.8.1 E-R 模型和规范化 …… 202
8.8.2 属性和联系的命名 …… 203
8.8.3 为了性能去规范化 …… 203
8.8.4 其他设计问题 …… 204
8.9 时态数据建模 …… 204
8.10 总结 …… 206
术语回顾 …… 206
实践习题 …… 206
习题 …… 208
文献注释 …… 209

第 9 章 应用设计和开发 …… 210

9.1 应用程序和用户界面 …… 210
9.2 Web 基础 …… 211
9.2.1 统一资源定位符 …… 211
9.2.2 超文本标记语言 …… 211
9.2.3 Web 服务器和会话 …… 213
9.3 servlet 和 JSP …… 214
9.3.1 一个 servlet 的例子 …… 215
9.3.2 servlet 会话 …… 216
9.3.3 servlet 的生命周期 …… 216
9.3.4 servlet 支持 …… 216
9.3.5 服务器端脚本 …… 217
9.3.6 客户端脚本 …… 218
9.4 应用架构 …… 219
9.4.1 业务逻辑层 …… 220
9.4.2 数据访问层和对象－关系映射 …… 220
9.4.3 Web 服务 …… 222
9.4.4 断连操作 …… 222
9.5 快速应用开发 …… 222
9.5.1 构建用户界面的工具 …… 223
9.5.2 Web 应用框架 …… 224
9.5.3 报表生成器 …… 224
9.6 应用程序性能 …… 225
9.6.1 利用缓存减少开销 …… 225
9.6.2 并行处理 …… 226
9.7 应用程序安全性 …… 226
9.7.1 SQL 注入 …… 226
9.7.2 跨站点脚本和请求伪造 …… 227
9.7.3 密码泄露 …… 227
9.7.4 应用程序认证 …… 228
9.7.5 应用级授权 …… 229
9.7.6 审计追踪 …… 230
9.7.7 修补 …… 230
9.8 加密及其应用 …… 231
9.8.1 加密技术 …… 231
9.8.2 数据库中的加密支持 …… 233
9.8.3 加密和认证 …… 233
9.9 总结 …… 234
术语回顾 …… 235
实践习题 …… 235
习题 …… 236
项目建议 …… 237
工具 …… 239
文献注解 …… 239

第三部分 数据存储和查询

第 10 章 存储和文件结构 …… 242

10.1 物理存储介质概述 …… 242
10.2 磁盘和快闪存储器 …… 244
10.2.1 磁盘的物理特性 …… 244
10.2.2 磁盘性能的度量 …… 245
10.2.3 磁盘块访问的优化 …… 246
10.2.4 快闪存储 …… 248
10.3 RAID …… 249
10.3.1 通过冗余提高可靠性 …… 249
10.3.2 通过并行提高性能 …… 250

10.3.3 RAID 级别 …… 250
10.3.4 RAID 级别的选择 …… 253
10.3.5 硬件问题 …… 253
10.3.6 其他的 RAID 应用 …… 254
10.4 第三级存储 …… 254
10.4.1 光盘 …… 254
10.4.2 磁带 …… 255
10.5 文件组织 …… 255
10.5.1 定长记录 …… 256
10.5.2 变长记录 …… 257
10.6 文件中记录的组织 …… 259
10.6.1 顺序文件组织 …… 259
10.6.2 多表聚簇文件组织 …… 260
10.7 数据字典存储 …… 261
10.8 数据库缓冲区 …… 262
10.8.1 缓冲区管理器 …… 262
10.8.2 缓冲区替换策略 …… 263
10.9 总结 …… 264
术语回顾 …… 265
实践习题 …… 265
习题 …… 266
文献注释 …… 267

第 11 章 索引与散列 …… 268

11.1 基本概念 …… 268
11.2 顺序索引 …… 269
11.2.1 稠密索引和稀疏索引 …… 269
11.2.2 多级索引 …… 270
11.2.3 索引的更新 …… 272
11.2.4 辅助索引 …… 272
11.2.5 多码上的索引 …… 273
11.3 B⁺树索引文件 …… 274
11.3.1 B⁺树的结构 …… 274
11.3.2 B⁺树的查询 …… 275
11.3.3 B⁺树的更新 …… 277
11.3.4 不唯一的搜索码 …… 281
11.3.5 B⁺树更新的复杂性 …… 282
11.4 B⁺树扩展 …… 283
11.4.1 B⁺树文件组织 …… 283
11.4.2 辅助索引和记录重定位 …… 284
11.4.3 字符串上的索引 …… 284
11.4.4 B⁺树索引的批量加载 …… 285
11.4.5 B 树索引文件 …… 285
11.4.6 闪存 …… 287
11.5 多码访问 …… 287
11.5.1 使用多个单码索引 …… 287
11.5.2 多码索引 …… 287
11.5.3 覆盖索引 …… 288
11.6 静态散列 …… 288
11.6.1 散列函数 …… 289
11.6.2 桶溢出处理 …… 290
11.6.3 散列索引 …… 291
11.7 动态散列 …… 292
11.7.1 数据结构 …… 292
11.7.2 查询和更新 …… 293
11.7.3 静态散列与动态散列比较 …… 297
11.8 顺序索引和散列的比较 …… 297
11.9 位图索引 …… 298
11.9.1 位图索引结构 …… 298
11.9.2 位图操作的高效实现 …… 299
11.9.3 位图和 B⁺树 …… 300
11.10 SQL 中的索引定义 …… 300
11.11 总结 …… 300
术语回顾 …… 301
实践习题 …… 302
习题 …… 303
文献注解 …… 304

第 12 章 查询处理 …… 305

12.1 概述 …… 305
12.2 查询代价的度量 …… 306
12.3 选择运算 …… 307
12.3.1 使用文件扫描和索引的选择 …… 307
12.3.2 涉及比较的选择 …… 309

12.3.3 复杂选择的实现 …… 309
12.4 排序 …… 310
12.4.1 外部排序归并算法 …… 310
12.4.2 外部排序归并的代价分析 …… 311
12.5 连接运算 …… 312
12.5.1 最套循环连接 …… 312
12.5.2 块最套循环连接 …… 313
12.5.3 索引最套循环连接 …… 314
12.5.4 归并连接 …… 314
12.5.5 散列连接 …… 317
12.6 其他运算 …… 320
12.6.1 去除重复 …… 320
12.6.2 投影 …… 320
12.6.3 集合运算 …… 320
12.6.4 外连接 …… 321
12.6.5 聚集 …… 322
12.7 表达式计算 …… 322
12.7.1 物化 …… 322
12.7.2 流水线 …… 323
12.8 总结 …… 325
术语回顾 …… 326
实践习题 …… 326
习题 …… 327
文献注解 …… 328

第 13 章 查询优化 …… 329

13.1 概述 …… 329
13.2 关系表达式的转换 …… 330
13.2.1 等价规则 …… 331
13.2.2 转换的例子 …… 333
13.2.3 连接的次序 …… 334
13.2.4 等价表达式的枚举 …… 335
13.3 表达式结果集统计大小的估计 …… 335
13.3.1 目录信息 …… 335
13.3.2 选择运算结果大小的估计 …… 336
13.3.3 连接运算结果大小的估计 …… 338
13.3.4 其他运算的结果集大小的估计 …… 339
13.3.5 不同取值个数的估计 …… 339
13.4 执行计划选择 …… 340
13.4.1 基于代价的连接顺序选择 …… 340
13.4.2 采用等价规则的基于代价的优化器 …… 342
13.4.3 启发式优化 …… 342
13.4.4 嵌套子查询的优化** …… 344
13.5 物化视图** …… 345
13.5.1 视图维护 …… 346
13.5.2 增量的视图维护 …… 346
13.5.3 查询优化和物化视图 …… 348
13.5.4 物化视图和索引选择 …… 348
13.6 查询优化中的高级话题** …… 348
13.6.1 top-K 优化 …… 348
13.6.2 连接极小化 …… 349
13.6.3 更新的优化 …… 349
13.6.4 多查询优化和共享式扫描 …… 349
13.6.5 参数化查询优化 …… 350
13.7 总结 …… 350
术语回顾 …… 351
实践习题 …… 351
习题 …… 353
文献注解 …… 353

第四部分 事务管理

第 14 章 事务 …… 356

14.1 事务概念 …… 356
14.2 一个简单的事务模型 …… 357
14.3 存储结构 …… 358
14.4 事务原子性和持久性 …… 359
14.5 事务隔离性 …… 360

14.6 可串行化 …… 363
14.7 事务隔离性和原子性 …… 366
14.7.1 可恢复调度 …… 366
14.7.2 无级联调度 …… 366
14.8 事务隔离性级别 …… 366
14.9 隔离性级别的实现 …… 368
14.9.1 锁 …… 368
14.9.2 时间戳 …… 368
14.9.3 多版本和快照隔离 …… 368
14.10 事务的 SQL 语句表示 …… 369
14.11 总结 …… 370
术语回顾 …… 371
实践习题 …… 371
习题 …… 372
文献注解 …… 372

第 15 章 并发控制 …… 373

15.1 基于锁的协议 …… 373
15.1.1 锁 …… 373
15.1.2 锁的授予 …… 375
15.1.3 两阶段封锁协议 …… 376
15.1.4 封锁的实现 …… 377
15.1.5 基于图的协议 …… 378
15.2 死锁处理 …… 379
15.2.1 死锁预防 …… 379
15.2.2 死锁检测与恢复 …… 380
15.3 多粒度 …… 381
15.4 基于时间戳的协议 …… 383
15.4.1 时间戳 …… 383
15.4.2 时间戳排序协议 …… 384
15.4.3 Thomas 写规则 …… 385
15.5 基于有效性检查的协议 …… 386
15.6 多版本机制 …… 387
15.6.1 多版本时间戳排序 …… 387
15.6.2 多版本两阶段封锁 …… 388
15.7 快照隔离 …… 388
15.7.1 更新事务的有效性检验步骤 …… 389
15.7.2 串行化问题 …… 389
15.8 插入操作、删除操作与谓词读 …… 391
15.8.1 删除 …… 391
15.8.2 插入 …… 392
15.8.3 谓词读和幻象现象 …… 392
15.9 实践中的弱一致性级别 …… 394
15.9.1 二级一致性 …… 394
15.9.2 游标稳定性 …… 394
15.9.3 跨越用户交互的并发控制 …… 394
15.10 索引结构中的并发** …… 395
15.11 总结 …… 397
术语回顾 …… 399
实践习题 …… 399
习题 …… 401
文献注解 …… 402

第 16 章 恢复系统 …… 403

16.1 故障分类 …… 403
16.2 存储器 …… 403
16.2.1 稳定存储器的实现 …… 404
16.2.2 数据访问 …… 404
16.3 恢复与原子性 …… 405
16.3.1 日志记录 …… 406
16.3.2 数据库修改 …… 407
16.3.3 并发控制和恢复 …… 407
16.3.4 事务提交 …… 408
16.3.5 使用日志来重做和撤销事务 …… 408
16.3.6 检查点 …… 410
16.4 恢复算法 …… 411
16.4.1 事务回滚 …… 411
16.4.2 系统崩溃后的恢复 …… 411
16.5 缓冲区管理 …… 412
16.5.1 日志记录缓冲 …… 412
16.5.2 数据库缓冲 …… 413
16.5.3 操作系统在缓冲区管理中的作用 …… 414

16.5.4 模糊检查点 …… 414
16.6 非易失性存储器数据丢失的故障 …… 415
16.7 锁的提前释放和逻辑 undo 操作 …… 415
16.7.1 逻辑操作 …… 416
16.7.2 逻辑 undo 日志记录 …… 416
16.7.3 有逻辑 undo 的事务回滚 …… 417
16.7.4 逻辑 undo 中的并发问题 …… 419
16.8 ARIES** …… 419
16.8.1 数据结构 …… 419
16.8.2 恢复算法 …… 421
16.8.3 其他特性 …… 422
16.9 远程备份系统 …… 423
16.10 总结 …… 424
术语回顾 …… 425
实践习题 …… 426
习题 …… 427
文献注解 …… 428

第五部分 系统体系结构

第 17 章 数据库系统体系结构 …… 430
17.1 集中式与客户-服务器体系结构 …… 430
17.1.1 集中式系统 …… 430
17.1.2 客户-服务器系统 …… 431
17.2 服务器系统体系结构 …… 432
17.2.1 事务服务器 …… 432
17.2.2 数据服务器 …… 434
17.2.3 基于云的服务器 …… 435
17.3 并行系统 …… 435
17.3.1 加速比和扩展比 …… 435
17.3.2 互连网络 …… 437
17.3.3 并行数据库体系结构 …… 437
17.4 分布式系统 …… 439
17.4.1 分布式数据库示例 …… 440
17.4.2 实现问题 …… 441
17.5 网络类型 …… 442
17.5.1 局域网 …… 442
17.5.2 广域网 …… 443
17.6 总结 …… 443
术语回顾 …… 444
实践习题 …… 444
习题 …… 445
文献注释 …… 445

第 18 章 并行数据库 …… 447
18.1 引言 …… 447
18.2 I/O 并行 …… 447
18.2.1 划分技术 …… 447
18.2.2 划分技术比较 …… 448
18.2.3 偏斜处理 …… 449
18.3 查询间并行 …… 450
18.4 查询内并行 …… 450
18.5 操作内并行 …… 451
18.5.1 并行排序 …… 451
18.5.2 并行连接 …… 452
18.5.3 其他关系运算 …… 455
18.5.4 运算的并行计算代价 …… 455
18.6 操作间并行 …… 456
18.6.1 流水线并行 …… 456
18.6.2 独立并行 …… 456
18.7 查询优化 …… 456
18.8 并行系统设计 …… 457
18.9 多核处理器的并行性 …… 458
18.9.1 并行性与原始速度 …… 458
18.9.2 高速缓冲存储器和多线程 …… 458
18.9.3 适应现代体系架构的数据库系统设计 …… 459
18.10 总结 …… 459
术语回顾 …… 460
实践习题 …… 460

习题 …… 461
文献注解 …… 462

第 19 章 分布式数据库 …… 463

19.1 同构和异构数据库 …… 463
19.2 分布式数据存储 …… 463
19.2.1 数据复制 …… 464
19.2.2 数据分片 …… 464
19.2.3 透明性 …… 465
19.3 分布式事务 …… 465
19.3.1 系统结构 …… 466
19.3.2 系统故障模式 …… 466
19.4 提交协议 …… 467
19.4.1 两阶段提交 …… 467
19.4.2 三阶段提交 …… 469
19.4.3 事务处理的可选择性模型 …… 469
19.5 分布式数据库中的并发控制 …… 471
19.5.1 封锁协议 …… 471
19.5.2 时间戳 …… 473
19.5.3 弱一致性级别的复制 …… 473
19.5.4 死锁处理 …… 474
19.6 可用性 …… 475
19.6.1 基于多数的方法 …… 476
19.6.2 读一个、写所有可用的方法 …… 477
19.6.3 站点重建 …… 477
19.6.4 与远程备份的比较 …… 477
19.6.5 协调器的选择 …… 477
19.6.6 为可用性而牺牲一致性 …… 478
19.7 分布式查询处理 …… 479
19.7.1 查询转换 …… 479
19.7.2 简单的连接处理 …… 480
19.7.3 半连接策略 …… 480
19.7.4 利用并行性的连接策略 …… 481
19.8 异构分布式数据库 …… 481
19.8.1 数据统一视图 …… 481
19.8.2 查询处理 …… 482
19.8.3 多数据库中的事务管理 …… 482
19.9 基于云的数据库 …… 483
19.9.1 云上的数据存储系统 …… 484
19.9.2 云上的传统数据库 …… 487
19.9.3 基于云的数据库的挑战 …… 488
19.10 目录系统 …… 488
19.10.1 目录访问协议 …… 489
19.10.2 LDAP：轻量级目录访问协议 …… 489
19.11 总结 …… 492
术语回顾 …… 493
实践习题 …… 493
习题 …… 495
文献注解 …… 495

第六部分 数据仓库、数据挖掘与信息检索

第 20 章 数据仓库与数据挖掘 …… 498

20.1 决策支持系统 …… 498
20.2 数据仓库 …… 499
20.2.1 数据仓库成分 …… 499
20.2.2 数据仓库模式 …… 500
20.2.3 面向列的存储 …… 501
20.3 数据挖掘 …… 501
20.4 分类 …… 502
20.4.1 决策树分类器 …… 503
20.4.2 其他类型的分类器 …… 505
20.4.3 回归 …… 507
20.4.4 分类器验证 …… 507
20.5 关联规则 …… 508
20.6 其他类型的关联 …… 509
20.7 聚类 …… 509
20.8 其他类型的数据挖掘 …… 510
20.9 总结 …… 511
术语回顾 …… 511
实践习题 …… 512

习题 …… 512
工具 …… 512
文献注解 …… 513

第 21 章 信息检索 …… 514

21.1 概述 …… 514
21.2 使用术语的相关性排名 …… 515
21.2.1 使用 TF-IDF 的排名方法 …… 515
21.2.2 基于相似性的检索 …… 516
21.3 使用超链接的相关性 …… 517
21.3.1 流行度排名 …… 517
21.3.2 PageRank …… 518
21.3.3 其他的流行度度量 …… 518
21.3.4 搜索引擎作弊 …… 519
21.3.5 将 TF-IDF 和流行度排名度量方法结合 …… 520
21.4 同义词、多义词和本体 …… 520
21.5 文档的索引 …… 521
21.6 检索的有效性度量 …… 522
21.7 Web 的抓取和索引 …… 523
21.8 信息检索：网页排名之外 …… 524
21.8.1 查询结果的多样化 …… 524
21.8.2 信息抽取 …… 524
21.8.3 问答系统 …… 525
21.8.4 查询结构化数据 …… 525
21.9 目录与分类 …… 526
21.10 总结 …… 527
术语回顾 …… 528
实践习题 …… 528
习题 …… 528
工具 …… 529
文献注解 …… 529

第七部分 特种数据库

第 22 章 基于对象的数据库 …… 532

22.1 概述 …… 532
22.2 复杂数据类型 …… 532
22.3 SQL 中的结构类型和继承 …… 534
22.3.1 结构类型 …… 534
22.3.2 类型继承 …… 536
22.4 表继承 …… 537
22.5 SQL 中的数组和多重集合类型 …… 538
22.5.1 创建和访问集合体值 …… 539
22.5.2 查询以集合体为值的属性 …… 539
22.5.3 嵌套和解除嵌套 …… 540
22.6 SQL 中的对象标识和引用类型 …… 541
22.7 O-R 特性的实现 …… 542
22.8 持久化程序设计语言 …… 543
22.8.1 对象的持久化 …… 544
22.8.2 对象标识和指针 …… 544
22.8.3 持久对象的存储和访问 …… 545
22.8.4 持久化 C++ 系统 …… 545
22.8.5 持久化 Java 系统 …… 547
22.9 对象 - 关系映射 …… 548
22.10 面向对象与对象 - 关系 …… 548
22.11 总结 …… 549
术语回顾 …… 549
实践习题 …… 550
习题 …… 551
工具 …… 551
文献注解 …… 552

第 23 章 XML …… 553

23.1 动机 …… 553
23.2 XML 数据结构 …… 555
23.3 XML 文档模式 …… 558
23.3.1 文档类型定义 …… 558
23.3.2 XML Schema …… 560
23.4 查询和转换 …… 563
23.4.1 XML 树模型 …… 563
23.4.2 XPath …… 563
23.4.3 XQuery …… 565
23.5 XML 应用程序接口 …… 568

23.6 XML 数据存储 …… 569
23.6.1 非关系的数据存储 …… 569
23.6.2 关系数据库 …… 570
23.6.3 SQL/XML …… 572
23.7 XML 应用 …… 573
23.7.1 存储复杂结构数据 …… 573
23.7.2 标准化数据交换格式 …… 573
23.7.3 Web 服务 …… 574
23.7.4 数据中介 …… 574
23.8 总结 …… 575
术语回顾 …… 576
实践习题 …… 576
习题 …… 577
工具 …… 578
文献注解 …… 578

第八部分 高级主题

第 24 章 高级应用开发 …… 580

24.1 性能调整 …… 580
24.1.1 提高函数集合的特性 …… 580
24.1.2 批量加载和更新的调整 …… 581
24.1.3 缓冲位置 …… 582
24.1.4 可调参数 …… 583
24.1.5 硬件调整 …… 584
24.1.6 模式调整 …… 585
24.1.7 索引调整 …… 585
24.1.8 使用物化视图 …… 586
24.1.9 物理设计的自动调整 …… 586
24.1.10 并发事务调整 …… 587
24.1.11 性能模拟 …… 589
24.2 性能基准程序 …… 589
24.2.1 任务集 …… 589
24.2.2 数据库应用类型 …… 589
24.2.3 TPC 基准程序 …… 590
24.3 应用系统开发的其他问题 …… 591
24.3.1 应用系统测试 …… 591
24.3.2 应用系统移植 …… 592
24.4 标准化 …… 592
24.4.1 SQL 标准 …… 593
24.4.2 数据库连接标准 …… 593
24.4.3 对象数据库标准 …… 594
24.4.4 基于 XML 的标准 …… 594
24.5 总结 …… 595
术语回顾 …… 595
实践习题 …… 596
习题 …… 596
文献注解 …… 597

第 25 章 时空数据和移动性 …… 598

25.1 动机 …… 598
25.2 数据库中的时间 …… 598
25.2.1 SQL 中的时间规范 …… 599
25.2.2 时态查询语言 …… 599
25.3 空间与地理数据 …… 600
25.3.1 几何信息表示 …… 600
25.3.2 设计数据库 …… 601
25.3.3 地理数据 …… 602
25.3.4 空间查询 …… 603
25.3.5 空间数据的索引 …… 604
25.4 多媒体数据库 …… 607
25.4.1 多媒体数据格式 …… 607
25.4.2 连续媒体数据 …… 607
25.4.3 基于相似性的检索 …… 608
25.5 移动性和个人数据库 …… 608
25.5.1 移动计算模型 …… 609
25.5.2 路由和查询处理 …… 610
25.5.3 广播数据 …… 610
25.5.4 连接断开与一致性 …… 610
25.6 总结 …… 612
术语回顾 …… 612
实践习题 …… 613
习题 …… 613
文献注解 …… 613

第 26 章　高级事务处理 …… 615

26.1　事务处理监控器 …… 615
26.1.1　TP 监控器体系结构 …… 615
26.1.2　使用 TP 监控器进行应用协调 …… 617
26.2　事务工作流 …… 618
26.2.1　工作流说明 …… 619
26.2.2　工作流的故障原子性需求 … 620
26.2.3　工作流执行 …… 620
26.2.4　工作流恢复 …… 621
26.2.5　工作流管理系统 …… 621
26.3　电子商务 …… 622
26.3.1　电子目录 …… 622
26.3.2　市场 …… 622
26.3.3　订单结算 …… 623
26.4　主存数据库 …… 623
26.5　实时事务系统 …… 625
26.6　长事务 …… 625
26.6.1　不可串行化的执行 …… 626
26.6.2　并发控制 …… 627
26.6.3　嵌套事务和多级事务 …… 627
26.6.4　补偿事务 …… 628
26.6.5　实现问题 …… 628
26.7　总结 …… 629
术语回顾 …… 629
实践习题 …… 630
习题 …… 630
文献注解 …… 631

第九部分　实例研究

第 27 章　PostgreSQL …… 634

27.1　概述 …… 634
27.2　用户界面 …… 634
27.2.1　交互式终端界面 …… 634
27.2.2　图形界面 …… 635
27.2.3　编程语言接口 …… 636
27.3　SQL 变化和扩展 …… 636
27.3.1　PostgreSQL 类型 …… 637
27.3.2　规则和其他主动数据库特征 …… 638
27.3.3　可扩展性 …… 639
27.4　PostgreSQL 中的事务管理 …… 642
27.4.1　PostgreSQL 的并发控制 …… 642
27.4.2　恢复 …… 647
27.5　存储和索引 …… 647
27.5.1　表 …… 648
27.5.2　索引 …… 648
27.6　查询处理和优化 …… 650
27.6.1　查询重写 …… 650
27.6.2　查询规划和优化 …… 650
27.6.3　查询执行器 …… 651
27.6.4　触发器和约束 …… 651
27.7　系统结构 …… 652
文献注解 …… 652

第 28 章　Oracle …… 654

28.1　数据库设计和查询工具 …… 654
28.1.1　数据库和应用设计工具 …… 654
28.1.2　查询工具 …… 654
28.2　SQL 的变化和扩展 …… 655
28.2.1　对象 - 关系特性 …… 655
28.2.2　Oracle XML DB …… 655
28.2.3　过程化语言 …… 656
28.2.4　维度 …… 656
28.2.5　联机分析处理 …… 656
28.2.6　触发器 …… 656
28.3　存储和索引 …… 656
28.3.1　表空间 …… 657
28.3.2　段 …… 657
28.3.3　表 …… 657
28.3.4　索引 …… 659
28.3.5　位图索引 …… 659
28.3.6　基于函数的索引 …… 660

28.3.7 连接索引 …… 660
28.3.8 域索引 …… 660
28.3.9 划分 …… 660
28.3.10 物化视图 …… 661
28.4 查询处理和优化 …… 662
28.4.1 执行方法 …… 662
28.4.2 优化 …… 663
28.4.3 并行执行 …… 665
28.4.4 结果高速缓存 …… 666
28.5 并发控制与恢复 …… 666
28.5.1 并发控制 …… 666
28.5.2 恢复的基本结构 …… 667
28.5.3 Oracle 数据卫士 …… 668
28.6 系统体系结构 …… 668
28.6.1 专用服务器：内存结构 …… 668
28.6.2 专用服务器：进程结构 …… 669
28.6.3 共享服务器 …… 669
28.6.4 Oracle Real Application Clusters …… 669
28.6.5 自动存储管理器 …… 670
28.6.6 Oracle Exadata …… 670
28.7 复制、分布以及外部数据 …… 671
28.7.1 复制 …… 671
28.7.2 分布式数据库 …… 671
28.7.3 外部数据源 …… 671
28.8 数据库管理工具 …… 672
28.8.1 Oracle 企业管理器 …… 672
28.8.2 自动工作负载存储 …… 672
28.8.3 数据库资源管理 …… 672
28.9 数据挖掘 …… 672
文献注解 …… 673

第 29 章 IBM DB2 Universal Database …… 674

29.1 概述 …… 674
29.2 数据库设计工具 …… 675
29.3 SQL 的变化和扩展 …… 675
29.3.1 XML 特性 …… 675
29.3.2 数据类型的支持 …… 676
29.3.3 用户自定义函数和方法 …… 676
29.3.4 大对象 …… 677
29.3.5 索引扩展和约束 …… 677
29.3.6 Web 服务 …… 677
29.3.7 其他特性 …… 678
29.4 存储和索引 …… 678
29.4.1 存储体系结构 …… 678
29.4.2 缓冲池 …… 679
29.4.3 表、记录和索引 …… 679
29.5 多维聚簇 …… 680
29.5.1 块索引 …… 681
29.5.2 块映射 …… 682
29.5.3 设计考虑 …… 682
29.5.4 对现有技术的影响 …… 682
29.6 查询处理和优化 …… 682
29.6.1 存取方法 …… 684
29.6.2 连接、聚集和集合运算 …… 684
29.6.3 对复杂 SQL 处理的支持 …… 684
29.6.4 多处理器查询处理特性 …… 684
29.6.5 查询优化 …… 685
29.7 物化的查询表 …… 685
29.7.1 查询路由到 MQT …… 686
29.7.2 MQT 的维护 …… 686
29.8 DB2 中的自治特性 …… 686
29.8.1 配置 …… 687
29.8.2 优化 …… 687
29.9 工具和实用程序 …… 687
29.10 并发控制和恢复 …… 688
29.10.1 并发与隔离 …… 688
29.10.2 提交与回滚 …… 689
29.10.3 日志与恢复 …… 689
29.11 系统体系结构 …… 690
29.12 复制、分布和外部数据 …… 691
29.13 商务智能特性 …… 691
文献注释 …… 692

第 30 章 Microsoft SQL Server …… 693

30.1 管理、设计和查询工具 …… 693

30.1.1 数据库开发和可视化数据库工具 …… 693

30.1.2 数据库查询和调优工具 …… 693

30.1.3 SQL Server Management Studio …… 695

30.2 SQL 变化和扩展 …… 696

30.2.1 数据类型 …… 696

30.2.2 查询语言增强 …… 697

30.2.3 例程 …… 697

30.2.4 带过滤的索引 …… 698

30.3 存储和索引 …… 699

30.3.1 文件组 …… 699

30.3.2 文件组内的空间管理 …… 699

30.3.3 表 …… 699

30.3.4 索引 …… 699

30.3.5 分区 …… 700

30.3.6 在线创建索引 …… 700

30.3.7 扫描和预读 …… 700

30.3.8 压缩 …… 700

30.4 查询处理和优化 …… 700

30.4.1 编译处理概述 …… 700

30.4.2 查询简化 …… 701

30.4.3 重排序和基于代价的优化 …… 701

30.4.4 更新计划 …… 702

30.4.5 优化时的数据分析 …… 702

30.4.6 部分搜索和启发式搜索 …… 702

30.4.7 查询执行 …… 703

30.5 并发与恢复 …… 703

30.5.1 事务 …… 703

30.5.2 封锁 …… 704

30.5.3 恢复和可用性 …… 705

30.6 系统体系结构 …… 706

30.6.1 服务器上的线程池 …… 706

30.6.2 内存管理 …… 706

30.6.3 安全性 …… 707

30.7 数据访问 …… 707

30.8 分布式异构查询处理 …… 708

30.9 复制 …… 709

30.9.1 复制模型 …… 709

30.9.2 复制选项 …… 709

30.10 .NET 中的服务器编程 …… 710

30.10.1 .NET 基本概念 …… 710

30.10.2 SQL CLR 宿主 …… 711

30.10.3 可扩展性协定 …… 711

30.11 XML 支持 …… 712

30.11.1 本地存储和组织 XML …… 713

30.11.2 查询和更新 XML 数据类型 …… 713

30.11.3 XQuery 表达式的执行 …… 714

30.12 SQL Server 服务代理 …… 714

30.13 商务智能 …… 716

30.13.1 SQL Server 集成服务 …… 716

30.13.2 SQL Server 分析服务 …… 717

30.13.3 SQL Server 报表服务 …… 718

文献注解 …… 718

第十部分 附录

附录 A 详细的大学模式 …… 722

参考文献 …… 729

索引 …… 754

第1章
Database System Concepts, 6E

引　言

数据库管理系统(DataBase-Management System, DBMS)由一个互相关联的数据的集合和一组用以访问这些数据的程序组成。这个数据集合通常称作**数据库**(database)，其中包含了关于某个企业的信息。DBMS 的主要目标是要提供一种可以方便、高效地存取数据库信息的途径。

设计数据库系统的目的是为了管理大量信息。对数据的管理既涉及信息存储结构的定义，又涉及信息操作机制的提供。此外，数据库系统还必须提供所存储信息的安全性保证，即使在系统崩溃或有人企图越权访问时也应保障信息的安全性。如果数据将被多用户共享，那么系统还必须设法避免可能产生的异常结果。

在大多数组织中信息是非常重要的，因而计算机科学家开发了大量的用于有效管理数据的概念和技术。这些概念和技术正是本书所关注的。在这一章里，我们将要介绍数据库系统的基本原理。

1.1　数据库系统的应用

数据库的应用非常广泛，以下是一些具有代表性的应用：

- 企业信息
  - 销售：用于存储客户、产品和购买信息。
  - 会计：用于存储付款、收据、账户余额、资产和其他会计信息。
  - 人力资源：用于存储雇员、工资、所得税和福利的信息，以及产生工资单。
  - 生产制造：用于管理供应链，跟踪工厂中产品的生产情况、仓库和商店中产品的详细清单以及产品的订单。
  - 联机零售：用于存储以上所述的销售数据，以及实时的订单跟踪，推荐品清单的生成，还有实时的产品评估的维护。
- 银行和金融
  - 银行业：用于存储客户信息、账户、贷款，以及银行的交易记录。
  - 信用卡交易：用于记录信用卡消费的情况和产生每月清单。
  - 金融业：用于存储股票、债券等金融票据的持有、出售和买入的信息；也可用于存储实时的市场数据，以便客户能够进行联机交易，公司能够进行自动交易。
- 大学：用于存储学生信息、课程注册和成绩。(此外，还存储通常的单位信息，例如人力资源和会计信息等。)
- 航空业：用于存储订票和航班的信息。航空业是最先以地理上分布的方式使用数据库的行业之一。

已　此页码为英文原书页码，与索引中的页码一致。——编辑注

- 电信业：用于存储通话记录，产生每月账单，维护预付电话卡的余额和存储通信网络的信息。

正如以上所列举的，数据库已经成为当今几乎所有企业不可缺少的组成部分。它不仅存储大多数企业都有的普通的信息，也存储各类企业特有的信息。

在20世纪最后的40年中，数据库的使用在所有的企业中都有所增长。在早期，很少有人直接和数据库系统打交道，尽管没有意识到这一点，他们还是与数据库间接地打着交道，比如，通过打印的报表（如信用卡的对账单）或者通过代理（如银行的出纳员和机票预订代理等）与数据库打交道。自动取款机的出现，使用户可以直接和数据库进行交互。计算机的电话界面（交互式语音应答系统）也使得用户可以直接和数据库进行交互，呼叫者可以通过拨号和按电话键来输入信息或选择可选项，来找出如航班的起降时间或注册大学的课程等。

20世纪90年代末的互联网革命急剧地增加了用户对数据库的直接访问。很多组织将他们的访问数据库的电话界面改为Web界面，并提供了大量的在线服务和信息。比如，当你访问一家在线书店，浏览一本书或一个音乐集时，其实你正在访问存储在某个数据库中的数据。当你确认了一个网上订购，
2 你的订单也就保存在了某个数据库中。当你访问一个银行网站，检索你的账户余额和交易信息时，这些信息也是从银行的数据库系统中取出来的。当你访问一个网站时，关于你的一些信息可能会从某个数据库中取出，并且选择出那些适合显示给你的广告。此外，关于你访问网络的数据也可能会存储在一个数据库中。

因此，尽管用户界面隐藏了访问数据库的细节，大多数人甚至没有意识到他们正在和一个数据库打交道，然而访问数据库已经成为当今几乎每个人生活中不可缺少的组成部分。

也可以从另一个角度来评判数据库系统的重要性。如今，像Oracle这样的数据库系统厂商是世界上最大的软件公司之一，并且在微软和IBM等这些有多样化产品的公司中，数据库系统也是其产品线的一个重要组成部分。

1.2 数据库系统的目标

数据库系统作为商业数据计算机化管理的早期方法而产生。作为20世纪60年代这类方法的典型实例之一，考虑大学组织中的一个部分，除其他数据外，需要保存关于所有教师、学生、系和开设课程的信息。在计算机中保存这些信息的一种方法是将它们存放在操作系统文件中。为了使用户可以对信息进行操作，系统中应有一些对文件进行操作的应用程序，包括：

- 增加新的学生、教师和课程。
- 为课程注册学生，并产生班级花名册。
- 为学生填写成绩、计算绩点（GPA）、产生成绩单。

这些应用程序是由系统程序员根据大学的需求编写的。

随着需求的增长，新的应用程序被加入到系统中。例如，某大学决定创建一个新的专业（例如，计算机科学），那么这个大学就要建立一个新的系并创建新的永久性文件（或在现有文件中添加信息）来记录关于这个系中所有的教师、这个专业的所有学生、开设的课程、学位条件等信息。进而就有可能需要编写新的应用程序来处理这个新专业的特殊规则。也可能会需要编写新的应用程序来处理大学中的新规则。因此，随着时间的推移，越来越多的文件和应用程序就会加入到系统中。

以上所描述的典型的**文件处理系统**（file-processing system）是传统的操作系统所支持的。永久记录被存储在多个不同的文件中，人们编写不同的应用程序来将记录从有关文件中取出或加入到适当的文件中。在数据库管理系统（DBMS）出现以前，各个组织通常都采用这样的系统来存储信息。

3 在文件处理系统中存储组织信息的主要缺陷包括：

- **数据的冗余和不一致**（data redundancy and inconsistency）。由于文件和程序是在很长的一段时间内由不同的程序员创建的，不同文件可能有不同的结构，不同程序可能采用不同的程序设计语言写成。此外，相同的信息可能在几个地方（文件）重复存储。例如，如果某学生有两个专业（例如，音乐和数学），该学生的地址和电话号码就可能既出现在包含音乐系学生记录的文件中，又出现在包含数学系学生记录的文件中。这种冗余除了导致存储和访问开销增大外，还可

能导致**数据不一致性**(data inconsistency)，即同一数据的不同副本不一致。例如，学生地址的更改可能在音乐系记录中得到反映而在系统的其他地方却没有。

- **数据访问困难**(difficulty in accessing data)。假设大学的某个办事人员需要找出居住在某个特定邮编地区的所有学生的姓名，于是他要求数据处理部门生成这样的一个列表。由于原始系统的设计者并未预料到会有这样的需求，因此没有现成的应用程序去满足这个需求。但是，系统中却有一个产生所有学生列表的应用程序。这时该办事人员有两种选择：一种是取得所有学生的列表并从中手工提取所需信息，另一种是要求数据处理部门让某个程序员编写相应的应用程序。这两种方案显然都不太令人满意。假设编写了相应的程序，几天以后这个办事人员可能又需要将该列表减少到只列出至少选课60学分的那些学生。可以预见，产生这样一个列表的程序又不存在，这个职员就再次面临前面那两种都不尽如人意的选择。

  这里需要指出的是，传统的文件处理环境不支持以一种方便而高效的方式去获取所需数据。我们需要开发通用的、能对变化的需求做出更快反应的数据检索系统。

- **数据孤立**(data isolation)。由于数据分散在不同文件中，这些文件又可能具有不同的格式，因此编写新的应用程序来检索适当数据是很困难的。

- **完整性问题**(integrity problem)。数据库中所存储数据的值必须满足某些特定的一致性约束(consistency constraint)。假设大学为每个系维护一个账户，并且记录各个账户的余额。我们还假设大学要求每个系的账户余额永远不能低于零。开发者通过在各种不同应用程序中加入适当的代码来强制系统中的这些约束。然而，当新的约束加入时，很难通过修改程序来体现这些新的约束。尤其是当约束涉及不同文件中的多个数据项时，问题就变得更加复杂了。

- **原子性问题**(atomicity problem)。如同任何别的设备一样，计算机系统也会发生故障。一旦故障发生，数据就应被恢复到故障发生以前的一致的状态。对很多应用来说，这样的保证是至关重要的。让我们看看把A系的账户余额中的500美元转入B系的账户余额中的这样一个程序。假设在程序的执行过程中发生了系统故障，很可能A系的余额中减去的500美元还没来得及存入B系的余额中，这就造成了数据库状态的不一致。显然，为了保证数据库的一致性，这里的借和贷两个操作必须是要么都发生，要么都不发生。也就是说，转账这个操作必须是原子的——它要么全部发生要么根本不发生。在传统的文件处理系统中，保持原子性是很难做到的。

- **并发访问异常**(concurrent-access anomaly)。为了提高系统的总体性能以及加快响应速度，许多系统允许多个用户同时更新数据。实际上，如今最大的互联网零售商每天就可能有来自购买者对其数据的数百万次访问。在这样的环境中，并发的更新操作可能相互影响，有可能导致数据的不一致。设A系的账户余额中有10 000美元。假如系里的两个职员几乎同时从系的账户中取款(例如分别取出500美元和100美元)。这样的并发执行就可能使账户处于一种错误的(或不一致的)状态。假设每个取款操作对应执行的程序是读取原始账户余额，在其上减去取款的金额，然后将结果写回。如果两次取款的程序并发执行，可能它们读到的余额都是10 000美元，并将分别写回9500美元和9900美元。A系的账户余额中到底剩下9500美元还是9900美元视哪个程序后写回结果而定，而实际上正确的值应该是9400美元。为了消除这种情况发生的可能性，系统必须进行某种形式的管理。但是，由于数据可能被多个不同的应用程序访问，这些程序相互间事先又没有协调，管理就很难进行。

  作为另一个例子，假设为确保注册一门课程的学生人数不超过上限，注册程序维护一个注册了某门课程的学生计数。当一个学生注册时，该程序读入这门课程的当前计数值，核实该计数还没有达到上限，给计数值加1，将计数存回数据库。假设两个学生同时注册，而此时的计数值是(例如)39。尽管两个学生都成功地注册了这门课程，计数值应该是41，然而两个程序执行可能都读到值39，然后都写回值40，导致不正确地只增加了1个注册人数。此外，假设该课程注册人数的上限是40；在上面的例子中，两个学生都成功注册，就导致违反了40个学生为注册上限的规定。

- **安全性问题**(security problem)。并非数据库系统的所有用户都可以访问所有数据。例如在大学中，

工资发放人员只需要看到数据库中关于财务信息的那个部分。他们不需要访问关于学术记录的信息。
5 但是，由于应用程序总是即席地加入到文件处理系统中来，这样的安全性约束难以实现。

以上问题以及一些其他问题，促进了数据库系统的发展。接下来我们将看一看数据库系统为了解决上述在文件处理系统中存在的问题而提出的概念和算法。在本书的大部分篇幅中，我们在讨论典型的数据处理应用时总以大学作为实例。

1.3 数据视图

数据库系统是一些互相关联的数据以及一组使得用户可以访问和修改这些数据的程序的集合。数据库系统的一个主要目的是给用户提供数据的抽象视图，也就是说，系统隐藏关于数据存储和维护的某些细节。

1.3.1 数据抽象

一个可用的系统必须能高效地检索数据。这种高效性的需求促使设计者在数据库中使用复杂的数据结构来表示数据。由于许多数据库系统的用户并未受过计算机专业训练，系统开发人员通过如下几个层次上的抽象来对用户屏蔽复杂性，以简化用户与系统的交互：

- **物理层**(physical level)。最低层次的抽象，描述数据实际上是怎样存储的。物理层详细描述复杂的底层数据结构。
- **逻辑层**(logical level)。比物理层层次稍高的抽象，描述数据库中存储什么数据及这些数据间存在什么关系。这样逻辑层就通过少量相对简单的结构描述了整个数据库。虽然逻辑层的简单结构的实现可能涉及复杂的物理层结构，但逻辑层的用户不必知道这样的复杂性。这称作**物理数据独立性**(physical data independence)。数据库管理员使用抽象的逻辑层，他必须确定数据库中应该保存哪些信息。
- **视图层**(view level)。最高层次的抽象，只描述整个数据库的某个部分。尽管在逻辑层使用了比较简单的结构，但由于一个大型数据库中所存信息的多样性，仍存在一定程度的复杂性。数据库系统的很多用户并不需要关心所有的信息，而只需要访问数据库的一部分。视图层抽象的定义正是为了使这样的用户与系统的交互更简单。系统可以为同一数据库提供多个视图。



图 1-1 数据抽象的三个层次

这三层抽象的相互关系如图 1-1 所示。

通过与程序设计语言中数据类型的概念进行类比，我们可以弄清各层抽象间的区别。大多数高级程序设计语言支持结构化类型的概念。例如，我们可以定义如下记录[⊖]：

```
type instructor = record
        ID: char(5);
        name: char(20);
        dept_name: char(20);
        salary: numeric(8, 2);
        end;
```

以上代码定义了一个具有四个字段的新记录 *instructor*。每个字段有一个字段名和所属类型。对一个大学来说，可能包括几个这样的记录类型：

- *department*，包含字段 *dept_name*、*building* 和 *budget*。
- *course*，包含字段 *course_id*、*title*、*dept_name* 和 *credits*。

⊖ 实际的类型说明依赖于所使用的语言。C 和 C++ 使用 **struct** 说明，Java 没有这样的说明，而可以定义一个简单的类来起到同样的作用。

- *student*，包含字段 *ID*、*name*、*dept_name* 和 *tot_cred*。

在物理层，一个 *instructor*、*department* 或 *student* 记录可能被描述为连续存储位置组成的存储块。编译器为程序设计人员屏蔽了这一层的细节。与此类似，数据库系统为数据库程序设计人员屏蔽了许多最底层的存储细节。而数据库管理员可能需要了解数据物理组织的某些细节。

在逻辑层，每个这样的记录通过类型定义进行描述，正如前面的代码段所示。在逻辑层上同时还要定义这些记录类型的相互关系。程序设计人员正是在这个抽象层次上使用某种程序设计语言进行工作。与此类似，数据库管理员常常在这个抽象层次上工作。

最后，在视图层，计算机用户看见的是为其屏蔽了数据类型细节的一些应用程序。与此类似，视图层上定义了数据库的多个视图，数据库用户看到的是这些视图。除了屏蔽数据库的逻辑层细节以外，视图还提供了防止用户访问数据库的某些部分的安全性机制。例如，大学注册办公室的职员只能看见数据库中关于学生的那部分信息，而不能访问涉及教师工资的信息。

1.3.2 实例和模式

随着时间的推移，信息会被插入或删除，数据库也就发生了改变。特定时刻存储在数据库中的信息的集合称作数据库的一个**实例**(instance)。而数据库的总体设计称作**数据库模式**(schema)。数据库模式即使发生变化，也不频繁。

数据库模式和实例的概念可以通过与用程序设计语言写出的程序进行类比来理解。数据库模式对应于程序设计语言中的变量声明(以及与之关联的类型的定义)。每个变量在特定的时刻会有特定的值。程序中变量在某一时刻的值对应于数据库模式的一个实例。

根据前面我们所讨论的不同的抽象层次，数据库系统可以分为几种不同的模式。**物理模式**(physical schema)在物理层描述数据库的设计，而**逻辑模式**(logical schema)则在逻辑层描述数据库的设计。数据库在视图层也可以有几种模式，有时称为**子模式**(subschema)，它描述了数据库的不同视图。

在这些模式中，因为程序员使用逻辑模式来构造数据库应用程序，从其对应用程序的效果来看，逻辑模式是目前最重要的一种模式。物理模式隐藏在逻辑模式下，并且通常可以在应用程序丝毫不受影响的情况下被轻易地更改。应用程序如果不依赖于物理模式，它们就被称为是具有**物理数据独立性**(physical data independence)，因此即使物理模式改变了它们也无需重写。

在介绍完下一节数据模型的概念后，我们将学习描述模式的语言。

1.3.3 数据模型

数据库结构的基础是**数据模型**(data model)。数据模型是一个描述数据、数据联系、数据语义以及一致性约束的概念工具的集合。数据模型提供了一种描述物理层、逻辑层以及视图层数据库设计的方式。

下文中，我们将提到几种不同的数据模型。数据模型可被划分为四类：

- **关系模型**(relational model)。关系模型用表的集合来表示数据和数据间的联系。每个表有多个列，每列有唯一的列名。关系模型是基于记录的模型的一种。基于记录的模型的名称的由来是因为数据库是由若干种固定格式的记录来构成的。每个表包含某种特定类型的记录。每个记录类型定义了固定数目的字段(或属性)。表的列对应于记录类型的属性。关系数据模型是使用最广泛的数据模型，当今大量的数据库系统都基于这种关系模型。第2－8章将详细介绍关系模型。
- **实体－联系模型**(entity-relationship model)。实体－联系(E-R)数据模型基于对现实世界的这样一种认识：现实世界由一组称作实体的基本对象以及这些对象间的联系构成。实体是现实世界中可区别于其他对象的一件"事情"或一个"物体"。实体－联系模型被广泛用于数据库设计。第7章将详细探讨该模型。
- **基于对象的数据模型**(object-based data model)。面向对象的程序设计(特别是Java、C++或C#)已经成为占主导地位的软件开发方法。这导致面向对象数据模型的发展，面向对象的数据模型可以看成是E-R模型增加了封装、方法(函数)和对象标识等概念后的扩展。对象－关系数据模

型结合了面向对象的数据模型和关系数据模型的特征。第 22 章将讲述对象－关系数据模型。

- **半结构化数据模型**(semistructured data model)。半结构化数据模型允许那些相同类型的数据项含有不同的属性集的数据定义。这和早先提到的数据模型形成了对比：在那些数据模型中所有某种特定类型的数据项必须有相同的属性集。**可扩展标记语言**(eXtensible Markup Language，XML)被广泛地用来表示半结构化数据。这将在第 23 章中详述。

在历史上，**网状数据模型**(network data model)和**层次数据模型**(hierarchical data model)先于关系数据模型出现。这些模型和底层的实现联系很紧密，并且使数据建模复杂化。因此，除了在某些地方仍在使用的旧数据库中之外，如今它们已经很少被使用了。对此感兴趣的读者，在附录 D 和附录 E 中有相关的概述。

1.4 数据库语言

数据库系统提供**数据定义语言**(data-definition language)来定义数据库模式，以及**数据操纵语言**
9 (data-manipulation language)来表达数据库的查询和更新。而实际上，数据定义和数据操纵语言并不是两种分离的语言，相反地，它们简单地构成了单一的数据库语言(如广泛使用的 SQL 语言)的不同部分。

1.4.1 数据操纵语言

数据操纵语言(Data-Manipulation Language，DML)是这样一种语言，它使得用户可以访问或操纵那些按照某种适当的数据模型组织起来的数据。有以下访问类型：

- 对存储在数据库中的信息进行检索。
- 向数据库中插入新的信息。
- 从数据库中删除信息。
- 修改数据库中存储的信息。

通常有两类基本的数据操纵语言：

- **过程化 DML**(procedural DML)要求用户指定需要什么数据以及如何获得这些数据。
- **声明式 DML**(declarative DML)(也称为**非过程化 DML**)只要求用户指定需要什么数据，而不指明如何获得这些数据。

通常声明式 DML 比过程化 DML 易学易用。但是，由于用户不必指明如何获得数据，数据库系统必须找出一种访问数据的高效途径。

查询(query)是要求对信息进行检索的语句。DML 中涉及信息检索的部分称作**查询语言**(query language)。实践中常把查询语言和数据操纵语言作为同义词使用，尽管从技术上来说这并不正确。

目前有很多商业性的或者实验性的数据库查询语言，我们在第 3 章、第 4 章和第 5 章中将学习最广泛使用的查询语言 SQL。我们还会在第 6 章中学习一些其他的查询语言。

我们在 1.3 节讨论的抽象层次不仅可以用于定义或构造数据，而且还可以用于操纵数据。在物理层，我们必须定义可高效访问数据的算法；在更高的抽象层次，我们则强调易用性。目标是使人们能够更有效地和系统交互。数据库系统的查询处理器部件(我们将在第 12 章和第 13 章学习)将 DML 的查询语句翻译成数据库系统物理层的动作序列。

1.4.2 数据定义语言

数据库模式是通过一系列定义来说明的，这些定义由一种称作**数据定义语言**(Data-Definition
10 Language，DDL)的特殊语言来表达。DDL 也可用于定义数据的其他特征。

数据库系统所使用的存储结构和访问方式是通过一系列特殊的 DDL 语句来说明的，这种特殊的 DDL 称作**数据存储和定义**(data storage and definition)语言。这些语句定义了数据库模式的实现细节，而这些细节对用户来说通常是不可见的。

存储在数据库中的数据值必须满足某些**一致性约束**(consistency constraint)。例如，假设大学要求一个系的账户余额必须不能为负值。DDL 语言提供了指定这种约束的工具。每当数据库被更新时，数

据库系统都会检查这些约束。通常，约束可以是关于数据库的任意谓词。然而，如果要测试任意谓词，可能代价比较高。因此，数据库系统实现可以以最小代价测试的完整性约束。

- **域约束**(domain constraint)。每个属性都必须对应于一个所有可能的取值构成的域(例如，整数型，字符型，日期/时间型)。声明一种属性属于某种具体的域就相当于约束它可以取的值。域约束是完整性约束的最基本形式。每当有新数据项插入到数据库中，系统就能方便地进行域约束检测。
- **参照完整性**(referential integrity)。我们常常希望一个关系中给定属性集上的取值也在另一关系的某一属性集的取值中出现(参照完整性)。例如，每门课程所列出的系必须是实际存在的系。更准确地说，一个 *course* 记录中的 *dept_name* 值必须出现在 *department* 关系中的某个记录的 *dept_name* 属性中。数据库的修改会导致参照完整性的破坏。当参照完整性约束被违反时，通常的处理是拒绝执行导致完整性被破坏的操作。
- **断言**(assertion)。一个断言就是数据库需要时刻满足的某一条件。域约束和参照完整性约束是断言的特殊形式。然而，还有许多约束不能仅用这几种特殊形式表达。例如，"每一学期每一个系必须至少开设5门课程"，必须表达成一个断言。断言创建以后，系统会检测其有效性。如果断言有效，则以后只有不破坏断言的数据库更新才被允许。
- **授权**(authorization)。我们也许想对用户加以区别，对于不同的用户在数据库中的不同数据值上允许不同的访问类型。这些区别以授权来表达，最常见的是：**读权限**(read authorization)，允许读取数据，但不能修改数据；**插入权限**(insert authorization)，允许插入新数据，但不允许修改已有数据；**更新权限**(update authorization)，允许修改，但不能删除数据；**删除权限**(delete authorization)，允许删除数据。我们可以赋予用户所有的权限，或者没有或部分拥有这些权限。 11

正如其他任何程序设计语言一样，DDL 以一些指令(语句)作为输入，生成一些输出。DDL 的输出放在**数据字典**(data dictionary)中，数据字典包含了**元数据**(metadata)，元数据是关于数据的数据。可把数据字典看作一种特殊的表，这种表只能由数据库系统本身(不是常规的用户)来访问和修改。在读取和修改实际的数据前，数据库系统先要参考数据字典。

1.5 关系数据库

关系数据库基于关系模型，使用一系列表来表达数据以及这些数据之间的联系。关系数据库也包括 DML 和 DDL。在第 2 章中，我们简单介绍关系模型的基本概念。多数的商用关系数据库系统使用 SQL 语言，该语言将在第3章、第4章和第5章中详细介绍。第6章我们将讨论其他有影响的语言。

1.5.1 表

每个表有多个列，每个列有唯一的名字。图 1-2 展示了一个关系数据库示例，它由两个表组成：其一给出大学教师的细节，其二给出大学中各个系的细节。

第一个表是 *instructor* 表，表示，例如，ID 为 22222 的名叫 Einstein 的教师是物理系的成员，他的年薪为 95 000 美元。第二个表是 *department* 表，表示，例如，生物系在 Watson 大楼，经费预算为 90 000 美元。当然，一个现实世界的大学会有更多的系和教师。在本书中，我们使用小型的表来描述概念。相同模式的更大型的例子可以在联机的版本中得到。

关系模型是基于记录的模型的一个实例。基于记录的模型，之所以有此称谓，是因为数据库的结构是几种固定格式的记录。每个表包含一种特定类型的记录。每种记录类型定

ID	name	dept_name	salary
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

a) *instructor* 表

dept_name	building	budget
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

b) *department* 表

图 1-2 关系数据库的一个实例

义固定数目的字段或属性。表的列对应记录类型的属性。

不难看出，表可以如何存储在文件中。例如，一个特殊的字符（比如逗号）可以用来分隔记录的不同属性，另一特殊的字符（比如换行符）可以用来分隔记录。对于数据库的开发者和用户，关系模型屏蔽了这些低层实现细节。

我们也注意到，在关系模型中，有可能创建一些有问题的模式，比如出现不必要的冗余信息。例如，假设我们把系的 *budget* 存储为 *instructor* 记录的一个属性。那么，每当一个经费预算的值（例如，物
12 理系的经费预算）发生变化时，这个变化必须被反映在与物理系相关联的所有教员记录中。在第 8 章，我们将研究如何区分好的和不好的模式设计。

1.5.2 数据操纵语言

SQL 查询语言是非过程化的。它以几个表作为输入（也可能只有一个），总是仅返回一个表。下面是一个 SQL 查询的例子，它找出历史系的所有教员的名字：

```
select instructor.name
from instructor
where instructor.dept_name = 'History';
```

这个查询指定了从 *instructor* 表中要取回的是 *dept_name* 为 History 的那些行，并且这些行的 *name* 属
13 性要显示出来。更具体点，执行本查询的结果是一个表，它有一列 *name* 和若干行，每一行都是 *dept_name* 为 History 的一个教员的名字。如果这个查询运行在图 1-2 中的表上，那么结果将有两行，一个是名字 El Said，另一个是名字 Califieri。

查询可以涉及来自不止一个表的信息。例如，下面的查询将找出与经费预算超过 95 000 美元的系相关联的所有教员的 ID 和系名。

```
select instructor.ID, department.dept_name
from instructor, department
where instructor.dept_name = department.dept_name and
      department.budget > 95000;
```

如果上述查询运行在图 1-2 中的表上，那么系统将会发现，有两个系的经费预算超过 95 000 美元——计算机科学系和金融系；这些系里有 5 位教员。于是，结果将由一个表组成，这个表有两列（*ID*，*dept_name*）和五行（12121，Finance），（45565，Computer Science），（10101，Computer Science），（83821，Computer Science），（76543，Finance）。

1.5.3 数据定义语言

SQL 提供了一个丰富的 DDL 语言，通过它，我们可以定义表，完整性约束、断言，等等。

例如，以下的 SQL DDL 语句定义了 *department* 表：

```
create table department
    (dept_name    char(20),
     building     char(15),
     budget       numeric(12, 2));
```

上面的 DDL 语句执行的结果就是创建了 *department* 表，该表有 3 个列：*dept_name*、*building* 和 *budget*，每个列有一个与之相关联的数据类型。在第 3 章我们将更详细地讨论数据类型。另外，DDL 语句还更新了数据字典，它包含元数据（见 1.4.2 节）。表的模式就是元数据的一个例子。

1.5.4 来自应用程序的数据库访问

SQL 不像一个通用的图灵机那么强大；即，有一些计算可以用通用的程序设计语言来表达，但无法通过 SQL 来表达。SQL 还不支持诸如从用户那儿输入、输出到显示器，或者通过网络通信这样的动作。这样的计算和动作必须用一种宿主语言来写，比如 C、C++ 或 Java，在其中使用嵌入式的 SQL 查询来访问数据库中的数据。**应用程序**（application program）在这里是指以这种方式与数据库进行交互的
14 程序。在大学系统的例子中，就是那些使学生能够注册课程、产生课程花名册、计算学生的 GPA、产生工资支票等的程序。

为了访问数据库，DML 语句需要由宿主语言来执行。有两种途径可以做到这一点：

- 一种是通过提供应用程序接口（过程集），它可以用来将 DML 和 DDL 的语句发送给数据库，再取回结果。

  与 C 语言一起使用的开放数据库连接（ODBC）标准，是一种常用的应用程序接口标准。Java 数据库连接（JDBC）标准为 Java 语言提供了相应的特性。
- 另一种是通过扩展宿主语言的语法，在宿主语言的程序中嵌入 DML 调用。通常用一个特殊字符作为 DML 调用的开始，并且通过预处理器，称为 **DML 预编译器**（DML precompiler），来将 DML 语句转变成宿主语言中的过程调用。

1.6 数据库设计

数据库系统被设计用来管理大量的信息。这些大量的信息并不是孤立存在的，而是企业行为的一部分；企业的终端产品可以是从数据库中得到的信息，或者是某种设备或服务，数据库对它们起到支持的作用。

数据库设计的主要内容是数据库模式的设计。为设计一个满足企业需求模型的完善的数据库应用环境还要考虑更多的问题。在本书中，我们先着重讨论数据库查询语句的书写以及数据库模式的设计，第 9 章将讨论应用设计的整个过程。

1.6.1 设计过程

高层的数据模型为数据库设计者提供了一个概念框架，去说明数据库用户的数据需求，以及将怎样构造数据库结构以满足这些需求。因此，数据库设计的初始阶段是全面刻画预期的数据库用户的数据需求。为了完成这个任务，数据库设计者有必要和领域专家、数据库用户广泛地交流。这个阶段的成果是制定出用户需求的规格文档。

下一步，设计者选择一个数据模型，并运用该选定的数据模型的概念，将那些需求转换成一个数据库的概念模式。在这个**概念设计**（conceptual-design）阶段开发出来的模式提供了企业的详细概述。设计者再复审这个模式，确保所有的数据需求都满足并且相互之间没有冲突。在检查过程中设计者也可以去掉 [15]
一些冗余的特性。这一阶段的重点是描述数据以及它们之间的联系，而不是指定物理的存储细节。

从关系模型的角度来看，概念设计阶段涉及决定数据库中应该包括哪些属性，以及如何将这些属性组织到多个表中。前者基本上是商业的决策，在本书中我们不进一步讨论。而后者主要是计算机科学的问题，解决这个问题主要有两种方法：一种是使用实体 - 联系模型（见 1.6.3 节），另一种是引入一套算法（通称为规范化），这套算法将所有属性集作为输入，生成一组关系表（见 1.6.4 节）。

一个开发完全的概念模式还将指出企业的功能需求。在**功能需求说明**（specification of functional requirement）中，用户描述数据之上的各种操作（或事务），例如更新数据、检索特定的数据、删除数据等。在概念设计的这个阶段，设计者可以对模式进行复审，确保它满足功能需求。

现在，将抽象数据模型转换到数据库实现进入最后两个设计阶段。在**逻辑设计阶段**（logical-design phrase），设计者将高层的概念模式映射到要使用的数据库系统的实现数据模型上；然后设计者将得到的特定于系统的数据库模式用到**物理设计阶段**（physical-design phrase）中，在这个阶段中指定数据库的物理特性，这些特性包括文件组织的形式以及内部的存储结构。这些内容将在第 10 章中讨论。

1.6.2 大学机构的数据库设计

为了阐明设计过程，我们来看如何为大学做数据库设计。初始的用户需求说明可以基于与数据库用户的交流以及设计者自己对大学机构的分析。这个设计阶段中的需求描述是制定数据库的概念结构的基础。以下是大学的主要特性：

- 大学分成多个系。每个系由自己唯一的名字（*dept_name*）来标识，坐落在特定的建筑物（*building*）中，有它的经费预算（*budget*）。
- 每一个系有一个开设课程列表。每门课程有课程号（*course_id*）、课程名（*title*）、系名（*dept_name*）和学分（*credits*），还可能有先修要求（*prerequisites*）。

- 教师由个人唯一的标识号(*ID*)来标识。每位教师有姓名(*name*)、所在的系(*dept_name*)和工资(*salary*)。
- 学生由个人唯一的标识号(*ID*)来标识。每位学生有姓名(*name*)、主修的系(*dept_name*)和已修学分数(*tot_cred*)。
- 大学维护一个教室列表，详细说明楼名(*building*)、房间号(*room_number*)和容量(*capacity*)。
- 大学维护开设的所有课程(开课)的列表。每次开课由课程号(*course_id*)、开课号(*sec_id*)、年(*year*)和学期(*semester*)来标识。与之相关联的有学期(*semester*)、年(*year*)、楼名(*building*)、房间号(*room_number*)和时段号(*time_slot_id*，即上课的时间)。
- 系有一个教学任务列表，说明每位教师的授课情况。
- 大学有一个所有学生课程注册的列表，说明每位学生在哪些课程的哪次开课中注册了。

一个真正的大学数据库会比上述的设计复杂得多。然而，我们就用这个简化了的模型来帮助你理解概念思想，避免你迷失在复杂设计的细节中。

1.6.3 实体-联系模型

实体-联系(E-R)数据模型使用一组称作实体的基本对象，以及这些对象间的联系。实体是现实世界中可区别于其他对象的一件"事情"或一个"物体"。例如，每个人是一个实体，每个银行账户也是一个实体。

数据库中实体通过**属性**(attribute)集合来描述。例如，属性 *dept_name*、*building* 与 *budget* 可以描述大学中的一个系，并且它们也组成了 *department* 实体集的属性。类似地，属性 *ID*、*name* 和 *salary* 可以描述 *instructor* 实体。[⊖]

我们用额外的属性 *ID* 来唯一标识教师(因为可能存在两位教师有相同的名字和相同的工资)。必须给每位教师分配唯一的教师标识。在美国，许多机构用一个人的社会保障号(它是美国政府分配给每个美国人的一个唯一的号码)作为他的唯一标识。

联系(relationship)是几个实体之间的关联。例如，*member* 联系将一位教师和他所在的系关联在一起。同一类型的所有实体的集合称作**实体集**(entity set)。同一类型的所有联系的集合称作**联系集**(relationship set)。

数据库的总体逻辑结构(模式)可以用实体-联系图(entity-relationship diagram，E-R 图)进行图形化表示。有几种方法来画这样的图。最常用的方法之一是采用**统一建模语言**(Unified Modeling Language，UML)。在我们使用的基于 UML 的符号中，E-R 图如下表示：

- 实体集用矩形框表示，实体名在头部，属性名列在下面。
- 联系集用连接一对相关的实体集的菱形表示，联系名放在菱形内部。

作为例子，我们来看一下大学数据库中包括教师和系以及它们之间的关联的部分。对应的 E-R 图如图 1-3 所示。E-R 图表示出有 *instructor* 和 *department* 这两个实体集，它们具有先前已经列出的一些属性。这个图还指明了在教师和系之间的 *member* 联系。



图 1-3 E-R 图示例

除了实体和联系外，E-R 模型还描绘了数据库必须遵守的对其内容的某些约束。一个重要的约束是**映射基数**(mapping cardinality)，它表示通过某个联系集能与一实体进行关联的实体数目。例如，如果一位教师只能属于一个系，E-R 模型就能表达出这种约束。

⊖ 机敏的读者会注意到我们从描述 *instructor* 实体集的属性集中去掉了 *dept_name* 属性。在第 7 章中我们将详细解释为什么这样做。

实体-联系模型在数据库设计中使用广泛，在第7章中将详细研究。

1.6.4 规范化

设计关系数据库所用到的另外一种方法是通常被称为规范化的过程。它的目标是生成一个关系模式集合，使我们存储信息时没有不必要的冗余，同时又能很轻易地检索数据。这种方法是设计一种符合适当的范式(normal form)的模式，为确定一个关系模式是否符合想要的范式，我们需要额外的关于用数据库建模的现实世界中机构的信息。最常用的方法是使用**函数依赖**(functional dependency)，我们将在8.4节讨论。

为了理解规范化的必要性，我们看一看在不好的数据库设计中会发生什么问题。一个不好的设计可能会包括如下不良特性：

- 信息重复。
- 缺乏表达某些信息的能力。

17
18

我们在对大学例子修改后的数据库设计中讨论这些问题。

假设不将表 *instructor* 和 *department* 分开，我们使用单个表 *faculty*，它将两个表的数据合并在一起(见图1-4)。注意到在表 *faculty* 中有两行包含重复的关于历史系的信息，具体地说，是系所在的大楼和经费预算。这种修改后的设计中出现了我们不想要的信息重复。信息重复浪费存储空间，而且使得更新数据库变得复杂。假设我们想将历史系的经费预算从50 000美元改成46 800美元。这个修改必须在两行中都体现出来；这与原来的设计不同，原来只需要修改一行就可以了。因此，改变后的设计中更新操作的代价比原来的设计中大了。当我们在改变设计的数据库中进行更新时，我们必须保证与历史系有关的每个元组都被更新，否则我们的数据库将会显示出历史系有两个不同的经费预算数。

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

图1-4 *faculty* 表

现在我们再看一看缺乏表达某些信息的能力的问题。假设我们在大学里建立一个新的系。在上述改变了的数据库设计中，我们不能直接表示关于一个系的信息(*dept_name*, *building*, *budget*)，除非那个系在大学里至少有一位教员。这是因为 *faculty* 表中的行需要 *ID*、*name* 和 *salary* 的值。这意味着我们不能记录新建立的系的信息，直到这个新的系聘用了第一位教员。

这个问题的一个解决办法是引入**空值**(null)。空值表示这个值不存在(或者未知)，未知值可能是缺失(该值确实存在，但我们没有得到它)或不知道(我们不知道该值是否存在)。正如我们后面要看到的那样，空值很难处理，所以最好不要用它。如果我们不愿意处理空值，我们可以仅当一个系有至少一位教员时才为它建立系的信息项。而且，当系的最后一位教员离开时，我们必须删除系的信息。很明显，这种情况不是我们想要的，因为在我们原来的数据库设计里，不管一个系有没有教员，这个系的信息都是可以保存的，也不必使用空值。

19

规范化的详尽理论已经研究形成，它有助于形式化地定义什么样的数据库设计是不好的，以及如何得到我们想要的设计。第8章将讨论关系数据库设计，包括规范化。

1.7 数据存储和查询

数据库系统划分为不同的模块，每个模块完成整个系统的一个功能。数据库系统的功能部件大致

可分为存储管理器和查询处理部件。

存储管理非常重要，因为数据库常常需要大量存储空间。企业的大型数据库的大小达到数百个 gigabyte，甚至达到 terabyte。一个 gigabyte 大约等于 1000 个（实际上是 1024 个）megabyte（十亿字节），一个 terabyte 等于一百万个 megabyte（一万亿个字节）。由于计算机主存不可能存储这么多信息，所以信息被存储在磁盘上。需要时数据在主存和磁盘间移动。由于相对于中央处理器的速度来说数据出入磁盘的速度很慢，因此数据库系统对数据的组织必须做到使磁盘和主存之间数据的移动最小化。

查询处理也非常重要，因为它帮助数据库系统简化和方便了数据的访问。查询处理器使得数据库用户能够获得很高的性能，同时可以在视图的层次上工作，不必承受了解系统实现的物理层次细节的负担。将在逻辑层编写的更新和查询转变成物理层的高效操作序列，这是数据库系统的任务。

1.7.1 存储管理器

存储管理器是数据库系统中负责在数据库中存储的低层数据与应用程序以及向系统提交的查询之间提供接口的部件。存储管理器负责与文件管理器进行交互。原始数据通过操作系统提供的文件系统
20 存储在磁盘上。存储管理器将各种 DML 语句翻译为底层文件系统命令。因此，存储管理器负责数据库中数据的存储、检索和更新。

存储管理部件包括：

- **权限及完整性管理器**（authorization and integrity manager），它检测是否满足完整性约束，并检查试图访问数据的用户的权限；
- **事务管理器**（transaction manager），它保证即使发生了故障，数据库已保持在一致的（正确的）状态，并保证并发事务的执行不发生冲突；
- **文件管理器**（file manager），它管理磁盘存储空间的分配，管理用于表示磁盘上所存储信息的数据结构；
- **缓冲区管理器**（buffer manager），它负责将数据从磁盘上取到内存中来，并决定哪些数据应被缓冲存储在内存中。缓冲区管理器是数据库系统中的一个关键部分，因为它使数据库可以处理比内存更大的数据。

存储管理器实现了几种数据结构，作为系统物理实现的一部分：

- **数据文件**（data files），存储数据库自身；
- **数据字典**（data dictionary），存储关于数据库结构的元数据，尤其是数据库模式；
- **索引**（index），提供对数据项的快速访问。和书中的索引一样，数据库索引提供了指向包含特定值的数据的指针。例如，我们可以运用索引找到具有特定的 ID 的 *instructor* 记录，或者具有特定的 name 的所有 *instructor* 记录。散列是另外一种索引方式，在某些情况下速度更快，但不是在所有情况下都这样。

我们在第 10 章讨论存储介质、文件结构和缓冲区管理，第 11 章讨论通过索引和散列高效访问数据的方法。

1.7.2 查询处理器

查询处理器组件包括：

- **DDL 解释器**（DDL interpreter），它解释 DDL 语句并将这些定义记录在数据字典中；
- **DML 编译器**（DML compiler），将查询语言中的 DML 语句翻译为一个执行方案，包括一系列查
21 询执行引擎能理解的低级指令。

一个查询通常可被翻译成多种等价的具有相同结果的执行方案的一种。DML 编译器还进行查询优化（query optimization），也就是从几种选择中选出代价最小的一种。

- **查询执行引擎**（query evaluation engine），执行由 DML 编译器产生的低级指令。

第 12 章将介绍查询执行，查询优化器选择合适的执行策略的方法将在第 13 章中讨论。

1.8 事务管理

通常，对数据库的几个操作合起来形成一个逻辑单元。如 1.2 节所示的例子是一个资金转账，其

中一个系(A系)的账户进行取出操作，而另一个系(B系)的账户进行存入操作。显然，这两个操作必须保证要么都发生要么都不发生。也就是说，资金转账必须完成或根本不发生。这种要么完成要么不发生的要求称为**原子性**(atomicity)。除此以外，资金转账还必须保持数据库的一致性。也就是说，A和B的余额之和应该是保持不变的。这种正确性的要求称作**一致性**(consistency)。最后，当资金转账成功结束后，即使发生系统故障，账户A和账户B的余额也应该保持转账成功结束后的新值。这种保持的要求称作**持久性**(durability)。

事务(transaction)是数据库应用中完成单一逻辑功能的操作集合。每一个事务是一个既具原子性又具一致性的单元。因此，我们要求事务不违反任何的数据库一致性约束。也就是说，如果事务启动时数据库是一致的，那么当这个事务成功结束时数据库也应该是一致的。然而，在事务执行过程中，必要时允许暂时的不一致，因为无论是A取出的操作在前还是B存入的操作在前，这两个操作都必然有一个先后次序。这种暂时的不一致虽然是必需的，但在故障发生时，很可能导致问题的产生。

适当地定义各个事务是程序员的职责，事务的定义应使之能保持数据库的一致性。例如，资金从A系的账户转到B系的账户这个事务可以被定义为由两个单独的程序组成：一个对账户A执行取出操作，另一个对账户B执行存入操作。这两个程序的依次执行可以保持一致性。但是，这两个程序自身都不是把数据库从一个一致的状态转入一个新的一致的状态，因此它们都不是事务。

原子性和持久性的保证是数据库系统自身的职责，确切地说，是**恢复管理器**(recovery manager)的职责。在没有故障发生的情况下，所有事务均成功完成，这时要保证原子性很容易。但是，由于各种 22
各样的故障，事务并不总能成功执行完毕。为了保证原子性，失败的事务必须对数据库状态不产生任何影响。因此，数据库必须被恢复到该失败事务开始执行以前的状态。这种情况下数据库系统必须进行**故障恢复**(failure recovery)，即检测系统故障并将数据库恢复到故障发生以前的状态。

最后，当多个事务同时对数据库进行更新时，即使每个单独的事务都是正确的，数据的一致性也可能被破坏。**并发控制管理器**(concurrency-control manager)控制并发事务间的相互影响，保证数据库一致性。**事务管理器**(transaction manager)包括并发控制管理器和恢复管理器。

事务处理的基本概念在第14章介绍，并发事务的管理在第15章讨论，第16章详细介绍故障恢复。

事务的概念已经广泛应用在数据库系统和应用当中。虽然最初是在金融应用中使用事务，现在事务已经使用在电信业的实时应用中，以及长时间的活动如产品设计和工作流管理中。事务概念更多的应用还会在第26章讨论。

1.9 数据库体系结构

现在我们可以给出一个数据库系统各个部分以及它们之间联系的图了(见图1-5)。

数据库系统的体系结构在很大程度上取决于数据库系统所运行的计算机系统。数据库系统可以是集中式的、客户/服务器式的(一台服务器为多个客户机执行任务)；也可以针对并行计算机体系结构设计数据库系统；分布式数据库包含地理上分离的多台计算机。

在第17章我们将讨论现代计算机系统的一般结构。第18章将描述如何实现数据库的各种动作——尤其是查询处理中的动作——以适应并行处理。第19章将描述分布式数据库中的各种问题，以及如何处理那些问题。问题包括如何存储数据，如何确保在多个节点上执行的事务的原子性，如何执行并发控制，以及遇到故障时如何能提高可用性。除此之外还将讨论分布式查询处理以及目录系统。

今天数据库系统的大多数用户并不直接面对数据库系统，而是通过网络与其相连。因此我们可区分远程数据库用户工作用的**客户机**(client)和运行数据库系统的**服务器**(server)。 23

数据库应用通常可分为两或三个部分，如图1-6所示。在一个**两层体系结构**(two-tier architecture)中，应用程序驻留在客户机上，通过查询语言表达式来调用服务器上的数据库系统功能。像ODBC和JDBC这样的应用程序接口标准被用于进行客户端和服务器的交互。

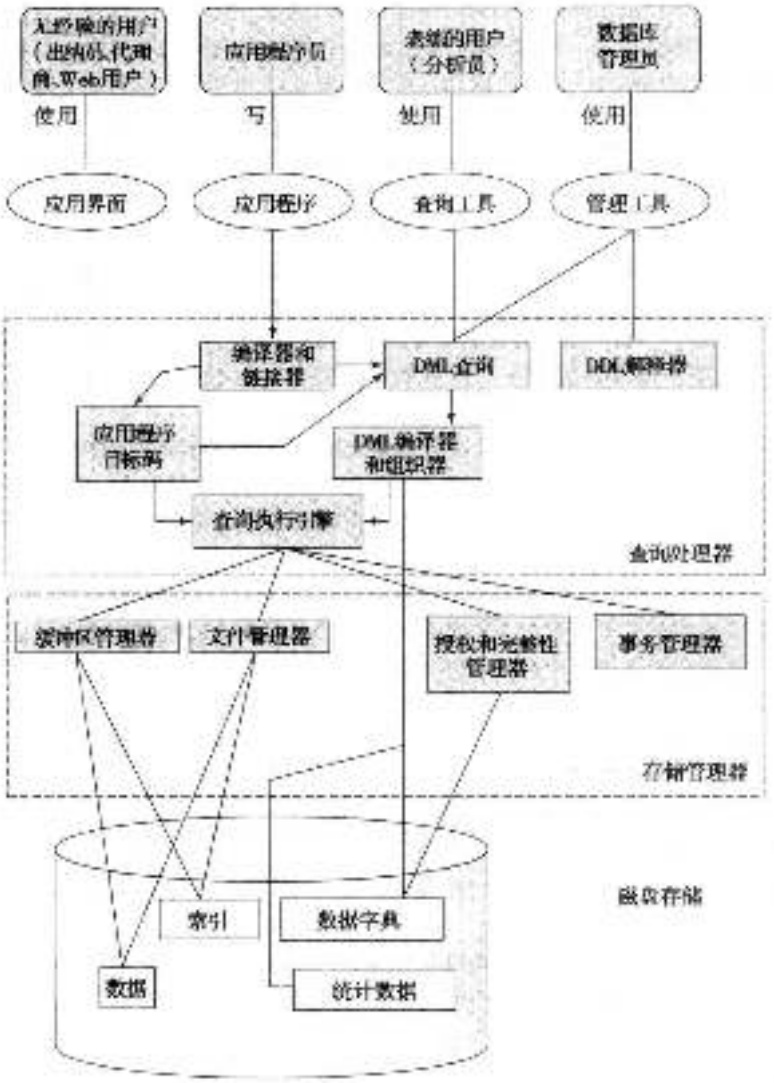


图 1-5 系统体系结构

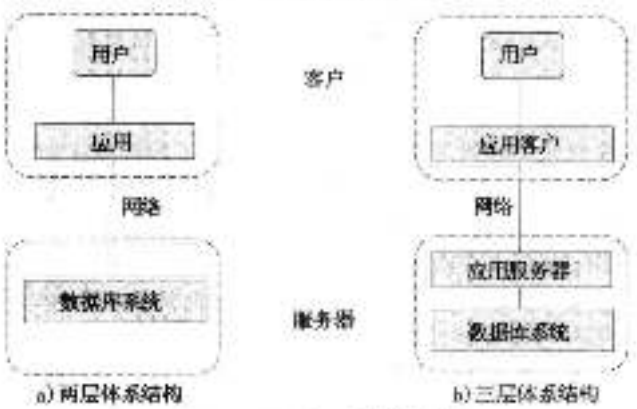


图 1-6 两层和三层体系结构

而在一个**三层体系结构**(three-tier architecture)中，客户机只作为一个前端并且不包含任何直接的

数据库调用。客户端通常通过一个表单界面与**应用服务器**(application server)进行通信。而应用服务器与数据库系统通信以访问数据。应用程序的**业务逻辑**(business logic),也就是说在何种条件下做出何种反应,被嵌入到应用服务器中,而不是分布在多个客户机上。三层结构的应用更适合大型应用和互联网上的应用。

1.10 数据挖掘与信息检索

数据挖掘(data mining)这个术语指半自动地分析大型数据库并从中找出有用的模式的过程。和人工智能中的知识发现(也称为**机器学习**(machine learning))或者统计分析一样,数据挖掘试图从数据中寻找规则或模式。但是,数据挖掘和机器学习、统计分析不一样的地方在于它处理大量的主要存储在磁盘上的数据。也就是说,数据挖掘就是在数据库中发现知识。

从数据库中发现的某些类型的知识可以用一套**规则**(rule)表示。下面是一条规则的例子,非形式化地描述为:"年收入高于50 000美元的年轻女性是最可能购买小型运动车的人群"。当然这条规则并不是永远正确的,但它有一定的"支持度"和"置信度"。其他类型的知识表达方式有联系不同变量的方程式,或者通过其他机制根据某些已知的变量来预测输出。

还有很多其他类型的有用模式以及发现不同模式的技术。在第20章我们将研究一些模式的例子以及如何自动地从数据库中得出这些模式。

通常在数据挖掘中还需要人参与,包括数据预处理使数据变为适合算法的格式,在已发现模式的后处理中找到新奇的有用模式。给定一个数据库,可能有不止一种类型的模式,需要人工交互地选有用类型的模式。由于这个原因,现实中的数据挖掘是一个半自动的过程。但是,在我们的描述中,主要介绍挖掘的自动处理过程的部分。

商业上已经开始利用逐渐发展的联机数据来支持对于业务活动的更好的决策,例如储备哪些物品,如何更好地锁定目标客户以提高销售额。但是,它们的许多查询都相当复杂,有些类型的信息甚至使用SQL都不能抽取出来。

目前有几种技术和工具可用于帮助做决策支持。一些数据分析的工具让分析人员能够从不同的角度观察数据。其他的分析工具提前计算出大量数据的汇总信息,以更快响应查询。现在的SQL标准也增加了支持数据分析的成分。

大型企业有各种不同的可用于业务决策的数据来源。要在这些各种各样的数据上高效地执行查询,企业建立了**数据仓库**(data warehouse)。数据仓库从多个来源收集数据,建立统一的模式,驻留在单个节点上。于是,就为用户提供了单个统一的数据界面。

文本数据也爆炸式增长。文本数据是非结构化的,与关系数据库中严格的结构化数据不同。查询非结构化的文本数据被称为**信息检索**(information retrieval)。信息检索系统和数据库系统很大程度上是相同的——特别是基于辅助存储器的数据存储和检索。但是信息系统领域与数据库系统所强调的重点是不同的。信息系统重点强调基于关键词的查询、文档与查询的相似度、以及文档的分析、分类和索引。第20章和第21章我们将讨论决策支持,包括联机分析处理、数据挖掘、数据仓库和信息检索。

1.11 特种数据库

数据库系统的一些应用领域受到关系数据模型的限制。其结果是,研究人员开发了几种数据模型来处理这些领域的应用,包括基于对象的数据模型和半结构化数据模型。

1.11.1 基于对象的数据模型

面向对象程序设计已经成为占统治地位的软件开发方法学。这导致**面向对象数据模型**(object-based data model)的发展,面向对象模型可以看作E-R模型的扩展,增加了封装、方法(函数)和对象标识。继承、对象标识和信息封装(信息隐藏),以及对外提供方法作为访问对象的接口,这些是面向对象程序设计的关键概念,现在在数据建模中也找到了应用。面向对象数据模型还支持丰富的类型系统,包括结构和集合类型。在20世纪80年代,开发了好几个基于面向对象数据模型的数据库系统。

现在主要的数据库厂商都支持**对象－关系数据模型**(object-relational data model)，这是一个将面向对象数据模型和关系数据模型的特点结合在一起的数据模型。它扩展了传统的关系模型，增加了新的特征如结构和集合类型，以及面向对象特性。第22章介绍对象－关系数据模型。

1.11.2 半结构化数据模型

半结构化数据模型允许那些相同类型的数据项有不同的属性集的数据说明。这和早先提到的数据模型形成了对比：在那些数据模型中所有某种特定类型的数据项必须有相同的属性集。

XML语言设计的初衷是为文本文档增加标签信息，但由于它在数据交换中的应用而变得日益重要。XML提供了表达含有嵌套结构的数据的方法，能够灵活组织数据结构。这对于一些非传统数据来说非常重要。第23章介绍XML语言、XML格式的数据的各种查询表示方法，以及不同XML数据格式之间的转换。

1.12 数据库用户和管理员

数据库系统的一个主要目标是从数据库中检索信息和往数据库中存储新信息。使用数据库的人员可分为数据库用户和数据库管理员。

1.12.1 数据库用户和用户界面

根据所期望的与系统交互方式的不同，数据库系统的用户可以分为四种不同类型。系统为不同类型的用户设计了不同类型的用户界面。

- **无经验的用户**(naïve user)是默认经验的用户，他们通过激活事先已经写好的应用程序同系统进行交互。例如，大学的一位职员需要往A系中添加一位新的教师时，激活一个叫做*new_instructor*的程序。该程序要求这位职员输入新教师的名字，她的新*ID*、系的名字(即A)以及她的工资额。

  此类用户的典型用户界面是表格界面，用户只需填写表格的相应项就可以了。无经验的用户也可以很简单地阅读数据库产生的报表。

  作为另外一个例子，我们考虑一个学生，他在课程注册的过程中想通过Web界面来注册一门课程。应用程序首先验证该用户的身份，然后允许她去访问一个表格，她可以在表格中填入想填的信息。表格信息被送回给服务器上的Web应用程序，然后应用程序确定该课程是否还有空额(通过从数据库中检索信息)，如果有，就把这位学生的信息添加到数据库中的该课程花名册中。

- **应用程序员**(application programmer)是编写应用程序的计算机专业人员。有很多工具可以供应用程序员选择来开发用户界面。**快速应用开发**(Rapid Application Development，RAD)工具是使应用程序员能够尽量少编写程序就可以构造出表格和报表的工具。

- **老练的用户**(sophisticated user)不通过编写程序来同系统交互，而是用数据库查询语言或数据分析软件这样的工具来表达他们的要求。分析员通过提交查询来研究数据库中的数据，所以属于这一类用户。

- **专门的用户**(specialized user)是编写专门的、不适合于传统数据处理框架的数据库应用的富有经验的用户。这样的应用包括：计算机辅助设计系统、知识库和专家系统、存储复杂结构数据(如图形数据和声音数据)的系统，以及环境建模系统。在第22章中我们将要讨论几个这样的应用。

1.12.2 数据库管理员

使用DBMS的一个主要原因是可以对数据和访问这些数据的程序进行集中控制。对系统进行集中控制的人称作**数据库管理员**(DataBase Administrator，DBA)。DBA的作用包括：

- **模式定义**(schema definition)。DBA通过用DDL书写的一系列定义来创建最初的数据库模式。
- **存储结构及存取方法定义**(storage structure and access-method definition)。
- **模式及物理组织的修改**(schema and physical-organization modification)。由数据库管理员(DBA)对

模式和物理组织进行修改，以反映机构的需求变化，或为提高性能选择不同的物理组织。 28

- **数据访问授权**(granting of authorization for data access)。通过授予不同类型的权限，数据库管理员可以规定不同的用户各自可以访问的数据库的部分。授权信息保存在一个特殊的系统结构中，一旦系统中有访问数据的要求，数据库系统就去查阅这些信息。
- **日常维护**(routine maintenance)。数据库管理员的日常维护活动有：
  - 定期备份数据库，或者在磁带上或者在远程服务器上，以防止像洪水之类的灾难发生时数据丢失。
  - 确保正常运转时所需的空余磁盘空间，并且在需要时升级磁盘空间。
  - 监视数据库的运行，并确保数据库的性能不因一些用户类实了花费时间较多的任务就下降很多。

1.13 数据库系统的历史

从商业计算机的出现开始，数据处理就一直推动着计算机的发展。事实上，数据处理自动化早于计算机的出现。Herman Hollerith 发明的穿孔卡片，早在 20 世纪初就用来记录美国的人口普查数据，并且用机械系统来处理这些卡片和列出结果。穿孔卡片后来被广泛用作将数据输入计算机的一种手段。

数据存储和处理技术发展的年表如下：

- **20 世纪 50 年代和 20 世纪 60 年代初**：磁带被用于数据存储。诸如工资单这样的数据处理已经自动化了，数据存储在磁带上。数据处理包括从一个或多个磁带上读取数据，并将数据写回到新的磁带上。数据也可以由一叠穿孔卡片输入，而输出到打印机上。例如，工资增长的处理是通过将增长表示到穿孔卡片上，在读入一叠穿孔卡片时同步地读入保存主要工资细节的磁带。记录必须有相同的排列顺序。工资的增加额将被加入到从主磁带读出的工资中，并被写到新的磁带上，新磁带将成为新的主磁带。

  磁带(和卡片组)都只能顺序读取，数据规模可以比内存大得多。因此，数据处理程序被迫以一种特定的顺序来对数据进行处理，读取和合并来自磁带和卡片组的数据。
- **20 世纪 60 年代末和 20 世纪 70 年代**：20 世纪 60 年代末硬盘的广泛使用极大地改变了数据处理的情况，因为硬盘允许直接对数据进行访问。数据在磁盘上的位置是无关紧要的，因为磁盘上的任何位置都可在几十毫秒内访问到。数据由此摆脱了顺序访问的限制。有了磁盘，我们就可以创建网状和层次的数据库，它可以将表和树这样的数据结构保存在磁盘上。程序员可以构建和操作这些数据结构。 29

  由 Codd[1970] 撰写的一篇具有里程碑意义的论文定义了关系模型和在关系模型中查询数据的非过程化方法，由此关系型数据库诞生了。关系模型的简单性和能够对程序员屏蔽所有实现细节的能力具有真正的诱惑力。最后，Codd 因其所做的工作获得了声誉很高的 ACM 图灵奖。
- **20 世纪 80 年代**：尽管关系模型在学术上很受重视，但是最初并没有实际的应用，这是因为它被认为性能不好；关系型数据库在性能上还不能和当时已有的网状和层次数据库相提并论。这种情况直到 System R 的出现才得以改变，这是 IBM 研究院的一个突破性项目，它开发了能构造高效的关系型数据库系统的技术。Astrahan 等[1976] 和 Chamberlin 等[1981]给出了关于 System R 的很好的综述。完整功能的 System R 原型导致了 IBM 的第一个关系型数据库产品 SQL/DS 的出现。与此同时，加州大学伯克利分校开发了 Ingres 系统。它后来发展成具有相同名字的商品化关系数据库系统。最初的商品化关系型数据库系统，如 IBM DB2、Oracle、Ingres 和 DEC Rdb，在推动高效处理声明性查询的技术上起到了主要的作用。到了 20 世纪 80 年代初期，关系型数据库已经可以在性能上与网状和层次型数据库进行竞争了。关系型数据库是如此简单易用，以至于最后它完全取代了网状/层次型数据库，因为程序员在使用后者时，必须处理许多底层的实现细节，并且不得不将他们要做的查询任务编码成过程化的形式。更重要的，他们在设计应用程序时还要时时考虑效率问题，而这需要付出很大的努力。相反，在关系型数据库中，几乎所有的底层工作都由数据库自动来完成，使得程序员可以只考虑逻辑层的工作。自从在 20 世纪 80 年代取得了统治地位以来，关系模型在数据模型中一直独占鳌头。

在20世纪80年代人们还对并行和分布式数据库进行了很多研究，同样在面向对象数据库方面也有初步的工作。

- **20世纪90年代初**：SQL语言主要是为决策支持应用设计的，这类应用是查询密集的；而20世纪80年代数据库的支柱是事务处理应用，它们是更新密集的。决策支持和查询再次成为数据库的一个主要应用领域。分析大量数据的工具有了很大的发展。

在这个时期许多数据库厂商推出了并行数据库产品。数据库厂商还开始在他们的数据库中
30 加入对象－关系的支持。

- **20世纪90年代**：最重大的事件就是互联网的爆炸式发展。数据库比以前有了更加广泛的应用。现在数据库系统必须支持很高的事务处理速度，而且还要有很高的可靠性和24×7的可用性（一天24小时，一周7天都可用，也就是没有进行维护的停机时间）。数据库系统还必须支持对数据的Web接口。
- **21世纪第一个十年**：21世纪的最初五年中，我们看到了XML的兴起以及与之相关联的XQuery查询语言成为了新的数据库技术。虽然XML广泛应用于数据交换和一些复杂数据类型的存储，但关系数据库仍然构成大多数大型数据库应用系统的核心。在这个时期，我们还见证了"自主计算/自动管理"技术的成长，其目的是减少系统管理开销；我们还看到了开源数据库系统应用的显著增长，特别是PostgreSQL和MySQL。

在21世纪第一个十年的后几年中，用于数据分析的专门的数据库有很大增长，特别是将一个表的每一个列高效地存储为一个单独的数组的列存储，以及为非常大的数据集的分析而设计的高度并行的数据库系统。有几个新颖的分布式数据存储系统被构建出来，以应对非常大的Web节点如Amazon、Facebook、Google、Microsoft和Yahoo!的数据管理需求，并且这些系统中的某些现在可以作为Web服务提供给应用开发人员使用。在管理和分析流数据如股票市场报价数据或计算机网络监测数据方面也有重要的工作。数据挖掘技术现在被广泛部署应用，应用实例包括基于Web的产品推荐系统和Web页面上的相关广告自动布放。

1.14 总结

- **数据库管理系统**（DataBase-Management System，DBMS）由相互关联的数据集合以及一组用于访问这些数据的程序组成。数据描述某特定的企业。
- DBMS的主要目标是为人们提供方便、高效的环境来存储和检索数据。
- 如今数据库系统无所不在，很多人每天直接或间接地与数据库系统打交道。
- 数据库系统设计用来存储大量的信息。数据的管理既包括信息存储结构的定义，也包括提供处理信息的机制。另外数据库系统还必须提供所存储信息的安全性，以处理系统崩溃或者非授权
31 访问企图，如果数据在多个用户之间共享，系统必须避免可能的异常结果。
- 数据库系统的一个主要目的是为用户提供数据的抽象视图，也就是说，系统隐藏数据存储和维护的细节。
- 数据库结构的基础是**数据模型**（data model）：一个用于描述数据、数据之间的联系、数据语义和数据约束的概念工具的集合。
- 关系数据模型是最广泛使用的将数据存储到数据库中的模型。其他的数据模型有面向对象模型、对象－关系模型和半结构化数据模型。
- **数据操纵语言**（Data-Manipulation Language，DML）是使得用户可以访问和操纵数据的语言。当今广泛使用的是非过程化的DML，它只需要用户指明需要什么数据，而不需指明如何获得这些数据。
- **数据定义语言**（Data-Definition Language，DDL）是说明数据库模式和数据的其他特性的语言。
- 数据库设计主要包括数据库模式的设计。实体－联系（E-R）数据模型是广泛用于数据库设计的数据模型，它提供了一种方便的图形化的方式来观察数据、联系和约束。
- 数据库系统由几个子系统构成：

- 存储管理器(storage manager)子系统在数据库中存储的低层数据与应用程序和向系统提交的查询之间提供接口。
- 查询处理器(query processor)子系统编译和执行 DDL 和 DML 语句。

- **事务管理**(transaction management)负责保证不管是否有故障发生，数据库都要处于一致的(正确的)状态。事务管理器还保证并发事务的执行互不冲突。
- 数据库系统的体系结构受支持其运行的计算机系统的影响很大。数据库系统可以是集中式的，或者客户－服务器方式的，即一个服务器机器为多个客户机执行工作。数据库系统还可以设计成具有能充分利用并行计算机系统结构的能力。分布式数据库跨越多个地理上分布的互相分离的计算机。
- 典型地，数据库应用可被分为运行在客户机上的前端和运行在后端的部分。在两层的体系结构中，前端直接和后端运行的数据库进行通信。在三层结构中，后端又被分为应用服务器和数据库服务器。
- 知识发现技术试图自动地从数据中发现统计规律和模式。**数据挖掘**(data mining)领域将人工智能和统计分析研究人员创造的知识发现技术，与使得知识发现技术能够在极大的数据库上高效实现的技术结合起来。
- 有 4 种不同类型的数据库用户，按用户期望与数据库进行交互的不同方式来区分他们。为不同类的用户设计了不同的用户界面。

术语回顾

- 数据库管理系统(DBMS)
- 数据库系统应用
- 文件处理系统
- 数据不一致性
- 一致性约束
- 数据抽象
- 实例
- 模式
  - 物理模式
  - 逻辑模式
- 物理数据独立性
- 数据模型
  - 实体－联系模型
  - 关系数据模型
  - 基于对象的数据模型
  - 半结构化数据模型
- 数据库语言
  - 数据定义语言
  - 数据操纵语言
  - 查询语言
- 元数据
- 应用程序
- 规范化
- 数据字典
- 存储管理器
- 查询处理器
- 事务
  - 原子性
  - 故障恢复
  - 并发控制
- 两层和三层数据库体系结构
- 数据挖掘
- 数据库管理员(DBA)

实践习题

1.1 这一章讲述了数据库系统的几个主要的优点。它有哪两个不足之处？

1.2 列出 Java 或 C++ 之类的语言中的类型说明系统与数据库系统中使用的数据定义语言的 5 个不同之处。

1.3 列出为一个企业建立数据库的六个主要步骤。

1.4 除 1.6.2 节中已经列出的之外，请列出大学要维护的至少 3 种不同类型的信息。

1.5 假设你想要建立一个类似于 YouTube 的视频节点。考虑 1.2 节中列出的将数据保存在文件系统中的各个缺点。讨论每一个缺点与存储实际的视频数据和关于视频的元数据(诸如标题、上传它的用户、标签、观看它的用户)的关联。

1.6 在 Web 查找中使用的关键字查询与数据库查询很不一样。请列出这两者之间在查询表达方式和查询结果处什么方面的主要差异。

习题

1.7 列出四个你使用过的很可能使用了数据库来存储持久数据的应用。

1.8 列出文件处理系统和 DBMS 的两个主要区别。

1.9 解释物理数据独立性的概念，以及它在数据库系统中的重要性。

1.10 列出数据库管理系统的五个职责。对每个职责，说明当它不能被履行时会产生什么样的问题。

1.11 请给出至少两种理由说明为什么数据库系统使用声明性查询语言，如 SQL，而不是只提供 C 或者 C++ 的函数库来执行数据操作。

1.12 解释用图 1-4 中的表来设计会导致哪些问题。

1.13 数据库管理员的五种主要作用是什么？

1.14 解释两层和三层体系结构之间的区别。对 Web 应用来说哪一种更合适？为什么？

1.15 描述可能被用于存储一个社会网络系统如 Facebook 中的信息的至少 3 个表。

工具

如今已有大量的商业数据库系统投入使用，主要的有：IBM DB2（www.ibm.com/software/data/db2）、Oracle（www.oracle.com）、Microsoft SQL Server（www.microsoft.com/sql）、Sybase（www.sybase.com）和 IBM Informix（www.ibm.com/software/data/informix），其中一些对个人或者非商业使用或开发是免费的，但是对实际的部署是不免费的。

34

也有不少免费/公开的数据库系统，使用很广泛的有 MySQL（www.mysql.com）和 PostgreSQL（www.postgresql.org）。

在本书的主页 www.db-book.com 上可以获得更多的厂商网址的链接和其他信息。

文献注解

我们在下面列出了关于数据库的通用书籍、研究论文集和 Web 节点。后续各章提供了本章略述的每个主题的资料参考。

Codd [1970] 的具有里程碑意义的论文引入了关系模型。

关于数据库系统的教科书有 Abiteboul 等 [1995]、O'Neil 和 O'Neil [2000]、Ramakrishnan 和 Gehrke [2002]、Date [2003]、Kifer 等 [2005]、Elmasri 和 Navathe [2006]，以及 Garcia-Molina 等 [2008]。涵盖事务处理的教科书有 Bernstein 和 Newcomer [1997]，以及 Gray 和 Reuter [1993]。有一本书中包含了关于数据库管理的研究论文的汇集，这本书是 Hellerstein 和 Stonebraker [2005]。

Silberschatz 等 [1990]、Silberschatz 等 [1996]、Bernstein 等 [1998]，以及 Abiteboul 等 [2003] 给出了关于数据库管理已有成果和未来研究挑战的综合评述。ACM 的数据管理兴趣组的主页（www.acm.org/sigmod）提供了关于数据库研究的大量信息。数据库厂商的网址（参看上面的工具部分）提供了他们各自产品的细节。

35
|
36

第一部分
Part 1

关系数据库

数据模型是描述数据、数据联系、数据语义以及一致性约束的概念工具的集合。在这一部分中，我们将学习关系模型。

关系模型利用表的集合来表示数据和数据间的联系。第2章将专门介绍关系模型。其概念上的简单性使得它被广泛应用；今天大量的数据库产品都是基于关系模型的。关系模型在逻辑层和视图层描述数据，使用户不必关注数据存储的底层细节。在后面第二部分的第7章中讨论的实体－联系模型是一种更高层的数据模型，被广泛用于数据库设计。

为了让用户可以使用关系数据库中的数据，我们需要解决几个问题。最重要的问题是用户如何说明对数据的检索和更新请求，为此已经开发了好几种查询语言。第二个问题也很重要，就是数据完整性和数据保护。无论用户有意或无意地破坏数据，数据库都要保护数据，使其免遭破坏。

第3章、第4章和第5章讲述当今应用最普遍的一种查询语言——SQL语言。第3章和第4章介绍SQL及其中等程度的应用知识。第4章还将介绍通过数据库触发完整性约束以及授权机制，用来控制用户发出的哪些访问和更新操作是可以执行的。第5章介绍更为深入的主题，包括如何在编程语言中使用SQL，如何利用SQL进行数据分析。

第6章介绍三种形式的查询语言：关系代数、元组关系演算和域关系演算，它们是基于数学逻辑的声明式查询语言。这些形式语言构成了SQL，以及另外两种用户友好的语言QBE和Datalog的基础。（QBE和Datalog的介绍参见附录B，db-book.com上提供在线资料。）

第2章
Database System Concepts, 6E

关系模型介绍

在商用数据处理应用中，关系模型已经成为当今主要的数据模型。之所以占据主要位置，是因为和早期的数据模型如网络模型或层次模型相比，关系模型以其简易性简化了编程者的工作。

本章我们先学习关系模型的基础知识。关系数据库具有坚实的理论基础，我们在第6章学习关系数据库理论中与查询相关的部分，从第7章到第8章我们将考察其中用于关系数据库模式设计的部分，在第12章和第13章我们将讨论高效处理查询的理论。

2.1 关系数据库的结构

关系数据库由表(table)的集合构成，每个表有唯一的名字。例如，图2-1中的*instructor*表记录了有关教师的信息，它有四个列首：*ID*、*name*、*dept_name*和*salary*。该表中每一行记录了一位教师的信息，包括该教师的*ID*、*name*、*dept_name*以及*salary*。类似地，图2-2中的*course*表存放了关于课程的信息，包括每门课程的*course_id*、*title*、*dept_name*和*credits*。注意，每位教师通过*ID*列的取值进行标识，而每门课程则通过*course_id*列的取值来标识。

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

图2-1 *instructor*关系

图2-3给出的第三个表是*prereq*，它存放了每门课程的先修课程信息。该表具有*course_id*和*prereq_id*两列，每一行由一个课程对组成，这个课程对表示了第二门课程是第一门课程的先修课。

由此，*prereq*表中的每行表示了两门课程之间的联系：其中一门课程是另一门课程的先修课。作为另一个例子，我们考察*instructor*表，表中的行可被认为是代表了从一个特定的*ID*到相应的*name*、*dept_name*和*salary*值之间的联系。

一般说来，表中一行代表了一组值之间的一种联系。由于一个表就是这种联系的一个集合，表这个概念和数学上的关系这个概念是密切相关的，这也正是关系数据模型名称的由来。在数学术语中，元组(tuple)只是一组值的序列(或列表)。在n个值之间的一种联系可以在数学上用关于这些值的一个n元组(n-tuple)来表示，换言之，n元组就是一个有n个值的元组，它对应于表中的一行。

course_id	title	dept_name	credits
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

图2-2 *course*关系

这样，在关系模型的术语中，**关系**(relation)用来指代表，而元组(tuple)用来指代一个表记录，属性(attribute)指代的是表中的列。

考察图 2-1，我们可以看出 *instructor* 关系有四个属性：*ID*、*name*、*dept_name* 和 *salary*。

我们用关系实例(relation instance)这个术语来表示一个关系的特定实例，也就是所包含的一组特定的行。图 2-1 所示的 *instructor* 的实例有 12 个元组，对应于 12 个教师。

course_id	prereq_id
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

图 2-3 *prereq* 关系

本章我们将使用多个不同的关系来说明作为关系数据模型基础的各种概念。这些关系代表一个大学的一部分。它们并没有包含真实的大学数据库中的所有数据，这主要是为了简化表示。在第 7 章和第 8 章里我们将详细讨论如何判断适当的关系结构的相关准则。

由于关系是元组集合，所以元组在关系中出现的顺序并无关紧要的。因此，无论关系中的元组是像图 2-1 那样被排序后列出，还是像图 2-4 那样无序的，都没有关系；在上述两图中的关系是一样的，因为它们具有同样的元组集合。为便于说明，当我们在显示关系时，大多数情况下都按其第一个属性排序。

对于关系的每个属性，都存在一个允许取值的集合，称为该属性的**域**(domain)。这样 *instructor* 关系的 *salary* 属性的域就是所有可能的工资值的集合，而 *name* 属性的域是所有可能的教师名字的集合。

ID	name	dept_name	salary
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

图 2-4 *instructor* 关系的无序显示

我们要求对所有关系 r 而言，r 的所有属性的域都是原子的。如果域中元素被看作是不可再分的单元，则域是**原子的**(atomic)。例如，假设 *instructor* 表上有一个属性 *phone_number*，它存放教师的一组联系电话号码。那么 *phone_number* 的域就不是原子的，因为其中的元素是一组电话号码，是可以被再分为单个电话号码这样的子成分的。

重要的问题不在于域本身是什么，而在于我们怎样在数据库中使用域中元素。现在假设 *phone_number* 属性存放单个电话号码。即便如此，如果我们把电话号码的属性值拆分成国家编号、地区编号以及本地号码，那么我们还是把它作为非原子值来对待。如果我们把每个电话号码作为不可再分的单元，那么 *phone_number* 属性才会有原子的域。

在本章，以及第 3 章～第 6 章，我们假设所有属性的域都是原子的。在第 22 章中，我们将讨论对关系数据模型进行扩展以便允许非原子域。

空(null)值是一个特殊的值，表示值未知或不存在。如前所述，如果我们在关系 *instructor* 中包括属性 *phone_number*，则可能某教师根本没有电话号码，或者电话号码未提供。这时我们就只能使用空值来强调该值未知或不存在。以后我们会看到，空值给数据库访问和更新带来很多困难，因此应尽量避免使用空值。我们先假设不存在空值，然后在 3.6 节中我们将描述空值对不同操作的影响。

2.2 数据库模式

当我们谈论数据库时，我们必须区分**数据库模式**(database schema)和**数据库实例**(database instance)，前者是数据库的逻辑设计，后者是给定时刻数据库中数据的一个快照。

关系的概念对应于程序设计语言中变量的概念，而**关系模式**(relation schema)的概念对应于程序设计语言中类型定义的概念。

一般说来，关系模式由属性序列及各属性对应域组成。等第 3 章讨论 SQL 语言时，我们才去关心每个属性的域的精确定义。

关系实例的概念对应于程序设计语言中变量的值的概念。给定变量的值可能随时间发生变化；类似地，当关系被更新时，关系实例的内容也随时间发生了变化。相反，关系的模式是不常变化的。

尽管知道关系模式和关系实例的区别非常重要，我们常常使用同一个名字，比如 *instructor*，既指

代模式，也指代实例。在需要的时候，我们会显示地指明模式或实例。例如"*instructor* 模式"或"*instructor* 关系的一个实例"。然而，在模式或实例的含义清楚的情况下，我们就简单地使用关系的名字。

dept_name	building	budget
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

图 2-5 *department* 关系

考察图 2-5 中的 *department* 关系，该关系的模式是：

department (*dept_name*, *building*, *budget*)

请注意属性 *dept_name* 既出现在 *instructor* 模式中，又出现在 *department* 模式中。这样的重复并不是一种巧合。实际上，在关系模式中使用相同属性正是将不同关系的元组联系起来的一种方法。例如，假设我们希望找出在 Watson 大楼工作的所有教师的相关信息。我们首先在 *department* 关系中找出所有位于 Watson 的系的 *dept_name*。接着，对每一个这样的系，我们在 *instructor* 关系中找出与 *dept_name* 对应的教师信息。

我们继续看大学数据库的例子。

大学里的每门课程可能要讲授多次，可以在不同学期授课，甚至可能在同一个学期授课。我们需要一个关系来描述每次课的授课情况或分段情况。该关系模式为：

section (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*)

图 2-6 给出了 *section* 关系的一个示例。

我们需要一个关系来描述教师和他们所讲授的课程段之间的联系。描述此联系的关系模式是：

43

teaches (*ID*, *course_id*, *sec_id*, *semester*, *year*)

course_id	sec_id	semester	year	building	room_number	time_slot_id
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

图 2-6 *section* 关系

图 2-7 给出了 *teaches* 关系的一个示例。

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

图 2-7 *teaches* 关系

正如你可以料想的，在一个真正的大学数据库中还维护了更多的关系。除了我们已经列出的这些关系：*instructor*、*department*、*course*、*section*、*prereq* 和 *teaches*，在本书中我们还要使用下列关系：

- *student* (*ID*, *name*, *dept_name*, *tot_cred*)
- *advisor* (*s_id*, *i_id*)
- *takes* (*ID*, *course_id*, *sec_id*, *semester*, *year*, *grade*)
- *classroom* (*building*, *room_number*, *capacity*)
- *time_slot* (*time_slot_id*, *day*, *start_time*, *end_time*)

2.3 码

我们必须有一种能区分给定关系中的不同元组的方

去。这用它们的属性来表明。也就是说，一个元组的属性值必须是能够唯一区分元组的。换句话说，一个关系中没有两个元组在所有属性上的取值都相同。

超码(superkey)是一个或多个属性的集合，这些属性的组合可以使我们在一个关系中唯一地标识一个元组。例如，*instructor* 关系的 *ID* 属性足以将不同的教师元组区分开来，因此，*ID* 是一个超码。另一方面，*instructor* 的 *name* 属性却不是一个超码，因为几个教师可能同名。

形式化地描述，设 R 表示关系 r 模式中的属性集合。如果我们说 R 的一个子集 K 是 r 的一个超码，则限制了关系 r 中任意两个不同元组不会在 K 的所有属性上取值完全相等。即如果 t_1 和 t_2 在 r 中且 $t_1 \neq t_2$，则 $t_1.K \neq t_2.K$。

超码中可能包含无关紧要的属性。例如，*ID* 和 *name* 的组合是关系 *instructor* 的一个超码。如果 K 是一个超码，那么 K 的任意超集也是超码。我们通常只对这样的一些超码感兴趣，它们的任意真子集都不能成为超码。这样的最小超码称为**候选码**(candidate key)。

几个不同的属性集都可以做候选码的情况是存在的。假设 *name* 和 *dept_name* 的组合足以区分 *instructor* 关系的各个成员，那么 {*ID*} 和 {*name*, *dept_name*} 都是候选码。虽然属性 *ID* 和 *name* 一起能区分 *instructor* 元组，但它们的组合 {*ID*, *name*} 并不能成为候选码，因为单独的属性 *ID* 已是候选码。

我们用**主码**(primary key)这个术语来代表被数据库设计者选中的、主要用来在一个关系中区分不同元组的候选码。码(不论是主码、候选码或超码)是整个关系的一种性质，而不是单个元组的性质。关系中的任意两个不同的元组都不允许同时在码属性上具有相同的值。码的指定代表了被建模的事物在现实世界中的约束。

主码的选择必须慎重。正如我们所注意到的那样，人名显然是不足以作主码的，因为可能有多个人重名。在美国，人的社会保障号可以作候选码。而非美国居民通常不具有社会保障号，所以跨国企业必须设置他们自己的唯一标识符。另外也可以使用另一些属性的唯一组合作为码。

44
45

主码应该选择那些值从不或极少变化的属性。例如，一个人的地址就不应该作为主码的一部分，因为它很可能变化。另一方面，社会保障号却可以保证决不变化。企业产生的唯一标识符通常不变，除非两个企业合并了，这种情况下可能在两个公司中会使用相同的标识符，因此需要重新分配标识符以确保其唯一性。

习惯上把一个关系模式的主码属性列在其他属性前面；例如，*department* 中的 *dept_name* 属性最先列出，因为它是主码。主码属性还加上了下划线。

一个关系模式(如 r_1)可能在它的属性中包括另一个关系模式(如 r_2)的主码。这个属性在 r_1 上称作参照 r_2 的**外码**(foreign key)。关系 r_1 也称为外码依赖的**参照关系**(referencing relation)，r_2 叫做外码的被**参照关系**(referenced relation)。例如，*instructor* 中的 *dept_name* 属性在 *instructor* 上是外码，它参照 *department*，因为 *dept_name* 是 *department* 的主码。在任意的数据库实例中，从 *instructor* 关系中任取一个元组，比如 t_1，在 *department* 关系中必定存在某个元组，比如 t_2，使得 t_1 在 *dept_name* 属性上的取值与 t_2 在主码 *dept_name* 上的取值相同。

现在考察 *section* 和 *teaches* 关系。如下需求是合理的：如果一门课程是分段授课的，那么它必须至少由一位教师来讲授；当然它可能由不止一位教师来讲授。为了施加这种约束，我们需要保证如果一个特定的(*course_id*, *sec_id*, *semester*, *year*)组合出现在 *section* 中，那么该组合也必须出现在 *teaches* 中。可是，这组值并不构成 *teaches* 的主码。因为不止一位教师可能讲授同一个这样的课程段。其结果是，我们不能声明从 *section* 到 *teaches* 的外码约束(尽管我们可以在相反的方向上声明从 *teaches* 到 *section* 的外码约束)。

从 *section* 到 *teaches* 的约束是**参照完整性约束**(referential integrity constraint)的一个例子。参照完整性约束要求在参照关系中任意元组在特定属性上的取值必然等于被参照关系中某个元组在特定属性上的取值。

2.4 模式图

一个含有主码和外码依赖的数据库模式可以用**模式图**(schema diagram)来表示。图 2-8 展示了我们

大学组织的模式图。每一个关系用一个矩形来表示，关系的名字显示在矩形上方，矩形内列出各属性。主码属性用下划线标注。外码依赖用从参照关系的外码属性到被参照关系的主码属性之间的箭头来
46 表示。

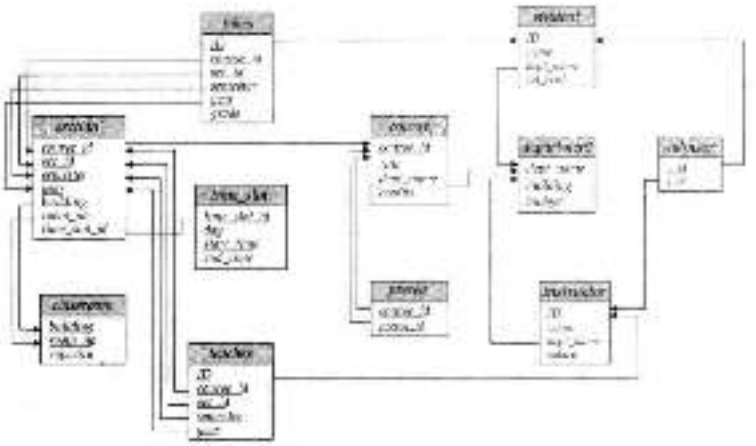


图 2-8 大学数据库的模式图

除外码约束之外，模式图中没有显示表示出参照完整性约束。在后面第 7 章，我们将学习一种不同的、称作实体 - 联系图的图形化表示。实体 - 联系图有助于我们表示几种约束，包括通用的参照完整性约束。

很多数据库系统提供图形化用户界面设计工具来建立模式图。我们将在第 7 章详细讨论模式的图形化表示。

在后面的章节中我们使用大学作为例子。图 2-9 给出了我们在例子中使用的关系模式，其中主码属性被标上了下划线。正如我们将在第 3 章中看到的一样，这对应于在 SQL 的数据定义语言中定义关系的方法。

```
classroom(building, room_number, capacity)
department(dept_name, building, budget)
course(course_id, title, dept_name, credits)
instructor(ID, name, dept_name, salary)
section(course_id, sec_id, semester, year, building, room_number, time_slot_id)
teaches(ID, course_id, sec_id, semester, year)
student(ID, name, dept_name, tot_cred)
takes(ID, course_id, sec_id, semester, year, grade)
advisor(s_ID, i_ID)
time_slot(time_slot_id, day, start_time, end_time)
prereq(course_id, prereq_id)
```

图 2-9 大学数据库模式

2.5 关系查询语言

查询语言(query language)是用户用来从数据库中请求获取信息的语言。这些语言通常比标准的程序设计语言层次更高。查询语言可以分为过程化的和非过程化的。在**过程化语言**(procedural language)中，用户指导系统对数据库执行一系列操作以计算出所需结果。在**非过程化语言**(nonprocedural language)中，用户只需描述所需信息，而不用给出获取该信息的具体过程。

实际使用的查询语言既包含过程化方式的成分，又包含非过程化方式的成分。我们从第 3 章到第 5 章学习被广泛应用的查询语言 SQL。

有一些"纯"查询语言：关系代数是过程化的，而元组关系演算和域关系演算是非过程化的。这些语言简洁且形式化，缺乏商用语言的"句法修饰"，但它们说明了从数据库中提取数据的基本技术。在

第 6 章，我们详细研究关系代数和关系演算的两种形式，即元组关系演算和域关系演算。关系代数包括一个运算的集合，这些运算以一个或两个关系为输入，产生一个新的关系作为结果。关系演算使用谓词逻辑来定义所需的结果，但不需给出获取结果的特定代数过程。

2.6　关系运算

所有的过程化关系查询语言都提供了一组运算，这些运算要么施加于单个关系上，要么施加于一对关系上。这些运算具有一个很好的、并且也是所需的性质：运算结果总是单个的关系。这个性质使得人们可以模块化的方式来组合几种这样的运算。特别是，由于关系查询的结果本身也是关系，所以关系运算可施加到查询结果上，正如施加到给定关系集上一样。

在不同语言中，特定的关系运算的表示是不同的，但都符合我们在本章所描述的通用结构。在第 3 章，我们将给出在 SQL 中表达关系运算的特殊方式。

最常用的关系运算是从单个关系（如 *instructor*）中选出满足一些特定谓词（如 *salary* > 85 000 美元）的特殊元组。其结果是一个新关系，它是原始关系（*instructor*）的一个子集。例如，如果我们从图 2-1 的 *instructor* 关系中选择满足谓词"工资大于 85 000 美元"的元组，我们得到的结果如图 2-10 所示。

47
48

ID	*name*	*dept_name*	*salary*
12121	Wu	Finance	90000
22222	Einstein	Physics	95000
33456	Gold	Physics	87000
83821	Brandt	Comp. Sci.	92000

图 2-10　选择工资大于 85 000 美元的 *instructor* 元组的查询结果

另一个常用的运算是从一个关系中选出特定的属性（列）。其结果是一个只包含那些被选择属性的新关系。例如，假设我们从图 2-1 的 *instructor* 关系中只希望列出教师的 *ID* 和工资，但不列出 *name* 和 *dept_name* 的值，那么其结果有 *ID* 和 *salary* 两个属性，如图 2-11 所示。结果中的每个元组都是从 *instructor* 关系中的某个元组导出的，不过只具有被选中的属性。

ID	*salary*
10101	65000
12121	90000
15151	40000
22222	95000
32343	60000
33456	87000
45565	75000
58583	62000
76543	80000
76766	72000
83821	92000
98345	80000

图 2-11　从 *instructor* 关系中选取属性 *ID* 和 *salary* 的查询结果

连接运算可以通过下述方式来结合两个关系：把分别来自两个关系的元组对合并成单个元组。有几种不同的方式来对关系进行连接（正如我们将在第 3 章中看到的）。图 2-12 显示了一个连接来自 *instructor* 和 *department* 表中元组的例子，新元组给出了有关每个教师及其工作所在系的信息。此结果是通过把 *instructor* 关系中的每个元组和 *department* 关系中对应于教师所在系的元组合并形成的。

图 2-12 所示的连接被称作自然连接，在这种连接形式中，对于来自 *instructor* 关系的一个元组与 *department* 关系中的一个元组来说，如果它们在 *dept_name* 属性上的取值相同，那它们就是匹配的。所有这样匹配的元组对都会在连接结果中出现。通常说来，两个关系上的自然连接运算所匹配的元组在两个关系共有的所有属性上取值相同。

笛卡儿积运算从两个关系中合并元组，但不同于连接运算的是，其结果包含来自两个关系元组的所有对，无论它们的属性值是否匹配。

ID	*name*	*salary*	*dept_name*	*building*	*budget*
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
12121	Wu	90000	Finance	Painter	120000
15151	Mozart	40000	Music	Packard	80000
22222	Einstein	95000	Physics	Watson	70000
32343	El Said	60000	History	Painter	50000
33456	Gold	87000	Physics	Watson	70000
45565	Katz	75000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
76543	Singh	80000	Finance	Painter	120000
76766	Crick	72000	Biology	Watson	90000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000

图 2-12　*instructor* 关系和 *department* 关系的自然连接结果

因为关系是集合，所以我们可以在关系上施加标准的集合运算。并运算在两个"相似结构"的表（比如一个所有毕业生的表和一个所有大学生的表）上执行集合并。例如，我们可以得到一个系中所有学生的集合。另外的集合运算如交和集合差也都可以被执行。

正如我们此前讲到的，我们可以在查询结果上施加运算。例如，如果我们想找出工资超过85 000美元的那些教师的 *ID* 和 *salary*，我们可以执行上述例子中的前两种运算。首先我们从 *instructor* 关系中选出 *salary* 值大于85 000美元的元组，然后从结果中选出 *ID* 和 *salary* 两个属性。结果关系如图2-13所示，由 *ID* 和 *salary* 构成。在此例中，我们可以任一次序来执行运算，但正如我们将看到的，并非在所有情况下均可如此。

ID	*salary*
12121	90000
22222	95000
33456	87000
83821	92000

图2-13 选择工资大于85 000美元的教师的 *ID* 和 *salary* 属性的结果

有时候，查询结果中包含重复的元组。例如，如果我们从 *instructor* 关系中选出 *dept_name* 属性，就会有好几种重复的情况，其中包括"Comp. Sci."，它出现了二次。一些关系语言严格遵守集合的数学定义，去除了重复。另一些考虑到从大的结果关系中去除重复需要大量相关的处理，就保留了重复。在后面这类情况中，关系并非是纯粹数学意义上的真正关系。

当然，数据库中的数据是随时间而改变的。关系可以随新元组的插入、已有元组的删除或更改元组在特定属性上的值而更新。整个关系可被删除，新的关系可被创建。

从第3章到第5章，我们将讨论如何使用SQL语言来表示关系的查询和更新。

关系代数

关系代数定义了在关系上的一组运算，对应于作用在数字上的普通代数运算，如加法、减法或乘法。正如作用在数字上的代数运算以一个或多个数字作为输入，返回一个数字作为输出，关系代数运算通常以一个或两个关系作为输入，返回一个关系作为输出。

第6章将详细介绍关系代数，下面我们给出几个运算的概述：

符号（名字）	使用示例
σ（选择）	$\sigma_{salary>85000}(instructor)$
	返回输入关系中满足谓词的行
Π（投影）	$\Pi_{ID,salary}(instructor)$
	对输入关系的所有行输出指定的属性，从输出中去除重复元组
$\bowtie$（自然连接）	$instructor \times department$
	从两个输入关系中输出这样的元组对：它们在具有相同名字的所有属性上取值相同
$\times$（笛卡儿积）	$instructor \times department$
	从两个输入关系中输出所有的元组对（无论它们在共同属性上的取值是否相同）
$\cup$（并）	$\Pi_{name}(instructor) \cup \Pi_{name}(student)$
	输出两个输入关系中元组的并

2.7 总结

- **关系数据模型**（relational data model）建立在表的集合的基础上。数据库系统的用户可以对这些表进行查询，可以插入新元组、删除元组以及更新（修改）元组。表达这些操作的语言有几种。
- 关系的**模式**（schema）是指它的逻辑设计，而关系的**实例**（instance）是指它在特定时刻的内容。数据库的模式和实例的定义是类似的。关系的模式包括它的属性，还可能包括属性类型和关系上的约束，比如主码和外码约束。

- 关系的**超码**(superkey)是一个或多个属性的集合，这些属性上的取值保证可以唯一识别出关系中的元组。候选码是一个最小的超码，也就是说，它是一组构成超码的属性集，但这组属性的任意子集都不是超码。关系的一个候选码被选作**主码**(primary key)。
- 在参照关系中的**外码**(foreign key)是这样的一个属性集合：对于参照关系中的每个元组来说，它在外码属性上的取值肯定等于被参照关系中某个元组在主码上的取值。
- **模式图**(schema diagram)是数据库中模式的图形化表示，它显示了数据库中的关系、关系的属性、主码和外码。
- **关系查询语言**(relational query language)定义了一组运算集，这些运算可作用于表上，并输出表作为结果。这些运算可以组合成表达式，表达所需的查询。
- **关系代数**(relational algebra)提供了一组运算，它们以一个或多个关系为输入，返回一个关系作为输出。诸如SQL这样的实际查询语言是基于关系代数的，但增加了一些有用的句法特征。

术语回顾

- 表
- 关系
- 元组
- 空值
- 数据库模式
- 数据库实例
- 关系模式
- 关系实例
- 码
  - 超码
  - 候选码
  - 主码
- 外码
  - 参照关系
  - 被参照关系
- 属性
- 域
- 原子域
- 参照完整性约束
- 模式图
- 查询语言
  - 过程化语言
  - 非过程化语言
- 关系运算
  - 选择元组
  - 选择属性
  - 自然连接
  - 笛卡儿积
  - 集合运算
- 关系代数

实践习题

2.1 考虑图2-14所示关系数据库。这些关系上适当的主码是什么？

2.2 考虑从 *instructor* 的 *dept_name* 属性到 *department* 关系的外码约束，给出对这些关系的插入和删除示例，使得它们破坏外码约束。

2.3 考虑 *time_slot* 关系。假设一个特定的时间段可以在一周之内出现多次，解释为什么 *day* 和 *start_time* 是该关系主码的一部分，而 *end_time* 却不是。

2.4 在图2-1所示 *instructor* 的实例中，没有两位教师同名。我们是否可以据此断定 *name* 可用来作为 *instructor* 的超码(或主码)？

2.5 先执行 *student* 和 *advisor* 的笛卡儿积，然后在结果上执行基于谓词 $s\_id = ID$ 的选择运算，最后的结果是什么？(采用关系代数的符号表示，此查询可写作 $\sigma_{s\_id=ID}(student \times advisor)$。)

employee(*person-name*, *street*, *city*)
works(*person-name*, *company-name*, *salary*)
company(*company-name*, *city*)

图2-14 习题2.1、习题2.7和习题2.12的关系数据库

2.6 考虑下面的表达式，哪些使用了关系代数运算的结果来作为另一个运算的输入？对于每个表达式，说明表达式的含义。

a. $(\sigma_{grade='A'}(takes) \times student)$

b. $(\sigma_{tot>200}(takes \times student))$

c. $\Pi_{ID,name,course\_id}(student \times takes)$

2.7 考虑图2-14所示关系数据库。给出关系代数表达式来表示下列每一个查询：

a. 找出居住在"Miami"城市的所有员工姓名。

49
53

b. 找出工资在 100 000 美元以上的所有员工姓名。

c. 找出居住在"Miami"并且工资在 100 000 美元以上的所有员工姓名。

2.8 考虑图 2-15 所示银行数据库。对于下列每个查询，给出一个关系代数表达式：

a. 找出位于"Chicago"的所有支行名字。

b. 找出在支行"Downtown"有贷款的所有贷款人姓名。

```
branch (branch_name, branch_city, assets)
customer (customer_name, customer_street, customer_city)
loan (loan_number, branch_name, amount)
borrower (customer_name, loan_number)
account (account_number, branch_name, balance)
depositor (customer_name, account_number)
```

图 2-15 习题 2.8、习题 2.9 和习题 2.13 的银行数据库

习题

2.9 考虑图 2-15 所示银行数据库。

a. 适当的主码是什么？

b. 给出你选择的主码，确定适当的外码。

2.10 考虑图 2-8 所示 *advisor* 关系，*advisor* 的主码是 *s_id*。假设一个学生可以有多位指导老师。那么，*s_id* 还是 *advisor* 关系的主码吗？如果不是，*advisor* 的主码会是什么呢？

54 2.11 解释术语关系和关系模式在意义上的区别。

2.12 考虑图 2-14 所示关系数据库。给出关系代数表达式来表示下列每一个查询：

a. 找出为"First Bank Corporation"工作的所有员工姓名。

b. 找出为"First Bank Corporation"工作的所有员工的姓名和居住城市。

c. 找出为"First Bank Corporation"工作且挣钱超过 10 000 美元的所有员工的姓名、街道地址和居住城市。

2.13 考虑图 2-15 所示银行数据库。对于下列每个查询，给出一个关系代数表达式：

a. 找出贷款额度超过 10 000 美元的所有贷款号。

b. 找出所有这样的存款人姓名，他拥有一个存款额大于 6000 美元的账户。

c. 找出所有这样的存款人姓名，他在"Uptown"支行拥有一个存款额大于 6000 美元的账户。

2.14 列出在数据库中引入空值的两个原因。

2.15 讨论过程化和非过程化语言的相对优点。

文献注解

IBM San Jose 研究实验室的 E. F. Codd 于 20 世纪 60 年代末提出了关系模型[Codd [1970]]。这一工作使 Codd 在 1981 年获得了声望很高的 ACM 图灵奖[Codd[1982]]。

在 Codd 最初的论文发表之后，几个研究项目开始进行，它们的目标是构造实际的关系数据库系统，其中包括 IBM San Jose 研究实验室的 System R、加州大学 Berkeley 分校的 Ingres，以及 IBM T. J. Watson 研究实验中心的 Query-by-Example。

大量关系数据库产品现在可以从市场上购得。其中包括 IBM 的 DB2 以及 Informix、Oracle、Sybase 和微软的 SQL Server。开源关系数据库系统包括 MySQL 和 PostgreSQL。微软的 Access 是一个单用户的数据库产品，它作为微软 Office 套件的一部分。

55 / 56 Atzeni 和 Antonellis[1993]、Maier[1983]以及 Abiteboul 等[1995]是专门讨论关系数据模型的文献。

SQL

商业性使用或实验性使用的数据库查询语言有好几种。在本章以及第4章和第5章，我们学习使用最为广泛的查询语言：SQL。

尽管我们说SQL语言是一种"查询语言"，但是除了数据库查询，它还具有很多别的功能。它可以定义数据结构，修改数据库中的数据以及说明安全性约束条件等。

我们的目的并不是提供一个完整的SQL用户手册，而是介绍SQL的基本结构和概念。SQL的各种实现可能在一些细节上有所不同，或者只支持整个语言的一个子集。

3.1 SQL查询语言概览

SQL最早的版本是由IBM开发的，它最初被叫做Sequel，在20世纪70年代早期作为System R项目的一部分。Sequel语言一直发展至今，其名称已变为SQL(结构化查询语言)。现在有许多产品支持SQL语言，SQL已经很明显地确立了自己作为标准的关系数据库语言的地位。

1986年美国国家标准化组织(ANSI)和国际标准化组织(ISO)发布了SQL标准：SQL-86。1989年ANSI发布了一个SQL的扩充标准：SQL-89。该标准的下一个版本是SQL-92标准，接着是SQL:1999，SQL:2003，SQL:2006，最新的版本是SQL:2008。文献注解中提供了关于这些标准的参考文献。

SQL语言有以下几个部分：

- **数据定义语言**(Data-Definition Language，DDL)：SQL DDL提供定义关系模式，删除关系以及修改关系模式的命令。
- **数据操纵语言**(Data-Manipulation Language，DML)：SQL DML提供从数据库中查询信息，以及在数据库中插入元组、删除元组，修改元组的能力。
- **完整性**(integrity)：SQL DDL包括定义完整性约束的命令，保存在数据库中的数据必须满足所定义的完整性约束。破坏完整性约束的更新是不允许的。
- **视图定义**(view definition)：SQL DDL包括定义视图的命令。
- **事务控制**(transaction control)：SQL包括定义事务的开始和结束的命令。
- **嵌入式SQL和动态SQL**(embedded SQL and dynamic SQL)：嵌入式和动态SQL定义SQL语句如何嵌入到通用编程语言，如C、C++和Java中。
- **授权**(authorization)：SQL DDL包括定义对关系和视图的访问权限的命令。

本章我们给出对SQL的基本DML和DDL特征的概述。在此描述的特征自SQL-92以来就一直是SQL标准的部分。

在第4章我们提供对SQL查询语言更详细的介绍，包括：(a)各种连接的表达；(b)视图；(c)事务；(d)完整性约束；(e)类型系统；(f)授权。

在第5章我们介绍SQL语言更高级的特征，包括：(a)允许从编程语言中访问SQL的机制；(b)SQL函数和过程；(c)触发器；(d)递归查询；(e)高级聚集特征；(f)为数据分析设计的一些特征，它们在SQL:1999中引入，并在SQL的后续版本中使用。在后面第22章，我们将概述SQL:1999引入的对SQL的面向对象扩充。

尽管大多数SQL实现支持我们在此描述的标准特征，读者还是应该意识到不同SQL实现之间的差异。大多数SQL实现还支持一些非标准的特征，但不支持一些更高级的特征。为一些发现在此描述的

一些语言特征在你使用的系统中不起作用，请参考你的数据库系统用户手册，看看它所支持的特征究竟是什么。

3.2 SQL 数据定义

数据库中的关系集合必须由数据定义语言(DDL)指定给系统。SQL 的 DDL 不仅能够定义一组关系，还能够定义每个关系的信息，包括：

- 每个关系的模式。
- 每个属性的取值类型。
- 完整性约束。
- 每个关系维护的索引集合。
- 每个关系的安全性和权限信息。
- 每个关系在磁盘上的物理存储结构。

我们在此只讨论基本模式定义和基本类型，对 SQL DLL 其他特征的讨论将放到第 4 章和第 5 章进行。

3.2.1 基本类型

SQL 标准支持多种固有类型，包括：

- **char**(n)：固定长度的字符串，用户指定长度 n。也可以使用全称 **character**。
- **varchar**(n)：可变长度的字符串，用户指定最大长度 n，等价于全称 **character varying**。
- **int**：整数类型(和机器相关的整数的有限子集)，等价于全称 **integer**。
- **smallint**：小整数类型(和机器相关的整数类型的子集)。
- **numeric**(p, d)：定点数，精度由用户指定。这个数有 p 位数字(加上一个符号位)，其中 d 位数字在小数点右边。所以在一个这种类型的字段上，**numeric**(3, 1)可以精确储存 44.5，但不能精确存储 444.5 或 0.32 这样的数。
- **real**, **double precision**：浮点数与双精度浮点数，精度与机器相关。
- **float**(n)：精度至少为 n 位的浮点数。

更多类型将在 4.5 节介绍。

每种类型都可能包含一个被称作空值的特殊值。空值表示一个缺失的值，该值可能存在但并不为人所知，或者可能根本不存在。在可能的情况下，我们希望禁止加入空值，正如我们马上将看到的那样。

char 数据类型存放固定长度的字符串。例如，属性 A 的类型是 **char**(10)。如果我们为此属性存入字符串"Avi"，那么该字符串后会追加 7 个空格来使其达到 10 个字符的串长度。反之，如果属性 B 的类型是 **varchar**(10)，我们在属性 B 中存入字符串"Avi"，则不会增加空格。当比较两个 **char** 类型的值时，如果它们的长度不同，在比较之前会自动在短值后面加上额外的空格以使它们的长度一致。

当比较一个 **char** 类型和一个 **varchar** 类型的时候，也许读者会期望在比较之前会自动在 **varchar** 类型后面加上额外的空格以使长度一致；然而，这种情况可能发生也可能不发生，这取决于数据库系统。其结果是，即便上述属性 A 和 B 中存放的是相同的值"Avi"，$A = B$ 的比较也可能返回假。我们建
59 议始终使用 **varchar** 类型而不是 **char** 类型来避免这样的问题。

SQL 也提供 **nvarchar** 类型来存放使用 Unicode 表示的多语言数据。然而，很多数据库甚至允许在 **varchar** 类型中存放 Unicode(采用 UTF-8 表示)。

3.2.2 基本模式定义

我们用 **create table** 命令定义 SQL 关系。下面的命令在数据库中创建了一个 *department* 关系：

```
create table department
    (dept_name varchar (20),
     building varchar (15),
     budget numeric (12, 2),
     primary key (dept_name));
```

上面创建的关系具有三个属性，*dept_name* 是最大长度为 20 的字符串，*building* 是最大长度为 15 的字符串，*budget* 是一个 12 位的数，其中 2 位数字在小数点后面。**create table** 命令还指明了 *dept_name* 属性是 *department* 关系的主码。

create table 命令的通用形式是：

```
create table r
    (A1  D1,
    A2  D2,
    ...,
    An  Dn,
    <完整性约束1>,
    ...,
    <完整性约束k>);
```

其中 r 是关系名，每个 A_i 是关系 r 模式中的一个属性名，D_i 是属性 A_i 的域。也就是说 D_i 指定了属性 A_i 的类型以及可选的约束，用于限制所允许的 A_i 取值的集合。

create table 命令后面用分号结束，本章后面的其他 SQL 语句也是如此，在很多 SQL 实现中，分号是可选的。

SQL 支持许多不同的完整性约束。在本节我们只讨论其中少数几个：

- **primary key**(A_1, A_2, …, A_n)：**primary-key** 声明表示属性 A_1, A_2, …, A_n 构成关系的主码。主码属性必须非空且唯一，也就是说没有一个元组在主码属性上取空值，关系中也没有两个元组在所有主码属性上取值相同。虽然主码的声明是可选的，但为每个关系指定一个主码通常会更好。
- **foreign key**(A_1, A_2, …, A_n) **references** *s*：**foreign key** 声明表示关系中任意元组在属性(A_1, A_2, …, A_n)上的取值必须对应于关系 *s* 中某元组在主码属性上的取值。

  图 3-1 给出了我们在书中使用的大学数据库的部分 SQL DDL 定义。*course* 表的定义中声明了"**foreign key** (*dept_name*) **references** *department*"。此外码声明表明对于每个课程元组来说，该元组所表示的系名必然存在于 *department* 关系的主码属性(*dept_name*)中。没有这个约束的话，就可能有某门课程指定了一个不存在的系名。图 3-1 还给出了表 *section*、*instructor* 和 *teaches* 上的外码约束。

```
create table department
    (dept_name      varchar(20),
    building        varchar (15),
    budget          numeric (12,2),
    primary key (dept_name));

create table course
    (course_id      varchar (7),
    title           varchar (50),
    dept_name       varchar (20),
    credits         numeric (2,0),
    primary key (course_id),
    foreign key (dept_name) references department);

create table instructor
    (ID             varchar (5),
    name            varchar (20) not null,
    dept_name       varchar (20),
    salary          numeric (8,2),
    primary key (ID),
    foreign key (dept_name) references department);

create table section
    (course_id      varchar (8),
    sec_id          varchar (8),
    semester        varchar (6),
    year            numeric (4,0),
    building        varchar (15),
    room_number     varchar (7),
    time_slot_id    varchar (4),
    primary key (course_id, sec_id, semester, year),
    foreign key (course_id) references course);

create table teaches
    (ID             varchar (5),
    course_id       varchar (8),
    sec_id          varchar (8),
    semester        varchar (6),
    year            numeric (4,0),
    primary key (ID, course_id, sec_id, semester, year),
    foreign key (course_id, sec_id, semester, year)
references section,
    foreign key (ID) references instructor);
```

图 3-1 大学数据库的部分 SQL 数据定义

- **not null**：一个属性上的 **not null** 约束表明在该属性上不允许空值。换句话说，此约束把空值排除在该属性域之外。例如在图 3-1 中，*instructor* 关系的 *name* 属性上的 **not null** 约束保证了教师的姓名不会为空。

有关外码约束的更多细节以及 **create table** 命令可能包含的其他完整性约束将在后面 4.4 节介绍。

SQL 禁止破坏完整性约束的任何数据库更新。例如，如果关系中一条新插入或新修改的元组在任意一个主码属性上有空值，或者元组在主码属性上的取值与关系中的另一个元组相同，SQL 将标记一个错误，并阻止更新。类似地，如果插入的 *course* 元组在 *dept_name* 上的取值没有出现在 *department* 关系中，就会破坏 *course* 上的外码约束，SQL 会阻止这种插入的发生。

一个新创建的关系最初是空的。我们可以用 **insert** 命令将数据加载到关系中。例如，如果我们希望插入如下事实：在 Biology 系有一个名叫 Smith 的教师，其 *instructor_id* 为 10211，工资为 66 000 美元，可以这样写：

```
insert into instructor
    values (10211, 'Smith', 'Biology', 66000);
```

值被给出的顺序应该遵循对应属性在关系模式中列出的顺序。插入命令有很多有用的特性，后面将在 3.9.2 节进行更详细的介绍。

我们可以使用 **delete** 命令从关系中删除元组。命令

```
delete from student;
```

将从 *student* 关系中删除所有元组。其他格式的删除命令允许指定待删除的元组；我们将在 3.9.1 节对删除命令进行更详细的介绍。

如果要从 SQL 数据库中去掉一个关系，我们使用 **drop table** 命令。**drop table** 命令从数据库中删除关于被去掉关系的所有信息。命令

```
drop table r;
```

是比

```
delete from r;
```

更强的语句。后者保留关系 r，但删除 r 中的所有元组。前者不仅删除 r 的所有元组，还删除 r 的模式。一旦 r 被去掉，除非用 **create table** 命令重建 r，否则没有元组可以插入到 r 中。

我们使用 **alter table** 命令为已有关系增加属性。关系中的所有元组在新属性上的取值将被设为 *null*。**alter table** 命令的格式为：

```
alter table r add A D;
```

其中 r 是现有关系的名字，A 是待添加属性的名字，D 是待添加属性的域。我们可以通过命令

```
alter table r drop A;
```

从关系中去掉属性。其中 r 是现有关系的名字，A 是关系的一个属性的名字。很多数据库系统并不支持去掉属性，尽管它们允许去掉整个表。

3.3 SQL 查询的基本结构

SQL 查询的基本结构由三个子句构成：**select**、**from** 和 **where**。查询的输入是在 **from** 子句中列出的关系，在这些关系上进行 **where** 和 **select** 子句中指定的运算，然后产生一个关系作为结果。我们通过例子介绍 SQL 的语法，后面再描述 SQL 查询的通用结构。

3.3.1 单关系查询

我们考虑使用大学数据库例子的一个简单查询："找出所有教师的名字"。教师的名字可以在 *instructor* 关系中找到，因此我们把该关系放到 **from** 子句中。教师的名字出现在 *name* 属性中，因此我们把它放到 **select** 子句中。

```
select name
from instructor;
```

其结果是由属性名为 *name* 的单个属性构成的关系。如果 *instructor* 关系如图 2-1 所示，那么上述查询的结果关系如图 3-2 所示。

name
Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Katz
Califieri
Singh
Crick
Brandt
Kim

图 3-2 "**select** *name* **from** *instructor*"的结果

现在考虑另一个查询："找出所有教师所在的系名"，此查询可写为：

```
select dept_name
from instructor;
```

因为一个系有多个教师，所以在 *instructor* 关系中，一个系的名称可以出现不止一次。上述查询的结果是一个包含系名的关系，如图 3-3 所示。

在关系模型的形式化数学定义中，关系是一个集合。因此，重复的元组不会出现在关系中。在实践中，去除重复是相当费时的，所以 SQL 允许在关系以及 SQL 表达式结果中出现重复。因此，在上述 SQL 查询中，每个系名在 *instructor* 关系的元组中每出现一次，都会在查询结果中列出一次。

dept_name
Comp. Sci.
Finance
Music
Physics
History
Physics
Comp. Sci.
History
Finance
Biology
Comp. Sci.
Elec. Eng.

图 3-3 "**select** *dept_name* **from** *instructor*"的结果

有时候我们想要强行删除重复，可在 **select** 后加入关键词 **distinct**。如果我们想去除重复，可将上述查询重写为：

```
select distinct dept_name
from instructor;
```

在上述查询的结果中，每个系名最多只出现一次。

SQL 允许我们使用关键词 **all** 来显式指明不去除重复：

```
select all dept_name
from instructor;
```

既然保留重复元组是默认的，在例子中我们将不再使用 **all**。为了保证在我们例子的查询结果中删除重复元组，我们将在所有必要的地方使用 **distinct**。

select 子句还可带含有 +、−、*、/运算符的算术表达式，运算对象可以是常数或元组的属性。例如，查询

```
select ID, name, dept_name, salary * 1.1
from instructor;
```

返回一个与 *instructor* 一样的关系，只是属性 *salary* 的值是原来的 1.1 倍。这显示了如果我们给每位教师增长 10% 的工资的结果。注意这并不导致对 *instructor* 关系的任何改变。

SQL 还提供了一些特殊数据类型，如各种形式的日期类型，并允许一些作用于这些类型上的算术函数。我们在 4.5.1 节进一步讨论这个问题。

where 子句允许我们只选出那些在 **from** 子句的结果关系中满足特定谓词的元组。考虑查询"找出所有在 Computer Science 系并且工资超过 70 000 美元的教师的姓名"，该查询用 SQL 可以写为：

```
select name
from instructor
where dept_name = 'Comp. Sci.' and salary > 70000;
```

如果 *instructor* 关系如图 2-1 所示，那么上述查询的结果关系如图 3-4 所示。

SQL 允许在 **where** 子句中使用逻辑连词 **and**、**or** 和 **not**。逻辑连词的运算对象可以是包含比较运算符 <、<=、>、>=、= 和 <> 的表达式。SQL 允许我们使用比较运算符来比较字符串、算术表达式以及特殊类型，如日期类型。

name
Katz
Brandt

图 3-4 "找出所有在 Computer Science 系并且工资超过 70 000 美元的教师的姓名"的结果

在本章的后面，我们将研究 **where** 子句谓词的其他特征。

60 / 65

3.3.2 多关系查询

到此为止我们的查询示例都是基于单个关系的。通常查询需要从多个关系中获取信息。我们现在来学习如何书写这样的查询。

作为一个示例，假设我们想回答这样的查询："找出所有教师的姓名，以及他们所在系的名称和系所在建筑的名称"。

考虑 *instructor* 关系的模式，我们发现可以从 *dept_name* 属性得到系名，但是系所在建筑的名称是在

department 关系的 *building* 属性中给出的。为了回答查询，*instructor* 关系中的每个元组必须与 *department* 关系中的元组匹配，后者在 *dept_name* 上的取值相配于 *instructor* 元组在 *dept_name* 上的取值。

为了在 SQL 中回答上述查询，我们把需要访问的关系都列在 **from** 子句中，并在 **where** 子句中指定匹配条件。上述查询可用 SQL 写为：

```
select name, instructor.dept_name, building
from instructor, department
where instructor.dept_name = department.dept_name;
```

如果 *instructor* 和 *department* 关系分别如图 2-1 和图 2-5 所示，那么此查询的结果关系如图 3-5 所示。

注意 *dept_name* 属性既出现在 *instructor* 关系中，也出现在 *department* 中，关系名被用作前缀（在 *instructor.dept_name* 和 *department.dept_name* 中）来说明我们使用的是哪个属性。相反，属性 *name* 和 *building* 只出现在一个关系中，因而不需要把关系名作为前缀。

name	dept_name	building
Srinivasan	Comp. Sci.	Taylor
Wu	Finance	Painter
Mozart	Music	Packard
Einstein	Physics	Watson
El Said	History	Painter
Gold	Physics	Watson
Katz	Comp. Sci.	Taylor
Califieri	History	Painter
Singh	Finance	Painter
Crick	Biology	Watson
Brandt	Comp. Sci.	Taylor
Kim	Elec. Eng.	Taylor

图 3-5 “找出所有教师的姓名，以及他们所在系的名称和系所在建筑的名称”的结果

这种命名惯例常要出现在 **from** 子句中的关系具有可区分的名字。在某些情况下这样的要求会引发问题，比如当需要把来自同一个关系的两个不同元组的信息进行组合的时候。在 3.4.1 节，我们将看到如何使用更名运算来避免这样的问题。

现在我们考虑涉及多个关系的 SQL 查询的通用形式。正如我们前面已经看到的，一个 SQL 查询可以包括三种类型的子句：**select** 子句、**from** 子句和 **where** 子句。每种子句的作用如下：

- **select** 子句用于列出查询结果中所需要的属性。
- **from** 子句是一个查询求值中需要访问的关系列表。
- **where** 子句是一个作用在 **from** 子句中关系的属性上的谓词。

一个典型的 SQL 查询具有如下形式：

```
select A1, A2, ···, An
from r1, r2, ···, rm
where P;
```

每个 A_i 代表一个属性。每个 r_i 代表一个关系。P 是一个谓词。如果省略 **where** 子句，则谓词 P 为 **true**。

66
67

尽管各子句必须以 **select**，**from**，**where** 的次序写出，但理解查询所代表运算的最容易的方式是以运算的顺序来考察各子句：首先是 **from**，然后是 **where**，最后是 **select** [⊖]。

通过 **from** 子句定义了一个在该子句中所列出关系上的笛卡儿积。它可以用集合理论来形式化地定义，但最好通过下面的迭代过程来理解，此过程可为 **from** 子句的结果关系产生元组。

```
for each 元组 t1 in 关系 r1
    for each 元组 t2 in 关系 r2
        ···
        for each 元组 tm in 关系 rm
        把 t1, t2, ···, tm 连接成一个元组 t
        把 t 加入结果关系中
```

此结果关系具有来自 **from** 子句中所有关系的所有属性。由于在关系 r_1 和 r_2 中可能出现相同的属性名，正如我们此前所看到的，我们在属性名前加上关系名作为前缀，表示该属性来自于哪个关系。

⊖ 实践中，SQL 也许会将表达式转换成更高效执行的等价形式。我们将把效率问题推迟到第 12 章和第 13 章中探讨。

例如，关系 *instructor* 和 *teaches* 的笛卡儿积的关系模式为：

(*instructor.ID*, *instructor.name*, *instructor.dept_name*, *instructor.salary*
teaches.ID, *teaches.course_id*, *teaches.sec_id*, *teaches.semester*, *teaches.year*)

有了这个模式，我们可以区分出 *instructor.ID* 和 *teaches.ID*。对于那些只出现在单个模式中的属性，我们通常去掉关系名前缀。这种简化并不会造成任何混淆。这样我们可以把关系模式写为：

(*instructor.ID*, *name*, *dept_name*, *salary*
teaches.ID, *course_id*, *sec_id*, *semester*, *year*)

为举例说明，考察图2-1中的 *instructor* 关系和图2-7中的 *teaches* 关系。它们的笛卡儿积如图3-6所示，图中只包括了构成笛卡儿积结果的一部分元组。[⊖]

inst.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	CompSci	65000	10101	CS-101	1	Fall	2009
10101	Srinivasan	CompSci	65000	10101	CS-315	1	Spring	2010
10101	Srinivasan	CompSci	65000	10101	CS-347	1	Fall	2009
10101	Srinivasan	CompSci	65000	12121	FIN-201	1	Spring	2010
10101	Srinivasan	CompSci	65000	15151	MU-199	1	Spring	2010
10101	Srinivasan	CompSci	65000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2009
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2010
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2009
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2010
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2010
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
15151	Mozart	Music	40000	10101	CS-101	1	Fall	2009
15151	Mozart	Music	40000	10101	CS-315	1	Spring	2010
15151	Mozart	Music	40000	10101	CS-347	1	Fall	2009
15151	Mozart	Music	40000	12121	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2010
15151	Mozart	Music	40000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2009
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2010
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2009
22222	Einstein	Physics	95000	12121	FIN-201	1	Spring	2010
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

图3-6 *instructor* 关系和 *teaches* 关系的笛卡儿积

通过笛卡儿积把来自 *instructor* 和 *teaches* 中相互没有关联的元组组合起来。*instructor* 中的每个元组和 *teaches* 中的所有元组都要进行组合，即使是那些代表不同教师的元组。其结果可能是一个非常庞大的关系，创建这样的笛卡儿积通常是没有意义的。

反之，**where** 子句中的谓词用来限制笛卡儿积所建立的组合，只留下那些对所需答案有意义的组合。我们希望有个涉及 *instructor* 和 *teaches* 的查询，它把 *instructor* 中的特定元组 t 只与 *teaches* 中表示跟 t 相同教师的元组进行组合。也就是说，我们希望把 *teaches* 元组只和具有相同 *ID* 值的 *instructor* 元组进行匹配。下面的SQL查询满足这个条件，从这些匹配元组中输出教师名和课程标识。

68
69

```
select name, course_id
from instructor, teaches
where instructor.ID = teaches.ID;
```

⊖ 为节约了这小图3-6中表的宽度，我们把 *instructor.ID* 更名为 *inst.ID*。

注意上述查询只输出讲授了课程的教师，不会输出那些没有讲授任何课程的教师。如果我们希望输出那样的元组，可以使用一种被称作外连接的运算，外连接将在 4.1.2 节讲述。

如果 *instructor* 关系如图 2-1 所示，*teaches* 关系如图 2-7 所示，那么前述查询的结果关系如图 3-7 所示。注意教师 Gold、Califieri 和 Singh，由于他们没有讲授任何课程，就不出现在上述结果中。

如果我们只希望找出 Computer Science 系的教师名和课程标识，我们可以在 **where** 子句中增加另外的谓词，如下所示：

```
select name, course_id
from instructor, teaches
where instructor.ID = teaches.ID and instructor.dept_name = 'Comp. Sci.';
```

注意既然 *dept_name* 属性只出现在 *instructor* 关系中，我们在上述查询中可以只使用 *dept_name* 来替代 *instructor.dept_name*。

name	*course_id*
Srinivasan	CS-101
Srinivasan	CS-315
Srinivasan	CS-347
Wu	FIN-201
Mozart	MU-199
Einstein	PHY-101
El Said	HIS-351
Katz	CS-101
Katz	CS-319
Crick	BIO-101
Crick	BIO-301
Brandt	CS-190
Brandt	CS-190
Brandt	CS-319
Kim	EE-181

图 3-7 “对于大学中所有讲授课程的教师，找出他们的姓名以及所讲述的所有课程标识”的结果

通常说来，一个 SQL 查询的含义可以理解如下：

1. 为 **from** 子句中列出的关系产生笛卡儿积。
2. 在步骤 1 的结果上应用 **where** 子句中指定的谓词。
3. 对于步骤 2 结果中的每个元组，输出 **select** 子句中指定的属性（或表达式的结果）。

上述步骤的顺序有助于明白一个 SQL 查询的结果应该是什么样的，而不是这个结果是怎样被执行的。在 SQL 的实际实现中不会执行这种形式的查询，它会通过（尽可能）只产生满足 **where** 子句谓词的笛卡儿积元素来进行优化执行。我们在后面第 12 章和第 13 章学习那样的实现技术。

当书写查询时，需要小心设置合适的 **where** 子句条件。如果在前述 SQL 查询中省略 **where** 子句条件，就会输出笛卡儿积，那是一个巨大的关系。对于图 2-1 中的 *instructor* 样本关系和图 2-7 中的 *teaches* 样本关系，它们的笛卡儿积具有 $12 \times 13 = 156$ 个元组，比我们在书中能显示的还要多！在更糟的情况下，假设我们有比图中所示样本关系更现实的教师数量，比如 200 个教师。假使每位教师讲授 3 门课程，那么我们在 *teaches* 关系中就有 600 个元组。这样上述迭代过程会产生出 $200 \times 600 = 120\,000$ 个元组作为结果。

3.3.3 自然连接

在我们的查询示例中，需要从 *instructor* 和 *teaches* 表中组合信息。匹配条件是需要 *instructor.ID* 等于 *teaches.ID*。这是在两个关系中只有相同名称的所有属性。实际上这是一种通用的情况，也就是说，**from** 子句中的匹配条件在最通常的情况下需要在所有匹配名称的属性上相等。

为了在这种通用情况下简化 SQL 编程者的工作，SQL 支持一种被称作自然连接的运算，下面我们就来讨论这种运算。事实上 SQL 还支持几种另外的方式使得来自两个或多个关系的信息可以被连接（join）起来。我们已经见过怎样利用笛卡儿积和 **where** 子句谓词来连接来自多个关系的信息。连接来自多个关系信息的其他方式在 4.1 节介绍。

自然连接（natural join）运算作用于两个关系，并产生一个关系作为结果。不同于两个关系上的笛卡儿积，它将第一个关系的每个元组与第二个关系的所有元组都进行连接；自然连接只考虑那些在两个关系模式中都出现的属性上取值相同的元组对。因此，回到 *instructor* 和 *teaches* 关系的例子上，*instructor* 和 *teaches* 的**自然连接**计算中只考虑这样的元组对：来自 *instructor* 的元组和来自 *teaches* 的元组在共同属性 *ID* 上的取值相同。

70 / 71

结果关系如图 3-8 所示，只有 13 个元组，它们给出了关于每个教师以及该教师实际讲授的课程的信息。注意我们并没有重复列出那些在两个关系模式中都出现的属性，这样的属性只出现一次。还要注意列出属性的顺序：先是两个关系模式中的共同属性，然后是那些只出现在第一个关系模式中的属性，最后是那些只出现在第二个关系模式中的属性。

考虑查询“对于大学中所有讲授课程的教师，找出他们的姓名以及所讲述的所有课程标识”，此前我们曾把该查询写为：

ID	name	dept_name	salary	course_id	sec_id	semester	year
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

图 3-8 *instructor* 关系和 *teaches* 关系的自然连接

```
select name, course_id
from instructor, teaches
where instructor.ID = teaches.ID;
```

该查询可以用SQL的自然连接运算更简洁地写作：

```
select name, course_id
from instructor natural join teaches;
```

以上两个查询产生相同的结果。

正如我们此前所见，自然连接运算的结果是关系。从概念上讲，**from** 子句中的"*instructor* **natural join** *teaches*"表达式可以替换成执行该自然连接后所得到的关系。"然后在这个关系上执行 **where** 和 **select** 子句，就如我们在前面 3.3.2 节所看到的那样。 72

在一个 SQL 查询的 **from** 子句中，可以用自然连接将多个关系结合在一起，如下所示：

```
select A1, A2, ···, An
from r1 natural join r2 natural join ··· natural join rm
where P;
```

更为一般地，**from** 子句可以为如下形式：

```
from E1, E2, ···, En
```

其中每个 E_i 可以是单个关系或一个包含自然连接的表达式。例如，假设我们要回答查询"列出教师的名字以及他们所讲授课程的名称"。此查询可以用 SQL 写为：

```
select name, title
from instructor natural join teaches, course
where teaches.course_id = course.course_id;
```

先计算 *instructor* 和 *teaches* 的自然连接。正如我们此前所见，再计算该结果和 *course* 的笛卡儿积，**where** 子句从这个结果中提取出这样的元组：来自连接结果的课程标识与来自 *course* 关系的课程标识相匹配。注意 **where** 子句中的 *teaches.course_id* 表示自然连接结果中的 *course_id* 域，因为该域最终来自 *teaches* 关系。

相反，下面的 SQL 查询不会计算出相同的结果：

```
select name, title
from instructor natural join teaches natural join course;
```

为了说明原因，注意 *instructor* 和 *teaches* 的自然连接包括属性（*ID*, *name*, *dept_name*, *salary*, *course_id*, *sec_id*），而 *course* 关系包含的属性是（*course_id*, *title*, *dept_name*, *credits*）。作为这二者自然连接的结果，

⊖ 其结果是不可能用包含了诸如关系名的属性名来指代自然连接结果中的属性，例如 *instructor.name* 或 *teaches.course_id*，但是我们可以使用诸如 *name* 和 *course_id* 那样的属性名，它不带关系名。

需要来自这两个输入的元组既要在属性 *dept_name* 上取值相同，还要在 *course_id* 上取值相同。该查询将忽略所有这样的(教师姓名，课程名称)对：其中教师所讲授的课程不是他所在系的课程。前面一个查询会正确输出这样的对。

为了发挥自然连接的优点，同时避免不必要的相等属性带来的危险，SQL 提供了一种自然连接的构造形式，允许用户来指定需要哪些列相等。下面的查询说明了这个特征：

```
select name, title
from (instructor natural join teaches) join course using (course_id);
```

join···**using** 运算中需要给定一个属性名列表，其两个输入中都必须具有指定名称的属性。考虑运算 r_1 **join** r_2 **using**(A_1, A_2)，它与 r_1 和 r_2 的**自然连接**类似，只不过在 $t_1.A_1 = t_2.A_1$ 并且 $t_1.A_2 = t_2.A_2$ 成立的前提下，来自 r_1 的元组 t_1 和来自 r_2 的元组 t_2 就能匹配。即使 r_1 和 r_2 都具有名为 A_1 的属性，也不需要 $t_1.A_1 = t_2.A_1$ 成立。

这样，在前述 SQL 查询中，**连接**构造允许 *teaches.dept_name* 和 *course.dept_name* 是不同的，该 SQL 查询给出了正确的答案。

3.4 附加的基本运算

SQL 中还支持几种附加的基本运算。

3.4.1 更名运算

重新考察我们此前使用过的查询：

```
select name, course_id
from instructor, teaches
where instructor.ID = teaches.ID;
```

此查询的结果是一个具有下列属性的关系：

name, *course_id*

结果中的属性名来自 **from** 子句中关系的属性名。

但我们不能总是用这个方法派生名字，其原因有几点：首先，**from** 子句的两个关系中可能存在同名属性，在这种情况下，结果中就会出现重复的属性名；其次，如果我们在 **select** 子句中使用算术表达式，那么结果属性就没有名字；再次，尽管如上例所示，属性名可以从基关系导出，但我们也许想要改变结果中的属性名字。因此，SQL 提供了一个重命名结果关系中属性的方法，即使用如下形式的 **as** 子句：

73
|
74

old-name **as** *new-name*

as 子句既可出现在 **select** 子句中，也可出现在 **from** 子句中。[⊖]

例如，如果我们想用名字 *instructor_name* 来代替属性名 *name*，我们可以重写上述查询如下：

```
select name as instructor_name, course_id
from instructor, teaches
where instructor.ID = teaches.ID;
```

as 子句在重命名关系时特别有用。重命名关系的一个原因是把一个长的关系名替换成短的，这样在查询的其他地方使用起来就更为方便。为了说明这一点，我们重写查询"对于大学中所有讲授课程的教师，找出他们的姓名以及所讲述的所有课程标识"：

```
select T.name, S.course_id
from instructor as T, teaches as S
where T.ID = S.ID;
```

⊖ SQL 语法标准不包括关键字 **as**，其结果是在一些 SQL 实现中，特别是 Oracle 中，不允许在 **from** 子句中出现关键字 **as**。在 Oracle 的 **from** 子句中，"*old-name* **as** *new-name*"被写作"*old-name new-name*"。在 **select** 子句中允许使用关键字 **as** 来重命名属性，但它是可选项，在 Oracle 中可以省略。

重命名关系的另一个原因是为了适用于需要比较同一个关系中的元组的情况。为此我们需要把一个关系跟它自身进行笛卡儿积运算，如果不重命名的话，就不可能把一个元组与其他元组区分开来。假设我们希望写出查询："找出满足下面条件的所有教师的姓名，他们的工资至少比 Biology 系某一个教师的工资要高"，我们可以写出这样的 SQL 表达式：

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept_name = 'Biology';
```

注意我们不能使用 *instructor.salary* 这样的写法，因为这样并不清楚到底是希望引用哪一个 *instructor*。

在上述查询中，*T* 和 *S* 可以被认为是 *instructor* 关系的两个拷贝，但更准确地说是被声明为 *instructor* 关系的别名，也就是另外的名字。像 *T* 和 *S* 那样被用来重命名关系的标识符在 SQL 标准中被称作**相关名称**(correlation name)，但通常也被称作**表别名**(table alias)，或者**相关变量**(correlation variable)，或者**元组变量**(tuple variable)。

注意用文字表达上述查询更好的方式是："找出满足下面条件的所有教师的姓名，他们比 Biology 系教师的最低工资要高"。我们早先的表述更符合我们所写的 SQL。但后面的表述更直观，事实上它可以直接用 SQL 来表达，正如我们将在 3.8.2 节看到的那样。

3.4.2 字符串运算

SQL 使用一对单引号来标示字符串，例如'Computer'。如果单引号是字符串的组成部分，那就用两个单引号字符来表示，如字符串"it's right"可表示为"it''s right"。

在 SQL 标准中，字符串上的相等运算是大小写敏感的，所以表达式"'comp. sci.' = 'Comp. Sci.'"的结果是假。然而一些数据库系统，如 MySQL 和 SQL Server，在匹配字符串时并不区分大小写，所以在这些数据库中"'comp. sci.' = 'Comp. Sci.'"的结果可能是真。然而这种默认方式是可以在数据库级或特定属性级被修改的。

SQL 还允许在字符串上有多种函数，例如串联(使用"||")、提取子串、计算字符串长度、大小写转换(用 **upper**(*s*)将字符串 *s* 转换为大写或用 **lower**(*s*)将字符串 *s* 转换为小写)、去掉字符串后面的空格(使用 **trim**(*s*))，等等。不同数据库系统所提供的字符串函数集是不同的，请参阅你的数据库系统手册来获得它所支持的实际字符串函数的详细信息。

在字符串上可以使用 **like** 操作符来实现模式匹配。我们使用两个特殊的字符来描述模式：

- 百分号(%)：匹配任意子串。
- 下划线(_)：匹配任意一个字符。

模式是大小写敏感的，也就是说，大写字符与小写字符不匹配，反之亦然。为了说明模式匹配，考虑下列例子：

- 'Intro%'匹配任何以"Intro"打头的字符串。
- '%Comp%'匹配任何包含"Comp"子串的字符串，例如'Intro. to Computer Science'和'Computational Biology'。
- '___'匹配只含三个字符的字符串。
- '___%'匹配至少含三个字符的字符串。

在 SQL 中用比较运算符 **like** 来表达模式。考虑查询"找出所在建筑名称中包含子串'Watson'的所有系名"，该查询的写法如下：

```
select dept_name
from department
where building like '%Watson%';
```

为使模式中能够包含特殊模式的字符(即%和_)，SQL 允许定义转义字符。转义字符直接放在特殊字符的前面，表示该特殊字符被当成普通字符。我们在 **like** 比较运算中使用 **escape** 关键词来定义转义字符。为了说明这一用法，考虑以下模式，它使用反斜线(\)作为转义字符：

- **like** 'ab\%cd%' **escape** '\' 匹配所有以"ab%cd"开头的字符串。

• **like** 'ab\%cd%' **escape** '\' 匹配所有以"ab%cd"开头的字符串。

SQL 允许使用 **not like** 比较运算符搜寻不匹配项。一些数据库还提供 **ilike** 运算的变体，不区分大小写。

在 SQL:1999 中还提供 **similar to** 操作，它具备比 **like** 运算更强大的模式匹配能力。它的模式定义语法类似于 UNIX 中的正则表达式。

3.4.3 select 子句中的属性说明

星号"*"可以用在 **select** 子句中表示"所有的属性"，因而，如下查询的 **select** 子句中使用 *instructor.**：

```
select instructor.*
from instructor, teaches
where instructor.ID = teaches.ID;
```

表示 *instructor* 中的所有属性都被选中。形如 **select** * 的 select 子句表示 **from** 子句结果关系的所有属性都被选中。

3.4.4 排列元组的显示次序

SQL 为用户提供了一些对关系中元组显示次序的控制。**order by** 子句就可以让查询结果中元组按排列顺序显示。为了按字母顺序列出在 Physics 系的所有教师，我们可以这样写：

```
select name
from instructor
where dept_name = 'Physics'
order by name;
```

order by 子句默认使用升序。要说明排序顺序，我们可以用 **desc** 表示降序，或者用 **asc** 表示升序。此外，排序可在多个属性上进行。假设我们希望按 *salary* 的降序列出整个 *instructor* 关系。如果有几位教师的工资相同，就将它们按姓名升序排列。我们用 SQL 将该查询表示如下：

77

```
select *
from instructor
order by salary desc, name asc;
```

3.4.5 where 子句谓词

为了简化 **where** 子句，SQL 提供 **between** 比较运算符来说明一个值是小于或等于某个值，同时大于或等于另一个值的。如果我们想找出工资在 90 000 美元和 100 000 美元之间的教师的姓名，我们可以使用 **between** 比较运算符，如下所示：

```
select name
from instructor
where salary between 90000 and 100000;
```

它可以取代

```
select name
from instructor
where salary <= 100000 and salary >= 90000;
```

类似地，我们还可以使用 **not between** 比较运算符。

我们可以扩展前面看到过的查找教师名以及课程标识的查询，但考虑更复杂的情况，要求教师是生物系的："查找 Biology 系讲授了课程的所有教师的姓名和他们所讲授的课程"。为了写出这样的查询，我们可以在前面看到过的两个 SQL 查询的任意一个的基础上进行修改。在 **where** 子句中增加一个额外的条件。我们下面给出修改后的不使用自然连接的 SQL 查询形式：

```
select name, course_id
from instructor, teaches
where instructor.ID = teaches.ID and dept_name = 'Biology';
```

SQL 允许我们用记号 $(v_1, v_2, \cdots, v_n)$ 来表示一个分量值分别为 $v_1, v_2, \cdots, v_n$ 的 n 维元组。在元组

上可以运用比较运算符，按字典顺序进行比较运算。例如，$(a_1, a_2) <= (b_1, b_2)$ 在 $a_1 <= b_1$ 且 $a_2 <= b_2$ 时为真。类似地，当两个元组在所有属性上相等时，它们是相等的。这样，前述查询可被重写为如下形式：[⊖]

78

```
select name, course_id
from instructor, teaches
where (instructor.ID, dept_name) = (teaches.ID, 'Biology');
```

3.5 集合运算

SQL 作用在关系上的 **union**、**intersect** 和 **except** 运算对应于数学集合论中的 ∪、∩ 和 − 运算。我们现在来构造包含在两个集合上使用 **union**、**intersect** 和 **except** 运算的查询。

- 在 2009 年秋季学期开设的所有课程的集合：

```
select course_id
from section
where semester = 'Fall' and year = 2009;
```

- 在 2010 年春季学期开设的所有课程的集合：

```
select course_id
from section
where semester = 'Spring' and year = 2010;
```

在我们后面的讨论中，将用 $c1$ 和 $c2$ 分别指代包含以上查询结果的两个关系，并在图 3-9 和图 3-10 中给出作用在如图 2-6 所示的 *section* 关系上的查询结果。注意 $c2$ 包含两个对应于 *course_id* 为 CS-319 的元组，因为该课程有两个课程段在 2010 年春季开课。

course_id
CS-101
CS-347
PHY-101

图 3-9 $c1$ 关系，列出 2009 年秋季开设的课程

course_id
CS-101
CS-315
CS-319
CS-319
FIN-201
HIS-351
MU-199

图 3-10 $c2$ 关系，列出 2010 年春季开设的课程

3.5.1 并运算

为了找出在 2009 年秋季开课，或者在 2010 年春季开课或两个学期都开课的所有课程，我们可写查询语句：[⊜]

```
(select course_id
 from section
 where semester = 'Fall' and year = 2009)
 union
(select course_id
 from section
 where semester = 'Spring' and year = 2010);
```

与 **select** 子句不同，**union** 运算自动去除重复。这样，在如图 2-6 所示的 *section* 关系中，2010 年春季开设 CS-319 的两个课程段，CS-101 在 2009 年秋季和 2010 年秋季学期各开设一个课程段，CS-101 和 CS-319 在结果中只出现一次，如图 3-11 所示。

course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

图 3-11 $c1$ **union** $c2$ 的结果

⊖ 尽管这是 SQL-92 标准的一部分，但某些 SQL 实现中可能不支持这种形式。

⊜ 我们在每条 **select-from-where** 语句上使用的括号是可省略的，但易于阅读。

如果我们想保留所有重复，就必须用 **union all** 代替 **union**：

```
(select course_id
 from section
 where semester = 'Fall' and year = 2009)
 union all
(select course_id
 from section
 where semester = 'Spring' and year = 2010);
```

在结果中的重复元组数等于在 c1 和 c2 中出现的重复元组数的和。因此在上述查询中，每个 CS-319 和 CS-101 都将被列出两次。作为一个更深入的例子，如果存在这样一种情况：ECE-101 在 2009 年秋季学期开设 4 个课程段，在 2010 年春季学期开设 2 个课程段，那么在结果中将有 6 个 ECE-101 元组。

3.5.2 交运算

为了找出在 2009 年秋季和 2010 年春季同时开课的所有课程的集合，我们可写出：

```
(select course_id
 from section
 where semester = 'Fall' and year = 2009)
intersect
(select course_id
 from section
 where semester = 'Spring' and year = 2010);
```

结果关系如图 3-12 所示，它只包括一个 CS-101 元组。**intersect** 运算自动去除重复。例如，如果存在这样的情况：ECE-101 在 2009 年秋季学期开设 4 个课程段，在 2010 年春季学期开设 2 个课程段，那么在结果中只有 1 个 ECE-101 元组。

course_id
CS-101

图 3-12 c1 intersect c2 的结果

如果我们想保留所有重复，就必须用 **intersect all** 代替 **intersect**：

```
(select course_id
 from section
 where semester = 'Fall' and year = 2009)
intersect all
(select course_id
 from section
 where semester = 'Spring' and year = 2010);
```

在结果中出现的重复元组数等于在 c1 和 c2 中出现的重复次数中最少的那个。例如，如果 ECE-101 在 2009 年秋季学期开设 4 个课程段，在 2010 年春季学期开设 2 个课程段，那么在结果中有 2 个 ECE-101 元组。

3.5.3 差运算

为了找出在 2009 年秋季学期开课但不在 2010 年春季学期开课的所有课程，我们可写出：

```
(select course_id
 from section
 where semester = 'Fall' and year = 2009)
except
(select course_id
 from section
 where semester = 'Spring' and year = 2010);
```

该查询结果如图 3-13 所示。注意这正好是图 3-9 的 c1 关系减去不出现的 CS-101 元组。**except** 运算[⊖]从其第一个输入中输出所有不出现在第二个输入中的元组，也即它执行集差操作。此运算在执行集差操作之前自动去除输入中的重复。例如，如果 ECE-101 在 2009 年秋季学期开设 4 个课程段，在 2010 年

⊖ 某些 SQL 实现，特别是 Oracle，使用关键词 **minus** 代替 **except**。

春季学期开设2个课程段，那么在 **except** 运算的结果中将没有 ECE-101 的任何拷贝。

如果我们想保留所有重复，就必须用 **except all** 代替 **except**：

```
(select course_id
 from section
 where semester = 'Fall' and year = 2009)
except all
(select course_id
 from section
 where semester = 'Spring' and year = 2010);
```

course_id
CS-347
ECE-101

图3-13 c1 except c2 的结果

结果中的重复元组数等于在 c1 中出现的重复元组数减去在 c2 中出现的重复元组数（前提是此差为正）。因此，如果 ECE-101 在2009年秋季学期开设4个课程段，在2010年春季学期开设2个课程段，那么在结果中有2个 ECE-101 元组。然而，如果 ECE-101 在2009年秋季学期开设2个或更少的课程段，在2010年春季学期开设2个课程段，那么在结果中将不存在 ECE-101 元组。

3.6 空值

空值给关系运算带来了特殊的问题，包括算术运算、比较运算和集合运算。

如果算术表达式的任一输入为空，则该算术表达式（涉及诸如 +、-、* 或 /）结果为空。例如，如果查询中有一个表达式是 $r.A + 5$，并且对于某个特定的元组，$r.A$ 为空，那么对此元组来说，该表达式的结果也为空。

涉及空值的比较问题更多。例如，考虑比较运算"1 < **null**"。因为我们不知道空值代表的是什么，所以说上述比较为真可能是错误的。但是说上述比较为假也可能是错误的。如果我们认为比较为假，那么"**not** (1 < **null**)"就应该为真，但这是没有意义的。因而 SQL 将涉及空值的任何比较运算的结果视为 **unknown**（既不是谓词 **is null**，也不是 **is not null**，我们在本节的后面介绍这两个谓词）。这创建了除 *true* 和 *false* 之外的第三个逻辑值。

由于在 **where** 子句的谓词中可以对比较结果使用诸如 **and**、**or** 和 **not** 的布尔运算，所以这些布尔运算的定义也被扩展到可以处理 **unknown** 值。

- **and**：*true* **and** *unknown* 的结果是 *unknown*，*false* **and** *unknown* 结果是 *false*，*unknown* **and** *unknown* 的结果是 *unknown*。
- **or**：*true* **or** *unknown* 的结果是 *true*，*false* **or** *unknown* 结果是 *unknown*，*unknown* **or** *unknown* 结果是 *unknown*。
- **not**：**not** *unknown* 的结果是 *unknown*。

可以验证，如果 $r.A$ 为空，那么"1 < $r.A$"和"**not** (1 < $r.A$)"结果都是 unknown。

如果 **where** 子句谓词对一个元组计算出 **false** 或 **unknown**，那么该元组不能被加入到结果集中。

79
|
83

SQL 在谓词中使用特殊的关键词 **null** 测试空值。因而为找出 *instructor* 关系中 *salary* 为空值的所有教师，我们可以写成：

```
select name
from instructor
where salary is null;
```

如果谓词 **is not null** 所作用的值非空，那么它为真。

某些 SQL 实现还允许我们使用子句 **is unknown** 和 **is not unknown** 来测试一个表达式的结果是否为 unknown，而不是 true 或 false。

当一个查询使用 **select distinct** 子句时，重复元组将被去除。为了达到这个目的，当比较两个元组对应的属性值时，如果这两个值都是非空并且值相等，或者都是空，那么它们是相同的。所以诸如 {('A', null), ('A', null)} 这样的两个元组将被认为是相同的，即使在某些属性上存在空值。使用 **distinct** 子句会保留这样的相同元组的一份拷贝。注意上述对待空值的方式与谓词中对待空值的方式是不同的，在谓词中"null = null"会返回 unknown，而不是 true。

如果元组在所有属性上的取值相等，那么它们就被当作相同元组，即使某些值为空。上述方式还

应用于集合的并、交和差运算。

3.7 聚集函数

聚集函数是以值的一个集合（集或多重集）为输入、返回单个值的函数。SQL 提供了五个固有聚集函数：

- 平均值：**avg**。
- 最小值：**min**。
- 最大值：**max**。
- 总和：**sum**。
- 计数：**count**。

84 **sum** 和 **avg** 的输入必须是数字集，但其他运算符还可作用在非数字数据类型的集合上，如字符串。

3.7.1 基本聚集

考虑查询"找出 Computer Science 系教师的平均工资"。我们书写该查询如下：

```
select avg (salary)
from instructor
where dept_name = 'Comp. Sci.';
```

该查询的结果是一个具有单属性的关系，其中只包含一个元组，这个元组的数值对应 Computer Science 系教师的平均工资。数据库系统可以给结果关系的属性一个任意的名字，该属性是由聚集产生的。然而，我们可以用 **as** 子句给属性赋个有意义的名称，如下所示：

```
select avg (salary) as avg_salary
from instructor
where dept_name = 'Comp. Sci.';
```

在图 2-1 的 *instructor* 关系中，Computer Science 系的 4 个值是 75 000 美元、65 000 美元和 92 000 美元，平均工资是 232 000/3 = 77 333.33 美元。

在计算平均值时保留重复元组是很重要的。假设 Computer Science 系增加了第四位教师，其工资正好是 75 000 美元。如果去除重复的话，我们会得到错误的答案（232 000/4 = 58 000 美元），而正确的答案是 76 750 美元。

有些情况下在计算聚集函数前需先剔掉重复元组。如果我们确实想剔除重复元组，可在聚集表达式中使用关键词 **distinct**。比方有这样一个查询示例"找出在 2010 年春季学期讲授一门课程的教师总数"，在该例中不论一个教师讲授了几个课程段，他只应被计算一次。所需信息包含在 *teaches* 关系中，我们书写该查询如下：

```
select count (distinct ID)
from teaches
where semester = 'Spring' and year = 2010;
```

我们经常使用聚集函数 **count** 计算一个关系中元组的个数。SQL 中该函数的写法是 **count(*)**。因此，要找出 *course* 关系中的元组数，可写成：

```
select count(*)
from course;
```

85 由于在 *ID* 前面有关键字 **distinct**，所以即使某位教师教了不止一门课程，在结果中他也仅被计数一次。

SQL 不允许在用 **count(*)** 时使用 **distinct**。在用 **max** 和 **min** 时使用 **distinct** 是合法的，尽管结果并无差别。我们可以使用关键词 **all** 替代 **distinct** 来说明保留重复元组，但是，既然 **all** 是默认的，就没必要这么做了。

3.7.2 分组聚集

有时候我们不仅希望将聚集函数作用在单个元组集上，而且也希望将其作用到一组元组集上；在 SQL 中可用 **group by** 子句实现这个愿望。**group by** 子句中给出的一个或多个属性是用来构造分组的。

在 **group by** 子句中的所有属性上取值相同的元组将被分在一个组中。

作为示例，考虑查询"找出每个系的平均工资"，该查询书写如下：

```
select dept_name, avg(salary) as avg_salary
from instructor
group by dept_name;
```

图 3-14 给出了 *instructor* 关系中的元组按照 *dept_name* 属性进行分组的情况。分组是计算查询结果的第一步，在每个分组上都要进行指定的聚集计算，查询结果如图 3-15 所示。

相反，考虑查询"找出所有教师的平均工资"。我们把此查询写成如下形式：

```
select avg (salary)
from instructor;
```

在这里省略了 **group by** 子句，因此整个关系被当作是一个分组。

ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

图 3-14 *instructor* 关系的元组按照 *dept_name* 属性分组

86

作为在元组分组上进行聚集操作的另一个例子，考虑查询"找出每个系在 2010 年春季学期讲授一门课程的教师人数"。有关每位教师在每个学期讲授每个课程段的信息在 *teaches* 关系中。但是，这些信息需要与来自 *instructor* 关系的信息进行连接，才能够得到每位教师所在的系名。这样，我们把此查询写成如下形式：

```
select dept_name, count (distinct ID) as instr_count
from instructor natural join teaches
where semester = 'Spring' and year = 2010
group by dept_name;
```

其结果如图 3-16 所示。

dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

图 3-15 查询"找出每个系的平均工资"的结果关系

dept_name	*instr_count*
Comp. Sci.	3
Finance	1
History	1
Music	1

图 3-16 查询"找出每个系在 2010 年春季学期讲授一门课程的教师人数"的结果关系

当 SQL 查询使用分组时，一个很重要的事情是需要保证出现在 **select** 语句中但没有被聚集的属性只能是出现在 **group by** 子句中的那些属性。换句话说，任何没有出现在 **group by** 子句中的属性如果出现在 **select** 子句中的话，它只能出现在聚集函数内部，否则这样的查询就是错误的。例如，下述查询是错误的，因为 *ID* 没有出现在 **group by** 子句中，但它出现在了 **select** 子句中，而且没有被聚集：

```
/* 错误查询 */
select dept_name, ID, avg (salary)
from instructor
group by dept_name;
```

在一个特定分组（通过 *dept_name* 定义）中的每位教师都有一个不同的 *ID*，既然每个分组只输出一个元组，那就无法确定选哪个 *ID* 值作为输出。其结果是，SQL 不允许这样的情况出现。

3.7.3 having 子句

有时候，对分组限定条件比对元组限定条件更有用。例如，我们也许只对教师平均工资超过 42 000美元的系感兴趣。该条件并不针对单个元组，而是针对 **group by** 子句构成的分组。为表达这样的查询，我们使用 SQL 的 **having** 子句。**having** 子句中的谓词在形成分组后才起作用，因此可以使用聚集函数。我们用 SQL 表达该查询如下：

```
select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name
having avg (salary) > 42000;
```

其结果如图 3-17 所示。

dept_name	avg_salary
Physics	91000
Elec. Eng.	80000
Finance	85000
Comp. Sci.	77333
Biology	72000
History	61000

图 3-17 查询"找出系平均工资超过 42 000 美元的那些系中教师的平均工资"的结果关系

与 **select** 子句的情况类似，任何出现在 **having** 子句中，但没有被聚集的属性必须只现在 **group by** 子句中，否则查询就被当成是错误的。

包含聚集、**group by** 或 **having** 子句的查询的含义可通过下述操作序列来定义：

1. 与不带聚集的查询情况类似，最先根据 **from** 子句来计算出一个关系。

2. 如果出现了 **where** 子句，**where** 子句中的谓词将应用到 **from** 子句的结果关系上。

3. 如果出现了 **group by** 子句，满足 **where** 谓词的元组再通过 **group by** 子句形成分组。如果没有 **group by** 子句，满足 **where** 谓词的整个元组集被当作一个分组。

4. 如果出现了 **having** 子句，它将应用到每个分组上；不满足 **having** 子句谓词的分组将被抛弃。

5. **select** 子句利用剩下的分组产生出查询结果中的元组，即在每个分组上应用聚集函数来得到单个结果元组。

为了说明在同一个查询中同时使用 **having** 子句和 **where** 子句的情况，我们考虑查询"对于在 2009 年讲授的每个课程段，如果该课程段有至少 2 名学生选课，找出选修该课程段的所有学生的总学分（*tot_cred*）的平均值"。

```
select course_id, semester, year, sec_id, avg (tot_cred)
from takes natural join student
where year = 2009
group by course_id, semester, year, sec_id
having count ( ID ) >= 2;
```

注意上述查询需要的所有信息来自关系 *takes* 和 *student*，尽管此查询是关于课程段的，却并不需要与 *section* 进行连接。

3.7.4 对空值和布尔值的聚集

空值的存在给聚集运算的处理带来了麻烦。例如，假设 *instructor* 关系中有些元组在 *salary* 上取空值。考虑以下计算所有工资总额的查询：

```
select sum(salary)
from instructor;
```

由于一些元组在 *salary* 上取空值，上述查询待求和的值中就包含了空值。SQL 标准并不认为总和本身为 *null*，而是认为 **sum** 运算符应忽略输入中的 *null* 值。

总而言之，聚集函数根据以下原则处理空值：除了 **count**(*) 外所有的聚集函数都忽略输入集合中的空值。由于空值被忽略，有可能造成参加函数运算的输入值集合为空集。规定空集的 count 运算值为 0，其他所有聚集运算在输入为空集的情况下返回一个空值。在一些更复杂的 SQL 结构中空值的影响会更难以琢磨。

87
|
89

在 SQL:1999 中引入了布尔（boolean）数据类型，它可以取 **true**、**false**、**unknown** 三个值。有两个聚集函数：**some** 和 **every**，其含义正如直观意义一样，可用来处理布尔（boolean）值的集合。

3.8 嵌套子查询

SQL 提供嵌套子查询机制。子查询是嵌套在另一个查询中的 **select-from-where** 表达式。子查询嵌套在 **where** 子句中，通常用于对集合的成员资格、集合的比较以及集合的基数进行检查。从 3.8.1 到 3.8.4 节我们学习在 **where** 子句中嵌套子查询的用法。在 3.8.5 节我们学习在 **from** 子句中嵌套的子查询。在 3.8.7 节我们将看到一类被称作标量子查询的子查询是如何只现在一个表达式所返回的单个值可以只现的任何地方的。

3.8.1 集合成员资格

SQL允许测试元组在关系中的成员资格。连接词 **in** 测试元组是否是集合中的成员，集合是由 **select** 子句产生的一组值构成的。连接词 **not in** 则测试元组是否不是集合中的成员。

作为示例，考虑查询"找出在2009年秋季和2010年春季学期同时开课的所有课程"。先前，我们通过对两个集合进行交运算来书写该查询，这两个集合分别是：2009年秋季开课的课程集合与2010年春季开课的课程集合。现在我们采用另一种方式，查找在2009年秋季开课的所有课程，看它们是否也是2010年春季开课的课程集合中的成员。很明显，这种方式得到的结果与前面相同。但我们可以用SQL中的 **in** 连接词书写该查询。我们从找出2010年春季开课的所有课程开始，写出子查询：

```
(select course_id
 from section
 where semester = 'Spring' and year = 2010)
```

然后我们需要从子查询形成的课程集合中找出那些在2009年秋季开课的课程。为完成此项任务可将子查询嵌入外部查询的 **where** 子句中。最后的查询语句是：

```
select distinct course_id
from section
where semester = 'Fall' and year = 2009 and
    course_id in (select course_id
                  from section
                  where semester = 'Spring' and year = 2010);
```

该例说明了在SQL中可以用多种方法书写同一查询。这种灵活性是有好处的，因为它允许用户用最接近自然的方法去思考查询。我们将看到在SQL中有许多这样的冗余。

我们以与 **in** 结构类似的方式使用 **not in** 结构。例如，为了找出所有在2009年秋季学期开课、但不在2010年春季学期开课的课程，我们可写出：

```
select distinct course_id
from section
where semester = 'Fall' and year = 2009 and
    course_id not in (select course_id
                      from section
                      where semester = 'Spring' and year = 2010);
```

in 和 **not in** 操作符也能用于枚举集合。下面的查询找出既不叫"Mozart"，也不叫"Einstein"的教师的姓名：

```
select distinct name
from instructor
where name not in ('Mozart', 'Einstein');
```

在前面的例子中，我们是在单属性关系中测试成员资格。在SQL中测试任意关系的成员资格也是可以的。例如，我们可以这样来表达查询"找出(不同的)学生总数，他们选修了 *ID* 为10101的教师所讲授的课程段"：

```
select count (distinct ID)
from takes
where (course_id, sec_id, semester, year) in (select course_id, sec_id, semester, year
                                              from teaches
                                              where teaches.ID = 10101);
```

3.8.2 集合的比较

作为一个说明该集合查询能够对集合进行比较的例子，考虑查询"找出满足下述条件的所有教师的姓名，他们的工资至少比Biology系某一个教师的工资要高"。在3.4.1节，我们将此查询写作：

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept_name = 'Biology';
```

91

但是 SQL 提供另外一种方式书写上面的查询。短语"至少比某一个要大"在 SQL 中用 > some 表示。此结构允许我们用一种更靠近此查询的文字表达的形式来写上面的查询：

```
select name
from instructor
where salary > some (select salary
                     from instructor
                     where dept_name = 'Biology');
```

子查询

```
(select salary
 from instructor
 where dept_name = 'Biology')
```

产生 Biology 系所有教师的所有工资值的集合。当元组的 *salary* 值至少比 Biology 系教师的所有工资值集合中某一成员高时，外层 **select** 的 **where** 子句中 > **some** 的比较为真。

SQL 也允许 < **some**，<= **some**，>= **some**，= **some** 和 <> **some** 的比较。作为练习，请验证 = **some** 等价于 **in**，然而 <> **some** 并不等价于 **not in**。[⊖]

现在我们来修改一下我们的查询。找出满足下面条件的所有教师的姓名，他们的工资值比 Biology 系每个教师的工资都高。结构 > **all** 对应于词组"比所有的都大"。使用该结构，我们写出查询如下：

```
select name
from instructor
where salary > all (select salary
                    from instructor
                    where dept_name = 'Biology');
```

类似于 **some**，SQL 也允许 < **all**，<= **all**，>= **all**，= **all** 和 <> **all** 的比较。作为练习，请验证 <> **all** 等价于 **not in**，但 = **all** 并不等价于 **in**。

作为集合比较的另一个例子，考虑查询"找出平均工资最高的系"。我们首先写一个查询来找出每个系的平均工资，然后把它作为子查询嵌套在一个更大的查询中，以找出那些平均工资大于等于所有系平均工资的系。

```
select dept_name
from instructor
group by dept_name
having avg (salary) >= all (select avg (salary)
                            from instructor
                            group by dept_name);
```

3.8.3 空关系测试

SQL 还有一个特性可测试一个子查询的结果中是否存在元组。**exists** 结构在作为参数的子查询非空时返回 **true** 值。使用 **exists** 结构，我们还能以另外一种方法书写查询"找出在 2009 年秋季学期和 2010 年春季学期同时开课的所有课程"：

```
select course_id
from section as S
where semester = 'Fall' and year = 2009 and
    exists (select *
```

⊖ 在 SQL 中关键词 **any** 对义于 **some**。早期 SQL 版本中仅允许使用 **any**，后来的版本为了避免和英语中 *any* 一词在语言上的混淆，又添加了另一个可选择的关键词 **some**。

```
    from section as T
    where semester = 'Spring' and year = 2010 and
        S.course_id = T.course_id);
```

上述查询还说明了SQL的一个特性，来自外层查询的一个相关名称（上述查询中的*S*）可以用在**where**子句的子查询中。使用了来自外层查询相关名称的子查询被称作相关子查询（correlated subquery）。

在包含了子查询的查询中，在相关名称上可以应用作用域规则。根据此规则，在一个子查询中只能使用此子查询本身定义的，或者在包含此子查询的任何查询中定义的相关名称。如果一个相关名称既在子查询中定义，又在包含该子查询的查询中定义，则子查询中的定义有效。这条规则类似于编程语言中通用的变量作用域规则。

我们可以用**not exists**结构测试子查询结果集中是否不存在元组。我们可以使用**not exists**结构模拟集合包含（即超集）操作：我们可将"关系*A*包含关系*B*"写成"**not exists**(*B* **except** *A*)"。（尽管**contains**运算符并不是当前SQL标准的一部分，但这一运算符曾出现在某些早期的关系系统中。）为了说明**not exists**操作符，考虑查询"找出选修了Biology系开设的所有课程的学生"。使用**except**结构，我们可以书写此查询如下：

92
~
93

```
select S.ID, S.name
from student as S
where not exists ((select course_id
                   from course
                   where dept_name = 'Biology')
                  except
                  (select T.course_id
                   from takes as T
                   where S.ID = T.ID));
```

这里，子查询

```
(select course_id
 from course
 where dept_name = 'Biology')
```

找出Biology系开设的所有课程集合。子查询

```
(select T.course_id
 from takes as T
 where S.ID = T.ID)
```

找出*S.ID*选修的所有课程。这样，外层**select**对每个学生测试其选修的所有课程集合是否包含Biology系开设的所有课程集合。

3.8.4 重复元组存在性测试

SQL提供一个布尔函数，用于测试在一个子查询的结果中是否存在重复元组。如果作为参数的子查询结果中没有重复的元组，**unique**结构[⊖]将返回**true**值。我们可以用**unique**结构书写查询"找出所有在2009年最多开设一次的课程"，如下所示：

```
select T.course_id
from course as T
where unique (select R.course_id
              from section as R
              where T.course_id = R.course_id and
                  R.year = 2009);
```

⊖ 此结构并未被广泛实现。

注意如果某门课程不在 2009 年开设，那么子查询会返回一个空的结果，**unique** 谓词在空集上计算出真值。

在不使用 **unique** 结构的情况下，上述查询的一种等价表达方式是：

```
select T.course_id
from course as T
where 1 <= (select count(R.course_id)
            from section as R
            where T.course_id = R.course_id and
                R.year = 2009);
```

我们可以用 **not unique** 结构测试在一个子查询结果中是否存在重复元组。为了说明这一结构，考虑查询"找出所有在 2009 年最少开设两次的课程"，如下所示：

```
select T.course_id
from course as T
where not unique (select R.course_id
                  from section as R
                  where T.course_id = R.course_id and
                      R.year = 2009);
```

形式化地，对一个关系的 **unique** 测试结果为假的定义是，当且仅当在关系中存在着两个元组 t_1 和 t_2，且 $t_1 = t_2$。由于在 t_1 或 t_2 的某个域为空时，判断 $t_1 = t_2$ 为假，所以尽管一个元组有多个副本，只要该元组有一个属性为空，**unique** 测试就有可能为真。

3.8.5 from 子句中的子查询

SQL 允许在 **from** 子句中使用子查询表达式。在此采用的主要观点是：任何 **select-from-where** 表达式返回的结果都是关系，因而可以被插入到另一个 **select-from-where** 中任何关系可以出现的位置。

考虑查询"找出系平均工资超过 42 000 美元的那些系中教师的平均工资"。在 3.7 节我们使用了 **having** 子句来书写此查询。现在我们可以不用 **having** 子句来重写这个查询，而是通过如下这种在 **from** 子句中使用子查询的方式：

```
select dept_name, avg_salary
from (select dept_name, avg (salary) as avg_salary
      from instructor
      group by dept_name)
where avg_salary > 42000;
```

94
95

该子查询产生的关系包含所有系的名字和相应的教师平均工资。子查询的结果属性可以在外层查询中使用，正如上例所示。

注意我们不需要使用 **having** 子句，因为 **from** 子句中的子查询计算出了每个系的平均工资，早先在 **having** 子句中使用的谓词现在出现在外层查询的 **where** 子句中。

我们可以用 **as** 子句给此子查询的结果关系起个名字，并对属性进行重命名。如下所示：

```
select dept_name, avg_salary
from (select dept_name, avg (salary)
      from instructor
      group by dept_name)
      as dept_avg (dept_name, avg_salary)
where avg_salary > 42000;
```

子查询的结果关系被命名为 *dept_avg*，其属性名是 *dept_name* 和 *avg_salary*。

很多（但并非全部）SQL 实现都支持在 **from** 子句中嵌套子查询。请注意，某些 SQL 实现要求对每一个子查询结果关系都给一个名字，即使该名字从不被引用；Oracle 允许对子查询结果关系命名（省略掉关键字 **as**），但必不允许对关系中的属性重命名。

作为另一个例子，假设我们想要找出在所有系中工资总额最大的系。在此 **having** 子句是无能为力的，但我们可以用 **from** 子句中的子查询轻易地写出如下查询：

```
select max (tot_salary)
from (select dept_name, sum(salary)
      from instructor
      group by dept_name) as dept_total (dept_name, tot_salary);
```

我们注意到在 **from** 子句嵌套的子查询中不能使用来自 **from** 子句其他关系的相关变量。然而 SQL-2003 允许 **from** 子句中的子查询用关键词 **lateral** 作为前缀，以使访问 **from** 子句中在它前面的表或子查询中的属性。例如，如果我们想打印每位教师的姓名，以及他们的工资和所在系的平均工资。可书写查询如下：

```
select name, salary, avg_salary
from instructor I1, lateral (select avg(salary) as avg_salary
                             from instructor I2
                             where I2.dept_name = I1.dept_name);
```

没有 **lateral** 子句的话，子查询就不能访问来自外层查询的相关变量 *I1*。目前只有少数 SQL 实现支持 **lateral** 子句，比如 IBM DB2。 [96]

3.8.6 with 子句

with 子句提供定义临时关系的方法，这个定义只对包含 **with** 子句的查询有效。考虑下面的查询，它找出具有最大预算值的系。

```
with max_budget (value) as
    (select max(budget)
     from department)
select budget
from department, max_budget
where department.budget = max_budget.value;
```

with 子句定义了临时关系 *max_budget*，此关系在随后的查询中马上被使用了。**with** 子句是在 SQL:1999 中引入的，目前有许多（但并非所有）数据库系统都提供了支持。

我们也能用 **from** 子句或 **where** 子句中的嵌套子查询书写上述查询。但是，用嵌套子查询会使得查询语句晦涩难懂。**with** 子句使查询在逻辑上更加清晰，它还允许在一个查询内的多个地方使用视图定义。

例如，假设我们要查出所有工资总额大于所有系平均工资总额的系，我们可以利用如下 **with** 子句写出查询：

```
with dept_total (dept_name, value) as
    (select dept_name, sum(salary)
     from instructor
     group by dept_name),
     dept_total_avg(value) as
    (select avg(value)
     from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value >= dept_total_avg.value;
```

我们当然也可以不用 **with** 子句来建立等价的查询，但是那样会复杂很多，而且也不易看懂。作为练习，你可以把它转化为不用 **with** 子句的等价查询。

3.8.7 标量子查询

SQL 允许子查询出现在返回单个值的表达式能够出现的任何地方，只要该子查询只返回包含单个属性的单个元组；这样的子查询称为**标量子查询**（scalar subquery）。例如，一个子查询可以用到下面例子的 **select** 子句中，这个例子列出所有的系以及它们拥有的教师数： [97]

```
select dept_name,
    (select count(*)
     from instructor
     where department.dept_name = instructor.dept_name)
     as num_instructors
from department;
```

上面例子中的子查询保证只返回单个值，因为它使用了不带 **group by** 的 **count**(*) 聚集函数。此例也说明了对相关变量的使用，即使用在外层查询的 **from** 子句中关系的属性，例如上例中的 *department.dept_name*。

标量子查询可以出现在 **select**、**where** 和 **having** 子句中。也可以不使用聚集函数来定义标量子查询。在编译时并非总能判断一个子查询返回的结果中是否有多个元组，如果在子查询被执行后其结果中有不止一个元组，则产生一个运行时错误。

注意从技术上讲标量子查询的结果类型仍然是关系，尽管其中只包含单个元组。然而，当在表达式中使用标量子查询时，它出现的位置是单个值出现的地方，SQL 就从该关系中包含单属性的单元组中取出相应的值，并返回该值。

3.9 数据库的修改

目前为止我们的注意力集中在对数据库的信息抽取上。现在我们将展示如何用 SQL 来增加、删除和修改信息。

3.9.1 删除

删除请求的表达与查询非常类似。我们只能删除整个元组，而不能只删除某些属性上的值。SQL 用如下语句表示删除：

```
delete from r
where P;
```

其中 P 代表一个谓词，r 代表一个关系。**delete** 语句首先从 r 中找出所有使 $P(t)$ 为真的元组 t，然后把它们从 r 中删除。如果省略 **where** 子句，则 r 中所有元组将被删除。

注意 **delete** 命令只能作用于一个关系。如果我们想从多个关系中删除元组，必须在每个关系上使用一条 **delete** 命令。**where** 子句中的谓词可以和 **select** 命令的 **where** 子句中的谓词一样复杂。在另一种极端情况下，**where** 子句可以为空，请求

98

```
delete from instructor;
```

将删除 *instructor* 关系中的所有元组，*instructor* 关系本身仍然存在，但它变成空的了。

下面是 SQL 删除请求的一些例子：

- 从 *instructor* 关系中删除与 Finance 系教师相关的所有元组：

```
delete from instructor
where dept_name = 'Finance';
```

- 删除所有工资在 13 000 美元到 15 000 美元之间的教师：

```
delete from instructor
where salary between 13000 and 15000;
```

- 从 *instructor* 关系中删除所有这样的教师元组，他们在位于 Watson 大楼的系工作：

```
delete from instructor
where dept_name in (select dept_name
                    from department
                    where building = 'Watson');
```

此 **delete** 请求首先找出所有位于 Watson 大楼的系，然后将属于这些系的 *instructor* 元组全部删除。注意，虽然我们一次只能从一个关系中删除元组，但是通过在 **delete** 的 **where** 子句中嵌套 **select-**

from-where，我们可以引用任意数目的关系。**delete** 请求可以包含嵌套的 **select**，该 **select** 引用待删除元组的关系。例如，假设我们想删除工资低于大学平均工资的教师记录，可以写出如下语句：

```
delete from instructor
where salary < (select avg (salary)
                from instructor);
```

该 **delete** 语句首先测试 *instructor* 关系中的每一个元组，检查其工资是否小于大学教师的平均工资。然后删除所有符合条件的元组，即所有低于平均工资的教师。在执行任何删除之前先进行所有元组的测试是至关重要的，因为若有些元组在其余元组未被测试前先被删除，则平均工资将会改变，这样 **delete** 的最后结果将依赖于元组被处理的顺序！ 99

3.9.2 插入

要往关系中插入数据，我们可以指定待插入的元组，或者写一条查询语句来生成待插入的元组集合。显然，待插入元组的属性值必须在相应属性的域中。同样，待插入元组的分量数也必须是正确的。

最简单的 **insert** 语句是单个元组的插入请求。假设我们想要插入的信息是 Computer Science 系开设的名为"Database Systems"的课程 CS-437，它有 4 个学分。我们可写成：

```
insert into course
    values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

在此例中，元组属性值的排列顺序和关系模式中属性排列的顺序一致。考虑到用户可能不记得关系属性的排列顺序，SQL 允许在 **insert** 语句中指定属性。例如，以下 SQL **insert** 语句与前述语句的功能相同。

```
insert into course (course_id, title, dept_name, credits)
    values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
insert into course (title, course_id, credits, dept_name)
    values ('Database Systems', 'CS-437', 4, 'Comp. Sci.');
```

更通常的情况是，我们可能想在查询结果的基础上插入元组。假设我们想让 Music 系每个修满 144 学分的学生成为 Music 系的教师，其工资为 18 000 美元。我们可写作：

```
insert into instructor
    select ID, name, dept_name, 18000
    from student
    where dept_name = 'Music' and tot_cred > 144;
```

和本节前面的例子不同的是，我们没有指定一个元组，而是用 **select** 给出一个元组集合。SQL 先执行这条 **select** 语句，求出将要插入到 *instructor* 关系中的元组集合。每个元组都有 *ID*、*name*、*dept_name*（Music）和工资（18 000 美元）。

在执行插入之前先执行完 **select** 语句是非常重要的。如果在执行 **select** 语句的同时执行插入动作，如果在 *student* 上没有主码约束的话，像 100

```
insert into student
    select *
    from student;
```

这样的请求就可能会插入无数元组。如果没有主码约束，上述请求会重新插入 *student* 中的第一个元组，产生该元组的第二份拷贝。由于这个副本现在是 *student* 中的一部分，**select** 语句可能找到它，于是第三份拷贝被插入到 *student* 中。第三份拷贝又可能被 **select** 语句发现，于是又插入第四份拷贝，如此等等，无限循环。在执行插入之前先完成 **select** 语句的执行可以避免这样的问题。这样，如果在 *student* 关系上没有主码约束，那么上述 **insert** 语句就只是把 *student* 关系中的每个元组都复制一遍。

在讨论 **insert** 语句时我们只考虑了这样的例子：待插入元组的每个属性都被赋了值。但是有可能待插入元组中只给出了模式中部分属性的值，那么其余属性将被赋空值，用 *null* 表示。考虑语求：

```
insert into student
    values ('3003', 'Green', 'Finance', null);
```

此请求所插入的元组代表了一个在 Finance 系、*ID* 为"3003"的学生，但其 *tot_cred* 值是未知的。考虑查询：

```
select ID
from student
where tot_cred > 45;
```

既然"3003"号学生的 *tot_cred* 值未知，我们不能确定它是否大于 45。

大部分关系数据库产品有特殊的"bulk loader"工具，它可以向关系中插入一个非常大的元组集合。这些工具允许从格式化文本文件中读出数据，且执行速度比同等目的的插入语句序列要快得多。

3.9.3 更新

有些情况下，我们可能希望在不改变整个元组的情况下改变其部分属性的值。为达到这一目的，可以使用 **update** 语句。与使用 **insert**、**delete** 类似，待更新的元组可以用查询语句找到。

101 假设要运行年度工资增长，所有教师的工资将增长 5%。我们写出：

```
update instructor
set salary = salary * 1.05;
```

上面的更新语句将在 *instructor* 关系的每个元组上执行一次。

如果只给那些工资低于 70 000 美元的教师涨工资，我们可以这样写：

```
update instructor
set salary = salary * 1.05
where salary < 70000;
```

总之，**update** 语句的 **where** 子句可以包含 **select** 语句的 **where** 子句中的任何合法结构（包括嵌套的 **select**）。和 **insert**、**delete** 类似，**update** 语句中嵌套的 **select** 可以引用待更新的关系。同样，SQL 首先检查关系中的所有元组，看它们是否应该被更新，然后才执行更新。例如，请求"对工资低于平均数的教师涨 5% 的工资"可以写为如下形式：

```
update instructor
set salary = salary * 1.05
where salary < (select avg (salary)
                from instructor);
```

我们现在假设给工资超过 100 000 美元的教师涨 3% 的工资，其余教师涨 5%。我们可以写两条 **update** 语句：

```
update instructor
set salary = salary * 1.03
where salary > 100000;

update instructor
set salary = salary * 1.05
where salary <= 100000;
```

注意这两条 **update** 语句的顺序十分重要。假如我们改变这两条语句的顺序，工资略少于 100 000 美元的教师将增长 8% 的工资。

SQL 提供 **case** 结构，我们可以利用它在一条 **update** 语句中执行前面的两种更新，避免更新次序引
102 发的问题：

```
update instructor
set salary = case
        when salary <= 100000 then salary * 1.05
        else salary * 1.03
    end
```

case 语句的一般格式如下：

```
case
    when pred1 then result1
    when pred2 then result2
    ...
    when predn then resultn
    else result0
end
```

当 i 是第一个满足的 $pred_1$，$pred_2 \cdots pred_n$ 时，此操作就会返回 $result_i$；如果没有一个谓词可以满足，则返回 $result_0$。case 语句可以用在任何应该出现值的地方。

标量子查询在 SQL 更新语句中也非常有用。它们可以用在 **set** 子句中。考虑这样一种更新：我们把每个 *student* 元组的 *tot_cred* 属性值设为该生成功学完的课程学分的总和。我们假设如果一个学生在某门课程上的成绩既不是'F'，也不是空，那么他成功学完了这门课程。我们需要使用 **set** 子句中的子查询来写出这种更新，如下所示：

```
update student S
set tot_cred = (
    select sum(credits)
    from takes natural join course
    where S.ID = takes.ID and
        takes.grade <> 'F' and
        takes.grade is not null);
```

注意子查询使用了来自 **update** 语句中的相关变量 S。如果一个学生没有成功学完任何课程，上述更新语句将把其 *tot_cred* 属性值设为空。如果想把这样的属性值设为 0 的话，我们可以使用另一条 **update** 语句来把空值替换为 0。更好的方案是把上述子查询中的"**select sum**(*credits*)"子句替换为如下使用 **case** 表达式的 **select** 子句：

```
select case
    when sum(credits) is not null then sum(credits)
    else 0
    end
```

103

3.10 总结

- SQL 是最有影响力的商用市场化的关系查询语言。SQL 语言包括几个部分：
  - □ **数据定义语言**(DDL)。它提供了定义关系模式、删除关系以及修改关系模式的命令。
  - □ **数据操纵语言**(DML)。它包括查询语言，以及往数据库中插入元组、从数据库中删除元组和修改数据库中元组的命令。
- SQL 的数据定义语言用于创建具有特定模式的关系。除了声明关系属性的名称和类型之外，SQL 还允许声明完整性约束，例如主码约束和外码约束。
- SQL 提供多种用于查询数据库的语言结构，其中包括 **select**、**from** 和 **where** 子句。SQL 支持自然连接操作。
- SQL 还提供了对属性和关系重命名，以及对查询结果按特定属性进行排序的机制。
- SQL 支持关系上的基本集合运算，包括并、交和差运算，它们分别对应于数学集合论中的 $\cup$、$\cap$ 和 $-$ 运算。
- SQL 通过在通用真值 true 和 false 外增加真值"unknown"，来处理对包含空值的关系的查询。
- SQL 支持聚集，可以把关系进行分组，在每个分组上单独运用聚集。SQL 还支持在分组上的集合运算。
- SQL 支持在外层查询的 **where** 和 **from** 子句中嵌套子查询。它还在一个表达式返回的单个值所允许出现的任何地方支持标量子查询。
- SQL 提供了用于更新、插入、删除信息的结构。

术语回顾

- 数据定义语言
- 数据操纵语言
- 数据库模式
- 数据库实例
- 关系模式
- 关系实例
- 主码
- 外码
  - 参照关系
  - 被参照关系
- 空值
- 查询语言
- SQL 查询结构
  - **select** 子句
  - **from** 子句
  - **where** 子句
- 自然连接运算
- **as** 子句
- **order by** 子句
- 相关名称(相关变量，元组变量)
- 集合运算
  - **union**
  - **intersect**
  - **except**
- 空值
  - 真值"unknown"
- 聚集函数
  - **avg**, **min**, **max**, **sum**, **count**
  - **group by**
  - **having**
- 嵌套子查询
- 集合比较
  - <, <=, >, >=, =, <>, **some**, **all**
  - **exists**
  - **unique**
- **lateral** 子句
- **with** 子句
- 标量子查询
- 数据库修改
  - 删除
  - 插入
  - 更新

实践习题

3.1 使用大学模式，用 SQL 写出如下查询。(建议在一个数据库上实际运行这些查询，使用我们在本书的 Web 网站 db-book.com 上提供的样本数据。上述网站还提供了如何建立一个数据库和加载样本数据的说明。)

a. 找出 Comp. Sci. 系开设的具有 3 个学分的课程名称。

b. 找出名叫 Einstein 的教师所教的所有学生的标识，保证结果中没有重复。

c. 找出教师的最高工资。

d. 找出工资最高的所有教师(可能有不止一位教师具有相同的工资)。

e. 找出 2009 年秋季开设的每个课程段的选课人数。

f. 从 2009 年秋季开设的所有课程段中，找出最多的选课人数。

g. 找出在 2009 年秋季拥有最多选课人数的课程段。

104
|
105

3.2 假设给你一个关系 *grade_points*(*grade*, *points*)，它提供从 *takes* 关系中用字母表示的成绩等级到数字表示的得分之间的转换。例如，"A"等级可指定为对应于 4 分，"A −"对应于 3.7 分，"B +"对应于 3.3 分，"B"对应于 3 分，等等。学生在某门课程(课程段)上所获得的等级分值被定义为该课程段的学分乘以该生得到的成绩等级所对应的数字表示的得分。

给定上述关系和我们的大学模式，用 SQL 写出下面的每个查询。为简单起见，可以假设没有任何 *takes* 元组在 *grade* 上取 *null* 值。

a. 根据 ID 为 12345 的学生所选修的所有课程，找出该生所获得的等级分值的总和。

b. 找出上述学生等级分值的平均值(GPA)，即用等级分值的总和除以相关课程学分的总和。

c. 找出每个学生的 ID 和等级分值的平均值。

3.3 使用大学模式，用 SQL 写出如下插入、删除和更新语句。

a. 给 Comp. Sci. 系的每位教师涨 10% 的工资。

b. 删除所有未开设过(即没有出现在 *section* 关系中)的课程。

c. 把每个在 *tot_cred* 属性上取值超过 100 的学生作为同系的教师插入，工资为 10 000 美元。

3.4 考虑图 3-18 中的保险公司数据库，其中加下划线的是主码。为这个关系数据库构造出如下 SQL 查询：

a. 找出 2009 年其车辆出过交通事故的人员总数。

b. 向数据库中增加一个新的事故，对每个必需的属性可以设定任意值。

c. 删除"John Smith"拥有的马自达车(Mazda)。

person (*driver_id*, *name*, *address*)
car (*license*, *model*, *year*)
accident (*report_number*, *date*, *location*)
owns (*driver_id*, *license*)
participated (*report_number*, *license*, *driver_id*, *damage_amount*)

图 3-18 习题 3.4 和习题 3.14 的保险公司数据库

3.5 假设有关系 *marks*(*ID*, *score*)，我们希望基于如下标准为学生评定等级：如果 *score* < 40 得 F；如果 40 ≤ *score* < 60 得 C；如果 60 ≤ *score* < 80 得 B；如果 80 ≤ *score* 得 A。写出 SQL 查询完成下列操作：

a. 基于 *marks* 关系显示每个学生的等级。

b. 找出各等级的学生数。

3.6 SQL 的 **like** 运算符是大小写敏感的，但字符串上的 lower( ) 函数可用来实现大小写不敏感的匹配。为了说明是怎么用的，写出这样一个查询：找出名称中包含了"sci"子串的系，忽略大小写。

3.7 考虑 SQL 查询

```
select distinct p.a1
from p, r1, r2
where p.a1 = r1.a1 or p.a1 = r2.a1
```

在什么条件下这个查询选择的 *p*.*a*1 值要么在 *r*1 中，要么在 *r*2 中？仔细考察 *r*1 或 *r*2 可能为空的情况。

3.8 考虑图 3-19 中的银行数据库，其中加下划线的是主码。为这个关系数据库构造出如下 SQL 查询：

a. 找出银行中所有有账户但无贷款的客户。

b. 找出与"Smith"居住在同一个城市、同一个街道的所有客户的名字。

c. 找出所有支行的名称，在这些支行中都有居住在"Harrison"的客户所开设的账户。

branch(*branch_name*, *branch_city*, *assets*)
customer (*customer_name*, *customer_street*, *customer_city*)
loan (*loan_number*, *branch_name*, *amount*)
borrower (*customer_name*, *loan_number*)
account (*account_number*, *branch_name*, *balance*)
depositor (*customer_name*, *account_number*)

图 3-19 习题 3.8 和习题 3.15 的银行数据库

3.9 考虑图 3-20 的雇员数据库，其中加下划线的是主码。为下面每个查询写出 SQL 表达式：

a. 找出所有为"First Bank Corporation"工作的雇员名字及其居住城市。

b. 找出所有为"First Bank Corporation"工作且薪金超过 10 000 美元的雇员名字、居住街道和城市。

c. 找出数据库中所有不为"First Bank Corporation"工作的雇员。

d. 找出数据库中工资高于"Small Bank Corporation"的每个雇员的所有雇员。

e. 假设一个公司可以在好几个城市有分部。找出位于"Small Bank Corporation"所有所在城市的所有公司。

f. 找出雇员最多的公司。

g. 找出平均工资高于"First Bank Corporation"平均工资的那些公司。

106
107

employee(*employee_name*, *street*, *city*)
works(*employee_name*, *company_name*, *salary*)
company(*company_name*, *city*)
manages(*employee_name*, *manager_name*)

图 3-20 习题 3.9、习题 3.10、习题 3.16、习题 3.17 和习题 3.20 的雇员数据库

3.10 考虑图 3-20 的关系数据库，给出下面每个查询的 SQL 表达式：

a. 修改数据库使"Jones"现在居住在"Newtown"市。

b. 为"First Bank Corporation"所有工资不超过 100 000 美元的经理增长 10% 的工资，对工资超过 100 000 美元的只增长 3%。

习题

3.11 使用大学模式，用 SQL 写出如下查询。

a. 找出所有至少选修了一门 Comp. Sci. 课程的学生姓名，保证结果中没有重复的姓名。

b. 找出所有没有选修在 2009 年春季之前开设的任何课程的学生的 ID 和姓名。

c. 找出每个系教师的最高工资值。可以假设每个系至少有一位教师。

d. 从前述查询所计算出的每个系最高工资中选出最低值。

108 3.12 使用大学模式，用 SQL 写出如下查询。

a. 创建一门课程"CS-001"，其名称为"Weekly Seminar"，学分为 0。

b. 创建该课程在 2009 年秋季的一个课程段，*sec_id* 为 1。

c. 让 Comp. Sci. 系的每个学生都选修上述课程段。

d. 删除名为 Chavez 的学生选修上述课程段的信息。

e. 删除课程 CS-001。如果在运行此删除语句之前，没有先删除这门课程的授课信息（课程段），会发生什么事情？

f. 删除课程名称中包含"database"的任意课程的任意课程段所对应的所有 *takes* 元组。在课程名的匹配中忽略大小写。

3.13 写出对应于图 3-18 中模式的 SQL DDL。在数据类型上做合理的假设，确保声明主码和外码。

3.14 考虑图 3-18 中的保险公司数据库，其中加下划线的是主码。对这个关系数据库构造如下的 SQL 查询：

a. 找出和"John Smith"的车有关的交通事故数量。

b. 对事故报告编号为"AR2197"中的车牌是"AABB2000"的车辆损坏保险费用更新到 3000 美元。

3.15 考虑图 3-19 中的银行数据库，其中加下划线的是主码。为这个关系数据库构造出如下 SQL 查询：

a. 找出在"Brooklyn"的所有支行都有账户的所有客户。

b. 找出银行的所有贷款额的总和。

c. 找出总资产至少比位于 Brooklyn 的某一家支行要多的所有支行名字。

3.16 考虑图 3-20 中的雇员数据库，其中加下划线的是主码。给出下面每个查询对应的 SQL 表达式：

a. 找出所有为"First Bank Corporation"工作的雇员名字。

b. 找出数据库中所有居住城市和公司所在城市相同的雇员。

c. 找出数据库中所有居住的街道和城市与其经理相同的雇员。

109 d. 找出工资高于其所在公司雇员平均工资的所有雇员。

e. 找出工资总和最小的公司。

3.17 考虑图 3-20 中的关系数据库，给出下面每个查询对应的 SQL 表达式：

a. 为"First Bank Corporation"的所有雇员增长 10% 的工资。

b. 为"First Bank Corporation"的所有经理增长 10% 的工资。

c. 删除"Small Bank Corporation"的雇员在 *works* 关系中的所有元组。

3.18 列出两个原因，说明为什么空值可能被引入到数据库中。

3.19 证明在 SQL 中，< > **all** 等价于 **not in**。

3.20 给出图 3-20 中雇员数据库的 SQL 模式定义。为每个属性选择合适的域，并为每个关系模式选择合适的主码。

3.21 考虑图 3-21 中的图书馆数据库。用 SQL 写出如下查询：

a. 打印借阅了任意由"McGraw-Hill"出版的书的会员名字。

b. 打印借阅了所有由"McGraw-Hill"出版的书的会员名字。

c. 对于每个出版商，打印借阅了多于五本由该出版商出版的书的会员名字。

d. 打印每位会员借阅书籍数量的平均值。考虑这样的情况：如果某会员没有借阅任何书籍，那么该会员根本不会出现在 *borrowed* 关系中。

member(*memb_no*, *name*, *age*)
book(*isbn*, *title*, *authors*, *publisher*)
borrowed(*memb_no*, *isbn*, *date*)

图 3-21 习题 3.21 的图书馆数据库

3.22 不使用 **unique** 结构，重写下面的 **where** 子句：

where unique (**select** *title* from *course*)

3.23 考虑查询：

```
select course_id, semester, year, sec_id, avg (tot_cred)
from takes natural join student
where year = 2009
group by course_id, semester, year, sec_id
having count (ID) >= 2;
```

解释为什么在 **from** 子句中还加上与 *section* 的连接不会改变查询结果。

3.24 考虑查询：

```
with dept_total (dept_name, value) as
    (select dept_name, sum(salary)
    from instructor
    group by dept_name),
dept_total_avg(value) as
    (select avg(value)
    from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value >= dept_total_avg.value;
```

不使用 **with** 结构，重写此查询。

工具

很多关系数据库系统可以从市场上购得，包括 IBM DB2、IBM Informix、Oracle、Sybase，以及微软的 SQL Server。另外还有几个数据库系统可以从网上下载并免费使用，包括 PostgreSQL、MySQL（除几种特定的商业化使用外是免费的）和 Oracle Express edition。

大多数数据库系统提供了命令行界面，用于提交 SQL 命令。此外，大多数数据库还提供了图形化的用户界面（GUI）。它们简化了浏览数据库、创建和提交查询，以及管理数据库的任务。还有商品化的 IDE，用于在多个数据库平台上运行 SQL，包括 Embarcadero 的 RAD Studio 与 Aqua Data Studio。

pgAdmin 工具为 PostgreSQL 提供了 GUI 功能；phpMyAdmin 为 MySQL 提供了 GUI 功能。NetBeans IDE 提供了一个 GUI 前端，可以与很多不同的数据库交互，但其功能有限；Eclipse IDE 通过几种不同插件支持类似的功能，这些插件包括 Data Tools Platform（DTP）和 JBuilder。

本书的 Web 网站 db-book.com 提供了 SQL 模式定义和大学模式的样本数据。该 Web 网站还提供了如何建立和访问一些流行的数据库系统的说明。本章讨论的 SQL 结构是 SQL 标准的一部分，但一些特征可能没被某些数据库所支持。Web 网站上列出了这些不相容的特征，在这些数据库上执行查询时需要多加考虑。

110
111

文献注解

SQL 的最早版本 Sequel 2 由 Chamberlin 等[1976]描述。Sequel 2 是从 Square 语言(Boyce 等[1975]以及 Chamberlin 和 Boyce [1974])衍生出来的。美国国家标准 SQL-86 在 ANSI [1986]中描述。IBM 系统应用体系结构对 SQL 的定义由 IBM[1987]给出。SQL-89 和 SQL-92 官方标准可分别从 ANSI[1989]和 ANSI[1992]获得。

介绍 SQL-92 语言的教材包括 Date 和 Darwen[1997]、Melton 和 Simon[1993]以及 Cannan 和 Otten [1993]。Date 和 Darwen [1997]以及 Date[1993a]都包含了从编程语言角度对 SQL-92 的评论。

介绍 SQL:1999 的教科包括 Melton 和 Simon[2001]以及 Melton[2002]。Eisenberg 和 Melton [1999]提供了对 SQL:1999 的概览。Donahoo 和 Speegle[2005]从开发者的角度介绍了 SQL。Eisenberg 等[2004]提供了对 SQL:2003 的概览。

SQL:1999、SQL:2003、SQL:2006 和 SQL:2008 标准都是作为 ISO/IEC 标准文档集被发布的，24.4 节将介绍关于它们的更多细节。标准文档中塞满了大量的信息，非常难以阅读，主要用于数据库系统的实现。标准文档可以从 Web 网站 http://webstore.ansi.org 上购买。

很多数据库产品支持标准以外的 SQL 特性，还可能不支持标准中的某些特性。关于这些特性的更多信息可在各产品的 SQL 用户手册中找到。

SQL 查询的处理，包括算法和性能等问题，将在第 12 章和第 13 章讨论。关于这些问题的参考文献也在那里。

112

中级 SQL

本章我们继续学习 SQL。我们考虑具有更复杂形式的 SQL 查询、视图定义、事务、完整性约束、关于 SQL 数据定义的更详细介绍以及授权。

4.1 连接表达式

在 3.3.3 节我们介绍了**自然连接**运算。SQL 提供了连接运算的其他形式，包括能够指定显式的**连接谓词**（join predicate），能够在结果中包含被自然连接排除在外的元组。本节我们将讨论这些连接的形式。

本节的例子涉及 *student* 和 *takes* 两个关系，分别如图 4-1 和图 4-2 所示。注意到对于 ID 为 98988 的学生，他在 2010 夏季选修的 BIO-301 课程的 1 号课程段的 *grade* 属性为空值。该空值表示这门课程的成绩还没有得到。

ID	name	dept_name	tot_cred
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

图 4-1 *student* 关系

4.1.1 连接条件

在 3.3.3 节我们介绍了如何表达自然连接，并且介绍了 **join…using** 子句，它是一种自然连接的形式，只需要在指定属性上的取值匹配。SQL 支持另外一种形式的连接，其中可以指定任意的连接条件。

on 条件允许在参与连接的关系上设置通用的谓词。该谓词的写法与 **where** 子句谓词类似，只不过使用的是关键词 **on** 而不是 **where**。与 **using** 条件一样，**on** 条件出现在连接表达式的末尾。

考虑下面的查询，它具有包含 **on** 条件的连接表达式：

```
select *
from student join takes on student.ID = takes.ID;
```

ID	course_id	sec_id	semester	year	grade
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	null

图 4-2 *takes* 关系

上述 **on** 条件表明：如果一个来自 *student* 的元组和一个来自 *takes* 的元组在 *ID* 上的取值相同，那么它们是匹配的。在上例中的连接表达式与连接表达式 *student* **natural join** *takes* 几乎是一样的，因为自然连接运算也需要 *student* 元组和 *takes* 元组是匹配的。这两者之间的一个区别在于：在上述连接查询结果中，*ID* 属性出现两次，一次是 *student* 中的，另一次是 *takes* 中的，即使它们的 *ID* 属性值是相同的。

113
114

实际上，上述查询与以下查询是等价的（换言之，它们产生了完全相同的结果）：

```
select *
from student, takes
where student.ID = takes.ID;
```

正如我们此前所见，关系名用来区分属性名 *ID*，这样 *ID* 的两次出现被分别表示为 *student.ID* 和 *takes.ID*。只显示一次 *ID* 值的查询版本如下：

```
select student.ID as ID, name, dept_name, tot_cred,
    course_id, sec_id, semester, year, grade
from student join takes on student.ID = takes.ID;
```

上述查询的结果如图 4-3 所示。

ID	name	dept_name	tot_cred	course_id	sec_id	semester	year	grade
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2009	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2009	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2009	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2009	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2010	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2009	A
19991	Brandt	History	80	HIS-351	1	Spring	2010	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2010	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2009	B-
45678	Levy	Physics	46	CS-101	1	Fall	2009	F
45678	Levy	Physics	46	CS-101	1	Spring	2010	B+
45678	Levy	Physics	46	CS-319	1	Spring	2010	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2009	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2009	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2010	A-
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2009	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2010	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2009	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2009	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2010	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2009	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2010	null

图 4-3 *student* **join** *takes* **on** *student.ID* = *takes.ID* 的结果，其中省略了 *ID* 的第二次出现

on 条件可以表示任何 SQL 谓词，从而使用 **on** 条件的连接表达式就可以表示比自然连接更为丰富的连接条件。然而，正如上例所示，使用带 **on** 条件的连接表达式的查询可以用不带 **on** 条件的等价表达式来替换，只要把 **on** 子句中的谓词移到 **where** 子句中即可。这样看来，**on** 条件似乎是一个冗余的 SQL 特征。

但是，引入 **on** 条件有两个优点。首先，对于我们马上要介绍的、被称作外连接的这类连接来说，**on** 条件的表现与 **where** 条件是不同的。其次，如果在 **on** 子句中指定连接条件，并在 **where** 子句中出现其余的条件，这样的 SQL 查询通常更容易让人读懂。

4.1.2 外连接

假设我们要显示一个所有学生的列表，显示他们的 *ID*、*name*、*dept_name* 和 *tot_cred*，以及他们所选修的课程。下面的查询好像检索出了所需的信息：

```
select *
from student natural join takes;
```

遗憾的是，上述查询与想要的结果是不同的。假设有一些学生，他们没有选修任何课程。那么这些学生在 *student* 关系中所对应的元组与 *takes* 关系中的任何元组都对，都不会满足自然连接的条件。从而这些学生的数据就不会出现在结果中。这样我们就看不到没有选修任何课程的学生的任何信息。例如，在图 4-1 的 *student* 关系和图 4-2 的 *takes* 关系中，*ID* 为 70557 的学生 Snow 没有选修任何课程，Snow 出现在 *student* 关系中，但是 Snow 的 ID 号没有出现在 *takes* 的 *ID* 列中。从而 Snow 不会出现在自然连接的结果中。

更为一般地，在参与连接的任何一个或两个关系中的某些元组可能会以这种方式"丢失"。外连接（outer join）运算与我们已经学过的连接运算类似，但通过在结果中创建包含空值元组的方式，保留了那些在连接中丢失的元组。

例如，为了保证在我们前例中的名为 Snow 的学生出现在结果中，可以在连接结果中加入一个元

组。它在来自 *student* 关系的所有属性上的值被设置为学生 Snow 的相应值，在所有余下的来自 *takes* 关系属性上的值被设为 *null*，这些属性是 *course_id*、*sec_id*、*semester* 和 *year*。

实际上有三种形式的外连接：

- **左外连接**（left outer join）只保留出现在**左外连接**运算之前（左边）的关系中的元组。
- **右外连接**（right outer join）只保留出现在**右外连接**运算之后（右边）的关系中的元组。
- **全外连接**（full outer join）保留出现在两个关系中的元组。

115 ~ 116

相比而言，为了与外连接运算相区分，我们此前学习的不保留未匹配元组的连接运算被称作**内连接**（inner join）运算。

我们现在详细解释每种形式的外连接是怎样操作的。我们可以按照如下方式计算左外连接运算：首先，像前面那样计算出内连接的结果；然后，对于在内连接的左侧关系中任意一个与右侧关系中任何元组都不匹配的元组 t，向连接结果中加入一个元组 r，r 的构造如下：

- 元组 r 从左侧关系得到的属性被赋为 t 中的值。
- r 的其他属性被赋为空值。

图 4-4 给出了下面查询的结果：

```
select *
from student natural left outer join takes;
```

与内连接的结果不同，此结果中包含了学生 Snow（*ID* 70557），但是在 Snow 对应的元组中，在那些只出现在 *takes* 关系模式中的属性上取空值。

ID	*name*	*dept_name*	*tot_cred*	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2009	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2009	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2009	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2009	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2010	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2009	A
19991	Brandt	History	80	HIS-351	1	Spring	2010	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2010	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2009	B-
45678	Levy	Physics	46	CS-101	1	Fall	2009	F
45678	Levy	Physics	46	CS-101	1	Spring	2010	B+
45678	Levy	Physics	46	CS-319	1	Spring	2010	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2009	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2009	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2010	A-
70557	Snow	Physics	0	*null*	*null*	*null*	*null*	*null*
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2009	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2010	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2009	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2009	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2010	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2009	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2010	*null*

图 4-4　*student* **natural left outer join** *takes* 的结果

作为使用外连接运算的另一个例子，我们可以写出查询"找出所有一门课程也没有选修的学生"：

```
select ID
from student natural left outer join takes
where course_id is null;
```

右外连接和**左外连接**是对称的。来自右侧关系中的不匹配左侧关系任何元组的元组被补上空值，并加入到右外连接的结果中。这样，如果我们使用右外连接来重写前面的查询，并交换列出关系的次序，如下所示：

```
select *
from takes natural right outer join student;
```

我们得到的结果是一样的，只不过结果中属性出现的顺序不同（见图 4-5）。

ID	courseId	secId	semester	year	grade	name	deptName	totCred
00128	CS-101	1	Fall	2009	A	Zhang	Comp. Sci.	102
00128	CS-347	1	Fall	2009	A-	Zhang	Comp. Sci.	102
12345	CS-101	1	Fall	2009	C	Shankar	Comp. Sci.	32
12345	CS-190	2	Spring	2009	A	Shankar	Comp. Sci.	32
12345	CS-315	1	Spring	2010	A	Shankar	Comp. Sci.	32
12345	CS-347	1	Fall	2009	A	Shankar	Comp. Sci.	32
19991	HIS-351	1	Spring	2010	B	Brandt	History	80
23121	FIN-201	1	Spring	2010	C+	Chavez	Finance	110
44553	PHY-101	1	Fall	2009	B	Peltier	Physics	56
45678	CS-101	1	Fall	2009	F	Levy	Physics	46
45678	CS-101	1	Spring	2010	B+	Levy	Physics	46
45678	CS-319	1	Spring	2010	B	Levy	Physics	46
54321	CS-101	1	Fall	2009	A-	Williams	Comp. Sci.	54
54321	CS-190	2	Spring	2009	B+	Williams	Comp. Sci.	54
55739	MU-199	1	Spring	2010	A	Sanchez	Music	38
70557	*null*	*null*	*null*	*null*	*null*	Snow	Physics	0
76543	CS-101	1	Fall	2009	A	Brown	Comp. Sci.	58
76543	CS-319	2	Spring	2010	A	Brown	Comp. Sci.	58
76653	EE-181	1	Spring	2009	C	Aoi	Elec. Eng.	60
98765	CS-101	1	Fall	2009	C-	Bourikas	Elec. Eng.	98
98765	CS-315	1	Spring	2010	B	Bourikas	Elec. Eng.	98
98988	BIO-101	1	Summer	2009	A	Tanaka	Biology	120
98988	BIO-301	1	Summer	2010	*null*	Tanaka	Biology	120

图 4-5 *takes* **natural right outer join** *student* 的结果

全外连接是左外连接与右外连接类型的组合。在内连接结果计算出来之后，左侧关系中不匹配右侧关系任何元组的元组被添上空值并加到结果中。类似地，右侧关系中不匹配左侧关系任何元组的元组也被添上空值并加到结果中。

作为使用全外连接的例子，考虑查询："显示 Comp. Sci. 系所有学生以及他们在 2009 年春季选修的所有课程段的列表。2009 年春季开设的所有课程段都必须显示，即使没有 Comp. Sci. 系的学生选修这些课程段"。此查询可写为：

```
select *
from (select *
      from student
      where dept_name = 'Comp. Sci.')
  natural full outer join
  (select *
   from takes
   where semester = 'Spring' and year = 2009);
```

on 子句可以和外连接一起使用。下述查询与我们见过的第一个使用"*student* **natural left outer join** *takes*"的查询是相同的，只不过属性 *ID* 在结果中出现两次。

```
select *
from student left outer join takes on student.ID = takes.ID;
```

正如我们前面提到的，**on** 和 **where** 在外连接中的表现是不同的。其原因是外连接只为那些对相应内连接结果没有贡献的元组补上空值并加入结果。**on** 条件是外连接声明的一部分，但 **where** 子句却不是。在我们的例子中，ID 为 70557 的学生"Snow"所对应的 *student* 元组的情况就说明了这样的差异。假设我们把前述查询中的 **on** 子句谓词换成 **where** 子句，并使用 **on** 条件 *true*：

```
select *
from student left outer join takes on true
where student.ID = takes.ID;
```

117
119

早先的查询使用带 **on** 条件的左外连接，包括元组（70557，Snow，Physics，0，*null*，*null*，*null*，*null*，*null*），因为在 *takes* 中没有 *ID* = 70557 的元组。然而在后面的查询中，每个元组都满足连接条件 *true*，因此外连接不会产生用补上空值的元组。外连接实际上产生了两个关系的笛卡儿积。因为在 *takes* 中没有 *ID* = 70557 的元组，每次当外连接中出现 *name* = "Snow" 的元组时，*student.ID* 与 *takes.ID* 的取

值必然是不同的，这样的元组会被 **where** 子句谓词排除掉。从而学生 Snow 不会出现在后面查询的结果中。

4.1.3 连接类型和条件

为了把常规连接和外连接区分开来，SQL 中把常规连接称作**内连接**。这样连接子句就可以用 **inner join** 来替换 **outer join**，说明使用的是常规连接。然而关键词 **inner** 是可选的。当 **join** 子句中没有使用 **outer** 前缀，默认的连接类型是 **inner join**。从而，

```
select *
from student join takes using (ID);
```

等价于：

```
select *
from student inner join takes using (ID);
```

类似地，**natural join** 等价于 **natural inner join**。

图 4-6 给出了我们所讨论的所有连接类型的列表。从图中可以看出，任意的连接形式（内连接、左外连接、右外连接或全外连接）可以和任意的连接条件（自然连接、**using** 条件连接或 **on** 条件连接）进行组合。

连接类型
inner join
left outer join
right outer join
full outer join

连接条件
natural
on <predicate>
using $(A_1, A_2, \ldots, A_n)$

图 4-6 连接类型和连接条件

4.2 视图

在上面的所有例子中，我们一直都在逻辑模型层操作，即我们假定了给定的集合中的关系都是实际存储在数据库中的。 120

让所有用户都看到整个逻辑模型是不合适的。出于安全考虑，可能需要向用户隐藏特定的数据。考虑一个职员需要知道教师的标识、姓名和所在系名，但是没有权限看到教师的工资值。此人应该看到的关系由如下 SQL 语句所描述：

```
select ID, name, dept_name
from instructor;
```

除了安全考虑，我们还可能希望创建一个比逻辑模型更符合特定用户直觉的个人化的关系集合。我们可能希望有一个关于 Physics 系在 2009 年秋季学期所开设的所有课程段的列表，其中包括每个课程段在哪栋建筑的哪个房间授课的信息。为了得到这样的列表，我们需要创建的关系是：

```
select course.course_id, sec_id, building, room_number
from course, section
where course.course_id = section.course_id
  and course.dept_name = 'Physics'
  and section.semester = 'Fall'
  and section.year = '2009';
```

我们可以计算出上述查询的结果并存储下来，然后把存储关系提供给用户。但如果这样做的话，一旦 *instructor*、*course* 或 *section* 关系中的底层数据发生变化，那么所存储的查询结果就不再与在这些关系上重新执行查询的结果匹配。一般说来，对像上例那样的查询结果进行计算并存储不是一种好的方式（尽管也存在某些例外情况，我们会在后面讨论）。

相反，SQL 允许通过查询来定义"虚关系"，它在概念上包含查询的结果。虚关系并不预先计算并存储，而是在使用虚关系的时候才通过执行查询被计算出来。

任何像这种不是逻辑模型的一部分，但作为虚关系对用户可见的关系称为**视图**（view）。在任何给

定的实际关系集合上能够支持大量视图。

4.2.1 视图定义

我们在 SQL 中用 **create view** 命令定义视图。为了定义视图，我们必须给视图一个名称，并且必须提供计算视图的查询。**create view** 命令的格式为：

121 **create view** v **as** < query expression >;

其中 < query expression > 可以是任何合法的查询表达式，v 表示视图名。

重新考虑需要访问 *instructor* 关系中除 *salary* 之外的所有数据的职员。这样的职员不应该被允许访问 *instructor* 关系的权限（我们将在后面 4.6 节介绍如何进行授权），相反，可以把视图关系 *faculty* 提供给职员。此视图的定义如下：

```
create view faculty as
select ID, name, dept_name
from instructor;
```

正如前面已经解释过的，视图关系在概念上包含查询结果中的元组，但并不进行预计算和存储。相反，数据库系统存储与视图关系相关联的查询表达式。当视图关系被访问时，其中的元组是通过计算查询结果而被创建出来的。从而，视图关系是在需要的时候才被创建的。

为了创建一个视图，列出 Physics 系在 2009 年秋季学期所开设的所有课程段，以及每个课程段在那些建筑的哪个房间授课的信息，我们可以写出：

```
create view physics_fall_2009 as
  select course.course_id, sec_id, building, room_number
  from course, section
  where course.course_id = section.course_id
      and course.dept_name = 'Physics'
      and section.semester = 'Fall'
      and section.year = '2009';
```

4.2.2 SQL 查询中使用视图

一旦定义了一个视图，我们就可以用视图名指代该视图生成的虚关系。使用视图 *physics_fall_2009*，我们可以用下面的查询找到所有于 2009 年秋季学期在 Watson 大楼开设的 Physics 课程：

```
select course_id
from physics_fall_2009
where building = 'Watson';
```

在查询中，视图名可以出现在关系名可以出现的任何地方。

视图的属性名可以按下述方式显式指定：

```
create view departments_total_salary(dept_name, total_salary) as
  select dept_name, sum (salary)
  from instructor
  group by dept_name;
```

122 上述视图给出了每个系中所有教师的工资总和。因为表达式 **sum**(*salary*) 没有名称，其属性名是在视图定义中显式指定的。

直观上，在任何给定时刻，视图关系中的元组集是该时刻视图定义中的查询表达式的计算结果。因此，如果一个视图关系被计算并存储，一旦用于定义该视图的关系被修改，视图就会过期。为了避免这一点，视图通常这样来实现：当我们定义一个视图时，数据库系统存储视图的定义本身，而不存储定义该视图的查询表达式的执行结果。一旦视图关系出现在查询中，它就被已存储的查询表达式代替。因此，无论我们何时执行这个查询，视图关系都被重新计算。

一个视图可能被用到定义另一个视图的表达式中。例如，我们可以如下定义视图 *physics_fall_2009_watson*，它列出了于 2009 年秋季学期在 Watson 大楼开设的所有 Physics 课程的标识和房间号：

```
create view physics_fall_2009_watson as
    select course_id, room_number
    from physics_fall_2009
    where building = 'Watson';
```

其中 *physics_fall_2009_watson* 本身是一个视图关系，它等价于：

```
create view physics_fall_2009_watson as
    (select course_id, room_number
     from (select course.course_id, building, room_number
           from course, section
           where course.course_id = section.course_id
           and course.dept_name = 'Physics'
           and section.semester = 'Fall'
           and section.year = '2009')
     where building = 'Watson';
```

4.2.3 物化视图

特定数据库系统允许存储视图关系，但是它们保证：如果用于定义视图的实际关系改变，视图也跟着修改。这样的视图被称为**物化视图**(materialized view)。

例如，考察视图 *departments_total_salary*。如果上述视图是物化的，它的结果就会存放在数据库中。然而，如果一个 *instructor* 元组被插入到 *instructor* 关系中，或者从 *instructor* 关系中删除，定义视图的查询结果就会变化，其结果是物化视图的内容也必须更新。类似地，如果一位教师的工资被更新，那么 *departments_total_salary* 中对应于该教师所在系的元组必须更新。 123

保持物化视图一直在最新状态的过程称为**物化视图维护**(materialized view maintenance)，或者通常简称**视图维护**(view maintenance)，这将在 13.5 节进行介绍。当构成视图定义的任何关系被更新时，可以马上进行视图维护。然而某些数据库系统在视图被访问时才执行视图维护。还有一些系统仅采用周期性的物化视图更新方式。在这种情况下，当物化视图被使用时，其中的内容可能是陈旧的，或者说过时的。如果应用需要最新数据的话，这种方式是不适用的。某些数据库系统允许数据库管理员来控制在每个物化视图上需要采取上述的哪种方式。

频繁使用视图的应用将会从视图的物化中获益。那些需要快速响应基于大关系上聚集计算的特定查询也会从创建与查询相对应的物化视图中受益良多。在这种情况下，聚集结果很可能比定义视图的大关系要小得多，其结果是利用物化视图来回答查询就很快。它甚至免了读取大的底层关系。当然，物化视图查询所带来的好处还需要与存储代价和增加的更新开销相权衡。

SQL 没有定义指定物化视图的标准方式，但是很多数据库系统提供了各自的 SQL 扩展来实现这项任务。一些数据库系统在底层关系变化时，总是把物化视图保持在最新状态；但有另外一些系统允许物化视图过时，但周期性地重新计算物化视图。

4.2.4 视图更新

尽管对查询而言，视图是一个有用的工具，但如果我们用它们来表达更新、插入或删除，它们可能带来严重的问题。困难在于，用视图表达的数据库修改必须被翻译为对数据库逻辑模型中实际关系的修改。

假设我们此前所见的视图 *faculty* 被提供给一个职员。既然我们允许视图名出现在任何关系名可以出现的地方，该职员可以这样写出：

```
insert into faculty
    values ('30765', 'Green', 'Music');
```

这个插入必须表示为对 *instructor* 关系的插入，因为 *instructor* 是数据库系统用于构造视图 *faculty* 的实际关系。然而，为了把一个元组插入到 *instructor* 中，我们必须给出 *salary* 的值。存在两种合理的解决方法来处理该插入：

- 拒绝插入，并向用户返回一个错误信息。
- 向 *instructor* 关系插入元组 ('30765', 'Green', 'Music', null)。 124

通过视图修改数据库的另一类问题发生在这样的视图上：

```
create view instructor_info as
select ID, name, building
from instructor, department
where instructor.dept_name = department.dept_name;
```

这个视图列出了大学里每个教师的 *ID*、*name* 和建筑名。考虑如下通过该视图的插入：

```
insert into instructor_info
    values ('69987', 'White', 'Taylor');
```

假设没有标识为 69987 的教师，也没有位于 Taylor 大楼的系。那么向 *instructor* 和 *department* 关系中插入元组的唯一可能的方法是：向 *instructor* 中插入元组 ('69987', 'White', *null*, *null*)，并向 *department* 中插入元组 (*null*, 'Taylor', *null*)。于是我们得到如图 4-7 所示的关系。但是这个更新并没有产生出所需的结果，因为视图关系 *instructor_info* 中仍然不包含元组 ('69987', 'White', 'Taylor')。因此，通过利用空值来更新 *instructor* 和 *department* 关系以得到对 *instructor_info* 所需的更新是不可行的。

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
69987	White	*null*	*null*

instructor

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Electrical Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000
null	Taylor	*null*

department

图 4-7 插入元组后的 *instructor* 和 *department* 关系

由于如上所述的种种问题，除了一些有限的情况之外，一般不允许对视图关系进行修改。不同的数据库系统指定了不同的条件以允许更新视图关系；请参考数据库系统手册以获得详细信息。通过视图进行数据库修改的通用问题已经成为重要的研究课题，文献注解中引用了一些这方面的研究。

一般说来，如果定义视图的查询对下列条件都能满足，我们称 SQL 视图是**可更新的**(updatable)（即视图上可以执行插入、更新或删除）：

- **from** 子句中只有一个数据库关系。
- **select** 子句中只包含关系的属性名，不包含任何表达式、聚集或 **distinct** 声明。
- 任何没有出现在 **select** 子句中的属性可以取空值；即这些属性上没有 **not null** 约束，也不构成主码的一部分。
- 查询中不含有 **group by** 或 **having** 子句。

在这些限制下，下面的视图上允许执行 **update**、**insert** 和 **delete** 操作：

```
create view history_instructors as
select *
from instructor
where dept_name = 'History';
```

即便是在可更新的情况下，下面这些问题仍然存在。假设一个用户尝试向视图 *history_instructors* 中插入元组 ('25566', 'Brown', 'Biology', 100000)，这个元组可以被插入到 *instructor* 关系中，但是由于它不满足视图所要求的选择条件，它不会出现在视图 *history_instructors* 中。

在默认情况下，SQL 允许执行上述更新。但是，可以通过在视图定义的末尾包含 **with check option** 子句的方式来定义视图。这样，如果向视图中插入一条不满足视图的 **where** 子句条件的元组，数据库系统将拒绝该插入操作。类似地，如果新值不满足 **where** 子句的条件，更新也会被拒绝。

SQL:1999 对于何时可以在视图上执行插入、更新和删除有更复杂的规则集，该规则集允许通过一类更大视图进行更新。但是这些规则过于复杂，我们就不在这里讨论了。

125
|
126

4.3 事务

事务(transaction)由查询和(或)更新语句的序列组成。SQL标准规定当一条SQL语句被执行，就隐式地开始了一个事务。下列SQL语句之一会结束一个事务：

- **Commit work**：提交当前事务，也就是将该事务所做的更新在数据库中持久保存。在事务被提交后，一个新的事务自动开始。
- **Rollback work**：回滚当前事务，即撤销该事务中所有SQL语句对数据库的更新。这样，数据库就恢复到执行该事务第一条语句之前的状态。

关键词 **work** 在两条语句中都是可选的。

当在事务执行过程中检测到错误时，事务回滚是有用的。在某种意义上，事务提交就像对编辑文档的变化存盘，而回滚就像不保存变化退出编辑。一旦某事务执行了 **commit work**，它的影响就不能用 **rollback work** 来撤销了。数据库系统保证在发生诸如某条SQL语句错误、断电、系统崩溃这些故障的情况下，如果一个事务还没有完成 **commit work**，其影响将被回滚。在断电和系统崩溃的情况下，回滚会在系统重启后执行。

例如，考虑一个银行应用，我们需要从一个银行账户上把钱转到同一家银行的另一个账户。为了这样做，我们需要更新两个账户的余额，把需要转移的资金额从一个账户划走，并把它加到另一个账户上。如果在从第一个账户上划走资金以后，但在把这笔资金加入第二个账户之前发生了系统崩溃，那么银行账户就会不一致。如果在第一个账户划走资金之前先往第二个账户存款，并且在存款之后马上发生系统崩溃，那么也会出现类似的问题。

作为另一个例子，考虑我们正在使用的大学应用的例子。我们假设 *student* 关系中每个元组在 *tot_cred* 属性上的取值需要保持在最新状态，只要学生成功修完一门课程，该属性值就要更新。为了这样做，只要 *takes* 关系被更新为记录一位学生成功修完一门课程的信息(通过赋予适当的成绩)，相应的 *student* 元组也必须更新。如果执行这两个更新的任务在执行完一个更新后，但在第二个更新前崩溃了，数据库中的数据就是不一致的。

一个事务或者在完成所有步骤后提交其行为，或者在不能成功完成其所有动作的情况下回滚其所有动作。通过这种方式数据库提供了对事务具有**原子性**(atomic)的抽象，原子性也就是不可分割性。要么事务的所有影响被反映到数据库中，要么任何影响都没有(在回滚之后)。 127

如果把事务的概念应用到上述应用中，那些更新语句就会作为单个事务执行。在事务执行其某一条语句时出错会导致事务早先执行的语句的影响被撤销，从而不会让数据库处于部分更新状态。

如果程序没有执行两条命令中的任何一条而终止了，那么更新要么被提交要么被回滚。SQL标准并没有指出究竟执行哪一种，如何选择依赖于具体实现。

在很多SQL实现中，默认方式下每个SQL语句自成一个事务，且一执行完就提交。如果一个事务要执行多条SQL语句，就必须关闭单独SQL语句的自动提交。如何关闭自动提交也依赖于特定的SQL实现，尽管在诸如JDBC或ODBC那样的应用编程接口中存在标准化方式来完成这项工作。我们将在5.1.1和5.1.2节分别学习JDBC和ODBC。

一个较好的选择是，作为SQL:1999标准的一部分(但目前只有一些SQL实现支持)，允许多条SQL语句包含在关键字 **begin atomic…end** 之间。所有在关键字之间的语句构成了一个单一事务。

我们将在第14章学习事务的更多特性；第15章和第16章介绍在单个数据库中实现事务的相关问题，而在第19章介绍跨多个数据库上实现事务的相关问题，这是为了处理不同银行的账户之间转账的问题，不同银行有不同的数据库。

4.4 完整性约束

完整性约束保证授权用户对数据库所做的修改不会破坏数据的一致性。因此，完整性约束防止的是对数据的意外破坏。

完整性约束的例子有：

- 教师姓名不能为 *null*。
- 任意两位教师不能有相同的教师标识。
- *course* 关系中的每个系名必须在 *department* 关系中有一个对应的系名。
- 一个系的预算必须大于 0.00 美元。

一般说来，一个完整性约束可以是属于数据库的任意谓词，但检测任意谓词的代价可能太高。因此，大多数数据库系统允许用户指定那些只需极小开销就可以检测的完整性约束。

在 3.2.2 节我们已经见过了一些完整性约束的形式。本节我们将学习更多的完整性约束形式。在
128 第 8 章我们学习另一种被称作**函数依赖**的完整性约束形式，它主要应用在模式设计的过程中。

完整性约束通常被看成是数据库模式设计过程的一部分，它作为用于创建关系的 **create table** 命令的一部分被声明。然而，完整性约束也可以通过使用 **alter table** *table-name* **add** *constraint* 命令施加到已有关系上，其中 *constraint* 可以是关系上的任意约束。当执行上述命令时，系统首先保证关系满足指定的约束。如果满足，那么约束被施加到关系上；如果不满足，则拒绝执行上述命令。

4.4.1 单个关系上的约束

我们在 3.2 节中描述了如何用 **create table** 命令定义关系表。**create table** 命令还可以包括完整性约束语句。除了主码约束之外，还有许多其他可以包括在 **create table** 命令中的约束。允许的完整性约束包括：

- **not null**。
- **unique**。
- **check**(<谓词>)。

我们将在下面的几节中讨论上述每一种约束。

4.4.2 not null 约束

正如我们在第 3 章中讨论过的，空值是所有域的成员。因此在默认情况下是 SQL 中每个属性的合法值。然而对于一些属性来说，空值可能是不合适的。考虑 *student* 关系中的一个元组，其中 *name* 是 *null*。这样的元组给出了一个未知学生的学生信息；因此它不含有有用的信息。类似地，我们不会希望系的预算为 *null*。在这些情况下，我们希望禁止空值，我们可以通过如下声明来通过限定属性 *name* 和 *budget* 的域来排除空值：

name **varchar**(20) **not null**
budget **numeric**(12, 2) **not null**

not null 声明禁止在该属性上插入空值。任何可能导致向一个声明为 **not null** 的属性插入空值的数据库修改都会产生错误诊断信息。

许多情况下我们希望避免空值。尤其是 SQL 禁止在关系模式的主码中出现空值。因此，在我们的
129 大学例子中，在 *department* 关系上如果声明属性 *dept_name* 为 *department* 的主码，那它就不能为空。因此它不必显式地声明为 **not null**。

4.4.3 unique 约束

SQL 还支持下面这种完整性约束：

unique ($A_1, A_2, \cdots, A_m$)

unique 声明指出属性 $A_1, A_2, \cdots, A_m$ 形成了一个候选码；即在关系中没有两个元组能在所有列出的属性上取值相同。然而候选码属性可以为 *null*，除非它们已被显式地声明为 **not null**。回忆一下，空值不等于其他的任何值。（这里对空值的处理与 3.8.4 节中定义的 **unique** 结构一样。）

4.4.4 check 子句

当应用于关系声明时，**check**(P)子句指定一个谓词 P，关系中的每个元组都必须满足谓词 P。

通常用 **check** 子句来保证属性值满足指定的条件，实际上创建了一个强大的类型系统。例如，在创建关系 *department* 的 **create table** 命令中的 **check**(*budget* > 0) 子句将保证 **budget** 上的取值是正数。

作为另一个例子，考虑如下语句：

```
create table section
  (course_id varchar (8),
   sec_id varchar (8),
   semester varchar (6),
   year numeric (4, 0),
   building varchar (15),
   room_number varchar (7),
   time_slot_id varchar (4),
   primary key (course_id, sec_id, semester, year),
   check (semester in ('Fall', 'Winter', 'Spring', 'Summer')));
```

这里我们用 **check** 子句模拟了一个枚举类型，通过指定 *semester* 必须是'Fall'、'Winter'、'Spring'或'Summer'中的一个来实现。这样，**check** 子句允许以有力的方式对属性域加以限制，这是大多数编程语言的类型系统都不能允许的。

根据SQL标准，**check** 子句中的谓词可以是包括子查询在内的任意谓词。然而，当前还没有一个广泛使用的数据库产品允许包含子查询的谓词。 130

4.4.5 参照完整性

我们常常希望保证在一个关系中给定属性集上的取值也在另一关系的特定属性集的取值中出现。这种情况称为**参照完整性**(referential integrity)。

正如我们此前在3.2.2节所见，外码可以用作为SQL中 **create table** 语句一部分的 **foreign key** 子句来声明。我们用大学数据库SQL DLL定义的一部分来说明外码声明，如图4.8所示。*course* 表的定义中有一个声明"**foreign key**(*dept_name*) **references** *department*"。这个外码声明表示，在每个课程元组中指定的系名必须在 *department* 关系中存在。没有这个约束，就可能会为一门课程指定一个不存在的系名。

更一般地，令关系 r_1 和 r_2 的属性集分别为 R_1 和 R_2，主码分别为 K_1 和 K_2。如果要求对 r_2 中任意元组 t_2，均存在 r_1 中元组 t_1 使得 $t_1.K_1 = t_2.\alpha$，我们称 R_2 的子集 α 为参照关系 r_1 中 K_1 的外码(foreign key)。

这种要求称为**参照完整性约束**(referential integrity constraint)或**子集依赖**(subset dependency)。后一种称法是由于上述参照完整性可以表示为这样一种要求：r_2 中 α 上的取值集合必须是 r_1 中 K_1 上的取值集合的子集。请注意，为使参照完整性约束有意义，α 和 K_1 必须是相容的属性集；也就是说，要么 α 等于 K_1，要么它们必须包含相同数目的属性，并且对应属性的类型必须相容(这里我们假设 α 和 K_1 是有序的)。不同于外码约束，参照完整性约束通常不要求 K_1 是 r_1 的主码；其结果是，r_1 中可能有不止一个元组在属性 K_1 上取值相同。

默认情况下，SQL中外码参照的是被参照表中的主码属性。SQL还支持一个可以显式指定被参照关系的属性列表的 **references** 子句。然而，这个指定的属性列表必须声明为被参照关系的候选码，要么使用 **primary key** 约束，要么使用 **unique** 约束。在更为普遍的参照完整性约束形式中，被参照的属性不必是候选码，这样的形式还不能在SQL中直接声明。SQL标准提供了另外的结构用于实现这样的约束，4.4.7节将描述这样的结构。

我们可以使用如下的简写形式作为属性定义的一部分，并声明该属性为外码：

dept_name **varchar**(20) **references** *department*

当违反参照完整性约束时，通常的处理是拒绝执行导致完整性破坏的操作(即进行更新操作的事务被回滚)。但是，在 **foreign key** 子句中可以指明：如果被参照关系上的删除或更新动作违反了约束，那么系统必须采取一些步骤通过修改参照关系中的元组来恢复完整性约束，而不是拒绝这样的动作。 131
考虑在关系 *course* 上的如下完整性约束定义：

```
create table classroom
        (building varchar(15),
        room_number varchar(7),
        capacity numeric(4, 0),
        primary key (building, room_number))
    create table department
        (dept_name varchar(20),
        building varchar(15),
        budget numeric(12, 2) check (budget > 0),
        primary key (dept_name))
    create table course
        (course_id varchar(8),
        title varchar(50),
        dept_name varchar(20),
        credits numeric(2, 0) check (credits > 0),
        primary key (course_id),
        foreign key (dept_name)references department)
    create table instructor
        (ID varchar(5),
        name varchar(20) not null
        dept_name varchar(20),
        salary numeric(8, 2), check (salary > 29000),
        primary key (ID),
        foreign key (dept_name) references department)
    create table section
        (course_id varchar(8),
        sec_id varchar(8),
        semester varchar(6), check (semester in
            ('Fall', 'Winter', 'Spring', 'Summer')),
        year numeric(4, 0), check (year > 1759 and year < 2100)
        building varchar(15),
        room_number varchar(7),
        time_slot_id varchar(4),
        primary key (course_id, sec_id, semester, year),
        foreign key (course_id) references course,
        foreign key (building, room_number) references classroom)
```

图 4-8 大学数据库的部分 SQL 数据定义

```
create table course
    ( ...
    foreign key (dept_name) references department
        on delete cascade
        on update cascade,
    ...);
```

由于有了与外码声明相关联的 **on delete cascade** 子句，如果删除 *department* 中的元组导致了此参照完整性约束被违反，则删除并不被系统拒绝，而是对 *course* 关系作"级联"删除，即删除参照了被删除系的元组。类似地，如果更新被参照字段时违反了约束，则更新操作并不被系统拒绝，而是将 *course* 中参照的元组的 *dept_name* 字段也改为新值。SQL 还允许 **foreign key** 子句指明除 **cascade** 以外的其他动作，如果约束被违反，可将参照域（这里是 *dept_name*）置为 *null*（用 **set null** 代替 **cascade**），或者置为域的默认值（用 **set default**）。

如果存在涉及多个关系的外码依赖链，则在链一端所做的删除或更新可能传至整个链。实践习题 4.9 中的一个有趣的情况是，在一个关系上定义的 **foreign key** 约束所参照的关系就是它自己。如果一个级联更新或删除导致的对约束的违反不能通过进一步的级联操作解决，则系统中止该事务。于是，该事务所做的所有改变及级联动作将被撤销。

空值使得 SQL 中参照约束的语义复杂化了。外码中的属性允许为 *null*，只要它们没有被声明为 **not null**。如果给定元组中外码的所有列上均取非空值，则对该元组采用外码约束的通常定义。如果某外码列为 *null*，则该元组自动被认为满足约束。

这样的规定有时不一定是正确的选择，因此 SQL 也提供一些结构使你可以改变对空值的处理；我们在此不讨论这样的结构。

4.4.6 事务中对完整性约束的违反

事务可能包括几个步骤，在某一步之后完整性约束也许会暂时被违反，但是后面的某一步也许就会消除这个违反。例如，假设我们有一个主码为 *name* 的 *person* 关系，还有一个属性是 *spouse*，并且 *spouse* 是在 *person* 上的一个外码。也就是说，约束要求 *spouse* 属性必须包含在 *person* 表里出现的名字。假设我们希望在上述关系中插入两个元组，一个是关于 John 的，另一个是关于 Mary 的，这两个元组的配偶属性分别设置为 Mary 和 John，以此表示 John 和 Mary 彼此之间的婚姻关系。无论先插入哪个元组，插入第一个元组的时候都会违反外码约束。在插入第二个元组后，外码约束又会满足了。

为了处理这样的情况，SQL 标准允许将 **initially deferred** 子句加入到约束声明中；这样完整性约束不是在事务的中间步骤上检查，而是在事务结束的时候检查。一个约束可以被指定为可延迟的（deferrable），这意味着默认情况下它会被立即检查，但是在需要的时候可以延迟检查。对于声明为可延迟的约束，执行 **set constraints** *constraint-list* **deferred** 语句作为事务的一部分，会导致对指定约束的检查被延迟到该事务结束时执行。

然而，读者应该注意的是默认的方式是立即检查约束，而且许多数据库实现不支持延迟约束检查。

如果 *spouse* 属性可以被取为 *null*，我们可以用另一种方式来避开在上面例子中的问题：在插入 John 和 Mary 元组时，我们设置其 *spouse* 属性为 *null*，然后再更新它们的值。然而，这个技术需要更大的编程量，而且如果属性不能设为 *null* 的话，此方法就不可行。

4.4.7 复杂 check 条件与断言

本节描述 SQL 标准所支持的为外一些用于声明完整性约束的结构。然而，读者应该注意的是，这些结构目前还没有被大多数数据库系统支持。

正如 SQL 标准所定义的，**check** 子句中的谓词可以是包含子查询的任意谓词。如果一个数据库实现支持在 **check** 子句中出现子查询，我们就可以在关系 *section* 上声明如下所示的参照完整性约束：

check (*time_slot_id* **in** (**select** *time_slot_id* **from** *time_slot*))

这个 **check** 条件检测在 *section* 关系中每个元组的 *time_slot_id* 的确是在 *time_slot* 关系中某个时间段的标识。因此这个条件不仅在 *section* 中插入或修改元组时需要检测，而且在 *time_slot* 关系改变时也需要检测（如在 *time_slot* 关系中一当一个元组被删除或修改的情况下）。

在我们的大学模式上，另一个自然的约束是：每个课程段都需要有至少一位教师来讲授。为了强制实现此约束，一种方案是声明 *section* 关系的属性集（*course_id*, *sec_id*, *semester*, *year*）为外码，它参照 *teaches* 关系中的相应属性。遗憾的是，这些属性并未构成 *teaches* 关系的主码。如果数据库系统支持在 check 约束中出现子查询的话，可以使用与 *time_slot* 属性类似的 check 约束来强制实现上述约束。

复杂 **check** 条件在我们希望确保数据完整性的时候是很有用的，但其检测开销可能会很大。例如，**check** 子句中的谓词不仅需要在 *section* 关系发生更新时计算，而且也可能在 *time_slot* 关系发生更新时检测，因为 *time_slot* 在子查询中被引用了。

一个**断言**（*assertion*）就是一个谓词，它表达了我们希望数据库总能满足的一个条件。域约束和参照完整性约束是断言的特殊形式。我们在前面用大量篇幅介绍了这几种形式的断言，是因为它们容易检测并且适用于很多数据库应用。但是，还有许多约束不能仅用这几种特殊形式来表达。如有两个这样的例子：

- 对于 *student* 关系中的每个元组，它在属性 *tot_cred* 上的取值必须等于该生所成功修完课程的学分总和。

- 每位教师不能在同一个学期的同一个时间段在两个不同的教室授课。

SQL 中的断言为如下形式：

```
create assertion <assertion-name> check <predicate>;
```

图 4-9 给出了我们如何用 SQL 写出第一个约束的示例。由于 SQL 不提供"for all X, $P(X)$"结构（其中 P 是一个谓词），我们只好通过等价的"not exists X such that not $P(X)$"结构来实现此约束。这一结构可以用 SQL 来表示。

```
create assertion credits_earned_constraint check
    (not exists (select ID
                 from student
                 where tot_cred <> (select sum(credits)
                                    from takes natural join course
                                    where student.ID = takes.ID
                                    and grade is not null and grade <> 'F' ));
```

图 4-9 一个断言的例子

我们把第二个约束的声明留作练习。

当创建断言时，系统要检测其有效性。如果断言有效，则今后只有不破坏断言的数据库修改才被允许。如果断言较复杂，则检测会带来相当大的开销。因此，使用断言应该特别小心。由于检测和维护断言的开销较大，一些系统开发者省去了对一般性断言的支持，或者只提供易于检测的特殊形式的断言。

目前，还没有一个广泛使用的数据库系统支持在 **check** 子句的谓词中使用子查询或 **create assertion** 结构。然而，如果数据库系统支持触发器的话，可以通过使用触发器来实现等价的功能，触发器将在 5.3 节介绍，5.3 节还将介绍如何用触发器来实现 *time_slot_id* 上的参照完整性约束。

4.5 SQL 的数据类型与模式

在第 3 章中，我们介绍了一些 SQL 支持的固有数据类型，如整数类型、实数类型和字符类型。SQL 还支持一些其他的固有数据类型，我们将在下面描述。我们还将描述如何在 SQL 中创建基本的用户定义类型。

4.5.1 SQL 中的日期和时间类型

除了我们在 3.2 节介绍过的基本数据类型以外，SQL 标准还支持与日期和时间相关的几种数据类型：

- **date**：日历日期，包括年（四位）、月和日。
- **time**：一天中的时间，包括小时、分和秒。可以用变量 **time**(p) 来表示秒的小数点后的数字位数（这里默认值为 0）。通过指定 **time with timezone**，还可以把时区信息连同时间一起存储。
- **timestamp**：**date** 和 **time** 的组合。可以用变量 **timestamp**(p) 来表示秒的小数点后的数字位数（这里默认值为 6）。如果指定 **with timezone**，则时区信息也会被存储。

日期和时间类型的值可按如下方式说明：

```
date '2001 - 04 - 25'
time '09:30:00'
timestamp '2001 - 04 - 25 10:29:01.45'
```

日期类型必须按照如上年月日的格式顺序指定。**time** 和 **timestamp** 的秒部分可能会有小数部分，像上述时间戳中的情况一样。

我们可以利用 **cast** e **as** t 形式的表达式来将一个字符串（或字符串表达式）e 转换成类型 t，其中 t 是

---

[1] 我们假设不在第二课程采用远程授课的形式。另一个约束是指定"一位教师在一个给定学期的相同时间段不能讲授两门课程"，由于有时候课程是交叉注册的，即对一门课程给出了两个标识和名称，所以可能不满足此约束。

date、**time**、**timestamp** 中的一种。字符串必须符合正确的格式，像本段开头说的那样。当需要时，时区信息可以从系统设置中得到。

我们可以利用 **extract**(*field* **from** *d*)，从 **date** 或 **time** 值 *d* 中提取出单独的域，这里的域可以是 **year**、**month**、**day**、**hour**、**minute** 或者 **second** 中的任意一种。时区信息可以用 **timezone_hour** 和 **timezone_minute** 来提取。

SQL 定义了一些函数以获取当前日期和时间。例如，**current_date** 返回当前日期，**current_time** 返回当前时间(带有时区)，还有 **localtime** 返回当前的本地时间(不带时区)。时间戳(日期加上时间)由 **current_timestamp**(带有时区)以及 **localtimestamp**(本地日期和时间，不带时区)返回。

SQL 允许在上面列出的所有类型上进行比较运算，也允许在各种数字类型上进行算术运算和比较运算。SQL 还支持 **interval** 数据类型，它允许在日期、时间和时间间隔上进行计算。例如，假设 *x* 和 *y* 都是 **date** 类型，那么 *x* − *y* 就是时间间隔类型，其值为从日期 *x* 到日期 *y* 间隔的天数。类似地，在日期或时间上加减一个时间间隔将分别得到新的日期或时间。

4.5.2 默认值

SQL 允许为属性指定默认值，如下面的 **create table** 语句所示：

```
create table student
  (ID varchar (5),
   name varchar (20) not null,
   dept_name varchar (20),
   tot_cred numeric (3, 0) default 0,
   primary key (ID));
```

tot_cred 属性的默认值被声明为 0。这样，当一个元组被插入到 *student* 关系中，如果没有给出 *tot_cred* 属性的值，那么该元组在此属性上的取值就被置为 0。下面的插入语句说明了在插入操作中如何省略 *tot_cred* 属性的值：

```
insert into student(ID, name, dept_name)
  values ('12789', 'Newman', 'Comp. Sci.');
```

4.5.3 创建索引

许多查询只涉及文件中的少量记录。例如，像这样的查询"找出 Physics 系的所有教师"，或"找出 *ID* 为 22201 的学生的 *tot_cred* 值"，只涉及学生记录中的一小部分。如果系统读取每条记录并——检查其 *ID* 域是否为"22201"，或者 *dept_name* 域是否取值为"Physics"，这样的方式是很低效的。

在关系的属性上所创建的**索引**(index)是一种数据结构，它允许数据库系统高效地找到关系中那些在索引属性上取给定值的元组，而不用扫描关系中的所有元组。例如，如果我们在 *student* 关系的属性 *ID* 上创建了索引，数据库系统不用读取 *student* 关系中的所有元组，就可以直接找到任何像 22201 或 44553 那样具有指定 *ID* 值的记录。索引也可以建立在一个属性列表上，例如在 *student* 的属性 *name* 和 *dept_name* 上。

我们将在后面第 11 章学习索引是如何实现的，包括一种被称作 B$^+$ 树的索引，它是一种特别的、广泛使用的索引类型。

尽管 SQL 语言没有给出创建索引的正式语法定义，但很多数据库都支持使用如下所示的语法形式来创建索引：

```
create index studentID_index on student(ID);
```

上述语句在 *student* 关系的属性 *ID* 上创建了一个名为 *studentID_index* 的索引。

如果用户提交的 SQL 查询可以从索引的使用中获益，那么 SQL 查询处理器就会自动使用索引。例如，给定的 SQL 查询是选出 *ID* 为 22201 的 *student* 元组，SQL 查询处理器就会使用上面定义的 *studentID_index* 索引来找到所需元组，而不用读取整个关系。

4.5.4 大对象类型

许多当前的数据库应用需要存储可能很大(KB 级)的属性，例如一张照片，或者非常大的属性

(MB 级甚至 GB 级)，例如高清晰度的医学图像或视频片断。因此 SQL 提供字符数据的大对象数据类型(**clob**)和二进制数据的大对象数据类型(**blob**)。在这些数据类型中字符"lob"代表"Large OBject"。例如，我们可以声明属性

book_review **clob** (10KB)
image **blob** (10MB)
movie **blob** (2GB)

对于包含大对象(好几个 MB 甚至 GB)的结果元组而言，把整个大对象放入内存中是非常低效和不现实的。相反，一个应用通常用一个 SQL 查询来检索出一个大对象的"定位器"，然后在宿主语言中用这个定位器来操纵对象。应用本身也是用宿主语言书写的。例如，JDBC 应用编程接口(5.1.1 节描述)允许获取一个定位器而不是整个大对象；然后用这个定位器来一点一点地取出这个大对象，而不是一次取出全部，这很像用一个 **read** 函数调用从操作系统文件中读取数据。

4.5.5 用户定义的类型

SQL 支持两种形式的用户定义数据类型。第一种称为**独特类型**(distinct type)，我们将在这里介绍。
138 另一种称为**结构化数据类型**(structured data type)，允许创建具有嵌套记录结构、数组和多重集的复杂数据类型。在本章我们不介绍结构化数据类型，而是在后面第 22 章描述。

一些属性可能会有相同的数据类型。例如，用于学生名和教师名的 *name* 属性就可能有相同的域：所有人名的集合。然而，*budget* 和 *dept_name* 的域肯定应该是不同的。*name* 和 *dept_name* 是否应该有相同的域，这一点就不那么明显了。在实现层，教师姓名和系的名字都是字符串。然而，我们通常不认为"找出所有与某个系同名的教师"是一个有意义的查询。因此，如果我们在概念层而不是物理层来看待数据库的话，*name* 和 *dept_name* 应该有不同的域。

更重要的是，在现实中，把一个教师的姓名赋给一个系名可能是一个程序上的错误；类似地，把一个以美元表示的货币值直接与一个以英镑表示的货币值进行比较几乎可以肯定是程序上的错误。一个好的类型系统应该能够检测出这类赋值或比较。为了支持这种检测，SQL 提供了**独特类型**(distinct type)的概念。

可以用 **create type** 子句来定义新类型。例如，下面的语句：

create type *Dollars* **as numeric** (12, 2) **final**;
create type *Pounds* **as numeric** (12, 2) **final**;

把两个用户定义类型 *Dollars* 和 *Pounds* 定义为总共 12 位数字的十进制数，其中两位放在十进制小数点后。(在此关键字 **final** 并不是真的有意义，它是 SQL:1999 标准要求的，其原因我们不在这里讨论了；一些系统实现允许忽略 **final** 关键字。)然后新创建的类型就可以用作关系属性的类型。例如，我们可以把 *department* 表定义为：

create table *department*
(*dept_name* **varchar** (20),
building **varchar** (15),
budget Dollars);

尝试为 *Pounds* 类型的变量赋予一个 *Dollars* 类型的值会导致一个编译时错误。尽管这两者都是相同的数值类型。这样的赋值很可能是由程序错误引起的，或许是程序员忘记了货币之间的区别。为不同的货币声明不同的类型能帮助发现这些错误。

由于有强类型检查，表达式(*department.budget* + 20)将不会被接受，因为属性和整型常数 20 具有不同的类型。一种类型的数值可以被转换(也即 *cast*)到另一个域，如下所示：

139 **cast** (*department.budget* **to** *numeric*(12, 2))

我们可以在数值类型上做加法，但是为了把结果存回到一个 *Dollars* 类型的属性中，我们需要用另一个类型转换表达式来把数值类型转换回 *Dollars* 类型。

SQL 提供了 **drop type** 和 **alter type** 子句来删除或修改以前创建过的类型。

在把用户定义类型加入到 SQL(在 SQL:1999 中)之前，SQL 有一个相似但稍有不同的概念：域

(domain)(在SQL-92中引入)，它可以在基本类型上施加完整性约束。例如，我们可以定义一个域*DDollars*，如下所示：

```
create domain DDollars as numeric(12, 2) not null;
```

*DDollars*域可以用作属性类型，正如我们用*Dollars*类型一样。然而，类型和域之间有两个重大的差别：

1. 在域上可以声明约束，例如**not null**，也可以为域类型变量定义默认值。然而在用户定义类型上不能声明约束或默认值。设计用户定义类型不仅是用它来指定属性类型，而且还将它用在不能施加约束的地方对SQL进行过程扩展。

2. 域并不是强类型的。因此一个域类型的值可以被赋给另一个域类型，只要它们的基本类型是相容的。

当把**check**子句应用到域上时，允许模式设计者指定一个谓词，被声明为来自该域的任何变量都必须满足这个谓词。例如，**check**子句可以保证教师工资域中只允许出现大于给定值的值：

```
create domain YearlySalary numeric(8, 2)
  constraint salary_value_test check(value >= 29000.00);
```

*YearlySalary*域有一个约束来保证年薪大于或等于29 000.00美元。**constraint** *salary_value_test*子句是可选的，它用来将该约束命名为*salary_value_test*。系统用这个名字来指出一个更新违反了哪个约束。

作为另一个例子，使用**in**子句可以限定一个域只包含指定的一组值：

```
create domain degree_level varchar(10)
  constraint degree_level_test
    check (value in ('Bachelors', 'Masters', or 'Doctorate'));
```

[140]

在数据库实现中对类型和域的支持

尽管本节描述的**create type**和**create domain**结构是SQL标准的部分，但这里描述的这些结构形式还没有被大多数数据库实现完全支持。PostgreSQL支持**create domain**结构，但是其**create type**结构具有不同的语法和解释。

IBM DB2支持**create type**的一个版本，它使用**create distinct type**语法，但不支持**create domain**。微软的SQL Server实现了**create type**结构的一个版本，支持域约束，与SQL的**create domain**结构类似。

Oracle不支持在此描述的任何一种结构。然而，SQL还定义了一个更复杂的面向对象类型系统，我们将在后面第22章学习。通过使用不同形式的**create type**结构，Oracle、IBM DB2、PostgreSQL和SQL Server都支持面向对象类型系统。

4.5.6 create table的扩展

应用常常要求创建与现有的某个表的模式相同的表。SQL提供了一个**create table like**的扩展来支持这项任务：

```
create table temp_instructor like instructor;
```

上述语句创建了一个与*instructor*具有相同模式的新表*temp_instructor*。

当书写一个复杂查询时，把查询的结果存储成一个新表通常是很有用的；这个表通常是临时的。这里需要两条语句，一条用于创建表(具有合适的列)，另一条用于把查询结果插入到表中。SQL:2003提供了一种更简单的技术来创建包含查询结果的表。例如，下面的语句创建了表*t1*，该表包含一个查询的结果：

```
create table t1 as
  (select *
   from instructor
   where dept_name = 'Music')
with data;
```

在默认情况下，列的名称和数据类型是从查询结果中推导出来的。通过在关系名后面列出列名，可以给列显式指派名字。

正如 SQL:2003 标准所定义的，如果省略 **with data** 子句，表会被创建，但不会载入数据。但即使
141 在省略 **with data** 子句的情况下，很多数据库实现还是通过默认方式往表中加载了数据。注意几种数据库实现都用不同语法支持 **create table**… **like** 和 **create table**… **as** 的功能；请参考相应的系统手册以获得进一步细节。

上述 **create table**… **as** 语句与 **create view** 语句非常相似，并且都用查询来定义。两者主要的区别在于当表被创建时表的内容被加载，但视图的内容总是反映当前查询的结果。

4.5.7 模式、目录与环境

要理解模式和目录的形成，需要考虑文件系统中文件是如何命名的。早期的文件系统是平面的，也就是说，所有的文件都存储在同一个目录下。当然，当代的文件系统有一个目录（或者文件夹）结构，文件都存储在子目录下。要单独命名一个文件，我们必须指定文件的完整路径名。例如，/users/avi/db-book/chapter3.tex。

跟早期文件系统一样，早期数据库系统也只为所有关系提供一个命名空间。用户不得不相互协调以保证他们没有对不同的关系使用同样的名字。当代数据库系统提供了三层结构的关系命名机制。最顶层由**目录**（catalog）构成，每个目录都可以包含**模式**（schema）。诸如关系和视图那样的 SQL 对象都包含在模式中。（一些数据库实现用术语"数据库"代替术语"目录"）。

要在数据库上做任何操作，用户（或程序）都必须先连接到数据库。为了验证用户身份，用户必须提供用户名以及密码（通常情况下）。每个用户都有一个默认的目录和模式，这个组合对用户来说是唯一的。当一个用户连接到数据库系统时，将为该连接设置好默认的目录和模式。这对应于当用户登录进一个操作系统时，把当前目录设置为用户的主（home）目录。

为了唯一标识出一个关系，必须使用一个名字，它包含三部分，例如：

catalog5. *univ_schema.* *course*

当名字的目录部分被认为是连接的默认目录时，可以省略目录部分。这样如果 *catalog5* 是默认目录，我们可以用 *univ_schema.course* 来唯一标识上述关系。

如果用户想访问存在于另外的模式中的关系，而不是该用户的默认模式，那就必须指定模式的名字。然而，如果一个关系存在于特定用户的默认模式中，那么连模式的名字也可以省略。这样，如果 *catalog5* 是默认目录并且 *univ_schema* 是默认模式，我们可以只用 *course*。

当有多个目录和模式可用时，不同应用和不同用户可以独立工作而不必担心命名冲突。不仅如此，
142 一个应用的多个版本（一个产品版本，其他是测试版本）可以在同一个数据库系统上运行。

默认目录和模式是为每个连接建立的 SQL **环境**（SQL environment）的一部分。环境还包括用户标识（也称为授权标识符）。所有通常的 SQL 语句，包括 DDL 和 DML 语句，都在一个模式的环境中运行。

我们可以用 **create schema** 和 **drop schema** 语句来创建和删除模式。在大多数数据库系统中，模式还随着用户账户的创建而自动创建，此时模式名被置为用户账号名。模式要么建立在默认目录中，要么建立在创建用户账户时所指定的目录中。新创建的模式成为用户账户的默认模式。

创建和删除目录依据实现的不同而不同，这不是 SQL 标准中的一部分。

4.6 授权

我们可能会给一个用户在数据库的某些部分授予几种形式的权限。对数据的授权包括：

- 授权读取数据。
- 授权插入新数据。
- 授权更新数据。
- 授权删除数据。

每种类型的授权都称为一个**权限**（privilege）。我们可以在数据库的某些特定部分（如一个关系或视图）上授权给用户所有这些类型的权限，或者完全不授权，或者这些权限的一个组合。

当用户提交查询或更新时，SQL执行先基于该用户曾获得过的权限检查此查询或更新是否是授权过的。如果查询或更新没有经过授权，那么将被拒绝执行。

除了在数据上的授权之外，用户还可以被授予在数据库模式上的权限，例如，可以允许用户创建、修改或删除关系。拥有某些形式的权限的用户还可以把这样的权限转授（授予）给其他用户，或者撤销（收回）一种此前授出的权限。本节我们将学习每个这样的权限是如何用SQL来指定的。

最大的授权形式是被授予数据库管理员的。数据库管理员可以授权新用户，重构数据库，等等。这种权限方式和操作系统中的超级用户、管理员或操作员的权限是类似的。

4.6.1 权限的授予与收回

SQL标准包括**select**、**insert**、**update**和**delete**权限。权限**所有权限**（all privileges）可以用作所有允许权限的简写形式。一个创建了新关系的用户将自动被授予该关系上的所有权限。

SQL数据定义语言包括授予和收回权限的命令。**grant**语句用来授予权限。此语句的基本形式为： 142

```
grant <权限列表>
on <关系名或视图名>
to <用户/角色列表>;
```

权限列表使得一个命令可以授予多个权限。角色的概念将在后面4.6.2节讨论。

关系上的**select**权限用于读取关系中的元组。下面的**grant**语句授予数据库用户Amit和Satoshi在*department*关系上的**select**权限：

grant select on *department* **to** *Amit*, *Satoshi*;

该授权使得这些用户可以在*department*关系上执行查询。

关系上的**update**权限允许用户修改关系中的任意元组。**update**权限既可以在关系的所有属性上授予，又可以只在某些属性上授予。如果**grant**语句中包括**update**权限，将被授予**update**权限的属性列表可以出现在紧跟关键字**update**的括号中。属性列表是可选项，如果省略属性列表，则授予的是关系中所有属性上的update权限。

下面的**grant**语句授予用户Amit和Satoshi在*department*关系的*budget*属性上的更新权限：

grant update (*budget*) **on** *department* **to** *Amit*, *Satoshi*;

关系上的**insert**权限允许用户往关系中插入元组。**insert**权限也可以指定属性列表；对关系所作的任何插入必须只针对这些属性，系统将其余属性要么赋默认值（如果这些属性上定义了默认值），要么赋*null*。

关系上的**delete**权限允许用户从关系中删除元组。

用户名**public**指系统的所有当前用户和将来的用户。因此，对**public**的授权隐含着对所有当前用户和将来用户的授权。

在默认情况下，被授予权限的用户/角色无权把此权限授予其他用户/角色。SQL允许用授予权限来指定权限的接受者可以进一步把权限授予其他用户。我们将在4.6.5节详细讨论这个特性。 144

值得注意的是，SQL授权机制可以对整个关系或一个关系的指定属性授予权限。但是，它不允许对一个关系的指定元组授权。

我们使用**revoke**语句来收回权限。此语句的形式与**grant**几乎是一样的：

```
revoke <权限列表>
on <关系名或视图名>
from <用户/角色列表>;
```

因此，要收回前面我们所授予的那些权限，我们书写下列语句：

revoke select on *department* **from** *Amit*, *Satoshi*;
revoke update (budget) **on** *department* **from** *Amit*, *Satoshi*;

如果被收回权限的用户已经把权限授予了其他用户，权限的收回会更加复杂。我们将在4.6.5节回到这个问题。

4.6.2 角色

考虑在一个大学里不同人所具有的真实世界角色。每个教师必须在同一组关系上具有同种类型的权限。无论何时指定一位新的教师，就都必须被单独授予所有这些权限。

一种更好的方式是指明每个教师应该被授予的权限，并单独标示出哪些数据库用户是教师。系统可以利用这两条信息来确定每位教师的权限。当雇佣了一位新的教师时，必须给他分配一个用户标识符，并且必须将他标示为一位教师，而不需要重新单独授予教师权限。

角色(role)的概念运用了此观念。在数据库中建立一个角色集。可以给角色授予权限，就和给每个用户授权的方式完全一样。每个数据库用户被授予一组他有权扮演的角色(也可能是空的)。

在我们的大学数据库里，角色的例子可以包括 *instructor*、*teaching_assistant*、*student*、*dean* 和 *department_chair*。

另一个不是很合适的方法是建立一个 *instructor* 用户标识。允许每位教师用 *instructor* 用户标识来连接数据库。该方式的问题是它不可能鉴别出到底是哪位教师执行了数据库更新，从而导致安全隐患。使用角色的好处是需要用户用他们自己的用户标识来连接数据库。

145 任何可以授予给用户的权限都可以授予给角色。给用户授予角色就跟给用户授权一样。

在 SQL 中创建角色如下所示：

```
create role instructor;
```

然后角色就可以像用户那样被授予权限，如在这样的语句中：

```
grant select on takes
to instructor;
```

角色可以授予给用户，也可以授予给其他角色，如这样的语句：

```
grant dean to Amit;
create role dean;
grant instructor to dean;
grant dean to Satoshi;
```

因此，一个用户或一个角色的权限包括：

- 所有直接授予用户/角色的权限。
- 所有授予给用户/角色所拥有角色的权限。

注意可能存在着一个角色链；例如，角色 *teaching_assistant* 可能被授予所有的 *instructor*。接着，角色 *instructor* 被授予所有的 *dean*。这样，角色 *dean* 就继承了所有被授予给角色 *instructor* 和 *teaching_assistant* 的权限，还包括直接赋给 *dean* 的权限。

当一个用户登录到数据库系统时，在此会话中用户执行的动作拥有所有直接授予该用户的权限，以及所有(直接地或通过其他角色间接地)授予该用户所拥有角色的权限。这样，如果一个用户 Amit 被授予了角色 *dean*，用户 Amit 就拥有所有直接授予给 Amit 的权限，以及授予给 *dean* 的权限，再加上授予给 *instructor* 和 *teaching_assistant* 的权限，如果像上面那样，这些角色被(直接地或间接地)授予给角色 *dean* 的话。

值得注意的是，基于角色的授权概念并没有在 SQL 中指定，但在很多的共享应用中，基于角色的授权被广泛应用于存取控制。

4.6.3 视图的授权

在我们的大学例子中，考虑有一位工作人员，他需要知道一个给定系(比如 Geology 系)里所有员工的工资。该工作人员无权看到其他系中员工的相关信息。因此，该工作人员对 *instructor* 关系的直接访问必须被禁止。但是，如果他要访问 Geology 系的信息，就必须得到在一个视图上的访问权限，我们
146 称该视图为 *geo_instructor*，它仅由属于 Geology 系的那些 *instructor* 元组构成。该视图可以用 SQL 定义如下：

```
create view geo_instructor as
  (select *
   from instructor
   where dept_name = 'Geology');
```

假设该工作人员提出如下SQL查询：

```
select *
from geo_instructor;
```

显然，该工作人员有权看到此查询的结果。但是，当查询处理器将此查询转换为数据库中实际关系上的查询时，它产生了一个在 *instructor* 上的查询。这样，系统必须在开始查询处理以前，就检查该工作人员查询的权限。

创建视图的用户不需要获得该视图上的所有权限。他得到的那些权限不会为他提供超越他已有权限的额外授权。例如，如果一个创建视图的用户在用来定义视图的关系上没有 **update** 权限的话，那么他不能得到视图上的 **update** 权限。如果用户创建一个视图，而此用户在该视图上不能获得任何权限，系统会拒绝这样的视图创建请求。在我们的 *geo_instructor* 视图例子中，视图的创建者必须在 *instructor* 关系上具有 **select** 权限。

正如我们将在5.2节看到的那样，SQL支持创建函数和过程，在函数和过程中可以包括查询与更新。在函数或过程上可以授予 **execute** 权限，以允许用户执行该函数或过程。在默认情况下，和视图类似，函数和过程具有其创建者所拥有的所有权限。在效果上，该函数或过程的运行就像其被创建者调用了那样。

尽管此行为在很多情况下是恰当的，但是它并不总是恰当的。从SQL:2003开始，如果函数定义有一个额外的 **sql security invoker** 子句，那么它就在调用该函数的用户的权限下执行，而不是在函数定义者的权限下执行。这就允许创建的函数库能够在与调用者相同的权限下运行。

4.6.4 模式的授权

SQL标准为数据库模式指定了一种基本的授权机制：只有模式的拥有者才能够执行对模式的任何修改，诸如创建或删除关系，增加或删除关系的属性，以及增加或删除索引。 147

然而，SQL提供了一种 **references** 权限，允许用户在创建关系时声明外码。SQL的 **references** 权限可以与 **update** 权限类似的方式授予到特定属性上。下面的 **grant** 语句允许用户 Mariano 创建这样的关系，它能够参照 *department* 关系的码 *dept_name*：

```
grant references (dept_name) on department to Mariano;
```

初看起来，似乎没有理由不允许用户创建参照了其他关系的外码。但是，回想一下外码约束限制了被参照关系上的删除和更新操作。指定 Mariano 在关系 *r* 中创建了一个外码，它参照 *department* 关系的 *dept_name* 属性，然后在 *r* 中插入一条属于 Geology 系的元组。那么就再也不可能从 *department* 关系中将 Geology 系删除，除非同时也修改关系 *r*。这样，Mariano 定义的外码限制了其他用户将来的行为；因此，需要有 **references** 权限。

继续使用 *department* 关系的例子，如果要创建关系 *r* 上的 **check** 约束，并且该约束有参照 *department* 的子查询，那么还需要有 *department* 上的 references 权限。其原因与外码约束的情况类似，因为参照了一个关系的 check 约束限制了对该关系可能的更新。

4.6.5 权限的转移

获得了某些形式授权的用户可能被允许将此授权传递给其他用户。在默认方式下，被授予权限的用户/角色无权把得到的权限再授予给另外的用户/角色。如果我们在授权时允许接受者把得到的权限再传递给其他用户，我们可以在相应的 **grant** 命令后面附加 **with grant option** 子句。例如，如果我们希望授予 Amit 在 *department* 上的 **select** 权限，并且允许 Amit 将该权限授予给其他用户，我们可以写：

```
grant select on department to Amit with grant option;
```

一个对象（关系/视图/角色）的创建者拥有该对象上的所有权限，包括给其他用户授权的权限。

作为一个例子，考虑大学数据库中 *teaches* 关系上更新权限的授予。假设最初数据库管理员将

teaches 上的更新权限授给用户 U_1、U_2 和 U_3。他们接下来又可以将这一授权传递给其他用户。指定权限从一个用户到另一个用户的传递可以表示为**授权图**(authorization graph)。该图中的顶点是用户。

考虑 *teaches* 上更新权限所对应的授权图。如果用户 U_i 将 *teaches* 上的更新权限授给 U_j，则图中包含边 $U_i \rightarrow U_j$。图的根是数据库管理员。在图 4-10 所示的示例图中，注意 U_1 和 U_2 都给用户 U_3 授权了；而 U_3 只从 U_1 处获得了授权。

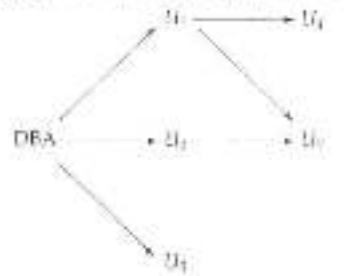


图 4-10 权限授予图(U_1，U_2，…，U_5 是用户，DBA 代表数据库管理员)

用户具有权限的充分必要条件是：当且仅当存在从授权图的根(即代表数据库管理员的顶点)到代表该用户顶点的路径。

4.6.6 权限的收回

假设数据库管理员决定收回用户 U_1 的授权。由于 U_4 从 U_1 处获得过授权，因此其权限也应该被收回。可是，U_4 既从 U_1 处又从 U_2 处获得过授权。由于数据库管理员没有从 U_2 处收回 *teaches* 上的更新权限，U_4 继续拥有 *teaches* 上的更新权限。如果 U_2 最终从 U_3 处收回授权，则 U_3 失去权限。

一对狡猾的用户可能企图通过相互授权来破坏权限收回规则。例如，如果 U_2 最初由数据库管理员授予了一种权限，U_2 进而把此权限授予给 U_3。假设 U_3 现在把此权限授回给 U_2。如果数据库管理员从 U_2 收回权限，看起来好像 U_2 保留了通过 U_3 获得的授权。然而，注意一旦管理员从 U_2 收回权限，在授权图中就不存在从根到 U_2 或 U_3 的路径了。这样，SQL 保证从这两个用户那里都收回了权限。

正如我们刚才看到的那样，从一个用户/角色那里收回权限可能导致其他用户/角色也失去该权限。这一行为称作级联收回。在大多数的数据库系统中，级联是默认行为。然而，**revoke** 语句可以申明 **restrict** 来防止级联收回：

revoke select on *department* **from** Amit, Satoshi **restrict**;

148
|
149

这种情况下，如果存在任何级联收回，系统就返回一个错误，并且不执行收权动作。

可以用关键字 **cascade** 来替换 **restrict**，以表示需要级联收回；然而，**cascade** 可以省略，就像我们前述例子中的那样，因为它是默认行为。

下面的 **revoke** 语句仅仅收回 grant option，而并不是真正收回 **select** 权限：

revoke grant option for select on *department* **from** Amit;

注意一些数据库实现不支持上述语法；它们采用另一种方式：收回权限本身，然后不带 grant option 最新授权。

级联收回在许多情况下是不合适的。假定 Satoshi 具有 *dean* 角色，他将 *instructor* 授给 Amit，后来 *dean* 角色从 Satoshi 收回(也许由于 Satoshi 离开了大学)；Amit 继续被雇佣为教职工，并且还应该保持 *instructor* 角色。

为了处理以上情况，SQL 允许权限由一个角色授予，而不是由用户来授予。SQL 有一个与会话所关联的当前角色的概念。默认情况下，一个会话所关联的当前角色是空的(某些特殊情况除外)。一个会话所关联的当前角色可以通过执行 **set role** *role_name* 来设置。指定的角色必须已经授予给用户，否则 **set role** 语句执行失败。

如果要在授予权限时将授权人设置为一个会话所关联的当前角色，并且当前角色不为空的话，我们可以在授权语句后面加上

granted by current_role

子句。

假设将角色 *instructor*(或其他权限)授给 Amit 是用 **granted by current_role** 子句实现的，当前角色被设置为 *dean* 而不是授权人(用户 Satoshi)，那么，从 Satoshi 处收回角色/权限(包括角色 *dean*)就不会导致收回以角色 *dean* 作为授权人所授予的权限，即使 Satoshi 是执行该授权的用户；这样，即使在 Satoshi 的权限被收回后，Amit 仍然能够保持 *instructor* 角色。

4.7 总结

- SQL支持包括内连接、外连接在内的几种连接类型，以及几种形式的连接条件。
- 视图关系可以定义为包含查询结果的关系。视图是有用的，它可以隐藏不需要的信息，可以把信息从多个关系收集到一个单一的视图中。 150
- 事务是一个查询和更新的序列，它们共同执行某项任务。事务可以被提交或回滚。当一个事务被回滚，该事务执行的所有更新所带来的影响将被撤销。
- 完整性约束保证授权用户对数据库所做的改变不会导致数据一致性的破坏。
- 参照完整性约束保证出现在一个关系的给定属性集上的值同样出现在另一个关系的特定属性集上。
- 域约束指定了在一个属性上可能取值的集合。这种约束也可以禁止在特定属性上使用空值。
- 断言是描述为表达式，它指定了我们要求总是为真的谓词。
- SQL数据定义语言提供对定义诸如 **date** 和 **time** 那样的附有域类型以及用户定义域类型的支持。
- 通过SQL授权机制，可以按照在数据库中不同数据值上数据库用户所允许的访问类型对他们进行区分。
- 获得了某种形式授权的用户可能允许将此授权传递给其他用户。但是，对于权限怎样在用户间传递我们必须很小心，以保证这样的权限在将来的某个时候可以被收回。
- 角色有助于根据用户在组织机构中所扮演的角色，把一组权限分配给用户。

术语回顾

- 连接类型
  - 内连接和外连接
  - 左外连接、右外连接和全外连接
  - natural 连接条件、using 连接条件和 on 连接条件
- 视图定义
- 物化视图
- 视图更新
- 事务
  - 提交
  - 回滚
  - 原子事务
- 完整性约束
- 域约束
- 唯一性约束
- check 子句
- 参照完整性
  - 级联删除
  - 级联更新
- 断言
- 日期和时间类型
- 默认值
- 索引
- 大对象
- 用户定义类型
- 域
- 目录
- 模式
- 授权
- 权限
  - 选择
  - 插入
  - 更新
  - 所有权限
  - 授予权限
  - 收回权限
  - 授予权限的权限
  - grant option
- 角色
- 视图授权
- 执行授权
- 调用者权限
- 行级授权

实践习题

4.1 用SQL写出下面的查询：

a. 显示所有教师的列表，列出他们的ID，姓名以及所讲授课程段的编号。对于没有讲授任何课程段的教师，确保将课程段编号显示为0。在你的查询中应该使用外连接，不能使用标量子查询。

b. 使用标量子查询，不使用外连接写出上述查询。

c. 显示2010年春季开设的所有课程的列表，包括讲授课程段的教师的姓名。如果一个课程段有不止一位教师讲授，那么有多少位教师，此课程段在结果中就出现多少次。如果一个课程段没有任何教师，它也要出现在结果中，相应的教师名置为"-"。

d. 显示所有系的列表，包括每个系中教师的总数，不能使用标量子查询。确保正确处理没有教师的系。

4.2 不使用SQL外连接运算也可以在SQL中计算外连接表达式。为了阐明这个事实，不使用外连接表达式重写下面每个SQL查询：

a. **select** * **from** *student* **natural left outer join** *takes*
b. **select** * **from** *student* **natural full outer join** *takes*

151 / 152 4.3 假设有三个关系 $r(A, B)$、$s(B, C)$ 和 $t(B, D)$。其中所有属性声明为非空。考虑表达式：

- r **natural left outer join** (s **natural left outer join** t)
- (r **natural left outer join** s) **natural left outer join** t

a. 给出关系 r、s 和 t 的实例，使得在第二个表达式的结果中，属性 C 有一个空值但属性 D 有非空值。
b. 在第一个表达式的结果中，上述模式中的 C 为空且 D 非空有可能吗？解释原因。

4.4 **测试 SQL 查询**(testing SQL query)：为了测试一个用文字表达的查询能否正确地用 SQL 写出，通常 SQL 查询会在多个测试数据库上执行，并用人工来检测在每个测试数据库上 SQL 查询的结果是否与用文字表达的意图相匹配。

a. 在 3.3.3 节我们见过一个错误的 SQL 查询，它希望找出每位教师讲授了哪些课程；该查询计算 *instructor*、*teaches* 和 *course* 的自然连接，结果是无意地造成了让 *instructor* 和 *course* 的 *dept_name* 属性取值相等。给出一个数据集样例，能有助于发现这种特别的错误。
b. 在创建测试数据库时，对每个外码来说，在被参照关系中创建与参照关系中任何元组都不匹配的元组是很重要的。请用大学数据库上的一个查询样例来解释原因。
c. 在创建测试数据库时，假若外码属性可空的话，在外码属性上创建具有空值的元组是很重要的(SQL 允许在外码属性上取空值，只要它们不是主码的一部分，并且没有被声明为 **not null**)。请用大学数据库上的一个查询样例来解释原因。

提示：使用来自习题 4.1 中的查询。

4.5 基于习题 3.2 中的查询，指出如何定义视图 *student_grades*(*ID*, *GPA*)，它给出了每个学生的等级分值的平均值，回忆一下，我们用关系 *grade_points*(*grade*, *points*) 来把用字母表示的成绩等级转换为用数字表示的得分。你的视图定义要确保能够正确处理在 *takes* 关系的 *grade* 属性上取 *null* 值的情况。

153 4.6 完成图 4-8 中的大学数据库的 SQL DDL 定义以包括 *student*、*takes*、*advisor* 和 *prereq* 关系。

4.7 考虑图 4-11 所示的关系数据库。给出这个数据库的 SQL DDL 定义。指出应有的参照完整性约束，并把它们包括在 DDL 定义中。

employee(*employee_name*, *street*, *city*)
works(*employee_name*, *company_name*, *salary*)
company(*company_name*, *city*)
manages(*employee_name*, *manager_name*)

图 4-11 习题 4.7 和习题 4.12 的雇员数据库

4.8 正如在 4.4.7 节所讨论的，我们希望能够满足约束"每位教师不能在同一个学期的同一个时间段在两个不同的教室授课"。

a. 写出一个 SQL 查询，它返回违反此约束的所有(*instructor*, *section*)组合。
b. 写出一个 SQL 断言来强制实现此约束(正如在 4.4.7 节所讨论的那样，当代的数据库系统不支持这样的断言，尽管它们是 SQL 标准的一部分)。

4.9 SQL 允许外码依赖指向同一个关系，如下面的例子所示：

```
create table manager
    (employee_name  var char(20) not null
    manager_name  var char(20) not null,
    primary key employee_name,
    foreign key (manager_name) references manager
    on delete cascade)
```

这里 *employee_name* 是 *manager* 表的码，意味着每个雇员最多只有一个经理。外码子句要求每个经理都是一个雇员。请准确解释当 *manager* 关系中一个元组被删除时会发生什么情况。

4.10 SQL 提供一个称为 **coalesce** 的 n 维数组运算。其定义如下：**coalesce**(A_1, A_2, …, A_n)返回序列 A_1, A_2, …, A_n 中第一个非空值 A_i，如果所有的 A_1, A_2, …, A_n 个为 *null*，则返回 *null*。

假设 a 和 b 是模式分别为 $A(name, address, title)$ 和 $B(name, address, salary)$ 的关系。如何用带 **on** 条件和 **coalesce** 运算的**全外连接**(full outer-join)运算来表达 a **natural full outer join** b。要保证结果关系中不包含属性 *name* 和 *address* 的两个副本，并且即使 a 和 b 中的某些元组在属性 *name* 和 *address* 上取空值的情况下，所给出的解决方案仍然是正确的。

4.11 一些研究者提出带标记的(*marked*)空值的概念。带标记空值 $\perp_1$ 与它自身相等，但是如果 $i \neq j$，那么 $\perp_i \neq \perp_j$。带标记空值的一个应用是允许通过视图进行特定的更新。考虑视图 *instructor_info*(4.2节)，如何利用带标记空值来允许通过 *instructor_info* 插入元组(99999，"Johnson"，"Taylor")。

习题

4.12 对于图4-11中的数据库，写出一个查询来找到那些没有经理的雇员。注意一个雇员可能只是没有列出其经理，或者可能有 *null* 经理。使用外连接书写查询，然后不用外连接再重写查询。

4.13 在什么情况下，查询

```
select *
from student natural full outer join takes natural full outer join course
```

将包含在属性 *title* 上取空值的元组？

4.14 给定学生每年修到的学分总数，如何定义视图 *tot_credits*(*year*, *num_credits*)。

4.15 给出如何用 **case** 运算表达习题4.10中的 **coalesce** 运算。

4.16 如本章定义的参照完整性约束正好涉及两个关系。考虑包括如图4-12所示关系的数据库。假设我们希望要求每个出现在 *address* 中的名字必须出现在 *salaried_worker* 或者 *hourly_worker* 中，但不一定要求在两者中同时出现。

a. 给出表达这种约束的语法。

b. 讨论为了使这种形式的约束生效，系统必须采取什么行动。

salaried_worker (*name*, *office*, *phone*, *salary*)
hourly_worker (*name*, *hourly_wage*)
address (*name*, *street*, *city*)

图4-12 习题4.16的雇员数据库

4.17 当一个经理如 Satoshi 授予权限的时候，授权应当由经理角色完成，而不是由用户 Satoshi 完成。解释其原因。

4.18 假定用户 A 拥有关系 r 上的所有权限，该用户把关系 r 上的查询权限以及授予该权限的权限授予给 **public**。假定用户 B 将 r 上的查询权限授予 A。这是否会导致授权图中的环？解释原因。

4.19 将每个关系存放在一个单独的操作系统文件中的数据库系统可能使用操作系统的授权机制，而不是自己定义一种专门的机制。请论述这种方法的一个优点和一个缺点。

文献注解

对于SQL参考资料请参考第3章的文献注解。

SQL用于决定视图更新能力的规则，以及更新操作如何影响底层的数据库关系，都定义在SQL:1999标准中。Melton 和 Simon[2001]中对此进行了总结。

第5章
Database System Concepts, 6E

高级 SQL

在第3章和第4章我们详细地介绍了 SQL 的基本结构。在本章中我们将介绍 SQL 的一些高级特性。[⊖]本章首先介绍如何使用通用程序设计语言来访问 SQL，这对于构造用数据库存取数据的应用有重要意义。我们将介绍两种在数据库中执行程序代码的方法：一种是通过扩展 SQL 语言来支持程序的操作；另一种是在数据库中执行程序语言中定义的函数。接下来本章将介绍触发器，用于说明当特定事件（例如在某个表上进行元组插入、删除或更新操作）发生时自动执行的操作。然后本章将讨论递归查询和 SQL 支持的高级聚集特性。最后，我们将对联机分析处理（OLAP）系统加以介绍，它可用于海量数据的交互分析。

5.1 使用程序设计语言访问数据库

SQL 提供了一种强大的声明性查询语言。实现相同的查询，用 SQL 写查询语句比用通用程序设计语言要简单得多。然而，数据库程序员必须能够使用通用程序设计语言，原因至少有以下两点：

- 因为 SQL 没有提供通用程序设计语言那样的表达能力，所以 SQL 并不能表达所有查询要求。也就是说，有可能存在这样的查询，可以用 C、Java 或 Cobol 编写，而用 SQL 做不到。要写这样的查询，我们可以将 SQL 嵌入到一种更强大的语言中。
- 非声明性的动作（例如打印一份报告、和用户交互，或者把一次查询的结果送到一个图形用户界面中）都不能用 SQL 实现。一个应用程序通常包括很多部分，查询或更新数据只是其中之一，而其他部分则用通用程序设计语言实现。对于集成应用来说，必须用某种方法把 SQL 与通用编程语言结合起来。

可以通过以下两种方法从通用编程语言中访问 SQL：

- 动态 SQL：通用程序设计语言可以通过函数（对于过程式语言）或者方法（对于面向对象的语言）来连接数据库服务器并与之交互。利用动态 SQL 可以在运行时以字符串形式构建 SQL 查询，提交查询，然后把结果存入程序变量中，每次一个元组。动态 SQL 的 SQL 组件允许程序在运行时构建和提交 SQL 查询。

  在这一章中，我们将介绍两种用于连接到 SQL 数据库并执行查询和更新的标准。一种是 Java 语言的应用程序接口 JDBC（5.1.1 节）。另一种是 ODBC（5.1.2 节），它最初是为 C 语言开发的，后来扩展到其他语言如 C++、C#和 Visual Basic。
- 嵌入式 SQL：与动态 SQL 类似，嵌入式 SQL 提供了另外一种使程序与数据库服务器交互的手段。然而，嵌入式 SQL 语句必须在编译时全部确定，并交给预处理器。预处理程序提交 SQL 语句到数据库系统进行预编译和优化，然后它把应用程序中的 SQL 语句替换成相应的代码和函数，最后调用程序语言的编译器进行编译。5.1.3 节涵盖嵌入式 SQL 的内容。

把 SQL 与通用程序语言相结合的主要挑战是：这些语言处理数据的方式互不兼容。在 SQL 中，数据的主要类型是关系。SQL 语句在关系上进行操作，并返回关系作为结果。程序设计语言通常一次操

⊖ 注意关于章节的先后顺序：数据库设计（第7章和第8章）可以脱离本章独立学习，完全可以先学习数据库设计，再读本章内容。然而，对于强调编程能力的课程而言，在学习了5.1节之后可以做更多的实验练习，所以我们建议在学习数据库设计之前先掌握这部分的内容。

作一个变量，这些变量大致相当于关系中一个元组的一个属性的值。因此，为了在同一应用中综合这两类语言，必须提供一种转换机制，使得程序语言可以处理查询的返回结果。

5.1.1 JDBC

JDBC标准定义了Java程序连接数据库服务器的**应用程序接口**(Application Program Interface，API)(JDBC原来是**Java数据库连接**(Java Database Connectivity)的缩写，但其全称现在已经不用了)。 157 158

图5-1给出了一个利用JDBC接口的Java程序的例子。它向我们演示了如何打开数据库连接，执行语句，处理结果，最后关闭连接。我们将在本节详细讨论这个实例。注意，Java程序必须引用java.sql.*，它包含了JDBC所提供功能的接口定义。

```
public static void JDBCexample(String userid, String passwd)
{
    try
    {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
                    "jdbc:oracle:thin:@db.yale.edu:1521:univdb",
                    userid, passwd);
        Statement stmt = conn.createStatement();
        try {
            stmt.executeUpdate(
                    "insert into instructor values('77987', 'Kim', 'Physics', 98000)");
        } catch (SQLException sqle)
        {
            System.out.println("Could not insert tuple. " + sqle);
        }
        ResultSet rset = stmt.executeQuery(
                    "select dept_name, avg (salary) "+
                    " from instructor "+
                    " group by dept_name");
        while (rset.next()) {
            System.out.println(rset.getString("dept_name") + " " +
                        rset.getFloat(2));
        }
        stmt.close();
        conn.close();
    }
    catch (Exception sqle)
    {
        System.out.println("Exception : " + sqle);
    }
}
```

图5-1 JDBC代码示例

5.1.1.1 连接到数据库

要在Java程序中访问数据库，首先要打开一个数据库连接。这一步需要选择要使用哪个数据库，例如，可以是你的机器上的一个Oracle实例，也可以是运行在另一台机器上的一个PostgreSQL数据库。只有在打开数据库连接以后，Java程序才能执行SQL语句。 159

可以通过调用DriverManager类(在java.sql包中)的getConnection方法来打开一个数据库连接。该方法有三个参数。[1]

- getConnection方法的第一个参数是以字符串类型表示的URL，指明服务器所在的主机名称(在我们的例子里是db.yale.edu)以及可能包含的其他信息，例如，与数据库通信所用的协议(在我们的例子里是jdbc:oracle:thin:，我们马上就会看到为什么需要它)，数据库系统用来通信的端口号(在我们的例子中是2000)，还有服务器端使用的特定数据库(在我们的例子中是univdb)。注意JDBC只是指定API而不指定通信协议，一个JDBC驱动器可能支持多种协议，

[1] 有多种版本的getConnection函数，它们所接受的参数各不相同，我们只介绍其中最常用的一个。

我们必须指定一个数据库和驱动器都支持的协议。协议的详细内容是由提供商设定的。

- getConnection 方法的第二个参数用于指定一个数据库用户标识，它为字符串类型。
- 第三个参数是密码，它也是字符串类型。（注意，把密码直接写在 JDBC 代码中会增加安全性隐患，因为你的代码有可能会被某些未被授权的用户所访问。）

在图 5-1 中的例子中，我们已经建立了一个 Connection 对象，其句柄是 conn。

每个支持 JDBC（大多数的数据库提供商都支持）的数据库产品都会提供一个 JDBC 驱动程序（JDBC driver），该驱动程序必须被动态加载才能实现 Java 对数据库的访问。事实上，必须在连接数据库之前完成驱动程序的加载。

在图 5-1 中程序的第一行调用 Class.forName 函数完成驱动程序的加载，在调用时需要通过参数来指定一个实现了 java.sql.Driver 接口的实体类。这个接口的功能是为了实现不同层面的操作之间的转换，一边是与产品类型无关的 JDBC 操作，另一边是与产品相关的、在所使用的特定数据库管理系统中完成的操作。图中的实例采用了 Oracle 的驱动程序，oracle.jdbc.driver.OracleDriver [⊖]。该驱动程序包含在一个 jar 文件里，可以从提供商的网站下载，然后放在 Java 的类路径（classpath）里，用于 Java 编译器访问。

用来与数据库交换信息的具体协议并没有在 JDBC 标准中定义，而是由所使用的驱动程序决定的。有些驱动程序支持多种协议，使用哪一种更合适取决于所连接的数据库支持什么协议。我们的例子里，在打开一个数据库连接时，字符串 jdbc:oracle:thin: 指定了 Oracle 支持的一个特定协议。

5.1.1.2 向数据库系统中传递 SQL 语句

一旦打开了一个数据库连接，程序就可以利用该连接来向数据库发送 SQL 语句用于执行。这是通过 Statement 类的一个实例来完成的。一个 Statement 对象并不代表 SQL 语句本身，而是实现了可以被 Java 程序调用的一些方法，通过参数来传递 SQL 语句并被数据库系统所执行。我们的例子在连接变量 conn 上创建了一个 Statement 句柄（stmt）。

我们既可以使用 executeQuery 函数又可以用 executeUpdate 函数来执行一条语句。这取决于这条 SQL 语句是查询语句（如果是查询语句，自然会返回一个结果集），还是像更新（update）、插入（insert）、删除（delete）、创建表（create table）等这样的非查询性语句。在我们的例子里，stmt.executeUpdate 执行了一条更新语句，向 instructor 关系中插入数据。它返回一个整数，表示被插入、更新或者删除的元组个数。对于 DDL 语句，返回值是 0。try {…} catch {…} 结构让我们可以捕捉 JDBC 调用产生的异常（错误情况），并显示给用户适当的出错信息。

5.1.1.3 获取查询结果

示例程序用 stmt.executeQuery 来执行一次查询。它可以把结果中的元组集合提取到 ResultSet 对象变量 rset 中并每次取出一个进行处理。结果集的 next 方法用来查看在集合中是否还存在至少一个尚未取回的元组，如果存在的话就取出。next 方法的返回值是一个布尔变量，表示是否从结果集中取回了一个元组。可以通过一系列的名字以 get 为前缀的方法来得到所获取元组的各个属性。方法 getString 可以返回所有的基本 SQL 数据类型的属性（被转换成 Java 中的 String 类型的值），当然也可以使用像 getFloat 那样一些约束性更强的方法。这些不同的 get 方法的参数既可以是一个字符串类型的属性名称，又可以是一个整数，用来表示所需获取的属性在元组中的位置。图 5-1 给出了两种在元组中提取属性值的办法：利用属性名提取（dept_name）或者利用属性位置提取（2，代表第二个属性）。

Java 程序结束的时候语句和连接都将被关闭。注意关闭连接是很重要的，因为数据库连接的个数是有限制的；未关闭的连接可能导致超过这一限制。如果发生这种情况，应用将不能再打开任何数据库连接。

⊖ 其他产品的类似的驱动名称如：IBM DB2：com.ibm.db2.jcc.app.DB2Driver；Microsoft SQL Server：com.microsoft.sqlserver.jdbc.SQLServerDriver；PostgreSQL：org.postgresql.Driver；MySQL：com.mysql.jdbc.Driver。Sun 公司还提供了一种"桥接驱动器"，可以把 JDBC 调用转换成 ODBC。该驱动仅适用于那些支持 ODBC 但不支持 JDBC 的厂商。

5.1.1.4 预备语句

我们也可以通过以"?"来代表以后再给出的实际值，而创建一个预备语句。数据库系统在准备了查询语句的时候对它进行编译。在每次执行该语句时(用新值替换"?")，数据库可以重用预先编译的查询的形式，应用新值进行查询。图 5-2 的代码框架给出了如何使用预备语句的示例。

```
PreparedStatement pStmt = conn.prepareStatement(
                "insert into instructor values(?,?,?,?)");
pStmt.setString(1, "88877");
pStmt.setString(2, "Perry");
pStmt.setString(3, "Finance");
pStmt.setInt(4, 125000);
pStmt.executeUpdate();
pStmt.setString(1, "88878");
pStmt.executeUpdate();
```

图 5-2 JDBC 代码中的预备语句

可以使用 Connection 类的 prepareStatement 方法来提交 SQL 语句用于编译。它返回一个 PreparedStatement 类的对象。此时还没有执行 SQL 语句。执行需要 PreparedStatement 类的两个方法 executeQuery 和 executeUpdate。但是在它们被调用之前，我们必须使用 PreparedStatement 类的方法来为"?"参数设定具体的值。setString 方法以及诸如 setInt 等用于其他的 SQL 基本类型的其他类似的方法使我们能够为参数指定值。第一个参数用来确定我们为哪个"?"设定值(参数的第一个值是 1，区别于大多数的 Java 结构的从 0 开始)。第二个参数是我们要设定的值。

在图 5-2 中的例子里，我们预备一个 **insert** 语句，设定"?"参数，并且调用 executeUpdate。例子中的最后两行显示，参数设定保持不变，直到我们特别地进行重新设定。这样，最后的语句调用 executeUpdate，元组("88878"，"Perry"，"Finance"，125000)被插入到数据库。

在同一查询编译一次然后设置不同的参数值执行多次的情况下，预备语句使得执行更加高效。然而，预备语句有一个更加重要的优势，它使得只要使用了用户输入值，即使是只运行一次，预备语句都是执行 SQL 查询的首选方法。假设我们读取了一个用户输入的值，然后使用 Java 的字符串操作来构造 SQL 语句。如果用户输入了某些特殊字符，例如一个单引号，除非我们采取额外工作对用户输入进行检查，否则生成的 SQL 语句会出现语法错误。setString 方法为我们自动完成检查，并插入需要的转义字符，以确保语法的正确性。

在我们的例子中，假设用户已经输入了 *ID*、*name*、*dept_name* 和 *salary* 这些变量的值，相应的元组将被插入到关系 *instructor* 中。假设我们不用预备语句，而是使用如下的 Java 表达式把字符串连接起来构成查询：

```
"insert into instructor values( ' " + ID + " ', ' " + name + " ',
    ' " + " ' + dept_name + " ', ' " + salary + " ')"
```

并且查询通过 Statement 对象的 executeQuery 方法被直接执行。现在，如果用户在 ID 或者 name 域中敲入一个单引号，查询语句就会出现语法错误。一个教员的名字很有可能带有引号(例如"O'Henry")。

160
|
162

也许以上的例子会被认为是一个小问题，而某些情况会比这糟糕得多。一种叫做 SQL 注入(SQL injection)的技术可以被恶意黑客用来窃取数据或损坏数据库。

假设一个 Java 程序输入一个字符串 *name*，并且构建下面的查询：

```
"select * from instructor where name = ' " + name + " ' "
```

如果用户没有输入一个名字，而是输入：

```
X' or 'Y' = 'Y
```

这样，产生的语句就变成：

```
"select * from instructor where name = ' " + "X' or 'Y' = 'Y" + " ' "
```

即

```
select * from instructor where name = 'X' or 'Y' = 'Y'
```

在生成的查询中，**where** 子句总是真，所以查询结果返回整个 instructor 关系。更诡计多端的恶意用户甚至可以编写输入值以输出更多的数据。使用预备语句就可以防止这类问题，因为输入的字符串将被插入转义字符，因此最后的查询变为：

```
"select" from instructor where name = 'X\' or \'Y\' = \'Y'
```

这是无害的查询语句，返回结果为空集。

比较老的系统允许多个由分号隔开的语句在一次调用里被执行。此功能正逐渐被淘汰，因为恶意
163 的黑客会利用 SQL 注入技术插入整个 SQL 语句。由于这些语句在 Java 程序所有者的权限上运行，像删除表(drop table)这样毁灭性的 SQL 语句会被执行。SQL 应用程序开发者必须警惕这种潜在的安全漏洞。

5.1.1.5 可调用语句

JDBC 还提供了 CallableStatement 接口来允许调用 SQL 的存储过程和函数(稍后在 5.2 节描述)。此接口对函数和过程所扮演的角色跟 prepareStatement 对查询所扮演的角色一样：

```
CallableStatement cStmt1 = conn.prepareCall("{ ? = call some_function(?) }");
CallableStatement cStmt2 = conn.prepareCall("{ call some_procedure(?, ?) }");
```

函数返回值和过程的对外参数的数据类型必须先用方法 registerOutParameter() 注册，它们可以用与结果集用的方法类似的 get 方法获取。请参看 JDBC 手册以获得更细节的信息。

5.1.1.6 元数据特性

正如我们此前提到的，一个 Java 应用程序不包含数据库中存储的数据的声明。这些声明是 SQL 数据定义语言(DDL)的一部分。因此，使用 JDBC 的 Java 程序必须要么将关于数据库模式的假设硬编码到程序中，要么直接在运行时从数据库系统中得到那些信息。后一种方法更可取，因为它使得应用程序可以更健壮地处理数据库模式的变化。

回想一下，当我们提交一个使用 executeQuery 方法的查询时，查询结果被封装在一个 ResultSet 对象中。接口 ResultSet 有一个 getMetaData() 方法，它返回一个包含结果集元数据的 ResultSetMetaData 对象。ResultSetMetaData 进一步又包含查找元数据信息的方法，例如结果的列数、某个特定列的名称，或者某个特定列的数据类型。这样，即使不知道结果的模式，我们也可以方便地执行查询。

下面的 Java 代码片段使用 JDBC 来打印出一个结果集中所有列的名称和类型。代码中的变量 rs 假定是执行查询后所获得的一个 ResultSet 实例。

```
ResultSetMetaData rsmd = rs.getMetaData();
for(int i = 1; i <= rsmd.getColumnCount(); i++) {
    System.out.println(rsmd.getColumnName(i));
    System.out.println(rsmd.getColumnTypeName(i));
}
```

getColumnCount 方法返回结果关系的元数(属性个数)。这使得我们能够遍历每个属性(请注意，和
164 JDBC 的惯例一致，我们从 1 开始)。对于每一个属性，我们采用 getColumnName 和 getColumnType-Name 两个方法分别得到它的名称和数据类型。

DatabaseMetaData 接口提供了查找数据库元数据的机制。接口 Connection 包含一个 getMeta-Data 方法用于返回一个 DatabaseMetaData 对象。接口 DatabaseMetaData 进一步又含有大量的方法可以用于获取程序所连接的数据库和数据库系统的元数据。

例如，有些方法可以返回数据库系统的产品名称和版本号。另外一些方法可以用来查询数据库系统所支持的特性。

还有其他可以返回数据库本身信息的方法。图 5.3 中的代码展示了如何找出数据库中的关系的列(属性)信息。变量 conn 假定存储了一个已经打开的数据库连接。方法 getColumns 有四个参数：一个目录名称(null 表示目录名称被忽略)、一个模式名称模板、一个表名称模板，以及一个列名称模板。模式名称、表名称和列名称的模板可以用于指定一个名字或一个模板。模板可以使用 SQL 字符串匹配特殊字符如"%"和"_"；例如模板"%"匹配所有的名字。只有满足特定名称或模板的模式中的表的列被检索到。结果集中的每行包含一个列的信息。结果集中的行包括若干个列，如目录名称、模式、表和列、列的类型、等等。

```
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rs = dbmd.getColumns(null, "univdb", "department", "%");
    // getColumns 的参数:关系,模式名,表名,列名
    // 返回值:每列返回一行,包含一系列属性,例如:COLUMN_NAME, TYPE_NAME
while(rs.next()) {
    System.out.println(rs.getString("COLUMN_NAME"),
        rs.getString("TYPE_NAME");
}
```

图 5.3 在 JDBC 中使用 DatabaseMetaData 查找列信息

DatabaseMetaData 还提供了获取数据库信息的一些其他方法，比如，可以用来获取关系（getTables()）、外码参照（getCrossReference()）、授权、数据库限制如最大连接数等的元数据。

元数据接口可以用于许多不同的任务。例如，它们可以用于编写数据库的浏览器。该浏览器允许用户找出一个数据库中的关系表，检查它们的模式，检查表中的行，用选择操作查看想要的行，等等。元数据信息可以用于使这些任务的代码更通用：例如，用来显示一个关系中的行的代码可以用这样的 165
方法编写使得它能够在所有可能的关系上工作，无论这些关系的模式是什么。类似地，可以编写这样的代码，它获得一个查询字符串，执行查询，然后把结果打印成一个格式化的表；无论提交了的实际查询是什么，这段代码都可以工作。

5.1.1.7 其他特性

JDBC 提供了一些其他特性，如**可更新的结果集**（updatable result sets）。它可以从一个在数据库关系上执行选择和/或投影操作的查询中创建一个可更新的结果集。然后，一个对结果集中的元组的更新将引起对数据库关系中相应元组的更新。

回想一下 4.3 节，事务把多个操作封装成一个可以被提交或者回滚的原子单元。

默认情况下，每个 SQL 语句都被作为一个自动提交的独立的事务来对待。JDBC 的 Connection 接口中的方法 setAutoCommit() 允许打开或关闭这种行为。因此，如果 conn 是一个打开的连接，则 conn.setAutoCommit(false) 将关闭自动提交。然后事务必须用 conn.commit() 显式提交或用 conn.rollback() 回滚。自动提交可以用 conn.setAutoCommit(true) 来打开。

JDBC 提供处理大对象的接口而不要求在内存中创建整个大对象。为了获取大对象，ResultSet 接口提供方法 getBlob() 和 getClob()，它们与 getString() 方法相似，但是分别返回类型为 Blob 和 Clob 的对象。这些对象并不存储整个大对象，而是存储这些大对象的定位器，即指向数据库中实际大对象的逻辑指针。从这些对象中获取数据与从文件或者输入流中获取数据非常相似，可以采用像 getBytes 和 getSubString 这样的方法来实现。

反向操作时，为了向数据库里存储大对象，可以用 PreparedStatement 类的方法 setBlob（int parameterIndex, InputStream inputStream）把一个类型为**二进制大对象**（blob）的数据库列与一个输入流关联起来。当预备语句被执行时，数据从输入流被读取，然后被写入数据库的二进制大对象中。与此相似，使用方法 setClob 可以设置**字符大对象**（clob）列，该方法的参数包括该列的序号和一个字符串流。

JDBC 还提供了行集（row set）特性，允许把结果集打包起来发送给其他应用程序。行集既可以向后又可以向前扫描，并且可被修改。行集一旦被下载下来就不再是数据库本身的内容了，所以我们在这里并不对其做详细介绍。

5.1.2 ODBC

开放数据库互连（Open DataBase Connectivity，ODBC）标准定义了一个 API，应用程序用它来打开一个数据库连接，发送查询和更新，以及获取返回结果等。应用程序（例如图形界面、统计程序包或者电子表格）可以使用相同的 ODBC API 来访问任何一个支持 ODBC 标准的数据库。 166

每一个支持 ODBC 的数据库系统都提供一个和客户端程序相连接的库，当客户端发出一个 ODBC API 请求，库中的代码就可以和服务器通信来执行被请求的动作并取回结果。

图 5-4 给出了一个使用 ODBC API 的 C 语言代码示例。利用 ODBC 和服务器通信的第一步是，建立

一个和服务器的连接。为了实现这一步，程序先分配一个 SQL 的环境变量，然后是一个数据库连接句柄。ODBC 定义了 HENV、HDBC 和 RETCODE 几种类型。程序随后利用 SQLConnect 打开和数据库的连接，这个调用有几个参数，包括数据库的连接句柄、要连接的服务器、用户的身份和密码等。常数 SQL_NTS 表示前面参数是一个以 null 结尾的字符串。

```
void ODBCexample()
{
    RETCODE error;
    HENV env; /* 环境变量变量 */
    HDBC conn; /* 数据库连接 */

    SQLAllocEnv(&env);
    SQLAllocConnect(env, &conn);
    SQLConnect(conn, "db.yale.edu", SQL_NTS, "avi", SQL_NTS,
               "avipasswd", SQL_NTS);
    {
        char deptname[80];
        float salary;
        int lenOut1, lenOut2;
        HSTMT stmt;

        char * sqlquery = "select dept_name, sum (salary)
            from instructor
            group by dept_name";
        SQLAllocStmt(conn, &stmt);
        error = SQLExecDirect(stmt, sqlquery, SQL_NTS);
        if (error == SQL_SUCCESS) {
            SQLBindCol(stmt, 1, SQL_C_CHAR, deptname, 80, &lenOut1);
            SQLBindCol(stmt, 2, SQL_C_FLOAT, &salary, 0, &lenOut2);
            while (SQLFetch(stmt) == SQL_SUCCESS) {
                printf ("%s %g\n", deptname, salary);
            }
            }
            SQLFreeStmt(stmt, SQL_DROP);
        }
        SQLDisconnect(conn);
        SQLFreeConnect(conn);
        SQLFreeEnv(env);
}
```

图 5-4 ODBC 代码示例

一旦一个连接建立了，C 语言就可以通过 SQLExecDirect 语句把命令发送到数据库。因为 C 语言的变量可以和查询结果的属性绑定，所以当一个元组被 SQLFetch 语句取回的时候，结果中相应的属性的值就可以放到对应的 C 变量里了。SQLBindCol 做这项工作：在 SQLBindCol 函数里面第二个参数代表选择属性中哪一个位置的值，第三个参数代表 SQL 应该把属性转化成什么类型的 C 变量。再下一个参数给出了存放变量的地址。对于诸如字符数组这样的变长类型，最后两个参数还要给出变量的最大长度和一个位置来存放元组取回时的实际长度。如果长度域返回一个负值，那么代表着这个值为空(null)。对于定长类型的变量如整型或浮点型，最大长度的域被忽略。然而当长度域返回一个负值时表示该值为空值。

SQLFetch 在 while 循环中一直执行，直到 SQLFetch 返回一个非 SQL_SUCCESS 的值。在每一次 fetch 过程中，程序把值存放在利用 SQLBindCol 所说明的 C 变量中并把它们打印出来。

在会话结束的时候，程序释放语句的句柄，断开与数据库的连接，同时释放连接和 SQL 环境句柄。好的编程风格要求检查每一个函数的结果，确保它们没有错误。为了简洁，我们在这里忽略了大部分检查。

可以创建带有参数的 SQL 语句，例如，insert into department values(?,?,?)。问号是为将来提供值的占位符。上面的语句可以先被"准备"，也就是在数据库中先编译，然后可以通过为占位符提供具体

值来反复执行——在该例中，为 *department* 关系提供系名、楼宇名和预算数。

ODBC 为各种不同的任务定义了函数，例如查找数据库中所有的关系，以及查找数据库中某个关系的列的名称和类型，或者一个查询结果的列的名称和类型。

在默认情况下，每一个 SQL 语句都被认为是一个自动提交的独立事务。调用 SQLSetConnectOption(conn, SQL_AUTOCOMMIT, 0)可以关闭连接 conn 的自动提交，事务必须通过显式地调用 SQLTransact(conn, SQL_COMMIT)来提交或通过显式地调用 SQLTransact(conn, SQL_ROLLBACK)来回滚。

ODBC 标准定义了符合性级别(*conformance level*)，用于指定标准定义的功能的子集。一个 ODBC 实现可以仅提供核心级特性，也可以提供更多的高级特性(level 1 或 level 2)。level 1 需要支持取得目录的有关信息，例如什么关系存在，它们的属性是什么类型的等。level 2 需要更多的特性，例如发送和提取参数值数组以及检索有关目录的更详细信息的能力。

SQL 标准定义了**调用级接口**(Call Level Interface, CLI)，它与 ODBC 接口类似。

ADO. NET

ADO. NET API 是为 Visual Basic. NET 和 C#语言设计的，它提供了一系列访问数据的函数，与 JDBC 在上层架构没有什么不同，只是在细节上有差别。像 JDBC 和 ODBC 一样，ADO. NET API 可以访问 SQL 查询的结果，以及元数据，但使用起来比 ODBC 简单得多。可以使用 ADO. NET API 来访问支持 ODBC 的数据库，此时，ADO. NET 调用被转换成 ODBC 调用。ADO. NET API 也可以用在某些非关系数据源上，例如微软的 OLE-DB、XML(在第 23 章介绍)，以及微软最近开发的实体框架。有关 ADO. NET 的更多信息请参考文献注解。

5.1.3 嵌入式 SQL

SQL 标准定义了嵌入 SQL 到许多不同的语言中，例如 C，C++，Cobol，Pascal，Java，PL/I 和 Fortran。SQL 查询所嵌入的语言被称为**宿主语言**，宿主语言中使用的 SQL 结构被称为**嵌入式 SQL**。

使用宿主语言写出的程序可以通过嵌入式 SQL 的语法访问和修改数据库中的数据。一个使用嵌入式 SQL 的程序在编译前必须先由一个特殊的预处理器进行处理。嵌入的 SQL 请求被宿主语言的声明以及允许运行时刻执行数据库访问的过程调用所代替。然后，所产生的程序由宿主语言编译器编译。这是嵌入式 SQL 与 JDBC 或 ODBC 的主要区别。

在 JDBC 中，SQL 语句是在运行时被解释的(即使是利用预备语句特性对其进行准备也是如此)。当使用嵌入式 SQL 时，一些 SQL 相关的错误(包括数据类型错误)可以在编译过程中被发现。

为使预处理器识别嵌入式 SQL 请求，我们使用 EXEC SQL 语句，格式如下：

EXEC SQL ＜嵌入式 SQL 语句＞;

嵌入式 SQL 的确切语法依赖于宿主语言。例如，当宿主语言是 Cobol 时，语句中的分号用 END-EXEC 来代替。

我们在应用程序中合适的地方插入 SQL INCLUDE SQLCA 语句，表示预处理器应该在此处插入特殊变量以用于程序和数据库系统间的通信。

在执行任何 SQL 语句之前，程序必须首先连接到数据库。这是用下面语句实现的：

EXEC SQL **connect to** *server* **user** *user-name* **using** *password*;

这里，*server* 标识将要建立连接的服务器。

在嵌入的 SQL 语句中可以使用宿主语言的变量，不过前面要加上冒号(:)以区别于 SQL 变量。如此使用的变量必须声明在一个 DECLARE 区段里，见下面的代码。不过声明变量的语法还是因循宿主语言的惯例。

```
EXEC SQL BEGIN DECLARE SECTION;
int credit_amount;
EXEC SQL END DECLARE SECTION;
```

嵌入式 SQL 语句的格式和本章描述的 SQL 语句类似。但这儿要指出几点重要的不同之处。

167
169

为了表示关系查询，我们使用**声明游标**（declare cursor）语句。然而这时并不计算查询的结果，而程序必须用 **open** 和 **fetch** 语句（本章后面将讨论）得到结果元组。接下来我们将看到，使用游标的方法与 JDBC 中对结果集的迭代处理是很相似的。

考虑我们使用的大学模式。假设我们有一个宿主变量 *credit_amount*，声明方法如前所见，我们想找出学分高于 *credit_amount* 的所有学生的名字。我们可写出查询语句如下：

```
EXEC SQL
    declare c cursor for
    select ID, name
    from student
    where tot_cred > :credit_amount;
```

上述表达式中的变量 *c* 被称为该查询的**游标**（*cursor*）。我们使用这个变量来标识该查询，然后用 **open** 语句来执行查询。

我们的例子中的 **open** 语句如下：

```
EXEC SQL open c;
```

这条语句使得数据库系统执行这条查询并把执行结果存于一个临时关系中。当 **open** 语句被执行的时
170 候，宿主变量（*credit_amount*）的值就会被应用到查询中。

如果 SQL 查询出错，数据库系统将在 SQL 通信区域（SQLCA）的变量中存储一个错误诊断信息。

然后我们利用一系列的 **fetch** 语句把结果元组的值赋给宿主语言的变量。**fetch** 语句要求结果关系的每一个属性有一个宿主变量相对应。在我们的查询例子中，需要一个变量存储 *ID* 的值，另一个变量存储 *name* 的值。假设这两个变量分别是 *si* 和 *sn*，并且都已经在 DECLARE 区段中被声明。那么以下语句：

```
EXEC SQL fetch c into :si, :sn;
```

产生结果关系中的一个元组。接下来应用程序就可以利用宿主语言的特性对 *si* 和 *sn* 进行操作了。

一条单一的 **fetch** 语句只能得到一个元组。如果我们想得到所有的结果元组，程序中必须包含对所有元组执行的一个循环。嵌入式 SQL 为程序员提供了对这种循环进行管理的支持。虽然关系在概念上是一个集合，查询结果中的元组还是有一定的物理顺序的。执行 SQL 的 **open** 语句后，游标指向结果的第一个元组。执行一条 **fetch** 语句后，游标指向结果中的下一个元组。当后面不再有待处理的元组时，SQLCA 中变量 SQLSTATE 被置为"02000"（意指"不再有数据"）；访问该变量的确切的语法依赖于所使用的特定数据库系统。于是，我们可以用一条 **while** 循环（或其他类似循环语句）来处理结果中的每一个元组。

我们必须使用 **close** 语句来告诉数据库系统删除用于保存查询结果的临时关系。对于我们的例子，该语句格式如下：

```
EXEC SQL close c;
```

用于数据库修改（**update**、**insert** 和 **delete**）的嵌入式 SQL 表达式不返回结果。因此，这种语句表达起来在某种程度上相对简单。数据库修改请求格式如下：

```
EXEC SQL < 任何有效的 update, insert 或 delete 语句 >;
```

前面带冒号的宿主语言的变量可以出现在数据库修改语句的表达式中。如果在语句执行过程中出错，SQLCA 中将设置错误诊断信息。

也可以通过游标来更新数据库关系。例如，要为音乐系的每个老师的 *salary* 属性都增加 100，我们
171 可以声明这样一个游标：

```
EXEC SQL
    declare c cursor for
    select *
    from instructor
    where dept_name = 'Music'
    for update;
```

然后我们利用在游标上的 **fetch** 操作对元组进行迭代(就像我们先前看到的例子一样)，每取到一个元组我们都执行以下的代码：

```
EXEC SQL
    update instructor
    set salary = salary + 100
    where current of c;
```

可以用 EXEC SQL COMMIT 语句来提交事务，或者用 EXEC SQL ROLLBACK 进行回滚。

嵌入式 SQL 的查询一般是在编写程序时被定义的。但是在某些比较不常见的情况下查询需要在运行时被定义。例如，一个应用程序接口可能会让用户来指定某个关系的一个或多个属性上的选择条件，然后在运行时只利用用户做了选择的属性的条件来构造 SQL 查询的 **where** 子句。此种情况下，可以使用"EXEC SQL PREPARE < query-name > FROM : < variable >"这样格式的语句，在运行时构造和准备查询字符串；并且可以在查询名字上打开一个游标。

SQLJ

与其他的嵌入式 SQL 的实现方法相比，SQLJ(SQL 嵌入 Java 中)提供了相同的特性，但是它使用了一种不同的语法，更接近 Java 的固有特性，比如迭代器。例如，SQLJ 使用句法#sql 代替 EXEC SQL，并且不使用游标，而是用 Java 迭代器接口来获取查询结果。因此执行查询的结果被储存在 Java 迭代器里，然后利用 Java 迭代器接口中的 next( ) 方法来逐步遍历结果元组，就像前面例子中对游标使用 **fetch** 一样。必须对迭代器的属性进行声明，其类型应该与 SQL 查询结果中的各属性的类型保持一致。下面的代码段演示了迭代器的使用方法。

```
#sql iterator deptInfoIter ( String dept_name, int avgSal);
deptInfoIter iter = null;
#sql iter = { select dept_name, avg( salary)
              from instructor
              group by dept_name };
while ( iter. next( ))
    String deptName = iter. dept_name( );
    int avgSal = iter. avgSal( );
    System. out. println( deptName + " " + avgSal);
}
iter. close( );
```

IBM 的 DB2 和 Oracle 都支持 SQLJ，并且提供从 SQLJ 代码到 JDBC 代码的转换器。该转换器可以在编译时连接数据库来检查查询的语法是否正确，并用来确保查询结果的 SQL 类型与所赋值的 Java 变量类型相一致。从 2009 年年初开始，其他的数据库系统就不支持 SQLJ 了。

我们在这里不详细介绍 SQLJ，更多信息请参看参考文献。

5.2 函数和过程

我们已经介绍了 SQL 语言的几个内建函数。在本节中，我们将演示开发者如何来编写他们自己的函数和过程，把它们存储在数据库里并在 SQL 语句中调用。函数对于特定的数据类型比如图像和几何对象来说特别有用。例如，用在地图数据库中的一个线段数据类型可能有一个相关函数用于判断两个线段是否交叠。一个图像数据类型可能有一个相关函数用于比较两幅图的相似性。

函数和过程允许"业务逻辑"作为存储过程记录在数据库中，并在数据库内执行。例如，大学里通常有许多规章制度，规定在一个学期里每个学生能选多少课，在一年里一个个职的教师至少要上多少节课，一个学生最多可以在多少个专业中注册，等等。尽管这样的业务逻辑能够被写成程序设计语言过程并完全存储在数据库以外，但把它们定义成数据库中的存储过程有几个优点。例如，它允许多个应用访问这些过程，允许当业务规则发生变化时进行单个点的改变，而不必改变应用系统的其他部分。应用代码可以调用存储过程，而不是直接更新数据库关系。

SQL 允许定义函数、过程和方法。定义可以通过 SQL 的有关过程的组件，也可以通过外部的程序设计语言，例如 Java、C 或 C++。我们首先查看 SQL 中的定义，然后在 5.2.3 节了解如何使用外部语言中的定义。

我们在这里介绍的是 SQL 标准所定义的语法，然而大多数数据库都实现了它们自己的非标准版本的语法。例如 Oracle（PL/SQL）、Microsoft SQL Server（Transact-SQL）和 PostgreSQL（PL/pgSQL）所支持的过程语言都与我们在这里描述的标准语法有所差别。我们将在后面用 Oracle 来举例说明某些不同之处。更进一步的详细信息可参见各自的系统手册。尽管我们介绍的部分语法在这些系统上并不支持，但是所阐述的概念在不同的实现上都是适用的，只是语法上有所区别。

172 ~ 173

5.2.1 声明和调用 SQL 函数和过程

假定我们想要这样一个函数：给定一个系的名字，返回该系的教师数目。我们可以如图 5-5 所示定义函数。[⊖]这个函数可以用在返回教师数大于 12 的所有系的名称和预算的查询中：

```
select dept_name, budget
from department
where dept_count(dept_name) > 12;
```

```
create function dept_count(dept_name varchar(20))
    returns integer
    begin
    declare d_count integer;
        select count(*) into d_count
        from instructor
        where instructor.dept_name= dept_name
    return d_count;
    end
```

图 5-5 SQL 中定义的函数

SQL 标准支持返回关系作为结果的函数；这种函数称为**表函数**（table functions）。[⊜]考虑图 5-6 中定义的函数。该函数返回一个包含某特定系的所有教师的表。注意，使用函数的参数时需要加上函数名作为前缀（*instructor_of. dept_name*）。

这种函数可以如下在一个查询中使用：

```
select *
from table (instructor_of ('Finance'));
```

这个查询返回"金融"系的所有教师。在上面的简单情况下直接写这个查询而不用以表为值的函数也是很直观的。但通常以表为值的函数可以被看作**带参数的视图**（parameterized view），它通过允许参数把视图的概念更加一般化。

```
create function instructor_of (dept_name varchar(20))
    returns table (
        ID varchar (5),
        name varchar (20),
        dept_name varchar (20),
        salary numeric (8,2))
return table
    (select ID, name, dept_name, salary
    from instructor
    where instructor.dept_name = instructor_of.dept_name);
```

图 5-6 SQL 中定义的表函数

SQL 也支持过程。*dept_count* 函数也可以写成一个过程：

```
create procedure dept_count_proc (in dept_name varchar(20), out d_count integer)
    begin
        select count(*) into d_count
        from instructor
        where instructor.dept_name = dept_count_proc.dept_name
    end
```

关键字 **in** 和 **out** 分别表示待赋值的参数和为返回结果而在过程中设置值的参数。

⊖ 如果要输入自己的函数和过程，应该写"**create or replace**"而不是 **create**，这样便于在测试时编译代码（对旧的函数进行替换）。

⊜ 这个特性最早出现在 SQL：2003 中。

可以从一个 SQL 过程中或者从嵌入式 SQL 中使用 **call** 语句调用过程：

```
declare d_count integer;
call dept_count_proc('Physics', d_count);
```

过程和函数可以通过动态 SQL 触发，如 5.1.1.4 节中 JDBC 语法所示。

SQL 允许多个过程同名，只要同名过程的参数个数不同。名称和参数个数用于标识一个过程。SQL 也允许多个函数同名，只要这些同名的不同函数的参数个数不同，或者对于那些有相同参数个数的函数，至少有一个参数的类型不同。

174
|
175

5.2.2 支持过程和函数的语言构造

SQL 所支持的构造赋予了它与通用程序设计语言相当的几乎所有的功能。SQL 标准中处理这些构造的部分称为**持久存储模块**(Persistent Storage Module，PSM)。

变量通过 **declare** 语句进行声明，可以是任意的合法 SQL 类型。使用 **set** 语句进行赋值。

一个复合语句有 **begin**…**end** 的形式，在 **begin** 和 **end** 之间会包含复杂的 SQL 语句。如我们在 5.2.1 节中曾看到的那样，可以在复合语句中声明局部变量。一个形如 **begin atomic**…**end** 的复合语句可以确保其中包含的所有语句作为单一的事务来执行。

SQL:1999 支持 **while** 语句和 **repeat** 语句，语法如下：

```
while 布尔表达式 do
        语句序列;
end while

repeat
        语句序列;
until 布尔表达式
end repeat
```

还有 **for** 循环，它允许对查询的所有结果重复执行：

```
declare n integer default 0;
for r as
        select budget from department
        where dept_name = 'Music'
do
    set n = n + r.budget
end for
```

程序每次获取查询结果的一行，并存入 **for** 循环变量(在上面例子中为 *r*)中。语句 **leave** 可用来退出循环，而 **iterate** 表示跳过剩余语句从循环的开始进入下一个元组。

SQL 支持的条件语句包括 if-then-else 语句，语法如下：

```
if 布尔表达式
    then 语句或复合语句
elseif 布尔表达式
    then 语句或复合语句
else 语句或复合语句
end if
```

SQL 也支持 case 语句，类似于 C/C++ 语言中的 case 语句(加上我们在第 3 章看到的 case 表达式)。

图 5-7 提供了一个有关 SQL 的过程化结构的更大型一点的例子。图中定义的函数 *registerStudent* 首先确认选课的学生数没有超过该课所在教室的容量，然后完成学生对该课的注册。函数返回一个错误代码，这个值大于等于 0 表示成功，返回负值表示出错，同时以 **out** 参数的形式返回消息来说明失败的原因。

SQL 程序语言还支持发信号通知**异常条件**(exception condition)，以及声明句柄(handler)来处理异常，代码如下：

```
declare out_of_classroom_seats condition
declare exit handler for out_of_classroom_seats
begin
sequence of statements
end
```

```
——在确保教室能容纳下的前提下注册一个学生
——如果成功注册，返回 0，如果超过教室容量则返回 -1
create function registerStudent(
        in s_id varchar(5),
        in s_courseid varchar (8),
        in s_secid varchar (8),
        in s_semester varchar (6),
        in s_year numeric (4, 0),
        out errorMsg varchar(100))
returns integer
begin
    declare currEnrol int;
    select count(*) into currEnrol
    from takes
    where course_id = s_courseid and sec_id = s_secid
          and semester = s_semester and year = s_year;
    declare limit int;
    select capacity into limit
    from classroom natural join section
    where course_id = s_courseid and sec_id = s_secid
          and semester = s_semester and year = s_year;
    if (currEnrol < limit)
      begin
        insert into takes values
            (s_id, s_courseid, s_secid, s_semester, s_year, null);
        return(0);
      end
    ——否则，已经达到课程容量上限
    set errorMsg = 'Enrollment limit reached for course ' || s_courseid
        || ' section ' || s_secid;
    return(-1);
end;
```

图 5.7 学生注册课程的过程

在 **begin** 和 **end** 之间的语句可以执行 **signal** *out_of_classroom_seats* 来引发一个异常。这个句柄说明，如果条件发生，将会采取动作经由 **begin end** 中的语句。另一个可选的动作将是 **continue**，它继续从引发异常的语句的下一条语句开始执行。除了用确定义的条件，还有一些预定义的条件，比如 **sqlexception**、**sqlwarning** 和 **not found**。

过程和函数的非标准语法

尽管 SQL 标准为过程和函数定义了语法，但是很多数据库并不严格遵照标准，在语法支持方面存在很多变化。这种情况的原因之一是这些数据库通常在语法标准制定之前就已经引入了对过程和函数的支持机制，然后一直沿用最初的语法。在这里把每个数据库所支持的语法罗列出来并不现实，不过我们可以介绍一下 Oracle 的 PL/SQL 与标准语法不同的一些方面，下面是图 5-5 中的函数在 PL/SQL 里定义的一个版本。

```
create or replace function dept_count(dept_name in instructor.dept_name%type)
return integer
as
    d_count integer;
begin
    select count(*) into d_count
    from instructor
    where instructor.dept_name = dept_name;
    return d_count;
end;
```

这两个版本在概念上相似，但还是存在很多次要的语法上的区别，可以利用其中一些不同来作为区分两个函数版本的手段。尽管没有在这里展示，但是PL/SQL中关于控制流的语法与在这里所列出的也有所不同。

可以看到，通过加%type前缀，PL/SQL允许把某个类型设置为关系的一个属性的类型。另一方面，PL/SQL并不直接支持返回一个表的功能，只能通过创建表类型这样的间接方法来实现。其他数据库所支持的过程语言同样含有很多语法和语义上的差别。更多信息请查看相关语言的参考资料。

5.2.3 外部语言过程

尽管对SQL的过程化扩展非常有用，然而可惜的是这些并不被跨数据库的标准的方法所支持。即使是最基本的特性在不同数据库产品中都可能有不同的语法和语义。所以，程序员必须针对每个数据库产品学习一门新语言。还有另一种方案可以解决语言支持的问题，即在一种命令式程序设计语言中定义过程，然后从SQL查询和触发器的定义中来调用它。

SQL允许我们用一种程序设计语言定义函数，比如Java、C#、C或C++。这种方式定义的函数会比SQL中定义的函数效率更高，无法在SQL中执行的计算可以由这些函数执行。

外部过程和函数可以这样指定(注意，确切的语法决定于所使用的特定数据库系统)：

```
create procedure dept_count_proc( in dept_name varchar(20),
                                  out count integer)
language C
external name '/usr/avi/bin/dept_count_proc'

create function dept_count(dept_name varchar(20))
returns integer
language C
external name '/usr/avi/bin/dept_count'
```

通常来说，外部语言过程需要处理参数(包含**in**和**out**参数)和返回值中的空值，还需要传递操作失败/成功的状态，以方便对异常进行处理。这些信息可以通过几个额外的参数来表示：一个指明失败/成功状态的**sqlstate**值、一个存储函数返回值的参数，以及一些指明每个参数/函数结果的值是否为空的指示器变量。还可以通过其他机制来解决空值的问题，例如，可以传递指针而不是值。具体采用哪种方法取决于数据库。不过，如果一个函数不关注这些情况，可以在声明语句的上方添加额外的一行**parameter style general**指明外部过程/函数只使用说明的变量并且不处理空值和异常。

用程序设计语言定义并在数据库系统之外编译的函数可以由数据库系统代码来加载和执行。不过这么做存在危险，那就是程序中的错误可能破坏数据库内部的结构，并且绕过数据库系统的访问-控制功能。如果数据库系统关心执行的效率胜过安全性则可以采用这种方式执行过程。关心安全性的数据库系统一般会将这些代码作为一个单独进程的一部分来执行，通过进程间通信，传入参数的值，取回结果。然而，进程间通信的时间代价相当高；在典型的CPU体系结构中，一个进程通信所需的时间可以执行数万到数十万条指令。

如果代码用Java或C#这种"安全"的语言书写，则会有第三种可能：在数据库进程本身的沙盒(sandbox)内执行代码。沙盒允许Java或C#代码访问它的内存区域，但阻止代码直接在查询执行过程的内存中做任何读操作或者更新操作，或者访问文件系统中的文件。(在如C这种语言中创建一个沙盒是不可能的，因为C语言允许通过指针不加限制地访问内存。)避免进程间通信能够大大降低函数调用的时间代价。

当今的一些数据库系统支持外部语言例程在查询执行过程中的沙盒里运行。例如，Oracle和IBM DB2允许Java函数作为数据库过程中的一部分运行。Microsoft SQL Server允许过程编译成通用语言运行程序(CLR)来在数据库过程中执行；这样的过程可以用C#或Visual Basic编写。PostgreSQL允许在Perl、Python和Tcl等多种语言中定义函数。

176
179

5.3 触发器

触发器(trigger)是一条语句，当对数据库作修改时，它自动被系统执行。要设置触发器机制，必须满足两个要求：

- 指明什么条件下执行触发器。它被分解为一个引起触发器被检测的事件和一个触发器执行必须满足的条件。
- 指明触发器执行时的动作。

一旦我们把一个触发器输入数据库，只要指定的事件发生，相应的条件满足，数据库系统就有责任去执行它。

5.3.1 对触发器的需求

[180] 触发器可以用来实现未被SQL约束机制指定的某些完整性约束。它还是一种非常有用的机制，用来当满足特定条件时对用户发警报或自动开始执行某项任务。例如，我们可以设计一个触发器，只要有元组被插入 *takes* 关系中，就更新 *student* 关系中选课的学生所对应的元组，把该课的学分加入这个学生的总学分中。作为另一个应用触发器的例子，假设一个仓库希望每种物品的库存保持一个最小量：当某种物品的库存少于最小值的时候，自动发出一个订货单。在更新某种物品的库存的时候，触发器会比较这种物品的当前库存和它的最小库存，如果库存数量等于或小于最小值，就会生成一个新的订单。

注意，触发器系统通常不能执行数据库以外的更新。因此，在上面的库存补充的例子中，我们不能用一个触发器去直接在外部世界下订单，而是在存放订单的关系中添加一个关系记录。我们必须另外创建一个持久运行的系统进程来周期性扫描该关系并订购产品。某些数据库系统提供了内置的支持，可以使用上述方法从SQL查询和触发器中发送电子邮件。

5.3.2 SQL中的触发器

现在我们来看如何在SQL中实现触发器。我们在这里介绍的是SQL标准定义的语法，但是大部分数据库实现的是非标准版本的语法。尽管这里所述的语法可能不被这些系统支持，但是我们阐述的概念对于不同的实现方法都是适用的。我们将在本章末尾讨论非标准的触发器实现。

图5-8展示了如何使用触发器来确保关系 *section* 中属性 *time_slot_id* 的参照完整性。图中第一个触发器的定义指明该触发器在任何一次对关系 *section* 的插入操作执行之后被启动，以确保所插入元组的 *time_slot_id* 字段是合法的。一个SQL插入语句可以向关系中插入多个元组，在触发器代码中的 **for each row** 语句可以显式地在每一个被插入的行上进行迭代。**referencing new row as** 语句建立了一个变量 *nrow*(称为**过渡变量**(transition variable))，用来在插入完成后存储所插入行的值。

```
create trigger timeslot_check1 after insert on section
    referencing new row as nrow
    for each row
    when (nrow.time_slot_id not in (
            select time_slot_id
            from time_slot)) /* time_slot 中不存在该 time_slot_id */
    begin
      rollback
    end;
    create trigger timeslot_check2 after delete on time_slot
    referencing old row as orow
    for each row
    when (orow.time_slot_id not in (
            select time_slot_id
            from time_slot) /* 在 time_slot 中该槽被删除的 time_slot_id */
    and orow.time_slot_id in (
            select time_slot_id
            from section)) /* 在 section 中仍有对该 time_slot_id 的引用 */
    begin
      rollback
    end;
```

图5-8 使用触发器来维护参照完整性

when 语句指定一个条件。仅对于满足条件的元组系统才会执行触发器中的其余部分。**begin atomic** …**end** 语句用来将多行 SQL 语句集成为一个复合语句。不过在我们的例子中只有一条语句，它对引起触发器被执行的事务进行回滚。这样，所有违背参照完整性约束的事务都将被回滚。从而确保数据库中的数据满足约束条件。

只检查插入时的参照完整性还不够，我们还需要考虑对关系 *section* 的更新，以及对被引用的表 *time_slot* 的删除和更新操作。图 5-8 定义的第二个触发器关注的是 *time_slot* 表的删除。该触发器检查被删除元组的 *time_slot_id* 要么还在 *time_slot* 中，要么 *section* 里不存在包含这个 *time_slot_id* 值的元组；否则将违背参照完整性。

为了保证参照完整性，我们也必须为处理 *section* 和 *time_slot* 的更新来创建触发器；我们接下来将介绍如何在更新时执行触发器，不过，该如何定义这些触发器就留给读者作为练习。

对于更新来说，触发器可以指定哪个属性的更新使其执行；而其他属性的更新不会让它产生动作。例如，为了指定当更新关系 *takes* 的属性 *grade* 时执行触发器，我们可以这样写：

after update of *takes* **on** *grade*

referencing old row as 子句可以建立一个变量用来存储已经更新或删除行的旧值。**referencing new row as** 子句除了插入还可以用于更新操作。

如图 5-9 所示，当关系 *takes* 中元组的属性 *grade* 被更新时，需要用触发器来维护 *student* 里元组的 *tot_cred* 属性，使其保持实时更新。只有当属性 *grade* 从空值或者 'F' 被更新为代表课程已经完成的具体分数时，触发器才被激发。除了 *nrow* 的使用，**update** 语句都属于标准的 SQL 语法。

181
~
182

```
create trigger credits_earned after update of takes on (grade)
    referencing new row as nrow
    referencing old row as orow
    for each row
    when nrow.grade <> 'F' and nrow.grade is not null
        and (orow.grade = 'F' or orow.grade is null)
    begin atomic
        update student
        set tot_cred = tot_cred +
            (select credits
             from course
             where course.course_id = nrow.course_id)
        where student.id = nrow.id;
    end;
```

图 5-9 使用触发器来维护 *credits_earned* 值

本例中触发器的更实际的实现还可以解决分数更正的问题，包括把成功结课的分数改成不及格分数，以及向关系 *takes* 中插入含有表示成功结课的 *grade* 值的元组。读者可以把这些实现作为练习。

另举一个使用触发器的例子，当删除一个 *student* 元组的事件发生时，需要检查关系 *takes* 中是否存在与该学生相关的项，如果有则删除它。

许多数据库系统支持各种别的触发器事件，比如当一个用户（应用程序）登录到数据库（即打开一个连接）的时候，或者当系统停止的时候，或者当系统设置改变的时候。

触发器可以在事件（insert、delete 或 update）之前激发，而不仅是事件之后。在事件之前被执行的触发器可以作为避免非法更新、插入或删除的额外约束。为了避免执行非法动作而产生错误，触发器可以采取措施来纠正问题，使更新、插入或删除操作合法化。例如，假设我们想把一个教师插入某个系，而该系的名称并不在关系 *department* 中，那么触发器就可以提前向关系 *department* 中加入该系名称，以避免在插入时产生外码冲突。另一个例子是，假设所插入分数的值为空白则表明该分数发生缺失。我们可以定义一个触发器，将这个值用 **null** 值代替。**set** 语句可以用来执行这样的修改。这种触发器的例子如图 5-10 所示。

我们可以对引起插入、删除或更新的SQL语句执行单一动作，而不是对每个被影响的行执行一个动作。要做到这一点，我们用 **for each statement** 子句来替代 **for each row** 子句。可以用子句 **referencing old table as** 或 **referencing new table as** 来指向包含所有的被影响的行的临时表(称为过渡表(transition table))。过渡表不能用于 **before** 触发器，但是可以用于 **after** 触发器，无论它们是语句触发器还是行触发器。这样，在过渡表的基础上，一个单独的SQL语句就可以用来执行多个动作。

```
create trigger avgnull before update of sales
    referencing new row as nrow
    for each row
    when (nrow.grade = ' ')
    begin atomic
        set nrow.grade = null;
    end;
```

图5-10 使用set来更改插入值的例子

非标准的触发器语法

尽管我们在这里介绍的触发器语法是SQL标准的一部分，并被IBM DB2所支持，但是大多数其他的数据库系统用非标准的语法来说明触发器，不一定实现了SQL标准的所有特性。我们将在下面概述其中的一些不同；更进一步的细节请参考相关的系统手册。

例如，与SQL标准语法不同的是，Oracle的语法中，在 **referencing** 语句中并没有关键词 **row**，而且在 **begin** 之后没有关键词 **atomic**。在 **update** 语句中嵌入的 **select** 语句对 *nrow* 的引用必须以冒号(:)为前缀，用以向系统说明变量 *nrow* 是在SQL语句之外所定义的。更不一样的是，在 **when** 和 **if** 子句中不允许包含子查询。可以用下面的方法绕过这个限制：把复杂的谓词从 **when** 子句移到单独的查询中，用本地变量保存查询结果，然后在一个 **if** 子句里引用该变量，并把触发器的内容移动到相关的 **then** 子句里。更进一步，Oracle中的触发器不允许直接执行事务回滚；但是，可以使用函数 **raise_application_error** 来代替它，它在回滚事务的同时返回错误信息给执行更新的用户/应用程序。

另一个例子是，在Microsoft SQL Server中用关键字 **on** 来替代 **after**，不支持 **referencing** 子句，而是用 **deleted** 和 **inserted** 保留元组变量来表示被影响的行的旧值和新值。另外，语法中略去了 **for each row** 子句，并用 **if** 来替代 **when**。不支持 **before** 语句样式，但是支持 **instead of** 语法。

在PostgreSQL中，触发器的定义不包含执行内容，而是对每一行调用一个过程，使用 **old** 和 **new** 来访问包含该行的旧值和新值的变量。触发器不进行回滚，而是发出一个异常，该异常中包含了相关的错误信息。

183
|
184

触发器可以设为有效或者无效：默认情况下它们在创建时是有效的，但是可以通过使用 **alter trigger** *trigger_name* **disable**(某些数据库使用另一种语法，比如 **disable trigger** *trigger_name*)将其设为无效。设为无效的触发器可以重新设为有效。通过使用命令 **drop trigger** *trigger_name*，触发器也可以被丢弃，即将其永久移除。

回到仓库库存的例子，假设有如下的关系：

- *inventory*(*item*, *level*)，表示物品在仓库中的当前库存量。
- *minlevel*(*item*, *level*)，表示物品应该保持的最小库存量。
- *reorder*(*item*, *amount*)，表示当物品小于最小库存量的时候要订购的数量。
- *orders*(*item*, *amount*)，表示物品被订购的数量。

注意我们已经很小心地在仅当物品库存量从大于最小值降到最小值以下的时候才下达一个订单。如果只检查更新后的新数值是否小于最小值，就可能在物品已经被重订购的时候错误地下达一个订单。我们可以用图5-11中的触发器来重新订购物品。

尽管触发器在SQL:1999之前并不是SQL标准的一部分，但是它仍被广泛地应用在基于SQL的数据库系统中。遗憾的是，每个数据库系统都实现了自己的触发器语法，导致彼此不能兼容。我们在这里用的是SQL:1999的触发器语法，它与IBM的DB2以及Oracle数据库系统的语法比较相似，但不完全一致。

```
create trigger reorder after update of level on inventory
    referencing old row as orow, new row as nrow
    for each row
    when nrow.level <= (select level
                        from minlevel
                        where minlevel.item = orow.item)
    and orow.level > (select level
                      from minlevel
                      where minlevel.item = orow.item)
    begin atomic
        insert into orders
            (select item, amount
             from reorder
             where reorder.item = orow.item);
    end;
```

图5-11 重新订购物品的触发器例子

5.3.3 何时不用触发器

触发器有很多合适的用途，例如我们刚刚在5.3.2节中看到的那些，然而有一些场合最好用别的技术来处理。比如，我们可以用触发器替代级联特性来实现外码约束的**on delete cascade**特性。然而这样不仅需要完成更大的工作量，而且使数据库中实现的约束集合对于数据库用户来说更加难以理解。

举另外一个例子，可以用触发器来维护物化视图。例如，如果我们希望能够快速得到每节课所注册的学生总数，我们可以创建一个关系来实现这个功能：

section_registration(*course_id*, *sec_id*, *semester*, *year*, *total_students*)

它由以下查询所定义：

select *course_id*, *sec_id*, *semester*, *year*, **count**(*ID*) **as** *total_students*
from *takes*
group by *course_id*, *sec_id*, *semester*, *year*;

在对关系*takes*进行插入、删除或更新时，每门课的*total_students*的值必须由触发器来维护到最新状态。维护时可能要对*section_registration*做插入、更新或删除，这些都相应地写在触发器里。

然而，许多数据库现在支持物化视图，由数据库系统自动维护（见4.2.3节）。因此没必要编写代码让触发器来维护这样的物化视图。

触发器也被用来复制或者备份数据库：在每一个关系的插入、删除或更新的操作上创建触发器，将改变记录在称为**change**或**delta**的关系上。一个单独的进程将这些改变复制到数据库的副本。然而，现代的数据库系统提供内置的数据库复制工具，使得复制在大多数情况下不必使用触发器。本书将在第19章详细讨论复制数据库。

触发器的另一个问题是当数据从一个备份的拷贝[6]中加载，或者一个站点上的数据库更新复制到备份站点的时候，触发器动作的非故意执行。在该情况下，触发器动作已经执行了，通常不应该再次执行。在加载数据的时候，触发器应当显式设为无效。对于要接管主系统的备份复制系统，触发器应该一开始就设为无效，而在备份站点接管了主系统的业务后，再设为有效。作为取代的方法，一些数据库系统允许触发器定义为**not for replication**，保证触发器不会在数据库备份的时候在备份站点执行。另一些数据库系统提供了一个系统变量用于指明该数据库是一个副本，数据库动作在其上是重放；触发器会检查这个变量，如果为真则退出执行。这两种解决方案都不需要显式地将触发器设为失效或有效。

写触发器时，应特别小心，这是因为在运行期间一个触发器错误会导致引发该触发器的动作语句

[6] 我们将在第16章详细讨论数据库备份和从故障中恢复的内容。

失败。而且，一个触发器的动作可以引发另一个触发器。在最坏的情况下，这甚至会导致一个无限的触发链。例如，假设在一个关系上的插入触发器里有一个动作引起在同一关系上的另一个（新的）插入，该新插入动作也会引起另一个新插入，如此无穷循环下去。有些数据库系统会限制这种触发器链的长度（例如 16 或 32），把超过此长度的触发器链看作是一个错误。另一些系统把引用特定关系的触发器标记为错误，对该关系的修改导致了位于链首的触发器被执行。

触发器是很有用的工具，但是如果有其他候选方法就最好别用触发器。很多触发器的应用都可以用适当的存储过程来替换，后者我们在 5.2 节已经介绍过了。

5.4 递归查询**

考虑图 5-12 中的例子，关系 *prereq* 的实例包含大学开设的各门课程的信息以及每门课的先修条件。[⊖]

course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

图 5-12 关系 *prereq*

假设我们想知道某个特定课程（例如 CS-347）的直接或者间接的先修课程。也就是说，我们想找到这样的课，要么是选修 CS-347 的直接先决条件，要么是选修 CS-347 的先修课程的先决条件，以此类推。

因此，如果 CS-301 是 CS-347 的先修课程，并且 CS-201 是 CS301 的先修课程，还有 CS-101 是 CS-201 的先修课程，那么 CS-301、CS-201、CS-101 都是 CS-347 的先修课程。

关系 *prereq* 的**传递闭包**（transitive closure）是一个包含所有（*cid*，*pre*）对的关系，*pre* 是 *cid* 的一个直接或间接先修课程。有许多要求计算与此类似的层次（hierarchy）的传递闭包的应用。例如，机构通常由几层组织单元构成。机器由部件构成，而部件又有子部件，如此类推；例如，一辆自行车可能有子部件如车轮和踏板，它们又有子部件如轮胎、轮圈、辐条。可以对这种层次结构使用传递闭包来找出，例如，自行车中的所有部件。

5.4.1 用迭代来计算传递闭包

一个写上述查询的方法是使用迭代：首先找到 CS-347 的那些直接先修课程，然后是第一个集合中的所有课程的先修课程，如此类推。迭代持续进行，直到某次循环中没有新课程加进来才停止。图 5-13 显示了执行这项任务的函数 *findAllPrereqs*（*cid*）；这个函数以课程的 *course_id* 为参数（*cid*），计算该课程所有直接或间接的先修课程并返回它们组成的集合。

过程中用到了三个临时表：

- *c_prereq*：存储要返回的元组集合。
- *new_c_prereq*：存储在前一次迭代中找到的课程。
- *temp*：当对课程集合进行操作时用作临时存储。

注意，SQL 允许使用命令 **create temporary table** 来创建临时表；这些表仅在执行查询的事务内部才可用，并随事务的完成而被删除。而且，如果 *findAllPrereqs* 的两个实例同时运行，每个实例都拥有自己的临时表副本；假设它们共享一份副本，结果就会出错。

该过程在 **repeat** 循环之前把课程 *cid* 的所有直接先修课程插入 *new_c_prereq* 中。**repeat** 循环首先把 *new_c_prereq* 中所有课程加入 *c_prereq* 中。然后，它为 *new_c_prereq* 中的所有课程计算先修课程，从结果中筛选掉此前已经计算出来是 *cid* 的先修课的课程，把剩下的存放在临时表 *temp* 中。最后，它把 *new_c_prereq* 中的内容替换成 *temp* 中的内容。当找不到新的（间接）先修课程时，**repeat** 循环终止。

以 CS-347 为参数调用以上过程，图 5-14 显示了每次迭代中找到的先修课程。

我们注意到在函数中使用 **except** 子句保证即使在先修关系中存在环时（非正常情况），函数也能工作。例如，如果 *a* 为 *b* 的先修课，*b* 为 *c* 的先修课，*c* 为 *a* 的先修课，则存在一个环。

⊖ *prereq* 的实例与我们以前使用的有所不同，其原因当我们用它来解释递归查询时就能看得很清楚了。

```
create function findAllPrereqs(cid varchar(8))
        ——找出 cid 的所有(直接或间接)先修课程
returns table (course_id varchar(8))
    ——关系 prereq(course_id, prereq_id) 指明每一门课程是另一门课与直接先修课
begin
    create temporary table c_prereq (course_id varchar(8));
        —— 表 c_prereq 存储将要返回的课程集合
    create temporary table new_c_prereq (course_id varchar(8));
        ——表 new_c_prereq 包含上一次迭代中发现的课程
    create temporary table temp (course_id varchar(8));
        ——表 temp 用来存储中间结果
    insert into new_c_prereq
        select prereq_id
        from prereq
        where course_id = cid;
    repeat
        insert into c_prereq
            select course_id
            from new_c_prereq;
        insert into temp
            (select prereq.course_id
             from new_c_prereq, prereq
             where new_c_prereq.course_id = prereq.prereq_id
            )
            except (
            select course_id
            from c_prereq
            );
        delete from new_c_prereq;
        insert into new_c_prereq
            select *
            from temp;
        delete from temp;
        until not exists (select * from new_c_prereq)
    end repeat;
    return table c_prereq;
end
```

图 5-13 找到其门课程的所有先修课程

尽管在课程先修关系中不存在环是不现实的，但循环可能在其他应用中存在。例如，假设我们有一个关系 *flights*(*to*, *from*)表示哪个城市可以从其他城市直接飞达。我们可以写类似于 *findAllPrereqs* 函数的代码来找到所有从某个给定城市出发通过一次或一系列的飞行可以到达的城市。我们所要做的只是用 *flights* 代替 *prereq* 并且替换相应的属性名称。在这个情况下可能存在可达关系的环，但函数仍然会正确工作，因为它会将所有已见过的城市去掉。

Iteration Number	Tuples in c1
0	
1	(CS-301)
2	(CS-301), (CS-201)
3	(CS-301), (CS-201)
4	(CS-301), (CS-201), (CS-101)
5	(CS-301), (CS-201), (CS-101)

图 5-14 函数 *findAllPrereqs* 迭代过程中产生的 CS-347 的先修课程

5.4.2 SQL 中的递归

用迭代表示传递闭包是很不方便的。还有另一种方法，即使用递归视图定义，这种方法更加容易使用。

我们可以用递归为某个指定课程如 CS-347 定义先修课程集合，方法如下。CS-347 的(直接或间接的)先修课程是：

- CS-347 的先修课程。
- 作为 CS-347 的(直接或间接的)先修课程的先修课程的课程。

注意，第二条是递归，因为它用 CS-347 的先修课程来定义 CS-347 的先修课程。传递闭包的其他例子

如找出一个给定部件的所有直接或间接子部件同样可以类似地递归定义。

从 SQL:1999 开始，SQL 标准中用 **with recursive** 子句来支持有限形式的递归，在递归中一个视图（或临时视图）用自身来表达自身。例如，递归查询可以用于精确地表达传递闭包。回忆 **with** 子句用于定义一个临时视图，该视图的定义只对定义它的查询可用。附加的关键字 **recursive** 表示该视图是递归的。

例如，用图 5-15 中显示的递归 SQL 视图，我们可以找到每一对（*cid*, *pre*），其中 *pre* 是 *cid* 的直接或间接的先修课程。

任何递归视图都必须被定义为两个子查询的并：一个非递归的**基查询**（base query）和一个使用递归视图的**递归查询**（recursive query）。在图 5-15 所示的例子中，基查询是关系 *prereq* 上的选择操作，而递归查询则计算 *prereq* 和 *rec_prereq* 的连接。

185
190

对递归视图的含义最好的理解如下。首先计算基查询并把所有结果元组添加到视图关系 *rec_prereq*（初始时为空）中。然后用当前视图关系的内容计算递归查询，并把所有结果元组加回到视图关系中。持续重复上述步骤直至没有新的元组添加到视图关系中为止。得到的视图关系实例就称为递归视图定义的一个**不动点**（fixed point）。（术语"不动"是指不会再有进一步的改变。）这样视图关系被定义为正好包含处于不动点实例中的元组。

```
with recursive rec_prereq(course_id, prereq_id) as (
        select course_id, prereq_id
        from prereq
    union
        select rec_prereq.course_id, prereq.prereq_id
        from prereq, rec_prereq
        where prereq.course_id = rec_prereq.prereq_id
    )
select *
from rec_prereq;
```

图 5-15 SQL 中的递归查询

把上述逻辑应用到我们的例子中，我们将首先通过执行基查询找到每门课的直接先修课程。递归查询会在每次迭代过程中增加一层课程，直到达到课程 – 先修课程关系的最大层次。在这时将不会再有新的元组添加到视图中，同时达到了一个不动点。

为了找到指定课程的先修课程，以 CS-347 为例，我们可以加入 where 子句"**where** *rec_prereq.course_id* = 'CS-347'"来修改外层查询。对如此条件的查询进行求值，方法之一是：先使用迭代技术计算 *rec_prereq* 的所有内容，然后从结果中选择 *course_id* 为 CS-347 的那些元组。但是，这样需要为所有课程计算（课程，先修课程）对，其中，除了涉及 CS-347 的之外，其他的都不相关。事实上，数据库系统没有必要使用上述迭代技术计算递归查询的完整结果然后进行筛选。可以使用其他效率更高的技术来得到相同的结果，比如 *findAllPrereqs* 函数中用到的方法看起来就更简单些。更多信息请参阅参考文献。

递归视图中的递归查询是有一些限制的：具体地说，该查询必须是**单调的**（monotonic），也就是说，如果视图关系实例 V_1 是实例 V_2 的超集的话，那么它在 V_1 上的结果必须是它在 V_2 上的结果的超集。直观上，如果更多的元组被添加到视图关系，则递归查询至少应该返回与以前相同的元组集，并
191 且还可能返回另外一些元组。

特别指出，递归查询不能用于任何下列构造，因为它们会导致查询非单调：

- 递归视图上的聚集。
- 在使用递归视图的子查询上的 **not exists** 谓词。
- 右端使用递归视图的集合差（**except**）运算。

例如，如果递归查询是 $r - v$ 的形式，其中 v 是递归视图，如果我们在 v 中增加一个元组，查询结果可能会变小；可见该查询不是单调的。

只要递归查询是单调的，递归视图的含义就可以用迭代过程来定义；如果递归查询是非单调的，则视图的含义很难确定。因此 SQL 要求查询必须是单调的。递归查询将在 B.3.6 节 Datalog 查询语言的环境中更加详细地讨论。

SQL 还允许使用 **create recursive view** 代替 **with recursive** 来创建递归定义的永久视图。一些系统实现支持使用不同评估的递归查询；请参考各自的系统手册以获得更多细节。

5.5　高级聚集特性**

此前我们已经看到，SQL 对聚集的支持是十分强大的，可以很便捷地完成一般性的任务。然而，有些任务很难用基本的聚集特性来高效实现。在本节中，我们将研究为完成这些任务而向 SQL 中引入的一些特性。

5.5.1　排名

从一个大的集合中找出某值的位置是一个常见的操作。例如，我们可能希望基于学生的平均绩点（GPA）赋予他们在班级中的名次：GPA 最高的学生排名第 1，次高分学生排名第 2，等等。另一种相关的查询类型是找某个值在一个（允许重复值的）集合中所处的百分点，比如排在后 1/3、中间 1/3 或是前 1/3。虽然这样的查询可用构造 SQL 语句来完成，但表达困难且计算效率低。编程人员通常借助于将部分代码写在 SQL 中，另一部分代码写在程序设计语言中的方式去实现它。我们在这里讨论 SQL 中如何对这类查询进行直接表达。

在我们的大学例子中，关系 *takes* 存储了每个学生在所选的每一门课程上所获得的成绩。为了演示排名，我们假设有一个视图 *student_grades*(*ID*, *GPA*)，它给出了每个学生的平均绩点。[⊖]

排名是用 **order by** 说明来实现的。下面的查询给出了每个学生的名次：

192

```
select ID, rank() over (order by (GPA) desc) as s_rank
from student_grades;
```

注意这里没有定义输出中的元组顺序，所以元组可能不按名次排序。需要使用一个附加的 **order by** 子句得到排序的元组，如下所示：

```
select ID, rank() over (order by (GPA) desc) as s_rank
from student_grades
order by s_rank;
```

有关排名的一个基本问题是如何处理多个元组在排序属性上具有相同值的情况。在我们的例子中，这意味着如果有两个 GPA 相同的学生应如何处理的问题。**rank** 函数对所有在 **order by** 属性上相等的元组赋予相同的名次。例如，如果两个学生具有相同的最高 GPA，则两人都得到第 1 名。下一个给出的名次将是 3 而不是 2，所以如果三个学生得到了次高的 GPA，他们都将得到第 3 名，接下来的一个或者多个学生将得到第 6 名，等等。另有一个 **dense_rank** 函数，它不在等级排序中产生隔断。在上面的例子中，具有次高成绩的元组都得到第 2 名，具有第三高的值的元组得到第 3 名，等等。

可以使用基本的 SQL 聚集函数来表达上述查询，查询语句如下：

```
select ID, (1 + (select count(*)
                 from student_grades B
                 where B.GPA > A.GPA)) as s_rank
from student_grades A
order by s_rank;
```

显然，正如上述语句所描述的那样，一个学生的排名就是那些 GPA 比他高的学生的数目再加 1。然而，计算每个学生的排名所耗时间与关系的大小呈线性增长，导致整体耗时与关系大小呈平方量级增长。对于大型关系，上述查询可能要花很长时间来执行。相比之下，**rank** 子句的系统实现能够对关系进行排序并在相当短的时间内计算排名。

排名可在数据的不同分区里进行。例如，我们可能会希望按照系而不是在整个学校范围内对学生排名。假设有一个视图，它像 *student_grades* 那样定义，但是包含系名：*dept_grades*(*ID*, *dept_name*, *GPA*)。那么下面的查询就给出了学生们在每个分区里的排名：

193

⊖　用 SQL 语句来创建视图 *student_grades* 比较难，因为我们必须把关系 *takes* 中的字母评分转换成数字，并且要根据课程与学分数来考虑该课成绩所占的权重。该视图的定义是在习题 4.5 中要做的。

```
select ID, dept_name,
       rank () over (partition by dept_name order by GPA desc) as dept_rank
from dept_grades
order by dept_name, dept_rank;
```

外层的 **order by** 子句将结果元组按系名排序，在各系内按照名次排序。

一个单独的 **select** 语句中可以使用多个 **rank** 表达式，因此，通过在同一个 **select** 语句中使用两个 **rank** 表达式的方式，我们可以得到总名次以及系内的名次。当排名（可能带有分区）与 **group by** 子句同时出现的时候，**group by** 子句首先执行，分区和排名在 **group by** 的结果上执行。因此，得到聚集值之后可以用来做排名。本来也可以不用 *student_grades* 视图来完成排名查询，而是用一个单独的 **select** 子句来写。其细节留给读者作为习题。

利用将排名查询嵌入外层查询中的方法，排名函数可用于找出排名最高的 n 个元组。详细实现作为习题。注意，查找排名最低的 n 个元组与查找具有相反排序顺序的排名最高的 n 个元组是一样的。有些数据库系统提供非标准 SQL 扩展直接指定只需得到前 n 个结果。这样的扩展不需要 rank 函数且简化了优化器的工作。例如，一些数据库允许在 SQL 查询后面添加 **limit** n 子句，用来指明只输出前 n 个元组；这个子句可以与 **order by** 子句连用以获取排名最高的 n 个元组。如下面的查询所示，该查询以 GPA 大小顺序来获取前十个学生的 ID 和 GPA：

```
select ID, GPA
from student_grades
order by GPA
limit 10;
```

然而，**limit** 子句不支持分区，所以我们如果不进行排名的话就不能得到每个分区内的前 n 个元组。更甚的是，如果多个学生有相同的 GPA，很有可能其中某个学生被包括到前十，而另一个却不在其中。

还有其他几个可以用来替代 **rank** 的函数。例如，某个元组的 **percent_rank** 以分数的方式给出了该元组的名次。如果某个分区[⊖]中有 n 个元组且某个元组的名次为 r，则该元组的百分比名次定义为 $(r-1)/(n-1)$（如果该分区中只有一个元组则定义为 *null*）。函数 **cume_dist**（累积分布的简写），对一个元组定义为 p/n，其中 p 是分区中排序值小于或等于该元组排序值的元组数，n 是分区中的元组数。函数
194 **row_number** 对行进行排序，并且按行在排序顺序中所处位置给每行一个唯一行号。具有相同排序值的不同的行将按照非确定的方式得到不同的行号。

最后，对于一个给定的常数 n，排名函数 **ntile**(n)按照给定的顺序取得每个分区中的元组，并把它们分成 n 个具有相同元组数目的桶。[⊖]对每个元组，**ntile**(n)给出它所在的桶号（从 1 开始计数）。该函数对构造基于百分比的直方图特别有用。我们可以用下面的查询来演示根据 GPA 把学生分为四个等级：

```
select ID, ntile(4) over (order by (GPA desc)) as quartile
from student_grades;
```

空值的存在可能使排名的定义复杂化，这是因为我们不清楚空值应该出现在排序顺序的什么位置。SQL 允许用户通过使用 **nulls first** 或 **nulls last** 以指定它们出现的位置，例如：

```
select ID, rank() over (order by GPA desc nulls last) as r_rank
from student_grades;
```

5.5.2 分窗

窗口查询用来对于一定范围内的元组计算聚集函数。这个特性很有用。比如计算一个固定时间区间的聚集值；这个时间区间被称为一个窗口。窗口可以重叠，这种情况下一个元组可能对多个窗口都有贡献。这与此前我们看到的分区是不一样的，因为分区查询中一个元组只对一个分区有贡献。

⊖ 如果没有使用某式的分区，则全部集合看成一个单独的分区。

⊖ 如果一个分区中的所有元组的数量不能被 n 整除，则每个桶中的元组数最多相差 1。为了使每个桶中的元组数量相同，具有同样排序属性值的元组可能不确定地赋予不同的桶。

趋势分析是分窗的应用案例之一，这里考虑我们前面讲的销售货物的例子。由于天气等原因，销售量可能会一天天很大程度地变化（例如暴风雪、洪水、飓风、地震在一段时间内一般会降低销售量）。然而，经历足够长的一段时间，影响就会较小（继续前面的例子，由于天气原因造成的销售量下降可能会"赶上去"）。另一个使用分窗查询的例子是股票市场的趋势分析。我们在交易和投资的网站上可以看到各种各样的"移动平均线"。

要写查询来计算一个窗口的聚集值，用我们已经学到的那些特性是相对简单的，例如计算一个固定的三天时间区间的销售量。但是，如果我们想对每个三天时间区间都如此计算，那么查询就变得棘手了。

SQL 提供了分窗特性用于支持这样的查询。假设我们有一个视图 *tot_credits*(*year*, *num_credits*)，它记录了每年学生选读的总学分。[⊖]注意，在这个关系中对于每个年份最多有一个元组。考虑如下查询： 195

```
select year, avg(num_credits)
        over (order by year rows 3 preceding)
        as avg_total_credits
from tot_credits;
```

这个查询计算指定顺序下的这三个元组的均值。因此，对于 2009 年，如果关系 *tot_credits* 中含有 2008 年和 2007 年分别唯一对应的元组，该窗口定义所求的结果就是 2007 年、2008 年和 2009 年的值的平均数。每年的均值也可以通过类似的方法来计算。对于关系 *tot_credits* 中最早的一年，均值的计算就只包含该年本身，然而对于第二年来说，需要对两年的值来求平均。注意，如果关系 *tot_credits* 在某个特定年份有不止一个元组，那么元组按年份来排序就有多种可能。在这种情况下，若序元组是基于排列顺序而定义的，该顺序取决于具体实现方法，就不是唯一定义的了。

假设我们并不想回溯固定数量的元组，而是把截面所有年份都包含在窗口内。这意味着要关注的前面的年数并不固定。我们编写下面的代码来得到截面所有年的平均总学分：

```
select year, avg(num_credits)
        over (order by year rows unbounded preceding)
        as avg_total_credits
from tot_credits;
```

也可以用关键字 **following** 来替换 **preceding**。如果我们在例子中这样做，*year* 值就表示窗口的起始年份，而不是结束年份。类似地，我们可以指定一个窗口，它在当前元组之前开始、在其之后结束：

```
select year, avg(num_credits)
        over (order by year rows between 3 preceding and 2 following)
        as avg_total_credits
from tot_credits;
```

我们还可以用 **order by** 属性上值的范围而不是用行的数目来指定窗口。为了得到一个包含当前年以及前面四年的范围，我们可以这样写： 196

```
select year, avg(num_credits)
        over (order by year range between year - 4 and year)
        as avg_total_credits
from tot_credits;
```

一定要注意上例中的关键字 **range** 的使用。对于 2010 年，从 2006 年到 2010 年（包括边界）的数据都将被返回，不管该范围内实际有多少元组。

在我们的例子里，所有元组都是针对整个学校而言的。假如不是这样，而是把每个系的学分数据用视图 *tot_credits_dept*(*dept_name*, *year*, *num_credits*)来表示，它记录了学生在指定年份中选某个系开设的课程的所有学分数。（我们仍旧把对该视图的定义留给读者作为练习。）我们编写如下的分窗查询，按照 *dept_name* 分区，对每个系分别计算：

⊖ 按照本书中大学的例子来定义这个视图，我们把这个作为练习留给读者。

```
select dept_name, year, avg(tot_credits)
        over ( partition by dept_name
            order by year rows between 3 preceding and current row)
        as avg_total_credits
from tot_credits_dept;
```

5.6 OLAP **

联机分析处理(OLAP)系统是一个交互式系统，它允许分析人员查看多维数据的不同种类的汇总数据。联机一词表示分析人员必须能够提出新的汇总数据的请求，几秒钟之内在线得到响应，无需为了看到查询结果而被迫等待很长的时间。

有很多可用的 OLAP 产品，其中有些随 Microsoft SQL Server 和 Oracle 等数据库产品绑定，另一些是独立的工具。许多 OLAP 工具的最初版本以数据驻留在内存中为前提。小规模数据可以用电子表格进行分析，例如 Excel。然而，对于超大规模数据的 OLAP，需要把数据存储在数据库中，并通过数据库的支持来高效地完成数据预处理和在线查询处理。在本节中，我们将研究 SQL 支持上述任务的扩展特性。

5.6.1 联机分析处理

考虑这样一个应用，某商店想要找出流行的服装款式。我们假设衣服的特性包括商品名、颜色和尺寸，并且我们有一个关系 *sales*，它的模式是

sales(*item_name*, *color*, *clothes_size*, *quantity*)

假设 *item_name* 可以取的值为 skirt、dress、shirt、pants；*color* 可以取的值为 dark、pastel、white；*clothes_size* 可以取的值为 small、medium、large；*quantity* 是一个整数值，表示一个给定条件|*item_name*, *color*, *clothes_size*|的商品数量。关系 *sales* 的一个实例如图 5-16 所示。

item_name	color	clothes_size	quantity
skirt	dark	small	2
skirt	dark	medium	5
skirt	dark	large	1
skirt	pastel	small	11
skirt	pastel	medium	9
skirt	pastel	large	15
skirt	white	small	2
skirt	white	medium	5
skirt	white	large	3
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
shirt	dark	small	2
shirt	dark	medium	6
shirt	dark	large	6
shirt	pastel	small	4
shirt	pastel	medium	1
shirt	pastel	large	2
shirt	white	small	17
shirt	white	medium	1
shirt	white	large	10
pants	dark	small	14
pants	dark	medium	6
pants	dark	large	0
pants	pastel	small	1
pants	pastel	medium	0
pants	pastel	large	1
pants	white	small	3
pants	white	medium	0
pants	white	large	2

图 5-16 关系 *sales* 的例子

197
|
198

统计分析通常需要对多个属性进行分组。给出一个用于数据分析的关系，我们可以把它的某些属性看作**度量属性**(measure attribute)，因为这些属性度量了某个值，而且可以在其上进行聚集操作。例如，关系 *sales* 的属性 *quantity* 就是一个度量属性，因为它度量了卖出商品的数量。关系的其他属性中的某些(或所有)属性可看作是**维属性**(dimension attribute)，因为它们定义了度量属性以及度量属性的汇总可以在其上进行观察的各个维度。在关系 *sales* 中，*item_name*、*color* 和 *clothes_size* 是维属性。(关系 *sales* 的更真实的版本还包括另一些维属性，例如时间和销售地点，以及其他的度量属性，例如该次销售的货币值。)

能够模式化为维属性和度量属性的数据统称为**多维数据**(multidimensional data)。

为了分析多维数据，管理者可能需要查看如图 5-17 所示那样显示的数据。该表显示了 *item_name* 和 *color* 值之间不同组合情况下的商品数目。*clothes_size* 的值指定为 **all**，意味着表中显示的数值是在所有的 *size* 值上的数据汇总(也就是说，我们把"small"、"medium"和"large"的商品都汇总到一个组里)。

clothes_size **all**

item_name \ *color*	dark	pastel	white	total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pants	20	2	5	27
total	62	54	48	164

图 5-17 关系 *sales* 的关于 *item_name* 和 *color* 的交叉表

图 5-17 所示的表是一个**交叉表**(cross-tabulation 或简写为 cross-tab)的例子，也可称之为**转轴表**(pivot-table)。一般说来，一个交叉表是从一个关系(如 R)导出来的，由关系 R 的一个属性(如 A)的值构成其行表头，关系 R 的另一个属性(如 B)的值构成其列表头。例如，在图 5-17 中，属性 *item_name* 对应 A(属性值为"skirt"、"dress"、"shirt"和"pants")，而属性 *color* 对应 B(取值"dark"、"pastel"和"white")。

每个单元可记为(a_i, b_j)，其中 a_i 代表 A 的一个取值，b_j 代表 B 的一个取值。转轴表中不同单元的值按照如下方法从关系 R 推导出来：如果对于任何(a_i, b_j)值，最多只存在一个元组具有该值，则该单元中的值由那个元组得到(如果存在那个元组的话)。例如，它可以是该元组中一个或多个其他属性的值。如果对于一个(a_i, b_j)值，可能存在多个元组具有该值，则该单元中的值必须由这些元组上的聚集得到。在我们的例子中，所用的聚集是所有不同 *clothes_size* 上的 *quantity* 属性值之和，如图 5-17 交叉表上方的"*clothes_size*: **all**"所表明的那样。这样，单元(skirt, pastel)的值就是 35，因为关系 *sales* 有三个元组满足该条件，它们的值分别为 11、9 和 15。

在我们的例子中，交叉表还另有一列和一行，用来存储一行/列中所有单元的总和。大多数的交叉表中都有这样的汇总行和汇总列。

将二维的交叉表推广到 n 维，可视作一个 n 维立方体，称为**数据立方体**(data cube)。图 5-18 表示一个在 *sales* 关系上的数据立方体。该数据立方体有三个维，即 *item_name*、*color* 和 *clothes_size*，其度量属性是 *quantity*，每个单元由这三个维的值确定。正如交叉表一样，数据立方体中的每个单元包含了一个值。在图 5-18 中，一个单元包含的值显示在该单元的一个面上；该单元其他可见的面显示为空白。所有的单元都包含值，即使它们并不可见。某一维的取值可以是 **all**，此时就如交叉表显示的情形一样，该单元包含了该维上所有值的汇总数据。

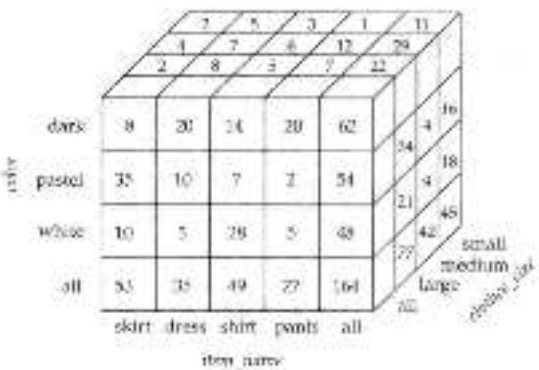


图 5-18 三维数据立方体

将元组分组做聚集的不同方式有很多。在图 5-18 的例子中，有三种颜色、四类商品以及三种尺码，从而立方体的大小为 $3 \times 4 \times 3 = 36$。要是把汇总值也包括进来，我们就得到一个 $4 \times 5 \times 4$ 的立方体，其大小为 80。事实上，对于一个 n 维的表，可在其 n 个维[⊖]的 2^n 个子集上进行分组并执行聚集操作。

在 OLAP 系统中，数据分析人员可以交互地选择交叉表的属性，从而能够在相同的数据上查看不同内容的交叉表。每个交叉表是一个多维数据立方体上的二维视图。例如，数据分析人员可以选择一个在 *item_name* 和 *clothes_size* 上的交叉表，也可以选择一个在 *color* 和 *clothes_size* 上的交叉表。改变交叉表中的维的操作叫做**转轴**(pivoting)。

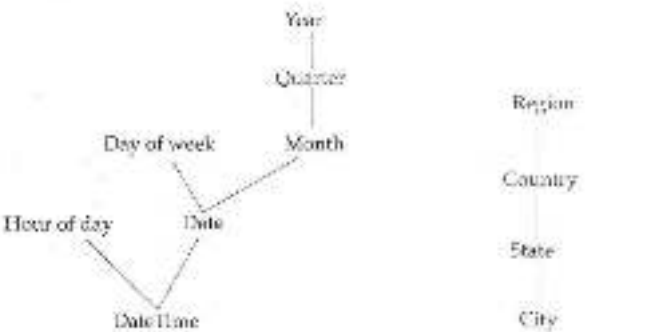
⊖ 在所有的 n 个维的集合上分组只有在该表含有重复数据的情况下才是有意义的。

OLAP 系统允许分析人员对于一个固定的 *clothes_size* 值（比如 large 而不是所有 size 的总和）来查看一个在 *item_name* 和 *color* 上的交叉表。这样的操作称为切片（slicing），因为它可以看作是查看一个数据立方体的某一片。该操作有时也叫做切块（dicing），特别是当有多个维的值是固定的时候。

当一个交叉表用于观察一个数据立方体时，不属于交叉表部分的维属性的值显示在交叉表的上方。如图 5-17 所示，这样的属性值可以是 **all**，表示交叉表中的数据是该属性所有值的汇总。切片/切块仅由这些属性所选的特定值构成，这些值显示在交叉表的上方。

OLAP 系统允许用户按照概念的粒度级别观察数据。从较细粒度数据转到较粗粒度（通过聚集）的操作叫做上卷（rollup）。在我们的例子中，从表 *sales* 上的数据立方体出发，在属性 *clothes_size* 上执行上卷操作得到我们例子中的交叉表。相反的操作，也就是将较粗粒度数据转化为较细粒度数据，称作下钻（drill down）。显然，较细粒度数据不可能由较粗粒度数据产生；它们必须由原始数据产生，或者由更细粒度的汇总数据产生。

分析人员可能希望从不同的细节层次观察某一个维。例如，类型为 **datetime** 的属性包括一天的日期和时间。对于一个只对粗略时间感兴趣的分析人员来说，使用精确到秒（或者更小）的时间可能是无意义的，因为他可能只查看小时的值。一个对一周中某天的销售情况感兴趣的分析人员可能会将日期映射到一周的某天并只查看映射后的信息。还有的分析人员可能会对一个月、一个季度或一年的聚集数据感兴趣。



a) 时间层次结构　　b) 地理位置层次结构

图 5-19　不同的维上的层次结构

一个属性的不同细节层次可以组织成一个层次结构（hierarchy）。图 5-19a 显示了一个在 **datetime** 属性上的层次结构。作为另一个例子，图 5-19b 显示了地理位置的层次结构：city 在层次结构的最底层，state 在它上层，country 在生上一层，region 在最顶层。在前面的例子中，衣服可以按类别分组（例如，男装或女装），这样，在关于衣服的层次结构中，*category* 将在 *item_name* 之上。在实际值层面上，裙子和女服在女装类别下面，短裤和衬衫在男装类别下面。

分析人员可能对查看区分为男装和女装的衣服销售情况感兴趣，而对单个值不感兴趣。在查看了女装和男装层次上的聚集值之后，分析人员可能想要沿层次结构下钻以便查看单个值。一个正在查看某个细节层次的分析人员也可能沿层次结构上卷以便查看较粗层次上的聚集值。两种层次可以显示在同一个交叉表中，如图 5-20 所示。

clothes_size: all

category	*item_name*	*color*			
		dark	pastel	white	total
womenswear	skirt	8	8	10	58
	dress	20	20	5	30
	subtotal	28	28	15	68
menswear	pants	14	14	28	49
	shirt	20	20	8	27
	subtotal	34	34	33	76
total		62	62	48	164

图 5-20　对 *sales* 在属性 *item_name* 上分层后的交叉表

5.6.2　交叉表与关系表

交叉表与通常存储在数据库中的关系表是不同的。这是因为交叉表中的列的数目依赖于实际的数据。数据值的变化可能导致增加更多的列。这对于数据存储来说是不期望见到的。然而，作为向用户做的展现，交叉表是比较令人满意的。对于没有汇总值的交叉表，可以把它直接表示为具有固定列数的关系形式。对于有汇总行/列的交叉表，可以通过引入一个特殊值 **all** 以表示于汇总值，如图 5-21 示。实际上，SQL 标准使用 **null** 值代替 **all**，但是，为了避免与通常的 **null** 值混淆，我们将继续沿用 **all**。

考虑元组（skirt，**all**，**all**，53）和（dress，**all**，**all**，35）。这两个元组是这样得到的：我们消除在 *color* 和 *clothes_size* 上具有不同值的各个元组，然后将 *quantity* 的值替换成一个聚集值（即数量的和），

这样就得到了这两个元组。值 **all** 可以被看作是代表一个属性的所有值的集合。在 *color* 和 *clothes_size* 这两个维上具有 **all** 值的元组可以通过在关系 *sales* 的列 *item_name* 上进行 **group by** 操作之后进行聚集得到。类似地，在 *color*、*clothes_size* 上进行 **group by** 可用于得到在 *item_name* 上取 **all** 值的元组。不带任何属性的 **group by**（在 SQL 中直接将其省略）可用于得到在 *item_name*、*color* 和 *clothes_size* 上都取 **all** 值的元组。

item_name	color	clothes_size	quantity
skirt	dark	all	8
skirt	pastel	all	35
skirt	white	all	10
skirt	all	all	53
dress	dark	all	20
dress	pastel	all	10
dress	white	all	5
dress	all	all	35
shirt	dark	all	14
shirt	pastel	all	7
shirt	white	all	28
shirt	all	all	49
pants	dark	all	20
pants	pastel	all	2
pants	white	all	5
pants	all	all	27
all	dark	all	62
all	pastel	all	54
all	white	all	48
all	all	all	164

图 5.21 图 5.17 中数据的关系表达

也可以用关系来表示层次结构。例如，裙子和女服属于女装类，短裤和衬衫属于男装类，这样的层次可以用关系 *itemcategory*(*item_name*, *category*) 来表示。可以把该关系与关系 *sales* 连接，得到一个包含各个商品所属类别的关系；在该连接关系上进行聚集操作，我们就可以得到带有层次结构的交叉表。另一个例子是，城市的层次结构可以用单个关系 *city_hierarchy*(ID, *city*, *state*, *country*, *region*) 来表示，也可以用多个关系表示。每个关系把结构中的某层的值映射到其下一层。我们在这里假设城市有唯一的标识符，它存储在属性 ID 中，用来避免当两个城市名称相同时发生混淆，比如，在美国的伊利诺斯州和密苏里州都有叫做 Springfield 的城市。

199
203

OLAP 的实现

最早的 OLAP 系统使用内存中的多维数组存储数据立方体，称作**多维 OLAP**（multidimensional OLAP，MOLAP）系统。后来，OLAP 工具集成到关系系统中，数据存储到关系数据库里。这样的系统称为**关系 OLAP**（relational OLAP，ROLAP）系统。复合系统将一些汇总数据存在内存中，基表数据和另一些汇总数据存在关系数据库中，称之为**混合 OLAP**（hybrid OLAP，HOLAP）系统。

许多 OLAP 系统实现为客户－服务器系统。服务器端包含关系数据库和任何 MOLAP 数据立方体，客户端系统通过与服务器通信获得数据的视图。

在一个关系上计算整个数据立方体（所有的分组）的一种朴素方法是使用任何一种标准算法来计算聚集操作，一次计算一个分组。朴素算法需要对关系进行大量的扫描操作。有一种简单的优化是从表集（*item_name*, *color*, *clothes_size*）中，而不是从原始关系中计算聚集（*item_name*, *color*）。

对于标准的 SQL 聚集函数，我们可以用一个按属性集 B 分组的聚集来计算按属性集 A 进行分组的聚集（若 $A \subseteq B$）。读者可以以此作为练习（见习题 5.24），但是要注意计算 **avg** 时，我们还需要 **count** 值。（对于一些非标准的聚集函数，比如 median，不能像上面那样计算聚集，这里所说的优化不能应用于这样的非可分解的聚集函数。）从另一个聚集而不是从原始关系中计算聚集可以大大减少所读的数据量。更进一步的改进也是有可能的，例如，通过一遍数据扫描计算多个分组。

早期的 OLAP 实现预先计算并存储整个数据立方体，即对所属性的所有子集进行分组。预先计算可以使 OLAP 查询在几秒钟内得出结果，即使数据集可能包含数百万的元组，总计达上 G 的数据。然而，对于 n 维属性，有 2^n 个分组，属性上的层次更增多了这个数量。这导致整个数据立方体通常大于形成数据立方体的原始关系，而且在许多情况下存储整个数据立方体是不可行的。

作为对预先计算并存储所有可能的分组的替代方案，预先计算并存储某些分组，按需计算其他分组是有意义的。从原始关系中计算查询可能需要相当长的时间，取而代之的做法是从其他预先计算的查询中计算这些查询。例如，假设某个查询要求按（*item_name*, *color*）进行分组，它没有预先计算。该查询的结果可以基于（*item_name*, *color*, *clothes_size*）上的汇总计算得出，假设这是我们预先计算了的。对于在考虑用于存储预计算结果的可用存储限制的条件下如何选择一个好的用于预先计算的分组集，请查看相关文献注解。

5.6.3 SQL 中的 OLAP

有些 SQL 实现，例如 Microsoft SQL Server 和 Oracle，支持在 SQL 中使用 **pivot** 子句，用于创建交叉表。对于图 5-16 中的关系 *sales*，使用如下查询：

```
select *
from sales
pivot (
    sum(quantity)
    for color in ('dark', 'pastel', 'white')
)
order by item_name;
```

可以返回图 5-22 所示的交叉表。注意，**pivot** 子句中的 **for** 子句用来指定属性 *color* 的哪些值应该在转轴的结果中作为属性名出现。属性 *color* 本身从结果中被去除，然而其他所有属性都被保留，另外，新产生的属性的值被设定为来自于属性 *quantity*。在多个元组对给定单元有贡献的情况下，**pivot** 子句中的聚集操作指定了把这些值进行汇总的方法。在上面的例子中，对 *quantity* 值求和。

注意，pivot 子句本身并不计算我们在图 5-17 所示的转轴表中所看到的部分和。但是，可以首先像我们马上就要讲的那样生成图 5-21 所示的关系表达，然后对这个表达应用 pivot 子句，以得到同样的结果。这种情况下，**all** 值必须已被列在 **for** 子句中，并且 **order by** 子句需要做调整使得 **all** 被排在最后。

单个 SQL 查询使用基本的 **group by** 结构生成不了数据立方体中的数据，因为聚集是对维属性的若干个不同分组来计算的。由于这个原因，SQL 包含了一些函数，用来生成 OLAP 所需的分组。下面我们讨论这些特性。

SQL 支持 **group by** 结构的泛化形式用于进行 **cube** 和 **rollup** 操作。**group by** 子句中的 **cube** 和 **rollup** 结构允许在单个查询中运行多个 **group by** 查询，结果以单个关系的形式返回，其样式类似于图 5-21 中的关系。

再次以零售商店为例，关系是

sales (*item_name*, *color*, *clothes_size*, *quantity*)

我们可以编写简单的 **group by** 查询来得到每个商品名所对应商品的销售量：

item_name	clothes_size	dark	pastel	white
skirt	small	2	11	2
skirt	medium	5	9	3
skirt	large	1	15	3
dress	small	2	4	2
dress	medium	6	3	3
dress	large	12	3	0
shirt	small	2	4	17
shirt	medium	6	1	1
shirt	large	6	2	10
pants	small	14	1	3
pants	medium	6	0	0
pants	large	0	1	2

图 5-22 对图 5-16 中的关系 *sales* 进行 SQL pivot 操作的结果

```
select item_name, sum(quantity)
from sales
group by item_name;
```

这个查询的结果如图 5-23 所示。注意，这和图 5-17 的最后一列（或者同样可以说，图 5-18 所示立方体的第一行）表达的是同一组数据。

item_name	quantity
skirt	53
dress	35
shirt	49
pants	27

图 5-23 查询结果

类似地，我们可以得到每种颜色的商品的销售量，等等。在 **group by** 子句中用多个属性，我们就能知道在某个参数集合条件下每类商品卖出了多少。例如，我们可以编写下面的代码通过商品名和颜色来拆分关系 sales：

```
select item_name, color, sum(quantity)
from sales
group by item_name, color;
```

图 5-24 给出了上述查询的结果。注意，这和图 5-17 的前四行和前四列（或者同样可以说，图 5-18 所示的立方体的前四行和前四列）表达的是同一组数据。

但是，如果想用这种方式生成整个数据立方体，我们就不得不为以下每个属性集写一个独立的查询：

{(item_name, color, clothes_size), (item_name, color), (item_name, clothes_size), (color, clothes_size), (item_name), (color), (clothes_size), ()}

()表示空的 **group by** 列表。

cube 结构使我们能在一个查询中完成这些任务：

```
select item_name, color, clothes_size, sum(quantity)
from sales
group by cube(item_name, color, clothes_size);
```

上述查询产生了一个模式如下的关系：

(*item_name*, *color*, *clothes_size*, **sum**(*quantity*))

这样，该查询结果实际上是一个关系，对于在特定分组中不出现的属性，在结果元组中包含 *null* 作为其值。例如，在 *clothes_size* 上进行分组所产生的元组模式是(*clothes_size*, **sum**(*quantity*))，通过在属性 *item_name* 和 *color* 上加入 *null*，它们被转换成(*item_name*, *color*, *clothes_size*, **sum**(*quantity*))上的元组。

item_name	color	quantity
skirt	dark	8
skirt	pastel	35
skirt	white	10
dress	dark	20
dress	pastel	10
dress	white	5
shirt	dark	14
shirt	pastel	7
shirt	white	28
pants	dark	20
pants	pastel	2
pants	white	5

图 5-24 查询结果

数据立方体关系经常相当庞大。上面的 cube 查询，有 3 种可能的颜色、4 种可能的商品名和 3 种尺寸，共有 80 个元组。图 5-21 的关系是由 *item_name* 和 *color* 上的分组操作所生成的。它们用 all 来替代空值，这样更容易被一般用户读懂。为了在 SQL 中生成该关系，我们设法把 *null* 替换成 all。查询：

```
select item_name, color, sum(quantity)
from sales
group by cube(item_name, color);
```

产生图 5-21 中的关系，含有 null。使用 SQL 的 **decode** 和 **grouping** 函数可以实现 **all** 的替换。**decode** 函数概念简单但是语法有些难以读懂。细节请查看下面的文本框中的内容。

DECODE 函数

decode 函数用来对元组中的属性进行值的替换。decode 的大体形式是：

decode(*value*, *match-1*, *replacement-1*, *match-2*, *replacement-2*, …, *match-N*, *replacement-N*, *default-replacement*);

它把值(*value*)与匹配(*match*)值进行比较。如果发现匹配，就把属性值用相应的替代值来替换。如果未匹配成功，那么属性值被替换成缺省的替代值(*default-replacement*)。

decode 函数并不像我们期望的那样对 null 值进行处理，这是因为，正如我们在 3.6 节所看到的那样，null 上的谓词等价于 **unknown**，最终被当作成 **false**。为了解决这个问题，我们应用 **grouping** 函数，当参数是 **cube** 或 **rollup** 产生的 null 值时它返回 1，否则返回 0。于是，图 5-21 中的关系可以用下面的查询来计算，*null* 替换成 **all** 了：

```
select decode(grouping(item_name), 1, 'all', item_name) as item_name
    decode(grouping(color), 1, 'all', color) as color
    sum(quantity) as quantity
from sales
group by cube(item_name, color);
```

rollup 结构与 **cube** 结构基本一样，只有一点不同，就是 **rollup** 产生的 **group by** 查询较少一些。我们看到，**group by cube** (*item_name*, *color*, *clothes_size*)生成使用部分属性(或者全部属性，或者不用任何属性)可以产生的全部 8 种实现 **group by** 查询的方法。而在下面的查询中：

```
select item_name, color, clothes_size, sum(quantity)
from sales
group by rollup(item_name, color, clothes_size);
```

group by rollup(*item_name*, *color*, *clothes_size*)只产生四个分组：

{(*item-name*, *color*, *clothes_size*), (*item_name*, *color*), (*item_name*), ()}

注意，**rollup** 中的属性按照顺序不同有所区分：最后一个属性（在我们例子中是 *clothes_size*）只在一个分组中出现，倒数第二个属性出现在两个分组中，以此类推，第一个属性出现在（除了全分组之外
208 的）所有组中。

我们用 **rollup** 产生的特定分组有什么用？这些组对于层次结构（例如图 5-19 所示）具有频繁的实用价值。对于位置的层次结构（*Region*，*Country*，*State*，*City*），我们可能希望用 *Region* 来分组，以得到不同区域的销售情况。之后我们可能希望"下钻"到区域内部的国家层面，也就是说我们将以 *Region* 和 *Country* 来分组。继续下钻，我们希望以 *Region*，*Country* 和 *State* 分组，然后以 *Region*，*Country*，*State* 和 *City* 来分组。**rollup** 结构允许我们为求更深层的细节而设定这样下钻的序列。

多个 **rollup** 和 **cube** 可以在一个单独的 **group by** 子句中使用。例如，下面的查询

```
select item_name, color, clothes_size, sum(quantity)
from sales
group by rollup(item_name), rollup(color, clothes_size);
```

产生了以下分组：

$$\{(item\_name, color, clothes\_size), (item\_name, color), (item\_name), (color, clothes\_size), (color), ()\}$$

为了理解其原因，我们注意到 **rollup**(*item_name*)产生了两个分组：{(*item_name*)，()}，**rollup**(*color*，*clothes_size*)产生了三个分组：{(*color*，*clothes_size*)，(*color*)，()}。这两部分的笛卡儿积得到上面显示的六个分组。

rollup 和 **cube** 子句都不能完全控制分组的产生。例如，我们不能指定让它们只生成分组{(*color*，*clothes_size*)，(*clothes_size*，*item_name*)}。我们可以用在 **having** 子句中使用 **grouping** 结构的方法去产生这类限制条件的分组，实现细节作为习题留给读者。

5.7 总结

- SQL 查询可以从宿主语言通过嵌入和动态 SQL 激发。ODBC 和 JDBC 标准给 C、Java 等语言的应用程序定义接入 SQL 数据库的应用程序接口。程序员能够编写通过这些 API 来访问数据库。
- 函数和过程可以用 SQL 提供的过程扩展来定义，它允许迭代和条件（if-then-else）语句。
- 触发器定义了当某个事件发生而且满足相应条件时自动执行的动作。触发器有很多用处，例如实现业务规则、审计日志，甚至执行数据库系统外的操作。虽然触发器只是在不久前作为 SQL:1999 的一部分加入 SQL 标准的，但是大多数数据库系统已经支持触发器很久了。
209 - 一些查询，如传递闭包，或者可以用迭代表示，或者可以用递归 SQL 查询表示。递归可以用递归视图，或者用递归的 **with** 子句定义。
- SQL 支持一些高级的聚集特性，包括排名和分窗查询，这些特性简化了一些聚集操作的表达方式，并提供了更高效的求值方法。
- 联机分析处理（OLAP）工具帮助分析人员用不同的方式查看汇总数据，使他们能够洞察一个组织的运行。
  1. OLAP 工具工作在以维属性和度量属性为特性的多维数据之上。
  2. 数据立方体由以不同方式汇总的多维数据构成。预先计算数据立方体有助于提高汇总数据的查询速度。
  3. 交叉表的显示允许用户一次查看多维数据的两个维及其汇总数据。
  4. 下钻、上卷、切片和切块是用户使用 OLAP 工具时执行的一些操作。
- 从 SQL:1999 标准开始，SQL 提供了一系列的用于数据分析的操作符，其中包括 **cube** 和 **rollup** 操作。有些系统还支持 **pivot** 子句，可以很方便地生成交叉表。

术语回顾

- JDBC
- ODBC
- 预备语句
- 访问元数据
- SQL 注入
- 嵌入式 SQL

- 游标
- 可更新的游标
- 动态 SQL
- SQL 函数
- 存储过程
- 过程化结构
- 外部语言例程
- 触发器
- before 和 after 触发器
- 过渡变量和过渡表
- 递归查询
- 半满查询
- 排名函数
  - Rank
  - Dense rank
  - Partition by
- 分窗
- 联机分析处理(OLAP)
- 多维数据
  - 度量属性
  - 维属性
  - 转轴
  - 数据立方体
  - 切片和切块
  - 卷和下钻
- 交叉表

实践习题

5.1 描述何种情况下你会选择使用嵌入式 SQL，而不是仅仅使用 SQL 或某种通用程序设计语言。

5.2 写一个使用 JDBC 元数据特性的 Java 函数，该函数用 ResultSet 作为输入参数，并把结果输出为用合适的名字作为列名的表格形式。

5.3 写一个使用 JDBC 元数据特性的 Java 函数，该函数输出数据库中的所有关系列表，为每个关系显示它的属性的名称和类型。

5.4 说明如何用触发器来保证约束"一位教师不可能在一个学期的同一时间段在不同的教室里教课"。(要知道，对关系 *teaches* 或 *section* 的改变都可能使该约束被破坏。)

5.5 写一个触发器，用于当 *section* 和 *time_slot* 更新时维护从 *section* 到 *time_slot* 的参照完整性约束。注意，我们在图 5-8 中写的触发器不包含更新操作。

5.6 为了维护关系 *student* 的 *tot_cred* 属性，完成下列任务：

a. 修改定义在 *takes* 更新时的触发器，使其对于能影响 *tot_cred* 值的所有更新都有效。

b. 写一个能与关系 *takes* 的插入操作相关的触发器。

c. 在什么前提下，关系 *course* 上不创建触发器是有道理的？

5.7 考虑图 5-25 中的银行数据库。视图 *branch_cust* 定义如下：

```
create view branch_cust as
    select branch_name, customer_name
    from depositor, account
    where depositor.account_number = account.account_number
```

假设视图被物化，也就是，这个视图被计算且存储。写触发器来维护这个视图，也就是，该视图在对关系 *depositor* 或 *account* 的插入和删除时保持最新。不必管更新操作。 210 211

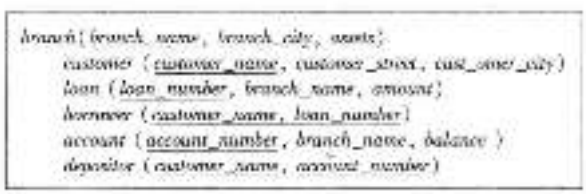


图 5-25 习题 5.7、习题 5.8 和习题 5.28 中用到的银行数据库

5.8 考虑图 5-25 中的银行数据库。写一个 SQL 触发器来执行下列动作：在对账户执行 delete 操作时，对账户的每一个拥有者，检查他是否有其他账户，如果没有，把他从 *depositor* 关系中删除。

5.9 说明如何使用 **rollup** 表达 **group by cube**(a, b, c, d)；答案只允许含有一个 **group by** 子句。

5.10 对于给定的一个关系 S(*student*, *subject*, *marks*)，写一个查询，利用排名操作找出总分数排在前 n 位的学生。

5.11 考虑 5.6 节中的关系 *sales*，写一个 SQL 查询，计算该关系上的立方体(cube)操作，给出图 5-21 中的关系。不要使用 **cube** 结构。

习题

5.12 考虑有如下关系的雇员数据库：

- *emp*(*ename*, *dname*, *salary*)
- *mgr*(*ename*, *mname*)

和图 5-26 中的 Java 代码。该代码使用 JDBC 应用程序接口。假设用户 id、密码、主机名等都正确。请用简洁的语言描述一下 Java 程序都做了什么。（也就是说，用类似于"查找x所部门的经理"这样的话来表达，而不是一行一行地对 Java 语句进行解释。）

```
import java.sql.*;
public class Mystery {
   public static void main(String[] args) {
       try {
           Connection con=null;
           Class.forName("oracle.jdbc.driver.OracleDriver");
           con=DriverManager.getConnection(
               "jdbc:oracle:thin:star/X@//edgar.cse.lehigh.edu:1521/XE");
           Statement s=con.createStatement();
           String q;
           String empName = "dog";
           boolean more;
           ResultSet result;
           do {
               q = "select mname from mgr where ename = '" + empName + "'";
               result = s.executeQuery(q);
               more = result.next();
               if (more) {
                   empName = result.getString("mname");
                   System.out.println (empName);
               }
           } while (more);
           s.close();
           con.close();
       } catch(Exception e){e.printStackTrace();} }}
```

图 5-26 习题 5.12 的 Java 代码

5.13 假设你要在 Java 中定义一个 MetaDisplay 类，它包含方法 static void printTable(String r)，该方法以关系 *r* 为输入参数，执行查询"**select** * **from** *r*"，然后以合适的表格来显示输出结果，表头代表每一列的属性名。

a. 要想以指定的表格形式输入结果，你需要知道关系 *r* 的哪些信息？

b. JDBC 的哪个（些）函数能帮你获取所需的信息？

c. 用 JDBC 应用程序接口来编写方法 printTable(String r)。

5.14 用 ODBC 重新做习题 5.13，定义函数 void printTable(char * r) 来代替原来的方法。

5.15 考虑有两个关系的雇员数据库

employee(*employee_name*, *street*, *city*)
works(*employee_name*, *company_name*, *salary*)

212
213

这里主码用下划线标出。写出一个查询找出这样的公司，它的雇员的平均工资比"First Bank Corporation"的平均工资要高。

a. 使用合适的 SQL 函数。

b. 不使用 SQL 函数。

5.16 使用 **with** 子句而不是函数调用来重写 5.2.1 节的查询，返回数目多于 12 的系的名称和预算。

5.17 把使用嵌入式 SQL 与在 SQL 中使用定义在一个通用程序设计语言中的函数这两种情况进行比较。在什么情况下你会考虑用哪个特性？

5.18 修改图 5-15 中的递归查询来定义一个关系

prereq_depth(*course_id*, *prereq_id*, *depth*)

这里属性 *depth* 表示在课程和先修课程之间存在多少层中间的先修关系。直接的先修课程的深度为0。

5.19 考虑关系模式

part(*part_id*, *name*, *cost*)
subpart(*part_id*, *subpart_id*, *count*)

关系 *subpart* 中的一个元组 $(p_1, p_2, 3)$ 表示部件编号为 p_2 的部件是部件编号为 p_1 的部件的直接子部件，并且 p_1 中包含3个 p_2。注意 p_1 本身可能有自己的子部件。写一个递归SQL查询输出编号为"P-100"的部件的所有子部件。

5.20 再一次考虑习题5.19中的关系模式。用非递归SQL写一个JDBC函数来找出"P-100"的总成本，包括它的所有子部件的成本。注意考虑一个部件可能有重复出现多次的同一个子部件的情况。如果需要，可以在Java中使用递归。

5.21 假设有两个关系 r 和 s，r 的外码 B 参照 s 的主码 A。描述如何用触发器实现从 s 中删除元组时的 **on delete cascade** 选项。

5.22 一个触发器的执行可能会引发另一个被触发的动作。大多数数据库系统都设置了级联的深度限制。解释为什么它们要设置这样的限制。

5.23 考虑图5-27中的关系 r。请给出下面查询的结果。 214

select *building*, *room_number*, *time_slot_id*, **count**(*)
from *r*
group by rollup (*building*, *room_number*, *time_slot_id*)

building	*room_number*	*time_slot_id*	*course_id*	*sec_id*
Garfield	399	A	BIO-101	1
Garfield	399	B	BIO-101	1
Saucon	651	A	CS-101	2
Saucon	550	C	CS-319	1
Painter	705	D	MU-199	1
Painter	403	D	FIN-201	1

图5-27 习题5.23中的关系 r

5.24 对SQL聚集函数 **sum**、**count**、**min** 和 **max** 中的每一个，说明在给定多重集合 S_1 和 S_2 上的聚集值的条件下，如何计算多重集合 $S_1 \cup S_2$ 上的聚集值。

在上述的基础之上，在给定属性 $T \subseteq S$ 上的分组聚集值的条件下，对下面的聚集函数，给出计算关系 $r(A, B, C, D, E)$ 的属性的子集 S 上的分组聚集值的表达式。

a. **sum**、**count**、**min** 和 **max**。

b. **avg**。

c. 标准差。

5.25 在5.5.1节中，我们使用习题4.5中的视图 *student_grades* 来写基于绩点平均值对学生排名次的查询。修改这个查询，只显示前十名的学生（也就是说，从第一名到第十名的那些学生）。

5.26 给出一个具有两个分组的例子，它不能使用带有 **cube** 和 **rollup** 的单个 **group by** 子句表达。

5.27 对于给定的一个关系 $r(a, b, c)$，说明如何使用扩展SQL特性产生一个 c 对 a 的直方图。将 a 分成20个等分的区（即每个区含有 r 中5%的元组，并按 a 排序）。

5.28 考虑图5-25中的银行数据库，以及关系 *account* 的 *balance* 属性。写一个SQL查询，计算 *balance* 值的直方图，从0到目前最大账户余额的范围分成三个相等的区域。 215

工具

大部分数据库销售商把OLAP工具当作他们数据库系统的一部分或作为附加应用程序来提供。这些工具包括微软公司的OLAP工具，Oracle Express 和 Informix Metacube。有些工具被整合到大型"商业智能"产品中，例如IBM Cognos。很多公司还提供面向特定应用（比如客户关系管理）的分析工具，例如Oracle Siebel CRM。

文献注解

对于 SQL 标准和有关 SQL 的书籍请参考第 3 章的文献注解。

java.sun.com/docs/books/tutorial 是一个极好的了解更多最新 Java 和 JDBC 技术资源的站点。有关 Java（包括 JDBC）的书籍也可以在该 URL 中找到。ODBC 的 API 在 Microsoft［1997］和 Sanders［1998］中描述过。Melton 和 Eisenberg［2000］提供了 SQLJ、JDBC 和相关技术的入门向导。更多关于 ODBC、ADO 和 ADO.NET 的信息能在 msdn.microsoft.com/data 上找到。

对于 SQL 中函数和过程的章节，许多数据库产品支持标准说明以外的一些特性，而有些则不支持某些标准内的特性。关于这些特性的更多信息可在各产品的 SQL 用户手册中找到。

最初有关断言和触发器的 SQL 提议在 Astrahan 等［1976］、Chamberlin 等［1976］及 Chamberlin 等［1981］中作了讨论。Melton 和 Simon［2001］、Melton［2002］，以及 Eisenberg 和 Melton［1999］提供了涵盖 SQL:1999 的教科书，该版本的 SQL 标准中首次包含了触发器。

递归查询处理最早是在一种叫做 Datalog 的查询语言中详细研究的，该语言基于数学逻辑，语法类似于逻辑程序设计语言 Prolog 的语法。Ramakrishnan 和 Ullman［1995］对该领域的研究结果做了综述，其中涵盖了从递归定义的视图中查询元组的子集的优化技术。

Gray 等［1995］和 Gray 等［1997］描述了数据立方体操作符。关于计算数据立方体的高效算法在 Agarwal 等［1996］、Harinarayan 等［1996］、Ross 和 Srivastava［1997］中有描述。SQL:1999 中对扩展聚集支持的描述可在数据库系统（比如 Oracle 和 IBM DB2）的产品手册中找到。

目前有大量关于如何高效执行“top-k”查询（即只返回排名为前 k 的结果）的研究。Ilyas 等［2008］对这方面工作做了综述。

[216]

形式化关系查询语言

从第2章～第5章，我们介绍了关系模型，并详细讲述了SQL。在本章中我们将介绍SQL所基于的形式化模型，同时它也是其他关系查询语言的基础。

本章内容包括三种形式化语言。我们首先介绍关系代数，它为广泛应用的SQL查询语言打下了基础。然后我们学习元组关系演算和域关系演算，它们都是基于数学逻辑的声明式查询语言。

6.1 关系代数

关系代数是一种过程化查询语言。它包括一个运算的集合，这些运算以一个或两个关系为输入，产生一个新的关系作为结果。关系代数基本运算有：选择、投影、并、集合差、笛卡儿积和更名。在基本运算以外，还有一些其他运算，即集合交、自然连接和赋值。我们将用基本运算来定义这些运算。

6.1.1 基本运算

选择、投影和更名运算称为一元运算，因为它们对一个关系进行运算。另外三个运算对两个关系进行运算，因而称为二元运算。

6.1.1.1 选择运算

选择(selelct)运算选出满足给定谓词的元组。我们用小写希腊字母sigma(σ)来表示选择，而将谓词写作σ的下标。参数关系在σ后的括号中。因此，为了选择关系*instructor*中属于"物理(Physics)"系的那些元组，我们应写作：

$$\sigma_{dept\_name = \text{"Physics"}}(instructor)$$

如果*instructor*关系如图6-1所示，则从上述查询产生的关系如图6-2所示。

通过书写：

$$\sigma_{salary > 90\,000}(instructor)$$

我们可以找到工资额大于90 000美元的所有元组。

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

图6-1 关系*instructor*

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

图6-2 $\sigma_{dept_name = \text{"Physics"}}(instructor)$的结果

通常，我们允许在选择谓词中进行比较，使用的是=，≠，<，≤，>和≥。另外，我们可以用连词*and*(∧)、*or*(∨)和*not*(¬)将多个谓词合并为一个较大的谓词。因此，为了找到物理系中工资额大于90 000美元的教师，我们需要书写：

$$\sigma_{dept\_name = \text{"Physics"} \wedge salary > 90\,000}(instructor)$$

选择谓词中可以包括两个属性的比较。我们以关系*department*为例说明。为了找出那些系名与楼名相同的系，我们可以这样写：

$$\sigma_{\text{building}=\text{"Watson"}}(department)$$

SQL 与关系代数

关系代数中的术语选择(select)与我们在 SQL 中所使用的关键词 select 具有不一样的含义，这是历史原因造成的。在关系代数中，术语 select 对应于我们在 SQL 中使用的 where。我们在这里强调二者含义的区别，以减少可能出现的混淆。

6.1.1.2 投影运算

假设我们想要列出所有教师的 *ID*、*name* 和 *salary*，而不关心 *dept_name*。**投影**(project)运算使得我们可以产生这样的关系。投影运算是一元运算，它返回作为参数的关系，但把某些属性排除在外。由于关系是一个集合，所以所有重复行均被去除。投影用大写希腊字母 pi(Π)表示，列举所有我们希望在结果中出现的属性作为Π的下标，作为参数的关系在跟随Π后的括号中。因此，我们可以写如下的查询来得到上述的教师列表：

$$\Pi_{ID,name,salary}(instructor)$$

图 6-3 显示了此查询产生的关系。

ID	name	salary
10101	Srinivasan	65000
12121	Wu	90000
15151	Mozart	40000
22222	Einstein	95000
32343	El Said	60000
33456	Gold	87000
45565	Katz	75000
58583	Califieri	62000
76543	Singh	80000
76766	Crick	72000
83821	Brandt	92000
98345	Kim	80000

图 6-3 $\Pi_{ID,name,salary}(instructor)$ 的结果

6.1.1.3 关系运算的组合

关系运算的结果自身也是一个关系，这一事实是非常重要的。考虑一个更复杂的查询"找出物理系的所有教师的名字"，我们写作：

$$\Pi_{name}(\sigma_{dept\_name="Physics"}(instructor))$$

请注意，对投影运算而言，我们没有给出一个关系的名字来作为其参数，而是用一个对关系进行求值的表达式来作为参数。

一般地说，由于关系代数运算的结果同其输入的类型一样，仍为关系，所以我们可以把多个关系代数运算组合成一个**关系代数表达式**(relational-algebra expression)。将关系代数运算组合成关系代数表达式如同将算术运算(如 +、−和 ÷)组合成算术表达式一样。我们将在 6.1.2 节给出关系代数表达式的形式化定义。

6.1.1.4 并运算

假设有一个查询，要找出开设在 2009 年秋季学期或者 2010 年春季学期或者这二者皆开的所有课程的集合。关系 *section*(图 6-4)中包含这些信息。为了找到 2009 年秋季学期开设的所有课程的集合，我们可以这样写：

$$\Pi_{course\_id}(\sigma_{semester="Fall" \wedge year=2009}(section))$$

为了找出在 2010 年春季学期开设的课程，我们可以这样写：

$$\Pi_{course\_id}(\sigma_{semester="Spring" \wedge year=2010}(section))$$

为了实现该查询，我们需要将这两个集合**并**(union)起来；即我们需要出现在这两个集合之一的或同时出现在这两个集合中的所有课程段的 ID。我们通过二元运算"并"来获得这些数据。"并"运算像集合论中一样用∪表示。于是，所需表达式为：

$$\Pi_{course\_id}(\sigma_{semester="Fall" \wedge year=2009}(section)) \cup \Pi_{course\_id}(\sigma_{semester="Spring" \wedge year=2010}(section))$$

此查询产生的关系如图 6-5 所示。需要注意的是，尽管有 3 门课在 2009 年秋季开设，6 门课在 2010 年春季开设，但是在结果中共有 8 个元组。由于关系是集合，所以像 CS-101 这样在两个学期都开设的重复值只留下了单个元组。

可以看到，在我们的例子中，做并运算的两个集合都由 *course_id* 值构成。概括地说，我们必须保证做并运算的关系是相容的。例如，将 *instructor* 关系和 *student* 关系做并运算就没有意义。尽管两个关系都包含 4 个属性，但是二者在 *salary* 和 *tot_cred* 的域上是不同的。在大多数情况下这两个属性的并集

course_id	sec_id	semester	year	building	room_number	time_slot_id
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

图 6-4　关系 *section*

是没有意义的。因此，要使并运算 $r \cup s$ 有意义，我们要求以下两个条件同时成立：

1. 关系 r 和 s 必须是同元的，即它们的属性数目必须相同。

2. 对所有的 i，r 的第 i 个属性的域必须和 s 的第 i 个属性的域相同。

请注意 r 和 s 可以是数据库关系或者作为关系代数表达式结果的临时关系。

course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

图 6-5　2009 年秋季学期或 2010 年春季学期或者这两个学期都开设的课程

6.1.1.5　集合差运算

用 − 表示的**集合差**（set-difference）运算使得我们可以找出在一个关系中而不在另一个关系中的那些元组。表达式 $r - s$ 的结果即一个包含所有在 r 中而不在 s 中的元组的关系。

我们可以通过书写如下表达式来找出所有开设在 2009 年秋季学期但是在 2010 年春季学期不开的课程：

$$\Pi_{course\_id}(\sigma_{semester = \text{'Fall'} \wedge year = 2009}(section)) - \Pi_{course\_id}(\sigma_{semester = \text{'Spring'} \wedge year = 2010}(section))$$

该查询产生的关系如图 6-6 所示。

course_id
CS-347
PHY-101

图 6-6　在 2009 年秋季学期开设但在 2010 年春季学期不开的课程

就像并运算一样，我们必须保证集合差运算在相容的关系间进行。因此，为使集合差运算 $r - s$ 有意义，我们要求关系 r 和 s 是同元的，且对于所有的 i，r 的第 i 个属性的域与 s 的第 i 个属性的域都相同。

6.1.1.6　笛卡儿积运算

用 × 表示的**笛卡儿积**（Cartesian-product）运算使得我们可以将任意两个关系的信息组合在一起。我们将关系 r_1 和 r_2 的笛卡儿积写作 $r_1 \times r_2$。

回忆一下，关系定义为一组域上的笛卡儿积的子集。从这个定义，我们应该已经对笛卡儿积运算的定义有了直观的认识。但是，由于相同的属性名可能同时出现在 r_1 和 r_2 中，我们需要提出一个命名机制来区别这些属性。我们这里采用的方式是找到属性所来自的关系，把关系名附加到该属性上。例如，$r = instructor \times teaches$ 的关系模式为：

(*instructor.ID*, *instructor.name*, *instructor.dept_name*, *instructor.salary*
teaches.ID, *teaches.course_id*, *teaches.sec_id*, *teaches.semester*, *teaches.year*)

用这样的模式，我们可以区别 *instructor.ID* 和 *teaches.ID*。对那些只在两个关系模式之一中出现的属性，我们通常省略其关系名前缀。这样的简化不会导致任何歧义。这样，我们就可以将 r 的关系模式写作：

(*instructor.ID*, *name*, *dept_name*, *salary*
teaches.ID, *course_id*, *sec_id*, *semester*, *year*)

这个命名规则规定作为笛卡儿积运算参数的关系名字必须不同。这一规定有时会带来一些问题，例如当某个关系渐要与自身做笛卡儿积时。当我们在笛卡儿积中使用关系代数表达式的结果时也会产生类似问题，因为我们必须给关系一个名字以引用其属性。在 6.1.1.7 节中，我们将学习如何通过更名运算来避免这些问题。

我们已经知道 $r = instructor \times teaches$ 的关系模式，那么到底哪些元组会出现在 r 中呢？或许你已经想到了，如下所述每一个可能的元组对都形成 r 的一个元组，元组对中一个来自关系 *instructor*（图 6-1）而另一个来自关系 *teaches*（图 6-7）。因此，正如你可以从图 6-8 看到的那样，r 是一个很大的关系，而该图只显示了构成 r 的部分元组。[1]

ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

图 6-7 关系 *teaches*

inst.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2009
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2010
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2009
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2010
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2010
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
15151	Mozart	Music	40000	10101	CS-101	1	Fall	2009
15151	Mozart	Music	40000	10101	CS-315	1	Spring	2010
15151	Mozart	Music	40000	10101	CS-347	1	Fall	2009
15151	Mozart	Music	40000	12121	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2010
15151	Mozart	Music	40000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2009
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2010
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2009
22222	Einstein	Physics	95000	10101	FIN-201	1	Spring	2010
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...

图 6-8 *instructor* × *teaches* 的结果

假设 *instructor* 中有 n_1 个元组，*teaches* 中有 n_2 个元组。那么我们可以有 $n_1 \times n_2$ 种方式来选择元组对——每个关系中选出一个元组；因此 r 中有 $n_1 \times n_2$ 个元组。特别地，请注意对 r 中的某些元组 t 来说，可能 $t[instructor.ID] \neq t[teaches.ID]$。

一般地说，如果有关系 $r_1(R_1)$ 和 $r_2(R_2)$，则关系 $r_1 \times r_2$ 的模式是 R_1 和 R_2 串接而成的。关系 R 中包含所有满足以下条件的元组 t：r_1 中存在元组 t_1 且 r_2 中存在元组 t_2，使得 $t[R_1] = t_1[R_1]$ 且 $t[R_2] = t_2[R_2]$。

[1] 注意，在图 6-8 和图 6-9 中，为了减小表格的宽度，我们把 *instructor.ID* 重命名为 *inst.ID*。

假设我们希望找出物理系的所有教师，以及他们教授的所有课程。为了实现这一要求，我们同时需要关系 *instructor* 和关系 *teaches* 中的信息。如果我们写：

$$\sigma_{dept\_name = \text{"Physics"}}(instructor \times teaches)$$

则其结果就是图 6-9 所示关系。

inst_ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
22222	Einstein	Physics	95000	10101	CS-437	1	Fall	2009
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2010
22222	Einstein	Physics	95000	12121	FIN-201	1	Spring	2010
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2009
22222	Einstein	Physics	95000	32343	HIS-351	1	Spring	2010
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
33456	Gold	Physics	87000	10101	CS-437	1	Fall	2009
33456	Gold	Physics	87000	10101	CS-315	1	Spring	2010
33456	Gold	Physics	87000	12121	FIN-201	1	Spring	2010
33456	Gold	Physics	87000	15151	MU-199	1	Spring	2010
33456	Gold	Physics	87000	22222	PHY-101	1	Fall	2009
33456	Gold	Physics	87000	32343	HIS-351	1	Spring	2010
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

图 6-9 $\sigma_{dept\_name = \text{"Physics"}}(instructor \times teaches)$ 的结果

这里我们得到的关系中只包含物理系的教师。然而，*course_id* 列却可能包含并非这些教师所教授的课程。（如果你不明白为什么会发生这种情况，请不要忘了笛卡儿积中保留了所有可能的由一个来自 *instructor* 的元组和一个来自 *teaches* 的元组构成的元组对。）

由于笛卡儿积运算将 *instructor* 中的每个元组同 *teaches* 中的每个元组进行联系，所以我们可以确定，如果一名教师来自于物理系并教授某门课程（在关系 *teaches* 中有对应的元组），那么 $\sigma_{dept\_name = \text{"Physics"}}(instructor \times teaches)$ 中必定存在某个元组，它包含该教师的名字，并且满足 *instructor.ID* = *teaches.ID*。因此，如果我们用：

$$\sigma_{instructor.ID = teaches.ID}(\sigma_{dept\_name = \text{"Physics"}}(instructor \times teaches))$$

就可以得到在 *instructor* × *teaches* 中那些只与物理系教师以及他们所教课程相关的元组。

最后，由于我们只需要物理系教师的名字，以及他们所教课程的 *course_id*，我们进行投影：

$$\Pi_{name, course\_id}(\sigma_{instructor.ID = teaches.ID}(\sigma_{dept\_name = \text{"Physics"}}(instructor \times teaches)))$$

此表达式的结果如图 6-10 所示，这就是我们的查询的正确答案。可以看到，虽然 Gold 属于物理系，但是（从关系 *teaches* 可知）他不教课，因此未出现在结果中。

name	*course_id*
Einstein	PHY-101

图 6-10 $\Pi_{name, course\_id}(\sigma_{instructor.ID = teaches.ID}(\sigma_{dept\_name = \text{"Physics"}}(instructor \times teaches)))$ 的结果

217
225

需要注意，用关系代数表达查询的方法并不唯一。考虑如下查询：

$$\Pi_{name, course\_id}(\sigma_{instructor.ID = teaches.ID}((\sigma_{dept\_name = \text{"Physics"}}(instructor)) \times teaches))$$

注意这两个查询的细微差别：上面的查询是对 *instructor* 进行选择运算，把 *dept_name* 限制为 Physics，之后再进行笛卡儿积；相比之下，在前面介绍的查询中，在选择运算之前先计算笛卡儿积。然而这两种查询是等价的（equivalent），也就是说，在任何数据库上它们都能得到相同的结果。

6.1.1.7 更名运算

关系代数表达式的结果没有可供我们引用的名字，这一点与数据库中的关系有所不同。如果能给它们赋上名字那将是十分有用的；我们可以通过小写希腊字母 rho（ρ）表示的**更名**（rename）运算来完成这一任务。对给定关系代数表达式 E，表达式

$$\rho_x(E)$$

返回表达式 E 的结果，并把名字 x 赋给了它。

对于一个关系 r 来说，它自身被认为是一个（平凡的）关系代数表达式。因此，我们也可以将更名运算运用于关系 r，这样可得到具有新名字的一个相同的关系。

更名运算的另一形式如下。假设关系代数表达式 E 是 n 元的，则表达式

$$\rho_{x(A_1, A_2, \cdots, A_n)}(E)$$

返回表达式 E 的结果，并赋给它名字 x，同时将各属性更名为 A_1，A_2，…，A_n。

我们以查询"找出大学里的最高工资"作为例子来演示关系的更名运算。我们的策略是：首先计算出一个由非最高工资组成的临时关系，然后计算关系 $\Pi_{salary}(instructor)$ 和刚才算出的临时关系之间的集合差，从而得到结果。

1. 步骤 1：为了计算该临时关系，我们需要比较所有工资的值。要做这样的比较，可以通过计算笛卡儿积 $instructor \times instructor$ 并构造一个选择来比较任意两个出现在同一元组中的工资。我们的首要任务是设计一种机制来区别两个 *salary* 属性：我们将使用更名运算来改变其中一个对教师关系进行引用的名字；这样我们就可以无歧义地两次引用这个关系。

226

现在可以把非最高工资构成的临时关系写作：

$$\Pi_{instructor.salary}(\sigma_{instructor.salary < d.salary}(instructor \times \rho_d(instructor)))$$

通过这一表达式可以得到 *instructor* 中满足如下条件的那些工资：在关系 *instructor*（更名为 *d*）中还存在比它更高的工资。结果中包含了除最高工资以外的所有工资。图 6-11 表示这个关系。

salary
65000
90000
40000
60000
87000
75000
62000
72000
80000
92000

图 6-11 子表达式 $\Pi_{instructor.salary}(\sigma_{instructor.salary < d.salary}(instructor \times \rho_d(instructor)))$ 的结果

2. 步骤 2：查找大学里最高工资的查询可写作：

$$\Pi_{salary}(instructor) - \Pi_{instructor.salary}(\sigma_{instructor.salary < d.salary}(instructor \times \rho_d(instructor)))$$

该查询的结果如图 6-12 所示。

salary
95000

图 6-12 大学里的最高工资

更名运算不是必需的，因为可以对属性使用位置标记。我们可以用位置标记隐含地作为关系的属性名，用 $1、$2、…指代第一个属性、第二个属性，以此类推。位置标记也适用于关系代数表达式运算的结果。下面的关系代数表达式演示了位置标记的用法，我们用它来写先前介绍的计算非最高工资的表达式：

$$\Pi_{\$4}(\sigma_{\$4 < \$8}(instructor \times instructor))$$

注意，笛卡儿积把两个关系的属性串接起来。因此，在笛卡儿积 $(instructor \times teaches)$ 中 $4 代表第一个 *instructor* 的属性 *salary*，而 $8 代表第二个 *instructor* 的属性 *salary*。如果一个二元运算需要在作为其运算对象的两个关系之间进行区分，类似的位置标记也可以用来作为关系名称。例如，$R1 可以指代第一个作为运算对象的关系，而 $R2 可以指代第二个关系。然而，位置标记对人而言不够方便，因为属性的位置是一个数字，不像属性名那样易于记忆。所以在这本书里我们不采用位置标记。

6.1.2 关系代数的形式化定义

6.1.1 节中的运算使我们能够给出关系代数中表达式的完整定义。关系代数中基本的表达式是如下二者之一：

- 数据库中的一个关系。
- 一个常数关系。

常数关系可以用在{}内列出它的元组来表示，例如：{(22222, Einstein, Physics, 95000), (76543, Singh, Finance, 80000)}。

关系代数中一般的表达式是由更小的子表达式构成的。设 E_1 和 E_2 是关系代数表达式，则以下这些都是关系代数表达式：

- $E_1 \cup E_2$。
- $E_1 - E_2$。
- $E_1 \times E_2$。
- $\sigma_P(E_1)$，其中 P 是 E_1 的属性上的谓词。
- $\Pi_S(E_1)$，其中 S 是 E_1 中某些属性的列表。
- $\rho_x(E_1)$，其中 x 是 E_1 结果的新名字。

6.1.3 附加的关系代数运算

关系代数的基本运算足以表达任何关系代数查询。但是，如果我们把自己局限于基本运算，某些常用查询表达出来会显得冗长。因此，我们定义附加的一些运算，它们不能增强关系代数的表达能力，却可以简化一些常用的查询。对每个新运算，我们给出一个只采用基本运算的等价表达式。

227
|
228

6.1.3.1 集合交运算

我们要定义的第一个附加的关系代数运算是集合交(intersection)(∩)。假设我们希望找出在2009年秋季和2010年春季都开设的课程。使用集合交运算，我们可以写为

$$\Pi_{course\_id}(\sigma_{semester="Fall" \wedge year=2009}(section)) \cap \Pi_{course\_id}(\sigma_{semester="Spring" \wedge year=2010}(section))$$

此查询产生的关系如图6-13所示。

course_id
CS-101

图6-13 在2009年秋季学期和2010年春季学期都开设的课程

可注意，任何使用了集合交的关系代数表达式，我们都可以通过用一对集合差运算替代集合交运算来重写，如下所示：

$$r \cap s = r - (r - s)$$

因此，集合交不是基本运算，不能增强关系代数的表达能力。只不过写 $r \cap s$ 比写 $r - (r - s)$ 要方便。

6.1.3.2 自然连接运算

对某些要涉到笛卡儿积的查询进行简化常常是我们需要的。通常情况下，涉及笛卡儿积的查询中会包含一个对笛卡儿积结果进行选择的运算。该选择运算大多数情况下会要求进行笛卡儿积的两个关系在所有相同属性上的值相一致。

6.1.1.6中的例子把表 *instructor* 和 *teaches* 的信息相结合，匹配条件是要求是 *instructor.ID* 和 *teaches.ID* 相等。在这两个关系中只有它们有相同的属性名。

二元运算自然连接使得我们可以将某些选择和笛卡儿积运算合并为一个运算。我们用连接(join)符号⋈来表示它。自然连接运算首先形成它的两个参数的笛卡儿积，然后基于两个关系模式中都出现的属性上的相等性进行选择，最后还要去除重复属性。回到关系 *instructor* 和 *teaches* 的例子中，计算 *instructor* ⋈ *teaches* 的自然连接时，只考虑由 *instructor* 和 *teaches* 中在相同属性 *ID* 上有相同值的元组组成 229
的元组对。结果关系如图6-14所示，只包含13个元组，每个元组记录了某个教师及其实际教授的一门课程的信息。注意，我们并不重复记录那些在两个关系的模式中都出现的属性。还要注意所列出的属性的顺序：排在最前面的是两个关系模式的相同属性，其次是只属于第一个关系模式的属性，最后是只属于第二个关系模式的属性。

尽管自然连接的定义很复杂，这种运算使用起来却很方便。举例来说，考虑一下查询"找出所有教师的姓名，连同他们教的所有课程的 *course_id*"，用自然连接我们可以将此查询表述如下：

$$\Pi_{name, course\_id}(instructor \bowtie teaches)$$

ID	name	dept_name	salary	course_id	sec_id	semester	year
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

图 6-14 关系 *instructor* 同 *teaches* 进行自然连接的结果

由于 *instructor* 和 *teaches* 的模式中具有相同属性 *ID*，自然连接运算只考虑在 *ID* 上值相同的元组对。它将每个这样的元组对合并为单一的元组，其模式为两个模式的并，即(*ID*, *name*, *dept_name*, *salary*, *course_id*)。在执行投影后，我们得到的关系如图 6-15 所示。

name	course_id
Srinivasan	CS-101
Srinivasan	CS-315
Srinivasan	CS-347
Wu	FIN-201
Mozart	MU-199
Einstein	PHY-101
El Said	HIS-351
Katz	CS-101
Katz	CS-319
Crick	BIO-101
Crick	BIO-301
Brandt	CS-190
Brandt	CS-319
Kim	EE-181

图 6-15 $\Pi_{name, course\_id}$ (*instructor* ⋈ *teaches*) 的结果

考虑两个关系模式 R 和 S，当然，它们都是属性名的一个列表。如果我们认为模式是集合而不是列表，我们可以用 $R \cap S$ 表示同时出现在 R 和 S 中的那些属性名，用 $R \cup S$ 表示出现在 R 中或 S 中或在二者中都出现的那些属性名。类似地，出现在 R 中而不出现在 S 中的属性名用 $R - S$ 表示，出现在 S 中而不出现在 R 中的属性名用 $S - R$ 表示。请注意这里的并、交、差运算都是属性的集合上的，而不是关系上的。

我们现在就可以给出自然连接的形式化定义。设 $r(R)$ 和 $s(S)$ 是两个关系。r 和 s 的**自然连接**(natural join)表示为 $r \bowtie s$，是模式 $R \cup S$ 上的一个关系，其形式化定义如下：

$$r \bowtie s = \Pi_{R \cup S}(\sigma_{r.A_1 = s.A_1 \wedge r.A_2 = s.A_2 \wedge \cdots \wedge r.A_n = s.A_n}(r \times s))$$

其中 $R \cap S = \{A_1, A_2, \cdots, A_n\}$。

请注意如果关系 $r(R)$ 和 $s(S)$ 不含有任何相同属性，即 $R \cap S = \varnothing$，那么 $r \bowtie s = r \times s$。

这里再给出一个使用自然连接的例子，写查询"找出计算机系(Comp. Sci.)的所有教师，以及他们教授的所有课程的名称"。

$$\Pi_{name, title}(\sigma_{dept\_name = \text{'Comp. Sci.'}}(instructor \bowtie teaches \bowtie course))$$

此查询的结果关系如图 6-16 所示。

name	title
Brandt	Game Design
Brandt	Image Processing
Katz	Image Processing
Katz	Intro. to Computer Science
Srinivasan	Intro. to Computer Science
Srinivasan	Robotics
Srinivasan	Database System Concepts

图 6-16 $\Pi_{name, title}(\sigma_{dept\_name = \text{'Comp. Sci.'}}(instructor \bowtie teaches \bowtie course))$ 的结果

注意上式中我们在 *instructor* ⋈ *teaches* ⋈ *course* 中没有加入括号来表明自然连接在这三个关系间执行的顺序。上述情况下有两种可能：

$$(instructor \bowtie teaches) \bowtie course$$
$$instructor \bowtie (teaches \bowtie course)$$

我们没有说明我们希望的是哪个表达式，因为二者是等价的。也就是说，自然连接是可结合的(associative)。

theta 连接是自然连接的扩展，它使得我们可以把一个选择运算和一个笛卡儿积运算合并为单独的一个运算。考虑关系 $r(R)$ 和 $s(S)$，并设 θ 是模式 $R \cup S$ 的属性上的谓词，则 theta 连接(theta join)运算 $r \bowtie_{\theta} s$ 定义如下：

$$r \bowtie_{\theta} s = \sigma_{\theta}(r \times s)$$

6.1.3.3 赋值运算

有时通过给临时关系变量赋值的方法来写关系代数表达式会很方便。**赋值**(assignment)运算用 ← 表示，与程序语言中的赋值类似。为了说明这个运算，我们来看自然连接运算的定义。我们可以把 $r \bowtie s$ 写作：

$$\begin{aligned} temp1 &\leftarrow R \times S \\ temp2 &\leftarrow \sigma_{r.A_1 = s.A_1 \wedge r.A_2 = s.A_2 \wedge \cdots \wedge r.A_n = s.A_n}(temp1) \\ result &= \Pi_{R \cup S}(temp2) \end{aligned}$$

赋值的执行不会使得把某个关系显示给用户看，而是将←右侧的表达式的结果赋给←左侧的关系变量。该关系变量可以在后续的表达式中使用。

使用赋值运算，我们可以把查询表达为一个顺序程序，该程序由一系列赋值加上一个其值被作为查询结果显示的表达式组成。对关系代数查询而言，赋值必须是赋给一个临时关系变量。对永久关系的赋值形成了对数据库的修改。请注意，赋值运算不能增强关系代数的表达能力，但是可以使复杂查询的表达变得简单。

6.1.3.4 外连接运算

外连接(outer join)运算是连接运算的扩展，可以处理缺失的信息。假设有一些教师不教课。那么关系 *instructor*(图 6-1)中这些教师对应的元组将不满足与关系 *teaches*(图 6-7)进行自然连接的条件。所以在图 6-14 所示的自然连接的结果中找不到他们的数据。以教师 Califieri、Gold 和 Singh 为例，因为他们不教课，所以就不会出现在自然连接的结果中。

230
↓
232

更一般地，参加连接的一个或两个关系中的某些元组可能会这样地"丢失"。外连接(outer join)运算与前面学过的自然连接运算比较类似，不同之处在于它在结果中创建带空值的元组，以此来保留在连接中丢失的那些元组。

我们可以用外连接运算来避免上述的信息丢失。外连接运算有三种形式：左外连接，用⟕表示；右外连接，用⟖表示；全外连接，用⟗表示。这三种形式的外连接都要计算连接，然后在连接结果中添加额外的元组。例如，表达式 *instructor* ⟕ *teaches* 和 *teaches* ⟖ *instructor* 的结果分别如图 6-17 和图 6-18 所示。

ID	name	dept_name	salary	course_id	sec_id	semester	year
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
33456	Gold	Physics	87000	null	null	null	null
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
58583	Califieri	History	62000	null	null	null	null
76543	Singh	Finance	80000	null	null	null	null
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

图 6-17 *instructor* ⟕ *teaches* 的结果

ID	courseId	secId	semester	year	name	dept_name	salary
10101	CS-101	1	Fall	2009	Srinivasan	Comp. Sci.	65000
10101	CS-315	1	Spring	2010	Srinivasan	Comp. Sci.	65000
10101	CS-347	1	Fall	2009	Srinivasan	Comp. Sci.	65000
12121	FIN-201	1	Spring	2010	Wu	Finance	90000
15151	MU-199	1	Spring	2010	Mozart	Music	40000
22222	PHY-101	1	Fall	2009	Einstein	Physics	95000
32343	HIS-351	1	Spring	2010	El Said	History	60000
33456	*null*	*null*	*null*	*null*	Gold	Physics	87000
45565	CS-101	1	Spring	2010	Katz	Comp. Sci.	75000
45565	CS-319	1	Spring	2010	Katz	Comp. Sci.	75000
58583	*null*	*null*	*null*	*null*	Califieri	History	62000
76543	*null*	*null*	*null*	*null*	Singh	Finance	80000
76766	BIO-101	1	Summer	2009	Crick	Biology	72000
76766	BIO-301	1	Summer	2010	Crick	Biology	72000
83821	CS-190	1	Spring	2009	Brandt	Comp. Sci.	92000
83821	CS-190	2	Spring	2009	Brandt	Comp. Sci.	92000
83821	CS-319	2	Spring	2010	Brandt	Comp. Sci.	92000
98345	EE-181	1	Spring	2009	Kim	Elec. Eng.	80000

图 6-18 *teaches* ⟕ *instructor* 的结果

左外连接(left outer join)(⟕)取出左侧关系中所有与右侧关系的任一元组都不匹配的元组，用空值填充所有来自右侧关系的属性，再把产生的元组加到自然连接的结果中。图 6-17 中，元组(58583, Califieri, History, 62000, *null*, *null*, *null*, *null*)即是这样的元组。所有来自左侧关系的信息在左外连接结果中都得到保留。

右外连接(right outer join)(⟖)与左外连接相对称：用空值填充来自右侧关系的所有与左侧关系的任一元组都不匹配的元组，将结果加到自然连接的结果中。图 6-18 中，(58583, *null*, *null*, *null*, *null*, Califieri, History, 62000)即是这样的元组。因此，所有来自右侧关系的信息在右外连接结果中都得到保留。

全外连接(full outer join)(⟗)既做左外连接又做右外连接，既填充左侧关系中与右侧关系的任一元组都不匹配的元组，又填充右侧关系中与左侧关系的任一元组都不匹配的元组，并把结果都加到连接的结果中。

注意，从左外连接的例子到右外连接的例子，我们交换了运算对象的顺序，这样它们都保留了 *instructor* 的元组，因此包含相同的内容。对于我们的示例关系，*teaches* 的元组在 *instructor* 中总能找到与之对应的元组，所以 *teaches* ⟕ *instructor* 与 *teaches* ⟖ *instructor* 产生的结果是一样的。如果在 *teaches* 中有元组在 *instructor* 中找不到对应元组，该元组将会出现在 *teaches* ⟕ *instructor* 的结果以及 *teaches* ⟖ *instructor* 的结果中，所默认的部分用空值填充。外连接的更多例子(以 SQL 语法表达)请参考 4.1.2 节。

由于外连接可能产生含有空值的结果，我们需要了解不同的关系代数运算是如何处理空值的。3.6 节涉及 SQL 环境中的这个问题。相同的概念也适用于关系代数的情况，我们在这里不再赘述。

233
↓
234

请注意，很有趣的是外连接运算可以用基本关系代数运算表示。例如，左连接运算 $r ⟕ s$ 可以写成：

$$(r ⨯ s) ∪ (r - \Pi_R(r ⋈ s)) ⨯ \{(null, \cdots, null)\}$$

其中常数关系 $\{(null, \cdots, null)\}$ 的模式是 $S - R$。

6.1.4 扩展的关系代数运算

我们现在来介绍另外几个关系代数运算，它们可以实现一些不能用基本的关系代数运算来表达的查询。这些运算被称为**扩展的关系代数**(extended relational-algebra)运算。

6.1.4.1 广义投影

第一个运算是广义投影(generalized-projection)，它通过允许在投影列表中使用算术运算和字符串函数等来对投影进行扩展。广义投影运算形式为：

$$\Pi_{F_1, F_2, \cdots, F_n}(E)$$

其中 E 是任意关系代数表达式，而 F_1，F_2，…，F_n 中的每一个都是涉及常量以及 E 的模式中属性的算术表达式。最基本的情况下算术表达式可以仅仅是一个属性或常量。通常来说，在表达式中可以使用

对数值属性或者产生数值结果的表达式的 +、−、×、/等代数运算。广义投影还允许其他数据类型上的运算，比如对字符串的串接。

例如，表达式：

$$\Pi_{ID,name,dept\_name,salary/12}(instructor)$$

可以得到每个教师的 *ID*、*name*、*dept_name* 以及每月的工资。

6.1.4.2 聚集

第二个扩展的关系代数运算是聚集运算 $\mathcal{G}$，可以用来对值的集合使用聚集函数，例如计算最小值或者求平均值。

聚集函数(aggregate function)输入值的一个汇集，将单一值作为结果返回。例如，聚集函数 **sum** 输入值的一个汇集，返回这些值的和。因此，将函数 **sum** 用于汇集

{1, 1, 3, 4, 4, 11}

235

上，返回值 24。聚集函数 **avg** 返回值的平均，当用于上述汇集，返回值为 4。聚集函数 **count** 返回汇集中元素的个数，对上述汇集将返回 6。除此之外，常用聚集函数还有 **min** 和 **max**，它们分别返回汇集中的最小值和最大值，对上面的汇集分别返回 1 和 11。

使用聚集函数对其进行操作的汇集中，一个值可以出现多次，值出现的顺序是无关紧要的。这样的汇集称为**多重集**(multiset)。集合(set)是多重集的特例，其中每个值都只出现一次。

我们用关系 *instructor* 来说明聚集的概念。假设我们希望找出所有教师的工资总和，这一查询的关系代数表达式为：

$$\mathcal{G}_{sum(salary)}(instructor)$$

符号 $\mathcal{G}$ 是字母 G 的书法体；读作"书法体 G"。关系代数运算 $\mathcal{G}$ 表示聚集将被应用，它的下标说明采用的聚集运算。上述表达式的结果是一个单一属性的关系，且只包含单独的一行，其值表示所有教师的工资总和。

有时，在计算聚集函数前我们必须去除重复值。如果我们想去除重复，我们仍然使用前面的函数名，但用连字符将"**distinct**"附加在函数名后(如 **count-distinct**)。以查询"找出在 2010 年春季学期教课的教师数"为例。在这里，每名教师只应计算一次，而不管他教了几门课程。关系 *teaches* 包含了所需的信息，所以我们把此查询表达如下：

$$\mathcal{G}_{count\text{-}distinct(ID)}(\sigma_{semester='Spring' \wedge year=2010}(teaches))$$

聚集函数 **count-distinct** 可以确保：即使某位教师授课多于一门，在结果中也只对他计数一次。

有时候我们希望对一组元组集合而不是单个元组集合执行聚集函数。作为示例，考虑查询"求出每个系的平均工资"。我们把此查询表达如下：

$$_{dept\_name}\mathcal{G}_{average(salary)}(instructor)$$

图 6-19 显示了通过 *dept_name* 属性对关系 *instructor* 进行分组得到的元组，这是计算查询结果的第一步。然后对每个组执行指定的聚集，查询结果如图 6-20 所示。

ID	name	dept_name	salary
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

图 6-19 通过属性 *dept_name* 对关系 *instructor* 分组后的元组

dept_name	salary
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

图 6-20 查询"求出每个系的平均工资"的结果关系

作为对比，对于查询"求出所有教师的平均工资"，我们可以如下表达：

$$\mathcal{G}_{\text{average}\to\text{avg\_salary}}(instructor)$$

此时，从运算符 $\mathcal{G}$ 的左边去掉了属性 *dept_name*，这样一来整个关系就被当作单个组来执行聚集。

多重集关系代数

与关系代数不同的是，SQL 允许在输入关系以及查询结果中存在元组的多重拷贝。SQL 标准说明了如下定义：在一个查询的输出结果中每个元组有多少个拷贝取决于所对应的元组在输入关系中的拷贝个数。

为了模拟 SQL 的这种模式，我们定义了一种被称为**多重集关系代数**（multiset relational algebra）的关系代数版本来对多重集（即存在重复的集合）进行操作。多重集关系代数的基本运算有如下定义：

1. 如果在 r_1 中元组 t_1 有 c_1 份拷贝，并且 t_1 满足选择 σ_θ，那么在 $\sigma_\theta(r_1)$ 中元组 t_1 有 c_1 份拷贝。

2. 对于 r_1 中元组 t_1 的每份拷贝，在 $\Pi_A(r_1)$ 中都有一个与之对应的 $\Pi_A(t_1)$，$\Pi_A(t_1)$ 表示单个元组 t_1 的投影。

3. 如果在 r_1 中元组 t_1 有 c_1 份拷贝，在 r_2 中元组 t_2 有 c_2 份拷贝，那么在 $r_1 \times r_2$ 中就有元组 $t_1.t_2$ 的 $c_1 * c_2$ 份拷贝。

例如，假设关系 r_1 的模式是 (A, B)，r_2 的模式是 (C)，它们是如下的多重集：

$$r_1 = \{(1, a), (2, a)\},\ r_2 = \{(2), (3), (3)\}$$

那么 $\Pi_B(r_1)$ 就得到 $\{(a), (a)\}$，而 $\Pi_B(r_1) \times r_2$ 的结果是

$$\{(a, 2), (a, 2), (a, 3), (a, 3), (a, 3), (a, 3)\}$$

根据 SQL 中的相应含义（见 3.5 节），我们还可以用相似的方法来定义多重集的并、交以及集合差运算。而对于聚集运算来说，在多重集下的定义并没有什么变化。

聚集运算（aggregation operation）$\mathcal{G}$ 通常的形式如下：

$$_{G_1, G_2, \cdots, G_n}\mathcal{G}_{F_1(A_1), F_2(A_2), \cdots, F_n(A_n)}(E)$$

其中 E 是任意关系代数表达式，G_1，G_2，…，G_n 是用于分组的一系列属性；每个 F_i 是一个聚集函数，每个 A_i 是一个属性名。运算的含义如下，表达式 E 的结果中元组以如下方式被分成若干组：

1. 同一组中所有元组在 G_1，G_2，…，G_n 上的值相同。

2. 不同组中元组在 G_1，G_2，…，G_n 上的值不同。

236 | 238

因此，各组可以用属性 G_1，G_2，…，G_n 上的值来唯一标识。对每个组 $(g_1, g_2, \cdots, g_n)$ 来说，结果中有一个元组 $(g_1, g_2, \cdots, g_n, a_1, a_2, \cdots, a_n)$，其中对每个 i，a_i 是将聚集函数 F_i 作用于该组的属性 A_i 上的多重值集所得到的结果。

作为聚集运算的特例，属性列 G_1，G_2，…，G_n 可以是空的，在这种情况下，会有唯一一组包含关系中所有的元组。这相当于没有分组。

SQL 与关系代数

通过关系代数运算和 SQL 运算的比较，可以清楚地看到二者之间联系紧密。典型的 SQL 查询解式如下：

select $A_1, A_2, \cdots, A_n$
from $r_1, r_2, \cdots, r_m$
where P

每个 A_i 表示一个属性，每个 r_i 代表一个关系。P 表示一个谓词。这个查询等价于多重集关系代数表达式：

$$\Pi_{A_1,A_2,\cdots,A_n}(\sigma_P(r_1 \times r_2 \times \cdots \times r_m))$$

如果省略掉 **where** 子句，那么谓词 P 就是 **true**。

更复杂的 SQL 查询也可以用关系代数来重写。例如，查询：

select A_1, A_2, **sum**(A_3)
from r_1, r_2, …, r_m
where P
group by A_1, A_2

等价于：

$$_{A_1,A_2}\mathcal{G}_{sum(A_3)}(\Pi_{A_1,A_2,\cdots,A_n}(\sigma_P(r_1 \times r_2 \times \cdots \times r_m)))$$

from 子句中的连接表达式可以用关系代数中与之等价的连接表达式来写；我们把详细内容留给读者作为练习。然而 **where** 或者 **select** 子句中的子查询却不能像这样直接用关系代数来重写。这是因为默认与子查询结构等价的关系代数运算。为了解决这个问题，有人提出了对关系代数的扩展，但是这些内容不在本书讨论范围之内。

6.2 元组关系演算

当书写关系代数表达式时，提供了产生查询结果的过程序列，这个序列能生成查询的答案。与之相比，元组关系演算是非过程化的（nonprocedural）查询语言。它只描述所需信息，而不给出获得该信息的具体过程。

元组关系演算中的查询表达为：

$$\{t \mid P(t)\}$$

[239]

也就是说，它是所有使谓词 P 为真的元组 t 的集合。和前面的记法一样，我们用 $t[A]$ 表示元组 t 在属性 A 上的值，并用 $t \in r$ 表示元组 t 在关系 r 中。

在给出元组关系演算的形式化定义之前，我们先回头看几个在 6.1.1 节中我们用关系代数表达式书写过的查询。

6.2.1 查询示例

我们希望找出所有工资在 80 000 美元以上的教师的 *ID*、*name*、*dept_name* 和 *salary*：

$$\{t \mid t \in instructor \wedge t[salary] > 80000\}$$

假设我们只需要 *ID* 属性，而不是关系 *instructor* 的所有属性。为了用元组关系演算来书写这个查询，我们需要为模式（*ID*）上的关系写一个表达式。我们需要（*ID*）上在 *instructor* 中对应元组的属性 *salary* > 80 000 的那些元组。为了表述这样的要求，我们将要引入数理逻辑中的"存在"这一结构。记法

$$\exists t \in r(Q(t))$$

表示"关系 r 中存在元组 t 使谓词 $Q(t)$ 为真"。

用这种记法，我们可以将查询"找出工资大于 80 000 美元的所有教师的 *ID*"表述为：

$$\{t \mid \exists s \in instructor(t[ID] = s[ID] \wedge s[salary] > 80000)\}$$

我们可以这样来读上述表达式："它是所有满足如下条件的元组 t 的集合：在关系 *instructor* 中存在元组 s 使 t 和 s 在属性 *ID* 上的值相等，且 s 在属性 *salary* 上的值大于 80 000 美元"。

元组变量 t 只定义在 *ID* 属性上，因为这一属性是对 t 进行限制的条件所涉及的唯一属性。对比，结果得到（*ID*）上的关系。

考虑查询"找出位置在 Watson 楼的系中的所有教师姓名"。这个查询比前一个稍微复杂一些，因为它涉及 *instructor* 和 *department* 两个关系。但是，正如我们将看到的一样，所需的只不过是在元组关系

演算表达式中使用两个"存在"子句，并通过 *and*(∧)把它们连接起来。我们将此查询表达如下：

$$\{t \mid \exists s \in instructor(t[name] = s[name] \\ \wedge \exists u \in department(u[dept\_name] = s[dept\_name] \\ \wedge u[building] = \text{"Watson"})\}$$

[240]

元组变量 u 保证该系位于 Watson 楼，元组变量 s 被限制到与 u 的 *dept_name* 相同。查询产生的结果如图 6-21所示。

为了找出在 2009 年秋季学期或者 2010 年春季学期或者这两个学期都开设的所有课程的集合，我们在关系代数中使用了并运算。在元组关系演算中，我们将用到用 *or*(∨)连接的两个"存在"子句：

$$\{t \mid \exists s \in section(t[course\_id] = s[course\_id] \\ \wedge s[semester] = \text{"Fall"} \wedge s[year] = 2009) \\ \vee \exists u \in section(u[course\_id] = t[course\_id]) \\ \wedge u[semester] = \text{"Spring"} \wedge u[year] = 2010)\}$$

name
Einstein
Crick
Gold

图 6-21 位置在 Watson 楼的系中的所有教师姓名

此表达式给出所有至少满足下列两个条件之一的 *course_id* 元组的集合：

- 在关系 *section* 中满足 *semester* = Fall 且 *year* = 2009 的某个元组包含该 *course_id*。
- 在关系 *section* 中满足 *semester* = Spring 且 *year* = 2010 的某个元组包含该 *course_id*。

如果同一门课在 2009 年秋季和 2010 年春季学期都开课，那么它的 *course_id* 在结果中也只出现一次，因为集合的数学定义不允许重复成员。此查询的结果已在前面的图 6-5 中展示。

假设我们现在只想找到那些在 2009 年秋季和 2010 年春季学期都开设的课程的 *course_id*。所需做的只不过是把上述表达式中的 *or*(∨)用 *and*(∧)替代。

$$\{t \mid \exists s \in section(t[course\_id] = s[course\_id] \\ \wedge s[semester] = \text{"Fall"} \wedge s[year] = 2009) \\ \wedge \exists u \in section(u[course\_id] = t[course\_id]) \\ \wedge u[semester] = \text{"Spring"} \wedge u[year] = 2010)\}$$

此查询结果如图 6-13 所示。

现在考虑查询"找出 2009 年秋季学期开设而 2010 年春季学期不开的所有课程"。这一查询的元组 [241] 关系演算表达式同上面的表达式类似，只是使用了 *not*(¬)符号：

$$\{t \mid \exists s \in section(t[course\_id] = s[course\_id]) \\ \wedge s[semester] = \text{"Fall"} \wedge s[year] = 2009) \\ \wedge \neg \exists u \in section(u[course\_id] = t[course\_id]) \\ \wedge u[semester] = \text{"Spring"} \wedge u[year] = 2010)\}$$

这个元组关系演算表达式用 $\exists s \in section(\cdots)$ 子句来要求该 *course_id* 开设在 2009 年秋季学期，用 $\neg \exists u \in section(\cdots)$ 子句来去掉那些作为在 2010 年春季学期开设的课程出现在关系 *section* 中的 *course_id*。

下面我们要考虑的查询将用到蕴含。用 $\Rightarrow$ 表示。公式 $P \Rightarrow Q$ 表示"P 蕴含 Q"，即"如果 P 为真，则 Q 必然为真"。$P \Rightarrow Q$ 逻辑上等价于 $\neg P \vee Q$。用蕴含而不是 *not* 和 *or* 常常可以更直观地表达查询。

我们来看这样一个查询："找出所有那些选了生物(Biology)系全部课程的学生"。为了用元组关系演算书写此查询，我们引入"对所有的"结构，用 $\forall$ 表示。记为

$$\forall t \in r(Q(t))$$

表示"对关系 r 中所有元组 t，Q 均为真"。

此查询的表达式如下：

$$\{t \mid \exists r \in student(r[ID] = t[ID]) \wedge \\ \forall u \in course(u[dept\_name] = \text{"Biology"} \Rightarrow \\ \exists s \in takes(s[ID] = t[ID] \\ \wedge s[course\_id] = u[course\_id]))\}$$

我们可以这样来读上述表达式："它是所有满足如下条件的学生(即(*ID*)上的元组 t)的集合：对关系 *course* 中所有的元组 u，如果 u 在 *dept_name* 属性上的值是'Biology'，那么在关系 *takes* 中一定存在一个包含该学生 *ID* 以及该课程 *course_id* 的元组"。

注意上述查询中有一点很微妙：如果生物系没有开设任何课程，则所有的学生 *ID* 都满足条件。在这种情况下，上述查询表达式的第一行非常关键。如果没有条件：

$$\exists r \in student(r[ID] = t[ID])$$

那么若生物系没有课程，则任何一个 t 值(包括不是关系 *student* 里学生 *ID* 的值)都会符合要求。 [242]

6.2.2 形式化定义

我们现在可以给出形式化定义了。元组关系演算表达式具有如下形式：

$$\{t \mid P(t)\}$$

其中 P 是一个公式。公式中可以出现多个元组变量。如果元组变量不被 $\exists$ 或 $\forall$ 修饰，则称为自由变量。因此，在

$$t \in instructor \wedge \exists s \in department(t[dept\_name] = s[dept\_name])$$

中，t 是自由变量，而元组变量 s 称为受限变量。

元组关系演算的公式由原子构成。原子可以是如下的形式之一：

- $s \in r$，其中 s 是元组变量而 r 是关系(我们不允许使用 $\notin$ 运算符)。
- $s[x] \Theta u[y]$，其中 s 和 u 是元组变量，x 是 s 所基于的关系模式中的一个属性，y 是 u 所基于的关系模式中的一个属性，Θ 是比较运算符($<$，$\leq$，$=$，$\neq$，$>$，$\geq$)；我们要求属性 x 和 y 所属域的成员能用 Θ 比较。
- $s[x] \Theta c$，其中 s 是元组变量，x 是 s 所基于的关系模式中的一个属性，Θ 是比较运算符，c 是属性 x 所属域中的常量。

我们根据如下规则，用原子构造公式：

- 原子是公式。
- 如果 P_1 是公式，那么 $\neg P_1$ 和 (P_1) 也都是公式。
- 如果 P_1 和 P_2 是公式，那么 $P_1 \vee P_2$、$P_1 \wedge P_2$ 和 $P_1 \Rightarrow P_2$ 也都是公式。
- 如果 $P_1(s)$ 是包含自由元组变量 s 的公式，且 r 是关系，则

$$\exists s \in r(P_1(s)) \text{ 和 } \forall s \in r(P_1(s))$$

也都是公式。

正如关系代数中一样，我们也可以写出一些形式上不一样的等价表达式。在元组关系演算中，这种等价性包括如下三条规则：

1. $P_1 \wedge P_2$ 等价于 $\neg(\neg(P_1) \vee \neg(P_2))$。
2. $\forall t \in r(P_1(t))$ 等价于 $\neg \exists t \in r(\neg P_1(t))$。
3. $P_1 \Rightarrow P_2$ 等价于 $\neg(P_1) \vee P_2$。 [243]

6.2.3 表达式的安全性

最后还要讨论一个问题。元组关系演算表达式可能产生一个无限的关系。例如如果我们写出表达式

$$\{t \mid \neg(t \in instructor)\}$$

不在 *instructor* 中的元组有无限多个，大多数这样的元组所包含的值甚至根本不在数据库中！显然，我们不希望有这样的表达式。

为了帮助我们对元组关系演算进行限制，引入了元组关系公式 P 的域(domain)这一概念。直观地说，P 的域(用 $dom(P)$ 表示)是 P 所引用的所有值的集合。它们既包括 P 自身用到的值，又包括 P 中涉及的关系的元组中出现的所有值。因此，P 的域是 P 中显式出现的值及名称出现在 P 中的那些关系的所有值的集合。例如，$dom(t \in instructor \wedge t[salary] > 80\,000)$ 是包括 80 000 和出现在 *instructor* 中的所

有值的集合。另外，$dom(\neg(t \in instructor))$ 是 *instructor* 中出现的所有值的集合，因为关系 *instructor* 在表达式中出现。

如果出现在表达式 $\{t \mid P(t)\}$ 结果中的所有值均来自 $dom(P)$，则我们说表达式 $\{t \mid P(t)\}$ 是安全的。表达式 $\{t \mid \neg(t \in instructor)\}$ 不安全，因为 $dom(\neg(t \in instructor))$ 是所有出现在 *instructor* 中的值的集合，但是很可能有某个不在 *instructor* 中的元组 t，它包含不在 *instructor* 中出现的值。我们在这一节中所写的其他元组关系演算表达式都是安全的。

像 $\{t \mid \neg(t \in instructor)\}$ 这样的不安全表达式中的元组个数是无限的，而安全的表达式一定包含有限的结果。因此我们限定只有那些安全的元组关系演算表达式才被认为是允许的。

6.2.4 语言的表达能力

限制在安全表达式范围内的元组关系演算和基本关系代数（包括运算符 $\cup$、$-$、$\times$、σ、Π 和 ρ，但是没有扩展的关系代数运算符，例如广义投影和聚集（$\mathcal{G}$））具有相同的表达能力。因此，对于每个只运用基本操作的关系代数表达式，都有与之等价的元组关系演算表达式；对于每个元组关系演算表达式，也都有与之等价的关系代数表达式。我们在此不证明这个断言，文献注解中有关于此证明的参考资料。在习题里包含了部分的证明。注意，没有任何一个元组关系演算等价于聚集运算，但是可以
[244] 对它进行扩展以支持聚集。扩展元组关系演算来处理算术表达式是很简单的。

6.3 域关系演算

关系演算的另一种形式称为**域关系演算**（domain relational calculus），使用从属性域中取值的域变量，而不是整个元组的值。尽管如此，域关系演算同元组关系演算联系紧密。

就像关系代数是 SQL 语言的基础一样，域关系演算是被广泛采用的 QBE 语言（参考附录 B.1）的理论基础。

6.3.1 形式化定义

域关系演算中的表达式形式如下：

$$\{< x_1, x_2, \cdots, x_n > \mid P(x_1, x_2, \cdots, x_n)\}$$

其中 x_1，x_2，…，x_n 代表域变量。和元组关系演算的情况一样，P 代表由原子构成的公式。域关系演算中的原子具有如下形式之一：

- $< x_1, x_2, \cdots, x_n > \in r$，其中 r 是 n 个属性上的关系，而 x_1，x_2，…，x_n 是域变量或域常量。
- $x \theta y$，其中 x 和 y 是域变量，而 θ 是比较运算符（$<$，$\leq$，$=$，$\neq$，$>$，$\geq$）。我们要求属性 x 和 y 所属域可用 θ 比较。
- $x \theta c$，其中 x 是域变量，θ 是比较运算符，而 c 是 x 作为域变量的那个属性域中的常量。

根据以下规则，我们用原子构造公式：

- 原子是公式。
- 如果 P_1 是公式，那么 $\neg P_1$ 和 (P_1) 也都是公式。
- 如果 P_1 和 P_2 是公式，那么 $P_1 \vee P_2$，$P_1 \wedge P_2$ 和 $P_1 \Rightarrow P_2$ 也都是公式。
- 如果 $P_1(x)$ 是 x 的一个公式，其中 x 是自由域变量，则

$$\exists x(P_1(x)) \text{ 和 } \forall x(P_1(x))$$

也都是公式。

[245] 我们把 $\exists a, b, c(P(a, b, c))$ 作为 $\exists a(\exists b(\exists c(P(a, b, c))))$ 的简写。

6.3.2 查询的例子

我们现在用域关系演算对前面讲到的例子给出查询表达式。请注意这些表达式和元组关系演算表达式的相似之处。

- 找出工资在 80 000 美元以上的教师的 *ID*、*name*、*dept_name* 和 *salary*：

$$\{< i, n, d, s > \mid < i, n, d, s > \in instructor \wedge s > 80000\}$$

- 找出工资大于 80 000 美元的所有教师的姓名：

$$\{ < i > \mid \exists n,d,s(< i,n,d,s > \in instructor \wedge s > 80000) \}$$

虽然第二个查询看起来同我们在元组关系演算中书写的很相似，但实际上两者有重要区别。在元组演算中，当我们对某个元组变量 s 写 $\exists s$ 时，我们立刻通过 $\exists s \in r$ 将它同某个关系相约束。可是，当我们在域演算中写 $\exists n$ 时，n 不指代一个元组，而指的是一个域值。因此，在用于公式 $< i, n, d, s > \in instructor$ 将 n 束缚到 *instructor* 关系中的教师以前，变量 n 的域是不受约束的。

下面我们给出一些域关系演算的查询示例：

- 找出在物理系的所有教师姓名，以及他们教授的所有课程的 *course_id*：

$$\{ < n,c > \mid \exists i,a,st,y(< i,c,a,s,y > \in teaches \wedge \exists d,s(< i,n,d,s > \in instructor \wedge d = "Physics")) \}$$

- 找出在 2009 年秋季学期或者 2010 年春季学期或者这两学期都开设的所有课程的集合：

$$\{ < c > \mid \exists a,s,y,b,r,t(< c,a,s,y,b,r,t > \in section \wedge s = "Fall" \wedge y = "2009") \vee \exists a,s,y,b,r,t(< c,a,s,y,b,r,t > \in section \wedge s = "Spring" \wedge y = "2010") \}$$

- 找出选了生物系开设的全部课程的所有学生：

$$\{ < i > \mid \exists n,d,tc(< i,n,d,tc > \in student) \wedge \forall ci,ti,da,cr(< ci,n,da,cr > \in course \wedge da = "Biology" \Rightarrow \exists si,se,y,g(< i,ci,si,se,y,g > \in takes)) \}$$

246

注意，如果生物系没开设任何课程，那么结果将包含所有学生。这与元组关系演算中的示例相一致。

6.3.3 表达式的安全性

我们知道，在元组关系演算(6.2 节)中可能写出产生结果为无限关系的表达式。这使得我们为元组关系演算表达式定义安全性。域关系演算中也有类似的情况。像

$$\{ < i,n,d,s > \mid \neg(< i,n,d,s > \in instructor) \}$$

这样的表达式是不安全的，因为它允许在结果中出现不在表达式的域中的值。

在域关系演算中，我们还必须考虑"存在"和"对所有的"子句中公式的形式。我们来看表达式

$$\{ < x > \mid \exists y(< x,y > \in r) \wedge \exists z(\neg(< x,z > \in r) \wedge P(x,z)) \}$$

其中 P 是涉及 x 和 z 的公式。对于公式的第一部分 $\exists y(< x,y > \in r)$，我们在测试时只需考虑 r 中的值。可是，对于公式的第二部分 $\exists z(\neg(< x,z > \in r) \wedge P(x,z))$，我们在测试时必须考虑不在 r 中出现的 x 值。由于所有关系都是有限的，不在 r 中出现的值无限多。因此，通常情况下如果不考虑 x 的取值有无限多种可能，我们就不能完成对第二部分的测试。事实上我们并不这样做。我们通过增加一些约束来限制上面这样的表达式。

在元组关系演算中，我们将所有存在量词修饰的变量限制在某个关系范围内。由于在域关系演算中还没有这样做，我们增加用于定义安全性的规则，以处理类似于上例中出现的情况。如果下列条件同时成立，我们认为表达式

$$\{ < x_1,x_2,\cdots,x_n > \mid P(x_1,x_2,\cdots,x_n) \}$$

是安全的：

1. 表达式的元组中所有值均来自 $dom(P)$。

2. 对每个形如 $\exists x(P_1(x))$ 的"存在"子公式而言，子公式为真当且仅当在 $dom(P_1)$ 中有某个值 x 使 $P_1(x)$ 为真。

3. 对每个形如 $\forall x(P_1(x))$ 的"对所有的"子公式而言，子公式为真当且仅当 $P_1(x)$ 对 $dom(P_1)$ 中所有值 x 均为真。

附加规则的目的是为了保证我们不需要测试无限多的可能性就可以完成对"存在"和"对所有的"子公式的测试。我们看安全性定义中的第二个规则。要使 $\exists x(P_1(x))$ 为真，我们只需找到一个 x 使 $P_1(x)$ 为真。通常，我们需要测试无数个值。但是，如果表达式是安全的，我们就知道可以只注意 $dom(P_1)$ 中的值。这种限制使我们考虑的元组减少到有限个。

247

形如 $\forall x(P_1(x))$ 的子公式情况类似。要断言 $\forall x(P_1(x))$ 为真，需要测试所有可能的值，因此必须检查无限多的值。跟前面一样，如果我们知道表达式是安全的，则只需要对 $dom(P_1)$ 中的值来测试 $P_1(x)$ 就已经足够了。

除了前面我们介绍的不安全查询的例子之外，本节例子中所给出的域关系演算表达式都是安全的。

6.3.4 语言的表达能力

当我们把域关系演算限制在安全表达式范围内时，它与被限制在安全表达式范围内的元组关系演算具有相同的表达能力。我们在前面已经知道受限的元组关系演算与关系代数等价，所以下述三者都是等价的：

- 基本关系代数（不包括扩展关系代数运算）。
- 限制在安全表达式范围内的元组关系演算。
- 限制在安全表达式范围内的域关系演算。

注意没有任何一个域关系演算等价于聚集运算，但是它可以扩展以支持聚集，并且扩展它来处理算术表达式是很简单的。

6.4 总结

- **关系代数**（relational algebra）定义了一集在表上运算且输出结果也是表的代数运算。这些运算可以组合使用来得到表达所希望查询的表达式。关系代数定义了关系查询语言中使用的基本运算。
- 关系代数运算可以分为：
  - 基本运算。
  - 附加的运算。可以用基本运算表达。
  - 扩展的运算。其中的一些扩展了关系代数的表达能力。
- 关系代数是一种简洁的、形式化的语言，不适合于那些偶尔使用数据库系统的用户。因此，商用数据库系统采用有更多"语法修饰"的语言。从第 3 章到第 5 章我们讨论了最有影响力的语言——SQL，它是基于关系代数的。 248
- **元组关系演算**（tuple relational calculus）和**域关系演算**（domain relational calculus）是非过程化语言，代表了关系查询语言所需的基本能力。基本关系代数是一种过程化语言，在能力上等价于被限制在安全表达式范围内的关系演算的这两种形式。
- 关系演算是所述的、形式化的语言，并不适合于那些偶尔使用数据库系统的用户。这两种形式化语言构成了两种更易使用的语言 QBE 和 Datalog 的基础，我们将在附录 B 中介绍它们。

术语回顾

- 关系代数
- 关系代数运算
  - 选择 σ
  - 投影 Π
  - 并 $\cup$
  - 集合差 $-$
  - 笛卡儿积 $\times$
  - 更名 ρ
- 附加的运算
  - 集合交 $\cap$
  - 自然连接 $\bowtie$
  - 赋值运算
  - 外连接
    - 左外连接 $⟕$
    - 右外连接 $⟖$
    - 全外连接 $⟗$
- 多重集
- 分组
- 空值
- 元组关系演算
- 域关系演算
- 表达式安全性
- 语言的表达能力

实践习题

6.1 使用大学的模式，用关系代数来表达下面这些查询：

a. 找出计算机（Comp. Sci.）系开设的占 3 个学分的课程的名称。

b. 找出选了教师 Einstein 所教课程的所有学生的 ID，注意结果中不能有重复。

c. 找出所有教师的最高工资。

d. 找出收入为最高工资的所有教师(可能有多个人工资相同)。

e. 找出2009年秋季学期开设的各个课程段的选课人数。

f. 找出2009年秋季学期开设的各个课程段中最大的选课人数。

g. 找出在2009年秋季学期选课人数最多的课程段。

249

6.2 考虑图6-22所示关系数据库，主码加了下划线。给出关系代数表达式来表示下列每一个查询：

a. 找出与其经理居住在同一城市同一街道的所有员工姓名。

b. 找出此数据库中不在"First Bank Corporation"工作的所有员工姓名。

c. 找出比"Small Bank Corporation"的所有员工收入都高的所有员工姓名。

employee(*person_name*, *street*, *city*)
works(*person_name*, *company_name*, *salary*)
company(*company_name*, *city*)
manages(*person_name*, *manager_name*)

图6-22 习题6.2、习题6.8、习题6.11、习题6.13和习题6.15的关系数据库

6.3 外连接是自然连接的一种扩展，它使得参与运算的关系中的元组在连接结果中不丢失。描述theta连接运算可以怎样扩展，使得在theta连接结果中来自左侧、右侧及两侧关系的元组不被丢失。

6.4 (除运算，division operation) 关系代数中的除运算符是÷，它的定义如下：设$r(R)$和$s(S)$是两个关系，并且$S \subseteq R$；即模式S中的每个属性都在模式R中。那么$r \div s$是模式$R-S$上的关系，即此模式中包含所有在R中而不在S中的属性。元组t属于$r \div s$当且仅当以下两个条件同时成立：

1. t在$\Pi_{R-S}(r)$中。

2. 对s中的每一个元组t_s，在r中都有元组t_r同时满足以下两个条件：

   a. $t_r[S] = t_s[S]$

   b. $t_r[R-S] = t$

有了以上定义，要求：

a. 用除运算符写一个关系代数表达式来找到选了计算机(Comp. Sci.)系课程的所有学生的ID。(提示：在除之前，把*takes*投影到ID和*course_id*，并用一个选择表达式生成所有计算机系课程*course_id*的集合。)

250

b. 如果不用除法，如何在关系代数中表达上述查询。(通过这样做，你就可以了解如何用其他关系代数运算来定义除运算。)

6.5 已知如下关系模式：

$$R = (A, B, C)$$
$$S = (D, E, F)$$

关系$r(R)$和$s(S)$已给出。请给出与下列每个表达式等价的元组关系演算表达式：

a. $\Pi_A(r)$

b. $\sigma_{B=17}(r)$

c. $r \times s$

d. $\Pi_{A,F}(\sigma_{C=D}(r \times s))$

6.6 设$R = (A, B, C)$，r_1和r_2都是模式R上的关系。请分别给出与下列表达式等价的域关系演算表达式：

a. $\Pi_A(r_1)$

b. $\sigma_{B=17}(r_1)$

c. $r_1 \cup r_2$

d. $r_1 \cap r_2$

e. $r_1 - r_2$

f. $\Pi_{A,B}(r_1) \bowtie \Pi_{B,C}(r_2)$

6.7 设$R = (A, B)$，$S = (A, C)$，$r(R)$和$s(S)$为关系。为下列查询写出关系代数表达式：

a. $\{ < a > \mid \exists (b)(< a, b > \in r \wedge b = 7)\}$

b. $\{ < a, b, c > \mid < a, b > \in r \wedge < a, c > \in s\}$

c. $\{ < a > \mid \exists c(< a, c > \in s \wedge \exists b_1, b_2(< a, b_1 > \in r \wedge < c, b_2 > \in r \wedge b_1 > b_2)) \}$

6.8 考虑图 6-22 所示的关系数据库，其中主码用下划线标示。为下面每个查询写出元组关系演算表达式：

251 a. 找出所有直接在经理"Jones"下工作的员工。

b. 找出直接在经理"Jones"下工作的员工居住的所有城市。

c. 找出"Jones"的经理的经理的名字。

d. 找出比住在"Mumbai"的所有员工的收入都高的那些员工。

6.9 描述如何将 SQL 中的连接表达式转换成关系代数。

习题

6.10 使用大学的模式，用关系代数表达如下查询。

a. 找出所有至少选了一门计算机(Comp. Sci.)系课程的学生。

b. 找出所有在 2009 年春季之前没有选任何课程的学生的 ID 和姓名。

c. 对于每个系，找到该系教师的最高工资。可以假设每个系至少有一名教师。

d. 在上一问找到每个系的最高工资的基础上，找出所有各系的最高工资的最小值。

6.11 考虑图 6-22 所示关系数据库，主码加了下划线。给出关系代数表达式来表示下列每一个查询：

a. 找出 First Bank Corporation 的所有员工姓名。

b. 找出 First Bank Corporation 所有员工的姓名和居住城市。

c. 找出 First Bank Corporation 所有年收入在 10 000 美元以上的员工姓名和居住的街道、城市。

d. 找出所有居住地与工作的公司在同一城市的员工姓名。

e. 假设公司可以位于几个城市中。找出满足下面条件的所有公司，它位于 Small Bank Corporation 所位

252 于的每一个城市。

6.12 使用大学的例子，用关系代数查询来找出多于一个教师授课的课程段，分别用以下两种方式来表达：

a. 使用聚集函数。

b. 不使用聚集函数。

6.13 考虑图 6-22 所示的关系数据库。分别给出下列查询的关系代数表达式：

a. 找出员工最多的公司。

b. 找出工资最少的员工所在公司。

c. 找出人均工资比 First Bank Corporation 人均工资高的公司。

6.14 考虑如下关于图书馆的关系模式：

$$member(\underline{memb\_no}, name, dob)$$
$$books(\underline{isbn}, title, authors, publisher)$$
$$borrowed(\underline{memb\_no}, \underline{isbn}, \underline{date})$$

用关系代数写出下列查询：

a. 找出借了任何由 McGraw-Hill 出版的书的成员的姓名。

b. 找出借了由 McGraw-Hill 出版的所有的书的成员的姓名。

c. 找出借了由 McGraw-Hill 出版的 5 本以上不同的书的成员的姓名和成员号。

d. 对每个出版商，找出借了该出版商的 5 本以上的书的成员的姓名和成员号。

e. 找出平均每个成员借了多少本书。下面的情况需要考虑在内：如果某个成员没有借任何书，那么他就根本不会出现在关系 *borrowed* 中。

6.15 考虑图 6-22 所示的员工数据库。为下面每个查询分别写出元组关系演算和域关系演算表达式：

a. 找出所有为 First Bank Corporation 工作的员工名字。

b. 找出所有为 First Bank Corporation 工作的员工名字和居住城市。

c. 找出所有为 First Bank Corporation 工作且年薪超过 10 000 美元的员工名字、居住街道和城市。

253 d. 找出居住城市和公司所在城市相同的所有员工。

e. 找出居住街道和城市与其经理相同的所有员工。

f. 找出数据库中所有本为 First Bank Corporation 工作的员工。

g. 找出工资高于 Small Bank Corporation 的每一位员工的员工。

h. 假设一个公司可以位于好几个城市。找出满足下面条件的所有公司，它位于 Small Bank Corporation 所位于的每一个城市。

6.16 设 $R=(A, B)$ 且 $S=(A, C)$，$r(R)$ 和 $s(S)$ 是关系。分别给出与下列域关系演算表达式等价的关系代数表达式：

a. $\{< a >\mid \exists b(< a,b > \in r \wedge b = 17)\}$

b. $\{< a,b,c >\mid < a,b > \in r \wedge < a,c > \in s\}$

c. $\{< a >\mid \exists b(< a,b > \in r) \vee \forall c(\exists d(< d,c > \in s) \Rightarrow < a,c > \in s)\}$

d. $\{< a >\mid \exists c(< a,c > \in s \wedge \exists b_1,b_2(< a,b_1 > \in r \wedge < c,b_2 > \in r \wedge b_1 > b_2))\}$

6.17 用 SQL 查询代替关系代数表达式来重做习题 6.16。

6.18 设 $R=(A, B)$ 且 $S=(A, C)$，$r(R)$ 和 $s(S)$ 是关系。使用特殊常量 *null*，分别书写等价于下列表达式的元组关系演算表达式：

a. $r \ltimes s$

b. $r \rtimes s$

c. $r \rtimes\ltimes s$

6.19 给出元组关系表达式，找出关系 $r(A)$ 中的最大值。

文献注释

关系代数最初的定义在 Codd[1970] 中给出；关系模型的扩展、关于在关系代数中引入空值的论述（RM/T 模型），以及外连接，这些内容都出现在 Codd[1979] 中。Codd[1990] 是 E. F. Codd 关于关系模型的所有文章的一个概括。外连接在 Date[1993b] 中也进行了讨论。

元组关系演算最初的定义在 Codd[1972] 中给出。元组关系演算和关系代数等价性的形式化证明在 Codd[1972] 中给出。关系演算的一些扩展已经被提出。Klug[1982] 和 Escobar-Molano 等 [1993] 描述了向分级聚集函数的扩展。

254

255

256

第二部分
Part 2

数据库设计

设计数据库系统的目的是为了管理大量信息。这些大量的信息并不是孤立存在的，而是某些企业业务的一部分。这些企业的终端产品可能是来自数据库中的信息；也可能是其他设备或服务，数据库则仅为其扮演一个支持者的角色。

本部分的前两章关注于数据库模式的设计。第7章讲述的实体-联系（E-R）模型是一种高层数据模型，与把所有数据用表来表示的方法不同，它将称作实体的基本对象和这些对象之间的联系区分开来。该模型通常用作数据库模式设计的第一步。

先前的章节曾非正式地介绍了关系数据库设计——关系模式的设计。然而，还存在用于区分好的数据库设计和不好的数据库设计的基本原理。这些原理被形式化为若干“范式”，这些范式提供了在不一致的可能性和特定查询效率之间的不同权衡。第8章讲述关系模式的规范化设计。

设计一个完整的数据库应用环境，并满足被建模企业的需求，需要关注更广泛的问题，很多这样的问题将在第9章讲述。该章首先介绍基于Web的应用程序接口的设计，然后描述如何利用多个抽象层次来构建大型应用。最后，给出了在应用程序级别和数据库级别的安全性的详细讨论。

第7章
Database System Concepts, 6E

数据库设计和 E-R 模型

到现在为止，在本书中我们已经假想了一个给定的数据库模式并研究了如何表述查询和更新。我们现在考虑究竟如何设计一个数据库模式。在本章中，我们关注于实体 - 联系（E-R）数据模型，它提供了一个找出在数据库中表示的实体以及实体间如何关联的方法。最终，数据库设计将会表示为一个关系数据库设计和一个与之关联的约束集合。本章讲述一个 E-R 设计如何转换成一个关系模式的集合以及如何在该设计中找到某些约束。然后，第 8 章详细考查一个关系模式的集合是否为一个好的或不好的数据库设计，并研究使用更广的约束集合来构建好的设计的过程。这两章覆盖数据库设计的基本概念。

7.1 设计过程概览

构建一个数据库应用是一个复杂的任务，包括设计数据库模式，设计访问和更新数据的程序，以及设计控制数据访问的安全模式。用户的需求在设计过程中扮演一个中心角色。本章主要关注于数据库模式的设计，不过本章后面会简要地概括其他几个设计任务。

设计一个完整的数据库应用环境，并满足被建模企业的需求，需要关注广泛的问题。数据库预期用途的这些其他方面在物理级、逻辑级和视图级影响着各种各样的设计选择。

7.1.1 设计阶段

[259] 对于小型的应用，由一个理解应用需求的数据库设计者直接决定要构造的关系、关系的属性以及其上的约束，这样也许是可行的。但是，这种直接的设计过程在现实的应用中是很难的。由于现实的应用常常很复杂，通常没有一个人能够理解应用的所有数据需求。数据库设计者必须与应用的用户进行交互以理解应用的需求，把它们以用户能够理解的高级别的形式表示出来，然后再将需求转化为较低级别的设计。一个高层数据模型为数据库设计者提供了一个概念框架，在该框架中以系统的方式定义了数据库用户的数据需求，以及满足这些需求的数据库结构。

- 数据库设计的最初阶段需要完整地刻画未来数据库用户的数据需求。为完成这个任务，数据库设计者需要同应用领域的专家和用户进行深入的沟通。这一阶段的产品是用户需求规格说明。虽然存在图形方式的用户需求表示技术，但是在本章中，我们仅限于采用文字的方式来描述用户需求。
- 接下来，设计者选择数据模型，并采用所选数据模型的概念将这些需求转化为数据库的概念模式。在此**概念设计**（conceptual design）阶段所产生的模式提供了一个对企业的详细综述。我们在本章中将研究的实体 - 联系模型通常用于表示概念设计。用实体 - 联系模型的术语来说，概念模式定义了数据库中表示的实体，实体的属性，实体之间的联系，以及实体和联系上的约束。通常，概念设计阶段会导致实体 - 联系图的构建，它提供了对模式的图形化描述。

  设计者检查此模式以确保所有数据需求都满足，并且互相不冲突。他还可以检查该设计以去除冗余的特征。在这个阶段，他关注的是描述数据及其联系，而不是定义物理存储细节。
- 完善的概念模式还指明企业的功能需求。在**功能需求规格说明**（specification of functional requirement）中，用户描述将在数据上进行的各类操作（或事务）。操作的例子包括修改或更

新数据，搜索并取回特定数据，以及删除数据。在概念设计的这一阶段，设计者可以检查所设计的模式，以确保其满足功能需求。

- 从抽象数据模型到数据库实现的转换过程在最后两个设计阶段中进行。
  - 在**逻辑设计阶段**（logical-design phase）中，设计者将高层概念模式映射到将使用的数据库系统的实现数据模型上。实现数据模型通常是关系数据模型，该阶段通常包括将以实体-联系模型定义的概念模式映射到关系模式。 260
  - 最后，设计者将所得到的系统特定的数据库模式使用到后续的**物理设计阶段**（physical-design phase）中。在该阶段，指明数据库的物理特征，这些特征包括文件组织格式和索引结构的选择。我们将在第10章和第11章讨论这些内容。

在应用建立之后，要改变数据库的物理模式相对地比较简单。但是，由于可能影响到应用程序代码中散布的大量的查询和更新操作，因此改变逻辑模式的任务执行起来常常更加困难。因此，在建立后续的数据库应用之前，谨慎实施数据库设计阶段是非常重要的。

7.1.2 设计选择

数据库设计过程的一个主要部分是决定如何在设计中表示各种类型的"事物"，比如人、地方、产品，诸如此类。我们使用实体（entity）这个术语来指示所有可明确识别的个体。在一个大学数据库中，实体的例子将包括教师、学生、系、课程和开课[2]。这些各种各样的实体以多种方式互相关联，而所有这些方式都需要在数据库设计中反映出来。例如，一名学生在一次开课中选课，而一名教师在一次开课中授课，授课和选课就是实体间联系的实例。

在设计一个数据库模式的时候，我们必须确保避免两个主要的缺陷。

- **冗余**：一个不好的设计可能会重复信息。例如，如果对于每一次开课我们都存储课程编号和课程名称，那么对于每一次开课我们就冗余地（即多次地、不必要地）存储了课程名称。对每次开课仅存储课程编号，并在一个课程实体中将课程名称和课程编号关联一次就足够了。

  冗余也可能出现在关系模式中。在目前我们所使用的大学的例子中，我们有一个开课信息的关系和一个课程信息的关系。假设换一个做法，我们只有一个关系，对一门课程的每一次开课我们将全部课程信息（课程编号、课程名、系名、学分）重复一次。很明显，关于课程的信息将冗余地存储。

  信息的这种冗余表达的最大问题是当对一条信息进行更新但没有将这条信息的所有拷贝都更新时，这条信息的拷贝会变得不一致。例如，拥有同一个课程编号的几次不同的开课可 261
  能会拥有不同的课程名称，于是会搞不清楚课程的正确名称是什么。理想的情况下，信息应该只出现在一个地方。
- **不完整**：一个不好的设计可能会使得企事业机构的某些方面难于甚至无法建模。例如，假设在上述案例（1）中，我们只有对应于开课的实体，而没有对应于课程的实体。从关系的角度说，这就是假设我们有单个关系，对一门课程的每一次开课都重复存储课程的所有信息。那么一门新课程的信息将无法表示，除非开设了该课程。我们可能会尝试将开课信息设置为空值的方法来解决这个有问题的设计。这种绕开问题的措施不仅不吸引人，而且有可能由于主码约束而无法实行。

仅仅避免不好的设计是不够的。可能存在大量好的设计，我们必须从中选择一个。作为一个简单的例子，考虑买了某产品的一个客户。该产品的销售是客户和产品之间的联系吗？还是销售本身是一个与客户和产品都相关的实体？这个选择，虽然简单，却可能对企业某些方面建模的好坏有很大的影响。考虑为一个现实企业中的大量实体和联系做类似这样的选择的需要，不难看出数据库设计是一个很有挑战性的问题。事实上我们将看到，它需要科学和"好的品味"的结合。

---

[2] 一门课程可能在多个学期开设，同样也可能一个学期中开设多次，我们称某课程的每次开设为一个课程段。

7.2 实体-联系模型

实体-联系（entity-relationship，E-R）数据模型的提出旨在方便数据库的设计。它是通过允许定义代表数据库全局逻辑结构的企业模式实现的。

E-R 模型在将现实世界企业的含义和交互映射到概念模式上非常有用，因此，许多数据库设计工具都利用了 E-R 模型的概念。E-R 数据模型采用了三个基本概念：实体集、联系集和属性，我们首先对此进行学习；E-R 模型还有一个相关联的图形表示（E-R 图），我们在本章后面学习。

7.2.1 实体集

实体（entity）是现实世界中可区别于所有其他对象的一个“事物”或“对象”。例如，大学中的每个人都是一个实体。每个实体有一组性质，其中一些性质的值可以唯一地标识一个实体。例如，一
[262] 个人可能具有 *person_id* 性质唯一标识这个人。因此，*person_id* 的值 677-89-9011 将唯一标识大学中一个特定的人。与此类似，课程也可以看作实体，而 *course_id* 唯一标识了大学中的某个课程实体。实体可以是实实在在的，如人或书；也可以是抽象的，如课程、课程段开课或者机票预订。

实体集（entity set）是相同类型即具有相同性质（或属性）的一个实体集合。例如，一所给定大学的所有教师的集合可定义为实体集 *instructor*。类似地，实体集 *student* 可以表示大学中所有学生的集合。

在建模的过程中，我么通常抽象地使用术语实体集，而不是指某个个别实体的特别集合。我们用术语实体集的**外延**（extension）来指属于实体集的实体的实际集合。因此，大学中实际教师的集合构成了实体集 *instructor* 的外延。我们在第 2 章看到过的联系和联系实例之间的区别和上述区别类似。

实体集不必互不相交。例如，可以定义大学里所有人的实体集（*person*）。一个 *person* 实体可以是 *instructor* 实体，可以是 *student* 实体，可以既是 *instructor* 实体又是 *student* 实体，也可以都不是。

实体通过一组**属性**（attribute）来表示。属性是实体集中每个成员所拥有的描述性性质。为某实体集指定一个属性表明数据库为该实体集中每个实体存储相似的信息；但每个实体在每个属性上都有各自的值。实体集 *instructor* 可能具有属性 *ID*、*name*、*dept_name*、和 *salary*。在现实生活中，可能还有更多的属性，如街道号、房间号、州、邮政编码和国家，但是为了使我们的例子简单，我们省略了这些属性。*course* 实体集可能的属性有 *course_id*、*title*、*dept_name* 和 *credits*。

每个实体的每个属性都有一个值（value）。例如，一个特定的 *instructor* 实体可能 *ID* 的值为 12121，*name* 的值为吴，*dept_name* 的值为金融，*salary* 的值为 90 000。

ID 属性用来唯一地标识教师，因为可能会有多个教师拥有相同的名字。在美国，许多企业发现将一个人的社会保障号[⊖]用作其值唯一标识该人的属性很方便。一般来说，大学必须给每个教师创建和分配一个唯一的标识符。

因此，数据库包括一组实体集，每个实体集包括任意数量的相同类型的实体。图 7-1 为一个大学数据库的一部分，其中有两个实体集：*instructor* 和 *student*。为了使图示简单，只显示了两个实体集的一部分属性。

一个大学数据库可能包含许多其他的实体集。例如，除了跟踪记录教师和学生外，大学还具有课
[263] 程信息，用实体集 *course* 来表示，它包括属性 *account_number*、*course_id*、*title*、*dept_name* 和 *credits*。在真实环境中，一个大学数据库可能会包含数十个实体集。

7.2.2 联系集

联系（relationship）是指多个实体间的相互关联。例如，我们可以定义关联教师 Katz 和学生 Shankar 的联系 *advisor*。这一联系指明 Katz 是学生 Shankar 的导师。

⊖ 在美国，政府分配给国家中每一个人一个单一的号码，称为社会保障号，用来唯一地标识一个人。任何一个人都仅有一个这样的社会保障号，并且没有两个人会拥有同样的社会保障号。

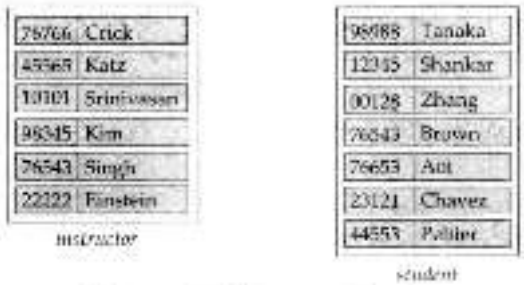


图7-1 实体集 *instructor* 和 *student*

联系集(relationship set)是相同类型联系的集合。正规地说，联系集是 $n \geqslant 2$ 个(可能相同的)实体集上的数学关系。如果 E_1，E_2，…，E_n 为实体集，那么联系集 R 是

$$\{(e_1, e_2, \cdots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \cdots, e_n \in E_n\}$$

的一个子集，而 $(e_1, e_2, \cdots, e_n)$ 是一个联系。

考虑图7-1中的两个实体集 *instructor* 和 *student*。我们定义联系集 *advisor* 来表示教师及学生之间的关联。这一关联如图7-2所示。

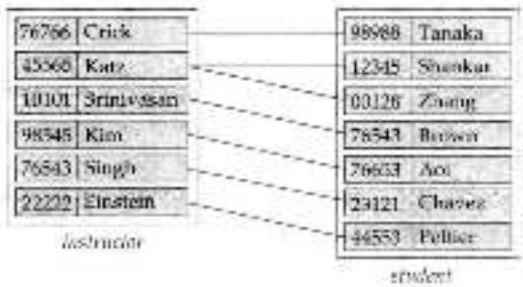


图7-2 联系集 *advisor*

再看另一个例子，考虑实体集 *student* 和 *section*。我们可以定义联系集 *takes* 来表示学生和该学生所注册的开课之间的关联。

实体集之间的关联称为**参与**；也就是说，实体集 E_1，E_2，…，E_n **参与**(participate)联系集 R。E-R模式中的一个**联系实例**(relationship instance)表示在所建模的现实世界企业中命名实体间的一个关联。例如，一个教师 *ID* 为45565的 *instructor* 实体Katz和一个学生 *ID* 为12345的 *student* 实体Shankar参与到 *advisor* 的一个联系实例中。这一联系实例表示在大学中教师Katz指导学生Shankar。

实体在联系中扮演的功能称为实体的**角色**(role)。由于参与一个联系集的实体集通常是互异的，因此角色是隐含的并且一般并不指定。但是，当联系的含义需要解释时角色是很有用的。当参与联系集的实体集并非互异的时候就是这种情况；也就是说，同样的实体集以不同的角色参与一个联系集多于一次。在这类联系集中，即有时称作**自环的**(recursive)联系集中，有必要用显式的角色名来指明实体是如何参与联系实例的。例如，考虑记录大学开设的所有课程的信息的实体集 *course*。我们用 *course* 实体的有序对来建模联系集 *prereq*，以描述一门课程($C2$)是另一门课程($C1$)的先修课。每对课程中的第一门课程具有课程 $C1$ 的角色，而第二门课程具有先修课 $C2$ 的角色。按照这种方式，所有的 *prereq* 联系通过($C1$，$C2$)对来表示，排除了($C2$，$C1$)对。

联系也可以具有**描述性属性**(descriptive attribute)。考虑实体集 *instructor* 和 *student* 之间的联系集 *advisor*。我们可以将属性 *date* 与该联系关联起来，以表示教师成为学生的导师的日期。教师Katz对应的实体和学生Shankar对应的实体之间的联系 *advisor* 的属性 *date* 的值为"10 June 2007"，表示Katz于2007年6月10日成为Shankar的导师。

图7-3所示为具有描述性属性 *date* 的联系集 *advisor*。注意，Katz在两个不同的日期成为了两名学生的导师。

作为联系的描述性属性的一个更实际的例子，考虑实体集 *student* 和 *section* 参与一个联系集 *takes*。我们也许希望在这个联系中用一个描述性属性 *grade*，来记录学生在这门课中取得的成绩。我们同样可以用一个描述性的属性 *for_credit* 来记录学生在这门课中是选修还是旁听（或出席）的情况。

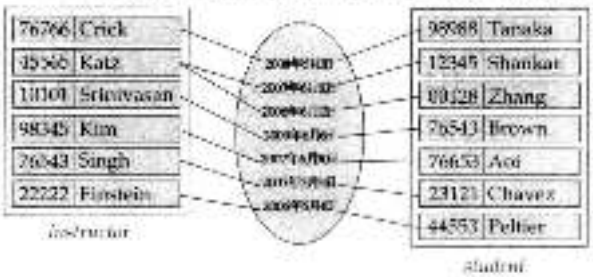


图 7-3 *date* 作为联系集 *advisor* 的属性

给定的联系集中的一个联系实例必须是由其参与实体唯一标识的，而不必使用描述属性。为了理
265 解这一点，假设我们要对一个教师成为一个特定学生的导师的所有日期建模。单值的属性 *date* 只能保存一个日期。我们不能通过同一个教师和学生之间的多个联系实例来表示多个日期，因为这些联系实例仅使用参与的实体是无法唯一标识的。正确的处理方法是创建一个多值属性 *date*，它可以保存所有的日期。

相同的实体集可能会参与到多于一个联系集中。在我们的例子中，*instructor* 和 *student* 实体集参与到联系集 *advisor* 中。另外，假设每个学生必须有一名教师作为他的系导师（本科或研究生），那么 *instructor* 和 *student* 实体集将参与到另一个联系集 *dept_advisor* 中。

联系集 *advisor* 和 *dept_advisor* 给出了二元（binary）联系集的例子，即涉及两个实体集的联系集。数据库系统中的大部分联系集都是二元的。然而，有时联系集会涉及多于两个实体集。

例如，假设我们有一个代表在大学内开展的所有研究项目的实体集 *project*，考虑实体集 *instructor*，*student* 和 *project*。每个项目可以有多个参与的学生和多个参与的教师。另外，每个参与项目的学生必须有一个教师指导他在项目中的工作。目前，我们忽略项目和教师以及项目和学生这两个关联，而关注哪个教师在一个特定项目上指导哪个学生。为了表达这个信息，我们通过关系 *proj_guide* 将三个实体集联系到一起，它表示某个学生在某个项目上接收了某个教师的指导。

注意，一个学生可以在不同的项目中有不同的教师作为导师，不能将这个联系描述成学生与教师
266 之间的二元关系。

参与联系集的实体集的数目称为联系集的**度**（degree）。二元联系集的度为 2；三元联系集的度为 3。

7.2.3 属性

每个属性都有一个可取值的集合，称为该属性的**域**（domain），或者**值集**（value set）。*course_id* 属性的域可能是特定长度的所有文本字符串的集合。类似地，属性 *semester* 的域可能是集合{秋，冬，春，夏}中的字符串。

正规地说，实体集的属性是将实体集映射到域的函数。由于一个实体集可能有多个属性，因此每个实体可以用一组（属性，数据值）对来表示，实体集的每个属性对应一个这样的对。例如，某个 *instructor* 实体可以用集合{(*ID*，76766)，(*name*，Crick)，(*dept_name*，生物)，(*salary*，72000)}来描述，该实体描述了一个叫 Crick 的人，他的教师编号为 76766，是生物系的成员，工资为 $72 000。从这里我们可以看出抽象模式与被建模的实际企业的结合。用来描述实体的属性值构成存储在数据库中的数据的一个重要部分。

E-R 模型中的属性可以按照如下的属性类型来进行划分：

- **简单**（simple）和**复合**（composite）属性。在我们的例子中，迄今为止已观的属性都是简单属性，也就是说，它们不能划分为更小的部分。另一方面，复合（composite）属性可以再划分为更小的部分（即其他属性）。例如，属性 *name* 可设计为一个包含 *first_name*，*middle_initial* 和 *last_name*

的复合属性。如果一个用户希望在一些场景中引用完整的属性，而在另外的场景中仅引用属性的一部分，则在设计模式中使用复合属性是一个好的选择。假设我们要给 *student* 实体集增加一个地址。地址可定义为包含属性 *street*、*city*、*state* 和 *zip_code* [⊖]的复合属性 *address*。复合属性帮助我们把相关属性聚集起来，使模型更清晰。

注意，复合属性可以是有层次的。在复合属性 *address* 中，其子属性 *street* 可以进一步分为 *street_number*、*street_name* 和 *apartment_number*。图 7-4 描述了 *instructor* 实体集的这些复合属性的例子。



图 7-4　复合属性教师 *name* 和 *address*

- **单值**(single-valued)**和多值**(multivalued)**属性**。我们的例子中的属性对一个特定实体都只有单独的一个值。例如，对某个特定的学生实体而言，*student_ID* 属性只对应于一个学生 *ID*。这样的属性称作是单值(single valued)的。而在某些情况下对某个特定实体而言，一个属性可能对应于一组值。假设我们往 *instructor* 实体集添加一个 *phone_number* 属性，每个教师可以有零个、一个或多个电话号码，不同的教师可以有不同数量的电话。这样的属性称作是多值(multivalued)的。作为另一个例子，我们可以往实体集 *instructor* 中添加一个属性 *dependent_name*，它列出所有的眷属。这个属性将是多值的，因为任何一个特定的教师可能有零个、一个或多个眷属。 267

  为了表示一个属性是多值的，我们用花括号将属性名括住，例如：{*phone_number*} 或者 {*dependent_name*}。

  在适当的情况下，可以对一个多值属性的取值数目设置上、下界。例如，一所大学可能将单个教师的电话号码个数限制在两个以内。在这个例子中设置限制表明 *instructor* 实体集的 *phone_number* 属性可以有 0 - 2 个值。

- **派生**(derived)**属性**。这类属性的值可以从别的相关属性或实体派生出来。例如，让我们假设 *instructor* 实体集有一个属性 *students_advised*，表示一个教师指导了多少个学生。我们可以通过统计与一个教师相关联的所有 *student* 实体的数目来得到这个属性的值。

  又如，假设 *instructor* 实体集具有属性 *age*，表示教师的年龄。如果 *instructor* 实体集还具有属性 *date_of_birth*，我们就可以从当前的日期和 *date_of_birth* 计算出 *age*。因此 *age* 就是派生属性。在这里，*date_of_birth* 可以称为基属性，或存储的属性。派生属性的值不存储，而是在需要时计算出来。

当实体在某个属性上没有值时使用**空**(null)值。空值可以表示"不适用"，即该实体的这个属性不存在值。例如，一个人可能没有中间名字。空还可以用来表示属性值未知。未知的值可能是缺失的(值存在，但我们没有该信息)，或不知道的(我们并不知道该值是否确实存在)。

例如，如果一个特定的教师的 *name* 值是空，我们推测这个值是缺失的，因为每个教师肯定有一个名字。*apartment_number* 属性的空值可能意味着地址不包括房间号(不适用)，或房间号是存在的但是我们不知道是什么(缺失的)，或者我们不知道房间号是否是该教师的地址的一部分(不知道的)。 268

7.3　约束

E-R 企业模式可以定义一些数据库中的数据必须要满足的约束。这一节讨论映射基数以及参与约束。

⊖　我们假定使用美国的地址格式，其中包括一个称作 *zip code* 的数字邮政编码。

7.3.1 映射基数

映射基数(mapping cardinality)，或基数比率，表示一个实体通过一个联系集能关联的实体的个数。

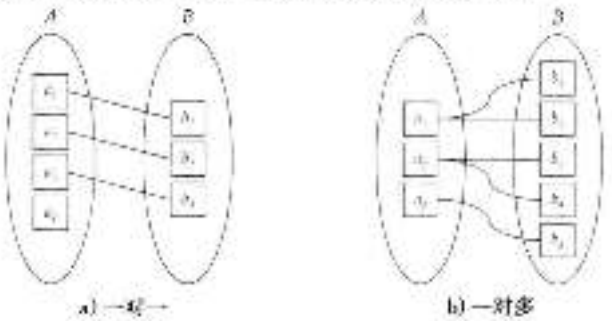
映射基数在描述二元联系集时非常有用，尽管它们可以用于描述涉及多于两个实体集的联系集。在这一节中，我们将只集中在二元联系集上。

对于实体集 A 和 B 之间的二元联系集 R 来说，映射基数必然是以下情况之一：

- **一对一**(one-to-one)。A 中的一个实体至多与 B 中的一个实体相关联，并且 B 中的一个实体也至多与 A 中的一个实体相关联(如图 7-5a 所示)。
- **一对多**(one-to-many)。A 中的一个实体可以与 B 中的任意数目(零个或多个)实体相关联，而 B 中的一个实体至多与 A 中的一个实体相关联(如图 7-5b 所示)。
- **多对一**(many-to-one)。A 中的一个实体至多与 B 中的一个实体相关联，而 B 中的一个实体可以与 A 中任意数目(零个或多个)实体相关联(如图 7-6a 所示)。
- **多对多**(many-to-many)。A 中的一个实体可以与 B 中任意数目(零个或多个)实体相关联，而且 B 中的一个实体也可以与 A 中任意数目(零个或多个)实体相关联(如图 7-6b 所示)。

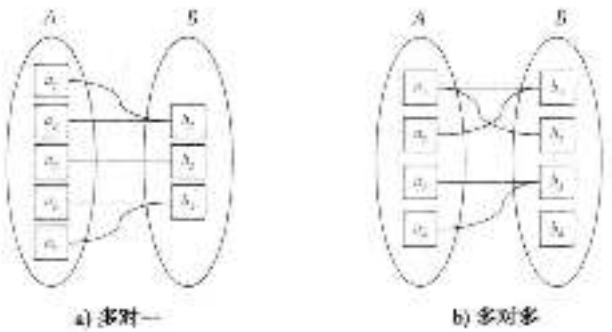
显然，一个特定联系集的适当的映射基数依赖于该联系集所建模的现实世界的情况。

作为例子，考虑 *advisor* 联系集。如果在一所特定的大学中，一名学生只能由一名教师指导，而一名教师可以指导多个学生，那么 *instructor* 到 *student* 的联系集是一对多的。如果一名学生可以由多名教师指导(比如学生可以由多名教师共同指导)，那么此联系集是多对多的。



a) 一对一　　b) 一对多

图 7-5　映射基数



a) 多对一　　b) 多对多

图 7-6　映射基数

7.3.2 参与约束

如果实体集 E 中的每个实体都参与到联系集 R 的至少一个联系中，实体集 E 在联系集 R 中的参与称为**全部**(total)的。如果 E 中只有部分实体参与到 R 的联系中，实体集 E 到联系集 R 的参与称为**部分**(partial)的。在图 7-5a 中，B 在联系集中的参与是全部的，而 A 在联系集中的参与是部分的；在图 7-5b 中，A 和 B 在联系集中的参与都是全部的。

例如，我们期望每个 *student* 实体通过 *advisor* 联系同至少一名教师相联系，因而 *student* 在联系集 *advisor* 中的参与是全部的。相反地，一个 *instructor* 不是必须要指导一个学生。因此，很可能只有一部分 *instructor* 实体通过 *advisor* 联系同 *student* 实体集相关联，于是 *instructor* 在 *advisor* 联系集中的参与是部分的。 269 | 270

7.3.3 码

我们必须有一个区分给定实体集中的实体的方法。从概念上来说，各个实体是互异的；但从数据库的观点来看，它们的区别必须通过其属性来表明。

因此，一个实体的属性的值必须可以唯一地标识该实体。也就是说，在一个实体集中不允许两个实体对于所有属性都具有完全相同的值。

2.3 节定义的关系模式的码的概念直接适用于实体集。即实体的码是一个足以区分每个实体的属性集。关系模式中的超码、候选码、主码的概念同样适用于实体集。

码同样用于唯一地标识联系，并从而将联系互相区分开来。我们在下面定义联系的码的相应概念。

实体集的主码使得我们可以区分实体集中不同的实体。我们需要一种类似的机制来区分联系集中不同的联系。

设 R 是一个涉及实体集 E_1，E_2，…，E_n 的联系集。设主码(E_i)代表构成实体集 E_i 的主码的属性集合。目前我们假设所有主码的属性名是互不相同的。联系集主码的构成依赖于同联系集 R 相关联的属性集合。

如果联系集 R 没有属性与之相关联，那么属性集合

$$primary\text{-}key(E_1) \cup primary\text{-}key(E_2) \cup \cdots \cup primary\text{-}key(E_n)$$

描述了集合 R 中的一个联系。

如果联系集 R 有属性 a_1，a_2，…，a_m 与之相关联，那么属性集合

$$primary\text{-}key(E_1) \cup primary\text{-}key(E_2) \cup \cdots \cup primary\text{-}key(E_n) \cup \{a_1, a_2, \cdots, a_m\}$$

描述了集合 R 中的一个联系。

在以上两种情况下，属性集合

$$primary\text{-}key(E_1) \cup primary\text{-}key(E_2) \cup \cdots \cup primary\text{-}key(E_n)$$

构成了联系集的一个超码。

如果实体集间主码的属性名称不是互不相同的，重命名这些属性以区分它们；实体集的名字加上属性名可以构成唯一的名称。如果一个实体集不止一次参与某个联系集（如 7.2.2 节中的 *prereq* 联系），则使用角色名代替实体集名构成唯一的属性名。 271

联系集的主码结构依赖于联系集的映射基数。例如，考虑在 7.2.2 节中的实体集 *instructor* 和 *student* 以及具有属性 *date* 的联系集 *advisor*。假设联系集是多对多的，那么 *advisor* 的主码由 *instructor* 和 *student* 的主码的并集组成。如果联系是从 *student* 到 *instructor* 多对一的，即每个学生最多只能有一个导师，则 *student* 的主码就是 *advisor* 的主码。而如果一名教师只能指导一名学生，即联系是从 *instructor* 到 *student* 多对一的，则 *instructor* 的主码就是 *advisor* 的主码。对于一对一的联系，两个候选码中的任意一个可以用作主码。

对于非二元联系，如果没有基数的限制，那么在本节开始时描述的超码就是唯一的候选码，并被选为主码。如果有基数的限制，主码的选择就复杂多了。因为我们还没有讨论如何在非二元关系中描述基数约束，所以我们在本章中不对此做更多的讨论。我们将在 7.5.5 节和 8.4 节详细地考虑这个问题。

7.4 从实体集中删除冗余属性

当我们使用 E-R 模型设计数据库时，我们通常从确定那些应当包含的实体集开始。例如，在我们迄今所讨论的大学机构中，我们想要包含如 *student* 和 *instructor* 等实体集。当决定好实体集后，我们必须挑选适当的属性，这些属性要表示我们在数据库中所捕获的不同的值。在大学机构中，我们为 *instructor* 实体集设计了包括 *ID*、*name*、*dept_name* 以及 *salary* 几个属性，我们还可以增加 *phone_number*、*office_number*、*home_page* 等属性。要包含哪些属性的选择决定于了解企业结构的设计者。

一旦选择好实体和它们相应的属性，不同实体间的联系集就建立起来了。这些联系集有可能会导

数不同实体集中的属性冗余，并需要将其从原始实体集中删除。为了说明这一点，考虑实体集 *instructor* 和 *department*：

- 实体集 *instructor* 包含属性 *ID*、*name*、*dept_name* 以及 *salary*，其中 *ID* 构成主码。
- 实体集 *department* 包含属性 *dept_name*、*building* 以及 *budget*，其中 *dept_name* 构成主码。

272 我们用关联 *instructor* 和 *department* 的联系集 *inst_dept* 对每个教师都有一个关联的系的情况建模。

属性 *dept_name* 在两个实体集中都出现了。由于它是实体集 *department* 的主码，因此它在实体集 *instructor* 中是冗余的，需要将其移除。

从实体集 *instructor* 中移除属性 *dept_name* 可能不是那么直观，因为我们在前几章所用到的关系 *instructor* 中具有 *dept_name* 属性。我们将在后面看到，当我们从 E-R 图构造一个关系模式时，只有当每个教师最多只与一个系关联时，属性 *dept_name* 才会添加到关系 *instructor* 中。如果一个教师有多个关联的系时，教师与系之间的联系会记录在一个单独的关系 *inst_dept* 中。

将教师和系之间的关联统一看成联系，而不是 *instructor* 的一个属性，使得表集关系明确，并有助于避免过早地假设每个教师只与一个系关联。

类似地，实体集 *student* 通过联系集 *student_dept* 与实体集 *department* 关联，因而 *student* 中不需要 *dept_name* 属性。

作为另一个例子，考虑开课(section)和开课的时段。每个时段都由 *time_slot_id* 标识，并且和上课时间的集合相关联，每次上课时间都由星期几、开始时间以及结束时间标识。我们打算使用多值复合属性对上课时间集合建模。假设我们对实体集 *section* 和 *time_slot* 按以下方式建模：

- 实体集 *section* 包含属性 *course_id*、*sec_id*、*semester*、*year*、*building*、*room_number* 以及 *time_slot_id*，其中(*course_id*、*sec_id*、*year*、*semester*)构成主码。
- 实体集 time_slot 包含主码[⊖]属性 *time_slot_id*，以及一个多值复合属性{(*day*、*start_time*、*end_time*)}[⊖]。

这些实体通过联系集 *sec_time_slot* 相互关联。

属性 *time_slot_id* 在两个实体集中均出现。由于它是实体集 *time_slot* 的主码，因此它在实体集 *section* 中是冗余的，并且需要将其删除。

作为最后的例子，假设我们有一个实体集 *classroom*，包含属性 *building*、*room_number* 以及 *capacity*，主码由 *building* 和 *room_number* 组成。再假设我们有一个联系集 *sec_class*，将 *section* 和 *classroom* 关联在一起。那么属性 *building*、*room_number* 在实体集 *section* 中是冗余的。

一个好的实体-联系设计不包含冗余的属性。对于我们的大学的例子，我们在下面列出实体集以

273 及它们的属性，主码以下划线标明。

- **classroom**：包含属性(*building*、*room_number*、*capacity*)。
- **department**：包含属性(*dept_name*、*building*、*budget*)。
- **course**：包含属性(*course_id*、*title*、*credits*)。
- **instructor**：包含属性(*ID*、*name*、*salary*)。
- **section**：包含属性(*course_id*、*sec_id*、*semester*、*year*)。
- **student**：包含属性(*ID*、*name*、*tot_cred*)。
- **time_slot**：包含属性(*time_slot_id*、{(*day*、*start_time*、*end_time*)})。

我们设计的联系集如下：

- **inst_dept**：关联教师和系。
- **stud_dept**：关联学生和系。
- **teaches**：关联教师和开课。

⊖ 我们将在后面看到主实体集 *time_slot* 构建的关系的主码包含 *day* 以及 *start_time*，然而，*day* 和 *start_time* 并不是实体集 *time_slot* 的主码一部分。

⊖ 我们可以给包含 *day*、*start_time* 以及 *end_time* 的复合属性选择一个名字，例如 *meeting*。

- **takes**：关联学生和开课，包含描述性属性 *grade*。
- **course_dept**：关联课程和系。
- **sec_course**：关联开课和课程。
- **sec_class**：关联开课和教室。
- **sec_time_slot**：关联开课和时段。
- **advisor**：关联学生和教师。
- **prereq**：关联课程和先修课程。

你可以验证没有任何一个实体集包含由联系集而造成冗余的属性；另外，你可以验证我们此前在第2章的图2-8中看到的大学数据库关系模式中的所有的信息（除了约束）全部包含在上述的设计中，只是关系设计中的几个属性被E-R设计中的联系所替代。

7.5 实体－联系图

1.3.3节曾经简要地介绍过，E-R图（E-R diagram）可以图形化表示数据库的全局逻辑结构。E-R图既简单又清晰，这些是致使E-R模型广泛使用的重要性质。

7.5.1 基本结构

E-R图包括如下几个主要构件： 274

- **分成两部分的矩形代表实体集**。本书中有阴影的第一部分包含实体集的名字，第二部分包含实体集中所有属性的名字。
- **菱形代表联系集**。
- **未分割的矩形代表联系集的属性**。构成主码的属性以下划线标明。
- **线段将实体集连接到联系集**。
- **虚线将联系集属性连接到联系集**。
- **双线显示实体在联系集中的参与度**。
- **双菱形代表连接到弱实体集的标志性联系集**（我们将在7.5.6节讲述标志性联系集和弱实体集）。

考虑图7-7中的E-R图，它由通过二元联系集 *advisor* 关联的两个实体集 *instructor* 和 *student* 组成。同 *instructor* 相关联的属性为 *ID*、*name* 和 *salary*。同 *student* 相关联的属性为 *ID*、*name* 和 *tot_cred*。如图7-7所示，实体集的属性中那些组成主码的属性以下划线标明。

如果一个联系集有关联的属性，那么我们将这些属性放入一个矩形中，并且用虚线将该矩形与代表联系集的菱形连接起来。例如，在图7-8中，我们有描述性属性 *date* 附带到联系集 *advisor* 上，表示教师成为导师的日期。



图7-7 对应于教师和学生的E-R图

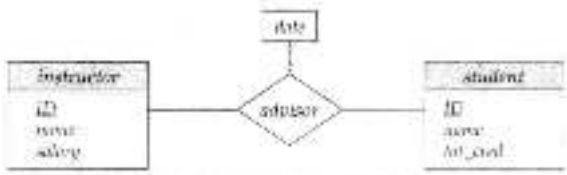


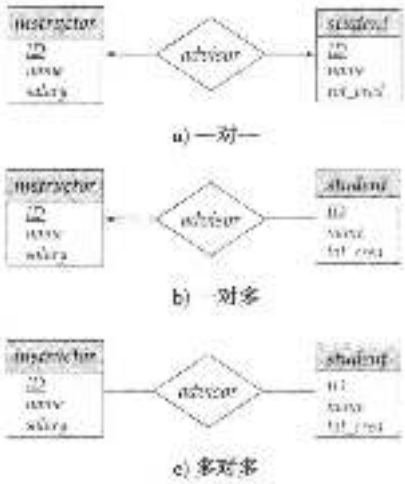
图7-8 联系集上附带了一个属性的E-R图 275

7.5.2 映射基数

实体集 *instructor* 和 *student* 之间的联系集 *advisor* 可以是一对一、一对多、多对一或多对多的。为了区别这些类型，我们在所讨论的联系集和实体集之间画一个箭头(→)或一条线段(—)，如下所示：

- 一对一：我们从联系集 *advisor* 向实体集 *instructor* 和 *student* 各画一个箭头(见图 7-9a)。这表示一名教师可以指导至多一名学生，并且一名学生可以有至多一位导师。
- 一对多：我们从联系集 *advisor* 画一个箭头到实体集 *instructor*，以及一条线段到实体集 *student*(见图 7-9b)。这表示一名教师可以指导多名学生，但一名学生可以有至多一位导师。
- 多对一：我们从联系集 *advisor* 画一条线段到实体集 *instructor*，以及一个箭头到实体集 *student*。这表示一名教师可以指导至多一名学生，但一名学生可以有多位导师。
- 多对多：我们从联系集 *advisor* 向实体集 *instructor* 和 *student* 各画一条线段(见图 7-9c)。这表示一名教师可以指导多名学生，并且一名学生可以有多位导师。

276



a) 一对一

b) 一对多

c) 多对多

图 7-9 联系

E-R 图还提供了一种描述每个实体参与联系集中的联系的次数的更复杂的约束的方法。实体集和二元联系集之间的一条边可以有一个关联的最大和最小的映射基数，用 $l..h$ 的形式表示，其中 l 表示最小的映射基数，而 h 表示最大的映射基数。最小值为 1 表示这个实体集在该联系集中全部参与，即实体集中的每个实体在联系集中的至少一个联系中出现。最大值为 1 表示这个实体参与至多一个联系，而最大值为 * 代表没有限制。

例如，考虑图 7-10，在 *advisor* 和 *student* 之间的边有 1..1 的基数约束，意味着基数的最小值和最大值都是 1。也就是，每个学生必须有且仅有一个导师。从 *advisor* 到 *instructor* 边上的约束 0..* 说明教师可以有零个或多个学生。因此，*advisor* 联系是从 *instructor* 到 *student* 的一对多联系，更进一步地讲，*student* 在 *advisor* 联系中的参与是全部的，表示一个学生必须有一个导师。



图 7-10 联系集上的基数约束

很容易将左侧边上的 0..* 曲解为联系 *advisor* 是从 *instructor* 到 *student* 多对一的，而这正好和下面的解释相反。

如果两条边都有最大值 1，那么这个联系是一对一的。如果我们在左侧边上标明基数约束 1..*，我们就可以说每名教师必须指导至少一名学生。

图 7-10 中的 E-R 图的另一种画法是在基数约束的位置画一条从 *student* 到 *advisor* 的双线，以及一个从 *advisor* 到 *instructor* 的箭头。这种画法可以强制实施同图 7-10 中所示约束完全一样的约束。

7.5.3 复杂的属性

图 7-11 说明了怎样在 E-R 图中表示复合属性。这里一个具有子属性 *first_name*、*middle_initial* 和 *last_name* 的复合属性 *name* 代替了 *instructor* 的简单属性 *name*。再例如，假定我们给实体集 *instructor* 增加一个地址。地址可以定义为具有属性 *street*、*city*、*state* 和 *zip_code* 的

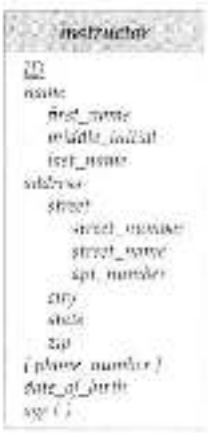


图 7-11 包含复合、多值和派生属性的 E-R 图

复合属性 *address*。属性 *street* 本身也是一个复合属性，其子属性为 *street_number*、*street_name* 和 *apartment_number*。

在图 7-11 还给出了一个由"|*phone_number*|"表示的多值属性 *phone_number* 和一个由"{*age*()}"表示的派生属性 *age*。

7.5.4 角色

在 E-R 图中，我们通过在菱形和矩形之间的连线上进行标注来表示角色。图 7-12 给出了 *course* 实体集和 *prereq* 联系集之间的角色标识 *course_id* 和 *prereq_id*。

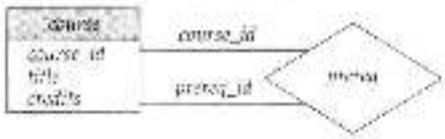


图 7-12 包含角色标识的 E-R 图

7.5.5 非二元的联系集

非二元的联系集也可以在 E-R 图中简单地表示。图 7-13 包含三个实体集 *instructor*、*student* 和 *project*，它们通过联系集 *proj_guide* 相关联。

277
278

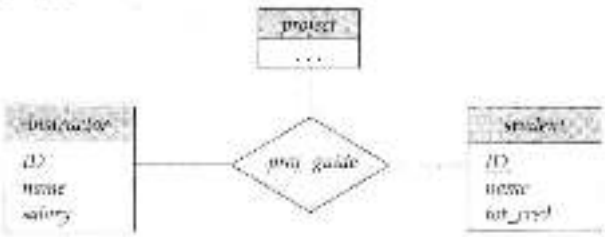


图 7-13 包含三元联系的 E-R 图

在非二元的联系集中，我们可以表示某些类型的多对一联系。假设一个 *student* 在每个项目上最多只能有一位导师。这种约束可用从 *proj_guide* 的边指向 *instructor* 的箭头来表示。

在一个联系集外我们至多允许一个箭头，因为在一个非二元的联系集外包含两个或更多箭头的 E-R 图可以用两种方法解释。假设实体集 A_1，A_2，…，A_n 之间有联系集 R，并且只有指向实体集 A_{i+1}，A_{i+2}，…，A_n 的边是箭头。那么，两种可能的解释为：

- 来自 A_1，A_2，…，A_i 的实体的一个特定组合可以和至多一个来自 A_{i+1}，A_{i+2}，…，A_n 的实体组合相关联。因而联系 R 的主码可以用 A_1，A_2，…，A_i 的主码的并集来构造。
- 对每个实体集 A_k，$i<k\leq n$，来自其他实体集的实体的每个组合可以和来自 A_k 的至多一个实体相关联。于是对于 $i<k\leq n$，每个集合 $\{A_1, A_2, \cdots, A_{k-1}, A_{k+1}, \cdots, A_n\}$ 构成一个候选码。

这两种解释在不同的书和系统中使用。为了避免混淆，我们只允许在一个联系集外有一个箭头，在这种情况下这两种解释是等价的。在第 8 章中(8.4 节)我们学习函数依赖，它允许以一种不会混淆的方式描述这两种解释。

7.5.6 弱实体集

考虑一个 *section* 实体，它由课程编号、学期、学年以及开课编号唯一标识。显然，开课实体和课程实体相关联。假定我们在实体集 *section* 和 *course* 之间创建了一个联系集 *sec_course*。

现在，发现 *sec_course* 中的信息是冗余的，由于 *section* 已有属性 *course_id*，它标识该开课所关联的课程。消除这种冗余的一个方法是删除联系 *sec_course*；然而，这么做使得 *section* 和 *course* 之间的联系 279
隐含于一个属性中，这并不是我们想要的。

消除这种冗余的另一个方法是在实体 *section* 中不保存属性 *course_id*，而只保存剩下的属性 *sec_id*、

year 以及 *semester*[⊖]。然而，这样的话实体集 *section* 就没有足够的属性唯一标识一个指定的 *section* 实体；即使每个 *section* 实体都是唯一的，不同课程的开课也可能会有相同的 *sec_id*、*year* 以及 *semester*。为解决这个问题，我们将联系 *sec_course* 视为一个特殊的联系，它给唯一标识 *section* 实体提供额外信息，即 *course_id*。

弱实体集的概念对上述想法进行了正式的规定。没有足够的属性以形成主码的实体集称作**弱实体集**(weak entity set)。有主码的实体集称作**强实体集**(strong entity set)。

弱实体集必须与另一个称作标识(identifying)或**属主实体集**(owner entity set)的实体集关联才能有意义。每个弱实体必须和一个标识实体关联；也就是说，弱实体集**存在依赖**(existence dependent)于标识实体集。我们称标识实体集**拥有**(own)它所标识的弱实体集。将弱实体集与其标识实体集相联的联系称为**标识性联系**(identifying relationship)。

标识性联系是从弱实体集到标识实体集多对一的，并且弱实体集在联系中的参与是全部的。标识性联系集不应该有任何描述性属性，因为这种属性中的任意一个都可以与弱实体集相关联。

在我们的例子中，*section* 的标识实体集是 *course*，将 *section* 实体和它们对应的 *course* 实体关联在一起的 *sec_course* 是标识性联系。

虽然弱实体集没有主码，但是我们仍需要区分依赖于特定强实体集的弱实体集中的实体的方法。弱实体集的**分辨符**(discriminator)是使得我们进行这种区分的属性集合。例如，弱实体集 *section* 的分辨符由属性 *sec_id*、*year* 以及 *semester* 组成，因为对每门课程来说，这个属性集唯一标识了这门课程的一次开课。弱实体集的分辨符也称为该实体集的**部分码**。

弱实体集的主码由标识实体集的主码加上该弱实体集的分辨符构成。在实体集 *section* 的例子中，它的主码是 {*course_id*, *sec_id*, *year*, *semester*}，其中 *course_id* 是标识实体集 *course* 的主码，{*sec_id*, *year*, *semester*} 区分同一门课程的不同 *section* 实体。

注意，我们可以选择使 *sec_id* 对于大学所提供的所有课程都不重复。在这种情况下实体集 *section* [280] 将会具有一个主码。然而，一个 *section* 的存在在概念上仍依赖于一个 *course*，通过使之成为弱实体集可以明确这种依赖关系。

在 E-R 图中，弱实体集和强实体集类似，以矩形表示，但是有两点主要的区别：

- 弱实体集的分辨符以虚下划线标明，而不是实线。
- 关联弱实体集和标识性强实体集的联系集以双菱形表示。

在图 7-14 中，弱实体集 *section* 通过联系集 *sec_course* 依赖于强实体集 *course*。



图 7-14 包含弱实体集的 E-R 图

该图还表明了如何使用双线表明全部参与：(弱)实体集 *section* 在联系 *sec_course* 中的参与是全部的，表示每次开课都必须通过 *sec_course* 同某门课程关联。最后，从 *sec_course* 指向 *course* 的箭头表示每次开课与单门课程相关联。

弱实体集可以参与标识性联系以外的其他联系。例如，*section* 实体可以和 *time_slot* 实体集参与一个联系，以标识开课的时间。弱实体集可以作为属主与另一个弱实体集参与一个标识性联系。一个弱实体集也可能与不止一个标识实体集关联。这样，一个特定的弱实体将被一个实体的组合标识，其中每个标识实体集有一个实体在该组合中。弱实体集的主码可以由标识实体集的主码的并集加上弱实体集的分辨符组成。

在某些情况下，数据库设计者会选择将一个弱实体集表示为属主实体集的一个多值复合属性。在

⊖ 注意，即使我们从实体集 *section* 中去掉了 *course_id* 属性，但因为后面会看清楚的一些原因，我们最终从实体集 *section* 构建的关系模式还是具有 *course_id* 属性的。

我们的例子中，这种方法需要实体集 *course* 具有一个多值复合属性 *section*。如果弱实体集只参与标识性联系，而且其属性不多，那么在建模时将其表示为一个属性更恰当。相反地，如果弱实体集参与到标识性联系以外的联系中，或者其属性较多，则建模时将其表示为弱实体集更恰当。很明显，*section* 不 281
符合建模成多值复合属性的要求，而将其建模为弱实体集更恰当。

7.5.7 大学的E-R图

图7-15展示了本书迄今所使用的大学所对应的E-R图。除了增加了若干约束，以及 *section* 为弱实体以外，该E-R图与我们在7.4节中看到的大学E-R模型的文字性描述等价。

在我们的大学数据库中，我们限制每名教师必须有且仅有一个相关联的系。因此，如图7-15所示，在 *instructor* 和 *inst_dept* 之间有一条双线，表示 *instructor* 在 *inst_dept* 中全部参与；即每名教师必须和一个系相关联。另外，存在一个从 *inst_dept* 到 *department* 的箭头，表示每个教师可以有至多一个相关联的系。

类似地，实体集 *course* 和 *student* 分别与联系集 *course_dept* 和 *stud_dept* 用双线连接，实体集 *section* 和联系集 *sec_time_slot* 亦如此。前面的两个联系分别通过箭头指向另一个联系 *department*，而第三个联系通过箭头指向 *time_slot*。

此外，图7-15显示联系集 *takes* 具有一个描述性属性 *grade*，以及每个学生有至多一位导师。该图还表明目前 *section* 是一个弱实体集，由属性 *sec_id*、*semester* 以及 *year* 组成分辨符；*sec_course* 是连接弱实体集 *section* 和强实体集 *course* 的标识性联系集。

7.6节将介绍E-R图能够如何用于转换为我们使用的各个关系模式。

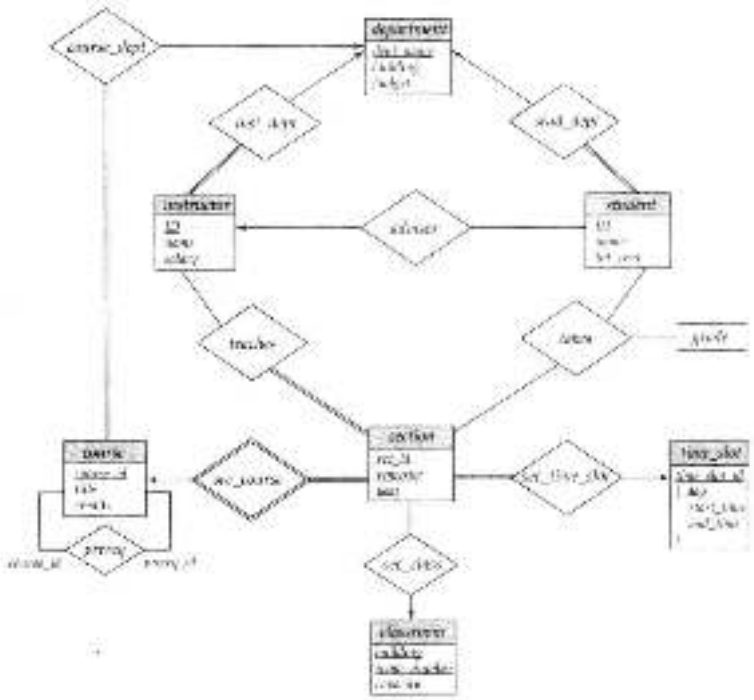


图7-15 大学的E-R图

7.6 转换为关系模式

我们可以将一个符合E-R数据库模式的数据库表示为一些关系模式的集合。在数据库设计中，对于每个实体集以及对于每个联系集，都有唯一的关系模式与之对应，关系模式名即为相应的实体集或联系集的名称。

E-R模型和关系数据库模型都是现实世界企业抽象的逻辑表示。由于两种模型采用类似的设计原

则，因此我们可以将 E-R 设计转换为关系设计。

这一节描述如何用关系模式来表示 E-R 模式，以及如何将 E-R 设计中提出的约束映射到关系模式上的约束。

7.6.1 具有简单属性的强实体集的表示

设 E 是只具有简单描述性属性 a_1，a_2，…，a_n 的强实体集。我们用具有 n 个不同属性的模式 E 来表示这个实体集。该模式的关系中的每个元组同实体集 E 的一个实体相对应。

对于从强实体集转换而来的模式，强实体集的主码就是生成的模式的主码。这个结论是从每个元组都对应于实体集中的一个特定实体这个事实直接得到的。

例如，考虑图 7-15 中 E-R 图的实体集 *student*。该实体集有三个属性：*ID*、*name* 和 *tot_cred*。我们用名为 *student* 的模式来表示这个实体集，它有三个属性：

$$student(\underline{ID},\ name,\ tot\_cred)$$

注意，由于学生 *ID* 是实体集的主码，因此它也是该关系模式的主码。

282
283

继续我们的例子，对于图 7-15 中的 E-R 图，除 *time_slot* 以外的所有的强实体集只有简单属性，从这些强实体集转换而来的模式为：

$$\begin{aligned}
&classroom(\underline{building},\ room_number,\ capacity)\\
&department(\underline{dept_name},\ building,\ budget)\\
&course(\underline{course_id},\ title,\ credits)\\
&instructor(\underline{ID},\ name,\ salary)\\
&student(\underline{ID},\ name,\ tot_cred)
\end{aligned}$$

如你所见，模式 *instructor* 和 *student* 与前面章节中用到的模式不同（它们不包含属性 *dept_name*）。后面很快还会讨论这个问题。

7.6.2 具有复杂属性的强实体集的表示

当一个强实体集具有非简单属性时，事情也加复杂一点。我们通过为每个子属性创建一个单独的属性来处理复合属性；我们并不为复合属性自身创建一个单独的属性。例如，考虑图 7-11 中表示的 *instructor* 实体集。对于复合属性 *name*，为 *instructor* 生成的模式包括属性 *first_name*、*middle_initial* 和 *last_name*；没有单独的属性或模式表示 *name*。类似地，对于复合属性 *address*，产生的模式包括属性 *street*、*city*、*state* 以及 *zip_code*。由于 *street* 是一个复合属性，因此它被替换成 *street_number*、*street_name* 以及 *apt_number*。8.2 节将再次讨论这个问题。

多值属性的处理不同于其他属性。我们已经看到，E-R 图中的属性通常都可以直接映射到相应关系模式的属性上。但是，多值属性是个例外；如我们将看到的，为了这些属性，创建新的关系模式。

派生的属性并不在关系数据模型中显式地表示出来。然而，它们可以在例如对象－关系数据模型的其他数据模型中表示为“方法”，第 22 章将对此进行讲述。

从具有复杂属性的实体集 *instructor* 转换而来，且不包含多值属性的关系模式为：

$$\begin{aligned}
instructor(&\underline{ID},\ first_name,\ middle_initial,\ last_name,\\
&street_number,\ street_name,\ apt_number,\\
&city,\ state,\ zip_code,\ date_of_birth)
\end{aligned}$$

对于一个多值属性 M，构建关系模式 R，该模式包含一个对应于 M 的属性 A，以及对应于 M 所在的实体集或联系集的主码的属性。

例如，考虑图 7-11 中的 E-R 图，它描绘了包含多值属性 *phone_number* 的实体集 *instructor*，*instructor* 的主码是 *ID*。为这个多值属性构建一个关系模式：

284

$$instructor\_phone(\underline{ID},\ \underline{phone\_number})$$

教师的每个电话号码都表示为该模式上的关系中的唯一一个元组。因此，如果有一个 *ID* 为 22222 的教师，电话号码为 555－1234 和 555－4321，关系 *instructor_phone* 将有两条元组（22222，555－1234）和（22222，555－4321）。

创建关系模式的主码，它由模式中的所有属性组成。在上面的例子中，主码由关系 *instructor_phone*

的两个属性一起组成。

另外，在多值属性构建的关系模式上建立外码约束，由实体集的主码所生成的属性去参照实体集所生成的关系。在上面的例子中，*instructor_phone* 关系上的外码约束是属性 *ID* 参照 *instructor* 关系。

在一个实体集只有两个属性的情况下——一个主码 B 以及一个多值属性 M——该实体集的关系模式只包含一个属性，即主码属性 B。可以删掉这个关系，同时保留具有属性 B 和对应 M 的属性 A 的关系模式。

例如，考虑图 7-15 中描绘的实体集 *time_slot*，其中 *time_slot_id* 是实体集 *time_slot* 的主码，有一个多值属性，而且恰好是复合的。这个实体集可以就按照以下从多值复合属性生成的模式表示：

$$time\_slot(\underline{time\_slot\_id}, \underline{day}, \underline{start\_time}, end\_time)$$

虽然没有在 E-R 图中表示为约束，但是我们知道不存在一个班的两次课在一周中的同一天的同一时间开始却在不同时间结束。基于这个约束，*end_time* 已从模式 *time_slot* 的主码中去除了。

从实体集生成的关系将只有一个属性 *time_slot_id*。支持此关系的优化有助于简化生成的数据库模式，即使它有一个与外码相关的缺点。7.6.4 节将对此简要讨论。

7.6.3　弱实体集的表示

设 A 是具有属性 a_1，a_2，…，a_n 的弱实体集。设 B 是 A 所依赖的强实体集，设 B 的主码包括属性 b_1，b_2，…，b_m。我们用名为 A 的关系模式表示实体集 A，该模式的每个属性对应以下集合中的一个成员：

$$\{a_1, a_2, \cdots, a_n\} \cup \{b_1, b_2, \cdots, b_m\}$$

对于从弱实体集转换而来的模式，该模式的主码由其所依赖的强实体集的主码与弱实体集的分辨符组合而成。除了创建主码之外，还要在关系 A 上建立外码约束，该约束指明属性 b_1，b_2，…，b_m 参照关系 B 的主码。外码约束保证表示弱实体的每个元组都有一个表示相应强实体的元组与之对应。 285

以图 7-15 所示 E-R 图中的弱实体集 *section* 为例说明。该实体集有属性：*sec_id*、*semester* 和 *year*。*section* 实体集所依赖的实体集 *course* 的主码是 *course_id*。因此，用来表示 *section* 的模式具有下面的属性：

$$section(\underline{course\_id}, \underline{sec\_id}, \underline{semester}, \underline{year})$$

该主码由实体集 *course* 的主码和 *section* 的分辨符（即 *sec_id*、*semester* 以及 *year*）组成。我们还在模式 *section* 上建立了一个属性 *course_id* 参照 *course* 模式的主码的外码约束，以及完整性约束"级联删除"[6]。由于外码约束上的"级联删除"规范，如果一个 *course* 实体被删除，那么所有与它相关联的 *section* 实体也被删除。

7.6.4　联系集的表示

设 R 是联系集，设 a_1，a_2，…，a_n 表示所有参与 R 的实体集的主码的并集构成的属性集合。设 R 的描述性属性（如果有）为 b_1，b_2，…，b_m。我们用名为 R 的关系模式表示该联系集。下面集合中的每一项表示为模式的一个属性：

$$\{a_1, a_2, \cdots, a_n\} \cup \{b_1, b_2, \cdots, b_m\}$$

7.3.3 节介绍过如何为一个二元联系集选择主码。如在该节中我们所看到的，从所有相关实体集中取所有主码属性能够用来标识一个指定元组，但是对于一对一、多对一和一对多联系集，这会得到一个比我们所需要的主码大的属性集合。因而如下选择主码：

- 对于多对多的二元联系，参与实体集的主码属性的并集成为主码。
- 对于一对一的二元联系集，任何一个实体集的主码都可以选作主码。这个选择是任意的。
- 对于多对一或一对多的二元联系集，联系集中"多"的那一方的实体集的主码构成主码。 286

---

[6] 4.4.5 节介绍了 SQL 中外码约束的"级联删除"性质。

- 对于边上没有箭头的 n 元联系集，所有参与实体集的主码属性的并集成为主码。
- 对于边上有一个箭头的 n 元联系集，不在"箭头"侧的实体集的主码属性为模式的主码。回想一下，一个联系集外只允许一个箭头。

我们还在关系模式 R 上建立外码约束，如下：对于每个与联系集 R 相关的实体集 E_i，我们建立一个关系模式 R 上的外码约束，R 中来自 E_i 主码属性的那些属性参照表示关系模式 E_i 的主码。

以图 7-15 中 E-R 图的联系集 *advisor* 为例说明。此联系集涉及如下两个实体集：

- *instructor*，主码为 *ID*。
- *student*，主码为 *ID*。

由于该联系集没有属性，因此 *advisor* 模式有两个属性，*instructor* 和 *student* 的主码。由于这两个属性具有相同的名字，因此将它们重命名为 *i_ID* 和 *s_ID*。因为 *advisor* 联系集是从 *student* 到 *instructor* 多对一的，所以关系模式 *advisor* 的主码是 *s_ID*。

我们还在关系 *advisor* 上建立了两个外码约束，属性 *i_ID* 参照 *instructor* 的主码，属性 *s_ID* 参照 *student* 的主码。

继续我们的例子，对于图 7-15 的 E-R 图，从联系集派生的模式如图 7-16 所示。

观察到对于联系集 *prereq*，与联系相关联的角色标识用作属性的名字，这是因为两个角色都参照同一个关系 *course*。

与 *advisor* 的情况类似，*sec_course*、*sec_time_slot*、*secClass*、*inst_dept*、*stud_dept* 以及 *course_dept* 每个关系的主码仅由两个相关联的实体集中的一个实体集的主码构成，因为每个对应的联系都是多对一的。

图 7-16 中并没有表示出外码，但是对于该图中的每一个关系都有两个外码约束，参照相关的两个实体集所构建出的两个关系。例如，*sec_course* 有参照 *section* 和 *classroom* 的外码，*teaches* 有参照 *instructor* 和 *section* 的外码，以及 *takes* 有参照 *student* 和 *section* 的外码。

teaches (*ID*, *course_id*, *sec_id*, *semester*, *year*)
takes (*ID*, *course_id*, *sec_id*, *semester*, *year*, *grade*)
prereq (*course_id*, *prereq_id*)
advisor (*s_ID*, *i_ID*)
sec_course (*course_id*, *sec_id*, *semester*, *year*)
sec_time_slot (*course_id*, *sec_id*, *semester*, *year*, *time_slot_id*)
sec_class (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*)
inst_dept (*ID*, *dept_name*)
stud_dept (*ID*, *dept_name*)
course_dept (*course_id*, *dept_name*)

图 7-16 由图 7-15 中 E-R 图的联系集派生出的模式

这个优化允许对包含多值属性的实体集 *time_slot* 只建立一个关系模式，而避免从关系模式 *sec_time_slot* 到由实体集 *time_slot* 生成的关系的外码的建立。因为我们舍弃了由实体集 *time_slot* 生成的关系。我们保留了从多值属性生成的名为 *time_slot* 的关系，但这个关系可能没有元组与某个 *time_slot_id* 相关联，或者有多条元组与某个 *time_slot_id* 相关联，因而 *sec_time* 中的 *time_slot_id* 不能参照这个关系。

精明的读者可能会想，为什么我们在此前的章节中没有见过模式 *sec_course*、*sec_time_slot*、*sec_class*、*inst_dept*、*stud_dept* 以及 *course_dept*。原因是我们迄今所描述的算法使得一些模式或者被消除，或者和其他模式合并。我们接下来讨论这个问题。

7.6.4.1 模式的冗余

连接弱实体集和相应强实体集的联系集比较特殊。如 7.5.6 节提到的，这样的联系集是多对一的，且没有描述性属性。另外，弱实体集的主码包含强实体集的主码。在图 7-14 的 E-R 图中，弱实体集 *section* 通过联系集 *sec_course* 依赖于强实体集 *course*。*section* 的主码是 {*course_id*, *sec_id*, *semester*, *year*}，*course* 的主码是 *course_id*。由于 *sec_course* 没有描述性属性，因此 *sec_course* 模式有属性 *course_id*、*sec_id*、*semester* 以及 *year*。表示实体集 *section* 的模式包含属性 *course_id*、*sec_id*、*semester* 以及 *year* 等。*sec_course* 关系中的每个 (*course_id*, *sec_id*, *semester*, *year*) 组合也将出现在 *section* 模式上的关系中，反之亦然。因此，*sec_course* 模式是冗余的。

一般情况下，连接弱实体集与其所依赖的强实体集的联系集的模式是冗余的，而且在基于 E-R 图的关系数据库设计中不必给出。

7.6.4.2　模式的合并

287
288

考虑从实体集 A 到实体集 B 的一个多对一的联系集 AB。用前面讲的关系 - 模式构建算法，得到三个模式：A、B 和 AB。进一步假设 A 在该联系中的参与是全部的：即实体集 A 中的每个实体 a 都必须参与到联系 AB 中。那么我们可以将 A 和 AB 模式合并成单个包含两个模式所有属性的并集的模式。合并后模式的主码是其模式中能入了联系集模式的那个实体集的主码。

让我们检验图 7-15 的 E-R 图中满足上述条件的关系以讲解：

- *inst_dept*。模式 *instructor* 和 *department* 分别对应于实体集 A 和 B。因此模式 *inst_dept* 可以和模式 *instructor* 合并。结果是 *instructor* 模式由属性 {*ID*, *name*, *dept_name*, *salary*} 组成。
- *stud_dept*。模式 *student* 和 *department* 分别对应于实体集 A 和 B。因此模式 *stud_dept* 可以和模式 *student* 合并。结果是 *student* 模式由属性 {*ID*, *name*, *dept_name*, *tot_cred*} 组成。
- *course_dept*。模式 *course* 和 *department* 分别对应于实体集 A 和 B。因此模式 *course_dept* 可以和模式 *course* 合并。结果是 *course* 模式由属性 {*course_id*, *title*, *dept_name*, *credits*} 组成。
- *sec_class*。模式 *section* 和 *classroom* 分别对应于实体集 A 和 B。因此模式 *sec_class* 可以和模式 *section* 合并。结果是 *section* 模式由属性 {*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*} 组成。
- *sec_time_slot*。模式 *section* 和 *time_slot* 分别对应于实体集 A 和 B。因此模式 *sec_time_slot* 可以和上一步中得到的模式 *section* 合并。结果是 *section* 模式由属性 {*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*} 组成。

在一对一的联系的情况下，联系集的关系模式可以跟参与联系的任何一个实体集的模式进行合并。

即使参与是部分的，我们也可以通过使用空值来进行模式的合并。在上面这个例子中，如果 *inst_dept* 是部分参与的，那么我们可以为那些没有相关联的系的教师在属性 *dept_name* 中存放空值。

最后，我们考虑表示联系集的模式上本应有的外码约束。参照每一个参与联系集的实体集的外码约束本应存在。我们舍弃了参照联系集模式所合并入的实体集模式的约束，然后将另一个外码约束加到合并的模式中。在上面的例子中，*inst_dept* 上有一个 *dept_name* 属性参照 *department* 关系的外码约束，
289
当模式 *inst_dept* 与 *instructor* 合并时，这个外码约束被加到 *instructor* 关系中。

7.7　实体 - 联系设计问题

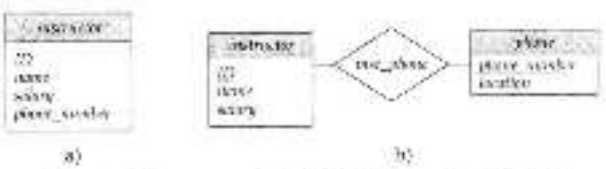
实体集和联系集的标记法并不精确，而且定义一组实体及它们的相互联系可能有多种不同的方式。本节讨论 E-R 数据库模式设计中的一些基本问题。设计过程将在 7.10 节更详细地讨论。

7.7.1　用实体集还是用属性

考虑具有新增属性 *phone_number* 的实体集 *instructor*（见图 7-17a）。显然电话可以作为一个单独的实体，具有属性 *phone_number* 和 *location*；地点可以是电话所处于的办公室或家，移动电话则可以用值“移动”来表示。如果我们采用这样的观点，那么我们就不给 *instructor* 增加属性 *phone_number*，反之，我们创建：

- 实体集 *phone*，具有属性 *phone_number* 和 *location*。
- 联系集 *inst_phone*，表示教师与他们所拥有的电话之间的关联。

这种方法如图 7-17b 所示。



a)　　b)

图 7-17　给 *instructor* 实体集增加 *phone* 的两种方法

那么，教师的这两个定义之间的主要差别是什么呢？将电话看成一个属性 *phone_number* 暗示每个

教师恰好有一个电话号码与之相关联。将电话看成一个实体 *phone*，允许每个教师可以有若干个电话号码(包括零个)与之相关联。然而，也可以简单地将 *phone_number* 定义为多值属性，从而允许每个教师有多个电话。

那么，主要的差别是，在一个人可能希望保存关于电话的额外信息，如它的位置，或类型(移动的、传真的或普通的老式电话)，或共享该电话的所有的人时，将电话看作一个实体是一种更好的建模方式。因此，把电话视为一个实体比把它视为一个属性的方式更具通用性；而且当通用性可能有用的时候，这种定义方式就更为适合了。

相反，将(一名教师的) *name* 属性视作一个实体就不太合适；将 *name* 说成是一个实体本身就很不
290 具有说服力(与电话相反)。因此，恰当的做法是将 *name* 视作 *instructor* 实体集的一个属性。

由此自然就产生两个问题：什么构成属性？什么构成实体集？很遗憾，对这两个问题并不能简单地回答。区分它们主要依赖于被建模的现实世界的企业的结构，以及被讨论的属性的相关讲义。

一个常见的错误是用一个实体集的主码作为另一个实体集的属性，而不是用联系。例如，即使每名教师只指导一名学生，将 *student* 的 *ID* 作为 *instructor* 的属性也是不正确的。用 *advisor* 联系代表学生和教师之间的关联才是正确的方法，因为这样可以明确地表示出两者之间的关系而不是将这种关系隐含在属性中。

人们常犯的与此相关的另一个错误是将相关实体集的主码属性作为联系集的属性。例如，*ID*(*student* 的主码属性)和 *ID*(*instructor* 的主码)不应该在 *advisor* 联系中作为属性出现。这样做是不对的，因为在联系集中已经隐含了这些主码属性[⊖]。

7.7.2 用实体集还是用联系集

一个对象最好被表述为实体集还是联系集并不总是显而易见的。如图 7-15 所示，我们用 *takes* 联系集对学生选择课程(的某一次开课)建模。另一种方法是想象对于每个学生选的每门课程有一个课程 - 注册记录。那么，我们用一个叫作 *registration* 的实体集代表课程 - 注册记录。每个 *registration* 实体恰好与一个学生和一次开课相关联，因此我们有两个联系集，一个将课程 - 注册记录和学生关联，另一个将课程 - 注册记录和课程关联。如图 7-18 所示，我们将图 7-15 中 section 和 student 实体集之间的 *takes* 联系集用一个实体集和两个联系集替代：

- *registration*，代表课程 - 注册记录的实体集。
- *section_reg*，关联 *registration* 和 *course* 的联系集。
- *student_reg*，关联 *registration* 和 *student* 的联系集。

291 注意，我们使用双线表示 *registration* 实体全部参与。

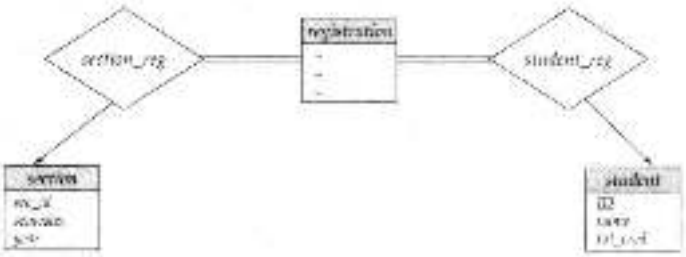


图 7-18 用 *registration* 和两个联系集替代 *takes*

图 7-15 和图 7-18 的方法都准确表达了大学的信息，但是使用 *takes* 的方法更紧凑也更可取。然而，如果注册办公室通过课程 - 注册记录与其他信息相关联，那么最好让它本身为一个实体。

在决定用实体集还是联系集时可采用的一个准则是，当描述发生在实体间的行为时采用联系集。这一方法在决定是否将某些属性表示为联系可能更适合时也很有用。

⊖ 我们以后会看到，当从 E-R 模式中创建关系模式时，这些属性可能会出现在从 *advisor* 联系集创建出的模式中，但是，它们并不应该出现在 *advisor* 联系集中。

7.7.3 二元还是 n 元联系集

数据库中的联系通常都是二元的。一些非二元的联系实际上可以用多个二元联系更好地表示。例如，可以创建一个二元联系 *parent*，将一个孩子与他/她的母亲和父亲相关联。然而，这一联系也可以用两个二元联系分别来表示，即 *mother* 和 *father*，分别将孩子与他/她的母亲和父亲相关联。使用 *mother* 和 *father* 两个联系集我们可以记录孩子的母亲，即使我们不知道父亲是谁；而对于这种情况三元联系 *parent* 中必须有一个空值。所以在这个例子中用二元联系更好。

事实上，一个非二元的(n 元，$n>2$)联系集总可以用一组不同的二元联系集来替代。简单起见，考虑一个抽象的三元($n=3$)联系集 R，它将实体集 A、B 和 C 联系起来。用实体集 E 替代联系集 R，并创建三个联系集，如图7-19所示：

- R_A，关联 E 和 A。
- R_B，关联 E 和 B。
- R_C，关联 E 和 C。

292



a) b)

图7-19 三元联系与三个二元联系

如果联系集 R 有属性，那么将这些属性赋给实体集 E；进一步，为 E 创建一个特殊的标识属性(因为它必须能够通过其属性值来区别实体集中的各个实体)。针对联系集 R 中的每个联系(a_i, b_i, c_i)，在实体集 E 中创建一个新的实体 e_i。然后，在三个新联系集中，分别插入新联系如下：

- 在 R_A 中插入(e_i, a_i)。
- 在 R_B 中插入(e_i, b_i)。
- 在 R_C 中插入(e_i, c_i)。

可以将这一过程直接推广到 n 元联系集的情况。因此，在概念上可以限制 E-R 模型只包含二元联系集。然而，这种限制并不总是令人满意的。

- 对于为表示联系集而创建的实体集，我们可能不得不为其创建一个标识属性。该标识属性和额外所需的那些联系集增加了设计的复杂程度以及对总的存储空间的需求(我们将在7.6节看到这一点)。
- n 元联系集可以更清晰地表示几个实体集参与单个联系集。
- 有可能无法将三元联系上的约束转变为二元联系上的约束。例如，考虑一个约束，表明 R 是从 A、B 到 C 多对一的：也就是，来自 A 和 B 的每一对实体最多与一个 C 实体关联。这种约束就不能用联系集 R_A、R_B 和 R_C 上的基数约束来表示。

考虑7.2.2节中的联系集 *proj_guide*，它关联 *instructor*、*student* 和 *project*。不能直接将 *proj_guide* 拆分为 *instructor* 和 *project* 之间的二元联系和 *student* 和 *project* 之间的二元联系。如果这么做，可以记录教 293
师 Katz 同学生 Shankar 和 Zhang 一起参与项目 A 和 B；然而无法记录 Katz 同 Shankar 一起参与项目 A 并且同 Zhang 一起参与项目 B，而不是同 Zhang 一起参与项目 A 或者同 Shankar 一起参与项目 B。

联系集 *proj_guide* 可以通过创建一个如上所述的新实体集来拆分为二元联系。然而，这么做却不是很自然。

7.7.4 联系属性的布局

一个联系的映射基数比率会影响联系属性的布局。因此，一对一或一对多联系集的属性可以放到一个参与该联系的实体集中，而不是放到联系集中。例如，我们指明 *advisor* 是一个一对多的联系集，也就是一个教师可以指导多个学生，但每个学生只能有一个导师。在这种情况下，表示教师何时成为

学生导师的属性 *date* 可以与 *student* 实体集相关联，如图 7-20 所示。（为了保持图例简单，只显示了两个实体集的部分属性。）由于每个 *student* 实体最多和一个 *instructor* 实例相关联，因此将属性 *date* 放在 *student* 实体集中和将属性 *date* 放在 *advisor* 联系集中具有相同的含义。一对多联系集的属性仅可以置于参与联系的"多"方的实体集中。而对于一对一的联系集，联系的属性可以放到任意一个参与联系的实体中。

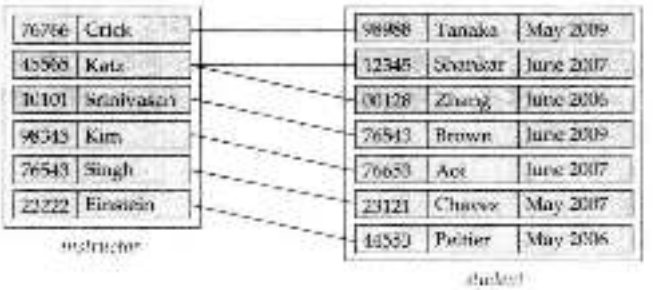


图 7-20 *date* 作为 *student* 实体集的属性

设计时将描述性属性作为联系集的属性还是实体集的属性这一决定应该反映出被建模企业的特点。
294 设计者可以选择保留 *date* 作为 *advisor* 的属性，以显式地表明指导关系的日期，而不是学生成员状态的其他一些方面（例如，被大学录取的日期）。

属性位置的选择在多对多联系集中体现得更清楚。回到刚才的例子，让我们指出可能更符合实际的情况，定义 *advisor* 为一个多对多的联系集，表明一个教师可以指导一个或多个学生，而一个学生可以被一个或多个教师指导。如果想要表示一个特定的教师成为一个特定学生的导师的日期，*date* 则必须作为联系集 *advisor* 的属性，而不是任何一个参与的实体集的属性。例如，如果将 *date* 作为 *student* 的属性，则我们无法知道哪个教师在该特定日期成为他的导师。当一个属性是由参与的实体集联合确定而不是由单独的某个实体集确定时，该属性就必须放到多对多联系集中。图 7-3 给出了作为联系属性时 *date* 的位置。为了图例的简单，只显示了两个实体集的部分属性。

7.8 扩展的 E-R 特性

虽然基本的 E-R 概念已足以对大多数数据库特征建模，但数据库的某些方面可以通过对基本 E-R 模型作某些扩展来更恰当地表述。这一节将讨论以下扩展 E-R 特性：特化、概化、高层和低层实体集、属性继承和聚集。

为了有助于讨论，我们将用一个稍微更复杂的大学数据库模式。特别是，我们将通过定义具有属性 *ID*、*name* 以及 *address* 的实体集 *person* 来对学校中不同的人建模。

7.8.1 特化

实体集可能包含一些子集，子集中的实体在某些方面区别于实体集中的其他实体。例如，实体集中的某个实体子集可能具有不被该实体集中所有实体所共享的一些属性。E-R 模型提供了表示这种与众不同的实体组（子集）的方法。

例如，实体集 *person* 可进一步归类为以下两类之一：

- *employee*。
- *student*。

这两个类中的每一个都用一个属性集来描述，包括实体集 *person* 的所有属性加上可能的附加属性。例如，*employee* 实体可进一步用属性 *salary* 来描述，而 *student* 实体可进一步用 *tot_cred* 属性来描述。在实体集内部进行分组的过程称为**特化**（specialization）。*person* 的特化使得我们可以根据他们是雇员还是学
295 生来区分人；一般来说，一个人可以是一个雇员、一个学生，都是，或者都不是。

另一个例子，假设大学希望将学生分为两类，研究生和本科生。给研究生配备办公室，而给本科生安排宿舍。每一个学生类型都通过属性集来描述，这个属性集包括 *student* 实体集的所有属性和附加

的属性。

大学可以创建 *student* 的两个特化，*graduate* 和 *undergraduate*。如我们此前所看到的，学生实体用属性 *ID*、*name*、*address* 以及 *tot_cred* 描述。实体集 *graduate* 将具有 *student* 的所有属性以及一个附加属性 *office_number*。实体集 *undergraduate* 将具有 *student* 的所有属性以及一个附加属性 *residential_college*。

我们可以不断重复地使用特化来完善设计。例如，大学雇员可进一步划分为以下两类之一：

- *instructor*。
- *secretary*。

每类雇员都用包括实体集 *employee* 的所有属性以及附加属性的属性集来描述。例如，*instructor* 实体可以进一步由属性 *rank* 来描述，而 *secretary* 实体可以由属性 *hours_per_week* 来描述。进一步，*secretary* 实体可以参与实体集 *secretary* 和 *employee* 之间的 *secretary_for* 联系，它标识了有秘书协助的雇员。

一个实体集可以根据多个可区分的特征进行特化。在我们的例子中，雇员实体间的可区分特征是雇员所从事的工作。同时，对一个特化可以基于一个人是临时（有限任期）雇员还是长期雇员，从而有实体集 *temporary_employee* 和 *permanent_employee*。当一个实体集上形成了多于一种特化时，某个特定实体可能同时属于多个特化实体集。例如，一个特定的雇员可以既是一个临时的雇员，又是一个秘书。

在E-R图中，特化用从特化实体指向另一方实体的空心箭头来表示（如图7-21所示）。我们称这种关系为ISA关系，它代表"is a"，表示"是一个"，例如，一个教师"是一个"雇员。

我们在E-R图中描述特化的方法取决于一个实体集是否可能属于多个特化实体集或者它是否必须属于至多一个特化实体集。前者（允许多个集）称为**重叠特化**（overlapping specialization），后者（允许至多一个）称为**不相交特化**（disjoint specialization）。对于一个重叠特化（例如 *student* 和 *employee* 作为 *person* 的特化的情况），分开使用两个箭头。对于一个不相交特化（例如 *instructor* 和 *secretary* 作为 *employee* 的特化的情况），使用一个箭头。特化关系还可能形成**超类 - 子类**（superclass-subclass）联系。高层和低层实体集按普通实体集表示——即包含实体集名称的矩形。

7.8.2 概化

从初始实体集到一系列不同层次的实体子集的细化代表了一个**自顶向下**（top-down）的设计过程，在这个设计过程中，显式地产生出差别。设计过程也可以**自底向上**（bottom-up）进行。多个实体集根据共同具有的特性综合成一个较高层的实体集。数据库设计者可能一开始就标识了：

- *instructor* 实体集，具有属性 *instructor_id*、*instructor_name*、*instructor_salary* 以及 *rank*。
- *secretary* 实体集，具有属性 *secretary_id*、*secretary_name*、*secretary_salary* 以及 *hours_per_week*。

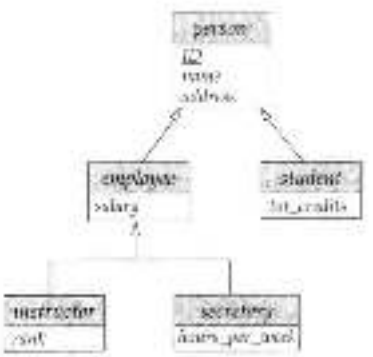


图7-21 特化和概化

就所具有的属性而言，在 *instructor* 实体集和 *secretary* 实体集间存在共性。从概念上来说，这两个实体集包含相同的属性：也就是标识符、姓名和工资属性。这种共性可以通过**概化**（generalization）来表达，概化是高层实体集与一个或多个低层实体集间的包含关系。在我们的例子中，*employee* 是高层实体集，而 *instructor* 和 *secretary* 是低层实体集。在这种情况下，在两个低层实体集中，概念上相同的属性有不同的名字。为了进行概化，这些属性必须读于相同的名字并由高层实体 *person* 表示。我们使用 *ID*、*name* 和 *address* 作为属性名称，正如我们在7.8.1节中所看到的那样。 296 / 297

高层与低层实体集也可以分别称作**超类**（superclass）和**子类**（subclass）。实体集 *person* 是子类 *employee* 和 *student* 的超类。

对于所有实际应用来说，概化只不过是特化的逆过程。为企业设计E-R模型时，我们将配合使用这两个过程。在E-R图中，我们对概化和特化的表示不作区分。为了使设计模式完全体现数据库应用和数据库用户的要求，我们会通过区分（特化）或综合（概化）来产生新的实体层次。这两种方式的区别主要在于它们的出发点和总体目标。

特化从单一的实体集出发，通过创建不同的低层实体集来强调同一实体集中不同实体间的差异。低层实体集可以有不适用于高层实体集中所有实体的属性，也可以参与到不适用于高层实体集中所有实体的联系中。设计者采用特化的原因正是为了表达这种与众不同的特征。如果 *student* 和 *employee* 与 *person* 实体拥有完全相同的属性，并且与 *person* 实体参与完全相同的联系，则没有必要特化 *person* 实体集。

概化的进行基于这样的认识：一定数量的实体集共享一些共同的特征（即用相同的属性描述它们，且它们都参与到相同的联系集中）。概化是在这些实体集的共性的基础上将它们综合成一个高层实体集。概化用于强调低层实体集间的相似性并隐藏它们的差异；由于共享属性的不重复出现，它还使得表达简洁。

7.8.3 属性继承

由特化和概化所产生的高层和低层实体的一个重要特性是**属性继承**（attribute inheritance）。高层实体集的属性被低层实体集**继承**（inherit）。例如，*student* 和 *employee* 继承了 *person* 的属性。因此，*student* 用属性 *ID*、*name* 和 *address* 以及附加属性 *tot_cred* 来描述；而 *employee* 用属性 *ID*、*name* 和 *address* 以及附加属性 *salary* 来描述。属性继承适用于所有低层实体集，因此 *employee* 的子类 *instructor* 和 *secretary* 从 *person* 继承了属性 *ID*、*name* 和 *address*，另外又从 *employee* 继承了 *salary*。

低层实体集（或子类）同时还继承地参与其高层实体（或超类）所参与的联系集。和属性继承类似，参与继承适用于所有低层实体集。例如，假设实体集 *person* 和 *department* 参与 *person_dept* 联系集，那么，实体集 *person* 的子类实体集 *student*、*employee*、*instructor* 和 *secretary* 也都和 *department* 隐式地参与到
298 *person_dept* 联系中。以上这些实体集可以参与到 *person* 实体参与的任何联系中。

对 E-R 图的一个给定的部分来说，不管它是通过特化还是通过概化得到的，其结果都是一样的：

- 高层实体集所关联的所有属性和联系适用于它的所有低层实体集。
- 低层实体集特有的性质仅适用于特定的低层实体集。

在后面的叙述中，虽然我们常常只说概化，但我们所讨论的性质是两个过程所共有的。

图 7-21 所示为实体集的**层次结构**（hierarchy）。在图 7-21 中，*employee* 是 *person* 的低层实体集，同时又是 *instructor* 和 *secretary* 实体集的高层实体集。在层次结构中，给定的实体集作为低层实体集只参与到一个 ISA 联系中，即在这个图中实体集只具有**单继承**（single inheritance）。如果一个实体集作为低层实体集参与到多个 ISA 联系中，则称这个实体集具有**多继承**（multiple inheritance），且产生的结构被称为**格**（lattice）。

7.8.4 概化上的约束

为了更准确地对企业建模，数据库设计者可能选择在特定概化上设置某些约束。一类约束包含判定哪些实体能成为给定低层实体集的成员。成员资格可以是下列中的一种：

- **条件定义的**（condition-defined）。在条件定义的低层实体集中，成员资格的确定基于实体是否满足一个显式的条件或谓词。例如，假设高层实体集 *student* 具有属性 *student_type*。所有 *student* 实体都根据 *student_type* 属性进行评估。只有满足条件 *student_type* = "研究生" 的实体才允许属于 *graduate_student* 低层实体集。所有满足条件 *student_type* = "本科生" 的实体都包含于 *undergraduate_student*。由于所有低层实体都基于同一属性（在这里基于 *student_type*）进行评估，因此这种类型的概化称作是**属性定义的**（attribute-defined）。
- **用户定义的**（user-defined）。用户定义的低层实体集不是通过成员资格条件来限制，而是由数据库用户将实体指派给某个实体集。例如，如果我们假设大学雇员在 3 个月的雇佣期后被分配到 4 个工作组中的一个。因此我们用高层实体集 *employee* 的 4 个低层实体集来表示工作组。一个给定的员工并不是根据某个明确定义的条件自动分配到某个特定工作组实体中。反而是负责决策的用户根据个人观点进行工作组的分配。该分派是通过将一个实体加入某个实体集的操作由
299 实现的。

另一类约束涉及在一个概化中一个实体是否可以属于多个低层实体集。低层实体集可能是下述情况之一：

- **不相交**(disjoint)。不相交约束要求一个实体至多属于一个低层实体集。在我们的例子中，*student* 实体在 *student* – *type* 属性上只能满足一个条件：一个实体可以是研究生或本科生，但不能既是研究生又是本科生。
- **重叠**(overlapping)。在重叠概化中，同一个实体可以同时属于同一个概化中的多个低层实体集。我们再来看员工工作组的例子，并假设某些雇员参加到多个工作组中。在这种情况下给定雇员就可以出现在 *employee* 的多个低层工作组实体集中。因此这种概化是重叠的。

在图 7-21 中，我们假定一个人可以既是雇员又是学生。我们通过分开的箭头表示这种重叠概化：一个箭头从 employee 指向 person，另一个箭头从 student 指向 person。然而，instructor 和 secretary 的概化是不相交的，我们对此用单个箭头表示。

最后一类约束是对概化的**完全性约束**(completeness constraint)，定义高层实体集中的一个实体是否必须至少属于该概化/特化的一个低层实体集。这种约束可以是下述情况之一：

- **全部概化**(total generalization)或**特化**(specialization)。每个高层实体必须属于一个低层实体集。
- **部分概化**(partial generalization)或**特化**(specialization)。允许一些高层实体不属于任何低层实体集。

部分概化是默认的。我们可以在 E-R 图中表示全部概化，即通过在图中加入关键词"total"，并画一条从关键词到相应的空心箭头(表示不相交概化)的虚线，或者画一条到空心箭头集合(表示重叠概化)的虚线。

student 的概化是全部的：所有的学生实体都要么是研究生要么本科生。由于通过概化产生的高层实体集通常只包括低层实体集中的实体，因此由概化产生的高层实体集的完全性约束通常是全部的。如果概化是部分的，高层实体就可以不限制出现在任何低层实体集中。工作组实体集就是部分特化的例子。由于员工在工作 3 个月后才分配到某个工作组中，因此某些 *employee* 实体可能不属于任何低层工作组实体集。

我们可以通过对 *employee* 的部分的、重叠的特化过程来更完全地表示出工作组实体集的特征。从 *graduate_student* 和 *undergraduate_student* 到 *student* 的概化是全部的、不相交的概化。然而，完全性约束 300
和不相交约束彼此并没有依赖关系。约束模式也可以是部分－不相交或全部－重叠的。

我们可以看到，对给定概化或特化使用约束带来某些插入和删除需求。例如，当在一个全部的完全性约束时，插入到高层实体集中的实体还必须插入到至少一个低层实体集中。在条件定义的约束中，所有满足条件的高层实体必须插入到相应的低层实体集中。最后，从高层实体集删除的实体也必须从它所属于的所有相应低层实体集中删除。

7.8.5　聚集

E-R 模型的一个局限性在于它不能表达联系间的联系。为了说明这种结构的必要性，考虑我们此前看到的 *instructor*、*student* 和 *project* 之间的三元联系 *proj_guide*(如图 7-13 所示)。

现在假设每位在项目上指导学生的教师需要记录月评估报告。我们将评估报告建模为一个主码为 *evaluation_id* 的实体 *evaluation*。记录一个 *evaluation* 对应的(*student*、*project*、*instructor*)组合的另一个方法是在 *instructor*、*student*、*project* 和 *evaluation* 之间建立一个四元(四路)联系集 *eval_for*。(四元的联系是必需的——例如，*student* 和 *evaluation* 之间的二元联系无法让我们表示某个 *evaluation* 对应的(*project*, *instructor*)组

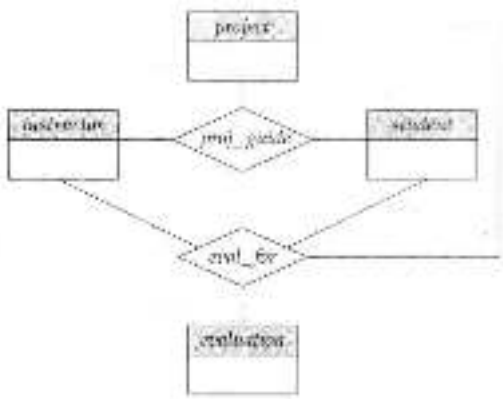


图 7-22　包含冗余联系的 E-R 图

合)。通过使用基本的E-R模型构建，我们得到了图7-22所示的E-R图。(为了简洁，省略了实体集的属性。)

看上去联系集 *proj_guide* 和 *eval_for* 可以合并到一个联系集中。然而，我们不应该将它们合并到一起，因为一些 *instructor*、*student*、*project* 组合可能没有相关联的 *evaluation*。

但是，按照这种方法产生的图存在冗余信息，因为在 *eval_for* 中的每个 *instructor*、*student*、*project* 组合肯定也在 *proj_guide* 中。如果 *evaluation* 是一个值而不是一个实体，我们可以将 *evaluation* 作为联系集 *proj_guide* 的一个多值属性。然而，当某个 *evaluation* 和其他实体相关联时，这种方法不可行；例如，每份评估报告可能和负责评估报告的后续处理以发放奖学金的 *secretary* 相关联。

对类似上述情况建模的最好办法是使用聚集。**聚集**(aggregation)是一种抽象，通过这种抽象，联系被视为高层实体。这样，对于我们的例子，我们将联系集 *proj_guide*(关联实体集 *instructor*、*student* 和 *project*)看成一个名为 *proj_guide* 的高层实体集。这种实体集可以像对任何其他实体集一样来处理。这样我们就可以在 *proj_guide* 和 *evaluation* 之间创建一个二元联系 *eval_for* 来表示某个 *evaluation* 对应哪个(*student*, *project*, *instructor*)组合。图7-23所示为用聚集概念来表示以上这种情况。

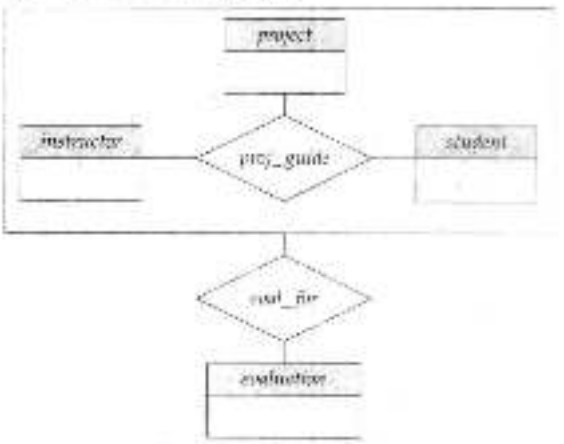


图7-23 包含聚集的E-R图

7.8.6 转换为关系模式

我们现在开始介绍扩展的E-R特性如何转换为关系模式。

7.8.6.1 概化的表示

为包含概化的E-R图进行关系模式设计有两种不同方法。尽管我们在下面的讨论中考虑图7-21的概化的情况，但为了简化讨论，我们谈到的内容实际上只包括第一层的低层实体集——*employee* 和 *student*。我们假定 *ID* 是 *person* 的主码。

- 为高层实体集创建一个模式。为每个低层实体集创建一个模式，模式中的属性包括对应于低层实体集的每个属性，以及对应于高层实体集主码的每个属性。因此，对于图7-21的E-R图(忽略 *instructor* 和 *secretary* 实体集)，有三个模式：

  $$person(\underline{ID}, name, street, city)$$
  $$employee(\underline{ID}, salary)$$
  $$student(\underline{ID}, tot\_cred)$$

  高层实体集的主码属性变成既是高层实体集的主码属性也是所有低层实体集的主码属性。如上例中下划线标记所示。

  另外，我们在低层实体集上建立外码约束，其主码属性参照创建自高层实体集的关系的主码。在上面的例子中，*employee* 的属性 *ID* 会参照 *person* 的主码，对于 *student* 类似。

- 如果概化是不相交且完全的——如果不存在同时属于两个同级的低层实体集的实体，且如果高层实体集的任何实体也都是某个低层实体集的成员——那么可以采用另一种表示方法。这时，我们不需要为高层实体集创建任何模式，只需要为每个低层实体集创建一个模式，模式中的属性包括对应于低层实体集的每个属性，以及对应于高层实体集的每个属性。那么，对于图7-21的E-R图，有两个模式：

  $$employee(\underline{ID}, name, street, city, salary)$$
  $$student(\underline{ID}, name, street, city, tot\_cred)$$

301 | 303

这两个模式都将高层实体集 *person* 的主码属性 *ID* 作为它们的主码。

第二种方法的一个缺点在于定义外码约束。为了说明这个问题，假定我们有一个与实体集 *person*

相关的联系集 R。用第一种方法，当从该联系集创建一个关系模式 R 时，主可以在 R 上建立参照 *person* 模式的外码约束。遗憾的是，如果用第二种方法，R 上的外码约束无法参照单一的一个关系。为了避免这个问题，需要创建一个关系模式 *person*，该模式至少包含实体 *person* 的主码属性。

如果将第一种方法用于重叠概化，某些值就会不必要地存储多次。例如，如果一个人既是雇员又是学生，*street* 和 *city* 的值就会存储两次。

如果概化是不相交的但不是全部的——有的人既不是雇员又不是学生——则将需要一个额外的模式

$$person(\underline{ID}, name, street, city)$$

来表示这样的人。然而，上述外码约束问题仍然存在。为了解决这个问题，假定在 *person* 关系中额外表示了雇员和学生，遗憾的是，姓名、街道以及城市的信息就要在 *person* 关系和 *student* 关系中冗余地存储。同样，雇员的信息在 *person* 关系和 *employee* 关系中冗余地存储。这就促使将姓名、街道和城市信息只存储在 *person* 关系中，而把这些信息从 *student* 和 *employee* 中移除。如果我们这么做的话，结果就跟我们上面讲述的第一种方法完全一样了。

7.8.6.2 聚集的表示

为包含聚集的 E-R 图进行模式设计是很直接的。考虑图 7-23 中的 E-R 图，表示聚集 *proj_guide* 和实体集 *evaluation* 之间联系的联系集 *eval_for* 的模式包含对应实体集 *evaluation* 主码中的每个属性以及联系集 *proj_guide* 主码中的每个属性。它还包含对应于联系集 *eval_for* 的任意描述性属性（如果存在的话）。然后，根据我们已经定义的规则，在聚集的实体集中转换联系集和实体集。

当把聚集像其他实体集一样看待时，我们此前看到的用于在联系集上创建主码和外码约束的规则，也同样可以应用于与聚集相关联的联系集。聚集的主码会定义该聚集的联系集的主码。不需要单独的关系来表示聚集；而使用从定义该聚集的联系创建出来的关系就可以了。

7.9 数据建模的其他表示法

一个应用的数据模型的图形表示对于数据库模式的设计至关重要。数据库模式的构建不仅需要数据建模专家，而且还需要了解应用的需求但可能并不熟悉数据建模的领域专家。一个直观的图形表示使两类专家间的信息沟通变得简单，因而尤其重要。 304

目前已经提出了许多种对数据建模的表示法，其中 E-R 图和 UML 类图的应用最为广泛。对于 E-R 表示法还没有一个统一的标准，不同的书籍以及 E-R 图软件使用不同的表示法。我们在本书第 6 版中选择了一种特定的表示法，它不同于本书以往版本使用的表示法，我们将在本节的稍后解释原因。

在本节剩下的部分中，我们学习一些可选择的 E-R 图表示法，以及 UML 类图表示法。为了将我们的表示法与其他表示法比较，图 7-24 总结了我们此前在 E-R 图表示法中用到的符号集。

7.9.1 E-R 图的其他表示法

图 7-25 列出一部分广泛使用的可选择的 E-R 表示法。表示实体的属性的一种可选的方法是将它们放入与代表实体的方框所连接的椭圆形中。主码属性以下划线表明。在图 7-25 的最上部展示了上述表示法。联系的属性也可以用类似的方式表示，即将包含属性的椭圆与表示联系的菱形连接起来。

如图 7-25 所示，联系上的基数约束可以用多种不同的方法表示。在图 7-25 的左部展示了一种可选择的方法，即在联系外面的边上标记 * 和 1，通常用来表示多对多、一对一和多对一联系。一对多的情况与多对一的情况是对称的，在此没有画出。

在图 7-25 的右部展示了另一种可选择的表示法，联系集是用实体集之间的连线而不是菱形来表示；因此只有二元联系才可以如此表示。如图 7-25 所示，在这种表示法中基数约束用"鸡爪形"表示法表示。在 $E1$ 和 $E2$ 之间的联系 R 中，两侧的鸡爪表示 $E1$ 到 $E2$ 的多对多的关系。在这个表示法中，竖线代表全部参与。不过要注意，在实体 $E1$ 和 $E2$ 间的联系 R 中，如果 $E1$ 对 R 全部参与，则要线放置在对面，即靠近 $E2$ 的位置；类似地，在对面画一个圆圈表示部分参与。

图 7-25 的底部是概化的另一种可选择的表示方法，即用三角形替代空心箭头。

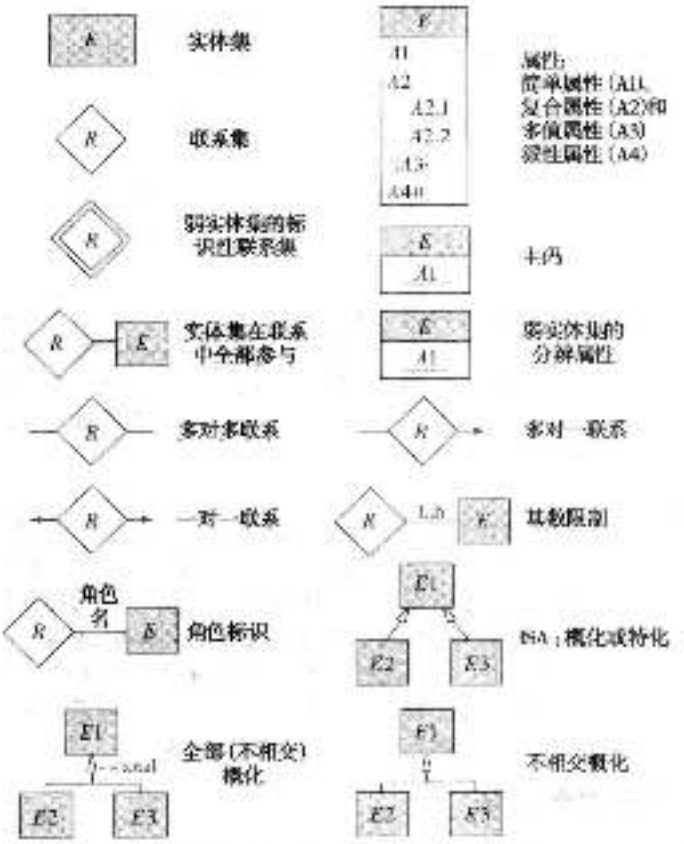


图 7-24 E-R 图表示法中使用的符号

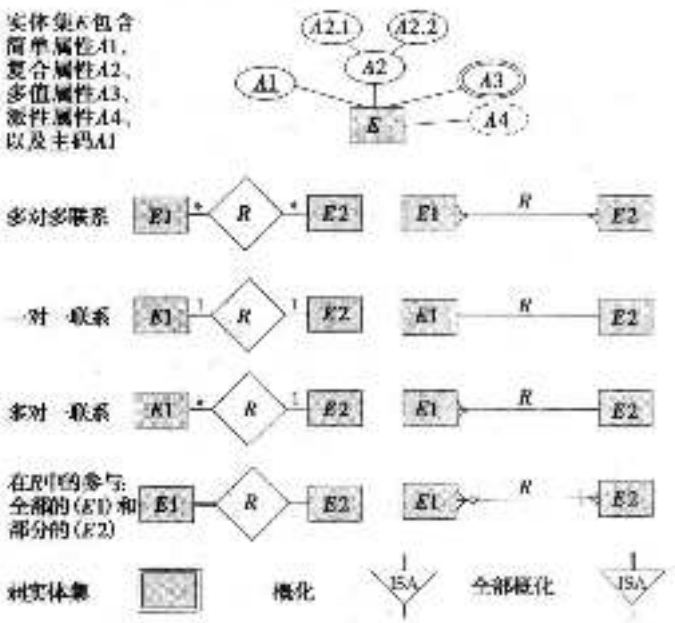


图 7-25 其他可选择的 E-R 图表示法

在本书第 5 版及以前的版本中，我们用椭圆形表示属性，用三角形表示概化，如图 7-25 所示。[1]

椭圆形表示属性并用菱形表示联系的表示法接近于Chen在他引入E-R模型概念的论文中所用的E-R图的原始形式，那种表示法目前称为陈氏表示法。

美国国家标准和技术研究院于1993年定义了一个称为IDEF1X的标准，其中使用了鸦爪形符号并在联系的边上加上竖线用于指明全部参与，加上空心圆表示部分参与，还包括了其他没有列出的符号。

随着统一标记语言(Unified Markup Language，UML)(将在7.9.2节介绍)使用的增多，我们选择更新E-R表示法，使之更接近UML类图的形式；7.9.2节将说明它们之间的联系。和此前的表示法相比，新的表示法对属性的表示更加紧凑，并且更接近许多E-R建模工具所支持的表示法，此外也更接近UML类图表示法。

目前有多种用于构建E-R图的工具，每个工具都有它自己的表示法变体。其中的一些工具甚至提供在几种不同的E-R表示法变体间的选择。更多信息请参考文献注解部分的引用文献。

在E-R图中的实体集与从这些实体集创建的关系模式之间一个关键的不同点在于，E-R联系对应的关系模式中的属性，比如*instructor*的属性*dept_name*，在E-R图中的实体集中并没有表示出来。一些数据建模工具允许用户在同一个实体的两种视图间选择，一种是不包含这些属性的实体视图，而另一种是包含这些属性的关系视图。

305
|
308

7.9.2 统一建模语言UML

实体-联系图有助于对软件系统的数据表示部分建模。然而，数据表示只构成整个系统设计的一部分。其他部分包括系统用户界面的建模，系统功能模块的规范定义以及它们之间的交互等。**统一建模语言**(Unified Modeling Language，UML)是由对象管理组织(Object Management Group，OMG)主持开发的一个标准，它是为了建立软件系统不同部分的规范定义而提出的。UML的一些组成部分为：

- **类图**(class diagram)。类图和E-R图相类似。本节将说明类图的一些特征及其与E-R图的关系。
- **用况图**(use case diagram)。用况图说明了用户和系统之间的交互，特别是用户所执行的任务中的每一步操作(如取钱或注册课程)。
- **活动图**(activity diagram)。活动图说明了系统不同部分之间的任务流。
- **实现图**(implementation diagram)。实现图在软件构件层和硬件构件层说明了系统的各组成部分以及它们之间的联系。

在这里我们不准备提供UML各部分的细节。关于UML的参考文献请参考文献注解。不过，我们将通过一些例子来说明UML中与数据建模有关的部分的一些特征。

图7-26显示了几个E-R图的构造和与它们等价的UML类图的构造。我们将在下面描述这些构造。事实上UML为对象建模，而E-R为实体建模。对象和实体很像，也有属性，但是另外还提供一组函数(称为方法)，它们可在对象属性的基础上调用以计算值，或也更新对象本身。类图除了可以说明属性外，还可以说明方法。我们在第22章讨论对象。UML不支持复合或多值属性，派生属性与不带参数的函数等价。由于类支持封装，因此UML允许属性和函数带有前缀"+"、"-"或"#"，这分别表示公共、私有以及受保护的访问。私有属性只能在类的方法中使用，而受保护的属性只能在类和它的子类的方法中使用；了解Java、C++或C#的人应该对此很熟悉。

在UML术语中，联系集称为**关联**(association)；为了与E-R术语一致，我们将仍称它们为联系集。在UML中我们通过划一条线段连接实体集来表示二元联系集。我们将联系集的名称写在线段的附近。我们还可以通过将角色的名称写在靠近实体集的线段上，来说明联系集中一个实体集的角色。另一种方法是，我们可将联系集的名字写在方框里，和联系集的属性写在一起，并用虚线把这个方框连接到表示联系集的连线上。这个方框可以看作一个实体集，如同E-R图中的聚集一样，也可以与其他实体集一起参与联系。

从UML 1.3版开始，UML通过使用与E-R图中使用的一样的菱形表示法支持非二元联系。而在更早版本的UML中无法直接表示非二元联系——非二元联系必须用我们在7.7.3节中看见的技术转换成二元联系。即使对于二元联系，UML也允许使用菱形表示法，但是大部分设计者使用线段表示法。

基数约束在UML中和E-R图中一样用$l..h$的形式表示，其中l表示参与联系的实体的最小个数，h则表示最大个数。然而，如图7-26所示，你应该注意到约束的位置和在E-R图中约束的位置正好相

反。*E2* 边上的约束 0..* 和 *E1* 边上的 0..1 表示每个 *E2* 实体可以参与至多一个联系，而每个 *E1* 实体可以参与很多联系；换句话说，该联系是从 *E2* 到 *E1* 多对一的。

像 1 或 * 这样的单个值可以写在边上；边上的单个值 1 视为与 1..1 等价，而 * 等价于 0..*。UML 支持概化，表示法与 E-R 表示法基本相同，包括不相交概化和重叠概化的表示方式。

UML 类图还包含一些其他的表示法，与我们见过的 E-R 表示法并不对应。例如，连接两个实体集的一条线的一端有一个小菱形，表示菱形这一端的实体集包含另一个实体集（包含关系在 UML 术语中称为"聚合"，不要将聚合的这种用法同 E-R 模型中聚集的含义混淆）。例如，一个车辆实体可能包含一个发动机实体。

UML 类图还提供了表示面向对象语言的特征的表示法，例如接口。关于 UML 类图的更多信息，请参看文献注解中的引用文献。

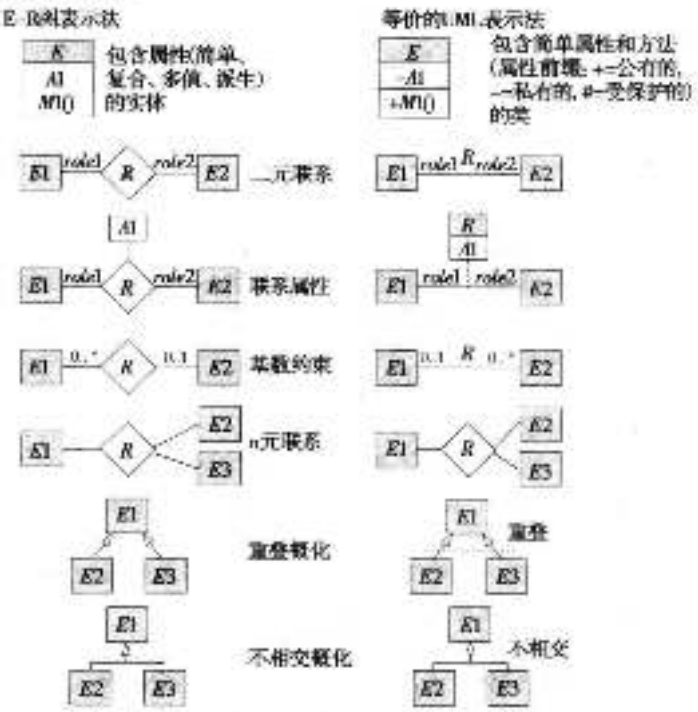


图 7-26 UML 类图表示法中使用的符号

7.10 数据库设计的其他方面

本章对模式设计的扩展讨论可能会造成模式设计是数据库设计的唯一组成部分的错误印象。我们将在后续章节中更完整地讲述一些其他的考虑，而在此，我们先简要地概览一下。

7.10.1 数据约束和关系数据库设计

我们已经看到，使用 SQL 可以表达多种数据约束，包括主码约束、外码约束、check 约束、断言和触发器。约束用于多种目的。最明显的一个目的是自动的一致性保持。通过在 SQL 数据定义语言中表达约束，设计者能够确保数据库系统自己执行这些约束。这比让每个应用程序自己独立地执行约束要可靠得多。同时，这种机制也为更新和添加约束提供了一个集中的位置。

显式声明约束的另一个优点是一些约束在关系数据库模式的设计中特别有用。例如，如果我们知道社会保障号唯一地标识一个人，那么我们就可以使用一个人的社会保障号来关联与这个人相关的数据，即使这些数据出现在多个关系中。与此相反，比如，眼睛的颜色不是唯一标识符。眼睛的颜色不能用于关联与某个人相关的多个关系中的数据，因为这样就无法将一个人的数据同其他具有同样眼睛颜色的人的相关数据区分开来。

在7.6节中，通过使用设计中指定的约束，我们为一个给定的E-R设计完成了一个关系模式集合。在第8章中，我们将这个思想及其相关约束形式化，并展示它将如何辅助关系数据库模式的设计。关系数据库设计的规范化方法使我们能够在给定的设计是好的设计时用准确的方式进行表述，并且能够把不好的设计转变为较好的设计。我们将看到，这个从实体－联系设计开始并根据该设计规则地生成关系模式的过程给整个设计过程提供了一个好的开始。 309 310

数据约束在确定数据的物理结构时同样有用，可以将彼此紧密相关的数据存储在磁盘上邻近的区域，以便在磁盘访问时提高效率。当索引建立在主码上时，索引结构工作得更好。

每次数据库更新时，执行约束会在性能上带来潜在的高代价。对于每次更新，系统都必须检查所有的约束，然后要么拒绝与约束冲突的更新，要么运行相应的触发器。性能损失的严重性不仅仅取决于更新的频率，而且依赖于数据库的设计方式。对某些类型的约束进行检测的真实效率，是第8章中对关系数据库模式设计的讨论的一个重要方面。

7.10.2 使用需求：查询、性能

数据库系统的性能是绝大多数企业信息系统的一个关键因素。性能不仅与计算能力的有效利用以及所使用的存储硬件有关，而且受到与系统交互的人的效率以及依赖数据库数据的处理的效率的影响。

以下是效率的两个主要度量方法：

- **吞吐量**(throughput)——每单位时间里能够处理的查询或更新(通常指事务)的平均数量。
- **响应时间**(response time)——单个事务从开始到结束所需的平均时间或者最长时间。

以批量的方式处理大量事务的系统关注于达到高吞吐量。与人交互或者时间苛刻的系统则通常关注于响应时间。这两个度量并不等价。高吞吐量的目的是为了获得系统部件的高利用率。这样做的时候可能会导致某些事务延迟，直到它们能更高效地运行的时候才处理。所以这些延迟的事务的响应时间就很差。

大多数商用数据库系统长期以来都关注于吞吐量，但是，包括基于Web的应用和电信信息系统等在内的许多应用都要求好的平均响应时间和适当限制内的最差响应时间。

了解预期最频繁的查询的类型有助于设计过程。涉及连接的查询比不涉及连接的查询需要更多的资源用以计算。在需要连接的情况下，数据库管理员可以选择创建一个索引，加快连接的计算。对于查询——不论是否涉及连接——可以创建索引以加速常常出现在查询中的选择谓词(SQL的**where**子句)的计算。查询中另一个影响索引选择的因素是更新和读操作的混合。当一个索引可能加速查询的同时，它也可能减缓更新的速度，因为更新会为维护索引的准确性而强制性地带来额外的工作。 311

7.10.3 授权需求

授权约束同样会影响数据库的设计。因为SQL允许在数据库逻辑设计组件的基础上将访问权限授予用户。一个关系模式可能需要分解为两个或多个模式，以便于在SQL中授予访问权限。比如，一个雇员记录可以包括与工资单、工作职责和医疗保险相关的数据。由于企业的不同管理部门分别管理该数据的不同部分，有些用户需要访问工资数据，但却不能访问工作数据和医疗数据等。如果这些数据在一个表中，希望得到的访问划分通过使用视图尽管依然可行，但更加麻烦。当数据在一个计算机网络中的多个系统中分散存放时，这种方式的数据划分则是必不可少的，这个问题将在第19章讨论。

7.10.4 数据流、工作流

数据库应用通常是大型企业应用的一部分，这样的应用不仅与数据库系统交互，而且会同各种专门的应用交互。例如，在一个制造公司中，计算机辅助设计(CAD)系统会用于辅助新产品的设计。CAD系统可以通过SQL语句从数据库中抽取数据，在其内部处理这些数据，也许同时和产品设计人员进行交互，然后再更新数据库。在这个过程中，数据的控制权会在几个产品设计人员和其他人之间传递。再看一个例子，考虑一个差旅费报告。它由一个出差归来的雇员写成(可能利用某个专门的软件包)，然后依次交给该雇员的经理，可能的其他高层经理，最后交到了财务部门以报销费用(在此处它将与该企业的财务信息系统交互)。

术语工作流表示一个流程中的数据和任务的组合，前面给出了这种流程的两个例子。当工作流在

用户间移动以及用户执行他们在工作流中的任务时，工作流会与数据库系统交互。除了工作流操作的数据之外，数据库还可以存储工作流自身的数据，包括构成工作流的任务以及它们在用户之间移动的路径。因此工作流可以列出一系列对数据库的查询和更新，而这些可能会作为数据库设计过程的一个部分考虑进来。换句话说，对企业建模要求我们不仅要理解数据的语义，还要理解使用这些数据的业
312 务流程。

7.10.5 数据库设计的其他问题

数据库设计通常不是一个一蹴而就的工作。一个组织的需求不断发展，它所需要存储的数据也会相应地发展。在最初的数据库设计阶段，或者在应用的开发中，数据库设计者都有可能意识到在概念、逻辑和物理模式层次上要有所改变。模式上的改变会影响到数据库应用的方方面面。一个好的数据库设计会预先估计一个组织将来的需要，设计出的模式在需求发展时只需要做最少的改动即可满足要求。

区分预期持久的基本约束和预期要改变的约束非常重要。例如，教师编号能唯一地标识一名教师的约束是基本的。另一方面，大学可能有一项规定，一名教师只能属于一个系，这条规定也许会在将来改变。如果允许联合任命的话，只允许每个教师属于一个系的数据库设计在允许联合任命时会需要较大的改动。只要每个教师只有一个主系，这种联合任命就可以通过新增一个关系来表示，而不需要修改 *instructor* 关系。而如果规定改成允许一个教师有多个主系时，则数据库设计就需要进行较大的改动。一个好的设计应该不止考虑当前的规定，还应该避免或者最小化由预计或有可能发生的改变而带来的改动。

进一步说，数据库所服务的企业很可能会与其他企业交互，因此，多个数据库可能需要进行交互。不同模式之间的数据转换在现实世界的应用中是一个重要的问题。对于这个问题提出了很多解决方案。我们将在第 23 章学习的 XML 数据模型，在不同应用间交换数据时，广泛用于表示数据。

最后，不言而喻，数据库设计在两个意义上是面向人的工作：系统的最终用户是人（即使有应用程序位于数据库和最终用户之间）；数据库设计者需要与应用领域的专家进行广泛交互以理解应用的数据需求。所有涉及数据的人都有需要和偏好，为了数据库设计和部署在企业中获得成功，这些都应当考虑到。

7.11 总结

- 数据库设计主要涉及数据库模式的设计。**实体 - 联系**（Entity-Relationship，E-R）数据模型是一个广泛用于数据库设计的数据模型。它提供了一个方便的图形化表示方法以查看数据、联系和约束。
313
- E-R 模型主要用于数据库设计过程。它的发展是为了帮助数据库设计，这是通过允许定义企业模式（enterprise schema）实现的。这种企业模式代表数据库的全局逻辑结构。该全局结构可以用 E-R 图（E-R diagram）图形化地表示。
- **实体**（entity）是在现实世界中存在并且区别于其他对象的对象。我们通过把每个实体同描述该实体的一组属性相关联来表示区别。
- **联系**（relationship）是多个实体间的关联。相同类型的联系的集合为**联系集**（relationship set），相同类型的实体的集合为**实体集**（entity set）。
- 术语超码（superkey）、**候选码**（candidate key）以及**主码**（primary key）同适用于关系模式一样适用于实体和联系集。在确定一个联系集的主码时需要小心，因为它由来自一个或多个相关的实体集的属性组成。
- **映射的基数**（mapping cardinality）表示通过联系集可以和另一实体相关联的实体的个数。
- 不具有足够属性构成主码的实体集称为**弱实体集**（weak entity set）。具有主码的实体集称为**强实体集**（strong entity set）。
- E-R 模型的各种性质为数据库设计者提供了大量的选择，使设计人员可以最好地表示被建模的企业。在某些情况中，概念和对象可以用实体、联系或属性来表示。企业总体结构的各方面可以用弱实体集、特化、特化或聚集很好地描述。设计者通常需要在简单的、紧凑的模型与更精确但也更复杂的模型之间进行权衡。

- 用E-R图定义的数据库设计可以用关系模式的集合来表示。数据库为每个实体集和联系集都有唯一的关系模式与之对应，其名称即为相应的实体集或联系集的名称。这是从E-R图转换为关系数据库设计的基础。
- **特化**(specialization)和**概化**(generalization)定义了一个高层实体集和一个或多个低层实体集之间的包含关系。特化是取出高层实体集的一个子集来形成一个低层实体集。概化是用两个或多个不相交的(低层)实体集的并集形成一个高层实体集。高层实体集的属性被低层实体集继承。
- **聚集**(aggregation)是一种抽象，其中联系集(和跟它们相关的实体集一起)被看作高层实体集，并且可以参与联系。
- UML是一种常用的建模语言。UML类图广泛用于对类建模以及一般的数据建模。 314

术语回顾

- 实体-联系数据模型
- 实体和实体集
  - 属性
  - 域
  - 简单和复合属性
  - 单值和多值属性
  - 空值
  - 派生属性
- 超码、候选码以及主码
- 联系和联系集
  - 二元联系集
  - 联系集的度
  - 描述性属性
  - 超码、候选码以及主码
  - 角色
  - 自环联系集
- E-R图
- 映射基数
  - 一对一联系
  - 一对多联系
  - 多对一联系
  - 多对多联系
- 参与
  - 全部参与
  - 部分参与
- 弱实体集和强实体集
  - 分辨符属性
  - 标识联系
- 特化和概化
  - 超类和子类
  - 属性继承
  - 单和多继承
  - 条件定义的和用户定义的成员资格
  - 不相交概化和重叠概化
  - 全部概化和部分概化
- 聚集
- UML
- UML类图

实践习题

7.1 为车辆保险公司构建一个E-R图，它的每个客户有一辆或多辆车。每辆车关联零次或任意次事故的记录。每张保险单为一辆或多辆车保险，并与一个或多个保费支付相关联。每次支付只针对特定的一段时间。具有关联的到期日和缴费日。

7.2 考虑一个用于记录学生在不同开课(section)的不同考试中所得成绩的数据库。 315

- 为数据库构造一个E-R图，其中，将考试建模为实体并使用一个三元联系。
- 构造另一个E-R图，其中，只用*students*和*section*间的二元联系。保证在特定*student*和*section*对之间只存在一个联系；而且你可以表示出学生在不同考试中所得成绩。

7.3 设计一个E-R图用于跟踪记录你最喜欢的球队的成绩。你应该保存打过的比赛，每场比赛的比分，每场比赛的上场队员以及每个队员在每场比赛中的统计数据。总的统计数据应该被建模成派生属性。

7.4 考虑一个E-R图，其中相同实体集出现数次，且它的属性重复出现多次。为什么这样的冗余是应尽量避免的不良设计？

7.5 E-R图可视为一个图。下面这些术语对于一个企业模式的结构意味着什么？

a. 图是非连通的。

b. 图是有环的。

7.6 如7.7.3节描述及图7-27b所示(属性没有列出)，考虑用多个二元联系来表示一个三元联系。

a. 给出E、A、B、C、R_A、R_B和R_C的一个简单实例，这个实例不能对应于A、B、C和R的任何实例。

b. 修改图7-27b的E-R图，引入约束，确保E、A、B、C、R_A、R_B和R_C的任意满足约束的实例将对应于A、B、C和R的一个实例。

c. 修改以上的转换以处理该三元联系上的全部参与约束。

d. 以上表示方式需要我们为E创建一个主码属性。试问如何将E看成弱实体集从而不需要主码属性？

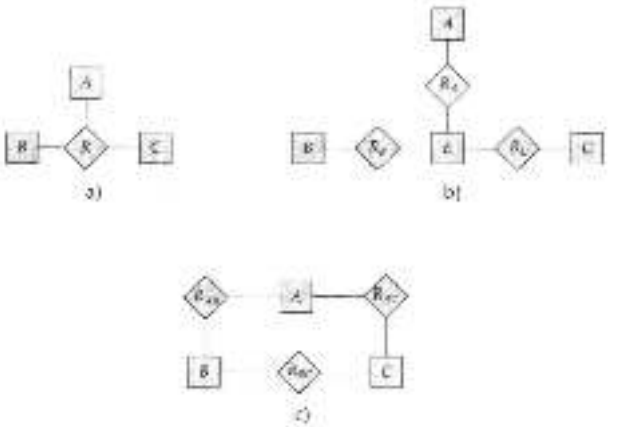


图 7-27 实践习题 7.6 和习题 7.24 的 E-R 图

7.7 一个弱实体集总可以通过给自己的属性中加入其标识实体集的主码属性而变成一个强实体集。权衡一下如果我们这么做会产生什么样的冗余。

7.8 考虑一个关系，例如 *sec_course*，产生十多对一的联系 *sec_course*。在这个关系上创建的主码约束和外码约束是否强制实施多对一的基数约束？解释为什么。

7.9 假设 *advisor* 联系是一对一的。为了确保执行一对一基数约束，在关系 *advisor* 上需要哪些额外的约束？

7.10 考虑实体集 A 和 B 之间多对一的联系 R。假设从 R 生成的关系和 A 生成的关系合并了。在 SQL 中，参与外码约束的属性可以为空。解释如何使用 SQL 上的 not null 约束来强制实施 A 在 R 中的全部参与约束。

7.11 在 SQL 中，外码约束只能参照被参照关系的主码属性，或者其他用 unique 约束声明为超码的属性。这导致多对多联系（或者一对多联系的“一”方）上的全部参与约束在从该联系创建的关系上无法用关系上的主码、外码以及非空约束强制实施。

a. 解释为什么。

b. 解释如何使用复杂的 check 约束或断言（见 4.4.7 节）强制实施全部参与约束。（遗憾的是，在目前广泛使用的数据库中并不支持这些特性。）

7.12 图 7-28 所示为概化和特化的一个格结构（属性没有给出）。针对实体集 A、B 和 C，说明如何从高层实体集 X 和 Y 继承属性。讨论 X 的一个属性和 Y 的某个属性同名时应如何处理。

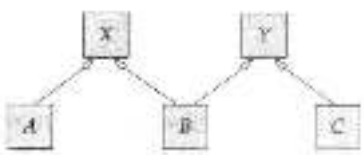


图 7-28 实践习题 7.12 的 E-R 图

7.13 **时间变化**（temporal change）：E-R 图通常对一个企业在某个时间点的状态建模。假设我们希望追踪时间变化，即数据随时间的变化。例如，张可能在 2005 年 9 月 1 日至 2009 年 5 月 31 日之间是一名学生，而 Shankar 在 2008 年 5 月 31 日至 2008 年 12 月 5 日以及 2009 年 6 月 1 日至 2010 年 1 月 5 日期间的导师是 Einstein。类似地，一个实体或联系的属性值会随时间发生变化。例如 *course* 的 *title* 和 *credits*，*instructor* 的 *salary* 甚至 *name*，以及 *student* 的 *tot_cred*。

对时间变化建模的一种方法如下。我们定义一个新的数据类型**有效时间**（valid_time），它是一个时间段或者时间段的集合。然后我们将每一个实体和联系都与一个 *valid_time* 属性关联起来，记录实体或联系有效的时间段。一个时间段的结束时间可以是无穷，例如，如果 Shankar 在 2008 年 9 月 2 日成为学生，并且目前仍为学生，我们可以将 Shankar 实体 *valid_time* 时间段的结束时间表示为无穷。类似地，我们将值随时间变化的属性建模为值的集合，每一个值具有自己的 *valid_time*。

a. 画一个包含 *student* 和 *instructor* 实体以及 *advisor* 联系并具有上述扩展以追踪时间变化的 E-R 图。

b. 将上面的 E-R 图转换为一个关系集合。

很明显，上面产生的关系集非常复杂，使得像写 SQL 语句这样的任务比较难完成。另一种使用得

比较多的方法是在设计 E-R 模型的时候忽略时间变化(尤其是属性值随时间变化)，然后修改由 E-R 模型生成的关系以追踪时间变化，8.9 节将对此进行讨论。 316 318

习题

7.14 解释主码、候选码和超码这些术语之间的区别。

7.15 为医院构建一个包含一组病人和一组医生的 E-R 图。为每个病人关联一组不同的检查和化验记录。

7.16 为实践习题 7.1～7.3 中的每个 E-R 图构建适当的关系模式。

7.17 扩展实践习题 7.3 中的 E-R 图，以追踪单个联赛中所有队伍的相同信息。

7.18 解释弱实体集和强实体集之间的区别。

7.19 我们可以通过简单地增加一组适当的属性，将任意弱实体集转变成强实体集。那么，我们为什么还要弱实体集呢?

7.20 考虑图 7-29 中的 E-R 图，它对一家网上书店建模。

a. 列出实体集及其主码。

b. 假设书店向其顾客的商品中增加了蓝光光盘和可下载视频。相同的商品可能在一种格式中存在，或在两种格式中都存在且具有不同的价格。扩展 E-R 图来为这个新增需求建模，忽略对购物篮的影响。

- 现在用概化来扩展 E-R 图，从而对包含书、蓝光光盘或可下载视频的任意组合的购物篮进行建模。

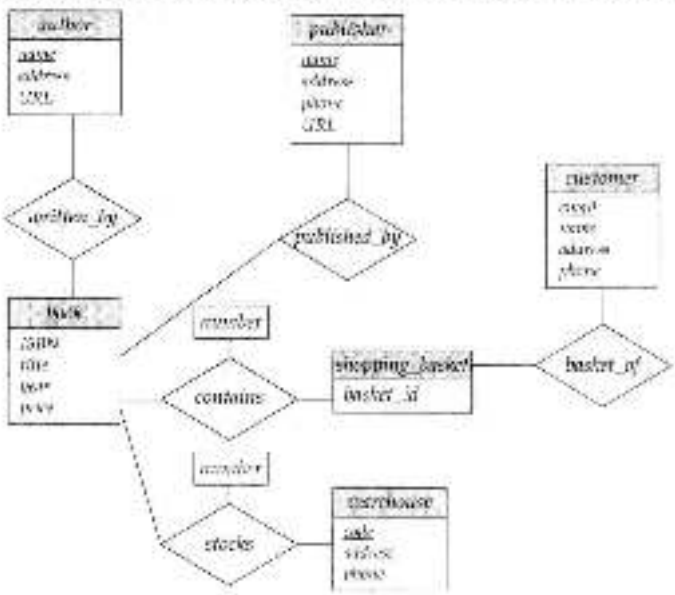


图 7-29 习题 7.20 的 E-R 图

7.21 为一个汽车公司设计一个数据库，用于协助它的经销商维护客户记录以及经销商库存，并协助销售人员订购车辆。

每辆车由车辆编号(Vehicle Identification Number，VIN)唯一标识，每辆单独的车都是公司委托的特定品牌的特定模型(例如，XF 是塔塔汽车捷豹品牌的模型)。每个模型都可以有不同的选项，但是一辆车可能只有一些(或没有)可用的选项。数据库需要保存关于模型、品牌、选项的信息，以及每个经销商、顾客和车的信息。

你的设计应该包括 E-R 图，关系模式的集合，以及包括主码约束和外码约束的一组约束。

7.22 为全球性的快递公司(例如 DHL 或者 FedEX)设计一个数据库。数据库必须能够追踪(寄件的)客户和(取件的)客户；有些客户可能两者都是。由于每个包裹必须是可标识且可追踪的，因此数据库必须能够存储包裹的位置信息以及它的历史位置。位置包括卡车、飞机、机场和仓库。 319

你的设计应该包括 E-R 图，关系模式的集合，以及包括主码约束和外码约束的一组约束。

7.23 为航空公司设计一个数据库，数据库必须追踪客户和他们的预订、航班、他们在航班上的状态、座位分配，以及未来航班的时刻表和飞行路线。

你的设计应该包含 E-R 图、关系模式的集合，以及包括主码约束和外码约束的一组约束。

7.24 在 7.7.3 节，我们用多个二元联系（如图 7-27b 所示）来表示一个三元联系（见图 7-27a）。考虑图 7-27c 中的另一种方式。讨论这两种用多个二元联系来表示一个三元联系的方式的相对优点。

320

7.25 考虑 7.6 节中所示从图 7-15 的 E-R 图转化而来的关系模式。对于每个模式，指出应该创建的外码约束（如果有的话）。

7.26 为机动车辆销售公司设计一个概化—特化层次结构。该公司出售摩托车、小客车、货车和大巴。论述你在层次结构的各层设定属性位置的合理性。说明为什么它们不应放在较高的层次或较低的层次。

7.27 说明条件定义的和用户定义的约束的差别。系统能自动检查哪种约束？解释你的答案。

7.28 说明不相交约束和重叠约束之间的区别。

7.29 解释全部约束和部分约束之间的区别。

工具

许多数据库系统提供支持 E-R 图的数据库设计工具。这些工具有助于设计者创建 E-R 图，并且它们可在数据库中自动创建相应的表。可参看第 1 章文献注解中给出的数据库系统厂商的网址。

还存在一些独立于数据库并且支持 E-R 图和 UML 类型的数据建模工具。画图工具 Dia 是一款免费软件，支持 E-R 图和 UML 类图。商用工具包括 IBM Rational Rose（www.ibm.com/software/rational）、Microsoft Visio（见 www.microsoft.com/office/visio）、CA 的 ERwin（www.ca.com/us/datamodeling.aspx）、Poseidon for UML（www.gentleware.com）和 SmartDraw（www.smartdraw.com）。

文献注解

E-R 数据模型由 Chen[1976]提出。使用扩展 E-R 模型的关系数据库的逻辑设计方法学由 Teorey 等[1986]给出。由美国国家标准与技术研究院（NIST）所发布的信息建模综合定义（IDEF1X）标准 IDEF1X[1993]定义了 E-R 图标准。然而，目前使用的 E-R 表示法有很多种。

Thalheim[2000]提供了关于 E-R 建模研究的一个详尽的教科书。Batini 等[1992]以及 ElMasri 和 Navathe[2006]给出了与此相关的基本教授。Davis 等[1983]提供了关于 E-R 模型的论文集。

321
322

到 2009 年为止，UML 的版本已经升至 2.2，而 UML 2.3 版本正接近最终版。关于 UML 标准及工具的更多信息，请参看 www.uml.org。

关系数据库设计

在本章中，我们考虑为关系数据库设计模式的问题。其中的许多问题和我们在第7章中使用E-R模型时所考虑的设计问题相似。

一般而言，关系数据库设计的目标是生成一组关系模式，使我们存储信息时避免不必要的冗余，并且让我们可以方便地获取信息。这是通过设计满足适当范式(normal form)的模式来实现的。要确定一个关系模式是否属于期望的范式，我们需要数据库所建模的真实企业的信息。其中的一部分信息存在于设计良好的E-R图中，但是可能还需要关于该企业的额外信息。

本章介绍基于函数依赖概念的关系数据库设计的规范方法。然后根据函数依赖及其他类型的数据依赖来定义范式。不过，我们首先从给定实体-联系设计转换得到的模式的角度，考察关系设计的问题。

8.1 好的关系设计的特点

第7章对实体-联系设计的研究为创建关系数据库设计提供了一个很好的起点。我们在7.6节中看到，可以直接从E-R设计生成一组关系模式。显然，生成的模式集的好(或差)首先取决于E-R设计的质量。本章后面将研究评价一组关系模式的满意度的精确方法。不过，通过利用我们已学过的概念，我们可以向好的设计迈一大步。

为了易于参照，我们在图8-1中重复大学数据库的模式。

classroom(*building*, *room_number*, *capacity*)
department(*dept_name*, *building*, *budget*)
course(*course_id*, *title*, *dept_name*, *credits*)
instructor(*ID*, *name*, *dept_name*, *salary*)
section(*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*)
teaches(*ID*, *course_id*, *sec_id*, *semester*, *year*)
student(*ID*, *name*, *dept_name*, *tot_cred*)
takes(*ID*, *course_id*, *sec_id*, *semester*, *year*, *grade*)
advisor(*s_ID*, *i_ID*)
time_slot(*time_slot_id*, *day*, *start_time*, *end_time*)
prereq(*course_id*, *prereq_id*)

图8-1 大学数据库模式

8.1.1 设计选择：更大的模式

现在，让我们探究一下这个关系数据库设计的特点，以及一些其他的选择方案。假定我们用以下这个模式代替*instructor*模式和*department*模式：

inst_dept (*ID*, *name*, *salary*, *dept_name*, *building*, *budget*)

这表示在*instructor*和*department*对应的关系上进行自然连接的结果。这似乎是个好主意，因为某些查询可以用更少的连接来表达，除非我们仔细考虑产生我们的E-R设计的大学的实际状况。

让我们考虑图8-2中*inst_dept*关系的实例。注意，我们对系里的每个教师都不得不重复一遍系信息("*building*"和"*budget*")。例如，关于计算机科学系的信息是(Taylor, 100000)，教师Katz、Srinivasan和Brandt的元组中均包含该信息。

所有这些元组的预算数额统一这一点很重要，否则我们的数据库将会不一致。在使用 *instructor* 和 *department* 的原始设计中，对每个预算数额存储一次。这说明使用 *inst_dept* 是一个坏主意，因为它重复存储预算数额，且要承担某些用户可能更新一条元组而不是所有元组中的预算数额，并因此产生不一致的风险。

即使我们决定容许冗余的问题，*inst_dept* 模式仍存在其他问题。假设我们在大学里创立一个新系。在上面的设计选择中，我们无法直接表示关于一个系的信息(*dept_name*, *building*, *budget*)，除非该系在学校中有至少一位教师。这是因为 *inst_dept* 表中的元组需要 *ID*、*name* 和 *salary* 的值。这意味着我们不能记录新成立的系的信息，直到该新系录用了第一位教师为止。在旧的设计中，模式 *department* 可以处理这个情况，但是在修改的设计中，我们将不得不创建一条 *building* 和 *budget* 为空值的元组。在一些情况下，空值会带来麻烦，正如我们在 SQL 的学习中所看到的。然而，如果我们认为这种情况对于我们不是问题的话，那么我们可以继续使用该修改的设计。

ID	name	salary	dept_name	building	budget
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

图 8-2 *inst_dept* 表

8.1.2 设计选择：更小的模式

再假定我们从 *inst_dept* 模式开始。我们将如何发现它需要信息重复，且应该分成 *instructor* 和 *department* 两个模式呢？

通过观察模式 *inst_dept* 上实际关系的内容，我们能够注意到为每位与系关联的教师都要列一遍办公楼和预算所导致的信息重复。但是，这是一个不可靠的处理方式。一个真实的数据库拥有大量模式以及数量甚至更多的属性。元组的数量可以为数百万或更多。探查重复将代价高昂。这个方法还存在一个甚至更根本的问题。它使我们无法确定没有重复是否仅仅是一个"幸运的"特别情况，或者它是否为一般规则的表现。在我们的例子中，我们将如何知道在我们的大学中每个系(由系名唯一标识)必须位于单个办公楼中并且必须具有单个预算数额呢？计算机科学系的预算数额出现二次且完全相同的情况仅是一个巧合吗？如果不回到企业本身并了解它的规则，我们就无法回答这些问题。特别是，我们将需要发现大学要求每个系(由系名唯一标识)必须只有一个办公楼和一个预算值。

324
325

在 *inst_dept* 的例子中，我们创建 E-R 设计的过程成功避免了这种模式的产生。但是，这个幸运的情况并不总出现。因此，我们需要让数据库设计者即使在 *dept_name* 不是所讨论模式的主码的情况下，也能定义如"*dept_name* 的每个特定的值对应至多一个 *budget*"这样的规则。换句话说，我们需要写这样一条规则"如果存在模式(*dept_name*, *budget*)，则 *dept_name* 可以作为主码"。这条规则被定义为**函数依赖**(functional dependency)

$$dept\_name \rightarrow budget$$

给定这样一条规则，我们现在就有足够的信息来发现 *inst_dept* 模式的问题了。由于 *dept_name* 不能是 *inst_dept* 的主码(因为一个系可能在模式 *inst_dept* 上的关系中需要多条元组)，因此预算数额可能会重复。

类似于这些的观察及由它们导出的规则(在此为函数依赖)让数据库设计者可以发现一个模式应该分成分解成两个或多个模式的情况。不难发现，分解 *inst_dept* 的正确方法是同原始设计一样分解成模式 *instructor* 和 *department*。对于具有大量属性和多个函数依赖的模式，找到正确的分解要难得多。为了

处理这种情况，我们将依靠本章后面所介绍的规范方法。

并不是所有的模式分解都是有益的。考虑我们拥有的所有模式都由一个属性构成这样一个极端的情况。任何类型的有意义的联系都无法表示。现在考虑一个不那么极端的情况，我们把 *employee* 模式（见 7.8 节）：

$$employee\ (ID,\ name,\ street,\ city,\ salary)$$

分解为以下两个模式：

$$employee1\ (ID,\ name)$$
$$employee2\ (name,\ street,\ city,\ salary)$$

这个分解的缺陷在于企业可能拥有两个同名的雇员。在实际中这不是不可能的，因为许多文化中都有某些非常流行的名字。当然，每个人都会有一个唯一的雇员号，这就是 *ID* 能够作为主码的原因。举例来说，让我们假定两个雇员，均名为 Kim，在该大学工作，并且在原始设计中的 *employee* 模式上的关系中有以下元组：

(57766, Kim, Main, Perryridge, 75000)
(98776, Kim, North, Hampton, 67000)

图 8-3 所示为这些元组、利用分解产生的模式所生成的元组，以及我们试图用自然连接重新生成原始元组所得到的结果。如我们在图 8-3 中看到的，那两个原始元组将能够两个新的元组出现于结果中，这两个新的元组将属于这两个名为 Kim 的职员的数据值错误地混合在一起。虽然我们拥有较多的元组，但是实际上从以下意义来看我们却有较少的信息。我们能够指出某个街道、城市和工资信息属于一个名为 Kim 的人，但是我们无法区分是哪一个 Kim。因此，我们的分解无法表达关于大学雇员的某些重要的事实。显然，我们想要避免这样的分解。我们将这样的分解称为**有损分解**（lossy decomposition），反之则称为**无损分解**（lossless decomposition）。

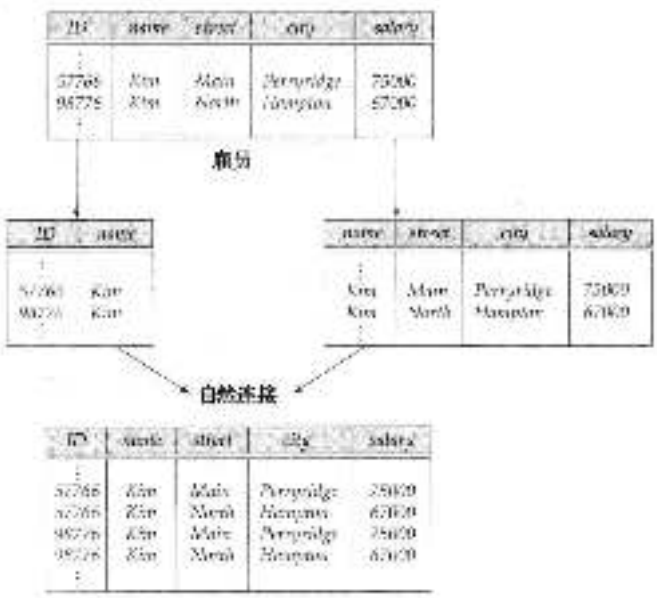


图 8-3 一个不好的分解导致的信息丢失

8.2 原子域和第一范式

E-R 模型允许实体集和联系集的属性具有某些程度的子结构。特别地，它允许像图 7-11 中的 *phone_number* 这样的多值属性以及组合属性（若如具有子属性 *street*、*city*、*state* 以及 *zip* 的属性 *address*）。当从包含这些类型的属性的 E-R 设计创建表时，要消除这种子结构。对于组合属性，让每个子属性本身成为一个属性。对于多值属性，为多值集合中的每个项创建一条元组。

326
327

在关系模型中，我们将属性不具有任何子结构这个思想形式化。一个域是**原子的**(atomic)，如果该域的元素被认为是不可分的单元。我们称一个关系模式 R 属于**第一范式**(First Normal Form，1NF)，如果 R 的所有属性的域都是原子的。

名字的集合是一个非原子值的例子。例如，如果关系 *employee* 的模式包含一个属性 *children*，它的域元素是名字的集合，该模式就不属于第一范式。

组合属性，比如包含了属性 *street*、*city*、*state* 和 *zip* 的属性 *address*，也具有非原子域。

假定整数是原子的，那么整数的集合是一个原子域；然而，所有整数集的集合是一个非原子域。区别在于，我们一般不认为整数具有子部分，但是我们认为整数的集合具有子部分——构成该集合的那些整数。不过，重要的问题不是域本身是什么，而是在数据库中如何使用域元素。如果我们认为每个整数是一列有序的数字，那么全部整数的域就是非原子的。

作为上述观点的实际例证，考虑一个机构，它给雇员分配下述样式的标识号：前两个字母表示系，剩下的四位数字是雇员在该系内的唯一号码。例如这样的号码可以是"CS0012"和"EE1127"。这样的标识号可以分成更小的单元，因此是非原子的。如果一个关系模式的一个属性的域是由如上编码的标识号所组成，则该模式不属于第一范式。

当采用这种标识号时，雇员所属的系可以通过编写解析标识号结构的代码得到。这么做需要额外的编程，而且信息是在应用程序中而不是在数据库中编码。如果这种标识号用作主码，还会产生进一步的问题：当一个雇员换了系时，该雇员的标识号在它所出现的每个地方都必须修改(这会是一个困难的任务)，否则解释该号码的代码将会给出错误的结果。

根据以上讨论，我们可能会使用形如"CS-101"的课程标识号，其中"CS"表示计算机科学系。这意味着课程标识号的域不是原子的。使用该系统的人认为这样的域不是原子的。然而，只要数据库应用没有将标识号拆开并将标识号的一部分解析为系的缩写，它仍然将该域视为原子的。*course* 模式将系名
328 作为一个单独的属性存储，数据库应用就可以根据这个属性值找到课程所属系，而不需要解析课程标识号中特定的字母。因此，我们的大学模式可以被认为属于第一范式。

使用以集合为值的属性会导致冗余存储数据的设计，进而会导致不一致。比如，数据库设计者可能不将教师和课程安排之间的联系表示为一个单独的关系 *teaches*，而是尝试为每一个教师存储一个课程段标识号的集合，并为每一个课程段存储一个教师标识号的集合。(*section* 和 *instructor* 的主码用作标识号。)每当关于哪位教师讲授哪个课程段的数据变化时，必须在两个地方执行更新：课程段的教师集合，以及教师的课程段集合。对两个更新的执行失败，会导致数据库处于不一致的状态。只保留其中一个集合，即要么课程段的教师集合，要么教师的课程段集合，将避免重复的信息；然而，只保留其中一个集合将使某些查询变得复杂，并且也不好确定保留哪一个。

有些类型的非原子值使用的时候要小心，但是它们可能很有用。例如，组合值属性常常很有用，而且很多情况下以集合为值的属性也是有用的，所以在 E-R 模型里这两种值都是支持的。在许多含有复杂结构的实体域中，强制使用第一范式会给应用程序员造成不必要的负担，他必须编写代码把数据转换成原子形式。从原子形态来回转换数据也会有运行时的额外开销。所以在这样的域里支持非原子的值是很有用的。事实上，如我们将在第 22 章看到的，现代数据库系统确实支持很多类型的非原子值。然而，本章我们限定只讨论属于第一范式的关系，因此，所有的域都是原子的。

8.3 使用函数依赖进行分解

在 8.1 节中，我们知道存在一个规范方法判断一个关系模式是否应该分解。这个方法基于码和函数依赖的概念。

在讨论关系数据库设计的算法时，我们需要针对任意的关系及其模式讨论，而不只是讨论例子。回想第 2 章对关系模型的介绍，我们在这里对我们的表示法进行概述。

- 一般情况下，我们用希腊字母表示属性集(例如 α)。我们用一个小写的罗马字母后面跟一个用一对圆括号括住的大写字母来指关系模式(例如 $r(R)$)。我们用表示法 $r(R)$ 表示该模式是关系 r 的，R 表示属性集。不过当我们不关心关系的名字时经常简化我们的表示法而只用 R。

当然，一个关系模式是一个属性集，但是并非所有的属性集都是模式。当使用一个小写的希腊字母时，我们是指一个有可能是模式也可能不是模式的属性集。当我们希望指明属性集一定为一个模式时，就使用罗马字母。 [329]

- 当属性集是一个超码时，我们用 K 表示它。超码属于特殊的关系模式，因此我们使用术语"K 是 $r(R)$ 的超码"。
- 我们对关系使用小写的名字。在我们的例子中，这些名字是企图有实际含义的（例如 *instructor*），而在我们的定义和算法中，我们使用单个字母，比如 r。
- 当然，一个关系在任意给定时间都有特定的值；我们将那看作一个实例并使用术语"r 的实例"。当我们明显在讨论一个实例时，我们可以仅用关系的名字（例如 r）。

8.3.1　码和函数依赖

一个数据库对现实世界中的一组实体和联系建模。在现实世界中，数据上通常存在各种约束（规则）。例如，在一个大学数据库中预计要保证的一些约束有：

- 学生和教师通过他们的 ID 唯一标识。
- 每个学生和教师只有一个名字。
- 每个教师和学生只（主要）关联一个系。[1]
- 每个系只有一个预算值，且只有一个关联的办公楼。

一个关系的满足所有这种现实世界约束的实例，称为关系的**合法实例**（legal instance）；在一个数据库的合法实例中所有关系实例都是合法实例。

几种最常用的现实世界约束可以形式化地表示为码（超码、候选码以及主码），或者下面所定义的函数依赖。

在 2.3 节中，曾定义超码的概念为可以唯一标识关系中一条元组的一个或多个属性的集合。在这里重新表述该定义如下：令 $r(R)$ 是一个关系模式。R 的子集 K 是 $r(R)$ 的**超码**（superkey）的条件是：在关系 $r(R)$ 的任意合法实例中，对于 r 的实例中的所有元组对 t_1 和 t_2 总满足，若 $t_1 \neq t_2$，则 $t_1[K] \neq t_2[K]$。也就是说，在关系 $r(R)$ 的任意合法实例中没有两条元组在属性集 K 上可能具有相同的值。显然，如果 r 中没有两条元组在 K 上具有相同的值，那么在 r 中一个 K 值唯一标识一条元组。 [330]

鉴于超码是能够唯一标识整条元组的属性集，函数依赖让我们可以表达唯一标识某些属性的值的约束。考虑一个关系模式 $r(R)$，令 $\alpha \subseteq R$ 且 $\beta \subseteq R$。

- 给定 $r(R)$ 的一个实例，我们说这个实例**满足**（satisfy）**函数依赖** $\alpha \rightarrow \beta$ 的条件是：对实例中所有元组对 t_1 和 t_2，若 $t_1[\alpha] = t_2[\alpha]$，则 $t_1[\beta] = t_2[\beta]$。
- 如果在 $r(R)$ 的每个合法实例中都满足函数依赖 $\alpha \rightarrow \beta$，则我们说该函数依赖在模式 $r(R)$ 上成立（hold）。

使用函数依赖这一概念，我们说如果函数依赖 $K \rightarrow R$ 在 $r(R)$ 上成立，则 K 是 $r(R)$ 的一个超码。也就是说，如果对于 $r(R)$ 的每个合法实例，对于实例中每个元组对 t_1 和 t_2，凡是 $t_1[K] = t_2[K]$，总有 $t_1[R] = t_2[R]$（即 $t_1 = t_2$），则 K 是一个超码。[2]

函数依赖使我们可以表示不能用超码表示的约束。在 8.1.2 节中我们曾考虑模式：

inst_dept (*ID*, *name*, *salary*, *dept_name*, *building*, *budget*)

在该模式中函数依赖 *dept_name* → *budget* 成立，因为对于每个系（由 *dept_name* 唯一标识）都存在唯一的预算数额。

属性对（*ID*, *dept_name*）构成 *inst_dept* 的一个超码，我们将这一事实记做：

---

[1] 一个教师或一个学生可以和多个系相关联，例如兼职教师或者辅修系。我们简化的大学模式只对每个教师或学生用关联的主系建模。一个实际的大学模式会在另外的关系中表示次要的关联。

[2] 注意，我们在这里假设关系为集合。SQL 处理多集，并且 SQL 中声明一个属性集 K 为主码不仅需要当 $t_1[K] = t_2[K]$ 时 $t_1 = t_2$，还需要没有重复的元组。SQL 还要求 K 集中的属性不能赋予空值。

$$ID, dept\_name \rightarrow name, salary, building, budget$$

我们将以两种方式使用函数依赖：

- 判定关系的实例是否满足给定函数依赖集 F。
- 说明合法关系集上的约束。因此，我们将只关心满足给定函数依赖集的那些关系实例。如果我们希望只考虑模式 R 上满足函数依赖集 F 的关系，我们说 F 在 $r(R)$ 上成立。

让我们考虑图 8-4 中的关系 r 的实例，看看它满足什么函数依赖。注意到 $A \rightarrow C$ 是满足的。存在两条元组的 A 值为 a_1。这些元组具有相同的 C 值，c_1。类似地，A 值为 a_2 的两条元组具有相同的 C 值，c_2。没有其他元组对具有相同的 A 值。但是，函数依赖 $C \rightarrow A$ 是不满足的。为了说明这一点，考虑元组 $t_1 = (a_1, b_1, c_1, d_1)$ 和元组 $t_2 = (a_2, b_2, c_1, d_2)$。这两条元组具有相同的 C 值，c_1，但它们具有不同的 A 值，分别为 a_1 和 a_2。因此，我们找到一对元组 t_1 和 t_2，使得 $t_1[C] = t_2[C]$，但 $t_1[A] \neq t_2[A]$。

A	B	C	D
a_1	b_1	c_1	d_1
a_2	b_2	c_1	d_2
a_1	b_2	c_2	d_2
a_2	b_3	c_2	d_3
a_3	b_1	c_1	d_4

图 8-4 关系 r 的实例的例子

有些函数依赖称为平凡的(trivial)，因为它们在所有关系中都满足。例如，$A \rightarrow A$ 在所有包含属性 A 的关系中满足。从字面上看函数依赖的定义，我们知道，对所有满足 $t_1[A] = t_2[A]$ 的元组 t_1 和 t_2，有 $t_1[A] = t_2[A]$。同样，$AB \rightarrow A$ 也在所有包含属性 A 的关系中满足。一般地，如果 $\beta \subseteq \alpha$，则形如 $\alpha \rightarrow \beta$ 的函数依赖是平凡的。

重要的是，要认识到一个关系实例可能满足某些函数依赖，它们并不需要在关系的模式上成立。在图 8-5 的 *classroom* 关系的实例中，我们发现 *room_number* → *capacity* 是满足的。但是，我们相信，在现实世界中，不同建筑里的两个教室可以具有相同的房间号，但具有不同的空间大小。因此，有时有可能存在一个 *classroom* 关系的实例，不满足 *room_number* → *capacity*。所以，我们不将 *room_number* → *capacity* 包含于 *classroom* 关系的模式上成立的函数依赖集中。然而，我们会期望函数依赖 *building*, *room_number* → *capacity* 在 *classroom* 模式上成立。

building	*room_number*	*capacity*
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

图 8-5 *classroom* 关系的实例

给定关系 $r(R)$ 上成立的函数依赖集 F，有可能会推断出某些其他的函数依赖也一定在该关系上成立。例如，给定模式 $r(A, B, C)$，如果函数依赖 $A \rightarrow B$ 和 $B \rightarrow C$ 在 r 上成立，可以推出函数依赖 $A \rightarrow C$ 也一定在 r 上成立。这是因为，给定 A 的任意值，仅存在 B 的一个对应值，且对于 B 的那个值，只能存在 C 的一个对应值。我们稍后将在 8.4.1 节学习如何进行这种推导。

331
↓
332

我们将使用 F^+ 符号来表示 F 集合的闭包(closure)。也就是能够从给定 F 集合推导出的所有函数依赖的集合。显然，F^+ 包含 F 中所有的函数依赖。

8.3.2 Boyce-Codd 范式

我们能达到的较满意的范式之一是 Boyce-Codd 范式(Boyce-Codd Normal Form，BCNF)。它消除所有基于函数依赖能够发现的冗余，虽然，如我们将在 8.6 节中看到的，可能有其他类型的冗余还保留着。具有函数依赖集 F 的关系模式 R 属于 BCNF 的条件是，对 F^+ 中所有形如 $\alpha \rightarrow \beta$ 的函数依赖(其中 $\alpha \subseteq R$ 且 $\beta \subseteq R$)，下面至少有一项成立：

- $\alpha \rightarrow \beta$ 是平凡的函数依赖(即，$\beta \subseteq \alpha$)。
- α 是模式 R 的一个超码。

一个数据库设计属于 BCNF 的条件是，构成该设计的关系模式集中的每个模式都属于 BCNF。

我们已经在 8.1 节中见过了不属于 BCNF 的关系模式的例子：

$$inst\_dept\ (ID, name, salary, dept\_name, building, budget)$$

函数依赖 *dept_name* → *budget* 在 *inst_dept* 上成立，但是 *dept_name* 并不是超码(因为一个系可以有多个不同的教师)。在 8.1.2 节中，我们看到把 *inst_dept* 分解为 *instructor* 和 *department* 是一个更好的设计。模式 *instructor* 属于 BCNF。所有成立的非平凡的函数依赖，例如：

$$ID \rightarrow name, dept\_name, salary$$

在箭头的左侧包含 *ID*，且 *ID* 是 *instructor* 的一个超码(事实上，在这个例子中，是主码)。(也就是说，在不包含 *ID* 的另一侧上，对于 *name*、*dept_name* 和 *salary* 的任意组合不存在非平凡的函数依赖。)因此，*instructor* 属于 BCNF。

类似地，*department* 模式属于 BCNF，因为所有成立的非平凡函数依赖，例如：

$$dept\_name \rightarrow building, budget$$

333

在箭头的左侧包含 *dept_name*，且 *dept_name* 是 *department* 的一个超码(和主码)。因此，*department* 属于 BCNF。

我们现在讲述分解不属于 BCNF 的模式的一般规则。设 R 为不属于 BCNF 的一个模式。则存在至少一个非平凡的函数依赖 $\alpha \rightarrow \beta$，其中 α 不是 R 的超码。我们在设计中用以下两个模式取代 R：

- $(\alpha \cup \beta)$
- $(R - (\beta - \alpha))$

在上面的 *inst_dept* 例子中，α = *dept_name*，β = {*building*, *budget*}，且 *inst_dept* 被取代为：

- $(\alpha \cup \beta)$ = (*dept_name*, *building*, *budget*)
- $(R - (\beta - \alpha))$ = (*ID*, *name*, *dept_name*, *salary*)

在这个例子中，结果是 $\beta - \alpha = \beta$。我们需要像上述那样的表述规则，从而正确处理箭头两边都出现的属性的函数依赖。技术上的原因我们会在 8.5.1 节介绍。

当我们分解不属于 BCNF 的模式时，产生的模式中可能有一个或多个不属于 BCNF。在这种情况中，需要进一步分解，其最终结果是一个 BCNF 模式集合。

8.3.3 BCNF 和保持依赖

我们已经看到多种表达数据库一致性约束的方式：主码约束、函数依赖、check 约束、断言和触发器。在每次数据库更新时检查这些约束的开销很大，因此，将数据库设计成能够高效地检查约束是很有用的。特别地，如果函数依赖的检验仅需要考虑一个关系就可以完成，那么检查这种约束的开销就很低。我们将看到，在有些情况下，到 BCNF 的分解会妨碍对某些函数依赖的高效检查。

对此举例，假定我们对我们的大学机构做一个小的改动。在图 7-15 的设计中，一位学生只能有一位导师。这是由从 *student* 到 *advisor* 的联系集 *advisor* 为多对一而推断出的。我们要做的"小"改动是一个教师只能和单个系关联，且一个学生可以有多个导师，但是一个给定的系中至多一位。[⊖]

利用 E-R 设计实现这个改动的一种方法是，把联系集 *advisor* 替换为涉及实体集 *instructor*、*student* 和 *department* 的二元联系集 *dept_advisor*，它是从{*instructor*, *student*}对到 *department* 多对一的，如图8-6所示。该 E-R 图指明了"一个学生可以有多位导师，但是对应于一个给定的系最多只有一个"的约束。

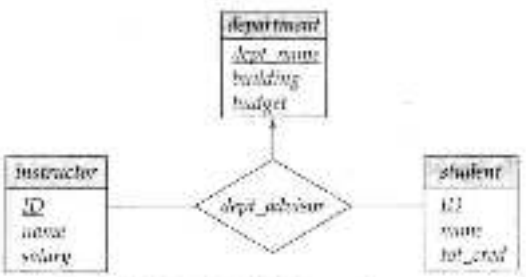


图 8-6 联系集 *dept_advisor*

对应于新的 E-R 图，*instructor*、*department* 和 *student* 的模式没有变。然而，从 *dept_advisor* 导出的模式现在为：

$$dept\_advisor\ (s\_ID, i\_ID, dept\_name)$$

虽然没有在 E-R 图中指明，不过假定我们有附加的约束"一位教师只能在一个系担任导师。"

⊖ 这样的约束对于双专业的学生显然是没意义的。

那么，下面的函数依赖在 *dept_advisor* 上成立：

$$i\_ID \rightarrow dept\_name$$
$$s\_ID, dept\_name \rightarrow i\_ID$$

第一个函数依赖产生于我们的需求"一位教师只能在一个系担任导师"。第二个函数依赖产生于我们的需求"对于一个给定的系，一个学生可以有至多一位导师"。

注意，在这种设计中，每次一名教师参与一个 *dept_advisor* 联系的时候，我们都不得不重复一次系的名称。我们看到 *dept_advisor* 不属于 BCNF，因为 *i_ID* 不是超码。根据 BCNF 分解规则，得到：

$$(s\_ID, i\_ID)$$
$$(i\_ID, dept\_name)$$

334 ~ 335 上面的两个模式都属于 BCNF。（事实上，你可以验证，按照定义任何只包含两个属性的模式都属于 BCNF。）然而注意，在 BCNF 设计中，没有一个模式包含函数依赖 $s\_ID, dept\_name \rightarrow i\_ID$ 中出现的所有属性。

由于我们的设计使得该函数依赖的强制实施在计算上很困难，因此我们称我们的设计不是**保持依赖的**（dependency preserving）。[三]由于常常希望保持依赖，因此我们考虑另外一种比 BCNF 弱的范式，它允许我们保持依赖。该范式称为第三范式。[四]

8.3.4 第三范式

BCNF 要求所有非平凡函数依赖都形如 $\alpha \rightarrow \beta$，其中 α 为一个超码。第三范式（3NF）稍微放宽了这个约束。它允许左侧不是超码的某些非平凡函数依赖。在定义 3NF 之前，我们回想到候选码是最小的超码——任何真子集都不是超码的超码。

具有函数依赖集 F 的关系模式 R 属于第三范式（third normal form）的条件是：对于 F^+ 中所有形如 $\alpha \rightarrow \beta$ 的函数依赖（其中 $\alpha \subseteq R$ 且 $\beta \subseteq R$），以下至少一项成立：

- $\alpha \rightarrow \beta$ 是一个平凡的函数依赖。
- α 是 R 的一个超码。
- $\beta - \alpha$ 中的每个属性 A 都包含于 R 的一个候选码中。

注意上面的第三个条件并没有说单个候选码必须包含 $\beta - \alpha$ 中的所有属性；$\beta - \alpha$ 中的每个属性 A 可能包含于不同的候选码中。

前两个条件与 BCNF 定义中的两个条件相同。3NF 定义中的第三个条件看起来很不直观，并且它的用途也不是显而易见的。在某种意义上，它代表 BCNF 条件的最小放宽。以确保每一个模式都有保持依赖的 3NF 分解。它的用途在后面介绍 3NF 分解时会变得更清楚。

注意任何满足 BCNF 的模式也满足 3NF，因为它的每个函数依赖都将满足前两个条件中的一条。所以 BCNF 是比 3NF 更严格的范式。

3NF 的定义允许某些 BCNF 中不允许的函数依赖。只满足 3NF 定义中第三个条件的依赖 $\alpha \rightarrow \beta$ 在 BCNF 中是不允许的，但在 3NF 中是允许的。[五]

336 现在我们再次考虑联系集 *dept_advisor*。它具有以下函数依赖：

$$i\_ID \rightarrow dept\_name$$
$$s\_ID, dept\_name \rightarrow i\_ID$$

在 8.3.3 节中我们说函数依赖"$i\_ID \rightarrow dept\_name$"导致 *dept_advisor* 模式不属于 BCNF。注意这里 $\alpha = i\_ID$，$\beta = dept\_name$，并且 $\beta - \alpha = dept\_name$。由于函数依赖 $s\_ID, dept\_name \rightarrow i\_ID$ 在 *dept_advisor* 上成立，于是属性 *dept_name* 包含于一个候选码中。因此 *dept_advisor* 属于 3NF。

[三] 从技术上来说，由于存在逻辑上蕴涵该依赖的其他依赖，因此属性在打开一个模式中都不会出现的依赖也被含蓄地强制实施了。稍后我们将在 8.4.5 节讨论这个问题。

[四] 你可能注意到我们跳过了第二范式。第二范式只有历史意义，已经不在实际中使用了。

[五] 这些依赖是**传递依赖**（transitive dependences）的例子（参见实践习题 8.16）。3NF 的原始定义最初由传递依赖而来的，我们所使用的定义与之等价但更容易理解。

我们已经看到，当不存在保持依赖的 BCNF 设计时，必须在 BCNF 和 3NF 之间进行权衡。这些权衡将在 8.5.4 节详细讨论。

8.3.5 更高的范式

在某些情况中，使用函数依赖分解模式可能不足以避免不必要的信息重复。考虑在 *instructor* 实体集定义中的小变化。我们为每个教师记录一组孩子名字以及一组电话号码。电话号码可以被多个人共享。因此，*phone_number* 和 *child_name* 将是多值属性，并且根据我们从 E-R 设计生成模式的规则，我们会有两个模式，多值属性 *phone_number* 和 *child_name* 中每个属性对应一个模式：

$$(ID, child\_name)$$
$$(ID, phone\_number)$$

如果我们合并这些模式而得到

$$(ID, child\_name, phone\_number)$$

我们会发现该结果属于 BCNF，因为只有平凡的函数依赖成立。因此我们可能认为这样的合并是个好主意。然而，通过考虑有两个孩子和两个电话号码的教师的例子，我们会发现这样的合并是一个坏主意。例如，令 *ID* 为 99999 的教师有两个孩子，叫作"David"和"William"，以及有两个电话号码，512－555－1234 和 512－555－4321。在合并的模式中，我们必须为每个家属重复一次电话号码：

(99999, David, 512－555－1234)
(99999, David, 512－555－4321)
(99999, William, 512－555－1234)
(99999, William, 512－555－4321)

如果我们不重复电话号码，且只存储第一条和最后一条元组，我们就记录了家属名字和电话号码，但是结果元组将暗指 David 对应于 512－555－1234，而 William 对应于 512－555－4321。我们知道，这 337
是不正确的。

由于基于函数依赖的范式并不足以处理这样的情况，因此定义了另外的依赖和范式。我们在 8.6 和 8.7 节对此进行讲述。

8.4 函数依赖理论

在例子中我们已经看到，作为检查模式是否属于 BCNF 或 3NF 这一过程的一部分，能够对函数依赖进行系统地推理是很有用的。

8.4.1 函数依赖集的闭包

我们将会看到，给定模式上的函数依赖集 F，我们可以证明某些其他的函数依赖在模式上也成立。我们称这些函数依赖被 F"**逻辑蕴涵**"。当检验范式时，只考虑给定的函数依赖集是不够的；除此以外，还需要考虑模式上成立的所有函数依赖。

更正式地，给定关系模式 $r(R)$，如果 $r(R)$ 的每一个满足 F 的实例也满足 f，则 R 上的函数依赖 f 被 r 上的函数依赖集 F **逻辑蕴涵**(logically imply)。

假设我们给定关系模式 $r(A, B, C, G, H, I)$ 及函数依赖集：

$$A \rightarrow B$$
$$A \rightarrow C$$
$$CG \rightarrow H$$
$$CG \rightarrow I$$
$$B \rightarrow H$$

那么函数依赖

$$A \rightarrow H$$

被逻辑蕴涵。也就是说，我们可以证明，一个关系只要满足给定的函数依赖集，这个关系也一定满足 $A \rightarrow H$。假设元组 t_1 及 t_2 满足

$$t_1[A] = t_2[A]$$

由于已知 $A \rightarrow B$，因此由函数依赖的定义推出

338

$$t_1[B] = t_2[B]$$

那么，又由于已知 $B \rightarrow H$，因此由函数依赖的定义推出

$$t_1[H] = t_2[H]$$

因此，我们证明了，对任意两个元组 t_1 及 t_2，只要 $t_1[A] = t_2[A]$，就一定有 $t_1[H] = t_2[H]$。而这正是 $A \rightarrow H$ 的定义。

令 F 为一个函数依赖集。F 的闭包是被 F 逻辑蕴涵的所有函数依赖的集合，记作 F^+。给定 F，可以由函数依赖的形式化定义直接计算出 F^+。如果 F 很大，则这个过程将会很长而且很难。这一 F^+ 运算需要论证，就像刚才用于证明 $A \rightarrow H$ 在依赖集例子的闭包中的那种论证。

公理(axiom)，或推理规则，提供了一种用于推理函数依赖的更为简单的技术。在下面的规则中，我们用希腊字母(α，β，γ，…)表示属性集，用字母表中从开头起的大写罗马字母表示单个属性。我们用 $\alpha\beta$ 表示 $\alpha \cup \beta$。

我们可以使用以下三条规则去寻找逻辑蕴涵的函数依赖。通过反复应用这些规则，可以找出给定 F 的全部 F^+。这组规则称为 Armstrong 公理(Armstrong's axiom)，以纪念首次提出这一公理的人。

- **自反律**(reflexivity rule)。若 α 为一属性集且 $\beta \subseteq \alpha$，则 $\alpha \rightarrow \beta$ 成立。
- **增补律**(augmentation rule)。若 $\alpha \rightarrow \beta$ 成立且 γ 为一属性集，则 $\gamma\alpha \rightarrow \gamma\beta$ 成立。
- **传递律**(transitivity rule)。若 $\alpha \rightarrow \beta$ 和 $\beta \rightarrow \gamma$ 成立，则 $\alpha \rightarrow \gamma$ 成立。

Armstrong 公理是**正确有效的**(sound)，因为它们不产生任何错误的函数依赖。这些规则是**完备的**(complete)，因为，对于给定函数依赖集 F，它们能产生全部 F^+。文献注解提供了对正确有效性和完备性的证明的参考。

虽然 Armstrong 公理是完备的，但是直接用它们计算 F^+ 会很麻烦。为进一步简化，我们列出另外的一些规则。可以用 Armstrong 公理证明这些规则是正确有效的(参见实践习题 8.4、8.5 及习题 8.26)。

- **合并律**(union rule)。若 $\alpha \rightarrow \beta$ 和 $\alpha \rightarrow \gamma$ 成立，则 $\alpha \rightarrow \beta\gamma$ 成立。
- **分解律**(decomposition)。若 $\alpha \rightarrow \beta\gamma$ 成立，则 $\alpha \rightarrow \beta$ 和 $\alpha \rightarrow \gamma$ 成立。
- **伪传递律**(pseudotransitivity rule)。若 $\alpha \rightarrow \beta$ 和 $\gamma\beta \rightarrow \delta$ 成立，则 $\alpha\gamma \rightarrow \delta$ 成立。

339 让我们将规则应用于模式 $R = (A, B, C, G, H, I)$ 及函数依赖集 F：$A \rightarrow B$，$A \rightarrow C$，$CG \rightarrow H$，$CG \rightarrow I$，$B \rightarrow H$。这里列出 F^+ 中的几个依赖：

- $A \rightarrow H$。由于 $A \rightarrow B$ 和 $B \rightarrow H$ 成立，因此使用传递律。可以看到使用 Armstrong 公理证明 $A \rightarrow H$ 成立比本节前面直接使用定义论证要简单得多。
- $CG \rightarrow HI$。由于 $CG \rightarrow H$ 和 $CG \rightarrow I$ 成立，因此由合并律推出 $CG \rightarrow HI$。
- $AG \rightarrow I$。由于有 $A \rightarrow C$ 且 $CG \rightarrow I$，因此由伪传递律推出 $AG \rightarrow I$ 成立。

$AG \rightarrow I$ 的另一种推理方法如下：我们在 $A \rightarrow C$ 上使用增补律从而推出 $AG \rightarrow CG$。在这个依赖和 $CG \rightarrow I$ 上使用传递率，我们推出 $AG \rightarrow I$。

图 8-7 给出形式化地示范如何使用 Armstrong 公理计算 F^+ 的过程。在这个过程中，当一个函数依赖加入 F^+ 时，它可能已经存在了，这时 F^+ 没有变化。我们将在 8.4.2 节看到另一种计算 F^+ 的算法。

```
F+ = F
repeat
    for each F+ 中的函数依赖 f
        在 f 上应用自反律和增补律
        将结果加入到 F+ 中
    for each F+ 中的一对函数依赖 f1 和 f2
        if f1 和 f2 可以使用传递律结合起来
            将结果加入到 F+ 中
until F+ 不再发生变化
```

图 8-7 计算 F^+ 的过程

函数依赖的左边和右边都是 R 的子集。由于包含 n 个元素的集合有 2^n 个子集，因此共有 $2^n \times 2^n = 2^{2n}$ 个可能的函数依赖，其中 n 是 R 中的属性个数。除最后一次迭代外，每一次执行 repeat 循环都至少往 F^+ 里加入一个函数依赖；因此，该过程保证可以终止。

8.4.2　属性集的闭包

如果 $\alpha \rightarrow B$，我们称属性 B 被 α **函数确定**(functionally determine)。要判断集合 α 是否为超码，我们必须设计一个算法，用于计算被 α 函数确定的属性集。一种方法是计算 F^+，找出所有左半部为 α 的函数依赖，并合并这些函数依赖的右半部。但是这么做开销很大，因为 F^+ 可能很大。

在本节后面我们将看到，一个用于计算被 α 函数确定的属性集的高效算法不仅可以用来判断 α 是否为超码，还可以用于其他的一些任务。

令 α 为一个属性集。我们将函数依赖集 F 下被 α 函数确定的所有属性的集合称为 F 下 α 的闭包，记为 α^+。图 8-8 是以伪码写的计算 α^+ 的算法。输入是函数依赖集 F 和属性集 α。输出存储在变量 *result* 中。

```
result := α;
    repeat
        for each 函数依赖 β→γ in F do
            begin
                if β⊆ result then result := result ∪ γ;
            end
    until (result 不变)
```

图 8-8　计算 F 下 α 的闭包 α^+ 的算法

为解释该算法如何进行，我们将用它计算 8.4.1 节中定义的函数依赖集下的 $(AG)^+$。开始时 $result = AG$。在第一次执行 **repeat** 循环测试各个函数依赖时，我们发现

- 由 $A \rightarrow B$，于是将 B 加入 *result*。这是因为，我们观察到 $A \rightarrow B$ 属于 F，$A \subseteq result$(即 AG)。所以 $result := result \cup B$。
- 由 $A \rightarrow C$，*result* 变为 $ABCG$。
- 由 $CG \rightarrow H$，*result* 变为 $ABCGH$。
- 由 $CG \rightarrow I$，*result* 变为 $ABCGHI$。

我们第二次执行 **repeat** 循环时，没有新属性加入 *result*，算法终止。

让我们看看图 8-8 的算法为什么正确。第一步正确，因为 $\alpha \rightarrow \alpha$ 总是成立(由自反律)。我们说，对 *result* 的任意子集 β，有 $\alpha \rightarrow \beta$。由于开始 **repeat** 循环时 $\alpha \rightarrow result$ 为真。因此只要 $\beta \subseteq result$ 且 $\beta \rightarrow \gamma$，就可将 γ 加入 *result* 中。而由自反律得到 $result \rightarrow \beta$，故由传递律得到 $\alpha \rightarrow \beta$。再应用传递律就得到 $\alpha \rightarrow \gamma$(由 $\alpha \rightarrow \beta$ 及 $\beta \rightarrow \gamma$)。由合并律可推出 $\alpha \rightarrow result \cup \gamma$，所以 α 函数确定 **repeat** 循环中产生的任意新结果。也就是说，该算法所返回的任意属性都属于 α^+。

容易证明，该算法找出 α^+ 的全部属性。在执行当中，如果存在属性属于 α^+ 却还不属于 *result*，则必定存在函数依赖 $\beta \rightarrow \gamma$(其中 $\beta \subseteq result$)并且 γ 中至少有一个属性不在 *result* 中。当该算法结束时，所有的函数依赖都已经处理过，γ 中的属性都已经加到 *result* 中；因此我们可以确定 α^+ 中的所有属性都在 *result* 中。

340
|
341

经证明，在最坏情况下，该算法的执行时间为 F 集合规模的二次方。有一个更快的算法(但稍微复杂一点儿)，执行时间与 F 的规模呈线性关系；这个算法作为实践习题 8.8 的一部分进行介绍。

属性闭包算法有多种用途：

- 为了判断 α 是否为超码，我们计算 α^+，检查 α^+ 是否包含 R 中的所有属性。
- 通过检查是否 $\beta \subseteq \alpha^+$，我们可以检查函数依赖 $\alpha \rightarrow \beta$ 是否成立(或换句话说，是否属于 F^+)。也就是说，我们用属性闭包计算 α^+，看它是否包含 β。本章后面我们将会看到这种检查非常有用。
- 该算法给了我们另一种计算 F^+ 的方法：对任意的 $\gamma \subseteq R$，我们找出闭包 γ^+；对任意的 $S \subseteq \gamma^+$，我们输出一个函数依赖 $\gamma \rightarrow S$。

8.4.3　正则覆盖

假设我们在一个关系模式上有一个函数依赖集 F。每当用户在该关系上执行更新时，数据库系统必须确保此更新不破坏任何函数依赖，也就是说，F 中的所有函数依赖在新数据库状态下仍然满足。

如果更新操作破坏了 F 上的任一个函数依赖，系统必须回滚该更新操作。

我们可以通过测试与给定函数依赖集具有相同闭包的简化集的方式来减小检测冲突的开销。因为简化集和原集具有相同的闭包，所以满足函数依赖简化集的数据库也一定满足原集，反之亦然。但是，简化集更便于检测。稍后我们将看到简化集是如何构造的。首先，我们需要一些定义。

如果去除函数依赖中的一个属性不改变该函数依赖集的闭包，则称该属性是**无关的**(extraneous)。无关属性(extraneous attribute)的形式化定义如下：考虑函数依赖集 F 及 F 中的函数依赖 $\alpha \rightarrow \beta$。

- 如果 $A \in \alpha$ 并且 F 逻辑蕴含 $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$，则属性 A 在 α 中是无关的。
- 如果 $A \in \beta$ 并且函数依赖集 $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ 逻辑蕴含 F，则属性 A 在 β 中是无关的。

例如，假定我们在 F 中有函数依赖 $AB \rightarrow C$ 和 $A \rightarrow C$，那么 B 在 $AB \rightarrow C$ 中是无关的。再比如，假定我们在 F 中有函数依赖 $AB \rightarrow CD$ 和 $A \rightarrow C$，那么 C 在 $AB \rightarrow CD$ 的右半部中是无关的。

使用无关属性的定义时注意蕴涵的方向：如果将左半部与右半部交换，蕴含关系将总是成立的。也就是说，$(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$ 总是逻辑蕴涵 F，同时 F 也总是逻辑蕴涵 $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$。

下面介绍如何有效地验证一个属性是否无关。令 R 为一关系模式，且 F 是在 R 上成立的给定函数依赖集。考虑依赖 $\alpha \rightarrow \beta$ 中的一个属性 A。

- 如果 $A \in \beta$，为了检验 A 是否是无关的，考虑集合
$$F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$$
并检验 $\alpha \rightarrow A$ 是否能够由 F' 推出。为此，计算 F' 下的 α^+（α 的闭包）；如果 α^+ 包含 A，则 A 在 β 中是无关的。
- 如果 $A \in \alpha$，为了检验 A 是否是无关的，令 $\gamma = \alpha - \{A\}$，并且检查 $\gamma \rightarrow \beta$ 是否可以由 F 推出。为此，计算在 F 下的 γ^+（γ 的闭包）；如果 γ^+ 包含 β 中的所有属性，则 A 在 α 中是无关的。

例如，假定 F 包含 $AB \rightarrow CD$、$A \rightarrow E$ 和 $E \rightarrow C$。为检验 C 在 $AB \rightarrow CD$ 中是否是无关的，我们计算 $\{AB \rightarrow D, A \rightarrow E, E \rightarrow C\}$ 下 AB 的属性闭包。闭包为 $ABCDE$，包含 CD，所以我们推断出 C 是无关的。

F 的**正则覆盖**(canonical cover) F_c 是一个依赖集，使得 F 逻辑蕴含 F_c 中的所有依赖，并且 F_c 逻辑蕴涵 F 中的所有依赖。此外，F_c 必须具有如下性质：

- F_c 中任何函数依赖都不含无关属性。
- F_c 中函数依赖的左半部都是唯一的。即，F_c 中不存在两个依赖 $\alpha_1 \rightarrow \beta_1$ 和 $\alpha_2 \rightarrow \beta_2$，满足 $\alpha_1 = \alpha_2$。

函数依赖集 F 的正则覆盖可以按图 8-9 中描述的那样计算。需要重视的一点是，当检验一个属性是否无关时，检验时用的是 F_c 当前值中的函数依赖，而不是 F 中的依赖。如果一个函数依赖的右半部只包含一个属性，例如 $A \rightarrow C$，并且这个属性是无关的，那么我们将得到一个右半部为空的函数依赖。这样的函数依赖应该删除。

```
F_c = F
repeat
    使用合并律将 F_c 中所有形如 α1 → β1 和 α1 → β2 的依赖
        替换为 α1 → β1β2
    在 F_c 中寻找一个函数依赖 α → β，它在 α 或在 β 中
        具有一个无关属性
        /* 注意，使用 F_c 而非 F 检验无关属性 */
    如果找到一个无关属性，则将它从 F_c 中的 α → β 中删除
until (F_c 不变)
```

图 8-9 计算正则覆盖

可以证明 F_c 的正则覆盖 F_c 与 F 具有相同的闭包；因此，验证是否满足 F_c 等价于验证是否满足 F。但是，从某种意义上说，F_c 是最小的——它不含无关属性，并且它合并了具有相同左半部的函数依赖。所以验证 F_c 比验证 F 本身更容易。

考虑模式(A, B, C)上的如下函数依赖集F：

$$A \rightarrow BC$$
$$B \rightarrow C$$
$$A \rightarrow B$$
$$AB \rightarrow C$$

让我们来计算F的正则覆盖。

- 存在两个函数依赖在箭头左边具有相同的属性集：

$$A \rightarrow BC$$
$$A \rightarrow B$$

我们将这些函数依赖合并成$A \rightarrow BC$。

- A在$AB \rightarrow C$中是无关的，因为F逻辑蕴涵$(F - \{AB \rightarrow C\}) \cup \{B \rightarrow C\}$。这个断言为真，因为$B \rightarrow C$已经在函数依赖集中。
- C在$A \rightarrow BC$中是无关的，因为$A \rightarrow BC$被$A \rightarrow B$和$B \rightarrow C$逻辑蕴涵。

于是，正则覆盖为：

$$A \rightarrow B$$
$$B \rightarrow C$$

给定函数依赖集F，可能集合中有一整条函数依赖都是无关的，也就是说删掉它不改变F的闭包。我们可以证明F的正则覆盖F_c不包含这种无关的函数依赖。利用反证法，假设F_c中存在这种无关函数依赖，则依赖的右半部分属性将是无关的，这与正则覆盖的定义矛盾。

正则覆盖未必是唯一的。例如，考虑函数依赖集$F = \{A \rightarrow BC, B \rightarrow AC, C \rightarrow AB\}$。如果我们对$A \rightarrow BC$进行无关性检验，会发现在$F$下$B$和$C$都是无关的。然而，两个都删掉是不对的！计算正则覆盖的算法选择其中一个删除。那么，

342
↓
344

1. 如果删除C，我们得到集合$F' = \{A \rightarrow B, B \rightarrow AC, C \rightarrow AB\}$。现在，在$F'$下，在$A \rightarrow B$右半部$B$就不是无关的了。继续执行算法，我们发现$C \rightarrow AB$右半部的$A$和$B$都是无关的，这导致两种正则覆盖

$$F_c = \{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$$
$$F_c = \{A \rightarrow B, B \rightarrow AC, C \rightarrow B\}$$

2. 如果删除B，我们得到集合$\{A \rightarrow C, B \rightarrow AC, C \rightarrow AB\}$。这种情况与前面的情况是对称的，将导致两种正则覆盖

$$F_c = \{A \rightarrow C, C \rightarrow B, B \rightarrow A\}$$
$$F_c = \{A \rightarrow C, B \rightarrow C, C \rightarrow AB\}$$

作为练习，你能再找到一个F上的正则覆盖吗？

8.4.4 无损分解

令$r(R)$为一个关系模式，F为$r(R)$上的函数依赖集。令R_1和R_2为R的分解。如果用两个关系模式$r_1(R_1)$和$r_2(R_2)$替代$r(R)$时没有信息损失，则我们称该分解是**无损分解**(lossless decomposition)。更精确地说，我们称分解是无损的，如果对于所有的合法数据库实例(即满足指定的函数依赖和其他约束的数据库实例)，关系r包含与下述SQL查询结果相同的元组集：

```
select *
from (select R1 from r)
    natural join
    (select R2 from r)
```

这用关系代数可以更简洁地表示为：

$$\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$$

换句话说，如果我们把 r 投影至 R_1 和 R_2 上，然后计算投影结果的自然连接，我们仍然得到一模一样的 r。不是无损分解的分解称为**有损分解**（lossy decomposition）。**无损连接分解**（lossless-join decomposition）和**有损连接分解**（lossy-join decomposition）这两个术语有时用来代替无损分解和有损分解。

345 作为有损连接的一个例子，回想 8.1.2 节中 *employee* 模式的分解：

$$employee1\ (ID,\ name)$$
$$employee2\ (name,\ street,\ city,\ salary)$$

如图 8-3 所示，*employee1* 与 *employee2* 的结果是原关系 *employee* 的一个超集，但是分解是有损的，因为当有两个或多个雇员具有相同的名字时，连接的结果丢失了关于哪个雇员标识和哪个地址及工资相关联的信息。

我们可以用函数依赖来说明什么情况下分解是无损的。令 R、R_1、R_2 和 F 如上。R_1 和 R_2 是 R 的无损分解，如果以下函数依赖中至少有一个属于 F^+：

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$

换句话说，如果 $R_1 \cap R_2$ 是 R_1 或 R_2 的超码，R 上的分解就是无损分解。正如我们前面看到的，我们可以用属性闭包的方法来高效地检验超码。

为举例说明，考虑模式

$$inst\_dept\ (ID,\ name,\ salary,\ dept\_name,\ building,\ budget)$$

我们在 8.1.2 节中将其分解为 *instructor* 和 *department* 模式：

$$instructor\ (ID,\ name,\ dept\_name,\ salary)$$
$$department\ (dept\_name,\ building,\ budget)$$

考虑这两个模式的交集，即 *dept_name*。我们发现，由于 $dept\_name \rightarrow dept\_name,\ building,\ budget$，因此满足无损分解条件。

对于一个模式一次性分解成多个模式的情况，判定无损分解就更复杂了。参看文献注解的相关主题。

虽然对二元分解的测试显然是无损连接的一个充分条件，但只有当所有约束都是函数依赖时它才是必要条件。后面我们将看到几种其他类型的约束（特别是 8.6.1 节讨论的称为多值依赖的一种约束类型），它们即使在不存在函数依赖的情况下仍可保证一个分解是无损连接的。

8.4.5 保持依赖

346 使用函数依赖理论描述保持依赖，要比我们在 8.3.3 节中采用的即席方法容易得多。

令 F 为模式 R 上的一个函数依赖集，R_1，R_2，…，R_n 为 R 的一个分解。F 在 R_i 上的限定（restriction）是 F^+ 中所有只包含 R_i 中属性的函数依赖的集合 F_i。由于一个限定中的所有函数依赖只涉及一个关系模式的属性，因此判定这种依赖是否满足可以只检查一个关系。

注意定义限定时用到的是 F^+ 中的所有函数依赖，而不仅仅是 F 中的。例如，假设 $F = \{A \rightarrow B,\ B \rightarrow C\}$，且我们有一个到 AC 和 AB 的分解。F 在 AC 上的限定则包含 $A \rightarrow C$，因为 $A \rightarrow C$ 属于 F^+，尽管它并没有在 F 中。

限定 F_1，F_2，…，F_n 的集合是能高效检查的依赖集。我们现在必须要问是否只检查这些限定就够了。令 $F' = F_1 \cup F_2 \cup \cdots \cup F_n$。$F'$ 是模式 R 上的一个函数依赖集，不过通常 $F' \neq F$。但是，即使 $F' \neq F$，也有可能 $F'^+ = F^+$。如果后者为真，则 F 中的所有依赖都被 F' 逻辑蕴涵，并且，如果我们证明 F' 是满足的，则我们就证明了 F 也满足。我们称具有性质 $F'^+ = F^+$ 的分解为**保持依赖的分解**（dependency-preserving decomposition）。

图 8-10 给出了判定保持依赖性的算法。输入是分解的关系模式集 $D = \{R_1, R_2, \cdots, R_n\}$ 和函数依赖集 F。因为要计算 F^+，所以这个算法的开销很大。我们将考虑另外两个方法来替代图 8-10 中的算法。

首先，请注意如果 F 中的每一个函数依赖都可以在分解得到的某一个关系上验证，那么这个分解就是保持依赖的。这是一种简单的验证保持依赖性的方法；但是，它并不总是有效。有些情况下，尽管这个分解是保持依赖的，但是在 F 中存在一个依赖，它无法在分解后的任意一个关系上验证。所以这个验证方法只能用作一个易于检查的充分条件，如果验证失败我们也不能断定该分解就不是保持依赖的，而是将不得不采用一般化的验证方法。

```
计算 F+;
    for each D 中的模式 Ri do
        begin
            Fi := F+ 在 Ri 中的限定;
        end
    F' := ∅
    for each 限定 Fi do
        begin
            F' = F' ∪ Fi
        end
    计算 F'+;
    if (F+ = F'+) then return (true)
                  else return (false);
```

图 8-10 保持依赖性的验证

我们现在给出第二种避免计算 F^+ 的验证保持依赖的方法。我们会在列出验证方法之后解释该方法的思想。该验证方法对 F 中的每一个 $\alpha \rightarrow \beta$ 使用下面的过程：

```
result = α
repeat
    for each 分解后的 Ri
        t = (result ∩ Ri)+ ∩ Ri
        result = result ∪ t
    until (result 没有变化)
```

这里的属性闭包是函数依赖集 F 下的。如果 *result* 包含 β 中的所有属性，则函数依赖 $\alpha \rightarrow \beta$ 保持。分解是保持依赖的当且仅当上述过程中 F 的所有依赖都保持。

上述验证方法背后的两个关键的思想如下：

- 第一个思想是验证 F 中的每个函数依赖 $\alpha \rightarrow \beta$，看它是否在 F' 中保持（F' 的定义如图8-10所示）。为此，我们计算 F' 下 α 的闭包；当该闭包包含 β 时，该依赖得以保持。该分解是保持依赖的当（且仅当）F 中的所有函数依赖都保持。
- 第二个思想是使用修改后的属性闭包算法计算 F' 下的闭包，不用先真正计算出 F'。我们希望避免 F' 的计算，因为计算开销很大。注意到 F' 是 F_i 的并集，而 F_i 是 F 在 R_i 上的限定。该算法计算出 F 上（*result* $\cap R_i$）的属性闭包，并与 R_i 的闭包求交，然后将结果属性集加入 *result*；这一系列步骤等价于计算 F_i 下的 *result* 闭包。在 while 循环中对每个 i 重复该步骤就得到了 F' 下的 *result* 闭包。

为了理解这个修改后的属性闭包算法运算正确的原因，我们注意到对于每个 $\gamma \subseteq R_i$，$\gamma \rightarrow \gamma^+$ 是 F^+ 中的一个函数依赖，并且 $\gamma \rightarrow \gamma^+ \cap R_i$ 是 F' 在 R_i 上的限定 F_i 中的函数依赖。反之，如果 $\gamma \rightarrow \delta$ 出现在 F_i 中，则 δ 是 $\gamma^+ \cap R_i$ 的子集。

该判定方法的代价是多项式时间，而不是计算 F^+ 所需的指数时间的代价。

8.5 分解算法

现实世界的数据库模式要比能装在书页中的例子大得多。因此，我们需要能生成属于适当范式的设计的算法。本节给出 BCNF 和 3NF 的相应算法。

347
|
348

8.5.1 BCNF 分解

BCNF 的定义可以直接用于检查一个关系是否属于 BCNF。但是，计算 F^+ 是一个繁重的任务。下面首先描述判定一个关系是否属于 BCNF 的简化检验方法。如果一个关系不属于 BCNF，它可以被分解以创建属于 BCNF 的关系。本节后面将描述一种创建关系的无损分解的算法，使得该分解属于 BCNF。

8.5.1.1 BCNF 的判定方法

在某些情况下，判定一个关系是否属于 BCNF 可以作如下简化：

- 为了检查非平凡的函数依赖 $\alpha \rightarrow \beta$ 是否违反 BCNF，计算 α^+（α 的属性闭包），并且验证它是否包含 R 中的所有属性，即验证它是否是 R 的超码。
- 检查关系模式 R 是否属于 BCNF，仅须检查给定集合 F 中的函数依赖是否违反 BCNF 就足够了，不用检查 F^+ 中的所有函数依赖。

我们可以证明如果 F 中没有函数依赖违反 BCNF，那么 F^+ 中也不会有函数依赖违反 BCNF。

遗憾的是，当一个关系分解后，后一步过程就不再适用。也就是说，当我们判定 R 上的一个分解 R_i 是否违反 BCNF 时，只用 F 就不够了。例如，考虑关系模式 $R(A, B, C, D, E)$，其函数依赖集 F 包括 $A \rightarrow B$ 和 $BC \rightarrow D$。假设 R 分解成 $R_1(A, B)$ 和 $R_2(A, C, D, E)$。现在，因为 F 中没有一个函数依赖只包含来自 (A, C, D, E) 的属性，所以我们也许会误认为 R_2 满足 BCNF。实际上，F^+ 中有一个函数依赖 $AC \rightarrow D$（可以由为传递律从 F 中的两个依赖推出），这表明 R_2 不属于 BCNF。所以，我们也许需要一个属于 F^+ 但不属于 F 的依赖，来证明一个分解后的关系不属于 BCNF。

BCNF 的另一种判定方法有时会比计算 F^+ 上的所有函数依赖简单。为了检查 R 分解后的关系 R_i 是否属于 BCNF，我们应用如下判定：

- 对于 R_i 中属性的每个子集 α，确保 α^+（F 下 α 的属性闭包）要么不包含 $R_i - \alpha$ 的任何属性，要么包含 R_i 的所有属性。

如果 R_i 上有某个属性集 α 违反该条件，考虑如下的函数依赖，可以证明它出现在 F^+ 中：

$$\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$$

上面这个函数依赖说明 R_i 违反 BCNF。

8.5.1.2 BCNF 分解算法

我们现在能给出一个关系模式分解的一般方法以满足 BCNF。图 8-11 所示为实现该过程的一个算法。若 R 不属于 BCNF，则可用这个算法将 R 分解成一组 BCNF 模式 $R_1, R_2, \cdots, R_n$。这个算法利用违反 BCNF 的依赖进行分解。

```
result := {R};
    done := false;
    计算 F+;
    while (not done) do
        if (result 中存在模式 Ri 不属于 BCNF)
        then begin
            令 α → β 为一个在 Ri 上成立的非平凡函数依赖，满足 α → Ri 不属于 F+，并且 α ∩ β = ∅;
            result := (result - Ri) ∪ (Ri - β) ∪ (α, β);
    end
    else done := true;
```

图 8-11 BCNF 分解算法

用这个算法产生的分解不仅是一个 BCNF 分解，而且是一个无损分解。来看一下为什么该算法只产生无损分解，我们注意到，当我们用 $(R_i - \beta)$ 和 (α, β) 取代模式 R_i 时，依赖 $\alpha \rightarrow \beta$ 成立，而且 $(R_i - \beta) \cap (\alpha, \beta) = \alpha$。

如果我们没有要求 $\alpha \cap \beta = \emptyset$，那么 $\alpha \cap \beta$ 中的那些属性就不会出现在模式 $(R_i - \beta)$ 中，而依赖 $\alpha \rightarrow \beta$ 也不再成立。

容易看出在 8.3.2 节中对 *inst_dept* 的分解结果可以通过应用该算法获得。函数依赖 *dept_name* → *building*, *budget* 满足 $\alpha \cap \beta = \emptyset$ 条件，因此被选择用来分解该模式。

BCNF 分解算法所需时间为原始模式规模的指数级别，因为该算法检查分解中的一个关系是否满足 BCNF 的代价可以达到指数级。文献注解提供了一个算法的参考文献，该算法能够在多项式时间内计算 BCNF 分解。但是，这个算法可能"过于规范化"，即，分解不必要分解的关系。

349
350

举一个更长的使用 BCNF 分解算法的例子，假定我们有一个数据库设计使用了以下的 *class* 模式：

class (*course_id*, *title*, *dept_name*, *credits*, *sec_id*, *semester*, *year*, *building*, *room_number*, *capacity*, *time_slot_id*)

我们要求在 *class* 上成立的函数依赖集合为

$$course\_id \rightarrow title,\ dept\_name,\ credits$$
$$building,\ room\_number \rightarrow capacity$$
$$course\_id,\ sec\_id,\ semester,\ year \rightarrow building,\ room\_number,\ time\_slot\_id$$

该模式的一个候选码是 {*course_id*, *sec_id*, *semester*, *year*}。

我们可以按如下方式将图 8-11 的算法用于 *class* 例子：

- 函数依赖：

$$course\_id \rightarrow title,\ dept\_name,\ credits$$

成立，但 *course_id* 不是超码。因此，*class* 不属于 BCNF。我们将 *class* 替换为：

$$course\ (course\_id,\ title,\ dept\_name,\ credits)$$
$$class-1\ (course\_id,\ sec\_id,\ semester,\ year,\ building,\ room\_number,\ capacity,\ time\_slot\_id)$$

course 上成立的唯一一个非平凡函数依赖的箭头左侧包含 *course_id*。由于 *course_id* 是 *course* 的码，因此关系 *course* 属于 BCNF。

- *class* - 1 的一个候选码为 {*course_id*, *sec_id*, *semester*, *year*}，函数依赖：

$$building,\ room\_number \rightarrow capacity$$

在 *class* - 1 上成立，但 {*building*, *room_number*} 不是 *class* - 1 的超码。我们将 *class* - 1 替换为：

$$classroom\ (building,\ room\_number,\ capacity)$$
$$section\ (course\_id,\ sec\_id,\ semester,\ year,\ building,\ room\_number,\ time\_slot\_id)$$

classroom 和 *section* 属于 BCNF。

于是，*class* 分解为三个关系模式 *course*、*classroom* 和 *section*，它们都属于 BCNF。这些模式对应于我们在本章和前面章节使用的模式。你可以验证该分解既是无损的，又是保持依赖的。

8.5.2 3NF 分解

图 8-12 给出了将模式转化为 3NF 的保持依赖且无损的分解的算法。该算法中使用的依赖集 F_c 是 F 的正则覆盖。注意，该算法考虑的是模式 R_j 的集合（$j = 1, 2, \cdots, i$）；初始 $i = 0$ 且该集合为空。

让我们将该算法用于 8.3.4 节中的例子，我们曾证明

$$dept\_advisor\ (s\_ID,\ i\_ID,\ dept\_name)$$

是属于3NF 的，虽然它不属于 BCNF。该算法使用 F 中以下的函数依赖：

$$f_1: i\_ID \rightarrow dept\_name$$
$$f_2: s\_ID,\ dept\_name \rightarrow i\_ID$$

```
令 F_c 为 F 的正则覆盖;
i := 0;
for each F_c 中的函数依赖 α → β
    i := i + 1;
    R_i := αβ;
if 模式 R_j, j = 1, 2, ..., i 都不包含 R 的候选码
then
    i := i + 1;
    R_i := R 的任意候选码;
/* (可选)移除冗余关系 */
repeat
    if 模式 R_k 包含于另一个模式 R_l 中
    then
    /* 删除 R_k */
    R_k := R_i;
    i := i - 1;
until 不再有可以删除的 R_k
return(R_1, R_2, ..., R_i)
```

图 8-12 到 3NF 的保持依赖且无损的分解

在 F 的任意一个函数依赖中都不存在无关属性，因此 F_c 包含 f_1 和 f_2。该算法生成模式（*i_ID*, *dept_name*）作为 R_1，以及模式（*s_ID*, *dept_name*, *i_ID*）作为 R_2。该算法随即发现 R_2 包含一个候选码，因此不再继续创建关系模式。

生成的模式集可能会包含冗余的模式，即一个模式 R_k 包含另一个模式 R_l 所有的属性。例如，上面的 R_1 包含 R_2 所有的属性。这个算法会删除所有这种包含于其他模式的模式。在删除的 R_l 上验证的任意一个依赖在相应的关系 R_k 上也验证，即使删除了 R_l，分解仍旧是无损的。

现在我们再次考虑 8.5.1.2 节中的 *class* 模式，并运用 3NF 分解算法。我们所列出的函数依赖集恰

351
352

好是正则覆盖。因此，该算法给出同样的三个模式 *course*、*classroom* 和 *section*。

上例说明了3NF算法的一个有趣的特性：有时，结果不仅属于3NF，而且也属于BCNF。这暗示了生成BCNF设计的另一个方法：首先使用3NF算法，然后对于3NF设计中任何一个不属于BCNF的模式，用BCNF算法分解。如果结果不是保持依赖的，则还原到3NF设计。

8.5.3 3NF算法的正确性

3NF算法通过为正则覆盖中的每个依赖显式地构造一个模式确保依赖的保持。该算法通过保证至少有一个模式包含被分解模式的候选码，确保该分解是一个无损分解。实践习题8.14深入审视这种算法是以保证无损分解的证据。

这个算法也称为**3NF合成算法**(3NF synthesis algorithm)，因为它接受一个依赖集合，每次添加一个模式，而不是对初始的模式反复地分解。该算法的结果不是唯一确定的，因为一个函数依赖集有不止一个正则覆盖，而且，某些情况下该算法的结果依赖于该算法考虑 F_c 中的依赖的顺序。该算法有可能分解一个已经属于3NF的关系；不过，仍然保证分解是属于3NF的。

如果一个关系 R_i 在由该合成算法产生的分解中，则 R_i 属于3NF。回想当我们判定3NF时，考虑右半部是单个属性的函数依赖就足够了。因此，要看 R_i 是否属于3NF，你必须确认 R_i 上的任意函数依赖 $\gamma \rightarrow B$ 满足3NF的定义。假定该合成算法中产生 R_i 的函数依赖是 $\alpha \rightarrow \beta$。现在，因为 B 属于 R_i，而且 $\alpha \rightarrow \beta$ 产生 R_i，所以 B 一定或于 α 或 β。让我们考虑以下三种可能情况：

- B 既属于 α 又属于 β。在这种情况下，依赖 $\alpha \rightarrow \beta$ 将不可能属于 F_c，否则 B 在 β 中将是无关的。所以这种情况不成立。
- B 属于 β 但不属于 α。考虑两种情况：
  - γ 是一个超码。满足3NF的第二个条件。
  - γ 不是超码。则 α 必定包含某些不在 γ 中的属性。现在，由于 $\gamma \rightarrow B$ 在 F^+ 中，因此它一定是通过使用 γ 上的属性闭包算法从 F_c 推导出来的。该推导不可能用到 $\alpha \rightarrow \beta$，否则 α 一定包含在 γ 的属性闭包里，而这是不可能的，因为我们假定 γ 不是超码。现在，使用 $\alpha \rightarrow (\beta - B)$ 和 $\gamma \rightarrow B$，我们可以推导出 $\alpha \rightarrow B$(由于 $\gamma \subseteq \alpha\beta$；并且因为 $\gamma \rightarrow B$ 是非平凡的，因此 γ 不包含 B)。这表明 B 在 $\alpha \rightarrow \beta$ 的右半部是无关的，这也是不可能的，因为 $\alpha \rightarrow \beta$ 属于正则覆盖 F_c。所以，如果 B 属于 β，那么 γ 一定是超码并且满足3NF的第二个条件。
- B 属于 α 但不属于 β。

因为 α 是候选码，所以满足3NF定义中的第三个条件。

有趣的是，我们描述的3NF分解算法可以在多项式时间内实现，尽管判定给定关系是否属于3NF是NP-hard的(这意味着设计一个多项式时间的算法来解决该问题是不太可能的)。

8.5.4 BCNF和3NF的比较

对于关系数据库模式的两种范式——3NF和BCNF，3NF的一个优点是我们总可以在满足无损并保持依赖的前提下得到3NF设计。然而3NF也有一个缺点：我们可能不得不用空值表示数据项间的某些可能有意义的联系，并且存在信息重复的问题。

我们对应用函数依赖进行数据库设计的目标是：

1. BCNF。
2. 无损。
3. 保持依赖。

由于不是总能达到所有这三个目标，因此我们也许不得不在BCNF和保持依赖的3NF中做出选择。

值得注意的是，除了可以通过用主码或者唯一约束来声明超码的特殊情况外，SQL不提供指定函数依赖的途径。通过写断言来保证函数依赖(参见实践习题8.9)虽然有点麻烦，但也是有可能的。遗憾的是，目前没有数据库系统支持强制实施函数依赖所需的复杂断言，而且检查这些断言的开销很大。所以即使我们有一个保持依赖的分解，但如果我们使用标准SQL，我们只能对那些左半部是码的函数依赖进行高效的检查。

虽然在分解不能保持依赖时，对函数依赖的检查可能会用到连接，但倘若数据库系统支持物化视图上的主码约束，我们就可以通过使用物化视图的方法来降低开销，这是许多数据库系统都支持的。给定一个非保持依赖的 BCNF 分解，我们考虑正则覆盖 F，里每一个在分解中未保持的依赖。对于每一个这样的依赖 $\alpha \rightarrow \beta$，我们定义一个物化视图对分解中所有的关系计算连接，并且将结果投影到 $\alpha\beta$ 上。利用一个约束 **unique**(α)或者 **primary key**(α)，就可以在物化视图上很容易地检查函数依赖。 354

从负面来看，物化视图会需求一些时间和空间的额外开销；但是从正面来看，应用程序员不需要写代码来保持冗余数据在更新上的一致性。维护物化视图是数据库系统的工作，即在数据库更新时做出相应的更新。（本书后面的 13.5 节对数据库系统如何高效地进行物化视图的维护进行了概述。）

遗憾的是，目前绝大多数数据库系统不支持物化视图上的约束。虽然 Oracle 数据库支持物化视图上的约束，但它默认当访问视图时进行视图维护，而不是更新底层关系时；[⊖]结果是，约束冲突有可能在更新已经执行之后才检测出来，这使得该检测没有用。

因此，在不能得到保持依赖的 BCNF 分解的情况下，通常我们倾向于选择 BCNF，因为在 SQL 中检查非主码约束的函数依赖很困难。

8.6 使用多值依赖的分解

有些关系模式虽然属于 BCNF，但从某种意义上说仍存在信息重复的问题，所以看起来没有充分规范化。考虑大学的例子，一个教师可以和多个系相关联。

$$inst\ (ID,\ dept\_name,\ name,\ street,\ city)$$

敏锐的读者将发现这个模式是一个非 BCNF 模式，因为有函数依赖

$$ID \rightarrow name,\ street,\ city$$

并且 ID 不是 $inst$ 的码。

进一步假设一个教师可以有多个地址（比如说，一个冬天的家和一个夏天的家）。那么，我们不再想强制实施函数依赖"$ID \rightarrow street,\ city$"，不过当然，我们仍想强制实施"$ID \rightarrow name$"（即大学不处理有多个别名的教师的情况）。根据 BCNF 分解算法，得到两个模式： 355

$$r_1\ (ID,\ name)$$
$$r_2\ (ID,\ dept\_name,\ street,\ city)$$

这两个模式都属于 BCNF（回想到一个教师可以和多个系关联，并且一个系可以有多名教师，因此"$ID \rightarrow dept\_name$"和"$dept\_name \rightarrow ID$"都不成立）。

尽管 r_2 属于 BCNF，但是冗余仍然存在。对于一名教师所关联的每个系都要将该教师的每一个居住地址信息重复一次。为了解决这个问题，可以把 r_2 进一步分解为：

$$r_{21}\ (dept\_name,\ ID)$$
$$r_{22}\ (ID,\ street,\ city)$$

但是，并不存在任何约束来引导我们进行这个分解。

为处理这个问题，我们必须定义一种新的约束形式，称为多值依赖。正如我们对函数依赖所做的一样，我们将利用多值依赖定义关系模式的一种范式。这种范式称为**第四范式**（Fourth Normal Form，4NF），它比 BCNF 的约束更严格。我们将看到每个 4NF 模式也都属于 BCNF，但有些 BCNF 模式不属于 4NF。

8.6.1 多值依赖

函数依赖规定了某些元组不能出现在关系中。如果 $A \rightarrow B$ 成立，我们就不能有两个元组在 A 上的值相同而在 B 上的值不同。从另一个角度出发，多值依赖并不排除某些元组的存在，而是要求某种形式的其他元组存在于关系中。由于这个原因，函数依赖有时称为**相等产生依赖**（equality-generating dependency），而多值依赖称为**元组产生依赖**（tuple-generating dependency）。

⊖ 至少到 Oracle 10g 版本如此。

令 $r(R)$ 为一关系模式，并令 $\alpha \subseteq R$ 且 $\beta \subseteq R$。**多值依赖**（multivalued dependency）

$$\alpha \longrightarrow\!\!\!\!\!\rightarrow \beta$$

在 R 上成立的条件是，在关系 $r(R)$ 的任意合法实例中，对于 r 中任意一对满足 $t_1[\alpha] = t_2[\alpha]$ 的元组对 t_1 和 t_2，r 中都存在元组 t_3 和 t_4，使得

$$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$$
$$t_3[\beta] = t_1[\beta]$$
$$t_3[R - \beta] = t_2[R - \beta]$$
$$t_4[\beta] = t_2[\beta]$$
$$t_4[R - \beta] = t_1[R - \beta]$$

356

这个定义看似复杂，实则不然。图 8-13 所示为 t_1、t_2、t_3 和 t_4 的表格图。直观地，多值依赖 $\alpha \longrightarrow\!\!\!\!\!\rightarrow \beta$ 是说 α 和 β 之间的联系独立于 α 和 $R - \beta$ 之间的联系。若模式 R 上的所有关系都满足多值依赖 $\alpha \longrightarrow\!\!\!\!\!\rightarrow \beta$，则 $\alpha \longrightarrow\!\!\!\!\!\rightarrow \beta$ 是模式 R 上平凡的多值依赖。因此，如果 $\beta \subseteq \alpha$ 或 $\beta \cup \alpha = R$，则 $\alpha \longrightarrow\!\!\!\!\!\rightarrow \beta$ 是平凡的。

	α	β	$R - \alpha - \beta$
t_1	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
t_2	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$b_{j+1} \ldots b_n$
t_3	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$
t_4	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$a_{j+1} \ldots a_n$

图 8-13 $\alpha \longrightarrow\!\!\!\!\!\rightarrow \beta$ 的表格表示

为说明函数依赖和多值依赖的区别，我们再次考虑模式 r_2 及图 8-14 中所示的该模式上的一个关系的例子。我们必须为教师的每个地址重复一遍系名，并且必须为教师关联的每个系重复地址。这种重复是不必要的，因为教师与其地址间的联系独立于教师与系间的联系。如果一个 *ID* 为 22222 的教师与物理系关联，我们希望这个系与该教师的所有地址都关联。因此，图 8-15 的关系是非法的。要使该关系合法化，需要向图 8-15 的关系中加入元组（Physics，22222，Main，Manchester）及（Math，22222，North，Rye）。

ID	*dept_name*	*street*	*city*
22222	Physics	North	Rye
22222	Physics	Main	Manchester
12121	Finance	Lake	Horseneck

图 8-14 一个 BCNF 模式上的关系中有冗余的例子

ID	*dept_name*	*street*	*city*
22222	Physics	North	Rye
22222	Math	Main	Manchester

图 8-15 一个非法的 r_2 关系

将前面这个例子与多值依赖的定义相比较，我们发现我们需要多值依赖

$$ID \longrightarrow\!\!\!\!\!\rightarrow street, city$$

成立。（多值依赖 $ID \longrightarrow\!\!\!\!\!\rightarrow dept\_name$ 也可以。我们不久将会看到它们是等价的。）

与函数依赖相同，我们将以两种方式使用多值依赖：

1. 检验关系以确定它们在给定的函数依赖集和多值依赖集下是否合法。
2. 在合法关系集上指定约束；因此我们将只考虑满足给定函数依赖集和多值依赖集的关系。

请注意，若关系 r 不满足给定多值依赖，我们可以通过向 r 中增加元组构造一个确实满足多值依赖的关系 r'。

令 D 表示函数依赖和多值依赖的集合。D 的闭包 D^+ 是由 D 逻辑蕴涵的所有函数依赖和多值依赖的集合。与函数依赖相同，我们可以用函数依赖和多值依赖的规范定义根据 D 计算出 D^+。我们可以用这样的推理处理非常简单的多值依赖。幸运的是，实际问题中遇到的多值依赖都很简单。对于复杂的依赖，用推理规则系统来推导出依赖集会更好。

由多值依赖的定义，我们可以得出以下规则，对于 $\alpha, \beta \subseteq R$：

- 若 $\alpha \rightarrow \beta$，则 $\alpha \longrightarrow\!\!\!\!\!\rightarrow \beta$。换句话说，每一个函数依赖也是一个多值依赖。
- 若 $\alpha \longrightarrow\!\!\!\!\!\rightarrow \beta$，则 $\alpha \longrightarrow\!\!\!\!\!\rightarrow R - \alpha - \beta$。

附录 C.1.1 列出了多值依赖的一个推理规则系统。

8.6.2 第四范式

再次考虑 BCNF 模式的例子

$$r_2(ID, dept\_name, street, city)$$

其中多值依赖"$ID \twoheadrightarrow street, city$"成立。我们在 8.6 节的开头看到，尽管该模式属于 BCNF，但是这个设计并不理想，因为我们必须为每个系重复教师的地址信息。我们将看到，我们可以用给定的多值依赖改进数据库的设计，将该模式分解为第四范式。

函数依赖和多值依赖集为 D 的关系模式 $r(R)$ 属于**第四范式**(4NF)的条件是，对 D^+ 中所有形如 357
$\alpha \twoheadrightarrow \beta$ 的多值依赖(其中 $\alpha \subseteq R$ 且 $\beta \subseteq R$)，至少有以下之一成立： 358

- $\alpha \twoheadrightarrow \beta$ 是一个平凡的多值依赖。
- α 是 R 的一个超码。

数据库设计属于 4NF 的条件是构成该设计的关系模式集中的每个模式都属于 4NF。

请注意，4NF 定义与 BCNF 定义的唯一不同在于多值依赖的使用。4NF 模式一定属于 BCNF。为了了解这点，我们注意到，如果模式 $r(R)$ 不属于 BCNF，则 R 上存在非平凡的函数依赖 $\alpha \rightarrow \beta$，其中 α 不是超码。由于 $\alpha \rightarrow \beta$ 就意味着 $\alpha \twoheadrightarrow \beta$，因此 $r(R)$ 不属于 4NF。

令 $r(R)$ 为关系模式，$r_1(R_1)$，$r_2(R_2)$，…，$r_n(R_n)$ 为 $r(R)$ 的分解。要检验分解中的每一个关系模式 r_i 是否属于 4NF，我们需要找到每一个 r_i 上成立的多值依赖。回想到对于函数依赖集 F，F 在 R_i 上的**限定** F_i 是 F^+ 中所有只含 R_i 中属性的函数依赖。现在考虑函数依赖和多值依赖的集合 D。D 在 R_i 上的**限定**是集合 D_i，它包含

1. D^+ 中所有只含 R_i 中属性的函数依赖。
2. 所有形如：

$$\alpha \twoheadrightarrow \beta \cap R_i$$

的多值依赖，其中 $\alpha \subseteq R_i$ 且 $\alpha \twoheadrightarrow \beta$ 属于 D^+。

8.6.3 4NF 分解

我们可以将 4NF 与 BCNF 之间的相似之处应用到 4NF 模式分解中。图 8-16 给出了 4NF 分解算法。它与图 8-11 的 BCNF 分解算法相同，除了它使用多值依赖以及 D^+ 在 R_i 上的限定。

```
result := {R};
done := false;
计算 D+；给定模式 Ri，令 Di 表示 D+ 在 Ri 上的限定
while (not done) do
    if (result 中存在 Ri 不属于 4NF)
    then begin
        令 α →→ β 为 Ri 上成立的非平凡多值依赖
        使得 α → Ri 不属于 Di，并且 α ∩ β = ∅;
        result := (result - Ri) ∪ (Ri - β) ∪ (α, β);
    end
    else done := true;
```

图 8-16 4NF 分解算法

如果我们将图 8-16 的算法应用于 $(ID, dept\_name, street, city)$，我们发现 $ID \twoheadrightarrow dept\_name$ 是非平凡的多值依赖，并且 ID 不是该模式的超码。按照该算法，我们将它替换为两个模式：

$$r_{21}(ID, dept\_name)$$
$$r_{22}(ID, street, city)$$

这对模式属于 4NF，它消除了我们前面所遇到的冗余。

与单独考虑函数依赖时一样，我们希望得到无损的和保持依赖的分解。从下面关于多值依赖和无 359
损性的性质可以看出图 8-16 的算法只产生无损分解：

- 令 $r(R)$ 为一关系模式，D 为 R 上的函数依赖和多值依赖的集合。令 $r_1(R_1)$ 和 $r_2(R_2)$ 为 R 的一个分解。该分解是 R 的无损分解，当且仅当下面的多值依赖中至少有一个属于 D^+：

$$R_1 \cap R_2 \twoheadrightarrow R_1$$
$$R_1 \cap R_2 \twoheadrightarrow R_2$$

回想我们在 8.4.4 节提到的，若 $R_1 \cap R_2 \rightarrow R_1$ 或 $R_1 \cap R_2 \rightarrow R_2$，则 $r_1(R_1)$ 和 $r_2(R_2)$ 为 $r(R)$ 的无损分解。上面这个关于多值依赖的性质是对无损性更一般化的表述。它指明，对于每个将 $r(R)$ 分解为两个模式 $r_1(R_1)$ 和 $r_2(R_2)$ 的无损分解，两个依赖 $R_1 \cap R_2 \rightarrow R_1$ 或 $R_1 \cap R_2 \twoheadrightarrow R_2$ 中至少有一个成立。

当我们处理存在多值依赖的关系模式进行分解时，保持依赖的问题变得更加复杂。附录 C.1.2 将继续讨论这个问题。

8.7 更多的范式

第四范式不是"最终"的范式。正如我们前面看到的，多值依赖有助于我们理解并消除某些形式的信息重复，而这种信息重复用函数依赖是无法理解的。还有一些类型的约束称作**连接依赖**(join dependency)，它概化了多值依赖，并引出了另一种范式称作**投影 - 连接范式**(Project-Join Normal Form，PJNF)(PJNF 在某些书中称为**第五范式**，fifth normal form)。还有一类更一般化的约束，它引
360 出一种称作**域 - 码范式**(Domain-Key Normal Form，DKNF)的范式。

使用这些一般化的约束的一个实际的问题是，它们不仅难以推导，而且也没有形成一套具有正确有效性和完备性的推理规则用于约束的推导。因此 PJNF 和 DKNF 很少使用。附录 C 将更详细地介绍这些范式。

很明显，我们的讨论中没有**第二范式**(Second Normal Form，2NF)。因为它只具有历史的意义，所以我们没有对它进行讨论。在实践习题 8.17 中，我们对它简单地定义，并简做练习。

8.8 数据库设计过程

迄今为止我们详细讨论了范式和规范化的问题，本节将研究规范化是如何融合在整体数据库设计过程中的。

在本章的前面，从 8.3 节开始，我们假定给定关系模式 $r(R)$，并对之进行规范化。我们可以采用以下几种方法得到模式 $r(R)$：

1. $r(R)$ 可以是由 E-R 图向关系模式集进行转换时生成的。

2. $r(R)$ 可以是一个包含所有有意义的属性的单个关系。然后规范化过程中将 R 分解成一些更小的模式。

3. $r(R)$ 可以是关系的函数设计的结果，然后我们检验它们是否满足一个期望的范式。

在本节的剩余的部分，我们将考察这些方法之间的潜在关系，我们也将考察数据库设计中一些实际的问题，包括为了保证性能的去规范化，以及规范化未没检测到的不良设计的例子。

8.8.1 E-R 模型和规范化

当我们小心地定义 E-R 图，并正确地识别所有的实体时，由 E-R 图生成的关系模式应该就不需要太多进一步的规范化。然而，一个实体的属性之间有可能存在函数依赖。例如，假设 *instructor* 实体集包含属性 *dept_name* 和 *dept_address*，并且存在函数依赖 $dept\_name \rightarrow dept\_address$，那么我们就要规范化从 *instructor* 生成的关系。

大多数这种依赖的例子都是由不好的 E-R 图设计引起的。在上面的例子中，如果我们正确设计 E-R 图，我们将创建一个包含属性 *dept_address* 的 *department* 实体集以及 *instructor* 与 *department* 之间的
361 一个联系集。同样，包含两个实体集以上的联系集有可能会使产生的模式不属于所期望的范式。由于大部分的联系集都是二元的，因此这样的情况相对稀少。(事实上，某些变种形式的 E-R 图实际上很难或者不可能指定非二元的联系集。)

函数依赖有助于我们检测到不好的 E-R 设计。如果生成的关系模式不属于想要的范式，该问题可以在 E-R 图中解决。也就是说，规范化可以作为数据建模的一部分规范地进行。规范化既可以留给数据库设计者在 E-R 建模时靠直觉实现，也可以从 E-R 模型生成的关系模式上规范地进行，二者可任选其一。

细心的读者可能注意到，为了解释多值依赖和第四范式的必要性，我们不得不从一个不是由E-R设计导出的模式出发。事实上，创建 E-R 设计的过程倾向于产生 4NF 设计。如果一个多值依赖成立且不是被相应的函数依赖所隐含的，那么它通常由以下情况引起：

- 一个多对多的联系集。
- 实体集的一个多值属性。

对于多对多的联系集，每个相关的实体集都有自己的模式，并且存在一个额外的模式用于表示联系集。对于多值属性，会创建一个单独的模式，包含该属性以及实体集的主码（正如在实体集 *instructor* 中的 *phone_number* 属性的例子一样）。

关系数据库设计的泛关系方法从一个假设开始，它假定存在单个关系模式包含所有有意义的属性。这单个模式定义了用户和应用程序如何与数据库交互。

8.8.2 属性和联系的命名

数据库设计的一个期望的特性是**唯一角色假设**（unique-role assumption），这意味着每个属性名在数据库中只有唯一的含义。这使得我们不能使用同一个属性在不同的模式中表示不同的东西。例如，相反地，我们可能考虑使用属性 *number* 在模式 *instructor* 中表示电话号码，而在模式 *classroom* 中表示房间号。对模式 *instructor* 上的一个关系和模式 *classroom* 上的一个关系进行连接是毫无意义的。用户和应用程序开发人员必须小心操作以保证在每个场合使用了正确的 *number*，而对电话号码和房间号分别使用不同的属性名则能减少用户的错误。

虽然对不兼容的属性使用不同的名字是个好主意，但是如果不同关系中的属性有相同的含义，使用相同的名字可能就是个好主意了。因此我们对实体集 *instructor* 和 *student* 使用相同的属性名 "*name*"。如果不这样（即我们对教师和学生的名字用不同的命名规范），那么如果我们想通过创建一个实体集 *person* 来泛化这两个实体集，我们就不得不重命名属性。因此，即使我们当前没有对 *instructor* 和 *student* 的泛化。然而如果我们预见到这种可能性，最好还是在这两个实体集（和关系）中使用同样的名字。 362

尽管从技术上看，一个模式中的属性名的顺序无关紧要，然而习惯上把主码属性列在前面。这会使得查看默认输出（例如 **select** * 的输出）更加容易。

在大的数据库模式中，联系集（及其导出的模式）常常以相关实体集名称的拼接来命名，可能使用连字符或下划线连接。我们已经使用了几个这样的名字，例如 *inst_sec* 和 *student_sec*。我们使用名字 *teaches* 和 *takes* 而不使用更长的拼接名字。由于对于少数几个联系集，想起它们的相关实体集并不难，因而这是可以接受的。我们无法一直通过简单的拼接创建联系集的名字；例如，雇员之间的管理者或被管理联系，如果我们称之为 *employee_employee* 就没有什么意义。相似地，如果一对实体集间可能有多个联系集，联系集的名字就必须包含额外的部分以区别这些联系集。

不同的组织对于命名实体集有不同的习惯。举例来说，我们可能称一个学生的实体集为 *student* 或者 *students*。在数据库设计中我们采用了单数形式。使用单数或者复数形式都是可以接受的。只要所有实体集都一致地使用该习惯就可以了。

随着模式变得更大，联系集的数量不断增加，使用对属性、联系和实体一致的命名方式会使得数据库设计者和应用程序员更加轻松。

8.8.3 为了性能去规范化

有时候数据库设计者会选择包含冗余信息的模式，也就是说，没有规范化。对一些特定的应用，他们使用冗余来提高性能。不使用规范化模式的代价是用来保持冗余数据一致性的额外工作（以编码时间和执行时间计算）。

例如，假定每次访问一门课程时，所有的先修课都必须和课程信息一起显示。在规范化的模式中，需要连接 *course* 和 *prereq*。

一个不计算连接的方法是保存一个包含 *course* 和 *prereq* 的所有属性的关系。这使得显示"全部"课程信息更快。然而，对于每门先修课都要重复课程的信息。而且每当添加或删除先修课时应用程序就必须更新所有的副本。把一个规范化的模式变成非规范化的过程称为**去规范化**(denormalization)，设计者用它调整系统的性能以支持响应时间苛刻的操作。

现今许多数据库系统支持一种更好的方法，即使用规范化的模式，同时将 *course* 和 *prereq* 的连接作为物化视图额外存储。(回忆一下，物化视图是将结果存储在数据库中的视图，当它用到的关系更
363 新时也相应更新。)像去规范化一样，使用物化视图确实会有空间和时间上的开销；不过它也有优点，就是保持视图更新的工作是由数据库系统而不是应用程序员完成的。

8.8.4 其他设计问题

数据库设计中有一些方面不能用规范化描述，而它们也会导致不好的数据库设计。与时间或时间段有关的数据就存在这样的问题。这里我们给出一些例子；显然，这样的设计应该避免。

考虑一个大学数据库，我们想存储不同年份中每个系的教师总数。可以用关系 *total_inst*(*dept_name*, *year*, *size*)来存储所需要的信息。这个关系上的唯一的函数依赖是 *dept_name*, *year* → *size*，这个关系属于 BCNF。

另一个设计是使用多个关系，每个关系存储一个年份的系规模信息。假设我们感兴趣的年份为 2007、2008 和 2009；我们将得到形如 *total_inst_2007*，*total_inst_2008*，*total_inst_2009* 这样的关系，它们都建立在模式(*dept_name*, *size*)之上。因为每一个关系上的唯一的函数依赖是 *dept_name* → *size*，所以这些关系也属于 BCNF。

但是，这个设计显然是一个不好的主意——我们必须每年都创建一个新的关系，每年还不得不写新的查询从而把新的关系考虑进去。由于可能涉及很多关系，因此查询也将更加复杂。

然而还有一种方法可以表示同样的数据，即建立一个单独的关系 *dept_year*(*dept_name*, *total_inst_2007*, *total_inst_2008*, *total_inst_2009*)。这里唯一的函数依赖是从 *dept_name* 指向其他属性的依赖，该关系也属于 BCNF。这种设计也是一个不好的想法，因为它存在与前面类似的问题，就是每年我们都不得不修改关系模式并书写新的查询。由于可能涉及很多属性，因此查询也会变得更加复杂。

像 *dept_year* 关系中那样的表示方法，属性的每一个值一列，叫作**交叉表**(crosstab)。它们在电子数据表、报告和数据分析工具中广泛使用。虽然这些表示方法显示给用户看是很有用的，但是由于上面给出的原因，它们在数据库的设计中并不可取。如 5.6.1 节所述，为了显示方便，SQL 扩展已经提出了将数据从规范化的关系表示转换到交叉表的方法。

8.9 时态数据建模

假定我们在大学中保存的数据不仅包括每个教师的地址，还包括该大学所知道的所有以前的地址。我们可能提出诸如"找出在 1981 年居住在普林斯顿的所有教师"的查询。在该情况下，我们可能对于每个教师有多个地址。每个地址有一个关联的起始日期和终止日期，表示该教师居住在该地址
364 的时间。对于终止日期，一个特殊的值，比如空值或者像 9999 - 12 - 31 这样的相当远的未来的值，可以用来表示教师仍然居住在该地址。

一般来说，**时态数据**(temporal data)是具有关联的时间段的数据，在时间段之间数据**有效**(valid)[⊖]。我们使用数据的**快照**(snapshot)这个术语来表示一个特定时间点上该数据的值。这样，*course* 数据的一张快照给出了在一个特定时间点上的所有课程的(诸如名称和系等)所有属性的值。

由于某些原因，对时态数据建模是一个相当有挑战性的问题。例如，假定我们有一个 *instructor* 实体，我们希望它关联着一个随时间变化的地址。为了给一个地址加上时态信息，我们不得不建立一个

⊖ 有些其他时态数据模型会区分**有效时间**(valid time)和**事务时间**(transaction time)，后者记录该事实记录到数据库中的时间。为简单起见，我们忽略这样的细节。

多值属性，其中的每个值是一个组合值(包含一个地址和一个时间段)。除了随时间变化的属性值，实体本身可能也要关联一个有效时间。例如，一个学生实体可能有一个从入学日期到毕业(或者离校)日期的有效时间。联系也可能有关联的有效时间。例如，联系 *prereq* 可以记录该课程成为另一门课程的先修课的时间。这样我们就需要在属性值、实体集和联系集上加上有效时间段。在 E-R 图上增加这些细节会使其非常难于创建和理解。当前已经有多个扩展 E-R 表示法的提案，希望用简单的方式指明属性值或联系是随时间变化的，但是至今仍没有采用的标准。

当我们随时间监测数据值时，我们假定成立的函数依赖，如：

$$ID \rightarrow street,\ city$$

可能不再成立。而以下约束(用自然语言表述)则可能成立："对任意给定的时间 t，一个教师 *ID* 仅有一个 *street* 和 *city* 的值。"

$X \xrightarrow{T} Y$ 在某个特定时间点上成立的函数依赖称为时态函数依赖。形式化地说，**时态函数依赖**(temporal functional dependency) $X \xrightarrow{T} Y$ 在关系模式 $r(R)$ 上成立的条件是，对于 $r(R)$ 的所有合法实例，r 的所有快照都满足函数依赖 $X \rightarrow Y$。

我们可以扩展关系数据库设计的理论，把时态函数依赖考虑进去。但是，使用常规的函数依赖已经很难于推理了，而且几乎没有设计者做好了处理时态函数依赖的准备。

在实践中，数据库设计者退回到更简单的方法来设计时态数据库。一个常用的方法是设计整个数据库(包括 E-R 设计和关系设计)而忽略时态改变(等价于仅考虑一张快照)。在这之后，设计者研究多 365
个关系并决定哪些关系需要跟踪时态变化。

接下来的步骤是通过把起始和终止时间作为属性，为每个这样的关系添加有效时间信息。例如，考虑 *course* 关系。课程的名称可能随时间变化，这可以通过添加一个有效时间范围来处理；得到的模式为

course (*course_id*, *title*, *dept_name*, *start*, *end*)

该关系的一个实例可能有两条记录(CS－101，"Introduction to Programming"，1985－01－01，2000－12－31)和(CS－101，"Introduction to C"，2001－01－01，9999－12－31)。如果更新该关系，将该课程名称改为"Introduction to Java"，时间"9999－12－31"就应该更新为原来的值("Introduction to C")有效的终止时间；然后加入包含新名称"Introduction to Java"和相应的起始时间的一条新元组。

如果另一个关系有一个参照时态关系的外码，数据库设计者必须决定参照是针对当前版本的数据还是一个特定时间点的数据。例如，我们可以通过扩展 *department* 关系来记录系的办公楼和预算随时间的变化，但是一个来自 *instructor* 或 *student* 关系的参照可能并不关心以往的办公楼和预算，而是可能隐含地参照对应 *dept_name* 的时态当前记录。另一方面，一个学生的成绩单记录应当参照该学生修习该课程期间的课程名称。在后一种情况下，参照关系必须也记录时间信息，以便于从 *course* 关系中确定一条特定的记录。在我们的例子中，选课时的 *year* 和 *semester* 可以映射到一个代表性的时间/日期值，比如该学期开学日的午夜；该时间/日期值用于从时态 *course* 关系中识别一条特定的记录，从中查询 *title*。

一个时态关系中原始的主码无法再唯一地标识一条元组。为了解决这个问题，我们可以在主码中加入起始和终止时间属性。但是，仍然留下一些问题：

- 有可能存储时间段重叠的数据，该问题主码约束无法避免。如果系统支持自带的有效时间类型，它就能够检测并避免这种重叠的时间段。
- 为了指明参照这种关系的外码，参照元组必须包含起始和终止时间属性为它们的外码的一部分，其值也必须与被参照元组相匹配。进一步说，如果被参照元组更新(而且原本在未来的终止时间也更新)，则该更新必须传播到所有参照元组。

  如果系统以一种更好的方式支持时态数据，我们就能够允许参照元组指定一个时间点而不是一个时间范围，并且依靠系统保证在被参照关系中存在一条元组，该元组的有效时间段包含该时间点。例如，一条成绩单记录可以指定一个 *course_id* 和一个时间(比如一个学期的开学日 366
  期)，这样就足够在 *course* 关系中识别正确的记录了。

作为一个常见的特例，如果所有对时态数据的参照都仅仅指向当前数据，更简单的解决办法是不在关系中添加时间信息，而是为过去的值创建一个相应的有时态信息的 *history* 关系。例如，在我们的大学数据库中，我们可以使用已经创建的设计，忽略时态变化，仅仅存储当前信息。所有的历史信息都转移到历史关系中。这样，*instructor* 关系可以只存储当前地址，而关系 *instructor_history* 可以包含 *instructor* 所有的属性，以及额外的 *start_time* 和 *end_time* 属性。

尽管我们没有提供任何处理时态数据的形式化方法，我们所讨论的问题和我们所给出的例子会帮助你设计记录时态数据的数据库。处理时态数据中的进一步的问题，包括时态查询，稍后将在 25.2 节讲述。

8.10 总结

- 我们给出了数据库设计中易犯的错误，和怎样系统地设计数据库模式来避免这些错误。这些错误包括信息重复和不能表示某些信息。
- 我们给出了从 E-R 设计到关系数据库设计的发展过程。什么时候可以安全地合并模式，什么时候应该分解模式。所有有效的分解都必须是无损的。
- 我们描述了原子域和第一范式的假设。
- 我们介绍了函数依赖的概念，并且使用它介绍两种范式，Boyce-Codd 范式(BCNF)和第三范式(3NF)。
- 如果分解是保持依赖的，则给定一个数据库更新，所有的函数依赖都可以由单独的关系进行验证，无须计算分解后的关系的连接。
- 我们展示了如何用函数依赖进行推导。我们着重讲了什么依赖是一个依赖集蕴涵的。我们还定义了正则覆盖的概念，它是与给定函数依赖集等价的最小的函数依赖集。
- 我们描述了一个将关系分解成 BCNF 的算法，有一些关系不存在保持依赖的 BCNF 分解。
- 我们用正则覆盖将关系分解成 3NF，它比 BCNF 的条件弱一些。属于 3NF 的关系也许会含有冗余，但是总存在保持依赖的 3NF 分解。 367
- 我们介绍了多值依赖的概念，它指明仅用函数依赖无法指明的约束。我们用多值依赖定义了第四范式(4NF)。附录 C.1.1 给出了有关多值依赖推理的细节。
- 其他的范式，例如 PJNF 和 DKNF，消除了更多细微形式的冗余。但是，它们难以操作而且很少使用。附录 C 给出了这些范式的详细信息。
- 回顾本章中的要点可以发现，我们之所以可以对关系数据库设计定义严格的方法，是因为关系数据模型建立在严谨的数学基础之上。这是关系模型与我们已经学过的其他数据模型相比的主要优势之一。

术语回顾

- E-R 模型和规范化
- 分解
- 函数依赖
- 无损分解
- 原子域
- 第一范式(1NF)
- 合法关系
- 超码
- R 满足 F
- F 在 R 上成立
- Boyce-Codd 范式(BCNF)
- 保持依赖
- 第三范式(3NF)
- 平凡的函数依赖
- 函数依赖集的闭包
- Armstrong 公理
- 属性集的闭包
- F 在 R_i 上的限定
- 正则覆盖
- 无关属性
- BCNF 分解算法
- 3NF 分解算法
- 多值依赖
- 第四范式(4NF)
- 多值依赖的限定
- 投影－连接范式(PJNF)
- 域－码范式(DKNF)
- 泛关系
- 唯一角色假设
- 大规范化

实践习题

8.1 假设我们将模式 $r(A, B, C, D, E)$ 分解为 368

$$r_1(A, B, C)$$
$$r_2(A, D, E)$$

证明该分解是无损分解，如果如下函数依赖集 F 成立：

$$A \rightarrow BC \qquad CD \rightarrow E \qquad B \rightarrow D \qquad E \rightarrow A$$

8.2 列出图 8-17 所示关系满足的所有函数依赖。

8.3 解释如何用函数依赖表明：

- 实体集 *student* 和 *instructor* 间存在的一对一联系集。
- 实体集 *student* 和 *instructor* 间存在的多对一联系集。

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_3

图 8-17 实践习题 8.2 的关系

8.4 用 Armstrong 公理证明合并律的正确有效性。（提示：使用增补律可证，若 $\alpha \rightarrow \beta$，则 $\alpha \rightarrow \alpha\beta$。再次使用增补律，利用 $\alpha \rightarrow \gamma$，然后使用传递律。）

8.5 用 Armstrong 公理证明伪传递律的正确有效性。

8.6 计算关于关系模式 $r(A, B, C, D, E)$ 的如下函数依赖集 F 的闭包。

$$A \rightarrow BC$$
$$CD \rightarrow E$$
$$B \rightarrow D$$
$$E \rightarrow A$$

列出 R 的候选码。

8.7 用习题 8.6 中的函数依赖计算正则覆盖 F_c。 369

8.8 考虑图 8-18 中计算 α^+ 的算法。证明该算法比图 8-8（8.4.2 节）中提出的算法更高效，并且能正确计算 α^+。

```
result := ∅;
/* fdcount 是一个数组，它的第 i 个元素包含即将属于 α+ 的第 i 个函数依赖左半部的属性个数 */
for i := 1 to |F| do
    begin
        令 β → γ 表示第 i 个函数依赖;
        fdcount[i] := |β|;
    end
/* appears 是一个数组，每个属性对应数组的一个入口，属性 A 的入口是一列整数，该列中每个整数 i 指明 A 出现在
   第 i 个函数依赖的左半部 */
for each 属性 A do
    begin
        appears[A] := NIL;
        for i := 1 to |F| do
            begin
                令 β → γ 表示第 i 个函数依赖;
                if A ∈ β then 把 i 加到 appears[A] 中;
            end
    end
addin(α);
return(result);

procedure addin(α);
for each α 中的属性 A do
    begin
        if A ∉ result then
            begin
                result := result ∪ {A};
                for each appears[A] 的元素 i do
                    begin
                        fdcount[i] := fdcount[i] - 1;
                        if fdcount[i] = 0 then
                            begin
                                令 β → γ 表示第 i 个函数依赖;
                                addin(γ);
                            end
                    end
            end
    end
```

370 ~ 371

图 8-18 一个计算 α^+ 的算法

8.9 给定数据库模式 $R(a, b, c)$ 及模式 R 上的关系 r，写出检验函数依赖 $A \rightarrow c$ 是否在关系 r 上成立的 SQL 查询。并写出保证函数依赖的 SQL 断言。假设不存在空值。（虽然 SQL 标准中的某些部分，诸如断言等目前还没有在任何数据库实现中得到支持。）

8.10 我们对无损连接分解的讨论隐含假设了函数依赖左部的属性不能为空值。如果违反了这个性质，在分解中可能会出现什么错误？

8.11 在 BCNF 分解算法中，假设你用一个函数依赖 $\alpha \rightarrow \beta$ 将关系模式 $r(\alpha, \beta, \gamma)$ 分解为 $r_1(\alpha, \beta)$ 和 $r_2(\alpha, \gamma)$。

a. 你预期在分解后的关系上有什么样的主码和外码约束成立？

b. 如果在上述分解后的关系上没有强制实施外码约束，给出一个由于错误更新而导致不一致的例子。

c. 当用 8.5.2 节中的算法将一个关系分解为 3NF 时，你预期将有什么样的主码和外码依赖在分解后的模式上成立？

8.12 令 $R_1, R_2, \cdots, R_n$ 为模式 U 的分解。令 $u(U)$ 为一个关系，并且令 $r_i = \Pi_{R_i}(u)$。

证明

$$u \subseteq r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

8.13 证明实践习题 8.1 中的分解不是保持依赖的分解。

8.14 证明有可能通过保证至少有一个模式包含被分解模式的候选码来确保一个保持依赖的 3NF 模式分解是一个无损分解。（提示：证明到分解后模式上的所有的投影的连接不会具有比原关系多的元组。）

8.15 给出一个关系模式 R' 及函数依赖集 F' 的例子，使得至少有三种不同的无损分解可将 R' 转化为 BCNF。

8.16 令主(prime)属性为至少在一个候选码中出现的属性。令 α 和 β 为属性集，使得 $\alpha \rightarrow \beta$ 成立但 $\beta \rightarrow \alpha$ 不成立。令 A 为一属性，它不属于 α 或 β，并且 $\beta \rightarrow A$ 成立。我们称 A 传递依赖(transitively dependent)于 α。我们可以重新如下定义 3NF：具有函数依赖集 F 的关系模式 R 属于 3NF，如果 R 中没有非主属性 A 传递依赖于 R 的一个码。证明这个新定义等价于原定义。

8.17 函数依赖 $\alpha \rightarrow \beta$ 称为**部分依赖**(partial dependency)的条件是，存在 α 的真子集 γ 使得 $\gamma \rightarrow \beta$。我们称 β 部分依赖于 α。关系模式 R 属于**第二范式**(Second Normal Form，2NF)，如果 R 中的每个属性 A 都满足如下准则之一：

- 它出现在一个候选码中。
- 它没有部分依赖于一个候选码。

证明每个 3NF 模式都属于 2NF。（提示：证明每个部分依赖都是传递依赖。）

8.18 给出一个关系模式 R 和依赖集的例子，使得 R 属于 BCNF，但不属于 4NF。

习题

8.19 给出实践习题 8.1 中模式 R 的一个无损连接的 BCNF 分解。

8.20 给出实践习题 8.1 中模式 R 的一个无损连接并保持依赖的 3NF 分解。

8.21 将具有如下约束的下列模式规范化为 4NF。

books (*accessionno*, *isbn*, *title*, *author*, *publisher*)
users (*userid*, *name*, *deptid*, *deptname*)
accessionno → *isbn*
isbn → *title*
isbn → *publisher*
isbn →→ *author*
userid → *name*
userid → *deptid*
deptid → *deptname*

372 8.22 解释什么是信息重复和无法表示信息。解释为什么这些性质可能意味着不好的关系数据库设计。

8.23 为什么某些函数依赖称为平凡的函数依赖？

8.24 使用函数依赖的定义论证 Armstrong 公理(自反律、增补律、传递律)是正确有效的。

8.25 考虑下面提出的函数依赖规则：若 $\alpha \rightarrow \beta$ 且 $\gamma \rightarrow \beta$，则 $\alpha \rightarrow \gamma$。通过给出一个关系 r，满足 $\alpha \rightarrow \beta$ 且 $\gamma \rightarrow \beta$ 但不满足 $\alpha \rightarrow \gamma$，证明该规则不是正确有效的。

8.26 用 Armstrong 公理证明分解律的正确有效性。

8.27 用实践习题 8.6 中的函数依赖计算 B^+。

8.28 证明实践习题 8.1 中的模式 R 的如下分解不是无损分解：

$$(A, B, C)$$
$$(C, D, E)$$

提示：给出模式 R 上一个关系 r 的例子，使得

$$\Pi_{A,B,C}(r) \bowtie \Pi_{C,D,E}(r) \neq r$$

8.29 考虑如下关系模式 $r(A, B, C, D, E, F)$ 上的函数依赖集 F：

$$A \rightarrow BCD$$
$$BC \rightarrow DE$$
$$B \rightarrow D$$
$$D \rightarrow A$$

a. 计算 B^+。

b. (使用 Armstrong 公理)证明 AF 是超码。

c. 计算上述函数依赖集 F 的正则覆盖；给出你的推导的步骤并解释。

d. 基于正则覆盖，给出 r 的一个 3NF 分解。

e. 利用原始的函数依赖集，给出 r 的一个 BCNF 分解。

f. 你能否利用正则覆盖得到与上面的 r 相同的 BCNF 分解？

8.30 列出关系数据库设计的三个目标，并解释为什么要达到每个目标。 373

8.31 在关系数据库设计中，为什么我们有可能会选择非 BCNF 设计？

8.32 给定关系数据库设计的三个目标，有没有理由设计一个属于 2NF 但不属于任何一个更高范式的数据库模式？(2NF 的定义参见实践习题 8.17。)

8.33 给定一个关系模式 $r(A, B, C, D)$，$A \rightarrow\!\!\!\!\rightarrow BC$ 是否是能蕴涵 $A \rightarrow\!\!\!\!\rightarrow B$ 和 $A \rightarrow\!\!\!\!\rightarrow C$？如果是请证明之，否则给出一个反例。

8.34 解释为什么 4NF 是一个比 BCNF 更好的范式。

文献注解

对于关系数据库设计理论最初的讨论是在 Codd[1970]这篇早期论文中。在那篇论文中，Codd 引入了函数依赖，以及第一、第二和第三范式。

Armstrong 公理是由 Armstrong[1974]引入的。在 20 世纪 70 年代后期关系数据库理论有了显著的进展，这些结果收集在一些数据库理论的课本中，包括 Maier[1983]、Atzeni 和 Antonellis[1993]以及 Abiteboul 等[1995]。

BCNF 是在 Codd[1972]中引入的。Biskup 等[1979]给出了用于找到无损并保持依赖的 3NF 分解的算法。有关无损分解特性的基础结论在 Aho 等[1979a]中给出。

Beeri 等[1977]给出了一组多值依赖的公理，并证明这些公理是正确有效且完备的。4NF、PJNF 和 DKNF 的概念分别来自 Fagin[1977]、Fagin[1979]和 Fagin[1981]。关于规范化的进一步的参考文献，请参看附录 C 中的文献注解。

Jensen 等[1994]给出了一个时态数据库概念的术语表。Gregersen 和 Jensen[1999]给出了一个对 E-R 建模进行扩展以处理时态数据的综述。Tansel 等[1993]探讨了时态数据库的理论、设计和实现。Jensen 等[1996]描述了将依赖理论扩展至时态数据的方法。 374

第9章
Database System Concepts, 6E

应用设计和开发

实际上，对数据库的所有应用都发生在应用程序内。相应地，几乎所有的用户与数据库之间的交互都是通过应用程序间接发生的。因此毫不奇怪，数据库系统长期以来都支持诸如表单和 GUI 构建器等工具，用于快速开发与用户交互的应用程序。近几年来，Web 已成为最广泛使用的数据库用户界面。

在本章中，我们学习建立应用程序所需要的工具和技术，集中关注那些使用数据库存储数据的交互式应用程序。

在 9.1 节对应用程序和用户界面的介绍后，我们集中关注基于 Web 界面的应用的开发。首先 9.2 节对 Web 技术进行概述，然后 9.3 节讨论广泛用于构建 Web 应用的 Java Servlet 技术。9.4 节简要介绍 Web 应用程序体系架构。9.5 节讨论快速应用程序开发工具，而 9.6 节介绍构建大型 Web 应用程序中的性能问题。9.7 节讨论应用程序安全的问题。用 9.8 节结束本章，其中对加密及它在应用程序中的使用进行了介绍。

9.1 应用程序和用户界面

尽管许多人和数据库打交道，但很少有人用查询语言直接跟数据库系统交互。用户与数据库交互时最常用的方法是通过一个应用程序，它在前端提供用户界面，并在后端提供数据库接口。这种应用程序通常是通过一个基于表格的界面从用户得到输入，然后根据用户输入或将数据输入数据库或从数据库中抽取信息，并产生输出展示给用户。

375 举个应用程序的例子，考虑一个大学注册系统。和其他这种应用程序类似，注册系统首先需要你标识并认证你自己，通常使用用户名和密码。然后，应用程序使用你的身份从数据库中抽取信息，比如你的姓名和你已注册的课程，并展示该信息。应用程序提供了若干接口，让你可以注册课程及查询多种其他的信息，比如课程信息和教师信息。组织机构使用这种应用程序以自动完成多种任务，例如出售、购买、会计和薪资、人力资源管理以及库存管理等。

应用程序有可能正在使用，即使不能明显地看出来它们正在使用。例如，一个新闻网站可以提供对个人用户透明定制的页面，即使该用户在和网站交互时并没有显式填写任何表单。为了做到这点，它实际上运行了一个对每个用户生成定制页面的应用程序，例如，定制可以基于用户浏览文章的历史。

一个典型的应用程序包括一个处理用户界面的前端部分，一个和数据库通信的后端部分，以及一个包含"业务逻辑"的中间层，即执行特定的信息请求或更新的代码，强制执行诸如为了执行给定任务应该采取什么行动，或者谁可以执行什么任务的业务规则。

如图 9-1 所示，应用程序体系架构随时间而演变。诸如航班预订的应用程序自 20 世纪 60 年代以来一直在使用。在计算机应用早期，应用程序运行在一台大的"主机"计算机上，用户通过终端与应用程序交互，其中一些终端甚至支持表格。

随着个人计算机使用的普及，许多机构对内部应用程序使用了一个不同的体系架构，其中应用程序在用户的计算机上运行并访问中央数据库。这个架构通常称作"客户 - 服务器"架构，它允许创建强大的图形用户界面，这是早期基于终端的应用所不支持的。然而，软件必须安装在每个用户的机器上以运行应用程序，这就使得诸如升级的任务比较困难。即使在客户 - 服务器架构流行的个人计算机时
376 代，诸如航班预订这种从大量地理上分布的位置使用的应用程序仍旧选择主机架构。



图 9-1 不同时期的应用架构

在过去的 15 年中，Web 浏览器已成为数据库应用程序的通用前端，它通过互联网连接后端。浏览器使用一种标准的语法，超文本标记语言(HyperText Markup Language，HTML)标准，它既支持信息的格式化显示，又支持基于表格的界面的创建。HTML 标准是独立于操作系统或浏览器的，并且目前几乎每台计算机都安装了 Web 浏览器。因此一个基于 Web 的应用程序可被任何一台连接到互联网的计算机访问。

与客户－服务器架构不同，要使用基于 Web 的应用程序不需要在客户机上安装任何特定于应用的软件。然而，支持远超过普通 HTML 所能提供的功能的复杂用户界面在目前广泛使用，并且是用大多数 Web 浏览器都支持的脚本语言 JavaScript 构建的。与用 C 写的程序不同，JavaScript 程序可以运行在安全模式下，保证它们不会导致安全问题。JavaScript 程序被透明地下载到浏览器上，且不需要用户的计算机上任何显式的软件安装。

Web 浏览器提供用户交互前端的同时，应用程序构成了后端。通常，来自浏览器的请求发送给 Web 服务器，它依次执行应用程序以处理请求。有多种技术都可以用于创建运行在后端的应用程序，包括 Java servlet，Java Server Pages(JSP)、Active Server Pages(ASP)，或者脚本语言，比如 PHP、Perl 或 Python。

本章其余部分讲述如何构建这些应用。首先介绍用于构建 Web 界面的 Web 技术和工具及构建应用程序的技术，然后介绍应用架构，以及应用程序构建中的性能和安全问题。

9.2 Web 基础

在本节中，我们为了那些不熟悉 Web 底层技术的读者，回顾一下万维网背后的一些基本技术。

9.2.1 统一资源定位符

统一资源定位符(Uniform Resource Locator，URL)是 Web 上每个可访问文档在全球唯一的名字。URL 的一个例子如下：

http://www.acm.org/sigmod

URL 的第一部分表明文档如何被访问："http"表明文档将用**超文本传输协议**(HyperText Transfer Protocol，HTTP)访问，这是一个传输 HTML 文档的协议。第二部分给出一台具有 Web 服务器的机器的 377
名称。URL 的其余部分是文件在机器上的路径名，或者文档在机器中其他的唯一标识符。

URL 可以包含位于 Web 服务器上程序的标识符，以及传递给程序的参数。这样的 URL 的一个例子是：

http://www.google.com/search?q=silberschatz

它指出，www.google.com 服务器上的程序 search 应该执行，并带有参数 q = silberschatz。接收到这个 URL 请求时，Web 服务器使用给定的参数执行该程序。该程序返回一个 HTML 文档给 Web 服务器，它将该文档传送到前端。

9.2.2 超文本标记语言

图 9-2 是一个使用 HTML 格式表示的表的例子，而图 9-3 表示浏览器根据 HTML 表示的表而产生的显示图像。HTML 源文本显示了一些 HTML 标签。每个 HTML 页应包含在 html 标签内，该页的主体包

含在 body 标签中。表用 table 标签表示，其中包含的行用 tr 标签表示。表的表头行有用标签 th 说明的表单元，而普通行有用标签 td 说明的表单元。这里不再对标签进行更详细的介绍。关于 HTML 的更多信息，请参看文献注解中的参考文献。

图 9-4 展示了如何表示一个允许用户从菜单中选择人员类型（学生或教师）以及允许用户向文本框中输入一个数的 HTML 表单。图 9-5 展示了以上表单如何在 Web 浏览器中显示。表单中表示了两种接受输入的方式，但是 HTML 还支持几种其他的输入方式。form 标签的 action 属性表示当提交表单时（通过点击提交按键），表单数据应发送到 URL PersonQuery（该 URL 是相对于页面的 URL）。Web 服务器被设置成当该 URL 被访问时，便调用相应的带有用户提供的参数值 persontype 和 name（在 select 和 input 域中指定）的应用程序。该应用程序生成一个 HTML 文档，该文档随即传送回去并显示给用户；我们将在本章的后面看到如何创建这种程序。

```
<html>
<body>
<table border>
<tr> <th>ID</th>  <th>Name</th>  <th>Department</th> </tr>
<tr> <td>00128</td> <td>Zhang</td>  <td>Comp. Sci.</td> </tr>
<tr> <td>12345</td> <td>Shankar</td> <td>Comp. Sci.</td> </tr>
<tr> <td>19991</td> <td>Brandt</td>  <td>History</td> </tr>
</table>
</body>
</html>
```

图 9-2 HTML 格式的表格数据

ID	Name	Department
00128	Zhang	Comp. Sci.
12345	Shankar	Comp. Sci.
19991	Brandt	History

图 9-3 图 9-2 中 HTML 源文本的显示

```
<html>
<body>
<form action="PersonQuery" method=get>
Search for:
<select name="persontype">
    <option value="student" selected>Student </option>
    <option value="instructor"> Instructor </option>
</select> <br>
Name: <input type=text size=20 name="name" >
<input type=submit value="submit" >
</form>
</body>
</html>
```

图 9-4 一个 HTML 表单

HTTP 定义了两种将用户在 Web 浏览器端输入的值发送到 Web 服务器的方式。get 方法将值编码为 URL 的一部分。例如，如果 Google 搜索页使用一个表单，包含一个名为 q 的 get 方法的输入参数，并且用户键入字符串"silberschatz"并提交该表单，浏览器将会向 Web 服务器请求如下的 URL：

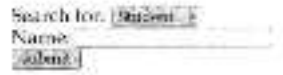


图 9-5 图 9-4 中 HTML 源文本的显示

http://www.google.com/search?q=silberschatz

而 post 方法将会向 www.google.com 页面发送一个请求，并将参数的值作为 Web 服务器和浏览器间交换的 HTTP 协议的一部分发送。图 9-4 中的表单表示该表单使用 get 方法。

尽管 HTML 代码可以用普通的文本编辑器创建，但是存在若干种可以使用图形界面直接创建 HTML 文本的编辑器。这些编辑器允许从菜单中选择诸如表单、菜单和表这样的结构加入到 HTML 文档中，而不是手工输入生成这些结构的代码。

HTML 支持样式表(stylesheet)，它可以改变如何显示 HTML 格式化结构的默认定义，以及其他显示属性，如页面的背景颜色等。层叠式样式表(Cascading StyleSheet，CSS)标准允许同一个样式表用于多个 HTML 文档，使一个 Web 站点上的所有页面具有独特而统一的样式。

9.2.3 Web 服务器和会话

Web 服务器(Web Server)是运行于服务器上的程序，它接收来自于 Web 浏览器的请求并以 HTML 文档的形式返回结果。浏览器和 Web 服务器通过 HTTP 通信。Web 服务器提供了强大的功能，而不只是简单的文档传输。最重要的功能是具有带用户提供的参数执行程序，并以 HTML 文档形式传送回结果的能力。

所以，Web 服务器可以作为中介接受对多种信息服务的访问。一个新的服务可以通过创建并安装提供服务的应用程序而创建。公共网关接口(Common Gateway Interface，CGI)标准定义了 Web 服务器如何与应用程序通信。应用程序通常通过 ODBC、JDBC 或者其他协议与数据库服务器通信，以获取或存储数据。

图 9-6 显示了一个使用三层体系结构搭建的 Web 应用，包括一个 Web 服务器、一个应用服务器和一个数据库服务器。使用多级服务器增加了系统的开销；CGI 接口为每个请求都开启一个新的进程为之服务，这导致其至更大的开销。

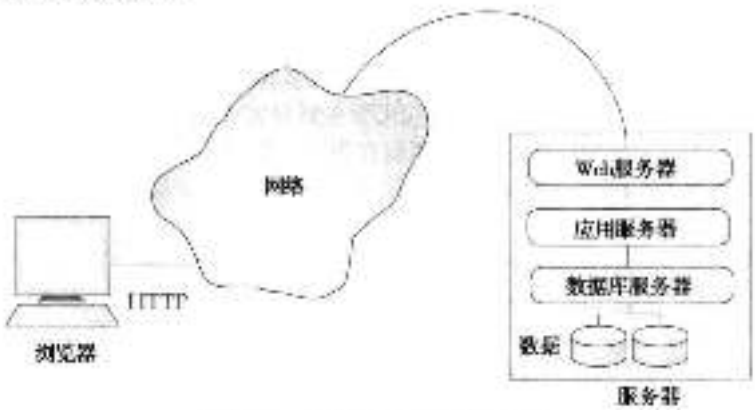


图 9-6 三层 Web 应用体系结构

因此目前大多数 Web 应用使用一种两层的 Web 应用体系结构，其中应用程序运行在 Web 服务器中，如图 9-7 所示。下面的章节将更加详细地研究基于两层体系结构的系统。

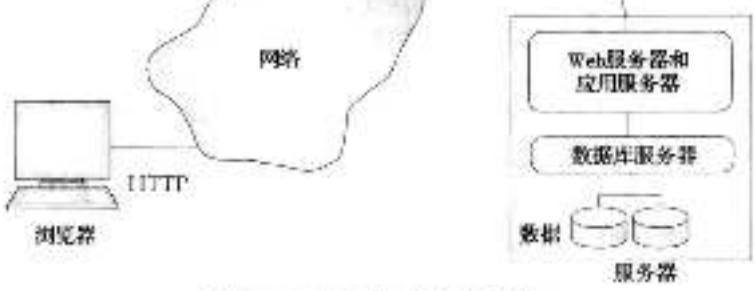


图 9-7 两层 Web 应用体系结构

在客户端与 Web 服务器之间不存在持续的连接；当 Web 服务器接收到一个请求时，临时创建一个连接以发送请求并接收来自 Web 服务器的响应。但是该连接可能会关闭，且下一个请求可以生成一个

新的连接。与此相反，当一个用户登录计算机，或是使用 ODBC 或 JDBC 连接到数据库时，会创建一个会话，且会话信息保留在服务器和客户端，直到会话结束——信息包括诸如该用户的用户标识符和用户已设置的会话参数等。HTTP 协议是**无连接**(connectionless)的一个重要原因在于，大多数计算机能同时容纳的连接数目是有限的。如果 Web 中大量的站点对单台服务器建立连接，就会超过限制，拒绝后续用户的服务。而使用无连接协议，当满足了请求时连接就可以马上断开，为其他请求[⊖]留出可用连接。

但是，大多数 Web 应用需要会话信息，以允许有意义的用户交互。例如，应用程序通常限制对信息的访问，且因此需要认证用户。每次会话应进行一次认证，会话中进一步的交互应不需重新认证。

380 | 381

尽管连接会关闭，但为了实现会话，需要在客户端存储额外的信息，并随会话中的每个请求返回；服务器使用这个信息来辨别出请求是用户会话的一部分。关于会话的额外信息同样必须在服务器端维护。

这种额外信息通常以**网络跟踪器**(cookie)的形式保存在客户端；简单来说，一个 cookie 是一小段包含标识信息的文本，并关联一个名字。例如，google.com 可能设置一个名为 prefs 的 cookie，它包含用户设置的偏好信息，比如语言偏好和每页显示的结果数目。对于每个搜索请求，google.com 都能从用户的浏览器中得到这个名为 prefs 的 cookie，然后根据指定的偏好显示结果。一个域(Web 站点)只能获取它自己设置的 cookie，而不能得到别的站点设置的 cookie，而且 cookie 名可以跨域复用。

为了跟踪用户会话的目的，应用程序可能会产生一个会话标识符(通常是一个当前没有用作会话标识符的随机数)，然后发送一个包含这个会话标识符的名为(比如)seessionid 的 cookie。该会话标识符也保存在服务器本地。当一个请求进来时，应用服务器向客户端请求名为 sessionid 的 cookie。如果客户端没有存储该 cookie，或者返回的值不是当前在服务器中记录的有效会话标识符，应用程序就认为该请求不是当前会话的一部分。如果 cookie 的值与一个存储的会话标识符匹配，则该请求就被识别为一个进行中的会话的一部分。

如果一个应用需要安全地鉴别用户，则它只能在对用户认证之后设置 cookie；例如，用户只有在提交了有效的用户名和密码之后才能通过认证。[⊜]

对于不需要高安全性的应用，比如公开的新闻站点，cookie 可以永久存储在浏览器和服务器中；它们识别出用户对同一个网站的后续访问，而不需要输入任何认证信息。对于要求更高安全性的应用，服务器可能会在有效期过后或用户注销时使会话无效(关闭)。(通常用户通过点击一个注销按钮来注销，它会提交一个注销表单，其动作就是使当前会话失效。)使一个会话失效仅仅是在应用服务器中的活动会话列表里丢弃该会话标识符。

382

9.3 servlet 和 JSP

在两层 Web 体系结构中，应用程序作为 Web 服务器本身的一部分运行。实现这种体系结构的一个方法是将 Java 程序加载到 Web 服务器中。Java servlet 规格说明定义了一种 Web 服务器与应用程序间通信的应用程序编程接口；Java 中的 HttpServlet 类实现了 servlet API 规格要求；实现特定功能的 servlet 类被定义为这个类的子类。[⊜]通常 servlet 一词指实现了 servlet 接口的 Java 程序(和类)。图 9-8 是一个 servlet 的例子；我们简短地介绍它的一些细节。

当服务器启动或者服务器接收到远程的 HTTP 请求执行某个特定 servlet 时，servlet 的代码被加载到

⊖ 为了性能方面的原因，连接可能会在短时间内保持，以允许于请求复用该连接。然而，不能保证连接将会保持，因此在设计应用时应当假设一旦处理请求连接就有可能关闭。

⊜ 用户标识符可以存储在客户端，比如一个叫做 userid 的 cookie 中。这样的 cookie 可以用于低安全要求的应用中，比如免登的 Web 站点识别它们的用户。但是，对于要求更高级别安全性的应用，这样的方法就会造成安全隐患：cookie 中的值可能在浏览器中被用户恶意篡改，该用户就能够伪装成另一个用户。将 cookie(比如名为 sessionid)设置为一个随机产生的会话标识符(一个很大的数字空间)，可以使得一个用户冒充另一个用户的可能性变得极小。而一个顺序生成的会话标识符，则很容易冒充。

⊜ 虽然我们的例子中只使用了 HTTP，但是 servlet 接口也能支持非 HTTP 请求。

Web服务器中。servlet的任务就是处理这种可能涉及访问数据库提取必要信息，并动态生成HTML页面返回给客户端浏览器的请求。

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class PersonQueryServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HEAD><TITLE> Query Result</TITLE></HEAD>");
        out.println("<BODY>");

        String persontype = request.getParameter("persontype");
        String number = request.getParameter("name");
        if(persontype.equals("student")) {
            … 寻找具有指定姓名的学生的代码 …
            … 使用JDBC与数据库通信 …
            out.println("<table BORDER COLS=3>");
            out.println(" <tr> <td>ID</td> <td>Name: </td>" +
                  " <td>Department</td> </tr>");
            for(... each result ...){
                … 获取 ID、name 和 dept_name
                … 放入变量 ID、name 和 deptname
                out.println(" <tr> <td>" + ID + "</td>" +
                      "<td>" + name + "</td>" +
                      "<td>" + deptname + "</td></tr>");
            };
            out.println("</table>");
        }
        else {
            … 同上，但是是对于教师 …
        }
        out.println("</BODY>");
        out.close();
    }
}
```

图9-8 servlet代码示例

9.3.1 一个servlet的例子

servlet通常用于对HTTP请求动态生成响应。它们可以访问HTML表单提供的输入，执行"业务逻辑"以决定给出什么样的响应，然后生成HTML输出并发送回浏览器。

图9-8所示为实现图9-4中表单的servlet代码的例子。这个servlet叫做PersonQueryServlet，同时表单指明了action = "PersonQuery"。必须告知Web服务器这个servlet是用来处理PersonQuery请求的。该表单指定使用HTTP的get机制进行参数传递，这样servlet代码中定义的doGet()方法被调用。

每次请求都会生成一个新的线程，在线程中执行调用，因此多个请求就可以并行处理。任何来自Web网页上的表单菜单和输入域中的值，和cookie一样，由一个为请求创建的HttpServletRequest类对象传入，然后请求的应答由一个HttpServletResponse类对象返回。

该例子中的doGet()方法通过request.getParameter()抽取参数的type和number的值，并使用它们的值执行数据库查询。用于访问数据库以及从查询结果获取属性值的代码没有表示出来，有关如何使用JDBC访问数据库的细节请参考5.1.1.4节。servlet代码将查询结果输出到HttpServletResponse类的对象

response 中返回给请求者。将结果输出到 response 是通过首先从 response 中取得 PrintWriter 对象 out，然后向 out 以 HTML 格式输出结果而实现的。

9.3.2 servlet 会话

回想一下，浏览器和 Web 服务器之间的交互是无状态的。也就是说，每次浏览器向服务器发出一个请求，浏览器都要连接服务器，请求某些信息，然后从服务器断开连接。cookie 可以用来识别一个请求与前一个请求是来自于同一个浏览器会话。但是，cookie 构成一个低级别机制，程序员需要一个更好的抽象概念来处理会话。

383
384

servlet API 提供了一种跟踪会话并存储会话相关信息的方法。HttpServletRequest 类的 getSession(false)方法的调用获取出发出请求的浏览器对应的 HttpSession 对象。参数值为 true 将表示当请求是新请求时，必须创建新的会话对象。

当 getSession()方法被调用时，服务器会先要求客户端返回具有指定名字的 cookie。如果客户没有该名字的 cookie，或者返回的值与任何进行中的会话都不匹配，则请求不是进行中的会话的一部分。在这种情况下，getSession()将返回一个空值，并且 servlet 会将用户指引到一个登录页面。

登录页面可以允许用户提供用户名和密码。登录页面对应的 servlet 可以验证密码与用户匹配(例如，通过在数据库中查找认证信息)。如果该用户正常通过认证，登录 servlet 将会执行 getSession(true)，返回一个新的会话对象。为了创建一个新的会话，Web 服务器内部执行以下任务：在客户的浏览器中设置一个 cookie(比如名为 sessionId)，用会话标识符作为它所关联的值，创建一个新的会话对象，并将会话标识符的值与会话对象关联。

servlet 代码还能够在 HttpSession 对象中存储和查找(属性名，值)对，以便在一个会话内的多个请求间维持状态。例如，在用户被认证并创建了会话对象之后，登录 servlet 可以通过在 getSession()所返回的会话对象上执行方法

```
session.setAttribute("userId", userId)
```

将用户的 user-id 存储为会话的一个参数；假定在 Java 变量 userid 中包含用户标识符。

如果请求属于一个进行中的会话，浏览器就会返回该 cookie 的值，且 getSession()会返回对应的会话对象。servlet 通过在上述所返回的会话对象上执行方法

```
session.getAttribute("userId")
```

就能够从会话对象中获取会话参数，比如 user-id。如果属性 userid 没有设置，该函数会返回一个空值，表明该客户端的用户还没有通过认证。

9.3.3 servlet 的生命周期

385

servlet 的生命周期由它部署于的 Web 服务器控制。当有客户端请求一个特定的 servlet 时，服务器首先检查是否存在该 servlet 的实例。如果不存在，Web 服务器就将 servlet 类加载到 Java 虚拟机(Java Virtual Machine，JVM)中，并创建该 servlet 类的一个实例。另外，服务器调用 init()方法将该 servlet 实例初始化。注意，每个 servlet 实例仅在加载的时候初始化一次。

在确定 servlet 实例的确存在之后，服务器调用 servlet 的 service()方法，并以一个 request 对象和一个 response 对象作为参数。在默认情况下，服务器创建一个新的线程以执行 service()方法；因此，对一个 servlet 的多个请求就能够并行处理，而不必等待上一个请求执行完。service()方法适当调用 doGet()或 doPost()方法。

当不再需要的时候，可以通过调用 destroy()方法停止一个 servlet。服务器可以设置为如果在一段时间内没有对某个 servlet 的请求，则自动停止这个 servlet；该时间段是服务器的一个参数，可以根据应用适当地设置。

9.3.4 servlet 支持

许多应用服务器都提供 servlet 的内嵌支持。最流行的应用服务器之一是 Apache Jakarta 项目中的 Tomcat 服务器。其他支持 servlet 的应用服务器包括 Classfish、JBoss、BEA Weblogic 应用服务器、Oracle 应用服务器以及 IBM 的 WebSphere 应用服务器。

开发 servlet 应用程序最好的方式是利用一个 IDE，比如 Tomcat 或 Glassfish 服务器内嵌的 Eclipse 或者 NetBeans。

除了基本的 servlet 支持之外，应用服务器通常还提供多种有用的服务。它们允许应用程序部署或者停止，并提供监视应用服务器状态的功能，包括性能统计。如果一个 servlet 文件修改，某些应用服务器能够监测到，然后透明地重新编译并重新加载该 servlet。许多应用服务器还允许服务器在多台机器上并行运行以提高性能，并将请求分发至适当的机器上。许多应用服务器还支持 Java 2 Enterprise Edition(J2EE)平台，它针对多种任务都提供支持和 API，比如处理对象、跨多个应用服务器并行处理以及处理 XML 数据(XML 将在后面第 23 章介绍)。

9.3.5 服务器端脚本

用一种诸如 Java 或 C 的程序设计语言编写一段甚至很简单的 Web 应用程序也是很花时间的一项任务，它需要很多行代码以及熟悉该语言中错综复杂的事物的程序员。另一种方法，也就是**服务器端脚本**(server-side scripting)，为创建许多应用程序提供了一种简单得多的方式。脚本语言提供了可嵌入 HTML 文档中的结构。在服务器端脚本中，服务器会在传送 Web 页面之前执行嵌入在 HTML 页面内容中的脚本。每段脚本在执行时会生成加入到页面中的文本(或可能甚至从页面中删除内容)。脚本的源代码将从页面中删除，因此客户端可能根本没察觉页面原先含有任何代码。执行的脚本可能包含数据 [386]
库上执行的 SQL 代码。

一些广泛使用的脚本语言包括 Sun 的 Java Server Pages(JSP)、Microsoft 的 Active Server Pages(ASP)和后续的 ASP.NET、PHP 脚本语言、ColdFusion 标记语言(ColdFusion Markup Language，CFML)以及 Ruby on Rails。许多脚本语言也允许用诸如 Java、C#、VBScript、Perl 以及 Python 语言写的代码嵌入到 HTML 页面中或被 HTML 页面调用。例如，JSP 允许 Java 代码嵌入 HTML 页面中，而 Microsoft 的 ASP.NET 和 ASP 支持内嵌的 C#和 VBScript。这些语言中的大部分都带有库和工具，它们一起构成了 Web 应用开发的框架。

接下来简要介绍 **Java Server Pages(JSP)**，一种允许 HTML 程序员将静态的 HTML 和动态生成的 HTML 混合在一起的脚本语言。其动机在于，对于许多动态的 Web 页面，其中大多数的内容仍然是静态的(也就是说，不论该页面何时生成，总是显示相同的内容)。Web 页面的动态内容(比如，根据表单的参数生成)通常是页面的一小部分。通过写 servlet 代码来创建这样的页面导致大量的 HTML 被写作 Java 字符串。JSP 则允许 Java 代码嵌入静态的 HTML 中；嵌入的 Java 代码生成该页面的动态部分。JSP 脚本实际上转换为 servlet 代码进行编译，但是应用程序员却从撰写大量 Java 代码以创建 servlet 的困境中解脱出来了。

图 9-9 所示为一个包含了内嵌的 Java 代码的 JSP 页面的源代码文本。脚本中的 Java 代码用 <% …% > 括起来以区别于周围的 HTML 代码。代码使用 request.getParameter( ) 以获取属性 name 的值。

```
<html>
<head> <title> Hello </title> </head>
<body>
 <% if(request.getParameter("name") == null)
   { out.println("Hello World"); }
   else { out.println("Hello, " + request.getParameter("name")); }
 %>
</body>
</html>
```

图 9-9 一个包含 Java 代码的 JSP 页面

当浏览器请求一个 JSP 页面时，应用服务器根据该页面生成 HTML 输出，并发送回浏览器。JSP 页面中的 HTML 部分正如所输出的。[令] 在 HTML 输出中，在 <% …% > 中嵌入的所有 Java 代码都用它向 [387]
out 对象输出的文本代替。在图 9-9 的 JSP 代码中，如果没有值输入给表单参数 name，则脚本输出

令 JSP 允许一种更复杂的嵌入，即 HTML 代码在 Java 的 if-else 语句内部，它根据一条件等于真或非真而有条件地输出到输出。我们在这里省略细节。

"Hello World"；如果输入了一个值，则脚本输出"Hello"，后面跟着名字。

PHP

PHP 是一种广泛用于服务器端脚本的脚本语言。PHP 代码可以像 JSP 中的方式一样与 HTML 混合使用。字母串"<? php"表示 PHP 代码的开始，字母串"? >"表示 PHP 代码的结束。下列代码与图 9-9 中的 JSP 代码执行相同的动作。

```
<html>
<head> <title> Hello </title> </head>
<body>
 <?php if (isset($_REQUEST['name']))
  { echo "Hello World"; }
  else { echo 'Hello,' . $_REQUEST['name']; }
 ?>
</body>
</html>
```

数组 $_REQUEST 包含请求的参数。注意，数组用参数名索引；在 PHP 中数组可以用任意字符串索引，而不仅是数字。函数 isset( ) 检查数据元素是否已初始化。echo( ) 函数将它的参数输出到 HTML。两个字符串间的操作符"."将字符串连起来。

一个适当配置的 Web 服务器将把任意以".php"结尾的文件解释为 PHP 文件。如果请求该文件，Web 服务器将以与处理 JSP 文件类似的方法处理该文件，并将生成的 HTML 返回给浏览器。

PHP 语言有许多库，包括使用 ODBC(类似于 Java 中的 JDBC)访问数据库的库。

一个更实际的例子可能执行更复杂的操作，例如使用 JDBC 从数据库中查询值。

JSP 也支持标签库(*tag library*)的概念，它允许使用标签，这些标签看起来非常像 HTML 标签，却是在服务器端解释，并用相应生成的 HTML 代码取代。JSP 提供了一个标准标签集，其中定义了变量和控制流(迭代器、if-then-else)，以及基于 Javascript 的表达式语言(但在服务器端解释)。标签集是可扩展的，并且许多标签库都已实现。例如，存在一个标签库支持对大数据集的分页显示，还有一个库
388 简化了对日期和时间的显示和解析。关于 JSP 标签库的更多信息可以参看文献注解中的参考资料。

9.3.6 客户端脚本

文档中程序代码的嵌入允许 Web 页是**活动的**(active)，通过在本地执行程序以实现活动，例如动画，而不仅仅是展示被动的文字和图片。这种程序主要用于在 HTML 与 HTML 表单提供的有限的交互能力之上，与用户进行灵活的交互。此外，与每次交互都发送给服务器端处理相比，在客户端执行程序极大加快了交互的速度。

支持这种程序的一个危险在于，如果系统设计得不小心，Web 页中(或等同地，在电子邮件消息中)内嵌的程序代码能够在用户的计算机上执行恶意操作。恶意操作会包括窃取私人信息。在计算机上删除或修改信息，甚至控制计算机并将代码传播到其他计算机(例如通过电子邮件)。近年来许多邮件病毒都是通过这种方式广泛传播的。

Java 语言日趋流行的一个原因是它为在用户的计算机上执行程序提供了一种安全模式。Java 代码能够编译成独立于平台的"字节码"，它可以在任意支持 Java 的浏览器上执行。与本地程序不同的是，作为网页的一部分所下载的 Java 程序(小应用程序)无权执行任何可能是破坏性的操作。它们可以在屏幕上显示数据，或者与下载网页的服务器进行网络通信以获取更多的信息。然而，它们不能访问本地文件，执行任何系统程序，或和任何其他计算机进行网络通信。

Java 是一种完全成熟的程序设计语言，另外有一些更简单的语言，称为**脚本语言**(scripting language)，在提供与 Java 相同的保护的同时，能够丰富用户的交互。这些语言模块能够嵌入到 HTML 文档中的构件。**客户端脚本语言**(client-side scripting language)是用于在客户端的 Web 浏览器上执行的语言。

其中，*JavaScript* 语言在目前使用最为广泛。目前这代 Web 界面广泛使用 JavaScript 脚本语言构造复杂的用户界面。JavaScript 用于多种任务。例如，用 JavaScript 写的函数可以用于对用户输入执行错误

检查(验证)，例如日期字串是否符合格式，或者输入的值(例如年龄)是否在适当的范围内。在输入数据时，甚至在数据发送往 Web 服务器之前，这些检查就已在浏览器上执行。

图 9-10 所示为用于验证表单输入的 JavaScript 函数的一个示例。该函数在 HTML 文档的 head 段声明。该函数检查所输入的课程学分是一个大于 0 并且小于 16 的数字。form 标签表示该验证函数在表单提交时调用。如果验证失败，则将一个警告框显示给用户，而如果验证成功，则将表单提交给服务器。

```
<html>
<head>
<script type="text/javascript">
  function validate() {
    var credits=document.getElementById("credits").value;
    if (isNaN(credits) || credits<=0 || credits>=16) {
      alert("Credits must be a number greater than 0 and less than 16");
      return false
    }
  }
</script>
</head>

<body>
<form action="createCourse" onsubmit="return validate()">
  Title: <input type="text" id="title" size="20"><br />
  Credits: <input type="text" id="credits" size="2"><br />
  <input type="submit" value="Submit">
</form>
</body>
</html>
```

图 9-10 用于验证表单输入的 JavaScript 示例

JavaScript 可以用于动态修改所显示的 HTML 代码。浏览器将 HTML 代码解析为内存中的一个树结构，它也由称为**文档对象模型**(Document Object Model，DOM)标准定义的。JavaScript 代码能够修改该树结构以执行某些操作。例如，假设一个用户需要输入几行数据，例如一个清单中的几项。一个包含文本框以及其他形式输入方法的表格可以用来收集用户输入。表格可能有一个默认大小，但是如果需要更多行，用户可以点击标有(比如)"添加项目"的按钮，可以将这个按钮设置为调用修改 DOM 树以向表格额外增加一行的 JavaScript 函数。

虽然 JavaScript 语言已经标准化，但是浏览器之间仍存在差别，特别是 DOM 模型的一些细节。因此，在一个浏览器上运行的 JavaScript 代码有可能在另一个浏览器上无法运行。为了避免这种问题，最好使用一个 JavaScript 库，例如 Yahoo 的 YUI 库，它使得代码的编写独立于浏览器。库中的函数在内部能够找出正在使用哪种浏览器，并向浏览器发送相应生成的 JavaScript。关于 YUI 和其他库的更多信息可以参看本章结尾的 9.5.1 节。

今天，JavaScript 广泛用于创建动态网页，它使用统称为 Ajax 的几种技术。用 JavaScript 编写的程序和 Web 服务器异步通信(即在后台，不阻断用户和 Web 浏览器的交互)，并能够获取数据并显示。

389
|
390

作为使用 Ajax 的一个示例，考虑一个网站表单，允许你选择国家，并且一旦选择了一个国家，你就可以从这个国家的州列表中选择一个州。在国家选定之前，州的下拉列表是空的。Ajax 框架允许在选定一个国家时，在后台从该网站下载州的列表，并且一获取到这个列表，它就添加到下拉列表中，使你可以选择州。

还有一些专用脚本语言用于专门的任务，比如动画(例如，Flash 和 Shockwave)以及三维建模(虚拟现实标记语言(Virtual Reality Markup Language，VRML))。目前，Flash 不仅被广泛用于动画，还用于处理流媒体视频内容。

9.4 应用架构

为了处理大型应用的复杂性，通常将它们分为若干层：

- 表示层或用户界面层，它处理用户交互。单个应用程序可能有若干个不同版本的展示层，对应于不同类型的界面，例如 Web 浏览器，以及手机用户界面，它的屏幕相比较小很多。

  在很多实现中，基于**模型 - 视图 - 控制器**(Model-View-Controller，MVC)架构，表示/用户界面层本身在概念上分为多层。**模型**(model)对应于下文中所述的业务 - 逻辑层。**视图**(view)定义数据的显示；单个底层模型根据用于访问应用所指定的软件/设备可以具有不同的视图。**控制器**(controller)接收事件(用户操作)，在模型上执行操作，并返回一个视图给用户。MVC 架构用于多个 Web 应用框架，这将在 9.5.2 节讨论。

- **业务逻辑**(business-logic)层，提供对数据和数据操作的高级视图。9.4.1 节详细讨论业务逻辑层。

- **数据访问**(data access)层，提供业务逻辑层和底层数据库间的接口。当底层数据库是关系数据库时，许多应用使用面向对象语言编写业务逻辑层，并使用面向对象数据模型。在这种情况中，数据访问层还提供从业务逻辑所使用的面向对象数据模型到数据库所支持的关系模型的映射。9.4.2 节详细讨论这种映射。

图 9-11 所示为上述各层，以及处理来自 Web 浏览器的一条请求所采取的一系列步骤。图 9-11 中箭头上的标记表示步骤的顺序。当应用服务器接收到请求时，控制器向模型发送一条请求。模型使用
391 业务逻辑处理请求，其中可能涉及更新模型中的对象，接着创建一个结果对象。模型依次使用数据访问层更新数据中的信息或从数据库获取信息。模型生成的结果对象发送到视图模板，它对结果生成一个 HTML 视图以在 Web 浏览器上显示。该视图可能根据用于查看结果的设备的特点来定制。例如它是否是一个具有大屏幕的计算机显示器，或者是手机上的小屏幕。

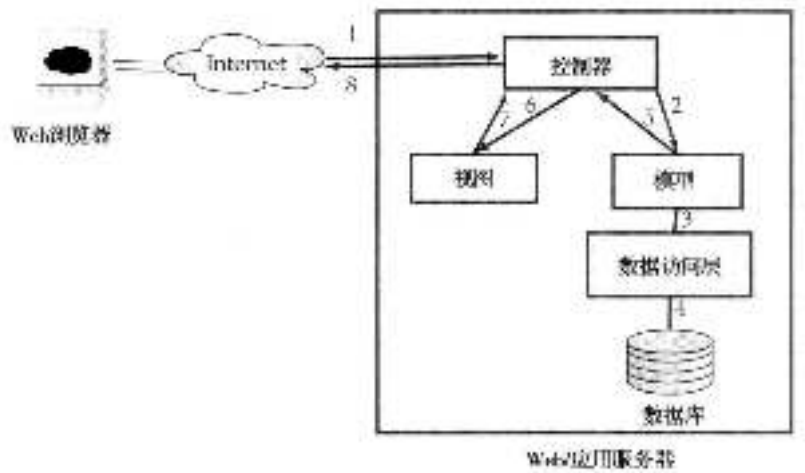


图 9-11 Web 应用程序概览

9.4.1 业务逻辑层

一个用于管理大学的应用程序的业务逻辑层可能会提供实体的抽象，比如学生、教师、课程、课程段等，以及操作，比如大学录取学生、学生注册课程等。实现这些操作的代码保证业务规则(business rule)被满足；例如，代码要保证只有当学生已经完成课程先修课并且已经支付学费时才能注册课程。

另外，业务逻辑包含**工作流**(workflow)，它描述如何处理一个涉及多个参与者的特定任务。例如，如果一名候选人申请大学，存在一个工作流规定首先由设备看并批准申请，并且如果第一个阶段批准后，接下来应该由谁查看申请，如此下去，直到给学生发出入学邀请或者寄出拒信。工作流管理也需要处理错误情况；例如，如果接收/拒绝的最后期限还没到，则可能需要运行主管，使他能够干预并确
392 保申请已处理。工作流在 26.2 节中有详细的讨论。

9.4.2 数据访问层和对象－关系映射

在最简单的场景中，业务逻辑层和数据库使用相同的数据模型，数据访问层只是隐藏了与数据库连接的细节。然而，当用面向对象程序设计语言编写业务－逻辑层时，很自然就会将数据建模为对象以及对象上调用的方法。

在早期实现中，程序员不得不为了从数据库中获取数据以创建对象以及将更新后的对象存回数据库而编写代码。然而，这种数据模型间的人工转换很麻烦并且容易出错。解决这个问题的一种方法是开发一个本身存储对象和对象之间关系的数据库系统。这种数据库称作**面向对象数据库**(object-oriented database)，在第 22 章将对其详细介绍。然而，由于许多技术和商业原因，面向对象数据库没有取得商业上的成功。

另一种方法是使用传统关系数据库来存数据，但自动建立从关系中的数据到内存中按需创建(由

于通常没有足够的内存存储数据库中所有的数据）的对象的映射，以及反向映射，从而将更新后的对象以关系的形式保存回数据库。

HIBERNATE 示例

作为 Hibernate 使用示例，创建一个对应于 student 关系的 Java 类如下：

```
public class Student {
        String ID;
        String name;
        String department;
        int tot_cred;
        Student(String id, String name, String dept, int totcreds); //构造函数
}
```

为了准确，类属性应该声明为私有，并应该提供 getter/setter 方法以访问属性，但是我们省略这些细节。

从 Student 的类属性到 student 关系属性的映射在映射文件中以 XML 格式表示。我们再次省略细节。

下面的代码段创建了一个 Student 对象并将它保存到数据库中：

```
Session session = getSessionFactory().openSession();
Transaction txn = session.beginTransaction();
Student stud = new Student("12328", "John Smith", "Comp. Sci.", 0);
session.save(stud);
txn.commit();
session.close();
```

Hibernate 自动生成所需的 SQL insert 语句，以在数据库中创建一个 student 元组。

为了检索学生，可以用以下代码段：

```
Session session = getSessionFactory().openSession();
Transaction txn = session.beginTransaction();
List students =
        session.find("from Student as s order by s.ID asc");
for (Iterator iter = students.iterator(); iter.hasNext(); ) {
        Student stud = (Student) iter.next();
        --输出学生信息--
}
txn.commit();
session.close();
```

以上代码段使用了用 Hibernate 的 HQL 查询语言表达的一个查询。HQL 查询被 Hibernate 自动翻译成 SQL 并执行，然后把结果转换成一个 Student 对象对象。for 循环遍历该列表中的对象并将输出它们。

已经有好几个实现这种**对象 - 关系映射**（object-relational mapping）的系统被开发出来。Hibernate 系统广泛用于将 Java 对象映射到关系。在 Hibernate 中，从每个 Java 类到一个或多个关系的映射都记录在一个映射文件中。例如，该映射文件可以记录一个叫做 *Student* 的 Java 类映射到关系 *student*，同时 Java 属性 ID 映射到属性 *student*. ID 等。*properties* 文件记录了数据库的信息，例如数据库运行所在的主机，以及用于连接数据库的用户名和密码。程序需要打开一个会话，建立到数据库的连接。一旦会话建立，Java 中创建的 *Student* 对象 *stud* 就可以通过调用 session. save(*stud*) 保存到数据库中。Hibernate 代码生成用于将数据存储到 *student* 关系中的 SQL 命令。

一组对象可以通过执行用 Hibernate 查询语言所写的查询从数据库中获取；这与用 JDBC 执行查询并返回包含一组元组的 ResultSet 是类似的。或者，单个对象可以通过提供它的主码来获取。获取到的对象可以在内存中更新；当运行中的 Hibernate 会话上的事务被提交时，Hibernate 通过在数据库中的关系上进行相应的更新，自动保存更新的对象。

虽然 E-R 模型中的实体自然地对应诸如 Java 的面向对象语言中的对象，但联系通常并不对应。Hibernate 支持将这种联系映射为关联对象的集合的能力。例如，*student* 和 *section* 之间的 *takes* 联系可以通过将 *section* 的集合和每个 *student* 关联，并将 *student* 的集合和每个 *section* 关联来建模。一旦指定好适

393 | 394 当的映射，Hibernate 就会根据数据库关系 *takes* 自动地充这些集合，并且对于集合的更新也会在提交时反映回数据库关系中。

上述特点帮助提供给程序员一个不需要考虑关系存储细节的高级数据模型。然而，Hibernate 和其他对象-关系映射系统一样，也允许程序员直接用 SQL 访问底层关系。这种直接访问对于编写复杂查询以生成报表至关重要。

Microsoft 开发了一种数据模型，称作**实体数据模型**(Entity Data Model)，可以将它看成多种实体-联系模型，以及一个相关的框架，叫作 ADO.NET 实体框架，它能在实体数据模型和关系数据库间映射数据。该框架还提供一种类 SQL 查询语言，称作**实体 SQL**(Entity SQL)，它直接在实体数据模型上操作。

9.4.3 Web 服务

在过去，大部分 Web 应用只使用应用服务器及其相关的数据库中的数据。近年来，Web 中存在许多种类的数据，需要通过程序来处理，而不是直接显示给用户；程序可能作为后端应用的一部分运行，或可能是在浏览器中运行的脚本。通常，通过实际的 Web 应用编程接口访问这些数据，即使用 HTTP 协议发送一个函数调用请求，在应用服务器上执行，然后将结果发送回主调程序。支持这种接口的系统被称为 **Web 服务**(Web service)。

有两种方法广泛用于实现 Web 服务。较简单的方法为**代表性状态传输**(REpresentation State Transfer，REST)，它通过在应用服务器对 URL 的标准 HTTP 请求执行 Web 服务函数调用，参数作为标准 HTTP 请求的参数发送。应用服务器执行该请求(有可能涉及服务器端数据库的更新)，生成结果并对其编码，然后将结果作为 HTTP 请求的结果返回。服务器对一个特定请求的 URL 可以使用任意编码；XML 和一种叫作 **JavaScript 对象符号**(JavaScript Object Notation，JSON)的 JavaScript 对象编码广泛使用。请求方解析返回的页面以访问所返回的数据。

在许多 REST 风格的 Web 服务(即使用 REST 的 Web 服务)的应用中，请求方是运行在 Web 浏览器中的 JavaScript 代码；该代码使用函数调用的结果更新浏览器屏幕。例如，当你在 Web 上的地图界面上滚动显示时，地图中需要新增的显示部分可以使用 REST 风格的接口通过 JavaScript 代码获取，然后显示在屏幕上。

一个更加复杂的方法有时称作"大 Web 服务"，它对参数和结果使用 XML 编码，使用一种专门的语言正规定义 Web API，并在 HTTP 协议上构建了一个协议层。23.7.3 节有对这个方法的详细描述。

9.4.4 断连操作

许多应用都希望即使当一个客户端从应用服务器断开连接时仍支持某些操作。例如，一个学生可
395 | 能希望填写申请表即使她的笔记本电脑从网络断开，但在笔记本电脑重新连接网络的时候将它保存回去。另一个例子是，如果将一个电子邮件客户端构建为一个 Web 应用，一个用户可能希望书写电子邮件即使她的笔记本电脑从网络断开，并在重新连接到网络时将邮件发送出去。构建这种应用需要在客户端机器中本地存储，最好是以数据库的形式。Gears 软件(最初由 Google 开发)是一个浏览器插件，它提供一个数据库、一个本地 Web 服务器，并支持在客户端并行执行 JavaScript。该软件在多种操作系统/浏览器平台上同样运行，允许应用程序支持丰富的功能而并不需要安装任何的软件(除了 Gears 自己以外)。Adobe 的 AIR 软件还能够为运行在 Web 浏览器之外的 Internet 应用的构建提供类似的功能。

9.5 快速应用开发

如果构建 Web 应用时没有使用工具或库构建用户界面，则构建用户界面所需的编程工作可能远大于业务逻辑和数据库访问所需。有几种方法可以用以减少构建应用程序所需的工作：

- 提供一个函数库，以用最小的编程量生成用户界面元素。
- 在集成开发环境中提供拖放功能，允许将用户界面元素从菜单中拖至页面的设计视图。该集成开发环境通过调用库函数生成创建用户界面元素的代码。

- 根据声明规范自动生成用户界面的代码。

在创建 Web 之前，所有这些方法就都已经作为**快速应用开发**（Rapid Application Development，RAD）工具的一部分用于创建用户界面了，并且目前也广泛用于创建 Web 应用。

用来快速开发数据库应用接口的工具的例子包括 Oracle Forms、Sybase PowerBuilder 以及 Oracle Application EXpress（APEX）。另外，为 Web 应用开发所设计的工具，例如 Visual Studio 和 Netbeans VisualWeb，支持用于快速开发基于数据库的应用的 Web 界面的一些功能。

我们在 9.5.1 节中学习创建用户界面的工具，并在 9.5.2 节中学习支持自动根据系统模型生成代码的框架。

9.5.1 构建用户界面的工具

许多 HTML 结构最好是由适当定义的函数生成，而不是作为网页代码的一部分来编写。例如，地 396
址表单通常需要一个包含国家或州名的菜单。最好定义一个输出该菜单的函数并在需要的时候调用该函数，而不是每次用到的时候编写冗长的 HTML 代码以生成所需的菜单。

菜单通常最好由数据库中的数据生成，比如一个包含国家名或州名的表。生成菜单的函数执行数据库查询并用查询结果如写菜单。增添一个国家或州则只需要改变数据库，而不需要改变应用程序代码。这个方法有个潜在的缺陷，即需要增加数据库交互。不过可以通过在应用服务器端缓存查询结果以最小化这个开销。

类似地，输入日期和时间或需要验证的输入的表单最好通过调用适当定义的函数生成。这些函数能够输出在浏览器端执行验证的 JavaScript 代码。

对于许多数据库应用，显示查询的结果集是一项共有的任务。可以构建一个泛型函数，将 SQL 查询（或 ResultSet）作为参数，并以表格形式显示查询结果（或 ResultSet）中的元组。可以使用 JDBC 元数据调用从查询结果中找到诸如列数以及列的名称和类型的信息；然后该信息用于显示查询结果。

为了处理查询结果非常大的情况，查询结果显示功能可以提供对结果的分页。该功能可以在一页中显示固定条数的记录，并提供控制以跳到下一页或前一页或者跳到结果的指定页。

遗憾的是，没有（广泛使用的）用于实现上述用户界面任务的标准 Java API 函数。构建这样的一个库将会是一个有趣的编程项目。

不过，存在其他一些工具，比如 JavaServer Faces（JSF）框架，它就支持上面所列功能。JSF 框架包含一个实现这些功能的 JSP 标签库。Netbeans IDE 具有一个叫做 VisualWeb 的组件，它构建在 JSF 上，提供一个可以将属性自定义的用户界面组件拖放到一个页面的可视化开发环境。例如，JSF 提供创建下拉菜单的组件，或者显示从数据库查询中获取数据的表格的组件。JSF 还支持在组件上验证规范，例如做选择或强制输入，或者限制一个数或一个日期在指定范围内。

Microsoft 的 Active Server Pages（ASP）及其最近的版本 Active Server Pages. NET（ASP. NET），是 JSP/Java 的一种广泛使用的替代品。ASP. NET 与 JSP 类似，其中可以在 HTML 代码中嵌入 Visual Basic 或者 C#语言的代码。另外，ASP. NET 提供多种控件（脚本命令），在服务器端解释并生成 HTML 返回给客户端。这些控件能显著地简化 Web 界面的构建。我们对这些控件带来的好处给出一个简要的概览。 397

例如，诸如下拉菜单和列表框这样的控件可以与一个 DataSet 对象关联。DataSet 对象与 JDBC 的 ResultSet 对象相似，通常也是通过在数据库上执行查询而创建的。HTML 菜单内容从 DataSet 对象的内容生成；例如，一个查询可能将一个机构中所有部门的名字检索到 DataSet 中，并且关联的菜单将会包含这些名字。这样，基于数据库内容的菜单就能以极少量的编码方便地创建出来。

验证控件可以添加到表单的输入域中；这声明性地定义了有效性约束，诸如值的范围或者输入是否是必需的（即用户必须提供的值）。服务器创建适当的结合 JavaScript 的 HTML 代码，以在用户的浏览器中执行有效性验证。对于无效输入所显示的错误消息可以与每个验证控件相关联。

可以指定用户操作在服务器端具有相关联的操作。例如，一个菜单控件可以指定从菜单中选择一个值具有一个相关联的服务器端动作（生成 JavaScript 代码以探查选择事件并初始化服务器端操作）。显示属于所选值的数据的 Visual Basic/C#代码可以与服务器上的该动作相关联。因此，从菜单中选择一个值会引起页面上相关联的数据的更新，而不需要用户点击提交按钮。

DataGrid 控件提供了一个非常方便的方法显示查询结果。一个 DataGrid 和一个 DataSet 对象相关联，DataSet 对象通常是一个查询的结果。服务器生成将查询结果显示为表的 HTML 代码。列标题根据查询结果的元数据自动生成。另外，DataGrid 提供了一些功能，比如分页，以及允许用户在所选列上对结果排序。所有实现这些特性的 HTML 代码以及服务器端功能都是由服务器自动生成的。DataGrid 甚至允许用户编辑数据并将更改提交回服务器。应用开发人员可以指定一个函数在一行数据被编辑时执行，这就可以实现数据库上的更新。

Microsoft Visual Studio 为创建使用这些特性的 ASP 页面提供了图形用户界面，进一步减少了编程的工作。

关于 ASP.NET 的更多信息，请参看文献注解中的参考文献。

9.5.2 Web 应用框架

有多种 Web 应用开发框架都提供一些常用的特性，比如：

- 一个包含对象-关系映射的面向对象模型，从而在关系数据库中保存数据（如 9.4.2 节所述）。
- 一种（相对）声明式的方式，说明一个表单具有在用户输入上的验证约束。系统据此生成 HTML
398 和 JavaScript/Ajax 代码以实现表单。
- 一个模板脚本系统（与 JSP 类似）。
- 一个控制器，将诸如表单提交的用户交互事件映射到处理该事件的适当函数上。控制器还管理身份验证和会话。一些框架还提供管理身份验证的工具。

这样，这些框架就以一种集成的方式提供了多种构建 Web 应用所需的特性。通过根据声明规范生成表单，以及透明地管理数据访问，这些框架最小化了 Web 应用程序员所需完成的代码量。

这样的框架有许多，它们基于不同语言。其中一些较为广泛使用的框架包括 Ruby on Rails，它是基于 Ruby 程序设计语言的，还有 JBoss Seam、Apache Struts、Swing、Tapestry 以及 WebObjects，它们都是基于 Java/JSP 的。其中的一些（比如 Ruby on Rails 和 JBoss Seam）提供自动生成简单 CRUD Web 界面的工具；也就是支持对象/元组的创建、阅读、更新和删除的界面，它是通过对象模型或数据库生成代码的。这种工具在使简单应用快速运行时特别有用，并且可以编辑生成的代码以构建更复杂的 Web 界面。

9.5.3 报表生成器

报表生成器是从数据库生成人们可读的概要报告的工具。它将生成格式化文本和概要图表（例如条形图或饼状图）与查询数据库集成在一起。例如，一个报表可能会显示过去的两个月中每个月每个销售地区的总销售额。

应用开发人员可以利用报表生成器的格式化工具来指定报表的格式。变量可以用来存储参数（如月、年）以及定义报表中的域。表、图、条形图或其他的图可以通过对数据库的查询来定义。查询定义可以利用存储在变量里的参数值。

一旦在报表生成器工具上定义了一个报表的结构，就可以保存它并可以在任何时候执行它来产生报表。报表生成器系统提供了多种工具来组织表格式输出，如定义表和列标题，显示表中每一组的小计，自动将一个长的表格分成若干页，在每一页末尾显示本页小计等。

图 9-12 是一个格式化报表的例子。报表中的数据是按顺序对信息进行聚集产生的。

许多厂商都提供报表生成工具，例如 Crystal Reports 和 Microsoft（SQL Server Reporting Services）。一些应用套件（例如 Microsoft Office）提供了将来自数据库的格式化查询结果直接嵌入到文档中的方法。Crystal Reports 提供的图表生成工具或者诸如 Excel 提供的电子表格都可以用于从数据库访问数据，并生成表格化的数据展现或者使用图表或图的图形化方式展现数据。这种图表可以嵌入文本文档中。图表最初由执行数据库查询生成的数据创建；查询在需要的时候可以重新执行并重新生成图表，以获得概要报表的当前版本。

除了生成静态报表，报表生成工具也支持创建交互式报表。例如，用户可以"下钻"到感兴趣的区域，比如从一个显示全年总销量的聚集视图转移到某一年的月销售图表。在前面 5.6 节已讨论过这种操作。

Acme供应有限公司
年度销售报告

日期: 2009年1月1日至3月31日

地　区	类　别	销　量	小　计
北部	计算机硬件	1 000 000	
	计算机软件	500 000	
	所有类别		1 500 000
南部	计算机硬件	200 000	
	计算机软件	400 000	
	所有类别		600 000
		总计	2 100 000

图 9-12　一个格式化报表

9.6　应用程序性能

Web 站点可能被来自全球的数百万人访问，每秒钟要处理数千次的请求。对于最流行的站点甚至更多。确保对请求的服务有较短的响应时间是 Web 站点开发人员面临的一个主要挑战。为了做到这点，应用开发人员通过利用诸如缓存等技术尽可能加快处理单个请求的速度，并通过使用多个应用服务器进行并行处理。我们在下面简要描述这些技术。数据库应用调优在后面的第 24 章(24.1 节)详细讲述。

9.6.1　利用缓存减少开销

多种缓存技术用于充分利用事务之间的共性。例如，假设服务于每个用户请求的应用程序代码都需要通过 JDBC 连接数据库。因为创建一个新的 JDBC 连接需要几个毫秒的时间，所以在要支持很高的事务处理速率时，为每次请求创建一个新连接是不合适的。

399
↓
400

连接池(connection pooling)用来减轻这种开销。它是这样工作的：连接池管理器(应用服务器的一部分)创建一个打开的 ODBC/JDBC 连接的池。与打开一个新的数据库连接不同，对用户请求提供服务的代码(通常是一个 servlet)向连接池申请(请求)一个连接，然后在代码(servlet)完成处理时将连接归还给连接池。如果在请求连接时连接池没有未使用的连接，则打开一个新的数据库连接(要小心不要超过数据库能同时支持的最大连接数目)。如果有很多打开的连接在一段时间内没有使用，连接池管理器可能会关闭一些打开的数据库连接。许多应用服务器以及较新的 ODBC/JDBC 驱动程序提供内置的连接池管理器。

在创建 Web 应用时，许多程序员常犯的一个错误是忘记关闭一个打开的 JDBC 连接(或者等同地，在用连接池的时候，忘记把连接归还给连接池)。每个请求都打开一个新的数据库连接，而且数据库很快就达到了同时能支持的连接数目的上限。这样的问题通常在小规模测试的时候不会暴露出来，因为数据库通常能允许数百个打开的连接，而仅会在频繁使用的时候表现出来。其实程序员以为连接像 Java 程序分配的内存，会自动回收，遗憾的是，这不会发生，并且程序员有责任关闭他们所打开的连接。

某些请求可能导致向数据库重新提交完全相同的查询。只要数据库中的查询结果没有改变，通过缓存先前的查询结果并重用它们，数据库通信的开销可以大大减少。一些 Web 服务器支持这样的查询结果缓存；如果不支持的话，可以在应用程序代码中显式实现缓存。

通过缓存响应一个请求所发送的最终 Web 页面可以进一步减少开销。如果新请求和前面的一个请求参数完全一致，请求不需要执行更新，并且结果 Web 页面在缓存中，那么它就可以重用，从而避免重新计算该页面的开销。缓存可以在 Web 页的片段级别实现，然后它们被组装以创建完整的 Web 页。

缓存的查询结果和缓存的 Web 页面是物化视图的形式。如果底层数据库数据改动，缓存的结果必须废弃，或重新计算，或甚至增量更新，就如物化视图的维护(在后面 13.5 节介绍)那样。某些数据库系统(比如 Microsoft SQL Server)为应用服务器提供了一种方式在数据库注册查询，并且在查询结果改变时从数据库得到通知(notification)。这种通知机制可以用于确保缓存在应用服务器中的查询结果是最新的。

9.6.2 并行处理

处理非常重的负载的一个常用的方法是并行运行许多台应用服务器，每个处理一小部分请求。一
401 个 Web 服务器或者一个网络路由器可以用于将每个客户端的请求路由到一台应用服务器。来自一个特定客户端会话的所有请求都必须送到同一个应用服务器，因为服务器要维护客户端会话的状态。比如，可以通过将所有来自一个 IP 地址的请求都路由到相同的应用服务器来确保这个性质。然而，底层数据库被所有应用服务器共享，因此用户看到的数据库是一致的。

由于在以上架构中数据库是共享的，因此它很容易成为瓶颈。应用程序设计人员通过在应用服务器端缓存查询结果，特别地注意最小化数据库请求的数量。另外，当需要时则使用并行数据库系统，这将在第 18 章讲述。

9.7 应用程序安全性

应用程序安全性要在 SQL 授权处理的范围之外对几种安全威胁和问题进行处理。

第一个必须要强制实施安全性的地方是在应用程序里。为此，应用程序必须对用户进行身份验证，并确保用户只允许完成授权的任务。

存在许多方法使一个应用程序的安全性能够受到威胁，即使数据库系统本身是安全的。不好的应用程序代码也会使应用程序的安全性受到威胁。本节首先介绍几种安全漏洞，它们能够允许黑客采取行动绕过应用程序采取的身份认证和授权检查，并解释如何防止这种漏洞。本节后面介绍安全认证和细粒度授权的技术。然后我们介绍审计追踪，它能够帮助从未经授权的访问和错误更新中恢复。我们通过介绍数据隐私问题结束本节。

9.7.1 SQL 注入

在 SQL 注入(SQL injection)攻击中，攻击者设法获取一个应用程序来执行攻击者生成的 SQL 查询。在 5.1.1.4 节中，我们看到了一个 SQL 注入漏洞的例子，如果用户输入直接与 SQL 查询连成一串并提交给数据库。作为 SQL 注入漏洞的另一个例子，考虑图 9-4 中所示的表单源文本。假设在图 9-8 所示的相应的 servlet 中用以下 Java 表达式创建了一个 SQL 查询串：

```
String query = "select * from student where name like '%" + name + "%'"
```

402 其中 name 是用户输入的包含字符串的变量，并继而在数据库上执行该查询。利用 Web 表单的恶意攻击者则可以键入诸如"'; < some SQL statement >; --"的字符串，替代一个有效的学生姓名，其中 < some SQL statement > 表示任何攻击者希望的 SQL 语句。servlet 则将执行以下字符串：

```
select * from student where name like'%'; < some SQL statement >; --'
```

攻击者插入的引号结束该字符串，后面的分号终止该查询，然后攻击者接下来所插入的文本被解析为第二条 SQL 查询，而右引号已被注释掉。这样，该恶意用户成功插入被应用程序执行的任意 SQL 语句。该语句可以导致重大的损害，因为它能够绕过应用程序代码中实现的所有安全措施，在数据库上进行任意的操作。

如 5.1.1.4 节所述，为了避免这种攻击，最好使用预备语句来执行 SQL 查询。当设置预备语句的一个参数时，JDBC 自动添加转义字符，从而使得用户提供的引号无法再终止字符串。等价地，添加这种转义字符的操作也可以在与 SQL 查询连接之前用在输入字符串上，而不使用预备语句。

SQL 注入危险的另一个来源是基于表单中指定的选择条件和排序属性动态创建查询的应用。例如，一个应用可能允许用户指定用哪个属性对查询结果排序。假设该应用从表单变量 orderAttribute 中获取属性名，并创建了一个查询串如下：

```
String query = "select * from takes order by " + orderAttribute;
```

即使用于获取输入的 HTML 表单通过提供表单试图限定所允许的值，一个恶意用户仍可以发送一个任意字符串替代一个有意义的 orderAttribute 值。为了避免这种 SQL 注入，应用程序应该在追加 orderAttribute 变量值之前确保它是所允许的值(在例子中为属性名)。

9.7.2 跨站点脚本和请求伪造

一个允许用户输入诸如评论或姓名的文本，然后将其保存并在以后显示给其他用户的网站，有可能会受到一种攻击，叫做**跨站点脚本**(Cross-Site scripting，XSS)攻击。在这种攻击中，一个恶意用户并不输入一个有效的姓名或评论，而是输入用诸如 JavaScript 或 Flash 的客户端脚本语言编写的代码。当另一个用户浏览输入的文本时，浏览器将会执行脚本，它可能会进行一些操作，比如将私人的 cookie 信息发送回恶意用户，或甚至在一个用户可能登录的另一个不同的 Web 服务器中执行操作。 403

例如，假设该脚本执行的时候用户恰巧登录了他的银行账户。该脚本可以将有关银行账户登录的 cookie 信息发送回恶意用户，他就可以用该信息连接到银行的 Web 服务器，欺骗它使它相信该连接来自原用户。或者，该脚本可以适当设置参数并访问银行网站上的适当网页，以执行转账。事实上即使不写脚本而只是用如下的一行代码就能够使这个问题发生：

```
< img src =
  " http: //myhank. com/transfermoney? account = 1000&toaccount = 14523" >
```

假定 URL myhank. com/transfermoney 接受指定的参数，并进行了转账。这后一种漏洞又称作**跨站点请求伪造**(Cross-Site Request Forgery 或 XSRF，有时也称作 CSRF)。

XSS 可以用其他一些方式实现，比如引诱用户访问有恶意脚本嵌入其网页的网站。存在其他更加复杂的 XSS 或 XSRF 攻击，我们在此不再细讲。为了防止此类攻击，需要完成两件事：

- **防止你的网站被用来发动 XSS 或 XSRF 攻击。**
  最简单的技术就是禁止用户输入的任何文本中的任何 HTML 标签。存在检测或去除这些标签的函数。这些函数可以用来防止 HTML 标签，并就此防止脚本展示给其他用户。在有些情况下 HTML 格式是有用的，并且在这种情况下可以使用那种解析文本并允许有限的 HTML 结构，但不允许其他危险结构的函数。这些必须小心地设计，因为有时包含一个跟图片一样无害的结构也可能是危险的，如果图片显示软件中有一个能发现的错误。
- **防止你的网站被其他站点发动的 XSS 或 XSRF 攻击。**
  如果该用户已经登录你的网站，并访问了另一个易受 XSS 攻击的网站，在该用户的浏览器上执行的恶意代码则可能在你的网站上执行操作，或将与你的网站关联的会话信息发送回试图利用它的恶意用户。无法完全防止这种攻击，但是你可以采取几个步骤最小化风险。
  - HTTP 协议允许一个服务器检查访问页的引用页(referrer)，即用户为了初始化访问页面点击的链接所在网页的 URL。通过检查引用页是否有效，例如，引用页 URL 是同一个网站上的网页，源自用户访问的不同网页上的 XSS 攻击则可以防止。
  - 除了只使用 cookie 来标识一个会话外，还可以将会话限制在原始验证它的 IP 地址上。这样，即使一个恶意用户得到了 cookie，他也可能无法从另一台计算机登录。 404
  - 决不要使用 GET 方法执行任何更新。这防止了利用 < img src . . . > 的攻击，如我们前面所看到的。事实上，由于其他原因，比如页面的刷新重复了本应只发生一次的操作，HTTP 标准建议 GET 方法不应该执行任何更新。

9.7.3 密码泄露

应用程序开发人员必须处理的另一个问题是在应用程序代码的明文中保存密码。例如，诸如 JSP 脚本的程序通常在明文中包含密码。如果这种脚本保存在一个 Web 服务器可访问的目录中，一个外部用户就可能能够访问脚本的源码，并获取应用程序使用的数据库账户的密码。为了避免这种问题，许多应用服务器提供用编码的形式保存密码的机制，在传送给数据库之前服务器对其解码。该功能去除了在应用程序中将密码另存为明文的需要。然而，如果解码密钥也容易暴露的话，这种方法就并不完

全有效了。

作为处理易受泄露的数据库密码的另一种措施，许多数据库系统对于数据库的访问只限制在给定网络地址集合中。该机制通常运行在应用服务器端。从其他网络地址企图连接数据库会被拒绝。这样，除非恶意用户能够登录应用服务器，否则即使他获取了数据库密码的访问权限他也无法造成任何损害。

9.7.4 应用程序认证

认证是指验证连接到应用程序的人/软件的身份。认证最简单的形式由一个密码构成，当一个用户连接到应用程序时必须出示该密码。遗憾的是，密码容易泄露，例如通过试猜，或者通过嗅探网络中的数据包，如果密码没有加密就传送的话。对于关键应用需要更健壮的方案，比如网上银行账户。加密是更健壮的认证方案的基础。通过加密的认证在 9.8.3 节讲述。

许多应用使用**双因素认证**(two-factor authentication)，其中两个独立的因素(即信息或程序的片段)用于识别一个用户；这两个因素不应该具有相同的弱点；例如，如果一个系统只需要两个密码，二者都可能以同样的方式泄露(例如通过网络嗅查，或通过用户使用的计算机上的病毒)。虽然诸如指纹或虹膜扫描仪的生物识别技术可以用于在认证点上用户是物理存在的情况，但是跨网络时它们就不是很
405 有用了。

在大部分这种双因素认证方案中，密码用作第一个因素。通过 USB 接口连接的智能卡或其他加密设备，可基于加密技术用于认证(见 9.8.3 节)，它们广泛用于第二个因素。

一次性密码设备，每分钟生成一个新的伪随机数，也广泛用于第二个因素。给每个用户一个设备，为了进行认证，用户必须输入认证时设备上所显示的数字以及密码。每个设备生成不同的伪随机数序列。应用服务器能够生成与用户的设备相同的伪随机数序列，在认证时在将显示的数字处停下来，并验证数字是否匹配。该方案需要设备中与服务器端的时钟始终是合理紧密同步的。

而第二个因素的另一种方法是当一个用户想登录一个应用时，给用户的手机(其号码是事先已注册的)发送一条包含(随机生成的)一次性密码的短信。用户必须拥有一部具有该号码的手机以接收短信，然后将一次性密码和她的密码一起输入用以认证。

然而即使使用双因素认证，用户可能仍很容易受到**中间人**(man-in-the-middle)攻击。在这种攻击中，一个试图连接应用的用户被转向一个虚假网站，它接受用户的密码(包括第二因素密码)，并立即用该密码到原始应用中认证。9.8.3.2 节描述的 HTTPS 协议用于为用户认证网站(使得用户不会连接到虚假网站)。HTTPS 协议还对数据加密，并防止中间人攻击。

当用户访问多个网站时，用户通常会因为不得不分别在每个网站上认证自己而感到不快，通常每个网站的密码是不同的。有系统允许用户向一个中央认证服务进行认证，并且其他网站和应用可以通过中央认证服务对用户进行认证；于是相同的密码可以用于访问多个站点。LDAP 协议广泛用于实现这种中央认证点：一些机构实现了包含用户名和密码信息的 LDAP 服务器，并且应用程序使用 LDAP 服务器对用户进行认证。

除了认证用户外，中央认证服务还能够提供其他的服务，例如，向应用程序提供用户的信息，比如姓名、E-mail 以及地址信息。这消除了在每个应用中分别输入这些信息的需要。如 19.10.2 节所述，LDAP 可以用于这个任务。其他的目录系统，如微软的活动目录，也提供认证用户以及提供用户信息的机制。

单点登录(single sign-on)系统还允许用户只认证一次，且多个应用通过一个认证服务对用户的身份
406 进行验证，而不需要再次认证。也就是说，一旦用户登录一个站点，他就不需要在其他使用相同单点登录服务的站点输入他的用户名和密码。这种单点登录机制已长期用于网络认证协议，比如 Kerberos，并且目前已经有面向 Web 应用的实现。

安全断言标记语言(Security Assertion Markup Language，SAML)是一个在不同安全域间交换认证和授权信息的标准，以提供跨机构单点登录。例如，假设一个应用需要提供对一所特定学校(比如说耶鲁)所有学生的访问。该学校可以建立一个基于 Web 的服务实施认证。假设一个连接到该应用的用户具有用户名，比如"joe@yale.edu"。该应用将该用户转向耶鲁大学的认证服务对用户认证。而并不直接对用户认证，并告诉应用该用户是谁且可能提供一些额外的信息，比如用户的类别(学生或教师)或

者其他相关的信息。该用户的密码以及其他认证因素不会显示给应用，并且用户也不需要对应用程序显式地注册。不过，当认证用户时，该应用程序必须信任学校的认证服务。

OpenID 标准是另一种跨机构单点登录的标准，并在近年来逐渐被接受。许多流行的网站，比如 Google、Microsoft、Yahoo! 等，是 OpenID 认证提供方。任何作为 OpenID 客户端的应用程序则可以使用这些提供方中的任意一个来认证用户；例如，一个具有 Yahoo! 账户的用户可以选择 Yahoo! 作为认证提供方。该用户被重定向到 Yahoo! 进行认证，并且成功认证后被透明地重定向回应用程序，并能够继续使用应用程序。

9.7.5 应用级授权

虽然 SQL 标准支持一种比较灵活的基于角色的授权系统（如 4.6 节所述），但 SQL 授权模型在一个典型应用中对于用户授权的管理还是非常有限的。例如，假设你想要所有学生都可以看到他们自己的成绩，但是看不到其他人的成绩。这样的授权就无法在 SQL 中表示。原因至少有两点：

1. **缺乏最终用户信息。**随着 Web 规模的增长，数据库访问十有八九来自 Web 应用服务器。最终用户在数据库本身上通常没有个人用户标识，并且数据库中可能只存在单个用户标识对应于应用服务器的所有用户。因此，SQL 中的授权规范在上述情景中无法使用。

应用服务器能够认证最终用户，并继而将认证信息传递给数据库。在本节中我们将假设函数 *sys_context*, *user_id*() 返回一个正在执行的查询所代表的应用程序用户的标识。[1] [402]

2. **缺乏细粒度的授权。**如果我们授权学生只查看他们自己的成绩的话，授权必须是在单条元组的级别上。在目前的 SQL 标准中这种授权是不可能的，SQL 标准只允许在整个关系或视图上，或者关系或视图的指定属性上授权。

我们可以通过为每个学生在 *takes* 关系上创建一个只显示该学生成绩的视图，绕过这个限制。虽然这样在理论上可行，但是这将会非常繁琐，因为我们需要为学校中每一个注册的学生创建一个这样的视图，这是非常不实际的。[2]

另一种方法是创建一个视图形如：

```
create view studentTakes as
select *
from takes
where takes.ID = sys_context('user_id')
```

能即向用户授权这个视图，而不是底层的 *takes* 关系。然而，代表学生所执行的查询现在就必须是在视图 *studentTakes* 上，而不是原始 *takes* 关系上，而代表教师所执行的查询就可能需要使用不同的视图。结果开发应用程序的任务变得更为复杂。

目前，授权的任务通常全部由应用程序执行，绕过 SQL 授权机制。在应用级别，授权用户访问特定接口，并可能进一步被限制只能查看或更新某些数据项。

虽然在应用程序中执行授权给应用程序开发人员带来很大的灵活性，但也存在一些问题：

- 检查授权的代码和应用程序的其他代码混合在一起。
- 通过应用程序的代码实现授权，而不是在 SQL 声明中授权，使得难以确保没有漏洞。由于一个疏忽，一个应用程序可能没有检查权限，而允许未授权的用户访问机密数据。 [406]

验证所有应用程序都完成所有需要的授权检查，涉及遍历所有应用服务器的代码，在一个大的系统中是个非常艰巨的任务。也就是说，应用程序有一个非常大的"表面积"，这使保护应用程序的任务异常艰难。并且事实上，安全漏洞在各种实际生活应用中都有发现。

---

[1] 在 Oracle 中，一个使用 Oracle 的 JDBC 驱动的 JDBC 连接可以用 OracleConnection.setClientIdentifier(userId) 函数设置最终用户标识，并且一个 SQL 查询可以用 sys_context('USERENV', 'CLIENT IDENTIFIER') 函数检索用户标识。

[2] 数据库系统只于管理大关系，但是管理添加视图的模式信息时会假定数量很小，从而提高整体性能。

与此相反，如果一个数据库直接支持细粒度的授权，则可以在 SQL 级别指定并强制执行授权规则，这样表面积就会小很多。即使一些应用接口无意中省略了所需的授权检查，SQL 级授权也能够防止非授权操作的执行。

一些数据库系统提供了细粒度的授权机制。例如，Oracle 的**虚拟私有数据库**（Virtual Private Database，VPD）允许系统管理员将一个函数关联一个关系；该函数返回一个谓词，该谓词必须加入到任何一个使用该关系的查询中（对于被更新的关系可以定义不同的函数）。例如，通过使用我们获取应用程序用户标识的语法，关系 *takes* 的函数可以返回一个谓词，比如：

$$ID = rvi\_context.user\_id()$$

该谓词加入到每个使用关系 *takes* 的查询的 **where** 子句中。结果（假定应用程序将 *user_id* 的值设置为学生的 *ID*），每个学生只能看见她选择的课程所对应的元组。

这样，VPD 提供了在关系的指定元组（或行）上的授权，并因此称为行级授权（row-level authorization）机制。上述添加谓词的方法中一个潜在的问题是，它可能会显著地改变一个查询的含义。例如，如果一个用户写了一条查询来查找所有课程的平均成绩，她最后将只能得到她的成绩的平均值，而非所有课程的平均成绩。虽然系统对于重写的查询会给出“正确的”答案，但是这个答案与用户可能认为她所提交的查询并不对应。

关于 Oracle VPD 的更多信息请参见文献注解。

9.7.6 审计追踪

审计追踪（audit trail）是关于应用程序数据的所有更改（插入/删除/更新）的日志，以及一些信息，如哪个用户执行了更改和什么时候执行的更改。如果应用程序安全性被破坏，或者即使安全性没有被破坏但是一些更新错误执行，一个审计追踪能够（a）帮助找出发生了什么，以及可能是由谁执行的操作，并（b）帮助修复安全漏洞或错误更新造成的损害。

例如，如果发现一个学生的成绩不正确，则可以检查审计日志，以定位该成绩是什么时候以及如何被更新，并找出执行这个更新的用户。该学校还可以利用审计追踪来跟踪这个用户所做的所有更新，从而找到其他错误或欺骗性的更新，并将它们更正。

409

审计追踪还可以用于探查安全漏洞，在安全漏洞中用户账户易泄露并被入侵者访问。例如，在用户每次登录时，她可能被通知审计追踪中从最近一次登录开始所做的所有更新，如果用户发现一个更新并不是由她执行的，则有可能该账户泄露。

可以通过在关系更新操作上定义适当的触发器来创建一个数据库级审计追踪（利用标识用户名和时间的系统定义变量）。然而，很多的数据库系统提供了内置机制创建审计追踪，使用更加方便。具体如何创建审计追踪的细节随数据库系统的不同而不同，具体细节应参考数据库系统用户手册。

数据库级审计追踪对于应用程序来说通常是不够的，因为它们通常无法追踪应用程序的最终用户是谁。另外，它在一个低级别记录更新，即关系中元组的更新，而并不是在较高的级别，即业务逻辑级别。因此，应用程序通常创建一个较高级别的审计追踪，例如，记录执行了什么操作，何人、何时，以及请求源自哪个 IP 地址。

一个相关的问题是防止审计追踪本身被威胁应用程序安全的用户修改或删除。一个可能的解决方法是将审计追踪备份到入侵者无法访问的另一台机器中，每条追踪记录一旦生成立即复制。

9.7.7 隐私

目前，越来越多的个人数据都可以在线获取，人们也越来越担心他们的数据隐私。例如，大多数人都希望他们的私人医疗数据保持私密而不会公开暴露。但是，这些医疗数据又必须开放给治疗病人的医生和急救人员。许多国家都具有针对这种数据隐私的法律，定义了数据在什么时候以及对谁可以显示。违反隐私法在某些国家能够导致刑事处罚。访问这种隐私数据的应用程序的创建必须小心，谨记隐私法律。

另一方面，聚集后的隐私数据在许多任务中可以扮演重要的角色，比如检测药物的副作用，或者探查流行病的蔓延。如何使这些数据可以为执行这种任务的研究人员所用，而又不侵犯个人的隐私，

这是一个重要的现实问题。例如，假定一家医院隐藏了患者的名字，但是给研究人员提供了患者的出生日期和邮政编码（这二者可能对研究人员都有用）。在很多情况下，仅使用这两个信息就可以唯一标识患者（使用外部数据库中的信息），侵犯了他的隐私。在这个特定情况下，一个解决方法是隐邮政编码一起只提供出生年份而不提供出生日期，这对于研究人员可能就足够了。对大多数人，这将不足以唯一确定一个人。[⊖] 410

另举一个例子，Web 站点通常会收集个人数据，比如地址、电话、电子邮件以及信用卡信息。这种信息在执行交易时可能会需要，比如从商店购买商品。但是，顾客可能不希望这些信息公开给其他组织，或者可能希望其中部分信息（比如信用卡号码）能够在一段时间之后清除，避免它们在发生安全问题时落入未授权的人手中。许多 Web 站点允许客户指定他们的隐私偏好，并且必须确保遵守这些偏好设置。

9.8 加密及其应用

加密是指将数据转换成一种除非使用反过程解密否则不可读的形式的过程。加密算法使用一个加密密钥执行加密，并需要一个解密密钥（它可以和加密密钥相同，这是由使用的加密算法决定的）来执行解密。

加密最早是用于发送消息，用一个只有发送者和接收者知道的密钥加密消息。即使消息被敌方截获，不知道密钥的敌方也将无法解密并理解该消息。今天，加密广泛使用，它在多种应用中保护传输中的数据，比如互联网中的数据传输，以及移动电话网络中的数据传输。我们将在 9.8.3 节中看到，加密还用于执行其他一些任务（比如身份验证）。

在数据库方面，加密用来以一种安全的方式存储数据，从而即使数据被一个未授权的用户获取（例如，一个包含数据的笔记本电脑被偷），如果没有解密密钥的话数据也无法访问。

目前，许多数据库都存储敏感客户信息，比如信用卡号码、姓名、指纹、签名以及身份证号码，例如在美国就是社会保险号。一个获取了这种数据的访问权限的罪犯能够将其用于多种非法活动，比如使用一个信用卡号码购买物品，或者甚至用他人的名字申请信用卡。诸如信用卡公司的机构利用个人信息来识别谁在请求服务或物品。这种个人信息的泄露使罪犯可以假冒其他人并获取服务和物品；这种假冒称作**身份盗窃**（identity theft）。因此，存储这种敏感数据的应用程序必须非常小心地保护它们不被盗用。 411

为了减少敏感信息被罪犯获取的机会，目前许多国家和州通过法律要求任何存储这种敏感信息的数据库必须以加密形式存储信息。因此，一个不保护其数据的企业一旦发生数据盗用，就可以被追究刑事责任。因而，加密是任何存储这种敏感信息的应用程序的重要部分。

9.8.1 加密技术

加密数据的技术数不胜数。简单的加密技术可能无法提供足够的安全性，因为未授权用户可能会轻易地就将编码破译。作为弱加密技术的例子，考虑用字母表中的下一个字母替代每个字母的情况。这样，

Perryridge

就变成了

Qfsszsjehf

如果未授权用户仅仅看到"Qfsszsjehf"，她可能还不具有充足的信息破译编码。但是，如果侵入者看到大量加密后的支行名称，她就能够使用关于字符相对频率的统计数据来猜测所做的替代是如何的（例如，E 是英语中最常用的字母，接下来是 T、A、O、N、I 等）。

一个好的加密技术具有如下性质：

⊖ 对于年纪特别大的人，由于比较稀少，即使只是出生年份加上邮政编码都足以唯一确定一个人。因此对于 80 岁以上的人，可以提供值的一个范围，比如 80 岁及以上，而不是其实际年龄。

- 对于授权用户，加密数据和解密数据相对简单。
- 加密模式不应依赖于算法的保密，而应依赖于被称作加密密钥的算法参数，该密钥用于加密数据。在**对称密钥**(symmetric-key)加密技术中，加密密钥也用于解密数据。相反，在**公钥加密**(public-key)(也称作**非对称密钥**(asymmetric-key))加密技术中，存在两个不同的密钥，公钥和私钥，分别用于加密和解密数据。
- 对入侵者来说，即使当他已经获得加密数据的访问权限了，确定解密密钥仍是极其困难的。在非对称密钥加密的情况中，即使已有公钥，推断出私钥也是极其困难的。

扩展加密标准(Advanced Encryption Standard, AES)是一种对称密钥加密算法，它作为一种加密标准于2000年被美国政府所采用，并且目前广泛使用。该标准基于**Rijndael算法**(Rijndael algorithm)(根
412 据发明人V. Rijmen和J. Daemen命名)。该算法每次对一个128位的数据块操作，密钥的长度可以是128、192或256位。该算法运行一系列步骤，以一种解码时可逆的方式将数据块中的位打乱，并将其与一个来自加密密钥的128位的"回合金钥"做异或操作。对于每一个加密的数据块都由加密密钥生成一个新的回合金钥。在解密过程中，再次根据加密密钥生成回合金钥，并且逆转加密过程从而恢复原始数据。一个称作**数据加密标准**(Data Encryption Standard, DES)的早期标准于1977年采用，并在早期广泛使用。

对于任意对称密钥加密模式的使用，授权用户通过一个安全机制得到加密密钥。这个要求是它的一大弱点，因为模式的安全不高于加密密钥传输机制的安全。

公钥加密(public-key encryption)是另一种模式，它避免了对称密钥加密技术面临的一些问题。它基于两个密钥：一个公钥和一个私钥。每个用户U_i有一个公钥E_i和一个私钥D_i。所有公钥都是公开的：任何人都可以看到。每个私钥都只由拥有它的用户知道。如果用户U_1想要存储加密数据，U_1就使用公钥E_1加密数据。解密需要私钥D_1。

因为加密密钥对每个用户公开，所以我们有可能利用这一模式安全地交换信息。如果用户U_2希望与U_1共享数据，那么U_2就用U_1的公钥E_1来加密数据。由于只有用户U_1知道如何对数据解密，因此信息可以安全地传输。

要使公钥加密发挥作用，在给定公钥后，必须有一个让人很难推断出私钥的加密模式。这样的模式确实存在，并且建立在如下条件基础之上：

- 存在一个高效算法，测试某个数字是否为素数。
- 对于求解一个数的素数因子，没有高效算法。

为了这一模式，数据被看作一组整数。我们通过计算两个大素数P_1和P_2的积来创建公钥。私钥由(P_1，P_2)对构成。如果只知道乘积P_1P_2，解密算法无法使用成功；它需要P_1和P_2各自单独的值。由于公开的只是乘积P_1P_2，因此未授权用户为了窃取数据就需要对P_1P_2做因数分解。通过将P_1和P_2选得足够大(超过100位)，我们可以使对P_1P_2做因数分解的代价极高(即使在最快的计算机上，计算时间也需要以年来计算)。

关于公钥加密的细节以及该技术性质的数学证明请参见文献注解。

尽管使用上述模式的公钥加密是安全的，但是它的计算代价很高。一种广泛用于安全通信的混合
413 模式如下：随机产生一个对称加密密钥(例如基于AES)，使用公钥加密模式以一种安全的方式交换，并使用该密钥对随后传输的数据进行对称密钥加密。

词典攻击(dictionary attack)使得对诸如标识符或名字等小值的加密变得复杂，特别是当加密密钥公开时。例如，如果生日域被加密了，当一个攻击者试图对一个特定的加密值e解密时，他可以试着对任何可能的生日加密，直到他找到一个生日，其加密后的值与e匹配。即使当加密密钥未公开，也可以利用数据分布的统计信息找出一个加密的值在一些情况下代表什么，比如年龄或邮政编码。例如，如果在数据库中18岁是最普遍的年龄，则加密后的年龄值中出现最多的通常可推断出代表18。

可以通过在加密之前往值的末尾添加额外随机位(并在解码后将其删除)来防止词典攻击。这种额外位在AES中称为**初始化矢量**(initialization vector)，或在其他情况下称为*salt*位，它能够对词典攻击提供良好的保护。

9.8.2 数据库中的加密支持

当前，许多文件系统和数据库系统都支持数据加密。这种加密使数据免受那些能够访问数据却无法访问解密密钥的人的攻击。在文件系统加密的情况下，要加密的数据通常是大文件以及包含文件信息的目录。

而在数据库的情况下，加密可以在多个不同级别上进行。在最低的级别上，使用数据库系统软件的密钥，可以加密包含数据库数据的磁盘块。当从磁盘上获取一个磁盘块时，它先被解密然后以平常的方式使用。这种磁盘块级别的加密抵御了那些能访问磁盘内容但不能访问密钥的攻击者。

在一个高级别上，关系中指定的(或所有的)属性可以用加密的形式存储。在这种情况中，关系的每个属性可以具有不同的加密密钥。目前，许多数据库支持指定属性级别的加密以及整个关系所有级别或者数据库中所有关系的加密。指定属性的加密通过让应用程序只对诸如信用卡号码等包含敏感值的属性加密，最小化解密的开销。然而，虽然单个属性或关系可以加密，但是数据库通常不允许主码和外码属性加密，并不支持加密属性上的索引。如前文所述，加密还需要使用额外随机位来防止词典攻击。

为了访问加密的数据，显然需要一个解密密钥。单个主加密密钥可能用于所有的加密数据；在属性级别的加密中，可以对不同的属性使用不同的加密密钥。在这种情况下，可以将不同属性的解密密 414
钥保存在一个文件或关系中(通常称为"钱夹")，它本身也用主密钥加密。

一个需要访问加密属性的数据库的连接必须提供主密钥；除非提供了主密钥，否则该连接将无法访问加密数据。主密钥要保存在应用程序中(通常在一台不同的计算机上)，或者由数据库用户记住，并在用户连接到数据库时提供。

数据库级别的加密具有需要相对小的时空开销的优点，并且不需要对应用程序进行修改。例如，如果笔记本电脑数据库中的数据需要防范来自计算机本身的窃贼，就可以使用这种加密。类似地，访问数据库备份磁带的人，如果没有解密密钥则将不能访问磁带中包含的数据。

在数据库中执行加密的另一个方法是在数据发送到数据库之前对其加密。于是，应用程序则必须在将数据发送给数据库之前对其加密，并当获取到数据时对其解密。与在数据库系统中执行加密不同，这种数据加密方法需要对应用程序进行大量的修改。

9.8.3 加密和认证

基于密码的认证广泛用于操作系统和数据库。然而，密码的使用具有一些缺陷，特别是在网络上的时候。如果一个窃听者能够"窃听"网络上传送的数据，她就可能能够当密码在网络上传送时找到密码。一旦窃听者具有用户名和密码，她就可以假装成合法用户连接到数据库。

一个较为安全的机制包括一个询问－回答(challengE-Response)系统。数据库系统发送一个询问字符串给用户，用户用一个密码作为加密密钥对询问字符串加密，然后返回结果。数据库系统可以通过用同样的密码把字符串解密并检查结果是不是和原始询问字符串一样来验证用户的身份。这个机制确保没有密码在网络上传输。

公钥系统可以在询问－回答系统中用于加密。数据库系统使用用户的公钥加密询问字符串，并把它发送给用户。用户用她的私钥对字符串解密，并把结果返回给数据库系统。数据库系统随后检查该应答。这个方案具有新增的优势，它不在可能被系统管理员看到的数据库中存储密码。

将一个用户的私钥存储在计算机(甚至是个人计算机)中是有风险的，如果该计算机受到攻击，密钥可能会暴露给攻击者。该攻击者就可以假冒该用户。智能卡(smart card)对于该问题提供了一种解决方案。在一张智能卡中，密码可以存储在一块嵌入的芯片中；智能卡的操作系统保证该密码绝对不会 415
被读取，但是允许数据被发送至卡上，使用私钥[⊖]进行加密或解密。

9.8.3.1 数字签名

公钥加密的另一个有趣的应用是**数字签名**(digital signature)，它用来验证数据的真实性；数字签名

⊖ 智能卡也提供了其他功能，比如数字化现金存储和支付，不过这与我们的讨论无关。

扮演文档上物理签名的电子角色。私钥用来对数据"签名"，即加密，且签名后的数据可以公开。所有人都可以通过用公钥解密数据来验证签名，但没有私钥的人无法生成签名的数据。(注意在这个方案中公钥和私钥角色的互换)这样，我们就可以认证(authenticate)该数据：也就是我们可以验证数据确实是由声称创建这些数据的人所创建。

另外，数字签名也可以用来确保认可(nonrepudiation)。也就是，在一个人创建了数据过后声称自己没有创建它(声称没有签支票的电子等价)的情况下，我们可以证明那个人肯定创建了该数据(除非她的私钥被泄露给其他人)。

9.8.3.2 数字证书

通常认证是一个双向的过程。交互实体的双方都要向对方认证自己的身份。这种成对的认证是很有必要的即使当客户端访问一个 Web 站点时，可以防止一个恶意站点冒充成一个合法的 Web 站点。例如，如果网络路由器被攻击了，数据被重新路由到恶意站点的时候，这种冒充就可能发生。

为了确保用户与真实的 Web 站点交互，她必须拥有该站点的公钥。这带来了一个问题：该用户如何获取公钥——如果公钥存储在该 Web 站点上，恶意站点也可以提供一个不同的密钥，用户将无法验证提供的公钥是否是真实的。认证可以通过数字证书(digital certificate)系统来处理，公钥由一个公钥公开的认证机构签名。例如，根认证机构的公钥保存在标准的 Web 浏览器中。它们签署的证书可以使用保存的公钥来验证。

两级系统将会给根认证机构带来创建证书这一巨大负担，因此采用多级系统取而代之。该系统有一个或多个根认证机构，并在每个根认证机构下有一棵认证机构树。每个机构(除了根机构)都有其父机构签署的一个数字证书。

416 认证机构 A 签署的一个数字证书由一个公钥 K_A 和一个可以用公钥 K_B 解密的加密文本 E 构成。该加密文本包括该证书签发给的团体名称以及它的公钥 K_C。在认证机构 A 不是根认证机构的情况下，该加密文本还包含由其父认证机构签发给 A 的数字证书；这个证书认证了密钥 K_A 本身。(该证书可能依次包含上一级父机构的证书，直至根机构)

要验证一个证书，加密文本 E 通过使用公钥 K_B 解密从而获得团体名称(即拥有该网站的机构的名称)；另外，如果 A 不是一个根机构，其公钥 K_A 会递归地使用 E 中包含的数字证书进行验证；递归在由根机构签发的证书到达时终止。对证书的验证建立起一个认证特定站点的链，并且提供了站点的名称以及认证的公钥。

数字证书广泛用于为用户认证 Web 站点，以防止恶意站点冒充成其他 Web 站点。在 HTTPS 协议(HTTP 协议的安全版本)中，站点向浏览器提供它的数字证书，然后浏览器将证书显示给用户。如果用户接受该证书，浏览器则使用提供的公钥来加密数据。一个恶意的站点将能够访问该证书，但是不能访问私钥，因此无法解密浏览器发送的数据。只有拥有相应私钥的真实站点，能够解密浏览器发送的数据。我们注意到公钥/私钥加密和解密的代价远远高于使用对称私钥进行加密/解密的代价。为了减少加密代价，HTTPS 实际上在认证之后创建一个一次性的对称密钥，并在随后的会话中采用该对称密钥加密数据。

数字证书也可以用于认证用户。用户必须向站点提交一个包含她的公钥的数字证书，该站点验证该证书是由一个可信的机构签发。而后该用户的公钥则可以用在询问-应答系统中，以确保用户拥有相应的私钥，并以此认证用户。

9.9 总结

- 在后端使用数据库并与用户交互的应用程序自 20 世纪 60 年代以来一直在使用。应用程序架构在此期间不断发展。目前，大多数应用程序都使用 Web 浏览器作为它们的前端，数据库作为它们的后端，以及一个应用服务器介于其间。
- HTML 提供了定义将超链接与表单功能相结合的界面的能力。Web 浏览器通过 HTTP 协议与 Web 服务器通信。Web 服务器可以将请求传递给应用程序，并将结果返回浏览器。
- 417 Web 服务器执行应用程序以实现所需的功能。servlet 是一个广泛使用的机制，它用于编写能够作为

Web 服务器进程的一部分运行的应用程序，从而减少开销。另外还存在几种由 Web 服务器解释的服务器端脚本语言，作为 Web 服务器的一部分提供应用程序功能。

- 有多种客户端脚本语言——Javascript 使用最为广泛——在浏览器端提供更丰富的用户交互。
- 复杂的应用程序通常具有一个多层的架构，包括一个实现业务逻辑的模型、一个控制器、以及一个用于显示结果的查看机制。它们可能还包括一个实现了对象 - 关系映射的数据访问层。许多应用程序实现并使用 Web 服务，允许在 HTTP 上调用函数。
- 目前已经开发出了许多工具用于快速应用开发，特别是用来减少构建用户界面所需的工作。
- 诸如多种形式的缓存（包括查询结果缓存和连接池）以及并行处理的技术用于提高应用程序性能。
- 应用程序开发人员必须小心注意安全问题，以防止受到攻击，比如 SQL 注入攻击及跨站点脚本攻击。
- SQL 授权机制是粗粒度的，并且对于处理大量用户的应用只具有有限的价值。目前，应用程序完全在数据库系统之外实现了细粒度的、元组级别的授权，以处理大量应用程序用户。虽然元组级访问控制和处理大量应用程序用户的数据库扩展已开发出，但还没有成为标准。
- 保护数据的隐私是数据库应用的一项重要任务。许多国家都有法律规定保护某些类型的数据，比如信用卡信息或医疗数据。
- 加密在保护信息以及认证用户和 Web 站点中扮演了关键角色。对称密钥加密和公钥加密是两种相对立但广泛应用的加密方法。在许多国家和州，对存储在数据库中的某些敏感数据的加密是法律规定的。
- 加密还在为应用程序认证用户，为用户认证 Web 站点以及数字签名中扮演了关键角色。

术语回顾

- 应用程序
- 数据库的 Web 界面
- 超链接
- 统一资源定位符（URL）
- 表单
- 超文本传输协议（HTTP）
- 公共网关接口（CGI）
- 无连接协议
- cookie
- 会话
- servlet 及 servlet 会话
- 服务器端脚本
- JSP
- PHP
- ASP. NET
- 客户端脚本
- Javascript
- 文档对象模型（DOM）
- 小应用程序
- 应用程序架构
- 展示层
- 模型 - 视图 - 控制器（MVC）架构
- 业务逻辑层
- 数据访问层
- 对象 - 关系映射
- Hibernate
- 超文本标记语言（HTML）
- Web 服务
- REST 服务
- 快速应用开发
- Web 应用框架
- 报表生成器
- 连接池
- 查询结果缓存
- 应用程序安全性
- SQL 注入
- 跨站点脚本（XSS）
- 跨站点请求伪造（XSRF）
- 认证
- 双因素认证
- 中间人攻击
- 中央认证
- 单点登录
- OpenID
- 虚拟私有数据库（VPD）
- 审计追踪
- 加密
- 对称密钥加密
- 公钥加密
- 词典攻击
- 询问 - 回答
- 数字签名
- 数字认证

实践习题

9.1 虽然 Java 程序通常要比 C 或 C++ 程序运行得慢，而 servlet 却比使用公共网关接口（CGI）的程序性能好的主要原因是什么？

9.2 列出无连接协议相比较有连接协议的优缺点。

9.3 考虑为在线购物系统写的一个未经仔细编写的 Web 应用程序，该应用程序将每件商品的价格以隐藏的表单变量保存在发送给顾客的页面中；当顾客提交表单时，隐藏的表单变量中的信息用于计算顾客的账单。这个设计中的漏洞在哪里？（为一个真正的实例，其中在探测并解决问题之前，在线购物系统

418
|
419

的顾客使该漏洞暴露了出来。)

9.4 考虑未经仔细编写的另一个 Web 应用程序，它使用 servlet 检查是否有活动的会话，但并不检查用户是否授权访问某页面，而依靠指向页面的连接只显示给已授权用户的事实。这种机制下的缺陷是什么？(有一个真正的实例，其中某个学校招生站点的申请者在登录网站后可以利用这个漏洞，并且可以查看他们没有授权查看的信息；然而，此未授权的访问被探测出来了，访问该信息的人由于违反招生规定而被驳回。)

9.5 列出使用缓存提高 Web 服务器性能的三种方式。

9.6 netstat 命令(在 Linux 和 Windows 中可用)显示计算机上活动的网络连接。试解释如何使用该命令以发现是否某网页将不关闭它所打开的连接，或者是否使用了连接池并且不将连接返回连接池。你应该考虑到使用连接池的时候，连接可以不立即关闭。

9.7 SQL 注入漏洞的检测：

a. 提出一个检测应用程序的方法，以发现在文本输入上是否容易受到 SQL 注入攻击。

b. SQL 注入能够与其他形式的输入一同出现吗？如果可以，你要如何检测漏洞呢？

9.8 为了安全，一个数据库关系可能对某些属性的值加密。数据库系统为什么不支持加密属性上的索引呢？用你对这个问题的答案解释为什么数据库系统不允许主码属性上的加密。

9.9 练习 9.8 提出了在某些属性上加密的问题。然而，一些数据库系统支持整个数据库的加密。试解释练习 9.8 中提出的问题在加密整个数据库时该如何避免。

9.10 假设某人获得一个公司并得到了一个证书授予机构颁发的证书。这对于被假冒公司认证的事物(例如购买订单或者程序等)，以及对于其他公司认证的事物有什么影响？

420 9.11 在任何数据库系统中，最重要的数据项或许都是用来控制数据库访问的密码。为密码的安全存储设计一个机制，确保你的机制允许系统检测由试图登录系统的用户所提供的密码。

习题

9.12 为下面这个非常简单的应用写一个 servlet 以及关联的 HTML 代码：允许用户提交一个表单，该表单包含一个值，比如说 n，然后得到的响应包含 n 个“x”号。

9.13 为下面这个简单应用写一个 servlet 以及关联的 HTML 代码：允许用户提交一个表单，该表单包含一个数字，比如说 n，然后得到一个响应说出数值 n 在此前提交的次数。每个数值此前提交的次数应当存储在数据库中。

9.14 写一个对用户进行认证的 servlet(基于存储在数据库关系中的用户名和密码)，并在验证通过之后设置一个名为 *userid* 的会话变量。

9.15 什么是 SQL 注入攻击？解释它的工作原理，以及必须采取什么措施以预防 SQL 注入攻击。

9.16 编写管理连接池的伪码。你的伪码必须包括一个创建连接池的函数(以数据库连接字符串、数据库用户名和密码作为输入参数)，一个向连接池请求连接的函数，一个将连接归还连接池的函数以及关闭连接池的函数。

9.17 解释术语 CRUD 和 REST。

9.18 目前，许多网站都使用 Ajax 提供丰富的用户界面。列出两个不需要查源码即可表明站点使用 Ajax 的特点。利用上述特点，找到三个使用 Ajax 的站点；你可以查看网页的 HTML 源码以检验该网站是否真正使用了 Ajax。

9.19 XSS 攻击：

a. XSS 攻击是什么？

b. 如何使用引用页域以探查 XSS 攻击？

9.20 什么是多因素认证？它是如何帮助防止密码被盗的？

9.21 考虑 9.7.5 节所述的 Oracle **虚拟私有数据库**(VPD)功能，以及一个基于大学模式的应用。

421
- 应该生成什么谓词(使用了查询)以使每个教员只看见和他们所教课程相关联的 *takes* 元组。
- 给出一个 SQL 查询，使得添加了谓词的查询的结果是未添加谓词的原始查询的结果的子集。
- 给出一个 SQL 查询，使得添加了谓词的查询的结果包含一条不在未添加谓词的原始查询的结果中的元组。

9.22 对存储在数据库中的数据加密有哪两个好处?

9.23 假定你希望对关系 *takes* 的变化建立审计追踪:

a. 定义触发器来建立审计追踪,把日志信息记入一个关系,例如 *takes_trail*。日志信息应包括用户 id(假设函数 *user_id*()提供该信息)和一个时间戳,以及旧的和新的值。你还要提供 *takes_trail* 关系的模式。

b. 以上实现方法是否能保证自愿滥数据库管理员(或者那些设法得到管理员密码的人)所做的更新也会被审计追踪。解释你的答案。

9.24 黑客能够欺骗你相信他们的 Web 站点实际上是你所信任的一个 Web 站点(比如一家银行或者信用卡 Web 站点)。他们可以使用误导性电子邮件,或者甚至通过入侵网络基础设施将目的地为(比如) *mybank.com* 的网络传输重新路由到黑客的网站。如果你在黑客的站点上输入用户名和密码,该站点就会把它记录下来,以后就能用它来侵入你在真实站点的账户。当你使用一个 URL,比如 *https://mybank.com*,HTTPS 协议可以用来防止这样的攻击。解释该协议如何使用数字证书来验证站点的合法性。

9.25 解释什么是用于验证的询问-应答系统。为什么它比传统的基于密码的系统更安全?

项目建议

下面都是较大的项目,可以由一组学生在一个学期内完成。项目的难度可以通过增加或者减少功能来调整。

项目9.1 选一个你最喜爱的交互网站,比如 Bebo, Blogger, Facebook, Flickr, Last.FM, Twitter, WikiPedia;这只是一些例子,还有更多其他的交互网站。这些网站中的大部分都管理大量的数据,并使用数据库存储并处理这些数据。实现你挑选的一个网站的功能的子集。需要注意的是,即使实现这种网站的功能的一个有意义的子集也远超过了一个课程设计要求的工作量,不过可以找一些有趣的功能来实现,足够满足课程设计的工作量即可。 [422]

目前流行的网站大部分都大量使用 JavaScript 创建丰富的界面。指定这种界面非常耗时,所以至少在最初你可能会希望用它实现你的项目的界面,然后在时间允许的情况下再添加更多的功能。利用 Web 应用开发框架或网上的 Javascript 库,比如 Yahoo 用户界面库,来加快你的开发。

项目9.2 创建一个"混合网站",即用诸如 Google 或 Yahoo 地图 API 的 Web 服务创建一个交互网站。例如,这些地图 API 提供一种在网页上把小地图及在地图上重叠的其他信息的方法。你可以实现一个餐厅推荐系统,使用户提交餐厅信息,比如位置、菜肴、价格范围以及评分信息。用户搜索的结果可以在地图上显示。你可以允许类 Wikipedia 的功能,比如让用户可以添加信息并编辑其他用户添加的信息,并安排可以删除恶意更新的审核人。你还可以实现社区功能,比如给你的朋友提供的评分更高的权重。

项目9.3 你的学校可能使用了一个课程管理系统,比如 Moodle, Blackboard 或 WebCT。实现这样一种课程管理系统的功能的子集。例如,你可以提供作业提交和评分功能,包括为学生和老师/助教提供讨论作业的评分的机制。你还可以提供问卷调查或其他机制以获得反馈。

项目9.4 考虑实践习题7.3(第7章)中的 E-R 模型,它表示一个联赛中各队的信息。设计并实现一个基于 Web 的系统来输入、更新和查看这些数据。

项目9.5 设计并实现一个购物车系统,该系统可以让购买者将商品放置到购物车中(你可以决定为每件商品提供的信息),最后一起购买。你可以扩展并使用第7章中习题7.20中的 E-R 模式。你应该检查商品是否有货并按你觉得适合的方式处理商品缺货的情况。

项目9.6 设计并实现一个基于 Web 的系统为一所大学的课程记录学生注册信息和成绩信息。

项目9.7 设计并实现一个系统记录学生表现信息——具体来说,即每个学生在可门课程中的每次作业或考试中所获得的分数,以及对各项成绩进行(加权)求和以得到课程总成绩。作业/考试的数量不可以预先定义;即,任何时候都可以加入更多的作业或考试。该系统还应该可以支持等级评定,允许对于不同的级别指定分贴点。 [423]

你也可以将它与项目9.6中的学生注册系统(可能由另一个项目组实现)相结合。

项目 9.8 设计和实现一个基于 Web 的系统，用来在你的学校中预订教室。它必须支持周期性的预订（整个学期中每周固定的天/时间），也要支持在周期性预订中取消某个指定的讲座。

你也可以将它与项目 9.6 中的学生注册系统（可能由另一个项目组实现）相结合，这样可以为课程预订教室，取消讲座或新增额外讲座可以通过一个单独的界面来记录，并且要将其反映到教室预订情况中，然后通过电子邮件通知给所有的学生。

项目 9.9 设计并实现一个系统，管理在线的多选题考试。你应该支持分布式的考题提交（由助教提交）。负责该课程的人可以编辑题目，并且可以从已有的考题集合中创建试卷。你还应该能够在线管理考试，可以让所有学生都在一个固定的时间参加考试；或者让学生在任何时间都可以开始考试，但是从开始到结束有一个时间限制（支持一种或者两种方案都支持）。并且规定时间结束时将成绩反馈给学生。

项目 9.10 设计并实现一个系统，管理电子邮件顾客服务。到达的电子邮件进入一个公用的缓冲池中。一组顾客服务代理人负责答复电子邮件。如果某封电子邮件是正在答复的序列的一部分（用电子邮件的 in-reply-to 域来跟踪），这个邮件应该由最先答复它的那个代理人来答复。系统应该追踪所有到达的邮件和答复，这样代理人可以在答复一封电子邮件之前看到顾客以前问过的所有问题。

项目 9.11 设计并实现一个简单的电子商场。该电子商场可以按照不同的类别（应该形成一个层次结构）列出用于销售或购买的商品。你可能还希望支持提醒服务：一个用户可以在某一个特定类别中注册自己感兴趣的商品，也许还有一些其他限制，而不公开她的兴趣。当这样的商品出现时，系统会通知该用户。

项目 9.12 设计并实现一个基于 Web 的新闻组系统，用户应该可以订阅新闻组，并且浏览新闻组中的文章。该系统跟踪用户阅读过的文章使它们不会再次显示。该系统还提供对旧文章的搜索支持。你可能还希望提供文章的评分服务，这样，高分的文章高亮度显示，从而使繁忙的用户可以跳过低分的文章。

424

项目 9.13 设计并实现一个基于 Web 的系统，管理一个运动"锦标"。许多人来注册，并给予初始的排名（可能基于以前的表现）。任何人可以通过比赛挑战其他人，而排名按照比赛结果进行相应的调整。一个调整排名的简单系统仅是当胜者原先排在败者后面时，在排名中将胜者移到败者的前面。你也可以尝试实现更加复杂的排名调整系统。

项目 9.14 设计并实现一个出版物列表服务。这个服务应该允许输入有关出版物的信息，例如书号、作者、年份、出版物出现地点、页数等。作者应该是一个具有属性（如名字、机构、部门、电子邮件、地址和主页）的单独实体。

你的应用应该支持同一数据的多个视图。例如，你应该提供某个给定作者的所有出版物（比如按年份排序），或者提供来自某个给定机构或部门的作者的所有出版物。你也应该支持在整个数据库或每个视图上按关键词搜索。

项目 9.15 所有组织中的一项常见任务是收集一组人的结构化信息。例如，一个经理可能需要要求职员输入他们的休假计划；一个教授可能希望从学生中收集关于某个特定主题的反馈；或者一个组织活动的学生希望允许其他学生注册活动，或者某人希望针对某个主题进行一个在线投票。

建立一个允许用户很容易创建信息收集活动的系统。当创建一个活动时，该活动的创建人必须定义谁是有资格参加的，为了达到这个目的，你的系统必须维护用户信息，并且允许定义一个用户子集的谓词。活动的创建人应当能够指定一组用户需要提供的输入（有类型，默认值和有效性检查）。活动应当关联一个截止时间，并能够向尚未提交信息的用户发送通知。活动创建者可以选择是在指定的日期或时间自动执行截止，还是登录系统然后宣布截止。提交信息的统计信息能够生成——为了达到这个目的，应当允许活动创建者对输入的信息建立简单的汇总。活动创建者可以选择将部分汇总信息公开以让所有用户都能查看，要么持续地（比如有多少人已经响应），要么在截止期之后（比如平均回馈分值）。

项目 9.16 建立一个函数库以简化 Web 界面的生成。你必须至少实现以下函数：显示 JDBC 结果集（以表格形式）的函数，创建不同种类的文本和数字输入（带有验证规则，比如输入类型和可选

范围，在客户端用适当的 Javascript 代码执行）的函数，输入日期和时间值（带有默认值）的函数，基于结果集生成菜单项的函数。要得到加分，可以允许用户设置格式参数，比如颜色和字体，以及在表单中提供分页支持（隐藏的表单参数可以用来指示需要显示的页）。建立一个数据库应用样例以展示这些函数的使用。 [425]

项目 9.17 设计并实现一个基于 Web 的多用户日历系统。该系统需追踪每个用户的约会。包括多发事件，如聚会，及共享事件（事件创建者更新事件将反映至所有分享该事件的用户）。系统需提供安排多用户事件的界面，该界面允许事件创建者添加受邀请的用户。系统需提供事件的电子邮件通知。要得到加分，可以实现一个 Web 服务，该服务可以用在客户端上运行的报醒程序中。

工具

开发一个 Web 应用需要多个软件工具，比如应用服务器、编译器、像 Java 或 C#这样的程序设计语言的编辑器以及其他可选的工具，比如 Web 服务器。有一些对 Web 应用开发提供支持的集成开发环境。两款最流行的开源 IDE 为 IBM 开发的 Eclipse 以及 Sun 领导开发的 Netbeans。微软的 Visual Studio 是在 Windows 环境中使用最为广泛的 IDE。

支持 servlet 和 JSP 的应用服务器包括 Apache Tomcat（jakarta.apache.org）、Glassfish（glassfish.dev.java.net）、JBoss（jboss.org）以及 Caucho 的 Resin（www.caucho.com）。Apache Web 服务器（apache.org）是目前使用最为广泛的 Web 服务器。微软的 IIS（Internet Information Services）是一款广泛用于微软 Windows 平台并支持微软的 ASP.NET（msdn.microsoft.com/asp.net/）的 Web 兼应用服务器。

IBM 的 WebSphere（www.software.ibm.com）为 Web 应用开发和部署提供了许多软件工具，包括应用服务器、IDE、应用集成中间件、业务流程管理软件以及系统管理工具。

上述工具中有些是开源软件可以免费使用，有些对于非商业用途或个人使用是免费的，而另外一些则需要付费。更多信息可查看各相关网站。

Yahoo! 用户界面（YUI）的 JavaScript 库（developer.yahoo.com/yui）广泛用于创建浏览器的 JavaScript 程序。

文献注解

有关 servlet 的信息，包括指南、标准说明和软件，可以在 java.sun.com/products/servlet 上找到。有关 JSP 的信息可以在 java.sun.com/products/jsp 上找到。关于 JSP 标签库的信息也能在该 URL 上找到。关于 .NET 框架的信息以及关于使用 ASP.NET 进行 Web 应用开发的信息可以在 msdn.microsoft.com 上找到。

Atreya 等[2002]提供了部署数字签名的教科书，其中包含 X.509 数字证书以及公钥基础设施。 [426]

第三部分
Part 3

数据存储和查询

虽然数据库系统提供了数据的高层视图，但最终数据必须以比特的形式存储在一个或多个存储设备上。如今，绝大多数的数据库将数据存储在磁盘上（而且越来越多地在闪存上），并将数据取入内存用于处理，或者将数据转贝到磁带和其他备份设备上用于归档存储。存储设备的物理特性对数据的存储方式影响重大，这尤其是因为对磁盘上随机数据片段的访问比内存访问慢得多：磁盘访问需要几十毫秒，而内存访问只需十分之一微秒。

第 10 章从物理存储介质的概览开始，包括将由于故障而引起数据丢失的可能性最小化的机制。然后，该章描述记录是如何映射到文件，然后又如何映射到磁盘中的比特的。

许多查询只涉及了文件中一小部分的记录。索引是一种有助于无须检查所有记录而快速定位所需关系记录的结构。本教材的索引就是一个索引的实例，尽管它是为人使用的，这与数据库索引不同。第 11 章描述数据库系统使用的几种索引类型。

用户查询的执行必须基于存放在存储设备中的数据库的内容。我们可以方便地将查询分成一些更小的操作。这些操作可以大致对应于关系代数操作。第 12 章描述如何处理查询，并给出用于实现单独操作的算法，然后概述为了处理查询，这些操作是如何同步执行的。

处理一个查询有许多可选的方法，这些方法的执行代价有很大的不同。查询优化是指寻找执行一个给定查询的最低代价方法的过程。第 13 章描述查询优化过程。

第10章
Database System Concepts, 6E

存储和文件结构

前面各章强调了数据库的较高层模型。例如，在概念层或逻辑层上，我们认为关系模型中的数据库是表的集合。的确，数据库用户需要关注的层次正是数据库的逻辑模型。因为数据库系统的目标是简化和协助对数据的访问，数据库系统的用户不必被系统实现的物理细节所困扰。

但是，在本章以及第11章、第12章和第13章，作为对较低层的研究，我们将描述实现前几章给出的数据模型和语言的不同方法。我们将从基本存储介质（如磁盘和磁带系统）的特性开始，然后我们将定义不同的数据结构，这些数据结构使我们能够快速地访问数据。我们将考虑几种可供选择的数据结构，各种数据结构适合于不同类型的数据访问。数据结构的最终选择依赖于系统的使用方式和特定机器的物理特性。

10.1 物理存储介质概述

大多数计算机系统中存在多种数据存储类型。可以根据访问数据的速度、购买介质时每单位数据的成本，以及介质的可靠性对这些存储介质进行分类。以下是几种有代表性的介质：

- **高速缓冲存储器**（cache）。高速缓冲存储器是最快最昂贵的存储介质。高速缓冲存储器一般很小，由计算机系统硬件来管理它的使用。在数据库系统中，我们不需要考虑高速缓冲存储器的存储管理。但值得注意的是，在设计盘高处理的数据结构和算法时，数据库实现者也会注意高速缓冲存储器的影响。

- **主存储器**（main memory）。主存储器是用于存放可处理的数据的存储介质。通用机器指令在主存储器上执行。尽管在个人电脑上的主存储器可以包含几个GB的数据，甚至在大型服务器系统中包含数百个GB的数据，但是一般情况下它对于存储整个数据库来说还是太小（或太昂贵）。如果发生电源故障或者系统崩溃，主存储器中的内容通常会丢失。
- **快闪存储器**（flash memory）。快闪存储器不同于主存储器的地方是在电源关闭（或故障）时数据可保存下来。目前有两种类型的快闪存储器，称作*NAND*和*NOR*快闪。对于既定价格，*NAND*快闪拥有更高的存储容量，并且广泛作为一些设备的数据存储使用，例如照相机、音乐播放器和手机，同时也越来越多地用于笔记本电脑。相比于主存储器，快闪存储器拥有每字节更低的价格，以及具有非易失性，即便电源切断，它也能保留存储的数据。

快闪存储器还广泛地用于在"USB盘"中存储数据，它可以插入电脑设备的通用串行总线（Universal Serial Bus，USB）槽。这种USB盘已经成为在计算机系统之间传输数据的流行手段（以前的软盘起相同的作用，但如今它因有限的容量而废弃）。

在存储中等数量数据时，快闪存储器也在作为磁盘存储器的替代品越来越多地使用。这种磁盘驱动器替代品就是所谓的固态驱动器（solid-state drive）。在2009年，一个64GB的固态硬盘驱动器售价不到200美元，并且容量范围可达160GB。此外，快闪存储器广泛使用在服务器系统中，通过缓存经常使用的数据来提高性能，因为它提供比磁盘更快的访问速度，和比主存储器更大的存储容量（对于既定价格）。

- **磁盘存储器**（magnetic-disk storage）。用于长期联机数据存储的主要介质是磁盘。通常整个数据库都存储在磁盘上。为了能够访问数据，系统必须将数据从磁盘移到主存储器。在完成指定的操作后，修改过的数据必须写回磁盘。

在 2009 年，磁盘容量从 80GB 到 1.5TB，并且 1TB 的磁盘价格约为 100 美元。磁盘容量以大约每年 50% 的速度增长，而且每年我们都可以期待更大容量的磁盘出现。磁盘存储器不会因为系统故障和系统崩溃丢失数据。磁盘存储设备本身有时可能会发生故障，导致数据的损坏，但是发生磁盘故障的概率比发生系统崩溃的概率小得多。

- **光学存储器**（optical storage）。光学存储器最流行的形式是光盘（Compact Disk，CD），它可以容纳大约 700MB 的数据，播放约 80 分钟。数字视频光盘（Digital Video Disk，DVD）的每一盘面可以容纳 4.7GB 或者 8.5GB 的数据（一张双面盘最多可以容纳 17GB 的数据）。可以用**数字万能光盘**（digital versatile disk）代替**数字视频光盘**（digital video disk）作为 DVD 的全称，因为 DVD 可以存储任何数字数据，而不仅仅是视频数据。数据通过光学的方法存储到光盘上，并通过激光器读取。一种称作蓝光（*Blu-ray*）DVD 的更高容量格式可以单层存储 27GB 或双层存储 54GB 数据。

  用于只读光盘（CD-ROM）和只读 DVD（DVD-ROM）的光盘是不可写的，但是提供预先记录的数据。有些"记录一次"的光盘（CD-R）和 DVD（DVD-R 和 DVD+R）只可以写一次，这样的光盘也称为**写一次读多次**（Write-Once，Read-Many，WORM）光盘。也有"写多次"的光盘（CD-RW）和 DVD（DVD-RW，DVD+RW 和 DVD-RAM），它们都可以多次写入数据。

  自动光盘机（jukebox）系统包含少量驱动器和大量可按要求（通过机械手）自动装入某一驱动器的光盘。

- **磁带存储器**（tape storage）。磁带存储器主要用于备份数据和归档数据。尽管磁带比磁盘便宜得多，但是访问数据也比磁盘慢得多，这是因为磁带必须从头顺序访问。因为这个原因，磁带存储器称为顺序访问（sequential-access）的存储器。相对而言，磁盘存储器称为直接访问（direct-access）的存储器，因为它可以从磁盘的任何位置读取数据。

  磁带具有很大的容量（现在的磁带可以容纳 40～300GB 的数据），并且可以从磁带设备中移出，因此它们非常适合进行便宜的归档存储。磁带库（自动磁带机）用于存储异常巨大的数据集合，比如可能有几百 TB（$1TB = 10^{12}$ 字节）的卫星数据，或少数情况下甚至是若干 PB 的数据（$1PB = 10^{15}$ 字节）。

根据不同存储介质的速度和成本，可以把它们按层次结构组织起来（见图 10-1）。层次越高，这种存储介质的成本就越贵，但是速度就越快。当我们沿着层次结构向下，存储介质每比特的成本下降，但是访问时间会增加。这种权衡是合理的：如果某一存储系统比另一种存储系统快并且便宜（其他特性相同），那么就没有理由使用那种又慢又昂贵的存储系统。实际上，当磁带和半导体存储器变得更快更便宜时，很多早期存储设备，包括纸带机和磁心存储器，都进了博物馆。当磁盘还很昂贵并且只有很小的存储容量时，我们用磁带存储正在使用的数据。但是在今天，几乎所有正在使用的数据都存储在磁盘上，只在极少数情况下存储在磁带或自动光盘机中。

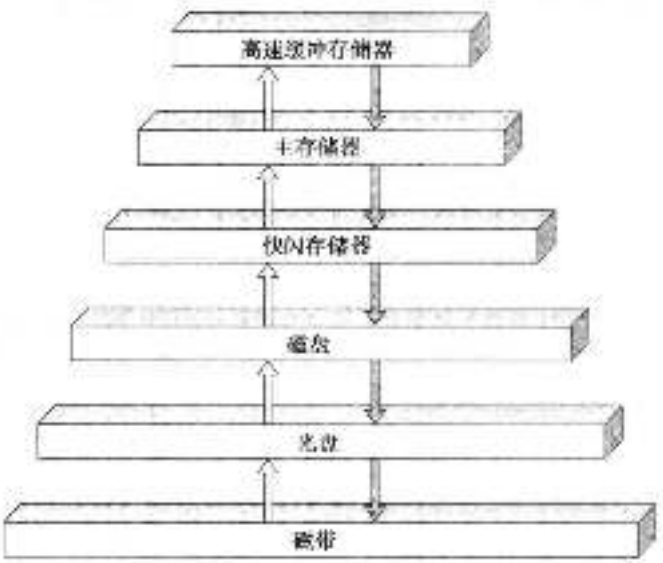


图 10-1　存储设备层次结构

最快的存储介质（例如高速缓冲存储器和主存储器）称为**基本存储**（primary storage）。层次结构中基本存储介质的下一层介质（例如磁盘）称为**辅助存储**（secondary storage）或**联机存储**（online storage）。层次结构中最底层的介质（如磁带

机和自动光盘机)称为三级存储(tertiary storage)或脱机存储(offline storage)。

不同存储系统除了速度和成本不同之外，还存在一个存储易失性的问题。易失性存储(volatile storage)在没有断电后将丢失所有内容。在图 10-1 所示的层次结构中，从主存储器向上的存储系统都
[431] 是易失性存储，而主存储器之下的存储系统都是非易失性存储。为了保护数据，必须将数据写到非易失性存储(nonvolatile storage)中去。我们将在第 16 章重新阐述这个问题。

10.2 磁盘和快闪存储器

磁盘为现代计算机系统提供了绝大部分的辅助存储。尽管磁盘容量每年都在增长，但是大型应用对存储容量的需求也在快速增长，在有些情况下甚至比磁盘容量的增长还要快。一个非常大型的数据库可能需要上百张磁盘。近几年来，快闪存储器存储容量快速增加，并且在一些应用中逐渐成为磁盘存储的竞争者。

10.2.1 磁盘的物理特性

磁盘在物理上是相对简单的(见图 10-2)。磁盘的每一个盘片(platter)是扁平的圆盘。它的两个表面都覆盖着磁性物质，信息就记录在表面上。盘片由铝金属或玻璃制成。

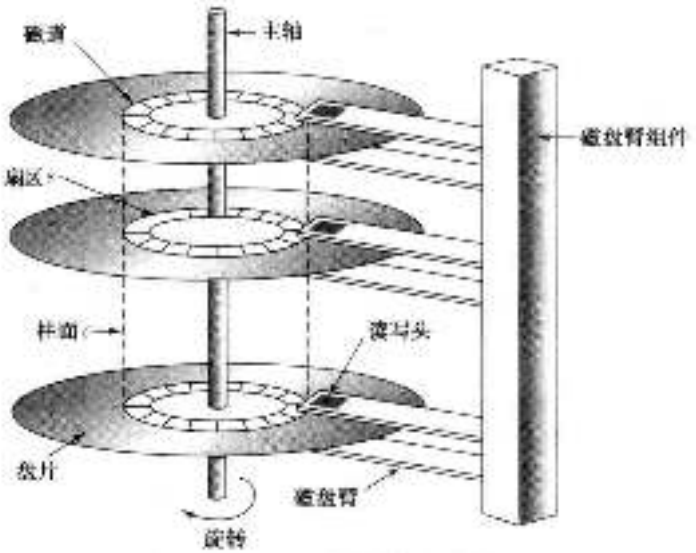


图 10-2 磁盘移动头机制

当磁盘被使用时，驱动马达使磁盘以恒高的恒定速度旋转(通常为每秒 60、90 或 120 转，也可达到每秒 250 转)。有一个读写头恰好位于盘片表面的上方。盘片的表面从逻辑上划分为**磁道**(track)，磁道又划分为**扇区**(sector)。扇区是从磁盘读出和写入信息的最小单位。对于现在的磁盘，扇区大小一般为 512 字节，每一个盘片有约 50 000 ~ 100 000 条磁道，每条磁盘有 1 ~ 5 个盘片。内侧的磁道(离转轴近的地方)长度较短，在现代的磁盘中，外侧的磁道比内侧的磁道拥有更多的扇区。一般而言，内侧每磁道大约包含 500 ~ 1000 个扇区，而外侧每磁道大约包含 1000 ~ 2000 个扇区。对于不同模式的磁盘，上面的数字会有变化。高容量的模式通常在每条磁道上含有更多的扇区，而且在每个盘片上有更多的磁道。

通过反转磁性物质磁化的方向，**读写头**(read-write head)将信息磁化存储到扇区中。

磁盘的每个盘片的每一面都有一个读写头，读写头通过在盘片上移动来访问不同的磁道。一张磁盘通常包括很多个盘片，所有磁道的读写头安装在一个称为**磁盘臂**(disk arm)的单独装置上，并且一起移动。安装在转轴上的所有磁盘盘片和安装在磁盘臂上的所有读写头统称为**磁头 - 磁盘装置**(head-disk assembly)。因为所有盘片上的读写头一起移动，所以当某一个盘片的读写头在第 i 条磁道上时，所有其他盘片的读写头也都在各自盘片的第 i 条磁道上。因此，所有盘片的第 i 条磁道合在一

起称为第 i 个**柱面**(cylinder)。

现在，市场上占主导地位的是盘片直径为 3.5 英寸的磁盘。它们比早先流行的大直径磁盘(可以达到 14 英寸)成本低、寻道时间快(由于寻道距离变短)，而且还可以提供更大的存储容量。小直径磁盘用于便携设备(如笔记本电脑，以及手持电脑和便携音乐播放器)。 432

为了增大记录密度，读写头尽可能地靠近磁盘盘片的表面。读写头一般浮于盘片表面之上几微米；磁盘的旋转产生气流，而磁头装置的形状使其在气流作用下恰好浮于盘片表面之上。因为读写头离盘片表面非常近，所以盘片必须制作得非常平。 433

读写头损坏是一个问题。如果读写头接触到盘片的表面，读写头会刮掉磁盘上的记录介质，破坏存放在那里的数据。在早期的磁盘上，读写头接触盘片表面所刮掉的介质将散落到其他盘片及其读写头之间，引起更多的损坏，因此一个读写头的损坏可能导致整张磁盘失效。目前使用的磁盘驱动器使用一层磁金属薄膜作为存储记录的介质。这样的磁盘和以前的涂氧化膜的磁盘相比，不易因为读写头损坏而产生故障。

磁盘控制器(disk controller)作为计算机系统和实际的磁盘驱动器硬件之间的接口。在现代磁盘系统中，磁盘控制器在磁盘驱动单元内部实现。它接受高层次的读写扇区的命令，然后开始动作，如移动磁盘臂到正确的磁道，并实际地读写数据。磁盘控制器为它所写的每个扇区附加**校验和**(checksum)，校验和是从写到扇区中的数据计算得到的。当读取一个扇区时，磁盘控制器用读到的数据再一次计算校验和，并且把它与存储的校验和比较。如果数据被破坏，则新计算出的校验和与存储的校验和不一致的可能性就很高。如果发生了这样的错误，磁盘控制器就会重读几次；如果错误继续发生，磁盘控制器就会发出一个读类作失败的信号。

磁盘控制器执行的另一个有趣的任务是**坏扇区的重映射**(remapping of bad sector)。当磁盘初始格式化时，或试图写一个扇区时，如果磁盘控制器检测到一个损坏的扇区，它会把这个扇区在逻辑上映射到另一个物理位置(从为此目的而留出的额外扇区中分配)。重映射记录在磁盘或非易失性存储器中，而写操作在新的位置上执行。

磁盘通过一个高速互连通道连接到计算机系统。有大量的通用接口用于将磁盘连接到计算机。现今一般使用的接口是：(1)SATA，也叫串行 ATA(serial ATA [台])，最新版本的 SATA 称作 SATA II 或者 SATA3 Gb(ATA 标准的旧版本称作 PATA，或并行 ATA，或者 IDE，在早期广泛地使用，并且现在依旧可用)，(2)小型计算机系统互联(Small-Computer-System Interconnect，SCSI，发音为"scuzzy")，(3)SAS，也叫串行附着 SCSI(serial attached SCSI)，和(4)光纤通道接口(Fibre Channel Interface)。便携式外部磁盘系统经常使用 USB 接口或 IEEE 1394 火线接口。

磁盘通常可以通过电缆直接与计算机系统的磁盘接口相连，也可以放置到远端并通过高速网络与磁盘控制器相连。在**存储区域网**(Storage Area Network，SAN)体系结构中，大量的磁盘通过高速网络与许多计算机服务器相连。通常磁盘采用**独立磁盘冗余阵列**(Redundant Array of Independent Disk，RAID，将在 10.3 节介绍)技术进行本地化组织，从而给服务器一个很大且非常可靠的磁盘的逻辑视图。尽管可能被网络分开，计算机和磁盘子系统继续通过 SCSI，SAS 和光纤通道接口互相通信。通过存储区域网对磁盘的远程访问，意味着磁盘可以由并行运行一个应用程序的不同部分的多台计算机所共享。远程访问同时也意味着包含重要数据的磁盘可以存放在有系统管理员监视和维护的集中式服务器房间中，而不是将它们分散在一个组织的不同部分中。 434

网络附加存储(Network Attached Storage，NAS)由 SAN 发展而来。NAS 很像 SAN，但是它通过使用网络文件系统协议(如 NFS 或 CIFS)提供文件系统接口，而不是看似一张大磁盘的网络存储器。

10.2.2　磁盘性能的度量

磁盘质量的主要度量指标是容量、访问时间、数据传输率和可靠性。

访问时间(access time)是从发出读写请求到数据开始传输之间的时间。为了访问(即读或写)磁

台　自 20 世纪 80 年代，ATA 是一种存储设备连接标准。

盘上指定扇区的数据，磁盘臂首先必须移动，以定位到正确的磁道，然后等待磁盘旋转，直到指定的扇区出现在它下方。磁盘臂重定位的时间称为**寻道时间**(seek time)，它随磁盘臂移动距离的增大而增大。典型的寻道时间在 2～30 毫秒之间，依赖于目的磁道距离磁盘臂的初始位置有多远。较小的磁盘因为移动距离较短而寻道时间较短。

平均寻道时间(average seek time)是寻道时间的平均值，是在一个(均匀分布的)随机请求的序列上计算得到的。假设所有的磁道包含相同的扇区数，同时我们忽略读写头开始移动和结束移动所花费的时间，我们可以得到平均寻道时间是最坏情况下寻道时间的 1/3。考虑到前面忽略的这些因素，平均寻道时间大约是最长寻道时间的 1/2。现在，平均寻道时间在 4～10 毫秒之间，依赖于磁盘模式。

一旦读写头到达了所需的磁道，等待访问的扇区出现在读写头下所花费的时间称为**旋转等待时间**(rotational latency time)。现在一般的磁盘转速在每分钟 5400 转(每秒 90 转)到每分钟15 000转(每
435 秒 250 转)之间，或者等价地，在每转 4 毫秒到每转 11 毫秒之间。平均情况下，磁盘需要旋转半周才能使所要访问的扇区开始处于读写头的下方。因此磁盘的**平均旋转等待时间**是磁盘旋转一周时间的 1/2。

访问时间是寻道时间和旋转等待时间的总和，范围在 8～20 毫秒之间。一旦待访问数据的第一个扇区来到读写头下方，**数据传输**就开始了。**数据传输率**(data-transfer rate)是从磁盘获得数据或者向磁盘存储数据的速率。目前的磁盘系统支持每秒 25～100MB 的数据最大传输率。对于磁盘的内侧磁道，数据传输率明显低于最大传输率，例如，一张最大传输率为每秒 100MB 的磁盘，其内侧磁道持续传输率约为每秒 30MB。

最后一个经常使用的磁盘度量标准是**平均故障时间**(Mean Time To Failure，MTTF)，这是磁盘可靠性的度量标准。磁盘(或其他任何系统)的平均故障时间是，平均说来我们可以期望系统无故障连续运行的时间量。据生产商声称，现代磁盘的平均故障时间在 500 000～1 200 000 小时之间(大约 57～136 年)。事实上，这里声称的平均故障时间是基于全新磁盘发生故障的可能性计算的。这里给出的数据的意思是说，对于给定的 1 000 张相对较新的磁盘，如果 MTTF 是 1 200 000 小时，则平均来说，在 1200 小时内这些磁盘中将会有一张发生故障。平均故障时间为 1 200 000 小时并不意味着该磁盘预期可以工作 136 年！大多数磁盘预期可以工作 5 年左右，但是使用了几年之后，其故障发生率会显著提高。

台式机的磁盘驱动器通常支持每秒 150MB 传输率的串行 ATA(SATA)接口，或者支持每秒 300MB 传输率的 SATA-II 3Gb 接口。PATA5 接口支持每秒 133MB 的传输率。为服务器系统设计的磁盘驱动器大都支持提供最高可达每秒 320MB 传输率的 Ultra320 SCSI 接口，或者是提供每秒 3GB 或者 6GB 传输率的串行附着 SCSI(SAS)接口。通过网络连接到服务器的存储区域网络(SAN)设备通常使用光纤通道 FC－2Gb 或 4－Gb 接口，该接口提供最高可达每秒 256 或 512MB 的传输率。一个接口的传输率由连接到这个接口的所有磁盘所共享。除了某些只允许一张磁盘连接一个接口的串口。

10.2.3 磁盘块访问的优化

磁盘 I/O 请求是由文件系统和大多数操作系统具有的虚拟内存管理器产生的。每个请求指定了要访问的磁盘地址，这个地址是以块号的形式提供的。一个**块**(block)是一个逻辑单元，它包含固定数目的连续扇区。块大小在 512 字节到几 KB 之间。数据在磁盘和主存储器之间以块为单位传输。术语**页**(page)常用来指块，尽管在有些语境(例如闪存)中指的是另外的含义。

来自磁盘的连续请求有可以归类成顺序访问模式或随机访问模式。在一个**顺序访问**(sequential
436 access)模式中，连续的请求会请求处于相同的磁道或是相邻的磁道上连续的块。在顺序访问中读取块时，第一块可能需要一次磁盘寻道，但是相继的请求既不需要寻道，也不需要对相邻磁道的寻道，这比对一条更远处的磁道寻道要快。

与之相反，在**随机访问**(random access)模式中，相继的请求会请求那些随机位于磁盘上的块。每一个请求都需要一次磁盘寻道。一张磁盘在一秒钟内能满足的随机块访问的数量取决于寻道时间，并且通常是每秒 100－200 的访问次数。由于每次寻道只有少量的数据(一块数据)被读取，因此随机

访问模式的数据传输率明显低于顺序访问模式。

为了提高访问块的速度，产生了许多技术。

- **缓冲**(buffering)。从磁盘读取的块暂时存储在内存缓冲区中，以满足将来的要求。缓冲通过操作系统和数据库系统共同运作。关于数据库缓冲将在10.8节进行更详细的讨论。
- **预读**(read-ahead)。当一个磁盘块被访问时，相同磁道的连续块也被读入内存缓冲区中，即使没有针对这些块的即将来临的请求。在顺序访问的情况下，当它们被请求时，这种预读可以确保许多块已经在内存中，减少了磁盘寻道和读取每块的旋转等待时间。操作系统也经常对操作系统文件预先读取连续的块。但是，预读对于随机块访问并不是很有用。
- **调度**(scheduling)。如果需要把一个柱面上的几个块从磁盘传输到主存储器，我们可以按块经过读写头的顺序发出访问块的请求，从而节省访问时间。如果所需的块在不同的柱面上，按照使磁盘臂移动最短距离的顺序发出访问块的请求是非常有利的。**磁盘臂调度**(disk-arm-scheduling)算法试图把对磁道的访问按照能增加可以处理的访问数量的方式排序。通常使用的算法是**电梯算法**(elevator algorithm)，这种算法的工作方式与许多电梯的工作方式非常相似。假设一开始，磁盘臂从最内端的磁道向磁盘的最外端移动。在电梯算法的控制下，对每条有访问请求的磁道，磁盘臂都在那条磁道停下，为这条磁道的请求提供服务，然后继续向外移动，直到没有对更外层磁道的等待请求。这时，磁盘臂掉转方向，开始向内侧移动，同样在每条有请求的磁道处停下，直到没有更靠近中心的磁道上有请求在等待。接着，它掉转方向，开始一个新的周期。磁盘控制器通常对读请求进行重新排序以提高性能，因为它最清楚磁盘块的组织、磁盘盘片的旋转位置和磁盘臂的位置。 437
- **文件组织**(file organization)。为了减少块访问时间，我们可以按照与预期的数据访问方式最接近的方式来组织磁盘上的块。例如，如果我们预计一个文件将顺序地访问，那么理想情况下我们应该使文件的所有块存储在连续的相邻柱面上。老的操作系统，如IBM大型机的操作系统，给程序员提供了对文件位置的精确控制，允许程序员保留一组柱面来存储一个文件。然而，这种控制给程序员或系统管理员带来了决策上的负担，例如要决定给一个文件分配多少柱面，并且当数据插入文件或从文件中删除时，重新组织文件需要昂贵的代价。

  随后的操作系统，如UNIX和Windows操作系统，对用户隐藏了磁盘的组织，并且由操作系统内部来管理空间的分配。尽管它们不能保证一个文件的所有块顺序分布，但它们一次为一个文件分配多个连续的块(一个区(extent))。然后，文件的顺序访问只需每个区寻道一次，而不是每个块寻道一次。经过一段时间，一个连续的文件将变得碎片化(fragmented)，即它的块散布在整张磁盘上。为了减少碎片，系统可以对磁盘上的数据进行一次备份，然后再恢复整张磁盘。恢复操作将每个文件的块连续地(或几乎连续地)写回。一些系统(如Windows操作系统的不同版本)提供扫描整张磁盘然后移动块以减少碎片的工具。这种技术所实现的性能提高非常显著。
- **非易失性写缓冲区**(nonvolatile write buffer)。因为主存储器中的内容在发生电源故障时将全部丢失，所以关于数据库更新的信息必须记录到磁盘上，这样才能在系统崩溃时得以保存。因此，更新操作密集的数据库应用的性能，如事务处理系统的性能，高度依赖于磁盘写操作的速度。

  我们可以使用**非易失性随机访问存储器**(NonVolatile Random-Access Memory，NV-RAM)大幅度地加速写磁盘的操作。NV-RAM的内容在发生电源故障时不会丢失。一种实现NV-RAM的常用方法是使用有备用电池的RAM。其思想是，当数据库系统(或操作系统)请求往磁盘上写一个块时，磁盘控制器将这个块写到NV-RAM缓冲区中，然后立即通知操作系统写操作已成功完成。当磁盘上没有其他请求时，或者当NV-RAM缓冲区满时，磁盘控制器将这些数据写到磁盘上相应的目标地址处。当数据库系统请求写一个块时，只有在NV-RAM缓冲区满时，才会有一个延迟。从系统崩溃中进行恢复时，NV-RAM中所有缓冲的未完成的写操作将写回到磁盘。某些特定的高端磁盘会使用NV-RAM缓冲区，但是NV-RAM更多应用于"RAID

控制器”，我们将在 10.3 节研究 RAID。

- 日志磁盘(log disk)。减少写等待时间的另一种方法是使用日志磁盘，即一种专门用于写顺序日志的磁盘，这和非易失性 RAM 缓冲区的使用非常相似。对日志磁盘的所有访问都是顺序的，这从根本上消除了寻道时间，并且一次可以写几个连续的块，使得写日志磁盘比随机的写要快许多倍。和缓冲一样，数据也必须写到它们在磁盘上的实际位置。但是数据库系统不需要等待这种写操作的完成，日志磁盘会在以后完成写操作。进一步说，日志磁盘可以为了最少化磁盘臂的移动而重排写操作的顺序。如果系统在实际磁盘写操作完成以前崩溃，在系统恢复后，系统可以读取日志磁盘，找到那些还没有完成的写操作并将它们完成。

438

支持上述日志磁盘的文件系统称作**日志文件系统**(journaling file system)。日志文件系统甚至可以在没有单独的日志磁盘的情况下实现，此时它将数据和日志存储在同一张磁盘上。这样做可以节省财务开支，但是会降低系统性能。

大多现代文件系统都实现了日志，并在写入内部文件系统的信息(如文件分配信息)时使用日志磁盘。早期文件系统允许在不使用日志磁盘的情况下对写操作重新排序，一旦系统崩溃，磁盘上的文件系统数据结构将会有被破坏的风险。例如，假定一个使用链表的文件系统，在链表的末端插入一个新结点时，先写入新结点的数据，然后更新来自前面的结点的指针。还假设写操作重新排序，使得指针先更新，而系统在写入新结点之前崩溃，那么结点的内容将会是此前磁盘上的废数据，从而导致数据结构的破坏。

为了处理这种数据结构被破坏的可能性，早期文件系统必须在系统重新启动时执行一个文件系统一致性检验，以保证数据结构是一致的。且如果它们不一致，必须执行额外的步骤还原它们以保证一致性。这些检验会使得系统崩溃后的重启动有较长的延时，而且这个延时随着磁盘系统容量的增加而会得更加严重。日志文件系统允许快速重启动而不需要这样的一致性检验。

但是，由应用程序执行的写操作一般不写入日志磁盘中。数据库系统实现它们自己形式的日志，我们将在第 16 章研究。

10.2.4 快闪存储

正如 10.1 节所提到的，一共有两种快闪存储器，即 NOR 快闪和 NAND 快闪。NOR 快闪允许随机访问闪存中的单个字，并且拥有和主存可媲美的读取速度。而 NAND 快闪和 NOR 快闪不同，它的读取需要将整个数据页从 NAND 快闪取到主存储器中，该数据页通常包括 512～4096 字节。NAND 快闪中的页和磁盘中的扇区十分相似。但是 NAND 快闪明显比 NOR 快闪便宜，并且拥有更高的存储容量，目前更广泛使用。

439

采用 NAND 快闪构建的存储系统提供与磁盘存储器相同的面向块的接口。与磁盘相比，闪存可以提供更快的随机存取：一个数据页，从闪存中可以在约 1～2 微秒内检索到，而从磁盘上的一个随机访问则需要 5～10 毫秒。闪存具有比磁盘低的传输速率，一般来说为每秒 20MB。最近的一些闪存的传输速率增加到每秒 100～200MB。然而，同态驱动器并行地使用多块闪存芯片，将传输速率提高到每秒 200MB 以上，这比大多数磁盘的传输速率更快。

闪存的写入稍有些复杂。写一个闪存页面通常需要几微秒。然而，一旦写入，闪存的页面不能直接覆盖。它必须先擦除然后再重写。一次擦除操作可以在多个页面执行，称为**擦除块**(erase block)。这种操作需时约 1～2 毫秒。一个擦除块的大小(在闪存文献中通常描述为“块”)通常明显比存储系统的块大很多。此外，对一个闪存页可以擦除多少次存在限制，通常大约为 10 万～100 万次。一旦达到此限制，存储位就可能发生错误。

闪存系统通过映射逻辑页码到物理页码，限制了慢擦除速度和更新限制的影响。当一个逻辑页更新时，它可以重新映射到任何已擦除的物理页，它原来的位置可以随后擦除。每个物理页都有一个小的存储区域来保存它的逻辑地址；如果逻辑地址重新映射到一个不同的物理页，则原来的物理页被标记为已删除。因此，通过扫描物理页，我们可以发现每个逻辑页的位置。为了快速访问，逻辑到物理的页面映射被复制到内存的**转换表**(translation table)中。

包含多个删除页面的块将会定期清除，并注意先复制这些块中未删除的页面到不同块（转换表中对这些未删除的页面进行更新）中。由于每个物理页面只能更新固定次数，因此擦除多次的物理页面被标记成"冷数据"，换句话说，数据很少更新；没有擦除多次的页用来存储"热数据"，即，频繁更新的数据。这种在物理块中均匀分布擦除操作的原则称作**损耗均衡**（wear leveling），而且通常是由闪存控制器透明地执行。如果一个物理页由于过多次更新而损坏，它可以不再使用，而不影响整个闪存。

所有的上述动作通过一个叫做**闪存转换层**（flash translation layer）的软件层完成。在这一层之上，闪存存储器看起来和磁盘存储器一样，都是供同样的面向页/扇区的接口。除了闪存存储器快得多。因此，文件系统和数据库存储结构可以看到相同的底层存储结构逻辑视图，无论是闪存存储器或磁盘存储器。

混合硬盘驱动器（hybird disk drive）是结合了小容量闪存存储器的硬盘系统，对频繁访问的数据，该驱动器作为缓存使用。频繁访问但很少更新的数据最适合于缓存在闪存存储器中。 [440]

10.3　RAID

虽然磁盘的存储容量在迅速增长，但是，有些应用（特别是 Web、数据库和多媒体应用）的数据存储需求增长太快，以致在这些应用中需要大量的磁盘来存储数据。

在一个系统中使用大量的磁盘为提高数据读写速率提供了机会，如果并行访问磁盘的话。很多独立的读和写操作也可以并行执行。此外，这种组织方式提供了提高数据存储可靠性的潜力，因为可以在多张磁盘中存储冗余信息。因此一个磁盘的故障不会导致数据的丢失。

为了提高性能和可靠性，人们提出了统称为**独立磁盘冗余阵列**（Redundant Array of Independent Disk，RAID）的多种磁盘组织技术。

在过去，系统设计者认为用多张较小且廉价的磁盘构成的系统替代大而昂贵的磁盘是一种合算的方法；小磁盘系统的每兆字节的价格比大磁盘系统便宜。事实上，RAID 中的 I 现在代表的意思是独立（independent），原先代表的意思是廉价（inexpensive）。然而现在，所有的磁盘物理上都比较小，大容量磁盘每兆字节的价格实际上也降低了。使用 RAID 是因为它有更高的可靠性和更高的执行效率，而不再是因为经济原因。使用 RAID 的另一个关键理由是它易于管理和操作。

10.3.1　通过冗余提高可靠性

让我们首先考虑可靠性问题。N 张磁盘组成的集合中其张磁盘发生故障的概率比特定的一张磁盘发生故障的概率要高。假设一张磁盘发生故障的平均时间为 100 000 小时（稍大于 11 年）。这样，由 100 张磁盘组成的阵列发生磁盘故障的平均时间为 100 000/100 = 1000 小时（约为 42 天），这个时间一点也不长！如果我们只存储了数据的一个拷贝，那么任何一张磁盘发生故障都将导致大量数据的丢失（正如 10.2.1 节讨论的那样）。这样高的数据丢失率是无法接受的。

引入**冗余**（redundancy）是解决这个可靠性问题的方法，即存储正常情况下不需要的额外信息，但这些信息可在发生磁盘故障时用于重建丢失的信息。这样，即使有一张磁盘发生了故障，数据也不会丢失，从而延长了磁盘发生故障的有效平均时间（这里只计导致数据丢失或数据不可用的磁盘故障）。

实现冗余最简单（但最昂贵）的方法是复制每一张磁盘。这种技术称为**镜像**（mirroring）（或者有时称为影子）。这样，一张逻辑磁盘由两张物理磁盘组成，并且每一次写操作都要在两张磁盘上执行。 [441]
如果其中一张磁盘发生了故障，数据可以从另一张磁盘读出。只有当第一张磁盘的故障被修复之前，第二张磁盘也发生了故障时，数据才会丢失。

采用镜像技术的磁盘的平均故障时间（这里的故障是指数据的丢失）依赖于单张磁盘的平均故障时间和**平均修复时间**（mean time to repair）。平均修复时间是替换发生故障的磁盘并且恢复这张磁盘上的数据所花费的（平均）时间。假设两张磁盘发生故障是相互独立的，即一张磁盘的故障和另一张磁盘的故障之间没有联系。那么，如果一张单独磁盘的平均故障时间是 100 000 小时，平均修复时间是 10 小时，则镜像磁盘系统的**平均数据丢失时间**（mean time to data loss）为 $100\,000^2/(2 * 10) =$

$500 * 10^6$ 小时，即 57 000 年！（这里就不再探究推导过程，文献注解中的参考文献里提供了细节。）

读者应该意识到磁盘故障之间的独立性假设是不合理的。电源故障和自然灾害，如地震、火灾和洪水，将导致两张磁盘在同一时间毁坏。当磁盘逐渐老化，发生故障的可能性将随之增加，同时增加了在第一张磁盘被修复时，第二张磁盘也发生故障的机会。尽管有这些方面需要考虑，镜像磁盘系统仍然比单一磁盘系统提供了更高的可靠性。现在已有平均数据丢失时间在 500 000 – 1 000 000 小时（或 55 – 110 年）的镜像磁盘系统。

电源故障是特别需要考虑的问题，因为它们比自然灾害的发生频繁得多。如果在电源发生故障时没有正在向磁盘传输数据，那么电源故障就不必考虑。然而，即使有磁盘镜像，如果正在对两张磁盘上相同的块进行写操作，并且在两个块完全写完之前电源发生故障，那么这两个块将处于不一致的状态。解决这个问题的方法是，先写一个拷贝，再写另一个；这样，两个拷贝中有一个总是一致的。当我们在电源发生故障后重新启动时，需要做一些额外的动作，以从不完全的写操作中恢复。这个问题将在习题 10.3 中讨论。

10.3.2 通过并行提高性能

现在让我们考虑对多张磁盘进行并行访问的好处。通过磁盘镜像，处理读请求的速度将翻倍，因为读请求可以发送到任意一张磁盘上（只要组成一对的两张磁盘都是可用的，而且一般情况下总是如此）。每个读操作的传输速率和单一磁盘系统中的传输速率是一样的，但是每单位时间内读操作的数目将翻倍。

我们可以通过在多张磁盘上进行**数据拆分**（striping data）来提高传输速率。数据拆分最简单的形式是将每个字节按比特分开，存储到多个磁盘上。这种拆分称为**比特级拆分**（bit-level striping）。例如，如果我们有一个由 8 张磁盘组成的阵列，我们将每个字节的第 i 位写到第 i 张磁盘上。这 8 张磁盘组成的阵列可以被看成一张单一的磁盘，这张磁盘的一个扇区的大小是通常大小的 8 倍，并且更重要的是它的传输速率是通常速率的 8 倍。在这样的组织结构中，每张磁盘都参与每次的访问（读或写），
442 所以它每秒所处理的访问操作的总数和一张磁盘所处理的是一样的，但是在相同时间内，它每次访问所读取的数据量是一张磁盘上的 8 倍。位级拆分可以推广到磁盘总数为 8 的倍数或 8 的因子的情况。例如，如果我们使用由 4 张磁盘组成的阵列，每个字节的第 i 位和第 $4 + i$ 位写到第 i 张磁盘上。

块级拆分（block-level striping）是将块拆分到多张磁盘。它把磁盘阵列看成一张单独的大磁盘，并且给块进行逻辑编号，这里我们假设块的逻辑编号从 0 开始。对于 n 张磁盘的阵列，块级拆分将磁盘阵列逻辑上的第 i 个块存储到第 $(i \bmod n) + 1$ 张磁盘上；即用第 $\lfloor i/n \rfloor$ 个物理块存储逻辑块 i。例如，有 8 张磁盘，逻辑磁盘块 0 存储在磁盘 1 的第 0 个物理块上；逻辑磁盘块 11 存储在磁盘 4 的第 1 个物理块上。当读一个大文件时，块级拆分可以同时从 n 张磁盘上并行地读取 n 个块，从而获得对于大的读操作的高数据传输率。当读取单块数据时，其数据传输率与单个磁盘上的数据传输率相同，但是剩下的 $n - 1$ 张磁盘可以自由地进行其他操作。

块级拆分是最常用的数据拆分形式。其他层次的拆分，例如将扇区按字节拆分或者将块按扇区拆分，也是可以实现的。

总之，磁盘系统中的并行有两个主要的目的：

1. 负载平衡多个小的访问操作（块访问），以提高这种访问操作的吞吐量。
2. 并行执行大的访问操作，以减少大访问操作的响应时间。

10.3.3 RAID 级别

镜像提供了高可靠性，但它十分昂贵。拆分提供了高数据传输率，但不能提高可靠性。通过结合"奇偶校验位"（在下文描述）和磁盘拆分思想，从而以较低的代价提供数据冗余，人们已经提出了一些不同的替换方案。这些方案具有不同的成本和性能之间的权衡，并已分为若干 RAID 级别（RAID level）。图 10-3 表示这些级别（在图 10-3 中，P 表示纠错位，C 表示数据的第二个拷贝）。对于所有的级别，该图描绘了 4 张磁盘的数据，而其中描绘的其余磁盘用于存储故障恢复时所使用的的冗余信息。

- **RAID 0 级**(RAID level 0)，指块级拆分但没有任何冗余(例如镜像或奇偶校验位)的磁盘阵列。图 10-3a 给出了一个大小为 4 的磁盘阵列。
- **RAID 1 级**(RAID level 1)，指的是使用块级拆分的磁盘镜像。图 10-3b 给出了一个镜像的组织结构，存储 4 张磁盘的数据。

  注意一些厂商使用 **RAID 1 + 0** 级或 **RAID 10** 级来指代使用拆分的镜像。使用 RAID 1 级指代不使用拆分的镜像。不使用拆分的镜像可以与磁盘阵列一起使用，以呈现为一张单一的大型可靠磁盘：如果每张磁盘有 M 块，逻辑块 $0 \sim M-1$ 存储在磁盘 0 上，逻辑块 $M \sim 2M-1$ 存储在磁盘 1(第二张磁盘)上，依次类推，且每张磁盘都有镜像。[1]
- **RAID 2 级**(RAID level 2)，也称为内存风格的纠错码(Error-Correcting-Code，ECC)组织结构，使用奇偶校验位。长期以来，内存系统使用奇偶校验位来实现错误检测和纠正。内存系统中的每个字节都有一个与之相联系的奇偶校验位，它记录了这个字节中为 1 的位数是偶数(奇偶校验位 = 0)还是奇数(奇偶校验位 = 1)。如果这个字节中有一位被破坏(1 变成 0，或 0 变成 1)，那么这个字节的奇偶校验位就会改变，而与存储的奇偶校验位不匹配。同样地，如果存储的奇偶校验位被破坏，它就不能和计算出的奇偶校验位相匹配。因此，内存系统可以检测到所有的 1 位错误。纠错码机制存储两个或更多的附加位，并且如果有一位被破坏，它可以重建数据。



图 10-3　RAID 级别

通过把字节拆分存储到多张磁盘上，纠错码的思想可以直接用于磁盘阵列。例如，每个字节的第一位存储在磁盘 0 中，第二位存储在磁盘 1 中，如此下去，直到第 8 位存储在磁盘 7 中，纠错位存储在其余的磁盘中。

图 10-3c 显示了 2 级方案。标记为 P 的磁盘存储了纠错位。如果其中一张磁盘发生了故障，可以从其他磁盘中读出字节的其余位和相关的纠错位，并用它们来重建被破坏的数据。图 10-3c 显示了一个大小为 4 的磁盘阵列。注意，对于 4 张磁盘的数据，RAID 2 级方案只需要 3 张磁盘的开销，而不像 RAID 1 级需要 4 张磁盘的开销。

- **RAID 3 级**(RAID level 3)，位交叉的奇偶校验组织结构。RAID 3 级在 RAID 2 级的基础上进行了改进。与内存系统不同，磁盘控制器能够检测一个扇区是否正确地读出，所以可以使用一个单一的奇偶校验位来检错和纠错。它的思想如下：如果一个扇区被破坏，系统能准确地知道是哪个扇区坏了，并且对扇区中的每一位，系统可以通过计算其他磁盘上对应扇区的对应位的奇偶值来推断出该位是 1 还是 0。如果其余位的奇偶校验位等于存储的奇偶校验位，则丢失的位是 0，反之为 1。

---

[1] 注意一些厂商使用术语 RAID 0 + 1 来指代 RAID 的一个版本，该版本同拆分包含一个 RAID 0 阵列，并用另一个阵列作为该阵列的镜像。它与 RAID 1 的区别在于如果一张磁盘发生错误，包含该磁盘的 RAID 0 阵列变为不可用，镜像的阵列仍然可以使用，因此没有数据丢失。在一张磁盘发生错误时，这一定程不如 RAID 1，因为 RAID 0 阵列中的其他磁盘在 RAID 1 中可以继续使用，但是在 RAID 0 + 1 中仍然是无用的。

RAID 3 级和 RAID 2 级一样好，但是在额外磁盘的数目方面更加节省（它只有一张磁盘的开销），所以在实际中并不使用 RAID 2。图 10-3d 表示 RAID 3 级方案。

RAID 3 级与 RAID 1 级相比有两个好处。RAID 3 级对多张常规磁盘只需要一个奇偶校验磁盘，而 RAID 1 级对每张磁盘都需要一张镜像磁盘。因此 RAID 3 级减少了存储的开销；因为使用 N 道数据拆分的 RAID 3 级对一个字节的读写散布在多张磁盘中，所以它使用 N 道数据拆分读写一个块的传输率是 RAID 1 级的 N 倍。但另一方面，因为每张磁盘都要参与每个 I/O 请求，所以 RAID 3 级每秒钟支持的 I/O 操作数较少。

- **RAID 4 级**（RAID level 4），块交叉的奇偶校验组的结构。它像 RAID 0 级一样使用块级拆分，此外在一张独立的磁盘上为其他 N 张磁盘上对应的块保留一个奇偶校验块。图 10-3e 表示了这种方法。如果一张磁盘发生故障，可以使用奇偶校验块和其他磁盘上对应的块来恢复发生故障的磁盘上的块。

443
445

它读取一个块只访问一张磁盘，因此允许其他的请求在其他磁盘上执行。这样，每个访问操作的数据传输率较低，但可以并行地执行多个读操作，从而能产生较高的总 I/O 传输率。由于所有磁盘可以并行地读，因此读取大量数据的操作有很高的传输率。写入大量数据的操作也有很高的传输率，因为数据和奇偶校验位可以并行地写。

但是，小的独立的写操作不能并行地执行。写一个块需要访问存储这个块的磁盘和存储奇偶校验位的磁盘，因为存储奇偶校验位的块需要更新。另外，为了计算新的奇偶校验位，需要读出存储奇偶校验位的块的旧值和要写入的块的旧值。因此，一个单独的写操作需要 4 次磁盘访问：两次读操作以读取两个旧块，两次写操作以写入两个新块。

- **RAID 5 级**（RAID level 5），块交叉的分布奇偶校验位的组织结构。RAID 5 级在 RAID 4 级的基础上进行了改进，将数据和奇偶校验位都分布到所有的 $N+1$ 张磁盘中，而不是在 N 张磁盘上存储数据并在一张磁盘上存储奇偶校验位。在 RAID 5 级中，所有磁盘都能参与对读请求的服务，而不像 RAID 4 级中存储奇偶校验位的磁盘不参与读操作，所以 RAID 5 级增加了在一段给定的时间中能处理的请求总数。对每个由 N 个逻辑磁盘块组成的集合来说，其中一张磁盘存储奇偶校验块，而其余的 N 张磁盘存储逻辑磁盘块。

图 10-3f 表示了这种方法的设置，校验块 P 分布到所有磁盘上。例如，在 5 张磁盘组成的阵列中，逻辑块 $4i$、$4i+1$、$4i+2$、$4i+3$ 对应的奇偶校验块 P_i，存储在第 $i \bmod 5$ 张磁盘中；余下的 4 张磁盘中对应的块存储 $4i \sim 4i+3$ 这四个数据块。下面的表描述了 0～19 这前 20 个块和它们的奇偶校验块在磁盘中是如何排列的。对于以后的块重复这样的存储模式。

P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4

注意奇偶校验块不能和其所对应的数据块存储在同一张磁盘上，如果这样，一张磁盘发生故障会导致数据和奇偶校验位的丢失，数据将是不可恢复的。RAID 5 级包含 RAID 4 级，因为它在相同的成本下提供了更好的读写性能，所以 RAID 4 级在实际中未使用。

- **RAID 6 级**（RAID level 6），P+Q 冗余方案。它和 RAID 5 级非常相似，但是存储了额外的冗余信息，以应对多张磁盘发生故障的情况。RAID 6 不使用奇偶校验的方法，而是使用像 Reed-Solomon 码（参见文献注释）之类的纠错码。如图 10-3g 所示，这种方法与 RAID 5 级对每 4 位数据存储 1 个奇偶校验位不同，它为每 4 位数据存储 2 位的冗余信息，这样系统可容忍两张磁盘发生故障。

446

最后我们要注意的是，对于这里描述的基本 RAID 策略，人们提出了许多变种，不同的厂商采用不同的术语来描述其变种。

10.3.4　RAID 级别的选择

选择 RAID 级别应该考虑以下的因素：

- 所需的额外磁盘存储带来的花费。
- 在 I/O 操作数量方面的性能需求。
- 磁盘故障时的性能。
- 数据重建过程（即，故障磁盘上的数据在新磁盘上重建的过程）中的性能。

重建故障磁盘上数据的时间可能很长，并且时间随着使用的 RAID 级别不同而不同。RAID 1 级的数据重建是最简单的，因为数据可以从另一张磁盘中拷贝得到。对其他级别，我们需要访问磁盘阵列中所有其他磁盘来重建故障磁盘上的数据。如果需要提供连续的可用数据，如高性能数据库所做的那样，那么 RAID 系统的**重建性能**（rebuild performance）将是一个重要因素。此外，因为重建时间占了大部分的恢复时间，所以重建性能也会影响平均数据丢失时间。

RAID 0 级用于数据安全性不是很重要的高性能应用。因为 RAID 2 级和 RAID 4 级被 RAID 3 级和 RAID 5 级所包含，所以 RAID 级别的选择只限于在剩下的级别中进行。比特级拆分（RAID 3 级）不如块级拆分（RAID 5 级），这是因为块级拆分对于大量数据的传输有与 RAID 3 级同样好的数据传输率，同时对于小量数据的传输使用更少的磁盘。对于小量数据传输，磁盘访问时间占主要地位，所以并行读取并没有带来好处。事实上，对于小量数据传输，RAID 3 级可能比 RAID 5 级的性能更差，这是由于只有当所有磁盘上的相关扇区都读出后传输才算完成，因此采用 RAID 3 级的磁盘阵列的平均延迟变得非常接近于单张磁盘在最坏情况下的延迟，从而抵消了其较高传输速率的长处。当前，许多 RAID 的实现并不支持 RAID 6 级，但是 RAID 6 级提供比 RAID 5 级更高的可靠性，可以用于数据安全十分重要的应用。

在 RAID 1 级和 RAID 5 级中做出选择十分困难。由于 RAID 1 级提供最好的写操作的性能，因此在例如数据库系统日志文件的存储这样的应用中使用广泛。RAID 5 级与 RAID 1 级相比具有较低的存储负载，但写操作需要更高的寻问开销。对于经常进行读操作而很少进行写操作的应用，RAID 5 级是首选。

许多年以来，磁盘的存储容量以每年超过 50% 的速度增长，每字节的价格却以相同的速率下降。因而，对许多现存的具有中等存储要求的数据库应用而言，磁盘镜像所需的额外磁盘存储的开销相对变小了（但是，对于像视频数据存储这样的高速率存储的应用而言，额外磁盘存储的开销仍然是一个重要问题）。存取速度的增长相对缓慢（大约每 10 年翻三倍），但是每秒所需的 I/O 操作数飞速增长，特别是对 Web 应用服务器。 447

RAID 5 级增加了写单个逻辑块所需的 I/O 操作数，在写性能方面付出了较长时间的代价。因而，RAID 1 级成为许多具有中等存储需求和高 I/O 需求的应用的 RAID 级别选择。

RAID 系统的设计者还需要做出一些其他决定。例如，一个阵列中应该有多少磁盘？每个奇偶校验位应保护几位数据？如果阵列中的磁盘越多，数据传输率就越高，但是系统将会越昂贵。如果一个奇偶校验位保护的数据位越多，存储奇偶校验位的空间开销就会越小，但是这会增加在第一张发生故障的磁盘被修复前第二张磁盘也发生故障的机会，从而导致数据丢失。

10.3.5　硬件问题

选择 RAID 实现要考虑的另一个因素在硬件层上。RAID 可以在不改变硬件层，只修改软件的基础上实现，这样的 RAID 称为**软件 RAID**（software RAID）。然而，建立支持 RAID 的专用硬件也会带来下面所介绍的很大的好处。具有专用硬件支持的系统称为**硬件 RAID**（hardware RAID）系统。

硬件 RAID 实现能够使用非易失性 RAM 在需要执行的写操作执行之前记录它们。如果发生电源故障，在系统恢复时，它可以从非易失性 RAM 中获得有关未完成的写操作的信息，并完成它们。如果没有这种硬件支持，就需要做一些额外的工作，检测在电源故障前只有部分写入的块（参看实践习题 10.3）。

即使所有的写操作全部完成，也有很小的可能发生磁盘中的某个扇区在某个时刻不可读，即使先

而它已经成功写入了。个别扇区数据损失的原因可能是多种多样的，包括制造缺陷或者一条轨道在相邻轨道被复写入时发生数据损坏。这样丢失的数据是先前成功写入的，有时称为潜在故障(latent failure)或位衰(bit rot)。当发生这样的故障时，如果发现得早，则数据可以通过 RAID 组织中的其他磁盘进行恢复。但是，如果该故障未被发现，并且在其他磁盘的一个扇区存在潜在故障时，那么单个磁盘故障可能导致数据丢失。

为了尽量减少数据丢失的可能，良好的 RAID 控制器会进行擦洗(scrubbing)，也就是说，在磁盘空闲时期，对每张磁盘的每一个扇区进行读取，如果发现某个扇区无法读取，则数据从 RAID 组织的其余磁盘中进行恢复，并写回到扇区中。(如果物理扇区损坏，磁盘控制器将逻辑扇区地址重新映射到
448 磁盘上其他的物理扇区地址。)

一些硬件 RAID 实现允许热交换(hot swapping)，就是在不切断电源的情况下将出错磁盘用新的磁盘替换。由于磁盘的替换不需要等待系统关闭这段时间，因此热交换减少了平均恢复时间。事实上，现在的许多关键系统都以 24×7 的时间表运行，即：一天运行 24 小时，一周运行 7 天。因而没有时间关闭系统和替换故障磁盘。进一步而言，许多 RAID 实现给每一个磁盘阵列(或一个磁盘阵列集)分配一张空闲磁盘。当系统中有磁盘发生了故障，空闲磁盘会立即代替故障磁盘工作。这样可以显著降低平均修复时间，减小数据丢失的机会。可以从容地替换掉故障磁盘。

在 RAID 系统中，电源、磁盘控制器甚或系统互连都可能成为使 RAID 系统停止工作的单个故障发生点。为了避免这种可能性，好的 RAID 实现提供多个备用电源(有电池备份，可以使 RAID 在电源故障后仍能继续工作)。这样的 RAID 系统还拥有多个磁盘接口，和 RAID 系统与计算机系统的多个互连(或者与计算机系统网络的连接)。因而，任何单个部件的故障不会使 RAID 系统停止工作。

10.3.6 其他的 RAID 应用

RAID 的概念已经推广到其他的存储设备上，包括磁带阵列和无线系统上的数据广播。在运用于磁带阵列时，RAID 可以在磁带阵列中的一盘磁带毁坏后恢复磁带上的数据。当运用于数据广播时，数据块分成小单元，连同奇偶校验单元一起传播。如果其中一个单元由于某种原因没有接收到，它可以从其他的单元中重新构造出来。

10.4 第三级存储

在大型数据库系统中，一些数据可能要放置在第三级存储中。两种最常用的第三级存储介质是光盘和磁带。

10.4.1 光盘

在发布软件、多媒体数据(如声音和图像)和其他电子出版物方面，光盘已经成为一种流行的介质。光盘具有 640～700MB 的存储容量。此外，光盘的大批量生产十分便宜。在一些需要大量数据的应用中，数字视频光盘(DVD)正在代替光盘。DVD-5 格式的光盘可以存储 4.7GB 的数据(在一个记录层)，而 DVD-9 格式的光盘可以存储 8.5GB 的数据(在两个记录层)。在光盘的两面都存储就具有更大的容量：如，作为 DVD-5 格式和 DVD-9 格式的双面存储版本，DVD-10 格式和 DVD-18 格式分别可以
449 存储 9.4GB 和 17GB 的数据。蓝光 DVD 格式具有 27～54GB 的更高的单碟存储容量。

由于 CD 和 DVD 驱动器的激光头组件更重，因此比一般的磁盘驱动器需要更长的寻道时间(通常是 100 毫秒)。虽然最快的 CD 和 DVD 驱动器转速和低档的磁盘驱动器转速接近，大约是每分钟 3000 转，但是仍比一般的磁盘驱动器转速慢。CD 驱动器的旋转速度原来是和音频 CD 的标准一致，DVD 驱动器的旋转速度原来是和 DVD 视频标准一致，但是，现在的驱动器旋转速度比标准速度高很多倍。

CD 和 DVD 的数据传输率比磁盘的数据传输率要慢一些。目前的 CD 驱动器的读取速度大约是每秒 3～6MB，而 DVD 是每秒 8～20MB。与磁盘一样，光盘在外侧轨道存储的数据比内侧轨道要多。光盘的数据传输率用 n× 表示，意思是驱动器支持的传输速率是标准速率的 n 倍。现在常用的 CD 是 50 倍速、DVD 是 16 倍速。

因为可记录一次的光盘(CD-R、DVD-R 和 DVD+R)容量大，比磁盘有更长的寿命，而且可以在远程存储和移除，所以适合于数据分发，尤其适合于数据的归档存储。因为它们不能重写，所以它们可用于存储不应更改的信息，比如审计追踪信息。可多次重写的版本(像 CD-RW、DVD-RW、DVD+RW 和 DVD-RAM)也可用于数据归档。

自动光盘机(jukebox)是存储大量的光盘(可以达到几百张)的设备，它可以按照需求自动将光盘装载到少量驱动器(通常是 1～10 个驱动器)中的一个上。这样一个系统的总计存储总量可以是若干个 TB。当光盘被访问时，机械手把它从架子移到驱动器上(在驱动器中的光盘首先要放回到架子上)。光盘的装载和卸载时间通常是几秒的数量级，这比光盘的访问时间要长得多。

10.4.2　磁带

尽管相对而言磁带的保存时间更长久些，并且能够存储大量的数据，但是它与磁盘和光盘相比速度放慢。更重要的是，磁带只能进行顺序存取。因此磁带不能提供辅助存储所需的随机访问，虽然在历史上，磁带是先于磁盘被作为辅助存储介质使用的。

磁带主要用于备份，存储不经常使用的数据，以及作为将数据从一个系统转到另一个系统的脱机介质。磁带还应用于存储大量数据，例如视频和图像数据，它们不需要迅速地访问，或者因为数据量太大以至于磁盘存储太昂贵。

磁带绕在一条轴上，通过卷绕或反卷来经过读写头。移动到磁带上正确的位置可能需要几秒钟甚至几分钟，而不是几毫秒。然而一旦定位完成，磁带设备能够以接近磁盘设备的密度和速度来写数据。 450
磁带的容量取决于磁带的长度、宽度和读写头所能读写的密度。当前市场上的磁带格式千差万别。当前磁带的容量也相差甚远，有几个 GB 的**数字音频磁带**(Digital Audio Tape，DAT)格式，10－40GB 的**数字线性磁带**(Digital Linear Tape，DLT)格式，100GB 或者更高的 **Ultrium** 格式，以及 330GB 的 Ampex **螺旋扫描**(Ampex helical scan)磁带格式。数据传输率在每秒几到几十 MB 的数量级。

磁带设备是相当可靠的，好的磁带驱动系统对刚写的数据执行一次读操作以确保数据正确记录。但是磁带能可靠地读写的次数是有限的。

自动磁带机(tape jukebox)，像自动光盘机一样，存放大量的磁带，并有少量可用于安装磁带的驱动器。它用于存储大量数据，存储范围最大可达若干 PB(10^{15}字节)，而存取时间在几秒到几分钟的数量级。需要如此巨大的数据存储的应用包括通过遥感卫星搜集数据的影像系统和用于电视广播的大型视频库。

一些磁带格式(如 Accelis 格式)支持更快的寻道时间(是几十秒的数量级)，是为那些从自动磁带机获取数据的应用设计的。许多其他的磁带格式提供更大的容量，但是以更慢的存取速度为代价。这样的格式对于不需要快速访问的数据备份是很理想的。

磁带驱动器已经无法跟上磁盘驱动器容量的巨大增长和对应的存储代价的降低。尽管磁带的成本很低，但磁带驱动器和磁带库的成本明显高于一张磁盘的成本：一个能存储几 TB 数据的磁带库可能价值几万美元。对于大量应用而言，较之磁带备份，备份数据到磁盘驱动器已经成为一种划算的选择。

10.5　文件组织

一个数据库被映射到多个不同的文件，这些文件由底层的操作系统来维护。这些文件永久地存在于磁盘上。一个**文件**(file)在逻辑上组织成为记录的一个序列。这些记录映射到磁盘块上。因为文件由操作系统作为一种基本结构提供，所以我们将假定作为基础的文件系统是存在的。我们需要考虑用文件表示逻辑数据模型的不同方式。

每个文件分成定长的存储单元，称为**块**(block)。块是存储分配和数据传输的基本单元。大多数数据库默认使用 4～8KB 的块大小，但是当创建数据库实例时，许多数据库允许指定块大小。更大的块在一些数据库应用中是很有用的。

一个块可能包括很多条记录。一个块所包含的确切的记录集合是由使用的物理数据组织形式所
[451] 决定的。我们假定没有记录比块更大。这个假定对多数数据处理应用都是现实的，例如我们所举的大学例子。当然，数据库中会有几种大数据项，例如图片，可能比一个块要大很多。10.5.2 节会简要地讨论如何通过单独地存储大数据项，并且存储指向记录中大数据项的指针来处理类似的大数据项。

此外，我们需要要求每条记录包含在单个块中，换句话说，没有记录是部分包含在一个块中，部分包含在另一个块中。这个限制简化并加速数据项访问。

在关系数据库中，不同关系的元组通常具有不同的大小。把数据库映射到文件的一种方法是使用多个文件，在任意一个文件中只存储一个固定长度的记录。另一种选择是构造自己的文件，使之能够容纳多种长度的记录。然而，定长记录文件比变长记录文件更容易实现。很多用于定长记录文件的技术可以应用到变长的情况。因此，我们首先考虑定长记录文件，并且稍后考虑变长记录存储。

10.5.1 定长记录

例如，让我们考虑由大学数据库中的 *instructor* 记录组成的一个文件。文件中的每个记录定义（使用伪代码）如下：

```
type instructor = record
    ID varchar(5);
    name varchar(20);
    dept_name varchar(20);
    salary numeric(8, 2);
end
```

假设每个字符占 1 个字节，numeric(8, 2) 占 8 个字节。假设我们为每个属性 *ID*、*name* 和 *dept_name* 分配可以容纳的最大字节数，而不是分配可变的字节数。于是 *instructor* 记录占 53 个字节。一个简单的方法是使用前 53 个字节来存储第一条记录，接下来的 53 个字节存储第二条记录，以此类推（如图 10-4 所示）。然而这种简单的方法有两个问题：

记录0	10101	Srinivasan	Comp. Sci.	65000
记录1	12121	Wu	Finance	90000
记录2	15151	Mozart	Music	40000
记录3	22222	Einstein	Physics	95000
记录4	32343	El Said	History	60000
记录5	33456	Gold	Physics	87000
记录6	45565	Katz	Comp. Sci.	75000
记录7	58583	Califieri	History	62000
记录8	76543	Singh	Finance	80000
记录9	76766	Crick	Biology	72000
记录10	83821	Brandt	Comp. Sci.	92000
记录11	98345	Kim	Elec. Eng.	80000

图 10-4 包含 account 记录的文件

- 除非块的大小恰好是 53 的倍数（一般是不太可能的），否则一些记录会跨过块的边界。即一条记录的一部分存储在一个块中，而另一部分存储在另一个块中。于是，读写这样一条记录需要两次块访问。
- 从这个结构中删除一条记录十分困难。删除的记录所占据的空间必须由文件中的其他记录来填充，或者我们必须用一种方法标记删除的记录使得它可以被忽略。

为了避免第一个问题，我们在一个块中只分配它能完整容纳下的最大的记录数（这个数字可以很容易通过块大小除以记录大小计算出来，并废弃小数部分）。每个块中余下的字节就不使用了。

当一条记录被删除时，我们可以把紧跟其后的记录移动到被删记录先前占据的空间，依次类推，直到被删记录后面的每一条记录都向前做了移动（见图 10-5）。这种方法需要移动大量的记录。简单地将文件的最后一条记录移到被删记录所占据空间中可能更容易一些（如图 10-6 所示）。

记录0	10101	Srinivasan	Comp. Sci.	65000
记录1	12121	Wu	Finance	90000
记录2	15151	Mozart	Music	40000
记录4	32343	El Said	History	60000
记录5	33456	Gold	Physics	87000
记录6	45565	Katz	Comp. Sci.	75000
记录7	58583	Califieri	History	62000
记录8	76543	Singh	Finance	80000
记录9	76766	Crick	Biology	72000
记录10	83821	Brandt	Comp. Sci.	92000
记录11	98345	Kim	Elec. Eng.	80000

图 10-5 图 10-4 中的文件，删除了记录 3 并且移动所有记录

移动记录以占据被删记录所释放空间的做法并

不理想，因为这样做需要额外的块访问操作。由于插入操作通常比删除操作更频繁，因此让已被删记录占据的空间空着、一直等到随后进行的插入操作重用这个空间，这样做是可以接受的。仅在被删记录上做一个标记是不够的，因为当插入操作执行时，找到这个可用空间十分困难。因此我们需要引入额外的结构。

452 ↓ 453

记录0	10101	Srinivasan	Comp. Sci.	65000
记录1	12121	Wu	Finance	90000
记录2	15151	Mozart	Music	40000
记录11	98345	Kim	Elec. Eng.	80000
记录4	32343	El Said	History	60000
记录5	33456	Gold	Physics	87000
记录6	45565	Katz	Comp. Sci.	75000
记录7	58583	Califieri	History	62000
记录8	76543	Singh	Finance	80000
记录9	76766	Crick	Biology	72000
记录10	83821	Brandt	Comp. Sci.	92000

图 10-6　图 10-5 中的文件，删除了记录 3 并且移动最后一条记录

在文件的开始处，我们分配一定数量的字节作为**文件头**(file header)。文件头将包含有关文件的各种信息。到目前为止，我们需要在文件头中存储的只有内容被删除的第一个记录的地址。我们用这第一个记录来存储第二个可用记录的地址，依次类推。我们可以直观地把这些存储的地址看作指针，因为它们指向一个记录的位置。于是，被删除的记录形成了一条链表，经常称为**空闲列表**(free list)。图 10-7 给出的是图 10-4 中的文件在删除第 1、4 和 6 条记录后的情况。

在插入一条新记录时，我们使用文件头所指向的记录，并改变文件头的指针以指向下一个可用记录。如果没有可用的空间，我们就把这条新记录添加到文件末尾。

对定长记录文件的插入和删除是容易实现的，因为被删除记录留出的可用空间恰好是新入记录所需要的空间。如果我们允许文件中包含不同长度的记录，这样的匹配将不再成立。插入的记录可能无法放入一条被删除记录所释放的空间中，或者只能占用这个空间的一部分。

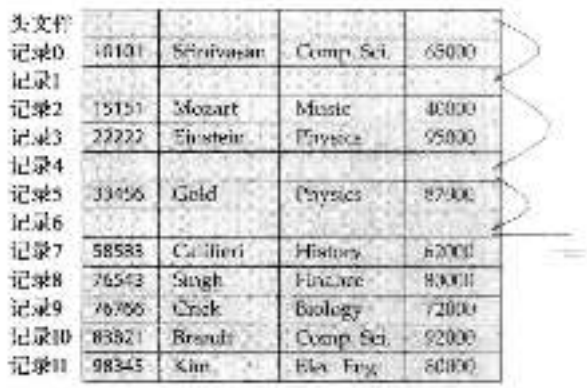


图 10-7　删除了第 1、4 和 6 条记录的图 10-4 中的文件

10.5.2　变长记录

变长记录以下面几种方式出现在数据库系统中：

- 多种记录类型存储在一个文件中存储。
- 允许一个或多个字段是变长的记录类型。
- 允许可重复字段的记录类型，例如数组或多重集合。

实现变长记录存在不同的技术，任何这样的技术都必须解决两个不同的问题：

- 如何描述一条记录，使得单个属性可以轻松地抽取
- 在块中如何存储变长记录，使得块中的记录可以轻松地抽取。

一条有变长度属性的记录表示通常具有两个部分：初始部分是定长属性，接下来是变长属性。对于定长属性，如数字值、日期或定长字符串，分配存储它们的值所需的字节数。对于变长属性，如 varchar 类型，在记录的初始部分中表示为一个对(偏移量，长度)值，其中偏移量表示在记录中该属性的数据开始的位置，长度表示变长属性的字节长度。在记录的初始定长部分之后，这些属性的值是连续存储的。因此，无论是定长还是变长，记录初始部分存储有关每个属性的固定长度的信息。

图 10-8 描述这样一个记录表示的例子。该图显示了一个 *instructor* 记录，它的前三个属性 *ID*、*name* 和 *dept_name* 是变长字符串，其第 4 个属性 *salary* 是一个人小固定的数值。我们假设偏移量和长度值存储在两个字节中，即每个属性占 4 个字节。*salary* 属性假设用 8 个字节存储，并且每个字符串占用和其拥有字符数一样多的字节数。

454 ↓ 455

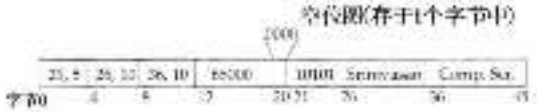


图 10-8 变长记录的表示

这个图也说明了**空位图**(null bitmap)的使用，它用来表示记录的哪个属性是空值。在这个特定的记录中，如果 *salary* 是空值，该位图的第 4 位将置 1，存储在 12 – 19 字节的 *salary* 值将被忽略。由于记录有 4 个属性，尽管更多属性需要更多字节，但该记录的空位图只占用 1 个字节。在一些表示中，空位图存储在记录的开头，并且对于空属性不存储数据(值或偏移量/长度)。这种表示以抽取记录属性的额外工作为代价来节省存储空间。对于记录拥有大量字段，并且大多数都是空的特定应用，这样的表示特别有用。

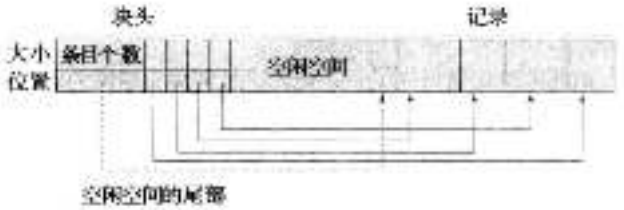


图 10-9 分槽的页结构

我们接下来处理在块中存储变长记录的问题。**分槽的页结构**(slotted-page structure)一般用于在块中组织记录，如图 10-9 所示。每个块的开始处有一个块头，其中包含以下信息：

1. 块头中记录条目的个数。
2. 块中空闲空间的末尾处。
3. 一个由包含记录位置和大小的记录条目组成的数组。

实际记录从块的尾部开始连续排列。块中空闲空间是连续的，在块头数组的最后一个条目和第一条记录之间。如果插入一条记录，在空闲空间的尾部给这条记录分配空间，并且将包含这条记录大小和位置的条目添加到块头中。

如果一条记录被删除，它所占用的空间被释放，并且它的条目被设置成被删除状态(例如这条记录的大小被设置为 –1)。此外，块中在被删除记录之前的记录将被移动，使得由删除而产生的空闲空间被重用，并且所有空闲空间仍然存在于块头数组的最后一个条目和第一条记录之间。块头中的空闲空间末尾指针也要做适当修改。只要块中有空间，使用类似的技术可以使记录增长或缩短。移动记录的代价并不高，因为块的大小是有限制的；典型的值为 4 ~ 8KB。

分槽的页结构要求没有指针直接指向记录。取而代之，指针必须指向块头中记有记录实际位置的条目。在支持指向记录的间接指针的同时，这种间接层次允许移动记录以防止在块的内部出现碎片空间。

数据库常常要存储比磁盘块大得多的数据，例如，一张图片或一个音频记录的大小可能是几 MB，而一个视频文件可能达到几个 GB。回忆一下我们以前讲过 SQL 支持 blob 和 clob 类型，它们存储二进制大对象和字符大对象。

大多数关系数据库限制记录不大于一个块的大小以简化缓冲区管理和空闲空间管理。大对象常常存储到一个特殊文件(或文件的集合)中而不是与记录的其他(短)属性存储在一起。然后一个指向该对象的(逻辑)指针存储到包含该大对象的记录中。大对象常常表示我们在 11.4.1 节要学习的 B⁺ 树文件组织。B⁺ 树文件组织允许我们读取一个完整的对象，或对象中指定的字节范围，以及插入和删除对象的部分。

---

† 在这儿，“页”和“块”同义。

10.6　文件中记录的组织

迄今为止，我们研究了如何在一个文件结构中表示记录。关系是记录的集合。给定一个记录的集合之后，下一个问题就是如何在文件中组织它们。下面是在文件中组织记录的几种可能的方法：

- **堆文件组织**(heap file organization)。一条记录可以放在文件中的任何地方，只要那个地方有空间存放这条记录。记录是没有顺序的。通常每个关系使用一个单独的文件。
- **顺序文件组织**(sequential file organization)。记录根据其"搜索码"的值顺序存储。10.6.1 节描述了这种组织方式。
- **散列文件组织**(hashing file organization)。在每条记录的某些属性上计算一个散列函数。散列函数的结果确定了记录应放到文件的哪个块中。第 11 章描述这种组织方式。它与那一章所描述的索引结构密切相关。

456
457

通常，每个关系的记录用一个单独的文件存储。但是在**多表聚簇文件组织**(multitable clustering file organization)中，几个不同关系的记录存储在同一个文件中。而且，不同关系的相关记录存储在相同的块中，于是一个 I/O 操作可以从所有关系中取到相关的记录。例如，两个关系做连接运算时相匹配的记录被认为是相关的。10.6.2 节描述这种组织方式。

10.6.1　顺序文件组织

顺序文件(sequential file)是为了高效处理按某个搜索码的顺序排序的记录而设计的。**搜索码**(search key)是任何一个属性或者属性的集合。它没有必要是主码，甚至也无须是超码。为了快速地按搜索码的顺序获取记录，我们通过指针把记录链接起来。每条记录的指针指向按搜索码顺序排列的下一条记录。此外，为了减少顺序文件处理中的块访问数，我们在物理上按搜索码顺序或者尽可能地接近按搜索码顺序存储记录。

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

图 10-10　*instructor* 记录组成的顺序文件

图 10-10 表示了在大学例子中由 *instructor* 记录组成的顺序文件。在该例子中，记录按搜索码顺序存储，这里使用 *ID* 作为搜索码。

顺序文件组织形式允许记录按排序的顺序读取；这对于显示以及第 12 章将研究的特定查询处理算法非常有用。

然而，在插入和删除记录时维护记录的物理顺序是十分困难的，因为一次单独的插入或删除操作导致移动很多记录是代价很高的。我们可以按照前面看到的那样，使用指针链表来管理删除。对插入操作，应用如下规则：

458

1. 在文件中定位按搜索码顺序处于待插入记录之前的那条记录。

2. 如果这条记录所在块中有一条空闲记录(即删除后留下来的空间)，就在这里插入新的记录。否则，将新记录插入到一个溢出块中。不管哪种情况，都要调整指针，使其能按搜索码顺序把记录链接在一起。

图 10-11 表示图 10-10 所示文件在插入记录(32222，Verdi，Music，48000)之后的情况。图 10-11 中的结构允许快速插入新的记录，但是迫使顺序处理文件的应用程序不得不按与记录的物理顺序不一样的顺序来处理记录。

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
32222	Verdi	Music	48000

图 10-11　执行插入后的顺序文件

如果需要存储在溢出块中的记录相当少，

这种方法会工作得很好。然而，搜索码顺序和物理顺序之间的一致性可能会完全丢失，在这种情况下，顺序处理将变得效率十分低下。此时，文件应该重组(reorganized)，使得它再一次在物理上顺序存放。这种重组的代价是很高的，并且必须在系统负载很低的时候执行。需要重组的频率依赖于新记录插入的频率。在插入很少发生的极端情况下，使文件在物理上总保持排序的顺序是可能的。在这种情况下，
[259] 图 10-10 中的指针就是不需要的。

10.6.2 多表聚簇文件组织

很多关系数据库系统将每个关系存储在单独的文件中，以便它们可以利用操作系统所提供的文件系统的所有好处。通常，关系的元组可以表示成定长记录。因此，关系可以映射到一个简单的文件结构上。关系数据库系统的这种简单实现非常适合于廉价的数据库实现，例如，嵌入式系统和便携式设备中的数据库实现。在这种系统中，数据库的规模很小，因此复杂的文件结构不会带来什么好处。而且，在这样的环境中，必须使数据库系统目标代码总量非常小。简单的文件结构可以减少实现这个系统的代码量。

这种简单的关系数据库实现在数据库的规模增大时就不令人满意了。我们已经看到，仔细地设计记录在块中的分配和块自身的组织方式可以获得性能上的好处。显而易见，一个更复杂的文件结构将更有效，即使我们保留在一个独立的文件中存储一个关系的策略。

然而，很多大型数据库系统在文件管理方面并不直接依赖于下层的操作系统，而是让操作系统分配给数据库系统一个大的操作系统文件。数据库系统把所有关系存储在这个文件中，并且自己管理这个文件。

即使一个文件中有多个关系，默认情况下，多数数据库在一个给定块中只存储一个关系的记录。这简化了数据管理。但是，在某些情况下，在一个块中存储多个关系的记录会很有用。为了理解在一个块中存储多个关系的记录的好处，我们考虑对大学数据库的如下 SQL 查询：

```
select dept_name, building, budget, ID, name, salary
from department natural join instructor
```

这个查询计算 *department* 关系和 *instructor* 关系的连接。因此，对 *department* 的每个元组，系统必须找到具有相同 *dept_name* 的 *instructor* 元组。理想情况下，这些记录通过索引的帮助来定位，这将在第 11 章讨论。然而，不管这些记录如何定位，它们都需要从磁盘上传输到主存储器中。在最坏的情况下，每个记录将处于不同的块中，这使我们为查询所需的每条记录执行一次读块操作。

作为一个具体的例子，考虑分别在图 10-12 和图 10-13 中的 *department* 和 *instructor* 关系(简化起见，我们只包括了前面所用到的关系中元组的一个子集)。在图 10-14 中，我们给出了一个为高效执行涉及 *department* 自然连接 *instructor* 的查询而设计的文件结构。每个系的 *instructor* 元组存储在具有相应 *dept_name* 的 *department* 元组附近。这种结构将两个关系的元组混合在一起，而允许对连接的高效处理。当读取 *department* 关系的一个元组时，包含这个元组的整个块从磁盘拷贝到主存储器中。因为相应的 *instructor* 元组存储在靠近 *department* 元组的磁盘上，所以包含 *department* 元组的块包含处理查询所需的 *instructor* 关系的元组。如果一个系有太多的教师，以至于 *instructor* 记录不能存储在一个块中，则其余的记录出现在临近的块中。

dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Physics	Watson	70000

图 10-12 *department* 关系

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

图 10-13 *instructor* 关系

多表聚簇文件组织(multitable clustering file organization)是一种在每一块中存储两个或者更多个关系的相关记录的文件结构，如图 10-14 所示。这样的文件组织允许我们使用一次块的读操作来读取满足连接条件的记录。因此，我们可以更高效地处理这种特殊的查询。

在图 10-14 所示的例子中，来自 *instructor* 记录的 *dept_name* 属性被省略掉，这是因为它可以从相关

department 记录中指明出来，该属性可能会在一些实现中保留，以简化属性的访问。我们假定每条记录包含它所属关系的标识符，尽管这没有在图 10-14 中表示出来。

我们对将多个表聚集到一个文件中的使用加速了对特定连接（*department* 和 *instructor* 的连接）的处理，但是它导致其他类型查询的处理变慢。例如：

```
select *
from department
```

与在单一文件中存储单一关系的策略相比，这个查询需要访问更多的块，因为每个块现在明显包含更少的 *department* 记录。为了在图 10-14 的结构中找到 *department* 关系的所有元组，我们可以用指针把这个关系的所有记录链接起来，如图 10-15 所示。

何时使用多表聚簇依赖于数据库设计者所认为的最频繁的查询类型。多表聚簇的谨慎使用可以在查询处理中产生明显的性能提高。

Comp. Sci.	Taylor	100000
45564	Katz	75000
10101	Srinivasan	65000
83821	Brandt	92000
Physics	Watson	70000
33456	Gold	87000

图 10-14　多表聚簇文件结构

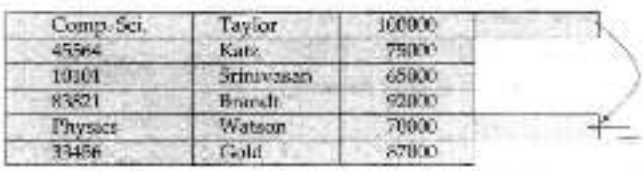


图 10-15　带指针链的多表聚簇文件结构

10.7　数据字典存储

到目前为止，我们只考虑了关系本身的表示。一个关系数据库系统需要维护关于关系的数据，如关系的模式等。一般来说，这样的"关于数据的数据"称为**元数据**（metadata）。

关于关系的关系模式和其他元数据存储在称为**数据字典**（data dictionary）或**系统目录**（system catalog）的结构中。系统必须存储的信息类型有：

- 关系的名字。
- 每个关系中属性的名字。
- 属性的域和长度。
- 在数据库上定义的视图的名字和这些视图的定义。
- 完整性约束（例如，码约束）。

此外，很多系统为系统的用户保存了下列数据：

- 授权用户的名字。
- 关于用户的授权和账户信息。
- 用于认证用户的密码或其他信息。

数据库可能还会存储关于关系的统计数据和描述数据。例如：

- 每个关系中元组的总数。
- 每个关系所使用的存储方法（例如，聚簇或非聚簇）。

数据字典也会记录关系的存储组织（顺序、散列或堆），和每个关系的存储位置：

- 如果关系存储在操作系统文件中，数据字典将会记录包含每个关系的文件名。
- 如果数据库把所有关系存储在一个文件中，数据字典可能将包含每个关系中记录的块记在如链表这样的数据结构中。

460
|
462

第 11 章研究索引，在那一章中，我们将看到有必要存储关于每个关系的每个索引的信息：

- 索引的名字。
- 被索引的关系的名字。
- 在其上定义索引的属性。
- 构造的索引的类型。

实际上，所有这些元数据信息组成了一个微型数据库。一些数据库系统使用专用的数据结构和代码来存储这些信息。通常人们更倾向于在数据库中存储关于数据库本身的数据。通过使用数据库来存储系统数据，我们简化了系统的总体结构，并且允许使用数据库的全部能力来对系统数据进行快速访问。

关于如何使用关系来表示系统元数据的确切选择必须由系统设计者来决定。图 10-16 是一种可能的表示，加下划线的为主码。在这种表达方式中，关系 *Index_metadata* 中的属性 *index_attributes* 假定包含由一个或多个属性组成的列表，这些属性可以表示成形如"*dept_name*, *building*"的字符串。因此，*Index_metadata* 不符合第一范式。尽管它可以规范化，但是上面的表示可能对于存取数据而言会更加有效。数据字典通常存储成非规范化的形式，以便进行快速的存取。

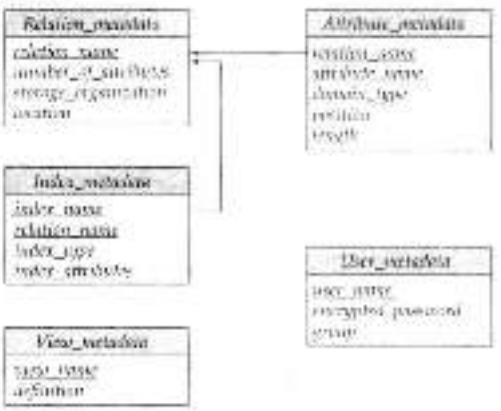


图 10-16 描述系统元数据的关系模式

当数据库系统从关系中查找记录时，它必须首先通过 *Relation_metadata* 关系来查找关系的位置和存储组织，然后通过该信息取回记录。但是，*Relation_metadata* 关系自身的存储组织和位置必须记录在其他地方（例如，在数据库自身的代码段中，或者数据库中的一个固定位置），这是因为我们需要使用这些信息找到 *Relation_metadata* 的内容。

10.8 数据库缓冲区

数据库系统的一个主要目标就是尽量减少磁盘和存储器之间传输的块数目。减少磁盘访问次数的一种方法是在主存储器中保留尽可能多的块。这样做的目标是最大化要访问的块已经在主存储器中的几率，这样就不再需要访问磁盘。

因为在主存储器中保留所有的块是不可能的，所以需要管理主存储器中用于存储块的可用空间的分配。**缓冲区**（buffer）是主存储器中用于存储磁盘块的拷贝的那一部分。每个块总有一个拷贝存放在磁盘上，但是在磁盘上的拷贝可能比在缓冲区中的拷贝旧。负责缓冲区空间分配的子系统称为**缓冲区管理器**（buffer manager）。

10.8.1 缓冲区管理器

463
↓
464

当数据库系统中的程序需要磁盘上的块时，它向缓冲区管理器发出请求（即调用）。如果这个块已经在缓冲区中，缓冲区管理器将这个块在主存储器中的地址传给请求者。如果这个块不在缓冲区中，缓冲区管理器首先在缓冲区中为这个块分配空间，如果需要的话，会把其他块移出主存储器，为这个新块腾出空间。移出的块仅当它自从最近一次写回磁盘后被修改过才被写回磁盘。然后缓冲区管理器把请求的块从磁盘读入缓冲区，并将这个块在主存储器中的地址传给请求者。缓冲区管理器的内部动作对发出磁盘块请求的程序是透明的。

如果你熟悉操作系统的概念，你会发现缓冲区管理器几乎和大多数操作系统中的虚拟存储管理器是一样的。它们的一点区别是数据库的大小会比机器的硬件地址空间大得多，因此存储器地址不足以对所有磁盘块进行寻址。此外，为了更好地为数据库系统服务，缓冲区管理器必须使用比典型的虚拟存储器管理策略更加复杂的技术：

- **缓冲区替换策略**（buffer replacement strategy）。当缓冲区中没有剩余空间时，在新块读入缓冲区之前，必须把一个块从缓冲区中移出。多数操作系统使用最近最少使用（Least Recently Used，

LRU）策略，即最近访问最少的块被写回磁盘，并从缓冲区中移走。这种简单的方法可以加以改进以用于数据库应用。

- **被钉住的块**（pinned block）。为了使数据库系统能够从系统崩溃中恢复（见第 16 章），限制一个块写回磁盘的时间是十分必要的。例如：当一个块上的更新操作正在进行时，大多数恢复系统不允许将该块写回磁盘。不允许写回磁盘的块称为**被钉住的**（pinned）块。尽管很多操作系统不提供对被钉住的块的支持，但是这个特性对可从崩溃中恢复的数据库系统十分重要。
- **块的强制写出**（forced output of block）。在某些情况下，尽管不需要一个块所占用的缓冲区空间，但必须把这个块写回磁盘。这样的写操作称为块的**强制写出**（forced output）。我们将在第 16 章看到需要强制写出的原因：简单地说，主存储器的内容，包括缓冲区的内容，在崩溃时将丢失，而磁盘上的数据一般在崩溃时得以保留。

10.8.2　缓冲区替换策略

对缓冲区中的块替换策略而言，目标是减少对磁盘的访问。对通用程序来说，精确预言哪个块将被访问是不可能的。因此，操作系统使用过去的块访问模式来预言未来的访问。通常我们假定最近访问过的块最有可能再一次被访问。因此，如果必须替换一个块，则替换最近访问最少的块。这种方法称为**最近最少使用**（Least Recently Used，LRU）**块替换策略**。

在操作系统中，LRU 是一个可以接受的替换策略。然而，数据库系统能够比操作系统更准确地预测未来的访问模式。用户对数据库系统的请求包括若干步，通过查看执行用户请求所需操作的每一步，数据库系统通常可以预先确定哪些块将是需要的。因此，与依赖过去而预测将来的操作系统不同，数据库系统至少可以得到短期内的将来信息。

为了说明关于未来块访问的信息是如何使我们改进 LRU 策略的，考虑如下 SQL 查询处理：

```
select *
from instructor natural join department
```

假设所选择的处理这个请求的策略由图 10-17 所示的伪码程序给出。（第 12 章将研究其他更有效的策略。）

```
for each instructor 中的元组 i do
    for each department 中的元组 d do
        if i[dept_name] = d[dept_name]
        then begin
            令 x 为下列式子定义的元组：
            x[ID] := i[ID]
            x[dept_name] := i[dept_name]
            x[name] := i[name]
            x[building] := d[building]
            x[budget] := d[budget]
            将元组 x 包含进 instructor ⋈ department 的结果中
        end
    end
end
```

图 10-17　计算连接的过程

假设该例子中的两个关系存储在不同的文件中。在这个例子中，我们可以看到，一旦 *instructor* 中的一个元组处理过，这个元组就不再需要了。因此，一旦处理完 *instructor* 元组构成的一个完整的块，这个块就不再需要存储在主存储器中了，尽管它刚刚被用过。一旦 *instructor* 块中最后一个元组处理完毕，就应该命令缓冲区管理器释放这个块所占用的空间。这个缓冲区管理策略称为立即丢弃（toss-immediate）策略。

现在考虑包含 *department* 元组的块。对 *instructor* 关系中的每个元组，我们需要检查 *department* 元组的每个块。当一个 *department* 块处理完毕后，我们知道这个块要到所有其他 *department* 块处理完才会被再次访问。因此，最近最常使用的 *department* 块将是最后一个要再次访问的块，最近最少使用的

465
466

department 块是接着要访问的块。这个假设正好与构成 LRU 策略基础的假设相反。实际上，上述过程中块替换的最优策略是**最近最常使用**(Most Recently Used，MRU)策略。如果必须从缓冲区中移除一个 *department* 块，MRU 策略将选择最近最常使用的块(当块被使用时不能被替换)。

在这个例子中，要使 MRU 策略正确地工作，系统必须把当前正在处理的 *department* 块钉住。在块中最后一个 *department* 元组处理完毕后，这个块就不再被钉住，成为最近最常使用的块。

除了使用系统所拥有的关于被处理的请求的知识，缓冲区管理器也可以使用有关一个请求将访问某个特定关系的概率的统计信息。例如，用来记录关系的逻辑模式和它们的物理存储信息的数据字典(在 10.7 节中进行了详细介绍)，是数据库中最经常访问的部分之一。因此，缓冲区管理器应尽量不把数据字典从主存储器中移除，除非有其他因素决定了非这样做不可。第 11 章将讨论文件的索引。因为对文件索引的访问可能比文件本身更频繁，所以不是迫不得已，缓冲区管理器一般不应把索引块从主存储器中移除。

理想的数据库块替换策略需要数据库操作的知识(包括正在执行的操作和将来会执行的操作)。没有哪个策略能够很好地处理所有可能的情况。实际上，绝大多数数据库系统都使用 LRU 策略，尽管这种策略有其缺点。在习题中我们将探究其他可选择的策略。

除了块被再次访问的时间以外，缓冲区管理器使用的块替换策略还受其他因素所影响。如果系统并发地处理多个用户的请求，并发控制子系统(见第 15 章)可能会延迟某些请求，以保证数据库的一致性。如果缓冲区管理器从并发控制子系统获得了关于哪些请求被延迟的信息，它就可以使用这些信息来改变它的块替换策略。具体地说，活动的(非延迟的)请求所需要的块可以因为被延迟的请求所需要的块换出而在缓冲区中得以保留。

崩溃－恢复子系统(见第 16 章)对块替换施加了严格的约束。如果一个块被改了，不允许缓冲区管理器将这个块在缓冲区中的新内容写回磁盘，因为这将破坏块的旧内容。取而代之的做法是，块管理器在写回块之前，必须从崩溃－恢复子系统处寻求许可。崩溃－恢复子系统在允许缓冲区管理器写回
467 需要的块之前，可能要求将某些其他块强制写出。在第 16 章中，我们将精确定义缓冲区管理器与崩溃－恢复子系统之间的交互。

10.9 总结

- 大多数计算机系统中存在多种数据存储类型。我们可以根据访问数据的速度、购买介质时每单位数据的成本和介质的可靠性，对这些存储介质进行分类。可用的介质有高速缓冲存储器、主存储器、快闪存储器、磁盘、光盘和磁带。
- 存储介质的可靠性由两个因素决定：电源故障或系统崩溃是否导致数据丢失，存储设备发生物理故障的可能性有多大。
- 通过保留数据的多个拷贝，我们可以减少物理故障的可能性。对磁盘来说可以使用镜像技术。或者可以使用更复杂的基于独立磁盘冗余阵列(RAID)的方法。通过把数据拆分到多张磁盘上，可以提高大数据量访问的吞吐率；通过引入多张磁盘上的冗余存储，可以显著提高可靠性。有几种不同的 RAID 组织形式，它们具有不同的成本、性能和可靠性特征。最常用的是 RAID 1 级(镜像)和 RAID 5 级。
- 我们可以把一个文件从逻辑上组织成映射到磁盘块上的一个记录序列。把数据库映射到文件的一种方法是使用多个文件，每个文件只存储固定长度的记录。另一种方法是构造文件，使之能适应多种长度的记录。分槽的页方法广泛应用于在磁盘块中处理变长记录。
- 因为数据以块为单位在磁盘存储器和主存储器之间传输，所以采取用一个单独的块包含相关联的记录的方式，将文件记录分配到不同的块中是可取的。如果我们能够仅使用一次块访问就可以存取我们想要的多个记录，就能节省磁盘访问次数。因为磁盘访问通常是数据库系统性能的瓶颈，所以仔细设计块中记录的分配可以获得显著的性能提高。
- 数据字典，也称为系统目录，用于记录元数据，即关于数据的数据，例如关系名、属性名和类型、存储信息、完整性约束和用户信息。
- 减少磁盘访问数量的一种方法是在主存储器中保留尽可能多的块。因为在主存储器中保留所有的块

是不可能的，所以需要为块的存储而管理主存储器中可用空间的分配。缓冲区是主存储器的一部分，可用于存储磁盘块的拷贝。负责分配缓冲区空间的子系统称为缓冲区管理器。 468

术语回顾

- 物理存储介质
  - □ 高速缓冲存储器
  - □ 主存储器
  - □ 快闪存储器
  - □ 磁盘存储器
  - □ 光盘存储器
- 磁盘存储器
  - □ 盘面
  - □ 硬盘
  - □ 软盘
  - □ 磁道
  - □ 扇区
  - □ 读写头
  - □ 磁盘臂
  - □ 柱面
  - □ 磁盘控制器
  - □ 校验和
  - □ 坏磁道的重映射
- 磁盘性能度量
  - □ 访问时间
  - □ 寻道时间
  - □ 旋转延迟
  - □ 数据传输率
  - □ 平均故障时间(MTTF)
- 磁盘块
- 磁盘块访问优化
  - □ 磁盘臂调度
  - □ 电梯算法
  - □ 文件组织
  - □ 消除碎片
  - □ 非易失性写缓冲区
  - □ 非易失性随机存取存储器(NV-RAM)
  - □ 日志磁盘
- 独立磁盘冗余阵列(RAID)
  - □ 镜像
  - □ 数据拆分
  - □ 比特级拆分
  - □ 块级拆分
- RAID 级别
  - □ 0 级(块级拆分，没有冗余)
  - □ 1 级(块级拆分，镜像)
  - □ 3 级(比特级拆分，奇偶校验)
  - □ 5 级(块级拆分，分布式奇偶校验)
  - □ 6 级(块级拆分，P+Q 冗余)
- 重建的性能
- 软件 RAID
- 硬件 RAID
- 热交换
- 第三级存储
  - □ 光盘
  - □ 磁带
  - □ 自动光盘/磁带机
- 文件
- 文件组织
  - □ 文件头
  - □ 空闲列表
- 变长记录
  - □ 分槽的页结构
- 大对象
- 堆文件组织
- 顺序文件组织
- 散列文件组织
- 多表聚簇文件组织
- 搜索码
- 数据字典
- 系统目录
- 缓冲区
  - □ 缓冲区管理
  - □ 被钉住的块
  - □ 块的强制写出
- 缓冲区替换策略
  - □ 最近最少使用(LRU)
  - □ 立即丢弃
  - □ 最近最常使用(MRU)

实践习题

10.1　考虑表 10-18 所描述的 4 张磁盘上的数据和奇偶校验块的排列：B_i 表示数据块；P_i 表示奇偶校验块。奇偶校验块 P_i 是数据块 B_{i+1} ~ B_{i+4} 的校验块。这种排列如果有问题的话，会是什么问题？

盘1	盘2	盘3	盘4
B_1	B_2	B_3	B_4
P_1	B_5	B_6	B_7
B_8	P_2	B_9	B_{10}
⋮	⋮	⋮	⋮

图 10-18　数据和奇偶块排列

10.2　闪存存储：

a. 内存中用来映射逻辑页码到物理页码的闪存转换表如何创建？

b. 假设你有一个 64GB 的快闪存储系统，页面大小为 4096 字节。假设每一页有 32 位地址，并且转换表用数组形式存储，则该闪存转换表将有多大？ 469

c. 如果经常有大范围的连续逻辑页码映射到连续物理页码，请建议如何缩减转换表的大小。 470

10.3　当一个磁盘块正在被写的时候发生电源故障会导致块只写了一部分。假设这个部分写的块可以探测出来。一个原子的写块操作是一种或者完成对磁盘块的写，或者什么都不写(即不会部分写)的操作。提出一种利用原子的写块操作实现如下 RAID 策略的方法。你的方法应该包括从故障中恢复时的工作。

a. RAID 1 级(镜像)

b. RAID 5 级(块级拆分,分布的奇偶校验)

10.4 考虑从图 10-6 的文件中删除记录 5。比较下列实现删除的技术的相对优点:

a. 移动记录 6 到记录 5 所占用的空间,然后移动记录 7 到记录 6 所占用的空间。

b. 移动记录 7 到记录 5 所占用的空间。

c. 标记记录 5 被删除,不移动任何记录。

10.5 给出经过下面每一步后图 10-7 中文件的结构:

a. 插入(24556, Turnamian, Finance, 98000)。

b. 删除记录 2。

c. 插入(34556, Thompson, Music, 67000)。

10.6 考虑关系 *section* 和 *takes*。给出这两个关系的一个实例,包括 3 次开课,每次开课有 5 个学生选课。给出一个使用多表聚簇的这些关系的文件结构。

10.7 考虑下面在文件中跟踪空闲空间的位图技术,对文件中的每个块,在位图中维护两位。如果块的 0% – 30% 是满的,这两位用 00 表示,30% – 60% 用 01 表示,60% – 90% 用 10 表示,高于 90% 用 11 表示。这样的位图即使对很大的文件也可以保存在内存中。

a. 描述在记录插入和删除时如何保持位图的更新。

b. 概括位图技术与空闲列表相比在搜索空闲空间和更新空闲空间信息方面的好处。

471 10.8 快速找出一个块是否存在于缓冲区中,假如存在,它存于缓冲区的何处。这很重要。假设数据库缓冲区非常庞大,针对上述任务,你该使用什么(内存)数据结构?

10.9 对于下面的每种情况,给出一个关系代数表达式和一个查询处理策略的例子:

a. MRU 优于 LRU。

b. LRU 优于 MRU。

习题

10.10 列出你日常使用的计算机上用到的物理存储介质。给出每种介质上数据存取的速度。

10.11 磁盘控制器对扇区的重映射是如何影响数据检索率的?

10.12 RAID 系统通常允许你更换发生故障的磁盘而不需要停止对系统的访问。因此,发生故障的磁盘上的数据必须在系统运行的同时重建,并写到新更换的磁盘上。哪一级 RAID 在重建和持续磁盘访问之间的冲突最小?解释你的答案。

10.13 RAID 系统中的擦洗是什么,为什么擦洗很重要?

10.14 在变长记录表示中,用一个空位图来表示一个属性是否为空值。

a. 对于变长字段,如果值为空,那么偏移量字段和长度字段中存储什么?

b. 在一些应用中,元组拥有大量的属性,其中大多数属性为空。你能否更改这样的记录的表示,使得一个空值属性的开销仅为在空位图中的一个位。

10.15 解释为什么在磁盘块上分配记录的策略会显著影响数据库系统的性能。

10.16 如果可能,查出在你的本地计算机系统上运行的操作系统所使用的缓冲区管理策略和它提供的控制页面替换的机制。讨论它提供的控制替换策略如何能够对数据库系统的实现有用。

10.17 列出下列存储关系数据库的每个策略的两个优点和两个缺点:

472 a. 在一个文件中存储一个关系。

b. 在一个文件中存储多个关系(甚至是整个数据库)。

10.18 在顺序文件组织中,为什么即使当前只有一条输出记录,也要使用一个缓冲块?

10.19 给出关系 *Index_metadata* 的规范化形式,并解释为什么使用规范化的形式会导致更差的性能?

10.20 如果你的一些数据不应该在磁盘崩溃时丢失,且这些数据是写密集的,你如何存储这些数据?

10.21 在早期的磁盘中,每条磁道中扇区的数量对所有磁道都是一样的。现代的磁盘在外侧磁道上有较多的扇区,在内侧磁道上有较少的扇区(由于它们的长度更短)。这样的改变对磁盘速度的三个主要指标各有什么影响。

10.21 标准的缓冲区管理器假定每个页的大小和读取代价都是相同的。设想一个缓冲区管理器使用对对象

引用的比率，即一个对象在此前的 n 秒被访问的频率，而不是 LRU。假设我们要在缓冲区中存储变长和变读取代价（例如闪盘，它的读取代价取决于它所在的磁盘）的对象。缓冲器管理器可以怎样选择要删除缓冲区中的哪个页？

文献注解

Hennessy 等[2006]是一本广泛使用的计算机体系结构方面的教科书，它的内容涵盖了转换旁视缓冲区（translation look-aside buffer）、高速缓冲存储器和存储器管理单元的硬件特性。Rosch[2003]给出了对计算机硬件的一个优秀的概括性介绍，其内容涵盖了各种类型的存储技术，例如磁盘、光盘、磁带和存储接口。Patterson[2004]很好地讨论了延时的改进如何落后于带宽（传输率）的改进。

随着 CPU 速度的迅速增长，CPU 上的高速缓冲存储器比主存要更快。尽管数据库系统不控制数据如何存储在高速缓冲存储器中，但是，对在主存中组织数据、并以能够尽可能地利用高速缓冲存储器的方式编写程序的需求在不断上升。在这个领域的成果包括 Rao 和 Ross[2000]、Ailamaki 等[2001]和 Zhou 和 Ross[2004]、Garcia 和 Korth[2005]、Cieslewicz 等[2009]。

当代磁盘驱动器的详细规范可以从它们的制造商的 Web 站点上得到，例如：Hitachi、LaCie、Iomega、Seagate、Maxtor 和 Western Digital。

Patterson 等[1988]以及 Chen 和 Patterson[1990]给出了关于廉价磁盘冗余阵列（RAID）的讨论。Chen 等[1994]给出了对 RAID 原理和实现的一个极好的综述。Pless[1998]描述了 Reed-Solomon 码。

针对移动系统中的缓存数据讨论参见 Imielinski 和 Badrinath[1994]、Imielinski 和 Korth[1996]、Chandrasekaran 等[2003]。 473

特定数据库系统（如 IBM DB2、Oracle、Microsoft SQL Server 和 PostgreSQL）的存储结构都分别记录在它们的系统说明手册中。

大部分操作系统教材（包括 Silberschatz 等[2008]）都讨论了缓冲区管理。Chou 和 Dewitt[1985]给出了数据库系统中缓冲区管理的算法，并介绍了一种性能评估方法。 474

第11章
Database System Concepts, 6E

索引与散列

许多查询只涉及文件中的少量记录。例如，类似"找出物理系所有教师"或者"找出 *ID* 是 22201 的学生的总学分"的查询，就只涉及学生记录中的一小部分。如果系统读取 *instructor* 关系中每一个元组并检查 *dept_name* 值是否为"物理"，这样的操作方式是低效的。同样，只是为了查找一个 *ID* 是"32556"的元组而读取整个 *student* 关系也是低效的。理想情况下，系统应能够直接定位这些记录。为了允许这种访问方式，我们设计了与文件相关联的附加的结构。

11.1 基本概念

数据库系统中文件索引的工作方式非常类似于本书的索引。如果我们希望了解本书中某个特定主题(用一个词或者词组指定)的内容，可以在书后的索引中查找主题，找到它出现的页，然后读这些页，寻找我们需要的信息。索引中的词是按顺序排列的，因此，要找到我们所需要的词就容易了。而且，索引比书小得多，从而能进一步减少所需精力。

数据库系统中的索引与图书馆中书的索引所起的作用一样。例如，为了根据给定 *ID* 检索一条 *student* 记录，数据库系统首先会查找索引，找到相应记录所在的磁盘块，然后取出该磁盘块，得到所需的 *student* 记录。

在存储了数千条学生记录的大型数据库中，维护一个排序的学生 *ID* 列表的效果并不好，因为索引本身将非常大；而且，即使通过排序的索引减少了搜索时间，查找一个学生也仍然是非常费时的工作。我们可以使用更复杂的索引技术来作为替代。我们将在本章讨论几种这样的技术。

有两种基本的索引类型：

475

- **顺序索引**。基于值的顺序排序。
- **散列索引**。基于将值平均分布到若干散列桶中。一个值所属的散列桶是由一个函数决定的，该函数称为**散列函数**(hash function)。

我们将考虑用于顺序索引和散列的几种技术。没有哪一种技术是最好的，只能说某种技术对特定的数据库应用是最适合的。对每种技术的评价必须基于下面这些因素。

- **访问类型**(access type)：能有效支持的访问类型。访问类型可以包括找到具有特定属性值的记录，以及找到属性值落在某个特定范围内的记录。
- **访问时间**(access time)：在查询中使用该技术找到一个特定数据项或数据项集所需的时间。
- **插入时间**(insertion time)：插入一个新数据项所需的时间。该值包括找到插入这个新数据项的正确位置所需的时间，以及更新索引结构所需的时间。
- **删除时间**(deletion time)：删除一个数据项所需的时间。该值包括找到待删除项所需的时间，以及更新索引结构所需的时间。
- **空间开销**(space overhead)：索引结构所占用的额外存储空间。情若存储索引结构的额外空间大小适度，通常牺牲一定的空间代价来换取性能的提高是值得的。

通常需要在一个文件上建立多个索引。例如，可能按作者、主题或者书名来查找一本书。

用于在文件中查找记录的属性或属性集称为**搜索码**(search key)。注意这里的码的定义与主码、候选码以及超码中的定义不同。(遗憾的是)码在现实中具有多种广为接受的含义。使用本文中搜索码的概念，我们认为，如果一个文件上有多个索引，那么它就有多个搜索码。

11.2　顺序索引

为了快速随机访问文件中的记录，可以使用索引结构。每个索引结构与一个特定的搜索码相关联。正如书中的索引或者图书馆目录一样，顺序索引按顺序存储搜索码的值，并将每个搜索码与包含该搜索码的记录关联起来。

被索引文件中的记录自身也可以按照某种排序顺序存储，正如图书馆中的书按某些属性（如杜威十进制数）顺序存放一样。一个文件可以有多个索引，分别基于不同的搜索码。如果包含记录的文件按照某个搜索码指定的顺序排序，那么该搜索码对应的索引称为**聚集索引**（clustering index）。聚集索引也称为**主索引**（primary index）；主索引这个术语看起来是表示建立在主码上的索引，但实际上它可以建立在任何搜索码上。聚集索引的搜索码常常是主码，尽管并非必须如此。搜索码指定的顺序与文件中记录的物理顺序不同的索引称为**非聚集索引**（nonclustering index）或**辅助索引**（secondary index）。常用术语"聚集的"（clustered）和"非聚集的"（nonclustered）来代替"聚集"（clustering）和"非聚集"（nonclustering）。 476

在 11.2.1～11.2.3 节中，我们假定所有文件都按照某些搜索码顺序排列。这种在搜索码上有聚集索引的文件称作**索引顺序文件**（index-sequential file）。它们是数据库系统中最早采用的索引模式之一。是针对既需顺序处理整个文件又需随机访问单独记录的应用而设计的。11.2.4 节将讨论辅助索引。

图 11-1 是取自大学例子中 *instructor* 记录的一个顺序文件。在图 11-1 所示的例子中，教师 *ID* 用作搜索码，记录按照该搜索码顺序存放。

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

图 11-1　*instructor* 记录的顺序文件

11.2.1　稠密索引和稀疏索引

索引项（index entry）或**索引记录**（index record）由一个搜索码值和指向具有该搜索码值的一条或者多条记录的指针构成。指向记录的指针包括磁盘块的标识和标识磁盘块内记录的块内偏移量。

我们可以使用的顺序索引有两类。 477

- **稠密索引**（dense index）：在稠密索引中，文件中的每个搜索码值都有一个索引项。在稠密聚集索引中，索引项包括搜索码值以及指向具有该搜索码值的第一条数据记录的指针。具有相同搜索码值的其余记录顺序地存储在第一条数据记录之后，由于该索引是聚集索引，因此记录按照相同的搜索码值排序。

  在稠密非聚集索引中，索引必须存储指向所有具有相同搜索码值的记录的指针列表。

- **稀疏索引**（sparse index）：在稀疏索引中，只为搜索码的某些值建立索引项。只有当关系按搜索码排列顺序存储时才能使用稀疏索引，换句话说，只有索引是聚集索引时才能使用稀疏索引。和稠密索引一样，每个索引项也包括一个搜索码值和指向具有该搜索码值的第一条数据记录的指针。为了定位一条记录，我们找到其最大搜索码值小于或等于所查找记录的搜索码值的索引项。然后从该索引项指向的记录开始，沿着文件中的指针查找，直到找到所需记录为止。

图 11-2 和图 11-3 分别是为 *instructor* 文件建立的稠密索引和稀疏索引。假如我们现在要查找 *ID* 是

"22222"的教师记录。利用图 11-2 的稠密索引，我们可以顺着指针直接找到第一条所需记录。因为 *ID* 是主码，所以数据库中只存在一条这样的记录。于是搜索完成。如果使用稀疏索引(图 11-3)，就找不到一个 *ID* 为"22222"的索引项。因为"22222"之前的最后一项是"10101"(数字排序)，于是我们顺着该指针查找，然后按顺序读取 *instructor* 文件，直到查找到所需的记录。

考虑一本(印刷好的)字典。每页页眉都顺序地列出了该页中按字母序出现的第一个单词。字典中每页顶部的单词共同构成了字典页内容的稀疏索引。

再举一个例子，假设搜索码值并不是一个主码。图 11-4 表示为以 *dept_name* 为搜索码的 *instructor* 文件的稠密聚集索引。在这种情况下，*instructor* 文件按照搜索码 *dept_name* 排序，而不是 *ID*，否则建立在 *dept_name* 上的索引将变成非聚集索引。假设我们正在查找历史系的记录。利用图 11-4 的稠密索引，我们按照指针直接找到历史系的第一条记录。处理此记录，按照该记录中的指针找到按搜索码 *dept_name* 排序的下一条记录。继续处理记录，直到我们遇到一条不是历史系的记录。

正如我们所看到的那样，利用稠密索引通常可以比稀疏索引更快地定位一条记录。但是，稀疏索引也有比稠密索引优越的地方：它所占空间较小，并且插入和删除时所需的维护开销也较小。

系统设计者必须在存取时间和空间开销之间进行权衡。尽管有关这一权衡的决定依赖于具体的应用，但是为每个块建一个索引项的稀疏索引是一个较好的折中。原因在于，处理数据库查询的开销主要由把块从磁盘读到主存中的时间决定。一旦将块放入主存，扫描整个块的时间就是可以忽略的。使用这样的稀疏索引，可以定位包含所要查找记录的块。这样，只要记录不在溢出块(见 10.6.1 节)中，就能使块访问次数最小，同时能保持索引尽可能小(因而也就减少了空间开销)。

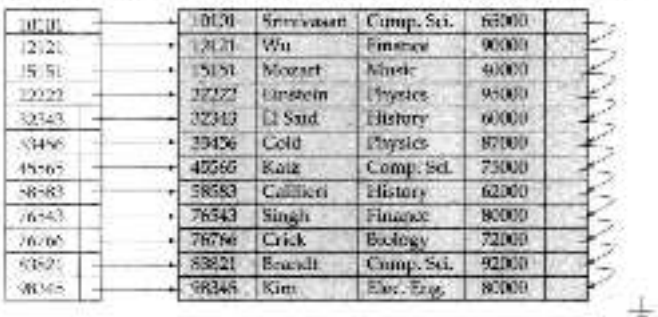


图 11-2 稠密索引

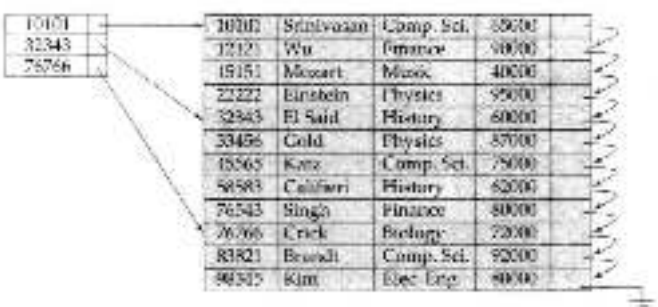


图 11-3 稀疏索引

要使上述技术完全通用，必须考虑具有相同搜索码值的记录跨多个块的情况。可以很容易地修改我们的方法使其能处理这样的情况。

11.2.2 多级索引

假设我们在具有 1 000 000 个元组的关系上建立了稠密索引。索引项比数据记录要小，因此我们假设一个 4KB 的块中可以容纳 100 条索引项。这样，我们的索引需要占用 10 000 个块。假如该关系具有 100 000 000 个元组，那么索引就需要占用 1 000 000 个块，即 4GB 的空间。这样大的索引以顺序文件的方式存储在磁盘上。

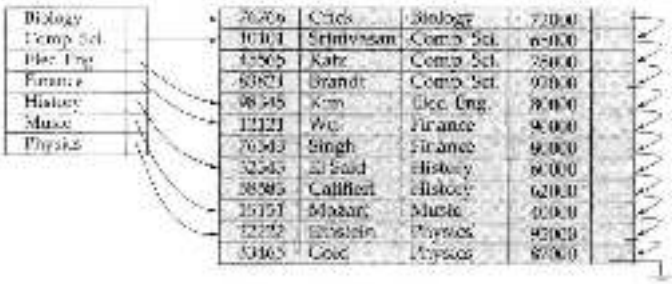


图 11-4　稀疏索引

如果索引小到可以放在主存中，搜索一个索引项的时间就会很短。但是，如果索引过大而不能放在主存中，那么当需要时，就必须从磁盘中取索引块。（即使索引比计算机的主存小，但内存还需要处理其他一些任务，因此也可能不能将整个索引放置在内存中。）于是搜索一个索引项需要多次读取磁盘块。

可以在索引文件上使用二分法搜索来定位索引项，但是搜索的开销依然很大。如果索引占据 b 个磁盘块，二分法搜索需要读取 $\lceil \log_2(b) \rceil$ 个磁盘块。（$\lceil x \rceil$ 表示大于或等于 x 的最小整数，即向上取整。）对于占用 10 000 个块的索引，二分法搜索需要 14 次读块操作。在读一个块平均需要 10 毫秒的磁盘系统中，该搜索将耗时 140 毫秒。这也许看上去不是很长，但是一秒钟内我们只能进行 7 次索引搜索，而一会儿我们将会看到，一个更有效的搜索机制可以让我们一秒钟内执行更多次搜索。请注意，如果使用了溢出块，那么就不能使用二分法搜索。这种情况下通常采用顺序搜索，需要读块 b 次，这将耗费更长的时间。因此，搜索一个大的索引可能是一个相当耗时的过程。

478
480

为了处理这个问题，我们像对待其他任何顺序文件那样对待索引文件，并且在原始的内层索引上构造一个稀疏的外层索引，如图 11-5 所示。注意到索引项总是有序的，这使得外层索引可以是稀疏的。为了定位一条记录，我们首先在外层索引上使用二分法搜索找到其最大搜索码值小于或等于所需搜索码值的记录。指针指向一个内层索引块。我们扫描这一块，直到找到其最大搜索码值小于或等于所需搜索码值的记录。这条记录的指针指向包含所查找记录的文件块。

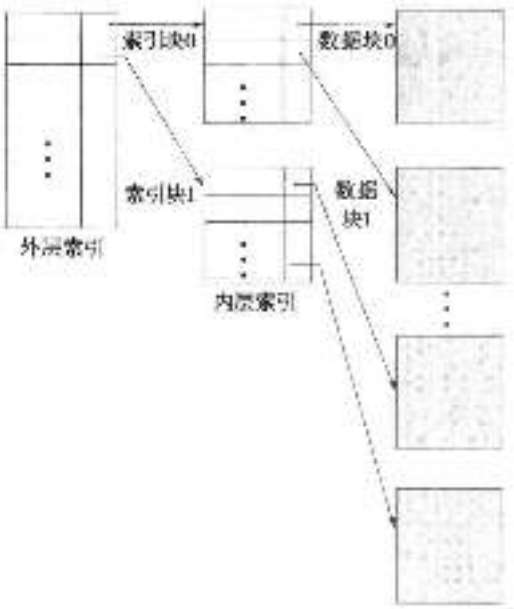


图 11-5　二级稀疏索引

在该例子中，占用 10 000 个块的内层索引需要外层索引中有 10 000 个索引项，这些索引项仅占用 100 个块。如果我们假设外层索引已经在主存中，那么当使用多级索引时，一次查询只需要读取一个索引块，而不像我们使用二分法搜索时读取 14 个块。因此，我们每秒可以执行 14 次索引查找。

如果文件极其庞大，甚至外层索引也可能大到不能装入主存。对于具有 100 000 000 个元组的关系，内层索引将占用 1 000 000 个块，外层索引占用10 000个块，或者 40MB。因为在主存中有很多需求，所以有可能不能为这个特定的外层索引而预留出如此多的主存。在这种情况下，可以创建另一级索引。事实上，可以根据需要多次重复此过程。具有两级或两级以上的索引称为多级（multilevel）索引。利用多级索引搜索记录与用二分法搜索记录相比需要的 I/O 操作要少得多[1]。

---

[1] 在早期的基于磁盘的索引中，将级索引有一个相应的物理存储单位。因此，我们可以在磁道、柱面和磁盘级别上创建索引。今天看来，这样一个层次是不合理的，因为磁盘子系统隐藏了磁盘存储的物理细节，并且磁盘数和每张磁盘的盘片数与柱面数或每条磁道的字节数相比是非常小的。

多级索引和树结构紧密相关，正如用于内存索引的二叉树。稍后将在 11.3 节讨论这种关系。

11.2.3 索引的更新

无论采用何种形式的索引，每当文件中有记录插入或删除时，索引都需要更新。此外，如果文件中的记录更新，任何搜索码属性受影响的索引也必须更新。例如，如果一个教师的系发生了变化，那么在 *instructor* 的 *dept_name* 属性上的索引也必须相应地更新。这样的记录更新可以设计为对旧记录的删除，以及随后对新记录的插入。因此，我们只需要考虑索引的插入和删除，并不需要明确地考虑更新。

我们首先描述单级索引的更新算法。

- **插入**。系统首先用出现在待插入记录中的搜索码值进行查找，并根据索引是稠密索引还是稀疏索引而进行下一个操作。

  □ 稠密索引：

  1. 如果该搜索码值不在索引中，系统就在索引中合适的位置插入具有该搜索码值的索引项。
  2. 否则进行如下操作：
     a. 如果索引项存储的是指向具有相同搜索码值的所有记录的指针，系统就在索引项中增加一个指向新记录的指针。
     b. 否则，索引项存储一个仅指向具有相同搜索码值的第一条记录的指针，系统把待插入的记录放到具有相同搜索码值的其他记录之后。

  481 ~ 482

  □ 稀疏索引：我们假设索引为每个块保存一个索引项。如果系统创建一个新的块，它会将新块中出现的第一个搜索码值（按照搜索码的顺序）插入到索引中。另一方面，如果这条新插入的记录含有块中的最小搜索码值，那么系统就更新指向该块的索引项；否则，系统对索引不做任何改动。

- **删除**。为删除一条记录，系统首先查找要删除的记录，然后下一步的操作取决于索引是稠密索引还是稀疏索引。

  □ 稠密索引：

  1. 如果被删除的记录是具有这个特定搜索码值的唯一的一条记录，系统就从索引中删除相应的索引项。
  2. 否则采取如下操作：
     a. 如果索引项存储的是指向所有具有相同搜索码值的记录的指针，系统就从索引项中删除指向被删除记录的指针。
     b. 否则，索引项存储的是指向具有该搜索码值的第一条记录的指针。在这种情况下，如果被删除的记录是具有该搜索码值的第一条记录，系统就更新索引项，使其指向下一条记录。

  □ 稀疏索引：

  1. 如果索引不包含具有被删除记录搜索码值的索引项，则索引不必做任何修改。
  2. 否则系统采取如下操作：
     a. 如果被删除的记录是具有该搜索码值的唯一记录，系统用下一个搜索码值（按搜索码顺序）的索引记录替换相应的索引记录。如果下一个搜索码值已经有一个索引项，则删除而不是替换该索引项。
     b. 否则，如果该搜索码值的索引记录指向被删除的记录，系统就更新索引项，使其指向具有相同搜索码值的下一条记录。

多级索引的插入和删除算法是对上述算法的一个简单扩充。在插入和删除时，系统对底层索引的更新如上所述。而对于第二层而言，底层索引不过是一个包含记录的文件。因此，如果底层索引发生了改变，第二层索引就可以像上面描述的那样进行更新。如果还有更高层的索引，可以采用同样的技术更新更高层的索引。

11.2.4 辅助索引

辅助索引必须是稠密索引，对每个搜索码值都有一个索引项，而且对文件中的每条记录都有一个

指针。而聚集索引可以是稀疏索引，可以只存储部分搜索码值，因为主如前面所描述的那样，通过顺序扫描文件的一部分，我们总可以找到两个有索引项的搜索码值之间的搜索码值所对应的记录。如果辅助索引只存储部分搜索码值，两个有索引项的搜索码值之间的搜索码值所对应的记录可能存在于文件中的任何地方，并且我们通常只能通过扫描整个文件才能找到它们。 483

候选码上的辅助索引看起来和稠密聚集索引没有太大的区别，只不过索引中一系列的连续值指向的记录不会连续存放的。然而，一般来说，辅助索引的结构可能和聚集索引不同。如果聚集索引的搜索码不是候选码，索引只要指向具有该特定搜索码值的第一条记录就足够了，因为其他的记录可以通过对文件运行顺序扫描得到。

反之，如果辅助索引的搜索码不是一个候选码，仅仅只有指向每个搜索码值的第一条记录的指针是不够的。具有同一个搜索码值的其他记录可能分布在文件的任何地方，因为记录按聚集索引而不是辅助索引的搜索码顺序存放。因此，辅助索引必须包含指向每一条记录的指针。

我们可以用一个附加的间接指针层来实现非候选码的搜索码上的辅助索引。在这样的辅助索引中，指针并不直接指向文件，而是指向一个包含文件指针的桶。图 11-6 给出了这样的一个辅助索引结构，它在 *instructor* 文件的搜索码 *salary* 上使用了一个附加的间接指针层。

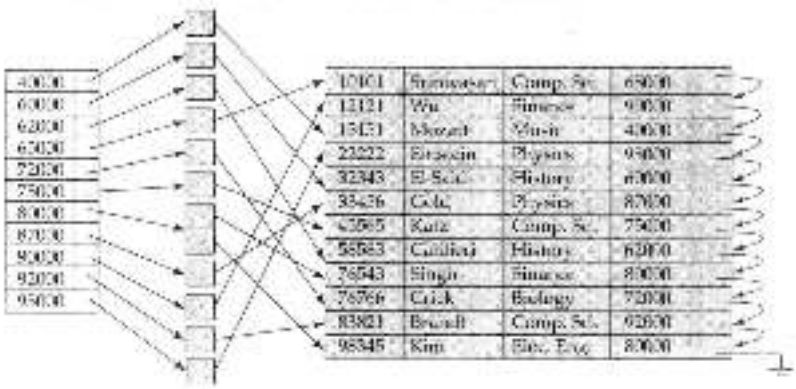


图 11-6　*instructor* 文件的辅助索引，基于非候选码 *salary*

按聚集索引顺序对文件进行顺序扫描是非常有效的，因为文件中记录的物理存储顺序和索引顺序一致。但是，我们不能（除了极少数特殊情况外）使存储文件的物理顺序既和聚集索引的搜索码顺序相同，又和辅助索引的搜索码顺序相同。由于辅助码的顺序和物理码的顺序不同，因此如果我们想要按辅助码的顺序对文件运行顺序扫描，那么每读一条记录都很可能需要从磁盘读入一个新的块，这是很慢的。 484

索引的自动生成

如果一个关系声明为有一个主码，大多数数据库实现会在主码上自动创建一个索引。只要一个元组插入到关系中，该索引就可以用来检查没有违反主码约束（即没有重复的主码值）。假如主码上没有索引，只要插入一个元组，整个关系就必须被读取，以确保满足主码约束。

前面所描述的关于删除和插入的过程也适用于辅助索引，所采用的操作和为文件中的每条记录存储一个指针的稠密索引需要的操作一样。如果文件具有多个索引，无论何时修改文件，它的每个索引都必须更新。

辅助索引能够提高使用聚集索引搜索码以外的码的查询性能。但是，辅助索引显著增加了数据库更新的开销。数据库设计者根据对查询和更新相对频率的估计来决定哪些辅助索引是需要的。

11.2.5　多码上的索引

虽然我们迄今所看到的例子在搜索码中只含有单个属性，但一般来说一个搜索码可以有多个属性。一个包含多个属性的搜索码称为**复合搜索码**（composite search key）。这个索引的结构和任何其他索引一样，唯一不同的地方是搜索码不是单个属性，而是一个属性列表。这个搜索码可以表示为形式如

$(a_1, \cdots, a_n)$的一组值，其中$A_1, \cdots, A_n$是索引属性。搜索码值按字典序(lexicographic ordering)排序。例如，考虑有两个属性的搜索码，如果$a_1 < b_1$或$a_1 = b_1$且$a_2 < b_2$，则$(a_1, a_2) < (b_1, b_2)$。字典序和单词按字母排序基本相同。

举例来说，考虑*takes*关系上的一个复合搜索码(*course_id*, *semester*, *year*)。这样一个索引在查找所有特定学期/年注册了特定课程的学生时是很有用的。一个在复合码上的有序索引可以用来有效地查找某些其他类型的查询，就像我们将在11.5.2节中见到的一样。

11.3 B⁺树索引文件

485 索引顺序文件组织最大的缺点在于，随着文件的增大，索引查找性能和数据顺序扫描性能都会下降。虽然这种性能下降可以通过对文件进行重新组织来弥补，但是我们不希望频繁地进行重组。

B⁺树(B⁺-tree)索引结构是在数据插入和删除的情况下仍能保持其执行效率的几种使用最广泛的索引结构之一。B⁺树索引采用平衡树(balanced tree)结构，其中树根到树叶的每条路径的长度相同。树中每个非叶结点有$\lceil n/2 \rceil - n$个子女，其中n对特定的树是固定的。

我们将看到B⁺树结构会增加文件插入和删除处理的性能开销，同时会增加空间开销。但是即使对更新频率较高的文件来说，这种开销也是可接受的，因为这样能够减小文件重组的代价。此外，由于结点有可能是半空的(如果它们具有最少子结点数的话)，这将造成空间的浪费。但是，考虑到B⁺树所带来的性能提高，这种空间开销也是可以接受的。

11.3.1 B⁺树的结构

B⁺树索引是一种多级索引，但是其结构不同于多级索引顺序文件。典型的B⁺树结点结构如图11-7所示。它最多包含$n-1$个搜索码值$K_1, K_2, \cdots, K_{n-1}$，以及$n$个指针$P_1, P_2, \cdots, P_n$。每个结点中的搜索码值非序存放，因此，如果$i < j$，那么$K_i < K_j$(假设目前没有重复的码值)。



图11-7 典型的B⁺树结点

我们首先考察叶结点(leaf node)的结构。对$i = 1, 2, \cdots, n-1$，指针P_i指向具有搜索码值K_i的一条文件记录。指针P_n有特殊的作用，我们将稍后讨论。

图11-8是*instructor*文件的B⁺树的一个叶结点，其中我们设n等于4，搜索码是*name*。

知道了叶结点的结构之后，我们来看一看搜索码值是如何赋给特定结点的。每个叶结点最多可有$n-1$个值。我们允许叶结点包含的值的个数最少为$\lceil (n-1)/2 \rceil$。在B⁺树例子中$n=4$，每个叶子必须包含最少两个并且最多3个值。

各叶结点中值的范围互不重合，除非有重复的搜索码值，在这种情况下，一个值可能出现在多个叶结点中。说明确些，如果L_i和L_j是两个叶结点且$i < j$，那么L_i中的所有搜索码值都小于或等于L_j中的所有搜索码值。要使B⁺树索引成为稠密索引(通常情况)，各搜索码值都必须出现在某个叶结点中。

现在我们可以来解释指针P_n的作用了。因为各叶结点之间按照所含的搜索码值大小有一个线性的顺
486 序，所以我们可以用P_n将叶结点按搜索码顺序串在一起。这种排序可以对文件进行高效的顺序处理。

B⁺树的非叶结点(nonleaf node)形成叶结点上的一个多级(稀疏)索引。非叶结点的结构和叶结点的相同，只不过非叶结点中所有的指针都是指向树中结点的指针。一个非叶结点可以容纳最多n个指针，同时必须至少容纳$\lceil n/2 \rceil$个指针。结点的指针数称为该结点的扇出。非叶结点也称为内部结点(internal node)。

让我们考虑一个包含m个指针的结点($m \leq n$)。对$i = 2, 3, \cdots, m-1$，指针P_i指向一棵子树，该子树包含的搜索码值小于K_i且大于等于K_{i-1}。指针P_m指向子树中所含搜索码值大于等于K_{m-1}的那一部分，而指针P_1指向子树中所含搜索码值小于K_1的那一部分。

根结点与其他非叶结点不同，它包含的指针数可以小于$\lceil n/2 \rceil$；但是，除非整棵树只有一个结点，否则根结点必须至少包含两个指针。对任意n，我们总可以构造满足上述要求的B⁺树。

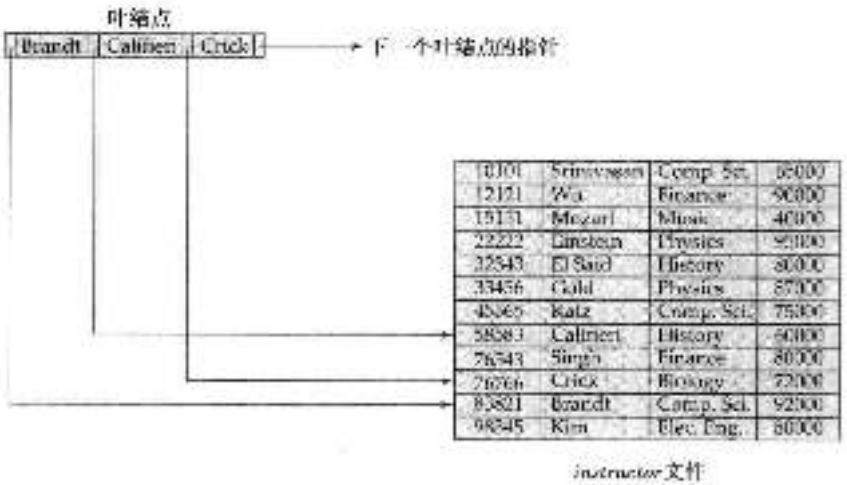
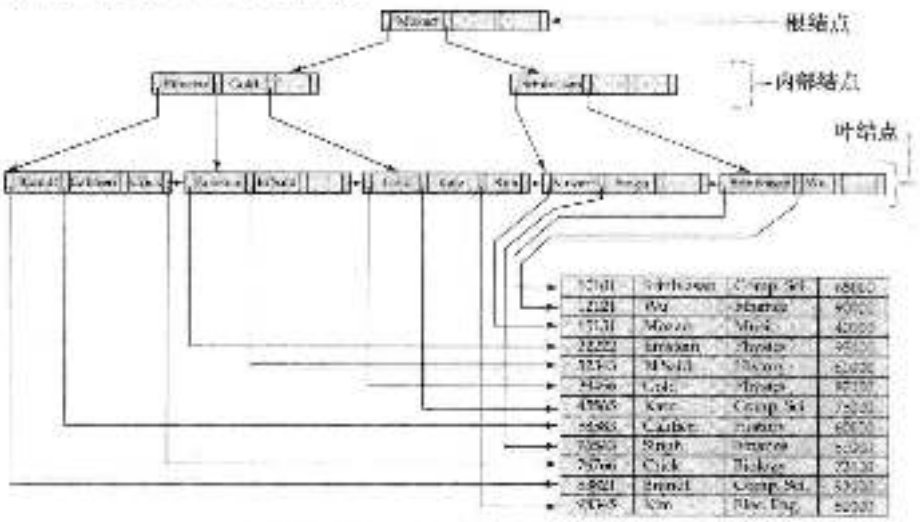


图 11-8　*instructor* 的 B⁺树索引($n=4$)的一个叶结点

例 11-9 给出了 *instructor* 文件($n=4$)的一棵完整的 B⁺树。为了简化起见，我们忽略了空指针。图 11-8 中不包含箭头的指针区域可以理解为包含空值。

例 11-10 给出了 *instructor* 文件($n=6$)的另一棵完整的 B⁺树。可以观察到这棵树的高度小于前面 [487] $n=4$的树。

这些 B⁺树的例子都是平衡的，即从根到叶结点的每条路径长度都相同。对于 B⁺树来说这是一个必需的性质。实际上 B⁺树的"B"就表示"平衡"(balanced)的意思。正是 B⁺树的这一平衡性保证了 B⁺树索引有良好的查找、插入和修改性能。



例 11-9　*instructor* 文件的 B⁺树($n=4$)

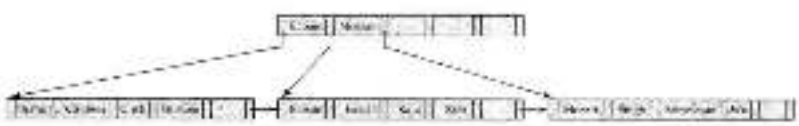


图 11-10　*instructor* 文件的 B⁺树($n=6$)

11.3.2　B⁺树的查询

让我们考虑一下如何处理 B⁺树上的查询。假设我们要找出搜索码值为 V 的所有记录。图 11-11 表

示执行这个任务的函数 find( ) 的伪代码。

直观地看，如果树中存在指定的值，那么该函数就从树的根结点开始，向下周游树直到它到达包含指定值的叶结点。具体地说，从根结点作为当前结点出发，函数会重复如下步骤直到到达一个叶结点：首先，检查当前结点，查找最小的 i 使得搜索码值 K_i 大于等于 V。假设找到了这样的值，如果 K_i 等于 V，那么将当前结点置为由 P_{i+1} 指向的结点；而如果 K_i 大于 V，那么将当前结点置为由 P_i 指向的结点。如果没有找到这样的 K_i 值，那么显然 $V \geq K_{n-1}$，其中 P_n 是结点中的最后一个非空指针。在这个情况下，将当前结点置为由 P_n 指向的结点。重复上述过程，周游整棵树直到访问叶结点。

488

在叶结点中，如果包含等于 V 的搜索码值，令 K_i 为第一个这样的值，则指针 P_i 指向具有搜索码值 K_i 的记录。该函数返回叶结点 L 和索引 i。如果在叶结点中没有找到等于 V 的搜索码值，即在关系中不存在具有码值为 V 的记录，函数 *find*( ) 则返回空值，显示失败。

要处理重复的搜索码，需要对图 11-11 中的 find( ) 函数进行修改。对于叶结点和内部结点中重复的搜索码，如果 $i<j$，则 $K_i < K_j$ 不一定成立，但是肯定地 $K_i \leq K_j$ 成立。而且，P_i 所指向的子树中的记录可能包含小于或等于 K_i 的值（要理解为什么是这样的，请考虑 P_i 和 P_{i-1} 所指向的，都包含一个重复的搜索码值 v 的两个相邻的叶结点，这里 $K_{i-1}=v$）。要解决这个问题，我们必须修改 find( ) 函数中的循环，置 $C=C.P_i$，即使 $V=C.K_i$。而且，由此达到的叶结点 C 可能仅包含小于 V 的搜索码（即使 V 在树中并不存在）；在这种情况下，find 过程必须置 $C=C$ 的右兄弟，并且再次检查 C 是否包含 V。我们把这个修改过的 find 过程称作 findFirst，它返回值 V 在树中的第一次出现。

如图 11-11 所示的程序 printAll 描述如何检索所有搜索码值等于 V 的记录。printAll 过程调用 findFirst，去找出具有 V 的第一次出现的结点 L，然后在结点 L 中遍历其余的码，以找出具有搜索码值 V 的其他记录。如果结点 L 至少包含一个大于 V 的搜索码值，那么不存在更多的值为 V 的记录。否则，由指针 P_n 指向的下一个叶结点有可能包含值为 V 的搜索码。然后必须搜索由指针 P_n 指向的结点来进一步查找具有搜索码值为 V 的记录。如果由 P_n 指向的结点的最大搜索码也是 V，那么可能需要周游更多叶结点，从而找到所有匹配记录。在 printAll( ) 中的 **repeat** 循环执行叶结点的周游，直到找到所有匹配记录。

一个实际的实现可能提供 find( ) 的一个版本，它支持与 JDBC ResultSet 所提供的类似的迭代接口，正如我们在 5.1.1 节中所看到的一样。这种迭代接口可以提供一个 next( ) 方法，该方法可以反复调用，来查找具有特定搜索码值的连续记录。和 printAll( ) 相似，next( ) 方法在叶结点进行遍历。但是每次调用只执行一步，返回离开前遍历到的记录，这样连续的 next( ) 调用遍历相继的记录。为了简单起见，我们省略了细节，将迭代接口伪码作为练习留给感兴趣的读者。

B$^+$ 树也可以查找所有搜索码值在特定区间 (L, U) 的记录。例如，在关系 *instructor* 的属性 *salary* 上的 B$^+$ 树中，我们查找所有工资在特定区间例如 (50 000, 100 000)（换句话说，即介于 50 000 和 100 000 之间的所有工资）的所有 *instructor* 记录。这样的查找称作**范围查询**（range query）。我们可以创建一个 printRange(L, U) 函数来执行这样的查找，该函数的主体和 printAll( ) 一样，除了以下不同：printRange( ) 调用 find(L) 而不是调用 find(V)，然后和函数 printAll( ) 一样周游记录。不同的是 printRange( ) 在满足 $L.K_i > U$ 而不是 $L.K_i > V$ 时停止。

在处理一个查询的过程中，我们需要遍历树中从根到某个叶结点的一条路径。如果文件中有 N 个搜索码值，那么这条路径的长度不超过 $\lceil \log_{\lceil n/2 \rceil}(N) \rceil$。

实际上只需访问几个结点。结点的大小一般等于磁盘块大小，通常为 4KB。如果搜索码的大小为 12 字节，磁盘指针的大小为 8 字节，那么 n 大约为 200。即使采用更保守的估计，假设搜索码大小达到 32 字节，n 也大约为 100。在 $n=100$ 的情况下，如果文件中搜索码值共有 1 百万个，一次查找也只需要访问 $\lceil \log_{50}(1\,000\,000) \rceil = 4$ 个结点。因此，查找时最多只需要从磁盘读 4 个块。通常根结点访问频繁，很可能在缓冲区中，因此一般只要从磁盘读取三个或更少的磁盘块。

B$^+$ 树结构与内存中树结构（如二叉树）的一个重要的区别在于结点的大小及其造成的树的高度的不同。二叉树的结点很小，每个结点最多有两个指针。而 B$^+$ 树的结点非常大（一般是一个磁盘块的大小），每个结点可以有大量指针。因此，B$^+$ 树一般肥而矮，不像二叉树那样瘦而高。在平衡二叉树中，查找路径的长度可达 $\lceil \log_2(N) \rceil$，其中 N 为搜索码值的个数。当 N 如上例中那样为 1 000 000 时，平衡

二叉树大约需要访问 20 个结点。如果每个结点在不同的磁盘块中，处理一次查找需要读 20 个块，而 B⁺树只须读 4 个块。区别是显著的，因为一次块读取可能要求一次磁盘臂寻道，而且一张典型的磁盘上一次磁盘寻道的块读取需要 10 毫秒。

```
function find( value V )
    /* 假设没有重复码返回叶结点 C 和索引 i 使得 C.P_i 指向第一条搜索码值等于 V 的记录 */
        置 C = 根结点
        while C 不是叶结点 begin
            令 i = 满足 V≤C.K_i 的最小值
            if 没有这样的 i then begin
                令 P_n = 结点中最后一个非空指针
                置 C = C.P_n
            end
                else if ( V = C.K_i )
                    then 置 C = C.P_{i+1}
                else C = C.P_i /* V < C.K_i */
            end
            /* C 是叶结点 */
            设 i 是满足 K_i = V 的最小值
            if 有这样的 i 存在
                then 返回(C, i)
                else 返回空 /* 没有码值等于 V 的记录存在 */

    procedure printAll( value V )
    /* 打印所有搜索码值等于 V 的记录 */
        置 done = false
        置(L, i) = findFirst(V)
        if( (L, i) 为空 ) return
        repeat
            repeat
            打印指针 L.P_i 指向的记录
            置 i = i + 1
        until( i > L 中码的数目或 L.K_i > V )
        if( i > L 中码的数目 )
            then L = L.P_n
            else 置 done = true;
        until ( done or L 为空 )
```

图 11-11　B⁺树上的查询

11.3.3　B⁺树的更新

当一条记录从关系中插入或者删除，建立在该关系上的索引必须相应地更新。回想前面，记录的更新操作可以设计为对旧记录的删除，以及随即对更新后记录的插入。因此我们仅考虑插入和删除的情况即可。

插入和删除要比查找更加复杂，因为结点可能因为插入而变得过大而需要**分裂**(split)或因为删除而变得过小(指针数少于 $\lceil n/2 \rceil$)而需要**合并**(coalesce)(即合并结点)。此外，当一个结点分裂或一对结点合并时，我们必须保证 B⁺树能保持平衡。为了引入 B⁺树中处理插入和删除的思想，我们下面暂时假设结点从来不会变得过大或过小。在这个假设下，插入和删除将按如下方式进行。

- **插入**。在函数 find()(见图 11-11)中使用和查找一样的技术，我们首先找到搜索码值将出现的叶结点。然后在叶节点中插入一条新记录(即一对搜索码值和记录指针)，使得插入后搜索码仍然有序。
- **删除**。使用和查找一样的技术，我们通过查找已删除记录的搜索码值，找到包含待删除项的叶结点；如果多项含有相同的搜索码值，我们遍历所有这些相同搜索码值的项，直到找到指向被删除记录的项。然后我们从叶结点中移除该项。该叶结点中已删除项右边的所有项左移一个位置，以便在删除该项后不会有空隙。

现在我们通过处理结点分裂和结点合并，来考虑插入和删除的一般情况。

11.3.3.1 插入

现在我们考虑结点必须分裂的一个插入的例子。假设我们需要往 *instructor* 关系中插入一条 *name* 值为"Adams"的记录。然后我们需要往图 11-9 所示的 B+ 树中插入一条值为"Adams"的项。按照查找算法，我们发现"Adams"应出现在包含"Brandt"、"Califieri"和"Crick"的页结点中。该页结点中已没有插入搜索码值"Adams"所需的空间。因此该结点分裂为两个结点。图 11-12 表示在在插入"Adams"后叶结点分裂成两个叶结点。其中一个叶结点的搜索码值为"Adams"和"Brandt"，另一个为"Califieri"和"Crick"。一般来说，我们将这 n 个搜索码值（叶结点中原有的 $n-1$ 个值再加上新插入的值）分为两组，前 $\lceil n/2 \rceil$ 个放在原来的结点中，剩下的放在一个新结点中。

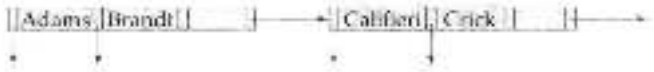


图 11-12 插入"Adams"时叶结点的分裂

在分裂一个结点后，我们必须将新的叶结点插入到 B+ 树结构中。在这个例子中，新结点以"Califieri"作为其最小搜索码值。我们需要将该搜索码值插入已分裂结点的父结点中。图 11-13 中的 B+ 树给出了插入后的结果。因为父结点中有空间容纳插入的搜索码值，所以可以直接插入而无须分裂结点。若没有空间，则父结点必须分裂，并需要在它的父结点中插入一个项。在最坏的情况下，从叶结点到根结点的路径上的所有结点必须分裂。如果根本身也分裂了，那么整棵树的深度就加大了。

一个非叶结点的分裂与叶结点的分裂略有不同。图 11-14 表示在图 11-13 所示的树中插入搜索码"Lamport"的结果。"Lamport"将要插入的叶结点已经含有"Gold"、"Katz"和"Kim"，因此叶结点需要分裂。新的右手侧结点从分裂中产生，并含有搜索码值"Kim"和"Lamport"。一个（Kim，$n1$）项必须添加到父结点中，其中 $n1$ 指向新的结点。然而，父结点上已经没有空间来增加新的项，因此父结点必须要分裂。为此，从概念上父结点临时扩张，新的项被添加，然后过满结点立刻分裂。

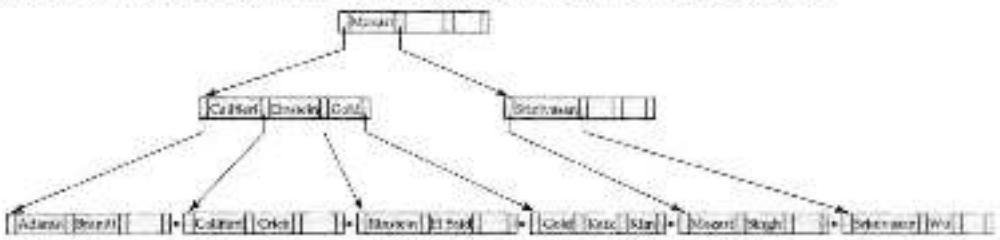


图 11-13 把"Adams"插入图 11-9 的 B+ 树中

当一个过满的非叶结点分裂时，孩子指针的分裂介于原始的和新创建的结点之间。在该例子中，原始结点保留了前三个指针，右边新创建的结点包含剩下的两个指针。但是，搜索码值的处理略有不同。位于移动到右边结点的指针之间的搜索码值（在该例子中，该值为"Kim"）和指针一起移动，而位于留在左边的指针之间的搜索码值（在该例子中是"Califieri"和"Einstein"）依旧不变。

但是，位于留在左边的指针和移动到右边结点的指针之间的搜索码值要特殊对待。在该例子中，搜索码值"Gold"位于三个移动到左边结点的指针以及两个移动到右边结点的指针之间。因此"Gold"值不会添加到任意一个分裂结点中。相反，（Gold，$n2$）项添加到父结点中，$n2$ 指针指向分裂产生的新结点。在本例中，父结点就是根结点，并且有足够的空间来插入新的项。

往 B+ 树中进行插入的一般技术是确定插入发生的叶结点 l。如果产生分裂，则将新结点插入结点 l 的父结点中。如果这一插入导致分裂，就沿着树向上递归处理，直到不再产生分裂或创建一个新的根结点为止。

图 11-15 列出了插入算法的伪码。insert 过程用两个辅助过程 insert_in_leaf 和 insert_in_parent 将一个搜索码 – 值指针对插入到索引中。在伪码中，L、N、P、T 表示指向结点的指针，其中 L 代表叶结点。$L.K_i$ 和 $L.P_i$ 分别表示结点 L 中第 i 个值和第 i 个指针；$T.K_i$ 和 $T.P_i$ 也可同理使用。伪码还利用函数 *parent*(N) 来找出结点 N 的父结点。在一开始寻找叶结点时，我们可以计算从根到叶的路径上的结点列表，以后就可以利用它来高效地寻找这条路径中任何一个结点的父结点。

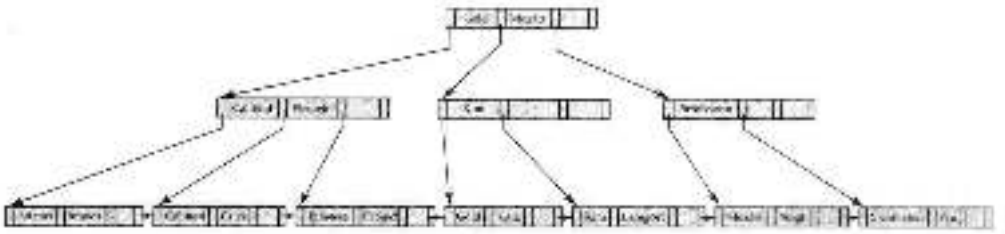


图 11-14　在图 11-9 的 B+树中插入"Lamport"

insert_in_parent 过程接收的参数为 N、K'、N'，其中结点 N 已分裂为 N 和 N'，K' 是 N' 中最小的值。该过程将修改 N 的父结点以记录可能进行的分裂。insert_into_index 和 insert_in_parent 过程都用一块临时内存区 T 来存储即将分裂的结点的内容。这些过程可以修改为直接把被分裂结点的内容复制到新建结点中，以减少数据复制的时间。然而，用临时内存区 T 可以简化这些过程。

```
procedure insert(value K, pointer P)
    if (树为空) 创建一个空叶结点 L，同时它也是根结点
    else 找到应该包含值 K 的叶结点 L
    if (L 已含搜索码少于 n-1 个)
        then insert_in_leaf (L, K, P)
        else begin  /* L 已经含有 n-1 个搜索码了，分裂 L */
            创建结点 L'
            把 L.P1, …, L.Kn-1 复制到可以存储 n 个(指针，搜索码值)对的内存块 T 中
            insert_in_leaf (T, K, P)
            令 L'.Pn = L.Pn；令 L.Pn = L'
            从 L 中删除 L.P1, …, L.Pn-1, L.K1, …, L.Pn
            把 T.P1, …, T.Kn/2 从 T 复制到 L 中，L 以 L.P1 作为开始
            把 T.Kn/2+1, …, T.Kn 从 T 复制到 L'中，L'以 L'.P1 作为开始
            令 K'表示 L'中最小搜索码值
            insert_in_parent (L, K', L')
        end

procedure insert_in_leaf (node L, value K, pointer P)
    if (K 比 L.K1 小)
        then 把 P、K 插入 L 中，紧接在 L.P 前面
        else begin
            令 K 表示 L 中小于等于 K 的最大搜索码值
            把 P、K 插入 L 中，紧接在 L.K 前面
        end
procedure insert_in_parent (node N, value K', pointer N')
    if (N 是树的根结点)
        then 在 L 上插入(P, P')
        else begin
            创建结点 R 包含 N、K'、N'  /* N 和 N'将是指针 */
            令 R 成为树的根结点
            return
        end
令 P = N 的父结点
if (P 包含的指针少于 n 个)
    then 将(K', N')插到 N 后面
    else begin /* 分裂 P */
            将 P 复制到可以存放 P 以及(K', N')的内存块 T 中
            创建结点(K', N')插入 T 中，紧接在 N 后面
            删除所有 P 中所有项；创建结点 P'
            把 T.P1, …, T.Kn/2 复制到 P 中
令 K' = T.Kn/2
把 T.Pn/2+1, …, T.Pn+1 复制到 P'中
insert_in_parent (P, K', P')
    end
```

图 11-15　在 B+树中插入索引项

11.3.3.2 删除

现在我们来考虑使树结点中指针变得过少的删除操作。首先我们从图 11-13 的 B+ 树中删除"Srinivasan"，产生的 B+ 树见图 11-16。接下来我们考虑删除操作是怎样执行的。我们首先用查找算法定位"Srinivasan"的索引项。当我们从叶结点中把"Srinivasan"的索引项删除后，叶结点就只剩下一个"Wu"项了。因此，在该例子中，由于 $n = 4$ 且 $1 < \lceil (n-1)/2 \rceil$，因此要么将该结点同一个兄弟结点合并，要么在结点间重新分配项，以此来保证每个结点至少半满。在该例子中，具有项"Wu"的太空的结点可以同它的左边兄弟结点合并。我们可通过把两个结点中的项移动到左边的兄弟结点中，并且删除现有的空右兄弟，从而合并结点。结点一旦删除，就同时也要删除父结点中指向刚刚删除结点的项。

在该例子中，(Srinivasan, n3)是要删除的项，其中 n3 是指向含有"Srinivasan"值的叶结点的指针。(在这种情况下，在非叶结点中将要删除的项和已经从叶结点中删除的值一样；当然，大部分删除情况不是这样的)。在删除上述项以后，含有搜索码值"Srinivasan"和两个指针的父结点现在变成含有一个指针(结点中最左边的指针)和没有搜索码值。因为对于 $n = 4$，$1 < \lceil n/2 \rceil$，所以父亲结点太空。(当 n 较大时，太空的结点将仍然含有一些值以及指针。)

在这种情况下，我们考虑兄弟结点，在该例子中，唯一的兄弟结点就是包含搜索码"Califieri"、"Einstein"和"Gold"的叶结点。如果可以，我们会尝试着同兄弟结点合并。但是在这种情况下，合并是不可能的。因为结点和它的兄弟结点一共拥有 5 个指针，超过了最大数量 4。解决这种情况的方法是在该结点和兄弟结点间重新分配(redistribute)指针，以使每个结点含有至少 $\lceil n/2 \rceil = 2$ 的孩子指针。方法，我们将来自左边兄弟结点(该指针指向含有"Mozart"的叶结点)的最右边的指针移动到太空的右边兄弟中。相反，太空的右边兄弟现在拥有两个指针，即最左边的指针和新移过来的指针，而且没有值将它们分开。事实上，将它们分开的值不会在任意一个结点中，而是在父结点中，位于从父结点指向该结点以及从父结点指向其兄弟结点的指针之间。在该例子中，"Mozart"值分开这两个指针，并且在重新分配后位于右边兄弟结点中。指针的重新分配也意味着父结点中"Mozart"值不再正确地分开两个兄弟结点中的搜索码值。事实上，现在正确分开两个兄弟结点中搜索码的值是"Gold"，它在重新分配之前在左边兄弟结点中。

492 ~ 495

如图 11-16 中的 B+ 树所示，在重新分配兄弟结点的指针后，"Gold"值移动到父结点中，而原先父结点中的"Mozart"值下移到右边的兄弟结点中。

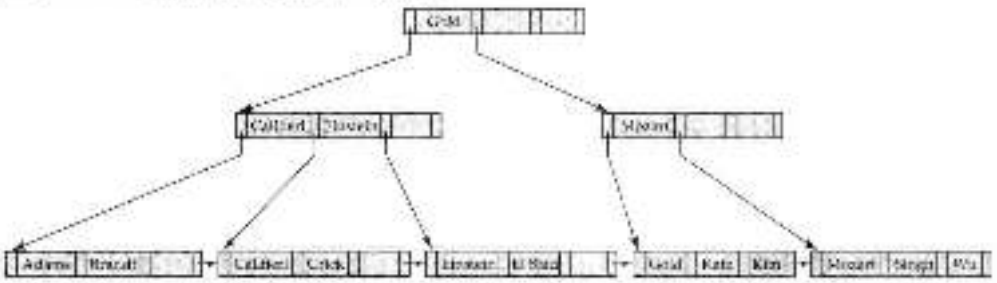


图 11-16 从图 11-13 删除 B+ 树中键值"Srinivasan"

接下来我们从图 11-16 的 B+ 树中删除搜索码值"Singh"和"Wu"。结果在图 11-17 中显示。其中第一个值的删除不会使叶结点太空，但是第二个值的删除会使叶子结点太空。太空的结点同它的兄弟结点不可能合并，因此需要执行值的重新分配。该分配将搜索码值"Kim"移动到包含"Mozart"的结点中，形成的树如图 11-17 所示。在父结点中分开两个兄弟结点的值改变了，从"Mozart"变成"Kim"。

现在我们考虑在上述树中删除"Gold"，该结果在图 11-18 中表示。该次删除导致了一个太空的叶结点，它可以同它的兄弟结点合并。父结点(包含"Kim"的非叶结点)项的删除致使父结点太空(该结点仅剩下一个指针)。这次父结点可以同它的兄弟结点合并。这次合并使得搜索码值"Gold"从父结点下移到合并结点中。作为合并的结果，从父结点中删除一项，而该父结点正好是树的根结点。该次删除使得根结点仅剩下一个孩子指针，没有搜索码值，违反了根结点至少有两个孩子的条件。因此，根结点被删除，其唯一的孩子结点成为根结点，并且 B+ 树的深度减 1。

值得一提的是，进行删除操作后，在 B+ 树的非叶结点中的关键码值可能在树的叶结点中并不存

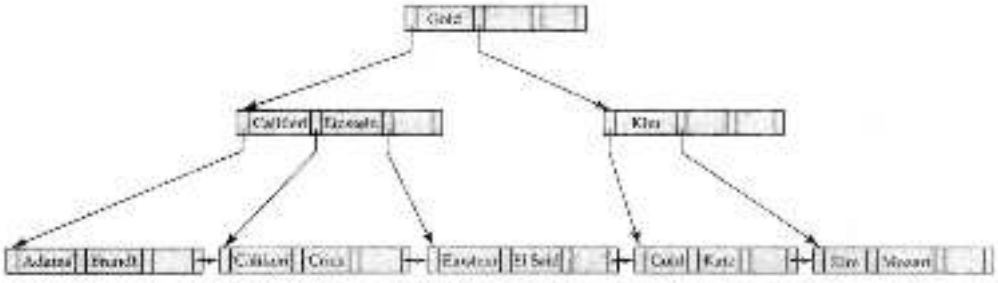


图 11-17 从图 11-16 所示 B+树中删除"Srinivasan"和"Wu"

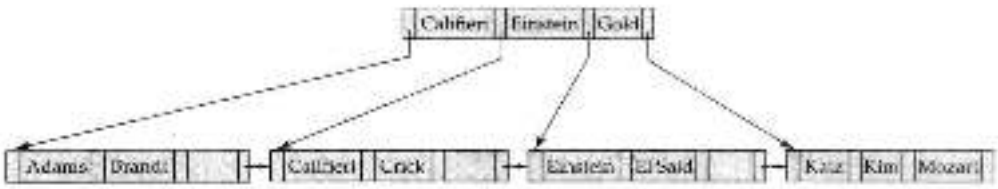


图 11-18 从图 11-17 所示 B+树中删除"Gold"

在。例如，在图 11-18 中，"Gold"值从叶结点中删除，但是它仍存在于非叶结点中。

一般地，为了在 B+树中删除一个值，我们要查找并删除该值。如果结点太小的话，我们从它的父结点中把它删除。这个删除导致删除算法的递归应用，直到到达树的根结点。删除后父结点要保持足够满，否则要进行重新分布。

图 11-19 给出了对 B+树运行删除的伪码。过程 swap_variables(N, N')仅仅交换两个(指针)变量 N 和 N'的值，交换对树本身毫无影响。伪码使用条件"指针/值太少"。对非叶结点，这个条件意味着少于$\lceil n/2 \rceil$个指针；对叶结点，这个条件意味着少于$\lceil (n-1)/2 \rceil$个值。伪码通过向相邻的结点借一个索引项来实现重新分布。我们也可以通过在两个结点间平分索引项来实现重新分布。伪码涉及从一个结点中删除索引项(K, P)。对叶结点而言，指向索引项的指针实际上在码值前面，因此指针 P 在码值 K 前面。而对于非叶结点，P 跟在码值 V 的后面。

11.3.4 不唯一的搜索码

假如一个关系可以拥有多个包含同一搜索码值的记录(换句话说，两条或者多条记录在索引属性上拥有相同的值)，那么该搜索码称为**不唯一搜索码**(nonunique search key)。

不唯一搜索码的一个问题在于记录删除效率方面。假设某个特殊的搜索码值出现很多次，并且拥有该搜索码值的一条记录将要被删除。那么删除操作可能会查找很多项，从而找出和该被删记录相对应的项，该搜索有可能遍历多个叶结点。

一种简单的解决方法是通过创建包含原始搜索码和其他属性的复合搜索码来确保搜索码唯一。对于所有记录，该复合搜索码是唯一的，这通常也被大多数数据库系统使用。这个额外的属性可以是指向记录的指针——记录号，或者是任何一个在拥有相同搜索码值的所有记录中值唯一的其他属性。这个额外属性也称为**唯一化**(uniquifier)属性。当要删除一条记录时，从记录中计算复合搜索码值，然后再从索引查找。因为值是唯一的，所以相应的叶子级项可以通过根结点到叶子结点的一次周游就可以找到，并且不需要进一步访问叶子级。因此，可以有效地删除记录。 496 ~ 497

当比较搜索码值时，具有原始搜索码值的查找可以简单地忽略唯一化属性值。

对于不唯一搜索码，一个码值在多少条记录中出现，B+树结构就存储该码值多少次。另外一种方法是在该树中每个码值只存储一次，并且为该搜索码值维护一个记录指针的桶(或者列表)来解决不唯一搜索码问题。因为它只存储码值一次，所以这种方法在空间上更有效，但是当 B+树实现时，它要带来较多的复杂性。如果在叶结点上保存桶，需要额外的代码来处理可变长的桶，并且处理当桶增长到比叶结点还大的情况。如果这些桶存储在单独的块中，在获取记录时可能需要额外的 I/O 操作。此外，

如果搜索码值出现次数较多，所方案也会带来记录删除效率不高的问题。

11.3.5 B+树更新的复杂性

尽管 B 树的插入和删除操作比较复杂，但它们需要较少的 I/O 操作，这很有价值，因为 I/O 操作比较费时。在最坏情况下，一次插入的 I/O 操作次数可能正比于 $\log_{\lceil n/2 \rceil}(N)$，其中 n 是结点中指针的最大值，N 是索引文件中的记录条数。

```
procedure delete (value K, pointer P)
    找到包含(K, P)的叶结点 L
    delete_entry(L, K, P)

procedure delete_entry(node N, value K, pointer P)
    从 N 中删除(K, P)
    if (N 是根结点 and N 只剩下一个子结点)
    then 使 N 的子结点成为该树的新的根结点并删除 N
    else if (N 中值/指针太少) then begin
        令 N'为 parent(N)的前一个或后一个子女
        令 K'为 parent(N)中指针 N 和 N'之间的值
        if (N 和 N'中的项能放在一个结点中)
        then begin  /* 合并结点 */
            if (N 是 N'的前一个结点) then swap_variables(N, N')
            if (N 不是叶结点)
                then 将 K'以及 N 中所有指针和值加到 N'中
                else 将 N 中所有(K_i, P_i)对加到 N'中; 令 N'.P_n = N.P_n
            delete_entry (parent(N), K', N); 删除结点 N
        end
    else begin /* 重新分布: 从 N'借一个索引项 */
        if (N'是 N 的前一个结点) then begin
            if (N 是非叶结点) then begin
                令 m 满足: N'.P_m 是 N'的最后一个指针
                从 N'中去除(N'.K_m-1, N'.P_m)
                插入(N'.P_m, K')并通过将其他指针和值右移使之成为
                        N 中的第一个指针和值
                用 N'.K_m-1 替换 parent(N)中的 K'
            end
            else begin
                令 m 满足: (N'.P_m, N'.K_m)是 N'中的最后一个指针/值对
                从 N'中去除(N'.P_m, N'.K_m)
                插入(N'.P_m, N'.K_m)并通过将其他指针和值右移使之成为
                        N 中的第一个指针和值
                用 N'.K_m 替换 parent(N)中的 K'
            end
        end
        else  …与 then 的情况对称…
    end
  end
end procedure
```

图 11-19 从 B+树中删除索引项

如果搜索码上没有相同的值，最坏情况下删除过程的复杂度同样正比于 $\log_{\lceil n/2 \rceil}(N)$。如果存在相同的值，删除操作须要在多个含有相同搜索码值的记录中进行查找，直到找到要删除的正确的项。这样的效率是较低的。但是，如 11.3.4 节所述，即便原始搜索码是不唯一，也通过增加唯一化属性可以使搜索码唯一，从而保证了在最坏情况下删除操作的复杂性也是一样的。

换句话说，以 I/O 操作来衡量的插入和删除操作的代价是和 B⁺ 树的高度成正比的，因此是比较低的。B⁺ 树的操作速度使得它在数据库实现中成为频繁使用的索引结构。

在实践中，B⁺ 树上的操作所导致的 I/O 操作要比最坏边界少。假设扇出为 100，并且假设对叶结点的访问是均匀分布的，那么叶结点的父结点被访问的次数可能是叶结点的 100 倍。相反地，对于相同的扇出，B⁺ 树中所有非叶结点的总数可能仅比叶子结点的 1/100 多一点。因此，在现在通常内存大 498
小可达几 GB 的情况下，对于频繁使用的 B⁺ 树，即便关系非常庞大，大部分非叶结点在被访问时已经 499
在数据库缓冲区中。因此，一次查询通常仅需要一次或两次 I/O 操作。在更新方面，叶结点分裂的可能性比较小。在扇出为 100 的情况下，100 次插入中仅有 1 次或 50 次插入中仅有 1 次会导致结点分裂，从而导致多个块被写入，这取决于插入的顺序。因此，平均情况下，一次插入需要更新块的 I/O 操作仅比一次略多。

尽管 B⁺ 树仅保证了结点至少半满，但是，如果项是以随机顺序插入，那么结点在平均情况下会比 2/3 更满。另一方面，如果项是以有序方式插入，结点将会仅仅半满。（我们将此留作后面的联系，让读者来计算为什么结点会仅仅半满。）

11.4　B⁺ 树扩展

本节讨论几个 B⁺ 树索引结构的扩展和变种。

11.4.1　B⁺ 树文件组织

在 11.3 节中提到，索引顺序文件组织最大的缺点是文件增大时性能下降；随着文件的增大，增加的索引记录所占百分比和实际记录之间变得不协同，不得不存储在溢出块中。我们通过在文件上使用 B⁺ 树索引来解决索引查找时性能下降的问题。通过用 B⁺ 树的叶级结点来组织存放实际记录的磁盘块，我们可以解决存储实际记录的性能下降问题。我们不仅把 B⁺ 树结构作为索引使用，而且把它作为一个文件中的记录的组织者。在 B⁺ 树文件组织（B⁺ – tree file organization）中，树的叶结点存储的是记录而不是指向记录的指针。图 11-20 给出了一个 B⁺ 树文件组织的例子。由于记录通常比指针大，因此一个叶结点中能存储的记录数目要比一个非叶结点中能存储的指针数目少。然而，叶结点仍然要求至少是半满的。

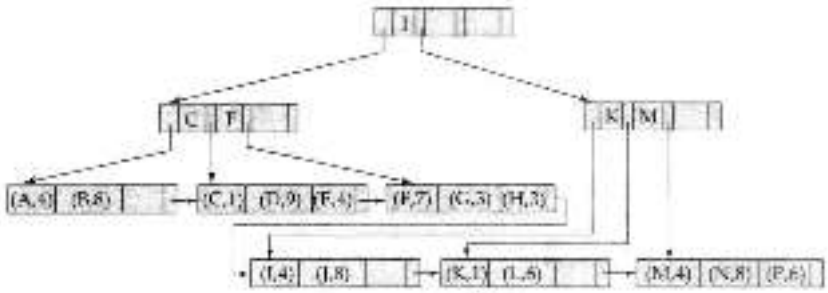


图 11-20　B⁺ 文件结构

B⁺ 树文件组织中记录的插入和删除与 B⁺ 树索引中索引项的插入和删除的处理方式一样。当插入 500
一条具有给定码值 v 的记录时，系统通过在 B⁺ 树中查找小于等于 v 的最大码来定位应该包含该记录的块。如果定位到的块有足够的空间来存放记录，系统就将该记录存放在该块中。否则，就像 B⁺ 树的插入那样，系统将该块分成两个，重新分布其中的记录（按 B⁺ 树码值的顺序），以给新记录创建空间。这种分裂以通常的形式在 B⁺ 树中自底向上传播。当删除一个记录时，系统首先从包含它的块中将它删除。如果块 B 因此不到半满，就要对 B 中的记录和相邻的块 B′ 中的记录重新分布。假定记录大小固

定，每个块包含的记录数至少应是它所能包含的最大记录数的一半。系统按照通常的方式更新 B+ 树的非叶结点。

当我们用 B+ 树作为文件组织方式时，空间的利用尤为重要，因为记录所占的空间很可能比码和指针所占的空间要大得多。通过在分裂和合并时在重新分布中涉及更多的兄弟结点，我们可以改善 B+ 树的空间利用率。这个技术对叶结点和非叶结点都可以用，具体方法如下所述：

在插入时，如果某个结点已满，系统就尝试把它的一些项重新分布到与它相邻的结点中，以给新项腾出空间。如果因为相邻结点本身已满而导致这种尝试失败，系统就要分裂该结点，并在一个相邻结点和分裂原始结点得到的两个结点这三者之间均匀地分配所有的项。由于这三个结点总共包含的项比两个结点所能容纳的多 1 个，因此每个结点大约为 2/3 满的。更确切地说，每个结点至少有 $\lfloor 2n/3 \rfloor$ 个项，其中 n 是每个结点能容纳的最大项数。（$\lfloor x \rfloor$ 表示小于或等于 x 的最大整数，也就是忽略它的小数部分（如果有的话）。）

在删除记录时，如果结点中的项数少于 $\lceil 2n/3 \rceil$，系统就试图从相邻的结点借入一项。如果两个兄弟结点都有 $\lceil 2n/3 \rceil$ 条记录，那么系统就不是借入一项，而是把这个结点及两个兄弟结点中的所有项均匀地分布到两个结点中，并删除第三个结点。我们能用这种方法是因为项的总数是 $3\lfloor 2n/3 \rfloor - 1$，它小于 $2n$。当使用三个相邻结点进行重新分布时，每个结点能保证有 $\lceil 3n/4 \rceil$ 个项。一般地，如果重分布涉及 m 个结点（$m-1$ 个兄弟结点），每个结点能保证包含至少 $\lfloor (m-1)n/m \rfloor$ 个索引项。然而，参与重新分布的兄弟结点越多，更新的代价也越高。

注意在 B+ 树索引或 B+ 树文件组织中，树中相邻的叶结点可能分布在磁盘中的不同位置。当一个文件组织最初建立在一组记录上的时候，可以实现把磁盘中基本连续的块分配给树中连续的叶结点。由此对叶结点的顺序扫描就相当于磁盘上几乎是顺序的扫描。随着在树中不断进行插入和删除操作，
501 这种顺序性逐渐丢失，对叶结点的顺序访问也需要越来越频繁地等待磁盘寻道。为了恢复这种顺序性也许需要对索引进行重建。

B+ 树文件组织可以用于存储大型数据对象，如 SQL clob 类型以及 blob 类型，这些数据对象可能比磁盘块还大，甚至达到好几个 GB。这么大的数据对象可以通过拆分成稍小的记录序列并组织成 B+ 树文件组织进行存储。拆分的记录可以按序编号，或者根据记录在大型对象中的字节偏移量进行编号，这样记录的编号就可以用作搜索码。

11.4.2 辅助索引和记录重定位

一些文件组织（如 B+ 树文件组织）可能改变记录的位置，即使记录并没有更新。举例来说，当 B+ 树文件组织中的一个叶结点分裂，一些记录会移动到新的结点中。在这种情况下，所有存储了那些指向重定位过的记录的指针的辅助索引都必须更新，即使记录中的值并没有改变。每个叶结点可能包含相当多的记录，而其中每条记录都可能在每个辅助索引中的不同位置。因此一个叶结点的分裂可能需要几十甚至几百次 I/O 操作来更新所有影响到的辅助索引，导致这个操作代价极其高昂。

为解决这个问题广泛使用的方法如下。在辅助索引中，不存储指向被索引的记录的指针，而是存储主索引搜索码属性的值。例如，假设我们有一个建立在 *instructor* 关系的 *ID* 属性上的主索引；还有一个建立在 *dept_name* 上的辅助索引，这个辅助索引中与每个系的名字存放在一起的是相应记录中 *ID* 值构成的列表，而不是指向这些记录的指针。

于是，由于叶结点分裂导致的记录的重定位就不需要对这样的辅助索引进行更新了。然而，用辅助索引定位一条记录现在就需要两步：首先用辅助索引找到主索引搜索码的值，然后用主索引来找到对应的记录。

上述方法大大降低了由文件重组导致的索引更新的代价，尽管它也增加了使用辅助索引访问数据的代价。

11.4.3 字符串上的索引

在字符串属性上创建 B+ 树索引会引起两个问题。第一个问题是字符串是变长的。第二个问题是字符串可能会很长，导致结点扇出降低以及相应地增加树的高度。

对于变长搜索码，即使结点都是满的，不同的结点也可能会有不同的扇出。如果一个结点已满，则它必须分裂，也就是说这个结点已经没有空间可以容纳新的搜索项，不管原来它有多少项。同样地，结点的合并和重新分布取决于结点中空间使用的比例，而不是根据结点所能容纳的最大项数。 502

使用**前缀压缩**(prefix compression)技术可以增加结点的扇出。使用前缀压缩技术，不用在非叶结点存储整个搜索码值。只须存储每个搜索码值的一个前缀，使得这个前缀足以将由该搜索码值分开的两棵子树中的码值区分开。例如，假如我们有一个建立在名字上的索引，非叶结点的码值可以是名字的一个前缀：如果由搜索码分开的两棵子树中跟"Silberschatz"最相近的码值分别是"Silas"和"Silver"，则在非叶结点中存储"Silb"就足够了，而不用存储全名"Silberschatz"。

11.4.4 B⁺树索引的批量加载

如我们先前看到的，在B⁺树中插入一条记录需要一些I/O操作，并且在最坏情况下与树的高度成正比，这通常还是比较小的(即便对于比较大的关系，一般为5次或者更少)。

现在考虑在大关系上建立B⁺树的情况。假设关系要比主存大很多，并且我们在关系上构建非聚集索引，使得该索引也比主存要大。在这种情况下，当扫描关系并且往B⁺树中添加项时，要访问的每个叶结点通常不会在数据库缓冲区中，因为项没有特定的排序。在这种块的随机访问中，每次在叶结点中添加一个项时，需要一次磁盘寻道来取回包含叶结点的块。当另一个项添加到块中，该块有可能会被盘缓冲区中剔出，导致另一次磁盘寻道将块回写到磁盘中。每次项的插入可能都需要这样一次随机读和随机写操作。

例如，如果一个关系含有1亿条记录，每次I/O操作需要10毫秒，那么它将至少需要100万秒的时间来创建索引，这仅仅是读叶结点的开销，还不算将节点更新到磁盘的写操作开销。很明显这时间开销非常大，相比，如果每条记录占用100字节，磁盘子系统可以以每秒50MB的速度传输数据，那么仅需要200秒的时间来读取整个关系。

将大量项一次插入到索引中称为索引的**批量加载**(bulk loading)。一种有效执行索引批量加载的方式如下：首先，创建一个含有关系索引项的临时文件，然后根据构建好的索引的搜索码来排序文件。最后扫描排序好的文件并且将项插入到索引中。目前存在对于大关系的排序算法，这将会在12.4节描述。假设主存容量够，即便对于一个很大的文件进行排序，该算法需要的I/O代价与读几次文件相当。

在将项插入到B⁺树之前进行排序具有明显的好处。在项按排序进行插入后，所有到特定叶结点的项将会连续出现，并且叶结点只需要写出一次。如果B⁺树开始时为空，在批量加载时结点就不需要从磁盘中读取。因此，即便许多项要插入到一个结点中，每个叶结点也只需要一次I/O操作。如果每个 503
叶结点包括100个项，叶子级将包括100万个结点，那么创建叶级只需100万次I/O操作。如果连续的叶结点被分配到连续的磁盘块上，那么这些I/O操作也可以是顺序的，并且需要很少的磁盘寻道。就当前的磁盘，相比于随机I/O操作的每块10毫秒，大部分顺序I/O操作估计只需要每块1毫秒。

我们稍后在12.4节中研究大关系排序的代价，但是粗略地估计，通过在插入B⁺树中前对项进行排序，索引可以在1000秒就构建好，与之相对比，采用随机插入则需要1 000 000秒的时间来构建。

如果B⁺树初始时是空的，那么就可以从叶子级自底向上来快速构建它，而不是使用常规的插入过程。在**自底向上B⁺树构建**(bottom-up B⁺-tree construction)中，通过我们所描述的对项进行排序后，我们将排序好的项分解到块中，并保证每个块中有尽可能多的项，由此产生的块形成B⁺树的叶级。每个块中的最小值以及指向块的指针用来构建下一级的B⁺树项，并且指向叶块。更深一级的树可以类似地利用下层每个结点中的最小值来构建，直到创建根结点。我们把这些细节留给读者作为练习。

在一个关系上创建索引时，大多数数据库系统实现了基于项排序和自底向上构建的有效技术。尽管在向有索引的关系上一次插入一个元组时，它们使用普通的插入程序。如果一次添加非常多的元组到已经存在的关系中，一些数据库系统会建议在该关系上的索引(除主码上的索引)应该删除，并且在插入元组后重新构建该索引，来有效地利用批量加载技术。

11.4.5 B树索引文件

B树索引(B-tree index)和B⁺树索引相似。两种方法的主要区别在于B树去除了搜索码值存储中的

冗余。在图 11-13 的 B⁺树中，搜索码"Califieri"、"Einstein"、"Gold"、"Mozart"和"Srinivasan"在非叶结点和叶结点中均出现。每个搜索码值都出现在某些叶结点中，有的还在非叶结点中重复出现。

在 B⁺树中，搜索码值可能同时出现在非叶结点和叶结点中。与 B⁺树不同，B 树只允许搜索码值出现一次（如果它们是唯一的）。图 11-21 所示 B 树表示与图 11-13 中的 B⁺树相同的搜索码值。由于 B
504 树中的搜索码不重复，因此可以用比相应 B⁺树索引更少的树结点来存储索引。然而，在 B 树中，由于出现在非叶结点中的搜索码值不会出现在其他地方，因此我们将不得不在非叶结点中为每个搜索码增加一个指针域。附加的这些指针指向文件记录或相应搜索码所对应的桶。

值得注意的是，许多数据库系统手册、行业文献文章以及行业专家使用术语 B 树来指代这里所述的 B⁺树数据结构。事实上，在当前的用法中这样定义也是合理的，因为术语 B 树和 B⁺树是同义词。尽管如此，在这本书中我们使用 B 树和 B⁺树原本的定义，来避免两种数据结构的混淆。

B 树叶结点一般的形式如图 11-22a 所示；非叶结点如图 11-22b 所示。叶结点和 B⁺树中的叶结点一样。在非叶结点中，指针 P_i 与 B⁺树中使用的树指针一样，而指针 B_i 是桶或文件记录的指针。在图 11-22 的一般化的 B 树中，叶结点中有 $n-1$ 个码，而非叶结点中有 $m-1$ 个码。这个差异的出现是因为非叶结点必须包含指针 B_i。这样就减少了这些结点所能容纳的搜索码个数。显然 $m<n$，但 m 和 n 的确切关系依赖于搜索码和指针的大小。

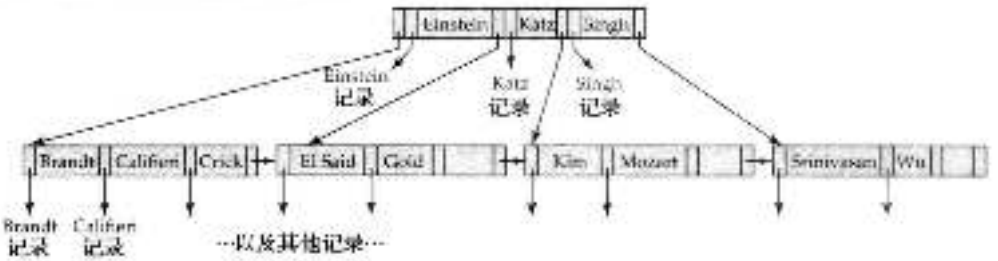


图 11-21 等价于图 11-13 中 B⁺树的 B 树



a) 叶结点



b) 非叶结点

图 11-22 典型的 B 树结点

在 B 树中进行一次查找所访问的结点数取决于搜索码所在的位置。对 B⁺树的查找需要遍历从树根到叶结点的路径。与此相比，B 树中有时不需到达叶结点就能找到想要的值。然而，B 树中存储在叶
505 级的码大约 n 倍于存储在非叶级的码，而 n 一般都相当大，因此较快找到特定值的好处将会比较小。而且比起 B⁺树来，B 树的非叶结点中存储的搜索码较少，这一事实意味着 B 树需出较小因而比相应的 B⁺树深度大。因此，在 B 树中查询某些搜索码比 B⁺树中快，而另一些值比 B⁺树慢，虽然总的来说查找时间仍然与搜索码数目取对数成正比。

B 树中的删除更加复杂。在 B⁺树中，被删除的项总是出现在叶结点中。在 B 树中，被删除的项可能出现在非叶结点中。我们必须从包含被删除项的子树中选择正确的值来作为替代。具体来说，如果搜索码 K_i 被删除，出现在指针 P_{i+1} 指向的子树中的最小搜索码必须移到原先 K_i 所占用的字段中。如果叶结点包含的项太少的话还需采取进一步的操作。与此相比，B 树中的插入只比 B⁺树中的插入略复杂一些。

B 树在空间上的优势对大的索引来讲意义不大，通常不能抵消我们已经到的那些不足。因此，许多数据库系统实现采用 B⁺树数据结构，即使（我们先前讨论过）他们将此数据结构称为 B 树。

11.4.6　闪存

到目前为止，我们对于索引的描述都是假设数据是存放在磁盘上的。尽管这种假设在大部分情况是真的，但是闪存容量急速增长，并且每 GB 的闪存价格日益下跌，使得许多应用中闪存存储成为替代磁盘存储的强大竞争者。一个自然的问题是，这种变化会怎样影响索引结构。

闪存存储是通过块来组织的，并且 B⁺ 树索引结构可以在闪存存储中使用。这为索引查询中加快访问速度带来的好处显而易见。相比于平均需要 10 毫秒来寻道和读块，闪存中的随机块读取在微秒级完成。因此与基于磁盘的数据查找要快得多。闪存中最合适的 B⁺ 树结点大小通常要比磁盘小得多。

闪存的唯一缺点就是它不允许物理层的数据就地更新，尽管它可以在逻辑上实现。每次更新操作转换成整个闪存块的一次拷贝加上写操作，要求块的旧拷贝随后清除，一个块的清除需要 1 毫秒。针对开发可以降低块清除次数的索引结构的研究正在进行。同时，标准的 B⁺ 树索引可以继续在闪存存储中使用，提供可以接受的更新性能，以及比磁盘存储更有效的查询性能。

11.5　多码访问

到现在为止，我们都隐含地假设只使用建立在一个属性上的一个索引来执行关系上的查询。但是对于某些类型的查询来说，如果存在多个索引则使用多个索引，或者使用建立在多属性搜索码上的索引，这样比较有利。 506

11.5.1　使用多个单码索引

假设 *instructor* 文件有两个索引，分别建立在 *dept_name* 和 *salary* 上。考虑如下查询："找出金融系中工资为 $80 000 的所有教师。"我们写作

```
select  ID
from    instructor
where   dept_name = 'Finance' and salary = 80000
```

处理这个查询可以有三种策略：

1. 利用 *dept_name* 上的索引，找出属于金融系的所有记录。检查每条记录是否满足 *salary* = 80 000。

2. 利用 *salary* 上的索引，找出所有工资等于 $80 000 的记录。检查每条记录是否满足 *dept_name* = 'Finance'。

3. 利用 *dept_name* 上的索引找出指向属于金融系的记录的所有指针。同样，利用 *salary* 上的索引找出指向工资等于 $80 000 的记录的所有指针。计算这两个指针集合的交。交集中的那些指针指向金融系中工资等于 $80 000 的记录。

上面三种策略中只有第三种利用了存在的多个索引的优势。然而，如果下面所有条件都成立，即使这种策略也可能是很糟糕的选择：

- 属于金融系的记录太多。
- 余额为 $80 000 的记录太多。
- 金融系中工资为 $80 000 的记录只有几个。

如果这些条件成立的话，为了得到一个很小的结果集，我们必须扫描大量指针。一种称为"位图索引"的索引结构在某些情况下可以加速第三种策略中使用的集合交的操作。位图索引将在 11.9 节介绍。 507

11.5.2　多码索引

这种情况下另一个可选的策略是在复合的搜索码（*dept_name*，*salary*）上建立和使用索引。也就是说，这一搜索码由系名和教师工资连接而成。

我们可以使用在上述复合的搜索码上的顺序（B⁺ 树）索引来高效地回答具有如下形式的查询：

```
select  ID
from    instructor
where   dept_name = 'Finance' and salary = 80000
```

在搜索码的第一个属性（*dept_name*）上指定一个等值条件以及在搜索码的第二个属性（*salary*）上指

定一个范围。这种形式的查询也能高效地处理，因为它们对应于搜索属性上的一个范围查询。

```
select  ID
from    instructor
where   dept_name = 'Finance' and salary < 80000
```

我们甚至可以使用搜索码(*dept_name*, *salary*)上的顺序索引来高效回答下面这种只涉及一个属性的查询。

```
select  ID
from    instructor
where   dept_name = 'Finance'
```

等值条件 *dept_name* = "Finance" 等价于一个范围查询，该范围查询的下界是(Finance, $-\infty$)，上界是(Finance, $+\infty$)。仅在 *dept_name* 属性上的范围查询也可以同样处理。

然而，使用建立在一个复合搜索码上的顺序索引结构是有一定的缺点的。作为示例，考虑以下的查询

```
select  ID
from    instructor
where   dept_name < 'Finance' and salary < 80000
```

我们可以使用建立在搜索码(*dept_name*, *salary*)上的顺序索引来回答这个查询：对于按字母顺序小于"Finance"的每个 *dept_name* 值，系统定位 *salary* 值为 80000 的那些记录。然而，由于文件中记录的顺序，每条记录可能位于不同的磁盘块，因此导致大量 I/O 操作。

[508] 这个查询和前面两个查询的区别在于第一个属性(*dept_name*)上的条件是比较条件而不是等值条件。这个条件不能对应于搜索码上的一个范围查询。

为了加速处理一般的复合搜索码的查询(可以有一个或多个比较操作)，我们可以使用若干特殊的结构。我们将在 11.9 节中考虑位图索引。还有一种称为 R 树的结构也可用于这一目的。R 树是 B$^+$ 树的扩展，用于处理在多个维上的索引。因为 R 树结构主要用于地理数据类型，所以我们将在第 25 章描述其结构。

11.5.3 覆盖索引

覆盖索引(covering index)存储一些属性(但不是搜索码属性)的值以及指向记录的指针。存储附加的属性值对于辅助索引是非常有用的，因为它们使我们仅仅使用索引就能够回答一些查询，甚至不需要找到实际的记录。

例如，假设我们有一个建立在 *instructor* 关系的 ID 属性上的非聚集索引。如果我们把 salary 属性的值与记录指针一起存储，我们就可以回答那些要求查 *salary* 值(但不是另一个属性 *dept_name*)的查询而不需要访问 *instructor* 记录。

在搜索码(*ID*, *salary*)上创建索引能够达到同样的效果，然而一个覆盖索引能够减小搜索码的大小，使非叶结点中有更大的扇出，从而潜在地降低索引的高度。

11.6 静态散列

顺序文件组织的一个缺点是我们必须访问索引结构来定位数据，或者必须使用二分法搜索，这将导致过多的 I/O 操作。基于**散列**(hashing)技术的文件组织使我们能够避免访问索引结构。散列也提供了一种构造索引的方法。在接下来的几节中，我们将学习基于散列的文件组织和索引。

在对散列的描述中，我们将使用术语**桶**(bucket)来表示能存储一条或多条记录的一个存储单位。通常一个桶就是一个磁盘块，但也可能小于或大于一个磁盘块。

正规地说，令 K 表示所有搜索码值的集合，令 B 表示所有桶地址的集合，**散列函数**(hash function) h 是一个从 K 到 B 的函数。我们用 h 表示散列函数。

为了插入一条搜索码为 K_i 的记录，我们计算 $h(K_i)$，它给出了存放该记录的桶的地址。我们目前假定桶中有容纳这条记录的空间。于是这条记录就存储到该桶中。

为了进行一次基于搜索码值 K_i 的查找，我们只需计算 $h(K_i)$，然后搜索具有该地址的桶。假定两

个搜索码 K_i 和 K_j 有相同的散列值，即 $h(K_i) = h(K_j)$。如果我们执行对 K_i 的查找，则桶 $h(K_i)$ 包含搜索码值是 K_i 以及 K_j 的记录。因此，我们必须检查桶中每条记录的搜索码值，以确定该记录是否是 [509]
我们要查找的记录。

删除也一样简单。如果待删除记录的搜索码值是 K_i，则计算 $h(K_i)$，然后在相应的桶中查找此记录并从中删除它。

散列可以用于两个不同的目的。在**散列文件组织**(hash file organization)中，我们通过计算所需记录搜索码值上的一个函数直接获得包含该记录的磁盘块地址。在**散列索引组织**(hash index organization)中，我们把搜索码以及与它们相关联的指针组织成一个散列文件结构。

11.6.1 散列函数

最坏的可能是散列函数把所有的搜索码值映射到同一桶中。这种函数并不是我们所期望的，因为所有的记录不得不存放到同一个桶里。查找一个需要的记录时将不得不检查所有记录。理想的散列函数把存储的码均匀地分布到所有桶中，使每个桶含有相同数目的记录。

由于设计时我们无法精确知道文件中将存储哪些搜索码值，因此我们希望选择一个把搜索码值分配到桶中并且具有下列分布特性的散列函数。

- 分布是均匀的。即散列函数从所有可能的搜索码值集合中为每个桶分配同样数量的搜索码值。
- 分布是随机的。即在一般情况下，不管搜索码值实际怎样分布，每个桶应分配到的搜索码值数目几乎相同。更确切地说，散列值不应与搜索码的任何外部可见的排序相关，例如字符的顺序或搜索码长度的顺序。散列函数应该表现为随机的。

为了阐明上述原则，我们为以 *dept_name* 为搜索码的 *instructor* 文件选择一个散列函数。我们选择的散列函数不仅要在我们一直使用的 *instructor* 文件上具有所需的特性，而且在拥有许多系的大型大学中的真实的 *instructor* 文件上也应如此。

假设我们决定使用 26 个桶，并且我们定义的散列函数将名字每是字母表中第 i 个字母的名字映射到第 i 个桶上。该散列函数虽具有简单这一优点，但是无法提供均匀分布。因为我们可以预见，例如，以 B 和 R 这样的字母开头的名字会比用 Q 和 X 之类的字母开头的要多。

现在假设我们想要搜索码 *salary* 上的一个散列函数。假设最低工资是 $30 000，最高工资是 $130 000。我们可以使用一个散列函数把这些值分成 10 个区间：$30 000 ~ $40 000、$40 001 ~ $50 000等。对于该函数，搜索码值的分布是均匀的(因为每个桶具有相同数目的不同 *salary* 值)，但不 [510]
是随机的。然而，工资在 $60 001 ~ $70 000 之间的记录比在 $30 001 ~ $40 000 之间的记录要普通得多。其结果是，记录的分布是不均匀的，某些桶得到的记录比其他桶多。如果该函数分布随机，即使搜索码中有这种相关性，分布的随机性也将使所有桶拥有大致相等的记录数，只要每个搜索码都仅对应一小部分记录。(如果记录中的大部分都对应同一个搜索码，那么不管使用何种散列函数，包含该搜索码的桶都会比其他桶拥有更多的记录。)

通常散列函数在搜索码中字符的内部二进制机器表示上执行计算。这种类型的一个简单函数是先计算码中字符的二进制表示的总和，然后返回该总和取桶数目的模。

图 11-23 给出了该方案的一个应用，它作用于 *instructor* 文件，具有 8 个桶，并假设字母表中的第 i 个字母用整数 i 表示。

下面的散列函数是散列字符串的一个较好选择。设 s 是长度为 n 的字符串，$s[i]$ 是字符串的第 i 个字节。散列函数如下定义：

$$s[0] \cdot 31^{(n-1)} + s[1] \cdot 31^{(n-2)} + \cdots + s[n-1] \qquad [511]$$

这个函数可以按如下方式高效实现：最初把散列函数值设为 0，然后从字符串的第一个字符开始迭代，直到最后一个字符为止，每一步迭代都把散列函数值乘以 31 再加上下一个字符的值(把字符看作整数)。上述表示似乎可以导致一个很大的数，但是实际上计算后是一个固定大小的正整数；每次乘法运算和除法运算自动模可能的最大整数值加 1 后运算得到。上述函数的结果再取桶数目的模得到的值就可以用作索引。

桶0

桶1

15151	Mozart	Music	40000

桶2

32343	El Said	History	60000
58583	Califieri	History	60000

桶3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

桶4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

桶5

76766	Crick	Biology	72000

桶6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

桶7

图 11-23 使用 *dept_name* 作为码的 *instructor* 文件的散列组织

散列函数的设计需要认真仔细。一个糟糕的散列函数可能导致查找所花费的时间与文件中搜索码数目成正比；一个设计良好的散列函数一般情况下查找所花费时间是一个（较小的）常数，而与文件中搜索码的个数无关。

11.6.2 桶溢出处理

到目前为止，我们一直假设，当插入一条记录时，记录映射到的桶有存储记录的空间。如果桶没有足够的空间，就会发生**桶溢出**（bucket overflow）。桶溢出的发生可能有以下几个原因：

- **桶不足。**桶数目（用 n_b 表示）的选择必须使 $n_b > n/f$，其中 n 表示将要存储的记录总数，f 表示一个桶中能存放的记录数目。当然，这种表示是以在选择散列函数时记录总数已知为前提的。
- **偏斜。**某些桶分配到的记录比其他桶多，所以即使其他桶仍有空间，某个桶也仍可能溢出。这种情况称为桶**偏斜**（skew）。偏斜发生的原因有两个：
  1. 多条记录可能具有相同的搜索码。
  2. 所选的散列函数可能会造成搜索码的分布不均。

为了减少桶溢出的可能性，桶的数目选为 $(n/f) \times (1+d)$，其中 d 是溢让因子，其典型值约为 0.2。有一些空间会浪费：桶中大约 20% 的空间是空的。但好处是减少了溢出的可能性。

尽管分配的桶比所需的桶多一些，但是桶溢出还是可能发生。我们用**溢出桶**（overflow bucket）来处理桶溢出问题。如果一条记录必须插入桶 b 中，而桶 b 已满，系统会为桶 b 提供一个溢出桶，并将此记录插入到这个溢出桶中。如果溢出桶也满了，系统会提供另一个溢出桶，如此继续下去。一个给定桶的所有溢出桶用一个链接列表链接在一起，如图 11-24 所示。使用这种链接列表的溢出处理称为**溢出链**（overflow chaining）。

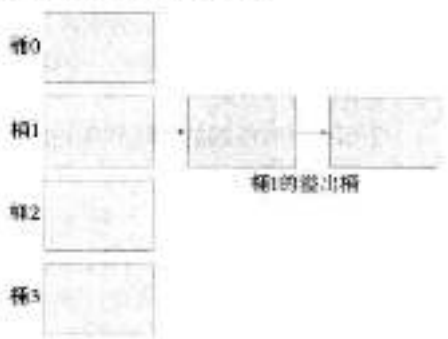


图 11-24 散列结构中的溢出链

为了处理溢出桶，我们需要将查找算法做轻微的改动。和前面一样，系统使用搜索码上的散列函数来确定一个桶 b。同前面一样，系统必须检查桶 b 中的所有记录，看是否

有匹配的搜索码。此外，如果桶 b 有溢出桶，则系统还要检查桶 b 的所有溢出桶中的记录。

我们刚刚描述的这种形式的散列结构称为**闭地址**(close addressing)，较少称为**闭散列**(closed hashing)。另一种方法称为**开地址**(open addressing)，它的桶集合是固定的，没有溢出链。它与闭地址不同，当一个桶满了以后，系统将记录插入到初始桶集合 B 的其他桶中。一种策略是使用下一个(按轮转顺序)有空间的桶，这个策略称为**线性探查法**(linear probing)。我们还可以使用其他策略，比如计算更多的散列函数的方法。开地址曾用于构造编译器和汇编器的符号表，而闭地址更适合用于数据库系统。原因在于开地址中的删除操作十分麻烦。一般来说，编译器和汇编器只是在符号表上做查找和插入操作。可是在数据库系统中，处理删除的能力和处理插入一样重要。相比，开地址在数据库实现中的重要性不大。

我们前面描述的散列方式一个很大的缺点是我们必须在实现系统时选择确定的散列函数，此后若被索引的文件变大或缩小，要想再改变它就不容易了。因为函数 h 将搜索码值映射到桶地址的固定集合 B 上，如果为了处理将来文件的增长而将 B 取得较大就会浪费空间。如果 B 太小，一个桶就会包含许多具有不同的搜索码值的记录，从而可能发生桶溢出。当文件变大时，性能就会受到影响。稍后 11.7 节将介绍如何动态改变桶的数目和散列函数。

512
~
513

11.6.3　散列索引

散列不仅可以用于文件的组织，还可以用于索引结构的创建。**散列索引**(hash index)将搜索码及其相应的指针组织成散列文件结构。我们按如下方法构造散列索引。我们将散列函数作用于搜索码以确定对应的桶，然后将此搜索码以及相应指针存入此桶(或溢出桶)中。图 11-25 给出了 *instructor* 文件上的一个辅助散列索引，其搜索码是 *ID*。图 11-25 中的散列函数计算 *ID* 的各位数字之和对 8 取模的结果。该散列索引有 8 个桶，每个桶的大小为 2(当然，现实中索引的桶会大得多)。其中一个桶有三个码映射到它，因此它有一个溢出桶。在这个例子中，*ID* 是 *instructor* 的主码，所以每个搜索码只对应一个指针。一般情况下，每个码可能对应多个指针。

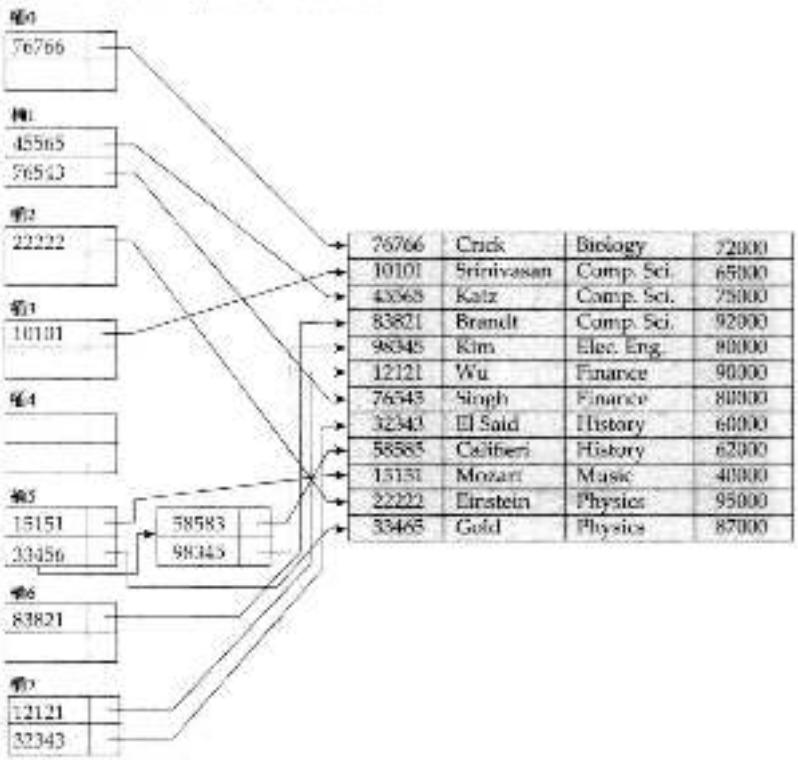


图 11-25　*instructor* 文件中在搜索码 *ID* 上的散列索引

我们使用术语散列索引来表示散列文件结构。同时也用它表示辅助散列索引。严格地说，散列索引只是一种辅助索引结构。散列索引从来不需要作为聚集索引结构来使用，因为如果一个文件自身是按散列组织的，就不必在其上另外建立一个独立的索引结构。不过，既然散列文件组织能像索引那样提供对记录的直接访问，我们不妨认为以散列形式组织的文件上也有一个聚集散列索引。

11.7 动态散列

正如我们已经看到的那样，前一节中的静态散列技术要求固定桶地址集合 B，这带来了一个很严重的问题。大多数数据库都会随时间而变大。如果我们准备为这样的数据库使用静态散列，我们有三种选择：

1. 根据当前文件大小选择散列函数。这种选择会使性能随数据库增大而下降。

2. 根据将来某个时刻文件的预计大小选择散列函数。尽管这样可以避免性能下降，但是初始时会造成相当大的空间浪费。

3. 随着文件增大，周期性地对散列结构进行重组。这种重组涉及一系列问题，包括新散列函数的选择，在文件中每条记录上重新计算散列函数，以及分配新的桶。重组是一个规模大、耗时的操作，而且重组期间必须禁止对文件的访问。

几种**动态散列**(dynamic hashing)技术允许散列函数动态改变，以适应数据库增大或缩小的需要。本节介绍一种动态散列技术，称为**可扩充散列**(extendable hashing)。文献注解列出了有关其他动态散列技术的参考。

11.7.1 数据结构

当数据库增大或缩小时，可扩充散列可以通过桶的分裂或合并来适应数据库大小的变化。这样可以保持空间的使用效率。此外，由于重组每次仅作用于一个桶，因此所带来的性能开销较低，可以接受。

514
515

使用可扩充散列时，我们选择一个具有均匀性和随机性特性的散列函数 h。但是，此散列函数产生的值范围相对较大，是 b 位二进制整数。一个典型的 b 值是 32。

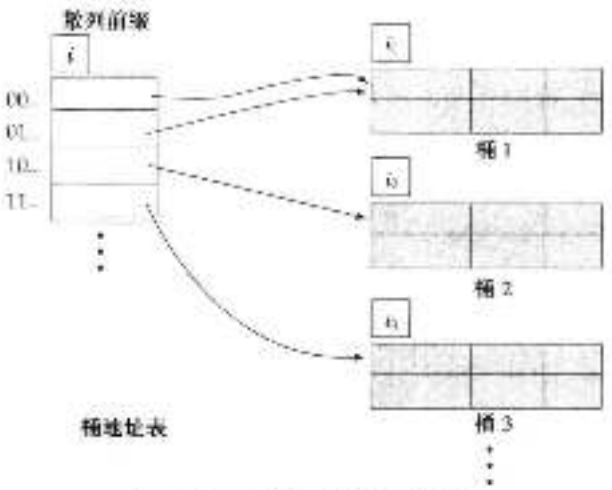


图 11-26 可扩充散列的一般结构

我们并不为每一个散列值创建一个桶。实际上，2^{32}超过 40 亿，除非是非常大的数据库，否则没有必要创建这么多桶。相反，我们是在把记录插入文件时按需建桶的。开始时，我们不使用散列值的全部 b 位，任意时刻我们使用的位数 i 满足 $0 \leq i \leq b$。这样的 i 个位用作附加的桶地址表中的偏移量。i 的值随着数据库大小的变化而增大或减小。

图 11-26 所示为一般的可扩充散列结构。图 11-26 中出现在桶地址表上方的 i 表明散列值 $h(K)$ 中有 i 位需要用来正确地定位对应于 K 的桶。当然，i 的值会随着文件增长而变化。尽管按出桶地址表中

的正确表项需要 i 位，但几个连续的表项可能指向同一个桶。所有这样的表项有一个共同的散列前缀，但这个前缀的长度可能小于 i。因此，我们给每一个桶附加一个整数值，用来表明共同的散列前缀长度。在图 11-26 中，与桶 j 相关联的整数是 i_j。桶地址表中指向桶 j 的表项编号为

$$2^{i-i_j}$$

11.7.2　查询和更新

我们现在来看如何在可扩充散列结构上执行查询、插入和删除。 [516]

为了确定含有搜索码值 K_l 的桶的位置，系统取得 $h(K_l)$ 的前 i 个高位，然后为这个位串会找对应的表项，再根据表项中的指针得到桶的位置。

要插入一条搜索码值为 K_l 的记录，系统按上述相同过程进行查找，最终定位到某个桶——假定为桶 j。如果该桶中有剩余空间，系统将该记录插入该桶即可；反之，如果桶 j 已满，系统必须分裂这个桶并将该桶中现有记录和新记录一起进行重新分配。为了分裂该桶，系统必须首先根据散列值确定是否需要增加所使用的位数。

- 如果 $i = i_j$，那么在桶地址表中只有一个表项指向桶 j。所以系统需要增加桶地址表的大小，以容纳由于桶 j 分裂而产生的两个桶指针。为了达到这一目的，系统考虑多引入散列值中的一位。系统将 i 的值加 1，从而使桶地址表的大小加倍。这样，原表中每个表项都被两个表项替代，两个表项都包含和原始表项一样的指针。现在桶地址表中有两个表项指向桶 j。这时，系统分配一个新的桶（桶 z），并让第二个表项指向此新桶。系统将 i_j 和 i_z 置为 i。接下来，桶 j 中的各条记录重新散列，根据前 i 位（记住 i 已加 1）来确定该记录是放在桶 j 中还是放到新创建的桶中。

  系统现在再次尝试插入该新记录。通常这一尝试会成功。但是，如果桶 j 中原有的所有记录和新插入的记录具有相同的散列值前缀，该桶就必须再次分裂，这是因为桶 j 中的所有记录和新插入的记录被分配到同一个桶中。如果散列函数已经过仔细挑选，一次插入导致两次或两次以上分裂是不太可能的，除非大量的记录具有相同的搜索码。如果桶 j 中所有记录搜索码值相同，那么多少次分裂也不能解决问题。在这种情况下，采用溢出桶来存储记录，就像在静态散列中那样。

- 如果 $i > i_j$，那么在桶地址表中有多个表项指向桶 j。因此，系统不需要扩大桶地址表就能分裂桶 j。我们发现指向桶 j 的所有表项的索引前缀的最左 i_j 位相同。系统分配一个新桶（桶 z），将 i_j 和 i_z 置为原 i_j 加 1 后得到的值。接下来系统需要调整桶地址表中原来指向桶 j 的表项。（注意，由于 i_j 有了新的值，并非所有表项的散列前缀的最左 i_j 位都相同。）系统让这些表项的前一半保持原样（指向桶 j），而使后一半指向新创建的桶（桶 z）。然后就像上一种情况那样，桶 j 中的各条记录被重新散列，分配到桶 j 或新桶 z 中。

  此时，系统重新尝试插入记录。失败的可能性依于其后，如果失败，则根据情况是 $i = i_j$ 还是 $i > i_j$ 继续做相应的处理。

注意，在这两种情况下，系统都只需要在桶 j 的记录上重新计算散列函数。

要删除一条搜索码值为 K_l 的记录，系统可以按前面的查找过程找到相应的桶，不妨设为桶 j。系统不仅要把搜索码从桶中删除，还要把记录从文件中删除。如果这时桶成为空的，那么桶也需要删除。注意，此时某些桶可能合并，桶地址表的大小也可能减半。决定哪些桶可以合并以及如何合并这些桶的过程留作习题。桶地址表的大小在何种条件下可以减小也留作习题。与桶的合并不同，若桶地址表很大，则改变该表的大小是一个开销相当大的操作。因此只有当桶数目减少很多时，减小桶地址表的大小才是值得的。

为了说明插入操作，我们使用图 11-1 中的 *instructor* 文件并假设搜索码为 *dept_name*，散列值有 32 位，如图 11-27 所示。假设该文件开始时是空的，如图 11-28 所示。一条一条地插入记录。为了用一个很小的结构演示可扩充散列的所有特性，我们需要做一个不实际的假设：一个桶只能容纳两条记录。

插入记录（10101, Srinivasan, Comp. Sci., 65000）。桶地址表包含一个指向桶的指针，系统插入该条记录。接着，我们插入记录（12121, Wu, Finance, 90000）。系统也把这条记录放到对应结构中的一个桶里。

当试图插入下一条记录(15151, Mozart, Music, 40000)时，我们发现桶已经满了。由于 $i=i_j$，因此需要增加所使用的散列值中的位数。现在我们使用 1 位，允许有 $2^1=2$ 个桶。这一位数的增加使桶地址表的大小必须加倍，增加到有两个表项。系统分裂桶，在新桶中放置那些搜索码的散列值以 1 开始的记录，而将其余的记录留在原来的桶中。图 11-29 给出了该结构在分裂后的状态。

dept_name	h(dept_name)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

图 11-27 dept_name 的散列函数

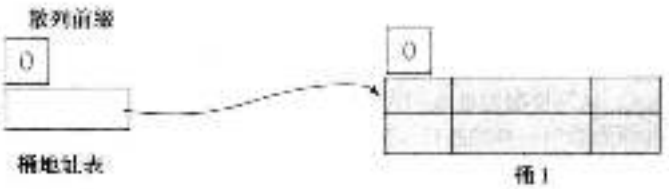


图 11-28 初始的可扩充散列结构

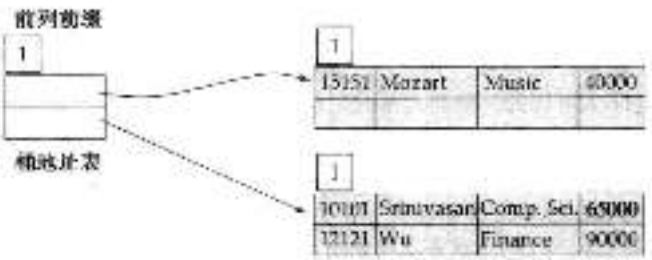


图 11-29 3 次插入操作后的散列结构

接下来，插入(22222, Einstein, Physics, 95000)。由于 h(Physics)的第一位是 1，因此必须将该记录插入桶地址表中表项"1"指向的桶。我们又一次发现桶满了并且 $i=i_j$。我们将使用的散列值位数增加到 2。这一位数增加使桶地址表的大小必须加倍，增加到有 4 个表项，如图 11-30 所示。由于图 11-29中桶列前缀为 0 的桶并未分裂，因此桶地址表中 00 和 01 表项都指向该桶。

对于图 11-29 中散列前缀为 1 的桶(正在分裂的桶)中的每一条记录，系统检查其散列值中的前两位，决定在新结构的哪一个桶中存放它。

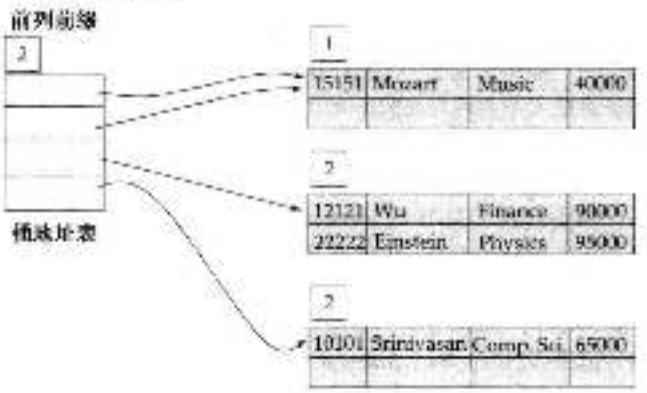


图 11-30 4 次插入操作后的散列结构

接下来，我们插入(32343, El Said, History, 60000)，它进入 Comp. Sci. 所在的同一个桶。然后又插入(33456, Gold, Physics, 87000)，这导致了桶溢出，引起位数增加和桶地址表的大小加倍(如图 11-31所示)。

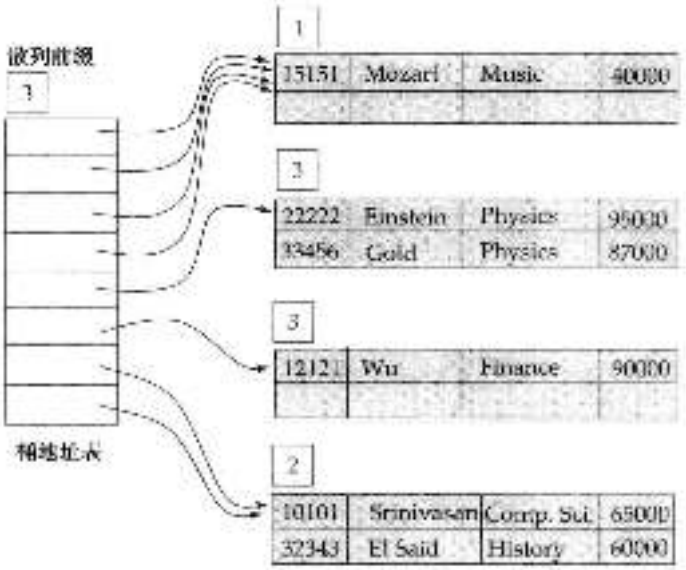


图 11-31　6 次插入操作后的散列结构

(45565, Katz, Comp. Sci., 75000)的插入引起另一次溢出。但是，这次溢出不必用增加位数来解决了，因为该桶有两个指向它的指针(如图 11-32 所示)。

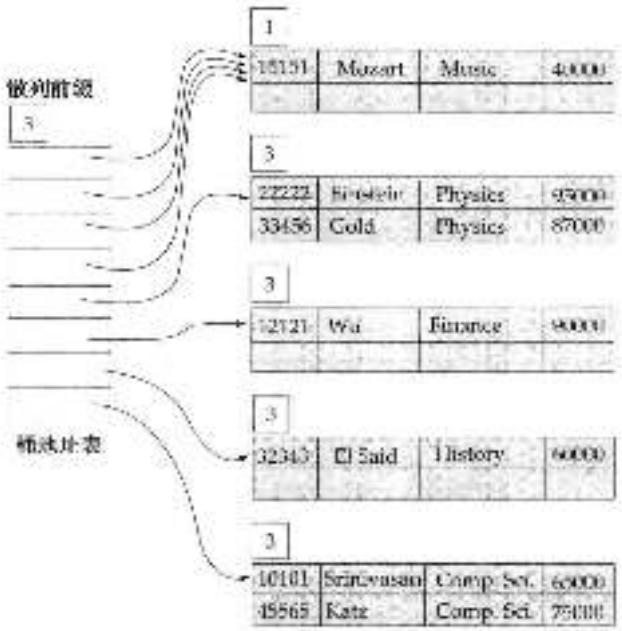


图 11-32　7 次插入操作后的散列结构

接下来插入"Califieri"、"Singh"、"Crick"的记录，它们不会带来桶溢出。但是，第二条 Comp. Sci. 记录(83821, Brandt, Comp. Sci., 92000)的插入会导致另外的溢出。这种溢出不能通过增加位数来解决，因为有二条记录具有相同的散列值。因此系统使用一个溢出桶，如 11-33 所示。我们继续这种方法进行，直到插入图 11-1 中的所有 *instructor* 记录为止。产生的结构如图 11-34 所示。

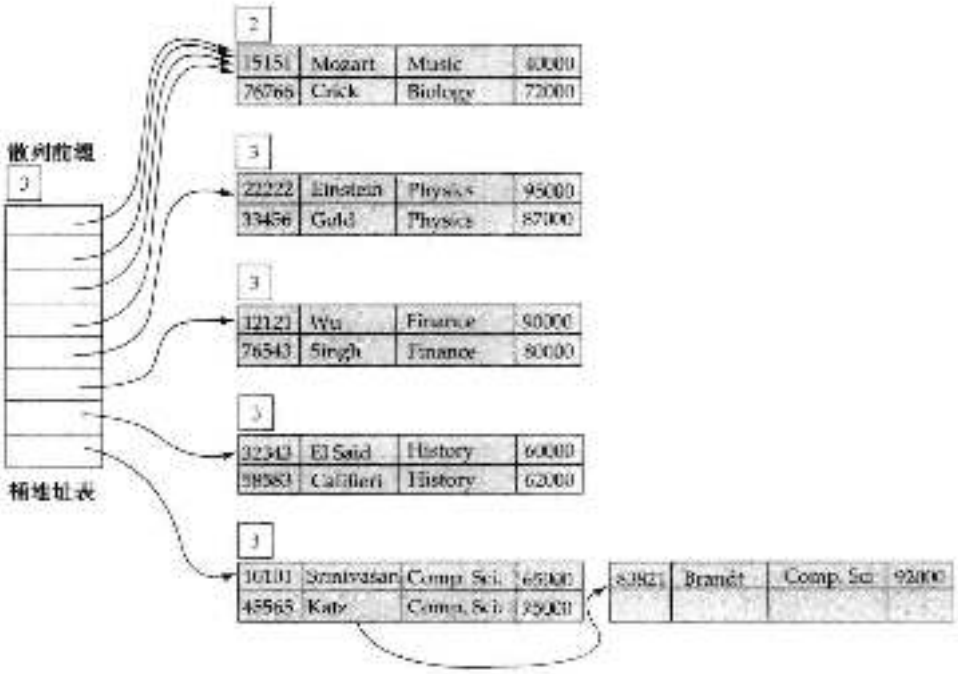


图 11-33 11 次插入操作后的散列结构

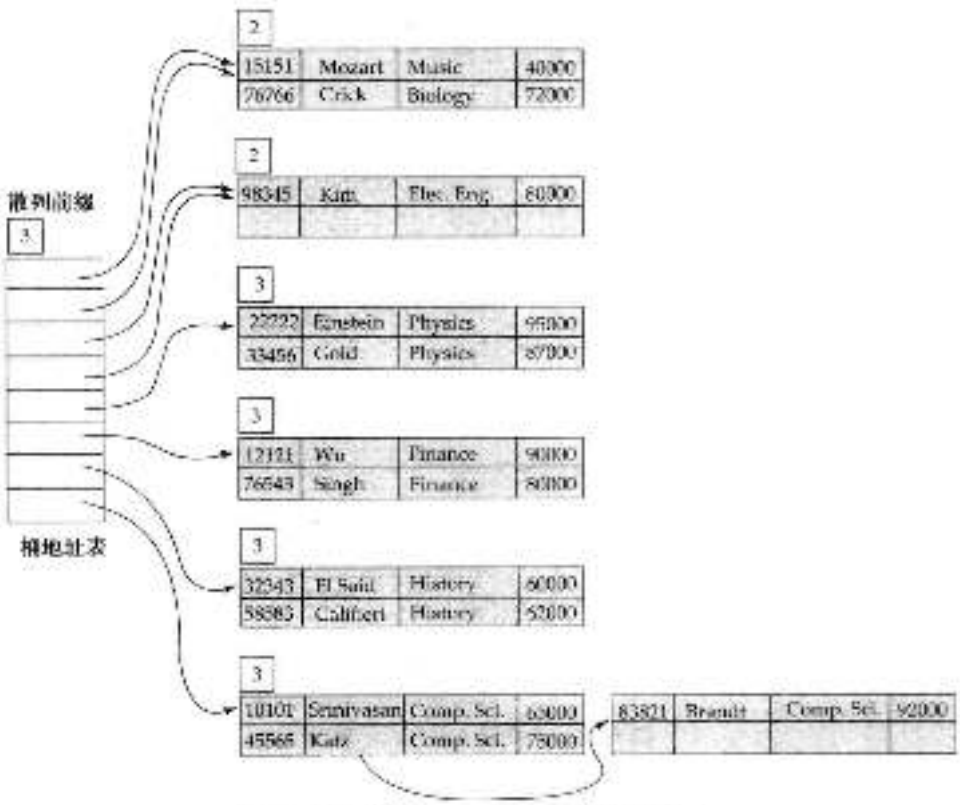


图 11-34 文件 *instructor* 的可扩充散列结构

11.7.3 静态散列与动态散列比较

现在让我们来看一看可扩充散列相对静态散列的优缺点。可扩充散列最主要的优点是其性能不随文件的增长而降低。此外，其空间开销是最小的。尽管桶地址表需要来了额外的开销，但该表为当前前缀长度的每个散列值存放一个指针，因此该表较小。可扩充散列与其他散列形式相比，主要的空间节省是不必为将来的增长保留桶：桶的分配是动态的。

可扩充散列的一个缺点在于查找涉及一个附加的间接层，因为系统在访问桶本身之前必须先访问桶地址表。这一额外的访问只对性能有一个微小的影响。尽管 11.6 节讨论的散列结构没有这一额外的间接层，但当它们变满后就失去了这个微小的性能优势。 517 ~ 522

因而，可扩充散列看来是一种非常吸引人的技术，只要我们愿意接受在实现它时增加的复杂性。关于可扩充散列实现更详细的资料见文献注解。

文献注解同时也提供了另一种动态散列形式——**线性散列**（linear hashing）的参考，这一方法以可能有更多溢出桶的代价为前提，避免了与可扩充散列相关联的额外的间接层。

11.8 顺序索引和散列的比较

我们已经看到了一些顺序索引方案和散列方案。可以用索引顺序组织或 B^+ 树组织将记录文件组织成顺序文件。另外，也可以用散列来组织文件。最后，可以将它们组织成堆文件，其中的记录不以任何特定方式排序。

各种方案在一定条件下各有其优点。数据库系统的实现者可提供多种方案，而将使用哪种方案的最后决定权留给数据库设计者。但是，这种方法需要实现者写更多的代码，加大了系统的成本和系统所占用的空间。大多数数据库系统支持 B^+ 树索引，并且有可能还额外支持某种形式的散列文件组织或散列索引。

要对关系的文件组织和索引技术做出明智的选择，实现者或数据库设计者必须考虑以下问题：

- 索引或散列组织的周期性重组代价是否可接受？
- 插入和删除的相对频率如何？
- 是否愿意以增加最坏情况下的访问时间为代价优化平均访问时间？
- 用户可能提出哪些类型的查询？

我们已经考虑了前三个问题，首先在我们回顾具体索引技术的相对优点时，然后在我们讨论散列技术时。第 4 个问题——预期的查询类型——不管对选择顺序索引还是散列都至关重要。

如果大多数查询形如

$$\begin{array}{l}\textbf{select } A_1, A_2, \cdots, A_k\\ \textbf{from } r\\ \textbf{where } A_i = c;\end{array}$$

那么，为了处理该查询，系统将在一个顺序索引或散列结构中为属性 A_i 查找值 c。对于这种形式的查询，散列的方案更可取。顺序索引的查找所需时间与关系 r 中 A_i 值的个数的对数成正比。但在散列结构中，平均查找时间是一个与数据库大小无关的常数。对于这种形式的查询，非散列的索引结构的唯一优点是最坏情况下的查找时间和关系 r 中 A_i 值的个数的对数成正比，而用散列时，最坏情况下所需的查找时间和关系 r 中 A_i 值的个数成正比。但是，用散列时最坏查找发生的可能性极小，因而在这种情况下散列更可取。 523

顺序索引技术在指出了一个值范围的查询中比散列更可取。这样的查询有如下形式：

$$\begin{array}{l}\textbf{select } A_1, A_2, \cdots, A_k\\ \textbf{from } r\\ \textbf{where } A_i \leqslant c_2 \textbf{ and } A_i \geqslant c_1\end{array}$$

换句话说，这一查询要找出 A_i 的值在 c_1 与 c_2 之间的所有记录。

让我们考虑如何用顺序索引处理该查询。首先查找值 c_1。一旦找到值 c_1 的桶，就可以顺着索引中的指针链顺序读取下一个桶，如此继续下去直到到达 c_2。

如果我们不用顺序索引而用散列结构，我们查找 c_1 并确定其对应的桶，但一般来说，决定下一个

必须检查的桶并不容易，因为好的散列函数总是将值随机地分散到各桶中。因而没有一个简单的概念能够表示"按顺序排序的下一个桶"。我们不能将桶按 A_r 的大小顺序串在一起的原因是每个桶都分配了许多搜索码值。既然值由散列函数随机地散布，在一定范围内的值就很可能散布在很多甚至全部桶中。因而，为了找到所需搜索码，我们不得不读取所有的桶。

通常设计者会使用顺序索引，除非他预先知道将来不会频繁使用范围查询，在这种情况下使用散列。散列组织对于在查询执行过程中创建的临时文件来说特别有用，如果需要基于码值查找并且不执行范围查询。

11.9 位图索引

位图索引是一种为多码上的简单查询设计的特殊索引。尽管每个位图索引都是建立在一个码之上的。

为了使用位图索引，关系中的记录必须按顺序编号，比如说从 0 开始。对于给定的一个 n 值，必
524 须能很简单地检索到编号为 n 的记录。如果记录是大小固定的，而且位于文件的连续块上，则实现这一点就更容易了。然后，该记录号就可以很简单地转化为一个块编号和一个指出块内记录的记录号。

考虑一个关系 r，它有一个属性 A 只能取很少的一些数值（例如，2～20）。例如，关系 *instructor_info* 可能有一个属性 *gender*，它只能取 m（男）和 f（女）值。另一个例子是属性 *income_level*，此处收入分成 5 级：*L1*：$0～$9999，*L2*：$10 000～$19 999，*L3*：$20 000～$39 999，*L4*：$40 000～$74 999，*L5*：$75 000～∞。在这里，原始数据可以取很多数值，但是数据分析者将这些数值划分成少数几个区间以简化数据分析。

11.9.1 位图索引结构

位图（bitmap）就是位的一个简单数组。在其最简单的形式中，关系 r 的属性 A 上的位图索引（bitmap index）基由 A 能取的每个值建立的位图构成的。每个位图都有和关系中的记录数相等数目的位。如果编号为 i 的记录在属性 A 上的值为 a_j，则值为 a_j 的位图中的第 i 个位设置为 1，而该位图上的其他所有位设置为 0。

在该例子中，对值 m 和 f 都分别有一个位图。如果号码为 i 的记录的 *gender* 值为 m，则 m 的位图中的第 i 位设置为 1，而 m 的位图中的其他位设置为 0。类似地，f 位图中等于 1 的位对应于属性 *gender* 的值为 f 的记录，而其他位都为 0。图 11-35 表示了关系 *instructor_info* 上的位图索引的例子。

我们现在考虑什么时候使用位图索引是有利的。用属性值为 m（或者 f）来检索所有记录，最简单的办法就是读取关系中的所有记录，而选择其中值为 m（或者 f）的记录。在这样的办法中，位图索引实际上并不能加快检索速度。尽管它可以让我们只读取某个特定性别的记录，但是很有可能文件的所有块都需要被读取到。

记录号	*ID*	*gender*	*income_level*
0	76766	*m*	*L1*
1	22222	*f*	*L2*
2	12121	*f*	*L1*
3	15151	*m*	*L4*
4	58583	*f*	*L3*

gender 的位图

m	10010
f	01101

income_level 的位图

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

图 11-35 关系 *instructor_info* 上的位图索引

事实上，位图索引主要在对多个码上的选择操作有用。假设如前所述，除了 *gender* 上的位图索引之外，还创建了属性 *income_level* 上的位图索引。

现在考虑一个选择收入在 $10 000 – $19 999 之间的女性的查询。这个查询可以表示成为

```
select *
from r
where gender = 'f' and income_level = 'L2';
```

为了计算这个选择，我们取属性 *gender* 值为 f 的位图和属性 *income_level* 值为 *L2* 的位图，然后执行两个位图的交(intersection)操作(差辑与)。换句话说，我们计算出了一个新的位图，如果前面两个位图的第 *i* 位值都为 1，则这个新位图的第 *i* 位值为 1；否则，为 0。在图 11-35 的例子中，*gender* 的位图 = f(01101) 和 *income_level* 的位图 = L2(01000) 的交得到位图 01000。

因为第一个属性可以取两个值，第二个属性可以取 5 个值，所以我们认为，平均 10 条记录中只有一条记录能满足两个属性上的组合条件。如果有更多的条件，则满足所有条件的记录在所有记录中所占的比例可能就很小了。这样，系统可通过在交操作以后的位图中找出值为 1 的所有位，然后检索相应的记录来计算出查询结果。如果满足条件的记录所占比例很大，则扫描整个关系将是代价更低的一种选择。

位图的另一个重要应用就是统计满足所给选择条件的元组数。这样的检索对于数据分析很重要。例如，我们希望找到有多少女性的收入水平为 *L2*，我们计算两个位图的交，然后统计交操作后得到的位图中值为 1 的位的数目。这样我们甚至可以在不需要访问关系的条件下从位图索引得到需要的结果。

和实际关系大小相比，位图索引通常比较小。典型的记录至少是几十或几百字节长，然而在位图中一位就可以代表一条记录。这样，单个位图占用的空间通常少于关系所占空间的 1%。例如，如果一个给定关系的记录长度为 100 字节，那么单个位图所占空间将是整个关系所占空间的 1% 的 1/8。如果关系的属性 *A* 可以取 8 个值，那么属性 *A* 上的位图索引将包含 8 个位图。这 8 个位图一起仅占用该关系大小的 1%。

删除记录会在顺序排列的记录之间产生间隙，因为移动记录(或者记录号)来填充间隙需要极大的代价。为了识别被删除的记录，我们可以存储一个**存在位图**(existence bitmap)，在该位图中如果第 *i* 位的值为 0，表示记录 *i* 不存在，否则为 1。我们将在 11.9.2 节了解存在位图的必要性。记录的插入不应该影响到其他记录的顺序编号。因此，我们可以通过在文件的末尾添加记录或者替换被删除的记录来完成插入操作。

525
↓
526

11.9.2　位图操作的高效实现

我们可以简单地用一个 **for** 循环来计算两个位图的交：第 *i* 个循环计算两个位图的第 *i* 位的与(**and**)。我们可以通过大多数计算机指令集支持的位模式 **and** 指令来显著加快交操作运算速度。根据不同的计算机体系结构，一个字(word)通常有 32 或 64 位。位模式 **and** 指令以两个字作为输入，输出一个字，该字的每一个位是输入的两个字对应位的与(**and**)。值得重点注意的是，位模式 **and** 指令可以用一次运算得到 32 位或者 64 位的交。

如果一个关系有 100 万条记录，每个位图将包含 100 万个位，相当于 128KB。假设字长是 32 位的，则为计算该例子中的两个位图的交，只需要 31 250 条指令。因此，位图的交运算就相当快。

正如位图的交运算可用于计算两个条件的与(**and**)，位图的并可用来计算两个条件的或(**or**)。位图并的过程和位图与的过程十分相似，不同之处在于，我们用位模式 **or** 指令代替位模式 **and** 指令。

补码操作可用于计算对某个条件取反的断言。例如 **not**(*income_level* = *L1*)。位图的补码可以用位图中每一位的补码来产生(1 的补码是 0，0 的补码是 1)。看起来，我们好像可以通过计算收入等级为 *L1* 的位图的补码来实现 **not**(*income_level* = *L1*)。但是，如果一些记录已经删除了，那么仅仅计算位图的补码是不够的。与删除记录相对应的位在原位图中为 0，但在补码中将为 1，而实际上该记录已经不存在了。当属性值为空(*null*)的时候，也会出现同样的问题。例如，如果 *income_level* 的值为空，在原来的位图中，值 *L1* 对应的位的值为 0，而在补码位图中为 1。

为了确保对应于被删除记录的位在结果中设置为 0，补码位图必须和存在位图进行交操作使得已删除记录对应的位为 0。类似地，处理空值时，补码位图也必须和空值(*null*)的补码位图进行交操作。[⊖]

⊖　处理诸如 is unknown 之类的断言会导致更多麻烦。这通常需要使用为一个位图来跟踪未知的操作结果。

可以通过一个巧妙的技术，使计算位图中值为 1 的位的数目变得很快。可以维护一个具有 256 个项的数组，其中第 i 个项存储 i 的二进制表示中值为 1 的位的个数。开始时设置总计数值为 0。我们取得位图中的每一个字节，将它作为数组的下标，然后将其中存储的值加到总计数值上。加操作的数目
527 将是整个元组数的 1/8，因此该计数过程是很有效的。一个大的数组（使用 $2^{16}=65536$ 个项）用多个字节来做数组下标，将会得到更高的加速比，但是需要更大的存储开销。

11.9.3 位图和 B⁺树

对于一些属性值经常出现，而另外一些属性值虽然也出现，但出现频率很小的关系，位图可以和一般的 B⁺树索引组合起来使用。在 B⁺树索引的叶结点中，对于每个值，我们通常保留以这个值为索引属性值的所有记录的列表。列表的每个元素可以是记录的标识符，至少有 32 位，而通常会更多。对于一个在许多记录中都出现的值，我们存储一个位图而不是记录的列表。

假设一个特殊的值 v 在 1/16 的关系记录中出现。令 N 为关系中的记录数目，并假设每条记录有一个 64 位的号来标识它。位图仅需要 1 位来表示每条记录，总共需要 N 位。相对而言，在值出现的地方，列表需要 64 位来表示每条记录，即总共 $64\times N/16=4N$ 位。因此，我们更倾向于使用位图来表示值 v 的记录列表。在该例子（使用 64 位记录标识）中，如果少于 1/64 的记录具有相同的值，对于标识具有这种特殊值的记录而言，则更倾向于使用列表表示，因为它比位图表示使用更少的位。如果多于 1/64 的记录具有相同的值，则倾向于使用位图表示。

因此，对于经常出现的那些值，我们可以在 B⁺树的叶结点中使用位图来作为一种压缩存储机制。

11.10 SQL 中的索引定义

SQL 标准并未为数据库用户或管理员提供任何在数据库系统中控制创建和维护何种索引的方法。由于索引是冗余的数据结构，因此索引对保证正确性来说并不是必需的。但是，索引对事务的高效处理十分重要，既包括更新事务又包括查询。索引对完整性约束的有效实施也很重要。

原则上，数据库系统可以自动决定创建何种索引。但是，由于索引的空间代价和索引对更新操作的影响，在维护何种索引上自动地做出正确选择并不容易。因此，大多数 SQL 实现允许程序员通过数据定义语言的命令对索引的创建和删除进行控制。

下面我们举例说明这些命令的语法。尽管我们给出的语法被很多数据库系统使用来支持，但它并不是 SQL 标准的一部分。SQL 标准不支持物理数据模式的控制，它将自身约束在逻辑数据模式层面。

528 我们使用 **create index** 命令创建索引，它的形式为

create index <index-name> **on** <relation-name> (<attribute-list>)

attribute list 是构成索引搜索码的关系属性列表。

为了在关系 *instructor* 上定义以 *dept_name* 为搜索码的名为 *dept_index* 的索引，我们写作

create index *dept_index* **on** *instructor*(*dept_name*)

如果我们想要声明该搜索码是一个候选码，就在索引定义中加入属性 unique。于是，命令

create unique index *dept_index* **on** *instructor*(*dept_name*)

声明了 *dept_name* 是 *instructor* 的一个候选码（这个有可能不是在我们的大学数据库中真正想要的）。如果当我们输入 **create unique index** 命令时 *dept_name* 不是候选码，系统就会给出错误消息，并且索引创建失败。如果创建该索引成功，接下来任何违反码声明的元组插入企图都将失败。注意，如果数据库系统支持 SQL 标准的 **unique** 声明，那么这里的 **unique** 特性就是多余的。

很多数据库系统也提供一种方法来详细说明要使用的索引类型（如 B⁺树或散列）。有些数据库系统也允许一个关系上的某个索引声明是聚集的；这样系统就以聚集索引的搜索码顺序来存储这个关系。

为索引指定的索引名在删除索引时是需要的。**drop index** 命令采用的形式是：

drop index <index-name>

11.11 总结

- 许多查询只涉及文件中很少一部分记录。为了减少查找这些记录的开销，我们可以为存储数据库的

文件创建索引。

- 索引顺序文件是数据库系统中最古老的索引模式之一。为了允许按搜索码顺序快速检索记录，记录按顺序存储，而无序记录链接在一起。为了允许快速的随机访问，使用了索引结构。
- 可以使用的索引类型有两种：稠密索引和稀疏索引。稠密索引对每个搜索码值都有索引项，而稀疏索引只对某些搜索码值包含索引项。 529
- 如果搜索码的排序序列和关系的排序序列相匹配，则该搜索码上的索引称为聚集索引。其他的索引称为非聚集索引或辅助索引。辅助索引可以提高不以聚集索引的搜索码作为搜索码的查询的性能。但是，辅助索引增加了修改数据库的开销。
- 索引顺序文件组织的主要缺陷是随着文件的增大，性能会下降。为了克服这个缺陷，可以使用 B^+ 树索引。
- B^+ 树索引采用平衡树的形式，即从树根到树叶的所有路径长度相等。B^+ 树的高度与以关系中的记录数 N 为底的对数成正比，其中每个非叶结点存储 N 个指针。N 值通常约为 50～100。因此，B^+ 树比其他的平衡二叉树（如 AVL 树）要矮很多，故定位记录所需的磁盘访问次数也较少。
- B^+ 树上的查询是直接而且高效的。然而，插入和删除要更复杂一些，但是仍然很有效。在 B^+ 树中，查询、插入和删除所需操作数与以关系中的记录数 N 为底的对数成正比，其中每个非叶结点存储 N 个指针。
- 可以用 B^+ 树去索引包含记录的文件，也可以用它组织文件中的记录。
- B 树索引和 B^+ 树索引类似。B 树的主要优点在于它去除了搜索码值存储中的冗余。主要缺点在于整体的复杂性以及结点大小给定时减小了扇出。在实际应用中，系统设计者几乎无一例外地倾向于使用 B^+ 树索引。
- 顺序文件组织需要一个索引结构来定位数据。相比之下，基于散列的文件组织允许我们通过计算所需记录搜索码值上的一个函数直接找出一个数据项的地址。由于设计时我们不能精确知道哪些搜索码值将存储在文件中，因此一个好的散列函数应该能均匀且随机地把搜索码值分散到各个桶中。
- 静态散列所用散列函数的桶地址集合是固定的。这样的散列函数不容易适应数据库随时间的显著增长。有几种允许修改散列函数的动态散列技术，可扩充散列是其中之一，它可以在数据库增长或缩减时通过分裂或合并桶来应付数据库大小的变化。 530
- 也可以用散列技术创建辅助索引；这样的索引称为散列索引。为使记法简便，假定散列文件组织中用于散列的搜索码上有一个隐式的散列索引。
- 像 B^+ 树和散列索引这样的有序索引可以用作涉及单个属性且基于相等条件的选择操作。当一个选择条件中涉及多个属性时，可以取多个索引中检索到的记录标识符的交。
- 对于索引属性只有少数几个不同值的情况，位图索引提供了一种非常紧凑的表达方式。位图索引的交操作相当得快，使得它成为一种支持多属性上的查询的理想方式。

术语回顾

- 访问类型
- 访问时间
- 插入时间
- 删除时间
- 空间开销
- 顺序索引
- 聚集索引
- 主索引
- 非聚集索引
- 辅助索引
- 索引顺序文件
- 索引记录/项
- 稠密索引
- 稀疏索引
- 多级索引
- 顺序扫描
- B^+ 树索引
- 叶结点
- 非叶结点
- 平衡树
- 范围查询
- 结点分裂
- 结点合并
- 不唯一搜索码
- B^+ 树文件组织
- 批量加载
- 自底向上 B^+ 树构建
- B 树索引
- 静态散列
- 散列文件组织
- 散列索引
- 桶
- 散列函数
- 桶溢出
- 偏斜
- 闭地址
- 动态散列
- 可扩充散列
- 多码访问

- 多码索引
- 位图索引
- 位图操作
  - 口交
  - 口并
  - 口补码
  - 口存在位图

531

实践习题

11.1 索引加速了查询处理，但是在作为潜在的搜索码的每一个属性上或者每一个属性组上创建索引，通常并不是一个很好的方法，请解释为什么。

11.2 在一个关系的不同搜索码上建立两个主索引一般来说是否可能？解释你的答案。

11.3 用下面的关键码值集合建立一个 B+ 树

(2, 3, 5, 7, 11, 17, 19, 23, 29, 31)

假设树初始为空，依按上升顺序加入。根据一个结点所能容纳指针数的下列情况分别构造 B+ 树：

a. 4

b. 6

c. 8

11.4 对习题 11.3 中的每一棵 B+ 树，给出下列各操作后树的形状：

a. 插入 9

b. 插入 10

c. 插入 8

d. 删除 23

e. 删除 19

11.5 考虑 11.4.1 节所述修改后的 B+ 树重分布模式。树的预期高度与 n 之间的函数关系是什么？

11.6 假设我们在一个文件上使用可扩充散列，该文件所含记录的搜索码依如下：

2, 3, 5, 7, 11, 17, 19, 23, 29, 31

如果散列函数为 $h(x) = x \bmod 8$，且每个桶可以容纳 3 条记录。给出此文件的可扩充散列结构。

532 11.7 进行下列各步以后，习题 11.6 中的可扩充散列结构如何变化？

a. 删除 11

b. 删除 31

c. 插入 1

d. 插入 15

11.8 给出 B+ 树函数 findIterator() 的伪码，该伪码类似于函数 find()，除了它返回如 11.3.2 节描述的迭代对象。同时，给出迭代类的伪代码，包括迭代对象中的变量和 next() 方法。

11.9 给出从可扩充散列结构中删除项的伪代码，包括何时和如何合并桶的细节。不必关心减少桶地址表的大小。

11.10 通过与桶地址表一起存储一个额外计数，提出一种有效的方法来测试是否能减少可扩充散列桶地址表的大小。详细描述在桶分裂、合并和删除时如何维护这个计数。（注意：减少桶地址表大小的操作是一个代价很高的操作，随后的插入可能导致表的再次增长。因此最好尽可能不要减少表的大小，只有当索引项的数目与桶地址表的大小相比较小时才这样做。）

11.11 考虑在图 11-1 中表示的关系 *instructor*。

a. 在属性 *salary* 上构建一个位图索引，把 *salary* 的值分成 4 个区间：小于 50000，50000 ~ 60000 以下，60000 ~ 70000 以下，70000 及其以上。

b. 考虑一个查询，即在金融系中所有工资为 80 000 美元或更多的教师。假设上述 salary 上的位图索引和 dept_name 上的位图索引可用概述回答该查询的步骤，并且给出为回答这个查询而构建的中间和最终位图。

11.12 如果索引项按照已排好的序插入，B^+ 树的每个叶结点的充满情况如何？解释为什么？

11.13 假设你有一个包含 n 条元组的关系 r。需要在上面建立一个辅助 B^+ 树索引。

a. 通过一次插入一条记录来建立 B^+ 树索引的代价是多少？给出计算公式。假设平均每个页面存储了 b 条索引项，并且所有高于叶级的结点都在内存中。

b. 假设一次随机磁盘访问的时间是 10 毫秒，在一个包含 1 000 万条记录的关系上建立索引的代价是多少？

c. 请写出如 11.4.4 节描述的自底向上构建 B^+ 树的伪代码。你可以假设存在一个可以有效排序大文件的函数。 533

11.14 为什么 B^+ 树文件组织的叶结点可能会丢失顺序性？

a. 请提议文件组织该如何重新组织来恢复顺序性。

b. 对于一个相当大的 n，另一种重新组织的方法是以 n 块为单位重新分配叶子页。当分配 B^+ 树的第一个叶子时，n 块单元中只有一块被使用，剩下的页是空闲的。如果有一页分裂，并且它的 n 块单元有空闲页，则新的页可以使用该空间。如果 n 块单元已经存满了，分配另一个 n 块单元，并且前 $n/2$ 叶子页被放入第一个 n 块单元，剩余的放入第二个 n 块单元中。简单起见，假设没有删除操作。

i. 假设没有删除操作，当第一个 n 块单元存满时，已分配空间的充满度的最坏情况是什么？

ii. 有没有可能出现这样的情况：分配给一个 n 结点块单元的叶子结点是不连贯的，也就是说有可能有两个叶子结点分配到同一个 n 结点块中，但是在它们之间的另一个叶子结点分配到了另一个不同的 n 结点块中？

iii. 在缓冲区对于存储一个 n 页的块来说是足够的这一合理假设下，B^+ 树的顺序扫描在最坏情况下需要多少次寻道？请把该数字和如果每次只为叶子页分配一个块的最坏情况进行比较。

iv. 为了提高空间利用率而将page重新分配给兄弟结点的技术如果与上述用于叶子结点的分配策略结合使用可能会更有效。请解释为什么。

习题

11.15 什么情况下使用稠密索引比使用稀疏索引更可取？解释你的答案。

11.16 聚集索引和辅助索引有何区别？

11.17 对习题 11.3 中的每一棵 B^+ 树，给出下列查询中涉及的步骤：

a. 找出搜索码值为 11 的记录。

b. 找出搜索码值在 7～17 之间（包括 7 和 17）的记录。

11.18 11.3.4 节列出的处理不唯一搜索码的解决方案是通过为每个搜索码添加一个附加的属性。这种改变会为 B^+ 树的高度带来什么样的影响？ 534

11.19 解释闭地址和开地址的区别。讨论这两种技术在数据库应用中的相对优点。

11.20 在散列文件组织中导致桶溢出的原因是什么？如何减少桶溢出的发生？

11.21 为什么对于一个可能在其上进行范围查询的搜索码而言散列结构不是最佳选择？

11.22 假设有一个关系 $r(A, B, C)$，带有一个搜索码为 (A, B) 的 B^+ 树索引。

a. 用这个索引查找满足 $10 < A < 50$ 条件的记录，最坏情况下的代价是多少？用获取的记录数目 n_1 和树的高度 h 来度量。

b. 用这个索引查找满足 $10 < A < 50 \wedge 5 < B < 10$ 条件的记录，最坏情况下的代价是多少？用满足条件的记录数目 n_2 和上面定义的 n_1 和 h 来度量。

c. 当 n_1 和 n_2 满足什么条件的时候这个索引对于查找满足 $10 < A < 50 \wedge 5 < B < 10$ 条件的记录是一个高效的手段？

11.23 假设你必须在大量的名字上建立一个 B^+ 树索引，这些名字的最大长度可能非常大（比如 40 个字符），并且这些名字长度的平均值本身也很大（比如 10 个字符）。解释一下如何用前缀压缩来使内部结点的平均扇出尽可能大。

11.24 假设一个关系以 B^+ 树文件组织的形式存储。假设辅助索引存储的记录标识符是非间接型中记录的指针。

a. 如果一个文件组织中发生磁盘页分裂，会对辅助索引产生什么影响？

b. 更新辅助索引中所有影响到的记录，代价是多少？

c. 使用一个逻辑记录标识符作为文件组织的搜索码如何能够解决这个问题？

d. 使用这样的逻辑记录标识符会带来什么附加的代价？

11.25 说明如何从其他位图中计算出存在位图，通过使用空(null)值位图。确保你的方法能在有空值存在的情况下仍然能够保持其正确性。

11.26 数据加密是如何影响索引方案的？特别地，在试图按排序顺序存储数据时，它是如何影响索引方案的？

11.27 我们对静态散列的描述作了这样的假设：一段大且连续的磁盘块能够分配给静态散列表。假设你只能分配 C 个连续磁盘块，提出如何实现可能远比 C 个磁盘块大的散列表。对磁盘块的访问仍然能够高效。

535

文献注解

有关索引和散列中所用的基本数据结构的讨论可在 Comer 等[1990]中找到。B 树索引最先由 Bayer [1972]以及 Bayer 和 McCreight[1972]引入的。Comer[1979]、Bayer 和 Unterauer [1977]以及 Knuth[1973]对 B⁺树进行了讨论。第 15 章的文献注解提供了有关允许对 B⁺树的并发访问和更新的研究工作的参考。Gray 和 Reuter[1993]在 B⁺树实现的问题上提供了很好的描述。

人们已经提出了几种可选的树和类树搜索结构。try 是一种其结构基于码中的"数字"(例如，字典中的 thumb 索引对每个字母有一个索引项)的树。这种树可能不像 B⁺树那样是平衡的。try 在 Ramesh 等[1989]、Orenstein[1982]、Litwin[1981]以及 Fredkin[1960]中进行了讨论。相关工作包括 Lomet[1981]中的数字 B 树。

Knuth [1973]对大量不同的散列技术做了分析。动态散列模式有几种：可扩充散列由 Fagin 等[1979]引入。线性散列由 Litwin[1978]和 Litwin[1980]引入；Rathi 等[1990]给出了线性散列与可扩充散列的性能比较；Ramakrishna 和 Larson[1989]给出的另一种模式以少数数据库更新开销很大为代价，使检索只须访问一次磁盘。分段散列是对多属性散列的扩充，见 Rivest[1976]、Burkhard[1976]和 Burkhard[1979]。

Vitter[2001]对外存的数据结构和算法给出了广泛的综述。

位图索引和它派生的位片索引(bit-sliced index)以及投影索引(projection index)由 O'Neil 和 Quass[1997]描述。它们在平台 AS400 上的 IBM Model 204 文件管理器中首次提出，它们对某些类型的查询提供了很大的加速比，在现在的大多数数据库系统中都有实现。近来，最近的有关位图索引的研究，可参考 Wu 和

536 Buchmann[1998]、Chan 和 Ioannidis[1998]、Chan 和 Ioannidis[1999]以及 Johnson[1999]。

查询处理

查询处理(query processing)是指从数据库中获取数据时涉及的一系列活动。这些活动包括：将用高层数据库语言表示的查询语句翻译为能在文件系统的物理层上使用的表达式，为优化查询而进行各种转换，以及查询的实际执行。

12.1 概述

查询处理步骤如图12-1所示。基本步骤包括：

1. 语法分析与翻译。
2. 优化。
3. 执行。

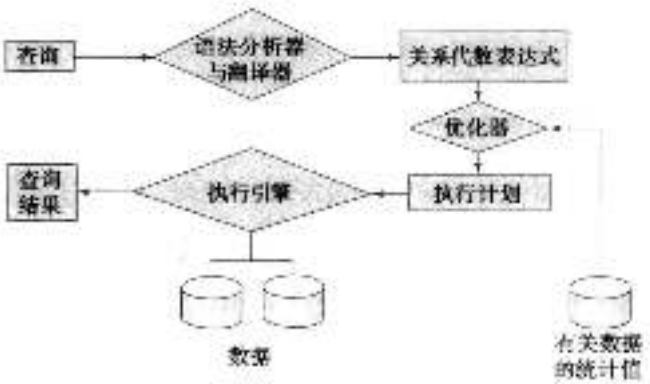


图12-1 查询处理的步骤

查询处理开始之前，系统必须将查询语句翻译成可使用的形式。如SQL这样的语言适合人使用，然而并不适合查询的系统内部表示。一种更有用的内部表示是建立在扩展的关系代数基础上的。

因此，查询处理中系统首先必须把查询语句翻译成系统的内部表示形式。该翻译过程类似于编译器的语法分析器所做的工作。在产生查询语句的系统内部表示形式的过程中，语法分析器检查用户查询的语法，验证查询中出现的关系名是数据库中的关系名等。系统构造该查询语句的语法分析树表示，然后将之翻译成关系代数表达式。如果查询是用视图形式表示的，翻译阶段还要用定义该视图的关系代数表达式来替换所有对该视图的引用。[⊖]大多数有关编译器的教科书中都有讨论语法分析的内容。 [537]

对于给出的一个查询，一般都会有多种计算结果的方法。例如，我们已知在SQL中，一个查询能够用几种不同的方式表示。每个SQL查询可以用其中的一种方式翻译成关系代数表达式。另外，查询的关系代数表达式形式仅仅部分地指定了如何执行查询；通常有许多方式来计算关系代数表达式。作为例子，考虑查询：

⊖ 对于物化视图，定义视图的表达式已经执行并存储了结果。以后，可以使用存储的关系，而不需要用定义视图的表达式替换视图。虚拟视图的处理与此不同，它通过不动点过程来处理，如5.4节和附录D.3.6所述。

```
select salary
from instructor
where salary < 75000;
```

该查询语句可翻译成下面两个关系代数表达式中的任意一个：

- $\sigma_{salary < 75000}(\Pi_{salary}(instructor))$
- $\Pi_{salary}(\sigma_{salary < 75000}(instructor))$

进一步地，我们可以用其中一种不同的算法来执行每个关系代数运算。例如，要实现前面的选择，可以通过扫描 *instructor* 的每个元组找出满足 *salary* < 75 000 条件的元组。如果存在属性 *salary* 上的 B^+ 树索引，我们可以改用它来定位元组。

要全面说明如何执行一个查询，我们不仅要提供关系代数表达式，还要对表达式加上注释来说明如何执行每个操作。注释可以声明某个具体操作所采用的算法，或将要使用的一个或多个特定的索引。加了"如何执行"注释的关系代数运算称为**计算原语**(evaluation primitive)。用于执行一个查询的原语操作序列称为**查询执行计划**(query-execution plan)或**查询计算计划**(query evaluation plan)。图 12-2 表示对所给查询例子的一个执行计划。图 12-2 中为选择运算指定了一个具体的索引(图 12-2 中用"索引 1"表示)。**查询执行引擎**(query-execution engine)接受一个查询执行计划，执行该计划并把结果返回给查询。



图 12-2 一个查询执行计划

给定查询的不同执行计划会有不同的代价。我们不能寄希望于用户写出具有最高效率执行计划的查询语句。相反，构造具有最小查询执行代价的查询执行计划应当是系统的责任。这项工作叫作**查询优化**。第 13 章将详细介绍查询优化。

一旦选定了查询计划，就用该计划来执行查询并输出查询结果。

以上对查询语句的处理步骤的描述是示意性的；并不是所有数据库系统都完全遵从这些步骤。举个例子来说，许多数据库系统并不用关系代数表达式来表示查询，而是采用基于给定 SQL 查询结构的带注释的语法分析树来表示。然而，我们此处所讲述的概念构成了数据库中查询处理的基础。

为了优化查询，查询优化器必须知道每个操作的代价。虽然精确计算代价是很难的，因为这依赖于许多参数(例如该操作实际能用的内存)，但对每个操作的执行代价得出一个粗略的估计是可能的。

这一章我们将学习在一个查询计划中如何执行单个的操作，以及如何估计它们的代价。第 13 章将回到查询优化的讨论。12.2 节概述如何度量某个查询的执行代价。12.3 – 12.6 节介绍单个关系代数操作的执行。某些操作可以组合成**流水线**(pipeline)，其中的每个操作都同时在输入元组上开始执行，即使某个操作的输入元组由另一个操作产生。12.7 节将讨论在一个查询执行计划中如何协调多个操作的执行，特别是如何将操作组织成流水线，以避免将中间结果立即写入磁盘。

538
↓
539

12.2 查询代价的度量

数据库中可以存在多种可能的查询计算计划，重要的是要能够根据它们的(估计)代价来对不同的计划做比较，并选择最佳方案。为此，我们必须估计个别操作的成本，并结合它们得到了一个查询计算计划的开销。因此，后面的章节会研究每一个操作的执行算法，也会略述如何估计操作的代价。

查询处理的代价可以通过该查询对各种资源的使用情况进行度量。这些资源包括磁盘存取、执行一个查询所用 CPU 时间，还有在并行/分布式数据库系统中的通信代价(我们将在第 18 – 19 章中讨论)。

然而在大型数据库系统中，在磁盘上存取数据的代价通常是最主要的代价，因为磁盘存取比内存操作速度慢。此外，CPU 速度的提升比磁盘速度的提升要快得多。这样很可能花费在磁盘存取上的时间仍将决定整个查询执行的时间。一个任务消耗的 CPU 时间比较难以估计，因为它取决于执行代码的低层详细情况。尽管实际应用中查询优化器的确把 CPU 时间考虑在内，但为了简化起见我们将忽略 CPU 时间，而仅仅用磁盘存取代价来度量查询执行计划的代价。

我们用传送磁盘块数以及搜索磁盘次数来度量查询计算计划的代价。假设磁盘子系统传输一个块

的数据平均消耗 t_T 秒，磁盘块平均访问时间（磁盘搜索时间加上旋转延迟）为 t_S 秒。则一次传输 b 个块以及执行 S 次磁盘搜索的操作将消耗 $b * t_T + S * t_S$ 秒。t_S 和 t_T 的值必须针对所使用的磁盘系统进行计算，而当今高端磁盘的典型数值通常是 $t_S = 4$ 毫秒和 $t_T = 0.1$ 毫秒（假定磁盘块的大小是 4KB，传输率为 40MB/秒[⊖]）。

通过把读磁盘块与写磁盘块区分开，我们可以进一步细化磁盘存取代价的估算，因为写磁盘块的代价通常是读磁盘块的两倍（这是由于磁盘系统在写完扇区后还会重新读取该扇区以验证写操作已经成功）。简化起见我们忽略这个细节，并把它留给读者对各种操作做出更精确的代价估算。

我们给出的代价估算没有包括将操作的最终结果写回磁盘的代价，当需要时再单独考虑。我们所考虑的所有的算法代价依赖于主存中缓冲区的大小。最好的情形是所有的数据可以读入到缓冲区中，不必再访问磁盘。最坏的情形是假定缓冲区只能容纳数目不多的数据块——大约每个关系一块。涉及代价估算时，我们通常假定是最坏的情形。 [540]

另外，尽管我们假定开始时数据必须从磁盘中读取出来，但是很可能我们访问的磁盘块已经在内存缓冲区中，为简化起见，对这种情况我们也忽略了。由此，执行一个查询计划过程中的实际磁盘存取代价可能会比估算的代价要小。

假设计算机中没有其他活动在进行，那么一个查询计算计划的**响应时间**（response time）（即执行计划所需的挂钟时间）就是所有的这些开销，并可以作为计划的代价的度量。遗憾的是，没有实际地执行计划，就很难估算计划的响应时间，原因如下：

1. 当查询开始执行，响应时间依赖于缓冲区中的内容；在对查询进行优化时，该信息是无法获取的，而且即便可以获取也很难用于计算。

2. 在具有多张磁盘的系统中，响应时间依赖于访问如何分布在各磁盘上。没有对分布在磁盘中的数据的详细了解这是很难估计的。

有趣的是，以额外的资源消耗为代价，计划可能获得更好的响应时间。例如，假如一个系统有多张磁盘，一个计划 A 需要额外的磁盘读取，但它并行地跨多张磁盘执行读，它可能比另一个计划 B 完成更快，尽管计划 B 有较少的磁盘读取，但它从同一张磁盘读。然而，如果一个查询的许多实例同时使用计划 A 来运行，整体响应时间可能比如果相同的这些实例使用计划 B 来执行要长，这是因为计划 A 带来更多的磁盘负载。

因此，优化器通常努力去尽可能降低查询计划总的**资源消耗**（resource consumption），而不是尽可能缩短响应时间。我们用于估计总的磁盘访问时间（包括寻道和数据传输）的模型，就是一个这样的基于资源消耗的查询代价模型的实例。

12.3 选择运算

在查询处理中，**文件扫描**（file scan）是存取数据最低级的操作。文件扫描是用于定位、检索满足选择条件的记录的搜索算法。在关系系统中，若关系保存在单个专用的文件中，采用文件扫描就可以读取整个关系。

12.3.1 使用文件扫描和索引的选择

考虑所有元组都保存在单个文件的关系上的一个选择运算。执行一个选择最简单的方式如下。

- A1（线性搜索）：在线性搜索中，系统扫描每一个文件块，对所有记录都进行测试，看它们是否满足选择条件。开始时需做一次磁盘搜索来访问文件的第一个块。如果文件的块不是顺序存放的，也许需要更多的磁盘搜索。不过为简化起见我们忽略这种情况。 [541]

  虽然线性搜索比其他实现选择操作的算法速度要慢，但它可用于任何文件，不用管该文件的顺序。索引的可用性，以及选择操作的种类。我们将要研究的其他算法并不能应用到所有情况下，但在可用情况下它们一般都比线性搜索要快。

⊖ 一些数据库系统会进行磁盘寻道和磁盘块传输的测试来估算平均寻道代价和磁盘块传输代价。并把这个步骤作为安装过程的一部分。

线性扫描以及其他选择算法的代价估计如图12-3所示。在图12-3中，我们使用h_i表示B^+树的高度。现实中的优化器通常假设树的根在内存的缓冲区中，因为它被频繁地访问。有的优化器甚至假设树的所有非叶级别都是保存在内存中的，因为它们被相对频繁地访问，并且B^+树结点中通常只有小于1%的是非叶结点。代价公式可以做适当的修改。

索引结构称为**存取路径**(access path)，因为它们提供了定位和存取数据的一条路径。在第11章中，我们曾指出按与物理顺序一致的顺序读取文件记录的效率较高。回想一下，主索引(也称为聚集索引)允许文件记录可以按与其在文件中的物理顺序一致的顺序进行读取。不是主索引的索引称为辅助索引。

使用索引的搜索算法称为**索引扫描**(index scan)。我们用选择谓词来指导我们在查询处理中选择使用哪个索引。使用索引的搜索算法有以下几种。

- **A2(主索引，码属性等值比较)**：对于具有主索引的码属性的等值比较，我们可以使用索引检索到满足相应等值条件的唯一一条记录。代价估计如图12-3所示。
- **A3(主索引，非码属性等值比较)**：当选择条件是基于非码属性A的等值比较时，我们可以利用主索引检索到多条记录。与前一种情况唯一不同的是，这种情况下需要取多条记录。然而，因为文件是依照搜索码进行排序的，所以这些记录在文件中必然是连续存储的。代价估计如图12-3所示。
- **A4(辅助索引，等值比较)**：使用等值条件的选择可以使用辅助索引。若等值条件是码属性上的，则该策略可检索到唯一一条记录；若索引字段是非码属性，则可能检索到多条记录。

在第一种情况下，只检索到一条记录。这种情况下的时间代价与主索引的情况一样(参看A2)。

在第二种情况下，每条记录可能存在于不同的磁盘块中，可能导致每检索到一条记录需要一次I/O操作，以及一次I/O操作需要一次搜索和一次磁盘块传输。如果每条记录位于不同磁盘块并且所取块是任意排列的，在这种情形下的最坏情形时间代价是$(b_r + n) \times (t_T + t_S)$，$n$是所取的记录数目。如果要检索大量记录，最坏情形开销甚至会比线性搜索更差。

542

如果内存缓冲区较大，那么包含该记录的块可能已经在缓冲区中。我们可以将包含该记录的块已经位于缓冲区中的可能性考虑进去，来构建选择操作的平均(average)或期望(expected)代价的估算。对于大的缓冲区，这样的估计会比最坏情形的估计小很多。

	算 法	开 销	原 因
A1	线性搜索	$t_S + b_r * t_T$	一次初始搜索加上b_r个块传输，b_r表示在文件中的块数量
A1	线性搜索，码属性等值比较	平均情形 $t_S + (b_r/2) * t_T$	因为检索一条元素满足条件，所以只要找到所需的记录，扫描就可以终止。在最坏的情形下，仍需要b_r个块传输
A2	B^+树主索引，码属性等值比较	$(h_i + 1) * (t_T + t_S)$	其中h_i表示索引的高度。索引查找穿过树的高度，再加上一次I/O来取记录；每个这样的I/O操作需要一次搜索和一次块传输
A3	B^+树主索引，非码属性等值比较	$h_i * (t_T + t_S) + b * t_T$	树的每层一次搜索，第一个块一次搜索，b是包含具有指定搜索码记录的块数。假定这些块是顺序存储(因为是主索引)的叶子块并且不需要额外搜索
A4	B^+树辅助索引，码属性等值比较	$(h_i + 1) * (t_T + t_S)$	这种情形和上述索引相似
A4	B^+树辅助索引，非码属性等值比较	$(h_i + n) * (t_T + t_S)$	其中n是所取记录数。)索引查找的代价和A3相似，但是每条记录可能存在不同的块上，这需要每条记录一次搜索。如果n值比较大，代价可能会非常高
A5	B^+树主索引，比较	$h_i * (t_T + t_S) + b * t_T$	和A3，非码属性等值比较情形一样
A6	B^+树辅助索引，比较	$(h_i + n) * (t_T + t_S)$	和A4，非码属性等值比较情形一样

图12-3 选择算法代价估计

在某些算法中，包括 A2，因为记录存储于树的叶子级结点，所以使用 B+树文件组织可以节省一次存取。

正如 11.4.2 节所描述的，当记录以 B+树文件组织或其他可能需要把记录重新配置的文件组织的方式存储时，辅助索引通常不存储指向记录的指针[⊖]。反之，辅助索引存储的是 B+树文件组织中作为码值的属性值。通过这种辅助索引存取一条记录的代价将更大：首先必须搜索辅助索引以找到主索引的搜索码值，然后查找主索引来找到记录。如果使用这样的索引，对辅助索引的代价公式的描述需要做适当的调整。

12.3.2 涉及比较的选择

考虑形如 $\sigma_{A\leq v}(r)$ 的选择。我们可以用线性搜索，或按以下方法之一使用索引来实现选择运算。

- **A5(主索引，比较)**：在选择条件是比较时，可使用顺序主索引(如 B+树主索引)。形如 $A>v$ 或 $A\geq v$ 的比较条件，可按以下方式使用 A 上的主索引来指导元组的检索。对于 $A\geq v$，我们在索引中寻找值 v，以检索出满足条件 $A=v$ 的首条记录。从该元组开始到文件末尾进行一次文件扫描就返回所有满足该条件的元组。对于 $A>v$，文件扫描从第一条满足 $A>v$ 的记录开始。这种情况下的代价估算跟 A3 的情况一样。

  对于形如 $A<v$ 或 $A\leq v$ 的比较式，没有必要查找索引。对于 $A<v$，只是简单地从文件头开始进行文件扫描，直到遇上(但不包含)首条满足 $A=v$ 的元组为止。$A\leq v$ 的情形类似，只是扫描直到遇上(但不包含)首条满足 $A>v$ 的元组为止。这两种情况下，索引都没有什么用处。
- **A6(辅助索引，比较)**：可以使用有序辅助索引指导涉及 $<$、$\leq$、$\geq$、$>$ 的比较条件的检索。对于"$<$"及"$\leq$"情形，扫描最底层索引块是从最小值开始直到 v 为止；对于"$>$"及"$\geq$"情形，扫描是从 v 开始直到最大值为止。

  辅助索引提供了指向记录的指针，但我们需要使用指针以取得实际的记录。由于连续的记录可能存在于不同的磁盘块中，因此每取一条记录可能需要一次 I/O 操作。和前面一样，一次 I/O 操作需要一次磁盘搜索和一次块传输。如果检索得到的记录数很大的话，使用辅助索引的代价甚至比线性搜索还要大。因此辅助索引应该仅在选择得到的记录很少时使用。

543 - 544

12.3.3 复杂选择的实现

到此为止，我们只考虑了形如 $A\ op\ B$ 的简单选择条件，其中 op 是等值或比较运算。现在我们来看看更复杂的选择谓词。

- **合取**：*合取选择(conjunctive selection)*形式如下，

$$\sigma_{\theta_1\wedge\theta_2\wedge\cdots\wedge\theta_n}(r)$$

- **析取**：*析取选择(disjunctive selection)*形式如下，

$$\sigma_{\theta_1\vee\theta_2\vee\cdots\vee\theta_n}(r)$$

  所有满足单个简单条件 θ_i 的记录的并集满足该析取条件。
- **取反**：选择操作 $\sigma_{\neg\theta}(r)$ 的结果就是关系 r 中对条件 θ 取值为假的元组的集合。如果没有空值的话，该结果就是那些不在 $\sigma_\theta(r)$ 中的元组的集合。

我们可以用下列算法之一来实现涉及多个简单条件的合取或析取的选择操作。

- **A7(利用一个索引的合取选择)**：首先我们判断是否存在某个简单条件中的某个属性上的一条存取路径。若存在，则可以用选择算法 A2 - A6 中的一个来检索满足该条件的记录。然后在内存缓冲区中，我们通过测试每条检索到的记录是否满足其余的简单条件，来最终完成这个操作。

  为减少代价，我们选择某个 θ_i 及 A1 - A6 算法之一，它们的组合可使 $\sigma_{\theta_i}(r)$ 的代价达到最小。算法 A7 的代价由所选算法的代价给出。
- **A8(使用组合索引的合取选择)**：某些合取选择可能可以使用合适的组合索引(*composite index*)

⊖ 回顾一下，如果用 B+树文件组织存储关系，则当叶结点分裂、合并或记录重新配置的时候记录可能在磁盘块之间移动。

（即在多个属性上建立的一个索引）。如果选择指定的是两个或多个属性上的等值条件，并且在
545 这些属性字段的集合上又存在组合索引，则可以直接搜索索引。索引的类型将决定使用 A2、A3、A4 算法中的哪一个。

- **A9（通过标识符的交实现合取选择）**：另一种可选的实现合取选择的方法是利用记录指针或记录标识符。该算法要求各个条件所涉及的字段上有带记录指针的索引。该算法对每个索引进行扫描，获取那些指向满足单个条件的记录的指针。所有检索到的指针的交集就是那些满足合取条件的指针的集合。然后算法利用该指针集合获取实际的记录。如果并非各个条件上均存在索引，则该算法要用剩余条件对所检索到的记录进行测试。

  算法 A9 的代价是扫描各个单独索引代价的总和加上获取检索到的指针列表的交集中的记录的代价。对指针列表进行排序并按照排序顺序检索记录能够减少该算法的代价。因此，（1）应把指向一个磁盘块中所有记录的指针归并到一起，这样只需通过一次 I/O 操作就可以获取到在该磁盘块中选择的所有记录；并且（2）磁盘块的读取也要按存储次序执行，这样磁盘臂的移动最少。12.4 节描述排序算法。

- **A10（通过标识符的并实现析取选择）**：如果在析取选择中所有条件上均有相应的存取路径存在，则逐个扫描索引获取满足单个条件的元组指针。检索到的所有指针的并集就是指向满足析取条件的所有元组的指针集。然后利用这些指针检索实际的记录。

  然而，即使只有其中的一个条件不存在存取路径，我们也不得不对这个关系进行一次线性扫描以找出那些满足该条件的元组。因此，只要析取式中有一个这样的条件，最有效的存取方法就是线性扫描，扫描的同时对每个元组进行析取条件测试。

具有取反条件的选择的实现留作练习（实践习题 12.6）。

12.4 排序

数据排序在数据库系统中有重要的作用，其原因有两个：第一个原因是 SQL 查询会指明对结果进行排序；第二个原因是当输入的关系已排序时，关系运算中的一些运算（如连接运算）能够得到高效实现。这个原因对查询处理而言也是同等重要。因此，本书先讨论排序，然后在 12.5 节中讨论连接运算。

通过在排序码上建立索引，然后使用该索引按序读取关系，可以完成对关系的排序。然而，这一过程仅仅在逻辑上通过索引对关系排序，而没有在物理上排序。因此，顺序读取元组可能导致每读一
546 个元组就要访问一次磁盘（磁盘搜索加上磁盘块传输）。由于记录数目可能比磁盘块的数目大很多，因此这样做的代价会很大。出于以上原因，有时需要在物理上对记录排序。

人们已对排序的有关问题进行了广泛的研究，这些研究既有针对内存中能够完全容纳的关系的，又有针对不能完全被内存容纳的关系的。在第一种情况下，可以利用标准的排序技术（如快速排序）。这里讨论如何处理第二种情况。

12.4.1 外部排序归并算法

对不能全部放在内存中的关系的排序称为**外排序**（external sorting）。外排序中最常用的技术是**外部排序归并**（external sort-merge）算法。下面讲述该算法。令 M 表示内存缓冲区中可以用于排序的块数，即内存的缓冲区能容纳的磁盘块数。

1. 第一阶段，建立多个排好序的**归并段**（run）。每个归并段都是排序过的，但仅包含关系中的部分记录。

```
i = 0;
repeat
    读入关系的 M 块数据或剩下的不足 M 块的数据;
    在内存中对关系的这一部分进行排序;
    将排好序的数据写到归并段文件 Ri 中;
    i = i + 1;
until 到达关系末尾
```

2. 第二阶段，对归并段进行归并。暂时假定归并段的总数 N 小于 M，这样我们可以为每个归并段

文件分配一个块，此外剩下的空间还应能容纳存放结果的一个块。归并阶段的工作流程如下：

为 N 个归并段文件 R_i 各分配一个内存缓冲块，并分别读入一个数据块；

```
repeat
    在对有缓冲块中按序挑选第一个元组；
    把该元组作为输出写出，并将其从缓冲块中删除；
    if 在第i个归并段文件 R_i 的缓冲块为空并且没有到达 R_i 末尾
        then 读入 R_i 的下一块到相应的缓冲块
until 所有的缓冲块均空
```

归并阶段的输出是已排序的关系。输出文件也被缓冲以减少写磁盘次数。上面的归并算法是对标准内存排序归并算法中的二路归并算法的推广；由于该算法对 N 个归并段进行归并，因此它称为 **N 路归并**（N-way merge）。 547

一般而言，若关系比内存大得多，则在第一阶段可能产生 M 个甚至更多的归并段，并且在归并阶段为每个归并段分配一个块是不可能的。在这种情况下，归并操作需要分多趟进行。由于内存足以容纳 $M-1$ 个缓冲块，因此每趟归并可以用 $M-1$ 个归并段作为输入。

最初部趟归并过程如下：头 $M-1$ 个归并段如前第 2 点所述进行归并得到一个归并段作为下一趟的输入。接下来的 $M-1$ 个归并段类似地进行归并，如此下去，直到所有的初始归并段都处理过为止。此时，归并段的数目减少到原来的 $1/(M-1)$，如果归并后的归并段数目仍大于等于 M，则以下一趟归并创建的归并段作为输入进行下一趟归并。每一趟归并段的数目均减少为原来的 $1/(M-1)$。如有需要归并过程将不断重复，直到归并段数目小于 M，此时作最后一趟归并，得到排序的输出结果。

图 12-4 显示了对一个示例关系进行外部归并排序的过程。为了方便说明，我们假定每块只能容纳 1 个元组（$f_r=1$），同时假定内存最多只能提供 3 个块：在归并阶段，两个块用于输入，另一块用于输出。



图 12-4 使用归并排序的外排序

12.4.2 外部排序归并的代价分析

下面我们计算外部归并排序的磁盘存取代价。令 b_r 代表包含关系 r 中记录的磁盘块数。在第一阶段要读入关系的每一数据块并写出，共需 $2b_r$ 次磁盘块传输。初始归并段数为 $\lceil b_r/M \rceil$。由于每一趟归并会使归并段数目减少为原来的 $1/(M-1)$，因此总共所需归并趟数为 $\lceil \log_{M-1}(b_r/M) \rceil$。对于每一趟归并，关系的每一数据块读写各一次，其中有两趟例外。首先，最后一趟可以只产生排序结果而不写入磁盘。其次，可能存在在某一趟中既没有读入又没有写出的归并段——例如，某一趟有 M 个归并段需归并，其中 $M-1$ 个被读入并归并，而另一个归并段在该趟归并中却未被访问。忽略后一种特殊情 548

泥(相对少地)所能节省的磁盘存取，关系外排序的磁盘块传输的总数为：

$$b_r(2\lceil \log_{M-1}(b_r/M)\rceil + 1)$$

把该公式用到图 12-4 所示的例子上，我们算出共需 $12 * (4+1) = 60$ 次块传输，这一结果可以在图 12-4 中得以验证。注意，上面的值不包括将最后结果写到外存的开销。

此外，我们还要加上磁盘搜索的代价。在产生归并段阶段需要为读取每个归并段的数据作磁盘搜索，也要为写回归并段作磁盘搜索。在归并阶段，如果每次从一个归并段读取 b_b 块数据(也就是说，把 b_b 个缓冲块分配给每个归并段)，则每一趟归并需要外$\lceil b_r/b_b\rceil$次磁盘搜索以读取数据[⊖]。尽管输出结果是顺序写回磁盘的，如果它和输入归并段在一个磁盘块上，磁盘头在写回连续的块的间隔中可能已经移到别处。这样我们需要为每趟归并加上总共$2\lceil b_r/b_b\rceil$次磁盘搜索，除了最后一趟以外(因为我们假定最终结果不写回磁盘)。假设输出阶段也分配了 b_b 个块，每一趟可以归并$\lfloor M/b_b\rfloor - 1$个归并段，则磁盘搜索的总次数为：

$$2\lceil b_r/M\rceil + \lceil b_r/b_b\rceil(2\lceil \log_{\lfloor M/b_b\rfloor - 1}(b_r/M)\rceil - 1)$$

如果我们把分配给每个归并段的缓冲块数 b_b 设为 1，把该公式用到图 12-4 所示的例子上，我们算出共需 $8 + 12 * (2 * 2 - 1) = 44$ 次磁盘搜索。

12.5 连接运算

这一节将探讨计算关系连接的几种算法，并分析各种算法的代价。

我们用**等值连接**(equi-join)这个词来表示形如 $r \bowtie_{A=B} s$ 的连接，其中 A、B 分别为关系 r 与 s 的属性或属性组。

我们使用下面的表达式作为例子：

549

$$student \bowtie takes$$

我们就用在第 2 章用过的同样的关系模式，假定关于这两个关系有如下信息：

- *student* 的记录数：$n_{student} = 5000$
- *student* 的磁盘块数：$b_{student} = 100$
- *takes* 的记录数：$n_{takes} = 10\,000$
- *takes* 的磁盘块数：$b_{takes} = 400$

12.5.1 嵌套循环连接

图 12-5 给出了一个计算两个关系 r 和 s 的 θ 连接 $r \bowtie_\theta s$ 简单算法。由于该算法主要由两个嵌套的 for 循环构成，因此它称为**嵌套循环连接**(nested-loop join)。由于算法中有关 r 的循环包含有关 s 的循环，因此关系 r 称为连接的**外层关系**(outer relation)，而 s 称为连接的**内层关系**(inner relation)。该算法使用了 $t_r \cdot t_s$ 这个记号，其中 t_r 和 t_s 表示 r、s 的元组，$t_r \cdot t_s$ 表示将 t_r 和 t_s 元组的属性值拼接而成的一个元组。

与选择算法中使用的线性文件扫描算法类似，嵌套循环连接算法不要求有索引，并且不管连接条件是什么，该算法均可使用。对此算法进行扩展来计算自然连接也是简单明了的，因为自然连接可表示为一个 θ 连接然后做去掉重复属性的投影运算。唯一需要的修改是在将 $t_r \cdot t_s$ 放入结果集之前删除 $t_r \cdot t_s$ 的重复属性。

```
for each 元组 t_r in r do begin
        for each 元组 t_s in s do begin
            测试元组对(t_r, t_s)是否满足连接条件 θ
            如果满足，把 t_r · t_s 加到结果中
        end
    end
```

图 12-5 嵌套循环连接

⊖ 更精确一点，由于归并段都是分开读取的，在读到归并段末尾的时候可能会读到小于 b_b 个块。因此我们也许对每个段都需要一次额外的磁盘搜索。简单起见我们省略了这个细节。

嵌套循环连接算法的代价很大，因为该算法为检查两个关系中的每一对元组。考虑嵌套循环连接算法的代价，所需考虑的元组对数目是 $n_r * n_s$，这里 n_r 指 r 中的元组数，n_s 指 s 中的元组数。对于关系 r 中的每一条记录，我们必须对 s 作一次完整的扫描。在最坏的情况下，缓冲区只能容纳每个关系的一个数据块，这时共需 $n_r * b_s + b_r$ 次块传输，这里 b_r 和 b_s 分别代表包含关系 r 和 s 中元组的磁盘块数。对每次扫描内层关系 s 我们只需一次磁盘搜索，因为它的数据是顺序读取的，读取关系 r 一共需要 b_r 次磁盘搜索。这样得到总的磁盘搜索次数为 $n_r + b_r$。在最好的情况下，内存有足够空间同时容纳两个关系，此时每一数据块只需读一次，从而只需 $b_r + b_s$ 次块传输，加上两次磁盘搜索。 550

如果其中一个关系能完全放在内存中，那么把这个关系作为内层关系来处理是有好处的。因为这样内层循环关系只需要读一次。所以，如果 s 小到可以装入内存，那么我们的策略只需 $b_r + b_s$ 次块传输和两次磁盘搜索——其代价与两个关系能同时装入内存的情形相同。

现在考虑 *student* 与 *takes* 的自然连接。暂时假设两个关系中没有任何索引可以利用，并且也不想创建任何索引。我们可以用嵌套循环来计算连接。假定连接中 *student* 是外层关系，*takes* 是内层关系。这时我们需要检查 $5000 * 10000 = 50 * 10^7$ 个元组。在最坏的情况下，块传输次数是 $5000 * 400 + 100 = 2\,000\,100$，加上 $5000 + 100 = 5100$ 次磁盘搜索；然而在最好的情况下，两个关系只需要读一次，然后进行计算，其代价最多只要 $100 + 400 = 500$ 次块传输和两次磁盘搜索——大大优于最坏的情况。如果我们把 *takes* 作为外层关系，把 *student* 作为内层关系，此策略在最坏情况下需要 $10000 * 100 + 400 = 1\,000\,400$ 次块传输加上 10 400 次磁盘搜索。块传输的数目显著地减少了，不过磁盘搜索的次数增加了。不过假定 $t_S = 4$ 毫秒，$t_T = 0.1$ 毫秒，总的时间代价还是减少了。

12.5.2 块嵌套循环连接

因缓冲区太小而内存不能完全容纳任何一个关系时，如果我们以块的方式而不是以元组的方式处理关系，仍然可以减少不少块读写次数。图 12-6 所示过程是**块嵌套循环连接**(block nested-loop join)，它是嵌套循环连接的一个变种，其中内层关系的每一块与外层关系的每一块形成一对。在每个块对中，一个块中的每一个元组与另一块的每一元组形成元组对，得到全体元组对。和前面一样，把满足连接条件的所有元组对添加到结果中。

```
for each 块 B_r of r do begin
    for each 块 B_s of s do begin
        for each 元组 t_r in B_r do begin
            for each 元组 t_s in B_s do begin
                测试元组对(t_r, t_s)是否满足连接条件
                如果满足，把 t_r · t_s 加到结果中
            end
        end
    end
end
```

图 12-6 块嵌套循环连接

块嵌套循环连接与基本的嵌套循环连接算法代价的主要差别在于：在最坏的情况下，对于外层关系中的每一个块，内层关系 s 的每一块只须读一次，而不是对外层关系的每一个元组读一次。因此， 551
在最坏的情况下，共需 $b_r * b_s + b_r$ 次块传输，这里 b_r 和 b_s 分别代表含有关系 r 和 s 中元组的磁盘块数。对内层关系的每一次扫描需要一次磁盘搜索，对外层关系的扫描需要每块一次磁盘搜索，这样总共是 $2b_r$ 次磁盘搜索。显然，如果内存不能容纳任何一个关系，则使用较小的关系作为外层关系更有效。在最好的情况下，内存能够容纳内层关系，需要 $b_r + b_s$ 次块传输加上两次磁盘搜索(在这种情况下我们把较小的关系作为内层关系)。

回到计算 *student* × *takes* 的例子，考虑用块嵌套循环连接算法来计算。在最坏的情况下，我们必须为 *student* 的每一个块读取 *takes* 的所有块，因此，在最坏的情况下，共需 $100 * 400 + 100 = 40\,100$ 次块传输加上 $2 * 100 = 200$ 次磁盘搜索。这个代价与采用基本嵌套循环连接算法所需的 $5000 * 400 + 100 =$

2 000 100次块传输和 5100 次磁盘搜索相比，是一个显著的改进，而最好的情况下的代价和原来一样——100 + 400 = 500 次块传输和两次磁盘搜索。

嵌套循环与块嵌套循环算法的性能可以进一步地改进：

- 如果自然连接或等值连接中的连接属性是内层关系的码，则对每个外层关系元组，内层循环一旦找到了首条匹配元组就可以终止。
- 在块嵌套循环连接算法中，外层关系可以不用磁盘块作为分块的单位，而以内存中最多能容纳的大小为单位，当然同时要留出足够的缓冲空间给内层关系及输出结果使用。也就是说，如果内存有 M 块，我们一次读取外层关系中的 $M-2$ 块，当我们读取到内层关系中的每一块时，我们把它与外层关系中的所有 $M-2$ 块做连接。这种改进使内层关系的扫描次数从 b_r 次减少到 $\lceil b_r/(M-2)\rceil$ 次，这里的 b_r 是外层关系所占的块数。这样全部代价为 $\lceil b_r/(M-2)\rceil * b_s + b_r$ 次块传输和 $2\lceil b_r/(M-2)\rceil$ 次磁盘搜索。
- 对内层循环轮流做向前、向后的扫描。该扫描方法对磁盘块读写请求进行排序，使得上一次扫描时留在缓冲区中的数据可以重用，从而减少磁盘存取次数。
- 若内层循环的连接属性上有索引，可以用更有效的索引查找法替代文件扫描法。这一改进在 12.5.3 节讲述。

12.5.3 索引嵌套循环连接

在嵌套循环连接（见图 12-5）中，若在内层循环的连接属性上有索引，则可以用索引查找替代文件
552 扫描。对于外层关系 r 的每一个元组 t_r，可以利用索引查找 s 中和元组 t_r 满足连接条件的元组。

上述连接方法称为索引嵌套循环连接（indexed nested-loop join），它可以在已有索引或者为了计算该连接而专门建立临时索引的情况下使用。

在关系 s 中查找和给定元组 t_r 满足连接条件的元组本质上是在 s 上做选择运算。例如，考虑 $student \bowtie takes$。假设 *student* 中有一个 *ID* 为"00128"的元组，那么 *takes* 中相应的元组就是那些满足选择条件 *ID* = "00128"的元组。

索引嵌套循环连接的代价可以如下计算。对于外层关系 r 的每一个元组，需要在关系 s 的索引上进行查找，检索相关元组。在最坏的情况下，缓冲区只能容纳关系 r 的一块和索引的一块。此时，读取关系 r 需 b_r 次 I/O 操作，这里的 b_r 指包含关系 r 中记录的磁盘块数。每次 I/O 操作需要一次磁盘搜索和一次块传输，因为磁盘头可能在 I/O 操作的间隙中移动过。对于关系 r 中的每个元组，在 s 上进行索引查找。这样，连接的时间代价可用 $b_r(t_T + t_S) + n_r * c$ 来计算，其中 n_r 是关系 r 中的记录数，c 是使用连接条件对关系 s 进行单次选择操作的代价。在 12.3 节中我们已经知道对单个选择算法（可能使用索引）如何估计代价，这可以使我们估算出 c 的值。

代价计算公式表明，如果两个关系 r、s 上均有索引时，一般把元组较少的关系作外层关系时效果较好。

例如，考虑 $student \bowtie takes$ 的索引嵌套循环连接，其中 *student* 作外层关系。假设关系 *takes* 在连接属性 *ID* 上有 B+ 树主索引，平均每个索引结点包含 20 个索引项。由于 *takes* 有 10 000 个元组，因此树高度为 4，存取实际数据还需要一次磁盘访问。又由于 $n_{student}$ 是 5000，因此总代价为 100 + 5000 * 5 = 25 100次磁盘访问，其中每次访问都需要一次磁盘搜索和一次磁盘块传输。反之，正如我们此前看到的，一次块嵌套循环连接操作需要 40 100 次块传输和 200 次磁盘搜索。尽管块传输的次数减少了，但磁盘搜索的代价增加了。这样总的代价还是增加了，因为一次磁盘搜索的代价比一次块传输要高。然而，如果我们在 *student* 关系上有一个选择操作使得行数显著地减少，索引嵌套循环连接可以比块嵌套循环连接快得多。

12.5.4 归并连接

归并连接（merge join）算法（又称**排序 - 归并 - 连接**（sort-merge join）算法）可用于计算自然连接和等值连接。令 $r(R)$、$s(S)$ 表示要执行自然连接运算的关系，并令 $R \cap S$ 表示两个关系的公共属性。假定两个关系均按属性 $R \cap S$ 排序，那么它们的连接可用与归并排序算法中的归并阶段非常类似的处理过程来计算。

12.5.4.1 归并算法

归并连接算法如图 12-7 所示。在这个算法中，*JoinAttrs* 表示 $R \cap S$ 中的属性；$t_r \bowtie t_s$ 表示具有相同 *JoinAttrs* 属性值的两个元组 t_r、t_s 的拼接，接着通过投影去除其中重复的属性。归并连接算法为每个关系分配一个指针。这些指针一开始指向相应关系的第一个元组。随着该算法的进行，指针遍历整个关系。一个关系中在连接属性上具有相同值的一组元组被读入到 S_r 中。图 12-7 所示算法要求每个 S_r 元组集合都能装入主存；本节稍后将讲述如何扩展该算法以避免这一限制。接下来，另一关系中相应的元组(如果有的话)被读入，并在读入的同时加以处理。 553

```
pr := r 的第一个元组的地址;
ps := s 的第一个元组的地址;
while ( pr ≠ null and ps ≠ null ) do
    begin
        t_r := pr 所指向的元组;
        S_r := { t_r };
        让 ps 指向关系 s 的下一个元组;
        done := false;
        while ( not done and ps ≠ null ) do
        begin
            t_s := ps 所指向的元组;
            if ( t_r[JoinAttrs] = t_s[JoinAttrs] )
                then begin
                    S_r := S_r ∪ { t_s };
                    让 ps 指向关系 s 的下一个元组;
                end
            else done := true;
        end
        t_s := ps 所指向的元组;
        while ( ps ≠ null and t_s[JoinAttrs] < t_r[JoinAttrs] ) do
            begin
                让 ps 指向关系 s 的下一个元组;
                t_s := ps 所指向的元组;
            end
        while ( ps ≠ null and t_s[JoinAttrs] = t_r[JoinAttrs] ) do
            begin
                for each t_s in S_r do
                begin
                    将 t_r ⋈ t_s 加入结果中;
                end
                让 ps 指向关系 s 的下一个元组;
                t_s := ps 所指向的元组;
            end
    end
```

图 12-7 归并连接 554

图 12-8 给出了在连接属性 $a1$ 上排序的两个关系。利用图 12-7 中的两个关系具体地走一遍归并连接算法是很有启发性的。

图 12-7 所示的归并连接算法要求主存能容纳在连接属性上有相同值的元组集 S_r。即使关系 s 很大，这个要求也通常可以满足。如果对于某些连接属性值，S_r 太大而不可用内存，则可以在集合 S_r 与关系 r 中具有相同连接属性值的元组之间采用块嵌套循环连接。

如果任意一个输入关系 r 或 s 未被连接属性排序，那么可以先对它们进行排序，然后再使用归并连接算法。归并连接算法也可以很容易地从自然连接拓展到更一般的等值连接。

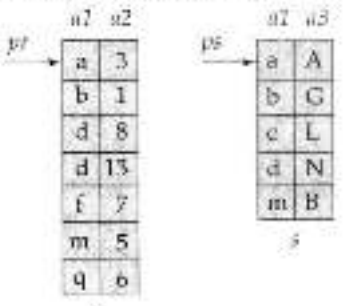


图 12-8 在归并连接中使用的已排序关系

12.5.4.2 代价分析

一旦关系已排序，在连接属性上有相同值的元组是连续存放的。所以已排序的每一元组只须读一次，因而每一块也只须读一次。由于两个文件都只须读一遍(假设所有集合 S_i 均可装入内存)，因此可知归并连接算法是高效的。所需磁盘块传输次数是两个文件块数之和：$b_r + b_s$。

假设为每个关系分配 b_b 个缓冲块。那么所需磁盘搜索次数为 $\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$。由于磁盘搜索代价远比数据传输高，假设还有额外的内存，因此为每个关系分配多个缓冲块是有意义的。例如，假设对于每个 4KB 的块，$t_T = 0.1$ 毫秒，$t_S = 4$ 毫秒，缓冲区大小为 400 块(或者 1.6 MB)，则每 4 毫秒的磁
555 盘搜索时间对应每 40 毫秒的传输时间。换句话说，磁盘搜索时间将只占传输时间的 10%。

如果任意一个输入关系 r 或 s 未按连接属性排序，那么必须先对它们进行排序，排序代价必须加在上述所有的代价上。假如一些集合 S_i 不能装入内存，代价将有轻微的增加。

假设将归并连接算法应用到我们的 *student* ⋈ *takes* 例子上，连接属性是 *ID*；假定两个关系已按连接属性 *ID* 排序。在这种情况下，归并连接需要 400 + 100 = 500 次块传输。如果我们假设在最坏情况下每个输入关系仅分配到一个缓冲块(即 $b_b = 1$)，则总共也需要 400 + 100 = 500 次磁盘搜索。实际上 b_b 值可以设得远比这个值高，因为我们仅需要对两个关系进行缓冲，所以磁盘搜索的代价远比上述的要小。

假设两个关系未排序，内存大小也属于最差情形：只有 3 块。代价计算如下所示：

1. 使用我们在 12.4 节得到的公式对 *takes* 进行排序需要 $\lceil \log_{3-1}(400/3) \rceil = 8$ 遍归并。则对 *takes* 关系进行排序需要 $400 * (2 * \lceil \log_{3-1}(400/3) \rceil + 1) = 6800$ 次块传输，再加上将结果写回的 400 次块传输。排序所需磁盘搜索的次数是 $2 * \lceil 400/3 \rceil + 400 * (2 * 8 - 1) = 6268$ 次，加上写回结果需要 400 次磁盘搜索，总共是 6668 次磁盘搜索，因为每个归并段只能分配到一个缓冲块。

2. 类似地对 *student* 运行排序需要 $\lceil \log_{3-1}(100/3) \rceil = 6$ 遍归并，即 $100 * (2 \lceil \log_{3-1}(100/3) \rceil + 1) = 1300$ 次块传输，还有将结果写回的 100 次块传输。对 *student* 进行排序所需的磁盘搜索次数为 $2 * \lceil 100/3 \rceil + 100 * (2 * 6 - 1) = 1164$，另外写回结果需要 100 次磁盘搜索，所以总共是 1264 次搜索。

3. 最后，归并这两个关系需要 400 + 100 = 500 次块传输和 500 次磁盘搜索。

因此，在关系未排序，内存大小仅能容纳 3 块的情况下，总共需要 9100 次块传输和 8932 次磁盘搜索。

如果内存大小能容纳 25 个磁盘块，关系未排序，则先排序然后归并连接的代价如下所示：

1. 对关系 *takes* 的排序可以只用一个归并步骤完成，总共花费 $400 * (2 \lceil \log_{25-1}(400/25) \rceil + 1) = 1200$ 次磁盘块传输。类似地，对关系 *student* 的排序需要 300 次磁盘块传输。把排序结果写回磁盘需要 400 + 100 = 500 次磁盘块传输，并且归并步骤还需要 500 次磁盘块传输以读回数据。把所有这些代价加起来一共是 2500 次磁盘块传输。

2. 如果我们假设每个归并段仅分配一个缓冲块，在这种情况下对 *takes* 关系进行排序以及把排序结果写回磁盘，所需的磁盘搜索次数是 $2 * \lceil 400/25 \rceil + 400 + 400 = 832$ 次。类似地，对关系 *student* 来说需要 $2 * \lceil 100/25 \rceil + 100 + 100 = 208$ 次磁盘搜索，再加上归并阶段读取已排序数据的 400 + 100 次磁盘搜
556 索，把这些代价加起来得到总代价为 1640 次磁盘搜索。

通过为每个归并段分配更多的缓冲块，磁盘搜索的次数能够显著地减少。例如，如果为每个归并段和输出结果保留 5 个缓冲块，则归并 *student* 的 4 个归并段的代价从 208 次磁盘搜索降到 $2 \times \lceil 100/25 \rceil + \lceil 100/5 \rceil + \lceil 100/5 \rceil = 48$ 次磁盘搜索。如果归并连接阶段为 *take* 和 *student* 保留 12 个缓冲块，则这一阶段的磁盘搜索次数将从 500 次降到 $\lceil 400/12 \rceil + \lceil 100/12 \rceil = 43$ 次。于是磁盘搜索的总次数是 251。

这样，如果关系未排序且内存大小是 25 块，总代价是 2500 次磁盘块传输加上 251 次磁盘搜索。

12.5.4.3 混合归并连接

当两个关系的连接属性上存在辅助索引时，可以对未排序的元组执行归并连接运算的变种。该算法通过索引扫描记录，从而按顺序检索记录。但这种变种有很大的缺陷，因为记录可能分散在文件的多个块中，从而每个元组的存取都可能导致一次磁盘访问，代价很大。

为避免这种代价，我们可以使用一种混合归并 - 连接技术，该技术把索引与归并连接相结合。假设两个关系中有一个排了序，另一个未排序，但未排序的关系在连接属性上有 B^+ 树辅助索引。混合归并 - 连接算法(hybrid merge-join algorithm)把已排序关系与 B^+ 树辅助索引叶结点进行归并。所得结果文件包含了已排序关系的元组及未排序关系的元组地址。然后，将该文件按未排序关系元组的地址进行

排序，从而能够对相关元组按照物理存储顺序进行有效的检索，最终完成连接运算。扩展这个技术用于处理两个未排序关系的工作留作练习。

12.5.5 散列连接

类似于归并连接算法，散列连接算法可用于实现自然连接和等值连接。在散列连接算法中，用散列函数 h 来划分两个关系的元组。此算法的基本思想是把这两个关系的元组划分成在连接属性值上具有相同散列值的元组集合。

我们假设：

- h 是将 *JoinAttrs* 值映射到 $\{0, 1, \cdots, n_h\}$ 的散列函数，其中 *JoinAttrs* 表示自然连接中 r 与 s 的公共属性。
- $r_0, r_1, \cdots, r_{n_h}$ 表示关系 r 的元组的划分，一开始每个都是空集。每个元组 $t_r \in r$ 被放入划分 r_i 中，其中 $i = h(t_r[JoinAttrs])$。
- $s_0, s_1, \cdots, s_{n_h}$ 表示关系 s 的元组的划分，一开始每个都是空集。每个元组 $t_s \in s$ 被放入划分 s_i 中，其中 $i = h(t_s[JoinAttrs])$。

557

散列函数 h 应当具有在第 11 章讨论过的"良好"特性——随机性和均匀性。关系的划分如图 12-9 所示。

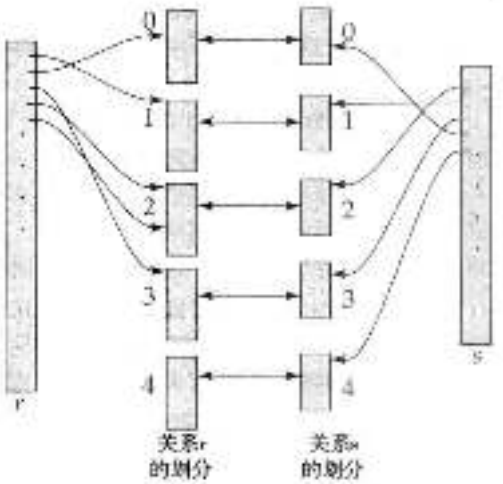


图 12-9 关系的散列划分

12.5.5.1 基本思想

散列连接算法背后的思想如下。如果关系 r 的一个元组与关系 s 的一个元组满足连接条件，那么它们在连接属性上就会有相同的值。若该值经散列函数映射到 i，则关系 r 的那个元组必在 r_i 中，而关系 s 的那个元组必在 s_i 中。因此，r_i 中的元组 r 只需要与 s_i 中的元组 s 相比较，而没有必要与其他任何划分里的元组 s 相比较。

例如，如果 d 是 *students* 中的一个元组，c 是 *takes* 中的一个元组，h 是元组属性 *ID* 上的散列函数，那么只有在 $h(c) = h(d)$ 时 d 与 c 才须比较。若 $h(c) \neq h(d)$，则 d 与 c 在属性 *ID* 上的值必不相等。然而，如果 $h(c) = h(d)$，我们必须检查 d 与 c 在连接属性上的值是否相同，因为可能 d 与 c 有不同的 *ID* 值却有相同的散列值。

```
/* 对关系 s 进行划分 */
for each 元组 t_s in s do begin
    i_s = h(t_s[JoinAttrs]);
    H_si = H_si ∪ {t_s};
end
/* 对关系 r 进行划分 */
for each 元组 t_r in r do begin
    i_r = h(t_r[JoinAttrs]);
    H_ri = H_ri ∪ {t_r};
End
/* 对每一划分进行连接 */
for i := 0 to n_h do begin
    读 H_si 在内存中建立其散列索引;
    for each 元组 t_r in r_i do begin
        检索 s_i 的散列索引，定位所有满足 t_s[JoinAttrs] = t_r[JoinAttrs] 的元组 t_s
        for each 匹配的元组 t_s in H_si do begin
            将 t_r 与 t_s 加入结果中;
        end
    end
end
```

图 12-10 散列连接

图 12-10 显示了**散列连接**(hash join)算法计算关系 r 与关系 s 自然连接的详细过程。和归并连接算法一样，$r_i \times r_j$ 表示元组 r_i、r_j 属性的拼接，然后对其投影除去重复属性。对关系进行划分后，散列连接算法的其余部分对各个划分对 $s_i(i=0, 1, \cdots, n_h)$ 进行单独的索引嵌套循环连接。为此，该算法先为每个 s_i 构造(build)散列索引，然后用 r_i 中的元组进行**探查**(probe)(即在 s_i 中查找)。关系 s 称为**构造用输入**(build input)，关系 r 称为**探查用输入**(probe input)。

s_i 的散列索引是在内存中建立的，因此并不需要访问磁盘以检索元组。用于构造此索引的散列函数与前面使用的散列函数 h 必须是不同的，但仍是只对连接属性进行散列映射。在进行索引嵌套循环连接时，系统使用该索引检索那些与探查用输入相匹配的记录。

构造阶段与探查阶段只需要对构造用输入与探查用输入作一次扫描。将散列连接算法推广到计算更普遍的等值连接是很容易的。

应选择足够大的 n_h 值，以使对于任意的 i，内存中可以容纳构造用输入关系的划分 s_i 中的元组以及划分上的散列索引。内存中可以不必容纳探查关系的划分。显然，我们最好用较小的输入关系作为构造用输入关系。如果构造用关系有 b_s 块，那么要使 n_h 个划分中每一个划分小于等于 M，n_h 值至少应是 $\lceil b_s/M \rceil$。更准确些，我们还应当考虑该划分上散列索引占用的内存空间，因此 n_h 值应当相应取大一点。简单起见，在分析中我们有时忽略散列索引所需的内存空间。

12.5.5.2 递归划分

如果 n_h 的值大于或等于内存块数，因为没有足够的缓冲块，所以关系的划分不可能一趟完成。这
558 | 559 时，完成关系的划分需要重复多趟。在每一趟中，输入的最大划分数不超过用于输出的缓冲块数。每一趟产生的存储桶在下一趟中分别被读入并再次划分，产生更小的划分。当然，每趟划分中所用的散列函数与上一趟所用的散列函数是不同的。系统不断重复输入的分裂过程直到构造用输入关系的每个划分都能被内存容纳为止。这种划分方法称为**递归划分**(recursive partitioning)。

在 $M > n_h + 1$，或等价地，$M > (b_s/M) + 1$——可以近似简化为 $M > \sqrt{b_s}$ 时，关系不需要进行递归划分。例如，考虑如下情形：内存大小是 12MB，分成 4KB 大小的块，则共有 3K(3072)个块。我们可用这样大小的内存对含有大至 3K × 3K(即 36GB)的关系进行划分。类似地，为避免递归划分，1GB 大小的关系需 $\sqrt{256K}$ 个内存块，约 2MB。

12.5.5.3 溢出处理

当 s_i 的散列索引大于主存时，构造用输入关系 s 的划分 i 发生**散列表溢出**(hash-table overflow)。如果构造用输入关系在连接属性上具有相同值的元组数很多，或所选散列函数不符合随机性、均匀性，则会发生散列表溢出。在这两种情况下，某些划分所含元组数多于平均数，而另一些划分所含元组数比平均数少；这种划分称为是**偏斜的**(skewed)。

少量的偏斜可以通过增加划分个数，使得每个划分的期望大小(包括该划分上的散列索引在内)比内存容量略小。划分数会因此有少量增加，增加的数目称为**避让因子**(fudge factor)。这个因子通常大概是 12.5.5 节所描述的方法计算出的散列划分数的 20%左右。

即使我们通过用避让因子，在划分的大小上采取了保守的态度，散列表溢出仍然在所难免。散列表溢出可以通过**溢出分解**或**溢出避免**的方法来进行处理。如果在构造阶段发现了散列索引溢出，则进行**溢出分解**(overflow resolution)。溢出分解过程如下。任给 i，若发现 s_i 太大，我们就用另一个散列函数将之进一步划分成更小的划分。类似地，r_i 也使用新的散列函数进行同样处理。而只有相匹配的划分中的元组才需要连接。

与溢出分解法不同，**溢出避免**(overflow avoidance)法进行谨慎的划分，保证构造阶段不会有溢出发生。在溢出避免法中，首先将构造用输入关系 s 划分成许多小划分，然后把某些划分组合在一起，但要确保每个组合后的划分都能被内存容纳。探查用关系 r 必须按照与关系 s 上的组合划分相同的方式进行划分，但 r 的大小无关紧要。

如果 s 中有大量元组在连接属性上有相同的值，溢出分解与溢出避免法在某些划分上可能会失效。
560 | 在这种情况下，我们并不采用创建内存散列索引，然后用嵌套循环连接算法对划分进行连接的方法，而是在那些划分上使用其他的连接技术，如块嵌套循环连接。

12.5.5.4 散列连接的代价

下面我们考虑散列连接的代价。我们的分析假定没有散列表溢出发生。首先考虑不需要递归划分的情形。

- 两个关系 r、s 的划分需要对这两个关系分别进行一次完整的读入与写回，该操作需要 $2(b_r + b_s)$ 次块传输，这里 b_r 和 b_s 分别代表含有关系 r 和 s 中元组的磁盘块数。在构造与探查阶段每个划分读入一次，这又需要 $b_r + b_s$ 次块传输。划分所占用的块数可能比 $b_r + b_s$ 略多，因为有的块只是部分满的。由于 n_h 个划分中的每个划分都可能有一个部分满的块，而这个块需写回、读入各一次，因此对于每个关系而言存取这些部分满的块至多增加 $2 * n_h$ 次的开销。从而散列连接的代价估计需要：

$$3(b_r + b_s) + 4n_h$$

次块传输。其中 $4n_h$ 的开销与 $b_r + b_s$ 相比是很小的，可以忽略。

- 假设为输入缓冲区和每个输出缓冲区分配了 b_b 个块，划分总共需要 $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$ 次磁盘搜索。在构造和探查阶段，每个关系中 n_h 个划分中的每一个仅需一次磁盘搜索，因为每个划分都可以顺序地读取。这样，散列连接需要 $2\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil + 2n_h$ 次磁盘搜索。

现在来看看需要递归划分的情形。每一遍预计可将划分的大小减小为原来的 $1/(M-1)$；不断重复操作直到每个划分最多占 M 块为止。则划分关系 s 所需的遍数预计为 $\lceil \log_{M-1}(b_s) - 1 \rceil$。

- 由于在每一遍中，需要对关系 s 的每一块进行读入、写出，因此划分过程中总共需要 $2b_s\lceil \log_{M-1}(b_s) - 1 \rceil$ 次块传输。对关系 r 进行划分所需遍数与划分关系 s 是一样的，由此连接代价估计需要：

$$2(b_r + b_s)\lceil \log_{M-1}(b_s) - 1 \rceil + b_r + b_s$$

次块传输。 [561]

- 再次假设为每个划分分配了 b_b 个块用于缓冲。忽略构造和探查过程中相对少的磁盘搜索，用递归划分的散列连接需要

$$2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) \log_{M-1}(b_s) - 1$$

次磁盘搜索。

例如，考虑连接 *takes* ⋈ *student*。内存有 20 块，*student* 划分成 5 个划分，每个划分占 20 块，正好能装入内存。划分只需一遍。*takes* 类似地划分成 5 个划分，每个划分占 80 块。忽略写回部分满的块的代价，共需 $3(100 + 400) = 1500$ 次块传输。在划分过程中有足够的内存分配 3 个输入缓冲区和 5 个输出缓冲区，结果是 $2(\lceil 100/3 \rceil + \lceil 400/3 \rceil)$ 次磁盘搜索。

若主存较大，散列连接性能可以得到提高。当主存中可以容纳整个构造用输入关系时，n_h 可置为 0；此时，不管探查用输入的大小如何，不必将关系划分成临时文件，因而散列连接算法可以快速执行。其估计代价降为 $b_r + b_s$ 次磁盘块传输和两次磁盘搜索。

12.5.5.5 混合散列连接

混合散列 - 连接（hybrid hash-join）算法作了另一种优化。当内存相对较大，但还不足以容纳整个构造用输入关系时，该算法是很有用的。在划分阶段，散列连接算法需要为创建的每一划分提供一个内存块作为缓冲区，另需一个内存块作为输入缓冲区。为了降低搜索的影响，大的块数用作缓冲区，令 b_b 表示用作输入和每个划分的缓冲区的块数。因此划分两个关系共需 $n_h + 1$ 个内存块。如果内存大于 $(n_h + 1) * b_b$ 块，我们可以用剩余的内存块（$M - (n_h + 1) * b_b$ 块）作为构造用输入关系的第一个划分（即 s_0）的缓冲区，从而避免将其写出后再读入。更进一步，我们可以适当设计散列函数，使得 s_0 上的散列索引能装入 $M - (n_h + 1) * b_b$ 个内存块；这样，在关系 s 划分结束后，s_0 完全在内存中且可以在 s_0 上建立散列索引。

当对 r 进行划分时，r_0 的元组也不必写回磁盘；而是在元组产生时，系统利用它们去查找驻留在内存中的 s_0 散列索引，并产生连接后的元组。r_0 的元组使用后就可以抛弃了，因此 r_0 划分不占内存空间。这样，对于 r_0 与 s_0 的每一块可以读、写各节省一次。其他划分的元组按普通方式写出，以后再连接。在构造用输入关系只是稍大于内存时，混合散列连接算法可以大大降低代价。

若构造用输入关系大小为 b_r，则 n_r 近似为 b_r / M。因此，混合散列连接算法在 $M >> (b_r / M) + b_s$ 或 $M >> \sqrt{b_r * b_s}$ 时非常有用，记号 $>>$ 表示远大于。例如，设块大小为4KB，构造用输入关系大小为5GB，b_r 是20。那么混合连接算法在内存远大于20MB时是很有用的；现今计算机拥有几GB或更多的内
562 存是很常见的。假如我们用1GB来做连接算法，n_r 接近1GB，混合散列索引比散列索引节省20%。

12.5.5.6 复杂连接

嵌套循环连接与块嵌套循环连接可在任何连接条件下使用。其他的连接技术比嵌套循环连接及其变种效率更高，但只能处理简单的连接条件，如自然连接或等值连接。如果采用12.3.3节中用于处理复杂选择的技术，则可以用有效的连接技术实现具有复杂连接条件的连接，如合取式和析取式。

考虑下面带有合取条件的连接：

$$r \bowtie_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n} s$$

前面所述的一种或多种连接技术可以用于单个连接条件的计算，如 $r \bowtie_{\theta_1} s$、$r \bowtie_{\theta_2} s$、$r \bowtie_{\theta_3} s$ 等。我们可以通过先计算这些较简单的 $r \bowtie_{\theta_i} s$ 连接中的某一个来计算整个连接：每个中间结果中的元组对由 r 中的一个元组与 s 中的一个元组组成。完整的连接结果是中间结果中满足以下剩余条件的那些记录：

$$\theta_1 \wedge \cdots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \cdots \wedge \theta_n$$

这些条件可在产生 $r \times s$ 元组时进行测试。

具有析取条件的连接可按如下方式计算。考虑：

$$r \bowtie_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n} s$$

该连接可以通过对单个连接 $r \bowtie_{\theta_i} s$ 的元组取并集来计算：

$$(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \cdots \cup (r \bowtie_{\theta_n} s)$$

12.6节将讲述计算关系的并的算法。

12.6 其他运算

其他关系运算以及扩展关系运算，如去除重复、投影、集合运算、外连接、聚集，可以按从12.6.1－12.6.5节所述加以实现。

12.6.1 去除重复

我们可以用排序方法很容易地实现去除重复。排序时等值元组相互邻近，删除其他副本只留一个元组副本即可。对于外部归并排序而言，创建归并段文件时就可以发现重复元组，可在将归并段文件写
563 回磁盘之前去除重复元组，从而减少块传输次数。剩余的重复元组可在归并时去除。最后的经排序的归并段是没有重复元组的。最坏情形去除重复的代价估计与最坏情形对该关系进行排序的代价估计一样。

我们也可使用散列来实现去除重复，像在散列连接算法里一样。首先，基于整个元组上的一个散列函数对整个关系进行划分。接下来每个划分被读入内存，并建立内存散列索引。在创建散列索引时，只插入不在索引中的元组；否则，元组就被抛弃。划分中的所有元组处理完后，散列索引中的元组被写到结果中。其代价估算与散列连接中构造用输入关系的处理（划分以及读入每个划分）的代价一样。

由于去除重复的代价相对较大，因此SQL查询语言要求用户显式指明需要去除重复。若不指明则保留重复。

12.6.2 投影

我们可以通过如下方式比较容易地实现投影。首先对每个元组作投影，所得结果关系可能有重复记录，然后去除重复记录。去除重复可按12.6.1节描述的方法去进行。若投影列表中属性含有关系的码，则结果中不会有重复元组，因此不必作去除重复。广义投影可以用与投影一样的方式来实现。

12.6.3 集合运算

要实现集合运算并、交、差，我们首先对两个关系进行排序，然后对每个已排序的关系扫描一次，产生所需结果。在 $r \cup s$ 中，当同时对两个文件进行扫描发现相同的元组时，只保留其中一个。$r \cap s$ 的结果只包含在两个关系中同时出现的元组。类似地，通过只保留 r 中那些不属于 s 的元组，我们可以实现集合的差运算 $(r - s)$。

对所有这些运算，两个输入关系仅须扫描一次。因此如果关系按相同顺序排序，其代价为 $b_r + b_s$ 次块传输。假设最坏情况下每个关系只有一个缓冲块，则总共需要 $b_r + b_s$ 次磁盘搜索加上 $b_r + b_s$ 次块传输。分配额外的缓冲块可以减少磁盘搜索的次数。

若关系一开始未排序，还要考虑排序的代价。在执行集合运算时，任何排序顺序均可，只要两个输入关系都有相同的排序顺序。

散列提供了实现集合运算的另一种方法。对于每种运算，首先用相同的散列函数对两个关系进行划分，由此得到划分 r_0，r_1，…r_n 以及 s_0，s_1，…s_n。然后根据操作的不同，对 $i = 0$，1，…，n 的每一划分执行以下步骤：

- $r \cup s$
  1. 对 r_i 建立内存散列索引。 564
  2. 把 s_i 中的元组加入以上散列索引中。条件是该元组不在散列索引中。
  3. 把散列索引中的元组加入结果中。
- $r \cap s$
  1. 对 r_i 建立内存散列索引。
  2. 对 s_i 中的每个元组，检索散列索引，仅当它出现在其中时将该元组写到结果中。
- $r - s$
  1. 对 r_i 建立内存散列索引。
  2. 对 s_i 中的每个元组，检索散列索引，若它出现在其中则将之从散列索引中删除。
  3. 把散列索引中剩余的元组加入结果中。

12.6.4 外连接

回想一下4.1.2节所描述的外连接操作。例如，自然左外连接 *takes* 与 *student* 包含 *takes* 与 *student* 的连接；此外，对于 *takes* 中每一个在 *student* 中没有匹配元组的元组 t（即 t 中的 *ID* 不在 *student* 中），把如下的元组 s_t 加入结果中：对于 *takes* 模式中的全部属性，元组 s_t 与 t 有相同的值；s_t 的其余属性（来自 *student* 模式）取空值。

外连接可用以下两种策略之一来实现：

1. 计算相应的连接，然后将适当的元组加入连接结果中以得到外连接结果。考虑左外连接运算与两个关系：$r(R)$ 与 $s(S)$。为计算 $r \mathbin{⟕} s$，首先我们计算 $r \bowtie s$，将结果存为临时关系 q。接着，我们计算 $r - \Pi_R(q)$ 以得到所有属于 r 但未参与 θ 连接的元组。我们可以采用前面说过的用于计算连接、投影、集合差的任何算法来计算外连接。我们用空值填充这些元组中属于关系 s 的属性，然后将其加到 q 中，得到外连接的结果。

右外连接运算 $r \mathbin{⟖} s$ 等价于 $s \mathbin{⟕} r$，因此可以用与左外连接对称的方法实现。要实现全外连接运算 $r \mathbin{⟗} s$，我们可以先计算 $r \bowtie s$，然后和前面一样将左、右外连接的额外元组加入。

2. 对连接算法加以修改。将嵌套循环连接算法扩展为计算左外连接的算法很容易，只要把与内层关系的任何元组都不匹配的外层关系元组在填充空值后写到结果中即可。然而，要将嵌套循环连接扩展为计算完全外连接的算法是很困难的。 565

自然外连接与具有等值连接条件的外连接可以通过扩展归并连接与散列连接算法来计算。对归并连接加以扩展，可以用来计算完全外连接，方法如下：当两个关系的归并完成后，将两个关系中那些与另一个关系的任何元组都不匹配的元组填充空值后写到结果中。类似地，通过只将一个关系中不匹配的元组（在填充空值后）写到结果中，我们可以扩展归并连接以计算左、右外连接。由于关系是排序的，因此很容易判断一个元组是否与另一个关系的元组相匹配。例如，当 *taker* 与 *student* 归并连接完成后，元组按 *ID* 的顺序读入，对于每个元组很容易判断在另一个关系中是否有与之匹配的元组。

使用归并连接算法实现外连接的代价估计同相应连接的代价估计是一样的。唯一差异在于结果的大小以及由此引起的写出结果的块传输次数，而这一点在之前的代价估计中并没有考虑。

扩展散列连接算法以计算外连接留作练习（习题12.15）。

12.6.5 聚集

回忆 3.7 节讲述的聚集函数(操作)。例如，函数

```
select dept_name, avg(salary)
from instructor
group by dept_name
```

计算大学中每个系的平均工资。

聚集运算可以用与去除重复相类似的方法来实现。我们使用排序或散列，就像在去除重复中所做的。不同的是这里基于分组属性进行(上面的例子中是 *dept_name*)。但是，我们不是去除在分组属性上有相同值的元组，而是将之聚集成组，并对每一组应用聚集运算以获取结果。

对 **min**、**max**、**sum**、**count**、**avg** 等聚集函数而言，实现聚集运算的代价估计和去除重复的代价估计是一样的。

我们不必等到所有元组聚集成组后再进行聚集运算，而可以在组的构造过程中就实现 **sum**、**min**、**max**、**count**、**avg** 的聚集运算。对于 **sum**、**min** 与 **max**，当在同一组中发现了两个元组时，系统用包含聚集列上的 **sum**、**min** 或 **max** 值的单个元组替换它们。对于 **count** 运算，系统为每一组维护一个已
[566] 发现元组的计数值。最后，**avg** 运算可以实现如下，在组构造过程中，我们计算每一组的 **sum** 及 **count**，最后把 **sum** 除以 **count** 即获得平均值。

如果结果集的所有元组可以装入内存，则基于排序的实现方法与基于散列的实现方法不必将元组写到磁盘上。当元组读入时，就可以将之插入到一个有序的树结构中或插入到一个散列索引中。当使用以上聚集技术时，对于每一个组只需要保存一个元组，因此，内存中将可容纳有序树结构或散列索引。聚集可在 b_r 次块传输(和 1 次磁盘搜索)中完成，而用其他方法需要 $3b_r$ 次块传输(和最坏情况下 $2b_r$ 次磁盘搜索)。

12.7 表达式计算

目前为止，我们只研究了单个关系运算如何执行。现在我们考虑如何计算包含多个运算的表达式。一种最简易的方法是以适当的顺序每次执行一个操作；每次计算的结果被**物化**(materialized)到一个临时关系中以备后用。这一方法的缺点是需要构造临时关系，这些临时关系必须写到磁盘上(除非很小)。另一种方法是在**流水线**(pipeline)上同时计算多个运算，一个运算的结果传递给下一个，而不必保存临时关系。

12.7.1 – 12.7.2 节将介绍物化方法与流水线方法。我们将会看到这两种方法的代价相差很大，并且有的情况下只能用物化方法。

12.7.1 物化

要直观地理解如何计算一个表达式，最容易的方法是看一看以**运算符树**(operator tree)对表达式所做的图形化表示。考虑图 12-11 所示表达式：

$$\Pi_{name}(\sigma_{building = \text{"Watson"}}(department) \bowtie instructor)$$

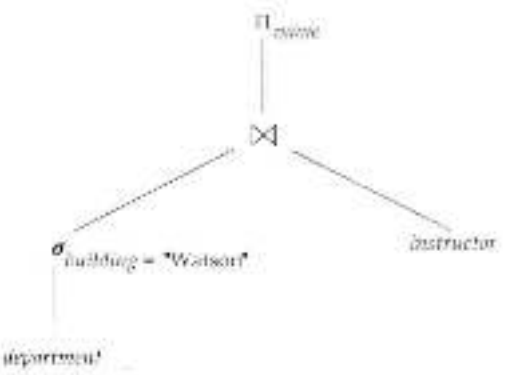


图 12-11 一个表达式的图形化表示

当采用物化方法时，我们从表达式的最底层的运算(在树的底部)开始。在该例子中，只有一个底层运算：*department* 上的选择运算。底层运算的输入是数据库中的关系。我们用前面研究过的算法执行这些运算，并将结果存储在临时关系中。在树的高一层中，使用这些临时关系来进行计算；这时的输入要么是临时关系，要么是来自数据库的关系。在该例子中，连接运算的输入是 *instructor* 关系及在 *department* 关系上进行选择所建立的临时关系。接着可以进行连接运算，建立起另一个临时关系。

通过重复这一过程，最终可以计算位于树的根结点的运算，从而得到表达式的最终结果。在该例

子中，通过执行根结点的投影运算，以连接运算产生的临时关系作为输入执行投影，就可以得到最终的结果。

上述计算方法称为**物化计算**(materialized evaluation)，因为运算的每个中间结果被创建(物化)，然后用于下一层的运算。

物化计算的代价不仅仅是那些所涉及的运算代价的总和。当估计算法的代价时，我们忽略了将运算结果写到磁盘上的代价。为了计算按如上所述执行一个表达式的代价，我们必须把所有运算的代价相加，还要加上把中间结果写回磁盘的代价。我们假设结果记录在缓冲区中积累，当缓冲区写满时，把它们写到磁盘上。写出块数 b_r 的代价可按 n_r/f_r 来估算，其中 n_r 是结果关系 r 中元组数的估计，f_r 是结果关系的块因子，即每块可容纳的关系 r 中的记录数。除了磁盘块传输时间，还需要加上磁盘搜索的时间。因为磁盘头在连续的写回操作之间可能会移动到别处。磁盘搜索的次数可以估算为 $\lceil b_r/b_b \rceil$，其中 b_b 是输出缓冲区的大小(以块数计算)。

双缓冲(double buffering)(即使用两个缓冲区，其中一个用于连续执行算法，另一个用于写出结果)允许 CPU 活动与 I/O 活动并行，从而提高算法执行速度。通过为输出缓冲区分配额外的磁盘块以及每次同时写出多个块，可以减少磁盘搜索次数。

12.7.2 流水线

通过减少查询执行中产生的临时文件数，可以提高查询执行的效率。减少临时文件数是通过将多个关系操作组合成一个操作的流水线来实现的，其中一个操作的结果将传送到下一个操作。这样的计算叫做**流水线计算**(pipelined evaluation)。 567 | 568

例如，考虑表达式 $\Pi_{a1,a2}(r \times s)$。如果采用物化方法，在执行中将创建存放连接结果的临时关系，然后在执行投影时又从连接结果中读入。这些操作可按如下方式组合：当连接操作产生一个结果元组时，该元组马上传送给投影操作去处理。通过将连接操作与投影操作组合起来，我们可以避免中间结果的创建，从而直接产生最终结果。

创建一个操作的流水线可以带来两个好处：

1. 它能消除读和写临时关系的代价，减少查询计算代价。

2. 如果一个查询计算计划的根操作符及其输入合并到流水线中，那么可以迅速开始产生查询结果。一旦它们生成，如果把结果显示给用户，这可能是非常有用的。因为不然的话在用户看到任何查询结果之前可能有一个长时间延迟。

12.7.2.1 流水线的实现

通过将所有操作组合起来构成流水线，构造一个单独的复合操作，我们可以实现流水线。尽管对于各种频繁发生的情况，这个方法是可行的；但一般而言，希望在构造一个流水线时能重用各个操作的代码。

在图 12-11 的例子中，三个操作均可放入一条流水线中：其中，选择操作的结果产生后传送给连接操作。接着，连接操作的结果产生后又传送给投影操作。由于一个操作的结果不必长时间保存，因此对内存的要求不高。然而，由于采用流水线，各个操作并非总是能立即获得输入来进行处理。

流水线可按以下两种方式之一来执行：

1. 在一条**需求驱动的流水线**(demand-driven pipeline)中，系统不停地向位于流水线顶端的操作发出需要元组的请求。每当一个操作收到需要元组的请求，它就计算下一个(若干个)元组并返回。如果该操作的输入不是来自流水线，则返回的下一个(若干个)元组可以由输入关系计算得到。同时系统记载目前为止已返回了哪些元组。如果该操作的某些输入来自流水线，那么该操作也发出请求以获得来自流水线输入的元组。该操作使用来自流水线输入的元组，计算输出元组，然后把它们传给父层。

2. 在一条**生产者驱动的流水线**(producer-driven pipeline)中，各操作并不等待元组请求，而是积极地(eagerly)产生元组。生产者驱动的流水线中的每一个操作作为系统中一个单独的进程或线程建模， 569
以处理流水线输入的元组流，并产生相应的输出元组流。

我们接下来会描述需求驱动的流水线和生产者驱动的流水线如何实现。

需求驱动的流水线中的每个操作可以用**迭代算子**(iterator)来实现。该迭代算子要供以下函数：*open*()、*next*()及*close*()。调用*open*()后，对*next*()的每次调用返回该操作的下一个输出元组。该操作为了获取所需输入元组，在其输入上按次序调用*open*()和*next*()。函数*close*()用于告知迭代算子不再需要元组了。迭代算子维护两次调用之间的执行**状态**(state)，使得下一个*next*()调用请求可以获取下面的结果元组。

例如，对一个采用线性搜索实现选择操作的迭代算子，*open*()操作开始一次文件扫描，迭代算子的状态记录文件中已扫描到的位置。当调用*next*()函数后，文件扫描从前次记录的位置继续执行；如果通过扫描文件找到了满足选择条件的下一个元组，则将所找到元组的位置存储到迭代算子状态中后将元组返回。一个归并-连接迭代算子的*open*()操作将打开其输入，若输入未排序则还将对它们排序。当调用*next*()时，迭代算子将返回下一个匹配的元组对。迭代算子的状态信息包含每次输入曾扫描过的位置。迭代算子的实现细节留作实践练习12.7。

而生产者驱动的流水线以不同的方式实现。对在生产者驱动的流水线中每一对相邻的操作，系统会创建一个缓冲区来保存从一个操作传递到下一个的元组。对应不同操作的进程或线程会并发执行。流水线底部的每个操作会不断产生输出元组，并将它们放入输出缓冲区中，直到缓冲区已满。该当得到来自更底层流水线的输入元组时，流水线的任何其他层操作会产生输出元组，直到输出缓冲区已满。一旦操作使用了来自流水线输入的元组，便会将它从输入缓冲区中移除。不管在哪种情况下，一旦输出缓冲区已满，该操作会等待，直到它的父操作将缓冲区中元组移除，以便缓冲区中有空间放入更多的元组。此时，该操作会产生更多的元组，直到缓冲区再次满了为止。操作重复这样的过程，直到生成所有的输出元组。

只有当一个输出缓冲区已满，或一个输入缓冲区已空，需要多的输入元组用于产生输出元组时，系统才需要在各操作之间切换。在一个并行处理系统中，流水线中的操作可在不同的处理器上并发执行(见第18章)。

使用生产者驱动的流水线方法可以被看作将数据从一棵操作树的底层推(push)上去的过程；而使用需求驱动的流水线方法可被看成是从树顶将数据拉(pull)上来的过程。在生产者驱动的流水线中，元组的产生是积极的，而在需求驱动的流水线中，元组消极地(lazily)按需求产生。需求驱动的流水线
570 比生产者驱动的流水线应用得更为普遍，因为它更容易实现。但是，生产者驱动的流水线在并行处理系统中非常有用。

12.7.2.2 流水线的执行算法

有些操作例如排序，其本质上是**阻塞操作**(blocking operation)，也就是说，它们可能不能输出任何结果，直到所有的输入元组被吸收为止[6]。

其他操作(例如连接)本身并不阻塞，但其具体的计算算法可能会阻塞。例如，散列连接算法是一个阻塞操作，因为在输出任何元组之前，它要求两个输入都被完全取回并被划分。而索引嵌套循环连接算法随着得到外部关系的元组就可以输出结果元组。因此，它称为在外部(左侧)关系上的**流水线化**(pipelined)，尽管因为在索引嵌套循环连接算法执行之前，必须充分构建索引所以导致在其索引(右边)输入上阻塞。

混合散列连接可以被看作是在探查关系上部分流水线的，因为当从探查关系中接收到元组时，它可以输出来自第一个划分中的元组。然而，不位于第一个划分中的元组只有在接收整个流水线输入关系后才能输出。如果构造用输入可以全部存放在内存中，则混合散列连接可以在探查用输入上提供流水线计算，或者如果构造用输入可以大部分存放在内存中，则混合散列连接可以在探查用输入上提供近似流水线计算。

如果两个输入均在连接属性上有序，并且连接条件是等值连接，则可以使用归并连接，并且在两个输入都可流水线化。

---

(6) 如果得知输入满足一些特殊的性质，例如已经排序或者部分排序，那么例如排序那样的阻塞操作就可能能够较早地输出元组。然而，在默认这种信息的情况下，阻塞操作是不能较早地输出元组。

然而，在更常见的情况下，我们希望进行流水线化的连接的两个输入并没有排序。另一种方法是**双流水线连接**(double-pipelined join)技术，如图12-12所示。该算法假定输入关系 r 和 s 的元组都是流水线化的。两个关系的元组放入单个队列中等待处理。特别的队列项，End_r 和 End_s，作为文件结束标记。在 r 和 s 的所有元组(分别地)都产生之后插入到队列中。为了有效地计算，应该在关系 r 和 s 上建立适当的索引。当元组添加到 r 和 s 中时，相关的索引也必须保持更新。当在 r 和 s 上使用散列索引时，由此产生的算法称为**双流水线散列连接**(double-pipeline hash-join)技术。

```
done_r := false;
done_s := false;
r := φ;
s := φ;
result := φ;
while not done_r or not done_s do
  begin
    if 队列空 then 等待，直到队列非空;
    t := 队列中的元一项;
    if t = End_r then done_r := true
    else if t = End_s then done_s := true
    else if t 属于输入关系 r then
      begin
        r := r ∪ {t};
        result := result ∪ (t ⋈ s);
      end
    else /* t 是属于输入关系 s 的元组 */
      begin
        s := s ∪ {t};
        result := result ∪ (r ⋈ t);
      end
  end
```

图12-12 流水线连接算法

图12-12中的双流水线连接算法假设两个输入都可以存放在内存中。如果两个输入比内存大，它仍然和平常情况一样可以使用双流水线连接技术，直到可用内存全部满了为止。当可用内存满了，假定当前的 r 和 s 的元组可以各自当被处于划分 r_0 和 s_0 中。随后到来的 r 和 s 的元组被各自分派给划分 r_1 和 s_1，并直接写入磁盘，而不会添加到内存索引中。但是，当它们写入到磁盘之前，被分配给 r_1 和 s_1 的元组会分别用来探查 r_0 和 s_0。因此 r_1 和 s_0 的连接以及 s_1 和 r_0 的连接同样以流水线方式执行。当 r 和 s 全都处理以后，必须执行 r_1 元组和 s_1 元组的连接，以完成连接操作。我们先前看到的任何连接技术都可以用来连接 r_1 和 s_1。

12.8 总结

- 对于一个查询，系统首先要做的事就是将之翻译成系统内部表示形式。对于关系数据库系统而言，内部形式通常是基于关系代数的。在产生查询的内部形式过程中，语法分析器检查用户查询语句的语法，验证出现在查询语句中的关系名是数据库中的关系名等。如果查询语句是用视图表达的，语法分析器把所有对视图名的引用转换成计算该视图的关系代数表达式。
- 给定一个查询，通常有许多计算它的方法。将用户输入的查询语句转换成等价的、执行效率更高的查询语句，这是优化器的责任。第13章描述查询优化。 571 | 572
- 对于包含简单选择的查询语句，可以通过线性扫描或利用索引来处理。通过计算简单选择结果的并和交，可以处理复杂选择操作。
- 可以用外部归并排序算法对大于内存的关系进行排序。
- 涉及自然连接的查询语句可以有多种处理方法，如何处理取决于是否有索引可用以及关系的物理存储形式。
  - 若连接的结果大小几乎与两个关系的笛卡儿积相当，则采用块嵌套循环连接策略较好。
  - 若存在索引，则可用索引嵌套循环连接。
  - 若关系已排序，则归并连接比较可取。左连接计算前对关系排序是有利的(为了能使用归并连接算法)。
  - 散列连接算法把关系划分成多个部分，使每个部分都能被内存所容纳。划分过程是通过连接属性上的散列函数来进行的，这样相应的划分对可以独立地进行连接。
- 去除重复、投影、集合操作(并、交、差)、聚集操作都可以用排序和散列实现。
- 外连接操作可以通过对连接算法的简单扩展来实现。
- 散列与排序在某种意义下是对偶的。因为任何能用散列实现的操作(如去除重复、投影、聚集、连接、外连接)也可用排序来实现，反之亦然；即任何能用排序来实现的操作也能用散列实现。
- 可以采用物化方法进行表达式的计算。系统计算每个子表达式的结果并将其存到磁盘上，然后用它

进行父表达式的计算。

- 流水线方法在子表达式产生输出的同时就在父表达式的计算中使用其输出结果，有助我们避免了将许多子查询的结果写到磁盘的操作。

术语回顾

- 查询处理
- 计算原语
- 查询执行计划
- 查询计算计划
- 查询执行引擎
- 查询代价度量
- 顺序 I/O
- 随机 I/O
- 文件扫描
- 线性搜索
- 使用索引的选择
- 存取路径
- 索引扫描
- 合取选择
- 析取选择
- 复合索引
- 标识符的交
- 外排序
- 外部排序归并
- 归并段
- N 路归并
- 等值连接
- 嵌套循环连接
- 块嵌套循环连接
- 索引嵌套循环连接
- 归并连接
- 排序－归并连接
- 混合归并连接
- 散列连接
  - 构造
  - 探查
  - 构造用输入
  - 探查用输入
  - 递归划分
  - 散列表溢出
  - 倾斜
  - 递归因子
  - 递归分解
  - 递归避免
- 混合散列连接
- 运算符树
- 物化计算
- 双缓冲
- 流水线计算
  - 需求驱动的流水线（消极、拉的方式）
  - 生产者驱动的流水线（积极、推的方式）
  - 迭代算子
- 双流水线连接

实践习题

12.1 （本题为了简单起见）假设一块中只有一个元组，内存最多容纳 3 块。当使用排序归并算法时，给出对下列元组按第一个属性排序各趟所产生的归并段：(kangaroo, 17)、(wallaby, 21)、(emu, 1)、(wombat, 13)、(platypus, 3)、(lion, 8)、(warthog, 4)、(zebra, 11)、(meerkat, 6)、(hyena, 9)、(hornbill, 2)、(baboon, 12)。

12.2 考虑图 12-13 中银行数据库的例子，其中主码以下划线标出。考虑下面的 SQL 语句：

573
↓
574

```
select T.branch_name
from branch T, branch S
where T.assets > S.assets and S.branch_city = "Brooklyn"
```

写一个与此等价的、高效的关系代数表达式，并论证你的选择。

12.3 设关系 $r_1(A, B, C)$，$r_2(C, D, E)$有如下特性：r_1 有 20 000 个元组，r_2 有 45 000 个元组，一块中可容纳 25 个 r_1 元组或 30 个 r_2 元组。估计使用以下连接策略计算 $r_1 \bowtie r_2$ 需要几次块传输和磁盘搜索：

a. 块嵌套循环连接

b. 块嵌套循环连接

c. 归并连接

d. 散列连接

12.4 如果索引是辅助索引并且在连接属性上多个元组有相同的值，则 12.5.3 节讲述的索引嵌套循环连接算法效率不高。为什么？找出一种利用排序减少内层关系元组检索代价的方法。在什么条件下该算法比混合连接算法更有效？

12.5 令 r、s 是没有索引的两个关系，并且假设两个关系也没有排序。假设内存无限大，计算 $r \bowtie s$ 代价最小（就 I/O 操作而言）的方法是什么？该算法需多大内存？

12.6 考虑图 12-13 的银行数据库，其中主码用下划线标出。假设关系 *branch* 在 *branch_city* 属性上有 B+ 树索引，此外到无其他索引。列出处理下列包含取反运算的选择操作的不同的方法？

a. $\sigma_{\neg(branch\_city < "Brooklyn")}(branch)$

b. $\sigma_{branch\_city = \text{"Brooklyn"}}(branch)$

c. $\sigma_{branch\_city < \text{"Brooklyn"} \vee assets < 5000}(branch)$

12.7 写出实现索引嵌套循环连接的迭代算子的伪码，其中外层关系是流水线的。要求伪码定义标准迭代算子的函数 open()、next()和 close()。显示在不同的调用之间迭代算子需要维护的状态信息。

branch(*branch_name*, *branch_city*, *assets*)
customer(*customer_name*, *customer_street*, *customer_city*)
loan(*loan_number*, *branch_name*, *amount*)
borrower(*customer_name*, *loan_number*)
account(*account_number*, *branch_name*, *balance*)
depositor(*customer_name*, *account_number*)

图 12-13 银行数据库

575

12.8 设计基于排序的和基于散列的算法，用于计算关系的除运算（有关除运算定义请参照第 6 章的实践练习）。

12.9 为每个归并段增加缓冲块，而用于缓冲归并段的总可用内存保持不变。这对于归并这些段的代价有什么影响？

习题

12.10 假设需要排序 40GB 的关系，使用 4KB 大小的块以及 40GB 大小的内存。假设一次搜索代价是 5 毫秒，磁盘传输速率是每秒 40MB。

a. 对于 $b_b = 1$ 和 $b_b = 100$ 的情况，分别计算排序该关系的代价（以秒为单位）。

b. 在上述每种情况下，各需要多少遍归并？

c. 假设用一个闪存设备来替代磁盘，它的搜索时间是 1 微秒，传输速率是每秒 40MB。在这样的设置下，对于 $b_b = 1$ 和 $b_b = 100$ 的情况，分别重新计算关系排序的代价（以秒为单位）。

12.11 考虑如下关系代数运算的扩展。描述如何使用排序和散列实现每个操作。

a. **半连接**（semijoin）（$\ltimes_\theta$）：$r \ltimes_\theta s$ 定义为 $\prod_R(r \bowtie_\theta s)$，其中 R 是模式 r 的属性集合。即在 r 中选择如下的元组 r_i：在 s 中存在元组 s_j，使得 r_i 与 s_j 能满足谓词 θ。

b. **反半连接**（anti-semijoin）（$\bar{\ltimes}_\theta$）：$r \bar{\ltimes}_\theta s$ 定义为 $r - \prod_R(r \bowtie_\theta S)$，其中 R 是模式 r 的属性集合。即在 r 中选择如下的元组 r_i：在 s 中不存在元组 s_j，使得 r_i 与 s_j 能满足谓词 θ。

12.12 为什么不应当要求用户显式地选择查询处理计划？是否存在希望用户知道两个候选的查询处理计划代价的情形？请做出解释。

12.13 两个关系都在物理上没排序，但各自都有在连接属性上排序的辅助索引。为混合归并连接算法设计一个变体以适应这种情况。

12.14 估算习题 12.13 中你给出的计算 $r_1 \bowtie r_2$ 的解决方案所需磁盘块传输和磁盘搜索次数，其中 r_1、r_2 按照实践习题 12.3 的定义。

576

12.15 12.5.5 节讲述的散列连接算法用于计算两个关系的自然连接。说明如何扩展散列连接算法，以计算自然左外连接、自然右外连接和自然全外连接。（提示：在散列索引中保存每个元组的额外信息，判断在探查用关系中是否有与散列索引中元组相匹配的元组。）将你的算法用到关系 *takes* 与 *student* 上。

12.16 流水线用于避免把中间结果写回磁盘。假设你需要用排序归并法对一个关系 r 排序，并且用归并连接法把结果与已排序的关系 s 做连接操作。

a. 描述关系 r 排序的输出如何组成流水线以避免写回磁盘？

b. 即使归并连接的两个输入都来自排序归并操作的输出，同样的思想还是适用的。然而可用的内存必须在两个归并操作之间共享（归并连接算法本身只需要很少的内存）。这样，必须共享内存对每个排序归并操作的代价有什么影响？

12.17 用伪码书写实现排序归并算法的一个版本的迭代算子，其中最后一次归并的结果使用流水线传

给下一个操作。要求你写定义标准迭代算子的函数 open( )、next( ) 和 close( )。显示在不同的调用之间迭代算子需要维护的状态信息。

12.18 假设你要计算 $G_{sum(C)}(r)$ 和 $_{A,B}\mathcal{G}_{sum(C)}(r)$。描述如何对 r 进行一次排序实现对这两个表达式的同时计算。

文献注解

查询处理器必须对查询语言中的语句做语法分析，并且必须将它们转换成某种内部表示。查询语言的语法分析与传统编程语言的语法分析几乎没有区别。大多数有关编译器的书都包含主要的语法分析技术，并从编程语言的角度描述了优化技术。

Graefe 和 McKenna[1993b]给出了一份关于查询执行技术的非常优秀的调研。

Knuth[1973]给出了关于外排序算法（包括其优化算法）的极好描述，叫做替代选择。这个算法能创建大小（平均来说）为内存大小两倍的初始归并段。Nyberg 等[1995]的研究显示，由于低效的处理器缓存行为，对产生归并段来说替代选择方法性能比内存中的快速排序性能要差，抵消了产生更长的归并段带来的好处。Nyberg 等[1995]提出一种高效的外排序算法，这种算法把处理器缓存的效应考虑在内。处理器缓存的效应考虑在内的查询执行算法已得到广泛的研究，作为例子可以参考 Harizopoulos 和 Ailamaki[2004]。

根据 20 世纪 70 年代中期所进行的性能研究，当时的数据库系统只使用嵌套循环连接和归并连接，包括 Blasgen 和 Eswaran[1976]，这些同 System R 的开发相联系的研究表明，嵌套循环连接或归并连接几乎总能提供最佳连接方式，因此，System R 中实现的连接算法只有这两种。然而，Blasgen 和 Eswaran[1976]没有包括对散列连接算法的分析。现在，散列连接被认为是效率很高的并得到广泛的使用。

散列连接算法最初为并行数据库系统而设计。散列连接技术在 Shapiro[1986]中描述。Zeller 和 Gray[1990]以及 Davison 和 Graefe[1994]所描述的散列连接技术，可以根据可用内存作调整，而这一点在多个查询可同时运行的系统上非常重要。Graefe 等[1998]描述了散列连接和散列组的使用，他们采用一种在一个流水线序列中对所有的散列连接进行相同划分的方法，在 Microsoft SQL Server 中对散列连接进行流水线化。

577
578

查询优化

对一个给定的查询，尤其是复杂查询，通常会有许多种可能的策略，**查询优化**(query optimization)就是从这许多策略中找出最有效的查询执行计划的一种处理过程。我们不期望用户能够写出一个能高效处理的查询，而是期望系统能够构造一个能让查询执行代价最小化的查询执行计划。这正是查询优化起作用的地方。

优化一方面在关系代数级别发生，即系统尝试找出一个与给出的表达式等价但执行起来更为高效的表达式。另一方面是为处理查询选择一个详细的策略，比如对一个操作的执行选择所用的算法，选择使用的特定索引等。

一个好的查询策略和一个差的查询策略在代价(执行时间)上通常会有相当大的区别，可能会相差好几个数量级。因此，即使查询只执行一次，系统为处理查询选择一个好的策略花费一定量的时间是完全值得的。

13.1 概述

考虑下面查询"找出 Music 系的所有教师名字以及每个教师所教授课程名称"的关系表达式：

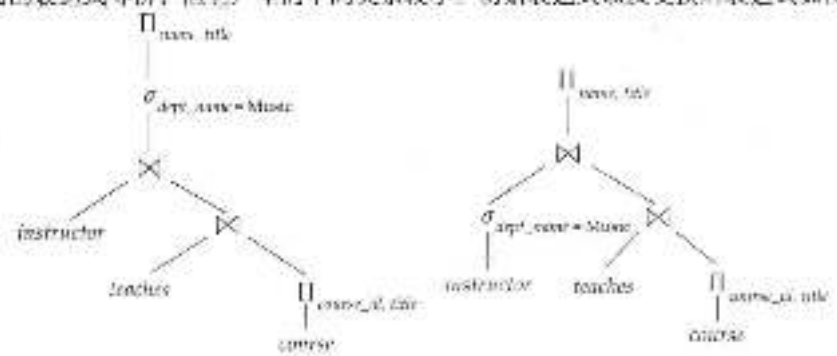
$$\Pi_{name,\ title}(\sigma_{dept\_name = 'Music'}(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course))))$$

注意，*course* 上(*course_id*, *title*)的投影是必需的，因为 *course* 和 *instructor* 有一个公共属性 *dept_name*，如果我们不通过投影来剔掉这个属性，则上面自然连接的表达式会仅返回 Music 系的课程，尽管 Music 系的一些教师可能教授其他系的课程。

以下表达式需产生一个很大的中间关系，*instructor* ⋈ *teaches* ⋈ Π*course_id*, *title*(*course*)。然而，我们只对这个关系的少数元组感兴趣(那些与音乐系的教师有关的)，并且只对这个关系的 10 个属性中的两个感兴趣。由于我们只关心 *instructor* 关系中与音乐系有关的元组，我们没有必要考虑不满足 *dept_name* = "Music"条件的元组。通过减少要访问的 *instructor* 关系的元组数，也就减少了中间结果的大小。现在将查询表示为如下的关系代数表达式：

$$\Pi_{name,\ title}(\sigma_{dept\_name = 'Music'}(instructor) \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course)))$$

它与我们前面的表达式等价，但它产生的中间关系较小。初始表达式以及变换后表达式如图 13-1 所示。



a) 初始表达式树　　b) 转化后的表达式树

图 13-1 等价表达式

一个执行计划确切地定义了每个运算应使用的算法，以及运算之间的执行应该如何协调。图 13-2 表明了图 13-1b 的表达式的一个可行的执行计划。正如我们看到的，每个关系运算可以有几种不同的算法，从而可产生其他的执行计划。在图 13-2 中，连接运算中的一个选择了散列连接，另一个则在连接的 ID 属性上将关系排序后，选用归并连接。在凡被标记流水线的边上，生产者的输出直接流向消费者，而不会被写入磁盘。

给定一个关系代数表达式，查询优化器的任务是产生一个查询执行计划，该计划能获得与原关系表达式相同的结果，并且得到结果集的执行代价最小（或至少是不比最小执行代价大多少）。

为了找到代价最小的查询执行计划，查询优化器需要产生一些能和给定表达式得到相同结果的候选计划，并选择代价最小的一个。查询执行计划的产生有三步：(1)产生基础上与给定表达式等价的表达式；(2)对所产生的表达式以不同方式作注释，产生不同的查询计划；(3)估计每个执行计划的代价，选择估计代价最小的一个。

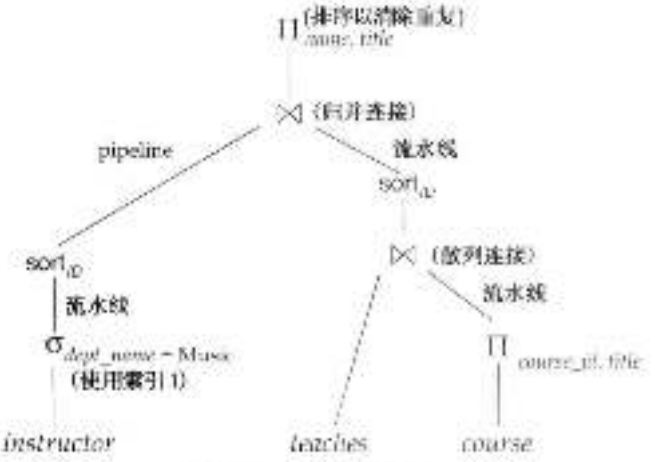


图 13-2 一个执行计划

查询优化器中第(1)～(3)步是交叉的：先产生一些表达式并加以注释，注释后产生执行计划，然后再进一步产生一些表达式并加以注释，依此类推。执行计划产生后，其代价通过关系的统计信息（如关系的大小和索引的深度）来估计。

要完成第一步，查询优化器需要产生与给定表达式等价的表达式，这是通过等价规则来进行的，等价规则详细说明了如何将一个表达式转换成逻辑上等价的另一个表达式。13.2 节将讲述这些规则。

13.3 节描述了如何估计一个查询计划中每个操作的结果的统计大小。将这些统计结果应用到第 12 章中的代价公式就可以估计出单个操作的代价。把这些单个操作的代价合起来就能够确定出执行给定关系代数表达式的代价估计，就像此前 12.7 节描述的那样。

13.4 节讲述如何选择一个查询执行计划。我们可以基于执行计划的代价估计来进行选择。由于代价是估计值，因此所选计划未必是代价最小的计划；但是，只要估计得好，该计划得可能是代价最小的计划，或其代价不比最小值大多少。

580
581

最后，物化视图能够帮助加速某些查询的处理。13.5 节将研究如何"维护"物化视图（即，使它们保持最新）以及如何利用物化视图执行查询优化。

13.2 关系表达式的转换

一个查询可以表示成多种不同的形式，每种形式具有不同的执行代价。在这一节里，我们不是仅仅按照给定的关系表达式那样去做，而是考虑其他可选的等价表达式。

如果两个关系表达式在每一个有效数据库实例中都会产生相同的元组集，则我们称它们是等价的（equivalent）。（回忆一下，一个有效的数据库实例是指满足在数据库模式中指定的所有完整性约束的数据库实例。）注意，元组的顺序是无关紧要的；两个表达式可能以不同的顺序产生元组，但只要元组集是一致的，就认为它们是等价的。

在 SQL 语言中，输入和输出都是元组的多重集合。关系代数的多重集合版本（在本节的示例栏中描述）用于评估 SQL 查询。若对于任意有效的数据库，两个表达式产生相同的元组多重集合，则称多重集合版本的这两个关系代数表达式是等价的。这一章的讨论是基于关系代数的。我们将关系代数的多重集合版本的扩充留给读者作为练习。

查看查询执行计划

许多数据库系统提供一种方式来查看执行给定查询所选择的查询执行计划。通常最好的方式是用数据库系统提供的图形用户界面来查看执行计划。然而，如果你采用命令行界面，许多数据库支持一条命令"**explain** <query>"的变种，来显示对于给定查询 <query> 所选择的执行计划。不同的数据库采用不同的严格语法：

- PostgreSQL 采用上述语法。
- Oracle 采用语法 **explain plan for**。但是，此命令将结果计划存储在一个叫 *plan_table* 的表中，而不是直接显示它。查询"**select * from table**(*dbms_xplan.display*);"显示存储的计划。
- DB2 采用和 Oracle 类似的方法，但是要求执行 db2exfmt 程序来显示存储的计划。
- SQL Server 要求提交查询执行命令 **set showplan_text on**，然后当提交查询后，显示查询计划，而不显示执行查询。

为计划估计的代价也随着计划显示。值得注意的是，代价通常没有以外在意义的（例如秒或 I/O 操作）单位表示，而以优化器采用的代价模型的单位表示。有些优化器显示两个估计代价数字，例如 PostgreSQL。第一个表示对第一个输出结果估计的代价，第二个表示对所有输出结果估计的代价。

13.2.1 等价规则

等价规则（equivalence rule）指出两种不同形式的表达式是等价的。我们可以用第二种形式的表达式代替第一种，或者反之——可以用第一种形式的表达式代替第二种，这是因为这两种表达式在任何有效的数据库中将产生相同的结果集。优化器利用等价规则将表达式转换成逻辑上等价的其他表达式。

下面列出关系代数表达式的一些通用等价规则。所列出的等价式中的一些在图 13-3 做了说明。我们用 θ、θ_1、θ_2 等表示谓词，L_1、L_2、L_3 等表示属性列表，而 E、E_1、E_2 等表示关系代数表达式。关系名 r 是关系代数表达式的特例，在 E 出现的任何地方它都可以出现。

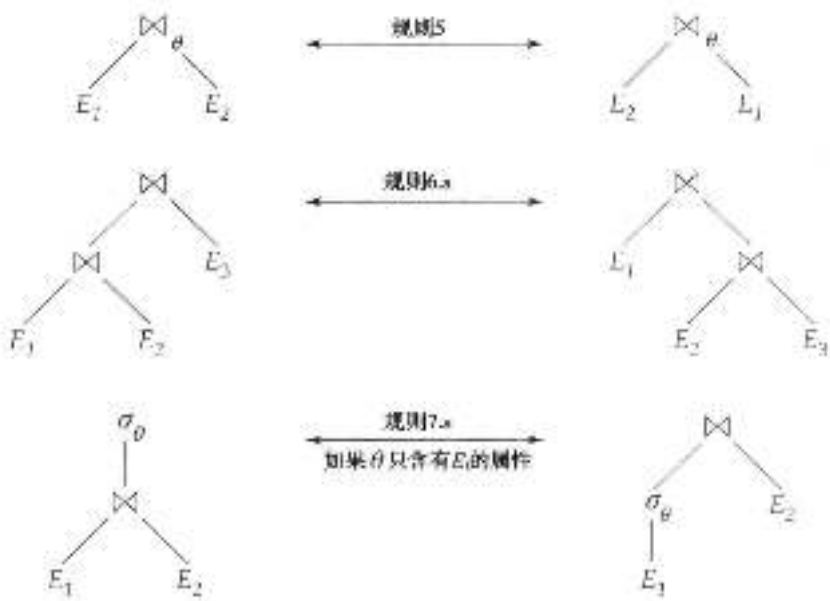


图 13-3 等价表达式的图形化表示

1. 合取选择运算可分解为单个选择运算的序列。该变换称为 σ 的级联：

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

582
583

2. 选择运算满足**交换律**（commutative）：

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. 一系列投影运算中只有最后一个运算是必需的，其余的可省略。该特换也可称为Π的级联：

$$\Pi_{L_1}(\Pi_{L_2}(\cdots(\Pi_{L_n}(E))\cdots)) = \Pi_{L_1}(E)$$

4. 选择操作可与笛卡儿积以及 θ 连接相结合：

a. $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$。

该表达式就是 θ 连接的定义。

b. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$。

5. θ 连接运算满足交换律：

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

事实上，左端和右端的属性顺序不同，所以如果考虑属性顺序的话，等式不成立。可以对等价规则的其中一端加入一个投影操作以适当重排属性，但为了简化，我们省略了该投影并且在大部分例子中忽略属性顺序。

回忆一下，自然连接是 θ 连接的特例，因此自然连接也满足交换律。

6. a. 自然连接运算满足**结合律**(associative)：

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

b. θ 连接具有以下方式的结合律：

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

其中 θ_2 只涉及 E_2 与 E_3 的属性。由于其中的任意一个条件都可为空，因此笛卡儿积（×）运算也满足结合律。连接运算满足结合律，交换律在查询优化中对于重排连接顺序是很重要的。

7. 选择运算在下面两个条件下对 θ 连接运算具有分配律：

584

a. 当选择条件 θ_0 中的所有属性只涉及参与连接运算的表达式之一（比如 E_1）时，满足分配律：

$$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_\theta E_2$$

b. 当选择条件 θ_1 只涉及 E_1 的属性，选择条件 θ_2 只涉及 E_2 的属性时，满足分配律：

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_\theta (\sigma_{\theta_2}(E_2))$$

8. 投影运算在下面条件下对 θ 连接运算具有分配律：

a. 令 L_1、L_2 分别代表 E_1、E_2 的属性。假设连接条件 θ 只涉及 $L_1 \cup L_2$ 中的属性，则：

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

b. 考虑连接 $E_1 \bowtie_\theta E_2$。令 L_1、L_2 分别代表 E_1、E_2 的属性集；令 L_3 是 E_1 中出现在连接条件 θ 中但不在 $L_1 \cup L_2$ 中的属性；令 L_4 是 E_2 中出现在连接条件 θ 中但不在 $L_1 \cup L_2$ 中的属性。那么：

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$$

9. 集合的并与交满足交换律。

$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$

集合的差运算不满足交换律。

10. 集合的并与交满足结合律。

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. 选择运算对并，交，差运算具有分配律：

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - \sigma_P(E_2)$$

类似地，上述等价规则将"－"替换成∪或∩时也成立。进一步有：

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - E_2$$

上述等价规则将"－"替换成∩时也成立，但将"－"替换成∪时不成立。

12. 投影运算对并运算具有分配律：

585

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

以上列出的只是等价规则的一部分。涉及扩展关系代数运算符（如外连接和聚集）的更多等价规则将在习题中讨论。

13.2.2 转换的例子

下面举例说明等价规则的用法。我们用大学这个例子，其关系模式如下：

Instructor(*ID*, *name*, *dept_name*, *salary*)
teaches(*ID*, *course_id*, *sec_id*, *semester*, *year*)
course(*course_id*, *title*, *dept_name*, *credits*)

在13.1节的例子中，表达式：

$$\Pi_{name,title}(\sigma_{dept\_name="Music" \wedge year=2009}(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course))))$$

转换成以下表达式：

$$\Pi_{name,title}((\sigma_{dept\_name="Music"}(instructor)) \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course)))$$

转换后表达式等价于原始关系代数表达式，但产生较小的中间关系。这一转换可用规则7.a来实现。记住，规则只是说两个表达式等价，而未表明孰优孰劣。

针对一个查询或查询的一部分，可以连续使用多条等价规则。作为一个例子，假设我们对原先的查询加以修改，将注意力限制在2009年教授一门课程的那些教师。新的关系代数查询可写为：

$$\Pi_{name,title}(\sigma_{dept\_name="Music" \wedge year=2009}(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course))))$$

我们不能把选择谓词直接应用到*instructor*关系上，因为该谓词同时牵涉*instructor*和*teaches*两个关系的属性。然而，我们可以先用规则6.a（自然连接的结合律）把连接*instructor* ⋈ (*teaches* ⋈ $\Pi_{course\_id,\ title}$(*course*))转换成(*instructor* ⋈ *teaches*) ⋈ $\Pi_{course\_id,\ title}$(*course*)：

$$\Pi_{name,title}(\sigma_{dept\_name="Music" \wedge year=2009}((instructor \bowtie teaches) \bowtie \Pi_{course\_id,\ title}(course))) \quad [586]$$

然后使用规则7.a，可将查询改写为：

$$\Pi_{name,title}((\sigma_{dept\_name="Music" \wedge year=2009}(instructor \bowtie teaches)) \bowtie \Pi_{course\_id,\ title}(course))$$

我们再看看该表达式的选择子表达式。利用规则1，可以把选择条件分解为两个选择条件，得到以下子表达式：

$$\sigma_{dept\_name="Music" \wedge year=2009}(instructor \bowtie teaches)$$

上述两个表达式均选择出同时满足*dept_name* = "Music"及*year* = 2009条件的元组。但后一表达式形式可以让我们应用7.a("尽早执行选择")这条规则，得到如下子表达式：

$$\sigma_{dept\_name="Music"}(instructor) \bowtie \sigma_{year=2009}(teaches)$$

初始表达式及经过上述转换后的最终表达式的图形化表示如图13-4所示。实际上我们也可以应用规则7.b直接得到最后的表达式，而不必用规则1将选择分解为两个选择。事实上，规则7.b本身可由规则1及7.a导出。

若一组等价规则中任意一条规则都不能由其他规则联合起来导出，称这组等价规则集为最小的(minimal)等价规则集。上例表明13.2.1节中的等价规则集不是最小的。一个与初始表达式等价的表达式可以用不同的方式产生。若不使用等价规则的最小集，则产生表达式的不同方法的数量也会增加。因此，查询优化器使用等价规则的最小集。

对上述例子的查询，现在考虑如下的形式：

$$\Pi_{name,title}((\sigma_{dept\_name="Music"}(instructor) \bowtie teaches) \bowtie \Pi_{course\_id,\ title}(course))$$

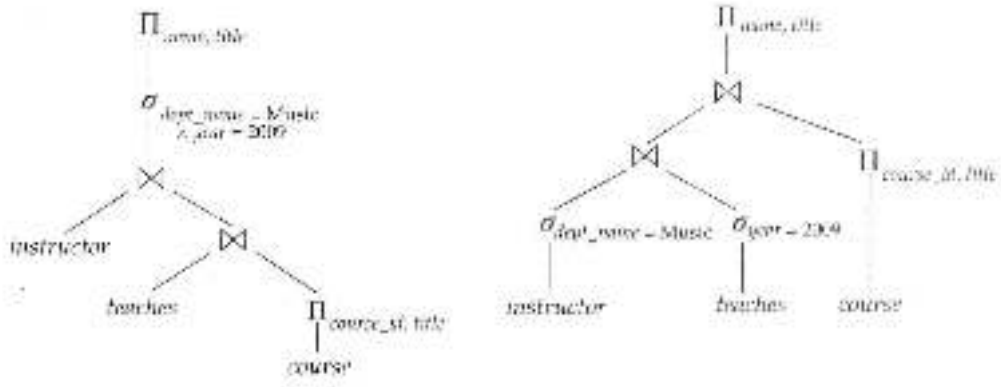
当计算子表达式时：

$$(\sigma_{dept\_name="Music"}(instructor) \bowtie teaches)$$

得到具有如下模式的关系：

(*ID*, *name*, *dept_name*, *salary*, *course_id*, *sec_id*, *semester*, *year*)

通过使用等价规则8.a及8.b先做投影运算，可以从模式中去除几个属性。要保留的属性是那些



a) 初始表达式树　　b) 多次转换后的表达式树

图 13-4　多次转换

出现在查询结果中或在后续运算中要处理的属性。通过去除不必要的属性，可以减少中间结果的列数，从而减小中间结果的大小。在该例子中，从 *instructor* 与 *teaches* 的连接中我们只需属性 *name* 和 *course_id*。因此可将表达式修改为：

$$\Pi_{name,title}((\Pi_{name,course\_id}(\sigma_{dept\_name='Music'}(instructor)) \bowtie teaches) \bowtie \Pi_{course\_id,title}(course))$$

投影 $\Pi_{name,course\_id}$ 减小了中间连接结果的大小。

13.2.3　连接的次序

一个好的连接运算次序对于减少临时结果的大小是很重要的，因此大部分查询优化器在连接次序上花了很多工夫。正如第 6 章及在等价规则 6.a 提到的，自然连接运算满足结合律。所以，对任给关系 r_1、r_2 和 r_3，有

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

虽然这两个表达式等价，但计算它们的代价可能不同。再次考虑表达式

$$\Pi_{name,title}((\sigma_{dept\_name='Music'}(instructor)) \bowtie teaches \bowtie \Pi_{course\_id,title}(course))$$

我们可以选择先计算 $teaches \bowtie \Pi_{course\_id,title}(course)$，然后将结果与

$$\sigma_{dept\_name='Music'}(instructor)$$

相连接。

不过，$teaches \bowtie \Pi_{course\_id,title}(course)$ 可能是一个很大的关系，因为每门课均对应一个元组。与此相反，

587
588

$$\sigma_{dept\_name='Music'}(instructor) \bowtie teaches$$

很可能是一个小关系。为说明这一点，我们注意到因为一所大学拥有很多系，很可能只有一小部分大学教师和音乐系相关。因此，上述表达式得到的结果对于音乐系的每个教师所教授的每门课有一个元组。这样，我们需要保存的临时关系比先计算 $teaches \bowtie \Pi_{course\_id,title}(course)$ 连接所需保存的关系要小。

执行查询还需要考虑其他因素。我们不关心属性在连接中出现的次序，因为在显示结果前改变属性的顺序是很容易的。因此，对于任何关系 r_1、r_2：

$$r_1 \bowtie r_2 = r_2 \bowtie r_1$$

即自然连接满足交换律（等价规则 5）。

利用自然连接满足结合律与交换律（规则 5 与 6）的性质，考虑如下关系代数表达式：

$$(instructor \bowtie \Pi_{name,course\_id}(course)) \bowtie teaches$$

请注意，因为 $\Pi_{name,course\_id}(course)$ 与 *instructor* 之间没有公共属性，所以以上连接为笛卡儿积。若 *instructor* 中有 a 个元组，$\Pi_{name,course\_id}(course)$ 中有 b 个元组，该笛卡儿积将产生 $a \times b$ 个元组，结果集中的

每个元组均为 *instructor* 的一个元组与 *course* 的一个元组所可能形成的元组对(该计算没有考虑教师是否教授课程)。因此，该笛卡儿积将产生较大的临时关系。然而，如果用户输入以上表达式，我们可以用自然连接的结合律与交换律把这个表达式转换成前面我们所用的更有效的表达式：

$$(instructor \bowtie teaches) \bowtie \Pi_{\text{[illegible]}}(course)$$

13.2.4 等价表达式的枚举

查询优化器使用等价规则系统地产生与给定表达式等价的表达式。在概念上，该流程如图 13-5 所描述。处理过程如下：给定一个表达式 E，等价表达式集合 EQ 最初只包含 E。现在，将 EQ 中的每条表达式与每条等价规则匹配。如果表达式 E_i 中有任何子表达式 e_i(作为特例，可能就是 E_i 本身)与等价规则的某一边相匹配，优化器就产生一个新的表达式。其中子表达式 e_i 被替换成与它相匹配的规则的另一边。结果表达式加入到 EQ 中。该处理过程不断进行下去，直到不再有新表达式产生为止。

上述处理方式无论在时间上还是在空间上代价都很大。但是采用两种关键思想，优化器可以极大地减少空间和时间上的开销。

1. 如果在子表达式 e_i 上使用等价规则把表达式 E_i 转换成 E'，则除了 e_i 及其转换，E_i 与 E' 有相同的子表达式。而且 e_i 及其转换通常也有许多相同的子表达式。可以采用一些表达式表示技术，使两个表达式指向共享的子表达式，这样可以明显减少对空间的需求。 589

2. 不必总是用等价规则产生所有可以产生的表达式。正如我们将在 13.4 节中看到的那样，如果考虑估计的执行代价，优化器可以避免检查某些表达式。使用这些技术，可以减少优化所需时间。

我们将会在 13.4.2 节再次讨论这些问题。

```
procedure genAllEquivalent(E)
EQ = {E}
repeat
    将 EQ 中的每条表达式 E_i 与每条等价规则 R_j 进行匹配
    如果 E_i 的某个子表达式与 R_j 的某一边相匹配
        生成一个与 E_i 等价的 E'，其中 e_i 被替换成 R_j 的另一边
        如果 E' 不在 EQ 中，则将其添加到 EQ 中
until 不再有新的表达式可以添加到 EQ 中
```

图 13-5 产生所有等价表达式的过程

13.3 表达式结果集统计大小的估计

一个操作的代价依赖于它的输入的大小和其他统计信息。给定一个表达式，如 $a \bowtie (b \bowtie c)$，估计 a 与 $(b \bowtie c)$ 的连接代价，我们需要有一些统计信息的估计，比如 $(b \bowtie c)$ 的大小。

在这一节中，首先列出一些有关存储在数据库目录中的关于数据库关系的统计信息，然后指出如何使用这些统计信息估计不同关系操作运算结果的统计信息。

在这一节里稍后可以看出，该估计并不十分精确，因为估计是基于一个不严密的假设。因此，某个具有最小执行代价估计的查询执行计划可能事实上并不具有最小的实际执行代价。然而，实践经验告诉我们，即使估计并不精确，具有最小代价估计的计划通常等于或接近于实际最小执行代价。

13.3.1 目录信息

数据库系统目录存储了有关数据库关系的下列统计信息： 590

- n_r，关系 r 的元组数。
- b_r，包含关系 r 中元组的磁盘块数。
- l_r，关系 r 中每个元组的字节数。
- f_r，关系 r 的块因子——一个磁盘块能容纳关系 r 中元组的个数。
- $V(A, r)$，关系 r 中属性 A 中出现的非重复值个数。该值与 $\Pi_A(r)$ 的大小相同。如果 A 是关系 r 的主码，则 $V(A, r)$ 等于 n_r。

需要的话，这最后一个统计值 $V(A, r)$ 也可以是对属性集维护的，而不是仅仅用于单独的属性。因此对于给定的属性集 A，$V(A, r)$ 就是 $\Pi_A(r)$ 的大小。

如果我们假设关系 r 的元组在物理上存储于一个文件中，则下面的等式成立：

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

关于索引的统计信息，比如 B$^+$ 树索引的高度和索引中叶节点的页数，也保存在目录中。

如果我们希望维护准确的统计信息，则每次关系修改时，必须同时更新这些统计信息。这样更新导致了一定量的开销。因此，许多系统并不是每次修改都更新统计信息，而是当系统处于轻负载时进行更新。这样，用于选择查询处理策略的统计信息可能并不完全精确。然而，如果在统计信息更新的间隔里没有发生太多的更新，那么统计信息用以对不同计划的相对代价进行较好的估计时将足够精确。

这里讲到的统计信息是简化过的。现实中的优化器经常维护更深入的统计信息用以提高对执行计划的代价估计的精确度。例如，大多数数据库将每个属性的取值分布为存为一张**直方图**(histogram)；在直方图中，每个属性的取值分成若干个区间，对于每个区间，直方图将属性值落在该区间中的元组个数与该区间相关联。图 13-6 给出了一个范围在 1～25 之间的整数数值型属性的直方图的示例。

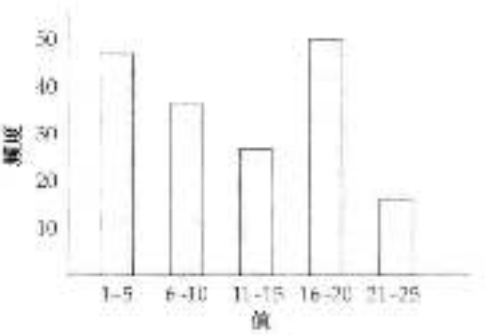


图 13-6 直方图示例

在关系数据库中使用的直方图除了记录属性落入每个区间的元组个数以外，通常也记录每个区间内不同的属性取值个数。

举一个直方图的例子来说，一个关系 *person* 的属性 *age* 的取值范围可分成 0～9，10～19，…，90～99(假设最大年龄值是 99)。对每个区间，我们记录那些 *age* 值落在该区间的 *person* 元组个数，以及落入该区间的不同年龄取值的个数。如果没有这样的直方图信息，优化器将不得不假设属性值的分布是均匀的，即每个区间具有同样的计数值。

一个直方图只占用很少的空间，因此不同属性上的直方图可以存储在系统目录里。数据库系统中使用的直方图有几种。例如，**等宽直方图**(equi_width histogram)把取值范围分成相等大小的区间，而**等深直方图**(equi_depth histogram)则调整区间的分界以便落入每个区间的取值个数相同。

13.3.2 选择运算结果大小的估计

对一个选择运算结果大小的估计依赖于选择谓词。我们首先考虑一个单独的等值谓词，其次考虑一个单独的比较谓词，最后考虑谓词联合。

- $\sigma_{A=a}(r)$：如果我们假设取值是均匀分布的(即，每个值以同样的概率出现)，并假设关系 r 的一些记录的属性 A 中的取值为 a，则可估计选择结果有 $n_r/V(A, r)$ 个元组。这里选择操作中的值 a 在一些记录中出现的假设通常是成立的，而且代价估计总是默认这一假设。然而，假设每个值以同样的概率出现通常是不现实的。关系 *takes* 的属性 *course_id* 就是一个假设不成立的例子。我们有理由预期受欢迎的本科课程比小的专业的研究生课程有更多学生。因此，一些 *course_id* 值出现的可能性要比其他的大。尽管平均分布假设通常不成立，但在许多情况下它是对现实的合理近似，并且能使我们的阐述相对简单。

591
|
592

  如果属性 A 上有一个直方图，我们可以定位值 a 所在的区间，然后修改上述估算式 $n_r/V(A, r)$，用该区间的频度计数代替 n_r，用该区间中出现的不同属性值的个数代替 $V(A, r)$。

- $\sigma_{A\le a}(r)$：考虑形如 $\sigma_{A\le a}(r)$ 的选择操作。如果在进行代价估计时，用于比较操作的值(a)已知，则可做更精确的估计。属性 A 的最小值 $\min(A, r)$ 和最大值 $\max(A, r)$ 可存储到目录中。假设值是平均分布的，我们可以对满足条件 $A \le a$ 的记录数进行下列估计：若 $a < \min(A, r)$，则为 0；若 $a \ge \max(A, r)$，则为 n_r；否则，为：

$$n_r * \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$$

如果属性 A 上存在直方图，我们就可以得到更精确的估计，我们将细节作为练习留给读者来完成。在某些情况下，比如查询是存储过程的一部分，查询优化不无法得到 v 的值。在这样的情况下，我们假设大约有一半的记录将满足比较条件。也就是说，我们假设结果集包含 $n_r/2$ 个元组。这个估计可能非常不精确，但这是在没有任何进一步信息的条件下所能采取的最好的办法了。

计算及维护统计数据

从概念上讲，关系上的统计数据可以理解为物化视图，该统计数据当关系修改时应该自动维护。遗憾的是，对于数据的每一次插入、删除和更新保持最新的统计数据开销是非常大的。另一方面，优化器一般不需要准确的统计数据：百分之几的错误可能导致一个不是最优的计划被选择，但是这个选择的计划可能比最优计划的代价仅高百分之几。因此，近似的统计数据是可以接受的。

利用统计数据可以近似的事实，数据库系统通过以下方式减少生成和维护统计数据的开销。

- 统计数据通常基于基础数据的部分集计算，而不用检查整个数据集合。例如，一个包含准确的直方图可以通过几千个元组计算出来，尽管一个关系包含百万或亿万的记录。然而，使用的样本必须是**随机抽样的样本**(random sample)。一个不是随机抽样的样本可能过度表现关系的一部分，从而误导结果。例如，如果我们采用教师的样本来计算工资的直方图，如果抽样过于表示低工资的教师，则直方图会导致错误的估计。目前数据库系统经常使用随机抽样来生成统计数据。抽样的相关研究请参见文献注解。
- 不是对于每个数据库更新都维护统计数据。事实上，一些数据库系统从不自动更新统计数据。它们依靠数据库管理员定期运行一条命令来更新统计数据。Oracle 和 PostgreSQL 提供一条称为 **analyze** 的数据库命令产生指定关系或者所有关系上的统计数据。IBM DB2 提供一个等价的称为 **runstats** 的命令。更多细节请查看系统手册。你应该意识到，由于不准确的统计数据，优化器有可能选择非常糟糕的计划。许多数据库系统(包括 IBM DB2、Oracle 和 SQL Server)会在某些时间点自动地更新统计数据。例如，系统可以近似跟踪一个关系中有多少元组，并且当元组数量经常变化时重新计算统计数据。另一种方法是比较一个关系扫描估计的基数和执行一个查询时真正的基数。如果它们差异显著，则对该关系启动统计数据的更新。

- 复杂选择：

  合取——合取选择是形式如下的选择操作：

  $$\sigma_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n}(r)$$

  我们按如下方式估计该选择的结果集的大小：对每个 θ_i，我们按照以前描述的那样估计选择 $\sigma_{\theta_i}(r)$ 的大小，记为 s_i。因此，关系中的一个元组满足选择条件 θ_i 的概率为 s_i/n_r。

  上述概率称为选择 $\sigma_{\theta_i}(r)$ 的**中选率**(selectivity)。假设各条件相互独立，则某个元组满足全部条件的概率是全体概率的乘积。因此，我们可以估计满足全部选择条件的元组数量为：

  $$n_r * \frac{s_1 * s_2 * \cdots * s_n}{n_r^n}$$

  析取——析取选择是形式如下的选择操作

  $$\sigma_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n}(r)$$

  所有满足单个条件 θ_i 的记录的并满足析取条件。

  如前所述，s_i/n_r 代表某元组满足条件 θ_i 的概率。元组满足整个析取式的概率为 1 减去元组不满足任何一个条件的概率，即：

  $$1 - \left(1 - \frac{s_1}{n_r}\right) * \left(1 - \frac{s_2}{n_r}\right) * \cdots * \left(1 - \frac{s_n}{n_r}\right)$$

将该值乘以 n_r，即得到满足该选择条件的元组数的估计。

593 / 594

- 取反——在无空值的情况下，选择 $\sigma_{\neg\theta}(r)$ 的结果集就是不在 $\sigma_\theta(r)$ 中的关系 r 的元组集。我们已知如何估计 $\sigma_\theta(r)$ 的元组数，因此 $\sigma_{\neg\theta}(r)$ 的元组数的估计为 n_r 减去 $\sigma_\theta(r)$ 的估计元组数。

  如果考虑空值，我们可以估计对条件 θ 的取值为未知的元组数目，然后从上面忽略空值的估计中减去该数目。对该数目的估计需要在目录中维护附加的统计信息。

13.3.3 连接运算结果大小的估计

在本节中，我们考虑如何估计连接运算结果集的大小。

笛卡儿积 $r \times s$ 包含 $n_r * n_s$ 个元组。$r \times s$ 中的每个元组占用 $l_r + l_s$ 个字节，由此我们可以计算出笛卡儿积的大小。

估计自然连接的大小比估计选择或笛卡儿积的大小要更复杂一些。令 $r(R)$ 和 $s(S)$ 为两个关系。

- 若 $R \cap S = \emptyset$ ——两个关系没有共同的属性——则 $r \bowtie s$ 与 $r \times s$ 结果一样，我们可用估算笛卡儿积结果集大小的方法进行估计。
- 若 $R \cap S$ 是 R 的码，则我们可知 s 的一个元组至多与 r 的一个元组相连接。因此，$r \bowtie s$ 中的元组数不会超过 s 中元组的数目。$R \cap S$ 是 S 的码的情形同上面的情形相对称。若 $R \cap S$ 构成了 S 中参照 R 的外码，则 $r \times s$ 中的元组数正好与 s 中元组数相等。
- 最难的情形是 $R \cap S$ 既不是 R 的码也不是 S 的码。在这种情况下，与进行选择运算的情况一样，我们假定每个值等概率出现。考虑 r 的元组 t，假定 $R \cap S = \{A\}$。我们估计元组 t 在 $r \times s$ 中产生：

$$\frac{n_s}{V(A,s)}$$

个元组，因为该值就是关系 s 中在属性 A 上具有给定值的平均元组数目。考虑 r 中的所有元组，我们估计在 $r \times s$ 中有：

$$\frac{n_r * n_s}{V(A,s)}$$

个元组。注意，如果将 r 与 s 的角色颠倒，那么我们估计在 $r \bowtie s$ 中有：

595

$$\frac{n_r * n_s}{V(A,r)}$$

个元组。在 $V(A,s) \neq V(A,r)$ 时，这两个估计是不同的。若发生这种情况，就可能有未参加连接的悬挂元组存在。因此这两个估计值中较小者可能比较准确。

当 r 中属性 A 的 $V(A, r)$ 个值与 s 中属性 A 的 $V(A, s)$ 个值相等的情况很少时，上述连接结果集大小估计值可能太高。然而，实际中这种情形很少发生，因为在大部分现实关系中悬挂元组要么不存在要么只占总元组数的一小部分。

更重要的是，前面的估计是在各个值等概率出现这一假设前提下做出的。若这个假设不成立，则必须用更复杂的估算方法。例如，如果我们在两个关系的连接属性上有直方图，并且两个直方图都有相同的区间，我们就可以在每个区间中使用上述估计方法，用值落入该区间的行数来代替 n_r 和 n_s，用该区间的不同取值个数代替 $V(A, r)$ 或 $V(A, s)$。然后我们把得到的每个区间大小估计值加起来就得到总的大小估计值。我们把两个关系在连接属性上都有直方图，但直方图的区间不一样的情况留给读者作为练习。

对于 θ 连接 $r \bowtie_\theta s$，我们可以把它重写成 $\sigma_\theta(r \times s)$ 的形式，利用笛卡儿积的大小估计和 13.3.2 节讲到的选择操作的大小估计相结合的方法得到它的估计。

举例说明对连接运算结果集大小的估计方法，考虑表达式

$$student \bowtie takes$$

假设有关这两个关系的目录信息如下：

- $n_{student} = 5000$
- $f_{student} = 50$，意味着 $b_{student} = 5000/50 = 100$
- $n_{takes} = 10\,000$
- $f_{takes} = 25$，意味着 $b_{takes} = 10\,000/25 = 400$
- $V(ID, takes) = 2500$，意味着只有一半的学生选课（这是不现实的，但是我们用它来表明在这个例子中我们的大小估计是正确的），并且在平均情况下，选课的学生每人选4门课。

depositortake 的属性 *ID* 是参照 *student* 的外码，而且 *take.ID* 上没有空值，因为 *ID* 是 *take* 主码中的一部分。因此 *student* ⋈ *takes* 的大小恰好是 $n_{takes} = 10\,000$。

下面我们在不使用外码信息情况下计算 *student* ⋈ *takes* 的大小估计值。根据 $V(ID, takes) = 2500$，以及 $V(ID, student) = 5000$，我们得到两个估计值：$5000 * 10\,000/2500 = 20\,000$ 及 $5000 * 10\,000/5000 = 10\,000$，我们取较小者。在这里，估计值较小者与我们早先用外码信息计算所得结果相同。 596

13.3.4 其他运算的结果集大小的估计

我们下面大略描述了估计其他关系代数运算的结果集大小的方法。

- **投影**：形如 $\Pi_A(r)$ 的投影的计算结果大小（元组数或记录数）估计为 $V(A, r)$，因为投影去除了重复元组。
- **聚集**：${}_A\mathcal{G}_F(r)$ 的大小是 $V(r)$，因为对 A 的任意一个不同取值在 ${}_A\mathcal{G}_F(r)$ 中有一个元组与其对应。
- **集合运算**：如果一个集合运算的两个输入是对同一个关系的选择，我们可以将该运算重写成析取、合取或取反。例如，$\sigma_{\theta_1}(r) \cup \sigma_{\theta_2}(r)$ 可以重写成 $\sigma_{\theta_1 \vee \theta_2}(r)$。同理，只要参与集合运算的两个关系是对同一个关系的选择，我们可以把交集重写成合取式，差集重写成取反。这样我们就可以使用在13.3.2节中的对涉及合取、析取和取反操作的选择的估计。

如果输入不是对同一个关系的选择运算，我们可以按下面的方法进行估计：对 $r \cup s$ 大小的估计为 r 与 s 大小之和；对 $r \cap s$ 大小的估计为 r 与 s 大小之差；对 $r - s$ 大小的估计为 r 的大小。以上三种估计可能不精确，但提供了结果集大小的上界。

- **外连接**：$r ⟕ s$ 的大小估计为 $r ⋈ s$ 的大小加上关系 r 的大小；对 $r ⟖ s$ 的估计与对 $r ⟕ s$ 的估计是对称的，而 $r ⟗ s$ 的估计为 $r ⋈ s$ 的大小加上关系 r 和关系 s 的大小。以上三种估计可能不精确，但提供了结果集大小的上界。

13.3.5 不同取值个数的估计

对于选择操作来说，其选择结果集中的某个属性（或属性集）A 上的不同取值个数 $V(A, \sigma_\theta(r))$ 可以按如下方式进行估计：

- 若选择条件 θ 令 A 取一个特定值（例如 $A = 3$），则 $V(A, \sigma_\theta(r)) = 1$。
- 若 θ 令 A 取一个指定值的集合（例如 $A = 1 \vee A = 3 \vee A = 4$），则 $V(A, \sigma_\theta(r))$ 即为这些指定值的个数。
- 若选择条件 θ 为 A *op* v 的形式，其中 *op* 为一个比较运算符，则 $V(A, \sigma_\theta(r))$ 估计为 $V(A, r) * s$，这里 s 是该选择操作的中选率。
- 对于选择操作的其他情况，我们假设属性 A 值的分布独立于选择条件给出的值的分布，则可以得到大约为 $\min(V(A, r), n_{\sigma_\theta(r)})$ 的估计值。对此种情况，可以使用概率理论推导出更精确的估计，但上述粗略估计已经很好了。 597

对于连接操作，其连接结果集中的某个属性（或属性集）A 上的不同取值个数 $V(A, r ⋈ s)$ 可以按如下方式进行估计：

- 若 A 中所有的属性全来自 r，则 $V(A, r \times s)$ 可估计为 $\min(V(A, r), n_{r \times s})$；类似地，若 A 中所有的属性全来自 s，则 $V(A, r \times s)$ 可估计为 $\min(V(A, s), n_{r \times s})$。
- 若 A 包含来自 r 的属性 $A1$ 和来自 s 的属性 $A2$，则 $V(A, r ⋈ s)$ 估计为：

$$\min(V(A1, r) * V(A2 - A1, s), V(A1 - A2, r) * V(A2, s), n_{r \times s})$$

注意到可能有些属性既在 $A1$ 中又在 $A2$ 中，令 $A1 - A2$ 和 $A2 - A1$ 分别代表只来自 r 和只来自 s

的 A 中的属性。同上，此种情况可以使用概率理论推导出更精确的估计，但上述粗略估计已经很好了。

不同取值的数目估计对于投影来说非常直截了当：在 $\Pi_A(r)$ 中和在 r 中的估计值是一样的。这一点在聚集操作的分组属性上也同样成立。为简便起见，对于 **sum**、**count** 和 **average** 的结果，我们可以假设所有的聚集值是各不相同的。对于 $\min(A)$ 和 $\max(A)$，不同取值的个数可估计为 $\min(V(A, r), V(G, r))$，这里的 G 代表分组属性。我们略去对其他操作的不同取值数估计的详细介绍。

13.4 执行计划选择

由于表达式中的每个运算可用不同的算法实现，因此产生表达式仅仅是查询优化过程的一部分。因此一个执行计划需要准确定义每个运算应使用什么算法以及如何协调各运算的执行。

利用 13.3 节的有关技术所估计的统计信息和第 12 章描述的对不同算法和计算方法的代价估计，我们可以对给定的执行计划进行代价估计。

基于代价的优化器(cost-based optimizer)从给定查询等价的所有查询执行计划空间中进行搜索，并选择估计代价最小的一个。我们已经看到如何使用等价规则来产生等价计划。采用随意等价规则并且基于代价的优化器是相当复杂的。首先 13.4.1 节介绍一个简单版本的基于代价的优化器，其中仅涉及连接顺序和连接算法选择。然后，13.4.2 节将简单描述如何创建一个基于等价规则的通用优化器，而不过多深入细节。

对于复杂查询来说，搜索整个可能的计划空间代价太高。许多优化器采用启发式方法来降低查询 [398] 优化的代价，同时承担找不到最优计划的潜在风险。13.4.3 节介绍一些启发式方法。

13.4.1 基于代价的连接顺序选择

SQL 中最常见的查询是由连接谓词以及 **where** 子句指定的选择所构成的几个关系的连接。在本节中，我们考虑如何为这类查询选择最优的连接顺序的问题。

对于一个复杂的查询，等价于给定查询计划的不同查询计划可能很多。为了说明这一点，考虑表达式：

$$r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

其中连接运算没有指定任何顺序。当 $n = 3$ 时，有 12 种不同的连接顺序：

$$r_1 \bowtie (r_2 \bowtie r_3) \quad r_1 \bowtie (r_3 \bowtie r_2) \quad (r_2 \bowtie r_3) \bowtie r_1 \quad (r_3 \bowtie r_2) \bowtie r_1$$
$$r_2 \bowtie (r_1 \bowtie r_3) \quad r_2 \bowtie (r_3 \bowtie r_1) \quad (r_1 \bowtie r_3) \bowtie r_2 \quad (r_3 \bowtie r_1) \bowtie r_2$$
$$r_3 \bowtie (r_1 \bowtie r_2) \quad r_3 \bowtie (r_2 \bowtie r_1) \quad (r_1 \bowtie r_2) \bowtie r_3 \quad (r_2 \bowtie r_1) \bowtie r_3$$

一般而言，对于 n 个关系，有 $(2(n-1))!/(n-1)!$ 个不同的连接顺序。(我们将该式的计算留作练习 13.10)。对于涉及少量关系的连接而言，该数目还是可以接受的。例如 $n = 5$，此数是 1680。然而，当 n 增加时，这个数字迅速增长。对于 $n = 7$，此数变为 665 280；当 $n = 10$，此数大于 176 亿！

幸运的是，我们不必产生与给定表达式等价的所有表达式。例如，假设我们想找到以下表达式的最佳连接顺序：

$$(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$$

该表达式表示 r_1、r_2、r_3 首先进行连接(以某种顺序)，其结果再与 r_4、r_5 进行连接(以某种顺序)。计算 $r_1 \bowtie r_2 \bowtie r_3$ 有 12 种不同的连接次序。其结果再与 r_4、r_5 进行连接时又有 12 种顺序。因此，似乎我们需要检查 144 种连接顺序。然而，一旦我们给关系子集 (r_1, r_2, r_3) 找到了最佳连接顺序，我们可以用此顺序进一步与 r_4、r_5 进行连接。舍弃 $r_1 \bowtie r_2 \bowtie r_3$ 中其他代价较大的连接顺序。这样，我们不必一一检查 144 种连接顺序，而只须检查 12 + 12 种顺序。

```
procedure FindBestPlan(S)
        if (bestplan[S].cost ≠ ∞) // bestplan[S]已经计算好了
            return bestplan[S]
        if (S只包含一个关系)
            根据访问S的最佳方式设置bestplan[S].plan和bestplan[S].cost
    else for each S的非空子集S1，且S1 ≠ S
        P1 = FindBestPlan(S1)
        P2 = FindBestPlan(S - S1)
        A = 连接P1和P2的结果的最佳算法
        cost = P1.cost + P2.cost + A的代价
        if cost < bestplan[S].cost
            bestplan[S].cost = cost
            bestplan[S].plan = "执行P1.plan；执行P2.plan；
                    利用A连接P1和P2的结果"
    return bestplan[S]
```

图 13-7 连接顺序优化的动态规划算法

利用这种想法，我们可以设计一个动态规划算法来寻找最佳连接顺序。动态规划算法存储计算结果并将其重用，这个过程能大大缩短执行时间。

图 13-7 是对动态规划算法的一个递归程序实现。该程序尽早地在访问单个关系时就使用选择。很容易理解程序假设所有连接都是自然连接，尽管使用其他连接条件时过程也没有改变。采用任意连接条件，两个子表达式的连接可以理解为包含所有与两个子表达式属性相关的连接条件。

此过程将它计算出的执行计划存储在一个以关系集做索引的关联数组 *bestplan* 中。该关联数组的每个项包含两部分：S 的最佳计划的代价和该计划本身。$bestplan[S].cost$ 的值在未计算前初始化为 ∞。

该过程首先检查是否已算出对给定关系集 S 计算连接的最佳执行计划（并将其存储在联合数组 *bestplan* 中）；若是，则返回计算出的执行计划。

如果 S 只包括一个关系，访问 S（S 上的选择操作也考虑在内，如果有的话）的最佳方法记录在 *bestplan* 中。这个过程可能涉及用一个索引来定位元组，然后取出元组（通常称为索引扫描），或者扫描整个关系表（通常称为关系表扫描）。[⊖]如果除了通过索引扫描的选择条件，S 上还有其他选择条件，则添加一个选择运算到计划中以保证满足 S 上的所有选择。

否则，如果 S 包括多个关系，该过程尝试每一种将 S 分成两个不相交子集的方法。对每一种划分，该过程对两个子集中的每个子集递归地找出最佳执行计划，然后用该划分计算出整个执行计划的代价。[⊖]该过程从将 S 分成两个子集的所有可选方案中找出最佳方案，最佳方案及其代价存在数组 *bestplan* 中，然后由该过程返回。该过程的时间复杂度可证明为 $O(3^n)$（见实践练习 13.11）。

599
600

实际上，由关系集的连接生成的元组的顺序对于找到总体上最佳的连接顺序也是很重要的，因为它可以影响下一步连接的代价（如使用归并连接时）。若某个具体的元组排序顺序对后面的运算可能有用的话，我们称该特定的顺序是一个**感兴趣的排序顺序**（interesting sort order）。例如，连接 $r_1 \bowtie r_2 \bowtie r_3$ 所产生的结果对在 r_1 或 r_2 的公共属性上进行排序可能是有用的；而若结果在仅仅是 r_1 与 r_2 的公共属性上进行排序，就没有什么用处。在计算 $r_1 \bowtie r_2 \bowtie r_3$ 时，使用归并连接可能比使用其他连接技术代价大，但其输出结果可能是感兴趣的排序顺序。

因此，仅为有 n 个给定关系的关系集合的每个子集找出最佳连接顺序是不够的。我们应当为每个子集，为该子集连接结果的每个感兴趣的排序顺序找出最佳连接顺序。n 个关系的子集总数是 2^n，但感兴趣的排序顺序数目一般不大。因此，约有 2^n 个连接表达式需要存储。用于寻找最佳连接顺序的动

⊖ 如果一个索引包括用于查询的一个关系上的所有属性，则可以进行一次仅基于索引的扫描，该过程仅从索引中检索所需的属性，而不必取出实际的元组。

⊖ 请注意，连接 $P1$ 和 $P2$ 时考虑索引查询来循环连接，如果 $P2$ 只有一个关系 r，而且 r 的连接属性上有索引，则将 $P2$ 作为内层关系，基于 r 上的选择条件，计划 $P2$ 可能包含 r 的索引访问。为了使用索引嵌套循环连接，$P2$ 中不采用在 r 上使用选择条件进行索引查找，而是在 r 的连接属性索引返回的元组集上再检查选择条件。

态规划算法可以很容易地以扩展，以处理排序顺序。扩展的算法的代价依赖于关系集的每一个子集的感兴趣的排序顺序的数目，由于实际上这个数目并不大，因此该算法的时间代价仍为 $O(3^n)$。当 $n=10$ 时，该数约为 59 000，比之于 176 亿个不同的连接顺序好多了。更重要的是，所需存储比原先少得多，因为对于 r_1，…，r_{10} 的 1024 个子集的每一个感兴趣的排序顺序，我们只需要保存一次连接顺序。虽然这两个数仍随 n 迅速增长，但是通常参与连接的关系数少于 10，因而可以很容易地处理。

13.4.2 采用等价规则的基于代价的优化器

我们刚才看到的连接顺序优化可以处理最常见的查询，并执行关系集合的内连接。然而，显然许多查询使用其他的功能，例如聚集、外连接、嵌套查询。这些是无法通过连接顺序选择解决的。

许多优化器采用基于启发式转换的方法处理连接以外的结构，并且采用基于代价的连接顺序选择算法处理仅包含连接和选择的子表达式。我们不涉及大部分与具体优化器相关的启发式方法的细节。然而，启发式转换广泛地应用于处理嵌套查询中。13.4.4 节将介绍更多细节。

然而，本节描述如何创建一个采用等价规则的基于代价的通用优化器。这种优化器可以处理各种各样的查询结构。

采用等价规则的好处是它易于扩展新的规则到优化器中来处理不同的查询结构。例如，嵌套查询
601 可以使用扩展的关系代数结构来表示，而且可以使用等价规则表示嵌套查询的转换。还可以为聚集和外连接操作定义等价规则，就像实践习题 13.1 和习题 13.2 描述的那样。

在 13.2.4 节中，我们看到了一个优化器如何系统地产生与给定查询等价的所有表达式。产生等价表达式的过程可以按如下方式修改为产生所有可能的执行计划：添加一类新的称为**物理等价规则**（physical equivalence rule）的等价规则，允许将例如连接这样的逻辑操作转换成像散列连接或嵌套循环连接这样的物理操作。通过将这类规则添加到原来的等价规则中，程序可以产生所有可能的执行计划。前面我们看到的代价估计技术可以用来选择最优的（即代价最低的）计划。

然而，即使我们不考虑产生执行计划，13.2.4 节介绍的程序代价也非常高。为了使该方法高效需要以下技术：

1. 一种节省空间的表达式表示形式，来避免应用等价规则时创建相同子表达式的多个副本。
2. 检测相同表达式重复推导的有效技术。
3. 一种基于缓存（memoization）的动态规划形式，当第一次优化子表达式时，该缓存存储最优的查询执行计划；后续优化相同子表达式的请求通过返回已经缓存的计划进行处理。
4. 通过维护到任何时刻为止为任何子表达式产生的代价最低的计划，并且对其他任何比当前该子表达式代价最低计划的代价高的计划进行剪枝，避免产生所有可能的执行计划的技术。

具体细节要更复杂。这个方法由 Volcano 研究项目率先提出，并且 SQL Server 的查询优化器也是基于该技术。更多资料请参见文献注释。

13.4.3 启发式优化

基于代价优化的一个缺点是优化本身的代价。虽然查询优化的代价可以通过巧妙的算法来减少，但一个查询的不同执行计划数仍可能很大，在这个集合里找到最优计划仍需要很多计算代价。因此，查询优化器使用**启发式方法**（heuristics）来减少优化代价。

下面是启发式规则的一个例子。此规则用于对关系代数查询进行转换：
602 • 尽早执行选择运算。

一个启发式优化器不验证采用本规则转换后代价是否减少，而是直接加以使用。在 13.2 节的第一个转换例子中，选择运算被推进连接之中。

我们说以上规则是启发式的，因为这条规则通常会，但不总是有助于减少代价。我们来看一个使用以上规则将导致代价增加的例子，考虑表达式 $\sigma_\theta(r \bowtie s)$，其中 θ 只引用 s 的属性。选择的确可以先于连接计算。然而，若 r 相对于 s 非常小，并且在 s 的连接属性上有索引，但在 θ 所引用的属性上无索引，先做选择很可能不是个好主意。先做选择运算意味着直接对 s 进行选择，这需要对 s 个体元组进行一次扫描。就本例子而言，先用索引计算连接，然后去除不满足选择条件的元组，代价可能更小。

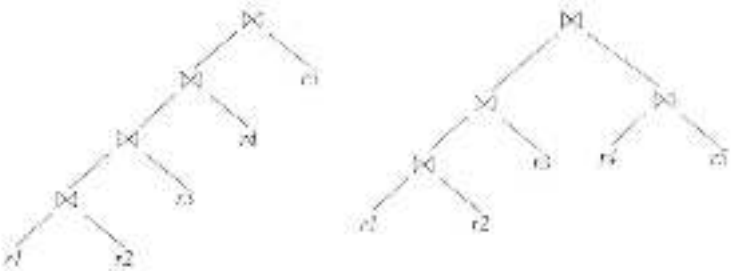
投影运算像选择运算一样可以减少关系的大小。因此，每当我们要产生临时关系时，只要有可能就立即执行投影是有好处的。该好处提示我们，伴随前面的"尽早执行选择运算"而来的还有一条规则：

- 尽早执行投影运算。

通常选择运算优先于投影运算执行比较好，因为选择运算可能会大大减小关系；并且选择运算可以利用索引存取元组。我们可以用类似于启发式选择规则中使用的例子来说明该启发式规则不总是有助于减少代价。

多数现实的查询优化器有更多的启发式规则来减少优化的代价。例如，许多查询优化器（如 System R 优化器[⊖]）并不考虑所有的连接顺序，而是只对特殊类型的连接次序进行搜索。System R 优化器仅考虑右操作对象是原始关系 r_1，r_2，…，r_n 之一的那些连接顺序。这种连接顺序称为**左深连接顺序**（left-deep join order）。左深连接顺序用于流水线计算特别方便，因为右操作对象是一个已存储的关系，每个连接只有一个输入来自流水线。

图 13-8 说明了左深连接树与非左深连接树的区别。考虑全体左深连接顺序所需时间代价是 $O(n!)$，比考虑所有连接顺序少得多。使用动态规划的优化技术，System R 优化器可以用近似 $O(n2^n)$ 的时间找到最佳连接顺序。读者可以把这一代价同找出总体上最佳的连接顺序所需时间 $O(3^n)$ 相比较。System R 优化器使用启发式规则在查询树中将选择与投影运算往下推。



a) 左深连接树　　b) 非左深连接树

图 13-8　左深连接树

某些版本的 Oracle 系统最初采用一种减少连接顺序选择代价的启发式优化方法，该方法大体上是这样进行的：对于一个 n 路连接，它考虑 n 个评估计划。每个计划使用一个左深连接顺序，各个计划 603
从一个不同的关系开始进行。通过基于可用的存取路径的排序反复选择参加下一个连接的"最佳"关系，启发式方法分别为 n 个执行计划构造连接顺序。基于可用的存取路径，每一个连接或选择嵌套循环连接算法，或选择排序归并连接算法。最后，基于使在内层关系上没有索引的嵌套循环连接的个数最小以及使排序归并连接的个数最小的原则，用启发式方法从 n 个执行计划中选一个。

一些系统采用将启发式计划选择与生成候选存取计划集成在一起的查询优化方法。System R 及其后续的 Starburst 项目采用基于 SQL 嵌套块概念的一个层次化过程。这里描述的基于代价的优化技术独立地应用到每个查询块上。一些数据库产品的优化器，如 IBM DB2 和 Oracle，就是基于上述方法并进行了扩展以处理其他操作（如聚集操作）。对于复合 SQL 查询语句（使用了 ∩、∪、− 运算），优化器对每个成分分别进行处理，然后所有执行计划结合在一起形成总的执行计划。

许多优化器允许为查询优化指定一个成本预算。当超过**优化成本预算**（optimization cost budget）时停止搜索最优计划，返回当前找到的最优计划。预算本身可能动态设置：例如，如果一个查询找到一个低开销的计划，则降低预算，使得当前找到的最优计划开销已经很低的时候，不会再花更多时间去优化查询。另一方面，如果当前找到的最优计划比较昂贵，则容易理解需要更多时间去优化，这会带来执行时间的明显减少。为了更好地利用这一思想，优化器通常先采用低价的启发式找到一个计划，然

⊖ System R 是 SQL 最早实现之一，而且它的优化器开创了基于代价的连接顺序优化。

604 后在已选计划的预算下，采用完整的基于代价的优化。

许多应用都会反复执行同样的查询，不过查询中的常数值不一样。例如，一个大学的应用可能反复执行一个查询来查找某个学生的课程注册情况，不过每次对不同的学生使用不同的 ID。作为一个启发式优化方法，许多查询优化器只对查询进行一次优化并缓存该查询执行计划，无论该查询最初提交的时候使用了什么常数值。每当该查询再一次执行，尽管可能这次使用新的常数值，缓存的查询执行计划还是会被重用（当然是使用新的数值）。对于新的常数值该查询的最优计划可能不同于由初始值确定的最优计划，但是作为启发式优化方法，该缓存的计划仍被重用。[⊖] 缓存和重用查询计划称为计划**缓存**（plan caching）。

即使使用启发式方法，基于代价的查询优化仍给查询处理带来不少开销。然而，增加的开销通常小于查询执行时间的节省，查询执行时间主要花在慢速的磁盘存取上。好的计划与差的计划在查询执行时间上差别可能很大，因此查询优化非常重要。如果应用中有许多经常运行的操作，那么其中的查询可以只优化一次，选中的查询计划存以后每一次运行时使用，在这样的应用中查询优化所能带来的代价节省更为明显。因此，大部分商品化系统都包含了相对复杂的优化器。文献注解给出了对实际数据库系统查询优化器进行讲解的参考资料。

13.4.4 嵌套子查询的优化*

SQL 从概念上将位于 **where** 子句中的嵌套子查询处理成传入参数并且返回一个单独值或一个值的集合（可能为空集）的函数。参数是嵌套子查询中用到的外层查询的变量（这些变量称作相关变量（correlation variable））。例如，假设我们有下面的查询，查找在 2007 年教授一门课程的所有教师的名字：

```
select name
from instructor
where exists (select *
              from teaches
              where instructor.ID = teaches.ID and teaches.year = 2007 );
```

605 在概念上，该子查询可视为一个函数，它获得一个参数（这里是 *instructor.ID*），并返回（具有同一 *ID* 的）教师 2007 年教授的所有课程的集合。

SQL（在概念上）按以下方式执行整个查询：首先计算位于外层查询的 **from** 子句中的关系的笛卡儿积，然后对结果中的每个元组用 **where** 子句中的谓词进行测试。在上述的例子中，就是测试子查询运算结果是否为空。

这种执行具有嵌套子查询的查询的技术称为**相关执行**（correlated evaluation）。相关执行效率不高，因为子查询对外层查询的每一个元组都进行单独的运算，从而可能导致大量的随机磁盘 I/O 操作。

因此，SQL 优化器尽可能地试图将嵌套子查询转换成连接的形式。有效的连接算法有助于避免昂贵的随机 I/O。在没有可能进行转换的情况下，优化器将子查询当作一个单独的表达式，单独优化它们，然后再进行相关执行。

作为将嵌套子查询转换为连接的例子，上述例子的查询可以重写成：

```
select name
from instructor, teaches
where instructor.ID = teaches.ID and teaches.year = 2007
```

（为了正确地反映 SQL 语义，该查询引出的重复元组数不能因重写而改变，稍后我们可以看到，可以修改这个重写的查询以保持这个性质。）

在这个例子中，嵌套子查询非常简单。而一般情况下，都不能将嵌套子查询关系直接移入外层查询的 **from** 子句里。取而代之，我们先建立一个包含嵌套查询的结果，但没有用取自外层查询的相关变量进行选择操作的临时关系，然后将这个临时表与外层查询进行连接。例如，下面形式的查询：

⊖ 对于学生注册查询，任何学生 ID 的查询计划几乎是相同的，但是如果一个查询涉及一个范围内的学生 ID，并且返回这个范围内所有学生 ID 的注册情况，则与范围大的情况相比较，范围小的情况可能有不同的最优计划。

```
select ···
from L1
where P1 and exists ( select *
                      from L2
                      where P2 )
```

其中 P_2 是一些较简单谓词的合取，可以重写为：

```
create table t1 as
        select distinct V
        from L2
        where P2'
select ···
form L1 , t1
where P1 and P2''
```

这里的 P_2' 包含 P_2 中的不涉及相关变量选择的谓词，而 P_2'' 又重新引入涉及相关变量选择的谓词（谓词中引用的关系相应地重命名）。这里的 V 包含嵌套子查询的相关变量选择中使用到的所有属性。 606

在该例子里，初始的查询可以转换成：

```
create table t1 as
    select distinct ID
    from teaches
    where year = 2007 ;
select name
from instructor , t1
where t1. ID = instructor. ID ;
```

如果我们不在乎每个元组有多少个重复的话，我们为描述临时关系的建立而重写的这个查询可以通过简化上述转换产生的查询得到。

这种用一个具有连接的查询（可能使用临时关系）去替代嵌套查询的过程称为去除相关（decorrelation）。

当嵌套子查询使用聚集，或者当嵌套子查询的结果用于等值测试，或者当嵌套子查询与外部查询的关联条件是 **not exists** 时，去除相关操作会更复杂。我们并不试图给出通用的算法，而是在文献注解里给出相关的条目。

从上述讨论可知，复杂嵌套子查询的优化是一个困难的工作，许多优化器仅做少量的去除相关工作。只要有可能，最好避免使用复杂嵌套子查询，因为我们不能确信查询优化器能够成功地将它们转换成一种能够有效运算的形式。

13.5 物化视图 **

当定义一个视图的时候，一般来说数据库只存储定义该视图的查询语句。与此相反，物化视图（materialized view）是一个其内容已计算并存储的视图。物化视图带来了冗余数据，因为其内容可以通过视图的定义和数据库中其他数据得到。然而，在许多情况下，读一个物化视图的内容比通过执行定义该视图的查询得到该视图的内容代价要低得多。

在一些应用中，物化视图在提高性能方面起到很大的作用。考虑下面这个视图，它得到每个系的薪水总额：

```
create view department_total_salary(dept_name, total_salary) as
select dept_name, sum(salary)
from instructor
group by dept_name;
```

607

假设要频繁地查询某个系薪水总额。计算这个视图需要读取每一个属于该系的 *instructor* 元组，然后累加薪水，这是一个耗时的工作。相比较而言，如果薪水总额的视图定义物化了，则只要在物化视

图中查找一个元组就可以得到某个系的薪水总额[⊖]。

13.5.1 视图维护

物化视图带来一个问题，就是它们必须能够在视图定义所使用的数据变化时保持更新。例如，一个教师的 *salary* 值更新了，物化视图中数据将与原始数据不一致，它必须进行更新。这种保持物化视图与原始数据同步更新的任务称作**视图维护**(view maintenance)。

视图可以通过人工书写的代码进行维护：即更新一个 *salary* 值的代码也同时对相应系的薪水总额进行更新。然而，这种方法比较容易出错，因为很容易遗漏某些更新 *salary* 的地方，导致物化视图和原始数据不匹配。

另一个维护物化视图的选择是对视图定义中每个关系的插入、删除和更新操作定义触发器。触发器必须根据导致触发器触发的变化对物化视图的内容进行修改。实现这个目的的一个简单方法就是每次更新时，对该物化视图全部重新进行计算。

一个更好的方法是只对物化视图的受影响部分进行修改，称为**增量的视图维护**(incremental view maintenance)。13.5.2 节将描述如何进行增量的视图维护。

现代数据库系统对增量的视图维护提供了更直接的支持。数据库系统编序员不再需要为视图维护定义触发器。取而代之的是，一旦一个视图声明为物化视图，数据库系统计算该视图的内容并在原始数据变化时对其进行增量更新。

大多数数据库系统执行**立即的视图维护**(immediate view maintenance)，即当更新发生时，增量的视图维护作为更新事务的一部分立即执行。一些数据库系统还提供**延迟的视图维护**(deferred view maintenance)，即视图维护延迟到更晚一些执行。例如，可能在白天收集更新，在晚上进行物化视图更新。这个方法减少了更新事务的开销，但是，使用延迟视图维护的物化视图可能与定义它们所使用的
608 基本关系不一致。

13.5.2 增量的视图维护

为了理解如何增量地维护物化视图，我们从考虑单独的操作开始，然后看一下如何处理一个完整的表达式。

导致一个物化视图过时的关系的改变操作包括插入、删除和更新。为了描述方便，我们将更新一个元组的操作替换为删除该元组，紧跟着插入一个更新后的元组的操作。这样我们就只需要考虑插入和删除。对一个关系或表达式的改变(插入和删除)称作它的**差异**(differential)。

13.5.2.1 连接操作

考虑物化视图 $v = r \bowtie s$。假设对关系 r 作插入一个元组集(记为 i_r)的改变。如果 r 的原值记为 r^{old}，r 的新值记为 r^{new}，则有 $r^{new} = r^{old} \cup i_r$。现在，该视图的原值 v^{old} 由 $r^{old} \bowtie s$ 给出，新值 v^{new} 由 $r^{new} \bowtie s$ 给出。我们可以将 $r^{new} \bowtie s$ 重写为 $(r^{old} \cup i_r) \bowtie s$，再重写为 $(r^{old} \bowtie s) \cup (i_r \bowtie s)$。也就是：

$$v^{new} = v^{old} \cup (i_r \bowtie s)$$

因此，为了更新物化视图 v，我们仅需要将元组 $i_r \bowtie s$ 加入到物化视图的原内容中。对 s 的插入完全按对 r 的插入对称的方式进行。

现在，假设对 r 的改变是删除一个元组集(记为 d_r)。由上面所述同样的理由，我们得到：

$$v^{new} = v^{old} - (d_r \bowtie s)$$

对 s 的删除操作与对 r 完全对称的方式进行。

13.5.2.2 选择和投影操作

考虑一个视图 $v = \sigma_\theta(r)$。如果我们对关系 r 做改变，插入一个元组集(记为 i_r)，则 v 的新值可以如下计算：

⊖ 对于一个中型大学或部门并不总是很大，但是在某些其他情况下差别可能非常大。例如，如果物化视图从一个包含数百万元组的销售关系中计算每种产品的销售总数，则从基础数据计算聚集值和存储物化视图之间的差别可能是几个数量级的。

$$v^{new} = v^{old} \cup \sigma_p(i_r)$$

类似地，如果我们对关系 r 做改变，删除一个元组集（记为 d_r），则 v 的新值如下计算：

$$v^{new} = v^{old} - \sigma_p(d_r)$$

投影是一个处理起来更困难的操作。考虑一个物化视图 $v = \Pi_A(r)$。假设关系 r 建立在模式 $R = (A, B)$ 的基础上，它包含两个元组 $(a, 2)$ 和 $(a, 3)$，则 $\Pi_A(r)$ 仅包含一个元组 (a)。如果我们从 r 中删除元组 $(a, 2)$，我们不能从 $\Pi_A(r)$ 中删除元组 (a)；如果我们这样做，则结果将是一个空关系，而事实上，$\Pi_A(r)$ 仍有一个单独的元组 (a)。这是因为同一个元组 (a) 可由两条途径得到，从中删除一个元组只是去除了得到元组 (a) 的一条途径，另一条仍然存在。 609

这个理由也给出了一种直观的解决办法：对投影（例如 $\Pi_A(r)$）中的每个元组，我们将保留一个计数，记录该元组由几条途径得到。

当从关系 r 中删除一个元组集 d_r 时，对 d_r 中的每个元组 t，我们做以下操作：令 $t.A$ 表示 t 在属性 A 上的投影。我们在物化视图中找到 $(t.A)$，然后对其中存储的计数减 1；如果计数变为 0，则从物化视图中删除 $(t.A)$。

处理插入操作相对较直接。当把元组集 i_r 插入到关系 r 中时，对 i_r 中的每个元组 t，我们做以下操作：如果 $(t.A)$ 已经在物化视图中存在，我们对其中存储的计数加 1；如果没有，我们将 $(t.A)$ 加入到物化视图中去，同时令其计数值为 1。

13.5.2.3 聚集操作

聚集操作的处理过程在某种程度上与投影操作类似。在 SQL 中的聚集操作有 **count**、**avg**、**min** 和 **max**。

- **count**：考虑一个物化视图 $v = {}_A\mathcal{G}_{count(B)}(r)$，它按属性 A 对 r 进行分组然后对属性 B 计数。

  当把元组集 i_r 插入到关系 r 中时，对 i_r 中的每个元组 t，我们做以下操作：我们在物化视图中查找组 $t.A$，如果查找不到，则将 $(t.A, 1)$ 加入到物化视图中；如果存在组 $t.A$，我们对该组的计数加 1。

  当从关系 r 中删除一个元组集 d_r 时，对 d_r 中的每个元组 t，我们做以下操作：我们在物化视图中找到组 $t.A$，然后对该组的计数值减 1；如果计数变为 0，则从物化视图中删除组 $t.A$。
- **sum**：考虑一个物化视图 $v = {}_A\mathcal{G}_{sum(B)}(r)$。

  当把元组集 i_r 插入到关系 r 中时，对 i_r 中的每个元组 t，我们做以下操作：我们在物化视图中查找组 $t.A$，如果查找不到，则将 $(t.A, t.B)$ 加入到物化视图中，另外，与我们在投影中所做的处理类似，我们存储一个计数值 1 与 $(t.A, t.B)$ 相关联；如果存在组 $t.A$，我们将 $t.B$ 的值加到该组的聚集值上，并对该组的计数加 1。

  当从关系 r 中删除一个元组集 d_r 时，对 d_r 中的每个元组 t，我们做以下操作：我们在物化视图中找到组 $t.A$，然后将该组的聚集值减去 $t.B$ 的值，同时也要对该组的计数减 1。如果计数变为 0，则从物化视图中删除组 $t.A$。

  如果不保留附加的计数值，我们就无法区分一个组的求和数为 0 还是该组的所有元组都已经删除了。 610
- **avg**：考虑一个物化视图 $v = {}_A\mathcal{G}_{avg(B)}(r)$。

  当插入和删除时直接更新一个视图的平均值是不可能的，因为这不仅依赖于原值和插入/删除的元组，而且还依赖于该组的元组数。

  对于 **avg** 这种情况的处理，可以采用另外的方法：我们用以前所述的方法维护 **sum** 和 **count** 聚集值，然后用总和除以计数来得到平均值。
- **min**, **max**：考虑一个物化视图 $v = {}_A\mathcal{G}_{min(B)}(r)$。（**max** 的情形完全类似。）

  对关系 r 上的插入操作的处理是直截了当的，但对关系 r 上的删除操作，维护聚集值 **min** 和 **max** 要更复杂一些。例如，如果从关系 r 中删除了某个组的对应最小值的那个元组，我们就必须查看关系 r 中同组的其他元组，从而找到新的最小值。

13.5.2.4 其他操作

对集合操作交的维护如下：对于给定的物化视图 $v = r \cap s$，当插入一个元组到关系 r 中时，我们检

查它是否在关系 s 中，如果在，则将它加入到 s 中。从关系 r 中删除一个元组时，如果它在交集合中存在，我们就将其从交集中删除。其他集合操作，并和集合差，可采用相似的方法进行处理。其处理的细节留给读者。

外连接的处理与连接的处理几乎完全一致，除了要做几个额外的工作外。对于在关系 r 上的删除，我们必须处理 s 中不再同 r 中任何元组相匹配的元组；对于在关系 r 上的插入，我们必须处理 s 中原先不与 r 中任何元组相匹配的元组，其处理的细节再次留给读者。

13.5.2.5 表达式的处理

到目前为止，我们已经明白了如何对一个单独的操作的结果进行增量更新。对于一个完整表达式的处理，我们可以从最小的子表达式开始，并导出用于计算每一个子表达式结果的增量变化的表达式。

例如，当一个元组集 i_r 插入到关系 r 中时，我们想要增量更新物化视图 $E_1 \bowtie E_2$。假设关系 r 只在 E_1 中使用，并假设要插入到 E_1 中的元组集由表达式 D_1 给出，则表达式 $D_1 \bowtie E_2$ 给出了要插入到 $E_1 \bowtie E_2$ 中的元组集。

对于表达式有关的增量视图维护的更进一步讨论可参见文献注解。

13.5.3 查询优化和物化视图

可以将物化视图看作普通的关系那样执行查询优化。然而，物化视图为优化提供了更有利的机会。

[611]

- 重写查询以利用物化视图：

  假设有物化视图 $v = r \bowtie s$ 可用，而且用户提交了一个查询 $r \bowtie s \bowtie t$。与直接优化用户所提交的查询相比，将该查询重写为 $v \bowtie t$ 可以提供比直接优化更有效的查询计划。因此，查询优化器的工作应该包括知道何时可利用物化视图来提高查询处理速度。

- 将使用物化视图的地方替换成它的定义：

  假设有物化视图 $v = r \bowtie s$ 可用，但其上没有任何索引，而用户提交了一个查询 $\sigma_{A=10}(v)$。假设关系 s 在公共属性 B 上有索引，关系 r 在属性 A 上有索引。该查询最好的执行计划可能是将 v 替换成 $r \bowtie s$，运行执行计划 $\sigma_{A=10}(r) \bowtie s$，这样通过分别使用 $r.A$ 和 $s.B$ 上的索引可以使选择和连接操作得到高效的执行。比较而言，直接在 v 上执行选择操作可能需要对 v 执行一次完全扫描，执行代价可能会更高。

文献注解给出了有关如何利用物化视图进行有效的查询优化的研究著作。

13.5.4 物化视图和索引选择

另外一个相关的优化问题是**物化视图选择**(materialized view selection)，顾名思义，就是"物化哪些视图集是最佳方案?"这个视图集的选择必须基于系统的**工作负载**(workload)，即反映系统通常负载的一系列查询和更新操作。一个简单的标准是，选择的物化视图集应能够使系统完成查询和更新的工作负载所耗费的总执行时间(包括维护物化视图所用的时间)最短。考虑到不同查询和更新的重要性不同，数据库管理员通常会修改这个标准：有些查询和更新需要有快速反应速度，而其余的可接受慢的反应速度。

索引在一定意义上与物化视图类似，它们也是衍生的数据，它们能够提高查询速度，也可能减慢更新速度。因此，**索引选择**(index selection)的问题虽然更简单一些，但它与物化视图选择的问题的关系也很密切。24.1.6～24.1.7节会对这些问题做详细讨论。

许多数据库系统提供了一些帮助数据库管理人员进行索引和物化视图选择的工具。这些工具检查查询和更新的历史纪录，并建议可进行物化的索引和视图。Microsoft SQL Server Database Tuning Assistant、IBM DB2 Design Advisor 和 Oracle SQL Tuning Wizard 等都是这种类型的工具。

13.6 查询优化中的高级话题*

[612]

除了我们已经介绍过的方法，还有许多优化查询的方法。本节介绍其中的一些。

13.6.1 top-K 优化

许多查询都要求返回在某些属性上排了序的结果，并且对于某特定 K 值，只取结果中的前 K 个。

有时范围 K 是显式指定的。例如，一些数据库支持 limit K 子句，它导致从查询结果中只返回前 K 个结果。其他数据库支持另一些方式的类似限制。在其他情况下，查询不指定一个限制，但优化器会允许指定一个提示，表明尽管查询可能会产生更多结果，但是只有前 K 个结果可能会被检索。

当 K 值很小时，如果一个查询优化计划先产生整个结果集，然后排序并产生前 K 个结果，这样做效率会非常低，因为绝大部分计算出来的中间结果都要舍弃。人们提出了一些技术来优化这类 top－K 查询。一种方法是使用能够产生有序的结果的流水线查询计划。另一种方法是估计将会出现在前 K 个输出中的排序属性上最大的值是什么，然后引入选择谓词剔除较大的值。如果在前 K 个值以外产生了多余的元组则舍弃掉，如果产生的元组过少则改变选择条件并重新执行选择操作。top－K 优化的相关研究请参见文献注释。

13.6.2 连接极小化

当查询通过视图产生时，有时实际进行连接的关系数比计算该查询所必须进行连接的关系数要大。例如，一个视图 v 可能包含 *instructor* 和 *department* 的连接，但是 v 的一个使用可能只涉及 *instructor* 的属性。*instructor* 的连接属性 *dept_name* 是指向 *department* 的外键。假设 *instructor.dept_name* 声明为非空，则与 *department* 的连接可以去掉，对于查询没有任何影响。因为在上述假设下，与 *department* 的连接既没有删除 *instructor* 中的任何元组，也没有产生 *instructor* 元组的额外拷贝。

上述从连接中去掉一个关系就是连接极小化的一个例子。事实上，连接极小化也可以用于其他情况。连接极小化的相关研究请参见文献注释。

13.6.3 更新的优化

更新查询通常在 set 和 where 子句中涉及子查询，这些也都要在更新优化时考虑在内。涉及在一个更新列上的选择条件的更新（例如给所有工资大于等于 $100 000 的职员加薪 10%）必须小心处理。如果更新是在选择条件扫描索引的执行过程中完成，一个更新的元组可能会在扫描之前和扫描之后重复插入索引两次；同一个职员的元组可能不正确地更新多次（在这种情况下可能是无限次）。涉及结果更新的子查询也同样会有相似的问题。

更新操作本身可能影响自己执行的问题称为万圣节问题（Halloween problem，在 IBM 发现这个问题 613
时以当天的节日命名）。这个问题可以通过执行这样的查询实现：首先定义更新，创建受到影响的元组列表，并在最后更新元组和索引。然而，把执行计划以这种方式分割会增加执行代价。更新计划可以通过先检查万圣节问题会否发生来进行优化，如果没有出现，则更新可以在查询执行的时候完成，从而减少更新开销。例如，如果更新不影响索引属性，则万圣节问题不会发生。即使影响索引属性，如果更新减小该值，而索引扫描以升序执行，更新过的元组在扫描时不会再次遇到。在这种情况下，查询执行的同时可以更新索引，从而降低总体成本。

导致大量更新的更新查询可能通过将更新批量收集，然后分别将批量的更新在其影响的索引上执行来优化。当将批量更新应用到一个索引上时，批数据先按索引序排序，这样的排序可以大大减少更新索引需要的随机 I/O 次数。

这种优化方式在许多数据库系统中都有实现。更新优化的相关研究请参见文献注释。

13.6.4 多查询优化和共享式扫描

当一起查询一起提交时，一个查询优化器可能发现不同查询之间共同的子表达式，仅执行它们一次并且在需要的时候重用。复杂的查询可能在查询的不同部分有重复的子表达式，可以采用同样的方法降低查询执行代价。这种优化称为**多查询优化**（multi-query optimization）。

公共子表达式消除（common subexpression elimination）通过计算并存储结果，优化程序中不同表达式之间共享的子表达式，在子表达式出现的地方重用结果。公共子表达式消除是编程语言编译器中优化算术表达式的一个标准优化方法。发现一批查询中为每个查询选择的执行计划中的公共子表达式在数据库查询优化中是非常有用的，并且在一些数据库中都有实现。然而，多查询优化在某些情况下可以做得更好：一个查询通常有一个以上的执行计划，相比较于为每个查询选择代价最小的执行计划，刻意地选择一个执行计划集合可能会带来更多的共享和更少的代价。更多查询优化的相关研究请参见

文献注释。

共享查询之间的关系扫描是一些数据库中实现的另一种有限形式的多查询优化。共享式扫描(shared-scan)优化的工作方式如下：不是对于需要扫描一个关系的每一个查询，都从磁盘上重复地读取该关系，而是从磁盘上读取一次数据，然后流水线地传递给每一个查询。共享式扫描在多个查询都
614 扫描一个大表("事实表"作为一个典型)时非常有用。

13.6.5 参数化查询优化

正如我们此前在13.4.3节中看到的，计划缓存是许多数据库中采用的启发式方法。回想一下，在计划缓存的情况下，如果一个查询涉及一些常数，则优化器选择的计划被缓存，并且当查询再次被提交时重用，即使查询中的常数不同。例如，假设一个查询将系的名称作为参数，并检索该系中的所有课程。采用计划缓存方法，当查询第一次执行时(例如对音乐系进行查询)选中一个计划。如果再对其他任何系进行查询，该计划被重用。

如果最优计划受查询中的常数值影响不大，则通过计划缓存重用计划是合理的。然而，如果计划受常数的影响，则可以使用参数化查询优化作为替代。

在**参数化查询优化**(parametric query optimization)中，不提供具体参数值，例如前面例子中的 *dept_name*，而对查询进行优化。然后，优化器提供几个计划，分别对于不同参数值是最优的。一个计划只有在对一些可能的参数值是最优的情况下才会被优化器输出。优化器输出的备选计划集合被存储。当提交一个带具体参数值的查询时，使用此前计算的计划集合中的最低价的计划，而不用执行一个完整的优化。找到这个最低价的计划所需的时间要远远短于重新优化的时间。参数化查询优化的相关研究参见文献注解。

13.7 总结

- 给定一个查询，一般有多种方法可以计算结果。系统负责将用户输入的查询转换成能够更有效执行的等价查询。为处理查询找出一个好的策略的过程称为查询优化。
- 复杂查询的执行涉及多次存取磁盘的操作。由于从磁盘中传输数据相对于主存速度和计算机系统的CPU速度要慢，因此进行一定量的处理以选择一个能够最小化磁盘存取的方法是完全值得的。
- 有很多等价规则供我们用于将一个表达式转化成等价表达式。我们使用这些规则系统地产生与所给查询等价的所有表达式。
615
- 每个关系代数表达式都表示某个特定的操作序列。选择查询处理策略的第一步就是找到一个关系代数表达式，使它与所给的表达式等价并且据估计有更小的执行代价。
- 数据库系统为执行一个操作所选择的策略依赖于每个关系的大小和列值的分布情况。数据库系统可以为每个关系 r 存储统计信息，从而能够基于这些可靠信息选择合适的策略。这些统计信息包括
  - 关系 r 中的元组数。
  - 关系 r 中的一个记录(元组)的大小(按字节计数)。
  - 关系 r 中的某个特定属性中出现的不同取值的数目。
- 许多数据库系统使用直方图来存储一个属性在每个区间上的取值个数。直方图通常采用取样来计算。
- 这些统计信息使得我们可以估计各种操作的结果集的大小和执行操作的代价。当处理一个查询的过程中有多个索引可用于辅助的时候，关系的统计信息特别有用，这些信息对查询处理策略的选择有很大的影响。
- 对每个表达式，我们可以用一些等价规则产生多个可选的执行计划。然后从中选择代价最小的执行计划。不少优化技术可以减少需要产生的可选表达式和执行计划的数量。
- 我们使用启发式方法来减少需要考虑的执行计划的数量，从而减少优化的代价。用于关系代数查询转换的启发式规则包括"尽早执行选择操作"、"尽早执行投影操作"和"避免笛卡儿积操作"。
- 物化视图可以用来加速查询处理。当原关系发生修改时，需要用增量的视图维护来高效地更新物化视图。利用包含一个操作的输入的变化量的代数表达式，能够完成对该操作的变化量的计算。其他与物化视图相关的问题还包括如何借助物化视图进行查询优化和如何选择需要物化的视图。
- 提出了一些先进的优化技术，包括top-K优化、连接极小化、更新优化、多查询优化和参数化查询优化。

术语回顾

- 查询优化
- 表达式转换
- 表达式的等价
- 等价规则
  - 连接的交换律
  - 连接的结合律
- 等价规则的最小集
- 等价表达式的枚举
- 统计信息的估计
- 目录信息
- 大小估计
  - 选择
  - 十选率
  - 连接
- 直方图
- 不同取值数的估计
- 随机抽样的样本
- 执行计划的选择
- 执行技术的相互作用
- 基于代价的优化
- 连接顺序的优化
  - 动态规划算法
  - 左深连接顺序
  - 诱兴趣的排列顺序
- 启发式优化
- 计划缓存
- 存取计划选择
- 相关执行
- 去除相关
- 物化视图
- 物化视图的维护
  - 重新计算
  - 增量维护
  - 插入
  - 删除
  - 更新
- 使用物化视图的查询优化
- 索引选择
- 物化视图选择
- top-K 优化
- 连接极小化
- 万圣节问题
- 多查询优化

实践习题

13.1 证明以下等价式成立。解释如何用它们提高某些查询的效率：

a. $E_1 \times \theta(E_2 - E_3) = (E_1 \times_\theta E_2) - (E_1 \times_\theta E_3)$

b. $\sigma_\theta({}_A\mathcal{G}_\gamma(E)) = {}_A\mathcal{G}_\gamma(\sigma_\theta(E))$，其中 θ 仅使用 A 的属性。

c. $\sigma_\theta(E_1 \bowtie E_2) = \sigma_\theta(E_1) \bowtie E_2$，其中 θ 仅使用 E_1 的属性。

13.2 对于下面每对表达式，给出关系实例说明表达式不等价：

a. $\Pi_A(R - S)$ 与 $\Pi_A(R) - \Pi_A(S)$

b. $\sigma_{B<4}({}_{\max(B)\text{ as }B}\mathcal{G}(R))$ 与 ${}_{\max(B)\text{ as }B}\mathcal{G}(\sigma_{B<4}(R))$

c. 在上述的表达式中，若 *max* 全改为 *min*，表达式依旧等价吗？

d. $(R \bowtie S) \bowtie T$ 与 $R \bowtie (S \bowtie T)$，换言之，自然左外连接不满足结合律。（提示：假设三个关系的模式分别为 $R(a, b_1)$，$S(a, b_2)$，$T(a, b_3)$。）

e. $\sigma_\theta(E_1 \bowtie E_2)$ 与 $E_1 \bowtie \sigma_\theta(E_2)$，其中 θ 仅使用 E_2 的属性。

13.3 SQL 语言允许关系有重复元组（见第 3 章）。

a. 对基本关系代数运算 σ、Π、$\times$、$\bowtie$、$-$、$\cup$ 和 $\cap$，给出它们在允许有重复元组出现情况下的定义，定义的方式应与 SQL 一致。

b. 检查等价规则 1～7.b 中哪一些满足 a 部分中定义的重复集版本的关系代数运算。

13.4 考虑关系 $r_1(A, B, C)$，$r_2(C, D, E)$ 和 $r_3(E, F)$，它们的主码分别为 A、C、E。假设 r_1 有 1000 个元组，r_2 有 1500 个元组，r_3 有 750 个元组。估计 $r_1 \bowtie r_2 \bowtie r_3$ 的大小，给出一个有效地计算这个连接的策略。

13.5 考虑习题 13.4 中的关系 $r_1(A, B, C)$，$r_2(C, D, E)$ 及 $r_3(E, F)$。假设除了整个模式外没有主码。令 $V(C, r_1)$ 为 900，$V(C, r_2)$ 为 1100，$V(E, r_2)$ 为 50，$V(E, r_3)$ 为 100。假设 r_1 有 1000 个元组，r_2 有 1500 个元组，r_3 有 750 个元组。估计 $r_1 \bowtie r_2 \bowtie r_3$ 的大小，给出一个有效地计算这个连接的策略。

13.6 假设关系 *department* 在 *building* 属性上有 B$^+$-树索引，此外别无其他索引。处理下列包含否定条件的选择操作的最佳方法是什么？

a. $\sigma_{\neg(\text{building} < \text{"Watson"})}(department)$

b. $\sigma_{\neg(\text{building} = \text{"Watson"})}(department)$

c. $\sigma_{\neg(\text{building} < \text{"Watson"} \wedge \text{budget} < 55000)}(department)$

13.7 考虑查询

```
select *
from r , s
where upper(r.A) = upper(s.A);
```

其中"upper"是个函数，该函数返回输入参数中所有小写字母被换成相应大写字母后的结果。

618

a. 找出你使用的数据库系统中为这个查询产生的计划。

b. 有些数据库系统对这个查询会采用非常低效的(块)嵌套循环连接。简单解释对于该查询，如何使用排列连接或者归并连接。

13.8 给出下列表达式等价的条件：

$$_{A}\mathcal{G}_{agg(C)}(E_1 \bowtie E_2) \text{ and } (_{A}\mathcal{G}_{agg(C)}(E_1)) \bowtie E_2$$

其中 agg 表示任何聚合操作。如果 agg 是 **min** 或 **max** 中的一个，上述条件可以放宽成什么？

13.9 考虑优化中感兴趣的排列顺序问题。假设给你一个查询，计算一个关系集合 S 的自然连接。$S1$ 是 S 的一个子集，$S1$ 的感兴趣的排列顺序是什么？

13.10 请说明 n 个关系可以构成 $(2(n-1))!/(n-1)!$ 个不同的连接顺序。

提示：一棵**完全二叉树**(complete binary tree)的每个内部结点都正好有两个子结点。利用以下事实：拥有 n 个叶结点的不同完全二叉树的个数为：

$$\frac{1}{n}\binom{2(n-1)}{(n-1)}$$

如果你愿意的话，你也可以从 n 个叶结点的不同二叉树的个数公式推导出 n 个叶结点的不同完全二叉树的个数。n 个叶结点的不同二叉树的个数为：

$$\frac{1}{n+1}\binom{2n}{n}$$

这个数字就是**卡特兰数**(Catalan number)，它的衍生式可以在任何一本数据结构或算法的标准教材中找到。

13.11 证明计算连接次序的最小时间代价是 $O(3^n)$。假设存储和查看关系集合的有关信息(如该集合的最佳连接次序以及该连接次序的代价)所需时间是常量。(如果你感觉做本题有困难，至少证明更宽松的时间界限 $O(2^{2n})$。)

13.12 如果像 System R 优化器那样只考虑左深连接树，证明寻找最有效连接次序的时间大约为 $n2^n$，假定只有一个感兴趣的排序次序。

13.13 考虑图 13-9 中的银行数据库，主码用下划线标识。为这个关系数据库构建以下 SQL 查询：

619

a. 对关系 *account* 写一个嵌套查询，从每一个名称以 B 打头的分行中找出具有最大余额的所有账户。

b. 不使用嵌套子查询重写前面的查询，换言之，去除查询的相关性。

c. 对去除此类查询相关性给出一个处理过程(类似于 13.4.4 节描述的那样)。

branch (<u>*branch_name*</u>, *branch_city*, *assets*)
customer (<u>*customer_name*</u>, *customer_street*, *customer_city*)
loan (<u>*loan_number*</u>, *branch_name*, *amount*)
borrower (<u>*customer_name*, *loan_number*</u>)
account (<u>*account_number*</u>, *branch_name*, *balance*)
depositor (<u>*customer_name*, *account_number*</u>)

图 13-9 习题 13.13 的银行数据库

13.14 半连接操作的集合定义如下：

$$r \ltimes s = \Pi_R(r \bowtie s)$$

其中 R 是 r 模式的属性集合。多集合版本的半连接操作返回相同的元组集合，不过每个元组的拷贝数量和其在 r 中的相同。

考虑我们在 13.4.4 节中看到的嵌套查询：找出在 2007 年教授一门课程的所有教师名字。使用多集合版本的半连接操作符写出查询的关系表达式，确保每个名字的重复数量和 SQL 查询中相同。(半连接操作符在嵌套查询去除相关中广泛应用。)

习题

13.15 假设关系 *department* 在 (*dept_name*, *building*) 属性上有 B⁺树索引。处理下列选择操作的最佳方法是什么？

$$\sigma_{(building < \text{'Watson'}) \wedge (budget < 55000) \wedge (dept\_name = \text{'Music'})}(department)$$

13.16 使用 13.2.1 节中的等价规则，证明如何通过一系列转换导出下列等价式：

a. $\sigma_{\theta_1,\theta_2,\theta_3}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(\sigma_{\theta_3}(E)))$ 620

b. $\sigma_{\theta_1 A \theta_2}(E_1 \bowtie_{\theta} E_2) = \sigma_{\theta_1}(E_1 \bowtie_{\theta} (\sigma_{\theta_2}(E_2)))$，其中 θ_2 仅使用 E_2 的属性

13.17 考虑下面两个表达式 $\sigma_\theta(E_1 \supset\!\times E_2)$ 和 $\sigma_\theta(E_1 \times E_2)$。

a. 通过举例说明两个表达式一般是不等价的。

b. 给出一个谓词 θ 上的简单条件，使得满足条件时两个表达式是等价的。

13.18 我们说一个等价规则集是完备的，如果对于任意两个等价的表达式，都可以使用一系列等价规则从其中一个表达式推导出另一个。13.2.1 节讲述的等价规则集是完备的吗？提示：考虑等式 $\sigma_{\theta_1 \vee \theta_2}(r) = \ldots$。

13.19 请解释如何用直方图来估算形如 $\sigma_{\theta_1 \vee \theta_2}(r)$ 的选择操作的大小。

13.20 假设两个关系 r 和 s 在属性 $r.A$ 和 $s.A$ 上分别有直方图，但是所取区间不同。请给出如何用直方图估计 $r \bowtie s$ 的大小。提示：进一步细分每个直方图的区间。

13.21 考虑查询

```
select A, B
from r
where r.B < some (select B
                  from s
                  where s.A = r.A)
```

说明如何使用练习 13.14 中定义的多集合版本半连接操作符去除上述查询中的相关性。

13.22 从插入和删除两方面描述如何增量维护下面操作的结果集：

a. 并和集合差

b. 左外连接

13.23 给出一个定义物化视图的表达式的例子和两种情况（有关输入关系的统计信息集合与关系数据的变化量）。其中一种情况下，增量视图维护优于重新计算；另一种情况下，重新计算更优。

13.24 假设你想得到 $r \bowtie s$ 在 r 的一个属性上排序的结果，并且只想要前 K 个结果，这里的 K 比较小。请给出执行查询的较好方案。

a. 当在 r 上的指向 s 的外码属性上进行连接时，外码属性声明为非空；

b. 当不是在外码上进行连接时。 621

13.25 考虑关系 $r(A, B, C)$，在属性 A 上有索引。给出一个查询的例子，其结果仅用索引就可得到，而不用查看关系表中的元组。（仅用索引而不必访问实际关系表的查询计划称为仅用索引的计划。）

13.26 假定你有一个更新查询 U。给出一个 U 上的简单充分条件，使得不论选择任何执行计划或者是否存在索引，万圣节问题都不会发生。

文献注解

Selinger 等[1979]所做的创造性工作描述了 System R 优化器的存取路径选择。System R 的优化器是最早的关系查询优化器之一。Starburst 的查询处理在 Haas 等[1989]中描述，这是 IBM DB2 查询优化的基础。

Graefe 和 Mckenna[1993a]描述了 Volcano，它是一个基于等价规则的查询优化器。Volcano 与它的后继者 Cascades[Graefe [1995]]形成了 Microsoft SQL server 查询优化的基础。

查询结果的统计信息的估计，比如结果集大小，在 Ioannidis 和 Poosala[1995]、Poosala 等[1996]和 Ganguly 等[1996]以及其他文献中有描述。值的非平均分布会给查询大小和代价的估计带来问题，用值分布直方图进行代价估计技术可以解决此类问题。Ioannidis 和 Christodoulakis[1993]、Ioannidis 和 Poosala[1995]，

以及 Poosala 等[1996] 在这一领域给出了成果。随机抽样广泛应用于在统计学中构建直方图，但是数据库内容中直方图的构建问题在 Chaudhuri 等[1998]中讨论。

Klug[1982]是有关包含聚集函数的关系代数表达式的优化的早期著作。包含聚集操作的查询优化在 Yan 和 Larson[1995]以及 Chaudhuri 和 Shim[1994]文中有阐述。包含外连接的查询优化在 Rosenthal 和 Reiner [1984]、Galindo-Legaria 和 Rosenthal[1992]、Galindo-Legaria[1994]中有描述。top-K 查询的优化在 Carey 与 Kossmann[1998]和 Bruno 等[2002]中有阐述。

嵌套子查询的优化在 Kim[1982]、Ganski 和 Wong[1987]、Dayal[1987]、Seshadri 等[1996]，以及较新的 Galindo-Legaria 与 Joshi[2001]中进行讨论。

Blakeley 等[1986]描述了维护物化视图的技术。物化视图执行计划的优化在 Vista[1998]和 Mistry 等[2001]中讨论。Chaudhuri 等[1995]描述了有物化视图参与的查询优化。索引的选择和物化视图的选择在 Ross 等[1996]和 Chaudhuri 和 Narasayya[1997]中讨论。

top-K 查询的优化在 Carey 与 Kossmann[1998]和 Bruno 等[2002]中有阐述。一些用于极小化连接的技术统称为表格优化。表格的概念由 Aho 等[1979b]和 Aho 等[1979a]引入，Sagiv 和 Yannakakis[1981]作了进一步扩展。

参数化查询优化算法由 Ioannidis 等[1992]、Ganguly[1998]和 Hulgeri 与 Sudarshan[2003]提出。Selli [1988]和 Roy 等[2000]描述了多查询优化。Roy 等[2000]提出如何将多查询优化集成到一个基于 Volcano 的查询优化器中。

622
624

Galindo-Legaria 等[2004]描述了数据库更新操作的查询处理和优化，包括索引维护的优化，物化视图的维护计划和约束完整性检查，以及解决万圣节问题的技术。

第四部分
Part 4

事务管理

术语事务指的是构成单一逻辑工作单元的操作的集合。比如：将钱从一个账户转到另一个账户就是一个事务，该事务包括分别针对每个账户的两个更新。

事务的所有动作要么全部执行，要么由于出错而撤销事务的部分影响，这一点非常重要，这个特性叫做原子性。而且，一旦事务成功执行，其影响必须保存在数据库中——一个系统故障不应该导致数据库忽略成功完成的事务，这个特性叫做持久性。

在一个有多个事务并发执行的数据库系统中，如果对共享数据的更新不加以控制，事务就可能看到由别的事务的更新引起的中间状态的不一致，这种情况会导致对数据库中存储的数据的错误更新。所以，数据库系统必须提供隔离机制以保证事务不受其他并发执行的事务影响。这个特性叫做隔离性。

第 14 章详细阐述事务的概念，包括事务的原子性、持久性、隔离性和由事务抽象提供的其他特性。特别地，本章通过可串行化的概念来准确描述隔离性的定义。

第 15 章阐述几种实现隔离性的并发控制技术。第 16 章阐述数据库恢复管理部件，它实现了数据库的原子性与持久性。

总体来说，数据库系统的事务管理部件使得应用程序开发人员能够把注意力集中在单个事务上，而不必考虑并发和容错等问题。

第14章
Database System Concepts, 6E

事　务

通常，从数据库用户的观点来看，数据库中一些操作的集合被认为是一个独立单元。比如，从顾客的立场来看，从支票账户到储蓄账户的资金转账是一次单一的操作；而在数据库系统中，这是由几个操作组成的。显然，有一点是最基本的，这些操作要么全部发生，要么由于出错而全不发生。资金从支票账户支出而未转入储蓄账户的情况是不可接受的。

构成单一逻辑工作单元的操作集合称作**事务**(transaction)。即使有故障，数据库系统也必须保证事务的正确执行——要么执行整个事务，要么属于该事务的操作一个也不执行。此外，数据库系统必须以一种能避免引入不一致性的方式来管理事务的并发执行。在资金转账的例子中，一个计算顾客总金额的事务可能在资金转账事务从支票账户支出金额之前查看支票账户余额，而在资金存入储蓄账户之后查看储蓄账户余额。结果，它就会得到不正确的结果。

本章介绍事务处理的基本概念。并发事务处理和故障恢复的细节问题分别在第15章和第16章讨论。事务处理的更进一步话题在第26章讨论。

14.1　事务概念

事务是访问并可能更新各种数据项的一个程序执行**单元**(unit)。事务通常由高级数据操纵语言(代表性的是SQL)或编程语言(例如，C++或Java)通过JDBC或ODBC嵌入式数据库访问书写的用户程序的执行所引起。事务用形如**begin transaction**和**end transaction**语句(或函数调用)来界定。事务由**begin transaction**与(**end transaction**之间执行的全体操作组成。

这些步骤集合必须作为一个单一的、不可分割的单元出现。因为事务是不可分割的，所以要么执行其全部内容，要么就根本不执行。因此，如果一个事务开始执行，但是由于某些原因失败，则
627 事务对数据库造成的任何可能的修改都要撤销。无论事务本身是否失败(例如，如果它除以零)，或者操作系统崩溃，或者计算机本身停止运行，这项要求都要成立。正如我们将会看到的，确保这个要求是困难的，因为对数据库的一些修改可能仅仅存在事务的主存变量中，而另一些已经写入数据库并存储到磁盘上。这种"全或无"的特性称为**原子性**(atomicity)。

此外，由于事务是一个单一的单元，它的操作不能看起来是被其他不属于该事务的数据库操作分隔开的。尽管我们希望表现在事务的用户级上，但是我们知道事实是有相当大的区别的。即使单条SQL语句也会涉及许多分开的数据库访问，并且一个事务可能会由多条SQL语句构成。因此，数据库系统必须采取特殊处理来确保事务正常执行而不被来自并发执行的数据库语句所干扰。这种特性称为**隔离性**(isolation)。

即使系统能保证一个事务的正确执行，如果此后系统崩溃，结果系统"忘记"了该事务，那么这项工作的意义也不大了。因此，即使崩溃后事务的操作也必须是持久的。这种特性称为**持久性**(durability)。

因为上述三个特性，事务就成了构造与数据库的交互的一种理想方式。这使我们必须加强对事务本身的要求。事务必须保持数据库的一致性——如果一个事务作为原子从一个一致的数据库状态开始独立地运行，则事务结束时数据库也必须再次是一致的。这种一致性要求超出我们此前看到的数据完整性约束(例如主码约束、参照完整性、**check**约束等)。相反，事务会被期望得更多，以保证太过复杂而不能用SQL构建数据完整性并且依赖于程序的一致性约束。如何实现这个则是编写事

务的程序员的职责。这种特性称为**一致性**(consistency)。

将上述内容更简明地重新描述，我们要求数据库系统维护事务的以下性质。

- **原子性**：事务的所有操作在数据库中要么全部正确反映出来，要么完全不反映。
- **一致性**：隔离执行事务时(换言之，在没有其他事务并发执行的情况下)保持数据库的一致性。
- **隔离性**：尽管多个事务可能并发执行，但系统保证，对于任何一对事务 T_i 和 T_j，在 T_i 看来，T_j 或者在 T_i 开始之前已经完成执行，或者在 T_i 完成之后开始执行。因此，每个事务都感觉不到系统中有其他事务在并发地执行。
- **持久性**：一个事务成功完成后，它对数据库的改变必须是永久的，即使出现系统故障。 628

这些性质通常称为 ACID 特性(ACID property)，这一缩写来自 4 条性质的第一个英文字母。

正如我们此后看到的，确保隔离性有可能对系统性能造成较大的不利影响。由于这个原因，一些应用在隔离性上会采取一些妥协。我们将在学习严格执行 ACID 特性后学习这些妥协。

14.2 一个简单的事务模型

因为 SQL 是一种很大而复杂的语言，所以我们采用一种简单的数据库语言来开始学习事务，该语言关注数据何时从磁盘移动到主存以及何时从主存移动到磁盘。这样，我们忽略了 SQL 插入和删除操作，推迟到 15.8 节再去考虑它们。在简单语言中，对数据的实际操作仅限于算术操作。后面，我们会在一个有更丰富的操作集合并且基于 SQL 的真实环境中讨论事务。在简单模型中数据项只包含一个单一的数据值(在例子中是一个数字)。每个数据项由一个名字所标识(在例子中通常是一个字母，例如 A、B、C 等)。

我们将采用一个由几个账户和一个访问和更新账户的事务集合构成的简单的银行应用来阐明事务的概念。事务访问以下两个操作访问数据。

- **read**(X)：从数据库把数据项 X 传送到执行 **read** 操作的事务的主存缓冲区的一个名称为 X 的变量中。
- **write**(X)：从执行 **write** 的事务的主存缓冲区的变量 X 中把数据项 X 传回数据库中。

重要的是要知道一个数据项的变化是只出现在主存中，还是已经写入磁盘上的数据库。在实际数据库系统中，**write** 操作不一定立即更新磁盘上的数据；**write** 操作的结果可以临时存储在某处，以后再写到磁盘上。但是目前我们假设 **write** 操作立即更新数据库。我们将在第 16 章回到这个话题。

设 T_i 是从账户 A 过户 \$50 到账户 B 的事务。这个事务可以定义为：

T_i: **read**(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B). 629

现在让我们逐个考虑 ACID 特性(为了便于讲解，我们不按 A－C－I－D 的次序来讲述它们)。

- **一致性**：在这里，一致性要求事务的执行不改变 A、B 之和。如果没有一致性要求，金额可能会被事务凭空创造或销毁！容易验证，如果数据库在事务执行前是一致的，那么事务执行后数据库仍将保持一致。

  确保单个事务的一致性是编写该事务的应用程序员的责任。完整性约束的自动检查给这项工作带来了便利，正如我们已经在 4.4 节讨论的。
- **原子性**：假设事务 T_i 执行前账户 A 和账户 B 分别有 \$1000 和 \$2000。现在假设在事务 T_i 执行时系统出现故障，导致 T_i 的执行没有成功完成。我们进一步假设故障发生在 **write**(A)操作执行之后 **write**(B)操作执行之前。在这种情况下，数据库中反映出来的是账户 A 有 \$950，而账户 B 有 \$2000。这次故障导致系统丢失了 \$50。特别地，我们注意到 $A + B$ 的和不再维持原状。

这样，由于故障，系统的状态不再反映数据库本应描述的现实世界的真实状态。我们把这种状态称为**不一致状态**(inconsistent state)。我们必须保证这种不一致性在数据库系统中是不可见的。但是请注意，系统必然会在某一时刻处于不一致状态。即使事务 T 能执行完，也仍然存在某一时刻账户 A 的金额是 \$950 而账户 B 的金额是 \$2000，这显然是一个不一致状态。然而这一状态最终会被账户 A 的金额是 \$950 且账户 B 的金额是 \$2050 这个一致的状态代替。这样，如果一个事务或者不开始，或者保证完成，那么这样的不一致状态除了在事务执行当中以外，在其他时刻是不可见的。这就是需要原子性的原因：如果具有原子性，某个事务的所有动作要么在数据库中全部反映出来，要么全部不反映。

保证原子性的基本思路如下：对于事务要执行写操作的数据项，数据库系统在磁盘上记录其旧值。这个信息记录在一个称为**日志**的文件中。如果事务没能完成它的执行，数据库系统从日志中恢复旧值，使得看上去事务从未执行过。14.4 节将进一步讨论这些想法。保证原子性是数据库系统本身的责任；具体来说，这项工作由称作**恢复系统**(recovery system)的一个数据库组件处理，这个将在第 16 章详细讲述。

630

- **持久性**：一旦事务成功地完成执行，并且发起事务的用户已经被告知资金转账已经发生，系统就必须保证任何系统故障都不会引起与这次转账相关的数据丢失。持久性保证一旦事务成功完成，该事务对数据库所做的所有更新就都是持久的，即使事务执行完成后出现系统故障。

  现在我们假设计算机系统的故障将会导致内存中的数据丢失，但已写入磁盘的数据决不会丢失。第 16 章将会讨论预防磁盘上的数据丢失。我们可以通过确保以下两条中的任何一条来达到持久性：

  1. 事务做的更新在事务结束前已经写入磁盘。
  2. 有关事务已执行的更新信息已写到磁盘上，并且此类信息必须充分，能让数据库在系统出现故障后重新启动时重新构造更新。

  第 16 章将介绍的数据库恢复系统负责除了保证原子性之外还保证持久性。

- **隔离性**：如果几个事务并发地执行，即使每个事务都能确保一致性和原子性，它们的操作会以人们所不希望的某种方式交叉执行，这也会导致不一致的状态。

  正如我们先前看到的，例如，在 A 至 B 转账事务执行过程中，当 A 中总金额已减去转账额并已写回 A，而 B 中总金额加上转账额后还未写回 B 时，数据库暂时是不一致的。如果另一个并发运行的事务在这个中间时刻读取 A 和 B 的值并计算 $A+B$，它将会得到不一致的值。更进一步，如果第二个事务基于它读取的不一致值对 A 和 B 进行更新，即使两个事务都完成后，数据库仍可能处于不一致状态。

  一种避免事务并发执行而产生问题的途径是串行地执行事务——一个接一个地执行。然而，我们将会在 14.5 节看到，事务并发执行能显著地改善性能。因此人们提出了许多其他的解决方法，它们允许多个事务并发地执行。

  14.5 节讨论事务并发地执行所引起的问题。事务的隔离性确保事务并发执行后的系统状态与这些事务以某种次序一个接一个地执行后的状态是等价的。14.6 节将进一步讨论隔离的原则。确保隔离性是数据库系统中称作**并发控制系统**(concurrency-control system)的部件的责任，这个稍后将在第 15 章讨论。

631

14.3 存储结构

为了理解如何确保事务的原子性和持久性，我们需要更好地理解数据库中各种数据项如何存储和访问。

在第 10 章中我们看到存储介质可以通过它们的相对速度、容量、故障弹性区分为易失性存储器或非易失性存储器。我们回顾这些概念，并介绍另一种新的存储器，即**稳定性存储器**。

- **易失性存储器**(volatile storage)：易失性存储器中的信息通常在系统崩溃后不会幸存。这种存储器的例子包括主存储器和高速缓冲存储器。易失性存储器的访问非常快，一方面是因为内存访问本身的速度，另一方面是因为可以直接访问易失性存储器中的任何数据项。
- **非易失性存储器**(nonvolatile storage)：非易失性存储器中的信息会在系统崩溃后幸存。非易失性存储器的例子包括用于在线存储的二级存储设备(如磁盘和闪存)，以及用于存档存储的三级存储设备(如光介质和磁带)。根据目前的技术，非易失性存储器比易失性存储器慢，特别是对于随机访问。然而，二级存储设备和三级存储设备容易受到故障的影响，导致信息丢失。
- **稳定性存储器**(stable storage)：稳定性存储器中的信息永远不会丢失。(应该对永远持有怀疑态度，因为理论上永远不能保证。例如，尽管可能性很小，也有可能出现黑洞吞噬地球从而永久地销毁所有数据！)尽管稳定性存储器在理论上不可能获得，但是可以通过技术近似使得数据丢失的可能性很小以至于其微。为了实现稳定性存储器，我们可以复制几种非易失性存储器介质(通常是磁盘)中的信息，并且采用独立故障模式。更新必须很小心以保证更新过程中稳定性存储器的故障不会导致信息丢失。16.2.1节将讨论稳定性存储器的实现。

这几种不同存储器类型之间的区别在实际中没有我们介绍得这么明显。例如，某些系统提供备用电池使得一些主存可以在系统崩溃或电源故障中幸存下来。闪如某些RAID控制器。

为了一个事务能够持久，它的修改应该写入稳定性存储器。同样，为了一个事务是原子的，日志记录需要在对磁盘上的数据库做任何改变之前写入稳定性存储器。显然，一个系统保证的持久性和原子性的程度取决于稳定性存储器的实现到底有多稳定。在某些情况下，磁盘的一个单一拷贝是足够的，但是对于其数据非常有价值和事务非常重要的应用程序需要多个拷贝，或者换句话说，更接近于理想 632
化的稳定性存储器。

14.4 事务原子性和持久性

正如我们先前所注意到的，事务并非总能成功地执行完成。这种事务称为**中止**(aborted)了。我们如果要确保原子性，中止事务必须对数据库的状态不造成影响。因此，中止事务对数据库所做过的任何改变必须撤销。一旦中止事务造成的变更被撤销，我们就说事务已**回滚**(rolled back)。恢复机制负责管理事务中止。典型的方法是维护一个**日志**(log)。每个事务对数据库的修改都首先会记录到日志中。我们记录执行修改的事务标识符、修改的数据项标识符以及数据项的旧值(修改前的)和新值(修改后的)。然后数据库才会修改。维护日志提供了重做修改以保证原子性和持久性的可能，以及撤销修改以保证在事务执行发生故障时的原子性的可能。第16章将会详细讨论基于日志的故障恢复。

成功完成执行的事务称为**已提交**(committed)。一个对数据库进行过更新的已提交事务使数据库进入一个新的一致状态，即使出现系统故障，这个状态也必须保持。

一旦事务已提交，我们不能通过中止它来撤销其造成的影响。撤销已提交事务所造成影响的唯一方法是执行一个**补偿事务**(compensating transaction)。例如，如果一个事务给一个账户加上了$20，其补偿事务应当从该账户减去$20。然而，我们不总是能够创建这样的补偿事务。因此，书写和执行一个补偿事务的责任就留给了用户，而不是通过数据库系统来处理。第26章包含对补偿事务的讨论。

我们需要更准确地定义一个事务成功完成的含义。为此我们建立了一个简单的抽象事务模型。事务必须处于以下状态之一。

- **活动的**(active)：初始状态，事务执行时处于这个状态。
- **部分提交的**(partially committed)：最后一条语句执行后。
- **失败的**(failed)：发现正常的执行不能继续后。
- **中止的**(aborted)：事务回滚并且数据库已恢复到事务开始执行前的状态后。
- **提交的**(committed)：成功完成后。

事务相应的状态图如图14-1所示。只有在事务已进入提交状态后，我们才说事务已提交。类似 633

地。仅当事务已进入中止状态，我们才说事务已中止。如果事务是提交的或中止的，它称为已经结束的(terminated)。

事务从活动状态开始。当事务完成它的最后一条语句后就进入了部分提交状态。此刻，事务已经完成执行，但由于实际输出可能仍临时驻留在主存中，因此一个硬件故障可能阻止其成功完成。于是事务仍有可能不得不中止。

接着数据库系统往磁盘上写入足够的信息，确保即使出现故障时事务所做的更新也能在系统重启后重新创建。当最后一条这样的信息写完后，事务就进入提交状态。

正如先前所提到的，我们现在假设故障不会引起磁盘上的数据丢失。磁盘上数据丢失的处理技术将在第16章讨论。

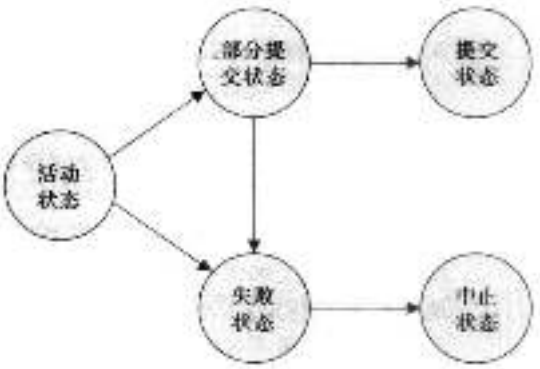


图 14-1 事务状态图

系统判定事务不能继续正常执行后(例如，由于硬件或逻辑错误)，事务就进入失败状态。这种事务必须回滚。这样，事务就进入中止状态。此刻，系统有两种选择。

- 它可以重启(restart)事务，但仅当引起事务中止的是硬件错误或不是由事务的内部逻辑所产生的软件错误时。重启的事务被看成是一个新事务。
- 它可以杀死(kill)事务，这样做通常是由于事务的内部逻辑造成的错误，只有重写应用程序才能改正，或者由于键入错误，或所需数据在数据库中没有找到。

在处理可见的外部写(observable external write)，比如写到用户屏幕，或者发送电子邮件时，我们
634 必须要小心。由于写的结果可能已经在数据库系统之外看到，因此一旦发生这种写操作，就不能再抹去。大多数系统只允许这种写操作在事务进入提交状态后发生。实现这种模式的一种方法是在非易失性存储设备中临时写下与外部写相关的所有数据，然后在事务进入提交状态后再执行真正的写操作。如果在事务已进入提交状态而外部写操作尚未完成之时，系统出现了故障，数据库系统就可以在重启后(用存储在非易失性设备中的数据)执行外部写操作。

在某些情况下处理外部写操作会更复杂，例如，我们假设外部动作是在自动取款机上支付现金，并且系统恰好在支付现金之前发生故障(我们假定现金能自动支付)，当系统重新启动时再执行现金支付将毫无意义，因为用户可能已经离开。在这种情况下，重新启动时系统应该执行一个补偿事务，比如将现金存回用户的账户。

作为另一个例子，考虑一个用户在Web上进行预订。很可能在预订事务刚刚提交后数据库系统或应用服务器发生崩溃，也有可能在预订事务刚刚提交后用户的网络连接丢失。在上述任意一种情况下，即使事务已经提交，外部写也并没有发生。为了处理这种情况，应用程序必须设计成当用户再次连接到Web应用程序时，她可以看到她的事务是否成功。

对于某些特定应用，允许处于活动状态的事务向用户显示数据也许是我们所期望的，特别对将运行几分钟或几小时的长周期事务来说。遗憾的是，除非牺牲事务原子性，否则我们不能允许这种可见的数据输出。第26章会讨论另外的事务模型，它们支持交互式长周期事务。

14.5 事务隔离性

事务处理系统通常允许多个事务并发地执行。正如我们先前看到的，允许多个事务并发更新数据引起许多数据一致性的复杂问题。在存在事务并发执行的情况下保证一致性需进行额外工作；如果我们强制事务串行地(serially)执行将简单得多——一次执行一个事务，每个事务仅当前一事务执行完后才开始。然而，有两条很好的理由允许并发。

- **提高吞吐量和资源利用率**。一个事务由多个步骤组成。一些涉及I/O活动；还有一些涉及CPU
635 活动。在计算机系统中CPU与磁盘可以并行运作。因此，I/O活动可以与CPU处理并行进行。

利用 CPU 与 I/O 系统的并行性，多个事务可并行执行。当一个事务在一张磁盘上进行读写时，另一个事务可在 CPU 上运行，第三个事务又可在另一张磁盘上进行读写。所有这些技术增加了系统的**吞吐量**(throughput)——即给定时间内执行的事务数增加。相应地，处理器与磁盘利用率(utilization)也提高；换句话说，处理器与磁盘空闲或者没有做有用的工作的时间减少。

- **减少等待时间**。系统中可能运行着各种各样的事务，一些较短，一些较长。如果事务串行地执行，短事务可能得等待它前面的长事务完成，这可能导致难以预测的延迟。如果各事务计对数据库的不同部分进行操作，让它们并发地执行会更好，它们之间可以共享 CPU 周期与磁盘存取。并发执行可以减少执行事务时不可预测的延迟。此外，也可减少平均**响应时间**(average response time)：即一个事务从提交到完成所需的平均时间。

在数据库中使用并发执行的动机在本质上与操作系统中使用多道程序(multiprogramming)的动机是一样的。

当多个事务并发地执行时，可能违背隔离性，这导致即使每个事务都正确执行，数据库的一致性也可能被破坏。这一节将讲述调度的概念，以帮助读者识别哪些是可以保证一致性的执行序列。

数据库系统必须控制事务之间的交互，以防止它们破坏数据库的一致性。系统通过称为**并发控制机制**(concurrency-control scheme)的一系列机制来保证这一点。第 15 章将研究并发控制机制，现在我们集中考虑正确的并发执行这一概念。

我们再来看看 14.1 节中的简化银行系统，其中有多个账户以及存取、更新这些账户的一组事务。设 T_1、T_2 是将资金从一个账户转移到另一个账户的两个事务，事务 T_1 是从账户 A 过户 \$50 到账户 B 的事务，它定义为：

T_1: **read**(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

636

事务 T_2 是从账户 A 将存款余额的 10% 过户到账户 B 的事务，它定义为：

T_2: **read**(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

并发性趋势

当前计算领域的一些发展趋势带来大量可能的并发性。数据库系统利用并发性提高系统的整体性能，使得并发运行的事务数量可能随之增多。

早期的计算机只有一个处理器，因此，计算机中没有真正意义的并发性。唯一表现出的并发性是操作系统使几个不同的任务或进程共享处理器。现代计算机可能有很多个处理器，这将使得一个计算机中有真正不同的进程。然而，即使是一个处理器，如果它有多核，也能够同时运行多个进程。英特尔酷睿双核处理器就是一个众所周知的多核处理器例子。

数据库系统有两种方法利用多处理器和多核的优势：一种是发现单个事务或查询内的并行性，另一种是支持大量并发的事务。

许多服务提供商现在采用一批计算机而不是大型主机来提供服务。他们做出这样的选择是基于这种方案的低成本。这样的结果是带来了更大限度的并发性支持。

文献注释介绍了计算机架构和并行计算的进展。第 18 章将介绍利用多处理器和多核构建并行数据库系统的方法。

假设账户 A 和账户 B 当前的值分别是 $1000 和 $2000。假设两个事务一个一个地执行，先是 T_1，然后是 T_2。该执行顺序如图 14-2 所示。在图 14-2 中，指令序列自顶向下或时间顺序排列，T_1 的指令出现在左栏，T_2 的指令出现在右栏。按图 14-2 的顺序执行后，账户 A 与 B 中最终的值分别为 $855 与 $2145。因此，账户 A 与 B 的资金总数（即 $A+B$）在两个事务执行后保持不变。

637

类似地，如果事务一个一个地执行，先是 T_2，然后是 T_1，那么相应的执行顺序如图 14-3 所示。同样，正如所预期的，$A+B$ 之和仍然保持不变。账户 A 与 B 中最终的值分别为 $850 与 $2150。

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B - 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

图 14-2 调度 1：一个串行调度。T_2 跟在 T_1 之后

前面所描述的执行顺序称为调度（schedule）。它们表示指令在系统中执行的时间顺序。显然，一组事务的一个调度必须包含这一组事务的全部指令，并且必须保持指令在各个事务中出现的顺序。例如，在任何一个有效的调度中，事务 T_1 中指令 **write**(A) 必须在指令 **read**(B) 之前出现。请注意，我们在调度中包括了 **commit** 操作来表示事务已经进入提交状态。在下面的讨论中，我们将称第一种执行顺序为调度 1（T_2 跟在 T_1 之后），称第二种执行顺序为调度 2（T_1 跟在 T_2 之后）。

这两个调度是**串行的**（serial）。每个串行调度由来自各事务的指令序列组成，其中属于同一事务的指令在调度中紧接在一起。回顾组合数学中一个众所周知的公式，我们知道，对于有 n 个事务的事务组，共有 $n!$ 个不同的有效串行调度。

当数据库系统并发地执行多个事务时，相应的调度不必是串行的。若有两个并发执行的事务，操作系统可能先选其中的一个事务执行一小段时间，然后切换上下文，执行第二个事务一段时间，接着又切换回第一个事务执行一段时间，如此下去。在多个事务的情形下，所有事务共享 CPU 时间。

多种执行顺序是有可能的，因为来自两个事务的各条指令可能是交叉执行的。一般而言，在 CPU 切换到另一事务之前准确预测 CPU 将执行某个事务的多少条指令是不可能的。[⊖]

回到前面的例子，假设两个事务并发执行。一种可能的调度如图 14-4 所示，当它执行完成后，我们到达的状态与先执行 T_1 后执行 T_2 的串行调度一样，$A+B$ 之和保持不变。

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	

图 14-3 调度 2：一个串行调度，T_1 跟在 T_2 之后

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

图 14-4 调度 3：等价于调度 1 的一个并发调度

不是所有的并发执行都能得到正确的结果。举个例子，考虑如图 14-5 所示的调度，该调度执行后，到达的状态是账户 A 与 B 中最终的值分别为 $950 与 $2100。这个最终状态是一个不一致状态，因

⊖ 包含 n 个事务的事务组的可能调度数量非常大，有 $n!$ 个不同的串行调度。考虑所有的事务的步骤可以交错的方式，调度的总数要比 $n!$ 大得多。

为在并发执行过程中多出了850。实际上，两个事务执行后 $A+B$ 之和未能保持不变。

如果并发执行的控制完全由操作系统负责，许多调度都是可能的，包括像上述调度那样使数据库处于不一致状态的调度。保证所执行的任何调度都能使数据库处于一致状态，这是数据库系统的任务，数据库系统中负责完成此任务的是并发控制(concurrency-control)部件。

在并发执行中，通过保证所执行的任何调度的效果都与没有并发执行的调度效果一样，我们可以确保数据库的一致性。也就是说，调度应该在某种意义上等价于一个串行调度。这种调度称为**可串行化**(serializable)调度。

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

图14-5 调度4：一个导致不一致状态的并发调度

14.6 可串行化

在我们考虑数据库系统并发控制部件如何保证串行化之前，我们考虑如何确定一个调度是可串行化的。显然，串行调度是可串行化的，但是如果许多事务的步骤交错执行，则很难确定一个调度是否是可串行化的。由于事务就是程序，因此要确定一个事务有哪些操作、多个事务的操作如何相互作用是有困难的。由于这个缘故，我们将不会考虑一个事务对某一数据项可执行的各种不同类型的操作，而只考虑两种操作：**read** 和 **write**。我们这样假设，在数据项 Q 上的 **read**(Q) 和 **write**(Q)指令之间，事务可以对驻留在事务局部缓冲区中的 Q 的拷贝执行任意操作序列。按这种模式，从调度的角度来看，事务唯一重要的操作是 **read** 与 **write** 指令。commit 操作尽管相关，但是我们到14.7节才考虑它。因此，我们在调度中通常只显示 **read** 与 **write** 指令，正如图14-6所示调度3中所表示的那样。

本节讨论不同形式的等价调度，但是重点关注其中一种称为**冲突可串行化**(conflict serializability)的形式。

我们考虑一个调度 S，其中含有分别属于 I 与 J 的两条连续指令 I_i 与 I_j ($i \neq j$)。如果 I 与 J 引用不同的数据项，则交换 I 与 J 不会影响调度中任何指令的结果。然而，若 I 与 J 引用相同的数据项 Q，则两者的顺序是重要的。由于我们只处理 **read** 与 **write** 指令，因此需考虑以下4种情形：

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

图14-6 调度3：只显示 read 与 write 指令

1. I = **read**(Q)，J = **read**(Q)。I 与 J 的次序无关紧要，因为不论其次序如何，T_i 与 T_j 读取的 Q 值总是相同的。

2. I = **read**(Q)，J = **write**(Q)。若 I 先于 J，则 T_i 不会读取由 T_j 的指令 J 写入的 Q 值；若 J 先于 I，则 T_i 读到由 T_j 写入的 Q 值。因此 I 与 J 的次序是重要的。

638
641

3. I = **write**(Q)，J = **read**(Q)。I 与 J 的次序是重要的，其原因类似前一情形。

4. I = **write**(Q)，J = **write**(Q)。由于两条指令均为 **write** 指令，因此指令的顺序对 T_i 与 T_j 没有什么影响。然而，调度 S 的下一条 **read**(Q)指令读取的值将受到影响，因为数据库里只保留了两条 **write** 指令中后一条的结果。如果在调度 S 的指令 I 与 J 之后没有其他的 write(Q)指令，则 I 与 J 的顺序直接影响调度 S 产生的数据库状态中 Q 的最终值。

因此，只有在 I 与 J 个为 **read** 指令时，两条指令的执行顺序才是无关紧要的。

当 I 与 J 是不同事务在相同的数据项上的操作，并且其中至少有一个是 **write** 指令时，我们说 I 与 J 是冲突(conflict)的。

为了说明冲突指令的概念，我们考虑图14-6中的调度3。T_1 的 **write**(A)指令与 T_2 的 **read**(A)指令相冲突。然而，T_2 的 **write**(A)指令与 T_1 的 **read**(B)指令不冲突，因为两条指令访问不同的数据项。

设 I 与 J 是调度 S 的两条连续指令。若 I 与 J 是属于不同事务的指令且不冲突，则可以交换 I 与 J 的顺序得到一个新的调度 S'。S 与 S' 等价，因为除了 I 与 J 外，其他指令的次序与原来相同，而 I 与 J 的顺序无关紧要。

在图 14-6 的调度 3 中，由于 T_3 的 **write**(A)指令与 T_1 的 **read**(B)指令不冲突，因此可以交换这些指令得到一个等价的调度——图 14-7 所示的调度 5。不管系统初始状态如何，调度 3 与调度 5 均得到相同的最终系统状态。

我们继续交换非冲突指令如下：

- 交换 T_1 的 **read**(B)指令与 T_2 的 **read**(A)指令。
- 交换 T_1 的 **write**(B)指令与 T_2 的 **write**(A)指令。
- 交换 T_1 的 **write**(B)指令与 T_2 的 **read**(A)指令。

经过以上交换的结果是一个串行调度。即图 14-8 所示的调度 6。注意调度 6 和调度 1 完全一样，但是后者只显示了 **read** 和 **write** 指令。这样，我们说明了调度 3 等价于一个串行调度。该等价性意味着不管初始系统状态如何，调度 3 将与某个串行调度产生相同的最终状态。

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

图 14-7 调度 5：交换调度 3 的一对指令得到的调度

如果调度 S 可以经过一系列非冲突指令交换转换成 S'，我们称 S 与 S' 是**冲突等价**(conflict equivalent)的。[⊖]

不是所有的串行调度相互之间都冲突等价。例如，调度 1 和调度 2 不是冲突等价的。

由冲突等价的概念引出了冲突可串行化的概念：若一个调度 S 与一个串行调度冲突等价，则称调度 S 是冲突可串行化(conflict serializable)的。那么，因为调度 3 冲突等价于串行调度 1，所以调度 3 是冲突可串行化的。

T_3	T_4
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

图 14-8 调度 6：与调度 3 等价的一个串行调度

最后，考虑图 14-9 所示的调度 7；该调度仅包含事务 T_3 与 T_4 的重要操作(即 read 与 write)。这个调度不是冲突可串行化的，因为它既不等价于串行调度 $< T_3, T_4 >$，也不等价于串行调度 $< T_4, T_3 >$。

为确定一个调度是否冲突可串行化，我们这里给出了一个简单有效的方法。设 S 是一个调度，我们由 S 构造一个有向图，称为**优先图**(precedence graph)。该图由两部分组成 $G = (V, E)$，其中 V 是顶点集，E 是边集，顶点集由所有参与调度的事务组成，边集由满足下列三个条件之一的边 $T_i \rightarrow T_j$ 组成：

- 在 T_i 执行 **read**(Q)之前，T_j 执行 **write**(Q)。
- 在 T_i 执行 **write**(Q)之前，T_j 执行 **read**(Q)。
- 在 T_i 执行 **write**(Q)之前，T_j 执行 **write**(Q)。

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

图 14-9 调度 7

如果优先图中存在边 $T_i \rightarrow T_j$，则在任何等价于 S 的串行调度 S' 中，T_i 必出现在 T_j 之前。



a) b)

图 14-10 a)调度 1 与 b)调度 2 的优先图

例如，调度 1 的优先图如图 14-10a 所示，图 14-10a 中只有一条边 $T_1 \rightarrow T_2$，因为 T_1 的所有指令均在 T_2 的首条指令之前执行。类似地，图 14-10b 表示的是调度 2 的优先图，该图仅含一条边 $T_2 \rightarrow T_1$，因为 T_2 的所有指令均在 T_1 的首条指令之前执行。

调度 4 的优先图如图 14-11 所示。因为 T_3 执行 **read**(A)先于 T_4 执行 **write**(A)，所以图 14-11 含有边 $T_3 \rightarrow T_4$。又因 T_3 执行 **read**(B)先于 T_4 执行 **write**(B)，所以图 14-11 还含有边 $T_3 \rightarrow T_4$。

如果调度 S 的优先图有环，则调度 S 是非冲突可串行化的，如果优先图无环，则调度 S 是冲突可串行化的。

⊖ 们用冲突等价这个术语把我们定义的等价和本节后面将介绍的其他定义区别开。

串行化顺序(serializability order)可通过**拓扑排序**(topological sorting)得到，拓扑排序用于计算与优先图的输序相一致的线性顺序。一般而言，通过拓扑排序可以获得多个线性顺序。例如，图14-12a有两种可接受的线性顺序，如图14-12b与图14-12c所示。

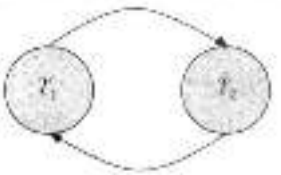


图14-11 调度4的优先图

因此，要判定冲突可串行化，需要构造优先图并调用一个环检测算法。环检测算法可在标准的算法课本中找到。环检测算法，例如那些基于深度优先搜索的环检测算法，需要 n^2 数量级的运算，其中 n 是优先图中顶点数(即事务数)。

回到前面的例子，注意到调度1与调度2的优先图(见图14-10)的确不包含环。而调度4的优先图(见图14-11)却有一个环，说明该调度不是冲突可串行化的。

有可能存在两个调度，它们产生相同的结果，但它们不是冲突等价的。例如，考虑事务 T_1，它从账户 B 过户$10到账户 A。设调度8如图14-13所示。我们说调度8不与串行调度 $<T_1, T_2>$ 冲突等价，因为在调度8中，T_1 的 **write**(B)指令与 T_2 的 **read**(B)指令冲突。这在优先图中产生了一条 $T_1 \rightarrow T_2$ 的边。类似地，我们看到 T_1 的 **write**(A)指令与 T_2 的 **read** 指令冲突，产生了一条 $T_1 \rightarrow T_2$ 的边。这表示优先图有环，即调度8不是可串行化的。然而，执行调度8或串行调度 $<T_1, T_2>$ 后，账户 A 与 B 中最终的值是相同的，即分别为$960与$2040。

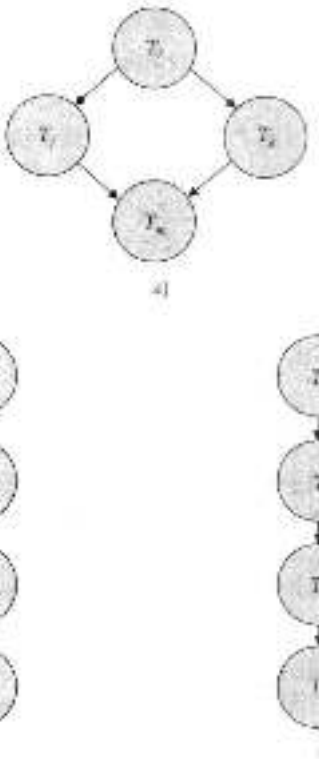


图14-12 拓扑排序示例

从这个例子可以看出，存在比冲突等价定义限制松一些的调度等价定义。对于系统来说，要确定调度8与串行调度 $<T_1, T_2>$ 产生的结果相同，系统必须分析 T_1 与 T_2 所进行的计算，而不只是分析其 **read** 和 **write** 操作。通常，这种分析难于实现并且计算代价很大。在该例子中，最后的结果和串行调度是一样的，因为从数学的角度递增和递减是可交换的。虽然这在该例子中比较简单，但是一般情况下并非如此，因为一个事务可能会表示为一条复杂的SQL语句，或一个有JDBC调用的Java程序等。

不过，存在一些别的纯粹基于 **read** 与 **write** 操作的调度等价定义。其中一个是视图等价，并且引出视图可串行化的概念。视图可串行化因为其计算的高度复杂性在实际中并不常用。[⊖]因此，我们推迟到第15章中再讨论视图可串行化，但是为了表述完整起见，这儿请注意，调度8的例子不是视图可串行化的。

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

图14-13 调度8

⊖ 判定调度是否视图可串行化的问题已被证明是属于NP-完全问题，因此几乎肯定不存在有效的判定视图可串行化的算法。

14.7 事务隔离性和原子性

642
646

至此，我们在隐含地假定无事务故障的前提下学习了调度。现在我们讨论在并发执行过程中事务故障所产生的影响。

不论什么原因，如果事务 T_i 失败了，我们必须撤销该事务的影响以确保其原子性。在允许并发执行的系统中，原子性要求依赖于 T_i 的任何事务 T_j（即 T_j 读取了 T_i 写的数据）也中止。为确保这一点，我们需要对系统所允许的调度类型做一些限制。

在下面两节中，我们从事务故障恢复的角度讲述什么样的调度是可接受的，第15章将讲述如何保证只产生这种可接受的调度。

14.7.1 可恢复调度

考虑图14-14所示的部分调度9，其中事务 T_2 只执行一条指令：**read**(A)。我们称之为**部分调度**（partial schedule），因为在 T_2 中没有包括 **commit** 或 **abort** 操作。注意 T_2 执行 **read**(A) 指令后立即提交。因此 T_2 提交时 T_1 仍处于活跃状态。现假定 T_1 在提交前发生故障。T_2 已读取了由 T_1 写入的数据项 A 的值。因此，我们说 T_2 **依赖**（dependent）于 T_1。因此，我们必须中止 T_2 以保证事务的原子性。但 T_2 已提交，不能再中止。这样就出现了 T_1 发生故障后不能正确恢复的情形。

T_1	T_2
read(A)	
write(A)	
	read(A)
	commit
read(B)	

图14-14 调度9：一个可恢复的调度

调度9是不可恢复调度的一个例子。一个**可恢复调度**（recoverable schedule）应满足：对于每对事务 T_i 和 T_j，如果 T_j 读取了之前由 T_i 所写的数据项，则 T_i 先于 T_j 提交。例如，如果要使调度9是可恢复的，则 T_2 应该推迟到 T_1 提交后再提交。

14.7.2 无级联调度

即使一个调度是可恢复的，要从事务 T_i 的故障中正确恢复，可能需要回滚若干事务。当其他事务读取了由事务 T_i 所写数据项时就会发生这种情形。举一个例子，考虑图14-15所示的部分调度。事务 T_8 写入 A 的值，事务 T_9 读取了 A 的值。事务 T_9 写入 A 的值，事务 T_{10} 读取了 A 的值。假定此时事务 T_8 失败，T_8 必须回滚。由于 T_9 依赖于 T_8，因此事务 T_9 必须回滚。由于 T_{10} 依赖于 T_9，因此 T_{10} 必须回滚。这种因单个事务故障导致一系列事务回滚的现象称为**级联回滚**（cascading rollback）。

647

级联回滚导致撤销大量工作，是我们不希望发生的。我们希望对调度加以限制，避免级联回滚发生。这样的调度称为**无级联调度**。现先述说，**无级联调度**（cascadeless schedule）应满足：对于每对事务 T_i 和 T_j，如果 T_j 读取了先前由 T_i 所写的数据项，则 T_i 必须在 T_j 这一读操作前提交。容易验证每一个无级联调度也都是可恢复的调度。

T_8	T_9	T_{10}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
abort		

图14-15 调度10

14.8 事务隔离性级别

可串行性是一个有用的概念，因为当程序员对事务编码时，它允许程序员忽略与并发性相关的问题。如果事务在独立执行时保证数据库一致性，那么可串行性就能确保并发执行时也具有一致性。然而，对于某些应用，保证可串行性的那些协议可能只允许最小的并发度。在这种情况下，我们采用较弱级别的一致性。为了保证数据库的正确性，使用较弱级别一致性给程序员增加了额外负担。

SQL标准也允许一个事务这样规定：它可以以一种与其他事务不可串行化的方式执行。例如，一个事务可能在**未提交读**级别上操作，这里允许事务读取甚至还未提交的记录。SQL为那些不要求精确结果的长事务提供这种特征。如果这些事务要在可串行化的方式下执行，它们就会干扰其他事务，造成其他事务执行的延迟。

SQL标准规定的隔离性级别如下。

648

- **可串行化**（serializable）：通常保证可串行化调度。然而，正如我们将要解释的，一些数据库系统对该隔离性级别的实现在某些情况下允许非可串行化执行。

- **可重复读**(repeatable read)：只允许读取已提交数据，而且在一个事务两次读取一个数据项期间，其他事务不得更新该数据。但该事务不要求与其他事务可串行化。例如，当一个事务在查找满足某些条件的数据时，它可能找到一个已提交事务插入的一些数据，但可能找不到该事务插入的其他数据。
- **已提交读**(read committed)：只允许读取已提交数据，但不要求可重复读。比如，在事务两次读取一个数据项期间，另一个事务更新了该数据并提交。
- **未提交读**(read uncommitted)：允许读取未提交数据。这是SQL允许的最低一致性级别。

以上所有隔离性级别都不允许脏写(dirty write)，即如果一个数据项已经被另外一个尚未提交或中止的事务写入，则不允许对该数据项执行写操作。

许多数据库系统运行时的默认隔离性级别是已提交读。在SQL中，除了接受系统的默认设置，还可以显式地设置隔离性级别。例如，语句"**set transaction isolation level serializable**;"将隔离性级别设置为可串行化，其他隔离性级别可类似设定。Oracle、PostgreSQL和SQL Server均支持上述语法。DB2使用语法"**change isolation level**"以及其自身提供的隔离性级别缩写。

修改隔离性级别必须作为事务的第一条语句执行。此外，如果单条语句的自动提交默认打开，则必须关闭；可以采用API函数来做这件事，例如我们在5.1.1.7节中看到的JDBC方法**Connection.setAutoCommit(false)**。此外，在JDBC方法中，**Connection.setTransactionIsolation(int level)**可以用来设置隔离性级别。更多细节请参阅JDBC手册。

一个应用程序设计者可能会为了提高系统性能而接受级别弱的隔离性级别。正如我们将在14.9节和第15章所看到的，确保可串行化可能会迫使一个事务等待另一个事务，或者在某些情况下，由于该事务无法作为可串行化执行的一部分而被迫中止。虽然为了性能可能会带来短暂的数据库不一致风险，但是如果我们能确保这种不一致性是和应用程序无关的，则这种权衡是合理的。

实现隔离性级别有很多方法。只要实现确保可串行化，数据库应用程序的设计者或者应用程序的用户就不需要知道这些实现的细节，除非需要处理性能问题。遗憾的是，尽管隔离性级别设置为可串行化，但一些数据库系统实际上采用了较弱的隔离性级别来实现，这并不排除所有非可串行化的可能性。我们将在14.9节再次讨论这个问题。如果无论显式地或隐式采用较弱的隔离性级别，则应用程序的设计者必须知道一些实现细节，从而避免或者最小化由于缺乏可串行化保证所带来的不一致的可能性。 [649]

现实世界中的可串行化

可串行化调度是保证一致性的理想方法，但是在日常工作中，我们无需如此严格的要求。一个提供商品销售的网站可能会列出某个存货商品，当一个用户选择该商品并且在结账过程中，该商品是不可用的。从数据库的角度来看，这就是一个不可重复读。

作为另一个例子，考虑在航空旅行中选择座位。假设一个旅客已经预订好行程，并且正在为每次航班选择座位。许多航空公司的网站允许用户查看不同的航班并选择一个座位，然后要求用户确认该选择。而其他旅客也可能同时正在选择同样航班的座位或者更改选择的座位。因此，旅客看到的空余座位事实上是变化的，但是旅客所看到的只是当他开始座位选择流程时空余座位的一个快照。

即使两个旅客同时选择座位，他们很可能选择不同的座位，也不会发生冲突。然而，事务是非可串行化的，由于一个旅客读取的数据是此前其他旅客更新的，会导致优先图中的一个环。如果两个旅客同时选择了一个座位，其中的一个则不会获得他所选择的座位。不过，这种情况很容易解决，只要在更新的空余座位信息上，要求这个旅客再次选择即可。

通过限制一个时刻只允许一个用户选择某一个航班的座位，可保证可串行化。然而，这样做可能会带来很长的延迟，因为旅客需要等待他的航班变成可以选择座位。特别地，如果一个旅客花费很长时间来选择一个座位，则会给其他旅客带来严重问题。另外的做法是，这类事务通常可以分成一个需要用户交互的部分以及一个专门在数据库上运行的部分。在上述例子中，数据库事务将检查旅客选中的座位是否仍然空闲，如果是，则更新数据库中的座位选择信息。只有在没有用户交互的情况下，数据库上运行的事务才能保证可串行化。

14.9 隔离性级别的实现

至此，我们已经知道一个调度应具有什么样的特性才能保证数据库处于一致状态，并保证事务故障得以安全地处理。

我们可以使用多种**并发控制机制**(concurrency-control scheme)来保证，即使在有多个事务并发执行
650 时，不管操作系统在事务之间如何分时共享资源(如 CPU 时间)，都只产生可接受的调度。

举一个并发控制机制的简单例子，考虑如下机制：一个事务在开始前获得整个数据库的锁(lock)，并在它提交之后释放这个锁。当一个事务持有锁时，其他事务就不允许获得这个锁，因此必须等待锁释放。由于采用了封锁策略，因此一次只能执行一个事务。所以只会产生串行调度。这样的调度很明显是可串行化的，并且容易证明也是可恢复的和无级联的。

这样的并发控制机制的性能低下，因为它迫使事务等到前面的事务结束后才能开始。换句话说，这种机制提供的并发程度很低。正如我们在 14.5 节中看到的那样，并发执行有诸多性能方面的益处。

并发控制机制的目的是获得高度的并发性，同时保证所产生的调度是冲突可串行化或视图可串行化的、可恢复的，并且是无级联的。

下面大致介绍一些重要的并发控制机制是如何工作的，然后第 15 章会介绍更多细节。

14.9.1 锁

一个事务可以封锁其访问的数据项，而不用封锁整个数据库。在这种策略下，事务必须在足够长的时间内持有锁来保证可串行化，但是这一周期又要足够短致使不会过度影响性能。对于我们将在 14.10 节中看到的数据项的访问依赖于一条 **where** 子句的 SQL 语句则情况更为复杂。第 15 章将介绍一个简单并且广泛用来确保可串行化的两阶段封锁协议。简单地说，两阶段封锁要求一个事务有两个阶段，一个阶段只获得锁但不释放锁，第二个阶段只释放锁但是不获得锁。(实际上，通常只有当事务完成所有操作并且提交或中止时才释放锁。)

如果我们有两种锁，则封锁的结果将进一步得到改进：共享的和排他的。共享锁用于事务读的数据项，而排他锁用于事务写的数据项。许多事务可以同时持有一个数据项上的共享锁，但是只有当其他事务在一个数据项上不持有任何锁(无论共享锁或排他锁)时，一个事务才允许持有该数据项上的排他锁。这两种锁模式以及两阶段封锁协议在保证可串行化的前提下允许数据的并发读。

14.9.2 时间戳

另一类用来实现隔离性的技术为每个事务分配一个**时间戳**(timestamp)，通常是当它开始的时候。对于每个数据项，系统维护两个时间戳。数据项的读时间戳记录读该数据项的事务的最大(即最近的)
651 时间戳。数据项的写时间戳记录写入该数据项当前值的事务的时间戳。时间戳用来确保在访问冲突情况下，事务按照事务时间戳的顺序来访问数据项。当不可能访问时，违例事务将会中止，并且分配一个新的时间戳重新开始。

14.9.3 多版本和快照隔离

通过维护数据项的多个版本，一个事务允许读取一个旧版本的数据项，而不是被另一个未提交或者在串行化序列中应该排在后面的事务写入的新版本的数据项。有许多多版本并发控制技术，其中一个是实际中广泛应用的称为**快照隔离**(snapshot isolation)的技术。

在快照隔离中，我们可以想象每个事务开始时有其自身的数据库版本或者快照。[⊖]它从这个私有版本中读取数据，因此和其他事务所做的更新隔离开。如果事务更新数据库，更新只出现在其私有版本中，而不是实际的数据库本身中。当事务提交时，和更新有关的信息将保存，使得更新被写入"真正的"数据库。

当一个事务 T 进入部分提交状态后，只有在没有其他并发事务已经修改该事务想要更新的数据项的情况下，事务进入提交状态。而不能提交的事务则中止。

⊖ 当然，在现实中，不会复制整个数据库，只有改变的数据项才会保留多个版本。

快照隔离可以保证读数据的尝试永远无须等待(不像封锁的情况)。只读事务不会中止;只有修改数据的事务有很小的中止风险。由于每个事务读取它自己的数据库版本或快照,因此读数据不会导致此后其他事务的更新尝试被迫等待(不像封锁的情况)。因为大部分事务是只读的(并且大多数其他事务读数据的情况多于更新),所以这是与锁相比往往带来性能改善的主要原因。

矛盾的是,快照隔离带来的问题是它提供了太多的隔离。考虑两个事务 T 和 T'。在一个串行化调度中,要么 T 看到 T' 所做的所有更新,要么 T' 看到 T 所做的所有更新,因为在串行化顺序中一个必须在另一个之后。在快照隔离下,任何事务都不能看到对方的更新。这是在串行化调度中不会出现的。在许多(事实上,大多数)情况下,两个事务的数据访问不会冲突,因此没有什么问题。然而,如果 T 读取 T' 更新的某些数据项并且 T' 读取 T 更新的某些数据项,则可能两个事务都无法读取对方的更新。结果可能会导致我们将会在第 15 章看到的数据库不一致状态,而这个在可串行化执行中当然是不会出现的。 652

Oracle、PostgreSQL 和 SQL Server 提供快照隔离的选项。Oracle 和 PostgreSQL 使用快照隔离来实现可串行化隔离级别。因此,它们的可串行化的实现在某些特殊的情况下会导致允许一个非可串行化的调度。而 SQL Server 在标准级别以外增加了一个称为快照的隔离性级别,来提供快照隔离选项。

14.10 事务的 SQL 语句表示

4.3 节介绍了 SQL 中标识事务开始和结束的语法。现在我们已经介绍过一些保持事务 ACID 特性的问题,我们已经准备好考虑如何在用一系列 SQL 语句表示事务时保证这些特性,而不像到目前为止我们仅限制了简单的读和写的模型。

在简单模型中,我们假设存在一些数据项集合。虽然我们允许数据项的值改变,但是不允许数据项被创建或删除。然而,在 SQL 中,**insert** 语句用来创建新数据,**delete** 语句用来删除数据。事实上,这两条语句是 **write** 操作,因为它们改变了数据库,但是它们与其他事务操作的交互与我们在简单模型中看到的不同。例如,考虑如下在大学数据库上的查询语句,查找所有工资超过 $90000 的教师。

```
select ID, name
from instructor
where salary > 90000;
```

采用示例的关系 *instructor*(见附录 A.3),我们发现只有 Einstein 和 Brandt 满足条件。现在假设在执行该查询的同一时间,另外一个用户插入一条新的名为"James"并且工资为 $10 000 的教师数据。

```
insert into instructor values ('11111', 'James', 'Marketing', 100000);
```

查询结果会取决于该插入是先于还是后于执行的查询而有所不同。在这两个事务的并发执行中,从直观上看它们是冲突的,然而这种冲突在简单模型中无法发现。这种情况称为**幻象现象**(phantom phenomenon),因为冲突存在于一个"幻象"数据上。

简单事务模型要求指定一个具体的数据项作为操作的参数来执行在该数据项上的操作。在简单模型中,我们从 **read** 和 **write** 步骤中就可以看到哪个数据项被使用。但是在一条 SQL 语句中,具体的数据项(元组)可能被一条 **where** 语句中的谓词所决定。因此,如果在事务多次运行之间数据库发生改变,那么即使是同一个事务,在多次不同运行中也可能会使用不同的数据项。 653

解决上述问题的一种方法是要从识别在并发控制时仅考虑事务访问的元组是不够的;并发控制还需要考虑找到事务访问元组所需的信息。这些用于寻找元组的信息可能会被插入和删除所更新,或者在有索引的情况下,该信息还可能由于搜索键属性更新而更新。例如,如果采用封锁机制来进行并发控制,则用于追踪关系中元组的数据结构,以及索引结构都必须适当地封锁。然而,这种封锁可能会在一些情况下导致低的并发度。最大化并发性的索引封锁协议将会在 15.8.3 节介绍,它在插入、删除、带有谓词的查询中都保证了可串行化。

让我们再次考虑查询:

```
select ID, name
from instructor
where salary > 90000;
```

和以下 SQL 更新：

```
update instructor
set salary = salary * 0.9
where name = 'Wu';
```

我们在判断该查询到底是否和这条更新语句冲突时面临一个有趣的现象。如果我们的查询读取整个 *instructor* 关系，则它读取了与'Wu'相关的元组，因此和更新冲突。然而，如果存在索引，使得该查询可以直接访问 *salary* > 90 000 的元组，则该查询根本不会访问'Wu'的元组，因为在该实例关系中'Wu'的初始工资为 $90 000，且更新后减少为 $81 000。

但是，使用上述方法，看起来一个冲突是否存在依赖于底层系统的查询处理决策，而与用户级两条 SQL 语句的含义无关！另一种并发控制方法是如果一次插入、删除、更新会影响一个谓词所选择的元组，则将其看作与关系上的谓词冲突。在上述例子的查询中，谓词是"*salary* > 90 000"，则一个将'Wu'的工资从 $90 000 更新为比 $90 000 更高的更新，或者将'Einstein'的工资从高于 $90 000 更新到低于或等于 $90 000 的更新都会和谓词发生冲突。基于这种思想的封锁称为**谓词锁**(predicate locking)。
654 然而，这种封锁代价大，因此实际中很少使用。

14.11 总结

- 事务是一个程序执行单位，它访问且可能更新不同的数据项。理解事务这个概念对于理解与实现数据库中的数据更新是很关键的，只有这样才能保证并发执行与各种故障不会导致数据库处于不一致状态。
- 事务具有 ACID 特性：原子性、一致性、隔离性、持久性。
  - 原子性保证事务的所有影响在数据库中要么全部反映出来，要么根本不反映；一个故障不能让数据库处于事务部分执行后的状态。
  - 一致性保证若数据库一开始是一致的，则事务(单独)执行后数据库仍处于一致状态。
  - 隔离性保证并发执行的事务相互隔离，使得每个事务感觉不到系统中其他事务的并发执行。
  - 持久性保证一旦一个事务提交后，它对数据库的改变不会丢失，即使系统可能出现故障。
- 事务的并发执行提高了事务吞吐量和系统利用率，也减少了事务等待时间。
- 计算机中不同的存储介质包括易失性存储器、非易失性存储器和稳定性存储器。易失性存储器(例如 RAM)中的数据当计算机崩溃时丢失。非易失性存储器(如磁盘)中的数据在计算机崩溃时不会丢失，但是可能会由于磁盘损毁而丢失。稳定性存储器中的数据永远不会丢失。
- 必须支持在线访问的稳定性存储器与磁盘镜像或者其他形式的提供冗余数据存储的 RAID 接近。对于离线或归档的情况，稳定性存储器可以由存储在物理安全位置的数据的多个磁带备份所构成。
- 多个事务在数据库中并发执行时，数据的一致性可能不再维持。因此系统必须控制各并发事务之间的相互作用。
  - 由于事务是保持一致性的单元，所以事务的串行执行能保持一致性。
655 - 调度捕获影响事务并发执行的关键操作，如 **read** 和 **write** 操作，而忽略事务执行的内部细节。
  - 我们要求事务集的并发执行所产生的任何调度的执行效果等价于由这些事务按某种串行顺序执行的效果。
  - 保证这个特性的系统称为保证了可串行化。
  - 存在几种不同的等价概念，从而引出了冲突可串行化与视图可串行化的概念。
- 事务并发执行所产生的调度的可串行化可以通过多种并发控制机制中的一种来加以保证。
- 给定一个调度，我们可以通过为该调度构造优先图及搜索是否无环来判定它是否冲突可串行化。然而，有更好的并发控制机制可用来保证可串行化。
- 调度必须是可恢复的，以确保：若事务 a 看到事务 b 的影响，当 b 中止时，a 也要中止。
- 调度最好是无级联的，这样不会由于一个事务的中止引起其他事务的级联中止。无级联性是通

过只允许事务读取已提交数据来保证的。

- 数据库的并发控制管理部件负责处理并发控制机制。第15章阐述并发控制机制。

术语回顾

- 事务
- ACID 特性
  - 原子性
  - 一致性
  - 隔离性
  - 持久性
- 不一致状态
- 存储器类型
  - 易失性存储器
  - 非易失性存储器
  - 稳定性存储器
- 并发控制系统
- 故障恢复系统
- 事务状态
  - 活动的
  - 部分提交的
  - 失败的
  - 中止的
  - 提交的
  - 已结束的
- 事务
  - 重启
  - 杀死
- 可见的外部写
- 并发执行
- 串行执行
- 调度
- 操作冲突
- 冲突等价
- 冲突可串行化
- 可串行化判定
- 优先图
- 可串行化顺序
- 可恢复调度
- 级联回滚
- 无级联调度
- 并发控制机制
- 封锁
- 多版本
- 快照隔离

实践习题

14.1 假设存在永远不出现故障的数据库系统，对这样的系统还需要故障恢复管理器吗？

14.2 考虑一个文件系统，比如你最喜欢的操作系统的文件系统。

a. 创建和删除文件分别包括哪些步骤，向文件中写数据呢？

b. 试说明原子性和持久性问题与创建和删除文件以及向文件中写数据有什么关系。

14.3 数据库系统实现者比文件系统实现者更注意 ACID 特性，为什么会这样？

14.4 论证下面的说法：必须从（慢速的）磁盘获取数据或执行长事务时，事务的并发执行更为重要，而当数据存在于主存中且事务非常短时，没那么重要。

14.5 既然每一个冲突可串行化调度都是视图可串行化的，我们为什么强调冲突可串行化而非视图可串行化呢？

14.6 考虑图 14-16 所示的优先图，相应的调度是冲突可串行化的吗？解释你的回答。

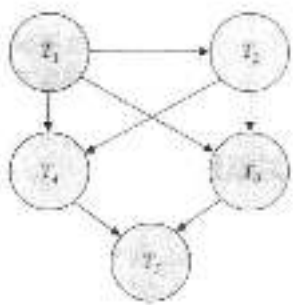


图 14-16 实践习题 14.6 的优先图

14.7 什么是**无级联调度**？为什么要求无级联调度？是否存在要求允许级联调度的情况？解释你的回答。

14.8 **丢失更新**（lost update）异常是指如果事务 T_i 读取了一个数据项，然后另一个事务 T_k 写该数据项（可能基于先前的读取），然后 T_i 写该数据项，于是 T_k 所做的更新丢失了，因为 T_i 的更新覆盖了 T_k 写入的值。

a. 给出一个表示丢失更新异常的调度实例。

b. 给出一个表示丢失更新异常的调度实例，表明在已提交读隔离性级别下该异常也可能存在。

c. 解释为什么在**可重复读**隔离性级别下丢失更新异常不可能发生。

14.9 考虑一个采用快照隔离的银行数据库系统，描述一个出现非可串行化调度会为银行带来问题的特定场景。

14.10 考虑一个采用快照隔离的航空公司数据库系统，描述一个出现非可串行化调度但是航空公司为了更好的整体性能而愿意接受的特定场景。

14.11 一个调度的定义假设操作可以完全按时间顺序排列。考虑一个运行在多处理器系统上的数据库系统，它并不总是能对于运行在不同处理器上的操作确定一个准确的顺序。但是，一个数据项上的操作全部可以排序。

以上情况是否对冲突可串行化的定义造成问题？解释你的答案。

习题

14.12 列出 ACID 特性，解释每一特性的用途。

14.13 事务从开始执行直到提交或终止，其间要经过几个状态。列出所有可能出现的事务状态序列，解释每一种状态变迁出现的原因。

14.14 解释串行调度和可串行化调度的区别。

14.15 考虑以下两个事务：

T_3: **read**(A);
read(B);
if $A = 0$ **then** $B := B + 1$;
write(B);

T_4: **read**(B);
read(A);
if $B = 0$ **then** $A := A + 1$;
write(A);

设一致性需求为 $A = 0 \vee B = 0$，初值是 $A = B = 0$。

a. 说明包括这两个事务的每一个串行执行都保持数据库的一致性。

b. 给出 T_3 和 T_4 的一次并发执行，执行产生不可串行化调度。

c. 存在产生可串行化调度的 T_3 和 T_4 的并发执行吗？

14.16 给出两个事务的一个串行化调度的例子，其中事务的提交顺序与串行化序列不同。

14.17 什么是可恢复调度？为什么要求调度的可恢复性？存在要求允许出现不可恢复调度的情况吗？解释你的回答。

14.18 为什么数据库系统支持事务的并发执行，尽管需要额外编程工作来确保并发执行不会引起任何问题？

14.19 解释为何已提交读隔离性级别保证调度是无级联的。

14.20 对于以下隔离性级别，给出一个满足该隔离性级别但不是可串行化的调度实例：

656 | 659

a. 未提交读

b. 已提交读

c. 可重复读

14.21 假设除了 **read** 和 **write** 操作，我们允许一个操作 **pred_read**(r, P) 读取关系 r 中所有满足谓词 P 的元组。

a. 给出一个使用 **pred_read** 的调度实例来展示一个有幻象现象的非可串行化调度。

b. 给出一个调度实例，其中一个事务在关系 r 上用 **pred_read**，另一个事务删除 r 中的一个元组，但是调度中没有幻象冲突。（为此，你需要给出关系 r 的表结构，并且显示被删除元组的值）。

文献注解

Gray 和 Reuter[1993] 在其教科书中全面讨论了事务处理的概念、技术和实现，包括并发控制和恢复问题。Bernstein 和 Newcomer[1997] 在其教科书中讨论了事务处理的多个方面。

Eswaran 等[1976] 对可串行化的概念进行了形式化描述，这是同他们在 System R 的并发控制上的工作相关联的。

660 事务处理各个具体方面（如并发控制和故障恢复）的参考文献见第 15、16 和 26 章。

并发控制

在第14章中我们了解了事务最基本的特性之一是隔离性。然而，当数据库中有多个事务并发执行时，事务的隔离性不一定能保持。为保持事务的隔离性，系统必须对并发事务之间的相互作用加以控制；这种控制是通过一系列机制中的一个称为并发控制的机制来实现的。第26章讨论允许非可串行化调度的并发控制机制。在这一章中，我们考虑并发执行事务的管理，并且我们忽略故障。第16章将讨论系统如何从故障中恢复。

[661]

如我们将会看到的，并发控制有许多种机制。没有哪种机制是明显最好的；每种机制都有优势。在实践中，最常用的机制有两分段封锁和快照隔离。

15.1 基于锁的协议

确保隔离性的方法之一是要求对数据项以互斥的方式进行访问，换句话说，当一个事务访问某个数据项时，其他任何事务都不能修改该数据项。实现该需求最常用的方法是只允许事务访问当前该事务持有锁(lock)的数据项。14.9节介绍过锁的概念。

15.1.1 锁

给数据项加锁的方式有多种，在这一节中，我们只考虑两种：

1. **共享的**(shared)：如果事务 T_i 获得了数据项 Q 上的**共享型锁**(shared-mode lock)(记为S)，则 T_i 可读但不能写 Q。
2. **排他的**(exclusive)：如果事务 T_i 获得了数据项 Q 上的**排他型锁**(exclusive-mode lock)(记为X)，则 T_i 既可读又可写 Q。

我们要求每个事务都要根据自己将对数据项 Q 进行的操作类型申请(request)适当的锁。该事务将请求发送给并发控制管理器。事务只有在并发控制管理器授予(grant)所需锁后才能继续其操作。这两种锁类型的使用可以让多个事务读取一个数据项但是限制同时只能有一个事务进行写操作。

更普遍地说，对于给定的一个锁类型集合，我们可在它们上按如下方式定义一个**相容函数**(compatibility function)：令 A 与 B 代表任意的锁类型，假设事务 T_i 请求对数据项 Q 加 A 类型锁，而事务 T_j($T_i \neq T_j$)当前在数据项 Q 上拥有 B 类型锁。尽管数据项 Q 上存在 B 类型锁，如果事务 T_i 可以立即获得数据项 Q 上的锁，则我们就说 A 类型锁与 B 类型锁是**相容的**(compatible)。这样的一个函数可以通过矩阵方便地表示出来。本节所用的两类锁的相容关系由图15-1所示的矩阵 comp 给出。当且仅当类型 A 与类型 B 是相容的，该矩阵的一个元素 **comp**(A, B)具有 true 值。

注意共享型与共享型是相容的，而与排他型不相容。在任何时候，一个具体的数据项上可同时有(被不同的事务持有的)多个共享锁。此后的排他锁请求必须一直等待直到该数据项上的所有共享锁被释放。

	S	X
S	true	false
X	false	false

图15-1 锁相容性矩阵 comp

一个事务通过执行 **lock-S**(Q)指令来申请数据项 Q 上的共享锁。类似地，一个事务通过执行 **lock-X**(Q)指令来申请排他锁。一个事务能够通过 **unlock**(Q)指令来释放数据项 Q 上的锁。

要访问一个数据项，事务 T_i 必须首先给该数据项加锁。如果该数据项已被另一事务加上了不相容类型的锁，则在所有其他事务持有的不相容类型锁被释放之前，并发控制管理器不会授予锁。因此，T_i 只好**等待**(wait)，直到所有其他事务持有的不相容类型锁被释放。

事务 T_i 可以释放先前加在某个数据项上的锁。注意，一个事务只要还在访问数据项，它就必须拥有该数据项上的锁。此外，让事务在对数据项作最后一次访问后立即释放该数据项上的锁也未必是可取的，因为有可能不能保证可串行性。

举一个例子，再考虑我们在第 14 章中介绍的银行业务系统。令 A 与 B 是事务 T_1 与 T_2 访问的两个账户，事务 T_1 从账户 B 转 50 美元到账户 A 上(见图 15-2)，事务 T_2 显示账户 A 与 B 上的总金额，即 $A+B$(见图 15-3)。

```
T1: lock-x(B);
    read(B);
    B := B − 50;
    write(B);
    unlock(B);
    lock-x(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(A).
```

图 15-2 事务 T_1

```
T2: lock-s(A);
    read(A);
    unlock(A);
    lock-s(B);
    read(B);
    unlock(B);
    display(A + B).
```

图 15-3 事务 T_2

假设账户 A 与 B 的金额分别为 100 美元与 200 美元。如果这两个事务串行执行，以 T_1、T_2 或 T_2、T_1 的顺序执行，则事务 T_2 将显示的值为 300 美元。然而，如果两个事务并发地执行，则有可能出现如图 15-4 所示的调度 1。在这种情况下，事务 T_2 显示 250 美元，这是不对的。出现这种错误的原因是由于事务 T_1 过早释放数据项 B 上的锁，从而导致事务 T_2 看到一个不一致的状态。

T_1	T_2	并发控制管理器
lock-x(B)		
		grant-x(B, T_1)
read(B)		
B := B − 50		
write(B)		
unlock(B)		
	lock-s(A)	
		grant-s(A, T_2)
	read(A)	
	unlock(A)	
	lock-s(B)	
		grant-s(B, T_2)
	read(B)	
	unlock(B)	
	display(A + B)	
lock-x(A)		
		grant-x(A, T_1)
read(A)		
A := A + 50		
write(A)		
unlock(A)		

图 15-4 调度 1

该调度显示了事务执行的动作以及并发控制管理器授权加锁的时刻。申请加锁的事务在并发控制管理器授权加锁之前不能执行下一个动作。因此，锁的授予必然是在事务申请锁操作与事务的下一动作的间隔内。至于在此期间内授权加锁的准确时间并不重要：我们不妨假设锁正好在事务的下一动作前获得。因此，在本章余下部分所描述的调度中我们将去掉并发控制管理器授予锁的那一栏。我们让读者自己去推断何时授予锁。

现在假定事务结束后才释放锁。事务 T_1 对应于 T_1，它延迟了锁释放(见图 15-5)，事务 T_4 对应于 T_2，它延迟了锁释放(见图 15-6)。

```
T3: lock-x(B);
    read(B);
    B := B − 50;
    write(B);
    lock-x(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(B);
    unlock(A).
```

图 15-5 事务 T_3
(事务 T_3 延迟释放锁)

你可以验证，在调度1中导致显示不准确总和250美元的读写顺序，在事务 T_3 与 T_4 的调度中没有再出现的可能。我们也可以用别的一些调度，在任何调度当中，T_4 将不会显示不一致的结果；稍后我们将明白其中缘由。

```
T3: lock-S(A);
    read(A);
    lock-S(B);
    read(B);
    display(A + B);
    unlock(A);
    unlock(B).
```

图15-6 事务 T_3（事务 T_3 延迟释放锁）

遗憾的是，封锁可能导致一种不受欢迎的情形。考察如图15-7所示的有关事务 T_3 与 T_4 的部分调度，由于 T_3 在 B 上拥有排他锁，而 T_4 正在申请 B 上的共享锁，因此 T_4 等待 T_3 释放 B 上的锁；类似地，由于 T_4 在 A 上拥有共享锁，而 T_3 正在申请 A 上的排他锁，因此 T_3 等待 T_4 释放 A 上的锁。于是，我们进入了这样一种僵个事务都不能正常执行的状态。这种情形称为死锁(deadlock)。当死锁发生时，系统必须回滚两个事务中的一个。一旦某个事务回滚，该事务锁住的数据项就被解锁，其他事务就可以访问这些数据项，继续自己的执行。15.2节将再讨论死锁处理这个问题。

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
unlock(A)	

图15-7 调度2

我们如果不使用封锁，或者我们对数据项进行读写之后立即解锁，那么我们可能会进入不一致的状态。另一方面，如果在申请对另一数据项加锁之前如果我们不对当前锁住的数据项解锁，则可能会发生死锁。在某些情形下有办法避免死锁，我们将在15.1.5节讨论。然而，一般而言，如果我们为了避免不一致状态而采取封锁，则死锁是堆之而来的必然产物。产生死锁显然比产生不一致状态要好，因为它们可以通过回滚事务加以解决，而不一致状态可能引起现实中的问题，这是数据库系统不能处理的。

我们将要求在系统中的每一个事务遵从称为**封锁协议**(locking protocol)的一组规则，这些规则规定事务何时对数据项们进行加锁、解锁。封锁协议限制了可能的调度数目。这些调度组成的集合是所有可能的可串行化调度的一个真子集。我们将讲述几个封锁协议，它们只允许冲突可串行化调度，从而保证隔离性。在此之前，我们需要先给出几个定义。

令 $T_0, T_1, \cdots, T_n$ 是参与调度 S 的一个事务集，如果存在数据项 Q，使得 T_i 在 Q 上持有 A 型锁，后来，T_j 在 Q 上持有 B 型锁，且 $\mathbf{comp}(A, B) = \text{false}$，则我们称在 S 中 T_i **先于**(precede) T_j，记为 $T_i \rightarrow T_j$。如果 $T_i \rightarrow T_j$，这一居先意味着在任何等价的串行调度中，T_i 必须出现在 T_j 之前。注意这个图与14.6节中用于检测冲突可串行性的优先图是类似的。指令之间的冲突对应于锁类型之间的不相容性。

如果调度 S 是那些遵从封锁协议规则的事务集的可能调度之一，我们称调度 S 在给定的封锁协议下是**合法的**(legal)。当且仅当其所有合法的调度为冲突可串行化时，我们称一个封锁协议**保证**(ensure)冲突可串行性；换句话说，对于任何合法的调度，其关联的→关系是无环的。

15.1.2 锁的授予

当事务申请对一个数据项加某一类型锁，且没有其他事务在该数据项上加上了与此类型相冲突的锁，则可以授予锁。然而，必须小心防止出现下面的情形。假设事务 T_3 在一数据项上持有共享锁，另一事务 T_4 申请在该数据项加排他锁。显然，事务 T_4 必须等待事务 T_3 释放共享锁。同时，事务 T_5 可能申请对该数据项加共享锁，加锁请求与已授予 T_3 的锁是相容的，因此可以授权 T_5 加共享锁。此时，T_5 可能释放锁，但 T_4 还必须等待 T_3 完成。可是，可能又有一个新的事务 T_6 申请对该数据项加共享锁，并在 T_3 完成之前授予锁。事实上，有可能存在一个事务序列，其中每个事务申请对该数据项加共享锁，每个事务在授权加锁后一小段时间内释放锁，而 T_4 总是不能在该数据项上加排他锁。事务 T_4 可能永远不能取得进展。这称为**饿死**(starved)。

我们可以通过按如下方式授权加锁来避免事务饿死：当事务 T_i 申请对数据项 Q 加 M 型锁时，并发控制管理器授权加锁的条件是：

1. 不存在在数据项 Q 上持有与 M 型锁冲突的锁的其他事务。
2. 不存在等待对数据项 Q 加锁且先于 T_i 申请加锁的事务。

因此，一个加锁请求就不会被其后的加锁申请阻塞。

15.1.3 两阶段封锁协议

保证可串行性的一个协议是**两阶段封锁协议**(two-phase locking protocol)。该协议要求每个事务分两个阶段提出加锁和解锁申请。

1. **增长阶段**(growing phase)：事务可以获得锁，但不能释放锁。
2. **缩减阶段**(shrinking phase)：事务可以释放锁，但不能获得新锁。

最初，事务处于增长阶段，事务根据需要获得锁。一旦该事务释放了锁，它就进入了缩减阶段，并且不能再发出加锁请求。

例如，事务 T_3 与 T_4 是两阶段的。与此相反，事务 T_1 与 T_2 不是两阶段的。注意，解锁指令不必非得出现在事务末尾。例如，就事务 T_3 而言，我们可以把 **unlock**(B)指令放在紧跟指令 **lock-X**(A)之后，并且仍然保持两阶段封锁特性。

我们可以证明两阶段封锁协议保证冲突可串行化。对于任何事务，在调度中该事务获得其最后加锁的位置(增长阶段结束点)称为事务的**封锁点**(lock point)。这样，多个事务可以根据它们的封锁点进行排序，实际上，这个顺序就是事务的一个可串行化顺序。我们将此证明留为习题(见实践习题 15.1)。

两阶段封锁并不保证不会发生死锁。不难发现事务 T_3 与 T_4 是两阶段的，但在调度 2 中(如图 15-7 所示)，它们却发生死锁。

回想一下，在 14.7.2 节中，除了调度可串行化外，调度还应该是无级联的。在两阶段封锁协议下，级联回滚可能发生。作为示例，考虑如图 15-8 所示的部分调度。每个事务都遵循两阶段封锁协议，但在事务 T_5 的 **read**(A)步骤之后事务 T_5 发生故障，从而导致 T_6 与 T_7 级联回滚。

级联回滚可以通过将两阶段封锁修改为**严格两阶段封锁协议**(strict two-phase locking protocol)加以
667 避免。这个协议除了要求封锁是两阶段之外，还要求事务持有的所有排他锁必须在事务提交后方可释放。这个要求保证未提交事务所写的任何数据在该事务提交之前均以排他方式加锁，防止其他事务读这些数据。

另一个两阶段封锁的变体是**强两阶段封锁协议**(rigorous two-phase locking protocol)，它要求事务提交之前不得释放任何锁。我们很容易验证在强两阶段封锁条件下，事务可以按其提交的顺序串行化。

考察下面两个事务，我们只写出了其中较为重要的 **read** 与 **write** 指令：

T_8: **read**(a_1);
read(a_2);
...
read(a_n);
write(a_1).

T_9: **read**(a_1);
read(a_2);
display($a_1 + a_2$).

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

图 15-8 在两阶段封锁下的部分调度

如果我们采用两阶段封锁协议，则 T_8 必须对 a_1 加排他锁。因此，两个事务的任何并发执行方式都相当于串行执行。然而，请注意，T_9 仅在其执行结束，写 a_1 时需要对 a_1 加排他锁。因此，如果 T_9 开始时对 a_1 加共享锁，然后在需要时将其变更为排他锁，那么我们可以获得更高的并发度。因为 T_8 与 T_9 可以同时访问 a_1 与 a_2。

以上观察提示我们对基本的两阶段封锁协议加以修改，使之允许**锁转换**(lock conversion)。我们将提供一种将共享锁升级为排他锁，以及将排他锁降级为共享锁的机制。我们用**升级**(upgrade)表示从共享到排他的转换，用**降级**(downgrade)表示从排他到共享的转换。锁转换不能随意进行。锁升级只能发
668 生在增长阶段，而锁降级只能发生在缩减阶段。

回到我们的例子，事务 T_8 与 T_9 可在修改后的两阶段封锁协议下并发执行，如图 15-9 所示。其中的调度是不完整的，它只给出了一些封锁指令。

注意，事务试图升级数据项 Q 上的锁时可能不得不等待。这种情况发生在另一个事务当前对 Q 持有共享锁时。

像基本的两阶段封锁协议一样，具有锁转换的两阶段封锁协议只产生冲突可串行化的调度，并且事务可以根据其封锁点作串行化。此外，如果排他锁直到事务结束时才释放，则调度是无级联的。

对于一个事务集，可能存在某些不能通过两阶段封锁协议得到的冲突可串行化调度。然而，如果要通过非两阶段封锁协议得到冲突可串行化调度，我们或者需要事务的附加信息，或者要求数据库中的数据项集合有一定的结构或顺序。在本章后面当我们考虑其他封锁协议时，我们将看到例子。

T_{28}	T_{29}
lock-S(a_1)	
	lock-S(a_3)
lock-S(a_2)	
	lock-S(a_4)
lock-S(a_3)	
lock-S(a_4)	
	unlock(a_3)
	unlock(a_4)
lock-S(a_1)	
upgrade(a_1)	

图 15-9 带有锁转换的不完整调度

严格两阶段封锁与强两阶段封锁（含锁转换）在商用数据库系统中广泛使用。

这里介绍一种简单却广泛使用的机制，它基于来自事务的读、写请求，自动地为事务产生适当的加锁、解锁指令：

- 当事务 T_i 进行 **read**(Q)操作时，系统就产生一条 **lock-S**(Q)指令，该 **read**(Q) 指令紧跟其后。
- 当事务 T_i 进行 **write**(Q)操作时，系统检查 T_i 是否已在 Q 上持有共享锁。若有，则系统发出 **upgrade**(Q)指令，后接 **write**(Q)指令。否则系统发出 **lock-X**(Q)指令，后接 **write**(Q)指令。 669
- 当一个事务提交或中止后，该事务持有的所有锁都被释放。

15.1.4 封锁的实现

锁管理器(lock manager)可以实现为一个过程，它从事务接受消息并反馈消息。锁管理器过程针对锁请求消息返回授予锁消息，或者要求事务回滚的消息（发生死锁时）。解锁消息只需要得到一个确认回答，但可能引发为其他等待事务的授予锁消息。

锁管理器使用以下数据结构：锁管理器为目前已加锁的每个数据项维护一个链表，每一个请求为链表中一条记录，按请求到达的顺序排序。它使用一个以数据项名称为索引的散列表来查找链表中的数据项（如果有的话）；这个表叫**散锁表**（lock table）。一个数据项的链表中每一条记录表示由某个事务提出的请求，以及它请求什么类型的锁，该记录还表示该请求是否已授予锁。

图 15-10 是一个锁表的示例，该表包含 5 个不同数据项 I4、I7、I23、I44 和 I912 的锁。锁表采用溢出链，因此对于锁表的每一个表项都有一个数据项的链表。每一个数据项都有一个已授予锁或等待授予锁的事务列表，已授予锁的事务用深色阴影方块表示，等待授予锁的事务则用浅色阴影方块表示。为了简化图形我们省略了锁的类型。举个例子，从图 15-10 中可以看到，T23 在数据项 I912 和 I7 上已被授予锁，并且正在等待在 I4 上加锁。

虽然图 15-10 上没有标示出来，但锁表还应当维护一个基于事务标识符的索引，这样它可以有效地确定给定事务持有的锁集。

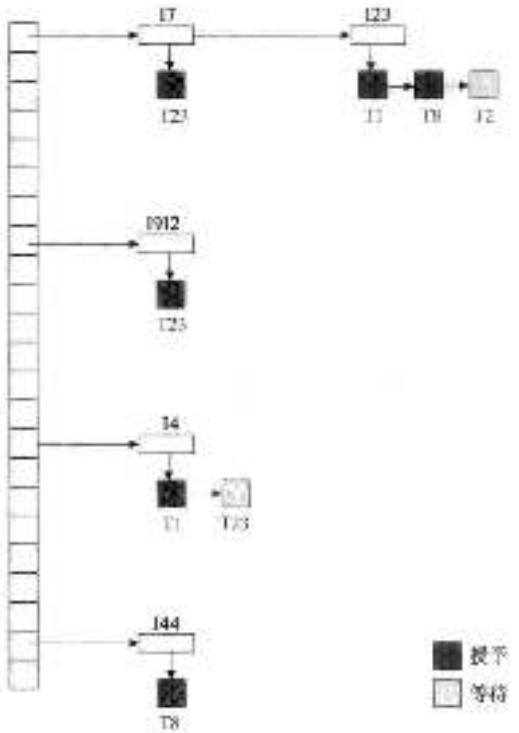


图 15-10 锁表

锁管理器这样处理请求：

- 当一条锁请求消息到达时，如果相应数据项的链表存在，在该链表末尾增加一个记录；否则，新建一个仅包含该请求记录的链表。

  在当前没有加锁的数据项上总是授予第一次加锁请求。但当事务向已被加锁的数据项申请加锁时，只有当该请求与当前持有的锁相容，并且所有先前的请求都已授予锁的条件下，锁管理器才为该请求授予锁。否则，该请求只好等待。
- 当锁管理器收到一个事务的解锁消息时，它将与该事务相对应的数据项链表中的记录删除，然后检查随后的记录，如果有，如前所述，就看该请求能否被授权，如果能，锁管理器授权该请求并处理其后记录，如果还有，类似地一个接一个地处理。
- 如果一个事务中止，锁管理器删除该事务产生的正在等待加锁的所有请求。一旦数据库系统采取适当动作撤销该事务（见16.3节），该中止事务持有的所有锁将被释放。

这个算法保证了锁请求无饿死现象，因为在先前接收到的请求正在等待加锁时，后来者不可能获得授权。我们稍后将在15.2.2节学习检测和处理死锁。17.2.1节描述另一个实现——在锁申请/授权上利用共享内存取代消息传递。

15.1.5 基于图的协议

正如15.1.3节提到的，如果我们要开发非两阶段协议，我们需要有关每个事务如何存取数据库的附加信息。有多种不同模型可以为我们提供这些附加信息，每一种所提供的信息量大小不同。最简单的模型要求我们事先知道访问数据项的顺序。在已知这些信息的情况下，有可能构造出非两阶段的封锁协议，并且无论如何，仍然保证冲突可串行性。

670
671

为获取这些事先的知识，我们要求所有数据项集合 $\mathbf{D}=\{d_1, d_2, \cdots, d_h\}$ 满足偏序→：如果 $d_i \rightarrow d_j$，则任何既访问 d_i 又访问 d_j 的事务必须首先访问 d_i，然后访问 d_j。这种偏序可以是数据的逻辑或物理组织的结果，也可以只是为了并发控制而加上的。

偏序意味着集合 **D** 可以视为有向无环图，称为**数据库图**（database graph）。在本节中，为了简单起见，我们只关心那些带根树的图。我们将给出一个称为树形协议的简单协议，该协议只使用排他锁。其他更复杂的基于图的封锁协议可以参阅文献注解中给出的有关文献。

在**树形协议**（tree protocol）中，可用的加锁指令只有 **lock-X**。每个事务 T_i 对一数据项最多能加一次锁，并且必须遵从以下规则：

1. T_i 首次加锁可以对任何数据项进行。
2. 此后，T_i 对数据项 Q 加锁的前提是 T_i 当前持有 Q 的父项上的锁。
3. 对数据项解锁可以随时进行。
4. 数据项被 T_i 加锁并解锁后，T_i 不能再对该数据项加锁。

所有满足树形协议的调度是冲突可串行化的。

为了说明这个协议，考察如图15-11所示的数据库图。下面4个事务遵从该图的树形协议。我们只列出了加锁、解锁指令：

672

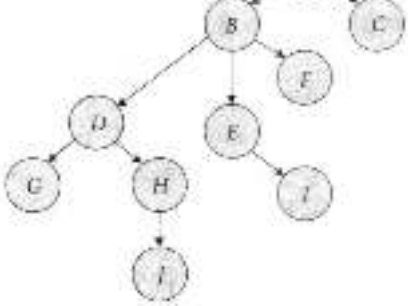


图15-11 树形结构数据库图

T_{10}: **lock-X**(B); **lock-X**(E); **lock-X**(D); **unlock**(B); **unlock**(E); **lock-X**(G); **unlock**(D); **unlock**(G).
T_{11}: **lock-X**(D); **lock-X**(H); **unlock**(D); **unlock**(H).
T_{12}: **lock-X**(B); **lock-X**(E); **unlock**(E); **unlock**(B).
T_{13}: **lock-X**(D); **lock-X**(H); **unlock**(D); **unlock**(H).

这4个事务参与的一个可能的调度如图15-12所示。注意，在执行时，事务 T_{10} 持有两棵互相分离的子树的锁。

不难发现如图15-12所示的调度是冲突可串行化的。可以证明树形协议不仅保证冲突可串行性，而且保证不会产生死锁。

图 15-12 所示的树形协议不保证可恢复性和无级联回滚。为了保证可恢复性和无级联回滚，可以将协议修改为在事务结束前不允许释放排他锁。直到事务结束前一直持有排他锁降低了并发性。这里有一个提高并发性的替代方案，但它只保证可恢复性：为每一个发生了未提交写操作的数据项，我们记录是哪个事务最后对它执行了写操作。当事务 T_i 执行了对未提交数据项的读操作，我们就在最后对该数据项执行了写操作的事务上记录一个 T_i 的**提交依赖**(commit dependency)。在有 T_i 的提交依赖的所有事务提交完成之前，T_i 不得提交。如果其中任何一个事务中止，T_i 也必须中止。

与两阶段封锁协议不同，树形协议不会产生死锁，因此不需要回滚，这一点树形封锁协议要优于两阶段封锁协议。树形封锁协议的另一个优点是可以在较早时候释放锁。较早解锁减少了等待时间，增加了并发性。

然而，在某些情况下，该协议也有其缺点，事务有可能必须给那些根本不访问的数据项加锁。例如，如果某事务要访问图 15-11 所示数据库图中的数据项 A 与 J，则该事务不仅要给 A 与 J 加锁，而且还要给 B、D 与 H 数据项加锁。这种额外的封锁导致封锁开销增加，可能造成额外的等待时间，并且可能引起并发性降低。此外，如果事先没有得到哪些数据项需要加锁的知识，事务将必须给树根加锁，这会大大降低并发性。

对于一个事务集，可能存在某些不能通过树形封锁协议得到的冲突可串行化调度。事实上，一些两阶段封锁协议中可行的调度在树形封锁协议下是不可行的，反之亦然。习题中对这样的调度的例子进行了探讨。

T_{10}	T_{11}	T_{12}	T_{13}
lock-X(B)			
	lock-X(D) lock-X(H) unlock(D)		
lock-X(E) lock-X(D) unlock(B) unlock(E)			
		lock-X(B) lock-X(E)	
	unlock(H)		
lock-X(G) unlock(D)			
			lock-X(D) lock-X(H) unlock(D) unlock(H)
		unlock(E) unlock(B)	
unlock(G)			

图 15-12 在树形协议下的可串行化调度

15.2 死锁处理

如果存在一个事务集，该集合中的每个事务在等待该集合中的另一个事务。那么我们说系统处于死锁状态。更确切地说，存在一个等待事务集 $\{T_0, T_1, \cdots T_n\}$，使得 T_0 正等待被 T_1 锁住的数据项，T_1 正等待被 T_2 锁住的数据项，…，T_{n-1} 正等待被 T_n 锁住的数据项，且 T_n 正等待被 T_0 锁住的数据项。在这种情况下，没有一个事务能取得进展。

此时，系统唯一的补救措施是采取激烈的动作，如回滚某些陷于死锁的事务。事务有可能只部分回滚，即事务回滚到这样的点之前，它在该点得到一个锁，而释放该锁就可以解决死锁。

处理死锁问题有两种主要的方法。我们可以使用**死锁预防**(deadlock prevention)协议保证系统永不进入死锁状态。另一种方法是，我们允许系统进入死锁状态，然后试着用**死锁检测**(deadlock detection)与**死锁恢复**(deadlock recovery)机制进行恢复。正如我们将要看到的那样，两种方法均有可能引起事务回滚。如果系统进入死锁状态的概率相对较高，则通常使用死锁预防机制；否则，使用检测与恢复机制会更有效。

注意，检测与恢复机制带来各种开销，不仅包括在运行时维护必要信息及执行检测算法的代价，还要包括从死锁中恢复所固有的潜在损失。 673 674

15.2.1 死锁预防

预防死锁有两种方法。一种方法是通过对加锁请求进行排序或要求同时获得所有的锁来保证不会发生循环等待。另一种方法比较接近死锁恢复，每当等待有可能导致死锁时，进行事务回滚而不是等待加锁。

第一种方法下最简单的机制要求每个事务在开始之前封锁它的所有数据项。此外，要么一次全部封锁要么全不封锁。这个协议有两个主要的缺点：(1)在事务开始前通常很难预知哪些数据项需要封

锁。(2)数据项使用率可能很低，因为许多数据项可能封锁很长时间却用不到。

防止死锁的另一种机制是对所有的数据项强加一个次序，同时要求事务只能按次序规定的顺序封锁数据项。我们曾经在树形协议中讲述过一个这样的机制，其中采用一个偏序的数据项。

该方法的一个变种是使用数据项与两阶段封锁关联的全序。一旦一个事务锁住了某个特定的数据项，它就不能申请顺序中位于该数据项前面的数据项上的锁。只要在事务开始执行之前，它要访问的数据项集是已知的，该机制就容易实现。如果使用了两阶段封锁，潜在的并发控制系统就不需要重改；所需要的是，保证锁的申请按照正确的顺序。

防止死锁的第二种方法是使用抢占与事务回滚。在抢占机制中，若事务 T_j 所申请的锁已被事务 T_i 持有，则授予 T_i 的锁可能通过回滚事务 T_i 被**抢占**(preempted)，并将锁授予 T_j。为控制抢占，我们给每个事务赋一个唯一的时间戳，系统仅用时间戳来决定事务应当等待还是回滚。并发控制仍使用封锁机制。若一个事务回滚，则该事务重启时保持原有的时间戳。已提出利用时间戳的两种不同的死锁预防机制：

1. **wait-die** 机制基于非抢占技术。当事务 T_i 申请的数据项当前被 T_j 持有，仅当 T_i 的时间戳小于 T_j 的时间戳(即，T_i 比 T_j 老)时，允许 T_i 等待。否则，T_i 回滚(死亡)。

例如，假设事务 T_{14}、T_{15}及 T_{16}的时间戳分别为 5、10 与 15。如果 T_{14}申请的数据项当前被 T_{15}持有，则 T_{14}将等待。如果 T_{16}申请的数据项当前被 T_{15}持有，则 T_{16}将回滚。

2. **wound-wait** 机制基于抢占技术，是与 wait-die 相反的机制。当事务 T_i 申请的数据项当前被 T_j 持有，仅当 T_i 的时间戳大于 T_j 的时间戳(即，T_i 比 T_j 年轻)时，允许 T_i 等待。否则，T_j 回滚(T_j 被 675 T_i 伤害)。

回到前面的例子，对于事务 T_{14}、T_{15}及 T_{16}。如果 T_{15}申请的数据项当前被 T_{16}持有，则 T_{15}将从 T_{16}抢占该数据项，T_{16}将回滚。如果 T_{14}申请的数据项当前被 T_{15}持有，则 T_{14}将等待。

两种机制都面临的主要问题是可能发生不必要的回滚。

另一种处理死锁的简单方法基于**锁超时**(lock timeout)。在这种方法中，申请锁的事务至多等待一段给定的时间。若在此期间内未授予该事务锁，则称该事务超时，此时该事务自己回滚并重启。如果确实存在死锁，卷入死锁的一个或多个事务将超时并回滚，允许其他事务继续。该机制介于死锁预防(不会发生死锁)与死锁检测及恢复之间，死锁检测与恢复在 15.2.2 节讲述。

超时机制的实现极其容易，并且如果事务是短事务并且长时间等待很可能由死锁引起的，该机制运作良好。然而，一般而言很难确定一个事务超时之前应等待多长时间。如果已发生死锁，等待时间太长导致不必要的延迟。如果等待时间太短，即便没有死锁，也可能引起事务回滚，造成资源浪费。该机制也可能会产生饿死。因此，基于超时的机制应用有限。

15.2.2 死锁检测与恢复

如果系统没有采用能保证不产生死锁的协议，那么系统必须采用检测与恢复机制。检查系统状态的算法周期性地激活，判断有无死锁发生。如果发生死锁，则系统必须试着从死锁中恢复。为实现这一点，系统必须：

- 维护当前将数据项分配给事务的有关信息，以及任何尚未解决的数据项请求信息。
- 提供一个使用这些信息判断系统是否进入死锁状态的算法。
- 当检测算法判定存在死锁时，从死锁中恢复。

本节详细讲述上述问题。

15.2.2.1 死锁检测

死锁可以用称为**等待图**(wait-for graph)的有向图来精确描述。该图由 $G=(V, E)$ 对组成，其中 V 676 是顶点集，E 是边集。顶点集由系统中的所有事务组成，边集 E 的每一元素是一个有序对 $T_i \rightarrow T_j$。如果 $T_i \rightarrow T_j$ 属于 E，则存在从事务 T_i 到 T_j 的一条有向边，表示事务 T_i 在等待 T_j 释放所需数据项。

当事务 T_i 申请的数据项当前被 T_j 持有时，边 $T_i \rightarrow T_j$ 被插入等待图中。只有当事务 T_j 不再持有事务 T_i 所需数据项时，这条边才从等待图中删除。

当且仅当等待图包含环时，系统中存在死锁。在该环中的每个事务称为处于死锁状态。要检测死锁，系统需要维护等待图，并定期性地激活一个在等待图中搜索环的算法。

为说明这些概念，考虑图 15-13 所示的等待图，它说明了下面这些情形：

- 事务 T_{25} 在等待事务 T_{26} 与 T_{29}。
- 事务 T_{27} 在等待事务 T_{26}。
- 事务 T_{26} 在等待事务 T_{29}。

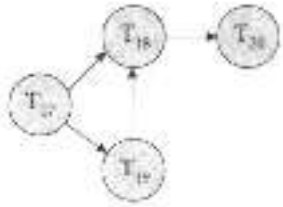


图 15-13 无环等待图

由于该等待图无环，因此系统没有处于死锁状态。

现在假设事务 T_{29} 申请事务 T_{27} 持有的数据项。边 $T_{29}\rightarrow T_{27}$ 被加入等待图中，得到图 15-14 所示的系统新状态。此时，该等待图包含环：

$$T_{26}\rightarrow T_{25}\rightarrow T_{27}\rightarrow T_{26}$$

意味着事务 T_{25}、T_{26} 与 T_{27} 都处于死锁状态。

由此，引出下面的问题：我们何时激活检测算法？答案取决于两个因素：

1. 死锁发生频度怎样？
2. 有多少事务将受到死锁的影响？

如果死锁频繁发生，则检测算法应比通常情况下激活得更频繁。分配给处于死锁状态的事务的数据项在死锁解除之前不能为其他事务获取。此外，等待图中环的数目也可能增大。在最坏的情况下，我们要在每个分配请求不能立即满足时激活检测算法。 677

15.2.2.2 从死锁中恢复

当一个检测算法判定存在死锁时，系统必须从死锁中**恢复**(recover)。解除死锁最通常的做法是回滚一个或多个事务，需采取的动作有三个。

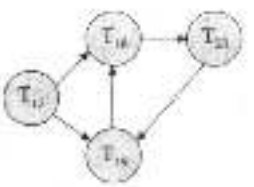


图 15-14 有一个环的等待图

1. **选择牺牲者**：给定处于死锁状态的事务集，为解除死锁，我们必须决定回滚哪一个(或哪一些)事务以打破死锁。我们应使事务回滚带来的代价最小。可惜"最小代价"这个词并不准确。很多因素影响事务回滚代价，其中包括：
   a. 事务已计算了多久，并在完成其指定任务之前该事务还将计算多长时间。
   b. 该事务已使用了多少数据项。
   c. 为完成事务还需使用多少数据项。
   d. 回滚时将牵涉多少事务。
2. **回滚**：一旦我们决定了要回滚哪个事务，就必须决定该事务回滚多远。最简单的方法是**彻底回滚**(total rollback)：中止该事务，然后重新开始。然而，事务只回滚到可以解除死锁处会更有效。这种**部分回滚**(partial rollback)要求系统维护所有正在运行事务的额外状态信息。确切地说，需要记录锁的申请/授予序列和事务执行的更新。死锁检测机制应当确定，为打破死锁，选定的事务需要释放哪些锁。选定的事务必须回滚到获得这些锁的第一个之前，并取消它在此之后的所有动作。恢复机制必须能够处理这种部分回滚。而且，事务必须能够在部分回滚之后恢复执行。有关参考文献见文献注解。 678
3. **饿死**：在系统中，如果选择牺牲者主要基于代价因素，有可能同一事务总是被选为牺牲者。这样一来，该事务总是不能完成其指定任务，这样就发生**饿死**(starvation)。我们必须保证一个事务被选为牺牲者的次数有限(且较少)。最常用的方案是在代价因素中包含回滚次数。

15.3 多粒度

到目前为止所讲的并发控制机制中，我们将一个个数据项作为进行同步执行的单元。

然而，某些情况下需要把多个数据项聚为一组，将它们作为一个同步单元。因为这样会更好。例如，如果事务 T_i 需要访问整个数据库，使用的是一种封锁协议，则事务 T_i 必须给数据库中每个数据项加锁。显然，执行这些加锁操作是很费时的。要是 T_i 能够只发出单个封锁整个数据库的加锁请求，那会更好。另一方面，如果事务 T_i 只需要存取少量数据项，就不应要求给整个数据库加锁，否则并发性就丧失了。

我们需要的是一种允许系统定义多级粒度(granularity)的机制。这通过允许各种大小的数据项并定义数据粒度的层次结构，其中小粒度数据项嵌套在大粒度数据项中来实现。这种层次结构可以图形化地表示为树。注意，这里所描述的树与树形协议(见15.1.5节)所用的树的概念完全不同。多粒度树中的非叶结点表示与其后代相联系的数据。在树形协议中，每个结点是一个相互独立的数据项。

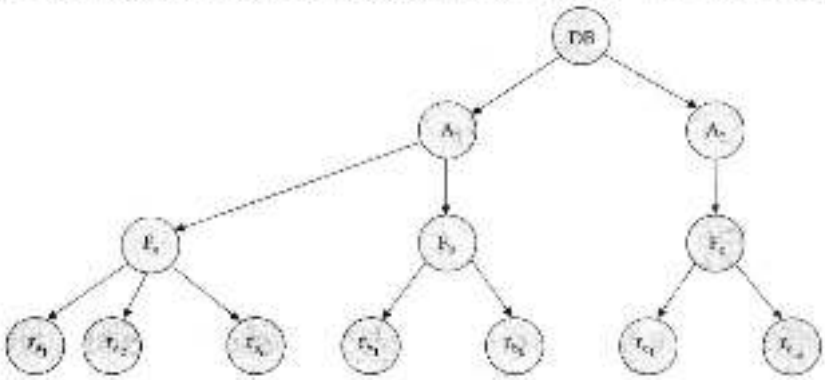


图 15-15 粒度层次图

作为例子，考虑图15-15所示的树。它由4层结点组成，最高层表示整个数据库。其下是 *area* 类型的结点，数据库恰好由这些区域组成。每个区域又以 *file* 类型结点作为子结点，每个区域恰好由作为其子结点的文件结点组成。没有任何文件处于一个以上区域中。最后，每个文件由 *record* 类型的结点组成。和前面一样，文件恰好由作为其子结点的记录组成。并且任何记录不能同时属于多个文件。

树中每个结点都可以单独加锁。正如我们在两阶段封锁协议中所做的那样，我们将使用**共享**(shared)锁与**排他**(exclusive)锁。当事务对一个结点加锁，或为共享锁或为排他锁，该事务也以同样类型的锁隐式地封锁这个结点的全部后代结点。例如，若事务 T_i 给图15-15中的 F_a **显式**(explicit lock)地
679 加排他锁，则事务 T_i 也给所有属于该文件的记录**隐式**(implicit lock)地加排他锁。没有必要显式地给 F_a 中的单条记录逐个加锁。

假设事务 T_j 希望封锁文件 F_a 的记录 r_{a_2}。由于 T_i 显式地给 F_a 加锁，因此意味着 r_{a_2} 也被加锁(隐式地)。但是，当 T_j 发出对 r_{a_2} 加锁的请求时，r_{a_2} 没有显式加锁！系统如何判定是否 T_j 可以封锁 r_{a_2} 呢？T_j 必须从树根到 r_{a_2} 进行遍历，如果发现此路径上某个结点的锁与要加的锁类型不相容，则 T_j 必须延迟。

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

图 15-16 相容性矩阵

现假设事务 T_k 希望封锁整个数据库。为此，它只需给层次结构图的根结点加锁。不过，请注意 T_k 给根结点加锁不会成功，因为目前 T_i 在树的某部分持有锁(具体地说，对文件 F_a 持有锁)。但是，系统是怎样判定根结点是否可以加锁呢？一种可能的方法是搜索整棵树。然而，这个方法破坏了多粒度封锁机制的初衷。获取这种知识的一个更有效的方法是引入一种新的锁类型，即**意向锁类型**(intention lock mode)。如果一个结点加上了意向锁，则意味着要在树的较低层进行显式加锁(也就是说，以更小的粒度加锁)。在一个结点显式加锁之前，该结点的全部祖先结点均加上了意向锁。因此，事务不必搜索整棵树就能判定能否成功地给一个结点加锁。希望给某个结点(如 Q)加锁的事务必须遍历从根到 Q 的路径，在遍历树的过程中，该事务给各结点加上意向锁。

与共享锁相关联的是一种意向锁，与排他锁相关联的是另一种意向锁。如果一个结点加上了共享

型意向(intention-shared (IS) mode)锁，那么将在树的较低层进行显式封锁，但只能加共享锁。类似地，如果一个结点加**排他型意向**(intention-exclusive(IX) mode)锁，那么将在树的较低层进行显式封锁，可以加排他锁或共享锁。最后，若一个结点加上了**共享排他型意向锁**(shared and intention-exclusive (SIX) mode)，则以该结点为根的子树显式地加了共享锁，并且将在树的更低层显式地加排他锁。这些锁类型的相容函数如图 15-16 所示。 680

多粒度封锁协议(multiple-granularity locking protocol)采用这些锁类型保证可串行性。每个事务 T_i 要求按如下规则对数据库 Q 加锁：

1. 事务 T_i 必须遵从图 15-16 所示的锁类型相容函数。
2. 事务 T_i 必须首先封锁树的根结点，并且可以加任意类型的锁。
3. 仅当 T_i 当前对 Q 的父结点具有 IX 或 IS 锁时，T_i 对结点 Q 可加 S 或 IS 锁。
4. 仅当 T_i 当前对 Q 的父结点具有 IX 或 SIX 锁时，T_i 对结点 Q 可加 X、SIX 或 IX 锁。
5. 仅当 T_i 未曾对任何结点解锁时，T_i 可对结点加锁（也就是说，T_i 是两阶段的）。
6. 仅当 T_i 当前不持有 Q 的子结点的锁时，T_i 可对结点 Q 解锁。

多粒度协议要求加锁按自顶向下的顺序(根到叶)，而锁的释放则按自底向上的顺序(叶到根)。

作为该协议的例子，考虑图 15-15 所示的树及下面的事务：

- 假设事务 T_{21} 读文件 F_a 的记录 r_{a6}。那么，T_{21} 需给数据库、区域 A_1，以及 F_a 加 IS 锁(按此顺序)，最后给 r_{a6} 加 S 锁。
- 假设事务 T_{22} 要修改文件 F_a 的记录 r_{a9}。那么，T_{22} 需给数据库、区域 A_1，以及文件 F_a(按此顺序)加 IX 锁，最后给记录 r_{a9} 加 X 锁。
- 假设事务 T_{23} 要读取文件 F_a 的所有记录。那么，T_{23} 需给数据库和区域 A_1(按此顺序)加 IS 锁， 681
  最后给 F_a 加 S 锁。
- 假设事务 T_{24} 要读取整个数据库。它在给数据库加 S 锁后就可读取。

我们注意到事务 T_{21}、T_{23} 与 T_{24} 可以并发地存取数据库。事务 T_{22} 可以与 T_{21} 并发执行，但不能与 T_{23} 或 T_{24} 并发执行。

该协议增强了并发性，减少了锁开销。它在由如下事务类型混合而成的应用中尤其有用：

- 只访问几个数据项的短事务。
- 生成整个文件或一组文件来生成报表的长事务。

类似的封锁协议可以用于数据粒度按有向无环图组织的数据库系统中，参考文献见文献注解。与在两阶段封锁协议中一样，在多粒度协议中也可能存在死锁。有多种技术可以用来减少多粒度协议中死锁发生的频度，甚至可以彻底消除死锁。文献注解给出了这些技术的相关资料。

15.4 基于时间戳的协议

到目前为止我们所讲述的封锁协议中，每一对冲突事务的次序是在执行时由二者都申请的，但类型不相容的第一个锁决定的。另一种决定事务可串行化次序的方法是事先选定事务的次序。其中最常用的方法是时间戳排序机制。

15.4.1 时间戳

对于系统中每个事务 T_i，我们把一个唯一的固定时间戳和它联系起来，此时间戳记为 $TS(T_i)$。该时间戳是在事务 T_i 开始执行前由数据库系统赋予的。若事务 T_i 已赋予时间戳 $TS(T_i)$，此时有一新事务 T_j 进入系统，则 $TS(T_i) < TS(T_j)$。实现这种机制可以采用下面这两个简单的方法：

1. 使用**系统时钟**(system clock)的值作为时间戳；即，事务的时间戳等于该事务进入系统时的时钟值。

2. 使用**逻辑计数器**(logical counter)，每赋予一个时间戳，计数器增加计数；即，事务的时间戳等于该事务进入系统时的计数器值。 682

事务的时间戳决定了串行化顺序。因此，若 $TS(T_i) < TS(T_j)$，则系统必须保证所产生的调度等价于事务 T_i 出现在事务 T_j 之前的某个串行调度。

要实现这个机制，每个数据项 Q 需要与两个时间戳值相关联：

- **W-timestamp**(Q)表示成功执行 **write**(Q)的所有事务的最大时间戳。
- **R-timestamp**(Q)表示成功执行 **read**(Q)的所有事务的最大时间戳。

每当有新的 **read**(Q) 或 **write**(Q)指令执行时，这些时间戳就更新。

15.4.2 时间戳排序协议

时间戳排序协议(timestamp-ordering protocol)保证任何有冲突的 **read** 或 **write** 操作按时间戳顺序执行。该协议运作方式如下：

1. 假设事务 T_i 发出 **read**(Q)。
   a. 若 TS(T_i) < W-timestamp(Q)，则 T_i 需要读入的 Q 值已被覆盖。因此，read 操作被拒绝，T_i 回滚。
   b. 若 TS(T_i) ≥ W-timestamp(Q)，则执行 read 操作，R-timestamp(Q)被设置为 R-timestamp(Q)与 TS(T_i)两者的最大值。
2. 假设事务 T_i 发出 **write**(Q)。
   a. 若 TS(T_i) < R-timestamp(Q)，则 T_i 产生的 Q 值是先前所需要的值，且系统已假定该值不会再产生。因此，**write** 操作被拒绝，T_i 回滚。
   b. 若 TS(T_i) < W-timestamp(Q)，则 T_i 试图写入的 Q 值已过时。因此，**write** 操作被拒绝，T_i 回滚。
   c. 其他情况，系统执行 **write** 操作，将 W-timestamp(Q)设置为 TS(T_i)。

如果事务 T_i 由于发出 **read** 或 **write** 操作而被并发控制机制回滚，则系统赋予它新的时间戳并重新启动。

683 为说明这个协议，我们考虑事务 T_{25} 与 T_{26}。事务 T_{25} 显示账户 A 与 B 的内容：

T_{25}: **read**(B);
read(A);
display(A + B).

事务 T_{26} 从账户 B 转 50 美元到账户 A，然后显示两个账户的内容：

T_{26}: **read**(B);
$B := B - 50$;
write(B);
read(A);
$A := A + 50$;
write(A);
display(A + B).

在说明时间戳协议下的调度时，我们将假设事务在执行第一条指令之前的那一刻被赋予时间戳。因此，在如图 15-17 所示的调度 3 中，TS(T_{25}) < TS(T_{26})，这是满足时间戳协议的一个可能的调度。

我们注意到前面的执行过程也可以由两阶段封锁协议产生。不过，存在满足两阶段封锁协议却不满足时间戳协议的调度，反之亦然(见习题 15.29)。

时间戳排序协议保证冲突可串行化，这因为冲突操作被时间戳顺序进行处理。

该协议保证无死锁，因为不存在等待的事务。但是，当一系列冲突的短事务引起长事务反复重启时，可能导致长事务饿死的现象。如果发现一个事务反复重启，与之冲突的事务应当暂时阻塞，以使该事务能够完成。

该协议可能产生不可恢复的调度。然而，该协议可以进行扩展，用以下几种方法之一来保证调度可恢复：

- 在事务末尾执行所有的写操作能保证可恢复性和无级联性。这

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

图 15-17 调度 3

些写操作必须具有下述意义的原子性：在写操作正在执行的过程中，任何事务都不许访问已写完的任何数据项。

- 可恢复性和无级联性也可以通过使用一个受限的封锁形式来保证。由此，对未提交数据项的读操作被推迟到更新该数据项的事务提交之后（见习题 15.30）。
- 可恢复性可以通过跟踪未提交写操作来单独保证，一个事务 T_i 读取了其他事务所写的数据，只有在其他事务都提交之后，T_i 才能提交。15.1.5 节概述过的提交依赖可以用作这个目的。

15.4.3 Thomas 写规则

我们现在给出对时间戳排序协议的一种修改，它允许的并发程度比 15.4.2 节中的协议要高。让我们考虑图 15-18 所示的调度 4，并应用时间戳排序协议。由于 T_{27} 先于 T_{28} 开始，因此假定 TS(T_{27}) < TS(T_{28})。T_{27} 的 **read**(Q)操作成功，T_{28} 的 **write**(Q)操作也成功。当 T_{27} 试图进行 write(Q)操作时，我们发现 TS(T_{27}) < W-timestamp(Q)，因为 W-timestamp(Q) = TS(T_{28})。所以，T_{27} 的 write(Q) 操作被拒绝且事务 T_{27} 回滚。

T_{27}	T_{28}
read(Q)	
	write(Q)
write(Q)	

图 15-18 调度 4

虽然事务 T_{27} 回滚是时间戳排序协议所要求的，但这是不必要的。由于 T_{28} 已经写入了 Q，因此 T_{27} 想要写入的值将永远不会被读到。满足 TS(T_i) < TS(T_{28})的任何事务 T_i 试图进行 **read**(Q)操作时均回滚，因为 TS(T_i) < W-timestamp(Q)。满足 TS(T_i) > TS(T_{28})的任何事务 T_i 必须读出 T_{28} 写入的 Q 值，而不是 T_{27} 想要写入的值。

根据以上分析，我们可以修改时间戳排序协议而得到一个新版本的协议，该协议在某些特定的情况下忽略过时的 **write** 操作。协议中有关 **read** 操作的规则保持不变；但 **write** 操作的协议规则与 15.4.2 节中的时间戳排序协议略有区别。 684 - 685

时间戳排序协议所做的这种修改称为 Thomas 写规则(Thomas' write rule)：假设事务 T_i 发出 **write**(Q)。

1. 若 TS(T_i) < R-timestamp(Q)，则 T_i 产生的 Q 值是先前所需要的值，且系统已假定该值不会再产生。因此，**write** 操作被拒绝，T_i 回滚。
2. 若 TS(T_i) < W-timestamp(Q)，则 T_i 试图写入的 Q 值已过时。因此，这个 **write** 操作可忽略。
3. 其他情况，执行 **write** 操作，将 W-timestamp(Q)设置为 TS(T_i)。

以上规则与 15.4.2 节中规则的区别在第二条，时间戳排序协议在 T_i 发出 **write**(Q)且 TS(T_i) < W-timestamp(Q)时要求 T_i 回滚。而在修改后协议对这种情况的处理是，在 TS(T_i) ≥ R-timestamp(Q)时，我们忽略过时的 **write** 操作。

通过忽略写，Thomas 写规则允许非冲突可串行化但是正确的调度。这些允许的非冲突可串行化调度满足视图可串行化调度的概念(见示例框)。Thomas 写规则实际上是通过删除事务发出的过时的 **write** 操作来使用视图可串行性。对事务的这种修改使得系统可以产生本章中其他协议所不能产生的可串行化调度。例如，图 15-18 所示调度 4 是非冲突可串行化的，因此在两阶段封锁协议、树形封锁协议或时间戳排序协议中都是不可能的。在 Thomas 写规则下，T_{27} 的 **write**(Q)操作将忽略，产生视图等价于串行调度 < T_{27}, T_{28} > 的一个调度。

视图可串行化

有一种等价形式比冲突等价约限制要宽松，但这种等价形式与冲突等价一样只基于事务的 **read** 与 **write** 操作。

考虑两个调度 S 与 S'，参与两个调度的事务集是相同的，若满足下面三个条件，调度 S 与 S' 就称为**视图等价**(view equivalent)的：

1. 对于每个数据项 Q，若事务 T_i 在调度 S 中读取了 Q 的初始值，那么在调度 S' 中 T_i 也必须读取 Q 的初始值。

2. 对于每个数据项 Q，若在调度 S 中事务 T_i 执行了 **read**(Q)并且读取的值是由事务 T_j 执行 **write**(Q)操作产生的，则在调度 S' 中，T_i 的 **read**(Q)操作读取的值 Q 也必须是由 T_j 的同一个 **write**(Q)操作产生的。

3. 对于每个数据项 Q，若在调度 S 中有事务执行了最后的 write(Q)操作，则在调度 S' 中该事务也必须执行最后的 write(Q)操作。

条件 1 与 2 保证在两个调度中的每个事务都读取相同的值，从而进行相同的计算。条件 3 与条件 1、2 一起保证两个调度得到相同的最终系统状态。

视图等价的概念引出了视图可串行化的概念。如果某个调度视图等价于一个串行调度，则我们说这个调度是**视图可串行化**(view serializable)的。

举个例子，假设我们在调度 4 中增加事务 T_{29}，由此得到以下视图可串行化的调度(调度 5)：

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

实际上，调度 5 视图等价于串行调度 $<T_{27}, T_{28}, T_{29}>$，因为在两个调度中 **read**(Q)指令均是读取 Q 的初始值，两个调度中 T29 均最后写入 Q 值。

每个冲突可串行化调度都是视图可串行化的，但存在非冲突可串行化的视图可串行化调度。事实上，调度 5 就不是冲突可串行化的，因为每对连续指令均冲突，从而交换指令是不可能的。

注意，在调度 5 中，事务 T_{28} 与 T_{29} 执行 **write**(Q)操作之前没有执行 **read**(Q)操作。这样的写操作称作**盲目写操作**(blind write)。盲目写操作存在于任何非冲突可串行化的视图可串行化调度中。

15.5 基于有效性检查的协议

在大部分事务是只读事务的情况下，事务发生冲突的频率较低。因此，许多这样的事务，即使在没有并发控制机制监控的情况下执行，也不会破坏系统的一致状态。并发控制机制带来代码执行的开销和可能的事务延迟，采用开销较小的机制是我们所希望的。减小开销面临的困难是我们事先并不知道哪些事务将陷入冲突中。为获得这种知识，我们需要一种监控系统的机制。

686
687

有效性检查协议(validation protocol)要求每个事务 T_i 在其生命周期中按两个或三个阶段执行。这取决于该事务是一个只读事务还是一个更新事务。这些阶段顺序地列在下面。

1. **读阶段**(read phase)：在这一阶段中，系统执行事务 T_i。各数据项值被读入并保存在事务 T_i 的局部变量中。所有 **write** 操作都是对局部临时变量进行的，并不对数据库进行真正的更新。

2. **有效性检查阶段**(validation phase)：对事务 T_i 进行有效性测试(下面将会介绍)，判定是否可以执行 **write** 操作而不违反可串行性。如果事务有效性测试失败，则系统终止这个事务。

3. **写阶段**(write phase)：若事务 T_i 已通过有效性检查(第 2 步)，则保存 T_i 任何写操作结果的临时局部变量值被复制到数据库中。只读事务忽略这个阶段。

每个事务必须按以上顺序经历这些阶段。然而，并发执行的事务的三个阶段可以是交叉执行的。

为进行有效性检测，我们需要知道事务 T_i 的各个阶段何时进行。为此，我们将三个不同的时间戳与事务 T_i 相关联。

1. **Start**(T_i)：事务 T_i 开始执行的时间。

2. **Validation**(T_i)：事务 T_i 完成该阶段并开始其有效性检查的时间。

3. **Finish**(T_i)：事务 T_i 完成写阶段的时间。

我们利用时间戳 Validation(T_i)的值，通过时间戳排序技术决定可串行性顺序。因此，值 TS(T_i) = Validation(T_i)，并且若 TS(T_j) < TS(T_k)，则产生的任何调度必须等价于事务 T_j 出现在 T_k 之前的某个串行调度。我们选择 Validation(T_i)而不是 Start(T_i)作为事务 T_i 的时间戳是因为在冲突频度很低的情况

下期望有更快的响应时间。

事务 T_i 的**有效性测试**(validation test)要求任何满足 $TS(T_j) < TS(T_i)$ 的事务 T_j 必须满足下面两条件之一：

1. $Finish(T_j) < Start(T_i)$。因为 T_j 在 T_i 开始之前完成其执行，所以可串行性次序得到了保证。

2. T_i 所写的数据项集与 T_j 所读数据项集不相交，并且 T_j 的写阶段在 T_i 开始其有效性检查阶段之前完成 ($Start(T_i) < Finish(T_j) < Validation(T_i)$)。这个条件保证 T_j 与 T_i 的写不重叠。因为 T_j 的写不影响 T_i 的读，又因为 T_i 不可能影响 T_j 的读，从而保证了可串行性次序。 [688]

作为例子，我们再一次考虑事务 T_{25} 与事务 T_{26}。假设 $TS(T_{25}) < TS(T_{26})$，则有效性检查阶段成功地产生调度 6，如图 15-19 所示。注意，只有在 T_{25} 的有效性检查阶段之后才写实际的变量。因此，T_{26} 读取 B 与 A 的旧值，且该调度是可串行化的。



图 15-19 调度 6，采用有效性检查得到的一个调度

有效性检查机制自动预防级联回滚，因为只有发出写操作的事务提交后实际的写才发生。然而，存在长事务饿死的可能，原因是一系列冲突的短事务引起长事务反复重启。为了避免饿死，与之冲突的事务应当暂时阻塞，以使该长事务能够完成。

在有效性检查机制中，由于事务乐观地执行，假定它们能够完成执行并且最终有效，因此也称为**乐观的并发控制**(optimistic concurrency control)机制。与之相反，封锁和时间戳排序是悲观的，因为当它们检测到一个冲突时，它们强迫事务等待或回滚，即使该调度有可能是冲突可串行化的。

15.6 多版本机制

目前为止讲述的并发控制机制要么延迟一项操作要么中止发出该操作的事务来保证可串行性。例如，**read** 操作可能由于相应值还未写入而延迟；或因为它要读取的值已被覆盖而被拒绝执行(即，发出 **read** 操作的事务必须中止)。如果每一数据项的旧值拷贝保存在系统中，这些问题就可以得免。

在**多版本并发控制**(multiversion concurrency control)机制中，每个 **write**(Q)操作创建 Q 的一个新**版本**(version)。当事务发出一个 **read**(Q)操作时，并发控制管理器选择 Q 的一个版本进行读取。并发控制机制必须保证用于读取的版本的选择能保持可串行性。由于性能原因，一个事务能容易而快速地判定读取哪个版本的数据项也是很关键的。 [689]

15.6.1 多版本时间戳排序

时间戳排序协议可以扩展为多版本的协议。对于系统中的每个事务 T_i，我们将一个唯一的静态时间戳与之关联，记为 $TS(T_i)$。如 15.4 节所述，数据库系统在事务开始前赋予该时间戳。

对于每个数据项 Q，有一个版本序列 $<Q_1, Q_2, \cdots Q_m>$ 与之关联。每个版本 Q_k 包含三个数据字段：

- **Content** 是 Q_k 版本的值。
- **W-timestamp**(Q)是创建 Q_k 版本的事务的时间戳。
- **R-timestamp**(Q)：所有成功地读取 Q_k 版本的事务的最大时间戳。

事务(如 T_i)通过发出 **write**(Q)操作创建数据项 Q 的一个新版本 Q_k，版本的 content 字段保存事务 T_i 写入的值，系统将 W-timestamp 与 R-timestamp 初始化为 $TS(T_i)$。每当事务 T_j 读取 Q_k 的值且 R-timestamp(Q_k) < $TS(T_j)$时，R-timestamp 的值就更新。

下面展示的**多版本时间戳排序机制**(multiversion timestamp-ordering scheme)保证可串行性。该机制运作如下：假设事务 T_i 发出 **read**(Q)或 **write**(Q)操作。令 Q_k 表示 Q 满足如下条件的版本，其写时间戳是小于或等于 $TS(T_i)$的最大写时间戳。

1. 如果事务 T_i 发出 **read**(Q)，则返回值是 Q_k 的内容。

2. 如果事务 T_i 发出 **write**(Q)，且若 $TS(T_i) <$ R-timestamp(Q_k)，则系统回滚事务 T_i。另一方面，

若 TS(T_i) < W-timestamp(Q_k)，则系统覆盖 Q_k 的内容；否则（若 TS(T_i) > R-timestamp(Q_k)），创建 Q 的一个新版本。

规则 1 的理由是显然的。一个事务读取位于其前的最近版本。第二条规则规定当一个事务执行写操作"太迟"时将强迫中止，更确切地说，如果 T_i 试图写入其他事务应该已读取了的版本，则我们不允许该写操作成功。

690 不再需要的版本根据以下规则删除：假设有某数据项的两个版本 Q_k 与 Q_j，这两个版本的 W-timestamp 都小于系统中最老的事务的时间戳，那么 Q_k 和 Q_j 中较旧的版本将不会再用到，因而可以删除。

多版本时间戳排序机制有一个很好的特性：读请求从不失败且不必等待。在典型的数据库系统中，读操作比写操作频繁，因而这个优点对于实践来说至关重要。

然而这个机制也存在两个不好的特性。首先，读取数据项需求更新 R-timestamp 字段，于是产生两次潜在的磁盘访问而不是一次。其次，事务间的冲突通过回滚而不是等待来解决。这种做法开销可能很大。15.6.2 节讲述一个可以减轻这个问题的算法。

多版本时间戳排序机制不保证可恢复性和无级联性。按照与基本的时间戳排序机制一样的方法进行扩充，我们可以使之成为可恢复的和无级联的。

15.6.2 多版本两阶段封锁

多版本两阶段封锁协议（multiversion two-phase locking protocol）希望将多版本并发控制的优点与两阶段封锁的优点结合起来。该协议对**只读事务**（read-only transaction）与**更新事务**（update transaction）加以区分。

更新事务执行强两阶段封锁协议；即它们持有全部锁直到事务结束。因此，它们可以按提交的次序进行串行化。数据项的每个版本有一个时间戳，这种时间戳不是真正基于时钟的时间戳，而是一个计数器，我们称之为 **ts-counter**，这个计数器在提交处理中增加计数。

只读事务在开始执行前，数据库系统读取 **ts-counter** 的当前值来作为该事务的时间戳。只读事务在执行读操作时遵从多版本时间戳排序协议。因此，当只读事务 T_i 发出 **read**(Q)时，返回值是小于 TS(T_i)的最大时间戳的版本的内容。

当更新事务读取一个数据项时，它在获得该数据项上的共享锁后读取该数据项最新版本的值。当更新事务想写一个数据项时，它首先要获得该数据项上的排他锁，然后为此数据项创建一个新版本。写操作在新版本上进行，新版本的时间戳最初置为∞，它大于任何可能的时间戳值。

当更新事务 T_i 完成其任务后，它按如下方式进行提交：首先，T_i 将它创建的每一版本的时间戳设置为 **ts-counter** 的值加 1；然后，T_i 将 **ts-counter** 增加 1。在同一时间内只允许有一个更新事务进行提交。

这样，在 T_i 增加了 **ts-counter** 之后启动的只读事务将看到 T_i 更新的值，而那些在 T_i 增加 ts-counter
691 之前就启动的只读事务将看到 T_i 更新之前的值。无论哪种情况，只读事务均不必等待加锁。多版本两阶段封锁也保证调度是可恢复的和无级联的。

版本删除类似于多版本时间戳排序中采用的方式。假设有某数据项的两个版本 Q_k 与 Q_j，两个版本的时间戳都小于系统中最老的只读事务的时间戳，则两个版本中较旧的将不会再使用，可以删除。

15.7 快照隔离

快照隔离是一种特殊的并发控制机制，并且在商业和开源系统中广泛接受，包括 Oracle、PostgreSQL 和 SQL Server。14.9.3 节介绍了快照隔离。这儿将更详细地观察它如何工作。

从概念上讲，快照隔离在事务开始执行时给它数据库的一份"快照"。然后，事务在该快照上操作，和其他并发事务完全隔离开。快照中的数据值仅包括已经提交的事务所写的值。这种隔离对于只读事务来说是理想的，因为它们不用等待，也不会被并发管理器中止。当然，更新数据库的事务在将更新写入数据库之前，必须处理与其他并发更新的事务之间存在的潜在冲突。更新操作发生在事务的

私有工作空间中，直到事务成功提交，此时更新写入数据库。当允许事务 T 提交时，事务 T 变为提交状态，并且 T 对数据库的所有写操作都必须作为一个原子操作执行，以保证其他事务的快照要么包括 T 的所有更新，要么任何都不包括。

15.7.1 更新事务的有效性检验步骤

决定是否允许一个更新事务的提交需要小心。潜在地，两个并发执行的事务可能会更新同一个数据项。由于两个事务的操作用它们各自的私有快照隔离，因此任何事务都看不到对方所做的更新。如果两个事务都允许写入数据库，第一个更新的写入将被第二个覆盖。结果导致一个**更新丢失**(lost update)。显然地，这必须预防。快照隔离有两个变种，都防止了更新丢失。它们称为**先提交者获胜**和**先更新者获胜**。两种方法都基于检查事务的并发事务。一个事务称为与 T **并发**(concurrent with T)，如果在从 T 开始到执行这个检查之间的任何时刻该事务是活跃的或者部分提交的。

按照**先提交者获胜**(first committer wins)方法，当事务 T 进入部分提交状态，以下操作作为一个原子操作执行： 692

- 检查是否有与 T 并发执行的事务，对于 T 打算写入的某些数据，该事务已经将更新写入数据库。
- 如果发现这样的事务，则 T 中止。
- 如果没有发现这样的事务，则 T 提交，并且将更新写入数据库。

这种方法称为"先提交者获胜"，因为如果事务发生冲突，第一个使用上述规则检查的事务成功地写入更新，而随后的事务则被迫中止。有关如何实现以上检查的细节在习题 15.19 中讨论。

按照**先更新者获胜**(first updater wins)方法，系统采用一种仅用于更新操作的锁机制(读操作不受此影响，因为它们不获得锁)。当事务 T_i 试图更新一个数据项时，它请求该数据项的一个写锁。如果没有另一个并发事务持有该锁，则获得锁后执行以下步骤：

- 如果这个数据项已经被任何并发事务更新，则 T_i 中止。
- 否则 T_i 能够执行其操作，可能包括提交。

然而，如果另一个并发事务 T_j 已经持有该数据项的写锁，则 T_i 不能执行，并且执行以下规则：

- T_i 等待直到 T_j 中止或提交。
  - 如果 T_j 中止，则锁被释放并且 T_i 可以获得锁。当获得锁后，执行前面描述的对于并发事务的更新检查：如果有另一个并发事务更新过该数据项，则 T_i 中止；否则，执行其操作。
  - 如果 T_j 提交，则 T_i 必须中止。

当事务提交或中止时，锁被释放。

这种方法称为"先更新者获胜"，因为如果事务发生冲突，允许第一个获得锁的事务提交并完成其更新。其后试图更新的事务中止，除非第一个更新者随后由于其他原因中止。(除了等待看第一个事务 T_j 是否中止，其后的更新者 T_i 可以在发现它所想要的写锁被 T_j 持有时立刻中止。)

15.7.2 串行化问题

快照隔离在实践中很有吸引力，因为其开销低并且没有中止发生，除非两个并发的事务更新同一个数据项。

然而，我们前面所给出的，以及在实践中实现的快照隔离机制存在一个严重的问题：**快照隔离不能保证可串行化**。即使在将快照隔离作为串行化隔离级别实现的 Oracle 中也是如此！下面举例说明快照隔离下可能的非可串行化执行，并且说明如何解决。 693

1. 假设有两个并发事务 T_i 和 T_j，以及两个数据项 A 和 B。假设 T_i 读取 A 和 B，然后更新 B，而 T_j 读取 A 和 B，然后更新 A。简单起见，假设没有其他并发事务。由于 T_i 和 T_j 是并发的，因此任何事务在其快照内不能看到对方的更新。但是，因为它们更新的是不同数据项，不管系统采用先提交者获胜或者先更新者获胜，两者都可以提交。

然而，优先图有一个环。优先图中从 T_i 到 T_j 有一条边，因为 T_i 在 T_j 写 A 之前读取 A 的值。同样，优先图中从 T_j 到 T_i 也有一条边，因为 T_j 在 T_i 写 B 之前读取 B 的值。由于优先图中存在环，因此结果

是非可串行化的。

一对事务中的每一个都读取对方写的数据，但是不存在两者同时写的数据，这种情况称为写偏斜(write skew)。举一个写偏斜的具体例子，考虑一个银行场景。假设银行通过完整性约束限制一个客户的支票账户和储蓄账户的余额之和不能为负。假设一个客户的支票账户余额和储蓄账户余额分别为100美元和200美元。假设事务 T_{38} 读这两个余额，进行完整性约束核实，从支票账户中取出200美元。假设另一个并发执行的事务 T_{39} 也经过完整性约束核实，从储蓄账户中取出200美元。由于每个事务都在其快照上进行完整性约束检验，如果它们同时执行，任何一方都会相信取款后的余额之和为100美元，因此取款并不违反完整性约束。由于两个事务更新不同的数据项，它们之间不存在更新冲突，因此在快照隔离下两者都可以提交。

遗憾的是，在 T_{38} 和 T_{39} 都提交后的状态中，余额之和为-100美元，违反了完整性约束。这种违反不可能出现在 T_{38} 和 T_{39} 的串行执行中。

值得注意的是，由数据库强制执行的完整性约束，例如主键和外键约束，不能在快照上进行检查；否则将可能有两个并发事务同时插入主键相同的记录，或者一个事务插入一个外键值，而另一个并发事务从引用的表中删除该值。相反，数据库系统必须在数据库的当前状态下检查约束，作为提交时有效性检验的一部分。

694 2. 在接下来的例子中，我们考虑两个并发更新事务本身是可串行化的且不存在任何问题，但是当一个只读事务出现在某个时刻时正好会造成问题。

假设有两个并发事务 T_{1} 和 T_{2}，以及两个数据项 A 和 B。假设 T_{1} 读取 B，然后更新 B，而 T_{2} 读取 A 和 B，然后更新 A。同时并发执行这两个事务没有问题。由于 T_{1} 只访问数据项 B，在数据项 A 上没有冲突，因此优先图中没有环。优先图中唯一的边是从 T_{2} 指向 T_{1}，因为 T_{2} 在 T_{1} 写入 B 前读取 B 的值。

然而，假设 T_{1} 提交时 T_{2} 仍然活跃。假设 T_{2} 提交后但 T_{2} 尚未提交时，一个新的只读事务 T_{3} 进入系统，而且 T_{3} 读取 A 和 B。它的快照包括 T_{1} 的更新，由于 T_{1} 已经提交。然而，由于 T_{2} 未提交，它的更新还未写入数据库，因此不包含在 T_{3} 的快照中。

考虑优先图中添加的与 T_{3} 相关的边。优先图中有一条从 T_{2} 到 T_{3} 的边，因为 T_{3} 在 T_{2} 读取 B 之前写入 B 的值。优先图中还有一条从 T_{3} 到 T_{2} 的边，因为 T_{3} 在 T_{2} 写 A 之前读取 A 的值。这导致优先图中出现环，说明调度是非可串行化的。

上述异常可能不像它们乍一看起来那样麻烦。回想串行化的原因是为了确保在事务并发执行时的数据库保持一致性。因为一致性是目标，所以我们可以接受潜在的非可串行化执行，如果我们确信那些可能出现的非可串行化执行不会导致不一致。上述第二个例子只有在提交只读事务(T_{3})的应用关心 A 和 B 的更新出现乱序时才会出现问题。在这个例子中，我们没有指定何个事务必须保持的数据库一致性约束。如果我们是在处理一个金融数据库，T_{3} 读取非可串行的更新时会出现严重问题。而如果 A 和 B 是同一个课程的两次开课的注册数，则 T_{3} 不要求理想的可串行化，并且我们可以从应用中得知更新频率足够低，因此 T_{3} 读取中的不准确也是不重要的。

数据库必须在提交时而非快照上检查完整性约束这一事实也帮助避免在某些情况下的不一致。一些金融应用程序创建连续的序列号。例如在给账单编号时，将新账单号设置为当前最大账单号加1。如果两个事务并发执行，每一方在其快照中都会看到相同的账单集合，每一方都会创建具有相同账单号的新账单。695 两个事务都通过快照隔离的有效性检验，因为它们不更新相同的元组。然而，执行是非串行化的。由此产生的数据库状态无法从两个事务的任何串行执行中得到。创建两个具有相同账单号的账单可能会导致严重的法律后果。

上述问题是一个幻象的例子，15.8.3节将描述它，因为任何一个事务执行的插入操作都会和另一个事务读取最大账单号的读操作冲突，但是这种冲突不能通过快照隔离检测到。[6]

幸运的是，在大多数应用程序中账单号会声明为主键，数据库系统会在快照之外检查主键冲突，

---

[6] SQL标准中的术语幻象问题都非可重复读问题，导致一些人声称快照隔离避免了幻象问题。然而，根据我们对幻象冲突的定义，该声称是无效的。

并且回滚事务中的一方。[⊖]

程序开发人员可以通过将以下 **for update** 子句附加到 SQL 查询语句之后来防止这种快照异常：

```
select *
from instructor
where ID = 22222
for update;
```

添加 **for update** 子句会使系统出于并发控制的目的将读取的数据作为刚被更新来对待。在第一个关于写偏斜的例子中，如果 **for update** 子句附加到查询账户余额的查询语句之后，那么两个并发事务中只有一个允许提交，因为看起来两个事务都对支票账户和储蓄账户余额进行更新。

在第二个非可串行化执行的例子中，如果事务 T_2 的发起者希望避免异常，**for update** 子句可以附加到查询语句之后，尽管事实上没有更新。在该例子中，如果 T_2 采用 **select for update**，则当它读取 A 和 B 时作为对它们的更新。结果将是或者 T_1 或者 T_2 中止，并稍后作为一个新事务重试。这将会得到一个可串行化的执行。在这个例子中，另外两个事务中的查询不需要添加 **for update** 子句，不必要的 **for update** 子句会导致并发度的显著下降。

存在规范的方法（见文献注释）用来判断给定一个事务的混合运行，是否存在快照隔离下非可串行化执行的风险，并且决定引入何种冲突（例如采用 **for update** 子句）来保证可串行化。当然，这些方法只有在我们事先知道要执行什么事务时才有效。在一些应用中，所有事务都是预先确定的事务集中的事务，这使得这种分析成为可能。然而，如果应用允许不受限制的、临时的事务，则这种分析是不可能的。696

在三个广泛应用的支持快照隔离的系统中，SQL Server 提供了一个可串行化隔离级别选项，它能真正保证可串行化，以及一个快照隔离级别选项，它提供快照隔离的性能优势（连同上面讨论的潜在异常）。在 Oracle 和 PostgreSQL 中，可串行化隔离级别仅提供快照隔离。

15.8 插入操作、删除操作与谓词读

目前为止我们一直把注意力集中在 **read** 与 **write** 操作上。这就限制了事务只能处理已存在于数据库中的数据项。一些事务不仅要求访问已存在的数据项，而且要求创建新的数据项；还有一些事务要求能删除数据项。为考察这样的事务如何影响并发控制，我们引入如下操作：

- **delete**(Q)：从数据库中删除数据项 Q。
- **insert**(Q)：插入一个新的数据项 Q 到数据库中并赋予 Q 一个初值。

事务 T_i 试图在 Q 被删除后执行 **read**(Q)操作将导致 T_i 中的逻辑错误。类似地，事务 T_i 在 Q 被插入之前执行 **read**(Q)操作也会导致 T_i 中的逻辑错误。试图删除并不存在的数据项也是一个逻辑错误。

15.8.1 删除

要理解 **delete** 指令怎样影响并发控制，我们必须弄清 **delete** 指令何时与另一指令发生冲突。令 I_i 与 I_j 分别是 T_i 与 T_j 的指令，它们连续地出现于调度 S 中。令 I_i = **delete**(Q)。我们考虑几条指令 I_j：

- I_j = **read**(Q)：I_i 与 I_j 冲突。如果 I_i 出现在 I_j 之前，事务 T_j 将出现逻辑错误。如果 I_j 出现在 I_i 之前，事务 T_j 可以成功执行 **read** 操作。
- I_j = **write**(Q)：I_i 与 I_j 冲突。如果 I_i 出现在 I_j 之前，事务 T_j 将出现逻辑错误。如果 I_j 出现在 I_i 之前，事务 T_j 可以成功执行 **write** 操作。
- I_j = **delete**(Q)：I_i 与 I_j 冲突。如果 I_i 出现在 I_j 之前，事务 T_j 将出现逻辑错误。如果 I_j 出现在 I_i 之前，事务 T_i 将出现逻辑错误。
- I_j = **insert**(Q)：I_i 与 I_j 冲突。假设数据项 Q 在执行 I_i 与 I_j 之前不存在。那么，若 I_i 出现在 I_j 之前，事务 T_j 将出现逻辑错误。如果 I_j 出现在 I_i 之前，则没有逻辑错误。类似地，如果 Q 在执行 I_i 与 I_j 之前已存在，则如果 I_j 出现在 I_i 之前会出现逻辑错误。反过来则不会。

⊖ 账单号重复问题在 I. I. T. Bombay 金融应用中实际发生过几次，其账单号不是作为主键（原因过于复杂，不在此讨论），问题是由财务审计员发现的。

697 我们可以总结如下：

- 在两阶段封锁协议下，一数据项可以删除之前，在该数据项上必须请求加排他锁。
- 在时间戳排序协议下，必须执行类似于为 **write** 操作进行的测试。假设事务 T_i 发出 **delete**(Q)：
  - 如果 TS(T_i) < R-timestamp(Q)，则 T_i 将要删除的 Q 值已被满足 TS(T_j) > TS(T_i)的事务 T_j 读取。因此，**delete** 操作被拒绝，T_i 回滚。
  - 如果 TS(T_i) < W-timestamp(Q)，则满足 TS(T_j) > TS(T_i)的事务 T_j 已经写过 Q。因此，**delete** 操作被拒绝，T_i 回滚。
  - 否则，执行 **delete** 操作。

15.8.2 插入

我们已经看到 **insert**(Q)操作与 **delete**(Q)操作冲突。类似地，**insert**(Q)操作与 read(Q)操作或 **write**(Q)操作冲突。在数据项存在之前不能进行 **read** 或 **write** 操作。

由于 **insert**(Q)给数据项 Q 赋值，在并发控制中应当类似于处理 **write** 操作那样处理 **insert** 操作：

- 在两阶段封锁协议下，如果 T_i 执行 **insert**(Q)操作，就在新创建的数据项 Q 上赋予 T_i 排他锁。
- 在时间戳排序协议下，如果 T_i 执行 **insert**(Q)操作，就把 R-timestamp(Q)与 W-timestamp(Q)的值设置成 TS(T_i)。

15.8.3 谓词读和幻象现象

考虑事务 T_{29}，它在大学数据库上执行以下 SQL 查询：

```
select count(*)
from instructor
where dept_name = 'Physics';
```

事务 T_{29}需要访问 *instructor* 关系中与 Physics 系相关的所有元组。

考虑事务 T_{30}，它执行以下 SQL 插入：

```
insert into instructor
    values(11111, 'Feynman', 'Physics', 94000);
```

698 令 S 是包含事务 T_{29}与 T_{30}的调度。由于下面的原因，我们预计冲突是可能存在的：

- 如果 T_{29}在计算 **count**(*)时使用 T_{30}新近插入的元组，则 T_{29}读取 T_{30}写入的值。因此，在等价于 S 的串行调度中，T_{30}必须先于 T_{29}。
- 如果 T_{29}在计算 **count**(*)时未使用 T_{30}新近插入的元组，则在等价于 S 的串行调度中，T_{29}必须先于 T_{30}。

这两种情况中的第二种让人感到奇怪。T_{29}与 T_{30}没有访问共同的元组，但它们相互冲突！事实上，T_{29}与 T_{30}在一个幻象元组上发生冲突。如果并发控制在元组级粒度上进行，该冲突将难以发现。结果是，系统也许不能防止出现非可串行化调度。我们把这种现象称为**幻象现象**(phantom phenomenon)。

除了幻象问题，我们还需要处理 14.10 节中我们看到的情况，一个事务利用索引查询 *dept_name* = "Physics"的元组，因此它不读取其他系的任何元组。如果另一个事务更新这些元组中的一个，把它的系名称改为 Physics，一个相当于幻象问题的问题则会出现。这些问题都是由谓词读导致的，并且有共同的解决方法。

为防止幻象现象，我们允许 T_{29}阻止其他事务在关系 *instructor* 上创建 *dept_name* = "Physics"的新元组，并且阻止将一个已有的 *instructor* 元组的系名更新为 Physics。

为找到关系 *instructor* 中满足条件 *dept_name* = "Physics"的所有元组，T_{29}必须搜索整个 *instructor* 关系，或至少搜索关系上的一个索引。到现在为止，我们隐含地假设事务所访问的数据项仅仅是元组。然而，事务 T_{29}是一个读取关系中有哪些元组这一信息的事务的例子，而事务 T_{30}则是更新这种信息的事务的例子。

显然，仅仅封锁所访问的元组是不够的；用来找出事务访问的元组的信息也需要封锁。

封锁用来找出要访问的元组的信息可以通过将一个数据项与关系关联在一起来实现；该数据项代

表一种信息，事务用它来查找关系中的元组。读取关系中有哪些元组这一信息的事务，如事务 T_{32}，必须以共享模式封锁对应于该关系的数据项。更新关系中有哪些元组这一信息的事务，如事务 T_{31}，必须以排他模式封锁对应于该关系的数据项上。这样，事务 T_{32} 与 T_{31} 将在一个真实的数据项上发生冲突，而不是在幻象上发生冲突。类似地，使用索引来检索元组的事务必须封锁索引本身。

不要将多粒度封锁中对整个关系的封锁与这儿讲的对对应于关系的数据项的封锁相混淆。通过封锁该数据项，一个事务只是阻止其他事务对关系中有哪些元组这一信息的修改。对元组的封锁仍是需要的。直接访问某个元组的事务可以被授予该元组上的锁，即使另一个事务在对应于关系本身的数据项上持有排他锁。 699

封锁对应于关系的一个数据项或者封锁整个索引的主要缺点是并发程度低——往关系中插入不同元组的两个事务也不能并发执行。

较好的解决方法是使用索引封锁(index-locking)技术，它避免封锁整个索引。在关系中插入元组的任何事务必须在该关系上维护的每一个索引中插入有关信息。通过使用索引的封锁协议，我们可以消除幻象现象。为简单起见，我们只考虑 B^+ 树索引。

正如我们在 11 章中看见的那样，每一个搜索码值与索引中的一个叶结点相关联。查询通常使用一个或多个索引来访问关系。插入操作必须在关系的所有索引中插入新元组。在该例子中，假定在关系 *instructor* 的 *dept_name* 上建立了索引。那么，事务 T_{31} 必须修改包含码值"Physics"的叶结点。如果 T_{32} 读取同一叶结点，查找有关 Physics 系的所有元组，则 T_{31} 与 T_{32} 在该叶结点上发生冲突。

通过将幻象现象实例转换为对索引叶结点上封锁的冲突，**索引封锁协议**(index-locking protocol)利用了关系上索引的可用性。该协议运作如下：

- 每个关系至少有一个索引。
- 只有首先在关系的一个或多个索引上找到元组后，事务 T_i 才能访问关系上的这些元组。为了达到索引封锁协议的目的，全表扫描看作一个索引上所有叶结点的扫描。
- 进行查找(不管是区间查找还是点查找)的事务 T_i 必须在它要访问的所有索引叶结点上获得共享锁。
- 在没有更新关系 r 上的所有索引之前，事务 T_i 不能插入、删除或更新关系 r 中的元组 t_i。该事务必须获得插入、删除或更新所影响的所有索引叶结点上的排他锁。对于插入与删除，受影响的叶结点是那些(插入后)包含或(删除前)包含元组搜索码值的叶结点。对于更新，受影响的叶结点是那些(修改前)包含搜索码旧值的叶结点，以及(修改后)包含搜索码新值的叶结点。
- 元组照常获得锁。
- 必须遵循两阶段封锁协议规则。

请注意，索引封锁协议并不关注索引内部结点的并发控制问题。最小化锁冲突的索引并发控制技术将在 15.10 节介绍。 700

封锁一个索引叶结点阻止了该结点上的任何更新，即使这个更新实际上并没有与谓词冲突。一个称为码值封锁的变种将在 15.10 节作为索引并发控制的一部分介绍，它可以使这样的假的锁冲突最小化。

如 14.10 节所指出的，事务之间的冲突看起来取决于低级别的系统查询处理决策，而与用户级别的事务含义无关。另一种并发控制的方法允许给查询中的谓词加共享锁，如关系 *instructor* 上的谓词"*salary* > 90000"。关系上的插入和删除操作都要检查是否满足谓词。若满足，则存在锁冲突，插入和删除操作要等待直到谓词锁被释放。对于更新操作，元组的初始值和最终值都要检查是否满足谓词。这些冲突的插入、删除和更新操作影响谓词选中的元组集合，因此不能允许它们与已获得(共享的)谓词锁的查询并发执行。我们称以上的协议为**谓词锁**(predicate locking)。[⊖]谓词锁在实践中不采用，因为相比较于索引封锁机制，它的实现代价很大，但又不能带来显著的额外好处。

⊖ 术语谓词锁用于对谓词使用共享锁和排他锁的协议的一个版本，因此更加复杂。我们在此给出的版本仅含有在谓词上的共享锁，因此也叫做**精确锁**(precision locking)。

谓词锁技术存在一些变种，可以用来消除本章中其他并发控制协议下的幻象现象。然而，许多数据库，如 PostgreSQL（版本 8.1）和 Oracle（版本 10g）（就我们所知）都没有实现索引封锁和谓词锁，而且即使将隔离性级别设置为可串行化，也很容易由于幻象问题导致非可串行性。

15.9 实践中的弱一致性级别

14.5 节讨论了 SQL 标准中的隔离级别：可串行化、可重复读、已提交读和未提交读。本节首先简要介绍一些比可串行化级别弱的一致性级别的数老技术，并将它们与 SQL 标准级别相关联。然后我们将讨论在 14.8 节中已简要讨论过的涉及用户交互的事务并发控制问题。

15.9.1 二级一致性

二级一致性（degree two consistency）的目的是在不必保证可串行性的前提下防止发生级联中止。二级一致性的封锁协议采用两阶段封锁协议中我们用过的同样的两类锁，共享锁（S）和排他锁（X）。事务必须持有适当的锁才能访问数据项，但是可阶段动作不是必需的。

与两阶段封锁的情况大不相同，这里共享锁可以在任何时候释放，并且锁可以在任何时间获得，
[701] 而排他锁只有在事务提交或中止后才能释放。该协议不保证可串行性。实际上，事务有可能两次读取同一数据项却得到不同结果，如在图 15-20 中，T_{32} 在 T_{33} 写 Q 值之前和之后读取 Q 值。

显然，读取不是可重复读，但是由于直到事务提交前一直持有排他锁，因此没有事务可以读取未提交的值。因此，二级一致性是已提交读隔离性水平的一种特殊实现。

T_{32}	T_{33}
lock-S(Q)	
read(Q)	
unlock(Q)	
	lock-X(Q)
	read(Q)
	write(Q)
	unlock(Q)
lock-S(Q)	
read(Q)	
unlock(Q)	

图 15-20 具有二级一致性的非可串行化调度

15.9.2 游标稳定性

游标稳定性（cursor stability）是二级一致性的一种形式，它是为利用游标对关系中的元组进行迭代的程序而设计的。它不封锁整个关系，游标稳定性保证：

- 正被迭代处理的元组被加上共享锁。
- 任何被更改的元组被加上排他锁，直至事务提交。

这些规则保证了二级一致性，不要求两阶段封锁，没有保证同度的可串行性。实践中游标稳定性用于频繁访问的关系，以作为一种提高并发性和改善系统性能的方法。无论是否存在非可串行化的调度，利用游标稳定性的应用程序必须在编码上确保数据库的一致性。所以，游标稳定性的用途局限于某些专门的并且具有简单一致性约束的情况。

15.9.3 跨越用户交互的并发控制

[702] 并发控制协议通常考虑的是不涉及用户交互的事务。现在考虑 14.8 节中涉及用户交互的航空公司座位选择的例子。假设我们从初始时将座位空闲情况显示给用户，直到确定座位选择的所有步骤都作一个事务。

如果使用两阶段封锁协议，飞机上所有座位都被加上共享锁，直到用户选择完座位。在这期间其他事务不允许更新座位分配情况。显然，这种锁是非常糟糕的，因为用户可能需要很长的时间来做出选择，甚至放弃交易但不显式地取消。可以采用时间戳协议或者有效性检验来代替，避免加锁出现的问题，但是这两种协议会在另一个用户 B 更新座位分配信息时中止用户 A 的事务，即使用户 B 选择的座位和用户 A 选择的座位并不冲突。快照隔离是在这种情况下最好的选择，因为只要用户 B 没有选择和用户 A 相同的座位，A 的事务就不会中止。

然而，快照隔离要求数据库记录一个事务的更新信息，即使该事务已经提交，但只要任何其他并发的事务仍然是活跃的，这对于长事务会存在问题。

另一种方法是将涉及用户交互的事务划分成两个或者更多的事务，使得没有事务跨越用户交互。如果我们的座位选择事务按照这样进行划分，则第一个事务将读取空闲座位，第二个事务将完成分配选择的座位。如果第二个事务编写不慎，会出现在分配座位给用户的时候，没有检查该座位是否同时分配给其他用户，导致更新丢失的问题。为避免这个问题，正如 14.8 节所述，第二个事务只有在座位

没有同时分配给其他用户时才能将其分配。

上述思想在另一种并发控制机制中被优化了，该机制使用存储在元组中的版本号来避免更新丢失。每个关系模式中添加一个额外的版本号属性。当创建元组时初始化为0。当一个事务（第一次）读取它试图更新的一个元组时，它记录该元组的版本号。读操作作为一个独立的数据库事务执行，因此任何所获得的锁被立即释放。更新操作在本地完成，并作为提交过程的一部分将页到数据库中。采用作为原子执行的以下步骤（即作为数据库事务的一部分）：

- 对于每个更新的元组，事务检查当前的版本号是否等于第一次读取事务时的版本号。
  1. 如果版本号匹配，则执行数据库中该元组的更新，并且它的版本号加1。
  2. 如果版本号不匹配，事务中止，回滚它所执行的所有更新。 705

如果对于所有更新元组的版本号检查成功，事务提交。简单一些的时间戳可以用来代替版本号，而对机制没有任何影响。

注意到上述机制和快照隔离的相似性。版本号检查实现了快照隔离中的先提交者获胜规则，而且在事务活跃时间很长的情况下也可使用。然而，与快照隔离不同，事务的读操作可能不对应数据库中的一个快照。与有效性检验协议不同，事务的读操作不进行检验。

我们将以上的机制称为**不做读有效性检查的乐观并发机制**（optimistic concurrency control without read validation）。不做读有效性检验的乐观并发控制机制提供了一种弱的串行化水平，并不保证可串行化。该机制的一个变种是在提交的时候，除了检验写操作外，还采用版本号来检验读操作，以保证事务读取的元组在初次读取之后没有更新。这种机制即我们前面看到的乐观并发控制机制。

上述机制已经被应用程序开发人员广泛地用在涉及用户交互的事务处理中。该机制的一个诱人的特点是它可以轻松地实现在数据库系统顶层。作为提交过程的一部分，检验和更新操作作为一个单独的数据库事务执行，并采用数据库的并发控制机制保持提交过程的原子性。以上机制同样用在Hibernate对象关系映射系统（见9.4.2节）和其他对象关系映射系统中，并称为乐观并发控制（即使取消读操作不检验）。在Hibernate中涉及用户交互的事务称为**对话**（conversation），以区分它们和采用版本号的常规事务检验。对象关系映射系统主将数据库元组在内存中以对象的形式进行缓存，在缓存的对象上执行事务。对象上的更新在事务提交时转化为数据库上的更新。数据可能会在缓存中存在很长的时间，如果事务更新缓存的数据，则会有更新丢失的风险。因此，Hibernate和其他关系对象映射系统采用透明的版本号检查作为提交过程的一部分。（如果需要可串行化，Hibernate允许程序员绕开缓存，直接在数据库上执行事务。）

15.10 索引结构中的并发*

对索引结构访问的处理可以像处理访问其他数据库结构那样进行，并应用前面讲述过的并发控制技术。然而，由于索引访问频繁，它们将成为封锁竞争的集中点，从而导致低并发度。幸运的是，索引不必像其他数据库结构那样处理。事务在两次索引查找期间，发现索引结构发生了变化，这是完全可以接受的，只要索引查找返回正确的元组集。因此，只要维护索引的准确性，对索引运行非可串行 704
化并发存取是可接受的。

我们讲述两种并发访问B+树的技术。关于处理B+树的其他技术，以及处理其他索引结构的技术可参阅的相关文献请参见文献注解。

我们所讲述的处理B+树的技术基于封锁机制，但既不采用两阶段封锁也不采用树形协议。用于查找、插入与删除的算法是第11章中使用的算法，只是做了小小的修改。

第一种技术叫作**蟹行协议**（crabbing protocol）：

- 当查找一个码值时，蟹行协议首先用共享模式锁住根结点。沿树向下遍历，它在子结点上获得一个共享锁，以便向更远处遍历，在子结点上获得锁以后，它释放父结点上的锁。它重复该过程直至叶结点。
- 当插入或删除一个码值时，蟹行协议采取如下行动：
  1. 采取与查找相同的协议直至希望的叶结点，到此为止，它只获得（和释放）共享锁。

- 它用排他锁封锁该叶结点，并且插入或删除码值。
- 如果需要分裂一个结点或将它与兄弟结点合并，或者在兄弟结点之间重新分配码值，就行协议用排他锁封锁父结点，在完成这些操作后，它释放该结点和兄弟结点上的锁。

  如果父结点需要分裂、合并或重新分布码值，该协议保留父结点上的锁，以同样方式分裂、合并或重新分布码值，并且传播更远。否则，该协议释放父结点上的锁。

该协议的名字来源于螃蟹走路的方式：先移动一边的腿，然后另一边的，如此交替进行。该协议的封锁过程，从上往下和从下往上(发生分裂、合并或重新分布的情况)，就像螃蟹移动一样。

一旦某个操作释放了一个结点上的锁，别的操作就可以访问该结点。在向下的搜索操作和由于分裂、合并或重新分布传播向上的操作之间有可能出现死锁，系统能很容易地处理这种死锁，它先让搜索操作释放锁，然后从树根重启它。

705 第二种技术使用一个改进版本的 B^+ 树，称为 **B-link 树**(B-link tree)，避免了在获取另一个结点的锁时还占有一个结点的锁，获得更多的并发性。B-link 树要求每个结点(包括内部结点，不仅仅是叶结点)维护一个指向右兄弟结点的指针，这个指针是必要的，因为在一个结点正在分裂时进行的查找可能不仅要查找该结点而且可能要查找该结点的右兄弟结点(如果存在的话)。我们以后将用一个例子来说明这个技术。但首先我们列出修改后的 **B-link 树封锁协议**(B-link-tree locking protocol)。

- **查找**：B^+ 树的每个结点在访问之前必须加共享锁。非叶结点上的锁应该在对 B^+ 树的其他任何结点发出加锁请求前释放。如果结点分裂与查找同时发生，所希望的搜索码值可能不再位于查找过程中所访问的某个结点所代表的那些值的范围内。在这种情况下，搜索码值在兄弟结点所代表的范围内，系统循着指向右兄弟结点的指针能找到该兄弟结点。不过，叶结点的封锁遵循两阶段封锁协议以避免幻象现象，如 15.8.3 节所述。
- **插入与删除**：系统遵循查找规则，定位要进行插入或删除的叶结点。该结点的共享锁升级为排他锁，然后进行插入或删除。受插入或删除影响的叶结点封锁遵循两阶段封锁协议，以避免幻象现象，如 15.8.3 节所述。
- **分裂**：如果事务使一个结点分裂，则按 11.3 节的算法创建新结点并作为原结点的右兄弟。设置原结点和新产生结点的右兄弟指针。接着，事务释放原结点的排他锁(假若它是一个内部结点；叶结点以两阶段形式加锁)，然后发出对父结点加排他锁的请求，以便插入指向新结点的指针。(没有必要对新结点加锁或释锁。)
- **合并**：执行删除后，如果一个结点的搜索码值太少，则必须将要与之合并的那个结点加上排他锁。一旦这两个结点合并，就发出对父结点加排他锁的请求，以便删除要被删除的结点。此时，事务释放已合并结点的锁。除非父结点也需再合并，不然释放其锁。

注意这个重要的事实：插入与删除操作可能封锁一个结点，释放该结点，然后又对它重新封锁。此外，与分裂或合并操作并发执行的查找可能发现所需搜索码值被分裂或合并操作移到右兄弟结点。

为说明这一点，考虑图 15-21 所示的 B-link 树。假设有两个针对该 B-link 树的并发操作：

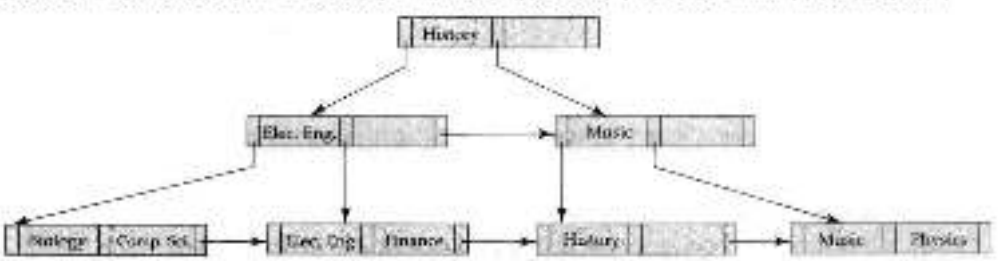


图 15-21 *department* 文件的 n = 3 的 B-link 树

1. 插入"Chemistry"。

706 2. 查找"Comp. Sci."。

我们假设插入操作首先开始，它为"Chemistry"执行查找，发现要插入"Chemistry"的结点已满，于

是它将该结点的共享锁转换为排他锁，并创建新结点。这样，原结点包含搜索码值"Biology"与"Chemistry"，新结点包含搜索码值"Comp. Sci."。

现在假设发生上下文切换，导致控制传递给查找操作。该查找操作访问根结点，然后沿指向根结点左子结点的指针而下。接着访问那个结点，并得到指向左子结点的指针。该左子结点原先包含搜索码值"Biology"与"Comp. Sci."。由于该结点目前被插入操作以排他方式封锁，因此查找操作必须等待。注意，此时查找操作不持有任何锁！

插入操作现在释放了叶结点并重新对父结点加锁，这次以排他方式封锁。它完成了插入操作，得到图15-22所示的B-link树。查找操作继续进行。然而，它拥有的指针指向错误的叶结点。于是它顺着右兄弟结点指针定位下一个结点。如果该结点仍不正确，则继续顺着该结点的右兄弟指针查找。可以证明，如果查找操作拥有指向错误结点的指针，则沿着右兄弟结点指针，查找操作最终可以到达正确的结点。

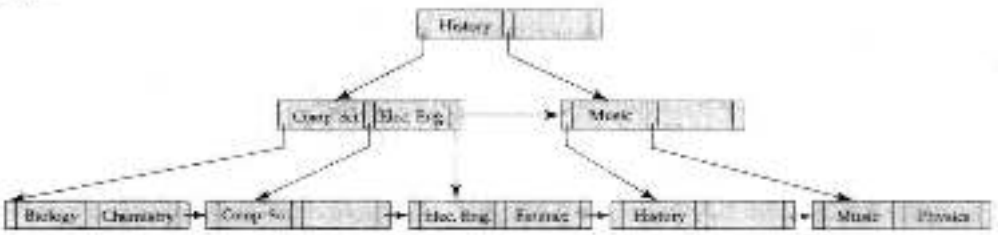


图15-22 插入"Chemistry"到图15-21所示的B-link树中

查找与插入不会引起死锁。删除操作时的结点合并可能引起不一致性，因为查找操作可能在父结点更新前已经从父结点读取了指向被删除结点的指针，然后试图访问被删除的结点。查找操作将必须从根结点重新开始。不对结点进行合并可以避免这样的不一致性。这个解决方案使某些结点包含的搜索码值太少，违背了B+树的某些特性。然而，在大部分数据库中，插入操作比删除操作频繁，因此包含搜索码值太少的结点可能较快地得到更多的值。

一些索引并发控制机制不是以两阶段形式封锁索引叶结点，而是对个别的码值使用**码值封锁**(key-value locking)，允许其他码值从同一个叶结点插入或删除，这样码值封锁提供了增强了的并发性。但是，朴素地使用码值封锁可能引发幻象现象；为防止幻象现象可以采用下一码封锁(next-key locking)技术。在这个技术中，每一次索引查找不仅封锁查找范围内的多个码(或单个码，在点查找时)，而且封锁下一个码值——也就是刚好比范围内最后一个码值大的码值；并且，每一次插入必须不仅封锁要插入的值，而且包括下一个码值。这样，如果一个事务试图插入一个值到另一个事务的索引查找范围之内时，这两个事务将在插入码值的下一个码值上冲突。同样，删除也必须封锁被删除值的下一个码值，来保证检测得到它与别的查询的查找范围的并发冲突。

15.11 总结

- 当多个事务在数据库中并发地执行时，数据的一致性可能不再维持。系统有必要控制各事务之间的相互作用，这是通过称为并发控制机制的多种机制中的一种来实现的。
- 为保证可串行性，我们可以使用多种并发控制机制。所有这些机制要么延迟一个操作，要么中止发出该操作的事务。最常用的机制是各种封锁协议、时间戳排序机制、有效性检查技术与多版本机制。
- 封锁协议是一组规则，这些规则阐明了事务何时对数据库中的数据项进行加锁和解锁。
- 两阶段封锁协议仅在一个事务完全释放任何数据项上的锁时才允许该事务封锁新数据项。该协议保证可串行性，但不能避免死锁。在没有关于数据项访问方式信息的情况下，两阶段封锁协议对于保证可串行性既是必要的又是充分的。
- 严格两阶段封锁协议要求事务持有的所有排他锁必须在事务结束时方可释放，其目的是保证结果调度的可恢复性和无级联性。强两阶段封锁协议要求事务持有的所有锁必须在事务结束时方可释放。
- 基于图的封锁协议对访问数据项的顺序加以限制，从而不需要使用两阶段封锁还能够保证可串行性，

而且又能够保证不会产生死锁。

- 许多种封锁协议都不能防止死锁。一种可以防止死锁的方法是使用数据项的一种顺序，并且按与该顺序一致的次序申请加锁。
- 另一种防止死锁的方法是使用抢占与事务回滚。为控制抢占，我们给每个事务赋予一个唯一时间戳。这些时间戳用于决定事务是等待还是回滚。如果一个事务回滚，它在重启时保持原有时间戳。wound-wait 机制是一个抢占机制。
- 如果没有预防死锁，系统必须用死锁检测与恢复机制来处理它们。为此，系统构造了一个等待图。当且仅当等待图包含环时，系统处于死锁状态。当一个检测算法判定死锁存在，系统必须从死锁中恢复。系统通过回滚一个或多个事务来解除死锁。
- 某些情况下把多个数据项聚为一组，将它们作为聚集数据项来处理，其效果可能更好，这就导致了粒度的多个级别。我们允许各种大小的数据项，并定义数据项的层次，其中小数据项嵌套于大数据项之中。这种层次结构可以图形化地表示为树。封锁按从根结点到叶结点的顺序进行；解锁则按从叶结点到根结点的顺序进行。该协议保证可串行性，但不能避免死锁。
- 时间戳排序机制通过事先在每对事务之间选择一个顺序来保证可串行性。系统中的每个事务对应一个唯一的固定时间戳。事务的时间戳决定了事务的可串行化顺序。这样，如果事务 T_i 的时间戳小于事务 T_j 的时间戳，则该机制保证产生的调度等价于事务 T_i 出现在事务 T_j 之前的一个串行调度。该机制通过回滚违反该次序的事务来保证这一点。
- 在大部分事务是只读事务的情形下，冲突频度较低，这种情况下有效性检查机制是一个适当的并发控制机制。系统中的每个事务对应一个唯一的固定时间戳。串行性次序是由事务的时间戳决定的。在该机制中，事务不会延迟。不过，事务要完成必须通过有效性检查。如果事务未通过有效性检查，则该事务回滚到初始状态。
- 多版本并发控制机制基于在每个事务写数据项时为该数据项创建一个新版本。读操作发出时，系统选择其中的一个版本进行读取。利用时间戳，并发控制机制保证以确保可串行性的方式选取要读取的版本。读操作总能成功。
  - 在多版本时间戳排序中，写操作可能引起事务的回滚。
  - 在多版本两阶段封锁中，写操作可能导致封锁等待或死锁。
- 快照隔离是一种基于有效性检查的多版本并发控制协议，与多版本两阶段封锁协议不同。它不需要将事务声明为只读或更新的。快照隔离不保证可串行化，但是许多数据库系统仍然支持它。
- 仅当要删除元组的事务在该元组上具有排他锁时，**delete** 操作才能够进行。在数据库中插入新元组的事务在该元组上被授予排他锁。
- 插入操作可能导致幻象现象，这时插入操作与查询发生逻辑冲突，尽管两个事务可能没有存取共同的元组。如果封锁仅加在事务访问的元组上，这种冲突就检测不到。关系中用于查找元组的数据需要加锁，索引封锁技术要求对某些索引结点加锁来解决这个问题。所加的这些锁保证所有事务在实际的数据项上发生冲突，而不是在幻象上。
- 弱级别的一致性用于一些应用中，在这些应用中，查询结果的一致性不是至关重要的，而使用可串行性会使查询对事务的处理起反作用。二级一致性是这种弱级别的一致性之一，游标稳定性是二级一致性的一个特例，而且已广泛应用。
- 涉及用户交互的事务并发控制是一个有挑战性的任务。应用程序通常实现一种基于采用元组中存储的版本号来验证写操作的机制。这种机制提供了弱可串行化水平，而且可以实现在应用层，而无需修改数据库。
- 可以为特殊的数据结构开发特殊的并发控制技术。通常，特殊的技术用到 B^+ 树上，以允许较大的并发性。这些技术允许对 B^+ 树进行非可串行化访问，但它们保证 B^+ 树结构是正确的，并保证对数据库本身的存取是可串行化的。

709
|
710

术语回顾

- 并发控制
- 锁类型
  - 共享(S)锁
  - 排他(X)锁
- 锁
  - 相容性
  - 申请
  - 等待
  - 授予
- 死锁
- 饿死
- 封锁协议
- 合法调度
- 两阶段封锁协议
  - 增长阶段
  - 缩减阶段
  - 封锁点
  - 严格两阶段封锁
  - 强两阶段封锁
- 锁转换
  - 升级
  - 降级
- 基于图的协议
  - 树形协议
  - 提交依赖
- 死锁处理
  - 预防
  - 检测
  - 恢复
- 死锁预防
  - 顺序加锁
  - 抢占锁
  - wait-die 机制
  - wound-wait 机制
  - 基于超时的机制
- 死锁检测
  - 等待图
- 死锁恢复
  - 全部回滚
  - 部分回滚
- 多粒度
  - 显式锁
  - 隐式锁
  - 意向锁
- 意向型锁
  - 共享型意向(IS)
  - 排他型意向(IX)
  - 共享排他型意向(SIX)
- 多粒度封锁协议
- 基于时间戳的协议
- 时间戳
  - 系统时钟
  - 逻辑计数器
  - W-timestamp(Q)
  - R-timestamp(Q)
- 时间戳排序协议
  - Thomas 写规则
- 基于有效性检查的协议
  - 读阶段
  - 有效性检查阶段
  - 写阶段
  - 有效性测试
- 多版本并发控制
- 多版本时间戳排序
- 多版本两阶段封锁
  - 只读事务
  - 更新事务
- 快照隔离
  - 更新丢失
  - 先提交者获胜
  - 先更新者获胜
  - 写偏斜
  - select for update
- 插入和删除操作
- 幻象现象
- 索引封锁协议
- 谓词锁
- 弱一致性级别
  - 二级一致性
  - 游标稳定性
- 不做读有效性验证的乐观并发控制
- 对话
- 索引中的并发
  - 蟹行协议
  - B-link 树
  - B-link 树封锁协议
  - 下一码封锁

实践习题

15.1 证明两阶段封锁协议保证冲突可串行化，并且事务可以根据其封锁点串行化。

15.2 考虑下面两个事务：

T_{31}: read(A);
read(B);
if $A=0$ then $B:=B+1$;
write(B);

T_{32}: read(B);
read(A);
if $B=0$ then $A:=A+1$;
write(A).

给事务 T_{31} 与 T_{32} 增加加锁、解锁指令，使它们遵从两阶段封锁协议。这两个事务会引起死锁吗？

15.3 强两阶段封锁协议带来什么好处？它与其他形式的两阶段封锁协议相比有何异同？

15.4 考虑一个按有根树方式组织的数据库。假设我们在每对结点之间插入一个虚结点。证明，如果我们在

711
712

由此构成的新树上遵从树形协议，我们可以得到的并发度比在原始树上遵从树形协议的更高。

15.5 用例子证明：存在在树形封锁协议下可行，而在两阶段封锁协议下不可行的调度，反之亦然。

15.6 考虑以下对树形封锁协议的扩展，它既允许使用共享锁又允许使用排他锁：

- 事务可以是只读事务，在这种情况下它只申请共享锁；也可以是更新事务，此时只申请排他锁。
- 每个事务必须遵从树形协议规则。只读事务可以首先封锁任何数据项，而更新事务必须首先封锁根结点。

证明该协议保证可串行性并能避免死锁。

15.7 考虑以下基于图的封锁协议，它只允许加排他锁，并且在带根有向无环数据图上运作：

- 事务首先可以封锁任何结点。
- 要封锁任何其他的结点，事务必须在该结点的大部分父结点上持有锁。

证明该协议保证可串行性并能避免死锁。

15.8 考虑以下基于图的封锁协议，它只允许加排他锁，并且在带根有向无环数据图上运作：

- 事务首先可以封锁任何结点。
- 要封锁任何其他的结点，事务必须已经访问该结点的所有父结点，并且必须在该结点的一个父结点上持有锁。

证明该协议保证可串行性并能避免死锁。

15.9 在持久化程序设计语言中封锁不是显式进行的。访问对象（或相应页）时必须加锁。大部分现代操作系统允许用户对页面设置访问保护（不许访问、读、写），并且违反存取保护的内存访问将导致违反保护错误（如，参见 UNIX 的 mprotect 命令）。说明访问保护机制在持久化程序设计语言中如何用于页级封锁。

15.10 考虑除 **read** 与 **write** 操作之外还包含原子操作 **increment** 的一个数据库系统。令 V 是数据项 X 的值，操作：

increment(X) by C

左一个原子步骤中将 X 的值设为 $V + C$。如果事务不执行 read(X)，则事务不能获知 X 的值。图 15-23表示三种锁类型的锁相容阵：共享型、排他型和增量型。

a. 证明：如果所有事务按相应的类型封锁它们所访问的数据项，则两阶段封锁保证可串行性。

b. 证明：包含增量型锁可以增加并发度。（提示：在所举的银行例子中，考虑支票的票据交换事务）。

15.11 在时间戳排序中，**W-timestamp**(Q) 表示成功执行 **write**(Q) 的所有事务的最大时间戳。现在，假设我们将之定义为最近成功执行 **write**(Q) 的事务的时间戳。这种措辞上的变化会带来什么不同？阐释你的答案。

	S	X	I
S	true	false	false
X	false	false	false
I	false	false	true

图 15-23 锁相容矩阵。

15.12 采用多粒度封锁机制比采用单封锁粒度的等价系统需要更多或更少的锁。针对每种情况各举一例，并比较所允许的相对并发量。

15.13 考虑 15.5 节基于有效性检查的并发控制机制。证明：若选择 Validation(T_i) 而不是 Start(T_i) 作为事务 T_i 的时间戳，则如果事务间发生冲突的次数确实很低，我们可望有较好的响应时间。

713 - 714

15.14 对于下面的每个协议，说明促使你使用某个协议的实际应用原因以及不使用的原因：

- 两阶段封锁。
- 具有多粒度封锁的两阶段封锁。
- 树形协议。
- 时间戳排序。
- 有效性检查。
- 多版本时间戳排序。
- 多版本两阶段封锁。

15.15 解释为什么使用下述事务执行技术会比仅仅使用严格两阶段封锁能够得到更好的性能：跟基于有效性检查技术中一样，首先执行事务而无须获得任何锁，也不向数据库执行任何写操作。但跟有效性

检查技术中不一样的是它既不检查有效性也不在数据库上执行写操作，而是采用严格两阶段封锁重新运行事务。（提示：考虑磁盘I/O等待。）

15.16 考虑时戳排序协议，以及两个事务，一个执行写两个数据项 p 和 q，另一个执行读这两个数据项。试给出一个调度，使得第一个事务写操作的时间戳测试失败，引起该事务重启，并依次引起另一个事务的级联中止。并说明是怎样导致这两个事务都饿死的。（两个或多个进程执行，但它们都由于与其他进程的交互作用而无法完成它们的任务，这种情形叫做活锁（livelock）。）

15.17 设计一个基于时间戳的能避免幻象现象的协议。

15.18 假设我们采用15.1.5节中的树形协议来管理对B$^+$树的并发访问，由于在影响到根结点的插入中也可能发生分裂，因此看来插入操作在整个操作完成之前都不能释放锁。在什么情况下有可能提前释放锁呢？

15.19 快照隔离采用有效性检验步骤，当事务 T 写数据项之前，检查 T 是否有其他并发事务已经写过该数据项。

a. 一种简单的实现方式对于每个事务采用一个开始时间戳和一个提交时间戳，另外还有一个更新集合来记录事务更新的数据项。解释如何利用事务时间戳和更新集合来实现先提交者获胜机制中的检验。你可以假设检验和其他提交过程是串行执行的，即一次只有一个事务。 715

b. 解释如何修改上述机制，不用更新集合，而为每个数据项分配一个写时间戳，来实现先提交者获胜机制下有效性检验步骤作为提交过程的一部分。同样，你可以假设检验和其他提交过程是串行执行的。

c. 先提交者获胜机制可以采用上述时间戳来实现，除了当获得排他锁时立刻执行有效性检验，而不是在提交时执行。

i. 解释如何给数据项分配写时间戳来实现先提交者获胜机制。

ii. 说明由于锁机制，如果提交时重复有效性检验，结果不会发生变化。

iii. 解释在这种情况下，为什么没有必要串行地执行有效性检验和其他提交过程。

习题

15.20 严格两阶段封锁协议带来什么好处？会产生哪些弊端？

15.21 大部分数据库系统实现采用严格两阶段封锁协议。说明该协议流行的三点理由。

15.22 考虑树形协议的一个变种，它称为森林协议。数据库按有根树的森林的方式组织。每个事务 T_i 必须遵从以下规则：

- 每棵树上的首次封锁可以在任何数据项上进行。
- 树上的第二次以及此后的封锁申请仅当被申请结点的父结点上有锁时才能发出。
- 数据项解锁可在任何时候进行。
- 事务 T_i 释放数据项后不能再次封锁该数据项。

证明森林协议不保证可串行性。

15.23 在什么条件下避免死锁比允许死锁发生然后检测的方式代价更小？

15.24 如果通过死锁避免机制避免了死锁后，饿死仍有可能吗？解释你的答案。

15.25 在多粒度封锁中，隐式封锁与显式封锁有什么不同？

15.26 尽管SIX锁在多粒度封锁中很有用，但排他共享意向（XIS）锁则无用。为什么？ 716

15.27 多粒度协议中的规则指出，仅当事务 T_i 当前对 Q 的父结点持有IX或IS锁时，T_i 对结点 Q 可加S或IS锁。已知SIX和S锁比IX和IS锁更强，为什么协议不允许当父结点持有SIX或S锁时对该结点加S或IS锁？

15.28 当一个事务在时间戳排序协议下回滚，它被赋予新时间戳。为什么它不能简单地保持原有时间戳？

15.29 证明：存在满足两阶段封锁协议却不满足时间戳协议的调度，反之亦然。

15.30 在时间戳协议的一个修改版中，我们要求测试提交位以判定read请求是否必须等待。解释要交位如何防止级联中止。为什么该测试对write请求是不必要的。

15.31 如练习15.19讨论的，快照隔离可以通过时间戳有效性检验的形式来实现。然而，与保证可串行化的多版本时间戳排序机制不同，快照隔离不保证可串行化。解释导致这种差异结果的两种协议之间

关键的区别。

15.32 列出练习 15.19 描述的基于时间戳实现的先提交者获胜版本的快照隔离与 15.9.3 节描述的不做读有效性检验的乐观并发控制的主要相似点和区别。

15.33 解释幻象现象。为什么尽管采用两阶段封锁协议，该现象仍可能导致不正确的并发执行？

15.34 解释使用二级一致性的原因，这种方法有什么缺点？

15.35 请给出码值封锁调度的例子，说明如果查找、插入、删除操作中的任何一个不对下一码值进行封锁，都可能出现本发现的幻象。

15.36 许多事务更新一个公共数据项（例如，一个支行的现金余额）和若干私有数据项（例如，多个个人账户余额）。解释如何通过对事务操作进行排序来提高并发度（和吞吐量）。

15.37 考虑下面这个封锁协议：所有数据项都被编号，并且当解锁一个数据项时，只有标号更大的数据项
717 才可以加锁。锁可以在任何时候释放。只能使用排他锁。用例子说明这个协议不能保证可串行化。

文献注解

Gray 和 Reuter [1993] 在其教科书中全面讨论了事务处理的概念，包括并发控制的概念和实现细节。Bernstein 和 Newcomer [1997] 在其教科书中讨论了事务处理的多个方面，包括并发控制。

两阶段封锁协议由 Eswaran 等 [1976] 引入。树形协议来自 Silberschatz 与 Kedem [1980]。其他的工作在更一般的图上给出非两阶段封锁协议由 Yannakakis 等 [1979]、Kedem 与 Silberschatz [1983]，以及 Buckley 与 Silberschatz [1985] 给出。Korth [1983] 探讨了由基本的共享和排他锁方式可以得到的多种封锁方式。

实践习题 15.4 来自 Buckley 与 Silberschatz [1984]。实践习题 15.6 来自 Kedem 与 Silberschatz [1983]。实践习题 15.7 来自 Kedem 与 Silberschatz [1979]。实践习题 15.8 来自 Yannakakis 等 [1979]。实践习题 15.10 来自 Korth [1983]。

多粒度数据项封锁协议来自 Gray 等 [1975]。Gray 等 [1976] 给出了详细的描述。Kedem 和 Silberschatz [1983] 对任意封锁方式（允许更多的锁类而不仅仅是读和写）的多粒度封锁做了规范化。这种方法包括称为更新锁的一类锁，以处理锁转换。Carey [1983] 将多粒度的思想扩展到基于时间戳的并发控制。为保证不产生死锁而产生的一种扩展协议由 Korth [1982] 给出。

基于时间戳的并发控制机制来自 Reed [1983]。Buckley 与 Silberschatz [1983] 给出了一种不须回滚且保证可串行化的时间戳算法。有效性检查的并发控制机制来自 Kung 与 Robinson [1981]。

多版本时间戳排序由 Reed [1983] 引入。Silberschatz [1982] 中给出了一种多版本树形锁协议。

二级一致性在 Gray 等 [1975] 中介绍。SQL 中的一致性（或隔离性）的级别在 Berenson 等 [1995] 中做了解释和评论。许多商业数据库系统使用与加锁相结合的基于版本的方法。PostgreSQL、Oracle、和 SQL Server 全部支持在 15.6.2 节中提到的快照隔离协议。细节请分别参考第 27、28 和 30 章。

需要注意的是，在 PostgreSQL（版本 8.1.4）和 Oracle（版本 10g）中将隔离级别设置为串行化的结果是采用快照隔离，并不保证可串行性。Fekete 等 [2005] 介绍如何通过重写事务引入冲突，使得在快照隔离下保证可串行化执行。这些冲突保证在快照隔离下事务不能并发执行。Jorwekar 等 [2007] 介绍了一种方法，给定一个运行在快照隔离下的（参数化）事务集合，该方法可以检验这些事务是否存在非串行化的风险。

Bayer 与 Schkolnick [1977] 对 Johnson 与 Shasha [1993] 对 B$^+$ 树中的并发作了研究。15.10 节所给技术基于 Kung 与 Lehman [1980]，以及 Lehman 与 Yao [1981]。ARIES 系统中采用的码值封锁技术为 B$^+$ 树访问提供了
718 较高的并发性。在 Mohan [1990a]、Mohan 和 Narang [1992] 中描述。Ellis [1987] 给出了一个用于线性散列的
720 并发控制技术。

恢复系统

计算机系统与其他任何设备一样易发生故障。故障的原因多种多样，包括磁盘故障、电源故障、软件错误、机房失火，甚至人为破坏。一旦有任何故障发生，就可能会丢失信息。因此，数据库系统必须预先采取措施，以保证即使发生故障，也可以保持第14章所讲的事务的原子性和持久性。**恢复机制**(recovery scheme)是数据库系统必不可少的组成部分，它负责将数据库恢复到故障发生前的一致的状态。恢复机制还必须提供**高可用性**(high availability)，即：它必须将数据库崩溃后不能使用的时间缩减到最短。

16.1 故障分类

系统可能发生的故障有很多种，每种故障需要不同的方法来处理。在本章中，我们将只考虑如下类型的故障。

- **事务故障**(transaction failure)。有两种错误可能造成事务执行失败：
  - **逻辑错误**(logical error)。事务由于某些内部条件而无法继续正常执行，这些内部条件如非法输入、找不到数据、溢出或超出资源限制。
  - **系统错误**(system error)。系统进入一种不良状态(如死锁)，结果事务无法继续正常执行。但该事务可以在以后的某个时间重新执行。
- **系统崩溃**(system crash)。硬件故障，或者是数据库软件或操作系统的漏洞，导致易失性存储器内容的丢失，并使得事务处理停止。而非易失性存储器仍完好无损。

  硬件错误和软件漏洞致使系统终止，而不破坏非易失性存储器内容的假设称为**故障－停止假设**(fail-stop assumption)。设计良好的系统在硬件和软件层有大量的内部检查，一旦有错误发生就会将系统停止。因此，故障－停止假设是合理的。
- **磁盘故障**(disk failure)。在数据传送操作过程中由于磁头损坏或故障造成磁盘块上的内容丢失。其他磁盘上的数据拷贝，或三级介质(如DVD或磁带)上的归档备份可用于从这种故障中恢复。 [721]

要确定系统如何从故障中恢复，我们首先需要确定用于存储数据的设备的故障方式。其次，我们必须考虑这些故障方式对数据库内容有什么影响。然后我们可以给出在故障发生后仍保证数据库一致性以及事务的原子性的算法。这些算法称为恢复算法，由两部分组成：

1. 在正常事务处理时采取措施，保证有足够的信息可用于故障恢复。
2. 故障发生后采取措施，将数据库内容恢复到某个保证数据库一致性、事务原子性及持久性的状态。

16.2 存储器

正如我们在第10章中所看到的，数据库中的各种数据项可在多种不同存储介质上存储并访问。在14.3节中，我们看到存储介质可以按照它们相对的速度、容量和顾应故障的能力来划分。我们把存储器分为以下三类：

- **易失性存储器**(volatile storage)
- **非易失性存储器**(nonvolatile storage)
- **稳定存储器**(stable storage)

稳定存储器，或更准确地说是接近稳定的存储器，在恢复算法中起到至关重要的作用。

16.2.1 稳定存储器的实现

要实现稳定存储器，我们需要在多个非易失性存储介质（通常是磁盘）上以独立的故障模式复制所
722 需信息，并且以受控的方式更新信息，以保证数据传送过程中发生的故障不会破坏所需信息。

前面（第10章）讲到RAID系统保证了单个磁盘的故障（即使发生在数据传送过程中）不会导致数据丢失。最简单并且最快的RAID形式是磁盘镜像，即在不同的磁盘上为每个磁盘块保存两个拷贝。RAID的其他形式代价低一些，但性能也差一些。

但是，RAID系统不能防止由于灾难（如火灾或洪水）而导致的数据丢失。许多系统通过将归档备份存储在磁带上并转移到其他地方来防止这种灾难。但是，由于磁带不能被连续不断地移至其他地方，最后一次磁带被移至其他地方以后所做的更新可能会在这样的灾难中丢失。更安全的系统远程为稳定存储器的每一个块保存一份拷贝，除在本地磁盘系统进行块存储外，还通过计算机网络写到远程去。由于在往本地存储器输出块的同时也要输出到远程系统，一旦输出操作完成，即使发生火灾或洪水这样的灾难，输出结果也不会丢失。我们在16.9节学习这种远程备份系统。

本节剩余的部分将讨论如何在数据传送过程中保护存储介质不受故障损害。在内存和磁盘存储器间进行块传送有以下几种可能结果：

- **成功完成**（successful completion）。传送的信息安全地到达目的地。
- **部分失败**（partial failure）。传送过程中发生故障，目标块有不正确信息。
- **完全失败**（total failure）。传送过程中故障发生得足够早，目标块仍完好无缺。

我们要求，如果**数据传送故障**（data-transfer failure）发生，系统能检测到并且调用恢复过程将块恢复成为一致的状态。为达到这个要求，系统必须为每个逻辑数据库块维护两个物理块：若是镜像磁盘，则两个块在同一个地点；若是远程备份，则一个块在本地，另一个在远程节点。输出操作的执行如下：

1. 将信息写入第一个物理块。
2. 当第一次写成功完成时，将相同信息写入第二个物理块。
3. 只有第二次写成功完成时，输出才算完成。

如果在对块进行写的过程中系统发生故障，有可能一个块的两个拷贝互相不一致。在恢复过程中，对于每一个块，系统需要检查它的两个拷贝。如果它们相同并且没有检测到错误存在，则不需要采取进一步动作。（前面讲到，磁盘块中的某些错误，如部分写块，可由存储在每个块中的校验和检测
723 到。）如果系统检测到一个块中有错误，则可以用另一个块的内容替换这一块的内容。如果两个块都没有检测出错误，但它们的内容不一致，则我们用第二块的值替换第一块的内容，或者用第一块的值替换第二块的内容。不管用哪个方法，恢复过程都保证，对稳定存储器的写要么完全成功（即更新所有拷贝），要么没有任何改变。

在恢复过程中要求比较每一对相应块的开销太大。通过使用少量非易失性RAM，跟踪正在进行的对块的写操作，我们可以大大降低开销。在恢复时，只须比较正在写的块。

将块写到远程节点的协议类似于将块写到镜像磁盘系统的协议，我们在第10章讨论过了，特别是实践练习10.3中。

我们可以将这个过程很容易地推广为允许为稳定存储器的每一个块使用任意多的拷贝。尽管使用大量拷贝比使用两个拷贝发生故障的可能性要低，但通常只用两个拷贝模拟稳定存储器是合理的。

16.2.2 数据访问

正如第10章中所看到的，数据库系统驻留于非易失性存储器（通常为磁盘），在任何时间都只有数据库的部分内容在主存中。[⊖]数据库分成称为块（block）的定长存储单位。块是磁盘数据传送的单位。

⊖ 有一类特殊的数据库系统，称作主存数据库系统，它的整个数据库可以一次全部载入内存。26.4节讨论这样的系统。

可能包含多个数据项。我们假设没有数据项跨两个或多个块。这个假设对于大多数数据处理应用，例如银行或大学，都是正确的。

事务由磁盘向主存输入信息，然后再将信息输出回磁盘。输入和输出操作以块为单位完成。位于磁盘上的块称为**物理块**(physical block)，临时位于主存的块称为**缓冲块**(buffer block)。内存中用于临时存放块的区域称为**磁盘缓冲区**(disk buffer)。

磁盘和主存间的块移动是由下面两个操作引发的：

1. input(B)传送物理块 B 至主存。
2. output(B)传送缓冲块 B 至磁盘，并替换磁盘上相应的物理块。

这一机制如图 16-1 所示。

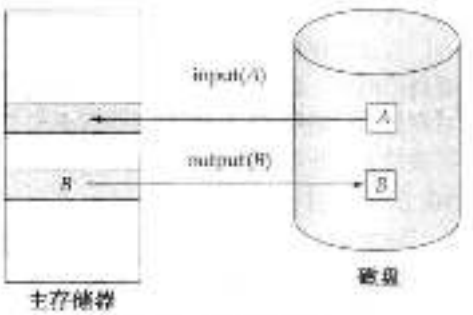


图 16-1 块存储操作

在概念上，每个事务 T_i 有一个私有工作区，用于保存 T_i 所访问及更新的所有数据项的拷贝。该工作区在事务初始化时由系统创建；在事务提交或中止时由系统删除。事务 T_i 的工作区中保存的每一个数据项 X 记为 x_i。事务 T_i 通过在其工作区和系统缓冲区之间传送数据，与数据库系统进行交互。我们使用下面两个操作来传送数据：

1. read(X)将数据项 X 的值赋予局部变量 x_i。该操作执行如下：
   a. 若 X 所在块 B_X 不在主存中，则发指令执行 input(B_X)。
   b. 将缓冲块中 X 的值赋予 x_i。
2. write(X)将局部变量 x_i 的值赋予缓冲块中的数据项 X。该操作执行如下：
   a. 若 X 所在块 B_X 不在主存中，则发指令执行 input(B_X)。
   b. 将 x_i 的值赋予缓冲块 B_X 中的 X。

注意这两个操作都可能需要将块从磁盘传送到主存。但是，它们都没有特别指明需要将块从主存传送到磁盘。

缓冲块最终写到磁盘，要么是因为缓冲区管理器出于其他用途需要内存空间，要么是因为数据库系统希望将 B 的变化反映到磁盘上。如果数据库系统发指令执行 output(B)，则我们称数据库系统对缓冲块 B 进行**强制输出**(force-output)。

当事务第一次需要访问数据项 X 时，它必须执行 read(X)。该事务然后对 X 的所有更新都作用于 x_i。在一个事务执行中的任何时间点，事务都可以执行 write(X)，以在数据库中反映 X 的变化；在对 X 进行最后的写之后，当然必须做 write(X)。

724
725

对 X 所在的缓冲块 B_X 的 output(B_X)操作不需要在 write(X)执行后立即执行，因为块 B_X 可能包含其他仍在被访问的数据项。因此，可能一段时间以后才真正执行输出。注意，如果在 write(X) 操作执行后但在 output(B_X)操作执行前系统崩溃，X 的新值并未写入磁盘，于是就丢失了 X 的新值。正如我们很快就会看到的，数据库系统执行额外的动作来保证，即使发生了系统崩溃，由提交的事务所做的更新也不会丢失。

16.3 恢复与原子性

再来考虑简化的银行系统和事务 T_i。它将 $50 从账户 A 转到账户 B，A 和 B 的初始值分别为 $1000 和 $2000。假设在 T_i 执行过程中，在 output(B_A)之后，output(B_B)之前，发生了系统崩溃，其中 B_A 和 B_B 表示 A 和 B 所在的缓冲块。由于内存的内容丢失，因此我们无法知道事务的结局。

当系统重新启动时，A 的值会是 $950，而 B 的值是 $2000，这显然和事务 T_i 的原子性需求不一致。遗憾的是，没有办法通过检查数据库状态来找出在系统崩溃发生前哪些块已经输出，哪些块还没有。有可能事务已经完成了。对稳定存储器中的数据库初始状态 A 和 B 的值分别为 $1000 和 $1950 进行了更新；也可能事务没有对稳定存储器产生任何影响，A 和 B 的值初始就是 $950 和 $2000；或者更新后的 B 已经输出了，而更新后的 A 还没有输出，或更新后的 A 已经输出了，而更新后的 B 还

没有输出。

我们的目标是要么执行 T_i 对数据库的所有修改，要么都不执行。但是，若 T_i 执行多处数据库修改，就可能需要多个输出操作，并且故障可能发生于某些修改完成后而全部修改完成前。

为达到保持原子性的目标，我们必须在修改数据库本身之前，首先向稳定存储器输出信息，描述要做的修改。我们将看到，这种信息能帮助我们确保已提交事务所做的所有修改都反映到数据库中（或者在故障后的恢复过程中反映到数据库中）。这种信息还能帮助我们确保中止事务所做的任何修改都不会持久存在于数据库中。

16.3.1 日志记录

使用最为广泛的记录数据库修改的结构就是日志（log）。日志是日志记录（log record）的序列。它记录数据库中的所有更新活动。

726 日志记录有几种。**更新日志记录**（update log record）描述一次数据库写操作，它具有如下几个字段：

- **事务标识**（transaction identifier），是执行 write 操作的事务的唯一标识。
- **数据项标识**（data-item identifier），是所写数据项的唯一标识。通常是数据项在磁盘上的位置，包括数据项所驻留的块的块标识和块内偏移量。
- **旧值**（old value），是数据项的写前值。
- **新值**（new value），是数据项的写后值。

我们将一个更新日志记录表示为 $<T_i, X_j, V_1, V_2>$，表明事务 T_i 对数据项 X_j 执行了一个写操作。写操作前 X_j 的值是 V_1，写操作后 X_j 的值是 V_2。其他专门的日志记录用于记录事务处理过程中的重要事件，如事务的开始以及事务的提交或中止。如下是一些日志记录类型：

- $<T_i$ start$>$。事务 T_i 开始。
- $<T_i$ commit$>$。事务 T_i 提交。
- $<T_i$ abort$>$。事务 T_i 中止。

后面将介绍几种其他的日志记录类型。

每次事务执行写操作时，必须在数据库修改前建立该次写操作的日志记录并把它加到日志中。一旦日志记录已存在，就可以根据需要将修改输出到数据库中。并且，我们有能力撤销已经输出到数据库中的修改。这是利用日志记录中的旧值字段来做的。

为了从系统故障和磁盘故障中恢复时能使用日志记录，日志必须存放在稳定存储器中。现在我们假设每一个日志记录创建后立即写入稳定存储器中的日志的尾部，在 16.5 节中我们将看到什么时候可以放宽这个要求，以减少写日志带来的开销。由于日志包含所有的数据库活动的完整记录，因此日志中存储的数据量会变得非常大，在 16.3.6 节我们将看到什么时候可以安全删除日志信息。

影子拷贝和影子页面

在**影子拷贝**（shadow-copy）模式下，想要更新数据库的事务应首先创建数据库的一个完整拷贝。所有的更新在数据库的这个新拷贝上进行，而不去动那个原来的拷贝。**影子拷贝**（shadow copy）如果在任何一个点上事务需要中止，系统仅仅删除这个新拷贝。数据库的旧拷贝没有受到影响。数据库的当前拷贝由一个指针来标识，称作数据库指针，它存放在磁盘上。

如果事务部分提交（即，执行它的最后一条语句），那么它如下进行提交：首先，要求操作系统确保数据库的新拷贝的所有页面都写到磁盘上。（为此目的，UNIX 系统使用 fsync 命令。）在操作系统将所有页面都写到磁盘上之后，数据库系统更新数据库指针，让它指向数据库的新拷贝；然后新拷贝变成数据库的当前拷贝。然后删除掉数据库的旧拷贝。在更新后的数据库指针写到磁盘上这一时间点，我们说该事务已经提交了。

影子拷贝的实现实际上依赖于对数据库指针的写是原子的；即，或者它的所有字节全都写出，或者没有任何字节写出。磁盘系统提供对整个块的原子更新，或至少是对一个磁盘扇区的。换句话说，磁盘系统保证它会原子地更新数据库指针，只要我们确保数据库指针完全处于单个扇区中，而这是我们通过将数据库指针存放在块的开头所能够保证的。

影子拷贝模式普遍用于正文编辑器(保存文件等价于事务提交，不保存文件就退出等价于事务中止)。影子拷贝可以用于小的数据库，但拷贝一个大型数据库会是相当昂贵的。影子拷贝的一个变种，称作**影子页面**(shadow-paging)，它采用如下方式来减少拷贝工作量：此种模式使用一个包含指向所有页面的指针的页表；页表自身和所有更新的页面就拷贝到一个新的位置。事务没有更新的任何页面都不拷贝，而新的页表只存储一个指向原来页面的指针。当提交事务时，它原子地更新指向页表(页表的作用和数据库指针相同)的指针，以指向新的拷贝。

遗憾的是，影子页面对于并发事务不能很好地工作，在数据库中它没有广泛使用。

16.3.2 数据库修改

正如我们前面已经注意到的，事务在对数据库进行修改前创建了一个日志记录。日志记录使得系统在事务必须中止的情况下能够对事务所做的修改进行撤销；并且在事务已经提交但在修改已存放到磁盘上的数据库中之前系统崩溃的情况下能够对事务所做的修改进行重做。为了使我们能够理解恢复过程中日志记录的作用，我们需要考虑事务在进行数据项修改中所采取的步骤：

1. 事务在主存中自己私有的部分执行某些计算。
2. 事务修改主存的磁盘缓冲区中包含该数据项的数据块。
3. 数据库系统执行 **output** 操作，将数据块写到磁盘中。

727
↓
728

如果一个事务执行了对磁盘缓冲区或磁盘自身的更新，我们说这个事务修改了数据库；而对事务在主存中自己私有的部分进行的更新不算数据库修改。如果一个事务直到它提交时都没有修改数据库，我们就说它采用了**延迟修改**(deferred-modification)技术。如果数据库修改在事务仍然活跃时发生，我们就说它采用了**立即修改**(immediate-modification)技术。延迟修改所付出的开销是，事务需要创建更新过的所有的数据项的本地拷贝；而且如果一个事务读它更新过的数据项，它必须从自己的本地拷贝中读。

本章描述的恢复算法支持立即修改。正如所描述的，即使对于延迟修改，它们也能正确工作，但是当与延迟修改一起使用时可以进行优化，以减少开销；我们将细节留作练习。

恢复算法必须考虑多种因素，包括：

- 有可能一个事务已经提交了，虽然它所做的某些数据库修改还仅仅存在于主存的磁盘缓冲区中，而不在磁盘上的数据库中。
- 有可能处于活动状态的一个事务已经修改了数据库，而作为后来发生的故障的结果，这个事务需要中止。

由于所有的数据库修改之前必须建立日志记录，因此系统有数据项修改前的旧值和要写给数据项的新值可以用。这就使得系统能执行适当的 *undo* 和 *redo* 操作：

- undo 使用一个日志记录，将该日志记录中指明的数据项设置为旧值。
- redo 使用一个日志记录，将该日志记录中指明的数据项设置为新值。

16.3.3 并发控制和恢复

如果并发控制模式允许一个事务 T_i 修改过的数据项 X 在 T_i 提交前进一步地由另一个事务 T_j 修改，那么通过将 X 重置为它的旧值(T_i 更新 X 之前的值)来撤销 T_i 的影响同时也会撤销 T_j 的影响。为避免这样的情形发生，恢复算法通常要求如果一个数据项被一个事务修改了，那么在该事务提交或中止前不允许其他事务修改该数据项。

这一要求可以通过对更新的数据项获取排他锁，并且持有该锁直至事务提交来保证；换句话说，
729 通过使用严格两阶段封锁。快照隔离性和基于有效性验证的并发控制技术在有效性验证时，在修改数据项之前，也要获取数据项上的排他锁，直至事务提交；其结果是，即使通过这些并发控制协议，上述要求也能得到满足。

后面在 16.7 节讨论，在一定的情况下可以如何放松上述要求。

在采用快照隔离性或有效性验证进行并发控制时，事务所做的数据库更新（从概念上）是延迟到事务部分提交时；延迟修改技术与这些并发控制模式自然吻合。然而，值得注意的是，快照隔离性的某些实现采用了立即修改技术，而根据需要提供了一个逻辑快照：当事务需要读被并发的事务更新的一个数据项时，就生成该数据项（已经更新）的一个拷贝，在这个拷贝上，并发事务所做的更新回滚。类似地，数据库的立即修改与两阶段封锁自然吻合，但延迟修改也可以和两阶段封锁一起使用。

16.3.4 事务提交

当一个事务的 commit 日志记录——这是该事务的最后一个日志记录——输出到稳定存储器后，我们就说这个事务提交（commit）了；这时所有更早的日志记录都已经输出到稳定存储器中。于是，在日志中就有足够的信息来保证，即使发生系统崩溃，事务所做的更新也可以重做。如果系统崩溃发生在日志记录 $<T_i$ commit$>$ 输出到稳定存储器之前，事务 T_i 将回滚。这样，包含 commit 日志记录的块的输出是单个原子动作，它导致一个事务的提交。[⊖]

对于大多数基于日志的恢复技术，包括本章描述的技术，不是在一个事务提交时必须将包含该事务修改的数据项的块输出到稳定存储器中，可以在以后的某个时间再输出。16.5.2 节进一步讨论这个问题。

16.3.5 使用日志来重做和撤销事务

我们现在提供一个关于如何使用日志来从系统崩溃中进行恢复以及在正常操作中对事务进行回滚的概览。但是，我们将故障恢复和回滚过程的细节推迟到 16.4 节。

730 考虑简化的银行系统。令 T_0 是一个事务，它将 50 美元从账户 A 转到账户 B：

```
T0 : read(A);
     A := A − 50;
     write(A);
     read(B);
     B := B + 50;
     write(B).
```

令 T_1 是一个事务，它从账户 C 中取出 100 美元。

```
T1 : read(C);
     C := C − 100;
     write(C).
```

日志包含与这两个事务相关信息的部分如图 16-2 所示。

```
<T0 start>
<T0, A, 1000, 950>
<T0, B, 2000, 2050>
<T0 commit>
<T1 start>
<T1, C, 700, 600>
<T1 commit>
```

图 16-2 系统日志中与 T_0 和 T_1 相应的部分

图 16-3 显示了一个可能的顺序，在这个顺序中，作为 T_0 和 T_1 的执行结果，对于数据库系统和日志的实际的输出都发生了。[⊜]

使用日志，系统可以对付任何故障，只要它不导致非易失性存储器中信息的丢失。恢复系统使用两个恢复过程，它们都利用日志来找到每个事务 T_i 更新过的数据项的集合，以及它们各自的旧值和新值。

- redo(T_i) 将事务 T_i 更新过的所有数据项的值都设置成新值。

⊖ 一个块的输出可以通过对付数据传输故障的技术而做成忘了的，正如 16.2.1 节所描述的。

⊜ 请注意，如果使用延迟修改技术，就不能得到这个顺序，因为在 T_0 提交之前数据库不会修改，对于 T_1 也一样。

通过 redo 来执行更新的顺序是非常重要的：当从系统崩溃中恢复时，如果对一个特定数据项的多个更新的执行顺序不同于它们原来的执行顺序，那么该数据项的最终状态将是一个错误的值。大多数的恢复算法，包括 16.4 节描述的算法，都没有把每个事务的重做分别执行，而是对日志进行一次扫描，在扫描过程中每遇到一个 redo 日志记录就执行 redo 动作。这种方法能确保保持更新的顺序，并且效率更高，因为仅需要整体读一遍日志，而不是对每个事务读一遍日志。

日志	数据
<T_0 start>	
<T_0, A, 1000, 950>	
<T_0, B, 2000, 2050>	
	A = 950
	B = 2050
<T_0 commit>	
<T_1 start>	
<T_1, C, 700, 600>	
	C = 600
<T_1 commit>	

图 16-3 与 T_0 和 T_1 相应的系统日志和数据库状态

- undo(T_i) 将事务 T_i 重新过的所有数据项的值都恢复成旧值。

  在 16.4 节中描述的恢复机制中：

  - undo 操作不仅将数据项恢复成它的旧值，而且作为撤销过程的一个部分，还写日志记录来记下所执行的更新。这些日志记录是特殊的 **redo-only** 日志记录，因为它们不需要包含所更新的数据项的旧值。

    与重做过程一样，执行更新的顺序是非常重要的；我们还是将细节推迟到 16.4 节中。
  - 当对于事务 T_i 的 undo 操作完成后，它写一个 <T_i **abort**> 日志记录，表明撤销完成了。

  正如我们将在 16.4 节中看到的，对于每一个事务，undo(T_i) 只执行一次，如果在正常的处理中该事务回滚，或者在系统崩溃后的恢复中既没有发现事务 T_i 的 **commit** 记录，也没有发现事务 T_i 的 **abort** 记录。其结果是，在日志中每一个事务最终或者有一条 **commit** 记录，或者有一条 **abort** 记录。

发生系统崩溃之后，系统查阅日志以确定为保证原子性需要对哪些事务进行重做，对哪些事务进行撤销。

- 如果日志包括 <T_i start> 记录，但既不包括 <T_i **commit**>，也不包括 <T_i **abort**> 记录，则需要对事务 T_i 进行撤销。
- 如果日志包括 <T_i **start**> 记录，以及 <T_i **commit**> 或 <T_i **abort**> 记录，需要对事务 T_i 进行重做。如果日志包括 <T_i **abort**> 记录还要进行重做，看来比较奇怪。要明白这是为什么，请注意如果在日志中有 <T_i **abort**> 记录，日志中也会有 undo 操作所写的那些 redo-only 日志记录。于是，这种情况下最终结果将是对 T_i 所做的修改进行撤销。这一轮做的冗余简化了恢复算法，并使得整个恢复过程变得更快。

作为一个描述，让我们回到银行的例子，有事务 T_0 和 T_1，按照 T_1 跟在 T_0 后面的顺序执行。假定在事务完成之前系统崩溃。我们将要考虑三种情况。在图 16-4 中显示了各种情况下的日志。

首先，我们假定崩溃恰好发生在事务 T_0 的

write(B)

步骤的日志记录已经写到稳定存储器之后（见图 16-4a）。当系统重新启动时，它在日志中找到记录 <T_0 **start**>，但是没有相应的 <T_0 **commit**> 或 <T_0 **abort**> 记录。这样，事务 T_0 必须撤销。于是执行 undo(T_0)。其结果是，（磁盘上）账户 A 和账户 B 的值分别恢复成 $1000 和 $2000。

其次，我们假定崩溃恰好发生在事务 T_1 的

write(C)

步骤的日志记录已经写到稳定存储器之后（见图 16-4b）。当系统重新启动时，需要采取两个恢复动作。因为 <T_1 **start**> 记录出现在日志中，但是没有 <T_1 **commit**> 或 <T_1 **abort**> 记录，所以必须执行 **undo**(T_1)。因为日志中既包括 <T_0 **start**> 记录，又包括 <T_0 **commit**> 记录，所以必须执行 **redo**(T_0)。在整个恢复过程结束时，账户 A、B、和 C 的值分别为 $950、$2050 和 $700。

最后，我们假定崩溃恰好发生在事务 T_1 的日志记录：

<T_1 **commit**>

已经写到稳定存储器之后（见图 16-4c）。当系统重新启动时，因为 <T_0 **start**> 记录和 <T_0 **commit**> 记录都在日志中，并且 <T_1 **start**> 记录和 <T_1 **commit**> 记录也都在日志中，所以 T_0 和 T_1 都必须重做。

731 ↓ 733

在系统执行 **redo**(T_0) 和 **redo**(T_1)过程后，账户 A、B 和 C 的值分别为 $950、$2050 和 $600。



图 16-4 在三个不同时间显示的同一个日志

16.3.6 检查点

当系统故障发生时，我们必须检查日志，决定哪些事务需要重做，哪些需要撤销。原则上，我们需要搜索整个日志文件来确定该信息。这样做有两个主要的困难：

1. 搜索过程太耗时。

2. 根据我们的算法，大多数需要重做的事务已把其更新写入数据库中。尽管对它们重做不会造成不良后果，但会使恢复过程变得更长。

为降低这种开销，引入检查点。

下面描述一个简单的检查点，它(a)在执行检查点操作的过程中不允许执行任何更新，(b)在执行检查点的过程中将所有更新过的缓冲块都输出到磁盘。后面会讨论如何通过放松这两条要求来修改检查点和恢复过程，以提供更高的灵活性。

检查点的执行过程如下：

1. 将当前位于主存的所有日志记录输出到稳定存储器。

2. 将所有修改的缓冲块输出到磁盘。

3. 将一个日志记录 < **checkpoint** L > 输出到稳定存储器，其中 L 是执行检查点时正活跃的事务的列表。

在检查点执行过程中，不允许事务执行任何更新动作，如往缓冲块中写入或写日志记录。16.5.2 节会讨论如何强制满足这个要求。

在日志中加入 < **checkpoint** L > 记录使得系统提高恢复过程的效率。考虑在检查点前完成的事务 T_i。对于这样的事务，< T_i **commit** > 记录(或 < T_i **abort** > 记录)在日志中出现在 < **checkpoint** > 记录之前。T_i 所做的任何数据库修改都必然已在检查点前或作为检查点本身的一部分写入数据库。因此，在恢复时就不必再对 T_i 执行 **redo** 操作了。

734 在系统崩溃发生之后，系统检查日志以找到最后一条 < **checkpoint** L > 记录(这可以通过从尾端开始反向搜索日志来进行，直至遇到第一条 < **checkpoint** L > 记录)。

只需要对 L 中的事务，以及 < **checkpoint** L > 记录写到日志中之后才开始执行的事务进行 undo 或 redo 操作。让我们把这个事务集合记为 T。

- 对 T 中所有事务 T_i，若日志中既没有 < T_i **commit** > 记录，也没有 < T_i **abort** > 记录，则执行 undo(T_i)。
- 对 T 中所有事务 T_i，若日志中有 < T_i **commit** > 记录或 < T_i **abort** > 记录，则执行 redo(T_i)。

请注意，要找出事务集合 T，和确定 T 中的每个事务是否有 commit 或 abort 记录出现在日志中，我们只需要检查日志中从最后一条 checkpoint 日志记录开始的部分。

作为一个例子，考虑事务集合 $\{T_0, T_1, \cdots, T_{66}\}$。假设最近的检查点发生在事务 T_{67} 和 T_{68} 执行的过程中，而 T_{69} 和下标小于 67 的所以事务在检查点之前都已完成。于是，在恢复机制中只需要考虑事务 T_{67}、T_{68}、…、T_{100}。其中已完成(即已提交或已终止)的需要重做；否则就是未完成的，需要撤销。

考虑检查点日志记录中的事务集合 L。对于 L 中的每一个事务 T_i，如果它没有提交，那么为了对该事务进行撤销，可能需要该事务发生在检查点日志记录之前的所有日志记录。无论如何，一旦检查点完成了，我们就不再需要位于最先出现的日志记录 < T_i **start** > (这儿的 T_i 是 L 中的任何事务)之前的所有日志记录了。当数据库系统需要回收这些记录占用的空间时，就可以清掉这些日志记录。

在检查点过程中不允许事务对缓冲块或日志进行任何更新，这一要求会引起相当的麻烦，因为这样一来在检查点进行的过程中事务处理就必须停顿下来。**模糊检查点**(fuzzy checkpoint)是这样的检查点，它即使在缓冲块正在写出时也允许事务执行更新。16.5.4 节对模糊检查点模式进行描述。然后 16.8 节共描述一个检查点模式，它不仅是模糊的，而且甚至不要求在检查点时刻将所有修改过的缓冲块输出到磁盘中。

16.4 恢复算法

到目前为止，在对故障恢复的讨论中，我们确定了需要对哪些事务进行重做和需要对哪些事务进行撤销，但是我们没有给出进行这些动作的详细算法。现在我们来给出使用日志记录从事务故障中恢复的完整恢复算法，以及将最近的检查点和日志记录结合起来以从系统崩溃中进行恢复的算法。 735

本节描述的恢复算法要求未提交的事务更新过的数据项不能被任何其他事务修改。直至更新它的事务或者提交或者中止。这一限制在 16.3.3 节曾经讨论过。

16.4.1 事务回滚

首先考虑正常操作时(即不是从系统崩溃中恢复时)的事务回滚。事务 T_i 的回滚如下执行。

1. 从后往前扫描日志，对于所发现的 T_i 的每一个形如 $<T_i, X_j, V_1, V_2>$ 的日志记录：
   a. 值 V_1 被写到数据项 X_j 中，并且
   b. 往日志中写一个特殊的只读日志记录 $<T_i, X_j, V_1>$。其中 V_1 是在本次回滚中数据项 X_j 恢复成的值。有时这种日志记录称作补偿日志记录(compensation log record)。这样的日志记录不需要 undo 信息，因为我们绝不会需要撤销这样的 undo 操作。后面会解释如何使用这些日志记录。
2. 一旦发现了 $<T_i$, **start**$>$ 日志记录，就停止从后往前的扫描，并往日志中写一个 $<T_i$, **abort**$>$ 日志记录。

请注意，现在事务所做的或者我们为事务做的每一个更新动作，包括将数据项恢复成其旧值的动作，都记录到日志中了。在 16.4.2 节中我们将看到为什么这是个好办法。

16.4.2 系统崩溃后的恢复

崩溃发生后当数据库系统重启时，恢复动作分两阶段进行：

1. **在重做阶段**，系统通过从最后一个检查点开始正向地扫描日志来重放所有事务的更新。重放的日志记录包括在系统崩溃之前已回滚的事务的日志记录，以及在系统崩溃发生时还没有提交的事务的日志记录。这个阶段还确定在系统崩溃发生时未完成，因此必须回滚事务。这样的未完成事务或者在检查点时是活跃的，因此会出现在检查点记录的事务列表中，或者是在检查点之后开始的；而且，这样的未完成事务在日志中既没有 $<T_i$, **abort**$>$ 记录，也没有 $<T_i$, **commit**$>$ 记录。

   扫描日志的过程中所采取的具体步骤如下：
   a. 将要回滚的事务的列表 undo-list 初始设定为 $<$**checkpoint** $L>$ 日志记录中的 L 列表。 736
   b. 一旦遇到形为 $<T_i, X_j, V_1, V_2>$ 的正常日志记录或形为 $<T_i, X_j, V_2>$ 的 redo-only 日志记录，就重做这个操作；也就是说，将值 V_2 写给数据项 X_j。
   c. 一旦发现形为 $<T_i$, **start**$>$ 的日志记录，就把 T_i 加到 undo-list 中。
   d. 一旦发现形为 $<T_i$, **abort**$>$ 或 $<T_i$, **commit**$>$ 的日志记录，就把 T_i 从 undo-list 中去掉。

   在 redo 阶段的末尾，undo-list 包括在系统崩溃之前尚未完成的所有事务，即，既没有提交也没有完成回滚的那些事务。
2. **在撤销阶段**，系统回滚 undo-list 中的所有事务。它通过从尾端开始反向扫描日志来执行回滚。
   a. 一旦发现属于 undo-list 中的事务的日志记录，就执行 undo 操作，就像在一个失败事务的回滚过程中发现了该日志记录一样。

b. 当系统发现 undo-list 中事务 T_i 的 $< T_i$, start > 日志记录，它就往日志中写一个 $< T_i$, abort > 日志记录，并且把 T_i 从 undo-list 中去掉。

c. 一旦 undo-list 变为空表，即系统已经找到了开始时位于 undo-list 中的所有事务的 $< T_i$, start > 日志记录，则撤销阶段结束。

当恢复过程的撤销阶段结束之后，就可以重新开始正常的事务处理了。

请注意，在重做阶段从最近的检查点记录开始重放每一个日志记录。换句话说，重启动恢复的这个阶段重复检查点之后执行并且其日志记录已经到达稳定存储器中的日志的所有更新动作。这些动作包括未完成事务的动作和为回滚失败的事务所执行的动作。这些动作按照它们原先执行的次序重复；因此，这一过程称作**重复历史**(repeating history)。虽然这看起来是浪费，但即使对失败事务也重复，这样的做法简化了恢复过程。

图 16-5 显示了在正常操作中由日志记录下的动作，以及在故障恢复中执行的动作的一个例子。在图 16-1 中显示的日志中，在系统崩溃之前，事务 T_1 已提交，事务 T_0 已完全回滚。请注意在 T_2 的回滚中如何恢复数据项 B 的值。还请注意检查点记录，它的活跃事务列表包含 T_0 和 T_2。

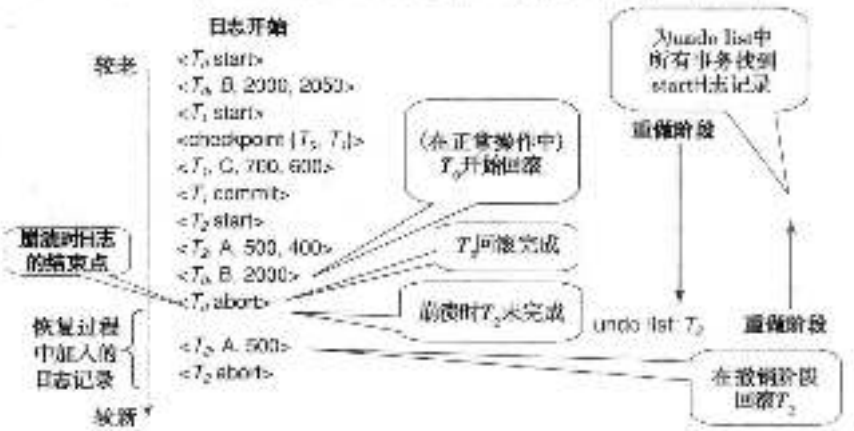


图 16-5 记录在日志中的动作和恢复中的动作的例子

当从崩溃中恢复时，在重做阶段，系统对最后一个检查点记录之后的所有操作执行 redo。在这个阶段中，undo-list 初始时包含 T_0 和 T_2；当 T_1 的 **commit** 日志记录被发现时，T_1 先被从表中去掉，而当 T_2 的 **start** 日志记录被发现时，T_2 被加到表中。当事务 T_0 的 **abort** 日志记录被发现时，T_0 被从 undo-list 中去掉，只剩 T_2 在 undo-list 中。撤销阶段从尾端开始反向扫描日志，当发现 T_2 更新 A 的日志记录时，将 A 恢复成旧值，并往日志中写一个 redo-only 日志记录。当发现 T_2 开始的日志记录时，就为 T_2 添加一条 **abort** 记录。由于 undo-list 不再包含任何事务了，因此撤销阶段终止，恢复完成。

16.5 缓冲区管理

在本节中，我们考虑几个微妙的细节，它们对实现保证数据一致性且只增加少量与数据库交互的开销的故障恢复机制非常重要。

16.5.1 日志记录缓冲

迄今为止，我们假设每个日志记录在创建时都输出到稳定存储器。该假设增加了大量系统执行的开销。其原因是：通常向稳定存储器的输出是以块为单位进行的。在大多数情况下，一个日志记录比一个块要小得多。因此，每个日志记录的输出转化成在物理上大得多的输出。另外，正如 16.2.1 节中所看到的，向稳定存储器输出一块在物理层上可能涉及几个输出操作。

将一个块输出到稳定存储器的开销非常高，因此最好是一次输出多个日志记录。为了达到这个目的，我们将日志记录写到主存的日志缓冲区中，日志记录在输出到稳定存储器以前临时保存在那儿。可以聚集多个日志记录在日志缓冲区中，然后用一次输出操作输出到稳定存储器中。稳定存储器中

737
↓
738

的日志记录顺序必须与写入日志缓冲区的顺序完全一样。

由于使用了日志缓冲区，日志记录在输出到稳定存储器前可能有一段时间只存在于主存(易失性存储器)中。由于系统发生崩溃时这种日志记录会丢失，我们必须对恢复技术增加一些要求以保证事务的原子性：

- 在日志记录 $<T_i$ **commit**$>$ 输出到稳定存储器后，事务 T_i 进入提交状态。
- 在日志记录 $<T_i$ **commit**$>$ 输出到稳定存储器前，与事务 T_i 有关的所有日志记录必须已经输出到稳定存储器。
- 在主存中的数据块输出到数据库(非易失性存储器)前，所有与该数据块中数据有关的日志记录必须已经输出到稳定存储器。

  这一条规则称为**先写日志**(Write-Ahead Logging，WAL)规则。(严格地说，WAL 规则只要求日志中的 undo 信息已经输出到稳定存储器中，而 redo 信息允许以后再写。这一区别与 undo 信息和 redo 信息分别存储在不同的日志记录中的系统是相关的。)

这二条规则表明，在某些情况下某些日志记录必须已经输出到稳定存储器中。而提前输出日志记录不会造成任何问题。因此，当系统发现需要将一个日志记录输出到稳定存储器时，如果主存中有足够的日志记录可以填满整个日志记录块，就将其整个输出。如果没有足够的日志记录填入该块，那么就将主存中的所有日志记录填入一个部分填充的块，并输出到稳定存储器。

将缓冲的日志写到磁盘有时称为**强制日志**(log force)。

16.5.2 数据库缓冲

16.2.2 节描述了两层存储层次结构的使用。系统将数据库存储在非易失性存储器(磁盘)中，并且在需要时将数据块调入主存。由于主存通常比整个数据库小得多，因此可能在需要将块 B_2 调入内存的时候覆盖主存中的块 B_1。如果 B_1 已经修改过，那么 B_1 必须在输入 B_2 前被输出。与 10.8.1 节讨论的一样，这个存储层次结构类似于操作系统中标准的虚拟内存概念。

人们可能期望事务在提交时强制地将修改过的所有的块都输出到磁盘。这样的策略称作**强制**(force)策略。另一种方法是**非强制**(no-force)策略，即使一个事务修改了某些还没有写回到磁盘的块，也允许它提交。本章描述的所有恢复算法即使在非强制策略的情况下也能正确工作。非强制策略使得事务能更快速地提交；而且它使得在一个块输出到稳定存储器之前可以将多个更新积聚在一起，这可 739
以大大减小频繁更新的块的输出操作的数目。其结果是，大多数系统所采用的标准的方法是非强制策略。

类似地，人们可能期望一个仍然活跃的事务修改过的块都不应该写出到磁盘。这一策略称作**非窃取**(no-steal)策略。另一种方法是**窃取**(steal)策略，允许系统将修改过的块写到磁盘，即使做这些修改的事务还没有全部提交。只要遵守先写日志规则，本章研究的所有的恢复算法都能正确工作，即使采用窃取策略。而且，非窃取策略不适合于执行大量更新的事务，因为缓冲区可能被已更新过但不能逐出到磁盘的页面占满，进而使得事务不能进展下去。其结果是，大多数系统所采用的标准方法是窃取策略。

为阐明先写日志要求的必要性，我们考虑银行例子中的事务 T_0 和 T_1。假设日志的状态是

$$<T_0, \text{start}>$$
$$<T_0, A, 1000, 950>$$

并假设事务 T_1 发指令执行 **read**(B)。假设 B 所在的块不在主存中，并且主存已满。假设选择 A 所在的块输出到磁盘上。如果将该块输出到磁盘上后系统崩溃，数据库中账户 A、B 和 C 的值分别就是 $950、$2000 和 $700，这是不一致的数据库状态。但是，由于有 WAL 要求，因此日志记录

$$<T_0, A, 1000, 950>$$

必须在输出 A 所在块之前输出到稳定存储器中。系统可以在恢复时使用该日志记录将数据库恢复到一致状态。

当要将块 B_1 输出到磁盘时，所有与块 B_1 中的数据相关的日志记录必须在块 B_1 输出之前先输出到稳定存储器中。重要的是，在块 B_1 正在输出时，不能有往块 B_1 中的写正在进行，因为这样的写会违反先写日志规则。我们可以通过使用特殊的封锁方法来保证没有正在进行的写：

- 在事务对一个数据项执行写操作之前，它要获得数据项所在的块的排他锁。当更新执行完后立即释放该锁。
- 当一个块要输出时，采取以下动作序列：
  - 获取该块上的排他锁，以确保没有任何事务正在对该块执行写操作。
  - 将日志记录输出到稳定存储器，直至与块 B_1 相关的所有日志记录都输出了。
  - 将块 B_1 输出到磁盘。
  - 一旦块输出完成，就释放锁。

[740]

缓冲块上的锁与用于事务并发控制的锁无关，按照非两阶段的方式释放这样的锁对于事务可串行性没有任何影响。这样的锁以及其他类似的短期持有的锁，通常称作闩锁（latch）。

缓冲块上的锁还可以用来保证在检查点进行的过程中缓冲块不更新，而且没有新的日志记录产生。可以通过要求在检查点操作开始执行之前必须获得在所有的缓冲块上的排他锁以及对日志的排他锁来实施这一限制。一旦检查点操作完成就可以释放这些锁。

数据库系统通常有一个不断地在缓冲块间循环，将修改过的缓冲块输出到磁盘的过程。在输出缓冲块时当然必须遵循上述的封锁协议。作为不断地输出修改过的缓冲块的结果，缓冲区中脏块（即缓冲区中修改过，但还没有输出的块）的数目被大幅减小了。于是，在检查点过程中需要输出的块的数目减小了；而且，当需要从缓冲区中逐出一个块时，很可能就有不脏的块可以被逐出，使得输入马上可以进行，而不必等待输出的完成。

16.5.3 操作系统在缓冲区管理中的作用

我们可以用下面两种方法之一来管理数据库缓冲区：

1. 数据库系统保留部分主存作为缓冲区，并对它进行管理，而不是让操作系统来管理。数据库系统按照 16.5.2 节讨论的那些要求管理数据块的传送。

这种方法的缺点是限制了主存使用的灵活性。缓冲区必须足够小，使其他应用有足够的主存满足需要。但是，即使其他应用并未运行，数据库系统也不能利用所有可用内存。同样地，非数据库应用也不能使用为数据库缓冲区保留的那部分主存，即使数据库缓冲区的一些页并未使用。

2. 数据库系统在操作系统提供的虚拟内存中实现其缓冲区。因为操作系统知道系统中的所有进程的内存需求，所以最好由它决定哪个缓冲块在什么时候必须强制输出到磁盘中。但是，为保证 16.5.1 节讲到的先写日志要求，不应由操作系统自己写数据库缓冲页，而应由数据库系统强制输出缓冲块。数据库系统在将相关日志记录写入稳定存储器后，强制输出缓冲块至数据库中。

[741]

遗憾的是，几乎所有当前的操作系统都完全控制虚拟内存。操作系统保留磁盘空间以存储当前不在主存的那些虚拟内存页，这些空间称为**交换区**（swap space）。如果操作系统决定输出一个块 B_x，该块就输出到磁盘交换区中，并且没有办法让数据库系统控制缓冲块的输出。

因此，如果数据库缓冲区在虚拟内存中，数据库文件和虚拟内存中的缓冲区之间的数据传送必须由数据库系统管理，它可以实现前面提到的先写日志的要求。

这种方法可能导致额外的数据到磁盘的输出。如果块 B_x 由操作系统输出，则那个块不是输出到数据库中，而是输出到为操作系统的虚拟内存准备的交换区中。当数据库系统需要输出 B_x 时，操作系统可能需要先从它的交换区输入 B_x。因此，这里可能不止一次 B_x 输出，而可能需要两次 B_x 输出（一次是由操作系统进行，一次是由数据库系统进行）和一次 B_x 输入。

尽管两种方法都有一些缺点，但总要选择其一，除非操作系统设计成支持数据库日志的要求。

16.5.4 模糊检查点

16.3.6 节描述的检查点技术要求对数据库的所有更新在检查点进行过程中暂缓执行。若缓冲区中页的数量很大，则完成一个检查点的时间会很长，这会导致事务处理中不可接受的中断。

为避免这种中断，可以修改检查点技术，使之允许在 checkpoint 记录写入日志后，但在修改过的缓冲块写到磁盘前开始做更新。这样产生的检查点称为**模糊检查点**(fuzzy checkpoint)。

由于只有在写入 checkpoint 记录之后页面才输出到磁盘，因此系统有可能在所有页面写完之前崩溃。这样，磁盘上的检查点可能是不完全的。一种处理不完全检查点的方法是：将最后一个完全检查点记录在日志中的位置存在磁盘上固定的位置 last_checkpoint 上。系统在写入 checkpoint 记录时不更新该信息，而是在写 checkpoint 记录前，创建所有修改过的缓冲块的列表。只有在该列表中的所有缓冲块都输出到磁盘上以后，last_checkpoint 信息才会更新。 742

即使使用模糊检查点，正在输出到磁盘的缓冲块也不能更新，虽然其他缓冲块可以并发地更新。必须遵守先写日志协议，使得与一个块有关的(undo)日志记录在该块输出前已写到稳定存储器中。

16.6 非易失性存储器数据丢失的故障

到目前为止，我们只考虑了一种情况，就是故障导致了易失性存储器中的信息丢失，而非易失性存储器中的内容完整无损的情况。尽管导致非易失性存储器中内容丢失的故障很少，我们仍然需要准备好对这种类型的故障加以处理。本节只讨论磁盘存储器。这些讨论也适用于其他类型的非易失性存储器。

基本的方法是周期性地将整个数据库的内容**转储**(dump)到稳定存储器中，比如一天一次。例如，我们可以将数据库转储到一盘或多盘磁带。如果发生了一个导致物理数据库块丢失的故障，系统就可以用最近的一次转储将数据库复原到前面的一致状态。一旦复原完成，系统再利用日志将数据库系统恢复到最近的一致状态。

数据库转储的一种方法要求在转储过程中不能有事务处于活跃状态，并且必须执行一个类似于检查点的过程：

1. 将当前位于主存的所有日志记录输出到稳定存储器中。
2. 将所有缓冲块输出到磁盘中。
3. 将数据库的内容拷贝到稳定存储器中。
4. 将日志记录 < **dump** > 输出到稳定存储器中。

第 1、2 和 4 步对应于 16.3.6 节中检查点的那三个步骤。

为从非易失性存储器数据丢失中恢复，系统利用最近一次转储将数据库复原到磁盘中。然后，根据日志，重做最近一次转储后所做的所有动作。注意，这里不必执行任何 undo 操作。

在非易失性存储器部分故障的情况，例如单个块或少数块故障，只需要对那些块进行复原，并只对那些块进行重做。

数据库内容的转储也称为**归档转储**(archival dump)，因为我们可以将转储归档，以后可以用它们 743
来查看数据库的旧状态。数据库转储和缓冲区的检查点机制很类似。

大多数的数据库系统还支持 **SQL 转储**(SQL dump)，它将 SQL DDL 语句和 SQL insert 语句写到文件中，这些语句可以重执行，以重新创建数据库。当将数据移植到数据库的另一个实例中或数据库软件的另一个版本中时，这样的转储非常有用，因为在另外的数据库实例或数据库软件版本中，物理位置或布局可能是不同的。

这里描述的简单转储过程开销较大，原因有以下两个。首先，整个数据库都必须拷贝到稳定存储器中，这导致大量的数据传送。其次，由于在转储过程中要中止事务处理，这样就浪费了 CPU 周期。**模糊转储**(fuzzy dump)机制对此作了改进，它允许转储过程中事务仍是活跃的。这类似于模糊检查点机制；详细描述可参见文献注释。

16.7 锁的提前释放和逻辑 undo 操作

在事务处理中使用到的任何索引，例如 B^+ 树，都可以当作一般的数据来对待，但是为了提高并发度，我们可以采用 15.10 节描述的 B^+ 树并发控制算法，允许按非两阶段的方式提前释放锁。作为提前释放锁的结果，有可能一个 B^+ 树结点的值被一个事务 T_1 主新过，插入项 (V, R_1)，然后又被另一个

事务 T_2 更新，在同一个结点中插入项 (V_2, R_2)，甚至在 T_1 完成执行之前就移动项 (V_1, R_1)。[⊖] 在这一点上，我们不能通过用 T_1 执行插入前的旧值代替结点的内容来撤销事务 T_1，因为这也会撤销事务 T_2 所执行的插入；事务 T_2 可能仍然会提交（或者可能已经提交了）。在这个例子中，对插入 (V_1, R_1) 的影响进行撤销的唯一办法是执行一个对应的删除操作。

本节其余部分讨论如何扩充 16.4 节讲的恢复算法，来支持锁的提前释放。

16.7.1 逻辑操作

插入和删除操作是一类操作的例子，它们需要逻辑 undo 操作，因为它们提前释放锁；我们把这样的操作称作**逻辑操作**（logical operation）。这类提前的锁释放不仅对索引很重要，而且对于其他频繁存取和更新的系统数据结构上的操作也很重要；这样的例子包括追踪包含一个关系中诸记录的块、一个块中的空闲空间、一个数据库中的空闲块的数据结构。如果在这样的数据结构上执行操作后不提前释放锁，事务就趋向于是顺序执行的，影响系统性能。

744 冲突可串行化理论向操作做了扩展，基于什么操作与什么其他操作有冲突。例如，如果 B⁺ 树的两个插入操作插入不同的键值，则它们互相不冲突，即使它们都更新相同的索引页中互相重叠的区域。但是，如果使用相同的键值，则插入和删除操作与其他的插入和删除操作冲突，也与读操作冲突。关于这一话题，请阅读文献注解以获取更多的信息。

操作在执行时需要获得低级别的锁，操作完成就释放锁；然后相应的事务必须按两阶段方式保持高级别的锁，以防止并发事务执行冲突动作。例如，当在一个 B⁺ 树页面上执行插入操作时，就获得在该页面上的一个短时间的锁，从而允许该页中的项在插入过程中在页内移动；一旦页面更新完成，就释放这个短时间的锁。这样的提前释放锁使得第二次插入可以在同一页面上执行。但是，每个事务必须获得在插入或删除的键值上的锁，并按两阶段方式持有锁，以防止并发的事务在相同的键值上执行冲突的读、插入、或删除操作。

一旦释放了低级别的锁，就不能用更新的数据项的旧值来对操作进行撤销，而必须通过执行一个补偿操作来撤销；这样的操作称作**逻辑 undo 操作**（logical undo operation）。重要的是，在操作中获得的低级别的锁要足以执行后来对操作的逻辑 undo，理由在 16.7.4 节中讲。

16.7.2 逻辑 undo 日志记录

要允许操作的逻辑 undo，在执行修改索引的操作之前，事务创建一个 $< T_i, O_j,$ operation-begin $>$ 日志记录，其中 O_j 是该操作实例的唯一标识。[⊖] 当系统执行这个操作时，它为这个操作所做的所有更新按正常方式创建更新日志记录。于是，对于该操作所做的所有更新，通常的旧值和新值信息照常写出；在该操作完成之前事务需要回滚的情况下，需要旧值信息。当操作结束时，它写一个形如 $< T_i, O_j,$ operation-end, $U >$ 的 **operation-end** 日志记录，其中 U 表示 undo 信息。

例如，如果操作往 B⁺ 树中插入一个项，则 undo 信息 U 会指出要执行删除操作，并指明是哪一棵 B⁺ 树，以及从树中删除哪个项。记关于操作的这类信息的日志称作**逻辑日志**（logical logging）。与此相反，记关于旧值和新值信息的日志称作**物理日志**（physical logging），对应的日志记录称作**物理日志记录**（physical logging record）。

请注意，在上述机制中，逻辑日志仅用于撤销，不用于重做；redo 操作全部使用物理日志记录来
745 执行。这是因为系统故障之后数据库状态可能反映一个操作的某些更新，而不反映其他操作的更新，这依赖于在故障之前哪些缓冲块已写到磁盘。像 B⁺ 树这样的数据结构会处于不一致的状态，逻辑 redo 和逻辑 undo 操作都不能在不一致状态的数据结构上执行；要执行逻辑 redo 或 undo，磁盘上的数据库状态必须是**操作一致**（operation consistent）的，即不应该有任何操作的部分影响。然而，正如我们将要看到的，在逻辑 undo 操作执行之前，恢复机制的重做阶段的物理 redo 处理以及使用物理日志记录的 undo

---

⊖ 请回忆，一个项包括一个键值和一个记录标识，或者在 B⁺ 树文件结构树叶层次的情况下包括一个键值和一个记录。

⊖ operation-begin 日志记录在日志中的位置可以用作唯一标识。

处理确保被一个逻辑 undo 操作存取的数据库中的部分处于一个操作一致的状态。

如果一个操作在一行上执行多次与执行一次结果相同，则称该操作是幂等的(idempotent)。往 B+ 树中插入一个项这样的操作可能不是幂等的，因此恢复算法必须保证已经执行过的操作不会再执行。与此相反，物理日志记录是幂等的，因为无论所记录的更新执行一次或多次，对应的数据项都会是同样的值。

16.7.3 有逻辑 undo 的事务回滚

当回滚事务 T_i 时，从后往前扫描日志，对事务 T_i 的日志记录做如下处理：

1. 在扫描中遇到的物理日志记录像以前描述的那样处理，除了下面马上要简短描述的需跳过的那些记录。使用由操作产生的物理日志记录对不完全的逻辑操作进行撤销。

2. 由 **operation-end** 标记的已完成的逻辑操作的回滚与此不同。一旦系统发现一个 $<T_i, O_j,$ **operation-end**, $U>$ 日志记录，就做以下特殊动作：

a. 它通过使用日志记录中的 undo 信息 U 来回滚该操作。在对操作的回滚中，它将所执行的更新记入日志，就像操作首次执行时进行的更新一样。

在操作回滚的最后，数据库系统不产生 $<T_i, O_j,$ **operation-end**, $U>$ 日志记录，而是产生 $<T_i, O_j,$ **operation-abort** $>$ 日志记录。

b. 现在对日志的反向扫描的继续进行，系统跳过事务 T_i 的所有日志记录，直至遇到 $<T_i, O_j,$ **operation-begin** $>$ 日志记录。

请注意，系统在回滚中将所执行的更新的物理 undo 信息记入日志，而不是使用一个只读的补偿日志记录。这是因为在逻辑 undo 进行的过程中可能发生系统崩溃，当恢复时系统必须完成逻辑 [746] undo；要做到这一点，重启恢复将使用物理 undo 信息撤销早先的 undo 的部分影响，然后再重新执行逻辑 undo。

还请注意，在回滚中当遇到 **operation-end** 日志记录时跳过物理日志记录能保证，一旦操作完成，物理日志记录中的旧值就不会用来进行回滚。

3. 如果系统遇到一个 $<T_i, O_j,$ **operation-abort** $>$ 日志记录，它就跳过前面所有的记录(包括 O_j 的 **operation-end** 记录)，直至它找到 $<T_i, O_j,$ **operation-begin** $>$ 日志记录。

仅在要回滚的事务事先已经部分回滚的情况下才会遇到 **operation-abort** 日志记录。回忆一下，逻辑操作可能不是幂等的，因此逻辑 undo 操作不能多次执行。在前面的回滚过程中发生崩溃，事务已经部分回滚的情况下，前面的日志记录必须跳过，以防止同一操作的多次回滚。

4. 与前面一样，当遇到 $<T_i,$ start $>$ 日志记录时，事务回滚就完成了，系统往日志中添加一个 $<T_i,$ abort $>$ 日志记录。

如果在一个逻辑操作进行的过程中发生故障，则当事务回滚时不会找到该操作的 operation-end 日志记录。但是，对于该操作所执行的每一个更新，在日志中都有其 undo 信息，是以物理日志记录中的旧值的形式出现的。物理日志记录将用来回滚未完成的操作。

现在假设当系统崩溃发生时一个 undo 操作正在进行中，如果崩溃发生时一个事务正在回滚，就可能发生这样的事情。于是就会发现在 undo 操作中所写的物理日志记录。使用这些物理日志记录就能撤销此部分的 undo 操作自身。继续进行日志的反向扫描，会遇到原来操作的 operation-end 日志记录，于是会再次执行 undo 操作。使用物理日志记录回滚早先的 undo 操作的部分影响使数据库进入一个一致状态，从而使得逻辑 undo 操作可以再次执行。

图 16-6 显示了由两个事务产生的日志的例子，这些事务对一个数据项的值进行增加或减少。事务 T_2 在操作 O_1 完成后对数据项 C 上锁的提前释放使得事务 T_1 能够用 O_2 更新该数据项，甚至不用等到事务 T_2 完成，但这就使逻辑 undo 成为必要。逻辑 undo 需要对数据项增加或减少一个值，而不是恢复数据项的旧值。

图 16-6 中的注释说明，在操作完成之前，回滚可以执行物理 undo；在操作完成并释放低级别锁之后，回滚必须通过减少或增加一个值来执行，而不是恢复旧值。在图 16-6 中的例子中，T_2 通过往 C 中增加 100 来回滚操作 O_1；与此相反，对于数据项 B(它没有涉及锁的提前释放)，进行了物理的 undo。请注意，T_1 对 C 执行了一个更新并提交了，它的更新 O_3 在 O_1 的撤销之前执行，往 C 中增加了 200。

这个更新保持下来，尽管 O_1 撤销了。

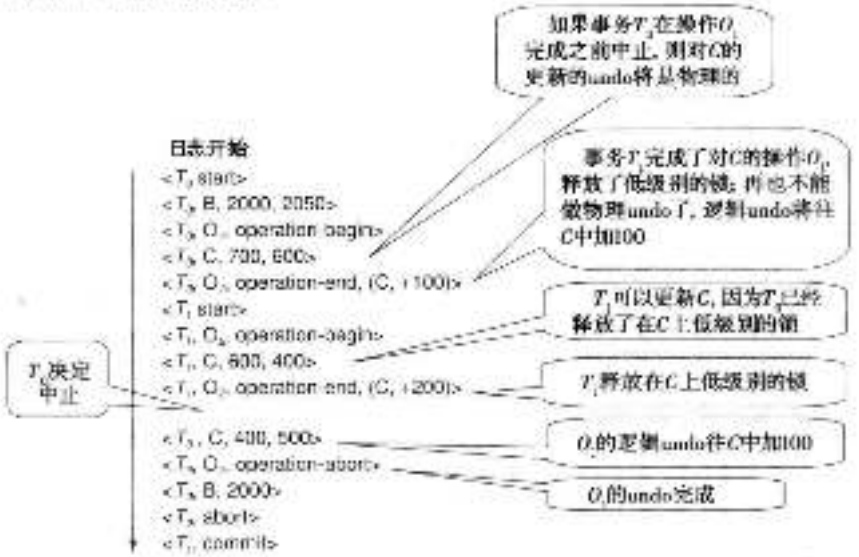


图 16-6 有逻辑 undo 操作的事务回滚

图 16-7 显示了一个使用逻辑 undo 日志从系统崩溃中恢复的例子。在这个例子中，在检查点时，事务 T_1 是活跃的，正在执行操作 O_4。在重做阶段，O_4 在检查点日志记录之后的动作重做。当系统崩溃时，T_2 正在执行操作 O_3，但是操作是不完全的。在重做阶段结束时，undo-list 包含 T_1 和 T_2。在撤销阶段，使用物理日志记录中的旧值对操作 O_3 进行撤销，将 C 置成 400；使用 redo-only 日志记录将这一操作记到日志中。然后遇到 T_2 的 **start** 记录，导致将 <T_2, **abort**> 记到日志中，并将 T_2 从 undo-list 中去掉。

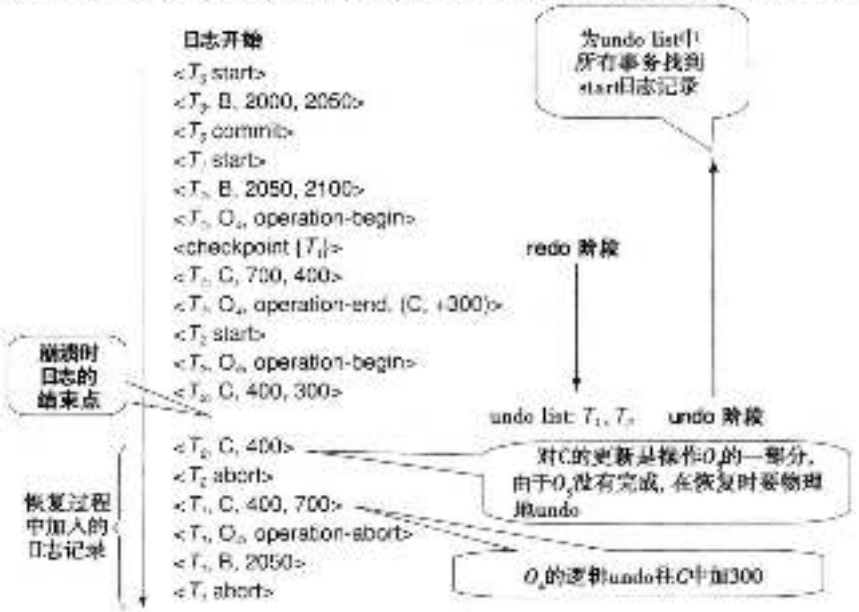


图 16-7 有逻辑 undo 操作的故障恢复动作

扫描中遇到的下一个日志记录是 O_4 的 operation-end 记录；通过给 C 的值增加 300 来执行这个操作的逻辑 undo，（这件事是物理地记了日志的），然后增加一个 O_4 的 operation-abort 日志记录。跳过作为 O_4 的一部分的物理日志记录，直至遇到 O_4 的 operation-begin 日志记录。在这个例子中，没有夹在中间的其他日志记录，但一般说来，在我们到达 operation-begin 日志记录之前可能遇到其他事务的日志记

录；当然这样的日志记录不应该就这(除非它们是相应事务已完成操作的一部分，并且算法跳过这些记录)。在遇到 O_4 的 operation-begin 日志记录之后，又遇到 T_1 的一个物理日志记录，对它物理地回滚。最后遇到 T_1 的 **start** 日志记录；这导致往日志中添加一个 $<T_1$，**abort**$>$ 日志记录，并且 T_1 被从 undo-list 中删掉。这时候 undo-list 为空，撤销阶段完成了。

16.7.4 逻辑 undo 中的并发问题

正如前面已经提到过的，操作中获取的低级别的锁要足以支持后来对该操作的逻辑 undo 的执行。这一点很重要；否则在正常处理中的并发操作可能导致撤销阶段中的问题。例如，假设事务 T_1 的操作 O_1 的逻辑 undo 与并发执行的事务 T_2 的操作 O_2 会在数据项层次上冲突，O_2 没完成时 O_1 完成了。还假设系统崩溃时两个事务都没提交。O_2 的物理更新日志记录可能出现在 O_1 的 operation-end 记录之前或之后，在恢复过程中 O_1 的逻辑 undo 中所做的更新可能会完全地或部分地被 O_2 的物理 undo 中写的旧值所覆盖。如果 O_1 已获得了 O_1 逻辑 undo 所需的所有低级别锁，上述问题就不会发生，因为这样的话，就不会有此类并发执行的 O_2 了。 747 749

如果原来的操作和它的逻辑 undo 操作都访问同一个页面(这样的操作称作物理逻辑操作，在 16.8 节讨论)，则上述封锁要求容易满足。否则，在决定需要获得什么样的低级别锁时，要考虑具体操作的细节。例如，B$^+$ 树上的更新操作可以获得一个在根结点上的短时间锁，以确保操作的串行执行。关于采用逻辑 undo 日志的 B$^+$ 树并发控制和恢复，请参看文献注解。在文献注解中还可以看到一个另外的方法，称作多级恢复，该方法放松对锁的要求。

16.8 ARIES*

ARIES 恢复方法是恢复方法的技术发展水平的最佳例子。16.4 节讨论的恢复技术和 16.7 节描述的逻辑 undo 日志技术是模仿 ARIES 设计的，但为了突出关键概念和便于理解，它已大大简化。不同的是，ARIES 使用了一些减少恢复时间、以及减少检查点开销的技术。特别地，ARIES 能够避免重做许多已重做过的日志中记录的操作，并减少日志信息量。其代价是大大增加了复杂度，但物有所值。

与我们先前给出的恢复算法的主要区别是，ARIES：

1. 使用一个**日志顺序号**(Log Sequence Number，LSN)来标识日志记录，并将 LSN 存储在数据库页中，来标识哪些操作已经在一个数据库页上实施过了。

2. 支持**物理逻辑** redo(physiological redo)操作，它是物理的，因为受影响的页从物理上标识出来，但在页内它可能是逻辑的。

例如，如果采用分槽的页结构(见 10.5.2 节)，从页中删除一条记录会导致该页中的许多记录被调整。采用物理 redo 日志，必须为该页中受调整影响的每个字节记录日志，而采用物理逻辑日志，可以记录下删除操作，其结果是日志记录会小得多。重做该删除操作将删除那条记录并根据需要调整其他记录。

3. 使用**脏页表**(dirty page table)来最大限度地减少恢复时不必要的重做。正如前面所说过的，脏页是那些在内存中已更新，而磁盘上的版本还未更新的页。

4. 使用**模糊检查点**机制，只记录脏页信息和相关的信息，甚至不要求将脏页写到磁盘。它不是在 750 检查点时将脏页写入磁盘，而是连续地在后台刷新脏页面。

本节剩下部分将给出 ARIES 概览，ARIES 的完全描述请参见文献注解。

16.8.1 数据结构

ARIES 中每个日志记录都有一个唯一标识该记录的日志顺序号(LSN)，该标号概念上只是一个逻辑标识，日志中记录产生得越晚，其标号数字越大。实际上，LSN 是以一种可以用来定位磁盘上的日志记录的方式产生的。典型地，ARIES 将一个日志分为多个日志文件，其中每个文件有一个文件号。当一个日志文件增长到某个限度，ARIES 将后面的日志记录添加到一个新的日志文件中；该新日志文件的文件号比前个记录一文件的大 1。这样 LSN 由一个文件号以及在该文件中的偏移量组成。

每一页也维护一个叫页日志顺序号(PageLSN)的标识。每当一个更新操作(无论物理的还是物理逻辑的)发生在某页上时，该操作将其日志记录的 LSN 存储在该页的 PageLSN 域中。在恢复的撤销阶段，

LSN 值小于或等于该页的 PageLSN 值的日志记录将不在该页上执行，因为它的动作已经反映在该页上了。稍后我们会讲到，通过结合将记录 PageLSN 作为检查点过程的一部分的机制，ARIES 甚至能够避免读其日志记录的操作已经在磁盘上反映的许多页。这样，恢复时间大量减少了。

要在物理逻辑 redo 操作时保证幂等性，PageLSN 是必不可少的，因为对一已实施物理逻辑 redo 类作的页重新实施会造成对页的错误更改。

在更新正在执行时，页不应往磁盘上写，因为在磁盘上页的部分更新状态下，物理逻辑操作不能重做。因此，ARIES 在缓冲页上加闩锁以阻止它们在正更新时被写往磁盘。只有在更新完成，以及更新的日志记录已写入日志之后，才释放该缓冲页闩锁。

每个日志记录也包含同一事务的前一日志记录的 LSN，该值存放在 PrevLSN 字段中，它使得一个事务的日志记录能够由后往前提取，而不必读整个日志。在事务回滚过程中会产生一些特殊的 redo-only 日志记录，在 ARIES 中称为补偿日志记录(Compensation Log Record，CLR)，它和我们前面讲的恢复机制中的 redo-only 日志记录的作用相同。而且 CLR 还起到该机制中 operation-abort 日志记录的作用。CLR 有一个额外的字段，叫做 UndoNextLSN，记录当事务被回滚时，日志中下一个需要 undo 的日志的 LSN。该字段和我们早先讲的恢复机制中的 operation-abort 日志记录中的操作标识作用相同，它有助于跳过那些已经回滚的日志记录。

脏页表(DirtyPageTable)包含一个在数据库缓冲区中已更新的页的列表。它为每一页保存其 PageLSN 和一个称为 RecLSN 的字段，其中 RecLSN 用于标识已经实施于该页的磁盘上的版本的日志记录。当一页插入到脏页表(当它首次在缓冲池中修改)时，RecLSN 的值被设置成日志的当前末尾。只要页被写入磁盘，就从脏页表中移除该页。

检查点日志记录(checkpoint log record)包含脏页表和活动事务的列表。检查点日志记录也为每个事务记录其 LastLSN，即该事务所写的最后一个日志记录的 LSN。磁盘上一个固定的位置记录最后一个(完整的)检查点日志记录的 LSN。

图 16-8 描述了 ARIES 中使用的一些数据结构。图 16-8 中显示的日志记录前面加上了其 LSN 作为前缀；在实际实现中，这可能并不显式地存储，而是通过在日志中的位置就能推断出来。日志记录中的数据项标识分两部分显示，例如 4894.1：第一部分标明页码，第二部分标明页中的一条记录(假定采用分槽的页结构)。请注意对日志的显示是最新的记录在顶部，磁盘上较老的日志记录在图 16-8 中较低的位置显示。

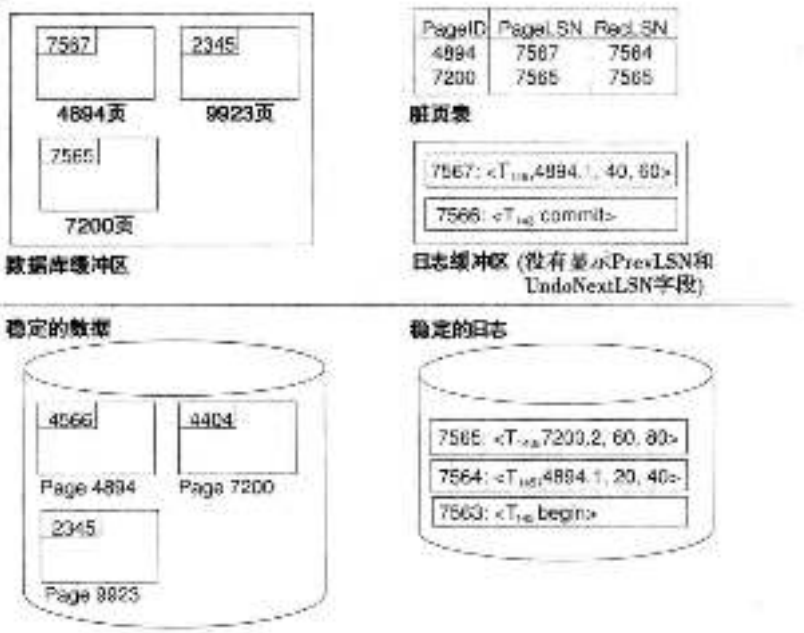


图 16-8 ARIES 中使用的数据结构

每个页面(无论在缓冲区中还是在磁盘中)有一个 PageLSN 字段。你可以验证，最后一个日志记录是对页面4894进行更新，它的 LSN 是7567。通过将缓冲区中页面的 PageLSN 与稳定存储器中对应页面的 PageLSN 进行比较，你可以看到脏页表包含了关于缓冲区中自从从稳定存储器中取来后已修改过的所有页面的条目。脏页表中的 RecLSN 条目反映当页面被加到脏页表中时日志末端的 LSN，它应该大于或等于稳定存储器中该页的 PageLSN。

16.8.2 恢复算法

ARIES 从系统崩溃中恢复的过程经历三个阶段。

- **分析阶段**(analysis pass)：这一阶段决定哪些事务要撤销，哪些页在崩溃时是脏的，以及重做阶段应从哪个 LSN 开始。
- **redo 阶段**(redo pass)：这一阶段从分析阶段决定的位置开始，执行重做，重复历史，将数据库恢复到发生崩溃前的状态。
- **undo 阶段**(undo pass)：这一阶段回滚在发生崩溃时那些不完全的事务。

16.8.2.1 分析阶段

分析阶段找到最后的完整检查点日志记录，并从该记录读入脏页表。它然后将 RedoLSN 设置为脏页表中页的 RecLSN 的最小值，如果没有脏页，它就将 RedoLSN 设置为检查点日志记录的 LSN。重做阶段从 RedoLSN 开始扫描日志，该点之前的日志记录已经反映到磁盘上的数据库页中。分析阶段将要撤销的事务列表 undo-list 初始设置为检查点日志记录中的事务列表，分析阶段也为 undo-list 中的每一个事务从检查点日志记录中读取其最后一个日志记录的 LSN。

分析阶段从检查点继续正向扫描，只要找到一个不在 undo-list 中的事务的日志记录，就把该事务添加到 undo-list 中。只要找到一个事务的 end 日志记录，就把该事务从 undo-list 中删除。到分析阶段结束时，所有留在 undo-list 中的事务将在后面的撤销阶段中回滚。分析阶段也记录 undo-list 中每一个事务的最后一个记录，它在撤销阶段中有用。

一旦分析阶段发现一个在页上更新的日志记录，它还更新脏页表。如果该页不在脏页表中，分析阶段就将它添加到脏页表，并设置该页的 RecLSN 为该日志记录的 LSN。 751 753

16.8.2.2 重做阶段

重做阶段通过重演所有没有在磁盘页中反映的动作来重复历史。重做阶段从 RedoLSN 开始正向扫描日志，只要它找到一个更新日志记录，它就执行如下动作：

1. 如果该页不在脏页表中，或者该更新日志记录的 LSN 小于脏页表中该页的 RecLSN，重做阶段就跳过该日志记录。

2. 否则重做阶段就从磁盘调出该页，如果其 PageLSN 小于该日志记录的 LSN，就重做该日志记录。

注意如果上述两个测试中的任何一个是否定的，那么该日志记录的作用已经反映到页面中；否则日志记录的作用就还没有反映到页面中。由于 ARIES 允许非幂等性的物理逻辑日志记录，因此如果一个日志记录的影响已经反映到页面中，该日志记录就不应该重做。如果第一个测试是否定的，那么甚至不必从磁盘取出该页去检查它的 PageLSN。

16.8.2.3 撤销阶段和事务回滚

撤销阶段比较直截了当。它对日志进行一遍反向扫描，对 undo-list 中的所有事务进行撤销。撤销阶段仅检查 undo-list 中事务的日志记录；用分析阶段所记录的最后一个 LSN 来找到 undo-list 中每个日志的最后一个日志记录。

每当找到一个更新日志记录，就用它来执行一个 undo(无论是在正常处理过程的事务回滚中，还是在重启动的撤销阶段中)。撤销阶段产生一个包含 undo 执行动作(必须是物理逻辑的)的 CLR，并将该 CLR 的 UndoNextLSN 设置为该更新日志记录的 PrevLSN 值。

如果遇到一个 CLR，则它的 UndoNextLSN 值指明了该事务需要 undo 的下一个日志记录的 LSN；该事务的在其后面的日志记录已经回滚了。除 CLR 之外的那些日志记录，PrevLSN 字段指明该事务需要 undo 的下一个日志记录的 LSN。在撤销阶段中的每一站中，要执行的下一个日志记录是 undo-list 中所有事务的下一个日志记录 LSN 中最大的一个。

图 16-9 描述了 ARIES 基于一个样例日志所执行的恢复动作。假设磁盘上最后一个完整的检查点指针指向 LSN 为 7568 的检查点日志记录。在图 16-9 中用箭头指示日志记录中的 PrevLSN 值，用虚线箭头指示那个 LSN 为 7565 的补偿日志记录中的 UndoNextLSN 值。分析阶段从 LSN 7568 开始，当分析阶段完成时，RedoLSN 为 7564。于是，重做阶段必须从 LSN 为 7564 的日志记录开始。请注意，该 LSN 小于检查点日志记录的 LSN，因为 ARIES 检查点算法不将修改过的页面刷新到稳定存储器。在分析阶段结束时脏页表会包含检查点日志记录中的页面 4894 和 7200，以及页面 2390，它是被 LSN 为 7570 的日志记录修改过的。在本例中分析阶段结束时，需要撤销的事务列表仅包括 T_{145}。

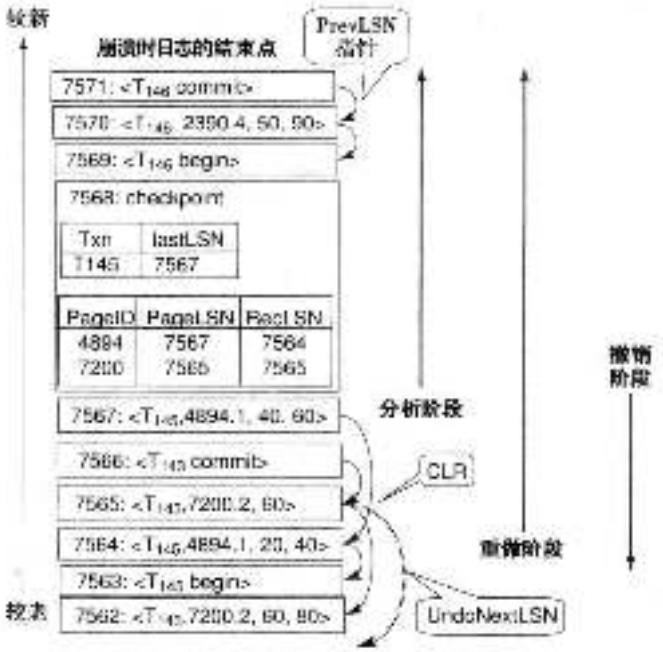


图 16-9 ARIES 中的恢复动作

上述例子的重做阶段从 LSN 7564 开始，对其页面出现在脏页表中的日志记录进行重做。撤销阶段仅需要对事务 T_{145} 进行撤销，因此从它的 LastLSN 值 7567 开始，反向扫描，直至在 LSN 7563 处遇到 < T_{145} start > 记录。

16.8.3 其他特性

ARIES 提供的其他关键特性包括以下几方面。

- **嵌套的顶层动作**(nested top action)：ARIES 允许对即使事务回滚也不应该撤销的那些操作记日志；例如，如果一个事务分配一个页面给一个关系，那么即使该事务回滚，此页面分配也不能撤销，因为其他事务可能已经存储记录在这个页面上了。这样的不应该回滚的操作称作嵌套的顶层动作。这样的操作可以模拟为其 undo 动作什么都不做的操作。在 ARIES 中，这样的操作通过创建一个虚拟的 CLR 来实现，设置其 UndoNextLSN，使得事务回滚跳过由此操作产生的日志记录。
- **恢复的独立性**(recovery independence)：有些页能够独立于其他页进行恢复，以便它们甚至能够在别的页正在恢复时使用。如果一张磁盘的某些页出错，它们无须停止其他页上的事务处理就能恢复。
- **保存点**(savepoint)：事务能够记录保存点，并能部分回滚到一个保存点。这对于死锁处理特别有用，因为事务能够回滚到某个点来允许释放必要的锁，然后从那个点重新开始。

  程序员还可以使用保存点来部分地撤销一个事务，然后继续执行；对于处理在事务执行中发现的某些类型的错误，这个方法会是很有用的。

- **细粒度的封锁**(fine-grained locking)：ARIES恢复算法可以和允许在索引中元组级封锁而不是页级封锁的索引并发控制算法一起使用。这大大地提高了并发性。
- **恢复最优化**(recovery optimization)：重做表能用于在重做时预先提取页，而不是只在系统找到一个要应用到页的日志记录时才提取该页。重做也可能是不按顺序的：当页正在被从磁盘提取时，重做可以排迟，在页被取出来后，重做再执行。同时，其他日志记录能够继续处理。

总之，ARIES算法代表恢复算法的最新技术发展水平，它综合运用了为提高并发度、减少日志开销和缩短恢复时间所设计的各种优化技术。

16.9 远程备份系统

传统的事务处理系统是集中式或客户/服务器模式的系统。这样的系统易受自然灾难(如火灾、洪水和地震)的攻击。要求事务处理系统无论系统故障还是自然灾难都能运行的需求日益高涨。这种系统必须提供**高可用性**(high availability)：即系统不能使用的时间必须非常地短。

我们可以这样来获得高可用性，在一个站点执行事务处理，称为**主站点**(primary site)，使用一个**远程备份**(remote backup)站点，这里有主站点所有数据的备份。远程备份站点有时也叫**辅助站点**(secondary site)，重着重新在主站点上执行，远程站点必须保持与主站点同步。我们通过发送主站点的所有日志记录到远程备份站点来达到同步。远程备份站点必须物理地与主站点分离——例如，我们可以将它放在一个不同的州，这样发生在主站点的灾难不会破坏远程备份站点。图16-10显示了远程备份系统的体系结构。 756

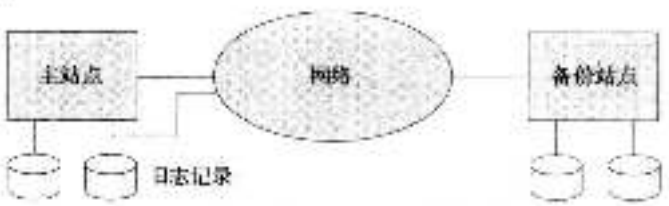


图16-10 远程备份系统的体系结构

当主站点发生故障，远程备份站点就接管处理。但它首先使用源于主站点的(主许已过时的)数据拷贝，以及收到的来自主站点的日志记录执行恢复。事实上，远程备份站点执行的恢复动作就是主站点要恢复时需执行的恢复动作。对于标准的恢复算法增加修改，就可用于远程备份站点的恢复。一旦恢复执行完成，远程备份站点就开始处理事务。

即使主站点的数据全部丢失，系统也能恢复，因此，相对于单站点系统而言，系统的可用性大大地提高了。

在设计一个远程备份系统时有几个问题必须考虑。

- **故障检测**(detection of failure)。对于远程备份系统而言，检测什么时候主站点发生故障是很重要的。通信线路故障会使远程备份站点误以为主站点已发生故障。为避免这个问题，我们在主站点和备份站点之间维持几条具有独立故障模式的通信线路。例如，可以使用几种互相独立的网络连接，或许包括通过电话线路的调制解调器连接。这些连接可能由那些能通过电话系统进行通信的操作人员的手工干预来提供支持。
- **控制权的移交**(transfer of control)。当主站点发生故障时，备份站点就接管处理并成为新的主站点。当原来的主站点恢复后，它可以作为远程备份站点工作，抑或再次接管并作为主站点。在任意情况下，原主站点都必须收到一份在它故障期间备份站点上所执行更新的日志。

  移交控制权最简单的办法是原主站点从原备份站点收到redo日志，并将它们应用到本地以赶上更新。然后原主站点就可以作为远程备份站点工作，如果控制权必须回传，原备份站点可以假装发生故障，导致原主站点重新接管。 757
- **恢复时间**(time to recovery)。如果远程备份站点上的日志增长到很大，恢复就会花很长时间。远程备份站点可以周期性地处理它收到的redo日志，并执行一个检查点，从而日志中早期的部分

可以删除。这样，远程备份站点接管的延迟显著缩短。

采用热备份(hot-spare)配置可使备份站点几乎能在一瞬间接管。在该配置中，远程备份站点不断地处理到达的 redo 日志记录，在本地进行更新。一旦检测到主站点发生故障，备份站点就通过回滚未完成的事务来完成恢复；然后就做好处理新事务的准备。

- **提交时间**(time to commit)。为保证已提交事务的更新是持久的，只有在其日志记录到达备份站点之后才能宣称该事务已提交。该延迟会导致等待事务提交的时间变长，因此某些系统允许较低程度的持久性。持久性的程度可以按如下分类：
  - **一方保险**(one-safe)。事务的提交日志记录一写入主站点的稳定存储器，事务就提交。

    这种机制的问题是，当备份站点接管处理时，已提交事务的更新可能还没有在备份站点执行，这样，该更新好像丢失了。当主站点恢复后，丢失的更新不能直接并入，因为它可能与后来在备份站点上执行的更新相冲突。因此，可能需要人工干预来使数据库回到一致状态。
  - **两方强保险**(two-very-safe)。事务的提交日志记录一写入主站点和备份站点的稳定存储器，事务就提交。

    这种机制的问题是，如果主站点或备份站点其中的一个停工，事务处理就无法进行。因此，虽然丢失数据的可能性很小，但其可用性实际上比单站点的情况还低。
  - **两方保险**(two-safe)。如果主站点和备份站点都是活跃的，该机制与两方强保险机制相同。如果只有主站点是活跃的，事务的提交日志记录一写入主站点的稳定存储器事务，就允许它提交。

    这一机制提供了比两方强保险机制更好的可用性，同时避免了一方保险机制出现的事务丢失问题。它导致比一方保险机制较慢的提交，但总的来说利多于弊。

一些商用共享磁盘系统提供一定级别的容错，成为集中式系统和远程备份系统的折中。在这些商用系统中，CPU 故障不会导致系统故障，而是由其他 CPU 接管并执行恢复。恢复动作包括回滚正在故障 CPU 上执行的事务以及恢复这些事务所持有的锁。由于数据在共享磁盘上，因此不需要日志记录的传送。但我们应该保护数据，防止磁盘故障造成的丢失，例如，使用 RAID 磁盘组织。

758

另一种达到高可用性的可选方法是使用分布式数据库，将数据复制到不止一个站点。此时事务更新任何一个数据项，都要求更新其所有副本。第 19 章将讨论分布式数据库，包括复制。

16.10 总结

- 计算机系统与其他机械或电子设备一样容易发生故障。造成故障的原因有很多，包括磁盘故障、电源故障和软件错误。这些情况造成了数据库系统信息的丢失。
- 除系统故障外，事务也可能因各种原因造成失败，例如破坏了完整性约束或发生死锁。
- 数据库系统的一个重要组成部分就是恢复机制，它负责检测故障以及将数据库恢复至故障发生前的某一状态。
- 计算机中的各种存储器类型有易失性存储器、非易失性存储器和稳定存储器。易失性存储器(如 RAM)中的数据在计算机发生故障时会丢失。非易失性存储器(如磁盘)中的数据在计算机发生故障时一般不丢失，只是偶尔由于某些故障如磁盘故障才会丢失。稳定存储器中的数据从不丢失。
- 必须能联机访问的稳定存储器用镜像磁盘或 RAID 的其他形式模拟，它们提供冗余数据存储。脱机或归档稳定存储器可靠是数据的多个磁带备份，并存放在物理上安全的地方。
- 一旦故障发生，数据库系统的状态可能不再一致，即它不能反映数据库试图保存的现实世界的状态。为保持一致性，我们要求每个事务都必须是原子的。恢复机制的责任就是要保证原子性和持久性。
- 在基于日志的机制中，所有的更新都记入日志，并存放在稳定存储器中。当事务的最后一个日志记录，即该事务的 commit 日志记录，输出到稳定存储器时，就认为这个事务已提交。
- 日志记录包括所有更新过的数据项的旧值和新值。当系统崩溃后需要对更新进行重做时，就使用新值。如果在正常操作中事务中止，回滚事务所做的更新时需要用到旧值；在事务提交之前发生系统崩溃的情况下，回滚事务所做的更新也需要用到旧值。

759

- 在延迟修改机制中，事务执行时所有 write 操作要延迟到事务提交时才执行。那时，系统在执行延迟写中会用到日志中与该事务有关的信息。在延迟修改机制中，日志记录不需要包含已更新的数据项的旧值。
- 为减少搜索日志和重做事务的开销，我们可以使用检查点技术。
- 当前的恢复算法基于重复历史的概念。在恢复的重做阶段重新（自最后一个已完成的检查点以来）主备条件中所做的所有动作。重复历史的做法将系统状态恢复到系统崩溃之前最后一个日志记录输出到稳定存储器时的系统状态。然后从这个状态开始执行一个撤销阶段，反向处理未完成事务的日志记录。
- 不完全事务的撤销写中特殊的 redo-only 日志记录和一个 abort 日志记录。然后，就可以认为该事务已完成，不必再对它进行撤销。
- 在事务处理所基于的存储模型中，主存储器中有一个日志缓冲区、一个数据库缓冲区和一个系统缓冲区。系统缓冲区中有系统目标码页面和事务的局部工作区域。
- 恢复机制的高效实现需要尽可能减少向数据库和稳定存储器写回的数目。日志记录在开始时可以保存在易失性的日志缓冲区中，但是当下述情况之一发生时必须写到稳定存储器中：
  - 在 $<T_i$, **commit** > 日志记录可以输出到稳定存储器之前，与事务 T_i 相关的所有日志记录必须已经输出到稳定存储器中。
  - 在主存中的一个数据块输出到（非易失性存储器中的）数据库之前，与该块中的数据相关的所有日志记录必须已经输出到稳定存储器中。
- 当前的恢复技术支持高并发性封锁技术，例如用于 B+ 树并发控制的封锁技术。这些技术允许提前释放通过插入或删除这样的操作获得的低级别的锁，低级别的锁允许别的事务其他的这类操作可以执行。低级别的锁被释放之后，不能进行物理 undo，而需要进行逻辑 undo，例如，用删除来对插入做 undo。事务保持高级别的锁以确保并发的事务不会执行这样的动作。它可能导 760
  致一个操作的逻辑 undo 是不可能的。
- 为从灾域非易失性存储器中数据丢失的故障中恢复，我们必须周期性地将整个数据库的内容转储到稳定存储器中——例如每天一次。如果发生了导致物理数据库块丢失的故障，我们使用最近一次转储将数据库恢复至前面的某个一致状态。一旦完成该恢复，我们采用日志将数据库系统恢复至最当前的一致状态。
- ARIES 恢复机制是最新技术水平的机制，它支持一些提供更大并发性、削减日志开销和最小化恢复时间的特性。它也是基于重复历史的，并允许逻辑 undo 操作。该机制连续不断地清洗页，从而不需要在检查点时清洗所有页。它使用日志顺序号（LSN）来实现各种优化从而减少恢复所花时间。
- 远程备份系统提供了很高程度的可用性，允许事务处理即使在主站点遭受火灾、洪水或地震的破坏时也能继续。主站点上的数据和日志记录连续不断地备份到远程备份站点。如果主站点发生故障，远程备份站点就执行一定的恢复动作，然后接管事务处理。

术语回顾

- 恢复机制
- 故障分类
  - 事务故障
  - 逻辑错误
  - 系统错误
  - 系统崩溃
  - 数据传输故障
- 故障—停止假设
- 磁盘故障
- 存储器类型
  - 易失性存储器
  - 非易失性存储器
  - 稳定存储器
- 块
  - 物理块
  - 缓冲块
- 磁盘缓冲区
- 强制输出
- 基于日志的恢复
- 日志
- 日志记录
- 更新日志记录
- 延迟的修改
- 立即的修改
- 未提交的修改
- 检查点
- 恢复算法

- 重启动恢复
- 事务回滚
- 物理 undo
- 物理日志
- 事务回滚
- 检查点
- 重启动恢复
- 重做阶段
- 撤销阶段
- 重复历史
- 缓冲区管理
- 日志记录缓冲
- 先写日志(WAL)
- 强制日志
- 数据库缓冲
- 闩锁
- 操作系统与缓冲区管理
- 模糊检查点
- 锁的提前释放
- 逻辑操作
- 逻辑日志
- 逻辑 undo
- 非易失性存储器数据丢失
- 文档转储
- 模糊转储
- ARIES
  - 日志顺序号(LSN)
  - 页面顺序号(PageLSN)
  - 物理逻辑 redo
  - 补偿日志记录(CLR)
  - 脏页表
  - 检查点日志记录
  - 分析阶段
  - 重做阶段
  - 撤销阶段
- 高可用性
- 远程备份系统
  - 主站点
  - 远程备份站点
  - 辅助站点
- 故障检测
- 控制权移交
- 恢复时间
- 热备份配置
- 提交时间
  - 一方保险
  - 两方很保险
  - 两方保险

实践习题

16.1 请解释为什么 undo-list 中事务的日志记录必须由后往前进行处理，而 redo-list 中事务的日志记录则必须由前往后进行处理。

16.2 请解释检查点机制的目的。应该间隔多长时间做一次检查点？执行检查点的频率对以下各项有何影响？

- 无故障发生时的系统性能如何？
- 从系统崩溃中恢复所需的时间多长？
- 从介质(磁盘)故障中恢复所需的时间多长？

761
|
762

16.3 某些数据库系统允许系统管理员在正常日志(用于从系统崩溃中恢复)和归档日志(用于从介质(磁盘)故障中恢复)这两种日志形式间进行选择。采用 16.4 节的恢复算法，对于这两种情况的每一种，一个日志记录在什么时候可以删除？

16.4 请描述如何对 16.4 节的恢复算法进行修改，以实现保存点和执行对保存点的回滚。(保存点在 16.8.3 节描述。)

16.5 假设在数据库中使用延迟的修改技术。

a. 更新日志记录中的旧值还需要吗？为什么？

b. 如果没有将旧值存放在更新日志记录中，则显然无法对事务进行撤销。其结果是需要对恢复的重做阶段做什么样的修改？

c. 通过将更新过的数据项保存在事务的局部存储中，并且直接从数据库缓冲区中读没有更新过的数据项，可以实行延迟的修改。请给出如何高效地实现数据项的读，要保证事务看到它自己的更新。

d. 如果事务进行大量的更新，那么上述技术会有什么问题？

16.6 影子页技术需要对页表进行拷贝。假设页表表示成 B^+ 树形式。

a. 请说明如何在 B^+ 树的新拷贝和影子拷贝之间共享尽可能多的结点，假设更新仅对叶结点的项进行，并且没有插入和删除。

b. 即使做了上述优化，对于进行少量更新的事务，日志的方法仍然比影子拷贝的方法开销小得多。请解释为什么。

16.7 假设我们(错误地)修改了 16.4 节的恢复算法，对于事务回滚中执行的动作不记日志。当从系统崩溃中恢复时，于是早先已经回滚了的事务就会被包括在 undo-list 中，并且被再次回滚。请给出一个例子，说明在恢复的撤销阶段执行的动作如何会导致一个不正确的数据库状态。(提示：考虑已中止事务更新过，然后又被一个提交的事务更新的一个数据项。)

16.8 作为一个事务的结果分配给一个文件的磁盘空间不应该释放，即使该事务回滚了。请解释这是为什么，并解释 ARIES 如何保证这样的动作不回滚。 [763]

16.9 假设一个事务删除了一条记录，甚至在这个事务提交之前，所产生的自由空间就被分配给另一个事务插入的一条记录。

a. 如果第一个事务需要回滚的话，会产生什么问题？

b. 如果使用页级别的封锁，而不是元组级别的封锁，那么这还是个问题吗？

c. 如果支持元组级别的封锁，请解释如何通过在特殊的日志记录中记下提交后的动作，并在提交后执行它们，来解决这个问题。要确保在你给出的机制中这样的动作恰好执行一次。

16.10 请解释为什么交互式事务的恢复比批处理事务的恢复更难处理。是否有简单的方法处理该困难？（提示：考虑用于提取现金的自动取款机事务。）

16.11 有时事务在完成提交之后不得不撤销，因为它错误地执行了，比如银行出纳员的错误输入。

a. 举例说明采用通常的事务撤销机制来撤销这样一个事务会导致不一致状态。

b. 一个处理这种状况的途径是使整个数据库回到该错误事务提交前的某一状态（称为时点（point-in-time）恢复），该点之后提交的事务回滚其影响。

请对 16.4 节的恢复算法加以修改，使用数据库转储来实现时点恢复。

c. 时点之后的无错误事务可以从逻辑上重新执行，但不能使用它们的日志记录重新执行，为什么？

习题

16.12 从 I/O 开销的角度解释易失性存储器、非易失性存储器和稳定存储器三类存储器的区别。

16.13 稳定存储器是不可能实现的。

a. 请解释为什么。

b. 请解释数据库系统如何对待这个问题。 [764]

16.14 如果与某块有关的某些日志记录没有在该块输出到磁盘前输出到稳定存储器中，请解释数据库可能会如何被变得不一致。

16.15 请描述非强取的和强制的缓冲区管理策略的缺点。

16.16 物理逻辑 redo 日志可以显著减小日志开销，特别是对于分槽的页记录组织。请解释这是为什么。

16.17 请解释为什么逻辑 undo 日志广泛使用，而逻辑 redo 日志（除了物理逻辑 redo 日志）很少使用。

16.18 考虑图 16-5 中的日志。假设恰好在 < T_i abort > 日志记录写出之前系统崩溃。请解释在系统恢复时会发生什么。

16.19 假设有一个事务已运行了很长时间，但仅做了很少的更新。

a. 如果使用 16.4 节的恢复算法，该事务对恢复时间的影响是什么？如果用 ARIES 恢复算法呢？

b. 该事务对于删除老的日志记录的影响是什么？

16.20 考虑图 16-6 中的日志。假设在恢复中发生了崩溃，发生的时间恰好在操作 O_4 的 operation abort 日志记录写出之前。请解释系统再次恢复时会发生什么。

16.21 请对于下述情况，从开销的角度，对基于日志的恢复和影子拷贝机制进行比较：数据要加入到新分配的磁盘页中（换句话说，如果事务中止的话，没有旧值需要恢复）。

16.22 在 ARIES 恢复算法中：

- 如果在分析阶段开始时，某个页面不在检查点脏页表中，我们需要将任何 redo 记录应用于该页吗？为什么？
- RedoLSN 是什么，如何应用它来尽可能减少不必要的重做？

16.23 请解释系统崩溃和“灾难”的区别。

16.24 为以下各项需求，指出在远程备份系统中，持久性程度最合适的选择：

a. 必须避免数据丢失，但失去一起可用性可以接受。

b. 事务提交必须迅速完成，甚至可以冒在灾难发生时丢失一些已提交事务的风险。

c. 要求高可用性和持久性，但事务提交协议的较长运行时间是可以接受的。 [765]

16.25 Oracle 数据库系统使用 undo 日志记录来提供快照隔离模式下的数据库的一种实现技术。事务 T_i 所看

到的快照视图反映了当 T_i 开始时所有已提交事务的更新，以及 T_i 的更新；所有其他事务的更新对 T_i 是不可见的。

请描述一种缓冲区管理机制，它能向事务提供缓冲区的页面的一个快照。说明如何使用日志来生成快照视图的细节。你可以假设操作及其 undo 动作只影响一个单独的页面。

文献注解

Gray 和 Reuter[1993]是一本优秀的教科书，提供了关于恢复的信息资料，包括有趣的实现和历史细节。Bernstein 和 Goodman[1981]是早期关于并发控制和恢复的信息资料教材。

System R 关于恢复机制的概述由 Gray 等[1981]给出。关于数据库系统的各种恢复技术的指导性和概述性文章包括 Gray[1978]、Lindsey 等[1980]和 Verhofstad[1978]。检查检查点和影子转储的概念是在 Lindsey 等[1980]中阐述的。Haerder 和 Reuter[1983]提供了恢复原理的全面概述。

恢复方法的最新技术水平在 ARIES 方法中得到了最好的体现，在 Mohan 等[1992]和 Mohan[1990b]中描述。Mohan 和 Levine[1992]给出了 ARIES 的扩展 ARIES IM，使用逻辑 undo 日志来优化 B+ 树并发控制和恢复。ARIES 和它的几个变种用在好几个数据库产品中，包括 IBM DB2 和 Microsoft SQL Server。Oracle 的恢复在 Lahiri 等[2001]中描述。

索引结构的特殊恢复技术在 Mohan 和 Levine[1992]，以及 Mohan[1993]中作了阐述。Mohan 和 Narang[1994]描述了客户—服务器架构的恢复技术。Mohan 和 Narang[1992]描述了并行数据库体系架构的恢复技术。

Weikum[1991]描述了可串行性理论到一般化版本来操作进行中的短时间低级别锁，以及与长时间高级别锁的结合。在 16.7.3 节中，我们看到这样的需求：一个操作应该获得该操作的逻辑 undo 可能需要的所有低级别锁。可以通过在执行任何逻辑 undo 操作之前首先执行所有的物理 undo 操作来放宽这一要求。在 Weikum 等[1990]中给出这一想法的一个一般化版本，称作多级恢复，它允许多个级别的逻辑操作，在恢复时逐级地进行撤销阶段。

766 灾难恢复的远程备份算法在 King 等[1991]和 Polyzois 和 Garcia-Molina[1994]中给出。

第五部分
Part 5

系统体系结构

数据库系统的体系结构受数据库系统运行所在的底层计算机系统的影响很大。数据库系统可以是集中式的，其中由一台服务器机器执行数据库上的操作。数据库系统也可以设计为并行计算机体系结构。分布式数据库则跨越多个处理位置上分离的机器。

第17章首先概述运行在服务器系统上的数据库系统的体系结构，这种体系结构用于集中式和客户－服务器体系结构上。本章概述各种不同的处理过程，它们共同实现一个数据库的功能。然后，讲述并行计算机体系结构，以及为不同类型的并行计算机设计的并行数据库体系结构。最后，概括建立分布式数据库系统的体系结构问题。

第18章描述数据库的各种操作(尤其是查询处理)是如何实现以便具有并行处理能力的。

第19章介绍分布式数据库中出现的若干问题，并描述如何处理这些问题。这些问题包括怎样存储数据，怎样确保在多个站点执行的事务的原子性，如何执行并发控制，以及在故障发生时如何提供高可用性。此章还描述基于云的数据存储系统、分布式查询处理以及目录系统。

第17章
Database System Concepts, 5E

数据库系统体系结构

数据库系统的体系结构受数据库系统运行所在的底层计算机系统的影响很大，尤其是受计算机体系结构中的联网、并行和分布这些方面的影响：

- 计算机的联网使得某些任务在服务器系统上执行，而另一些任务在客户系统上执行。这种工作任务的划分促使了客户－服务器数据库系统的产生。
- 计算机系统中的并行处理能够加速数据库系统的活动，使其对事务做出更快速的响应，并在每秒内处理更多的事务。查询能够以一种充分利用计算机系统提供的并行能力的方式来处理。对并行查询处理的需求促使了并行数据库系统的产生。
- 在一个组织机构的多个站点间对数据进行分布，可以使得数据能存放在产生它们或最需要它们的地方，而仍能被其他站点或其他部门访问。在不同站点上保存数据库的多个副本还使得大型组织机构甚至可以在一个站点受水灾、火灾或地震等自然灾害影响的情况下仍能继续进行数据库操作。分布式数据库系统用来处理地理上或管理上分布在多个数据库系统中的数据。

我们在本章研究数据库系统的体系结构，从传统的集中式系统开始，然后讨论客户－服务器数据库系统、并行数据库系统以及分布式数据库系统。

17.1 集中式与客户－服务器体系结构

集中式数据库系统是运行在单台计算机系统上，不与其他计算机系统交互的数据库系统。这种数据库系统范围很广，既包括运行在个人计算机上的单用户数据库系统，也包括运行在高端服务器
[769] 系统上的高性能数据库系统。另一方面，客户－服务器系统在功能上划分为一个服务器系统和多个客户端系统。

17.1.1 集中式系统

现代通用的计算机系统包括一到多个处理器，以及若干设备控制器，它们通过公共总线连接在一起，提供对共享内存的访问(如图17-1所示)。处理器具有本地的高速缓冲存储器，用于存放主存储器

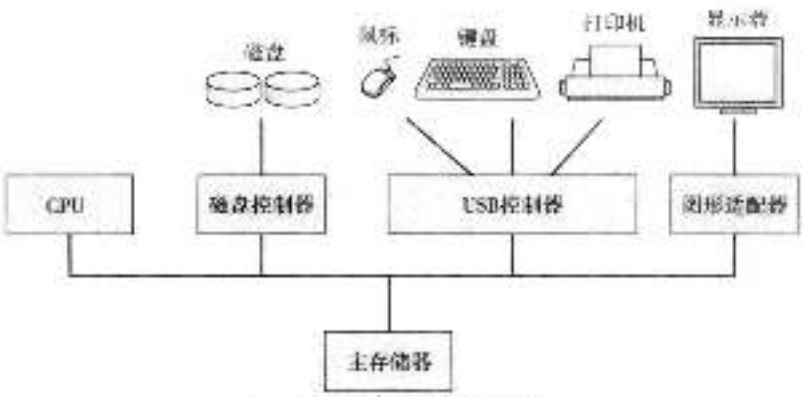


图17-1 集中式计算机系统

中部分数据的本地拷贝，从而加快对数据的访问。每个处理器都可能有几个独立的核（core），每个核可以执行独立的指令流。每个设备控制器负责一种特定类型的设备（例如磁盘驱动器、音频设备或视频播放设备）。处理器和设备控制器可以并发工作，它们竞争对于主存储器的访问。高速缓冲存储器减少了处理器需要访问共享主存储器的次数，从而减少了对主存储器访问的竞争。

我们将使用计算机的方式分为两类：单用户系统和多用户系统。个人计算机和工作站属于第一类。**典型的单用户系统**（single-user system）是个人使用的桌面系统，通常只有一个处理器和一至两张硬盘，而且通常一次只有一个用户使用计算机。另一方面，典型的**多用户系统**（multiuser system）有更多的硬盘和更多的存储器，可能具有多个处理器。它为大量的与系统远程连接的用户服务。

为单用户使用设计的数据库系统一般不提供多用户数据库所提供的许多特性。特别地，它们不支持并发控制，当仅有一个用户能进行更新时并发控制是不需要的。故障恢复支持在这样的系统中是不存在的或者是非常有限的，比如可能只是在更新之前简单地做一个数据库备份。一些这样的系统不支持 SQL，而是提供一种更简单的查询语言，例如 QBE 的变种。比较而言，为多用户系统设计的数据库系统支持我们在前面学习过的全部事务特征。 [770]

虽然当今使用的大多数通用计算机系统有多个处理器，但它们是**粗粒度并行**（coarse-granularity parallelism）的，只具有几个处理器（一般大约 2～4 个），它们共享一个主存。在这样的机器上运行的数据库一般不会将单个查询划分到多个处理器上，而是在每个处理器上运行一个查询，以允许多个查询能并发地运行。因此，这样的系统能提供更高的吞吐量，也就是说，尽管单个事务并没有运行得更快，但在每秒钟内能运行更多的事务。

为单处理器机器设计的数据库已经能够支持多任务，允许多个进程以分时的方式在同一个处理器上运行，使用户感觉多个进程在并行地运行。因此，粗粒度并行机在逻辑上看起来和单处理器机是一样的，为分时计算机设计的数据库系统很容易就能适应在其上运行。

与此相反，**细粒度并行**（fine-granularity parallelism）机拥有大量的处理器。在这样的机器上运行的数据库系统致力于将用户提交的单个任务（例如查询）并行地执行。我们在 17.3 节学习并行数据库系统的体系结构。

并行性正在成为未来设计数据库系统的一个关键问题。虽然现今具有多核处理器的计算机系统只有几个核，但是未来的处理器将有大量的核。[⊖]因此，并行数据库系统，这种曾运行在特殊设计的硬件上的专用系统将成为主流。

17.1.2　客户－服务器系统

由于个人计算机变得速度更快，能力更强，价格更低，因此集中式系统体系结构发生了变化。连接到集中式系统的终端被个人计算机所替代。相应地，以前由集中式系统直接执行的用户界面功能也由个人计算机来处理。其结果是，集中式系统现在起到**服务器系统**（server system）的作用，它满足由客户系统产生的请求。图 17-2 给出了客户－服务器系统的通用结构。



图 17-2　客户－服务器系统的通用结构

数据库系统提供的功能可以大致分为两部分：前端和后端。后端负责存取结构、查询计算和优化、并发控制以及故障恢复。数据库系统的前端包括 SQL 用户界面、表格界面、报表生成工具以及数据挖

⊖ 其原因与计算机体系结构的产热和功耗问题有关。相对于让处理器速度显著加快，计算机架构师正在利用芯片设计方面的先进技术来把更多的核排布于单块芯片上。这一趋势可能还要持续一段时间。

771 凝与分析工具(参见图 17-3)。前端与后端之间的接口通过 SQL 或者应用程序来实现。

我们在第 3 章介绍诸如 ODBC 和 JDBC 之类的标准是为客户端和服务器的接口开发的。使用 ODBC 或 JDBC 接口的任意客户端可以连接到任意提供该接口的服务器。

特定的应用程序，比如电子数据表和统计分析包，直接使用客户－服务器接口从后端服务器访问数据。实际上，它们为特定的任务提供特殊的前端。

正如我们以前在图 1-6(见第 1 章)中所看到的，处理大量用户的系统采用的是三层体系结构，其前端是 Web 浏览器，它与应用服务器交互。应用服务器实际上扮演了数据库服务器的客户端。

一些事务处理系统提供**事务远程过程调用**(transactional remote procedure call)接口来把客户端连接到服务器。这些调用在程序员看来像普通的过程调用，但是来自一个客户端的所有远程过程调用在服务器端封装到一个单独的事务中。所以，如果事务中止，服务器可以撤销单个远程过程调用的影响。

17.2 服务器系统体系结构

服务器系统可以大致分为事务服务器和数据服务器两类。



772 图 17-3 前端功能和后端功能

- **事务服务器**(transaction-server)系统也称作查询服务器(query-server)系统，它提供一个接口，使得客户端可以发出执行一个动作的请求，服务器响应客户端的请求，执行该动作，并将结果送回给客户端。通常，客户端机器将事务传输到服务器系统来执行，然后把结果返回到客户端负责显示数据。可以用 SQL 表达请求，或者使用特定的应用程序接口。
- **数据服务器系统**(data-server system)使得客户端可以与服务器交互，以诸如文件或页面这样的单位对数据进行读取或更新。例如，文件服务器提供文件系统接口，使客户端能够进行文件的创建、更新、读取和删除。数据库系统的数据服务器提供更强的功能，所支持的数据单位比文件要小，可以是页面、元组或对象。数据服务器提供对数据的索引机制，以及事务机制，事务机制保证了即使客户端机器或进程发生故障，数据也不会处于不一致状态。

综上所述，事务－服务器体系结构是目前为止应用最广泛的体系结构。17.2.1 节和 17.2.2 节将对事务服务器和数据服务器体系结构进行详细描述。

17.2.1 事务服务器

现今典型的事务服务器系统包括多个在共享内存中访问数据的进程，如图 17-4 所示。组成数据库系统的进程包括以下几类。

- **服务器进程**(server process)：这些接受用户查询(事务)、执行查询并返回结果的进程。查询可能是从一个用户接口，或者是从一个运行嵌入式 SQL 的用户进程，或者是由 JDBC、ODBC 或类似协议提交给服务器进程的。一些数据库系统为每个用户会话使用一个单独的进程，还有一些数据库系统为所有用户会话使用一个数据库进程，但是使用多线程使多个查询并发执行。

（线程（thread）类似于进程，但是多个线程作为同一个进程的一部分执行，而且一个进程的所有线程在同一个虚拟内存空间中运行。一个进程的多个线程可以并发执行。）许多数据库系统使用一种混合的体系结构，有多个进程，每个进程运行多个线程。

- **锁管理器进程**（lock manager process）：该进程实现锁管理器功能，包括锁授予、锁释放和死锁检测。
- **数据库写进程**（database writer process）：有一个或者多个进程，来将修改过的缓冲块输出到基于连续方式的磁盘中。
- **日志写进程**（log writer process）：该进程将日志记录从日志记录缓冲区输出到稳定存储器上。服务器进程简单地将日志记录添加到在共享内存中的日志记录缓冲区中，如果需要强制输出日志，就会请求日志写进程输出日志记录。
- **检查点进程**（checkpoint process）：该进程定期执行检查点操作。
- **进程监控进程**（process monitor process）：该进程监控其他进程，一旦有进程失败，它将为失败进程执行恢复动作，比如中止失败进程正在执行的所有事务，然后重新启动进程。

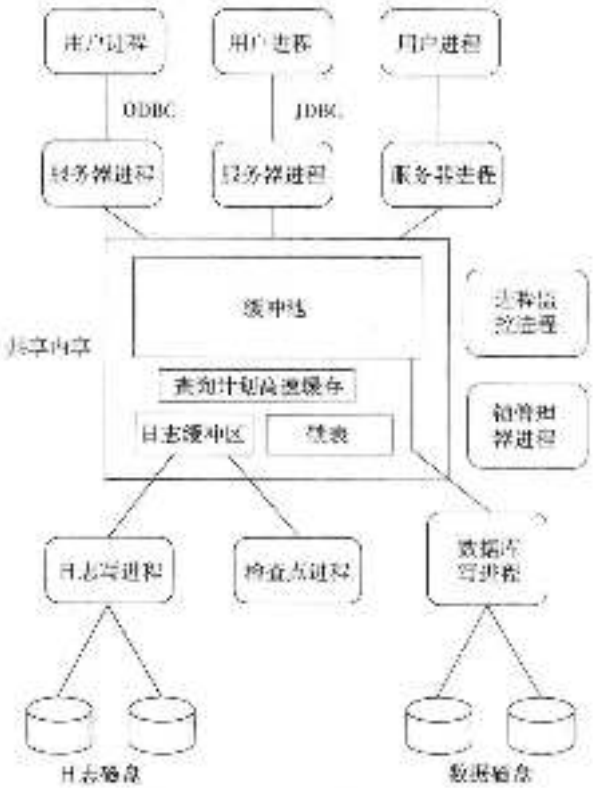


图 17-4　共享内存和进程结构

共享内存包含所有的共享数据，比如：

- 缓冲池。
- 锁表。
- 日志缓冲区，包含将输出到稳定存储器上的日志记录。
- 高速缓存的查询计划，同一个查询再次提交时可以重用。

所有数据库进程都可以访问共享内存中的数据。由于多个进程可能读取或更新共享内存中的数据结构，因此必须有一种机制来确保一次最多只有一个进程来修改一个数据结构，而且当一个数据结构正在被其他进程修改时不能有进程读该数据结构。这种互斥（mutual exclusion）可以借助于称为信号量的操作系统功能来实现。另一种实现方法开销更小，是使用计算机硬件支持的专门的原语（atomic

instruction)，有一种类型的原语测试一个内存位置并自动将其置 1。有关互斥的更多实现细节可以在任意一本标准的操作系统教科书中找到。互斥机制也可用来实现门锁。

为了避免消息传递的开销，在许多数据库系统中，服务器进程通过直接更新锁表(在共享内存中)来实现封锁，而不是向锁管理器进程发送锁请求消息。锁请求过程执行锁管理器进程在得到一个锁请求时将采取的操作。锁请求和释放操作类似于 15.1.4 节中的相应操作，但是有两个明显的区别：

- 由于多个服务器进程可以访问共享内存，因此必须在锁表上确保互斥存取。
- 如果因为锁冲突而不能立刻获得锁，锁请求代码可以监控锁表以检查何时可以授予锁。锁释放代码更新锁表以标记出哪个进程已授予锁。

为了避免重复检查锁表，锁请求代码可以使用操作系统信号量来等待锁授予通知，而锁释放代码必须使用信号量机制来通知等待的事务：它们的锁已经授予了。

即使系统通过共享内存来处理锁请求，它仍然也需要锁管理器进程来检测死锁。

17.2.2 数据服务器

数据服务器系统应用于局域网中，其中客户端与服务器之间具有高速的连接，客户机在处理能力上与服务器机相当，并且要执行的任务是计算密集型的。在这样的环境中，有理由采用如下的做法：把数据传送到客户机，在客户机上进行所有的处理(这样的处理可能要花一些时间)，然后再把数据传回到服务器机。请注意，这种体系结构需要将后端的所有功能都放到客户端。在面向对象数据库系统
[775] 中数据服务器体系结构尤为常见(见第 22 章)。

由于客户端与服务器之间通信的时间代价与本地存储器引用的代价相比要高得多(毫秒与不到 100 纳秒之比)，于是这样的体系结构引出了一些有意思的问题：

- **页面传送**(page shipping)与**项传送**(item shipping)。数据通信的单位可以是粗粒度的，例如页面；也可以是细粒度的，例如元组(或面向对象数据库系统中的对象)。我们使用项(item)这个术语来代表元组和对象。

  如果通信的单位是单个项，那么消息传送的开销与待传输数据的数量相比是高的。相反，当请求一个项时，把不久将可能用到的其他项也传送回去是有意义的。将有些项在请求之前就取过来称作**预读取**(prefetching)。如果多个项驻留在一页上，那么页面传送也可以看作是一种预读取，因为当进程要求访问页面上的一个项时，该页面上的所有项都将传送过去。
- **自适应锁粒度**(adaptive lock granularity)。对于传送给客户机的数据项，通常会由服务器给它们授予锁。页面传送的一个缺点是给予客户机的封锁粒度可能太大——加在页面上的锁意味着锁住了该页上所有的项。即使客户端不访问该页上的某些项，它也隐含地得到了所有预读取项上的锁，其他客户机对这些项加锁的请求可能被阻止，而这是不必要的。因此，引出了锁逐步降级(de-escalation)的技术，即服务器可以要求其客户端将预读取项上的锁传回来。如果客户机不需要某个预读取的项，它就可以把该项上的锁传回给服务器，然后该锁就可以分配给其他客户端。
- **数据高速缓冲存储**(data caching)。只要有足够的存储空间可用，为一个事务而传送到客户端的数据可以高速缓存(cached)到客户端，即使在该事务完成之后，同一个客户端上的后续事务可以使用高速缓存的数据。然而，存在一个**缓存一致性**(cache coherency)问题：即使事务找到了缓存的数据，它还必须确认这些数据是最新的，因为在这些数据被高速缓存后可能有另外的客户端对它们进行过更新。所以，仍然需要与服务器交换一条消息以检查数据的有效性，并得到该数据上的锁。
- **锁高速缓存**(lock caching)。如果各客户端对数据基本上是分割使用的，一个客户端很少请求其他客户端也需要的数据，那么锁也可以高速缓存在客户机中。假设一个客户端在高速缓冲区中找到了一个数据项，同时在其中找到了存取该数据项所需要的锁，那么不需要与服务器通信就可以进行存取过程。然而，服务器必须跟踪高速缓存的锁：如果一个客户端向服务器请求一个锁，服务器必须从那些缓存锁的客户机上收回(call back)该数据项上所有冲突的锁。如果考虑

到机器故障的可能性，这个任务就变得更加复杂。此技术与锁逐步降级技术的不同之处在于，776
缓存是跨事务的，否则，这两项技术就是类似的了。

文献注解提供了关于客户－服务器数据库系统的更多的信息。

17.2.3　基于云的服务器

服务器通常属于提供服务的企业，但是服务供应商至少部分依赖于属于"第三方"的服务器的现象正在成为一种逐渐增长的趋势。该"第三方"既不是客户也不是服务供应商。

使用第三方服务器的一种模式是将整个服务外包到另一家公司，这家公司用自己的软件在其自有的计算机上部署服务。这使得服务提供商可以忽视大多数技术细节并专注于服务的营销。

另一种使用第三方服务器的模式是**云计算**(cloud computing)。服务供应商运行其自己的软件，但软件是运行在另一家公司提供的计算机上。在这种模式下，第三方不提供的任何应用软件，它只提供机器集群。这些机器并不是"真正"的机器，而是通过软件模拟的。利用软件可以让一台真正的计算机模拟多台独立的计算机。这样的模拟机称为**虚拟机**(virtual machine)。服务供应商在这些虚拟机上运行其软件(可能包括数据库系统)。云计算的一个重要优势是服务供应商可以根据需要添加机器，并且当负载减轻时释放机器。无论在资金利用还是能源使用方面，这种模式都被证明是具有高成本效益的。

第三种模式把云计算服务用作数据服务器，这种基于云的数据存储系统将在 19.9 节详细介绍。使用基于云存储的数据库应用程序可能运行在相同的云(即同一组机器)上或者在其他云上。文献注解提供了有关云计算系统的更多信息。

17.3　并行系统

并行系统通过并行地使用多个处理器和磁盘来提高处理速度和 I/O 速度。并行计算机正变得越来越普及，相应地使得并行数据库系统的研究变得更加重要。有些应用需要查询非常大的数据库(TB 数量级，即 10^{12} 字节)，有些应用需要在每秒钟里处理很大数量的事务(每秒数千个事务)。这些应用的需求推动了并行数据库系统的发展。集中式数据库系统和客户－服务器数据库系统的能力不够强大，不足以处理这样的应用。

在并行处理中，许多操作是同时执行的，而不是串行处理的(即按顺序执行各个计算步骤)。**粗粒度**(course-grain)并行机由少量能力强大的处理器组成；而**大规模并行**(massive parallel)机或**细粒度并行**(fine-grain parallel)机使用数千个更小的处理器。当今所有的高端计算机都提供了一定程度的粗粒度并行性，它们至少具有两个或 4 个处理器。大规模并行计算机与粗粒度并行机的区别在于它支持的并行程度要高得多。市场上可以买到具有数百个处理器和磁盘的并行计算机。777

对数据库系统性能的度量有两种主要方式。(1)**吞吐量**(throughput)：在给定时间段内所能完成任务的数量。(2)**响应时间**(response time)：单个任务从提交到完成所需的时间。对于处理大量小事务的系统，通过并行地处理多个事务可以提高它的吞吐量。对于处理大事务的系统，通过并行地执行每个事务中的子任务可以缩短它的响应时间，同时提高它的吞吐量。

17.3.1　加速比和扩展比

并行性研究中的两个重要问题是加速比和扩展比。通过增加并行度在更短的时间内运行一个给定的任务称为**加速比**(speedup)。通过增加并行度来处理更大的任务称为**扩展比**(scaleup)。

考虑一个在具有一定数量的处理器和磁盘的并行系统上运行的数据库应用。现在假设我们通过增加处理器、磁盘以及系统其他部件的数量来扩大系统规模。目标是使处理任务所需的时间与所分配的处理器和磁盘的数量成反比。假设在较大机器上执行一项任务的时间是 T_L，在较小机器上执行相同任务的时间是 T_S。由于并行性而获得的加速比定义为 T_S/T_L。当较大系统拥有的资源(处理器、磁盘等)是较小系统的资源的 N 倍时，如果获得的加速比是 N，那么称该并行系统实现了**线性加速比**(linear speedup)。如果获得的加速比小于 N，则称该系统实现了**亚线性加速比**(sublinear speedup)。图 17-5 描述了线性和亚线性的加速比。

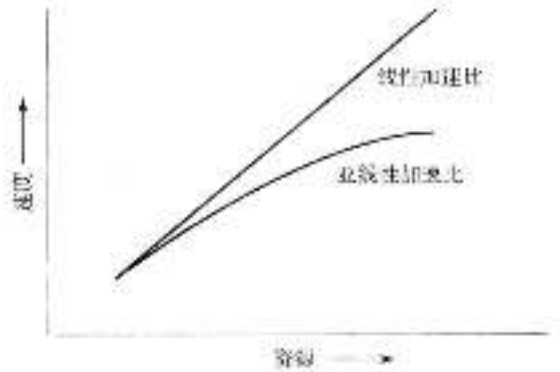


图 17-5 资源增加时的加速比

扩展比指的是通过提供更多资源，在相同时间内处理更大任务的能力。令 Q 是一个任务，Q_N 是一个比 Q 大 N 倍的任务。假设 Q 在给定机器 M_S 上的执行时间是 T_S，任务 Q_N 在比 M_S 大 N 倍的并行机器 M_L 上的执行时间是 T_L。于是扩展比定义为 T_S/T_L。如果 $T_L = T_S$，则称并行系统 M_L 在任务 Q 上实现了**线性扩展比**(linear scaleup)。如果 $T_L > T_S$，则称系统实现了**亚线性扩展比**(sublinear scaleup)。图 17-6 描述了线性和亚线性的扩展比(其中的资源随问题规模成比例增长)。依赖于度量任务规模的方法的不同，在并行数据库系统中有两种类型的相关扩展比：

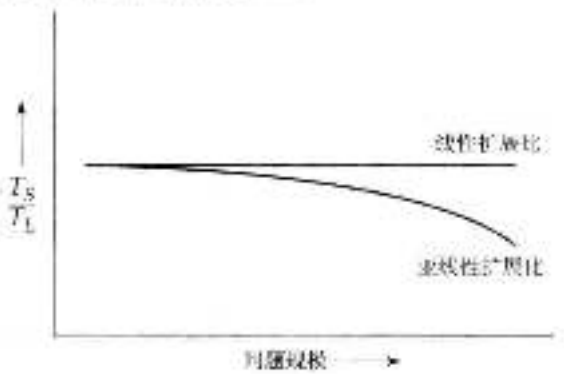


图 17-6 问题规模和资源增加时的扩展比

- **批量型扩展比**(batch scaleup)。在这种扩展比中，数据库规模增大，而任务是那些运行时间依赖于数据库规模的大型工作。这种工作的一个例子是：扫描一个其规模与数据库规模成正比的关系。于是，数据库规模可以作为问题规模的度量。批量型扩展比也应用于科学应用，例如，执行 N 倍精细度的查询，或进行 N 倍长度的模拟。
- **事务型扩展比**(transaction scaleup)。在这种扩展比中，数据库事务提交率增长，并且数据库规模增长与事务提交率增长成正比。这类扩展比适用于事务是小更新类的那种事务处理系统，例如银行账户的存款和取款，而且创建的账户越多，事务的提交率就越高。这种事务处理特别适合于并行执行，因为事务可以在不同的处理器上并发与相互独立地运行，而且即使数据库规模增大了，每个事务也大致耗费同样多的时间。

扩展比通常是度量并行数据库系统效率更重要的标准。数据库系统中引入并行性的目的通常是为了保证即使在数据库规模和事务数量增长时，数据库系统仍能以可接受的速度运行。通过提高并行性来增强系统能力，为企业的增长提供了一条更平稳的路线，这比用速度更快的机器(即使假设有这样的机器存在)来替换集中式系统要好。然而，在使用扩展比进行度量时，我们还必须关心绝对性能数目：具有线性扩展比的机器可能比没达到线性扩展比的机器执行效果差，原因很简单，后者从一开始就要快得多。

778
|
779

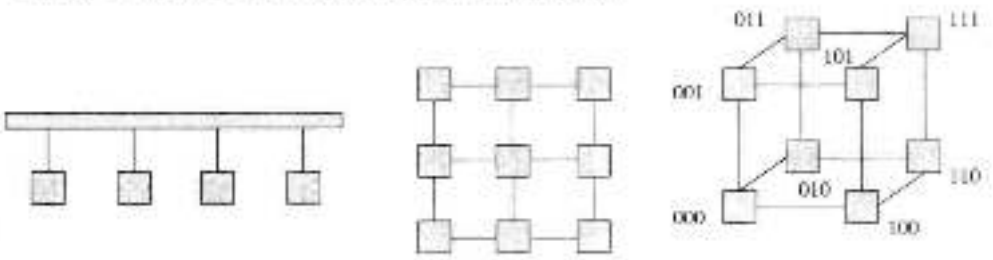
有若干因素影响并行操作的效率，并且可能同时降低加速比与扩展比。

- **启动代价**(start up cost)。存在与单个进程初始化相关的启动代价。在包括数以千计的进程的并行操作中，启动时间可能掩盖实际的处理时间，降低加速比。
- **干扰**(interference)。由于在并行系统中执行的进程经常需要访问共享资源，因此在每个新进程与原有进程竞争共享资源(例如系统总线、共享硬盘，甚至锁)时，就会发生干扰，使速度下降。这种现象会同时影响加速比和扩展比。
- **偏斜**(skew)。我们通过将单个任务分解为多个并行的步骤来减小平均步骤的规模。然而，最慢的那个步骤的服务时间将决定整个任务的服务时间。通常很难将一个任务分解为规模完全相等的多个部分，而面规模分配的方式常常是偏斜的。例如，如果将规模为100的任务分成10个部分，并且分解是偏斜的，则有可能有些任务的规模小于10，而另一些任务的规模大于10；即使有一个任务的规模刚好是20，并行地运行这些任务所得到的加速比也只能是5，而不是我们所期望的10。

17.3.2 互连网络

并行系统包括一套组件(处理器、存储器和磁盘)，这些组件之间通过**互连网络**(interconnection network)相互通信。图17-7显示了三种普遍使用的互连网络类型：

- **总线**(bus)。所有系统组件可以通过一条单独的通信总线来发送和接收数据。这种互连形式如图17-7a所示。总线可以是以太网或并行互连。总线结构非常适合只有少量处理器的情况。然而，它并不能随着并行度增大而很好地进行扩展，因为总线在同一时间只能处理来自一个组件的通信。
- **网格**(mesh)。组件作为网格中的结点，每个组件都和网格中它的所有邻接组件相连接。在二维网格中，每个结点与4个邻接结点相连接；而在三维网格中，每个结点与6个邻接结点相连接。图17-7b显示了一个二维网格。没有直接连接的结点间的相互通信可以通过将消息经由一系列直接互连的中间结点来传送的方式进行。随着组件数目的增加，通信链的数目也随之增大，因此当并行度增大时，网格的通信能力能更好地扩展。 780
- **超立方体**(hypercube)。系统组件按二进制编号，如果两个组件编号的二进制表示正好相差1位，那么它们之间是相互连接的。于是，n个组件中的每一个将与$\log(n)$个其他组件相连接。图17-7c显示了有8个结点的一个超立方体。在超立方体互连中，每个组件发出的消息至多经由$\log(n)$个链接就可以到达任何其他组件。比较而言，在网格架构中，一个组件与另一个组件的距离可能是$2(\sqrt{n}-1)$个链接。(如果网格互连在网格的边缘进行绕接，那么距离可能是$\sqrt{n}$个链接)。因此，超立方体中的通信延迟显著低于网格。



a) 总线　　b) 网格　　c) 超立方体

图17-7 互连网络

17.3.3 并行数据库体系结构

并行机器有若干种体系结构模型。图17-8所示的是其中最重要的几种。(在图17-8中，M表示主存储器，P表示处理器，圆柱体表示磁盘。)

- **共享内存**(shared memory)。所有处理器共享一个公共的主存储器(见图17-8a)。

- **共享硬盘**(shared disk)。所有处理器共享一组公共的磁盘(见图 17-8b)。共享硬盘系统有时又称作集群(cluster)。
- **无共享**(shared nothing)。各处理器既不共享公共的主存储器，又不共享公共的磁盘(见图 17-8c)。
- **层次的**(hierarchical)。这种模型是前三种体系结构的混合(见图 17-8d)。

[781] 17.3.3.1～17.3.3.4 节对这些模型逐一进行详细讨论。

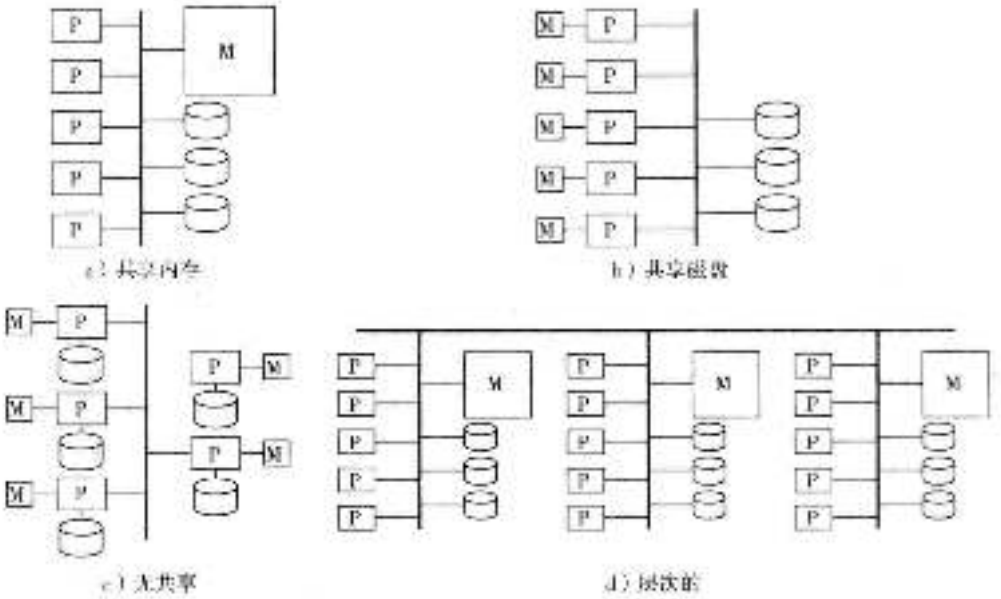


图 17-8 并行数据库体系结构

数据服务器系统上用来加速事务处理的技术，比如 17.2.2 节概述的数据和锁高速缓冲存储以及锁逐步降级，都可以用在共享硬盘的并行数据库和无共享的并行数据库中。事实上，它们对于在这样的系统中高效地进行事务处理是非常重要的。

17.3.3.1 共享内存

在**共享内存**(shared-memory)体系结构中，处理器和磁盘通过总线或互连网络来访问一个公共的主存储器。共享内存的优点在于处理器之间的通信效率极高，存放在共享内存中的数据可以被任何处理器访问，而不需要由软件来移动。一个处理器可以通过往内存中写(这通常只需要不到一微秒的时间)的办法来向其他处理器传送消息，比通过通信机制来传递消息要快得多。共享内存机器的缺点是这种体系结构的规模不能超过 32 个或 64 个处理器，因为总线或互连网络会变成瓶颈(因为它是所有处理器共享的)。到达某一个点之后，增加更多的处理器也没有什么帮助，因为各个处理器会将其大多数时间
[782] 花在等待占用总线以访问主存储器上。

共享内存体系结构通常在每个处理器上有很大的高速缓冲存储器，从而尽量减少对共享内存的访问。然而，至少有一些数据不在高速缓冲存储器中，必须对共享内存进行访问，而且，这些高速缓冲存储器必须保持一致。即，如果一个处理器对某一内存位置执行写操作，则必须从每个缓存该数据的处理器中更新或者是删除这个数据。保持高速缓存一致性的开销将随着处理器数目的增加而上升。因此，共享内存机器的规模扩大不能超过某一个点；当前共享内存机器不能支持多于 64 个处理器。

17.3.3.2 共享硬盘

在**共享硬盘**(shared-disk)模型中，所有处理器都可以通过互联网络直接访问所有磁盘，但是每个处理器有自己私有的内存。与共享内存体系结构相比，共享硬盘体系结构有两个优点：第一，由于每个处理器有自己的主存储器，因此存储器总线就不再是瓶颈了；第二，这种体系结构给出了一种经济的方法来提供一定程度的**容错性**(fault tolerance)：如果一个处理器(或它的主存储器)发生故障，其他处理器可以代替它的工作，这是因为数据库驻留在磁盘上，而磁盘是所有处理器都可以访问的。我们

可以通过使用第 10 章介绍的 RAID 体系结构来使得磁盘子系统自身是容错的。在许多应用中共享硬盘体系结构是可接受的。

共享硬盘系统的主要问题也是可扩展性。虽然存储器总线不再是瓶颈了，但与磁盘子系统互连现在成为了瓶颈；尤其在数据库对磁盘进行大量访问的情况下更是这样。与共享内存系统相比，共享硬盘系统中处理器的数目可以更大一些，但是处理器之间的通信要通过通信网络，所以速度要慢一些（在没有专用通信硬件的情况下会达到几毫秒）。

17.3.3.3　无共享

无共享（shared-nothing）系统中，机器的每个结点包括一个处理器、一个内存，以及一张或多张磁盘。一个结点上的处理器可以通过高速互连网络与另一个结点上的另一个处理器通信。一个结点作为该结点所拥有的磁盘上的数据的服务器来运作。由于本地磁盘访问由各个处理器的本地磁盘提供，因此无共享模型克服了所有的 I/O 都需要通过同一个互连网络的缺点，只有访问非本地磁盘的查询及其结果关系才需要通过网络传送。而且，无共享系统中的互连网络通常设计成可扩展的，使得当更多结点加入时，其传输能力也随之增强。因此，无共享体系结构支具可扩展性，而且可以很容易地支持大量的处理器。无共享系统的主要缺点是通信的代价和非本地磁盘访问的代价，这些代价比共享内存或共享硬盘体系结构中的代价要高，因为数据传送涉及两端的软件交互。

783

17.3.3.4　层次的

层次的体系结构（hierarchical architecture）综合了共享内存、共享硬盘和无共享体系结构的特点。在最上层，系统由通过互连网络连接起来的若干个结点组成，这些结点之间互不共享硬盘或主存储器，因此最上层是一种无共享的体系结构。系统的每个结点实际上可以是具有少量处理器的共享内存系统。或者，每个结点可以是一个共享硬盘的系统，而且共享一组硬盘的系统中的每一个又可以是一个共享内存的系统。因此，一个系统可以构造成层次的，其底层是具有少量处理器的共享内存的体系结构，其顶层是无共享的体系结构，也许中间层是共享硬盘的体系结构。图 17-8d 描述了一个层次的体系结构，它具有若干个共享内存的结点，通过无共享的体系结构连接在一起。当今的商用并行数据库系统运行在几种这样的体系结构之上。

为了降低这样的系统的程序设计复杂性，产生了**分布式虚拟存储器**（distributed virtual-memory）体系结构，这样的体系结构中逻辑上有一个共享的主存储器，但物理上有多个互不相交的主存储器系统；虚拟存储器映射硬件与系统软件的配合使得每个处理器可以将这些互不相交的主存储器看成是一个单一的虚拟主存储器。由于访问速度依赖于页面是否在本地而不同，因此这样的体系结构也称为**非一致性内存体系结构**（NonUniform Memory Architecture，NUMA）。

17.4　分布式系统

在**分布式数据库系统**（distributed database system）中，数据库存储在几台计算机中。分布式系统中的计算机之间通过诸如高速私有网络或因特网那样的通信媒介相互通信。它们不共享主存储器或磁盘。分布式系统中的计算机在规模和功能上是可变的，小到工作站，大到大型机系统。

对于分布式系统中的计算机有多种不同的称呼，例如**站点**（site）或**结点**（node），取决于提及它们时的上下文。我们主要采用站点（site）这个术语，以强调这些系统的物理分布。分布式系统的通用结构如图 17-9 所示。

无共享并行数据库与分布式数据库之间的主要区别在于，分布式数据库一般是地理上分离的，分别管理的，并且互连速度更低。另一个主要区别是：在分布式数据库系统中，我们将事务区分为局部事务和全局事务。**局部事务**（local transaction）是仅访问在发起事务的那个站点上的数据的事务。另一方面，**全局事务**（global transaction）或者访问发起事务的站点之外的某个站点上的数据，或者访问几个不同站点上的数据。

784

建立分布式数据库系统有几个原因，包括数据共享、自治性和可用性。

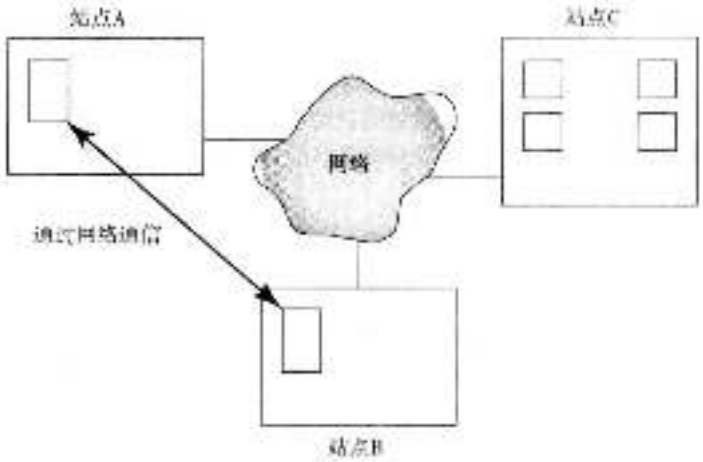


图 17-9 分布式系统

- **数据共享**(sharing data)。建立分布式数据库系统的主要优点是，它提供了一个环境使一个站点上的用户可以访问存放在其他站点上的数据。例如，在一个分布式大学系统中，每个校区存储与该校区相关的数据。一个校区的用户可以访问另一个校区的数据。如果没有这种能力，那么要把学生记录从一个校区传送到另一个校区，就需要借助于与现有系统相互关联的外部机制。
- **自治性**(autonomy)。通过数据分布的方式来共享数据的主要优点在于，每个站点可以对本地存储的数据保持一定程度的控制。在集中式系统中，中心站点的数据库管理员对数据库进行控制。在分布式系统中，有一个全局数据库管理员负责整个系统。有一部分职责被委派给每个站点的本地数据库管理员。每个管理员可以有不同程度的**局部自治**(local autonomy)，其程度的不同依赖于分布式数据库系统的设计。可以进行本地自治通常是分布式数据库的一个主要优势。
- **可用性**(availability)。在分布式系统中，如果一个站点发生故障，其他站点可能还能继续运行。特别地，如果数据项在几个站点上进行了**复制**(replicate)，需要某个特定数据项的事务可以在这几个站点中的任何一个上找到该数据项。于是，一个站点的故障不一定意味着整个系统停止运转。

785

系统必须能检测到一个站点发生了故障，并需要采取适当的动作来从故障中恢复。系统不能再使用故障站点的服务。最后，当故障站点恢复了或修复好了，还需要有一定的机制来将它平滑地重新集成到系统中。

虽然分布式系统的故障恢复比集中式系统更复杂，但分布式系统中即使一个站点发生了故障，系统的绝大部分还能继续运行，这一能力使系统的可用性大大增强。对用于实时应用的数据库系统来说，可用性是至关重要的。例如，如果不能对航班数据进行访问，将导致潜在的机票购买者流失到竞争对手那里去了。

17.4.1 分布式数据库示例

考虑一个银行系统，它有 4 家支行，位于 4 个不同的城市。每家支行有自己的计算机，有维护该分行所有账户的数据库。每个这样的配置称作一个站点。另外还有一个站点维护关于该银行的所有支行信息。

为说明站点上两类事务的差异(局部的和全局的)，考虑如下事务：给位于 Valleyview 支行的账户 A-177 中增加 50 美元。如果在 Valleyview 支行发起该事务，那么它是一个局部事务；否则，它就是一个全局事务。将 50 美元从账户 A-177 转到 Hillside 支行中的账户 A-305 的事务是一个全局事务，因为事务的执行要访问两个不同站点上的账户。

在一个理想化的分布式数据库系统中，站点将共享一个公共的全局模式（尽管有些关系可能只存放在其中某些站点上），所有站点运行相同的分布式数据库管理软件，而且各个站点互相知道对方的存在。如果从头开始创建一个分布式数据库，它确实有可能达到上述目标。然而，现实中分布式数据库需要通过连接多个已存在的数据库系统来构造，每个数据库都有自己的模式而且可能运行不同的数据库管理软件。这样的系统有时称作**多数据库系统**（multidatabase system）或**异构分布式数据库系统**（heterogeneous distributed database system）。19.8 节将讨论这些系统，给出在作为组成成分的各个系统异构的情况下，如何达到一定程度的全局控制。

17.4.2　实现问题

在构建分布式数据库系统时，事务的原子性是个重要问题。如果一个事务在两个站点上运行，除非系统设计者非常小心，否则它可能在一个站点上提交而在另一个站点上中止，导致不一致状态。一些事务提交协议确保这种情况不会出现。**两阶段提交**（Two-Phase Commit，2PC）协议是这些协议中应用最广泛的协议。 786

2PC 的基本思想是每个站点将事务执行到部分提交状态，然后将提交决定权交给一个单独的协调站点。事务这一时刻在该站点上称为处于准备状态。仅当该事务在每个执行它的站点上均达到准备状态时，协调站点才决定提交该事务。否则（例如，如果事务在任一个站点上中止），协调站点会决定中止该事务。每个执行该事务的站点必须遵循协调站点的决定。如果当事务处于准备状态时一个站点发生故障，当该站点从故障状态恢复时它必须处于一个状态——提交或中止该事务，这取决于协调站点的决定。2PC 协议将在 19.4.1 节详细描述。

并发控制是分布式数据库的另一个问题。由于一个事务可以访问分布在几个站点上的数据项，因此这几个站点上的事务管理器可能需要协调以实现并发控制。如果用到锁（现实中经常会遇到的），则可以在包含被访问数据项的站点上局部地执行封锁，然而可能会发生涉及由多个站点发起的事务的死锁。所以死锁检测必须跨越多个站点。分布式系统中更容易发生故障，因为不仅仅是计算机可能出现故障，而且通信链路也可能发生故障。当故障发生时，数据项的复制是分布式数据库继续运转的关键。对复制数据的并发控制更加复杂。19.5 节提供有关分布式数据库中并发控制的详细介绍。

基于单个程序单元执行多个动作的标准事务模型，常常不适合执行跨越多个数据库边界的任务，这些数据库不能或不打算合作实现诸如 2PC 那样的协议。有另一种方法通常用于这类任务，其他基于用于通信的**持久消息**（*persistent messaging*）（用于通信）的方法一般用于这类任务中。持久消息在 19.4.3 节进行讨论。

当要执行的任务很复杂，涉及多个数据库和/或与人有多次交互时，任务的协调以及为任务确保事务的属性就变得更加复杂。**工作流管理系统**（*workflow management system*）就是为执行这类任务而设计的。26.2 节讲述了工作流管理系统。

当某个组织机构为了实现一个应用而必须在分布式体系结构和集中式体系结构之间做出选择时，系统架构师必须平衡将数据进行分布的优点和缺点。我们已经看到使用分布式数据库的优点。分布式数据库系统的主要缺点是为保证各站点间的正确协作而增加的复杂性。这种增加的复杂性表现为几种形式：

- **软件开发代价**（software-development cost）。实现一个分布式数据库系统要更加困难，因此，代价更高。
- **更大的出错可能性**（greater potential for bug）。由于构成分布式系统的各个站点并行地运行，因此更难以保证算法的正确性，尤其是当系统的一部分发生故障时的运行，以及从故障中的恢复。有可能存在非常微妙的错误。 787
- **增加的处理开销**（increased processing overhead）。消息的交换以及为了达到站点间的协作而需要的附加计算，这些是集中式系统中所没有的开销。

分布式数据库设计的方法有好几种，可以是完全分布式设计，也可以是有很大限度集中的设计。我们将在第 19 章学习这些方法。

17.5 网络类型

分布式数据库和客户-服务器系统的建立都以通信网络为基础。有两种基本的网络类型：局域网(local-area network)和广域网(wide-area network)，两者的主要区别在于它们的地理分布方式不同。局域网由在小的地理范围(例如单个建筑物或几个相邻的建筑物)内分布的若干处理器构成。相反，广域网由大的地理范围(例如全美国或全世界)内分布的许多自治的处理器构成。这些差别意味着通信网络的速度和可靠性方面的显著差异，并且也反映到分布式操作系统的设计中。

17.5.1 局域网

局域网(Local-Area network，LAN)(见图 17-10)作为计算机之间互相通信和共享数据的一种方式，出现于 20 世纪 70 年代初期。人们意识到，对于许多企业来说，使用大量的小计算机，每台计算机上有它自己的独立应用，比使用单个大系统要更经济。由于每台小计算机都可能需要访问全套的外围设备(例如磁盘和打印机)，又由于在一个企业中可能需要某种形式的数据共享，因此很自然地要把这些小系统连接成一个网络。

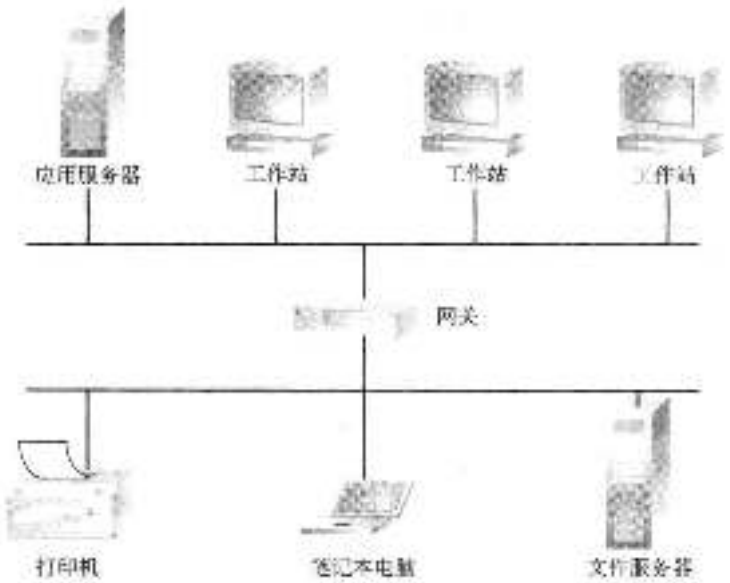


图 17-10 局域网

LAN 一般用于办公室环境中。在这种系统中所有站点之间距离都很近，因此其通信链接的速度比广域网高，而且错误率比广域网低。局域网中最常用的链接是双绞线、同轴电缆、光纤以及无线连接。通信速度从每秒几十 GB(例如无线局域网)，到每秒 1GB 的 10 亿比特以太网。最新以太网的标准是 10GB。

存储区域网(Storage-Area Network，SAN)是为连接大型存储设备(磁盘)与使用数据的计算机(见图 17-11)而设计的一种特殊类型的高速局域网。

因而存储区域网有助于构建大规模共享磁盘系统(*shared-disk system*)。使用存储区域网把多个计算机连接到大型存储设备的动机和使用共享硬盘的数据库本质上是一样的，即：

788
789

- 可通过增加更多的计算机来扩展规模。
- 高可用性，因为即使一台机器发生故障，数据仍然可访问。

在存储设备中使用的 RAID 组织是为了确保数据的高可用性，即使单张磁盘发生故障，数据处理也可以继续。存储区域网构建时常常会有冗余，例如站点间有多条通路，因此如果网络的一部分(比如链路或者连接)发生故障，网络功能仍能继续。

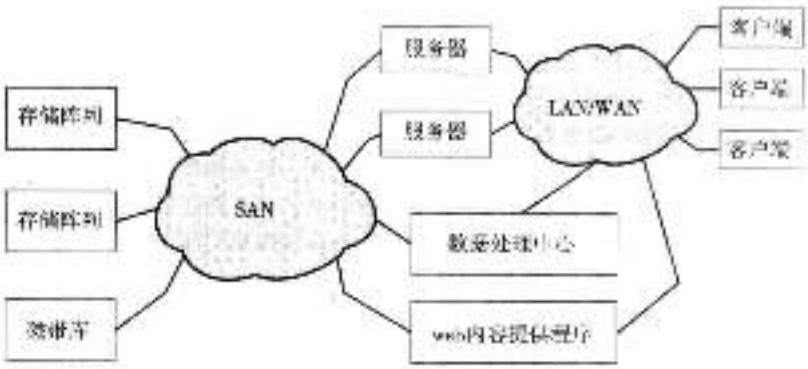


图 17-11 存储区域网

17.5.2 广域网

广域网(Wide-Area Network，WAN)出现在20世纪60年代后期，主要作为一个学术研究项目，提供站点间的高效通信，使得大范围的用户能够方便并且经济地共享硬件和软件。在20世纪60年代初期开发了能够通过电话线路将远程终端连接到中央计算机的系统，但是这不是真正的WAN。阿帕网(Arpanet)是设计和开发的第一个广域网，阿帕网的工作始于1968年，现在阿帕网已经从一个包括4个站点的试验性网络成长为一个世界范围的网络——**因特网**(Internet)，它包括数亿的计算机系统。因特网上典型的连接是光纤线路，有时是卫星信道。典型的广域链路的数据传输率的范围从每秒几MB到每秒几百GB。链路的末端(终端用户站点)传统上采用最慢的链路，通常基于数字用户环线(Digital Subscriber Line，DSL)技术(支持每秒几MB)，或者是基于固定电话线路的拨号调制解调器连接(最高支持每秒Kb)。现在典型的链路末端是电缆网调制解调器或光纤连接(它们各自支持每秒几十MB)，或者支持每秒几MB的无线连接。

除了数据传输率的限制外，在WAN中的通信还必须克服巨大的延迟(latency)：在世界范围内传输一条消息可能要用几百毫秒的时间，这既是由于光传输的延迟，也是由于消息传播路径上各个路由的队列等待延迟。设计数据和计算资源在地理上分布的应用时必须非常小心，以避免这些延迟过度地影响系统性能。

WAN可以划分为两种类型：

- **非持续连接**(discontinuous connection)的WAN，例如基于移动无线连接的WAN，主机只是在部分的时间里与网络相连接。
- **持续连接**(continuous connection)的WAN，例如有线因特网，主机在所有时间内都与网络相连接。

非持续连接的网络通常不允许跨站点的事务，但是可以提供远程数据的本地副本，并且周期性地(例如每天夜间)刷新这些副本。对于那些一致性要求不是特别强的应用，例如支持共享系统、Lotus Notes之类的组件系统，允许在本地对远程数据进行更新，然后再周期性地将更新传播回远程站点。有可能在不同站点上发生更新冲突，需要检测到这种冲突并加以解决。后面在25.5.4节中会描述一种检测更新冲突的机制，然而解决更新冲突的机制依赖于不同的应用。 290

17.6 总结

- 集中式数据库系统完全运行在单台计算机上。随着个人计算机和局域网的发展，数据库前端功能不断地移向客户机，而后端功能由服务器系统提供。客户－服务器接口协议推动了客户－服务器数据库系统的发展。
- 服务器可以是事务服务器，也可以是数据服务器。尽管在提供数据库服务方面，事务服务器的使用大大超过数据服务器的使用。
  - 事务服务器有多个进程，可能运行在多个处理器上，所以这些进程要访问公共数据，比如数

据库缓冲区，系统将这些数据存放在共享内存中。除了处理查询的进程，还有执行诸如锁和日志管理以及检查点等任务的系统进程。

- 数据服务器系统提供给用户的是未加工的数据。这样的系统通过把数据和锁高速缓存在客户端，来努力使客户端和服务器之间的通信最小化。并行数据库系统使用类似的优化。

- 并行数据库系统由通过高速互连网络连接在一起的多台处理器和多张硬盘构成。加速比衡量通过增加并行性可以得到的对单个事务的处理速度的增长。扩展比衡量通过增加并行性可以得到的处理大量事务的能力。干扰、偏斜和启动代价是得到理想的加速比和扩展比的障碍。
- 并行数据库体系结构包括共享内存、共享硬盘、无共享以及层次的体系结构。这些体系结构在可扩展性以及通信速度方面各有千秋。
- 分布式数据库系统是部分独立的一组数据库系统，它们共享一个公共模式（理想情况下），并且协调地处理访问非本地数据的事务。系统之间通过通信网络来相互通信。
- 局域网连接分布在小的地理范围内的结点，比如连接单个建筑或几个相邻建筑。广域网连接分布在大的地理范围内的结点。现在 Internet 是使用最广泛的广域网。
- 存储区域网是一种特殊形式的局域网，是为大型存储设备和多台计算机之间提供快速互连而设计的。

术语回顾

- 集中式系统
- 服务器系统
- 粗粒度并行
- 细粒度并行
- 数据库进程结构
- 互斥
- 线程
- 服务器进程
  - 锁管理进程
  - 数据库写进程
  - 日志写进程
  - 检查点进程
  - 进程监视进程
- 客户－服务器系统
- 查询服务器
- 数据服务器
  - 预读取
  - 逐步降级
  - 数据高速缓冲存储
  - 缓存一致性
  - 锁高速缓冲存储
  - 收回
- 并行系统
- 吞吐量
- 响应时间
- 加速比
  - 线性加速比
  - 亚线性加速比
- 扩展比
  - 线性扩展比
  - 亚线性扩展比
  - 批量型扩展比
  - 事务型扩展比
- 启动代价
- 干扰
- 偏斜
- 互连网络
  - 总线
  - 网格
  - 超立方体
- 并行数据库体系结构
  - 共享内存
  - 共享硬盘（集群）
  - 无共享
  - 层次的
- 容错性
- 分布式虚拟存储器
- 非一致性内存体系结构（NUMA）
- 分布式系统
- 分布式数据库
  - 站点（结点）
  - 局部事务
  - 全局事务
  - 局部自治性
- 多数据库系统
- 网络类型
  - 局域网（LAN）
  - 广域网（WAN）
  - 存储局域网（SAN）

实践习题

17.1 将共享结构存储在一个专用进程的本地内存中，而不是存储在共享内存中，通过与该进程间的通信存取共享数据。这种体系结构的缺陷是什么？

17.2 在典型的客户－服务器系统中，服务器机器比客户机的能力要强得多。也就是说，其处理器速

度更快，可能有多个处理器，有更大的内存和磁盘容量。请考虑另外一种设想，其中客户机和服务器机器能力相同。构建这样一个客户 - 服务器系统有意义吗？为什么？哪一种设想更适合于数据服务器体系结构？

17.3　考虑一个基于客户 - 服务器体系结构的数据库系统，其服务器是一个数据服务器。

a. 客户和服务器间的互连速度对于选择是进行元组传送还是进行页面传送有什么影响？

b. 如果采用页面传送，数据在客户端的高速缓存可以组织成元组缓存或者页面缓存。页面缓存以页面为单位存储数据，而元组缓存以元组为单位存储数据。假设元组比页面小，请描述元组缓存比页面缓存优越的一个地方。

17.4　假设一个事务是将 SQL 嵌入到 C 中书写的，大约 80% 的时间花在运行 SQL 代码上，剩余 20% 的时间花在运行 C 代码上。如果仅仅对 SQL 代码实施并行，那么可以期望得到多大的加速比？说明理由。

17.5　一些数据库操作，比如连接操作，在数据（例如，连接涉及的其中一个关系表中的数据）是存放在内存中或者不是存放在内存中这两种情况下，会出现速度上的明显差异。请说明这个事实如何解释了**超线性加速比**（superlinear speedup）现象，这里一个应用程序的加速比高于分配给它的资源数量的增长。

17.6　并行系统通常有这样的网络结构：多个包含 n 个处理器的集合连接到单个以太网交换机上，而这些以太网交换机本身又连接到另一台以太网交换机上。这个架构是否相当于总线、网格或超立方体架构？如果不，你会如何描述这种互连架构？

习题

17.7　如果不需要对单个查询进行并行化，为什么将数据库系统从单处理器机器移植到多处理器机器相对比较容易？

793

17.8　对于处理短事务的客户 - 服务器关系数据库，常采用事务服务器体系结构，而对于处理较长事务的面向对象数据库系统，常采用数据服务器体系结构。请给出两条理由，解释为什么数据服务器适合于面向对象数据库而不适合于关系数据库。

17.9　什么是锁逐步降级？什么情况下需要锁逐步降级？为什么当数据传送的单位是数据项时不需要锁逐步降级？

17.10　假设你负责一个公司的数据库的运行，主要任务是处理事务。假设该公司每年都在迅速增长，大到使得公司当前的计算机系统已不再适用。当你选择一台新的并行计算机时，你最关注加速比、批量型扩展比，还是事务型扩展比？为什么？

17.11　典型的数据库系统由一组共享一个共享内存区域的进程（或线程）实现。

a. 如何控制对于共享内存区域的存取？

b. 两阶段锁协议适用于并行化访问共享内存中的数据结构吗？请解释你的答案。

17.12　允许用户并发访问数据库系统的共享内存区域是否明智？请解释你的答案。

17.13　在事务处理系统中，对于线性加速比起负面影响的有哪些因素？对于共享内存、共享硬盘、无共享这几种体系结构中的每一种，分别说明哪个因素可能是最重要？

17.14　内存系统可以分为多个模块，在一个特定时刻，每个模块可以处理一个独立的请求。这种存储架构会对共享内存系统所支持的处理器数量产生什么影响？

17.15　考虑一个银行，它有若干站点，每个站点运行一个数据库系统。假设这些数据库之间唯一的交互方式是利用持久消息进行电子转账，这样的系统称得上是分布式数据库吗？为什么？

文献注解

Hennessy 等[2006]对计算机架构领域提供了极好的介绍，Abadi[2009]提供了对云计算和在该环境中运行数据库事务所面临的挑战的非常好的介绍。

Gray 和 Reuter[1993]是一本描述事务处理的教科书，包括对客户 - 服务器体系结构和分布式系统

的描述。第5章的文献注释提供了有关ODBC、JDBC以及其他数据库访问API的更多信息的参考文献。

DeWitt和Gray[1992]对并行数据库系统进行了综述，包括其体系结构和性能度量。Duncan[1990]对并行计算机体系结构进行了综述。Dubois和Thakkar[1992]是一本关于可伸缩共享内存体系结构的文集。运行着Rdb的DEC集群是共享磁盘数据库体系结构的早期商业用户之一。Rdb现在属于Oracle，命名为Oracle Rdb。Teradata数据库机器是最早使用无共享数据库体系结构的商用系统之一。Grace和Gamma研究原型也使用了无共享体系结构。

Ozsu和Valduriez[1999]是介绍分布式数据库系统的教科书。关于并行和分布式数据库系统的进一步的参考文献在第18章和第19章的文献注解中分别列出。

794
796

Comer[2009]、Halsall[2006]和Tanenbaum[2002]、Kurose和Ross[2005]、Peterson和Davie[2007]对计算机网络进行了一般性概述。

并行数据库

本章讨论基于关系数据模型的并行数据库系统所使用的基本算法。特别地，我们将重点放在多张磁盘上的数据放置，以及关系运算的并行实现的评估上。它们两个对于一个并行数据库的成功是有帮助的。

18.1 引言

二十多年前，并行数据库系统几乎一度被全部否定，甚至一些最坚定的拥护者都不再支持它。但是今天，每一个数据库系统供应商都成功地打开了并行数据库产品的市场。以下几种趋势促成了这一转变：

- 随着计算机的大量使用，企业组织的事务需求增长了。而且，万维网的发展创造了许多拥有数以百万计浏览者的站点。从这些浏览者收集来的越来越多的数据为许多公司产生了特别庞大的数据库。
- 企业组织正使用这些不断增长的大量数据（例如有关用户所购买的商品，用户所点击的 Web 链接，以及用户打电话的时间这些方面的数据），来计划市场行为和定价。用于这种目的的查询称作**决策支持查询**（decision-support query），而这样的查询所需要的数据量可能高达 TB 级。单处理器系统无法按所需速度处理如此大量的数据。
- 数据库查询的面向集合特性很自然地将自身引向并行化。许多商品化和研究性系统已经证明了并行查询处理的能力和可扩展性。
- 随着微处理器价格的下降，并行机器已变得很普遍，而且价格并不昂贵。
- 各独立处理器也可以利用多核架构来将其转化为并行机器。

正如第 17 章所讨论的，并行性用来提供加速比，即由于提供了更多的诸如处理器和磁盘那样的资源，查询可以执行得更快。并行性还用来提供扩展比，即通过增大并行度来处理更大的工作负载，而无须延长响应时间。

我们在第 17 章列举了并行数据库系统的几种不同体系结构：共享内存、共享磁盘、无共享和层次的体系结构。简单地说，在共享内存体系结构中，所有处理器共享一个公共的主存储器和磁盘；在共享磁盘体系结构中，各处理器有自己独立的主存储器，但共享公共的磁盘；在无共享的体系结构中，处理器间既不共享主存储器，也不共享磁盘；而层次的体系结构含有多个结点，这些结点间既不共享主存储器，也不共享磁盘，但是每个结点在内部是共享内存或共享磁盘的体系结构。

18.2 I/O 并行

I/O 并行（I/O parallelism）最简单的形式是指通过将关系划分到多张磁盘上来缩减从磁盘上对关系进行检索所需的时间。并行数据库环境中数据划分最通用的形式是水平分区。在**水平分区**（horizontal partitioning）中，关系中的元组划分（或分散）到多张磁盘上，使得每个元组驻留在一张磁盘上。人们已经提出了好几种划分策略。

18.2.1 划分技术

我们介绍一种基本的数据划分策略。假定要把数据划分到 n 张磁盘 D_0，D_1，…，D_{n-1} 上。

- **轮转法**（round-robin）。该策略对关系进行任意顺序的扫描；将第 i 个元组送到标号为 $D_{i \bmod n}$ 的磁

盘上。轮转模式保证了元组在多张磁盘上的平均分布，即每张磁盘上具有大致相同数目的元组。

- **散列划分**(hash partitioning)。这种分散策略将给定关系模式中的一个或多个属性指定为划分属性。选定一个值域为 $0, 1, \cdots, n-1$ 的散列函数。原始关系的每个元组基于划分属性进行散列。如果散列函数返回 i，则把该元组放到磁盘 D_i 上。[⊖]
- **范围划分**(range partitioning)。这种策略将每张磁盘分配具有连续属性值范围的元组。它选定一个划分属性 A 和一个**划分向量**(partitioning vector) $[v_0, v_1, \cdots, v_{n-2}]$；若 $i<j$，则 $v_i<v_j$。对关系
[798] 进行如下划分：考虑元组 t，它满足 $t[A]=x$。如果 $x<v_0$，则将 t 放置在磁盘 D_0 中；如果 $x \geqslant v_{n-2}$，则将 t 放到磁盘 D_{n-1} 中；如果 $v_i \leqslant x<v_{i+1}$，则将 t 放到磁盘 D_{i+1} 中。

例如，在标号为 0、1、2 的三张磁盘上进行范围划分，可能将属性值小于 5 的元组分配到磁盘 0，属性值在 5～40 之间的元组分配到磁盘 1，将属性值大于 40 的元组分配到磁盘 2。

18.2.2 划分技术比较

一旦将一个关系划分到多张磁盘上，我们就可以使用所有磁盘来对它进行并行检索。同样，当对一个关系进行了划分，就可以并行地将它写到多张磁盘上。因此，有 I/O 并行与没有 I/O 并行相比，读或写整个关系的传输速度要快得多。然而，读取整个关系，或扫描一个关系，仅是访问数据的一种方式。数据访问可以分类如下：

1. 扫描整个关系。
2. 按相关性定位元组，(例如，*employee_name* = "Campbell")。这种查询称作**点查询**(point query)，它寻找在特定属性上取特定值的元组。
3. 定位出给定属性取值处于指定范围内的所有元组，(例如，10 000 < *salary* < 20 000)。这种查询称作**范围查询**(range query)。

不同划分技术支持这些访问类型的效率是不同的：

- **轮转法**(round-robin)。这种模式非常适合于希望对每个查询顺序地读整个关系的应用。若采用这种模式，点查询和范围查询的处理都很复杂，因为 n 张磁盘全都要用于查找。
- **散列划分**(hash partitioning)。这种模式最适合于基于划分属性的点查询。例如，如果一个关系基于 *telephone_number* 属性进行划分，那么我们可以通过将划分用的散列函数作用到 555－3333 上，然后查找该磁盘来回答"找出 *telephone_number* = 555－3333 的职工记录"的查询。将查询对应到单张磁盘节省了在多张磁盘上初始化查询的启动代价，并且使得其他磁盘空闲下来以处理其他查询。

散列划分对于顺序扫描整个关系也有用。如果散列函数是一个很好的随机化函数，并且划分属性构成关系的码，那么每张磁盘上的元组数目就会大致相同，不会有太大差异。因此，扫描关系所需的时间大致就是在单个磁盘系统中扫描关系所需时间的 $1/n$。

然而，这种模式不能很好地适用于非划分属性上的点查询。基于散列的划分也不能很好地回答范
[799] 围查询，因为一般说来，散列函数不保持一个范围内的互相临近。因此，回答范围查询需要扫描所有磁盘。

- **范围划分**(range partitioning)。这种模式很适合于在划分属性上的点查询和范围查询。对于点查询，我们可以参考划分向量来定位元组所在的磁盘。对于范围查询，我们参考划分向量来找出元组可能驻留的磁盘范围。在这两种情况下，查询范围都缩小到那些正好可能含有任何感兴趣元组的磁盘上。

这种特性的优点是，如果查询范围内只有少量元组，那么一般说来查询只会发送给一张磁盘，而不是发送给所有的磁盘。因为可以将其他磁盘用于回答其他查询，所以范围划分在具有较好的响应时间的同时还获得了更高的吞吐量。另一方面，如果查询范围内有许多元组(当查询范围占关系域的较大

⊖ 散列函数的设计在 11.6.1 节讨论。

比例时会是这样)。那么就会有许多元组必须从少量磁盘中检索出来，导致这些磁盘上的 I/O 瓶颈(热点)。在这种**执行偏斜**(execution skew)的例子中，所有的处理发生在一个或少数几个分区中。相反，对于这样的查询，散列分布和轮转法划分会用到所有的磁盘，在具有几乎同样吞吐量的同时极大地减少了查询的响应时间。

划分类型还影响到其他关系运算，例如连接，我们将在 18.5 节讨论。因此，划分技术的选择也依赖于需要执行的运算。一般而言，更倾向于使用散列划分和范围划分，而不是轮转法划分。

在拥有许多磁盘的系统中，可以用如下方法来确定要把一个关系划分到多少张磁盘上：如果关系仅包含少量元组，在一张磁盘块中就能放下，那么最好将该关系放到单张磁盘中。大型关系更倾向于划分到所有可用的磁盘上。如果一个关系包括 m 张磁盘块，且系统中有 n 张磁盘可用，那么应该给这个关系分配 $\min(m, n)$ 张磁盘。

18.2.3 偏斜处理

当划分一个关系时(采用不是轮转法的其他技术)，元组的分布有可能发生**偏斜**(skew)，即某些分区中放置了很高百分比的元组，而其他分区中只有很少的元组。偏斜可能的表现形式分类如下：

- 属性值偏斜。
- 划分偏斜。

属性值偏斜(attribute value skew)指的是某些值出现在许多元组的划分属性中。在划分属性上取相同值的所有元组落入同一分区中，从而导致了偏斜。**划分偏斜**(partition skew)指的是这样的事实：即使 800
不存在属性值偏斜，划分也可能会出现负载不均衡。

不管采用范围划分还是采用散列划分，属性值偏斜都会导致划分偏斜。如果没有仔细选择好划分向量，那么范围划分会导致划分偏斜。如果散列函数选得好，那么散列划分导致划分偏斜的可能性就小。

在 17.3.1 节中已经谈到，即使很小的偏斜也可能导致性能的显著降低。随着并行度的提高，偏斜变成了更加严重的问题。例如，如果将一个具有 1000 个元组的关系分成 10 个部分，并且这个划分是偏斜的，那么就会出现一些分区内的元组数小于 100，而一些分区内的元组数大于 100。即使碰巧只有一个分区内的元组数是 200，那么我们并行存取这些分区所能得到的加速比将只能是 5，而不是我们所期望的 10。如果将同样的关系分成 100 个部分，每个分区平均含有 10 个元组。如果有一个分区内的元组数目达到 40(这是有可能的，因为分区的数量很大)，那么我们并行访问这些元组所能得到的加速比就是 25，而不是 100。因此，我们看到了随着并行度的增大，由于偏斜而导致的加速比的损失也增大。

平衡的范围划分向量(balanced range-partitioning vector)可以通过排序的方法来构建：首先将关系按划分属性进行排序，然后对关系用排序顺序进行扫描。每读过该关系的 $1/n$，就将下一个元组的划分属性值加入划分向量。这里，n 表示要创建的分区数量。在许多元组都在划分属性值上取相同值的情况下，这种技术仍然会导致某种偏斜。这种方法的主要缺点是做初始排序所带来的额外的 I/O 开销。

通过为每个关系的每个属性创建和存储该属性值的频率表或**直方图**(histogram)，可以降低由于构建平衡的范围划分向量而产生的 I/O 开销。图 18-1 是一个取值范围在 1～25 之间的整型值属性的直方图。由于直方图只占用很少的空间，因此几个不同属性上的直方图可以存储在系统目录中。给定划分属性上的直方图，可以直接构造出平衡的范围划分函数。如果没有存储直方图，可以通过对关系进行采样来近似地计算出来。计算时仅使用来自该关系的磁盘块中随机选取的子集中的元组。

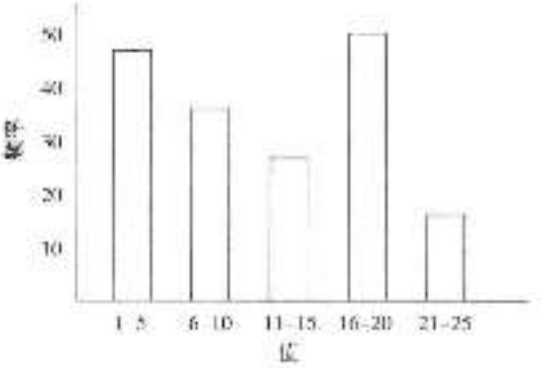


图 18-1 直方图示例

另一种使偏斜的影响达到最小的方法是使用虚处理器，特别是针对范围划分带来的偏斜。在虚处

理器(virtual processor)方案中，我们假设存在比实际处理器数量多好几倍的大量虚处理器。我们可以使用本章后面要学习到的任意划分技术以及查询求值技术，但是需要将元组和操作映射到虚处理器，而不是实际的处理器。然后再将虚处理器依次映射到实际的处理器，这一般通过轮转法划分来实现。

[801] 这种方法的思想是：即使由于偏斜使得在一个范围内有比其他范围更多的元组，这些元组也将划分到多个虚处理器范围上。轮转法将虚处理器分配到实际的处理器，这样就将额外的工作分布到多个实际的处理器中，以使一个处理器不必承担所有的负担。

18.3 查询间并行

查询间并行(interquery parallelism)中，不同查询或事务彼此并行地执行。这种形式的并行可以提高事务吞吐量。然而，单个事务的响应时间不会比事务以串行方式运行时更快。因此，查询间并行的主要用处是扩展事务处理系统，使它在每秒钟内能支持更大数量的事务。

查询间并行是数据库系统中最容易支持的一种并行形式，特别是在共享内存的并行系统中。为单处理器系统设计的数据库系统可以不做改动或者做很少的改动，就能用到共享内存的并行体系结构中，因为即使在串行的数据库系统中也支持并行处理。在串行机器中以分时并发方式运行的多个事务在共享内存的并行体系结构中将以并行的方式运行。

在共享磁盘或无共享的体系结构中支持查询间并行要更为复杂。各处理器必须以一种协同的方式来执行某些任务，例如封锁和日志，这就需要它们互相之间传送消息。并行数据库系统还必须保证两个处理器不会同时独立更新相同的数据；而且，当一个处理器访问或更新数据时，数据库系统必须保证该处理器在它的缓冲池中拥有该数据的最新版本。确保版本是最新的问题称作高速缓存一致性

[802] (cache-coherency)问题。

多种协议可用于保证高速缓存一致性；通常将高速缓存一致性协议与并发控制协议集成在一起，以减小它们的开销。用于共享磁盘系统中的一个这样的协议如下：

1. 事务对一个页面进行任何读或写访问之前，先用相应的共享或排他模式封锁该页面。一旦事务获得了页面的共享锁或排他锁后，它立刻从共享磁盘中读取该页面的最新版本。

2. 在事务释放一个页面的排他锁之前，它将该页面刷新到共享磁盘中，然后释放封锁。

这个协议保证，当事务对页面设置共享锁或排他锁时，它能得到该页面的正确版本。

更复杂的协议避免了上述协议所需要的对磁盘的重复读写。这样的协议在释放排他锁时并不将页面写到磁盘上。当事务获得共享锁或排他锁时，如果页面的最新版本在某个处理器的缓冲池中，事务就从该缓冲池中获得该页面。设计这种协议必须考虑到对并发请求的处理。对于共享磁盘协议可按如下方案扩充以适用于无共享的体系结构：每个页面有一个**本地处理器**(home processor)P_i，并存储在磁盘 D_i 中。当其他处理器想对页面进行读写时，由于它们不能与页面所在的磁盘直接通信，它们将请求发给该页面的本地处理器 P_i，其他动作与共享磁盘协议中的一样。

Oracle 和 Oracle Rdb 系统是支持查询间并行的共享磁盘并行数据库系统的实例。

18.4 查询内并行

查询内并行(intraquery parallelism)指的是单个查询在多个处理器和磁盘上并行执行。对于加快运行时间长的查询的速度，采用查询内并行非常重要。查询间并行对这样的任务没有帮助，因为每个查询还是串行运行的。

为了说明查询的并行计算，考虑一个要求对关系进行排序的查询；假设该关系已经基于其个属性通过范围划分到多张磁盘上，并且要进行的排序是基于划分属性的。排序运算可以这样实现：对每个分区并行排序，然后将排好序的各个分区连接在一起，得到最终排好序的关系。

因此，我们可以通过并行地执行各个运算来使一个查询并行化。还有另外一个对单个查询进行并行计算的来源：一个查询的运算符树可能包含多个运算。我们可以对其中某些互不依赖的运算并行地执行，从而使运算符树的计算并行化。而且，正如第 12 章所提到的，我们可以采用流水线方式，将一

[803] 个运算的输出传送给另一个运算，这两个运算可以在各自的处理器上并行地执行，一个运算正在生成

的输出(甚至在输出生成的同时)供另一个运算使用。

总的来说，单个查询的执行可以有两种不同的并行化方式：

- **操作内并行**(intraperation parallelism)。我们可以通过并行地执行每个运算，如排序、选择、投影和连接，来加快一个查询的处理速度。18.5 节将讨论操作内并行。
- **操作间并行**(interperation parallelism)。我们可以通过并行地执行一个查询表达式中的多个不同的运算，来加快一个查询的处理速度。18.6 节将讨论这种并行形式。

这两种并行形式是互相补充的，并可以同时应用在一个查询中。因为通常一个查询中的运算数目与每个运算所处理的元组数目相比是很小的，所以当并行度增大时，第一种形式取得的效果更佳。然而，由于当前一般并行系统中处理器的数目相对较小，因此两种并行形式都是重要的。

在下面关于查询的并行化讨论中，我们假设查询是只读(read only)的。并行查询计算的算法选择依赖于机器的体系结构。在描述中，我们采用无共享的体系结构模式，而不对每种体系结构都分别给出算法。因此，我们显式地给出什么时候需要将数据从一个处理器传送到另一个处理器。我们可以很容易地通过采用其他体系结构来模拟这种模式，因为在共享内存体系结构中，数据的传送可以通过共享内存来进行；在共享磁盘体系结构中，数据的传送可以通过共享磁盘来进行。因此，用于无共享体系结构的算法也可以应用到其他体系结构中。有时候我们会提到，在共享内存或共享磁盘的系统中可以如何进一步对算法进行优化。

为了简化算法表述，假定有 n 个处理器 P_0，P_1，…，P_{n-1}，和 n 块磁盘 D_0，D_1，…，D_{n-1}，其中磁盘 D_i 是与处理器 P_i 关联的。实际系统中每个处理器可能有多块磁盘。将该算法扩展到允许每个处理器有多张磁盘并不困难：我们只须让 D_i 是一组磁盘就行了。然而，为表示简单起见，我们在此假定 D_i 是单块磁盘。

18.5　操作内并行

由于关系运算在包含大量元组集合的关系上进行，因此我们可以通过在关系的不同子集上并行地执行来实现运算的并行化。由于一个关系中元组的数目可以很大，因此潜在的并行程度也很大。因此，在数据库系统中操作内并行是自然的。18.5.1～18.5.3 节将研究一些常用的关系运算的并行版本。 804

18.5.1　并行排序

假设我们希望对存放在 n 张磁盘 D_0，D_1，…，D_{n-1} 上的一个关系进行排序。如果该关系已经进行了范围划分，并且划分属性正是排序要基于的属性，那么，正如 18.2.2 节所提到的，我们可以分别对每个分区进行排序，然后把各个排序结果连接起来，得到完全排好序的关系。由于元组划分到 n 张磁盘中，因此通过并行存取减少了读取整个关系所需要的时间。

如果关系是按任意其他方式划分的，我们可以采用下述方法之一来对其排序：

1. 按排序属性对它进行范围划分，然后分别对各个分区排序。
2. 采用外部排序一归并算法的并行版本。

18.5.1.1　范围划分排序

范围划分排序(range-partitioning sort)可分为两步来进行：首先对关系进行范围划分，然后分别对每个分区进行排序。当对关系采用范围划分的方法进行排序时，没有必要将关系范围划分到它所存放的那些处理器集合或者磁盘集合上。假设我们选择处理器 P_0，P_1，…，P_{m}(其中 $m < n$)来对关系进行排序。排序运算中涉及两个步骤：

1. 采用范围划分策略对关系中的元组进行重新分布，使得处于第 i 个范围的所有元组都发送给处理器 P_i，它将关系临时存放在磁盘 D_i 中。

为了实现范围划分，每个处理器并行地从它的磁盘上读取元组，并将这些元组发送到它们的目的处理器。每个处理器 P_0，P_1，…，P_m 也都会接收到属于其分区的那部分元组，并且在本地存储这些元组。这一步需要磁盘 I/O 和通信开销。

2. 每个处理器在本地对关系中属于自己的分区进行排序，不与其他处理器交互。每个处理器在不同的数据集上执行相同的运算，即排序。在不同的数据集上并行地执行相同的运算称作**数据并**

行(data parallelism)。

最后的合并运算非常简单，因为开始阶段的范围划分就保证了对于 $1 \leq i < j \leq m$，处理器 P_i 中的码值全都小于处理器 P_j 中的码值。

[805] 我们必须使用一个好的范围划分向量来进行范围划分，这样每个分区中的元组数目就会大致相同。也可以使用直方图划分来减少偏斜。

18.5.1.2 并行的外部排序归并

除了范围划分外，另外一种替代方法是**并行的外部排序归并**(parallel external sort-merge)。假设关系已经划分到磁盘 D_0，D_1，…，D_{n-1} 中，(对关系进行怎样的划分无关紧要)。并行的外部排序归并按如下步骤进行：

1. 每个处理器 P_i 在本地对磁盘 D_i 中的数据进行排序。
2. 然后，系统对每个处理器已排好序的归并段进行合并，得到最终的排序输出。

第 2 步中对已排序归并段的合并可以按下述动作序列并行执行：

1. 系统将每个处理器 P_i 排好序的分区范围划分(都采用相同的划分向量)到处理器 P_0，P_1，…，P_{n-1} 中。系统用排序顺序发送元组使得每个处理器接收的都是排好序的元组流。
2. 每个处理器 P_i 在接收到元组流时进行归并，得到一个已排好序的归并段。
3. 系统将处理器 P_0，P_1，…，P_{n-1} 中排好序的归并段串接起来，得到最终结果。

如上所述的动作序列导致一种有趣的**执行偏斜**(execution skew)形式，因为一开始每个处理器都把分区 0 中的所有块发送到 P_0，然后每个处理器把分区 1 中的所有块发送到 P_1，依此类推。因此，在并行发送的时候，元组的接收却变成了串行的：首先只有 P_0 接收元组，接着只有 P_1 接收元组，依此类推。为了避免这个问题，每个处理器都重复地向每个分区发送一个数据块。换句话说，每个处理器发送每个分区的第一个数据块，然后发送每个分区的第二个数据块，依此类推。其结果是，所有处理器都会并行地接收数据。

有些机器，例如 Teradata Purpose-Built Platform Family 机，采用专门的硬件来执行归并。Teradata 机器中的内连网 BYNET 可以归并来自多个处理器的输出，以得到单个排好序的输出。

18.5.2 并行连接

连接运算需要系统测试元组对，看它们是否满足连接条件；如果满足，系统就把该元组对加到连接的输出中。并行连接算法设法将这些待检测的元组对划分到多个处理器上，然后每个处理器在本地

[806] 计算部分连接。于是，系统从各个处理器收集结果，以产生最终结果。

18.5.2.1 基于划分的连接

对于某些类型的连接，例如等值连接和自然连接，可以将两个输入关系划分到多个处理器上，并在每个处理器上计算本地的连接。假定我们使用 n 个处理器，要连接的关系是 r 和 s。**基于划分的连接**(partitioned join)以这样的方式工作：系统将关系 r 和 s 分别分成 n 个分区，记为 r_0，r_1，…，r_{n-1} 和 s_0，s_1，…，s_{n-1}。系统将划分 r_i 和 s_i 送到处理器 P_i，并在本地进行其连接计算。

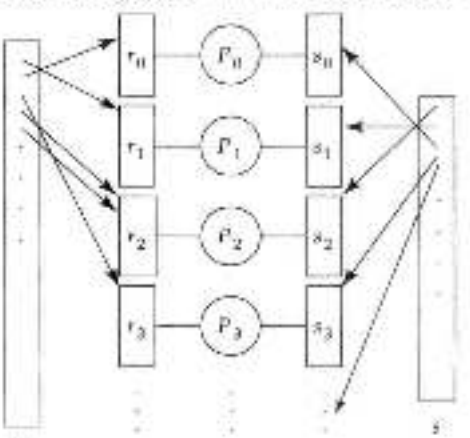


图 18-2 基于划分的并行连接

只有在连接是等值连接(例如，$r \bowtie_{r.A=s.B} s$)而且使用在它们的连接属性上同样的划分函数对关系 r 和 s 进行划分的情况下，基于划分的连接技术才能正确工作。划分的思想与散列连接的划分步骤的思想完全一样。然而，在基于划分的连接中，有两种不同的划分 r 和 s 的方法：

- 基于连接属性进行范围划分。
- 基于连接属性进行散列划分。

不管采用何种划分，都必须对两个关系使用相同的划分函数。对于范围划分，两个关系所采用的划分向量必须相同。对于散列划分，两个关系所使用的散列函数必须相

同。图 18-2 描述了基于划分的并行连接中所采用的划分方法。

一旦对关系进行了划分，我们就可以在每个处理器 P_i 上采用任意一种连接技术在本地计算 r_i 和 s_i 的连接。例如，散列连接、归并连接或嵌套循环连接都可以使用。因此，我们可以采用划分的方法使任何连接技术并行化。

807

如果关系 r 和 s 中有一个或两个已经基于连接属性进行了划分（通过散列划分或范围划分），那么划分所需做的工作就大大减少了。如果关系没有进行划分，或者不是基于连接属性进行的划分，那么就需要对元组进行重新划分。每个处理器 P_i 读入磁盘 D_i 中的元组，对于每个元组 t，计算出它所属的分区 j，并把元组 t 发送给处理器 P_j，处理器 P_j 将元组存放到磁盘 D_j 上。

通过将某些元组放在内存的缓冲区中，而不是写到磁盘中，可以减少 I/O，从而优化每个处理器本地使用的连接算法。18.5.2.3 节描述这样的优化。

当采用范围划分时，偏斜就表现为一个特殊的问题，因为将参与连接的一个关系划分为同等大小的分区的划分向量可能将另一个关系划分为大小差异很大的分区。划分向量应该使 $|r_i| + |s_i|$（即 r_i 和 s_i 的大小之和）对于所有的 $i=0, 1, \cdots, n-1$ 大致相等。如果散列函数选得好，散列划分可能具有更小的偏斜，除非有许多元组在连接属性上取相同的值。

18.5.2.2　分片 - 复制连接

划分并非对所有类型的连接都适用。例如，如果连接条件是一个不等式，如 $r \bowtie_{r.A<s.B} s$，那么有可能 r 中的所有元组与 s 中的某些元组相连接（反之亦然）。因此，可能没有一种容易地划分 r 和 s 的方法，使得分区 r_i 中的元组只与分区 s_i 中的元组相连接。

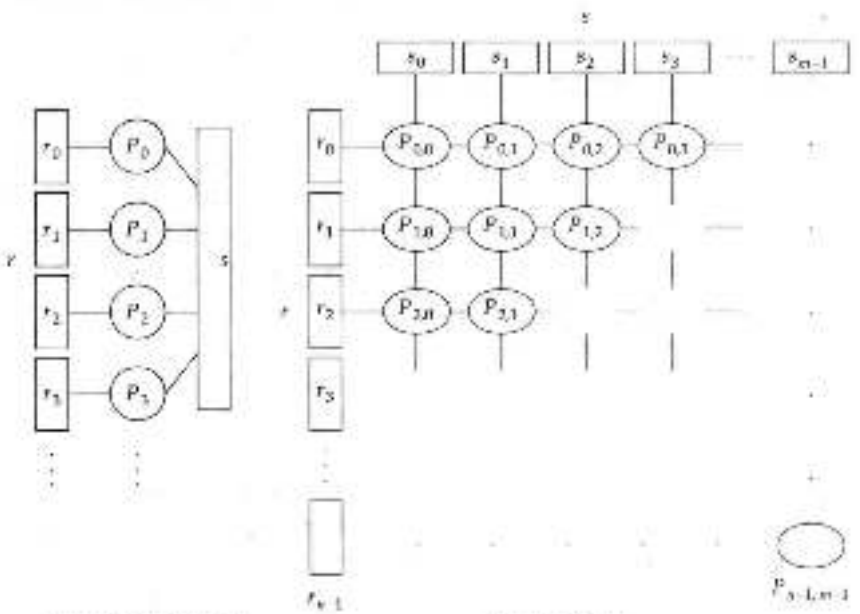
我们可以采用一种称作分片 - 复制的技术来并行地执行这种连接。首先考虑分片 - 复制的一种特殊情况：**非对称的分片 - 复制连接**（asymmetric fragment-and-replicate join），其工作方法如下：

1. 系统对其中一个关系，例如 r，进行划分。任何一种划分技术可以用在关系 r 上，包括轮转法划分。

2. 系统将另一个关系 s 复制到所有的处理器上。

3. 然后，处理器 P_i 采用任何一种连接技术在本地计算 r_i 和整个 s 的连接。

非对称的分片 - 复制模式如图 18-3a 所示。如果 r 已经是经过划分存储的，则不必执行第 1 步的划分，只须将 s 复制到所有处理器上就可以了。



a）非对称的分片和复制　　b）分片和复制

图 18-3　分片 - 复制模式

分片－复制连接(fragment-and-replicate join)的一般情况如图 18-3b 所示。其工作方式如下：系统将关系 r 划分为 n 个分区 $r_0, r_1, \cdots, r_{n-1}$，并将关系 s 划分为 m 个分区 $s_0, s_1, \cdots, s_{m-1}$。同前面一样，对 r 和 s 可以采用任意的划分技术。m 和 n 的值不必相等，但它们值的选择必须保证至少有 $m * n$ 个处理器。非对称的分片和复制是一般的分片和复制的一种特殊情况，即 $m = 1$。与非对称的分片和复制相比，分片和复制减小了每个处理器上关系的规模。

设处理器标记为 $P_{0,0}, P_{0,1}, \cdots, P_{0,n-1}, P_{1,0}, \cdots, P_{m-1,n-1}$。处理器 $P_{i,j}$ 计算 r_i 和 s_j 的连接。每个处理器必须得到它所处理的分区中的那些元组，为了做到这一点，系统将 r_i 复制到处理器 $P_{i,0}, P_{i,1}, \cdots, P_{i,n-1}$（这形成了图 18-3b 中的一行），并将 s_i 复制到处理器 $P_{0,i}, P_{1,i}, \cdots, P_{m-1,i}$（这形成了图 18-3b 中的一列）。在每个处理器 $P_{i,j}$ 上可以采用任何一种连接技术。

对于任何连接条件都可以采用分片和复制方法，因为 r 中的每个元组都能与 s 中的每个元组对应进行检测。所以，在不能采用基于划分的连接的情况下可以采用这种方法。

当两个关系的规模基本相同时，分片和复制连接的代价通常比基于划分的连接要高，因为分片－复制连接必须至少要复制其中的一个关系。然而，如果其中一个关系(例如 s)很小，那么将 s 复制到所有处理器上可能比基于连接属性对 r 和 s 进行划分代价要小。在这种情况下，即使能够采用基于划分的连接，也最好采用非对称的分片和复制连接。

18.5.2.3 基于划分的并行散列连接

808 | 809

12.5.5 节所讲的基于划分的散列连接可以并行地执行。假设有 n 个处理器 $P_0, P_1, \cdots, P_{n-1}$，以及两个关系 r 和 s，r 和 s 被划分到多张磁盘中。回忆 12.5.5 节，应该选用较小的关系作为建立散列表的关系。如果 s 的规模比 r 小，那么并行的散列连接算法按如下步骤执行：

1. 选择一个散列函数，例如 h_1，它取得 r 和 s 中每个元组的连接属性的值，并将元组映射到 n 个处理器中的一个。令 r_i 表示关系 r 中映射到处理器 P_i 上的元组；类似地，令 s_i 表示关系 s 中映射到处理器 P_i 上的元组。每个处理器 P_i 读取 s 中位于其磁盘 D_i 上的元组，并基于散列函数 h_1 将每个元组发送到适当的处理器上。

2. 当目的处理器 P_i 接收到 s_i 的元组时，它用另一个散列函数 h_2 对它们作进一步划分，处理器用该散列函数在本地计算散列连接。这一步的划分与串行的散列连接算法的划分阶段完全一样。每个处理器 P_i 在执行这个步骤时与其他处理器完全独立。

3. 一旦将 s 的元组进行了分布之后，系统采用与前面相同的方法，通过散列函数 h_1 将较大的关系 r 在 n 个处理器上进行重新分布。目的处理器在接收到每个元组时，用散列函数 h_2 对它进行重新划分，就像串行散列连接算法中对探查关系的划分一样。

4. 每个处理器 P_i 对 r 和 s 的本地分区 r_i 和 s_i 上执行散列连接算法的创建和探查过程，产生散列连接最终结果的一个分区。

每个处理器上的散列连接与其他处理器之间是相互独立的，并且接收 r_i 和 s_i 的元组类似于从磁盘中读取这些元组。因此，第 12 章描述的散列连接的任何一种优化也都可以应用于并行的情况。特别地，我们可以采用混合式散列连接算法，将某些输入元组缓存在内存中，从而避免将它们写出以及再读入的代价。

18.5.2.4 并行嵌套循环连接

为了说明基于分片－复制并行化的使用，考虑关系 s 比关系 r 小得多的情况。假设关系 r 是采用划分的方式存储的，其基于划分的属性是无关紧要的。还假设在关系 r 的每一个分区上存在关系 r 在连接属性上的一个索引。

我们采用非对称的分片和复制方法，将关系 s 进行复制，并利用关系 r 现有的分区。存放关系 s 分区的每个处理器 P_i 读取存放在 D_i 上的关系 s 中的元组，并将这些元组复制到其余的每个处理器 P_j 中。在这个阶段的最后，关系 s 被复制到存放关系 r 的元组的所有站点中。

现在，每个处理器 P_i 对关系 s 和关系 r 的第 i 个分区进行索引嵌套循环连接。我们可以将索引嵌套循环连接与对关系 s 的元组划分重叠起来做，从而减少将关系 s 的元组写到磁盘、再读出来的代价。然

810 | 面，关系 s 的复制必须与连接同步，从而在每个处理器 P_i 的内存缓冲区中能有足够的空间来存放已经

接收到但尚未用于连接的关系 s 的元组。

18.5.3　其他关系运算

其他关系运算的实现也可以并行化：

- **选择**(selection)。令选择运算为 $\sigma_\theta(r)$。首先考虑 θ 形如 $a_i = v$ 的情况，其中 a_i 是一个属性，v 是一个值。如果关系 r 是基于属性 a_i 进行划分的，那么选择在单个处理器中执行。如果 θ 形如 $l \leq a_i \leq u$，即 θ 是一个范围选择，而且该关系基于 a_i 进行了范围分布，那么该选择在其分区与指定值范围相重叠的每个处理器中执行。在所有其他情况下，选择在所有处理器中并行地执行。
- **去重**(duplicate elimination)。去重可以通过排序来进行，可以使用任意一种并行排序技术，并优化为一旦在排序过程中出现重复就将其消除掉。我们还可以通过将元组进行划分(采用范围划分或散列划分)并在每个处理器中进行本地去重，来使消除重复并行化。
- **投影**(projection)。不去重的投影可以在元组从磁盘读入时并行地进行。如果要消除重复，那么可以采用刚才描述的任意技术。
- **聚集**(aggregation)。考虑一个聚集运算。通过将关系基于分组属性进行划分，然后在每个处理器中本地计算聚集的值，我们可以使运算并行化。采用散列划分或者范围划分都可以。如果关系已经基于分组属性进行了划分，那么可以跳过第一步。

  我们可以在划分前部分地计算聚集的值，来减少划分时传输元组的代价，至少对于常用的聚集函数可以这样做。考虑关系 r 上的一个聚集运算，基于属性 A 分组，在属性 B 上使用聚集函数 **sum**。系统可以在每个处理器 P_i 上对存储在磁盘 D_i 中的 r 元组执行该运算。这个计算在每个处理器上得到了具有部分总和的元组：对于磁盘 D_i 中存放的 r 元组中在属性 A 上的每个取值，都有 P_i 中的一个元组与之对应。系统对分组属性 A 的本地聚集结果进行划分，然后在每个处理器 P_i 上再次进行聚集运算(在具有部分总和的元组上进行)，以得到最终结果。

  这种优化的结果是，划分过程中需要送到其他处理器上的元组数目更小。可以很容易地将这一想法扩展到 **min** 和 **max** 聚集函数上。对 **count** 和 **avg** 聚集函数的扩展留在习题 18.12 中去做。

其他运算的并行化在一些习题中有阐述。

18.5.4　运算的并行计算代价

我们通过将 I/O 分布到多张磁盘上，并将 CPU 工作分布到多个处理器上来达到并行。如果这样的划分不需要任何开销就能实现，如果在工作的划分中没有偏斜，那么使用 n 个处理器的并行运算所用的时间就是单个处理器上执行相同运算所用时间的 $1/n$。我们已经知道如何估算诸如连接或是选择那样的运算的代价。那么并行处理的时间代价就是串行处理相同运算的时间代价的 $1/n$。

我们还必须考虑以下代价：

- **启动代价**(start-up cost)。用于在多个处理器上初始化运算。
- **偏斜**(skew)。在多个处理器之间划分工作而产生的，有些处理器得到的元组数目多于其他处理器。
- **对资源的竞争**(contention for resource)。例如对内存、磁盘、通信网络的竞争所导致的延迟。
- **组装代价**(cost of assembling)。从每个处理器传送过来的部分结果组成最终结果所花费的代价。

并行运算所需的时间可以估算为：

$$T_{part} + T_{asm} + \max(T_0, T_1, \cdots, T_{n-1})$$

其中 T_{part} 是用于划分关系的时间，T_{asm} 是组装结果所需的时间，T_i 是处理器 P_i 执行运算所需的时间。假设元组的分布没有任何偏斜，那么送到每个处理器的元组数可以估算为元组总数的 $1/n$。再忽略掉竞争，于是每个处理器 P_i 执行运算的代价 T_i 就可以通过第 12 章所描述的技术估算出来。

上述估算是一种乐观的估算，因为通常是有偏斜的。虽然将单个查询分解成若干个并行的步骤减少了平均步骤的规模，但处理整个查询所需的时间是由最慢的那个步骤的处理时间决定的。例如，基

于划分的并行计算的速度仅与并行执行中最慢的那个速度相同。因此，在处理器间划分工作时的任何偏斜都会对性能造成很大影响。

划分中的偏斜问题与串行散列连接中的划分溢出问题（见第 12 章）密切相关。当进行散列划分时，我们可以采用为散列连接所研制的解决溢出和避免溢出的技术来处理偏斜。我们可以像 18.2.3 节描述
812 的那样，使用平衡的范围划分和虚处理器划分来将范围划分导致的偏斜降到最小。

18.6 操作间并行

操作间并行有两种形式：流水线并行和独立并行。

18.6.1 流水线并行

正如第 12 章所讨论的，流水线方式是减小数据库查询处理计算代价的重要途径。回顾在流水线中，甚至在第一个运算 A 尚未产生出完全的输出元组集合之前，运算 A 的输出元组就被第二个运算 B 使用。在串行计算中采用流水线执行的主要优点是可以不必往磁盘中写任何中间结果，就将一系列这样的运算进行下去。

并行系统采用流水线的主要原因与串行系统相同。但同时，流水线也可以提供并行性，就像硬件设计中采用指令的流水线提供并行性一样。可以在不同的处理器上同时运行 A 和 B，使得在 A 产生结果元组的同时，B 使用它们。这种形式的并行称作**流水线并行**（pipelined parallelism）。

考虑 4 个关系上的连接：

$$r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$$

我们可以建立一条流水线，允许三个连接并行地计算。假设指定处理器 P_1 计算 $temp_1 \leftarrow r_1 \bowtie r_2$，指定 P_2 计算 $r_3 \bowtie temp_1$。当 P_1 计算出 $r_1 \bowtie r_2$ 中的元组时，它就将这些元组提供给处理器 P_2。于是在 P_1 完成它的计算之前，P_2 就得到了 $r_1 \bowtie r_2$ 中的一些元组。P_2 可以利用它得到的元组开始计算 $temp_1 \bowtie r_3$，尽管这时 P_1 还没有完全计算出 $r_1 \bowtie r_2$。与此类似，当 P_2 计算出 $(r_1 \bowtie r_2) \bowtie r_3$ 中的元组时，它就将这些元组提供给 P_3，由 P_3 来计算这些元组与 r_4 的连接。

当只有少量处理器时，流水线并行是有用的，但它的可扩展性不好。首先，流水线的链一般来说达不到足够的长度来提供较高的并行度。其次，对于那些直到访问了所有输入才产生输出的关系运算（例如集差运算）不能采用流水线。最后，对于一种运算的执行代价比其他运算高得多这种经常出现的情况，流水线方法只能获得很小的加速比。

考虑所有情况，当并行度较高时，流水线对于提高并行度的重要性不如基于划分的并行。采用流
813 水线的真正原因是流水线执行可以避免将中间结果写到磁盘中。

18.6.2 独立并行

查询表达式中互不依赖的运算可以并行地执行。这种并行形式称作**独立并行**（independent parallelism）。

考虑连接 $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$。显然，我们可以并行地计算 $temp_1 \leftarrow r_1 \bowtie r_2$ 和 $temp_2 \leftarrow r_3 \bowtie r_4$。当这两个计算完成后，我们计算：

$$temp_1 \bowtie temp_2$$

为进一步提高并行性，我们可以将 $temp_1$ 和 $temp_2$ 中的元组流水线化，作为对 $temp_1 \bowtie temp_2$ 计算的输入，然后对它用流水线连接来计算（见 12.7.2.2 节）。

独立并行与流水线并行相同，也不能提供较高的并行度，在高度并行的系统中也没有太大的用处。尽管在低度并行中它是有用的。

18.7 查询优化

关系技术取得成功的一个重要原因是查询优化器。回忆一下，查询优化器分析一个查询，在给出相同结果的多个可能的执行计划中找出代价最低的执行计划。

并行查询计算的查询优化器比串行查询计算的查询优化器更为复杂。首先，代价模型更复杂，因为必须将划分代价考虑在内，而且还必须考虑诸如偏斜、资源竞争那样的问题。更重要的问题是如何

进行查询的并行化。假设我们以某种方式选出了一个表达式(从表达查询的多个等价的表达式中)，用来计算该查询。12.1节已经讨论过，可以用运算符树来表示这个表达式。

为了在并行系统中计算一个运算符树，我们必须做出以下决定：

- 如何对每个运算进行并行化，为每个运算采用多少个处理器。
- 在多个处理器上对哪些运算进行流水线化，哪些运算以并行方式独立地执行，哪些运算一个接一个地串行执行。

这些决定构成了调度(scheduling)该执行树的任务。

优化问题的另一个方面是确定应该分配给执行树中每个运算的各种类型的资源，如处理器、磁盘和内存。例如，可最大限度地利用并行的资源可能看起来很明智，但对特定运算不并行地执行也是个好主意。那些计算需求比通信开销小得多的运算应该和其邻近的某个运算聚集在一起，否则并行的优点就被通信开销给抵消了。 814

需要考虑的一个问题是长的流水线并不有利于其很好地利用资源。除非运算是粗粒度的，否则流水线中最后一个运算可能要等待很长时间才能得到输入，而它却占据着宝贵的资源，例如内存。因此，应该避免长的流水线。

并行执行计划的候选数目比串行执行计划的数目要大得多。因此，通过考虑所有的候选计划来优化并行查询比优化串行查询的代价要高得多。所以，我们通常采用启发式方法来减少需要考虑的并行执行计划的数目。这里描述两种常用的启发式方法。

第一种启发式方法是仅考虑那些利用所有的处理器，对每个运算都并行化，并且不采用任何流水线的执行计划。Teradata系统中采用的就是这种方法。寻找这种最佳执行计划类似于在串行系统中做查询优化。主要区别在于如何进行划分，以及采用何种代价估算公式。

第二种启发式方法是选择最高效的串行执行计划，然后将该执行计划中的运算并行化。Volcano并行数据库使一种称为**交换算子**(exchange-operator)模型的并行化模型流行起来。此模型使用现有运算的实现，在数据的本地拷贝上进行处理，并结合交换操作，也就是在不同处理器间移动数据。交换算子可被引入到执行计划中，以将执行计划转化为并行执行计划。

还有另外一个优化的维度是设计物理存储组织来加快查询。不同查询有不同的最佳物理组织。数据库管理员必须选择一种适合于预期的混合数据库查询的物理组织。因此，并行查询优化领域很复杂，而且依然是一个活跃的研究领域。

18.8 并行系统设计

到目前为止本章一直关注于数据存储和查询处理的并行化。由于大规模并行数据库系统主要用于存储大量的数据，并用于处理基于这些数据的决策支持查询，因此这些主题在并行数据库系统中是最重要的。如果我们要处理大量的输入数据，那么来自外部数据源的数据并行加载是一个重要的需求。

一个大型的并行数据库还必须注意这些可用性问题：

- 在某些处理器和磁盘发生故障时的恢复性。
- 数据和模式发生改变时的联机重组。

在此我们来讨论这些问题。 815

由于具有大量的处理器和磁盘，因此至少有一个处理器或磁盘发生故障的概率就会比具有单磁盘的单处理器系统要大得多。如果并行系统设计得不好，那么当任何一个部件(处理器或磁盘)发生故障时，系统就会停止工作。假设单个处理器或磁盘发生故障的概率很小，那么系统发生故障的概率随处理器和磁盘的数目呈线性增长。如果单个处理器或磁盘每5年发生一次故障，那么具有100个处理器的系统每18天就会发生一次故障。

因此，像Teradata以及IBM Informix XPS这样的大规模并行数据库系统都设计成即使一个处理器或磁盘发生了故障，系统还能运行。数据至少复制到两个处理器上。如果一个处理器失效，它所存储的数据还能够从另一个处理器访问到。系统追踪失效的处理器，并将其工作分配给能正常运行的处理器。

对于失效站点中所存储数据的请求将自动发送到存放了该数据副本的备份站点上去。如果处理器 A 的所有数据都复制到一个单独的处理器 B 中，那么 B 将必须处理对 A 的所有请求，同时还要处理对它自己的请求，这就会使 B 成为瓶颈。因此，一个处理器中数据的副本应该分配到多个其他处理器中。

当我们处理大量的数据(TB 范围内)时，像创建索引这样的简单操作，以及在关系中增加一列这样的模式改变，会花费很长的时间——可能要数小时甚至数天。因此如果在进行这样的操作的过程中让数据库不可用是无法接受的。大多数数据库系统允许这样的操作联机(online)运行，即可以在系统执行其他事务的同时进行这样的操作。

例如，考虑**联机索引创建**(online index construction)。支持这一特性的系统即使正在某个关系上创建索引时，也允许在该关系上进行插入、删除和更新。因此，建立索引操作不能像一般情况下那样以共享模式封锁整个关系，而是需要该进程记住在它活跃期间所发生的更新，并将这些变化合并到正在建立的索引中(目前大多数数据库系统支持联机索引创建，因为这一特性非常重要，即使是对于非并行的数据库系统)。

近年来，许多公司已经开发出新的并行数据库产品，包括 Netezza、DATAllegro(已被微软收购)、Greenplum 和 Aster Data。每个这样的产品运行在包含几十个到几千个节点的系统上，每个节点运行一个底层数据库实例；每个产品在多个数据库实例上管理自己的数据分区，以及并行查询处理。

Netezza、Greenplum 和 Aster Data 使用 PostgreSQL 作为底层数据库；DATAllegro 原本采用 Ingres 作为底层数据库系统，但被微软收购之后就转到 SQL Server 上了。通过建立在现有数据库系统之上的方式，这些系统能够充分利用底层数据库的数据存储、查询处理和事务管理特征，让它们能够将更多精力
816 专注于数据分区(包括复制和容错)，加快进程间通信，并行查询处理，以及并行查询优化。使用诸如 PostgreSQL 那样的公共领域数据库的另一个好处是，每个结点的软件成本非常低；相反，商用数据库在每个处理器上有着很高的成本。

还值得一提的是，Netezza 和 DATAllegro 实际上销售的是数据仓库"设备"，包括硬件和软件，允许客户以最小代价建立并行数据库。

18.9 多核处理器的并行性

并行性在当今大多数计算机中已经司空见惯，甚至是在一些最小的计算机中，那是由于日益计算机体系结构的趋势所致。其结果是，现今所有数据库系统实际上都运行在并行平台上。本节将简要地探讨这一架构趋势出现的原因及其对数据库系统设计和实现的影响。

18.9.1 并行性与原始速度

由于计算机的出现，处理器速度呈指数级增长，每 18～24 个月增加一倍。这种增长源于在一块单位面积的硅芯片上能够放入的晶体管数目是呈指数增加的，即众所周知的摩尔定律(Moore's law)，其命名来自英特尔联合创始人戈登·摩尔(Gordon Moore)。从技术上讲，摩尔定律并不是定律，而是一个关于技术发展趋势的观察和预测。直到最近，晶体管数量的增长及其大小的减少导致不断出现更快的处理器。尽管技术发展继续符合摩尔定律的预测，但出现了另一个因素使得处理器速度的增长放缓。快速的处理器在功耗上是低效的。这会造成功耗和成本、移动电脑的电池寿命和热散逸(使用的所有能量最终都转换化为热)方面的问题。因此，现代的处理器通常不是单处理器，而是由一块芯片上的多个处理器组成。为了保持在单芯片上的多处理器与传统处理器之间的区别，核(core)这个术语用于单芯片处理器。这样我们可以说一台机器上有一个多核处理器。[⊖]

18.9.2 高速缓冲存储器和多线程

每个核都具备处理一个独立的机器指令流的能力。但是，由于处理器处理数据的速度比从主存上访问数据要快，因此主存可能成为限制整体性能的瓶颈。出于这个原因，电脑设计师在计算机系统中设置了包括一个或多个级别的高速缓冲(cache)存储器。高速缓冲存储器的每字节成本比主内存高，但

⊖ 在此处用的核这个术语和早期计算中所使用的术语不同，后者指基于硅性核心的工存技术。

提供了更快的存取时间。在多级高速缓存设计中，高速缓存分成 L_1、L_2 等级别，其中 L_1 是最快的高速缓存（因此每字节的价格也最贵，所以容量也最小）。L_2 是第二快的，依此类推。其结果是将第 10 章讨论的存储体系架构扩展到在主存下包含不同级别的高速缓存。 817

尽管数据库系统可以控制在磁盘和主存之间传输数据，但电脑硬件仍可控制在各个级别的高速缓存之间以及在高速缓存与主存之间的数据传输。尽管缺乏这种直接控制，数据库系统的性能仍然受到高速缓存使用方式的影响。如果一个线程要访问的数据项不在高速缓存中，那么它必须从主存获取。因为主存比处理器要慢得多，所以当一个核等待来自主存的数据时，处理速度就可能会蒙受巨大损失。这种等待称为高速缓存失效（cache miss）。

计算机设计者试图减小高速缓存失效所带来的影响的一种方式是采用多线程（multithreading）。线程（thread）是一个执行流，它与运行在同一个核上的其他线程共享内存。[⊖]如果在一个核上当前执行的线程遇到高速缓存失效（或其他类型的等待），那么核就会执行另外的线程，从而在等待时不浪费计算速度。

线程与了在多核之上的另一种并行性。每个新一代的处理器都支持更多的核和更多的线程。Sun UltraSPARC T2 处理器有 8 个核，每个核能支持 8 个线程。总的来说，在一块处理器芯片上能支持 64 个线程。

原始速度增长放缓同时核数量增长这样的体系架构上的趋势对数据库系统设计产生了重大影响，这点我们很快就会看到。

18.9.3 适应现代体系架构的数据库系统设计

可能看起来数据库系统是一个有效利用大量核和线程的理想范围，因为数据库系统支持大量并发事务。然而，有很多因素使得对现代处理器的优化利用充满了挑战。

当我们允许更高的并发度去利用现代处理器的并行性时，我们会增加高速缓存中的数据量。这可能会导致更多的高速缓存失效，甚至可能多到需要多线程的核来等待来自主存中的数据。

并发事务需要某种形式的并发控制来保证第 14 章讨论过的 ACID 属性。当并发事务访问相同数据时，必须对该并发访问施加某种限制。这些限制，无论是基于封锁、时间戳还是验证，都会导致等待或由于事务中止而损失工作。为了避免长时间等待或大量的工作损失，理想情况是并发事务的冲突很少，但试图确保减少冲突会增加需要存放到高速缓存中的数据量，并导致更多的高速缓存失效。

最后，存在一些被所有事务所共享的数据库系统组件。在使用封锁的系统中，锁表被所有事务共享，对锁表的访问可能会成为瓶颈。类似的问题也同样存在于其他形式的并发控制中。同样，服务于所有事务的缓冲管理器、日志管理器和恢复管理器也是潜在的瓶颈。 818

由于具有大量的并发事务可能不能最好地利用现代处理器的优势，因此需要找到允许多核在单个事务上工作的方式。这就要求数据库查询处理器找到有效的并行查询方式，而不会对高速缓存产生过多的需求。这可以通过建立数据库查询操作的流水线以及通过寻找单个数据库操作的并行化方式来实现。

适应于多核和多线程系统的数据库系统设计和数据库查询处理仍然是一个活跃的研究领域。更多细节请参见文献注解。

18.10 总结

- 在过去 20 年中，并行数据库已经得到了广泛的商业认可。
- 在 I/O 并行中，把关系划分到多张可用的磁盘中，从而使检索速度更快。三种常用的划分技术是轮转法划分、散列划分和范围划分。
- 偏斜是一个主要的问题，特别是当并行度增高时。平衡的划分向量、使用直方图以及虚处理器划分是用于减少偏斜的技术。
- 在查询间并行中，我们并发地运行不同的查询以提高吞吐量。

⊖ 在技术上，以操作系统的术语来讲，称其执行实体。

- 查询内并行试图减少运行查询的代价。有两类查询内并行：操作内并行和操作间并行。
- 我们采用操作内并行来并行地执行关系运算，例如排序和连接。因为关系运算是面向集合的，所以操作内并行对关系运算处理自然的。
- 对于像连接这样的二元运算，有两种基本的并行化方法。
  - 在基于划分的并行中，两个关系分成几个部分，而且 r 中的元组仅与 s 中的元组进行连接。基于划分的并行仅适用于自然连接和等值连接。
  - 在分片和复制中，两个关系都被划分，并且每个划分都被复制。在非对称的分片和复制中，一个关系被复制，而另一个关系被划分。与基于划分的并行不同，分片和复制以及非对称的分片和复制对于任何连接条件都适用。

819

这两种并行技术都可以与任何一种连接技术结合使用。

- 在独立的并行中，互不依赖的多个不同的操作按并行方式执行。
- 在流水线并行中，处理器在计算一个操作结果的同时将结果发送给另一个操作，无须等待整个操作的完成。
- 并行数据库中的查询优化比串行数据库中的查询优化要复杂得多。
- 现代的多核处理器引入了并行数据库中新的研究问题。

术语回顾

- 决策支持查询
- I/O 并行
- 水平划分
- 划分技术
  - 轮转法
  - 散列划分
  - 范围划分
- 划分属性
- 划分向量
- 点查询
- 范围查询
- 偏斜
  - 执行偏斜
  - 属性值偏斜
  - 划分偏斜
- 偏斜处理
  - 平衡的范围划分向量
  - 直方图
  - 虚处理器
- 查询间并行
- 高速缓存一致性
- 查询内并行
  - 操作内并行
  - 操作间并行
- 并行排序
  - 范围划分排序
  - 并行的外部排序归并
- 数据并行
- 并行连接
  - 基于划分的连接
  - 分片 - 复制连接
  - 非对称的分片 - 复制连接
  - 基于划分的并行散列连接
  - 并行嵌套循环连接
- 并行选择
- 并行去重
- 并行投影
- 并行聚集
- 并行计算的代价
- 操作间并行
  - 流水线并行
  - 独立并行
- 查询优化
- 调度
- 交换算子模型
- 并行系统设计
- 联机索引创建
- 多核处理器

实践习题

18.1 当在范围划分属性上进行范围选择时，可能仅需要访问一张磁盘，请描述这一特性的优点和缺点。

18.2 对于下述每种任务，哪种并行形式（查询间并行、操作间并行或操作内并行）可能是最重要的？

a. 提高具有许多小查询的系统的吞吐量。

b. 在磁盘和处理器的数目都很大的情况下，提高具有少量大查询的系统的吞吐量。

18.3 对于流水线并行，即便当有许多处理器可用时，在单个处理器上用流水线来执行多个操作通常是个好主意。

a. 请解释原因。

b. 如果机器采用共享内存体系结构，你在 a 部分提出的论据还成立吗？请解释成立或不成立的原因。

c. 对于独立的并行，a 部分提出的论据成立吗？（也就是说，即使在不将操作组织到流水线中，而且有许多处理器可用的情况下，在同一个处理器上执行几个操作是否仍然是个好主意呢？）

18.4 考虑采用范围划分的对称的分片和复制来处理连接。如果连接条件形如 $|r.A - s.B| \leq k$，其中 k 是一

个小常数，$|x|$表示取 x 的绝对值，带有这种连接条件的连接称作**带状连接**(band join)，那么你如何来优化查询计算？

18.5 回顾一下，可以利用直方图来建立负载均衡的范围划分。

a. 假设你有一个取值范围为 1～100 的直方图，划分为 10 个范围：1～10，11～20，…，91～100，其出现频率分别为 15，5，20，10，10，5，5，20，5，5。请给出一个负载均衡的范围划分函数，将这些值划分成 5 个部分。

b. 请写出一个算法，对于给定的包括 n 个范围的频率分布直方图，计算出划分为 p 个分区的一个均衡的范围划分。

820
↓
821

18.6 大规模并行数据库系统对每个数据项都会在附属于不同处理器的磁盘上存储一个额外的副本，以避免当一个处理器失效时所引起的数据丢失。

a. 不是把来自一个处理器的数据项的额外副本保存到一个单点备份的处理器上，更好的方法是把一个处理器的数据项副本划分到多个处理器上。请解释原因。

b. 解释如何利用虚处理器划分来有效实施前述副本的划分。

c. 如果采用 RAID 存储，而不是存储每个数据项的额外副本，有哪些优点和缺点？

18.7 假设我们要对一个已经划分的关系进行索引。这种划分的思想(包括虚处理器划分)能够应用于索引吗？解释你的答案，考虑下面两种情况(为了简单起见，假设划分和索引都是在单个属性上进行的)。

a. 索引建立在关系的划分属性上。

b. 索引基于的属性不是关系的划分属性。

18.8 假设已经为一个关系选好了非常平衡的范围划分向量，但是这个关系随后更新了，导致划分失衡。即使采用虚处理器划分，某个虚处理器最终也可能在更新后得到相当大量的元组，从而需要重新划分。

a. 假设一个虚拟处理器有相当多的元组(比如说，超过平均值的两倍)。解释如何通过分裂分区从而增加虚处理器的数量来实现重新划分。

b. 如果虚处理器不是轮流分配的，虚拟分区可以以任意的方式分配给处理器，并使用映射表来记录分配信息。如果某个特定的节点过载(相比其他节点而言)，请解释如何均衡负载。

c. 假设没有更新，那么在执行重新划分或者重新分配虚处理器的过程中是否需要中断查询处理？解释你的答案。

822

习题

18.9 对于轮转法、散列划分和范围划分这三种划分技术，请各给出一个查询实例，使得应用该划分技术能提供最快的响应。

18.10 当关系在它的一个属性上采用：

- 散列划分
- 范围划分

进行划分时，哪些因素会导致偏斜？对于上述两种情况，分别可以采取什么措施来减小偏斜？

18.11 请给出一个连接的实例，它不是简单的等值连接，但可以采用基于划分的并行。应该基于什么属性来进行划分？

18.12 对于下述每种运算，给出一种好的并行化方法。

a. 差运算。

b. 使用 **count** 运算的聚集。

c. 使用 **count distinct** 运算的聚集。

d. 使用 **avg** 运算的聚集。

e. 左外连接，其连接条件只涉及相等比较。

f. 左外连接，其连接条件涉及相等之外的比较。

g. 完全外连接，其连接条件涉及相等之外的比较。

18.13 请描述流水线并行的优点和缺点。

18.14 假设你希望使用无共享体系结构的并行系处理一个由大量小事务组成的工作负载。

a. 在这样的情形中需要查询内并行吗？如果不需要，为什么？哪种形式的并行适用呢？

b. 对于这样的工作负载，哪种形式的倾斜会很显著呢？

c. 假设大多数事务都会访问一条 account 记录，该记录包括一个 account_type 属性，以及一条对应的 account_type_master 记录，它提供了有关账户类型的信息。你会如何划分表（或）复制数据以加速事务的执行？你可以假定 account_type_master 关系极少会更新。

823

18.15 对关系进行划分所基于的属性可以对查询代价产生很大的影响。

a. 给定在单个关系上的一个 SQL 查询工作负载，什么属性会作为划分的候选属性？

b. 基于这样的工作负载，你会如何在不同的划分技术中进行选择？

c. 有可能在关系的不止一个属性上进行划分吗？请解释你的答案。

文献注解

在 20 世纪 70 年代末和 80 年代初，当关系模型取得了相当稳固的地位时，人们意识到关系运算是高度可并行的，而且具有很好的数据流特性。因而发起了包括 GAMMA（DeWitt 等[1990]）、XPRS（Stonebraker 等[1989]）和 Volcano（Graefe [1990]）在内的几个研究项目来研究关系运算并行执行的实用性。

Teradata 是最早的商用并行数据库系统之一，并继续占有着一个很大的市场份额。Red Brick 数据仓库是另一个早期的并行数据库系统，后来 Red Brick 被 Informix 收购，而 Informix 自己又被 IBM 收购。更新的并行数据库系统包括 Netezza、DATAllegro（现为微软的一部分）、Greenplum 和 Aster Data。

Joshi[1991]以及 Mohan 和 Narang[1992]讨论了并行数据库中的封锁问题。Bose 等[1989]、Mohan 和 Narang[1992]、Rahm[1993]讨论了并行数据库系统的高速缓存一致性协议。Carey 等[1991]讨论了客户－服务器系统中的高速缓存问题。

Graefe 和 McKenna[1993b]对于查询处理（包括查询的并行处理）给了一个很好的综述。交换算子模型是 Graefe [1990]以及 Graefe 和 McKenna[1993b]提出的。

DeWitt 等[1992]讨论了并行排序。Garcia 和 Korth[2005]以及 Chen 等[2007]讨论了基于多核和多线程处理器的并行排序。Nakayama 等[1984]、Richardson 等[1987]、Kitsuregawa 和 Ogawa[1990]、Wilschut 等[1995]，以及其他文献讨论了并行连接算法。

Walton 等[1991]、Wolf[1991]、Dewitt 等[1992]描述了并行连接中的倾斜处理。

Lu 等[1991]、Ganguly 等[1992]描述了并行查询优化技术。

自 2005 年以来，每年举办一次现代硬件上的数据管理（Data Management on Modern Hardware，DaMoN）

824 国际研讨会，其论文集讨论了适用于多核和多线程体系结构的数据库系统设计和查询处理算法。

第19章
Database System Concepts, 6E

分布式数据库

不同于并行系统中处理器是紧密耦合的并组成单个数据库系统，分布式数据库系统由松散耦合的站点组成，这些站点不共享物理部件。此外，运行在每个站点上的数据库系统可以有实质上的相互独立程度。第17章讨论了分布式系统的基本结构。

每个站点都可以参与到事务的执行中，这些事务所访问的数据可以位于一个站点上也可以位于几个站点上。集中式数据库系统和分布式数据库系统之间的主要差别在于，在前者中数据存放于单个地方，在而后者中数据存放于几个地方。数据的这种分布给事务处理和查询处理带来了很多困难。本章将论述这些困难。

首先在19.1节中，首先将分布式数据库分为同构或异构两类。然后19.2节论述如何在分布式数据库中存储数据的问题。19.3节概述分布式数据库系统中的事务处理模型。19.4节描述如何通过使用特殊的提交协议来实现分布式数据库中的原子事务。19.5节描述分布式数据库中的并发控制。19.6节概述如何通过复制来提供分布式数据库中的高可用性，使得即使出现故障，系统仍然可以继续处理事务。19.7节叙述分布式数据库中的查询处理。19.8节概述处理异构数据库的问题。19.10节描述目录系统，它可以视为一种特殊形式的分布式数据库。

在本章中，我们将用图19-1的银行数据库来解释我们的例子。

```
branch(branch_name, branch_city, assets)
account (account_number, branch_name, balance)
depositor (customer_name, account_number)
```

图19-1 银行数据库

19.1 同构和异构数据库

在**同构分布式数据库**(homogeneous distributed database)系统中，所有站点都使用相同的数据库管理系统软件，它们彼此了解，共同合作处理用户的请求。在这样的系统中，本地站点放弃了它们的部分自治性，以修改模式或者数据库管理系统软件这方面的权利的方式。为使事务处理能在多个站点间运行，数据库管理系统软件还必须和其他站点合作来交换与事务有关的信息。 [825]

相反，在**异构分布式数据库**(heterogeneous distributed database)中，不同站点可能使用不同的模式和不同的数据库管理系统软件。站点之间可能彼此并不了解，在合作处理事务的过程中，它们可能仅提供有限的功能。模式的差别经常是查询处理中的主要问题，而软件的差异成为处理访问多站点事务的一个障碍。

在本章中，我们关注同构的分布式数据库。不过，19.8节将简要讨论异构分布式数据库系统中的问题。

19.2 分布式数据存储

考虑一个要存储到数据库中的关系 r。在分布式数据库中存储这个关系有两种方法：

- **复制**(replication)。系统维护这个关系的几个相同的副本(拷贝)，并把每个副本存储在不同的站点上。复制的替代方式是只存储关系 r 的一份拷贝。

- **分片**(fragmentation)。系统把关系划分为几个片，并把每个片存储在不同的站点上。

分片和复制可以组合：一个关系可以划分为几个片，并且每个片可以有几个副本。下面几个节将详细阐述每种这样的技术。

19.2.1 数据复制

如果关系 r 被复制，则关系 r 的拷贝会存放在两个或多个站点上。在最极端的情况下，我们采用全**复制**(full replication)，这时拷贝会存放在系统中的每个站点上。

复制有很多优点和缺点：

826 - **可用性**(availability)。如果包含关系 r 的站点之一发生故障，那么关系 r 可以在另一个站点中找到。因此，即使一个站点发生了故障，系统仍可以继续处理涉及 r 的查询。
- **增加的并行度**(increased parallelism)。如果大多数对关系 r 的访问都只会导致对该关系的读取，那么几个站点可以并行地处理涉及 r 的查询。r 的副本越多，在事务执行的站点上发现所需数据的机会就越大。因此，数据复制将站点之间的数据移动减到了最小。
- **增加的更新开销**(increased overhead on update)。系统必须保证关系 r 的所有副本是一致的，否则可能产生错误的计算。因此，只要 r 更新了，更新就必须传播到包含其副本的所有站点。结果增加了开销。例如，在银行系统中，账户信息被复制到不同站点中，必须保证特定账户中的余额在所有站点上是一致的。

通常情况下，复制提高了 **read** 操作的性能，并增加了数据对只读事务的可用性。但是，更新事务的开销会增大。控制几个事务对复制数据的并发更新比我们在第 15 章学习的集中式系统中更为复杂。我们可以通过选择关系的副本之一作为 r 的**主拷贝**(primary copy)来简化关系 r 的副本管理。例如，在银行系统中，账户可以同与其开户站点关联起来。类似地，在机票预订系统中，航班可以与其起飞的站点关联起来。19.5 节将讨论用于分布式并发控制的主拷贝方案和其他可选方案。

19.2.2 数据分片

如果关系 r 是分片的，那么 r 划分为多个分片 r_1，r_2，…，r_n 中。这些分片包含足够的信息使得能够重构出原始关系 r。有两种不同的方案用于对关系分片：水平分片和垂直分片。水平分片通过将 r 的每个元组分给一个或多个分片来划分关系。垂直分片通过对关系 r 的模式 R 进行分解来划分关系。

在**水平分片**(horizontal fragmentation)中，关系 r 划分为多个子集 r_1，r_2，…，r_n。关系 r 的每个元组必须至少属于其中一个分片，以使得在需要时可以重构出原始关系。

举一个例子，*account* 关系可以划分为几个不同的分片，每个分片由属于特定支行的账户的元组构成。如果银行系统只有两个支行，即 Hillside 和 Valleyview，那么就有两个不同的分片：

$$account_1 = \sigma_{branch\_name = \text{"Hillside"}}(account)$$

$$account_2 = \sigma_{branch\_name = \text{"Valleyview"}}(account)$$

827 水平分片通常用来把元组保持在它们最常使用的站点上以最小化数据的传输。

一般而言，水平分片可以定义为整个关系 r 上的一个选择。也就是说，使用谓词 P_i 来构造分片 r_i：

$$r_i = \sigma_{P_i}(r)$$

通过将所有分片合并来重构关系 r，即：

$$r = r_1 \cup r_2 \cup \cdots \cup r_n$$

在该例子中，分片是不相交的。通过改变用于构造分片的选择谓词，可以让 r 的特定元组出现在不止一个 r_i 中。

垂直分片最简单的形式和分解(见第 8 章)是一样的。$r(R)$ 的**垂直分片**(vertical fragmentation)涉及定义模式 R 的几个属性子集 R_1，R_2，…，R_n，使得：

$$R = R_1 \cup R_2 \cup \cdots \cup R_n$$

r 的每个分片 r_i 定义为：

$$r_i = \Pi_{R_i}(r)$$

分片采用的方式应使得我们可以通过采用自然连接来从分片中重构出关系 r：

$$r = r_1 \bowtie r_2 \bowtie r_3 \bowtie \cdots \bowtie r_n$$

保证关系 r 能重构的一种方法是在每个 R_i 中包含 R 的主码属性。更一般地，可以使用任何超码。在模式 R 中加入一个称为 *tuple-id* 的特殊属性通常会比较方便。元组的 tuple-id 值是唯一的，可以将该元组与其他所有元组区别开来。这样 tuple-id 属性就作为扩展后模式的一个候选码，并包含到每个 R_i 中。元组的物理地址或逻辑地址可以用作 tuple-id，因为每个元组有一个唯一的地址。

为了说明垂直分片，考虑一个大学数据库，它包含关系 *employee_info* 来存储每位员工的 *employee_id*、*name*、*designation* 和 *salary*。出于保密的缘故，这个关系可能分为关系 *employee_private_info*（包含 *employee_id* 和 *salary*）以及另一个关系 *employee_public_info*（包含属性 *employee_id*、*name* 和 *designation*），可能同样出于安全考虑，它们可能存储在不同的站点。

两种类型的分片可以应用到单个模式上；例如，对关系水平划分后得到的分片可以进一步垂直划分。分片也可以复制。一般而言，一个分片可以复制，分片的副本可以进一步分片，依此类推。 828

19.2.3　透明性

分布式数据库系统的用户不要求知道数据的物理位置在哪里或者在特定本地站点上的数据应如何访问。该特点称为**数据透明性**（data transparency），它可以有几种形式：

- **分片透明性**（fragmentation transparency）。用户不要求知道关系是如何分片的。
- **复制透明性**（replication transparency）。在用户看来，每个数据对象逻辑上都是唯一的。分布式系统可能为了提高系统性能或者数据可用性而复制对象。用户不必关心什么数据对象被复制了，也不必关心副本存放在何处。
- **位置透明性**（location transparency）。用户无须知道数据的物理位置。只要用户事务提供数据标识符，分布式数据库系统应能够找到任何数据。

数据项（例如关系、分片和副本）必须有唯一的名字。在集中式数据库中这种特性容易保证。但是，在分布式数据库中我们必须小心，保证两个站点没有对不同的数据项使用相同的名字。

解决这个问题的一种方法是要求所有的名字都要在中央**名字服务器**（name server）中注册。名字服务器有助于确保同样的名字不会用于不同的数据项。我们也可以使用名字服务器根据给定项的名字来定位数据项。但是，这种方法受制于两个主要的缺点。首先，当数据项通过它们的名字来定位时，名字服务器可能成为性能瓶颈，导致性能低下。其次，如果名字服务器崩溃，分布式系统中的任何站点都不可能继续运行。

一种更为广泛使用的可选方法要求每个站点都将其自身的站点标识符作为前缀加到它所产生的任何名字前面。这种方法保证没有两个站点会产生相同的名字（因为每个站点都有唯一的标识符）。此外不需要中央控制。但是，这种方法无法实现位置透明性，因为站点标识符被附加到名字上了。这样，关系 *account* 就可能需要用 *site17.account* 或者 *account@site17* 来引用，而不是简单地用 *account* 来引用。许多数据库系统使用站点的互联网地址（IP 地址）来标识。

为了解决这个问题，数据库系统可以为数据项创建一个另外的名字或**别名**（alias）。这样用户就可以用简单的名字来引用数据项，而简单的名字被系统翻译为完整的名字。从别名到真实名字的映射可以存储在每个站点上。有了别名，用户就可以无须了解数据项的物理位置。此外，如果数据库管理员决定将数据项从一个站点移到另一个站点，用户也不会受到影响。

用户应该无须引用数据项的特定副本。相反，系统应该决定在 read 请求时引用哪个副本，在 write 请求时应该更新所有副本。通过维护目录表（系统用它来确定数据项的所有副本），我们可以保证完成上述工作。 829

19.3　分布式事务

在分布式系统中对各种数据项的访问常常通过事务来完成。事务必须保持 ACID 特性（见 14.1 节）。我们需要考虑两种类型的事务。**局部事务**（local transaction）是那些只在一个局部数据库中访问和更新数据的事务；**全局事务**（global transaction）是那些多个局部数据库中访问和更新数据的事务。局部

事务的 ACID 特性可以用在第 14 – 16 章中讨论的方式来保证。但是对于全局事务来说，这项任务就复杂得多了，因为多个站点可能参与到事务的执行中。这些站点中的一个发生故障，或者连接这些站点的通信链路发生故障，都可能导致错误的计算。

在本节中，我们学习分布式数据库的系统结构及其可能的故障模式。在 19.4 节中，我们学习为了保证全局事务原子性提交的协议。在 19.5 节中，我们学习分布式数据库中并发控制的协议。在 19.6 节中，我们学习分布式数据库在出现各种类型的故障的情况下如何继续工作。

19.3.1 系统结构

每个站点都有其自身的局部事务管理器，其功能是保证在该站点上执行的那些事务的 ACID 特性。各个事务管理器相互协作以执行全局事务。为了理解这样的管理器是怎样实现的，考虑一个事务系统的抽象模型，其中每个站点包括两个子系统：

- **事务管理器**(transaction manager)管理那些访问存储在一个局部站点中的数据的事务(或子事务)的执行。注意每个这样的事务既可以是局部事务(即只在该站点上执行的事务)，也可以是全局事务(即在几个站点上执行的事务)的一部分。
- **事务协调器**(transaction coordinator)协调在该站点上发起的各个事务(既有局部的也有全局的)的执行。

整个系统的体系结构如图 19-2 所示。

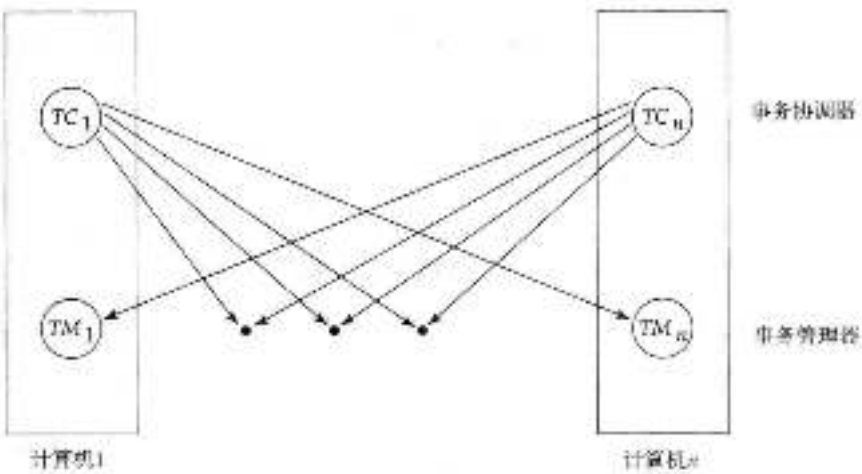


图 19-2 系统体系结构

事务管理器的结构在许多方面同集中式系统的结构是类似的。每个事务管理器都要负责：

- 维护一个用于恢复目的的日志。
- 参与到一个合适的并发控制方案，以协调在该站点上执行的事务的并发执行。

正如我们将要看到的，为了适应事务的分布性，我们需要修改恢复方案和并发方案。

在集中式环境中不需要事务协调器子系统，因为事务所访问的数据仅在单个站点上。因名思义，事务协调器负责协调该站点上发起的所有事务的执行。对每个这样的事务，协调器负责：

- 启动事务的执行。
- 将事务分成一些子事务，并将这些子事务分派给合适的站点去执行。
- 协调事务的中止，这可能导致事务在所有站点上都提交或在所有站点上都中止。

19.3.2 系统故障模式

830 ~ 831

分布式系统可能遭受和集中式系统所遭受的相同类型的故障(例如软件错误、硬件错误或磁盘崩溃)，但在分布式环境中还有另外一些需要处理的故障类型。基本的故障类型包括：

- 站点故障。
- 消息丢失。

- 通信链路故障。
- 网络划分。

分布式系统中总是可能发生消息的丢失或损坏。系统采用传输控制协议(如 TCP/IP)来处理这样的错误。关于这类协议的信息在有关网络的标准教科书中可以找到(见文献注解)。

然而，如果两个站点 A 和 B 不直接相连，消息必须通过一系列的通信链路从一个站点路由到另一个站点。如果有一条通信链路发生故障，通过该链路传送的消息必须重新路由。在某些情况下，可能发现网络中的另一条路由，这样消息仍能到达它们的目的地。在另一些情况下，故障可能导致在某些站点对之间没有了连接。如果一个系统分成了两个(或更多)子系统，称为**分区**(partition)，分区之间缺乏任何连接，该系统就是**被划分**(partitioned)的。注意在这种定义下，分区可能由单个站点构成。

19.4　提交协议

如果我们要保证原子性，执行事务 T 的所有站点就必须在执行的最终结果上达成一致。T 必须做到要么在所有站点上都提交，要么在所有站点上都中止。为了保证这个特性，T 的事务协调器必须执行一个提交协议。

两阶段提交(Two-Phase Commit，2PC)协议是最简单且使用最广泛的提交协议之一，将在 19.4.1 节介绍。另外一种是**三阶段提交**(Three-Phase Commit，3PC)协议，它避免了 2PC 协议的某些缺点但也增加了复杂性和开销。19.4.2 节将简要概述 3PC 协议。

19.4.1　两阶段提交

我们首先描述两阶段提交(2PC)协议在一般的操作中是如何进行的，然后描述它如何处理故障，并在最后描述它如何实现恢复和并发控制。

考虑一个从站点 S_i 发起的事务 T，设 S_i 的事务协调器是 C_i。

19.4.1.1　提交协议

当 T 完成其执行时(即执行 T 的所有站点都通知 C_i 已完成了 T 的执行)，C_i 启动 2PC 协议。

- **阶段 1**。C_i 将记录 **<prepare** T**>** 加到日志中，并强制日志写入稳定存储器上。接着它将一条 **prepare** T 消息发送到执行 T 的所有站点上。当收到这样一条消息时，站点上的事务管理器确定它是否愿意提交 T 中属于它的那部分。如果答案是"不"，事务管理器就把记录 **<no** T**>** 加到日志中，然后通过向 C_i 发送一条 **abort** T 消息来作为响应。如果答案为"是"，事务管理器就把记录 **<ready** T**>** 加到日志中，并将日志(包括所有与 T 相关的日志记录)强制写入稳定存储器上。然后事务管理器通过向 C_i 回复一条 **ready** T 消息作为回答。 832
- **阶段 2**。当 C_i 收到所有站点对 **prepare** T 消息的回答时，或者自 **prepare** T 消息发送后经过了一个预定的时间间隔时，C_i 就可以确定是将事务 T 提交还是中止。如果 C_i 接受到来自所有参与站点的 **ready** T 消息，那么事务 T 可以提交。否则，事务 T 必须中止。根据结论，要么将记录 **<commit** T**>** 要么将记录 **<abort** T**>** 加到日志中，并将日志强制写入稳定存储器上。这时，事务的最终结果就已经确定了。此后，协调器向所有参与站点发送消息 **commit** T 或 **abort** T。当站点收到此消息时，它就把此消息记录到日志中。

在向协调器发送 **ready** T 消息之前，执行 T 的站点可以在任何时候无条件地中止 T。一旦发出 **ready** T 消息，站点上的事务就称为处于**就绪状态**(ready state)。在效果上，**ready** T 消息是站点所做的承诺：即按照协调器的命令来提交 T 或中止 T。为了做出这样的承诺，所需信息必须首先存储在稳定存储器中。否则，如果站点在发送 **ready** T 后崩溃，它可能就不能兑现自己的承诺。总而言之，事务拥有的锁必须继续保留直到事务结束。

由于提交事务需要个体一致，因此一旦有至少一个站点回答 **abort** T，事务 T 的最终结果就决定了。因为作为协调器的站点 S_i 是执行 T 的站点之一，所以协调器可以单方面决定中止 T。关于 T 的最后结论是在协调器将此结论(提交或中止)写入日志并强制将其写入稳定存储器时确定的。在 2PC 协议的某些实现中，站点在协议第二阶段的最后向协调器发送 **acknowledge** T 消息。当协调器收到所有站

点发来的 **acknowledge** T 消息时，它就把记录 < complete T > 加到日志中。

19.4.1.2 故障处理

2PC 协议对于不同类型的故障有不同方式的反应：

- **参与站点故障**(failure of a participating site)。如果协调器 C_i 检测到某个站点发生了故障，它将采取如下行动：如果站点在用 **ready** T 消息回答 C_i 前发生故障，则协调器假定该站点是用 **abort** T 消息来回答的。如果站点在协调器接收到从该站点发来的 **ready** T 消息后发生故障，则协调器就按照通常的方式执行提交协议的剩余部分，忽略该站点的故障。

833

当参与站点 S_k 从故障中恢复时，它必须检查它的日志来决定故障发生时正在执行中的事务的最终结果。设 T 是一个这样的事务。我们考虑每种可能的情况：

  - 日志包含 < **commit** T > 记录。在这种情况下，该站点执行 **redo**(T)。
  - 日志包含 < **abort** T > 记录。在这种情况下，该站点执行 **undo**(T)。
  - 日志包含 < **ready** T > 记录。在这种情况下，该站点必须询问 C_i 以决定 T 的最终结果。如果 C_i 正在工作，它就通知 S_k 关于 T 是否提交或中止的信息。在前一种情况下，S_k 执行 **redo**(T)；在后一种情况下，S_k 执行 **undo**(T)。如果 C_i 出现故障，S_k 就必须试图从其他站点找到 T 的最终结果。它通过向系统中所有站点发送 **querystatus** T 消息来进行这一操作。站点收到这样一条消息，就必须查阅其日志来判定 T 是否在该站点上执行过，如果是，还要看 T 是否提交或中止。接着它把这样的结果告知 S_k。如果没有站点能提供给当的信息(即 T 是否提交或中止)，那么 S_k 既不能中止 T 也不能提交 T。关于 T 的决定被推迟到 S_k 能得到所需信息时为止。因此，S_k 必须定期地向其他站点重发 **querystatus** 消息。它不断这么做，直到包含所需信息的某个站点恢复为止。注意 C_i 所在站点总会有所需的信息。
  - 日志没有包含关于 T 的控制记录(**abort**、**commit**、**ready**)。因此，我们知道 S_k 在响应来自 C_i 的 **prepare** T 消息前发生了故障。由于 S_k 的故障排除了发送这样一个响应的可能性，根据我们的算法，C_i 必须中止 T。因此，S_k 必须执行 **undo**(T)。

- **协调器故障**(failure of the coordinator)。如果协调器在为事务 T 执行提交协议的过程中发生故障，那就必须由参与的那些站点来决定事务 T 的最终结果。我们将会看到，在特定情况下，参与站点不能决定是否提交或中止 T，因此这些站点必须等待发生故障的协调器恢复。
  - 如果活跃站点在其日志中包含 < **commit** T > 记录，则 T 必须提交。
  - 如果活跃站点在其日志中包含 < **abort** T > 记录，则 T 必须中止。
  - 如果某些活跃站点在其日志中没有包含 < **ready** T > 记录，则发生故障的协调器 C_i 不可能已经决定将 T 提交，因为在其日志中没有 < **ready** T > 记录的站点不会已经向 C_i 发送过 **ready** T 消息。但是，协调器可能已经决定中止 T 而不是提交 T。与等待 C_i 恢复相比，中止 T 更

834

为可取。

  - 如果上述情况均不成立，则所有活跃站点在它们的日志中都有 < **ready** T > 记录，但没有别的控制记录(如 < **abort** T > 或 < **commit** T >)。由于协调器已发生故障，因此不等到协调器恢复，就不可能确定是否已经做出决定，就算已做了决定也不知道做出的决定是什么。因此，活跃站点必须等待 C_i 的恢复。由于 T 的最终结果仍然是一个疑问，因此 T 可能继续占用系统资源。例如，如果使用封锁，T 可能拥有活跃站点的数据上的锁。这种情况不是所希望的，因为可能需要数小时或者数天之后 C_i 才能重新变得活跃。在这段时间内，其他事务可能被迫等待 T。其结果是，数据项不仅在发生故障的站点(C_i)上不可用，而且在活跃站点上也不可用。这种情况称为阻塞(blocking)问题，因为 T 在站点 C_i 恢复的过程中阻塞。

- **网络划分**(network partition)。当网络被划分时，存在两种可能性：
  - 协调器和它的所有参与者处于一个分区中。这种情况下，故障对提交协议没有影响。
  - 协调器和它的参与者属于几个分区。从其中一个分区的站点的角度来看，就好像其他分区中的站点发生了故障一样。不在协调器所位于的分区中的站点只需要执行协议来处理协调器的

故障。协调器以及与协调器在同一分区中的站点遵循平常的提交协议，假设其他分区中的站点发生了故障。

因此，2PC 协议的主要缺陷在于协调器故障可能导致阻塞，这种情况下必须等到 C_i 恢复才能做出提交或中止 T 的决定。

19.4.1.3　恢复与并发控制

当故障站点重新启动时，我们可以通过使用如 16.4 节描述的恢复算法来执行恢复。为了处理分布式提交协议，恢复过程必须特殊对待**疑问事务**(in-doubt transaction)；疑问事务是发现有 <**ready** T> 日志记录，但既未发现 <**commit** T> 日志记录，也未发现 <**abort** T> 日志记录的事务。如同 19.4.1.2 节描述的那样，恢复站点必须通过与其他站点的联系来确定这种事务的提交 - 中止状态。

但是，如果恢复像刚才所描述的那样执行，该站点上正常的事务处理就只有等所有疑问事务提交或回滚后才能开始。找出疑问事务的状态可能会很慢，因为可能不得不与多个站点进行联系。此外，如果协调器发生故障，并且到的任何站点都没有关于未完成事务的提交 - 中止状态信息，在使用 2PC 时就存在恢复被阻塞的潜在可能。其结果是，执行重启恢复的站点在很长一段时间内都保持不可使用状态。

为了防止这个问题，恢复算法通常提供对在日志中记载封锁信息的支持。(我们在此假设封锁用于并发控制。)该算法所写的日志记录不是 <**ready** T>，而是 <**ready** T, L> 日志记录，其中 L 是写入日志记录时事务 T 持有的所有写锁的列表。在恢复时，在执行局部恢复动作之后，对每个疑问事务 T 而言，<**ready** T, L> 日志记录(从日志中读到)中所记载的所有写锁都需要重新获取。 835

当所有疑问事务的封锁重新获取完成后，即使在疑问事务的提交 - 中止状态确定之前，该站点上的事务处理就可以开始了。疑问事务的提交或回滚与新事务的执行是并发进行的。这样，站点恢复就更快了，并且不再会阻塞。注意如果新事务持有的封锁与疑问事务持有的任何写锁冲突的话，那么新事务只有等与之冲突的疑问事务提交或回滚后才能继续运行。

19.4.2　三阶段提交

三阶段提交(3PC)协议是对两阶段提交协议的扩展，它在特定假设下避免了阻塞问题。特别地，它假设不发生网络划分，并且不超过 k 个站点发生故障，这里的 k 是某个预先定义的数值。在这些假设下，该协议通过引入一个额外的第三阶段来避免阻塞，在该阶段多个站点会涉及提交的决定。协调器不是在其持久存储中直接记录提交决定，而是首先保证至少有 k 个其他站点知道它打算提交事务。如果协调器出故障，剩下的站点首先选择一个新的协调器。这个新的协调器从剩下的站点中检查协议的状态；如果协调器已经决定提交，那么其他 k 个被通知的站点中至少有一个还在工作而且确保提交决定会考虑。如果一些站点知道旧的协调器打算提交事务，那么新的协调器重新开始协议的第三阶段。否则新的协调器中止该事务。

虽然 3PC 协议有令人满意的特性——它只在 k 个站点故障时才会阻塞，但是它有一个缺点是网络划分会与超过 k 个站点发生故障产生同样的效果。而这将导致阻塞。协议还必须小心地实现以保证网络划分(或者多于 k 个站点发生故障)不会导致不一致性，即在一个分区中事务提交，而在另一个分区中事务中止。由于其开销的原因，3PC 协议没有广泛使用。请参看文献注解中的文献来了解 3PC 协议的更多细节。

19.4.3　事务处理的可选择性模型

对于许多应用，两阶段提交的阻塞问题是不可接受的。这里的问题是单个事务在多个站点上工作的概念。这一节描述如何使用持久消息来避免分布式提交的问题，然后简单概述工作流的更大的问题；26.2 节将从更多细节上考虑工作流。

为了理解持久消息，考虑如何在两个不同的银行之间转账资金，其中每个银行都有它自己的计算机。一种方法是产生一个跨越两个站点的事务，并采用两阶段提交来保证原子性。然而，该事务可能需要更新银行的总余额，并且阻塞会对每个银行的所有其他事务产生严重影响，因为几乎银行中的所有事务都会更新银行的总余额。 836

相反，考虑如何通过银行支票来进行资金转账。银行首先从可用的余额里扣除支票金额，并把支票打印出来。然后，支票被物理地送到它需要存入的另一家银行。在审核支票之后，银行根据支票金额来增加本地的余额。支票就构成了两家银行之间的一次消息传递。为了使资金不消失也不错误地增长，支票必须不能丢失，复制或者存入多次。当银行的计算机通过网络相连时，持久消息就提供和支票同样的服务（当然要快得多）。

持久消息（persistent message）是这样的消息：如果发送该消息的事务提交，则不管是否发生故障，它都保证传送给接受者恰好一次（既不多也不少）；如果该事务中止则持久消息保证不传送。我们马上将会看到，在一般网络通道的上面，采用数据库恢复技术来实现持久消息。相反，通常的消息在某些情况下可能会丢失或者甚至传送多次。

持久消息的错误处理比两阶段提交更复杂。例如，如果支票要存入的账户已经关闭，则支票必须被送回并且退还到原来的账户。因此两个站点都必须有错误处理代码，以及处理持久消息的代码。相反，使用两阶段提交，错误可以被事务检测到，该事务就不会扣除第一家银行的资金。

可能出现的异常情况类型取决于应用，因此数据库系统不可能自动处理异常。发送和接受持久消息的应用程序必须包含处理异常情况的代码，并且可以使系统恢复到一致性状态。例如，如果接受资金的账户已经关闭，则丢失正在转账的资金是不可接受的；资金必须归还给原来的账户，并且如果由于某些原因不能这样处理，必须发出警告来人工解决这种问题。

在许多应用中消除阻塞的好处和实现使用持久消息的系统所需付出的额外代价相比是值得的。事实上，由于故障会导致本地数据访问的阻塞，因此很少有组织机构会同意支持在该组织机构之外发起的事务的两阶段提交。因此持久消息在执行跨越组织机构边界的事务中扮演了重要角色。

工作流提供了一种通用的事务处理模型，涉及多个站点并可能需要人工处理特定的步骤。例如，
837 当银行接收到一个借贷申请时，在批准或者拒绝这个借贷申请之前，它必须经过很多步骤，包括联系外部信用检查代理。所有步骤一起形成了一个工作流。我们将在26.2节学习工作流的更多细节。我们还注意到持久消息在分布式环境中形成了工作流的基础。

我们现在考虑持久消息的实现（implementation）。持久消息可以通过下面这些协议在不可靠的消息传递基础架构之上实现，这种基础架构可能丢失消息或多次传送消息。

- **发送站点协议**（sending site protocol）：当事务希望发送持久消息时，它在专用关系 *messages_to_send* 中写一条包含消息的记录，而不是直接向外发送消息。这个消息也被赋予一个唯一的消息标识符。

  消息传送进程监控该关系，当发现一条新消息时，它将消息发送到其目的地。通常的数据库并发控制机制保证系统进程只在写消息的事务提交之后才能读取消息，如果该事务中止，通常的恢复机制将从该关系中删除消息。

  消息传送进程在它收到目的站点的确认之后才从该关系中删除消息，如果它没有收到来自目的站点的确认，在一段时间之后它将重发消息。它如此反复直到收到确认为止。万一发生永久故障，系统将在一段时间之后决定消息无法传送。然后调用由应用程序提供的异常处理代码来处理故障。

  在事务提交之后才将消息写到关系并处理它，保证了当且仅当事务提交之后消息才会传送。重复发送消息保证了即使在（临时的）系统或者网络故障的时候，消息也会传送。
- **接收站点协议**（receiving site protocol）：当站点接收到持久消息时，它运行一个事务将消息加入到专用关系 *received_messages* 中，前提是该消息没有出现在该关系中（唯一的消息标识符允许检测出重复）。在事务提交之后，或者消息已经出现在该关系中时，接收站点向发送站点发回一个确认。

注意到在事务提交前发送确认是不安全的，因为随后发生的系统故障可能导致消息的丢失。有必要检查消息是否早先已经接收过以避免消息的多次传送。

在许多消息系统中，尽管随机的消息延迟很少发生，但是有可能发生。因此，为了安全，消息必须永不从关系 *received_messages* 中删除。删除消息会导致无法检测到重复传送。但是这样做的结果会导

致关系 *received_messages* 的无限增长。为了处理这个问题，每条消息被赋予一个时间戳，并且如果接收到的消息的时间戳比某个截止期老，则该消息被丢弃。在关系 *received_messages* 中记录的比截止期老的所有消息都可以删除。 838

19.5　分布式数据库中的并发控制

这里说明第 15 章所讨论的某些并发控制模式该做何种修改以使之适用于分布式环境。我们假设每个站点都参与到提交协议的执行中，以保证全局事务的原子性。

本节描述的协议需要对数据项的所有副本进行更新。如果任何一个包含数据项副本的站点发生故障，那么对该数据项的更新就不能进行。19.6 节描述即使在一些站点或者链接发生故障时仍能继续事务处理，从而提供高可用性的一些协议。

19.5.1　封锁协议

第 15 章描述的各种封锁协议都可用于分布式环境中。需要做的唯一改变是锁管理器处理复制数据的方式。我们给出适用于数据能在几个站点上复制的环境下的几种可能的模式。和第 15 章一样，我们将假设存在共享和排他封锁模式。

19.5.1.1　单一锁管理器方式

在**单一锁管理器**(single lock-manager)方式下，系统维护位于单个选定站点(如 S_i)中的单一锁管理器。所有封锁和解锁请求都在站点 S_i 上处理。当事务需要给数据项加锁时，它就向 S_i 发封锁请求。锁管理器决定锁是否能够立即授予。如果锁能够授予，锁管理器就给发起封锁请求的站点发一条告知此结果的消息。否则，请求就推迟直到锁能授予为止，这时才能发送消息给发起封锁请求的站点。事务可以从该数据项副本所在站点中的任何一个来读取该数据项。在写情况下，所有存在该数据项副本的站点都必须参与到写操作中。

这个模式有这些优点：

- **实现简单**(simple implementation)。该模式处理封锁请求需要两条消息，处理解锁请求需要一条消息。
- **死锁处理简单**(simple deadlock handling)。由于所有封锁和解锁请求在一个站点上处理，因此可以直接运用第 15 章讨论的死锁处理算法。

这个模式的缺点是：

- **瓶颈**(bottleneck)。站点 S_i 成为瓶颈，因为所有请求都必须在这里处理。
- **脆弱性**(vulnerability)。如果 S_i 发生故障，就会丢失并发控制器。要么必须停止处理过程，要么必须像 19.6.5 节描述的那样使用恢复方案来使一个后备站点从 S_i 那里接手封锁管理。 839

19.5.1.2　分布式锁管理器

通过**分布式锁管理器**(distributed lock-manager)方法可以达到优点和缺点之间的一种折中方案。在该方案中封锁管理器功能分布在多个站点上。

每个站点维护一个本地锁管理器，它的功能是管理存储在该站点上的那些数据项的封锁和解锁请求。当事务想要封锁数据项 Q(Q 未复制，且存储在站点 S_i 上)时，就发送一条消息到站点 S_i 上的锁管理器来请求一个锁(以特定的锁模式)。如果数据项 Q 以不相容的模式封锁，那么请求就会延迟到锁可以授予为止。一旦锁管理器决定允许封锁请求，就向发起者发消息来表明其封锁请求已经许可。

19.5.1.3 – 19.5.1.6 节将讨论处理数据项复制的几种可选方法。

分布式锁管理器模式具有实现简单的优点，而且减轻了协调者成为瓶颈的程度。它有合理的低开销，处理封锁请求需要两次消息传输，并且处理解锁请求只需一次消息传输。但是，由于封锁和解锁请求不再在单个站点上处理，因此死锁处理更加复杂：即使在单个站点内没有死锁，仍可能会在站点之间发生死锁。为了检测全局的死锁，第 15 章讨论的死锁处理算法必须修改，对此我们将在 19.5.4 节讨论。

19.5.1.3 主副本

在系统使用数据复制时，我们可以选择一个副本作为**主副本**(primary copy)。对每个数据项 Q 而言，Q 的主副本必然正好位于一个站点上，我们称其为 Q 的**主站点**(primary site)。

当事务需要封锁数据项 Q 时，它在 Q 的主站点上请求封锁。和前面一样，对该请求的响应会推迟到该请求能满足为止。主副本使得对已复制数据的并发控制可以像处理未复制的数据一样。这种相似性简化了实现。然而，如果 Q 的主站点发生故障，即使包含副本的其他站点是可以访问的，Q 也不能被访问了。

19.5.1.4 多数协议

多数协议(majority protocol)以这种方式工作：如果数据项 Q 在 n 个不同的站点复制，则封锁请求消息必须发送到存储 Q 的 n 个站点中一半以上的站点。每个锁管理器(只要是牵涉的)确定锁是否能立即授予，和前面一样，响应推迟到该请求能满足时为止。事务只有在成功获得 Q 的多数副本上的锁之后才能开始 Q 上的操作。

[840] 我们暂时假定在所有副本上执行写操作时，要求包含副本的所有站点都是可用的。但是，多数协议的主要优点是它能扩展来处理站点故障。正如我们将在 19.6.1 节中看到的那样，此协议还以分散的方式来处理复制的数据，因此避免了集中控制的缺点。但是，它有这些缺点：

- **实现**(implementation)。多数协议比前面的方案在实现上更为复杂。它至少需要 $2(n/2+1)$ 条消息来处理封锁请求，至少需要 $(n/2+1)$ 条消息来处理解锁请求。
- **死锁处理**(deadlock handling)。除了因为使用分布式锁管理器方式而产生的全局死锁问题，即使只有一个数据项被封锁也可能发生死锁。作为例子，考虑看一个有 4 个站点和完全复制的系统。假设事务 T_1 和 T_2 希望以排他模式封锁数据项 Q。事务 T_1 可能在站点 S_1 和 S_3 上成功封锁 Q，而事务 T_2 可能在站点 S_2 和 S_4 上成功封锁 Q。然后两个事务必须等待获得第二个锁；于是就发生了死锁。幸运的是，通过要求所有站点以相同的预定顺序来请求数据项副本上的封锁，我们可以比较容易地避免这样的死锁。

19.5.1.5 有偏协议

有偏协议(biased protocol)是另一种处理复制的方法。与多数协议的区别是对共享锁请求的待遇比排他锁请求要优越一些。

- **共享锁**(shared lock)。当事务需要封锁数据项 Q 时，它只需要向包含 Q 的副本的一个站点上的锁管理器请求 Q 上的锁。
- **排他锁**(exclusive lock)。当事务需要封锁数据项 Q 时，它需要向包含 Q 的副本的所有站点上的锁管理器请求 Q 上的锁。

和前面一样，对请求的响应推迟到该请求能满足为止。

有偏模式具有附加到读(read)操作上的开销比多数协议要小的优点。通常情况下读(read)频率比写(write)频率要高得多。这种开销的节省就尤为明显。但是，写上的额外开销是此模式的一个缺点。此外，有偏协议和多数协议一样具有死锁处理复杂的缺点。

19.5.1.6 法定人数同意协议

法定人数同意(quorum consensus)协议是多数协议的一种推化。法定人数同意协议给每个站点分配一个非负的权重。它为数据项 x 上的读和写操作分配两个整数，称作**读法定人数**(read quorum)Q_r 和写

[841] **法定人数**(write quorum)Q_w。它们必须满足下面的条件，其中 S 是 x 所有的所有站点的总权重：

$$Q_r + Q_w > S \text{ and } 2Q_w > S$$

为了执行读操作，必须封锁足够的副本，使它们的总权重至少是 r。为了执行写操作，必须封锁足够的副本，使它们的总权重至少是 w。

法定人数同意方法的好处是：通过合适地定义读和写法定人数，它能够选择性地降低读或者写封锁的代价。例如，如果读法定人数很小，读操作只须获得较少的锁，但是写法定人数就会比较高，因此写操作需要获得更多的锁。当然，如果某些站点被分配了较大的权重(例如，那些很少可能发生故障的站点)，那么在封锁请求时需要访问的站点会更少一些。事实上，通过合适地设置权重和法定人数，

法定人数同意协议可以模拟多数协议和有偏协议。

和多数协议一样，法定人数同意协议能扩展为甚至在站点故障情况下工作，正如我们将在19.6.1节中看到的那样。

19.5.2 时间戳

15.4节中的时间戳模式后面的基本思想是：每个事务被给予一个唯一的时间戳，系统用时间戳来决定串行化顺序。因而，在将集中式模式推广到分布式模式时，我们的首要任务就是设计一种产生唯一时间戳的方案。这样，前面的各种协议就可以直接运用于无复制的环境。

产生唯一时间戳的方法主要有两种，一种是集中式的，而另一种是分布式的。在集中式模式中，由单个站点来分发时间戳。这个站点可以用逻辑计数器或其自身的本地时钟来达到此目的。

在分布式模式中，每个站点使用逻辑计数器或本地时钟来产生唯一的局部时间戳。通过将唯一的局部时间戳和站点标识符（也必须是唯一性的）串接起来，我们得到了唯一的全局时间戳（见图19-3）。串接的顺序是很重要的！我们把站点标识符放在重要性低的位置，以保证一个站点上产生的全局时间戳不会总是比另一个站点上产生的全局时间戳大。请比较这种产生唯一时间戳的技术和19.2.3节介绍的产生唯一名称的技术。

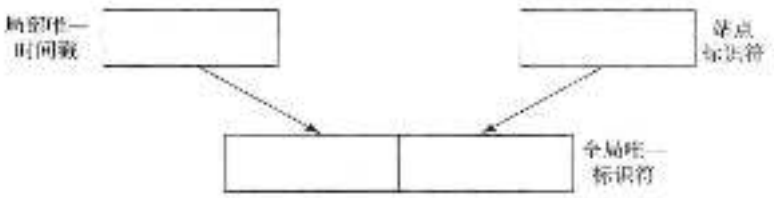


图19-3 唯一时间戳的产生

842

如果一个站点产生局部时间戳的速率比其他站点高，我们仍然会遇到问题。在这种情况下，快站点的逻辑计数器会比其他站点的大。因此，快站点产生的所有时间戳会比其他站点产生的时间戳大。我们需要一种机制来保证整个系统中局部时间戳能公平地产生。我们在每个站点 S_i 中定义一个**逻辑时钟**（logical clock）（LC_i），后者产生唯一的局部时间戳。逻辑时钟可以用计数器来实现。每产生一个新的局部时间戳计数器就递增。为了保证不同的逻辑时钟是同步的，我们要求无论什么时候，只要具有时间戳 $<x, y>$ 的事务 T_i 访问站点 S_j 并且 x 大于 LC_j 的当前值，S_j 就增大其逻辑时钟。在这种情况下，站点 S_j 将其逻辑时钟增大到值 $x+1$。

如果用系统时钟来产生时间戳，只要任意站点的系统时钟都不会运行过快或过慢，那么时间戳的分配就是公平的。由于时钟可能不是完全精确的，因此必须采用和用于逻辑时钟类似的技术来保证没有时钟比另一个时钟更快或更慢。

19.5.3 弱一致性级别的复制

现在许多商用数据库都支持某种形式的复制。在**主从复制**（master-slave replication）的情况下，数据库允许在主站点上更新，并自动将更新传播到其他站点上的副本。事务可以在其他站点上读取副本，但是不允许对它们更新。

这种复制的一个重要特性是事务在远程站点上没有得到锁。为了保证运行在副本站点上的事务要到数据库的一致性（但可能是过期的）视图，副本应该反映十站点上数据的**事务一致快照**（transaction-consistent snapshot）；即，副本应该反映在串行化顺序中先于某个事务的那些事务所做的所有更新，而不应该反映在串行化顺序中后于该事务的那些事务所做的任何更新。

数据库可能配置成在主站点发生更新之后马上传播更新，或者只是周期性地传播更新。

主从复制对于分发信息特别有用。例如从一个组织机构的中央办公室到分支办公室。这种复制形式的另一个应用是创建数据库的副本以运行大的查询，这样查询就不会与事务交互。更新应该周期性地传播（例如，每个晚上），这样更新传播就不会影响到查询处理。

Oracle数据库系统支持**快照创建**（create snapshot）语句，该语句可以创建远程站点上一个关系或一

组关系的事务一致性快照副本。它也支持快照刷新，可以通过重新计算或增量更新快照来实现。Oracle
843 支持自动刷新，刷新可以是连续的也可以是按周期性时间间隔的。

在**多主副本**(multimaster replication)(也称为**可在任何地方更新的复制**(update-anywhere replication))情况下，允许在数据项的任何副本上更新，并且自动传播到所有副本。这种模型是用于管理分布式数据库副本的基本模型。事务更新本地的副本，并由系统透明地更新其他副本。

更新副本的一种方法是使用我们见过的一种分布式并发控制技术，以两阶段提交来实现立即更新。许多数据库系统使用有偿协议来作为它们的并发控制技术，其中写操作必须封锁和更新所有副本，且读操作仅封锁和读取任意副本。

许多数据库系统提供另一种形式的更新：它们在一个站点上更新，采用**延迟传播**(lazy propagation)将更新传播到其他站点，而不是把更新立即传播到所有副本的操作作为事务执行更新的一部分。即使在一个站点与网络断连时，基于延迟传播的方案仍允许进行事务处理(包括更新)，因而提高了可用性。但遗憾的是，这样做是以牺牲一致性为代价的。在使用延迟传播时，以下两种方法之一通常被采用：

- 在副本上的更新被转化为在主站点上的更新，然后延迟传播到所有副本。这种方法保证对数据项的更新以串行化顺序进行，但是可能发生可串行化问题，因为事务可能读取某些其他数据项的旧值并用它来执行更新。
- 更新在任何副本上执行，并传播到所有其他副本。由于同一个数据项可能在多个站点并发地更新，因此这种方法可能导致更多的问题。

当更新传播到其他站点(我们将在25.5.4节中看到如何操作)时，由于缺少分布式的并发控制，可能会检测到一些冲突，但是解决冲突涉及已提交事务的回滚，这样就无法保证已提交事务的持久性。并且，可能需要人为干涉来处理冲突。因此上面的方案应该避免或者小心使用。

19.5.4 死锁处理

第15章中的死锁预防和死锁检测算法只要做一些修改就可以用于分布式系统中。例如，我们可以通过在系统数据项之间定义一棵全局树就可以使用树协议。类似地，正如我们在19.5.2节中看到的那样，时间戳排序方法可以直接用于分布式环境。

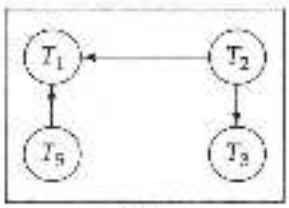
死锁预防可能导致不必要的等待和回滚。此外，特定的死锁预防技术与不采用这些技术的情况相比，可能需要更多站点参与到事务的执行中。

844 如果我们允许发生死锁并依赖于死锁检测，那么分布式系统中的主要问题就是决定如何维护等待图。处理这个问题的常用技术要求每个站点维护一个**局部等待图**(local wait-for graph)。图中的节点对应于目前占有或请求该站点上任何局部数据项的所有事务(局部的和非局部的)。例如，图19-4描述了一个包含两个站点的系统，每个站点维护自己的局部等待图。注意事务 T_1 和 T_2 在两个图中都出现。这表明它们在两个站点上都有数据项请求。

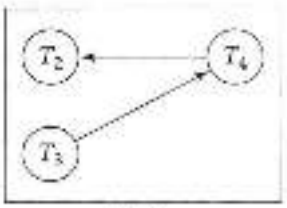
这些局部等待图以一种通常的方式来构造，用于表达局部事务和数据项。当站点 S_i 上的事务 T_j 需要站点 S_k 上的资源时，它就向站点 S_k 发请求消息。如果该资源被事务 T_k 占用，系统就把一条 $T_j \rightarrow T_k$ 的边插入到站点 S_k 的局部等待图中。

显然，如果任意的局部等待图中存在环，就已经发生了死锁。另一方面，任意的局部等待图中都不存在环却并不意味着没有死锁。为了说明这个问题，我们考虑图19-4中的局部等待图。每个等待图都是无环的；然而，系统中却存在死锁，因为局部等待图的并包含一个环。这个例在图19-5中给出。

在**集中式死锁检测**(centralized deadlock detection)方法中，系统在单个站点中构造和维护一个**全局等待图**(global wait-for graph)(所有局部图的并)；该站点是死锁检测的协调器。由于系统中存在通信延迟，我们必须区分两类等待图。真实图描述了系统在任意时刻真实的但不为人知的状态，就像无所不知的观察者所能看到的那样。构造图是控制器在执行控制器算法中产生的一种近似。显然，控制器必须以下述方式产生构造图：即无论何时调用检测算法，报告的结果都是正确的。这种情况下的正确是指：如果存在死锁，会迅速报告它；而如果系统报告了死锁，它就确实处于死锁状态。



站点 S_1



站点 S_2

图 19-4　局部等待图

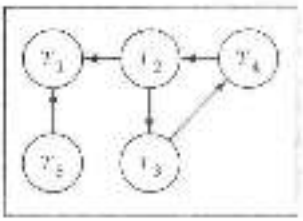


图 19-5　图 19-4 的全局等待图

全局等待图可以在这些情况下重构或更新：

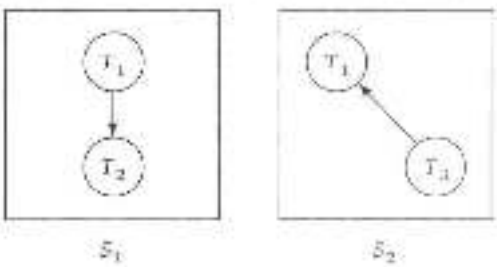
- 每当一条新的边插入到局部等待图或从中删除一条边时。
- 周期性地，当局部等待图中发生大量改变时。
- 每当协调器需要调用环检测算法时。

在协调器调用死锁检测算法的时候，协调器搜索它的全局图。如果它发现环，就选出环中的一个牺牲者去回滚。协调器必须告知所有站点特定事务被选作牺牲者。于是这些站点将该牺牲事务回滚。

在下述情况下，该方案可能产生不必要的回滚：

- 在全局等待图中存在假环(false cycle)。作为例子，考察图 19-6 中的局部等待图所代表的系统快照。假设 T_2 释放它在站点 S_1 中所占用的资源，导致从 S_1 中删除边 $T_2 \rightarrow T_3$。接着事务 T_2 请求站点 S_2 上被 T_3 所占用的资源，导致在 S_2 中增加边 $T_2 \rightarrow T_3$。如果来自 S_2 的消息 **insert** $T_2 \rightarrow T_3$ 早于来自 S_1 的消息 **remove** $T_2 \rightarrow T_3$，则协调器在 **insert** 后(但在 **remove** 前)可能会发现假环 $T_2 \rightarrow T_3 \rightarrow T_2$。死锁恢复可能启动，尽管并没有发生死锁。

  注意假环现象在两阶段封锁中是不会发生的。假环的可能性通常是很小，不会导致严重的性能问题。



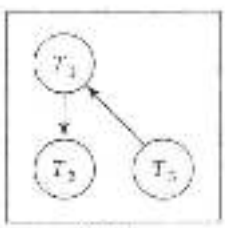


图 19-6　全局等待图中的假环

- 当死锁确已发生且牺牲事务已经选定，但其中一个事务由于与该死锁无关的原因而中止。例如，假设图 19-4 中站点 S_1 决定中止 T_2。同时，协调器发现了环并选定 T_3 作为牺牲者。此时 T_2 和 T_3 都回滚，尽管只有 T_2 需要回滚。

死锁检测可以分布式方式进行，由几个站点来承担任务的几个部分，而不是在单个站点上运行。但是，这样的算法会更加复杂和更加昂贵。关于这类算法请参考文献注解中的文献。

19.6　可用性

使用分布式数据库的一个目的是**高可用性**(high availability)：即，分布式数据库必须在几乎所有时候都能工作。特别地，因为在大型分布式系统中发生故障的可能性更大，即使在各种类型的故障发生的时候，分布式数据库也必须能继续工作。在故障期间仍能继续工作的能力称为**健壮性**(robustness)。

为了使分布式系统是健壮的，它必须检测故障，能重新配置系统以便继续计算，并在处理器或链路修复后能进行恢复。

不同类型的故障以不同方式处理。例如，消息丢失通过重传来处理。对通过某链路的一条消息进行重复的重传，却接收不到确认，这常常是链路故障的征兆。网络通常会试图为此消息寻找替换路径。无法找到这样的路径常常是网络划分的征兆。

但是，明确区分站点故障和网络划分一般是不可能的。系统通常可以检测到故障的发生，但它有可能不能识别出故障类型。例如，假设站点 S_1 不能同 S_2 通信。可能是 S_2 发生了故障，但另一种可能是由于 S_1 和 S_2 之间的链路故障而导致了网络划分。通过在站点间使用多条链路可以部分解决这个问题，这样即使一条链路发生故障站点仍能保持连接。然而，多条链路故障仍可能发生，因此在有些情况下我们无法确定是发生了站点故障还是网络划分。

845
847

假设站点 S_1 已经发现有故障发生。这时 S_1 必须发起一个过程，此过程使得系统可以重新配置并继续通常的操作模式。

- 如果在发生故障时，事务在故障或者无法访问的站点上是活跃的，则这些事务就应该中止。最好是立即中止这样的事务，因为它们可能持有在仍然活跃的站点的数据上的锁；等待发生故障或无法访问的站点变得重新可访问可能会阻碍可操作站点上的其他事务。然而，在某些情况下，当数据对象被复制后，即使有些副本不能访问，它仍然可能继续进行读和更新。在这种情况下，当故障站点恢复的时候，如果它有任何数据对象的副本，它必须得到这些数据对象的当前值，而且必须确保它能接收到所有未来的更新。我们将在 19.6.1 节阐述这个问题。
- 如果被复制的数据存储在故障或者无法访问的站点上，就应重新目录，使得查询不再引用该故障站点上的拷贝。当站点重新连接时，必须注意确保该站点上的数据的一致性，正如我们将在 19.6.3 节中看到的那样。
- 如果故障站点是某些子系统的中央服务器，就必须进行选举来决定新的服务器（见 19.6.5 节）。中央服务器的例子包括名字服务器、并发协同器或者全局死锁检测器。

由于通常情况下，不可能区分网络链路故障和站点故障，因此任何重新配置方案都必须设计成能够在网络划分的情况下正确运行。特别地，为确保一致性必须避免这些情况：

- 两个或更多的中央服务器在不同的分区中选出。
- 不止一个分区更新某个复制的数据项。

虽然传统数据库系统非常重视一致性，但是目前有很多应用比一致性更加看重可用性。为这样的系统设计的复制协议是不同的，这点将在 19.6.6 节讨论。

19.6.1 基于多数的方法

可以对 19.5.1.4 节中基于多数方法的分布式并发控制进行修改使其能在发生故障时仍能工作。在这种方法中，每个数据对象都存储它的一个版本号来检测最后一次写入它的时间。每当事务写对象的时候，它还要以如下方式来更新版本号：

848

- 如果数据对象 a 被复制到 n 个不同的站点，那么一条锁请求消息必须被发送到存储 a 的 n 个站点中的一半以上。事务直到成功获得了 a 的大多数副本上的锁之后才对 a 进行操作。
- 读操作检查所有已经加锁的副本，并从具有最高版本号的副本读取值。（它们还可以选择把该值写回到低版本号的副本。）写操作和读操作一样，也要读所有的副本来找到最高版本号（这一步通常通过读操作在事务中早已完成，并且结果可以重用）。新版本号是一个比最高版本号还要高的版本号。写操作写它已获得封锁的所有副本，并将所有副本的版本号设置为新的版本号。

在事务处理中的故障（无论是网络划分还是站点故障）是可以容忍的，只要：（1）提交时可用的站点包含被写的所有对象的大多数副本，同时（2）在读取过程中，需要读大多数副本来找到版本号。如果违反了这些要求，事务必须中止。只要满足这些要求，两阶段提交协议就可以像通常那样在可用的站点上使用。

在这种模式中，重建就很不足道了：什么都不需要做。这是因为写已经更新了大多数副本，而读会读取大多数副本并找到至少一个有最新版本号的副本。

用于多数协议的版本编号技术也可以用于使法定人数同意协议工作在出现故障的情况下。我们把（简单的）细节留给读者。然而，如果某些站点被赋予较高的权重，则防止系统处理事务的故障的危险性也会增加。

19.6.2 读一个、写所有可用的方法

作为法定人数同意的特殊例子，我们可以将单位权重授予所有站点来实施有偏协议：设置读法定人数为1，并设置写法定人数为n(所有站点)。在这种特殊情况下，不需要使用版本号；但是，即使只要容纳数据项的一个站点发生故障，对该数据项的写操作就不能进行，这是由于写法定人数不够。因为必须写所有的副本，所以这种协议被称为读一个、写所有(read one, write all)协议。

为了使发生故障时工作能够继续进行，我们倾向于使用读一个、写所有可用(read one, write all available)协议。在这种方法中，读操作与读一个、写所有(read one, write all)模式一样处理：任何可用副本都可以读取，并在那个副本上获得读锁。写操作转移到所有副本上；并需要获得所有副本上的写锁。如果一个站点宕机，事务管理器不等待该站点恢复而继续进行。

尽管这种方法看起来非常吸引人，但它有几个复杂的因素。特别地，暂时的通信故障会可能使得站点表现为不可用，导致写不能执行，但是当链路恢复时，该站点不会意识到它必须执行一些重建动作来弥补它已经丢失的写。另外，如果发生网络划分，由于认为在其他划分中的站点都已经发生故障死机，每个划分可能对相同的数据项进行更新。 849

在绝不会发生网络划分的情况下，可以使用读一个、写所有可用的模式，但是在发生网络划分的情况下，它会导致不一致性。

19.6.3 站点重建

将修复的站点或链路重建到系统中要小心。当故障站点恢复时，它必须发起一个过程来更新其系统表，使之反映在该站点宕机时所发生的变化。如果该站点有任何数据项的副本，它必须获得这些数据项的当前值，并保证它能接收到以后的所有更新。站点重建绝不像第一眼看起来那么简单，因为在该站点处于恢复中时可能存在对数据项的更新。

一种容易的解决办法是当故障站点重新连入时暂时停止整个系统。但是，在大多数应用中，这种暂时的停止可能会引起无法接受的破坏。已经开发出一些技术允许故障站点在并发执行对数据项的并发更新的同时进行重建。在对任意数据项授予读或写锁之前，站点必须确保它已经更新成为最新的数据项。如果一条故障链路恢复，两个或更多分区可以重新连接起来。由于网络划分限制了某些或所有站点可允许的操作，应该将该链路的恢复迅速通知给所有站点。有关分布式系统中恢复的更多信息请参见文献注解。

19.6.4 与远程备份的比较

我们在16.9节中学习的远程备份系统和分布式数据库中的复制是提供高可用性的两种可供选择的方法。这两种方案的主要区别是：使用远程备份系统时，诸如并发控制和恢复那样的操作在单个站点上执行，并且只有数据和日志记录被复制到其他站点。特别地，远程备份系统有助于避免两阶段提交及其导致的开销。而且，事务只须和一个站点(主站点)联系，并因此避免了在多个站点运行事务代码的开销。因此远程备份系统提供了一种比复制开销低的高可用性方法。

另一方面，通过使用多个可用副本并使用多数协议，复制可以提供更高的可用性。

19.6.5 协调器的选择

我们给出的几种算法需要用到协调器。如果协调器因为它所位于的站点发生故障而失效，系统只有在另一站点重新启动一个新协调器才能继续执行。能够继续执行的一种方法是通过维护协调器的一个备份，准备在协调器失效时承担其职责。 850

备份协调器(backup coordinator)是这样的一个站点，除了其他任务以外，它在本地维护足够的信息，使它可以在给分布式系统带来最小限度干扰的情况下担当起协调器的角色。所有发给协调器的消息会被协调器及其备份同时接收。备份协调器同真实协调器一样，执行相同的算法并维护相同的内部状态信息(例如，对并发协调器来说是锁表)。协调器和它的备份之间在功能上的唯一区别在于，备份不会采取任何影响到其他站点的动作。这些动作留给真实协调器来执行。

在备份协调器检测到真实协调器发生故障的情况下，它就担当起协调器的角色。由于备份协调器拥有故障协调器具有的所有可用信息，因此无须中断就可以继续处理。

备份方式的主要优点是能立即对处理继续下去的能力。如果备份没有准备好来承担协调器的职责，新指定的协调器为了执行协调任务，就必须从系统中的所有站点寻找信息。故障协调器常常是某些必需信息的唯一来源。在这种情况下，可能有必要中止几个（或所有）活跃事务，并在新协调器的控制下重新启动它们。

因此，备份协调器方式避免了分布式系统从协调器故障恢复所需的大量延迟。其缺点在于协调器任务重复执行的开销。此外，协调器及其备份需要定期通信，以保证它们的活动是同步的。

简言之，备份协调器方式引入正常处理时的开销来使得系统可以从协调器故障中迅速恢复。

在缺少指定的备份协调器或者为了处理多种故障的情况下，可以由活跃站点来动态地选出新的协调器。**选举算法**（election algorithm）使各站点以分散的方式选出一个站点作为新的协调器。选举算法需要系统中每个活跃站点都关联一个唯一的标识号。

用于选举的**威逼算法**（bully algorithm）的工作过程如下：为了保持符号和讨论的简单性，假设站点 S_i 的标识号为 i，并假设选出的协调器总是具有最大标识号的活跃站点。这样，当协调器发生故障时，该算法必须选出具有最大标识号的活跃站点。该算法必须将该号码送到系统中的每个活跃站点。另外，该算法必须提供一种能使从崩溃中恢复的站点识别出当前协调器的机制。假设站点 S_i 发出的请求在预定的时间间隔 T 内没有得到协调器的回答。在这种情况下，假设协调器发生了故障，并且 S_i 试图选举
851 自己作为新协调器的站点。

站点 S_i 给每个具有更大标识号的站点发一条选举消息。接着 S_i 就等待一个时间间隔 T，等这些站点中的任何一个作出回答。如果在时间 T 内没有收到任何回答，它就假设具有大于 i 的号码的所有站点都已经发生故障，于是选举自己来作新协调器的站点，并给具有小于 i 的标识号的所有活跃站点发消息，通知这些站点它已成为新协调器所在站点。

如果 S_i 收到回答，它就开始一个时间间隔 T'，来接收通知自己一个具有更大标识号的站点已选中的消息。（某些别的站点正选举自己为协调器，并应该在时间 T' 内报告结果。）如果在 T' 时间内没有收到消息，那么它就假设具有更大标识号的站点发生了故障，站点 S_i 重新开始该算法的执行。

当故障站点恢复后，它马上开始执行相同的算法。如果没有具有更大标识号的活跃站点，恢复后的站点就强迫具有较小标识号的所有站点让自己成为协调器站点，即使当前有一个具有较小标识号的活跃协调器。正因为如此，这个算法称为威逼算法。如果出现网络划分，威逼算法在每个分区中选举出一个独立的协调器；为了保证至多选举出一个协调器，获胜的站点应该另外查证多数站点在它们的分区中。

19.6.6 为可用性而牺牲一致性

到目前为止我们所看到的协议需要（加权的）大多数站点位于一个分区内以进行更新。位于少数分区内的站点是不能进行更新的：如果网络故障造成了多于两个分区，可能没有分区包含大部分站点。在这种情况下，系统将会完全无法用于更新，并且根据读法定人数，甚至可能导致无法用于读。我们前面看到的号个部可用协议提供了可用性，但不提供一致性。

理想情况下，即使面对划分，我们也希望获得一致性和可用性。遗憾的是，这是不可能的，这一事实就是所谓的 CAP 定理（CAP theorem）所确定的。它表明任何分布式数据库最多只能具有以下三个性质中的两个：

- 一致性（consistency）。
- 可用性（availability）。
- 划分容忍性（partition-tolerance）。

CAP 定理的证明借助于下述复制数据上的一致性定义：如果在复制数据上执行一组操作（读和写）
852 的结果与在单个站点上以某种串行次序执行这些操作的结果相同，并且该串行次序与每个进程（事务）发出操作的次序相一致，那么就称该组操作在复制数据上是一致的。一致性的概念类似于事务的原子性，只是其中每个操作被看作一个事务，且服于事务的原子性性质。

在任何大规模分布式系统中，划分是不可避免的，其结果是必须牺牲可用性或者一致性。我们前面看到的方案在面临划分时为了一致性而牺牲了可用性。

考虑一个基于 Web 的社交网络系统，其数据被复制在三台服务器上，并且出现了一个网络划分，它阻止了这些服务器之间的相互通信。由于没有一个分区包含大多数站点，因此不可能在任意一个分区上执行更新。如果这些服务器中的一台与用户在同一个分区中，用户实际上已经访问了数据，但会无法更新数据，因为另一个用户可能正在另一个分区中并发更新相同的对象，这将造成潜在的不一致性。与银行数据库中相对，社交网络系统中不一致性的风险要小。这种系统的设计者可以决定一个能够访问系统的用户应该允许在可访问的任何副本上执行更新，哪怕存在不一致的风险。

与需要满足 ACID 性质的银行数据库那样的系统不同，上述诸如社会网络系统那样的系统要求满足 BASE 性质：

- 基本上可用(basically available)。
- 软状态(soft state)。
- 最终一致性(eventually consistent)。

主要的需求是可用性，哪怕以一致性为代价。即使在出现划分的情况下，也应该允许更新，下面将以写所有可用协议为例(这类似于 19.5.3 节描述的多主站点复本)。软状态指的是数据库状态可能不能准确确定的性质，由于网络划分每个副本有可能有一定程度不同的状态。最终一致性要求，当解决划分之后，最终所有副本之间将形成一致。

最后一步要求能够识别不一致的数据项副本：如果一个副本是另一个副本的更早版本，应该用较新的版本取代更早的版本。然而，还是可能出现对一个共同的基本副本进行独立更新而产生的两个版本。一种检测这种不一致更新的方案叫版本向量方案，这将在 25.5.4 节描述。

应对不一致性更新的一致性恢复需要把各种更新以某种对应用来说有意义的方式进行整合。此步骤不能由数据库来处理；相反，数据库检测并通知应用程序有关不一致性的现象，然后由应用程序决定如何处理不一致性。 [853]

19.7　分布式查询处理

在第 13 章中，我们看到计算一个查询的结果有各种方法。我们调查了几种技术来选择一种处理查询的策略，它使得花费在计算结果上的时间总和最小。对集中式系统来说，衡量特定策略的代价的基本准则是磁盘访问量。在分布式系统中，我们必须考虑另外的几个因素，包括：

- 数据在网络上的传输代价。
- 通过在几个站点并行处理查询的一部分而可能得到的性能提升。

数据在网络上传输和数据转入和转出磁盘的相对代价依赖于网络类型和磁盘速度而变化很大。因此，通常情况下我们不能仅仅注意磁盘开销或网络开销。相反，我们必须在这两者之间取得良好的折中。

19.7.1　查询转换

考虑一个极其简单的查询："找出 *account* 关系中的所有元组。"虽然这个查询很简单(实际上是微不足道的)，但对它的处理不是很不足道的，因为正如我们在 19.2 节中看到的那样，*account* 关系可能分片，复制，或者既分片又复制。如果 *account* 关系被复制，我们就需要选择副本。如果所有副本都没有分片，我们选择传输代价最小的副本。但是，如果副本分片了，选择就不那么容易了，因为我们需要计算一些连接或并来重构 *account* 关系。在这种情况下，我们这个简单例子的策略可能会很多。在这种情况下，通过穷举所有可能的策略来对查询进行优化是不现实的。

分片透明性意味着用户可以书写这样的查询：

$$\sigma_{\text{branch\_name = "Hilla"}}(account)$$

由于 *account* 定义为：

$$account_1 \cup account_2$$

由名字翻译模式产生的表达式为：

$$\sigma_{\text{branch\_name = "Hilla"}}(account_1 \cup account_2)$$

采用第 13 章的查询优化技术，我们可以自动简化上述表达式。结果表达式为： [854]

$$\sigma_{\text{branch-city}=\text{"Hillside"}}(account_1) \cup \sigma_{\text{branch-city}=\text{"Hillside"}}(account_2)$$

此表达式包括两个子表达式。第一个只涉及 $account_1$，因此可以在 Hillside 站点上求值。第二个只涉及 $account_2$，因此可以在 Valleyview 站点上求值。

在对

$$\sigma_{\text{branch-city}=\text{"Hillside"}}(account_1)$$

求值时还可以进一步优化。由于 $account_1$ 只包含属于 Hillside 支行的元组，因此我们可以去掉选择运算。在对

$$\sigma_{\text{branch-city}=\text{"Hillside"}}(account_2)$$

求值时，我们可以运用 $account_2$ 分片的定义来得到：

$$\sigma_{\text{branch-city}=\text{"Hillside"}}(\sigma_{\text{branch-name}=\text{"Valleyview"}}(account))$$

不管 *account* 关系的内容是什么，此表达式都是空集。

因此，我们最终的策略就是让 Hillside 站点返回 $account_1$ 作为查询结果。

19.7.2 简单的连接处理

正如我们在第 13 章中看到的那样，选择查询处理策略的一个主要决定就是选择连接策略。考虑下面的关系代数表达式：

$$account \bowtie depositor \bowtie branch$$

假设这三个关系都既没复制又没分片，且 *account* 存储在站点 S_1 上，*depositor* 存储在 S_2 上，*branch* 存储在 S_3 上。设 S_I 表示发出查询的站点。系统需要在站点 S_I 产生结果。下面是处理这个查询可以采取的几种策略：

- 将所有三个关系的副本都送到站点 S_I。使用第 13 章的技术，在站点 S_I 上选择一种本地处理整个查询的策略。
- 将关系 *account* 的一个副本送到站点 S_2，并在 S_2 上计算 $temp_1 = account \bowtie depositor$。将 $temp_1$ 从 S_2 送到 S_3，并在 S_3 上计算 $temp_2 = temp_1 \bowtie branch$。将结果 $temp_2$ 送到 S_I。
- 设计对称一种类似的策略，只是交换 S_1、S_2、S_3 的角色。

855 没有一种策略总是最好的。必须考虑的因素中包括传输的数据量、在一对站点间传输数据块的代价以及各个站点上处理的相对速度。考虑上面列出的前两种策略。假设 S_2 和 S_3 上的索引对于计算连接是有用的。如果我们将所有三个关系都送到站点 S_I，我们要么需要在 S_I 上重新创建这些索引，要么使用不同的但可能更昂贵的连接策略。重新创建索引必须承担额外的处理开销和额外的磁盘访问。在使用第二种策略的情况下，一个可能很大的关系（$account \bowtie depositor$）必须从 S_2 送到 S_3。此关系中，客户每拥有一个账户，其名字就要重复一次。因此，同第一种策略相比，第二种策略可能导致额外的网络传输。

19.7.3 半连接策略

假设我们希望计算表达式 $r_1 \bowtie r_2$，其中 r_1 和 r_2 分别存储在站点 S_1 和 S_2 上。令 r_1 和 r_2 的模式为 R_1 和 R_2。假设我们希望在站点 S_1 得到结果。如果 r_2 中有许多元组不能和 r_1 中的任何元组相连接，那么把 r_2 送到 S_1 就必须传输那些对结果毫无贡献的元组。我们希望能在把数据送到 S_1 以前去掉这样的元组，特别是在网络代价高的时候。

为了实现所有这些，一种可能的策略是：

1. 在 S_1 计算 $temp_1 \leftarrow \Pi_{R_1 \cap R_2}(r_1)$。
2. 将 $temp_1$ 从 S_1 送到 S_2。
3. 在 S_2 计算 $temp_2 \leftarrow r_2 \bowtie temp_1$。
4. 将 $temp_2$ 从 S_2 送到 S_1。
5. 在 S_1 计算 $r_1 \bowtie temp_2$，产生的关系和 $r_1 \bowtie r_2$ 一样。

在考虑这种策略的效率之前，让我们来验证此策略能计算出正确的结果。在第 3 步中，$temp_2$ 是 $r_2 \bowtie \Pi_{R_1 \cap R_2}(r_1)$ 的结果。在第 5 步中，计算：

$$r_1 \bowtie r_2 \bowtie \Pi_{R_2 \cap R_3}(r_3)$$

由于连接是可结合并且可交换的，因此可以将此表达式重写为：

$$(r_1 \bowtie \Pi_{R_2 \cap R_3}(r_3)) \bowtie r_2$$

由于 $r_2 \bowtie \Pi_{(R_2 \cap R_3)}(r_3) = r_2$，因此这个表达式事实上等于 $r_1 \bowtie r_2$，此表达式就是我们要计算的表达式。

当 r_2 中有相对较少的元组对连接有贡献时，这种策略的优势尤其明显。如果 r_1 是一个涉及选择的关系代数表达式的结果，这种情况就可能发生。在这种情况下，$temp_2$ 的元组数可能明显少于 r_2。这种策略节省的开销源于只须将 $temp_2$ 而不是 r_2 的全部送到 S_1。增加的开销源于将 $temp_3$ 送到 S_2。如果 r_2 856
中对连接有贡献的元组比例足够小，传输 $temp_3$ 的开销就远小于只传输 r_2 中的部分元组而非全的开销。

这种策略称为**半连接策略**(semijoin strategy)，以关系代数中的半连接运算符来命名。记为$\ltimes$。r_1 和 r_2 的半连接记为 $r_1 \ltimes r_2$，即：

$$\Pi_R(r_1 \bowtie r_2)$$

因此，$r_1 \ltimes r_2$ 选出了关系 r_1 中对 $r_1 \bowtie r_2$ 有贡献的那些元组。在第 3 步中，$temp_2 = r_2 \ltimes r_1$。

对于几个关系的连接来说，这种策略可扩展到一系列的半连接步骤。一个关于将半连接用于查询优化的坚实的理论体系已经发展起来。这方面的一些理论在文献注解中有引用。

19.7.4　利用并行性的连接策略

考察对 4 个关系的一个连接：

$$r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$$

其中关系 r_i 存储在站点 S_i。假设结果必须在站点 S_1 给出。有很多可能的并行求值策略。(第 18 章详细研究过并行查询处理的问题)。在一种这样的策略中，将 r_3 送到 S_1，并在 S_1 上计算 $r_3 \bowtie r_4$。同时，将 r_2 送到 S_4，并在 S_4 上计算 $r_2 \bowtie r_1$。在计算 $r_2 \bowtie r_1$ 的过程中站点 S_4 就可以把已经计算出的元组送到 S_1，而不是等到整个连接计算完。同样，S_4 也可以这样把 $(r_2 \bowtie r_1)$ 中元组送到 S_1。采用 12.7.2.2 节的流水线连接技术，一旦 $(r_3 \bowtie r_4)$ 和 $(r_2 \bowtie r_1)$ 的元组到达 S_1，就可以开始计算 $(r_1 \bowtie r_2) \bowtie (r_3 \bowtie r_4)$。因此，$S_1$ 上最终连接结果的计算可以同 S_1 上 $(r_3 \bowtie r_4)$ 的计算以及 S_4 上 $(r_2 \bowtie r_1)$ 的计算并行地进行。

19.8　异构分布式数据库

很多新的数据库应用需要来自各种先前存在的数据库中的数据，这些数据库位于异构的硬件及软件环境的集合中。操纵位于异构分布式数据库中的信息需要在已有数据库系统之上增加一个软件层。这个软件层称为**多数据库系统**(multidatabase system)。局部的数据库系统可以采用不同的逻辑模型以及数据定义和数据操纵语言，并可以在它们的并发控制和事务管理机制上存在差异。多数据库系统构造了逻辑上集成数据库的一种虚拟，而不需要将数据库在物理上集成。

将异构系统完全集成为一个同构的分布式数据库通常是困难的或不可能的： 857

- **技术上的困难**(technical difficulty)。基于已有数据库系统的应用程序的投资可能是巨大的，而将这些应用进行转化的代价可能高不可及。
- **组织上的困难**(organizational difficulty)。即使集成在技术上是可能的，它在政策上可能是不可行的，因为已有数据库系统属于不同的公司或组织。在这种情况下，对多数据库系统来说，允许局部数据库系统在局部数据库及其数据上运行的事务方面保持高度自治(autonomy)是很重要的。

由于这些原因，多数据库系统带来的显著优势超出了它们的开销。本节从数据定义和查询处理的立场，给出构造多数据库环境所面临的挑战的一个概览。

19.8.1　数据统一视图

每个本地数据库管理系统可能使用不同的数据模型。比如，有些可以采用关系模型，而另一些可能采用更早的数据模型，如网状模型(见附录 D)或层次模型(见附录 E)。

由于多数据库系统需要提供虚拟的、集成的单一数据库系统，因此必须使用一个公共的数据模型。

一种通用的选择是关系模型，并将 SQL 作为公共查询语言。事实上，现在存在多个系统使 SQL 查询能应用到非关系的数据库管理系统之上。

另一个困难是提供公共的概念模式。每个局部系统都提供它自己的概念模式。多数据库系统必须将这些独立的模式集成为一个公共模式。模式集成是一项复杂的任务，主要是由于语义的差异。

模式集成不是在数据定义语言之间简单地直接转换。相同的属性名可能出现在不同的局部数据库中但具有不同的含义。在一个系统中使用的数据类型可能不被另一个系统支持，而且类型之间的转换可能也不简单。即使对于相同的数据类型，也可能由于数据的物理表示而出现问题：一个系统可能用 8 位 ASCII，而另一个用 16 位 Unicode，另外还可能用 EBCDIC；浮点表示可能不同；整数可以用高位在前或低位在前的形式表示。在语义层，一个系统中表示长度的整数值可能是英寸而另一个系统中可能是毫米。因此造成整数相等只能是一种近似概念这样的尴尬境地（正如浮点数通常的情况一样）。相同的字符可能出现在不同系统的不同语言中。例如，美国的系统可能表示城市"Cologne"，而德国的系统表示"Köln"。

所有这些看似细小的区别必须正确记录到公共全局概念模式中。必须提供转换函数。必须为与系统相关的行为（例如，非字母字符的排序在 ASCII 中和在 EBCDIC 中是不同的）附注索引。正如我们前面已经注意到的，另一种将各个数据库转换为一种公共格式的方法在不废弃已有应用程序的前提下是行不通的。

858

19.8.2 查询处理

异构数据库的查询处理会比较复杂。有以下一些问题：

- 给定一个在全局模式上的查询，该查询可能必须翻译成需要执行查询的每个站点的本地模式上的查询。查询结果必须翻译回全局模式。

  通过为每个数据源书写包装器（wrapper）可以简化该任务，包装器提供一种全局模式中的本地数据的视图。包装器也将全局模式上的查询翻译成本地模式上的查询，并将结果翻译回全局模式。包装器可以由单独站点来提供，或者也可以作为多数据库系统的一部分来单独书写。

  包装器甚至可用于为非关系数据源提供关系视图，例如网页（可能具有表单界面）、平面文件、层次和网状数据库，以及目录系统。
- 某些数据源可能仅提供有限的查询能力；例如，它们可能支持选择，但不支持连接。它们甚至可能限制选择的形式，允许选择仅在特定域上进行；具有表单界面的 Web 数据源就是这种数据源的一个例子。查询匹此可能必须分开执行，使得部分在数据源执行，部分在发出查询的站点上执行。
- 一般来说，为了回答给定查询可能需要访问不止一个站点。可能必须处理从这些站点获得的结果以去除重复。假设一个站点包含满足选择条件 *balance* < 100 的 *account* 元组，而另一个站点包含满足 *balance* > 50 的 *account* 元组。在整个 *account* 关系上的查询将需要访问这两个站点，并去除由余额在 50 - 100 之间的元组（这些元组在两个站点重复存在）所导致的重复答案。
- 在异构数据库中的全局查询优化是困难的，因为查询执行系统可能不知道可选的查询计划在不同站点上执行的代价。通常的解决方案是只依靠本地级别的优化，而且仅仅在全局级别使用启发式策略。

中间件（mediator）系统是集成多个异构数据源、提供数据的一个集成的全局视图并提供在全局视图上的查询工具的系统。不像成熟的多数据库系统，中间件系统不会干扰事务处理。（中间件和多数据库这两个术语经常可以交换使用，并且称为中间件的系统可能支持受限形式的事务。）术语**虚拟数据库**（virtual database）用来指代多数据库/中间件系统，因为这两种系统提供了具有全局模式的单一数据库的表象，尽管数据是以本地模式存在于多个站点上。

859

19.8.3 多数据库中的事务管理

多数据库系统支持两种事务类型：

1. **局部事务**（local transaction）。这些事务由每个本地数据库系统执行，不受多数据库系统的控制。

2. **全局事务**(global transaction)。这些事务是在多数据库系统的控制下执行的。

多数据库系统知道局部事务可能运行在本地站点上的这一事实，但它不知道正在执行的是什么样的特定事务，或者它们可能访问什么样的数据。

要确保每个数据库系统的局部自治性，要求数据库系统自身的软件不能发生变化。因此在一个站点上的数据库系统不能与在其他任何站点上的数据库系统直接通信，以同步执行在几个站点上活跃的全局事务。

由于多数据库系统不能控制本地事务的执行，因此每个本地系统必须使用并发控制方案(例如，两阶段锁或时间戳)来确保它的调度是可串行化的。此外，在使用封锁的情况下，本地系统必须能够提防本地死锁的可能性。

保证局部可串行化并不足以确保全局可串行化。举个例子，考虑两个全局事务 T_1 和 T_2，每个都访问并更新分别位于 S_1 和 S_2 站点上的两个数据项 A 和 B。假设本地调度是可串行化的。仍然可能出现这种情况：在站点 S_1，T_1 在 T_2 之后执行，而在站点 S_2，T_2 在 T_1 之后执行，这就导致了全局调度的不可串行化。事实上，即使全局事务之间不是并发的(也就是说，全局事务只有在它前面的事务提交或中止之后才能提交)，局部可串行化也不足以保证全局可串行化(见实践习题 19.14)。

根据本地数据库系统的实现，全局事务可能不能控制其本地子事务的精确封锁行为。因此，即使所有本地数据库系统都遵循两阶段封锁，它也可能只能确保每个局部事务是遵循协议规则的。例如，一个本地数据库系统可能提交其子事务并释放封锁，而在另一个本地系统中的子事务仍在执行。如果本地系统允许控制封锁行为，并且所有系统都遵循两阶段封锁，那么多数据库系统可以确保全局事务以两阶段方式封锁，进而冲突事务的锁点能够决定它们的全局串行化次序。但是，如果不同的本地系统采用不同的并发控制机制，这种直观的全局控制方法就行不通了。 [860]

尽管在多数据库系统中并发执行全局事务和局部事务，但是有很多确保一致性的协议。有些协议是基于施加足够的条件来确保全局可串行性。其他的仅保证一种弱于可串行性的一致性，但通过较少的限制手段来实现这种一致性。26.6 节描述不带可串行性的一致性方法；在文献注解中引用了其他的方法。

早期的多数据库系统将全局事务限制为只读事务。它们由此避免全局事务给数据带来不一致性的可能性，但是没有作出足够的限制来确保全局可串行性。的确可以得到这种全局调度并开发一套方案来保证全局可串行性，我们将在实践习题 19.15 中让你们去实现这两种设想。

有很多通用方案用于确保在可以执行更新和只读事务的环境中的全局可串行性。几种这样的方案是基于**标签**(ticket)思想的。称作标签的特殊数据项在每个本地数据库系统中创建。访问站点上数据的每个全局事务必须在该站点上写标签。此要求确保全局事务在它们所访问的站点上发生直接的冲突。进而，全局事务管理器可以通过控制被访问标签的次序来控制全局事务的串行化顺序。文献注解中有对这类方案的引用。

如果我们要在每个站点都不会产生直接的本地冲突的环境中保证全局可串行化，那就必须作出一些关于本地数据库系统所允许的调度的假设。例如，如果本地调度是那些提交次序与串行化次序总保持一致的调度，我们可以通过只控制事务的提交次序来确保可串行化。

在多数据库系统中的一个相关问题是全局的原子性提交。如果所有本地系统都遵循两阶段提交协议，该协议便可用于实现全局的原子性。然而，如果本地系统不是作为分布式系统的一部分来设计的话，就可能无法参与到这样的协议中。即使本地系统能够支持两阶段提交，但拥有该系统的组织机构可能不愿意允许在发生阻塞的情况下等待。在这种情况下，可能会作出允许在特定的失效模式中不遵守原子性的妥协。文献中有对这些问题的进一步讨论(参见文献注解)。

19.9　基于云的数据库

云计算(cloud computing)是在 20 世纪 90 年代末和 21 世纪出现在计算领域中的一个相对新的概念，开始命名为**软件即服务**(software as a service)。最初的软件服务供应商提供的是驻留在他们自己的机器上的、特定的、可定制的应用程序。随着软件供应商开始提供通用的计算机来作为服务，并且客户可

以在这些计算机上运行他们所选的应用软件。云计算的概念也随之发展。客户可以与云计算供应商达
861 成协议来获得具有特定能力和特定数据存储量的特定数量的机器。机器数量和存储容量都可以根据需要来增加和缩减。除了提供计算服务，很多供应商还提供其他服务，比如数据存储服务、地图服务以及能够通过使用 Web 服务应用编程接口来访问的其他服务。

很多企业正在寻找云计算与服务的双赢模式。它免除了企业客户维护一个大型的系统支撑团队的需要，并允许新的企业不必在计算系统上进行大量的前期资本投入就可以开始运作。还有，随着企业需求的增长，可用按需添加更多的资源(计算和存储)；云计算供应商通常拥有很大的计算机集群，使得他们能够容易地按需分配资源。

很多供应商都提供云服务。其中包括传统的计算供应商以及正在寻找利用它们拥有的大规模基础设施来代替它们的核心业务的公司，比如 Amazon 和 Google。

需要为非常大量的用户(范围从数百万到数亿)存储和检索数据的 Web 应用已经成为基于云的数据库的主要驱动者。这些应用的需求同传统数据库应用是不同的，因为它们比一致性更看重可用性和扩展性。近年来已经开发出几种基于云的数据存储系统来满足这类应用的需求。19.9.1 节将讨论在云上创建这种数据存储系统的问题。

在 19.9.2 节，我们考虑在云上运行传统数据库系统的问题。基于云的数据库同时具备同构和异构系统的特点。虽然数据被某个组织(客户)拥有并且是某个统一的分布式数据库的一部分，但底层的计算机被另一个组织(服务供应商)拥有和管理。计算机距离客户的位置很近，是通过 Internet 访问的。因此，异构分布式系统的一些挑战依然存在，特别是考虑到事务处理的情况。尽管如此，异构系统的很多组织性和政治性挑战是可以避免的。

最后，19.9.3 节将讨论云数据库目前面临的几种技术性和非技术性挑战。

19.9.1 云上的数据存储系统

Web 上的应用有惊人的可扩展性要求。常见的应用具有数亿用户，并且许多应用在一年之内，甚至在几个月之内，它们的负载就成倍增长。要满足这类应用对数据管理的需求，数据必须在数千个处理器上进行划分。

在过去几年中已经开发并部署了在云上的许多数据存储系统来满足这类应用对数据管理的需求。它们包括 Google 的 *Bigtable*；来自 Amazon 的 *Simple Storage Service*(*S3*)，它给 *Dynamo* 提供了 Web 接口，
862 是一种键-值存储系统；来自 FaceBook 的 *Cassandra*，它和 Bigtable 类似；来自雅虎的 *Sherpa/PNUTS*；来自微软的 *Azure* 环境的数据存储组件；以及其他几个系统。

本节提供对这些数据存储系统的体系架构的概览。虽然一些人把这些系统称作分布式数据库系统，但它们并不提供被认为是目前数据库系统标准的很多特性，比如对 SQL 的支持，或者对具有 ACID 性质的事务的支持。

19.9.1.1 数据表示

作为 Web 应用对数据管理需求的例子，考虑用户的个人简介，由一家组织机构运行的很多不同的应用需要访问用户简历。简历包含各种属性，并且会频繁添加存储在简历中的属性。某些属性可能包含复杂的数据。简单的关系表示往往不足以表示这些复杂数据。

一些基于云的数据存储系统支持用于表示这种复杂数据的 XML(将在第 23 章介绍)。另一些支持 **JavaScript 对象表示法**(JavaScript Object Notation，JSON)表示，越来越多地接受 JSON 来表示复杂数据。XML 和 JSON 表示提供了一条记录所包含的属性集以及这些属性类型的灵活性。然而还有一些系统，比如 Bigtable，为复杂数据定义了它们自己的数据模型，包括支持具有非常多的可选列的记录。本节后面再来探讨 Bigtable 数据模型。

此外，很多这样的 Web 应用要么并不需要全面的查询语言支持，要么至少在没有这样的支持下可以进行管理。主要的数据访问模式是存储具有相关码的数据，并根据这个码检索数据。在上述用户简历的例子中，用户简历数据的码可以是用户的标识符。有一些应用在概念上需要连接，但通过视图物化的形式来实现连接。比如，在一个社交网络应用中，每个用户应该可以看到来自其所有朋友的新邮件。遗憾的是，当数据分布在很多机器上时，找到朋友的集合然后查询每个人来找到他们的邮件会带

来很大的延迟。一种替代的方式如下：只要一个用户发送邮件，就向该用户的所有朋友发送一条消息，并且用新邮件的摘要来更新与每个朋友相关联的数据。当该用户检查更新时，所有所需数据已经在某个地方可用并可以快速检索。

因此，云数据存储系统的核心是基于两个基本函数：用于存储具有相关码的值的 **put(key, value)** 和用于检索与特定码关联的存储值的 **get(key)**。一些系统额外提供码值上的范围查询，比如 Bigtable。

在 Bigtable 中，一条记录不会作为单个值来存储，而是划分或组成属性来分开存储。因此，一个属性值的码在概念上由(记录标识符、属性名)组成。每个属性值在 Bigtable 中只是一个字符串。要获得一条记录的所有属性，需要使用范围查询或者仅由记录标识符构成的更精确的前缀匹配查询。get() 函数返回属性名和对应的值。为了高效地检索出一条记录的所有属性，存储系统按码的次序存放各个项，这样一条特定记录的所有属性值就是聚集在一起的。 863

JSON

JavaScript 对象表示法或 JSON 是复杂数据类型的文本表示方式，它广泛用于应用之间的数据传输，以及复杂数据存储。JSON 支持基本数据类型，如整数、实数和字符串类型，以及数组类型和"对象"，对象是(属性名、值)对的集合。JSON 对象的一个例子是：

```
{
"ID": "22222",
"name": {
"firstname": "Albert",
"lastname": "Einstein"
},
"deptname": "Physics",
"children": [
{"firstname": "Hans", "lastname": "Einstein" },
{"firstname": "Eduard", "lastname": "Einstein"
}
]
}
```

上面的例子说明对象包含(属性名、值)对，并且数组是用方括号界定的。JSON 可以看成是 XML 的一种简化形式；XML 将在第 23 章介绍。

开发了一些库用于在 JSON 表示和用在 JavaScript 和 PHP 脚本中以及其他编程语言中的对象表示之间的数据转换。

实际上，记录标识符本身可以层次化构造。尽管对于 Bigtable 自身来说，记录标识符只是一个字符串。例如，一个存放通过网络爬虫抓取到的页面的应用可以把如下形式的 URL：

www. cs. yale. edu/people/silberschatz. html

映射到记录标识符：

edu. yale. cs. www/people/silberschatz. html 864

这样页面就按照有用的顺序聚集起来。作为另一个例子，在 JSON 例子(见 JSON 的示例框)中的记录可用具有"22222"标识符的记录来表示，它有多个属性名，比如"name. lastname"、"deptname"、"children[1]. firstname"或"children[2]. lastname"。

此外，通过在记录标识符前面简单地加上应用名和表名作为前缀，单个 Bigtable 实例可以为多个应用存储数据，每个应用具有多个表。

数据存储系统基本上都允许存储数据项的多个版本。版本号往往采用时间戳来标识，但也可以通过一个整数值标识的方式来代替，每创建一个数据项的新版本时该整数值就递增。查找可以指定数据项的所需版本，或者挑选版本号最高的版本。比如，在 Bigtable 中，码实际上由三部分组成：(记录标识符、属性名、时间戳)。

19.9.1.2 划分与检索数据

当然，数据划分是数据存储系统中处理超大规模数据的关键。和常规并行数据库不同，它常常不

可能提前决定划分函数。此外，如果负载增加，就需要增加更多的服务器，并且每个服务器应该能够增量式地承担部分负载。

为了解决这两种问题，数据存储系统通常把数据划分成相对小的单元（在这种系统上的小可能意味着数 MB 的量级）。这种划分通常叫表块（tablet），反映了每个表块作为表中一部分的事实。数据划分应该根据搜索码来进行，这样对特定码值的请求就定位到单个表块；否则每个请求需要在多个站点上处理，大大增加了系统负载。有两种方法可以使用：或者在码上直接使用范围划分，或者在码上应用散列函数并且在散列函数的结果上应用范围划分。

表块所在的站点作为该表块的主站点。所有更新通过该站点路由，然后把更新传播到表块的副本上。查找也发送到相同的站点，这样读与写才是一致的。

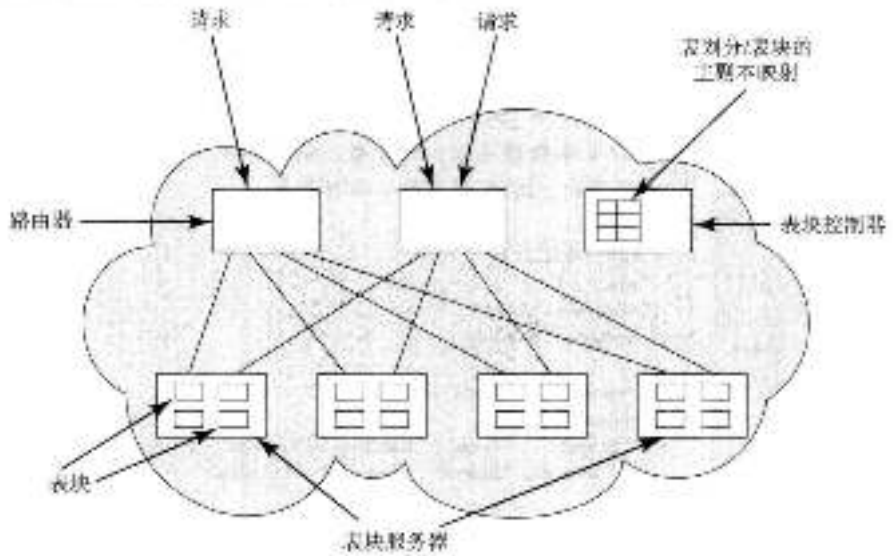


图 19-7 云数据存储系统体系结构

把数据划分到表块上并不是预先确定好的，而是动态进行的。随着数据的插入，如果表块变得过大，就会分裂成更小的部分。此外，即使表块没有达到分裂的指标，如果在这个表块上的负载（获取/上传操作）过大，这个表块也可能拆分成更小的表块，这些小的表块可以分布在两个或多个站点上来分担负载。通常表块的数量远多于站点数量，其原因与并行数据库中使用虚拟划分一样。

知道整个系统中哪个站点负责特定表块是非常重要的。这可以通过表块控制器站点来实现。此站点跟踪划分函数，以便把 get() 请求映射到一个或多个表块。此外还跟踪从表块到站点的映射函数，以找到哪个站点负责哪个表块。进入系统的每个请求必须路由到正确的站点；如果只有一个表块控制
865 器站点负责此项任务，那么它很快就会过载。相反，映射信息可复制到路由器站点集上，这些路由器站点把请求路由到具有相应表块的站点上。当拆分或移动表块时，用于更新映射信息的协议被设计成不使用封锁的方式；其结果是请求可能在错误的站点上结束。对此问题的处理是通过检测该站点不再负责请求所指定的码，并根据最新的映射信息来重新路由请求。

图 19-7 描述了云数据存储系统的体系结构，它大致基于 PNUTS 架构。其他系统也表现了类似的功能，尽管它们的体系结构可能有变化。例如，Bigtable 并没有独立的路由器；划分和表块－服务器的映射信息是存放在 Google 文件系统中的，客户端从文件系统中读取信息，并决定向何处发送它们的请求。

19.9.1.3 事务和复制

云上的数据存储系统通常不完全支持事务的 ACID。两阶段提交的代价太高，而且两阶段提交可能在失效发生时导致堵塞，这是典型的 Web 应用不能接受的。这就意味着这种系统通常甚至不支持事务一致性的二级索引；二级索引可能按照与存储数据所使用的码不同的属性进行划分，从而插入或更新

就需要更新两个站点，这就需要两阶段提交。在最好情况下，这样的系统支持在单个表块内数据上的事务，通过单个主站点来控制。Sherpa/PNUTS 还提供测试与设置函数，允许在数据项当前版本与指定 866
版本号相同的情况下对数据项进行更新。如果数据项的当前版本号比指定版本号要新，那么更新就不会执行。应用可以用测试与设置函数来实现一种受限形式的基于有效性检查的并发控制，其有效性检查被限制在单个表块中的数据项上。

在具有数千个站点的系统中，几乎可以肯定在任何时间都会有几个站点宕机。云上的数据存储系统必须在即使有很多站点宕掉的情况下也能够继续正常的处理。这种系统把数据（比如表块）复制到同一个集群中的多台机器上，这样即使一个集群中的某些机器宕掉，这些数据的副本还可能是可用的。（集群（cluster）是在一个数据中心的机器的集合。）比如，谷歌文件系统（Google File System，GFS）是一个分布式容错的文件系统，它把所有文件系统块复制到一个集群中的三个或多个站点上。只要至少有一个数据副本可用，正常的操作就可以继续（关键的系统数据，比如文件到站点的映射，复制到更多的站点上，其中的大部分必须可用）。此外，副本还可以在地理上分布的集群之间使用，至于其原因我们很快就会看到。

由于每个表块只由单个主站点控制，如果这个站点失效，那么表块应该重新指派给具有该表块副本的一个不同的站点，该站点就成为该表块新的主站点。对表块的更新记录为日志，并且日志本身也备份。当站点失效时，该站点上的表块指派给其他站点；每个表块新的主站点负责执行恢复动作，使用日志来使其表块副本达到最新的一致性状态，此后可以在该表块上执行更新和查找。

比如，在 Bigtable 中映射信息存储在一个索引结构中，并且这个索引和实际的表块数据一起存储在文件系统中。表块数据的更新并没有立即执行，但日志数据是。文件系统确保文件系统数据被复制而且面对集群中的几个站点失效时依然可用。因此，当一个表块被重新指派时，该表块新的主站点能够访问到最新的日志数据。另一个方面，雅虎的 Sherpa/PNUTS 系统显式地把表块复制到集群中的多个站点上，而不是使用分布式文件系统，并且采用可靠的分布式消息系统来实现高可用性日志。

遗憾的是，整个数据中心变得不可用并非是不常见的——例如，由于自然灾害或者火灾。因此对于高可用性而言，远程站点的备份是非常重要的。对于很多 Web 应用，在远距离网络上的往返延迟会严重影响性能，在使用 Ajax 应用中这个问题会更加严重，Ajax 应用需要在浏览器和应用之间进行多轮通信。为了处理这个问题，用户连接到地理上离他们最近的应用服务器，并且数据在多个数据中心进行备份，这样其中一个副本可能靠近应用服务器。

但是，其结果是网络划分的危险增加了。假设大多数 Web 应用对可用性的重视超过了一致性，云 867
上的数据存储系统通常允许即使在发生划分时也执行更新，并提供对随后恢复一致性的支持，正如之前在 19.6.6 节介绍的那样。我们在 19.5.3 节看到的具有更新的延迟传播的多主复制常常用于处理更新。延迟传播意味着更新并不作为更新事务的一部分来传播给副本，尽管通常采用消息传递基础设施尽可能快地传播它们。

除了对数据项副本的更新传播，对二级索引的更新，或者对特定类型物化视图（比如之前在 19.9.1.1 节我们看到的社交网络应用中，来自好友的更新）的更新，都可以使用消息传递基础设施来进行发送。二级索引本质上是表，和正常的表一样基于索引搜索码划分：对表中一条记录的更新可以映射到表上二级索引中的一个或多个表块的更新。对这种二级索引或物化视图的更新没有事务性保证，只是尽最大努力保证更新尽快到达其目的地。

19.9.2 云上的传统数据库

我们现在考虑在云上实现一个支持 ACID 特性和查询的传统分布式数据库系统的问题。

计算工具是一个老概念，可以追溯到 20 世纪 60 年代。此概念的首次出现是在分时系统中，其中多个用户共享地访问单台大型计算机。后来，在 20 世纪 60 年代末，出现了虚拟机（virtual machine）的概念，在此概念中用户在感觉上好像拥有私人计算机，但实际上是单台计算机在模拟多台虚拟机。

云计算充分利用了虚拟机的概念来提供计算服务。虚拟机提供了很大的灵活性，因为客户可以选择他们自己的软件环境，不仅包括应用软件，而且包括操作系统。如果客户的计算需求较低，多个客户的虚拟机可以运行在单台物理计算机上。另一方面，如果客户的虚拟机负载很高，单台计算机可以

分配给一个客户的每台虚拟机。客户可以请求多台虚拟机，用于在其上运行应用程序。通过简单地增加或释放虚拟机，这使得随着负载的增加和缩减来添加或削减计算能力变得简单。

拥有一组虚拟机非常适合易于并行化的应用。数据库系统，正如我们所看到的，都属于这一类。每台虚拟机可以在本地运行数据库系统代码，并且表现方式类似于在一个同构分布式数据库系统中的站点。

19.9.3 基于云的数据库的挑战

与从头构建一个计算基础设施相比，基于云的数据库当然有几个重要的优势，并且对特定应用来
[868] 说实际上是必不可少的。

然而，基于云的数据库系统也有一些缺点，对此我们现在来探讨。与单纯的、很大程度上独立运行并行计算的计算应用不同，为了下述目的，分布式数据库系统需要在站点之间进行频繁通信与协调：

- 访问在另一台物理机器上的数据，要么是因为另一台虚拟机拥有该数据，要么是因为数据存储在独立于该虚拟机主机的存储服务器上。
- 获取远程数据上的封锁。
- 通过两阶段提交确保原子性事务的提交。

在早期的分布式数据库的研究中，我们（隐式地）假定数据库管理员能够控制数据的物理位置。在云系统中，数据的物理位置由供应商控制，并不是客户。因此，数据的物理放置就通信代价方面来说可能不是最理想的，并且这可能导致大量的远程封锁访求和虚拟机之间的大量数据传送。高效的查询优化要求优化器能够准确度量操作的代价。由于缺乏数据的物理放置知识，因此优化器不得不依赖于可能非常不准确的估计，导致低劣的执行策略。因为远程访问相比本地访问要慢，所以这些问题可能会对性能产生重大影响。

上述问题对于在云上实现传统数据库应用是一个特别的挑战，虽然对于简单数据存储系统来说没有多大挑战。接下来讨论的几种挑战可以同样地应用于这两种应用场景。

复制问题进一步加剧了基于云的数据管理的复杂性。云系统为了可用性而复制客户数据。事实上，如果没有维护特定级别的可用性，许多合约都有条款对供应商施加处罚。供应商实现这种复制时是不具备对于应用的特定知识的。由于复制是在云控制之下的，并不在数据库系统的控制之下，因此当数据库系统访问数据时必须小心，以确保读取的是数据的最新版本。如果不能正确考虑这些问题可能导致原子性或隔离性特性的损失。在目前许多云数据库应用中，应用本身可能需要为一致性承担一些责任。

云计算用户必须愿意接受他们的数据被另一家组织机构掌握的事实。这可能会带来在安全性和法律责任方面的各种风险。如果云供应商遭受安全性攻击，客户数据可能泄露，导致客户面临来自其用户的法律挑战。然而客户不能直接控制云供应商的安全性。如果云供应商选择将数据（或数据的副本）存储在其他国家，这些问题将变得更加复杂。各种法律司法权在它们的隐私法中是不同的。例如，如
[869] 果一家德国公司的数据被复制到纽约的服务器上，那么将采取美国的隐私法律，而不是德国的或欧盟的隐私法律。云供应商可能需要把客户数据发布给美国政府，尽管客户从不知道其数据将会落入美国司法权管辖之内。

特定的云供应商向它们的客户提供对他们的数据如何分布和复制的各种程度的控制。一些供应商直接向它们的客户提供数据库服务，而不需要客户为在原始存储和虚拟机上运行他们自己的数据库系统而签订合约。

云服务市场持续快速发展，但很显然签订云服务合约的数据库管理员必须考虑各种技术、经济和法律方面的问题，以确保数据的隐私和安全，保证 ACID 特性（或可接受的近似性质），以及足够的性能，尽管数据可能分布在广泛的地理区域上。文献注解提供了一些有关这些主题的当前见解。许多新的文献可能在未来几年出现，云数据库中的许多当前问题正由研究团体设法解决。

19.10 目录系统

考虑一家组织机构，它希望建立对该机构中各种人都可用的、有关其员工的数据；这类数据的例

子包括名称、头衔、员工号、地址、电子邮箱地址、电话号码、传真号码等。在计算机化以前，组织机构会创建物理的员工目录并把它们分发到机构中。甚至在现在，电话公司仍建立物理的客户目录。

一般而言，目录是关于某些类别的对象（例如人）的信息列表。目录可用于寻找有关特定对象的信息，或者反方向找到满足特定需求的对象。在物理电话目录范围内，满足前向查找的目录叫**白页**（white page），而满足反向查找的目录称为**黄页**（yellow page）。

在当今的网络时代，仍然存在对目录的需求，如果有什么不同的话，那就是变得更重要了。然而，当今的目录需要在计算机网络中是可用的，而不是有一种物理（纸张）形式中。

19.10.1 目录访问协议

目录信息可以通过 Web 界面使用，许多机构都是这么做的，尤其是电话公司。这样的界面是适合于人的。然而，程序也需要访问目录信息。目录可用于存储其他类型的信息，很像文件系统目录。例如，网络浏览器可以在目录系统中存储个人书签和其他浏览器设置。这样用户可以从多个地方（例如在家或在工作室）来访问同样的设置，而不需要共享文件系统。 870

已经开发出几种**目录访问协议**（directory access protocol）来提供访问目录中数据的标准化方式。其中，现在使用最广泛的是**轻量级目录访问协议**（Lightweight Directory Access Protocol，LDAP）。

显然在例子中的所有类型的数据都可以方便地存入数据库系统，并通过诸如 JDBC 或 ODBC 这样的协议来访问。这样就有一个问题：为什么要提出一种特殊协议用于访问目录信息呢？对这个问题至少有两个答案。

- 首先，目录访问协议是适合有限类型的数据访问的简化协议。它们与数据库访问协议是并行发展的。
- 其次，而且也是重要的，类似于文件系统目录名称，目录系统以一种分层的方式提供一种简单的对象命名机制，可用于分布式目录系统中来指明每台目录服务器中存储了何种信息。例如，一台特定的目录服务器可能存储了贝尔实验室在 Murray Hill 的员工信息，而另一台可能存储了贝尔实验室在 Bangalore 的员工信息，并给两个站点控制它们的本地数据的自治权。目录访问协议可用于通过网络从两个目录中获取数据。更重要的是，目录系统可以设置成在没有用户干预的情况下自动地将一个站点上所做的查询转发到另一个站点上。

因为这些原因，一些机构利用目录系统，使得通过目录访问协议可以在线获得机构信息。机构目录中的信息可用于各种目的，例如查找人的地址、电话号码或邮件地址，查找人们所在的部门，以及跟踪部门的层次结构。目录也可用于鉴别用户身份：应用可以收集鉴别信息，例如用户输入的密码，并利用目录鉴别它们。

正如可能预期的那样，几种目录的实现证明，使用关系数据库存储数据是有利的，而不是创建特殊目的的存储系统。

19.10.2 LDAP：轻量级目录访问协议

为多个客户端服务的目录系统通常作为一台或多台服务器来实现。客户端使用由目录系统定义的应用编程接口与目录服务器通信。目录访问协议还定义了数据模型和访问控制。

由国际标准化组织（ISO）定义的 X.500 **目录访问协议**（X.500 directory access protocol）是访问目录信息的标准。但是，此协议相当复杂，没有广泛使用。**轻量级目录访问协议**（Lightweight Directory Access Protocol，LDAP）提供了 X.500 的许多特性，但复杂性降低了，并广泛使用。本节剩余的部分将概述 LDAP 的数据模型和访问协议的细节。 871

19.10.2.1 LDAP 数据模型

在 LDAP 中目录存储的是类似于对象的**条目**（entry）。每个条目必须有**可区别名称**（Distinguished Name，DN），它唯一地标识了这个条目。DN 又由**相对可区别名称**（Relative Distinguished Name，RDN）的序列组成。例如，一个条目可能具有如下可区别名称：

cn = Silberschatz，ou = Computer Science，o = Yale University，c = USA

如你所看到的那样，在这个例子中的可区别名称是名称和（组织机构的）地址的组合，以人名开始，然

后给出组织部门(ou)、组织(o)和国家(c)。可区别名称的构成顺序和通常的邮件地址顺序一致，而不是在为文件指定路径名时采用的相反的顺序。用于一个 DN 的 RDN 集合由目录系统的模式来定义的。

条目也可以有属性。LDAP 提供二进制、字符串和时间类型，另外还有用于电话号码的 tel 类型，以及用于地址的 PostalAddress 类型(行间用 $ 字符分开)。与关系模型中的属性不同，这里的属性默认为多值的，因此可能为一个条目存储多个电话号码或者地址。

LDAP 允许定义带有属性名和类型的**对象类**(object class)。在定义对象类时可以使用继承。而且，可以指定条目属于一个或多个对象类。定义条目所属的一个最特殊的对象类是不必要的。

条目按照它们的可区别名称组织成一棵**目录信息树**(Directory Information Tree，DIT)。树的叶级条目通常表示特定的对象。作为内部结点的条目表示如组织部门、组织或国家这样的对象。一个结点的子结点具有包含其父结点所有 RDN 的 DN，以及一个或多个附加的 RDN。例如，一个内部结点可能有一个 DN：**c = USA**，在它以下的所有条目对 RDN c 都有 **USA** 这个值。

完整的可区别名称不必存储在条目中。系统可以通过从条目向上遍历 DIT，收集 **RDN = value** 的成分以产生完整的可区别名称的方式来产生一个条目的可区别名称。

条目可以有不止一个可区别名称——例如，对于一个在不止一家组织机构中的人的条目。为了处理这种情况，DIT 的叶结点可以是**别名**(alias)，它指向树的另一个分支中的条目。

19.10.2.2 数据操纵

与 SQL 不同，LDAP 既没有定义数据定义语言，也没有定义数据操纵语言。然而，LDAP 定义了一种用于执行数据定义和操纵的网络协议。LDAP 的用户既可以使用应用编程接口，也可以使用由不同厂商提供的工具来执行数据的定义和操纵。LDAP 还定义了一种文件格式，称为 LDAP 数据交换格式
872 (LDAP Data Interchange Format，LDIF)，它可用于存储和交换信息。

LDAP 中的查询机制十分简单，只有选择和投影，没有连接。查询必须指定以下几点：

- 一个基——DIT 中的一个结点——通过给出其可区别名称(从根结点到该结点的路径)。
- 一个搜索条件，可以是各个属性上的条件的布尔组合。支持相等、通配符匹配和近似相等(近似相等的精确定义因具体系统而异)。
- 一个范围，可以就是基，也可以是基和它的子基，或者是基以下的整棵子树。
- 要返回的属性。
- 结果和资源消耗的数量限制。

查询还可以指定是否自动取消引用别名；如果关闭了取消引用别名，则别名条目可以作为答案返回。

查询 LDAP 数据源的一种方式是使用 LDAP URL。LDAP URL 的例子是：

ldap:// //codex.cs.yale.edu/o = Yale University，c = USA

ldap:// //codex.cs.yale.edu/o = Yale University，c = USA?? sub? cn = Silberschatz

第一个 URL 返回具有组织机构为 Yale University 并且国家为 USA 的服务器上所有条目的所有属性。第二个 URL 在具有可区别名称 **o = Yale University，c = USA** 的结点的子树上执行搜索查询(选择)**cn = Silberschatz**。URL 中的问号分隔开不同的域。第一个域是可区别名称，这里是 **o = Yale University，c = USA**。第二个域是待返回的属性列表，这里为空，意味着返回所有属性。第三个属性(sub)指定待搜索的整棵子树。最后的参数是搜索条件。

查询 LDAP 目录的第二种方式是使用应用编程接口。在图 19-8 中显示了用于连接到一台 LDAP 服务器并在该服务器上运行查询的一段 C 代码。这段代码首先通过 ldap_open 和 ldap_bind 来打开与 LDAP 服务器的连接。然后通过 ldap_search_s 来执行查询。ldap_serearch_s 的参数包括 LDAP 连接句柄、搜索执行的基的 DN、搜索范围、搜索条件、待返回的属性列表和一个称为 attrsonly 的属性。该属性如果设置为 1，将导致只返回结果的模式，而没有实际元组。最后的参数是输出参数，它将查询结果作为 LDAPMessage 结构返回。

第一个 **for** 循环反复迭代并输出结果中的每个条目。注意一个条目可能有多个属性，第二个 **for** 循环输出每个属性。由于 LDAP 中的属性可以是多值的，因此第三个 **for** 循环输出属性的每个值。调用
873 **ldap_msgfree** 和 **ldap_value_free** 来释放由 LDAP 库分配的内存空间。图 19-8 没有显示用于处理错误条

件的代码。

LDAP API 也包含创建、更新和删除条目的函数，同时包含 DIT 上的其他操作。每个函数调用的行为就像一个单独的事务；LDAP 不支持多个更新的原子性。

```
#include <stdio.h>
#include <ldap.h>
main() {
    LDAP *ld;
    LDAPMessage *res, *entry;
    char *dn, *attr, *attrList[] = {"telephoneNumber", NULL};
    BerElement *ptr;
    int vals, i;
    ld = ldap_open("corvette.cs.yale.edu", LDAP_PORT);
    ldap_simple_bind(ld, "avi", "avi-passwd");
    ldap_search_s(ld, "c=Yale University, c=USA", LDAP_SCOPE_SUBTREE,
                  "cn=Silberschatz", attrList, /*attrsonly*/ 0, &res);
    printf("found %d entries", ldap_count_entries(ld, res));
    for (entry=ldap_first_entry(ld, res); entry != NULL;
                        entry = ldap_next_entry(ld, entry))
    {
        dn = ldap_get_dn(ld, entry);
        printf("dn: %s", dn);
        ldap_memfree(dn);
        for (attr = ldap_first_attribute(ld, entry, &ptr);
                    attr != NULL;
                    attr = ldap_next_attribute(ld, entry, ptr))
        {
            printf("%s: ", attr);
            vals = ldap_get_values(ld, entry, attr);
            for (i=0; vals[i] != NULL; i++)
                printf("%s, ", vals[i]);
            ldap_value_free(vals);
        }
    }
    ldap_msgfree(res);
    ldap_unbind(ld);
}
```

图 19-8 用 C 编写的 LDAP 代码示例

19.10.2.3 分布式目录树

有关一家组织机构的信息可以分成多个 DIT，其中每个存储关于某些条目的信息。DIT 的后缀 874
（suffix）是 **RDN = value** 对的序列，用于识别 DIT 存储了何种信息。这些对串接到其余的由从条目到根的遍历所产生的可区别名称上。例如，某个 DIT 的后缀可能是 **o = Lucent**、**c = USA**，而另一个的后缀可能是 **o = Lucent**、**c = India**。DIT 可以被组织机构和地理位置来分开。

DIT 中的结点可能包含对另一个 DIT 中另一个结点的引用（referral）：例如，在 **o = Lucent**、**c = USA** 下的组织机构部门贝尔实验室可以有它自己的 DIT。在这种情况下，对于 **o = Lucent**、**c = USA** 的 DIT 将有一个结点 **ou = Bell Labs**，表示指向贝尔实验室的 DIT 的引用。

引用是有助于把分布的目录集合组织成一个集成系统的关键构件。当服务器得到在 DIT 上的查询时，它可以返回一个引用给客户端，然后客户端在引用的 DIT 上发布查询。访问引用的 DIT 是透明的，用户无须知道过程。另一种方式是，服务器本身可以对引用的 DIT 发布查询并与本地计算结果一起返回查询结果。

LDAP 使用的分层命名机制有助于发现对一家组织机构的组成部分上的信息的控制。进而，引用机制有助于将一家组织机构中的所有目录集成到单个虚拟目录中。

组织机构经常选择通过地理（例如，一家组织机构可能为具有该组织的大机构的每个站点维护一个目录）或者通过组织结构（例如，每家组织单位，比如部门，维持它自己的目录）来分割信息，虽然这并不是 LDAP 的要求。

虽然复制不是当前 LDAP 版本 3 标准的一部分，但很多 LDAP 实现支持 DIT 的主从和多主副本。在

LDAP 中对复制的标准化工作正在进行中。

19.11 总结

- 分布式数据库系统由站点的集合构成，每个站点维护一个本地数据库系统。各个站点能够处理局部事务，这些事务访问的数据仅位于该单个站点上。此外，站点可以参与到全局事务的执行中，这些全局事务访问多个站点上的数据。全局事务的执行需要在站点之间进行通信。
- 分布式数据库可能是同构的，其中所有站点都有共同的模式和数据库系统代码，或者是异构的，其中模式和系统代码可能不同。
- 关于在分布式数据库中存储关系涉及几个问题，包括复制和分片。系统应尽量减小用户需要了解关系如何存储的程度。这是非常重要的。

875

- 分布式系统可能遭受与集中式系统相同类型的故障。但是，分布式环境中还有另一些需要处理的故障，包括站点故障、链路故障、消息丢失以及网络划分。在分布式故障恢复模式的设计中需要考虑每个这样的问题。
- 为了保证原子性，执行事务 T 的所有站点必须在执行的最终结果上取得一致。T 要么在所有站点上提交，要么在所有站点上中止。为了保证这一特性，T 的事务协调器必须执行一种提交协议。使用最广泛的提交协议是两阶段提交协议。
- 两阶段提交协议可能导致阻塞。在这种情况下，事务的命运必须等到故障站点（协调器）恢复后才能确定。为了减少阻塞的可能性，可以使用三阶段提交协议。
- 持久消息为分布式事务处理系统提供了一种可选模式。该模式将单个事务拆分成在不同数据库执行的多个部分。持久消息（无论是否发生故障，都保证正好只传送一次）被传送到需要采取动作的远程站点。虽然持久消息避免了阻塞问题，但是应用程序开发者必须编写代码来处理各种类型的故障。
- 在集中式系统中使用的各种并发控制方案修改后可用于分布式环境。
  - 就封锁协议而言，细微的唯一改变是锁管理器的实现方式。这里有各种不同的方式。可以采用一个或多个中央协调器。而如果采用分布式锁管理器方式，复制的数据就必须特殊对待。
  - 处理已复制数据的协议包括主副本协议、多数协议、有偏协议和决定人数同意协议。它们在开销方面和在发生故障时工作的能力方面各有不同的取舍权衡。
  - 就时间戳和有效性验证方案而言，所需的唯一修改是开发一种产生全局唯一性时间戳的机制。
  - 许多数据库系统支持延迟复制，其中更新被传播到执行更新的事务的范围之外的副本。这样的工具必须小心使用，因为它们可能导致不可串行化的执行。

876

- 分布式锁管理器环境中的死锁检测需要多个站点之间的合作，因为甚至在没有局部死锁的情况下也可能有全局死锁。
- 为了提供高可用性，分布式数据库必须检测故障，重构系统以使计算能够继续进行，并在处理器或链路修复之后能够恢复。由于要在网络划分和站点故障之间进行区分是很困难的，因此这个任务就变得非常复杂。

  通过使用版本号，可以对多数协议进行扩展使其在即使存在故障的情况下仍允许进行事务处理。虽然该协议代价昂贵，但它无论在何种类型的故障下都能工作。可以使用较小代价的协议来处理站点故障，但是它们假设不会发生网络划分。
- 一些分布式算法需要使用协调器。为了提供高可用性，系统必须维护一个准备好在协调器故障时能继续其职责的备份副本。另一种方法是在协调器发生故障后选出新的协调器。确定哪个站点应该作为协调器的算法称为选举算法。
- 分布式数据库上的查询可能需要访问多个站点。可以使用几种优化技术来识别需要访问的最佳站点集。查询可以依据关系的分片来自动重写，然后可以在每个分片的副本之间做出选择。可以应用半连接技术减少跨不同站点的关系（或相应的分片或副本）连接中所涉及的数据传输。
- 异构分布式数据库允许站点有它们自己的模式和数据库系统代码。多数据库系统提供了一种环境，在其中新的数据库应用可以访问位于多种异构软硬件环境的各个先前存在数据库中的数

据。局部数据库系统可以采用不同的逻辑模型以及数据定义和数据操纵语言，并且可以在它们的并发控制和事务管理机制上存在差别。多数据库系统虚拟了逻辑上的数据库集成，不需要物理上的数据库集成。

- 为了响应超大规模 Web 应用对数据存储的需求，近年来在云上构建了大量数据存储系统。这些数据存储系统允许扩展到地理上分布的数千个结点上，而且具有高可用性。然而，它们并不支持通常的 ACID 特性，而且在划分时以副本一致性为代价来获得可用性。当前的数据存储系统还不支持 SQL，而且只提供简单的 **put()/get()** 接口。即使对于传统数据库云计算也是有吸引力的，但因为缺乏对数据存放和地理副本的控制，也面临一些挑战。
- 目录系统可视为一种特殊形式的数据库，其中信息按照一种分层的方式组织，类似于文件系统中文件的组织方式。目录通过标准化目录访问协议（例如 LDAP）来访问。目录可以分布到多个站点上来提供对各个站点的自治。目录可以包含对其他目录的引用，这有助于建立集成视图，借此查询被发送给单个目录，并且在所有相关目录上透明地执行。 877

术语回顾

- 同构分布式数据库
- 异构分布式数据库
- 数据复制
- 主副本
- 数据分片
  - 水平分片
  - 垂直分片
- 数据透明性
  - 分片透明性
  - 复制透明性
  - 位置透明性
- 名字服务器
- 别名
- 分布式事务
  - 局部事务
  - 全局事务
- 事务管理器
- 事务协调器
- 系统故障模式
- 网络划分
- 提交协议
- 两阶段提交协议（2PC）
  - 就绪状态
  - 疑问事务
  - 阻塞问题
- 三阶段提交协议（3PC）
- 持久消息
- 并发控制
- 单锁管理器
- 分布式锁管理器
- 副本协议
  - 主副本
  - 多数协议
  - 有偏协议
  - 法定人数同意协议
- 时间戳
- 主从复制
- 多主（任意地方更新）复制
- 事务一致性快照
- 延迟传播
- 死锁处理
  - 局部等待图
  - 全局等待图
  - 假环
- 可用性
- 健壮性
  - 基于多数的方法
  - 读一个、写所有
  - 读一个、写所有可用
  - 站点重构
- 协调器选择
- 备份协调器
- 选举算法
- 威逼算法
- 分布式查询处理
- 半连接策略
- 多数据库系统
  - 自治
  - 中间件
  - 局部事务
  - 全局事务
  - 确保全局可串行化
  - 标签
- 云计算
- 云数据存储
- 表块
- 目录系统
- LDAP：轻量级目录访问协议
  - 可区别名称（DN）
  - 相对可区别名称（RDN）
  - 目录信息树（DIT）
- 分布式目录树
  - DIT 前缀
  - 引用

实践习题

19.1 为局域网设计的分布式数据库与为广域网设计的分布式数据库会有哪些区别？

19.2 要建立一个高可用性的分布式系统，你必须知道会发生什么类型的故障。

a. 列出分布式系统中可能的故障类型。

b. 在你的 a 小题列表中的哪些故障也可能出现在集中式系统中？

19.3 考虑在事务的2PC中发生的故障。对你在实践习题19.2a中列出的每种可能的故障，解释是否存在故障，2PC怎样保证事务的原子性。

19.4 考虑一个有两个站点A和B的分布式系统。站点A能区别出下列情况吗？

- B崩溃。
- A和B之间的连接断开。
- B强度过载，并且响应时间比正常情况下长100倍。

878
879

你的答案对于分布式系统中的恢复有何启示？

19.5 本章描述的持久消息模式依赖于时间戳，并结合了丢弃接收到的过于陈旧的消息。给出一个基于序列号而不是时间戳的替代方案。

19.6 给出读一次、写所有可用方法导致错误状态的一个例子。

19.7 解释在分布式系统中的数据复制和维护远程备份站点之间的差异。

19.8 给出一个即使更新时得到了主要的（主）副本上的排他锁，延迟复制仍能够导致数据库状态不一致的例子。

19.9 考虑下面的死锁检测算法：当站点S_i上的事务T_i请求站点S_j上某事务T_j的资源时，发送带时间戳n的请求消息。边(T_i, T_j, n)被插入到S_i的局部等待图中。仅当T_j已接收到请求消息但不能立即授予所请求的资源时，才把边(T_i, T_j, n)插入到S_j的局部等待图中。同一站点上从事务T_i到T_j的请求按照通常的方式处理；边(T_i, T_j)上不与时间戳相关联。中央协调器通过对系统中的每个站点发送初始化消息来调用检测算法。

一旦接收到这样的消息，站点就发送它自己的局部等待图给协调器。注意此图包含该站点所拥有的关于真实图状态的所有局部信息。等待图反映了站点的瞬间状态，但是它没有与任何其他站点同步。当控制者从每个站点都接收到回答后，它构造如下的图：

- 对于系统中的每个事务，该图包含一个顶点。
- 该图有边(T_i, T_j)当且仅当：
  - 在某个等待图中有边(T_i, T_j)。
  - 边(T_i, T_j, n)（对某些n）在不止一个等待图中出现。

证明，如果在构造的图中有环，那么系统处于死锁状态，并且如果在构造的图中没有环，那么系统在该算法开始执行时没有处于死锁状态。

19.10 考虑按*plant_number*水平分片的关系：

880

$$employee\ (name,\ address,\ salary,\ plant\_number)$$

假设每个分片有两个副本：一个存在New York站点，且另一个存在本地的工厂站点。为下述在San Jose站点提出的查询设计一种好的处理策略。

a. 找出Boca厂的所有员工。

b. 找出所有员工的平均工资。

c. 找出在以下每个站点薪酬最高的员工：Toronto、Edmonton、Vancouver、Montreal。

d. 找出公司中薪酬最低的员工。

19.11 对图19-9中的关系计算$r \times s$。

A	B	C
1	2	3
4	5	6
7	2	4
5	3	2
8	9	7

r

C	D	E
3	4	5
3	6	8
2	3	2
1	1	1
1	2	3

s

图19-9 实践习题19.11中的关系

19.12 给一个在云上的理想化的应用程序的例子，再给另一个很难在云上成功实现的例子，解释你的答案。

19.13　假设 LDAP 的功能可以在数据库系统之上实现。那么 LDAP 标准需要什么？

19.14　考虑一个多数据库系统，它保证在任何时候最多只有一个全局事务是活跃的，并且每个本地站点保证局部可串行化。

a. 给出多数据库系统可以保证在任何时候最多只有一个活跃的全局事务的方法。

b. 举例说明尽管有上述假设，还是有可能导致不可串行化的全局调度。

19.15　考虑一个多数据库系统，其中每个本地站点保证局部可串行化，并且所有全局事务都是只读的。

a. 举例说明在这样的系统中可能产生不可串行化的执行。 [881]

b. 说明你如何使用标签机制来保证全局可串行化。

习题

19.16　讨论集中式数据库和分布式数据库的相对优点。

19.17　解释下述说法有何区别：分片透明性、复制透明性、位置透明性。

19.18　数据复制和分片何时有用？解释你的答案。

19.19　解释透明性和自治的概念。为什么从人的因素的角度来看这些概念是需要的？

19.20　如果我们把第 15 章中多粒度协议的分布式版本应用于分布式数据库，负责 DAG 根结点的站点可能成为瓶颈。假设我们修改协议如下：

- 根结点上只允许有意向模式的锁。
- 所有事务被自动授予根结点上所有可能的意向模式锁。

说明这些修改可以无须允许任何非串行化调度来减轻这个问题。

19.21　研究并总结你正使用的数据库系统为处理由于更新的延迟传播而导致的不一致性状态而提供的工具。

19.22　讨论 19.5.2 节介绍的用来产生全局唯一性时间戳的两种方法的优缺点。

19.23　考虑关系：

$$employee(name, address, salary, plant\_number)$$
$$machine(machine\_number, type, plant\_number)$$

假设 *employee* 关系按 *plant_number* 水平分片，且每个片元本地存储在它所对应的工厂站点。假设整个 *machine* 关系存在 Armonk 站点。为以下每个查询设计一种好的处理策略。

a. 找出包含机器号 1130 的工厂的所有员工。

b. 找出包含机器类型为"milling machine"的工厂的所有员工。

c. 找出 Almaden 工厂的所有机器。 [882]

d. 找出 $employee \bowtie machine$。

19.24　对习题 19.23 的每种策略，说明你选择的策略如何依赖于下列因素：

a. 查询提出的站点。

b. 需要结果的站点。

19.25　表达式 $r_1 \bowtie r_2$ 必然等于 $r_2 \bowtie r_1$ 吗？在什么条件下 $r_1 \bowtie r_2 = r_2 \bowtie r_1$？

19.26　如果使用云数据存储服务来存储两个关系 r 和 s，而且我们需要对 r 和 s 进行连接，为什么将此连接作为物化视图来维护可能是有用的？在你的答案中，务必区分出"有用"的各种含义：整体吞吐量、空间使用效率，以及对用户查询的响应时间。

19.27　为什么云计算服务通过使用虚拟机的方式能最好地支持传统数据库系统，而不是通过直接在服务供应商的实际机器上运行的方式？

19.28　描述如何使用 LDAP 来提供对数据的多层分层视图，而不用复制基础级数据。

文献注解

教材中关于分布式数据库的讨论由 Ozsu 和 Valduriez[1999] 提供。Breitbart 等[1999b] 提供了对分

布式数据库的概述。

分布式数据库中事务概念的实现由 Gray[1981]和 Traiger 等[1982]给出。2PC 协议由 Lampson 和 Sturgis[1976]开发。三阶段提交协议来自 Skeen[1981]。Mohan 和 Lindsay[1983]讨论了 2PC 的两种修改版，可做假设提交和假设中止，它们通过考虑事务命运、定义默认假设来减少 2PC 的开销。

19.6.5 节中的威逼算法来自 Garcia-Molina[1982]。分布式时钟同步在 Lamport[1978]中讨论。分布式并发控制由 Bernstein 和 Goodman[1981]介绍。

R*的事务管理器在 Mohan 等[1986]中描述。分布式并发控制模式的验证技术由 Schlageter[1981]和 Bassiouni[1988]描述。

Gray 等[1996]在数据仓库环境中再讨论了复制数据的并发更新问题。Anderson 等[1998]讨论了关于延迟复制不一致性的问题。Breitbart 等[1999a]描述了用于处理复制的延迟更新协议。

各种数据库系统的用户手册提供了它们如何处理复制和一致性的详细信息。Huang 和 Garcia-Molina[2001]阐述了复制消息系统中恰好一次的语义。

Knapp[1987]综述了分布式死锁检测的文献。实践习题 19.9 来自 Stuart 等[1984]。

Epstein 等[1978]，Hevner 和 Yao[1979]讨论了分布式查询处理。Daniels 等[1982]讨论了 R*采用的分布式查询处理方法。

Ozcan 等[1997]阐述了多数据库中的动态查询优化。Adali 等[1996]和 Papakonstantinou 等[1996]描述了中间件系统中的查询优化问题。

Mehrotra 等[2001]讨论了多数据库系统中的事务处理。Georgakopoulos 等[1994]介绍了标签方案。Mehrotra 等[1991]介绍了 2LSR。

在 Ooi 和 S. Parthasarathy[2009]中有一组关于云系统上数据管理的论文。Chang 等[2008]描述了 Google 的 Bigtable 的实现，而 Cooper 等[2008]描述了雅虎的 PNUTS 系统。Brantner 等[2008]介绍了使用 Amazon 的 S3 基于云的存储来构建数据库的经验。Lomet 等[2009]介绍了在云系统中使事务正确工作的方法。Brewer[2000]推测了 CAP 定理，并由 Gilbert 和 Lynch[2002]形式化及证明。

883
884

Howes 等[1999]提供了涵盖 LDAP 的教材。

第六部分
Part 6

数据仓库、数据挖掘与信息检索

数据库查询通常设计为提取特定信息，例如账户的余额或者客户账户余额的总和。然而，为帮助制定公司策略而设计的查询通常需要访问来自多个数据源的大量数据。

数据仓库是从多个数据源中进行数据采集，并以一种兼容的、统一的数据库模式进行存储的数据仓储。存放在数据仓库中的数据将用于各种复杂聚集和统计分析，常常为数据分析而开发 SQL 结构，正如我们在第 5 章所见。此外，知识发现技术也用于尝试发现数据中的规则和模式。例如，零售商可能发现特定产品往往一起购买，就可以利用这样的信息来形成营销策略。这种从数据中发现知识的过程称为数据挖掘。第 20 章将阐述数据仓库和数据挖掘的问题。

在到目前为止的讨论中，我们集中在相对简单的、结构完整的数据上。然而，在互联网上在组织机构的内网上以及在个人用户的电脑上都存在大量非结构化的文本数据。用户希望从这些庞大的、大部分是文本的信息中用诸如关键字查询那样的简单查询机制来发现相关的信息。信息检索领域处理这类非结构化数据的查询，并特别关注查询结果的排序。尽管该领域的研究已有数十年历史，但随着万维网的发展它仍在飞速成长。第 21 章给出了对信息检索领域的介绍。

第20章
Database System Concepts, 6E

数据仓库与数据挖掘

企业已开始对不断增长的数据进行联机发掘，以对其商业交易活动做出更好的决策，比如采购何种货物以及如何最好地定位客户群以增加销售额。挖掘这些数据有两个方面的问题。第一个方面是把来自各个数据源的数据汇集到一个中心仓库中，这个中心仓库叫做数据仓库。与数据仓库相关的问题包括处理脏数据（即带有某些错误的数据）的技术，以及对大量数据的高效存储和索引的技术。

第二个方面是分析收集到的数据来发现可以成为商务决策基础的信息或者知识。某些类型的数据分析可以通过使用计算联机分析处理（OLAP）的SQL结构（我们已在5.6节（第5章）中见到过）来实现，还可以通过使用OLAP的工具和图形界面来实现。另一种从数据中获得知识的方法是使用数据挖掘（data mining），其目标是在大量数据中检测出各类类型的模式。数据挖掘是对具有类似目标的各种类型的统计技术的补充。

20.1 决策支持系统

数据库应用可广义地划分为事务处理和决策支持系统。事务处理系统是用来记录有关事务的信息的系统，比如公司的产品销售信息，或者大学的课程注册和成绩信息。事务处理系统现今已得到广泛应用，组织机构已经积累了由这类系统产生的大量信息。决策支持系统的目标是从事务处理系统存储的细节信息中提取出高层次的信息，并利用这些高层次信息来做出各种决策。决策支持系统帮助经理决定商店该采购什么产品，工厂该生产什么产品，或者大学该录取哪些申请者。

887

例如，公司数据库常常包含大量关于客户库交易的信息。所需的信息存储规模可能高达数百GB，对于大型零售连锁店甚至达到TB级。零售商的交易信息可能包括顾客姓名或标识（比如信用卡号）、购买的商品、支付的价钱以及购买日期。所购商品信息可能包括商品名称、生产商、型号、颜色和大小。顾客信息可能包括信贷历史、年龄、住址、年龄、其全数自背景

这种大型数据库可以作为制定商业务决策的信息宝库，譬如要采购什么商品或者提供多少折扣。例如，一家零售公司可能发现在太平洋西北部突然进行购买法兰绒衬衫，意识到了这是可能的趋势，就可能开始在该地区的商店储备大量的这种衬衫。作为另一个例子，一家汽车公司在查询其数据库时可能发现，其大多数跑车是由年薪超过50 000美元的年轻的女士买的。该公司可能调整其市场定位，以吸引更多这类妇女来购买它的小型跑车，并且可以避免把经费浪费在试图吸引其他人群来购买这种汽车上。在这两个例子中，公司识别出了客户行为的模式并利用该模式来制定商业务决策。

用于决策支持的数据存储和检索带来了几个问题：

- 尽管许多决策支持查询可以用SQL书写，但对一些要么无法用SQL表示，要么无法简单地用SQL来表示。为此已经推出了几种SQL扩展以使更易于进行数据分析。5.6节介绍了用于数据分析和联机分析处理（OLAP）技术的SQL扩展。
- 数据库查询语言并不适合对数据执行详细的统计分析（statistical analyse）。有几种软件包，例如SAS或S++，有助于统计分析。这些软件包与数据库通过界面连接，允许将大量数据存储到数据库中并且可以高效地检索以用于分析。统计分析领域本身是一个大的学科，更多信息可参见文献注解中的参考文献。
- 大型公司拥有形式多样的数据源用于制定商务决策，这些数据源可能按不同模式存储数据。甚

于性能因素(也基于管理控制因素)，数据源通常不允许公司其他部门检索所需数据。

为了能够在这些不同的数据上高效执行查询，一些公司已创建了数据仓库。数据仓库位于一个单独的节点上，使用统一的模式从多个数据源收集数据。因此，它们给用户提供一个单独的、统一的数据接口。20.2节研究创建和维护数据仓库的问题。

- 知识发现技术试图从数据中自动发现统计规则和模式。数据挖掘领域将人工智能研究者和统计分析家发明的知识发现技术结合起来，同时采用高效的实现技术使它们能够用于超大型数据库。20.3节将讨论数据挖掘。 888

决策支持(decision support)领域在广义上可以看作需要广上述所有领域，尽管有些人狭义地使用这个术语，排除了统计分析和数据挖掘。

20.2 数据仓库

大公司在许多地方都有分支，每个分支都可能产生大量数据。例如，大型零售连锁店拥有成百上千家商店，而保险公司可能拥有来自十几个各地分支机构的数据。而且，大型组织机构有复杂的内部组织结构，因此不同的数据可能位于不同的地点，或者在不同的操作系统中，或者具有不同的模式。例如，生产问题数据和顾客意见数据可能存储在不同的数据库系统中。组织机构经常从外部数据源购买数据(比如用于产品推广的邮件列表，或由信用机构提供的消费者的信用评分)，用于决定用户的信誉。[⊖]

企业决策者需要访问来自多个这种数据源的信息。在各个数据源上独立查询既麻烦又低效。而且，数据源可能只存储当前数据，而决策者可能还需要访问历史数据；例如，有关在去年里购买模式是如何改变的信息可能非常重要。数据仓库对这些问题提供了一种解决方案。

数据仓库(data warehouse)是一个将从多个数据源中收集来的信息以统一模式存储在单个站点上的仓储(或归档)。一旦收集完毕，数据会存储很长时间，允许访问历史数据。因此，数据仓库给用户提供了一个单独的、统一的数据接口，易于决策支持查询的书写。而且，通过从数据仓库单访问用于支持决策的信息，决策者可以保证在线的事务处理系统不受决策支持负载的影响。

20.2.1 数据仓库成分

图20-1显示了一个典型的数据仓库的架构，并阐明了数据收集、数据存储以及查询和对数据分析 889
的支持。下面是创建数据仓库时需要说明的一些问题：

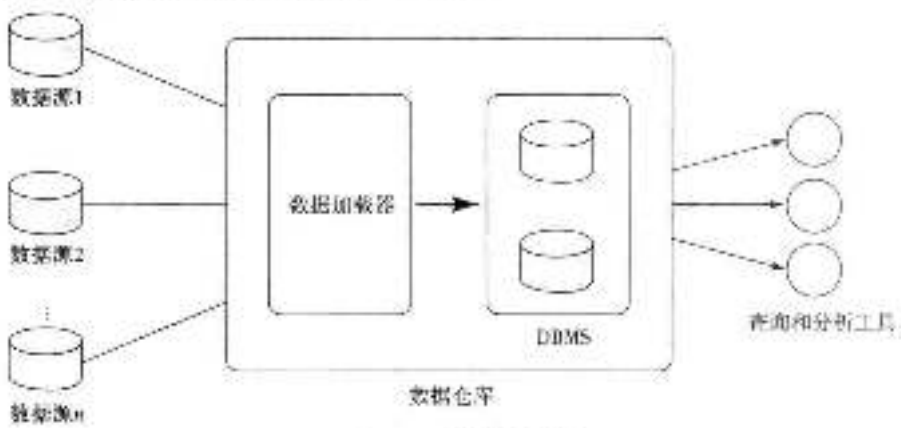


图20-1 数据仓库架构

- **何时和如何收集数据**。在收集数据的**源驱动架构**(source-driven architecture)中，数据源连续地(发生事务处理时)或周期性地(例如，每个晚上)传输新的信息。在**目标驱动架构**(destination-

⊖ 信用机构是一种公司，它从多个数据源收集消费者信息，并为每个消费者计算消费信用度。

driven architecture)中，数据仓库周期地给数据源发送需要新数据的请求。

除非数据源更新通过两阶段提交的方式在数据仓库中复制，否则数据仓库不会与数据源保持同步走款。两阶段提交代价十分昂贵，通常不作为一种可选方案，所以数据仓库通常会有稍微过期的数据。然而这对于决策支持系统通常不是问题。

- **使用何种模式**。单独构造的各个数据源很可能具有不同模式。事实上，它们甚至可能使用不同的数据模型。数据仓库的部分任务就是进行模式整合，并将数据转化成整合后的模式后再进行存储。其结果是，存储在数据仓库中的数据可看作是数据源数据的一个物化视图，而不单单是数据源数据的一个拷贝。
- **数据转换和清理**。对数据的纠正和预处理任务称作**数据清理**(data cleansing)。数据源经常传送大量具有略微不一致性的数据，这种不一致性是可以纠正的。例如，姓名经常拼错，地址中可能有街道、地区或者城市名称的拼写错误，或者邮编输入错误。这些可以通过参考有关每个城市的街道名和邮编的数据库来进行合理纠正。这种任务所需要的数据大致匹配称为**模糊查找**(fuzzy lookup)。

  从多个数据源收集的地址列表可能具有重复，需要在**合并－清除操作**(merge-purge operation)
890 中消除这些重复(这种操作也称作**去重**(deduplication))。一所住宅中多个人的记录可以组合为一组，这样每所住宅只须发送一封邮件，此操作称作**住宅操作**(householding)。

  数据除了要清理，可能还需要采用多种方式进行**转换**(transform)，例如改变度量单位，或者通过将来自多个源关系的数据进行连接从而将数据转换为一种不同的模式。数据仓库一般有图形工具来支持数据转换。这些工具允许将转换表示为框，通过在框之间连上边来表示数据的流动。条件框能把数据引至转换过程中合适的下一步。图 20-7 给出了一个例子，它使用微软 SQL Server 提供的图形工具来指定转换。
- **如何传播更新**。数据源中关系的更新必须传播到数据仓库。如果数据仓库中的关系与数据源中的完全一样，那么传播就直接了当了。如果不一致，更新传播问题基本上就是视图维护问题，此问题已在 13.5 节讨论过了。
- **汇总何种数据**。事务处理系统产生的原始数据可能量非常大，无法在线存储。然而，通过只维护由关系上的聚集得到的汇总数据，而不是维护整个关系，我们就可以回答许多查询。例如，我们可以存储按商品名和类别汇总的服装销售总额，而不是存储服装销售的所有数据。

假设关系 r 由汇总关系 s 取代。虽然可以允许用户提出查询请求，就像关系 r 是可以在线得到的一样。如果查询只需要汇总数据，则有可能利用 s 将其转化为一种等价的形式，见 13.5 节。

将数据存入数据仓库所涉及的不同步骤称为**抽取**(extract)、**转换**(transform)和**加载**(load)，或者称为 **ETL** 任务；抽取指的是从源收集数据，而加载指的是把数据装入数据仓库中。

20.2.2 数据仓库模式

典型的数据仓库具有为数据分析而设计的模式，使用诸如 OLAP 的工具。因此，数据通常是多维数据，具有维属性和度量属性。包含多维数据的表称作**事实表**(fact table)。它通常很大。一个记录零售商店销售信息的表就是事实表的一个典型例子，其中每个元组对应一个售出商品。*sales* 表的维可以包括何种商品(通常是商品标识，比如条形码中使用的标识)、商品售出日期、商品售出地点(商店)、哪个顾客购买该商品等。度量属性可能包括售出商品数量及商品价格。

为了最小化存储需求，维属性通常是一些短的标识，作为参照其他称作**维表**(dimension table)的表
891 的外码。例如，事实表 *sales* 可能有属性 *item_id*、*store_id*、*customer_id* 和 *date*，以及度量属性 *number* 和 *price*。属性 *store_id* 是参照维表 *store* 的外码，表 *store* 含有其他属性，比如商店位置(城市、州、国家)。*sales* 表的 *item_id* 属性是参照维表 *item_info* 的外码，表 *item_info* 还含有一些信息，比如商品名称、商品所属类别以及其他的商品细节信息，如颜色和尺寸。属性 *customer_id* 是参照 *customer* 表的外码，表 *customer* 含有诸如顾客姓名和地址那样的属性。我们还可以将 *date* 属性看作参照 *date_info* 表的外键，表 *date_info* 给出了每个日期的月、季和年的信息。

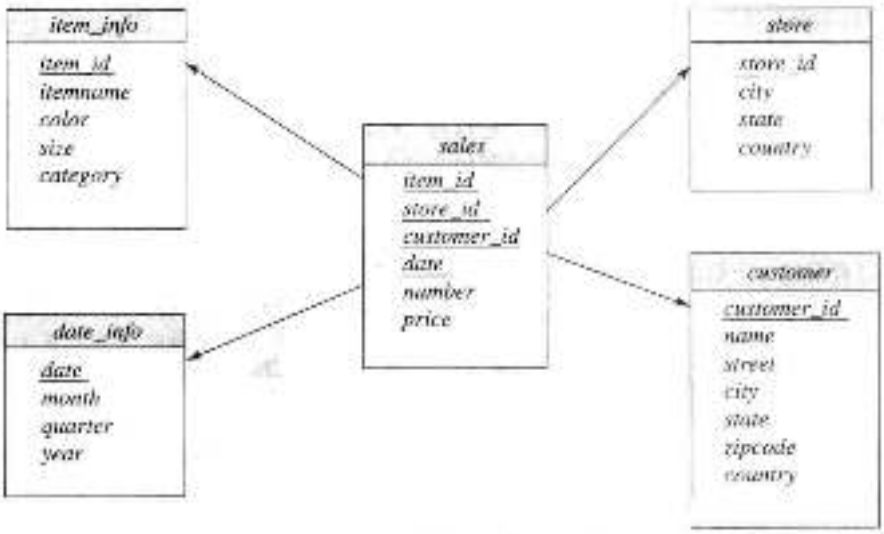


图 20-2　某个数据仓库的星形模式

图 20-2 给出了结果模式。这种具有一个事实表、多个维表以及从事实表到维表的外码的模式称为**星型模式**(star schema)。更复杂的数据仓库设计可能含有多级维表；例如，*item_info* 表可能有 *manufacturer_id* 属性，它是参照给出厂商细节信息的另一个表的外码。这种模式称作雪花模式(snowflake schema)。复杂数据仓库设计也可能有不止一个事实表。

20.2.3　面向列的存储

传统数据库把元组的所有属性存储在一起，再把元组存放在文件中。这种存储方式称为面向行的存储(*row-oriented storage*)。相反，**在面向列的存储**(column-oriented storage)中，关系的每个属性存储在单独的文件中，来自相邻元组的值存放在文件中连续的位置上。假设数据类型的大小固定，一个关系的第 i 个元组在属性 A 上的取值可以通过访问与属性 A 对应的文件并读取偏移量为 $(i-1)$ 倍属性 A 的 [892]
值大小(以字节为单位)处的值而得到。

面向列的存储和面向行的存储对比，至少有两个主要优势：

1. 当查询只需要访问一个包含大量属性的关系中的几个属性时，其余属性并不需要从磁盘提取到内存中。相反，在面向行的存储中，如果一些属性和查询所用的属性相邻存储，那么不仅要把这些不相干的属性提取到内存中，而且它们还可能被预取到处理器高速缓存中，浪费了缓存空间和内存的带宽。

2. 把相同类型的值存储在一起提高了压缩效率；压缩不仅可以大大降低磁盘存储的成本，而且减少了从磁盘检索数据的时间。

另一方面，面向列的存储的缺点是存储或读取单个元组需要多次 I/O 操作。

根据上述折中的结果，面向列的存储没有广泛用于事务处理应用。然而，面向列的存储在数据仓库应用中获得了越来越多的认可，因为数据仓库应用很少访问单个元组，而是需要扫描和聚合多个元组。

Sybase IQ 是早期使用面向列存储的产品之一，但现在有几个研究项目和公司已经开发了基于面向列存储系统的数据库。这些系统已证明能够显著增加许多数据仓库应用的性能。关于如何实现面向列的存储以及在这种存储方式上如何优化和处理查询，请参考文献注解中的文献。

20.3　数据挖掘

术语**数据挖掘**(data mining)泛指半自动地分析大型数据库以发现有用模式的处理过程。类似于人工智能里的知识发现(也叫机器学习)或统计分析，数据挖掘试图从数据中发现规则和模式。然而，数据挖掘与机器学习和统计的不同在于，它处理主要存储在磁盘上的大量数据。也就是说，数据挖掘处

是"数据库中的知识发现"。

从数据库中发现的某些类型的知识可以用一个**规则**(rule)集来表示。下面是一个非正式地描述的规则的例子："年收入超过 50 000 美元的年轻妇女是最有可能购买小型跑车的人。"当然，这样的规则并不总是正确的，就像我们将要看到的那样，它具有"支持度"和"置信度"。其他类型的知识可用将不
893 同变量相互联系起来的等式来表达，或者利用其他一些在已知某些变量值的条件下预测结果的机制来表达。

可能有用的模式的可能类型是多样的，找到不同类型的模式可采用不同的技术。我们将研究一些模式的例子，看看从一个数据库中如何自动导出它们。

数据挖掘通常具有人工的成分，包括预处理数据使之成为算法可接受的形式，以及将发现的模式进行后期处理，以找出其中有用的异常模式。从给定数据库中发现的模式可能不止一种类型，可能需要人工交互以选出有用的模式类型。由于这个原因，数据挖掘在现实生活中确实是一种半自动的处理过程。然而，在描述中，我们重点集中在挖掘的自动化方面。

所发现的知识有大量应用。最广泛的应用是那些需要某种形式的**预测**(prediction)的应用。例如，当一个人申请信用卡时，信用卡公司要预测这个人是否有好的信用风险。预测基于这个人的已知属性，比如年龄、收入、债务以及过去的债务偿还历史。做出预测的规则来源于过去和现在的信用卡持有人的相同属性以及他们所表现的行为，比如他们是否未履行信用卡付款责任。其他类型的预测包括：预测哪些客户可能转到竞争者那里去(可以给这些客户提供特别的折扣吸引他们不离开)，预测哪些人有可能对推销邮件("垃圾邮件")做出反应，或者预测使用哪种方式的电话卡可能会欺诈。

另一类应用是寻找**关联**(association)，比如可能会一起购买的书籍。如果某位顾客购买了一本书，在线书店可以建议购买其他的相关书籍。如果某人购买了一架照相机，系统可以建议可能需要同照相机一起购买的附件。好的销售人员知道这样的模式并利用它们来进行额外的销售。挑战在于这个过程的自动化。其他类型的关联可能导致发现因果关系。例如，发现某种新引进的药与心脏病之间的意外联系导致发现这种药可能会在某些人群中引发心脏病，然后将这种药从市场上撤出。

关联是**描述性模式**(descriptive pattern)的一个例子；**聚类**(cluster)是这种模式的另一个例子。例如，一百多年前在一口水井周围发现了一连串的伤寒病症，这使人们发现水井中的水受到了污染并正在传播伤寒杆菌。疾病聚类检测甚至在现在也一直保持其重要性。

20.4 分类

894 如 20.3 节所提到的，预测是数据挖掘中最重要的类型之一。我们介绍分类，学习用于创建一种称作决策树分类器的技术，然后研究其他一些预测技术。

抽象地说，**分类**(classification)问题是这样的：给出属于某几个类之一的项，并给出项的过去实例(叫做**训练实例**(training instance))以及它们所属的类，问题是预测一个新项所属的类。因为新实例的类是未知的，所以必须使用该实例的其他属性来预测它所属的类。

通过寻找能够把给定数据分成可不相交的组的规则，可以实现分类。例如，假设信用卡公司要决定是否给一个申请者发放信用卡。该公司有关于这个人的各种信息，如她的年龄、教育背景、年收入和当前债务情况，这些信息可用来做出决策。

这些信息中有些与申请者的信誉程度有关，而有些可能无关。为了做出决策，公司根据每个客户的付款历史给当前客户样本集中的每个客户指定一个优、良、中或差的信誉等级。然后，公司尝试找出规则，根据有关个人的信息而不是根据实际的付款历史(对于新客户无法得到这类信息)，将它的当前客户分成优、良、中或差。让我们只考虑两个属性：教育水平(拥有的最高学历)和收入。规则可能是下面的形式：

$$\forall person\ P,\ P.\ degree = masters\ \textbf{and}\ P.\ income > 75,000 \Rightarrow P.\ credit = excellent$$

$$\forall person\ P,\ P.\ degree = bachelors\ \textbf{or}\ (P.\ income \geqslant 25,000\ \textbf{and}\ P.\ income \leqslant 75,000) \Rightarrow P.\ credit = good$$

对于其他信誉等级(中和差)也可用类似的规则来表示。

创建分类器的过程开始于数据样本，称作**训练集**(training set)。对训练集中的每个元组，其所属的类是已知的。例如，信用卡应用的训练集可以是已有的客户，他们的信誉程度已根据其付款历史得到。实际的数据或人群可能由所有人组成，包括那些不存在的客户。我们将看到，创建一个分类器有多种的方式。

20.4.1　决策树分类器

决策树分类器是一种广泛使用的分类技术。顾名思义，**决策树分类器**(decision tree classifier)使用一棵树：每个叶结点有一个相关联的类，每个内部结点有一个与其关联的谓词(或者更一般地说是一个函数)。图 20-3 显示了决策树的一个例子。

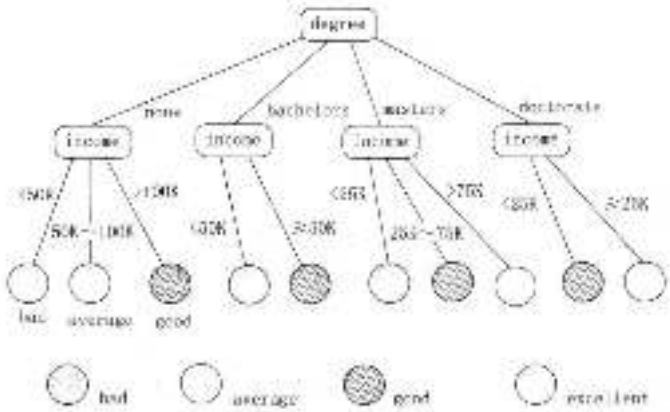


图 20-3　分类树

为了对一个新实例进行分类，我们从根开始遍历这棵树直至到达叶结点：在内部结点上使用该实例数据计算谓词(或函数)，从而找出应该走到哪个子结点。这个过程一直持续到我们到达叶结点为止。例如，如果一个人的学历水平是硕士且其收入是 40K，我们从根开始沿着标记为"masters"的边走，通过标记为"25K – 75K"的边到达叶结点。该叶结点的类是"good"，因此我们预测这个人的信用风险等级是良。

20.4.1.1　构造决策树分类器

接下来的问题是在给定训练实例集后如何构造决策树分类器。构造决策树分类器最通用的方法是使用**贪心**(greedy)算法，它从根开始递归地向下构造树。初始时只有一个结点及根结点，所有训练实例都与该结点关联。

在每个结点上，如果所有或者"几乎所有"与该结点关联的训练实例属于同一个类，则该结点成为与该类相关联的叶结点；否则，必须选择一个**划分属性**(partitioning attribute)和**划分条件**(partitioning condition)用来产生子结点。与每个子结点相关联的数据是满足该子结点划分条件的训练实例的集合。在例子中，选择了 *degree* 属性，构造了 4 个子结点，每个子结点代表一个学历值。4 个子结点的条件分别是 *degree* = none，*degree* = bachelors，*degree* = masters 和 *degree* = doctorate。与每个子结点关联的数据由满足与该子结点相关联的条件的训练实例构成。在对应于硕士(master)的那个结点上，选择了属性 *income*，其值的范围被分成 0 – 25K，25K – 50K，50K – 75K 和 75K 以上这些区间。与每个结点关联的数据由这样的训练实例构成：它们在 *degree* 属性上取值为硕士且在 *income* 属性上取值分别属于这些范围之一。由于在 *degree* = masters 结点下，范围 25K – 50K 和范围 50K – 75K 对应的类相同，作为优化，可以将两个范围合并成单个范围 25K – 75K。

895

896

20.4.1.2　最优划分

直观地，通过选择一个划分属性序列，我们从"不纯的"所有训练实例的集合开始，到得到"纯的"叶结点为止。在此，"不纯的"意指它包含来自许多类的实例，"纯的"意指在每个叶结点上所有训练实例只属于一个类。稍后我们将看到如何定量地度量纯度。为了判断在一个结点上选择特定属性和条件来划分数据的优劣，我们测量利用该属性进行划分所得到的子结点上数据的纯度，选出能得到最大纯

度的属性和条件。

有几种方法可以定量地测定一个训练实例集 S 的纯度。假设有 k 个类，S 的实例中属于类 i 的实例所占比例为 p_i。有一种称作**吉尼度量**(Gini measure)的纯度的测定方法，定义为：

$$\text{Gini}(S) = 1 - \sum_{i=1}^{k} p_i^2$$

当所有实例属于单个类时，Gini 值为 0，在每个类有相同实例数目的情况下它达到其最大值 $(1 - 1/k)$。另一种纯度测定方法是**熵度量**(entropy measure)，其定义为：

$$\text{Entropy}(S) = -\sum_{i=1}^{k} p_i \log_2 p_i$$

当所有实例属于单个类时，Entropy 值为 0，在每个类有相同实例数目的情况下它达到其最大值。熵度量来源于信息论。

如果一个集合 S 划分成多个集合 S_i，$i = 1, 2, \cdots, r$，我们可以按下述方法测定该集合的结果集的纯度：

$$\text{Purity}(S_1, S_2, \cdots, S_r) = \sum_{i=1}^{r} \frac{|S_i|}{|S|} purity(S_i)$$

即，纯度为所有集合 S_i 的纯度的加权平均值。上述公式可以与纯度的吉尼度量和熵度量一起使用。

这样，根据将 S 划分成 S_i，$i = 1, 2, \cdots, r$ 的特定划分所得到的**信息增益**(information gain)为：

897

$$\text{Information\_gain}(S, \{S_1, S_2, \cdots, S_r\}) = \text{purity}(S) - \text{purity}(S_1, S_2, \cdots, S_r)$$

划分成少数几个集合比划分成许多集合更可取，因为前者能产生更简单、更有意义的决策树。每个集合 S_i 中的元素数量也可能要考虑；否则，虽然划分对于几乎所有的元素都是一样的，但集合 S_i 能否有 0 个元素或 1 个元素将在所分集合的数量上产生很大差异。一个特定划分的**信息内容**(information content)可用熵的形式定义为：

$$\text{Information\_content}(S, \{S_1, S_2, \cdots, S_r\}) = -\sum_{i=1}^{r} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

综上可得出一个定义：对某个属性的**最优划分**(best split)是能够给出**最大信息增益率**(information gain ratio)的划分，定义为：

$$\frac{\text{Information\_gain}(S, \{S_1, S_2, \cdots, S_r\})}{\text{Information\_content}(S, \{S_1, S_2, \cdots, S_r\})}$$

20.4.1.3 寻找最优划分

我们如何找到某个属性的最优划分？如何划分一个依赖于该属性类型的属性。属性既可以是**连续取值的**(continuous valued)，即这些值可以按照对分类有意义的方式排序，比如说年龄或收入；也可以是**类别的**(categorical)，即它们没有有意义的顺序，比如部门名称或者国家名称。我们不能指望对部门名或国名的排序能对分类产生任何意义。

通常数字(整数/实数)属性被看作是连续取值的而字符串属性被当作类别的，但这也可能由系统用户来控制。在例子中，我们将属性 *degree* 当作类别的，并将属性 *income* 当作连续取值的。

我们首先考虑如何找到连续取值属性的最优划分。为简单起见，我们将只考虑连续取值属性的**二次分割**(binary split)，即划分的结果是得到两个子结点。**多路分割**(multiway split)的情况更为复杂，有关这个主题请参见文献注解中的文献。

为了找到连续取值属性的最优二次分划，我们首先将训练实例中的属性值排序，然后计算每个值上所作划分的信息收益。例如，如果训练实例在某个属性上的取值为 1、10、15 和 25，所考虑的切分点就是 1、10 和 15。在每种情况下，小于或等于该切分点的值组成一个划分，其余值构成另一个划分。该属性上的最优二次分划是能得到最大信息增益的划分。

对于类别属性，我们可以采用多路分划，每个属性值对应一个子结点。如果类别属性只有少数几个可区分的值，比如学历或性别，这种划分的效果会很好。但是，如果属性有许多可区分的值，比如大公司中的部门名称，为每个值创建一个子结点并不是一个好主意。在这种情况下，我们将试图将多

898 个值合并为一个子结点，以产生更少数量的子结点。关于如何实现请参阅文献注解中的文献。

20.4.1.4　决策树构造算法

构造决策树的主要思路是计算不同属性和不同划分条件，选择出能够产生最大信息增益率的属性和划分条件。同样的过程递归作用在每个划分结果集上，从而递归地构造出一棵决策树。如果数据可很好地分类，则递归过程在集合纯度为 0 时结束。然而，数据通常含有噪声，或者某个集合可能非常小以至于将它进一步划分可能从统计角度看来并不恰当。在这种情况下，递归过程在集合纯度"足够高"时终止，所产生叶结点的类定义为该集合的大多数元素所属的类。通常，树的不同分枝可能生长到不同的层数。

图 20-4 给出了一个递归构造决策树过程的伪码。它将训练实例集 S 作为参数。当集合有足够的纯度，或者集合 S 非常小以至于对其做进一步划分已经没有统计意义时，递归过程终止。参数 δ_1 和 δ_2 定义了纯度和集合大小的截止值，系统可以给出它们的默认值，但用户给的值可以覆盖系统默认值。

```
procedure GrowTree(S)
    Partition(S);

procedure Partition(S)
    if (purity(S) > δ1 or |S| < δ2) then
        return;
对于每个属性 A
    计算基于属性 A 的划分;
使用找到的最好划分(根据所有属性得到)把 S 划分成 S1, S2, ···, Sr
for i = 1, 2, ···, r
    Partition(Si);
```

图 20-4　决策树的递归构造

决策树的构造算法多种多样，我们只概述了其中几个的显著特征，有关细节见文献注解。对非常大型的数据集，因为划分需要进行重复拷贝，所以代价可能会很昂贵。因此开发了几种算法用于在训练数据比可用内存更大的情况下来最小化 I/O 和计算代价。

一些算法还对所生成决策树的子树进行修剪以减小过度适应(overfitting)：如果一棵子树高度适应于特定的训练数据以至于对其他数据产生许多分类错误，则称它是过度适应的。子树的修剪通过把它替换成叶结点来实现。有不同的启发式修剪方法：一种启发式方法使用部分训练数据来构造树。另一部分训练数据用于树的测试。当它发现如果一棵子树用叶结点代替会减少测试实例上的分类错误时，则会修剪那棵子树。

899

如果需要的话，我们可以从一棵决策树产生分类规则。我们对每个叶结点产生如下规则：左边是通往某个叶结点的路径上所有划分条件的合取，所属的类是在该叶结点上大多数训练实例所属的类。一个这样的分类规则的例子是：

$$degree = masters \textbf{ and } income > 75000 \Rightarrow excellent$$

20.4.2　其他类型的分类器

除决策树分类器以外还有几种其他类型的分类器。三种非常有用的分类器分别是神经网络分类器(*neural net classifier*)、贝叶斯分类器(*Bayesian classifier*)和支持向量机(*Support Vector Machine*)分类器。神经网络分类器使用训练数据训练人工神经网络。关于神经网络有大量的文献，我们在此不对它做深入考虑。

贝叶斯分类器(Bayesian classifier)在训练数据中寻找每个类的属性值分布；当给定一个新实例 d 时，对于每个类 c_j，使用分布信息估计实例 d 属于类 c_j 的概率，记为 $p(c_j \mid d)$。所用的方式下面将简要介绍。具有最大概率的类成为实例 d 预测所属的类。

为了找到实例 d 属于类 c_j 的概率 $p(c_j \mid d)$，贝叶斯分类器使用如下所示的贝叶斯定理(Bayes' theorem)：

$$p(c_j \mid d) = \frac{p(d \mid c_j)p(c_j)}{p(d)}$$

其中 $p(d \mid c_j)$ 是给定类 c_j 的前提下产生实例 d 的概率，$p(c_j)$ 是类 c_j 出现的概率，$p(d)$ 是实例 d 出现的概率。其中，由于 $p(d)$ 对所有类来说都是一样的，因此可忽略。$p(c_j)$ 可简化为属于类 c_j 的训练实例所占的比例。

例如，让我们考虑只有一个属性 *income* 用于分类的特殊情况。并假设我们需要分类的这个人的年收入是 76 000 美元。我们假设可以把收入值划分到几个桶中，并假定包含 76 000 的桶所包含收入值的范围是(75 000，80 000)美元。假设在优类的实例中，收入在(75 000，80 000)美元的概率为 0.1，而在良类实例中，收入在(75 000，80 000)美元的概率为 0.05。再假设共有 0.1 比例的人分类为优，有 0.3 比例的人分类为良。那么，对于优类的 $p(d \mid c_j)p(c_j)$ 的值是 0.01，而对于良类，它的值是 0.015。因此这个人就会划分到良类中。

一般情况下，分类需要考虑多个属性。因而，要找到 $p(d \mid c_j)$ 的精确值是很困难的。因为它需要从用于分类的属性值的所有组合中找到实例 c_j 的分布。这种组合(比如包含学位值和其他属性的收入桶)的个数可能非常大。利用有限的训练集来找这种分布，对于大多数组合来说，甚至没有一个训练集
900 与它们匹配，从而导致不正确的分类决策。为了避免这个问题，并简化分类任务，朴素贝叶斯分类器(naive Bayesian classifier)假设属性是独立分布的，从而估计：

$$p(d \mid c_j) = p(d_1 \mid c_j) * p(d_2 \mid c_j) * \cdots * p(d_n \mid c_j)$$

也就是说，给定所属类 c_j，实例 d 出现的概率是 d 的每个属性值 d_i 出现概率的乘积。

对每个类 c_j，概率 $p(d_i \mid c_j)$ 根据每个属性 i 的值的分布导出。该分布通过属于每个类 c_j 的训练实例来计算；分布通常用直方图来估计。例如，我们可以将属性 i 的值域分成相等的间隔，并存储落在每个间隔内的类 c_j 的实例所占的比例。给定属性 i 的值 d_i，$p(d_i \mid c_j)$ 的值就简化为落在 d_i 所属间隔中且属于类 c_j 的实例所占的比例。

贝叶斯分类器的一个显著优势是，它可以对具有未知或空属性值的实例分类，在概率计算中正好忽略了未知或空属性值。比较而言，决策树分类器无法处理这样的情况；某个实例在用于更进一步向下遍历决策树的划分属性上取空值。

支持向量机(Support Vector Machine，SVM)是一类在大范围应用中给出非常精确的分类的分类器。这里介绍有关支持向量机分类器的一些基本的、直观的知识。更深入的信息请参阅文献注解中的文献。

支持向量机分类器最好以几何方式来解释。在最简单的情况下，考虑在一个二维平面上的点集，有些点属于类 A，有些点属于类 B，我们给定一些分类类别(A 或者 B)已知的点作为训练集，我们需要用这些训练点构建一个点分类器。这种情况如图 20-5 所示，其中属于类 A 的点用 × 标记，而属于类 B 的点用○标记。

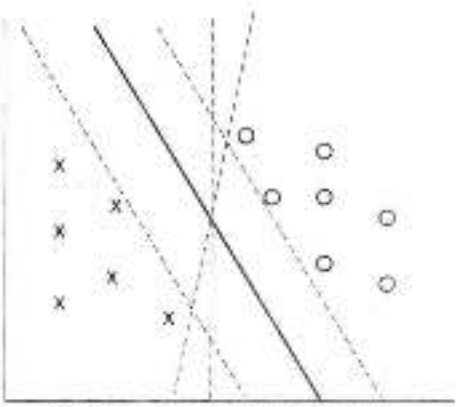


图 20-5 支持向量机分类器示例

假设我们可以在平面上画一条直线，使得属于类 A 的所有点都位于一边，且属于类 B 的所有点都位于另一边。那么，这条直线就可以用于给新的点进行分类，这些新的点的分类我们事先并不知道。但可能会存在很多这样的线，它们能够把类 A 中的点和类 B 中的点区分开来。图 20-5 给出了几条这样的线。支持向量机分类器选择这样的线，它到每个类(来自训练数据集中的点)中最近点的距离都最大。然后用这条线(称作**最高限界线**(maximum margin line))把其他点分到类 A 或 B 中去，这取决于这些点位于这条线的哪一边。在图 20-5 中，最高限界线用粗体标出，而其他线用虚线画出。

上述知识可以泛化到多于二维的场景，允许多个属性用于分类；在这种情况下，分类器会找一个划分平面，而不是一条线。然后，首先使用一种称作**核函数**(*kernel function*)的特定函数对输入点进行转换，支持向量机分类器就可以找到对点集进行划分的非

线性曲线。这在不能用线或平面来分离点的情况下是很重要的。由于噪声的存在，一个类中的某些点可能位于其他类的点的中间。在这种情况下，可能不存在任何线或者有意义的曲线来分离这两个类中的点；然后，会选择把这些点最准确地分到两个类中的线或曲线。

虽然支持向量机的基本表示是用于二分类器，即，只对两个类进行分类，但是它们也可以用于对多个类进行分类，如下所示：如果存在 N 个类，我们建立 N 个分类器，分类器 i 执行二分类，它把一个点要么分到 i 类，要么分到不属于 i 类。给定一个点，每个分类器 i 还输出一个值，表示给定点和类 i 的关联程度。然后我们在给定点上应用所有 N 个分类器，并选择出关联度值最高的类。

20.4.3　回归

回归(regression)处理的是值的预测，而不是类的预测。给定一个变量集的值 X_1，X_2，…，X_n，我们希望预测变量 Y 的值。例如，我们可以将教育程度当作一个数值，收入当作另一个数值，并在这两个变量的基础上，我们希望预测不履约的可能性，它可能是不履约发生机会的百分比，或是涉及不履约的数量。

一种方法是推断系数 a_0，a_1，a_2，…，a_n，使得

$$Y = a_0 + a_1 * X_1 + a_2 * X_2 + \cdots + a_n * X_n$$

找出这样的线性多项式称作**线性回归**(linear regression)。一般来说，我们期望找到一条适合数据的曲线(用多项式或其他形式定义)，所以该过程又称作**曲线拟合**(curve fitting)。 901 902

由于数据中的噪声或由于该关系不完全是多项式的，这种拟合可能只是一种近似，因此回归旨在找到能给出最大可能拟合的系数。在统计学里有求解回归系数的标准技术。我们在此不讨论那些技术，但是在文献注释中提供有关参考文献。

20.4.4　分类器验证

验证分类器是很重要的，也就是说，在决定把分类器用于某个应用之前，先测试它的分类错误率。考虑一个分类问题的例子：分类器需要根据某些输入(这里准确的输入是不重要的)来预测一个人是否患有一种特殊疾病 X。正预测表示这个人患有该疾病，而负预测表示这个人不患有该疾病。(术语正/负预测可用于任何二分类问题，而不只是疾病分类。)

测试用例集的已知结果(在例子中，是已知某人是否真的患有该疾病的病例)用来衡量分类器的质量(也就是错误率)。预测为正并且该人的确患有该病的情况叫**真命中**(true positive)，而预测为正但该人没有患此病的情况叫**假命中**(false positive)。类似地可以定义预测为负的情况下的**真舍弃**(true negative)和**假舍弃**(false negative)。

给定一个测试用例集，设 t_pos、f_pos、t_neg 和 f_neg 分别表示所产生的真命中、假命中、真舍弃和假舍弃的数量。设 pos 和 neg 表示真值和假值的实际数量(显然有 $pos = t\_pos + f\_pos$ 和 $neg = t\_neg + f\_neg$ 成立)。

分类质量可以通过多种不同方式来度量：

1. **正确性**(accuracy)，定义为 $(t\_pos + t\_neg)/(pos + neg)$，也就是分类器给出正确分类的次数所占比例。

2. **召回率**(recall)(也称为**灵敏度**(sensitivity))定义为 t_pos/pos，也就是分为正值的情况有多少实际上是真的。

3. **准确率**(precision)，定义为 $t\_pos/(t\_pos + f\_pos)$，即有多少预测为正的情况是正确的。

4. **特异性**(specificity)，定义为 t_neg/neg。

在特定应用中应该采用何种度量方式取决于该应用的需求。例如，高召回率对于筛选试验很重要，筛选试验后接受精确的测试，以致不会漏掉患有疾病的患者。相反，一个研究者希望找到几个患有疾病的实际病人以作进一步的跟踪，但不对找出所有病人感兴趣，就可能更看重准确率而不是召回率。不同分类器可能适用于每个这样的应用。习题 20.5 将对此类问题进行进一步探讨。 903

一个结果已知的测试用例集，既可用于分类器训练，也可用于分类器质量的度量。把同样的测试用例集既用作训练又用作分类器质量度量并不好，因为在训练过程中，分类器已经知道测试用例的正

确分类，这可能会导致人为质量somewhat的高质量。因此一个分类器的质量度量必须建立在它训练过程中没有遇见过的测试用例上。

因此，把可用的测试用例的一个子集用于训练，把不相交的另一个子集用作验证。在**交叉验证**(cross validation)中，可用的测试用例分成k个部分，编号为$1 \sim k$，譬如按照如下方式创建k个不同的测试集：在使用其他$k-1$部分来训练分类器之后，测试集i使用第i部分作为验证。在计算质量度量之前，把来自所有k个测试集的结果(t_pos、f_pos等)加起来。交叉验证比仅仅把数据划分为单个训练集和单个测试集的方式提供了更准确的度量。

20.5 关联规则

零售商店经常会对人们购买的不同商品之间的**关联**(association)感兴趣。有关这种关联的例子是：

- 买面包的人很有可能也买牛奶。
- 购买《数据库系统概念》(Database System Concepts)这本书的人很有可能也买《操作系统概念》(Operating System Concepts)这本书。

关联信息可以有多种使用方式。当一个顾客购买了某本书，在线书店可以建议相关的书籍。杂货店可以决定将面包放在靠近牛奶的地方，便于购物者更快地完成其购物任务，因为它们通常是一起购买的。或者商店可以将它们放在货架一行的两头，并将其他相关商品放在中间以吸引购物者在从这一行商品的一端走到另一端时也购买这些商品。商店对一种商品打折，可能不会对另一种与之关联的商品打折，因为顾客不管怎样都很可能购买这种商品。

关联规则的一个例子是：

$$bread \Rightarrow milk$$

在杂货店交易的上下文中，该规则说明购买面包的顾客也倾向于购买牛奶的概率很高。一条关联规则一定有一个相关的**个体总数**(population)：该个体总数由一个**实例**(instance)集构成。在杂货店的
[904] 例子中，个体总数可能包含所有的杂货店购买行为，每次购买行为为一个实例。在书店的例子中，个体总数可能包含所有购过书的人，不管他们是何时购的，每个顾客就是一个实例。在书店的例子中，分析人员决定什么时候购书是不重要的；然而对于杂货店的例子来说，分析人员可以决定关注单次购买行为，忽略同一个顾客多次光顾的情况。

规则有相关的支持度和相关的置信度。它们是在个体总数的上下文中定义的：

- **支持度**(support)度量的是同时满足规则前提和结论的个体总数所占的比例。

  例如，假设包括牛奶和螺丝刀的购买行为仅占所有购买行为的0.001，则规则

  $$milk \Rightarrow screwdrivers$$

  的支持度很低。该规则甚至可能没有什么统计学意义——也许仅有一次购买行为既包括牛奶又包括螺丝刀。商业贸易通常对具有低支持度的规则不感兴趣，因为那些规则只有少数顾客参与，不值得考虑。

  另一方面，如果所有购买行为的50%包含牛奶和面包，则涉及面包和牛奶(没有其他商品)的规则的支持度相对较高，这样的规则可能是值得注意的。至于到底多少才是值得考虑的最小支持度取决于应用。

- **置信度**(confidence)度量的是当前提为真时结论为真的频率。例如，如果在包括面包的购物行为中有80%又包括了牛奶，则规则：

  $$bread \Rightarrow milk$$

  的置信度为80%。具有较低置信度的规则是没有意义的。在商业应用中，规则的置信度通常比100%小很多，而在其他领域，例如在物理学中，规则可能具有高的置信度。

  注意，虽然$bread \Rightarrow milk$与$milk \Rightarrow bread$具有相同的支持度，但它们的置信度可能会有很大差异。

为了发现形如：

$$i_1, i_2, \cdots, i_n \Rightarrow i_x$$

的关联规则，我们首先找到具有足够支持度的项集，称作大项集(large itemset)。在例子中，我们查找包含在足够多的实例中的项集。稍后我们将看到如何计算大项集。

对每个大项集，我们输出具有足够置信度的所有规则，这些规则涉及且只涉及该集合中的所有元素。对于每个大项集 S，对于每个子集 $s \subset S$，如果 $S - s \Rightarrow s$ 有足够的置信度，我们输出规则 $S - s \Rightarrow s$，此规则的置信度由 s 的支持度除以 S 的支持度给定。 [905]

我们现在考虑如何产生所有的大项集。如果项集的可能数量较小，单次数据扫描足以检测出所有项集的支持度水平。为每个项集维护一个初始值为 0 的计数。当获取一条购买记录时，对于每个项集来说，如果此项集中的所有项都包含在该次购买中，则此项集的计数值增加。例如，如果一次购买行为包括项 a、b 和 c，则 $\{a\}$、$\{b\}$、$\{c\}$、$\{a, b\}$、$\{a, c\}$、$\{b, c\}$ 和 $\{a, b, c\}$ 的计数值增加。在扫描结束时，那些计数值足够大的项集所对应的项的关联程度就高。

项集数量呈指数形式增长，过大的项数会使困扰描述的过程不可行。幸运的是，几乎所有项集通常具有很低的支持度，而且已经开发了一些优化方法来去除大部分这样的不必考虑的项集。这类技术对数据库进行多次扫描，每次扫描只考虑某些集合。

在产生大项集的 **apriori** 技术中，第一次扫描仅考虑含有单个项的项集。第二遍扫描考虑有两个项的项集，依次类推。

在一遍扫描结束时，所有具有足够支持度的集合作为大项集输出。一遍扫描结束时，所发现的具有太小支持度的集合被删除。一旦某个集合被删除，就不需要再考虑它的任何超集。换句话说，在第 i 次扫描中，我们只需统计那些大小为 i 的项集的支持度。已发现它们的所有子项集具有足够高的支持度。要保证此特性只须测试所有大小为 $i-1$ 的子项集即可。对于某个 i，当第 i 次扫描结束时，我们将会发现不存在具有足够支持度的大小为 i 的项集，从而我们无须考虑大小为 $i+1$ 的任何项集。然后计算终止。

20.6　其他类型的关联

使用朴素的关联规则有几个缺点。其中一个主要缺点是许多关联是可预知的，所以这样的关联不太有意义。例如，如果许多人购买谷物，并有许多人购买面包，我们可以预测会有相当多的人两者都买，即使这两种购买行为之间没有任何联系。实际上，即使购买谷物会对购买面包产生一定程度上的负面影响(也就是说，购买谷物的顾客可能会比一般顾客少购买面包)，谷物和面包这两种购买行为之间的关联仍然会有较高的支持度。

更有趣的是从预期的同时发生的两件事中是否出现**偏离**(deviation)。用统计术语来表述，我们寻找的是项之间的**相互关联**(correlation)；相互关联可以是正的，此时同时发生率可能比期望值要高；相互关联也可以是负的，此时项的同时发生率比预期的要小。这样，如果购买面包的行为与购买谷物的行为不相关(not correlated)，那么就不报道这样的关联，即使这两者之间有强的关联关系。有一些相互关联的标准度量广泛应用于统计学领域。有关相互关联的更多信息可参阅有关统计学方面的标准教科书。

数据挖掘应用的另一个重要类别是序列关联(或序列相互关联)。时间序列数据，比如连续几天的 [906]
股票价格，就是一个序列数据的例子。股市分析人员试希望找出股市价格序列之间的关联。这种关联的一个例子就像下面的规则："无论何时债券价格上升，股票价格会在两天内下跌。"发现这种序列之间的关联可以帮助我们做出聪明的投资决策。有关这个主题的研究请参阅文献注解中的文献。

来自时态模式的偏离通常是有趣的。例如，如果某家公司每年以稳定的速率增长，从通常增长率中出现的偏差是令人惊奇的。如果冬装在夏季的销售量下降是不会令人惊讶的，因为我们可以通过历年情况预测出这种情况。我们没有从过去经验中预期到的偏离可能被认为是有趣的。挖掘技术可以发现与人们基于过去的时态或序列模式所做出的预测的偏离。有关这个主题的研究请参阅文献注解中的文献。

20.7　聚类

直观地，聚类是指在给定数据中找到点的簇的问题。聚类(clustering)问题可以从不同方式的距离

度量中形式化而来。一种方式是将它表述为把点分组为 k 个集合(对于给定的 k)的问题，使得这些点到它们所属聚类质心的平均距离最小。[⊖]另一种方式是对点分组使得每个聚类中每对点之间的平均距离最小。还有其他一些定义，详细信息见文献注解。但所有这些定义背后的意图都是将相似的点一起划分到一个单独的集合中去。

另一类聚类出现在生物学分类系统中。(这样的分类系统并不试图对类进行预测，而是试图将相关项目聚类到一起。)例如，猫和人聚类到哺乳动物类，而鲸鱼和蜥蜴聚类到爬行动物类。哺乳动物和爬行动物都属于共同的脊椎动物类。哺乳动物的聚类还有子聚类，例如食肉类和灵长类。我们由此得到层次聚类(hierarchical clustering)。给定不同物种的特性，生物学家创建了一个复杂的层次聚类模式，将相关物种一起聚合到不同的层次等级中。

层次聚类也可用于其他领域——例如，对文档聚类。Internet 目录系统(比如 Yahoo! 的目录)将相
[907] 关文档按层次方式聚类(见 21.9 节)。层次聚类算法可分为**凝聚聚类**(agglomerative clustering)算法或**分裂聚类**(divisive clustering)算法，前者从构造小的聚类开始，然后创建更高等级；后者首先创建层次聚类的更高等级，之后将每个聚类结果细分为更低等级的聚类。

统计学界已经对聚类进行了广泛研究。数据库研究提供了能够对大型数据集(无法放入主存)聚类的可伸缩聚类算法。Birch 聚类算法就是一种这样的算法。直观地，数据点被插入到一个多维树结构中(基于 25.3.5.3 节描述的 R 树)，并根据与树的内部结点中的代表点的邻近性引导它进入适当的叶结点。这样，附近的点在叶结点中聚类到一起并汇总(如果内存中无法存放更多的点)。这个第一阶段聚类的结果生成了一个可以放入内存中的、经过部分聚类的数据集。接着，标准聚类技术可以在内存中的数据上执行以得到最终聚类。关于 Birch 算法，以及包括层次聚类算法在内的其他聚类技术请参考文献注解中的文献。

聚类的一个有趣应用是预测一个人可能对何种新电影(或书、或音乐)感兴趣，基于以下几点：

1. 此人以往对电影的偏爱。
2. 有以往类似偏爱的其他人。
3. 这些人对新电影的偏爱。

这个问题的解决方法之一如下：为了找到过去有相似喜好的人，我们根据他们对电影的偏爱创建人的聚类。聚类的正确性可以通过下述改进：基于他们的相似性对以往的电影聚类。这样即使有些人并没有观看相同的电影，如果他们曾经观看了相似的电影，那么他们也将聚类到一起。我们可以重复这个聚类过程，交替地对人聚类，然后对电影，然后对人，以此类推，直到达到平衡为止。给定一个新的用户，我们基于该用户对已看过的电影的偏爱，找出与该用户最相似的用户聚类。然后我们可以预测，该新用户可能会对受该用户所在聚类欢迎的电影聚类中的电影感兴趣。事实上，这个问题是一个协同过滤(*collaborative filtering*)的实例，用户协同完成过滤信息的任务以找出感兴趣的信息。

20.8 其他类型的数据挖掘

文本挖掘(text mining)将数据挖掘技术应用到文本文档中。例如，有些工具可以根据用户曾经访问过的页面形成聚类，这样可以帮助用户在浏览他们的浏览历史时找到他们曾经访问过的页面。例如，页面之间的距离可以基于这些页中共有的词汇(见 21.2.2 节)进行计算。另一种应用是根据与其他页面
[908] 的相似性(见 21.9 节)自动地将页面分类到 Web 目录中。

数据可视化(data visualization)系统帮助用户检查大量的数据，并以可视化的方式发现模式。数据的可视化显示(比如图、表和其他图形化表示)使数据得以简洁地呈现给用户。单个图形屏幕能够对相当大量的文本屏幕所表达的信息进行编码。例如，如果用户想要找出工厂的生产问题是否与工厂位置相关，有问题的位置可以在地图上用某种特殊颜色(比如红色)编码。这样用户就可以快速发现出现问

⊖ 一个点集的质心定义为一个点，它在每个维上的坐标是该集合中所有点在相应维上坐标的平均值。以二维为例，点集 $\{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$ 的质心由 $\left(\frac{\sum_{i=1}^{n} x_i}{n}, \frac{\sum_{i=1}^{n} y_i}{n}\right)$ 给定。

题的地方，然后用户可以对为什么问题发生在那些位置做出假设，并可以在数据库上定量验证该假设。

作为另一个例子，数值信息可以编码为颜色，并且显示成与屏幕区域的一个像素那样小。为了探测商品对之间的关联，我们可以使用二维像素矩阵，每行和每列各代表一种商品。两种商品都购买的交易所占百分比可以用像素的颜色强度来编码。具有高关联度的商品将显示为屏幕上的亮点——可在更暗的背景中容易地发现。

数据可视化系统并不自动发现模式，但它们为用户发现模式提供了系统支持。由于人非常擅长发现可视化模式，因此数据可视化是数据挖掘的一个重要组成部分。

20.9 总结

- 决策支持系统分析由事务处理系统收集的在线数据，以帮助人们做出商业决策。由于现在大多数组织机构都进行了广泛的计算机化，因此有非常大量的信息可用于决策支持。决策支持系统有不同的形式，包括 OLAP 系统和数据挖掘系统。
- 数据仓库有助于收集和归档重要的操作数据。数据仓库用于基于历史数据的决策支持和分析，例如趋势预测。对来自输入数据源的数据进行清理通常是数据仓库中的一项重要任务。数据仓库的模式一般是多维的，包括一个或一些非常大的事实表以及几个小得多的维表。
- 在很多数据仓库应用程序中，面向列的存储系统能提供良好的性能。
- 数据挖掘是一个能半自动地分析大型数据库以找出有用模式的过程。数据挖掘的应用有许多，比如基于以往示例的数值预测，购买行为关联的发现，以及人和电影的自动聚类。
- 分类处理的是：基于训练用例的属性和训练用例实际所属的类，通过利用测试用例的属性来预测测试用例所属类。分类器类型有多种，例如：
  - 决策树分类器。这种分类器通过基于训练用例所构造的一棵树来执行分类，该树的叶结点具有类别标签。对每个测试用例遍历这棵树以找到一个叶结点，该叶结点所属的类即是预测的类。有几种技术可用于构造决策树，其中大部分是基于贪心的启发式方法。
  - 贝叶斯分类器的构造比决策树分类器更简单，并且在属性值缺失或为空的情况下工作得更好。
  - 支持向量机是另一种广泛应用的分类技术。
- 关联规则识别经常同时出现的项，比如同一位顾客可能购买的一些商品。相互关联找出与期望关联等级的偏离。
- 其他类型的数据挖掘包括聚类、文本挖掘和数据可视化。

术语回顾

- 决策支持系统
- 统计分析
- 数据仓库
  - 数据收集
  - 源驱动架构
  - 目标驱动架构
  - 数据清理
    - 合并—清除
    - 住宅操作
  - 抽取、转换、加载(ETL)
- 数据仓库模式
  - 事实表
  - 维表
  - 星型模式
- 面向列的存储
- 数据挖掘
- 预测
- 关联
- 分类
  - 训练数据
  - 测试数据
- 决策树分类器
  - 划分属性
  - 划分条件
  - 纯度
    - 吉尼度量
    - 熵度量
  - 信息增益
  - 信息内容
  - 信息增益率
  - 连续值属性
  - 类别属性
  - 二分分划
  - 多路分划
  - 过度适应
- 贝叶斯分类器
  - 贝叶斯定理
  - 朴素贝叶斯分类器
- 支持向量机(SVM)
- 回归
  - 线性回归
  - 曲线拟合
- 验证
  - 正确性
  - 召回率

- □ 准确率
- □ 特异性
- □ 交叉验证
- 关联规则
- □ 个体总数
- □ 支持度
- □ 置信度
- □ 大项集
- 其他类型的关联
- 聚类
- □ 层次聚类
- □ 凝聚聚类
- □ 分裂聚类
- 文本挖掘
- 数据可视化

实践习题

20.1 与目的驱动架构相比，描述用于数据仓库的数据收集的源驱动架构的优缺点。

20.2 在支持数据仓库的数据库系统中，为什么面向列的存储有潜在的优势？

20.3 假设有两条分类规则：一条表示薪水介于 10 000 – 20 000 美元之间的人具有好信用级；另一条表示薪水介于 20 000 – 30 000 美元之间的人具有好信用级。在何种情况下这两条规则可以在不丢失任何信息的前提下替换为单条规则：薪水介于 10 000 – 30 000 美元之间的人具有好信用级。

20.4 考虑图 20-2 描述的模式。给出一条 SQL 查询，按照商店和日期以及商店和日期上的层次来汇总销售数量和价格。

910 911

20.5 考虑这样一个分类器问题：分类器预测一个人是否患有一种特殊的疾病。假设 95% 被测试的人并没有患这种病。（也就是说，测试用例的 *pos* 对应于 5%，并且 *neg* 对应于 95%。）考虑下面的分类器：

- 分类器 C_1 总是预测负值（当然这是一个相当没用的分类器）。
- 分类器 C_2 从实际患病的人中预测出 80% 的正值，但也从没患病的人中预测了 5% 的正值。
- 分类器 C_3 从实际患病的人中预测出 95% 的正值，但也从没患病的人中预测了 20% 的正值。

给定上述分类器，回答下面的问题：

a. 对于上述每种分类器，计算它们的正确性、准确率、召回率和特异性。

b. 如果你要用分类结果去对这种疾病进行进一步筛选，你会选择其中哪个分类器？

c. 另一方面，假设你要利用分类结果开始药物治疗，如果药物可能会对不患有该疾病的人产生有害影响，你该选择其中哪个分类器？

习题

20.6 画一张图表示在图表 A 中所示的大学例子中的 *classroom* 关系如何在面向列的存储结构中存储。

20.7 解释为什么该套循环连接算法（见 12.5.1 节）在以面向列的方式存储的数据库中工作很差。给出另一种能够工作得更好的单代算法，并解释为什么你的解决方案是更好的。

20.8 在每个结点上使用二分划分换件构造一个决策树分类器，使用如下所示关系 $r(A, B, C)$ 中的元组作为训练数据；属性 C 代表所属的类。表示出最终的树，其中每个结点显示各个属性的最佳分划及其信息增益值。

$(1, 2, a)$，$(2, 1, a)$，$(2, 5, b)$，$(3, 3, b)$，$(3, 6, b)$，$(4, 5, b)$，$(5, 5, c)$，$(6, 3, b)$，$(6, 7, c)$

912

20.9 假设一家服装商店所有交易的一半购买了牛仔裤，商店所有交易的三分之一购买了T恤衫。再假设购买牛仔裤的交易中有一半也购买了T恤衫。写出你能从上述信息中推导出来的所有（非平凡的）关联规则，并给出每条规则的支持度和置信度。

20.10 考虑寻找大项集的问题。

a. 对通过使用单趟数据扫描得到的一个给定项集，描述如何找到其支持度。假定项集和相关信息（如计数）可以放入内存中。

b. 假设一个项集的支持度小于 f。说明没有该项集的任何超集的支持度会大于或等于 f。

20.11 创建一个事务集合的小例子，说明虽然有很多事务包含两种商品，也就是说包含两种商品的项集有很高的支持度，但是购买其中一种商品的行为可能对购买另外一种的行为产生负相关。

20.12 一本书中部分、章、节和小节的组织和聚类有关。解释其原因以及是什么形式的聚类。

20.13 以你喜欢的体育运动为例，建议一个运动队如何利用预测挖掘技术。

工具

对于我们在这一章学习的每种应用都有各式各样的工具可用。大多数数据库厂商把 OLAP 工具作为其数

据库系统的一部分或作为另加的应用软件提供。其中包括来自微软公司、SAP、IBM 和 Oracle 的 OLAP 工具。Mondrian OLAP 服务器是公共领域的 OLAP 服务器。很多公司为特定应用提供分析工具，比如客户关系管理。

主要的数据库厂商还提供了与其数据库系统相结合的数据仓库产品。这为数据建模、清理、加载和查询提供了支持功能。Web 站点 www.dwinfocenter.org 提供了关于数据仓库产品的信息。

还有许多各种种类的通用数据挖掘工具，包括 SAS 研究院的数据挖掘套件、IBM Intelligent Miner 和 Oracle。也有一些开源的数据挖掘工具，比如广泛使用的 Weka 和 RapidMiner。开源的商务智能套件 Pentaho 有一些组件，包括 ETL 工具，Mondrian OLAP 服务器和基于 Weka 的数据挖掘工具。

把通用的挖掘工具用到特定应用中需要很多专业技术。因此，开发了大量的挖掘工具用于处理专门的应用。网站 www.kdnuggets.com 提供了关于挖掘软件、解决方案和出版物等的广泛目录。

文献注解

可以从诸如 Bulmer[1979]和 Ross[1999]那样的标准统计教科书中找到统计函数的定义。

Poe[1995]和 Mattison[1996]提供了讲述数据仓库的教材。Zhuge 等[1995]描述了数据仓库环境下中的视图维护。Chaudhuri 等[2003]描述了用于数据清理的模糊匹配技术。而 Sarawagi 等[2002]描述了一个采用主动学习技术进行去重的系统。

Abadi 等[2008]提出了面向列和面向行存储的比较，包括相关的查询处理和优化问题。

Witten 和 Frank[1999]、Han 和 Kamber[2000]提供了讲述数据挖掘的教科书。Mitchell[1997]是有关机器学习的经典教材，并详细讲述了分类技术。Fayyad 等[1995]广泛收集了关于知识发现和数据挖掘的文献。Kohavi 和 Provost[2001]收集了关于数据挖掘和电子商务应用的文献。

Agrawal 等[1993b]提供了早期在数据库上进行数据挖掘的概述。Agrawal 等[1992]和 Shafer 等[1996]描述了用于计算具有大训练集的分类器的算法；本章介绍的决策树构造算法基于 Shafer 等[1996]的 SPRINT 算法。Cortes 和 Vapnik[1995]介绍了支持向量机上的几个关键结果，而 Cristianini 和 Shawe-Taylor[2000]提供了讲述支持向量机的教材。

Agrawal 等[1993a]引入了关联挖掘的概念。Agrawal 和 Srikant[1994]介绍了关联规则挖掘的有效算法。Srikant 和 Agrawal[1996a]、Srikant 和 Agrawal[1996b]描述了挖掘不同形式的关联规则的算法。Chakrabarti 等[1998]介绍了挖掘意外时序模式的技术；Sarawagi[2000]描述了数据立方体和数据挖掘集成的技术。

聚类在统计学领域有漫长的研究历史，Jain 和 Dubes[1988]提供了讲述聚类的教材。Ng 和 Han[1994]描述了空间聚类技术。用于大数据集的聚类技术由 Zhang 等[1996]给出。Breese 等[1998]对协同过滤的不同算法进行了经验分析。Konstan 等[1997]给出了用于新闻文章的协同过滤技术。

Chakrabarti[2002]和 Manning 等[2008]提供了讲述信息检索的教材，包括深入讲述了有关文本和超文本数据的数据挖掘任务，比如分类与聚类。Chakrabarti[2000]提供了超文本挖掘技术的综述，比如超文本分类和聚类。

913
914

第21章
Database System Concepts, 6E

信息检索

不同于关系数据库中严格结构化的数据，文本数据是非结构化的。术语**信息检索**通常指的就是非结构化文本数据的查询。信息检索系统与数据库系统有很多类似之处，特别是在辅助存储器上的数据存储和检索方面。然而，信息系统领域与数据库系统领域的侧重点不同，它强调基于关键字的查询、文档与查询的相关性，以及文档的分析、分类和索引等问题。当前，Web搜索引擎已不局限于文档检索，而同时研究更为广泛的问题来满足用户的信息需求，譬如显示哪些信息作为关键字查询的结果。

21.1 概述

信息检索(information retrieval)领域与数据库领域的研究是并行发展的。在信息检索领域使用的传统模型中，信息组织成文档，而且设想文档数量很大。存储在文档中的数据是非结构化的，没有相关的模式。基于用户输入来定位相关文档构成了信息检索的过程。用户输入包括关键字、示例文档等。

Web提供了一种与Internet上信息资源进行接触和交互的便利方式。然而，面对Web的一个长期存在的问题是存储信息的爆炸，用户在定位感兴趣的信息时很少能得到指导性帮助。信息检索在使Web成为一个高效和有用的工具(尤其是对于研究人员)方面起到了关键性的作用。

信息检索系统的传统例子是在线图书馆目录和在线文档管理系统，比如那些存储报纸文章的系统。这些系统中的数据组织成文档的集合。报纸文章或(图书馆目录中的)目录条目就是文档的例子。在
915 Web环境中，每个HTML页面通常被认为是一份文档。

使用这类系统的用户可能想要检索一份特定的文档或一类特定的文档。用户预期的文档通常用关键字(keyword)集合来描述。例如，关键字"数据库系统"可能用来定位数据库系统方面的图书，而关键字"股票"和"丑闻"可能用来定位股市丑闻方面的文章。文档本身已经与一组关键字相关联。如果文档的关键字包含用户提供的关键字，就被检索出来。

基于关键字的信息检索不仅可用于检索文本数据，还可用于检索其他类型的数据，比如视频或音频数据。这些数据与一些描述性关键字相关联。例如，与一部视频电影关联的关键字可以是它的片名、导演、演员、类别等；而一张图片或一段视频剪辑可能会有标签，标签就是描述这张图片或这段视频剪辑的关键字。

这种模型与传统数据库系统使用的模型有几点不同：

- 数据库系统处理信息检索系统中未涉及的一些操作。例如，数据库系统涉及更新，以及有关并发控制和持久性所需的事务处理需求。这些事情在信息系统中并不重要。类似地，数据库系统处理按相对复杂的数据模型组织的结构化信息(比如关系模型或面向对象数据模型)；而信息检索系统传统上使用了一个简单得多的模型，它的数据库中的信息是简单地按照非结构化文档集合组织的。
- 信息检索系统处理数据库系统中未得到足够重视的一些操作。例如，信息检索领域处理了查询非结构化文档集的问题，关注诸如关键字查询、根据文档与查询相关度的估计对文档排名的问题。

除了仅包含一组词语的简单关键字查询外，信息检索系统一般允许使用由关键字和逻辑连接词(*and*、*or*、*not*)组成的查询表达式。例如，用户可以要求找出包含关键字"摩托车 *and* 维护"的所有文档，或者找出包含关键字"计算机 *or* 微处理器"的文档，甚至包含关键字"计算机 *but not* 数据库"的文

档。没有上述任何连接词且包含关键字的查询指定以 *and* 作为隐式的关键字连接。

在全文(full text)检索中，每份文档中的所有词都当作关键字。对于非结构化文档，因为可能无法得到有关信息来判断文档中哪些词为关键字，所以全文检索是必要的。由于文档中的所有的词都是关键字，因此我们将用术语(term)来表示文档中的词。 916

对于没有连接词的查询，信息检索系统用其最简单的形式定位并返回所有包含该查询中全部关键字的文档；对于连接词会按照你期望的方式处理。更复杂的系统会估计文档对某个查询的相关性，这样文档可以按照估计的相关度顺序显示。它们利用有关术语出现频度的信息和超链接信息估计相关性。

以 Web 搜索引擎为代表的信息检索系统目前已超越了仅仅基于排名来检索文档。当前，搜索引擎旨在通过判断一个查询所涉及的主题并同时呈现被判定为相关的 Web 页面和有关该主题的其他信息，来满足用户的信息需求。例如，给定"板球"这一查询术语，搜索引擎会显示当前正在进行的或最近举行的板球比赛比分，而不仅仅是显示与板球相关的排名高的文档。又如，为了回答"纽约"这一查询，除了与纽约相关的 Web 网页之外，搜索引擎还会显示纽约的地图和图片。

21.2 使用术语的相关性排名

满足某个查询表达式的所有文档的集合可能是非常巨大的，特别是，Web 上有数十亿的文档，Web 搜索引擎上的大多数关键字查询都会找到数十万包含查询关键字的文档。全文检索使这个问题变得更糟：每份文档可能包含许多术语，那些仅仅附带地提及的术语和与文档真正相关的术语会等同对待。结果会检索到一些不相关的文档。

因此，信息检索系统估计文档与查询的相关性，并且只返回高度相关的文档作为结果。相关性排名不是一门精密科学，但已存在一些普遍认可的方法。

21.2.1 使用 TF-IDF 的排名方法

第一个要解决的问题是，给定一个特定的术语 t，某份特定文档 d 与该术语的相关性如何。一种处理方法是用该文档中该术语的出现次数作为对相关性的度量，这是基于一个假设：相关的术语很有可能在文档中提及多次。只统计一个术语的出现次数通常不是一个好的相关性指示器：首先，出现次数取决于文档的长度；其次，某个术语出现 10 次的文档的相关性可能并不是术语只出现 1 次的文档的相关性的 10 倍。

有一种用于度量 $TF(d, t)$(文档 d 对术语 t 的相关性)的方法是：

$$TF(d, t) = \log\left(1 + \frac{n(d,t)}{n(d)}\right)$$ 917

这里的 $n(d)$ 表示文档中的术语个数，$n(d, t)$ 表示文档 d 中术语 t 的出现次数。注意，这个公式考虑了文档长度。文档中该术语的出现次数越多相关性越大，尽管它不是直接正比于出现次数。

许多系统利用其他信息改进上面的公式。例如，如果该术语出现在标题、作者列表或摘要里，则认为该文档与该术语具有更高的相关性。类似地，如果一个术语在文档中首次出现的位置靠后，则认为该文档可能比首次出现的位置靠前的文档的相关性要小。上面的想法可以形式化为所示公式 $TF(d, t)$ 的扩展。在信息检索领域，不管实际使用的是哪一个公式，一份文档对一个术语的相关性称作**术语频率**(Term Frequency，TF)。

一个查询 Q 可能包含多个关键字。一份文档对含有两个或更多关键字的查询的相关性估计是通过将该文档对每个关键字的相关性度量结合在一起得到的。一个将度量值结合的简单方法是把它们加起来。然而，我们不能同样对待所有用作关键字的术语。假设一个查询使用两个术语，一个出现的频率高，例如"database"，另一个出现的频率低，例如"Silberschatz"。包含"Silberschatz"但不包含"database"的文档应比包含"database"但不包含"Silberschatz"的文档的相关性高。

为了解决上面的问题，使用**逆文档频率**(Inverse Document Frequency，IDF)对术语赋权值。逆文档频率定义为：

$$IDF(t) = \frac{1}{n(t)}$$

这里的 $n(t)$ 表示包含该术语 t 的文档(系统所索引的文档)的数量。于是一份文档 d 对一个术语集 Q 的**相关性**(relavance)定义为:

$$r(d, Q) = \sum_{t \in Q} TF(d,t) \times IDF(t)$$

如果允许用户指定查询中术语的权重 $w(t)$,则该度量可以进一步改进。此时,通过在上述公式中把 $TF(t)$ 乘以 $w(t)$,即可将用户定义的权重也考虑进去。

上述利用术语频率以及逆文档频率作为度量一份文档的相关性的方法称为 **TF-IDF** 方法。

几乎所有的文本文档(英语)都包含诸如"and"、"or"、"a"之类的词。由于它们的逆文档频率非常低,因此这些单词对查询目的来说是无用的。信息检索系统定义了一个词集合,称作**停用词**(stop word),它包括大约 100 个最常用的词,创建索引时从文档中去除该集合中的词。这类词不作为关键字
918 使用,如果在用户提供的关键字中出现就要被去除。

当一个查询包含多个术语时,另一个需要考虑的因素是这些术语在文档中的**接近度**(proximity)。如果文档中术语彼此接近,则该文档的相关性应该比其中术语彼此疏远的文档要高。可以修改 $r(d, Q)$ 的公式将接近度也考虑进去。

给定一个查询 Q,信息检索系统的工作是按照文档与 Q 的相关性的降序返回文档。由于相关的文档可能很多,因此信息检索系统一般只返回具有最高估计相关性度量的前几份文档,同时允许用户与系统交互请求更多的文档。

21.2.2 基于相似性的检索

一些信息检索系统允许**基于相似性的检索**(similarity-based retrieval)。这里,用户可以向系统给出文档 A,然后请求系统检索与 A"相似"的文档。举例来说,一份文档与另一份文档的相似性的定义可以基于它们之间共有的术语。一种方法是找出 A 中具有最高 $TF(A,t) * IDF(t)$ 值的 k 个术语,然后使用这 k 个术语作为一个查询去找出与其他文档的相关性。查询中的这些术语自身以 $TF(A,t) * IDF(t)$ 为权。

更一般地,文档之间的相关性是用**余弦相似性**(cosine similarity)度量来定义的。设出现在这两份文档中任意一份文档中的术语用 t_1, t_2, …, t_n 来表示。令 $r(d,t) = TF(d,t) * IDF(t)$。这样,文档 d 和 e 之间的余弦相似性度量可以定义为:

$$\frac{\sum_{i=1}^{n} r(d, t_i) r(e, t_i)}{\sqrt{\sum_{i=1}^{n} r(d, t_i)^2} \sqrt{\sum_{i=1}^{n} r(e, t_i)^2}}$$

很容易可以证明:一份文档对其自身的余弦相似性度量等于 1,而两个彼此之间不含共有术语的文档之间的余弦相似性度量等于 0。

"余弦相似性"这个名称来自于这样一个事实:上述公式计算两个向量夹角的余弦,而每个向量就代表一份文档。向量定义如下:设在被考虑的所有文档中共有 n 个词,定义一个 n 维空间,每个词作为其中的一个维。一份文档 d 用这个空间中的一个点来表示,这个点的第 i 个坐标的值为 $r(d, t_i)$。文档 d 所对应的向量是从原点(所有坐标值都为 0 的点)连接到代表该文档的点。这样一个将文档视作 n 维空间中的点和向量的模型称作**向量空间模型**(vector space model)。

如果与文档 A 相似的文档集合很大,系统可以向用户展示其中一部分相似的文档,允许用户选择最相关的几个,然后基于与 A 的相似性和与所选文档的相似性开始一个新的搜索。由此得到的文档集合很可能就是用户想要查找的文档。这个思想称为**相关反馈**(relevance feedback)。

919 相关反馈还可以用来帮助用户在一个匹配给定查询关键字的大文档集合中,找到相关的文档。在这种情况下,可能会允许用户从返回的文档中标识出一份或几份文档是相关的;然后,系统就可以利用这些标识出的文档找出其他相似的文档。由此得到的文档集合就很可能是用户想要查找的文档。另外一种可以替代相关反馈的方法要求用户通过增加关键字来修改查询;相关反馈除了能够给出一个更好的文档最终集合作为答案之外,也更加易于使用。

当文档的数量非常大时,为了呈现给用户一个有代表性的文档集合,搜索系统可以基于文档的余弦相似性对它们进行聚类。然后,来自每个聚类的一些文档将会显示,从而在应答集合中表示了多个

类别。聚类已经在20.7节做过，人们已经开发了多种技术用来对文档集合进行聚类。有关聚类的更多信息请参考文献注解。

作为网页相似的一种特殊情况，一个文档在Web上常常会有多个副本。例如，如果某个Web站点镜像了另一个Web站点的内容，这种情况就会发生。在这种情形下，返回一个排名很高的文档的多个副本作为多个独立的答案是没有道理的；重复的文档应当去除，应当仅仅返回一个副本作为答案。

21.3 使用超链接的相关性

早期的Web搜索引擎仅使用像21.2节描述的那些基于TF-IDF的相关性度量来对文档排名。然而，当在非常庞大的文档集合（比如所有Web页面的集合）上使用这些技术时，这些技术就会有一些局限性。特别是许多Web页面都含有一个典型的搜索引擎查询所指定的所有关键字，而用户真正需要的一些页面常常只出现几次这些查询术语，因此不会得到一个很高的TF-IDF得分。

但是，研究人员很快意识到Web页面具有纯文本文档所没有的非常重要的信息，即超链接。可以使用这些链接来获得更好的相关性排名。特别是一个页面的相关性排名会受到那些指向该页面的超链接的巨大影响。本节研究如何利用超链接来对Web页面进行排名。

21.3.1 流行度排名

流行度排名（popularity ranking，也称为**威望度排名**（prestige ranking））的基本思想是找到流行的页面，并且把它们的位置排在同样包含指定关键字的其他页面之前。由于大多数搜索是为了从一些较流行的页面中找到信息，因此把那些流行的页面排在靠前的位置一般来说是一个好主意。例如，术语"google"可能在大量的页面中出现，但在所有那些含有术语"google"的页面中，页面google.com是最流行的一个。因此对于一个含有术语"google"的查询来说，页面google.com应该排名为与之最相关的[920]答案。

对某个页面的相关性度量的传统方法，比如我们在21.2节所看到的基于TF-IDF的度量方法，可以与该页面的流行度相结合，从而得出对于相应查询的页面的相关性的总体度量。具有最高总体相关性度量值的页面会作为查询的最优答案而返回。

这引出了如何定义以及如何发现页面流行程度的问题。一种方法是找出一个页面被访问的次数，使用这个数来度量站点的流行程度。然而，没有该站点的配合是不可能获得这类信息的，而且即便一些站点愿意公开这类信息，向所有站点获取该信息是困难的。站点甚至会提供虚假的关于访问频率的信息，以使得自己的排名提高。

一个非常有效的替代方法是使用指向一个页面的超链接来度量其流行度。许多人都有书签文件，里面包含指向他们经常使用的站点的链接。出现在大量书签文件中的站点可以被推断为是非常流行的站点。书签文件通常都是私下进行存储的，是不能在Web上得到的。然而，确实有许多用户，他们维护着含有指向他们最喜欢的Web页面的链接的Web页面。许多Web站点也含有指向其他相关站点的链接，而这些链接也可以用来推断那些链接所指向站点的流行度。Web搜索引擎可以取得Web页面（通过一种称为抓取的过程，我们会在21.7节描述），分析它们以便找出这些页面间的链接。

估计一个页面的流行度的第一个方法是使用链接到该页面的页面数目作为其流行度的度量。然而，这种方法自身有其缺点：许多站点包含很多有用的页面，但是外部的链接通常仅仅指到该站点的根页面。根页面则包含到该站点的其他页面的链接。这样，这些其他的页面会被误认为不是非常流行，于是在回答查询的时候会得到一个较低的排名。

一种替代方法是将流行度与站点相关联，而不是和页面相关联。一个站点的所有页面获得该站点的流行度。一个流行的站点上的除根页面之外的页面也会从该站点的流行度中受益。然而，这又会引出一个问题：是什么构成了一个站点呢？通常来说，一个页面的URL的Internet地址前缀就构成了对应于该页面的站点。然而，有许多站点都含有大量的几乎不相关的页面，比如说大学的主页服务器和门户网站，比如groups.yahoo.com或groups.google.com。对于这些网站来说，其网站中一部分内容的流行度并不能暗示该网站另一部分内容的流行度。

还有一种更简单的替代方法，它允许从流行页面到它们链接到的页面间进行**威望度的传递**

(transfer of prestige)。在这种模式下，与民主主义一人一票的原则不同，从一个流行页面 x 指向页面 y 的一个链接与从一个不是那么流行的页面 z 指向页面 y 的链接相比，前者会把更多的威望度赋予页面 y。[⊖]

这个流行度的概念事实上形成了一个循环定义，由于一个页面的流行度是由其他页面的流行度来
921 定义的，在页面之间可能会出现链接的循环。然而，页面的流行度可以通过线性联立方程组来定义，它可以使用矩阵操作技术来求解。定义线性方程组的方式可以使其有一个唯一并且良好定义的解。

有趣的是，我们注意到强调流行度排名的基本思想实际上是相当老的，它在 20 世纪 50 年代就在社会学家发展起来的社会网络理论中首次出现。在社会网络环境中，目标是定义人们的威望度。举个例子：由于许多人都认识美国总统，因此他的威望度会很高。如果多个有威望的人都认识某人，那么这个人的威望度也会很高，即使没有很多人认识她。使用线性联立方程组来定义流行度度量也可以追溯到这个工作。

21.3.2 PageRank

Web 搜索引擎 Google 引入了 PageRank，这是一个页面流行度的度量。它是基于指向该页面的页面的流行度的页面流行度度量。利用 PageRank 流行度度量来对一个查询的应答进行排名可以使得到的结果大幅优于以前使用过的排名技术所得到的结果。以至于 Google 能在非常短的时间内成为使用最广泛的搜索引擎。

PageRank 可以使用**随机游走模型**(random walk model)来直观地理解。假设一个正在上网的人在 Web 页面上进行如下所述的随机游走(周游)：首先从一个随机的 Web 页面开始第一步，而每一步随机游走者采取下述策略之一：游走者以 δ 的概率跳到一个随机选择的 Web 页面，还有就是游走者以 $1-\delta$ 的概率从当前 Web 页面中随机选择一个链向外部的链接并打开该链接。这样，一个页面的 PageRank 就是随机游走者在任意一个给定的时间点访问页面的概率。

注意到许多 Web 页面指向的那些页面更有可能被访问到，因此这些页面就会有一个更高的 PageRank。类似地，具有高 PageRank 的 Web 页面所指向的页面，被访问到的概率也较高，这样，它们就会有一个更高的 PageRank。

PageRank 可以使用如下所述的一组线性方程来定义。首先，Web 页面被赋予整数标识符。这样定义跳转概率矩阵 T：将 $T[i,j]$ 设为一个正在沿着从页面 i 引出的链接前进的随机游走者沿链接走向页面 j 的概率。假定从 i 出发的每个链接都是等概率的：$T[i,j]=1/N_i$，其中 N_i 是页面 i 引出的链接数目。矩阵 T 中的多数元素为 0，因此最好用邻接表来表示。这样，页面 j 的 PageRank $P[j]$ 可以定义为：

$$P[j]=\delta/N+(1-\delta)*\sum_{i=1}^{n}(T[i,j]*P[i])$$

其中 δ 是 0～1 之间的一个常数，N 是页面数目；δ 代表随机游走过程中某一步为跳转的概率。

如上产生的方程组通常使用迭代技术求解，从令 $P[i]$ 等于 $1/N$ 开始。迭代过程的每一步利用前次
922 迭代产生的 P 的值计算 $P[i]$ 的新值。当在某次迭代中任意 $P[i]$ 值上发生的最大变化低于某个临界值时，迭代过程停止。

21.3.3 其他的流行度度量

像 PageRank 这样基本的流行度度量在对查询结果排名过程中扮演了重要的角色，但绝不是唯一的因素。页面的 TF-IDF 得分用来判断其对于查询关键字的相关程度，它必须与流行度排名结合起来。其他因素也必须考虑在内，以便处理 PageRank 以及相关的流行度度量的局限性。

有关站点被访问的频繁程度的信息是一个非常有用的流行度的度量，但是正如前文所述，这一信息往往是难以获得的。不过，对于作为返回结果的网页链接，搜索引擎的确记录了该网页被点击的频度比例。这个比例可以用于站点流行度的度量。为了记录该频度比例，搜索引擎不直接返回指向目标

⊖ 在某种意义上，这与对名人(如电影明星)对某个产品的认可给予额外的权重很相似，因此，虽然在实践中它是有效的且已经得到广泛使用，从意义仍然是值得商榷的。

网页链接，而是提供一个经过搜索引擎自身站点指向目标网页的间接链接。搜索引擎通过该间接链接记录了页面的点击行为，并且浏览器被透明地重定向至原始链接。[⊖]

PageRank 算法的一个缺点就是它设定了一个流行度度量，而这个度量并不把查询关键字纳入考虑范围之内。例如：页面 google.com 很可能有一个非常高的 PageRank，因为许多站点都包含指向其的链接。假设它包含一个顺便提到的词，比如"Stanford"（多年前 Google 的高级搜索页面确实包含这个词）。一个对于关键字 Stanford 的查询就会把 google.com 作为排名最高的结果返回，排在更相关的结果（如 Stanford 大学的主页）之前。

一个广泛使用的解决这个问题的方法是：利用指向某个页面的链接的锚文本中的关键字来判断该页面与什么话题高度相关。链接的锚文本由出现在 HTML 的 **a href** 标签中的文本组成。例如：链接

**<a href = "http://stanford.edu" > Stanford University **

的锚文本就是"Stanford University"。如果某站点有许多指向 stanford.edu 的链接在它们的锚文本中出现 Stanford 这个词，那么该页面就可以被判断为是与关键字 Stanford 非常相关的。锚文本附近的文本也可以考虑进来。例如，某个 Web 站点可能会含有文本"Stanford's home page is here"，但是它仅仅使用了"here"这个词作为指向 Stanford 的 Web 站点的链接的锚文本。

基于锚文本的流行度和其他流行度度量，以及 TF-IDF 度量结合起来，就得到一个查询应答的整体排名（将于 21.3.5 节中讲述）。作为一种实现技巧，出现在锚文本中的词常常被视为页面的一部分；其 [923]
词频是根据所在页面的流行度来计算的。而后，TF-IDF 排名方法自动地将这些词考虑在内。

定义流行度时，作为一个替代方法，我们可以把关键字考虑进来，就是仅仅利用那些包含查询关键字的页面来计算某个流行度的度量，而不是使用所有可得到的 Web 页面计算流行度。这个方法代价更高，因为流行度排名的计算必须在得到每个查询时动态地执行，而 PageRank 是静态地计算一次，然后被所有查询复用。Web 搜索引擎每天处理数十亿计的查询，它们承担不起化费那么多时间用来回答一个查询的代价。结果是，虽然这个方法能够得到更好的结果，但是使用得并不非常广泛。

HITS 算法就是基于上面的想法：首先找到包含查询关键字的的页面，然后就利用这些有关页面集合计算流行度度量。此外，它还引入了**链接中心**和**权威页**的概念。一个**链接中心**（hub）是一个存储了到许多有关页面的链接的页面，它可能本身并不包含实际的主题信息，而是指向包含实际信息的页面。比较而言，一个**权威页**（authority）是一个包含了实际主题信息的页面，虽然它可能并不包含指向许多有关页面的链接。这样每个页面得到了作为链接中心的一个威望值（**链接中心威望**）和作为权威页的另一个威望值（**权威页威望**）。像以前一样，威望的定义是循环的，并且是由线性联立方程组定义的。若一个页面指向许多具有高权威页威望的页面，则该页面得到较高的链接中心威望；若一个页面被许多具有高链接中心威望的站点所链接，则该页面得到较高的权威页威望。对于给定的一个查询，具有高权威页威望的页面的排名比其他页面要高。对于进一步细节请参考文献注解。

21.3.4 搜索引擎作弊

搜索引擎作弊（search engine spamming）指的是尝试建立 Web 页面或页面集合，其被设计用来使得站点对于某些查询得到一个高的相关度排名，即使这些站点实际上并不是流行的站点。例如：一个旅游站点想对于含有关键字"旅游"的查询得到高排名。它可以通过在其页面中重复多次"旅游"这个词来得到高的 TF-IDF 得分。[⊖]即使一个与旅游无关的站点，比如一个色情站点，也可以这么做，然后会在对于旅游这个词的查询上排名较高。事实上，这种对于 TF-IDF 的作弊在早期的 Web 搜索中非常常见。在这些作弊站点与那些试图发现作弊行为并拒绝给它们高排名的搜索引擎之间曾经有一场旷日持久的

⊖ 有时候这个间接链接对用户是隐藏的。比如当你把鼠标指针指向 Google 查询结果中的一个链接（比如 db-book.com），这个链接就会直接显示到那个网站。尽管如此，至少在 2009 年中旬，实际上当你点击一个链接的时候，结果页中的 Javascript 代码会重写该链接，使其通过 Google 站点间接地访问目标链接。如果你使用浏览器中的后退按钮后退到查询结果页面，并且用鼠标指向那个链接，那么该链接的 URL 就可见了。

⊖ Web 页面中的重复词会误导用户；作弊者能够解决这个问题。对于一个相同的 URL，他们向搜索引擎和其他用户或文本阅的页面；或者让这些重复词不可见，例如把这些词用小的白色字体显示在白色背景之上。

战争。

诸如 PageRank 这样的流行度排名模式使得搜索引擎作弊行为更加难以实施。这是由于仅仅依靠重
924 复词来得到 TF-IDF 高分已经不够了。然而，即使这些技术出现还是有可能使用作弊来欺骗的：通过建立一个彼此互相指向的 Web 页面的集合来增加它们的流行度排名。诸如使用站点而不是页面作为排名单元(选择适度规范化的跳转概率)的技术已经提出，用来避免某些作弊技术，但是对其他作弊技术并不是足够有效。搜索引擎作弊者与搜索引擎之间的战争即使到了今天还依然在延续。

HITS 算法的链接中心与权威页方法更容易受到作弊行为的影响。作弊者可以建立一个包含指向关于某个主题的好的权威页的链接的 Web 页面，从而使该页面得到一个高的链接中心分数。此外，作弊者的 Web 页面包括链接，这些链接指向的是他们希望流行起来的页面。但是与这个主题并不是很相关。因为这些链接的页面被一个具有高链接中心分数的页面所指向，所以这些页面会得到一个高的但不是应得的权威页分数。

21.3.5 将 TF-IDF 和流行度排名度量方法结合

我们已经探讨过了两种广泛应用于排名的特征，即 TF-IDF 和流行度(如 PageRank)。TF-IDF 本身已经综合反映了许多因素，包括原始术语频度、逆文档频率、在指向另一页面的锚文本中出现的术语，以及一组其他的因素，例如出现在标题中的术语出现在文档前部的术语以及较大字体的术语等。

如何将从各个因素得来的页面的各种分值结合起来，从而产生出一个全局的评价是信息检索系统需要解决的一个重要问题。在搜索引擎初期，人们创造函数式来将诸多分值整合为一个全局分数。但是当今的搜索引擎应用机器学习的方法来决定如何获得整合分数。典型地，分值合并公式是固定的，但是公式以不同评分因素的权重作为参数。通过使用由人工排名得到的查询结果作为训练数据集，机器学习算法可以计算出在多个查询上表现最佳的每个评价因素的权重的值。

我们注意到，多数搜索引擎并不公开他们如何计算相关性排名；他们认为公开排名技术将会有利于竞争者赶超自己，并且使得搜索引擎欺骗行为更加容易实现，这将导致更差的搜索结果质量。

21.4 同义词、多义词和本体

考虑利用查询"摩托车维护"查找关于摩托车维护的文档的问题。假定每份文档的关键字是标题中的词和作者名，那么标题是摩托车维修的文档将不会被找出，因为"维护"这个词并没有在它的标题中出现。

我们可以采取使用**同义词**(synonym)的方法解决这个问题。对每个词都可以定义一组同义词。每个
925 词出现时都可以被它的所有同义词(包括它自己)的 or 所代替。于是，查询"摩托车 *and* 维修"被替换为"摩托车 *and*(维修 *or* 维护)"，这个查询就可以找出所要的文档了。

基于关键字的查询还受到一个相反问题——**多义词**(homonym)的困扰。多义词是指一个单词具有多种含义。例如，单词 object 作为名词和动词具有不同含义。单词 table 可以指一个饭桌，也可以是关系数据库中的一个表。

事实上，使用同义词扩展查询是有危险的：同义词自身可能含有不同的含义。例如："allowance"对于"maintenance"这个词的其中一个意思来说是同义词，但是与查询"motorcycle maintenance"中用户想要的意思不同：使用了与用户想要的不同含义的同义词的文档也被检索到。这样用户会感到惊奇：如果一个特定的检索到的文档(例如使用"allowance"这个词)既不包含用户指定的关键字，也不包含那些在文档中意指的含义与指定关键字同义的词，为什么系统会认为这份特定的文档是相关的。因此，不首先向用户验证同义词的含义就使用同义词扩展查询不是个好主意。

对于上述问题的一个更好的方法是使系统能够理解文档中的每个词代表什么**概念**(concept)。类似地，使系统理解用户在寻找什么概念，并且返回阐释了用户感兴趣的概念的文档。支持**基于概念的查询**(concept-based querying)的系统必须分析每份文档，消除文档中每个词的歧义，并用其所代表的概念替换它。消除歧义通常通过查看文档中该词周围的其他词来完成。例如：如果某份文档包含诸如"数据库"或"查询"这样的词，那么 table 这个词很可能该被"table：data"(数据表)概念替代，而如果某份文档在 table 这个词附近包含诸如家具、椅子或木头这样的词，那么 table 这个词该被"table：furniture"概

念替代。由于查询只包含非常少的几个词，因此基于附近词消除歧义对于用户查询来说通常更加困难。因此基于概念的查询系统会提供好几个替代概念给用户，由用户在搜索继续之前选择一个或多个概念。

基于概念的查询有几个优点：例如某种语言的查询可以检索到其他语言的文档，只要它们是关于同一个概念的。如果用户不懂文档所使用的语言，那么接下来就可以使用自动翻译机制。然而，当处理数十亿文档时，对文档进行处理来消除词的歧义的代价是非常高的。因此，互联网搜索引擎原本通常不支持基于概念的查询。不过，基于概念的查询正在快速地引起越来越多的兴趣。基于概念的查询系统已经建成，并已用于其他大量的文档集合。

基于概念的查询可以通过开发概念层次来进一步扩展。例如，假设某人发出了一个查询"会飞的动物"，一个包含有关"会飞的哺乳动物"的信息的文档当然是相关的，因为哺乳动物也是动物。然而，这两个概念是不一样的，于是仅仅概念的匹配不会允许该文档作为结果返回。基于概念的查询系统能够支持基于概念层次的文档检索。

926

本体(ontology)是反映概念间联系的层次结构。最常用的联系是 **is-a** 联系。例如，美洲豹 *is-a* 哺乳动物，哺乳动物 *is-a* 动物。其他联系(诸如 part-of)也是有可能出现的，例如，机翼是 *part-of* 飞机。

WordNet 系统利用相互联系的词(在 WordNet 术语中称为一个合集)定义了大量的概念。与一个合集相互联系的词是一个概念的同义词，一个词当然可能是好几个不同概念的同义词。除了同义词之外，WordNet 还定义了多义词和其他联系。特别是，它定义的 *is-a* 和 *part-of* 联系用来连接概念，并有效定义了一个本体。Cyc 工程是建立本体的另一项努力。

除了语言级别的本体，针对特定领域可以定义不同本体，用来处理与那些领域相关的术语。例如，商业领域的本体已经建立起来，用来标准化其术语。对于为解决订单处理和其他组织间数据流的处理建立标准下层基础，这是至关重要的一步。又如，一个医药保险公司需要从医院获取包含诊断信息和治疗信息的资料。能够使术语标准化的本体可以帮助医院工作人员无歧义地理解这些资料。这将极大地帮助资料分析，例如，跟踪一段特定时间内某种疾病的病例发生的数目。

建立链接多种语言的本体也是有可能的。例如，WordNet 系统已经针对不同语言建立起来了，语言间的共同概念能够互相链接。这样一个系统可以用来完成文本的翻译。在信息检索的上下文环境中，多语言的本体可以用来实现多种语言文档间基于概念的搜索。

在使用本体来进行基于概念的查询方面的最大努力在于**语义网络**(Semantic Web)。语义网络是由万维网联盟(World Wide Web Consortium)倡导的。它包含一组能够以基于语义或含义的方式将 Web 上的数据连接起来的工具、标准以及语言。与存储于集中的存储仓库不同的是，语义网络被设计为允许如同万维网那样的分散的、分布式的增长。这一特点成就了万维网的成功。做到这一点的关键在于集成多个分布式本体的能力。于是，任何可以访问互联网的用户都可以向语义网络添加数据。

21.5 文档的索引

一个有效的索引结构对于信息检索系统查询的高效处理是十分重要的。包含指定关键字的文档可以通过使用**倒排索引**(inverted index)来高效地定位。倒排索引将每个关键字 K_i 映射到包含 K_i 的文档的列表(文档标识的列表)S_i 上。例如，如果文档 d_1、d_9 和 d_{21} 包含术语"Silberschatz"，那么术语 Silberschatz 的倒排表则表示为"d_1；d_9；d_{21}"。为了支持基于关键字相似性的相关度排名，这样的索引 927
可能不只提供文档标识，还提供关键字在文档中出现位置的列表。例如，"Silberschatz"出现在文档 d_1 的第 21 个位置，文档 d_9 的第 1 和 19 个位置上以及文档 d_{21} 的第 4、29、46 个位置上；那么带有位置信息的倒排表将是："d_1/21；d_9/1，19；d_{21}/4，29，46"。倒排表也可以包含文档的术语频率。

这些索引都必须存储在磁盘上，每个列表 S_i 都可能分布在多个磁盘页上。为了最小化检索每个列表 S_i 所需要的 I/O 操作次数，系统可能试图将每份文档列表 S_i 保存在一组连续的磁盘页上。这样，仅通过一次磁盘寻道就可以检索到整个列表。B^+ 树索引可以用来把每个关键字 K_i 映射到与其关联的倒排表 S_i 上。

and 操作用来找出包含一个给定的关键字集合中所有关键字 K_1，K_2，…，K_n 的文档。实现 *and* 操作时，我们首先检索出分别包含各个关键字的所有文档的文档标识集 S_1，S_2，…，S_n。集合的交集 $S_1 \cap$

$S_1 \cdots \cap S_n$ 给出了所求的文档集合的文档标识。*or* 操作用来给出至少包含关键字 K_1、K_2、…、K_n 中的一个的所有文档的集合。我们通过计算其并集 $S_1 \cup S_2 \cdots \cup S_n$ 来实现 *or* 操作。*not* 操作用来找出不包含给定关键字 K_i 的文档。给定文档标识集合 S，我们可以通过求差 $S - S_i$ 去掉那些包含指定关键字 K_i 的文档，其中 S_i 是包含关键字 K_i 的文档标识的集合。

给定一个查询中的关键字集合，许多信息检索系统并不一定要求检索到的文档包含所有的关键字（除非明确使用了一个 *and* 操作）。在这种情况下，会检索所有包含至少一个关键字的文档（如同 *or* 操作一样），但要用相关度度量对它们进行排名。

为了在相关度排名中使用术语频率，索引结构应该附加地维护每份文档中术语出现的次数。为了减少这种开销，可以使用一种只使用少量 bit 位的压缩表示，用于对术语频率的近似。索引也应该存储每个术语的文档频率（即包含该术语的文档个数）。

如果流行度排名独立于术语索引（如 PageRank 中的情况），列表 S_i 可以根据流行度排名（其次，对于流行度相同的文档，根据文档 id 排名）。这样，就可以使用一个简单的合并来计算 *and* 和 *or* 操作。对于 *and* 操作的情况，如果我们忽略 TF-IDF 对于相关度分数的贡献，仅仅要求文档包含给定关键字，并且如果用户只需要前 K 个答案，那么一旦得到了 K 个答案，合并就可以停止了。一般而言，考虑了 TF-IDF 分数后，最终分数较高的返回结果往往也具有较高的流行度分数，并将出现在结果列表的前端。能够估计剩余结果集可能取得的最高分数的技术已经开发出来。这些技术可以用来识别尚未处理的答案已不可能成为前 K 个答案的一部分的情况。这样，处理过程可以提早结束。

928 但是，按照流行度分数对答案进行排名在回答长倒排表扫描问题时并不非常高效。因为它忽略了 TF-IDF 分数的贡献。一种可行的处理方法是将每个术语的倒排表切成两部分。第一部分包含拥有高 TF-IDF 分值的文档（例如，该术语出现在标题内的文档或者包含该术语的锚文本所指向的文档）。第二部分包含所有的文档。倒排表的两部分均可按照（流行度、文档 ID）的形式有序存储。对于一个给定的查询，通过合并各个术语的倒排表的第一部分往往会得到一些具有较高的整体分数的答案。如果用倒排表的第一部分没有找到足够多的高分数答案，倒排表的第二部分可以用于找到所有的剩余的答案。如果某份文档具有较高的 TF-IDF 分数，使用合并倒排表第一部分的方法可以发现这样的文档。相关文献请参考文献注释。

21.6 检索的有效性度量

每个关键字都可能被大量的文档所包含，因此紧凑的表示是减少索引占用空间的关键。于是，对应于一个关键字的文档集合以一种压缩的形式维护。这样可以节省存储空间。但有的时候，索引的这种存储方式使得检索只是近似的：有一些相关文档可能没有检索到（称作**误丢弃**（false drop）或**误舍弃**（false negative））；或检索到了一些不相关的文档（称作**误选中**（false positive））。一个好的索引结构将不存在任何误丢弃，但可以有一些误选中，因为系统随后可以通过查看文档中实际包含的关键字来滤除它们。在 Web 索引中，也不希望有误选中，因为可能无法迅速地存取实际的文档用于过滤。

用于度量一个信息检索系统回答查询的好坏的度量有两个：第一个是**查准率**（precision），度量检索到的文档真正与查询相关的百分比；第二个是**查全率**（recall），度量与查询相关的文档中被检索到的文档所占百分比。理想情况下，两者都应该是 100%。

查准率和查全率对于理解一个特定的文档排名策略执行得好坏也是两个重要的衡量尺度。相关性排名策略可能带来误舍弃和误选中，但在一个更深刻的意义下，

- 当对文档进行相关性排名时，一些相关文档可能排名较低，这就会发生误舍弃。如果我们取来所有的文档（包括那些排名很低的文档），则只会有很少的误舍弃。然而，很少有人会查看最先返回的数十个以外的文档，因此可能错过一些没有排在前面的相关文档。事实上，误舍弃的多少取决于查看的文档的多少。因此，我们不用一个单独的数字作为查全率的度量，而是把查全率的度量表示为取回的文档数量的一个函数。

929

- 当不相关文档比相关文档排在更前面时会发生误选中。这也取决于查看的文档的多少。可以选择将查准率的度量作为取回的文档数量的一个函数。

一个更好且更直观的用于度量查准率的可选方法是将其作为查全率的一个函数进行度量。使用这种结合的度量方法，如果需要，查准率和查全率都可以作为文档数量的函数进行计算。

例如，我们可以说当查全率为50%时，查准率是75%，而当查全率为75%时，查准率降低到60%。一般来说，我们可以画一张关联查准率与查全率的图。这些度量可以针对单个查询进行计算，然后在查询基准测试中对于一批查询来求平均。

然而与度量查准率和查全率相关的另一个问题在于如何定义哪些文档是真正相关的，哪些不是。事实上，为了决定一份文档是否相关，需要有对自然语言的理解和对查询旨在理解。研究人员因此创建了文档和查询的集合，并人工地将文档标记为与该查询相关或不相关。不同的相关性排名系统可以运行在这些集合之上，以度量它们对多个查询的平均查准率和查全率。

21.7 Web的抓取和索引

网络爬虫(web crawler)是定位和收集Web上的信息的程序。它们沿着已知文档中存在的超文本链接递归地找到其他文档。网络爬虫从一组可由人工设定的初始链接开始，依据URL链接抓取Web上的页面。随后，爬虫定位抓取到的页面中所包含的所有URL链接信息，如果这些链接所指向的页面没有被抓取过，而且也不存在于当前的待抓取集合中，那么爬虫就把它们加入到待抓取的URL链接集合中。这一过程将以不断抓取待抓取集合中的页面并处理这些页面中的链接的形式反复进行。通过重复以上过程，所有可以由初始集合中的URL出发以任意的链接顺序到达的页面都将最终被抓取到。

由于Web上的文档数量很大，因此要想在短时间里抓取整个Web是不可能的。实际上，所有的搜索引擎都只是搜索部分而不是全部Web，它们的爬虫对它们需索的网页执行一次单独的扫描抓取需要数周或数月的时间。爬虫抓取过程中通常有许多进程，在多台主机上运行。数据库存储了需要搜索的一个链接(或站点)集合，将该集合中的链接交给每个爬虫进程。抓取过程中发现的新链接被添加进数据库中，并可能会立即或在稍后的时间里抓取。必须周期性地重新提取网页(即重新抓取)以获得更新信息，并去除不存在的站点，以使搜索索引中的信息保持合理的更新程度。

相关参考文献中有很多进行Web抓取的实用细节。例如在动态网页中生成的无限链接序列(称作**爬虫陷阱**(spider trap))，页面抓取的优先顺序以及保证Web站点不会被由爬虫发出的频繁的访问请求所淹没。

抓取到的页面被交给可能运行在不同机器上的威廉值计算系统和索引系统进行处理。进行威廉值计算和索引的系统本身并行地在多台主机上运行。当它们完成了对页面威廉值计算并且加入到索引中之后，这些页面就可以丢掉了。不过，这些页面往往是被搜索引擎缓存了起来，以便即使该页面所在原始站点无法访问，搜索引擎用户仍然可以快速地访问这个缓存页面。 930

将搜索到的网页加到正用于查询的索引中不是一个好主意，因为这样做将需要索引上的并发控制，从而会影响查询和更新的性能。替代的办法是，使用索引的一个副本回答查询，而用新搜索到的网页对另一个副本进行更新。周期性地，这两个副本互换，对旧的副本进行更新，而新的副本用于查询。

为了支持非常高的查询速度，索引可能存储在主存中，而且有多台这样的机器；系统选择性地将查询路由到不同的主机上以取得主机间的负载均衡。流行的搜索引擎经常有数万台主机承担各种各样的爬虫抓取任务、索引任务以及对于用户查询的应答任务。

网页爬虫依赖于可以由超链接所到达的所有相关页面。但是，许多包含有大量数据的站点也许不会以超链接页面的形式提供所有的数据。相反，这些站点提供了检索接口。通过使用这些接口，用户可以输入术语或者选择菜单项以获得检索结果。例如，用于存储航新信息的数据库通常没有指向包含航班信息页面的超链接，而是以搜索界面的方式提供给用户的。于是，这类站点中的信息是不能被传统的网络爬虫所访问到的。这些信息通常被称为**深度Web**(deep Web)信息。

深度Web爬虫(deep Web crawler)通过猜测在检索界面中输入什么样的术语是合理的，如何选择菜单项等方式来抽取这样的信息。通过输入每一个可能的检索词/选项，并执行这样的搜索界面，它可以抽取到包含其他方法无法得到的数据信息的网页。例如Google搜索引擎能够提供的搜索结果就包含了由深度Web爬虫所抓取到的数据。

21.8 信息检索：网页排名之外

信息检索系统原本被设计为寻找与一个查询请求相关联的文本文档。后来扩展为在 Web 上寻找与查询相关的页面。人们使用搜索引擎来完成各种任务：小到如定位一个要用到的 Web 站点这样的简单任务，大到寻找与某个所关心的主题相关的信息。Web 搜索引擎在定位欲访问的站点方面表现已非常出色。但是提供所关心的主题的相关信息的任务却难了很多。这一节将探讨一些可行方法。

对于能够(以有限的程度)理解文档和基于这样(有限)的理解回答问题的系统的需要越来越多。一
931 种方法就是根据非结构化的文档建立结构化的信息，并基于结构化信息回答问题。另一种方法应用自然语言技术找到与某个问题(自然语言短语构成)相关的文档，并将文档的相关段落作为问题的答案返回。

21.8.1 查询结果的多样化

现今，搜索引擎不仅仅能够返回一个与查询相关的 Web 页面排名列表，同时还能够返回相关的图片及视频结果。更进一步，很多站点可以提供动态变化的内容，例如体育竞技比分、股市行情。为了获得这些站点的当前信息，用户需要首先点击查询结果。取而代之的是，搜索引擎实现了能够从特定的域名获取数据的"控件"。这些数据包括体育竞技比分、股票价格以及天气状况等。这些控件还可以把获得的数据以良好的图像形式展示出来，以此作为查询结果。搜索引擎必须将可用的控件依照与检索词的相关程度进行排名，并连同 Web 页面、图像、视频以及其他类型的结果一起，展示最相关的控件结果。因此，一个查询结果拥有丰富的结果类型。

检索词常常是有歧义的。例如，一个查询"eclipse"可以用来指日食或月食，也可以用来指一个叫做 Eclipse 的集成开发环境(IDE)。如果返回结果中排名高的所有页面都是有关 IDE 的。那么，寻找日食和月食信息的用户将会非常失望。因此，搜索引擎试图提供一组涉及多种主题的结果，从而尽可能避免用户对结果很不满意的可能性。为了实现这样的目的，搜索引擎必须消除页面中使用的同语的歧义。例如，它必须能够确定某个页面中的 eclipse 指的是 IDE 还是天文现象。这样，对于一个查询，搜索引擎将试图提供与该词的最普遍含义相关的结果。

从 Web 页面中获取的结果需要归纳成为查询结果中的小片段(snippet)。在传统方法中，搜索引擎提供被检索的关键字周围的一些词语作为结果片段以便于指示该页面所包含的内容。不过，很多领域中的小片段可以通过更加有意义的方式产生。例如，如果用户查询饭店，搜索引擎可以生成除了指向饭店主页的链接之外，还包含饭店评级、电话号码以及指向地图的链接的小片段。这样的内容具体的小片段通常是由来自数据库(例如一个存储饭店信息的数据库)中的数据所生成的。

21.8.2 信息抽取

信息抽取(information extraction)系统将信息从文本形式转换为更结构化的形式。例如，一则房地产广告可能会以文本形式来描述一栋房屋的属性，例如"位于 Queens 街区的两间卧室和三间浴室的住宅，100 万美元"。一个信息抽取系统可能会从这样的广告中抽取出诸如卧室数量、浴室数量、价格以
932 及邻居这样的属性。原始的广告内容可能使用了多种词汇，如 2 室、二室、两卧等来描述两间卧室。抽取出来的信息可以用于以标准方式来构建数据。从而，用户可以说明他所感兴趣的是带有两间卧室的房屋；查询系统就可以基于结构化的数据来返回所有相关的房屋，即使原始广告中所使用的词汇不尽相同。

一个维护公司信息的数据库的组织，可能会使用信息抽取系统从报刊文章中自动抽取信息。抽取出的信息会与感兴趣的属性的变化有关，诸如辞职、解雇或公司领导间的会晤等。

又如，定位于寻找学术研究文献的搜索引擎(如 Citeseer 和 Google Scholar)抓取万维网中很可能是学术论文的页面。它们通过检查特定的条件来确定一份文档是不是一篇学术研究文章。这些条件包括是否出现了"参考文献"、"引用"以及"摘要"等词汇。然后，它们利用信息抽取技术来提取标题、作者列表以及文本的引文信息。抽取出的引用信息可以用来建立文章与引文以及文章与引用本文的文章之间的链接关系。这些引用链接关系对学术研究者非常有用。

已经建成了几个系统用来对特定应用进行信息抽取。他们利用的是语言学技术，以及特定领域的用户定义规则(比如房地产广告、学术出版物)。对于有限的范围(如一个特定的Web站点)，手动指定用于信息抽取的匹配模式是可行的。例如，在一个特定的站点，一个形如"价格：<数字>$"(其中<数字>指任意的数值)的模式可以用来匹配价格的指定位置。可以为有限数量的Web站点手动指定这样的模式。

但是，在如今包含了数以百万计的Web站点的大规模网络中，手动生成这样多的模式是更本不可能的。机器学习技术可以从一组训练数据中学习出这样的模式，因而广泛应用于信息抽取的自动执行过程中。

在抽取出的信息的一小部分中往往包含错误。这是由于某些页面包含语法上符合匹配模式的信息，但是不真正指定一个值(例如价格)。使用简单的匹配模式的信息抽取独立地匹配页面的一部分，更容易产生错误。机器学习技术能够基于模式之间的交互执行更加复杂多变的分析任务，从而尽可能地减少抽取出的信息中的错误数量，同时尽可能地扩大抽取出的信息总量。更多信息请参考文献注解。

21.8.3 问答系统

信息检索系统把问题焦点集中在针对一个给定的查询找到其相关的文档。然而，一个查询的答案可能就仅仅在文档的一部分中，或者是在好几份文档的各个小部分中。问答(question answering)系统试图对用户提出的问题提供直接的答案。例如，像"谁杀死了林肯？"这样的问题最好是能够用一行"亚伯拉罕·林肯于1865年被约翰·威尔克斯·布斯射杀"这样的文字来回答。注意到这个答案实际上并不包含"杀死"或"谁"这些词，但是系统推断出"谁"能够用一个名字来回答，而"杀死"与"射杀"是有关的。 933

以Web中的信息为目标的问答系统通常根据一个提交的问题产生一个或多个关键字查询，使用Web搜索引擎执行这些关键字查询，并解析返回的文档找到回答这个问题的文档段落。许多语言学技术和启发式方法已经用来产生关键字查询，并从文档中找到相关的段落。

问答系统中的一个难题是对于同一个问题不同的文档会给出不同的答案。例如，如果问题是："长颈鹿有多高？"不同的文档给出不同的数字作为答案。这些答案构成了一个数值的分布。问答系统可以选择平均值或中间值来作为答案返回。为了向用户反映答案不准确的情况，系统也许会一并返回均值和标准差(例如，均值为16英尺，标准差为2英尺)，或者一个基于均值和标准差的范围值(例如，在14－18英尺之间)。

当前的问答系统能力有限，因为它们并不真正理解问题以及用来解答问题的文档。然而，对于大量简单的问答任务来说，它们是有用的。

21.8.4 查询结构化数据

结构化数据主要以关系或XML形式表示。多个系统已经建成以支持在关系数据和XML数据上进行关键字查询(见第23章)。这些系统的共同主题就在于找到包含指定关键字的结点(元组或XML元素)，并找到它们之间的连接路径(或是在XML数据情况下的共同祖先)。

例如，在某个大学数据库中查询"Zhang Katz"可能会找到出现在 *student* 关系的某个元组中的 *name* 字段为"Zhang"，还有在 *instructor* 关系的某个元组中的 *name* 字段为"Katz"，以及在 *advisor* 关系中连接这两个元组的某条路径。其他的路径，例如学生"Zhang"修读一门由"Katz"讲授的课程，也会作为该查询的应答而找到。当用户不知道正确的模式，并且也不想费力去写一个SQL查询来定义他想要查询什么的时候，上述这样的查询就可以用来对数据进行即席的浏览和查询。事实上，期望那些非专业用户用结构化查询语言写出查询是不合理的，而关键字查询就很自然。

由于查询并没有完全地定义，因此它们可能会有许多不同类型的答案，这些答案必须是经过排名的。很多种技术已经提出用于在这样的设定下对答案进行排名，包括基于连接路径长度的以及基于指定边的方向和权值技术的。基于诸如外键和IDREF链接的，为元组和XML元素指定流行度排名的技术也已经提出了。更多有关在关系数据和XML数据之上进行关键字查询的信息请参考文献注解。 934

21.9 目录与分类

一个典型的图书馆用户可能会使用图书目录来定位自己所需的图书。然而，当她从书架上拿取图书时，她很可能会浏览位于附近的其他图书。图书馆用把相关的图书放在一起的方式组织图书。因此，与读者想要的书物理上接近的书可能也是读者感兴趣的，这样的安排对于读者浏览这些书是有意义的。

为了将有关的书放在一起，图书馆使用一种**分类层次**(classification hierarchy)。有关科学的书被分类到一起。在这个图书集合里进行更细的分类：计算机科学类图书组织到一起，数学类图书组织在一起等。由于数学和计算机科学有一定的联系，因此相关的图书集合存放在物理上彼此邻近的地方。在分类层次中的另一个级别上，计算机科学图书被分解为几个子领域，例如操作系统、语言和算法。图21-1表示了一个图书馆可能使用的分类层次。因为图书只能保存在一个地方，所以图书馆里的每本图书被分到分类层次中某个确切的位置上。

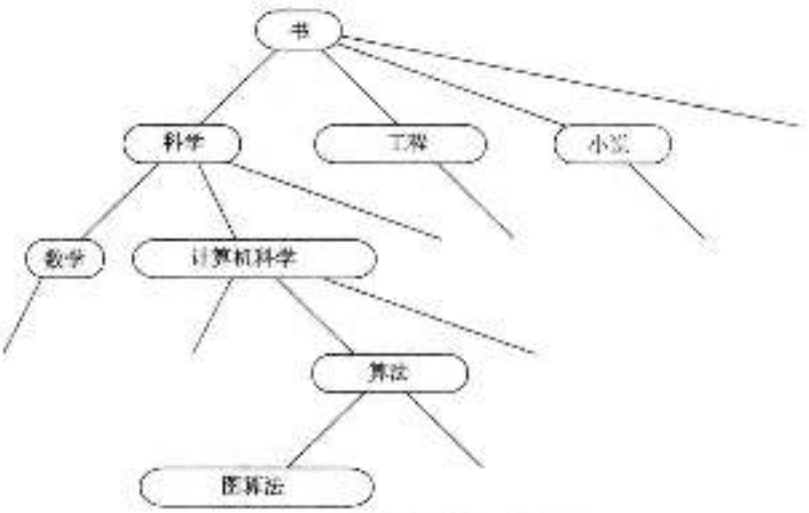


图21-1 一个图书馆系统的分类层次

在信息检索系统中，不需要将相关文档存储在一起。然而，为了允许用户浏览，此类系统需要有
935 某种地组织文档。因此，此类系统会使用与图书馆使用的分类层次类似的分类层次。而且，当显示一个特定文档的时候，也会显示一些层次上相近的文档的简短描述。

在信息检索系统中，不需要将文档保存到层次结构中的单独位置上。一个为计算机研究人员讲述数学的文档可以分在数学下面也可以分在计算机科学下面。存储在每个位置上的是文档的标识(即一份文档的指针)，使用该标识可以很容易地取得该文档的内容。

由于这种灵活性，不仅可以把一份文档分类到两个位置上，而且分类层次中的子领域本身也可以出现在两个领域下。“图算法”文档的类别可以出现在数学下面也可以出现在计算机科学下面。因此分类层次现在变成了一个有向无环图(Directed Acyclic Graph，DAG)，如图21-2所示。图算法的一份文档可能出现在DAG图中的一个单独的位置上，但可以通过不同的路径访问。

一个**目录**(directory)就是一个分类DAG结构。目录的每个叶结点存储了与该叶结点代表的主题相关的文档的链接。内部结点也可能包含一些链接，例如指向无法归类到任何子结点下的文档的链接。

为了找到有关一个主题的信息，用户需要从目录的根开始，沿着路径向下搜索DAG，直到抵达代表所需主题的一个结点为止。在向下浏览目录时，用户不仅可以找到他感兴趣的主题方面的文档，而且还可以找到分类层次中的相关文档和相关分类。用户通过在相关分类中浏览文档(或子类)可以了解一些新信息。

将Web中的大量可用信息组织成一个目录结构是一项严峻的任务。

- 第一个问题是精确确定目录层次结构。

- 第二个问题是，给定一份文档，决定哪个目录结点处与该文档相关的分类。

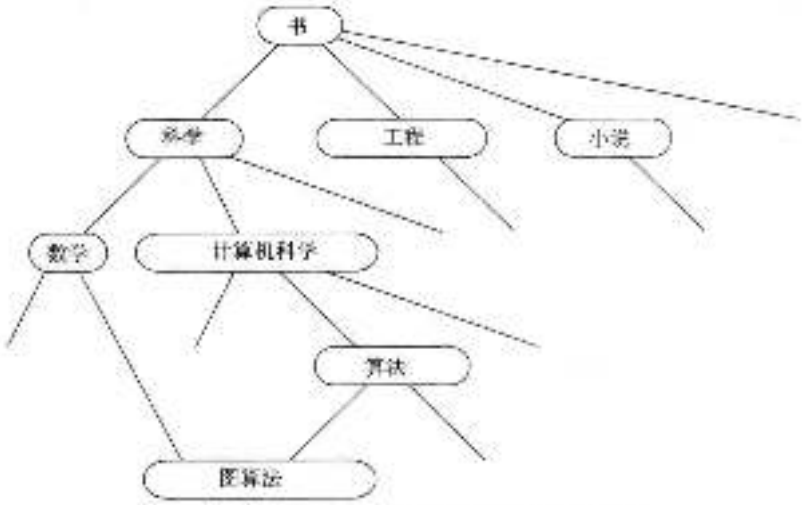


图 21-2 一个图书馆信息检索系统的分类 DAG

为了处理第一个问题，像 Yahoo! 这样的门户网站有"Internet 图书管理员"组，负责提出分类层次并对它持续地修正。

第二个问题也可以通过图书馆管理员手工地解决，或者由 Web 站点维护人员负责决定他们的站点应该处于目录层次中的什么位置。也有一些基于计算文档与已经分类的文档的相似性来自动确定该文档的位置的技术。

维基百科(wikipedia)是一部在线的百科全书。它从相反的方向解决分类问题。维基百科中的每个页面都包含一个所属**类别**(category)的列表。例如，截 2009 年所见，维基百科上关于长颈鹿的页面的一个所属类别是"非洲哺乳动物"。而"非洲哺乳动物"本身又包含于"按地理划分哺乳动物"类别中。后者又包含于属于"脊椎动物"的"哺乳动物"类别中，以此类推。类别结构对于浏览同一类别中的其他实例是很有用的。例如，查找其他的非洲哺乳动物或者其他的哺乳动物。相反地，一个查找哺乳动物的查询可以使用这种类别信息来推断出长颈鹿是一种哺乳动物。维基百科的类别结构不是树状的，而几乎是 DAG 形式的。事实上，它也不是一个 DAG，因为其中存在一些循环结构，这很可能反映了分类错误。

21.10 总结

- 信息检索系统用于存储和查询如文档那样的文本数据。与数据库系统相比，它们使用更简单的数据模型，但能在受限的模型里提供更强大的查询能力。
- 查询试图通过指定关键字集合来定位用户感兴趣的文档。用户心里所想的查询往往不能精确地表述，因此，信息检索系统按基于潜在的相关性对答案排名。
- 相关性排名利用多种类型的信息，诸如：
  - 术语频率：每个术语对每份文档的重要性。
  - 逆文档频率。
  - 流行度排名。
- 文档相似性用于检索与一个示例文档相似的文档。余弦度量值用于定义相似度，它基于向量空间模型。
- PageRank 和链接中心/权威页排名是基于指向页面的链接对页面威望度赋值的两种方法。PageRank 度量可以用随机游走模型来直观地理解。锚文本信息也可以用来计算单个关键字意义上的流行度。信息检索系统需要整合多种因素(包括 TF-IDF 和 PageRank)来获得对页面的全局评分。
- 搜索引擎作弊试图使一个页面得到高的(但不是应得的)排名。

936
|
937

- 同义词和多义词使信息检索的任务复杂化。基于概念的查询旨在找到含有指定概念的文档，而与指定该概念所使用的确切的词(以及语言)无关。本体利用诸如 is-a 或 part-of 这样的关系将概念联系起来。
- 倒排索引用来对关键字查询做出应答。
- 查准率和查全率是信息检索系统有效性的两种度量。
- Web 搜索引擎使用爬虫搜索 Web 找到网页，然后分析它们以计算其威望度度量，并为它们建立索引。
- 已经开发出了相关技术能够从文本数据中抽取出结构化信息，在结构化数据之上运行关键字查询，并对用户使用自然语言提出的简单问题给出直接的答案。
- 目录结构和分类用来将文档与其他相似文档归类到一起。

术语回顾

- 信息检索系统
- 关键字搜索
- 全文检索
- 术语
- 相关度排名
  - 术语频率
  - 逆文档频率
  - 相关度
  - 接近度
- 基于相似性的检索
  - 向量空间模型
    - 余弦相似度度量
  - 相关反馈
- 停用词
- 使用超链接的相关度
  - 流行度/威望度
  - 威望度传播
- PageRank
  - 随机游走模型
- 基于锚文本的相关度
- 链接中心/权威页排名
- 搜索引擎作弊
- 同义词
- 多义词
- 概念
- 基于概念的查询
- 本体
- 语义网络
- 倒排索引
- 误丢弃
- 误舍弃
- 误选中
- 查准率
- 查全率
- 网络爬虫
- 深度 Web
- 查询结果多样化
- 信息抽取
- 问答系统
- 查询结构化数据
- 目录
- 分类层次
- 类别

实践习题

21.1 计算本章的每个实践习题与查询"SQL 关系"之间的相关度(使用术语频率和逆文档频率的定义)。

21.2 假设你想要查找至少包含给定的 n 个关键字中的 k 个的文档。再假设你有一个关键字索引，它给你提供了包含某个特定关键字的文档标识的(有序)列表。给出一个高效算法来找出所需的文档集。

21.3 假定 2^n 矩阵(即使采用邻接表表示)无法放入主存中，怎样实现计算 PageRank 的迭代技术呢?

21.4 一个包含某个词(比如"leopard"(美洲豹))的文档该怎样索引，使得使用更泛化的概念(诸如"carnivore"(食肉动物)或"mammal"(哺乳动物))的查询才能够高效地检索到该文档?你可以假设这个概念分层并不是很深，因此每个概念只有少数几个泛化(然而，一个概念通常可以有许多特化)。你也可以假定已经为你提供了一个函数，可以对于一份文档中的每个词返回其概念。另外，一个使用特化概念的查询怎样才能检索到使用泛化概念的文档?

938
/
939

21.5 假设倒排表是在块中维护的，每个块记录了表中余下的块中文档的最高的流行度排名以及 TF-IDF 得分。如果用户仅仅只想要排名最高的 k 个答案，怎样做才能使得倒排表的合并过程早些停止?

习题

21.6 简单地将术语频率定义为一份文档中该术语出现的次数，请给出由本习题以及下一道习题组成的文档集合中每一个术语的 TF-IDF 得分。

21.7 举个小例子，要包括 4 个具有 PageRank 的小文档，并对这些用 PageRank 排名的文档创建倒排表。你不必计算 PageRank，只需要假设对于每个页面都有某些值即可。

21.8 假设你想要在数据库的一个元组集合之上进行关键字查询，其中每个元组只有几个属性，每个属性只含有几个词。在这样的上下文中，术语频率的概念有意义吗？逆文档频率的概念呢？请解释你的答案。也请考虑一下怎样使用 TF-IDF 概念来定义两个元组的相似性？

21.9 希望做些宣传和推广的 Web 站点可以加入一个 Web 圈子中，它们建立指向圈中其他站点的链接，以换取圈中其他的站点建立指向它们的链接。请问：类似这样的圈子对于诸如 PageRank 这样的流行度排名技术有什么影响？

21.10 Google 搜索引擎提供了一个特性，就是 Web 站点能够显示由 Google 提供的广告。这些提供的广告是基于页面内容的。请问：给定页面的内容，Google 可能会怎样为一个页面挑选该提供哪些广告呢？

21.11 一种创建 PageRank 的特定关键字版本的方法是修改随机跳转，从而使得跳转只可能跳转到含有关键字的页面。这样，那些不含关键字，但离含有关键字的页面很近（用链接来度量）的页面也会得到对应那个关键字的非零排名值。

a. 给出定义这样一个特定关键字版本的 PageRank 的公式。

b. 给出计算一个页面与含有多个关键字的查询之间的相关度的公式。

21.12 可以利用外键以及 IDREF 边取代超链接，从而将使用超链接进行流行度排名的想法扩展到关系数据和 XML 数据上。请问：这样的排名模式怎样才能够在下述应用中有价值？

a. 一个文献目录数据库，其中含有从文章到文章作者的链接以及从每篇文章到其引用的每篇文章的链接。

b. 一个销售数据库，其中含有从每条销售记录到销售出去的物品的链接。

还请回答：为什么在一个记录了哪个演员出演了哪些影片的电影数据库中，威望度排名只能给出几乎没有意义的结果？

940

21.13 误选中和误舍弃有何不同？如果一个信息检索查询不允许遗漏任何相关信息，含有误选中或误舍弃是可接受的吗？为什么？

工具

Google（www.google.com）是目前最流行的搜索引擎，但还有许多其他的搜索引擎，诸如 Microsoft Bing（www.bing.com）和 Yahoo! 搜索（search.yahoo.com）。站点 searchenginewatch.com 提供了有关搜索引擎的各种信息。Yahoo!（dir.yahoo.com）和开放目录项目（dmoz.org）提供了 Web 站点的分类层次。

文献注解

Manning 等[2008]、Chakrabarti[2002]、Grossman 和 Frieder[2004]、Witten 等[1999]，以及 Baeza-Yates 和 Ribeiro-Neto[1999]提供了描述信息检索的教科书。特别是，Chakrabarti[2002]和 Manning 等[2008]详细覆盖了网络爬虫、排名技术以及其他有关信息检索的挖掘技术（如文本分类和聚类）。

Brin 和 Page[1998]剖析了包括 PageRank 技术在内的 Google 搜索引擎，而称为 HITS 的基于链接中心和权威页的排名技术由 Kleinberg[1999]阐述。Bharat 和 Henzinger[1998]阐述了对 HITS 排名技术的改进。这些技术以及其他基于流行度的排名技术（和用来避免搜索引擎作弊的技术）在 Chakrabarti[2002]中有详细阐述。Chakrabarti 等[1999]说明了针对 Web 站点的爬虫搜索，用来找出与某个特定主题有关的页面。Chakrabarti[1999]提供了一个关于 Web 资源发现的综述。

Witten 等[1999]详细介绍了文档索引。Jones 和 Willet[1997]是一本信息检索方面的论文选集。Salton[1989]是一本有关信息检索系统的早期教科书。Brin 和 Page[1998]讨论了在页面的排名和索引中存在的许多现实问题（比如 Google 搜索引擎的早期版本中的解决方法）。遗憾的是，当今领先的搜索引擎并没有公开它们进行页面排名的方法细节。

Citeseer 系统（citeseer.ist.psu.edu）维护了一个关于出版物（文章）的非常大的数据库，以及出版物之间的引用链接，并使用引用来对出版物排名。它包括一个用来基于出版物的年份调整引用排名的技术，作为对随着时间的流逝，一个出版物的引用自然会增加这个事实的补偿。如果不进行调整，更古老的文档会得到一个比它们真正应得的排名更高的排名。Google Scholar（scholar.google.com）提供了一个包含引用关系类似的可检索的学术论文数据库。值得注意的是，这些系统利用信息抽取技术来推断标题、作者列表以及文末的引文信息。然后，它们通过（近似）匹配文章标题以及作者列表来创建论文之间的引用链接。

信息抽取和问答系统在人工智能领域已经有相当长的一段历史了。Jackson 和 Moulinier[2002]提供了教科书式的涵盖了自然语言处理技术的描述，并强调了信息抽取。Soderland[1999]讲述了使用 WHISK 系统进行信息抽取，同时，Appelt 和 Israel[1999]提供了一个关于信息抽取的教程。

文本检索年会(TREC)有很多专题。每个专题定义了一个问题和一个基础框架，用来测试问题的解决方案的质量。TREC 的详细信息可以在 trec.nist.gov 上找到。有关 WordNet 的更多信息可以在 wordnet.princeton.edu 和 globalwordnet.org 上找到。Cyc 系统的目标是提供关于人类的人类知识的正式表达。它的知识库包含大量术语和关于每个术语的断言。Cyc 也包括对于自然语言的理解和歧义消解的支持。有关 Cyc 系统的信息可以在 cyc.com 和 opencyc.org 上找到。

Dalvi 等[2009]讨论了网页检索向概念和语义而非关键字演进的观点。国际语义网络年会(ISWC)是一个展示语义网络方面的新进展的重要会议。详见 semanticweb.org。

941
942 Agrawal 等[2002]、Bhalotia 等[2002]以及 Hristidis 和 Papakonstantinou[2002]涵盖了关系数据的关键字查询。XML 数据的关键字查询在 Florescu 等[2000]和 Guo 等[2003]以及其他文献中有所阐释。

第七部分
Part 7

特种数据库

数据库系统在几个应用领域由于关系数据模型后限而受到限制。其结果是，研究人员基于面向对象的方法开发了几个数据模型，来处理这些应用领域的问题。

对象－关系模型在第 22 章中描述，它结合了关系模型和面向对象模型的特性。这一模型提供了面向对象语言的丰富的类型系统，并与关系相结合作为数据存储的基础。它将继承不仅应用于类型，而且应用于关系。对象－关系数据模型提供了从关系数据库平滑迁移的路径，这对关系数据库厂商很有吸引力。其结果是，从 SQL：1999 开始，SQL 标准都在其类型系统中包括了若干面向对象特性，同时继续采用关系模型作为基础的模型。

术语面向对象数据库用来描述这样的数据库系统，它支持来自面向对象程序设计语言的对数据的直接存取，而无须关系查询语言作为数据库界面。第 22 章中也提供了对面向对象数据库的简单介绍。

XML 语言最初设计为给正文文档添加标注信息的一种途径，但是由于它在数据交换中的应用，现在已经变得非常重要。XML 提供对于具有嵌套结构的数据的一种表示方法，而且它允许在数据构造上的很大的灵活性，这对于某些种类的非传统数据是很重要的。第 23 章描述 XML 语言，然后给出对 XML 表示的数据表达查询的不同方法，包括 XQuery XML 查询语言和 SQL 的扩展 SQL/XML，它允许创建嵌套的 XML 输出。

第22章
Database System Concepts, 6E

基于对象的数据库

传统的数据库应用由数据处理任务组成，例如银行业务和工资管理。这些领域只有相对简单的数据类型，非常适合于关系模型。当数据库系统应用到更加广阔的应用领域，如电脑辅助设计和地理信息系统时，关系模型带来的局限就暴露出来，成为了一个障碍。解决的方法是引入允许处理复杂数据类型的基于对象的数据库。

22.1 概述

程序员在使用关系数据模型时要面对的第一个障碍是关系模型支持的有限的类型系统。复杂应用领域需要相应的复杂数据类型，如嵌套的记录结构、多值属性和继承。它们为传统程序设计语言所支持。这些特性事实上都为 E-R 和扩展 E-R 概念所支持，但必须被转化成较为简单的 SQL 数据类型。**对象－关系数据模型**(object-relational data model)通过提供更加丰富的类型系统扩展了关系数据模型，它包括了复杂数据类型和面向对象。关系查询语言，特别是 SQL，需要相应的扩展以处理更丰富的类型系统。这种扩展试图在扩展建模能力的同时保持关系的基础，特别是对数据的声明式访问。**对象－关系数据库系统**(object-relational database system)，也就是基于对象－关系模型的数据库系统，为想要使用面向对象特性的关系数据库用户提供了一个方便的移植途径。

第二个障碍在于，很难通过用 C++ 或 Java 等程序设计语言编写的程序访问数据库中的数据。仅仅扩展数据库支持的类型系统不足以彻底解决这个问题。数据库的类型系统和程序设计语言的类型系统之间的差别使得数据的存储和获取更加复杂，因而需要最小化这种差别。只能使用一种不同于程序设计语言的语言(如 SQL)来表达数据库访问也使得程序员的工作更加艰难。很多应用都需要允许直接访问数据库中数据的程序设计语言结构或扩展，而非通过一个中间语言例如 SQL 来访问。

945

在本章中，我们首先解释发展复杂数据类型的动机。然后我们研究对象－关系数据库系统，特别是用 SQL：1999 和 SQL：2003 引入的新特性。注意，大多数数据库产品只支持这里描述的 SQL 特性的一个子集，而且对于所支持的特性而言，其语法也与标准稍有不同。这是由于标准成文之前，商业数据库系统就开始向市场引入对象－关系特性所造成的。需要查阅所使用的数据库系统的用户手册以发现它支持哪些特性。

接着，我们说明对于面向对象程序设计语言的本地类型系统中的数据的持久化的支持问题。实践中使用了两种方法：

1. 建立**面向对象的数据库系统**(object-oriented database system)，即一个以本地方式支持面向对象类型系统，而且允许面向对象编程语言使用本地语言的类型系统直接访问数据的数据库系统。

2. 自动地将以编程语言的本地类型表示的数据转换为关系数据库的表示形式，反之亦然。数据转换由**对象－关系映射**(object-relational mapping)来说明。

我们将对这两种方法做简要介绍。

最后，我们概述哪些情况下使用对象－关系方法更好，哪些情况下使用面向对象方法更好，并介绍在两者之间进行选择的标准。

22.2 复杂数据类型

传统的数据库应用在概念上只有简单数据类型。基本数据项是相当小的记录，且记录的域是原

子的，即它们没有被进一步结构化，并且第一范式成立（见第 8 章）。而且只有少数几种记录类型。

最近几年，对处理更加复杂的数据类型的方法的需求逐渐增多。例如，考虑地址信息。虽然一个完整的地址可以看成一个字符串类型的原子数据项，但这种方式可能隐藏了详细信息。比如街道地址、城市、州和邮编，这些也可能是查询所感兴趣的内容。另一方面，如果一个地址被分解成几个部分（街道地址、城市、州和邮编）来表示，书写查询将更为复杂，因为要提及每一个域。一个更好的可选择的方法是允许结构化数据类型，即允许 *address* 类型包含子部分 *street_address*、*city*、*state* 和 *postal_code*。

作为另一个例子，考虑 E-R 模型中的多值属性。这样的属性很自然，例如电话号码的表示，因为一个人可能有多个电话。对于这个例子，作为可选方法，通过创建一个新的关系来达到规范化的 [946]
目的是昂贵且不自然的。

利用复杂的类型系统，我们可以直接地表示 E-R 模型中诸如复合属性、多值属性、一般化和特殊化等概念，而不需要翻译转换到关系模型的复杂过程。

在第 8 章中，我们定义了第一范式（1NF），它要求所有的属性具有**原子的域**。回忆一下，如果域的元素被认为是不可分的单元，那么这个域是原子的。

1NF 假设对于我们考虑的数据库应用的例子是合理的。但是并非所有应用都可以用 1NF 关系很好地建模。例如，某些应用的用户将数据库看成是对象（或实体）的集合而不是记录的集合。这些对象可能需要若干条记录来描述。一个简单应用的接口需要用户对对象的直观理解和数据库系统中数据项的概念之间的一一对应。

例如，考虑一个图书馆应用，假设我们希望对每本书存储如下信息：

- 书名。
- 作者列表。
- 出版商。
- 关键字集合。

可以看出，如果我们对上述信息定义一个关系，很多域都是非原子的。

- **作者**。一本书可能有很多作者，我们可以用一个数组来表示。但是，我们可能需要查找所有 Jones 为作者之一的书。因此，我们所感兴趣的是域元素"作者"的一个子部分。
- **关键字**。如果我们为一本书存储一个关键字集合，我们希望可以获取所有包含某个或某几个指定关键字的书。因此，我们将关键字集合这个域看成是非原子的。
- **出版商**。不同于关键字和作者，出版商并没有一个以集合为值的域，但是我们可以将出版商视为由名称和部门两个子域组成的。这种观念使得出版商域成为非原子的。

图 22-1 展示了一个示例关系——*books*。

title	*author_array*	*publisher*	*keyword_set*
		(*name*, *branch*)	
Compilers	{Smith, Jones}	(McGraw-Hill, New York)	{parsing, analysis}
Networks	{Jones, Frick}	(Oxford, London)	{Internet, Web}

图 22-1　非 1NF 的书籍关系——*books*

为了简单起见，我们假设书名唯一标识了一本书。[⊖]那么我们就可以用下面的模式来表达同样的信息，主键属性用下划线标记：

- *authors* (*title*, *author*, *position*)
- *keywords* (*title*, *keyword*)
- *books4* (*title*, *pub_name*, *pub_branch*)

以上的模式满足 4NF。图 22-2 给出了图 22-1 中数据的规范化表示。

尽管不用嵌套关系也足以充分表达我们的示例书籍数据库，但是使用嵌套关系可以产生一个更

⊖　这个假设在现实世界中并不成立。图书一般通过唯一标识每本出版物的 10 位 ISBN 码进行识别。

易于理解的模型。一个信息检索系统的典型用户或程序员根据具有作者信息的书将该数据库看作一个非 1NF 的设计模型。4NF 设计需要在查询时连接多个关系，而非 1NF 设计则使得很多类型的查询更容易。

与此相反，在一些其他情况下使用第一范式形式来表达可能会更好。例如，考虑我们大学例子中的 *takes* 关系。这个关系是 *student* 和 *section* 之间的多对多关系。我们当然可以为每个学生存储一组课程段，或者为每个课程段存储一组学生，或者二者都存储。如果两个都存储，将会出现数据冗余（某个特定学生和某个特定课程段之间的关系将会被存储两次）。

使用如集合、数组的复杂数据类型的能力在很多应用里是有用的，但必须小心使用。

title	*author*	*position*
Compilers	Smith	1
Compilers	Jones	2
Networks	Jones	1
Networks	Frick	2

authors

title	*keyword*
Compilers	parsing
Compilers	analysis
Networks	Internet
Networks	Web

keywords

title	*pub_name*	*pub_branch*
Compilers	McGraw-Hill	New York
Networks	Oxford	London

books4

图 22-2 *books* 关系的 4NF 版本

22.3 SQL 中的结构类型和继承

在 SQL：1999 之前，SQL 的类型系统由一个很简单的预定义的类型的集合构成。SQL：1999 为 SQL 增加了一个扩展类型系统，允许结构类型和类型继承。

22.3.1 结构类型

结构类型允许直接表示 E-R 设计中的复合属性。例如，我们可以定义如下结构类型以表示一个由成分属性 *firstname* 和 *lastname* 构成的复合属性 *name*：

```
create type Name as
    (firstname varchar(20),
     lastname varchar(20))
    final;
```

类似地，下面的结构类型可以用于表示一个复合属性 *address*：

```
create type Address as
    (street varchar(20),
     city varchar(20)
     zipcode varchar(9))
    not final;
```

这种类型在 SQL 中被称作用户定义（user-defined）类型[1]。上面的定义对应图 7-4 中的 E-R 图。语句中指明的 **final** 和 **not final** 与子类型定义相关，我们稍后将在 22.3.2 节讲述。[2]

我们现在可以使用这些类型在关系里创建复合属性，只需简单地声明一个属性为这样一种类型。例如，我们可以创建如下的一个 *person* 表：

```
create table person(
    name Name,
    address Address,
    dateOfBirth date);
```

复合属性的成分可以通过"点"记号来访问；例如 *name.firstname* 返回姓名属性的"名"成分。对属性 *name* 的访问将会返回一个结构类型 *Name* 的值。

947
949

我们也可以创建一个表，表的行是用户定义的类型。例如，我们可以定义一个 *PersonType* 类型，并创建一个 *person* 表如下[3]：

---

[1] 为了解释我们早先关于在标准推出之前商业数据库系统定义专门的方法的注释，我们指出，Oracle 要求在 **as** 之后应当有关键字 **object**。

[2] 对 *Name* 注明 **final** 表示我们不能为 *name* 创建子类型，而对 *Address* 注明 **not final** 表示我们可以为 *address* 创建子类型。

[3] 事实上，在大多数系统中，对大小写不敏感，将不允许 *name* 被同时用作属性名和数据类型。

```
create type PersonType as (
    name Name,
    address Address,
    dateOfBirth date)
    not final
create table person of PersonType;
```

在 SQL 中定义复合属性的一种可选方法是使用无名称的**行类型**(row type)。例如，表示人的信息的关系可以使用如下的行类型进行创建：

```
create table person_r (
    name row (firstname varchar(20),
              lastname varchar(20)),
    address row (street varchar(20),
                 city varchar(20),
                 zipcode varchar(9)),
    dateOfBirth date);
```

这个定义与前面的表定义是等价的，只是这里 *name* 属性和 *address* 属性有无名称的类型。表的行也有一个无名称的类型。

下面的查询示例了如何访问一个复合属性的成分属性。这个查询找出每个人的姓和城市。

```
select name.lastname, address.city
from person;
```

结构类型上可以定义**方法**(method)。我们将方法的声明作为结构类型的类型定义的一部分： 950

```
create type PersonType as (
    name Name,
    address Address,
    dateOfBirth date)
    not final
method ageOnDate(onDate date)
    returns interval year;
```

我们单独地创建方法的主体：

```
create instance method ageOnDate(onDate date)
    returns interval year
    for PersonType
begin
    return onDate - self.dateOfBirth;
end
```

注意 **for** 子句指出这个方法是对哪个类型定义的，关键字 **instance** 指出这个方法在 *Person* 类型的实例上执行。变量 **self** 代表正在调用方法的 *Person* 实例。方法的主体可以包含过程语句，我们在前面的 5.2 节已经看到过了。方法中可以更新正在执行该方法的实例的属性。

可以在类型的实例上调用方法。如果我们已经创建了一个 *PersonType* 类型的表 *person*，我们可以像下面这样调用 *ageOnDate*( ) 方法来查询每个人的年龄：

```
select name.lastname, ageOnDate(current_date)
from person;
```

在 SQL：1999 中，**构造器函数**(constructor function)用来创建结构类型的值。与结构类型同名的函数就是这个结构类型的构造器函数。例如，我们可以像这样为 *Name* 类型声明一个构造器：

```
create function Name(firstname varchar(20), lastname varchar(20))
returns Name
begin
    set self.firstname = firstname;
    set self.lastname = lastname;
end
```

然后我们就可以用 **new** *Name*('John', 'Smith') 创建 *Name* 类型的一个值。我们可以通过在圆括

951 号中列出其属性来构造一个行类型的值。例如，如果我们声明 *name* 属性为一个由 *firstname* 和 *lastname* 成分组成的行类型，我们可以为它创建这个值：('Ted', 'Codd')，而不需使用构造器。

默认的情况下，每一个结构类型都有一个不带参数的构造器，它将属性设为默认值。任何其他的构造器必须被显式地创建。同一个结构类型可以有不止一个构造器；虽然它们有相同的名字，但是它们必须以参数的个数和类型来相互区分。

下面的语句示例了如何在 *Person* 关系中创建一个新的元组。假设类似于我们为 *Name* 做出的定义，*Address* 上已经定义了一个构造器。

```
insert into Person
  values
    ( new Name('John', 'Smith'),
    new Address('20 Main St', 'New York', '11001'),
    date '1960-8-22');
```

22.3.2 类型继承

假定有如下的关于人的类型定义：

```
create type Person
    (name varchar(20),
    address varchar(20));
```

我们可能希望在数据库中为那些是学生和教师的人存储一些额外的信息。由于学生和教师也同样是人，我们就可以在 SQL 中使用继承来定义学生和教师类型：

```
create type Student
    under Person
    (degree varchar(20),
    department varchar(20));
create type Teacher
    under Person
    (salary integer,
    department varchar(20));
```

Student 和 *Teacher* 都继承了 *Person* 的属性，即 *name* 和 *address*。*Student* 和 *Teacher* 被称为 *Person* 的子类型，*Person* 是 *Student* 的父类型，同时也是 *Teacher* 的父类型。

像属性一样，结构类型的方法也被它的子类型继承。不过，子类型可以通过在方法声明中用 952 **overriding method** 代替 **method** 来重新声明该方法，以重定义该方法的效用。

SQL 标准还需要在类型定义的尾部有一个额外的域，取值为 **final** 或 **not final**。关键字 **final** 表示不能从给定类型创建子类型，而 **not final** 表示可以创建子类型。

现在假定我们要存储关于助教的信息，这些助教同时既是学生又是教师，甚至可能是在不同的系里。如果类型系统支持**多重继承**(multiple inheritance)，即一个类型可以被声明为多个类型的子类型，我们就可以做到这一点。注意 SQL 标准并不支持多重继承，尽管将来的 SQL 标准可能支持。因此这里我们仅仅在概念层面加以讨论。

例如，如果我们的类型系统支持多重继承，我们可以试着为助教定义一个类型如下：

```
create type TeachingAssistant
    under Student, Teacher;
```

TeachingAssistant 将继承 *Student* 和 *Teacher* 的所有属性，但是却有一个问题，因为 *name*、*address* 和 *department* 属性同时存在于 *Student* 和 *Teacher* 中。

name 和 *address* 属性实际上是从同一个来源——*Person* 继承来的，因此同时从 *Student* 和 *Teacher* 中继承这两个属性不会引起冲突。然而 *department* 属性是在 *Student* 和 *Teacher* 中分别定义的。事实上，一个助教可能是某个系的学生同时又是另一个系中的教师。为了避免两次出现的 *department* 之间的冲突，我们可以使用 **as** 子句将它们重新命名，如下面的 *TeachingAssistant* 类型定义所示：

```
create type TeachingAssistant
    under Student with (department as student_dept),
        Teacher with (department as teacher_dept);
```

在 SQL 中，和在大多数其他的语言中一样，一个结构类型的值必须恰好只有一个最明确类型，即每一个值在创建时必须被关联到一个确定的类型，称为它的**最明确类型**(most-specific type)。通过继承，它也与其最明确类型的每个父类型相关联。举例来说，假定一个实体具有类型 *Person*，同时又具有类型 *Student*，那么这个实体的最明确类型为 *Student*，因为 *Student* 是 *Person* 的子类型；然而一个实体不能同时既具有类型 *Student* 又具有类型 *Teacher*，除非这个实体具有一个如 *TeachingAssistant* 那样既是 *Student* 的子类型又是 *Teacher* 的子类型的类型(这在 SQL 中是不可能的，因为 SQL 不支持多重继承)。 953

22.4　表继承

SQL 中的子表对应 E-R 概念中的特殊化/一般化。例如，假设我们如下定义 *people* 表：

```
create table people of Person;
```

则我们可以如下定义表 *students* 和 *teachers* 作为 *people* 的**子表**(subtable)：

```
create table students of Student
    under people;
create table teachers of Teacher
    under people;
```

子表的类型(在上例中是 *Student* 和 *Teacher* 表)必须是父表类型(在以上例子中是 *Person* 表)的子类型，因此，出现在表 *people* 中的每一个属性也出现在其子表 *students* 和 *teachers* 中。

进一步说，当我们声明 *students* 和 *teachers* 作为 *people* 的子表时，*students* 或 *teachers* 中出现的每一个元组也隐式地存在于 *people* 中。所以，如果一个查询用到 *people* 表，它将查找到的不仅仅是直接插入到这个表中的元组，而且还包含插入到它的子表中的元组，也就是 *students* 和 *teachers* 中的元组。然而，这个查询只能访问那些出现在 *people* 中的属性。

SQL 允许我们在查询中使用"**only** *people*"代替 *people* 来查找只在 *people* 中而不在它的子表中的元组。关键字 **only** 还可以在 delete 和 update 语句中使用。如果不使用 **only** 关键字，超表(例如 *people*)上的删除语句还会删除原先插入到子表(例如 *students*)中的元组；例如，语句

```
delete from people where P;
```

将删除 *people* 表中所有满足 *P* 的元组，以及子表 *students* 和 *teachers* 中所有满足 *P* 的元组。如果在上面的语句中增加了 **only** 关键字，插入到子表中的元组不会受到影响，即使它们满足 **where** 子句。以后对超表的查询将仍然能够查找到这些元组。

概念上，多重继承对表来说也是可能的，正如对于类型是可能的一样。例如，我们可以创建一个类型为 *TeachingAssistant* 的表：

```
create table teaching_assistants
of TeachingAssistant
    under students, teachers;
```
954

作为声明的结果，每一个在 *teaching_assistants* 中出现的元组也隐式地在表 *teachers* 和 *students* 中出现，从而也出现在 *people* 表中。但是我们注意到 SQL 并不支持表的多重继承。

对子表有一些一致性要求。在陈述约束之前，我们需要一个定义：如果子表和父表中的元组在所有的继承属性上具有同样的值，则称子表中的元组与父表中的元组**对应**(correspond)。因此，相对应的元组表示同一个实体。

子表的一致性要求为：

1. 父表的每个元组至多可以与它的每个直接子表的一个元组相对应。

2. SQL 有一个附加的约束，所有互相对应的元组必须由同一个元组派生出来(插入到一个表中)。

例如，若没有第一个条件，我们就可能在 *students*(或 *teachers*)中有两个元组与同一个人相对应。

第二个条件排除了 *people* 中的一个元组同时对应 *students* 中的一个元组和 *teachers* 中的一个元组的

情况，除非所有这些元组都以元组出现(这是由于一个元组被插入到同时是 *teachers* 和 *students* 的子表的 *teaching_assistants* 表中)。

由于 SQL 不支持多继承，所以第二个条件实际上阻止了一个人既是教师又是学生的情况。即使支持多继承，这个问题在没有子表 *teaching_assistants* 时也会出现。显然，即使没有公共子表 *teaching_assistants*，为可以让一个人既是教师又是学生的情况建模也是很有用的。因此，去掉第二个条件是有用的。这样会在不要求对象必须有一个最明确类型的情况下，允许它具有多个类型。

例如，仍然假定我们有具有子类型 *Student* 和 *Teacher* 的类型 *Person*，以及相应的具有子表 *teachers* 和 *students* 的表 *people*。然后我们就能够使 *teachers* 中的一个元组和 *students* 中的一个元组与 *people* 中的同一个元组相对应。这就不需要 *Student* 和 *Teacher* 的子类型 *TeachingAssistant* 类型了。我们无需创建一个 *TeachingAssistant* 类型，除非我们希望存储额外属性或以特定于既是学生又是教师的人的方式重定义方法。

但是，遗憾的是，我们注意到，由于一致性要求 2，SQL 禁止这种情况。因为 SQL 也不支持多重继承，对于一个人既是学生又是教师的情况，我们不能用继承来建模。因此，SQL 子表不能用于表示 E-R 模型中的重叠特化。

我们当然可以创建单独的表以表示重叠特化/一般化，而不需要使用继承。其过程在前面 7.8.6.1 小节进行了描述。在上面的例子中，我们可以创建表 *people*、*students* 和 *teachers*，其中
[955] *students* 和 *teachers* 包含 *Person* 类型的主键属性以及 *Student* 和 *Teacher* 类型各自的其他特定属性。*people* 表将含有所有人的信息，包括学生和教师。然后我们增加适当的参照完整性约束以确保学生和教师也在 *people* 表中表示。

换句话说，我们可以利用 SQL 已有的特性，通过一些在定义表和在查询中指定连接以获取需要的属性方面的额外的努力，来构造我们自己的改进的子表机制的实现。

我们注意到 SQL 定义了一个称为 **under** 的特权，当在另外一个类型或表的下面创建一个子类型或者子表时需要这个特权。定义这个特权的动机与 **references** 特权类似。

22.5 SQL 中的数组和多重集合类型

SQL 支持两种集合体类型：数组和多重集合。数组类型是在 SQL: 1999 中增加的，而多重集合类型是在 SQL: 2003 中增加的。回想，多重集合是一个无序集合，一个元素可能在其中出现多次。多重集合与集合相似，区别只在于集合只允许每个元素最多出现一次。

假定我们希望记录有关图书的信息，每本书都包含一个关键字的集合。并且假定我们希望将图书作者的名字以数组的形式存储；不同于多重集合中的元素，数组中的元素是有序的。从而我们可以区分第一作者和第二作者，等等。下面的例子示例了如何在 SQL 中定义这些以数组和多重集合为值的属性：

```
create type Publisher as
    (name varchar(20),
     branch varchar(20));
create type Book as
    (title varchar(30),
     author_array varchar(20) array[10],
     pub_date date,
     publisher Publisher,
     keyword_set varchar(20) multiset);
create table books of Book;
```

第一条语句定义了一个称为 *Publisher* 的类型，它包括两个成分：*name* 和 *branch*。第二条语句定义了一个结构类型 *Book*，包含 *title*、*author_array*、出版日期、出版社(类型为 *Publisher*)和关键字的多重集合，其中 *author_array* 是一个最多可以存储 10 个作者姓名的数组。最后，创建了一个包含类型
[956] 为 *Book* 的元组的 *books* 表。

注意我们使用数组而不是多重集合来存储作者的姓名，因为作者的顺序一般是有意义的，而我

们认为与图书关联的关键字的顺序是没有意义的。

一般而言，E-R 模式中的多值属性可以映射到 SQL 中的以多重集合为值的属性；如果顺序是重要的，可以使用 SQL 数组来代替多重集合。

22.5.1　创建和访问集合体值

在 SQL：1999 中，一个数组的值可以用这种方式创建：

array['Silberschatz', 'Korth', 'Sudarshan']

类似地，关键字的多重集合可以这样构造：

multiset['computer', 'database', 'SQL']

因此，我们可以创建一个 *books* 关系中定义的类型的元组为：

('Compilers', **array** 'Smith', 'Jones'], **new** *Publisher*('McGraw-Hill', 'New York'),
multiset('parsing', 'analysis'])

这里我们已经通过以适当的参数调用 *Publisher* 的构造器函数为 *Publisher* 属性创建了一个值。注意 *Publisher* 的这个构造器函数必须被显示地创建，而不是默认存在的；可以用与前面 22.3 节中我们看到的 *Name* 的构造器函数相似的方式声明它。

如果我们希望将上面的元组插入到 *books* 关系中，我们可以执行这个语句：

```
insert into books
values ('Compilers', array['Smith', 'Jones'],
                     new Publisher('McGraw-Hill', 'New York'),
                     multiset['parsing', 'analysis']);
```

我们可以通过指定数组索引，例如 *author_array*[1]，来访问或更新数组中的元素。

22.5.2　查询以集合体为值的属性

现在我们考虑怎样处理查询中的以集合体为值的属性。一个计算得到集合体的表达式可以出现在关系名可能出现的任何地方，例如像下个段落中展示的那样出现在 **from** 子句中。我们使用先前定义的 *books* 表。

如果我们想找出所有的以"database"为关键字之一的书，我们可以使用这个查询： [957]

```
select title
from books
where 'database' in (unnest(keyword_set));
```

请注意我们使用 **unnest**(*keyword_set*) 的位置，这个位置在无嵌套关系的 SQL 中本来是要求一个 **select-from-where** 的子表达式。

如果我们知道一本特定的书有三个作者，我们会这样写：

```
select author_array[1], author_array[2], author_array[3]
from books
where title = 'Database System Concepts';
```

现在，假定我们想得到一个关系，它包含形式为"书名，作者名"的对，对应每本书和书的每个作者。我们可以使用这样的查询：

```
select B.title, A.author
from books as B, unnest(B.author_array) as A(author);
```

由于 *books* 的 *author_array* 属性是一个以集合体为值的字段，**unnest**(*B.author_array*) 可以用在一个 **from** 子句中，即应该出现一个关系的位置。注意元组变量 *B* 对这个表达式是可见的，因为它先前已经在 **from** 子句中定义了。

当对一个数组解除嵌套时，上面的查询丢失了数组中元素的顺序信息。**unnest with ordinality** 子句可以用于获得该信息，如下面的查询所示。这个查询可以用于从 *books* 关系生成我们以前见到过的 *authors* 关系。

```
select title, A.author, A.position
from books as B,
        unnest(B.author_array) with ordinality as A(author, position);
```

with ordinality 子句生成一个额外的属性记录数组中元素的位置。一个类似的但是不使用 **with ordinality** 子句的查询，可以用于生成 *keyword* 关系。

22.5.3 嵌套和解除嵌套

将一个嵌套关系转换成具有更少（或没有）以关系为值的属性的形式的过程称为解除嵌套（unnesting）。*books* 关系有 *author_array* 和 *keyword_set* 两个属性是集合体，以及 *title* 和 *publisher* 两个属性不是集合体。假定我们想要将该关系转化为一个单一的平面关系，使其不包含该表关系和结构类型作为属性。我们可以使用以下查询来执行这个任务：

[958]

```
select title, A.author, publisher.name as pub_name, publisher.branch
        as pub_branch, K.keyword
from books as B, unnest(B.author_array) as A(author),
        unnest(B.keyword_set) as K(keyword);
```

from 子句中的变量 *B* 被声明为以 *books* 为取值范围。变量 *A* 被声明为以书 *B* 的 *author_array* 中的作者为取值范围，同时 *K* 被声明为以书 *B* 的 *keyword_set* 中的关键字为取值范围。图 22-1 显示了 *books* 关系的一个实例，图 22-3 显示了我们称之为 *flat_books* 的关系，它是上面的查询的结果。注意，*flat_books* 关系是满足 1NF 的，因为它的所有属性都是原子的值。

title	author	pub_name	pub_branch	keyword
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

图 22-3 *flat_books*：对 *books* 关系的 *author_array* 属性和 *keyword_set* 属性解除嵌套的结果

将一个 1NF 关系转化为嵌套关系的反向过程称为**嵌套**（nesting）。嵌套可以用 SQL 中的分组操作的一个扩展来执行。在 SQL 中分组的常规使用中，要对每个组（逻辑上）创建一个临时的多重集合关系，并且在这个临时关系上应用一个聚集函数以获得一个单一（原子）值。**collect** 函数返回值的多重集合，因此我们可以创建一个嵌套关系，不是创建一个单一值。假定给定一个 1NF 关系 *flat_books*，如图 22-3 所示。下面的查询在属性 *keyword* 上对关系进行了嵌套：

```
select title, author, Publisher(pub_name, pub_branch) as publisher,
        collect(keyword) as keyword_set
from flat_books
group by title, author, publisher;
```

在图 22-3 所示的 *flat_books* 关系上执行这个查询的结果如图 22-4 所示。

title	author	publisher	keyword_set
		(pub_name, pub_branch)	
Compilers	Smith	(McGraw-Hill, New York)	{parsing, analysis}
Compilers	Jones	(McGraw-Hill, New York)	{parsing, analysis}
Networks	Jones	(Oxford, London)	{Internet, Web}
Networks	Frick	(Oxford, London)	{Internet, Web}

图 22-4 *flat_books* 关系的一个部分嵌套版本

如果我们还要将作者属性嵌套到多重集合中，我们可以使用查询：

```
select title, collect(author) as author_set,
    Publisher(pub_name, pub_branch) as publisher,
        collect(keyword) as keyword_set
from flat_books
group by title, publisher;
```

另一种创建嵌套关系的方法是在 **select** 子句中使用子查询。子查询方法的一个优势是在子查询中可以使用 **order by** 子句，从而以创建数组所需的顺序来产生结果。下面的查询解释了这种方法；关键字 **array** 和 **multiset** 指明数组和多重集合将用子查询的结果来创建：

```
select title,
    array(select author
          from authors as A
          where A.title = B.title
          order by A.position) as author_array,
    Publisher(pub_name, pub_branch) as publisher,
multiset (select keyword
          from keywords as K
          where K.title = B.title) as keyword_set,
from books4 as B;
```

系统为外层查询的 **from** 和 **where** 子句生成的每个元组执行 **select** 子句中的两个子查询。注意来自外层查询的 *B.title* 属性被用到嵌套查询中，以确保对每个书名仅生成正确的作者和关键字集合。

SQL：2003 在多重集合上提供了多种操作符，包括：函数 **set**(M)，它返回多重集合 M 的一个无重复元组版本；聚集操作 **intersection**，它返回一个组中所有多重集合的交集；聚集操作 **fusion**，它返回一个组中所有多重集合的并集；谓词 **submultiset**，它检查一个多重集合是否包含在另一个多重集合中。

959
960

SQL 标准不提供除了赋予新值以外的任何更新多重集合属性的方法。例如，要从多重集合属性 A 中删除一个值 v，我们必须将其置为（A **except all multiset** v）。

22.6　SQL 中的对象标识和引用类型

面向对象的程序设计语言提供了引用对象的能力。类型的一个属性可以是对一个指定类型的对象的引用。例如，在 SQL 中我们可以定义一个 *Department* 类型，它有一个 *name* 域和一个对 *Person* 类型进行引用的 *head* 域，并且定义一个 *Department* 类型的表 *departments*，如下所示：

```
create type Departmentt(
    name varchar(20),
    head ref(Person) scope people);
create table departments of Departmentt;
```

这里，引用被限制在 *people* 表中的元组。在 SQL 中，对指向一个表的元组的引用范围（scope）的限制是强制的，它使引用的行为与外码类似。

我们可以省略掉类型声明中的 **scope** *people* 声明，而在 **create table** 语句中作补充：

```
create table departments of Departmentt
    (head with options scope people);
```

被引用的表必须有一个属性来存储元组的标识符。我们通过在 **create table** 语句中增加一个 **ref is** 子句来声明这个称为**自引用属性**（self-referential attribute）的属性：

```
create table people of Person
ref is person_id system generated;
```

这里，*person_id* 是一个属性名，不是关键字，而 **create table** 语句指定该标识符是由数据库自动生成的。

为了初始化一个引用属性，我们需要得到被引用元组的标识符。我们可以通过查询得到一个元组的标识符的值。因此，为了创建一个有引用值的元组，我们可以首先创建带有空引用的元组，然后单独地设置其引用：

961

```
insert into departments
    values('CS', null);
update departments
    set head = (select p.person_id
                from people as p
                where name = 'John')
where name = 'CS';
```

与系统生成标识符不同的另一种方法是允许用户生成标识符。自引用属性的类型必须指定为被引用表的类型定义的一部分，而且表定义中必须指定引用是用户生成(user generated)的：

```
create type Person
    (name varchar(20),
     address varchar(20))
    ref using varchar(20);
create table people of Person
    ref is person_id user generated;
```

当向 *people* 中插入元组时，我们必须为标识符提供值：

```
insert into people (person_id, name, address) values
    ('01284567', 'John', '23 Coyote Run');
```

People 表、其父表以及其子表中均不能有其他元组具有同样的标识符。这样我们在向 *departments* 中插入元组时就可以使用标识符的值，而不需要一个单独的查询来获取标识符：

```
insert into departments
    values('CS', '01284567');
```

还有一种可能的方法是通过在类型定义中包含 **ref from** 子句以使用一个已有的主码值作为标识符：

962

```
create type Person
    (name varchar(20) primary key,
     address varchar(20))
    ref from(name);
create table people of Person
    ref is person_id derived;
```

注意表定义必须指明引用是衍生出来的，而且还必须指定一个自引用属性名。这样当向 *departments* 中插入元组时，我们就可以使用：

```
insert into departments
    values('CS', 'John');
```

SQL:1999 中使用 -> 符号对引用取内容。考虑前面定义的 *departments* 表，我们可以使用这个查询来找出各部门负责人的名字和地址：

```
select head->name, head->address
from departments;
```

"*head->name*"这样的表达式称为**路径表达式**(path expression)。

由于 *head* 是对 *people* 表中一个元组的引用，上述查询中的 *name* 属性就是 *people* 表中元组的 *name* 属性。引用可以用来隐藏连接操作；在上面的例子中，如果不使用引用，则 *department* 的 *head* 字段就会被声明为 *people* 表的一个外码。要找出一个部门负责人的姓名和地址，我们就需要显式地连接 *departments* 关系与 *people* 关系。使用引用大大简化了查询。

我们可以使用 **deref** 操作来返回引用所指向的元组，并接着访问它的属性，如下所示：

```
select deref(head).name
from departments;
```

22.7 O-R 特性的实现

对象-关系数据库系统基本上是对已有关系数据库系统的扩展。在数据库系统的很多层次上都

明显需要改变。但是，为了使对存储系统的代码（关系存储、索引等）的改动最小化，对象－关系系统支持的复杂数据类型可以转化为关系数据库的较简单的类型系统。

为了理解如何进行转换，我们只需查看 E-R 模型的某些特性是如何转换成关系的。例如，E-R 模型中的多值属性对应于对象－关系模型中的以多重集合为值的属性。复合属性大致对应于结构类型。E-R 模型中的 ISA 层次对应于对象－关系模型中的表继承。

我们在 7.6 节中看到的将 E-R 模型的特性转换到表的技术，可以通过某些扩展用于在存储层将对象－关系数据转换成关系型的数据。 963

子表可以在无需复制所有继承字段的情况下，以如下两种方式之一有效地存储：

- 每一个表只存储主码（可能是从父表中继承下来的）和局部定义的属性。继承属性（主码以外的）不需要存储，可以通过基于主码的与其父表的连接得到。
- 每一个表存储所有继承的和局部定义的属性。当插入一个元组时，它仅仅存储在它所插入的那个表中，它的每个父表通过推断判断它的存在。因为不需要连接，所以可以更快地访问元组的所有属性。

  但是，一旦类型系统允许一个实体出现在两个子表中而不出现在这两个子表的公共子表中，这种表达可能造成信息的重复。更进一步，很难将指向父表的外码转化为子表上的约束；为了有效地实现这种外码，父表必须被定义为视图，且数据库系统必须支持视图上的外码。

实现时可以选择直接表示数组和多重集合，或者选择在内部使用一个标准化表达。标准化表达倾向于占用更多的空间，且需要一个额外的连接/分组代价以收集数组或多重集合中的数据。但是，标准化表达可能更加易于实现。

ODBC 和 JDBC 应用程序接口已被扩展以获取和存储结构类型。JDBC 提供一个 getObject( ) 方法，它类似于 getString( )，但是它返回一个 Java Struct 对象，从中可以抽取出结构类型的成分。还可以将一个 Java 类与一个 SQL 结构类型关联，然后 JDBC 将在类型之间进行转换。详细信息见 ODBC 或 JDBC 参考手册。

22.8　持久化程序设计语言

不同于传统程序设计语言，数据库语言直接操纵持久的数据。即使创建数据的程序已经终止，这些数据仍然继续存在。数据库中的关系和关系中的元组都是持久数据的例子。相比之下，传统程序设计语言所直接操纵的唯一一种持久数据就是文件。

对数据库的访问只是现实世界应用中的一个组成部分。即便一种数据操纵语言（比如 SQL）访问数据是相当高效的，仍然需要一种程序设计语言来实现应用中的其他组成部分，如用户界面或与其他计算机的通信。将数据库语言连接到程序设计语言上的传统方法是在程序设计语言中嵌入 SQL。 964

持久化程序设计语言（persistent programming language）是一种结构扩充了的用于处理持久数据的程序设计语言。持久化程序设计语言与嵌入 SQL 的语言可以用至少以下两种方法进行区分：

1. 带有嵌入语言的宿主语言的类型系统通常与数据操纵语言的类型系统有所不同。宿主语言和 SQL 之间的任何类型转化都由程序员负责。要求程序员来执行这项任务有若干缺陷：
   - 对象和元组之间进行转换的代码是在面向对象类型系统之外执行的，因此存在没有检测到的错误的几率更高。
   - 数据库中元组的面向对象格式和关系格式之间的转换需要大量的代码。格式转换代码，以及从一个数据库中装入和抽出数据的代码，可能构成了应用所需全部代码的相当大的比例。

     相比之下，在持久化程序设计语言中，查询语言被完全集成到宿主语言中，共用相同的类型系统。对象在数据库中的创建和存储可以不需要任何显式的类型或格式转换；任何所需的格式转换都被透明地执行。
2. 使用嵌入式查询语言的程序员负责编写从数据库取出数据放入内存的显式的代码。一旦执行了任何数据更新，程序员都必须显式地编写代码将更新后的数据写回到数据库中。

相比之下，在持久化程序设计语言中，程序员可以操作持久数据，而无需显式地编写代码将数据取到内存或写回到磁盘。

965 在这一节，我们描述如何将面向对象程序设计语言(例如 C++ 和 Java)扩展为持久化程序设计语言。这些语言特性允许程序员直接在程序设计语言中操纵数据，而不必通过一个像 SQL 那样的数据操纵语言。这样，它们能够比其他方式，如嵌入 SQL，提供程序设计语言与数据库的更加紧密的集成。

不过，持久化程序设计语言也有一定的不足之处，当我们要决定是否使用它时必须记住这一点。由于程序设计语言通常很强大，因此发生破坏数据库的编程错误相对容易。语言的复杂性使得自动的高级优化(例如减少磁盘 I/O)更加困难。对于许多应用来说，支持声明性查询是很重要的，但是目前持久化程序设计语言并不能很好地支持声明性查询。

在这一节，我们描述对已有的程序设计语言增加持久化时必须解决的若干概念问题。我们首先解决独立于语言的问题，并在接下来的小节中讨论 C++ 语言和 Java 语言的特有问题。但是，我们没有覆盖语言扩展的细节；尽管已经有若干标准提出，但没有一个是被普遍认可的。请参考文献注释中的参考文献以了解更多具体语言扩展和进一步的实现细节。

22.8.1 对象的持久化

面向对象程序设计语言中已经有了对象的概念、用于定义对象类型的类型系统以及用于创建对象的结构。不过，这些对象是临态的——程序终止时随之消亡，正如 Java 或 C 程序中的变量在程序结束后消亡一样。如果我们想要将这样的一种语言变成数据库程序设计语言，第一步就是要提供一种方法使对象持久化。人们已经提出了许多方法。

- **按类持久。**一种最简单但最不方便的方法是声明一个类为持久的。那么该类的所有对象默认情况下都是持久对象，且持久类的对象都是瞬态的。

  由于经常需要在单个类中同时有瞬态对象和持久对象，因此这种方法是不灵活的。在很多面向对象数据库系统中，声明一个类是持久的被解释为该类中的对象可以被持久化，而不是该类的所有对象都是持久的。这些类或许更适合被称作"可持久的"类。
- **按创建持久。**这种方法通过扩展创建瞬态对象的语法，引入用于创建持久对象的新语法。因此，一个对象是持久的还是瞬态的，取决于它们是如何创建的。有些面向对象数据库系统采用了这种方法。
- **按标志持久。**这是上面方法的一个变体，是在对象被创建后将它们标志为持久的。所有的对象都被创建为瞬态对象，但是，如果一个对象要在程序结束后持久存在，那么必须在程序终止前显式地将这个对象标志为持久的。与前一种方法不同，这种方法将决定对象是持久的还是瞬态的推迟到对象创建以后。
- **按可达性持久。**一个或多个对象被显式地声明为(根)持久对象。对于所有其他对象来说，当(且仅当)它们从根结点通过由一个或更多引用构成的序列可达时，它们才是持久的。

  因此，所有被根持久对象引用的对象(即其对象标识符存储在根持久对象中)是持久的。同时，所有被这些对象引用的对象也是持久的，且它们所引用的对象也是持久的，依此类推。

  这种方案有一个好处是可以很容易将整个数据结构变成持久的，只需声明这种结构的根为持久的就可以了。然而，数据库系统就需要面临需要沿着引用链查找以检测对象是否持久的
966 负担，而其代价可能是很高的。

22.8.2 对象标识和指针

在一个尚未被扩展以处理持久化的程序设计语言中，当创建一个对象时，系统返回一个瞬态对象的标识符。瞬态对象标识符只有当创建它的程序正在执行时才是有效的；程序结束之后，对象就被删除了，同时其标识符也就没有任何意义了。当一个持久对象被创建时，它被赋予一个持久对象标识符。

对象标识的概念与程序设计语言中的指针之间存在有趣的关系。要得到内置标识，最简单的方法是通过指向存储空间物理位置的指针。特别是，在许多面向对象语言（例如 C++）中，隐含对象标识符实际上就是一个内存指针。

然而，一个对象与存储空间物理位置之间的关联可能随时间发生改变，标识有多种持久程度：

- **过程内部**。标识仅在单个过程的执行期间保持。过程内部标识的例子是过程内的局部变量。
- **程序内部**。标识仅在单个程序或查询的执行期间保持。程序内部标识的例子是程序设计语言中的全局变量。主存或虚拟内存指针只提供程序内部标识。
- **程序之间**。从一个程序执行到另一个程序执行标识都会保持。指向磁盘上的文件系统数据的指针提供了程序之间的标识，但是当数据在文件系统中的存储方式发生变化时，它们有可能改变。
- **持久的**。标识的保持不仅仅跨越了各个程序的执行，还跨越了数据的结构重组。这种持久化正是面向对象系统所要求的。

在诸如 C++ 这样的语言的持久化扩展中，持久对象的对象标识符用“持久化指针”来实现。与内存指针不同，持久化指针即使在程序执行结束后，并且跨越某些形式的数据重组，仍然是有效的。程序员可以像使用程序设计语言中内存指针一样使用持久化指针。从概念上，我们可以将持久化指针看作指向数据库中某个对象的指针。

22.8.3　持久对象的存储和访问

在数据库中存储一个对象的含义是什么？显然，对象的数据部分必须要对每个对象单独进行存储。在逻辑上，实现类的方法的代码应该与类的类模式定义在一起，作为数据库模式的一部分存储在 967
数据库中。然而，许多实现简单地将代码存储在数据库之外的文件中，以避免将如编译器这样的系统软件集成到数据库系统中。

在数据库中寻找对象的方法有好几种。一种方法是为对象命名，正如我们为文件命名一样。这种方法适用于对象数目相对较少的情况，而不能扩展到上百万个对象上。第二种方法是将对象标识符或指向对象的持久化指针暴露出来，使它们可以在外部存储。与名称不同，这些指针不必是易于记忆的，它们甚至可能就是指向数据库内部的物理指针。

第三种方法是存储对象的集合体，并允许程序在集合体上迭代寻找所需对象。对象的集合体本身可以被建模为集合体类型的对象。集合体类型包括集合、多重集合（即同一个值可能出现多次的集合）、列表等。集合体的一种特殊情形是类区间（class extent），它是属于该类的所有对象的集合体。当一个类存在类区间时，那么该类的一个对象一旦被创建，该对象会被自动插入到类区间中，并且一旦对象被删除，该对象也会被从类区间中删除。由于我们能检查类中的所有对象，正如我们可以检查一个关系中的所有元组一样，类区间允许像关系一样对待类。

多数面向对象数据库系统都支持以上三种访问持久对象的方法。它们将标识符赋予所有的对象。一般只对类区间和其他集合体对象以及其他所选择的对象赋予名字，但对绝大多数对象并没有命名。它们通常为那些可以有持久对象的类维护类区间，但是在很多实现中，类区间只包括类的持久对象。

22.8.4　持久化 C++ 系统

已经有一些基于 C++ 的持久化扩展的面向对象数据库（参见文献注解）。它们在系统架构上有一定的差异，然而从程序设计语言的角度来说它们有很多公共的特性。

C++ 语言的一些面向对象特性为持久化提供支持，而无需改变语言本身。例如，我们可以声明一个名为 Persistent_Object 的类，它具有一些属性和方法来支持持久化；其他任何应该持久的类可以成为这个类的子类，从而继承对持久化的支持。C++ 语言（像其他现代程序设计语言一样）也允许我们根据运算对象的类型重新定义标准函数名和操作符，如 +、−、指针内容操作符（->），等等。这种能力被称为重载；它用于重定义操作符，使得当这些操作符在持久对象上操作时能够按所需要的方式工作。

通过类库来提供持久化支持具有只需对 C++ 做极少的必要修改的优点，而且相对容易实现。然而，它也有缺陷，程序员必须要花更多的时间书写处理持久对象的程序，并且很难在模式中指定完整 968

强约束或提供对声明性查询的支持。一些持久化 C++ 的实现支持对 C++ 语法的扩展以使这些工作更简单一些。

在向 C++（和其他语言）增加持久化支持时，需要注意以下这些方面：

- **持久指针**：必须定义一个新的数据类型以表示持久指针。例如，ODMG C++ 标准定义了一个模板类 d_Ref<T> 以表示指向类 T 的持久指针。在这个类上的取值操作符被重定义为从磁盘中（如果还没有存在于内存中）获取对象，并返回一个指向缓冲区的内存指针，该缓冲区是获取对象的位置。因此如果 p 是一个指向类 T 的持久指针，可以使用标准语法如 p->A 或者 p->f() 来访问类 T 的属性 A 或者调用类 T 的方法 f。

  ObjectStore 数据库系统使用一个不同的方法来实现持久指针。它使用一般的指针类型来存储持久化指针。这带来了两个问题：（1）内存指针大小可能只有 4 字节，对于大于 4G 的数据库而言太小了；（2）当一个对象在磁盘中移动，指向其旧物理位置的内存指针就没有意义了。ObjectStore 使用一种称为"hardware swizzling"的技术解决这两个问题：它从数据库预取对象到内存中，并用内存指针替换持久指针，当数据被存储回磁盘时，用持久指针替换内存指针。当数据在磁盘上时，存储在内存指针域的值并不是真正的持久指针，而应该用这个值在一个包含完整的持久指针的值的表中查找。

- **持久对象的创建**：C++ 的 new 操作符用于通过定义一个"重载的"版本的操作符（用额外的参数指定它应该被创建在数据库中）来创建持久对象。因此应该调用 new (db) T() 来创建一个持久对象，而不是用 new T()，其中 db 标识数据库。

- **类区间**：每个类的类区间是自动创建和维护的。ODMG C++ 标准要求类的名字以附加参数的形式传递给 new 操作。这还允许了通过传递不同的名字来维护类的多个区间。

- **联系**：类之间的联系常常通过存储每个对象指向其相关对象的指针来实现。关联到给定类的多个对象的对象将存储一个指针的集合。因此如果一对对象之间存在联系，那么每个对象都应该存储一个指向另一个对象的指针。持久化 C++ 系统提供了一种指定这种完整性限制，并通过自动创建和删除指针来强制这种完整性限制的方法；例如，如果一个从对象 a 指向对象 b 的指针被创建，则自动向对象 b 增加一个指向对象 a 的指针。

969

- **迭代器接口**：由于程序需要在类成员上迭代，就需要一个接口以在类区间的成员上迭代。迭代器接口还允许指定选择条件，从而只有满足选择谓词的对象才需要被获取。

- **事务**：持久化 C++ 系统提供对启动、提交或回滚一个事务的支持。

- **更新**：对程序设计语言提供持久化支持的一个目标是允许透明的持久化。也就是说，一个操作在对象上的函数应该不需要知道这个对象是持久的；因此，同样的函数可以使用在对象上而不需考虑它们是否是持久的。

  但是存在一个后果问题，那就是很难探知对象在何时完成了更新。一些对 C++ 的持久化扩展要求程序员通过调用一个函数 mark_modified() 显示指定一个对象已被修改了。除了增加了程序员的工作，这种方法还增加了造成数据库损坏的程序错误出现的几率。如果一个程序设计人员忽略了对 mark_modified() 的调用，可能发生这种情况：某个事务发出的更新可能永远不会被传送给数据库，而同一事务的另一个更新被传送了，这将破坏事务的原子性。

  其他系统，例如 ObjectStore，使用操作系统/硬件所提供的对内存保护的支持来检测对内存块的写操作并标记这个块为一个脏块，脏块将在稍后被写到磁盘上。

- **查询语言**：迭代器对简单选择查询提供支持。为了支持更加复杂的查询，持久化 C++ 系统定义了一个查询语言。

很多基于 C++ 的面向对象数据库系统是在 20 世纪 80 年代末到 20 世纪 90 年代初被开发出来的。但是，这种数据库的市场被证实比预期的小很多。因为通过如 ODBC 或 JDBC 的接口使用 SQL 就可以充分满足大多数的应用需求。结果，那个时期开发的很多面向对象数据库系统都已不复存在了。对象数据管理团体（ODMG）在 20 世纪 90 年代定义了对 C++ 和 Java 增加持久化的标准。但是，这个团体在 2002 年左右停止了活动。ObjectStore 和 Versant 是最初的面向对象数据库系统中仍今还存在的。

尽管面向对象数据库系统没有获得人们所希望的商业上的成功，对程序设计语言增加持久化的动机仍然存在。很多有高性能需求的应用运行在面向对象数据库系统上；使用 SQL 会对很多这种系统强加过高的性能代价。使用目前提供了对包括引用在内的复杂数据类型的支持的对象－关系数据库，在 SQL 数据库中存储程序设计语言对象变得更加容易。用对象－关系数据库作为后端的新一代的面向对象数据库系统可能会出现。

970

22.8.5　持久化 Java 系统

最近几年 Java 语言的使用经历了巨大的成长。在 Java 程序中支持数据持久化的需求也相应地增长。为 Java 中的持久化创建一个标准的最初尝试是由 ODMG 协会倡导的；随后协会终止了该工作，而是将其设计迁移到了 **Java 数据库对象**（Java Database Object，JDO）之上。该项研究是由 Sun Microsystems 协助的。

用于 Java 程序对象持久化的 JDO 模型不同于用于 C++ 程序的持久化支持的模型。其特性包括：

- **按可达性持久**：对象并不是在数据库中显式地创建的。显式地注册一个对象为持久的（使用 PersistenceManager 类的 makePersistent( ) 方法）使得这个对象是持久的。另外，任何从一个持久对象可达的对象都是持久的。
- **字节代码加强**：其对象可能被持久化的类在一个配置文件（以 .jdo 为后缀）中指定，而不是在 Java 代码中声明这个类为持久的。执行一个依赖于实现的加强（enhancer）程序，该程序读入配置文件并完成两个任务。第一，它可能在数据库中创建结构以存储类的对象；第二，它修改字节代码（通过编译 Java 程序而生成）以处理与持久化相关的任务。以下是这种修改的一些例子：
  - 任何访问对象的代码可以被修改为首先检查对象是否在内存中，如果不在，则执行一些步骤将其读入内存中。
  - 任何修改对象的代码被修改为额外地记录该对象已被修改。这些代码还可能被修改为保存一个预更新值，以防这个更新需要被撤销（即如果事务被回滚）。

  可能也会执行对字节代码的其他修改。这种字节代码修改是可能的，因为字节代码是跨平台的标准，且比编译过的对象代码包含更多的信息。
- **数据库映射**：JDO 没有定义数据如何存储在后端数据库。例如，一个普遍的场景是在一个关系数据库中存储对象。加强程序可能在数据库中创建一个合适的模式以存储类对象。它到底如何做到这一点是依赖于实现的，且不是由 JDO 定义的。一些属性可能被映射为关系属性，而其他属性可能以序列化的形式存储，被数据库看作一个二进制对象来对待。JDO 实现可能允许通过定义一个合适的映射将已有的关系数据视为对象。

971

- **类区间**：每个声明为持久的类的类区间是自动创建和维护的。所有持久对象被自动增加到对应于其类的类区间中。JDO 程序可能访问一个类区间，并在被选择的成员上迭代。Java 提供的 Iterator 接口可以用于创建类区间上的迭代器，并在类区间的成员上打过。JDO 还允许当在一个类区间上创建迭代器时指定选择条件，只有满足选择条件的对象才能被获取。
- **单个引用类型**：对瞬态对象的引用和对持久对象的引用之间在类型上没有区别。

  一种实现这种指针类型的统一的方法是将整个数据库载入到内存，用内存指针代替所有持久指针。更新完成后，执行相反的过程，将已更新的对象写回到磁盘中。这样的方法对于大型数据库将是非常低效的。

  我们现在描述另一种可选的方法，它允许持久对象在需要时被自动取到内存中，同时允许所有内存对象包含的引用为内存引用。当一个对象 A 被取回，为其引用的每个对象 B_i 创建一个**空对象**（hollow object），A 的内存副本具有对每个 B_i 对应的空对象的引用。当然，系统必须保证如果一个对象 B_i 已经被取回，引用将指向这个已经取回的对象而不是创建一个新的空对象。类似地，如果一个对象尚未被取回，但它被早先已经取回的另一个对象引用，那么它将已经有了一个为它创建的空对象；对已有空对象的引用被重用，而不是创建一个新的空对象。

  这样，对于每个已经取回的对象 O，O 中的每个引用或者指向一个已经取回的对象，或者指向一个空对象。空对象形成了一个环绕着已取回对象的边缘（fringe）。

一旦程序员试图访问一个空对象 O，加强的字节代码会检测到它，并从数据库中取出对象。当这个对象被取回，对 O 引用的所有对象执行创建空对象的相同过程。此后，对该对象的访问允许被继续。[6]

需要一个将持久指针映射到内存引用的内存索引结构来实现这个方案。在把对象写回到磁盘上时，需要使用这个索引来完成在要写回磁盘的副本中用持久指针代替内存引用的工作。 [972]

22.9 对象－关系映射

到目前为止，我们已经看到了两种将面向对象数据模型和面向对象编程语言整合到数据库系统中的方法。**对象－关系映射**(object-relational mapping)系统提供了把面向对象编程语言和数据库整合起来的第三种方法。

对象－关系映射系统建立于传统关系数据库之上。它允许程序员定义数据库关系中的元组与程序设计语言中的对象之间的映射。不同于持久化程序设计语言，对象是瞬态的，并且没有持久的对象标识。

可以基于属性上的选择条件来获取一个对象或对象的集合；按照选择条件从底层数据库中将相关数据取出，然后，基于预先指定的对象和关系之间的映射，由获取的数据创建出一个或多个对象。程序可以选择性地更新这样的对象、创建新对象或者指定某个对象将被删除。随后发出一个执行存储的命令，从对象到关系的映射将被用于在数据库中相应地更新、插入或者删除元组。

9.4.2节对对象－关系映射系统做了宏观的描述，并特别介绍了已被广泛使用的Hibernate系统，它为Java提供了对象－关系映射。

对象－关系映射系统的首要目标是通过向程序员提供对象模型来减轻他们建造应用的工作，同时保持了在底层使用一个健壮的关系数据库所带来的好处。另一个额外的好处是，在操纵缓存于内存中的对象时，相比直接访问下层数据库，对象－关系系统可以带来巨大的性能提升。

对象－关系映射系统也提供查询语言。程序员可以利用查询语言直接写出对对象模型的查询请求。这种查询被翻译为下层关系数据库上的SQL查询，结果对象也是从SQL查询结果转换而来的。

一个负面影响是，对象－关系映射系统可能会面临大规模数据更新所带来的巨大的开销，而且可能仅提供有限的查询能力。不过，绕过对象－关系映射系统而直接更新数据库，以及直接用SQL书写复杂的查询也都是可行的。对于大多数应用，对象－关系映射系统利大于弊。最近几年内，对象－关系映射系统已被广泛采用。

22.10 面向对象与对象－关系

我们现在已经研究了对象－关系数据库，它是建立在关系模型之上的面向对象数据库；也研究了面向对象数据库，它是建立在持久化程序设计语言基础上的；还研究了对象－关系映射系统，它在传统关系数据库之上构造了一个对象层。 [973]

以上每种方法都定位于不同的市场。SQL语言的声明性本质和有限的能力(与程序设计语言相比)为防止程序设计错误对数据造成破坏提供了很好的保护，同时使得一些高级优化(例如减少I/O)相对容易。(我们已在第13章阐述了关系表达式的优化。)对象－关系系统的目标在于通过使用复杂数据类型使得数据建模和查询更加容易。典型的应用包括复杂数据(包括多媒体数据)的存储和查询。

然而，对于那些主要在主存中运行以及对数据库进行大量访问的特定类型的应用来说，声明性语言(如SQL)会带来显著的性能损失。持久化程序设计语言定位于那些有高性能要求的应用。它们提供了对持久数据的低开销存取，并且消除了数据转换的需求(如果这些数据将要用程序设计语言来进行

---

[6] 上面介绍的使用空对象的技术与hardware swizzling技术(在前面的22.8.4节提到过)紧密相关。一些持久化C++实现使用hardware swizzling以为持久指针和内存指针提供一个单指针类型。hardware swizzling使用操作系统提供的虚拟内存保护技术来检测对页的访问，并在需要时从数据库中获取页。相反，Java版本修改字节代码以将查空对象，而不是使用内存保护，且在需要时获取对象而不是从数据库中获取整个页。

操纵的话）。但是，它们更易于导致由程序错误而引起的数据损坏，并且通常缺乏强大的表达能力。典型的应用包括 CAD 数据库。

对象－关系映射系统允许程序员用对象模型来生成应用程序，同时使用传统数据库系统来存储数据。从而，它们结合了被广泛使用的关系数据库系统的健壮性和用对象模型编写应用的威力。不过，它们需要承受在对象模型和用于存储数据的关系模型之间进行数据转换所带来的运算开销。

我们可以这样总结各种数据库系统的能力：

- **关系系统**：简单数据类型，功能强大的查询语言，高保护性。
- **基于持久化程序设计语言的面向对象数据库**：复杂的数据类型，与程序设计语言集成，高性能。
- **对象－关系系统**：复杂数据类型，强大的查询语言，高保护性。
- **对象－关系映射系统**：集成于程序设计语言中的复杂数据类型，设计为位于关系数据库系统之上的一层。

这些描述一般来说是成立的，但是请记住对有些数据库系统来说这里的分界线是模糊的。例如，以持久化程序设计语言为基础而建立的面向对象数据库系统可以在一个关系数据库系统或对象－关系数据库系统之上实现。这样的系统的性能可能低于那些直接建立在存储系统上的面向对象数据库系统，但它提供了关系系统具有的一些更强的保护性保证。 974

22.11　总结

- 对象－关系数据模型通过提供一个更丰富的类型系统（包括集合体类型和面向对象）来扩展关系数据模型。
- 集合体类型包括嵌套关系、集合、多重集合和数组，对象－关系模型允许表的属性为集合体。
- 面向对象提供了子类型和子表的继承，以及对象（元组）引用。
- SQL 标准包括 SQL 数据定义与查询语言的扩展以处理新的数据类型和面向对象。其中包括对以集合体为值的属性、继承以及元组引用的支持。这种扩展试图在扩展建模能力的同时保持关系的基础，尤其是对数据的声明性的访问。
- 对象－关系数据库系统（即基于对象－关系模型的数据库系统）为希望使用面向对象特性的关系数据库用户提供了方便的移植途径。
- C++ 和 Java 的持久化扩展无缝且正交地将持久化整合到了已有的程序设计语言结构中，因而易于使用。
- ODMG 标准为在 C++ 中创建和访问持久对象定义了类和其他结构，而 JDO 标准为 Java 提供了相同的功能。
- 对象－关系映射系统为存储在关系数据库中的数据提供了对象访问方式。对象是瞬态的，并无持久对象标识的概念。对象是由关系数据按需创建的，而且对对象的更新是通过更新关系数据来实现的。对象－关系映射系统已被广泛采用。相比之下，持久化程序设计语言的应用要少见一些。
- 我们讨论了持久化程序设计语言和对象－关系系统的区别，并且给出了二者的选择标准。

术语回顾

- 嵌套关系
- 嵌套关系模型
- 复杂类型
- 集合体类型
- 大对象类型
- 集合
- 数组
- 多重集合
- 结构类型
- 方法
- 行类型
- 构造器
- 继承
  - 单继承
  - 多重继承
- 类型继承
- 显明的类型
- 表继承
- 子表
- 重叠的子表
- 引用类型
- 引用的范围
- 自引用属性
- 路径表达式

- 嵌套和解除嵌套
- SQL 函数和过程
- 持久化程序设计语言
- 持久化
  - 按类
  - 按创建
  - 按标记
  - 按可达性
- ODMG C++ 绑定
- ObjectStore
- JDO
  - 按可达性持久
  - 版
  - 空对象
- 对象 - 关系映射

实践习题

22.1 一个汽车租赁公司为其当前车队中的所有车辆维护着一个数据库。对于所有的车辆，数据库中包括的信息有车辆识别号、牌照号、制造商、型号、购买日期以及颜色。对于某些类型的车辆还包含特殊的数据：

- 卡车：载货容量。
- 跑车：马力，对租用者的年龄限制。
- 厢式货车：载客数目。
- 越野车：离地距离，驱动系统（四轮或两轮驱动）。

为这个数据库构造 SQL 模式定义，在适当的地方使用继承。

22.2 考虑这样一个数据库模式，关系 *Emp* 的属性如下所示，其类型被指定为多值属性。

Emp = (*ename*, *ChildrenSet* **multiset**(*Children*), *SkillSet* **multiset**(*Skills*))

Children = (*name*, *birthday*)

Skills = (*type*, *ExamSet* **setof**(*Exams*))

Exams = (*year*, *city*)

975
↓
976

a. 用 SQL 定义上面的模式，为每个属性使用适当的类型。

b. 使用上面的模式，用 SQL 写出下面的查询。

i. 找出所有符合条件的职员的名字，其孩子出生在 2000 年 1 月 1 日以后（包括 2000 年 1 月 1 日）。

ii. 找到那些在"Dayton"城市参加过技能类型为"typing"的考试的职员。

iii. 列出关系 *Emp* 中所有的技能类型。

22.3 考虑图 22-5 中的包含成分属性、多值属性和衍生属性的 E-R 图。



图 22-5 包含成分属性、多值属性和衍生属性的 E-R 图

a. 给出对应于这个 E-R 图的 SQL 模式定义。

b. 给出上面定义的每一个结构类型的构造器。

22.4 考虑图 22-6 中的关系模式。

employee (*person_name*, *street*, *city*)
works (*person_name*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*person_name*, *manager_name*)

图 22-6　实践习题 22.4 的关系数据库

a. 给出对应于该关系模式的 SQL 中的一个模式定义，但是要使用引用来表达外码关系。
b. 基于上面的模式使用 SQL 书写习题 6.13 中给出的每个查询。

22.5　假设你受聘为顾问，负责为客户的应用选择一个数据库系统。对于下面这些应用中的每一种，说明你将推荐哪种类型的数据库系统（关系数据库、基于持久化程序设计语言的面向对象数据库、对象－关系数据库，无需指定某个商业产品），并且解释你的推荐的正确性。
a. 为飞机制造商开发的计算机辅助设计系统。
b. 为政府机关设计的对候选人的捐款进行追踪的系统。
c. 支持电影制作的信息系统。

22.6　面向对象模型中对象的概念与实体－联系模型中的实体的概念有什么不同？

习题

22.7　重新设计实践习题 22.2 中的数据库使之满足第一范式和第四范式。列出你所设想的任何函数依赖和多值依赖。同时列出在第一范式和第四范式的模式中应该存在的所有的参照完整性约束。

22.8　考虑实践习题 22.2 中的模式。
a. 给出 SQL DDL 语句来创建关系 *Emp4*，它与关系 *Emp* 具有相同的信息，但是以多重集合为值的属性 *ChildrenSet*、*SkillsSet* 和 *ExamsSet* 被替换为以数组为值的属性 *ChildrenArray*、*SkillsArray* 和 *ExamsArray*。
b. 写出一个查询将数据从 *Emp* 的模式转化为 *Emp4* 的模式，其中 children 数组根据出生日期排序，skills 数组根据技能类型排序，exams 数组根据年排序。
c. 写出 SQL 语句对 *Emp* 关系做如下更新：为名为 George 的职员增加一个出生于 2001 年 2 月 5 日的孩子 Jeb。
d. 写出 SQL 语句实现与上面相同的更新，但是作用在 *Emp4* 关系上。要保证 children 数组仍然根据出生日期排序。

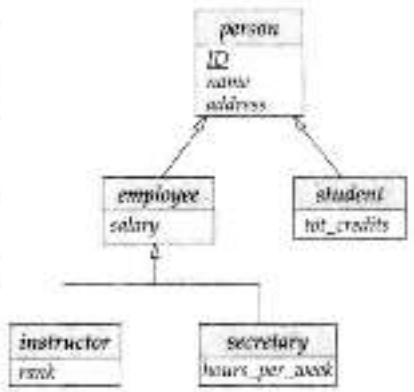


图 22-7　特殊化和一般化

22.9　考虑 22.4 节中的表 *people*，以及继承自表 *people* 的表 *students* 和 *teachers* 的模式。给出一个表示相同信息的满足第一范式的关系模式，请问想子表上的约束，给出为使得关系模式的每个数据库实例也可以用一个带有继承的模式实例所表示，所必须施加在该关系模式上的所有约束。

22.10　解释类型 x 与引用类型 **ref**(x) 之间的区别。什么情况下你会选择使用引用类型？

22.11　考虑图 22-7 中的 E-R 图，其中包含了子类型和子表方式的特殊化结构。
a. 给出该 E-R 图的一个 SQL 模式定义。
b. 给出一个 SQL 查询，找出所有不是秘书的人员的名字。
c. 给出一个 SQL 查询，打印既不是职员又不是学生的人员的姓名。
d. 能否在你创建的模式中创建一个人，使其既是职员又是学生？解释如何创建，否则说明为什么不可能创建。

22.12　设想一个 JDO 数据库有一个对象 A，A 引用对象 B，B 又引用对象 C。假设所有对象初始时都在磁盘上。设想一个程序首先解除引用 A，然后沿着 A 的引用解除引用 B，最后解除引用 C。给出每次解除引用后表示在内存中的对象，以及它们的状态（空或有值，以及它们的引用字段的值）。

977
↓
979

工具

在对对象－关系特性的支持上，不同的数据库产品之间有巨大的差异。Oracle 大概是主要数据库厂商中

具有最多扩展支持的。Informix 数据库系统提供对很多对象－关系特性的支持。Oracle 和 Informix 都是在 SQL：1999 完成之前提供对象－关系特性的，它们还具有一些不属于 SQL：1999 的特性。

关于 ObjectStore 和 Versant 的信息，包括试用版本的下载，可以从它们各自的 Web 站点（objectstore.com 和 versant.com）上获得。Apache DB 项目（db.apache.org）提供了一个 Java 的对象－关系映射工具，它同时支持 ODMG Java 和 JDO API。JDO 的一个参考实现可以从 sun.com 获得；请使用搜索引擎查询完整的 URL。

文献注解

有很多 SQL 的面向对象扩展已经被提出。POSTGRES（Stonebraker 和 Rowe［1986］以及 Stonebraker［1986］）是对象－关系系统的一个早期实现。其他早期对象－关系系统包括 O_2 的 SQL 扩展（Bancilhan 等［1989］）和 UniSQL（UniSQL［1991］）。SQL：1999 是一个扩展的（而且延期很久的）标准化研究的成果，最初由向 SQL 中加入面向对象特性开始，而最终添加了很多其他的特性，例如我们前面看到的程序化结构。对多重集合类型的支持被增加为 SQL：2003 的一部分。

Melton［2002］集中讲述了 SQL：1999 的对象－关系特性。Eisenberg 等［2004］给出了对 SQL：2003 的一个概览，包括它对多重集合的支持。

许多面向对象数据库系统是在 20 世纪 80 年代末到 20 世纪 90 年代初开发的，其中比较有名的商业系统有 ObjectStore（Lamb 等［1991］）、O_2（Lecluse 等［1988］）和 Versant。Cattell［2000］详细地介绍了对象数据库
980 标准 ODMG。Rose［2002］、Tyagi 等［2003］以及 Jordan 和 Russell［2003］介绍了 JDO。

XML

可扩展标记语言(Extensible Markup Language，XML)并不是为数据库应用而设计的。事实上，就像万维网以超文本标记语言(HTML)为基础一样，XML的根源为文档管理，并从一种用来构造大型文档的语言(称为标准通用标记语言(SGML))派生而来。但是，不同于SGML和HTML，XML是用来表示数据的。当一个应用程序必须与另一个应用程序进行通信或是从一些其他的应用程序中整合信息的时候，XML作为一种数据格式特别有用。当XML在这些场合中使用时，会出现许多数据库问题，包括如何组织、操纵和查询XML数据。这一章我们介绍XML，同时讨论使用数据库技术的XML数据管理和XML文档格式数据的交换。

23.1 动机

要理解XML，理解其根源是作为文档标记语言是很重要的。标记(markup)这个术语意指文档中任何不想用于打印输出的部分。例如，一个作者创建了一份最终将要在杂志上出版的文档，他可能想要做一些关于如何排版的注解。采用一种书写这些注解的方式来将注解和实际内容区分开是很重要的，有了这种方式，像"这个词用大号字体，粗体显示"或"这里插入换行"这样的注解就不会被印刷在杂志上。这样的注解传达了关于文档的额外信息。在电子文档处理中，标记语言(markup language)就是对文档的哪部分是内容、哪部分是标记以及标记含义的形式化描述。

正如数据库系统从物理文件处理中演化发展为提供独立的逻辑视图，标记语言从说明如何打印文档各部分的指令演化发展为指定内容的功能。比如，通过功能化标记，表示章节标题的文本(对于本节而言，是"动机"这个词)就被标记为章节的标题，而不是以大号、粗体字打印的文本。从排版角度来看，这种功能化标记允许文档在不同情况下有不同的格式。功能化标记还有助于一个大文档的不同部分或是一个大网站的不同页面以统一的方式格式化。更重要的是，功能化标记帮助记录文档每个部分在语义上分别代表什么内容，并相应地帮助自动提取文档的关键部分。

对于包括HTML、SGML以及XML在内的标记语言家族，标记采用的形式是封闭在尖括号(< >)内的标签(tag)。标签都是成对使用的，以<tag>和</tag>来界定该标签所指的那部分文档的开始和结束。例如，文档的标题可以如下标记：

```
<title>Database System Concepts</title>
```

和HTML不同，XML没有指定的标签集，每个应用可以选择自己需要的标签集。这项特性是XML主要用于数据表示和交换而HTML主要用于文档格式化的关键所在。

例如，在我们采用的大学应用中，系、课程以及教师信息可以表示为XML文档的一部分，如图23-1和图23-2所示。注意如department、course、instructor和teaches这样的标签的使用。为了使例子简单，我们使用忽略了课程段信息的大学模式简化版。我们还使用ID标签来表示教师的标识符。稍后我们将分析其原因。

这些标签为每个值提供了上下文环境，并且允许识别值的语义。对于这个例子，XML数据表示并不提供任何比传统关系数据表示更明显的优势；但是由于它简洁，所以我们使用这个例子作为讲解实例。

图23-3显示了关于购物订单的信息如何在XML中进行表示，它是对XML的一个更加现实的使用。典型地，购物订单由一个机构生成并发送给另一个机构。在传统方式下，这些订单由购买者打印在纸上并发送给供应商；供应商手工地将数据重新输入计算机系统。这个缓慢的过程可以通过在购买者和供应

商之间电子化传送信息而大幅提高速度。类似的表示方式允许购物订单里的所有信息被自然地表示在单个文档中。（真实的购物订单比这个简化例子中描述的订单具有更为丰富的信息。）XML 提供了给数据加标签的标准方式；当然这两个机构必须在购物订单中出现什么标签以及这些标签的含义上达成一致。

```
<university>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </budget>
    </department>
    <course>
        <course_id> CS-101 </course_id>
        <title> Intro. to Computer Science </title>
        <dept_name> Comp. Sci </dept_name>
        <credits> 4 </credits>
    </course>
    <course>
        <course_id> BIO-301 </course_id>
        <title> Genetics </title>
        <dept_name> Biology </dept_name>
        <credits> 4 </credits>
    </course>
```

后部见图23-2

图 23-1 （部分）大学信息的 XML 表示

```
    <instructor>
        <IID> 10101 </IID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 65000 </salary>
    </instructor>
    <instructor>
        <IID> 83821 </IID>
        <name> Brandt </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 92000 </salary>
    </instructor>
    <instructor>
        <IID> 76766 </IID>
        <name> Crick </name>
        <dept_name> Biology </dept_name>
        <salary> 72000 </salary>
    </instructor>
    <teaches>
        <IID> 10101 </IID>
        <course_id> CS-101 </course_id>
    </teaches>
    <teaches>
        <IID> 83821 </IID>
        <course_id> CS-101 </course_id>
    </teaches>
    <teaches>
        <IID> 76766 </IID>
        <course_id> BIO-301 </course_id>
    </teaches>
</university>
```

图 23-2 续图 23-1

```
<purchase-order>
    <identifier> P-101 </identifier>
    <purchaser>
        <name> Cray Z. Coyote </name>
        <address> Mesa Flats, Route 66, Arizona 12345, USA </address>
    </purchaser>
    <supplier>
        <name> Acme Supplies </name>
        <address> 1 Broadway, New York, NY, USA </address>
    </supplier>
    <itemlist>
        <item>
            <identifier> RS1 </identifier>
            <description> Atom powered rocket sled </description>
            <quantity> 2 </quantity>
            <price> 199.95 </price>
        </item>
        <item>
            <identifier> SG2 </identifier>
            <description> Superb glue </description>
            <quantity> 1 </quantity>
            <unit-of-measure> liter </unit-of-measure>
            <price> 29.95 </price>
        </item>
    </itemlist>
    <total_cost> 429.85 </total_cost>
    <payment_terms> Cash-on-delivery </payment_terms>
    <shipping_mode> 1-second-delivery </shipping_mode>
</purchase-order>
```

图 23-3 一份购物订单的 XML 表示

与关系数据库中的数据存储相比，XML 表示方式可能效率不高，因为标签名称在整个文档中被反复使用。尽管有这些不利之处，但 XML 表示方式在用于机构间的数据交换，以及在文件中存储复杂结构信息时有相当大的优势：

- 首先，标签的存在使得消息是**自描述的**(self-documenting)；也就是说，不需要参考模式就可以理解文本的含义。比如我们可以很容易地阅读上述段落。
- 其次，文档的格式不严格。例如，如果某些发送者增加了一些附加信息，如记录访问某账户最后日期的标签 last_accessed，XML 数据接收者可以简单地忽略这个标签。图 23-3 给出了另一个例子，其中标识符为 SG2 的物品有一个被称为 unit-of-measure 的指定标签，而第一件物品并没有这个标签。当物品根据重量或体积排序时需要使用这个标签，而当物品只是简单地根据数量排序时，则可以省略这个标签。

  这种识别和忽略未预料到的标签的能力使得数据格式可以随时间不断演化，而不需要舍弃现有的应用程序。类似地，同样的标签可以多次出现的能力使得可以容易地表示多值属性。
- 再次，XML 允许嵌套结构，图 23-3 中所示的购物订单显示了拥有嵌套结构的好处。每份购物订单都有一个购买者和一个物品清单作为它的两个嵌套结构。而每件物品中都嵌套地包含物品标识符、说明和价格，而购买者嵌套地包含姓名和地址。

  这样的信息在关系模式中将被拆分成多个关系。物品信息将存储在一个关系中。购买者信息存储在第二个关系中，购物订单存储在第三个关系中，而购物订单、购买者和物品之间的联系将存储在第四个关系中。

  关系化表示有助于避免冗余。例如，在规范化关系模式中，对于每个物品标识符的物品说明只存储一次。但在 XML 购物订单中，如果多个购物订单订购了相同的物品，物品说明可能会重复出现。尽管如此，当需要与外部团体进行信息交换时，即使要付出冗余的代价，能够将与一份购物订单相关的所有信息集中到一个单一的嵌套结构中，仍然是有吸引力的。
- 最后，由于 XML 格式被广泛接受，所以有各种各样的工具可用来帮助对它的处理，包括创建和读取 XML 数据的程序设计语言 API、浏览器软件和数据库工具。

我们在后面的 23.7 节介绍 XML 数据的几个应用。正如 SQL 是查询关系数据的主导语言，XML 已经成为数据交换的主导格式。

982
|
985

23.2 XML 数据结构

XML 文档中基本的结构是**元素**(element)。一个元素就是一对互相匹配的开始和结束标签，以及它们之间出现的所有文本。

XML 文档必须有一个独立的**根**(root)元素来包含文档中的所有其他元素。在图 23-1 的例子中，<university>元素构成了根元素。此外，XML 文档中的元素必须正确地**嵌套**(nest)，比如：

```
<course> ··· <title> ··· </title> ··· </course>
```

是正确的嵌套，而

```
<course> ··· <title> ··· </course> ··· </title>
```

就不是正确的嵌套。

尽管正确的嵌套是一种直觉特性，我们仍可以更形式化地定义它。如果文本在某元素的开始标签和结束标签之间出现，那么称该文本出现在该元素的**上下文中**(in the context of)。如果每个开始标签都在同一个父元素的上下文中有唯一的结束标签与之匹配，那么该标签就是正确嵌套的。

注意文本可能会和一个元素的子元素混合在一起，如图 23-4 所示。和一些 XML 的其他特性一样，这种自由在文档处理环境中比在数据处理环境中更有意义，而且对表示结构化程度更高的数据如 XML 中的数据库内容并不是特别有用。

这种在其他元素内嵌套元素的能力为信息表示提供了一种可选的途径。图 23-5 展示了图 23-1 中部分大学信息的表示，但 course 元素嵌套在 department 元素中。这种嵌套表示能容易地找出一个系所开设的所有课程。类似地，教师所讲授课程的标识符被嵌套在 instructor 元素中。如果某位教师讲授多门

课程，那么对应的 instructor 元素中将会有多个 course_id 元素。限于篇幅，图 23-5 中省略了教师 Brandt 和 Crick 的详细信息，但它们与教师 Srinivasan 的结构相似。

```
...
    <course>
        This course is being offered for the first time in 2009
        <course_id> BIO-399 </course_id>
        <title> Computational Biology </title>
        <dept_name> Biology </dept_name>
        <credits> 3 </credits>
    </course>
...
```

图 23-4 文本与子元素的混合

```
<university-1>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
        <course>
            <course_id> CS-101 </course_id>
            <title> Intro. to Computer Science </title>
            <credits> 4 </credits>
        </course>
        <course>
            <course_id> CS-347 </course_id>
            <title> Database System Concepts </title>
            <credits> 3 </credits>
        </course>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </budget>
        <course>
            <course_id> BIO-301 </course_id>
            <title> Genetics </title>
            <credits> 4 </credits>
        </course>
    </department>
    <instructor>
        <ID> 10101 </ID>
        <name> Srinivasan </name>
        <dept name> Comp. Sci. </dept name>
        <salary> 65000. </salary>
        <courseid> CS-101 </courseid>
    </instructor>
</university-1>
```

图 23-5 大学信息的嵌套 XML 表示

尽管 XML 中的嵌套表示是自然的，但是，这会导致数据存储的冗余。例如，如图 23-6 所示，假设某位教师讲授的课程的细节被嵌套存储于 instructor 元素中，如果某门课程被多位教师讲授，课程信息（如题目、院系及学分）会冗余地存储在每位与该课程相关联的教师中。

986
987

为了避免连接，嵌套表示在 XML 数据交换应用中广为使用。例如，一次订购会在多份购物订单上冗余地存储发送者和接收者的完整地址，而规范化表示可能需要将购物订单记录与关系 *company_address* 进行连接以获得地址信息。

除了元素以外，XML 还定义了**属性**（attribute）的概念。例如，课程的标识符可以表示为课程的一个属性，如图 23-7 所示。一个元素的属性作为在标签的结束符 ">" 之前的 *name = value* 对出现。属性是字符串，不包含标记。此外，属性在给定标签中只可以出现一次，不像子元素那样可以重复。

注意在文档构造环境中，子元素和属性之间的区别很重要——属性是隐式的、不出现在打印或显示文档中的文本。但是在数据库和 XML 的数据交换应用中，这种区别不那么重要。选择将数据表示为属性还是子元素常常是随意的。总之，仅将属性用于表示标识符，而将所有其他数据存放于子元素中

是明智的。

```
<university-2>
    <instructor>
        <ID> 10101 </ID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 65000 </salary>
        <teaches>
            <course>
                <course_id> CS-101 </course_id>
                <title> Intro. to Computer Science </title>
                <dept_name> Comp. Sci. </dept_name>
                <credits> 4 </credits>
            </course>
        </teaches>
    </instructor>

    <instructor>
        <ID> 83821 </ID>
        <name> Brandt </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 92000 </salary>
        <teaches>
            <course>
                <course_id> CS-101 </course_id>
                <title> Intro. to Computer Science </title>
                <dept_name> Comp. Sci. </dept_name>
                <credits> 4 </credits>
            </course>
        </teaches>
    </instructor>
</university-2>
```

图 23-6 嵌套 XML 表示中的冗余

```
...
    <course course_id= "CS-101">
        <title> Intro. to Computer Science </title>
        <dept_name> Comp. Sci. </dept_name>
        <credits> 4 </credits>
    </course>
...
```

图 23-7 属性的使用

最后一个有关句法的注意事项是，有一种形为 < element > </element > 的元素，它不包含任何子元素或文本，可以缩写为 < element/ > ；但是缩写的元素可以包含属性。

由于设计 XML 文档是为了应用程序之间的交换，为了允许组织机构指定全球唯一的名字作为文档中的元素标签使用，所以就引入了**名字空间**(namespace)机制。名字空间的思想就是在每个标签或属性的前面加上通用资源标识符(比如网址)。举例来说，如果耶鲁大学想确保创建的 XML 文档中的标签不会与任何商业伙伴的 XML 文档中的标签重复，可以在每个标签名称前面加上一个唯一标识符，并用冒号相隔。该大学可能使用如下 Web URL：

http://www.yale.edu

作为唯一标识符。在每个标签中都使用长唯一标识符很不方便，所以名字空间标准提供了一种定义标识符缩写的方法。

在图 23-8 中，根元素(university)有个属性 xmlns:yale，它声明 yale 被定义为上述给定 URL 的缩写。这个缩写随后可用于各种元素标签中，如图 23-8 中所示。

一份文档可以有不止一个名字空间，它们声明为根元素的一部分。不同元素可以与不同名字空间相关联。可以通过在根元素中使用属性 xmlns 代替 xmlns:yale 来定义默认名字空间(default namespace)。没有显式名字空间前缀的元素就属于默认名字空间。

```
<university xmlns:yale="http://www.yale.edu">
    ...
    <yale:course>
        <yale:course_id> CS-101 </yale:course_id>
        <yale:title> Intro. to Computer Science </yale:title>
        <yale:dept_name> Comp. Sci. </yale:dept_name>
        <yale:credits> 4 </yale:credits>
    </yale:course>
    ...
</university>
```

图 23-8 通过使用名字空间来指定唯一标签名

有时我们需要存储包含标签的值而又不想将其中的标签解释为 XML 的标签。为了能这样做，XML 允许这样的结构：

```
<![CDATA  < course > ··· </course > ]]>
```

因为文本 < course > 包含在 CDATA 中，所以它被作为正常的文本数据，而不是当作标签。术语 CDATA 代表字符数据。

23.3 XML 文档模式

数据库有模式，用来限制什么信息可以存储在数据库中并限制存储信息的数据类型。与此相反，在默认情况下 XML 文档可以不需任何相关模式而被创建：这样，一个元素可以有任意的子元素或属性。虽然在 XML 文档给定了数据格式的自描述特性的情况下，这样的自由有时是可以接受的，但是当 XML 文档必须作为应用程序的一部分进行自动处理时，甚至当大量的相关数据需要在 XML 中进行格式化时，这种自由通常是没有用处的。

在此，我们介绍作为 XML 标准组成部分的第一种模式定义语言——文档类型定义，以及新近被定义的其替代者——XML 模式（XML schema）。另外一个也被使用的 XML 模式定义语言称作 Relax NG，但是我们在这里并不介绍它；关于 Relax NG 的更多信息，请参阅文献注解部分的参考文献。

23.3.1 文档类型定义

文档类型定义（Document Type Definition，DTD）是 XML 文档的一个可选部分。DTD 的主要目的与模式很像：对文档中出现的信息进行约束和类型限定。但是事实上 DTD 并不限制基本类型（如整数和字符串）意义上的类型，它只限制元素中子元素和属性的出现。DTD 主要是有关一个元素中可以出现何种模式的子元素的一系列规则。图 23-9 展示了一个大学信息文档的 DTD 实例的一部分，图 23-1 中的 XML 文档就遵循该 DTD。

```
<!DOCTYPE university [
    <!ELEMENT university ( (department|course|instructor|teaches)* )>
    <!ELEMENT department ( dept_name, building, budget)>
    <!ELEMENT course ( course_id, title, dept_name, credits)>
    <!ELEMENT instructor (IID, name, dept_name, salary)>
    <!ELEMENT teaches (IID, course_id)>
    <!ELEMENT dept_name( #PCDATA )>
    <!ELEMENT building( #PCDATA )>
    <!ELEMENT budget( #PCDATA )>
    <!ELEMENT course_id ( #PCDATA )>
    <!ELEMENT title ( #PCDATA )>
    <!ELEMENT credits( #PCDATA )>
    <!ELEMENT IID( #PCDATA )>
    <!ELEMENT name( #PCDATA )>
    <!ELEMENT salary( #PCDATA )>
]>
```

图 23-9 一个 DTD 示例

每个声明都以一个元素的子元素的正则表达式形式出现。这样，在图 23-9 的 DTD 中，一个 university 元素包含一个或多个 course、department 或 instructor 元素；"|"类操作符意指"或"，而"+"操作符意指"一个或多个"。还有两个操作符在这里没有出现："*"操作符用来表示"零个或多个"，而

"?"操作符用来指定一个可选的元素(即"零个或一个")。

course 元素包含了元素 course_id、title、dept_name 和 credits(按此顺序)。类似地，在 DTD 中，department 和 instructor 将它们的关系模式属性定义为子元素。

最后，元素 course_id、title、dept_name、credits、building、budget、IID、name 和 salary 都被声明为 #PCDATA 类型。关键字#PCDATA 表示文本数据；它的名字在历史上来源于"被解析的字符数据"(parsed character data)。另外两种专门的类型声明是 empty(是说这个元素没有内容)和 any(是说对这个元素的子元素没有限制；也就是说，任何元素甚至没有在 DTD 中提到的元素都可以作为该元素的子元素出现)。一个元素没有声明等同于将其显式地声明为 any 类型。

每个元素所允许的属性也在 DTD 中声明。与子元素不同，属性没有顺序之分。属性可以指定为 CDATA、ID、IDREF 或 IDREFS 类型。CDATA 类型只是说这个属性包含字符数据。而另外三个就没有这么简单，一会儿将对它们进行详细解释。例如，下面是来自 DTD 中的一行，它指定 course 元素有个类型为 course_id 的属性，并且该属性必须有值。

```
<! ATTILIST course course_id CDATA #REQUIRED >
```

属性必须有一个类型声明和一个默认声明。默认声明可以包含该属性的一个默认值，也可以包含#REQUIRED(意思是每个元素在该属性上必须指定一个值)或#IMPLIED(意思是指没有提供默认值，文档可以忽略这个属性)。如果一个属性有默认值，那么对每个没有为该属性指定值的元素，这个默认值在读取 XML 文档时被自动填进去。

988
991

类型为 ID 的属性提供该元素的唯一标识符。出现在一个元素的 ID 属性中的值一定不能在同一文档中的任何其他元素中出现。一个元素最多只有一个属性允许为 ID 类型。(在 XML 表示中，为了避免与 ID 类型的冲突，我们将 *instructor* 关系的 *ID* 属性更名为 IID。)

类型为 IDREF 的属性是对一个元素的引用。这个属性必须包含出现在文档中某个元素的 ID 属性上的值。IDREFS 类型允许以空格分开的一个引用列表。

图 23-10 展示了一个 DTD 示例，其中课程、系和教师的标识用 ID 属性来表示，而它们之间的联系用 IDREF 和 IDREFS 属性来表示。course 元素使用 course_id 作为它们的标识符属性。为了做到这一点，course_id 被设置为 course 的一个属性而不是子元素。此外，每个 course 元素还包含一个与该 course 对应的 department 的 IDREF 属性，以及一个 IDREFS 属性 instructors，它表示讲授该课程的那些教师。department 元素具有一个称作 dept_name 的标识符属性，instructor 元素具有一个称作 IID 的标识符属性，以及一个 IDREF 属性 dept_name，表示该教师所在的系。

```
<!DOCTYPE university-3 [
    <!ELEMENT university ( (department|course|instructor)+ )>
    <!ELEMENT department ( building, budget )>
    <!ATTLIST department
        dept_name ID #REQUIRED >
    <!ELEMENT course (title, credits )>
    <!ATTLIST course
        course_id ID #REQUIRED
        dept_name IDREF #REQUIRED
        instructors IDREFS #IMPLIED >
    <!ELEMENT instructor ( name, salary )>
    <!ATTLIST instructor
        IID ID #REQUIRED
        dept_name IDREF #REQUIRED >
    <!-- declarations for title, credits, building,
        budget, name and salary -->
]>
```

图 23-10 具有 ID 和 IDREFS 属性类型的 DTD

图 23-11 展示了一个基于图 23-10 中 DTD 的 XML 文档示例。

ID 和 IDREF 属性在面向对象和对象-关系数据库中扮演着同样的引用机制的角色，允许构造复杂数据关系。

```
<university-3>
    <department dept_name="Comp. Sci.">
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department dept_name="Biology">
        <building> Watson </building>
        <budget> 90000 </budget>
    </department>
    <course course_id="CS-101" dept_name="Comp. Sci"
                                    instructors="10101 83821">
        <title> Intro. to Computer Science </title>
        <credits> 4 </credits>
    </course>
    <course course_id="BIO-301" dept_name="Biology"
                                    instructors="76766">
        <title> Genetics </title>
        <credits> 4 </credits>
    </course>
    <instructor IID="10101" dept_name="Comp. Sci.">
        <name> Srinivasan </name>
        <salary> 65000 </salary>
    </instructor>
    <instructor IID="83821" dept_name="Comp. Sci.">
        <name> Brandt </name>
        <salary> 72000 </salary>
    </instructor>
    <instructor IID="76766" dept_name="Biology">
        <name> Crick </name>
        <salary> 72000 </salary>
    </instructor>
</university-3>
```

图 23-11 具有 ID 和 IDREF 属性的 XML 数据

文档类型定义与 XML 的文档格式继承有很强的联系。由于这个原因，将文档类型定义作为数据处理应用中的 XML 类型结构在很多方面是不合适的。尽管如此，大量的数据交换格式是以 DTD 来定义的，因为它们是最初标准的一部分。下面是一些以 DTD 作为模式机制的局限性：

992
↓
993

- 单个文本元素和属性不能更进一步限定类型。比如，balance 元素不能被限制为一个正数。缺乏这样的约束在数据处理和交换应用中是有问题的。这就必须包含一些代码来验证元素和属性的类型。
- 很难用 DTD 机制来指定子元素的无序集合。顺序在数据交换中很少有所需用处（不像文档排版那样严格）。虽然图 23-9 中或操作符（|操作符）和 * 或 + 类操作符的组合允许指定标签的无序集合，可是很难指定每个标签只可以出现一次。
- ID 和 IDREF 中缺乏类型限定。这样就没有办法来指定 IDREF 或 IDREFS 属性应该引用的元素类型。于是，图 23-10 中的 DTD 不能阻止一个 course 元素的"dept_name"属性引用其他 course 的现象，尽管这样做毫无意义。

23.3.2 XML Schema

为了弥补 DTD 机制的缺陷，发展出了一种更完善的模式语言——**XML Schema**。我们给出 XML Schema 的概览，然后列出它对 DTD 改进的一些方面。

XML Schema 定义了很多内置类型，例如 string、integer、decimal、date 和 boolean。此外，它允许用户自定义类型，这些用户自定义类型可能是增加了限制的简单类型，或是使用诸如 complexType 和 sequence 那样的构造符构造出的复杂类型。

图 23-12 和图 23-13 显示了图 23-9 中的 DTD 是如何用 XML Schema 来表示的。下面我们介绍图中示例的 XML Schema 特性。

```
<xs: schema xmlns: xs = "http: //www. w3. org/2001/XMLSchema" >
<xs: element name = "university" type = "universityType" / >
<xs: element name = "department" >
      <xs: complexType >
            <xs: sequence >
                  <xs: element name = "dept_ name" type = "xs: string"/ >
                  <xs: element name = "building" type = "xs: string"/ >
                  <xs: element name = "budget" type = "xs: decimal"/ >
            </xs: sequence >
      </xs: complexType >
</xs: element >
<xs: element name = "course" >
      <xs: complexType >
            <xs: sequence >
                  <xs: element name = "course_ id" type = "xs: string"/ >
                  <xs: element name = "title" type = "xs: string"/ >
                  <xs: element name = "dept_ name" type = "xs: string"/ >
                  <xs: element name = "credits" type = "xs: decimal"/ >
            </xs: sequence >
      </xs: complexType >
</xs: element >
<xs: element name = "instructor" >
      <xs: complexType >
            <xs: sequence >
                  <xs: element name = "IID" type = "xs: string"/ >
                  <xs: element name = "name" type = "xs: string"/ >
                  <xs: element name = "dept_ name" type = "xs: string"/ >
                  <xs: element name = "salary" type = "xs: decimal"/ >
            </xs: sequence >
      </xs: complexType >
</xs: element >
```

后面见图 23-13

图 23-12 图 23-9 中 DTD 的 XML Schema 版本

```
<xs:element name="teaches">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="IID" type="xs:string"/>
            <xs:element name="course_id" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:complexType name="UniversityType">
    <xs:sequence>
        <xs:element ref="department" minOccurs="0"
                maxOccurs="unbounded"/>
        <xs:element ref="course" minOccurs="0"
                maxOccurs="unbounded"/>
        <xs:element ref="instructor" minOccurs="0"
                maxOccurs="unbounded"/>
        <xs:element ref="teaches" minOccurs="0"
                maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
</xs:schema>
```

图 23-13 续图 23-12

第一个要注意的问题是 XML Schema 中的模式定义本身是利用 XML Schema 定义的各种标签用 XML 语法指定的。为了避免与用户定义标签发生冲突，我们在 XML Schema 标签上增加名字空间的前缀 "xs:"，这个前缀通过根元素的 xmlns: xs 声明与 XML Schema 名字空间相关联：

```
<xs: schema xmlns: xs = "http: //www. w3. org/2001/XMLSchema" >
```

注意任何名字空间前缀都可用来替换 xs；因此我们可以用"xsd:"替换模式定义中的所有"xs:"，而不需改变模式定义的含义。XML Schema 定义的所有类型必须加上这个名字空间前缀。

第一个元素是根元素 university，它的类型被指定为后面声明的 UniversityType，接着这个例子定义

了元素 department、course、instructor 和 teaches 的类型。注意每个元素都用带 xs: element 标签的元素指定，其内容包含类型的定义。

department 类型被定义为复杂类型，并进而被指定为一个由元素 dept_name、building 和 budget 构成的序列。任何具有属性或嵌套子元素的类型必须被指定为复杂类型。

或者，元素的类型可以通过属性 type 被指定为一个预定义类型，观察 XML Schema 类型 xs: string 和 xs: decimal 是如何被用于限制数据元素(如 dept_name 和 credits)的类型的。

最后，这个例子定义了 UniversityType 类型，它包括每个 department、course、instructor 和 teaches 的零个或多个出现。注意用 ref 指明前面定义元素的出现。XML Schema 可以用 minOccurs 和 maxOccurs 来定义子元素出现的最少次数和最多次数。最少次数和最多次数的默认值都为 1，因此必须显式指定这两个值以允许出现零个或多个 department、course、instructor 和 teaches 元素。

属性用 xs: attribute 标签来指定。例如，我们可以通过在 department 元素的声明中增加：

```
<xs: attribute name = "dept_name"/>
```

来把 dept_name 定义为属性。在上述属性声明中增加属性 use = "required" 表示该属性必须被指定，而 use 的默认值为 optional。属性声明应直接出现在包含它的 complexType 声明之下，即使元素是嵌套在序列声明中的。

我们可以使用 xs: complexType 元素来创建命名的复杂类型。除了要为 xs: complexType 元素增加 name = typeName (其中 typeName 是我们要赋给类型的名字)属性外，语法与图 23-12 中用于 xs: complexType 元素的语法相同。然后我们就可以通过使用 type 属性用这个已命名的类型来指定元素的类型，就如我们在例子中使用 xs: decimal 和 xs: string 一样。

994
|
996

除了定义类型，关系模式还允许指定约束。XML Schema 允许指定键和键引用，对应于 SQL 中的主码和外码定义。在 SQL 中，主码约束或唯一性约束保证关系中的属性值不会重复出现。在 XML 的上下文环境中，我们需要指定一个范围，在这个范围内值是唯一的并形成键。selector 是定义约束范围的路径表达式，field 声明指定形成键的元素或属性。[⊖]要指定 dept_name 构成根元素 university 下的 department 元素的键，我们在模式定义中增加下面的约束声明：

```
<xs: key name = "deptKey">
    <xs: selector xpath = "/university/department"/>
    <xs: field xpath = "dept_name"/>
</xs: key>
```

相应地，从 course 到 department 的外码约束可以定义如下：

```
<xs: name = "courseDeptFKey" refer = "deptKey">
    <xs: selector xpath = "/university/course"/>
    <xs: field xpath = "dept_name"/>
</xs: keyref>
```

注意 refer 属性指定被引用的键声明的名字，而 field 声明指明引用属性。

XML Schema 有很多优于 DTD 之处，目前正被广泛应用。我们从上述例子中所看到的优点有：

- 它允许把元素中出现的文本限制为专门的类型(如特定格式的数字类型)或复杂类型(如其他类型元素的序列)。
- 它允许创建用户定义类型。
- 它允许唯一性和外键约束。
- 它与名字空间结合以允许文档的不同部分遵循不同的模式。

除我们已经看到的特性外，XML Schema 还支持若干 DTD 不支持的其他特性，如：

- 它允许为创建专门的类型而对类型进行限制，如指定最小和最大值。
- 它允许以使用继承的形式来扩展复杂类型。

⊖ 这里我们使用熟悉的语法来书写简单路径表达式，XML 拥有丰富的路径表达式语法，称作 XPath，我们将在 23.4.2 节中加以介绍。

我们对 XML Schema 只进行了概括性的介绍，要了解更多有关 XML Schema 的信息，请参考文献注解中的参考文献。 [997]

23.4 查询和转换

考虑到越来越多的应用使用 XML 来交换数据，作为数据中介并存储数据，有效管理 XML 数据的工具就变得越来越重要了。XML 数据的查询和转换工具对于从体积庞大的 XML 数据中抽取信息以及在 XML 的不同表示(模式)之间转换数据来说无为重要。就像关系查询的输出是关系一样，XML 查询的输出可以是 XML 文档。因此查询和转换可以结合在一个工具中。

在本节我们介绍 XPath 和 XQuery 语言：

- XPath 是一种用于路径表达式的语言，事实上也是 XQuery 的基石。
- XQuery 是查询 XML 数据的标准语言。它的模型是仿照 SQL 的，但由于它要处理嵌套 XML 数据，因此与 SQL 有明显的差异。XQuery 也结合了 XPath 表达式。

XSLT 语言是为 XML 转换而设计的另一种语言。不过，它主要用于文档格式化而非数据管理应用中。因此本书将不对此进行讨论。

本章末尾的"工具"部分提供了对几种软件的引用，这些软件可用于执行由 XPath 和 XQuery 书写的查询。

23.4.1 XML 树模型

这些语言都使用了 XML 数据的**树模型**(tree model)。一份 XML 文档被建模为一棵树(tree)，其结点(node)对应于元素和属性。元素结点可以有子结点，子结点可以是该元素的子元素或属性。相应地，除根元素外的每个结点(不管是属性还是元素)都有一个元素是其父结点。XML 文档中元素和属性的顺序根据树的子结点的顺序建模。XML 数据的树模型中使用了父亲、孩子、祖先、后代以及兄弟等术语。

元素的文本内容可以建模为该元素的文本子结点。包含由于子元素的插入而导致间断的文本的元素可以有多个文本子结点。例如，包含"this is a <bold> wonderful </bold> book"的元素将有一个子元素孩子对应于 bold 元素以及两个文本子结点对应于"this is a"和"book"。由于这种结构在数据表示中并不常用，我们将假设元素不会既包含文本又包含子元素。 [998]

23.4.2 XPath

XPath 通过路径表达式指向 XML 文档的部分内容。这种语言可以看作是面向对象和对象-关系数据库(见 22.6 节)中简单路径表达式的扩展。XPath 标准的当前版本是 XPath 2.0，我们的介绍就基于这个版本的。

XPath 中的**路径表达式**(path expression)是由"/"(替代了 SQL 中用于分隔定位步骤的"."操作符)隔开的定位步骤的一个序列。路径表达式的结果是一个结点的集合。例如，在图 23-11 的文档中，XPath 表达式

```
/university-3/instructor/name
```

会返回这些元素：

```
<name>Srinivasan</name>
<name>Brandt</name>
```

表达式

```
/university-3/instructor/name/text()
```

会返回相同的名字，但是没有外围标签。

路径表达式是从左到右执行计算的。像目录层次结构一样，初始的'/'表示文档的根。注意，这是一个在作为文档标签的 <university-3> "之上"的抽象的根。

当计算路径表达式时，任意时刻路径的结果由文档中某些结点的一个有序集合构成。初始时"当前"元素集合只包含一个结点，即抽象的根。当路径表达式的下一步骤是一个元素名，如 instructor 时，此步骤的结果由当前元素集中元素的子元素中具有该指定名字的元素所对应的结点构成。这些结点随之成为路径表达式计算的下个步骤所使用的当前元素集。因此，表达式：

/university - 3

返回对应于标签：

< university - 3 >

的单个结点。而：

/university - 3/instructor

[999] 返回对应于

instructor

的两个结点，它们是结点：

university - 3

的子结点。

路径表达式的结果是经过最后一步路径表达式计算后所得到的结点集合。每个步骤返回结点的顺序与它们在文档中出现的顺序一致。

由于多个孩子可以有同样的名字，所以结点集中结点的数量可能在每个操作步骤中增加或减少。还可以使用"@"符号访问属性值，例如，/university - 3/course/@course_id 返回 course 元素的 course_id 属性的所有值的集合。默认情况下，并不考虑 IDREF 链接；我们将在稍后看到如何处理 IDREF。

XPath 支持许多其他特性：

- 可以在路径的每个操作步骤中使用选择谓词，选择谓词包含在方括号中，例如，

  /university - 3/course[credits >=4]

  返回学分大于或等于 4 的 course 元素，而：

  /university - 3/course[credits >=4]/@course_id

  返回这些课程的课程标识。

  我们可以通过列出一个子元素而不使用任何比较运算的方式来测试该子元素是否存在。例如，如果我们只是将上面的">=4"去掉，这个表达式就会返回具有学分子元素的所有课程的标识，而不考虑学分的值。

- XPath 提供了一些可以作为谓词的一部分使用的函数，包括测试当前结点在兄弟顺序当中的位置以及聚集函数 count()，它计算与函数所作用的表达式相匹配的结点数量。例如，在图 23-6 中的 XML 表示中，路径表达式：

  /university - 3/instructor[count(./teaches/course) > 2]

  返回讲授两门以上课程的教师。布尔连接 and 和 or 可以在谓词中使用，而函数 not(…)可用于否定。

- 函数 id("foo")返回属性的类型为 ID 且值为"foo"的结点(如果存在的话)。函数 id 甚至可以应
[1000] 用于引用的集合，或者包含用空格隔开的多个引用的字符串，如 IDREFS。例如，路径：

  /university - 3/course/id(@dept_name)

  返回被 course 元素的 dept_name 属性引用的所有系元素。而：

  /university - 3/course/id(@instructors)

  返回被 course 元素的 instructors 属性所引用的教师元素。

- 操作符 | 允许对表达式结果进行合并。例如，给定图 23-11 中所示的模式下的数据，我们可以使用表达式：

  /university - 3/course[@dept_name = "Comp. Sci"] |
  /university - 3/course[@dept_name = "Biology"]

  来查找 Computer Science 和 Biology 所开设课程的并集。但是 | 操作符不能嵌套在其他操作符中。还有一点值得注意，"并"中的结点依据它们在文档中出现的顺序被返回。

- XPath 表达式可以使用"//"来跳过多层结点。举例来说，表达式/university - 3//name 找出/

university-3 元素下的任意位置上的所有 name 元素，而不管它们被包含在哪个元素中，也不管在 university-3 和 name 元素之间存在多少个层次的元素。这个例子演示了在没有关于模式全部知识的情况下查找所需数据的能力。

- 路径中的操作步骤不需要只是在当前结点集的子结点中进行选择。事实上，这只是路径中一个步骤可能执行的若干方向（如父亲、兄弟、祖先和后代）中的一个。我们在这里省略详细描述，但是请注意，上面描述的"//"是用于指定"所有后代"的一种缩写形式，而".."指定父亲。
- 内置函数 doc(name)返回一个已命名文档的根；这个名字可以是文件名或是 URL。该函数返回的根可用于路径表达式中以访问文档的内容。因此，路径表达式可以应用在特定文档上，而不是当前默认文档上。

  例如，如果我们的大学例子中的大学数据包含在文件"university.xml"中，下述路径表达式将返回大学中的所有系：

```
doc("university.xml")/university/department
```

[1001]

函数 collection(name)与 doc 相似，但返回以 name 标识的文档集合。例如，collection 函数可用于打开一个被视为文档集合的 XML 数据库。XPath 表达式中随后的元素可以从集合中选择出适当的文档。

在大多数例子中，我们假设表达式是在数据库上下文中计算的，该数据库隐式地提供了一个"文档"的集合，XPath 表达式就在这个集合上进行计算。在这种情况下，我们不需要使用 doc 函数或 collection 函数。

23.4.3 XQuery

万维网联盟（World Wide Web Consortium，W3C）开发了 XQuery，作为 XML 标准的查询语言。我们的讨论基于 2007 年 1 月 23 日发布的 W3C 推荐版本 XQuery 1.0。

23.4.3.1 FLWOR 表达式

XQuery 查询模仿 SQL 查询，但是与 SQL 有明显的差异。它包含五个部分：**for**、**let**、**where**、**order by** 和 **return**。它们被称为"FLWOR"（读作"flower"）表达式，FLWOR 中的字母代表这五个部分。

如下所示是一个简单的 FLWOR 表达式，它基于图 23-11 中使用了 ID 和 IDREFS 的 XML 文档，返回学分大于 3 的课程标识。

```
for $ x in /university-3/course
let $ courseId := $ x/@ course_id
where $ x/credits > 3
return <course_id> { $ courseId } </course_id>
```

for 子句就像 SQL 中的 **from** 子句，指定在 XPath 表达式结果范围内变动的变量。当指定的变量多于一个时，结果包含这些变量所有可能值的笛卡儿积，就像 SQL 的 **from** 子句所做的一样。

let 子句允许将 XPath 表达式结果赋值给变量名以简化表达。**where** 子句同 SQL 的 **where** 子句一样，对来自 **for** 子句的连接元组执行附加测试。**order by** 子句与 SQL 的 **order by** 子句一样，允许对输出排序。最后，**return** 子句允许构造 XML 形式的结果。

FLWOR 查询不需要包含所有子句；例如一个查询可能只包含 **for** 和 **return** 子句，而省略 **let**、**where** 和 **order by** 子句。前面的 XQuery 查询中就没有包含 **order by** 子句。事实上，由于这个查询很简单，我们可以容易地去掉 **let** 子句，并把 **return** 子句中的 $courseId 变量替换为 $x/@ course_id。总而言之，由于 **for** 子句使用了 XPath 表达式，选择可以发生在 XPath 表达式内部。这样，可以只用 **for** 和 **return** 子句构造一个等价的查询：

[1002]

```
for $x in /university-3/course[credits > 3]
return <course_id> { $x/@ course_id } </course_id>
```

然而，**let** 子句有助于简化复杂查询。还要注意如果右侧的路径表达式返回一个由多个元素或值构成的序列，那么由 **let** 子句赋值的变量可能包含由多个元素或值所构成的序列。

观察在 **return** 子句中对大括号（"{ }"）的使用。当 XQuery 找到一个表达式开头的元素时，如

< course_id >，它将这个元素的内容看作普通的 XML 文本，只有大括号内部的内容会被当作表达式计算。因此，如果我们省略掉上述 **return** 子句中的大括号，结果将包含字符串"$x/@course_id"的多个拷贝，其中每个都包含在一个 course_id 标签中。而大括号中的内容是被当作表达式计算的。注意即使当大括号出现在引号中时，这个规定也是成立的。因此，我们可以通过用下述 **return** 子句替换上述 **return** 子句来修改上述查询，使其返回一个带 course 标签并以课程标识作为属性的元素。

```
return < course course_id = "{ $x/@course_id }" />
```

XQuery 提供了使用 **element** 和 **attribute** 构造器来构造元素的另一种方法。例如，如果前面查询中的 **return** 子句用下面的 **return** 子句替换，查询将返回具有 course_id 和 dept_name 属性且包含 title 和 credits 子元素的元素。

```
return element course {
    attribute course_id { $x/@course_id },
    attribute dept_name { $x/dept_name },
    element title { $x/title },
    element credits { $x/credits }
}
```

注意同前面一样，需要使用大括号以使一个字符串被当作表达式计算。

23.4.3.2 连接

在 XQuery 中指定连接与在 SQL 中指定连接很相似。图 23-1 中 course、instructor 和 teaches 元素的
1003 连接可以这样用 XQuery 来写：

```
for $c in /university/course,
    $i in /university/instructor,
    $t in /university/teaches
where $c/course_id = $t/course_id
    and $t/IID = $i/IID
return <course_instructor> { $c $i } </course_instructor>
```

可以用 XPath 的选择操作表达同样的查询：

```
for $c in /university/course,
    $i in /university/instructor,
    $t in /university/teaches[ $c/course_id = $t/course_id
        and $t/IID = $i/IID ]
return <course_instructor> { $c $i } </course_instructor>
```

XQuery 中的路径表达式与 XPath 2.0 中的路径表达式相同。路径表达式可以返回单个值或元素，或返回一个值或元素的序列。如果没有模式信息，就可能无法确定一个路径表达式是返回单个值还是一个值的序列。这样的路径表达式可以出现在比较运算如 =、< 和 > = 中。

XQuery 对序列上的比较运算有一个有趣的定义。例如，如果 $x/credits 的结果是单个值，表达式 $x/credits > 3 可能具有通常的含义；但如果其结果是一个包含多个值的序列，那么表达式在至少有一个值大于 3 时为真。类似地，表达式 $x/credits = $y/credits 的值在第一个表达式返回值中的任意一个与第二个表达式返回值中的任意一个相等时为真。如果这种操作不合适，可以使用操作符 eq、ne、lt、gt、le、ge 来代替。这些操作符在它们的任意一个输入是一个多个值的序列时就会报错。

23.4.3.3 嵌套查询

XQuery FLWOR 表达式可以嵌套在 **return** 子句中，以生成没有出现在源文件中的元素嵌套。例如，图 23-5 中展示的是嵌套在院系元素内部的课程元素的 XML 结构，可以从图 23-1 中的结构用图 23-14 所示的查询来生成。

这个查询还引入了 $d/* 句法，它是指绑定到变量 $d 的结点（或结点序列）的所有孩子。类似地，$d/text() 给出一个元素的不带标签的文本内容。

XQuery 提供很多可以应用在元素或值的序列上的聚集函数，如 sum() 和 count()。应用在序列上的 distinct-values() 函数返回不含重复值的序列。由路径表达式返回的值的序列（集合体）中可能有些值
1004 是重复的，因为它们在文档中是重复的，尽管文档中的每个结点至多可以在 XPath 表达式结果中出现

次。XQuery 还支持很多其他函数，更多信息请参见文献注解中的参考文献。这些函数实际上在 XPath 2.0 和 XQuery 中是通用的，而且可以使用在任何 XPath 路径表达式中。

```
<university-1>
{
    for $d in /university/department
    return
        <department>
            { $d/* }
            { for $c in /university/course[dept_name = $d/dept_name]
              return $c }
        </department>
}
{
    for $i in /university/instructor
    return
        <instructor>
            { $i/* }
            { for $c in /university/teaches[ID = $i/ID]
              return $c/course_id }
        </instructor>
}
</university-1>
```

图 23-14 用 XQuery 创建嵌套结构

为了避免名字空间冲突，函数与如下名字空间相关：

http://www.w3.org/2005/xpath-functions

该名字空间有一个默认名字空间前缀 fn。因此，这些函数可以以 fn:sum 或 fn:count 的形式被无歧义地调用。

由于 XQuery 不提供 **group by** 构造，聚集查询可以用路径或 FLOWR 表达式（嵌套在 **return** 子句中的）上的聚集函数来书写。例如，以下在 university XML schema 上的查询找出每个系所有教师的总薪酬：

```
for $d in /university/department
return
    <department-total-salary>
        <dept_name> { $d/dept_name/text() } </dept_name>
        <total_salary> { fn:sum(
            for $i in /university/instructor[dept_name = $d/dept_name]
            return $i/salary
        ) } </total_salary>
    </department-total-salary>
```

23.4.3.4 结果的排序

[1005]

在 XQuery 中可以用 **order by** 子句对结果排序。例如，这个查询输出根据 name 子元素排序的所有教师元素：

```
for $i in /university/instructor
order by $i/name
return <instructor> { $i/* } </instructor>
```

可以使用 **order by** $i/name **descending** 使结果按降序排列。

可以在嵌套的多个层次上进行排序。例如，我们通过下面的查询，可以获得根据系名排序的大学信息的嵌套表示，其中每门课程按课程标识排序：

```
<university-1> {
    for $d in /university/department
    order by $d/dept_name
    return
        <department>
            { $d/* }
            { for $c in /university/course[dept_name = $d/dept_name]
```

```
                order by $c/course_id
                return <course> { $c/* } </course>
            </department>
    } </university-1>
```

23.4.3.5 函数和类型

XQuery 提供很多内置函数，如数字型函数和字符串匹配与操作函数。除此以外，XQuery 支持用户自定义函数。下面的用户自定义函数以教师标识为输入，返回该教师所属院系开设的所有课程的列表：

```
declare function local:dept_courses($id as xs:string) as element(course)* {
 for $i in /university/instructor[ID = $id],
  $c in /university/courses[dept_name = $i/dept_name]
 return $c
}
```

在上例中用到的名字空间前缀 xs: 是 XQuery 预定义的，表示与 XML Schema 名字空间相关联。而名字空间 local: 也是预定义的，表示与 XQuery 本地函数相关联。

函数参数和返回值的类型声明是可选的，可以省略。XQuery 使用 XML Schema 的类型系统。
1006 element 类型允许元素带有任何标签，而 element(course) 允许元素带有 course 标签。类型可以带有后缀 * 以表示该具有类型值的一个序列。例如，函数 dept_courses 的定义指定其返回值为 course 元素的一个序列。

下面的查询展示了函数调用，它打印名为 Srinivasan 的那些教师所在院系开设的课程：

```
for $i in /university/instructor[name = "Srinivasan"],
return local:dept_courses($i/ID)
```

当需要时，XQuery 进行自动的类型转换。例如，如果用一个由字符串表示的数值型的值与一个数值类型进行比较，将自动进行从字符串到数值类型的类型转换。当一个元素被传给以字符串值为参数的函数时，通过拼接包含（嵌套）在元素内的所有文本值的方式进行类型转换。因此，对于检测字符串 a 是否包含字符串 b 的函数 contains(a, b)，在其第一个参数被设置为一个元素时，可以用于检测元素 a 是否包含嵌套在其内部任何地方的字符串 b。XQuery 还提供在类型之间进行转换的函数。例如，number(x) 将一个字符串转换为一个数。

23.4.3.6 其他特性

XQuery 提供大量其他特性，例如 if-then-else 结构，它可以用在 **return** 子句中；以及存在量词与全称量词，它们可以应用于 **where** 子句的谓词中。例如，存在量词可以在 **where** 子句中通过使用：

some $e **in** *path* **satisfies** *P*

来表达，这里 path 是一个路径表达式，P 是一个可以使用 $e 的谓词。全称量词可以通过使用 **every** 代替 **some** 来表达。

例如，欲查找所有教师的薪酬都高于 50 000 美元的系，我们可以使用下面的查询：

```
for $d in /university/department
where every $i in /university/instructor[dept_name = $d/dept_name]
        satisfies $i/salary > 50000
return $d
```

但是请注意，如果某个系没有教师，那么该系也显然满足上述条件。可以用一个额外的子句：

```
and fn:exists(/university/instructor[dept_name = $d/dept_name])
```

1007 来保证系中至少有一名教师。子句中使用的内置函数 exists() 在输入参数不为空的情况下返回 true。

XQJ 标准提供了向 XML 数据库提交 XQuery 查询和取回 XML 结果的 API。其功能类似于 JDBC API。

23.5 XML 应用程序接口

随着人们对 XML 作为一种数据表示与交换格式的广泛认可，用于操纵 XML 数据的软件工具也大量出现。有两种程序化操纵 XML 的标准模型，每种都适用于多种流行的程序设计语言。这两种 API 都

可以用于解析一份 XML 文档并创建该文档的内存表示。它们用于处理单个 XML 文档的应用。但是注意，它们不适合查询大的 XML 数据的集合，描述性查询机制如 XPath 和 XQuery 更适合于这项任务。

其中一种流行 XML 的标准 API 基于文档对象模型(Document Object Model，DOM)，它把 XML 内容看作树，其每个元素用结点表示，称为 DOMNode。程序可以从根结点开始，以一种导航的方式来访问文档的组成部分。

DOM 库可用于大多数常用的编程语言，甚至也出现在 Web 浏览器中，用于操纵显示给用户的文档。为了对 DOM 有一个初步了解，我们在此概述一些用于 DOM 的 Java API 中的接口和方法。

- Java DOM API 提供了一个称为 Node 的接口，以及从 Node 接口继承下来的 Element 和 Attribute 接口。
- Node 接口提供诸如 getParentNode()、getFirstChild()和 getNextSibling()这样的方法，用于从根结点开始遍历 DOM 树。
- 元素的子元素可以按名字通过 getElementsByTagName(name)来访问。该函数返回一个具有指定标签名的所有子元素的列表；该列表中的单个成员可以通过 item(i)方法来访问，此方法返回列表中的第 i 个元素。
- 元素的属性值可以通过名字访问，所采用的方法是 getAttribute(name)。
- 元素的文本值被建模为 Text 结点，它是该元素结点的一个子结点；没有子元素的元素结点只有一个这样的子结点。作用在 Text 结点上的 getData()方法返回其文本内容。

DOM 还提供各种用于更新文档的函数，如增加和删除一个结点的属性及元素子结点，设置结点值，等等。

书写实际的 DOM 程序还需要更多的细节，详情请参见文献注解中的文献。

DOM 可用于访问存储在数据库中的 XML 数据，可以构建一个以 DOM 作为其访问和修改数据的主要接口的 XML 数据库。然而，DOM 接口不支持任何形式的描述性查询。

第二种通用的编程接口是 *Simple API for XML*(SAX)，它是一种事件(event)模型，被设计成提供在语法分析器和应用程序之间的通用接口。这种 API 基于事件句柄(event handler)的概念而构建，是由与语法分析事件相关联的用户指定函数构成的。语法分析事件对应于文档组成部分的识别。例如，当找到一个元素的开始标签时就产生一个事件，而当找到结束标签时产生另一个事件。一个文档的不同片段总是可以按从开始到结束的顺序找出。

SAX 应用开发者为每个事件创建句柄函数，并注册它们。当一份文档被 SAX 语法分析器读入时，随着每个事件的产生，用事件描述(如元素标签或文本内容)作为参数来调用句柄函数。然后句柄函数执行它们的任务。例如，为了构建一棵表示 XML 数据的树，用于属性或元素开始事件的句柄函数可以在一棵部分构建好的树上增加一个(或多个)结点。开始标签和结束标签的事件句柄也必须追踪树中的当前结点，新结点必须接到这个结点上；元素开始事件将新元素设置为当前结点，以使后续子结点必须连接到该结点。对应的元素结束事件将设置该结点的父结点为当前结点，后续子结点将必须连接到当前结点上。

SAX 通常比 DOM 需要更多的程序设计工作，但是当应用程序需要创建自己的数据表示时，它有助于避免创建 DOM 树的开销。如果将 DOM 用于这样的应用，创建 DOM 树将会产生不必要的空间和时间开销。

23.6 XML 数据存储

很多应用都要求存储 XML 数据。存储 XML 数据的一种方式是将其以文档形式存储在文件系统中，而第二种方式是构建专用的数据库来存储 XML 数据。还有一种方法是把 XML 数据转换为关系表示，并将其存储到关系数据库中。本节简要概括了几种存储 XML 数据的可选方法。

23.6.1 非关系的数据存储

有几种可选的方法可以在非关系的数据存储系统中存储 XML 数据：

- 存储在平面文件中(store in flat file)。既然 XML 首先是一种文件格式，一种自然的存储机制就

[1009] 是将其存储到平面文件中。第 1 章中概述了这种使用文件系统作为数据库应用基础的方法的许多缺点，尤其是它缺少数据隔离、原子性、并发访问以及安全性。然而，作用在文件数据上的 XML 工具的广泛使用使得访问和查询存储在文件中的 XML 数据相对容易一些，因此，这种存储形式可能对某些应用而言是足够的。

- **创建 XML 数据库**(create an XML database)。XML 数据库是指使用 XML 作为其基本数据模型的数据库。早期的 XML 数据库在基于 C++ 的面向对象数据库上实现了文档对象模型，这使得许多面向对象数据库基础架构得以重用，同时还提供了标准的 XML 接口。XQuery 或其他 XML 查询语言的加入提供了描述性查询。其他实现在提供事务支持的存储管理器之上建立了整个 XML 存储和查询的基础架构。

尽管有几个专门为存储 XML 数据而设计的数据库已经建立，但从底层向上构建起一个拥有完全特征的数据库系统是一项非常复杂的任务。这样的数据库不但必须支持 XML 数据存储和查询，还要支持其他数据库特性，如事务、安全性、客户端数据访问以及各种管理工具。这使得使用一个现有的数据库系统来提供这些工具，并将 XML 数据存储和查询实现在关系抽象之上，或者实现为与关系抽象相平行的层次较为合理。我们在 23.6.2 节学习这些方法。

23.6.2 关系数据库

由于关系数据库被广泛应用于现有应用程序中，将 XML 数据存储于关系数据库中使数据可以被现有应用程序所访问会带来很大的益处。

如果数据一开始就是由关系模式生成的，且 XML 仅仅是作为关系数据的一种数据交换形式来使用，那么将 XML 数据转换成关系形式通常是直截了当的。然而，在很多应用中 XML 数据并不是由关系模式生成的，将这些数据转换为关系形式进行存储可能就不那么直接了。特别地，嵌套元素以及重复出现的元素(对应于集合值属性)使 XML 数据的关系形式存储复杂化。下面我们介绍几种可选的存储方法。

23.6.2.1 存储为字符串

小的 XML 文档可以存储为关系数据库元组中的字符串(**clob**)值。对于大的 XML 文档，其顶层元素有很多子元素，可以通过将每个子元素作为字符串存储到一个单独的元组中来处理。例如，图 23-1 中的 XML 数据可以存储为 *elements*(*data*)关系中的一个元组集合，其中每个元组的 *data* 属性以字符串形式存储一个 XML 元素(department、course、instructor 或 teaches)。

[1010] 尽管上述表示方法容易使用，但数据库系统并不知道所存储元素的模式。其结果是不能直接查询数据。事实上，如果不扫描关系中的所有元组并检验每个元组中存储的字符串的内容，即使是简单的选择，诸如查找所有的 department 元素，或是查找系名为"Comp. Sci."的 department 元素，也是不可能实现的。

关于这个问题的一个部分解决方案是把不同类型的元素存储在不同的关系中，并将一些关键元素的值存储为关系的属性以便索引。比如，在我们的例子中，关系将会是 *department_elements*、*course_elements*、*instructor_elements* 以及 *teaches_elements*，每个都有一个 *data* 属性。每个关系可以有额外的属性来存储一些子元素的值，例如 *dept_name*、*course_id* 或者 *name*。因此，采用这种表示方式可以有效地回答想得到具有指定系名的 department 元素的查询。这种方案依赖于 XML 数据的类型信息，比如该数据的 DTD。

某些数据库系统，例如 Oracle，支持**函数索引**(function index)，它有助于避免 XML 字符串和关系属性之间的属性重复。不同于一般的在属性值上的索引，函数索引能够根据用户自定义函数作用于元组上的结果来构建。例如，可以根据这样一个用户自定义函数来构建函数索引，该函数返回一个元组中 XML 字符串的 dept_name 子元素的值。然后该索引能像 *dept_name* 属性上的索引一样被使用。

上述方法的缺点在于，很大一部分 XML 信息存储在字符串中，而将所有信息保存在关系中也是可行的，接下来我们将介绍几种这样的方法。

23.6.2.2 树表示法

任意的 XML 数据可以被模式化为树，并用一个关系来保存：

nodes(*id*, *parent_id*, *type*, *label*, *value*)

XML数据中的每个元素和属性都被赋予一个唯一标识符。对于每个元素和属性，都有一个元组被插入到 *nodes* 关系中。该元组的标识是 *id*，其父结点标识是 *parent_id*，结点类型是属性或元素，元素或属性的名称是 *label*，并且元素或属性的文本值是 *value*。

如果必须保存元素和属性的次序信息，可以给 *nodes* 关系增加一个额外的属性 *position*，用来指明该子结点在其父结点的所有子结点中的相对位置。作为一个习题，读者可以用这种技术来表示图23-1中的XML数据。

这种表示法的优点在于，所有的XML信息可以直接用关系形式表示出来，而且许多XML查询可以翻译为关系查询并在数据库系统内部执行。然而，它也有缺点，即每个元素都被分解成很多部分，而且将子元素重组为元素需要大量的连接运算。 1011

23.6.2.3 映射到关系

在此方法中，已知模式的XML元素被映射为关系和属性。未知模式的元素按字符串存储或按树存储。

对于每个模式已知并且其类型为复杂类型(即包含属性或子元素)的元素类型(包括子元素)都创建一个关系。如果文档的根元素没有任何属性，那么可以在该步中忽略根元素。关系的属性定义如下：

- 这些元素的所有属性都存储为该关系的以字符串为值的属性
- 如果元素的一个子元素是简单类型(也就是说，不能有属性或子元素)，就向关系中增加一个属性来表示该子元素。这个关系属性的类型在默认情况下为一个字符串值。但是如果子元素有XML Schema类型，可以使用与之对应的SQL类型。

  例如，当应用到图23-1的数据模式(DTD或者XML Schema)中的 *department* 元素时，*department* 元素的子元素 *dept_name*，*building* 以及 *budget* 都变为 *department* 关系的属性。将该过程应用于剩余元素，我们重新得到在前几章中使用过的原始关系模式。
- 否则，创建一个对应于子元素的关系(在它的子元素上递归地使用相同的规则)，并且：
  - 在代表元素的关系中增加一个标识属性。(即使元素有多个子元素，标识属性也只增加一次。)
  - 在代表子元素的关系中增加 *parent_id* 属性，用于存储其父元素的标识。
  - 如果必须保持顺序，那么在代表子元素的关系中增加 *position* 属性。

  例如，如果将上述过程应用于图23-5中数据对应的模式上，我们得到如下关系：

  department(*id*, *dept_name*, *building*, *budget*)

  course(*parent_id*, *course_id*, *dept_name*, *title*, *credits*)

这个方法还可以发生变化。例如，可以将最多出现一次的子元素所对应的关系"平面化"到其父关系中，方法是把它的所有属性移到其父关系中。文献注解中给出了将XML数据表示为关系的其他方法的参考文献。 1012

23.6.2.4 发布和分解XML数据

当XML用于在商业应用之间交换数据时，绝大多数数据来源于关系数据库。关系数据库中的数据必须被发布(publish)(即转换为XML格式)以导出给其他应用。导入的数据必须被分解(shred)，即从XML转换回规范化关系形式并存储在关系数据库中。尽管应用代码可以执行发布和分解操作，但如此通用的操作应该能够在任何可能的地方自动执行转换而无需书写应用代码。数据库厂商花费了巨大的努力以使他们的数据库产品支持XML。

一个支持XML的数据库支持把关系数据发布为XML的自动机制。用于发布数据的映射或简单或复杂。一个简单的、从关系到XML的映射可能为表的每一行创建一个XML元素，并使该行的每一列成为XML元素的一个子元素。图23-1中的XML Schema可以通过使用这种映射从大学信息的关系表示创建出来。这种映射可以直接自动生成。这种关系数据的XML视图可被视为虚拟的XML文档，并且XML查询可以在虚拟XML文档上执行。

更复杂的映射允许创建嵌套结构。人们已经开发出了在 **select** 子句中进行嵌套查询的SQL扩展，

以允许简单地创建嵌套的 XML 输出，我们在 23.6.3 节概述这些扩展。

将 XML 数据分解为关系表示的映射也必须被定义。对于由关系表示创建的 XML 数据，用于分解数据的映射就是发布这些数据所用映射的逆过程。对于一般情况，可以根据 23.6.2.3 节所介绍的方法来生成映射。

23.6.2.5 在关系数据库中的本地存储

一些关系数据库支持 XML 的**本地存储**(native storage)。这类系统将 XML 数据存储为字符串或更有效的二进制表示，而不需要将数据转换为关系形式。尽管 CLOB 和 BLOB 数据类型可以提供底层存储机制，系统还是引入了新的数据类型 **xml** 来表示 XML 数据。诸如 XPath 和 XQuery 这样的 XML 查询语言被提供用来查询 XML 数据。

一个带有 **xml** 类型的属性的关系可用于存储 XML 文档的集合；每个文档作为一个类型为 **xml** 的值存储在一个单独的元组中。创建特殊用途的索引来索引 XML 数据。

一些数据库系统提供对 XML 数据的本地支持。它们提供 **xml** 数据类型并允许在 SQL 查询中嵌套 XQuery 查询。XQuery 查询可以在单个 XML 文档上执行，并能嵌入到 SQL 查询中以允许它在一个文档集中的每份文档(其中每份文档存储在一个单独的元组中)上执行。例如，为了获得关于 Microsoft SQL Server 2005 对 XML 本地支持的更多细节，请参见 30.11 节。

1013

23.6.3 SQL/XML

虽然 XML 被广泛用于数据交换，结构化数据仍然广泛存储在关系数据库中。将关系数据转换为 XML 表示的需求已很常见。为了满足这种需求，SQL/XML 标准定义了 SQL 的一个标准扩展，允许创建嵌套的 XML 输出。这个标准有几个部分，包括将 SQL 类型映射到 XML Schema 类型的标准方法，将关系模式映射到 XML 模式的标准方法，以及 SQL 查询语言扩展。

例如，*department* 关系的 SQL/XML 表示将有一个 XML Schema，其最外层元素为 *department*，每个元组映射为一个 XML 元素 *row*，且每个关系属性映射为一个同名(通过一些转换解决与名字中特殊字符的不相容问题)的 XML 元素。一个完整的、有多个关系的 SQL 模式也可以用类似的方式映射为 XML。图 23-15 给出了包含 *department* 和 *course* 关系的图 23-1 中(部分)*university* 数据的 SQL/XML 表示。

```
<university>
    <department>
        <row>
            <dept_name> Comp. Sci. </dept_name>
            <building> Taylor </building>
            <budget> 100000 </budget>
        </row>
        <row>
            <dept_name> Biology </dept_name>
            <building> Watson </building>
            <budget> 90000 </budget>
        </row>
    </department>
    <course>
        <row>
            <course_id> CS-101 </course_id>
            <title> Intro. to Computer Science </title>
            <dept_name> Comp. Sci </dept_name>
            <credits> 4 </credits>
        </row>
        <row>
            <course_id> BIO-301 </course_id>
            <title> Genetics </title>
            <dept_name> Biology </dept_name>
            <credits> 4 </credits>
        </row>
    </course>
</university>
```

图 23-15 (部分)大学信息的 SQL/XML 表示

SQL/XML 向 SQL 中增加了一些操作符和聚集运算以允许从扩展 SQL 中直接创建 XML 输出。**xmlelement** 函数可用于创建 XML 元素，而 **xmlattributes** 可用于创建属性，如下面的查询所示：

```
select xmlelement( name "course",
                   xmlattributes( course_id as course_id, dept_name as dept_name),
                   xmlelement( name "title", title),
                   xmlelement( name "credits", credits) )
from course
```

上述查询为每门课程创建一个 XML 元素，其中课程标识符和系名用属性来表示，课程名和学分用子元素表示。查询结果看起来像是图 23-11 中的 course 元素，但是没有 instructor 属性。**xmlattributes** 操作符用 SQL 属性名生成 XML 属性名，也可以像上述查询那样替换为使用 **as** 子句。

xmlforest 操作符简化了 XML 结构的构造。它的语法和行为与 **xmlattributes** 类似。只不过它创建的是一个子元素的森林(集合)，而不是属性的列表。该操作符有多个变元，它为每个变元创建一个元素，使用属性的 SQL 名作为 XML 元素名。**xmlconcat** 操作符可用于将由子表达式创建的元素连接成一

片森林。

当用于构建属性的SQL值为空时，忽略该属性。在构建元素主体时，忽略空值。

SQL/XML还提供一个聚集函数 **xmlagg**，它根据所操作的值集合创建一个XML元素的森林(集合)。下述查询为每个开设了课程的系创建一个元素，该系所开设的所有课程作为其子元素。由于查询中有 **group by** *dept_name* 子句，聚集函数将作用在每个系开设的所有课程上，创建出 *course_id* 元素的一个序列。

```
select xmlelement(name "department",
                  dept_name,
                  xmlagg(xmlforest(course_id)
                         order by course_id))
from course
group by dept_name
```

如上述查询所示，SQL/XML允许对 **xmlagg** 创建的序列排序。在文献注解部分给出的参考文献中可以找到关于SQL/XML的更多信息。 1014 1015

23.7 XML应用

我们现在概括介绍XML在存储和传递(交换)数据以及访问Web服务(信息资源)方面的一些应用。

23.7.1 存储复杂结构数据

很多应用需要存储结构化的但不易建模为关系的数据。例如考虑一个应用(如浏览器)必须要存储用户的使用偏好设置。通常有大量的域必须被记录，如主页、安全设置、语言设置和显示设置。一些域是多值的，如一个信任站点的列表，或是如书签列表那样的可能有序的列表。传统上，应用程序使用某种文本表示类型来存储这些数据。今天，大多数这种应用更愿意以XML格式来存储这些配置信息。早先使用的即席文本表示需要做很多工作来进行设计，以及创建词法分析器以读取文件并将数据转化为程序可以使用的形式，而XML表示避免了这些步骤。

基于XML的表示现在已被广泛用作存储文档、电子表单数据以及作为办公应用软件包的一部分的其他数据。Open Office软件包以及其他办公软件包所支持的开放文档格式(ODF)以及微软办公软件包所支持的Office Open XML(OOXML)格式都是基于XML的文档表示标准。它们是两种使用最广泛的可编辑文档表示形式。

XML还用于表示必须在一个应用的不同部分之间进行交换且具有复杂结构的数据。例如，数据库系统可能使用XML来表示查询执行计划(一个关系代数表达式，带有额外的关于如何执行操作的信息)。这允许在没有共享数据结构的情况下，系统的一部分生成查询执行计划而另一部分显示它。例如，数据可能由服务器系统生成，并发送到显示数据的客户端系统。

23.7.2 标准化数据交换格式

用于各种特殊应用的、基于XML的数据表示标准已经开发出来了，其应用范围很广，从商业应用(比如银行业和运输业)到科学应用(比如化学和分子生物学)。下面是一些例子：

- 化学工业需要有关化学制品的信息，例如它们的分子结构及其各种重要属性，如沸点和熔点、热值、在不同溶剂中的溶解性。ChemML就是表示此类信息的一个标准。
- 在运输业中，货物的承运者、客户以及税务官员需要货物的运输记录，其中包括关于所运输的货物、由何人发往何地、送达给何人以及送达何地、货物的货币价值等详细信息。 1016
- 一个能够进行商品买卖的在线商场(所谓的B2B市场)需要的信息如商品目录(包括详细的商品描述和价格信息)、存货清单、建议售价和购物订单。例如，用于电子商务应用的RosettaNet标准为数据表示定义了XML模式和语义，并为信息交换定义了标准。

用规范化关系模式对这种复杂的数据需求进行建模将会产生出大量的关系，这些关系并不直接对应于被建模的对象。这样的关系通常会有大量属性；在XML中显式地将属性/元素的名字与值表示在一起有助于避免属性间的混淆。嵌套元素表示有助于在冗余代价可以接受的条件下，减少必须被表示

的关系的数量，以及为获得必要信息所需要的连接次数。举例来说，在我们的大学的例子中，如图 23-5 那样列出在 department 元素中嵌套了 course 元素的系而生成的格式对某些应用而言比图 23-1 中的规范化表示更加自然，特别是在供人阅读方面。

23.7.3 Web 服务

应用常常需要从组织外部获取数据，或从同一组织中使用不同数据库的其他部门获取数据。在很多这样的情况下，外部组织或部门并不希望用 SQL 直接访问它的数据库，但是愿意通过预定义接口提供受限形式的信息。

当信息直接由人使用时，组织机构提供基于 Web 的形式，用户可以输入值并获得 HTML 形式的所需信息。但是在很多应用中，这样的信息需要被软件程序访问，而不是由终端用户访问。提供 XML 形式的查询结果是一种明确的需求。除此以外，用 XML 格式为查询指定输入值也是有意义的。

本质上，信息提供者定义过程，这些过程的输入和输出都是 XML 格式的。由于 HTTP 协议被广泛应用且可以穿越机构为防止来自互联网的、不需要的通信而使用的防火墙，因此它被用来传输输入和输出信息。

简单对象访问协议(Simple Object Access Protocol，SOAP)为过程调用定义了一个标准，它使用 XML 来表示过程的输入和输出。SOAP 为表示过程名和结果状态指示符(如失败/错误指示符)定义了标准的 XML schema。过程的参数和结果是嵌入在 SOAP XML 标题中的、依赖于应用的 XML 数据。

典型情况下，使用 HTTP 作为 SOAP 的传输协议，但是也可能使用一个基于消息的协议(例如
1017 SMTP 协议上的电子邮件)。目前 SOAP 标准被广泛使用。例如，Amazon 和 Google 提供基于 SOAP 的过程以执行搜索和其他活动，这些过程可以被为用户提供高层服务的其他应用调用。SOAP 标准独立于下层的程序设计语言，一个运行某种语言(如 C#)的站点调用运行在一个不同语言(如 Java)上的服务是可能的。

提供这样一个 SOAP 过程集合的站点被称作 **Web 服务**(Web service)。目前已经定义了一些标准以支持 Web 服务。**Web 服务描述语言**(Web Services Description Language，WSDL)是用于描述 Web 服务能力的语言。WSDL 提供传统程序设计语言中接口定义(或函数定义)所提供的工具，指定什么函数是可用的，以及它们的输入和输出类型。除此以外，WSDL 允许在调用 Web 服务时指定 URL 和所使用的网络端口号。还有一个标准称为**通用描述、发现和集成**(Universal Description，Discovery，and Integration，UDDI)，它定义了如何创建一个可用 Web 服务的目录，以及程序如何在目录中搜索以找到满足其需求的 Web 服务。

下面的例子展示了 Web 服务的价值。一家航空公司可能定义了一个 Web 服务，提供一组可以被旅行网站调用的过程，这其中可能包括用于查找航班计划和价格信息的过程，以及用于预订航班的过程。旅行网站可能与由不同航空公司、酒店和其他公司提供的多个 Web 服务交互，以向顾客提供旅行信息并进行旅程预定。通过支持 Web 服务，每家公司允许在顶层构建一个有用的服务，它整合了各个独立的服务。用户只需与这一个网站交互就可以进行旅程预定，而不需要联系多个独立的网站。

要调用一个 Web 服务，客户端必须准备适当的 SOAP XML 消息，并将它发送给服务；当接收到以 XML 编码的结果时，客户端必须从 XML 结果中提取信息。在诸如 Java 和 C#那样的程序设计语言中有标准的 API，用于从 SOAP 消息中创建和提取信息。

在文献注解部分的参考文献中可以找到关于 Web 服务的更多信息。

23.7.4 数据中介

比较购物是数据中介的一个实例，其中关于商品、库存、价格以及运送费用的数据是从销售特定商品的各种不同网站中提取出来的。聚集起来的结果信息要比单个站点提供的单独信息有价值得多。

个人财务管理程序是在银行业务环境中的类似应用。考虑一个消费者，他有很多不同的账户要管理，比如银行账户、信用卡账户，以及退休金账户。假设这些账户由不同机构所持有，提供对客户所

有账户的集中式管理是我们要面临的一个主要的挑战。基于 XML 的中介对该问题的处理是通过从持有客户账户的各个金融机构各自的网站上提取客户信息的 XML 表示来解决的。如果机构将这些信息以标准 XML 格式(如 Web 服务)导出，这些信息就可以很容易地提取出来。对于不提供标准 XML 格式导出信息的机构，可以使用包装器(wrapper)软件来根据网站返回的 HTML 网页生成 XML 数据。包装器应用需要经常维护，因为它们依赖于 Web 页面的格式细节，而这些细节是经常变化的。尽管如此，中介带来的价值常常证实为开发和维护包装器所付出的努力是值得的。 [1018]

一旦有了能够从各数据源提取信息的基础工具，就可以使用中介(mediator)应用在一个单一的模式下将提取到的信息综合起来。这可能需要对来自各个站点的 XML 数据作进一步的转换，因为不同的站点可能用不同的方式构造相同的信息。它们还可能用不同的名字为同样的信息命名(如 acct_number 和 account_id)，或者甚至可能使用相同的名字表示不同的信息。中介必须决定一种单一的模式来表示所有必要信息，同时必须提供在不同表示形式之间进行数据转换的代码。在 19.8 节介绍分布式数据库时对这样的问题进行了更详细的讨论。XML 查询语言，如 XSLT 和 XQuery，在不同 XML 表示之间进行转换这一任务上发挥了重要作用。

23.8 总结

- 如同 Web 所基于的超文本标记语言(HTML)一样，可扩展标记语言(XML)也是源自标准通用标记语言(SGML)。XML 原本是为了给 Web 文档提供功能标记，但如今它已经成为应用间数据交换格式的事实标准。
- XML 文档包含元素，元素的开头和结尾用相匹配的开始和结束标签来标记。元素可以包含任意多层嵌套的子元素。元素还可以有属性。在数据表示中，通常可以任意选择用属性或是子元素来表示信息。
- 元素可以有一个类型为 ID 的属性，用于存储该元素的唯一标识符。通过使用类型为 IDREF 的属性，元素还可以存储指向其他元素的引用。类型为 IDREFS 的属性可以存储一个引用列表。
- 文档可以选择通过文档类型定义(DTD)来指定其模式。文档的 DTD 指定什么元素可以出现，它们如何嵌套，以及每个元素可以有什么属性。尽管 DTD 被广泛使用，但它有一些局限。例如，它们不能提供类型系统。
- XML Schema 是旨在指定 XML 文档模式的标准机制。它提供大量的基本类型，以及用于创建复杂类型和声明完整约束(包括键约束和外键(keyref)约束)的结构。 [1019]
- XML 数据可以用树结构表示，树结点对应于元素和属性。元素的嵌套由树表示中的父－子结构来反映。
- 路径表达式可用于遍历 XML 树结构和定位数据。XPath 是路径表达式的一种标准语言。它允许用类似于文件系统路径的方式来指定所需元素，另外它还支持选择和其他特性。XPath 还构成了其他 XML 查询语言的一部分。
- XQuery 语言是查询 XML 数据的标准语言。它有与 SQL 相似的结构，包括 **for**、**let**、**where**、**order by** 和 **return** 子句。然而它支持很多用于处理 XML 树特性的扩展，并且允许将 XML 文档转换为另一种具有明显不同结构的文档。XPath 路径表达式构成了 XQuery 的一部分。XQuery 支持嵌套查询和用户自定义函数。
- DOM 和 SAX API 广泛应用于对 XML 数据的程序式访问。在很多程序设计语言中这些 API 都是可用的。
- 可以选择几种不同的方法来存储 XML 数据。XML 数据还可以存储在文件系统中，或存储在使用 XML 作为其内部表示的 XML 数据库中。
- XML 数据可以存储为关系数据库中的字符串。或者，用一些关系将 XML 数据表示为树。作为另一种选择，可以按照将 E-R 图映射为关系模式的同样方式将 XML 数据映射为关系。通过在 SQL 中增加 **xml** 数据类型使得关系数据库中的 XML 本地存储更加容易。
- XML 被使用在各种应用中，如存储复杂数据，在组织之间以一种标准化形式交换数据，数据中介以及 Web 服务。Web 服务使用以 XML 作为编码参数和结果的机制，提供远程过程调用接口。

术语回顾

- 可扩展标记语言(XML)
- 超文本标记语言(HTML)
- 标准通用标记语言
- 标记语言
- 标签
- 自描述
- 元素
- 根元素
- 嵌套元素
- 属性
- 名字空间
- 默认名字空间
- 模式定义
- 文档类型定义
  - ID
  - IDREF 和 IDREFS
- XML Schema
  - 简单和复杂类型
  - 序列类型
  - key 和 keyref
  - 出现次数约束
- XML 数据的树模型
- 结点
- 查询和转换
- 路径表达式
- XPath
- XQuery
  - FLOWR 表达式
    - for
    - let
    - where
    - order by
    - return
  - 连接
  - 嵌套 FLWOR 表达式
  - 排序
- XML API
- 文档对象模型(DOM)
- XML 简单 API(SAX)
- XML 数据存储
  - 非关系数据存储
  - 关系数据库存储
    - 存储为字符串
    - 树表示形式
    - 映射到关系
    - 发布和分解
    - 支持 XML 的数据库
    - 本地存储
    - SQL/XML
- XML 应用
  - 存储复杂数据
  - 数据交换
  - 数据中介
  - SOAP
  - Web 服务

实践习题

23.1 给出包含图 23-1 中相同数据的大学信息的另一种表示，但是使用属性而非子元素。同时给出该表示的 DTD 或 XML Schema。

23.2 给出用于下述嵌套关系模式的 XML 表示的 DTD 或 XML Schema：

$$Emp = (ename,\ ChildrenSet\ \textbf{setof}(Children),\ SkillsSet\ \textbf{setof}(Skills))$$
$$Children = (name,\ Birthday)$$
$$Birthday = (day,\ month,\ year)$$
$$Skills = (type,\ ExamsSet\ \textbf{setof}(Exams))$$
$$Exams = (year,\ city)$$

1020 | 1021

23.3 基于实践习题 23.2 中的模式，用 XPath 书写查询列出 *Emp* 中的所有技能类型。

23.4 在图 23-11 中的 XML 表示上用 XQuery 书写查询，找出每个系所有教师的薪酬总额。

23.5 在图 23-1 中的 XML 表示上用 XQuery 书写查询，计算 department 元素与 course 元素的左外连接。(提示：使用全称量词。)

23.6 给定图 23-11 中使用了 ID 和 IDREFS 的大学信息表示，用 XQuery 书写出查询来输出内部嵌套有相关联的 course 元素的 department 元素。

23.7 给出一个关系模式来表示图 23-16 中的 DTD 片断所指定的书目信息。关系模式必须保持 author 元素的顺序。可以假设只有 books 和 articles 作为 XML 文档中的顶层元素出现。

```
<! DOCTYPE bibliography [
      <! ELEMENT book( title, author + , year, publisher, place?) >
      <! ELEMENT article( title, author + , journal, year, number, volume, pages?) >
      <! ELEMENT author( last_name, first_name) >
      <! ELEMENT title( #PCDATA) >
      …对 year, publisher, place, journal, year, number, volume, pages, last_name
      和 first_name 的类似 PCDATA 声明)
] >
```

图 23-16 书目数据的 DTD

23.8 给出图23-1中XML数据的树表示，并使用23.6.2节中描述的 *nodes* 和 *child* 关系来表示该树。

23.9 考虑如下递归DTD：

```
<!DOCTYPE parts [
        <!ELEMENT part (name, subpartinfo*)>
        <!ELEMENT subpartinfo (part, quantity)>
        <!ELEMENT name(#PCDATA)>
        <!ELEMENT quantity(#PCDATA)>
]>
```

a. 给出对应于上述DTD的一个小的数据实例。

b. 给出如何将这个DTD映射成关系模式。可以假设part的名称是唯一的，即无论一个part出现在哪里，它的subpart结构都是一样的。

c. 用XML Schema创建一个对应于此DTD的模式。 1022

习题

23.10 通过给定一个DTD，给出如何用XML来表示22.2节中的非1NF的 *books* 关系。

23.11 用XQuery写出如下查询，使用实践习题23.2中的模式。

a. 找出孩子生日在三月份的所有雇员的名字。

b. 找出那些在"Dayton"城市参加过技能类型为"typing"的考试的雇员。

c. 列出 *Emp* 中的所有技能类型。

23.12 考虑图23-3中给出的XML数据。假设我们希望找到订购了两份或两份以上标识符为123的部件的购物订单。观察下面这个为解决该问题而做的尝试：

```
for sp in purchaseorder
where sp/part/id = 123 and sp/part/quantity >= 2
return sp
```

解释为什么这个查询可能返回某些订购不到两份123部件的购物订单，给出正确的查询。

23.13 用XQuery写出一个查询来转换习题23.10中的数据嵌套，也就是说，在嵌套的最外层，输出中必须有对应于作者的元素，并且每个这样的元素必须在内部嵌套对应于该作者所写的所有书的项。

23.14 给出图7-29中信息的XML表示的DTD。创建一个单独的元素类型来表示每种关系，但是要使用ID和IDREF来实现主码和外码。

23.15 给出习题23.14中DTD的XML Schema表示。

23.16 用XQuery对图23-16中书目的DTD片断写出以下查询：

a. 找出在同一年内发有一本书和一篇文章的所有作者。

b. 显示按年排序的书和文章。

c. 列出有一个以上作者的书。

d. 找出所有在标题中包含"database"且作者名字中包含"Hank"（不管是姓还是名）的书。 1023

23.17 给出图23-3中所示XML购物订单模式的一个关系映射，使用23.6.2.3节所介绍的方法。如果商品的标识函数决定商品描述，购买者和供货商名字分别函数决定购买者和供货商的地址，给出如何去除关系模式中冗余的建议。

23.18 用SQL/XML写出查询语句，将大学数据从我们在前面章节所使用的关系模式转换为 *university-1* 和 *university-2* 的XML Schema。

23.19 同习题23.18一样，写出查询将大学数据转换到 *university-1* 和 *university-2* 的XML Schema，但是这次是通过在默认的SQL/XML数据库到XML的映射上书写XQuery查询来实现。

23.20 一种分解XML文档的方法是使用XQuery将模式转化为相应关系模式的SQL/XML映射，然后反向使用SQL/XML映射以生成关系。

作为示例，给出一个XQuery查询将数据从 *university-1* XML Schema转化到图23-15所示的SQL/XML模式。

23.21 考虑23.3.2节中的XML Schema范例，写出XQuery查询完成如下任务：

a. 检查是否满足23.3.2节所示的键约束。

b. 检查是否满足 23.3.2 节所示的键引用约束。

23.22 考虑实践习题 23.7，假设作者还可能作为顶层元素出现。那么需要对关系模式进行什么修改？

工具

公用领域内已经有很多用于处理 XML 的工具。W3C 网站 www.w3.org 上有介绍各种 XML 相关标准的页面，以及指向软件工具（如语言实现）的链接。很多 XQuery 实现的列表可以在 www.w3.org/XML/Query 上获得。Saxon D（saxon.sourceforge.net）和 Galax（http://www.galaxquery.org/）作为学习工具也有用的，虽然它们都没有被设计用来处理大数据库。Exist（exist-db.org）是一个开源的 XML 数据库，它支持很多特性。一些商业数据库，包括 IBM DB2、Oracle 和 Microsoft SQL Server 支持 XML 的存储，通过各种 SQL 扩展进行发布以及使用 XPath 和 XQuery 的查询。

文献注解

1024 万维网联盟（W3C）作为 Web 相关标准的标准主体，包括基本 XML 以及所有 XML 相关语言，如 XPath、XSLT 和 XQuery，在 www.w3.org 上有大量定义 XML 库关标准的技术报告。这个网站还包含关于实现各种标准的软件的指南和链接。

XQuery 语言是从一种称作 Quilt 的 XML 查询语言发展而来的。Quilt 本身包括了早期语言（如在 23.4.2 节中讨论的 XPath，以及其他两种 XML 查询语言：XQL 和 XML-QL）所拥有的很多特性。Chamberlin 等[2000]介绍了 Quilt。Deutsch 等[1999]介绍了 XML-QL 语言。W3C 在 2009 年中期发布了对 XQuery 进行扩展的一个推荐候选（candidate recommendation），其中包含对更新的支持。

Katz 等[2003]提供了涵盖 XQuery 的详细教科书。可以在 www.w3.org/TR/xquery 上找到 XQuery 规范。该网站还提供了对 XQuery 扩展的规范，包括 XQuery 更新工具和 XQuery 脚本扩展。Florescu 等[2000]和 Amer-Yahia 等[2004]概述了将关键字查询集成到 XML 的方法。

Funderburk 等[2002a]、Florescu 和 Kossmann[1999]、Kanne 和 Moerkotte[2000]，以及 Shanmugasundaram 等[1999]介绍了 XML 数据的存储。Eisenberg 和 Melton[2004a]提供了 SQL/XML 的概览，而 Funderburk 等[2002b]概括介绍了 SQL/XML 和 XQuery。请参考第 28 章～第 30 章以获得关于商用数据库对 XML 支持的更多信息。Eisenberg 和 Melton[2004b]对 XQuery 的 XQJ API 进行了概述，而标准定义可以在 http://www.w3.org/TR/xquery 上找到。

XML 索引、查询处理和查询优化：过去几年中，对 XML 数据的索引以及 XML 查询处理和优化是一个兴趣浓厚的领域。该领域发表了很多论文。索引中遇到的一个挑战是：查询可以指定在路径上的选择，例如/a/b//c[d = "CSE"]。通过索引必须能够高效地获取满足路径指定和值选择的结点。在 XML 数据索引方面的工作包括 Pal 等[2004]和 Kaushik 等[2004]。如果数据是被分解的并存放于关系中，对路径表达式的计算对应于对连接的计算。人们已经提出了几种技术来高效地计算这类连接，特
1025 别是在路径表达式指定了任意后继结点（//）的情况下。人们还提出了几种对 XML 数据中的结点进行编
1026 号的技术，用于高效地检查一个结点是否是另一个结点的后继，参见诸如 O'Neil 等[2004]。XML 查询优化方面的工作包括 McHugh 和 Widom[1999]、Wu 等[2003]以及 Krishnaprasad 等[2004]。

第八部分
Part 8

高级主题

第24章包括了许多应用开发中的高级主题，首先介绍了为提高应用系统速度而进行的性能调优；接着讨论了衡量商用数据库系统性能的基准；然后叙述了应用开发中诸如应用测试和应用移植的问题。本章以对标准化过程和现有数据库语言标准的总结作为结束。

第25章描述了空间和时态数据类型、多媒体数据，以及在数据库中存储这些数据的问题，本章同时还介绍了有关移动计算系统的数据库问题。

最后，第26章描述了几个高级事务处理技术，包括事务处理监视器、事务工作流和电子商务中的事务处理问题；然后讨论了主存数据库系统和实时事务系统，并以长事务的讨论作为结束。

1027
1028

第 24 章
Database System Concepts, 6E

高级应用开发

应用开发有许多任务。我们已经在第 7 章到第 9 章看到了如何设计和构建应用系统。对应用系统期望的性能是应用系统设计的一个方面。事实上，一旦应用系统建立以后，人们常常发现它的运行速度比设计者期望的要慢，或者每秒钟处理的事务数量少于需求的数量。一个应用系统花费过长时间执行请求的动作，在最好的情况下会引起用户的不满，在最坏的情况下应用系统完全不能使用。

通过性能调整可以大幅度提高应用系统的运行速度。性能调整包括发现和消除瓶颈，以及添加适当的硬件，如内存或硬盘。应用系统开发者可以有多种方法进行性能调整。数据库系统管理员也可以有多种方式来提高应用系统的处理速度。

基准是一些任务的标准化集合，它们有助于刻画数据库系统的性能特征。甚至在应用系统构建之前，基准对于大致了解应用系统硬件和软件的要求都是很有用的。

应用系统在开发过程中必须进行测试。测试要求生成数据库状态和测试输入，并验证输出是否匹配预期的输出。我们讨论应用系统测试的问题。遗产系统是已经过时的、通常基于老一代技术的应用系统。然而，它们常常位于组织机构的核心，运行一些关键任务应用。我们概述了与遗产系统接口的问题，如何移植遗产系统的问题，以及如何用重现代的系统取代遗产系统。

标准对于应用系统开发来说非常重要，特别是在互联网时代，因为应用之间需要相互通信以执行有用的任务。人们已经提出的各种不同的标准会影响数据库应用的开发。

24.1 性能调整

[1029] 系统性能调整包括调整各种参数和选择设计方案，以提高系统在特定应用下的性能。数据库系统设计的各个方面（其范围从高层次方面，如模式和事务设计，到数据库参数，如缓冲区大小，直到硬件问题，如磁盘数目）都影响着应用系统的性能。为了提高系统性能，所有这些方面都可以调整。

24.1.1 提高面向集合的特性

当来自一个应用程序的 SQL 查询被执行时，通常情况下一个查询会被频繁地执行，但是带有不同的参数值。每次调用除了在服务器上的处理开销，还有与服务器通信的开销。

例如，考虑一个程序，它遍历各个系，调用嵌入式的 SQL 查询来找出系中所有教师的薪酬总额：

```
select sum(salary)
from instructor
where dept_name = ;
```

如果 *instructor* 关系在 *dept_name* 上没有聚集索引，每个这样的查询将导致对整个关系扫描一遍，即使有这样的索引，对每个 *dept_name* 的值也会有随机的 I/O 操作请求。

相反，我们可以使用单个 SQL 查询来找出每个系的总薪酬费用：

```
select dept_name, sum (salary)
from instructor
group by dept_name;
```

这个查询通过对 *instructor* 关系的单遍扫描来实现，从而避免对每个系的随机 I/O。其结果可以通过单次通信提取到客户端，然后客户端程序可以遍历结果来找出每个系的聚集值。像上述那样把多个 SQL 查询合并为单个 SQL 查询在很多情况下可以大大降低执行开销。例如，如果 *instructor* 关系非常

大，并且有很多的系。

JDBC API 也提供了一个称作**批量更新**（batch update）的功能，允许使用单次数据库通信来执行多个插入操作。图 24-1 展示了这个功能的使用情况。当执行 executeBatch（）方法时，图中所示的代码只需要与数据库通信一次，相比之下，我们前面在图 5-2 中看到的类似代码没有批量更新功能。在没有批量更新的情况下，需要与数据库通信的次数与待插入的教师一样多。批量更新功能也使得数据库可以一次处理一批插入，这很可能会比一连串的单记录插入更为有效。

```
PreparedStatement pStmt = conn.prepareStatement(
                "insert into instructor values(?,?,?,?)");
pStmt.setString(1, "88877");
pStmt.setString(2, "Perry");
pStmt.setInt(3, "Finance");
pStmt.setInt(4, 125000);
pStmt.addBatch();
pStmt.setString(1, "88878");
pStmt.setString(2, "Thierry");
pStmt.setInt(3, "Physics");
pStmt.setInt(4, 100000);
pStmt.addBatch(); pStmt.executeBatch();
```

图 24-1 JDBC 中的批量更新

在客户端—服务器系统中广泛使用的另一种用来减少通信开销和 SQL 编译的技术是使用存储过程。其中查询以过程的形式存储在服务器上，它们可以被预编译。客户端可以调用这些存储过程，而不是发送一系列查询。

提高面向集合特性的另一个方面在于用**嵌套子查询**（nested subquery）重写查询。在过去，许多数据库系统的优化器不是特别好，所以一个查询是如何写的将极大地影响到它是如何被执行的，从而对性能也有很大影响。当今先进的优化器甚至可以改变写得很糟糕的查询，并有效地执行它们，所以调整单个查询的需求没有以前那么重要了。然而，许多优化器并不能很好地优化包含嵌套子查询的复杂查询。

我们在第 13.4.4 节看到了去除嵌套子查询相关性的技术。如果一个子查询没有被去除相关性，它会被重复地执行，这可能导致大量的随机 I/O。与此相反，去除相关性允许使用高效的面向集合操作，例如连接，来最小化随机 I/O。大多数数据库查询优化器引入了一些形式的去除相关性，但有些只能处理很简单的嵌套子查询。优化器选择的执行计划是在前面第 13 章描述的。如果优化器不能成功去除嵌套子查询的相关性，该查询可以通过手工重写来去除相关性。

24.1.2 批量加载和更新的调整

当把大量数据加载到数据库中（称为**批量加载**（bulk load）操作）时，如果通过单独的 SQL 插入语句来执行插入，性能通常会非常差。其中一个原因是解析每个 SQL 查询的开销；一个更重要的原因是，为每个插入的元组分别执行完整性约束检查和索引更新会导致大量的随机 I/O 操作。如果大批量执行插入，可以以一种更加面向集合的方式完成完整性约束检查和索引更新，从而大大降低开销，把性能提升一个数量级或更高并不少见。

1030
1031

为了支持批量加载操作，大多数数据库系统提供了**批量导入**（bulk import）功能，以及相应的**批量导出**（bulk export）功能。批量导入功能从文件读取数据，并以一种非常高效的方式执行完整性约束检查以及索引维护。这样的批量导入/导出功能支持的常见输入和输出文件格式包括文本文件，它们带有如逗号或制表符那样的符号来分隔属性值，每个记录单独占一行（这样的文件格式称为逗号分隔的值或制表符分隔的值格式）。批量导入/导出功能也支持数据库特定的二进制格式以及 XML 格式。批量导入/导出功能的名字随数据库不同而不同。在 PostgreSQL 里，这些功能被称为 pg_dump 和 pg_restore（PostgreSQL 也提供了一个 SQL 命令 **copy**，它提供了类似的功能）。Oracle 里的批量导入/导出功能称为 **SQL*Loader**，在 DB2 里该功能被称为 load，在 SQL Server 里该功能被称为 bcp（SQL Server 还提供了一个称作 **bulk insert** 的 SQL 命令）。

我们现在考虑调整批量更新的情况。假设我们有一个关系 *funds_received*（*dept_name*，*amount*），它为每个系存储到账的资金（例如，通过电子资金转账）。假设现在我们想给相应系的预算余额增加经费。为了使用 SQL 更新语句来完成这项任务，我们必须为 *department* 关系中的每个元组在 *funds_received* 关系上执行一次查找。我们可以按如下方式使用更新子句中的子查询来完成这项任务：为简单起见，我们假设 *funds_received* 关系中对每个系最多包含一个元组。

```
update department set budget = budget +
    (select amount
     from funds_received
     where funds_received.dept_name = department.dept_name)
where exists(
    select *
    from funds_received
    where funds_received.dept_name = department.dept_name);
```

注意更新语句的 **where** 子句里的条件确保了只有在 *funds_received* 中有相应元组的账户才会被更新，而 **set** 子句中的子查询则计算出要给每个这样的系增加的经费量。

有许多应用程序需要如上所示的更新操作。通常情况下，有一个我们称之为主表（master table）
1032 的表，对主表的更新是批量进行的。现在主表必须相应地更新。SQL：2003 提供了一种特殊结构，称为 **merge** 结构，来简化执行这样的信息合并的任务。例如，上面的主要可以使用 **merge** 表示如下：

```
merge into department as A
using   (select *
         from funds_received) as F
      on (A.dept_name = F.dept_name)
    when matched then
        update set budget = budget + F.amount;
```

当来自 **using** 子句子查询的一条记录与 *department* 关系中的一条记录匹配时，就执行 **when matched** 子句，它可以在该关系上执行一个更新操作；在这种情况下，在 *department* 关系中匹配的记录按如上所示的方式被更新。

merge 语句还可以有一个 **when not matched then** 子句，它允许往关系中插入新记录。在上面的例子里，当 *funds_received* 的元组没有匹配上的系，插入动作可以使用以下子句来创建一个新的系记录（在 *building* 值上取空值）：

```
when not matched then
    insert values (F.dept_name, null, F.budget)
```

虽然在这个例子里不是非常有意义[3]，但在其他情况下 **when not matched then** 子句是相当有用的。例如，假设本地关系是主关系的一个副本，且我们从主关系接收到了更新的以及新插入的记录。**merge** 语句可以更新匹配上的记录（这些是被更新的旧记录）并且插入未匹配上的记录（这些是新记录）。

目前并非所有的 SQL 实现都支持 **merge** 语句，进一步的细节请参见相应的系统手册。

24.1.3 瓶颈位置

大多数系统的性能（至少在它们调整之前）通常主要受制于一个或几个部件的性能，这样的部件称为瓶颈（bottleneck）。例如，一个程序可能有 80% 的时间花在代码中的一个小循环上，而其余 20%
1033 的时间用在剩余的代码上；那么这个小循环就是一个瓶颈。提高非瓶颈部件的性能对于提高系统的总体速度没多大帮助，在上述例子中，提高剩余代码的速度不可能使总体速度提高 20% 以上，而提高该瓶颈循环的速度在最好的情况下可以将总体速度提高将近 80%。

因此，在对系统进行性能调整时，我们必须首先试着找出瓶颈是什么，然后通过提高导致这些瓶颈的系统部件的性能来消除瓶颈。消除一个瓶颈后，可能另一个部件又成为瓶颈。在一个性能均衡的系统中，任何单个的部件都不会成为瓶颈。如果系统中存在瓶颈，没有构成瓶颈的那些部件就不能被充分使用，因而可以使用性能较低、价格较便宜的部件来代替。

对简单的程序来说，在代码的各个部分花费的时间决定了总体执行时间。然而，数据库系统要复杂得多，可以用排队系统（queueing system）来建模。一个事务向数据库系统请求各种服务，从进入一个服务器进程开始，在执行过程中需要读磁盘，需要 CPU 周期，以及并发控制所需要的锁。每个

[3] 在这里一个更好的动作是把该新记录插入到一个错误关系中，但是用 merge 语句不能这样来实现。

这样的服务都有一个队列与之相联，而小事务可能把它们大多数的时间都花费在队列(尤其是磁盘 I/O 队列)等待上，而不是代码的执行上。图 24-2 举例说明了数据库系统中的一些队列。

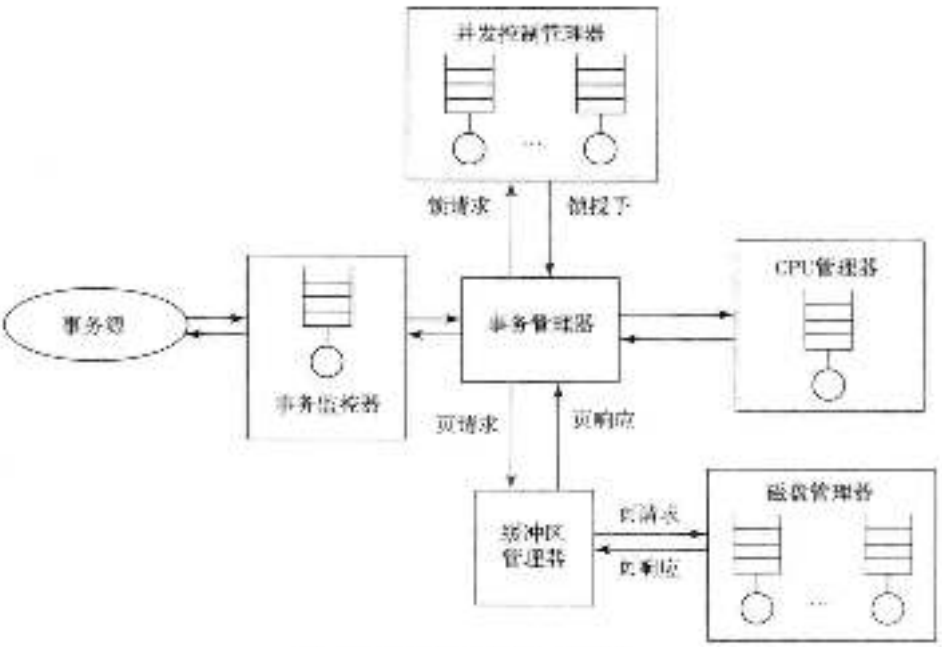


图 24-2 数据库系统中的队列

由于数据库系统中存在大量队列，数据库系统中的瓶颈常常表现为某个特定服务的队列很长，或者等价地说，某个特定服务的利用率很高。如果请求的分布非常均匀，并且为一个请求服务的时间小于或等于下一请求到来的时间，那么每个请求都会发现资源是空闲的，因此可以立即开始执行而不需等待。遗憾的是，数据库系统中请求的到达从不是均匀的，而总是随机的。

如果一个资源(如磁盘)的利用率低，那么当提出请求时，该资源很可能是空闲的，这时请求的等待时间为 0。假设请求的到达均匀地随机分布，队列长度(以及相应的等待时间)随利用率呈指数增长；当利用率接近 100%时，队列长度急剧增加，导致等待时间过长。因此资源的利用率应保持足够低，使队列长度很短。经验表明，利用率在 70%左右较好，而超过 90%就太大，因为这会带来显著的延迟。要进一步了解关于排队系统理论(通常称作**排队论**(queueing theory))的更多知识，请参阅文献注解中所给出的参考文献。

24.1.4 可调参数

数据库管理员可以在三个级别上对数据库系统进行调整。最低级别是在硬件层上。这一级上调整系统的选项包括：如果磁盘 I/O 是瓶颈，则增加磁盘或使用 RAID 系统；如果磁盘缓冲容量是瓶颈，则增加更多内存；如果 CPU 使用是瓶颈，则改用更快的处理器。

第二个级别由数据库系统参数组成，比如缓冲区大小和检查点间隔。可调的数据库系统参数的集合取决于特定的数据库系统。大多数数据库系统手册提供了有关哪些数据库系统参数可调以及如何选择参数值这些方面的信息。设计良好的数据库系统自动进行尽可能多的调整，以减轻用户或数据库管理员的负担。例如，很多数据库系统的缓冲区大小是固定的，但是是可调的。如果系统能够根据诸如页错误率这样的指标来自动调整缓冲区大小，那么数据库管理员就不必为调整缓冲区大小而烦恼。

第三级别是最高级别。它包括模式和事务。管理员可以调整模式的设计、创建的索引以及执行的事务来提高性能。这一级的调整与系统相对独立。

这三级的调整相互影响。当对系统进行调整时，我们必须将三者结合起来考虑。例如，在某个

更高级别上所做的调整可能导致硬件瓶颈从磁盘系统上移到 CPU 上，反之亦然。

24.1.5 硬件调整

1034 即使在设计良好的事务处理系统中，如果事务所需的数据在磁盘上，那么每个事务通常至少需
1035 要几次 I/O 操作。调整事务处理系统的一个重要因素就是确保磁盘子系统能够处理 I/O 操作所需要的速率。例如，考虑一个访问时间大约是 10 毫秒的磁盘，其平均传输率是每秒 25MB 到 100MB(当今相当典型的磁盘)，这样的磁盘为每次 4KB 的 I/O 操作提供每秒略少于 100 次的随机访问。如果每个事务只需 2 次 I/O 操作，单个磁盘每秒最多可支持 50 个事务。要想每秒支持更多的事务，唯一的方法是增加磁盘的数量。如果系统每秒需支持 n 个事务，每个事务执行 2 次 I/O 操作，那么数据必须被分(或划分)在至少 $n/50$ 个磁盘上(忽略偏斜的影响)。

在此必须注意的是，限制因素并不是磁盘容量，而是能够随机进行数据访问的速度(相应地取决于磁盘臂的移动速度)。将更多的数据存到内存中，可减少每个事务进行 I/O 操作的次数。如果所有数据都放在内存中，除了写操作以外将不会有磁盘 I/O 操作。把经常使用的数据放在内存中以减少磁盘 I/O 次数而增加的额外的内存开销是值得的。把不经常使用的数据放在内存中是一种浪费，因为内存比磁盘要昂贵得多。

问题是，如果用给定数量的钱购买磁盘或内存，如何分配这笔钱才能获得每秒最大的事务量呢？每秒减少一次 I/O 操作将节省：

(每个磁盘驱动器的价格)/(每个磁盘每秒的访问量)

因此，如果一个特定的页每秒被访问 n 次，由于把该页放在内存中而带来的节省将是上面值的 n 倍。在内存中存储一页的成本是：

(每 MB 内存的价格)/(每 MB 内存的页数)

因此，收支平衡点的计算如下：

$$n * \frac{\text{每个磁盘驱动器的价格}}{\text{每个磁盘每秒的存取量}} = \frac{\text{每 MB 内存的价格}}{\text{每 MB 内存的页数}}$$

我们可将上面的公式变形，并用当前的值替换上面公式中的每个参数，就会得到 n 的值；如果某一页的访问频率大于该值，就值得买足够的内存来存储它。对于随机访问的页来说，基于当前的磁盘技术，内存和磁盘的价格(我们假设每个磁盘大约 50 美元，即每 MB0.020 美元)，可得到 n 的值是大约每秒 1/6400 次(或相当于接近两小时一次)。基于几年前的磁盘和内存的成本和速度，相应的值为 5 分钟一次。

这个推理可由最初被称为 **5 分钟规则**(5-minute rule)的经验规则来描述：如果一页在 5 分钟里被使用
1036 的次数多于一次，它就应当存储在内存中。换句话说，几年前此规则建议购买足够的内存来存储所有那些平均 5 分钟至少被访问一次的页。今天，值得买足够的内存来存储所有那些平均 2 小时至少被访问一次的页。对于访问不那么频繁的数据，应购买足够的磁盘以支持这些数据所需的 I/O 访问率。

对于顺序访问的数据，每秒可能会读取非常多的页。假设一次读取 1MB 的数据，几年前我们有 **1 分钟规则**(1-minute rule)，就是说，如果顺序访问的数据一分钟内至少访问一次，就应该把它们放到内存中。根据我们之前的例子，用现在的内存和磁盘成本，相应的数字是 30 秒左右。令人惊讶的是，这个数字多年来并没有很大变化，因为磁盘传输率已经大大提高了，尽管一兆内存的价格相对于磁盘价格已经大大降低了。

显然，每次 I/O 操作读取的数据量对上述时间的影响很大；实际上，如果每次 I/O 操作读或写大约 100KB 的数据，5 分钟规则仍然适用。

5 分钟经验规则及其变体只考虑了 I/O 操作的次数，并没有考虑诸如响应时间这样的因素。有些应用甚至将不常使用的数据也保存在内存中，以提供低于或与磁盘访问时间相当的响应时间。

随着闪存以及基于闪存的"固态硬盘"的普及，系统设计者现在可以把经常使用的数据存储在闪存存储器上，而不是磁盘上。另外，在**闪存作为缓冲区**(flash-as-buffer)的方式中，闪存存储器被用作持久缓冲区，每一块在磁盘上都有永久的位置，但是只要它被经常使用就被存储在闪存中，而不是写入磁盘。当闪存存储器满时，不经常使用的块就被收回，而且如果它从磁盘读出后被更新过，就把它写

回到磁盘。

闪存作为缓冲区方式需要改变数据库系统本身。即使数据库系统不支持闪存作为缓冲区，数据库管理员也可以控制关系或索引到磁盘的映射，并给经常使用的关系/索引分配闪存存储。大多数数据库系统支持的表空间功能可用作映射控制，即通过在闪存存储器上创建表空间，并把所期望的关系和索引分配到该表空间。然而，在比关系更细的粒度级别上控制映射，需要更改数据库系统的代码。

除了主存和硬盘，"5 分钟"规则已经扩展到数据可以存储到闪存的情况下。更多信息请参阅文献注释中所给出的参考文献。

调整的另一个方面是选择 RAID 1 还是 RAID 5 的问题。答案取决于数据更新的频繁程度。因为 RAID 5 比 RAID 1 在随机写上要慢得多；RAID 5 执行单个随机的写请求需要两次读和两次写。如果一个应用程序为支持特定的吞吐率每秒要执行 r 次随机读和 w 次随机写，用 RAID 5 实现，每秒需要 $r+4w$ 次 I/O 操作，而用 RAID 1 实现，每秒需 $r+2w$ 次 I/O 操作。将这个结果除以每秒 100 次 I/O 操作 1037
（当代磁盘的 I/O 速度），我们可以算出为支持每秒所需的 I/O 操作数所需的磁盘数量。对于许多应用来说，r 和 w 是很大的，$(r+w)/100$ 个磁盘足以容纳全部数据的两个副本。对于这些应用，如果使用 RAID 1，所需的磁盘数实际上要少于使用 RAID 5 所需的磁盘数！只有在数据存储量要求很大，而更新率（尤其是随机更新率）较小时，或者说对于非常大而且非常"冷"的数据，RAID 5 才是有效的。

24.1.6　模式调整

在所采用的范式的约束下，可以对关系进行垂直划分。例如，考虑 *course* 关系，其模式为

$$course(course\_id,\ title,\ dept\_name,\ credits)$$

其中 *course_id* 是码。在所采用范式（BCNF 和第三范式）的约束下，我们可以将 *course* 关系划分为两个关系：

$$course\_credit(\underline{course\_id},\ credits)$$
$$course\_title\_dept(\underline{course\_id},\ title,\ dept\_name)$$

由于 *course_id* 是码，这两种表示在逻辑上等价，但是它们的性能特征不同。

如果大多数对课程信息的访问只是查找 *course_id* 和 *credits*，那么这些访问可以在 *course_credit* 关系上运行，由于不读取 *title* 和 *dept_name* 属性，这些访问可能在一定程度上加快。由于相同的原因，缓冲区中能够放下的 *course_credit* 元组比相应的 *course* 元组多，这也可以提高性能。如果 *title* 和 *dept_name* 属性很大，那么效果尤为明显。因此，在这种情况下由 *course_credit* 和 *course_title_dept* 构成的模式比由 *course* 关系构成的模式更为优越。

另一方面，如果大多数对课程信息的访问同时需要 *dept_name* 和 *credits*，那么使用 *course* 关系会更好，因为这样可以避免 *course_credit* 和 *course_title_dept* 的连接开销。另外，存储开销也会更小，因为这时只有一个关系，而且属性 *course_id* 不会重复。

列存储（column store）方式存储数据是基于垂直划分的，但是通过把关系的每个关系属性（列）都存储在一个单独的文件中使垂直划分达到了极限。好几个数据仓库应用已经展示了列存储的良好性能。

提高性能的另一个技巧是存储解除规范化的关系（denormalized relation），例如 *instructor* 和 *department* 的连接，其中关于 *dept_name*、*building* 和 *budget* 信息会为每位教师重复一次。每当更新发生时，维护关系的一致性需要付出更大的代价。但是，读取教师姓名及其所在建筑的查询速度将会提高，因为 *instructor* 和 *department* 的连接已经预先计算好了。如果这样的查询执行得很频繁，并且需要尽可能高效地执行，那么使用解除规范化的关系不无裨益。 1038

物化视图能够提供解除规范化关系所带来的好处，但是需要额外的存储代价。我们将在 24.1.8 节中介绍物化视图的性能调整。物化视图与解除规范化关系相比，一个主要的优点是维护冗余数据的一致性成为数据库管理系统而不是程序员的任务。因此，只要数据库系统支持，物化视图是更可取的。

为一种不使用物化视图来提高连接计算速度的方法是将连接中匹配的记录聚集到相同的磁盘页面上。我们已经在 10.6.2 节中看到过这样的聚集文件组织。

24.1.7　索引调整

我们可以调整数据库系统中的索引来提高性能。如果查询是瓶颈，那么我们常常可以通过在关系

上创建适当的索引来加快查询。如果更新是瓶颈，那么可能是索引太多，这些索引在关系被更新时也必须被更新。删除索引可以加快某些更新。

索引类型的选择也很重要。一些数据库系统支持不同类型的索引，例如散列索引和B树索引。如果经常使用范围查询，那么B树索引比散列索引更适合。是否使索引成为聚集索引是另一个可调参数。一个关系上只允许一个聚集索引，聚集索引按照索引属性顺序来存储关系。通常应该将有利于最大数量的查询和更新的索引设为聚集索引。

为了有助于确定建立何种索引，以及确定每个关系上的哪个索引（如果有多个的话）应该是聚集索引，大多数商用数据库系统提供了调参向导（tuning wizard），对此将在24.1.9节中进行详细介绍。这些工具使用查询和更新（称为工作负载）的历史信息来估算不同索引对工作负载中查询和更新的执行时间所产生的影响，并根据这些估算值来建议创建何种索引。

24.1.8 使用物化视图

维护物化视图可以在很大程度上加快某类查询的速度，特别是聚集查询。回顾13.5节的例子，需要频繁地查询每个系的工资总额（通过累加该系每位教师的工资数额得到）。在那一节里我们看到，创建一个物化视图为每个系存储其工资总额，可以极大地加快此类查询的速度。

但是，物化视图必须小心使用，因为不仅需要存储物化视图的空间开销，而且更重要的是，还需要有维护物化视图的时间开销。在**立即视图维护**（immediate view maintenance）的情况下，如果一个事务的更新操作影响到物化视图，那么这个物化视图必须作为该事务的一部分进行更新，因此事务的运行会变慢。在**延迟视图维护**（deferred view maintenance）的情况下，物化视图会在以后更新。在物化视图更
1039 新前，物化视图可能会与数据库关系不一致。例如，物化视图可以在查询使用到该物化视图时再进行更新，或者进行周期性的更新。使用延迟的视图维护减轻了更新事务的负担。

数据库管理员负责物化视图的选择和视图维护策略。数据库管理员可以通过检查工作负载中的查询类型来手动地做出选择，找出哪些查询需要执行得更快一些，哪些查询或更新可以执行得更慢一些。通过检查，数据库管理员可以选择一个适当的物化视图集合。例如，管理员可能发现某种聚集操作使用频繁，从而选择将它物化；或是发现某个特定的连接运算使用频繁，因此决定将它物化。

但是，即使对于中等规模的查询类型集来说，手工选择物化视图仍然是很费劲的，并且要做出一种较好的选择也是很困难的，因为这需要了解不同方案的代价。只有查询优化器能够在没有实际执行查询的情况下估算出较为精确的代价值。因此一个好的物化视图集只有通过试验和错误才能得到，也就是说，物化一个或多个视图，运行工作负载，测量在该工作负载中执行查询所需的时间。管理员重复这个过程，直到得到一个能够给出可接受性能的物化视图集为止。

一种更好的选择是数据库系统本身提供选择物化视图的机制，并与查询优化器集成在一起。我们将在24.1.9节中详细介绍这种方法。

24.1.9 物理设计的自动调整

大多数商用数据库系统现在都提供工具，帮助数据库管理员进行索引和物化视图的选择，以及其他与物理数据库设计相关的任务，比如如何在并行数据库系统中对数据进行分区。

这些工具检查工作负载（workload）（查询和更新的历史信息），建议需要创建的索引和物化的视图。数据库管理员可以指定加快不同查询的重要程度，当这些工具选择要进行物化的视图时会把它们考虑在内。调整必须在应用程序完全开发之前进行。在开发数据库上的实际数据库内容可能比较少，而在产品数据库上的实际数据库内容则要多得多。因此，一些调整工具还允许数据库管理员指定期望的数据库大小和相关的统计信息。

例如，微软的数据库调整助手允许用户问"what if"这种问题，用户可以挑选一个视图，然后查询优化器就会估算物化该视图对工作负载整体代价的影响，以及对工作负载中的不同类型的查询或更新的个别代价的影响。

索引和物化视图的自动选择通常按照这种方式实现：列举出不同的方案，通过使用工作负载，查
1040 询优化器估算选择每种方案的代价以及所获得的好处。因为设计方案的数目可能相当大，工作负载也

很大，必须谨慎设计选择技术。

第一步是生成工作负载。通常的做法是记录一段时间内执行的所有查询和更新。然后，选择工具执行**工作负载压缩**(workload compression)，即用少量更新和查询生成工作负载的代表。例如，对同一表格的多次更新可以用单次更新和相应于更新发生次数的权重来代表。对同一表格的多次查询可以类似地由带适当权重的代表所替代。这样，执行不频繁且代价不高的查询可能会被丢弃到考虑范围之外，而代价最高的查询可能被最先提交。这种工作负载压缩对于大工作负载是非常必要的。

在查询优化器的帮助下，工具可以产生一组索引和物化视图，加快在压缩后的工作负载中查询和更新的执行。从这些索引和物化视图的不同组合中可以试图找到最好的组合。然而，详尽穷举的方法是不实际的，因为潜在的索引和物化视图的数量已经很大了。它们的每个子集又都是一个潜在的设计方案，这导致了指数级的方案数目。启发式方法可用于降低方案空间，即减少考虑的组合数目。

用于索引和物化视图选择的贪心启发式方法操作如下：估算物化不同索引或者视图所带来的好处(使用查询优化器的代价估算函数作为子程序)，然后选择那些具有最大收益，或是在单位存储空间上具有最大收益(即收益值除以存储该索引或者视图所需要的空间)的索引或视图。在计算收益时，索引或者视图的维护代价必须考虑在内。一旦该启发式方法选定了一个索引或者视图，其他索引或者视图的收益值可能会发生变化，因此该启发式方法需要重新计算它们，来选择下一个最佳索引或者物化视图。重复执行此过程，直到存储索引或物化视图的可用磁盘空间用完，或者剩余的候选索引、视图的维护代价已经超过使用它们给查询带来的收益。

现实世界的索引和物化视图选择工具通常结合一些贪心选择的元素，但使用其他技术得到更好的结果。它们还支持物理数据库设计的其他方面，例如如何对并行数据库中的关系进行分区，或者对一个关系使用何种物理存储机制。

24.1.10 并发事务调整

不同类型事务的并发执行有时会由于锁的竞争而导致性能低下。我们首先考虑更常见的读写竞争的情况，然后再考虑写写竞争的情况。

作为一个**读写竞争**(read-write contention)的例子，考虑银行数据库的如下情形。白天，大量小的更新事务几乎是连续不断地执行。假设同时还运行一个大的查询来计算各个部门的统计数据。如果该查询在一个关系上进行扫描，在它运行时就可能阻塞所有对该关系的更新，这对系统性能来说将是一个 1041
灾难性的影响。

一些数据库系统，例如 Oracle、PostgreSQL 和 Microsoft SQL Server，支持快照隔离，此时查询在数据的某个快照上执行，这样更新就可以继续并发执行了(快照隔离的细节在 15.7 节中介绍)。如果可用，对于大的查询应该使用快照隔离，以避免上述情形中的锁竞争。在 SQL Server 中，在事务开始时执行下列语句

```
set transaction isolation level snapshot
```

会导致对那个事务使用快照隔离。在 Oracle 和 PostgreSQL 中，在上述命令中用关键字 **serializable** 替换关键字 **snapshot** 具有相同的效果，因为这些系统在把隔离性级别设置为可串行化时，实际上都使用快照隔离。

如果快照隔离不可用，一种替代的选择是，在更新很少或是不存在更新时再执行大的查询。然而，对于支持 Web 站点的数据库，也许根本不存在这种更新很少的时候。

另外一种替代方法是使用更弱的一致性级别，例如**已提交读**(read committed)隔离性级别。这时查询的执行对并发更新的影响最小，但不能保证查询结果的一致性。应用语义决定了是否可以接受这种近似的(不一致的)答案。

我们现在考虑**写写竞争**(write-write contention)的情况。在锁机制下更新非常频繁的数据项会导致很差的性能，因为许多事务会等待这些数据项上的锁。这些数据项被称为**更新热点**(update hot spot)。即使在快照隔离下，更新热点也会带来问题，由于写验证失败导致频繁的事务中止。更新热点导致的一种经常发生的情况如下：事务需要给待插入到数据库中的数据项分配唯一标识符，为了这么做它们读取并递增存储在数据库某个元组中的序号计数器的值。如果插入频繁，并且序号计数器以两阶段的

方式被封锁，则包含序号计数器的元组成为一个热点。

一种提高并发度的方法是在序号计数器被读取并递增后立即释放其上的锁。但这样做之后，即使事务中止，对序号计数器的更新都不应该回滚。为了理解其中的原因，假设 T_1 递增序列计数器，然后在 T_1 提交前 T_2 也递增序号计数器；如果 T_1 中止，它的更新回滚，无论是把计数器恢复到原来的值，还是递减计数器，都将导致 T_2 使用的序号值被后继事务重用。

1042 大多数数据库提供了一个特殊的结构来创建**序号计数器**(sequence counter)，它实现早期的、非两阶段的、锁释放的、伴随特殊情况处理的 undo 日志，使得事务中止时对计数器的更新不会被回滚。SQL 标准允许使用下列命令来创建序号计数器：

create sequence *counter1* ;

在上述的命令中，*counter1* 是序号的名称，可以用不同的名称创建多个序号。得到一个序号值的语法非标准化的。在 Oracle 中，*counter1.nextval* 在递增序号值后返回序号的下一个值。在 PostgreSQL 中调用函数 *nextval*('*counter1*') 有相同的效果，而 DB2 使用的语法为 **nextval for** *counter1*。

SQL 标准提供的另一个方案是使用显式的序号计数器，当目标是为插入到关系中的元组提供唯一标识符时这是非常有用的。为此，可以在关系的一个整数属性(通常此属性也被声明为主码)的声明上添加关键字 **identity**。如果在对该关系进行插入的语句中，没有指定此属性的值，对于每个新插入的元组会自动创建一个唯一的新值。在内部使用非两阶段封锁的序号计数器来实现 **identity** 声明，当每次插入一个元组时就递增计数器。虽然语法不同，包括 DB2 和 SQL Server 在内的多个数据库都支持 **identity** 声明。PostgreSQL 支持一种称作 **serial** 的数据类型，它提供了相同的效果：PostgreSQL 通过透明地创建非两阶段封锁的序号来实现 **serial** 类型。

值得注意的是，如果事务中止，事务对一个序号的获得也不能被回滚(由于前面讨论的原因)。这样事务中止就可能导致插入到数据库中的元组出现序号空缺。例如，如果获得序号为 1002 的事务没有提交，可能会有标识符值为 1001 和 1003 的元组，但没有值为 1002 的元组。这种空缺在某些应用中是不允许的。例如，一些金融应用要求账单或收据号不能有空缺。数据库提供的序号和自动递增的属性不能用于此类应用，因为它们会导致空缺。以两阶段方式封锁的、存储在正常元组中的序号计数器不会产生这种空缺，因为事务中止会恢复序号计数器的值，而下一个事务将得到相同的序号，从而避免了空缺。

长的更新事务会带来系统日志的性能问题，并增加系统崩溃后的恢复时间。如果一个事务执行了很多更新，系统日志甚至会在事务完成前就满了，这时事务将不得不被回滚。如果一个更新事务运行很长时间(尽管只有少量更新)，日志系统设计得又不够好，那么它可能会阻塞对日志中旧的部分的删除。这样的阻塞也可能导致日志满。

1043 为了避免这样的问题，很多数据库系统对单个事务所能进行的更新数目进行了严格限制。即使系统不做这样的限制，将大的更新事务尽可能地分成一组较小的更新事务通常也是有帮助的。例如，给大公司的每个员工加薪的事务可以拆分为一系列小事务，每个小事务对一个小范围内的员工进行更新。这种事务称作**小型批处理事务**(mini-batch transaction)。但是，使用小型批处理事务必须小心。首先，如果在员工集合上存在并发更新，小事务集合的结果可能并不等价于执行单个大事务的结果。其次，如果发生故障，就可能有一些员工的工资由于事务的提交得到增长，而其他员工的工资却没有增加。为了避免这种情况，一旦系统从故障中恢复，我们应立即执行该批处理中剩余的事务。

不管是只读的还是更新的长事务，还可能导致锁表变满。如果一个查询扫描一个大型关系，查询优化器将确保它得到关系锁而不是获得大量的元组锁。但是，如果一个事务执行大量的小型查询或更新，它可能会获得大量的锁，导致锁表变满。

为了避免这个问题，一些数据库提供了自动的**锁升级**(lock escalation)。利用这个技术，如果一个事务已经获得了大量的元组锁，元组锁就被升级为页锁，甚至整个关系上的锁。回顾多粒度封锁机制(15.3 节)，一旦得到较粗粒度级别的锁，就没有必要再记录较细粒度级别的锁。因此可以从锁表中删除元组锁项，释放空间。在不支持锁升级的数据库上，事务有可能显式地获得关系锁，从而避免对元组锁的获取。

24.1.11 性能模拟

为了在数据库系统安装前测试其性能，我们可以构造数据库系统的性能模拟模型。图24-2中所示的每个服务，如CPU、每个磁盘、缓冲区和并发控制，在模型中都将被模拟。模拟模型并不模拟服务的细节，而只是模仿每个服务的某些方面，例如服务时间(service time)，即一旦请求处理开始直至结束所需要的时间。因此，模型可以仅仅基于磁盘平均访问时间来模拟磁盘访问。

由于服务请求通常需要排队等待，在模拟模型中每个服务有一个与之关联的队列。一个事务由一系列请求构成。请求到达后就排到队列中去等待，并根据服务策略(如先来先服务)获得服务。不同服务(如CPU和磁盘)的模型在概念上是并行操作的，以表示实际系统中这些子系统并行操作的事实。

一旦建立起事务处理的模拟模型，系统管理员就可以在其上进行许多实验。用到达率不同的模拟事务做实验，管理员可以得到系统在不同负载情况下的表现情况。管理员还可以通过改变每种服务的服务时间来做实验，以发现性能对每种服务的敏感程度。还可以改变系统参数，这样就可以在模拟模型上进行性能调校。 1044

24.2 性能基准程序

由于数据库服务器变得越来越标准化，产品性能成为不同厂商产品的主要差异因素。**性能基准程序**(performance benchmark)是一套用于量化软件系统性能的任务。

24.2.1 任务集

由于大多数软件系统(如数据库)都很复杂，不同厂商的实现中会有许多不同。因此，针对不同任务它们在性能上存在显著差异。一个系统在某个特定任务上可能是最有效的；而另一系统可能在另一个不同的任务上最有效。因此，仅用一个任务来量化系统性能通常是不够的，而要用一个称作该能基准程序的标准化任务集合来度量系统性能。

将来自多个任务的性能值结合起来的工作必须小心进行。假设我们有两个任务T_1和T_2，并且我们用给定时间(如1秒)内所运行的每类事务的数量来作为系统吞吐量的度量。假设系统A运行T_1时是每秒99个事务，运行T_2时是每秒1个事务。类似地，设系统B每秒运行T_1和T_2时都是50个事务。还假设工作负载是两种类型事务的一个等比例混合。

如果我们取这两对数(即99和1与50和50)的均值，看起来似乎这两个系统的性能一样。但是，以这种形式取均值是错误的——如果每种类型运行50个事务，系统A要用50.5秒才能完成，而系统B只要2秒就可以完成！

上例表明，如果有不止一种类型的事务，对性能进行简单的度量会导致错误。综合性能值的正确方法是采用工作负载的**完成时间**(time to completion)，而不是采用每类事务吞吐量的平均值。这样，对于具体工作负载，我们可以用每秒事务数准确地计算系统性能。因此，系统A执行每个事务平均需要50.5/100即0.505秒，而系统B执行每个事务平均需要0.02秒。用吞吐量来描述，系统A平均每秒执行1.98个事务，而系统B平均每秒执行50个事务。假设每类事务发生的可能性相等，则综合不同事务类型吞吐量的正确方法是求这些吞吐量的**调和平均数**(harmonic mean)。n个吞吐量t_1，…，t_n的调和平均数定义为： 1045

$$\frac{n}{\left(\frac{1}{t_1}+\frac{1}{t_2}+\cdots+\frac{1}{t_n}\right)}$$

对我们的例子来说，系统A吞吐量的调和平均数为1.98，而系统B吞吐量的调和平均数为50。因此，对于由这两种示例类型事务等量混合而成的工作负载而言，系统B大约比系统A快25倍。

24.2.2 数据库应用类型

联机事务处理(OnLine Transaction Processing，OLTP)和**决策支持**(decision support)(包括**联机分析处理**(OnLine Analytical Processing，OLAP))是数据库系统所处理的两大类应用。这两类任务具有不同的需求。一方面，支持频繁的更新事务需要有高的并发度和能加速事务提交处理的好技术。另一方面，

决策支持又需要有好的查询执行算法和查询优化。一些数据库系统的体系结构被调整为适用于事务处理；而另一些数据库系统(如 Teradata 并行数据库系统系列)的体系结构被调整为适用于决策支持。另外还有一些厂商力争在这两类任务间取得平衡。

通常应用中混合着事务处理和决策支持的需求。因此，对一个应用来说，哪个数据库系统最好取决于该应用中这两类需求的混合比例。

假设我们分别有这两类应用的吞吐量数值，并且当前的应用是这两类事务的混合。由于事务间的**干扰**(interference)，即使求吞吐量数值的调和平均数时我们也必须小心。例如，一个长的决策支持事务可能获得很多锁，这可能阻碍所有更新事务的进行。吞吐量的调和平均数只能在事务互不干扰时使用。

24.2.3 TPC 基准程序

事务处理性能委员会(Transaction Processing Performance Council，TPC)定义了一系列数据库系统基准程序的标准。

TPC 基准程序定义得非常详细。它定义了关系的集合和元组的大小。它并没有把关系中的元组数定义为一个固定的数值，而是定义为随着称为每秒事务数目的倍数，以反映更高的事务执行速度可能会与更多的账户数目相联系。用来度量性能的尺度是吞吐量，用**每秒事务数**(Transaction Per Second，
[1046] TPS)表示。在测量性能时，系统必须提供在一定范围内的响应时间，这样，获得高吞吐量就不能以很长的响应时间为代价。另外，对商业应用来说，代价是非常重要的。因此，TPC 基准程序还用**每 TPS 的代价值**(price per TPS)来度量性能。一个大系统可能每秒有很高的事务数，但可能代价很高(即每 TPS 的代价值很高)。此外，公司宣称其系统的 TPC 基准程序测试值时，必须有外部的审查，以确保系统如实地遵循了基准程序的定义，包括对事务 ACID 特性的完全支持。

该系列中的第一个标准是 **TPC-A 基准程序**(TPC-A benchmark)，它定义于 1989 年。此基准程序通过单一类型的事务来模拟典型的银行应用，该事务模拟银行出纳员办理的取款和存款业务。它至新几个关系(例如银行余额、出纳员余额和客户余额)，并且在一个审计跟踪关系中增加一条记录。该基准程序还和终端进行通信，以模拟实际系统的端到端性能。**TPC-B 基准程序**(TPC-B benchmark)是为了测试数据库系统及运行它的操作系统的核心性能而设计。它去除了 TPC-A 基准程序中处理用户、通信和终端的部分，集中测试后端的数据库服务器。现在 TPC-A 和 TPC-B 都没有使用了。

TPC-C 基准程序(TPC-C benchmark)是为了模拟比 TPC-A 基准程序更复杂的系统而设计的。TPC-C 基准程序主要考虑一个订购于续环境中的主要活动。例如输入和传送订单、记录付款、检查订单状态和监控库存量。TPC-C 基准程序仍被广泛应用于联机事务处理(OLTP)系统中。更近期的 TPC-E 基准程序的目标也是 OLTP 系统，但是它是基于一家经纪公司的模型，该公司的客户与公司交互并产生事务，该公司随之与金融市场交互以执行事务。

TPC-D 基准程序(TPC-D benchmark)是为了测试数据库系统在决策支持查询中的性能而设计的。决策支持系统现在正变得越来越重要。TPC-A，TPC-B 和 TPC-C 基准程序刻画事务处理工作负载下的性能，不应被用来测量决策支持查询中的性能。TPC-D 中的"D"代表**决策支持**(decision support)。TPC-D基准程序的模式是模拟一个销售/分发应用，包括零件、供应商、客户、订单和一些辅助信息。关系的大小定义为一个比率，数据库大小为全部关系元小之和，以十亿字节(GB)为单位来描述。TPC-D 等级因子为 1 表示 TPC-D 基准程序运行于 1GB 的数据库上，等级因子为 10 表示 TPC-D 基准程序运行于 10GB 的数据库上。此基准程序的工作负载包含一个由 17 个 SQL 查询构成的集合，以模拟在决策支持系统上执行的通用任务。其中一些查询使用了复杂的 SQL 特性，例如聚集和嵌套查询。

基准程序使用者很快意识到各种 TPC-D 的查询可以通过使用物化视图或其他冗余信息来显著提高
[1047] 速度。有些应用，如阶段性的报告工作中，查询可以预先确定并可以仔细选取物化视图以加速查询。然而，有必要考虑维护物化视图的开销。

TPC-H 基准程序(TPC-H benchmark)(这里的"H"表示**即席**(ad hoc))是 TPC-D 基准程序的缩化。它的模式与 TPC-D 相同，但有 22 个查询。其中 16 个来自 TPC-D。此外，还有两个更新，一组插入和一组删除。TPC-H 禁止使用物化视图和其他冗余信息，并且只允许在主码和外码上创建索引。这个基

准程序模拟即席查询。这种查询不能提前预知，所以不可能提前创建恰当的物化视图。一个已经不再被使用的变体 TPC-R（这里的"R"代表报告（reporting））允许使用物化视图和其他冗余信息。

TPC-H 用如下方式度量性能：**能力测试**（power test）每次按时间顺序执行查询和更新，3600 秒除以所有查询执行时间（以秒为单位）的几何平均数，就得到每小时的查询数量。**吞吐量测试**（throughput test）并行执行多个工作流，每个工作流执行全部 22 个查询，另外还有一个并行更新工作流。用整体的运行时间计算每小时的查询数量。

每小时复合查询度量（composite query per hour metric）是一个全面的度量，用能力度量和吞吐量度量乘积的平方根表示。**复合代价/性能度量**（composite price/performance metric）由系统代价除以复合代价得到。

TPC-W Web 商务基准程序（TPC-W Web commerce benchmark）是一个端对端的基准程序，模拟具有静态内容（主要是图像）和从数据库中生成的动态内容的 Web 站点。它明确指出允许高速缓存动态内容，因为这对于加快 Web 站点的速度非常有用。此基准程序模拟了一个电子书店。与其他 TPC 基准程序类似，提供了不同的等级因子。主要性能度量是**每秒 Web 交互数**（Web Interaction Per Second，WIPS）和每 WIPS 的代价值。但是，TPC-W 基准程序已经不再被使用了。

24.3 应用系统开发的其他问题

在这一节中，我们讨论应用系统开发中的两个问题：应用系统测试和应用系统移植。

24.3.1 应用系统测试

程序测试包括设计一个**测试集**（test suite），即一个测试用例的集合。测试不是一次性的过程，因为程序在不断演化。程序的更改可能导致意想不到的错误出现。这样的错误称为程序回归（regression）。因此，程序每次改变后，都必须重新进行测试。程序每次改变后让人来运行测试通常是不可行的。相反，预期的测试输出和每个测试用例都一起存储在测试集里。**回归测试**（regression testing）包括把程序运行在测试集中的每个测试用例上，并检查程序是否产生预期的测试输出。 10.4.8

在数据库应用中，一个测试用例由两个部分组成：数据库状态，以及一个应用程序特定接口的输入。

SQL 查询可能有难以捕获的细微错误。例如，当一个查询实际上应该执行 $r \times s$ 时，它却可能执行 $r \bowtie s$。只有当测试数据库有一个 r 元组，它没有匹配的 s 元组时这两个查询间的差别才能被发现。因此，创建可以捕获经常发生的错误的测试数据库是很重要的。这样的错误称为**突变**（mutation）。因为它们通常是查询（或程序）的小变化。在一个预期查询和该查询的突变上产生不同输出的测试用例，称为**消灭突变**（kill the mutant）。一个测试集应该有能消灭（大多数）经常发生的突变的测试用例。

如果一个测试用例执行对数据库的更新，为了检查它是否正确执行，必须验证数据库的内容与预期的内容相匹配。因此，预期的输出不仅包括显示在用户屏幕上的数据，而且包括数据库状态（的更新）。

由于数据库状态可能相当大，多个测试用例可能共享同一个数据库状态。如下事实使测试变得复杂：如果一个测试用例执行对数据库的更新，随后运行在同一个数据库上的其他测试用例的结果可能与预期结果不匹配。然后其他测试用例就会被错误地报告为失败。为了避免这个问题，每当一个测试用例执行更新，在测试运行后数据库状态必须恢复到其原始状态。

测试还可以用来确保应用程序符合性能要求。为了进行这种**性能测试**（performance testing），测试数据库必须和真实数据库一样大。在某些情况下，已经存在可以进行性能测试的数据。在其他情况下，必须生成一个所需大小的测试数据库。有几个工具可用于生成这种测试数据库。这些工具确保生成的数据满足约束，比如主码约束和外码约束。它们可能额外生成看起来有意义的数据，例如，通过使用有意义的姓名填充姓名属性，而不是使用随机字符串。有些工具还允许指定数据分布，例如，一个大学数据库可能要求这样一个分布：大多数学生在 18 岁至 25 岁之间，并且大部分教师在 25 岁至 65 岁之间。

即使有一个现有的数据库，机构通常也不希望把敏感数据泄露给可能进行性能测试的外部机构。

在这种情况下，可以复制真实数据库的一个副本，并以这样一种方式修改副本中的值：使得任何敏感数据，如信用卡号、社会保险号或出生日期，被**模糊**(obfuscated)处理。大多数情况下通过用随机生成的值替换真实值来完成模糊处理(注意如果该值是主码，还要更新所有对该值的引用)。另一方面，如果应用程序的执行依赖于值，比如在一个基于出生日期执行不同动作的应用中的出生日期，模糊处理
[1049] 可以对值进行小的随机变化，而不是完全替换它。

24.3.2 应用系统移植

遗产系统(legacy system)是老一代应用系统，一个机构正在使用它，但该机构又想用另外的应用系统替换它。例如，许多机构内部开发了应用系统，而又可能决定用商业产品代替它们。在一些案例中，遗产系统可能使用与当代的标准和系统不兼容的老技术。现在运转着的一些遗产系统有几十年历史，基于诸如网状或层次数据模型的数据库技术，或者使用没有数据库的 Cobol 和文件系统。这些系统可能仍然包含有价值的数据，并可能支持关键性的应用。

用新的应用系统替换遗产应用系统，在时间和金钱方面的代价通常都很大。因为它们一般非常庞大，包括由多组程序员经过几十年开发而成的数百万行代码。它们包含的大量数据必须被导入到新的应用系统中。还可能使用的是完全不同的模式。由老应用系统转换到新应用系统还会涉及大量员工的再培训。在进行转换时通常必须没有中断，从老系统中输入的数据通过新系统也能获得。

许多机构试图避免替换遗产系统，改为设法让它们与新系统交互操作。一种在关系数据库与遗产数据库之间进行交互操作的方法是，在遗产系统上建立一个层，称为**包装层**(wrapper)，它使得遗产系统看上去像个关系数据库。包装层可以提供对 ODBC 或其他互连标准(如 OLE-DB)的支持，用于对遗产系统的查询和更新。包装层负责将关系查询和更新转化为对遗产系统的查询和更新。

当某个机构决定用新系统替换遗产系统时，需要进行一个称作**逆向工程**(reverse engineering)的过程。逆向工程包括检查遗产系统代码，提出所需数据模型(如 E-R 模型或面向对象数据模型)中的设计模式。逆向工程还要检查代码找出实现的程序和过程，以得到系统的高层模型。之所以需要逆向工程是因为遗产系统常常没有编写的高层文档资料和全面的系统设计。当提出新系统的设计时，开发者回顾旧系统的设计，这样就可以进行改进而不只是重新实现。还需要扩充的结构工作来支持遗产系统提供的所有功能(例如用户界面和报表系统)，整个过程叫做**工程再设计**(re-engineering)。

当新系统创建并且测试过之后，系统必须装载遗产系统中的数据。所有进一步的活动都由新系统来开展。但是，这种突然向新系统的转移被称为**大爆炸方法**(big-bang approach)。它有几个风险：首先，用户可能不熟悉新系统界面；其次，新系统中可能存在测试中没有发现的错误或性能问题。这些
[1050] 问题可能导致公司的巨大损失，因为它们执行诸如销售和购买部件的关键事务的能力会受到严重影响。在某些极端情况下，尝试转换失败后，新系统甚至被放弃，而重新使用遗产系统。

另外一种可选的方案叫做**胆小鬼方法**(chicken-little approach)，逐增地替换遗产系统的功能。例如，使用新的用户界面，后端仍然使用遗产系统，反之亦然。另外一种选择是，只对那些能够从遗产系统中分离出来的功能使用新系统。任何一种情况都需要遗产系统和新系统并存一段时间。因此需要在遗产系统上开发和使用包装层，以提供所需的与新系统交互操作的功能。因此，这种方案需要更高的开发费用。

24.4 标准化

标准(standard)定义了软件系统的接口。例如，标准可以定义编程语言的语法和语义、应用程序接口中的函数，甚至数据模型(如面向对象数据库标准)。现在的数据库系统非常复杂，常常由多个独立创建的、需要交互的部件组成。例如，客户端程序的创建可以独立于后端系统，但二者必须能够彼此交互。具有多个异构数据库系统的公司可能需要在数据库之间交换数据。在这种场合下，标准发挥着重要的作用。

正式标准(formal standard)由标准化组织或行业组织通过公开的程序来制定。占统治地位的产品有时会成为**事实标准**(de facto standard)，因为它们没有通过任何正规的公认程序而被作为标准广泛接受。一些正式标准，如 SQL-92 和 SQL: 1999 标准的许多方面，会引导市场的**预见标准**(anticipatory

standard)。它们先定义一些特性，然后厂商才在产品中实现这些特性。而另外的情况下，标准或标准的一部分是反应标准(reactionary standard)，因为它们试图标准化一些已经有厂商实现的甚至已经成为事实标准的特性。在很多方面 SQL-89 是反应性的，因为它对 IBM SAA SQL 标准以及其他数据库中已经出现的某些特征(如完整性检查)进行标准化。

正式标准委员会通常包括厂商代表、用户组成员、来自标准化组织(如国际标准化组织(ISO)和美国国家标准协会(ANSI))的成员，以及来自专业团体(如电子电气工程师协会(IEEE))的成员。正式标准委员会定期集会，成员对标准中的特性提出增加或修改建议。在一段(通常被延长的)时间的讨论、修改建议以及全体审阅后，成员对是否接受或拒绝某一特性进行投票。标准制定和实现后，经过一段时间，其缺点会变得明显，新的需求变得显著。于是开始了标准的更新过程，并且通常几年后会发布 1051
标准的新版本。这样的循环通常每几年重复一次，直到最终(也许是很多年以后)标准在技术上变得无关紧要或失去其用户基础。

由数据库任务组提出的网状数据库标准 DBTG CODASYL 是用于数据库的早期正式标准之一。因为 IBM 占据了数据库市场的很大份额，所以之前 IBM 数据库产品曾建立过事实标准。随着关系数据库的发展，很多新的竞争者加入到数据库产业中，因此产生了对正式标准的需求。最近几年，微软提出了许多规范，这些也已经成为事实标准。著名的例子是 ODBC，如今在非微软环境中也在使用。Sun Microsystems 提出的 JDBC 规范，是另一个被广泛使用的事实标准。

本节从很高的层次对不同标准进行了概述，重点集中在标准的目的上。本章末尾的文献注释给出了关于本节中所提到标准的详细描述的参考文献。

24.4.1　SQL 标准

由于 SQL 是使用最广泛的查询语言，在对其进行标准化上人们已经做过很多工作。ANSI 和 ISO，以及各家数据库厂商在这项工作中起到了主导作用。SQL-86 是最初的版本。IBM 系统应用体系结构(Systems Application Architecture，SAA)的 SQL 标准于 1987 年发布。随着人们对更多特性的需求，正式 SQL 标准的更新版本 SQL-89 和 SQL-92 被制定出来。

SQL：1999 版本的 SQL 标准，给 SQL 增加了许多特性。在前面章节中我们已经见到过许多这样的特征。SQL：2003 版本的 SQL 标准是对 SQL：1999 标准的细微扩充。一些特征，例如 SQL：1999 的 OLAP 特性(见 5.6.3 节)被说明为对 SQL：1999 标准早期版本的改善，并没有等到发布在 SQL：2003 中。

SQL：2003 标准被分为如下几个部分：

- 第 1 部分：SQL/Framework 给出了对标准的概览。
- 第 2 部分：SQL/Foundation 定义了标准的基本元素——类型、模式、表、视图、查询以及更新语句、表达式、安全模型、谓词、赋值规则、事务管理等。
- 第 3 部分：SQL/ CLI(调用层接口)定义了应用程序对 SQL 的接口。
- 第 4 部分：SQL/ PSM(持久存储模块)定义了 SQL 的过程性扩展。
- 第 9 部分：SQL/MED(外部数据管理)定义了 SQL 系统到外部资源的接口标准。通过书写包装层，系统设计者可以将外部数据源如文件或非关系数据库中的数据，当做"外来"表来处理。 1052
- 第 10 部分：SQL/ OLB(对象语言绑定)定义了 Java 中嵌入 SQL 的标准。
- 第 11 部分：SQL/Schemata(信息模式和定义模式)定义了标准目录接口。
- 第 13 部分：SQL/JRT(Java 程序和类型)定义了访问 Java 中程序和类型的标准。
- 第 14 部分：SQL/XML 定义了 XML 相关的规范。

缺失部分包括诸如临时数据、分布式事务处理和多媒体数据这样的特性，对此在标准上还没有达成一致。

SQL 标准的最新版本是 SQL：2006 和 SQL：2008，前者增加了几个与 XML 相关的特性，后者引入了许多对 SQL 语言的扩展。

24.4.2　数据库连接标准

ODBC 标准是广泛应用的客户端应用程序与数据库系统之间进行通信的标准。ODBC 是基于 X/

Open 行业协会和 SQL Access Group 制定的 SQL **调用层接口**(Call Level Interface，CLI)标准，但有所扩展。ODBC API 定义了一个 CLI、一个 SQL 语法定义和关于允许的 CLI 调用序列的规则。该标准还定义了 CLI 和 SQL 语法的一致级别。例如，CLI 核心级包括连接数据库、准备和执行 SQL 语句、取回结果或状态值以及管理事务的命令。下一个一致级别(级别 1)要求支持对目录信息检索以及其他一些核心级 CLI 之上的特性；级别 2 要求进一步的特性，如发送和检索参数值数组以及检索更加详细的目录信息的能力。

ODBC 允许一个客户端同时连接到多个数据源，并在这些数据源之间进行切换，但各个数据源上的事务是独立的；ODBC 不支持两阶段提交。

分布式系统提供比客户－服务器系统更通用的环境。X/Open 协会也为数据库互操作制定了 **X/Open XA 标准**(X/Open XA standard)。这些标准定义了兼容数据库应该提供的事务管理原语(如事务开始、提交、中止和准备提交)。事务管理器可以调用这些原语，利用两阶段提交实现分布式事务。XA 标准独立于数据模型和客户端与数据库之间交换数据的特定接口。因此，我们可以利用 XA 协议实现
1053 分布式事务系统，在这样的系统中，一个事务可以既能访问关系数据库又能访问面向对象的数据库，而事务管理器通过两阶段提交保证全局一致性。

有许多数据源不是关系数据库，事实上有可能根本不是数据库，如平面文件和邮件存储。微软的 **OLE-DB** 是一个 C++ 应用程序接口，其目的与 ODBC 类似，但是对于非数据库数据源，它只提供有限的查询和更新功能。与 ODBC 类似，OLE-DB 提供一些结构，用于连接数据源、开始会话、执行命令、以行集(结果行的集合)的形式取回结果。

但是，OLE-DB 与 ODBC 在许多方面是不同的。为了支持提供有限特征的数据源，OLE-DB 的特征分为许多接口，一个数据源可以只实现接口的一个子集。OLE-DB 程序可以与数据源协商，找到数据源支持的接口。在 ODBC 中，命令都是在 SQL 中。在 OLE-DB 中，命令可以在数据源支持的任何语言中。有些数据源可能支持 SQL，或者 SQL 的一个受限子集，而其他数据源可能只提供简单的功能，如访问平面文件中的数据，却不提供任何查询功能。OLE-DB 与 ODBC 的另一个主要区别在于，OLE-DB 的行集是一个可以通过共享内存被多个应用程序共享的对象。一个行集对象可以被某个应用程序更新，其他共享该对象的应用程序将被告知这个改变。

同样，由微软创建的**活动数据对象**(Active Data Object，ADO)应用程序接口提供了一个易于使用的 OLE-DB 功能接口，可以从脚本语言(如 VBScript 和 JScript)中进行调用。更新的 **ADO. NET** 应用程序接口是为用诸如 C#和 Visual Basic. NET 那样的 .NET 语言编写的应用程序而设计的。除了提供简化的接口，它还提供了称为数据集的抽象概念，允许断开连接的数据访问。

24.4.3 对象数据库标准

面向对象数据库领域中的标准到目前为止主要是由 OODB 厂商驱动的。**对象数据库管理组**(Object Database Management Group，ODMG)是由 OODB 厂商组成的，对 OODB 数据模型和语言接口进行标准化的团体。ODMG 所定义的 C++ 语言接口已在第 22 章中进行了概述。ODMG 已不再活跃。JDO 是为 Java 增加持久性的标准。

对象管理组(Object Management Group，OMG)是由公司组成的一个协会，其目标是制定基于面向对象模型的分布式软件应用的标准体系结构。OMG 提出了对象管理体系结构(OMA)参考模型。对象请求代理(ORB)是 OMA 体系结构中的一个组件，它为分布式对象提供透明的消息分发，因而对象的物理位置无关紧要。**通用对象请求代理体系结构**(Common Object Request Broker Architecture，CORBA)为 ORB 提供了详细的说明，还包括了用于定义数据交换中所采用数据类型的**接口描述语言**(Interface
1054 Description Language，IDL)。当数据在数据表示不同的系统间传递时，IDL 有助于提供数据转换。

微软提出了**实体数据模型**(entity data model)，它结合了实体联系和面向对象数据模型的思想，以及一个称作**语言集成查询**(Language Integrated Querying 或 LINQ)的方法，该方法用于把查询集成到编程语言中。这些很有可能会成为事实标准。

24.4.4 基于 XML 的标准

人们为多种不同的应用定义了大量不同的基于 XML(见第 23 章)的标准。这些标准中有许多都与

电子商务有关。它们包括非盈利团体发布的标准以及企业为建立事实标准而付出的努力。

RosettaNet 属于前者，它是一个工业协会利用基于 XML 的标准来帮助计算机与信息技术产业中的供应链管理。供应链管理指购买机构运行所需要的材料和服务。相反，因客关系管理指公司交互的前端，是与客户打交道的。供应链管理要求很多事情的标准化，例如：

- **全局公司标识**(global company identifier)：RosettaNet 指明唯一识别公司的系统，使用 9 位数字的标识符，称作数据通用编号方式系统(DUNS)。
- **全局产品标识**(global product identifier)：RosettaNet 指明了一个 14 位数字的全局商业项目编号(GTIN)，用于标识产品和服务。
- **全局类别标识**(global class identifier)：这是一个联合国/标准产品和服务码(UN/SPSC)的 10 位数字层次编码，用于对产品和服务进行分类。
- **贸易伙伴之间的接口**(interface between trading partners)：RosettaNet 伙伴接口过程(PIP)定义了伙伴之间的商业过程。PIP 是基于 XML 的系统对系统的会话：它们定义了处理过程所涉及的商业文档格式和语义以及完成事务所采取的步骤。作为例子，这些步骤可能包括获得产品和服务的信息、购货订单、订单货品计价、付款、订单状态请求、存货管理、包括服务担保在内的售后支持等。设计、配置、处理过程和质量信息的交换还有可能协调跨机构的制造活动。

电子市场的参与者可能将数据存储在多种数据库系统中。这些系统可能使用不同的数据模型、数据格式和数据类型。此外，数据间可能存在语义差异(公制对英制，不同流通货币，等等)。电子市场的标准包括使用 XML 模式包装每个这样的异构系统的方法。这些 XML 包装器为分布在所有市场参与 [1055]
方的数据建立了统一视图的基础。

简单对象访问协议(Simple Object Access Protocol，SOAP)是一种远程过程调用标准，它使用 XML 编码数据(参数和结果)，利用 HTTP 作为传输协议。这样，函数调用就成为一个 HTTP 请求。SOAP 由万维网联盟(W3C)支持，并且获得了产业界的广泛支持。SOAP 可用于各种应用。例如，在 B2B 的电子商务中，在某个站点上运行的应用程序可以通过 SOAP 访问其他站点上的数据并执行动作。

在 23.7.3 节已经详细介绍了 SOAP 和 Web 服务。

24.5　总结

- 调整数据库系统参数和更高级别的数据库设计(如模式、索引和事务)对于实现高性能至关重要。查询可以进行调整以提高集合面向性，而批量加载功能可以大大加快数据导入到数据库中的速度。

  要调整的最好办法是确定瓶颈所在，然后消除瓶颈。数据库系统通常有多种可调参数，如缓冲区大小、内存大小和磁盘数量。可以选择适当的索引和物化视图集合，以使总体代价达到最小。可以调整事务使锁竞争达到最小。实现隔离和支持早期锁释放的序号编号功能是减少读写和写写竞争的有用工具。
- 性能基准程序在对数据库系统进行比较方面扮演了重要的角色，尤其在数据库系统变得越来越与标准兼容时。TPC 基准程序是使用广泛、不同的 TPC 基准程序可以用于不同工作负载下的数据库系统性能的比较。
- 应用程序在开发时和部署前需要进行大量的测试。测试能用来捕获错误的，以及确保达到性能目标。
- 遗产系统是基于老一代技术(如非关系数据库或甚至直接基于文件系统)的系统。当运行关键任务系统时，遗产系统与新一代系统之间的连接通道是很重要的。从遗产系统到新一代系统的移植必须非常小心以避免破坏，这种移植是非常昂贵的。
- 由于数据库系统的复杂性和互操作的需要，标准对数据库系统来说很重要。SQL 有其正式标准。事实标准(如 ODBC 和 JDBC)和被行业组织所采纳的标准(如 CORBA)，在客户－服务器数据库系统的发展中发挥了重要作用。 [1056]

术语回顾

- 性能调整
- 面向集合
- 批处理更新(JDBC)

- 批量加载
- 批量更新
- 合并语句
- 瓶颈
- 队列系统
- 可调参数
- 硬件调整
- 五分钟规则
- 一分钟规则
- 模式调整
- 索引调整
- 物化视图
- 立即视图维护
- 延迟视图维护
- 事务调整
- 锁竞争
- 序号
- 小型批处理事务
- 性能模拟
- 性能基准程序
- 服务时间
- 完成时间
- 数据库应用类型
- TPC 基准程序
  - TPC-A
  - TPC-B
  - TPC-C
  - TPC-D
  - TPC-E
  - TPC-H
- 每秒的网络交互数
- 同步测试
- 消灭突变
- 遗产系统
- 逆向工程
- 工程再设计
- 标准化
  - 正式标准
  - 事实标准
  - 预见标准
  - 反应标准
- 数据库连接标准
  - ODBC
  - OLE-DB
  - X/Open XA 标准
- 对象数据库标准
  - ODMG
  - CORBA
- 基于 XML 的标准

实践习题

24.1 很多应用程序需要产生每个事务的序列号。

1057 a. 如果一个序号计数器采用两段封锁协议，可能成为一个并发瓶颈。解释产生这种情况的原因。

b. 很多数据库系统支持内置的序号计数器，采用非两阶段封锁协议：当一个事务请求一个序号时，计数器上锁，累加，然后释锁。

i. 解释这样的计数器如何提高并发度。

ii. 解释在故障提交事务的序号之间存在空缺的原因。

24.2 假设给定一个关系 $r(a, b, c)$。

a. 给出一个例子，说明什么情况下在属性 a 上的等值选择查询的性能会受关系 r 是如何聚集的影响很大。

b. 假设还有在属性 b 上的范围选择查询。r 如何聚集会使 $r.a$ 上的等值选择查询和 $r.b$ 上的范围选择查询都能高效地回答？请解释你的答案。

c. 如果上述聚集方法是不可能的，给出一个建议，如何通过选择合适的索引使两种类型查询都能高效执行。假设你的数据库支持只有索引的计划（即如果一个查询请求的所有信息在一个索引上都可获得，那么数据库能产生一个只使用索引而不需访问关系的执行计划）。

24.3 假设数据库应用程序看起来没有瓶颈，即 CPU 和磁盘使用率都较高，且所有数据库队列大致平衡。那是否意味着应用程序就不能进一步调整性能？解释你的答案。

24.4 假设一个系统运行三种类型的事务。A 类事务以 50/s 的速度运行，B 类事务以 100/s 的速度运行，C 类事务以 200/s 的速度运行。假设混合事务中包含 25% 的 A 类事务，25% 的 B 类事务，50% 的 C 类事务。

a. 假设事务之间没有干扰，系统的平均事务吞吐率是多少？

b. 什么因素会导致不同类型的事务之间的干扰，以至于计算得到的吞吐率不正确？

24.5 列出预见标准和反应标准相比的优点和缺点。

习题

1058 24.6 找出你特别喜爱的数据库提供的所有性能信息。至少找出以下这些：有哪些当前正在执行或者最近执行过的查询，它们各消耗了哪些资源（CPU 和 I/O），有些页面片段的请求导致了缓冲区遗漏（可能的话针对每个查询），有些锁被高度争夺。你还可以从操作系统上获得关于 CPU 和 I/O 使用率的信息。

24.7 a. 调节数据库系统的哪三个主要层次可以提高性能？

b. 对每个层次请举出两个例子说明调节是如何进行的。

24.8 当进行性能调整时，应该首先调整硬件（通过增加磁盘或者内存），还是应该首先调整事务（通过增加索引或者物化视图）。解释你的答案。

24.9 假设你的应用程序有这样的事务：每个事务访问并更新存储在 B+ 树文件组织的大关系中的单个元组。假定 B+ 树的所有内部结点都在内存中，但只有非常少量的叶子页面能放进内存。解释如何计算支持每秒钟 1000 个事务的工作负载最少需要的磁盘数。利用 10.2 节给出的磁盘参数值，计算需要的磁盘数。

24.10 将一个长事务分解成一系列小事务的动机是什么？其结果会引发什么问题？这些问题如何避免？

24.11 假设内存价格下降一半，磁盘访问速度（每秒的访问次数）加倍，其他因素保持不变。此改变对于 5 分钟规则和 1 分钟规则将会有什么影响？

24.12 列出 TPC 基准程序的至少 4 个有助于得到实际的和可信赖的评测结果的特性。

24.13 TPC-D 基准程序为什么会被 TPC-H 和 TPC-R 基准程序所替代？

24.14 解释应用程序的哪些特性有助于你决定最好选择 TPC-C、TPC-H 和 TPC-R 中的哪个来对应用程序进行建模。

文献注解

Kleinrock［1975］是关于排队论的经典教科书。

数据库系统基准程序的一个早期建议（Wisconsin 基准程序）由 Bitton 等［1983］提出。TPC-A、TPC-B 和 TPC-C 基准程序在 Gray［1991］里介绍。所有 TPC 基准程序描述以及基准程序结果的联机版本可以在 URL 为 www.tpc.org 的万维网上获得；该站点还包含有关新基准程序建议的最新信息。OODB 的 OO1 基准程序在 Cattell 和 Skeen［1992］中描述；007 基准程序在 Carey 等［1993］中描述。 1059

Shasha 和 Bonnet［2002］提供了数据库调整方面的详细报告。O'Neil 和 O'Neil［2000］是一本很好地描述了性能度量和调整的教科书。Gray 和 Graefe［1997］描述了 5 分钟规则和 1 分钟规则，Graefe［2008］最近扩充到考虑了主存、闪存和磁盘的组合。

Ross 等［1996］、Chaudhuri 和 Narasayya［1997］、Agrawal 等［2000］和 Mistry 等［2001］中讨论了索引选择和物化视图选择。Zilio 等［2004］、Dageville 等［2004］和 Agrawal 等［2004］中描述了 IBM DB2、Oracle 和 Microsoft SQL Server 支持的调整功能。

有关 ODBC、OLE-DB、ADO 和 ADO.NET 的信息可以在 Web 站点 www.microsoft.com/data 中找到，并且有关该主题的许多书籍可以通过 www.amazon.com 找到。每季发行的 *ACM Sigmod Record* 中有一个常规章节用于讲述数据库中的标准。

关于 XML 的标准和工具的大量信息可以在 Web 站点 www.w3c.org 在线找到。有关 RosettaNet 的信息可以在 Web 站点 www.rosettanet.org 找到。

Cook［1996］讲述了商务处理的工作流设计。Umar［1997］讲述了工作流设计系统产品处理中的问题。 1060

第25章
Database System Concepts, 6E

时空数据和移动性

在数据库的大部分历史中，存储在数据库中的数据类型相对简单。这可以从SQL的早期版本只支持非常有限的数据类型上反映出来。但是，随着时间的流逝，在数据库中处理诸如时态数据、空间数据和多媒体数据那样的复杂数据类型的需求不断增强。

另一个主要趋势造就了其自身的一些问题：移动计算机的发展，从膝上型计算机和袖珍管理器开始，最近几年发展成为带有内置计算机的移动电话，以及在商业应用中越来越多地使用的各种可嵌入计算机。

在本章中，我们学习几种数据类型，以及与这些应用相关的另外一些数据库问题。

25.1 动机

在详细阐述每个主题之前，我们总结一下各种数据类型的动机，以及在处理它们时的一些重要问题。

- **时态数据**(temporal data)。大多数数据库系统对世界的当前状态建模，比如当前的客户、当前的学生和当前提供的课程。在许多应用中，存储和检索有关过去状态的信息非常重要。历史信息可以手工加入到模式设计中。但是，这项工作可以通过数据库对时态数据的支持而大大简化，我们将在25.2节学习时态数据。
- **空间数据**(spatial data)。空间数据包括**地理数据**(geographic data)，如地图和相关信息，以及**计算机辅助设计数据**(computer-aided-design data)，如集成电路设计或者建筑设计。空间数据应用最初将数据作为文件保存到文件系统中，这和早期的商务应用相同。但是随着数据复杂度和数据量，以及用户数量的增加，在文件系统中存储和检索数据的即席方式已被证明不能满足许多使用空间数据应用的需求。

1061

空间数据应用需要数据库系统提供的功能——尤其是有效存储和查询大量数据的能力。一些应用可能还需要其他的数据库特性，如对被存储数据的某个部分进行原子更新、持久性以及并发控制。在25.3节中，我们学习为支持空间数据而需要对传统数据库进行的扩展。

- **多媒体数据**(multimedia data)。在25.4节中，我们学习存储诸如图像、视频和音频数据这样的多媒体数据的数据库系统所需要的特性。视频和音频数据的显著特性是显示这些数据要求数据检索具有稳定的、可预先确定的速率；因此，这样的数据称作**连续媒体数据**(continuous-media data)。
- **移动数据库**(mobile database)。在25.5节中，我们学习移动计算系统对数据库的要求。移动计算系统包括膝上型计算机、上网本电脑和高端手机等，它们通过无线数字通信网络连接到基站。这种计算机与第19章讨论的分布式数据库系统不同，在和网络断开连接的时候仍需要能够运行。它们也只具备有限的存储能力，这就需要特殊的内存管理技术。

25.2 数据库中的时间

数据库对其外部真实世界的某些方面的状态进行建模。一般而言，数据库只对真实世界的一个状态——当前状态建模，不会保存有关过去状态的信息，除非可能用作审计跟踪。当真实世界的状态改变了，数据库被更新，有关过去状态的信息就丢失了。但是，在许多应用中，存储和检索过去状态的

信息非常重要。例如，病人数据库必须保存关于病人医疗历史的信息。工厂监控系统可能保存关于工厂中传感器当前和过去的读入的信息以用于分析。存储关于真实世界的时间经历状态的信息的数据库叫做**时态数据库**(temporal database)。

在考虑数据库系统中的时间问题时，我们必须区分出系统中测量的时间和真实世界中观察到的时间。一个事实的**有效时间**(valid time)是一个时间段的集合，在这些时间段内该事实在现实世界中为真。一个事实的**事务时间**(transaction time)是该事实当前处于数据库系统中的时间段。后一种时间基于事务当行化顺序并由系统自动产生。注意有效时间段是一个真实世界中的概念，它不能自动产生且必须提供给系统。

时态关系(temporal relation)的每个元组具有一个该元组为真时的相关时间，这个时间可以是有效时间或事务时间。当然，可以同时存储有效时间和事务时间，这种情况下的关系称为**双时态关系** 1062
(bitemporal relation)。图25-1显示了一个时态关系的例子。为了简化表示，每个元组只有一个与之相关联的时间段，因此，对应于每个元组为真的互不相交的各个时间段，只存在元组的一次表示。这里的时间段是用一对 *from* 和 *to* 属性来表示的，实际实现时可能会有一个包含这两个字段的结构类型，也许叫做 *Interval*。注意有些元组在 *to* 时间列上出现"∗"，这些星号表明元组在 *to* 时间列的值发生改变之前一直为真。因此，这些元组当前为真。尽管时间以文本形式给出，但是在内部存储时可以采取更紧凑的形式，例如可以存储从某个固定日期的某一固定时刻(比如1900年1月1日12:00)起所经过的秒数，而秒数可以转换回通常的文本形式。

ID	name	dept_name	salary	from	to
10101	Srinivasan	Comp. Sci.	61000	2007/1/1	2007/12/31
10101	Srinivasan	Comp. Sci.	65000	2008/1/1	2008/12/31
12121	Wu	Finance	82000	2006/1/1	2006/12/31
12121	Wu	Finance	87000	2007/1/1	2007/12/31
12121	Wu	Finance	90000	2008/1/1	2008/12/31
98345	Kim	Elec. Eng.	80000	2005/1/1	2008/12/31

图25-1 一个时态关系 *instructor*

25.2.1 SQL中的时间规范

如我们在第4章所见，SQL标准定义了 **date**，**time** 和 **timestamp** 类型。**date** 类型包含表示年的四位数字(1-9999)、表示月的两位数字(1-12)和表示日的两位数字(1-31)。**time** 类型包含表示小时的两位数字表示分钟的两位数字、表示秒的两位数字以及可选的小数位。秒字段可以超过60，以允许润秒的情况。由于地球旋转速度存在微小变化，在有些年里需要增加润秒来进行校准。**timestamp** 类型包含 **date** 和 **time** 字段，其中秒字段上有六位小数。

因为世界上的不同地区有不同的本地时间，经常需要指出时间所在的时区。**全球协调时间**(Universal Coordinated Time，UTC)是指定时间的标准参考点，而本地时间被定义成 UTC 的偏移。(这个标准的简写是 UTC，而不是 UCT，因为"Universal Coordinated Time"在法语中被写作"*universel temps coordonné*"，而它是法语的简写。)SQL 还支持 **time with time zone** 和 **timestamp with time zone** 两种类型，它们用本地时间加上本地时间对 UTC 的偏移来指定时间。例如，我们可以用美国东部标准时间和偏移 -6：00 来表示时间，因为美国东部标准时间比 UTC 晚六个小时。

SQL 支持一种叫做 **interval** 的类型，它使我们可以表示像"一天"或者"两天零五个小时"这样的一段时间，而无需指出这段时间何时开始。这个概念和以前我们用到的时间段概念不同，前面提到的时 1063
间段是指有指定开始和结束时间的一段时间。[⊖]

25.2.2 时态查询语言

没有时态信息的数据库关系有时叫做**快照关系**(snapshot relation)，因为它反映了现实世界的一个

⊖ 许多时态数据库研究者认为，这种类型应该称为**跨度**(span)，因为它并不指明确切的开始或结束时间，只是两者之间的一个时间跨度。

快照中的状态。因此，时态关系在时间点 t 的快照是关系中在 t 时刻为真的元组通过投影去除时间戳属性后的集合。时态关系上的快照操作产生该关系在指定时刻（或者在不指定时刻的情况下就指当前时刻）的快照。

时态选择（temporal selection）是涉及时间属性的选择；**时态投影**（temporal projection）是一种投影，其中的元组继承了原始关系中相应元组的时间。**时态连接**（temporal join）是一种连接，连接结果中元组的时间是产生该元组的两个元组的时间的交。如果时间不相交，则从结果中去除该元组。

谓词 *precedes*、*overlaps* 和 *contains* 可以用在时间段上。它们的含义应该很明显。*intersect* 运算可以用在两个时间段上，得到单个（可能为空）的时间段。然而，两个时间段的并可能是单个时间段，也可能不是。

如我们在 8.9 节所见，在时态关系中使用函数依赖必须小心。尽管任一给定时刻教师号可以函数决定其工资，但显然工资可能随时间发生变化。如果对模式 R 的所有合法实例 r 而言，r 的所有快照都满足函数依赖 $X \rightarrow Y$，则说**时态函数依赖**（temporal functional dependency）$X \xrightarrow{T} Y$ 在关系模式 R 上成立。

人们已经提出了一些扩展 SQL 以增强其对时态数据支持的建议。但至少到 SQL：2008 为止，除与时间相关的数据类型和操作以外，SQL 还没有提供任何对时态数据的特殊支持。

25.3 空间与地理数据

数据库支持空间数据对于有效存储、索引以及查询基于空间位置的数据非常重要。例如，假设我们希望在数据库中存储一组多边形，然后查询该数据库以找出与给定多边形相交的所有多边形。我们不能使用标准的索引结构（如 B 树或散列索引）来有效地回答这样的查询。有效处理上述查询需要专用的索引结构如 R 树（我们将在后面学习）来执行这个任务。

有两种类型的空间数据特别重要：

- **计算机辅助设计**（Computer-Aided-Design，CAD）**数据**，它包括关于物体（如建筑、汽车或飞机）是如何构造的空间信息。另外的有关计算机辅助设计数据库的重要例子是集成电路和电子设备规划。
- **地理数据**（geographic data），例如道路图、土地利用图、地形海拔图，显示边界的政治用地图、土地所有权地图等。**地理信息系统**（geographic information system）是为存储地理数据而定制的专用数据库。

1064

许多数据库系统都增加了对地理数据的支持，如 IBM DB2 Spatial Extender、Informix Spatial DataBlade 和 Oracle Spatial。

25.3.1 几何信息表示

图 25-2 说明了各种几何结构如何以规范化的形式在数据库中表示。我们这里要强调的是，几何信息的表示可以用多种不同方式，我们只描述了其中的几种。

线段可以由其端点的坐标表示。比如，在一个地图数据库中，一个点的两个坐标可以是它的纬度和经度。**折线**（polyline）（也叫**线串**（linestring））由相连的线段序列组成，可以由一个包含线段端点坐标的顺序列表来表示。对于任意曲线，我们可以将它划分成一个线段序列，从而用折线来近似地表示。这种表示对二维特征（如道路）是有用的：这里，道路的宽度相对于整张地图的大小来说是足够小的，可以将其认为是一条线。有些系统还将圆弧作为原语，允许将曲线表示为圆弧序列。

我们可以通过按顺序列出多边形的顶点来表示多边形，如图 25-2 所示。[⊖] 顶点的列表指出了多边形区域的边界。在另一种表示方式中，多边形可以分割成一组三角形，如图 25-2 所示。该过程称为**三角剖分**（triangulation）。任意多边形都可以被三角剖分。复杂多边形被赋予一个标识，它所分割出来的每个三角形都具有该多边形的标识。圆和椭圆可以由相应类型来表示，也可以用多边形来近似表示。

基于列表的折线或多边形表示对于查询处理往往很方便。当底层数据库支持时，这种非一范式表

⊖ 一些参考文献中使用闭合多边形这个术语来表示我们所说的多边形，并称折线为开放多边形。

示也可以使用。因此我们可以使用固定大小的元组（以第一范式形式）表示折线。我们可以赋予折线或曲线一个标识，每条线段用一个单独的元组表示，该元组也带有相应多边形或曲线的标识。类似的，多边形的三角剖分表示允许用第一范式关系来表示多边形。

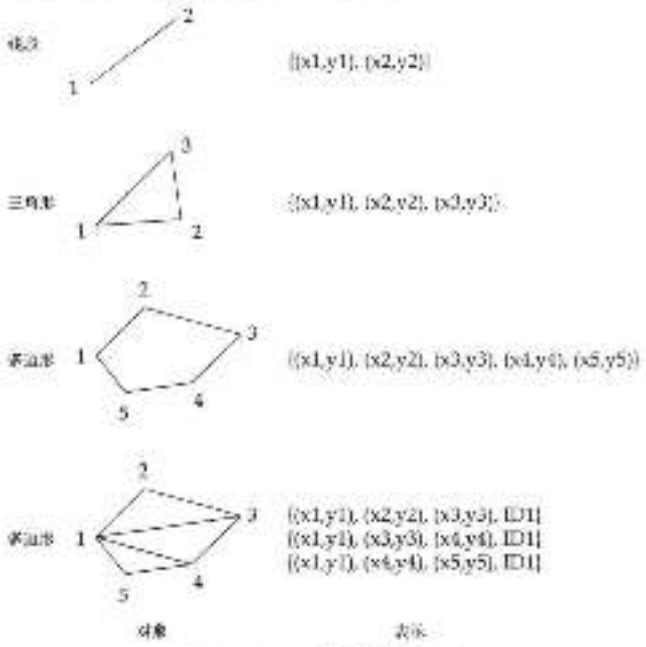


图 25.2　几何结构的表示

三维空间中点与线段的表示类似于其在二维空间中的表示，唯一的区别是点多了额外的 z 坐标。类似地，从二维到三维，平面图形（如三角形、矩形及其他多边形）的表示没有太大变化。四面体和立方体可以用类似于三角形和矩形的方法来表示。我们可以通过将多面体分割成若干四面体来表示任意多面体，就像我们对多边形的三角剖分。我们也可以通过列出多面体所有的面来表示多面体，每个面本身是一个多边形，使用这种表示还需要指出该面的哪一侧是多面体的内侧。

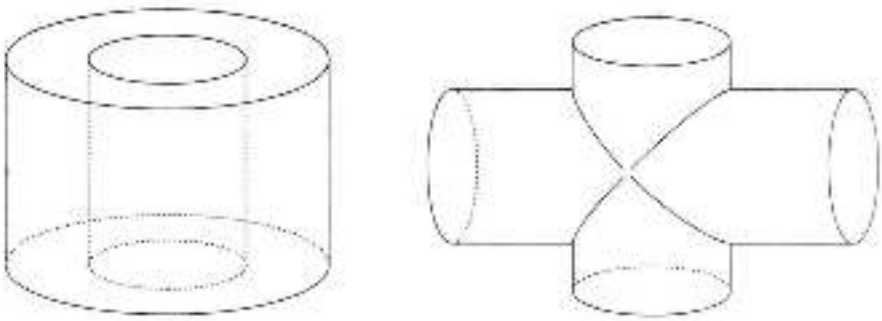
25.3.2　设计数据库

传统上，**计算机辅助设计**（Computer-Aided-Design，CAD）系统在编辑或做其他处理时将数据存储在内存中，并且在一次编辑结束时将数据写回文件。这种方式的缺点包括将数据由一种形式转化为另一种形式的开销（编程的复杂性和时间开销），以及即使只需要其中一部分也必须读入整个文件。对于大型设计，如大规模集成电路设计或整架飞机设计，可能不能将整个设计全放到内存中。面向对象数据库设计者很大程度上受 CAD 系统对数据库需求的驱动。面向对象数据库将设计中的组件表示成对象，并且对象间的连接表明了设计的构造方式。 1065 | 1066

设计数据库中存储的对象通常是几何对象。简单的二维几何对象包括点、线、三角形、矩形和更为一般化的多边形。复杂的二维对象可以通过简单对象的并、交、差运算得到。类似地，复杂的三维对象可以用更简单的对象（如球体、圆柱体和立方体）的并、交、差运算得到，如图 25.3 所示。三维表面也可以用**线框模型**（wireframe model）表示，其本质是将表面建模成一组更简单的对象，如线段、三角形和矩形。

设计数据库也存储有关对象的非空间信息，如构造对象的材料。我们通常可以用标准数据建模技术来对这些信息建模。这里我们只关心空间方面。

设计中必须执行多种空间运算。例如，设计者可能想要检索对应于某个特定感兴趣区域的设计部分。25.3.5 节讨论的空间索引结构对这类任务是有用的。空间索引结构是多维的，处理的是二维或二维数据，而不仅仅是 B^+ 树提供的那种简单的一维排序。



a）圆柱体的差　　b）圆柱体的并

图 25-3 复杂二维对象

空间完整性约束，如"两根水管不应在同一位置"，对于在设计数据库中防止录入错误是很重要的。如果设计是手工做的，这种错误就经常发生，并且只有当原型构建出来时才会被检测出来。因此，这些错误修改起来代价很高。数据库对空间完整性约束的支持可以帮助人们避免设计错误，从而保证设计的一致性。这种完整性检查的实现主要依赖于有效多维索引结构的可用性。

25.3.3 地理数据

地理数据本质上是空间数据，但与设计数据相比有几个方面的不同。地图和卫星图像是地理数据的典型例子。地图不仅可以提供位置信息，如关于边界、河流和道路的信息，而且还提供更多的与位置相关的详细信息，如海拔、土壤类型、土地使用和年降雨量。

25.3.3.1 地理数据的应用

地理数据库有多种用途，包括联机地图服务、交通导航系统、公共服务设施的分布网络信息（如电话、电力和供水系统）以及为生态学家和规划者提供的土地使用信息。

基于 Web 的道路地图服务成为使用非常广泛的地图数据应用。在最简单的情况下，这些系统可用于生成所需区域的联机道路地图。联机地图的一个重要好处在于它可以容易地缩放至所需大小，也就是为定位相关特征而进行的放大或缩小。道路地图服务还存储关于道路和服务的信息，如道路布局、道路上的速度限制、道路状况、道路间的连接以及单行限制。有了这些有关道路的附加信息，地图就可以用来获得从一个地方到另一个地方的指向，并用于自动旅行规划。用户可以查询有关服务的联机信息来定位诸如能够提供所需服务和一定价格范围内的旅馆、加油站或餐馆。近年来，一些基于 Web 的地图服务定义了 API，允许程序员创建定制的地图，它包括来自该地图服务以及其他来源的数据。这种定制的地图可用来显示诸如特定区域内的待售或待租的房子，或者商店和餐馆。

交通导航系统是安装在车辆上的提供道路地图和旅行规划服务的系统，它们包括一个全球定位系统（Global Positioning System，GPS）部件，它使用 GPS 卫星广播的信息，以几十米的精度确定当前位置。拥有这样的系统，司机永远[⊖]也不会迷路，GPS 部件用纬度、经度和海拔的形式来确定位置，导航系统可以查询地理数据库以找出车辆当前的位置和所位于的道路。

用于公共设施信息的地理数据库随着地下电缆和管道网络的增长变得非常重要。如果没有详细地图，一种设施的工程可能毁坏另一种设施的电缆，导致大范围的服务中断。地理数据库加上精确定位系统有助于避免这类问题。

1067
|
1068

25.3.3.2 地理数据的表示

地理数据（geographic data）可以分为两类：

- **光栅数据**（raster data）。这种数据由二维或更高维的位图或像素图组成。二维光栅图像的典型例子是地区的卫星图像。除实际图像之外，数据还包括图像的位置（例如由它的角的纬度和经度

⊖ 原书误为"没有"！

指定)和分辨率(由总像素数，或者在地理数据环境中更为普遍的每像素覆盖面积所指定)。

光栅数据通常用**栅格**(tile)表示，每个栅格覆盖固定大小的区域。可以通过显示与区域重叠的所有栅格来显示更大的区域。为了允许在不同的缩放级别显示数据，就为每个缩放级别创建一个单独的栅格集。一旦通过用户界面(例如Web浏览器)设置了缩放级别，与待显示区域重叠的该缩放级别的栅格就被检索和显示出来。

光栅数据可以是三维的。例如，同样在卫星帮助下测得的不同地区在不同高度上的温度。时间可以形成另一个维度，例如，不同时间上不同点的表面温度度量。

- **矢量数据**(vector data)。矢量数据由基本几何对象构成，如点、线段、折线、三角形和其他二维多边形，以及圆柱体、球体、立方体和其他三维多面体。在地理数据环境中，点通常用纬度和经度表示，此外在与高度相关的地方还用海拔表示。

  地图数据常以矢量形式表示。道路常被表示成折线。地理特征(如大型湖泊)，或者甚至是政治特征(如州和国家)，可以表示成复杂多边形。某些特征(如河流)可以表示成复杂曲线或复杂多边形，这取决于其宽度是否相关。

关于地区的地理信息(如年降雨量)可以表示成一个数组，也就是以光栅的形式。为提高空间效率，该数组可以用压缩形式存储。在25.3.5.2节中，我们研究这种数组的另一种表示，它使用称为四叉树的数据结构。

作为另一种选择，我们可以以矢量形式用多边形来表示区域信息，其中每个多边形代表一个区域，该区域内部的数据值是相同的。在某些应用中矢量表示比光栅表示更简洁，而且对于某些任务它也更为精确，如在表示道路时，将区域划分成像素(可能相当大)会导致位置信息精确度的损失。但是，矢量表示不适合于数据本质上就基于光栅的应用，如卫星图像。 1069

地形(topographical)信息，即有关表面上每个点的海拔(高度)的信息，可以用光栅形式表示。另外还可以通过将表面划分成覆盖(近似)等高区域的多边形，而以向量形式来表示，其中每个多边形关联单个的高度值。作为另一种选择，可以对表面进行**三角剖分**(triangulated)(即划分成三角形)，每个三角形用它的每个角的纬度、经度和海拔表示。后一种表示称为**三角剖分的不规则网络**(Triangulated Irregular Network，TIN)表示，这种简洁表示对于产生一个区域的三维视图尤其有用。

地理信息系统通常包含光栅和矢量两种数据，当给用户显示结果时可以合并这两种类型的数据。例如，地图应用通常包含了关于道路、建筑和其他陆标的卫星图像和矢量数据。地图显示通常**重叠**(overlay)各种不同的信息；例如，道路信息可以重叠在卫星图像背景上，构成一个混合显示。事实上，地图通常由多个层组成，这些层以自底向上的顺序显示；来自较高层的数据出现在较低层的数据之上。

另外有趣的是，注意即使实际上以矢量形式存储的信息在发送给用户界面(如Web浏览器)前也可以转换为光栅形式。一个原因是，即使网页浏览器不支持脚本语言(解释和显示矢量数据所需的)也仍可以显示地图数据；第二个原因可能是防止终端用户提取和使用矢量数据。

地图服务(如Google Maps和Yahoo! Maps)提供了API，允许用户创建专用的地图显示，包括重叠在标准地图数据上的专用数据的应用。例如，一个网站可能显示一个区域的地图，将有关餐馆的信息重叠在地图上。可以动态地构造这种重叠，例如，只显示具有特定风格的餐馆，或者允许用户更改缩放级别或平移显示。用于特定语言(典型的是JavaScript或Flash)的地图API是构建在提供底层地图数据的Web服务上的。

25.3.4 空间查询

有许多类型的查询都涉及空间位置。

- **临近查询**(nearness query)要求找出位于特定位置附近的对象。找出位于给定位置的给定距离内的所有餐馆的查询就是临近查询的一个例子。**最近邻居查询**(nearest-neighbor query)要求找出离特定点最近的对象。例如，我们可能想要找出最近的加油站。注意这个查询不必指定距离的范围，因此即使不知道最近的加油站有多远，也可以提出这个查询。
- **区域查询**(region query)处理的是空间区域。这种查询可以找出部分或全部位于指定区域内的对象。找出给定城镇的地理边界内的所有零售店的查询就是一个例子。 1070

- 查询也可能要求区域的交和并。例如，给出区域信息，如年降雨量和人口密度，查询可能要求找出年降雨量低并且人口密度高的所有区域。

计算区域的交的查询可以看作是计算两个空间关系的**空间连接**(spatial join)。举例来说，一个关系代表降雨量，另一个代表人口密度，而位置则扮演连接属性的角色。一般地，给定两个关系，其中每个关系都包含空间对象，则两个关系的空间连接要么生成若干对相交的对象，要么生成这些对象的相交区域。

对矢量数据有效计算空间连接的连接算法有几种。尽管可以采用嵌套循环连接和索引嵌套循环连接(使用空间索引)，但对空间数据不能使用散列连接和排序归并连接。研究人员已经提出了一些基于两个关系上空间索引结构的坐标遍历的连接技术。更多信息请参见文献注解。

一般地，空间数据查询可能是空间和非空间请求的结合。例如，我们可能想要找出可以提供素食的、每餐花费少于10美元的最近的餐馆。

由于空间数据固有的图形化，我们通常使用图形化查询语言对它们进行查询。这种查询的结果也以图形化方式显示，而不是以表的形式显示。用户可以调用界面上的各种操作，比如选择一个查看区域(例如，通过指向并点击曼哈顿的西部)、放大或缩小、根据选择条件选择显示什么(例如，有多于三个卧室的房子)、重叠多张地图(例如，多于三个卧室的房子与一张显示低犯罪率区域的地图重叠)等。图形界面构成了前端。现已提出了对SQL的扩展，允许关系数据库有效地存储和检索空间信息，并且也允许空间和非空间混合条件的查询。这些扩展包括允许抽象数据类型(如线、多边形、位图)，以及允许空间条件(如包含或重叠)。

25.3.5 空间数据的索引

有效访问空间数据需要索引。诸如散列索引和B树那样的传统索引结构是不合适的，因为它们只处理一维数据，而空间数据通常是二维或更高维的。

25.3.5.1 k-d树

要理解如何对由二维或更高维构成的空间数据建立索引，我们先考虑一维数据中点的索引。树结
1071 构(如二叉树和B树)是不断地将空间划分成更小的部分。例如，二叉树的每个内部结点将一维区间划分成两个部分。位于左边区间的点进入左子树，位于右边区间的点进入右子树。在平衡二叉树中，划分的选择使得存储在子树中的大约一半的点落入每个区间。类似地，B树的每一层将一个一维区间划分成多个部分。

我们可以用直觉建立二维空间以及更高维空间的树结构。一种称为k-d树(k-d tree)的树结构是用于多维索引的一种早期结构。k-d树的每一层将空间划分成两个部分。树的顶层结点按一维进行划分，下一层结点按另一维进行划分，依此类推，各维循环往复。划分以这种方式进行：在每个结点上大约有一半存储在子树中的点落入一侧，而另一半落入另一侧。当一个结点中的点数少于给定的最大点数时，划分结束。图25-4表示了一组二维空间中的点，以及这组点的k-d树表示。每条线对应于树中的一个结点，叶结点中的最大点数设为1。图中的每条线(除了外边的方框)对应于k-d树的一个结点。图中线的标号表示相应结点出现在树中的层数。

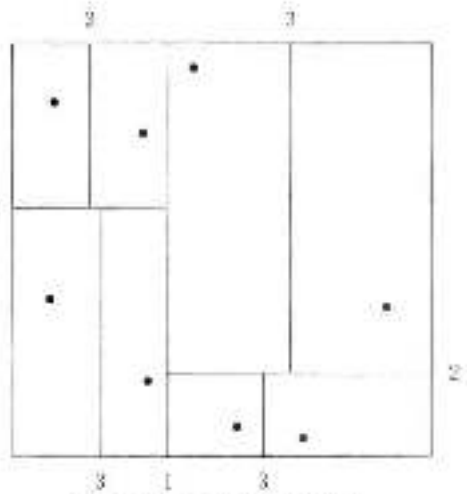


图25-4 用k-d树划分空间

k-d-B树扩展了k-d树，允许每个内部结点有多个子结点，如同B树扩展了二叉树以减小树的高度一样。k-d-B树比k-d树更适合于辅助存储器。

25.3.5.2 四叉树

二维数据的另一种表示形式是**四叉树**(quadtree)。用四叉树划分空间的例子如图25-5所示。点的
1072 集合与图25-4中的一样。四叉树的每个结点对应于一个矩形区域空间。顶层结点对应于整个目标空间。四叉树中的每个非叶结点将其区域分成四个同等大小的象限，相应地，每个这样的结点有四个子

结点对应于四个象限。叶结点拥有从零到某个固定最大数目之间的点数。相应地，如果对应于一个结点的区域有多于最大数目的点数，就需要为该结点创建子结点。在图 25-5 所示的例子中，叶结点的最大点数设为 1。

这种类型的四叉树称为 PR 四叉树(PR quadtree)，表示它存储的是点，并且空间的划分是根据区域而不是根据所存储的实际点集。我们可以用区域四叉树(region quadtree)存储数组(光栅)信息。如果某结点所覆盖的区域内的所有数组值都相同，那么区域四叉树的这一结点是叶结点。否则，它被进一步划分为四个具有相等区域的子结点，从而是内部结点。区域四叉树中的每个结点对应于值的一个子数组。对应于叶结点的子数组要么只包含单个数组元素，要么包含多个具有相同值的数组元素。

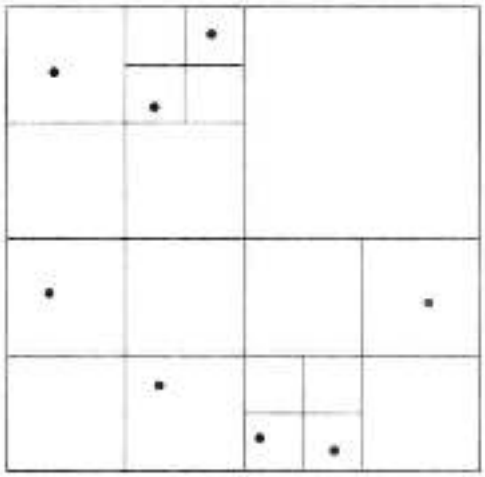


图 25-5　用四叉树划分空间

线段和多边形的索引带来了新的问题。为此可以对 k-d 树和四叉树进行扩展。但是，线段或多边形可能跨越分界线。如果这样的话，就必须将它分割开，然后在其各个部分所出现的每个子树中加以表示。线段或多边形的多次出现会导致存储和查询效率低下。

25.3.5.3　R 树

一种称作 R 树(R-tree)的存储结构对诸如点、线段、矩形和其他多边形对象的索引是很有用的。R 树是平衡树结构，其中被索引的对象存储在叶结点上，这一点非常像 B⁺树。但它并不使用值的范围，而是用矩形**边界框**(bounding box)与每个树结点关联。叶结点的边界框是平行于包含叶结点中所有存储对象的坐标轴的最小矩形。类似地，内部结点的边界框是平行于包含其子结点边界框的坐标轴的最小矩形；一个对象(例如一个多边形)的边界框定义为平行于包含该对象的坐标轴的最小矩形。

每个内部结点存储子结点的边界框和指向子结点的指针。每个叶结点存储被索引的对象，还可以有选择地存放对象的边界框；边界框有助于加快检查矩形与被索引的对象是否重叠的速度——如果查询矩形与对象的边界框不重叠，则它也不可能与对象重叠。(如果索引对象就是矩形，当然不需要存储边界框，因为边界框就是矩形的。)

图 25-6 中的例子表示了一组矩形(用实线画出)及对应于这组矩形的 R 树中结点的边界框(用虚线画出)。注意为了在图示中突出边界框，画边界框的时候在内部留了一些额外空间。实际上，这些框应该更小一些，恰好适合它们包含的对象；也就是说，边界框 B 的每条边应该紧接 B 中所包含的至少一个对象或边界框。

R 树本身如图 25-6 的右边所示。图中用 BB_i 表示边界框 i 的坐标。

现在我们来看如何在 R 树上执行查找、插入和删除操作。

- **查找**：如图所示，兄弟结点所对应的边界框可能重叠；相反，B⁺树、k-d 树和四叉树的区间是不重叠的。下层，搜索包含一个点的对象必须遍历其边界框包含该点的所有子结点；其结果是可能需要搜索多条路径。类似地，找出与给定对象相交的所有对象的查询就必须向下遍历其边界框矩形与给定对象相交的每个结点。
- **插入**：当我们往 R 树中插入一个对象时，我们选择一个叶结点来存放该对象。理想情况是我们应该找到一个叶结点，它有存放一个新项的空间，而且它的边界框包含该对象的边界框。但是，这样的结点可能不存在；即使存在，找到该结点也可能非常昂贵，因为不可能通过从根向下的一次遍历就能找到它。在每个内部结点我们都可能发现多个子结点的边界框包含该对象的边界框，而每个这样的子结点都要搜索。由此得到下面的启发式算法：在从根结点开始的遍历中，如果有多个子结点的边界框包含该对象的边界框，R 树算法就从中任取一个。如果没有一个子结点满足这个条件，算法选择一个其边界框与该对象边界框重叠最大的子结点继续遍历。

1073

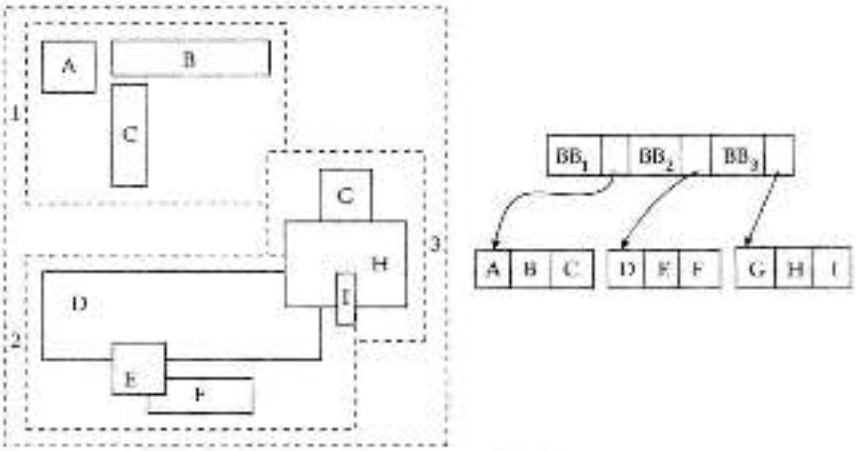


图 25-6 一棵 R 树

一旦到达叶结点，如果该结点已满，算法就要进行结点分裂(必要时分裂会向上传播)。其方式非常类似于 B⁻树的插入。与 B⁺树的插入类似，R 树的插入算法确保树的平衡性。另外，插入算法还保证叶结点和内部结点的边界框保持一致，即叶结点的边界框包含存储在该叶结点中的所有对象的边界框，同时内部结点的边界框包含所有子结点的边界框。

这个插入过程与 B⁻树插入过程的主要区别在于结点的分裂方式。在 B⁺树中，有可能找到一个中间值，使得一半的项比它小，一半的项比它大。但此性质不能推广到超过一维的情况；也就是说，对于多维的情况，不可能总能将项集分裂为两个边界框不相交的集合。取而代之的一种启发式算法是：将项集 S 分裂为两个不相交的集合 S_1 和 S_2，使得它们的边界框的总面积最小。另一种启发式算法将项集分裂为两个重叠面积最小的集合 S_1 和 S_2。分裂之后得到的两个结点分别包含 S_1 和 S_2 中的项。找到最小总面积或最小重叠的分裂方法本身代价很大，因此还使用代价更小的启发式算法，如二次方分裂(quadratic split)启发式算法。(这种启发式算法的名称来源于算法所花的时间是项数目的二次方这个事实。)

1074
1075

二次方分裂(quadratic split)启发式算法这样进行：首先，它从 S 中选取两个项 a 和 b，使得如果将它们放到同一个结点中将产生具有最大浪费空间的边界框；即 a 和 b 的最小边界框的面积减去 a 与 b 的面积之和后的结果最大。启发式算法将 a 和 b 这两个项分别放到集合 S_1 和 S_2 中。

然后该算法进行迭代，每次迭代将一个剩下的项加入集合 S_1 或 S_2 中。在每次迭代时，对于每个剩余项 e，令 $i_{e,1}$ 表示将 e 加入 S_1 时 S_1 的边界框增加的大小；$i_{e,2}$ 表示对于 S_2 相应增加的大小。在每次迭代中，该启发式算法选择 $i_{e,1}$ 与 $i_{e,2}$ 差异最大的那个项，如果 $i_{e,1}$ 小于 $i_{e,2}$ 则将其加入 S_1 中，否则加入 S_2 中。也就是说，每次迭代选择对于 S_1 或 S_2 有"最大优先权"的项。当分配完所有项之后，或者当 S_1 或 S_2 中的一个拥有了足够的项(其余项必须全部加入另一个集合，才能使这两个集合所创建的结点都达到所需的最小占有量)停止迭代。此时，该启发式算法就把所有未分配的项加入含有较少项的集合中。

- 删除：删除与 B⁺树删除的执行类似，如果一个结点不满，就从兄弟结点借来项或与兄弟结点合并。另一种可选的方法是把不满结点中的所有项重新分配到兄弟结点中，其目的是提高 R 树的项聚集度。

有关 R 树的插入和删除操作以及 R 树的各种变体(称作 R⁺树或者 R*树)的更多细节，请参见文献注解。

R 树的存储效率比 k-d 树或四叉树更高，因为每个对象只存储一次，并且我们可以很容易地保证每个结点至少是半满的。但是，查询可能更慢一些，因为要搜索多条路径。用四叉树作空间连接比用 R 树更简单，因为一个区域内的所有四叉树都以相同的方式划分。但是，由于 R 树及其变体具有更高

的存储效率及其与 B 树的相似性，它们在支持空间数据的数据库系统中已经很普及。

25.4　多媒体数据库

诸如图像、音频和视频这样的多媒体数据作为日益流行的数据形式，现在几乎总是存储在数据库之外的文件系统中。当多媒体对象数量相对较少时，这种存储方式不会有什么问题，因为数据库所提供的功能往往不那么重要。

但是当存储的多媒体对象数量很大时，数据库功能就变得重要起来了。随之，事务更新、查询机制和索引等问题也变得很重要。多媒体对象常常有描述性属性，如指明它们是何时创建的、由谁创建的以及它们属于哪一类这样的属性。为这种多媒体对象构造数据库的一种方法是用数据库存储描述性属性，并跟踪存储多媒体对象的文件。

但是，将多媒体数据存在数据库之外使得数据库功能更难以提供，譬如基于实际多媒体数据内容上的索引。它还会造成不一致性，譬如一个文件在数据库中作了记录，但其内容丢失了，或相反。因此更好的方式是将数据本身存储在数据库中。

如果要将多媒体数据存储在数据库中，就必须解决几个问题。

- 数据库必须支持大对象，因为诸如视频之类的多媒体数据可能会占据几个 GB 的存储空间。许多数据库系统不支持超过几个 GB 的对象。更大的对象被分裂为小的部分并存储在数据库中。或者，多媒体对象可以存储在文件系统中，而数据库中包含指向对象的指针，典型的指针就是文件名称。SQL/MED 标准(MED 代表外部数据管理)允许将外部数据(如文件)作为数据库的一部分对待。使用 SQL/MED，对象作为数据库的一部分出现，但可以在外部存储。我们在 25.4.1 节讨论多媒体数据格式。
- 某些类型的数据(如音频和视频)的检索要求必须保证以一种平稳的速率来传输数据。这种数据有时称为**等时数据**(isochronous data)或**连续媒体数据**(continuous-media data)。例如，如果没有及时提供音频数据，声音就会有间断。如果数据提供得太快，系统缓冲区就可能会溢出，造成数据丢失。我们在 25.4.2 节讨论连续媒体数据。
- 许多多媒体数据库应用都需要基于相似性的检索。例如，在存储指纹图像的数据库中，提供了要查询的指纹图像，数据库中与该查询指纹相似的指纹必须被检索出来。诸如 B⁺ 树和 R 树这样的索引结构不能用于这个目的，需要创建特殊的索引结构。我们在 25.4.3 节讨论基于相似性的检索。

25.4.1　多媒体数据格式

由于表示多媒体数据需要大量字节数，所以有必要用压缩的形式存储和传输多媒体数据。对于图像数据，使用最广泛的格式就是 JPEG，它是以制定它的标准组织联合图像专家组(Joint Picture Experts Group)来命名的。我们可以通过将视频的每一帧用 JPEG 格式编码来存储视频数据，但这样编码会有浪费，因为视频中连续的帧往往几乎是一样的。移动图像专家组(Moving Picture Experts Group)提出了对视频和音频数据进行编码的一系列 MPEG 标准，这些编码利用帧序列之间的共性达到更高程度的压缩。MPEG-1 标准存储 1 分钟的每秒 30 帧的视频和音频约要 12.5MB(比较而言，只用 JPEG 形式的视频数据约需 75MB)。但是，MPEG-1 编码要损失一些视频质量，其程度粗略相当于 VHS 录像带。MPEG-2 标准是为数字广播系统和数字视频磁盘(DVD)设计的。它带来的视频质量损失是可忽略的。MPEG-2 把 1 分钟的视频和音频压缩到约 17MB。MPEG-4 提供进一步压缩视频的技术，用可变带宽来支持视频数据在很大带宽范围内的网络上进行传输；还有几个竞争的标准用于音频编码，包括代表 MPEG-1 Layer 3 的 MP3、RealAudio、Windows Media Audio 以及其他格式。高清的带有音频的视频可用几个 MPEG-4 的变体进行编码，包括 MPEG-4 AVC 和 AVCHD。

25.4.2　连续媒体数据

最重要的连续媒体数据类型是视频和音频数据(例如，电影数据库)。连续媒体系统的特点是要求实时的信息传输：

- 数据传输必须足够快，使得音频或视频输出没有间断。
- 数据传输的速度不会导致系统缓冲区溢出。
- 必须保持不同数据流之间的同步。这个要求在某些情况下尤其重要。例如，一个人说话时显示嘴唇动作的视频必须与其说话的音频同步。

为了能够预先适时地向大量数据使用者提供数据，从磁盘取得数据时必须仔细协调。通常，数据以周期性的循环方式获取。在每次循环中（设为 n 秒），为每个使用者取得 n 秒的数据并存储在内存缓冲区中，而前一次循环取得的数据从内存缓冲区中传送给使用者。循环周期是一种折中：短周期使用较少的内存但是需要更多的磁盘臂的移动，这样会浪费资源；而长周期能减少磁盘臂移动但是增加了所需内存，而且可能导致初始数据传输的延迟。当新的需求到来时，**接纳控制**（admission control）开始起作用：也就是说，系统检测（每周期中的）可用资源能否满足需求，如果能满足则接纳；否则拒绝。

对连续媒体数据传输的大量研究都是处理诸如磁盘阵列和磁盘失效这样的问题。有关文献注解可得到详细内容。

一些厂商提供视频点播服务器。当前的系统基于文件系统，因为现有的数据库系统不能提供这些应用所需的实时响应。视频点播系统的基本体系结构包括：

- **视频服务器**（video server）。多媒体数据存储在多个磁盘上（通常是用 RAID 的配置）。包含大量
1078 数据的系统可使用三级存储器存储不常访问的数据。
- **终端**（terminal）。人们通过各种设备观看多媒体数据，这些设备统称为终端。这种设备的例子包括个人计算机和连接到**机顶盒**（set-top box）的电视机，机顶盒是一种小的廉价计算机。
- **网络**（network）。从服务器向多个终端传送多媒体数据需要有高容量的网络。

电缆网络上的视频点播服务应用非常广泛。

25.4.3 基于相似性的检索

在许多多媒体应用中，数据只是近似地在数据库中描述。25.4 节中的指纹数据就是一个例子。其他例子还有：

- **图片数据**（pictorial data）。对于在数据库中的表示略微有些不同的两张图片或图像，用户可能认为是相同的。例如，某个数据库可能存储了商标设计。当一个新商标要注册时，系统首先需要识别出以前注册过的所有类似商标。
- **音频数据**（audio data）。现已开发出基于语音的用户界面，允许用户通过说话来发出命令或识别某个数据项。这就必须检测出用户输入与系统中存储的命令或数据项的相似性。
- **手书数据**（handwritten data）。手写输入可用于识别存储在数据库中的手书数据项或命令。这里再次需要相似性检测。

相似性的概念常常是主观的和针对用户的。但是，相似性检测往往比语音或手书识别更成功。因为输入可以与已经在系统中的数据进行比较，这样系统可做的选择集合就受限了。

现在已经有一些算法利用相似性检测来找出给定输入的最佳匹配。许多语音激活系统已有商业应用，特别是用于电话应用和交通工具控制。相关参考文献请参见文献注解。

25.5 移动性和个人数据库

大型商用数据库传统上是存储在中央计算设备上的。在分布式数据库应用中，通常有强大的中央数据库和网络管理。然而有这样几个技术趋势共同产生了一些应用，在这些应用里这种集中的控制和管理假设并不完全正确：

- 膝上型电脑、笔记本电脑或上网本电脑的广泛使用。
1079
- 具有计算机能力的手机的广泛使用。
- 基于无线局域网络、蜂窝数字包网络和其他技术的相对低成本的无线数字通信基础设施的发展。

移动计算（mobile computing）在许多应用中都证明是有用的。许多出差的商务人员都使用膝上型电脑，以便在途中可以工作和访问数据。邮递服务使用移动计算机来辅助包裹的跟踪。紧急响应服务使

用移动计算机在灾难、医疗急救及类似情况的现场访问信息，并且输入与当前情况有关的数据。手机日益成为不仅仅是语音电话服务的设备，而且还是允许电子邮件和 Web 访问的移动计算机。移动计算机的新应用还在继续出现。

无线计算带来了一种新形势：计算机不再有固定的位置和网络地址。**位置相关查询**(location-dependent query)是由移动计算机促成的一类有趣的查询。在这类查询中，用户(计算机)的位置是查询的一个参数。位置参数的值由全球定位系统(GPS)提供。例如，一个旅客信息系统为乘客提供关于旅馆、路边服务及类似信息。有关当前道路前方服务的查询必须根据用户位置、移动方向以及速度进行处理。逐渐地，导航辅助越来越多地成为汽车提供的一个内置功能。

能源(电池电源)对大多数移动计算机来说是一种稀有资源。这一限制影响了系统设计的许多方面。能源效率需求带来的更有趣的结果之一是小的移动设备大部分时间在休眠，只在大约每秒的小部分时间内是醒着的，以接收进来的数据和向外发送数据。这种行为对用于移动设备通信所使用的协议有重大影响。使用预定的数据广播来减少移动系统传输查询的需求是为一种减少能量需求的方法。

越来越多的数据会放在由用户管理的而不是由数据库管理员管理的计算机上。并且，这些计算机有时可能与网络断开连接。在许多情况下，用户在断开连接时仍能继续工作的需求与保持全局数据一致性的需求会产生矛盾。

用户很可能使用不止一种移动设备。不管在给定时间使用哪种设备，用户都需要能够看到他们的数据的最新版本。通常，这种能力由我们在 19.9 节中讨论过的云计算的某些实体来支持。

从 25.5.1 节到 25.5.4 节，我们讨论正在使用和正处于开发过程中的、用于解决移动及个人计算问题的技术。

25.5.1　移动计算模型

移动计算环境由移动计算机(称为**移动主机**(mobile host))和有线计算机网络构成。移动主机通过 1087
称为**移动支持站点**(mobile support station)的计算机与有线网络通信。每个移动支持站点管理在其**单元**(cell)(即它所覆盖的地理区域)内的那些移动主机。移动主机可以在单元间移动，因此必须有从一个移动支持站点到另一个移动支持站点的控制交接(handoff)。由于移动主机有时可能关闭电源，一台主机可能离开一个单元而随后在其他很远的单元内重新出现。因此，单元间的移动不一定是在相邻单元间进行的。在一个小区域内部，如一栋建筑内，移动主机可以通过无线局域网(LAN)相连，它比广域蜂窝网络提供了更低的连接成本，并且减少了交接的开销。

移动主机之间不通过移动支持站点的介入而直接通信是可能的。但是，这种通信只能是在临近主机之间发生。这种直接通信形式通常使用蓝牙(Bluetooth)，蓝牙使用短程数字无线电通信标准，允许以很高的速度(最高 721kb/s)运行 10 米之内范围的无线连接。蓝牙的构思是把蓝牙作为电缆的替代物。蓝牙的最大优势在于，它提供一种简单的、针对移动计算机、PDA、移动电话以及所谓智能设备的即席连接。

现在基于 801.11(a/b/g/n)标准的无线局域网络系统已经被广泛使用，基于 802.16(Wi-Max)标准的系统正在部署中。

移动计算的网络基础架构大部分由两种技术构成：无线局域网和基于包的蜂窝电话网络。早期的蜂窝系统使用模拟技术并设计用于语音通信。第二代数字系统仍然集中于语音应用。第三代(3G)和所谓的 2.5G 系统使用基于包的网络，更适于数据应用。在这些网络中，语音应用只是众多应用中的一种(尽管从经济角度来讲是重要的一种)。第四代(4G)技术包括 Wi-Max 和几个竞争者。

蓝牙、无线局域网以及 2.5G 和 3G 蜂窝网络使多种不同设备间的低成本通信成为可能。然而这种通信本身并不适合通常的数据库应用领域。有关这种通信的客户、监控和管理数据产生了巨大的数据库。无线通信的直接性产生了实时访问许多这种数据库的需要。这种即时性的需求为系统增加了另一维约束，我们将在 26.4 节进一步讨论这个问题。

许多移动计算机的大小和电源限制导致了另一种内存层次结构。它包括闪存存储器以取代磁盘存储器或附加在磁盘存储器之上，我们在 10.1 节讨论了闪存存储器。如果移动主机包含硬盘，硬盘可能允许在不用的时候部下，以节省能源。大小和能源的考虑同样限制了移动设备中所使用的显示器的类

和大小。移动设备设计者常常创建专用用户界面以工作在这些限制之下。但是，出于展现 Web 数据的需求，有必要创建数据的表示标准。**无线应用协议**(Wireless Application Protocol，WAP)就是一个无线因特网访问标准。基于 WAP 的浏览器访问使用**无线标记语言**(Wireless Markup Language，WML)的特
1081 殊 Web 页面，其中 WML 是一种基于 XML 的语言，它是针对移动和无线 Web 浏览的限制而设计的。

25.5.2 路由和查询处理

如果两台主机中有一台是移动主机，则这一对主机之间的路由可能会随时间的推移而发生变化。这个简单的事实会在网络层上产生戏剧性的影响，因为基于位置的网络地址已不再是系统内的常量了。

移动性也直接影响到数据库查询处理。正如我们在第 19 章看到的那样，我们在选择分布式查询处理策略时必须考虑通信代价。移动性导致了通信代价的动态改变，从而使得优化处理复杂化。另外，还有一些相互竞争的代价概念需要考虑：

- **用户时间**(user time)在许多商务应用中是十分宝贵的财富。
- **连接时间**(connection time)在某些蜂窝系统中是决定收费的单位。
- **传输的字节数或包数**(number of bytes，or packets，transferred)在某些数字编窝系统中是计算收费的单位。
- **基于每日不同时段的收费**(time-of-day-based charges)根据通信发生在高峰时段还是非高峰时段而不同。
- **能源**(energy)是有限的。通常，电池电源是珍贵的资源，必须优化其使用。无线电通信的一个基本原理是接收无线电信号需要的能量比发送无线电信号要少。因此，移动主机传输和接收数据所需的能源是不同的。

25.5.3 广播数据

我们常常希望频繁请求的数据由移动支持站点不断周期性地广播，而不是在需要的时候才发送到移动主机。这种**广播数据**(broadcast data)的典型应用是股市价格信息。使用广播数据有两个原因。首先，移动主机避免了发送数据请求的能源开销；其次，广播数据可以立刻被大量的移动主机接收到，而没有额外的开销。因此，可用的传输带宽得到了更有效的利用。

这样，移动主机可以在那些数据发送过来时接收数据，而不会因发送请求而耗费能源。移动主机可能有本地非易失性存储器来高速缓存广播数据，以备将来可以使用。当给定一个查询时，移动主机通过判断是否只使用缓存数据就可以处理该查询来优化能源开销。如果缓存数据不够用，可以有两种选择：等待数据被广播或发送数据请求。要做出这个决定，移动主机必须知道相关数据什么时候会
1082 广播。

广播数据可以按照固定的或变化的时间表发送。对于前一种情况，移动主机使用已知的固定时间表决定什么时候相关数据会发送。对于后一种情况，广播时间表本身必须在一个众所周知的无线电频率上以众所周知的时间间隔进行广播。

实际上，广播介质可以建模为具有高延迟的磁盘。对数据的请求可以认为是在所请求的数据被广播时得到了服务。传输时间表的作用类似于磁盘上的索引。文献注解中列出了最近的有关广播数据管理领域的研究论文。

25.5.4 连接断开与一致性

由于无线通信可能会基于连接时间付费，所以某些移动主机要求能在一段时间内断开连接。除了周期性地以物理方式或通过计算机网络连接到宿主计算机，不带无线连接的移动计算机在它们被使用的大部分时间内都是断开连接的。

在断开连接的时间段内，移动主机仍可以继续运行。移动主机用户可以对驻留于本地的或缓存在本地的数据进行查询和更新。这种情况产生了一些问题，特别是：

- **可恢复性**(recoverability)：如果在移动主机上发生灾难性故障，在断开连接的机器上进行的更新可能会丢失。由于移动主机代表单点故障，所以不可能很好地模拟稳定存储。
- **一致性**(consistency)：移动主机只有在重新连接后才能发现在本地缓存的数据可能已经过时。

与此类似，在移动主机上进行的更新只有在重新连接后才能传播给其他主机。

我们在第 19 章探讨了一致性问题，也讨论了网络分割。这里我们做进一步的详细讨论。在有线分布式系统中，分割被认为是一种故障模式；在移动计算中，通过断开连接造成的分割是操作的常规模式的一部分。所以即使存在分割，甚至有损失某些一致性的危险，仍然应当允许进行数据访问。

对于只被移动主机更新的数据，在移动主机重新连接时传播其上的更新是一个简单的问题。然而，如果移动主机缓存了可能被其他计算机更新的数据的只读副本，那么缓存的数据就可能变得不一致。当移动主机连接时，可以给它发送**失效报告**(invalidation report)，通知它有过期的缓存数据项。但是，当移动主机断开连接时，它可能会错过失效报告。解决这个问题的一种简单方法是在重新连接时对整个缓存都作失效处理，但这种极端的解决方法开销太大。文献注解中引用了几种缓存方案。

如果更新可以发生在移动主机和其他地方，检测更新冲突就更为困难。基于**版本编号**(version-numbering)的方案允许断开连接的主机对共享文件进行更新。这些方案不能保证更新是一致的。但是，它们保证如果两台主机独立地更新一份文档的同一个版本，那么当主机直接地或者通过公共主机交换 [1083]
信息时这种冲突最终会被检测出来。

当文档的多个副本被独立更新时，**版本向量方案**(version-vector scheme)检测它们之间的不一致性。这种方案允许文档的副本保存在多台主机中。虽然我们使用文档这个术语，这种方案也可以应用到任何其他数据项上，如关系的元组。

基本思路是：每台主机 i 针对其存储文档 d 的副本都存储一个**版本向量**(version vector)，即一个版本号的集合 $[V_{i,i}[j]]$，其中对于每台有可能更新该文档的其他主机 j 都有一项。当主机 i 更新文档 d 时，它将版本号 $V_{i,i}[i]$ 加一。

每当两台主机 i 和 j 相互连接时，它们交换更新的文档，这样它们都可获得文档的新版本。但是，在交换文档前，主机必须检查副本是否一致：

1. 如果两台主机的版本向量相同，即对每个 $k, V_{i,i}[k] = V_{j,j}[k]$，则文档 d 的副本是相同的。

2. 如果对每个 $k, V_{i,i}[k] \leqslant V_{j,j}[k]$，并且版本向量不相同，则文档 d 在主机 i 上的副本比在主机 j 上的副本旧。也就是说，文档 d 在主机 j 上的副本是对主机 i 上的副本进行了一次或多次修改之后得到的。这时，主机 i 用主机 j 上的副本替换其 d 的副本以及 d 的版本向量的副本。

3. 如果存在一对主机 k 和 m，使得 $V_{i,i}[k] > V_{j,j}[k]$ 且 $V_{i,i}[m] < V_{j,j}[m]$，则副本是不一致的；也就是说，主机 i 上的 d 的副本包含主机 k 所做的更新，而这些更新还没有传播到主机 j 上。同样地，d 在 j 上的副本包含主机 m 所做的更新，而这些更新还没有传播到主机 i 上。于是，d 的副本就是不一致的，因为 d 上分别进行了两次或多次更新。这时可能需要人工介入来合并更新。

版本向量方案最初用来处理分布式文件系统的故障。该方案的重要性在于移动计算机常常存储文件的副本，这些副本也出现在服务器系统中。实际上就构成了一个经常会断开连接的分布式文件系统。这一方案的另一个应用是在邮件系统中，其中的主机是周期性地而非持续性地连接，并且必须交换更新后的文档。

版本向量方案在有副本的数据库中也有应用，其中它可以应用于单个元组。例如，如果一份日历或住址名册是在一个移动设备以及一台主机上同时维护的，插入、删除和更新都可以发生在移动设备上，也可以发生在主机上。通过在单个日历项或联系人上应用版本向量方案，可以容易地处理这种情况：一个特定项已在移动设备上更新了，同时另一个不同的项在主机上更新。这种情况不会被认为是 [1084]
冲突。但是，如果同样的项在两个地方独立地更新，通过版本向量方案会检测到冲突。

然而，版本向量方案并不能解决更新共享数据中最困难、最重要的问题：不一致数据副本的一致化。许多应用可以自动执行一致化，这是通过在每台计算机上执行断开连接期间在远端计算机上执行过的更新操作来实现的。该方案在更新操作可交换时有效，也就是说，无论以怎样的顺序执行这些操作，其结果都是相同的。特定应用中有其他一些可用技术；但是，在最差的情况下，必须由用户来解决不一致性。如何自动处理这种不一致性以及如何辅助用户解决不能自动处理的不一致性仍然是一个研究的领域。

版本向量方案的另一个弱点是它要求在重新连接的移动主机和该主机的移动支持站点之间有实质

性通信。一致性检查可以延迟到需要数据时再进行，尽管这种延迟可能增加数据库整体的不一致性。

断开连接的可能性和无线通信的开销限制了第 19 章所讨论的分布式系统的事务处理技术的实用性。通常最好是由用户在移动主机上准备事务，但需要他们将事务提交给服务器执行，而不是在本地执行这些事务。跨越多台计算机并且其中包含移动主机的事务在事务提交过程中会面临长时间的阻塞，除非断开连接的情况极少发生或者是可以预测的。

25.6 总结

- 时间在数据库系统中扮演着重要的角色。数据库是现实世界的模型。尽管大部分数据库只模拟现实世界在一个时间点（在当前时间）上的状态，但时态数据库模拟的是现实世界随时间变化的状态。
- 时态关系中的事实与当它们有效时的时间相关联，而时间可以用时段的并来表示。时态查询语言简化了时间建模以及与时间相关的查询。
- 目前空间数据库正越来越多地用于存储计算机辅助设计数据和地理数据。
- 设计数据主要以矢量数据的形式存储；地理数据包含矢量数据和光栅数据的组合。空间完整性约束对于设计数据十分重要。
- 矢量数据可以编码成为第一范式数据，或者用非第一范式结构来存储，如列表。专用索引结构对于访问空间数据和处理空间查询尤为重要。
- R 树是 B 树的多维扩展；它和它的变体（如 R⁺ 树和 R* 树）在空间数据库中得到了广泛的应用。将空间以某种固定方式进行划分的索引结构（如四叉树）有助于处理空间连接查询。
- 多媒体数据库正变得越来越重要。诸如基于相似性的检索以及按可以确保的速率传输数据那样的问题是当前研究的重要课题。
- 移动计算系统的普及使得人们对在这类系统上运行的数据库系统产生了兴趣。在这类系统中的查询处理可能会涉及在服务器端数据库上的查找。查询代价模型必须包括通信代价，其中包括金钱上的成本和电池电源的成本，它们对于移动系统来说是相对较高的。
- 广播方式比点对点通信在向接收者上的成本要便宜得多，诸如股市之类数据的广播有助于移动系统低成本地接收数据。
- 连接断开操作、广播数据的使用和数据缓存是移动计算中正致力解决的三个重要问题。

1085

术语回顾

- 时态数据
- 有效时间
- 事务时间
- 时态关系
- 双时态关系
- 全球协调时间（UTC）
- 快照关系
- 时态查询语言
- 时态选择
- 时态投影
- 时态连接
- 空间和地理数据
- 计算机辅助设计（CAD）数据
- 地理数据
- 地理信息系统
- 三角剖分
- 设计数据库
- 地理数据
- 光栅数据
- 矢量数据
- 全球定位系统（GPS）
- 空间查询
- 临近查询
- 最近邻居查询
- 区域查询
- 空间连接
- 空间数据索引
- k-d 树
- k-d-B 树
- 四叉树
  - PR 四叉树
  - 区域四叉树
- R 树
  - 边界框
  - 二次方分裂
- 多媒体数据库
- 等时数据
- 连续媒体数据
- 基于相似性的检索
- 多媒体数据格式
- 视频服务器
- 移动计算
  - 移动主机
  - 移动支持站点
  - 单元
  - 交接
- 位置相关查询
- 广播数据
- 一致性
  - 失效报告
  - 版本向量方案

实践习题

25.1 时间有哪两种类型？怎样区分它们？为什么将一个元组与这两类时间联系起来是有意义的？

25.2 假设有一个关系包含 x，y 坐标和餐馆名。并假设查询只能是如下形式：查询指定一个点，问是否恰好有一家餐馆在这个点上。R 树或 B 树哪一种索引更好？为什么？

25.3 假设有一个空间数据库支持区域查询（圆形区域），但不支持最近邻居查询。描述一个算法，通过利用多个区域查询来找到最近的邻居。

25.4 假设要在 R 树中存储线段。如果线段与坐标轴不平行，它的边界框会很大，并包含大量的空白区域。

- 请说明在查询与一个给定区域相交的线段时，大边界框对性能的影响。
- 简要描述一种能提高此类查询性能的技术，并给出体现其优点的例子。提示：可以把线段分为更小的部分。

25.5 请给出一个使用 R 树索引来有效计算两个关系的空间连接的递归过程。（提示：使用边界框检查一对内部结点之下的叶结点项是否相交。）

25.6 描述 RAID 怎样组织（10.3 节）中的思想是如何应用于数据广播环境的，该环境中有时会有噪音阻止部分传输数据的接收。

25.7 定义一个重复广播数据的模型，其中广播介质建模为一个虚拟磁盘。描述这个虚拟磁盘的存取时间和数据传输率与一个普通硬盘的对应值的差异。 1086 ↓ 1087

25.8 考虑一个将所有文档保存在中央数据库的文档数据库。有些文档的副本存储在移动计算机上。假设移动计算机 A 在断开连接期间更新了文档 1 的副本，同时，移动计算机 B 在断开连接期间更新了文档 2 的副本。请说明在移动计算机重新连接时，版本向量方案如何保证对中央数据库和移动计算机的正确更新。

习题

25.9 如果通过增加一个时间属性来将关系转换为时态关系，函数依赖能够保持吗？这一问题在时态数据库中是怎样处理的？

25.10 考虑二维矢量数据，其中的数据项互不重叠。是否可以把这些矢量数据转换为光栅数据？如果可以，存储经过这种变换得到的光栅数据而非原始矢量数据有什么缺点？

25.11 研究你使用的数据库系统所提供的空间数据支持，实现以下内容：

a. 一个模式，描述饭店的地理位置和其他特征，如饭店提供的烹调风格和消费级别。

b. 一个查询，寻找中等价位的提供印度食物，并且距离你的房子（为你的房子假设一个任意的位置）5 英里以内的饭店。

c. 一个查询，为每个饭店寻找其与最近的一个具有相同的烹调风格和消费级别的饭店之间的距离。

25.12 在连续媒体系统中，如果数据传送过慢或过快会导致什么问题？

25.13 列出无线网络上的移动计算与传统分布式系统相区别的三个主要特性。

25.14 列出传统查询优化器没有考虑，但在移动计算的查询优化中需要考虑的三个因素。

25.15 给出版本向量方案无法保证可串行性的例子。（提示：使用实践习题 25.8 中的例子，假设文档 1 和文档 2 在移动计算机 A 和 B 上都可用，并考虑文档读出时没有被更新的可能性。） 1088

文献注解

Stam 和 Snodgrass［1988］和 Soo［1991］提供了关于时态数据管理的概览。Jensen 等［1994］给出了时态数据库概念的一个术语表，旨在统一术语。Tansel 等［1993］是有关时态数据库不同方面文章的一个汇集。Chomicki［1995］给出了管理时态完整性约束的技术。

Heywood 等［2002］是讲述地理信息系统的教科书。Samet［1995b］是关于空间索引结构上大量工作的一个综述。Samet［1990］和 Samet［2006］是讲述空间数据结构的教科书。Finkel 和 Bentley［1974］中有对四叉树的早期描述。Samet［1990］和 Samet［1995b］描述了四叉树的各种变体。Bentley［1975］描述了 k-d 树，Robinson［1981］描述了 k-d-B 树。Guttman［1984］最先提出了 R 树。R 树的各种扩展由 Sellis 等［1987］作了阐述，其

中描述了 R* 树。Beckmann 等[1990]描述了 R* 树。

Brinkhoff 等[1993]讨论了使用 R 树来实现空间连接。Lo 和 Ravishankar[1996]以及 Patel 和 DeWitt[1996]介绍了计算空间连接的基于划分的方法。Samet 和 Aref[1995]综述了空间数据模型、空间运算以及空间与非空间数据的集成。

Revesz[2002]是讲述约束数据库方面领域的教科书；时间间隔和空间区域可以看作是约束的特殊情况。

Samet[1995a]阐述了多媒体数据库中的研究问题。Faloutsos 和 Lin[1995]讨论了多媒体数据的索引。

Dashti 等[2003]是讲述流媒体服务器设计的教科书，包括对磁盘子系统上数据组织的详细介绍。Anderson 等[1992]、Rangan 等[1992]、Ozden 等[1994]、Freedman 和 DeWitt[1995]以及 Ozden 等[1996b]讨论了视频服务器。Berson 等[1995]和 Ozden 等[1996a]讨论了容错性。

1089
/
1090

Alonso 和 Korth[1993]以及 Imielinski 和 Badrinath[1994]研究了包括移动计算机的系统中的信息管理。Imielinski 和 Korth[1996]介绍了移动计算及关于该主题的一系列研究论文。

Popek 等[1981]以及 Parker 等[1983]阐述了用来检测分布式文件系统中的不一致性的版本向量方案。

第26章
Database System Concepts, 6E

高级事务处理

在第14章、第15章和第16章中，我们介绍了事务的概念，事务是一个程序的单位，它对各种数据项进行访问或更新，它的执行确保具有ACID特性。在那几章中，我们讨论了在可能发生故障以及可能并发运行多个事务的环境中，为保证ACID特性所采用的各种技术。

在本章中，我们比以前讨论过的基本机制更进一步，介绍高级的事务处理概念。包括事务处理监控器、事务工作流以及在电子商务环境中的事务处理。我们还涵盖主存数据库、实时数据库、长事务以及嵌套事务。

26.1 事务处理监控器

在20世纪70年代和80年代开发的**事务处理监控器**(TP monitor)系统，早期是为了响应单台计算机支持大量远程终端(例如机票预订终端)的需求。术语TP监控器原来代表远程处理监控器(teleprocessing monitor)。

TP监控器后来演化为提供对分布式事务处理的核心支持，于是TP监控器这个术语获得了它现在的含义。IBM公司的CICS TP监控器是最早的TP监控器之一，应用非常广泛。其他TP监控器包括Oracle Tuxedo和Microsoft Transaction Server。

Web应用服务器体系结构包括之前我们在9.3节学习的servlets，它支持TP监控器的许多特性，有时候被称为"TP lite"。Web应用服务器现在正被广泛使用，已经在许多应用中替代了传统TP监控器。然而，我们在本节所学习的它们的基本概念在本质上是一样的。 1091

26.1.1 TP监控器体系结构

大规模的事务处理系统建立在客户-服务器体系结构之上。构建这种系统的一种方式是为每个客户端分配一个服务器进程。服务器进行认证，然后执行客户端请求的动作。图26-1a描述了这种**每客户端进程模型**(process-per-client model)。这种模型在内存利用和处理速度方面存在几个问题：

- 每个进程的内存需求量很大。即使程序代码所用的内存是由所有进程共享的，每个进程还要耗费用于本地数据和打开文件描述符的内存，以及用于操作系统的开销，如支持虚拟内存的页表。
- 操作系统通过在进程间的切换来划分可利用的CPU时间，这种技术称作**多任务调度**(multitasking)。在一个进程和下一个进程之间的每次**上下文切换**(context switch)都需要相当大的CPU开销，即使在目前相当快的系统上，一次上下文切换也要花数百微秒的时间。

通过使用与所有远程客户端都连接的单服务进程可以避免上述问题。这种模式称作**单服务器模型**(single-server model)，如图26-1b所示。远程客户端将请求发送到服务器进程，然后由服务器进程执行这些请求。这种模型也用于客户-服务器环境中，客户端将请求发送到单服务器进程。服务器进程处理通常由操作系统执行的任务，例如用户认证。为了避免在处理一个客户端的长时间请求时阻塞其他客户端，服务器进程是**多线程**(multithread)的：服务器进程对每个客户端有一个线程来控制，并且事实上是在实现它自己的低开销多任务调度。它为一个客户端执行一段时间的代码，然后保存内部的上下文，并切换到另一个客户端的代码上。与完全多任务调度不同，在线程之间切换的代价很小(通常仅要几微秒)。

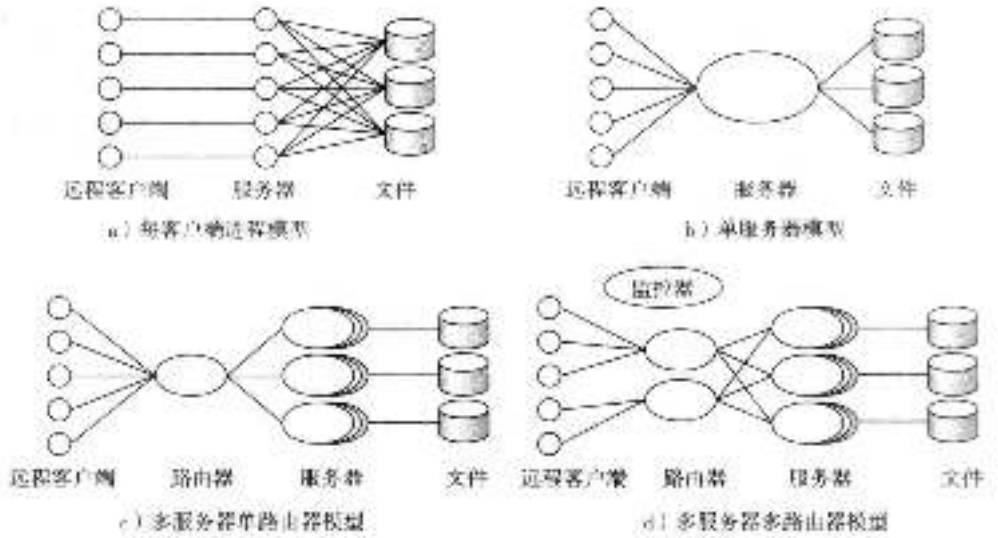


图 26-1 TP 监控器体系结构

基于单服务器模型的系统，例如 IBM CICS TP 监控器的原始版本，以及像 Novel 的 NetWare 这样的文件服务器，在资源有限的情况下成功地提供了较高的事务率。然而，它们存在一些问题，尤其是当多个应用访问同一数据库时：

- 由于所有应用作为单个进程运行，它们之间没有保护。一个应用中的错误会影响到所有其他应用。最好是每个应用作为一个单独的进程运行。
- 这样的系统不适合于并行数据库或分布式数据库，因为一个服务器进程不能同时在多台计算机上执行。（然而，在共享内存多处理器系统中可以支持一个进程中有多个并发线程。）在大型组织机构中这是一个严重的缺陷，因为其中的并行处理对处理大型负载十分关键，而且分布式数据正变得越来越普遍。

解决这些问题的一个办法是运行多个访问公共数据库的应用服务器进程，并且让客户端通过一个单独的路由请求的通信进程与应用通信。这种模型称作**多服务器单路由器模型**（many-server, single-router model）。如图 26-1c 所示。这种模型为多个应用提供独立的服务器进程；而且，每个应用可以有一个服务器进程池，其中任何一个进程都可以处理客户端会话。例如，请求可以路由到进程池中负载最轻的服务器上。如前所述，每个服务器进程本身可以是多线程的，因此它可以并发地处理多个客户端。作为更进一步的推广，应用服务器可以在并行数据库或分布式数据库的不同站点上运行，并且通信进程可以在这些进程间进行协同。

上述体系结构也广泛应用于 Web 服务器。Web 服务器有一个接收 HTTP 请求的主进程，然后将处理各个请求的任务分配给单独的进程（从进程池中选择）。每个进程本身是多线程的，以便处理多个请求。使用安全的编程语言，例如 Java、C#或 Visual Basic，允许 Web 应用服务器能够保护线程免受其他线程错误的影响。相比之下，如果使用 C 或 C++ 这样的语言，一个线程中像内存分配错误这样的错误可能引起其他线程的失败。

1092
↓
1093

一种更通用的体系结构是有多个进程而不仅是一个进程，来与客户端通信。客户端通信进程与一个或多个路由进程交互，多个路由将请求路由给恰当的服务器。因此新一代 TP 监控器具有一种不同的体系结构，称作**多服务器多路由器模型**（many-server, many-router model），如图 26-1d 所示。一个控制进程启动其他进程并监控它们的工作。超高性能的 Web 服务器系统也采用这种体系结构。路由进程通常是网络路由器，它们根据通信量的来源将到达同一互联网地址的通信量分流到不同的服务器计算机上。外部世界看似具有单一地址的单台服务器可能是服务器的集合。

TP 监控器的详细结构如图 26-2 所示。TP 监控器不仅仅是简单地将消息传送给应用服务器。当消息到达时，必须将它们放入到队列中；因此，有一个**队列管理器**（queue manager）来管理到达的消息。

队列可以是一个**持久队列**(durable queue)，即使在系统故障时其中的项也会保留下来。使用持久队列有助于确保消息一旦收到并保存在队列中，无论系统是否发生故障，消息最终都会得到处理。授权和应用服务器管理(例如，服务器启动和路由到服务器的消息)是 TP 监控器的进一步功能。TP 监控器通常提供日志、恢复和并发控制功能，允许应用服务器在需要时直接实现事务的 ACID 特性。

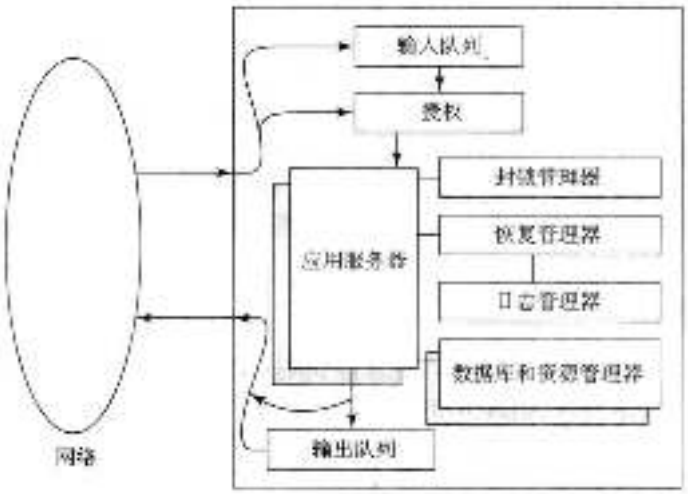


图 26-2　TP 监控器构成成分

最后，TP 监控器还提供对持久消息的支持。回想一下，持久消息(19.4.3 节)提供了一个保证：当(且仅当)事务提交，消息才会被发送。

除了这些功能之外，许多 TP 监控器还为像终端这样的哑客户端提供了表示功能(presentation facility)，以创建菜单或表格界面；由于哑客户端不再被广泛使用，这样的功能就不再重要了。

26.1.2　使用 TP 监控器进行应用协调

当前的应用经常要与多个数据库交互。它们还可能与遗产系统进行交互，例如直接建立在文件系统上的专用数据存储系统。最后，它们可能与远程站点上的用户或其他应用通信。因此，它们还必须与通信子系统交互。对于跨越在这些系统上的事务，能够对数据访问进行协调，并实现 ACID 特性是很重要的。

现代 TP 监控器为构建和管理这种由多个子系统(例如数据库、遗产系统和通信系统)构成的大型应用提供了支持。TP 监控器将每个子系统作为一个**资源管理器**(resource manager)，它提供对某些资源集的事务性访问。TP 监控器与资源管理器之间的接口由一组事务原语来定义，例如 *begin_transaction*、*commit_transaction*、*abort_transaction* 和 *prepare_to_commit_transaction*(用于两阶段提交)。当然，资源管理器还必须为应用提供其他服务，如提供数据。

资源管理器接口由 X/Open 分布式事务处理标准定义。许多数据库系统支持 X/Open 标准，并且可以充当资源管理器。TP 监控器(以及其他产品，例如支持 X/Open 标准的 SQL 系统)可以与资源管理器连接。

另外，TP 监控器还提供诸如持久消息和持久队列服务，起到支持事务的资源管理器的作用。TP 监控器可以担当访问这些服务和数据库系统的事务的两阶段提交的协调者。例如，当排到队列中的一个更新事务被执行时，会发送一条输出消息，然后请求事务从请求队列中删除。对于持久队列和持久消息，数据库与资源管理器之间的两阶段提交有助于确保无论是否出现故障，所有这些动作要么都发生，要么都不发生。

我们还可以使用 TP 监控器来管理包括多个服务器和大量客户端的复杂的客户 - 服务器系统。TP 监控器协调诸如系统检查点和系统关闭那样的活动。它提供安全性和客户端认证。它管理服务器缓冲池，使得不用中断数据库系统就能增加或删除服务器。最后，它控制故障的范围。如果一台服务器发生了故障，TP 监控器能检测到这一故障，中止正在进行的事务，并重新启动事务。如果一个结点发生

了故障，TP 监控器能够将事务移到其他结点的服务器上去，并重新恢复未完成的事务。当故障结点重新启动时，TP 监控器能够控制结点上的资源管理器的恢复。

在复制系统中，可以用 TP 监控器来隐藏数据库故障：远程备份系统（见 16.9 节）就是复制系统的一个例子。事务请求发送到 TP 监控器，TP 监控器再将消息传播给某个数据库副本（如果在远程备份系统中，就是主站点）。如果某个站点发生故障，TP 监控器可以透明地将消息路由到一个备份站点，以隐藏第一个站点的故障。

1094
1095

在客户－服务器系统中，客户端通常通过**远程过程调用**（Remote-Procedure-Call，RPC）机制与服务器交互。在 RPC 机制中客户端调用实际上在服务器端执行的过程，将结果返回给客户端。从调用 RPC 的客户端代码来看，该调用看起来就和调用本地过程一样。TP 监控器系统为它们的服务提供了一个**事务 RPC**（transactional RPC）接口。在这样的接口中，RPC 机制提供一些调用，用于将一系列 RPC 调用包括在一个事务中。这样，由一个 RPC 执行的所有更新都是在该事务的范围内执行的，如果发生故障的话可以回滚。

26.2 事务工作流

工作流（workflow）是一种活动，其中多个任务由不同处理实体以互相协同的方式执行。**任务**（task）定义了要做的某项工作，并且可以用多种方式来说明，包括文件中的文本描述、电子邮件消息、表单、消息或计算机程序。执行任务的**处理实体**（processing entity）可以是人或软件系统（例如，邮件发送器、应用程序或数据库管理系统）。

图 26-3 显示了工作流的一些例子。电子邮件系统就是工作流的一个简单例子。一条邮件消息的发送可能涉及几个邮件系统，它们接收和转发邮件消息，直至消息到达存放它的目的地。在数据库和相关文献中用来指代工作流的其他术语包括**任务流**（task flow）和**多系统应用**（multisystem application）。工作流任务有时也称为**步骤**（step）。

工作流应用	典型任务	典型处理实体
电子邮件路由	电子邮件消息	邮件发送器
借贷处理	表格处理	人、应用软件
订单处理	表格处理	人、应用软件、数据库管理系统

图 26-3 工作流示例

一般说来，工作流可能涉及一个或多个人。例如，考虑借贷的处理过程，相关的工作流如图 26-4 所示。想要贷款的人填写表格，然后由借贷工作人员检查这个表格。处理借贷申请的员工使用诸如信用征明局那样的资源来查证表格中的数据。当需要的所有信息收集完后，借贷工作人员可能决定批准这笔贷款，这一决定随后也许需要经一个或多个上级工作人员批准。之后才可以进行贷款。这里每个人执行一个任务，在借贷处理任务尚未自动化的银行里，任务间的协同一般是通过从一个工作人员向下一个工作人员传递借贷申请（附上票据和其他信息）来完成的。工作流的其他例子包括开支凭单处理、订单处理、信用卡事务处理。

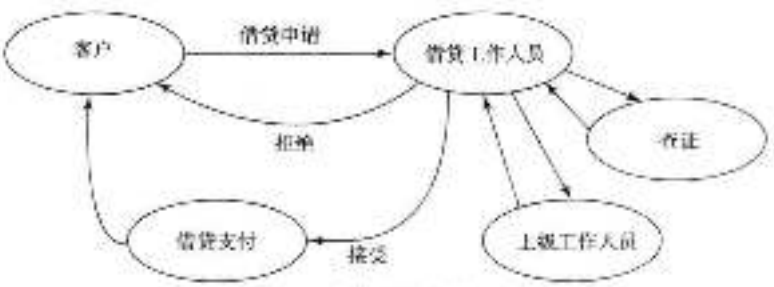


图 26-4 借贷处理中的工作流

当今，有关工作流的所有信息大都以数字化形式存储在一台或多台计算机中，而且，随着网络的发展，信息可以很容易地从一台计算机传送到另一台计算机。因此，组织机构工作流的自动化是可行的。例如，为了使借贷处理所涉及的任务自动化，我们可以把借贷申请和相关信息存储在数据库中。这样工作流本身就需要将责任从一个人转交给下一个人，甚至可能转交给能自动获取所需信息的程序。人员之间可以通过电子邮件之类的途径来协调他们的活动。

工作流在机构中以及机构之间变得越来越重要，其原因是多方面的。当今许多组织机构拥有多种软件系统，它们必须能够协同工作。例如，当一名员工加入一个机构时，有关这名员工的信息可能要提交到薪酬系统、图书馆系统、允许用户登录到计算机的认证系统、管理自助餐厅账户的系统，等等。信息的更新，例如当员工改变了身份或者离开了机构，也需要传播到所有的系统。

组织机构正令它们的服务日益自动化；例如，供货商可以为客户端提供自动化的系统以提交订单。当一份订单提交时也许需要执行几个任务，包括预留生产所订购产品的生产时间和递送该产品的递送服务时间。

为使工作流自动化，我们一般需要强调两个活动。第一个是**工作流说明**(workflow specification)：详细描述必须执行的任务并定义执行需求。第二个问题是**工作流执行**(workflow execution)，在执行工作流的同时必须提供与计算正确性、数据完整性和持久性相关的传统数据库系统安全措施。例如，由于系统崩溃造成的借贷申请或凭单丢失，或者多次处理都是不可接受的。事务工作流背后的思想是：使用事务的概念并将它扩展到工作流环境中。 1096 | 1097

这两个活动都很复杂，其原因是许多组织机构都使用多个独立管理的信息处理系统，在大多数情况下，这些系统是分别开发以对不同功能进行自动化的。工作流活动可能需要在几个这样的系统之间进行交互，每个活动执行一个任务，并且与人进行交互。

近年来人们已经开发出多个工作流系统。这里我们在一个相对抽象的层次上研究工作流系统的特性，而不深入到任何一个特定系统的细节。

26.2.1　工作流说明

任务的内部方面无需为了工作流的说明和管理的目的而建模。从任务的抽象视角来看，任务可以使用存放在它的输入变量中的参数，可以检索和更新本地系统中的数据，可以将它的结果存放到它的输出变量中，可以查询出其执行状态。在执行过程中的任何时间，**工作流状态**(workflow state)包括构成该工作流各个任务的状态的集合，以及工作流说明中所有变量的状态(值)。

任务之间的协调既可以静态地声明，也可以动态地声明。静态声明在工作流开始执行之前定义任务以及任务之间的依赖关系。例如，开支凭单工作流中的任务可能包括秘书、经理、会计按顺序对凭单的认可，以及最后支票的交付。任务之间的依赖关系可能很简单：每个任务必须在下一个任务开始之前完成。

该策略的推广是，对于工作流中每个任务的执行都有一个前提条件，这样工作流中所有可能的任务以及它们之间的依赖关系都可以预先知道，但是只有那些前提条件满足的任务才会被执行。前提条件可以通过如下的依赖关系来定义：

- 其他任务的**执行状态**(execution state)。例如，"任务 t_1 结束了，任务 t_2 才能开始"或者"如果任务 t_3 提交，则任务 t_4 必须中止"。
- 其他任务的**输出值**(output value)。例如，"如果任务 t_5 返回一个大于 25 的值，则任务 t_6 可以开始"或者"如果秘书审批任务返回 OK 值，则经理审批任务可以开始"。
- 由外部事件修改的**外部变量**(external variable)。例如，"上午 9 点以后任务 t_7 才能开始"或者"在任务 t_8 完成后的 24 小时内必须开始任务 t_9"。 1098

我们可以用通常的逻辑连接符(**or**、**and**、**not**)来将多个依赖关系组合在一起，形成复杂的调度前提条件。

任务动态调度的一个例子是电子邮件路由系统。对于给定的邮件消息要安排的下一个任务依赖于消息的目的地址是什么，以及哪些中间路由器在运行中。

26.2.2 工作流的故障原子性需求

工作流设计者可以规约工作流语义来声明工作流的**故障原子性**(failure-atomicity)需求。传统故障原子性概念会要求当任何任务发生故障时就导致整个工作流的失败。但是在许多情况下，当其中一个任务发生故障时工作流不一定失败，例如，可以通过在另一个站点执行一个功能上等价的任务来使工作流完成。因此，我们应该允许设计者定义工作流的故障原子性需求。系统必须保证工作流的每次执行都以一种满足设计者定义的故障原子性需求的状态终止。我们将这些状态称作工作流的**可接受终止状态**(acceptable termination state)。工作流的所有其他执行状态构成**不可接受终止状态**(nonacceptable termination state)的集合。这些状态可能违背故障原子性需求。

可接受终止状态可以指定为提交的或中止的。**提交的可接受终止状态**(committed acceptable termination state)是工作流达到了目标的执行状态。相反，**中止的可接受终止状态**(aborted acceptable termination state)是合法的终止状态，此时工作流未能达到它的目标。如果达到了一个中止的可接受终止状态，那么工作流的部分执行产生的所有不合要求的影响都应该按照该工作流的故障原子性需求予以撤销。

即使在发生系统故障的情况下，工作流也必须达到一个可接受终止状态。因此，如果当发生故障时工作流处于一个不可接受的终止状态，那么在系统恢复中必须使它进入一个可接受终止状态(中止或提交皆可)。

例如，借贷处理工作流中，在最终状态，要么当该贷款申请人根本不能获得贷款，要么支付贷款。在发生故障的情况下，比如验证系统长时间故障，可以用适当的解释将贷款申请退回给贷款申请人。这种结局构成一种中止的可接受终止。提交的可接受终止应该要么是批准贷款，要么是拒绝贷款。

一般说来，在工作流达到终止状态之前，任务可以提交并释放其所占资源。然而，如果这个多任务事务后来中止了，其故障原子性可能要求通过执行补偿任务(作为子事务)来撤销已完成任务(例如，已提交的子事务)所造成的影响。补偿的语义要求补偿事务最终成功地完成它的执行，也许需要若干次重新提交。

例如，在开支凭单处理工作流中，由于开始时经理批准了一张凭单，所以部门预算的余额减少了。
1099 如果后来由于故障或由于其他原因，这张凭单被拒绝了，那就必须通过补偿事务来恢复预算。

26.2.3 工作流执行

多个任务的执行可以由一个协调人来控制，或者由一个称作**工作流管理系统**(workflow-management system)的软件系统来控制。工作流管理系统包括一个调度器、多个任务代理和一个查询工作流系统状态的机制。任务代理通过处理实体控制任务的执行。调度器是一个对工作流进行处理的程序，它将不同的任务交付执行，监控各种事件，计算与任务间依赖有关的条件。调度器可以将一个任务交付(给任务代理)执行，也可以要求中止一个先前交付执行的任务。在多数据库事务的情况下，任务是子事务，处理实体是本地的数据库管理系统。调度器依据工作流说明，依加调度依赖并监督确保任务到达可接受的终止状态。

开发工作流管理系统有一种体系架构方法。**集中式体系结构**(centralized architecture)用单个调度器去调度所有并发执行工作流的任务。**部分分布式体系结构**(partially distributed architecture)对每个工作流实例化一个调度器。当并发执行问题可以从调度功能中分离出来时，后一种是很自然的选择。完全**分布式体系结构**(fully distributed architecture)没有调度器，而由各个任务代理通过相互之间的通信来协调它们的执行，以满足任务间的依赖关系和其他工作流执行的需求。

最简单的工作流执行系统采用前述的完全分布式方式，并且是基于消息的。消息可以由持久消息机制来实现，以提供有保证的传递。有些实现使用电子邮件进行消息发送，这种实现提供持久消息的很多特点，但通常不保证消息发送和事务提交的原子性。每个站点有一个任务代理来执行通过消息接收到的任务。执行中也可能会将消息交给人，由人随后去执行其他动作。当任务在一个站点上完成并需要到另一个站点去处理时，任务代理发消息给下一个站点。消息中包括关于待执行任务的所有相关信息。这种基于消息的工作流系统对于在部分时间内可能会断开连接的网络特别有用。

集中式方式适用于数据存储在中央数据库中的工作流系统。调度器通知各种代理(例如人或计算机程序)有任务需要执行，并且跟踪任务的完成情况。在集中式方式下跟踪工作流的状态比在完全分布式方式下要更容易。

调度器必须保证工作流终止于一个特定的可接受终止状态。理想情况是，在打算执行工作流之前，调度器应该检查该工作流，看它是否可能终止于一个不可接受的状态。如果调度器不能保证工作流将终止于一个可接受状态，它应该拒绝这样的工作流说明，而不应该试图去执行它。作为一个例子，我 1100
们考虑一个工作流，它包括两个任务，表示为子事务 S_1 和 S_2，其故障原子性需求指明要么两个子事务都提交，要么一个也不提交。如果 S_1 和 S_2 没有提供(两阶段提交的)准备提交状态，而且也没有补偿事务，那么就有可能到达一个子事务提交而另一个中止的状态，而且没有办法使两个子事务到达同一状态。因此，这样的工作流说明是不安全的(unsafe)，应该拒绝它。

在调度器中实现诸如上述的安全性检查可能是不可能的或者不现实的，这样保证工作流的安全性就变成了工作流说明的设计者的责任。

26.2.4 工作流恢复

工作流恢复(workflow recovery)的目标是保证工作流的故障原子性。恢复过程必须保证，如果任何一个工作流处理构件(包括调度器)发生故障，工作流最终都能达到一个可接受的终止状态(中止的或者提交的)。例如，在故障和恢复后调度器可以继续处理，就像什么事也没发生一样，从而提供向前的可恢复性。否则，调度器可以中止整个工作流(即达到一个全局中止状态)。无论在哪种情况下，都可能需要提交某些子事务或者甚至将某些子事务交付执行(例如，补偿子事务)。

我们假设工作流中涉及的处理实体有它们自己的恢复系统，并能处理它们的本地故障。为了恢复执行环境的上下文，故障恢复例程需要恢复调度器在发生故障时的状态信息，包括关于每个任务执行状态的信息。因此，必须将适当的状态信息作为日志记录到稳定存储器中。

我们还需要考虑消息队列的内容。当一个代理将任务交给另一个代理时，应该恰好交接一次。如果交接了两次，那么一个任务可能被执行了两次；如果没有交接，那么任务就丢失了。持久消息(19.4.3节)恰好提供了保证确实是单次交接的特性。

26.2.5 工作流管理系统

作为应用系统的一部分，工作流常常是手工编码的。例如，企业资源规划(ERP)系统，用于帮助协调整个企业内部的活动，其内部就嵌有许多工作流。

工作流管理系统的目标是简化工作流构造并使得它们更加可靠，这些通过允许工作流以高层的形式说明并按照相应说明执行来实现。有大量商用的工作流系统，其中一些是通用的工作流管理系统，而另一些则用于特殊的工作流，如订单处理或错误/失败报告系统。 1101

当今是组织机构间互联的世界，只在一个组织机构内部管理工作流是不够的。跨越组织边界的工作流变得越来越普遍。例如，考虑由某个组织下达订单，并与另一个补充该订单的组织通信。在每个组织中都有工作流与此订单相联系，为了最小化人工干预，工作流能协同工作是很重要的。

术语**业务流程管理**(business process management)用来指理业务流程相关的工作流管理。如今，应用越来越多地把它们的功能变成可以被其他应用调用的服务，通常使用 Web 服务体系结构。基于调用由多个应用提供的服务的系统体系结构称为**面向服务的体系结构**(Service Oriented Architecture，SOA)。当今的工作流管理是在这些服务的基层之上实现的。通过调用服务来控制工作流的处理逻辑称为**编排**(orchestration)。

基于 SOA 体系结构的业务流程管理系统包括 Microsoft 的 BizTalk Server，IBM 的 WebSphere Business Integration Server Foundation 和 BEA 的 WebLogic Process Edition，等等。

Web 服务业务流程执行语言(Web Services Business Process Execution Language，WS-BPEL)是基于 XML 的标准，用于指定 Web 服务和基于 Web 服务的业务流程(工作流)，可以通过业务流程管理系统执行。**业务流程建模标记法**(Business Process Modeling Notation，BPMN)是为工作流中的业务流程进行图形化建模的标准。**XML 流程定义语言**(XML Process Definition Language，XPDL)是业务流程定义的基

于 XML 的表示，它基于 BPMN 图。

26.3 电子商务

电子商务指通过电子手段（主要是通过因特网）执行与商务相关的各种活动的过程。活动类型包括：

- 预售活动，需要将待出售的产品或服务告知潜在买家。
- 销售过程，包括价格和服务质量的协商，以及其他合同事项。
- 市场：当同一个产品对应多个买家和卖家时，市场（如股票交易市场）有助于协商产品的销售价格。当只有一个卖家和多个买家时，可以举行拍卖；而只有一个买家和多个卖家时，可以举行反向拍卖。
- 销售付款。
- 交付产品或服务的相关活动。一些产品和服务可以通过因特网交付；对于其他的，因特网只用于提供运送信息和跟踪产品的运送。
- 客户支持和售后服务。

[1102]

数据库广泛应用于支持这些活动。对于有些活动，数据库的使用是直接的，但对于其他活动还存在有趣的应用开发方面的问题。

26.3.1 电子目录

任何电子商务站点都给用户提供该站点所供应的产品和服务目录。电子目录提供的服务可以有很大差别。

在最低限度上，电子目录必须提供浏览和搜索功能，以帮助客户发现他们寻找的产品。为了有助于浏览，产品应该组织成直观的层次结构，这样只需要点击很少次的超链接就可以引导客户找到他们感兴趣的产品。客户提供的关键词（例如，“数码相机”或者“计算机”）应能够加速寻找所需产品的过程。电子目录还应该提供这样的手段，使得客户能够容易地比较从竞争产品中进行的不同选择。

电子目录可以为客户定制。比如，一个零售商可能同意向一家大公司打折供应一些产品。这家公司的职员在为公司购买产品而浏览目录时，应该看见的价格单价是协商好的打折价格，而不是普通价格。由于法律对销售某些类型的商品有限制，未成年客户或者来自特定州或国家的客户，就不应该看到法律不允许卖给他们的商品。对于个人用户，目录还能够根据以前的购买历史进行个性化。例如，常客可以在某些商品上得到特殊的折扣。

支持这样的定制，需要在数据库中存储客户信息，以及特殊的价格/折扣信息和销售限制信息。在支持很高的事务处理率方面还有一些挑战，通常用对查询结果或所生成的 Web 页面进行高速缓存的方式来处理。

26.3.2 市场

当同一个产品有多个卖家或多个买家（或两者都有）时，市场有助于协商产品的销售价格。存在几种不同类型的市场：

- 在**反向拍卖**（reverse auction）系统中，买家提出要求，卖家们为了供应商品进行投标。提出最低价格的供应商获胜。在封闭投标系统中，出价是不公开的，而在公开投标系统中，出价是公开的。

[1103]

- 在**拍卖**（auction）中，有多个买家和一个卖家。为简单起见，假设每种待售商品只有一件实物。买家为待售商品进行投标。为商品出价最高的买家可以用投标价格买下该商品。

  当存在同一商品的多个实物时，情况变得更加复杂：假设有四件商品，一个投标者愿意用每个 10 美元的价钱买下 3 个，而另一个愿意用每个 13 美元的价钱买下 2 个。显然不可能同时满足两个投标者。如果商品没有售出就没有任何价值（如航班座位，必须在飞机起飞前卖出），卖家就简单地选出一个能获得最大收益的出标集合。否则决策就更为复杂。
- 在**交易**（exchange）中，如股票交易，存在多个卖家和多个买家。买家提出他愿意支付的最高价

钱，卖家提出他期望的最低价钱。经常存在一个做市商(market maker)，由他来匹配买家和卖家的投标，决定每次交易的价格(如按卖家的出标价格)。

还有其他一些类型更为复杂的市场。

在处理市场类型中遇到的数据库方面的问题有：

- 投标者在被允许投标之前需要验证身份。
- 出价(买家的或卖家的)需要安全地记录在数据库中。出价需要很快通知到市场中所涉及的其他人(例如所有的买家和卖家)，人数可能很多。
- 广播出价的延误可能导致某些参与者的财政损失。
- 在股票市场波动或接近拍卖结束时，交易量可能会非常巨大。因此，这种系统使用高度并行的、性能非常高的数据库。

26.3.3 订单结算

在选好商品(可能通过电子目录)并确定价格(可能通过电子市场)之后，就需要结算订单了。结算包括商品支付和商品运输。

一种简单的但是不安全的电子支付方式就是发送信用卡号。这里存在两个主要问题。首先，可能会有信用卡欺诈。当买家支付实物商品时，公司要确保运送地址和持卡人地址相匹配。这样其他的人不能收到该商品。但是以电子方式发送商品时不可能进行这种检查。其次，必须信任卖家只对达成协议的商品进行结账，并且不会将卡号发送给其他非授权的人滥用。

有几种协议可用于避免如上所列出的两个问题的安全支付。另外，这些协议提供了更好的私密性，据此卖家不会得到有关买家的任何不必要的细节，且信用卡公司不会得到有关所购商品的任何不必要的信息。所有传送的信息必须加密，这样任何在网络上拦截数据的人无法发现其中的内容。公/私钥加密广泛应用于这项任务中。 [1104]

这些协议还必须能防止**中间人攻击**(person-in-the-middle attack)，即有人冒充银行或信用卡公司，甚至卖家或买家，偷取秘密信息。假冒可以通过传递一个伪造的密钥，作为其他人的公钥(银行的、信用卡公司的、商家或买家的)。**数字证书**(digital certificate)系统可以防止假冒行为，据此公钥由一个认证机构签署，认证机构的公钥是公认的(或者它的公钥由另一所认证机构认证，以此类推，直到找到一个公认的公钥为止)。从公认的公钥出发，系统可以通过逆序检查证书的方式对其他密钥授权。前面在9.8.3.2节已经对数字证书进行了描述。

几个新的支付系统是在Web时代的初期开发的。其中之一是称为安全电子交易(Secure Electronic Transaction，SET)协议的安全支付协议。为了确保交易的安全性，协议需要在买家、卖家和银行之间进行多轮通信。还有一些系统提供更高的匿名性，就如同实验现金提供的那样。*DigiCash*支付系统就是这样的一个系统。在这样的系统中，在完成支付时不可能识别购买者。相反，使用信用卡可以很容易地识别购买者，即便在SET的情况下，也可以通过与信用卡公司或银行的合作来识别购买者。然而，由于技术和非技术两方面的原因，没有一个这样的系统在商业上成功。

目前，许多银行提供**安全支付网关**(secure payment gateway)，允许买方在银行的网站上在线支付，而不会向在线商家暴露信用卡或银行账户信息。当从在线商家购买商品时，买方的Web浏览器被重定向到支付网关，通过提供信用卡或银行账户信息完成支付，之后买方再次被重定向回商家的网站以完成购买。与SET或DigiCash协议不同，除了Web浏览器，在买方的机器上没有软件运行；其结果是在先前方法失败的场合，此方法取得了大范围的成功。

由PayPal系统使用的另一种方法是，让买方和商家在一个共同的平台上都拥有账户，转账完全发生在共同的平台内。买方首先使用信用卡往他的账户里充值，然后可以往商家账户里转账。这种方法对于小商家已经非常成功，因为它不需要买方或商家运行任何软件。

26.4 主存数据库

为了获得很高的事务处理率(每秒数百或数千的事务)，我们必须使用高性能硬件，而且必须使用并发性。但是，仅靠这些技术还不足以获得非常快的响应时间，因为磁盘I/O仍然是瓶颈——每次I/ [1105]

O需要大约10毫秒，而且这个数据并没有随着处理器速度的增长而相应减少。磁盘I/O通常是读的瓶颈，也是事务提交的瓶颈。长时间的磁盘延迟不仅增加了访问数据项的时间，而且限制了每秒存取的次数。[1]

我们可以通过增大数据库缓冲区来减小数据库系统受磁盘限制的程度。主存技术的进步使得我们能够以相对较低的代价构造相当大的主存。现在的商用64位系统能提供数十GB的主存。Oracle TimesTen是当前可用的主存数据库。在文献注解的参考文献中给出了有关主存数据库的更多信息。

对于某些应用，例如实时控制，必须将数据存储在主存中以满足性能要求。虽然至少有几个应用需要将几个GB的数据驻留在主存中，但大多数这类系统对于主存大小的要求并不是异常的大。随着主存大小的快速增长，会有越来越多的应用能够将数据存到主存中。

大的主存储器能使数据驻留在内存中，从而使事务处理更快。然而，仍然存在与磁盘有关的限制：

- 在事务提交前必须将日志记录写到稳定存储器中。由于大的主存所带来的性能提升可能导致日志处理成为瓶颈。我们可以通过使用非易失性RAM（例如通过有电池支持的内存来实现），在主存中建立稳定的日志缓冲区，从而缩减提交时间。由于日志带来的开销还可以通过本节后面讨论的组提交（group-commit）技术来减小。吞吐量（每秒的事务数目）仍然受日志磁盘的数据传输率的限制。
- 为了减小在恢复的时候必须重新执行的日志数量，仍然有必要写出被提交事务做了修改标记的缓冲块。如果更新率非常高，那么磁盘的数据传输率可能变成瓶颈。
- 如果系统崩溃了，所有的主存数据都会丢失。恢复时，系统的数据库缓冲是空的，对数据块进行访问时必须从磁盘输入。因此，即使恢复已经完成了，仍然需要一些时间才能把数据库完全装入到主存中，并恢复对事务的高速处理。

[1106] 另一方面，主存数据库为优化提供了机会：

- 由于主存比磁盘空间价格高，因此，设计主存数据库中的内部数据结构必须注意减小对空间的需求量。然而，数据结构中可以有跨多个页面的指针，这与磁盘数据库不同，其中跨越多个页面的I/O代价会非常高。例如，主存数据库中的树结构与B^+树不同，深度可以相对大一些，但应尽量少占空间。

  然而，高速缓冲存储器和主存储器之间的速度差异，以及在主存和高速缓存之间的数据传输是以高速缓存行（cache-line）（通常约64字节）为单位的事实，导致了这样的情况：高速缓存和主存之间的关系与主存和磁盘之间的关系没有什么不同（虽然有更小的速度差异）。其结果是，即使在主存数据库里，带有可以放入一个高速缓存行的小结点的B^+树被发现是相当有用的。
- 在数据存取前没有必要钉住内存中的缓冲页，因为缓冲页从来不会被替换。
- 查询处理技术应该设计成尽量减小空间开销，从而在执行查询时不会超出主存的限制；一旦发生这种情况会导致将页换出到交换区中，从而减慢查询处理速度。
- 一旦消除了磁盘I/O瓶颈，诸如封锁和闩那样的操作可能会变成瓶颈。必须改进这些操作的实现来消除这样的瓶颈。
- 可以对恢复算法进行优化，因为很少会需要将页写出以给其他页腾出空间。

处理事务T提交的过程中需要将下面这些记录写入稳定存储器中：

- 还没有输出到稳定存储器中的所有与T相关的日志记录。
- <T commit>日志记录。

这些输出操作经常需要输出只有部分填充的块。为了保证输出接近写满的块，我们采用组提交（group-commit）技术。系统不是在T完成时就试图提交T，而是等到有几个事务都完成了，或者从事务执行完后已经过了一个特定的时间段，然后将等待的这组事务一起提交。写出到稳定存储器上的块可能包含几个事务的记录。通过仔细选择组的大小和最大等待时间，系统可以保证当块写到稳定存储器

[1] 闪存的写延迟取决于是否需要先执行擦除操作。

时是满的，而无需事务等待过长的时间。平均说来，此技术减少了每个提交事务的输出操作。 1107

虽然组提交减少了日志带来的开销，但它造成了执行更新的事务在提交时的轻微延迟。延迟可能很小（比方说，10 毫秒），这对许多应用来说是可接受的。如果磁盘或磁盘控制器对写操作支持非易失性 RAM 缓存，这个延迟可以消除。一旦在非易失性 RAM 缓冲区中执行了写操作，事务就可以提交。在这种情况下，没有必要进行组提交。

注意，不只是对于主存数据库，即使在带有磁盘驻留数据的数据库中，组提交也是有用的。如果用闪存代替磁盘来存储日志记录，提交延迟会显著减少。然而，组提交仍然是有用的，因为它最大限度地减少了写出的页数，这就转化为闪存中的性能优势，因为页不能被覆盖地写，而且擦除操作的代价是昂贵的。（闪存存储系统将逻辑页重新映射到预先擦除的物理页，避免在写页面时的延迟，但最终必须执行擦除操作，作为页面旧版本的垃圾回收的一部分。）

26.5　实时事务系统

至此我们所考虑的完整性约束与存储在数据库中的值有关。在特定应用中，约束条件还包括指定任务必须完成的**截止时间**（deadline）。这类应用的例子包括工厂管理、交通控制以及调度。当把截止时间考虑在内时，执行的正确性就不再仅仅是数据库一致性的问题了。我们还关心截止时间延误了多少次，以及延误了多长时间。截止时间的特性如下：

- **硬截止时间**（hard deadline）。如果任务不能在其截止时间之前完成，可能出现严重问题，比如系统崩溃。
- **严格截止时间**（firm deadline）。如果任务在截止时间之后完成，它的价值为零。
- **软截止时间**（soft deadline）。如果任务在截止时间之后完成，它的价值逐渐缩小。随着延误时间的增长其价值趋近于零。

具有截止时间的系统称作**实时系统**（real-time system）。

实时系统中的事务管理必须将截止时间考虑在内。如果并发控制协议确定让事务 T_i 等待，那么它可能导致 T_i 错过截止时间。在这种情况下，可能最好是抢占持有锁的事务，而让 T_i 进行下去。然而，抢占的办法必须慎重使用，因为被抢占的事务所失去的时间（由于回滚和重启）可能导致该事务延误它的截止时间。遗憾的是，在给定情况下很难确定是回滚更好还是等待更好。

支持实时约束的一个主要困难来自事务执行时间的差异。在最好情况下，要访问的所有数据都在 1108
数据库缓冲区中。在最坏情况下，每次访问都导致将缓冲页写到磁盘中（还要先写必不可少的日志记录），然后再从磁盘读入包含待访问数据的页面。由于最坏情况下所需要的两次或更多次磁盘访问比最好情况下主存访问所花的时间要高好几个数量级，所以如果数据驻留在磁盘上，那么事务执行时间的估算就非常粗略。因此，如果需要满足实时约束，常常采用主存数据库。

然而，即使数据驻留在主存中，还会由于等待封锁、事务中止等导致执行时间的差异。研究者投入了相当大的精力来研究实时数据库的并发控制。他们扩展了封锁协议，为那些截止时间早的事务提供较高的优先级。他们发现优化的并发控制在实时数据库中执行得很好，即这些协议甚至比扩展的封锁协议更能减少截止时间的延误。文献注解中给出了在实时数据库领域中的研究工作的参考文献。

在实时系统中，最重要的问题不是绝对速度，而是截止时间。设计一个实时系统需要保证在不要求过多硬件资源的前提下，有足够的处理能力来满足截止时间的要求。尽管事务管理会导致执行时间的差异，但要达到这一目标仍是一个挑战性的问题。

26.6　长事务

事务概念原本是在数据处理应用的环境中提出的，在此环境中大多数事务不是交互式的，并且持续时间很短。虽然在此以及前面的第 14、15、16 章中讨论的技术能够很好地支持这些应用，但当事务的概念应用到涉及人机交互的数据库系统时，就出现了严重的问题。这些事务具有如下重要特性：

- **持续时间长**（long duration）。一旦人介入到活跃事务中，从计算机的角度看，该事务就变成了**长事务**（long-duration transaction），因为人的响应时间比计算机的速度慢。而且，在设计应用中，

人的活动可能会长达数小时、数天甚至更长。因此，从人的观点和从机器的观点看，事务的持续时间都很长。

- **暴露未提交数据**(exposure of uncommitted data)。由于事务可能中止，长事务产生并显示给用户的数据可能未提交。因此，用户或其他事务可能被迫去读未提交的数据。如果在一项工程中有几个用户在合作，那么用户事务可能需要在事务提交之前交换数据。
- **子任务**(subtask)。一个交互式事务可能包括一组由用户启动的子任务。用户可能希望中止其中
[1109] 一个子任务，而不必导致整个事务的中止。
- **可恢复性**(recoverability)。由于系统崩溃而中止一个交互式的长事务是不可接受的。活跃事务必须恢复到崩溃前不久的某个状态，以使人的工作丢失得相对较少。
- **性能**(performance)。在交互式事务系统中将性能好定义为响应时间快。这一定义与非交互式系统中的定义不同，非交互式事务系统的目标是高吞吐量(每秒的事务数目)。具有高吞吐量的系统能够有效利用系统资源。然而，在交互式事务的情况下，价格最高的资源是用户。如果要提高用户效率和满意程度，响应时间就应该快(从人的角度看)。在一个任务持续很长时间的情况下，响应时间应该是可预知的(即响应时间的变化应该不大)，这样用户就可以很好地安排自己的时间。

在 26.6.1 节到 26.6.5 节中，我们会看到为什么这五个特性与迄今所讨论的技术不相容。还将讨论如何对这些技术进行修改以适应交互式长事务。

26.6.1 不可串行化的执行

我们所讨论的这些特性使得前面章节中所用的只允许可串行化调度的要求变得不切实际。第 15 章中的每种并发控制协议对于长事务都有负面影响：

- **两阶段封锁**(two-phase locking)。当一个事务请求封锁而得不到时，它就被迫等待所要求的数据项被解锁。等待时间的长短与持有锁的事务的持续时间成比例。如果数据项被短事务封锁，我们预期等待时间会很短(除非发生了死锁或系统负载极重)。然而，如果数据项被长事务封锁，则会发生长时间的等待。长时间等待导致响应时间长，并且发生死锁的可能性增大。
- **基于图的协议**(graph-based protocol)。基于图的协议使得锁的释放比两阶段封锁协议要早，并且能避免死锁。然而，它对数据项强加了一个顺序。事务对于数据项的封锁必须以与该顺序一致的方式进行。其结果是，事务可能封锁的数据比它实际需要的要多。而且，在事务确信某个封锁不会再被使用之前，必须持有该封锁。这样，就可能发生长时间的封锁等待。
- **基于时间戳的协议**(timestamp-based protocol)。时间戳协议不需要事务等待。然而，在某些情况
[1110] 下，它确实需要事务中止。如果一个长事务中止了，就会丢失大量实质性的工作。对于非交互式事务来说，这种工作的丢失是难能问题。对于交互式事务来说，这还是个用户满意度的问题。用户很不希望发现他几个小时的工作都白做了。
- **有效性检查协议**(validation protocol)。与基于时间戳的协议一样，有效性检查协议通过中止事务来保证可串行性。

由此看来，强制要求可串行化可能会导致长时间的等待、长事务的中止，或者两者同时发生。文献注解中引用的一些理论结果证实了这一结论。

当我们考虑恢复问题时，会发现强制要求可串行化所带来的进一步的困难。前面我们讨论过级联回滚问题，即一个事务的中止可能导致其他事务的中止。这一现象是我们所不希望的，尤其对于长事务而言。如果使用封锁，又想避免级联回滚，就必须保持排他锁直至事务结束。然而，这样保持排他锁就增加了事务等待的时间。

由此看来，强制要求事务的原子性必然导致要么长时间等待的可能性增大，要么产生级联回滚的可能性。

在 15.7 节描述的快照隔离，可以提供对这些问题的部分解决方案，在 15.9.3 节中描述的无读有效性检查的乐观并发控制(optimistic concurrency control without read validation)协议也可以。后一种协议实际上是用来专门处理涉及用户交互的长事务。虽然无读有效性检查的乐观并发控制并不能保证可串

行性，但它得到了相当广泛的使用。

然而，当事务具有长的持续时间，就更有可能产生冲突的更新，进而导致额外的等待或中止。以上考虑为我们在本节其余部分考虑的另外一些有关并发执行正确性和事务恢复的概念打下了基础。

26.6.2 并发控制

数据库并发控制的基本目标是确保事务的并发执行不会导致数据库一致性的丢失。可以利用可串行化的概念来达到这一目标，因为所有可串行化的调度都能保持数据库的一致性。然而，并非所有保持数据库一致性的调度都是可串行化的。例如，再次考虑有两个账户 A 和 B 的银行数据库，其一致性要求是 $A+B$ 之和保持不变。虽然图 26-5 中的调度不是冲突可串行化的，但它却保持了 $A+B$ 的和不变。此例还说明了无可串行化条件下有关正确性概念的两个重要观点。

- 正确性依赖于特定的数据库一致性约束。
- 正确性依赖于每个事务所执行的操作的类型。

[1111]

通常，事务对低层次操作进行自动分析并检测这些操作对数据库一致性约束的影响是不可能的。然而，还有一些更简单的技术。其中之一是以数据库一致性约束为基础，将数据库划分成一些子数据库，分别管理每个子数据库上的并发。另一种技术是将一些操作（包括 **read** 和 **write**）看作是基本的低层次操作，并且对并发控制进行扩展以处理这些操作。

T_1	T_2
read(A)	
$A \leftarrow A - 50$	
write(A)	
	read(B)
	$B \leftarrow B - 10$
	write(B)
read(B)	
$B \leftarrow B + 50$	
write(B)	
	read(A)
	$A \leftarrow A + 10$
	write(A)

图 26-5 一个非冲突可串行化的调度

文献注释中引用了不要求可串行性而保证一致性的另外一些技术。这些技术中有许多都开发了多版本并发控制（参见 15.6 节）的变体。对于较早的、只需要一个版本的数据处理应用来说，多版本协议为存储额外的版本导致了较高的空间开销。由于许多新的数据库应用需要维护数据的多个版本，所以开发多版本的并发控制技术是符合实际的。

26.6.3 嵌套事务和多级事务

可以将一个长事务看成一组相关的子任务或子事务。通过一组子事务来构建事务，我们可以提高并行度，因为有可能并行地运行几个子事务。而且，处理子事务失败（由于中止、系统崩溃等造成）还可以不用回滚整个长事务。

一个嵌套的或多级的事务 T 包括一个子事务的集合 $T=\{t_1, t_2, \cdots, t_n\}$ 和 T 上的一个偏序 P。T 中的一个子事务 t_i 可以中止，而无需强制 T 中止。反过来，T 可以重启 t_i，或是简单地选择不运行 t_i。如果 t_i 提交了，这一动作并不使 t_i 成为永久的（与第 16 章中的情况不同）。相反，t_i 向 T 提交。如果 T 中止，t_i 可能仍会中止（或者需要补偿——见 26.6.4 节）。T 的执行不能违反偏序 P。也就是说，如果在优先图中出现边 $t_i \rightarrow t_j$，那么 $t_i \rightarrow t_j$ 必定不能在 P 的传递闭包中。

[1112]

嵌套可能有好几层深，表示将事务划分为子任务，子任务的子任务，如此等等。在嵌套的最底层，是我们前面用到的标准数据库操作 **read** 和 **write**。

如果允许 T 的子任务在完成的时候释放锁，那么 T 称为**多级事务**（multilevel transaction）。当一个多级事务代表一个长时间的活动时，该事务有时亦作一个**传奇**（saga）。另一种情况是，如果 T 的子事务 t_i 所持有的锁在 t_i 完成时自动地赋给 T，那么 T 就称作**嵌套事务**（nested transaction）。

虽然多级事务的主要实用价值在于复杂的长事务，但我们将用图 26-5 的简单例子来说明如何用嵌套来建立起更高级的操作，从而提高并发度。我们用子事务 T_{11} 和 T_{12} 来重写事务 T_1。这两个子事务执行增加或减少的操作：

- T_1 包括：
  - T_{11}，它从 A 中读掉 50。
  - T_{12}，它往 B 中加上 50。

类似地，我们用子事务 T_{21} 和 T_{22} 来重写事务 T_2，这两个子事务也执行增加或减少的操作：

- T_2 包括：
  - 执行 $T_{2,1}$，它从 B 中减掉 10。
  - 执行 $T_{2,2}$，它往 A 中加上 10。

对于 $T_{1,1}$、$T_{1,2}$、$T_{2,1}$、$T_{2,2}$没有说明顺序。这些子事务的任意执行都会产生正确结果。图 26-5 中的调度对应于调度 $< T_{1,1}, T_{2,1}, T_{1,2}, T_{2,2} >$。

26.6.4 补偿事务

为减少长时间等待的发生频率，我们将未提交的更新暴露给其他并发执行的事务。事实上，多级事务可能允许这种曝光。然而，曝光未提交数据产生了级联回滚的可能性。补偿事务（compensating transaction）的概念有助于我们处理这一问题。

设事务 T 被划分为几个子事务 t_1，t_2，…，t_n。在子事务 t_i 提交之后，它释放了它的封锁。现在，如果必须中止外层事务 T，那么它的子事务所造成的影响必须被撤销。假设子事务 t_1，…，t_k 已经提交了。在作出中止决定时 t_{k+1} 正在执行。我们可以通过中止子事务 t_{k+1} 来消除它的影响。然而，不可能中止子事务 t_1，…，t_k，因为它们已经提交了。

1113

取而代之，我们执行一个新的子事务（称作补偿事务）ct_i 去撤销子事务 t_i 的影响，每个子事务 t_i 都需要有一个补偿事务 ct_i。补偿事务必须以相反的顺序 ct_k，…，ct_1 来执行。下面给出几个补偿的例子：

- 考虑图 26-5 的调度，我们已经说明了它是正确的，尽管它不是种类可串行化的。每个子事务一旦完成就释放它的封锁。假设在 T_2 即将结束前，$T_{2,2}$ 已经释放了它的封锁之后，T_2 失败了。然后我们就运行一个对 $T_{2,2}$ 的补偿事务，它从 A 中减掉 10，并运行一个对 $T_{2,1}$ 的补偿事务，它往 B 中加上 10。
- 考虑事务 T_1 所做的一个数据库插入，其副作用造成了 B^+ 对索引的更新。该插入操作可能修改了 B$^+$ 树索引的几个结点。其他事务在访问数据（不是 T_1 新插入的记录）时可能已经读过这些结点。正如 16.7 节所提到的，我们可以通过删除 T_1 插入的记录来撤销这个插入的影响。其结果是一棵正确的、一致的 B^+ 树，但不一定与 T_1 开始之前的结构完全相同。因此删除是插入的补偿动作。
- 考虑一个长事务 T_1，它代表旅行预订。事务 T_1 有 3 个子事务：$T_{1,1}$ 预订飞机票；$T_{1,2}$ 预订出租车；$T_{1,3}$ 预订旅馆房间。假设旅馆取消了预订。我们不用撤销整个 T_1，对 $T_{1,3}$ 失败的补偿是删除原来的旅馆订单，并且建立一个新的订单。

如果系统在执行外层事务的中间崩溃了，那么当对它进行恢复时必须回滚它的子事务。16.7 节所描述的技术可以用于这一目的。

对事务失败的补偿需要利用失败事务的语义。对于某些操作，例如向 B^+ 树中增加或插入，很容易定义相应的补偿。对于更复杂的事务，可能需要应用编程者在对事务进行编码时定义补偿的正确形式。对于复杂的交互事务，可能需要系统与用户进行交互，以确定补偿的正确形式。

26.6.5 实现问题

本节讨论的事务概念为实现带来了很大困难。我们在这里列出了其中的几个，并讨论我们可以如何处理这些问题。

1114

长事务必须能够免遭系统崩溃的影响。我们能够确保这一点，办法是对已提交子事务执行 **redo**。对于崩溃时正活跃的短的子事务执行 **undo** 或补偿。然而，这些动作仅解决了部分问题。在典型的数据库系统中，诸如锁表、事务时间戳等这样的内部系统数据是存放在易失性存储器中的。为使长事务在崩溃后能够继续，这些数据必须恢复。因此，不仅需要对数据库的改变记日志，而且需要对与长事务有关的内部系统数据的改变记日志。

当数据库中存在特定类型的数据项时，记录更新日志就会变得更为复杂。数据项可能是一个 CAD 设计、文档的正文或者另一种复合设计的形式。这些数据项物理上很大。因此我们不希望在日志记录中既存储这种数据项的旧值，也存储其新值。

有两种方法可以减少保证大数据项可恢复性的开销：

- **操作日志**(operation logging)。在日志中只存放在数据项上执行的操作和数据项的名称。操作日志也称作**逻辑日志**(logical logging)。对每个操作，必须存在一个逆操作。我们用逆操作执行**undo**，用操作本身执行**redo**。通过操作日志进行恢复更为困难，因为**redo**和**undo**不是幂等的。而且，对于更新多个页面的操作使用逻辑日志是非常复杂的，其原因是更新过的某些页面(但并不是所有页面)可能已经写出到磁盘中了，因此在恢复时很难在磁盘镜像上进行该操作的**redo**或**undo**。如16.7节描述的那样，使用物理重做日志和逻辑撤销日志提供了逻辑日志并行性的优点，同时避免了上述缺陷。
- **日志和影子分页**(logging and shadow paging)。日志用于对小数据项的更新，而大数据项的可恢复性通常通过影子分页或复制写技术来实现。当我们使用影子分页时，通过只保存那些实际上被修改过的页面的副本可以降低开销。

尽管采用了这些技术，由于长事务和大数据项而引入的复杂性仍然使恢复过程复杂化。因此，我们希望允许对于特定的非关键数据不记日志，而依赖于脱机备份和人工干预。

26.7 总结

- 工作流是一些活动，涉及由不同处理实体实现的多个任务的协同执行。它们不仅存在于计算机应用中，也存在于几乎所有的组织机构活动中。随着网络的发展，以及多个自治数据库系统的存在，工作流提供了一种方便的方式来执行涉及多个系统的任务。
- 虽然对于这样的工作流应用来说，通常的ACID事务性要求太强了，或者不可能实现，但是工作流必须满足事务特性的一个有限集合，以保证进程不会处于一种不一致的状态。 [1115]
- 最初开发事务处理监控器是作为多线程服务器。由一个单独的进程为大量终端服务。由此演变而来，现在事务处理监控器为建立和管理有大量客户端和多个服务器的复杂的事务处理系统提供基础设施。它提供各种服务，例如客户端访问和服务器响应的持久队列、客户端消息到服务器的路由、持久消息、负载均衡、当事务访问多个服务器时的两阶段提交的协调。
- 电子商务系统已成为商务的核心部分。在电子商务系统中有几个数据库方面的问题。目录管理，特别是个性化目录，需要利用数据库实现。电子市场帮助确定和卖、反向和卖或交易中的产品的价格。处理这类交易需要高性能的数据库系统。订单由电子支付系统结算，这同样需要有非常高的事务处理率的高性能数据库系统。
- 某些系统采用开发了大容量主存以达到较高的系统吞吐量。在这样的系统中，日志成为瓶颈。依据组提交的概念，可以减少向稳定存储器输出的次数，从而缓解这个瓶颈。
- 有效管理持续时间长的交互式事务是一个更加复杂的问题，因为有长时间的等待，还因为有中止的可能性。由于第15章采用的并发控制技术使用等待或中止，或者两者都采用，因此必须考虑另一些可选技术。这些技术必须在不要求可串行性的条件下保证正确性。
- 长事务表示为嵌套事务，其最底层是数据库的原子操作。如果一个事务失败了，只有活动着的短事务中止。一旦所有短事务恢复了，活动的长事务就继续进行。如果外层事务失败，需要使用补偿事务对已提交的嵌套事务的更新进行撤销。
- 在具有实时约束的系统中，执行的正确性不仅包括数据库一致性，而且还包括满足截止时间。软和与条件在执行时间上的巨大差异使得具有时间约束的系统中的事务管理问题复杂化了。

术语回顾

- TP监控器
- TP监控器体系结构
  - 每客户进程
  - 单服务器
  - 多服务器单路由器
  - 多服务器多路由器
- 多任务调度
- 上下文切换
- 多线程服务器
- 队列管理器
- 应用协调器
  - 资源管理器
  - 远程过程调用(RPC)
- 事务工作流
  - 任务
  - 处理实体
  - 工作流说明
  - 工作流执行

- 工作流状态
  - 执行状态
  - 输出值
  - 外部变量
- 工作流的故障原子性
- 工作流终止状态
  - 可接受的
  - 不可接受的
  - 提交的
  - 中止的
- 工作流恢复
- 工作流管理系统
- 工作流管理系统体系结构
  - 集中式
  - 部分分布式
  - 完全分布式
- 业务流程管理
- 编排
- 电子商务
- 电子目录
- 市场
  - 拍卖
  - 反向拍卖
  - 交易
- 订单结算
- 数字证书
- 主存数据库
- 组提交
- 实时系统
- 截止时间
  - 硬截止时间
  - 严格截止时间
  - 软截止时间
- 实时数据库
- 长事务
- 未提交数据的曝光
- 不可串行化执行
- 嵌套事务
- 多级事务
- 传奇(Sagas)
- 补偿事务
- 逻辑日志

实践习题

1116 1117 26.1 类似于数据库系统，工作流系统同样需要并发和恢复管理。列出三个原因说明为什么我们不能简单地应用一个采用2PL、物理撤销日志和2PC的关系数据库系统。

26.2 考虑系统崩溃后的主存数据库系统的恢复。解释下列方法的相对优缺性：

- 在重新开始事务处理之前将整个数据库载入到主存中。
- 当事务请求数据时将该数据载入。

26.3 高性能事务系统一定是实时系统吗？为什么？

26.4 解释为什么要求长事务可串行化可能是不现实的。

26.5 考虑从持久消息的持久队列中发送消息的多线程处理过程。不同线程可以并发运行，试图发送不同的消息。在发送失败的情况下，消息必须重新存回队列中。像多级事务那样模拟每个线程执行的动作，这样只有在发送消息时才需要持有该队列上的封锁。

26.6 如果我们允许嵌套事务，请讨论第16章中介绍的每一种恢复模式需要做什么修改。另外，如果我们允许多级事务，请解释那会带来什么不同。

习题

26.7 解释TP监控器在管理内存和处理器资源方面如何比通常的操作系统更加有效。

26.8 比较TP监控器和支持servlet的Web服务器(这种服务器的昵称叫*ZP-hac*)的特征。

26.9 考虑你所在的大学接收新学生(或你所在的公司接收新员工)的过程。

a. 从学生申请手续开始，给出工作流的高层描述。

b. 指出可接受终止状态以及需要人工干预的步骤。

c. 指出可能的错误(包括超过截止时间)以及如何处理它们。

d. 调查你所在的学校里工作流的多少部分已经自动化了。

1118 26.10 回答下面关于电子支付系统的问题：

a. 解释为什么用信用卡号码执行的电子事务可能是不安全的。

b. 另一种选择是通过信用卡公司维护一个电子支付网关，接受支付业务的站点将客户转到网关站点来完成支付。

i. 解释如果网关不进行用户鉴别，那么这样的系统能够提供哪些好处。

ii. 解释如果网关有用户鉴别机制，那么还能够提供哪些进一步的好处。

c. 一些信用卡公司提供一种一次性信用卡号码作为一种更为安全的电子支付方法。客户连接到信用卡公司的网站得到一个一次性号码。解释与使用普通信用卡号码相比，这样的系统能够提供怎样的好处。也请解释与使用鉴别机制的电子支付网关相比，这种系统有哪些好处和不足。

d. 当交易是用现金进行的，上面所说的系统中有保证同样的隐私吗？解释你的答案。

26.11 如果整个数据库能放入主存，我们还需要数据库系统来管理数据吗？解释你的答案。

26.12 在组提交技术中，一个组里应该包括多少个事务？解释你的答案。

26.13 在使用先写日志的数据库系统中，最坏情况下从指定磁盘页读取一个数据项所需的磁盘访问次数是多少？解释为什么这对于实时数据库系统设计者提出了一个问题。提示：考虑磁盘缓冲的情况。

26.14 补偿事务的目的是什么？给出使用补偿事务的两个例子。

26.15 解释在工作流和长事务之间的联系。

文献注解

Gray 和 Reuter[1993]是描述事务处理系统详细的(并且极好的)教科书，包括有关 TP 监控器的几个章节。X/Open[1991]定义了 X/Open XA 接口。

Fischer[2006]是一本工作流系统的手册，由工作流管理协会联合出版。该协会的网址是 www.wfmc.org。我们对工作流的描述沿用了 Rusinkiewicz 和 Sheth[1995]的模型。

Loeb[1998]提供了安全电子交易的详细描述。 1119

Garcia-Molina 和 Salem[1992]给出了关于主存数据库的综述。Jagadish 等[1993]描述了为主存数据库设计的一个恢复算法。Jagadish 等[1994]描述了主存数据库的存储管理。

Lam 和 Kuo[2001]讨论了实时数据库。Haritsa 等[1990]、Hong 等[1993]、Pang 等[1995]讨论了实时数据库中的并发控制和调度。Ozsoyoglu 和 Snodgrass[1995]是对关于实时数据库和时态数据库研究的综述。

Moss[1985]、Lynch 和 Merritt[1986]、Moss[1987]、Haerder 和 Rothermel[1987]、Rothermel 和 Mohan[1989]、Weikum 等[1990]、Korth 和 Speegle[1990]、Weikum[1991]，以及 Korth 和 Speegle[1994]介绍了嵌套事务和多级事务。Lynch 等[1988]介绍了多级事务的理论层面。Garcia-Molina 和 Salem[1987]介绍了 Sagas 的概念。 1120

第九部分
Part 9

实例研究

本部分描述不同的数据库系统如何将本书前面所描述的各种概念相融合。我们首先在第27章介绍广泛使用的开源数据库系统PostgreSQL。第28、29、30章介绍三种广泛应用的商用数据库系统——IBM DB2、Oracle和Microsoft SQL Server。这三个系统代表了三种应用最为广泛的商用数据库系统。

这几章中每章都强调了各个数据库系统独有的特性：工具、SQL变化和扩展以及系统体系结构，包括存储组织、查询处理、并发控制与恢复以及复制能力。

这几章只包括所讲述数据库产品的关键方面，所以不能看成是这些产品的全面介绍。更何况由于产品经常改进，产品细节会发生变化。当使用某个特定版本的产品时，对于特定细节一定要参考用户手册。

记住本部分章节通常使用的是工业术语而不是学术术语。例如，用表代替关系，行代替元组，列代替属性。

第 27 章
Database System Concepts, 6E

PostgreSQL

Anastasia Ailamaki, Sailesh Krishnamurthy,
Spiros Papadimitriou, Bianca Schroeder,
Karl Schnaitter, and Gavin Sherry

PostgreSQL 是一个开放源代码的对象 - 关系数据库管理系统。它是由加州大学伯克利分校的 Michael Stonebraker 教授主持开发的 Postgres 系统的后代。Postgres 系统是最早的对象 - 关系数据库管理系统之一。名字"Postgres"来自于关系数据库系统的先驱之一 Ingres，它同样也是由伯克利的 Stonebraker 主持开发的。目前 PostgreSQL 支持 SQL:2003 的许多方面并提供许多特性，如复杂查询、外码、触发器、视图、事务一致性、全文检索和受限的数据复制。另外，用户可以为 PostgreSQL 扩展新的数据类型、函数、操作符或索引方法。PostgreSQL 支持多种程序设计语言(包括 C、C++、Java、Perl、Tcl 和 Python)以及 JDBC 和 ODBC 数据库接口。PostgreSQL 的另一个突出点是，它和 MySQL 是两种使用最为广泛的开源关系数据库系统。PostgreSQL 许可证是 BSD 许可证。这就允许任何人以任何目的使用、修改和发布 PostgreSQL 代码和文档而不需要付费。

27.1 概述

在二十年中，PostgreSQL 有过几次大的发布。最早的原型系统，名字叫做 Postgres，在 1988 年的 ACM SIGMOD 会议上演示过。1989 年发布给用户的第一个版本提供了许多特征，如可扩展数据类型、一个初步的规则系统和名为 Postquel 的查询语言。在随后的多个版本中增加了一个新的规则系统、对多存储管理器的支持以及改进的查询执行器。然后系统开发者关注于系统的便携性和性能，直到 1994 年增加了 SQL 语言解释器。之后系统以 Postgres95 这个新名字发布到 Web 上，随后被 Illustra Information Technologies(后又合并到 Informix，Informix 现在为 IBM 所有)商业化。到 1996 年，PostgreSQL 这个名字取代了 Postgres95，以反映原始的 Postgres 和兼容 SQL 的最新版本之间的关系。

PostgreSQL 实际上可以运行在所有类 UNIX 操作系统上，包括 Linux 和 Apple Macintosh OS X。早期版本的 PostgreSQL 服务器能运行在 Cygwin 环境中的 Microsoft Windows 下，这个环境提供 Windows 中的 Linux 仿真。在 2005 年 1 月发布的版本 8.0 提供了对 Microsoft Windows 的天然支持。

今天，PostgreSQL 用于实现若干不同的研究和产品应用(例如用于地理信息的 PostGIS 系统)，也在一些大学作为教学工具。在由大约 1000 名开发者组成的社区的努力下，系统在向着持续演进。在本章我们将解释 PostgreSQL 怎样工作，从用户界面和语言开始，直到系统的核心(数据结构和并发控制机制)。

27.2 用户界面

PostgreSQL 标准发布版本附带有管理数据库的命令行工具。然而，有众多支持 PostgreSQL 的商业和开源的图形管理和设计工具。软件开发人员也可以通过一组全面的编程接口来访问 PostgreSQL。

27.2.1 交互式终端界面

像大多数数据库系统一样，PostgreSQL 为数据库管理提供命令行工具。其主要的交互式终端客户是 psql，它模仿 UNIX shell，允许在服务器上执行 SQL 命令和一些其他操作(如客户端复制)。它的一些特性包括：

- **变量**(variable)。psql 提供变量替换特征，与通常的 UNIX 命令解释器相似。
- **SQL 替换**(SQL interpolation)。通过在变量名前放一个冒号，用户能将 psql 中的变量替换("interpolate")为正规的 SQL 语句。

- 命令行编辑(command-line editing)。psql 调用 GNU 读行库程序以进行便捷的行编辑，同时支持按 tab 键自动补全输入。

PostgreSQL 还可以通过 Tcl/Tk 解释器访问，它是一种灵活的脚本语言，常用于快速实现系统原型。这个功能通过在 Tcl/Tk 中加载 pgtcl 库来实现，并作为 PostgreSQL 可选的扩展进行发布。

27.2.2 图形界面

PostgreSQL 的标准发布版本不包含任何图形工具。然而已经有一些图形化的用户界面工具，用户可以选择商用产品或者开源软件。许多此类工具发布周期很快，下面所介绍的反映了本书写作时的情形。 1124

用于管理的图形化工具包括 pgAccess 和 pgAdmin。图 27-1 显示了后者。数据库设计工具包括 TORA 和 Data Architect，后者如图 27-2 所示。PostgreSQL 可以与若干商用表单设计和报表生成工具一起使用。可供选择的开源替代软件包括 Rekall(如图 27-3 和图 27-4 所示)、GNU Report Generator 和一个更加全面的工具包 GNU Enterprise。



图 27-1 pgAdmin III：一个开源的数据库管理 GUI

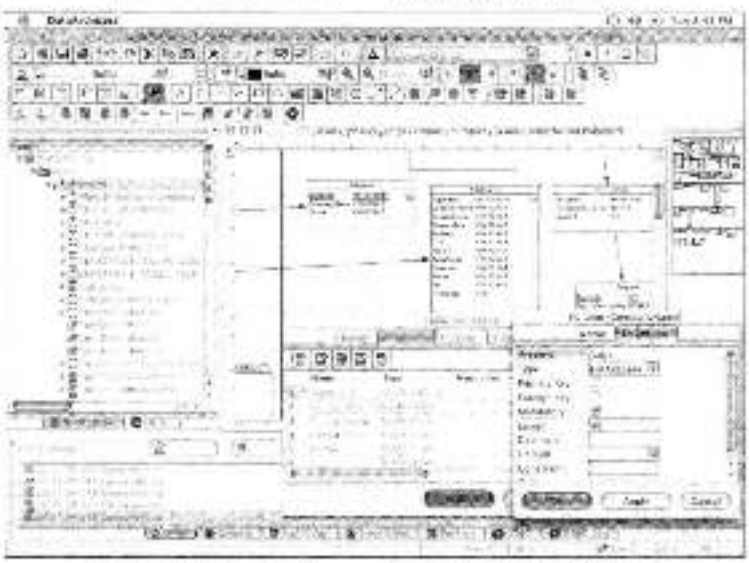


图 27-2 Data Architect：一个多平台数据库设计 GUI



图 27-3 Rekall：表单设计 GUI

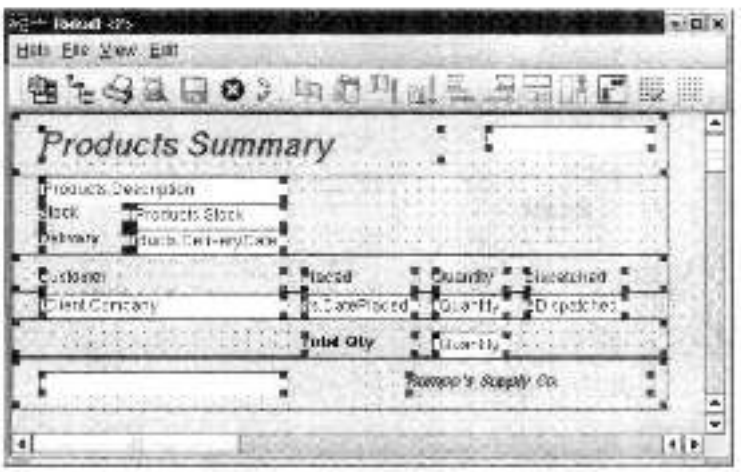


图 27-4 Rekall：报表设计 GUI

27.2.3 编程语言接口

PostgreSQL 提供支持 ODBC 和 JDBC 的原生接口，同时为大多数编程语言提供绑定接口，包括 C、C++、PHP、Perl、Tcl/Tk、ECPG、Python 和 Ruby。

libpq 库为 PostgreSQL 提供了 C 的 API，它也为大多数编程语言绑定的基础引擎。通过五个入口和线程安全的接口，libpq 库同时支持 SQL 命令和准备语句的同步和异步执行。libpq 的连接参数可以用几种灵活的方式来设置，如设置环境变量、将配置信息存储在本地文件中或者在一个 LDAP 服务器上创建条目。

27.3 SQL 变化和扩展

PostgreSQL 的当前版本支持几乎所有的初级 SQL92 特征，以及许多中级和完全级 SQL92 特征。它同时支持许多 SQL:1999 和 SQL:2003 特征，包括第 22 章所描述的大多数对象－关系特征和第 23 章描述的用于解析 XML 数据的 SQL/XML 特征。事实上，当前 SQL 标准的一些特征（例如数组、函数和继承）是由 PostgreSQL 及其相关开创的。它没有支持 OLAP 特征（尤其是 **cube** 和 **rollup** 操作），但来自

PostgreSQL 的数据可容易地导入到开源的外部 OLAP 服务器(如 Mondrian)和其他商用产品中。

27.3.1 PostgreSQL 类型

PostgreSQL 支持一些对于特定应用领域非常有用的非标准类型。进一步说，用户可以用 **create type** 命令定义新类型，包括新的低层次的基本类型，通常用 C 书写(见 27.3.3.1 节)。

27.3.1.1 PostgreSQL 类型系统

PostgreSQL 类型分为如下几类：

- **基本类型**(base type)。基本类型又称为**抽象数据类型**(abstract data type)，也就是封装状态和一组操作的模块。它们在 SQL 层以下实现，通常用像 C 这样的语言来实现(见 27.3.3.1 节)。例如 **int4**(已经包含在 PostgreSQL 中)或 **complex**(包含在可选的扩展类型中)。一个基本类型可以表示一个单独的标量值或者一个可变长度的数组值。对于数据库中存在的每个标量类型，PostgreSQL 自动创建一个数组类型用于存放相同标量类型的值。
- **复合类型**(composite type)。它们对应于表中的行，也就是说，它们是字段名字及其相应类型的列表。当创建一个表时就会隐式创建一个复合类型，当然用户也可以显式地构造它们。
- **域**(domain)。域类型是通过一个基本类型关联一个这种类型的值必须满足的约束来定义的。只要满足约束，域类型的值与所关联的基本类型的值就可以互换着使用。一个域可以拥有一个可选的默认值，其含义如同表中列所对应的默认值。
- **枚举类型**(enumerated type)。它们类似于如 C 和 Java 那样的编程语言中使用的 enum 类型。枚举类型实质上是一个指定了值的固定列表。在 PostgreSQL 中，枚举类型可以转换成它们名称的文本表示，但是这种转换在某些情况下必须显式地声明以确保类型安全。例如，如果没有显式转换成相容类型，不同枚举类型的值就不可以进行比较。
- **伪类型**(pseudotype)。当前，PostgreSQL 支持下述伪类型：*any*、*anyarray*、*anyelement*、*anyenum*、*anynonarray*、*cstring*、*internal*、*opaque*、*language handler*、*record*、*trigger* 和 *void*。它们不能用在复合类型中(从而不能用于表中的列)，但是可作为用户自定义函数的参数和返回类型。
- **多态类型**(polymorphic type)。*anyelement*、*anyarray*、*anynonarray* 和 *anyenum* 四种伪类型都称为**多态**(polymorphic)类型。携带此类参数的函数(相应地称为**多态函数**(polymorphic function))可以用在任何实际类型上。PostgreSQL 有一个简单的类型解析模式，要求：(1)在一个多态函数的任意一次调用中，一个多态类型的所有出现必须限制在相同实际类型上(也就是说，定义为 *f*(*anyelement*, *anyelement*)的函数只可以运行在一对相同的实际类型参数上)；(2)如果返回类型是多态的，那么至少有一个参数必须是相同的多态类型。

1125
1128

27.3.1.2 非标准类型

本节所描述的类型包含在标准发布版本中。通而言之，感谢 PostgreSQL 开放的本质，还有一些贡献以来的扩展类型。如度数和 ISBN/ISSN(见 27.3.3 节)。

几何数据类型(*point*、*line*、*lseg*、*box*、*polygon*、*path*、*circle*)用于地理信息系统中，用来表示二维空间对象，如点、线段、多边形、路径和圆。PostgreSQL 中提供许多函数和运算符来执行各种几何运算，例如缩放、平移、旋转和相交判断。PostgreSQL 进一步用 R 树来支持对这些类型的索引(见 25.3.5.3 节和 27.5.2.1 节)。

PostgreSQL 中的全文检索使用 *tsvector* 类型来表示一份文档，用 *tsquery* 类型表示一个全文查询。在将每个词的不同变体转换到一种共同的规范形式以后(例如，去除词干)，一个 *tsvector* 存储一份文档中可相互区分的词。PostgreSQL 提供函数将原始文本转换到一个 *tsvector*，并连接各个文档。一个 *tsquery* 声明了用于在候选文档中搜索的词。这些词之间通过布尔运算符连接。例如，查询"index & !(tree | hash)"是寻找包含"index"但没有用词"tree"或"hash"的文档。PostgreSQL 本身的设计就支持在全文本类型上的运算，包括语言特征和索引搜索。

PostgreSQL 提供存储网络地址的数据类型。这些数据类型允许网络管理应用程序用 PostgreSQL 数据库来存储它们的数据。对于熟悉计算机网络的读者，在此我们为这个特征提供一个简要的概述。IPv4、IPv6 和媒体访问控制(MAC)地址都有单独的类型(分别是 *cidr*、*inet* 和 *macaddr*)。*inet* 和 *cidr* 类

型都能存放 IPv4 和 IPv6 地址，并附带一个可选的子网掩码。它们主要的差别在于输入/输出的格式不同，以及这样的限制：无类别域间路由（Classless Internet Domain Routing，CIDR）地址不接受网络掩码右边带非零位的值。*macaddr* 类型用于存储 MAC 地址（典型的是以太网卡硬件地址）。PostgreSQL 支持对这些类型的索引和排序，以及一组操作（包括子网测试和将 MAC 地址映射为硬件制造商名称）。此外，这些类型提供输入错误校验。因此它们比无格式的文本域更好用。

PostgreSQL 的 *bit* 类型能存储固定的和变长的由 1 和 0 组成的串。对于这些值，PostgreSQL 支持位
1129 逻辑运算符和字符串操作函数。

27.3.2 规则和其他主动数据库特征

PostgreSQL 支持 SQL 约束和触发器（以及存储过程，见 27.3.3 节）。此外，它的一大特色是可在服务器上声明查询重写规则。

PostgreSQL 支持 **check** 约束、非空约束、主码和外码约束（带受限删除和级联删除）。

像许多其他关系数据库系统一样，PostgreSQL 支持触发器，它可用于非平凡约束和一致性检查或强制执行。触发器函数可用过程语言如 PL/pgSQL（见 27.3.3.4 节）或 C 语言来编写，但不能用普通 SQL 编写。触发器可以在 **insert**、**update** 或 **delete** 操作之前或之后执行，对于每个修改行执行一次或者每条 SQL 语句执行一次。

PostgreSQL 规则系统允许用户在数据库服务器上定义查询重写规则。与存储过程以及触发器不同，规则系统介于查询解析器和计划器之间，并根据规则集修改查询。当原始查询树转换成一个或多个树后，传递到查询计划器。因此，计划器拥有所有必要的信息（需要扫描的表、它们之间的关系、限定条件、连接信息等等），能做出一个有效的执行计划，即使涉及复杂的规则。

声明规则的通用语法是：

```
create rule rule_name as
    on | select | insert | update | delete |
    to table [ where rule_qualification ]
    do [ instead ] | nothing | command | ( command ; command ... )
```

本节其他部分提供了一些例子来展示规则系统的功能。关于规则如何匹配查询树以及查询树随后如何转换的更多细节，可以在 PostgreSQL 文档中找到（见文献注解）。规则系统在查询处理的重写阶段实现。27.6.1 节将对此进行解释。

首先，PostgreSQL 使用规则系统来实现视图。如下定义的视图

```
create view myview as select * from mytab;
```

将转化为下述规则定义：

```
create table myview ( same columns list as mytab );
create rule _return as on select to myview do instead
    select * from mytab;
```

myview 上的查询在执行前转换成在底层表 *mytab* 上的查询。**create view** 语法在这种情况下认为是
1130 更好的编程形式，因为它更简明，也防止了创建相互引用的视图（如果在声明规则时不小心，可能会出现这种情况，导致潜在的、让人迷惑的运行时错误）。然而，规则可用于显式地定义视图上的更新行为（**create view** 语句不允许这样做）。

作为另一个例子，考虑用户想要记录所有教员薪水的增涨情况。这可以通过下面的规则来完成：

```
create rule salary_audit as on update to instructor
    where new.salary <> old.salary
    do insert into salary_audit
    values ( current_timestamp, current_user,
             new.name, old.salary, new.salary );
```

最后，我们给出一条稍微更复杂的插入/更新规则。假设即将到来的薪水增涨存放在 *salary_increases(name, increase)* 表中。我们声明一个具有相同域的"虚拟"表 *approved_increases*，之后定义如下规则：

```
create rule approved_increases_insert
    as on insert to approved_increases
    do instead
    update instructor
            set salary = salary + new.increase
            where name = new.name;
```

然后下面的查询

```
insert into approved_increases select * from salary_increases;
```

将立刻更新 *instructor* 表中的所有薪水。由于规则中声明了 **instead** 关键字，表 *approved_increases* 不会改变。

规则和行触发器在功能上有一些重叠之处。PostgreSQL 规则系统可用于实现大部分触发器，但某些类型的约束（特别是外码）不能用规则来实现。同时，触发器具有附加的能力来生成错误信息以发出违反限制的信号，然而规则仅能悄悄地通过禁止无效的修改来实施数据完整性。另一方面，触发器不能用于视图上的更新（update）或删除（delete）行为，但规则却能。虽然在视图关系中没有真正的数据，触发器就绝不会被调用。

触发器和规则的一个重要区别在于，触发器对于每个受影响的行会迭代执行。另一方面，规则会在查询计划之前操作查询树。因此如果一条语句影响了许多行，那么规则会远比触发器高效得多。

PostgreSQL 中触发器和约束的实现机制将在 27.6.4 节中简要概述。 [1131]

27.3.3 可扩展性

像大多数关系数据库系统一样，PostgreSQL 将数据库、表、列等信息存储在通常所谓的系统目录（system catalog）中，在用户看来，它们和普通表一样。其他关系数据库系统在进行扩展时，典型的方式是通过改变源代码中的硬编码过程或加载供应商所写的特殊扩展模块。

和大多数关系数据库系统不同，PostgreSQL 更进一步，将更多信息存放在它的目录中：不仅有关于表和列的信息，还包括关于数据类型、函数、访问方法等方面的信息。因此，PostgreSQL 更容易让用户扩展，而方便了快速实现新的应用和存储结构的原型。通过动态加载共享对象，PostgreSQL 也能将用户编写的代码合并到服务器中。这提供了另一种用于编写扩展的途径，它可以在基于目录的扩展不够高效时使用。

此外，PostgreSQL 发布版本中的 contrib 模块包括许多用户函数（如数组迭代器、模糊串匹配、加密函数），基本类型（如加密的密码，ISBN/ISSN，*n* 维立方体）以及索引扩展（如 RD 树，针对分级标记的索引）。由于 PostgreSQL 的开源特性，有一个由 PostgreSQL 专家和爱好者组成的庞大社区正在积极地扩充 PostgreSQL，几乎每天都有所进展。扩充类型在功能上和内置类型是一样的（见 27.3.1.2 节），只是后者已经链接到服务器上，并在系统目录中预先注册好了。类似地，这也是内置函数和扩充函数之间的唯一区别。

27.3.3.1 类型

PostgreSQL 允许用户定义复合类型，枚举类型甚至新的基本类型。

复合类型的定义类似于表定义（事实上，后者隐含了前者）。独立的复合类型一般用于函数参数。例如，定义

```
create type city_t as (name varchar(80), state char(2));
```

允许函数接受和返回 *city_t* 的元组，即使没有表显式地包含这种类型的行。

枚举类型通过简单地列出值的名字，可以很容易地定义。下面的例子创建了一个枚举类型来表示一个软件产品的状态。

```
create type status_t as enum('alpha', 'beta', 'release');
```

在比较一个枚举类型的不同值时，所列出名字的顺序是很重要的。这对于如下语句是有用的： [1132]

```
select name from products
    where status > 'alpha';
```

它检索已经通过了 alpha 阶段的产品的名字。

可以直接向 PostgreSQL 中增加基本类型。在发布的 PostgreSQL 向导中的 complex.sql 和 complex.c 中可以找到一个例子。基本类型用 C 语言声明，例如：

```
typedef struct Complex {
    double x;
    double y;
} Complex;
```

下一步是定义以文本格式读写新类型值的函数（见 27.3.3.2 节）。随后，新的类型可以用如下语句注册：

```
create type complex (
    internallength = 16,
    input = complex_in,
    output = complex_out,
    alignment = double
);
```

假设文本 I/O 函数已经注册为 *complex_in* 和 *complex_out*。

用户也可以选择定义二进制 I/O 函数（为了更高效的数据转储）。扩展类型可以像 PostgreSQL 中已有的基本类型一样使用。事实上，它们唯一的不同在于扩展类型是动态加载并链接到服务器上的。此外，索引也很容易扩展以处理新的基本类型，请参考 27.3.3.3 节。

27.3.3.2 函数

PostgreSQL 允许用户定义在服务器上存储和执行的函数。PostgreSQL 也支持函数重载（也就是说，多个函数可用相同的名称声明，但是参数类型不同）。函数可写作普通 SQL 语句，也可以用几种过程语言（在 27.3.3.4 节讲述）来编写。最后，PostgreSQL 有一个应用编程接口用于添加用 C 语言编写的函数（在本节解释）。

用户自定义函数可以用 C 语言（或者一种具有兼容的调用约定的语言，例如 C++）编写。实际编码约定对于动态加载的用户自定义函数以及内部函数（它们被静态地链接到服务器中）而言本质上是一
[1133] 样的。因此，标准内部函数库是用户自定义 C 函数的编程举例的丰富资源。一旦包含某函数的共享库已经产生，类似如下的一个声明会将其注册到服务器上：

```
create function complex_out(complex)
    returns cstring
    as 'shared_object_filename'
    language C immutable strict;
```

共享对象文件的入口点假设与 SQL 函数名相同（这里是 *complex_out*），除非另外特别说明。

这里继续 27.3.3.1 节中的那个例子。应用编程接口隐藏了 PostgreSQL 的大部分内部细节。因此，上述 *complex* 值的文本输出函数的实际 C 代码非常简单：

```
PG_FUNCTION_INFO_V1(complex_out);
Datum complex_out(pg_function_args)
    Complex *complex = (Complex *) pg_getarg_pointer(0);
    char *result;
    result = (char *) palloc(100);
    snprintf(result, 100, "(%g,%g)", complex->x, complex->y);
    pg_return_cstring(result);
}
```

第一行声明了 complex_out 函数，之后的几行实现了这个输出函数。代码中使用了几个 PostgreSQL 特有的结构，比如 palloc 函数，它用于动态地分配由 PostgreSQL 内存管理器支配的内存。更多的细节可以在 PostgreSQL 的文档中找到（参见文献注解）。

PostgreSQL 中的聚集函数执行时，利用**状态转换**（state transition）函数更新**状态值**（state value）。对于聚集组中的每个元组值，状态转换函数都会被调用。例如，avg 操作符的状态由累加和以及值的计数组成。每当一个元组到达时，转换函数简单地将它的值增加到累加和中，同时计数增加 1。有时需

要调用一个终结函数基于状态信息计算出返回值。例如，对于 **avg** 的终结函数将简单地用累加和除以计数，并返回结果。

因此，定义一个新的聚集就如同定义这两个函数一样简单。以 *complex* 类型为例，如果 *complex_add* 是一个用户自定义函数，接收两个复杂参数并返回它们的和，那么 **sum** 聚集运算符可以用如下简单的声明来扩展以支持复杂类型的数字：

1134

```
create aggregate sum (
    sfunc = complex_add,
    basetype = complex,
    stype = complex,
    initcond = '(0, 0)'
);
```

注意函数重载的使用：PostgreSQL 将根据函数调用时参数的实际类型来调用适当的 *sum* 聚集函数。这里的 *basetype* 是参数类型，而 *stype* 是状态值类型。在这个例子中不需要终结函数，因为返回值就是状态值本身（也就是两个例子中的累加和）。

使用操作符语法也能调用用户自定义函数。除了用于函数调用的简单的"语法上的甜头"外，运算符声明也为查询优化器提供了提高性能的线索。这些线索可能包括的信息有可交换性、约束、连接选择性估计以及与连接算法相关的各种其他属性。

27.3.3.3 索引扩展

PostgreSQL 当前支持内置的 B 树和散列索引，还有两种 PostgreSQL 特有的索引方法：通用搜索树（GiST）和通用倒排索引（GIN），这对于全文索引是有用的（27.5.2.1 节将对这些索引结构进行解释）。最后，PostgreSQL 提供了对二维空间对象的 R 树索引，它背后是通过使用 GiST 索引来实现的。所有这些索引都很容易扩展来适应新的基本类型。

为一个类型添加索引扩展需要定义一个**操作算子类**（operator class），它封装如下信息：

- **索引方法策略**（index-method strategy）。在 **where** 子句中可以用一组运算符来作为限定词。特定的运算符集合取决于索引类型。例如，B 树索引能够检索对象的范围，因此该集合由五个运算符组成（<，<=，=，>= 和 >），所有运算符都可以出现在 **where** 子句中，同时包含一个 B 树索引。散列索引只允许相等性测试，而 R 树索引支持许多空间关系运算（如包含、在左边等等）。
- **索引方法支撑例程**（index-method support routine）。上述运算符集合通常不足以支持索引的操作。例如，散列索引需要一个函数来计算每个对象的散列值。R 树索引需要能够计算交集和并集，并估计索引对象的大小。

1135

例如，如果定义下述函数和运算符来比较两个复数值（见 27.3.3.1 节）的大小，那么我们可以通过下面的声明来使得这种对象可被索引：

```
create operator class complex_abs_ops
  default for type complex using btree as
        operator 1 <(complex, complex),
        operator 2 <=(complex, complex),
        operator 3 =(complex, complex),
        operator 4 >=(complex, complex),
        operator 5 >(complex, complex),
        function 1 complex_abs_cmp(complex, complex);
```

operator 语句定义了策略方法，而 **function** 语句定义了支撑方法。

27.3.3.4 过程化语言

存储函数和过程可以用许多过程化语言编写。此外，PostgreSQL 定义了一个应用编程接口，用于连接用于此目的的任意编程语言。编程语言可以按需注册，要么是可信的（trusted），要么是不可信的（untrusted）。后者允许对 DBMS 和文件系统进行非受限访问，同时用它们编写存储函数需要超级用户权限。

- **PL/pgSQL**。它是一种将过程化编程能力（如变量和控制流）添加到 SQL 中的可信语言，和

Oracle 的 PL/SQL 很相似。尽管不能逐字地将一种语言的代码变化为另一种，但移植通常是比较简单的。

- **PL/Tcl、PL/Perl 和 PL/Python**。这些语言利用了 Tcl、Perl 和 Python 的能力来编写服务器上的存储函数和过程。虽两种既有可信的版本也有不可信的版本（分别为 PL/Tcl、PL/Perl 和 PL/TclU、PL/PerlU），而 PL/Python 在本书写作时是不可信的。每种这样的语言都有允许通过特定语言接口来访问数据库系统的绑定。

27.3.3.5 服务器编程接口

服务器编程接口（Server Programming Interface，SPI）是一种应用编程接口，它允许用户自定义的 C 函数（见 27.3.3.2 节）在其函数内部运行任意的 SQL 命令。这使得编写用户自定义函数的用户只需要
1136 用 C 语言实现必要的部分，而很方便地利用关系数据库系统引擎的强大功能来完成大部分工作。

27.4 PostgreSQL 中的事务管理

PostgreSQL 中的事务管理同时使用快照隔离和两阶段锁协议。采用哪种协议取决于所执行语句的类型。对于 DML 语句[1]使用 15.7 节中讲述的快照隔离技术，这个快照隔离方案被认为是 PostgreSQL 中的多版本并发控制方案（MVCC）。另一方面，DDL 语句的并发控制基于标准的两阶段锁协议。

27.4.1 PostgreSQL 的并发控制

因为 PostgreSQL 使用的并发控制协议取决于应用所需要的隔离性级别（isolation level），我们就从描述 PostgreSQL 所提供的隔离性级别开始。然后我们描述隐藏在 MVCC 方案背后的关键思想，紧接着讨论 PostgreSQL 中 MVCC 的实现，以及 MVCC 的一些推论。我们以描述 DLL 语句的锁机制和讨论索引的并发控制来结束本节内容。

27.4.1.1 PostgreSQL 隔离性级别

SQL 标准定义了三种弱一致性级别以及一致性的可串行化级别，本书的大部分讨论都基于此。一些应用并不需要达到可串行化所提供的强一致性保证，提供弱一致性级别的目的就是允许这些应用具有更高的并发度。此类应用的例子包括收集数据库的统计信息并且不需要精确结果的长事务。

根据违反可串行化的三种现象，SQL 标准定义了不同的隔离性级别。这三种现象称作：读脏数据、不可重复读和读幻象，定义如下：

- **读脏数据**（dirty read）。事务读了由另一个尚未提交事务所写的值。
- **不可重复读**（nonrepeatable read）。一个事务在执行过程中对同一对象读了两次，第二次得到了不同的值，尽管在此期间该事务并没有改变其值。
- **读幻象**（phantom read）。事务重新执行返回结果为满足某搜索条件的行集合的查询，发现满足条件的行集合已经改变，这是由于另一个事务最近提交了。（有关幻象现象更详细的解释，包
1137 括幻象冲突的概念，见 15.8.3 节；消除幻象读不一定就能消除所有的幻象冲突。）

很显然上述每种现象都破坏了事务的隔离性，因此违反了可串行化。图 27-5 显示了 SQL 标准中所指定的四种 SQL 隔离性级别的定义——读未提交、读已提交、可重复读和可串行化——根据上面这些现象所进行的分类。PostgreSQL 支持四种不同隔离性级别中的两种：读已提交（它是 PostgreSQL 中默认的隔离性级别）和可串行化。然而，PostgreSQL 使用快照隔离实现了可串行化的隔离性级别，如同我们之前在 15.7 节所看到的那样，这并不真正确保可串行化。

---

[1] 一条 DML 语句是在一个表中更新或读数据的任何语句。也就是 select、insert、update、fetch 和 copy。DDL 语句影响整个表；例如，它们可以创建一个表或改变一个表的模式。DDL 语句和其他一些 PostgreSQL 特定的语句将在本节后面讨论。

隔离性级别	读脏数据	不可重复读	幻像
读未提交	可能	可能	可能
读已提交	不可能	可能	可能
可重复读	不可能	不可能	可能
可串行化	不可能	不可能	不可能

图 27-5 四种标准 SQL 隔离性级别的定义

27.4.1.2 DML 命令的并发控制

PostgreSQL 中使用的 MVCC 方案是我们在 15.7 节中看到的快照隔离协议的一种实现。MVCC 背后的关键思想是维护每一行的不同版本，而不同版本对应着在不同的时间点上该行的实例。这就允许事务通过选择每一行在得到快照之前提交的最近版本，来查看数据的一致性快照。MVCC 协议使用快照来确保每个事务看到数据库的一致性视图：在执行命令之前，事务选择数据的一个快照，处理在该快照中或是被同一事务的更早命令所创建的行版本。由于完全从整个事务来考虑，数据的视图是"一致"的，但快照没有必要和数据的当前状态相同。

使用 MVCC 的动机是让读者从不阻塞写者，反之亦然。读者访问行的最近版本，它是该事务快照的一部分。写者创建它们自己的单独的行拷贝，用于更新。27.4.1.3 节显示只有当两个写者试图更新相同行时，才会出现使得事务阻塞的冲突。相反，在标准的两阶段锁方式下，读者和写者都可能被阻塞，因为每个数据对象只有一个版本，读操作和写操作在访问任何数据前都需要获得相应的锁。

PostgreSQL 中的 MVCC 方案实现了快照隔离协议的最先更新者获胜(first-updater-wins)版本，通过在对行进行写时获取互斥锁，但对行进行读时使用快照(不用任何锁)。如同前面在 15.7 节中概述的那样，在获得互斥锁之后进行附加验证。 [1138]

27.4.1.3 PostgreSQL 中的 MVCC 实现

PostgreSQL MVCC 的核心概念是元组可见性(tuple visibility)。PostgreSQL 的元组指行的一个版本。元组可见性定义了在给定语句或事务的上下文中，表中行潜在的多个版本中哪个是有效的。基于在执行命令前所选择的数据库快照，事务决定了元组的可见性。

如果满足如下两个条件，一个元组对于事务 T 是可见的：

1. 元组被一个事务创建，该事务于事务 T 得到快照之前提交；
2. 该元组的更新(如果存在)由一个事务执行，而该事务要么：
   - 被中止，或者
   - 在事务 T 得到快照之后开始运行，或者
   - 在 T 得到快照时已经是活跃的。

更准确地说，如果一个元组是由 T 创建的并在其后没有被 T 更新，则对 T 也是可见的。为了简单起见，我们忽略了这种特殊情况的细节。

上述条件的目标是为了确保每个事务看到的都是数据的一致性视图。PostgreSQL 维护如下状态信息来高效地检查这些条件：

- *事务标识*(transaction ID)，在事务启动时分配给每个事务，同时也被当作时间戳。PostgreSQL 使用逻辑计数(见 15.4.1 节中的描述)来分配事务标识。
- 一个叫做 *pg_clog* 的日志文件，包含每个事务的当前状态。状态分为：处理中，已提交或已中止。
- 表中每个元组都有一个元组头，包括三个域：*xmin*，包含创建该元组的事务标识，因此又称为*创建事务标识*(creation-transaction ID)；*xmax*，包含替换或删除该元组的事务标识(如果没有删除或替换则为 *null*)，也称为*终止事务标识*(expire-transaction ID)；一个指向相同逻辑行的新版本的前向链接(如果存在新版本的话)。
- 一个 *SnapshotData* 数据结构要么在事务启动时创建，要么在查询启动时创建，这取决于隔离性级别(下面有更详细的阐述)。它的主要目的是决定一个元组对当前命令是否可见。*SnapshotData* 存储关于一个事务在创建时的状态的信息，包括一个活跃事务的列表和 *xmax*，其

值等于 1 加上到目前为止已经启动的事务中的最高标识。

xmax 的值被当作是事务可能被看作是可见的一个"截止"。

图 27-6 通过一个简单的例子演示了这种状态信息，包括一个只有一个表的数据库，即图 27-7 中的 *department* 表。*department* 表有三列：系的名称、系所在的建筑和系的预算。图 27-6 展示了 *department* 表的一小段，只含有对应于 Physics 系的那些行(的版本)。元组头中的信息暗示着该行最早由事务 100 创建，后来被事务 102 和事务 106 更新。图 27-6 还显示了相应 *pg_clog* 文件的一小段。根据 *pg_clog* 文件，事务 100 和 102 已提交，而事务 104 和 106 正在处理中。

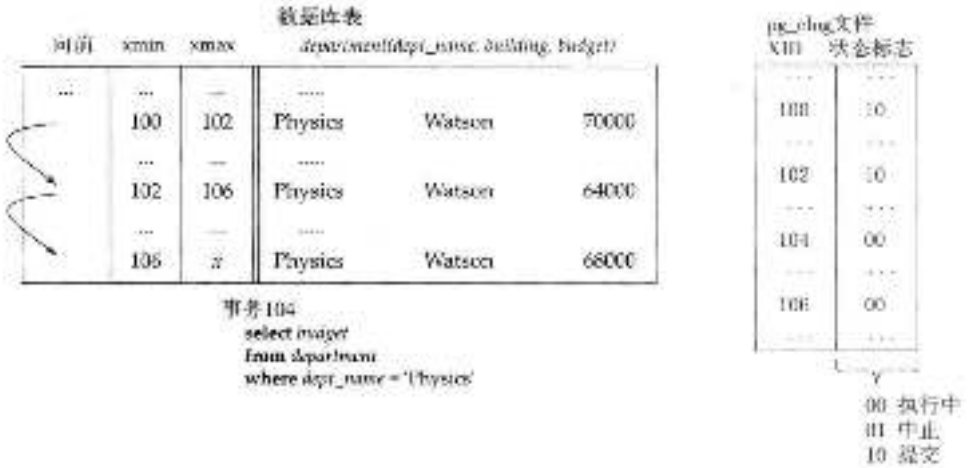


图 27-6 PostgreSQL 用于 MVCC 的数据结构

给定上述状态信息，对于一个元组可见的动个需要满足的条件可以重写如下：

1. 元组头中的创建事务标识
   a. 根据 *pg_clog* 文件来看是已提交事务，并且
   b. 小于被 *SnapshotData* 记录在 *xmax* 中的截止事务标识，并且
   c. 不是 *SnapshotData* 中存储的活跃事务之一。
2. 终止事务标识，如果它存在
   a. 根据 *pg_clog* 文件来看是中止事务，或者
   b. 大于或等于被 *SnapshotData* 记录在 *xmax* 中的截止事务标识，或者
   c. 存储在 *SnapshotData* 中的活跃事务之一。

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

图 27-7 *department* 关系

考虑图 27-6 中的数据库实例，假设事务 104 用到的 *SnapshotData* 简单地使用 103 作为截止事务标识 *xmax*，并且没有出现更早的活跃事务。在这种情况下，对应 Physics 系的那一行只有一个版本对事务 104 是可见的，那就是表中由事务 102 所创建的第二个版本。由事务 100 所创建的第一个版本不可见，因为它破坏了条件 2：这个元组的终止事务标识是 102，这是一个并未中止的事务，并且它的事务标识小于或等于事务 103。Physics 元组的第三个版本是不可见的，因为它由事务 106 产生，它的事务标识大于事务 103，这意味着在 *SnapshotData* 被创建时该版本尚未提交。此外，事务 106 正在运行中，这破坏了另一个条件。该行的第二个版本满足元组可见性的所有条件。

PostgreSQL MVCC 同 SQL 语句执行之间的具体交互细节取决于该语句是 **insert**、**select**、**update** 还是 **delete** 语句。最简单的情况是 **insert** 语句，它基于语句中的数据简单地创建一个新元组，初始化元组头(创建标识)，然后把这个新元组插入到表中。与两阶段锁的情况不同的是，这不需要与并发控制协议进行任何交互，除非插入需要检查完整性条件，如：唯一性或者外码约束。

当系统执行 **select**、**update** 或 **delete** 语句时，与 MVCC 协议的交互取决于应用所指定的隔离性级别。如果隔离性级别是读已提交，那么新语句的处理从创建一个新的 *SnapshotData* 数据结构开始(与

该语句是创建一个新事务，还是它只是现有事务的一部分无关）。接着，系统识别目标元组，就是关于 *SnapshotData* 可见且匹配语句搜索条件的那些元组。对于 **select** 语句，目标元组集构成查询的结果。

对于读已提交模式下的 **update** 或 **delete** 语句的情况，识别目标元组后，在实际更新或删除操作发生前，还需要一个额外的步骤。原因是元组的可见性保证了只有那些在进行中的 **update/delete** 语句之前已提交的事务所创建的元组才启动。然而，可能自查询开始以来，该元组已被另一个并发执行的事务更新或删除。通过查看该元组的终止事务标识可以检测到这种情况。如果终止事务标识对应着一个正在处理的事务，有必要先等待该事务完成。如果该事务中止，**update** 或 **delete** 语句继续进行并执行实际的修改。如果该事务提交，**update/delete** 语句的搜索条件需要重新计算，只有元组仍然满足这些条件，该行才可以修改。如果是对行的删除，则主要的步骤是更新旧元组的终止事务标识。行的更新也同样执行这个步骤，外加还会创建行的一个新版本，设置它的创建事务标识，同时把旧元组的前向链接指向新元组。

1139
1141

回到图 27-6 中的例子，假由一个 **select** 语句组成的事务 104 识别 Physics 行的第二个版本为目标元组并将其立即返回。如果事务 104 是一条更新语句，比如试图让 Physics 系的预算增加一些，那么它将必须等待事务 106 完成。随后它重新计算搜索条件，只有该条件仍然满足时更新才会进行。

使用上面描述的用于 **update** 和 **delete** 语句的协议只提供了读已提交隔离性级别。在某些方面会违反可串行化。首先，不可重复读是可能的。因为事务中的每个查询可能会看到数据库的不同快照，那么事务中的一个查询可能会看到已完成的 **update** 命令的结果，与此同时，相同事务中更早的查询却看不到。同理，当一个关系在查询之间被修改时，读幻象也是可能的。

为了提供 PostgreSQL 的可串行化隔离性级别，PostgreSQL MVCC 用两种方式来避免违反可串行化：首先，当确定元组的可见性时，事务内的所有查询都使用事务启动时的快照，而不是单个查询启动时的快照。这样事务中的后续查询总是看到相同的数据。

其次，在可串行化模式中更新和删除的处理方式同读已提交模式不同。同读已提交模式一样，当识别出一个满足搜索条件的可见的目标行，并且该行正被另一个并发事务所更新或者删除，则事务将等待。如果执行更新或者删除的并发事务中止，等待事务可以继续它自己的更新。然而，如果并发事务提交了，PostgreSQL 没法保证等待事务的可串行性。因此，等待事务将回滚并返回错误信息"由于并发更新，不能串行化访问"。

如何适当地处理像上面那样的错误消息取决于应用，可以中止当前事务并从起点开始重启整个事务。注意由于可串行化问题带来的回滚只可能出现在 **update** 和 **delete** 语句中，**select** 语句还是一样的，从不与任何其他事务冲突。

1142

27.4.1.4 使用 MVCC 的推论

使用 PostgreSQL MVCC 方案有三个不同方面的推论：(1)给存储系统带来了额外的负担，因为需要维护元组的不同版本；(2)开发并发应用需要更小心一些，因为与使用标准的两阶段封锁的系统相比，PostgreSQL MVCC 可能导致并发事务运行方式上微妙的但很重要的不同。(3)PostgreSQL 的性能取决于运行在其之上的工作负载的特点。下面将对 PostgreSQL MVCC 的推论进行更详细的描述。

创建和存储每行的多个版本会带来明显的存储开销。为了缓解这个问题，PostgreSQL 在可能的时候释放空间，识别和删除那些对于任何活跃的和未来的事务不再可见的，因而也不再需要的元组的版本。释放空间的任务是不可忽视的，因为索引可能指向不再需要的元组的位置，所以在重用这些空间之前需要删除这些引用。为了减轻这个问题，PostgreSQL 避免对有相同索引属性的元组的多个版本建立索引。这就允许任何发现了非索引元组的事务有效地释放被该元组所占用的空间。

为了进一步重用空间，PostgreSQL 提供了 **vacuum** 命令，它可以为每个被释放的元组正确地更新索引。PostgreSQL 利用一个后台进程来自动地**清扫**(vacuum)表。但是这个命令也可以直接由用户来执行。**vacuum** 命令提供两种操作模式：普通的 **vacuum** 简单地识别不再需要的元组，并使得这些空间可以重用，这种形式的命令可以与表的普通读写并行执行；**vacuum full** 命令进行更为广泛的处理，包括在块之间移动元组以试图把表压缩到最少数量的磁盘块上，这种形式会慢许多，同时在每个正在处理的表

上都需要一个排他锁。

由于 PostgreSQL 中使用了多版本并发控制，从其他环境中将应用程序移植到 PostgreSQL 中需要更加小心，以确保数据的一致性。举个例子，考虑执行一条 **select** 语句的事务 T_1。既然 PostgreSQL 中的读者不需要封锁数据，当 T_1 仍在运行时，另一个并发事务 T_2 可以覆盖 T_1 所读取和选择的数据。其结果是，一些由 T_1 返回的数据在 T_1 完成时可能不再是当前值了。T_1 也可能返回那些同时已被其他事务改变和删除的行。为了确保某行当前的有效性并防止它被并发更新，应用程序必须使用 **select for share**，或者通过恰当的 **lock table** 命令来显式地获得封锁。

对于包含的读操作比更新操作多得多的工作负载来说，PostgreSQL 的并发控制方法性能最好。因
1143 为在这种情况下，两个更新相冲突，进而导致事务回滚的几率极低。对于更新密集的工作负载来说，两阶段锁可能更有效，但这依赖于许多因素，如事务长度和死锁频率。

27.4.1.5 DDL 并发控制

前一节所描述的 MVCC 机制并未保护事务与影响到整个表的操作不冲突，例如删除表或者改变表模式的那些事务。为了这个目标，PostgreSQL 提供了显式的锁，强制要求 DDL 命令在执行之前获得这些锁。这些锁是基于表的（而不是基于行的），并且获取和释放的规则同严格的两阶段封锁协议一致。

图 27-8 列出了 PostgreSQL 所提供的所有锁类型、与之相冲突的锁以及使用它们的一些命令（其中 **create index concurrently** 命令在 27.5.2.3 节介绍）。锁类型的取名通常是基于历史的，未必反映锁的使用情况。例如，所有的锁都是表级锁，尽管一些锁名字中包含单词“行”。DML 命令只获取前三类的锁。这三类锁相互兼容，因为 MVCC 已经小心地保护这些操作互不冲突。DML 命令获取这些锁仅仅只是为了保护与其他 DDL 命令不冲突。

它们的主要目的是为 DDL 命令提供 PostgreSQL 内部的并发控制，而 PostgreSQL 应用程序也可以通过 **lock table** 命令来显式地获得图 27-8 中的所有锁。

锁名称	与之冲突的锁	需要该锁的操作
ACCESS SHARE	ACCESS EXCLUSIVE	select (query)
ROW SHARE	EXCLUSIVE ACCESS EXCLUSIVE	select for update (query) select for share (query)
ROW EXCLUSIVE	SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	update delete insert (query)
SHARE UPDATE EXCLUSIVE	SHARE UPDATE EXCLUSIVE SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	vacuum analyze create index concurrently
SHARE	ROW EXCLUSIVE SHARE UPDATE EXCLUSIVE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	create index
SHARE ROW EXCLUSIVE	ROW EXCLUSIVE SHARE UPDATE EXCLUSIVE SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	
EXCLUSIVE	All except ACCESS SHARE	
ACCESS EXCLUSIVE	All modes	drop table alter table vacuum full

图 27-8 表级锁模式

锁被记录在锁表中，锁表被实现为一个放在共享内存中的散列表，以标识被封锁对象的签名作为关键词。如果事务想获得某对象上的锁，而该锁被另一个事务以冲突模式所持有，则它需要等待锁的释放。锁等待用信号量实现。每个信号量都有唯一一个事务与之关联。当等待一个锁时，事务实际上

在与持有该锁的那个事务相关联的信号量之上等待。一旦锁持有者释放该锁，它将通过信号量通知(多个)等待事务。通过基于每执行者每锁实现就等待，而不是基于每对象每锁，PostgreSQL对于每个并发事务最多获取一个信号量，而不是每个可封锁对象一个信号量。

PostgreSQL中的死锁检测是基于超时机制的。默认情况下，如果某事务等待一个锁超过1秒就会触发死锁检测机制。死锁检测算法基于锁表中的信息构造一个等待图，并搜索此图中的循环依赖。如果发现了任何循环依赖，则意味着检测到了死锁，就发死锁检测的事务会中止并给用户返回一个错误。如果没有检测到环，则事务会在锁上继续等待。不同于一些商用系统，PostgreSQL并不动态调整锁超时参数，但是它允许管理员手动调整。理想情况下，这个参数的选择应该与事务生命周期相似，以优化死锁检测所耗费的时间和不存在死锁时运行死锁检测算法所空耗的工作之间的权衡。

27.4.1.6 封锁和索引

在PostgreSQL中，所有当前实现的索引允许被多个事务并发访问。这通过页级封锁通常是可能的，所以不同事务如果不请求同一页上的冲突锁就可以并行地访问索引。这些锁通常被短时间持有以避免死锁，除了被列索引之外，被列索引封锁页面时间更长，并可能陷入死锁。

27.4.2 恢复

历史上，PostgreSQL并没有将先写日志(WAL)用于恢复。因此在发生崩溃时不能保证一致性。崩溃将在也会导致不一致的索引结构，更糟糕会完全破坏表内容。因为只写了部分数据页。因此，从版本7.1开始，PostgreSQL采用了基于WAL的恢复机制。此方法类似于标准的恢复技术，比如ARIES(见16.8节)，但PostgreSQL中的恢复由于MVCC协议在某些方式上简化了。

1144
|
1145

首先，在PostgreSQL中，恢复不是必须撤销中止事务的影响：一个正中止的事务在*pg_clog*文件中登记一项，记录下它正中止的事实。结果，它创建的行的所有版本对任何其他事务都再不可见。这种方法唯一可能导致问题的情况是，某事务因相应的PostgreSQL进程崩溃而中止，而PostgreSQL进程在崩溃前没有机会创建*pg_clog*项。PostgreSQL按如下方式处理此问题：在检查*pg_clog*文件中另一事务的状态之前，它检查该事务是否运行在某个PostgreSQL进程中。如果当前没有任何PostgreSQL进程运行该事务，并且*pg_clog*文件表明事务还在运行中，则可以安全地假定事务已崩溃。同时更新事务的*pg_clog*项为"已中止"。

其次，恢复被简化了是由于这样的事实：PostgreSQL MVCC已经记录了WAL日志所需要的某些信息。更准确地说，没有必要在日志中记录事务的启动、提交和中止，因为MVCC在*pg_clog*中记录了每个事务的状态。

27.5 存储和索引

PostgreSQL的数据布局和存储的方法是为了实现以下目标：(1)实现简洁(2)易于管理。为了实现这样的目标所采取的一个措施是，PostgreSQL依赖于"已建立的"文件系统，而并非自己处理在原始磁盘分区上数据的物理布局。PostgreSQL在文件层次结构中维护了一个目录列表用于存储，按照惯例被称为表空间。每个PostgreSQL在安装时都初始化一个默认的表空间，新增的表空间可以在任何时候添加。当创建表、索引或整个数据库时，用户可以指定任何已有的表空间来存放相关的文件。创建驻留在不同物理设备上的多个表空间特别有用，这样速度更快的设备可以专用于存放有更高需求的数据。此外，存放在分离磁盘上的数据可以被更高效地并行访问。

由于PostgreSQL和文件系统之间的冲突，PostgreSQL存储系统的设计潜在地导致了一些性能上的局限。使用已建立的文件系统导致了双缓冲，从磁盘取出的块首先被放到文件系统的缓存(在内核空间)中，然后才会复制到PostgreSQL的缓存池。性能也会因为PostgreSQL在8KB的块中存放数据而受到限制，这可能和内核使用的块大小不匹配。在服务器安装的时候可以改变PostgreSQL的块大小，但这可能带来不利影响的结果：小的块限制了PostgreSQL有效存储大元组的能力，而当只有一小部分文件被访问时，大的块也是浪费的。

1146

另一方面，现代企业越来越多地使用外部存储系统，比如网络附加存储和存储区域网络，而不是与服务器相连的磁盘。此处的观点认为存储是一种服务，能够很容易地根据性能来单独管理和调试它。

这些系统使用的一种方法是RAID，如同10.3节介绍的那样，它同时提供并行和冗余存储。因为依赖于已建立的文件系统，PostgreSQL可以直接利用这些技术。因此，许多PostgreSQL开发者的感觉是，对于PostgreSQL用户的绝大多数应用而言，与管理的轻松和实现简单相比，性能上的限制是微不足道和合理的。

27.5.1 表

PostgreSQL中的基本存储单元是表。在PostgreSQL中，表存储在单文件中。这些文件采用10.5节中介绍的一种标准的分槽的页(slotted-page)格式。PostgreSQL的格式如图27-9所示。在每一页中，一个头(header)后都跟有一组"行指针"。一个行指针存储了该页中一个特定元组的偏移量(相对于每一页的开始位置)和长度。实际的元组是从每一页末尾开始，以与行指针相反的顺序存储。

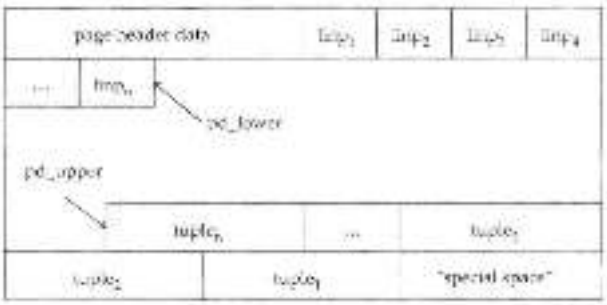


图27-9 PostgreSQL表的分槽的页格式

堆文件中的记录通过它的元组标识符(TID)加以标识。TID包括一个4字节的块ID(块ID指明了包含此元组的文件页)和一个2字节的槽ID。槽ID是对行指针数组的索引，通过它可以访问到该元组。

尽管这种架构允许同一页中添加或删除元组，基于PostgreSQL的MVCC方式，这些操作都不会真正地立即删除或替换行的旧版本。如同在27.4.1.4节中解释的，过期元组可以通过之后的命令来物理
[1147] 删除，从而在该页中形成空洞。通过行指针数组来间接访问元组的策略使得这些空洞可以重用。

元组的长度通常受数据页长度的限制。这使得要存储非常长的元组很困难。当PostgreSQL遇到如此大的元组时，它尽力"压缩"(toast)单个的大属性。在有些情况下，缩减一个属性可能通过对值的压缩来实现。如果这还不足以使元组缩减到可以放入页中(通常的情况)，则被缩减属性中的数据被一个引用所取代，它指向数据在该页之外的拷贝存储。

27.5.2 索引

PostgreSQL索引是一种数据结构，提供从搜索谓词到特定表的元组标识序列的动态映射。被返回的元组倾向于匹配搜索谓词，虽然在有些情况下谓词必须在堆文件中复查。PostgreSQL支持几种不同的索引类型，包括那些基于用户可扩展的访问方法的索引。尽管一种访问方法可能使用不同的页格式，PostgreSQL中的所有可用索引都使用在上面27.5.1节中介绍的分槽的页格式。

27.5.2.1 索引类型

PostgreSQL支持如下几种类型的索引：

- **B树**。默认索引类型是一种基于Lehman和Yao提出的B-link树的B+树索引(15.10节中介绍了支持高并发性操作的B-link树)。该类索引能有效支持对于有序数据的等值查询和范围查询，以及特定的模式匹配操作，如like表达式。
- **散列**。PostgreSQL的散列索引是线性散列的一种实现(想了解散列索引的更多内容，请参见11.6.3节)。该类索引仅对简单的等值操作有用。PostgreSQL使用的散列索引并未显示出比B树更好的查找性能，但它还需要更多的空间开销和维护代价。另外，散列索引是PostgreSQL中唯一不支持灾难恢复的索引。所以相对于散列索引而言，人们几乎总是更愿意使用B树索引。
- **GiST**。PostgreSQL支持一个高可扩展性索引，称作GiST，或者通用搜索树。GiST是一种平衡的树状结构的访问方法，使得精通特定数据类型(如图像数据)的领域专家可以无需关注数据库系

统的内部细节就可以容易地开发出增强性能的索引。使用 GiST 构建的一些索引实例包括 B 树和 R 树，还有用于多维立方体和全文检索的不那么传统的索引。可以通过创建如 27.3.3.3 节中介绍的操作算子类来实现一个新的 GiST 访问方法。GiST 的操作算子类不同于 B 树，每个 GiST 操作算子类可以有一个不同的策略集合来表示通过索引实现的搜索谓词。GiST 还依赖七个支撑函数来进行操作，如测试集合成员资格和用于页溢出的条目集分裂。 1148

有趣的是 PostgreSQL 原本实现的 R 树（见 25.3.5.3 节）在 8.2 版中被 GiST 算子类所取代。这使得 R 树可以利用 WAL 日志和在 8.1 版中添加到 GiST 中的并发能力的优势。因为原来实现的 R 树没有这些特点，这种改变显示了可扩展索引方法的好处。更多关于 GiST 索引的信息请参考文献注释中的文献。

- **GIN**。PostgreSQL 中索引的最新类型是通用倒排索引（GIN）。一个 GIN 索引把索引关键字和搜索关键字都当作集合，使得索引类型适合于面向集合的操作。GIN 的一个预期的应用是为全文搜索对文档进行索引。通过把文档和查询规约为搜索项的集合来实现。类似于 GiST，通过创建有合适支撑函数的操作算子类，GIN 索引可以扩展到处理任何比较运算。

  为了评估一个搜索，GIN 有效地识别和搜索关键字成员的索引关键字，并计算一个位图表明哪些被搜索元素是索引关键字的成员。这实现了对支撑函数的使用，该支撑函数从一个集合中提取成员并比较单独成员。基于位图和查询的断言，另一个支撑函数基于位图和初始谓词决定是否满足搜索谓词。如果没有完全索引属性就不能决定搜索谓词，则判定函数必须报告一个匹配并复查在堆文件中的谓词。

27.5.2.2 其他索引变形

对于上述某些索引类型，PostgreSQL 还支持更复杂的变形，比如：

- **多列索引**。该类索引对于定义在一个表中多个列上的谓词的合取很有效。多列索引只支持 B 树和 GiST 索引。
- **唯一性索引**。在 PostgreSQL 中，唯一性和主码约束可通过使用唯一性索引来实现。只有 B 树索引能定义为唯一性的。
- **表达式上的索引**。在 PostgreSQL 中，可以在列的任意标量表达式上创建索引，并不局限于表中特定的列。当讨论中的表达式"代价特别大"时——比如包含复杂的用户自定义运算——这种索引特别有用。举例来说，通过在表达式 *lower*(*column*) 上定义索引，并在查询中使用谓词 *lower*(*column*) = '*value*'，就能支持不区分大小写的比较操作。但表达式上的索引的一个缺点是其维护成本高。 1149
- **操作算子类**。用于建立、维护和使用列上索引的特定比较函数，是与该列的数据类型密切相关的。每种数据类型都有与之关联的默认"操作算子类"（在 27.3.3.3 节中介绍），它标识了通常使用的实际操作。大多数情况下，默认操作算子类通常是够用的。一些数据类型还可能拥有多个"有意义"的类。例如，在处理复数时，可能需要对实部或虚部建立索引。对于那些不使用标准的、面向现场校对规则的文本数据上的模式匹配操作（例如 like 操作），PostgreSQL 提供了一些无置的操作算子类（换句话说，特定于语言的排序顺序）。
- **部分索引**。这类索引建立在表的一个子集上，该子集用谓词定义。该索引只包含对应于表中那些满足此谓词的元组的项。部分索引适用于列包含很少量的值，但又可能多次出现的情况。在这种情况下，常见的值就不值得索引了，因为对于需要显表大部分数据的查询来说索引扫描是无益的。而排除了常见值的部分索引很小，且只引发少量 I/O 操作。由于不需要对部分索引运行大量的插入操作，它的维护成本也较小。

27.5.2.3 索引构建

使用 **create index** 命令可以为数据库添加索引。例如，下面的 DDL 语句在教员工资上创建了一个 B 树索引。

```
create index inst_sal_idx on instructor (salary);
```

该语句的执行是通过扫描 *instructor* 关系，找到可能对将来事务可见的行版本，然后把它们的索引属性

进行排序，并构建索引结构。在这个过程中，创建索引的事务持有 instructor 关系上的锁，防止并发的 **insert**、**delete** 和 **update** 语句。一旦这个过程完成，索引就已准备好可以使用了，表上的锁也被释放。

通过 **create index** 命令获得的锁对于某些应用来说可能会带来很大的不便，在这些应用中很难在创建索引时挂起更新操作。对于这种情况，PostgreSQL 提供了 **create index concurrently** 变体，它允许在构建索引时进行并发的更新。这由一个更复杂的、扫描两次基表的构建算法来实现。第一次表的扫描建立起索引的初始版本，类似于前面描述的普通索引的构建方式。如果表上有并发的更新，这个索引就可能缺失一些元组，但该索引具有良好的形式。所以它在准备好进行插入时会被标记。最后，算法第二次扫描表，插入其找到的还需要被索引的所有元组。这个扫描同样也会缺失并发更新的元组，但
[1150] 算法与其他事务同步以保证在第二次扫描中更新的元组，将由一个更新事务添加到索引中。因此，在第二次扫描表之后索引就可以使用了。由于这种两遍扫描的算法是耗时的，如果容易临时挂起表的更新，最好选择普通的 **create index** 命令。

27.6 查询处理和优化

当 PostgreSQL 收到一个查询请求时，它首先将其解析为一种内部表示，然后通过一系列的变换，得到能被执行器用于查询处理的一个查询计划。

27.6.1 查询重写

查询转换的第一个阶段是**重写**(rewrite)，这一步由 PostgreSQL 的规则系统负责。正如 27.3 节中所述，在 PostgreSQL 中，用户可以创建被不同事件(如 **update**、**delete**、**insert** 和 **select** 语句)所触发的规则。视图是系统通过将视图定义转换为 **select** 规则来实现的。当收到涉及视图上的 **select** 语句的查询请求时，有关该视图的 **select** 规则被触发，查询将根据视图的定义进行重写。

使用 **create rule** 命令将在系统中注册一条规则，此时该规则的信息存储在目录中。在进行查询重写时，这一目录将用于找出与给定查询相关的所有候选规则。

在重写阶段将首先处理所有的 **update**、**delete** 和 **insert** 语句，这通过触发所有相关规则来实现。这些语句可能很复杂，并包含 **select** 子句。然后，剩下的所有只涉及 **select** 语句的规则被触发。因为触发一条规则所引发的查询重写可能需要触发另外的规则，每次重写的查询形式都要反复地进行规则检查，直到不需要再触发规则为止。

在 PostgreSQL 中没有默认规则，只有用户显式定义的规则和视图定义中隐含的规则。

27.6.2 查询规划和优化

一旦查询被重写后，就进入到查询规划和优化阶段。在这个阶段，每个查询块被单独处理，并为之生成一个查询计划。这个规划过程由重写的查询的最内层子查询开始，自底向上产生，直至到达最外层的查询块。

在大多数情况下，PostgreSQL 中的优化器是基于代价的。其思想是为每个查询计划生成一个估计开销最小的访问计划。代价模型包括的参数有：顺序和随机页面存取的 I/O 代价，以及处理堆元组、索引元组和简单谓词的 CPU 开销。

[1151] 实际的优化过程是基于以下两种形式之一的：

- **标准规划器**(standard planner)。标准的规划器采用自底向上的动态规划算法来进行连接顺序的优化。在 IBM 研究机构于 20 世纪 70 年代开发的早期关系数据库系统 System R 中最早使用。13.4.1 节详细介绍了 System R 的动态规划算法，该算法每次用于一个单一的查询块上。
- **遗传查询优化器**(genetic query optimizer)。当查询块中的表数量很多时，System R 的动态规划算法代价会非常大。与其他默认使用贪心法或基于规则方法的商用系统不同，PostgreSQL 使用了一种更激进的方法：一种最初为了解决旅行商问题而开发出的遗传算法。有证据表明，遗传查询优化在具有涉及大约 45 个表的查询的生产系统中有成功应用。

由于规划器以自底向上的方式运行在查询块上，它能够在查询计划构建时进行一些转换工作。一个实例是在许多商用系统中常见的子查询-连接转换(通常在查询重写阶段实现)。当 PostgreSQL 遇到

一个非关联子查询(比如在一个视图上的查询所引发的子查询)时，通常可以把规划好的子查询"上移"，与顶层的查询块配合。但是，在 PostgreSQL 中将消除重复操作下推到更低层的查询块的转换通常不可能实现。

查询优化阶段产生了一个查询计划，它是一棵关系操作算子的树，每个操作算子代表在一个或多个元组集合上的一种特定操作。这些操作算子可以是一元的(比如排序、聚集)、二元的(比如嵌套循环连接)，或者 n 元的(比如集合的并)。

对于代价模型来说，关键是要精确估计出查询计划中每次操作所要处理的元组总数。这可以由优化器根据系统中为每个关系所维护的统计信息来推算。这些统计信息说明了每个关系的元组总数和关系的每个列的具体信息，比如列的基数(cardinality)，表中最常见的数值和该数值出现次数的列表，以及将列的值划分到若干有相同数量值的组中的直方图(即在 13.3.1 节中描述的等深直方图)。另外，PostgreSQL 还维护着列的值在物理行次序和逻辑行次序之间的统计相关性——它指出了利用索引扫描来检索出满足该列上谓词的元组的代价。数据库管理员通过周期性地运行 **analyze** 命令，来确保这些统计数据都是当前最新的。

27.6.3 查询执行器

执行模块负责处理优化器产生的查询计划。执行器遵循迭代器(iterator)模型，该模型为每个操作算子提供了四个实现函数(open、next、rescan 和 close)集。在 12.7.2.1 节中，将迭代器作为需求驱动流水线的一部分进行了介绍。PostgreSQL 的迭代器还支持一个额外的函数 rescan，它通过设置诸如索引扫描范围这样的参数来重置一个子计划(比如连接的内循环)。 1152

对执行器支持的重要的操作算子分类如下：

1. **访问方法**(access method)。PostgreSQL 中实际用于从磁盘对象中检索数据的访问方法是堆文件的顺序扫描、索引扫描和位图索引扫描。

- **顺序扫描**(sequential scan)。关系中的元组是从文件的第一块到最后一块顺序扫描。只有根据 27.4.1.3 节所介绍的事务隔离性规则是"可见的"元组才会返回给调用者。
- **索引扫描**(index scan)。给定一个搜索条件，如一个范围或等式谓词，这种访问方法会从相关的堆文件中返回匹配的元组集。该算子一次处理一个元组，开始是从索引中读取一个项，然后从堆文件中取相应的元组。在最坏情况下，这可能导致对于每个元组都要随机地取一次页面。
- **位图索引扫描**(bitmap index scan)。位图索引扫描减少了在索引扫描中过多的随机取页面的风险。这通过在两个阶段中处理元组来实现。第一个阶段读取所有的索引项并在一个位图中存储堆元组的标识。第二个阶段按顺序取出堆上的堆元组。这保证了每个堆页面只访问一次，并增加了顺序取页面的机会。另外，由多个索引建立的位图可以合并或求交，以便在访问堆之前计算复杂的布尔谓词。

2. **连接方法**(join method)。PostgreSQL 支持三种连接方法：排序归并连接、嵌套-循环连接(包括对于内层的索引嵌套循环变体)和混合散列连接(见 12.5 节)。

3. **排序**(sort)。PostgreSQL 中外排序通过 12.4 节介绍的算法来实现。输入被划分成若干非好序的归并段，然后这些归并段将被多路归并。初始归并段使用选择替换算法(replacement selection)来生成，然而 PostgreSQL 使用了优先级树，而不是固定了内存记录中数目的数据结构。这是因为 PostgreSQL 可以处理大小变化很大的元组，它努力确保对配置的排序内存空间的充分使用。

4. **聚集**(aggregation)。PostgreSQL 中的分组聚集要么是基于排序的，要么是基于散列的。当估计到不同分组的数目很大时，会使用基于排序的聚集，否则使用基于散列的聚集。

27.6.4 触发器和约束

在 PostgreSQL 中(不同于某些商用数据库系统)，在查询重写阶段没有实现诸如触发器和约束那样的动态数据库特性。相反，它们是作为查询执行器的一部分实现的。当用户注册触发器和约束时，它 1153
们的细节都与每一个相应的关系和索引的目录信息联系在一起。执行器通过对关系反复生成元组的修改，来执行 **update**、**delete** 和 **insert** 语句。对每一行的修改，执行器形式地在所需要的改变之前或其后

识别、触发和实施候选的触发器和约束。

27.7 系统结构

PostgreSQL 的系统结构遵循每事务每进程(process-per-transaction)模式。每一个运行中的 PostgreSQL 站点由一个集中式的、称作 **postmaster** 的协调进程来管理。这个 postmaster 进程负责初始化和关闭服务器，以及处理来自新客户端的连接请求。postmaster 为每个新连接的客户端分配一个后台服务器进程，该进程负责代表客户端执行查询，并将结果返回客户端。该系统结构如图 27-10 所示。

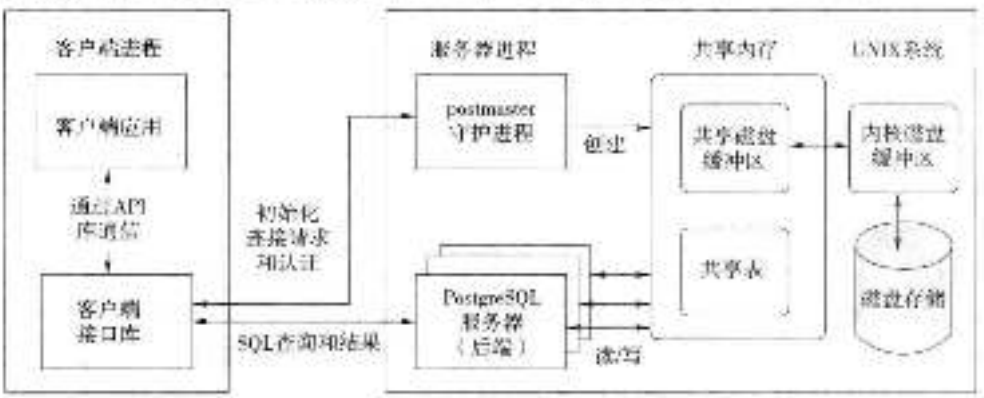


图 27-10 PostgreSQL 系统结构

客户端应用程序能够连接到 PostgreSQL 服务器，并通过 PostgreSQL 支持的多种数据库应用编程接口(如 libpq、JDBC、ODBC 和 Perl DBD)之一来提交查询请求。这些数据库应用编程接口是作为客户端库形式提供的。一个客户端应用程序的例子是包含在标准 PostgreSQL 发布里的命令行形式的 psql 程序。postmaster 负责处理初始的客户端连接。为此，它在一个已知端口上不断监听新的连接请求。在完成诸如用户认证这样的初始化步骤后，postmaster 将生成一个新的后台服务器进程来处理新客户端。在这种初始连接之后，客户端将只与后台服务器进程交互，提交查询请求并接收查询结果。这就是 PostgreSQL 采用的每连接每进程模式的核心。

后台服务器进程负责执行客户端提交的查询请求。执行一系列必要的查询步骤，包括解析、优化和执行。每个后台服务器进程一次只能处理一条查询。为了能够并发处理多条查询，应用程序必须维护与服务器的多个连接。

在任一给定时刻，可能有多个客户端连接到系统，因此可能有多个后台服务器进程并发执行。后台服务器进程通过主存缓存池来访问数据库中的数据，而该缓存池位于共享内存区，因此所有进程都有相同的数据视图。共享内存也用于实现服务器进程间的其他形式的同步，比如对数据项的封锁。

要使用共享内存作为通信媒介，就要求 PostgreSQL 服务器运行于一台机器上；在没有第三方软件包(如 Slony 复制工具)的帮助下，一个单服务器站点不能分散于多台机器上。但是，可以构建一个无共享的并行数据库系统，其中每个结点上都运行一个 PostgreSQL 实例；事实上，有几个商用并行数据库系统就是采用这种体系结构构建的，如 18.8 节所述。

文献注解

在 www.postgresql.org 上有大量有关 PostgreSQL 的在线文档。这个网站是关于 PostgreSQL 最新版本的权威的信息源，它定期更新有关 PostgreSQL 的内容。直到 PostgreSQL 版本 8 之前，在微软 Windows 平台上运行 PostgreSQL 的唯一方式是使用 Cygwin。Cygwin 是一个类似于 Linux 的环境，它允许从源程序开始重新构建 Linux 应用以运行于 Windows 之上。详细信息请参考 www.cygwin.com。有关 PostgreSQL 的书包括 Douglas 和 Douglas[2003]以及 Stinson[2002]。Stonebraker 等[1990]介绍了 PostgreSQL 所采用的规则。GiST 结构在 Hellerstein 等[1995]里有介绍。

PostgreSQL 的许多工具和扩展被 pgFoundry 文档化在 www.pgfoundry.org 上。其中包括 pgcl 库和本章提

到的 pgAccess 管理工具。pgAdmin 工具在网站 www.pgadmin.org 上有描述。数据库设计工具 TORA 和 Data Architect 分别在 tora.sourceforge.net 和 www.thekompany.com/products/dataarchitect 上有介绍。报表生成工具 GNU Report Generator 和 GNU Enterprise 分别在 www.gnu.org/software/gnu 和 www.gnuenterprise.org 上有介绍。开源的 Mondrian OLAP 服务器在 mondrian.pentaho.org 上有介绍。

除 PostgreSQL 之外，还有一个开源的 MySQL，它在 GNU 通用公共许可(General Public License)下可用于非商业应用。MySQL 可以嵌入到不免费发布源码的商业软件中，但这需要购买一个特别的许可证。关于这两个系统最新版本的比较在网上已有介绍。

1154 - 1156

第28章
Database System Concepts, 6E

Oracle

Hakan Jakobsson

当1977年Larry Ellison、Bob Miner和Ed Oates成立Oracle软件开发实验室时，还没有商品化的关系数据库产品存在。这家公司，后来更名为Oracle，开始构建作为商用产品的关系数据库管理系统，并成为RDBMS市场的先驱，至此一直在关系数据库市场上保持领导地位。这些年以来，它的产品和提供的服务已经超越了关系数据库服务器的范畴，还包括中间件和应用软件。

除了公司内部开发的产品以外，Oracle的产品还包括被它兼并的公司原本开发的软件。Oracle兼并的公司范围从小公司到大型公开上市交易的公司，包括Peoplesoft、Siebel、Hyperion和BEA。这些兼并的结果，使得Oracle拥有了非常广阔的企业软件产品的系列。

本章重点描述Oracle主要的关系数据库服务器和密切相关的产品。产品的新版本还在不断开发，因此产品的所有说明也会跟着改变。这里表述的特征集是基于Oracle11g的第一个发行版本。它是Oracle当前旗舰数据库产品。

28.1 数据库设计和查询工具

Oracle提供各种工具来进行数据库设计、查询、报表生成和数据分析，包括联机分析处理。这些工具，包括各种其他应用开发工具，是被称作Oracle Fusion Middleware(Oracle融合中间件)的软件产品系列的一部分。产品既包括使用Oracle的PL/SQL编程语言的传统工具，也包括基于Java/J2EE技术的更新的工具。软件支持开放的标准，如SOAP、XML、BPEL和UML。

28.1.1 数据库和应用设计工具

Oracle应用开发框架(Oracle Application Development Framework，ADF)是一个端到端的基于J2EE的
1157 开发框架，基于Model-View-Control(模型-视图-控制)设计模式的。在这个框架中，一个应用程序由多层组成。模型和业务服务层处理与数据源的交互，并包含业务逻辑。视图层处理用户接口，控制层处理应用流以及与其他层之间的交互。

Oracle ADF主要的开发工具是Oracle JDeveloper。它提供了一个支持Java、XML、PHP、HTML、JavaScript、BPEL、SQL和PL/SQL开发的集成开发环境，拥有对UML建模的内置支持。

Oracle Designer是一个数据库设计工具，它将业务逻辑和数据流转换成由应用逻辑的模式定义和过程脚本。它支持诸如E-R图、信息工程、对象分析和设计那样的建模技术。

Oracle还有一个用于数据仓库的应用开发工具，即Oracle Warehouse Builder。Warehouse Builder是对数据仓库的所有方面进行设计和部署的工具，包括模式设计、数据映射和转换、数据加载处理以及元数据管理。Oracle Warehouse Builder既支持3NF(第三范式)和星星模式，又能从Oracle Designer导入设计。这个工具再加上数据库特性(如外部表和表函数)，通常能消除对第三方抽取、转换和加载工具(ETL)的需求。

28.1.2 查询工具

Oracle提供用于即席查询、报表生成、数据分析(包括联机分析处理)的工具。

Oracle商务智能套件(Oracle Business Intelligence Suite，OBI)是一套综合工具，这些工具共享共同的面向服务体系架构。组件包括一个商务智能服务器和用于即席查询、仪表盘(dashboard)生成、报表和警报(alerting)的工具。这些软件共享用于数据访问和元数据管理的基础设施和服务，有一个共同的

安全模型和管理工具。

Oracle BI Answers 组件用于即席查询，是一个交互式工具，它给用户提供数据的逻辑视图，隐藏物理实现的细节。用户可用的对象以图形化的方式显示，用户可以通过点和点击(point-and-click)接口来构造查询。这种逻辑查询被发送到 Oracle BI 服务器组件，该组件接着生成物理查询。查询支持多个物理数据源，查询可以合并存储在关系数据库、OLAP 源和 Excel 表单中的数据。结果可以显示为图表、报表、转轴(pivot)表或者可钻取的仪表板，并可以保存和进行后续修改。

28.2 SQL 的变化和扩展

除了功能一致性(features and conformance)视图以外，Oracle 个部或部分支持 SQL：2003 的所有核心特性。此外，Oracle 支持大量的其他语言结构，其中有些遵从 SQL Foundation：2003 的可选特性，而 [1158]
其余的在语法或功能上是 Oracle 特有的。

28.2.1 对象－关系特性

Oracle 广泛支持对象－关系结构，包括：

- **对象类型**(object type)。类型层次中支持单继承模型。
- **集合类型**(collection type)。Oracle 支持变长数组 **varray** 和嵌套表。
- **对象表**(object table)。它们用于存储对象，同时提供对象属性的关系视图。
- **表函数**(table function)。这些函数可以生成行集合作为输出，可用于查询的 **from** 子句中。Oracle 中的表函数可以嵌套。如果用一个表函数来表示某种数据变换，那么嵌套的多层函数允许在一条语句中表达多重变换。
- **对象视图**(object view)。它们为存储在常规关系表中的数据提供虚拟的对象表视图。它们使得数据可以以面向对象的方式访问或显示，即使这些数据实际上以传统关系格式存储。
- **方法**(method)。它们可以用 PL/SQL、Java 或 C 来编写。
- **用户自定义聚集函数**(user-defined aggregate function)。它们可以像 **sum** 和 **count** 那样的内置函数一样用于 SQL 语句中。

28.2.2 Oracle XML DB

Oracle XML DB 为 XML 数据提供了数据库内的存储，并支持包括 XML Schema 和 XQuery 在内的丰富的 XML 功能集。它建立在被当作原生的 Oracle 数据类型的 **XMLType** 抽象数据类型之上。XML DB 提供了关于这种数据类型的数据在数据库中如何存储的若干选项，包括：

- 结构化为对象关系格式存储。这种格式通常有效利用了空间，允许使用各种标准的关系特征，如 B 树索引，但在 XML 文档和存储格式之间的映射上带来了一些开销。它主要适合于高度结构化的 XML 数据，并且映射中包含可控数量的关系表和连接。
- 作为文本中的非结构化存储。这种表示不需要任何映射，当插入或检索一份完整的 XML 文档时提供了高吞吐量，可是它通常不能非常有效地利用空间，并且当操作 XML 文档的部分时提供欠智能化的处理。 [1159]
- 二进制 XML 存储。这种表示方式是一种后解析(post-parse)的、XML 模式已知的二进制格式。它比非结构化存储的空间利用更高效，可以处理针对 XML 文档部分的操作。在处理高度非结构化数据时也比结构化格式更好，但不是总能有效利用空间。对于 XQuery 语句的处理，这种格式可能比使用结构化格式低效。

二进制和非结构化的表示形式都可以用一种称为 **XMLIndex** 的特殊类型的索引来索引。这种类型的索引允许文档片段基于它们相应的 XPath 表达式来索引。

在数据库内部存储 XML 数据意味着可以得到很多方面的 Oracle 功能上的优势，如备份、恢复、安全和查询处理。它允许把关系数据的访问作为 XML 处理的一部分，也允许把 XML 数据的访问作为 SQL 处理的一部分。一些 XML DB 的非常高级的特点包括：

- 支持 XQuery 语言(W3C XQuery 1.0 推荐的)。

- XSLT 处理使得可以在数据库内部完成 XSL 转换。
- XPath 重写优化可以加速以对象-关系形式存储的数据上的查询。通过把 XQuery 中使用的表达式翻译为直接作用在对象-关系列上的条件，这些列上的常规索引就可用于加快查询处理。

28.2.3 过程化语言

Oracle 有两种主要的过程化语言，分别是 PL/SQL 和 Java。PL/SQL 是 Oracle 存储过程最初使用的语言，其语法类似于 Ada 语言。Java 是通过数据库引擎内部的 Java 虚拟机来支持的。Oracle 提供了一个程序包来控制相关的过程、函数和变量封装为独立的单元。Oracle 支持 SQLJ（Java 内嵌的 SQL）和 JDBC，并提供一个工具来生成对应于用户自定义数据库类型的 Java 类定义。

28.2.4 维度

对于关系星型模式和多维数据库，维度建模是一种常用的设计技术。为了支持基于这种技术设计的数据库上的查询处理，Oracle 支持创建**维度**（dimension）作为元数据对象。这些元数据对象用于存放关于一个维度的各种类型的属性信息，但更重要的可能是关于关系的层次信息。参见 28.3.10 节中的
[1160] 例子。

28.2.5 联机分析处理

Oracle 以若干种不同的方式为分析型数据库处理提供支持。除了支持一个丰富的分析型 SQL 构造集合（立方体、上卷、集合分组、窗口函数等），Oracle 还支持关系数据库服务器内部原生的多维存储。多维数据结构允许基于数组的数据访问，并且在正确情况下，这类访问比传统关系访问方式有效得多。使用这种数据结构作为关系数据库的一个完整部分提供了以关系的或多维的格式存储数据的一种选择，同时还利用了 Oracle 在诸如备份与恢复、安全和管理工具方面的优势。

Oracle 为多维数据提供了称为**分析型工作空间**（analytic workspace）的存储容器。一个分析型工作空间同时包含了 OLAP 立方体的维数据和度量（或事实），并存储在一个 Oracle 表中。从传统关系的观点来看，存放在一个表中的立方体可以是一个不透明的对象，当中的数据通常不能直接根据表的行和列这些术语来解释。不过，数据库内部的 Oracle 的 OLAP 服务器知道如何解释和访问数据，使得可以用 SQL 来访问数据，就像数据是以常规的表格式来存储的一样。因此，可以用多维格式或传统关系格式来存放数据，这取决于哪个最适合，还可以在单条 SQL 查询中连接以两种表示形式存储的数据。物化视图可以使用任何一种表示形式。

除了在 Oracle 关系数据库中支持 OLAP，Oracle 的产品套件包括 Essbase。Essbase 是一个广泛使用的多维数据库，随着对 Hyperion 的收并成为 Oracle 的一部分。

28.2.6 触发器

Oracle 提供若干类型的触发器和激活这些触发器的时间和方式的几种选择。（见 5.3 节关于 SQL 中触发器的介绍。）触发器可以用 PL/SQL 或 Java 语言，或者 C 程序（callout）来编写。

对于在插入、更新和删除这样的 DML 语句上执行的触发器，Oracle 支持**行触发器**（row trigger）和**语句触发器**（statement trigger）。行触发器为每个受 DML 操作影响（譬如更新或删除）的行执行一次。一个语句触发器在每条语句中只执行一次。在每种情况下，根据触发器是在 DML 操作执行之前或之后激活，可以将触发器定义为前触发器或后触发器。

Oracle 允许为不能执行 DML 操作的视图创建**替代**（instead of）触发器。依赖于视图的定义，Oracle 可能无法将一条针对视图的 DML 语句明确地翻译成对底层基表的修改。因此，视图上的 DML 操作受到许多限制。用户可以在视图上创建替代触发器来人工指定在基表上该执行何种操作以响应视图上的
[1161] DML 操作。Oracle 执行触发器而不是 DML 操作，因此提供了一种绕过视图上对 DML 操作限制的机制。

Oracle 还有一些在各种其他事件上执行的触发器，这些事件包括启动或关闭数据库、服务器错误信息、用户登录或退出，以及如 **create**、**alter** 和 **drop** 语句那样的 DDL 语句。

28.3 存储和索引

按照 Oracle 的说法，一个**数据库**（database）是由存储在文件中的信息组成的，并且通过一个**实例**

(instance)来访问；它是一个共享存储区和一组与文件中的数据交互的进程。控制文件(control file)是一个小文件，包含一些启动或操作实例所需的非常高层的元数据。下一节将讲述上述数据和元数据的存储结构。

28.3.1 表空间

数据库由一个或多个称作表空间(table space)的逻辑存储单元组成。而每个表空间又由一个或多个称作数据文件(data file)的物理结构组成。数据文件可能是文件系统的一部分，或是原始的设备。

一个 Oracle 数据库通常会包括以下表空间：

- 系统(system)表空间和辅助的 **sysaux** 表空间总是被创建，它们包含数据字典表、触发器和存储过程的存储器。
- 创建的存储用户数据的表空间。虽然用户数据可以存储在系统表空间，但将用户数据与系统数据分离一般是值得的。通常，决定要创建另外哪些表空间是基于性能、可用性、可维护性以及易于管理来考虑的。譬如，拥有多个表空间对于后部备份和恢复操作比较有用。
- **撤销**(undo)表空间，它仅仅用于存放事务管理和恢复的撤销信息。
- **临时**(temporary)表空间。许多数据库操作需要对数据进行排序，如果排序不能在内存中完成，排序例程可能需要将数据暂时保存在硬盘上。为排序和散列操作分配临时表空间，能够使得涉及将数据溢出到硬盘的空间管理操作更加高效。

表空间也可以作为在数据库之间转移数据的一种方式。譬如，通常每隔一段时间就要把数据从事务系统转移到数据仓库中去。Oracle 通过简单地复制数据文件并导出和导入少量的数据字典元数据，就允许将一个表空间中的所有数据从一个系统转移到另一个系统。这些操作比从一个数据库卸载数据，然后用一个加载器将数据插入到另一个数据库中要快得多。Oracle 的这个特性称为**可移动表空间** 1162
(transportable table space)。

28.3.2 段

表空间中的空间被划分成一个个单元，称为**段**(segment)。每个段包含一种特定数据结构的数据。一共有四种类型的段：

- **数据段**(data segment)。表空间中的每个表都有自己的数据段，除非表被划分，表数据就存储在这里；若表被划分，每个分区都有一个数据段。(Oracle 中的划分在 28.3.9 节中描述。)
- **索引段**(index segment)。除了被划分的索引，每个表空间中的索引都有自己的索引段，被划分的索引在每个分区有一个索引段。
- **临时段**(temporary segment)。这些段用于当排序操作需要将数据写到硬盘上时，或者将数据插入到临时表中时。
- **撤销段**(undo segment)。这些段包含了撤销信息，使未提交事务可以回滚。在特殊的撤销表空间中，这些段是自动分配的。它们也在 Oracle 的并发控制模型和数据库恢复中起重要作用。这将在 28.5.1 节和 28.5.2 节中讲到。为了实现 Oracle 的撤销管理，使用"回滚段"这个术语。

在段这一层以下，空间以盘区(extent)的粒度级别进行分配。每个盘区由一组连续的数据库块(block)组成。数据库块是 Oracle 进行磁盘 I/O 的最低的粒度级别。数据库块的大小不必和操作系统中的块相同，但必须是它的倍数。

Oracle 提供存储参数，允许对如何分配和管理空间进行详细控制。这样的参数如下：

- 为给插入表中的行提供空间而即将要分配的新盘区的大小。
- 空间利用的百分比，用该比例来定一个数据库块已满，并且不允许更多的行插入到这个块中。(在块中留一些自由空间，可以允许已有的行通过更新而增大，却不会溢出这个块的空间。)

28.3.3 表

Oracle 中的标准表是以堆组织的；也就是说，表中行的存储位置不是由该行所包含的值决定的，而是在行被插入时确定的。但是，如果这个表被划分，行的内容影响到该行被存储到哪个分区中。这当中有几个特点和变化，如同 28.3.3.2 节介绍的，堆表可以选择为被压缩。 1163

Oracle 支持嵌套表：也就是说，表中列的数据类型可以是另一个表。嵌套表不是有序地存储在父表中，而是存储在单独的表中。

Oracle 支持临时表。临时表中数据的持续时间要么是插入这个数据的事务的时间，要么是用户会话的时间。数据对于会话而言是私有的，并且在其持续时间结束时自动删除。

簇(cluster)是表数据的另一种文件组织形式，在前面的 10.6.2 节中描述过，当时称为多表聚簇。这里用到"簇"这个术语，不要与 cluster 这个词的其他意思混淆了，如那些与硬件体系结构相关的意思。在一个簇的文件组织中，基于某些共同的列，来自不同表中的行被一起存储在相同的块中。譬如，一个部门表和一个雇员表可以聚簇在一起，这样部门表中的每行与所有在该部门工作的雇员对应的雇员行存储在一起。主码/外码的值用于确定存储位置。

簇的组织暗示了行属于一个特定的位置；譬如，一个新的雇员行必须与相同部门的其他的行插入到一起。所以，簇列上的索引是强制性的。另一种可选的组织方式是**散列簇**(hash cluster)。这里，Oracle 通过将一个散列函数作用到簇列的值上，以计算出一个行的位置。散列函数将行映射到散列簇中一个特定的块。由于根据簇列的值来访问一个行时不需要索引遍历，这种组织可以节省大量的磁盘 I/O。

28.3.3.1 按索引组织的表

在一个**按索引组织的表**(Index-Organized Table，IOT)中，记录存储在 Oracle 的 B 树索引中，而不是在堆中。之前在 11.4.1 节介绍了这种文件组织，当时称为 B* 树文件组织。一个 IOT 需要一个可识别的唯一的关键字用作索引关键字。虽然常规索引中的条目包含了索引行的关键字值和行标识，但 IOT 用行中剩余列的列值来代替行标识。与将数据存储在常规堆表中并在关键字列上创建索引相比，使用 IOT 可以同时提高性能和空间利用率。给定主码值，考虑查找一个行的所有列值。对于堆表，需要在探测索引之后再通过行标识访问表。对于 IOT，只需要探测索引。

按索引组织的表中非关键字列上的辅助索引与常规堆表上的索引不同。在堆表中，每行有一个固定不变的行标识。但是，B 树在插入或删除条目后会因增长或缩小而重新组织，并且不能保证 IOT 中的行会留在固定的地方。所以，IOT 上的辅助索引不包含通常的行标识，而是**逻辑行标识**(logical row-id)。逻辑行标识由两部分组成：对应于在创建索引时或在上次重建索引时行的存储位置的物理行标
1164 识，以及唯一性关键字的值。物理行标识作为"猜测"来引用，因为它在行移动后可能是错的。如果这样的话，逻辑行标识的另一部分，行的关键字的值就被用于访问该行；但是，此访问比猜测正确时要慢，因为它涉及对 IOT 的 B 树从根到叶结点的所有路径的遍历，潜在地导致了一些磁盘 I/O。然而，如果一个表是易变的并且大部分猜测可能是错的，那么只用关键值来创建辅助索引要更好一些(如 11.4.1 节所述)，因为使用不正确的猜测会导致磁盘 I/O 的浪费。

28.3.3.2 压缩

Oracle 的压缩特性允许数据以压缩格式存储，压缩极大地降低了存储数据所需的空间和检索数据所需的 I/O 类作数。Oracle 的压缩方法是一种无损的，基于字典的算法。它单独地压缩每个块。所有用来解压一个块的信息就包含在那个块自身中。算法把在块中重复出现的值替换为指向块中符号表(或字典)中该值所对应项的指针。项可以基于单独的列或者列的组合中重复的值。

Oracle 最初的表压缩是当数据被批量加载到表中时生成压缩的块格式，主要是用于数据仓库环境。更新的 OLTP 压缩特性还支持与常规 DML 类作相关的压缩。在后一种情况下，Oracle 只在达到了特定阈值量的数据被写入到块中之后才压缩块。因此，只有导致超过阈值的事务才会产生压缩数据块的任何开销。

28.3.3.3 数据安全

除了通常的诸如密码、用户权限和用户角色这样的访问控制特性，Oracle 还支持一些防止数据被非授权访问的特性，包括：

- **加密**(encryption)。Oracle 可以自动地以加密格式存储表数据，并能使用 AES 和 3DES 算法透明地加密和解密数据。可以对整个数据库加密，也可以只对表中的单个的列加密。此特性的主要动机是在通常的保护环境之外保护敏感数据，如当备份媒质被发送到远地。

- **数据库保险库**(database vault)。此特性的目的是为用户访问数据库提供职责分离。数据库管理员是有高度特权的用户，通常几乎可以在数据库中做任何事情。可是，让这样的人访问敏感的公司财务数据或关于其他员工的个人信息可能是不合适的或违法的。数据库保险库包括多种机制，可用于限制或监视高特权的数据库用户对敏感数据的访问。 1365
- **虚拟私人数据库**(virtual private database)。在前面9.7.5节描述过此特性，它允许附加的谓词被自动添加到访问一个给定表或视图的查询的**where**子句上。典型地，使用此特性使得附加谓词过滤掉所有用户无权看到的行。例如，两个用户可以提交相同的查询来找出在整个员工表中所有员工的信息。可是，如果存在一种策略限制了每个用户只能看到员工号与用户标识匹配的信息，自动添加的谓词确保每个查询只返回提交查询的用户所对应的员工信息。因此，给每个用户留下的印象是访问一个只包含了物理数据库中一个数据子集的虚拟数据库。

28.3.4 索引

Oracle支持几种不同类型的索引。最常使用的类型是建立在一列或多列上的B树索引。注意在Oracle(也在其他几种数据库系统中)的术语中，B树索引就是第11章中所说的B$^+$树索引。索引项有以下格式：对于在列col_1、col_2和col_3上的索引，表中每个至少在一个这样的列上取非空值的行可以形成这样的索引项：

$$< col_1 > < col_2 > < col_3 > < row\text{-}id >$$

这里$< col_i >$表示第i列上的取值，$< row\text{-}id >$是行的行标识。Oracle可以有选择地压缩项的前缀以节省空间。譬如，如果有许多重复的$< col_1 > < col_2 >$值的组合，则每个不同的$< col_1 > < col_2 >$前缀表示就可以在具有该值组合的项之间共享，而不是为每个这样的项显式存储。前缀压缩可以带来实质性的空间节省。

28.3.5 位图索引

位图索引(在11.9节中讲到)使用位图表示索引项。当被索引的列有适中数量的不同值的时候，这种表示可以实质性地节省空间(同样也节省了磁盘I/O)。Oracle中的位图索引使用与常规索引相同类型的B树结构来存储项。然而，列上的常规索引的项形如$< col_i > < row\text{-}id >$，位图索引的项形如：

$$< col_i > < start\ row\text{-}id > < end\ row\text{-}id > < compressed\ bitmap >$$

位图在概念上代表表中起始行标识至终止行标识之间所有可能的行占用的空间。一个块中这种可能的行的数量取决于块中可以容纳多少行，它是关于表中列的数量及其数据类型的一个函数。位图中每个位(bit)表示块中这样一个可能的行。如果该行在列上的取值等于索引项的值，这一位就设置为1。 1366
如果该行取其他值，或者这一行根本不在表中存在，这一位就设置为0。(其行在表中实际上并不存在是可能的，因为一个表的块中的行数可能比由计算得到的最大可能的行数要少。)如果差异很大，可能导致位图中出现长串连续的0，但是压缩算法能够处理这种0串，因而负面影响是有限的。

压缩算法是一种叫做字节对齐位图压缩(Byte-Aligned Bitmap Compression，BBC)的压缩技术的变体。本质上，位图中两个连续的1之间的距离足够小的部分存储为逐字位图(verbatim bitmap)。如果两个1之间的距离足够大，也就是说，它们之间有大量邻接的0，则存储的是0的长度，也即0的个数。

如果在查询的**where**子句中，在索引列上有多个条件时，位图索引允许把同一个表上的多个索引合并在同一个访问路径中。根据**where**子句中的条件，使用布尔运算来合并从不同索引中检索到的位图。所有布尔运算都是直接在位图的压缩形式上进行的，无须解压缩，并且结果(压缩的)位图表示匹配所有要求条件的那些行。

使用布尔运算来合并多个索引的能力并不只限于位图索引。Oracle可以把行标识转化为压缩位图形式，所以它能在位图运算的布尔树中的任何地方使用常规的B树索引，只要简单地在执行计划中把行标识-位图操作符放在索引访问之上即可。

根据经验，如果不同码值数目少于表中行数的一半，位图索引应该比常规B树索引空间利用率更高。例如，在一个具有100万行的表中，如果在少于50万个不同取值的列上创建的是位图索引，可能更节省空间。对于取很少的、不同值的列，例如，对于表示国家、州、性别、婚姻状况以及各种状态

标志属性的列，位图索引可能只需要常规 B 树索引所用空间的一小部分。在扫描索引的时候，任何这样的空间优势都能以更少的磁盘 I/O 带来相应的性能优势。

28.3.6 基于函数的索引

除了在表的一个或多个列上创建索引外，Oracle 还允许在包含一个或多个列的表达式上创建索引，比如 $col_1 + col_2 * 5$。例如，在表达式 *upper(name)* 上创建一个索引，其中 *upper* 是一个函数，它返回字符串的大写形式，而 *name* 是一个列，这就可以对 *name* 列执行大小写不敏感的搜索。为了高效地找到含有名称"van Gogh"的所有行，条件：

1167

```
upper(name) = 'VAN GOGH'
```

将在查询的 **where** 子句中使用。Oracle 随后将该条件与索引定义相匹配，并且不管存储在数据库中时 *name* 的大小写如何，最后该索引都能用于检索出匹配"van Gogh"的所有行。基于函数的索引可以创建为位图索引或者 B 树索引。

28.3.7 连接索引

连接索引是这样一种索引：索引中行标识所指向的关键字列并不在表中。Oracle 支持位图连接索引，主要用于星型模式(见 20.2.2 节)。例如，如果产品维表中有一个产品名字的列，事实表中在该关键字列上的位图连接索引可用于检索出对应于特定名称产品的事实表中的行。虽然产品名称并没有存储在事实表中，事实表和维表两个表中的行是如何对应的，取决于创建索引时指定的连接条件，并成为索引元数据的一部分。当处理查询时，优化器从查询的 **where** 子句中查找相同的连接条件，以决定是否可以应用连接索引。

Oracle 能够借助布尔位图运算的操作符，把事实表上的位图连接索引与同一个表上的其他索引(无论是否为连接索引)合并。

28.3.8 域索引

Oracle 允许非 Oracle 原生的索引结构对表进行索引。Oracle 服务器的这种扩展特性允许软件厂商为特定应用领域，如文本、空间数据和图像，开发所谓插卡式(cartridge)的功能，以提供标准 Oracle 索引类型未提供的索引功能。在实现索引的创建、维护和搜索逻辑时，索引设计者必须确保索引与 Oracle 服务器进行交互时遵循特定的协议。

域索引必须连同它所支持的操作符一起在数据字典中注册。Oracle 优化器认为域索引是表可能的访问路径之一。Oracle 允许为操作符注册代价函数，据此优化器可以比较使用域索引和使用其他访问路径的代价。

例如，高级文本搜索的域索引可能支持 *contains* 操作符。一旦注册这个操作符，该域索引就会当作是形如下述查询的一条访问路径：

```
select *
from employees
where contains(resume, 'UNIX');
```

1168 其中 *resume* 是 *employee* 表中的一个文本列。域索引既可以存储在外部数据文件中，也可以存储在 Oracle 的按索引组织的表内部。

借助于在行标识和位图表示之间的转换以及使用布尔位图运算，域索引可以与在同一访问路径下的其他索引(位图索引或者 B 树索引)合并。

28.3.9 划分

Oracle 支持多种对表和索引的水平划分方式，并且这种特性在 Oracle 支持特大数据库的能力中发挥着重要作用。对表或索引进行划分的能力有许多方面的优点。

- 备份和恢复更容易也更快速，因为可以对单独的分区运行而不必对整个表运行。
- 数据仓库环境下的加载操作不那么生硬：数据可以添加到新创建的分区，然后把分区添加到表中。这是一种瞬时性操作。同样地，在维护历史数据滚动窗口的数据仓库中，从表中丢弃存放废弃数据的分区非常容易。

- 查询性能获得实质性受益，因为处理查询时优化器可以识别只需要访问表的分区的一个子集（分区裁剪，partition pruning）。同样，在连接时优化器可以识别，不必把一个表的所有行和另一个表的所有行进行匹配，而只需要在匹配的分区对之间进行连接（按分区连接）。

被划分的表上的索引可以是**全局索引**（global index），或者是**局部索引**（local index）。全局索引中的项可以指向任何分区中的行。局部索引的表对每个划分有一个物理索引，它只包含该分区中的项。除非分区裁剪限制了查询只能在单个分区上，通过局部索引访问的表需要对多个独立的物理索引进行探查。可是，在数据仓库环境下局部索引具有这样的优势：新的数据可以加载到一个新的分区中并被索引，无需维护任何已有索引。（在数据加载中，加载后创建索引远比维护已有索引更高效。）类似地，删除一个旧的分区及其局部索引的物理部分也不需要维护任何索引。

被划分表中的每一行都与某一特定分区相关联。该关联基于划分列或被划分的表定义的部分列。有几种方法将列值映射到分区，这就形成了几种类型的划分：范围划分、散列划分、列表划分以及复合划分。每种划分都有不同的特征。 1169

28.3.9.1 范围划分

在范围划分中，划分标准是值的范围。此划分类型特别适合于日期列，其中在同一个日期范围内的所有行，比如一天或者一个月，都属于同一个分区。在数据仓库中，数据以固定间隔从事务型系统中导入，使用范围分区可有效实现历史数据的滚动窗口。每次数据加载都有其自己的新分区，这使得加载过程更快更有效。系统实际上把数据装载到和被划分表具有相同列定义的单独表中。然后它可以检查数据的一致性，清理数据，并为数据建立索引。之后，系统只要简单地改动数据字典中的元数据，就可以把这个单独的表作为被划分表的一个新分区。此操作几乎是瞬时的。

直到元数据改变之前，加载过程并不以任何方式影响被划分表中已存在的数据。加载中不需要对已有索引进行任何维护。旧数据可以通过简单丢弃其分区而从表中删除；此操作并不影响其他分区。

此外，数据仓库环境中的查询通常包含将查询限定在特定时间段内的条件，比如一个季度或者一个月。如果使用数据范围划分，查询优化器可以把数据访问限定到和查询相关的那些分区中，从而避免全表扫描。

划分既可以通过显式地设置一个结束点来创建，也可以基于一个固定范围来定义，如一天或一个月。后一种情况称为**区间划分**（interval partitioning），当试图插入一个行，且它的值没有落在之前存在的区间内时，包含该值的分区会自动创建。

28.3.9.2 散列划分

在散列划分中，散列函数根据划分列中的值把行映射到分区中。这种划分类型主要用在当把行均匀分布到各分区中很重要时，或者当按分区的连接对查询性能而言很重要时。

28.3.9.3 列表划分

在列表划分中，与特定分区相关的值在一个列表中声明。如果在划分列上的数据具有相对较小的离散值的集合，则这种划分类型是有用的。例如，如果每个分区列表具有属于相同区域内的州，一个带有州的列的表无疑可以按照地理区域进行划分。

28.3.9.4 复合划分

在复合划分中，被范围、区间或列表划分的表可以进行范围、列表或者散列子划分。例如，一个表可以在日期列上进行范围划分，并在频繁用于连接的列上进行散列子划分。这样的子划分允许在表连接时采用按分区的连接。 1170

28.3.9.5 参照划分

在参照划分中，划分关键字是基于为一个表的外键约束来定的。这种表之间的依赖允许自动、级联地进行维护操作。

28.3.10 物化视图

物化视图特性（见 4.2.3 节）允许 SQL 查询结果存储在一个表中，并用于以后的查询处理。此外，Oracle 维护这种物化结果，在查询涉及的表更新时对物化结果进行更新。在数据仓库中使用物化视图来加快查询处理，但这种技术还用于分布式或移动环境中的复制。

在数据仓库中，物化视图的一个通常用途是汇总数据。例如，一种常用的查询类型是询问"过去两年中每个季度的销售总额"。预先计算这种查询的结果或者部分结果，与从头开始从详细的销售记录中层层聚集得到结果相比，可以极大地加快查询处理。

Oracle 在处理查询时，可以利用任何有用的物化视图来支持自动查询重写。这种重写包括改变查询，以利用物化视图来代替查询中原来的表。此外，这种重写可能会添加额外的连接或者聚集处理，然而为了得到正确结果而可能需要它们。例如，如果查询需要季度的销售信息，查询重写可以利用物化的月销售视图，增加额外的聚集完成从月到季度的上卷。Oracle 具有一类叫做维(dimension)的元数据对象，它允许定义表中的层次关系。例如，对于星型模式中的时间维表，Oracle 能够定义一个维元数据对象来指定如何从日上卷到月，从月上卷到季度，从季度上卷到年，诸如此类。同样地，可以指定与地理相关的层次属性——例如，销售区域如何上卷到销售地域。查询重写逻辑之所以关注这些关系，是因为它们允许在更广泛的查询类别中使用物化视图。

物化视图的容器对象是表。这意味着可以对物化视图进行索引、划分或者其他控制，以提高查询性能。

当定义物化视图的查询所涉及表中的数据发生变化时，必须刷新物化视图以反映这些变化。Oracle 支持物化视图的完全刷新，也支持快速的增量刷新。在完全刷新中，Oracle 从头开始重新计算物化视图，如果底层表发生重大变化这可能是最佳选择，如同批量加载导致的变化。在快速刷新中，Oracle 使用底层表中发生变化的记录来更新视图。对视图的刷新可以将提交(on commit)作为改变底层表的事
[1171] 务的一部分来执行，也可以按需要(on demand)在之后的某个时间点执行。如果表中发生改变的行数量较少，快速刷新可能是更好的。关于物化视图能否被增量刷新，在查询类别上是有一些限制的(另有其他关于物化视图何时才能创建的限制)。

物化视图与索引的相似之处在于能够提高查询性能，但它占用空间，其创建和维护要消耗资源。为了帮助处理这种权衡，Oracle 提供了一个向导，在给定特定的查询工作负载作为输入的条件下，它可以帮助用户创建一个最划算的物化视图。

28.4 查询处理和优化

Oracle 在其查询处理引擎中支持大量的处理技术。在这里对一些更重要的技术进行简要描述。

28.4.1 执行方法

数据可以通过多种访问方法来访问：

- **全表扫描**(full table scan)。查询处理器扫描全表：它根据区块映射获取关于构成表的块的信息，并扫描这些块。
- **索引扫描**(index scan)。处理器根据查询中的条件创建起始键和/或终止键，并使用它扫描索引中的相关部分。如果有需要检索出的列，而这些列不是索引的一部分，则在索引扫描之后再通过索引行标识进行表访问。如果没有起始键或停止键可用，扫描就成为全索引扫描。
- **索引快速全扫描**(index fast full scan)。处理器扫描区块的方式与全表扫描中扫描表的区块相同。如果索引包含了表中所需要的所有表列，并且没有好的起始/终止键可以显著地减少常规索引扫描中所需扫描的索引部分，这种方法可能是访问数据最快的途径。这是因为快速全扫描能充分利用多个块的磁盘 I/O。然而，不像常规全扫描那样顺序遍历索引叶块，快速全扫描并不保证其输出保持索引排序的顺序。
- **索引连接**(index join)。如果查询只需要宽表(wide table)的一个小的列的子集，但没有单个索引包含所有这些列，处理器能够使用索引连接来产生相关信息，而不用访问表。方法是把几个共
[1172] 同包含所需列的索引连接起来。它执行连接的方式是在不同索引的行标识上进行散列连接。
- **簇和散列簇访问**(cluster and hash cluster access)。处理器使用簇码来访问数据。

Oracle 有几种方法把来自多个索引的信息合并在单个访问路径里。这种能力允许多个 **where** 子句条件一起使用来尽可能有效地计算出结果集。此功能包括在位图表示的行标识上执行布尔运算 **and**、**or** 和 **minus** 的能力。还有操作符可以把行标识列表映射到位图，或者反之。它允许常规的 B 树索引和

位图索引在同一个访问路径中一起使用。此外，对于许多对表的选择中用到 **count**(*) 的查询，通过应用 **where** 子句条件产生位图，查询结果可以直接通过对位图中所置的位进行计数来计算出来，不用访问表。

Oracle 在执行引擎中支持几种类型的连接：内连接、外连接、半连接（semijoin）和反连接（antijoin）。（Oracle 中的反连接返回左边输入关系中的行，这些行与右边输入关系的任何行都不匹配；在其他文献中此操作也叫做反半连接（anti-semijoin）。）它用下述三种方法之一来计算每种类型的连接：散列连接、非序归并连接或者嵌套循环连接。

28.4.2 优化

第 13 章讨论了查询优化的一般问题。这里，我们讨论 Oracle 系统中的优化。

28.4.2.1 查询转换

Oracle 在几个阶段上进行查询优化。一个这样的阶段是执行各种查询转换和重写，从根本上改变查询结构。另一个阶段是执行访问路径选择以确定访问路径、连接方法和连接次序。因为某些转换未必总是有益的，Oracle 使用基于代价的查询转换，其中转换和路径选择是交叉进行的。对于每个所尝试的转换，通过执行路径选择以生成代价估计，并基于结果执行计划的代价决定接受还是拒绝此转换。

Oracle 支持的转换和重写的一些主要类型如下：

- **视图合并**（view merging）。查询中的视图引用由视图定义替代。这种转换并非对所有视图都适用。
- **复杂视图合并**（complex view merging）。Oracle 为特定类型的视图提供此特性，它们不能进行常规的视图合并，因为它们在视图定义中有 **group by** 或 **select distinct**。如果这种视图与其他表连接，Oracle 能够替换用于 **group by** 或 **distinct** 的连接和排序或散列操作。 1173
- **子查询整平**（subquery flattening）。Oracle 有各种转换可以把不同种类的子查询转变为连接、半连接、或者反连接。这样的转换也称为去除相关，在 13.4.4 节中进行过简要描述。
- **物化视图重写**（materialized view rewrite）。Oracle 具有自动重写查询的能力以利用物化视图。如果查询中的某部分可以与一个已存在的物化视图匹配，Oracle 能够用对存储该物化视图的表的引用来替换这部分。如果需要，Oracle 添加连接条件或 **group by** 操作来保持查询的语义。如果有多个物化视图可以利用，Oracle 选取在减少所需处理的数据量方面最有利的一个。此外，Oracle 将重写后的查询和原始查询都提交给整个优化过程，为它们分别生成执行计划以及相关的代价估计，然后 Oracle 根据代价估计决定是执行重写的查询还是原始查询。
- **星型转换**（star transformation）。Oracle 支持一种技术来估算星型模式上的查询，称为星型转换（star transformation）。当查询包含事实表和维表的连接，并且对维表的属性进行选择时，查询按如下方式转换：删除事实表和维表之间的连接条件，并把每个维表的选择条件替换为如下形式的子查询：

```
fact_table.fk_i in
        (select pk from dimension_table_i
         where <conditions on dimension_table_i>)
```

对于每个具有一些限定谓词的维都产生一个这样的子查询。如果维具有雪花模式（见 20.2 节），子查询将包含对构成此维的适用表的一个连接。

Oracle 使用每个子查询返回的值来探查相应事实表列上的索引，得到一个位图作为结果。从不同子查询产生出的位图通过位图与运算进行合并。结果位图可以用于访问匹配的事实表行。所以，只有那些在事实表中，同时匹配约束维上条件的行才会被访问。不管确定为特定的维使用子查询是否划算，还是确定重写的查询是否比原始查询更好，都取决于优化器的代价估计。

28.4.2.2 访问路径选择

Oracle 有一个基于代价的优化器来决定连接顺序、连接方法以及访问路径。优化器所考虑的每个操作都有一个相应的代价函数，优化器试图产生总体代价最小的操作组合。 1174

在估计操作的代价时，优化器依据模式对象(如表、索引)上已计算出来的统计数据。这些统计数据包括关于对象大小、基数、表列的数据分布情况等信息。对于数据分布，Oracle 支持高度平衡(height-balanced)直方图和频率直方图。高度平衡直方图也称为等深直方图，在 13.3.1 节进行过描述。

为了方便收集优化器统计数据，Oracle 能够监视表上的修改活动，跟踪那些已经做了足够多修改的表，它们可能适合于重新计算统计数据。Oracle 还跟踪在查询的 **where** 子句中用到哪些列，把它们作为创建直方图的潜在候选者。用户可以通过一条命令，通知 Oracle 为那些已经标记为做了足够多修改的表刷新统计数据。Oracle 用采样来加速收集新统计信息的过程，并自动选择最小的、足够的样本百分比。它还决定标记列的分布是否有利于建立直方图；如果数据的分布趋于均衡，则 Oracle 用更简单的方法来表示列的统计数据。

在某些情况下，优化器不太可能仅仅依靠简单的列统计数据来精确估计查询的 **where** 子句中的条件的选择率。例如，条件可能是一个包含列的表达式，如 $f(col+3)>5$。另一类存在问题的查询是那些在列上有多个谓词，并且这些列存在某种形式的相关性的查询。估计这些谓词的联合选择率可能比较困难。Oracle 为此允许为表达式和分组的列创建统计数据。另外，Oracle 通过动态采样来处理这些问题。优化器可以随机采样表的一小部分，将所有相关的谓词应用到样本上，从而得出匹配的行的百分比。这个特性也用于处理那些数据生命周期和可见性可能阻碍常规统计数据收集的临时表。

在优化器代价模型中，Oracle 既使用 CPU 代价，也使用磁盘 I/O。为了平衡这两个部分，它保存了关于 CPU 速度和磁盘 I/O 性能的度量，作为优化器统计数据的一部分。Oracle 用于收集优化器统计数据的软件包负责计算这些度量。

对于有比较多连接操作的查询，优化器的搜索空间是一个问题。Oracle 有几种方法处理这个问题。优化器产生一个初始连接顺序，然后按该连接顺序选择最好的连接方法和访问路径。接着改变表的顺序，并为新的连接顺序选择最好的连接方法和访问路径，以此类推，同时保存目前为止最好的计划。如果要考虑的不同连接顺序的数量太大，以至于优化器所花费的时间与它执行当前已找到的最优计划所花的时间比起来可能较显著，Oracle 会中断优化。由于这种中断依赖于当前已找到的最好计划的代
[1125] 价估计，所以尽早找到一个好的计划很重要，这样优化器在更少量的连接顺序之后就可以停止，导致响应时间更短。Oracle 采用几种初始排序启发式法则来增加第一次连接顺序就是好策略的可能性。

对于所考虑的每种连接顺序，优化器可能执行额外的对表的扫描来决定连接方法和访问路径。这种额外扫描的目标是确定访问路径选择的全局副效应。比如一个特定的连接方法和访问路径的组合可以省略执行 **order by** 排序的需要。由于在局部考虑不同连接方法和访问路径的代价时，这种全局副效应可能并不明显，可用定位特定副效应的单独的扫描来发现可能的、整体代价更好的执行计划。

28.4.2.3 分区裁剪

对于划分的表，优化器试着把查询 **where** 子句中的条件和表划分的标准进行匹配，为的是避免访问结果所不需要的分区。例如，如果表以日期范围来划分，并且查询被限定在两个特定日期之间的数据上，优化器判定哪些分区包含了这两个特定日期之间的数据，并确保只访问这样的分区。这样的情形非常普遍，如果只需要分区的一个小的子集，加速比是最显著的。

28.4.2.4 SQL 调优顾问

除了常规的优化流程，Oralce 优化器还能作为 SQL Tuning Advisor(SQL 调优顾问)的一部分而用于调优模式中，为的是生成比平时更为高效的执行计划。这个特性对于反复生成相同 SQL 语句集合的打包应用特别有用，这样为了性能而调优这些语句的努力在将来是有益的。

Oralce 监控数据库活动，并自动将关于负载重的 SQL 语句的信息存放在工作负载存储器中，参见 28.8.2 节。负载重的 SQL 语句是那些用尽绝大部分资源的语句，因为它们执行了太多次或者每次执行代价就很大。这样的语句是调优的合理的候选者，因为它们对系统的影响最大。SQL 调优顾问可用于提供各种建议，从而提高这些语句的性能，这些建议可分为如下不同种类：

- **统计分析**(statistics analysis)。Oracle 检查优化器所需要的统计数据是否缺失或者陈旧，并且提供收集这些数据的建议。
- **SQL 轮廓记录**(SQL profiling)。SQL 语句的轮廓记录是为了帮助优化器下次优化此语句时能做

出更好决策的一组信息。如果优化器不能精确估计基数和选择率，它有时可能会生成低效的执行计划。由于数据的相关性或者使用了特定的构造类型会发生这样的事情。当在调优模式下运行优化器以创建轮廓记录时，优化器会使用动态采样和 SQL 语句的部分评估来验证其假设是否正确。如果它发现在优化过程的某些步骤中优化器的假设是错误的，它将为该步骤生成一个校正因子，使其成为轮廓记录的一部分。调优模式下的优化非常耗时，不过若轮廓记录的使用能显著提高语句性能，就是值得的。轮廓记录在创建之后会持久地存储，并在今后优化该语句的任何时候使用。轮廓记录能用于 SQL 语句调优，而无需改变语句内容，这一点很重要，因为数据库管理员通常不可能修改应用所产生的语句。 1176

- **访问路径分析**(access-path analysis)。基于优化器的分析，Oracle 建议创建额外的索引来加速语句的执行。
- **SQL 结构分析**(SQL structure analysis)。Oracle 建议改变 SQL 语句的结构，以便于更高效的执行。

28.4.2.5 SQL 计划管理

打包的应用程序经常生成大量重复执行的 SQL 语句。如果应用程序表现得足够好，数据库管理员通常不愿意修改数据库行为。如果修改导致性能更好，由于性能已经够好了，能提升的空间是有限的。另一方面，如果修改导致性能下降，如果一个关键查询的响应时间下降到不可接受，它可能破坏了应用程序。

改变行为的一个例子是改变查询的执行计划。这样的改变可能完全合理地反映了数据性质的变化，例如一个表增长得非常大。但是这样的改变也可能是许多其他动作的无意的结果，如收集优化器统计数据例程的改变或升级到具有新的优化器行为的 RDBMS 的新版本。

Oracle 的 SQL 计划管理特性是处理与执行计划变化有关的风险，方法是为工作负荷维护一个可信的执行计划集合，并仅在验证这些改变不会引起任何性能下降之后，才分阶段地进行查询优化器对计划的改变。此特性有三个组件：

1. **SQL 计划基线捕获**(SQL plan baseline capture)。Oracle 可以捕获工作负载的执行计划并为每条 SQL 语句存储历史计划。计划基线是工作负载的计划集合，具有可靠的性能特性并且是未来计划变动可以比较的基准。一条语句可以拥有不止一个基线计划。

2. **SQL 计划基线选择**(SQL plan baseline selection)。在优化器为 SQL 语句生成计划之后，它检查是否存在该语句的基线计划。如果语句在基线中存在但新的计划不同于任何已有计划，则使用优化器认为是最好的基线计划。新生成的计划将添加到该语句的计划历史中，并可能成为一个未来基线的一部分。 1177

3. **SQL 计划基线演化**(SQL plan baseline evolution)。周期性地试着让新生成的执行计划成为在基线中可信计划的一部分是有意义的。Oracle 支持带验证或不带验证地将新的计划添加到基线中。如果选择验证方式，Oracle 将执行新生成的计划，并将其性能与基线比较，以确保它不会导致性能衰退。

28.4.3 并行执行

通过将工作分配到一台多处理器计算机的多个进程中，Oracle 允许并行地执行单条 SQL 语句。这个特性对于计算密集型操作尤其有用，否则这样的计算将花费不可接受的长时间来执行。典型的例子是：需要处理大量数据的决策支持查询、数据仓库中的数据加载、索引的创建以及重建。

为了通过并行得到高的加速比，将语句执行中所涉及的工作分解成能够由不同的并行处理器独立处理的粒度是很重要的。根据操作类型的不同，Oracle 有几种不同的方法来对工作进行分割。

对于访问基本对象(表和索引)的操作，Oracle 能够通过数据的水平切片来分割工作。对于一些操作，比如全表扫描，每个这样的切片可以是一个范围内的块，每个并行的查询进程从范围的第一个块开始扫描表，到最后一个块结束。对于划分表上的某些操作，例如索引范围扫描，切片就是一个分区。基于块范围的并行更灵活，因为这可以基于各种标准来动态决定，而不受制于表的定义。

连接可以通过几种不同的方式来并行化。一种方式是，把连接的一个输入划分到并行进程中，让每个进程将其切片与连接的另一个输入做连接，这就是 18.5.2.2 节的非对称分片 - 复制方法。例如，

如果一个大表和一个小表做散列连接，Oracle 把大表划分到进程中，并将小表的副本广播给每个进程，然后每个进程将其切片与小表进行连接。如果两个表都大，把它们中的一个广播给所有进程的代价将大得难以承受。在这种情况下，Oracle 通过下述方法实现并行化：通过对连接列取值的散列把数据划分到进程中(18.5.2.1 节的基于划分的散列连接方法)。每个表被一组进程并行扫描，输出的每一行送到一组执行连接的一个进程中。这些进程中的哪一个得到该行数据，由连接列值上的散列函数决定。
[1178] 因此，每个连接进程只会得到可能匹配的行，并且任何可能匹配的行不会在不同的进程中结束。

Oracle 按执行排序的列上的取值范围进行并行排序操作(也就是使用 18.5.1 节的范围划分排序)。每个参与排序的进程都得到取值在其范围内的行，并在其范围内将这些行排序。为了最大化并行的优势，需要在并行进程中尽可能平均地划分这些行，这就出现了确定范围边界以产生好的数据分布的问题。Oracle 通过在确定范围边界之前对排序输入中的行动态采样出一个子集来解决这个问题。

28.4.3.1 进程结构

SQL 语句并行执行中的进程包括一个协调进程和许多并行服务器进程。协调进程负责给并行的服务器分配任务并且为发出该语句的用户进程收集和返回数据。并行度是并行的服务器进程的数量，这些服务器进程被指派来执行作为该语句一部分的原始操作。并行度由优化器决定，但是如果系统负载增加，并行度会动态下降。

并行服务器按照生产者/消费者模型操作。当处理某条语句需要一系列操作时，服务器的生产者集合执行第一个操作，并把结果数据送给消费者集合。比如，一个全表扫描后接排序，且并行度为 32，那么有 32 个生产者服务器执行表的扫描，并把结果送给 32 个消费者服务器，由它们执行排序。如果还需要后续操作，比如另一个排序，那么两组服务的角色互换。原先执行表扫描的服务器扮演消费者的角色，并使用第一次排序产生的输出来执行第二次排序。这样，通过在两组服务器之间来回传递数据，将这两组服务器的角色替换为生产者和消费者，就可以进行一系列的操作。服务器之间的通信通过共享内存硬件上的内存缓冲区进行，或者通过 MPP(无共享)配置和集群(共享磁盘)系统上的高速网络连接进行。

对于无共享的系统，进程之间访问磁盘数据的代价并不相同。在一个结点上运行的可以直接访问设备的进程比需要通过网络检索数据的进程能更快地处理在该设备上的数据。在给并行执行的服务器分配工作时，Oracle 利用了有关设备 - 结点、设备 - 进程邻近关系的知识，即直接访问设备的能力。

28.4.4 结果高速缓存

Oracle 的结果高速缓存特性允许查询或查询块(如：查询中引用的视图)的结果被高速缓存在内存中，如果相同查询被重新执行就可以重用。在底层表上的数据更新会使高速缓存的结果失效，所以这
[1179] 个特性对于相对静止表上的查询且查询结果集相对小的情况最合适。考虑一个使用的例子，Web 页面的某些部分存放在数据库中，相对于它的访问频率是不经常改变的。对于这样一个应用，结果高速缓存是比使用物化视图更轻量级的选择，后者可能需要显式地创建和管理新的持久数据库对象。

28.5 并发控制与恢复

Oracle 支持的并发控制和恢复技术提供了许多有用的特性。

28.5.1 并发控制

Oracle 的多版本并发控制机制是基于 15.7 节中描述的快照隔离协议。只读查询被赋予一个读一致性数据快照，它是存在于特定时刻的数据库的视图，包含所有在那个时刻提交的更新，不包括任何在那个时刻还没有提交的更新。这样就不使用读锁，只读查询在锁的方面不妨碍其他数据库的活动。

Oracle 同时支持语句级和事务级的读一致性：在开始执行一条语句或者一个事务(这要看采用何种一致性级别)时，Oracle 确定当前的系统改变号(System Change Number，SCN)。SCN 实质上充当一个时间戳，只是其中的时间是以事务提交来计算的，而不是真实时间。

如果在查询的过程中发现一个数据块的 SCN 比查询相关的 SCN 更高，说明显在原来查询的 SCN 的未来之后，该数据块被其他一些事务修改过，这些事务可能已经提交，也可能没有提交。因此，这个

块中的数据就不能出现在存在于该查询的 SCN 时刻的数据库的一致性视图中。反之，必须使用这个块中的一个更老的数据版本；特别是具有不超过该查询的 SCN 且最高的 SCN 的版本。Oracle 从撤销段中找回数据的那个版本(撤销段在 28.5.2 中讲述)。因此，只要撤销空间足够大，即使在自查询开始执行以来数据项已修改了好几次，Oracle 仍能返回该查询的一致性结果。如果具有所需 SCN 的块已不在撤销段中存在，查询将返回一个错误。它指明在给定系统上的活动的情况下，撤销表空间的大小设置不合理。

在 Oracle 并发模型中，读操作并不阻碍写操作，同时写操作也不阻碍读操作。该性质支持高并发度。特别地，这种方案允许在有大量事务性活动的系统上执行长时间运行的查询(例如报表查询)。在对查询使用读锁的数据库系统中，这种情形通常会出问题，因为查询可能要么得不到锁，要么长时间 1180
封锁大量数据，从而阻碍了这些数据上的事务性活动，并降低了并发性。(有些系统中使用的一种替代方法是采用较低级别的一致性，比如二级一致性，但是这样可能造成不一致的查询结果。)

Oracle 的并发模型用作闪回(flashback)特性的基础。这种特性允许用户在其会话中设置特定的 SCN 号或真实时间，然后对那个时刻的数据执行操作(假定那时候的数据还在撤销段中存在)。通常在一个数据库系统中，修改操作一旦提交，那就无法回到数据原来的状态。除非从备份中执行时间点恢复。然而，恢复一个很大的数据库的开销很大，尤其如果目的仅仅是为了找回被用户不小心删掉的数据项。闪回特性提供了一种简单得多的机制来处理用户错误。闪回特性具有将一个表或一个完整数据库恢复到某个更早时间点而无需从备份中恢复的能力，当数据在更早时间点上存在时对它们执行查询的能力，追踪一行或多行随时间如何变化的能力，在事务级别检查数据库变化的能力。

除了通过常规的撤销(undo)record能够做到的之外，我们可能希望能够追踪表的变化。(例如，公司管理章程可能需要在特定年限内可以追踪这样的变化。)为了这个目的，可以通过闪回归档(flashback archive)特性来追踪一个表，它创建一个表内部的历史版本。一个后台进程将撤销信息转换为历史表中的项，这可用于提供任意长时期内的闪回功能。

Oracle 支持两种 ANSI/ISO 隔离性级别，**读已提交**(read committed)和**可串行化**(serializable)。它不支持读脏数据，因为这也不需要。语句级的读一致性对应于读已提交隔离性级别，而事务级的读一致性对应于可串行化的隔离性级别。可以在会话中或者在单独的事务中设置隔离性级别。默认的是语句级别的读一致性(即读已提交)。

Oracle 采用行级别封锁，更新不同的行并不冲突。如果两个写操作试图修改同一行，那么一个必须等另一个要么提交，要么回滚，然后它才能要么返回一个写冲突错误，要么开始修改该行。写冲突错误的检测基于 15.7 节介绍的最先更新者胜版本的快照隔离(15.7 节还描述了会与快照隔离同时发生的非可串行化执行的特定情况，并概述了防止这种问题的技术)。在整个事务阶段，封锁都不释放。

除了用行级别封锁来防止 DML 活动引起的不一致性之外，Oracle 还使用表的封锁来防止 DDL 活动引起的不一致性。这种锁防止了当一个用户在一个未提交的事务正在访问某个表时，另外一个用户来删除该表。Oracle 在其常规的并发控制中，并不使用锁升级来把行锁升级为表锁。 1181

Oracle 自动检测死锁，解除死锁的方式是回滚陷入死锁中的一个事务。

Oracle 支持自治事务，这是在其他一些事务之内产生的独立事务。当 Oracle 调用自治事务时，它在一个隔离的环境中产生一个新事务。在控制回到调用事务之前，该新事务要么提交，要么回滚。Oracle 支持自治事务的多级嵌套。

28.5.2 恢复的基本结构

在 28.5.1 节描述的 Oracle 闪回技术可以用作一种恢复机制，但 Oracle 同样支持文件物理备份媒介的恢复。这里我们描述这种更传统的备份和恢复形式。

要了解 Oracle 怎样从故障(如磁盘崩溃)中恢复，了解所涉及的基本结构是很重要的。除了包括表和索引的数据文件，还有控制文件、redo 日志、归档的 redo 日志以及撤销段。

控制文件包含操作数据库所需的各种元数据，包括关于备份的信息。

Oracle 在 redo 日志中记录了数据库缓冲区中所有的事务性修改。它包括两个或者更多的文件。不管事务最终是否提交，它把修改作为引发该修改的操作的一部分记录下来。它不仅记录表中数据的改

变，还记录对索引和撤销段的改变。当 redo 日志填完后，由一个或者几个后台进程来进行归档（如果数据库在日志归档（archivelog）模式下运行）。

撤销段包括关于数据旧版本的信息（即撤销信息）。除了在 Oracle 的一致性模型中扮演的角色之外，这些信息还用于当修改数据项的事务回滚时，恢复这些数据项的旧版本。

为了能够从存储器故障中恢复，数据文件和控制文件应该定期备份。备份频率决定了在最坏情况下的恢复时间，因为如果备份是旧的，就要花更长的时间来恢复。Oracle 支持热备份，即在有事务性活动的联机数据库上执行备份。

在从备份中恢复的期间，Oracle 执行两个步骤来达到正好在故障前一刻存在的数据库的一致性状态。首先，Oracle 通过将（已存档的）redo 日志应用到备份上向前滚。此动作将数据库恢复到故障时存在的状态，但是由于 redo 日志中包含未提交数据，因此还不一定是一致性状态。第二步，Oracle 通过使用撤销段数据来回滚未提交事务，这样数据库就达到一致的状态了。

自最后一次备份以来有大量事务性活动的数据库上进行恢复会很耗时。Oracle 支持并行恢复，利
1182 用几个进程同时应用 redo 信息。Oracle 提供了一个图形化用户界面工具——恢复管理器（recovery manager），可自动执行与备份和恢复相关的大多数任务。

28.5.3 Oracle 数据卫士

为了保证高可用性，Oracle 提供了备用数据库特性——**数据卫士**（data guard）。（该特性和 16.9 节中讲述的远程备份相同。）备用数据库是常规数据库的拷贝，安装在另一个单独的系统中。如果主系统发生灾难性故障，那么备用系统被激活并接管控制，从而减少了故障对可用性的影响。Oracle 通过不断应用从主数据库传来的归档的 redo 日志，来保持备用数据库是最新的。备用数据库可以只读方式上线，并用于报表和决策支持查询。

28.6 系统体系结构

每当数据库应用执行 SQL 语句时，就有一个操作系统进程执行数据库服务器中的代码。可以通过配置 Oracle 来决定该操作系统进程是被它正处理的语句所独占专用，还是可以在多条语句间共享。后一种配置称为**共享服务器**（shared server），在进程和内存体系结构上有一些不同的特性。我们将先讨论**专用服务器**（dedicated server）体系结构，稍后讨论多线程服务器体系结构。

28.6.1 专用服务器：内存结构

Oracle 所用的内存主要分为三类：软件代码区（它是 Oracle 服务器代码驻留的内存的部分）、系统全局区（System Global Area，SGA）和程序全局区（Program Global Area，PGA）。

系统给每个进程分配一个 PGA 来保存其局部数据和控制信息。这个区域包含各种会话数据的堆栈空间以及正在执行的 SQL 语句所用的私有内存。它还包括执行语句时可能进行的排序和散列操作所需的内存。这类操作的性能对于可用内存的大小是敏感的。例如，与必须将溢出数据存放在磁盘上的散列连接相比，可以在内存中执行的散列连接要比需要利用磁盘的情况更快。由于可能有大量活跃的排序和散列操作同时存在（因为存在多个查询，同时每个查询内存在多个操作），确定为每个操作分配多少内存是主要的，特别是当系统负载可能波动时。如果一个操作不必要地溢出到磁盘，内存分配不足会导致额外的磁盘 I/O；而内存分配过多会引起系统颠簸。Oracle 让数据库管理员为可用于这些操作的
1183 内存总量指定目标参数。此目标的大小通常可基于系统可用的内存总量和关于这些内存应该如何在各种 Oracle 和非 Oracle 活动之间进行划分的一些计算。Oracle 会动态决定将目标内可用内存在活跃操作之间分配的最优方法，以最大化吞吐量。内存分配算法知道不同操作的内存与性能之间的关系，试图确保尽可能高效地利用可用内存。

SGA 是存放用户间共享结构的内存区域。它由几种主要结构组成，包括：

- **缓冲区高速缓存**（buffer cache）。这种高速缓存将频繁访问的数据块（来自表或索引）保存在内存中，以减少执行物理磁盘 I/O 的需求。除在全表扫描时的块访问之外，使用最近最少使用替换策略。但是，Oracle 允许建立多个具有不同数据替换策略的缓冲池。有些 Oracle 操作绕过缓冲

区高速缓存，直接从磁盘读取数据。

- **redo 日志缓冲区**(redo log buffer)。这种缓冲区用来保存还没有写到磁盘上的 redo 日志部分。
- **共享池**(shared pool)。Oracle 通过最小化每个用户所需的内存量，来寻求最大化的、可以并发使用数据库的用户数量。这其中一个很重要的概念是对 SQL 语句和用 PL/SQL 书写的过程化代码的内部表示的共享能力。当多个用户执行相同的 SQL 语句时，他们能共享表示该语句执行计划的大多数数据结构。只有每个语句调用的局部数据才需要放在私有内存中。

  表示 SQL 语句的数据结构中的可共享部分(包括语句文本)存放在共享池中。在共享池中高速缓存 SQL 语句也节约编译时间，因为对已被高速缓存的语句的新调用不必进行完整的编译过程。判断SQL语句是否与共享池中所存放的SQL语句相同，是基于精确的文本匹配和设置特定的会话参数来进行的。Oracle 能够用绑定变量来自动替换 SQL 语句的常量；若以后的查询除了常量值外都与共享池中以前的查询是一样的，则它们匹配。

  共享池还高速缓存了字典信息和各种控制结构。高速缓存字典元数据对缩短 SQL 语句的编译时间很重要。另外，共享池用于 Oracle 的结果高速缓存特性。

28.6.2 专用服务器：进程结构

执行 Oracle 服务器代码的进程有两种：执行 SQL 语句的服务器进程，执行各种管理以及与性能相 [1184]
关任务的后台进程。这些进程中有些是可选的，在某些情况下，相同类型的多个进程可用于性能因素。Oracle 能够生成大约 24 个不同类型的后台进程。一些最重要的后台进程有：

- **数据库写**(database writer)**进程**。当一个缓冲区从缓冲区高速缓存中移出时，如果自从它进入高速缓存以来已经改动过，那么它必须写回到磁盘中。这个任务就由数据库写进程来执行。通过释放缓冲区高速缓存中的空间，有利于提高系统性能。
- **日志写**(log writer)**进程**。日志写进程把 redo 日志缓冲区中的项写到磁盘上的 redo 日志文件中。每当一个事务提交时，它还把一条提交记录写到磁盘上。
- **检查点**(checkpoint)进程。当出现检查点时，检查点进程更新数据文件头。
- **系统监控**(system monitor)**进程**。该进程在必要时执行崩溃恢复。它还执行一些空间管理，回收临时段中未使用的空间。
- **进程监控**(process monitor)**进程**。该进程为失败的服务器进程执行进程恢复，释放资源并执行各种清理操作。
- **恢复**(recoverer)**进程**。恢复进程处理失效，并为分布式事务执行清理。
- **归档**(archiver)**进程**。每当在线日志文件写满后，归档进程把在线 redo 日志文件复制为归档 redo 日志。

28.6.3 共享服务器

通过在语句之间共享服务器进程，共享服务器配置增加了给定数量的服务器进程能够支持的用户数量。它与专用服务器体系结构主要在以下这些方面有所不同：

- 一个后台分派进程把用户请求路由给下一个可用服务器进程。在这么做时，它使用了 SGA 中的一个请求队列和一个响应队列。分派进程把新的请求放到请求队列中，服务器进程可以从中选取请求。当一个服务进程完成某个请求时，它把结果放到响应队列中。分派进程可从中选取结果并返回给用户。
- 由于一个服务器进程在多条 SQL 语句间共享，Oracle 并不在 PGA 中保留私有数据。相反，它把会话相关的数据存放在 SGA 中。 [1185]

28.6.4 Oracle Real Application Clusters

Oracle Real Application Clusters(RAC，Oracle 真实应用集群)特性允许在同一个数据库上运行多个 Oracle 实例(回想一下，在 Oracle 术语中，实例是后台进程和内存区域的结合)。这个特性使得 Oracle 能够在集群和 MPP(共享磁盘和无共享)的硬件体系结构上运行。将多个结点集群起来的能力大大提高了可扩展性和可用性，这在联机事务处理和数据仓库环境中都很有用。

这个特性的可扩展性优点是很明显的，因为越多的结点意味着越强的处理能力。在无共享体系结构中，向一个集群添加结点通常需要在结点间重新分配数据。Oracle 使用了共享磁盘体系结构，其中所有结点都能访问所有数据，因此可以添加更多结点到 RAC 集群中，而无需担心怎样在结点间划分数据。Oracle 通过诸如邻近关系和按划分连接这样的特性进一步优化了对硬件的使用。

RAC 也用来获得高可用性。如果一个结点失效，应用程序仍可利用剩余结点来访问数据库。剩余实例将自动回滚在失效结点上正被处理的未提交事务，以防它们阻碍剩余结点上的活动。RAC 还允许滚动地打补丁，所以每次可以将软件补丁应用到一个结点上而无需数据库停机。

Oracle 的共享磁盘体系结构避免了无共享体系结构在关于磁盘上的数据要么对结点是本地的，要么不是这个方面所存在的许多问题。在同一个数据库上运行多个实例还带来了一些在单实例情况下所不存在的技术问题。虽然有时可能将一个应用在多个结点间进行划分，以使各结点很少访问同一数据，但总有重叠的可能性，这会影响高速缓存管理。为了实现在多个结点上有效的高速缓存管理，Oracle 的**高速缓存融合**(cache fusion)特性允许数据块使用内部连接直接在不同实例的高速缓存之间流动，而不用写到磁盘上。

28.6.5 自动存储管理器

自动存储管理器(Automatic Storage Manager，ASM)是 Oracle 开发的一个容量管理器和文件系统。虽然 Oracle 可以与其他容量管理器和文件系统以及原始设备一起使用，然而 ASM 在优化性能的同时是为 Oracle 数据库特别设计的用于简化存储管理的。

ASM 管理磁盘的集合，称为**磁盘组**(disk group)，并对数据库开放了一个文件系统接口。(记住 Oracle 的表空间是用数据文件定义的。)组成 ASM 磁盘的实例包括磁盘或磁盘阵列的分区、逻辑卷和网络附属文件。ASM 将数据自动地分到一个磁盘组中的磁盘上，并为不同层次的镜像提供若干选择。

如果磁盘配置改变了，比如，为增加存储容量而加入更多磁盘时，磁盘组可能需要重新平衡以使
1186 数据均匀分布在所有磁盘上。重新平衡操作可以在数据库保持完全操作的同时在后台完成，并且对数据库性能的影响最小。

28.6.6 Oracle Exadata

Exadata 是一组可以运行在特定类型存储硬件上的存储器阵列 CPU 上的 Oracle 的库。虽然 Oracle 根本上基于共享磁盘体系结构，Exadata 在如下方面具有无共享的风格：某些通常在数据库服务器上执行的操作被移到那些只能访问自己本地数据的存储单元上。(每个存储单元由多个磁盘和一些多核 CPU 组成。)

将特定类型的处理过程卸载到存储器 CPU 的主要好处包括：

- 它允许对可用的处理能力的量进行大的但相对经济的扩展。
- 极大减少了需要从存储单元传送到数据库服务器的数据量，这是非常重要的，因为存储单元和数据库服务器之间的带宽通常是昂贵的，并常常是一个瓶颈。

当在 Exadata 存储器上执行查询时，需要检索的数据量减少了，这得益于若干可以推送到存储单元并在其本地执行的技术：

- **投影**(projection)。一个表可以有几百个列，但一个给定查询可能只需要访问它们的一个很小的子集。存储单元可以投影掉不需要的列且只将相关的列发送回数据库服务器。
- **表过滤**(table filtering)。数据库服务器可以给存储单元发送一个针对表的谓词列表，且只将匹配这些谓词的行回传给服务器。
- **连接过滤**(join filtering)。过滤机制允许使用布隆过滤器(Bloom filter)形式的谓词，它还可以基于连接条件将行过滤掉。

总地来说，将这些技术卸载到存储单元可以在数量级上加速查询处理。它需要存储单元除了可以将常规的、未修改的数据块传回给服务器，还可以传回特定行和列被移除的压缩版本。这种能力反过来要求存储软件能够理解 Oracle 的块格式和数据类型，并包含 Oracle 的表达式和谓词计算例程。

1187 除了给查询处理提供好处之外，Exadata 也可以通过执行块级别的修改追踪并只返回修改过的块来

加速增量备份。当创建一个新的表空间时，盘区的格式化工作也被卸载给了 Exadata 存储器。

Exadata 存储器支持所有的 Oracle 常规特性。一个数据库可以同时包含 Exadata 和非 Exadata 存储器。

28.7 复制、分布以及外部数据

Oracle 提供对具有两阶段提交的事务的复制和分布的支持。

28.7.1 复制

Oracle 支持几种类型的复制。（参阅 19.2.1 节关于复制的介绍。）其中的一种是，主站点的数据以物化视图的形式复制给其他站点。物化视图并不需要包含主站点上的所有数据，例如，它会由于安全因素而排除表中的特定列。Oracle 支持两种类型的物化视图用于复制：只读的和可更新的。可更新的物化视图可以修改，并将修改传播到主站点上的表。但只读的物化视图允许的视图定义范围更广。比如只读物化视图能用主站点表上的集合操作来定义。对主站点数据的修改通过物化视图刷新机制传播到副本。

Oracle 还支持同一数据有多个主站点，其中所有主站点处于同等地位。被复制的表能够在任何一个主站点上更新，并将此更新传播到其他站点。更新的传播可以是同步的，也可以是异步的。

对于异步复制，更新信息成批发送给其他主站点并被采用。由于相同数据可以被不同站点修改，这些修改可能相互冲突，这就可能需要基于某些商务规则的冲突解决策略。Oracle 提供了许多内置的冲突解决方法，如果需要它也允许用户书写他们自己的方法。

在同步复制中，一个主站点上的更新操作马上被传播到所有的其他站点。

28.7.2 分布式数据库

Oracle 支持跨越多个在不同系统上的数据库的查询和事务。通过网关的使用，远端系统可以包括非 Oracle 数据库。Oracle 具有内置的功能来优化包含了不同站点上的表的查询，检索相关数据并返回结果，就好像这是个标准的本地查询一样。Oracle 还通过内置的两阶段提交协议来透明地支持跨越多个站点的事务。

1188

28.7.3 外部数据源

Oracle 有几种支持外部数据源的机制。最常见的用途是在数据仓库中，从事务性系统定期加载大量数据。

28.7.3.1 SQL* Loader

Oracle 有一个直接加载工具 SQL* 加载器（SQL* Loader），它支持从外部文件中快速并行加载大量数据。它支持很多种数据格式，并可以在加载的数据上进行各种过滤操作。

28.7.3.2 外部表

Oracle 允许外部数据源（如平面文件）像普通表一样，在查询的 **from** 子句中引用。Oracle 通过描述 Oracle 列类型以及从外部数据到这些列的映射的元数据来定义外部表，还需要访问外部数据的访问驱动程序。Oracle 为平面文件提供了默认的驱动程序。

外部表特性主要是为了在数据仓库环境中进行抽取、转换和加载（ETL）操作。使用下述语句可以将数据从平面文件加载到数据仓库中：

```
create table table as
    select ... from <external table>
    where ...
```

通过在 **select** 列表或 **where** 子句中添加对数据的操作，转换和过滤操作能够作为同一条 SQL 语句的一部分来执行。由于这些操作既能用本地的 SQL 来表达，也能用 PL/SQL 或 Java 书写的函数来表达，外部表特性提供了非常强大的机制来表达各种数据转换和过滤操作。至于可扩展性，可以通过 Oracle 的并发执行特性来将外部表的访问并发化。

28.7.3.3 数据抽取导出和导入

Oracle 提供了一个导出工具来把数据和元数据卸载到转储文件。这些文件是使用一种专有格式的

普通文件，可以移动到另一个系统并使用相应的导入工具加载到另一个 Oracle 数据库。

28.8 数据库管理工具

Oracle 为用户提供了一系列用于系统管理和应用开发的工具和特性。在最近发布的 Oracle 中，着
[1189] 重强调了可管理性的概念，也就是说，减少创建和管理 Oracle 数据库的各个方面的复杂度。这需要许多方面的努力，包括数据库创建、调优、空间管理、存储管理、备份与恢复、内存管理、性能诊断和工作负载管理。

28.8.1 Oracle 企业管理器

Oracle 企业管理器（Oracle Enterprise Manager，OEM）是 Oracle 用于数据库系统管理的主要工具。它提供了易于使用的图形化用户界面，支持与管理 Oracle 数据库相关的大部分任务，包括配置、性能监测、资源管理、安全管理以及访问各种向导。除了数据库管理，OEM 还提供对 Oracle 应用和中间件软件栈的集成管理。

28.8.2 自动工作负载存储

自动工作负载存储（Automatic Workload Repository，AWR）是 Oracle 实现可管理性的基础设施的核心部分之一。Oracle 监控数据库系统上的活动并记录与工作负载和资源消耗相关的各种信息，并按定期的间隔在 AWR 中记录它们。通过追踪工作负载随时间变化的特点，Oracle 可以检测和帮助诊断从正常行为发生的偏离，如一个查询严重的性能退化、封锁竞争和 CPU 瓶颈。

在 AWR 中记录的信息为各种向导提供了基础，这些向导提供对系统性能各个方面的分析以及如何提高性能的建议。Oracle 具有 SQL 调优的向导、创建访问结构（如索引和物化视图）的向导、以及内存大小的向导。Oracle 还提供段碎片整理和 undo 大小的向导。

28.8.3 数据库资源管理

数据库管理员需要能够控制如何在各个用户或用户组之间分配硬件的处理能力。某些组可能执行交互式查询，则响应时间很关键；另一些组可能执行长程（long-running）报表，它可以在系统负载低时，作为后台中的批处理工作运行。还有一个重要方面是能够防止用户不小心提交代价特别昂贵的即席查询，这样的查询将过度地耽误其他用户。

Oracle 的数据库资源管理特性允许数据库管理员把用户划分成资源消费者组。每组有不同的优先级和性质。比如，高优先级的交互式用户组可以保证至少 60% 的 CPU。剩下的 CPU，加上高优先级组的 60% 中没有使用的任何部分，可以在优先级较低的资源消费者组之间进行分配。真正很低优先级的组可以只分配 0% 的 CPU。这就意味着这个组提交的查询，只能在有空闲的 CPU 周期可用时才能运行。
[1190] 可以为各个组设定并行执行的并行度限制。数据库管理员还能为每个组的一条 SQL 语句运行多长时间设定时间限制。当用户提交语句时，资源管理器估计执行这条查询所花费的时间，并且如果语句违反限制就返回一个错误。资源管理器还能为每个资源消费者组限制同时活跃的用户会话数量。资源管理器可以控制的其他资源包括 undo 空间。

28.9 数据挖掘

Oracle 数据挖掘（Oracle data mining）提供了一系列将数据挖掘过程嵌在数据库内部的算法。既可以在训练数据集上构建模型，也可以将该模型运用到对实际生产数据的评分中。与使用其他数据挖掘引擎相比，数据不需要离开数据库是一个重要优势。将潜在很大的数据集提取出来并插入到一个单独的引擎中，不仅很麻烦且代价昂贵，还可能阻碍当新数据进入数据库时就即时对它们进行评分。Oracle 提供用于有指导的学习和无监督指导的算法，包括：

- 分类——朴素贝叶斯、广义线性模型、支持向量机和决策树。
- 回归——支持向量机和广义线性模型。
- 属性重要性分析——最小描述长度。
- 异常检测——一类支持向量机。

- 聚类——增强的 k 均值聚类和正交的划分聚类。
- 关联规则——Apriori。
- 特征提取——非负的矩阵分解。

另外，Oracle 在数据库内部提供了一系列的统计函数，涵盖的领域包括线性回归、关联、交叉表、假设检验、分布拟合和 Pareto 分析。

Oracle 为数据挖掘功能提供两个界面，一个基于 Java，另一个基于 Oracle 的过程性语言 PL/SQL。一旦在 Oracle 数据库上创建了一个模型，它可以转移部署到其他 Oracle 数据库上。

文献注解

有关 Oracle 产品的最新产品信息，包括相关文档，都可以在以下 Web 站点找到：http://www.oracle.com 和 http://technet.oracle.com。

Oracle 用于为操作（如散列和排序）分配可用内存的智能算法在 Dageville 和 Zait[2002]中讨论。Murthy 和 Banerjee[2003]讨论了 XML 模式。Ptiss 和 Potapov[2003]描述了 Oralce 中的表压缩。Dageville 等[2004]讲述了自动的 SQL 调优。优化器的基于代价的查询转换框架在 Ahmed 等[2006]中介绍。Ziauddin 等[2008]讨论了 SQL 计划管理特性。Antoshenkov[1995]描述了 Oracle 中使用的字节对齐位图压缩技术，也可以查阅 Johnson[1999]。

IBM DB2 Universal Database

Sriram Padmanabhan

IBM 公司的 DB2 Universal Database 产品家族包括旗舰数据库服务器和最新的用于商务智能、信息集成和内容管理的相关产品。DB2 Universal Database Server 可用于多种硬件和操作系统平台上。它支持的服务器平台包括诸如大型主机、大规模并行处理器(MPP)和大型对称多处理器(SMP)服务器这样的高端系统；诸如四路和八路 SMP 这样的中等规模系统；工作站；甚至小型手持设备。它支持的操作系统包括 UNIX 变体，如 Linux、IBM AIX、Solaris 和 HP-UX，以及 Microsoft Windows、IBM MVS、IBM VM、IBM OS/400 和许多其他操作系统。DB2 Everyplace 版本支持诸如 PalmOS 和 Windows CE 这样的操作系统。甚至有一个免费的 DB2 版本，称为 DB2 Express-C，因为 DB2 接口和服务具有可移植性，应用程序可以从低端平台无缝移植到高端服务器。除了核心数据库引擎外，DB2 家族还包括一些其他产品，提供工具、管理、复制、分布式数据访问、普适数据访问、OLAP 和许多其他特性。图 29-1 描述了这个家族的不同产品。



图 29-1 DB2 产品家族

29.1 概述

DB2 的起源可以追溯到 IBM 的 Almaden 研究中心(当时称作 IBM San Jose 研究实验室)的 System R 项目。第一个 DB2 产品是 1984 年在 IBM 大型主机平台上发布的，随后运行于其他平台的版本陆续发布。IBM 研究成果在下述领域不断改进 DB2 产品：事务处理(先写日志和 ARIES 恢复算法)、查询处理和优化(Starburst)、并行处理(DB2 并行版本)、主动数据库支持(约束和触发器)、高级查询和数据仓库技术，例如物化视图、多维聚类、"自主"特性和对象-关系支持(ADT、UDF)。

因为 IBM 支持很多服务器和操作系统平台，DB2 数据库引擎由四种代码基本类型组成：(1)Linux、UNIX 和 Windows，(2)z/OS，(3)VM，(4)OS/400。它们都支持一个共同的数据定义语言 SQL 和管理接口的子集。然而，由于这些引擎的平台起源不同，它们多少存在一些不同的特征。本章重点集中在支持 Linux、UNIX 和 Windows 的 DB2 Universal Database(UDB)引擎。在其他 DB2 系统中有意思的特定

特性在合适的章节中强调。

到 2009 年为止，支持 Linux、UNIX 和 Windows 的 DB2 UDB 的最新版本是版本 9.7。DB2 9.7 版本包括若干新的特性，如将 XML 的原生支持扩展到无共享环境、对表和索引的本机压缩、自动存储管理和对过程化语言（如 SQL PL 和 Oracle 的 PL/SQL）改进的支持。

29.2　数据库设计工具

大多数工业数据库设计和计算机辅助软件工程（CASE）工具都可以用来设计 DB2 数据库。特别地，诸如 ERWin 和 Rational Rose 这样的数据建模工具允许设计者生成 DB2 特有的 DDL 语法。比如，Rational Rose 的 UML Data Modeler 工具可以为用户定义类型生成 DB2 特有的 **create distinct type** DDL 语句，并可以在随后的列定义里使用。大部分设计工具还支持逆向工程特性，读取 DB2 的目录表并为附加操作建立逻辑设计。这些工具支持约束和索引的生成。 1193 1194

DB2 使用 SQL 提供了对许多逻辑的和物理的数据库特性的支持。这些特性包括使用约束、触发器和 SQL 构造的递归。类似地，通过使用 SQL 语句还支持一些物理数据库特性，比如表空间、缓冲池以及分区。DB2 的控制中心的图形化用户界面（The Control Center GUI）工具允许设计者或管理员为这些特性发布恰当的 DDL。另一个工具叫 *db2look*，允许管理员获得一整套数据库 DDL 语句，包括表空间、表、索引、约束、触发器以及为测试或复制而创建精确的数据库模式副本而使用的函数。

DB2 控制中心包括多种与设计和管理相关的工具。对于设计，控制中心提供了服务器及其数据库、表、视图和所有其他对象的树状视图。它还允许用户定义新的对象，创建即席的 SQL 查询并查看查询结果。ETL、OLAP、复制和联邦的设计工具也整合在控制中心中。整个 DB2 家族都支持用于数据库定义和相关工具的控制中心。DB2 还为使用 IBM Rational Application Developer 产品和 Microsoft Visual Studio 产品开发的应用提供插件模块。

29.3　SQL 的变化和扩展

DB2 为数据库处理的各个方面提供丰富的 SQL 特性集。许多 DB2 特性和语法为 SQL-92 或 SQL-1999标准奠定了基础。在本节中，我们涵盖在 DB2 的 UDB 版本 8 中的 XML 对象 - 关系和应用集成特性，还有一些来自版本 9 的新特性。

29.3.1　XML 特性

DB2 中已包含了丰富的 XML 函数集。下面列出了几个可以用在 SQL 中的重要的 XML 函数，作为对 SQL 扩展的 SQL/XML（在前面 23.6.3 节中介绍过的）的一部分：

- **xmlelement**。用给定名称构建元素标记。例如函数调用 **xmlelement**（book）创建了 book 元素。
- **xmlattributes**。给元素构建属性集合。
- **xmlforest**。通过变元构建一个 XML 元素序列。
- **xmlconcat**。返回可变数量的 XML 变元的串接。
- **xmlserialize**。提供变元的面向字符的序列化版本。 1195
- **xmlagg**。返回一组 XML 值的串接。
- **xml2clob**。构建 XML 的字符大对象（**clob**）表示。然后这个 **clob** 可以被 SQL 应用检索。

XML 函数可以高效地合并在 SQL 中，以提供扩展的 XML 操纵能力。例如，假设有人需要从关系表 *orders*、*lineitem* 和 *product* 为第 349 号订单创建购买 - 订单的 XML 文档。在图 29-2 中我们给出了用来创建这样一份购买订单的带 XML 扩展的 SQL 查询。结果输出如图 29-3 所示。

```
select xmlelement(name "PO",
                  xmlattributes(poid, orderdate),
                  (select xmlagg(xmlelement(name "item",
                                            xmlattributes(itemid, qty, shipdate),
                                            (select xmlelement(name "itemdesc",
                                                               xmlattributes(name, price))
                                             from product
                                             where product.itemid = lineitem.itemid)))
                   from lineitem
                   where lineitem.poid = orders.poid))
from orders
where orders.poid= 349;
```

图 29-2 DB2 SQL XML 查询

```
<PO poid = "349" orderdate = "2004-10-01">
    <item itemid= "1", qty="10", shipdate="2004-10-03">
    <itemdesc name = "IBM ThinkPad T41", Price = "1000.00 USD"/>
    </item>
</PO>
```

图 29-3 id = 349 的 XML 形式的购买订单

DB2 的版本 9 以 **xml** 类型的形式支持 XML 数据的本地存储以及对 XQuery 语言的原生支持。引入了专门的存储、索引、查询处理和优化技术来有效地处理 XML 数据和用 XQuery 语言书写的查询，并扩展到 API 以处理 XML 数据和 XQuery。

29.3.2 数据类型的支持

DB2 提供对用户自定义数据类型（UDT）的支持。用户可以定义 *distinct* 或 *structured* 数据类型。distinct 数据类型是基于 DB2 内置数据类型的。但是，用户可以为这些新的类型定义附加的或替代的语义。比如，用户可以使用下述语句定义一个名为 *us_dollar* 的 distinct 数据类型：

```
create distinct type us_dollar as decimal(9, 2);
```

接下来，此用户可以在一个表中用 *us_dollar* 类型来创建一个字段（比如 *price*）。查询现在可以在谓词中使用这个类型的字段，如下所示：

```
select product from us_sales
where price > us_dollar (1000);
```

structured 数据类型是通常由两个或多个属性组成的复杂对象。例如，可以用下面的 DDL 来创建名为 *department_t* 的 structured 类型：

```
create type department_t as
        (deptname varchar(32),
        dephead varchar(32),
        faculty_count integer)
mode db2/sql;

create type point_t as
        (x_coord float,
        y_coord float)
mode db2/sql;
```

structured 类型可用于定义有类型（typed）的表：

```
create table dept of department_t;
```

可以创建一个类型层次和层次中继承特定方法和权限的表。structured 类型还可用于在表的列中定义嵌套属性。尽管这样的定义会破坏规范化准则，但是它可能适合于面向对象的应用，这些应用依赖于对象上的封装和定义良好的方法。

29.3.3 用户自定义函数和方法

另一个重要特性是用户可以定义他们自己的函数和方法的能力。随后这些函数可以用在 SQL 语

句和查询中。函数可以生成标量(单属性)或表(多属性行)作为它们的结果。用户可以用 **create function** 语句注册函数(标量的或是表的)。可以用普通的程序设计语言,(比如 C 或 Java)或是诸如 REXX 或 PERL 那样的脚本来书写函数。用户自定义函数(UDF)可以在保护(fenced)或非保护(unfenced)模式下操作。在保护模式下,函数由一个单独线程在它自己的地址空间中执行。在非保护模式下,数据库处理代理允许在服务器地址空间执行函数。UDF 可以定义一个便签本(工作)区,用以维护跨越不同调用的局部和静态变量。这样,UDF 可以对作为其输入的中间行进行有效的操作。在图 29-4 中,我们展示了 DB2 中的一个名为 *db2gse.GeogFilterDist* 的 UDF 定义。它指向一个特定的执行实际功能的外部方法。

1196
1197

```
create function db2gse.GeogFilterDist (
        operation integer, g1XMin double, g1XMax double,
        g1YMin double, g1YMax double, dist double,
        g2XMin double, g2XMax double, g2YMin double,
        g2YMax double )
    returns integer
    specific db2gse.GeogFilterDist
    external name 'db2gsefn!gsegeFilterDist'
    language C
    parameter style db2 sql
    deterministic
    not fenced
    threadsafe
    called on null input
    no sql
    no external action
    no scratchpad
    no final call
    allow parallel
    no dbinfo;
```

图 29-4 一个 UDF 的定义

方法是定义对象行为的另一种特性。与 UDF 不同,它们用特定结构的数据类型紧密封装。通过使用 **create method** 语句来注册方法。

DB2 同样支持对 SQL 的过程化扩展,使用的是 DB2 的 SQL PL 扩展,包括过程、函数和控制流。(SQL 标准的过程化特性在 5.2 节中介绍过。)另外,到版本 9.7 为止,为了兼容在 Oracle 上开发的应用,DB2 还支持很多 Oracle 的 PL/SQL 语言。

29.3.4 大对象

新的数据库应用需要操纵文本、图像、视频以及其他类型数据的能力。这些数据通常是相当大的数据。DB2 通过提供三种不同的大对象(LOB)类型来满足这些需求。每个 LOB 可以有 2GB 的大小。DB2 中的大对象是:(1)**二进制大对象**(blob),(2)**单字节字符大对象**(clob),(3)**双字节字符大对象**(dbclob)。DB2 将这些 LOB 作为单独的对象来组织,在表的每行维护指向其对应 LOB 的指针。根据应用需求,用户可以注册操纵这些 LOB 的 UDF。

1198

29.3.5 索引扩展和约束

DB2 新近的一个特性允许用户使用 **create index extension** 语句将创建索引扩展到从 structured 数据类型生成的码上。比如,使用部门名称,通过生成码,用户可以在基于先前定义的 *department_t* 数据类型的属性上创建索引。DB2 的空间扩展器使用索引扩展方法来创建如图 29-5 所示的索引。

最后,用户可以利用 DB2 中丰富的约束检查特性集来增强对象语义,如唯一性、有效性和继承。

29.3.6 Web 服务

DB2 可以将 Web 服务集成为生产者或消费者。利用 SQL 语句,可以定义一个 Web 服务来调用 DB2。DB2 中内置的 Web 服务引擎可处理作为结果的 Web 服务调用,并生成适当的 SOAP 响应。例如,如果一个叫做 *GetRecentActivity(cust_id)* 的 Web 服务调用了下述 SQL,结果应该是该用户最后的事务。

```
create index extension db2gse.spatial_index(
                        gS1 double, gS2 double, gS3 double)
from source key(geometry db2gse.ST_Geometry)
generate key using
        db2gse.GseGridIdxKeyGen(geometry..srid,
        geometry..xMin, geometry..xMax,
        geometry..yMin, geometry..yMax,
        gS1, gS2, gS3)

with target key(srsId integer,
        id integer, gX integer, gY integer, xMin double,
        xMax double, yMin double, yMax double)
search methods <conditions> <actions>
```

图 29-5 DB2 中的空间索引扩展

1199

```
select tm_id, amount, date
from transaction
where cust_id = <input>
order by date
fetch first 1 row only;
```

下面的 SQL 显示了 DB2 作为 Web 服务的消费者的情况。在这个例子中，用户自定义函数 *GetQuote*( ) 是一个 Web 服务。DB2 使用向导的 Web 服务引擎来生成 Web 服务调用。在这个例子中，*GetQuote* 对 *portfolio* 表中每个 *ticker_id* 返回一个数值型的报价值。

```
select ticker_id, GetQuote(ticker_id)
from portfolio;
```

29.3.7 其他特性

通过定义合适的 UDF，DB2 还支持 IBM 的 Websphere MQ 产品。读和写接口都可用 UDF 来定义。这些 UDF 可以并入 SQL 中，以对消息队列执行读或者写。

从版本 9 开始，DB2 支持通过基于标记的访问控制特性进行细粒度的授权，其作用类似于 Oracle 的虚拟私有数据库（在前面 9.7.5 节讲述过）。

29.4 存储和索引

DB2 中的存储和索引体系结构包括文件系统或磁盘管理层、管理缓冲池的服务、数据对象（比如表）、LOB、索引对象以及并发和恢复管理器。在这一节中我们概览通用的存储体系结构。另外，在下一节中我们描述 DB2 版本 8 的一个新的特点，叫做多维聚集。

29.4.1 存储体系结构

DB2 为管理逻辑数据库表提供的存储抽象在多结点和多磁盘环境中很有用。在多结点系统中可以定义结点组来支持在特定结点集上的表划分。这使得在将表分区分配给系统中不同结点时具有完全的灵活性。比如，大表可能划分到系统中的所有结点上，而小表可能只驻留在单个结点上。

在一个结点内，DB2 使用表空间来组织表。一个表空间包括一个或多个容器，容器是对目录、设备或文件的引用。一个表空间可以包含零个或多个数据库对象，如表、索引或 LOB。图 29-6 说明了这些概念。在该图中，为一个结点组定义了两个表空间：*humanres* 表空间分配了四个容器，而 *sched* 表空间只有一个容器。*employee* 和 *department* 表分配给了 *humanres* 表空间，而 *project* 表在 *sched* 表空间中。采用拆分（striping）将 *employee* 和 *department* 表的段（盘区）分配给 *humanres* 表空间的容器。DB2 允许管理员创建系统管理的或是 *DBMS* 管理的表空间。系统管理空间（SMS）是由底层操作系统维护的目录或
1200 文件系统。在 SMS 中，DB2 在目录中创建文件对象，并将数据分配给每个文件。数据管理空间（DMS）是 DB2 控制的原始设备或预分配的文件。这些容器的大小既不能增长也不能缩小。DB2 自己创建分配映射并管理 DMS 表空间。在这两种情况下，页的一个盘区是空间管理的单位。管理员可以为表空间选择盘区大小。

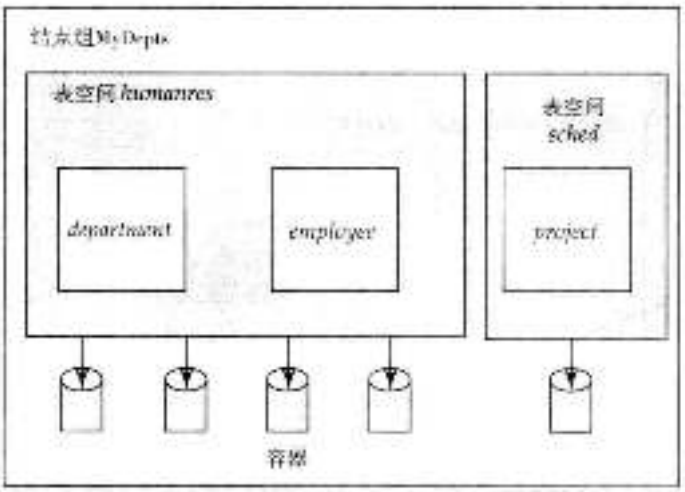


图 29-6 DB2 中的表空间与容器

作为一种默认行为，DB2 支持跨不同容器的拆分。例如，当数据插入到一个新建的表中时，分配第一块盘区给容器。一旦这块盘区满了，下一个数据页以轮转方式分配给下一个容器。拆分提供了两个重要的好处：并行 I/O 和负载均衡。

29.4.2 缓冲池

每个表空间可以与一个或者多个缓冲池相关联，以管理像数据和索引这样的不同对象。缓冲池是维护对象在内存中副本的通用共享的数据区域。这些对象在缓冲池中通常组织成页进行管理。DB2 允许用 SQL 语句来定义缓冲池。通过将缓冲池的配置参数选为自动设置，DB2 版本 8 具备在线地并且自动增长或者缩减缓冲池的能力。管理员能够给缓冲池增加更多页面或者缩减它的大小，而不停止数据库的活动。

```
create bufferpool <buffer-pool> ...                                    1201
alter bufferpool <buffer-pool> size <n>
```

DB2 还支持预取(prefetching)和使用单独线程的异步写(asynchronous write)。数据管理器部件基于查询访问模式触发对数据和索引页的预取。例如，表扫描总会触发预取数据页。索引扫描能触发索引页的预取。如果它们以聚集模式访问，还会触发数据页的预取。预取的数量和预取的大小是需要根据磁盘数和表空间中的容器数来初始化的可配置参数。

29.4.3 表、记录和索引

DB2 把关系数据作为页中的记录来组织。图 29-7 展示了一个表的逻辑视图及其相关索引。这个表包含了一组页。每页包含一组记录，它们要么是用户数据记录，要么是特殊的系统记录。表的每页包含了关于表及其状态的特殊系统记录。DB2 使用一条称为空闲空间控制记录(Free Space Control Record，FSCR)的空间映射记录来寻找表中的空闲空间。FSCR 记录通常包含一个 500 页的空间映射。FSCR 项是一个位掩码，提供页中剩余空间可能性的大致指示。插入或更新算法必须通过执行页中可用空间的物理检查来验证 FSCR 项。

索引也使用页来组织。这些页中包含了索引记录和指向子页面和兄弟页面的指针。DB2 内部提供对 B+树索引机制的支持。B+树索引包含内部页和叶结点页。索引在叶结点级有双向指针来支持正向和反向扫描。叶结点页包含指向表中记录的索引项。表中每条记录都能用其页面和槽的信息来唯一地标识，称为记录标识符或 RID。

DB2 在索引定义中支持"包括列"(include column)。例如：

```
create unique index I1 on T1(C1) include (C2);
```

包括进去的索引列使得 DB2 在任何可能的时候可以扩展对"index-only"查询处理技术的使用。附加

指令如 **minpctused** 和 **pctfree** 可用于控制索引页的合并及初始空间分配。

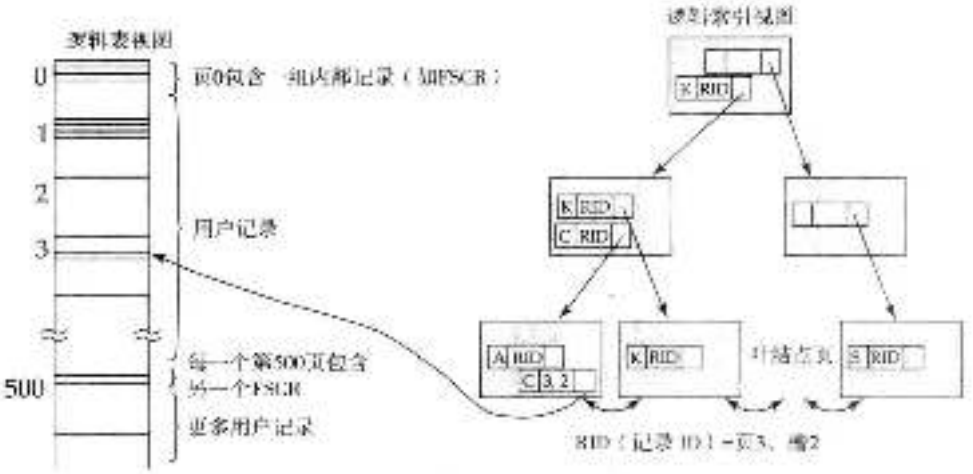


图 29-7 DB2 中的表和索引的逻辑视图

图 29-8 展示了 DB2 中典型的数据页格式。每个数据页包括一个页头和一个槽目录。槽目录是一个有 255 个项的数组，它指向页中记录的偏移量。图中展示了 473 号页在偏移量为 3800 处包含记录 0，偏移号为 3400 处包含记录 2，页 1056 在偏移号为 3700 处包含记录 1。这是一个指向记录 <473，2> 的重向指针。因此，记录 <473，2> 是条溢出记录，作为对原始记录 <1056，1> 的更新操作的结果而创建。DB2 支持不同的页规模，如 4KB、8KB、16KB 和 32KB。但是，每页中仅能包含 255 个用户记录。较大的页规模在诸如表中包含许多列的数据仓库那样的应用中是有用的。较小的页规模对具有频繁更新的操作型数据是有用的。

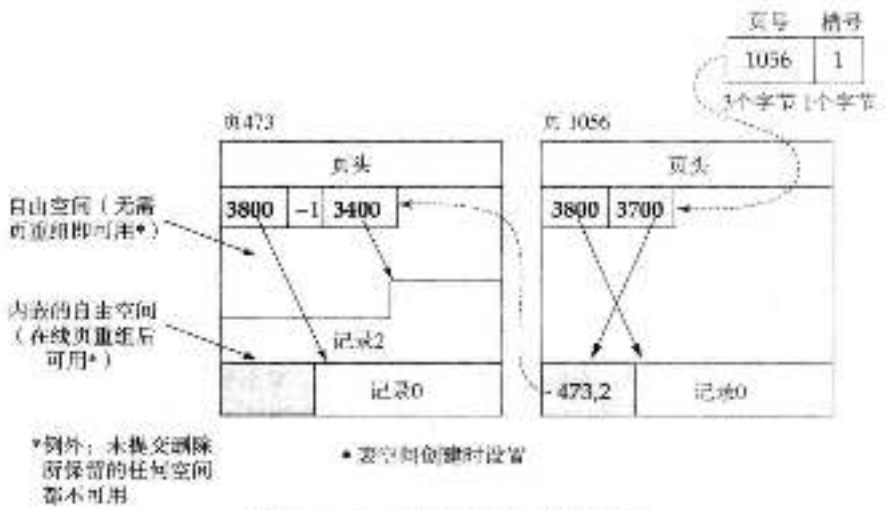


图 29-8 DB2 中数据页和记录的布局

29.5 多维聚簇

本节对 MDC 的主要特性提供简要的概述。有了这个特性，可以通过指定一个或多个列作为维来构建 DB2 的表，沿着这些维来聚簇表的数据。为此 DB2 包括了一个称作 **organize by dimensions** 的子句。例如，下面的 DDL 描述了一张销售表，通过 *storeId*、*year*（*orderDate*）和 *itemId* 属性作为维来组织。

1202
1203

```
create table sales( storeId int,
                    orderDate date,
                    shipDate date,
                    receiptDate date,
                    region int,
                    itemId int,
                    price float
                    yearOd int generated always as year(orderDate) )
organized by dimensions( region, yearOd, itemId );
```

每个这样的维都可以由一个或多个列组成，类似于索引码。事实上，“维块索引”（在后面描述）为每个指定的维自动生成并用于快速有效地访问数据。如果需要的话，会自动生成一个包含所有维码列的复合块索引，并用来在插入和更新活动中维护数据的聚簇。

维值的每个唯一组合构成了一个逻辑“单元”，它在物理上组织成页的块，其中块是磁盘上一组连续的页。包含在其中一个维块索引上具有特定码值的数据所在的那些页的块集合称为“切片”。表的每一页正好是某个块的一部分，并且表的所有块都包含相同数目的页，也就是块的大小。DB2 将块大小与表空间的盘区大小相关联，因此块边界可以联合盘区边界。

图 29-9 表明了这些概念。这个 MDC 表是沿着维 *year*(*orderDate*)[⊖]、*region* 和 *itemId* 来聚簇的。这个图表示了一个在每个维属性上只有两个值的简单逻辑立方体。事实上，维属性可以容易地扩展到大量的值而无需任何管理。图中子立方体代表逻辑单元。表中的记录存储在块中，块包含磁盘上相当于一个盘区的连续页。在图中，块用带阴影的椭圆来代表，并且根据在表中分配的盘区的逻辑顺序来编号。我们只显示了用维值 <197，Canada，2 > 标识的单元的几个数据块。格中的一列或一行代表某个特定维的切片。例如，在 *region* 维上包含值“Canada”的所有记录都位于立方体中由“Canada”列定义的切片所包含的块中。事实上，该切片的每个块只包含 *region* 字段为“Canada”的记录。 1204

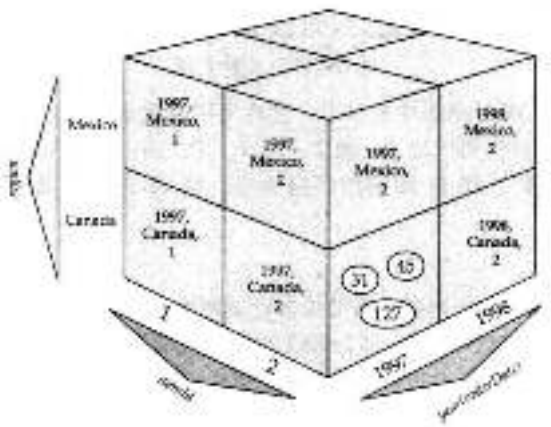
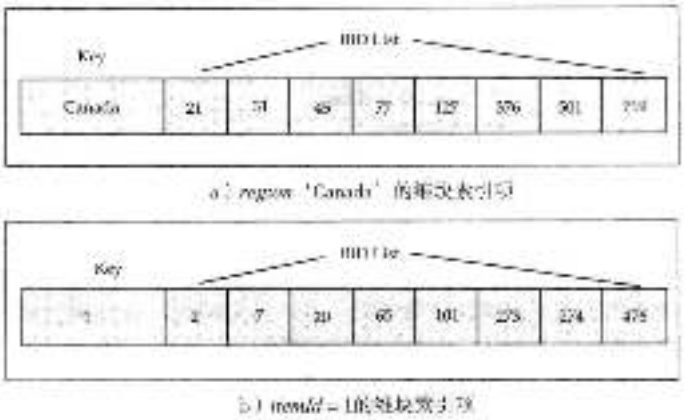


图 29-9 一个 MDC 表的物理分布的差集视图

29.5.1 块索引

在我们的例子中，维块索引创建在每个 *year*(*orderDate*)、*region* 和 *itemId* 属性之上。每个维块索引和传统 B 树索引的构造方式相同，不同的是在叶结点层次，码指向块标识符（BID）而不是记录标识符（RID）。因为每个块潜在地包含许多记录页，这些块索引比 RID 索引要小得多，而且只有当向一个单元中加入一个新块时，或者已存在块为空且从一个单元中移除时，才需要重新这些块索引。一个切片，或者包含页面的块集合（这些页面的所有记录具有一个维中的特定码值），在相关联的维块索引中由一个该码值的 BID 列表来表示。图 29-10 分别以 *region* 和 *itemId* 维的特定值为例说明了块的切片。

⊖ 通过使用一个生成函数来创建的。



a) region 'Canada' 的维块索引项

b) month = 10的维块索引项

图 29-10 块索引码项

在上面的例子中，要找到包含了在 region 维上取值为"Canada"的所有记录的切片，我们要在 region 维块索引中查找这个码值，找到如图 29-10a 所示的一个码。这个码正好指向对于特定值的 RID 集合。

29.5.2 块映射

块映射也是和表相关联的。块映射记录了属于表的每个块的状态。一个块可能处于多个状态中，例如使用中(in use)、空闲(free)、加载(loaded)、需要执行约束(requiring constraint enforcement)。数据管理层使用块的状态来确定多种处理选择。图 29-11 展示了表的块映射的例子。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
U	U	F	F	U	U	U	L	U	F	F	U	C	F	F	U	U	F	L	...

图 29-11 块映射项

块映射中的元素 0 表示 MDC 图表中的块 0。它的可用性状态为"U"，表明它正在使用中。然而，它是一个特殊块并且不包含任何用户记录。块 2、3、9、10、13、14 和 17 没有在表中使用。在块映射中被认为是"F"或者空闲。块 7 和 18 刚刚加载进表中。块 12 完整被加载并且需要在其上执行约束检查。

29.5.3 设计考虑

MDC 的一个至关重要的方面是选择合适的维集合来聚簇一个表，以及合适的块尺寸参数来使空间使用最小化。如果维和块尺寸选择得当，那么聚簇的好处表现为高性能和易于维护。另一方面，如果选择得不正确，性能会下降而且空间的使用可能会浪费。可以开发许多的调谐程序来组织表。这些包括变化维的数目、变化一个或多个维的粒度、变化块的尺寸(盘页尺寸)以及包含表的表空间的页面大小。一个或多个技术这样的技术可以联合使用以确定对表的最佳组织方式。

29.5.4 对现有技术的影响

人们很自然地会问新的 MDC 特性是否对普通表有不利影响或失去某些现有的 DB2 特性。所有现有的特性，例如二级 RID 索引、约束、触发器、定义物化视图以及查询处理选项，对 MDC 表都是可用的。因此，除了它们的增强的聚簇和处理能力，MDC 表就像普通表一样。

29.6 查询处理和优化

DB2 的查询编译器将查询转化为一棵操作树。查询操作树在执行时用于查询处理。DB2 支持一套丰富的查询操作，这使它可以考虑最佳的处理策略，并对执行复杂查询任务提供灵活性。

图 29-12 和图 29-13 展示了 DB2 中的一个查询和与之关联的查询计划。这是一个来自 TPC-H 基准测试的有代表性的复杂查询(查询 5)，它包含了几次连接和聚集。为这个特定例子选取的查询计划相当简单，因为没有为这些表定义索引和其他辅助结构，如物化视图。DB2 提供了各种"解释"工具，包

指控制中心的一个强大的可视化解释特性，能帮助用户理解查询执行计划的细节。图中给出的查询计划就是以该查询的可视化解释为基础的。可视化解释使得用户可以理解查询计划的不同操作的代价和其他相关性质。

```
-- TPCD Local Supplier Volume Query (Q5)
select n.name, sum(l.extendedprice*(1-l.discount)) as revenue
from tpcd.customer, tpcd.orders, tpcd.lineitem,
     tpcd.supplier, tpcd.nation, tpcd.region
where c.custkey = o.custkey and
      o.orderkey = l.orderkey and
      l.suppkey = s.suppkey and
      c.nationkey = s.nationkey and
      s.nationkey = n.nationkey and
      n.regionkey = r.regionkey and
      r.name = 'MIDDLE EAST' and
      o.orderdate >= date('1995-01-01') and
      o.orderdate < date('1995-01-01') + 1 year
group by n.name
order by revenue desc;
```

图 29-12　SQL 查询

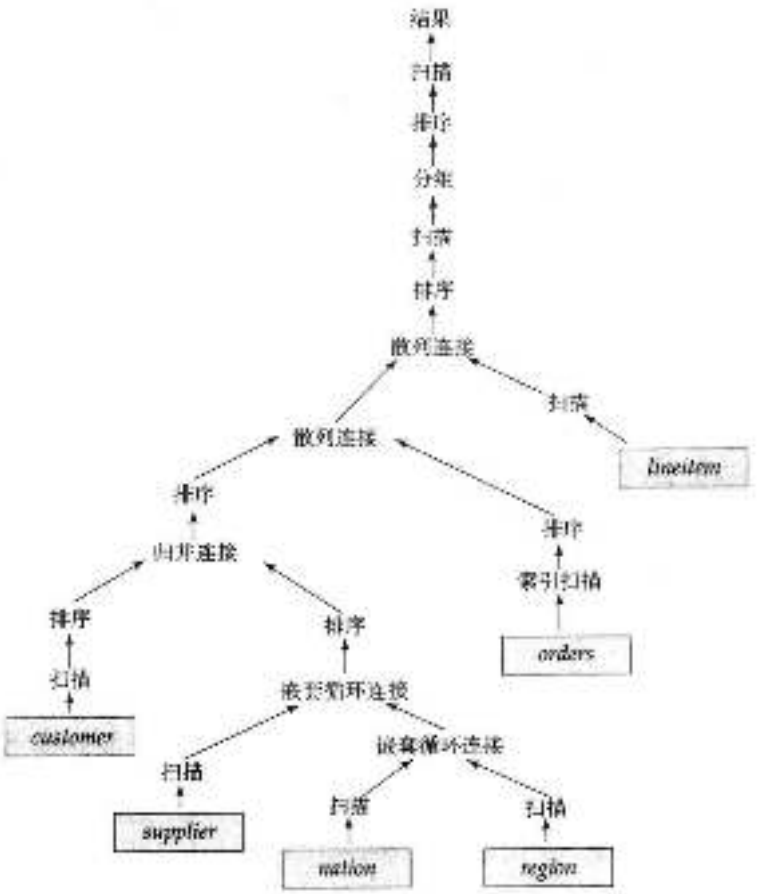


图 29-13　DB2 的查询计划(图形化解释)

所有 SQL 查询和语句都转化为查询树，无论它们有多么复杂。查询树的基本操作或者叶结点操作对数据库表中的记录进行操作，这些操作也称为访问方法。查询树的中间操作包括关系代数运算，如连接、集合运算和聚集。树的根结点产生查询或 SQL 语句的结果。

29.6.1 存取方法

DB2 支持关系表上的一个全面的存取方法集。存取方法的列表包括：

- **表扫描**(table scan)。这是最基本的方法，逐页地执行对表中所有记录的访问。
- **索引扫描**(index scan)。用索引来选出符合查询的特定记录。使用索引中的 RID 来访问符合条件的记录。当 DB2 发现了某种顺序访问模式时，就检测可能性来预取数据页。
- **块索引扫描**(block index scan)。这是用于 MDC 表的新的访问方法。使用其中一个块索引来扫描 MDC 数据块的一个特定集合。在块表扫描操作中访问和处理符合要求的块。
- **仅用索引**(index only)。在这种情况下，索引包含查询所需的所有属性。因此，扫描索引项就足够了。这种仅用索引的技术通常是一个性能较好的解决方案。
- **列表预取**(list prefetch)。当需要对有大量 RID 的非聚簇索引进行扫描时，这种访问方法是很好的选择。DB2 在 RID 上有一个排序操作，并按排好的次序从数据页中取记录。排序访问将 I/O 模式从随机变为顺序，并提供了预取机会。列表预取已被扩展到也能够处理块索引。
- **块和记录索引的"与"**(block and record index ANDing)。当 DB2 决定可用多于一个索引来限制基表中满足条件的记录的数目时，会采用这种办法。最有选择性的索引会被处理用来生成一个的 BID 或 RID 列表。然后处理具有次选择性的索引来返回满足条件的 BID 或 RID。只有出现在索引扫描结果的交集(AND 运算)中的 BID 或 RID 才有资格进行进一步处理。索引的 AND 运算的结果是满足条件的 BID 或 HID 的一个很小的列表，用来从基表中获取相应记录。
- **块和记录索引排序**(block and record index ordering)。如果能够用两个或多个块或记录索引来满足用 OR 运算符组合成的查询谓词，就可以使用这种策略。DB2 通过执行排序，然后获取记录的结果集来消除重复的 BID 或 RID。OR 索引已经扩展来考虑块和 HID 索引的结合情况。

查询的所有选择和投影谓词通常都下推到访问方法上。另外，为了减少指令路径，在"下推"模式中 DB2 会执行特定的操作，如排序和聚集。

MDC 特性利用为块索引扫描、块索引预取、块索引的"与"以及块索引的"或"而改进的新的存取方法集合来处理数据块。

29.6.2 连接、聚集和集合运算

DB2 支持许多用于这些运算的技术。对于连接，DB2 可以选择嵌套循环、排序合并和散列连接技术。为了描述连接和集合的二元运算，我们使用"外部"表和"内部"表的概念来区分两个输入流。当内部表非常小或者可以在连接谓词上使用一个索引来访问时，嵌套循环技术是有用的。排序归并连接和散列连接技术用于涉及大的外部表和内部表的连接。集合运算是用排序和归并技术来实现的。归并技术在并的情况下消除重复，而在交的情况下转发重复。DB2 还支持所有类型的外连接运算。

1205 / 1209

只要有可能，DB2 以早期的或"下推"的模式处理聚集运算。比如，分组聚集可以通过将聚集合并到分类阶段来执行。连接和聚集算法可以利用现代 CPU 中使用面向块的和高速缓存敏感的技术的超标量处理。

29.6.3 对复杂 SQL 处理的支持

DB2 最重要的一个方面是它以可扩展的方式使用查询处理基本结构来支持复杂 SQL 运算。这些复杂 SQL 技术包括对深层嵌套和关联查询以及约束、参照完整性和触发器的支持。因为大多数这样的动作被构建到查询计划中，DB2 能够进行调节并对更大数量的这类约束和动作提供支持。约束和完整性检查作为在插入、删除或更新的 SQL 语句上的查询树操作来建立。DB2 还支持用内置触发器来维护物化视图。

29.6.4 多处理器查询处理特性

为了支持 SMP(即共享内存)、MPP(即无共享)和 SMP(即共享磁盘的)集群模式的查询处理，DB2 用控制和数据交换原语扩展了查询操作的基本集合。DB2 将"表队列"抽象用于在不同结点或在相同结点上的线程之间的数据交换。表队列被当作缓冲区使用，它使用广播、一对一或定向多播方法将数据流定向到合适的接收端。控制操作用来创建线程并协调不同进程和线程的操作。

在所有这些模式中，DB2 利用一个协调进程来控制查询操作和收集最终结果。如果需要协调进程还可以执行一些全局数据库处理动作。合并局部聚集结果的全局聚集操作就是一个例子。子代理或从属线程在一个或多个结点上执行基本数据库操作。在 SMP 模式下，当共享数据时子代理使用共享主存在这些操作之间同步。在 MPP 模式下，在执行期间表队列机制为不同结点间的同步提供了缓冲区和流控制。DB2 使用广泛的技术来进行优化并高效处理在 MPP 或 SMP 环境下的查询。图 29-14 展示了一个在 4 结点 MPP 系统中执行的简单查询。在这个例子中，*sales* 表被划分在 4 个结点 P_1，…，P_4 上。这个查询通过生产代理(spawning agent)来执行，该代理在每个这样的结点上执行，以便扫描和过滤该结点上 sales 表中的行(叫做函数转移)，结果行发送到协调进程结点。 1210



图 29-14 使用函数转移的 DB2 MPP 查询处理

29.6.5 查询优化

为了进行转换和优化，DB2 的查询编译器使用查询的一种内部表示，称为查询图模型(QGM)。解析完 SQL 语句之后，DB2 在 QGM 上执行语义转换来实施约束、参照完整性和触发器。这些转化的结果是增强的 QGM。下一步，DB2 试图执行查询重写转换，这通常被认为是有益的。重写规则在适当的时候激活以执行所需转换。重写转换的例子包括(1)相关子查询的去除相关，(2)使用提前(early-out)处理将特定子查询转换为连接，(3)适当的时候将 **group by** 操作下推到连接以下，(4)对于原始查询的某些部分使用物化视图。

查询优化器部件使用这种改进和转换后的 QGM 作为其优化的输入。优化器是基于代价的，并使用一个可扩展的、规则驱动的框架。优化器可以配置成能在不同级别的复杂度上进行操作。在最高级，它使用动态规划算法来考虑所有查询计划的选项并挑选出代价最优的计划。在中间级，优化器不考虑特定的计划、访问方法(比如索引"或")或重写规则。在复杂度的最低级，优化器使用简单的贪心启发式方法来选择一个好的但不一定最佳的查询计划。优化器使用查询处理操作的细节模型，包括内存大小和预取，来得到对 I/O 和 CPU 开销的准确估计。它依赖数据统计来估计这些操作的基数和选择率。DB2 允许用户得到列级分布的细节直方图并用 **runstats** 工具将列进行组合。细节直方图包含了关于最频繁出现的值以及属性的基于分位数表示的频率分布的信息。优化器生成一个内部查询计划，被看作是对于特定优化层的最佳查询计划。这个查询计划被查询处理引擎转换成查询算子和相关数据结构的线程。 1211

29.7 物化的查询表

在 Linux、UNIX 和 Windows 中以及在 z/OS 平台上，DB2 支持物化视图。物化视图可以是任何定义在一个或多个表或者视图上的一般视图。物化视图是有用的，因为它维护视图数据的持久拷贝使得查询处理得更快。在 DB2 中物化视图称为**物化的查询表**(Materialized Query Table，MQT)。如图 29-15 中的例子所示，MQT 是通过使用 **create table** 语句来指定的。

在 DB2 中，MQT 可以引用其他 MQT 来建立依赖视图的树或者森林。这些 MQT 具有很高的可扩展性，它们可以在 MPP 环境中划分并且可以有 MDC 聚簇码。如果数据库引擎能够将查询无缝路由到 MQT，并且如果数据库引擎在任何可能的情况下能够高效地维护它们，MQT 是最有价值的。DB2 同时

提供了这两种特性。

```
create table emp_dept(dept_id integer, emp_id integer,
                      emp_name varchar(100), mgr_id integer) as
        select dept_id, emp_id, emp_name, mgr_id
        from employee, department
            data initially deferred
            refresh immediate -- (or deferred)
            maintained by user -- (or system)
```

图 29-15 DB2 物化的查询表

29.7.1 查询路由到 MQT

DB2 中的查询编译器的基本结构完美地适合于发挥 MQT 的全部功能。内部的 QCM 模型允许编译器对输入查询和可用 MQT 定义进行匹配，并选择适当的 MQT 进行考虑。匹配之后，编译器为了优化考虑几种选择，包括基本查询以及合适的 MQT 重路由版本。优化器在选出最优执行版本之前对这些选择进行循环遍历。重路由和优化的整个流程如图 29-16 所示。

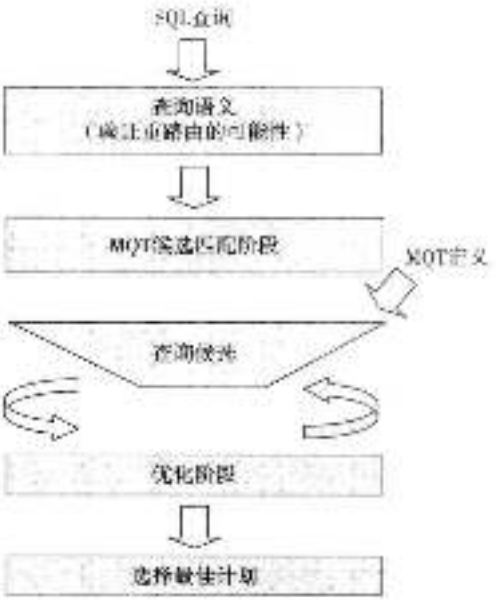


图 29-16 DB2 中的 MQT 匹配和优化

29.7.2 MQT 的维护

只有数据库引擎提供有效的维护技术时，MQT 才是有用的。有两方面需要维护：时间和开销。在时间方面有两种选择：立即和延迟。DB2 对这两种选择都支持。如果选择立即维护，系统就会创建内部触发器并编译到源对象的 **insert**、**update** 或 **delete** 语句中以处理对依赖的 MQT 的更新。在延迟维护的情况下，被更新的表转移到完整性模式，必须给出一条显式的 **refresh** 语句来执行维护。在开销方面的选择是：增量型和完全型。增量型维护是指只有最近更新的行才会用于维护。完全型维护是指整个 MQT 从它的源进行更新。图 29-17 中的矩阵显示了这两个方面以及沿着这些方面最有用的选择。例如，除非源非常小，否则立即维护和完全维护是不相容的。DB2 还允许由用户来维护 MQT。在这种情况下，MQT 的更新通过用户使用 SQL 或工具执行显式的处理来确定。

选择	增量型	完全型
立即	是，插入/更新/删除后	通常不
延迟	是加载后	是

图 29-17 DB2 中的 MQT 维护选项

下面的命令提供了一个简单的例子，在对物化视图 *emp_dept* 的一个源执行加载操作后，对其执行延迟维护。

```
load from newdata.txt of type del
insert into employee;

refresh table emp_dept
```

29.8 DB2 中的自治特性

DB2 UDB 为简化数据库的设计和管理提供了许多特性。自治计算包括一系列技术，这些技术允许

计算环境自治管理，且在面对安全性，系统负荷或其他因素方面发生外部和内部变化时减少外部依赖。增强自治计算有益于许多主题领域，例如配置、优化、保护以及监控。下面几节简要描述了配置和优化领域。

29.8.1 配置

DB2 提供对自动调整各种内存和系统配置参数的支持。例如，像缓冲池大小和排序堆大小这样的参数可以自动指定。在这种情况下，DB2 监控系统并根据工作负载特征缓慢地增加和缩减这些堆存储空间的大小。

29.8.2 优化

辅助数据结构(索引、MQT)和数据组织特性(划分、聚簇)是提高 DB2 中数据库处理性能的重要方法。在过去，数据库管理员(DBA)必须使用经验和公认的指导方针来选择有意义的索引、MQT、分区码以及聚簇码。给定潜在的选择数目，即使最好的专家也不可能在短时间内为给定工作负载找到这些特性的正确组合。DB2 包含了一个设计向导，对所有这些特性提供基于工作负载的建议。设计向导顾问工具使用优化技术自动分析工作负载以给出一组建议。设计向导的命令语法是：

```
db2advis  -d <DB name>  -i <workloadfile>  -m MICP
```

参数"-m"允许用户指定如下选项：

- M——物化的查询表。
- I——索引。
- C——聚簇，即 MDC。
- P——划分码选择。

这个向导在这些建议中充分利用 DB2 查询优化框架的能力。它使用输入的工作负载以及对建议的规模和时间上的约束作为其参数。倘若它利用 DB2 的优化框架，它对底层数据的模式和统计信息就会有个面的了解。该向导使用若干组合技术来识别索引、MQT、MDC 以及划分码以提高给定工作负载的性能。 1212 1214

优化的另一方面是均衡系统的处理负载。尤其是某些工具会增加系统负载并导致用户工作负载性能的极大降低。倘若在线工具是主要发展趋势，就需要对工具的负载消耗进行均衡。DB2 包含一个工具负载抑制机制。这种抑制技术基于反馈控制理论。它利用特定控制参数不断地调整和抑制备份工具的性能。

29.9 工具和实用程序

DB2 为易于使用和管理提供了许多工具。这些核心工具集由于来自供应商的大量工具扩大和增强了。

DB2 控制中心(DB2 Control Center)是使用和管理 DB2 数据库的主要工具。控制中心在许多工作站平台上运行。它由诸如服务器、数据库、表和索引这样的数据对象组织而成。它包含面向任务的接口来执行命令并允许用户生成 SQL 脚本。图 29-18 显示了控制中心主面板的屏幕截图。这个屏幕截图展示了结点 *Cronkarm* 上 DB2 实例中的 *Sample* 数据库的一系列表。管理员可以使用菜单来调用一套部件工具。控制中心的主要部件包括命令中心、脚本中心、日志、许可证管理、报警中心、性能监控器、可视化解释、远程数据库管理、存储管理和复制支持。命令中心允许用户和管理员发出数据库命令和 SQL。脚本中心允许用户运行交互式构成的或来自文件的 SQL 脚本。性能监控器允许用户监视数据库系统中的各种事件并获得性能快照。"敏捷指南"(SmartGuides)提供配置参数和安装 DB2 系统的帮助。存储过程构建器帮助用户开发并安装存储过程。可视化解释允许用户获得查询执行计划的图形化视图。索引向导为管理员提供关于性能因素的索引建议。

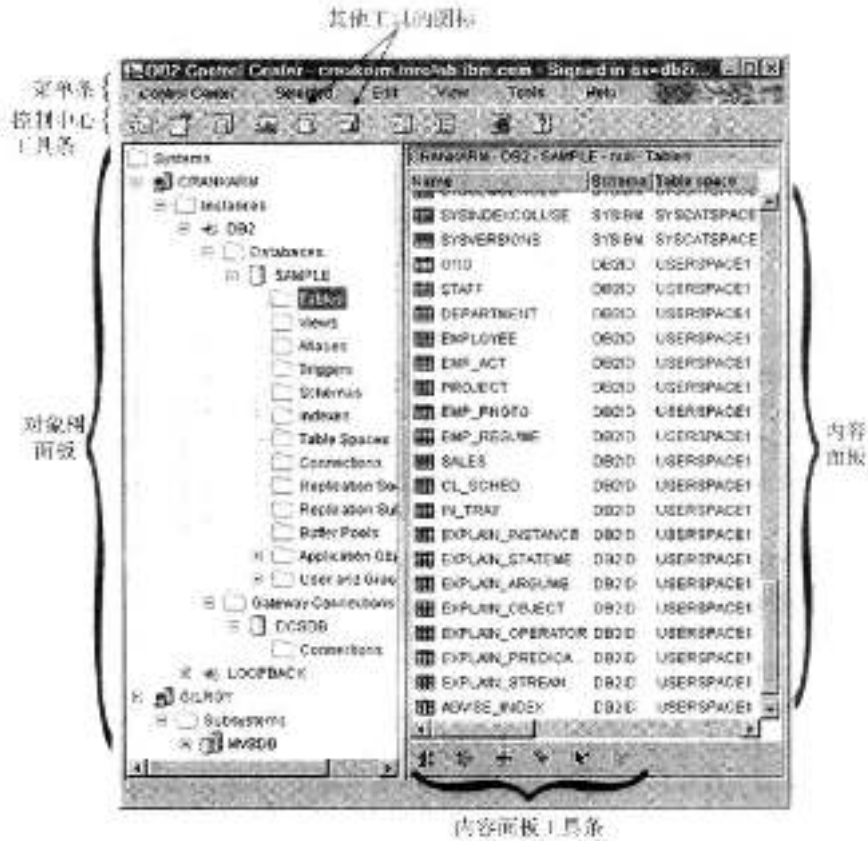


图 29-18 DB2 控制中心

虽然控制中心是许多任务的综合界面，DB2 也提供对大多数工具的直接访问。诸如解释工具、解释表和图形化解释那样的工具为用户提供了查询计划的详细分解。用户还允许修改统计数据（如果允许的话）来产生最佳查询计划。

有许多工具提供给管理员。DB2 为加载、导入、导出、重组、重分布和其他数据相关的实用程序提供全面支持。它们大部分都支持增量和在线的处理能力。例如，人们可以在在线模式下发布一条加载命令，以允许应用程序并发地访问表的原始内容。DB2 的实用程序完全能够运行在并行模式下。

此外，DB2 支持许多工具，例如：

- 用于维护数据库活动的审计追踪的审计设施。
- 用来控制不同应用优先级和执行时间的主管设施。
- 用来管理系统中查询作业的查询巡视器设施。
- 用于调试的跟踪和诊断设施。
- 用于在系统执行时跟踪资源和事件的事件监视设施。

1215 面向 OS/390 的 DB2 有着一个非常丰富的工具集。QMF 是一款广泛使用的工具，用于生成即席查
1216 询并将其整合到应用中。

29.10 并发控制和恢复

DB2 支持全面的并发控制、隔离和恢复技术。

29.10.1 并发与隔离

对于隔离，DB2 支持可重复读（RR）、读稳定性（RS）、游标稳定性（CS）和未提交读（UR）模式。RR、CS 以及 UR 模式不需要进一步解释。RS 隔离模式只封锁应用程序在一个工作单元中检索的那些

行。在随后的扫描中，保证应用程序看到所有这些行(如同 RR)，但可能还会看见新的满足条件的行。然而，对于一些遵循严格的 RR 隔离性的应用程序来说，这可能是种可接受的权衡。典型地，默认的隔离性级别是 CS，应用程序在绑定阶段可以选择它们的隔离性级别。大多数商业化可用的应用程序允许使用大多数隔离性级别，使用户可以针对他们的需求选择正确的应用程序版本。

各种隔离模式是用封锁来实现的。DB2 支持记录级和表级的封锁。它维护一个单独的具有封锁信息的锁表数据结构。如果锁表中的空间变得紧张，DB2 就将记录级锁升级为表级锁。DB2 为所有更新事务实现严格的两阶段封锁。事务持有写锁或更新锁直到提交或回滚时为止。图 29-19 展示了不同封锁模式和对它们的描述。这些封锁模式中包括一种支持最大化并发性的表级意向锁。而且，为了消除 Halloween 和幻象族问题，DB2 为影响索引扫描的更新实现了下一个关键字封锁和不同模式。

封锁模式	对象	解释
IN（无意向）	表空间，表	不带行锁的读
IS（意向共享）	表空间，表	带行锁的读
NS（下一关键字共享）	行	RS或CS隔离性级别的读锁
S（共享）	行，表	读锁
IX（意向排他）	表空间，表	意向更新行
SIX（共享意向排他）	表	行上无读锁，有更新行上有X锁
U（更新）	行，表	更新锁但允许他人读
NX（下一关键字排他）	行	在RR索引扫描中，为防止幻象读对插入/删除的下一关键字封锁
X（排他）	行，表	只允许未提交的读者访问
Z（超级排他）	表空间，表	完全的排他访问

图 29-19　DB2 封锁模式

通过使用 **lock table** 语句，事务可以将封锁粒度设置在表级。当应用程序知道所需要的隔离性级别是表级时，这是很有用的。而且，DB2 会为诸如重组和加载这样的工具选择合适的封锁粒度。这些工具的脱机版本通常以排他模式封锁表。这些工具的联机版本允许其他事务通过获得行锁来并发运行。

每个数据库上都运行一个死锁检测代理并周期性地检测事务间的死锁。死锁检测的时间间隔参数是可配置的。在发生死锁的情况下，代理就选择一个牺牲者并用一个 SQL 死锁错误代码来中止它。

29.10.2　提交与回滚

应用程序可以通过使用显式的 **commit** 或 **rollback** 语句来提交或回滚。应用程序还可以发出 **begin transaction** 和 **end transaction** 语句来控制事务的范围，但不支持嵌套事务。通常，当一个事务提交或回滚时，DB2 代表它释放这个事务所持有的所有锁。然而，如果用 **with hold** 子句声明了一条游标语句，那么提交后还会有一些锁保留下来。

29.10.3　日志与恢复

DB2 实现严格的 ARIES 日志和恢复方案。在写数据页之前或在事务提交时，使用先写(write-ahead)日志将日志记录刷新到持久日志文件中。DB2 支持两种类型的日志模式：循环日志和归档日志。在循环日志中，使用一套预先定义的主日志和辅助日志文件。循环日志在崩溃恢复或应用程序故障恢复中是有用的。在归档日志中，DB2 创建新的日志文件而旧的日志文件必须归档以释放文件系统的空间。归档日志用于执行前滚恢复。在这两种情况下，DB2 允许用户来配置日志文件的数量和日志文件的大小。

在频繁更新的环境中，可以对 DB2 进行配置，以寻求成组提交，为的是把多个日志写操作捆绑在一起。

DB2 支持事务回滚和崩溃恢复以及时间点或前滚恢复。在崩溃恢复的情况下，DB2 执行标准的

undo 处理阶段查到最后一个检查点并从该检查点起执行标准的 *redo* 处理阶段，以恢复到适当的数据库提交状态。对于时间点恢复，可以从备份中恢复数据库，并利用归档日志将它前滚到一个特定的时间点。前滚恢复命令既支持数据库级也支持表空间级。它还可以在多结点系统中的特定结点上发布。一个并行恢复方案通过使用许多 CPU 改进了 SMP 系统中的性能。DB2 通过实现全局检查点方案来执行跨越 MPP 结点的协同恢复。

1217
1218

29.11 系统体系结构

图 29-20 展示了 DB2 服务器中的一些不同的进程或线程。远程客户应用程序通过使用诸如 *db2tcpcm* 这样的通信代理来与数据库服务器连接。每个应用程序都分配一个称为 *db2agent* 线程的代理（在 MPP 或 SMP 环境中的协调代理）。这个代理以及它的下级代理执行与应用程序相关的任务。每个数据库有一组进程和线程执行如预取、从缓冲池中清空页、日志以及死锁检测这样的任务。最后，还有一组在服务器级的代理来执行如监测器、许可证服务器、并程创建和系统资源控制这样的任务。DB2 提供配置参数来控制在一个服务器中线程和进程的数目。几乎所有不同类型的代理都可以通过使用配置参数来控制。

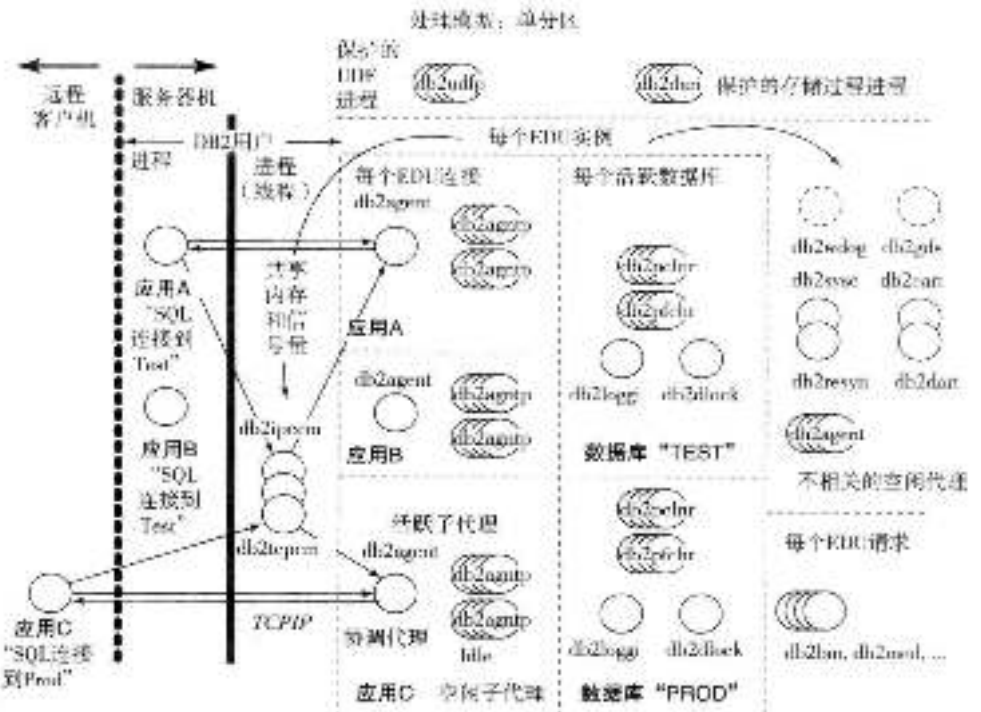


图 29-20 DB2 中的进程模型

图 29-21 展示了 DB2 中不同类型的内存段。代理或线程中的私有内存主要用于只与当前活动有关的局部变量和数据结构。例如，私有排序可以从代理的私有堆中分配内存。共享内存划分成服务器共享内存、数据库共享内存和应用程序共享内存。数据库级共享内存包含有用的数据结构，如缓冲池、封锁列表、应用程序包高速缓存和共享排序区。服务器和应用程序共享内存区域主要用于通用数据结构和通信缓冲区。

DB2 支持一个数据库有多个缓冲池。缓冲池可以用 **create bufferpool** 语句来创建，并且可以与表空间关联。出于多种原因，多个缓冲池是有用的，但应该对工作负载需求仔细分析后再定义多个缓冲池。DB2 支持全面的内存配置和调整参数的列表。这些包括所有大数据结构的堆区域（如默认的缓冲池、排序堆、程序包高速缓存、应用程序控制的堆和封锁列表区域）的参数。



图 29-21　DB2 内存结构

29.12　复制、分布和外部数据

DB2 Replication 是 DB2 家族的一个产品，它提供了不同于 DB2 的关系数据源（如 Oracle、Microsoft SQL Server、Sybase Adaptive Server Enterprise 和 Informix）以及非关系数据源（如 IBM 的 IMS）之间的复制能力。它由 *capture* 和 *apply* 部件组成，它们由管理界面来控制。变更捕获机制可以是"基于日志"的（针对 DB2 表）或"基于触发器的"（在其他数据源的情况下）。捕获到的变更保存在 DB2 Replication 控制下的临时阶段性表区域中。然后用普通 SQL 语句 insert、update 和 delete 来将这些有改动的分阶段中间表应用到目标表中。通过使用过滤条件和聚集，可以在中间分阶段的表上执行基于 SQL 的转换。结果行可以应用到一个或多个目标表。所有这些动作都由管理工具控制。 1219 1220

DB2 支持一种叫做队列复制的特性。队列（Q）复制使用 IBM 的消息队列产品建立一个队列传送机制来将捕获到的日志记录作为消息传递。这些消息在接收端从队列中提取出来并应用到目标上。应用过程可以是并行的，并允许用户指定冲突解决规则。

DB2 家族的另一个成员是 DB2 信息整合产品，它提供联邦、复制（使用上面描述的复制引擎）和搜索能力。联邦版本可以将远程 DB2 或其他关系数据库中的表整合成单个分布式数据库。用户和开发人员能够使用包装器技术以表格的形式访问各种非关系数据源。联邦引擎为跨越不同数据站点的查询优化提供一种基于代价的方法。

DB2 支持用户自定义的表函数，可以访问非关系的和外部的数据源。通过使用带 **returns table** 子句的 **create function** 语句来创建用户自定义表函数。使用这些特性，DB2 就可以参与到 OLE DB 协议中。

最后，DB2 提供对采用两阶段提交协议的分布式事务处理的完全支持。DB2 可以作为协调者或代理来支持分布式的 XA。作为协调者，DB2 可以执行两阶段提交协议的所有阶段。作为参与者，DB2 可以与任何商用分布式事务管理者交互。

29.13　商务智能特性

DB2 数据仓库版本是 DB2 家族中的贡献，它结合了商务智能特性。数据仓库版本以 DB2 引擎为基础，并用 ETL、OLAP、挖掘和在线登录特性来对其增强。DB2 引擎利用其 MPP 特性提供了可扩展性。在 MPP 模式中，DB2 所支持的配置能为大型数据库（TB）扩展到数百个结点。此外，像 MDC 和 MQT 这

样的特性为商务智能的复杂查询处理需求提供了支持。

商务智能的另一方面是联机分析处理或 OLAP。DB2 家族包括一个叫做立方体视图的特性，它提供了一种在 DB2 内部构造适当的数据结构和 MQT 的机制，可用于关系型 OLAP 处理。立方体视图为多维立方体提供了建模支持，并提供到关系型星型模式的映射机制。然后这个模型用于推荐适当的 MQT、索引和 MDC 定义来提高数据库上 OLAP 查询的性能。此外，立方体视图可以利用 DB2 为产生聚集立方
[1221] 体的 **cube by** 和 **rollup** 操作的原生支持。立方体视图是一个可用来将 DB2 与诸如 Business Objects、Microstrategy 和 Cognos 这样的 OLAP 供应商紧密结合在一起的工具。

此外，DB2 使用 DB2 OLAP 服务器对多维 OLAP 提供支持。DB2 OLAP 服务器可以通过 OLAP 技术从低层的 DB2 数据库建立起多维数据集市用于分析。来自 Essbase 产品的 OLAP 引擎被用在 DB2 OLAP 服务器中。

DB2 Alphablox 是一个提供联机的、交互的，报表和分析能力的新特性。Alphablox 特性的一个非常有吸引力的特点是，用一个称为 *blox* 的构建块方法来迅速地建立新的基于 Web 的分析表格。

为了深入分析，DB2 智能挖掘器为建模、打分和数据可视化提供了多个部件。挖掘使用户能够在大数据集上执行分类、预测、聚类、分段和关联。

文献注解

DB2 的起源可以追溯到 System R 项目(Chamberlin 等[1981])。IBM 的研究贡献包括许多领域，如事务处理(先写日志和 ARIES 恢复算法)(Mohan 等[1992])、查询处理和优化(Starburst)(Haas 等[1990])、并行处理(DB2 并行版本)(Baru 等[1995])、主动数据库支持(约束和触发器)(Cochrane 等[1996])、像物化视图这样的高级查询和仓库技术(Zaharioudakis 等[2000]，Lehner 等[2000])、多维聚簇(Padmanabhan 等[2003]、Bhattacharjee 等[2003])、自治特性(Zilio 等[2004])和对象－关系支持(ADT、UDF)(Carey 等[1999])。多处理器并行处理细节可以在 Baru 等[1995]中找到。Don Chamberlin 的书提供了对 DB2 早期版本的 SQL 和编程特性的很好回顾(Chamberlin[1996]，Chamberlin[1998])。C. J Date 和其他人的早期书籍提供了对 DB2 Universal Database for OS/390 的特性的很好回顾(Date[1989]，Martin 等[1989])。

DB2 用户手册提供了 DB2 各种版本的权威见解。大多数手册可以在线获得(http://www.software.ibm.com/db2)。用于开发者和管理员的 DB2 的书籍包括 Gunning[2008]、Zikopoulos 等
[1222] [2004]、Zikopoulos 等[2007]和 Zikopoulos 等[2009]。

Microsoft SQL Server

Sameet Agarwal, José A. Blakeley, Thierry D'Hers, Kctan Dawodi, César A. Galindo-Legaria, Gerald Hinson, Dirk Myers, Yaqar Pirzada, Bill Ramos, Balaji Rathakrishnan, Jack Richins, Michael Rys, Florian Waas, Michael Zwilling

Microsoft SQL Server 是一个关系型数据库管理系统，其范围涵盖膝上型计算机和台式机直至企业服务器。SQL Server 有一个兼容版本，基于 Windows Mobile 操作系统，用于手持设备，如袖珍 PC、智能手机和便携式媒体中心。SQL Server 最初是在 20 世纪 80 年代由 Sybase 为 UNIX 系统开发的，后来被微软移植到了 Windows NT 系统。从 1994 年开始，微软发布独立于 Sybase 开发的 SQL Server 版本，而 Sybase 在 20 世纪 90 年代后期停止使用 SQL Server 名称。最新的版本是 SQL Server 2008，有简易版、标准版和企业版，还有世界上多种语言的本地化版本。在本章，术语 SQL Server 是指 SQL Server 2008 和所有这些版本。

SQL Server 提供 SQL Server 多个拷贝之间以及与其他数据库系统的复制服务。它的分析服务，即系统的一个完整部分，包括联机分析处理(OLAP)和数据挖掘工具。SQL Server 提供了一个大的图形化工具集和向导，引导数据库管理员执行各种任务，例如建立定期备份、在服务器之间复制数据，以及调整数据库性能。有很多开发环境支持 SQL Server，包括微软的 Visual Studio 和相关产品，尤其是 .NET 的产品和服务。

30.1 管理、设计和查询工具

SQL Server 提供了一套工具，用来管理 SQL Server 的开发、查询、调优、测试和管理等各个方面。大部分这些工具是以 SQL Server Management Studio 为中心的。SQL Server Management Studio 为管理所有与 SQL Server 相关的服务提供通用的界面，包括数据库引擎、分析服务、报表服务、SQL ServerMobile 和集成服务。

30.1.1 数据库开发和可视化数据库工具

在设计数据库时，数据库管理员创建数据库对象，如表、列、关键字、索引、联系、约束和视图。为了帮助创建这些对象，SQL Server Management Studio 提供对可视化数据库工具的访问。这些工具提供了一种机制来帮助数据库设计：数据库设计器、表设计器和视图设计器。

数据库设计器是一个允许数据库所有者或所有者的代理来创建表、列、关键字、索引、联系和约束的可视化工具。在这个工具中，用户可以通过数据库图表和数据库对象进行交互。数据库图表图形化地显示了数据库的结构。视图设计器提供了可视化查询工具，允许用户通过使用 Windows 的拖放功能创建或修改 SQL 视图。图 30-1 显示了一个从 Management Studio 中打开的视图。

30.1.2 数据库查询和调优工具

SQL Server Management Studio 提供了若干工具来帮助应用开发过程。可以使用集成的查询编辑器来进行查询和存储过程的开发和测试。查询编辑器支持为各种环境创建和编辑脚本，包括 Transact-SQL、SQL Server 脚本语言 SQLCMD，用于数据分析的多维表示语言 MDX、SQL Server 数据挖掘语言 DMX、XML 分析语言 XMLA 和 SQL Server Mobile。进一步的分析可以通过使用 SQL Server 跟踪器(profiler)来进行。数据库调优建议由数据库调优向导提供。

30.1.2.1 查询编辑器

集成的查询编辑器提供了一个简单的图形化用户界面来运行 SQL 查询和显示结果。查询编辑器还

提供显示计划（showplan）（优化器为查询执行所选择的步骤）的图形化表示。查询编辑器集成了 Management Studio 的对象浏览器，允许用户拖放对象或表名到一个查询窗口，并帮助在任何表上建立 **select**、**insert**、**update** 或 **delete** 语句。

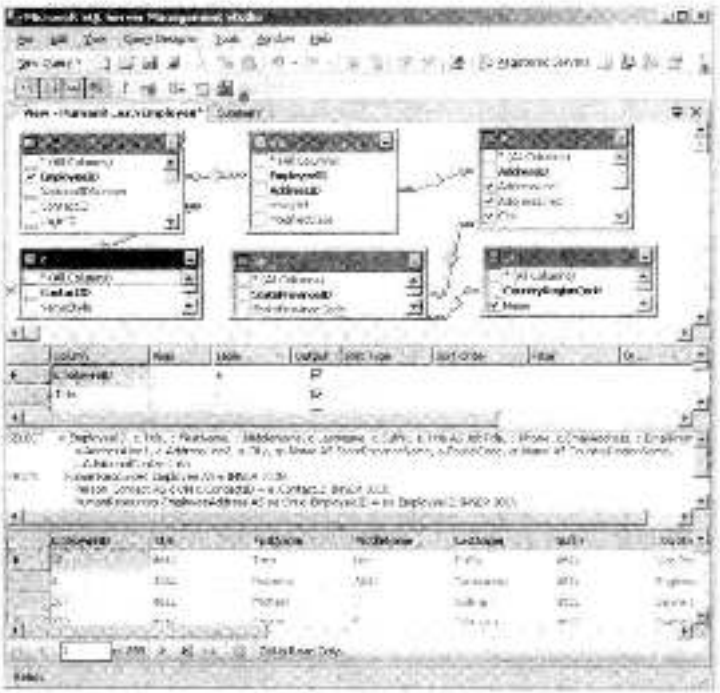


图 30-1 为 HumanResources.vEmployee 视图打开的视图设计器

数据库管理员或开发者能够用查询编辑器做以下工作：

- **分析查询**：查询编辑器能够为任何查询显示图形化的或者文本的执行计划，并且显示关于执行任何查询所需要的时间和资源的统计信息。
- **格式化 SQL 查询**：包括缩排和颜色语法编码。
- **为存储过程、函数以及基本 SQL 语句使用模板**：Management Studio 有许多预定义模板来建立 DDL 命令，或者用户也可以定义他们自己的模板。

图 30-2 显示了带查询编辑器的 Management Studio，展示了一个查询的图形化执行计划，它包括一个四表连接和一个聚集。

30.1.2.2 SQL 跟踪器（SQL Profiler）

SQL 跟踪器是一个图形化工具，它允许数据库管理员监视和记录 SQL Server 数据库引擎和分析服务的数据库活动。SQL 跟踪器能够实时显示所有服务器的活动，或者它可以建立过滤器，以集中监视特定的用户、应用程序或者特定类型命令的活动。SQL 跟踪器能够显示发送到任何 SQL 服务器实例上的任何 SQL 语句或存储过程（如果安全权限许可的话）。还能显示性能数据，表明运行查询所花的时间、需要多少 CPU 和 I/O 以及查询所采用的执行计划。

1224
1225

SQL 跟踪器允许下钻甚至深入到 SQL Server 内部以监视作为存储过程的部分而执行的每条语句、每个数据修改操作、每个封锁的获得或释放或数据库文件自动增长的每次发生。它能够捕获多个不同的事件以及每个事件的多个数据项。实际上 SQL Server 把跟踪功能分成两个独立的但相互联系的部分。SQL 跟踪器是客户端的跟踪工具。使用 SQL 跟踪器，被捕获的数据不仅可以在跟踪器的用户界面（UI）中显示，用户还可以选择将它们保存在文件或表中。跟踪器工具在事件发生时显示每个符合过滤标准的事件。一旦保存跟踪数据，SQL 跟踪器就能够读取保存的数据，用于显示或分析的目的。

在服务器端的是 SQL 跟踪工具，它管理由事件产生器生成的事件队列。一个消费者线程从队列中

读取事件并进行过滤，之后把它们送给请求这些事件的进程。追踪所关注的活动的主要单元是事件，事件可以是发生在 SQL Server 内部或者 SQL Server 和客户端之间的任何事情。比如，创建或删除一个对象是一个事件，执行一个存储过程是一个事件，获得或者释放一个锁是一个事件，从客户端发送一个 Transact-SQL 包到 SQL Server 也是一个事件。有一组存储的系统过程来定义哪些事件需要追踪，每个事件需要关注哪些数据，以及在何处保存从事件中收集到的信息。在事件上应用过滤器能够减少收集和存放的信息量。



图 30-2 带 **group by** 聚集的四表连接的显示计划

SQL Server 保证特定的关键信息总能收集到，并且可用作一种有用的审核机制。SQL Server 达到美国政府 C2 级安全认证，并且很多可跟踪事件仅仅用于满足 C2 级认证的要求。

30.1.2.3 数据库调优向导

如果有一组合适的索引可用，查询和更新通常可以执行得更快。在大型数据库中，为表设计可能是最好的索引是一项复杂的工作。它不仅需要完全了解 SQL Server 如何使用索引以及查询优化器如何做决策，还得了解应用程序或交互式查询实际上如何使用数据。SQL Server 数据库调优向导(DTA)是一个强大的工具。它基于对查询和更新负载的观察，来设计可能是最好的索引和带索引的(物化)视图。

DTA 能跨多个数据库进行调优，而且它的建议是基于工作负荷的，工作负荷可以是一个捕获的追踪事件的文件、一个 SQL 语句文件或是一个 XML 输入文件。可用 SQL 跟踪器来捕获一段时间内所有用户提交的所有 SQL 语句。然后 DTA 能够查看所有用户、所有应用程序、所有表的数据访问模式，并给出均衡的意见。

30.1.3 SQL Server Management Studio

除了提供对数据库设计和可视化数据库工具的访问之外，易用的 SQL Server Management Studio 支持集中管理多个 SQL Server 数据库引擎装置、分析服务、报表服务、集成服务和 SQL Server Mobile 的各个方面，包括安全、事件、警报、调度、备份、服务器配置、调优、全文搜索以及复制。SQL Server Management Studio 允许数据库管理员创建、修改和复制 SQL Server 数据库模式和对象，如表、视图和触发器。因为多个 SQL Server 装置可组织成组，作为一个单元对待，SQL Server Management Studio 能够同时管理几百个服务器。

尽管 SQL Server Management Studio 能够和 SQL Server 引擎运行在同一台计算机上，当运行在任何

Windows 2000(或更新的版本)机器上时，SQL Server Management Studio 也提供了同样的管理能力。此外，SQL Server 高效的客户－服务器体系结构使得利用 Windows 的远程访问(拨号网络)能力来进行监督和管理成为现实。

SQL Server Management Studio 把数据库管理员从必须了解完成一项任务的特定步骤和语法中解放出来。它提供向导来指导数据库管理员完成一个 SQL Server 安装的安装和维护过程。Management Studio 的界面如图 30-3 所示，它说明了如何从会话中直接创建为数据库备份的脚本。

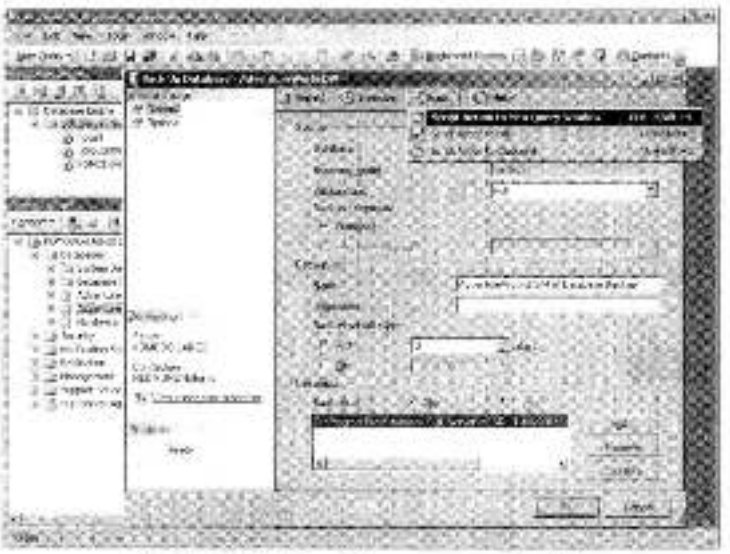


图 30.3 SQL Server Management Studio 界面

30.2 SQL 变化和扩展

SQL Server 允许应用程序开发者使用 Transact-SQL 或者 .NET 编程语言(例如 C#、Visual Basic、COBOL 或 J++)来编写服务器端的商务逻辑。Transact-SQL 是一种完全的数据库编程语言，包括数据定义和数据操纵语句、循环和条件语句、变量、过程以及函数。Transact-SQL 支持 SQL：2003 标准中
[1228] 大多数强制的 DDL 查询和数据修改语句以及结构。所支持的 SQL：2003 数据类型列表参见 30.2.1 节。除强制的特性外，Transact-SQL 也支持许多 SQL：2003 标准中可选的特性，例如通过查询、通用表的表达式、用户自定义函数和关系算子，如 **intersect** 和 **except**，还有其他的。

30.2.1 数据类型

SQL Server 2008 支持 SQL：2003 标准中所有强制的标量数据类型。SQL Server 还支持使用用户提供的名字定义系统类型别名的能力。别名和 SQL：2003 中的 distinct 类型功能相似，但并不与之完全兼容。

一些只有 SQL Server 才有的原始数据类型包括：

- 可变长度的大字符和二进制串类型，最大长度为 $2^{31}-1$ 字节，使用 **varchar/nvarchar/varbinary (max)** 数据类型，它们拥有一个类似于小字符和字节串类型的缺省模型。另外，它们支持一个称为 FILESTREAM 的存储属性，指定了每个单独列取值的数据存储为文件系统中单独的文件。FILESTREAM 存储允许使用本地文件系统 API 来更高性能地流式访问应用。
- XML 类型将在 30.11 节介绍，用于在表的列中存储 XML 数据。XML 类型可以选择用一个相关的 XML 模式集合来指定一个约束，使得类型的实例应该遵守模式集合中定义的一种 XML 类型。
- **sql_variant** 是一种标量数据类型，它可以包含任何 SQL 标量类型的值(除了大字符和二进制类型以及 **sql_variant**)。如果一个应用需要存储数据，但是该数据的类型在数据定义时不能预知，则要用到该类型。执行 **unpivot** 关系运算(见 30.2.2 节)所形成的列的类型也是 **sql_variant**。在

内部，系统保持对数据初始类型跟踪。在 **sql_varient** 列上可能做过滤，连接和排序。系统函数 **sql_varient_property** 能够返回存储在 **sql_varient** 类型的列中的实际数据的细节信息，包括基本类型和规模信息。

- **hierarchyId** 数据类型使得存储和查询层次数据更加容易。层次数据定义为一个相互之间通过层次关系关联的数据项的集合。其中数据的一个项都对一个项的父亲。通常的例子包括：组织结构，层次文件系统，项目的任务集合，语言术语的分类学，单继承类型的层次，部分－子部分联系和 Web 页面之间的链接图。
- SQL Server 支持存储和查询地球空间数据，即涉及地球的位置数据。这些数据的通用模型是平面和大地坐标系统。这两种系统主要的不同是后者考虑了陆地的曲率。SQL Server 支持几何学和地理学，对应于平面的和大地的模型。 1229

此外，SQL Server 支持 table 和 cursor 类型，它们不能用于表中的列，但可用作 Transact-SQL 语言里的变量：

- **表**(table)类型使得一个变量能够容纳行的集合。这种类型的实例主要用于存放存储过程的临时结果或以表为值的函数的返回值。table 变量和局部变量的行为一样。它有确定定义的作用范围，比如函数、存储过程或者是声明该变量的批处理中。在其作用范围内，table 变量能像普通的表一样使用。它能用于在 **select**、**insert**、**update** 和 **delete** 语句中的任何一个使用表或者表的表达式的地方。
- **游标**(cursor)类型使得能够引用游标对象。游标类型可用于声明变量，或者跨例程调用、引用游标的例程输入/输出变元。

30.2.2 查询语言增强

除了 SQL 关系运算，例如内连接和外连接，SQL Server 还支持关系运算 **pivot**、**unpivot** 和 **apply**。

- **pivot** 运算转换它的输入结果集的形状，从表示名－值对的两列转换为多列，来自输入的每个名称均为一列。输入中的名称列称为转轴(pivot)列。用户需要指定哪些名称要从输入转置到输出中的单独列。考虑表 *MonthlySales*(*ProductID*, *Month*, *SalesQty*)。下述使用 **pivot** 运算的查询为 Jan、Feb、Mar 每个月返回 *SalesQty* 作为单独的列。注意 **pivot** 运算还在表中所有其他列上执行隐式聚集，在 pivot 列上执行显式聚集。

```
select *
from MonthlySales pivot(sum(SalesQty) for Month in('Jan', 'Feb', 'Mar')) P;
```

  pivot 的反运算是 **unpivot**。

- **apply** 运算类似于连接。不过它右边的输入是一个表达式，可能包含了对左边输入中列的引用，例如一个以表为值的函数调用，它将来自左边输入的一列或多列作为变元。该运算产生的列的集合是来自两个输入的列的并。**apply** 运算可用于为其左边输入的每行计算右边输入的值，并在所有这些计算出的行上执行 **union all** 运算。**apply** 运算和 **joint** 一样有两类，分别是 **cross** 和 **outer**。这两类的不同之处在于它们如何处理右边输入产生空结果集的情况。在 **cross apply** 的情况下，这将导致来自左边输入的相应行不出现在结果中。在 **outer apply** 的情况下，来自左边 1230
  输入的行出现在结果中，且在右边输入的相应列上取 NULL 值。考虑一个以表为值的函数 *FindReports*，它的输入是给定雇员的 ID，并返回组织机构中直接或间接向该雇员报告的雇员的集合。以下的查询为 *Departments* 表中每个部门的经理调用这个函数：

```
select *
from Departments D cross apply FindReports(D.ManagerID)
```

30.2.3 例程

用户可以使用 Transact-SQL 或 .NET 语言编写例程，在服务器进程内部作为标量或表函数、存储过程和触发器来运行。所有这些例程在数据库中都可以用相应的 **create function**、**procedure**、**trigger** DDL 语句定义。标量函数可以在 SQL DML 或 DDL 语句内部的任意标量表达式中使用。以表为值的函

数可以用在 **select** 语句中任何允许出现表的地方。主体包含单条 SQL **select** 语句的 Transact-SQL 表值函数在引用该函数的查询中被当作视图(内联扩展)。由于表值函数允许输入变元，内联的表值函数就可看作参数化的视图。

30.2.3.1 带索引的视图

除了在 ANSI SQL 中定义的传统视图，SQL Server 还支持带索引的(物化)视图。带索引的视图充分增强了复杂决策支持查询的性能。该类查询检索基表中大量的行，并将大量信息聚集为简洁的总和、计数以及平均值。SQL Server 支持在一个视图上创建一个簇集索引以及任何数量的非簇集索引。一旦一个视图带索引，优化器就能在引用该视图或其基表的查询中使用它的索引。查询没必要为在查询计划中使用带索引的视图而显式提及视图，因为匹配是根据视图定义自动完成的。这样，现有查询能够从提升的效率中受益，因为可以直接从带索引的视图中检索数据，而无需重写。通过自动传播所有更新来维护带索引的视图和基本表是一致的。

30.2.4 带过滤的索引

带过滤的索引是一种优化的非聚集的索引，特别适合于从一个良好定义的数据子集中进行选择的
[1231] 查询。它使用一个过滤谓词来索引表中行的一部分。一个设计良好的带过滤的索引可以提高查询性能，降低索引维护开销，比全表索引更能减少索引存储代价。相对于全表索引，带过滤的索引可以提供下述好处：

- **改进查询性能和计划的质量。**设计良好的带过滤的索引改进查询性能和执行计划的质量，因为它比全表的非聚集索引小且有过滤统计信息。由于只包含在带过滤的索引中的行，过滤统计信息比全表统计信息更准确。
- **降低索引维护开销。**仅当数据操纵语言(DML)语句影响索引中的数据时，才进行索引维护。因为带过滤的索引更小且仅当该索引中的数据被影响时才维护，相对于全表非聚集索引，带过滤的索引降低了索引维护开销。有可能存在大量带过滤的索引，特别是当它们所包含的数据不经常被影响时。同样，如果带过滤的索引只包含频繁被影响的数据，这种更小的索引规模就小了更新统计信息的代价。
- **减少索引存储代价。**当不需要全表索引时，创建带过滤的索引能减少用于非聚集索引的磁盘存储。可以用多个带过滤的索引取代一个全表非聚集索引，而不会显著增加存储需求。

过滤的统计数据也可以显式地创建，独立于带过滤的索引。

30.2.4.1 可更新视图和触发器

通常，如果数据修改仅发生在视图的一个基表上，那么视图可以是 **update**、**delete** 或者 **insert** 语句的操作对象。对划分视图的更新操作会传播到多个基表。比如，下面的 **update** 将会把出版商"0736"的价格提高 10%：

```
update titleview
set price = price * 1.10
where pub_id = '0736'
```

对于影响到多个基表的数据修改，如果有一个为该操作定义的 **instead** 触发器，则该视图可以更新。**insert**、**update** 或 **delete** 操作的 **instead** 触发器可以定义在视图上，来指定必须在基表上执行更新以反映视图上的相应修改。

触发器是在基表或者视图上发出 DML(**update**、**insert** 或 **delete**)语句或 DDL 语句时自动执行的 Transact-SQL 或 .NET 过程。触发器是一种当数据被修改或执行 DDL 语句时允许自动执行商务逻辑的机
[1232] 制。触发器可以扩充描述性约束、默认值和规则的完整性检查逻辑。尽管只要描述性约束够用就应当优先使用它们，因为它们可以被查询优化器用来推断数据内容。

根据触发触发器的事件类型，触发器分为 DML 触发器和 DDL 触发器。DML 触发器是在正被修改的基表或视图上定义的。DDL 触发器是在整个数据库上对一条或多条 DDL 语句(例如 **create table**、**drop procedure** 等)定义的。

根据调用触发器的时间相对于触发它的动作的时间先后，可分为 **after** 触发器和 **instead** 触发器。

after 触发器在触发语句及随后的任何约束执行之后才执行。instead 触发器代替触发动作执行。instead 触发器可认为是类似于 before 触发器的，但是它实际上是取代了触发动作。在 SQL Server 中，DML after 触发器只能在基表上定义，而 DML instead 触发器能在基表或视图上定义。instead 触发器事实上允许任何视图都可以通过用户提供的逻辑而更新。DDL instead 触发器可在任意 DDL 语句上定义。

30.3 存储和索引

在 SQL Server 中，数据库是带一个包含数据的文件集合，它由单个事务日志来支持。SQL Server 中数据库是管理的主要单元，它还为物理结构（如表和索引）、逻辑结构（如约束和视图）提供容器。

30.3.1 文件组

为了有效管理数据库中的空间，数据库中的数据文件集被划分成若干组，称为文件组。每个文件组包含一个或多个操作系统文件。

每个数据库至少有一个文件组，称为主文件组。该文件组在系统表中保存了数据库的所有元数据。主文件组也可以存放用户数据。

如果用户创建了额外的自定义文件组，用户可以将单独的表、索引或者表的大对象列存放到特定的文件组中。从而显式地控制它们的位置。例如，用户可以选择把决定性能的关键索引存放到位于固态硬盘的文件组上。类似地，用户可以选择将包含视频数据的 varbinary(max)列放到为优化的 I/O 子系统上。

30.3.2 文件组内的空间管理

文件组的主要目的之一是进行有效的空间管理。所有数据文件都被划分成固定大小的 8KB 单元，称为页。分配系统负责把这些页分配给表和索引。分配系统的目标是把空间浪费的数量减到最小，同 1235
时保持数据库中的碎片数量最少，以保证好的扫描性能。为了实现这个目标，分配管理器通常以 8 个连续页（称作盘区）为单位进行所有页的分配和重分配。

分配系统通过各种位图来管理这些盘区。这些位图使得分配系统可以快速找到一个页或者盘区来进行分配。当执行一个表扫描或索引扫描时也用到这些位图。利用基于分配的位图做扫描的好处是，它允许按照磁盘顺序遍历属于表或索引叶结点层的所有盘区，这会大大提高扫描性能。

如果文件组中有不止一个文件，分配系统使用“比例填充”（proportional fill）算法来给该文件组上的任意对象分配盘区。每个文件都是按其中相对于其他文件自由空间的比例来填充的。这以近似相同的比率来填充文件组中的所有文件，并允许系统平均地使用文件组中的所有文件。文件也可以设置为若文件组用完空间则自动地增长。SQL Server 允许文件缩小。为了缩小一个数据文件，SQL Server 从文件的物理尾端将所有数据移到一个更靠近文件开头的位置，然后事实上缩小文件，把剩余空间释放给操作系统。

30.3.3 表

SQL Server 支持堆和聚集组织的表。在堆组织的表中，表的每行的位置完全由系统决定，用户不能用任何方式指定。堆中的行有一个固定的标识符，称作 row(RID)，该值一直保持不变，除非文件缩小且该行被移动。如果行变得是够大以至于它原先插入到的页面放不下了，该记录会移到别的地方，但在原来的位置保留了一个转发存根，这样，就仍然可以用其原先的 RID 来找到该记录。

在聚集索引组织的表中，表的行存放在一棵按索引的聚集码排序的 B^+ 树中。聚集索引的码也用作每行的唯一性标识符。聚集索引的码可以定义为非唯一性的。在这种情况下 SQL Server 添加一个隐藏列来使得该码唯一。聚集索引也用作搜索结构，来识别表中具有某个特定码值的行，或者扫描码在特定范围内的表中的一组行。聚集索引是最常用的表组织形式。

30.3.4 索引

SQL Server 还支持辅助的（非聚集）B^+ 树索引。如果查询仅涉及那些通过辅助索引就能得到的列，那就可以通过搜索来自索引叶结点层的页来处理该查询，而无需从聚集索引或堆中检索数据。拥有聚集索引的表上的非聚集索引包含聚集索引的码列。这样，聚集索引行可以移动到不同的页（通过分裂、1234

磁盘碎片整理甚至索引重建）而无需改变非聚集索引。

SQL Server 支持在表上添加计算列。计算列上的值是一个表达式，通常基于该行中其他列的值。SQL Server 允许用户在计算列上建立辅助索引。

30.3.5 分区

SQL Server 支持在表和非聚集索引上进行范围划分。分区索引由多棵 B+树组成，每个分区对应一棵 B+树。没有索引（堆）的分区表由多个堆组成，每个分区对应一个堆。简而言之，我们只涉及分区索引（聚集的或非聚集的），而在后面的讨论中忽略堆。

对大索引进行分区能使管理员更灵活地管理索引的存储，并提高一些查询的性能，因为分区扮演了粗粒度索引的角色。

对索引的分区由提供的分区函数和分区模式来共同指定。分区函数把分区列（索引中的任意列）的域映射到标号为 1 到 N 的分区。分区模式将分区函数产生的分区号映射到存储分区的特定文件组。

30.3.6 在线创建索引

在表上创建新索引和重建现有索引可以联机执行，例如，当 **select**、**insert**、**delete** 和 **update** 操作正在表上执行的时候。新索引的创建分三个阶段。第一个阶段为新索引简单地构造一棵空 B+树，并用目录显示新索引对维护操作是可用的。就是说，新索引必须由所有后续的 **insert**、**delete** 和 **update** 操作来维护，但它对查询不可用。第二阶段包括由扫描表来获得每行的索引列，对行排序并将它们插入新的 B+树中。这些插入必须和新 B+树中的其他行小心地交互，这些行是由在基表上的更新引起的索引维护操作插入在那里的。这种扫描是快照扫描，没有加锁，保证扫描看到仅具有在扫描开始时已提交事务的结果的修改表。这是通过使用 30.5.1 节中描述的快照隔离技术实现的。建立索引的最后阶段涉及更新目录以示索引创建已经完成，且索引对查询是可用的。

30.3.7 扫描和预读

[1235] SQL Server 中查询的执行包括对底层的表和索引的各种不同的扫描模式。这些包括有序和无序扫描，串行和并行扫描，单向和双向扫描，向前和向后扫描，全表或索引扫描以及范围或过滤扫描。

每种扫描模式都有一种预读机制，试图保持扫描先于查询执行的需要，以减少搜索和潜在开销，并充分利用磁盘空闲时间。SQL Server 预读算法利用来自查询执行计划的知识以促使预读，并保证只有那些查询真正需要的数据才被读取。预读的量根据缓冲池的大小、磁盘子系统能承受的 I/O 量以及查询执行时消耗数据的速率来自动定比。

30.3.8 压缩

SQL Server 支持对表和索引的行及页压缩。行压缩使用一种可变长度的数据类型格式，如整数，它在传统上被认为是定长的。页压缩去除了列上共同的前缀并为共同的值构建了一个逐页（per-page）字典。

30.4 查询处理和优化

SQL Server 的查询处理器基于一个可扩展的框架，它能很快地合并新的执行和优化技术。在扩展的关系代数中，任意 SQL 查询可用一棵操作树来表示。通过将这种代数的运算符抽象为迭代器（iterator），查询执行将数据处理算法封装为逻辑单元，这些单元之间使用 GetNextRow() 接口来通信。从初始查询树出发，查询优化器通过使用树形变换产生可选方案，并通过考虑迭代器行为和统计模型来估计待处理的行的数量，进而估计这些方案的执行代价。

30.4.1 编译处理概述

复杂查询提供了很大的优化机会，需要对查询块界线对运算符进行重排序，并仅基于估计的开销来选择查询计划。为了寻求这些机会，查询优化器与其他商用系统中采用的传统查询优化方法不同，它支持一个更通用的、纯代数的、基于级联优化器原型的框架。查询优化器是查询编译处理的一部分，由以下四步组成：

- **解析/绑定**(parsing/binding)。解析之后，绑定器使用目录来分解表名和列名。SQL Server 使用查询计划高速缓存来避免对相同的或结构相似的查询重复优化。如果没有高速缓存的计划可用，则产生一棵初始的操作树。操作树仅仅是关系运算的组合，并不受诸如查询块或派生表概念的限制，它们通常会阻碍优化。 [1236]
- **简化/规范化**(simplification/normalization)。优化器在操作树上运用简化规则得到一种规范的简化形式。在简化过程中，优化器决定并加载基数估计所需要的统计数据。
- **基于代价的优化**(cost-based optimization)。优化器用探测和实现规则来产生候选方案，估计执行开销，并选择预期开销最小的查询计划。探测规则为一个广泛的运算符集合实现重排序，包括连接和聚集重排序。实现规则引入候选方案，如归并连接和散列连接。
- **查询计划准备**(plan preparation)。优化器为选定的查询计划建立查询执行结构。

为获得最好的结果，基于代价的优化不是分阶段的。在每个阶段独立地优化查询的不同方面；它也不受限于单个的维，比如连接枚举。相反，一组转换规则定义了感兴趣的空间，并统一使用代价估计来选择一个有效的查询计划。

30.4.2 查询简化

简化时，只有保证产生更低代价的等价查询的转换才会应用。优化器尽量在操作树上把选择操作下推；它检测谓词间的冲突并考虑已声明的约束。它用冲突来标示可以从树中移除的子表达式。通常情况是消除那些从带不同约束的表中检索数据的 **union** 分支。

许多简化规则是上下文相关的。也就是说，只有在使用子表达式的上下文中，替代操作才有效。比如，如果随后有一个过滤操作将丢弃那些用 **null** 填充的不匹配的行，那么外连接可简化为内连接。另一个例子是，当以后不再用到参照表上的列时，可以消除外码上的连接。第一个例子是在重复不敏感的上下文中，这意味着传送一行的一个或多个拷贝并不影响查询结果。半连接以及 **distinct** 下的子表达式是重复不敏感的，例如，它们允许把 **union** 转换成 **union all**。

GbAgg 运算符用于分组和聚集，它创建分组，并有选择性地在各个分组上运用聚集函数。SQL 中用 **distinct** 关键字表达的消除重复只不过是一种不计算聚集函数的 *GbAgg*。在简化时，有关关键字和函数依赖的信息用于减少分组的列。

通过去除相关的查询描述并使用一些替代的连接变量，可以将子查询规范化。去除相关性并不是"子查询执行策略"，而仅仅是一个规范化步骤。然后在基于代价的优化中将考虑各种执行策略。 [1237]

30.4.3 重排序和基于代价的优化

在 SQL Server 中，转换完全集成到执行计划的基于代价的生成和选择中。查询优化器包括大约 350 条逻辑的和物理的转化规则。除了内连接的重排序，查询优化器还对来自标准关系代数(对 SQL 而言，带有重复)的外连接、半连接和反半连接运算使用重排序转换。*GbAgg* 操作也是重排序的，通过在可能的时候把它移到连接之下或之上。部分聚集，即引入一个新的 *GbAgg* 对随后的 *GbAgg* 的列的超集进行分组，是在连接和 **union all** 之下考虑的，并在并行计划中已是如此。详细描述请参考文献注解中给出的参考文献。

相关性执行在探测计划时考虑，最简单的情形是索引查找连接。SQL Server 将相关性执行建模为单个代数运算符，称为 **apply**，它在表 T 以及一个参数化的关系表达式 $E(t)$ 上操作。**Apply** 为 T 的每一行执行 E，其中 T 提供参数值。相关性执行被认为是不考虑在原始 SQL 表达式中使用子查询的一种替代执行。当表 T 很小并且索引能支持 $E(t)$ 有效的参数化执行时，这是一种很有效的策略。另外，当有重复的参数值时，我们考虑通过以下两种技术来减少 $E(t)$ 的执行次数：在参数值上对 T 排序，当参数值保持不变时可以重用单个 $E(t)$ 的结果；或者利用一个散列表保持对 $E(t)$ 关于较早参数值(的某些子集)的结果的追踪。

某些应用基于行所在组的一些聚集结果来选择行。比如，"找出那些余额比他们所在市场段的平均余额的两倍还多的客户"。这个 SQL 表达式需要自连接。在探测方案时，这种模式将被检测到，并考虑将在单次扫描上按每段执行作为自连接的替代方案。

在基于代价的优化中，还可以考虑利用物化视图。在物化视图使用中视图匹配和运算符重排序的相互影响可能并不明显，除非发生了某些其他的重排序。当发现一个视图与某个子表达式匹配时，包含该视图结果的表被加进来，作为相应表达式的替换方案。它也许比原来的表达式更好，也许更差。这依赖于数据分布和所能利用的索引。究竟选择哪种方案取决于对它们的代价估计。

为了估算一个计划的执行代价，模型考虑了子表达式执行的次数和行目标(row goal)，它是预计会被父操作消耗的行数。在top-n查询和*Apply/semijoin*情况下，行目标可以少于估计的基数。例如，只要
[1238] $E(i)$产生一行(即它测试**exists** $E(i)$)，*Apply/semijoin*就把行i从表I中输出。这样，$E(i)$输出的行目标就是1，为$E(i)$的这个行目标计算出$E(i)$子树的行目标，并用于代价估计。

30.4.4 更新计划

更新计划优化索引维护、约束检验、级联动作应用、物化视图维护。对于索引维护，更新计划并不是考虑每一行并为它维护所有索引，而是按每个索引实施修改。按关键字顺序将行排序并实施更新操作。这就把随机I/O减到最少，尤其是当待更新的行数很大时。约束是通过**assert**运算符来处理的，它执行一个谓词，如果结果是**false**就抛出一个错误。参照约束是通过**exists**谓词来定义的，它相应地变成一些半连接并通过考虑各种执行算法来进行优化。

万圣节问题(在前面13.6节介绍过)是指如下的反常：假如工资索引按升序读取，并且将工资增涨10%。更新的结果是，行会在索引中向前移动，于是它们又被发现并再次更新，导致无限循环。解决这个问题的一个办法是，把处理过程分成两个阶段：首先，读出所有将要更新的行，并把它们复制到某些临时空间中，然后从这个空间读取并实施所有更新。另一个办法是从另外一个索引中读，该索引中的行不会因为更新结果而移动。如果在待更新的行上运行排序或建立散列表，则有些执行计划提供了自动的阶段分离。万圣节问题防护是作为计划的性质而建模的。考虑提供所需性质的多个计划，并基于估计的执行代价选择其中一个。

30.4.5 优化时的数据分析

把统计数据收集作为正在运行的优化的一部分执行是SQL开创性的技术。计算结果的大小估计基于给定表达式中所用到的列上的统计数据。这些统计数据包括列值上的最大差异直方图和许多计算密度与行大小的计数器，等等。数据库管理员可以用扩展的SQL语法来显式创建统计数据。

不过如果给定列上没有统计数据可用，SQL Server的优化器将暂停正在进行的优化，并收集所需的统计数据。一旦统计数据计算出来，原来的优化利用新产生的统计数据再继续进行。后续查询的优化重用以往产生的统计数据。典型地，经过一小段时间后，频繁使用列的统计数据已经建好。为收集新的统计数据而发生的中断就变得不频繁了。通过保持对表中被修改行数的跟踪，为所有受影响的统计数据维护了一个过期度量。一旦过期超过了特定阈值，将重新计算统计数据，并重新编译高速缓存的查询计划以考虑改变后的数据分布。

统计数据可以异步地重新计算，这避免了同步计算引发的潜在的较长的编译时间。触发统计计算
[1239] 的优化潜在地使用旧的统计数据。然而，随后的查询能够利用重新计算的统计数据。这允许在优化花费的时间和结果查询计划的质量之间取得可接受的平衡点。

30.4.6 部分搜索和启发式搜索

基于代价的查询优化器将面临搜索空间爆炸问题，因为应用程序可能发出涉及几十个表的查询。为了解决该问题，SQL Server使用多个优化阶段，每个阶段运用查询转换来连续探测更大范围的搜索空间。

有简单而完整的转换来穷举所有优化方案，也有智能转换来实现各种启发式搜索。智能转换产生的查询计划在搜索空间中相隔很远，而简单转换探测相邻空间。优化阶段采用这两种转换的混合，首先强调智能转换，然后换成简单转换。子树上的最佳结果被保存下来，以便后续阶段可以利用早先生成的结果。每个阶段都需要权衡对立计划产生技术：

- **穷举产生替代方案**：为了生成完整的空间，优化器使用完全的、局部的、非冗余的转换。一条转换规则等价于一系列更原始的转换。这种规则只会增加额外的开销。

- **启发式产生候选方案**：少数几个可能的候选者（在代价估计的基础上选择出来）可能在原始转换规则方面相差很远。这里，所需的转换是非完全的、全局的和冗余的。

当产生出第一个查询计划以后，优化可以随时终止。这样的终止是基于已经发现的最佳计划的开销估计和已在优化中花去的时间。比如，如果一个查询只需在一些索引中查看某几行，则在早期阶段就可能很快产生一个非常廉价的计划，从而终止优化；在合适的时候这种方法允许随后容易地增加新的启发式方法，而不用在基于代价的计划选择或者搜索空间的穷举探测之间折中。

30.4.7 查询执行

执行算法支持基于排序和基于散列的处理，而且它们的数据结构被设计成优化利用处理器高速缓存。散列操作支持基本的聚集和连接，带有许多优化、扩展和对数据倾斜的动态调整。**flow-distinct** 运算符是 hash-distinct 的一个变体。一旦找到新的独特值，就输出行，而不用等处理完整个输入。这个操作对于那些使用 **distinct** 且只请求少数几行的查询很有效，例如使用 **top n** 结构。指定执行 $E(r)$ 的相关性计划，对表 T 的每一行 r，通常包含一些基于参数的索引查找。**异步预取**（asynchronous prefetching）允许缓存储引擎发出多个索引查找请求。它是通过这种方式实现的：对表 T 的行 r，发出一个无阻塞的索引查找请求，然后将 r 放在预取队列中。这些行从队列中取出并由 **apply** 用于执行 $E(r)$。$E(r)$ 的执行并不需要数据已经在缓冲池中，但是有引导的顺序操作会最大限度地利用硬件并提升性能。队列大小是由一个高速缓存函数动态决定的。如果 **apply** 的输出行无需排序，那么为了最小化 I/O 等待时间，从队列中取出行也可不按顺序。 1240

并行执行是通过 **exchange** 运算符实现的。它管理多线程、分区或广播数据，并将数据送入多个进程。查询优化器根据代价估计来决定 **exchange** 的位置。并行度根据当前系统利用率，在运行时动态决定。

索引计划由前面描述的那些部分组成。比如，我们以基于代价的方式考虑使用索引连接来解决谓词合取（或者用索引并来解决谓词析取）。这样的连接可以用 SQL Server 的任何一个连接算法来并行执行。仅仅为了把查询中需要的一组列拼接成一行，我们也考虑使用连接索引。它有时候比扫描基表要快。从辅助索引中取出记录 ID 并在基表中定位相应的行，与执行索引查找连接同样有效。为此，我们使用普通的相关执行技术，比如异步预取。

与存储引擎的通信通过 OLE-DB 实现，它允许访问那些实现该接口的其他数据源提供者。OLE-DB 是用于分布式和远程查询的机制，它由查询处理器直接驱动。数据源提供者可根据它们提供的功能范围来分类，包括从无索引能力的简单行集合基层者，到完全支持 SQL 的提供者。

30.5 并发与恢复

SQL Server 的事务、日志、封锁以及恢复子系统实现了数据库系统所需的 ACID 特性。

30.5.1 事务

在 SQL Server 中每条语句都是原子的。应用程序能够为每条语句指定不同的隔离性级别。单个事务可以包含的语句不仅可以是选择、插入、删除或更新记录的语句，还可以是创建或删除表、建立索引和批量导入数据。事务可以跨越远程服务器上的数据库。当事务散布在不同服务器之间时，SQL Server 使用一个称作微软分布式事务协调器（Microsoft Distributed Transaction Coordinator，MS DTC）的 Windows 操作系统服务来执行两阶段提交处理。MS DTC 支持 XA 事务协议，并连同 OLE-DB 一起，为异构系统间的 ACID 事务提供了基础。 1241

SQL Server 默认使用基于封锁的并发控制。SQL Server 还为游标提供了乐观的并发控制。乐观的并发控制是基于这样一种假设：多个用户之间资源冲突的可能性很小（但不是不可能），因此允许事务执行，无需封锁任何资源。只有当试图改变数据时，SQL Server 才检查资源以确定是否发生过冲突。如果发生冲突，应用必须重新读数据并再次尝试改变。应用可以选择通过比较值或者通过检查行上一个特殊的行版本列来检测改变。

SQL Server 支持的 SQL 隔离性级别有：未提交读、已提交读、可重复读和可串行化。已提交读是

默认级别。另外，SQL Server 支持两个基于快照的隔离性级别(前面在 15.7 节中描述过快照隔离)。

- **快照**(snapshot)：指一个事务中的任何语句读到的数据将是事务开始时所存在的数据的事务一致性版本。其效果就好像事务中的语句看到的是在事务开始时存在的已提交数据的一个快照。写操作使用 15.7 节描述的验证步骤来验证，只有验证成功才允许完成。
- **已提交读快照**(read committed snapshot)：指在一个事务中执行的每条语句看到的都是语句开始时存在的数据的事务一致性快照。这和已提交读隔离形成对比，在那里语句能看到在语句执行时已提交事务所提交的更新。

30.5.2 封锁

封锁是用于实施隔离性级别语义的主要机制。所有更新获得足够的在事务整个执行期间持有的排他锁以防止发生更新冲突。不同持续时间所持有的共享锁为查询提供不同的 SQL 隔离性级别。

SQL Server 提供了多粒度封锁，允许事务给不同类型的资源加锁(见图 30.4，其中资源按粒度由小到大的次序排列)。为了使封锁开销最小，SQL Server 按任务自动给资源加合适粒度的封锁。在越小粒度上加锁，比如在行上加锁，提高了并发度，但有更大的开销，因为如果有许多行需要封锁就必须持有更多的锁。

资源	描述
RID	行标识符，用于封锁堆中单独的一行
Key	在一个索引中的行锁，在可串行化事务中保护键的范围
Page	8KB的表或索引页
Extent	一组连续的8个数据页或索引页
Table	全表，包括所有数据和索引
DB	数据库

图 30.4 可封锁的资源

基本的 SQL Server 封锁模式有：共享锁(S)、更新锁(U)和排他锁(X)；还为多粒度锁提供了意向锁。更新锁用来防止一种常见的死锁形式：当多个会话读取、封锁、然后潜在地更新资源时会发生这种死锁。另一种封锁模式，叫码范围锁，只用在可串行化隔离性级别中，用于封锁索引中两行之间的范围。

30.5.2.1 动态锁

细粒度的锁可以提高并发度，但需要花费额外的 CPU 周期和内存来获得和持有很多封锁。对于很多查询，较粗的封锁粒度提供了更好的性能，同时没有(或有很小的)并发性损失。数据库传统上需要查询提示和表选项，让应用程序确定封锁粒度。另外，还有配置参数(通常是静态的)来确定分配多少内存给封锁管理器。

在 SQL Server 中，为获得查询中用到的每个索引的最佳性能和并发度，封锁粒度是自动优化的。而且，分配给封锁管理器的内存是基于系统其他部分(包括机器上的其他应用程序)的反馈而自动调整的。

在查询执行前，先对查询中用到的每个表和索引做封锁粒度的优化。封锁优化处理考虑了隔离性级别(即封锁保持多长时间)、扫描类型(范围扫描、探查扫描还是全表扫描)、估计扫描的行数、选择率(所访问行中，符合条件的行的百分数)、行密度(每页行数)、操作类型(扫描、更新)、用户在粒度上的限制以及可用的系统内存。

一旦执行查询，如果系统获得的封锁数量远远超过优化器的估计，或者可用的内存数量下降且不能支持所需的封锁数量，封锁粒度会自动升级到表级。

30.5.2.2 死锁检测

SQL Server 自动检测涉及锁和其他资源的死锁。例如，如果事务 A 拥有 Table1 的锁并等待可用的内存资源，而事务 B 有一些内存，但在封获得在 Table1 上的锁才会释放，那么这两个事务就形成了死锁。线程和通信缓冲区也会陷入死锁。当 SQL Server 检测到死锁时，它考虑每个事务已完成的工作量，

选择代价最小的事务作为死锁牺牲者，让它回滚。

频繁的死锁检测会有损系统性能。SQL Server 根据死锁发生频率，自动调整死锁检测频率。如果死锁不频繁，则检测算法每 5 秒运行一次。如果死锁频繁，则一旦某个事务等待锁，就会开始检测算法。 1242 1243

30.5.2.3 用于快照隔离的行版本

两种基于快照的隔离性级别使用行版本来达到查询隔离，同时不把查询阻塞在更新之后，反之亦然。在快照隔离下，更新和删除操作生成受影响的行的版本，并将它们存储到临时数据库中。当没有活跃事务再需要它们时，这些版本就是收集的垃圾。因此，在快照隔离下运行的查询不需要获得锁，而可以读取被另一个事务更新/删除的任何记录的旧版本。行版本还用于为联机建立索引操作提供表的快照。

30.5.3 恢复和可用性

SQL Server 被设计成可以从系统和媒介故障中恢复，恢复系统能够支持有极大缓冲池(100GB)和有上千个磁盘驱动器的机器。

30.5.3.1 从崩溃中恢复

逻辑上，日志是潜在无限的，由日志序列号(LSN)标识的日志记录流。物理上，该流的一部分存储在日志文件中。日志记录保存在日志文件中，直到它们被备份并且系统在回滚或复制时不再需要它们为止。日志文件根据所需存储的记录来自动调整文件大小。在不阻塞当前任何操作的情况下，附加的日志文件可以加到正在运行的数据库中(例如，在新的磁盘上)，所有日志被当作是一个连续文件。

SQL Server 的恢复系统与 ARIES 恢复算法(见 16.8 节)有很多方面是类似的，本节强调了它们之间的一些主要不同。

SQL Server 有一个称为**恢复间隔**(recovery interval)的配置选项，它允许管理员限制 SQL Server 在崩溃后恢复所花的时间长短。服务器动态调整检查点频率，以将恢复时间减少到恢复间隔之内。检查点从缓冲池刷新所有的脏页面，根据 I/O 系统的能力和它的当前工作负载进行调整，以有效地消除对正在运行事务的任何影响。

崩溃后重启时，系统启动多个线程(根据 CPU 数量自动调整)，以开始并行恢复多个数据库。恢复的第一阶段是日志分析过程，它建立一个脏页表和活跃事务列表。第二阶段是重做过程，从最后一个检查点开始重做所有操作。在重做阶段，脏页表用来促成数据页的预读。最后一个阶段是撤销阶段，其中未完成的事务回滚。撤销阶段实际上分成两部分，因为 SQL Server 使用两级恢复方案。第一级的事务(那些涉及诸如空间分配和页面分割这样的内部操作的事务)首先回滚，然后是用户事务。一旦第一级事务回滚完成，数据库就被接上线，且当执行最后的回滚操作时，就可用于开始新的用户事务。这由重做过程为那些在撤销阶段会回滚的所有未完成的用户事务重新获得封锁来实现。 1244

30.5.3.2 介质恢复

SQL Server 的备份和修复能力允许系统从很多故障中恢复，包括磁盘介质的丢失和损坏、用户错误以及服务器的永久损失。并且，备份和修复数据库还可以有其他用途，比如把数据库从一台服务器复制到另外一台，并维护备用系统。

SQL Server 有三种不同的恢复模型，用户可以为每个数据库从中选择恢复模型。通过指定一种恢复模型，管理员声明所需恢复能力的类型(如时间点修复和日志传送)以及实现它们所需的备份。备份可以发生在数据库、文件、文件组和事务日志上。所有备份都是模糊的且完全联机的；也就是说，它们在执行时并不妨碍任何 DML 或 DDL 操作。修复也可以联机进行，仅需将正在修复的数据库部分(例如损坏的磁盘块)离线。备份和修复操作是高度优化的，仅仅受进行备份的介质速度的限制。SQL Server 能够备份到磁盘和磁带设备上(并行多达 64 个)，并为第三方备份产品的使用提供高性能的备份 API。

30.5.3.3 数据库镜像

数据库镜像包括将数据库(主数据库)的每次更新立即复制到一个独立的、完整的数据库拷贝(镜像数据库)上，此拷贝通常放在另一台机器上。在主服务器上发生灾难或甚至仅仅为了维护的情况下，

系统将在几秒钟内自动故障转移到镜像上。应用程序使用的远信库知道镜像的存在，且在故障转移时会自动重新连接到镜像机器。主数据库上的事务日志块一旦产生，就发送到镜像并在镜像上重做日志记录，这样就实现了主数据库和镜像之间的紧密耦合。在完全安全模式下，直到事务的日志记录写到
[1245] 镜像的磁盘上，该事务才能提交。除了支持故障转移，镜像也可用于自动修复页面，其方法是一旦在尝试读页面时发现该页面损坏就从镜像复制它。

30.6 系统体系结构

SQL Server 实例是单个的操作系统进程，也是 SQL 执行中请求的命名终点。为了执行 SQL，应用程序通过各种客户端库(像 ODBC、OLE-DB 和 ADO.NET)来与 SQL Server 交互。

30.6.1 服务器上的线程池

为了使服务器上的上下文切换最少，并控制多道程序设计的程度，SQL Server 进程维护一个线程池来执行客户端请求。当请求从客户端到达时，分配一个线程来执行。该线程执行客户端发出的 SQL 语句，并把结果返回给客户端。一旦完成客户请求，线程就返回到线程池。除了用户请求，线程池也用来给内部后台任务分配线程，诸如：

- **懒散写(lazywriter)线程**：这个线程致力于保证特定数量的缓冲池是空闲的，在任何时候都可用于系统分配。该线程也同操作系统交互，来决定 SQL Server 进程所应该消耗的最佳内存量。
- **检查点(checkpoint)线程**：这个线程周期性地给所有数据库设检查点，以便在服务器重启时，为数据库维护快速的恢复间器。
- **死锁监视器(deadlock monitor)线程**：这个线程监视其他线程，在系统中寻找死锁。它负责先锁检测并选择牺牲者，以使系统继续运行。

当查询处理器选择一个并行计划来执行特定查询时，它能分配多个线程代表主线程执行该查询。由于 Windows NT 家族的操作系统提供本地线程支持，SQL Server 利用 NT 线程来执行。然而，在非常高端的系统中，SQL Server 可以配置成除了核心线程以外，还可以与用户模式线程一起运行，以避免在线程切换时核心上下文切换的开销。

30.6.2 内存管理

在 SQL Server 进程中内存有许多不同的用途：

- **缓冲池(buffer pool)**。系统中内存的最大消耗者就是缓冲池。缓冲池维护了最近使用过的数据库页的高速缓存。它使用一种带窃取、非强制策略的时钟替换算法。也就是说，未提交更新的
[1246] 缓冲页可能被替换出去("被窃取")，并且在事务提交时并不强制将缓冲页写到磁盘上。缓冲区也遵循先写日志协议，以保证崩溃和介质恢复的正确性。
- **动态内存分配(dynamic memory allocation)**。这是为执行用户提交的请求而动态分配的内存。
- **计划与执行高速缓存(plan and execution cache)**。这个高速缓存保存了系统中用户以前执行过的各种查询的已编译过的计划。这就允许不同用户可以共享相同的计划(节约内存)，同时也为相似查询节省了查询编译时间。
- **提供大内存(large memory grant)**。这用于消耗大量内存的查询操作符，比如散列连接和排序。

SQL Server 用一个精巧的内存管理方案给上述各种使用划分内存。单个内存管理器集中地管理 SQL Server 使用的内存。这个内存管理器负责在系统中各内存消费者之间动态划分和重新分配内存。它根据对内存的任何特定使用相关的成本效益分析来分配这些内存。所有部件都可以使用通用的 LRU 基础机制。这种高速缓存基础机制不仅跟踪高速缓存数据的生命周期，还跟踪创建和高速缓存这些数据而导致的相关 CPU 和 I/O 代价。这些信息用来确定不同数据高速缓存的相关代价。内存管理器的重点集中在丢弃那些最近没有使用且高速缓存代价小的高速缓存数据。例如，给定相同的访问频率，需要好几秒 CPU 时间来编译的复杂查询计划比简单计划更可能驻留在内存中。

内存管理器同操作系统交互，以动态决定它应该消耗系统总内存量中的多少。这使得 SQL Server 在使用系统中的内存方面非常积极，但还能保证在其他程序需要内存且不增加额外的页面错误时，能

把内存返还给系统。

另外内存管理器知道系统的 CPU 和内存的拓扑结构。特别地，它利用许多机器采用的 NUMA（非均匀的内存访问），并尝试维护在执行线程的处理器和它所访问的内存之间的位置。

30.6.3 安全性

SQL Server 为认证、授权、审计和加密提供了全面的安全机制和策略。认证可以通过 SQL Server 管理的用户名－密码对，也可以通过 Windows OS 账户。授权是通过把权力授予给模式对象或覆盖在诸如数据库或服务器实例那样的容器对象上来管理的。在检查授权的时候，权力被向上钻取并计算，说明主体的权力覆盖和角色成员资格。审计采用与权力相同的方式来定义——对于给定的主体或包含对象定义在 [1247]
模式对象上，并且在操作时它们是基于对象上的审计定义和关于主体的任何覆盖审计或角色成员资格的说明而动态计算的。为了不同目的的审计可以定义多个审计，如 Sarbanes Oxley 和 HIPAA [三]，可以分别管理而没有彼此破坏的风险。审计记录可以写到文件中或 Windows Security Log 中。

SQL Server 同时提供了数据的手动加密和透明加密。透明的数据加密在写到磁盘时加密所有的数据页和日志页，并在从磁盘读出时解密，所以数据驻留在磁盘上时是加密的，但对 SQL Server 用户却是明文，不需要修改应用。透明的数据加密比手动加密的 CPU 效率更高，因为数据只在写到磁盘时加密，并且这是在更大的单元、页上完成的，而不是在个别的数据单元上完成的。

有两件事情对用户的安全性甚至更为关键：(1) 整个代码库自身的质量，(2) 用户决定他们是否正确地保护了系统的能力。通过使用安全开发生命周期（Security Development Lifecycle）能提高代码库的质量。产品的所有开发人员和测试人员都经过安全培训。所有特性都被威胁建模以确保资产受到合适的保护。只要有可能，SQL Server 利用操作系统底层的安全特征而不是自己来实现，例如 Windows 操作系统认证（Windows OS Authorization）和用于审计记录的 Windows 安全日志（Windows Security Log）。另外，利用大量内部工具来分析代码库，寻找潜在的安全隐患。使用模糊测试 [二] 和测试威胁模型来验证安全性。在发布之前，还有对产品的最后一次安全性复查，响应计划也到位了，用于应对发布之后发现的安全问题——当发现问题之后就执行该计划。

提供大量特性来帮助用户正确地保护系统。一个这样的特性是一项称为默认关闭（off-by-default）的基本方针，其中许多不常用的或者那些需要额外安全性考虑的部件在默认情况下是完全禁用的。另外一个特性是最佳实践分析器，它针对可能导致安全漏洞的系统设置的配置给用户发出警告。基于策略的管理进一步允许用户去定义这些设置应该是什么样的，是警告或者是防止修改可能与认可的设置发生冲突。

30.7 数据访问

SQL Server 支持如下应用编程接口（API），用于构建数据密集型应用： [1248]

- **ODBC**。这是微软对标准 SQL：1999 调用层接口（CLI）的一个实现。它包括对象模型——远程数据对象（RDO）和数据访问对象（DAO）——这使得用像 Visual Basic 这样的编程语言来编写多层数据库应用更加容易。
- **OLE-DB**。这是为程序员设计的用来构建数据库组件的一个低层次的、面向系统的 API。该接口是根据微软的组件对象模型（Component Object Model，COM）构架的，它允许封装低层的数据库服务，如 rowset providers、ISAM providers 以及查询引擎。SQL Server 中使用 OLE-DB 来集成关系查询处理器和存储引擎，并且能够对 SQL 和其他外部数据源进行复制和分布式访问。和 ODBC 类似，OLE-DB 包括称为 ActiveX Data Objects（ADO）的更高层对象模型，让使用 Visual Basic 编写数据库应用程序更加简单。
- **ADO.NET**。这是为用 .NET 语言（例如 C# 和 Visual Basic .NET）编写的应用设计的 API。此接口

三 Sarbanes-Oxley 法案是美国政府金融规则法案，HIPAA 是美国政府卫生保健法律，它包含卫生保健相关信息的法则。

二 模糊测试是一种基于随机化的技术，关于料想之外的、可能无效的、输入的测试。

简化了 ODBC 和 OLE-DB 支持的一些公共数据访问模式。另外，它提供了新的数据集（data set）模型来支持无状态、非连贯的数据访问应用。ADO.NET 包括 ADO.NET 实体框架（entity framework），它是一个在这样的数据上进行编程的平台，这些数据将抽象层次从逻辑（关系）层提升到概念（实体）层，从而大大降低了应用和数据服务（如报表、分析和复制）的脆抗失配。概念数据模型是使用扩展关系模型实现的，即**实体数据模型**（Entity Data Model，EDM），它包含实体和联系作为最高级的概念。它包括一种用于 EDM 的称为**实体 SQL**（entity SQL）的查询语言，一个全面的映射引擎把概念层翻译为逻辑（关系）层，以及一组模型驱动工具帮助开发人员定义对象与实体到表之间的映射。

- LINQ。**集成语言的查询**（Language-INtegrated Query，**LINQ**）允许在编程语言（如 C#和 Visual Basic）中直接使用描述性的、面向集合的构造。查询表达式不是由外部工具或语言预处理器来处理的，而是由语言本身的最高级表达式。LINQ 使得查询表达式受益于丰富的元数据、编译时语法检查、静态类型和之前只对命令式代码可用的自动完成。LINQ 定义了一个通用的标准查询操作集合，允许遍历、过滤、连接、投影、排序和分组操作在任何基于.NET 的编程语言中可以用直接的甚至描述性的方式表达。C#和 Visual Basic 包支持查询理解，如：利用标准查询操作的语言语法扩展。
- **DB-Lib**。为 C API 而设的 DB-Library 是为了在 SQL-92 标准之前的 SQL Server 早期版本中使用而专门开发的 API。
- **HTTP/SOAP**。应用可以使用 HTTP/SOAP 请求来调用 SQL Server 查询和过程。应用可以使用这样的 URL，它指定了网络信息服务器（Internet Information Server，IIS）的虚拟根目录，该根目录引用了一个 SQL Server 实例。该 URL 可以包含 XPath 查询、Transact-SQL 语句或 XML 模板。

1249

30.8 分布式异构查询处理

SQL Server 的分布式异构查询能力允许通过运行在一台或多台机器上的 OLE-DB 数据提供者对多种关系型以及非关系型数据源进行事务查询与更新。SQL Server 提供两种方法在 Transact-SQL 语句中引用异构的 OLE-DB 数据源。链接服务器命名（linked-server-names）方法用系统存储过程将服务器名称和 OLE-DB 数据源关联起来。在这些链接服务器中的对象可以在 Transact-SQL 语句中使用下面描述的四部分命名约定来引用。例如，如果一台名为 *DeptSQLSrvr* 的链接服务器是在另一份 SQL Server 拷贝上定义的，下述语句引用该服务器上的一个表：

```
select *
from DeptSQLSrvr.Northwind.dbo.Employees;
```

OLE-DB 数据源在 SQL Server 中作为链接服务器注册。一旦定义了链接服务器，它的数据就能用四部分命名来访问：

```
< linked-server >.< catalog >.< schema >.< object >
```

下面的例子通过 Oracle 的 OLE-DB 提供者来建立连接到 Oracle 服务器的链接服务器：

```
exec sp_addlinkedserver OraSvr, 'Oracle 9.3', 'MSDAORA', 'OracleServer'
```

该链接服务器上的一个查询表达为：

```
select *
from OraSvr.CORP.ADMIN.SALES;
```

此外，SQL Server 支持内置的参数化的表值函数，称为 **openrowset** 和 **openquery**，它们允许用提供者支持的本地语言把未经解释的查询分别发送给提供者或链接服务器。下述查询合并了存储在 Oracle 服务器中和微软索引服务器（Microsoft Index Server）中的信息。它列出了包含词语“Data”和“Access”的所有文档以及它们的作者，并按作者的部门和名称排序。

1250

```
select e.dept, f.DocAuthor, f.FileName
from Orders.Corp.Admin.Employee e,
openquery( EmpFiles,
          'select DocAuthor, FileName
          from scope("c:\EmpDocs")
          where contains(' "Data" near() "Access" ') >0') as f
where e.name = f.DocAuthor
order by e.dept, f.DocAuthor;
```

关系引擎用 OLE-DB 接口打开链接服务器上的行集合(rowset)，获取行并管理事务。对于每个作为链接服务器访问的 OLE-DB 数据源，在运行 SQL Server 的服务器上必须有一个 OLE-DB 提供者，能在特定 OLE-DB 数据源上使用的 Transact-SQL 操作集合依赖于 OLE-DB 提供者的能力。只要性价比高，SQL Server 就把关系操作(比如连接、约束、投影、排序和分组操作)推给 OLE-DB 数据源。SQL Server 用微软的分布式事务协调器(Microsoft Distributed Transaction Coordinator)以及提供者的 OLE-DB 事务接口来保证跨越多个数据源的事务的原子性。

30.9 复制

SQL Server 的复制是一套技术，用来将数据和数据库对象从一个数据库复制并分布到另一个数据库，并跟踪变化，在不同数据库之间保持同步以维护一致性。SQL Server 复制还提供大部分数据库模式变化的内联复制，无须任何中断或重新配置。

典型的数据复制是为了增加数据可用性：由于建立报表的目的，复制能够让靠近自地理位置分散的站点的全部数据，并给在局域网上的远程用户，拨号连接或因特网上的移动用户分发数据。通过向外扩展以改善在各副本之间整体读的性能(这在为 Web 网站提供的中间层数据高速缓存服务中是常见的)，微软 SQL Server 复制还提高了应用的性能。

30.9.1 复制模型

SQL Server 把**出版－订阅**(Publish-Subscribe)比喻引入到数据库复制中，并把这种出版业比喻扩展到其整个复制管理和监视工具中。

出版者(publisher)是一个使得数据可用于复制到其他服务器的服务器。出版者可以有一个或多个出版物，每个代表逻辑相关的数据和数据库对象的集合。出版物中的离散对象(包括表、存储过程、用户自定义函数、视图、物化视图以及更多)称为**文章**(article)。向出版物增加一篇文章允许广泛定制该 [1251]
对象复制的方式，比如限制哪些用户可订阅以接受它的数据，以及数据复应该在表的投影或选择的基础上分别通过"水平"和"垂直"过滤器来如何过滤。

订阅者(subscriber)是从出版者那里接收复制数据的服务器。订阅者可以方便地从一个或多个出版者那里只订阅他们所需的出版物，无需考虑每个实现的复制选项的数量或类型。依赖于所选择的复制选项类型，订阅者要么用作只读副本，要么可以改变数据，并把改动自动传播回出版者，继而传播到所有其他副本。订阅者也可以重新出版他们所订阅的数据，支持和企业需求一样灵活的复制拓扑。

分发者(distributor)是扮演多种角色的服务器，这取决于所选择的复制选项。至少它用作存储历史和错误状态信息的仓库。在其他情况下，它被额外地用作存储与转发队列的中介，将复制的有效载荷的递送扩展到所有订阅者。

30.9.2 复制选项

微软 SQL Server 复制提供了很大范围内的复制选项。为了选择使用合适的复制选项，数据库设计者必须确定应用需求，考虑所涉及站点的自治操作以及所需的事务一致性程度。

快照复制(snapshot replication)完全按照在某一时刻出现的数据与数据库对象来复制和分发它们。快照复制无需跟踪持续的变化，因为变化不会增量地传播给订阅者。订阅者通过由出版定义的完全刷新的数据集来定期更新。快照复制的可用选项可以过滤发布数据，并允许订阅者修改复制数据并把这些变化传播回出版者。这种类型的复制最适合于较少量的数据，并且当更新通常影响到足够多的数据时，复制数据的完全刷新是有效的。

利用**事务复制**(transactional replication)，出版者把数据的初始快照传播给订阅者，然后以离散的事务或命令的形式把增量的数据修改转发给订阅者。增量修改的跟踪发生在 SQL Server 核心引擎的内部，它在出版数据库的事务日志中标记影响复制对象的事务。一个称作**日志读取代理**(log reader agent)的复制进程从数据库事务日志里读取这些事务，应用可选的过滤器，并把它们保存在分发数据库中。分发数据库的行为就像是支持事务复制的存储与转发机制的可靠队列(可靠队列和 26.1.1 节中描述的持久队列一样)。另外一个称为**分发代理**(distribution agent)的复制进程随后将这些变化转发到每个订阅者。与快照复制类似，事务复制为订阅者提供选项，使得更新要么使用两阶段提交来反映那些变化在发布
1252 方和订阅方是一致的，要么在订阅方将变化排队，由一个复制进程来异步唤醒，该复制进程稍后将变化传播给出版者。当需要保存多个更新之间的中间状态时，这种类型的复制更合适。

归并复制(merge replication)允许企业中每个副本完全自治地工作，无论是在线还是离线。系统在每个复制数据库里跟踪出版方和订阅方的出版对象变化的元数据，复制代理在复制对之间进行同步时把这些数据改变归并在一起，并通过自动冲突检测与消除来保证数据收敛。同步进程中使用的很多种冲突解决策略选项构建在复制代理中，定制的冲突消解方案可以通过使用存储过程或者可扩展的**组件对象模型**(COM)接口来编写。这种类型的复制并不复制所有中间状态，而只复制同步时的数据当前状态。当没有连接到任何网络时仍然需要副本具备自治更新的能力时，这种复制是合适的。

30.10 .NET 中的服务器编程

SQL Server 支持在 SQL Server 进程内托管 .NET 公共语言运行库(Common Language Runtime, CLR)，使得数据库程序员可以把业务逻辑编写为函数、存储过程、触发器、数据类型以及聚集。在数据库内部运行应用程序代码的能力为这类应用构架的设计增加了灵活性，它们要求业务逻辑靠近数据执行，且不能承受将数据转移到数据库外的中间过程来执行计算的代价。

.NET 公共语言运行库(CLR)是一种运行库环境，带有一种强类型的中间语言，可以运行如 C#、Visual Basic、C++、COBOL 和 J++以及其他的多种现代编程语言。支持内存的垃圾收集、抢占式线程、元数据服务(类型映射)、代码验证以及代码访问安全性。运行库使用元数据来定位和加载类，在内存中分配实例，处理方法调用，产生本地代码，强制安全性以及设置运行库上下文边界。

通过使用**汇编**(assembly)来将应用程序代码部署到数据库中，汇编是打包的单元，部署和 .NET 应用程序代码版本。将应用程序代码部署到数据库中为管理、备份和恢复整个数据库应用(代码和数据)提供了统一的方式。一只汇编注册到数据库中，用户可以通过使用可扩展性协议来利用 SQL DDL 语句将汇编中的入口点暴露出来。这些协议是定义好的并在这些 DDL 语句的执行中强制实施；这些 DDL 语句可以充当标量或者表函数、过程、触发器、类型和聚集。存储过程、触发器和函数通常需要执行 SQL 查询和更新，这是通过这样的组件来完成的，它在数据库进程内实现了所使用的 ADO.NET 数据
1253 访问 API。

30.10.1 .NET 基本概念

在 .NET 框架中，程序员用高级编程语言编写程序代码实现一个类，定义其结构(如域或类属性)和方法。有些这样的方法可以是静态函数。编译程序后产生一个文件，称为汇编(assembly)，它包含用**微软中间语言**(Microsoft Intermediate Language, MSIL)编译的代码，以及一个包含所有对依赖汇编引用的清单(manifest)。清单是每个汇编不可或缺的一部分，它使汇编能够自描述。汇编清单包含描述程序中定义的所有结构、域、属性、类、继承关系、函数和方法的汇编元数据。清单建立了汇编的标识，指定了构成汇编实现的文件，指定了构成汇编的类型和资源，详细说明了编译时对其他汇编的依赖，并指定了汇编正常运行所需要的权限集合。这些信息在运行时用来解决引用，执行版本绑定策略，并验证载入汇编的完整性。.NET 框架支持称为自定义属性(custom attribute)的带外(out-of-band)机制，用应用程序可能希望从元数据中探求的附加信息或方面来注释类、属性、函数和方法。所有 .NET 编译器处理这些注释时都不会解释它们，而是将它们存储到汇编的元数据中。所有这些注释可以和其他任何元数据一样，通过使用一组公共的映射 API 来进行检查。**托管代码**(managed code)指在 CLR 中而不是直接由操作系统执行的 MSIL。托管代码应用程序得到公共语言运行库服务，例如垃圾自动收集，

运行库类型检查以及安全性支持。这些服务帮助提供统一的、平台及语言无关的托管代码应用行为。在执行时，即时(Just-In-Time，JIT)编译器将 MSIL 翻译为本地代码(如 Intel X86 代码)。在此翻译过程中，代码必须通过一个验证过程，它检查 MSIL 和元数据以发现代码是否可以确定为类型安全的。

30.10.2 SQL CLR 宿主

SQL Server 和 CLR 是两种不同的运行库，具有不同的线程、调度和内存管理的内部模型。SQL Server 支持协作的、非抢占式线程模型，其中 DBMS 线程周期地或者在等待封锁或 I/O 时自动让出执行，而 CLR 支持抢占式线程模型。如果运行在 DBMS 内的用户代码可以直接调用操作系统(OS)的线程原语，那么它与 SQL Server 任务调度就集成得不好，并会降低系统的可扩展性。CLR 不区分虚拟内存和物理内存，但是 SQL Server 直接管理物理内存并要求在可配置的范围内使用物理内存。

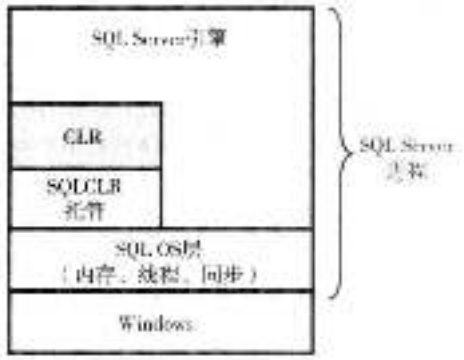


图 30-5 CLR 与 SQL Server 操作系统服务的集成

线程、调度和内存管理的不同模型为扩展 DBMS 以支持数千的并发用户会话提出了集成方面的挑战。当 CLR 寄居在 SQL Server 进程内时，SQL Server 通过模拟 CLR 的操作系统来解决这个问题。CLR 调用 SQL Server 实现的低层原语用于线程、调度、同步和内存管理(见图 30-5)。这种方法提供了下述可扩展性和可靠性优点：

公共线程、调度和同步(common threading, scheduling, and synchronization)。CLR 调用 SQL Server 的 API 既为运行用户代码也为其自身的内部使用(如垃圾收集和类的终结线程)而创建线程。为了在多个线程间实现同步，CLR 调用 SQL Server 同步对象。这使得当一个线程正在等待同步对象时，SQL Server 调度器能够调度其他任务。例如，当 CLR 开始收集垃圾时，它的所有线程都要等待垃圾收集完毕。由于 CLR 线程以及它们正在等待的同步对象对 SQL Server 调度器来说是已知的，SQL Server 调度器可以调度运行其他数据库任务的且不涉及 CLR 的线程。进而言之，这使得 SQL Server 能够检测到涉及 CLR 同步对象所持有封锁的死锁，并运用传统技术来消除死锁。SQL Server 调度器能够检测并停止很长时间没有让出的线程。将 CLR 线程与 SQL Server 线程挂钩的能力意味着 SQL Server 调度器能够识别出运行在 CLR 中的失控线程并管理它们的优先级，这样它们就不会消耗太多的 CPU 资源，从而影响系统的吞吐量。这些失控线程被并起并放回队列中。重复犯错的线程不会分配到对其他正在执行的工作线程来说不公平的时间片。如果犯错线程消耗了 50 倍被允许的时间片量，则在它被允许再次运行前将延后 50 次，因为调度器不能区分什么时候是计算过长并失控和什么时候是合法的长。

公共内存管理(common memory management)。CLR 调用 SQL Server 原语来分配和收回它的内存。由于 CLR 使用的内存是占有系统使用的总内存的，SQL Server 可以保持在其配置的内存范围内并保证 CLR 和 SQL Server 不会互相竞争内存。而且，当系统资源受限时，SQL Server 可以拒绝 CLR 的内存请求，并当其他任务需要内存时要求 CLR 减少其内存使用。

1254
↓
1255

30.10.3 可扩展性协定

所有运行在 SQL Server 进程内的用户托管的代码以扩展的形式与 DBMS 组件交互。当前的扩展包括标量函数、表函数、过程、触发器、标量类型和标量聚集。对于每一项扩展都有一个共同的协定来定义用户代码必须实现的属性或者服务，以便作为这些扩展之一，以及当调用托管代码时该扩展能从 DBMS 获得的服务。SQL CLR 利用类和存储在汇编元数据中的定制属性信息来强制用户代码实现这些可扩展性协定。所有用户汇编存储在数据库中。所有关系和汇编元数据在 SQL 引擎内通过一组统一的接口和数据结构来处理。当数据定义语言(DDL)语句所注册的一个特定的扩展函数、类型或聚集被处理时，系统通过分析其汇编元数据来确保用户代码实现相应的协定。如果协定实现了，那么 DDL 语句就成功，否则就失败。下一小节描述 SQL Server 当前执行的特定协定中的关键方面。

30.10.3.1 例程

一般地，我们将标量函数、过程和触发器归为例程。例程作为静态类函数实现，可以通过定制属性指定如下属性：

- **IsPrecise**。如果这个布尔属性为 **false**，则意味着例程体涉及不精确计算，如浮点操作。涉及不精确函数的表达式不能被索引。
- **UserDataAccess**。如果该属性的值为 **read**，那么例程读用户数据表。否则该属性的值为 *None*，表示例程不访问数据。不访问任何用户表的查询（直接地或通过视图和函数间接地）被认为不访问用户数据。
- **SystemDataAccess**。如果该属性的值为 **read**，那么例程读系统目录或虚拟系统表。
- **IsDeterministic**。如果该属性的值为 **true**，那么给定相同的输入值、本地数据库状态和执行的上下文环境，认为例程产生相同的输出值。
- **IsSystemVerified**。它指示确定性和精确性属性是否由 SQL Server 确认或执行（如内置插件、事务 SQL 函数），或者由用户指定（如 CLR 函数）。

1256

- **HasExternalAccess**。如果该属性的值为 true，那么例程访问 SQL Server 之外的资源，如文件、网络、Web 访问和注册表。

30.10.3.2 表函数

实现表值函数的类必须实现能在函数返回的行上进行循环的 IEnumerable 接口，描述返回表的模式（例如，列、类型）的方法，描述哪些列可以为唯一性码的方法，以及将行插入到表中的方法。

30.10.3.3 类型

实现用户自定义类型的类用 SqlUserDefinedType() 属性注释，该属性指定如下特性：

- **Format**。SQL Server 支持三种存储格式：本机的、用户自定义的和 .NET 序列化的。
- **MaxByteSize**。这是以字节为单位的类型实例的序列化二进制表示的最大大小。UDT 在长度上可以多达 2GB。
- **IsFixedLength**。这个布尔性质指定类型的实例是定长的还是变长的。
- **IsByteOrdered**。这个布尔性质指定类型实例的序列化二进制表示是否是二进制有序的。如果此属性为 true，系统可以直接在这种表示上执行比较而无需将类型实例实例化为对象。
- **Nullability**。所有系统内的 UDT 必须能通过支持包含布尔型 IsNull 方法的 INullable 接口来支持 *null* 值。
- **Type conversions**。所有 UDT 必须通过 ToString 和 Parse 方法实现与字符串的相互转换。

30.10.3.4 聚集

除了支持类型的特定，用户自定义聚集必须实现查询执行引擎要求的 4 种方法来初始化聚集实例的计算，将输入值累加到聚集提供的函数中，将聚集的部分计算合并，并检索最终聚集结果。聚集可以通过定制属性在它们的类定义中声明附加属性，这些属性被查询优化器用来为聚集计算导出替代的计划。

- **IsInvariantToDuplicates**。如果此属性为 true，那么将数据传给聚集的计算可以通过丢弃或引入新的去重操作来进行修改。

1257

- **IsInvariantToNulls**。如果此属性为 true，那么 null 行可以从输入中丢弃。然而，在 **group by** 操作的情况下必须注意不要丢弃整个组。
- **IsInvariantToOrder**。如果此属性为 true，那么查询处理器可以忽略 **order by** 子句并探索避免必须对数据排序的计划。

30.11 XML 支持

近年来关系数据库系统以很多不同的方式支持 XML。关系数据库系统中的第一代 XML 支持主要关心的是将关系数据导出为 XML（“发布 XML”），并将 XML 标记格式的关系数据导入回关系表示中（“分解 XML”）。这些系统所支持的主要应用场景是如下环境中的信息交换：其中 XML 用作“传输格式”，

并且关系模式和 XML 模式通常是相互独立地预定义的。为了适用于这种场景，微软 SQL Server 提供了扩展功能，比如针对 **XML**(for xml)发布的行集聚集器、OpenXML 行集提供程序以及基于行注解模式的 XML 视图技术。

为了存储结构可能随时间变化的半结构化数据以及存储文档而将 XML 数据分解成关系模式可能会相当困难或者低效。为了支持这样的应用，SQL Server 实现了基于 SQL：2003 标准中的 **xml** 数据类型的本地 XML。图 30-6 给出了在数据库中 SQL Server 的本地 XML 支持的高层体系图。它包括本地存储 XML 的能力，用 XML 模式集合对所存储的 XML 数据进行约束和标定类型的能力，以及查询和更新 XML 数据的能力。为了提供高效的查询执行，它还提供了几种 XML 专用索引类型。最后，本地的 XML 支持还集成了"分解"为关系数据以及从关系数据"发布"的功能。

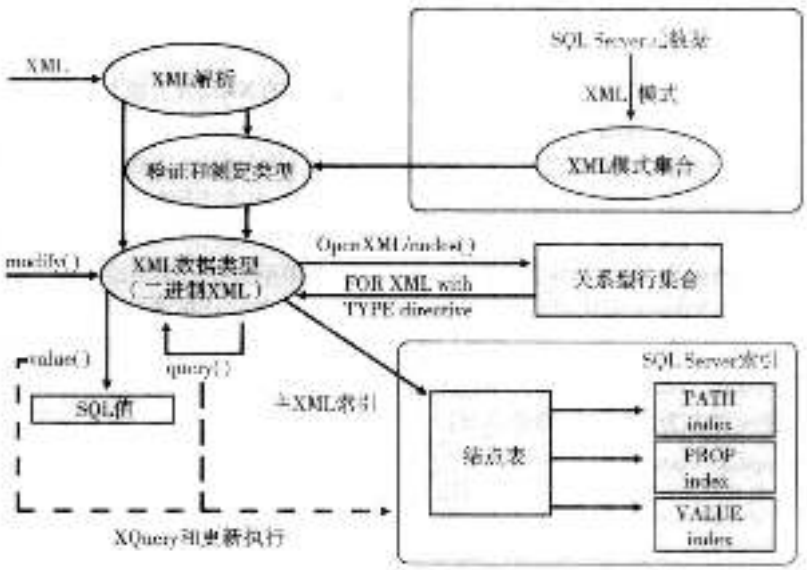


图 30-6 SQL Server 中本地 XML 支持的体系概览

30.11.1 本地存储和组织 XML

xml 数据类型可以存储 XML 文档和内容片断(多个文本或者顶部的元素结点)，并且是在 XQuery 1.0/XPath 2.0 数据模型的基础上定义的。该数据类型可用作存储过程的参数、变量以及列类型。

SQL Server 将 **xml** 类型的数据存储在作为 **blob** 的内部二进制格式中，并且为执行查询提供索引机制。该内部二进制格式提供了对原始 XML 文档的高效检索和重构，以及一定的空间节省(平均有 20%)。索引支持高效的查询机制，它可以利用关系查询引擎和优化器；更多细节将在后面 30.11.3 节中介绍。

SQL Server 提供了称为 **XML 模式集合**(XML schema collection)的数据库元数据概念。它关联一个 SQL 标识器和一个或多个目标命名空间的模式组件集合。

30.11.2 查询和更新 XML 数据类型

SQL Server 在 XML 数据类型上提供了几种基于 XQuery 的查询和修改能力。这些查询和修改能力通过使用定义在 **xml** 数据类型上的方法来支持。本节的其余部分将描述一些这样的方法。

每种方法将一个文字串作为查询串以及潜在的其他变元。XML 数据类型(该方法应用在其上)为路径表达式提供了上下文项，并用相关 XML 模式集合(如果没有提供集合，则该 XML 数据就被认为是未标定类型的)提供的所有类型信息构成范围内的模式定义。SQL Server 的 XQuery 实现是静态标定类型的，因此支持早期检测路径表达式的拼写错误、类型错误和基数不匹配的检查，以及某些附加的优化。

query 方法接收一个 XQuery 表达式并返回一个未标定类型的 XML 数据类型实例(如果需要标定该数据的类型，它可以转换为一个目标模式集合)。在 XQuery 规范术语中，我们设定构造模式为"strip"。

下面的例子展示了一个简单 XQuery 表达式，它摘抓了一份差旅报告文档中的复杂 Customer 元素，除了
1258 别的信息以外，该文档还包含一个名称、一个 ID 属性和销售指导信息，这些信息包含在标记的实际差
1259 旅报告注释中。这份概要针对具有销售指导的 Customer 元素展示了名称和销售指导：

```
select Report.query('
  declare namespace c = "um:example/customer";
    for $cust in /c:doc/c:customer
    where $cust/c:notes//c:salesLead
    return
      <customer id="{$cust/@id}">
        {$cust/c:name,
        $cust/c:notes//c:salesLead}
      </customer>')
from TripReports;
```

上述 XQuery 查询在存储于表 *TripReports* 的每行上的 *doc* 属性中的 XML 值上执行。该 SQL 查询结果中的每行包含了在一个输入行中的数据上执行该 XQuery 查询的结果。

value 方法接收一个 XQuery 表达式和一个 SQL 类型名称，从 XQuery 表达式的结果中抽取出单个的原子值，并将其词法形式转换为指定的 SQL 类型。如果 XQuery 表达式的结果为一个结点，则该结点的已标定类型的值将会隐式地抽取为一个原子值，然后转换为 SQL 类型（用 XQuery 的术语来说，该结点将被"原子化"；结果转换为 SQL）。注意 *value* 方法执行静态类型检查，至多返回一个值。

exist 方法接收一个 XQuery 表达式，如果该表达式产生一个非空结果则返回 1，否则返回 0。

最后，*modify* 方法提供了一种机制在子树层改变 XML 值，在树中指定位置插入一棵新子树，改变一个元素或者属性的值，以及删除子树。下面的例子删除了某年份以前的所有顾客的 salesLead 元素，该年份由 SQL 变量或者名为@year 的参数给出：

```
update TripReports
set Report.modify('
declare namespace c = "um:example/customer";
delete /c:doc/c:customer//c:salesLead[@year < sql:variable("@year")]')
```

30.11.3 XQuery 表达式的执行

正如前面所提到的，XML 数据以内部的二进制表示存储。但是，为了执行 XQuery 表达式，XML 数据类型在内部转换为一种称为结点表的形式。该内部结点表基本上使用行来表示结点。每个结点接受一个 OrdPath 标识符作为它的 nodeID（OrdPath 标识符是修改过的杜威（Dewey）十进制编号计划；关于 OrdPath 的更多信息请参见文献注解中的文献）。每个结点还包含码信息来指回到该结点所属的原始 SQL 行，有关名字和类型（以标记化的形式）的信息，值，以及其他信息。由于 OrdPath 将文档顺序和
1260 层次信息都进行了编码，因此结点表在码信息和 OrdPath 的基础上聚集，从而路径表达式或者子树重组都能利用简单的表扫描完成。

所有 XQuery 和更新表达式随后都翻译为这个内部结点表上的一棵代数操作运算符树。该树使用常用的关系运算符和一些专门为 XQuery 代数设计的运算符。然后结果树被嫁接至关系表达式的代数树中。因此最后查询执行引擎接收到单棵执行树，并且它能对其优化和执行。为了避免昂贵的运行时转换，用户可以通过采用主 XML 索引来预先物化结点表。SQL Server 进一步提供了二种 XML 辅助索引以使查询执行能够更多地利用索引结构：

- *path* 索引提供了对路径表达式简单类型的支持。
- *properties* 索引提供了对通常情况下的性质值比较的支持。
- *value* 索引非常适用于在比较中使用通配符的查询。

关于 SQL Server 中的 XML 索引和查询处理的更多信息请参见文献注解中的文献。

30.12 SQL Server 服务代理

通过在 SQL Server 中提供队排队、可靠消息的支持，服务代理帮助开发人员创建松耦合的分布式应

用。许多数据库应用使用异步处理来提高可扩展性和对交互型会话的响应时间。常用的异步处理方法是使用工作表。与在单个数据库事务中执行业务流程的所有工作不同，一个应用程序进行登录来去明当前未完成的工作。然后向工作表中插入一条待执行工作的记录。只要资源允许，应用程序处理工作表并完成业务流程。服务代理是数据库服务器的一部分，它直接支持该方法用于应用开发。Transact-SQL语言包括针对服务代理的 DDL 和 DML 语句。另外，SQL Server 中为服务代理提供了 SQL Server 管理对象（SQL Server Management Object，SMO）。这些允许从托管代码来编程访问服务代理对象。

以前的消息排队技术集中于单条消息。服务代理的基本通信单元是会话（conversation）——一个持久的、可靠的、全双工的消息流。SQL Server 保证一个会话中的消息按序刚好向一个应用分发一次。也可能指派一个从 1 到 10 的优先级给一个会话。来自较高优先级会话中的消息发送和接收要快于来自较低优先级会话的消息。会话发生在两个服务之间。服务是会话的命名端点。每个会话是一个会话组的一部分。相关会话可以与同一个会话组关联。 1261

消息是强标定类型的，例如，每条消息都有特定类型。SQL Server 可以选择性地验证消息是否是格式良好的 XML，消息是否为空，或者消息是否遵循某个 XML 模式。协定（contract）定义了会话所允许的消息类型，以及哪些会话参与者能够发送该类型的消息。SQL Server 为只需要可靠流的应用提供默认的协定和消息类型。

SQL Server 将消息存储在内部表中。这些表并不能直接访问，而是由 SQL Server 给出队列（queue）作为这些内部表的视图。应用从一个队列接收消息。一个接收操作返回来自同一个会话组的一个或多个消息。通过控制对底层表的访问，SQL Server 能够有效地执行消息排序、相关消息关联以及封锁。由于队列是内部表，它们不需要为备份、恢复、故障转移或者数据库镜像做特殊处理。应用表和关联的排队消息都由数据库进行备份、恢复和故障转移。当镜像故障转移完成时，存在于镜像数据库中的代理会话会在停止点继续执行——即使该会话发生在两个位于不同数据库中的服务之间。

服务代理操作的封锁粒度是会话组，而不是特定的会话或者单个消息。通过在会话组上施加封锁，服务代理自动帮助应用避免处理消息中的并发问题。当一个队列包含多个会话时，SQL Server 保证同一时间只有一个队列阅读者可以处理属于一个给定会话组的消息。这就消除了应用自身包含死锁避免逻辑的需要——这是在许多消息应用中常见的错误来源。该封锁语义的另一个好的副作用是应用可以选择使用会话标识作为存储和检索应用状态的键。相对于在传统消息排队系统中发现的原子消息原语，决定把会话形式化为通信原语带来了许多优点。上述编程模型的好处仅仅是这些优点的两个例子。

当一个队列包含了需要处理的消息时，SQL Server 能够自动激活存储过程。为了按照到达的通信量来调整运行的存储过程数量，激活逻辑监视队列以查看是否存在对另一个队列阅读者有用的工作。SQL Server 同时考虑已经存在的阅读者接收消息的速率以及可用的会话组数量来决定什么时候启动另一个队列阅读者。被激活的存储过程、存储过程的安全环境以及将启动的实例的最大数量都为单独的队列而配置。SQL Server 还提供了一个外部激活器（external activator）。当新的消息插入到队列中时，这个特性允许 SQL Server 之外的应用被激活。然后该应用可以接收和处理消息。借助于这样的做法，CPU 密集型工作可以卸载给 SQL Server 之外的应用，可能在一台不同的计算机上。并且，长周期任务（如：Web 服务调用）可以在阻塞数据库资源的情况下执行。外部激活器遵循与内部激活相同的逻辑，当消息积聚在一个队列中时，可以配置为激活一个应用的多个实例。 1262

作为对实例中异步消息的逻辑扩展，服务代理还提供了 SQL Server 实例间的可靠消息，允许开发人员轻松地建立分布式应用。会话能够出现在单个 SQL Server 实例中，或者出现在两个 SQL Server 实例之间。本地和远程会话使用同样的编程模型。

安全性和路由都是声明式配置的，不需要改变队列阅读者。SQL Server 使用路由（route）将服务名映射为会话中另一个参与者的网络地址。SQL Server 还能够为会话执行消息转发和简单的负载均衡。SQL Server 提供可靠的、恰好一次的按序分发，不论一个消息要经过多少个实例。跨越多个 SQL Server 实例的会话能够在网络层（点到点）和会话层（端到端）得到保护。当使用端到端的安全性时，直到消息到达最终目的地之前，消息内容都保持加密状态，而消息头对于消息所经过的每个 SQL Server 实例都是可

同的。在实例内部应用标准 SQL Server 权限。加密发生在消息离开实例的时候。

SQL Server 使用一种二进制协议在实例间发送消息。该协议将大消息拆成片断并允许来自多个消息的片段交叉。片断化使得 SQL Server 能够快速传递较小的消息，即使在一个大消息正在传输过程中的情况下。该二进制协议不使用分布式事务或者两阶段提交。相反，该协议要求接收者确认消息片断。SQL Server 简单地、重期性地重发消息片断，直到该片断被接收者确认。确认大都包含在回复消息头部，尽管在没有回复消息可用时会使用专用的回复消息。

SQL Server 包含了一个命令行诊断工具(*sdiagnose*)来帮助分析服务代理的部署并调查问题。这个工具可以在配置模式或运行时模式运行。在配置模式下，此工具检查一对服务是否可以交换消息并返回任何配置错误。这些错误的例子是：失效的队列和缺失的返回路由。在第二种模式下，此工具连接到两个或更多的 SQL Server 实例并监控 SQL 跟踪器事件，以发现运行时的服务代理问题。此工具的输出可以发送给文件以便自动处理。

30.13 商务智能

SQL Server 的商务智能组件包含三个子组件：

- SQL Server 集成服务(SQL Server Integration Service，SSIS)，它提供了从多源集成数据、执行与清洗数据并将数据转换为公共形式相关的转换，以及将数据加载到数据库系统的方法。

[1263]

- SQL Server 分析服务(SQL Server Analysis Service，SSAS)，它提供了 OLAP 和数据挖掘能力。
- SQL Server 报表服务(SQL Server Reporting Service，SSRS)。

集成服务、分析服务和报表服务各自在独立的服务器中实现，并且能彼此独立地安装在相同或不同的机器上。它们能通过本地连接、OLE-DB 或者 ODBC 驱动连接到各种数据源，比如平面文件、电子表格或者各种关系数据库系统。

它们一起提供了一个端到端的解决方案，进行抽取、转换和加载数据，然后对数据建模并添加分析功能，最后建立和分发数据上的报表。不同的 SQL Server 商务智能组件能够集成并相互辅助。这里有几个常见的利用组件组合的场景：

- 建立清洗数据的 SSIS 包，使用 SSAS 数据挖掘生成的模式。
- 使用 SSIS 加载数据到 SSAS 立方体，处理它，并且在 SSAS 立方体上执行报表。
- 建立 SSRS 报表来发布挖掘模型的发现，或者在 SSAS OLAP 组件中包含的数据。

下面的小节给出了每个这样的服务器组件的功能和体系结构的概览。

30.13.1 SQL Server 集成服务

微软 SQL Server 集成服务(SQL Server Integration Service，SSIS)是企业数据转换和数据集成的解决方案，可以用它从相异的源中抽取、转换、聚集和合并数据，并把它转移到单个或多个目的地。可以使用 SSIS 来执行以下任务：

- 从异构的数据存储中合并数据。
- 刷新数据仓库和数据集市中的数据。
- 在数据加载到目的地之前清洗数据。
- 批量加载数据到联机事务处理(OLTP)和联机分析处理(OLAP)数据库中。
- 发送通知。
- 建立商务智能到数据转换处理中。
- 自动管理功能。

[1264]

SSIS 为上述任务类别了一组完整的服务、图形化工具、可编程对象以及 API。这些提供创建大量的、健壮的和复杂的数据转换解决方案的能力，而无需任何定制编程。但是，API 和可编程对象可以在需要的时候用来创建定制元素或者将数据转换功能集成到定制应用中。

SSIS 数据流引擎类别了内存缓冲区，用于将数据从源移动到目的地，并调用从文件和关系数据库中抽取数据的源适配器。该引擎还提供修改数据的转换以及将数据加载到数据存储中的目标适配器。基于模糊(近似)匹配的去重是 SSIS 提供的转换的一个例子。如果需要的话，用户可以编写他们自己的

转换。图 30-7 显示了一个如何组合多种转换来清洗和加载图书销售信息的例子：来自销售数据的图书标题在出版数据库上进行匹配。如果没有匹配，就执行模糊查找来处理有小错误（比如拼写错误）的标题。有关可信赖数据来源的信息与清洗过的数据存储在一起。

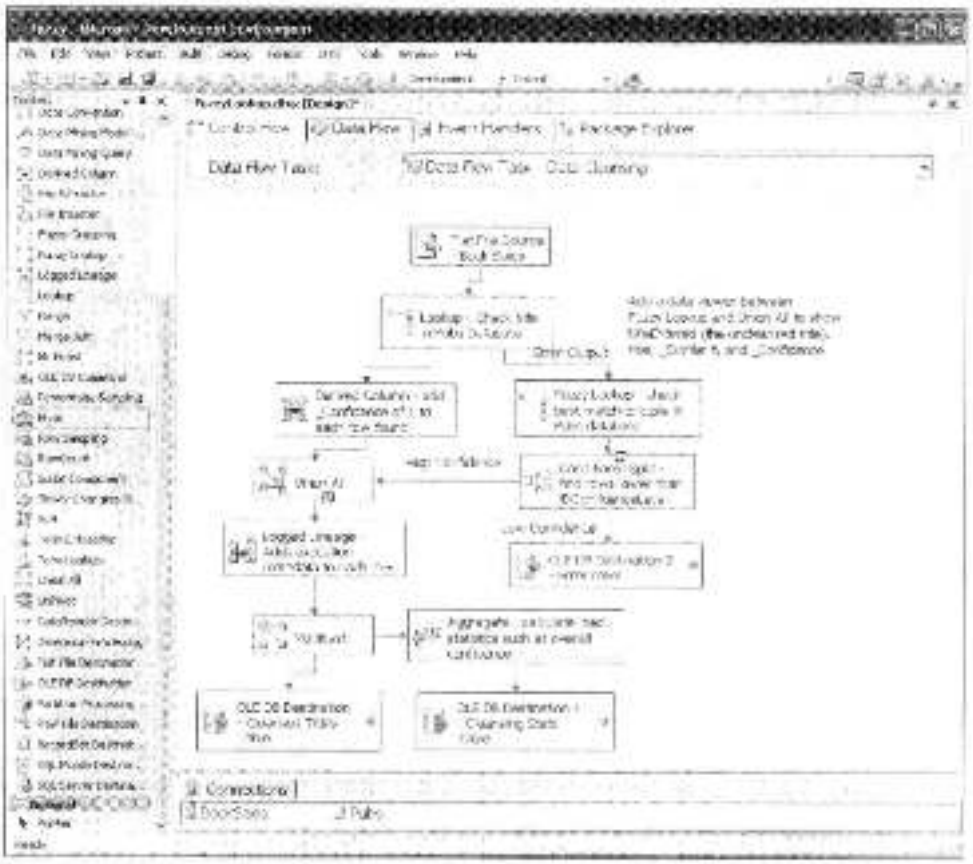


图 30-7 使用模糊查找的数据加载

30.13.2 SQL Server 分析服务

分析服务组件为商务智能应用提供联机分析处理（OLAP）和数据挖掘功能。分析服务支持瘦客户端体系架构。计算引擎在服务器端，因此查询在服务器端处理，避免了需要在客户端和服务器之间传输大量数据。

30.13.2.1 SQL Server 分析服务：OLAP

Analysis Server 实用统一维度模型（Unified Dimensional Model，UDM），它弥补了传统关系报表和 OLAP 即席分析之间的差异。统一维度模型（UDM）的角色是提供用户和数据源之间的桥梁。UDM 创建在一个或多个物理数据源之上，然后最终用户实用各种客户端工具之一（例如 Microsoft Excel）在 UDM 上发布查询。

并非只有 DataSource 模式的维度建模层，UDM 为定义强大而详尽的业务逻辑、规则以及语义定义提供了丰富的环境。通过定义元数据目录的本地语言翻译以及维度数据，用户能够在 UDM 数据上实用他们的本地语言（比如法语或者印地语）来浏览和生成报表。分析服务定义了复杂时间维（财务的、报告的、生产的，等等），并能够使用多维表达式（MDX）语言来定义强大的多维业务逻辑（年增长、年度至今）。UDM 允许用户定义面向业务的视角，每个视角仅表示模型的一个特定子集（度量、维、属性、业务规则，等等），该子集与一个特定的用户组相关。商务智能定义关键性能指标（Key Performance Indicator，KPI），它是用一些量业务健康程度的重要指标。这类 KPI 的例子包括销售额，

每员工收入以及客户保有率。UDM 允许定义这样的 KPI，使得能以更容易理解的方式来分组和显示数据。

30.13.2.2 SQL Server 分析服务：数据挖掘

SQL Server 提供各种挖掘技术，并有丰富的图形化界面来查看挖掘结果。所支持的挖掘算法包括：

- 关联规则（在交叉销售应用中很有用）。
- 分类和预测技术，比如决策树、回归树、神经网络和朴素贝叶斯。
- 时间序列预测技术，包括 ARIMA 和 ARTXP。
- 聚类技术，比如最大期望和 k-means（与序列聚类技术配合）。

1265
1266

另外，SQL Server 提供了可扩展的体系架构来插入第三方数据挖掘算法和可视化工具。

SQL Server 还支持数据挖掘扩展（Data-Mining Extensions，DMX）这种对 SQL 的扩展。DMX 是用来与数据挖掘模型交互的语言。就像 SQL 是用来与表和视图交互的一样。通过 DMX，可以创建和训练模型并将其存储在分析服务数据库中。然后，可以浏览模型以考虑模式，或者通过使用特殊的 **prediction join** 语法，将模型应用到新的数据上以执行预测。DMX 语言支持一些功能和构造，来容易地确定一个预测的类及其置信度，就像在推荐引擎中那样来预测相关项的列表，或者甚至返回关于一个预测的信息和支持事实。SQL Server 中的数据挖掘能够用于存储在关系或多维数据源中的数据上。通过特定的任务和转换，也能够支持其他数据源，允许在集成服务的操作员数据流水线上直接进行数据挖掘。数据挖掘结果可以通过图形化控制，OLAP 立方体上特殊的数据挖掘维或者简单地用报表服务中的报表来显现。

30.13.3 SQL Server 报表服务

报表服务是一个基于服务器的报表平台，可用来创建和管理包含来自关系和多维数据源的数据的表格式、矩阵式、图形化和自由格式的报表。创建的报表能够通过基于 Web 的连接来查看和管理。矩阵式报表可以为高层次的查看汇总数据，同时提供下钻报表中对细节的支持。参数化报表可用于基于运行时提供的值来过滤数据。用户可以从各种查看格式中选择喜欢的格式来在运行中提供报表，用于数据浏览或打印。还可以用 API 来扩展或将报表功能集成到定制的解决方案中。基于服务器的报表提供了一种方式用于集中报表的存储和管理，设置对报表和文件夹的策略和安全访问，控制报表如何处理和分发，以及报表如何在业务中使用的标准化。

文献注解

关于使用 C2 认证系统的 SQL Server 详细信息可以在 www.microsoft.com/Downloads/Release.asp?ReleaseID=25503 上获得。

SQL Server 的优化框架是基于 Graefe[1995] 提出的级联优化器原型的。Simmen 等[1996]对减少分组列的方案进行了讨论。Galindo-Legaria 和 Joshi[2001] 与 Elhemali 等[2007]提出了各种执行策略。SQL Server 在
1262 了各项的基于代价的优化中考虑了这些策略。Chaudhuri 等[1999]论述了关于 SQL Server 自调优方面的其他附加信息。Chaudhuri 和 Shim[1994] 与 Yan 和 Larson[1995] 论述了聚集运算的重排序。

Chatziantoniou 和 Ross[1997] 与 Galindo-Legaria 和 Joshi[2001] 提出了 SQL Server 用于请求当连接的 SQL 查询的一种替代方案。在该方案下，优化器检测模式并考虑每段执行。Pellenkoft 等[1997]论述了使用完全的、本地的和非冗余的转换集合来产生完整搜索空间的优化方案。Graefe 等[1998]讨论了支持基本的聚集、重读、有些优化、扩展以及关于数据倾斜的动态调整的散列操作。Graefe 等[1998]仅为了用查询所需要的列集合来装配行的目的而提出联合索引的思想。这种观点认为，该方法有时候比扫描基表要快。

Blakeley[1996] 与 Blakeley 和 Pizzo[2001]提供了通过 OLE-DB 与存储引擎通信相关的讨论。Blakeley 等[2005]详细描述了 SQL Server 的分布和异构查询功能的实现。Acheson 等[2004]提供了在 SQL Server 进程中集成 .NET CLR 的细节。

Blakeley 等[2008]更详细地描述了 UDT、UDAgg 和 UDF 的特定。Blakeley 等[2006]介绍了 ADO.NET 实体框架。Melnik 等[2007]描述了 ADO.NET 实体框架后面的映射技术。Ady 等[2007]提供了 ADO.NET 实体框架体系架构的概述。SQL：2003 标准在 SQL/XML[2004] 中定义。Rys[2001]提供了关于 SQL Server 2000

XML 功能的更多细节。Rys[2004]提供了对 **for xml** 聚集扩展的概述。关于能够在客户端或 CLR 中使用的 XML 功能的信息，请参考 http://msdn.microsoft.com/XML/Building-XML/XMLandDatabase/default.aspx 上的白皮书集。XQuery 1.0/XPath 2.0 数据模型在 Walsh 等[2007]中定义。Rys[2005]提供了在关系数据库环境中实现 XQuery 的技术的概述。OrdPath 计数方案在 O'Neil 等[2004]中讲述；Pal 等[2004]和 Baras 等[2005]提供了更多关于 SQL Server 2005 中的 XML 索引和 XQuery 代数化和优化方面的信息。 1268

第十部分
Part 10

附　录

附录A给出了用作我们的运行实例的大学数据库的全部细节内容，包括E-R图、SQL DDL和简单全书的示例数据。（DDL和示例数据可从本书网站db-book.com下载，作为实验性练习之用。）

其余的附录并没有包含在本书的印刷本中，但可以通过本书网站db-book.com在线下载。它们包括：

- 附录B（高级关系数据库设计），首先介绍了多值依赖的理论；回想一下，多值依赖是在第8章介绍的。接下来阐述了投影－连接范式，该范式基于一种叫做连接依赖的约束形式；连接依赖是多值依赖的一种泛化形式。该章以给出域码范式的另一种范式结束。
- 附录C（其他关系查询语言）首先给出了关系查询语言——基于样例的查询（QBE），这种语言是为非程序员的使用而设计的。在QBE中，查询看起来就像一组表，这些表中包含待检索数据的样例。随后介绍了微软Access的基于QBE的图形化查询语言。接着是Datalog语言，其语法仿照了逻辑编程语言Prolog。
- 附录D（网状模型）和附录E（层次模型）介绍了网状和层次数据模型。这两种数据模型产生于关系模型之前，并提供了比关系模型更低层次的抽象。它们抽象掉一些用于在磁盘上存储数据的实际数据结构的一些细节，但不是所有的细节。这些模型仅在少数遗留应用中使用。

从附录B到附录E，我们用一个具有如图2-15所示的模式的银行企业来阐述我们的概念。

附录 A |
Database System Concepts, 5E

详细的大学模式

在本附录中，将给出我们的运行实例大学数据库的全部详细内容。在 A.1 节我们介绍本书使用的完整模式，以及该模式对应的 E-R 图。在 A.2 节我们介绍运行大学实例的相对完整的 SQL 数据定义。除了列出每个属性的数据类型，我们还引入了大量约束。最后，我们在 A.3 节给出了对应于我们的模式的示例数据。从本书的网站 db-book.com 上可以获取用于创建模式中所有关系并向其中导入示例数据的 SQL 脚本。

A.1 完整模式

图 A-1 显示了本书使用的大学数据库的完整模式。图 A-2 显示了对应于该模式的 E-R 图，该图例穿个书。

```
classroom(building, room_number, capacity)
department(dept_name, building, budget)
course(course_id, title, dept_name, credits)
instructor(ID, name, dept_name, salary)
section(course_id, sec_id, semester, year, building, room_number, time_slot_id)
teaches(ID, course_id, sec_id, semester, year)
student(ID, name, dept_name, tot_cred)
takes(ID, course_id, sec_id, semester, year, grade)
advisor(s_ID, i_ID)
time_slot(time_slot_id, day, start_time, end_time)
prereq(course_id, prereq_id)
```

图 A-1 大学数据库的模式

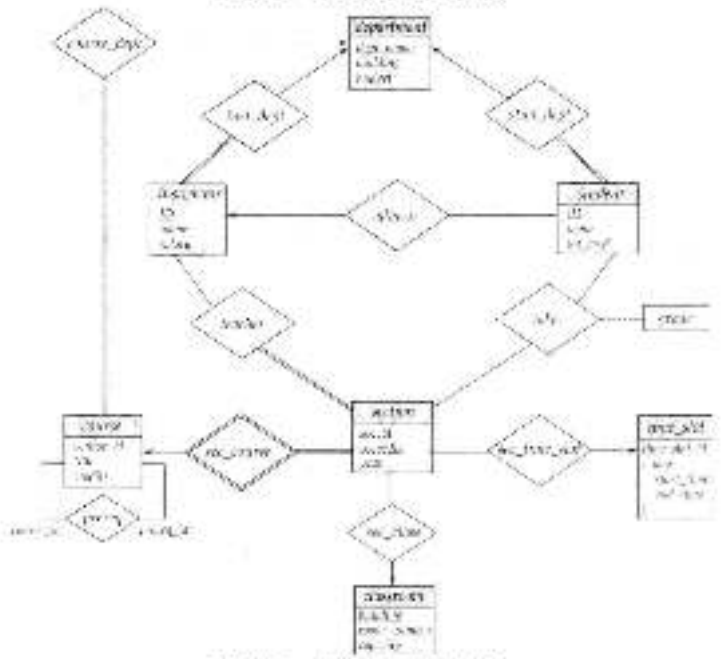


图 A-2 大学企业的 E-R 图

A.2 DDL

在本节中，给出关于我们的例子的相对完整的 SQL 数据定义。除了列出每个属性的数据类型，我们还引入了大量的约束。

```
create table classroom
  (building        varchar (15),
   room_number     varchar (7),
   capacity        numeric (4, 0),
   primary key (building, room_number));

create table department
  (dept_name       varchar (20),
   building        varchar (15),
   budget          numeric (12, 2) check (budget > 0),
   primary key (dept_name));

create table course
  (course_id       varchar (8),
   title           varchar (50),
   dept_name       varchar (20),
   credits         numeric (2, 0) check (credits > 0),
   primary key (course_id),
   foreign key (dept_name) references department
                   on delete set null);

create table instructor
  (ID              varchar (5),
   name            varchar (20) not null,
   dept_name       varchar (20),
   salary          numeric (8, 2) check (salary > 29000),
   primary key (ID),
   foreign key (dept_name) references department
                   on delete set null);

create table section
  (course_id       varchar (8),
   sec_id          varchar (8),
   semester        varchar (6) check (semester in
   ('Fall', 'Winter', 'Spring', 'Summer')),
   year            numeric (4, 0) check (year > 1701 and year < 2100),
   building        varchar (15),
   room_number     varchar (7),
   time_slot_id    varchar (4),
   primary key (course_id, sec_id, semester, year),
   foreign key (course_id) references course
                   on delete cascade,
   foreign key (building, room_number) references classroom
                   on delete set null);
```

在上述 DDL 中，如果元组的存在取决于引用的元组，我们为外码约束增加了 **on delete cascade** 声明。例如我们为从 *section*（根据前实体 *section* 产生）到 *course*（该前实体的标识性联系）的外码约束添加了 **on delete cascade** 声明。在其他外码约束中我们要么指定为 **on delete set null**（它通过把引用值设为空的方式来删除引用元组），要么不增加任何声明（它防止删除任何引用元组）。例如，如果删除一个系，我们不希望把相关的教师也删除；从 *instructor* 到 *department* 的外码约束实际上把 *dept_name* 属性设置为空。另一方面，关系 *prereq*（将在下面给出）的外码约束防止删除这样的课程：该课程作为另一门课程的先修课。对于下面将给出的 *advisor* 关系，如果删除一位教师，我们允许把 *i_ID* 设置为空，但是如果删除被引用的学生，那么就删掉 *advisor* 元组。

```
create table teaches
  (ID             varchar (5),
   course_id      varchar (8),
   sec_id         varchar (8),
   semester       varchar (6),
   year           numeric (4, 0),
   primary key (ID, course_id, sec_id, semester, year),
   foreign key (course_id, sec_id, semester, year) references section
                  on delete cascade,
   foreign key (ID) references instructor
                  on delete cascade);

create table student
  (ID              varchar (5),
   name            varchar (20) not null,
   dept_name       varchar (20),
   tot_cred        numeric (3, 0) check (tot_cred >= 0),
   primary key (ID),
   foreign key (dept_name) references department
                  on delete set null);

create table takes
  (ID              varchar (5),
   course_id       varchar (8),
   sec_id          varchar (8),
   semester        varchar (6),
   year            numeric (4, 0),
   grade           varchar (2),
   primary key (ID, course_id, sec_id, semester, year),
   foreign key (course_id, sec_id, semester, year) references section
                  on delete cascade,
foreign key (ID) references student
                  on delete cascade);

create table advisor
  (s_ID            varchar (5),
   i_ID            varchar (5),
   primary key (s_ID),
   foreign key (i_ID) references instructor (ID)
                  on delete set null,
foreign key (s_ID) references student (ID)
                  on delete cascade);

create table prereq
  (course_id       varchar(8),
   prereq_id       varchar(8),
   primary key (course_id, prereq_id),
   foreign key (course_id) references course
                  on delete cascade,
   foreign key (prereq_id) references course);
```

下面对表 *time_slot* 的 **create table** 语句可以在大多数数据库系统上运行，但不能在 Oracle 上工作（至少不适用于 Oracle 11 版本），因为 Oracle 不支持 SQL 标准类型 **time**。

```
create table time_slot
  (time_slot_id    varchar (4),
   day             varchar (1) check (day in ('M','T','W','R','F','S','U')),
   start_time      time,
   end_time        time,
   primary key (time_slot_id, day, start_time));
```

这些例子说明了在 SQL 中指定时间的语法：'08：30'、'13：55' 和 '5：30PM'。由于 Oracle 不

支持 **time** 类型，对于 Oracle 我们使用如下模式来代替：

```
create table time_slot
    (time_slot_id       varchar (4),
     day                varchar (1),
     start_hr           numeric (2) check (start_hr >= 0 and start_hr < 24),
     start_min          numeric (2) check (start_min >= 0 and start_min < 60),
     end_hr             numeric (2) check (end_hr >= 0 and end_hr < 24),
     end_min            numeric (2) check (end_min >= 0 and end_min < 60),
     primary key (time_slot_id, day, start_hr, start_min));
```

其区别在于：*start_time* 被替换成两个属性 *start_hr* 和 *start_min*，同样 *end_time* 被替换成属性 *end_hr* 和 *end_min*。这些属性上还有约束来保证只有代表有效时间值的数字才能出现在这些属性中。这个版本的 *time_slot* 模式在包括 Oracle 在内的所有数据库上都能运行。注意尽管 Oracle 支持 **datetime** 数据类型，但 **datetime** 包括特定的日、月、年以及时间，用在这里并不合适，因为我们只想用时间。把时间属性拆分成小时和分钟部分还有两种替代方案，但是它们都不可取。第一种替代方案是采用 **varchar** 类型，但是这样就难以在字符串上施加有效性约束，也不方便执行时间上的比较。第二种替代方案是把时间编码成整数，该整数表示从午夜以来的分钟（或秒）数，但是这种方案需要每次查询有额外代码来进行标准时间表示和整数编码之间的转换。因此我们选用两部分表示的方案。

A.3 示例数据

在本节我们将为前一节定义的每个关系提供示例数据，如图 A-3 ~ 图 A-14 所示。

building	*room_number*	*capacity*
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

图 A-3 *classroom* 关系

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

图 A-4 *department* 关系

course_id	*title*	*dept_name*	*credits*
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

图 A-5 *course* 关系

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

图 A-6 *instructor* 关系

course_id	sec_id	semester	year	building	room_number	time_slot_id
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

图 A-7 *section* 关系

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

图 A-8 *teaches* 关系

ID	name	dept_name	tot_cred
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

图 A-9 *student* 关系

ID	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	*null*

图 A-10 *takes* 关系

s_id	*i_id*
00128	45565
12345	10101
23121	76543
44553	22222
45678	22222
76543	45565
76653	98345
98765	98345
98988	76766

图 A-11 *advisor* 关系

time_slot_id	*day*	*start_time*	*end_time*
A	M	8:00	8:50
A	W	8:00	8:50
A	F	8:00	8:50
B	M	9:00	9:50
B	W	9:00	9:50
B	F	9:00	9:50
C	M	11:00	11:50
C	W	11:00	11:50
C	F	11:00	11:50
D	M	13:00	13:50
D	W	13:00	13:50
D	F	13:00	13:50
E	T	10:30	11:45
E	R	10:30	11:45
F	T	14:30	15:45
F	R	14:30	15:45
G	M	16:00	16:50
G	W	16:00	16:50
G	F	16:00	16:50
H	W	10:00	12:30

图 A-12 *time_slot* 关系

course_id	prereq_id
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

表 A-13 *prereq* 关系

time_slot_id	day	start_hr	start_min	end_hr	end_min
A	M	8	0	8	50
A	W	8	0	8	50
A	F	8	0	8	50
B	M	9	0	9	50
B	W	9	0	9	50
B	F	9	0	9	50
C	M	11	0	11	50
C	W	11	0	11	50
C	F	11	0	11	50
D	M	13	0	13	50
D	W	13	0	13	50
D	F	13	0	13	50
E	T	10	30	11	45
E	R	10	30	11	45
F	T	14	30	15	45
F	R	14	30	15	45
G	M	16	0	16	50
G	W	16	0	16	50
G	F	16	0	16	50
H	W	10	0	12	30

表 A-14 具有起始和终止时间被拆分成小时和分钟的 *time_slot* 关系

参考文献

[Abadi 2009] D. Abadi, "Data Management in the Cloud: Limitations and Opportunities", *Data Engineering Bulletin*, Volume 32, Number 1 (2009), pages 3–12.

[Abadi et al. 2008] D. J. Abadi, S. Madden, and N. Hachem, "Column-stores vs. row-stores: how different are they really?", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2008), pages 967–980.

[Abiteboul et al. 1995] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*, Addison Wesley (1995).

[Abiteboul et al. 2003] S. Abiteboul, R. Agrawal, P. A. Bernstein, M. J. Carey, et al. "The Lowell Database Research Self Assessment" (2003).

[Acheson et al. 2004] A. Acheson, M. Bendixen, J. A. Blakeley, I. P. Carlin, E. Ersan, J. Fang, X. Jiang, C. Kleinerman, B. Rathakrishnan, G. Schaller, B. Sezgin, R. Venkatesh, and H. Zhang, "Hosting the .NET Runtime in Microsoft SQL Server", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2004), pages 860–865.

[Adali et al. 1996] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian, "Query Caching and Optimization in Distributed Mediator Systems", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996), pages 137–148.

[Adya et al. 2007] A. Adya, J. A. Blakeley, S. Melnik, and S. Muralidhar, "Anatomy of the ADO.NET entity framework", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2007), pages 877–888.

[Agarwal et al. 1996] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi, "On the Computation of Multidimensional Aggregates", In *Proc. of the International Conf. on Very Large Databases* (1996), pages 506–521.

[Agrawal and Srikant 1994] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases", In *Proc. of the International Conf. on Very Large Databases* (1994), pages 487–499.

[Agrawal et al. 1992] R. Agrawal, S. P. Ghosh, T. Imielinski, B. R. Iyer, and A. N. Swami, "An Interval Classifier for Database Mining Applications", In *Proc. of the International Conf. on Very Large Databases* (1992), pages 560–573.

[Agrawal et al. 1993a] R. Agrawal, T. Imielinski, and A. Swami, "Mining Association Rules between Sets of Items in Large Databases", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993).

[Agrawal et al. 1993b] R. Agrawal, T. Imielinski, and A. N. Swami, "Database Mining: A Performance Perspective", *IEEE Transactions on Knowledge and Data Engineering*, Volume 5, Number 6 (1993), pages 914–925.

[Agrawal et al. 2000] S. Agrawal, S. Chaudhuri, and V. R. Narasayya, "Automated Selection of Materialized Views and Indexes in SQL Databases", In *Proc. of the International Conf. on Very Large Databases* (2000), pages 496–505.

[Agrawal et al. 2002] S. Agrawal, S. Chaudhuri, and G. Das, "DBXplorer: A System for Keyword-Based Search over Relational Databases", In *Proc. of the International Conf. on Data Engineering* (2002).

[Agrawal et al. 2004] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala, "Database Tuning Advisor for Microsoft SQL Server 2005", In *Proc. of the International Conf. on Very Large Databases* (2004).

[Agrawal et al. 2009] R. Agrawal, A. Ailamaki, P. A. Bernstein, E. A. Brewer, M. J.

Carey, S. Chaudhuri, A. Doan, D. Florescu, M. J. Franklin, H. Garcia-Molina, J. Gehrke, L. Gruenwald, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. F. Korth, D. Kossmann, S. Madden, R. Magoulas, B. C. Ooi, T. O'Reilly, R. Ramakrishnan, S. Sarawagi, and G. W. Michael Stonebraker, Alexander S. Szalay, "The Claremont Report on Database Research", *Communications of the ACM*, Volume 52, Number 6 (2009), pages 56–65.

[Ahmed et al. 2006] R. Ahmed, A. Lee, A. Witkowski, D. Das, H. Su, M. Zait, and T. Cruanes, "Cost-Based Query Transformation in Oracle", In *Proc. of the International Conf. on Very Large Databases* (2006), pages 1026–1036.

[Aho et al. 1979a] A. V. Aho, C. Beeri, and J. D. Ullman, "The Theory of Joins in Relational Databases", *ACM Transactions on Database Systems*, Volume 4, Number 3 (1979), pages 297–314.

[Aho et al. 1979b] A. V. Aho, Y. Sagiv, and J. D. Ullman, "Equivalences among Relational Expressions", *SIAM Journal of Computing*, Volume 8, Number 2 (1979), pages 218–246.

[Ailamaki et al. 2001] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving Relations for Cache Performance", In *Proc. of the International Conf. on Very Large Databases* (2001), pages 169–180.

[Alonso and Korth 1993] R. Alonso and H. F. Korth, "Database System Issues in Nomadic Computing", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993), pages 388–392.

[Amer-Yahia et al. 2004] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram, "TeXQuery: A Full-Text Search Extension to XQuery", In *Proc. of the International World Wide Web Conf.* (2004).

[Anderson et al. 1992] D. P. Anderson, Y. Osawa, and R. Govindan, "A File System for Continuous Media", *ACM Transactions on Database Systems*, Volume 10, Number 4 (1992), pages 311–337.

[Anderson et al. 1998] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool, "Replication, Consistency and Practicality: Are These Mutually Exclusive?", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1998).

[ANSI 1986] *American National Standard for Information Systems. Database Language SQL.* American National Standards Institute (1986).

[ANSI 1989] *Database Language SQL with Integrity Enhancement, ANSI X3, 135–1989.* American National Standards Institute, New York (1989).

[ANSI 1992] *Database Language SQL, ANSI X3,135–1992.* American National Standards Institute, New York (1992).

[Antoshenkov 1995] G. Antoshenkov, "Byte-aligned Bitmap Compression (poster abstract)", In *IEEE Data Compression Conf.* (1995).

[Appelt and Israel 1999] D. E. Appelt and D. J. Israel, "Introduction to Information Extraction Technology", In *Proc. of the International Joint Conferences on Artificial Intelligence* (1999).

[Apt and Pugin 1987] K. R. Apt and J. M. Pugin, "Maintenance of Stratified Database Viewed as a Belief Revision System", In *Proc. of the ACM Symposium on Principles of Database Systems* (1987), pages 136–145.

[Armstrong 1974] W. W. Armstrong, "Dependency Structures of Data Base Relationships", In *Proc. of the 1974 IFIP Congress* (1974), pages 580–583.

[Astrahan et al. 1976] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, "System R, A Relational Approach to Data Base Management", *ACM Transactions on Database Systems*, Volume 1, Number 2 (1976), pages 97–137.

[Atreya et al. 2002] M. Atreya, B. Hammond, S. Paine, P. Starrett, and S. Wu, *Digital Signatures*, RSA Press (2002).

[Atzeni and Antonellis 1993] P. Atzeni and V. D. Antonellis, *Relational Database Theory*, Benjamin Cummings (1993).

[Baeza-Yates and Ribeiro-Neto 1999] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, Addison Wesley (1999).

[Bancilhon et al. 1989] F. Bancilhon, S. Cluet, and C. Delobel, "A Query Language for the O_2 Object-Oriented Database", In *Proc. of the Second Workshop on Database Programming Languages* (1989).

[Baras et al. 2005] A. Baras, D. Churin, I. Cseri, T. Grabs, E. Kogan, S. Pal, M. Rys, and O. Seeliger. "Implementing XQuery in a Relational Database System" (2005).

[Baru et al. 1995] C. Baru et al., "DB2 Parallel Edition", *IBM Systems Journal*, Volume 34, Number 2 (1995), pages 292–322.

[Bassiouni 1988] M. Bassiouni, "Single-site and Distributed Optimistic Protocols for Concurrency Control", *IEEE Transactions on Software Engineering*, Volume SE-14, Number 8 (1988), pages 1071–1080.

[Batini et al. 1992] C. Batini, S. Ceri, and S. Navathe, *Database Design: An Entity-Relationship Approach*, Benjamin Cummings (1992).

[Bayer 1972] R. Bayer, "Symmetric Binary B-trees: Data Structure and Maintenance Algorithms", *Acta Informatica*, Volume 1, Number 4 (1972), pages 290–306.

[Bayer and McCreight 1972] R. Bayer and E. M. McCreight, "Organization and Maintenance of Large Ordered Indices", *Acta Informatica*, Volume 1, Number 3 (1972), pages 173–189.

[Bayer and Schkolnick 1977] R. Bayer and M. Schkolnick, "Concurrency of Operating on B-trees", *Acta Informatica*, Volume 9, Number 1 (1977), pages 1–21.

[Bayer and Unterauer 1977] R. Bayer and K. Unterauer, "Prefix B-trees", *ACM Transactions on Database Systems*, Volume 2, Number 1 (1977), pages 11–26.

[Bayer et al. 1978] R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, Springer Verlag (1978).

[Beckmann et al. 1990] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990), pages 322–331.

[Beeri et al. 1977] C. Beeri, R. Fagin, and J. H. Howard, "A Complete Axiomatization for Functional and Multivalued Dependencies", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1977), pages 47–61.

[Bentley 1975] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching", *Communications of the ACM*, Volume 18, Number 9 (1975), pages 509–517.

[Berenson et al. 1995] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A Critique of ANSI SQL Isolation Levels", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), pages 1–10.

[Bernstein and Goodman 1981] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Survey*, Volume 13, Number 2 (1981), pages 185–221.

[Bernstein and Newcomer 1997] P. A. Bernstein and E. Newcomer, *Principles of Transaction Processing*, Morgan Kaufmann (1997).

[Bernstein et al. 1998] P. Bernstein, M. Brodie, S. Ceri, D. DeWitt, M. Franklin, H. Garcia-Molina, J. Gray, J. Held, J. Hellerstein, H. V. Jagadish, M. Lesk, D. Maier, J. Naughton, H. Pirahesh, M. Stonebraker, and J. Ullman, "The Asilomar Report on Database Research", *ACM SIGMOD Record*, Volume 27, Number 4 (1998).

[Berson et al. 1995] S. Berson, L. Golubchik, and R. R. Muntz, "Fault Tolerant Design of Multimedia Servers", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), pages 364–375.

[Bhalotia et al. 2002] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword Searching and Browsing in Databases using BANKS", In *Proc. of the International Conf. on Data Engineering* (2002).

[Bharat and Henzinger 1998] K. Bharat and M. R. Henzinger, "Improved Algorithms for Topic Distillation in a Hyperlinked Environment", In *Proc. of the ACM SIGIR Conf. on Research and Development in Information Retrieval* (1998), pages 104–111.

[Bhattacharjee et al. 2003] B. Bhattacharjee, S. Padmanabhan, T. Malkemus, T. Lai, L. Cranston, and M. Huras, "Efficient Query Processing for Multi-Dimensionally Clustered Tables in DB2", In *Proc. of the International Conf. on Very Large Databases* (2003), pages 963–974.

[Biskup et al. 1979] J. Biskup, U. Dayal, and P. A. Bernstein, "Synthesizing Independent Database Schemas", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1979), pages 143–152.

[Bitton et al. 1983] D. Bitton, D. J. DeWitt, and C. Turbyfill, "Benchmarking Database Systems: A Systematic Approach", In *Proc. of the International Conf. on Very Large Databases* (1983).

[Blakeley 1996] J. A. Blakeley, "Data Access for the Masses through OLE DB", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996), pages 161–172.

[Blakeley and Pizzo 2001] J. A. Blakeley and M. Pizzo, "Enabling Component Databases with OLE DB", In K. R. Dittrich and A. Geppert, editors, *Component Database Systems*, Morgan Kaufmann Publishers (2001), pages 139–173.

[Blakeley et al. 1986] J. A. Blakeley, P. Larson, and F. W. Tompa, "Efficiently Updating Materialized Views", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1986), pages 61–71.

[Blakeley et al. 2005] J. A. Blakeley, C. Cunningham, N. Ellis, B. Rathakrishnan, and M.-C. Wu, "Distributed/Heterogeneous Query Processing in Microsoft SQL Server", In *Proc. of the International Conf. on Data Engineering* (2005).

[Blakeley et al. 2006] J. A. Blakeley, D. Campbell, S. Muralidhar, and A. Nori, "The ADO.NET entity framework: making the conceptual level real", *SIGMOD Record*, Volume 35, Number 4 (2006), pages 32–39.

[Blakeley et al. 2008] J. A. Blakeley, V. Rao, I. Kunen, A. Prout, M. Henaire, and C. Kleinerman, ".NET database programmability and extensibility in Microsoft SQL server", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2008), pages 1087–1098.

[Blasgen and Eswaran 1976] M. W. Blasgen and K. P. Eswaran, "On the Evaluation of Queries in a Relational Database System", *IBM Systems Journal*, Volume 16, (1976), pages 363–377.

[Boyce et al. 1975] R. Boyce, D. D. Chamberlin, W. F. King, and M. Hammer, "Specifying Queries as Relational Expressions", *Communications of the ACM*, Volume 18, Number 11 (1975), pages 621–628.

[Brantner et al. 2008] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska, "Building a Database on S3", In *Proc. of the ACM SIGMOD Conf. on*

Management of Data (2008), pages 251–263.

[Breese et al. 1998] J. Breese, D. Heckerman, and C. Kadie, "Empirical Analysis of Predictive Algorithms for Collaborative Filtering", In *Procs. Conf. on Uncertainty in Artificial Intelligence*, Morgan Kaufmann (1998).

[Breitbart et al. 1999a] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz, "Update Propagation Protocols For Replicated Databases", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1999), pages 97–108.

[Breitbart et al. 1999b] Y. Breitbart, H. Korth, A. Silberschatz, and S. Sudarshan, "Distributed Databases", In *Encyclopedia of Electrical and Electronics Engineering*, John Wiley and Sons (1999).

[Brewer 2000] E. A. Brewer, "Towards robust distributed systems (abstract)", In *Proc. of the ACM Symposium on Principles of Distributed Computing* (2000), page 7.

[Brin and Page 1998] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine", In *Proc. of the International World Wide Web Conf.* (1998).

[Brinkhoff et al. 1993] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient Processing of Spatial Joins Using R-trees", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993), pages 237–246.

[Bruno et al. 2002] N. Bruno, S. Chaudhuri, and L. Gravano, "Top-k Selection Queries Over Relational Databases: Mapping Strategies and Performance Evaluation", *ACM Transactions on Database Systems*, Volume 27, Number 2 (2002), pages 153–187.

[Buckley and Silberschatz 1983] G. Buckley and A. Silberschatz, "Obtaining Progressive Protocols for a Simple Multiversion Database Model", In *Proc. of the International Conf. on Very Large Databases* (1983), pages 74–81.

[Buckley and Silberschatz 1984] G. Buckley and A. Silberschatz, "Concurrency Control in Graph Protocols by Using Edge Locks", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1984), pages 45–50.

[Buckley and Silberschatz 1985] G. Buckley and A. Silberschatz, "Beyond Two-Phase Locking", *Journal of the ACM*, Volume 32, Number 2 (1985), pages 314–326.

[Bulmer 1979] M. G. Bulmer, *Principles of Statistics*, Dover Publications (1979).

[Burkhard 1976] W. A. Burkhard, "Hashing and Trie Algorithms for Partial Match Retrieval", *ACM Transactions on Database Systems*, Volume 1, Number 2 (1976), pages 175–187.

[Burkhard 1979] W. A. Burkhard, "Partial-match Hash Coding: Benefits of Redundancy", *ACM Transactions on Database Systems*, Volume 4, Number 2 (1979), pages 228–239.

[Cannan and Otten 1993] S. Cannan and G. Otten, *SQL—The Standard Handbook*, McGraw Hill (1993).

[Carey 1983] M. J. Carey, "Granularity Hierarchies in Concurrency Control", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1983), pages 156–165.

[Carey and Kossmann 1998] M. J. Carey and D. Kossmann, "Reducing the Braking Distance of an SQL Query Engine", In *Proc. of the International Conf. on Very Large Databases* (1998), pages 158–169.

[Carey et al. 1991] M. Carey, M. Franklin, M. Livny, and E. Shekita, "Data Caching Tradeoffs in Client-Server DBMS Architectures", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1991).

[Carey et al. 1993] M. J. Carey, D. DeWitt, and J. Naughton, "The OO7 Benchmark", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993).

[Carey et al. 1999] M. J. Carey, D. D. Chamberlin, S. Narayanan, B. Vance, D. Doole, S. Rielau, R. Swagerman, and N. Mattos, "O-O, What Have They Done to DB2?", In *Proc. of the International Conf. on Very Large Databases* (1999), pages 542–553.

[Cattell 2000] R. Cattell, editor, *The Object Database Standard: ODMG 3.0*, Morgan Kaufmann (2000).

[Cattell and Skeen 1992] R. Cattell and J. Skeen, "Object Operations Benchmark", *ACM Transactions on Database Systems*, Volume 17, Number 1 (1992).

[Chakrabarti 1999] S. Chakrabarti, "Recent Results in Automatic Web Resource Discovery", *ACM Computing Surveys*, Volume 31, Number 4 (1999).

[Chakrabarti 2000] S. Chakrabarti, "Data Mining for Hypertext: A Tutorial Survey", *SIGKDD Explorations*, Volume 1, Number 2 (2000), pages 1–11.

[Chakrabarti 2002] S. Chakrabarti, *Mining the Web: Discovering Knowledge from HyperText Data*, Morgan Kaufmann (2002).

[Chakrabarti et al. 1998] S. Chakrabarti, S. Sarawagi, and B. Dom, "Mining Surprising Patterns Using Temporal Description Length", In *Proc. of the International Conf. on Very Large Databases* (1998), pages 606–617.

[Chakrabarti et al. 1999] S. Chakrabarti, M. van den Berg, and B. Dom, "Focused Crawling: A New Approach to Topic Specific Web Resource Discovery", In *Proc. of the International World Wide Web Conf.* (1999).

[Chamberlin 1996] D. Chamberlin, *Using the New DB2: IBM's Object-Relational Database System*, Morgan Kaufmann (1996).

[Chamberlin 1998] D. D. Chamberlin, *A Complete Guide to DB2 Universal Database*, Morgan Kaufmann (1998).

[Chamberlin and Boyce 1974] D. D. Chamberlin and R. F. Boyce, "SEQUEL: A Structured English Query Language", In *ACM SIGMOD Workshop on Data Description, Access, and Control* (1974), pages 249–264.

[Chamberlin et al. 1976] D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner, and B. W. Wade, "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control", *IBM Journal of Research and Development*, Volume 20, Number 6 (1976), pages 560–575.

[Chamberlin et al. 1981] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost, "A History and Evaluation of System R", *Communications of the ACM*, Volume 24, Number 10 (1981), pages 632–646.

[Chamberlin et al. 2000] D. D. Chamberlin, J. Robie, and D. Florescu, "Quilt: An XML Query Language for Heterogeneous Data Sources", In *Proc. of the International Workshop on the Web and Databases (WebDB)* (2000), pages 53–62.

[Chan and Ioannidis 1998] C.-Y. Chan and Y. E. Ioannidis, "Bitmap Index Design and Evaluation", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1998).

[Chan and Ioannidis 1999] C.-Y. Chan and Y. E. Ioannidis, "An Efficient Bitmap Encoding Scheme for Selection Queries", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1999).

[Chandra and Harel 1982] A. K. Chandra and D. Harel, "Structure and Complexity of Relational Queries", *Journal of Computer and System Sciences*, Volume 15, Number 10 (1982), pages 99–128.

[Chandrasekaran et al. 2003] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah, "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World", In *First Biennial Conference on Innovative Data Systems Research* (2003).

[Chang et al. 2008] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data", *ACM Trans. Comput. Syst.*, Volume 26, Number 2 (2008).

[Chatziantoniou and Ross 1997] D. Chatziantoniou and K. A. Ross, "Groupwise Processing of Relational Queries", In *Proc. of the International Conf. on Very Large Databases* (1997), pages 476–485.

[Chaudhuri and Narasayya 1997] S. Chaudhuri and V. Narasayya, "An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server", In *Proc. of the International Conf. on Very Large Databases* (1997).

[Chaudhuri and Shim 1994] S. Chaudhuri and K. Shim, "Including Group-By in Query Optimization", In *Proc. of the International Conf. on Very Large Databases* (1994).

[Chaudhuri et al. 1995] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim, "Optimizing Queries with Materialized Views", In *Proc. of the International Conf. on Data Engineering* (1995).

[Chaudhuri et al. 1998] S. Chaudhuri, R. Motwani, and V. Narasayya, "Random sampling for histogram construction: how much is enough?", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1998), pages 436–447.

[Chaudhuri et al. 1999] S. Chaudhuri, E. Christensen, G. Graefe, V. Narasayya, and M. Zwilling, "Self Tuning Technology in Microsoft SQL Server", *IEEE Data Engineering Bulletin*, Volume 22, Number 2 (1999).

[Chaudhuri et al. 2003] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani, "Robust and Efficient Fuzzy Match for Online Data Cleaning", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2003).

[Chen 1976] P. P. Chen, "The Entity-Relationship Model: Toward a Unified View of Data", *ACM Transactions on Database Systems*, Volume 1, Number 1 (1976), pages 9–36.

[Chen et al. 1994] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-Performance, Reliable Secondary Storage", *ACM Computing Survey*, Volume 26, Number 2 (1994).

[Chen et al. 2007] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry, "Improving hash join performance through prefetching", *ACM Transactions on Database Systems*, Volume 32, Number 3 (2007).

[Chomicki 1995] J. Chomicki, "Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding", *ACM Transactions on Database Systems*, Volume 20, Number 2 (1995), pages 149–186.

[Chou and Dewitt 1985] H. T. Chou and D. J. Dewitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems", In *Proc. of the International Conf. on Very Large Databases* (1985), pages 127–141.

[Cieslewicz et al. 2009] J. Cieslewicz, W. Mee, and K. A. Ross, "Cache-Conscious Buffering for Database Operators with State", In *Proc. Fifth International Workshop on Data Management on New Hardware (DaMoN 2009)* (2009).

[Cochrane et al. 1996] R. Cochrane, H. Pirahesh, and N. M. Mattos, "Integrating Triggers and Declarative Constraints in SQL Database Systems", In *Proc. of the International Conf. on Very Large Databases* (1996), pages 567–578.

[Codd 1970] E. F. Codd, "A Relational Model for Large Shared Data Banks", *Communications of the ACM*, Volume 13, Number 6 (1970), pages 377–387.

[Codd 1972] E. F. Codd, "Further Normalization of the Data Base Relational Model", In *Rustin (1972)*, pages 33–64 (1972).

[Codd 1979] E. F. Codd, "Extending the Database Relational Model to Capture More Meaning", *ACM Transactions on Database Systems*, Volume 4, Number 4 (1979), pages 397–434.

[Codd 1982] E. F. Codd, "The 1981 ACM Turing Award Lecture: Relational Database: A Practical Foundation for Productivity", *Communications of the ACM*, Volume 25, Number 2 (1982), pages 109–117.

[Codd 1990] E. F. Codd, *The Relational Model for Database Management: Version 2*, Addison Wesley (1990).

[Comer 1979] D. Comer, "The Ubiquitous B-tree", *ACM Computing Surveys*, Volume 11, Number 2 (1979), pages 121–137.

[Comer 2009] D. E. Comer, *Computer Networks and Internets*, 5th edition, Prentice Hall (2009).

[Cook 1996] M. A. Cook, *Building Enterprise Information Architecture: Reengineering Information Systems*, Prentice Hall (1996).

[Cooper et al. 2008] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform", *Proceedings of the VLDB Endowment*, Volume 1, Number 2 (2008), pages 1277–1288.

[Cormen et al. 1990] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, MIT Press (1990).

[Cortes and Vapnik 1995] C. Cortes and V. Vapnik, *Machine Learning*, Volume 20, Number 3 (1995), pages 273–297.

[Cristianini and Shawe-Taylor 2000] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines and other Kernel-Based Learning Methods*, Cambridge University Press (2000).

[Dageville and Zait 2002] B. Dageville and M. Zait, "SQL Memory Management in Oracle9i", In *Proc. of the International Conf. on Very Large Databases* (2002), pages 962–973.

[Dageville et al. 2004] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin, "Automatic SQL Tuning in Oracle 10g", In *Proc. of the International Conf. on Very Large Databases* (2004), pages 1098–1109.

[Dalvi et al. 2009] N. Dalvi, R. Kumar, B. Pang, R. Ramakrishnan, A. Tomkins, P. Bohannon, S. Keerthi, and S. Merugu, "A Web of Concepts", In *Proc. of the ACM Symposium on Principles of Database Systems* (2009).

[Daniels et al. 1982] D. Daniels, P. G. Selinger, L. M. Haas, B. G. Lindsay, C. Mohan, A. Walker, and P. F. Wilms, "An Introduction to Distributed Query Compilation in R*", In *Schneider [1982]* (1982).

[Dashti et al. 2003] A. Dashti, S. H. Kim, C. Shahabi, and R. Zimmermann, *Streaming Media Server Design*, Prentice Hall (2003).

[Date 1983] C. J. Date, "The Outer Join", In *Proc. of the International Conference on Databases*, John Wiley and Sons (1983), pages 76–106.

[Date 1989] C. Date, *A Guide to DB2*, Addison Wesley (1989).

[Date 1993] C. J. Date, "How SQL Missed the Boat", *Database Programming and Design*, Volume 6, Number 9 (1993).

[Date 2003] C. J. Date, *An Introduction to Database Systems*, 8th edition, Addison Wesley (2003).

[Date and Darwen 1997] C. J. Date and G. Darwen, *A Guide to the SQL Standard*, 4th edition, Addison Wesley (1997).

[Davis et al. 1983] C. Davis, S. Jajodia, P. A. Ng, and R. Yeh, editors, *Entity-Relationship Approach to Software Engineering*, North Holland (1983).

[Davison and Graefe 1994] D. L. Davison and G. Graefe, "Memory-Contention Responsive Hash Joins", In *Proc. of the International Conf. on Very Large Databases* (1994).

[Dayal 1987] U. Dayal, "Of Nests and Trees: A Unified Approach to Processing Queries that Contain Nested Subqueries, Aggregates and Quantifiers", In *Proc. of the International Conf. on Very Large Databases* (1987), pages 197–208.

[Deutsch et al. 1999] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, "A Query Language for XML", In *Proc. of the International World Wide Web Conf.* (1999).

[DeWitt 1990] D. DeWitt, "The Gamma Database Machine Project", *IEEE Transactions on Knowledge and Data Engineering*, Volume 2, Number 1 (1990).

[DeWitt and Gray 1992] D. DeWitt and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems", *Communications of the ACM*, Volume 35, Number 6 (1992), pages 85–98.

[DeWitt et al. 1992] D. DeWitt, J. Naughton, D. Schneider, and S. Seshadri, "Practical Skew Handling in Parallel Joins", In *Proc. of the International Conf. on Very Large Databases* (1992).

[Dias et al. 1989] D. Dias, B. Iyer, J. Robinson, and P. Yu, "Integrated Concurrency-Coherency Controls for Multisystem Data Sharing", *Software Engineering*, Volume 15, Number 4 (1989), pages 437–448.

[Donahoo and Speegle 2005] M. J. Donahoo and G. D. Speegle, *SQL: Practical Guide for Developers*, Morgan Kaufmann (2005).

[Douglas and Douglas 2003] K. Douglas and S. Douglas, *PostgreSQL*, Sam's Publishing (2003).

[Dubois and Thakkar 1992] M. Dubois and S. Thakkar, editors, *Scalable Shared Memory Multiprocessors*, Kluwer Academic Publishers (1992).

[Duncan 1990] R. Duncan, "A Survey of Parallel Computer Architectures", *IEEE Computer*, Volume 23, Number 2 (1990), pages 5–16.

[Eisenberg and Melton 1999] A. Eisenberg and J. Melton, "SQL:1999, formerly known as SQL3", *ACM SIGMOD Record*, Volume 28, Number 1 (1999).

[Eisenberg and Melton 2004a] A. Eisenberg and J. Melton, "Advancements in SQL/XML", *ACM SIGMOD Record*, Volume 33, Number 3 (2004), pages 79–86.

[Eisenberg and Melton 2004b] A. Eisenberg and J. Melton, "An Early Look at XQuery API for Java (XQJ)", *ACM SIGMOD Record*, Volume 33, Number 2 (2004), pages 105–111.

[Eisenberg et al. 2004] A. Eisenberg, J. Melton, K. G. Kulkarni, J. E. Michels, and F. Zemke, "SQL:2003 Has Been Published", *ACM SIGMOD Record*, Volume 33, Number 1 (2004), pages 119–126.

[Elhemali et al. 2007] M. Elhemali, C. A. Galindo-Legaria, T. Grabs, and M. Joshi, "Execution strategies for SQL subqueries", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2007), pages 993–1004.

[Ellis 1987] C. S. Ellis, "Concurrency in Linear Hashing", *ACM Transactions on Database Systems*, Volume 12, Number 2 (1987), pages 195–217.

[Elmasri and Navathe 2006] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, 5th edition, Addison Wesley (2006).

[Epstein et al. 1978] R. Epstein, M. R. Stonebraker, and E. Wong, "Distributed Query Processing in a Relational Database System", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1978), pages 169–180.

[Escobar-Molano et al. 1993] M. Escobar-Molano, R. Hull, and D. Jacobs, "Safety and Translation of Calculus Queries with Scalar Functions". In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993), pages 253–264.

[Eswaran et al. 1976] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System", *Communications of the ACM*, Volume 19, Number 11 (1976), pages 624–633.

[Fagin 1977] R. Fagin, "Multivalued Dependencies and a New Normal Form for Relational Databases", *ACM Transactions on Database Systems*, Volume 2, Number 3 (1977), pages 262–278.

[Fagin 1979] R. Fagin, "Normal Forms and Relational Database Operators", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1979), pages 153–160.

[Fagin 1981] R. Fagin, "A Normal Form for Relational Databases That Is Based on Domains and Keys", *ACM Transactions on Database Systems*, Volume 6, Number 3 (1981), pages 387–415.

[Fagin et al. 1979] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible Hashing — A Fast Access Method for Dynamic Files", *ACM Transactions on Database Systems*, Volume 4, Number 3 (1979), pages 315–344.

[Faloutsos and Lin 1995] C. Faloutsos and K.-I. Lin, "Fast Map: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), pages 163–174.

[Fayyad et al. 1995] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, *Advances in Knowledge Discovery and Data Mining*, MIT Press (1995).

[Fekete et al. 2005] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha, "Making Snapshot Isolation Serializable", *ACM Transactions on Database Systems*, Volume 30, Number 2 (2005).

[Finkel and Bentley 1974] R. A. Finkel and J. L. Bentley, "Quad Trees: A Data Structure for Retrieval on Composite Keys", *Acta Informatica*, Volume 4, (1974), pages 1–9.

[Fischer 2006] L. Fischer, editor, *Workflow Handbook 2001*, Future Strategies (2006).

[Florescu and Kossmann 1999] D. Florescu and D. Kossmann, "Storing and Querying XML Data Using an RDBMS", *IEEE Data Engineering Bulletin (Special Issue on XML)* (1999), pages 27–35.

[Florescu et al. 2000] D. Florescu, D. Kossmann, and I. Monalescu, "Integrating Keyword Search into XML Query Processing", In *Proc. of the International World Wide Web Conf.* (2000), pages 119–135. Also appears in *Computer Networks*, Vol. 33, pages 119–135.

[Fredkin 1960] E. Fredkin, "Trie Memory", *Communications of the ACM*, Volume 4, Number 2 (1960), pages 490–499.

[Freedman and DeWitt 1995] C. S. Freedman and D. J. DeWitt, "The SPIFFI Scalable Video-on-Demand Server", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), pages 352–363.

[Funderburk et al. 2002a] J. E. Funderburk, G. Kiernan, J. Shanmugasundaram, E. Shekita, and C. Wei, "XTABLES: Bridging Relational Technology and XML", *IBM Systems Journal*, Volume 41, Number 4 (2002), pages 616–641.

[Funderburk et al. 2002b] J. E. Funderburk, S. Malaika, and B. Reinwald, "XML Programming with SQL/XML and XQuery", *IBM Systems Journal*, Volume 41, Number 4 (2002), pages 642–665.

[Galindo-Legaria 1994] C. Galindo-Legaria, "Outerjoins as Disjunctions", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1994).

[Galindo-Legaria and Joshi 2001] C. A. Galindo-Legaria and M. M. Joshi, "Orthogonal Optimization of Subqueries and Aggregation", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2001).

[Galindo-Legaria and Rosenthal 1992] C. Galindo-Legaria and A. Rosenthal, "How to Extend a Conventional Optimizer to Handle One- and Two-Sided Outerjoin", In *Proc. of the International Conf. on Data Engineering* (1992), pages 402–409.

[Galindo-Legaria et al. 2004] C. Galindo-Legaria, S. Stefani, and F. Waas, "Query Processing for SQL Updates", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2004), pages 844–849.

[Ganguly 1998] S. Ganguly, "Design and Analysis of Parametric Query Optimization Algorithms", In *Proc. of the International Conf. on Very Large Databases* (1998).

[Ganguly et al. 1992] S. Ganguly, W. Hasan, and R. Krishnamurthy, "Query Optimization for Parallel Execution", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1992).

[Ganguly et al. 1996] S. Ganguly, P. Gibbons, Y. Matias, and A. Silberschatz, "A Sampling Algorithm for Estimating Join Size", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[Ganski and Wong 1987] R. A. Ganski and H. K. T. Wong, "Optimization of Nested SQL Queries Revisited", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1987).

[Garcia and Korth 2005] P. Garcia and H. F. Korth, "Multithreaded Architectures and the Sort Benchmark", In *Proc. of the First International Workshop on Data Management on Modern Hardware (DaMoN)* (2005).

[Garcia-Molina 1982] H. Garcia-Molina, "Elections in Distributed Computing Systems", *IEEE Transactions on Computers*, Volume C-31, Number 1 (1982), pages 48–59.

[Garcia-Molina and Salem 1987] H. Garcia-Molina and K. Salem, "Sagas", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1987), pages 249–259.

[Garcia-Molina and Salem 1992] H. Garcia-Molina and K. Salem, "Main Memory Database Systems: An Overview", *IEEE Transactions on Knowledge and Data Engineering*, Volume 4, Number 6 (1992), pages 509–516.

[Garcia-Molina et al. 2008] H. Garcia-Molina, J. D. Ullman, and J. D. Widom, *Database Systems: The Complete Book*, 2nd edition, Prentice Hall (2008).

[Georgakopoulos et al. 1994] D. Georgakopoulos, M. Rusinkiewicz, and A. Seth, "Using Tickets to Enforce the Serializability of Multidatabase Transactions", *IEEE Transactions on Knowledge and Data Engineering*, Volume 6, Number 1 (1994), pages 166–180.

[Gilbert and Lynch 2002] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services", *SIGACT News*, Volume 33, Number 2 (2002), pages 51–59.

[Graefe 1990] G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990), pages 102–111.

[Graefe 1995] G. Graefe, "The Cascades Framework for Query Optimization", *Data Engineering Bulletin*, Volume 18, Number 3 (1995), pages 19–29.

[Graefe 2008] G. Graefe, "The Five-Minute Rule 20 Years Later: and How Flash Memory Changes the Rules", *ACM Queue*, Volume 6, Number 4 (2008), pages 40–52.

[Graefe and McKenna 1993a] G. Graefe and W. McKenna, "The Volcano Optimizer Generator", In *Proc. of the International Conf. on Data Engineering* (1993), pages 209–218.

[Graefe and McKenna 1993b] G. Graefe and W. J. McKenna, "Extensibility and Search Efficiency in the Volcano Optimizer Generator", In *Proc. of the International Conf. on Data Engineering* (1993).

[Graefe et al. 1998] G. Graefe, R. Bunker, and S. Cooper, "Hash Joins and Hash Teams in Microsoft SQL Server", In *Proc. of the International Conf. on Very Large Databases* (1998), pages 86–97.

[Gray 1978] J. Gray, "Notes on Data Base Operating System", In *Bayer et al. (1978)*, pages 393–481 (1978).

[Gray 1981] J. Gray, "The Transaction Concept: Virtues and Limitations", In *Proc. of the International Conf. on Very Large Databases* (1981), pages 144–154.

[Gray 1991] J. Gray, *The Benchmark Handbook for Database and Transaction Processing Systems*, 2nd edition, Morgan Kaufmann (1991).

[Gray and Graefe 1997] J. Gray and G. Graefe, "The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb", *SIGMOD Record*, Volume 26, Number 4 (1997), pages 63–68.

[Gray and Reuter 1993] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann (1993).

[Gray et al. 1975] J. Gray, R. A. Lorie, and G. R. Putzolu, "Granularity of Locks and Degrees of Consistency in a Shared Data Base", In *Proc. of the International Conf. on Very Large Databases* (1975), pages 428–451.

[Gray et al. 1976] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger, *Granularity of Locks and Degrees of Consistency in a Shared Data Base*, Nijssen (1976).

[Gray et al. 1981] J. Gray, P. R. McJones, and M. Blasgen, "The Recovery Manager of the System R Database Manager", *ACM Computing Survey*, Volume 13, Number 2 (1981), pages 223–242.

[Gray et al. 1995] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab and Sub-Totals", Technical report, Microsoft Research (1995).

[Gray et al. 1996] J. Gray, P. Helland, and P. O'Neil, "The Dangers of Replication and a Solution", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996), pages 173–182.

[Gray et al. 1997] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, "Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals", *Data Mining and Knowledge Discovery*, Volume 1, Number 1 (1997), pages 29–53.

[Gregersen and Jensen 1999] H. Gregersen and C. S. Jensen, "Temporal Entity-Relationship Models–A Survey", *IEEE Transactions on Knowledge and Data Engineering*, Volume 11, Number 3 (1999), pages 464–497.

[Grossman and Frieder 2004] D. A. Grossman and O. Frieder, *Information Retrieval: Algorithms and Heuristics*, 2nd edition, Springer Verlag (2004).

[Gunning 2008] P. K. Gunning, *ODB 9 for Developers*, MC Press (2008).

[Guo et al. 2003] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "XRANK: Ranked Keyword Search over XML Documents", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2003).

[Guttman 1984] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1984), pages 47–57.

[Haas et al. 1989] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh, "Extensible Query Processing in Starburst", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1989), pages 377–388.

[Haas et al. 1990] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. G. Lindsay, H. Pirahesh, M. J. Carey, and E. J. Shekita. "Starburst Mid-Flight: As the Dust Clears", *IEEE Transactions on Knowledge and Data Engineering*, Volume 2, Number 1 (1990), pages 143–160.

[Haerder and Reuter 1983] T. Haerder and A. Reuter, "Principles of Transaction-Oriented Database Recovery", *ACM Computing Survey*, Volume 15, Number 4 (1983), pages 287–318.

[Haerder and Rothermel 1987] T. Haerder and K. Rothermel, "Concepts for Transaction Recovery in Nested Transactions", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1987), pages 239–248.

[Halsall 2006] F. Halsall, *Computer Networking and the Internet : With Internet and Multimedia Applications*, Addison Wesley (2006).

[Han and Kamber 2000] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*, Morgan Kaufmann (2000).

[Harinarayan et al. 1996] V. Harinarayan, J. D. Ullman, and A. Rajaraman, "Implementing Data Cubes Efficiently", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[Haritsa et al. 1990] J. Haritsa, M. Carey, and M. Livny, "On Being Optimistic about Real-Time Constraints", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990).

[Harizopoulos and Ailamaki 2004] S. Harizopoulos and A. Ailamaki, "STEPS towards Cache-resident Transaction Processing", In *Proc. of the International Conf. on Very Large Databases* (2004), pages 660–671.

[Hellerstein and Stonebraker 2005] J. M. Hellerstein and M. Stonebraker, editors, *Readings in Database Systems*, 4th edition, Morgan Kaufmann (2005).

[Hellerstein et al. 1995] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer, "Generalized Search Trees for Database Systems", In *Proc. of the International Conf. on Very Large Databases* (1995), pages 562–573.

[Hennessy et al. 2006] J. L. Hennessy, D. A. Patterson, and D. Goldberg, *Computer Architecture: A Quantitative Approach*, 4th edition, Morgan Kaufmann (2006).

[Hevner and Yao 1979] A. R. Hevner and S. B. Yao, "Query Processing in Distributed Database Systems", *IEEE Transactions on Software Engineering*, Volume SE-5, Number 3 (1979), pages 177–187.

[Heywood et al. 2002] I. Heywood, S. Cornelius, and S. Carver, *An Introduction to Geographical Information Systems*, 2nd edition, Prentice Hall (2002).

[Hong et al. 1993] D. Hong, T. Johnson, and S. Chakravarthy, "Real-Time Transaction Scheduling: A Cost Conscious Approach", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993).

[Howes et al. 1999] T. A. Howes, M. C. Smith, and G. S. Good, *Understanding and Deploying LDAP Directory Services*, Macmillan Publishing (1999).

[Hristidis and Papakonstantinou 2002] V. Hristidis and Y. Papakonstantinou, "DISCOVER: Keyword Search in Relational Databases", In *Proc. of the International Conf. on Very Large Databases* (2002).

[Huang and Garcia-Molina 2001] Y. Huang and H. Garcia-Molina, "Exactly-once Semantics in a Replicated Messaging System", In *Proc. of the International Conf. on Data Engineering* (2001), pages 3–12.

[Hulgeri and Sudarshan 2003] A. Hulgeri and S. Sudarshan, "AniPQO: Almost Non-Intrusive Parametric Query Optimization for Non-Linear Cost Functions", In *Proc. of the International Conf. on Very Large Databases* (2003).

[IBM 1987] IBM, "Systems Application Architecture: Common Programming Interface, Database Reference", Technical report, IBM Corporation, IBM Form Number SC26-4348-0 (1987).

[Ilyas et al. 2008] I. Ilyas, G. Beskales, and M. A. Soliman, "A Survey of top-k query processing techniques in relational database systems", *ACM Computing Surveys*, Volume 40, Number 4 (2008).

[Imielinski and Badrinath 1994] T. Imielinski and B. R. Badrinath, "Mobile Computing—Solutions and Challenges", *Communications of the ACM*, Volume 37, Number 10 (1994).

[Imielinski and Korth 1996] T. Imielinski and H. F. Korth, editors, *Mobile Computing*, Kluwer Academic Publishers (1996).

[Ioannidis and Christodoulakis 1993] Y. Ioannidis and S. Christodoulakis, "Optimal Histograms for Limiting Worst-Case Error Propagation in the Size of Join Results", *ACM Transactions on Database Systems*, Volume 18, Number 4 (1993), pages 709–748.

[Ioannidis and Poosala 1995] Y. E. Ioannidis and V. Poosala, "Balancing Histogram Optimality and Practicality for Query Result Size Estimation", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), pages 233–244.

[Ioannidis et al. 1992] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis, "Parametric Query Optimization", In *Proc. of the International Conf. on Very Large Databases* (1992), pages 103–114.

[Jackson and Moulinier 2002] P. Jackson and I. Moulinier, *Natural Language Processing for Online Applications: Text Retrieval, Extraction, and Categorization*, John Benjamin (2002).

[Jagadish et al. 1993] H. V. Jagadish, A. Silberschatz, and S. Sudarshan, "Recovering from Main-Memory Lapses", In *Proc. of the International Conf. on Very Large Databases* (1993).

[Jagadish et al. 1994] H. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan, "Dali: A High Performance Main Memory Storage Manager", In *Proc. of the International Conf. on Very Large Databases* (1994).

[Jain and Dubes 1988] A. K. Jain and R. C. Dubes, *Algorithms for Clustering Data*, Prentice Hall (1988).

[Jensen et al. 1994] C. S. Jensen et al., "A Consensus Glossary of Temporal Database Concepts", *ACM SIGMOD Record*, Volume 23, Number 1 (1994), pages 52–64.

[Jensen et al. 1996] C. S. Jensen, R. T. Snodgrass, and M. Soo, "Extending Existing Dependency Theory to Temporal Databases", *IEEE Transactions on Knowledge and Data Engineering*, Volume 8, Number 4 (1996), pages 563–582.

[Johnson 1999] T. Johnson, "Performance Measurements of Compressed Bitmap Indices", In *Proc. of the International Conf. on Very Large Databases* (1999).

[Johnson and Shasha 1993] T. Johnson and D. Shasha, "The Performance of Concurrent B-Tree Algorithms", *ACM Transactions on Database Systems*, Volume 18, Number 1 (1993)

[Jones and Willet 1997] K. S. Jones and P. Willet, editors, *Readings in Information Retrieval*, Morgan Kaufmann (1997).

[Jordan and Russell 2003] D. Jordan and C. Russell, *Java Data Objects*, O'Reilly (2003).

[Jorwekar et al. 2007] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan, "Automating the Detection of Snapshot Isolation Anomalies", In *Proc. of the International Conf. on Very Large Databases* (2007), pages 1263–1274.

[Joshi 1991] A. Joshi, "Adaptive Locking Strategies in a Multi-Node Shared Data Model Environment", In *Proc. of the International Conf. on Very Large Databases* (1991).

[Kanne and Moerkotte 2000] C.-C. Kanne and G. Moerkotte, "Efficient Storage of XML Data", In *Proc. of the International Conf. on Data Engineering* (2000), page 198.

[Katz et al. 2004] H. Katz, D. Chamberlin, D. Draper, M. Fernandez, M. Kay, J. Robie, M. Rys, J. Simeon, J. Tivy, and P. Wadler, *XQuery from the Experts: A Guide to the W3C XML Query Language*, Addison Wesley (2004).

[Kaushik et al. 2004] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan, "On the Integration of Structure Indexes and Inverted Lists", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2004).

[Kedem and Silberschatz 1979] Z. M. Kedem and A. Silberschatz, "Controlling Concurrency Using Locking Protocols", In *Proc. of the Annual IEEE Symposium on Foundations of Computer Science* (1979), pages 275–285.

[Kedem and Silberschatz 1983] Z. M. Kedem and A. Silberschatz, "Locking Protocols: From Exclusive to Shared Locks", *Journal of the ACM*, Volume 30, Number 4 (1983), pages 787–804.

[Kifer et al. 2005] M. Kifer, A. Bernstein, and P. Lewis, *Database Systems: An Application Oriented Approach, Complete Version*, 2nd edition, Addison Wesley (2005).

[Kim 1982] W. Kim, "On Optimizing an SQL-like Nested Query", *ACM Transactions on Database Systems*, Volume 3, Number 3 (1982), pages 443–469.

[Kim 1995] W. Kim, editor, *Modern Database Systems*, ACM Press (1995).

[King et al. 1991] R. P. King, N. Halim, H. Garcia-Molina, and C. Polyzois, "Management of a Remote Backup Copy for Disaster Recovery", *ACM Transactions on Database Systems*, Volume 16, Number 2 (1991), pages 338–368.

[Kitsuregawa and Ogawa 1990] M. Kitsuregawa and Y. Ogawa, "Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Skew in the Super Database Computer", In *Proc. of the International Conf. on Very Large Databases* (1990), pages 210–221.

[Kleinberg 1999] J. M. Kleinberg, "Authoritative Sources in a Hyperlinked Environment", *Journal of the ACM*, Volume 46, Number 5 (1999), pages 604–632.

[Kleinrock 1975] L. Kleinrock, *Queueing Systems*, Wiley-Interscience (1975).

[Klug 1982] A. Klug, "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions", *Journal of the ACM*, Volume 29, Number 3 (1982), pages 699–717.

[Knapp 1987] E. Knapp, "Deadlock Detection in Distributed Databases", *ACM Computing Survey*, Volume 19, Number 4 (1987).

[Knuth 1973] D. E. Knuth, *The Art of Computer Programming, Volume 3*, Addison Wesley, Sorting and Searching (1973).

[Kohavi and Provost 2001] R. Kohavi and F. Provost, editors, *Applications of Data Mining to Electronic Commerce*, Kluwer Academic Publishers (2001).

[Konstan et al. 1997] J. A. Konstan, B. N. Miller, D. Maltz, J. L. Herlocker, L. R. Gordon, and J. Riedl, "GroupLens: Applying Collaborative Filtering to Usenet News", *Communications of the ACM*, Volume 40, Number 3 (1997), pages 77–87.

[Korth 1982] H. F. Korth, "Deadlock Freedom Using Edge Locks", *ACM Transactions on Database Systems*, Volume 7, Number 4 (1982), pages 632–652.

[Korth 1983] H. F. Korth, "Locking Primitives in a Database System", *Journal of the ACM*, Volume 30, Number 1 (1983), pages 55–79.

[Korth and Speegle 1990] H. F. Korth and G. Speegle, "Long Duration Transactions in Software Design Projects", In *Proc. of the International Conf. on Data Engineering* (1990), pages 568–575.

[Korth and Speegle 1994] H. F. Korth and G. Speegle, "Formal Aspects of Concurrency Control in Long Duration Transaction Systems Using the NT/PV Model", *ACM Transactions on Database Systems*, Volume 19, Number 3 (1994), pages 492–535.

[Krishnaprasad et al. 2004] M. Krishnaprasad, Z. Liu, A. Manikutty, J. W. Warner, V. Arora, and S. Kotsovolos, "Query Rewrite for XML in Oracle XML DB", In *Proc. of the International Conf. on Very Large Databases* (2004), pages 1122–1133.

[Kung and Lehman 1980] H. T. Kung and P. L. Lehman, "Concurrent Manipulation of Binary Search Trees", *ACM Transactions on Database Systems*, Volume 5, Number 3 (1980), pages 339–353.

[Kung and Robinson 1981] H. T. Kung and J. T. Robinson, "Optimistic Concurrency Control", *ACM Transactions on Database Systems*, Volume 6, Number 2 (1981), pages 312–326.

[Kurose and Ross 2005] J. Kurose and K. Ross, *Computer Networking — A Top-Down Approach Featuring the Internet*, 3rd edition, Addison Wesley (2005).

[Lahiri et al. 2001] T. Lahiri, A. Ganesh, R. Weiss, and A. Joshi, "Fast-Start: Quick Fault Recovery in Oracle", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2001).

[Lam and Kuo 2001] K.-Y. Lam and T.-W. Kuo, editors, *Real-Time Database Systems*, Kluwer Academic Publishers (2001).

[Lamb et al. 1991] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, "The ObjectStore Database System", *Communications of the ACM*, Volume 34, Number 10 (1991), pages 51–63.

[Lamport 1978] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, Volume 21, Number 7 (1978), pages 558–565.

[Lampson and Sturgis 1976] B. Lampson and H. Sturgis, "Crash Recovery in a Distributed Data Storage System", Technical report, Computer Science Laboratory, Xerox Palo Alto Research Center,Palo Alto (1976).

[Lecluse et al. 1988] C. Lecluse, P. Richard, and F. Velez, "O2: An Object-Oriented Data Model", In *Proc. of the International Conf. on Very Large Databases* (1988), pages 424–433.

[Lehman and Yao 1981] P. L. Lehman and S. B. Yao, "Efficient Locking for Concurrent Operations on B-trees", *ACM Transactions on Database Systems*, Volume 6, Number 4 (1981), pages 650–670.

[Lehner et al. 2000] W. Lehner, R. Sidle, H. Pirahesh, and R. Cochrane, "Maintenance of Automatic Summary Tables", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2000), pages 512–513.

[Lindsay et al. 1980] B. G. Lindsay, P. G. Selinger, C. Galtieri, J. N. Gray, R. A. Lorie, T. G. Price, G. R. Putzolu, I. L. Traiger, and B. W. Wade. "Notes on Distributed Databases", In Draffen and Poole, editors, *Distributed Data Bases*, pages 247–284. Cambridge University Press (1980).

[Litwin 1978] W. Litwin, "Virtual Hashing: A Dynamically Changing Hashing", In *Proc. of the International Conf. on Very Large Databases* (1978), pages 517–523.

[Litwin 1980] W. Litwin, "Linear Hashing: A New Tool for File and Table Addressing", In *Proc. of the International Conf. on Very Large Databases* (1980), pages 212–223.

[Litwin 1981] W. Litwin, "Trie Hashing", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1981), pages 19–29.

[Lo and Ravishankar 1996] M.-L. Lo and C. V. Ravishankar, "Spatial Hash-Joins", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[Loeb 1998] L. Loeb, *Secure Electronic Transactions: Introduction and Technical Reference*, Artech House (1998).

[Lomet 1981] D. G. Lomet, "Digital B-trees", In *Proc. of the International Conf. on Very Large Databases* (1981), pages 333–344.

[Lomet et al. 2009] D. Lomet, A. Fekete, G. Weikum, and M. Zwilling, "Unbundling Transaction Services in the Cloud", In *Proc. 4th Biennial Conference on Innovative Data Systems Research* (2009).

[Lu et al. 1991] H. Lu, M. Shan, and K. Tan, "Optimization of Multi-Way Join Queries for Parallel Execution", In *Proc. of the International Conf. on Very Large Databases* (1991), pages 549–560.

[Lynch and Merritt 1986] N. A. Lynch and M. Merritt, "Introduction to the Theory of Nested Transactions", In *Proc. of the International Conf. on Database Theory* (1986).

[Lynch et al. 1988] N. A. Lynch, M. Merritt, W. Weihl, and A. Fekete, "A Theory of Atomic Transactions", In *Proc. of the International Conf. on Database Theory* (1988), pages 41–71.

[Maier 1983] D. Maier, *The Theory of Relational Databases*, Computer Science Press (1983).

[Manning et al. 2008] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*, Cambridge University Press (2008).

[Martin et al. 1989] J. Martin, K. K. Chapman, and J. Leben, *DB2, Concepts, Design, and Programming*, Prentice Hall (1989).

[Mattison 1996] R. Mattison, *Data Warehousing: Strategies, Technologies, and Techniques*, McGraw Hill (1996).

[McHugh and Widom 1999] J. McHugh and J. Widom, "Query Optimization for XML", In *Proc. of the International Conf. on Very Large Databases* (1999), pages 315–326.

[Mehrotra et al. 1991] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz, "Non-Serializable Executions in Heterogeneous Distributed Database Systems", In *Proc. of the International Conf. on Parallel and Distributed Information Systems* (1991).

[Mehrotra et al. 2001] S. Mehrotra, R. Rastogi, Y. Breitbart, H. F. Korth, and A. Silberschatz, "Overcoming Heterogeneity and Autonomy in Multidatabase Systems.", *Inf. Comput.*, Volume 167, Number 2 (2001), pages 137–172.

[Melnik et al. 2007] S. Melnik, A. Adya, and P. A. Bernstein, "Compiling mappings to bridge applications and databases", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2007), pages 461–472.

[Melton 2002] J. Melton, *Advanced SQL:1999 – Understanding Object-Relational and Other Advanced Features*, Morgan Kaufmann (2002).

[Melton and Eisenberg 2000] J. Melton and A. Eisenberg, *Understanding SQL and Java Together : A Guide to SQLJ, JDBC, and Related Technologies*, Morgan Kaufmann (2000).

[Melton and Simon 1993] J. Melton and A. R. Simon, *Understanding The New SQL: A Complete Guide*, Morgan Kaufmann (1993).

[Melton and Simon 2001] J. Melton and A. R. Simon, *SQL:1999, Understanding Relational Language Components*, Morgan Kaufmann (2001).

[Microsoft 1997] Microsoft, *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference*, Microsoft Press (1997).

[Mistry et al. 2001] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham, "Materialized View Selection and Maintenance Using Multi-Query Optimization", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2001).

[Mitchell 1997] T. M. Mitchell, *Machine Learning*, McGraw Hill (1997).

[Mohan 1990a] C. Mohan, "ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operations on B-Tree indexes", In *Proc. of the International Conf. on Very Large Databases* (1990), pages 392–405.

[Mohan 1990b] C. Mohan, "Commit-LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems", In *Proc. of the International Conf. on Very Large Databases* (1990), pages 406–418.

[Mohan 1993] C. Mohan, "IBM's Relational Database Products:Features and Technologies", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993).

[Mohan and Levine 1992] C. Mohan and F. Levine, "ARIES/IM:An Efficient and High-Concurrency Index Management Method Using Write-Ahead Logging", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1992).

[Mohan and Lindsay 1983] C. Mohan and B. Lindsay, "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions", In *Proc. of the ACM Symposium on Principles of Distributed Computing* (1983).

[Mohan and Narang 1992] C. Mohan and I. Narang, "Efficient Locking and Caching of Data in the Multisystem Shared Disks Transaction Environment", In *Proc. of the International Conf. on Extending Database Technology* (1992).

[Mohan and Narang 1994] C. Mohan and I. Narang, "ARIES/CSA: A Method for Database Recovery in Client-Server Architectures", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1994), pages 55–66.

[Mohan et al. 1986] C. Mohan, B. Lindsay, and R. Obermarck, "Transaction Management in the R* Distributed Database Management System", *ACM Transactions on Database Systems*, Volume 11, Number 4 (1986), pages 378–396.

[Mohan et al. 1992] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", *ACM Transactions on Database Systems*, Volume 17, Number 1 (1992).

[Moss 1985] J. E. B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press (1985).

[Moss 1987] J. E. B. Moss, "Log-Based Recovery for Nested Transactions", In *Proc. of the International Conf. on Very Large Databases* (1987), pages 427–432.

[Murthy and Banerjee 2003] R. Murthy and S. Banerjee, "XML Schemas in Oracle XML DB", In *Proc. of the International Conf. on Very Large Databases* (2003), pages 1009–1018.

[Nakayama et al. 1984] T. Nakayama, M. Hirakawa, and T. Ichikawa, "Architecture and Algorithm for Parallel Execution of a Join Operation", In *Proc. of the International Conf. on Data Engineering* (1984).

[Ng and Han 1994] R. T. Ng and J. Han, "Efficient and Effective Clustering Methods for Spatial Data Mining", In *Proc. of the International Conf. on Very Large Databases* (1994).

[NIST 1993] NIST, "Integration Definition for Information Modeling (IDEF1X)", Technical Report Federal Information Processing Standards Publication 184, National Institute of Standards and Technology (NIST), Available at www.idef.com/Downloads/pdf/Idef1x.pdf (1993).

[Nyberg et al. 1995] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. B. Lomet, "AlphaSort: A Cache-Sensitive Parallel External Sort", *VLDB Journal*, Volume 4, Number 4 (1995), pages 603–627.

[O'Neil and O'Neil 2000] P. O'Neil and E. O'Neil, *Database: Principles, Programming, Performance*, 2nd edition, Morgan Kaufmann (2000).

[O'Neil and Quass 1997] P. O'Neil and D. Quass, "Improved Query Performance with Variant Indexes", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1997).

[O'Neil et al. 2004] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury, "ORDPATHs: Insert-Friendly XML Node Labels", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2004), pages 903–908.

[Ooi and S. Parthasarathy 2009] B. C. Ooi and e. S. Parthasarathy, "Special Issue on Data Management on Cloud Computing Platforms", *Data Engineering Bulletin*, Volume 32, Number 1 (2009).

[Orenstein 1982] J. A. Orenstein, "Multidimensional Tries Used for Associative Searching", *Information Processing Letters*, Volume 14, Number 4 (1982), pages 150–157.

[Ozcan et al. 1997] F. Ozcan, S. Nural, P. Koksal, C. Evrendilek, and A. Dogac, "Dynamic Query Optimization in Multidatabases", *Data Engineering Bulletin*, Volume 20, Number 3 (1997), pages 38–45.

[Ozden et al. 1994] B. Ozden, A. Biliris, R. Rastogi, and A. Silberschatz, "A Low-cost Storage Server for a Movie on Demand Database", In *Proc. of the International Conf. on Very Large Databases* (1994).

[Ozden et al. 1996a] B. Ozden, R. Rastogi, P. Shenoy, and A. Silberschatz, "Fault-Tolerant Architectures for Continuous Media Servers", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[Ozden et al. 1996b] B. Ozden, R. Rastogi, and A. Silberschatz, "On the Design of a Low-Cost Video-on-Demand Storage System", *Multimedia Systems Journal*, Volume 4, Number 1 (1996), pages 40–54.

[Ozsoyoglu and Snodgrass 1995] G. Ozsoyoglu and R. Snodgrass, "Temporal and Real-Time Databases: A Survey", *IEEE Transactions on Knowledge and Data Engineering*, Volume 7, Number 4 (1995), pages 513–532.

[Ozsu and Valduriez 1999] T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems*, 2nd edition, Prentice Hall (1999).

[Padmanabhan et al. 2003] S. Padmanabhan, B. Bhattacharjee, T. Malkemus, L. Cranston, and M. Huras, "Multi-Dimensional Clustering: A New Data Layout Scheme in DB2", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2003), pages 637–641.

[Pal et al. 2004] S. Pal, I. Cseri, G. Schaller, O. Seeliger, L. Giakoumakis, and V. Zolotov, "Indexing XML Data Stored in a Relational Database", In *Proc. of the International Conf. on Very Large Databases* (2004), pages 1134–1145.

[Pang et al. 1995] H.-H. Pang, M. J. Carey, and M. Livny, "Multiclass Scheduling in Real-Time Database Systems", *IEEE Transactions on Knowledge and Data Engineering*, Volume 2, Number 4 (1995), pages 533–551.

[Papakonstantinou et al. 1996] Y. Papakonstantinou, A. Gupta, and L. Haas, "Capabilities-Based Query Rewriting in Mediator Systems", In *Proc. of the International Conf. on Parallel and Distributed Information Systems* (1996).

[Parker et al. 1983] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline, "Detection of Mutual Inconsistency in Distributed Systems", *IEEE Transactions on Software Engineering*, Volume 9, Number 3 (1983), pages 240–246.

[Patel and DeWitt 1996] J. Patel and D. J. DeWitt, "Partition Based Spatial-Merge Join", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[Patterson 2004] D. P. Patterson, "Latency Lags Bandwidth", *Communications of the ACM*, Volume 47, Number 10 (2004), pages 71–75.

[Patterson et al. 1988] D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1988), pages 109–116.

[Pellenkoft et al. 1997] A. Pellenkoft, C. A. Galindo-Legaria, and M. Kersten, "The Complexity of Transformation-Based Join Enumeration", In *Proc. of the International Conf. on Very Large Databases* (1997), pages 306–315.

[Peterson and Davie 2007] L. L. Peterson and B. S. Davie, *Computer Networks: a Systems Approach*, Morgan Kaufmann Publishers Inc. (2007).

[Pless 1998] V. Pless, *Introduction to the Theory of Error-Correcting Codes*, 3rd edition, John Wiley and Sons (1998).

[Poe 1995] V. Poe, *Building a Data Warehouse for Decision Support*, Prentice Hall (1995).

[Polyzois and Garcia-Molina 1994] C. Polyzois and H. Garcia-Molina, "Evaluation of Remote Backup Algorithms for Transaction-Processing Systems", *ACM Transactions on Database Systems*, Volume 19, Number 3 (1994), pages 423–449.

[Poosala et al. 1996] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita, "Improved Histograms for Selectivity Estimation of Range Predicates", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996), pages 294–305.

[Popek et al. 1981] G. J. Popek, B. J. Walker, J. M. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel, "LOCUS: A Network Transparent, High Reliability Distributed System", In *Proc. of the Eighth Symposium on Operating System Principles* (1981), pages 169–177.

[Pöss and Potapov 2003] M. Pöss and D. Potapov, "Data Compression in Oracle", In *Proc. of the International Conf. on Very Large Databases* (2003), pages 937–947.

[Rahm 1993] E. Rahm, "Empirical Performance Evaluation of Concurrency and Coherency Control Protocols for Database Sharing Systems", *ACM Transactions on Database Systems*, Volume 8, Number 2 (1993).

[Ramakrishna and Larson 1989] M. V. Ramakrishna and P. Larson, "File Organization Using Composite Perfect Hashing", *ACM Transactions on Database Systems*, Volume 14, Number 2 (1989), pages 231–263.

[Ramakrishnan and Gehrke 2002] R. Ramakrishnan and J. Gehrke, *Database Management Systems*, 3rd edition, McGraw Hill (2002).

[Ramakrishnan and Ullman 1995] R. Ramakrishnan and J. D. Ullman, "A Survey of Deductive Database Systems", *Journal of Logic Programming*, Volume 23, Number 2 (1995), pages 125–149.

[Ramakrishnan et al. 1992] R. Ramakrishnan, D. Srivastava, and S. Sudarshan, *Controlling the Search in Bottom-up Evaluation* (1992).

[Ramesh et al. 1989] R. Ramesh, A. J. G. Babu, and J. P. Kincaid, "Index Optimization: Theory and Experimental Results", *ACM Transactions on Database Systems*, Volume 14, Number 1 (1989), pages 41–74.

[Rangan et al. 1992] P. V. Rangan, H. M. Vin, and S. Ramanathan, "Designing an On-Demand Multimedia Service", *Communications Magazine*, Volume 1, Number 1 (1992), pages 56–64.

[Rao and Ross 2000] J. Rao and K. A. Ross, "Making B+-Trees Cache Conscious in Main Memory", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2000), pages 475–486.

[Rathi et al. 1990] A. Rathi, H. Lu, and G. E. Hedrick, "Performance Comparison of Extendable Hashing and Linear Hashing Techniques", In *Proc. ACM SIGSmall/PC Symposium on Small Systems* (1990), pages 178–185.

[Reed 1983] D. Reed, "Implementing Atomic Actions on Decentralized Data", *Transactions on Computer Systems*, Volume 1, Number 1 (1983), pages 3–23.

[Revesz 2002] P. Revesz, *Introduction to Constraint Databases*, Springer Verlag (2002).

[Richardson et al. 1987] J. Richardson, H. Lu, and K. Mikkilineni, "Design and Evaluation of Parallel Pipelined Join Algorithms", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1987).

[Rivest 1976] R. L. Rivest, "Partial Match Retrieval Via the Method of Superimposed Codes", *SIAM Journal of Computing*, Volume 5, Number 1 (1976), pages 19–50.

[Robinson 1981] J. Robinson, "The k-d-B Tree: A Search Structure for Large Multidimensional Indexes", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1981), pages 10–18.

[Roos 2002] R. M. Roos, *Java Data Objects*, Pearson Education (2002).

[Rosch 2003] W. L. Rosch, *The Winn L. Rosch Hardcore Bible*, 6th edition, Que (2003).

[Rosenthal and Reiner 1984] A. Rosenthal and D. Reiner, "Extending the Algebraic Framework of Query Processing to Handle Outerjoins", In *Proc. of the International Conf. on Very Large Databases* (1984), pages 334–343.

[Ross 1990] K. A. Ross, "Modular Stratification and Magic Sets for DATALOG Programs with Negation", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990).

[Ross 1999] S. M. Ross, *Introduction to Probability and Statistics for Engineers and Scientists*, Harcourt/Academic Press (1999).

[Ross and Srivastava 1997] K. A. Ross and D. Srivastava, "Fast Computation of Sparse Datacubes", In *Proc. of the International Conf. on Very Large Databases* (1997), pages 116–125.

[Ross et al. 1996] K. Ross, D. Srivastava, and S. Sudarshan, "Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[Rothermel and Mohan 1989] K. Rothermel and C. Mohan, "ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions", In *Proc. of the International Conf. on Very Large Databases* (1989), pages 337–346.

[Roy et al. 2000] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobhe, "Efficient and Extensible Algorithms for Multi-Query Optimization", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2000).

[Rusinkiewicz and Sheth 1995] M. Rusinkiewicz and A. Sheth, "Specification and Execution of Transactional Workflows", In *Kim [1995]*, pages 592–620 (1995).

[Rustin 1972] R. Rustin, *Data Base Systems*, Prentice Hall (1972).

[Rys 2001] M. Rys, "Bringing the Internet to Your Database: Using SQL Server 2000 and XML to Build Loosely-Coupled Systems", In *Proc. of the International Conf. on Data Engineering* (2001), pages 465–472.

[Rys 2003] M. Rys, "XQuery and Relational Database Systems", In H. Katz, editor, *XQuery From the Experts*, pages 353–391. Addison Wesley (2003).

[Rys 2004] M. Rys, "What's New in FOR XML in Microsoft SQL Server 2005", http://msdn.microsoft.com/en-us/library/ms345137(SQL.90).aspx (2004).

[Sagiv and Yannakakis 1981] Y. Sagiv and M. Yannakakis, "Equivalence among Relational Expressions with the Union and Difference Operators", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1981).

[Salton 1989] G. Salton, *Automatic Text Processing*, Addison Wesley (1989).

[Samet 1990] H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison Wesley (1990).

[Samet 1995a] H. Samet, "General Research Issues in Multimedia Database Systems", *ACM Computing Survey*, Volume 27, Number 4 (1995), pages 630–632.

[Samet 1995b] H. Samet. "Spatial Data Structures", In *Kim [1995]*, pages 361–385 (1995).

[Samet 2006] H. Samet, *Foundations of Multidimensional and Metric Data Structures*, Morgan Kaufmann (2006).

[Samet and Aref 1995] H. Samet and W. Aref. "Spatial Data Models and Query Processing", In *Kim [1995]*, pages 338–360 (1995).

[Sanders 1998] R. E. Sanders, *ODBC 3.5 Developer's Guide*, McGraw Hill (1998).

[Sarawagi 2000] S. Sarawagi, "User-Adaptive Exploration of Multidimensional Data", In *Proc. of the International Conf. on Very Large Databases* (2000), pages 307–316.

[Sarawagi et al. 2002] S. Sarawagi, A. Bhamidipaty, A. Kirpal, and C. Mouli, "ALIAS: An Active Learning Led Interactive Deduplication System", In *Proc. of the International Conf. on Very Large Databases* (2002), pages 1103–1106.

[Schlageter 1981] G. Schlageter, "Optimistic Methods for Concurrency Control in Distributed Database Systems", In *Proc. of the International Conf. on Very Large Databases* (1981), pages 125–130.

[Schneider 1982] H. J. Schneider, "Distributed Data Bases", In *Proc. of the International Symposium on Distributed Databases* (1982).

[Selinger et al. 1979] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access Path Selection in a Relational Database System", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1979), pages 23–34.

[Sellis 1988] T. K. Sellis, "Multiple Query Optimization", *ACM Transactions on Database Systems*, Volume 13, Number 1 (1988), pages 23–52.

[Sellis et al. 1987] T. K. Sellis, N. Roussopoulos, and C. Faloutsos, "TheR⁺-Tree: A Dynamic Index for Multi-Dimensional Objects", In *Proc. of the International Conf. on Very Large Databases* (1987), pages 507–518.

[Seshadri et al. 1996] P. Seshadri, H. Pirahesh, and T. Y. C. Leung, "Complex Query Decorrelation", In *Proc. of the International Conf. on Data Engineering* (1996), pages 450–458.

[Shafer et al. 1996] J. C. Shafer, R. Agrawal, and M. Mehta, "SPRINT: A Scalable Parallel Classifier for Data Mining", In *Proc. of the International Conf. on Very Large Databases* (1996), pages 544–555.

[Shanmugasundaram et al. 1999] J. Shanmugasundaram, G. He, K. Tufte, C. Zhang, D. DeWitt, and J. Naughton, "Relational Databases for Querying XML Documents: Limitations and Opportunities", In *Proc. of the International Conf. on Very Large Databases* (1999).

[Shapiro 1986] L. D. Shapiro, "Join Processing in Database Systems with Large Main Memories", *ACM Transactions on Database Systems*, Volume 11, Number 3 (1986), pages 239–264.

[Shasha and Bonnet 2002] D. Shasha and P. Bonnet, *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*, Morgan Kaufmann (2002).

[Silberschatz 1982] A. Silberschatz, "A Multi-Version Concurrency Control Scheme With No Rollbacks", In *Proc. of the ACM Symposium on Principles of Distributed Computing* (1982), pages 216–223.

[Silberschatz and Kedem 1980] A. Silberschatz and Z. Kedem, "Consistency in Hierarchical Database Systems", *Journal of the ACM*, Volume 27, Number 1 (1980), pages 72–80.

[Silberschatz et al. 1990] A. Silberschatz, M. R. Stonebraker, and J. D. Ullman, "Database Systems: Achievements and Opportunities", *ACM SIGMOD Record*, Volume 19, Number 4 (1990).

[Silberschatz et al. 1996] A. Silberschatz, M. Stonebraker, and J. Ullman, "Database Research: Achievements and Opportunities into the 21st Century", Technical Report CS-TR-96-1563, Department of Computer Science, Stanford University, Stanford (1996).

[Silberschatz et al. 2008] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 8th edition, John Wiley and Sons (2008).

[Simmen et al. 1996] D. Simmen, E. Shekita, and T. Malkemus, "Fundamental Techniques for Order Optimization", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996), pages 57–67.

[Skeen 1981] D. Skeen, "Non-blocking Commit Protocols", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1981), pages 133–142.

[Soderland 1999] S. Soderland, "Learning Information Extraction Rules for Semi-structured and Free Text", *Machine Learning*, Volume 34, Number 1–3 (1999), pages 233–272.

[Soo 1991] M. Soo, "Bibliography on Temporal Databases", *ACM SIGMOD Record*, Volume 20, Number 1 (1991), pages 14–23.

[SQL/XML 2004] SQL/XML. "ISO/IEC 9075-14:2003, Information Technology: Database languages: SQL-Part 14: XML-Related Specifications (SQL/XML)" (2004).

[Srikant and Agrawal 1996a] R. Srikant and R. Agrawal, "Mining Quantitative Association Rules in Large Relational Tables", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[Srikant and Agrawal 1996b] R. Srikant and R. Agrawal, "Mining Sequential Patterns: Generalizations and Performance Improvements", In *Proc. of the International Conf. on Extending Database Technology* (1996), pages 3–17.

[Stam and Snodgrass 1988] R. Stam and R. Snodgrass, "A Bibliography on Temporal Databases", *IEEE Transactions on Knowledge and Data Engineering*, Volume 7, Number 4 (1988), pages 231–239.

[Stinson 2002] B. Stinson, *PostgreSQL Essential Reference*, New Riders (2002).

[Stonebraker 1986] M. Stonebraker, "Inclusion of New Types in Relational Database Systems", In *Proc. of the International Conf. on Data Engineering* (1986), pages 262–269.

[Stonebraker and Rowe 1986] M. Stonebraker and L. Rowe, "The Design of POSTGRES", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1986).

[Stonebraker et al. 1989] M. Stonebraker, P. Aoki, and M. Seltzer, "Parallelism in XPRS", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1989).

[Stonebraker et al. 1990] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos, "On Rules, Procedure, Caching and Views in Database Systems", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990), pages 281–290.

[Stuart et al. 1984] D. G. Stuart, G. Buckley, and A. Silberschatz, "A Centralized Deadlock Detection Algorithm", Technical report, Department of Computer Sciences, University of Texas, Austin (1984).

[Tanenbaum 2002] A. S. Tanenbaum, *Computer Networks*, 4th edition, Prentice Hall (2002).

[Tansel et al. 1993] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, *Temporal Databases: Theory, Design and Implementation*, Benjamin Cummings (1993).

[Teorey et al. 1986] T. J. Teorey, D. Yang, and J. P. Fry, "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model", *ACM Computing Survey*, Volume 18, Number 2 (1986), pages 197–222.

[Thalheim 2000] B. Thalheim, *Entity-Relationship Modeling : Foundations of Database Technology*, Springer Verlag (2000).

[Thomas 1996] S. A. Thomas, *IPng and the TCP/IP Protocols: Implementing the Next Generation Internet*, John Wiley and Sons (1996).

[Traiger et al. 1982] I. L. Traiger, J. N. Gray, C. A. Galtieri, and B. G. Lindsay, "Transactions and Consistency in Distributed Database Management Systems", *ACM Transactions on Database Systems*, Volume 7, Number 3 (1982), pages 323–342.

[Tyagi et al. 2003] S. Tyagi, M. Vorburger, K. McCammon, and H. Bobzin, *Core Java Data Objects*, prenticehall (2003).

[Umar 1997] A. Umar, *Application (Re)Engineering : Building Web-Based Applications and Dealing With Legacies*, Prentice Hall (1997).

[UniSQL 1991] *UniSQL/X Database Management System User's Manual: Release 1.2*, UniSQL, Inc. (1991).

[Verhofstad 1978] J. S. M. Verhofstad, "Recovery Techniques for Database Systems", *ACM Computing Survey*, Volume 10, Number 2 (1978), pages 167–195.

[Vista 1998] D. Vista, "Integration of Incremental View Maintenance into Query Optimizers", In *Proc. of the International Conf. on Extending Database Technology* (1998).

[Vitter 2001] J. S. Vitter, "External Memory Algorithms and Data Structures: Dealing with Massive Data", *ACM Computing Surveys*, Volume 33, (2001), pages 209–271.

[Walsh et al. 2007] N. Walsh et al. "XQuery 1.0 and XPath 2.0 Data Model", http://www.w3.org/TR/xpath-datamodel. currently a W3C Recommendation (2007).

[Walton et al. 1991] C. Walton, A. Dale, and R. Jenevein, "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins", In *Proc. of the International Conf. on Very Large Databases* (1991).

[Weikum 1991] G. Weikum, "Principles and Realization Strategies of Multilevel Transaction Management", *ACM Transactions on Database Systems*, Volume 16, Number 1 (1991).

[Weikum et al. 1990] G. Weikum, C. Hasse, P. Broessler, and P. Muth, "Multi-Level Recovery", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990), pages 109–123.

[Wilschut et al. 1995] A. N. Wilschut, J. Flokstra, and P. M. Apers, "Parallel Evaluation of Multi-Join Queries", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), pages 115–126.

[Witten and Frank 1999] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, Morgan Kaufmann (1999).

[Witten et al. 1999] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*, 2nd edition, Morgan Kaufmann (1999).

[Wolf 1991] J. Wolf, "An Effective Algorithm for Parallelizing Hash Joins in the Presence of Data Skew", In *Proc. of the International Conf. on Data Engineering* (1991).

[Wu and Buchmann 1998] M. Wu and A. Buchmann, "Encoded Bitmap Indexing for Data Warehouses", In *Proc. of the International Conf. on Data Engineering* (1998).

[Wu et al. 2003] Y. Wu, J. M. Patel, and H. V. Jagadish, "Structural Join Order Selection for XML Query Optimization", In *Proc. of the International Conf. on Data Engineering* (2003).

[X/Open 1991] *X/Open Snapshot: X/Open DTP: XA Interface*. X/Open Company, Ltd. (1991).

[Yan and Larson 1995] W. P. Yan and P. A. Larson, "Eager Aggregation and Lazy Aggregation", In *Proc. of the International Conf. on Very Large Databases* (1995).

[Yannakakis et al. 1979] M. Yannakakis, C. H. Papadimitriou, and H. T. Kung, "Locking Protocols: Safety and Freedom from Deadlock", In *Proc. of the IEEE Symposium on the Foundations of Computer Science* (1979), pages 286–297.

[Zaharioudakis et al. 2000] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata, "Answering Complex SQL Queries using Automatic Summary Tables", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2000), pages 105–116.

[Zeller and Gray 1990] H. Zeller and J. Gray, "An Adaptive Hash Join Algorithm for Multiuser Environments", In *Proc. of the International Conf. on Very Large Databases* (1990), pages 186–197.

[Zhang et al. 1996] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: An Efficient Data Clustering Method for Very Large Databases", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996), pages 103–114.

[Zhou and Ross 2004] J. Zhou and K. A. Ross, "Buffering Database Operations for Enhanced Instruction Cache Performance", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (2004), pages 191–202.

[Zhuge et al. 1995] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom, "View maintenance in a warehousing environment", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), pages 316–327.

[Ziauddin et al. 2008] M. Ziauddin, D. Das, H. Su, Y. Zhu, and K. Yagoub, "Optimizer plan change management: improved stability and performance in Oracle 11g", *Proceedings of the VLDB Endowment*, Volume 1, Number 2 (2008), pages 1346–1355.

[Zikopoulos et al. 2004] P. Zikopoulos, G. Baklarz, D. deRoos, and R. B. Melnyk, *DB2 Version 8: The Official Guide*, IBM Press (2004).

[Zikopoulos et al. 2007] P. Zikopoulos, G. Baklarz, L. Katsnelson, and C. Eaton, *IBM DB2 9 New Features*, McGraw Hill (2007).

[Zikopoulos et al. 2009] P. Zikopoulos, B. Tassi, G. Baklarz, and C. Eaton, *Break Free with DB2 9.7*, McGraw Hill (2009).

[Zilio et al. 2004] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden, "DB2 Design Advisor: Integrated Automatic Physical Database Design", In *Proc. of the International Conf. on Very Large Databases* (2004), pages 1087–1097.

[Zloof 1977] M. M. Zloof, "Query-by-Example: A Data Base Language", *IBM Systems Journal*, Volume 16, Number 4 (1977), pages 324–343.

索　引

索引中的页码为英文原书页码，与书中页边标注的页码一致。

2PC(两阶段提交). 参见 two-phase commit
3NF(第三范式). 参见 third normal form
3PC(三阶段提交). 参见 three-phase commit

abstract data type(抽象数据类型), 1127
access path(存取路径), 542
ACID property(ACID 特性). 参见
  atomicity(原子性); consistency(一致性); durability(持久性); isolation(隔离性)
Active Server Pages(ASP), 397
ADO. NET, 169, 395, 1249, 1253
Advanced Encryption Standard(AES)(扩展加密标准), 412-413
agglomerative clustering(凝聚聚类), 907-908
aggregate function(聚集函数)
  basic(基本的), 85-86
  Boolean value and(布尔值), 89-90
  SQL(结构化查询语言), 84
  fusion(融合), 960
  with grouping(分组), 86-88
  having clause(having 子句), 88-89
  null value and(空值), 89-90
aggregation(聚集)
  advanced feature of(高级特性), 192-197
  alternative notation for(替代记号), 304-310
  entity-relationship(E-R) model(实体-联系模型), 301-302, 304
  IBM DB2 and, 1209-1210
  intraoperation parallelism and(操作内并行), 811
  .NET Common Language Runtime(CLR)(.NET 公共语言运行库)and, 1257-1258
  OLAP and(联机分析处理), 197-209
  PostgreSQL and, 1153
  query optimization and(查询优化), 597
  query processing and(查询处理), 566-567
  ranking and(分级), 192-195
  relational algebra and(关系代数), 235-239
  representation of(表示), 304
  view maintenance and(视图维护), 610-611
  windowing(分窗), 195-197
Ajax, 390-391, 398, 867
alias(别名), 75, 355, 829, 872-873, 1229
alter table(修改表), 63, 129
alter trigger(修改触发器), 185
alter type(修改类型), 140
American National Standards Institute(ANSI)(美国国家标准化组织), 57, 1051
analysis pass(分析过程), 753
analytic workspace(分析工作区), 1161
and operation(与运算), 66, 83-84, 1174
and keyword(and 关键字), 92n8
Apache, 386, 399, 426, 980
Apple Macintosh OS X(苹果 Macintosh OS X), 1124
application design(应用程序设计), 418
  application architecture(应用程序体系结构) and, 391-396
  authentication and(鉴定), 405-407
  business-logic layer(业务逻辑层) and, 391-392
  client-server architecture(客户-服务器体系结构) and, 376-377
  common gateway interface(CGI)(通用网关接口), 380-381
  cookie and(一小段包含标识信息的文本), 382-385
  data access layer and(数据访问层), 391, 393, 395
  disconnected operation(断连操作)and, 395-396
  encryption(加密)and, 411-417
  HyperText Markup Language(HTML) and(超文本标记语言), 378-380
  HyperText Transfer Protocol(HTTP) and(超文本传输协议), 377-381, 383, 395, 404-406, 417
  Java Server Pages(JSP) and, 377, 383-391
  performance and(性能), 400-402
  rapid application development and(快速应用开发), 396-400
  security and(安全), 402-417
  servlet and(Java 服务器端程序), 383-391
  three-layer architecture and(三层体系结构), 318
  TP-monitor and(事务处理监控程序), 1095-1096
Uniform Resource Locator(URL)(统一资源定位符),

377-378
user interface and(用户界面), 375-377
World Wide Web and(万维网), 377-382
application development(应用程序开发)
performance benchmark and(性能基准), 1045-1048
performance tuning and(性能调整), 1029-1045
set orientation and(面向集合), 1030-1031
standardization and(标准化), 1051-1056
testing application and(应用程序测试), 1048-1051
update and(更新), 1030-1033
application migration(应用程序移植), 1050-1051
Application Program Interface (API)(应用程序接口)
ADO. NET, 169, 1054
application design and(应用程序设计), 383-386, 395
C++, 1054
customized map and(定制映图), 1068, 1070
DOM(文档对象模型), 1020
IBM DB2, 1196
Java, 158-166, 213, 383-386, 1018, 1030
LDAP(轻量级目录访问协议), 874
Microsoft SQL Server, 1229, 1245, 1248-1250, 1253-1255, 1265, 1267
ODBC(开放数据库互联), 166-169, 1053
PostgreSQL, 1125
SAX, 1020
standard for(标准), 1051-1056
system architecture and(系统体系结构), 772
XML(可扩展标记语言), 985, 1008-1009
apply operator(apply 运算符), 1230-1231
architecture(体系结构), 767
business logic and(业务逻辑), 25, 1158, 1221-1222, 1228, 1232, 1253, 1263-1267
centralized(集中式), 769-771
client-server(客户-服务器), 771-772
cloud-based(基于云的), 777
data server(数据服务器), 773, 775-777
data warehouse(数据仓库), 889-891
distributed system(分布式系统), 784-788
hierarchical(层次的), 781, 784
hypercube(超立方体), 781
mesh(网格), 780-781
network type and(网络类型), 788-791
parallel database(并行数据库), 797-820
parallel system(并行系统), 777-784
process monitor(进程监控), 774
server system(服务器系统), 772-777
shared-disk(共享硬盘), 781, 783, 789
shared memory(共享内存), 781-783
shared-nothing(无共享), 781, 783-784, 803
shared server(共享服务器), 1185
single-user system(单用户系统), 770-771
source-driven(源驱动的), 890
Storage-Area Network (SAN)(存储区域网), 789
thread pooling and(线程池), 1246
three-tier(三层), 25
TP-monitor(事务处理监控程序), 1092-1095
transaction-server(事务服务器), 773-775
two-phase commit protocol (2PC)(两阶段提交协议), 786-788
two-tier(两层), 24-25
wide-area network(WAN)(广域网), 788, 790-791
ARIES, 1146
analysis pass(分析阶段), 753
compensation log record(CLR)(补偿日志记录), 751-752, 754
data structure(数据结构), 751-753
dirty page table(脏页表), 750-755
fine-granted locking(细粒度封锁), 756
fuzzy checkpoint(模糊检查点), 750-752
log sequence number (LSN)(日志序列号), 750-755
nested top action(嵌套的顶层动作), 755-756
optimization and(优化), 756
physiological redo and(物理逻辑 redo), 750
recovery and(恢复), 751, 753, 756
redo pass(redo 阶段), 754
savepoint and(保存点), 756
transaction rollback and(事务回滚), 754-755
undo pass(undo 阶段), 754-755
Arpanet(阿帕网), 790
array type(数组类型), 956-961
asc expression(asc 表达式), 77-78
as clause(as 子句), 75-76
ASP. NET, 387
assertion(断言), 11, 135-136
assignment operation(赋值运算), 176, 217, 232, 1052
association(关联), 17, 43
entity set and(实体集), 290(参见 entity set)
relationship set and(关系集), 264-267, 308-309, 314

rule for(规则), 904-907
associative property(结合律), 584-585
Aster Data, 816
asymmetric-key encryption(非对称密钥加密), 412
atomic domain(原子域), 42
first normal form and(第一范式), 327-329
object-based database and(基于对象的数据库), 947
relational database design and(关系数据库设计), 327-329
atomicity(原子性), 4-5, 22-23, 104, 625
cascadeless schedule and(无级联其度), 647-648
commit protocol and(提交协议), 832-838
defined(定义的), 628
distributed transaction and(分布式事务), 830-832
isolation and(隔离性), 646-648
log record and(日志记录), 726-728, 730-734
recoverable schedule and(可恢复调度), 647
recovery system and(恢复系统), 726-735
storage structur and(存储结构), 632-633
workflow and(工作流), 1099-1100
attribute inheritance(属性继承), 298-299
attribute(属性)
atomic domain and(原子域), 327-329
classifier and(分类器), 896-897
closure of attribute set and(属性集闭包), 340-342
complex(复杂的), 277-278, 284-285
composite(复合的), 267
continuous valued(连续取值的), 898
decomposition and(分解), 329-338, 348-360
derived(导出的), 268
design issue(设计问题), 290-291
domain(域), 267
entity-relationship diagram and(实体联系图), 277-278
entity-relationship(E-R) model and(实体联系模型), 263, 267-269
entity set and(实体集), 283-286, 290-291
multiple-key access and(多码访问), 506-509
multivalued(多值), 267-268, 327-329
naming of(命名), 362-363
nesting(嵌套), 958-961
null value and(空值), 268-269
partitioned(划分), 896-897
placement of(布局), 294-295
search key and(搜索码), 476
simple(简单的), 267, 283-284
single-valued(单值的), 267-268
Unified Modeling Language(UML) and(统一建模语言), 308-310
uniquifier and(唯一化), 498-499
unnesting(解除嵌套), 958-961
value set of(值集), 267
xmlattribute xml(xml 属性), 1015
XML type and(可扩展标记语言类型), 990-998
attribute-value skew(属性值偏斜), 800-801
audit trail(审计追踪), 409-410
augmentation rule(增补律), 339
authentication(鉴定)
challenge-response system and(询问—回答系统), 415
digital certificate and(数字证书), 416-417
digital signature and(数字签名), 416-417
encryption and(加密), 415-417
security and(安全性), 405-407(参见 security)
single sign-on system and(单点登录系统), 406-407
smart card and(智能卡), 415-416
two-factor(两因素), 405-407
authorization(授权), 11, 21, 58
administrator and(管理员), 143
application-level(应用层), 407-409
check clause(check 子句), 148
database design and(数据库设计), 312
end-user information and(最终用户信息), 407-408
granting/revoking privileges(授予/收回权限), 29, 143-145, 149-150
lack of fine-grained(缺少细粒度), 408-409
role and(角色), 145-146
schema and(模式), 147-148
Security Assertion Markup Language(SAML) and(安全断言标记语言), 407
sql security invoker(sql security invoker 子句), 147
transfer of privilege(权限转移), 148-149
update and(更新), 147, 148
view (sql)(视图), 146-147
autonomy(自治), 858
availability(可用性)
CAP theorem and(CAP 定理), 852-853
consistency and(一致性), 852-853
coordinator selection(协调器的选择)and, 850-852
majority-based approach(基于多数的方法) and, 848-849
read one, write all approach(读一个、写所有方

法), 849-850
remote backup(远程备份)and, 850
robustness(健壮性)and, 847
site reintegration and(站点重建), 850
average response time(平均响应时间), 636
average seek time(平均寻道时间), 435, 540n2
avg expression(求平均表达式), 204
aggregate function and(聚集函数), 84-88
query processing and(查询处理), 566-567
relational algebra and(关系代数), 236

backup(备份), 186-187
application design and(应用程序设计), 415
distributed database and(分布式数据库), 839, 877
IBM DB2 and, 1215, 1218
Microsoft SQL Server and, 1223, 1227-1228, 1245, 1262
Oracle and, 1165, 1169, 1181-1183, 1187-1190
parallel database and(并行数据库), 816
recovery system and(恢复系统), 726-738
(参见 recovery system)
remote system for(远程系统), 756-759, 850, 1095-1096
system architecture and(系统体系结构), 770
transaction and(事务), 632, 1115
backup coordinator(备份协调器), 851
balanced range-partitioning(平衡的范围划分), 801
balanced tree(平衡树), 486
BASE property(基属性), 853
batch scaleup(批量型扩展比), 779
batch update(批量更新), 1030-1031
Bayesian classifier(贝叶斯分类器), 900-901
Bayes' theorem(贝叶斯定理), 900
BCNF(BCNF 范式). 参见 Boyce-Codd normal form
begin atomic... end, 128, 176, 181, 183-185
best split(最优划分), 897-899
biased protocol(有偏协议), 841
big-bang approach(大爆炸方法), 1050-1051
Bigtable, 862-867
binary split(二次划分), 898
bit-interleaved parity organization(位交错奇偶组织), 445
bit-level striping(比特级拆分), 442-444
bitmap index(位图索引), 507, 509, 531, 536
B+-tree and(B+树), 528
efficient operation of(高效操作), 527-528
existence(存在性), 526-527
Oracle and, 1166-1167
scan(扫描), 1153
sequential record and(顺序记录), 524-525
structure of(结构), 525-527
blind write(盲写), 687
blob(二进制大对象), 138, 166, 457, 502, 1013, 1198-1199, 1259
block-interleaved parity organization(块交错奇偶组织), 445-446
block-level striping(块级拆分), 443-444
block nested-loop join(块嵌套循环连接), 551-552
Boolean operation(布尔运算), 88, 89-90, 94, 161, 176, 873, 1256. 参见 specific operation
bottleneck(瓶颈), 882-883
application design and(应用程序设计), 402, 1029, 1033-1035, 1039
file structure and(文件结构), 468
system architecture and(系统体系结构), 782-783, 800, 816-819, 829, 839-840
transaction and(事务), 1106-1107, 1116
Boyce-Codd Normal Form(BCNF)(Boyce-Codd 范式)
decomposition algorithm and(分解算法), 349-356
dependency preservation and(保持依赖), 334-336
relational database design and(关系数据库设计), 333-336
broadcast data(广播数据), 1082-1083
B-tree(B 树)
application development and(应用程序开发), 1039
IBM DB2 and, 1205
index and(索引), 504-506, 530, 1039
Oracle and, 1159, 1164-1169, 1173
PostgreSQL and, 1135, 1148-1150
spatial data and(空间数据), 1064, 1071-1072, 1076, 1086
B+-tree(B+树), 1075
balanced tree and(平衡树), 486
bitmap index and(位图索引), 528
bulk loading of index and(索引批量加载), 503-504
deletion time and(删除时间), 491, 495-497, 499-501
extension and(扩展), 500-506
fanout and(扇出), 487
file organization and(文件组织), 500-502
flash memory and(闪存), 506

indexing string and(索引字符串), 502-503
insertion time and(插入时间), 491-495, 499-501
on multiple key(多码), 508
nonleaf node of(非叶结点), 487
nonunique search key and(不唯一的搜索码), 497-499
performance and(性能), 485-486
pointer and(指针), 486
query on(查询), 488-491, 538, 544
record relocation and(记录重定位), 502
secondary index and(辅助索引), 502
structure of(结构), 486-488
update and(更新), 491-500
buffer manager(缓冲区管理器), 21, 464-466
buffer pool(缓冲池), 1201-1202, 1220
buffer(缓冲区)
database buffering and(数据库缓存), 739-741
file structure and(文件结构), 437-438
force/no-force policy and(强制/非强制策略), 739-740
force output and(强制输出), 725-726
IBM DB2 and, 1200-1203
log-record buffering and(日志记录缓冲), 738-739
main-memory database and(主存数据库), 1105-1108
management of(管理), 738-743
multiple pool(多池), 1220
operating system role in(操作系统角色), 741-742
recovery system and(恢复系统), 738-743
replacement policy and(替换策略), 465-468
steal/no-steal policy and(抢占/非抢占策略), 740
transaction server and(事务服务器), 774
Write-Ahead Logging (WAL) rule and(先写日志规则), 739-741
bug(错误), 174n4
application design and(应用程序设计), 404, 1048-1050
recovery system and(恢复系统), 721-722
system architecture and(系统体系结构), 787-788
transaction and(事务), 1093, 1102
workflow and(工作流), 1101
build input(构造用输入), 558
bulk export(批量导出), 1032
bulk insert(批量插入), 1032
bulk load(批量加载), 503-504, 1031-1033
bully algorithm(威逼算法), 851, 852
business logic(业务逻辑), 25, 173
application design and(应用程序设计), 376, 383, 391-393, 396, 410, 418
IBM DB2 and, 1221-1222
Microsoft SQL Server and, 1228, 1252, 1253, 1263-1267
Oracle and, 1158
business-logic layer(业务逻辑层), 39, 391
bus system(总线系统), 780
BYNET, 806
byte-code(字节码), 389

C, 7, 14,
advanced SQL and(高级 SQL), 157, 169, 173, 180
application design and(应用程序设计), 387, 397-398
Microsoft SQL Server and, 1228, 1253
ODBC and(开放数据库互联), 167-168
Unified Modeling Language(UML) and(统一建模语言), 308
C++, 7n1, 14
advanced SQL and(高级 SQL), 169, 173
Microsoft SQL Server and, 1253
object-based database and(基于对象的数据库), 945
persistent system and(持久系统), 968-971
standard for(标准), 1054
Unified Modeling Language(UML) and(统一建模语言), 308
caching(高速缓存), 429
application design and(应用程序设计), 400-401
coherency and(一致性), 776
data server and(数据服务器), 776
lock and(锁), 776
multithreading and(多线程), 817-818
Oracle and, 1184
query plan and(查询计划), 605, 775
shared-memory architecture and(共享内存体系结构), 783
call(调用), 175
callable statement(可调用语句), 164
call back(回调), 776
Call Level Interface (CLI) Standard(调用级接口标准), 1053
candidate key(候选码), 45-46
canonical cover(正则覆盖), 342-345
CAP theorem(CAP 定理), 852-853

Cartesian product(笛卡儿积), 68
  equivalence and(等价), 584
  join expression and(连接表达式), 229-232
  query and(查询), 573, 584, 589, 595-596, 606, 616
  relational algebra and(关系代数), 50-51, 217, 222-232
  SQL and(结构化查询语言), 68-75, 120, 209
cascade(级联的), 133, 150, 186
cascadeless schedule(无级联调度), 647-648
cascading stylesheet (CSS) standard(层叠式样式表标准), 380
case construct(样例构造), 102-103
CASE tool(CASE 工具), 1194-1195
cast(强制类型转换), 136, 139-140
catalog(目录)
  application development and(应用程序开发), 1053
  distributed database and(分布式数据库), 830, 848
  e-catalog and(电子目录), 1103-1104
  IBM DB2 and, 1195, 1220
  index and(索引), 476(参见 index)
  information retrieval and(信息检索), 915, 935
  Microsoft SQL Server and, 1235-1236, 1250, 1256, 1266
  PostgreSQL and, 1151, 1154
  query optimization and(查询优化), 590-596, 1151
  SQL and(结构化查询语言), 142-143, 165, 168-169
  system(系统), 462, 468, 801, 1132
  transaction processing and(事务处理), 1116
  XML and(可扩展标记语言), 1017
category(分类), 935-937
centralized architecture(集中式体系结构), 769-771
centralized deadlock detection(集中式死锁检测), 845-846
challenge-response system(询问-回答系统), 415
change isolation level(改变隔离性级别), 649
check clause(check 子句), 130, 134
  assertion and(断言), 135-136
  authorization and(授权), 148
  data constraint and(数据约束), 310
  dependency preservation and(保持依赖), 334-336
  user-defined type and(用户自定义类型), 140
check constraint(check 约束), 134, 148, 310, 317, 334, 628, 1130
checkpoint log record(检查点日志记录), 752
checkpoint(检查点)
  fuzzy(模糊的), 750-752
  Microsoft SQL Server and, 1246
  Oracle and, 1185
  recovery system and(恢复系统), 734-735, 742-743
  transaction server and(事务服务器), 774
checksum(校验和), 434
classification hierarchy(分类层次), 935-937
classifier(分类器)
  Bayesian(贝叶斯), 900-901, 1191, 1266
  best split and(最优划分), 897-899
  decision-tree(决策树), 895-900
  entropy measure and(熵度量), 897
  Gini measure and(吉尼度量), 897
  information gain and(信息增益), 897-898
  kernel function and(内核函数), 901-902
  neural-net(神经网络), 900
  partition and(划分), 896-897
  prediction and(预测), 894-904
  purity and(纯度), 897
  regression and(回归), 902-903
  Support Vector Machine(支持向量机), 900-902
  training instance and(训练实例), 895
  validating(确认), 903-904
client-server system(客户-服务器系统), 23, 32, 204
  application design and(应用程序设计), 376-377, 1031, 1053, 1056
  Microsoft SQL Server and, 1228
  recovery system and(恢复系统), 756
  system architecture and(系统体系结构), 769-772, 777, 788, 791
  transaction processing and(事务处理), 1092-1096
client-side scripting(客户端脚本), 389-391
clob(字符大对象数据类型), 138, 166, 457, 502, 1010-1013, 1196-1199
cloud computing(云计算), 777
  challenge with(挑战), 868-870
  data representation and(数据表示), 863-865
  partition and(划分), 865-866
  replication and(复制), 866-868
  retrieval and(检索), 865-866
  storage and(存储), 862-863
  traditional database and(传统数据库), 868
  transaction and(事务), 866-868

cluster(聚类), 781
  agglomerative(凝聚), 907-908
  cloud computing and(云计算), 867
  data mining and(数据挖掘), 894, 907-908
  divisive(分裂), 907-908
  hierarchical(层次), 907-908
  IBM DB2 and, 1203-1207
  multidimensional(多维), 1203-1207
  Oracle and, 1173, 1186
  Real Application Cluster(RAC) and, 1186
coalescence(凝聚), 491, 706
code breaking(密码破解). 参见 encryption(加密)
ColdFusion Markup Language(CFML)(ColdFusion 标记语言), 387
collect function(集合函数), 959
collection volume(集合体), 957, 957-958
column-oriented storage(面向列的存储), 892-893
combinatorics(组合数学), 639
commit protocol(提交协议), 832-838
commit time(提交时间), 758
commit work(commit work 语句), 127
Common Gateway Interface(CGI) standard(公用网关接口标准), 380-381
Common Language Runtime(CLR)(公共语言运行库), 180
Common Object Request Broker Architecture (CORBA)(通用对象请求代理体系结构), 1054-1055
common subexpression elimination(通用子表达式删除), 614
commutative property(交换律), 584-585
compatibility function(相容函数), 662
Compensation Log Record(CLR)(补偿日志记录), 751-752, 754
complex data type(复杂数据类型), 28, 31, 138, 1061
  entity-relationship (E-R) model and(实体-联系模型), 946-947
  keyword and(关键字), 947-949
  normal form and(范式), 947
  object-based database and(基于对象的数据库), 945-949, 963, 970-974
  system architecture and(系统体系结构), 864
Component Object Model (COM)(组件对象模型), 1253
compression(压缩), 1077-1078
  application development and(应用程序开发), 1041
  data warehouses and(数据仓库), 893
  IBM DB2 and, 1194
  Microsoft SQL Server and, 1236
  Oracle and, 1165-1167
  prefix(前缀), 503, 1166
computer-aided design (CAD)(计算机辅助设计), 312, 1061, 1064-1068
conceptual design(概念设计), 15-16, 260
concurrency control(并发控制), 631, 636, 639, 709
  access anomaly and(访问异常), 5
  blind write and(盲写), 687
  consistency and(一致性), 695, 701-704, 710
  deadlock handling and(死锁处理), 674-679
  delete operation and(删除操作), 697-701
  distributed database and(分布式数据库), 835-836, 839-847
  DML command and(数据操纵语言命令), 1138-1139
  false cycle and(假环), 846-847
  IBM DB2 and, 1200-1203, 1217-1218
  index structure and(索引结构), 704-708
  insert operation and(插入操作), 697-701
  lock and(锁), 661-686, 839-842
  logical undo and(逻辑 undo), 749-750
  long-duration transaction and(长事务), 1111-1112
  Microsoft SQL Server and, 1241-1246
  multiple granularity and(多粒度), 679-682
  multiversion scheme and(多版本机制), 689-692, 1137-1146
  Oracle and, 1180-1183
  PostgreSQL and, 1137-1145
  predicate read and(谓词读), 697-701
  recovery system and(恢复系统), 729-730
  rollback and(回滚), 667, 670, 674-679, 685, 689, 691, 709
  serializability and(可串行化), 650, 662, 666-667, 671, 673, 681-690, 693-697, 701-704, 708
  snapshot isolation and(快照隔离), 692-697, 729-730
  timestamp-based protocol and(基于时间戳的协议), 682-686
  update and(更新), 867-868
  user interaction and(用户交互), 702-704
  validation and(有效性检查), 686-689, 729-730
  Web crawler and(网络爬虫), 930-931
condition-defined entity set(条件定义的实体

集)，299
confidence(置信度)，893，905
conformance level(一致性级别)，168-169
conjunction(连接)，545-546，594
connect to(连接到)，170
consistency(一致性)，11，22，104
  availability and(可用性)，852-853
  CAP theorem and(CAP 定理)，852-853
  concurrency control and(并发控制)，695，701-704，710
  cursor stability and(游标稳定性)，702
  deadlock and(死锁)，665-666
  degree-two(二级)，701-702
  distributed transaction and(分布式事务)，830-832
  logical operation and(逻辑运算)，746
  mobile network and(移动网络)，1083-1085
  replication with weak(弱一致性副本)，843-844
  requirement of(要求)，630
  transaction and(事务)，627-631，635-636，640，648-650，655
  user interaction and(用户交互)，702-704
  weak level of(弱一致性级别)，701-704
constraint(约束)
  condition-defined(条件定义的)，299
  decomposition and(分解)，354-355
  dependency preservation and(保持依赖)，334-336
  disjoint(不相交)，300
  entity-relationship (E-R) model and(实体－联系模型)，269-272，299-301
  IBM DB2 and，1199
  integrity(完整性)，4 (参见 integrity constraint)
  key(码)，271-272
  mapping cardinality and(映射基数)，269-270，276-277
  overlapping(重叠)，300
  partial(部分)，300
  participation(参与)，270
  PostgreSQL and，1130-1131，1153-1154
  total(全部)，300
  transaction and(事务)，628，1108-1109
  UML and(统一建模语言)，309-310
  user-defined(用户自定义)，299
contain(包含)，93
context switch(上下文切换)，1092
conversation group(会话组)，1262
cookie(一小段包含标识信息的文本)，382-385，403-405
core(核)，770
correlated subquery(相关子查询)，93
correlation(相关性)，906
correlation variable(相关变量)，605-607
count(计数)，84-86，89，566-567
count-distinct(不同计数)，236
count value(计数值)，204
crash(崩溃)，434，467-468
  action after(动作)，736-738
  algorithm for(算法)，735-738
  ARIES and，750-756
  checkpoint and(检查点)，734-735
  failure classification and(故障分类)，721-722
  recovery system and(恢复系统)，736-738(参见 recovery system)
  transaction and(事务)，628
create assertion(创建断言)，135
create distinct type(创建独特类型)，141，1194-1196，1197
create domain(创建域)，140-141
create function(创建函数)，175，177，189
create index(创建索引)，528-529，1150-1151
create or replace(创建或替换)，174n4
create procedure(创建过程)，175，179
create recursive view(创建递归视图)，192
create role(创建角色)，146
create schema(创建模式)，143
create snapshot(创建快照)，843-844
create table(创建表)，60-63，139，161
  with data(带数据)，141-142
  extension for(扩展)，141-142
  integrity constraint and(完整性约束)，129
  object-based database and(基于对象的数据库)，950，961-962
create table. . . as(创建……表)，142
create type(创建类型)，139-141
create unique index(创建唯一性索引)，529
create view(创建视图)，121-123，125，142，147，1130-1131
Cross-Site Request Forgery(XSRF)(跨站请求伪造)，403-404
cross-site scripting (XSS)(跨站脚本)，403-405
cross-tabulation(交叉表)，199-203，205，210
CRUD Web interface(CRUD Web 接口)，399
Crystal Report(水晶报表)，399-400
cube by(立方体)，1221-1222
cube construct(立方体构造)，206-210

current-date(当前日期), 137
cursor stability(游标稳定性), 702
curve fitting(曲线拟合), 902-903
Cyc project(Cyc 工程), 925, 927

data abstraction(数据抽象), 6-8
data access layer(数据访问层), 391, 393, 395
data analysis(数据分析)
  data mining and(数据挖掘), 893-910
  decision-support system and(决策支持系统), 887-891
  information retrieval and(信息检索), 915-938
  warehousing and(仓库), 887-891
database design(数据库设计), 257, 313-314
  adapting for modern architecture and(适应现代体系结构), 818-819
  alternative and(选择), 261-262, 304-310
  applications(应用), 375-418
  authorization requirement and(授权需求), 312
  automated tuning and(自动调整), 1040-1041
  bottom-up(自底向上), 297
  buffer and(缓冲区), 464-468
  client-server architecture and(客户-服务器体系结构), 32, 204, 376-377, 756, 766-767, 769-772, 777, 788, 791
  complexity of(复杂性), 260
  conceptual-design phase and(概念设计阶段), 15-16, 260
  data constraint and(数据约束), 310-311
  direct design process and(直接设计过程), 259-260
  encryption and(加密), 411-417
  entity-relationship (E-R) model and(实体-联系模型), 17-18, 249-313
  IBM DB2 and, 1194-1195
  incompleteness and(不完整), 262
  logical-design phase and(逻辑设计阶段), 16, 260-261
  Microsoft SQL Server and, 1223-1228
  normalization and(标准化), 18-20
  Oracle and, 1157-1158
  overview of process(过程概述), 259-262
  parallel system and(并行系统), 815-817
  phase of(阶段), 259-261
  physical-design phase and(物理设计阶段), 16, 261
  redundancy and(冗余), 261-262
  relational(关系的), 323-368
  specification of functional requirement and(功能需求说明), 16
  top-down(自顶向下), 297
  university and(大学), 16-17
  user need and(用户需求), 260
  user requirement and(用户需求), 311-312
  workflow and(工作流), 312-313
database graph(数据库图), 671-674
database instance(数据库实例), 42
database(数据库)
  abstraction and(抽象), 6-8, 10
  architecture and(体系结构), 28-29, 767(参见 architecture)
  buffering and(缓冲), 739-741(参见 buffers)
  concurrency control and(并发控制), 661-710(参见 concurrency control)
  distributed(分布式), 825-878, 1188
  dumping and(转储), 743-744
  file-processing system and(文件处理系统), 3-6
  force output and(强制输出), 725-726
  history of(历史), 29-31
  indexing and(索引), 475-531(参见 index)
  information retrieval and(信息检索), 915-937
  lock and(锁), 661-679(参见 lock)
  main-memory(主存), 1105-1108
  mobile(移动), 1079-1085
  modification and(修改), 98-103, 728-729
  multimedia(多媒体), 1076-1079
  normalization and(标准化), 18-20
  parallel(并行), 797-820
  personal(个人), 1079-1085
  recovery system and(恢复系统), 631, 633, 721-761(参见 recovery system)
  storage and(存储), 20-22, 427(参见 storage)
  time in(时间), 1062-1064
DataBase Administrator (DBA)(数据库管理员), 28-29, 149, 1152, 1214-1215, 1243
database schema(数据库模式). 参见 schema
Database Task Group(数据库任务组), 1052
database writer process(数据库写进程), 773
data cleansing(数据清洗), 890
data cube(数据立方体), 200, 206-210
data-definition language(DDL)(数据定义语言), 9-12, 14, 32
  authorization and(授权), 58
  basic type and(基本类型), 59-60
  concurrency control and(并发控制), 1144-1145

dumping and(转储), 743-744
IBM DB2 and, 1194-1197, 1204
index and(索引), 58
integrity and(完整性), 58
Microsoft SQL Server and, 1225, 1228-1233, 1245, 1253, 1256, 1261
Oracle and, 1162, 1181
PostgreSQL and, 1144-1145, 1150
querying and(查询), 21-22
schema definition and(模式定义), 28, 58, 60-63
security and(安全), 58
set of relation in(关系集合), 58-61
SQL basic and(SQL 基础), 57-63, 104
storage and(存储), 58
data dictionary(数据字典), 12, 21, 462-464
Data Encryption Standard (DES)(数据加密标准), 413
DataGrid control(DataGrid 控件), 398
data guard(数据卫士), 1183
data inconsistency(数据不一致性). 参见 consistency
data isolation(数据隔离性), 参见 isolation
DATAllegro, 816
Datalog(Datalog 语言), 37
data-manipulation language (DML)(数据操纵语言), 12-14
authorization and(授权), 143
compiler and(编译器), 21-22
concurrency control and(并发控制), 1138-1139, 1145
declarative(描述性的), 10
defined(定义的), 10, 32
host language and(宿主语言), 15
Microsoft SQL Server and, 1231-1233, 1245, 1261
Oracle and, 1161-1162, 1165, 1181
PostgreSQL and, 1137-1138
precompiler and(预编译器), 15
procedural/nonprocedural(过程化/非过程化), 10
querying and(查询), 21-22
snapshot isolation and(快照隔离), 1137(参见 snapshot isolation)
storage manager and(存储管理器), 20-21
triggers and(触发器), 1161-1162
data mediation(数据中介), 1018-1019
data mining(数据挖掘), 25-26, 33, 771-772, 887-889
association rule and(关联规则), 904-907
best split and(最优划分), 897-899
classification and(分类), 894-904
cluster and(聚类), 894, 907-908
data-visualization(数据可视化), 909
defined(定义的), 893
descriptive pattern and(描述性模式), 894
entropy measure and(熵度量), 897
Gini measure and(吉尼度量), 897
information gain and(信息增益), 897-898
Microsoft SQL Server and, 1266-1267
Oracle and, 1191
prediction and(预测), 894-904
purity and(纯度), 897
rule for(规则), 893-894
text(文本), 908
data model(数据模型). 参见 specific model
data parallelism(数据并行性), 805
data server system(数据服务器系统), 773, 775-777, 782
data storage and definition language(数据存储和定义语言), 11
data striping(数据拆分), 444
data-transfer rate(数据传输率), 435-436
data type(数据类型). 参见 type
data warehouses(数据仓库), 888
column-oriented storage and(面向列的存储), 892-893
component of(构件), 889-891
deduplication and(去重), 890-891
defined(定义的), 889
fact table and(事实表), 891-892
household and(住宅类作), 891
IBM DB2 and, 1194, 1221-1222
materialized view and(物化视图), 1171-1172
merger-purge operation and(合并-清除操作), 890-891
Microsoft SQL Server and, 1264
Oracle and, 1158, 1162, 1169-1172, 1178, 1189
transformation and(转换), 891
update and(更新), 891
Data Encryption Standard (DES)(数据加密标准), 413
datetime(时间), 201
DB-Lib, 1249
deadline(期限), 1108-1109
deadlock(死锁)
consistency and(一致性), 665-666

distributed database and(分布式数据库), 839, 841, 844-847
handling of(处理), 674-679
IBM DB2 and, 1217-1220
long-duration transaction and(长事务), 1110-1111
Microsoft SQL Server and, 1243-1244, 1246
PostgreSQL and, 1143-1145
prevention of(预防), 675-676
rollback and(回滚), 678-679
starvation and(饿死), 679
victim selection and(牺牲者选择), 678
wait-for graph and(等待图), 676-677, 676-678
decision support(决策支持), 1047
decision-support query(决策支持查询), 797
decision-support system(决策支持系统), 887-891
decision-tree classifier(决策树分类器), 895-900
declare(声明), 175-178
decode(解码), 208
decomposition(分解)
algorithm for(算法), 348-355
Boyce-Codd Normal Form and(Boyce-Codd 范式), 333-336, 349-355
dependency preservation and(保持依赖), 334-336
fourth normal form and(第四范式), 358-360
functional dependency and(函数依赖), 329-338, 355-360
higher normal form and(更高级范式), 337-338
key and(键), 330-333
lossless(无损), 345-346
lossless-join(无损连接), 345-346
lossy(有损), 345-346
multivalued dependency and(多值依赖), 355-360
relational database design and(关系数据库设计), 329-338, 348-360
third normal form and(第三范式), 336-337, 352-355
decomposition rule(分解规则), 339
DEC Rdb, 30
deduplication(去重), 890-891
Deep Web crawler(深度 Web 爬虫), 931
default value(默认值), 133, 137, 140, 144, 425, 899, 952, 991-992, 996, 1128
deferred integrity constraint(延迟的完整性约束), 134
degree-two consistency(二级一致性), 701-702
deletion(删除), 61, 63, 98-100, 102, 161
concurrency control and(并发控制), 697-701
EXEC SQL and, 171
hashing and(散列), 510, 513, 516, 523
integrity constraint and(完整性约束), 133
PostgreSQL and, 1130-1131
privilege and(权限), 143-145
transaction and(事务), 629, 653
trigger and(触发器), 183
view and(视图), 125
delta relation(delta 关系), 186
demand-driven pipelining(需求驱动流水线), 569-570
denormalization(解除规范化), 363-364
dependency preservation(保持依赖), 334-336, 346-348
desc(降序), 77-78
descriptive attribute(描述性属性), 参见 attribute
descriptive pattern(描述性模式), 894
deviation(偏离), 215, 906-907
dicing(切块), 201
dictionary attack(字典攻击), 414
digital certificate(数字证书), 416-417
digital signature(数字签名), 416
direct-access storage(随机访问存储), 431
directory(目录), 935-937
directory information tree (DIT)(目录信息树), 872-875
directory system(目录系统), 870-875
dirty block(脏块), 741
dirty page table(脏页表), 750-756, 1244-1245
dirty read(读脏数据), 1137, 1181
dirty write(写脏数据), 649
disable trigger(使触发器无效), 185
disconnected operation(断连操作), 395-396
disjoint entity set(不相交实体集), 300
disjoint specialization(不相交特殊化), 296-297
disjunction(析取), 545-546, 594
disk-arm-scheduling(磁盘臂调度), 437
disk controller(磁盘控制器), 434
distinct type(独特类型), 84-86, 138-141
distinguished name(DN)(可区分的名称), 872
distributed database(分布式数据库), 876-878
availability and(可用性), 847-853
cloud-based(基于云的), 861-870
commit protocol and(提交协议), 832-838
concurrency control and(并发控制), 835-836, 839-847
deadlock handling and(死锁处理), 844-847

directory system and(目录系统), 870-875
failure and(故障), 831-835
fragmentation and(分片), 826-829
heterogeneous(异构), 825-826, 857-861
homogeneous(同构), 825-826
join and(连接), 855-857
lock and(锁), 839-847
partition and(划分), 835
persistent messaging and(持久消息), 836-838
query processing and(查询处理), 854-860
recovery and(恢复), 835-836
replication and(复制), 826, 829, 843-844
storage and(存储), 826-830
timestamp and(时间戳), 842-843
transparency and(透明性), 829-830
unified view of data and(数据统一视图), 858-859
distributed-lock manager(分布式锁管理器), 840
distributed system(分布式系统)
autonomy and(自治), 785
availability and(可用性), 785
example of(例子), 786
global transaction and(全局事务), 784
greater potential for bug in(更大的出错可能性), 787-788
implementation and(实现), 786-788
increased processing overhead of(增加的处理开销), 788
local transaction and(局部事务), 784
node and(结点), 784
ready state and(就绪状态), 787
replication and(复制), 785
sharing data and(共享数据), 785
site and(站点), 784
software-development cost and(软件开发代价), 787
two-phase commit protocol(2PC) and(两阶段提交协议), 786-788
distributor(分发者), 1252
divisive clustering(分裂聚类), 907-908
Document Object Model(DOM)(文档对象模型), 390
Document Type Definition(DTD)(文档类型定义), 990-994
domain(域), 42
domain constraint(域约束), 11
Domain-Key Normal Form(DKNF)(域码范式), 360
domain relational calculus(域关系演算), 249
example query(举例查询), 246-247
expressive power of language(语言表达能力), 248
formal definition(形式化定义), 245
safety of expression(表达式安全性), 247-248
double-pipelined hash-join(双流水线散列连接), 571-572
drill down(下钻), 201
Driver-Manager class(驱动程序管理器类), 160
drop index(删除索引), 529
drop table(删除表), 63, 164
drop trigger(删除触发器), 185
drop type(删除类型), 140
dumping(转储), 743-744
duplicate elimination(消除重复), 563-564
durability(持久性), 22-23, 104, 625, 630-631
defined(定义的), 628
distributed transaction and(分布式事务), 830-832
one-safe(一方保险), 758
remote backup system and(远程备份系统), 758
storage structure and(存储结构), 632-633
two-safe(两方保险), 758
two-very-safe(两方强保险), 758
dynamic SQL(动态 SQL), 58, 158, 175

e-catalog(电子目录), 1103
Eclipse, 386
E-commerce(电子商务), 1102-1105
efficiency(效率), 6-8
election algorithm(选举算法), 851-852
embedded SQL(嵌入式 SQL), 58, 158, 169-173
empty relation(空关系), 93-94
encryption(加密)
Advanced Encryption Standard(AES)(扩展加密标准), 412-413
application of(应用), 411-417
asymmetric-key(非对称密钥), 412
authentication and(鉴定), 415-417
challenge-response system and(询问-回答系统), 415
database support and(数据库支持), 414-415
dictionary attack and(字典攻击), 414
digital certificate and(数字证书), 416-417
digital signature and(数字签名), 416
nonrepudiation and(不可否认性), 416
Oracle and, 1165-1166
prime number and(素数), 413

public-key(公钥), 412-414
Rijndael algorithm and(Rijndael 算法), 412-413
technique of(技术), 412-414
end-user information(最终用户信息), 407-408
enterprise information(企业信息), 1-2
Entity Data Model(实体数据模型), 395
entity-relationship (E-R) diagram(实体联系图), 17-18
alternative notation for(替代记号), 304-310
basic structure of(基本结构), 274-275
complex attribute(复杂属性), 277-278
entity set and(实体集), 279-281
generalization and(泛化), 298
identifying relationship and(标识性联系), 280
mapping cardinality(映射基数), 276
nonbinary relationship set(非二元联系集), 278-279
relationship set(联系集), 278-279
roles(角色), 278
university enterprise example(大学企业例子), 282-283
weak entity set(弱实体集), 279-281
entity-relationship (E-R) model(实体-联系模型), 9, 17-18, 259, 313-314, 963
aggregation and(聚集), 301-302, 304
alternative modeling data notation and(替代数据建模记号), 304-310
atomic domain and(原子域), 327-329
attribute and(属性), 263, 267-269, 290-291, 294-295, 298-299, 327-329
complex data type and(复杂数据类型), 946-947
constraint and(约束), 269-272
design issue and(设计问题), 290-295
enterprise schema and(企业模式), 262
entity set and(实体集), 262-267, 272-274, 279-286, 290-291, 296-298
extended feature(扩展特性), 295-304
generalization and(概化), 297-304
normalization and(标准化), 361-362
object-oriented data model and(面向对象数据模型), 27
reduction to relational schema and(转换为关系模式), 283-290
redundancy and(冗余), 272-274
relationship set and(联系集), 264-267, 286-290, 291-295, 296-297
specialization and(特化), 295-297
Unified Modeling Language (UML) and(统一建模语言), 308-310
entity set(实体集)
alternative notation for(替代记号), 304-310
attribute and(属性), 263, 284-285, 290-291
condition-defined(条件定义的), 299
defined(定义的), 262-263
design issue and(设计问题), 290-292
disjoint(不相交), 299
extension of(扩展), 263
generalization and(概化), 297-304
identifying relationship and(标识性联系), 280
overlapping(重叠), 299
property of(性质), 262-264
relationship set and(联系集), 264-267, 291-292
removing redundancy in(消除冗余), 272-274
role of(角色), 264-265
simple attribute and(简单属性), 283-284
strong(强), 283-285
subclass(子类), 298
superclass(超类), 298
superclass-subclass relationship and(超类-子类联系), 296-297
Unified Modeling Language (UML) and(统一建模语言), 308-310
user-defined(用户自定义), 299
weak(弱), 279-281, 285-286
Entity SQL(实体 SQL), 395
Enterprise Resource Planning (ERP) system(企业资源规划系统), 1101
entropy measure(熵度量), 897-898
equi-join(等值连接), 549-559, 563, 566, 571, 807, 819
equivalence(等价)
cost analysis and(代价分析), 601-602
join ordering and(连接次序), 588-589
relational algebra and(关系代数), 582-590
transformation example for(转换例子), 586-588
Error-Correcting Code (ECC) organization(纠错码组织), 444-445
ERWin, 1194
escape(转义), 77
evaluation primitive(计算原语), 539
every clause(every 子句), 90
except clause(except 子句), 82-83, 93, 188
exchange system(交换系统), 1104
exclusive-mode lock(排他型锁), 861

EXEC SQL, 169-173
execute(执行), 147
existence bitmap(存在位图), 526-527
exists clause(exists 子句), 93
extensibility contract(可扩展性约定), 1256-1258
external language routine (外部语言例程), 179-180
external sort-merge algorithm(外排序归并算法), 548-549

Facebook, 31, 862
factorial(阶乘), 639
fact table(事实表), 891-892
fail-stop assumption(故障-停止假设), 722
false cycle(假环), 846-847
false drop(误丢弃), 929
false negative(误舍弃), 903, 929-930
false positive(误选中), 903, 929-930
false value(假值), 90, 208
fanout(扇出), 487
fetching, 21, 138, 906, 1078, 1097
  advanced SQL and(高级 SQL), 161, 166-173, 176, 180, 194
  application design and(应用程序设计), 389-397, 1030, 1038
  IBM DB2 and, 1189, 1202, 1209, 1211, 1219
  information retrieval and(信息检索), 921, 929, 936(参见 information retrieval)
  Microsoft SQL Server and, 1241, 1251
  object-based database and(基于对象的数据库), 965, 969-972
  PostgreSQL and, 1137, 1146, 1151, 1153
  storage and(存储), 437, 439, 444(参见 storage)
  Web crawler and(网络爬虫), 930-931
Fibre Channel interface(光纤通道接口), 434, 436
fifth normal form(第五范式), 360
file header(文件头), 454
file manager(文件管理器), 21
file organization (文件组织), 3-4. 参见 also storage
  B+-tree and(B+树), 500-502
  blob(二进制大对象), 138, 166, 457, 502, 1013, 1198-1199, 1259
  block-access time and(块访问时间), 438
  clob(字符大对象数据类型), 138, 166, 457, 502, 1010-1013, 1196-1199
  fixed-length record and(定长记录), 452-454
  hashing(散列), 457
  heap file(堆文件), 457
  indexing and(索引), 475(参见 index)
  journaling system and(日志系统), 439
  multitable clustering and(多表聚簇), 458, 460-462
  pointer and(指针), 454
  security and(安全性), 5-6(参见 security)
  sequential(顺序), 457-459
  structured(结构化的), 451-468
  system structure and(系统结构), 451-452
  variable-length record and(变长记录), 454-457
file scan(文件扫描), 541-544, 550, 552, 570
final/not final expression(final/not final 表达式), 949, 953
fine-granularity parallelism(细粒度并行), 771
FireWire interface(火线接口), 434
first committer wins(先提交者胜), 692-693
first updater wins(先更新者胜), 693
flash storage(闪存), 403
  B+-tree and(B+树), 506
  cost of(代价), 439
  erase speed and(擦除速度), 440
  hybrid disk drive and(混合磁盘驱动器), 440-441
  NAND, 430, 439-440
  NOR, 430, 439
flash translation layer(闪存转换层), 440
floppy disk(软盘), 430
flow-distinct(flow-distinct 操作), 1240-1241
FLWOR (for, let, where, order by, return) expression(FLWOR 表达式), 1002-1003
forced output(强制写出), 465, 725-726
force policy(强制策略), 739-740
for each row clause(for each row 子句), 181-184
for each statement clause(for each statement 子句), 68, 183
foreign key(外码), 46, 61-62, 131-133
fourth normal form(第四范式), 356, 358-360
fragmentation(分片), 827-829
Free Space Control Record(FSCR)(自由空间控制记录), 1202
from statement(from 语句)
  aggregate function and(聚集函数), 84-90
  basic SQL query and(基本 SQL 查询), 63-74
  on multiple relation(多关系), 66-71
  natural join and(自然连接), 71-74

null value and(空值), 83-84
rename operation and(更名运算), 74-75
set operation and(集合运算), 79-83
on single relation(单关系), 63-66
string operation and(字符串运算), 76-79
subquery and(子查询), 95-96
full outer join(全外连接), 117-120, 233-234, 565-566
functional dependency(函数依赖), 18, 129
attribute set closure and(属性集闭包), 340-342
augmentation rule and(增补律), 339
BCNF algorithm and(BCNF 算法), 349-352
Boyce-Codd normal form and(Boyce-Codd 范式), 333-336
canonical cover and(正则覆盖), 342-345
closure of a set(集合闭包), 338-340
decomposition rule(分解规则), 339
dependency preservation and(保持依赖), 334-336, 346-348
extraneous(无关的), 342
higher normal form and(更高级范式), 337-338
key and(码), 330-333
lossless decomposition and(无损分解), 345-346
multivalued(多值), 355-360
pseudotransitivity rule(伪传递律), 339
reflexivity rule and(自反律), 339
theory of(理论), 338-348
third normal form and(第三范式), 336-337, 352-355
transitivity rule and(传递律), 339
union rule(合并律), 339
functionally determined attribute(函数决定的属性), 340-342
function-based index(基于函数的索引), 1167-1168
function(函数). 参见 specific function
declaring(声明), 174-175
external language routine and(外部语言例程), 179-180
IBM DB2 and, 1197-1198
language construct for(语言结构), 176-179
polymorphic(多态), 1128-1129
PostgreSQL, 1133-1135
state transition(状态转换), 1134
syntax and(语法), 173-174, 178
writing in SQL(用 SQL 书写), 173-180
XML and(可扩展标记语言), 1006-1007
fuzzy checkpoint(模糊检查点), 742-743, 750-752
generalization(概化)
aggregation and(聚集), 301-302
attribute inheritance and(属性继承), 298-299
bottom-up design and(自底向上设计), 297
condition-defined(条件定义的), 299
constraint on(约束), 299-301
disjoint(不相交), 300
entity-relationship(E-R) model and(实体-联系模型), 297-304
overlapping(重叠), 300
partial(部分), 300
representation of(表示), 302-304
subclass set and(子类集), 298
superclass set and(超类集), 298
top-down design and(自顶向下设计), 297
total(全部), 300
user-defined(用户自定义), 299
Generalized Inverted Index (GIN)(通用倒排索引), 1149
generalized-projection(广义投影), 235
Generalized Search Tree (GiST)(通用搜索树), 1148-1149
geographic data(地理数据), 1061
application of(应用), 1068
information system and(信息系统), 1065
raster data and(光栅数据), 1069
representation of(表示), 1065-1066, 1069-1070
spatial query and(空间查询), 1070-1071
vector data and(矢量数据), 1069
getColumnCount method(getColumnCount 方法), 164-165
getConnection method(getConnection 方法), 160
GET method(GET 方法), 405
getString method(getString 方法), 161
Gini measure(吉尼度量), 897
Glassfish, 386
global class identifier(全局类别标识), 1055
global company identifier(全局公司标识), 1055
Global Positioning System (GPS)(全球定位系统), 1068
global product identifier(全局产品标识), 1055
global wait-for graph(全局等待图), 845-847
Google, 31
application design and(应用程序设计), 378-382, 396, 407
distributed database and(分布式数据库), 862, 866-867

information retrieval and (信息检索), 933 (参见 information retrieval)
PageRank and, 922-925
grant (grant 语句), 144-150
granted by current role, 150
graph-based protocol (基于图的协议), 671-674
Greenplum, 816
group by (按……分组), 86-89, 96, 194, 203, 206-209
growing phase (增长阶段), 667-669

hacker (黑客). 参见 security
handoff (交接), 1081
hard disk (硬盘), 29-30
hardware RAID (硬件 RAID), 448
hardware tuning (硬件调整), 1035-1038
harmonic mean (调和平均数), 1046
hash cluster access (散列聚类访问), 1173
hash function (散列函数), 457-458, 476, 530-531
closed (闭), 513
data structure and (数据结构), 515-516
deletion and (删除), 510, 513, 516, 523-524
dynamic (动态), 515-523
extendable (可扩展的), 515
index and (索引), 514-515, 523-524
insertion and (插入), 513, 516-524
insufficient bucket and (桶不足), 512
join and (连接), 809-810
lookup and (查找), 516-518, 522, 524
open (开), 513
Oracle and, 1170
overflow and (溢出), 512-513
partitioning and (划分), 807
PostgreSQL and, 1148
query and (查询), 516-522
skew and (偏斜), 512
static (静态), 509-515, 522-523
update and (更新), 516-522
hash join (散列连接), 602
basics of (基本), 558-559
build input and (构造用输入), 558
cost of (代价), 561-562
double-pipelined (双流水线), 571-572
hybrid (混合), 562
overflow and (溢出), 560
query processing and (查询处理), 557-562
recursive partitioning and (递归划分), 539-540
skewed partitioning and (偏斜划分), 560
hash-table overflow (散列表溢出), 560
having (having 语句), 88-89, 96
heap file (堆文件), 457, 523, 1147-1149, 1153
heuristics (启发式), 1075-1076
data analysis and (数据分析), 899, 910
distributed database and (分布式数据库), 859
greedy (贪心), 910
IBM DB2 and, 1211
information retrieval and (信息检索), 934
Microsoft SQL Server and, 1240
Oracle and, 1176
parallel database and (并行数据库), 815
query optimization and (查询优化), 598-605, 615-616
Hibernate system (Hibernate 系统), 393-395
hierarchical architecture (层次结构), 781, 784
hierarchical classification (层次分类), 935-937
hierarchical clustering (层次聚类), 907-908
hierarchical data model (层次数据模型), 9
high availability (高可用性), 756
HIPAA, 1248
histogram (直方图), 195, 591-596, 616, 801, 901, 1152, 1175, 1211, 1239
HITS algorithm (HITS 算法), 925
home processor (本地处理器), 803
homonym (同义词), 925-927
horizontal fragmentation (水平分片), 827-828
horizontal partitioning (水平分区), 798
hot-spare configuration (热备份配置), 758
hot swapping (热交换), 449
householding (住宅操作), 891
HP-UX, 1193
hub (链接中心), 924
hybrid disk drive (混合磁盘驱动), 440-441
hybrid hash join (混合散列连接), 562
hybrid merge join (混合归并连接), 557
hybrid OLAP (HOLAP) (混合联机分析处理), 204
hypercube architecture (超立方体体系结构), 781
hyperlink (超链接)
PageRank and, 922-923, 925
popularity ranking and (流行度排名), 920-922
search engine spamming and (搜索引擎作弊), 924-925
HyperText Markup Language (HTML) (超文本标记语言), 378-380

client-side scripting and(客户端脚本), 388-391
DataGrid and, 398
embedded(嵌入的), 397
information retrieval and(信息检索), 915 (参见 Information retrieval)
Java Server Pages (JSP) and, 387-391
Rapid Application Development (RAD) and(快速应用开发), 397
security and(安全性), 402-417
server-side scripting and(服务器端脚本), 386-388
stylesheet and(样式表), 380
Web application framework and(Web 应用构架), 398-400
web session and(web 会话), 380-382
XML and(可扩展标记语言), 981
HyperText Transfer Protocol(HTTP)(超文本传输协议)
application design and(应用程序设计), 377-381, 383, 395, 404-406, 417
as connectionless(无连接的), 381
digital certificate and(数字证书), 417
man-in-the-middle attack and(中间人攻击), 406
REpresentation State Tran-fer (REST) and(表示状态转移), 395
Simple Object Access Protocol (SOAP) and(简单对象访问协议), 1017-1018, 1249-1250

IBM AIX, 1193
IBM DB2, 30, 96, 141, 160n3, 172, 180, 184-185, 216, 1121
administrative tool(管理工具), 1215-1216
autonomic feature(自动特性), 1214-1215
buffer pool and(缓冲池), 1201-1202
business intelligence feature(商务智能特性), 1221-1222
concurrency control and(并发控制), 1200-1203, 1217-1218
constraint and(约束), 1199
Control Center(控制中心), 1195, 1215
database-design tool and(数据库设计工具), 1194-1195
data type support and(数据类型支持), 1196-1197
Data Warehouse Edition(数据仓库版本), 1221
development of(开发), 1193-1193
distribution(分布), 1220-1221
external data and(外部数据), 1220-1221
indexing and(索引), 1199-1205
isolation and(隔离), 1217-1218
join and(连接), 1209-1210
large object and(大对象), 1198-1199
lock and(锁), 1217-1220
Massively Parallel Processor (MPP) and(大规模并行处理器), 1193
materialized view and(物化视图), 1212-1214
multidimensional clustering and(多维聚簇), 1203-1207
query processing and(查询处理), 593, 604, 612, 1207-1216
recovery and(恢复), 1200-1203
replication(复制), 1220-1221
rollback and(回滚), 1218
set operation and(集合运算), 1209-1210
SQL variation and(SQL 变种), 1195-1200
storage and(存储), 1200-1203
system architecture(系统体系结构), 1219-1220
System R and, 1193
Universal Database Server, 1193-1194
user-defined function and(用户自定义函数), 1197-1198
utility(实用程序), 1215-1216
Web service(Web 服务), 1199-1200
XML and(可扩展标记语言), 1195-1196
IBM MVS, 1193
IBM OS/400, 1193-1194
IBM VM, 1193-1194
IBM z/OS, 1194
identifier(标识), 546
global(全局), 1055
OrdPath, 1260-1261
standard and(标准), 1055-1056
tag and(标签), 982-985
identifying relationship(标识性联系), 280
identity declaration(identity 声明), 1043
if clause(if 子句), 184
Illustra Information Technology(Illustra 信息技术), 1123-1124
immediate-modification technique(立即修改技术), 729
incompleteness(不完整), 262
inconsistent state (不一致状态), 630. 参见 consistency
in construct(in 结构), 91, 92n8
incremental view maintenance(增量视图维护), 608-611

independent parallelism(独立并行), 814
indexed nested-loop join (索引嵌套循环连接), 552-553
index entry(索引项), 477
indexing string(索引字符串), 502-503
Index-Organized Table (IOT)(按索引组织表), 1164-1165
index record(索引记录), 477
index(索引), 21, 137-138, 530-531
  access time and(访问时间), 476, 479, 523
  access type and(访问类型), 476
  bitmap(位图), 507, 509, 524-528, 531, 536, 1166-1167
  block(块), 1205
  bulk loading of(批量加载), 503-504
  clustering(聚集), 476-477, 483-485, 542
  comparison and(比较), 544-545
  composite(复合), 545-546
  concurrency control and(并发控制), 704-708
  construction of(构建), 1150-1151
  covering(覆盖), 509
  definition in SQL and(用 SQL 定义), 528-529
  deletion time and(删除时间), 476, 483, 491, 495-501, 523-524
  dense(稠密), 477-483
  domain(域), 1168-1169
  on expression(对表达式的索引), 1149
  function-based(基于函数的), 1167-1168
  Generalized Inverted Index (GIN) and(通用倒排索引), 1149
  Hashed(散列), 476 (参见 hash function)
  IBM DB2 and, 1199-1205
  identifier and(标识), 546
  information retrieval and(信息检索), 927-929
  insertion time and(插入时间), 476, 482-483, 491-495, 499-501, 523-524
  inverted(倒排), 927-929
  join(连接), 1168 (参见 joins)
  linear search and(线性搜索), 541-542
  logical row-id and(逻辑行标识), 1164-1165
  materialized view and(物化视图), 612
  Microsoft SQL Server and, 1231-1236
  multicolumn(多列), 1149
  multilevel(多级别), 480-482
  multiple-key access and(多码访问), 485, 506-509
  nonclustering(非聚集), 477
  operator class and(操作算子类), 1150
  Oracle and, 1162-1173
  ordered(有序的), 475-485, 523-524
  partial(部分), 1150
  partition and(划分), 1169-1171
  performance tuning and(性能调优), 1039-1041
  persistent programming language and(持久化程序设计语言), 964-972
  pointer and(指针), 546
  PostgreSQL and, 1135-1136, 1146-1151
  primary(主), 476-477, 542, 544
  query processing and(查询处理), 541-544
  record relocation and(记录重定位), 502
  search key and(搜索码), 476
  secondary (辅助), 477, 483-485, 502, 542, 544-545
  selection operation and(选择运算), 541-544
  sequential(有序的), 485-486
  sorting and(排序), 547-549
  space overhead and (空间开销), 476, 479, 486, 522
  sparse(稀疏), 477-480, 482-483
  spatial data and(空间数据), 1071-1076
  support routine and(支撑例程), 1135-1136
  tree and(树), 1148-1149 (参见 tree)
  unique(唯一性), 1149
  update and(更新), 482-483
  XML(可扩展标记语言), 1160
information-extraction system (信息抽取系统), 932-933
information gain(信息增益), 897-898
information retrieval(信息检索), 25-26, 885, 938
  adjacency test and(邻接测试), 922-923
  application of(应用), 915-917, 931-935
  category and(分类), 935-937
  defined(定义的), 915
  development of field(领域开发), 915
  directory and(目录), 935-937
  false negative and(误舍弃), 929-930
  false positive and(误选中), 929-930
  homonym and(同义词), 925-927
  indexing of document and(文档索引), 927-929
  information extraction and(信息抽取), 932-933
  keyword and(关键字), 916-927
  measuring effectiveness of(有效性度量), 929-930
  ontology and(本体), 925-927
  PageRank and, 922-923, 925
  precision and(查准率), 929-930

query result diversity and(查询结果多样性), 932
question answering and(问答), 933-934
recall and(查全率), 929-930
relevance ranking using terms(使用术语的相关性排名), 917-920
relevance using hyperlink(使用超链接的相关性), 920-925
result diversity and(结果多样性), 932
search engine spamming and(搜索引擎作弊), 924-925
similarity-based(基于相似性的), 919-920
stop word and(停用词), 918
structured data query and(结构化数据查询), 934-935
synonym and(同义词), 925-927
TF-IDF approach and(TF-IDF 方法), 917-925
Web crawler and(网络爬虫), 930-931
Ingres, 30
inheritance(继承性), 298-299
overriding method(覆写方法), 952
SQL and(结构化查询语言), 949-956
structured type and(结构化类型), 949-952
table and(表), 954-956
type and(类型), 952-953
initially deferred integrity constraint(initially deferred 完整性约束), 134
inner join(内连接), 117-120, 601
inner relation(内层关系), 550
insertion(插入), 61, 100-102
concurrency control and(并发控制), 697-701
EXEC SQL and, 171
hashing and(散列), 513, 516-523
locking and(封锁), 705
phantom phenomenon and(幻象现象), 698-701
PostgreSQL and, 1130-1131
prepared statement and(预备语句), 162-164
privilege and(权限), 143-145
transaction and(事务), 629, 653
view and(视图), 124-125
instance(实例), 8, 904-905
instead of trigger(instead of 触发器), 1161-1162
Integrated Development Environment (IDE)(集成开发环境), 111, 307, 386, 397, 426, 434, 932
integrity constraint(完整性约束), 4, 58
add(添加), 129
alter table(alter table 命令), 129
assertion(断言), 135-136
check clause(check 子句), 130, 134-136
create table(create table 命令), 129, 130
deferred(延迟的), 134
example of(例子), 128
foreign key(外码), 131-133
functional dependency and(函数依赖), 129
hashing and(散列), 809-810
not null(非空), 129-130, 133
primary key(主码), 130-131
referential(参照的), 11, 46-47, 131-136, 151, 181-182, 628
schema diagram and(模式图), 46-47
on single relation(单关系), 129
unique(唯一性), 130-131
user-defined type and(用户自定义类型), 140
violation during transaction(事务中的违反), 133-134
XML and(可扩展标记语言), 1003-1004
integrity manager(完整性管理器), 21
intention-eXclusive (IX) mode(排他型意向模式), 680
intention-Shard (IS) mode(共享型意向模式), 680
interconnection network(互连网络), 780-781
interesting sort order(感兴趣的排序顺序), 601
Interface Description Language (IDL)(接口描述语言), 1064-1065
interference(冲突), 780
internal node(内部结点), 487
International Organization for Standardization (ISO)(国际标准化组织), 57, 871, 1061
Internet(因特网), 31
direct user access and(直接用户访问), 2
wireless(无线), 1081-1082
interoperation parallelism(操作间并行), 804, 813-814
interquery parallelism(查询间并行), 802-803
intersect(相交), 81-82, 585
intersect all(全部相交), 81-82
intersection(交), 50
intersection set(交集), 960
interval(间隔), 1063-1064
intraoperation parallelism(操作内并行)
aggregation and(聚集), 811
degree of parallelism and(并行度), 804
duplicate elimination and(去重), 811

operation evaluation cost and(操作计算代价), 812
parallel external sort-merge and(并行外排序-归并), 806
parallel join and(并行连接), 806-811
parallel sort and(并行排序), 805-806
projection and(投影), 811
range-partitioning sort and(范围划分排序), 805
selection and(选择), 811
intraquery parallelism(查询内并行), 803-804
invalidation report(失效报告), 1083
inverse document frequency (IDF)(逆文档频率), 918
I/O parallelism(I/O 并行)
hashing and(散列), 799-800
partitioning technique and(划分技术), 798-800
range scheme and(范围模式), 800
round-robin scheme and(轮转法模式), 799
skew handling and(偏斜处理), 800-802
is not null(非空), 83
is not unknown(非未知), 84
is null(为空), 83
isolation(隔离性), 4, 1094
atomicity and(原子性), 646-648
cascadeless schedule and(无级联的调度), 647-648
concurrency control and(并发控制), 631, 636-637, 639, 650, 1137-1138
defined(定义的), 628
dirty read and(读脏数据), 1137
distributed transaction and(分布式事务), 830-832
factorial and(阶乘), 639
improved throughput and(改进的吞吐量), 635-636
inconsistent state and(不一致状态), 631
level of(级别), 648-653
locking and(封锁), 651
multiple version and(多版本), 652-653, 1137-1138
nonrepeatable read and(不可重复读), 1137
Oracle and, 1181-1182
phantom read and(读幻象), 1137-1138
PostgreSQL and, 1137-1138, 1142
read committed(读提交), 649, 1042
read uncommitted(读未提交), 648, 649
recoverable schedule and(可恢复调度), 647
repeated read(可重复读), 649
resource allocation and(资源分配), 635-636
row versioning and(行版本), 1244
serializability and(可串行化), 640, 648-653
snapshot(快照), 652-653, 692-697, 704, 729-730, 1042, 1242, 1244
timestamp and(时间戳), 651-652
transaction and(事务), 628, 635-640, 646-653
utilization and(利用率), 636
wait time and(等待时间), 636
is unknown(未知), 84
item shipping(项传送), 776
iteration(迭代), 176, 188-190

J++, 1228, 1253
Jakarta Project(Jakarta 项目), 386
jar file(.jar 文件), 160
Java, 14, 157, 169, 173, 387, 945
DOM API(DOM 应用程序接口), 1008-1009
JDBC and(Java 数据库连接), 158-166
metadata and(元数据), 164-166
persistent system and(持久系统), 971-972
SQLJ and, 172
Unified Modeling Language(UML) and(统一建模语言), 308
Java 2 Enterprise Edition (J2EE), 386, 1157-1158
Java Database Object (JDO)(Java 数据库对象), 971
JavaScript(Java 描述语言)
application design and(应用程序设计), 389-391, 398
Representation State Transfer (REST) and(表示状态转移), 395
security and(安全性), 402-411
JavaScript Object Notation (JSON)(JavaScript 对象表示法), 395, 863-864
JavaServer Faces (JSF) framework(JSF 构架), 397
Java Server Pages (JSP)
application design and(应用程序设计), 377, 383-391
client-side scripting and(客户端脚本), 389-391
security and(安全性), 405
server-side scripting and(服务器端脚本), 386-388
servlet and(Java 服务器端程序), 383-391
Web application framework and(Web 应用构架), 399
JBoss, 386, 399
JDBC (Java Database Connectivity)(Java 数据库连接), 380, 1052, 1154

advanced SQL and(高级 SQL), 158-159
blob column(二进制大对象列), 166
caching and(高速缓存), 400-401
callable statement and(可调用语句), 164
clob column(字符大对象列), 166
connecting to database(连接到数据库), 159-161
E-R model and(实体关系模型), 269, 275
information protocol of(信息协议), 160-161
metadata feature and(元数据特性), 164-166
prepared statement and(预备语句), 162-164
query result retrieval and(查询结果检索), 161-162
shipping SQL statement to(传送 SQL 语句到), 161
updatable result set and(可更新结果集), 166
join dependency(连接依赖), 360
join(连接)
complex(复杂的), 563
condition and(条件), 114-115
cost analysis and(代价分析), 555-557, 599-601
distributed processing and(分布式处理), 855-857
equi-join(等值连接), 549-559, 563, 566, 571, 807, 819
filtering of(过滤), 1187
fragment-and-replicate(分片—复制), 808-809
full outer(全外), 117-120, 233-234, 565-566
hash join(散列连接), 539-540, 557-562, 571-572, 602
hybrid merge(混合归并), 557
IBM DB2 and, 1209-1210
inner(内), 117-120, 601
inner relation and(内层关系), 550
left outer(左外), 116-120, 233-235, 565-566
lossless decomposition and(无损分解), 345-346
merge-join(归并连接), 553-555
minimization and(最小化), 613
natural(自然), 71-74, 87, 113(参见 natural join)
nested-loop(嵌套循环), 550-553(参见 nested-loop join)
Oracle and, 1168, 1187
ordering and(排序), 588-589
outer(外), 115-120, 232-235, 565-566, 597
outer relation(外层关系), 550
parallel(并行), 806-811, 857
partitioned(划分), 539-540, 807-810
PostgreSQL and, 1153
prediction(预测), 1267
query processing and(查询处理), 549-566, 855-857
relational algebra and(关系代数), 229-232, 239
right outer(右外), 117-120, 233-235, 565-566
semijoin strategy and(半连接策略), 856-857
size estimation and(大小估计), 595-596
sort-merge-join(排序-归并-连接), 553
theta, 584-585
type and(类型), 115-120
view maintenance and(视图维护), 609
join using(条件连接), 74, 113-114
journaling file system(日志文件系统), 439
JPEG(Joint Picture Experts Group)(联合图像专家组), 1077
jukebox system(自动光盘机系统), 431

k-d tree(k-d 树), 1071-1072
kernel function(内核函数), 901-902
key(码), 45-46
constraint and(约束), 271-272
decomposition and(分解), 354-355
encryption and(加密), 412-418
entity-relationship(E-R) model and(实体-联系模型), 271-272
equality on(相等), 542
functional dependency and(函数依赖), 330-333
hashing and(散列), 509-519, 524
indexing and(索引), 476-508, 476-509, 524, 529
multiple access and(多路访问), 506-509
nonunique(不唯一), 497-499
smart card and(智能卡), 415-416
storage and(存储), 457-459
USB(通用串行总线), 430
uniquifier and(唯一化), 498-499
keyword(关键字)
complex data type and(复杂数据类型), 947-949
homonym and(同义同), 925-927
index and(索引), 927-929
ontology and(本体), 925-927
PostgreSQL and, 1130-1131
query simplification and(查询简化), 1237-1238
ranking and(排名), 915-925
search engine spamming and(搜索引擎作弊), 924-925
stop word and(停用词), 918
synonym and(同义词), 925-927

language construct(语言结构), 176-179
Language Integrated Querying (LINQ)(语言集成查询), 1055, 1249
large-object type(大对象类型), 138
latent failure(潜在故障), 448
lazy propagation(延迟传播), 844, 868
lazywriter(SQLServer 的后台服务线程之一), 1246
LDAP Data Interchange Format (LDIF)(LDAP 数据交换格式), 872
Least Recently Used (LRU) scheme(最近最少使用模式), 466-467
left outer join (左外连接), 116-120, 233-235, 565-566
legacy system(遗产系统), 1050-1051
Lightweight Directory Access Protocol (LDAP) (轻量级目录访问协议), 406, 871-875
like(like 运算), 76-77
linear regression(线性回归), 902
linear search(线性搜索), 541-542
linear speedup(线性加速比), 778-780
Linux, 1124, 1193-1194, 1212
Local-Area Network (LANs) (局域网), 788-789, 1081
localtimestamp(当地时间戳), 137
local wait-for graph(局部等待图), 845
location-dependent query(位置相关的查询), 1080
locking protocol(封锁协议), 666
  biased(有偏), 841
  distributed lock manager(分布式锁管理器), 840
  graph-based(基于图的), 671-674
  majority(多数), 840-841
  primary copy(主拷贝), 840
  quorum consensus(法定人数同意), 841-842
  single lock-manager(单一锁管理器), 839
  timestamping(时间戳), 842-843
  two-phase(两阶段), 667-669
lock manager(锁管理器), 670-671, 773
lock(锁)
  adaptive granularity and(自适应粒度), 776
  caching and(高速缓存), 776
  call back and(回调), 776
  compatibility function and(兼容性函数), 662
  concurrency control and(并发控制), 661-674
  deadlock and (死锁), 665-666, 674-679, 839, 841, 844-847, 1217-1220, 1243-1246
  distributed database and(分布式数据库), 839-847
  dynamic(动态的), 1243
  exclusive(排他), 651, 661-662, 668-669, 672-673, 679, 691, 698-702, 706-710, 729-730, 740-741, 803, 839, 841
  false cycle and(假环), 846-847
  fine-grained(细粒度), 756
  granting of(授予), 666-667
  growing phase and(增长阶段), 667-669
  IBM DB2 and, 1217-1220
  implementation of(实现), 670-671
  intention mode and(意向模式), 680
  logical undo operation and (逻辑 undo 操作), 744-750
  long-duration transaction and(长事务), 1110-1111
  lower/higher level(更低/更高级别), 745
  Microsoft SQL Server and, 1242-1244, 1246
  multiple granularity and(多粒度), 679-682
  multiversion scheme and(多版本机制), 691-692
  PostgreSQL and, 1143-1145
  recovery system and(恢复系统), 744-750
  request operation and(请求操作), 662-671, 675-680, 709
  shared(共享的), 661, 841
  shrinking phase and(收缩阶段), 667-669
  starvation and(饿死), 679
  timestamp and(时间戳), 682-686
  transaction server and(事务服务器), 773-775
  true matrix value and(真矩阵值), 662
  wait-for graph and(等待图), 676-678, 845-847
log disk(日志磁盘), 438-439
logical clock(逻辑时钟), 843
logical counter(逻辑计数器), 682
logical-design phase (逻辑设计阶段), 16, 260-261
logical error(逻辑错误), 721
logical logging(逻辑日志), 745-746, 1115
logical operation(逻辑运算)
  consistency and(一致性), 746
  early lock release and(锁的提前释放), 744-750
  rollback and(回滚), 746-749
  undo log record(undo 日志记录), 745-750
logical row-id(逻辑行 id), 1164-1165
logical undo operation(逻辑 undo 操作), 745-750
log record(日志记录)
  ARIES and, 750-756
  buffering and(缓冲), 738-739
  Compensation Log Record (CLR) and(补偿日志记录), 751-752, 754

identifier and(标识), 727
old/new value and(旧/新值), 727-728
physical(物理的), 745
recovery system and(恢复系统), 726-728, 730-734
redo and(重做), 729-734
steal/no-steal policy and(抢占/非抢占策略), 740
undo(撤销), 729-734, 745-746
Write-Ahead Logging (WAL) rule and(先写日志规则), 739-741
Log Sequence Number (LSN)(日志序列号), 750-755
log writer process(写日志进程), 773-774
long-duration transaction(长事务)
compensation transaction and(补偿事务), 1113-1114
concurrency control and(并发控制), 1111-1112
graph-based protocol and(基于图的协议), 1110
implementation issue(实现问题), 1114-1115
multilevel(多级别), 1111-1112
nesting and(嵌套), 1111-1112
nonserializable execution and(不可串行化的执行), 1110-1111
operation logging and(操作日志), 1115
performance and(性能), 1110
recoverability and(可恢复性), 1110
subtask and(子任务), 1109
timestamp and(时间戳), 1110
two-phase locking and(两阶段封锁), 1110
uncommitted data and(未提交数据), 1109
lookup(查找), 600, 1086
concurrency control and(并发控制), 700, 704-708
distributed database and(分布式数据库), 865, 867, 870
fuzzy(模糊), 890, 1266
index and(索引), 482, 485-500, 505-513, 516-518, 522, 524
Microsoft SQL Server and, 1238, 1241, 1266
PostgreSQL and, 1148
query processing and(查询处理), 544, 552-553
lossless-join decomposition(无损连接分解), 345-346
lossy decomposition(有损分解), 345-346
lost update(丢失修改), 692

machine learning(机器学习), 25-26
magnetic disk(磁盘), 430
block and(块), 436-439
buffering and(缓冲), 437-438
checksum and(校验和), 434
crash and(崩溃), 434
data-transfer rate and(数据传输率), 435-436
disk controller and(磁盘控制器), 434
failure classification and(故障分类), 722
hybrid(混合), 440-441
log disk and(日志磁盘), 438-439
mean time to failure and(平均故障时间), 436
optimization of disk-block access and(磁盘块访问优化), 436-439
parallel system and(并行系统), 781-782
performance measure of(性能度量), 435-436
physical characteristic of(物理特征), 432-435
read-ahead and(预读), 437
read-write head and(读-写磁头), 432-435
recording density and(记录密度), 433-434
Redundant Array of Independent Disk (RAID) and(独立磁盘冗余阵列), 441-449
scheduling and(调度), 437
scrubbing and(擦洗), 448
sector and(扇区), 432-434
seek-time and(寻道时间), 435-436
size of(大小), 433
main-memory database system(主存数据库系统), 724n1
majority protocol(多数协议), 840-841, 848-849
man-in-the-middle attack(中间人攻击), 406
many server, many-router model(多服务器多路由器模型), 1094
many-server, single-router model(多服务器单路由器模型), 1093
many-to-many mapping(多对多映射), 270, 276-277
many-to-one mapping(多对一映射), 270, 276
mapping cardinality(映射基数), 269-270, 276-277
markup language(标记语言). 参见 specific language
file processing and(文件处理), 981-982
structure of(结构), 981-990
tag and(标签), 982-985
transaction and(事务), 983-985
master-slave replication(主-从复制), 843-844
master table(主表), 1032-1033
materialization(物化), 567-568

Materialized Query Table (MQT) (物化查询表), 1212-1214, 1221
materialized view(物化视图), 123-124, 607
  aggregation and(求集), 610-611
  IBM DB2 and, 1212-1214
  index selection and(索引选择), 612
  join operation and(连接运算), 609
  maintenance and(维护), 608-611
  Oracle and, 1171-1172, 1174, 1188
  performance tuning and(性能调整), 1039-1040
  projection and(投影), 609-610
  query optimization and(查询优化), 611-612
  replication and(复制), 1251-1253
  selection and(选择), 609-610
max(max 函数), 84, 86, 96, 236, 566-567
Mean Time To Failure (MTTF) (平均故障时间), 436
Media Access Control (MAC) (媒体访问控制), 1129
mediator(中间件), 859-860, 1018-1019
memory(内存). 参见 storage
  buffer and(缓冲区), 1184 (参见 buffer)
  bulk loading of index and(索引批量加载), 503-504
  cache(高速缓存), 429, 817-818 (参见 caching)
  data access and(数据访问), 724-726
  flash(闪存), 403, 430, 439-441, 506
  force output and(强制输出), 725-726
  magnetic-disk(磁盘), 430, 432-439
  main(主要), 429-430
  main-memory database and (主存数据库), 1105-1108
  Microsoft SQL Server and, 1246-1247
  multitasking and(多任务调度), 1092-1095
  .NET Common Language Runtime (CLR) and(.NET 公共语言运行库), 1255-1256
  nonvolatile random-access (非易失性随机存取), 438
  optical(光盘), 430
  Oracle structure and(Oracle 结构), 1183-1184
  overflow and(溢出), 560
  persistent programming language and(持久化程序设计语言), 964-972
  query cost and(查询代价), 544
  recovery system and(恢复系统), 724-726 (参见 Recovery system)
  redo log buffer and(redo 日志缓冲区), 1184
  shared pool(共享池), 1184
merge-join(归并连接), 553
merge-purge operation(合并-清除操作), 890-891
merging(归并)
  complex(复杂的), 1173-1174
  duplicate elimination and(消除重复), 563-564
  Oracle and, 1173-1174
  parallel external sort-merge and(并行外排序-归并), 806
  performance tuning and(性能调整), 1033
  query processing and(查询处理), 547-549, 553-555, 557
mesh system(网格系统), 780-781
message delivery process(消息传递处理), 838
metadata(元数据), 12, 164-166
Microsoft(微软), 3, 31, 141
  advanced SQL and(高级 SQL), 160n3, 169, 173, 180, 184, 197, 205
  application design and(应用程序设计), 387, 395-401, 406-407
  distributed database and(分布式数据库), 863
  parallel database and(并行数据库), 816
  query optimization and(查询优化), 612
  storage and(存储), 438
Microsoft Active Server Pages (ASP), 397
Microsoft Database Tuning Assistant, 1040
Microsoft Distributed Transaction Coordinator (MS DTC), 1242
Microsoft Office, 55, 399, 1016
Microsoft SQL Server, 1042, 1121
  business intelligence and(商务智能), 1263-1267
  compilation and(编译), 1236-1237
  compression and(压缩), 1236
  concurrency control and(并发控制), 1241-1246
  data access and(数据访问), 1248-1250
  database mirroring and(数据库镜像), 1245-1246
  data mining and(数据挖掘), 1266-1267
  data type and(数据类型), 1229-1230
  design tool and(设计工具), 1223-1228
  development of(开发), 1223
  filegroup and(文件组), 1233-1234
  indexing and(索引), 1231-1236
  lock and(锁), 1242-1244
  management tool and(管理工具), 1223-1228
  memory management and(内存管理), 1246-1247
  page unit and(页单元), 1233-1234
  partition and(划分), 1235
  Query Editor(查询编辑器), 1224-1225

query processing and (查询处理), 1223-1231, 1236-1241, 1250-1251
read-ahead and(预读), 1235-1236
recovery and(恢复), 1241-1246
reordering and(重排序), 1238-1239
replication and(复制), 1251-1253
routine and(例程), 1231
security and(安全性), 1247-1248
server programming in .NET(.NET 服务器编程), 1253-1258
snapshot isolation and(快照隔离), 1242, 1244
SQL Profiler and, 1225-1227
SQL Server Broker and (SQL Server 代理), 1261-1263
SQL Server Management Studio and, 1223-1224, 1227-1228
SQL variation and(SQL 变化), 1228-1233
storage and(存储), 1233-1236
system architecture(系统体系结构), 1246-1248
table and(表), 1234
thread pooling and(线程池), 1246
trigger and(触发器), 1232-1233
tuning and(调整), 1224, 1227
type and(类型), 1257-1258
update and(更新), 1232-1233, 1239
Windows Mobile and, 1223
XML support and(XML 支持), 1258-1261
Microsoft Transaction Server, 1091
Microsoft Window, 195, 426, 1078
IBM DB2 and, 1193-1194, 1212
PostgreSQL and, 1124, 1155
SQL Server and, 1223-1224, 1228, 1242, 1246-1248
storage and(存储), 438
min(min 函数), 84, 86, 236, 566-567
minpctused(minpctused 指令), 1203
minus(minus 操作), 82n7
mirroring(镜像), 441-442, 444, 1245-1246
mobility(移动性), 1062, 1086
broadcast data and(广播数据), 1082-1083
consistency and(一致性), 1083-1085
disconnectivity and(断连), 1083-1085
handoff and(交接), 1081
invalidation report and(失效报告), 1083
mobile computer model and(移动计算机模型), 1080-1082
query and(查询), 1082
recoverability and(可恢复性), 1083
routing and(路由), 1082
update and(更新), 1083-1084
version-numbering scheme and(版本编号方案), 1083-1084
wireless communication and(无线通信), 1080-1082
Model-View-Control design(模型-视图-控制设计), 1157-1158
Most Recently Used (MRU) scheme(最近最常使用模式), 467
most-specific type(最明确类型), 953
MPEG (Moving Picture Experts Group)(移动图像专家组), 1077-1078
multicore processor(多核处理器), 817-819
multidatabase system(多数据库系统), 857-861
multidimensional data(多维数据), 199
multimaster replication(多主副本), 844
multimedia data(多媒体数据), 1062
multimedia database(多媒体数据库), 1076-1079
multiple granularity(多粒度)
concurrency control and(并发控制), 679-682
hierarchy definition for(层次定义), 679
Intention-eXclusive (IX) mode and(意向排他模式), 680
Intention-Shared (IS) mode and(意向共享模式), 680
locking protocol and(封锁协议), 681-682
Shared and Intention-eXclusive (SIX) mode and(共享意向排他模式), 680
tree architecture and(树体系结构), 679-682
multiple-key access(多码访问), 506-509
multiquery optimization(多查询优化), 614
multiset relational algebra(多重集关系代数), 238
multiset type(多重集合类型), 956-961
multisystem application(多系统应用), 1096
multitable cluster file organization(多表聚簇文件组织), 458, 460-462
multitasking(多任务调度), 771, 1092-1095
multithreading(多线程), 817-818, 1093
multivalued attribute (多值属性), 267-268, 327-329
multivalued dependency(多值依赖), 355-360
MultiVersion Concurrency Control (MVCC)(多版本并发控制)
DDL command and(数据定义语言命令), 1144-1145
DML command and(数据操纵语言命令),

1138-1139
implementation of(实现), 1139-1143
implication of using(使用……的推理), 1143-1144
index and(索引), 1145
isolation level and(隔离性级别), 1137-1138
lock and(锁), 1145
recovery and(恢复), 1145-1146
schema for(模式), 689-692
multiway split(多路分割), 898
MySQL, 31, 76, 111, 160n3, 1123, 1155

Naive Bayesian classifier(朴素贝叶斯分类器), 901, 1191, 1266
naive user(无经验的用户), 27-28
name server(名字服务器), 829
NAND flash memory(NAND 快闪存储器), 430, 440-441
natural join(自然连接), 49-50, 87, 113
condition and(条件), 114-115
full outer(全外), 117-120, 233-234, 565-566
inner(内), 117-120, 601
left outer(左外), 116-120, 233-235, 565
on condition and(on 条件), 114-115
outer(外), 115-120
SQL query and(SQL 查询), 71-74
relational algebra and(关系代数), 229-232
right outer(右外), 117-120, 233-234, 565-566
type and(类型), 115-120
nearest-neighbor query(最近邻居查询), 1070-1071
negation(取反), 595
nested-loop join(嵌套循环连接), 1071
IBM DB2 and, 1209-1210
Oracle and, 1173
parallel(并行), 807, 810-811
PostgreSQL and, 1152-1153
query optimization and(查询优化), 600n2, 602, 604
query processing and(查询处理), 550-555, 558-560, 565, 571, 573
nested subquery(嵌套子查询)
application development and(应用程序开发), 1031, 1047
duplicate tuple and(重复元组), 94-95
empty relation and(空关系), 93-94
from clause and(from 子句), 95-96
optimization of(优化), 605-607
scalar(标量), 97-98
set operation and(集合运算), 90-93
with clause and(with 子句), 97
nesting(嵌套)
ARIES and, 755-756
concurrency control and(并发控制), 679, 709
granularity and(粒度), 679
IBM DB2 and, 1197, 1209-1210, 1218
long-duration transaction and(长事务), 1112-1113
object-based database and(基于对象的数据库), 945, 948, 958-961
Oracle and, 1159, 1164, 1182
query and(查询), 601-607, 1004-1007, 1013-1014, 1017
transaction and(事务), 1091, 1112-1113, 1116, 1218
XML and(可扩展标记语言), 27, 943, 984-998, 1001, 1004-1007, 1010
.NET, 169
NetBean, 386, 397
.NET Common Language Runtime(CLR)(.NET 公共语言运行库)
aggregate and(聚集), 1257-1258
basic concept of(基本概念), 1254
extensibility contract and(扩展性约定), 1256-1258
Microsoft SQL Server and, 1253-1258
routine and(例程), 1256-1257
SQL hosting and(SQL 宿主), 1254-1256
table function and(表函数), 1256-1257
type and(类型), 1257-1258
Netezza, 816
Network(网络)
data model and(数据模型), 9, 1080-1082
local area(局域), 788-789, 1081
mobility and(移动性), 1079-1085
wide-area type and(广域类型), 788, 790-791
nextval for, 1043
node(结点). 参见 storage
B+-tree and(B+树), 485-506
coalescing(凝聚), 491, 706
distributed system and(分布式系统), 784
IBM DB2 and, 1200-1201
mesh architecture and(网格体系结构), 780-781
multiple granularity and(多粒度), 679-682
nonleaf(非叶), 487
overfitting and(过度适应), 899-900
splitting of(分裂), 491, 706

update and(更新), 491-500
XML and(可扩展标记语言), 998
no-force policy(非强制策略), 739-740
nonacceptable termination state(不可接受的终止状态), 1099
nonclustering(非聚集), 477
nondeclarative action(非描述性动作), 158
nonleaf node(非叶结点), 487
nonprocedural language(非过程化语言), 47-48
nonrepeatable read(不可重复读), 1137
nonrepudiation(不可否认性), 416
nonunique search key(不唯一的搜索码), 497-499
nonvolatile random-access memory (NVRAM)(非易失性随机存取存储器), 438
nonvolatile storage(非易失性存储器), 432, 632, 722, 724-726, 743-744
nonvolatile write buffer(非易失性写缓冲区), 438
NOR flash memory(NOR 闪存存储器), 430, 439
normal form(范式), 18
atomic domain and(原子域), 327-329
Boyce-Codd, 333-336, 349-352, 354-356
complex data type and(复杂数据类型), 947
domain-key(域码), 360
fifth(第五), 360
first(第一), 327-329
fourth(第四), 356, 358-360
higher(更高级), 337-338
join dependency and(连接依赖), 360
project-join(投影连接), 360
second(第二), 361
third(第三), 336-337
normalization(规范化), 16, 18-20
denormalization and(解除规范化), 363-364
entity-relationship (E-R) model and(实体-联系模型), 361-362
performance and(性能), 363-364
relational database design and(关系数据库设计), 361-362
no-steal policy(非抢占策略), 740
not connective(not 连词), 86
not exist(不存在), 93, 192
not in(不在内), 90-91, 92n8
not null(非空), 61, 83, 129-130, 133, 140
not unique(非唯一), 95
null bitmap(空位图), 456
null value(空值), 19, 83-84
aggregation with(聚集), 89-90
attribute and(属性), 268-269
decode and(解码), 208
decorrelation and(去除相关), 1174
file organization and(文件组织), 451-468
integrity constraint and(完整性约束), 128-130, 133-134
left outer join(左外连接), 233
OLAP and(联机分析处理), 202
query simplification and(查询简化), 1237-1238
right outer join(右外连接), 234-235
temporal data and(时态数据), 364-367
user-defined type and(用户自定义类型), 140
numeric(数值型), 59, 62
nvarchar, 60
N-way merge(*N* 路归并), 547

object-based database(基于对象的数据库), 975
array type and(数组类型), 956-961
collection volume and(集合体), 957-958
complex data type and(复杂数据类型), 946-949
correspondence and(对应), 955
feature implementation and(特性的实现), 963-964
inheritance and(继承), 949-956
mapping and(映射), 973
multiset type and(多重集合类型), 956-961
nesting and(嵌套), 945, 948, 958-961
object-identity type and(对象标识类型), 961-963
object-oriented vs. object-relational approach and(面向对象和对象-关系方法), 973-974
persistent programming language and(持久化程序设计语言), 964-972, 974
reference type and(引用类型), 961-963
relational data model and(关系数据模型), 945
structured type and(结构化类型), 949-953
unnesting and(解除嵌套), 958-961
Object Database Management Group (ODMG) 对象数据库管理小组, 1054-1055
object-oriented database(面向对象数据库), 393
object-oriented data model (面向对象数据模型), 27
object-relational data model (对象-关系数据模型), 27
object-relational mapping(对象-关系映射), 393, 946, 973
observable external write (可见的外部写), 634-635
ODBC (Open Database Connectivity) (开放数据库

互连), 380, 1052
advanced SQL and(高级 SQL), 166-169
API definition and(应用程序接口定义), 166-167
caching and(高速缓存), 401
conformance level and(一致性级别), 168-169
Microsoft SQL Server and, 1249
PostgreSQL and, 1154
standard for(标准), 1053-1054
type definition and(类型定义), 168
OLAP (OnLine Analytical Processing)(联机分析处理), 1046
all attribute and(所有属性), 201-203, 205
application for(应用), 197-201
cross-tabulation and(交叉表), 199-203, 205, 210
data cube and(数据立方体), 200, 206-210
decode function and(解码函数), 208
dicing and(切块), 201
drill down and(下钻), 201
implementation of(实现), 204
Microsoft SQL Server and, 1223, 1266
multidimensional data and(多维数据), 199
null value and(空值), 202
Oracle and, 1161
order by clause and(order by 子句), 205
pivot clause and(pivot 子句), 205, 210
relational table and(关系表), 202-203, 205
rollup and(上卷), 201, 206-210
slicing and(切片), 201
in SQL(用 SQL), 205-209
OLE-DB(对象连接和嵌套数据库), 1249
OLTP(OnLine Transaction Processing)(联机事务处理), 1046-1047, 1165, 1186, 1264
on condition(on 条件), 114-115
on delete cascade(级联删除), 133, 185
one-to-many mapping(一对多映射), 269, 276
one-to-one mapping(一对一映射), 269, 276
ontology(本体), 925-927
OOXML (Office Open XML), 1016
Open Document Format (ODF)(开放文档格式), 1016
open statement(open 语句), 170-171
operation logging(操作日志), 1115
operator tree(运算符树), 803-804
optical storage(光盘存储器), 430-431, 449-450
optimistic concurrency control without read validation(无读有效性检查的乐观并发控制), 704
Oracle, 3, 30, 216, 1121
access path selection and(访问路径选择), 1174
analytic workspace and(分析工作区), 1161
archiver and(归档进程), 1185
caching and(高速缓存), 1179-1180, 1184
checkpoint and(检查点), 1185
cluster and(聚类), 1173, 1186
compression and(压缩), 1165
concurrency control and(并发控制), 1180-1183
database administration tool and(数据库管理工具), 1189-1191
database design and(数据库设计), 355, 386, 396, 408n5, 409, 1157-1158
database writer(数据库写进程), 1185
data guard(数据卫士), 1183
data mining and(数据挖掘), 1191
data warehousing and(数据仓库), 1158
dedicated server and(专用服务器), 1183-1185
dimensional modeling and(维度建模), 1160, 1171
distribution and(分布), 1188-1189
encryption and(加密), 1165-1166
Exadata and, 1187-1188
external data and(外部数据), 1188-1189
hashing and(散列), 1170
index and(索引), 1162-1173
isolation level and(隔离性级别), 1181-1182
join and(连接), 1168, 1187
logical row-id and(逻辑行标识), 1164-1165
log writer and(写日志进程), 1185
materialized view and(物化视图), 1171-1172, 1174, 1188
memory structure and(内存结构), 1183-1184
optimizer of(优化器), 1174-1176
parallel execution and(并行执行), 1178
partition and(划分), 1169-1172, 1176
process monitor and(进程监控器), 1185
process structure and(进程结构), 1184-1185
projection and(投影), 1187
query optimization and(查询优化), 582, 593, 603-604, 612, 1173-1178
query processing and(查询处理), 1157-1158, 1162-1172
Real Application Cluster (RAC) and, 1186
recovery and(恢复), 1180-1183, 1185
replication and(复制), 1188-1189
result caching and(结果高速缓存), 1179-1180
security and(安全性), 1165-1166

segment and(段), 1163
serializability and(可串行化), 1181-1182
shared server and(共享服务器), 1185
as Software Development Laboratory(软件开发实验室), 1157
SQL basics and(SQL 基础), 55, 75n4, 82n7, 96, 141, 160-161, 172-174, 178, 180, 184-185, 197, 205
SQL Loader and(SQL 加载器), 1189
SQL Plan Management and(SQL 计划管理), 1177-1178
SQL Tuning Advisor(SQL 调优顾问), 1176-1177
SQL variation and(SQL 变种), 1158-1162
subquery flattening and(子查询整平), 1174
system architecture(系统体系结构), 795, 803, 843, 1183-1188
system monitor(系统监控器), 1185
table and(表), 1163-1166, 1172-1173, 1187, 1189
transaction and(事务), 649, 653, 692-693, 697, 710, 718
transformation and(转换), 1173-1174
tree and(树), 1191
trigger and(触发器), 1161-1162
update and(更新), 1179-1180
virtual private database and(虚拟专用数据库), 1166
XML DB and, 1159-1160
Oracle Application Development Framework (ADF) | Oracle 应用开发框架|, 1157-1158
Oracle Automatic Storage Manager(Oracle 自动存储管理器|, 1186-1187
Oracle AutomaticWorkload Repository (AWR) | Oracle 自动负载存储器|, 1190
Oracle Business Intelligence Suite (OBI) | Oracle 商务智能套件|, 1158
Oracle Database Resource Management(Oracle 数据库资源管理|, 1190-1191
Oracle Designer, 1158
Oracle EnterpriseManager (OEM) | Oracle 企业管理器|, 1190
Oracle JDeveloper, 1158
Oracle Tuxedo, 1091
or connective(or 连词), 66
order by(order by 子句), 77-78, 193
organize by dimension(按维组织), 1204
or operation(or 运算|, 83-84
outer-join (外连接), 115-120, 232-235, 565-566, 597
outer relation(外层关系), 550
overfitting(过度适应|, 899-900
overflow avoidance(溢出避免|, 560
overflow bucket(溢出桶), 512-514
overflow resolution(溢出分解|, 560
overlapping entity set(重叠实体集), 300
overlapping specialization(重叠特化), 296-297
overloading(重载|, 968

P+Q redundancy schema(P+Q 冗余模式|, 446
PageLSN(日志页顺序号), 751, 753, 754
PageRank, 922-925, 926
page shipping(页面传送), 776
parallel database(并行数据库)
cache memory and(缓存内存), 817-818
cost of(代价|, 797
decision-support query and(决策支持查询), 797
failure rate and(故障率), 816
increased use of(增长的使用|, 797
interoperation parallelism and(操作间并行), 813-814
interquery parallelism and(查询间并行), 802-803
intraoperation parallelism and(操作内并行), 804-812
intraquery parallelism and(查询内并行), 803-804
I/O parallelism and(I/O 并行), 798-802
Massively Parallel Processor(MPP) and(大规模并行处理器), 1193
multicore processor and(多核处理器), 817-819
multithreading and(多线程), 817-818
operator tree and(运算符树), 803-804
Oracle and, 1178-1179
partitioning technique and(划分技术), 798-799
pipeline and(流水线|, 814-815
query optimization and(查询优化), 814-817
raw speed and(原始速度), 817
skew and(偏斜), 800-801, 805-808, 812, 814, 819
success of(成功), 797
system design and(系统设计), 815-817
parallel external sort-merge(并行外排序-归并), 806
parallelism(并行|, 442-444
parallel join(并行连接|, 806, 857
fragment-and-replicate(分片-复制), 808-809
hash(散列), 809-810

nested-loop(嵌套循环), 810-811
partitioned(划分), 807-810
parallel processing(并行处理), 401-402
parallel system(并行系统)
coarse-grain(粗粒度), 777
fine-grain(细粒度), 777
hierarchical(层次), 781
interconnection network and(互连网络), 780-781
interference and(冲突), 780
massively parallel(大规模并行), 777-778
scaleup and(扩展比), 778-780
shared disk(共享磁盘), 781
shared memory(共享内存), 781-783
shared nothing(无共享), 781
skew and(偏斜), 780
speedup and(加速比), 778-780
start-up cost and(启动代价), 780
throughput and(吞吐量), 778
parameter style general(parameter style general 语句), 179
parametric query optimization(参数化查询优化), 615
parity bit(奇偶校验位), 444-446
parsing(解析)
application design and(应用程序设计), 388
bulk load and(批量加载), 1031-1033
query processing and(查询处理), 537-539, 572, 1236-1237
participation constraint(参与约束), 270
partitioning vector(划分向量), 798-799
partition(划分)
attribute and(属性), 896-897
availability and(可用性), 847-853
balanced range(平衡的范围), 801
classifier and(分类器), 896-897
cloud computing and(云计算), 865-866
composite(复合), 1170-1171
condition and(条件), 896-897
distributed database and(分布式数据库), 832, 835
hash(散列), 798-799, 807, 1170
join and(连接), 807-810
list(列表), 1170
Microsoft SQL Server and, 1235
Oracle and, 1169-1172, 1176
point query and(点查询), 799
pruning and(剪枝), 1176
query optimization and(查询优化), 814-815
range(范围), 798-800, 805, 1170
reference(引用), 1171
round-robin(轮转法), 798-801
scanning a relation and(扫描关系), 799
Partner Interface Process (PIP)(伙伴接口过程), 1055
password(密码). 参见 security
application design and(应用程序设计), 376, 382, 385, 393, 405-407, 415
dictionary attack and(字典攻击), 414
distributed database and(分布式数据库), 871
leakage of(泄露), 405
man-in-the-middle attack and(中间人攻击), 406
one-time(一次性), 406
single sign-on system and(单点登录系统), 406-407
SQL and(结构化查询语言), 142, 160, 168, 170
storage and(存储), 463-464
PATA (parallel ATA)(并行 ATA), 434
pctfree(pctfree 指令), 1203
performance(性能)
access time and(访问时间), 431-439, 447, 450-451, 476, 479, 523, 540-541, 817
application design and(应用程序设计), 400-402
B+-tree and(B+树), 485-486
caching and(高速缓存), 400-401
data-transfer rate and(数据传输率), 435-436
denormalization and(解除规范化), 363-364
magnetic disk storage and(磁盘存储), 435-436
parallel processing and(并行处理), 401-402
response time and(响应时间), 400(参见 response time)
seek time and(寻道时间), 433, 435-439, 450-451, 540, 555
sequential index and(顺序索引), 485-486
transaction time and(事务时间), 365n8, 1062
web application and(Web 应用), 377-382
performance benchmark(性能基准)
database-application class and(数据库应用类), 1046
suite of task(任务集), 1045-1046
Transaction Processing Performance Council (TPC)(事务处理性能委员会), 1046-1048
performance tuning(性能调整)
bottleneck location and(瓶颈位置), 1033-1035
bulk load and(批量加载), 1031-1033
concurrent transaction and(并发事务), 1041-1044

hardware and(硬件), 1035-1038
index and(索引), 1039-1041
materialized view and(物化视图), 1039-1041
parameter adjustment and(参数调整), 1029-1030, 1035
physical design and(物理设计), 1040-1041
RAID choice and(独立磁盘冗余阵列选择), 1037-1038
of schema(模式), 1038-1039
set orientation and(面向集合), 1030-1031
simulation and(模拟), 1044-1045
update and(更新), 1030-1033
Perl(Perl 语言), 180, 387, 1154
persistent messaging(持久消息), 836-837
persistent programming language(持久化程序设计语言), 974
approach for(方法), 966-967
byte code enhancement and(字节代码加强), 971
C++(C++语言), 968-971
class extent and(类区间), 969, 972
database mapping and(数据库映射), 971
defined(定义的), 965
iterator interface and(迭代接口), 970
Java(Java 语言), 971-972
object-based database and(基于对象的数据库), 964-972, 974
object identity and(对象标识), 967
object persistence and(对象持久化), 966-968
overloading and(重载), 968
persistent object and(持久对象), 969
pointer and(指针), 967, 969, 972
reachability and(可达性), 971
relationship and(关系), 969
simple reference type and(单引用类型), 972
transaction and(事务), 970
update and(更新), 970
person-in-the-middle attack(中间人攻击), 1105
phantom phenomenon(幻象现象), 698-701
phantom read(读幻象), 1137-1138, 1142, 1217-1218
PHP(PHP 脚本语言), 387-388
physical data independence(物理数据独立性), 6
physical-design phase(物理设计阶段), 16, 261
physiological redo(物理逻辑 redo), 750
pinned block(被钉住的块), 465
pipelining(流水线), 539, 568
demand-driven(需求驱动的), 569-570
double-pipelined hash-join and(双流水线散列连接), 571-572
parallel database and(并行数据库), 813-815
producer-driven(生产者驱动的), 569-571
pulling data and(下拉数据), 570-571
pivot clause(pivot 子句), 205, 210, 1230
plan caching(计划高速缓存), 605
PL/SQL, 173, 178
pointer(指针). 参见 index
application design and(应用程序设计), 409
child node and(子结点), 1074
concurrency control and(并发控制), 706-707
IBM DB2 and, 1199, 1202-1203
information retrieval and(信息检索), 936
main-memory database and(主存数据库), 1107
multimedia database and(多媒体数据库), 1077
Oracle and, 1165
persistent programming language and(持久化程序设计语言), 967, 969, 972
PostgreSQL and, 1134, 1147-1148
query optimization and(查询优化), 612
query processing and(查询处理), 544-546, 554
recovery system and(恢复系统), 727, 754
SQL basics and(SQL 基础), 166, 179-180
storage and(存储), 439, 452-462
point query(点查询), 799
polymorphic type(多态类型), 1128-1129
popularity ranking(流行度排名), 920-925
PostgreSQL, 31, 1121
access method and(访问方法), 1153
aggregation and(聚集), 1153
command-line editing and(命令行编辑), 1124
concurrency control and(并发控制), 692, 697, 701, 1137-1145
constraint and(约束), 1130-1131, 1153-1154
DML command and(数据操纵语言命令), 1138-1139
extensibility(可扩展性), 1132
function(函数), 1133-1135
Generalized Inverted Index(GIN) and(通用倒排索引), 1149
Generalized Search Tree(GiST) and(通用搜索树), 1148-1149
hashing and(散列), 1148
index and(索引), 1135-1136, 1146-1151
isolation level and(隔离性级别), 1137-1138, 1142

join and(连接), 1153
lock and(锁), 1143-1145
major releases of(主要发行版本), 1123-1124
MultiVersion Concurrency Control (MVCC) and(多版本并发控制), 1137-1146
operator class and(类操算子类), 1150
operator statement and(运算符语句), 1136
parallel database and(并行数据库), 816-817
performance tuning and(性能调整), 1042
pointer and(指针), 1134, 1147-1148
procedural language and(过程化语言), 1136
query optimization and(查询优化), 582, 593
query processing and(查询处理), 1151-1154
recovery and(恢复), 718
rollback and(回滚), 1142-1144
rule and(规则), 1130-1131
serializability and(可串行化), 1142-1143
server programming interface(服务器编程接口), 1136
sort and(排序), 1153
SQL basics and(SQL 基础), 140, 160, 173, 180, 184
state transition and(状态转换), 1134
storage and(存储), 1146-1151
system architecture(系统体系结构), 1154-1155
system catalog and(系统目录), 1132
transaction management in(事务管理), 649, 653, 1137-1146
tree and(树), 1148-1149
trigger and(触发器), 1153-1154
trusted/untrusted language and(可信/不可信语言), 1136
tuple ID and(元组标识符), 1147-1148
tuple visibility and(元组可见性), 1139
type(类型), 1126-1129, 1132-1133
updateable(更新), 1130, 1141-1144, 1147-1148
user interface(用户界面), 1124-1126
vacuum(vacuum 命令), 1143
precedence graph(优先图), 644
precision(查准率), 903
predicate read(谓词读), 697-701
prediction(预测)
classifier and(分类器), 894-904
data mining and(数据挖掘), 894-904
join and(连接), 1267
prepared statement(预备语句), 162-164
presentation facility(表示设施), 1094-1095
presentation layer(表示层), 391
prestige ranking(威望度排名), 920-925, 930-931
primary copy(主拷贝), 840
primary key(主码), 45-46, 60-62
decomposition and(分解), 354-355
entity-relationship (E-R) model and(实体-联系模型), 271-272
functional dependency and(函数依赖), 330-333
integrity constraint and(完整性约束), 130-131
primary site(主站点), 756
privacy(隐私), 402, 410-411, 418, 828, 869-870, 1104
privilege(权限)
all(全部), 143-144
execute and(执行), 147
granting of(授予), 143-145
public, 144
revoking of(撤回), 143-145, 149-150
transfer of(转移), 148-149
procedural DML(过程性数据操纵语言), 10
procedural language(过程性语言), 20
advanced SQL and(高级 SQL), 157-158, 173, 178
IBM DB2 and, 1194
Oracle and, 1160, 1191
PostgreSQL and, 1130, 1133, 1136
relational model and(关系模型), 47-48
procedure(过程)
declaring(声明), 174-175
external language routine and(外部语言例程), 179-180
language construct for(语言结构), 176-179
syntax and(语法), 173-174, 178
writing in SQL(用 SQL 写), 173-180
producer-driven pipeline(生产者驱动流水线), 569-570
Program Global Area (PGA)(程序全局区), 1183
programming language(编程语言). 参见 specific language
accessing SQL from(访问 SQL), 157-173
mismatch and(不匹配), 158
variable operation of(变量操作), 158
projection(投影)
intraoperation parallelism and(操作内并行), 811
Oracle and, 1187
query and(查询), 564, 597
view maintenance and(视图维护), 609-610

Project-Join Normal Form (PJNF) (投影连接范式), 360
project operation(投影运算), 219
PR quadtree(PR 四叉树), 1073
pseudotransitivity rule(伪传递律), 339
public-key encryption(公钥加密), 412-414
publishing(发布), 1013, 1251-1253
pulling data(下拉数据), 570-571
purity(纯度), 897
Python(Python 语言), 180, 377, 387, 1123, 1125, 1136

QBE, 37, 245, 770
quadratic split(二次方分裂), 1075-1076
quadtree(四叉树), 1069, 1072-1073
query(查询), 10
  ADO.NET and, 169
  availability and(可用性), 826-827
  B+-tree and(B+树), 488-491
  basic structure of SQL(SQL 基本结构), 63-71
  caching and(高速缓存), 400-401
  Cartesian product and(笛卡儿积), 50-51, 68-69, 71-75, 120, 209, 217, 222-229, 232, 573, 584, 589, 595-596, 606, 616
  complex data type and(复杂数据类型), 946-949
  correlated subquery and(相关子查询), 93
  Data-Definition Language (DDL) and(数据定义语言), 21-22
  Data-Manipulation Language (DML) and(数据操纵语言), 21-22
  decision-support(决策支持), 797
  delete and(删除), 98-100
  distributed database and(分布式数据库), 825-878 (参见 distributed database)
  hashing and(散列), 475, 516-522 (参见 hash function)
  index and(索引), 475 (参见 indices)
  information retrieval and(信息检索), 915-938
  insert and(插入), 100-101
  intermediate SQL and(中级 SQL), 113-151
  JDBC and(Java 数据库连接), 158-166
  location-dependent(位置相关), 1080
  metadata and(元数据), 164-166
  multiple-key access and(多码访问), 506-509
  on multiple relation(多关系), 66-71
  natural join and(自然连接), 71-74, 87, 113-120 (参见 join)
  nearest-neighbor(最近邻居), 1070-1071
  nested subquery(嵌套子查询), 90-98
  null values and(空值), 83-84
  object-based database and(基于对象的数据库), 945-975
  ODBC and(开放数据库互联), 166-169
  OLAP and(联机分析处理), 197-209
  Oracle and, 1171-1172
  PageRank and, 922-923
  parallel database and(并行数据库), 797-820
  persistent programming language and(持久化程序设计语言), 964-972
  point(点), 799
  programming language access and(编程语言访问), 157-173
  range(范围), 799
  read only(只读), 804
  recursive(递归), 187-192
  result diversity and(结果多样性), 932
  ResultSet object and(结果集对象), 159, 161, 164-166, 393, 397-398, 490
  retrieving result(检索结果), 161-162
  scalar subquery and(标量子查询), 97-98
  security and(安全性), 402-417
  servlet and(Java 服务器端程序), 383-391
  set operation and(集合运算), 79-83, 90-93
  on single relation(单关系), 63-66
  spatial data and(空间数据), 1070-1071
  string operation and(字符串运算), 76-79
  transaction server and(事务服务器), 775
  universal Turing machine and(通用图灵机), 14
  user requirement and(用户需求), 311-312
  view and(视图), 120-128
  XML and(可扩展标记语言), 998-1008
query cost(查询代价)
  Microsoft SQL Server and, 1237-1239
  optimization and(优化), 580-581, 590-602
  processing and(处理), 540-541, 544, 548, 555-557, 561
query evaluation engine(查询计算引擎), 22
query-evaluation plan(查询计算计划), 537-539
  choice of(选择), 598-607
  expression and(表达式), 567-572
  materialization and(物化), 567-568
  optimization and(优化), 579-616
  pipelining and(流水线), 568-572
  response time and(响应时间), 541

set operation and(集合运算), 564
viewing(视图), 582
query-execution engine(查询执行引擎), 539
query-execution plan(查询执行计划), 539
query language (查询语言), 249. 参见 specific language
accessing from a programming language(通过编程语言访问), 157-173
centralized system and(集中式系统), 770-771
domain relational calculus and(域关系演算), 245-248
expressive power of language(语言表达能力), 244, 248
formal relational(形式化关系), 217-248
nonprocedural(非过程化), 239-244
procedural(过程化), 217-239
relational algebra and(关系代数), 217-239
relational model and(关系模型), 47-48, 50
temporal(时态的), 1064
tuple relational calculus and(元组关系演算), 239-244
query optimization(查询优化), 22, 537, 539, 552-553, 562, 616
access path selection and(访问路径选择), 1174-1176
aggregation and(聚集), 597
cost analysis and(代价分析), 580-581, 590-602
distributed database and(分布式数据库), 854-855
equivalence and(等价), 582-588
estimating statistics of expression result(表达式结果集统计大小的估计), 590-598
heuristics in(启发式), 602-605
IBM DB2 and, 1211-1212
join minimization(连接最小化), 613
materialized view and(物化视图), 607-612
Microsoft SQL Server and, 1236-1241
multiquery(多查询), 614
nested subquery and(嵌套子查询), 605-607
Oracle and, 1173-1178
parallel database and(并行数据库), 814-817
parametric(参数化), 615
parallel execution and(并行执行), 1178-1179
partial search and(部分搜索), 1240
partition and(划分), 1174-1176
plan choice for(计划选择), 598-607
PostgreSQL and, 1151-1154
process structure and(进程结构), 1179
relational algebra and(关系代数), 579-590
result caching and(结果高速缓存), 1179-1180
set operation and(集合运算), 597
shared scan and(共享扫描), 614
simplification and(简化), 1237-1238
SQL Plan Management and(SQL 计划管理), 1177-1178
SQL Tuning Advisor and(SQL 调优顾问), 1176-1177
top-K, 613
transformation and(转换), 582-590, 1173-1174
update and(更新), 613-614
query processing(查询处理), 21-22, 30, 32
aggregation(聚集), 566-567
basic step of(基本步骤), 537
binding and(绑定), 1236-1237
comparison and(比较), 544-545
compilation and(编译), 1236-1237
cost analysis of(代价分析), 540-541, 544, 548, 555-557, 561
CPU speed and(CPU 速度), 540
distributed database and(分布式数据库), 854-857, 859-860
distributed heterogeneous(分布式异构), 1250-1251
duplicate elimination and(去重), 563-564
evaluation of expression(表达式计算), 567-572
executor module and(执行器模块), 1152-1153
file scan and(文件扫描), 541-544, 550, 552, 570
hashing and(散列), 557-562
IBM DB2 and, 1207-1216
identifier and(标识), 546
information retrieval and(信息检索), 915-937
join operation and(连接运算), 549-566
LINQ and, 1249
materialization and(物化), 567-568, 1212-1214
Microsoft SQL Server and, 1223-1231, 1236-1241, 1250-1251
mobile(移动), 1082
operation evaluation and(操作计算), 538-539
Oracle and, 1157-1158, 1172-1180
parsing and(解析), 537-539, 572-573, 1236-1237
pipelining and(流水线), 568-572
PostgreSQL and, 1151-1154
projection and(投影), 563-564
recursive partitioning and(递归划分), 539-540
relational algebra and(关系代数), 537-539

reordering and(重排序), 1238-1239
selection operation and(选择运算), 541-546
set operation and(集合运算), 564-565
sorting and(排序), 546-549
SQL and(结构化查询语言), 537-538
standard planner and(标准规划器), 1152
syntax and(语法), 537
transformation and(转换), 854-855
trigger and(触发器), 1153-1154
XML and(可扩展标记语言), 1259-1260
question answering(问答), 933-934
queueing system(排队系统), 1034-1035
quorum consensus protocol(法定人数同意协议), 841-842

random access(随机访问), 437
random sample(随机例子), 593
random walk model(随机游走模型), 922
range-partitioning sort(范围划分排序), 806
range-partitioning vector(范围划分向量), 801
range query(范围查询), 799
ranking(分级), 192-195
Rapid Application Development (RAD)(快速应用开发)
function library and(函数库), 396
report generator and(报表生成器), 399-400
user interface building tool and(用户界面构建工具), 396-398
Web application framework and(Web 应用构架), 398-399
raster data(光栅数据), 1069
Rational Rose, 1194
read-ahead(预读), 437
read committed(读提交)
application development and(应用程序开发), 1042
Microsoft SQL Server and, 1242
Oracle and, 1181
PostgreSQL and, 1138, 1141-1142
transaction management and(事务管理), 649, 658, 685, 701-702
read one, write all available protocol(读一个、写所有可用协议), 849-850
read one, write all protocol(读一个、写所有协议), 849
read only query(只读查询), 804
read quorum(读法定人数), 841-842
read uncommitted(读未提交), 648
read-write contention(读写竞争), 1041-1042
read/write operations(读/写操作), 653-654
real, double precision (浮点数与双精度浮点数), 59
real-time transaction system (实时事务系统), 1108-1109
recall(查全率), 903
recovery interval(恢复间隔), 1244-1245
recovery manager(恢复管理器), 22-23
recovery system(恢复系统), 186, 631, 760-761, 1083
action after crash(崩溃后动作), 736-738
algorithm for(算法), 735-738
ARIES, 750-756
atomicity and(原子性), 726-735
buffer management and(缓冲区管理), 738-743
checkpoint and(检查点), 734-735, 742-743
concurrency control and(并发控制), 729-730
data access and(数据访问), 724-726
database mirroring and(数据库镜像), 1245-1246
database modification and(数据库修改), 728-729
disk failure and(磁盘故障), 722
distributed database and(分布式数据库), 835-836
early lock release and(早期锁释放), 744-750
fail-stop assumption and(故障停止假设), 722
failure and(故障), 721-723, 743-744
force/no-force policy and(强制/非强制策略), 739-740
IBM DB2 and, 1200-1203, 1217-1218
logical undo operation and(逻辑 undo 操作), 744-750
log record and(日志记录), 726-728, 730-734, 738-739
Log Sequence Number (LSN) and(日志序列号), 750
long-duration transaction and(长事务), 1110
Microsoft SQL Server and, 1241-1246
Oracle and, 1180-1183
partition and(划分), 1169-1172
PostgreSQL and, 1145-1146
redo and(重做), 729-738
remote backup(远程备份), 723, 756-759, 850, 1095-1096
rollback and(回滚), 729-734, 736
shadow-copy scheme and(影子拷贝模式), 727
snapshot isolation and(快照隔离), 729-730
steal/no-steal policy and(抢占/非抢占策略), 740

storage and(存储), 722-726, 734-735, 743-744
successful completion and(成功完成), 723
undo and(撤销), 729-738
workflow and(工作流), 1101
write-ahead logging (WAL) rule and(先写日志规则), 739-741, 1145-1146
recovery time(恢复时间), 758
recursive partitioning(递归划分), 539-540
recursive query(递归查询), 187
iteration and(迭代), 188-190
SQL and(结构化查询语言), 190-192
transitive closure and(传递闭包), 188-190
recursive relationship set(递归联系集), 265
redo(重做)
actions after crash(崩溃后动作), 736-738
pass(遍), 754
phase(阶段), 736-738
recovery system and(恢复系统), 729-738
redundancy(冗余), 4, 261-262, 272-274
redundant array of independent disk (RAID)(独立磁盘冗余阵列), 435, 759, 1147
bit-level striping(比特级拆分), 442-444
Error-Correcting-Code (ECC) organization and(纠错码组织), 444-445
hardware issue(硬件问题), 448-449
hot swapping and(热交换), 449
levels(级别), 444-448
mirroring and(镜像), 441-442, 444
parallelism and(并行), 442-444
parity bit and(奇偶校验位), 444-446
performance reliability and(性能可靠性), 442-444
performance tuning and(性能调整), 1037-1038
recovery system and(恢复系统), 723
reliability improvement and(提高可靠性), 441-442
scrubbing and(擦洗), 448
software RAID and(软件独立磁盘冗余阵列), 448
striping data and(拆分数据), 442-444
reference(参照), 131-133, 148
referencing new row as(referencing new row as 子句), 181-182
referencing new table as(referencing new table as 子句), 183
referencing old row as(referencing old row as 子句), 182
referencing old table as(referencing old table as 子句), 183
referencing relation(参照关系), 46
referential integrity(参照完整性), 11, 46-47, 131-136, 151, 181-182, 628
referral(引用), 875
reflexivity rule(自反律), 339
region quadtree(区域四叉树), 1073
regression(回归), 902-903, 1048-1049
relational algebra(关系代数), 51-52, 248-249, 427
aggregate function(聚集函数), 235-239
assignment(赋值), 232
avg(求平均), 236
Cartesian-product(笛卡儿积), 222-226
composition of relational operation and(关系运算的组合), 219-220
count-distinct(count-distinct 运算), 236
equivalence and(等价), 582-588, 601-602
expression transformation and(表达式转换), 582-590
expressive power of language(语言表达能力), 244
formal definition of(形式化定义), 228
fundamental operation(基本运算), 217-228
generalized projection(广义投影), 235
join expression(连接表达式), 239
max(最大值函数), 236
min(最小值函数), 236
multiset(多重集), 238
natural-join(自然连接), 229-232
outer-join(外连接), 232-235
project operation(投影运算), 219
query optimization and(查询优化), 579-590
query processing and(查询处理), 537-539
rename(更名), 226-228
select operation(选择运算), 217-219
semijoin strategy and(半连接策略), 856-857
set-difference(集合差), 221-222
set-intersection(集合交), 229
SQL and(结构化查询语言), 219, 239
sum(求和), 235-236
union operation(并运算), 220-221
relational database design(关系数据库设计), 368
atomic domain and(原子域), 327-329
attribute naming(属性命名), 362-363
decomposition and(分解), 329-338, 348-360
design process and(设计过程), 361-364
feature of good(良好特征), 323-327
first normal form and(第一范式), 327-329
fourth normal form and(第四范式), 356, 358-360

functional dependency and(函数依赖), 329-348
larger schema and(更大模式), 324-325
multivalued dependency and(多值依赖), 355-360
normalization and(规范化), 361-362
relationship naming(关系命名), 362-363
second normal form and(第二范式), 336n5, 361
smaller schema and(更小模式), 325-327
temporal data modeling and(时态数据建模), 364-367
third normal form and(第三范式), 336-337
relational database(关系数据库)
access from application program and(通过应用程序访问), 14-15
Data-Definition Language and(数据定义语言), 14
Data-Manipulation Language (DML) and(数据操纵语言), 13-14
storage and(存储), 1010-1014
table and(表), 12-13
relational model(关系模型), 9
disadvantage of(缺点), 30
domain and(域), 42
key and(码), 45-46
natural join and(自然连接), 49-50
operation and(运算), 48-52
query language and(查询语言), 47-48, 50
referencing relation and(参照关系), 46
schema for(模式), 42-47, 302-304, 1012
structure of(结构), 39-42
table for(表), 39-44, 49-51, 202-205
tuple and(元组), 40-42, 49-50
relation instance(关系实例), 42-45, 264
relationship set(联系集)
alternative notation for(替代记号), 304-310
atomic domain and(原子域), 327-329
attribute placement and(属性布局), 294-295
binary vs. n-ary(二元与 n 元), 292-294
descriptive attribute(描述性属性), 267
design issue and(设计问题), 291-295
entity-relationship diagram and(实体联系图), 278-279
entity-relationship (E-R) model and(实体-联系模型), 264-267, 286-290, 296-297
entity set and(实体集), 291-292
naming of(命名), 362-363
nonbinary(非二元的), 278-279
recursive(递归), 265
redundancy and(冗余), 288
representation of(表示), 286-290
schema combination and(模式组合), 288-290
superclass-subclass(超类-子类), 296-297
Unified Modeling Language (UML) and(统一建模语言), 308-310
Relative Distinguished Name(RDN)(相对可区别名称), 872
relevance(相关性)
adjacency test and(相邻测试), 922-923
hub and(链接中心), 924
PageRank and, 922-923, 925
popularity ranking and(流行度排名), 920-922
ranking using TF-IDF(使用 TF-IDF 排名), 917-920, 925
search engine spamming and(搜索引擎作弊), 924-925
similarity-based retrieval and(基于相似性的检索), 919-920
TF-IDF approach and(TF-IDF 方法), 917-925, 928-929
using hyperlink and(使用超链接), 3421
Web crawler and(网络爬虫), 930-931
relevance feedback(相关反馈), 919-920
remote backup system(远程备份系统), 723, 756-759, 850, 1095-1096
Remote-Procedure-Call (RPC) mechanism(远程过程调用机制), 1096
rename operation(更名运算), 75-76, 226-228
repeat(重复), 176
repeatable read(可重复读), 649
repeat loop(重复循环), 188, 341, 343, 490
replication(复制)
cloud computing and(云计算), 866-868
distributed database and(分布式数据库), 843-844
Microsoft SQL Server and, 1251-1253
system architecture and(系统体系结构), 785, 826, 829
report generator(报表生成器), 399-400
Representation State Transfer (REST)(表示状态转移), 395
request forgery(请求伪造), 403-405
request operation(请求操作)
deadlock handling and(死锁处理), 675-679
lock and(锁), 662-671, 675-680, 709
lookup and(查找), 706
multiple granularity and(多粒度), 679-680
multiversion scheme and(多版本机制), 691

snapshot isolation and(快照隔离), 693
timestamp and(时间戳), 682
resource manager(资源管理器), 1095
response time(响应时间)
application design and(应用程序设计), 400, 1037, 1046
concurrency control and(并发控制), 688
E-R model and(E-R 模型), 311
Microsoft SQL Server and, 1261
Oracle and, 1176-1177, 1190
query evaluation plan and(查询计算计划), 541
query processing and(查询处理), 541
storage and(存储), 444, 1106, 1109-1110
transaction and(事务), 636
system architecture and(系统体系结构), 778, 798, 800, 802
restriction(约束), 149-150, 347
ResultSet object(结果集对象), 159, 161, 164-166, 393, 397-398, 490
revoke(撤回), 145, 149
right outer join(右外连接), 117-120, 233-235, 565-566
Rijndael algorithm(Rijndael 算法), 412-413
robustness(健壮性), 847
role(角色), 264-265
authorization and(授权), 145-146
entity-relationship diagram(实体联系图), 278
Unified Modeling Language (UML) and(统一建模语言), 308-310
rollback(回滚), 173
ARIES and, 754-755
cascading(级联), 667
concurrency control and(并发控制), 667, 670, 674-679, 685, 689, 691, 709
IBM DB2 and, 1218
logical operation and(逻辑运算), 746-749
PostgreSQL and, 1142-1144
recovery system and(恢复系统), 729-734, 736
remote backup system and(远程备份系统), 758-759
transaction and(事务), 736
timestamp and(时间戳), 685-686
undo and, 729-734
rollback work(rollback work 语句), 127
rollup(上卷), 201, 206-210, 1221-1222
RosettaNet, 1055
row trigger(行触发器), 1161-1162
R-tree(R 树), 1073-1076
Ruby on Rails, 367, 399
runstats(runstats 命令), 593

SAS (Serial attached SCSI)(串行附属 SCSI), 434
Sarbanes-Oxley Act(Sarbanes-Oxley 法案), 1248
SATA (serial ATA)(串行 ATA), 434, 436
savepoint(保存点), 756
scalar subquery(标量子查询), 97-98
scaleup(扩展比), 778-780
scheduling(调度)
Microsoft SQL Server and, 1254-1255
PostgreSQL and, 1127
query optimization and(查询优化), 814-815
storage and(存储), 437
transaction and(事务), 641, 1099-1100, 1108
schema definition(模式定义), 28
schema diagram(模式图), 46-47
schema(模式), 8
alternative notation for modeling data(数据建模的备代记号), 304-310
authorization on(授权), 147-148
basic SQL query structure and(基本 SQL 查询结构), 63-74
canonical cover and(正则覆盖), 342-345
catalog and(目录), 142-143
combination of(结合), 288-290
concurrency control and(并发控制), 661-710(参见 concurrency control)
Data-Definition Language (DDL) and(数据定义语言), 58, 60-63
data mining(数据挖掘), 893-910
data warehouse(数据仓库), 889-893
entity-relationship (E-R) model and(实体-联系模型), 262-313
functional dependency and(函数依赖), 329-348
generalization and(概化), 297-304
large(庞大), 324-325
locks and(锁), 661-686
performance tuning of(性能调整), 1038-1039
physical-organization modification and(物理组织修改), 28
recovery system and(恢复系统), 721-761
reduction to relational(转换为关系模式), 283-290
redundancy of(冗余), 288
relational algebra and(关系代数), 217-239

relational database design and(关系数据库设计), 323-368
relational model and(关系模型), 42-47
relationship set and(联系集), 286-288
shadow-copy(影子拷贝), 727
smaller(更小), 325-327
strong entity set and(强实体集), 283-285
timestamp and(时间戳), 682-686
tuple relational calculus(元组关系演算), 239-244
version-numbering(版本编号), 1083-1084
weak entity set(弱实体集) and, 285-286
XML document(可扩展标记语言文档), 990-998
scripting language(脚本语言), 389
scrubbing(擦洗), 448
search engine spamming(搜索引擎作弊), 924-925
search key(搜索码)
hashing and(散列), 509-519, 524
indexing and(索引), 476-509, 524, 529
nonunique(不唯一), 497-499
storage and(存储), 457-459
uniquifier and(唯一化), 498-499
secondary site(辅助站点), 756
second normal form(第二范式), 361
Secure Electronic Transaction (SET) protocol(安全电子交易协议), 1105
security(安全性), 5, 147
abstraction and(抽象), 6-8, 10
application design and(应用程序设计), 402-417
audit trail and(审计追踪), 409-410
authentication and(鉴定), 405-407
authorization and(授权), 11, 21, 407-409
concurrency control and(并发控制), 661-710(参见 concurrency control)
cross site scripting and(跨站脚本), 403-405
dictionary attack and(字典攻击), 414
encryption and(加密), 411-417, 1165-1166
end-user information and(最终用户信息), 407-408
GET method and(GET 方法), 405
integrity manager and(完整性管理器), 21
isolation and(隔离性), 628, 635-640, 646-653
key and(钥), 45-46
lock and(锁), 661-686(参见 lock)
long-duration transaction and(长事务), 1109-1115
man-in-the-middle attack and(中间人攻击), 406
Microsoft SQL Server and, 1247-1248
observable external write and(可见的外部写), 634-635
Oracle and, 1165-1166
password and(密码), 142, 160, 168, 170, 376, 382, 385, 393, 405-407, 415, 463-464, 871
person-in-the-middle attack and(中间人攻击), 1105
physical data independence and(物理数据独立性), 6
privacy and(隐私), 402, 410-411, 418, 828, 869-870, 1104
remote backup system and(远程备份系统), 756-759
request forgery and(请求伪造), 403-405
single sign-on system and(单点登录系统), 406-407
SQL injection and(SQL 注入), 402-403
unique identification and(唯一标识), 410-411
virtual private database and(虚拟专用数据库), 1166
Security Assertion Markup Language (SAML)(安全声明标记语言), 407
seek time(寻道时间), 433, 435-439, 450-451, 540, 555
select(select 子句), 363
aggregate function and(聚集函数), 84-90
attribute specification(属性说明), 77
basic SQL query and(基本 SQL 查询), 63-74
on multiple relation(多关系), 66-71
natural join and(自然连接), 71-74
null value and(空值), 83-84
privilege and(权限), 143-145, 148
ranking and(分级), 194
rename operation and(更名运算), 74-75
set membership and(集合成员), 90-91
set operation and(集合运算), 79-83
on single relation(单关系), 63-65, 63-66
string operation and(字符串运算), 76-79
select all(select all 子句), 65
select distinct(select distinct 子句), 64-65, 84-85, 91, 125
select-from-where(select-from-where 语句)
delete and(删除), 98-100
function/procedure writing and(函数/过程写), 174-180
inheritance and(继承), 949-956
insert and(插入), 100-101

join expression and (连接表达式), 71-74, 87, 113-120
natural join and(自然连接), 71-74, 87, 113-120
nested subquery and(嵌套子查询), 90-98
transaction and(事务), 651-654
type handling and(类型处理), 949-963
update and(更新), 101-103
view and(视图), 120-128
selection(选择)
comparison and(比较), 544-545
complex(复杂的), 545-546
conjunctive(合取), 545-546
disjunctive(析取), 545-546
equivalence and(等价), 582-588
file scan and(文件扫描), 541-544, 550, 552, 570
identifier and(标识), 546
index and(索引), 541-544
intraoperation parallelism and(操作内并行), 811
linear search and(线性搜索), 541-542
relational algebra and(关系代数), 217-219
SQL and(结构化查询语言),
view maintenance and(视图维护), 609-610
Semantic Web(语义网), 927
semistructured data model(半结构化数据模型), 9, 27
sensitivity(灵敏度), 903
Sequel, 57
sequence association(序列关联), 906-907
sequence counter(序号计数器), 1043
sequential-access storage (顺序访问存储器), 431, 436
sequential file(顺序文件), 459
sequential scan(顺序扫描), 1153
serializability(串行化)
blind write and(盲写), 687
concurrency control and(并发控制), 662, 666-667, 671, 673, 681-690, 693-697, 701-704, 708
conflict(冲突), 641-643
distributed database and(分布式数据库), 860-861
isolation and(隔离性), 648-653
Oracle and, 1181-1182
order of(顺序), 644-646
performance tuning and(性能调整), 1042
PostgreSQL and, 1142-1143
precedence graph and(优先图), 644
predicate read and(谓词读), 701
in the real world(在现实世界中), 650
snapshot isolation and(快照隔离), 693-697
topological sorting and(拓扑排序), 644-646
transaction and(事务), 640-646, 648, 650-653
view, 687
serializable schedule(可串行化调度), 640
Server Programming Interface (SPI)(服务器编程接口), 1136
server-side scripting(服务器端脚本), 386-388
server system(服务器系统)
categorization of(分类), 772-773
client-server(客户-服务器), 771-772
cloud-based(基于云的), 777
data server(数据服务器), 773, 775-777
transaction-server(事务服务器), 773-775
servlet(java 服务器端程序)
client-side scripting and(客户端脚本), 389-391
example of(例子), 383-384
life cycle and(生命周期), 385-386
server-side scripting and(服务器端脚本), 386-388
session and(会话), 384-385
support and(支持), 385-386
set clause(set 子句), 103
set default(set default 语句), 133
set difference(集合差), 50, 221-222, 585
set-intersection(集合交), 2229
set null(set null 语句), 133
set operation(集合运算), 79, 83
IBM DB2 and, 1209-1210
intersect(交), 50, 81-82, 585, 960
nested subquery and(嵌套子查询), 90-93
query optimization and(查询优化), 597
query processing and(查询处理), 564-565
set comparison and(集合比较), 91-93
union(并), 80-81, 220-221, 339, 585
set role(set role 语句), 150
set transaction isolation level serializable (set transaction isolation level serializable 语句), 649
shadow-copy scheme(影子拷贝模式), 727
shadowing(影子), 441-442
shadow-paging(影子分页), 727
Shared and Intention-eXclusive (SIX) mode(共享意向排他模式), 680
shared-disk architecture (共享磁盘体系结构), 781, 783, 789

shared-memory architecture(共享内存体系结构), 781-783
shared-mode lock(共享型锁), 661
shared-nothing architecture(无共享体系结构), 781, 783-784
shared scan(共享扫描), 614
Sherpa/PNUTS, 866-867
shredding(分解), 1013, 1258-1259
similarity-based retrieval(基于相似性的检索), 919-920, 1079
Simple API for XML (SAX)(XML 的简单应用程序接口), 1009
Simple Object Access Protocol (SOAP)(简单对象访问协议), 1017-1018, 1056, 1249-1250
single lock-manager(单锁管理器), 839-840
single-server model(单服务器模型), 1092-1093
single-valued attribute(单值属性), 267-268
site reintegration(站点重建), 850
skew(偏斜), 512
  attribute-value(属性值), 800-801
  parallel database and(并行数据库), 800-801, 805-808, 812, 814, 819
  parallel system and(并行系统), 780
  partitioning and(划分), 560, 800-801
slicing(切片), 201
Small-Computer-System Interconnect (SCSI)(小计算机系统互连接), 434
snapshot isolation(快照隔离), 652-653, 704, 1042
  Microsoft SQL Server and, 1244
  recovery system and(恢复系统), 729-730
  serializability and(可串行化), 693-697
  validation and(有效性检查), 692-693
snapshot replication(快照复制), 1252-1253
snapshot(快照)
  DML command and(数据操纵语言命令), 1138-1139
  Microsoft SQL Server and, 1242
  MultiVersion Concurrency Control (MVCC) and(多版本并发控制), 1137-1146
  PostgreSQL and, 1137-1146
  read committed(读提交), 1242
software RAID(软件独立磁盘冗余阵列), 448
Solaris, 1193
solid-state drive(固态驱动器), 430
some(some 子句), 90, 92, 92n8
sorting(排序), 546
  cost analysis of(代价分析), 548-549
  duplicate elimination and(去重), 563-564
  external sort-merge algorithm and(外排序-归并算法), 547-549
  parallel external sort-merge and(并行外排序-归并), 806
  PostgreSQL and, 1153
  range-partitioning(范围划分), 805
  topological(拓扑的), 644-646
  XML and(可扩展标记语言), 1106
sort-merge-join(排序-归并-连接), 553
space overhead(空间开销), 476, 479, 486, 522
spatial data(空间数据)
  computer-aided-design data and(计算机辅助设计数据), 1061, 1064-1068
  geographic data and(地理数据), 1061, 1064-1066
  indexing of(索引), 1071-1076
  query and(查询), 1070-1071
  representation of geometric information and(几何信息的表示), 1065-1066
  topographical information and(拓扑信息), 1070
  triangulation and(三角剖分), 1065
  vector data and(向量数据), 1069
specialization(特化)
  entity-relationship (E-R) model and(实体-联系模型), 295-296
  partial(部分), 300
  single entity set and(单实体集), 298
  total(全部), 300
specialty database(专业数据库), 943
  object-based database and(基于对象的数据库), 945-975
  XML and(可扩展标记语言), 981-1020
specification of functional requirement(功能需求说明), 16, 260
specificity(特异性), 903
speedup(加速比), 778-780
spider trap(爬虫陷阱), 930
SQL (Structured Query Language)(结构化查询语言), 10, 13-14, 57, 151, 210, 582
  accessing from a programming language(通过编程语言访问), 157-163
  advanced(高级), 157-210
  aggregate function(聚集函数), 84-90, 192-197
  application-level authorization and(应用级授权), 407-409
  application program and(应用程序), 14-15
  array type and(数组类型), 956-961

authorization and(授权), 58, 143-150
basic type and(基本类型), 59-60
blob and(二进制大对象), 138, 166, 457, 502, 1013, 1198-1199, 1259
bulk load and(批量加载), 1031-1033
catalog(目录), 142-143
clob and(字符大对象数据类型), 138, 166, 457, 502, 1010-1013, 1196-1199
CLR hosting and(CLR 宿主), 1254-1256
create table(create table 语句), 60-63, 141-142
database modification and(数据库修改), 98-103
Data-Definition Language (DDL) and(数据定义语言), 57-63, 104
Data-Manipulation Language (DML) and(数据操纵语言), 57-58, 104
data mining and(数据挖掘), 26
date/time type in(日期/时间类型), 136-137
decision-support system and(决策支持系统), 887-889
default value and(默认值), 137
delete and(删除), 98-100
dumping and(转储), 743-744
dynamic(动态), 58, 158
embedded(嵌入的), 58, 158, 169-173, 773
Entity(实体), 395
environment(环境), 43
function writing and(函数写), 173-180
IBM DB2 and, 1195-1200, 1210
index creation and(索引创建), 137-138, 528-529
inheritance and(继承), 949-956
injection and(注入), 402-403
insert and(插入), 100-101
integrity constraint and(完整性约束), 58, 128-136
intermediate(中级), 113-151
isolation level and(隔离性级别), 648-653
JDBC and(Java 数据库连接), 158-166
join expression and(连接表达式), 71-120 (参见 joins)
lack of fine-grained authorization and(缺少细粒度授权), 408-409
large-type object(大类型对象), 138
Management of External Data (MED) and(外部数据管理), 1077
Microsoft SQL Server and, 1223-1267
multiset type and(多重集合类型), 956-961
MySQL and, 31, 76, 111, 160n3, 1123, 1155
nested subquery and(嵌套子查询), 90-98
nonstandard syntax and(非标准语法), 178
null value and(空值), 83-84
object-based database and(基于对象的数据库), 945-975
ODBC and(开放数据库互联), 166-169
OLAP and(联机分析处理), 197-209
Oracle variation and(Oracle 变种), 1158-1162
overview of(概述), 57-58
persistent programming language and(持久化程序设计语言), 964-972
PostgreSQL and, 31 (参见 PostgreSQL)
prepared statement and(预备语句), 162-164
procedure writing and(过程写), 173-180
query processing and(查询处理), 537-538 (参见 query processing)
Rapid Application Development (RAD) and(快速应用开发), 397
relational algebra and(关系代数), 219, 239
rename operation and(更名运算), 74-80
report generator and(报表生成器), 399-400
ResultSet object and(结果集对象), 159, 161, 164-166, 393, 397-398, 490
revoking of privilege and(权限撤回), 149-150
role and(角色), 145-146
schema and(模式), 47, 58-63, 141-143, 147-148
security and(安全性), 402-403
select clause and(select 子句), 77
set operation and(集合运算), 79-83
as standard relational database language(标准关系数据库语言), 57
standard for(标准), 1052-1053
string operation and(字符串运算), 76-77
System R and, 30, 57
time specification in(时间描述), 1063-1064
transaction and(事务), 58, 127-128, 773 (参见 transaction)
transfer of privilege and(权限转移), 148-149
trigger and(触发器), 180-187
tuple and(元组), 77-78 (参见 tuple)
under privilege and, 956
update and(更新), 101-103
user-defined type(用户自定义类型), 138-141
view and(视图), 58, 120-128, 146-147
where clause predicate(where 子句谓词), 78-79
SQLLoader(SQL 加载器), 1032, 1189
SQL Access Group, 1053

SQL/DS, 30
SQL environment(SQL 环境), 143
SQLJ, 172
SQL Plan Management (SQL 计划管理), 1177-1178
SQL Profiler(SQL 跟踪器), 1225-1227
SQL Security Invoker(SQL Security Invoker 子句), 147
SQL Server Analysis Service (SSAS)(SQL Server 分析服务), 1264, 1266-1267
SQL Server Broker(SQL Server 代理), 1261-1263
SQL Server Integration Service (SSIS)(SQL Server 集成服务), 1263-1266
SQL Server Management Studio, 1223-1224, 1227-1228
SQL Server Reporting Service (SSRS)(SQL Server 报表服务), 1264, 1267
sqlstate, 179
SQL Transparent Data Encryption(SQL 透明数据加密), 1248
SQL Tuning Advisor(SQL 调优顾问), 1176-1177
SQL/XML standard(SQL/XML 标准), 1014-1015
Standard Generalized Markup Language (SGML)(标准通用标记语言), 981
standard(标准)
  ANSI(美国国家标准化组织), 57, 1051
  anticipatory(预见), 1051
  Call Level Interface (CLI)(调用层接口), 1053
  database connectivity(数据库连接), 1053-1054
  data pump export/import and(数据泵导出/导入), 1189
  DBTG CODASYL(网状数据库标准), 1052
  ISO(国际标准化组织), 57, 871, 1051
  ODBC(开放数据库互联), 1053-1055
  reactionary(反应), 1051
  SQL(结构化查询语言), 1052-1053
  Wi-Max, 1081
  XML(可扩展标记语言), 1055-1056
  X/Open XA, 1053-1054
Starburst, 1193
start-up cost(启动代价), 780
starvation(饿死), 679
Statement object(语句对象), 161-164
statement trigger(语句触发器), 1161-1162
state transition(状态转换), 1134
state value(状态值), 1134
statistics(统计)
  catalog information and(目录信息), 590-592
  computing(计算), 593
  join size estimation and(连接大小估计), 595-596
  maintaining(维护), 593
  number of distinct value and(不同值的数目), 597-598
  query optimization and(查询优化), 590-598
  random sample and(随机例子), 593
  selection size estimation and(选择大小估计), 592-595
steal policy(抢占策略), 740
step(步骤), 1096
stop word(停用词), 918
storage(存储), 427
  archival(归档), 431
  atomicity and(原子性), 632-633
  authorization and(授权), 21
  Automatic Storage Manager and(自动存储管理器), 1186-1187
  backup (备份), 431, 723, 756-759, 850, 1095-1096
  bit-level striping(比特级条分), 442-444
  buffer manager and(缓冲区管理器), 21 (参见 buffer)
  byte amount and(字节数), 20
  checkpoint and(检查点), 734-735, 742-743
  clob value and(字符大对象值), 1010-1011
  cloud-based(基于云的), 777, 862-863
  column-oriented(面向列的), 892-893
  content dump and(内容转储), 743
  cost per bit(每比特代价), 431
  crash and(崩溃), 467-468 (参见 crash)
  data access and(数据访问), 724-726
  data-dictionary(数据字典), 462-464
  data mining and(数据挖掘), 25-26, 893-910
  data-transfer rate and(数据传输率), 435-436
  data warehouse and(数据仓库), 888
  decision-storage system and(决策存储系统), 887-889
  direct-access(直接访问), 431
  distributed database and(分布式数据库), 826-830
  distributed system and(分布式系统), 784-788
  dumping and(转储), 743-744
  durability and(持久性), 632-633
  Error-Correcting-Code (ECC) organization and(纠错码组织), 444-445
  Exadata and, 1187-1188

file manager and(文件管理器), 21
file organization and(文件组织), 451-462
flash(闪存), 403, 430, 439-441, 506
flat file and(平面文件), 1009-1010
force output and(强制输出), 725-726
fragmentation and(分片), 826-829
hard disk and(硬盘), 29-30
IBM DB2 and, 1200-1203
index and(索引), 21 (参见 index)
information retrieval and(信息检索), 915-937
integrity manager and(完整性管理器), 21
jukebox(自动光盘机), 431
magnetic disk(磁盘), 430, 432-439
main memory and(主存), 429-430
Microsoft SQL Server and, 1233-1236
mirroring and(镜像), 441-442, 1245-1246
native(本地), 1013-1014
nonrelational data(非关系数据), 1009-1010
nonvolatile(非易失性), 432, 632, 722, 724-726, 743-744
optical(光盘), 430-431, 449-450
Oracle and, 1162-1172, 1186-1188
parallel system and(并行系统), 777-784
persistent programming language and(持久化程序设计语言), 967-968
physical media for(物理介质), 429-432
PostgreSQL and, 1146-1151
publishing/shredding data and(发布/分解数据), 1013, 1258-1259
punched card and(穿孔卡片), 29
query processor and(查询处理器), 21-22
recovery system and(恢复系统), 722-726 (参见 recovery system)
Redundant Array of Independent Disk (RAID) (独立磁盘冗余阵列), 435, 441-449
relational database and(关系数据库), 1010-1014
remote backup system and(远程备份系统), 723, 756-759, 850, 1095-1096
replication and(复制), 826, 829
scrubbing and(擦洗), 448
seek time and(寻道时间), 433, 435-439, 450-451, 540, 555
segment and(段), 1163
sequential-access(顺序访问), 431, 436
solid-state drive and(固态驱动器), 430
stable(稳定的), 632, 722-724
striping data and(拆分数据), 442-444
tape(磁带), 431, 450-451
tertiary(第三级), 431, 449-451
transaction manager and, 21 (参见 transaction)
transparency and(透明性), 829-830
volatile(易失性), 431, 632, 722
wallet and(钱包), 415
XML and(可扩展标记语言), 1009-1016
Storage Area Network (SAN) (存储区域网), 434-435, 789
storage manager(存储管理器), 20-21
string operation(字符串运算)
aggregate(聚集), 84
attribute specification(属性说明), 77
escape(转义), 77
JDBC and(Java 数据库连接), 158-166
like(like 运算符), 76-77
lower(lower 函数), 76
query result retrieval and(查询结果检索), 161-162
similar to(类似), 77
trim(trim 函数), 76
tuple display order(元组显示次序), 77-78
upper function(upper 函数), 76
where predicate(where 谓词), 78-79
striping data(拆分数据), 442-444
structured type(结构化类型), 138-141, 949-952
stylesheet(样式表), 380
sublinear speedup(亚线性加速比), 778-780
submultiset(submultiset 谓词), 960
suffix(后缀), 874
sum (求和), 84, 123, 207, 235-236, 566-567, 1134
superclass-subclass relationship(超类-子类联系), 296-297
superkey(超码), 45-46, 271-272, 330-333
superuser(超级用户), 143
Support Vector Machine (SVM) (支持向量机), 900-901, 1191
swap space(交换区), 742
Swing, 399
Sybase, 1223
Symmetric MultiProcessor (SMP) (对称多处理器), 1193
synonym(同义词), 925-927
sysaux(辅助系统表空间), 1172-1173
system architecture(系统体系结构). 参见 architecture
system catalog(系统目录), 462-464, 1132
System Change Number (SCN) (系统改变号),

1180-1181
system error(系统错误), 721
System R, 30, 57, 1193

table inheritance(表继承), 954-956
table(表), 12-13
  filtering and(过滤), 1187
  IBM DB2 and, 1200-1203
  materialized(物化的), 1212-1214
  Microsoft SQL Server and, 1230, 1234
  .NET Common Language Runtime (CLR) and(.NET 公共语言运行库), 1257-1258
  Oracle and, 1163-1166, 1187, 1189
  partition and(划分), 1169-1172
  relational model and(关系模型), 39-44, 49-51
  SQL Server Broker and(SQL Server 代理), 1262
tablespace(表空间), 1146, 1172-1173
tag library(标签库), 388
tag(标签)
  application design and(应用程序设计), 378-379, 388, 404
  information retrieval and(信息检索), 916
  XML and(可扩展标记语言), 982-986, 989, 994, 999, 1004, 1019
tape storage(磁带存储器), 431, 450-451
Tapestry, 399
task flow(任务流). 参见 workflow
Tcl, 180, 1123-1125, 1136
temporal data(时态数据), 1061
  interval and(间隔), 1063-1064
  query language and(查询语言), 1064
  relational database and(关系数据库), 364-367
  time in database and(数据库中的时间), 1062-1064
  timestamp and(时间戳), 1063-1064
  transaction time and(事务时间), 1062
temporal relation(时态关系), 1062-1063
Teradata Purpose-Built Platform Family | Teradata Purpose-Build 平台系列机), 806
Term Frequency (TF)(词频), 918
termination state(终止状态), 1099
tertiary storage(三级存储), 431, 449-451
TF-IDF approach(TF-IDF 方法), 928-929
theta join(theta 连接), 584-585
third normal form (3NF)(第三范式)
  decomposition algorithm and(分解算法), 352-355
  relational database and(关系数据库), 336-337, 352-355
Thomas' write rule(Thomas 写规则), 685-686
thread pooling(线程池), 1246
Three-Phase Commit (3PC) protocol(三阶段提交协议), 826
three-tier architecture(三层体系结构), 25
throughput(吞吐量)
  application development and(应用程序开发), 1037, 1045-1046
  defined(定义的), 311
  harmonic mean of(调和平均数), 1046
  improved(改进的), 635-636, 655
  log record and(日志记录), 1106
  main memory and(主存), 1116
  Microsoft SQL Server and, 1255
  Oracle and, 1159, 1184
  parallel system and(并行系统), 778
  performance and(性能), 1110
  range partitioning and(范围划分), 800
  storage and(存储), 444, 468
  system architecture and(系统体系结构), 771, 778, 800, 802, 819
  transaction and(事务), 635-636, 655
timestamp(时间戳), 136-167
  concurrency control and(并发控制), 682-686, 703
  distributed database and(分布式数据库), 842-843
  logical counter and(逻辑计数器), 682
  long-duration transaction and(长事务), 1110
  multiversion scheme and(多版本机制), 690-691
  ordering scheme and(排序模式), 682-685
  rollback and(回滚), 685-686
  temporal data and(时态数据), 1063-1064
  Thomas' write rule and(Thomas 写规则), 685-686
  transaction and(事务), 651-652
  with time zone(带时区), 1063
time to completion(完成时间), 1045
time with time zone(带时区时间), 1063
timezone(时区), 136-137, 1063
Tomcat, 386
top-down design(自顶向下设计), 297
top-K optimization(top-K 优化), 613
topographic information(地形信息), 1070
topological sorting(拓扑排序), 644-646
training instance(训练实例), 895
transactional replication(事务复制), 1252-1253
transaction control(事务控制), 58
transaction coordinator(事务协调器), 830-831,

834-835, 850-852
transaction manager (事务管理器), 21, 23, 830-831
transaction-processing monitor (事务处理监控器), 1091
  application coordination using (应用协调), 1095-1096
  architecture of(体系结构), 1092-1095
  durable queue and(持久队列), 1094
  many-server, many-router model and(多服务器多路由器模型), 1094
  many-server, single-router model and(多服务器单路由器模型), 1093
  multitasking and(多任务调度), 1092-1095
  presentation facility and(表示设施), 1094-1095
  single-server model and (单服务器模型), 1092-1093
  switching and(切换), 1092
Transaction Processing Performance Council (TPC) (事务处理性能委员会), 1046-1048
transaction(事务), 32, 625, 655-656, 1116
  aborted(中止), 633-634, 647
  action after crash(崩溃后动作), 736-738
  active(活动), 633
  advanced processing of(高级处理), 1091-1116
  association rule and(关联规则), 904-907
  atomicity and (原子性), 22-23, 628, 633-635, 646-648 (参见 atomicity)
  availability and(可用性), 847-853
  begin/end operation and(begin/end 操作), 627
  cascadeless schedule and(无级联的调度), 647-648
  check constraint and(check 约束), 628
  cloud computing and(云计算), 866-868
  commit protocol and(提交协议), 832-838
  committed (提交), 127, 633-635, 639, 647, 692-693, 730, 758, 832-838, 1107, 1218
  compensating(补偿), 633, 1113-1114
  concept of(概念), 627-629
  concurrency control and (并发控制), 661-710, 1241-1246 (参见 concurrency control)
  consistency and(一致性), 22, 627-631, 635-636, 640, 648-650, 655 (参见 consistency)
  crash and(崩溃), 628
  data mining and(数据挖掘), 893-910
  decision-storage system and (决策存储系统), 887-889
  defined(定义的), 22, 627
  distributed database and(分布式数据库), 830-832
  durability and(持久性), 22-23, 628, 633-635 (参见 durability)
  E-commerce and(电子商务), 1102-1105
  failure of(故障), 633, 721-722
  force/no-force policy and(强制/非强制策略), 739-740
  global(全局), 784, 830, 860-861
  integrity constraint violation and(违反完整性约束), 133-134
  isolation and(隔离性), 628, 635-640, 646-653 (参见 isolation)
  killed(杀死的), 634
  local(局部), 784, 830, 860-861
  lock and(锁), 661-669, 661-686 (参见 lock)
  log record and(日志记录), 726-728, 730-734
  long-duration(长周期), 1109-1115
  main-memory database and (主存数据库), 1105-1108
  multidatabase and(多数据库), 860-861
  multilevel(多级别), 1112-1113
  multitasking and(多任务调度), 1092-1095
  MultiVersion Concurrency Control (MVCC) and(多版本并发控制), 1137-1146
  multiversion scheme and(多版本机制), 689-692
  object-based database and(基于对象的数据库), 945-975
  observable external write and(可见的外部写), 634-635
  parallel database and(并行数据库), 797-820
  performance tuning and(性能调整), 1041-1044
  persistent messaging and(持久消息), 836-837
  persistent programming language and(持久化程序设计语言), 970
  person-in-the-middle attack and (中间人攻击), 1105
  PostgreSQL and, 1137-1146
  read/write operation and(读/写操作), 653-654
  real-time system and(实时系统), 1108-1109
  recoverable schedule and(可恢复调度), 647
  recovery manager and(恢复管理器), 22-23
  recovery system and(恢复系统), 631, 633 (参见 recovery system)
  remote backup system and (远程备份系统), 756-759
  restart of(重启), 634
  rollback and(回滚), 127, 736, 746-749, 754-755

serializability and(可串行化), 640-653
shadow-copy scheme and(影子拷贝模式), 727
simple model for(简单模型), 629-631
SQL Server Broker and (SQL Server 代理), 1261-1263
as SQL statement(SQL 语句), 653-654
starved(饿死的), 666
states of(状态), 633-635
steal/no-steal policy and(抢占/非抢占策略), 740
storage structure and(存储结构), 632-633
timestamp and(时间戳), 682-686
Two-Phase Commit(2PC) protocol and(两阶段提交协议), 786-788
uncommitted(未提交的), 648
as unit of program(程序单元), 627
validation and(有效性检查), 686-689
wait-for graph and(等待图), 676-678
workflow and(工作流), 836-838, 1096-1102
Write-Ahead Logging (WAL) rule and(先写日志规则), 739-740, 739-741
transaction scaleup(事务扩展比), 779
transactions-consistent snapshot(事务一致快照), 843-844
transaction-server system(事务服务器系统), 773-775
transaction per second (TPS)(每秒事务数), 1046-1047
transaction time(事务时间), 365n8, 1062
TransactSQL, 173
transfer of control(控制转移), 757
transfer of prestige(威望度转移), 921-922
transformation(转换)
equivalence rule and(等价规则), 583-586
example of(例子), 586-588
join ordering and(连接次序), 588-589
query optimization and(查询优化), 582-590
relational algebra and(关系代数), 582-590
XML and(可扩展标记语言), 998-1008
transition table(过渡表), 183-184
transition variable(过渡变量), 181
transitive closure(传递闭包), 188-190
transitivity rule(传递律), 339-340
transparency(透明性), 829-830, 854-855
tree(树), 1086
B, 504-506, 530, 1039, 1064, 1071-1072, 1076, 1086, 1135, 1148-1150, 1159, 1164-1169, 1173, 1205
B', 1234-1235 (参见 B'-tree)
decision-tree classifier and (决策树分类器), 895-900
Directory Information (DI)(目录信息), 872-875
distributed directory(分布式目录), 874-875
Generalized Search Tree (GiST) and(通用搜索树), 1148-1149
Index-Organized Table (IOT) and(索引组织表), 1164-1165
k-d, 1071-1072
multiple granularity and(多粒度), 679-682
Oracle and, 1164-1165, 1191
overfitting and(过度适应), 899-900
PostgreSQL and, 1148-1149
quadratic split and(二次方分裂), 1075-1076
quadtree(四叉树), 1069, 1072-1073
query optimization and(查询优化), 814-815 (参见 query optimization)
R, 1073-1076
scheduling and(调度), 814-815
spatial data support and (支持空间数据), 1064-1076
XML(可扩展标记语言), 998, 1011
trigger(触发器)
alter(修改), 185
disable(使无效), 185
drop(删除), 185
IBM DB2 and, 1210
Microsoft SQL Server and, 1232-1233
need for(需求), 180-181
nonstandard syntax and(非标准语法), 184
Oracle and, 1161-1162
PostgreSQL and, 1153-1154
recovery and(恢复), 186
in SQL(用 SQL), 181-187
transition table and(过渡表), 183-184
when not to use(当不使用时), 186-187
true negative(真舍弃), 903
true predicate(真谓词), 67
true relation(真关系), 90, 93
tuple ID(元组标识符), 1147-1148
tuple relational calculus(元组关系演算), 239, 249
example query(举例查询), 240-242
expressive power of language(语言表达能力), 244
formal definition(形式化定义), 243
safety of expression(表达式安全性), 244
tuple(元组), 40-42

aggregate function and(聚集函数), 84-90
Cartesian product and(笛卡儿积), 80
delete and(删除), 98-100
domain relational calculus and(域关系演算), 245-248
duplicate(重复的), 94-95
eager generation of(积极产生), 569-570
insert and(插入), 100-101
join and(连接), 550-553(参见 join)
lazy generation of(消极产生), 570-571
ordering display of(排序显示), 77-78
parallel database and(并行数据库), 797-820
pipelining and(流水线), 568-572
PostgreSQL and, 1137-1146
query structure and(查询结构), 68
query optimization and(查询优化), 579-616
query processing and(查询处理), 537-573
ranking and(分级), 192-195
relational algebra and(关系代数), 217-239, 582-590
set operation and(集合运算), 79-83
update and(更新), 101-103
view and(视图), 120-128
windowing and(分窗), 195-197
tuple visibility(元组可见性), 1139
two-factor authentication(双因素鉴定), 405-407
Two-Phase Commit (2PC) protocol(两阶段提交协议), 786-788, 832-836
two-tier architecture(两层体系结构), 24-25
type(类型), 1017, 1159
abstract data(抽象数据), 1127
array(数组), 956-961
base(基本), 1127
blob(二进制大对象), 138, 166, 457, 502, 1013, 1198-1199, 1259
clob(字符大对象), 138, 166, 457, 502, 1010-1013, 1196-1199
complex data(复杂数据), 946-949(参见 complex data type)
composite(复合), 1127
Document Type Definition (DTD)(文档类型定义), 990-994
enumerated(枚举), 1128
IBM DB2 and, 1196-1197
inheritance and(继承), 949-956
Microsoft SQL Server and, 1229-1230, 1257-1258
most-specific(最明确的), 953
multiset(多重集合), 956-961
.NET Common Language Runtime (CLR) and(.NET 公共语言运行库), 1257-1258
nonstandard(非标准的), 1129-1130
object-based database and(基于对象的数据库), 949-963
object identity(对象标识), 961-963
Oracle and, 1158-1160
performance tuning and(性能调整), 1043
polymorphic(多态), 1128-1129
PostgreSQL, 1126-1129, 1132-1133
pseudotype(伪类型), 1128
reference(引用), 961-963
user-defined(用户自定义), 138-141
single reference(单引用), 972
wide-area(广域), 788-791
XML(可扩展标记语言), 990-998, 1006-1007
UDF(用户自定义函数). 参见 user-defined function
Ultra320 SCSI interface(Ultra320 SCSI 接口), 436
Ultrium format(Ultrium 格式), 451
under privilege, 956
undo(撤销)
concurrency control and(并发控制), 749-750
logical operation and(逻辑运算), 745-750
Oracle and, 1163
recovery system and(恢复系统), 729-738
transaction rollback and(事务回滚), 746-749
undo pass(撤销过程), 754
undo phase(撤销阶段), 737
Unified Modeling Language (UML)(统一建模语言), 17-18
association and(关联), 308-309
cardinality constraint and(基数约束), 309-310
component of(构件), 308
relationship set and(联系集), 308-309
Uniform Resource Locator(URL)(统一资源定位符), 377-378
union(union 操作), 80-81, 585, 220-221
union all(union all 操作), 80
union rule(合并律), 339
unique, 94-95
decomposition and(分解), 354-355
integrity constraint and(完整性约束), 130-131
uniquifier(唯一化), 498-499
United States(美国), 17, 45, 263, 267n3, 411, 788, 858, 869, 922
Universal Coordinated Time(UTC)(全球协调时

间), 1063
Universal Description, Discovery, and Integration (UDDI)(通用描述、发现和集成), 1018
Universal Serial Bus (USB) slot(通用串行总线插槽), 430
universal Turing machine(通用图灵机), 14
university(大学), 2
  application design and(应用程序设计), 375, 392, 407-409
  concurrency control and(并发控制), 698
  database design and(数据库设计), 16-17
  database for(数据库), 3-8, 11-12, 15-19, 27, 30
  E-R model and(E-R 模型), 261-274, 280, 282, 292, 294-299
  indexing and(索引), 477, 510, 529
  query optimization and(查询优化), 586, 589, 605
  query processing and(查询处理), 566
  recovery system and(恢复系统), 724
  relational database design and(关系数据库设计), 323-330, 334, 355, 364-365
  relational model and(关系模型), 41, 43-48
  SQL and(结构化查询语言), 61-63, 70-72, 75, 99, 125-134, 145-150, 153, 170, 173, 187, 192-193, 197, 226-227
  storage and(存储), 452, 458, 460
  system architecture and(系统体系结构), 785, 828, 872
  transaction and(事务), 653
University of California, Berkeley(加州大学伯克利分校), 30, 1123
UNIX(UNIX 操作系统), 77, 438, 713, 727, 1124, 1154, 1193-1194, 1212, 1223
unknown(未知), 83, 90
unnesting(解除嵌套), 958-961
updatable result set(可更新结果集), 166
update-anywhere replication(随地更新副本), 844
update(更新), 101-103
  authorization and(授权), 147, 148
  B+-tree and(B+树), 491-497, 499-500
  batch(批量), 1030-1031
  complexity of(复杂性), 499-500
  concurrency control and(并发控制), 867-868
  data warehouse and(数据仓库), 891
  deletion time and(删除时间), 491, 495-500
  distributed database and(分布式数据库), 826-827
  EXEC SQL and, 171
  hashing and(散列), 516-522
  index and(索引), 482-483
  insertion time and(插入时间), 491-495, 499-500
  log record and(日志记录), 726-734
  lost(丢失), 692
  Microsoft SQL Server and, 1232-1233, 1239
  mobile(移动), 1083-1084
  Oracle and, 1179-1180
  performance tuning and(性能调整), 1030-1033, 1043-1044
  persistent programming language and(持久化程序设计语言), 970
  PostgreSQL and, 1130, 1141-1144, 1147-1148
  privilege and(权限), 143-145
  query optimization and(查询优化), 613-614
  shipping SQL statement to database(传送 SQL 语句到数据库), 161
  snapshot isolation and(快照隔离), 692-697
  trigger and(触发器), 182, 184
  view and(视图), 124-128
  XML and(可扩展标记语言), 1259-1260
user-defined entity set(用户自定义实体集), 299
user-defined function (UDF)(用户自定义函数), 1197-1198
user-defined type(用户自定义类型), 138-141
user interface(用户界面), 27-28
  application architecture and(应用程序体系结构), 391-396
  application program and(应用程序), 375-377
  as back-end component(后端组件), 376
  business-logic layer and(业务逻辑层), 391-392
  client-server architecture and(客户-服务器体系结构), 32, 204, 376-377, 756-772, 777, 788, 791
  client-side scripting and(客户端脚本), 389-391
  cloud computing and(云计算), 861-870
  Common Gateway Interface (CGI)(通用网关接口), 380-381
  cookie and(一小段包含标识信息的文本), 382-385, 403-405
  CRUD, 399
  data access layer and(数据访问层), 391, 393, 395
  disconnected operation and(断连操作), 395-396
  HyperText Transfer Protocol (HTTP) and(超文本传输协议), 377-383, 395, 404-406, 417

IBM DB2, 1195
mobile(移动), 1079-1085
persistent programming language and(持久化程序设计语言), 970
PostgreSQL and, 1124-1126
presentation layer and(表示层), 391
report generator and(报表生成器), 399-400
security and(安全性), 402-417
storage and(存储), 434 (参见 storage)
tool for building(构建工具), 396-398
Web service and(Web 服务), 395
World Wide Web and(万维网), 377-382
user requirement(用户需求), 15-16, 27-28
E-R model and(E-R 模型), 260, 298
performance and(性能), 311-312
response time and(响应时间), 311
throughput and(吞吐量), 311
using(使用), 114
utilization(利用率), 636

vacuum(vacuum 命令), 1143
validation(有效性检查), 703-704
classifier and(分类器), 903-904
concurrency control and(并发控制), 686-689
first committer wins and(首次提交者胜), 692-693
first updater wins and(首次更新者胜), 693
long-duration transaction and(长事务), 1111
phase of(阶段), 688
recovery system and(恢复系统), 729-730
snapshot isolation and(快照隔离), 692-693
view serializability and(视图可串行化), 687
valid time(有效时间), 365
varchar, 59-60, 62
VBScript(VBScript 脚本语言), 387
vector data(矢量数据), 1069
vector space model(向量空间模型), 919-920
version-numbering schema(版本编号方案), 1083-1084
version-vector scheme(版本向量方案), 1084
vertical fragmentation(垂直分片), 828
video server(视频服务器), 1078
view definition(视图定义), 58
view maintenance(视图维护), 608-611
view(视图), 120
authorization on(授权), 146-147
with check option(with check option 子语), 126
complex merging and(复杂归并), 1173-1174
create view and(创建视图), 121-125
cube(立方体), 1221-1222
deferred maintenance and(延迟维护), 1039-1040
definition(定义), 121-122
delete(删除), 125
immediate maintenance and(立即维护), 1039-1040
insert into(插入), 124-125
maintenance(维护), 124
materialized(物化的), 123-124 (参见 materialized view)
performance tuning and(性能调整), 1039-1041
SQL query and(SQL 查询), 122-123
update of(更新), 124-128
view serializability(视图可串行化), 687
virtual machine(虚拟机), 777
virtual processor(虚处理器), 801
Virtual Reality Markup Language (VRML)(虚拟现实标记语言), 390-391
Visual Basic, 169, 180, 397-398, 1228
VisualWeb, 397
volatile storage(易失性存储器), 431, 632, 722

wait-for graph(等待图), 676-678, 845-847
Web crawler(网络爬虫), 930-931
Weblogic, 386
WebObject, 399
Web server(网络服务器), 380-382
Web service(网络服务), 395, 1199-1200
Web Service Description Language (WSDL)(网络服务描述语言), 1018
WebSphere, 386
when clause(when 子句), 181, 184
where clause(where 子句), 311
aggregate function and(聚集函数), 84-90
basic SQL query and(基本 SQL 查询), 63-74
between(在……之间), 78
on multiple relation(多关系), 66-71
natural join and(自然连接), 71-74
not between(不在……之间), 78
null value and(空值), 83-84
query optimization and(查询优化), 605-607
rename operation and(更名运算), 74-75
security and(安全性), 409
set operation and(集合运算), 79-83
on single relation(单关系), 63-65, 65-66
string operation and(字符串运算), 76-79
transaction and(事务), 651-654

while loop(while 循环), 168, 171, 176
Wide-Area Network (WAN) (广域网), 788, 790-791
Wi-Max, 1081
windowing(分窗), 195-197
Windows Mobile, 1223
Wireless Application Protocol (WAP) (无线应用协议), 1081-1082
wireless communication(无线通信), 1079-1085
with check option(with check option 语句), 126
with clause(with 子句), 97, 190
with data(with data 子语), 141-142
with grant option(with grant option 子语), 148
with recursive clause(with recursive 子句), 190
with timezone(with timezone 子语), 136
WordNet, 927
workflow(工作流), 312-313, 836-838, 1017
  acceptable termination state and(可接受终止状态), 1099
  bug and(错误), 1101
  business-logic layer and(业务逻辑层), 391-392
  execution and(执行), 1097-1101
  external variable and(外部变量), 1098-1099
  failure and(故障), 1099-1102
  management system for(管理系统), 1101-1102
  multisystem application and(多系统应用), 1096
  nonacceptable termination state and(不可接受的终止状态), 1099
  performance and(性能), 1029-1048
  recovery of(恢复), 1101
  specification and(说明), 1097-1099
  step and(步骤), 1096
  task and(任务), 1096
  transactional(事务的), 1096-1102
workload compression(工作负载压缩), 1041
World Wide Web(万维网), 31, 885
  application design and(应用程序设计), 377-382
  cookie and(一小段包含标识信息的文本), 382-385, 403-405
  encryption and(加密), 411-417
  HyperText Markup Language (HTML) (超文本标记语言), 378-380
  HyperText Transfer Protocol (HTTP) and(超文本传输协议), 377-381, 383, 395, 404-406, 417
  information retrieval and(信息检索), 915 (参见 information retrieval)
  security and(安全性), 402-417
  service processing and(服务处理), 395
  Simple Object Access Protocol (SOAP) and(简单对象访问协议), 1017-1018
  three-layer architecture and(三层体系结构), 318
  Uniform Resource Locator (URL) (统一资源定位符), 377-378
  Web application framework and(Web 应用构架), 398-400
  Web server and(Web 服务器), 380-382
  XML and(可扩展标记语言), 1017-1018
World Wide Web Consortium (W3C) (万维网联盟), 927, 1056
wrapping(包装), 1055-1056
Write-Ahead Logging (WAL) (先写日志), 739-7841, 1145-1146
Write Once, Read-Many (WORM) disk(写一次读多次光盘), 431
write quorum(写法定人数), 841-842
write-write contention(写写竞争), 1042

X.500 directory access protocol(X.500 目录访问协议), 871
XML(eXtensible Markup Language)(可扩展标记语言), 31, 169, 386, 1020
  Application Program Interface(API) to(应用程序接口), 1008-1009
  application(应用), 1016-1019
  clob value and(字符大对象值), 1010-1011
  data exchange format and(数据交换格式), 1016-1017
  data mediation and(数据中介), 1018-1019
  data structure(数据结构), 986-990
  document schema(文档模式), 990-998
  Document Type Definition (DTD) (文档类型定义), 990-994
  as dominant format(主导格式), 985
  file processing and(文件处理), 981-982
  format flexibility of(格式灵活性), 985
  HTML and(超文本标记语言), 981
  IBM DB2 and, 1195-1196
  join and(连接), 1003-1004
  markup concept and(标记概念), 981-985
  Microsoft SQL Server and, 1258-1261
  nesting and(嵌套), 27, 943, 984-990, 995-998, 1001, 1004-1007, 1010
  Oracle XML DB and, 1159-1160
  publishing/shredding data and(发布/分解数

据), 1013
query and(查询), 998-1008, 1259-1260
relational database and(关系数据库), 1010-1014
relational map and(关系图), 1012
Simple Object Access Protocol (SOAP) and(简单对象访问协议), 1017-1018
sorting and(排序), 1006
SQL/XML standard and (SQL/XML 标准), 1014-1015
standard for(标准), 1055-1056
storage and(存储), 1009-1016
tag and (标签), 982-986, 989, 994, 999, 1004, 1019
textual context and(文本上下文), 986
transformation and(转换), 998-1008
tree model of(树模型), 998
update and(更新), 1259-1260
web service and(Web 服务), 1017-1018
wrapping and(包装), 1055-1056
xmlagg(xmlagg 函数), 1015
xmlattributes(xmlattributes 函数), 1015
xmlconcat(xmlconcat 算子), 1015
xmlelement(xmlelement 函数), 1015
xmlforest(xmlforest 算子), 1015
XMLIndex(XMLIndex 类型), 1160
XML Schema(XML 模式), 994-998
XMLType(XMLType 类型), 1159
X/Open XA standard (X/OPEN XA 标准), 1053-1054
XOR operation(XOR 运算), 413
XPath, 1160
document schema and(文档模式), 997
query and(查询), 998-1002
storage and(存储), 1009-1015
XQuery, 31, 998
FLWOR expression and (FLWOR 表达式), 1002-1003
function and(函数), 1006-1007
join and(连接), 103-104
Microsoft SQL Server and, 1260-1261
nested query and(嵌套查询), 1004-1005
Oracle and, 1160
sorting of result and(结果排序), 1006
storage and(存储), 1009-1015
transformation and(转换), 1002-1008
type and(类型), 1006-1007
XSLT(XSL 转换), 1160

Yahoo(雅虎), 390, 863
YUI library(YUI 库), 390