



SQL基础教程 (第3版)

[美] Chris Fehily 著
冯宇晖 贾文峰 译

 Amazon五星图书，自学与参考两相宜

 任务驱动，数百实例教你掌握SQL精髓

 触类旁通，展现主流DBMS的SQL语句异同



人民邮电出版社
POSTS & TELECOM PRESS

更多资源请访问我的新浪博客<http://blog.sina.com.cn/ckook>

版 权 声 明

Authorized translation from the English language edition entitled *SQL: Visual QuickStart Guide, Third Edition* by Chris Fehily, published by Pearson Education, Inc., publishing as Peachpit Press, Copyright © 2008 by Chris Fehily.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanic, including photocopying, recording, or by an information storage retrieval system, without permission of Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by POST & TELECOM PRESS Copyright © 2009.

本书中文简体字版由美国Pearson Education 授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制本书内容。

版权所有，侵权必究。

译者序

SQL是关系数据库编程的国际标准语言。本书系统、全面地介绍了标准SQL，并讲解了Microsoft Access、Microsoft SQL Server、Oracle、IBM DB2、MySQL、PostgreSQL等SQL实现及其与标准SQL之间的差异。

本书结构严谨、内容丰富，给出大量实例代码及经验技巧。本书适合SQL初学者，同时也可作为数据库应用开发人员和最终用户的参考书。作为初学者，可通过学习本书快速、全面地掌握SQL；而作为数据库应用开发人员和最终用户，可以通过本书提高开发和应用关系数据库的能力。

本书由冯宇晖、贾文峰翻译，其中文前、第1~6章、第7章前6节由贾文峰翻译，第7章7~9节及第8~15章由冯宇晖翻译，全书由冯宇晖负责统稿。限于译者的水平，译文中难免有错误与不足之处，敬请读者和同行批评指正。译者的邮箱：fengyh1999@163.com和jwf@lit.edu.cn。

前　　言

SQL是一种标准的编程语言，用于创建、更新和检索存储在数据库中的信息。使用SQL，你可以将通常的问题（“我们的客户居住在何地？”）转化为数据库系统能够理解的语句（`SELECT DISTINCT city, state FROM customers;`）。你可能已经知道如何通过图形化的查询或报表工具来检索此类信息，但对于复杂的问题，或许你也意识到此类工具存在诸多限制和障碍——这时就需要SQL了。

可以使用SQL来增加、更新、删除数据和数据库对象。所有现代关系型DBMS（数据库管理系统）都支持SQL，但不同产品的支持情况会有所不同。

本书涉及大多数流行DBMS的最新版本，并专门新增一章来介绍SQL技巧，包括最新的编程技巧提示、细微变化和高级主题，以及其他零星知识。

关于 SQL

SQL可以如下表述：

- 是一种编程语言；
- 容易学习；
- 是说明性语言；
- 是交互式或嵌入式语言；
- 是标准化语言；
- 用于改变数据和数据库对象；
- 不是首字母缩写词。

它是一种编程语言。SQL是一种可以用来编写程序，实现创建、修改、查询数据库的形式语言。数据库系统执行SQL程序，完成用户指定的任务，并显示结果或错误信息。编程语言有别于自然语言，它是为了特殊用途而设计的，其词汇量小、语句书写呆板且必须明确无误。如果你没有得到希望的结果，只能怪你自己的程序包含某些错误，而不是计算机错误地执行了指令（程序测试是编程的一项重要工作）。

和其他形式语言一样，SQL由一系列语法和语义规则定义。语法规则决定可以使用哪些单词和符号，以及如何将它们结合在一起。语义规则决定着语法正确的语句的真实含义。注意，SQL语句可能语法正确但语义错误。第3章会介绍SQL的语法和语义。

数据库与DBMS

数据库并不等同于用户使用的数据库软件，“Oracle是数据库”是不正确的说法。数据库软件被称作DBMS。数据库只是DBMS的一个组成部分，是数据本身；也就是说，它是一个存储结构化信息的容器（由至少一个文件组成）。除了控制数据库中数据的组织、完整性和检索，DBMS还要负

责其他任务，如物理存储、安全、备份、错误处理等。

本书中可以认为DBMS是RDBMS的简写，在这里R代表关系型。关系型DBMS依照关系模型（见第2章）而不是层次模型或网状模型来组织数据。本书只讲解关系型系统，所以当书中出现DBMS时，就是指RDBMS，第一个字母R（关系型）被省略了。

它容易学习。同其他编程语言相比，SQL学起来更容易。如果以前没有写过任何程序，你会觉得从自然语言转向形式语言是很难的。不过，SQL语句读起来很像句子，容易学习。初学编程的人可能会理解SQL语句“`SELECT au_fname, au_lname FROM authors ORDER BY au_lname;`”的意思是“以姓氏为序列出作者的姓名”，但他们将会发现，具有相同功能的C或Perl程序很难理解。

它是说明性语言。如果你从未编写过程序，那么跳过本段也不会影响你学习后面的内容。如果你用C或者PHP语言编写过程序，那意味着你有使用过程语言的经历，在编写这样的程序时需要指明得出结果所需的每个步骤。使用SQL这种说明性语言，只需描述想要的内容，而无需去管该如何做，数据库系统的优化器将决定“如何做”。因此，标准的SQL没有传统的流程控制结构，如`if-then-else`、`while`、`for`和`goto`语句。

为了说明这一点差异，本书给出了功能等同的Microsoft Access Visual Basic程序（VB属于过程语言）和SQL程序。代码0-1显示了从包含作者信息的表中检索作者姓名的VB程序。你无需了解整个程序，但要注意它如何使用`Do Until`循环来显式地定义如何提取数据。代码0-2显示了如何用一条SQL语句（与约20行的VB程序形成鲜明对比）完成相同的功能。使用SQL，只需指明要完成什么工作，数据库管理系统在内部决定并执行得出结果所需的具体操作。

代码0-1 这段Microsoft Access Visual Basic程序从一个包含作者信息的数据库表中获取姓名信息，并将结果存放在一个数组中

```
Sub GetAuthorNames()
    Dim db As Database
    Dim rs As Recordset
    Dim i As Integer
    Dim au_names() As String
    Set db = CurrentDb()
    Set rs = db.OpenRecordset("authors")
    rs.MoveLast
    ReDim au_names(rs.RecordCount - 1, 1)
    With rs
        .MoveFirst
        i = 0
        Do Until .EOF
            au_names(i, 0) = ! [au_fname]
            au_names(i, 1) = ! [au_lname]
            i = i + 1
        .MoveNext
    Loop
    End With
    rs.Close
    db.Close
End Sub
```

此外，代码0-2是一个简单的SQL查询。在对简单的SQL查询添加排序、过滤、联结等常见的操作后，使用这样一条SELECT语句就可以完成的任务，使用过程代码则可能需要100多行。

代码0-2 这条SQL语句完成的查询与代码0-1中Visual Basic程序完成的查询相同。Access内部的优化器可决定获取数据的最佳方式

```
SELECT au_fname, au_lname
FROM authors;
```

它是交互式或嵌入式语言。在交互式SQL环境中，用户输入的SQL命令直接发送到数据库管理系统，得到结果后立即显示。DBMS的服务器同时拥有图形和命令行工具，用于接受用户输入的SQL语句或包含SQL程序（脚本）的文本文件。

在开发数据库应用程序时，可以将SQL语句“嵌入”到编写程序所用的宿主语言（host language）中。宿主语言通常是一种通用语言（如C++、Java或COBOL）或脚本语言（如Perl、PHP或Python）。例如，一个PHP CGI脚本可以用SQL语句来查询MySQL数据库；MySQL将查询的结果返回给PHP变量，以便进一步分析或显示在网页上。根据前面的例子，可以将SQL语句嵌入到Visual Basic程序中（代码0-3）。

代码0-3 这里，Visual Basic作为嵌入式SQL的宿主语言

```
Sub GetAuthorNames2()
    Dim db As Database
    Dim rs As Recordset
    Set db = CurrentDb()
    Set rs = db.OpenRecordset("SELECT au_fname, au_lname FROM authors;")
    '--Do something with rs here.
    rs.Close
    db.Close
End Sub
```

本书只包括交互式SQL。一般情况下，任何可以交互使用的SQL语句，也都可以用在宿主语言中。但在DBMS、宿主语言和操作环境中，语法上略有差异。

它是标准化语言。SQL不属于任何公司。它是一个由国际标准化组织（ISO）和国际工程协会（IEC）共同领导的国际标准工作组定义的开放标准。美国国家标准协会（ANSI）参加了这个工作组，并已批准该标准（见图0-1）。因为“ISO/IEC SQL”不常用，所以在本书中使用更常见的“ANSI SQL”。本书依据的是2003年的SQL标准，除非另外指明，否则书中的*ANSI SQL*、*SQL:2003*和*SQL*所指是一样的。要了解更多相关内容，参见3.2节。

所有DBMS供应商都增加了专有功能来增强这种语言。这些扩展通常是额外的命令、关键字、函数、操作符、数据类型，还有流程控制结构（如if、while和goto语句）。微软、甲骨文和IBM对标准SQL增加了相当多的功能，于是分别形成了Transact-SQL、PL/SQL和SQL PL语言，可被视为这些供应商自己的语言，而不只是SQL的超集。某个供应商的扩展通常与其他供应商的产品不

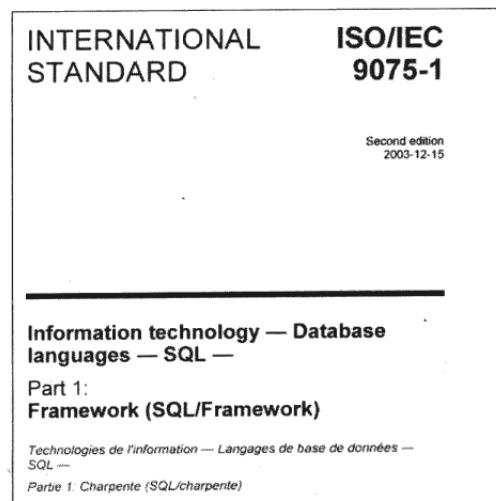


图0-1 这是ISO/IEC 9075:2003标准的封面，对SQL:2003给出了官方定义。你可以从www.ansi.org或www.iso.org购买电子版。该标准面向的读者不是SQL程序员，而是那些设计DBMS、编译器和优化器的人

兼容。本书没有介绍专有的SQL扩展，但当某供应商专有的SQL不符合书中给出的标准SQL例子时，会特别指出。

它用于改变数据和数据库对象。SQL语句分为以下3类。

- 数据操作语句（DML）用于在数据库中检索、计算、插入、编辑和删除数据。第4章至第10章介绍了数据操作语句SELECT、INSERT、UPDATE和DELETE。第14章介绍了语句START（或BEGIN）、COMMIT和ROLLBACK。
- 数据定义语句（DDL）用于创建、修改和销毁数据表、索引、视图等数据库对象。第11章至第13章介绍了数据定义语句CREATE、ALTER和DROP。
- 数据控制语句（DCL）用于授权某些用户查看、更改、删除数据或数据库对象。GRANT语句分配权限和角色（角色是一组权限的集合）。REVOKE语句用于取消权限。本书没有介绍GRANT和REVOKE，因为它们是数据库管理员的职责。书中介绍的所有DBMS（除了Access）都支持GRANT和REVOKE，但与SQL标准有一些差异。

它不是首字母缩写词。“SQL表示结构化查询语言”是一种普遍的误解。它只代表S-Q-L，不代表任何其他意思。为什么？因为ANSI就是这样解释的。官方名字是数据库语言SQL，如图0-1所示。而且，如果说SQL表示“结构化查询语言”，那会令刚开始学习SQL编程的新手感到困惑。业内人士认为“结构化查询语言”可能是SQL最糟糕的解释，因为SQL：

- 不是结构化的（无法将其分解成程序块或过程）；
- 不只用于查询（它不只包含SELECT语句）；
- 不是一种语言（它不满足图灵完备性，你在学习计算理论时会遇到这个概念）。

关于本书

本书将介绍如何使用SQL编程语言维护和查询数据库信息。第1章至第3章将介绍各种DBMS、关系模型和SQL语法，之后会继续沿袭读者已经熟悉的*Visual QuickStart*系列图书基于任务、图解式的编写风格。

尽管阅读本书不要求读者有编程经验，但希望你熟悉操作系统的文件系统，知道如何在命令行模式或shell（在较早的Windows版本里称作DOS提示符，在Mac OS X里称作Terminal）中输入命令。

本书只介绍最常用的语句，不是SQL的详尽指南。要了解更多的SQL语句知识，请参考所用DBMS文档或包含更详细SQL标准的参考书。

✓ 提示

- Peter Gulutzan和Trudy Pelzer所著的*SQL-99 Complete, Really* (CMP Books) 完整解释了SQL-99标准。要比SQL标准看着轻松，但没有提及具体的DBMS。
- Kevin Kline、Daniel Kline和Brand Hunt所著的*SQL in a Nutshell* (O'Reilly) 包含了本书介绍的DBMS（除Access以外），是比较全面的SQL:2003参考书。它适合于已经有一定基础的SQL程序员。
- Troels Arvin的“Comparison of Different SQL Implementations”解释了DBMS实现SQL各种功能的不同之处，包括源文件和其他SQL图书、文章、资源的链接。它包括了SQL:2003和与本书相同的DBMS（除Access以外），见<http://troels.arvin.dk/db/rdbms>。

配套站点

在www.fehily.com,你可以找到修正、更新代码和可供下载的示例数据库(见2.6节)。单击Contact链接就可以给我发对本书的问题、建议、修改及批评意见。

SQL Server与桌面DBMS

SQL服务器DBMS在客户/服务器网络中作为服务器端，它存储数据并响应客户端的SQL请求。客户端是一个应用程序或一台向服务器发送SQL请求并接受服务器响应的计算机。服务器真正执行对数据库的操作，客户端只是接收答案。如果网络使用了客户/服务器架构，客户端就是眼前桌面上的计算机，服务器则是放在另一房间、另一座大楼甚至另一个国家的强大的专业化设备。客户/服务器的请求和响应如何传递的规则在DBMS协议和类似ODBC、JDBC和ADO.NET的接口中定义。

桌面DBMS是独立的程序，它存储数据库并执行所有SQL指令，或作为SQL服务器的客户端。桌面DBMS不能接受其他客户端的请求（或者说，不能作为SQL服务器）。

SQL服务器包括Microsoft SQL Server、Oracle、DB2、MySQL和PostgreSQL。桌面系统包括Microsoft Access和FileMaker Pro。注意，SQL server（没有大写）指的是所有供应商的SQL服务器产品，SQL Server（大写）是Microsoft的SQL服务器产品。

读者对象

本书的读者对象是数据库应用开发人员和最终用户（不是数据库设计人员或管理人员），主要适合以下人员。

- 熟悉计算机但缺乏编程经验。
- 正在自学或跟随指导老师学习SQL。
- 对数据库没有兴趣，但因为工作需要必须处理大量结构化信息。这类人包括统计人员、流行病工作者、网站编程人员、气象工作者、工程师、会计、投资者、科研人员、分析师、销售代表、金融规划和交易员、办公室文员及经理人。
- 对易用但不够强大的图形查询工具不太满意的人。
- 正从桌面数据库转到DBMS服务器（参见“SQL Server与桌面DBMS”）。
- 已经了解一些SQL，不满足于简单的SELECT语句。
- 需要创建、修改或删除表、索引和视图等数据库对象。
- 需要将SQL代码嵌入到C、Java、Visual Basic、PHP、Perl或其他宿主语言。
- 作为网站程序员，需要在网页上展示查询结果。
- 需要桌面SQL的参考书。
- 因为数据列表已经变得太大、太复杂，而无法使用电子表格管理，正从Microsoft Excel转向Microsoft Access。

本书不适合希望学习以下内容的读者。

- 如何设计数据库（尽管在第2章简述了设计概念）。
- DBMS供应商在基本SQL语句上增加的扩展特性。

- 高级编程或管理。本书没有包含安装、权限、触发器、递归^①、存储过程、复制、备份、恢复、游标、整理、字符集、翻译、XML及面向对象扩展。

排版约定

本书使用以下排版约定。楷体表示引入的新术语。英文斜体表示变量。代码体表示SQL代码和代码及一般文本里的语法，也表示可执行的代码、文件名、目录（文件夹）名、URL和命令行提示符文本。粗代码体表示SQL代码片段和结果对应正文中的解释部分。斜代码体表示SQL代码中必须使用值来取代的变量，例如必须用真实的列名取代*column*。

语法约定

SQL是对行中断和每行长度都沒有限制的自由形式语言。为了使代码易于阅读和维护，本书在SQL语法图解和代码中使用一致的风格。

- 每个SQL语句开始于新行。
- 每次缩进为2个字符。
- 每个语句的子句开始于新的缩进的行。

```
SELECT au_fname, au_lname
  FROM authors
 ORDER BY au_lname;
```

- SQL是不区分大小写的，这意味着`myname`、`MyName`和`MYNAME`是同样的标识符。本书对于SQL关键字（如`SELECT`、`NULL`和`CHARACTER`）使用大写（见3.1节）；对于用户定义的值（如表、列和别名）使用小写。（对于一些DBMS，用户定义的标识符在引用时或其他一些场合是区分大小写的，因此最安全的方法，是在SQL程序中遵照标识符的大小写规定。）
- 表0-1显示了本书在语法图中使用的特殊符号。
- 在SQL代码中所有引号是直的引号（如'和"），而不是弯曲的引号（如'和“），弯曲的引号会使代码无法运行。
- 当一行太窄而无法容纳一整行代码或输出时，将其分为两段或两段以上；箭头→表示行的继续。

表0-1 语法符号

符 号	说 明
	竖线或管道符分隔可替换的项目，采用其中任何一个即可（不要输入竖线本身）。 <code>A B C</code> 读作“ <code>A或B或C</code> ”。不要将管道符与双管道符 混淆，后者是SQL的字符串连接符
[]	方括号包围一个或多个可选项（不要输入方括号本身）。 <code>[A B C]</code> 意味着“输入 <code>A</code> 或 <code>B</code> 或 <code>C</code> ，或不输入”， <code>[D]</code> 意味着“输入 <code>D</code> ，或不输入”
{ }	大括号包围一个或多个必选项。（不要输入大括号本身。） <code>{A B C}</code> 意味着“输入 <code>A</code> 或 <code>B</code> 或 <code>C</code> ”
...	省略号意味着前面的项可以重复任意次

① 要理解递归，就要先了解递归——这就是递归。

在特定的DBMS下使用SQL



这个图标表示某个产品未遵循SQL:2003标准。如果见到这个图标，就要注意特定供应商的SQL语法并不遵循标准，必须修改列出的SQL代码才能在DBMS上运行。例如，标准SQL连接两个字符串的操作符是||（双管道符），但Microsoft产品使用+（加号），MySQL使用CONCAT函数，需要将例子代码中所有的`a||b`变为`a + b`（如果使用Microsoft Access或Microsoft SQL Server）或`CONCAT(a,b)`（如果使用MySQL）。在大多数场合，SQL例子运行正常或只需进行小的修改。因为有的DBMS不支持某种功能，SQL代码偶尔也会无法工作。

本书涉及以下DBMS（详细内容见第1章）。

- Microsoft Access
- Microsoft SQL Server
- Oracle
- IBM DB2
- MySQL
- PostgreSQL

如果你使用其他DBMS（如Teradata、Sybase或Informix），遇到某个SQL例子无法运行，请查阅文档了解所用DBMS的SQL实现与SQL标准有何不同。

配置环境

要在计算机中改写后运行本书中的示例代码，需要用到：

- 文本编辑器
- 示例数据库
- 数据库管理系统

文本编辑器。在命令行提示符中输入简短或临时性的交互式SQL语句虽然很方便，但这样做就不能把有价值的SQL语句存储在文本文件里。文本编辑器是打开、创建、编辑文本文件的程序，它只包含可打印的字符、数字、符号，不支持字体、格式、不可见的符号、颜色、图像或其他文字处理软件中才有的复杂功能。操作系统都会包含免费的文本编辑器，如Windows有记事本，Unix有vi和emacs，Mac OS X有TextEdit。按照惯例，SQL文件的扩展名是.sql，但你也可以使用.txt（或其他任何扩展名）。

✓ 提示

- Windows用户可以不使用记事本而选择其他文本编辑工具，如TextPad（需要30美元，www.textpad.com）、EditPlus（需要30美元，www.editplus.com）或Vim（免费，www.vim.org）。
- 也可以在文字处理软件（如Microsoft Word）中输入SQL程序，然后将它们存储为纯文本文件，但这样做会带来维护方面的问题（且专业人士认为这种方式不太好）。

示例数据库。本书中大多数例子使用了同一个数据库，在2.6节中有这个数据库的说明。要创建这个示例数据库，请按照2.7节的指导进行操作。如果你正在使用DBMS服务器，就需要从数据库管理员那里得到许可才能运行创建、修改数据和数据库对象的SQL程序。

数据库管理系统。如何处理SQL？你不必处理——DBMS理解SQL，只要在其中输入SQL程序就行了。DBMS将运行程序并显示结果（见第1章）。

目 录

第 1 章 DBMS 介绍	1
1.1 运行 SQL 程序	1
1.2 Microsoft Access	3
1.3 Microsoft SQL Server	6
1.3.1 SQL Server 2000	7
1.3.2 SQL Server 2005/2008	9
1.4 Oracle	10
1.5 IBM DB2	12
1.6 MySQL	16
1.7 PostgreSQL	17
第 2 章 关系模型	20
2.1 表、列和行	21
2.1.1 表	21
2.1.2 列	21
2.1.3 行	22
2.2 主键	24
2.3 外键	25
2.4 联系	26
2.4.1 一对一	27
2.4.2 一对多	27
2.4.3 多对多	28
2.5 规范化	29
2.5.1 第一范式	29
2.5.2 第二范式	30
2.5.3 第三范式	31
2.5.4 其他范式	32
2.6 示例数据库	33
2.6.1 表 authors	33
2.6.2 表 publishers	34
2.6.3 表 titles	35
2.6.4 表 titles_authors	35
2.6.5 表 royalties	36
2.7 创建示例数据库	37
第 3 章 SQL 基础	40
3.1 SQL 语法	40
3.2 SQL 标准和一致性	42
3.3 标识符	43
3.4 数据类型	44
3.5 字符串类型	45
3.6 二进制大型对象类型	47
3.7 精确数字类型	48
3.8 近似数字类型	49
3.9 布尔类型	50
3.10 日期和时间类型	51
3.11 时间间隔类型	53
3.12 唯一标识符	54
3.13 其他数据类型	55
3.14 空值	55
第 4 章 从表中检索数据	58
4.1 使用 SELECT 和 FROM 检索列	58
4.2 使用 AS 创建列的别名	61
4.3 使用 DISTINCT 消除重复的行	62
4.4 使用 ORDER BY 排序行	63
4.5 使用 WHERE 筛选行	68
4.6 使用 AND、OR 和 NOT 组合及求反条件	71
4.6.1 AND 操作符	71
4.6.2 OR 操作符	72
4.6.3 NOT 操作符	73
4.6.4 AND、OR 和 NOT 一起使用	74
4.7 使用 LIKE 匹配模式	77

4.8 使用 BETWEEN 进行范围筛选	81	7.5 使用 CROSS JOIN 创建交叉联结	141
4.9 使用 IN 进行列表筛选	83	7.6 使用 NATURAL JOIN 创建自然联结	143
4.10 使用 IS NULL 测试空值	85	7.7 使用 INNER JOIN 创建内联结	146
第 5 章 操作符和函数	88	7.8 使用 OUTER JOIN 创建外联结	165
5.1 创建派生列	88	7.9 创建自联结	173
5.2 执行算术运算	89		
5.3 确定计算的顺序	92		
5.4 使用 连接串	92		
5.5 使用 SUBSTRING() 提取子串	95		
5.6 使用 UPPER() 和 LOWER() 更改串的大 小写	97		
5.7 使用 TRIM() 修整字符	99		
5.8 使用 CHARACTER_LENGTH() 得到串长度	101		
5.9 使用 POSITION() 查找子串	103		
5.10 执行日期及时间间隔运算	105		
5.11 获得当前日期和时间	106		
5.12 获得用户信息	108		
5.13 使用 CAST() 转换数据类型	109		
5.14 使用 CASE 计算条件值	112		
5.15 使用 COALESCE() 检查空值	115		
5.16 使用 NULLIF() 比较表达式	116		
第 6 章 汇总和分组数据	118		
6.1 使用聚合函数	118		
6.2 创建聚合表达式	119		
6.3 使用 MIN() 查找最小值	120		
6.4 使用 MAX() 查找最大值	120		
6.5 使用 SUM() 计算总和	121		
6.6 使用 AVG() 计算平均值	122		
6.7 使用 COUNT() 统计行数	124		
6.8 使用 DISTINCT 聚合不重复的值	125		
6.9 使用 GROUP BY 分组行	127		
6.10 使用 HAVING 筛选分组	132		
第 7 章 联结	135		
7.1 限定列名	135		
7.2 使用 AS 创建表的别名	136		
7.3 使用联结	137		
7.4 使用 JOIN 或 WHERE 创建联结	139		
		第 8 章 子查询	177
		8.1 理解子查询	177
		8.2 子查询语法	179
		8.3 子查询和联结	179
		8.4 简单子查询和相关子查询	182
		8.4.1 简单子查询	183
		8.4.2 相关子查询	183
		8.5 在子查询中限定列名	186
		8.6 子查询中的空值	187
		8.7 使用子查询作为列表达式	188
		8.8 使用比较操作符比较子查询的值	191
		8.9 使用 IN 测试集合成员资格	194
		8.10 使用 ALL 比较所有子查询的值	200
		8.11 使用 ANY 比较某些子查询的值	202
		8.12 使用 EXISTS 检测存在性	205
		8.13 比较等价查询	209
		第 9 章 集合操作	212
		9.1 使用 UNION 合并行	212
		9.2 使用 INTERSECT 查找相同行	217
		9.3 使用 EXCEPT 查找不同行	218
		第 10 章 插入、更新和删除行	220
		10.1 显示表结构	220
		10.2 使用 INSERT 插入行	223
		10.3 使用 UPDATE 更新行	228
		10.4 使用 DELETE 删除行	232
		第 11 章 创建、更改和删除表	235
		11.1 创建表	235
		11.2 理解约束	236
		11.3 使用 CREATE TABLE 创建新表	236
		11.4 使用 NOT NULL 禁止空值	238
		11.5 使用 DEFAULT 确定默认值	240
		11.6 使用 PRIMARY KEY 指定主键	242

11.7 使用 FOREIGN KEY 指定外键	244	15.4.2 Microsoft SQL Server	293
11.8 使用 UNIQUE 确保值唯一	248	15.4.3 Oracle	294
11.9 使用 CHECK 创建检查约束	250	15.4.4 IBM DB2	295
11.10 使用 CREATE TEMPORARY TABLE 创建临时表	252	15.4.5 MySQL	296
11.11 使用 CREATE TABLE AS 利用已存在表创建新表	254	15.4.6 PostgreSQL	297
11.12 使用 ALTER TABLE 修改表	258	15.5 分配排名	298
11.13 使用 DROP TABLE 删除表	259	15.6 计算修整均值	299
第 12 章 索引	261	15.7 随机选取行	300
12.1 使用 CREATE INDEX 创建索引	261	15.8 处理重复值	302
12.2 使用 DROP INDEX 删除索引	264	15.9 创建电话列表	304
第 13 章 视图	265	15.10 检索元数据	305
13.1 使用 CREATE VIEW 创建视图	265	15.10.1 Microsoft Access	305
13.2 通过视图检索数据	269	15.10.2 Microsoft SQL Server	305
13.3 通过视图修改数据	271	15.10.3 Oracle	306
13.3.1 通过视图插入行	271	15.10.4 IBM DB2	307
13.3.2 通过视图更新行	272	15.10.5 MySQL	307
13.3.3 通过视图删除行	273	15.10.6 PostgreSQL	308
13.4 使用 DROP VIEW 删除视图	274	15.11 处理日期	308
第 14 章 事务	275	15.11.1 Microsoft Access	309
第 15 章 SQL 技巧	280	15.11.2 Microsoft SQL Server	310
15.1 动态统计	280	15.11.3 Oracle	310
15.2 产生序列	283	15.11.4 IBM DB2	311
15.3 发现等差数列、递增数列和等值数列	287	15.11.5 MySQL	312
15.4 限定返回行的数量	291	15.11.6 PostgreSQL	313
15.4.1 Microsoft Access	292	15.12 计算中值	315
		15.13 查询极值	316
		15.14 改变动态统计的中流	317
		15.15 旋转结果	318
		15.16 处理层次结构	320
		索引	326

第1章

DBMS介绍

1

运行SQL程序需要DBMS (Database Management System, 数据库管理系统)。可以让DBMS运行在桌面(本地)计算机上,或者通过网络使用共享的DBMS。在后一种情况下,可以用桌面计算机连接运行在另一台机器上的DBMS服务器。运行DBMS的计算机称为主机(host)。

因为本书主要介绍SQL而非DBMS,所以将不再重复讲解安装和配置数据库软件。初看,这像是个一笔带过的借口,但其实配置DBMS因供应商、产品、版本、版本类型和操作系统而异,确实非常复杂。而且所有的DBMS都提供了大量的安装、管理、参考和指南文档(仅Oracle的安装手册就超过300页)。

1.1 运行 SQL 程序

本章将介绍如何在下面这些DBMS中运行SQL。

- Microsoft Access 2007
- Microsoft SQL Server 2008
- Oracle 11g
- IBM DB2 9.5
- MySQL 5.1
- PostgreSQL 8.3

这些系统是最流行的商业或开源DBMS。本书中的SQL例子已通过上述版本的测试。这些例子可以运行在更新的版本中,但是不一定能运行于较早的版本中。后续的版本一般会更符合SQL标准。

Microsoft Access的图形化界面每次只能运行一条SQL语句,而其他系统(所有的DBMS服务器)能够以交互模式(interactive mode)或脚本模式(script mode)运行SQL程序。在交互模式下,可以在命令行提示符下输入各条SQL语句然后单独查看每条语句的结果,因此输入和输出是交替的。在脚本模式(也称为批处理模式)下,将完整的SQL程序保存在一个文本文件(称为脚本或批处理文件)中,由命令行工具获取文件、执行程序并返回结果,而不需要人为干预。本章所有例子使用了同一个示例数据库和代码1-1中所示的SQL程序。同时,也介绍了命令行工具的少量语法,完整的语法请参考DBMS文档。

代码1-1 这个名为listing0101.sql的文件包含了一条简单的SQL SELECT语句(在后续的DBMS例子中,将用它来查询示例数据库)

```
SELECT au_fname, au_lname
```

```
FROM authors
ORDER BY au_lname;
```

命令行

大多数数据库专业人员喜欢通过DBMS命令行环境提交命令和SQL脚本，而不是采用鼠标操作菜单和图形化界面的窗口（数据库管理员不会通过单击的方式添加1 000个用户）。如果对DBMS不熟悉，可能会觉得命令行神秘且令人生畏，但一经体验就会发现它强大、简易而高速。图形化工具有一些优点，如：

- 支持剪贴板的剪切、复制和粘贴功能。
- 没有限制的水平和垂直滚动条。
- 通过拖动鼠标可以改变列的宽度。
- 便于查看历史命令和结果。

命令行爱好者也许想体验一下HenPlus (<http://henplus.sourceforge.net>)：一个跨DBMS、免费、全功能的SQL命令解释器。通过网页搜索*sql front end*或*sql client*，可以找到其他解释器。使用Windows的Unix爱好者能够通过Cygwin (www.cygwin.com) 或UWIN (www.research.att.com/sw/tools/uwin) 来运行流行的Unix shell。

2

路径名

路径名指定了在文件系统层次结构中一个目录或文件的唯一位置。绝对路径名是从目录树最顶层的节点（称为根）开始的完整的路径。相对路径名是相对于当前（或工作）目录的路径。在Windows中，绝对路径以一个反斜杠（\）或一个驱动器字母后跟一个冒号（：）和一个反斜杠（\）开始。在Unix或Mac OS X终端中，绝对路径以一个斜杠（/）开始。例如，C:\Program Files\Microsoft SQL Server（在Windows中）和/usr/local/bin/mysql（在Unix中）是绝对路径。scripts\listing0101.sql（在Windows中）和doc/readme.txt（在Unix中）是相对路径。网络中文件和文件夹的绝对路径也可以用双反斜杠和服务器名开始（例如，\\某个服务器）。在本书中，如果路径名包含空格，则会在整个路径名两端加上双引号。

路径名通常简称为路径。尽管依据上下文很容易明白指的是路径名还是路径，但是本书还是使用路径名，以避免与PATH环境变量混淆。

✓ 提示

- 当在脚本模式中指定SQL文件名时，可以使用绝对或相对路径名（参见“路径名”提要栏）。
- 要想从特定的目录（文件夹）运行命令行工具，路径中必须包括确实含有此工具的目录。路径是操作系统查找程序的目录列表。对于某些DBMS，由安装程序处理路径的详细信息；对于另一些DBMS，必须自行把工具目录添加到路径中。要查看路径，在命令行提示符下输入path（在Windows中）或echo \$PATH（在Unix或Mac OS X终端中）。要更改路径，需要将工具所在的目录添加到path环境变量中。要了解更多内容，请查找环境变量的帮助（在Windows中），或者在登录初始化文件（通常被命名为.bash_login、.bashrc、.cshrc、.login、.profile或.shrc）中修改路径命令（在Unix或Mac OS X中）。

3

其他DBMS

FileMaker Pro (www.filemaker.com) 是支持SQL子集的桌面数据库程序。可以通过SQL查询生成器工具或者执行SQL脚本设置运行SQL语句。

Sybase (www.sybase.com) 是商业DBMS服务器。Sybase和Microsoft曾经有过共享源代码的协议，因而它们的DBMS一度几乎是完全一样的。数年前，这两家公司分道扬镳，各自研发自己的数据库产品。但是，共享的历史意味着运行在Microsoft SQL Server上的绝大多数SQL例子能够很好地运行在Sybase Adaptive Server上。

Teradata (www.teradata.com) 是支持巨型数据库和大量事务的商业DBMS服务器。Teradata的SQL语言在很大程度上支持ANSI SQL，因此本书中的大多数例子几乎不用修改就可以在该DBMS中运行。

Firebird (www.firebirdsql.org) 是从Borland的InterBase DBMS发展而来的开源数据库。它是免费的，可以支持巨型数据库和大量事务，非常符合ANSI SQL标准，并能够运行在多种操作系统和硬件平台上。

SQLite (www.sqlite.org) 是开源的DBMS数据库引擎。它是免费的，可以支持巨型数据库和大量事务，初步符合ANSI SQL标准，能够运行在多种操作系统和硬件平台上。访问SQLite数据库的应用软件直接读写磁盘上的文件，而不需要中间服务器。

SAS (www.sas.com) 是商业统计和数据仓库系统。尽管SAS不是关系型DBMS，但能够使用ANSI或特定的DBMS SQL，通过PROC SQL或SAS/Access导入和导出SAS数据。SAS数据集相当于SQL表，观察资料(observation)相当于SQL行，而变量(variable)则相当于SQL列。

在 http://en.wikipedia.org/wiki/Comparison_of_relational_database_management_systems 的“Comparison of Relational Database Management Systems”中，可以查找到更多的信息和有用的链接。

1.2 Microsoft Access

Microsoft Access是支持中小规模数据库的商业桌面DBMS。可以在www.microsoft.com/office/access上学习Access并下载60天免费试用版。

本书主要讲解了Microsoft Access 2007，但也包括对于2000、2002（也称为Access XP）和2003的提示。要确定正在运行的Access是哪一个版本，在Access 2003或先前的版本中，选择Help→About Microsoft Office Access；在Access 2007或后续的版本中，单击Microsoft Office按钮（窗口左上角），选择Access Options→Resources（在左侧窗格中）→About。

在Access中，必须开启ANSI-92 SQL语法来运行本书中的大多数例子。

ANSI-89和ANSI-92 SQL之间的比较

在ANSI-89（对于Access是默认的）和ANSI-92 SQL语法模式之间进行转换应该格外小心。两种模式彼此不兼容，因此当创建数据库时需要选择一种模式且不再更改。数据类型、保留字和通配符因模式而异，因此在一种模式中创建的SQL语句在另一种模式中也许不能运行。较早的ANSI-89标准与ANSI-92相比是有局限的，因此对于新的数据库可以选择ANSI-92语法，更多信息请参见3.2节。

DBMS 如果使用Access作为前端去查询Microsoft SQL Server数据库，必须使用ANSI-92语法。
如果使用Access 97或先前的版本，则必须使用ANSI-89。

⇒ 为数据库开启ANSI-92 SQL语法

- (1) 在Access中，如果有必要先打开数据库。
- (2) 在Access 2003或之前的版本中，选择Tools→Options→Tables/Queries选项卡；在Access 2007或之后的版本中，单击Microsoft Office按钮，选择Access Options→Object Designers（在左侧窗格中）。
- (3) 在“SQL Server Compatible Syntax (ANSI 92)”的下面，选择“This Database”（图1-1）。

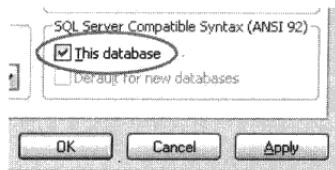


图1-1 对于打开的数据库，勾选“This database”复选框将开启ANSI-92 SQL语法模式

- (4) 单击OK按钮，Access关闭、压缩，然后重新启动数据库，新的设置生效。可以看到取决于安全设置的某些警告。

如果是偶尔使用Access的用户，可以利用查询设计网格（Query Design Grid）创建查询。在设计视图（Design View）中创建了查询后，Access会在后台创建等价的SQL语句。用户可以在SQL视图中查看、编辑和运行SQL语句。

⇒ 在Access 2000、2002或2003中运行SQL语句

- (1) 打开数据库，或按F11键切换到数据库窗口来打开数据库。
- (2) 在数据库窗口中，单击Queries（在Objects下面），然后单击工具栏上的New（图1-2）。
- (3) 在New Query对话框中，单击Design View，然后单击OK按钮（图1-3）。

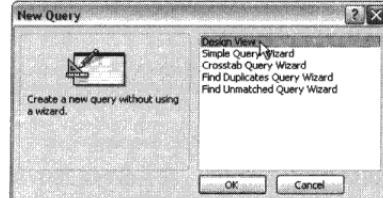
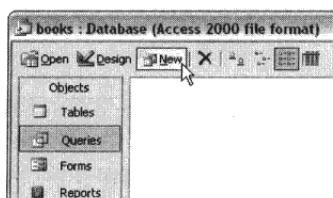


图1-2 在工具栏上单击New按钮，创建一个新的查询

图1-3 选择Design View，跳过帮助向导

- (4) 无需添加表或查询，在Show Table对话框中单击Close按钮（图1-4）。

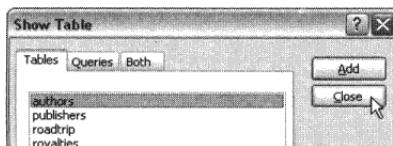


图1-4 因为SQL语句指定了表，因此不需要以图形化方式添加表

(5) 选择View→SQL View (图1-5)。

(6) 输入或粘贴SQL语句 (图1-6)。

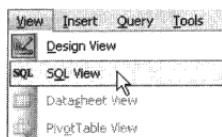


图1-5 SQL View隐藏了图形化查询网格，改用能输入或粘贴SQL语句的文本编辑器

```
Query1 : Select Query
SELECT au_fName, au_lName
FROM authors
ORDER BY au_lName;
```

图1-6 输入SQL语句

(7) 要运行SQL语句，在工具栏上单击或选择Query→Run (图1-7)。Access显示SELECT语句的结果 (图1-8)，但阻止或执行其他类型的SQL语句，有无警告信息取决于设置。

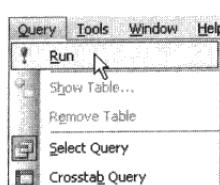


图1-7 运行这条语句

au_fName	au_lName
Sarah	Buchman
Wendy	Heydemark
Klee	Hull
Hallie	Hull
Christian	Kells
	Kellsey
Paddy	OFurniture
*	

图1-8 Access显示运行SELECT语句的结果

⇒ 在Access 2007中运行SQL语句

(1) 打开数据库。

(2) 在功能区中，选择Create选项卡，然后选择Other组→Query Design (图1-9)。

(3) 无需添加表或查询，在Show Table对话框中单击Close按钮 (图1-10)。



图1-9 选择Query Design跳过帮助向导

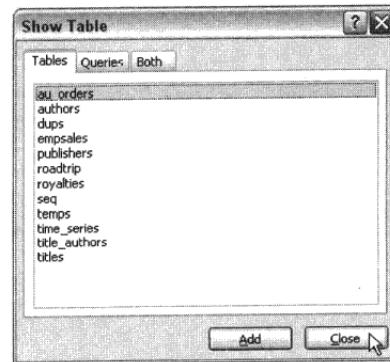


图1-10 因为SQL语句指定了表，因此不需要以图形化方式添加表

(4) 在功能区中，选择Design选项卡，然后选择Results组→SQL View (图1-11)。

(5) 输入或粘贴SQL语句 (图1-12)。

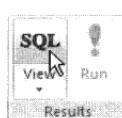


图1-11 SQL View隐藏了图形化查询网格，改用能输入或粘贴SQL语句的文本编辑器

```
Query1
SELECT au_fname, au_lname
FROM authors
ORDER BY au_lname;
```

图1-12 输入SQL语句

(6) 在功能区中，选择Design选项卡，然后选择Results组→Run（图1-13）。Access显示SELECT语句的运行结果（图1-14），但阻止或执行其他类型的SQL语句，有无警告信息取决于设置。

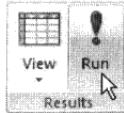


图1-13 运行这条语句

au_fname	au_lname
Carolyn	Buchman
Wendy	Heydemark
Klee	Hull
Hallie	Hull
Christian	Kells
Paddy	Kellsey
*	O'Furniture

图1-14 Access显示运行SELECT语句的结果

✓ 提示

- 通过Access查询对象只能运行一条SQL语句。要运行多条语句，可以使用多个查询对象或者宿主语言（如Visual Basic或C#）。
- 要在Access 2007或后续版本中显示查询列表，按F11键显示导航窗格（在左侧），单击窗格顶端的菜单并选择Object Type，然后再次单击菜单并选择Queries（图1-15）。

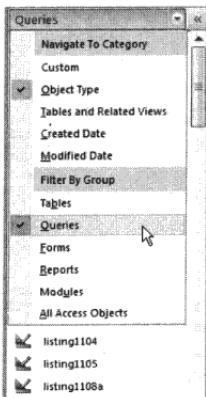


图1-15 Access 2007中新出现的导航窗格替代了之前版本中的数据库窗口（见图1-2）

1.3 Microsoft SQL Server

Microsoft SQL Server是支持巨型数据库和大量事务的商业DBMS。它只能运行在Microsoft的

Windows操作系统上，并且复杂到需要全职的数据库管理人员（Database Administrator, DBA）来进行和维护。

可以在www.microsoft.com/sql上学习SQL Server产品并下载180天免费试用版或者（永久）免费的SQL Server精简版。

本书主要讲解了Microsoft SQL Server 2008，但也包括对于2000和2005的相关提示。要确定正在运行的Microsoft SQL Server的版本，运行SQL Server命令行命令osql -E -Q "SELECT @@VERSION;" 或者运行查询SELECT SERVERPROPERTY('ProductVersion');或SELECT @@VERSION;。

SQL Server 2000、2005和2008

如果读者正在从SQL Server 2000升级到2005或2008，那么需要了解下面关于运行SQL程序的内容。

- SQL Server 2005和后续的版本支持某些2000不支持的标准的SQL特性（例如第9章讲述的EXCEPT和INTERSECT操作符），但本书中的大多数例子可以同样运行在2000、2005和2008中。
如果某个例子不能在所有版本中运行，请查阅DBMS提示。
- SQL Server 2005或2008的SQL Server Management Studio中的查询编辑器（Query Editor）替代了2000中的SQL查询分析器（Query Analyzer）。
- SQL Server 2005或2008的sqlcmd命令行工具替代了2000的osql。sqlcmd工具和osql有很多相同的选项（由于向下兼容，osql在2005或2008中仍然可用），运行sqlcmd -?可以显示语法摘要。
- SQL Server精简版是SQL Server 2005或2008的一个免费、易用的轻量版。SQL Server Management Studio Express是配套的图形化管理工具，可以单独下载或者和SQL Server精简版一起下载。

10

✓ 提示

- 可以使用SET ANSI_DEFAULTS ON选项让SQL Server更加符合标准的SQL。

1.3.1 SQL Server 2000

要在SQL Server 2000中运行SQL程序，可以使用SQL查询分析器图形化工具或者osql命令行工具。

⇒ 使用SQL查询分析器

- (1) 在Windows桌面上，选择Start→All Programs→Microsoft SQL Server→Query Analyzer。
- (2) 在Connect to SQL Server对话框中，选择Server（服务器）和Authentication Mode（验证模式），然后单击OK按钮。
- (3) 在靠近窗口顶部边缘工具栏的下拉列表中，选择一个数据库（图1-16）。
- (4) 要以交互方式运行SQL，可以在查询窗口中输入或粘贴SQL语句；或者运行SQL脚本，选择File→Open（或者按Ctrl+Shift+P快捷键），定位并选择脚本文件，单击Open按钮。
- (5) 选择Query→Execute（或者按F5键），SQL查询分析器在底部窗格中显示结果（图1-17）。

11

✓ 提示

- 可以在命令行提示符下运行isqlw，启动SQL查询分析器。

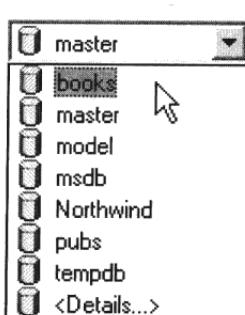


图1-16 SQL查询分析器使用被选择的数据库作为执行SQL语句的目标数据库

	au_fname	au_lname
1	Sarah	Buchman
2	Wendy	Heydemark
3	Hallie	Hull
4	Klee	Hull
5	Christian	Kells
6		Kellsey
7	Paddy	O'Furniture

图1-17 在SQL查询分析器中运行SELECT语句的结果

⇒ 以交互模式使用osql命令行工具

(1) 在命令行提示符下，输入：

```
osql -E -d dbname
```

这里-E选项表示SQL Server采用信任连接将不再要求密码，*dbname*是要使用的数据库的名字。

(2) 输入SQL语句。语句可以跨越多行，以分号(;)结束，然后按回车键。

(3) 输入go，再按回车键，显示结果(图1-18)。

⇒ 以脚本模式使用osql命令行工具

(1) 在命令行提示符下，输入：

```
osql -E -d dbname -n -i sql_script
```

这里-E选项表示SQL Server采用信任连接将不再要求密码，*dbname*是要使用的数据库的名字，-n选项表示在输出中禁用编号和提示符(>)。*sql_script*是包含SQL语句的文本文件，可以使用绝对或相对路径名。

12

(2) 按回车键，显示结果(图1-19)。

```
C:\scripts>osql -E -d books
1> SELECT au_fname, au_lname
2> FROM authors
3> ORDER BY au_lname;
4> go
au_fname      au_lname
Sarah        Buchman
Wendy        Heydemark
Hallie       Hull
Klee         Hull
Christian   Kells
Paddy        O'Furniture
(7 rows affected)
1>
```

图1-18 在osql交互模式中同样的SELECT语句

```
C:\scripts>osql -E -d books -n -i listing0101.sql
au_fname      au_lname
Sarah        Buchman
Wendy        Heydemark
Hallie       Hull
Klee         Hull
Christian   Kells
Paddy        O'Furniture
(7 rows affected)
C:\scripts>
```

图1-19 在osql脚本模式中同样的SELECT语句

⇒ 退出osql命令行工具

输入exit或quit后，按回车键。

⇒ 显示osql命令行选项

在命令行提示符下输入osql -?后，按回车键。

✓ 提示

- 当SQL Server要求指定用户名和密码而不是使用信任连接时，用-U *login_id*替换-E选项，*login_id*是用户名。osql将提示输入密码。
- 如果SQL Server运行在远程的网络计算机上，添加选项-S *server*以指定需要连接的SQL Server实例。向数据库管理员询问连接参数（当SQL Server运行在个人计算机上，而不是别处的某个服务器上时，-S选项也可以用于本地连接）。

13

1.3.2 SQL Server 2005/2008

如果在SQL Server 2005和2008中运行SQL程序，可以使用SQL Server Management Studio的图形化工具或者sqlcmd命令行工具。

⇒ 使用SQL Server Management Studio

(1) 在Windows桌面上，选择Start→All Programs→Microsoft SQL Server→SQL Server Management Studio。在SQL Server精简版中，程序被称作SQL Server Management Studio Express。

(2) 在Connect to Server对话框中，选择服务器和验证模式，然后单击Connect按钮。

(3) 在Object Explorer（左侧窗格）中，展开正在使用的服务器的数据库文件夹，然后选择数据库（图1-20）。如果Object Explorer不可见，选择View→Object Explorer（或按F8键）。

(4) 以交互模式运行SQL，单击工具栏上的New Query，或在Object Explorer中的数据库上右击，选择New Query，在右侧窗格的空白选项卡中输入或粘贴SQL语句。或者运行一个SQL脚本，选择File→Open→File（或按Ctrl+O快捷键），定位并选择脚本文件，然后单击Open按钮，文件内容将显示在右侧窗格的新选项卡中。

(5) 单击工具栏上的Execute，或者选择Query→Execute（或按F5键），SQL Server将在底部的窗格中显示结果（图1-21）。

14

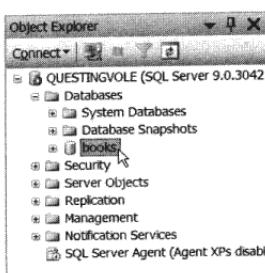


图1-20 在SQL Server Management Studio中使用选择的数据库作为执行SQL语句的目标数据库

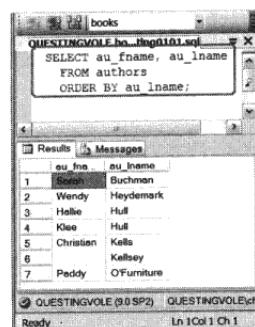


图1-21 在SQL Server Management Studio中运行SELECT语句的结果

⇒ 以交互模式使用sqlcmd命令行工具

(1) 在命令行提示符下, 输入:

```
sqlcmd -d dbname
```

*dbname*是要使用的数据库的名字。

(2) 输入SQL语句。语句可以跨过多行, 用分号(;)结束, 并按回车键。

(3) 输入`go`, 再按回车键显示结果(图1-22)。

⇒ 以脚本模式使用sqlcmd命令行工具

(1) 在命令行提示符下, 输入:

```
sqlcmd -d dbname -i sql_script
```

*dbname*是要使用的数据库的名字, *sql_script*是使用绝对或相对路径名、由SQL语句组成的文本文件。

(2) 按回车键显示结果(图1-23)。

```
C:\scripts>sqlcmd -d books
1> SELECT au_fname, au_lname
2> FROM authors
3> ORDER BY au_lname;
4> go
au_fname      au_lname
-----
Sarah          Buchman
Wendy          Heydemark
Hallie         Hull
Klee           Hull
Christian     Kells
              Kellsey
Paddy          O'Furniture
(7 rows affected)
1>
```

```
C:\scripts>sqlcmd -d books -i listing0101.sql
au_fname      au_lname
-----
Sarah          Buchman
Wendy          Heydemark
Hallie         Hull
Klee           Hull
Christian     Kells
              Kellsey
Paddy          O'Furniture
(7 rows affected)
C:\scripts>
```

图1-22 在sqlcmd交互模式下同样的SELECT语句

图1-23 在sqlcmd脚本模式下同样的SELECT语句

⇒ 退出sqlcmd命令行工具

输入`exit`或`quit`后, 按回车键。

⇒ 显示sqlcmd命令行选项

在命令行提示符下输入`sqlcmd -?`后, 按回车键。

✓ 提示

- `sqlcmd`默认采用信任连接。如果要指定用户名和密码, 添加选项`-U login_id`, *login_id*是用户名, `sqlcmd`将提示输入密码。
- 如果SQL Server运行在远程的网络计算机上, 添加选项`-S server`指定需要连接到的SQL Server实例。向数据库管理员询问连接参数(当SQL Server运行在个人计算机上, 而不是别处的某个服务器上时, `-S`选项也可以用于本地连接)。

15

16

1.4 Oracle

Oracle是支持巨型数据库和大量事务的商业DBMS。它可以运行在多种操作系统和硬件平台上,

非常复杂，需要全职的数据库管理员（DBA）来运行和维护。

可以在www.oracle.com上学习Oracle产品并下载免费的Oracle入门版本——Oracle精简版（XE）。文档在www.oracle.com/technology/documentation上。

本书主要针对Oracle 11g，也包括对于10g、9i和8i的提示。所运行的Oracle版本显示在登录SQL*Plus时最先出现的“Connected to”消息里（或者运行查询SELECT banner FROM v\$version;）。

要运行SQL程序，可以使用SQL*Plus（sqlplus）命令行工具。

✓ 提示

要在Windows中打开命令行提示符，选择Start→All Programs→Accessories→Command Prompt。

⇒ 以交互模式使用sqlplus命令行工具

(1) 在命令行提示符下，输入：

```
sqlplus user/password@dbname
```

*user*是Oracle用户名，*password*是密码，*dbname*是要连接的数据库的名字。为了安全起见，可以省略密码改为输入：

```
sqlplus user@dbname
```

SQL*Plus将提示输入密码。

(2) 输入SQL语句。语句可以跨多行，用分号（;）结束，并按回车键显示结果（图1-24）。

```
C:\scripts>sqlplus system@books
SQL*Plus: Release 11.1.0.6.0 - Production on Tue Feb 5
Copyright (c) 1982, 2007, Oracle. All rights reserved
Enter password:
Connected to:
Oracle Database 11g Release 11.1.0.6.0 - Production
SQL> SELECT au_fname, au_lname
  2  FROM authors
  3  ORDER BY au_lname;
AU_FNAME          AU_LNAME
-----          -----
Sarah            Buchman
Wendy            Heydemark
Hallie           Hull
Klee              Hull
Christian        Kelis
                  Kellsey
Paddy             O'Furniture
7 rows selected.
SQL>
```

图1-24 在sqlplus交互模式下运行SELECT语句的结果

⇒ 以脚本模式使用sqlplus命令行工具

在命令行提示符下，输入：

```
sqlplus user/password@dbname @sql_script
```

*user*是Oracle用户名, *password*是密码, *dbname*是要连接的数据库的名字, *sql_script*是使用绝对或相对路径名、由SQL语句组成的文本文件。为了安全起见, 可以省略密码, 改为输入:

`sqlplus user@dbname @sql_script`

18

SQL*Plus将提示输入密码(图1-25)。

⇒ 退出sqlplus命令行工具

输入exit或quit后, 按回车键。

⇒ 显示sqlplus命令行选项

在命令提示符下, 输入`sqlplus -H`后, 按回车键。这条命令会快速显示多页内容。如果想一页一页地浏览, 输入`sqlplus -H | more`, 然后按回车键。敲击空格键将显示下一页(图1-26)。

```
C:\scripts>sqlplus system@books @listing0101.sql
SQL*Plus: Release 11.1.0.6.0 - Production on Tue Feb 5
Copyright (c) 1982, 2007, Oracle. All rights reserved.
Enter password:
Connected to:
Oracle Database 11g Release 11.1.0.6.0 - Production

AU_FNAME      AU_LNAME
-----
Sarah          Buchman
Wendy          Heydemark
Hallie         Hull
Klee           Hull
Christian     Kells
              Kellsey
Paddy          O'Furniture

7 rows selected.

SQL>
```

图1-25 在sqlplus脚本模式下运行同样的SELECT语句

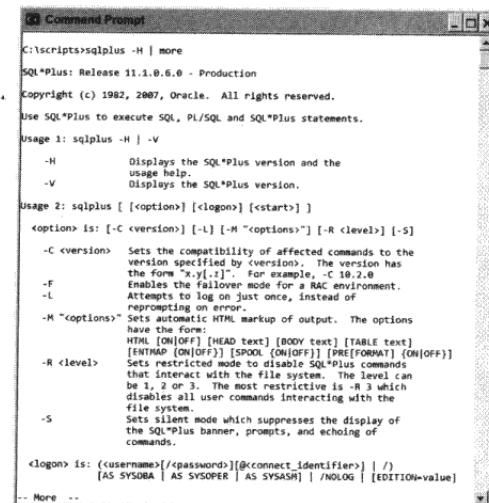


图1-26 sqlplus的帮助屏幕

✓ 提示

- 如果是在本地运行Oracle, 可以使用用户名system和创建数据库时指定的密码:

`sqlplus system@dbname`

如果要连接远程的Oracle数据库, 向数据库管理员询问连接参数。

- 另一种在Windows中打开SQL*Plus的方法是: 选择Start→All Programs→Oracle→Application Development→SQL Plus。

19

1.5 IBM DB2

IBM DB2是支持巨型数据库和大量事务的商业DBMS。它可以运行在多种操作系统和硬件平台上。它非常复杂, 需要全职的数据库管理员(DBA)来运行和维护。

可以在www.ibm.com/db2上学习DB2产品并下载免费90天的DB2版本或者(永久)免费的IBM DB2精简版。

本书主要讲解了DB2 9.5，也包括了对于8.0及之后版本的提示。要确定所运行的DB2的版本，运行DB2命令行命令db2level，或者运行查询SELECT service_level FROM SYSIBMADM.ENV_INST_INFO；

要运行SQL程序，在DB2 8.x中使用图形化工具命令中心（Command Center），也可以在DB2 9.x中使用图形化工具命令编辑器（Command Editor），或者db2命令行处理器（Command-line Processor，CLP）。

⇒ 使用Command Center (DB2 8.x)

(1) 启动Command Center。

这个过程因平台而异。例如在Windows中，选择Start→All Programs→IBM DB2→Command Line Tools→Command Center；在Unix和Linux中，使用命令db2cctr。如果Control Center是打开的，也可以选择Tools→Command Center或者单击图。

(2) 选择命令类型SQL语句和DB2 CLP的命令。

(3) 单击Interactive选项卡。

(4) 在Database Connection对话框中，单击图展开对象树，找到数据库文件夹，并选择一个数据库，然后单击OK按钮。

(5) 以交互模式运行SQL，在Command对话框中输入或粘贴SQL语句（图1-27）。或者以脚本模式运行SQL，单击Scripts选项卡，选择Script→Import，定位并选择脚本文件，然后单击OK按钮。

(6) 单击图或按Ctrl+回车快捷键，或者选择Interactive→Execute或Script→Execute。

交互式SQL的结果显示在Results选项卡上（图1-28），SQL脚本的结果显示在Script框下面的框内。

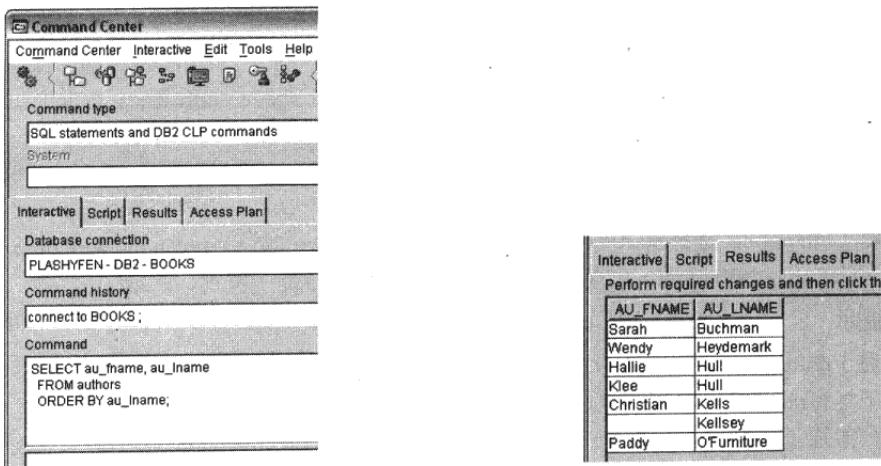


图1-27 在Command Center中准备运行的SELECT语句

图1-28 运行SELECT语句的结果

⇒ 使用Command Editor (DB2 9.x)

(1) 启动Command Editor。

这个过程因平台而异。例如在Windows中，选择Start→All Programs→IBM DB2→[db2_copy_name]→Command Line Tools→Command Editor；在Unix和Linux中，使用命令db2ce。如果Control Center是打开的，也可以选择Tools→Command Editor或者单击图。

(2) 在Commands选项卡的Target列表里选择一个数据库。如果需要的数据库不在Target列表中，单

击Add按钮（在列表右侧），选择一个数据库，然后单击OK按钮。

(3) 以交互模式运行SQL，在Commands选项卡的顶部框中输入或粘贴SQL语句（图1-29）。或者以脚本模式运行SQL，选择Selected→Open（或者按Ctrl+O快捷键），定位并选择脚本文件，然后单击OK按钮。

(4) 单击 或按Ctrl+回车快捷键，或者选择Selected→Execute。

Command Editor将在Query Results选项卡上显示结果（图1-30），Commands选项卡的底部框内将显示查询处理信息。

22

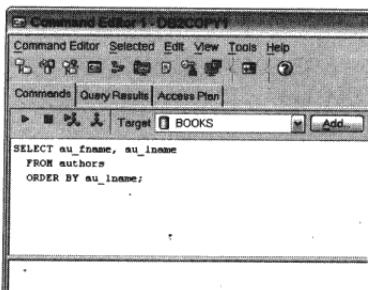


图1-29 在Command编辑器中准备运行的SELECT语句

Edits to these results are performed as search	
AU_FNAME	AU_LNAME
Sarah	Buchman
Wendy	Heydemark
Hellie	Hull
Klee	Hull
Christian	Kells
	Kellsey
Paddy	O'Furniture

图1-30 运行SELECT语句的结果

由于涉及父子进程的技术原因，只有Microsoft Windows用户必须使用一个专门的准备步骤去启动db2命令行处理器。

⇒ 在Windows中启动db2命令行处理器

□ 在命令行提示符下，输入db2cmd然后按回车键。

或者，对于DB2 8.x，选择Start→All Programs→IBM DB2→Command Line Tools→Command Window。

对于DB2 9.x，选择Start→All Programs →IBM DB2→[db2_copy_name]→Command Line Tools →Command Window。出现一个新的DB2 CLP命令行提示符窗口。

✓ 提示

□ 对于db2命令（随后介绍），必须使用DB2 CLP窗口。如果试图在正常的Windows命令行提示符下运行db2，DB2会给出“Command line environment not initialized.”错误信息。

□ 如果通过db2cmd命令打开一个新的DB2 CLP窗口，就可以关闭最初的命令行提示符窗口。

□ 在DB2 CLP窗口中，如果有必要，可以在运行db2命令前改变(cd)工作目录。

⇒ 以交互模式使用db2命令行处理器

(1) 在命令行提示符下，输入：

db2 -t

然后按回车键。-t选项表示db2以分号(;)结束语句。

db2=>提示符出现。

(2) 在db2提示符下，输入：

connect to dbname;

23

然后按回车键。*dbname*是要使用的数据库名。

(3) 输入SQL语句。语句可以跨多行，用分号结束，然后按回车键显示结果（图1-31）。

✓ 提示

□ 作为可选方式，可以直接在命令行中输入命令和SQL语句来避开db2=>提示符，例如：

```
db2 connect to books
db2 SELECT * FROM authors
```

□ 这里如果省略-t选项，则不再需要用分号来结束命令和SQL语句。

⇒ 以脚本模式使用db2命令行处理器

(1) 在命令行提示符下，输入：

```
db2 connect to dbname
```

*dbname*是要使用的数据库的名字。

(2) 在命令行提示符下，输入：

```
db2 -t -f sql_script
```

*sql_script*是使用绝对或相对路径名、由SQL语句组成的文本文件。-t选项表示db2以分号(;)结束语句。

(3) 按回车键显示结果（图1-32）。

```
DB2 CLP - DB2COPY1 - db2 >
db2 => connect to books;
Database Connection Information
Database server      = DB2/NT 9.5.0
SQL authorization ID = CHRIS
Local database alias = BOOKS
db2 => SELECT au_fname, au_lname
db2 (cont.) => FROM authors
db2 (cont.) => ORDER BY au_lname;
AU_FNAME          AU_LNAME
-----
Sarah             Buchman
Mandy            Heydemark
Hallie           Hull
Klee              Hull
Christian        Kells
Paddy            Kellsey
                  O'Furniture
7 record(s) selected.
db2 =>
```

图1-31 在db2交互模式下同样的SELECT语句

```
DB2 CLP - DB2COPY1
C:\scripts> db2 connect to books
Database Connection Information
Database server      = DB2/NT 9.5.0
SQL authorization ID = CHRIS
Local database alias = BOOKS
C:\scripts> db2 -t -f listing0101.sql
AU_FNAME          AU_LNAME
-----
Sarah             Buchman
Mandy            Heydemark
Hallie           Hull
Klee              Hull
Christian        Kells
Paddy            Kellsey
                  O'Furniture
7 record(s) selected.
C:\scripts>
```

图1-32 在db2脚本模式下运行同样的SELECT语句

✓ 提示

□ 在第(2)步中，添加选项-v可以在输出中显示*sql_script*的内容。

□ 另一种可选的脚本工具是db2batch。

⇒ 退出db2命令行工具

在db2提示符下，输入*quit;*，并按回车键（如果启动db2时不使用-t选项，则可以省略分号）。

⇒ 显示db2命令行选项

在命令行提示符下，输入db2?，并按回车键。

这条命令会快速显示多页内容。如果想一页一页地浏览，改为输入db2 ? | more后按回车键。按空格键将显示下一页（图1-33）。

✓ 提示

- 在db2 =>提示符下输入?;后按回车键，可以得到帮助信息（如启动db2时不使用-t选项，则省略分号）。
- Linux、Unix和Windows（即IBM所说的“LUW”）平台上的DB2和其他平台上的略微不同。本节讲解了LUW DB2，如果在非LUW平台上运行DB2（如z/OS或OS/390），命令可能会不同。

26

1.6 MySQL

MySQL是支持大型数据库和大量事务的开源DBMS。MySQL以它的高速和易用著称。它对于个人使用是免费的，并且可以运行在多种操作系统和硬件平台上。可以在www.mysql.com上下载MySQL。

本书主要讲解了MySQL 5.1，也包括对4.0之后版本的提示。

要确定所运行的MySQL版本，运行MySQL命令行命令mysql -v或运行查询SELECT VERSION();。要运行SQL程序，可以使用mysql命令行工具。

✓ 提示

- 要在Windows中打开命令行提示符，选择Start→All Programs→Accessories→Command Prompt。

⇒ 以交互模式使用mysql命令行工具

(1) 在命令行提示符下，输入：

```
mysql -h host -u user -p dbname
```

*host*是主机名，*user*是MySQL用户名，*dbname*是要使用的数据库的名字。对于省略-p选项，或在密码提示时按回车键而未输入密码的用户，MySQL将提示输入密码。

(2) 输入SQL语句。语句可以跨过多行，用分号(;)结束，然后按回车键显示结果（图1-34）。

⇒ 以脚本模式使用mysql命令行工具

(1) 在命令行提示符下，输入：

```
mysql -h host -u user -p -t dbname < sql_script
```

*host*是主机名，*user*是MySQL用户名，*dbname*是要使用的数据库的名字。对于省略-p选项，或在密码提示时按回车键而

```
C:\scripts>db2 ? | more
db2 [option ...] {db2-command | sql-statement | 
option: -a, -c, -d,-e{cls}, -f{file}, -i, -ihistfile, -o, -m, -n, 
db2-command | sql-statement | 
ACTIVATE DATABASE          GET CONTACTS           RECOVER
ADD CONTACT                GET/UPDATE DB CFG   REDISTRIBUTE DB PARTITION
ADD CONTACTGROUP          GET/UPDATE DSN CFG  REFRESH LOAD
ADD COLUMNS MANAGER        GET CURRENT SWITCHES REGISTER KSCOBJECT
ADD DEPARTITIONNUM         GET DESCRIPTION FOR HEALTH REGISTER XSCOBJECT
ADD XMLSCHEMA               GET NOTIFICATION LIST REORG INDEXES/TABLE
ARINIEVE LOG                GET HEALTH SNAPSHOT REORG INDEXES/TABLE
ATTACH                      GET INSTANCE          REORGCHN
AUTOMATE/UNCONFIGURE      GET INSTANCE FOR SWITCHES RESET ADMIN CFG
BACKUP DATABASE             GET RECOMMENDATIONS RESET ALERT CFG
BIND                         GET ROUTINE          RESET DB CFG
CATALOG APPLIC NODE        GET SNAPSHOT          RESET DBM CFG
CATALOG APPN NODE          HELP                  RESET MONITOR
CATALOG DATABASE            HISTORY              RESTORE DATABASE
CATALOG DCS DATABASE       IMPORT               RENAME TAPE
CATALOG LDAP DATABASE      INITIALIZE TAPE    ROLLFORWARD DATABASE
CATALOG LDAP NODE          INSPECT              RUNSTATS
CATALOG LOCAL NODE          LIST ACTIVE DATABASES
CATALOG MIGRUE NODE         LIST APPLICATIONS   RUNSTATS
CATALOG NETEIOS NODE        LIST COMMAND OPTIONS SET CLIENT
CATALOG ODBC DATA SOURCE   LIST DATABASE DIRECTORY SET RUNTIME DEGREE
CATALOG TCPIP NODE          LIST DB PARTITION GROUPS SET SPACESPACE CONTAINERS
CHANGE AUTOCOMMIT LEVEL    LIST DATASOURCES MANAGERS SET TABLE POSITION
COMPLETE XMLSCHEMA          LIST DCS PARTITIONS SET WRITE_PRIORITY
CREATE DATABASE             LIST DCS DIRECTORY START DATABASE MANAGER
CREATE TABLES CATALOG     LIST DRDA INDOUBT  START HADR
DECOMPOSE XML DOCUMENT    LIST INDODUCTION TRANSACTIONS STOP HADR
DREGISTER                   LIST NODE DIRECTORY TAKEOVER HADR
DESCRIBE                    LIST ODBC DATA SOURCES TERMINATE
DETACH                      LIST PACKAGES/TABLES UNCATALOG DATABASE
DROP CONTACT                LIST TABLESPACE CONTAINERS UNCATALOG DCS DATABASE
DROP CONTACTGROUP          LIST TABLESPACES     UNCATALOG LDAP DATABASE
-- More --
```

图1-33 db2的帮助屏幕

```
C:\scripts>mysql -h localhost -u root -p books
Enter password:
Welcome to the MySQL monitor. Commands end with;
Your MySQL connection id is 24
Server version: 5.1.23-rc-community MySQL Commun
Type 'help;' or '\h' for help. Type '\c' to clea
mysql> SELECT au_fname, au_lname
-> FROM authors
-> ORDER BY au_lname;
+-----+-----+
| au_fname | au_lname |
+-----+-----+
| Sarah    | Buchanan
| Wendy   | Heydemark
| Hallie  | Hull
| Klee    | Hull
| Christian | Kells
| Paddy   | Kellsey
|          | O'Furniture
+-----+-----+
7 rows in set (0.00 sec)
```

图1-34 在mysql交互模式下运行同样的SELECT语句的结果

27

未输入密码的用户，MySQL将提示输入密码。-t选项表示以表格的形式显示结果，如果需要制表符定界的输出，则可以省略此选项。*dbname*是要使用的数据库的名字。<重定向操作符表示从*sql_script*文件中读取，*sql_script*是使用绝对或相对路径名的、由SQL语句组成的文本文件。

(2) 按回车键显示结果(图1-35)。

au_fname	au_lname
Sarah	Buchman
Wendy	Heydemark
Hallie	Hull
Klee	Hull
Christian	Kells
	Kellsey
Paddy	O'Furniture

图1-35 在mysql脚本模式下运行同样的SELECT语句

⇒ 退出mysql命令行工具

输入quit或\q并按回车键。

⇒ 显示mysql命令行选项

在命令行提示符下，输入mysql -?，并按回车键。

这条命令会快速显示多页内容。如果想一页一页地浏览，改为输入mysql -? | more后，按回车键。按空格键将显示下一页(图1-36)。

✓ 提示

- 如果MySQL运行在远程的网络计算机上，可向数据库管理员询问连接参数。如果在本地(也就是自己的个人计算机上)运行MySQL，那么设置host为localhost，user为root，并使用安装MySQL时分配给root的密码。
- 作为命令行提示符的替代，可以使用www.mysql.com/products/tools上提供的图形化工具。
- 可以在www.opensource.org上获取更多的关于开源软件的信息。

1.7 PostgreSQL

PostgreSQL是支持大型数据库和大量事务的开源DBMS。PostgreSQL以它丰富的功能集和与标准SQL很高的一致性而著称。它是免费的，并且可以运行在多种操作系统和硬件平台上。可以从www.postgresql.org下载PostgreSQL。

图1-36 mysql的帮助屏幕

本书主要讲解了PostgreSQL 8.3，也包括了对7.1之后版本的提示。要确定所运行的PostgreSQL版本，运行PostgreSQL命令行命令`psql -v`，或运行查询`SELECT VERSION();`。

要运行SQL程序，可以使用`psql`命令行工具。

✓ 提示

- 要在Windows中打开命令行提示符，选择Start→All Programs→Accessories→Command Prompt。

⇒ 以交互模式使用`psql`命令行工具

- (1) 在命令行提示符下，输入：

```
psql -h host -U user -W dbname
```

`host`是主机名，`user`是PostgreSQL的用户名，`dbname`是要使用的数据库的名字。对于省略`-W`选项，或在密码提示时按回车键而未输入密码的用户，PostgreSQL将提示输入密码。

30

- (2) 输入SQL语句。语句可以跨过多行，用分号(;)结束，然后按回车键显示结果(图1-37)。

⇒ 以脚本模式使用`psql`命令行工具

- (1) 在命令行提示符下，输入：

```
psql -h host -U user -W -f sql_script dbname
```

`host`是主机名，`user`是PostgreSQL用户名，`dbname`是要使用的数据库的名字。对于省略`-W`选项，或在密码提示时按回车键而未输入密码的用户，PostgreSQL将提示输入密码。`-f`选项指定SQL文件`sql_script`的名称使用绝对或相对路径名、由SQL语句组成的文本文件。`dbname`是要使用的数据库的名称。

- (2) 按回车键显示结果(图1-38)。

```
C:\scripts> psql -h localhost -U postgres -W books
Password for user postgres:
Welcome to psql 8.3.0, the PostgreSQL interactive terminal.

Type: \copyright for distribution terms
      \h for help with SQL commands
      \? for help with psql commands
      \g or terminate with semicolon to execute query
      \q to quit

Warning: Console code page (437) differs from Windows
8-bit characters might not work correctly.
page "Notes for Windows users" for details

books=# SELECT au_fname, au_lname
books-#   FROM authors
books-#   ORDER BY au_lname;
au_fname | au_lname
-----+-----
Sarah    | Buchman
Wendy   | Heydemark
Hallie   | Hull
Klee     | Hull
Christian | Kells
          | Kellsey
Paddy    | O'Furniture
(7 rows)

books=#
```

图1-37 在`psql`交互模式下运行同样的SELECT语句的结果

```
C:\scripts> psql -h localhost -U postgres -W -f listing0101.sql books
Password for user postgres:
au_fname | au_lname
-----+-----
Sarah    | Buchman
Wendy   | Heydemark
Hallie   | Hull
Klee     | Hull
Christian | Kells
          | Kellsey
Paddy    | O'Furniture
(7 rows)

C:\scripts>
```

图1-38 在`psql`脚本模式下运行同样的SELECT语句

⇒ 退出psql命令行工具

输入`\q`并按回车键。

⇒ 显示psql命令行选项

在命令行提示符下，输入`psql -?`并按回车键。

这条命令会快速显示多页内容。如果想一页一页地浏览；改为输入`psql -? | more`后，按回车键。敲击空格键将显示下一页（图1-39）。



图1-39 psql的帮助屏幕

✓ 提示

- 如果PostgreSQL运行在远程的网络计算机上，可向数据库管理员询问连接参数。如果在本地（也就是自己的个人计算机上）运行PostgreSQL，那么设置`host`为`localhost`，设置`user`为`postgres`，并使用装配PostgreSQL时分配给`postgres`的密码。
- 可以设定环境变量`PGDATABASE`和`PGUSER`来指定默认的数据库和连接数据库时的用户名，参见PostgreSQL文档中的“环境变量”。
- 作为命令行提示符的替代，可以使用pgAdmin图形化工具。如果PostgreSQL没有自动安装pgAdmin，可以在<http://pgadmin.org>上免费下载。
- 可以在www.opensource.org上获取更多关于开源软件的信息。

关系模型

2

关于数据库设计的好书有很多，但本书并不是讲数据库设计的。然而，想成为一名优秀的SQL程序员，需要熟悉关系模型（relational model）——一种非常简单且适合组织和管理数据的数据模型（图2-1），它在优胜劣汰法则的作用下，战胜了竞争者——网络模型和层次模型。

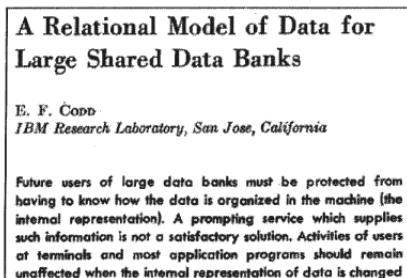


图2-1 可以在www.seas.upenn.edu/~zives/03f/cis550/codd.pdf上读到E.F. Codd的A Relational Model of Data for Large Shared Data Banks (*Communications of the ACM*, Vol. 13, No. 6, June 1970, pp. 377–387)。关系数据库建立在这篇论文定义的数据模型的基础之上

关系模型的基础是集合论（set theory），它让人们以数据集合而不是孤立的数据项或行的方式进行思考。该模型描述了如何在数据库表上运用与数学集合一样的方法执行常见的代数运算（比如并和交）（图2-2）。表（table）类似于集合，即有公共属性的不同元素的集合。例如，数学集合可能包含正整数，然而一个数据库表可能包含关于学生的信息。

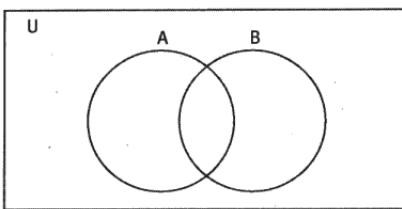


图2-2 依据集合论的基础知识，维恩图（Venn diagram）表示集合的运算结果。矩形（U）表示全域，里面的圆（A和B）表示对象集合。圆的相对位置和交叠显示了集合之间的关系。在关系模型中，圆就是表，矩形是一个数据库中的所有信息

2.1 表、列和行

首先，介绍一些术语。如果已经熟悉数据库，就可能听说过类似概念的其他术语。表2-1所示为这些术语间的关系。Codd的关系模型术语在第一列，标准的SQL和DBMS文档术语在第二列，第三列是传统的非关系文件处理的延续。本书将使用SQL术语（尽管在正式文本中SQL和模型的术语是不可以互换的）。

表2-1 类似的概念

模 型	SQL	文 件
关系	表	文件
属性	列	字段
元组	行	记录

2.1.1 表

从用户的视角来看，数据库是一个或多个表的集合体（并且只有表）。表

- 是存储数据的数据库结构。
- 包含了特定实体类型的数据。实体类型（entity type）是客观存在并且相互区别的有公共属性的对象、事件或者概念的类，如病人、电影、基因、天气状况、发票、工程或预约（病人和预约是不同的实体，因此将关于它们的信息存储在不同的表中）。
- 是由行（row）和列（column）构成的二维网格（图2-3和图2-4）。
- 在每一行和列的交叉点保存了称为值（value）的数据项（参考图2-3和图2-4）。
- 至少有一列且零或多行。一行也没有的表称为空表（empty table）。
- 在数据库里有唯一的名称（严格地讲，是在一个模式里）。

34

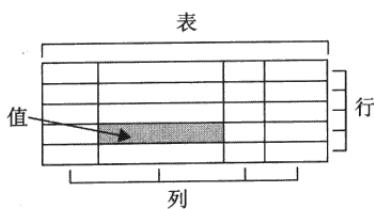


图2-3 这个网格是数据库的基本存储单元的抽象表示

au_id	au_fname	au_lname
A01	Sarah	Buchman
A02	Wendy	Heydemark
A03	Hallie	Hull
A04	Klee	Hull

图2-4 这个网格表示一个实际的（而不是抽象的）表，它通常出现在数据库软件和书中。这个表有3列、4行和 $3 \times 4 = 12$ 个值。最上面的一行不是行，而是显示列名的标题

2.1.2 列

给定表的列有以下特点。

- 每一列表示了表的实体类型的一个特定属性（或特性）。在employees表中，例如名为hire_date的列可能表示雇员被雇用的时间。
- 每一列有一个限制该列允许值范围的域。域（domain）是一组约束，包括限制值的数据类型、长度、格式、范围、唯一性、特定值和可空性（即值能为空或不能为空）。例如，如果hire_date需要有效的日期值，就不能在hire_date列中插入字符串值'jack'，而且也不能插入超出

35

- `hire_date` 范围（从公司成立之日到今天）的日期。可以通过数据类型（见第3章）和约束（见第11章）定义一个域。
- 列中的实体只有单一值（原子值），参见2.5节。
- 列的顺序（从左到右）是不重要的（图2-5）。
- 表中的每一列有唯一标识它的名称（在其他表中可以重新使用同样的列名）。

au_lname	au_id	au_fname
Hull	A04	Klee
Buchman	A01	Sarah
Hull	A03	Hallie
Heydemark	A02	Wendy

图2-5 行和列是无序的，意味着它们在表中的顺序对于表示信息来说不相关。
交换列或行不会改变表的含义，这张表和图2-4中的表传达了同样的信息

2.1.3 行

给定表的行有以下特点。

- 每一行记录了一个实体的事实，它是一个实体类型中的唯一实例，例如一个学生或预约的详细资料。
- 对于表的每一列，每一行都包含一个值或空值。
- 行的顺序（由顶至底）是不重要的（参考图2-5）。
- 在一个表中，没有两行是完全相同的。
- 表中的每一行都由它的主键（primary key）来唯一标识，参见2.2节。

✓ 提示

- 使用SELECT语句检索列和行，见第4章至第9章。使用INSERT、UPDATE和DELETE增加、更新和删除行，见第10章。使用CREATE TABLE、ALTER TABLE和DROP TABLE增加、更新、删除表和列，见第11章。
- 表具有较好的闭包特性，这将确保在一张表上完成任何运算都将得到另一张表（图2-6）。

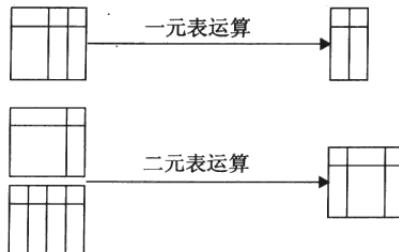


图2-6 闭包确保无论如何分割或合并表都将得到另一个表，这个特性使得可以串联表运算或将其实嵌套。一元（或一目）表运算是对一个表运算，得到一个结果表。二元（或二目）表运算是对两个表运算，得到一个结果表

- DBMS有两种类型的表：用户表和系统表。用户表（user table）存储用户定义的数据。系统表（system table）存储元数据（关于数据库的数据），如结构化信息、物理细节、执行统计和安全设置。全体系统表称为系统目录。DBMS自动连续创建和管理这些表。这个模式与所有数据存储在表中的关系模型规则相符（图2-7）。

Name	Owner	Type
authors	dbo	User
publishers	dbo	User
royalties	dbo	User
title_authors	dbo	User
titles	dbo	User
dtproperties	dbo	System
syscolumns	dbo	System
syscomments	dbo	System
sysdepends	dbo	System
sysfilegroups	dbo	System
sysfiles	dbo	System
sysfiles1	dbo	System
sysforeignkeys	dbo	System
sysfulltextcatalogs	dbo	System
sysfulltextnotify	dbo	System
sysindexes	dbo	System
sysindexkeys	dbo	System
sysmembers	dbo	System
sysobjects	dbo	System
syspermissions	dbo	System
sysproperties	dbo	System
sysprotects	dbo	System
sysreferences	dbo	System
systypes	dbo	System
sysusers	dbo	System

图2-7 DBMS存储系统信息的特定表叫做系统表。这里，有阴影的表是由Microsoft SQL Server对于本书中所使用的示例数据库创建并维护的系统表。可以像存取用户定义的表一样存取系统表，但不要改变它们，除非你特别熟悉

- 在实践中，表中的行数经常改变，但列数却很少改变。数据库的复杂性使得增加或删除列变得困难；列的改变会影响键、引用完整性、权限等。插入或删除行不会影响这些。
- 数据库设计人员根据用户的需求将值分到列中。例如，电话号码可能存放在单一的tel_no列中，也可能被分割后存放在county_code、area_code和subscriber_number列中，这依赖于用户要查询、分析和报告方面的需求。
- 表和电子表格相似是表面的。和电子表格不同的是：表不依赖于行和列的顺序，不能执行运算，不允许自由形态的数据实体，严格检查每个值的有效性，易于和其他表关联。
- SQL标准定义了关系型数据库结构的层次。一个目录包含一个或多个模式（用户拥有的对象和数据的集合）。一个模式包括一个或多个对象：基本表、视图和例程（函数/过程）。
- DBMS DBMS对于同样的概念有时使用其他术语。一个实例（类似于目录）至少包含一个数据库，一个数据库包括一个或多个模式，一个模式包括表、视图、权限、存储过程等。要明确地定义一个对象，在层次结构中每一层的每一项都需要唯一的名称（标识符）。表2-2所示为如何访问对象，参见3.3节。

表2-2 对象引用

平 台	访 问
Standard SQL	<i>catalog.schema.object</i>
Access	<i>database.object</i>
SQL Server	<i>server.database.owner.object</i>
Oracle	<i>schema.object</i>
DB2	<i>schema.object</i>
MySQL	<i>database.object</i>
PostgreSQL	<i>database.schema.object</i>

2.2 主键

数据库中的每个值都必须是可以访问的。值被存储在表中行和列的交叉点上，因此定位一个值时必须指明表、列和行。可以通过唯一的名称标识表或列，但行没有名称，需要一种称为主键（primary key，也叫主码）的不同的标识机制。主键有以下特点。

- 必需的。每个表必需有一个主键。在关系模型中，表被视为无序的行集合。因为这里没有下一行和上一行的概念，不能通过位置标识行。如果没有主键，某些数据将无法访问。
- 唯一的。因为主键标识了表中独立的一行。在一个表中不会有两行具有相同的主键值。
- 简单的或组合的。一个主键可能由一个表的一列或多列构成。只有一列的键称为简单主键（也叫简单关键字），多列的键称为组合键（也叫组合关键字）。
- 非空的。主键的值不能为空。对于组合键，不允许列的值为空，参见3.14节。
- 稳定的。主键的值一经创建，将很少改变。如果一个实体被删除了，其主键的值不能被新的实体再次使用。
- 最小的。主键只包括满足唯一性的必要的列。

学习数据库设计

学习产品数据库的设计，要先阅读一些关系代数、实体-联系（E-R）建模、Codd的关系模型、系统体系结构、空值、完整性和其他重要的概念方面的理论文章，作为基础。Chris Date的*An Introduction to Database Systems*(Addison-Wesley)受人欢迎；还有大量图书可供选择，有一种花钱少的选择是Date的*Database in Depth* (O'Reilly)。对于集合论和逻辑的现代介绍有Lex de Haan和Toon Koppelaars (Apress)的*Applied Mathematics for Database Professionals*，古典介绍则包括Robert Stoll的*Set Theory and Logic* (Dover)和Wilfrid Hodges (Penguin)文雅的*Logic*。可以在网上搜索E. F. Codd、Chris Date、Fabian Pascal和Hugh Darwen的文章。所有这些资料看起来有点多余，但你会对数据库在增加了一些表、约束、触发器和存储过程之后的复杂程度感到吃惊。不要认为理论不实用——同其他所有领域一样，掌握好知识，才能预见结果，并在情况变糟时避免一错再错。

要避开流行的垃圾，比如*Database Design for Dummies/Mere Mortals*。如果依赖它们的指导，就会创建一看便知结果有错的数据库，不能检索到想要的信息，反复输入同样的数据，或者输入的数据不知去向。这些书掩盖（或遗漏）重复性的处理工作（如选择标识名和面试主题专家）的首要原则。

数据库设计人员指明每个表的主键。这个过程至关重要，因为错误选择关键字将造成无法为一个表增加数据（行）。这里将讲述要点，但关于这个主题如果想学习更多内容可以阅读数据库设计的书。

假如需要为图2-8中的表选择主键。列`au_fname`和`au_lname`都不行，因为都违反了唯一性要求。将`au_fname`和`au_lname`组合成组合键也不行，因为两个作者也许同姓同名。因为姓名的不稳定性（人员离婚、公司合并或写法更改），姓名通常是很差的关键字。正确的选择是自定义的唯一标识作者的`au_id`。当自然或显而易见的标识符（如姓名）不能使用时，数据库设计人员将创建唯一标识符。

在定义主键后，DBMS将强制表数据的完整性。不能插入下面的行（因为列`au_id`的值A02已经在表中存在）：

A02 Christian Kells

也不能插入下面的行（因为`au_id`不能为空）：

NULL Christian Kells

下面这行是合法的：

A05 Christian Kells

✓ 提示

- 也可以参见11.6节。
- 实际上，主键经常被放置在表的初始（最左边）列。当一列名含有`id`、`key`、`code`或`num`时，该列可能是主键或主键的一部分（或者外键，将在下一节介绍）。
- 数据库设计人员经常放弃美国居民社会保险号这样的唯一标识符。他们代之以人造的、在用户组织中有深刻含义的内部信息编码的人造关键字。例如，一个雇员的ID也许嵌入了他被雇用的年份。其他原因（如隐私问题）同样推动了人造关键字的使用。
- 数据库设计人员在一个表中也许选择多个唯一候选键（也叫候选关键字），其中之一被指定为主键。在指定以后，其余的候选键成为预备键（也叫预备关键字）。候选键通常有不能为空和唯一性的约束，参见11.8节。
- 可以使用`au_id`和`au_lname`作为组合键，但这个组合违反了最小性要求。对于组合键的例子，参见2.6节中的表`title_authors`。
- **DBMS** DBMS提供了能够为每一行自动创建唯一标识值的数据类型和属性（例如，当插入一个新行时自动增加的整数），参见3.12节。

au_id	au_fname	au_lname
A01	Sarah	Buchman
A02	Wendy	Heydemark
A03	Hallie	Hull
A04	Klee	Hull

图2-8 在这个表中列`au_id`是主键

2.3 外键

不同实体的信息被存储在不同的表中，因此需要一种在表间定位的方法。关系模型提供了一种叫做外键（也叫外码）的机制使表关联。

外键有以下特点。

- 它是表中的一列（或一组列），其值与其他表中的值相关联或引用其他表中的值。
- 它确保表中的行在另一个表中有相对应的行。
- 包含外键的表是引用表或子表，另一个表是被引用表或父表。
- 外键建立了和父表主键（或任意候选键）的直接联系，所以外键的值受限于现有父表的键值，这个约束称为引用完整性。例如，在表`appointments`中的一个特定的行必须和表`patients`中的一行有关联，否则不存在或不能标识的病人也会有预约。孤立的行是子表中那些在父表没有关联的行。在正确设计的数据库中，不可以插入新的孤立的行或删除父表中关联的行，使现有子表中的行成为孤立的行。
- 外键值和父表主键有相同的域。回顾2.1节，域是定义了一列有效值的集合。
- 和主键值不同，外键值可以为空（空白），参阅本节提示。
- 外键可以和它的父表主键有不同的列名。
- 外键值在它们自己的表中通常不是唯一的。
- 明确第一点：实际上，外键能够引用自己的表（而不是其他表）中的主键。例如，带有主键`emp_id`的表`employees`，可以有引用列`emp_id`的外键`boss_id`，这种表类型叫做自引用。

图2-9显示了两个表之间主键和外键的关系。在定义了外键之后，DBMS将强制引用完整性。因为`pub_id`的值`P05`在父表`publishers`中不存在，不可以在子表`titles`中插入下面的行：

T07 I Blame My Mother P05

如果外键接受空值，则可以插入下面的行：

T07 I Blame My Mother NULL

这行是合法的：

T07 I Blame My Mother P03

✓ 提示

- 也可以参见11.7节。
- 当试图去更新或删除外键值指向的父表主键值时，SQL要求指定DBMS采取的引用完整性动作，参见11.7节中的提示。
- 允许外键为空值，使得强制引用完整性变得复杂。实际上，外键值一般只是暂时为空，通常是因为存在待定事项或待定结果，参见3.14节。

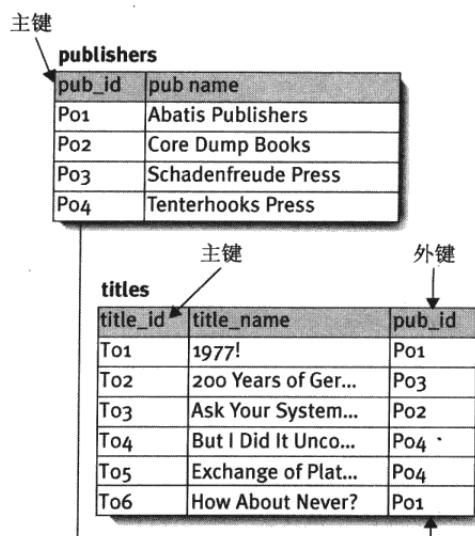


图2-9 表`titles`中的列`pub_id`，是引用表`publishers`中列`pub_id`的外键

2.4 联系

联系（relationship）是两个表公共列之间确定的关联。联系可以是：

- 一对一
- 一对多
- 多对多

2.4.1 一对一

在一对一联系中，表A中的每一行最多和表B有一个相匹配的行，表B中的每一行也最多和表A有一个相匹配的行。尽管将两个表的信息存在一个表中是可行的，但一对一联系通常被用于因安全原因隔离机密信息，分割一个整体表可以加快查询，或避免将空值插入表中（见3.14节）。

当一个表的主键同时还引用另一个表主键的外键时，就形成了一对一联系（图2-10和图2-11）。

titles	
title_id	title_name
To1	1977!
To2	200 Years of Ger...
To3	Ask Your System...
To4	But I Did It Unco...

royalties	
title_id	advance
To1	10000
To2	1000
To3	20000

图2-10 一对一联系。表titles中的每一行最多和表royalties有一个相匹配的行，表royalties中的每一行也最多和表titles有一个相匹配的行。这里，表royalties的主键也是外键（其引用表titles的主键）

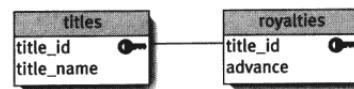


图2-11 本图显示了描述图2-10中一对一联系可选的另一种方式。连接线指出了关联的列，关键字符指出主键

2.4.2 一对多

在一对多联系中，表A中的每一行和表B有零个或多个相匹配的行，但表B中的每一行和表A仅有一个相匹配的行。例如，A出版社能出版许多本书，但一本书只能被一个出版社出版。

当一个表的主键作为多个表的外键时，就形成了一对多联系（图2-12和图2-13）。

publishers	
pub_id	pub_name
Po1	Abatis Publishers
Po2	Core Dump Books
Po3	Schadenfreude Press
Po4	Tenterhooks Press

titles		
title_id	title_name	pub_id
To1	1977!	Po1
To2	200 Years of Ger...	Po3
To3	Ask Your System...	Po2
To4	But I Did It Unco...	Po4
To5	Exchange of Plat...	Po4

图2-12 一对多联系。表publishers中的每一行和表titles有多个行相匹配，但表titles中的每一行和表publishers仅有一个行相匹配。这里，表publishers（一个表）的主键以表titles（多个表）的外键形式出现

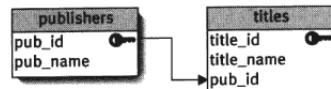


图2-13 描述图2-12中一对多联系的另一种方式。连接线的一端指出了一个表，带箭头的另一端指出了多个表

2.4.3 多对多

在多对多联系中，表A中的每一行和表B有零个或多个相匹配行，表B中的每一行和表A也有多个相匹配行。例如，一个作者能写多本书，一本书也能有多个作者。

仅当产生了第三个名为联结表（其组合键是两个表主键的组合，每一个组合键的单独列是一个外键）的表时，多对多联系就形成了。这种技术将对联结表中每一行产生唯一值，并可将多对多联系分割成两个独立的一对多联系（图2-14和图2-15）。

titles

title_id	title_name
T01	1977!
T02	200 Years of Ger...
T03	Ask Your System...
T04	But I Did It Unco...
T05	Exchange of Plat...

title_authors

title_id	au_id
T01	A01
T02	A01
T03	A05
T04	A03
T04	A04
T05	A04

authors

au_id	au_fname	au_lname
A01	Sarah	Buchman
A02	Wendy	Heydemark
A03	Hallie	Hull
A04	Klee	Hull

图2-14 多对多联系。表title_authors将表titles和authors之间的多对多联系分割成两个一对多联系。表titles中每一行和表title_authors有多个行相匹配，表authors中的每一行也一样。这里，表title_authors中的title_id，是引用表titles中主键的外键；表title_authors中的au_id，是引用表authors中主键的外键

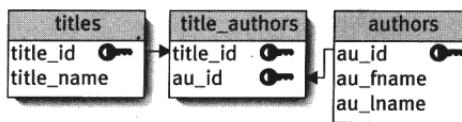


图2-15 描述图2-14中多对多联系的另一种方式

✓ 提示

- 联结（对多个表执行操作）将在第7章中讲解。
- 如果为表增加重复的组，就可以不用创建第三个表而建立多对多联系，但这种方法违反了第

一范式，见下一节。

- 一对多联系也叫父子或大纲-细节联系。
- 联结表(junction table)也叫做关联(associating)、链接(linking)、透视(pivot)、连接(connection)或交叉(intersection)表。

2

44

2.5 规范化

将书籍的所有信息（或任何实体类型）合并到一个集成的表中是可能的，但那张表将包含重复的数据，每个标题（行）将包含冗余的作者（author）、出版社（publisher）和稿酬（royalty）的详细信息。冗余是数据库用户和管理员的大敌，它导致数据库变得异常巨大，查询速度减慢；对于维护来说，也是可怕的（当某人搬家时，只应该在一个位置改变他的地址，而不是数千个位置）。

冗余导致多种更新异常，也就是插入、更新和删除行的操作困难。规范化(normalization)是通过修改表以减少冗余和矛盾的一系列步骤。在每一步之后，数据库都到达一个特定的范式(normal form)。关系模型定义了3种范式，以著名的序数命名。

- 第一范式 (1NF)
- 第二范式 (2NF)
- 第三范式 (3NF)

每一种范式都比前一种更健壮。符合3NF的数据库也符合2NF和1NF。规范化水平越高，表的数量就越多。无损分解(lossless decomposition)能确保表的分割不会引起信息丢失，依赖-保持分解(dependency-preserving decomposition)能确保联系不丢失。当表被分割时，存在匹配的主键和外键列不应被认为多余的数据。

规范化不是系统化。它是一个涉及重复表的分割、重新联结和精炼的迭代过程，直到数据库设计人员对结果（临时）满意为止。

45

2.5.1 第一范式

满足第一范式的表：

- 列仅包含原子值。

并且

- 没有重复的组。

原子值（也称为标量值），是不能再细分的单一值（图2-16）。重复的组是两个或多个逻辑相关联的列的集合（图2-17）。将数据存储在两个相关联的表中可以解决这个问题（图2-18）。

title_id	title_name	authors
T01	1977!	A01
T04	But I Did It Unconsciously	A03, A04
T11	Perhaps It's a Glandular Problem	A03, A04, A06

图2-16 在第一范式中，每一个表的行列交叉点必须包含一个不能再被细分、有意义的单一值。这个表的列authors给出了多个作者，因此违反了第一范式

title_id	title_name	author1	author2	author3
T01	1977!	A01		
T04	But I Did It Unconsciously	A03	A04	
T11	Perhaps It's a Glandular Problem	A03	A04	A06

图2-17 将列authors重新分配到重复的组也违反了第一范式。不能用多个列来描述一个实体的多个实例

title_id	title_name
T01	1977!
T04	But I Did It Unco...
T11	Perhaps It's a Gla...

title_id	au_id
T01	A01
T04	A03
T04	A04
T11	A03
T11	A04
T11	A06

图2-18 正确的设计方案是将author信息移入一个新的子表中，对每个title的作者都有
一行。父表的主键是title_id，子表的组合键是title_id和au_id

违反第一范式的数据库会引起以下问题。

- 在行列交叉点的多个值意味着表名、列名、键值组合在一起仍不足以标明数据库中每个值。
- 因为必须依赖于值的顺序，所以检索、插入、更新或删除一个值（在多个之中）是很困难的。
- 查询相当复杂（性能的杀手）。
- 更深层的规范化要解决的问题变得无法解决。

46

原 子 性

从数据库用户的角度看，原子值被认为是不可分割的。实际上，如一个日期、一个电话号码和一个字符串，本质上不是不可分割的。可以将日期分解为年、月和日；将电话号码分解为国家代码、地区代码和用户号码；将单独的字符串成一行。你所关心的重点是：DBMS提供了操作符和函数，如果有必要的话，你可以提取和操作原子值成分，例如substring()函数提取电话号码的地区代码，year()函数提取日期的年份。

2.5.2 第二范式

在给出第二范式的约束之前，先说明一下，当满足下列条件时，第一范式的表自动满足第二范式。

- 主键是一个列（也就是说，关键字不是组合的）。
- 或者
- 表中所有的列是主键的一部分（单一的或组合的）。

满足第二范式的表：

- 满足第一范式。

并且

- 非部分函数依赖。

如果表中一些组合键的（但不是全部）值确定了一个非键列的值，则表包含部分函数依赖。第二范式表是完全函数依赖，意味着如果组合键中任何一列值改变，将导致非键列的值需要被更新。

图2-19所示表中的组合键是title_id和au_id，非键列是au_order（指多个作者在书的封面上出现的顺序）和au_phone（作者的电话号码）。

对于每一个非键列，问：“如果仅知道部分主键的值，能确定非键列的值吗？”“否”的回答意味着非键列是完全函数依赖的（好的情况），“是”的回答意味着非键列是部分函数依赖的（糟糕的情况）。

对于列au_order，问题如下。

- 如果仅知道title_id，能确定au_order吗？否，因为对于同一个图书书名也许有多个作者。
- 如果仅知道au_id，能确定au_order吗？否，因为还需要知道具体图书书名。

好的情况——au_order是完全函数依赖并且能够在表中保持。这个依赖被写作：

$\{title_id, au_id\} \rightarrow \{au_order\}$

被读作“title_id和au_id决定au_order”或者“au_order依赖于title_id和au_id”。决定因素是箭头左边的表达式。

对于列au_phone，问题如下。

- 如果仅知道title_id，能确定au_phone吗？否，因为对于同一个图书书名也许有多个作者。
 - 如果仅知道au_id，能确定au_phone吗？是，因为作者的电话号码不依赖于标题。
- 糟糕的情况——au_phone是部分函数依赖，必须被移到别处（可能是表authors或表phone_numbers）以满足第二范式的规则。

2.5.3 第三范式

满足第三范式的表：

- 满足第二范式。

并且

- 没有传递依赖。

如果一个非键列的值确定了另一个非键列的值，则表包含传递依赖。在第三范式的表中，非键列相互独立并且只依赖于主键列。第三范式是第二范式之后的下一个逻辑步骤。

图2-20中表的主键是title_id，非键列是price（书的价格）、pub_city（出版社所在的城市）和pub_id（该书的出版社）。

title_authors	
title_id	●
au_id	●
au_order	
au_phone	

图2-19 au_phone 依赖于 au_id 而不是 title_id，因此这个表包含了部分函数依赖，不满足第二范式

titles	
title_id	●
price	
pub_city	
pub_id	

图2-20 pub_city 依赖于 pub_id，因此这个表包含了传递依赖，不是第三范式

对于每一个非键列，问：“如果知道一些其他的非键列的值，能确定一个非键列的值吗？”“否”的答案意味着列不是传递依赖（好的情况），“是”的答案意味着能确定值的列是传递依赖的列（糟糕的情况）。

对于列`price`，问题是：

- 如果知道`price`，能确定`pub_id`吗？否。
- 如果知道`price`，能确定`pub_city`吗？否。

对于列`pub_city`，问题是：

- 如果知道`pub_city`，能确定`price`吗？否。
- 如果知道`pub_city`，能确定`pub_id`吗？否，因为一个城市可能有多个出版社。

对于列`pub_id`，问题是：

- 如果知道`pub_id`，能确定`price`吗？否。
- 如果知道`pub_id`，能确定`pub_city`吗？是！因为书的出版地依赖于出版社。

糟糕的情况——`pub_city`传递依赖于`pub_id`，并且必须被移到别处（可能是一个`publishers`表）以满足第三范式的规则。

可以看到，只问“如果知道B能确定A吗？”对于发现传递依赖是不够的，还必须问“如果知道A能确定B吗？”

49

2.5.4 其他范式

还存在着更高级别的范式，但关系模型不需要（甚至提及）它们，它们只用在某些需要避免冗余的情况下。简要地说，它们是：

- Boyce-Codd范式（Boyce-Codd normal form, BCNF）是更严格第三范式版本。BCNF处理那些含有多个候选键、组合候选键或重叠候选键的表。如果表的每个确定因素都是候选键，则满足BCNF（确定列是指某些完全函数依赖的列）。
- 满足第四范式（4NF）的表是BCNF的，并且没有多值依赖（multivalued dependency, MVD）。当一个表至少包含三列、一列有多个行且其值与其他列中的一个行的值匹配时，发生多值依赖。
- 假设雇员能够被分派到多个项目中，且每一个雇员有多种技能。如果将这些信息填入一张表中，必须使用3个属性作为关键字，因为再少就不能唯一标识一行。在`emp_id`和`proj_id`之间的关系是多值依赖，因为对于表中每一对`emp_id/skill_id`的值，只有`emp_id`才能（不依赖于`skill_id`）确定关联的`proj_id`值的集合。在`emp_id`和`skill_id`之间的联系也是多值依赖，因为对于一个`emp_id/proj_id`对的`skill`值的集合，总是只依赖于`emp_id`。要将多值依赖的表转换为4NF，可以将每一对多值依赖移到一个新表中。
- 第五范式（5NF）的表满足4NF，并且没有联结依赖。联结依赖是多值依赖的泛化。5NF的目标是使表不能进一步分解为多个更小的表。5NF消除的冗余和异常很少，而且也不直观。在实际的数据库中，会看到1NF、2NF和3NF，偶尔也有4NF，4NF甚至3NF表也几乎总是5NF的。

非规范化

规范化产生的表的数量不断增加，为加快查询速度可能需要转而将数据库非规范化

(denormalize) (因为较少的表，会减少计算机联结和磁盘I/O操作的开销)。这种常见的方法为了性能而牺牲数据完整性，并带来一些其他问题。非规范化的数据库：

- 通常比规范化的数据库更难以理解。
- 通常会加快检索速度，但会降低更新速度。
- 增大了插入不一致数据的风险。
- 或许能改进某些数据库应用软件的性能，但会降低其他方面的性能（因为用户的表存取模式会随着时间而改变）。

需要非规范化不是关系模型的弱点，但揭示了DBMS中一个有缺陷的模型实现。非规范化数据库的一个普通用途是用作从其他表复制过来数据的永久日志。日志行是冗余的，但因为它们只是被插入（决不被更新），因此它们用作不受源表将来变化影响的审核记录。

2.6 示例数据库

随便翻看一本SQL或数据库设计的书，就会发现students/courses/teachers、customers/orders/products或者authors/books/publishers数据库。按照惯例，本书中绝大多数SQL例子使用名为books的authors/books/publishers数据库。例如名叫books的数据库，下面将介绍有关books的一些内容。

- 回顾2.1节，对于用户来说，数据库是表的集合体（并且只有表）。数据库books包括5个表，包含作者（author）、出版的图书书名（title）、出版社（publisher）和稿酬（royalty）等信息。图2-21使用本章以前介绍的图形约定描述了books中的表和关系。

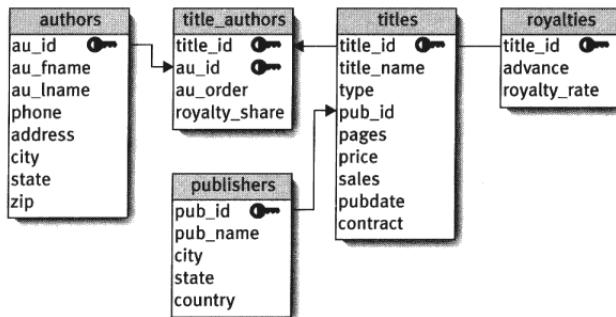


图2-21 示例数据库books

- 在第10章及以后的内容中，SQL语句修改（而不是仅仅检索）books中的数据。除非另外注明，在各章中都将以books的原始副本开始。换句话说，假设数据库在某一章中的改变不会延续到下一章。
- 在本节中提及的某些概念，如数据类型和空值，将在下一章进行讲解。
- books是一个教学工具，它的结构和大小并不接近于实际产品数据库的复杂程序。
- 要在自己的DBMS中创建示例数据库，参见2.7节。

2.6.1 表 authors

表authors描述了书的作者。每一个作者都有唯一标识符，也就是主键。表2-3显示了表authors

52

的结构。图2-22显示了表authors的内容。

表2-3 表authors的结构

列名	描述	数据类型	空?	键
au_id	作者的唯一标识符	CHAR(3)		主键
au_fname	作者的名	VARCHAR(15)		
au_lname	作者的姓	VARCHAR(15)		
phone	作者的电话号码	VARCHAR(12)	是	
address	作者的地址	VARCHAR(20)	是	
city	作者所在的城市	VARCHAR(15)	是	
state	作者所在的州	CHAR(2)	是	
zip	作者的邮政编码	CHAR(5)	是	

au_id	au_fname	au_lname	phone	address	city	state	zip
A01	Sarah	Buchman	718-496-7223	75 West 205 St	Bronx	NY	10468
A02	Wendy	Heydemark	303-986-7020	2922 Baseline Rd	Boulder	CO	80303
A03	Hallie	Hull	415-549-4278	3800 Waldo Ave, #14F	San Francisco	CA	94123
A04	Klee	Hull	415-549-4278	3800 Waldo Ave, #14F	San Francisco	CA	94123
A05	Christian	Kells	212-771-4680	114 Horatio St	New York	NY	10014
A06		Kellsey	650-836-7128	390 Serra Mall	Palo Alto	CA	94305
A07	Paddy	O'Furniture	941-925-0752	1442 Main St	Sarasota	FL	34236

图2-22 表authors的内容

2.6.2 表 publishers

53

表publishers描述了图书的出版社。每一个出版社都有唯一的标识符，也就是主键。表2-4显示了表publishers的结构。图2-23显示了表publishers的内容。

表2-4 表publishers的结构

列名	描述	数据类型	空?	键
pub_id	出版社的唯一标识符	CHAR(3)		主键
pub_name	出版社的名字	VARCHAR(20)		
city	出版社所在的城市	VARCHAR(15)		
state	出版社所在的州	CHAR(2)	是	
country	出版社所在的国家	VARCHAR(15)		

pub_id	pub_name	city	state	country
P01	Abatis Publishers	New York	NY	USA
P02	Core Dump Books	San Francisco	CA	USA
P03	Schadenfreude Press	Hamburg	NULL	Germany
P04	Tenterhooks Press	Berkeley	CA	USA

图2-23 表publishers的内容

2.6.3 表 titles

表titles描述了图书。每一本图书都有唯一的标识符，也就是主键。表titles包含了一个外键pub_id，引用了表publishers以指明书的出版社。表2-5显示了表titles的结构。图2-24显示了表titles的内容。

表2-5 表titles的结构

列名	描述	数据类型	空?	键
title_id	标题的唯一标识符	CHAR(3)		主键
title_name	书的标题	VARCHAR(40)		
type	书的类别	VARCHAR(10)	是	
pub_id	出版社的标识符	CHAR(3)		外键publishers(pub_id)
pages	页数统计	INTEGER	是	
price	封面价格	DECIMAL(5,2)	是	
sales	售出的总册数	INTEGER	是	
pubdate	出版日期	DATE	是	
contract	合同，如签有合同则为非零	SMALLINT		

title_id	title_name	type	pub_id	pages	price	sales	pubdate	contract
T01	1977!	history	P01	107	21.99	566	2000-08-01	1
T02	200 Years of German Humor	history	P03	14	19.95	9566	1998-04-01	1
T03	Ask Your System Administrator	computer	P02	1226	39.95	25667	2000-09-01	1
T04	But I Did It Unconsciously	psychology	P04	510	12.99	13001	1999-05-31	1
T05	Exchange of Platitudes	psychology	P04	201	6.95	201440	2001-01-01	1
T06	How About Never?	biography	P01	473	19.95	11320	2000-07-31	1
T07	I Blame My Mother	biography	P03	333	23.95	1500200	1999-10-01	1
T08	Just Wait Until After School	children	P04	86	10.00	4095	2001-06-01	1
T09	Kiss My Boo-Boo	children	P04	22	13.95	5000	2002-05-31	1
T10	Not Without My Faberge Egg	biography	P01	NULL	NULL	NULL	NULL	0
T11	Perhaps It's a Glandular Problem	psychology	P04	826	7.99	94123	2000-11-30	1
T12	Spontaneous, Not Annoying	biography	P01	507	12.99	100001	2000-08-31	1
T13	What Are The Civilian Applications?	history	P03	802	29.99	10467	1999-05-31	1

图2-24 表titles的内容

2.6.4 表 titles_authors

作者和图书有多对多联系，因为一个作者可以写多本图书，且一本图书可以有多个作者。表title-authors是表authors和titles的联结表，见2.4节。列title_id和au_id共同组成了一个组合键，每一列分别是一个引用titles和authors的外键。非键列指明在图书封面上的作者顺序（一本图书只有唯一作者时，总是1）和每一个作者收到的全部稿酬份额（一本图书只有唯一作者时，总是1.0）。表2-6

55 所示为表title_authors的结构。图2-25显示了表title_authors的内容。

表2-6 表title_authors的结构

列名	描述	数据类型	空?	键
title_id	标题标识符	CHAR(3)		主键, 外键titles(title_id)
au_id	作者标识符	CHAR(3)		主键, 外键authors(au_id)
au_order	书封面上作者姓名的顺序	SMALLINT		
royalty_share	作者稿酬的份额	DECIMAL(5,2)		

title_id	au_id	au_order	royalty_share
T01	A01	1	1.00
T02	A01	1	1.00
T03	A05	1	1.00
T04	A03	1	0.60
T04	A04	2	0.40
T05	A04	1	1.00
T06	A02	1	1.00
T07	A02	1	0.50
T07	A04	2	0.50
T08	A06	1	1.00
T09	A06	1	1.00
T10	A02	1	1.00
T11	A03	2	0.30
T11	A04	3	0.30
T11	A06	1	0.40
T12	A02	1	1.00
T13	A01	1	1.00

图2-25 表title_authors的内容

2.6.5 表royalties

表royalties指明付给每一本图书所有作者（不是每一个作者）的稿酬比例，还包括按照稿酬预先付给每一本图书所有作者（不是每一个作者）的全部预付款。royalties的主键是titles_id。表royalties和titles有一对一的联系，因此表royalties的主键也是一个引用了titles主键的外键。表2-7显示了表royalties的结构。图2-26显示了表royalties的内容。

表2-7 表royalties的结构

列名	描述	数据类型	空?	键
title_id	标题的唯一标识符	CHAR(3)		主键, 外键titles(title_id)
advance	预先付给作者的预付款	DECIMAL(9,2)	是	
royalty_rate	作者稿酬的比率	DECIMAL(5,2)	是	

title_id	advance	royalty_rate
T01	10000.00	0.05
T02	1000.00	0.06
T03	15000.00	0.07
T04	20000.00	0.08
T05	100000.00	0.09
T06	20000.00	0.08
T07	1000000.00	0.11
T08	0.00	0.04
T09	0.00	0.05
T10	NULL	NULL
T11	100000.00	0.07
T12	50000.00	0.09
T13	20000.00	0.06

图2-26 表royalties的内容

2.7 创建示例数据库

要在自己的DBMS中创建（或者重建）数据库books，可以访问www.fehily.com，单击本书上的下载链接，然后按照屏幕上的指示进行操作。

创建books分为两个步骤：

(1) 使用DBMS内置工具创建一个新的、空白的、名为books的数据库。

(2) 运行SQL脚本创建books中的表，并用数据填充。

代码2-1显示了一个标准的(ANSI) SQL脚本，用其创建示例数据库的表，并向其中插入行。

✓ 提示

□ **DBMS** 如果正在使用Microsoft Access，就不用分两步了，只需要在Access中打开.mdb文件。

57

代码2-1 这个名为book_standard.sql的标准SQL脚本，创建示例数据库books中的表并用数据填充。
在配套站点上下载的包含脚本版本的文件可能需要修改才能运行在特定的DBMS上

```

DROP TABLE authors;
CREATE TABLE authors
(
au_id    CHAR(3)      NOT NULL,
au_fname VARCHAR(15)  NOT NULL,
au_lname VARCHAR(15)  NOT NULL,
phone    VARCHAR(12)   ,
address  VARCHAR(20)   ,
city     VARCHAR(15)   ,
state    CHAR(2)       ,
zip      CHAR(5)       ,
CONSTRAINT pk_authors PRIMARY KEY (au_id)
);
INSERT INTO authors VALUES('A01','Sarah','Buchman','718-496-7223',

```

```

'75 West 205 St', 'Bronx', 'NY', '10468');
INSERT INTO authors VALUES('A02', 'Wendy', 'Heydermark', '303-986-7020',
  '2922 Baseline Rd', 'Boulder', 'CO', '80303');
INSERT INTO authors VALUES('A03', 'Hallie', 'Hull', '415-549-4278',
  '3800 Waldo Ave, #14F', 'San Francisco', 'CA', '94123');
INSERT INTO authors VALUES('A04', 'Klee', 'Hull', '415-549-4278',
  '3800 Waldo Ave, #14F', 'San Francisco', 'CA', '94123');
INSERT INTO authors VALUES('A05', 'Christian', 'Kells', '212-771-4680',
  '114 Horatio St', 'New York', 'NY', '10014');
INSERT INTO authors VALUES('A06', '', 'Kellsey', '650-836-7128',
  '390 Serra Mall', 'Palo Alto', 'CA', '94305');
INSERT INTO authors VALUES('A07', 'Paddy', 'O''Furniture', '941-925-0752',
  '1442 Main St', 'Sarasota', 'FL', '34236');

DROP TABLE publishers;
CREATE TABLE publishers
(
  pub_id  CHAR(3)      NOT NULL,
  pub_name VARCHAR(20)  NOT NULL,
  city     VARCHAR(15)  NOT NULL,
  state    CHAR(2)      ,
  country  VARCHAR(15)  NOT NULL,
  CONSTRAINT pk_publishers PRIMARY KEY (pub_id)
);
INSERT INTO publishers VALUES('P01', 'Abatis Publishers', 'New York', 'NY', 'USA');
INSERT INTO publishers VALUES('P02', 'Core Dump Books', 'San Francisco', 'CA', 'USA');
INSERT INTO publishers VALUES('P03', 'Schadenfreude Press', 'Hamburg', NULL, 'Germany');
INSERT INTO publishers VALUES('P04', 'Tenterhooks Press', 'Berkeley', 'CA', 'USA');

DROP TABLE titles;
CREATE TABLE titles
(
  title_id  CHAR(3)      NOT NULL,
  title_name VARCHAR(40)  NOT NULL,
  type       VARCHAR(10)   ,
  pub_id    CHAR(3)      NOT NULL,
  pages     INTEGER       ,
  price     DECIMAL(5,2)  ,
  sales     INTEGER       ,
  pubdate   DATE         ,
  contract  SMALLINT     NOT NULL,
  CONSTRAINT pk_titles PRIMARY KEY (title_id)
);
INSERT INTO titles VALUES('T01', '1977!', 'history', 'P01',
  107, 21.99, 566, DATE '2000-08-01', 1);
INSERT INTO titles VALUES('T02', '200 Years of German Humor', 'history', 'P03',
  14, 19.95, 9566, DATE '1998-04-01', 1);
INSERT INTO titles VALUES('T03', 'Ask Your System Administrator', 'computer', 'P02',
  1226, 39.95, 25667, DATE '2000-09-01', 1);
INSERT INTO titles VALUES('T04', 'But I Did It Unconsciously', 'psychology', 'P04',
  510, 12.99, 13001, DATE '1999-05-31', 1);
INSERT INTO titles VALUES('T05', 'Exchange of Platitudes', 'psychology', 'P04',
  201, 6.95, 201440, DATE '2001-01-01', 1);
INSERT INTO titles VALUES('T06', 'How About Never?', 'biography', 'P01',
  473, 19.95, 11320, DATE '2000-07-31', 1);
INSERT INTO titles VALUES('T07', 'I Blame My Mother', 'biography', 'P03',
  333, 23.95, 1500200, DATE '1999-10-01', 1);
INSERT INTO titles VALUES('T08', 'Just Wait Until After School', 'children', 'P04',
  86, 10.00, 4095, DATE '2001-06-01', 1);

```

```

INSERT INTO titles VALUES('T09','Kiss My Boo-Boo','children','P04',
    22,13.95,5000,DATE '2002-05-31',1);
INSERT INTO titles VALUES('T10','Not Without My Faberge Egg','biography','P01',
    NULL,NULL,NULL,NULL,0);
INSERT INTO titles VALUES('T11','Perhaps It''s a Glandular Problem','psychology','P04',
    826,7.99,94123,DATE '2000-11-30',1);
INSERT INTO titles VALUES('T12','Spontaneous, Not Annoying','biography','P01',
    507,12.99,100001,DATE '2000-08-31',1);
INSERT INTO titles VALUES('T13','What Are The Civilian Applications?','history','P03',
    802,29.99,10467,DATE '1999-05-31',1);

```

```

DROP TABLE title_authors;
CREATE TABLE title_authors
(
    title_id      CHAR(3)      NOT NULL,
    au_id         CHAR(3)      NOT NULL,
    au_order      SMALLINT     NOT NULL,
    royalty_share DECIMAL(5,2) NOT NULL,
    CONSTRAINT pk_title_authors PRIMARY KEY (title_id, au_id)
);
INSERT INTO title_authors VALUES('T01','A01',1,1.0);
INSERT INTO title_authors VALUES('T02','A01',1,1.0);
INSERT INTO title_authors VALUES('T03','A05',1,1.0);
INSERT INTO title_authors VALUES('T04','A03',1,0.6);
INSERT INTO title_authors VALUES('T04','A04',2,0.4);
INSERT INTO title_authors VALUES('T05','A04',1,1.0);
INSERT INTO title_authors VALUES('T06','A02',1,1.0);
INSERT INTO title_authors VALUES('T07','A02',1,0.5);
INSERT INTO title_authors VALUES('T07','A04',2,0.5);
INSERT INTO title_authors VALUES('T08','A06',1,1.0);
INSERT INTO title_authors VALUES('T09','A06',1,1.0);
INSERT INTO title_authors VALUES('T10','A02',1,1.0);
INSERT INTO title_authors VALUES('T11','A03',2,0.3);
INSERT INTO title_authors VALUES('T11','A04',3,0.3);
INSERT INTO title_authors VALUES('T11','A06',1,0.4);
INSERT INTO title_authors VALUES('T12','A02',1,1.0);
INSERT INTO title_authors VALUES('T13','A01',1,1.0);

```

```

DROP TABLE royalties;
CREATE TABLE royalties
(
    title_id      CHAR(3)      NOT NULL,
    advance       DECIMAL(9,2)      ,
    royalty_rate DECIMAL(5,2)      ,
    CONSTRAINT pk_royalties PRIMARY KEY (title_id)
);
INSERT INTO royalties VALUES('T01',10000,0.05);
INSERT INTO royalties VALUES('T02',1000,0.06);
INSERT INTO royalties VALUES('T03',15000,0.07);
INSERT INTO royalties VALUES('T04',20000,0.08);
INSERT INTO royalties VALUES('T05',100000,0.09);
INSERT INTO royalties VALUES('T06',20000,0.08);
INSERT INTO royalties VALUES('T07',1000000,0.11);
INSERT INTO royalties VALUES('T08',0,0.04);
INSERT INTO royalties VALUES('T09',0,0.05);
INSERT INTO royalties VALUES('T10',NULL,NULL);
INSERT INTO royalties VALUES('T11',100000,0.07);
INSERT INTO royalties VALUES('T12',50000,0.09);
INSERT INTO royalties VALUES('T13',20000,0.06);

```

2

59

60

在 之前的内容中很少介绍SQL。牢记这个式子：

SQL ≠ 关系模型

SQL是基于关系模型的，但没有忠实地实现它。SQL与模型不同的一点是，主键是可选的，而不是强制的。因此，没有主键的表可以接受重复的行，就会有某些数据无法访问。讨论全部差异超出了本书范围（参见2.2节中的“学习数据库设计”提要栏）。这些差异的结果是，由DBMS用户而非DBMS本身来实施关系结构。另一个结果是，在表2-1中，模型和SQL术语是不可以互换的。

伴随着警告，现在是学习SQL的时候了。SQL程序是按顺序执行的SQL语句序列。要编写一个程序，必须知道控制SQL语法的规则。本章说明如何编写有效的SQL语句，同时还包括数据类型（**data type**）和空值（**null**）。

61

3.1 SQL语法

图3-1显示了一个SQL语句的例子。不必关心这个语句的意思（语义），这里使用它来说明SQL语法。

注释。注释是可选、解释程序的文本。注释通常说明程序做什么和如何做，以及为什么代码被更改。注释是给人看的，编译器会忽略它们。注释以连续的两个连字符（--）开始并延续到行尾的方式被引入。

SQL语句。SQL语句是被关键字引入的标记的有效组合。标记是SQL语言基本的、不可分割的元素，且语法上不能缩减。标记包括关键字、标识符、操作符、字面量（常数）和标点符号。

子句。SQL语句至少有一条子句。一般而言，一个子句是一个被关键字引入的SQL语句片段，它是必需的或者可选的，并且必须按特定顺序给出。在这个例子中，**SELECT**、**FROM**、**WHERE**和**ORDER BY**引入了4个子句。

关键字。关键字是SQL的保留字，因为它们在语言中有特定的含义。在特定的上下文之外（如一个标识符）使用关键字将导致错误的发生。DBMS混合使用标准或非标准的关键字，请查阅DBMS文档中的**keywords**或**reserved words**。

标识符。标识符是数据库设计人员用于命名数据库对象（如表、列、别名、索引和视图）的词。在这个例子中，**au_fname**、**au_lname**、**authors**和**state**是标识符。要获得更多信息，可参见3.3节。

62

使语句终结的分号。一条SQL语句以分号（;）结束。

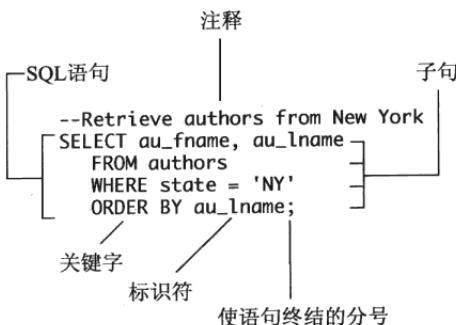


图3-1 带有注释的SQL语句

SQL是一种形式自由的语言。它的语句可以：

- 是大写字母或小写字母（例如，SELECT、select和sElEcT被认为是同一个关键字）。
- 只要不把单词、标记或引号字符串分割为两部分，可以在下一行继续。
- 和其他语句在同一行。
- 从任意一列开始。

尽管有这些灵活性，也应该采用一致的风格（图3-2）。使用大写关键字和小写标识符，并且在它自己的行上缩进每个子句。关于风格和语法的约定信息，请参见前言的“排版约定”和“语法约定”。

```

select au_fname
      , AU_LNAME
      FROM
authors WHERE state
= 'NY' order
      by
AU_LNAME
;
  
```

图3-2 关于如何格式化一条SQL语句没有太多规则。这条语句和图3-1中的语句是等价的

常见错误

某些常见的SQL程序错误如下。

- 遗漏了使语句终结的分号。
- 拼错了关键字或标识符。
- 匹配错误或无匹配的圆括号或引号。
- 列出子句的顺序颠倒。
- 没有用单引号环绕字符串或日期。
- 用引号环绕数字或关键字NULL。
- 错误匹配的表和列（例如输入SELECT royalty_share FROM authors，而不是SELECT royalty_share FROM title_authors）。

这些错误易于发现和改正，即使DBMS返回的是模糊或者无用的错误消息。在DBMS将一条语

句标记为错误之前，实际上真实的错误可能已经发生。例如，运行：

```
CREATE TABLE misspelled_name
```

DBMS将正确向后运行，产生一个名称错误的表。直到后面试图引用这个表的时候，错误才会显露出来，如：

63

```
SELECT * FROM correct_name
```

✓ 提示

- SQL语句指令性的关键字叫做谓词（verb），因为它指出要执行的动作。
- 要区分SELECT语句（从SELECT到分号的完整语句）和SELECT子句（指出输出列SELECT语句的一部分）。
- 某些DBMS支持括起来的注释，它以/*开始，延续一行或多行，以*/结束。可以将一个括起来的注释嵌套到另一个注释中。
- 表达式（expression）是求单1值的合法符号组合。可以组合操作符、标识符、字面量、函数、列名、别名等数学符号和逻辑符号。表3-1列出了某些常见的表达式和例子，这些表达式将在后面详细讲解。

64

表3-1 表达式的类型

类 型	例 子
Case	CASE WHEN n < > 0 THEN x/n ELSE 0 END
Cast	CAST(pubdate AS CHARACTER)
Datetime value	start_time + '01:30'
Interval value	INTERVAL '7' DAY * 2
Numeric value	(sales*price)/12
String value	'Dear 'llau_fnameli','

3.2 SQL 标准和一致性

SQL:2003是SQL委员会每隔几年更新一次的最新官方标准版（之前的版本分别发布于1986、1989、1992和1999年）。

每一个标准：

- 引入了新的语言元素。
- 明确或更新了先前标准中的元素。
- 有时删除了已存在的元素（因为新元素替代了它们，或者它们不再被DBMS供应商采用）。

标准是庞大的——几千页晦涩难懂的说明，没有供应商能够遵循（或愿意遵循）全部标准。供应商尽力遵循名为Core SQL的标准子集。这种一致性级别是供应商能够宣称他们符合标准SQL不得不达到的最低要求。SQL-92引入了一致性级别，SQL:1999也有它们的一致性级别，因此当读到DBMS的一致性声明时，注意它引用的是哪个SQL标准和哪个级别。

事实上，SQL-92经常被认为是标准，因为它定义了很多SQL语言中最重要的和不变的部分。除非另外指出，本书中的SQL元素是SQL-92的一部分，也是SQL:1999和SQL:2003的一部分。SQL-92最低的一致性级别称为Entry，而不是Core。

程序应该尽可能地遵循SQL标准。理想情况下，要编写可移植的SQL程序，甚至不用知道你是用哪个DBMS编程的。然而，SQL委员会不是由语言学家和关系模型纯理论者组成的，而是由商业DBMS供应商来驾驭和操纵的。结果，每一个DBMS供应商投入资源以接近最低的Entry或Core一致性要求，然后努力增加能在市场上区分他们产品的非标准功能——这就意味着，SQL程序将是不可移植的。这些供应商专有的特性，经常迫使你去修改或者重写那些SQL程序以便开发人员运行在不同DBMS上。

✓ 提示

- 要对照标准测试SQL代码，访问<http://developer.mimer.se/validator>，单击针对SQL 1992、1999或2003标准的验证链接。可以输入或粘贴SQL语句来检查它们是否符合标准，以及语句构成是否正确。
- **DBMS** 作为本书所讲解的DBMS之一，PostgreSQL对于SQL标准遵循的最纯粹。你所用的DBMS也许提供了设置以使其更好地符合SQL标准。例如，MySQL有ansi模式，Microsoft SQL Server有SET ANSI_DEFAULTS ON。

3.3 标识符

标识符是可以在数据库对象（无论是模式、数据库、列、关键字、索引、视图、约束或者任何CREATE语句所创建的内容）层次中明确地引用一个对象的名称。标识符在它的作用域（定义了什么地方和什么时间能够被引用）内必须是唯一的。一般而言：

- 数据库的名称在数据库服务器的一个特定实例上，必须是唯一的。
 - 表和视图的名称在给定的模式或数据库里，必须是唯一的。
 - 列、关键字、索引和约束的名称在给定的表或视图里，必须是唯一的。
- 这个模式允许对作用域不交迭的对象使用完全相同的名称。例如，可以对不同表中的列指定同样的名称，或者对于不同数据库中的表指定同样的名称。

✓ 提示

- 关于标识数据库对象，参见表2-2。
- **DBMS** 对于不同的DBMS，标识符需要唯一的作用域是不同的。例如，SQL Server需要一个仅对它的表唯一的索引名，而Oracle和DB2需要一个在整个数据库中唯一的索引名。

查阅DBMS文档中的identifiers或names。

标准的SQL有下列标识符命名规则。

- 可以长达128个字符。
- 必须以字母开头。
- 可以包含字母、数字和下划线(_)。
- 不可以包含空格或特殊字符（如#、\$、&、%或标点符号）。
- 不可以是保留关键字（除了被引号括起来的标识符以外）。

标准的SQL区分保留关键字和非保留关键字。不可以使用保留关键字作为标识符，因为它们在SQL中有特定的含义。例如，不能将表命名为select，或者将列命名为sum。非保留关键字仅在某些上下文中具有特殊的含义，能够在其他的上下文中被用作标识符。大多数非保留关键字实际上是内置的表和函数名称，因此即便它是最安全的，也绝不要使用它们作为标识符。

可以使用带引号的标识符，也称为分隔标识符，来打破某些SQL的标识符规则。带引号的标识符

66

是被双引号环绕着的名称。这个名称可以包含空格和特殊字符，是区分大小写的，并且可以是保留关键字。带引号的标识符可能干扰其他程序员，并导致第三方甚至供应商自己的工具也出现问题，因此使用它们通常是一个糟糕的主意。

这里对于选择标识符的名称还有一些建议。

- 即使DBMS有更少的限制，也要遵守标准规则（例如，Oracle的名称可以包含#和\$符号）。
- 在某些情况下，DBMS比标准有更多的限制（例如，MySQL的标识符最多只能有64个字符长）。
- 使用小写字母。
- 使用names_with_underscores比nameswithoutthem更容易让人明白。
- 在整个数据库中使用固定的名称和缩写，选用emp或employee并且坚持一贯。

✓ 提示

- 尽管不能使用（不带引号的）保留字作为标识符，但可以将其嵌入到标识符中。例如，group和max是非法的标识符，但groups和max_price是合法的。如果担心某些标识符在其他的非标准SQL语言中可能是保留字，可以在名称的末尾添加一个下划线（如element_），因为没有保留字以一个下划线结束。
- **DBMS** 可以使用双引号或者方括号（[]）来环绕SQL Server带引号的标识符，方括号是首选。在DB2中，可以使用保留字作为标识符（这样做不好，因为程序将不可移植）。MySQL的ANSI_QUOTES模式允许带双引号的标识符。DBMS都有自己的非标准的关键字，请查阅DBMS文档中的keywords或reserved words。
- 在MySQL中，下层操作系统是否区分大小写决定了数据库和表名称是否区分大小写。
- SQL标准要求DBMS在内部将标识符转换为大写字母。因此在SQL编译器内部，不带引号的标识符myname和带引号的标识符"MYNAME"（不是"myname"）是等价的。PostgreSQL不遵循标准，将其转换为小写字母。要编写可移植的程序，总是用引号引起一个特定的名称或从不用引号引起（而不是混合使用）。DBMS是否区分大小写不是固定不变的，因此最好是尊重用户定义的标识符的大小写。

67

3.4 数据类型

回顾2.1节，域是列允许的有效值的范围。要定义一个域，使用列的数据类型和约束（见第11章）。数据类型或列的类型有以下特征。

- 表中的每一列有单一的数据类型。
- 数据类型属于表3-2中列出的种类之一（每一种都将在下一节讲解）。

表3-2 数据类型的种类

种 类	存储数据
Character string（字符串）	字符串
Binary large object（二进制大型对象）	二进制数
Exact numeric（精确数字）	整数和小数数值
Approximate numeric（近似数字）	浮点数
Boolean（布尔值）	真值：真、假或未知
Datetime（日期时间）	日期和时间值
Interval（时间间隔）	日期和时间间隔

- 数据类型决定了列允许的值和它支持的运算。例如，整数类型表示在DBMS所定义的界限内的任何整数，并且支持普通的数学运算，如加、减、乘和除。但是整数不能表示'jack'这样的非数字值，并且不支持（例如转为大写和连接字符串的）字符运算。
- 数据类型将影响列的排序。整数1、2和10按数值排序是1、2、10。字符串‘1’、‘2’和‘10’按字典顺序排序，产生‘1’、‘10’、‘2’。字典顺序排序通过检查每一个字符的值来对字符串排序。这里，‘10’在‘2’的前面，因为‘1’（‘10’的第一个字符）按字典顺序是在‘2’之前的。要了解更多的排序信息，参见4.4节。
- 某些数据类型，如二进制对象，不能被索引（见第12章）。
- 可以在字符型、数值型、布尔型、日期和时间间隔型的列中存储字面量（常数）。表3-3所示为一些例子，下节将有更多的例子。不要混淆文本串'2009'和数值串2009。SQL标准将字面量定义为非空的常数。

表3-3 字面量的例子

字面量	例子
Character String（字符串）	'42', 'ennui', 'don''t', N'Jack'
Numeric（数字）	42, 12.34, 2., .001, -123, +6.33333, 2.5E2, 5E-3
Boolean（布尔值）	TRUE, FALSE, UNKNOWN
Datetime（日期）	DATE '2005-06-22', TIME '09:45:00', TIMESTAMP '2006-10-19 10:23:54'
Interval（时间间隔）	INTERVAL '15-3' YEAR TO MONTH, INTERVAL '22:06:5.5' HOUR TO SECOND

✓ 提示

- 使用语句CREATE TABLE和ALTER TABLE定义或更改列的数据类型，见第11章。
- 数据库设计人员应该慎重选择数据类型。错误地选择数据类型将导致无法将值插入列中；以及如果数据类型被更改，会造成数据丢失。
- SQL:2003舍弃了SQL-92的位串（bit-string）数据类型（BIT和BIT VARYING），进而支持二进制大对象。位串支持比BLOB小的二进制数据项。
- **DBMS** SQL标准将数据类型的实现细节留给DBMS供应商。因此，SQL数据类型和特定的DBMS数据类型不是直接对应，即使二者数据类型有同样的名称。在下面每一个数据类型小节的提示中，给出了等价的或类似的DBMS数据类型。某些DBMS数据类型有与SQL标准数据类型名称相对应的同义词。

3

68

69

70

3.5 字符串类型

使用字符串数据类型表示文本。

字符串（或串）有以下特征。

- 它是零个或多个字符的有序序列。
- 它的长度可以固定或可变。
- 它是区分大小写的（排序时'A'在'a'之前）。
- 在SQL语句中，字符串被单引号环绕。
- 它是表3-4所列出的类型之一。

表3-4 字符串类型

类 型	描 述
CHARACTER	表示固定个数的字符。以CHARACTER(<i>length</i>)形式指定列中存储的字符串的字符可以长达 <i>length</i> , 这里的长度 <i>length</i> 是一个大于或等于1的整数; 最大的 <i>length</i> 依赖于DBMS。当在CHARACTER(<i>length</i>)列中存储比 <i>length</i> 短的字符串时, DBMS用空格填充字符串的末尾以产生有精确长度的字符串。例如, CHARACTER(6)串'Jack'被存为'Jack'。CHARACTER和CHAR是同义词
CHARACTER VARYING	表示可变个数的字符。以CHARACTER VARYING(<i>length</i>)形式指定在列中存储的串中的字符可以长达 <i>length</i> , 这里的长度 <i>length</i> 是一个大于或等于1的整数; 最大的 <i>length</i> 依赖于DBMS。和CHARACTER不同, 当在CHARACTER VARYING(<i>length</i>)列中存储一个比 <i>length</i> 短的字符串时, DBMS直接存储而无需空格填充。例如, CHARACTER VARYING(6)串'Jack'被存为'Jack'。CHARACTER VARYING、CHAR VARYING和VARCHAR是同义词
NATIONAL CHARACTER	这种数据类型除了保存标准的多字节字符或Unicode字符(参见本章的提要栏)以外, 和CHARACTER是一样的。在SQL语句中, NATIONAL CHARACTER字符串除了在第一个引号的前面有一个N以外和CHARACTER是一样的, 例如N'8 本'. NATIONAL CHARACTER、NATIONAL CHAR和NCHAR是同义词
NATIONAL CHARACTER VARYING	这种数据类型除了保存标准的多字节字符或Unicode字符以外(参见NATIONAL CHARACTER)和CHARACTER VARYING是一样的。NATIONAL CHARACTER VARYING、NATIONAL CHAR VARYING和NCHAR VARYING是同义词
CLOB	字符大型对象(Character large object, CLOB)用于保存巨大量文本的图书馆数据库。例如, 单一的CLOB值可以保存一个完整的网页、书或者基因序列。CLOB不能被用作关键字或用在索引中, 并且支持的函数和操作符比CHAR和VARCHAR支持的要少。在宿主语言里, CLOB被唯一的定位符(游标)的值引用, 避免了在客户/服务器网络传输全部CLOB的开销。CLOB和CHARACTER LARGE OBJECT是同义的
NCLOB	国家字符大型对象(The national character large object, NCLOB)类型除了保存标准的多字节字符或Unicode字符(参见NATIONAL CHARACTER)以外, 和CLOB是一样的。NCLOB、NCHAR LARGE OBJECT、NATIONAL CHARACTER LARGE OBJECT是同义词

Unicode (统一字符编码标准)

计算机通过在内部指派给它们数字值, 来存储字符(字母、数字、标点、控制字符和其他符号)。编码决定了字符对数字值的映射。不同的语言和计算机操作系统使用不同的编码。标准的美国英语使用ASCII编码, 它给128 (2^7) 个不同的字符指派值, 并不多甚至不够保存现代欧洲语言中使用的所有拉丁字符, 远远少于所有中文汉字。

Unicode是单一的字符集合, 表示了世界上几乎所有的书写语言字符。Unicode能够编码多达4300 000万 (2^{32}) 个字符(使用UTF-32编码)。Unicode委员会发展并维护Unicode标准。要获得包含实际映射的Unicode标准, 可以访问www.unicode.org取得网络或印刷版本。

✓ 提示

- 两个连续的单引号表示字符串里面的一个单引号字符。例如, 输入'don''t'表示don't。双引号字符(")是单独的字符, 并且不需要这种特殊处理。
- 字符串的长度是零到*length*之间的一个整数。一个没有字符的串''(两个中间没有插入空格的单引号)称为空串(empty string)或长度为零的串(zero-length string)。空串被认为是长度为

零的VARCHAR。

- DBMS对固定长度的串进行排序和操作，通常要比变长的串快一些。
- 保持字符列尽可能短要胜过为它们留下“增长空间”。短列的排序和分组要比长列快一些。
- SQL:1999引入SQL语言的CLOB和NCLOB（但当时大多数DBMS已经有了类似的数据类型）。
- **DBMS** 表3-5所示为DBMS的字符串和类似的数据类型，参见DBMS文档中的“长度限制”和“使用限制”。

Oracle将空串作为空值处理，参见3.14节。

在MySQL的ANSI_QUOTES模式中，串字面量只可以被单引号引起，用双引号引起的串将被解释为标识符。

表3-5 DBMS的字符串类型

DBMS	类 型
Access	text、memo
SQL Server	char、varchar、text、nchar、nvarchar、ntext
Oracle	char、varchar2、nchar、nvarchar2、clob、nclob、long
DB2	char、varchar、long varchar、clob、dbclob、graphic、vargraphic、long vargraphic
MySQL	char、varchar、national char、national varchar、tinytext、text、mediumtext、longtext
PostgreSQL	char、varchar、text

3.6 二进制大型对象类型

二进制大型对象（binary large object，BLOB）数据类型用于存储二进制数据。

BLOB有以下特点。

- 类型名是BLOB或者BINARY LARGE OBJECT。
- 和存储长字符串的CLOB不同，BLOB存储长的字节序列。这两种数据类型是不兼容的。
- BLOB主要用于存储大数据量的多媒体数据（如图形、照片、音频或视频）、科学数据（如MRI图像或天气地图）及技术数据（如工程图样）。
- BLOB不能被用做关键字或索引，并且比其他数据类型支持的函数和操作符要少。因为一个BLOB小于另一个BLOB是没有意义的，所以BLOB只能进行等于（=）或者不等于（<>）的比较；而且也不能将BLOB在DISTINCT中使用，或在GROUP BY或ORDER BY子句以及列函数中使用。
- 在宿主语言里，BLOB被唯一的定位符（游标）的值引用，避免了跨越客户/服务器网络传输全部BLOB。

✓ 提示

- DBMS不会去解释BLOB，它们的含义由应用程序来决定。
- 二进制串字面量以十六进制（hexadecimal，缩写为hex）格式给出。十六进制数使用数字0~9以及字母A~F（大写形式或者小写形式）来表示。一个十六进制数等价于4 bit。在SQL语句中，十六进制数在它的第一个引号前有一个X，并且不能插入空格。例如，十六进制串X'4B'对应于01001011或串B'01001011'。
- SQL:1999在SQL语言中引入了BLOB（但当时大多数DBMS已经有了类似的数据类型）。

- DBMS** 表3-6所示为BLOB和类似的DBMS类型，参见DBMS文档中的“长度限制”和“使用限制”。

表3-6 DBMS的BLOB类型

DBMS	类 型
Access	ole object, attachment
SQL Server	binary, varbinary, image
Oracle	raw, long raw, blob, bfile
DB2	blob
MySQL	binary, varbinary, tinyblob, blob, mediumblob, longblob
PostgreSQL	bit, bit varying, bytea, oid

3.7 精确数字类型

精确数字数据类型用于表示精确的数字值。

精确数字值有以下特点。

- 它可以是负数、零或者正数。
- 它是整数或小数。整数是不用小数点表示的数，如-42、0、62 262。小数是在小数点的右侧有阿拉伯数字的数，如-22.06、0.0、0.000 3和12.34。
- 它有固定的精度和小数位数。精度是用于表示数字中有效数字的个数，它是小数点左侧和右侧全部数字的总个数。小数位数是小数点右侧数字的个数。很显然，小数位数不可以超过精度。要表示整数，应将小数的位数设置为零。参见本节提示中的例子。
- 它是表3-7所列出的类型之一。

表3-7 精确数字类型

类 型	描 述
NUMERIC	表示一个小数数字，存储在定义为NUMERIC(<i>precision</i> [, <i>scale</i>])的列中。精度大于或等于1；精度 <i>precision</i> 的最大精度依赖于DBMS。小数位数 <i>scale</i> 是0到精度之间的一个值。如果小数位数省略，默认值是零（它使数字成为一个有效的整数）
DECIMAL	这种数据类型类似于NUMERIC，并且某些DBMS定义了它们的等价物。不同的是，DBMS可以选择大于在DECIMAL(<i>precision</i> [, <i>scale</i>])中指定的精度。因此精度定义了最小精度，而不是像在NUMERIC中是精确的精度。DECIMAL和DEC是同义词
INTEGER	表示一个整数（integer）。能够存储在整数列中的最小值和最大值依赖于DBMS。INTEGER没有参数。INTEGER和INT是同义词
SMALLINT	这种数据类型除了可以保存一个依赖于DBMS的、更小范围的值以外，和INTEGER（整数）是一样的。SMALLINT没有参数
BIGINT	这种数据类型除了可以保存一个依赖于DBMS的、更大范围的值以外，和INTEGER（整数）是一样的。BIGINT没有参数

✓ 提示

- 表3-8所示为123.89如何以不同精度和小数位数的值进行存储。
- 不能用引号环绕数字字面量。
- 如果数字不涉及算术运算，则应该将数字存储为串。例如，将电话号码、邮政编码、美国社会保险号保存为串。这种技术可以保存空格并防止数据丢失。如果以数字而不是串方式存储

邮政编码'02116'，将丢失开头的0。

- 只包含整数的运算要比包含小数和浮点数的运算快得多。
- DBMS** 表3-9所示为精确数字和类似的DBMS类型。参见DBMS文档中的“长度限制”和“使用限制”。DBMS经常接受不能支持的类型名，将它们转换为合适、可支持的类型。例如，Oracle将INT转换为NUMBER(32)。

DBMS以16位的值(-32 768~32 767)实现SMALLINT，以32位的值实现INTEGER(-2 147 483 648~2 147 483 647)，以64位的值(百万的三次方)表示BIGINT。SQL:2003为SQL语言引入了BIGINT(但当时大多数DBMS已经有了类似的数据类型)。

表3-8 以123.89为例的精度和小数位数

定义的形式	存储的形式
NUMERIC(5)	124
NUMERIC(5,0)	124
NUMERIC(5,1)	123.9
NUMERIC(5,2)	123.89
NUMERIC(4,0)	124
NUMERIC(4,1)	123.9
NUMERIC(4,2)	Exceeds precision (超出精度)
NUMERIC(2,0)	Exceeds precision (超出精度)

表3-9 DBMS的精确数字类型

DBMS	类 型
Access	byte、decimal、integer、long integer
SQL Server	bigint、int、smallint、tinyint、decimal、numeric
Oracle	number、float
DB2	smallint、integer、bigint、decimal
MySQL	tinyint、smallint、mediumint、int、bigint、decimal
PostgreSQL	smallint、integer、bigint、numeric

3.8 近似数字类型

使用近似数字数据类型表示近似的数字值。

近似数字值有以下特点。

- 它可以是负数、零或者正数。
- 它是浮点数(实数)的近似值。
- 它通常用于表示在技术、科学、统计和财务运算中，非常小或非常大的数量。
- 它被使用在科学记数法(scientific notation)中。科学记数法中的数字被写作小数乘以10的整数次方的形式。大写的E是求幂符号，例如， $2.5\text{E}2 = 2.5 \times 10^2 = 250$ 。尾数表示有效数字的部分(这里是2.5)，指数是10的权(这里是2)。尾数和指数每一个都可以有符号： $-2.5\text{E}-2 = -2.5 \times 10^{-2} = -0.025$ 。
- 它有固定的精度但没有显式的小数位数(指数的符号和数量级决定了固有的小数位数)。精度使用(二进制)位存储尾数。要将二进制精度转换为十进制精度，乘以精度0.301 03。要将十进制精度转换为二进制精度，乘以十进制精度3.321 93。例如，24位可产生7位十进制数的精度。

53位可以产生15位十进制数字的精度。

- 它是表3-10所示的类型之一。

表3-10 近似数字类型

类 型	描 述
FLOAT	表示存储在以FLOAT(<i>precision</i>)形式定义的列中的浮点数字。精度大于或等于1，并以位值表示（不是数字值），最大精度依赖于DBMS
REAL	这种数据类型除了DBMS定义的精度以外，和FLOAT是一样的。REAL（实数）通常称为单精度数字。REAL没有参数
DOUBLE PRECISION	这种数据类型除了DBMS定义的必须大于REAL（实数）的精度以外，和FLOAT是一样的。DOUBLE PRECISION（双精度）没有参数

✓ 提示

- 不要用引号环绕数字字面量。

- DBMS** 表3-11所示为近似数字和类似的DBMS类型。参见DBMS文档中的“长度限制”和“使用限制”。DBMS经常接受不能实现的类型名，将它们转换为合适、可支持的类型。例如，PostgreSQL将float转换为double precision。

表3-11 DBMS的近似数字类型

DBMS	类 型
Access	single、double
SQL Server	float、real
Oracle	binary_float、binary_double
DB2	real、double
MySQL	float、double
PostgreSQL	real、double precision

3.9 布尔类型

布尔数据类型用于存储真值。

布尔值有以下特点。

- 类型名是BOOLEAN。

- 真值是用Boolean字面量TRUE、FALSE和UNKNOWN表示的True（真）、False（假）和Unknown（未知），真值将在4.6节中介绍。

- 空值(null)等价于Unknown真值，实际上通常被用于替代Unknown（大多数DBMS的布尔类型不接受字面量UNKNOWN），参见3.14节。

✓ 提示

- 不能用引号环绕布尔字面量。

- SQL:1999引入了SQL语句的BOOLEAN（布尔类型）。

- DBMS** 表3-12所示为布尔类型和类似的DBMS类型。参见DBMS文档中的“长度限制”和“使用限制”。在没有布尔类型可用的地方，可以使用一个位数或者小的整数来实现该数据类型。SQL程序员经常使用数字值表示真值，0意味着false（假）、1（或者任意非0的数字）意

意味着true（真），null意味着unknown（未知）。

表3-12 DBMS的布尔类型

DBMS	类 型
Access	yes/no
SQL Server	bit
Oracle	number(1)
DB2	decimal(1)
MySQL	boolean
PostgreSQL	boolean

3.10 日期和时间类型

使用日期和时间数据类型表示日期和时间。

日期和时间值有以下特点。

- 它们是依据于以前称为格林尼治时间的世界协调时间（Universal Coordinated Time, UTC）定义的。SQL标准要求每一个SQL会话有一个相对于UTC的默认偏差（被用于会话的持续时间）。例如，加利福尼亚的旧金山属-8时区。
- 阳历（gregorian calendar）的规则决定了日期（date）值是如何构成的，DBMS将拒绝不能识别为日期的值。
- 时间（time）值是基于24小时时钟的，也称为军用时间（military time）。例如，使用13:00而不是1:00 p.m.
- 连字符（-）分隔日期的各个部分，冒号（:）分隔时间的各个部分。当组合日期和时间时用空格分隔。
- 它是表3-13所示的类型之一。

表3-13 日期和时间类型

类 型	说 明
DATE	表示日期。DATE（日期）存储在被定义为DATE的、由3个整数段——YEAR、MONTH和DAY组成，并被格式化为 <code>yyyy-mm-dd</code> （长度为10，如 <code>2006-03-17</code> ）的列中。表3-14所示为各个域的有效值。DATE没有参数
TIME	表示一天中的时间。TIME（时间）存储在被定义为TIME、由3个整数段——HOUR、MINUTE和SECOND组成，并被格式化为 <code>hh:mm:ss</code> （长度为8，如 <code>22:06:57</code> ）的段中。可以用 <code>TIME(precision)</code> 定义小数的秒。精度 <code>precision</code> 是一个大于或等于0的小数的位数。最大精度（至少6位）依赖于DBMS。HOUR（时）和MINUTE（分）是整数，SECOND（秒）是十进制小数。格式是 <code>hh:mm:ss.ssss...</code> （长度9加上小数数字，如 <code>'22:06:57.1333'</code> ）。表3-14列出了各个段的有效值
TIMESTAMP	表示以空格分隔的DATE（日期）和TIME（时间）的组合。TIMESTAMP（时间戳）格式是 <code>yyyy-mm-dd hh:mm:ss</code> （长度为19，如 <code>2006-03-17 22:06:57</code> ）。可以用 <code>TIMESTAMP(precision)</code> 定义小数的秒。格式是 <code>yyyy-mm-dd hh:mm:ss.ssss...</code> （长度是20加上小数数字）
TIME WITH TIME ZONE	这种类型除了增加一个指出与UTC的小时偏差的字段——TIME_ZONE_OFFSET（时区偏差）以外，和TIME是一样的。TIME_ZONE_OFFSET时区偏差被格式化为INTERVAL HOUR TO MINUTE（见下一节），可以包含表3-14列出的值。可以给TIME追加TIME_ZONE（时区） <code>time_zone_offset</code> 来分配时区值（例如， <code>22:06:57 AT TIME ZONE -08:00</code> ）；也可以选择追加AT LOCAL，以指出时区对于会话是默认的（如 <code>22:06:57 AT LOCAL</code> ）。如果省略AT子句，所有的时间默认为AT LOCAL

(续)

类 型	说 明
TIMESTAMP WITH TIME ZONE	这种数据类型除了增加一个指出与UTC的小时偏差的字段——TIME_ZONE_OFFSET（时区偏差）以外，和TIMESTAMP是一样的。语法规则除了必须包括日期（如2006-03-17 22:06:57 AT TIME ZONE -08:00）以外，和TIME WITH TIME ZONE是一样的

✓ 提示

- 要获得系统时间，参见5.11节。
- 如果有相同的字段（表13-14），可以比较两个datetime（日期和时间）的值，参见4.5节和5.10节。

表3-14 日期和时间字段

字 段	有 效 值
YEAR	0001 to 9999
MONTH	01 to 12
DAY	01 to 31
HOUR	00 to 23
MINUTE	00 to 59
SECOND	00 to 61.999... (参阅本节中的提示)
TIME_ZONE_OFFSET	-12:59 to +13:00

- SECOND（秒）字段可以接受直到61.999…（而不是59）的值，允许将闰秒插入到一个特定的时间里，以保持地球时钟和恒星时间的同步性。
- datetime（日期和时间）字面量是datetime类型名，后面跟着空格，再跟着一个被单引号环绕的datetime（日期和时间）值。如DATE'yyyy-mm-dd'，TIME'hh:mm:ss'和TIMESTAMP'yyyy-mm-dd hh:mm:ss'。
- 标准的SQL不能处理B.C.E./B.C.（公元前）的日期，但所用的DBMS也许可以处理。
- timestamp（时间戳）经常被用于标记和它们所出现的行有关联的事件。例如在MySQL中，timestamp列被用于记录UPDATE操作的日期和时间。
- 数据类型TIME WITH TIME ZONE没有意义，因为在现实世界中时区没有意义，除非它们和一个日期关联（因为时区偏差在全年中是变化的），可以使用TIMESTAMP WITH TIME ZONE。
- 也可以参见15.11节。
- **DBMS** 表3-15所示为日期和时间类型以及类似的DBMS类型，参阅DBMS文档中的“长度限制”和“使用限制”。

DBMS允许输入以month-day-year、day-month-year或其他格式输入日期值和基于12小时时钟(a.m./p.m.)的时间值。显示的日期和时间格式可以与输入的不同。

在Microsoft Access中，不是用引号而是用#字符环绕日期和时间字面量，并且省略了数据类型名前缀。例如，标准的SQL日期DATE '2006-03-17'和Access的日期#2006-03-17#是等价的。

在Microsoft SQL Server中，省略了日期和时间字面量的数据类型名前缀。例如，标准的SQL日期DATE '2006-03-17'和SQL Server的日期'2006-03-17'是等价的。

在DB2中，省略了日期和时间字面量的数据类型名前缀。例如，标准的SQL日期DATE '2006-03-17'

和DB2的日期'2006-03-17'是等价的。

表3-15 DBMS的日期和时间类型

DBMS	类 型
Access	date/time
SQL Server	datetime、smalldatetime
Oracle	date、timestamp
DB2	date、time、timestamp
MySQL	date、time、datetime、timestamp、year
PostgreSQL	date、time、timestamp

3.11 时间间隔类型

DBMS DBMS对于标准SQL时间间隔类型的一致性是有欠缺或不存在的，而本节内容也许在实际中没有用处。DBMS有它们自己的扩展数据类型和函数，来计算时间间隔和执行日期和时间计算。

使用时间间隔数据类型表示时间值的集合或者时间的跨度。

时间间隔的值有以下特点。

- 它存储两个日期和时间值之间的时间数量。例如，09:00和13:30之间的时间间隔是04:30（4小时30分钟）。
- 它可以被用于增加或者减少日期和时间的值，参见5.10节。
- 它与日期和时间值有相同的字段(YEAR、HOUR和SECOND等)，但数字可以有一个+号(向前)或者-号(向后)以指示时间的方向。字段分隔符和datetime(日期和时间)值是一样的。
- 有两种类型：year-month时间间隔和day-time时间间隔。year-month时间间隔以年和整数月的形式表示时间间隔，day-time时间间隔以日、时、分和秒的形式表示时间间隔。
- 它有单一字段或多个字段的标识符。单一字段标识符按YEAR、MONTH、DAY、MINUTE或SECOND的形式定义，多个字段标识符按下面的形式定义。

start_field TO end_field

*start_field*是YEAR、DAY、HOUR或MINUTE，*end_field*是YEAR、MONTH、DAY、HOUR、MINUTE或SECOND。*end_field*必须是比*start_field*小的时间段。

例如，以INTERVAL HOUR形式定义的单一字段列，可以存储如“4 hours”或“25 hours”的时间间隔。以INTERVAL DAY TO MINUTE形式定义的多个字段列，可以存储如“2 days, 5 hours, 10 minutes”的时间间隔。

- 单一字段可以有指定字段长度(位置个数)的精度，如INTERVAL HOUR(2)。如果省略，则默认精度是2。SECOND字段可以有附加的指定小数点右侧位数的小数精度，如INTERVAL SECOND(5,2)。如果省略，则小数精度默认是6。
- 多个字段可以有针对*start_field*而非*end_field*的精度(除非*end_field*是SECOND，这种情况下可以有小数精度)。例如，INTERVAL DAY(3)TO MINUTE和INTERVAL MINUTE(2)TO SECOND(4)。
- 它是表3-16所示的类型之一。

表3-16 时间间隔类型

类 型	描 述
Year-month	这种时间间隔可以只包含year值、month值或两者都包含。有效的列的类型是INTERVAL YEAR、INTERVAL YEAR(<i>precision</i>)、INTERVAL MONTH、INTERVAL MONTH(<i>precision</i>)、INTERVAL YEAR TO MONTH或INTERVAL YEAR(<i>precision</i>)TO MONTH
Day-time	这种时间间隔可以只包含day值、hour值、minute值、second值或者是它们的组合。一些有效列类型的例子是INTERVAL MINUTE、INTERVAL DAY(<i>precision</i>)、INTERVAL DAY TO HOUR、INTERVAL DAY(<i>precision</i>)TO SECOND和INTERVAL MINUTE(<i>precision</i>)TO SECOND(<i>frac_precision</i>)

✓ 提示

- 时间间隔字面量是单词INTERVAL，后跟一个空格，再后跟用单引号环绕的时间间隔的值。例如，INTERVAL '15-3' (15年3个月) 和INTERVAL '22:06:5.5' (22小时6分5.5秒)。
- 时间间隔字面量是单词INTRVAL，后跟一个空格，再后跟用单引号环绕的时间间隔的值，再后跟时间间隔限定符。例如，INTERVAL '15-3' YEAR TO MONTH(15年3个月)和INTERVAL '22:06:5.5' HOUR TO SECOND (22小时6分5.5秒)。
- 参见15.11节。
- **DBMS** 表3-17所示为时间间隔和类似的DBMS类型，参阅DBMS文档中的“长度限制”和“使用限制”。

表3-17 DBMS的时间间隔类型

DBMS	类 型
Access	不支持
SQL Server	不支持
Oracle	interval year to month, interval day to second
DB2	不支持
MySQL	不支持
PostgreSQL	interval

3.12 唯一标识符

唯一标识符用于产生标识行的主键值（见2.2节）。标识符可以是全局唯一的（在全部上下文中非常大、唯一的随机数），或者只在一个特定的表中是唯一的（简单的序列数1, 2, 3…）。表3-18所示为DBMS的唯一标识符和属性，参阅DBMS文档中的“长度限制”和“使用限制”。SQL标准将带有自动增量值的列称为标识列，参见15.2节。

表3-18 唯一标识符

平 台	类型或属性
Standard SQL	IDENTITY
Access	autonumber、replication id
SQL Server	uniqueidentifier、identity
Oracle	rowid、urowid、sequences
DB2	Identity columns and sequences
MySQL	auto_increment attribute
PostgreSQL	serial、bigserial、uuid

全局唯一标识符

全局范围内唯一的ID称为全局唯一标识符（Universally Unique Identifier，UUID或Globally Unique Identifier，GUID）。当有定义为UUID数据类型的列时，DBMS按照ISO/IEC 9834-8:2005（www.itu.int/ITU-T/studygroups/com17/oid.html）或IETF RFC 4122（<http://tools.ietf.org/html/rfc4122>）自动在每一个新行中产生一个随机的UUID。

标准格式的UUID，如：

a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11

字母实际上是十六进制数。DBMS也许会使用一种替代的形式——十六进制大写字母，用大括号环绕，且省略连字符。UUID无法从技术上保证唯一性，但是产生重复ID的概率是非常小的，以至于它们被认为是单一的。更多的信息，可以参见http://en.wikipedia.org/wiki/Universally_Unique_Identifier。

3

82

用户定义数据类型

Microsoft SQL Server、Oracle、DB2和PostgreSQL可以产生用户定义类型（user-defined type，UDT）。最简单的UDT是一个带有附加检查和其他约束的、标准或内置的数据类型（CHARACTER、INTEGER等）。例如，可以定义数据类型marital_status，作为一个只允许值S、M、W、D或NULL（对应于单身、已婚、寡居、离婚或未知的）单字符的CHARACTER数据类型。更复杂的UDT类似于面向对象编程语言（如Java）中的类。可以定义一次UDT而在多个表中使用，无需在使用它的每个表中重复定义。查阅DBMS文档中的用户定义类型。在标准SQL中，UDT用语句CREATE TYPE创建。

3.13 其他数据类型

除了前面几节介绍的以外，标准的SQL还定义了其他数据类型，其中某些在实际中很少实现或使用，如ARRAY、MULTISET、REF和ROW。更有用的是，在不同DBMS中可用的扩展（非标准的）数据类型。依赖于所用的DBMS，数据类型用于：

- 网络和内部网地址
- 存储在数据库外的文件的链接
- 货币金额
- 地理（空间）坐标
- 数组和其他集合
- XML（可扩展标记语言）
- 全文检索
- 特定表的枚举值

83

3.14 空值

当数据不完整的时候，可以使用空值（null）表示缺失的或未知的值。

空值有以下特点。

- 在SQL语句中，关键字NULL表示空值。

- 空值用于表示也许绝不可知、也许以后才能决定或者不适用的值（把空值看作缺失值的记号或标记，而不是值）。
- 空值和0、只有空格的串或空串('')不同。在列**price**中的空值并不意味着该项没有价格或者它的价格是0，而是意味着价格是未知的或者还没有被设置（Oracle是一个特例，表示空串，参阅本节中的DBMS提示）。
- 空值可以出现在没有非空(**NOT NULL**)或主键完整性约束的任何数据类型的列中（见第11章）。在候选键中不允许使用空值。
- 要检测空值，参见4.10节。
- 空值彼此既不是相等，也不是不相等。不能确定一个空值和另一个值（包括另一个空值）是否匹配。因此，表达式`NULL = any_value`、`NULL <> any_value`、`NULL = NULL`和`NULL <> NULL`，既不是true（真）也不是false（假），而是unknown（未知），参见4.6节。
- 尽管空值彼此绝对不会相等。**DISTINCT**在特定列中将空值作为完全相同的情况来处理，参见4.3节。
- 当对包含空值的列排序时，空值是大于还是小于所有的非空值，这依赖于DBMS，参见4.4节。
- 空值通过计算传播。任何包含空值的算术表达式或操作的结果为空值，`(12*NULL)/4`是空值，参见第5章。
- 大多数聚合函数（比如**SUM()**、**AVG()**和**MAX()**）忽略空值，**COUNT(*)**除外，参见第6章。
- 如果使用包含空值的**GROUP BY**子句对列分组，则所有的空值将被放入单一的组中，参见6.9节。
- 空值影响联结(join)的结果，参见7.3节。
- 在子查询中空值将引发问题，参见8.6节。

✓ 提示

- 当解释包含有空值的结果时要注意。空值会引发很多问题和麻烦（这里列出了一些重要的），一些数据库专家主张用户尽可能少地使用它们或者根本就不使用它们（代之使用默认值或某些其他缺失数据方案）。不过，如果没有完全理解空值，就不可能成为一个有能力的SQL程序员。
- 也可以参见5.15节和5.16节。
- 可以为空意味着列中允许包含空值。使用**CREATE TABLE**或**ALTER TABLE**语句（见第11章）设置列的可空性。
- “空值”是不准确的说法，空值表示值的缺失。
- 不要把关键字**NULL**放入引号中。DBMS将把它视为字符串'NULL'，而不是空值。
- 可以从不允许空值的列中得到空值。在表**authors**中的列**au_id**不允许空值，但图3-3中的SELECT语句对于最大的**au_id**返回一个空值。
- 如果因为实际值没有意义（而不是未知），列中出现空值可以将列从表中分割，进而和其他表有一对一联系。在图3-4中，原始表**employees**有定义了雇员销售佣金的列**commission**。因为大多数雇员不是售货员，所以**commission**通常包含空值。为避免空值泛滥，将**commission**移到它自己的表中。

```

SELECT MAX(au_id)
  FROM authors
 WHERE au_lname = 'XXX';

MAX(au_id)
-----
NULL

```

图3-3 从不可为空的列中得到空值

employees		
emp_id	emp_name	commission
E01	Eli McLemore	NULL
E02	Monty Wendt	0.25
E03	Damien Shaw	NULL
E04	Russell Sager	NULL
E05	Jill Stallworth	NULL
E06	Pamela Gant	0.08



employees		commissions	
emp_id	emp_name	emp_id	commission
E01	Eli McLemore	E02	0.25
E02	Monty Wendt	E06	0.08
E03	Damien Shaw		
E04	Russell Sager		
E05	Jill Stallworth		
E06	Pamela Gant		

图3-4 通过将原始表（顶部的）分割成一对联系（底部的），消除空值

- **DBMS** 在结果中空值的显示因DBMS而异。例如，空值可能显示为NULL、(NULL)、<NULL>、-或者空格。

Oracle将空串('')作为空值处理。这种处理方式也许在后续版本中将不再沿用。Oracle建议在SQL代码中将空串和空值区别对待。因为这种行为可能会引发代码版本在DBMS间的转换问题。例如，在示例数据库里，表authors中的列au_fname被定义为NOT NULL。在Oracle中，作者Kellsey（作者A06）的名是一个空格(' '); 在其他的DBMS中，名是一个空串('')。关于示例数据库更多的信息，参见2.6节。



本章介绍SQL的重点内容——SELECT语句。大多数SQL任务会涉及使用这一语句（虽有时很复杂）来检索和操作数据。SELECT从数据库里的一个或多个表中检索行、列和派生值。它的语法是：

```
SELECT columns
  FROM tables
  [JOIN joins]
  [WHERE search_condition]
  [GROUP BY grouping_columns]
  [HAVING search_condition]
  [ORDER BY sort_columns];
```

✓ 提示

□ 斜体字表示代码中必须用值替代的变量，方括号（[]）表示可选的子句或项，参见前言中的“排版约定”和“语法约定”。

本章介绍SELECT、FROM、ORDER BY和WHERE，第6章介绍GROUP BY和HAVING，第7章介绍JOIN。按照约定，在查询中只调用SELECT语句，因为它返回一个结果集。DBMS文档和其他书中的查询也许会用到任意一条SQL语句。虽然SELECT语句是强大并且安全的，但它并不危险，因为你不会用它去增加、更改、删除数据或数据库对象（关于不安全的内容从第10章开始介绍）。

87

4.1 使用 SELECT 和 FROM 检索列

在最简单的形式中，SELECT语句从一个表中检索列，可以检索一列、多列或全部的列。SELECT子句列出需要显示的列，FROM子句指定从中提取列的表。

⇒ 从一个表中检索一列

输入：

```
SELECT column
  FROM table;
```

*column*是列的名称，*table*是包含*column*的表的名称（代码4-1和图4-1）。

代码4-1 列出作者所在的城市。结果见图4-1

```
SELECT city
  FROM authors;
```

city
Bronx
Boulder
San Francisco
San Francisco
New York
Palo Alto
Sarasota

4

图4-1 运行代码4-1的结果

⇒ 从一个表中检索多列

输入：

```
SELECT columns
  FROM table;
```

*columns*是两个或多个以逗号分隔的列的名称，*table*是包含*columns*的表的名称（代码4-2和图4-2）。

88

列将按照在*columns*中给出的顺序显示，而不是按照*table*中定义的顺序显示。

代码4-2 列出每个作者的名、姓、所在城市和州。结果见图4-2

```
SELECT au_fname, au_lname, city, state
  FROM authors;
```

au_fname	au_lname	city	state
Sarah	Buchman	Bronx	NY
Wendy	Heydemark	Boulder	CO
Hallie	Hull	San Francisco	CA
Klee	Hull	San Francisco	CA
Christian	Kells	New York	NY
	Kellsey	Palo Alto	CA
Paddy	O'Furniture	Sarasota	FL

图4-2 运行代码4-2的结果

⇒ 从一个表中检索全部的列

输入：

```
SELECT *
  FROM table;
```

*table*是表的名称（代码4-3和图4-3）。

89

列将按照*table*中定义的顺序显示。

代码4-3 列出表authors的全部列。结果见图4-3

```
SELECT *
FROM authors;
```

au_id	au_fname	au_lname	phone	address	city	state	zip
A01	Sarah	Buchman	718-496-7223	75 West 205 St	Bronx	NY	10468
A02	Wendy	Heydenmark	303-986-7020	2922 Baseline Rd	Boulder	CO	80303
A03	Hallie	Hull	415-549-4278	3800 Waldo Ave, #14F	San Francisco CA		94123
A04	Klee	Hull	415-549-4278	3800 Waldo Ave, #14F	San Francisco CA		94123
A05	Christian	Kells	212-771-4680	114 Horatio St	New York	NY	10014
A06		Kellsey	650-836-7128	390 Serra Mall	Palo Alto	CA	94305
A07	Paddy	O'Furniture	941-925-0752	1442 Main St	Sarasota	FL	34236

图4-3 运行代码4-3的结果

✓ 提示

- SELECT和FROM子句对于从表中检索列是必需的，其他子句是可选的。
- 闭包保证每一条SELECT子句的结果是一个表，参见2.1节中的提示。
- 因为有两个作者居住在旧金山，所以图4-1中的结果包含重复行。要消除重复行，参见4.3节。
- 结果中行的顺序可能与源表中行的顺序不同，参见4.4节。
- 使用NULL表示表中或结果中的空值，参见3.14节（代码4-4和图4-4）。
- 第7章至第9章介绍了如何从多个表中检索列。

代码4-4 列出每一个出版社所在的城市、州和国家。结果见图4-4

```
SELECT city, state, country
FROM publishers;
```

- 所有结果显示原始的、未格式化的值。例如，货币数量前缺少货币符号，数字的小数位数也许并不合适。虽然DBMS有非标准的函数可以在查询结果中格式化数字、日期和时间，最好使用报表工具（而不是数据检索工具）来格式化数据。例如，参见Microsoft SQL Server的datename()函数或MySQL的date_format()函数。

- 使用SELECT *经常是有风险的，因为表中列的个数或顺序一旦更改就有可能引起程序执行失败。同样地，SELECT *将无法被不熟悉表中列的人所理解。与在SELECT子句中给出特定列名的查询相比，SELECT *被跨越网络传递不需要的数据拖住，占用大量资源（要查看表是如何定义的，而不是列出它的行，参见10.1节）。
- 从一个表中选择确定列的操作，称为投影。

city	state	country
New York	NY	USA
San Francisco	CA	USA
Hamburg	NULL	Germany
Berkeley	CA	USA

图4-4 运行代码4-4的结果。列state在德国不适用。NULL表示一个空值，这和不可见的值（如空串或空格串）是截然不同的

4.2 使用 AS 创建列的别名

至此，在查询结果中，DBMS对于列的标题都是使用默认值（在结果中，列的默认标题源自表中定义的列名）。可以使用AS子句创建列的别名。列的别名是为了控制在结果中列的标题如何显示而指定的另一个名称（标识符）。如果列名不方便记忆，难以输入，或太长、太短，则可以使用列的别名。

在SELECT语句的SELECT子句中，列的别名直接跟在列名的后面。如果别名是保留关键字或者包含空格、标点或特殊字符，则用单引号或双引号将别名括起来。如果别名是一个只包含字母、数字和下划线的非保留字，则可以省略引号。如果想让特定列保留默认标题，则省略它的AS子句。

» 创建列的别名

输入：

```
SELECT column1 [AS] alias1,
       column2 [AS] alias2,
       ...
       columnN [AS] aliasN
  FROM table;
```

*column1, column2, …, columnN*是列名，*alias1, alias2, …, aliasN*是相应列的别名，*table*是包含*column1, column2, …, columnN*的表的名称。

代码4-5显示了AS子句的语法变化。图4-5显示了代码4-5的运行结果。

代码4-5 AS子句定义了在结果中显示的列的别名。这条语句显示了AS语法的可选结构。在程序里，挑选一种结构，并一直使用。结果见图4-5

```
SELECT au_fname AS "First name",
       au_lname AS 'Last name',
       city AS City,
       state,
       zip 'Postal code'
  FROM authors;
```

在标准SQL和大多数DBMS中，关键字AS是可选的，但应该始终使用它，并用对别名加双引号，以便SQL代码更具有可移植性和可读性。按照这种语法约定，代码4-5等价于：

```
SELECT au_fname AS "First name",
       au_lname AS "Last name",
       city     AS "City",
       state,
       zip      AS "Postal code"
  FROM authors;
```

✓ 提示

- 列的别名不会更改表中的列名。
- 要确定在表中定义的列名，参见10.1节。

First name	Last name	City	state	Postal code
Sarah	Buchman	Bronx	NY	10468
Wendy	Heydemark	Boulder	CO	80303
Hallie	Hull	San Francisco	CA	94123
Klee	Hull	San Francisco	CA	94123
Christian	Kells	New York	NY	10014
	Kellsey	Palo Alto	CA	94305
Paddy	O'Furniture	Sarasota	FL	34236

图4-5 运行代码4-5的结果

- 如果对别名加引号，则可以使用保留关键字。例如，查询SELECT SUM(sales) AS "Sum" FROM titles; 使用了保留字SUM作为列的别名。关键字的内容参见3.1节和3.3节。
- AS也可以用于命名派生列（它的值由其他的表达式而不只是列名决定），参见5.1节。
- 也可以用AS创建表的别名，参见7.2节。
- **DBMS** Microsoft Access和PostgreSQL引用列时需要AS关键字。

Oracle和DB2以大写形式显示不加引号的列名和列的别名。SQL * Plus (Oracle的命令行处理器) 将列的别名截短为表中列定义中指定的字符长度。例如，列的别名"Postal code" 在CHAR(5)列中显示为Posta。

DBMS对于在别名中嵌入空格、标点和特殊字符有限制，查阅DBMS文档中的SELECT或者AS说明。

92

4.3 使用 DISTINCT 消除重复的行

列经常包含重复的值，需要结果中重复值只列出一次是很正常的。如果输入代码4-6列出作者居住的州，结果（图4-6）包含不需要的重复值。DISTINCT关键字可以从结果中消除重复的行。请注意，DISTINCT的结果列形成了候选键（如果它们不包含空值）。

代码4-6 列出每个作者居住的州。结果见图4-6

```
SELECT state
  FROM authors;
```

⇒ 消除重复的行

输入：

```
SELECT DISTINCT columns
  FROM table;
```

columns是一个或多个以逗号分隔的列名，table是包含columns的表的名称（代码4-7和图4-7）。

代码4-7 列出作者居住的不同的州。关键字DISTINCT消除了结果中重复的行。结果见图4-7

```
SELECT DISTINCT state
  FROM authors;
```

✓ 提示

- 如果SELECT DISTINCT子句包含多列，则所有列值的组合决定了行的唯一性。代码4-8的运行结果如图4-8所示，它有两列包含了重复的行。代码4-9的运行结果是图4-9，它消除了双列的重复值。

state

NY
CO
CA
CA
NY
CA
FL

图4-6 运行代码4-6的结果。结果中包含不需要的重复行CA和NY

state

NY
CO
CA
FL

图4-7 运行代码4-7的结果。结果中没有重复的CA和NY

代码4-8 列出作者居住的城市和州。结果见图4-8

```
SELECT city, state
  FROM authors;
```

代码4-9 列出作者居住的不同城市和州。结果见图4-9

```
SELECT DISTINCT city, state
  FROM authors;
```

- 虽然空值是未知的，彼此绝不相等，但 DISTINCT认为所有的空值是彼此相等的。无论遇到多少个空值，SELECT DISTINCT在结果中只返回一个空值，参见3.14节。
 - 虽然SELECT语句的语法包括ALL关键字选项，但在实际中很少看到ALL，这是因为它表示默认的行为：显示所有的（包括重复的）行。
- `SELECT columns FROM table;`
等价于
`SELECT ALL columns FROM table;`
语法关系是：
`SELECT [ALL | DISTINCT] columns
 FROM table;`
- 如果表有被正确定义的主键，那么因为所有的行是唯一的，所以SELECT DISTINCT * FROM table; 和 SELECT * FROM table; 将返回相同的结果。
 - 也可以参见6.8节和15.8节。
 - 对于DISTINCT操作，DBMS执行内部排序去识别并消除重复的行。排序是需要大量计算开销的，除非不得不做，否则不要使用DISTINCT。

city	state
Bronx	NY
Boulder	CO
New York	NY
Palo Alto	CA
San Francisco	CA
San Francisco	CA
Sarasota	FL

图4-8 运行代码4-8的结果。结果中包含重复的行，加利福尼亚州的旧金山出现了两次

city	state
Bronx	NY
Boulder	CO
New York	NY
Palo Alto	CA
San Francisco	CA
Sarasota	FL

图4-9 运行代码4-9的结果。它的唯一性由 citystate 的组合而不是任何一列的值决定

4.4 使用 ORDER BY 排序行

查询结果中行是无序的，因此行可以按任意顺序显示。这种情形的出现是因为关系模型认为行的顺序是和表的操作不相关的。可以使用ORDER BY子句按照升序（最小到最大）或降序（最大到最小）对特定列的行排序，参见本节中的“排序顺序”提要栏。ORDER BY子句经常是SELECT子句中最后一个子句。

⇒ 按一列排序

输入：

```
SELECT columns
  FROM table
 ORDER BY sort_column [ASC | DESC];
```

95

*columns*是一个或多个以逗号分隔的列名, *sort_column*是在其上对结果进行排序的列的列名, *table*是包含*columns*和*sort_column*的表的名称 (*sort_column*不是必须要在*columns*中列出)。用ASC定义升序或者用DESC定义降序。如果没有指定排序的方向, 默认为ASC (代码4-10和代码4-11、图4-10和图4-11)。

代码4-10 列出作者的名、姓、城市和州, 按姓的升序排序。ORDER BY执行默认的升序排序, 因此ASC关键字是可选的 (在实际中, ASC通常被省略)。结果见图4-10

```
SELECT au_fname, au_lname, city, state
  FROM authors
 ORDER BY au_lname ASC;
```

代码4-11 列出作者的名、姓、所在的城市和州, 按名的降序排序。DESC关键字是必需的。结果见图4-11

```
SELECT au_fname, au_lname, city, state
  FROM authors
 ORDER BY au_fname DESC;
```

⇒ 按多列排序

输入:

```
SELECT columns
  FROM table
 ORDER BY sort_column1 [ASC | DESC],
          sort_column2 [ASC | DESC],
          ...
          sort_columnN [ASC | DESC];
```

*columns*是一个或多个以逗号分隔的列名, *sort_column1*, *sort_column2*, ..., *sort_columnN*是在其上对结果进行排序的列的列名, *table*是包含*columns*和排序列的表的名称 (排序列不是必须要在*columns*中列出)。行首先按*sort_column1*排序, 然后对*sort_column1*中值相等的行再按*sort_column2*中的值排序, 以此类推。对于每一个排序列, 用ASC定义升序排序, 用DESC定义降序排序。如果没有定义排序的方向, 默认为ASC (代码4-12和图4-12)。

代码4-12 列出作者的名、姓、所在城市和州。先按州的升序排序, 再按城市的降序排序。结果见图4-12

```
SELECT au_fname, au_lname, city, state
  FROM authors
 ORDER BY state ASC,
          city DESC;
```

au_fname	au_lname	city	state
Sarah	Buchman	Bronx	NY
Wendy	Heydemark	Boulder	CO
Hallie	Hull	San Francisco	CA
Klee	Hull	San Francisco	CA
Christian	Kells	New York	NY
	Kellsey	Palo Alto	CA
Paddy	O'Furniture	Sarasota	FL

图4-10 运行代码4-10的结果。结果按姓的升序排序

au_fname	au_lname	city	state
Wendy	Heydemark	Boulder	CO
Sarah	Buchman	Bronx	NY
Paddy	O'Furniture	Sarasota	FL
Klee	Hull	San Francisco	CA
Hallie	Hull	San Francisco	CA
Christian	Kells	New York	NY
	Kellsey	Palo Alto	CA

图4-11 运行代码4-11的结果。结果按名的降序排序。作者Kellsey的名是空串(''), 排在最后一个 (如果按升序排序, 则是第一个)

au_fname	au_lname	city	state
Hallie	Hull	San Francisco	CA
Klee	Hull	San Francisco	CA
	Kellsey	Palo Alto	CA
Wendy	Heydemark	Boulder	CO
Paddy	O'Furniture	Sarasota	FL
Christian	Kells	New York	NY
Sarah	Buchman	Bronx	NY

图4-12 运行代码4-12的结果

排序顺序

对数字、日期和时间值排序是明确的，对字符串排序却很复杂。DBMS使用排序序列或排序规则（collation）决定字符排序的顺序。排序规则定义了在字符集中每个字符的优先顺序。字符集取决于使用的语言，如欧洲语言（拉丁字符集）、希伯来语（希伯来文字符集）。排序规则也决定了是否区分大小写（'A'小于'a'吗？）、重音敏感度（'A'小于'A'吗？）、宽度敏感度（对于多字节还是Unicode字符）和其他的因素（如语言习惯等）。SQL标准没有定义详细的排序规则和字符集，因此每一个DBMS使用自己的排序策略和默认的排序规则。DBMS提供了显示当前排序规则和字符集的命令或者工具。例如，在Microsoft SQL Server中，可以运行命令exec sp_helpsort。查阅DBMS文档中的collation或sort order。

按子串排序

可以使用5.5节中介绍的函数，按字符串的特定部分对结果排序。例如，下面这个查询将按phone的末尾4个字符排序：

```
SELECT au_id, phone
  FROM authors
 ORDER BY
    substr(phone, length(phone)-3);
```

DBMS 这个查询可以运行在Oracle、DB2、MySQL和PostgreSQL中。在Microsoft SQL Server中，使用substring(phone, len(phone)-3, 4)。在Microsoft Access中，使用Mid(phone, len(phone)-3, 4)。

SQL允许在ORDER BY中指定列的相对位置编号而不是列名。位置编号引用的是结果中的列，而不是原始表中的列。使用列位置节省了输入，但如果在SELECT子句中对列重新排列，结果会不清晰，并容易带来错误。

» 按列相对位置排序

输入：

```
SELECT columns
  FROM table
 ORDER BY sort_num1 [ASC | DESC],
          sort_num2 [ASC | DESC],
          ...
          sort_numN [ASC | DESC];
```

columns是一个或多个以逗号分隔的列名，sort_num1, sort_num2, …, sort_numN是在1和columns列数之间的整数。每一个整数指定了某一列在columns中的相对位置。table是包含columns的表的名称（排序数字不能引用没有在columns中给出的列）。排序顺序和4.4.2节中是一样的（代码4-13和图4-13）。

au_fname	au_lname	city	state
Kellsey	Palo Alto	CA	
Hallie	Hull	San Francisco	CA
Klee	Hull	San Francisco	CA
Wendy	Heydemark	Boulder	CO
Paddy	O'Furniture	Sarasota	FL
Christian	Kells	New York	NY
Sarah	Buchman	Bronx	NY

图4-13 运行代码4-13的结果

代码4-13 列出作者的名、姓、所在城市和州。首先对州按升序排序（SELECT子句中的第4列），然后在每一个州中对姓按降序排序（第2列）。结果见图4-13

```
SELECT au_fname, au_lname, city, state
  FROM authors
 ORDER BY 4 ASC, 2 DESC;
```

排序和空值

排序是SQL违反空值不等于任何其他值（包括其他空值）概念的情形之一（逻辑比较NULL=NULL是未知的，而不是真）。当空值被排序时，它们被认为是彼此相等的。SQL标准将决定空值是大于还是小于所有非空值的权力留给DBMS。Microsoft Access、Microsoft SQL Server和MySQL将空值作为最小值处理（代码4-14和图4-14），Oracle、DB2和PostgreSQL将空值作为最大值处理，参见3.14节。

在Oracle中，使用带有NULL FIRST或NULL LAST的ORDER BY去控制空值排序行为。对于其他的DBMS，创建标记空值的派生列（见第5章），例如，CASE WHEN column IS NULL THEN 0 ELSE 1 END AS is_null，并在ORDER BY子句将它增加为第一列（用ASC或DESC）。

代码4-14 在排序列中的空值是第一个还是最后一个被列出，这依赖于DBMS。结果见图4-14

```
SELECT pub_id, state, country
  FROM publishers
 ORDER BY state ASC;
```

pub_id	state	country
P03	NULL	Germany
P04	CA	USA
P02	CA	USA
P01	NY	USA

图4-14 运行代码4-14的结果。结果对州按升序排序。运行这个查询的DBMS将空值作为最小值来处理，所以州为空值的行首先被列出来。若DBMS将空值作为最大值处理，将把同样的行列在最后

排序速度

最影响排序速度的3个因素，按照重要程度排列如下。

- 选择的行的数量。
- ORDER BY子句中列的数量。
- ORDER BY子句中列的长度。

通常将排序限制为所需的最小的行数。排序例程的运行时间和需要排序的行数不是线性比例的关系。因此，排序 $10n$ 行所需的时间要比排序 n 行所需时间的10倍长得多。如果可能，也可以设法减少排序的列的数量和表中定义的数据类型的长度。

✓ 提示

- 可以依据没有在SELECT子句中给出的列进行排序（代码4-15和图4-15），这种技术对于列的相对位置来说无效。
- 可以在ORDER BY中用列的别名替代列名（代码4-16和图4-16），参见4.2节。
- 可以在ORDER BY中多次使用同一列（但那是愚蠢的）。
- 如果ORDER BY列不能在结果中唯一标识每一行，有重复值的行将以任意顺序列出。尽管这是本书某些例子中的情形（见图4-10、图4-12和图4-13），但应该用足够多的ORDER BY列唯一确定各个行，特别是在要将结果显示给最终用户的情况下。

代码4-15 zip没有出现在要检索的列的列表中。结果见图4-15

```
SELECT city, state
  FROM authors
 ORDER BY zip ASC;
```

代码4-16 这个查询在ORDER BY子句中使用了列的别名。结果见图4-16

```
SELECT au_fname AS "First Name",
       au_lname AS "Last Name",
       state
  FROM authors
 ORDER BY state      ASC,
          "Last Name" ASC,
          "First Name" ASC;
```

city	state
New York	NY
Bronx	NY
Sarasota	FL
Boulder	CO
San Francisco	CA
San Francisco	CA
Palo Alto	CA

图4-15 运行代码4-15的结果。结果按zip编码升序排序。如果对一个不显示的列排序，行也许会以随机的顺序显示，将使最终用户感觉很混乱

First Name	Last Name	state
Hallie	Hull	CA
Klee	Hull	CA
	Kellsey	CA
Wendy	Heydemark	CO
Paddy	O'Furniture	FL
Sarah	Buchman	NY
Christian	Kells	NY

图4-16 运行代码4-16的结果

- 依照SQL标准，ORDER BY子句是CURSOR声明而不是SELECT语句的一部分。游标（cursor）是在数据库应用程序中定义的对象，超出了本书的范围。所有SQL实现都可以在SELECT语句中使用ORDER BY（因为DBMS隐式地建立了游标）。标准SQL在窗口函数（非本书内容）中也包含了ORDER BY。
 - 要基于逻辑条件排序，可对ORDER BY子句增加CASE表达式（见5.14节）。例如，对于下面这个查询，如果type是“history”，则按price排序；否则，将按sales排序。
- ```
SELECT title_id, type, price, sales
 FROM titles
 ORDER BY CASE WHEN type = 'history'
 THEN price ELSE sales END;
```
- 可以对表达式的结果排序，第5章介绍了如何使用函数和操作符创建表达式（代码4-17和图4-17）。
  - 可以在ORDER BY中混合使用列名、列的相对位置和表达式。

- 可以为经常排序的列创建索引（见第12章）。
- 实际显示的无序行的顺序是根据行在DBMS表中的物理顺序。因为物理顺序经常更改，例如在增加、更新、删除行或者创建索引时，所以不应该依赖物理顺序。

**代码4-17** 这个查询是按表达式排序，结果见图4-17。

因为在ORDER BY子句中重复表达式很麻烦，而且在结果中创建了更有意义的列的标记，所以为表达式创建了一个列的别名

```
SELECT title_id,
 price,
 sales,
 price * sales AS "Revenue"
 FROM titles
 ORDER BY "Revenue" DESC;
```

按列的相对位置排序在联合(UNION)查询中是有用的，参见9.1节。

**DBMS** DBMS对于能够出现在ORDER BY中的列有限制，这依赖于数据类型。例如，在Microsoft SQL Server中，不可以按ntext、text和image列排序；在Oracle中，不可以按blob、clob、nclob和bfile列排序。查阅DBMS文档中的SELECT和ORDER BY。

在Microsoft Access中，可以在ORDER BY中使用表达式列的别名。要运行代码4-17，或者在ORDER BY中重新输入表达式：

```
ORDER BY price * sales DESC
```

或者使用列的相对位置：

```
ORDER BY 4 DESC
```

100

## 4.5 使用 WHERE 筛选行

迄今为止，每一条SELECT语句的结果包含表中的每一行（对于特定的列）。可以使用WHERE子句从结果中筛选不想要的行。筛选功能给予了SELECT语句实权。在WHERE子句里指定查询条件，查询条件有一个或多个需要被表中的行满足的条件。条件或者断言是一个值为真(true)、假(false)或者未知(unknown)的逻辑表达式，条件为真的行出现在结果中；条件为假或未知的被排斥在外（空值产生的未知结果将在下一节中介绍）。SQL提供了表示不同条件类型的操作符（表4-1）。操作符定义为对值或者其他元素进行特定操作的符号或关键字。

表4-1 条件的类型

| 条件   | SQL操作符          | 条件   | SQL操作符  |
|------|-----------------|------|---------|
| 比较   | =、< >、<、<=、>、>= | 列表筛选 | IN      |
| 模式匹配 | LIKE            | 空值测试 | IS NULL |
| 范围筛选 | BETWEEN         |      |         |

| title_id | price | sales   | Revenue     |
|----------|-------|---------|-------------|
| T07      | 23.95 | 1500200 | 35929790.00 |
| T05      | 6.95  | 201440  | 1400008.00  |
| T12      | 12.99 | 100001  | 1299012.99  |
| T03      | 39.95 | 25667   | 1025396.65  |
| T11      | 7.99  | 94123   | 752042.77   |
| T13      | 29.99 | 10467   | 313905.33   |
| T06      | 19.95 | 11320   | 225834.00   |
| T02      | 19.95 | 9566    | 190841.70   |
| T04      | 12.99 | 13001   | 168882.99   |
| T09      | 13.95 | 5000    | 69750.00    |
| T08      | 10.00 | 4095    | 40950.00    |
| T01      | 21.99 | 566     | 12446.34    |
| T10      | NULL  | NULL    | NULL        |

图4-17 运行代码4-17的结果。结果按Revenue(price和sales的乘积)的降序列出标题

SQL的比较操作符比较两个值，求值的结果是真、假或者未知（表4-2）。数据类型决定了如何比较值。

- 字符串是按照字典顺序进行比较的。<意味着在前，>意味着在后，参见3.4节和4.4节。
- 数字是按照算术值进行比较。<意味着小于，>意味着大于。
- 日期和时间是按年代顺序进行比较。<意味着早于，>意味着晚于。日期和时间必须有相同的字段（年、月、日、小时等）以进行有意义的比较。

表4-2 比较操作符

| 操作符 | 描述  | 操作符 | 描述    |
|-----|-----|-----|-------|
| =   | 等于  | <=  | 小于或等于 |
| <>  | 不等于 | >   | 大于    |
| <   | 小于  | >=  | 大于或等于 |

只能对相同数据类型或类似数据类型进行比较。如果试图比较不同数据类型的值，DBMS将：

- 返回错误；
- 或者
- 对值不进行比较并返回没有行的结果；
- 或者
- 试图将值转换为相同的类型。如果成功，则比较它们；如果不成功，则返回错误。

### ⇒ 使用比较筛选行

输入：

```
SELECT columns
 FROM table
 WHERE test_column op value;
```

*columns*是一个或多个以逗号分隔的列名，*table*是包含*columns*的表的名称。在查找条件中，*test\_column*是*table*中的列的名称（*test\_column*不是必须在*columns*中列出）。*op*是一个在表4-2中列出的比较操作符。*value*是和*test\_column*中的值进行比较的值（代码4-18至代码4-20、图4-18至图4-20）。

| au_id | au_fname  | au_lname    |
|-------|-----------|-------------|
| A01   | Sarah     | Buchman     |
| A02   | Wendy     | Heydemark   |
| A05   | Christian | Kells       |
| A06   |           | Kellsey     |
| A07   | Paddy     | O'Furniture |

代码4-18 列出姓不是空值的作者。结果见图4-18

```
SELECT au_id, au_fname, au_lname
 FROM authors
 WHERE au_lname <> 'Hull';
```

代码4-19 列出没有签合同的图书的书名。结果见图  
4-19

```
SELECT title_name, contract
 FROM titles
 WHERE contract = 0;
```

图4-18 运行代码4-18的结果

| title_name                 | contract |
|----------------------------|----------|
| Not Without My Faberge Egg | 0        |

图4-19 运行代码4-19的结果

**代码4-20** 列出在2001年1月1日以后出版的图书的书名。结果见图4-20

```
SELECT title_name, pubdate
 FROM titles
 WHERE pubdate >= DATE '2001-01-01';
```

**✓ 提示**

- 在SELECT语句中，将WHERE子句放在ORDER BY子句之前（在两者都要出现的时候）。

□ 空值表示未知，不和任何内容值（甚至另一个空值）匹配。在比较中，包含空值的行将不在结果中。要比较空值，使用WHERE *test\_column* IS NULL（WHERE *test\_column* = NULL是不正确的），参见4.10节和3.14节。

- 比较的左边项和右边项可以比这里介绍的更复杂。普通的比较形式为：

*expr1 op expr2*

*expr1*和*expr2*是表达式。表达式是结果为单一值（每行）的列名、字面量、函数和操作符的任意组合。第5章更详细地讲解了表达式（代码4-21和图4-21）。

| title_name                   | pubdate    |
|------------------------------|------------|
| Exchange of Platitudes       | 2001-01-01 |
| Just Wait Until After School | 2001-06-01 |
| Kiss My Boo-Boo              | 2002-05-31 |

图4-20 运行代码4-20的结果

102

**代码4-21** 列出Revenue大于1 000 000美元的图书的书名。这个查询条件使用了算术表达式。结果见图4-21

```
SELECT title_name,
 price * sales AS "Revenue"
 FROM titles
 WHERE price * sales > 1000000;
```

- 为了提高速度，将表达式中的常数个数变为最小。例如，更改

```
WHERE col1 + 2 <= 10
```

为

```
WHERE col1 <= 8
```

最好的习惯是只将简单的列引用放在=的左边，而将更复杂的表达式放在=的右边。

- 一般而言，最快的比较是相等(=)，然后是不相等(<、<=、>、>=)，最慢的是不等于(≠)。如果可能，应该使用尽可能快的比较来表示条件。

- 不能在WHERE子句中使用SUM()或者COUNT()这样的聚合函数，见第6章。

- 从表中选择特定行的操作，称为限制。

- **DBMS** DBMS的排序规则决定了字符串比较是区分大小写的('A'≠'a')还是不区分大小写的('A'='a')。Microsoft Access、Microsoft SQL Server、DB2和MySQL默认执行不区分大小写的比较，Oracle和PostgreSQL默认执行区分大小写的比较。一般而言，区分大小写的比较要比不区分大小写的比较稍微快一些，参见5.6节。是否区分大小写在上下文中可以不同。

例如，MySQL的比较在WHERE比较中是不区分大小写的，但是在与串有关的函数中是区分大小写的。

| title_name                    | Revenue     |
|-------------------------------|-------------|
| Ask Your System Administrator | 1025396.65  |
| Exchange of Platitudes        | 1400008.00  |
| I Blame My Mother             | 35929790.00 |
| Spontaneous, Not Annoying     | 1299012.99  |

图4-21 运行代码4-21的结果

103

在Microsoft Access的日期字面量中，省略DATE关键字并用符号#而不是引号环绕文字。要运行代码4-20，更改WHERE子句中的日期为#2001-01-01#。

在Microsoft SQL Server和DB2的日期字面量中，省略DATE关键字。要运行代码4-20，更改WHERE子句中的日期为'2001-01-01'。

在PostgreSQL较早的版本中，要对实数（浮点数）和列NUMERIC或DECIMAL的值进行比较，显式地将实数转换为NUMERIC或DECIMAL，参见5.13节。

某些DBMS支持和 $\neq$ （不等于）意义相同的比较操作符!=。应该尽量使用 $\neq$ 以保证代码的可移植性。

#### 列的别名和WHERE

4

如果在SELECT子句中给列定义了别名（见4.2节），不能在WHERE子句中引用它。下面的查询是失败的，因为WHERE子句是在SELECT子句之前计算的，所以当计算WHERE子句时，别名copies\_sold还不存在。

```
-- Wrong
SELECT sales AS copies_sold
 FROM titles
 WHERE copies_sold > 100000;
```

改为使用在WHERE子句之前计算的FROM子句的子查询（见第8章）：

```
-- Correct
SELECT *
 FROM (SELECT sales AS copies_sold
 FROM titles) ta
 WHERE copies_sold > 100000;
```

这个解决方案不但适用于列的别名，而且也适用于在WHERE子句中引用的聚合函数、标量子查询和窗口函数。注意，在后面的查询中，子查询被命名为别名ta（表的别名）。所有的DBMS都接受表的别名，但不是全部都需要它们，参见8.7节。

104

## 4.6 使用 AND、OR 和 NOT 组合及求反条件

可以在单个的WHERE子句中定义多个条件，也就是说，可以基于在多个列中的值检索行。可以使用AND和OR操作符将两个或多个条件组合为一个复合条件。AND、OR和NOT是逻辑操作符。逻辑操作符，即布尔操作符，被设计为处理真值：真、假和未知。

如果已经用过其他语言编写程序（或学习过命题逻辑），将对二值逻辑（two-value logic, 2VL）体系很熟悉。在二值逻辑中，逻辑表达式的结果或者为true或者为false。2VL假定了完全的知识，其中所有的命题非真即假。但是，数据库模型的真实数据和我们关于世界的知识是不完全的。这就是为什么需要用空值去表示未知的值（见3.14节）。

2VL不足以表示知识的差距，所以SQL使用三值逻辑（three-value logic, 3VL）。在三值逻辑中，逻辑表达式的结果是true、false和unknown。如果复合条件的结果是false或unknown，行将被排斥在结果之外（要检索带有空值的行，参见4.10节）。

105

### 4.6.1 AND 操作符

AND操作符的主要特点如下。

- AND连接两个条件，并且只有当两个条件都为真时才返回真。
- 表4-3所示为用AND组合两个条件时的可能值。表中最左侧的列显示了第一个条件的真值，最上面的行显示了第二个条件的真值，每一个交叉点显示了AND的结果。这种类型的表称为真值表。

表4-3 AND真值表

| AND | 真  | 假 | 未知 |
|-----|----|---|----|
| 真   | 真  | 假 | 未知 |
| 假   | 假  | 假 | 假  |
| 未知  | 未知 | 假 | 未知 |

- 可以用AND连接任意个数的条件，包含在结果中的行必须所有条件为真。
- AND是可以交换位置的（不依赖于顺序）。WHERE condition1 AND condition2和WHERE condition2 AND condition1是等价的。
- 可以在圆括号中放入一个或全部条件。某些复合条件需要圆括号以强制计算的顺序。
- AND的例子见代码4-22和代码4-23、图4-22和图4-23。

106

**代码4-22** 列出价格低于20美元的传记类图书。结果见图4-22

```
SELECT title_name, type, price
 FROM titles
 WHERE type = 'biography' AND price < 20;
```

| title_name                | type      | price |
|---------------------------|-----------|-------|
| How About Never?          | biography | 19.95 |
| Spontaneous, Not Annoying | biography | 12.99 |

图4-22 运行代码4-22的结果

**代码4-23** 列出不居住在加利福尼亚，并且姓以字母H到Z开头的作者。结果见图4-23

```
SELECT au_fname, au_lname
 FROM authors
 WHERE au_lname >= 'H'
 AND au_lname <= 'Z'
 AND state <> 'CA';
```

| au_fname  | au_lname    |
|-----------|-------------|
| Wendy     | Heydemark   |
| Christian | Kells       |
| Paddy     | O'Furniture |

图4-23 运行代码4-23的结果。字符串比较的结果依赖于DBMS的排序顺序，参见4.4节

## 4.6.2 OR操作符

OR操作符的主要特点如下。

- OR连接了两个条件，如果任何一个条件为真或两个条件都为真，返回真。
- 表4-4显示了OR真值表。
- OR可以连接任意个数的条件，OR将检索出匹配任何一个条件或全部条件的行。
- 类似于AND，OR是可以交换位置的，列出条件的顺序无关紧要。
- 可以在圆括号中放入一个或全部条件。

表4-4 OR真值表

| OR | 真 | 假  | 未知 |
|----|---|----|----|
| 真  | 真 | 真  | 真  |
| 假  | 真 | 假  | 未知 |
| 未知 | 真 | 未知 | 未知 |

OR的例子见代码4-24和代码4-25、图4-24和图4-25。

代码4-25显示了条件中空值的影响。你可能认为如图4-25所示的结果会显示表publishers中所有的行，但是出版社P03（位于德国）所在的行却缺失了，因为对应的列state中包含了空值。空值引起所有OR条件的结果为未知，因此这些行被排斥在结果之外。要测试空值，参见4.10节。

107

4

**代码4-24** 列出居住在纽约州、科罗拉多州或者旧金山市的作者。结果见图4-24

```
SELECT au_fname, au_lname, city, state
 FROM authors
 WHERE (state = 'NY')
 OR (state = 'CO')
 OR (city = 'San Francisco');
```

**代码4-25** 列出位于加利福尼亚或者不位于加利福尼亚的出版社。这个例子是为了显示空值在条件中的影响而人为设计的。结果见图4-25

```
SELECT pub_id, pub_name, state, country
 FROM publishers
 WHERE (state = 'CA')
 OR (state <> 'CA');
```

### 4.6.3 NOT 操作符

NOT操作符的主要特点如下。

□ 和AND、OR不同，NOT不能连接两个条件，而是否定（取反）一个条件。

□ 表4-5显示了NOT真值表。

□ 在比较中，将NOT放在列名或表达式之前：

```
WHERE NOT state = 'CA' --Correct
```

不能放在操作符之前（即便在读的时候听起来更好一些）：

```
WHERE state NOT = 'CA' --Illegal
```

表4-5 NOT真值表

| 条 件 | NOT条件 |
|-----|-------|
| 真   | 假     |
| 假   | 真     |
| 未知  | 未知    |

□ NOT只对一个条件起作用。要否定两个或多个输入条件，对每一个条件重复使用NOT。例如，要列出类型不是传记，并且价格不低于20美元的图书的书名，输入

```
SELECT title_id, type, price
 FROM titles
 WHERE NOT type = 'biography'
 AND NOT price < 20; --Correct
```

| au_fname  | au_lname   | city          | state |
|-----------|------------|---------------|-------|
| Sarah     | Buchman    | Bronx         | NY    |
| Wendy     | Heydermark | Boulder       | CO    |
| Hallie    | Hull       | San Francisco | CA    |
| Klee      | Hull       | San Francisco | CA    |
| Christian | Kells      | New York      | NY    |

图4-24 运行代码4-24的结果

| pub_id | pub_name          | state | country |
|--------|-------------------|-------|---------|
| P01    | Abatis Publishers | NY    | USA     |
| P02    | Core Dump Books   | CA    | USA     |
| P04    | Tenterhooks Press | CA    | USA     |

图4-25 运行代码4-25的结果。出版社P03是缺失的，因为它的state是空值

而不是

```
SELECT title_id, type, price
 FROM titles
 WHERE NOT type = 'biography'
 AND price < 20; --Wrong
```

后面的子句是合法的，但返回的却是错误的结果。本节中的提示中会给出表示等价于NOT条件的方法。

- 在比较中，使用NOT经常和代码书写风格有关。下面的两个子句是等价的：

```
WHERE NOT state = 'CA'
WHERE state <> 'CA'
```

- 可以将条件放入圆括号内。

108

NOT的例子见代码4-26和代码4-27、图4-26和图4-27。

**代码4-26** 列出不居住在加利福尼亚州的作者。结果见图4-26

```
SELECT au_fname, au_lname, state
 FROM authors
 WHERE NOT (state = 'CA');
```

**代码4-27** 列出价格不低于20美元，并且已售出超过15 000册的图书的书名。结果见图4-27

```
SELECT title_name, sales, price
 FROM titles
 WHERE NOT (price < 20)
 AND (sales > 15000);
```

| au_fname  | au_lname    | state |
|-----------|-------------|-------|
| Sarah     | Buchman     | NY    |
| Wendy     | Heydermark  | CO    |
| Christian | Kells       | NY    |
| Paddy     | O'Furniture | FL    |

图4-26 运行代码4-26的结果

| title_name                    | sales   | price |
|-------------------------------|---------|-------|
| Ask Your System Administrator | 25667   | 39.95 |
| I Blame My Mother             | 1500200 | 23.95 |

图4-27 运行代码4-27的结果

#### 4.6.4 AND、OR 和 NOT 一起使用

可以在一个表达式中组合3个逻辑操作符。DBMS使用SQL的优先规则确定哪一个操作符先运算。优先规则将在5.3节中介绍，现在只需要知道，当复合条件表达式中使用多个逻辑操作符时，NOT是最先计算的，然后是AND，最后才是OR。可以使用圆括号改变这个顺序：圆括号内的先运算。当加圆括号嵌套使用时，最里面的条件先运算。在默认的优先规则下， $x \text{ AND NOT } y \text{ OR } z$ 和 $(x \text{ AND } (\text{NOT } y)) \text{ OR } z$ 是等价的。使用圆括号是明智的，胜过依赖默认的操作顺序，可使操作顺序更加清楚。

例如，如果想列出价格低于20美元的历史和传记类图书书名。代码4-28不能得出想要的结果，因为AND是在OR之前计算的，所以查询按照下面的顺序进行。

- (1) 找出所有低于20美元的传记类书名。
- (2) 找出所有历史类书名（不考虑价格）。
- (3) 在结果中列出两类书名的集合（图4-28）。

| title_id | type      | price |
|----------|-----------|-------|
| T01      | history   | 21.99 |
| T02      | history   | 19.95 |
| T06      | biography | 19.95 |
| T12      | biography | 12.99 |
| T13      | history   | 29.99 |

图4-28 运行代码4-28的结果。这个结果包含了两个并不想要的价格高于20美元的历史类书名

**代码4-28** 如果想列出价格低于20美元的历史类和传记类图书，这个查询将不能得出想要的结果，因为AND比OR有更高的优先级。结果见图4-28

```
SELECT title_id, type, price
 FROM titles
 WHERE type = 'history'
 OR type = 'biography'
 AND price < 20;
```

要修复这个查询，增加圆括号以强制OR运算先运行。代码4-29按照下面的顺序进行计算。

- (1) 找出传记类和历史类书名。
- (2) 在第(1)步找出的书名中，保留价格低于20美元的。
- (3) 在结果中，列出书名的子集（图4-29）。

**代码4-29** 要修复代码4-28，增加圆括号以强制OR在  
AND之前计算。结果见图4-29

```
SELECT title_id, type, price
 FROM titles
 WHERE (type = 'history'
 OR type = 'biography')
 AND price < 20;
```

| title_id | type      | price |
|----------|-----------|-------|
| T02      | history   | 19.95 |
| T06      | biography | 19.95 |
| T12      | biography | 12.99 |

图4-29 运行代码4-29的结果（已修改代码）

### 剖析WHERE子句

如果WHERE子句不能运行，可以通过分别显示每一个条件的结果来调试它。例如，要看在代码4-29中每个比较的结果，在SELECT子句输出列的列表中放入每一个比较表达式，连同要比较的值。

```
SELECT type,
 type = 'history' AS "Hist?",
 type = 'biography' AS "Bio?",
 price,
 price < 20 AS "<20?"
 FROM titles;
```

这个查询可以在Microsoft Access、MySQL和PostgreSQL上运行。如果DBMS将=号解释为赋值操作符，而不是比较操作符，必须将逻辑比较替换为等价的表达式。例如，在Oracle中，可以用INSTR(type,'history')替换type = 'history'，查询结果如下。

| type       | Hist? | Bio? | Price | <20? |
|------------|-------|------|-------|------|
| history    | 1     | 0    | 21.99 | 0    |
| history    | 1     | 0    | 19.95 | 1    |
| computer   | 0     | 0    | 39.95 | 0    |
| psychology | 0     | 0    | 12.99 | 1    |
| psychology | 0     | 0    | 6.95  | 1    |
| biography  | 0     | 1    | 19.95 | 1    |
| biography  | 0     | 1    | 23.95 | 0    |
| children   | 0     | 0    | 10.00 | 1    |
| children   | 0     | 0    | 13.95 | 1    |
| biography  | 0     | 1    | NULL  | NULL |
| psychology | 0     | 0    | 7.99  | 1    |
| biography  | 0     | 1    | 12.99 | 1    |
| history    | 1     | 0    | 29.99 | 0    |

如果比较为假，比较列显示0；如果为真，显示非0；如果为未知，显示空值。

109

4

110

### ✓ 提示

- 本节中的例子显示了和比较条件一起使用的AND、OR和NOT操作符，这些操作符可以和任意类型的条件一起使用。
- 如果查询条件只包含AND操作符，首先放置为真可能性最小的条件，查询将会变快。如果col1='A'的可能性比col2='B'小，那么：

```
WHERE col1='A' AND col2='B'
```

比

```
WHERE col2='B' AND col1='A'
```

快，因为如果第一个为假，则DBMS就不用去计算第二个表达式了。对于只包含OR操作符的查询条件，则按相反的方式操作：首先放置为真可能性最大的条件。如果条件有相同的可能性，首先放置复杂度最小的表达式。

这样设置是因为DBMS优化器是从左向右读取WHERE子句的（大多数情况下）。然而，Oracle的基于开销的优化器（与它的基于规则的优化器相反）从右到左读取。

### □ 输入

```
WHERE state = 'NY' OR 'CA' --Illegal
```

而不是

```
WHERE state = 'NY' OR state = 'CA'
```

是一种常见的错误类型。

111

- 很容易将正确的口头语句“翻译”成错误的SQL语句。如果说：“列出价格低于10美元的和高于30美元的书”，使人想起使用AND操作符。

```
SELECT title_name, price
 FROM titles
 WHERE price<10 AND price>30; --Wrong
```

但是这个查询没有行返回，因为使用了AND逻辑命令，对于一本书来说，价格低于10美元同时高于30美元是不可能的。OR的逻辑含义是找出满足任何一个标准，而不是同时满足所有标准的书。

```
WHERE price<10 OR price>30 --Correct
```

- 表4-6所示为表示同样条件的可选方法。第一个等价物是双重否定，接下来的两个是德摩·根定律，最后两个是分配律。

表4-6 等价条件

| 条 件            | 等 价 于                  |
|----------------|------------------------|
| NOT (NOT p)    | p                      |
| NOT (p AND q)  | (NOT p) OR (NOT q)     |
| NOT (p OR q)   | (NOT p) AND (NOT q)    |
| p AND (q OR r) | (p AND q) OR (p AND r) |
| p OR (q AND r) | (p OR q) AND (p OR r)  |

- 某些DBMS支持异或(XOR)逻辑操作符，仅当只有一个操作数为真时结果为真。 $p \text{ XOR } q$ 和 $(p \text{ AND } (\text{NOT } q)) \text{ OR } ((\text{NOT } p) \text{ AND } q)$ 是等价的。

□ DBMS 在MySQL 4.0.4及先前版本中，“假AND未知”的运算结果为未知，而不是假。

112

### 重新表示条件

要成为一名优秀的SQL（或任何一种语言）程序员，必须掌握表4-6中的定律。例如，想重新表示条件使查询运行得更快时，它们就特别有用。语句

```
SELECT * FROM mytable
WHERE col1 = 1
 AND NOT (col1 = col2 OR col3 = 3);
```

等价于

```
SELECT * FROM mytable
WHERE col1 = 1
 AND col2 <> 1
 AND col3 <> 3;
```

但是如果所用的DBMS优化器不是足够智能到可重新在内部表示前者的话，后者将运行得更快（条件 $col1 = col2$ 比将 $col1$ 和 $col2$ 与字面量相比较需要的计算开销大）。

也可使用这些定律将条件更改为其相反面。例如，条件

```
WHERE (col1='A') AND (col2='B')
```

的相反面是

```
WHERE (col1<>'A') OR (col2<>'B')
```

在这种情况下，用NOT去否定整个原始表达式更容易一些：

```
WHERE NOT ((col1='A') AND (col2='B'))
```

但是这种简单方法在包含多个AND、OR和NOT的复杂条件下运行并无效果。

这里有一个问题需要解决：只考虑下面的第一行代码，看看能否重复应用等价规则放进NOT操作符，直到它们只应用单独的表达式p、q和r：

```
NOT ((p AND q) OR (NOT p AND r))
= NOT (p AND q) AND NOT (NOT p AND r)
= (NOT p OR NOT q) AND (p OR NOT r)
```

4

113

## 4.7 使用 LIKE 匹配模式

在先前的例子中检索行是基于一列或多列的精确值，也可以使用LIKE依据部分信息检索行。如果不知道精确值（作者的姓是kel\*）或者想用类似值检索行（哪个作者居住在旧金山海湾地区），那么可以使用LIKE。

LIKE条件的主要特点如下。

- LIKE只对字符串起作用，对数字或日期不起作用。
- LIKE使用匹配对照值的模式。模式是被引起来的包含要匹配的字面量的字符串及通配符的结合。通配符是用于匹配部分值的特殊字符。表4-7列出通配符操作符，表4-8列出一些模式的例子。
- 字符串比较可以是区分大小写也可以是不区分大小写的，这依赖于DBMS，参见4.5节中的DBMS提示。

114

- 可以用NOT LIKE否定LIKE条件。
- 可以将LIKE条件与其他AND和OR条件组合。

表4-7 通配符操作符

| 操作符 | 匹 配            |
|-----|----------------|
| %   | 百分号%匹配零个或多个字符串 |
| _   | 下划线_匹配任意一个字符   |

表4-8 %和\_模式的例子

| 模 式     | 匹 配                                                                          |
|---------|------------------------------------------------------------------------------|
| 'A%'    | 匹配以A开头的长度大于或等于1的串，包括单个字母A。匹配'A'、'Anonymous'和'AC/DC'                          |
| '%s'    | 匹配以s结尾的长度大于或等于1的串，包括单个字母s。尾部带有空格的串（在s之后）不能匹配。匹配's'、'Victoria Falls'和'DBMSs'  |
| '%in%'  | 匹配在任意位置包含in的长度大于或等于2的串。匹配'in'、'inch'、'Pine'、'linchpin'和'lynchpin'            |
| '____'  | 匹配任何一个4个字符的串。匹配'ABCD'、'I am'和'Jack'                                          |
| 'Qua--' | 匹配以Qua开头的任何一个5个字符的串。匹配'Quack'、'Quaff'和'Quake'                                |
| '_re_'  | 匹配以re作为第2个和第3个字符的4个字符的串。匹配'Tree'、'area'和'fret'                               |
| '_re%'  | 匹配以任意字符开头，以re作为第2个和第3个字符的长度大于或等于3的串。匹配'Tree'、'area'、'fret'、'are'和'fretful'   |
| '%re_'  | 匹配以re作为倒数第2个和第3个字符后跟任意字符的长度大于或等于3的串。匹配'Tree'、'area'、'fret'、'red'和'Blood red' |

### ⇒ 通过匹配模式筛选行

输入：

```
SELECT columns
 FROM table
 WHERE test_column [NOT] LIKE
 'pattern';
```

columns是一个或多个以逗号分隔的列名，table是包含columns的表的名称。

在查询条件下，test\_column是table中的列名（test\_column不是必须在columns中列出），pattern是和test\_column中的值相比较的模式。pattern是和表4-8中列出的例子类似的串。指明NOT LIKE检索不匹配pattern的值的行（代码4-30至代码4-33、图4-30至图4-33）。

115

代码4-30 列出姓以Kel开头的作者。结果见图4-30

```
SELECT au_fname, au_lname
 FROM authors
 WHERE au_lname LIKE 'Kel%';
```

| au_fname  | au_lname |
|-----------|----------|
| Christian | Kells    |
|           | Kellsey  |

图4-30 运行代码4-30的结果

**代码4-31** 列出姓中以ll作为第3个和第4个字符的作者。结果见图4-31

```
SELECT au_fname, au_lname
 FROM authors
 WHERE au_lname LIKE '_ll%';
```

**代码4-32** 列出居住在旧金山海湾地区（该地区的邮政编码以94开头）的作者。结果见图4-32

```
SELECT au_fname, au_lname, city, state, zip
 FROM authors
 WHERE zip LIKE '94__';
```

**代码4-33** 列出居住地区号不是212、415和303的作者。这个例子显示了包含电话号码的3种可选模式。应该支持第一种可选方案，因为单一字符匹配（\_）比多个字符匹配（%）快。结果见图4-33

```
SELECT au_fname, au_lname, phone
 FROM authors
 WHERE phone NOT LIKE '212-__-__'
 AND phone NOT LIKE '415-__-%'
 AND phone NOT LIKE '303-%';
```

可以查找包含特殊通配符的值。使用ESCAPE关键字定义转义符，这样就可以按照字面量字符的方式查找百分号或下划线。紧挨在通配符之前的转义符去除了通配符的特殊含义。例如，如果转义符是!，在模式中!%查找文字%（未转义的通配符仍具有它们的特殊含义）。转义符不是试图检索的值的一部分；如果查找'50% OFF!'，选择其他不是!的转义符。表4-9所示为转义和不转义模式的一些例子，指定的转义符是!。

| au_fname  | au_lname |
|-----------|----------|
| Hallie    | Hull     |
| Klee      | Hull     |
| Christian | Kells    |
|           | Kellsey  |

图4-31 运行代码4-31的结果

| au_fname | au_lname | city          | state | zip   |
|----------|----------|---------------|-------|-------|
| Hallie   | Hull     | San Francisco | CA    | 94123 |
| Klee     | Hull     | San Francisco | CA    | 94123 |
|          | Kellsey  | Palo Alto     | CA    | 94305 |

图4-32 运行代码4-32的结果

| au_fname | au_lname    | phone        |
|----------|-------------|--------------|
| Sarah    | Buchman     | 718-496-7223 |
|          | Kellsey     | 650-836-7128 |
| Paddy    | O'Furniture | 941-925-0752 |

图4-33 运行代码4-33的结果

表4-9 转义和不转义模式

| 模 式     | 匹 配                        |
|---------|----------------------------|
| '100%'  | 未转义。匹配100后跟0个或多个字符的串       |
| '100!%' | 转义。匹配'100%'                |
| '_op'   | 未转义。匹配'top'、'hop'、'pop'，等等 |
| '!_op'  | 转义。匹配'_op'                 |

⇒ 匹配转义符

输入：

```
SELECT columns
 FROM table
 WHERE test_column [NOT] LIKE
 'pattern'
 ESCAPE 'escape_char';
```

除了ESCAPE子句以外，这个语法和4.7.1节中的SELECT语句是一样的。*escape\_char*是单个字符。在*pattern*中跟在*escape\_char*后的任意一个字符被解释为字面量，*escape\_char*本身不是查找模式的一部分（代码4-34和图4-34）。

**代码4-34** 列出包含百分号的书名。只有跟在转义符!后的%才有它的字面量含义，另外两个百分号仍然担当通配符。结果见图4-34

```
SELECT title_name
 FROM titles
 WHERE title_name LIKE '%!%%' ESCAPE '!';
```

#### ✓ 提示

- *test\_column*可以是一个表达式。
- 在LIKE之前的NOT和在*test\_column*之前的NOT是无关的（见4.6.3节）。以下两个子句是等价的。

```
WHERE phone NOT LIKE '212-%'
WHERE NOT phone LIKE '212-%'
```

甚至可以写出无谓的双重否定：

```
WHERE NOT phone NOT LIKE '212-%'
```

以检索有212地区号的每一个人。

- 通配符查找是很耗费时间的——特别是如果使用以%开头的模式。如果别的查找类型能够实现，就不要使用通配符。
- 在模式不包含通配符的最简单的情况下，LIKE像=比较一样（NOT LIKE像<>比较一样）。在许多情况下：

```
WHERE city LIKE 'New York'
```

等价于

```
WHERE city = 'New York'
```

但是，如果DBMS对于LIKE考虑尾部的空格而对于=不考虑，这些是比较不同的。如果这一点不重要，=形式通常要比LIKE快。

- **DBMS** Microsoft Access不支持ESCAPE子句。改为用方括号环绕通配符，表示它是字面量字符。要运行代码4-34，使用

```
WHERE title_name LIKE '%[%]%'
```

来替换WHERE子句。某些DBMS使用正则表达式去匹配模式。例如，Microsoft SQL Server支持POSIX类型正则表达式的有限变量。[ ]通配符匹配在范围或集合中的任意一个字符，[^]通配符匹配不在范围或集合中的任意一个字符，表4-10列出了一些例子。SQL标准对于正则表达式的匹配使用SIMILAR操作符。DBMS对正则表达式的支持有差异，查找DBMS文档中的“LIKE”、“正则表达式”或“模式匹配”。某些DBMS使用LIKE去查找数字及日期列。

表4-10 [ ]和[^]模式的例子

| 模 式       | 匹 配                     |
|-----------|-------------------------|
| '[a-c]at' | 匹配'bat'和'cat'，但不匹配'fat' |

|            |
|------------|
| title_name |
| -----      |

图4-34 运行代码4-34的结果。结果为空，没有图书书名含有%字符

(续)

| 模 式       | 匹 配                           |
|-----------|-------------------------------|
| '[bcf]at' | 匹配'bat'、'cat'和'fat'，但不匹配'eat' |
| '[^c]at'  | 匹配'bat'和'fat'，但不匹配'cat'       |
| 'se[^n]%' | 匹配以se开头并且第3个字符不是n的、长度大于或等于2的串 |

## 4.8 使用 BETWEEN 进行范围筛选

用BETWEEN确定给定值是否落在特定的范围之内。

BETWEEN条件的主要特点如下。

- BETWEEN适用于字符串、数字和日期。
- BETWEEN范围包含用AND分隔的低端值和高端值。低端值必须小于或等于高端值。
- BETWEEN子句使用起来方便、简短，它可以用AND改写。

```
WHERE test_column BETWEEN
 low_value AND high_value
```

等价于

```
WHERE (test_column >= low_value)
 AND (test_column <= high_value)
```

- BETWEEN指定的包含范围（inclusive range）包含查找的高端值和低端值。要指定不包含端点的排他范围，使用>和<比较而不是BETWEEN。

```
WHERE (test_column > low_value)
 AND (test_column < high_value)
```

- 字符串比较是否区分大小写，依赖于DBMS，参见4.5节中的DBMS提示。
- 可以用NOT BETWEEN否定BETWEEN条件。
- 可以用AND和OR将BETWEEN条件和其他条件组合。

### ⇒ 使用范围筛选行

输入：

```
SELECT columns
 FROM table
 WHERE test_column [NOT] BETWEEN
 low_value AND high_value;
```

columns是一个或多个以逗号分隔的列名，table是包含columns的表的名称。

在查找条件中，test\_column是table中的列名（test\_column不是必须在columns中列出），low\_value和high\_value指定了范围的端点（和test\_column中的值进行比较）。low\_value必须小于或等于high\_value，两个值都必须和test\_column的数据类型一样或者是可以比较的。指定NOT BETWEEN匹配位于范围之外的值（代码4-35至代码4-37、图4-35至图4-37）。

| au_fname  | au_lname | zip   |
|-----------|----------|-------|
| Sarah     | Buchman  | 10468 |
| Hallie    | Hull     | 94123 |
| Klee      | Hull     | 94123 |
| Christian | Kells    | 10014 |
|           | Kellsey  | 94305 |

4

118

119

图4-35 运行代码4-35的结果

**代码4-35** 列出居住地邮政编码在20 000~89 999之外的作者。结果见图4-35

```
SELECT au_fname, au_lname, zip
 FROM authors
 WHERE zip NOT BETWEEN '20000' AND '89999';
```

**代码4-36** 列出价格在10~19.95美元（包含端点）的图书。结果见图4-36

```
SELECT title_id, price
 FROM titles
 WHERE price BETWEEN 10 AND 19.95;
```

| title_id | price |
|----------|-------|
| T02      | 19.95 |
| T04      | 12.99 |
| T06      | 19.95 |
| T08      | 10.00 |
| T09      | 13.95 |
| T12      | 12.99 |

图4-36 运行代码4-36的结果

**代码4-37** 列出在2000年出版的图书。结果见图4-37

```
SELECT title_id, pubdate
 FROM titles
 WHERE pubdate BETWEEN DATE '2000-01-01'
 AND DATE '2000-12-31';
```

| title_id | pubdate    |
|----------|------------|
| T01      | 2000-08-01 |
| T03      | 2000-09-01 |
| T06      | 2000-07-31 |
| T11      | 2000-11-30 |
| T12      | 2000-08-31 |

图4-37 运行代码4-37的结果

### ✓ 提示

- *test\_column*可以是表达式。
- 在BETWEEN之前的NOT和在*test\_column*之前的NOT是不相关的，参见4.7节中的提示。
- 代码4-38显示了如何用排他范围（不包含端点10美元和19.95美元）改写代码4-36，结果见图4-38。
- 指定字符的范围需要仔细考虑。假设想查找以字母F开头的姓。因为将会返回姓是字母G（是字母G，是以字母G开头）的人，所以下面的子句将不能得出正确结果。

```
WHERE last_name BETWEEN 'F' AND 'G'
```

下面的子句显示了指定结束点的正确方法（在大多数情况下）。

```
WHERE last_name BETWEEN 'F' AND 'Fz'
```

**代码4-38** 列出价格在10美元和19.95美元之间（不包含端点）的图书。结果见图4-38

```
SELECT title_id, price
 FROM titles
 WHERE (price > 10)
 AND (price < 19.95);
```

- **DBMS** 在PostgreSQL较早的版本中，要将代码4-36和代码4-38中的浮点数转换为DECIMAL（十进制数），参见5.13节。要运行代码4-36和代码4-38，应把浮点文字变为CAST(19.95 AS DECIMAL)

| title_id | price |
|----------|-------|
| T04      | 12.99 |
| T09      | 13.95 |
| T12      | 12.99 |

图4-38 运行代码4-38的结果

在Microsoft Access的日期字面量中，省略DATE关键字，并且用#号替代引号环绕字面量。要运行代码4-37，应在WHERE子句中更改日期为#2000-01-01#和#2000-12-31#。

在Microsoft SQL Server和DB2的日期字面量中，省略DATE关键字。要运行代码4-37，应在WHERE子句中更改日期为'2000-01-01'和'2000-12-31'。

在某些DBMS中，低端值可以超过高端值，查阅DBMS文档中的WHERE或者BETWEEN。

120

4

## 4.9 使用 IN 进行列表筛选

用IN确定给定值是否匹配指定列表中的值。

IN条件的主要特点如下。

- IN可以处理字符串、数字、日期等一起。
- IN列表是由一个或多个逗号分隔、加上括号的值的列表。这个列表的项不需要特定的顺序。
- IN是方便、简短的子句，它是可以用OR改写。

```
WHERE test_column IN
 (value1, value2, value3)
```

等价于

```
WHERE (test_column = value1)
 OR (test_column = value2)
 OR (test_column = value3)
```

- 字符串比较是否区分大小写依赖于DBMS，参见4.5节中的DBMS提示。
- 可以用NOT IN否定IN条件。
- 可以使用AND和OR将IN条件和其他条件组合。

121

### ⇒ 使用列表筛选行

输入：

```
SELECT columns
 FROM table
 WHERE test_column [NOT] IN
 (value1, value2, ...);
```

columns是一个或多个以逗号分隔的列名，table是包含columns的表的名称。

在查询条件中，test\_column是在table中的列名（test\_column不是必须在columns中列出）。value1, value2,...是和test\_column中的值比较的一个或多个逗号分隔的值。列表值可以按任意的顺序显示，并且必须和test\_column中的数据类型是一样的或者是可以比较的。可以定义NOT IN匹配不在列表中的值（代码4-39至代码4-41、图4-39至图4-41）。

**代码4-39** 列出不居住在纽约州、新泽西州或加利福尼亚州的作者。结果见图4-39

```
SELECT au_fname, au_lname, state
 FROM authors
 WHERE state NOT IN ('NY', 'NJ', 'CA');
```

| au_fname | au_lname    | state |
|----------|-------------|-------|
| Wendy    | Heydemark   | CO    |
| Paddy    | O'Furniture | FL    |

图4-39 运行代码4-39的结果

**代码4-40** 列出预付款为0美元、1 000美元或5 000美元的图书。结果见图4-40

```
SELECT title_id, advance
 FROM royalties
 WHERE advance IN
 (0.00, 1000.00, 5000.00);
```

| title_id | advance |
|----------|---------|
| T02      | 1000.00 |
| T08      | 0.00    |
| T09      | 0.00    |

图4-40 运行代码4-40的结果

**代码4-41** 列出在2000年、2001年或2002年元旦出版的图书书名。结果见图4-41

```
SELECT title_id, pubdate
 FROM titles
 WHERE pubdate IN
 (DATE '2000-01-01',
 DATE '2001-01-01',
 DATE '2002-01-01');
```

| title_id | pubdate    |
|----------|------------|
| T05      | 2001-01-01 |

图4-41 运行代码4-41的结果

### ✓ 提示

- *test\_column*可以是表达式。
- IN前的NOT和*test\_column*前的NOT是不相关的，参见4.7节中的提示。
- 如果列表包含大量的值，使用一个IN条件替代多个OR条件会使代码变得更加易读（而且IN通常要比多个OR运行得快）。
- 为了加快速度，首先列出最可能的值。例如，如果是在测试美国的地址，首先列出最可能的州：WHERE state IN ('CA', 'TX', 'NY', 'FL', ..., 'VT', 'DC', 'WY')。
- 查找条件
 

```
WHERE col1 BETWEEN 1 AND 5
 AND col1 <> 3
```

 通常比下面的快：
 

```
WHERE col1 IN (1, 2, 4, 5)
```
- 如果使用IN替代多个OR的组合，复合条件的计算顺序易读并易于管理，参见4.6节。
- 可以使用IN确定给定值是否匹配子查询中的值，参见第8章。
- NOT IN等价于用AND对不等式进行组合的测试。下面的语句等价于代码4-39：

```
SELECT au_fname, au_lname, state
 FROM authors
 WHERE state <> 'NY'
 AND state <> 'NJ'
 AND state <> 'CA';
```

- 在Microsoft Access的日期字面量中，省略DATE关键字，并且用#号替代引号环绕文字。  
**DBMS** 要运行代码4-41，更改WHERE子句为

```
WHERE pubdate IN
 (#1/1/2000|,
 #1/1/2001|,
 #1/1/2002|)
```

在Microsoft SQL Server和DB2的日期字面量中，省略DATE关键字。要运行代码4-41，更改WHERE子句为

```
WHERE pubdate IN
 ('2000-01-01',
 '2001-01-01',
 '2002-01-01')
```

在PostgreSQL的较早的版本中，将代码4-40中的浮点数转换为DECIMAL（十进制数），参见5.13节。要运行代码4-40，更改WHERE子句为

```
WHERE advance IN
 (CAST(0.00 AS DECIMAL),
 CAST(1000.00 AS DECIMAL),
 CAST(5000.00 AS DECIMAL))
```

123

4

## 4.10 使用 IS NULL 测试空值

回顾3.14节，空值表示缺失或者未知的值。这种情况引发一个问题：因为未知的值不能满足特定条件，所以LIKE、BETWEEN、IN和其他的WHERE子句条件不能发现空值。空值不与其他值匹配——即便是其他空值。不能使用=或<>去测试一个值是否为空值。

例如，在表publishers中，注意出版社P03在列state中有一个空值，这是因为“州”在德国不适用（代码4-42和图4-42）。因为空值既不是加利福尼亚也不是非加利福尼亚，不能使用互补的比较去找出空值，它是未定义的（代码4-43和代码4-44、图4-43和图4-44）。

**代码4-42** 列出所有出版社的位置。结果见图4-42

```
SELECT pub_id, city, state, country
 FROM publishers;
```

**代码4-43** 列出位于加利福尼亚的出版社。结果见图4-43

```
SELECT pub_id, city, state, country
 FROM publishers
 WHERE state = 'CA';
```

**代码4-44** 列出不在加利福尼亚的出版社（这里是错误的方式，正确的方式见代码4-45）。结果见图4-44

```
SELECT pub_id, city, state, country
 FROM publishers
 WHERE state <> 'CA';
```

| pub_id | city     | state | country |
|--------|----------|-------|---------|
| P01    | New York | NY    | USA     |

| pub_id | city          | state | country |
|--------|---------------|-------|---------|
| P01    | New York      | NY    | USA     |
| P02    | San Francisco | CA    | USA     |
| P03    | Hamburg       | NULL  | Germany |
| P04    | Berkeley      | CA    | USA     |

图4-42 运行代码4-42的结果。列state不适用于德国的出版社

| pub_id | city          | state | country |
|--------|---------------|-------|---------|
| P02    | San Francisco | CA    | USA     |
| P04    | Berkeley      | CA    | USA     |

图4-43 运行代码4-43的结果。结果不包含出版社P03

**图4-44** 运行代码4-44的结果。这个结果也不包含出版社P03。毕竟条件state = 'CA' 和state <> 'CA' 不是互补的；空值不匹配任何值，所以不能使用迄今为止介绍过的各类条件来选择

为了避免严重错误，SQL提供了IS NULL来确定给定值是否为空值。IS NULL条件的主要特点如下。

- IS NULL可以应用于任意数据类型的列。
- 可以用IS NOT NULL否定IS NULL。
- 可以用AND和OR将IS NULL条件和其他条件组合。

124

### ⇒ 使用空值或非空值检索行

输入：

```
SELECT columns
 FROM table
 WHERE test_column IS [NOT] NULL;
```

*columns*是一个或多个以逗号分隔的列名，*table*是包含*columns*的表的名称。

在查询条件中，*test\_column*是在*table*中的列名（*test\_column*不是必须在*columns*中列出）。指定  
125 NOT NULL匹配非空值（代码4-45和代码4-46、图4-45和图4-46）。

#### 代码4-45 列出不在加利福尼亚的出版社（正确的方式）。结果见图4-45

```
SELECT pub_id, city, state, country
 FROM publishers
 WHERE state <> 'CA'
 OR state IS NULL;
```

| pub_id | city     | state | country |
|--------|----------|-------|---------|
| P01    | New York | NY    | USA     |
| P03    | Hamburg  | NULL  | Germany |

图4-45 运行代码4-45的结果。现在结果中有出版社P03

#### 代码4-46 列出出版日期已知（过去或将来）的传记。结果见图4-46

```
SELECT title_id, type, pubdate
 FROM titles
 WHERE type = 'biography'
 AND pubdate IS NOT NULL;
```

| title_id | type      | pubdate    |
|----------|-----------|------------|
| T06      | biography | 2000-07-31 |
| T07      | biography | 1999-10-01 |
| T12      | biography | 2000-08-31 |

图4-46 运行代码4-46的结果。如果没有NOT NULL条件，结果将包含图书T10

#### ✓ 提示

- *test\_column*可以是表达式。
- NULL前的NOT和*test\_column*之前的NOT是不相关的，参见4.7节中的提示。

- 如果包含空值的列是WHERE条件中的测试列，空值行被排除在结果之外。因为在列state中的空值不能和任何值比较，例如，下面的查询将可以检索表publishers中的所有行（见图4-42）。

```
SELECT pub_id, city, state, country
 FROM publishers
 WHERE country <> 'Canada';
```

要在列中禁止空值，参见11.4节。

- 再次重申，空值和空串（''）是不一样的。例如，在表authors中，列au\_fname对于作者A06（姓为Kellsey）包含一个空串。查找名的WHERE条件是

```
WHERE au_fname = ''
```

而不是

```
WHERE au_fname IS NULL
```

- DBMS Oracle将空串（''）作为空值处理，参见3.14节中的DBMS提示。

4

126

**操**作符和函数计算源自列的值、由系统确定的值、常数和其他数据的结果。可以执行如下运算。

- 算术运算——将每个人的工资减少10%。
- 字符串运算——将个人资料连接到通信地址里。
- 日期和时间运算——计算两个日期的时间间隔。
- 系统运算——找出DBMS的系统时间。

操作符是标识作用于一个或多个元素的操作的符号或关键字。称为操作数的元素是SQL表达式。回顾3.1节中的提示，表达式是计算结果为一个值（或空值）的符号和记号的任何合法组合。例如，在`price*2`中，`*`是操作符，`price`和`2`是操作数。

函数是用于执行专门任务的内置、已命名的程序。大多数函数接受用括号括起来的参数，参数是传递给完成任务的函数的值。参数可以是列名、字面量、嵌套的函数或更加复杂的表达式。例如，在`UPPER(au_lname)`中，`UPPER`是函数名，`au_lname`是参数。

127

## 5.1 创建派生列

可以使用操作符和函数创建派生列。派生列是一个计算结果，是用不同于对列简单引用的SELECT子句表达式创建的。派生列不会成为表中的永久列，它们用于显示或者报表目的。

派生列中的值经常是由现有列的值计算出来的，但也可以使用常数表达式（如串、数字或日期）或者系统值（如系统时间）创建派生列。代码5-1表示了一个产生普通算术运算的SELECT语句。因为它不从表中检索数据，所以不需要FROM子句，结果见图5-1。

**代码5-1** SELECT子句中的常数表达式。因为不需要从表中检索数据，所以不需要FROM子句。结果见图5-1

```
SELECT 2 + 3;
```

回顾2.1节，闭包确保每一个结果是一个表，所以这个最简单的结果是一个包含值5的 $1 \times 1$ 的表。如果检索列和常数，常数会出现在结果中的每一行（代码5-2和图5-2）。

2 + 3

-----

5

图5-1 运行代码5-1的结果。结果是一个一行一列的表

**代码5-2** 这里检索列和常数表达式。结果见图5-2

```
SELECT au_id, 2 + 3
FROM authors;
```

DBMS将给派生列分配一个默认的名字，通常是将表达式作为带引号的标识符。因为系统分配的名字太长，难以使用，以及不便于数据库应用程序的引用，应该使用AS子句显式地命名派生列，参见4.2节（代码5-3和图5-3）。

**代码5-3** 列出打10%折扣以后书的价格。如果删除AS子句，派生列将拥有DBMS定义的默认名。

结果见图5-3

```
SELECT title_id,
 price,
 0.10 AS "Discount",
 price * (1 - 0.10) AS "New price"
 FROM titles;
```

#### ✓ 提示

□ **DBMS** Oracle在SELECT语句中需要用到FROM子句，以便自动创建虚表DUAL。SELECT常数表达式，查阅Oracle文档中的DUAL table。要运行代码5-1，添加从DUAL中选择常数值的FROM子句。

```
SELECT 2 + 3
 FROM DUAL;
```

DB2在SELECT语句中需要用到FROM子句，以便自动创建虚表SYSIBM.SYSDUMMY1用于SELECT常数表达式，查阅DB2文档中的SYSIBM.SYSDUMMY1 table。要运行代码5-1，添加从SYSIBM.SYSDUMMY1中选择常数值的FROM子句。

```
SELECT 2 + 3
 FROM SYSIBM.SYSDUMMY1;
```

在PostgreSQL的较早版本中，将代码5-3中浮点数转换为DECIMAL（十进制数），见5.13节。要运行代码5-3，更改SELECT子句中的New price计算为

```
price * CAST((1 - 0.10) AS DECIMAL)
```

## 5.2 执行算术运算

单目（或一元）算术操作符对一个数字操作数执行数学运算并产生一个结果。 $-$ （取反）运算更改其操作数的符号，无用的 $+$ （等同）操作符不更改操作数。双目（或二元）算术操作符执行对两个数字操作数的数学运算并产生一个结果。这些操作符包括常用的 $+$ （加）、 $-$ （减）、 $*$ （乘）和 $/$ （除）。

au\_id 2 + 3

|     |   |
|-----|---|
| A01 | 5 |
| A02 | 5 |
| A03 | 5 |
| A04 | 5 |
| A05 | 5 |
| A06 | 5 |
| A07 | 5 |

图5-2 运行代码5-2的结果。每一行中的常数是重复的

title\_id price Discount New price

| title_id | price | Discount | New price |
|----------|-------|----------|-----------|
| T01      | 21.99 | 0.10     | 19.79     |
| T02      | 19.95 | 0.10     | 17.95     |
| T03      | 39.95 | 0.10     | 35.96     |
| T04      | 12.99 | 0.10     | 11.69     |
| T05      | 6.95  | 0.10     | 6.25      |
| T06      | 19.95 | 0.10     | 17.95     |
| T07      | 23.95 | 0.10     | 21.56     |
| T08      | 10.00 | 0.10     | 9.00      |
| T09      | 13.95 | 0.10     | 12.56     |
| T10      | NULL  | 0.10     | NULL      |
| T11      | 7.99  | 0.10     | 7.19      |
| T12      | 12.99 | 0.10     | 11.69     |
| T13      | 29.99 | 0.10     | 26.99     |

图5-3 运行代码5-3的结果

表5-1所示为SQL的算术操作符（*expr*是数字表达式）。

表5-1 算术操作符

| 操作符             | 作用                           | 操作符             | 作用                           |
|-----------------|------------------------------|-----------------|------------------------------|
| $-expr$         | 取反 <i>expr</i> 的符号           | $expr1 - expr2$ | <i>expr1</i> 减去 <i>expr2</i> |
| $+expr$         | 不更改 <i>expr</i>              | $expr1 * expr2$ | <i>expr1</i> 乘以 <i>expr2</i> |
| $expr1 + expr2$ | <i>expr1</i> 加上 <i>expr2</i> | $expr1 / expr2$ | <i>expr1</i> 除以 <i>expr2</i> |

## ⇒ 更改数字的符号

输入：

$-expr$

130

*expr*是数字表达式（代码5-4和图5-4）。

代码5-4 取反操作符更改数字的符号。结果见图5-4

```
SELECT title_id,
 -advance AS "Advance"
 FROM royalties;
```

## ⇒ 加、减、乘或除

输入  $expr1+expr2$ 表示加， $expr1-expr2$ 表示减， $expr1*expr2$ 表示乘， $expr1/expr2$ 表示除。*expr1*和*expr2*是数字表达式（代码5-5和图5-5）。

代码5-5 按收入（价格乘以销售量）递减排列传记。结果见图5-5

```
SELECT title_id,
 price * sales AS "Revenue"
 FROM titles
 WHERE type = 'biography'
 ORDER BY price * sales DESC;
```

## 其他操作符和函数

除SQL标准中定义的操作符和函数之外（或本书之外）所有DBMS都提供大量的操作符和函数。实际上，最新的标准也是在追随各种DBMS中已经存在多年的函数。先前的SQL标准功能比桌面计算器都差，查阅DBMS文档中的operators和functions，能够找到数学、统计学、金融、科学、三角学、换算、字符串、日期、位运算、系统、元数据、安全和其他词条。

## ✓ 提示

□ 包含空值的任何算术运算的结果是空值。

title\_id Advance

|     |             |
|-----|-------------|
| T01 | -10000.00   |
| T02 | -1000.00    |
| T03 | -15000.00   |
| T04 | -20000.00   |
| T05 | -100000.00  |
| T06 | -20000.00   |
| T07 | -1000000.00 |
| T08 | 0.00        |
| T09 | 0.00        |
| T10 | NULL        |
| T11 | -100000.00  |
| T12 | -50000.00   |
| T13 | -20000.00   |

图5-4 运行代码5-4的结果。注意0没有符号（既不是正数也不是负数）

title\_id Revenue

|     |             |
|-----|-------------|
| T07 | 35929790.00 |
| T12 | 1299012.99  |
| T06 | 225834.00   |
| T10 | NULL        |

图5-5 运行代码5-5的结果

- 如果在单个表达式中使用多个操作符，需要使用圆括号去控制计算的顺序，参见5.3节。
- 如果在算术表达式中使用多种数字数据类型，DBMS将把所有的数字（强制）转换为表达式中最复杂操作数的数据类型，并以这种类型返回结果。这个转换过程称为提升。例如，一个INTEGER（整数）和一个FLOAT（浮点数）相加，DBMS将整数转换为浮点数，然后进行数字加法，以浮点数形式返回结果。在某些情况下，必须显式地将一种数据类型转换为另一种数据类型，参见5.13节。
- 如果编写数据库应用程序或者更新行，注意数据类型对于某些算术运算不是封闭的。例如，如果两个SMALLINT相乘或相加，结果也许会超出SMALLINT列可以支持的范围。同样地，两个INTEGER相除不一定会产生一个INTEGER。
- **DBMS** 有时DBMS强制数学封闭，因此当用整数除以一个整数时必须要仔细。如果一个整数被一个整数除数除尽，这个结果可能是删除了结果中小数部分的整数结果。你可能认为代码5-6中的两个派生列应该包含同样的值，因为列pages（INTEGER）被两个相等的常数（10（整数）和10.0（浮点数））来除。Microsoft Access、Oracle和MySQL将返回期望的结果（图5-6a），但是Microsoft SQL Server、DB2和PostgreSQL将结果截取为一个整数（图5-6b）。

131

5

132

**代码5-6** 这个查询第一个派生列是用pages除以整型常数10，第二个派生列是用pages除以浮点型常数10.0。期望两个派生列中的值是一样的。结果见图5-6a和图5-6b

```
SELECT title_id,
 pages,
 pages/10 AS "pages/10",
 pages/10.0 AS "pages/10.0"
 FROM titles;
```

|     | title_id | pages | pages/10 | pages/10.0 |
|-----|----------|-------|----------|------------|
| T01 | 107      | 10.7  | 10.7     | 10.7       |
| T02 | 14       | 1.4   | 1.4      | 1.4        |
| T03 | 1226     | 122.6 | 122.6    | 122.6      |
| T04 | 510      | 51.0  | 51.0     | 51.0       |
| T05 | 201      | 20.1  | 20.1     | 20.1       |
| T06 | 473      | 47.3  | 47.3     | 47.3       |
| T07 | 333      | 33.3  | 33.3     | 33.3       |
| T08 | 86       | 8.6   | 8.6      | 8.6        |
| T09 | 22       | 2.2   | 2.2      | 2.2        |
| T10 | NULL     | NULL  | NULL     | NULL       |
| T11 | 826      | 82.6  | 82.6     | 82.6       |
| T12 | 507      | 50.7  | 50.7     | 50.7       |
| T13 | 802      | 80.2  | 80.2     | 80.2       |

图5-6a 运行代码5-6的结果。对于Microsoft Access、Oracle和MySQL，两个整数相除产生一个浮点数（和期望的一样）

|     | title_id | pages | pages/10 | pages/10.0 |
|-----|----------|-------|----------|------------|
| T01 | 107      | 10    | 10       | 10.7       |
| T02 | 14       | 1     | 1        | 1.4        |
| T03 | 1226     | 122   | 122      | 122.6      |
| T04 | 510      | 51    | 51       | 51.0       |
| T05 | 201      | 20    | 20       | 20.1       |
| T06 | 473      | 47    | 47       | 47.3       |
| T07 | 333      | 33    | 33       | 33.3       |
| T08 | 86       | 8     | 8        | 8.6        |
| T09 | 22       | 2     | 2        | 2.2        |
| T10 | NULL     | NULL  | NULL     | NULL       |
| T11 | 826      | 82    | 82       | 82.6       |
| T12 | 507      | 50    | 50       | 50.7       |
| T13 | 802      | 80    | 80       | 80.2       |

图5-6b 运行代码5-6的结果。对于Microsoft SQL Server、DB2和PostgreSQL，两个整数相除产生一个整数，结果中的小数部分被舍弃（和期望的不一样）

### 5.3 确定计算的顺序

优先级决定在一个表达式中使用的多个不同操作符的优先权。优先级高的操作符先计算。算术操作符（+、-、\*等）的优先级高于比较操作符（<、=、>等），比较操作符的优先级高于逻辑操作符（NOT、AND、OR）。因此，表达式

```
a or b * c >= d
```

等价于

```
a or ((b * c) >= d)
```

低优先级操作符比高优先级的绑定少。表5-2中由高至低列出了操作符的优先级。同一行中的操作符优先级相同。

结合性决定一个表达式中相邻的操作符具有相同优先级时的计算顺序。SQL使用从左到右的结合性。

表5-2 计算顺序（由高至低）

| 操作符            | 描述          | 操作符 | 描述  |
|----------------|-------------|-----|-----|
| +、-            | 单目的等同，单目的取反 | NOT | 逻辑非 |
| *、/            | 乘、除         | AND | 逻辑与 |
| +、-            | 加、减         | OR  | 逻辑或 |
| =、<>、<、<=、>、>= | 比较操作符       |     |     |

没必要去记这些信息。可以使用圆括号重写优先级和结合性规则（代码5-7和图5-7）。

**代码5-7** 第1列和第2列显示了如何使用圆括号重写优先级规则。第3列和第4列显示了如何使用圆括号重写结合性规则。结果见图5-7

```
SELECT 2 + 3 * 4 AS "2+3*4",
 (2 + 3) * 4 AS "(2+3)*4",
 6 / 2 * 3 AS "6/2*3",
 6 / (2 * 3) AS "6/(2*3);
```

2+3\*4 (2+3)\*4 6/2\*3 6/(2\*3)

-----

14      20      9      1

图5-7 运行代码5-7的结果

#### ✓ 提示

- 对于复杂的表达式添加圆括号（即便它们不是必要的）以确保期望的计算顺序，使代码更具可移植性和易读性，这是一种良好的编程风格。
- **DBMS** 表5-2是不完整的，它省略了某些标准的（如IN和EXISTS）和非标准的（DBMS特定的）操作符。要确定具体的DBMS使用的计算完成顺序，查阅DBMS文档中的precedence。要在Oracle中运行代码5-7，需添加子句FROM DUAL。要在DB2中运行，需添加子句FROM SYSIBM.SYSDUMMY1，参见5.1节中的DBMS提示。

### 5.4 使用||连接串

使用操作符||组合（即连接）串。这个操作符的主要特点如下。

- 操作符||是两个连续的竖线（即管道）字符。
- 连接不会在串间加入空格。

- 双目操作符||将两个串连接为一个串，如'formal'||'dehyde'等于'formaldehyde'。
- 可以将多个串连接合并为一个串，如'a'||'b'||'c'||'d'等于'abcd'。
- 与空串('')连接将保持串不变，如'a'||''||'b'等于'ab'。
- 包含空值的连接运算的结果为空值，如'a'||NULL||'b'等于NULL，(但Oracle例外，参见本节DBMS提示)。
- 要连接串和非串(如数字或日期和时间值)，如果DBMS不能隐式地转换类型，必须将非串转换为串，参见5.13节。

134

### » 连接串

输入：

```
string1 || string2
```

*string1*和*string2*是要连接的串。每一个操作数是一个串表达式，比如包含字符串的列、串字面量或者返回串的运算或函数的结果(代码5-8至代码5-11、图5-8至图5-11)。

5

#### 代码5-8 列出作者的姓和名(连接为一个列并且按姓/名排序)。结果见图5-8

```
SELECT au_fname || ' ' || au_lname
 AS "Author name"
 FROM authors
 ORDER BY au_lname ASC, au_fname ASC;
```

#### 代码5-9 按销售数的递减顺序列出传记的销售数。 这里，需要将sales从整数转换为串。结果 见图5-9

```
SELECT CAST(sales AS CHAR(7))
 || ' copies sold of title '
 || title_id
 AS "Biography sales"
 FROM titles
 WHERE type = 'biography'
 AND sales IS NOT NULL
 ORDER BY sales DESC;
```

#### 代码5-10 按出版日期的递减顺序列出传记。这里， 需要将pubdate从日期转换为串。结果见图 5-10

```
SELECT 'Title '
 || title_id
 || ' published on '
 || CAST(pubdate AS CHAR(10))
 AS "Biography publication dates"
 FROM titles
 WHERE type = 'biography'
 AND pubdate IS NOT NULL
 ORDER BY pubdate DESC;
```

| Author name       |
|-------------------|
| Sarah Buchman     |
| Wendy Heydemark   |
| Hallie Hull       |
| Klee Hull         |
| Christian Kells   |
| Kellsey           |
| Paddy O'Furniture |

图5-8 运行代码5-8的结果

| Biography sales                  |
|----------------------------------|
| 1500200 copies sold of title T07 |
| 100001 copies sold of title T12  |
| 11320 copies sold of title T06   |

图5-9 运行代码5-9的结果

| Biography publication dates       |
|-----------------------------------|
| Title T12 published on 2000-08-31 |
| Title T06 published on 2000-07-31 |
| Title T07 published on 1999-10-01 |

图5-10 运行代码5-10的结果

**代码5-11** 列出所有姓名为Klee Hull的作者。结果见图5-11

```
SELECT au_id, au_fname, au_lname
 FROM authors
 WHERE au_fname || ' ' || au_lname
 = 'Klee Hull';
```

| au_id | au_fname | au_lname |
|-------|----------|----------|
| A04   | Klee     | Hull     |

图5-11 运行代码5-11的结果

### ✓ 提示

- 可以在SELECT、WHERE和ORDER BY子句中或者任何允许使用表达式的位置使用||。
- 可以连接十六进制或二进制串，B'0100'||B'1011'等于B'01001011'。
- 代码5-11显示如何在WHERE子句中使用||，但事实上它是糟糕的SQL。表达子句的高效方式是：  

```
WHERE au_fname = 'Klee'
 AND au_lname = 'Hull'
```
- 可以使用TRIM()函数从连接的串中删除不需要的空格。回顾3.5节，CHAR的值用尾随的空格填补，有时会在连接的串中产生较长的、难看的空格延伸。例如，在图5-8中，TRIM()能删除名字Kellsey前的额外空格，参见5.7节。
- **DBMS** 在Microsoft Access中，连接操作符是+，转换函数是Format(string)。要运行代码5-8至代码5-11，需要更改连接表达式为（代码5-8）  
`au_fname + ' ' + au_lname`

135

对于代码5-9：

```
Format(sales)
→ + ' copies sold of title '
→ + title_id
```

对于代码5-10：

```
'Title '
→ + title_id
→ + ' published on '
→ + Format(pubdate)
```

对于代码5-11：

```
au_fname + ' ' + au_lname
→ = 'Klee Hull';
```

在Microsoft SQL Server中，连接操作符是+。要运行代码5-8到代码5-11，需要更改连接表达式为（代码5-8）  
`au_fname + ' ' + au_lname`

对于代码5-9：

```
CAST(sales AS CHAR(7))
→ + ' copies sold of title '
→ + title_id
```

对于代码5-10：

```
'Title '
→ + title_id
```

```

→ + ' published on '
→ + CAST(pubdate AS CHAR(10))

```

对于代码5-11:

```

au_fname + ' ' + au_lname
→ = 'Klee Hull';

```

在MySQL中，连接函数是CONCAT()。||操作符本身是合法的，但在MySQL中它的默认含义是逻辑OR（使用PIPES\_AS\_CONCAT模式可将||作为字符连接操作符而不是OR的同义词来处理）。CONCAT()接受任何参数并在必要时将非串转换为串（因此CAST()不是必需的）。要运行代码5-8至代码5-11，需要更改连接表达式为（代码5-8）：

```
CONCAT(au_fname, ' ', au_lname)
```

对于代码5-9:

```

CONCAT(sales,
→ ' copies sold of title ',
→ title_id)

```

对于代码5-10:

```

CONCAT('Title ',
→ title_id,
→ ' published on ',
→ pubdate)

```

对于代码5-11:

```

CONCAT(au_fname, ' ', au_lname)
→ = 'Klee Hull';

```

Oracle将空值当作空串来处理：'a'||NULL||'b'返回'ab'，参见3.14节中的DBMS提示。

Oracle、MySQL和PostgreSQL隐式地将非串转换为串。如果省略CAST()，代码5-9和代码5-10可以运行在这些DBMS上，请查阅DBMS文档中的concatenation或者conversion。

Oracle和DB2也支持CONCAT()函数。

5

136

## 5.5 使用 SUBSTRING()提取子串

使用函数SUBSTRING()提取串的一部分。这个函数的主要特点如下。

- 子串是源串的连续字符序列，包含空串或整个源串本身。
- SUBSTRING()提取开始于特定位置，并且延续特定个数字符的部分串。
- 空串的子串是空串。
- 如果参数为空，SUBSTRING()将返回空值（但Oracle例外，参见本节DBMS提示）。

### ⇒ 提取子串

输入：

```
SUBSTRING(string FROM start [FOR length])
```

137

*string*是从中提取子串的源串。*string*是串表达式（如包含字符串的列、串字面量，或者是返回串的运算或函数的结果）。*start*是定义子串开始位置的整数，*length*是定义子串长度的整数（返回字符的个数）。*start*从1开始计算。如果省略FOR *length*，SUBSTRING()返回从串的开头到结尾的所有字符（代码5-12至代码5-14、图5-12至图5-14）。

**代码5-12** 将出版社ID分割为字母部分和数字部分。出版社ID的字母部分是第一个字符，其余字符是数字部分。结果见图5-12

```
SELECT pub_id,
 SUBSTRING(pub_id FROM 1 FOR 1)
 AS "Alpha part",
 SUBSTRING(pub_id FROM 2)
 AS "Num part"
 FROM publishers;
```

| pub_id | Alpha | part | Num | part |
|--------|-------|------|-----|------|
| P01    | P     |      | 01  |      |
| P02    | P     |      | 02  |      |
| P03    | P     |      | 03  |      |
| P04    | P     |      | 04  |      |

图5-12 运行代码5-12的结果

**代码5-13** 列出来自纽约州和加利福亚尼州的作者名的首字母和姓。结果见图5-13

```
SELECT SUBSTRING(au_fname FROM 1 FOR 1)
 || '.'
 || au_lname
 AS "Author name",
 state
 FROM authors
 WHERE state IN ('NY', 'CO');
```

| Author name  | state |
|--------------|-------|
| S. Buchman   | NY    |
| W. Heydemark | CO    |
| C. Kells     | NY    |

图5-13 运行代码5-13的结果

**代码5-14** 列出其电话地区号是415的作者。结果见图5-14

```
SELECT au_fname, au_lname, phone
 FROM authors
 WHERE SUBSTRING(phone FROM 1 FOR 3)='415';
```

| au_fname | au_lname | phone        |
|----------|----------|--------------|
| Hallie   | Hull     | 415-549-4278 |
| Klee     | Hull     | 415-549-4278 |

图5-14 运行代码5-14的结果

### ✓ 提示

- 可以在SELECT、WHERE和ORDER BY子句中，或者任何允许使用表达式的位置使用SUBSTRING()。
- 可以在十六进制或二进制串中提取子串：SUBSTRING(B'01001011'FROM 5 FOR 4)返回B'1011'。
- **DBMS** 在Microsoft Access中，子串函数是Mid(*string*, *start* [, *length*]), 使用+去连接串。  
要运行代码5-12至代码5-14，更改子串表达式为（代码5-12）

```
Mid(pub_id, 1, 1)
Mid(pub_id, 2)
```

对于代码5-13：

```
Mid(au_fname, 1, 1) + '.' + au_lname
```

对于代码5-14：

```
Mid(phone, 1, 3)='415'
```

在Microsoft SQL Server中，子串函数是SUBSTRING(*string*, *start*, *length*)。使用+去连接串。

要运行代码5-12至代码5-14，更改子串表达式为（代码5-12）

```
SUBSTRING(pub_id, 1, 1)
SUBSTRING(pub_id, 2, LEN(pub_id)-1)
```

对于代码5-13:

```
SUBSTRING(au_fname, 1, 1)
→ +
→ + au_lname
```

对于代码5-14:

```
SUBSTRING(phone, 1, 3)='415'
```

在Oracle和DB2中，子串函数是SUBSTR(string, start [,length])。要运行代码5-12至代码 138  
5-14，更改子串表达式为（代码5-12）

```
SUBSTR(pub_id, 1, 1)
SUBSTR(pub_id, 2)
```

5

对于代码5-13:

```
SUBSTR(au_fname, 1, 1)
→ ||
→ || au_lname
```

对于代码5-14:

```
SUBSTR(phone, 1, 3)='415'
```

在MySQL中，使用CONCAT()运行代码5-13（见5.4节），更改连接表达式为：

```
CONCAT(
→ SUBSTRING(au_fname FROM 1 FOR 1),
→ '. ',
→ au_lname)
```

Oracle将空值作为空串处理：SUBSTR(NULL, 1, 2)将返回''，参见3.14节中的DBMS提示。

有的DBMS也许隐式地限制Start和Length参数（对于合理值来说太小或太大）。例如，子串函数自动用1替代负数的start，用串的长度替代过长的length，查阅DBMS文档中的substring。

MySQL和PostgreSQL也支持SUBSTR(string, start, length)形式的子串函数。 139

## 5.6 使用 UPPER()和 LOWER()更改串的大小写

使用函数UPPER()返回将小写字母转换为大写字母的串，使用函数LOWER()返回将大写字母转换为小写字母的串。这两个函数的主要特点如下。

- 大小写字符可以是小写格式（a）或大写格式（A）的字母。
- 大小写的更改只对字母起作用。数字、标点和空格不会更改。
- 大小写的更改对空串（''）不起作用。
- 如果参数为空，UPPER()和LOWER()将返回空值（但Oracle除外，参见本节DBMS提示）。

### 不区分大小写的比较

在DBMS中，默认执行区分大小写的WHERE子句比较，UPPER()或LOWER()经常被用于不区分大小

140

写的比较。

```
WHERE UPPER(au_fname) = 'JOHN'
```

如果确信数据是正确的，只查找合理的字母组合要比使用区分大小写的函数快。

```
WHERE au_fname = 'JOHN'
OR au_fname = 'John'
```

UPPER()和LOWER()对带有发音符的字符（如重音符和变音符）起作用。例如，UPPER('ö')是'Ö'。如果数据包含这样的字符，可以采用如WHERE UPPER(au\_fname) = 'JOSE'的方式进行不区分大小写的比较，保证DBMS不会丢失转换标志。UPPER('José')应该是'JOSE'，而不是'JOSE'，参见4.5节。

### » 转换串为大写或小写

要转换串为大写，输入：

```
UPPER(string)
```

要转换串为小写，输入：

```
LOWER(string)
```

*string*是串表达式，如包含字符串的列、串字面量，或者返回串结果的运算或函数。（代码5-15和代码5-16、图5-15和图5-16）。

**代码5-15** 以小写字母列出作者的名，以大写字母列出作者的姓。结果见图5-15

```
SELECT LOWER(au_fname) AS "Lower",
 UPPER(au_lname) AS "Upper"
 FROM authors;
```

**代码5-16** 列出包含不区分大小写的字符MO的图书的书名。要让这个查询得出正确结果，在LIKE模式里所有的字母必须是大写字母。结果见图5-16

```
SELECT title_name
 FROM titles
 WHERE UPPER(title_name) LIKE '%M%O%';
```

#### ✓ 提示

- 可以在SELECT、WHERE和ORDER BY子句中，或任何允许使用表达式的位置，使用UPPER()和LOWER()。
- UPPER()和LOWER()对没有大小写概念的字符集不起作用（如希伯来文和中文）。
- 在Microsoft Access中，大写函数和小写函数是UCase(string)和LCase(string)。要运行代码5-15和代码5-16，更改大小写表达为（代码5-15）

```
LCase(au_fname)
UCase(au_lname)
```

| Lower     | Upper       |
|-----------|-------------|
| sarah     | BUCHMAN     |
| wendy     | HEYDEMARK   |
| hallie    | HULL        |
| klee      | HULL        |
| christian | KELLS       |
|           | KELLSEY     |
| paddy     | O'FURNITURE |

图5-15 运行代码5-15的结果

| title_name                |
|---------------------------|
| 200 Years of German Humor |
| I Blame My Mother         |

图5-16 运行代码5-16的结果

对于代码5-16：

```
UCASE(title_name) LIKE '%MO%'
```

Oracle将空值作为空串处理：UPPER(NULL)和LOWER(NULL)返回''，参见3.14节中的DBMS提示。DBMS可能提供了其他字符串大小写函数（也就是转换大小写字母，或者将串转换为句首字母大写或标题格式），请查阅DBMS文档中的character functions和string functions。

141

## 5.7 使用 TRIM()修整字符

使用函数TRIM()删除串两端不需要的字符。这个函数的主要特点如下。

- 可以修整前导字符、尾随字符或者两者都修整（不能使用TRIM()修整串中间的字符）。
- 默认情况下，TRIM()修整空格，但是也可以删除任何不需要的字符，比如前导和尾随的零或者星号。
- TRIM()通常用于格式化结果和WHERE子句中的比较。
- TRIM()可用于修整CHAR值的尾随空格。回顾3.5节，DBMS自动添加空格到CHAR值尾端以创建满足定义长度的串。
- 修整对空串（''）不起作用。
- 如果参数为空，TRIM()将返回空值（但Oracle例外，参见本节DBMS提示）。

5

142

### ⇒ 修整串中的空格

输入：

```
TRIM([LEADING | TRAILING | BOTH]
 FROM string)
```

*string*是串表达式，如包含字符串的列、串字面量，或者返回串结果的运算或函数。定义LEADING删除前导空格，定义TRAILING删除尾随空格，定义BOTH同时删除前导和尾随空格。如果不指定，默认为BOTH（代码5-17和图5-17）。

143

**代码5-17** 这个查询分别删除了串'AAA'的前导、尾随以及全部前导和尾随空格。<和>字符显示删除空格以后串的范围。结果见图5-17

```
SELECT
 '< ' || ' AAA ' || '>' AS "Untrimmed",
 '<' || TRIM(LEADING FROM ' AAA ') || '>' AS "Leading",
 '<' || TRIM(TRAILING FROM ' AAA ') || '>' AS "Trailing",
 '<' || TRIM(' AAA ') || '>' AS "Both",
```

| Untrimmed | Leading | Trailing | Both  |
|-----------|---------|----------|-------|
| < AAA >   | <AAA>   | < AAA>   | <AAA> |

图5-17 运行代码5-17的结果

### ⇒ 修整串中的字符

输入：

```
TRIM([LEADING | TRAILING | BOTH]
 'trim_chars' FROM string)
```

*string*是要修整的串，*trim\_chars*是要从*string*中删除的一个或多个字符。每一个参数是一个串

表达式（如包含字符串的列、串字面量，或者返回串结果的运算或函数）。定义LEADING删除前导字符，定义TRAILING删除尾随字符，定义BOTH同时删除前导和尾随字符。如果不指定，默认为BOTH（代码5-18和代码5-19、图5-18和图5-19）。

**代码5-18** 删除以H开头的作者姓中前导的H。结果见图5-18

```
144
SELECT au_lname,
 TRIM(LEADING 'H' FROM au_lname)
 AS "Trimmed name"
 FROM authors;
```

**代码5-19** 列出忽略前导和尾随空格的、以T1开头的3个字符的title\_id。结果见图5-19

```
SELECT title_id
 FROM titles
 WHERE TRIM(title_id) LIKE 'T1_';
```

✓ 提示

- 可以在SELECT、WHERE和ORDER BY子句中或任何允许使用表达式的位置使用TRIM()。
- 在本章前面的代码5-8中，将作者的名和姓连接成一个列。图5-8的结果在作者Kellsey的前面包含额外的空格。因为Kellsey没有名，每一行中分隔名和姓的空格将显示出来。可以使用TRIM()删除这个前导空格。更改代码5-8中的连接表达式为

```
TRIM(au_fname || ' ' || au_lname)
```

- **DBMS** 在Microsoft Access中，用于修整前导空格的修整函数是LTrim(string)，用于修整尾随空格的是RTrim(string)，用于同时修整前导和尾随空格的是Trim(string)。使用函数Replace(string, find, replacement [, start[, count[, compare]]])修整非空格字符（实际上是用空串替代非空格字符）。使用+连接串。要运行代码5-17和代码5-18，更改修整表达式为（代码5-17）

```
'<' + ' AAA ' + '>'
'<' + LTRIM(' AAA ') + '>'
'<' + RTRIM(' AAA ') + '>'
'<' + TRIM(' AAA ') + '>'
```

对于代码5-18：

```
Replace(au_lname, 'H', '', 1, 1)
```

在Microsoft SQL Server中，用于修整前导空格的修整函数是LTRIM(string)，修整尾随空格的是RTRIM(string)。使用+连接串。要运行代码5-17，更改修整表达式为

```
'<' + ' AAA ' + '>'
'<' + LTRIM(' AAA ') + '>'
'<' + RTRIM(' AAA ') + '>'
```

| au_lname    | Trimmed name |
|-------------|--------------|
| Buchman     | Buchman      |
| Heydemark   | eydemark     |
| Hull        | ull          |
| Hull        | ull          |
| Kells       | Kells        |
| Kellsey     | Kellsey      |
| O'Furniture | O'Furniture  |

图5-18 运行代码5-18的结果

| title_id |
|----------|
| T10      |
| T11      |
| T12      |
| T13      |

图5-19 运行代码5-19的结果

```
'<' + LTRIM(RTRIM(' AAA ')) + '>'
```

SQL Server的LTRIM()和RTRIM()函数只能删除空格，而非任意的*trim\_chars*字符。可以嵌套和连接SQL的CHARINDEX()、LEN()、PATINDEX()、REPLACE()、STUFF()、SUBSTRING()及其他字符函数来实现任意字符修整。要运行代码5-18，更改修整表达式为

```
REPLACE(SUBSTRING(au_lname, 1, 1), 'H', '')
→ + SUBSTRING(au_lname, 2, LEN(au_lname))
```

要运行代码5-19，更改修整表达式为

```
LTRIM(RTRIM(title_id)) LIKE 'T1_'
```

在Oracle中，添加子句FROM DUAL运行代码5-17，参见5.1节。在*trim\_chars*中，Oracle禁止使用多个字符。

在DB2中，用于修整前导空格的修整函数是LTRIM(*string*)，修整尾随空格的是RTRIM(*string*)。  
要运行代码5-17，更改修整表达式为：

```
'<' || ' AAA ' || '>'
'<' || LTRIM(' AAA ') || '>'
'<' || RTRIM(' AAA ') || '>'
'<' || LTRIM(RTRIM(' AAA ')) || '>'
```

还必须添加子句FROM SYSIBM.SYSDUMMY1到代码5-17中，参见5.1节。

你可以嵌套或连接DB2的LENGTH()、LOCATE()、PSSTR()、REPLACE()、SUBSTR()及其他字符函数，来实现任意的字符修整。要运行代码5-18，更改修整表达式为

```
REPLACE(SUBSTR(au_lname, 1, 1), 'H', '')
→ || SUBSTR(au_lname, 2, LENGTH(au_lname))
```

要运行代码5-19，更改修整表达式为

```
LTRIM(RTRIM(title_id)) LIKE 'T1_'
```

在MySQL中，使用CONCAT()运行代码5-17（见5.4节）。更改连接表达式为

```
CONCAT('<', ' AAA ', '>')
CONCAT('<', TRIM(LEADING FROM ' AAA '), '>')
CONCAT('<', TRIM(TRAILING FROM ' AAA '), '>')
CONCAT('<', TRIM(' AAA '), '>')
```

Oracle将空值作为空串处理：TRIM(NULL)返回''，参见3.14节中的DBMS提示。

有的DBMS可能提供了填充函数，为串添加空格或其他的字符。例如，Oracle和PostgreSQL的填充函数是LPAD()和RPAD()，查阅DBMS文档中的character functions和string functions。

## 5.8 使用 CHARACTER\_LENGTH()得到串长度

使用函数CHARACTER\_LENGTH()返回串中字符的个数。这个函数的主要特点如下。

- CHARACTER\_LENGTH()返回一个大于或等于零的整数。
- CHARACTER\_LENGTH()统计的是字符数，而不是字节数。多字节字符或Unicode字符表示一个字符（要统计字节数，见本节提示）。
- 空串（''）的长度是零。

- 如果参数为空，`CHARACTER_LENGTH()`将返回空值（但Oracle例外，参见本节DBMS提示）。

### ⇒ 得到串的长度

输入：

`CHARACTER_LENGTH(string)`

`string`是串表达式，如包含字符串的列、串字面量，或者返回串结果的运算或函数（代码5-20至代码147 5-21、图5-20和图5-21）。

**代码5-20** 列出作者名的长度。结果见图5-20

```
SELECT au_fname,
 CHARACTER_LENGTH(au_fname) AS "Len"
 FROM authors;
```

**代码5-21** 列出书名少于30个字符的书，并按书名长度升序排序。结果见图5-21

```
SELECT title_name,
 CHARACTER_LENGTH(title_name) AS "Len"
 FROM titles
 WHERE CHARACTER_LENGTH(title_name) < 30
 ORDER BY CHARACTER_LENGTH(title_name) ASC;
```

#### ✓ 提示

- 可以在SELECT、WHERE和ORDER BY子句中或任何允许使用表达式的位置使用`CHARACTER_LENGTH()`。
- `CHARACTER_LENGTH()`和`CHAR_LENGTH()`是同义词。
- SQL也定义了`BIT_LENGTH()`和`OCTET_LENGTH()`函数。

`BIT_LENGTH(expr)`返回表达式中的位的个数，

`BIT_LENGTH(B'01001011')`返回8。

`OCTET_LENGTH(expr)`也返回表达式中字节的个数，

`OCTET_LENGTH(B'01001011')`返回1，`OCTET_LENGTH ('ABC')`返回3。

八进制的长度等于位的长度除以8（如果必要，近似为一个整数）。参见本节DBMS提示中关于DBMS的位长度和字节长度函数的信息。

- **DBMS** 在Microsoft Access和Microsoft SQL Server中，串长度函数是`LEN(string)`。要运行代码5-20和代码5-21，更改长度表达式为（代码5-20）

`LEN(au_fname)`

对于代码5-21：

`LEN(title_name)`

在Oracle和DB2中，串长度函数是`LENGTH(string)`。要运行代码5-20和代码5-21，更改长度表

| au_fname  | Len |
|-----------|-----|
| Sarah     | 5   |
| Wendy     | 5   |
| Hallie    | 6   |
| Klee      | 4   |
| Christian | 9   |
|           | 0   |
| Paddy     | 5   |

图5-20 运行代码5-20的结果

| title_name                    | Len |
|-------------------------------|-----|
| 1977!                         | 5   |
| Kiss My Boo-Boo               | 15  |
| How About Never?              | 16  |
| I Blame My Mother             | 17  |
| Exchange of Platitudes        | 22  |
| 200 Years of German Humor     | 25  |
| Spontaneous, Not Annoying     | 25  |
| But I Did It Unconsciously    | 26  |
| Not Without My Faberge Egg    | 26  |
| Just Wait Until After School  | 28  |
| Ask Your System Administrator | 29  |

图5-21 运行代码5-21的结果

达式为（代码5-20）

```
LENGTH(au_fname)
```

对于代码5-21：

```
LENGTH(title_name)
```

位统计和字节统计函数因DBMS而异。Microsoft Access有Len()，Microsoft SQL Server有DATALENGTH()，Oracle有LENGTHB()，DB2有LENGTHC()，MySQL有BIT\_COUNT()和OCTET\_LENGTHC()，PostgreSQL有BIT\_LENGTHC()和OCTET\_LENGTHC()。

Oracle将空串作为空值处理：LENGTH('')返回NULL。图5-20将在倒数第2行显示1（非0），因为在Oracle数据库中作者的名是''（空格）。要获得更多信息，参见3.14节中的DBMS提示。

148

5

## 5.9 使用 POSITION()查找子串

使用函数POSITION()在给定串中定位一个特定的子串。这个函数的主要特点如下。

- POSITION()返回一个大于或等于零的整数，表示在串中子串第一次出现的起始位置。
- 如果串中不含有子串，POSITION()将返回零。
- 串比较是否区分大小写依赖于DBMS，参见4.5节中的DBMS提示。
- 任何子串在空串（''）中的位置是零（但Oracle例外，参见本节DBMS提示）。
- 如果参数为空值，POSITION()将返回空值。

149

### ⇒ 查找子串

输入：

```
POSITION(substring IN string)
```

*substring*是要查找的串，*string*是被查找的串。每一个参数是一个串表达式，如包含字符串的列、串字面量，或者返回串的运算或函数的结果。POSITION()将返回*Substring*在*String*中出现的最小整数位置；如果没有找到子串，则返回零（代码5-22至代码5-23、图5-22至图5-23）。

**代码5-22** 列出子串e在作者名中的位置，列出子串ma在作者姓中的位置。结果见图5-22

```
SELECT
 au_fname,
 POSITION('e' IN au_fname) AS "Pos e",
 au_lname,
 POSITION('ma' IN au_lname) AS "Pos ma"
FROM authors;
```

**代码5-23** 列出在书名的前10个字符中包含字母u的书，并对标题按字母u所在的位置降序排序。结果见图5-23

```
SELECT title_name,
 POSITION('u' IN title_name) AS "Pos"
```

| au_fname  | Pos e | au_lname    | Pos ma |
|-----------|-------|-------------|--------|
| Sarah     | 0     | Buchman     | 5      |
| Wendy     | 2     | Heydermark  | 6      |
| Hallie    | 6     | Hull        | 0      |
| Klee      | 3     | Hull        | 0      |
| Christian | 0     | Kells       | 0      |
|           |       | Kellsey     | 0      |
| Paddy     | 0     | O'Furniture | 0      |

150

图5-22 运行代码5-22的结果

| title_name                    | Pos |
|-------------------------------|-----|
| Not Without My Faberge Egg    | 10  |
| Spontaneous, Not Annoying     | 10  |
| How About Never?              | 8   |
| Ask Your System Administrator | 7   |
| But I Did It Unconsciously    | 2   |
| Just Wait Until After School  | 2   |

图5-23 运行代码5-23的结果

```
FROM titles
WHERE POSITION('u' IN title_name)
 BETWEEN 1 AND 10
ORDER BY POSITION('u' IN title_name) DESC;
```

✓ 提示

- 可以在SELECT、WHERE和ORDER BY子句中或任何允许使用表达式的位置使用POSITION()。
- SQL标准也定义了函数OVERLAY()以替换子串。语法如下：

```
OVERLAY(string PLACING substring
 FROM start_position [FOR length])
```

例如，OVERLAY('Txxxxas' PLACING 'hom' FROM 2 FOR 4)是'Thomas'。在DBMS中等价的函数是REPLACE()（在Microsoft Access、Microsoft SQL Server、DB2和MySQL中）、REGEXP\_REPLACE()（在Oracle中）和OVERLAY()（在PostgreSQL中）。

- **DBMS** 在Microsoft Access中，位置函数是InStr(start\_position, string, substring)。要运行代码5-22和代码5-23，更改位置表达式为（代码5-22）

```
InStr(1, au_fname, 'e')
InStr(1, au_lname, 'ma')
```

对于代码5-23：

```
InStr(1, title_name, 'u')
```

在Microsoft SQL Server中，位置函数是CHARINDEX(substring, string)。要运行代码5-22和代码5-23，更改位置表达式为（代码5-22）

```
CHARINDEX('e', au_fname)
CHARINDEX('ma', au_lname)
```

对于代码5-23：

```
I CHARINDEX('u', title_name)
```

在Oracle中，位置函数是INSTR(string, substring)。要运行代码5-22和代码5-23，更改位置表达式为（代码5-22）

```
INSTR(au_fname, 'e')
INSTR(au_lname, 'ma')
```

对于代码5-23：

```
INSTR(title_name, 'u')
```

在DB2中，位置函数是POSSTR(string, substring)。要运行代码5-22和代码5-23，更改位置表达式为（代码5-22）

```
POSSTR(au_fname, 'e')
POSSTR(au_lname, 'ma')
```

对于代码5-23：

```
POSSTR(title_name, 'u')
```

Oracle将空串作为空值处理。INSTR(' ', substring)返回空值（不是0），参见3.14节中的DBMS提示。

可以嵌套和连接子串及位置函数以查找出现超过一次的子串，但DBMS提供了增强的位置函数。Microsoft Access有InStr()，Microsoft SQL Server有CHARINDEX()，Oracle有INSTR()，DB2有LOCATE()，MySQL有LOCATE()。

## 5.10 执行日期及时间间隔运算

**DBMS** 对标准的SQL日期及时间间隔操作符，和函数的符合程度是不一致的，因为DBMS通常提供了它们自己扩展的（非标准的）执行日期和时间运算的操作符和函数。关于日期和时间及时间间隔数据类型的更多信息，参见3.10节和3.11节。

使用与在本章前面介绍的“执行算术运算”中相同的操作符，执行日期及时间间隔运算。

普通的时间运算如下。

- 两个日期相减以计算它们之间的时间间隔。
- 对一个日期加上或减去一个时间间隔以得到将来或过去的日期。
- 对两个时间间隔相加或相减以得到一个新的时间间隔。
- 对一个时间间隔乘以或除以一个数字以得到一个新的时间间隔。

某些运算是未定义的。例如，两个日期相加是没有意义的。表5-3所示为包含日期及时间间隔的有效SQL操作符。在本节的“操作符重载”提要栏中解释了为什么执行不同的运算可以使用相同的操作符。

表5-3 日期和时间及时间间隔运算

| 运 算                 | 结 果  | 运 算                 | 结 果  |
|---------------------|------|---------------------|------|
| Datetime - Datetime | 时间间隔 | Interval - Interval | 时间间隔 |
| Datetime + Interval | 日期   | Interval * Numeric  | 时间间隔 |
| Datetime - Interval | 日期   | Interval / Numeric  | 时间间隔 |
| Interval + Datetime | 日期   | Numeric * Interval  | 时间间隔 |
| Interval + Interval | 时间间隔 |                     |      |

函数EXTRACT()将日期或时间间隔隔离为单一的字段，并以数字形式返回。EXTRACT()通常用在比较表达式中或者用于格式化结果。

### 操作符重载

回顾+、-、\*和/操作符也被用于数字运算，并且Microsoft DBMS还将+用于串的连接。操作符重载是给特定的操作符分配多个功能。运算的执行依赖于包含的操作数的数据类型。这里的+、-、\*和/操作符，对于数字的行为和对于日期及时间间隔（Microsoft环境中的串）的行为不同。DBMS也可能重载其他的操作符和函数。函数重载是给特定的函数分配多个行为（依赖于所含参数的数据类型）。例如，MySQL的CONCAT()函数（参见5.4节中的DBMS提示）既接受串参数也接受非串参数。非串参数导致CONCAT()去执行（不需要对串进行）附加转换。

#### » 提取日期或时间间隔的一部分

输入：

`EXTRACT(field FROM datetime_or_interval)`

`field`是要返回的`datetime_or_interval`的一部分。`field`是YEAR、MONTH、DAY、HOUR、MINUTE、SECOND、TIMEZONE\_HOUR或TIMEZONE\_MINUTE（见第3章中的表3-14）。`datetime_or_interval`是一个日期或时间间隔表达式，如包含日期或时间间隔值的列、日期或时间间隔字面量，或者返回日期或时间间隔结果的

运算或函数。如果 *field* 是 SECOND，EXTRACT() 返回一个 NUMERIC 值；否则，它返回一个 INTEGER 值（代码 5-24 和图 5-24）。

**代码 5-24** 列出在 2001 年上半年和 2002 年上半年出版的书，并按出版日期降序排序。结果见图 5-24

```
SELECT
 title_id,
 pubdate
FROM titles
WHERE EXTRACT(YEAR FROM pubdate)
 BETWEEN 2001 AND 2002
 AND EXTRACT(MONTH FROM pubdate)
 BETWEEN 1 AND 6
ORDER BY pubdate DESC;
```

#### ✓ 提示

- 可以在 SELECT、WHERE 和 ORDER BY 子句中或任何允许使用表达式的位置使用时间操作符和函数。
- 如果任何一个操作数或参数为空，则表达式返回空值。
- 也可以参见 15.11 节。
- **DBMS** 在 Microsoft Access 和 Microsoft SQL Server 中，提取函数是 DATEPART(*datepart*, *date*)。要运行代码 5-24，更改提取表达式为

```
DATEPART("yyyy", pubdate)
DATEPART("m", pubdate)
```

Oracle、MySQL 和 PostgreSQL 对于 EXTRACT() 中的 *field* 参数可以接受不同的或附加的值。

DB2 通过使用单独的函数（如 DAY()、HOUR() 和 SECOND()）提取一部分日期、时间间隔，来替代 EXTRACT()。要运行代码 5-24，更改提取表达式为

```
YEAR(pubdate)
MONTH(pubdate)
```

除了标准的算术操作符以外，DBMS 还提供了对日期添加时间间隔的函数。例如，Microsoft Access 和 Microsoft SQL Server 中的 DATEDIFF()，Oracle 中的 ADD\_MONTHS()，MySQL 中的 DATE\_ADD() 和 DATE\_SUB()。

复杂的日期和时间运算在 SQL 编程中非常普遍，因此所有的 DBMS 都提供了扩展的时间函数。

请查阅 DBMS 文档中的 date and time functions 和 datetime functions。

153

| title_id | pubdate    |
|----------|------------|
| T09      | 2002-05-31 |
| T08      | 2001-06-01 |
| T05      | 2001-01-01 |

图 5-24 运行代码 5-24 的结果

## 5.11 获得当前日期和时间

使用函数 CURRENT\_DATE、CURRENT\_TIME 和 CURRENT\_TIMESTAMP，从 DBMS 运行着的特定计算机系统时钟上获得当前日期和时间。

### ⇒ 获得当前日期和时间

要得到当前日期，输入：

```
CURRENT_DATE
```

要得到当前时间，输入：

```
CURRENT_TIME
```

要得到当前的时间戳 (timestamp)，输入：

CURRENT\_TIMESTAMP

CURRENT\_DATE 返回 DATE，CURRENT\_TIME 返回 TIME，CURRENT\_TIMESTAMP 返回 TIMESTAMP，参见 3.10 节（代码 5-25 和代码 5-26、图 5-25 和图 5-26）。

**代码 5-25** 打印当前的日期、时间和时间戳。结果见图 5-25

```
SELECT
 CURRENT_DATE AS "Date",
 CURRENT_TIME AS "Time",
 CURRENT_TIMESTAMP AS "Timestamp";
```

| Date       | Time     | Timestamp           |
|------------|----------|---------------------|
| 2002-03-10 | 10:09:24 | 2002-03-10 10:09:24 |

图 5-25 运行代码 5-25 的结果

**代码 5-26** 列出出版日期在当前日期的前后各 90 天的范围之内的书或出版日期未知的书，并按出版日期降序排列（这个查询的当前日期，参考图 5-25）。结果见图 5-26

```
SELECT title_id, pubdate
 FROM titles
 WHERE pubdate
 BETWEEN CURRENT_TIMESTAMP
 - INTERVAL 90 DAY
 AND CURRENT_TIMESTAMP
 + INTERVAL 90 DAY
 OR pubdate IS NULL
 ORDER BY pubdate DESC;
```

| title_id | pubdate    |
|----------|------------|
| T09      | 2002-05-31 |
| T10      | NULL       |

图 5-26 运行代码 5-26 的结果

### ✓ 提示

- 可以在 SELECT、WHERE 和 ORDER BY 子句中或任何允许使用表达式的位置，使用日期和时间函数。
- CURRENT\_TIME 和 CURRENT\_TIMESTAMP 都可以接受精度参数（定义包含在时间中的秒的十进制小数。例如，CURRENT\_TIME(6) 返回 SECOND 字段中有 6 位数字精度的当前时间。关于精度的更多信息，可参见 3.10 节。）
- 也可以参见 15.11 节。
- 在 Microsoft Access 中，日期系统函数是 DATE()、TIME() 和 NOW()。要运行代码 5-25，更改日期表达为

Date() AS "Date"  
Time() AS "Time"  
Now() AS "Timestamp"

要运行代码 5-26，更改 BETWEEN 子句为

BETWEEN Now() - 90  
 AND Now() + 90

在 Microsoft SQL Server 中，日期系统函数是 CURRENT\_TIMESTAMP() 或者它的同义词 GETDATE()，不支持 CURRENT\_DATE 和 CURRENT\_TIME。要运行代码 5-25，省略 CURRENT\_DATE 和 CURRENT\_TIME 表达式。要运行代码 5-26，更改 BETWEEN 子句为

BETWEEN CURRENT\_TIMESTAMP - 90  
 AND CURRENT\_TIMESTAMP + 90

在 Oracle 中，日期系统函数是 SYSDATE。Oracle 9i 和之后的版本支持 CURRENT\_DATE 和 CURRENT\_

`TIMESTAMP`（但不支持`CURRENT_TIME`）。代码5-25也需要`FROM DUAL`子句，参见5.1节中的DBMS提示。要运行代码5-25，更改语句为

```
SELECT SYSDATE AS "Date"
 FROM DUAL;
```

`SYSDATE`返回系统日期和时间，但是不显示时间，除非用函数`TOCHAR()`来格式化它：

```
SELECT TO_CHAR(SYSDATE,
 → 'YYYY-MM-DD HH24:MI:SS')
 FROM DUAL;
```

要运行代码5-26，更改`BETWEEN`子句为

```
BETWEEN SYSDATE - 90
 AND SYSDATE + 90
```

要在DB2中运行代码5-25，添加子句`FROM SYSIBM.SYSDUMMY1`，参见5.1节中的DBMS提示。要运行代码5-26，更改`WHERE`子句为

```
BETWEEN CURRENT_DATE - 90 DAYS
 AND CURRENT_DATE + 90 DAYS
```

要在PostgreSQL中运行代码5-26，更改`WHERE`子句为

```
BETWEEN CURRENT_TIMESTAMP - 90
 AND CURRENT_TIMESTAMP + 90
```

关于日期系统函数的信息，请查见DBMS文档中的“date and time functions”和“system functions”。

155

## 5.12 获得用户信息

使用函数`CURRENT_USER`标识数据库服务器中的活动用户。

### ⇒ 获得当前用户

输入：

```
CURRENT_USER
```

**代码5-27 显示当前用户。结果见图5-27**

```
SELECT CURRENT_USER AS "User";
```

### ✓ 提示

可以在`SELECT`、`WHERE`和`ORDER BY`子句中或任何允许使用表达式的位置，使用用户函数。

SQL也定义了`SESSION_USER`和`SYSTEM_USER`函数。当前用户表示授权标识符（SQL语句当前运行在它的授权之下）。当前用户有权限运行，比如说只有`SELECT`语句。会话用户表示和当前会话有关的授权ID。系统用户是被宿主操作系统标识的用户。DBMS确定用户值，这3个值可以一样也可以不一样。关于用户、会话和权限的更多信息，请查阅DBMS文档中的`authorization`、`sessions`、`user`或者`role`。

参见15.10节。

**DBMS** 要在Microsoft Access中运行代码5-27，更改语句为

| User             |
|------------------|
| -----<br>cfehily |

图5-27 运行代码5-27的结果

```
SELECT CurrentUser AS "User";
```

要在Oracle中运行代码5-27，更改语句为

```
SELECT USER AS "User" FROM DUAL;
```

要在DB2中运行代码5-27，更改语句为

```
SELECT CURRENT_USER AS "User"
FROM SYSIBM.SYSDUMMY1;
```

要在MySQL中运行代码5-27，更改语句为

```
SELECT USER() AS "User";
```

Microsoft SQL Server支持SESSION\_USER和SYSTEM\_USER。

MySQL支持SESSION\_USER()和SYSTEM\_USER()。Oracle的SYS\_CONTEXT()返回会话的用户属性。DB2支持SESSION\_USER和SYSTEM\_USER。PostgreSQL支持SESSION\_USER。关于用户系统函数的信息，查阅DBMS文档中的user或system functions。

156  
5

## 5.13 使用CAST()转换数据类型

在多数情况下，DBMS将自动转换数据类型。例如，它允许在字符表达式（如连接）中使用数字和日期，或者在混合算术表达式中自动提升数字（见5.2节中的提示）。当DBMS不能自动执行转换时，可以使用函数CAST()将一种数据类型的表达式转换为另一种数据类型。关于数据类型的信息，见3.4节。这个函数的主要特点如下。

- 隐式转换（implicit conversion）或强制转换（coercion）是那些发生在没有指定CAST()时的转换。显式转换（explicit conversion）是那些需要指定CAST()的转换。在某些情况下，转换是不允许的。例如，不可以将FLOAT转换为TIMESTAMP。
- 被转换的数据类型是源数据类型，结果数据类型是目标数据类型。
- 可以将任意的数字或日期数据类型转换为字符数据类型。
- 可以将字符数据类型转换为任何一种数据类型，只要字符串可以表示有效的目标数据类型的字面量（当将串转换为数字或日期值时，DBMS将删除前导和尾随空格）。
- 某些数字转换，比如将DECIMAL转换为INTEGER，将舍入或者截断值（值被舍入还是截断依赖于DBMS）。
- 将VARCHAR转换为CHAR会截断串。
- 如果新的数据类型没有足够的空间来显示转换的值，则转换可能引发错误。如果浮点数超出DBMS所允许的SMALLINT值的范围，将FLOAT转换为SMALLINT将会失败。
- 将NUMERIC转换为DECIMAL需要显式地转换，以防隐式转换中可能产生的精度或小数位数的损失。
- 从DATE到TIMESTAMP的转换中，结果中的时间部分可能是00:00:00（午夜）。
- 如果参数为空，CAST()将返回空值（但Oracle例外，参见本节DBMS提示）。

157

⇒ 将一种数据类型转换为另一种

输入：

```
CAST(expr AS data_type)
```

*expr*是要转换的表达式, *data\_type*是目标数据类型。*data\_type*在第3章中介绍过, 可以包含适用长度、精度和小数位数的数据类型之一。例如, 可以接受的*data\_type*值包含CHAR(10)、VARCHAR(25)、NUMERIC(5, 2)、INTEGER、FLOAT和DATE。如果*expr*的数据类型或值和*data\_type*不兼容的话, 将产生错误(代码5-28和代码5-29及图5-28a、图5-28b和图5-29)。

**代码5-28** 将书的价格从DECIMAL数据类型转换为INTEGER和CHAR(8)数据类型。<和>字符表示了CHAR(8)串的范围。结果见图5-28a或图5-28b, 这依赖于DBMS取整是舍入还是截断

```
SELECT
 price
 AS "price(DECIMAL)",
 CAST(price AS INTEGER)
 AS "price(INTEGER)",
 '<' || CAST(price AS CHAR(8)) || '>'
 AS "price(CHAR(8))"
FROM tiles;
```

| price(DECIMAL) | price(INTEGER) | price(CHAR(8)) |
|----------------|----------------|----------------|
| 21.99          | 21 <21.99 >    |                |
| 19.95          | 19 <19.95 >    |                |
| 39.95          | 39 <39.95 >    |                |
| 12.99          | 12 <12.99 >    |                |
| 6.95           | 6 <6.95 >      |                |
| 19.95          | 19 <19.95 >    |                |
| 23.95          | 23 <23.95 >    |                |
| 10.00          | 10 <10.00 >    |                |
| 13.95          | 13 <13.95 >    |                |
| NULL           | NULL NULL      |                |
| 7.99           | 7 <7.99 >      |                |
| 12.99          | 12 <12.99 >    |                |
| 29.99          | 29 <29.99 >    |                |

| price(DECIMAL) | price(INTEGER) | price(CHAR(8)) |
|----------------|----------------|----------------|
| 21.99          | 22 <21.99 >    |                |
| 19.95          | 20 <19.95 >    |                |
| 39.95          | 40 <39.95 >    |                |
| 12.99          | 13 <12.99 >    |                |
| 6.95           | 7 <6.95 >      |                |
| 19.95          | 20 <19.95 >    |                |
| 23.95          | 24 <23.95 >    |                |
| 10.00          | 10 <10.00 >    |                |
| 13.95          | 14 <13.95 >    |                |
| NULL           | NULL NULL      |                |
| 7.99           | 8 <7.99 >      |                |
| 12.99          | 13 <12.99 >    |                |
| 29.99          | 30 <29.99 >    |                |

图5-28a 运行代码5-28的结果。如果在转换它们为整数时DBMS截断十进制数字, 则得到这个结果

图5-28b 运行代码5-28的结果。如果在转换它们为整数时DBMS舍入十进制数字, 则得到这个结果

**代码5-29** 列出历史和传记书的sales(销售数量)和图书书名的一部分, 并按sales降序排序。CHAR(20)转换缩短了书名, 使得结果更容易读。结果见图5-29

```
SELECT
 CAST(sales AS CHAR(8))
 || ' copies sold of '
 || CAST(title_name AS CHAR(20))
 AS "History and biography sales"
```

#### History and biography sales

|         |                                     |
|---------|-------------------------------------|
| 1500200 | copies sold of I Blame My Mother    |
| 100001  | copies sold of Spontaneous, Not Ann |
| 11320   | copies sold of How About Never?     |
| 10467   | copies sold of What Are The Civilia |
| 9566    | copies sold of 200 Years of German  |
| 566     | copies sold of 1977!                |

图5-29 运行代码5-29的结果

```
FROM titles
WHERE sales IS NOT NULL
 AND type IN ('history', 'biography')
ORDER BY sales DESC;
```

159

### ✓ 提示

- 可以在SELECT、WHERE和ORDER BY子句中或者允许任何使用表达式的位置，使用CAST()。
- 扩大转换(widening conversion)是指那些没有数据丢失或不正确结果的转换。例如，将SMALLINT转换为INTEGER，因为INTEGER数据类型可以容纳SMALLINT数据类型所有可能的值，所以是扩大转换。相反的转换称为缩小转换，因为极端的INTEGER值不能用SMALLINT表示，所以会引起数据丢失。扩大转换总是被允许的，但缩小转换可能导致DBMS产生警告或者错误。
- **DBMS** 在Microsoft Access中有一组类型转换函数而不是只有CAST()函数，例如CStr(expr)、CInt(expr)和CDec(expr)分别将expr转换为串、整数和十进制数字。可以使用Space(number)为串添加空格，使用Left(string, length)截断串。使用+连接串。要运行代码5-28和代码5-29，更改转换表达式为（代码5-28）

```
CInt(price)
'<' + CStr(price) + '>'
```

对于代码5-29：

```
CStr(sales)
→ + Space(8 - Len(CStr(sales)))
→ + ' copies sold of '
→ + Left(title_name, 20)
```

在Microsoft SQL Server中，使用+连接串（代码5-28）

```
'<' + CAST(price AS CHAR(8)) + '>'
```

对于代码5-29：

```
CAST(sales AS CHAR(8))
→ + ' copies sold of '
→ + CAST(title_name AS CHAR(20))
```

在Oracle中，如果length比源串短，则不允许使用CHAR(length)字符转换，改为使用SUBSTR()截断串，参见5.5节中的DBMS提示。要运行代码5-29，更改CAST()表达式为

```
CAST(sales AS CHAR(8))
→ || ' copies sold of '
→ || SUBSTR(title_name, 1, 20)
```

在MySQL中，对于data\_type使用SIGNED替代INTEGER，使用CONCAT()连接串。要运行代码5-28和代码5-29，更改CAST()表达式为（代码5-28）

```
CAST(price AS SIGNED)
CONCAT('<', CAST(price AS CHAR(8)),
→ '>')
```

对于代码5-29：

```
CONCAT(
→ CAST(sales AS CHAR(8)),
→ ' copies sold of ',
→ CAST(title_name AS CHAR(20)))
```

Oracle将空串作为空值处理：CAST('' AS CHAR)将返回空值，参见3.14节中的DBMS提示。

在PostgreSQL较早的版本中，要将NUMERIC或DECIMAL列的值和实数（浮点数）进行比较，必须将实数显式地转换为NUMERIC或DECIMAL。例如，在PostgreSQL较早的版本中，因为列price的数据数据类型是DECIMAL(5,2)，下面的语句将执行失败：

```
SELECT price
 FROM titles
 WHERE price < 20.00;
```

这条语句解决了这个问题：

```
SELECT price
 FROM titles
 WHERE price < CAST(20.00 AS
 DECIMAL);
```

DBMS有附加的转换和格式化函数。例如，Microsoft SQL Server和MySQL中的CONVERT()，Oracle和PostgreSQL中的TO\_CHAR()、TO\_DATE()、TO\_TIMESTAMP()和TO\_NUMBER()，DB2中的TO\_CHAR()和TO\_DATE()，查阅DBMS文档中的conversion、cast或formatting functions。

160

## 5.14 使用CASE计算条件值

CASE表达式和它的缩写COALESCE()和NULLIF()，可以基于条件的真值（真、假或未知）采取行动。CASE表达式的主要特点如下。

- 如果编写过程序，你就会意识到SQL的CASE是过程语言中if-then-else、case或switch语句等价物，只是CASE是表达式而非语句。
- CASE被用于计算几个条件的值，并且对于第一个为真的条件返回一个值。
- CASE允许对列中的实际值显示一个可选的值。CASE不会更改数据本身。
- CASE常用更容易读的值代替编码或缩写词。如果列marital\_status包含整数代码1、2、3或者4，它分别意味着单身的、已婚的、离婚的或者寡居的，读者愿意去看解释性文本而不是编码（有些数据库设计人员喜欢使用编码，因为存储和管理小型编码要比解释性文本更快）。
- CASE有两种格式——简单格式或搜索格式。简单CASE表达式将一个表达式与一个简单表达式的集合进行比较以确定结果。搜索CASE表达式计算一个逻辑（布尔）表达式的集合以确定结果。
- 如果没有测试条件为真，则CASE返回可选的ELSE结果作为默认值。

161

### ⇒ 使用简单CASE表达式

输入：

```
CASE comparison_value
 WHEN value1 THEN result1
 WHEN value2 THEN result2
 ...
 WHEN valueN THEN resultN
 [ELSE default_result]
END
```

*value1, value2, …, valueN*是表达式。*result1, result2, …, resultN*是在相应的值与表达式 *comparison\_value*相匹配时返回的表达式。所有的表达式必须是一样的类型或者隐式地转换为一样的类型。

每个值按顺序和 *comparison\_value*比较。首先，比较 *value1*，如果它和 *comparison\_value*匹配，返回 *result1*；否则，*value2*和 *comparison\_value*比较，如果 *value2*和 *comparison\_value*匹配，返回 *result2*，以此类推。如果没有匹配，则返回 *default\_result*。如果省略 *ELSE default\_result*，默认 *ELSE NULL*（代码5-30和图5-30）。

162

5

**代码5-30** 将历史书的价格提高10%，将心理学书的价格提高20%，其他书的价格不变。结果见图5-30

```

SELECT
 title_id,
 type,
 price,
 CASE type
 WHEN 'history'
 THEN price * 1.10
 WHEN 'psychology'
 THEN price * 1.20
 ELSE price
 END
 AS "New price"
FROM titles
ORDER BY type ASC, title_id ASC;

```

⇒ 使用搜索CASE表达式

输入：

```

CASE
WHEN condition1 THEN result1
WHEN condition2 THEN result2
...
WHEN conditionN THEN resultN
[ELSE default_result]
END

```

|     | title_id   | type  | price | New price |
|-----|------------|-------|-------|-----------|
| T06 | biography  | 19.95 | 19.95 |           |
| T07 | biography  | 23.95 | 23.95 |           |
| T10 | biography  | NULL  | NULL  |           |
| T12 | biography  | 12.99 | 12.99 |           |
| T08 | children   | 10.00 | 10.00 |           |
| T09 | children   | 13.95 | 13.95 |           |
| T03 | computer   | 39.95 | 39.95 |           |
| T01 | history    | 21.99 | 24.19 |           |
| T02 | history    | 19.95 | 21.95 |           |
| T13 | history    | 29.99 | 32.99 |           |
| T04 | psychology | 12.99 | 15.59 |           |
| T05 | psychology | 6.95  | 8.34  |           |
| T11 | psychology | 7.99  | 9.59  |           |

图5-30 运行代码5-30的结果

*condition1, condition2, …, conditionN*是搜索表达式（搜索条件是一个或多个逻辑表达式，而它又是以AND或OR连接的复合表达式组成的，见4.5节）。*result1, result2, …, resultN*是当相应的条件值为真时返回的表达式。所有的表达式必须是一样的类型或者隐式地转换为一样的类型。

每个条件按顺序计算。首先，计算 *condition1*，如果为true，返回 *result1*；否则，计算 *condition2*。如果 *condition2*为true，返回 *result2*，以此类推。如果没有条件为真，则返回 *default\_result*。如果省略 *ELSE default\_result*，默认 *ELSE NULL*（代码5-31和图5-31）。

**代码5-31** 列出按不同的销售量范围分类的书，并按销售量升序排序。结果见图5-31

```

SELECT
 title_id,
 CASE

```

```

WHEN sales IS NULL
 THEN 'Unknown'
WHEN sales <= 1000
 THEN 'Not more than 1,000'
WHEN sales < 10000
 THEN 'Between 1,001 and 10,000'
WHEN sales < 100000
 THEN 'Between 10,001 and 100,000'
WHEN sales <= 1000000
 THEN 'Between 100,001 and 1,000,000'
ELSE 'Over 1,000,000'
END
AS "Sales category"
FROM titles
ORDER BY sales ASC;

```

**✓ 提示**

- 可以在SELECT、WHERE和ORDER BY子句中或任何允许使用表达式的位置，使用CASEO。
- 当返回结果后，CASE可能计算剩余WHEN子句中的表达式，也可能不计算，这依赖于DBMS。因为这个原因，要注意它令人厌烦的副作用，比如计算那些带来除数为零错误的表达式。
- 这个CASE表达式可以帮助防止除数为零的错误。

163

```

CASE
 WHEN n <> 0 THEN expr/n
 ELSE NULL
END

```

- 可以使用CASE以省略同样的函数调用。

```

WHERE some_function(col1) = 10
 OR some_function(col1) = 20

```

等价于

```

WHERE 1 =
CASE some_function(col1)
 WHEN 10 THEN 1
 WHEN 20 THEN 1
END

```

有些DBMS优化器运行CASE形式更快。

- 简单CASE表达式可以缩写为以下搜索CASE表达式：

```

CASE
 WHEN comparison_value = value1
 THEN result1
 WHEN comparison_value = value2
 THEN result2
 ...
 WHEN comparison_value = valueN
 THEN resultN
 [ELSE default_result]
END

```

| title_id | Sales                         | category |
|----------|-------------------------------|----------|
| T10      | Unknown                       |          |
| T01      | Not more than 1,000           |          |
| T08      | Between 1,001 and 10,000      |          |
| T09      | Between 1,001 and 10,000      |          |
| T02      | Between 1,001 and 10,000      |          |
| T13      | Between 10,001 and 100,000    |          |
| T06      | Between 10,001 and 100,000    |          |
| T04      | Between 10,001 and 100,000    |          |
| T03      | Between 10,001 and 100,000    |          |
| T11      | Between 10,001 and 100,000    |          |
| T12      | Between 100,001 and 1,000,000 |          |
| T05      | Between 100,001 and 1,000,000 |          |
| T07      | Over 1,000,000                |          |

图5-31 运行代码5-31的结果

- DBMS** Microsoft Access不支持CASE，改为使用函数Switch(*condition1, result1, condition2, result2, ...*)。要运行代码5-30和代码5-31，更改CASE表达式为（代码5-30）

```
Switch(
→ type IS NULL, NULL,
→ type = 'history', price * 1.10,
→ type = 'psychology', price *1.20,
→ type IN ('biography',
→ 'children', 'computer'), price)
```

对于代码5-31：

```
Switch(
→ sales IS NULL,
→ 'Unknown',
→ sales <= 1000,
→ 'Not more than 1,000',
→ sales <= 10000,
→ 'Between 1,001 and 10,000',
→ sales <= 100000,
→ 'Between 10,001 and 100,000',
→ sales <= 1000000,
→ 'Between 100,001 and 1,000,000',
→ sales > 1000000,
→ 'Over 1,000,000')
```

5

Oracle 9i和后续的版本可运行代码5-30和代码5-31。要在Oralce 8i中运行代码5-30，将简单CASE转换为搜索CASE表达式，或者使用函数DECODE(*comparison\_value, value1, result1, value2, result2, ..., default\_result*):

```
DECODE(type,
→ NULL, NULL,
→ 'history', price * 1.10,
→ 'psychology', price * 1.20,
→ price)
```

在PostgreSQL较早的版本中，要将代码5-30中将浮点数转换为DECIMAL，参见5.13节。要运行代码5-30，更改在CASE表达式中的新价格计算为

```
price * CAST((1.10) AS DECIMAL)
price * CAST((1.20) AS DECIMAL)
```

164

## 5.15 使用 COALESCE()检查空值

函数COALESCE()返回它的参数中第一个非空表达式。COALESCE()经常被用于在结果中用特定值替代空值的显示，这对用户发现空值困扰是有益的。COALESCE()是搜索CASE表达式的一般形式的缩写。

COALESCE(*expr1, expr2, expr3*)

等价于

```
CASE
 WHEN expr1 IS NOT NULL THEN expr1
 WHEN expr2 IS NOT NULL THEN expr2
 ELSE expr3
END
```

## ⇒ 返回第一个非空值

输入:

`COALESCE(expr1, expr2,...)`

`expr1, expr2, ...` 表示一个或更多由逗号分隔的表达式。所有表达式必须是一样的类型或隐式地转换为一样的类型。每一个表达式按顺序计算（从左到右），直到计算出一个非空值并返回。如果所有表达式为空，`COALESCE()`将返回空值（代码5-32和图5-32）。

**代码5-32** 列出出版社的位置。如果州为空值，显示  
N/A。结果见图5-32

```
SELECT
 pub_id,
 city,
 COALESCE(state, 'N/A') AS "state",
 country
FROM publishers;
```

## ✓ 提示

- 可以在SELECT、WHERE和ORDER BY子句中或任何允许使用表达式的位置，使用`COALESCE()`。
- 可以从一个不允许空值的列获得空值，参见第3章中的图3-3。
- **DBMS** Microsoft Access不支持`COALESCE()`，改为使用函数`Switch()`。要运行代码5-32，更改`COALESCE()`表达式为

```
Switch(state IS NOT NULL, state,
→ state IS NULL, 'N/A')
```

Oracle 9i和后续版本可以运行代码5-32。Oracle 8i不支持`COALESCE()`，改为使用函数`NVL(expr1, expr2)`。`NVL()`只接受两个表达式，对于3个或多个表达式要使用CASE。要在Oracle 8i中运行代码5-32，更改`COALESCE()`表达式为

```
NVL(state, 'N/A')
```

| pub_id | city          | state | country |
|--------|---------------|-------|---------|
| P01    | New York      | NY    | USA     |
| P02    | San Francisco | CA    | USA     |
| P03    | Hamburg       | N/A   | Germany |
| P04    | Berkeley      | CA    | USA     |

图5-32 运行代码5-32的结果

165

## 5.16 使用NULLIF()比较表达式

函数`NULLIF()`比较两个表达式，如果它们相等则返回空值，如果不相等则返回第一个表达式。`NULLIF()`通常被用于转换用户定义的缺失、未知或者不适用的空值。

有些人喜欢用数字-1或-99，或者串'N/A'、'Unknown'或'Missing'表示缺失值，胜过使用空值。DBMS对于包含空值的运算有清晰的规则，因此对于将用户定义的缺失值转换为空值，有时是需要的。例如，如果想去计算一列值的平均数，如将-1和真实的、未缺失的值混合在一起将得到错误的答案。可以改为使用`NULLIF()`将值-1转换为在计算过程中DBMS将忽略的空值。

`NULLIF()`是一个搜索CASE表达式的一般形式的缩写。

`NULLIF(expr1, expr2)`

等价于

```
CASE
 WHEN expr1 = expr2 THEN NULL
 ELSE expr1
END
```

### 避免除数为零

假设要计算不同学校的男女比率，当试图去计算没有女生的Lord of the Rings俱乐部的ratio时，发现下面的查询会失败并且产生除数为零的错误。

```
SELECT club_id, males, females,
 males/females AS ratio
 FROM school_clubs;
```

可以使用NULLIF去避免除数为零。将查询重写为：

```
SELECT club_id, males, females,
 males/NULIF(females,0)
 AS ratio
 FROM school_clubs;
```

任何数用NULL除都得NULL，不会有错误产生。

166

5

### » 如果两个表达式相等返回空值

输入：

`NULIF(expr1, expr2)`

*expr1*和*expr2*是表达式。NULLIF()比较*expr1*和*expr2*。如果它们相等，则函数返回空值；如果它们不相等，函数返回*expr1*。不可将*expr1*定义为字面量NULL（代码5-33和图5-33）。

**代码5-33** 在titles表中，如果没有合同，contract列包含零。该查询将零值转变为空值，非零值不受影响。结果见图5-33

```
SELECT
 title_id,
 contract,
 NULIF(contract, 0) AS "Null contract"
 FROM titles;
```

#### ✓ 提示

- 可以在SELECT、WHERE和ORDER BY子句中或任何允许使用表达式的位置，使用NULLIF()。
- DBMS** Microsoft Access不支持NULLIF()，改用表达式IIf(*expr1*=*expr2*, NULL, *expr1*)。要运行代码5-33，更改NULLIF()表达式为

`IIf(contract)=0,NULL,contract)`

在Oracle 9i和后续版本中可以运行代码5-33。

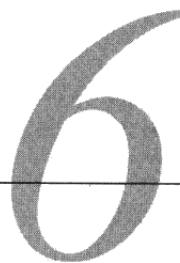
Oracle 8i不支持NULLIF()，要改为使用CASE。要在Oracle 8i中运行代码5-33，更改NULLIF()表达式为

```
CASE
 WHEN contract = 0 THEN NULL
 ELSE contract
END
```

| title_id | contract | Null contract |
|----------|----------|---------------|
| T01      | 1        | 1             |
| T02      | 1        | 1             |
| T03      | 1        | 1             |
| T04      | 1        | 1             |
| T05      | 1        | 1             |
| T06      | 1        | 1             |
| T07      | 1        | 1             |
| T08      | 1        | 1             |
| T09      | 1        | 1             |
| T10      | 0        | NULL          |
| T11      | 1        | 1             |
| T12      | 1        | 1             |
| T13      | 1        | 1             |

图5-33 运行代码5-33的结果

167



**前**面的内容介绍了只对单个行值进行操作的标量函数。本章将介绍对一组值进行操作以产生一个汇总值的SQL的聚合函数或集合函数。可以对行的集合应用聚合，这些行可以是：

- 表中所有的行；
- 那些由**WHERE**子句指定的行；
- 那些由**GROUP BY**子句创建的行。

对行分组的**GROUP BY**子句，经常和筛选组的**HAVING**子句一起使用。无论输入集包含多少行，聚合函数只返回单一的统计值，如总和、最小值或平均值。

169

有聚合函数和没有聚合函数的查询主要差异是，没有聚合的查询一行接一行地处理，每一行被独立地处理并放进结果中（**ORDER BY**和**DISTINCT**使DBMS查看所有的行，但它们本质上是后加工操作）。聚合查询做法完全不同，它们将表作为整体，并从中构造新行。

## 6.1 使用聚合函数

表6-1所示为SQL标准的聚合函数。

聚合函数的主要特点如下。

- 在表6-1中，表达式*expr*通常是一个列名，但它也可以是字面量、函数，或者连接或嵌套的列名、字面量和函数的任意组合。

表6-1 聚合函数

| 函 数              | 返 回                | 函 数                | 返 回                        |
|------------------|--------------------|--------------------|----------------------------|
| <b>MIN(expr)</b> | <i>expr</i> 中的最小值  | <b>AVG(expr)</b>   | <i>expr</i> 中的值的平均值（算术平均数） |
| <b>MAX(expr)</b> | <i>expr</i> 中的最大值  | <b>COUNT(expr)</b> | <i>expr</i> 中非空值的个数        |
| <b>SUM(expr)</b> | <i>expr</i> 中的值的总和 | <b>COUNT(*)</b>    | 表或集合的行数                    |

- **SUM()**和**AVG()**只对数字数据类型起作用。**MIN()**和**MAX()**对字符、数字、日期和时间数据类型起作用。**COUNT(expr)**和**COUNT(\*)**对所有数据类型都起作用。
- 除了**COUNT(\*)**以外，所有的聚合函数都将忽略空值（可以在聚合函数的参数中使用**COALESCE()**为空值替换一个值，见5.15节）。
- **COUNT(expr)**和**COUNT(\*)**绝对不会返回空值，而是返回一个正整数或零。如果集合中没有行或者只有包含空值的行，其他的聚合函数将返回空值。
- 对于聚合表达式的默认列标题因DBMS而异，可使用**AS**命名结果列，参见4.2节。

✓ 提示

- **DBMS** 提供附加的聚合函数计算其他的统计值，如标准偏差（standard deviation），参阅DBMS 文档中的aggregate functions或group functions。

## 6.2 创建聚合表达式

聚合函数可以灵活使用。本节将讲解什么是合法的聚合表达式以及什么不是。

- 聚合表达式不能出现在WHERE子句里。如果想查找有最高销售量的图书，不能使用：

```
SELECT title_id --Illegal
 FROM titles
 WHERE sales = MAX(sales);
```

- 不能在SELECT子句中混合使用非聚合表达式（一行接一行）和聚合表达式。SELECT子句必须全部包含非聚合表达式，或者全部包含聚合表达式。如果想查找有最高销售量的图书，不能使用：

```
SELECT title_id, MAX(sales)
 FROM titles; --Illegal
```

这个规则的一个例外情况是，对于分组列可以混合使用非聚合表达式和聚合表达式（参见6.9节）。

```
SELECT type, SUM(sales)
 FROM titles
 GROUP BY type; --Legal
```

- 可以在SELECT子句中使用多个聚合表达式：

```
SELECT MIN(sales), MAX(sales)
 FROM titles; --Legal
```

- 不可以嵌套聚合函数：

```
SELECT SUM(AVG(sales))
 FROM titles; --Illegal
```

- 可以在子查询中使用聚合表达式。下面这条语句查找有最高销售量的图书。

```
SELECT title_id, price --Legal
 FROM titles
 WHERE sales =
 (SELECT MAX(sales) FROM titles);
```

- 不可以在聚合表达式中使用子查询（见第8章），AVG(SELECT price FROM titles)是非法的。

✓ 提示

- **DBMS** Oracle可以在GROUP BY查询中嵌套聚合表达式。下面的例子计算所有类型图书销售量最大值的平均值。Oracle对于分组列type，先计算内部的聚合函数MAX(sales)，然后在聚合结果。

```
SELECT AVG(MAX(sales))
 FROM titles
 GROUP BY type; --Legal in Oracle
```

要在标准的SQL中运行这个查询，在FROM子句中使用子查询（见第8章）。

```
SELECT AVG(s.max_sales)
 FROM (SELECT MAX(sales) AS max_sales
```

171

```
FROM titles
GROUP BY type) s;
```

### 6.3 使用MIN()查找最小值

可以使用聚合函数MIN()查找值集中的最小值。

#### ⇒ 查找值集中的最小值

输入:

`MIN(expr)`

*expr*是列名、字面量或表达式。结果的数据类型和*expr*相同。

代码6-1和图6-1显示了包含MIN()的一些查询。第一个查询返回价格最低的图书的价格。第二个查询返回最早印刷的日期。第三个查询返回页数最少的历史书的页数。

代码6-1 某些MIN()查询。结果见图6-1

```
SELECT MIN(price) AS "Min price"
 FROM titles;

SELECT MIN(pubdate) AS "Earliest pubdate"
 FROM titles;

SELECT MIN(pages) AS "Min history pages"
 FROM titles
 WHERE type = 'history';
```

#### ✓ 提示

- MIN()适用于字符、数字、日期数据类型。
- 对于字符数据列，MIN()查找排序序列中的最低值，参见4.4节。
- DISTINCT对于MIN()是没有意义的，见6.8节。
- **DBMS** 串比较是否区分大小写依赖于DBMS，参见4.5节中的DBMS提示。

|                   |            |
|-------------------|------------|
| Min price         | -----      |
|                   | 6.95       |
| Earliest pubdate  | -----      |
|                   | 1998-04-01 |
| Min history pages | -----      |
|                   | 14         |

图6-1 运行代码6-1的结果

172

### 6.4 使用MAX()查找最大值

可以使用聚合函数MAX()查找值集中的最大值。

#### ⇒ 查找值集中的最大值

输入:

`MAX(expr)`

*expr*是列名、字面量或表达式。结果的数据类型和*expr*相同。

代码6-2和图6-2显示了包含MAX()的某些查询。第一个查询返回按字母顺序排在最后一位的作者的

姓。第二个查询返回最便宜的和最昂贵的书的价格，以及它们的价格差。第三个查询返回历史书中最高的收入（即价格×销售量）。

#### 代码6-2 一些MAX()查询。结果见图6-2

```
SELECT MAX(au_lname) AS "Max last name"
 FROM authors;

SELECT
 MIN(price) AS "Min price",
 MAX(price) AS "Max price",
 MAX(price) - MIN(price) AS "Range"
 FROM titles;

SELECT MAX(price * sales)
 AS "Max history revenue"
 FROM titles
 WHERE type = 'history';
```

| Max last name       |             |       |
|---------------------|-------------|-------|
| <hr/>               |             |       |
|                     | O'Furniture |       |
| Min price           | Max price   | Range |
| <hr/>               | <hr/>       | <hr/> |
| 6.95                | 39.95       | 33.00 |
| Max history revenue |             |       |
| <hr/>               |             |       |
|                     | 313905.33   |       |

图6-2 运行代码6-2的结果

#### ✓ 提示

- MAX()适用于字符、数字、日期数据类型。
- 对于字符数据列，MAX()查找排序序列中的最高值，参见4.4节。
- DISTINCT对于MAX()是没有意义的，参见6.8节。
- **DBMS** 字符串比较是否区分大小写依赖于DBMS，参见4.5节中的DBMS提示。当比较两个 VARCHAR串是否相等时，DBMS也许会给短串的右边加上空格后一位一位地比较串。在这种情况下，串'Jack'和'Jack'是相等的。参考DBMS文档（或者试验）以确定MAX()返回哪个串。

6

173

## 6.5 使用 SUM()计算总和

可以使用聚合函数SUM()查找值集的总和。

#### ⇒ 计算值集的总和

输入：

SUM(expr)

expr是列名、字面量或数字表达式。结果的数据类型至少和在expr中精度最高的精度一样。

代码6-3和图6-3显示了包含SUM()的一些查询。第一个查询返回给作者的全部预付款总和。第二个查询返回在2000年出版的书的总销售量。第三个查询返回所有书的价格、销售量和收入总和。在这里，注意一个数学常识：乘积之和不（必然）等于和的乘积。

#### 代码6-3 一些SUM()查询。结果见图6-3

```
SELECT SUM(advance) AS "Total advances"
 FROM royalties;

SELECT SUM(sales)
 AS "Total sales (2000 books)"
 FROM titles
```

```

WHERE pubdate
 BETWEEN DATE '2000-01-01'
 AND DATE '2000-12-31';

SELECT
 SUM(price) AS "Total price",
 SUM(sales) AS "Total sales",
 SUM(price * sales) AS "Total revenue"
FROM titles;

```

✓ 提示

- **SUM()** 只适用于数字数据类型。
- 如果一行没有，则总和是空值，而不是可能设想的零。
- **DBMS** 在 Microsoft Access 的日期字面量中，省略 DATE 关键字并用#字符替代引号括住“字面量”。要运行代码6-3，在第二个查询里更改日期字面量为#2000-01-01#和#2000-12-31#。在 Microsoft SQL Server 和 DB2 的日期字面量中，省略 DATE 关键字。要运行代码6-3，更改日期字面量为'2000-01-01'和'2000-12-31'。

174

## 6.6 使用 AVG() 计算平均值

可以使用聚合函数 AVG() 查找值集的平均值，即算术平均值。算术平均值是集合中量的总和，除以集合中量的个数。

⇒ 计算值集的平均值

输入：

**AVG(expr)**

*expr* 是列名、字面量或数字表达式。结果的数据类型至少和 *expr* 中精度最高的精度一样。

代码6-4和图6-4显示了包含 AVG() 的一些查询。第一个查询返回如果价格翻倍后图书的平均价格。第二个查询返回商业图书销售量的平均值和销售量的总和，因为表中不含有商业图书，这两个计算结果为空值（不是零）。第三个查询使用子查询（见第8章）列出销售量超过平均值的图书。

175

### 代码6-4 某些AVG()查询。结果见图6-4

```

SELECT AVG(price * 2) AS "AVG(price * 2)"
 FROM titles;

SELECT AVG(sales) AS "AVG(sales)",
 SUM(sales) AS "SUM(sales)"
 FROM titles
 WHERE type = 'business';

SELECT title_id, sales

```

|                                       |  |  |  |
|---------------------------------------|--|--|--|
| Total advances                        |  |  |  |
| -----                                 |  |  |  |
| 1336000.00                            |  |  |  |
| Total sales (2000 books)              |  |  |  |
| -----                                 |  |  |  |
| 231677                                |  |  |  |
| Total price Total sales Total revenue |  |  |  |
| -----                                 |  |  |  |
| 220.65 1975446 41428860.77            |  |  |  |

图6-3 运行代码6-3的结果

|                       |  |  |
|-----------------------|--|--|
| AVG(price * 2)        |  |  |
| -----                 |  |  |
| 36.775000             |  |  |
| AVG(sales) SUM(sales) |  |  |
| -----                 |  |  |
| NULL NULL             |  |  |
| title_id sales        |  |  |
| -----                 |  |  |
| T07 1500200           |  |  |
| T05 201440            |  |  |

图6-4 运行代码6-4的结果

```
FROM titles
WHERE sales >
 (SELECT AVG(sales) FROM titles)
ORDER BY sales DESC;
```

✓ 提示

- AVG()只适用于数字数据类型。
- 如果一行没有，则平均值是空值，而不是设想的零。
- 如果使用0或-1而不是空值表示缺失值，那么在AVG()计算中包含这样的数字会产生不正确的结果。使用NULLIF()将缺失值转换为空值，则它们将排斥在计算之外，参见5.16节。
- **DBMS MySQL**和之前的版本缺少对子查询的支持，且不能运行代码6-4中的第三个查询。

176

### 聚合和空值

聚合函数（除COUNT(\*)以外）忽略空值。如果聚合需要计算空值，可以通过使用COALESCE()以特定值替代每一个空值（见5.15节）。例如，下面的查询返回传记的平均销售量（在计算中包括被零替代的空值）。

```
SELECT AVG(COALESCE(sales,0))
 AS AvgSales
 FROM titles
 WHERE type = 'biography';
```

6

### SQL中的统计

SQL不是统计学编程语言，但可以使用内置函数和一些技巧去计算简单的描述统计量，如总和、平均值和标准偏差。对于非常复杂的分析，可以使用DBMS的OLAP（online analytical processing）组件或者将数据输出到专门的统计环境中（如Excel、R、SAS或者SPSS）。

不应该在SQL或宿主语言中自己编写统计程序。正确执行统计运算，即使是最简单的，也要权衡效率（算术运算所需的空间）、稳定性（取消极端数字）和准确性（处理病态值的集合）。可参见Ronald Thisted的*Elements of Statistical Computing*（Chapman & Hall/CRC）和John Monahan的*Numerical Methods of Statistics*（Cambridge University Press）。

可以偶尔使用少量的内置SQL函数组合，如用于标准误差的STDEV() / SQRT(COUNT())，但是对于关联、回归、ANOVA（不一致分析）或者矩阵算术，不能使用复杂的SQL表达式。查阅DBMS的SQL和OLAP文档，了解它提供了哪些函数。内置函数是不可移植的，但它们要比等价的查询表达式运行得更快、更正确。

函数MIN()和MAX()计算顺序统计值，顺序统计值是从按大小排序的数据集中派生出的值。著名的次序统计值包括修整平均值、秩、极差、众数和中值。第15章介绍了修整平均值、排名和中值。极差是最大值和最小值的差，如MAX(expr)-MIN(expr)。众数是最经常出现的值。一个数据集可以有多于一个的众数。因为众数不是稳定可靠的，所以它是弱描述统计值，这意味着对数据集增加一个小的数字，或者异常，或者不正确的值，就可以影响它。下面的查询查找示例数据库中图书价格的众数。

```
SELECT price, COUNT(*) AS frequency
 FROM titles
 GROUP BY price
 HAVING COUNT(*) >= ALL(SELECT COUNT(*) FROM titles GROUP BY price);
```

price有两个众数:

| price | frequency |
|-------|-----------|
| 12.99 | 2         |
| 19.95 | 2         |

177

## 6.7 使用 COUNT()统计行数

可以使用聚合函数COUNT()统计值集中行的个数。

COUNT()有两种形式。

- COUNT(*expr*)返回*expr*不为空的行数。
- COUNT(\*)返回集中全部行的个数，包括空值和重复值。

⇒ 统计非空的行数

输入:

COUNT(*expr*)

*expr*是列名、字面量或表达式。结果是大于或等于零的整数。

⇒ 统计包含空值的全部行的个数

输入:

COUNT(\*)

结果是大于或等于零的整数。

代码6-5和图6-5显示了包含COUNT(*expr*)和COUNT(\*)的一些查询。这3个查询统计了表titles的行数，它们除WHERE子句以外是相同的。在第一个查询中行数统计不同是因为列price包含了空值。在第二个查询中行数统计相同是因为在统计前，WHERE子句消除了含有空值价格的行。第三个查询显示了前两个查询结果行数统计的差。

代码6-5 某些COUNT()查询。结果见图6-5

```
SELECT
 COUNT(title_id) AS "COUNT(title_id)",
 COUNT(price) AS "COUNT(price)",
 COUNT(*) AS "COUNT(*)"
 FROM titles;
```

```
SELECT
 COUNT(title_id) AS "COUNT(title_id)",
 COUNT(price) AS "COUNT(price)",
```

| COUNT(title_id) | COUNT(price) | COUNT(*) |
|-----------------|--------------|----------|
| 13              | 12           | 13       |

| COUNT(title_id) | COUNT(price) | COUNT(*) |
|-----------------|--------------|----------|
| 12              | 12           | 12       |

| COUNT(title_id) | COUNT(price) | COUNT(*) |
|-----------------|--------------|----------|
| 1               | 0            | 1        |

图6-5 运行代码6-5的结果

```

COUNT(*) AS "COUNT(*)"
FROM titles
WHERE price IS NOT NULL;

SELECT
 COUNT(title_id) AS "COUNT(title_id)",
 COUNT(price) AS "COUNT(price)",
 COUNT(*) AS "COUNT(*)"
FROM titles
WHERE price IS NULL;

```

✓ 提示

- COUNT(*expr*)和COUNT(\*)适用于所有数据类型，并且绝对不会返回空值。
- 对于COUNT(\*)来说，DISTINCT是没有意义的，参见6.8节。
- COUNT(\*) – COUNT(*expr*)返回空值的数量， $((COUNT(*) - COUNT(expr)) * 100) / COUNT(*)$ 返回空值的百分比。

178

6

## 6.8 使用 DISTINCT 聚合不重复的值

可以使用 DISTINCT 消除聚合函数计算中重复的值，参见4.3节。聚合函数的一般语法如下：

*agg\_func*[*ALL* | *DISTINCT*] *expr*

*agg\_func*是MIN、MAX、SUM、AVG或COUNT。*expr*是列名、字面量或表达式。*ALL*对全部值应用聚合函数，*DISTINCT*指定考虑每一个唯一值。*All*是默认的，在实际中很少见到。

对于SUM()、AVG()和COUNT(*expr*)，*DISTINCT*在计算总和、平均值或统计行数之前消除重复值。对于MIN()和MAX()，*DISTINCT*没有意义，可以使用它，但它不会改变结果。不能对COUNT(\*)使用DISTINCT。

⇒ 计算表中不重复的值的总和

输入：

SUM(DISTINCT *expr*)

*expr*是列名、字面量或数字表达式。结果的数据类型至少和在*expr*中精度最高的数据精度相同。

⇒ 计算表中不重复的值的平均值

输入：

AVG(DISTINCT *expr*)

*expr*是列名、字面量或数学表达式。结果的数据类型至少和在*expr*中精度最高的数据精度相同。

⇒ 统计不重复的非空行的个数

输入：

COUNT(DISTINCT *expr*)

*expr*是列名、字面量或表达式。结果是一个大于或等于零的整数。

179

代码6-6中的查询返回书价的行数统计、总和及平均值。因为DISTINCT结果在计算中消除了重复的价格12.99美元和19.95美元，所以在图6-6中有DISTINCT和没有DISTINCT的价格结果是不同的。

**代码6-6 一些有DISTINCT的聚合查询。结果见图6-6**

```

SELECT
 COUNT(*) AS "COUNT(*)"
FROM titles;

SELECT
 COUNT(price) AS "COUNT(price)",
 SUM(price) AS "SUM(price)",
 AVG(price) AS "AVG(price)"
FROM titles;

SELECT
 COUNT(DISTINCT price)
 AS "COUNT(DISTINCT)",
 SUM(DISTINCT price)
 AS "SUM(DISTINCT)",
 AVG(DISTINCT price)
 AS "AVG(DISTINCT)"
FROM titles;

```

| COUNT(*)        |               |               |
|-----------------|---------------|---------------|
| 13              |               |               |
| COUNT(price)    | SUM(price)    | AVG(price)    |
| 12              | 220.65        | 18.3875       |
| COUNT(DISTINCT) | SUM(DISTINCT) | AVG(DISTINCT) |
| 10              | 187.71        | 18.7710       |

图6-6 运行代码6-6的结果

**✓ 提示**

- COUNT(DISTINCT)/COUNT()的比率表示值的集合重复的程度。比率为1或接近1，意味着集合中包含非常多的唯一值。比率越接近0，意味着重复行多。
- SELECT子句中的DISTINCT和聚合函数中的DISTINCT不会返回相同的结果。

在代码6-7中的3个查询统计了表title\_authors中作者ID。图6-7显示了结果。第一个查询统计了表中所有作者ID的行数。因为在DISTINCT被应用之前COUNT()已经完成了它的任务，并将返回的值放在单一行中，所以第二个查询返回了和第一个查询相同的结果。在第三个查询中，在COUNT()开始计算之前DISTINCT已被应用于作者ID。

180

**代码6-7 SELECT子句中的DISTINCT与聚合函数中的DISTINCT意义不同。结果见图6-7**

```

SELECT COUNT(au_id)
 AS "COUNT(au_id)"
FROM title_authors;

SELECT DISTINCT COUNT(au_id)
 AS "DISTINCT COUNT(au_id)"
FROM title_authors;

SELECT COUNT(DISTINCT au_id)
 AS "COUNT(DISTINCT au_id)"
FROM title_authors;

```

| COUNT(au_id)          |
|-----------------------|
| 17                    |
| DISTINCT COUNT(au_id) |
| 17                    |
| COUNT(DISTINCT au_id) |
| 6                     |

图6-7 运行代码6-7的结果

- 在同一个SELECT子句中混合使用没有DISTINCT，和有DISTINCT的聚合函数，会产生令人误解的结果。

代码6-8中的4个查询表示了4个没有DISTINCT的和有DISTINCT的总和与行数统计的组合。在如图6-8所示的结果中，只有第一个结果（没有DISTINCT）和最后一个结果（全部有DISTINCT）是算术一致的，可以使用AVG(price)和AVG(DISTINCT price)进行验证。在第二个和第三个查询（混合使用没有DISTINCT和有DISTINCT）中，不能使用总和除以统计的行数来计算有效的平均值。

**代码6-8** 混合使用没有DISTINCT和有DISTINCT的聚合函数得出不一致的结果。结果见图6-8

```

SELECT
 COUNT(price),
 AS "COUNT(price)",
 SUM(price)
 AS "SUM(price)"
FROM titles;

SELECT
 COUNT(price)
 AS "COUNT(price)",
 SUM(DISTINCT price)
 AS "SUM(DISTINCT price)"
FROM titles;

SELECT
 COUNT(DISTINCT price)
 AS "COUNT(DISTINCT price)",
 SUM(price)
 AS "SUM(price)"
FROM titles;

SELECT
 COUNT(DISTINCT price)
 AS "COUNT(DISTINCT price)",
 SUM(DISTINCT price)
 AS "SUM(DISTINCT price)"
FROM titles;

```

□ **DBMS** Microsoft Access不支持DISTINCT聚合函数。例如，下面这条语句在Access中是不合法的：

```
SELECT SUM(DISTINCT price)
FROM titles; --Illegal in Access
```

但是可以使用子查询改写它（见8.7节）：

```
SELECT SUM(price)
FROM (SELECT DISTINCT price
 FROM titles);
```

Access工作区不允许像代码6-8中第二个和第三个查询那样，混合使用没有DISTINCT和有DISTINCT的聚合函数。

MySQL 4.1和之前的版本支持COUNT(DISTINCT *expr*)，但不支持SUM(DISTINCT *expr*)和AVG(DISTINCT *expr*)，因此无法运行代码6-6和代码6-8。MySQL 5.0和后续的版本支持所有的DISTINCT聚合函数。

181  
182

| COUNT(price) | SUM(price) |
|--------------|------------|
| 12           | 220.65     |

| COUNT(price) | SUM(DISTINCT price) |
|--------------|---------------------|
| 12           | 187.71              |

| COUNT(DISTINCT price) | SUM(price) |
|-----------------------|------------|
| 10                    | 220.65     |

| COUNT(DISTINCT price) | SUM(DISTINCT price) |
|-----------------------|---------------------|
| 10                    | 187.71              |

图6-8 运行代码6-8的结果。在计数和求和上的差异指出了价格重复。从第二个(187.71/12)或第三个(220.65/10)查询中获得的平均值(sum/count)是不正确的。第一个(220.65/12)和第四个(187.71/10)查询产生的平均值一致

## 6.9 使用 GROUP BY 分组行

至此，可以使用聚合函数汇总一列中的所有值，或者仅匹配WHERE查询条件的值；还可以使用GROUP BY子句将一个表分隔成逻辑组（类别），并对每一组计算聚合统计量。

用一个例子来阐明概念。代码6-9使用GROUP BY统计每一位作者所写（或合写）的书的数量。在

SELECT子句中，列`au_id`标识每一位作者，派生列`num_books`统计每一位作者写的书的数量。GROUP BY子句导致对于每一个不同的`au_id`都计算`num_books`，而不是只对整个表计算一次。图6-9显示了结果。在这个例子中，`au_id`被称为分组列。

#### 代码6-9 列出每一位作者所写（或合写）的书的数量。

结果见图6-9

```
SELECT
 au_id,
 COUNT(*) AS "num_books"
FROM title_authors
GROUP BY au_id;
```

GROUP BY子句的主要特点如下。

GROUP BY子句位于WHERE子句之后、ORDER BY子句之前。

分组列可以是列名或派生列。

输入表中的列无法出现在聚合查询的SELECT子句中，除非它们也被包含在GROUP BY子句中。列在不同的行可能有不同的值，因此如果从整个表中产生一个单一的新行，将无法决定这些值中的哪个会被包含在结果中。下面的语句是不合法的，因为GROUP BY子句对于每一个type值仅返回一行；查询无法返回多个和一个type值关联的pub\_id值。

```
SELECT type, pub_id, COUNT(*)
FROM titles
GROUP BY type; --Illegal
```

183

如果SELECT子句包含复杂的非聚合表达式（而不只是一个简单的列名），GROUP BY表达式必须正确地匹配SELECT表达式。

可以在GROUP BY子句中指定多个分组列来嵌套分组。数据按最后指定的分组汇总。

如果分组列包含一个空值，在结果中这行会变为一个分组。如果分组列包含多个空值，空值将被放进一个分组中。分组包含多个空值并不意味着空值彼此相等。

可以在包含GROUP BY子句的查询中使用WHERE子句，在分组前消除行。

尽管表的别名被允许作为标识符，但不能在GROUP BY子句中使用列的别名，参见7.2节。

不使用ORDER BY子句，而使用GROUP BY子句返回没有特定顺序的分组。例如，要对代码6-9的结果按图书的数量降序排序，添加子句ORDER BY "num\_books" DESC。

#### ⇒ 分组行

输入：

```
SELECT columns
FROM table
[WHERE search_condition]
GROUP BY grouping_columns
[HAVING search_condition]
[ORDER BY sort_columns];
```

`columns`和`grouping_columns`是一个或多个以逗号分隔的列名，`table`是包含`columns`和`grouping_columns`的表的名称。出现在`columns`中的非聚合列也必须出现在`grouping_columns`中。列名在`grouping_`

| au_id | num_books |
|-------|-----------|
| A01   | 3         |
| A02   | 4         |
| A03   | 2         |
| A04   | 4         |
| A05   | 1         |
| A06   | 3         |

图6-9 运行代码6-9的结果

*columns*中的顺序决定了从最高到最低的分组级别。GROUP BY子句限制了结果中的行，在分组列中对于每一个不重复的值只显示一行。结果中的每一行都包含了与在分组列中的特定值关联的汇总数据。

如果语句包含WHERE子句，DBMS是在对

代码6-10和图6-10显示了在包含GROUP BY子句的查询中COUNT(*expr*)和COUNT(\*)之间的差异。表publishers在列state中包含了一个空值（德国的出版社P03）。回顾6.7节，COUNT(*expr*)统计非空值，而COUNT(\*)统计包括空值在内的所有值。在结果中，GROUP BY认可空值，并且为其创建一个空值分组。COUNT(\*)发现并统计列state中的一个空值，但是COUNT(state)对于空值分组为零，因为COUNT(state)在被排斥于统计之外的空值分组中发现一个空值，这就是为什么是零的原因。

代码6-10 这个查询说明了在GROUP BY查询中COUNT(*expr*)和COUNT(\*)之间的差异。结果见图6-10

```
SELECT
 state,
 COUNT(state) AS "COUNT(state)",
 COUNT(*) AS "COUNT(*)"
FROM publishers
GROUP BY state;
```

| state | COUNT(state) | COUNT(*) |
|-------|--------------|----------|
| NULL  | 0            | 1        |
| CA    | 2            | 2        |
| NY    | 1            | 1        |

图6-10 运行代码6-10的结果

如果非聚合列包含空值，使用COUNT(\*)而不是COUNT(*expr*)可能产生令人误解的结果。代码6-11和图6-11显示了对图书的每一种类型汇总销售量统计值。一本传记书的销售量值是空值，因此COUNT(sales)和COUNT(\*)差1。在第五列的平均值计算SUM/COUNT(sales)是算术一致的。然而，第六列的平均值SUM/COUNT(\*)就不是了。最后一列中的AVG(sales)证明了这个不一致（回顾6.8节代码6-8中的类似情形）。

代码6-11 如果*expr*包含空值，使用COUNT(*expr*)而不是COUNT(\*)得出算术一致的结果。结果见图6-11

```
SELECT
 type,
 SUM(sales) AS "SUM(sales)",
 COUNT(sales) AS "COUNT(sales)",
 COUNT(*) AS "COUNT(*)",
 SUM(sales)/COUNT(sales)
 AS "SUM/COUNT(sales)",
 SUM(sales)/COUNT(*)
 AS "SUM/COUNT(*)",
 AVG(sales) AS "AVG(sales)"
FROM titles
GROUP BY type;
```

| type       | SUM(sales) | COUNT(sales) | COUNT(*) | SUM/COUNT(sales) | SUM/COUNT(*) | AVG(sales) |
|------------|------------|--------------|----------|------------------|--------------|------------|
| biography  | 1611521    | 3            | 4        | 537173.67        | 402880.25    | 537173.67  |
| children   | 9095       | 2            | 2        | 4547.50          | 4547.50      | 4547.50    |
| computer   | 25667      | 1            | 1        | 25667.00         | 25667.00     | 25667.00   |
| history    | 20599      | 3            | 3        | 6866.33          | 6866.33      | 6866.33    |
| psychology | 308564     | 3            | 3        | 102854.67        | 102854.67    | 102854.67  |

图6-11 运行代码6-11的结果

代码6-12和图6-12显示了一个计算销售量的总和、销售量的平均值和每一种类型图书数量的简单的GROUP BY查询。在代码6-13和图6-13中，添加WHERE子句在分组以前去掉价格低于13美元的图书，还添加了ORDER BY子句按照每一种类型书的销售量的总和对结果降序排序。

**代码6-12** 这个简单的GROUP BY子句对于每一种类型的图书计算了一些汇总统计值。结果见图6-12

```
SELECT
 type,
 SUM(sales) AS "SUM(sales)",
 AVG(sales) AS "AVG(sales)",
 COUNT(sales) AS "COUNT(sales)"
FROM titles
GROUP BY type
```

| TYPE       | SUM(sales) | AVG(sales) | COUNT(sales) |
|------------|------------|------------|--------------|
| biography  | 1611521    | 537173.67  | 3            |
| children   | 9095       | 4547.50    | 2            |
| computer   | 25667      | 25667.00   | 1            |
| history    | 20599      | 6866.33    | 3            |
| psychology | 308564     | 102854.67  | 3            |

图6-12 运行代码6-12的结果

**代码6-13** 这里，对于代码6-12添加了WHERE和ORDER BY子句，去掉了价格低于13美元的图书并按销售量的总和对结果降序排序。结果见图6-13

```
SELECT
 type,
 SUM(sales) AS "SUM(sales)",
 AVG(sales) AS "AVG(sales)",
 COUNT(sales) AS "COUNT(sales)"
FROM titles
WHERE price >= 13
GROUP BY type
ORDER BY "SUM(sales)" DESC;
```

| type      | SUM(sales) | AVG(sales) | COUNT(sales) |
|-----------|------------|------------|--------------|
| biography | 1511520    | 755760.00  | 2            |
| computer  | 25667      | 25667.00   | 1            |
| history   | 20599      | 6866.33    | 3            |
| children  | 5000       | 5000.00    | 1            |

图6-13 运行代码6-13的结果

代码6-14和图6-14使用了多个分组列统计每一个出版社出版的每一种类型图书的数量。

**代码6-14** 列出每一个出版社的每一种类型图书的数量，并在按出版社ID升序排序的基础上，在内部按图书数量降序排序。结果见图6-14

```
SELECT
 pub_id,
 type,
 COUNT(*) AS "COUNT(*)"
FROM titles
GROUP BY pub_id, type
ORDER BY pub_id ASC, "COUNT(*)" DESC;
```

| pub_id | type       | COUNT(*) |
|--------|------------|----------|
| P01    | biography  | 3        |
| P01    | history    | 1        |
| P02    | computer   | 1        |
| P03    | history    | 2        |
| P03    | biography  | 1        |
| P04    | psychology | 3        |
| P04    | children   | 2        |

图6-14 运行代码6-14的结果

在代码6-15和图6-15中，再次使用了5.14节中的代码5-31，但用GROUP BY列出每一个销售量范围中图书的数量，替代了按它的销售量范围分类列出每一本书。

**代码6-15** 列出每一种被计算出销售量范围的图书的数量，并按销售量升序排序。结果见图6-15

```
SELECT
```

```

CASE
 WHEN sales IS NULL
 THEN 'Unknown'
 WHEN sales <= 1000
 THEN 'Not more than 1,000'
 WHEN sales <= 10000
 THEN 'Between 1,001 and 10,000'
 WHEN sales <= 100000
 THEN 'Between 10,001 and 100,000'
 WHEN sales <= 1000000
 THEN 'Between 100,001 and 1,000,000'
 ELSE 'Over 1,000,000'
END
AS "Sales category",
COUNT(*) AS "Num titles"
FROM titles
GROUP BY
CASE
 WHEN sales IS NULL
 THEN 'Unknown'
 WHEN sales <= 1000
 THEN 'Not more than 1,000'
 WHEN sales <= 10000
 THEN 'Between 1,001 and 10,000'
 WHEN sales <= 100000
 THEN 'Between 10,001 and 100,000'
 WHEN sales <= 1000000
 THEN 'Between 100,001 and 1,000,000'
 ELSE 'Over 1,000,000'
END
ORDER BY MIN(sales) ASC;

```

### ✓ 提示

- 使用WHERE子句将不想分组的行排斥在外，并且使用HAVING子句在分组后筛选行，下一节将介绍HAVING。
- 如果没有使用聚合函数，GROUP BY的作用类似于DISTINCT（代码6-16和图6-16）。关于DISTINCT的知识，参见4.3节。

**代码6-16** 这两个查询返回相同的结果。下面的方法更好。结果见图6-16

```

SELECT type
FROM titles
GROUP BY type;

SELECT DISTINCT type
FROM titles;

```

- 可以使用GROUP BY子句查找数据中的模式。在代码6-17和图6-17中，查找价格分类和平均销售量之间的关系。

| Sales category                | Num titles |
|-------------------------------|------------|
| Unknown                       | 1          |
| Not more than 1,000           | 1          |
| Between 1,001 and 10,000      | 3          |
| Between 10,001 and 100,000    | 5          |
| Between 100,001 and 1,000,000 | 2          |
| Over 1,000,000                | 1          |

图6-15 运行代码6-15的结果

187

6

| type       |
|------------|
| biography  |
| children   |
| computer   |
| history    |
| psychology |

图6-16 在代码6-16中的两条语句都返回这个结果

**代码6-17** 对于每一价格列出平均销售量，并按价格升序排序。结果见图6-17

```
SELECT price, AVG(sales) AS "AVG(sales)"
 FROM titles
 WHERE price IS NOT NULL
 GROUP BY price
 ORDER BY price ASC;
```

- 不要依靠GROUP BY排序结果。无论何时使用GROUP BY都要包含ORDER BY（尽管在一些例子中省略了ORDER BY）。在某些DBMS中，GROUP BY意味着ORDER BY。
- 在GROUP BY查询中，一个聚合函数返回多个值被称为矢量聚合。在缺少GROUP BY子句的查询中，一个聚合函数返回单个值被称为标量聚合。
- 应该为经常分组的列创建索引（见第12章）。
- 可以使用函数FLOOR( $x$ )分类数字值。FLOOR( $x$ )返回小于 $x$ 的最大整数。这个查询按照10美元的价格间隔来分组图书。

```
SELECT
 FLOOR(price/10)*10 AS "Category",
 COUNT(*) AS "Count"
 FROM titles
 GROUP BY FLOOR(price/10)*10;
```

结果是：

| Category | Count |
|----------|-------|
| 0        | 2     |
| 10       | 6     |
| 20       | 3     |
| 30       | 1     |
| NULL     | 1     |

类别0统计了价格在0.00美元到9.99美元之间的数量，类别10统计了价格在10.00美元到19.99美元之间的数量，依次类推（类似的函数CEILING( $x$ )返回大于 $x$ 的最小整数）。

- 在Microsoft Access中，在代码6-15中要使用Switch()函数替代CASE表达式，参见5.14节中的DBMS提示。  
MySQL 4.1和先前的版本不允许在GROUP BY子句中使用CASE，因此代码6-15将不能运行。  
MySQL 5.0和之后的版本可以运行它。

## 6.10 使用 HAVING 筛选分组

HAVING子句设置GROUP BY子句的条件，类似于WHERE与SELECT结合的情形。HAVING子句的主要特点如下。

- HAVING子句位于GROUP BY子句之后、ORDER BY子句之前。

| price | AVG(sales) |
|-------|------------|
| 6.95  | 201440.0   |
| 7.99  | 94123.0    |
| 10.00 | 4095.0     |
| 12.99 | 56501.0    |
| 13.95 | 5000.0     |
| 19.95 | 10443.0    |
| 21.99 | 566.0      |
| 23.95 | 1500200.0  |
| 29.99 | 10467.0    |
| 39.95 | 25667.0    |

图6-17 运行代码6-17的结果。例如忽略在23.95美元上的统计离群值，price和sales之间表现为弱的反比关系

- 正如WHERE子句限制了SELECT显示的行数，HAVING限制了GROUP BY显示的分组数。
  - WHERE查询条件在分组产生以前就被应用，而HAVING搜索条件在分组产生之后才被应用。
  - 除了HAVING可以包含聚合函数以外，HAVING语法类似于WHERE语法。
  - HAVING子句可以引用显示在SELECT列表中的任何一项。
- WHERE、GROUP BY和HAVING子句应用的顺序如下。
- (1) WHERE子句从FROM和JOIN子句指定的运算结果中筛选行。
  - (2) GROUP BY子句对WHERE子句的输出进行分组。
  - (3) HAVING子句对分组后的结果筛选行。

### ⇒ 筛选分组

在GROUP BY子句之后，输入：

`HAVING search_condition`

*search\_condition*用于筛选分组的搜索条件。*search\_condition*可以包含聚合函数，其他方面和4.5节介绍的WHERE搜索条件是相同的。可以对多个HAVING条件用AND、OR或NOT组合及求反。HAVING搜索条件应用于分组输出的行，只有满足搜索条件的行出现在结果中。只能对出现在GROUP BY子句或聚合函数中的列使用HAVING子句。

在代码6-18和图6-18中，再次使用了本章前面的代码6-9，但是使用HAVING仅列出写（或合写）过3本或3本以上书的作者所写书的数量，而不是列出每一位作者写（或合写）的书的数量。

**代码6-18** 列出写（或合写）过3本或3本以上书的每一位作者所写的书的数量。结果见图6-18

```
SELECT
 au_id,
 COUNT(*) AS "num_books"
FROM title_authors
GROUP BY au_id
HAVING COUNT(*) >= 3;
```

| au_id | num_books |
|-------|-----------|
| A01   | 3         |
| A02   | 4         |
| A04   | 4         |
| A06   | 3         |

图6-18 运行代码6-18的结果

在代码6-19和图6-19中，HAVING条件也是一个在SELECT子句中的聚合表达式。如果将AVG()表达式从SELECT列表中去掉，这个查询仍能运行（代码6-20和图6-20）。

**代码6-19** 列出平均收入大于1 000 000美元类型的图书的数量和平均收入。结果见图6-19

```
SELECT
 type,
 COUNT(price) AS "COUNT(price)",
 AVG(price * sales) AS "AVG revenue"
FROM titles
GROUP BY type
HAVING AVG(price * sales) > 1000000;
```

| type      | COUNT(price) | AVG revenue |
|-----------|--------------|-------------|
| biography | 3            | 12484878.00 |
| computer  | 1            | 1025396.65  |

图6-19 运行代码6-19的结果

**代码6-20** 如果在SELECT列表中没有AVG(price\*sales)，代码6-19仍能运行。结果见图6-20

```
SELECT
 type,
```

```
COUNT(price) AS "COUNT(price)"
FROM titles
GROUP BY type
HAVING AVG(price * sales) > 1000000;
```

在代码6-21和图6-21中，多个分组列统计了每一个出版社出版的每一种类型的图书的数量。HAVING条件去掉了出版社的特定类型中只有一种或更少图书的分组。这个查询检索了本章前面的代码6-14结果的子集。

**代码6-21** 对于一种图书类型多于一本图书的出版社，列出每一个出版社每一种类型的图书的数量。结果见图6-21

```
SELECT
 pub_id,
 type,
 COUNT(*) AS "COUNT(*)"
FROM titles
GROUP BY pub_id, type
HAVING COUNT(*) > 1
ORDER BY pub_id ASC, "COUNT(*)" DESC;
```

在代码6-22和图6-22中，WHERE子句首先去掉除出版社P03和P04图书以外的所有行。GROUP BY子句对WHERE子句的输出按照类型进行分组。最后，HAVING子句从分组后的结果中筛选行。

**代码6-22** 在出版社P03和P04的图书中，对于销售量的总和高于10 000美元且平均价格低于20美元的类型，按照类型列出销售量的总和及平均价格。结果见图6-22

```
SELECT
 type,
 SUM(sales) AS "SUM(sales)",
 AVG(price) AS "AVG(price)"
FROM titles
WHERE pub_id IN ('P03', 'P04')
GROUP BY type
HAVING SUM(sales) > 10000
 AND AVG(price) < 20;
```

### ✓ 提示

□ 通常，HAVING子句只包含聚合。在HAVING子句中定义的条件必须在分组运算完成之后再应用。在WHERE子句中定义的可以在分组运算之前应用的条件更加高效。例如，下面的语句是等价的，但是因为第一条语句减少了分组后的行数，因此更可取。

```
SELECT pub_id, SUM(sales) --Faster
 FROM titles
 WHERE pub_id IN ('P03', 'P04')
 GROUP BY pub_id
 HAVING SUM(sales) > 10000;

SELECT pub_id, SUM(sales) --Slower
 FROM titles
 GROUP BY pub_id
 HAVING SUM(sales) > 10000
 AND pub_id IN ('P03', 'P04');
```

| type      | COUNT(price) |
|-----------|--------------|
| biography | 3            |
| computer  | 1            |

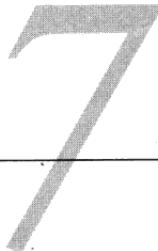
图6-20 运行代码6-20的结果

| pub_id | type       | COUNT(*) |
|--------|------------|----------|
| P01    | biography  | 3        |
| P03    | history    | 2        |
| P04    | psychology | 3        |
| P04    | children   | 2        |

图6-21 运行代码6-21的结果

| type       | SUM(sales) | AVG(price) |
|------------|------------|------------|
| psychology | 308564     | 9.31       |

图6-22 运行代码6-22的结果



**至** 此章的所有查询只是从一个表中检索行。本章讲解如何使用联结同时从多个表中检索行。回顾2.4节，关系是两个表中含义相同的列之间确定的关联。联结是使用关联的列将两个输入表合并为一个结果的表操作。可以串联多个联结，从无限个表中检索行。

为什么联结很重要呢？因为最重要的数据库信息不只是存储在各个表行中的内容，还包括相关行的集合之间隐含的关系。例如，在示例数据库中，表authors、publishers和titles中的每一行当然包含了重要的信息，但要全面理解和分析数据，了解谁写了什么，哪个出版社出版了什么，将稿酬支票付给谁了，付了多少等等，还是要依靠隐含关系。

本章讲解了各类联结，为什么使用联结，以及如何使用联结创建SELECT语句。

## 7.1 限定列名

本书2.1节提到过，列名在一个表中必须是唯一的，但是可以在其他表中使用。例如，示例数据库中的表authors和publishers都包含名为city的列。

要在包含多个表的查询中明确地标识某列，应该使用它的限定名称。限定名称是一个表名后跟一个点号和表中的列名。因为数据库中各表必须有不同的名字，因此限定名称可以在整个数据库中唯一地标识出某个列。

### ⇒ 限定列名

输入：

*table.column*

*column*是列名，*table*是包含*column*的表名（代码7-1和图7-1）。

**代码7-1** 限定名称解决了表authors和publishers中都有名为city的列的问题。结果见图7-1

```
SELECT au_id, authors.city
 FROM authors
 INNER JOIN publishers
 ON authors.city = publishers.city;
```

au\_id city

|     |               |
|-----|---------------|
| A03 | San Francisco |
| A04 | San Francisco |
| A05 | New York      |

✓ 提示

□ 可以在同一个语句中混合使用限定名称和非

图7-1 运行代码7-1的结果。结果列出了居住在某出版社所在城市的作者。联结的语法将在本章后面介绍

194

限定名称。

- 如果没有歧义，也就是说查询的表没有相同的列名，就不一定要用限定名称。但是，要提高系统性能，应该在联结查询中对所有列使用限定名称。
- 使用限定名称的另一个原因是确保表结构的更改不会引入二义性。如果某人对表**publishers**添加列**zip**，再从表**authors**（已经包含列**zip**）和**publishers**建立的查询中无限定地引用**zip**就会是含糊不清的。
- 在只包含一个表的查询中，限定名称仍然适用。事实上，每一列都有一个隐含的限定词。下面两条语句是等价的。

```
SELECT
 au_fname,
 au_lname
FROM authors;
```

和

```
SELECT
 authors.au_fname,
 authors.au_lname
FROM authors;
```

- **DBMS** 有的查询也许需要更多的限定词（取决于它在DBMS层次结构中的位置），例如需要对表使用服务器、数据库或模式名进行限定（见第2章中的表2-2）。下一节介绍的表别名，在需要长限定名称的SQL语句中非常有用。例如，在Microsoft SQL Server中的完整限定表名如下。

*server.database.owner.table*

Oracle 8i需要**WHERE**联结语法，见7.4节。要运行代码7-1，输入：

```
SELECT au_id, authors.city
 FROM authors, publishers
 WHERE authors.city =
 publishers.city;
```

195

## 7.2 使用 AS 创建表的别名

可以使用AS像4.2节创建列的别名那样创建表的别名。表的别名：

- 可以减少输入；
- 可以减少语句混乱；
- 只在语句持续的时间内存在；
- 不出现在结果中（不同于列的别名）；
- 不更改数据库中的表名；
- 在子查询的上下文中也称为相关名称，参见第8章。

### ⇒ 创建表的别名

在**FROM**或者**JOIN**子句中，输入：

*table [AS] alias*

*table*是表名，*alias*是它的别名。*alias*是一个单词，不用引号，只包含字母、数字或下划线，且不能使用空格、标点或特殊字符。AS关键字是可选的（代码7-2和图7-2）。

**代码7-2 表的别名使查询更简短和易读。注意，SELECT子句可以在后面的语句定义别名之前先使用别名。结果见图7-2**

```
SELECT au_fname, au_lname, a.city
 FROM authors a
 INNER JOIN publishers p
 ON a.city = p.city;
```

| au_fname  | au_lname | city          |
|-----------|----------|---------------|
| Hallie    | Hull     | San Francisco |
| Klee      | Hull     | San Francisco |
| Christian | Kells    | New York      |

图7-2 运行代码7-2的结果

### ✓ 提示

- 在本书中，为了DBMS间的可移植性省略了关键字AS（参见本节中的DBMS提示）。
- 实际上，表的别名较短（通常只有一两个字符），但长的名字也有效。
- 如果想使用特定表的实际名称，则省略它的别名。
- 别名隐藏了表名。如果为表命名了别名，则必须在所有的限定引用中使用它的别名。因为别名a遮蔽了表的名称authors，所以下面的语句是非法的。

```
SELECT authors.au_id
 FROM authors a; --Illegal
```

- **DBMS** PostgreSQL会向FROM子句中隐式地添加出现在SELECT子句中的表名，而这可能会引起意外的交叉联结。例如，查询SELECT titles.title\_id FROM titles t;交叉联结表titles，返回169 (13<sup>2</sup>) 行而不是报错。要避免这种情况发生，使用SET ADD\_MISSING\_FROM TO FALSE;。
- 在同一条SQL语句中，每一个表的别名必须是唯一的。
- 在自联结中，多次引用同一个表，则表的别名是必需的，参见7.9节。
- 也可以使用AS为视图分配别名，参见第13章。
- 不可以使用关键字作为表的别名，参见3.1节。
- **DBMS** 在Oracle中，当创建表的别名时，必须省略关键字AS。

Oracle 8i需要应用WHERE联结语法，参见7.4节。要运行代码7-2，输入：

```
SELECT a.au_fname, a.au_lname,
 a.city
 FROM authors a, publishers p
 WHERE a.city = p.city;
```

196

7

197

## 7.3 使用联结

可以使用联结从多个表中提取数据。本章的其余部分介绍了各类联结（表7-1），为什么要使用它们，以及如何创建使用它们的SELECT语句。

联结的主要特点如下。

表7-1 联结类型

| 联 结                     | 描 述                                                                    |
|-------------------------|------------------------------------------------------------------------|
| 交叉联结 (cross join)       | 返回第一个表的每一行和第二个表的所有行组合得到的表的所有行                                          |
| 自然联结 (natural join)     | 对第一个表的所有列和第二个表具有相同名字的列进行等同比较的联结                                        |
| 内联结 (inner join)        | 使用比较操作符基于每一个表中共同列的值，匹配源自两个表的行。内联结是最普通的联结类型                             |
| 左外联结 (left outer join)  | 返回左表中的所有行，不管与右表是否有匹配的联结列。如果左表中的行在右表没有匹配的行，关联的结果对于右表的所有SELECT子句列包含空值    |
| 右外联结 (right outer join) | 与左外联结相反。返回右表中的所有行。如果右表中的行没有匹配左表中的行，则对于左表返回空值                           |
| 全外联结 (full outer join)  | 返回左表和右表中所有的行。如果一行在另一表中没有匹配行，另一表的SELECT子句列包含空值。如果两个表之间匹配，则整个结果行包含两个表中的值 |
| 自联结 (self-join)         | 表和它的自身联结                                                               |

- 两个联结操作数（输入表）通常被称为第一个表和第二个表，但在表的顺序很重要的外联结（outer join）中，它们被称为左表和右表。
- 当满足查询中定义的联结条件时，表被一行行、一列列地联结。
- 不匹配的行被包括在内还是被排斥在外，这取决于联结类型。
- $\theta$ 联结 (theta join) 使用比较操作符 ( $=$ 、 $\neq$ 、 $<$ 、 $\leq$ 、 $>$ 或者 $\geq$ ) 在联结的列中进行值的比较。最普通的联结类型——等值联结 (equijoin) 是比较值是否相等的 $\theta$ 联结。
- 联结的联结列经常是关联的键列，但可以联结数据类型互相兼容的任意列（除了不需要特定联结列的交叉联结）。
- 要让联结是有意义的，需要比较的列值是定义在同一个域上的。例如，联结列titles.price 和royalties.advance是可能的，但结果没有意义。典型的联结条件指明了一个表中的外键和其他表中关联的主键（见2.2节和2.3节）。
- 如果关键字是组合的（有多个列），可以（并且通常应该）联结所有的关键字列。
- 被联结的列不必有相同的列名（除了自然联结以外）。
- 可以嵌套和串联联结两个以上的表，但要明白DBMS按照在查询中一次只执行两个表联结的方式工作。每一次联结中的两个表可以是数据库中的两个基表（base table）、一个基表和一个联结得到的表或者两个联结得到的表。
- SQL标准不限制出现在查询中的表（或联结）的数量，但是具体的DBMS将有内置限制，再则数据库管理员也许会设置比内置限制更严格的限制。一般不应联结五六个以上的表。
- 如果联结列包含空值，空值无法联结。空值表示未知的值（不被认为彼此相等或者不相等）。一个联结表的列中的空值仅当使用交叉联结或外联结时才可以返回，除非WHERE子句明确地将空值排斥在外。关于空值的知识，参见3.14节。
- 联结只存在于查询的持续时期内，并不是数据库或DBMS的一部分。
- 联结列的数据类型必须是兼容的；即DBMS可以将值转换为可以比较的共同类型。对于大多数DBMS来说，数字数据类型（如INTEGER、FLOAT和NUMERIC）、字符数据类型（CHAR、VARCHAR）、日期和时间数据类型（DATE、TIMESTAMP）是兼容的。不能联结二进制对象。
- 转换需要运算开销。为了提高性能，联结的列应该有相同的数据类型，包括是否允许空值。关于数据类型的知识，参见3.4节。

- 为了更快查询，应对联结列建立索引（参见第12章）。
- 可以将视图联结到表或其他视图（参见第13章）。
- 可以使用JOIN或者WHERE语法创建联结，见下一节。

### 域和比较

在联结和WHERE子句中比较的值，必须是可以进行有意义比较的。也就是说，它们必须有相同的数据类型和相同的意义。例如，示例数据库中的列au\_id和pub\_id有相同的数据类型（都是CHAR(3)，一个字母后跟两个数字），但是表示不同的事物，因此它们不能进行有意义的比较。

回顾2.1节，域是列允许值的集合。要防止无意义的比较，关系模型要求比较的列来自具有同样意义的域。然而，SQL和DBMS偏离了该模型，而且没有什么机制防止用户把智商和鞋子尺寸拿来作比较。如果创建了一个数据库应用程序，是阻止（或警告）用户进行浪费时间的无谓比较，还是让他们获得似是而非的比较结果（这更糟糕），完全取决于程序员。

199

7

## 7.4 使用 JOIN 或 WHERE 创建联结

有两种方法定义联结：使用JOIN或者WHERE语法。SQL-92和后续的标准规定使用JOIN语法，但较早的标准规定使用WHERE。因此，JOIN和WHERE两者都被广泛使用，在大多数的DBMS中都是合法的。

本节讲解包含两个表的JOIN和WHERE联结的一般语法。真实的查询中使用的实际语法将因联结类型、联结列的数量、联结表的数量和具体的DBMS语法要求而异。接下来的几节利用语法的图示和例子显示如何创建特定的联结。

### ⇒ 使用JOIN创建联结

输入：

```
SELECT columns
 FROM table1 join_type table2
 ON join_conditions
 [WHERE search_condition]
 [GROUP BY grouping_columns]
 [HAVING search_condition]
 [ORDER BY sort_columns];
```

*columns*是一个或多个逗号分隔的表达式，或者表*table1*或*table2*的列名。如果*table1*和*table2*有同样的列名，必须对全部查询列的引用进行限定，以防止二义性，参见7.1节。

*table1*和*table2*是被联结的表的名称。可以对表命名别名，参见7.2节。

*join\_type*定义了执行何种联结，如交叉联结、自然联结、内联结、左外联结、右外联结或者全外联结。

*join\_conditions*对于每一对联结的行计算定义了一种或多种联结条件（在交叉联结和自然联结中不允许使用ON子句）。联结条件使用下面的形式。

[*table1*.] *column op [table2.]column*

*op*通常是=，但可以是任意的比较操作符=、<>、<、<=、>或>=（见第4章中的表4-2）。可以用AND或OR组合多个联结条件，参见4.6节。

WHERE和ORDER BY子句已在第4章中介绍过，GROUP BY和HAVING已在第6章中介绍过。

200

## ⇒ 使用WHERE创建联结

输入：

```
SELECT columns
 FROM table1, table2
 WHERE join_conditions
 [GROUP BY grouping_columns]
 [HAVING search_condition]
 [ORDER BY sort_columns];
```

*columns*、*table1*和*table2*和前面7.4.1节中的参数具有相同含义。

*join\_conditions*除了*op*可以是指出联结类型的特定符号以外，也和7.4.1节中的参数具有相同的含义。**WHERE**子句也可以包含（非联结）筛选行的查询条件，参见4.5节。

**ORDER BY**子句已在第4章介绍过，**GROUP BY**和**HAVING**已在第6章介绍过。

201

代码7-3a和7-3b显示了使用**JOIN**和**WHERE**语法的等价查询，结果见图7-3。

## 代码7-3a 使用JOIN语法的联结。结果见图7-3

```
SELECT au_fname, au_lname, a.city
 FROM authors a
 INNER JOIN publishers p
 ON a.city = p.city;
```

| au_fname  | au_lname | city          |
|-----------|----------|---------------|
| Hallie    | Hull     | San Francisco |
| Klee      | Hull     | San Francisco |
| Christian | Kells    | New York      |

## 代码7-3b 使用WHERE语法的等价联结。结果见图7-3

```
SELECT au_fname, au_lname, a.city
 FROM authors a, publishers p
 WHERE a.city = p.city;
```

图7-3 运行代码7-3a和7-3b的结果

## 查询执行顺序

DBMS处理联结时，将按照以下顺序执行整个查询。

- (1) 在**JOIN**子句中应用联结条件。
- (2) 在**WHERE**子句中应用联结条件和查询条件。
- (3) 按照**GROUP BY**子句对行分组。
- (4) 在**HAVING**子句中对分组应用搜索条件。
- (5) 按照**ORDER BY**子句对结果排序。

## ✓ 提示

- 使用**WHERE**子句定义联结条件有点儿奇怪，但联结条件确实可以作为筛选器。当联结两个表时，DBMS在内部对左表的每一行和右表的每一行进行组对，形成交叉联结（见下一节），然后DBMS使用联结条件从交叉联结中筛选行（这只是从概念上如此理解，实际上任何时候DBMS的优化器都不会对每一个联结创建巨大的交叉联结表）。
- 相对于**WHERE**语法，我使用**JOIN**更可取，主要原因是**JOIN**使联结类型更直观清楚。左外联结要比A \*= B更清楚。但是，对于最普通的联结类型——简单的内联结——**WHERE**语法更容易理解。**JOIN**和**WHERE**语法都很流行，因此必须都要掌握才能看懂其他人创建的查询。

- 在三表联结中，只有一个表可以被用作另一个表和第三个表的桥梁。
- SELECT子句列表适用于引用联结表中的全部列或者列的任何子集。列表不必包含联结中每一个表的列。例如，在三表联结中，中间表的列没有一列需要出现在列表中。
- 被联结的列不必有相同的数据类型。如果数据类型不同，它们必须是可以比较的，或者DBMS可以隐式地将它们转换为一种共同的数据类型。如果数据类型不可以被隐式地转换，联结条件必须使用CAST()函数显式地转换数据类型。关于隐式和显示转换，参见5.13节。
- 如果使用带有两个以上联结条件的WHERE语法，几乎总用AND组合所有的联结条件。使用OR组合联结条件是合法的，但是结果很难解释。关于AND和OR的更多知识，参见4.6节。
- 大多数使用联结的查询可以改写为嵌套在另一个查询中的查询（子查询），反之，大多数子查询也可以改写为联结。关于子查询的知识，参见第8章。
- **DBMS** Oracle 8i和先前的版本不支持JOIN语法，应该使用WHERE联结。Oracle 9i和后续版本支持JOIN语法。DBMS也许会禁止联结带有特殊数据类型的列（特别是binary和long-text数据类型）。例如，Microsoft SQL Server禁止联结ntext、text和image列，Oracle禁止联结LOB列。查阅DBMS文档中有关joins的内容。

202

### USING子句

7

对于JOIN语法，如果被联结的列有相同的名字且被用于比较是否相等，SQL标准定义了替代ON子句的USING子句。

```
FROM table1 join_type table2
 USING (columns)
```

*columns*是逗号分隔的一个或多个列名的列表，圆括号是必需的。查询执行对于指定列对的等值联结，联结类型称为指定列联结。用USING重写代码7-3a如下。

```
SELECT au_fname, au_lname, city
 FROM authors
 INNER JOIN publishers
 USING (city);
```

USING子句的作用类似于自然联结，除非你不想联结两个表中具有相同名字的全部列对时使用它。请注意前面的USING例子只联结了两个表中的列city，然而自然联结将联结两个表共同的列city和state，参见7.6节。

USING是一个无需对SQL添加额外功能的便捷语法特征。USING子句总能够用JOIN语法的ON子句或者WHERE语法的WHERE子句改写。

**DBMS** Microsoft Access、Microsoft SQL Server和DB2不支持USING。在MySQL中使用USING查询时，需要限定SELECT子句中的公共列名。换句话说，要运行前面的例子，应该将SELECT子句中的city改为authors.city。

203

## 7.5 使用 CROSS JOIN 创建交叉联结

交叉联结可如下描述。

- 返回两个表中行的所有可能的组合。结果包含第一个表的所有行，第一个表的每一行与第二

个表的所有行组合。

- 不能使用联结条件。要创建交叉联结，如果使用JOIN语法，则省略ON子句；如果使用WHERE语法，则省略WHERE子句。
- 因为结果难以处理和解释，所以很少独立使用，而是以中间结果的形式出现在某些查询类型中（见15.1节和15.2节）。
- 即使针对很小的表，也可以产生非常巨大的结果。如果一个表有 $m$ 行而另一个表有 $n$ 行，则结果包含 $m \times n$ 行。
- 是需要很大计算开销和耗费时间的查询。
- 也被称为笛卡儿积或者交叉积。

#### ⇒ 创建交叉联结

输入：

```
SELECT columns
 FROM table1
CROSS JOIN table2
```

*columns*是一个或多个逗号分隔的表达式，或者*table1*或*table2*的列名。*table1*和*table2*是要联结的表名。如果表有某些共同的列名，用表名限定那些列名（代码7-4和图7-4）。

204

**代码7-4** 交叉联结显示两个表所有可能的行组合。结果见图7-4

```
SELECT
 au_id,
 pub_id,
 a.state AS "au_state",
 p.state AS "pub_state"
 FROM authors a
CROSS JOIN publishers p;
```

#### ✓ 提示

- 使用WHERE语法，代码等价于：

```
SELECT au_id, pub_id,
 a.state AS "au_state",
 p.state AS "pub_state"
 FROM authors a, publishers p;
```

- 使用SELECT \*检索两个表的所有列。下面这个查询检索表authors和publishers的所有列。

```
SELECT *
 FROM authors
CROSS JOIN publishers;
```

使用WHERE语法的等价语句为：

| au_id | pub_id | au_state | pub_state |
|-------|--------|----------|-----------|
| A01   | P01    | NY       | NY        |
| A02   | P01    | CO       | NY        |
| A03   | P01    | CA       | NY        |
| A04   | P01    | CA       | NY        |
| A05   | P01    | NY       | NY        |
| A06   | P01    | CA       | NY        |
| A07   | P01    | FL       | NY        |
| A01   | P02    | NY       | CA        |
| A02   | P02    | CO       | CA        |
| A03   | P02    | CA       | CA        |
| A04   | P02    | CA       | CA        |
| A05   | P02    | NY       | CA        |
| A06   | P02    | CA       | CA        |
| A07   | P02    | FL       | CA        |
| A01   | P03    | NY       | NULL      |
| A02   | P03    | CO       | NULL      |
| A03   | P03    | CA       | NULL      |
| A04   | P03    | CA       | NULL      |
| A05   | P03    | NY       | NULL      |
| A06   | P03    | CA       | NULL      |
| A07   | P03    | FL       | NULL      |
| A01   | P04    | NY       | CA        |
| A02   | P04    | CO       | CA        |
| A03   | P04    | CA       | CA        |
| A04   | P04    | CA       | CA        |
| A05   | P04    | NY       | CA        |
| A06   | P04    | CA       | CA        |
| A07   | P04    | FL       | CA        |

图7-4 运行代码7-4的结果

```
SELECT *
 FROM authors, publishers;
```

- 使用SELECT table.\*检索一个表的所有列。下面的查询检索表authors的所有行和表publishers的列pub\_id。

```
SELECT authors.*, p.pub_id
 FROM authors
 CROSS JOIN publishers p;
```

使用WHERE语法的等价语句为：

```
SELECT authors.*, p.pub_id
 FROM authors, publishers p;
```

- 使用JOIN语法得到n个表的交叉积，输入：

```
SELECT columns
 FROM table1
 CROSS JOIN table2
 ...
 CROSS JOIN tableN
```

使用WHERE语法的等价语句为：

```
SELECT columns
 FROM table1, table2, ..., tableN
```

- 交叉积经常会错误地产生。如果结果包含意想不到的大量的行，或许是因为遗漏了查询的联结条件。  
 □ 尽管交叉积实际上很少有用，但DBMS内部（理论上）会在每个联结处理过程的第一步中产生交叉积。在DBMS产生了交叉积之后，它使用SELECT子句列表删除列，使用联结和查询条件删除行。

#### □ 联结

*t1* CROSS JOIN *t2*

等价于下面的任何一个联结。

```
t1 INNER JOIN t2 ON 1 = 1
t1 LEFT OUTER JOIN t2 ON 1 = 1
t1 RIGHT OUTER JOIN t2 ON 1 = 1
t1 FULL OUTER JOIN t2 ON 1 = 1
```

*t1*和*t2*是表，并且1=1代表总为true的任意条件。内联结和外联结将在本章后面介绍。

- 交叉联结的一个实际应用是产生测试软件的数据集。假设有一个接收n个参数、每个参数有m个测试值的函数，可以通过计算n个表（每一个表有一列和m行，也就是一行包含一个测试值）的交叉积产生m×n个测试用例。即便对于每一个参数m是不同的，这种方法照样可以使用。  
 □ **DBMS** Microsoft Access和DB2对于交叉联结只支持WHERE语法。要运行代码7-4，使用本节第  
 一个提示中给出的语句。Oracle 8i不支持JOIN语法，应该使用WHERE联结。

7

205

## 7.6 使用 NATURAL JOIN 创建自然联结

自然联结可如下描述。

- 是等值联结的一个特例，它比较一个表中所有的列和其他表中具有相同名称的相应列的等同性。

- 只在输入表有一对或多对有意义、可以比较、名称一样的列时才起作用。
- 隐式地执行联结。不能在自然联结中定义ON或USING子句。
- 是一种便捷语法，可以使用JOIN语法的ON子句，或者WHERE语法的WHERE子句改写。

### ⇒ 创建自然联结

输入：

```
SELECT columns
 FROM table1
 NATURAL JOIN table2
```

*columns*是一个或多个逗号分隔的表达式，或者*table1*或*table2*的列名。具体的DBMS也许需要用表名限定同样的列名（参见本节中的DBMS提示）。*table1*和*table2*是联结表的表名。

*table1*中的列与*table2*中的同名列联结，并且比较是否相等。NATURAL JOIN创建自然内联结。要创建自然外联结，请参见本节中的提示。

DBMS在运行代码7-5时，会将表publishers的publishers.pub\_id列与表titles的titles.pub\_id  
206 列（这两列同名）中的值相等的行联结起来，结果见图7-5。

代码7-6在代码7-5的基础上添加了另一个联结，以检索给每一本书的预付款。WHERE条件检索预付款低于20 000美元的书。DBMS运行代码7-6时，不仅会联结表publishers和titles中的pub\_id列，还将联结表titles和royalties中的title\_id列，结果见图7-6。

#### 代码7-5 列出每一本书的出版社。结果见图7-5

```
SELECT
 title_id,
 pub_id,
 pub_name
 FROM publishers
 NATURAL JOIN titles;
```

#### 代码7-6 列出每一本书的出版社和预付款低于20 000美元的图书的预付款。结果见图7-6

```
SELECT
 title_id,
 pub_id,
 pub_name,
 advance
 FROM publishers
 NATURAL JOIN titles
 NATURAL JOIN royalties
 WHERE advance < 20000;
```

#### ✓ 提示

- 使用WHERE语法改写自然联结，应该使用带有WHERE子句（使用AND操作符组合联结条件）的等

| title_id | pub_id | pub_name            |
|----------|--------|---------------------|
| T01      | P01    | Abatis Publishers   |
| T02      | P03    | Schadenfreude Press |
| T03      | P02    | Core Dump Books     |
| T04      | P04    | Tenterhooks Press   |
| T05      | P04    | Tenterhooks Press   |
| T06      | P01    | Abatis Publishers   |
| T07      | P03    | Schadenfreude Press |
| T08      | P04    | Tenterhooks Press   |
| T09      | P04    | Tenterhooks Press   |
| T10      | P01    | Abatis Publishers   |
| T11      | P04    | Tenterhooks Press   |
| T12      | P01    | Abatis Publishers   |
| T13      | P03    | Schadenfreude Press |

图7-5 运行代码7-5的结果

| title_id | pub_id | pub_name            | advance |
|----------|--------|---------------------|---------|
| T01      | P01    | Abatis Publishers   | 10000   |
| T02      | P03    | Schadenfreude Press | 1000    |
| T03      | P02    | Core Dump Books     | 15000   |
| T08      | P04    | Tenterhooks Press   | 0       |
| T09      | P04    | Tenterhooks Press   | 0       |

图7-6 运行代码7-6的结果

值联结。每一个联结条件等同于输入表中有相同名称的每一列对。等价的WHERE查询分别是(代码7-5):

```
SELECT t.title_id, t.pub_id,
 p.pub_name
 FROM publishers p, titles t
 WHERE p.pub_id = t.pub_id;
```

和 (代码7-6):

```
SELECT t.title_id, t.pub_id,
 p.pub_name, r.advance
 FROM publishers p, titles t,
 royalties r
 WHERE p.pub_id = t.pub_id
 AND t.title_id = r.title_id
 AND r.advance < 20000;
```

207

- 使用内联结或外联结的JOIN语法改写自然联结，应该使用带有ON子句（使用AND操作符组合联结条件）的等值联结。每一个联结条件等同于输入表中有相同名字的每一列对。等价的JOIN查询分别是 (代码7-5):

```
SELECT t.title_id, t.pub_id,
 p.pub_name
 FROM publishers p
 INNER JOIN titles t
 ON p.pub_id = t.pub_id;
```

7

和 (代码7-6):

```
SELECT t.title_id, t.pub_id,
 p.pub_name, r.advance
 FROM publishers p
 INNER JOIN titles t
 ON p.pub_id = t.pub_id
 INNER JOIN royalties r
 ON t.title_id = r.title_id
 WHERE r.advance < 20000;
```

- 也可以使用带有USING子句的JOIN语法改写自然联结 (见7.4节中的提要栏)。NATURAL JOIN是USING的速记形式，它形成了一个在两个表中列名相同的列的列表。等价的USING查询分别是 (代码7-5):

```
SELECT title_id, pub_id,
 pub_name
 FROM publishers
 INNER JOIN titles
 USING (pub_id);
```

和 (代码7-6):

```
SELECT title_id, pub_id,
 pub_name, advance
 FROM publishers
 INNER JOIN titles
 USING (pub_id)
 INNER JOIN royalties
 USING (title_id)
```

```
WHERE advance < 20000;
```

- NATURAL JOIN**语法实际上产生了一个内联结: **NATURAL JOIN**等价于**NATURAL INNER JOIN**。可以通过下列语法创建自然外联结:

```
NATURAL LEFT [OUTER] JOIN
NATURAL RIGHT [OUTER] JOIN
NATURAL FULL [OUTER] JOIN
```

208

内联结和外联结将在本章后面介绍。

- 使用自然联结, 要确保两个联接表中所有相关列有相同的列名, 且所有非相关列有唯一列名。
- 使用自然联结可以把某些查询写得更简短和易懂, 但是要小心它们。如果在不知情时包含在联结中的列被增加、删除或重命名, 它们将返回意想不到的结果。
- 自然联结的含义在关系模型(见第2章)和SQL标准中略微有些不同。在关系模型中, 自然联结总是从外键到它的父键的联结。在SQL中, 自然联结是两个表中有相同名称的全部列(不一定正好是关键字列)的联结。参阅本章后面代码7-9中不包含关键字列的自然联结的例子。要使模型和SQL对于自然联结的定义一致起来, 数据库设计人员应该确保所有的外键有与它们的父键相同的名字, 并且所有的其他列都有不同的列名。
- DBMS** Microsoft Access、Microsoft SQL Server和DB2不支持**NATURAL JOIN**语法。要在这些DBMS中运行代码7-5和代码7-6, 可以使用**WHERE**语法(本节第一个提示中说明), 或者使用等价的**JOIN**语法(本节的第二个提示中说明)。

Oracle 8i不支持**JOIN**语法, 应该使用**WHERE**联结。

MySQL 4.1和之前的版本需要在自然联结中限定相同的列名。即要运行代码7-5和代码7-6, 应添加限定符(代码7-5):

```
SELECT
 t.title_id,
 t.pub_id,
 p.pub_name
FROM publishers p
NATURAL JOIN titles t;
```

和(代码7-6):

```
SELECT
 t.title_id,
 t.pub_id,
 p.pub_name,
 r.advance
FROM publishers p
NATURAL JOIN titles t
NATURAL JOIN royalties r
WHERE r.advance < 20000;
```

209

## 7.7 使用 INNER JOIN 创建内联结

内联结可如下描述。

- 使用比较操作符(=、<>、<、<=、>或>=)匹配两个表的行(基于每个表同名列的值)。例如, 可以检索表authors和title\_authors中作者标识符(列au\_id)相同的所有行。
- 只返回满足联结条件的联结行的结果。

- 是最普通的联结类型。

### ⇒ 创建内联结

输入：

```
SELECT columns
 FROM table1
 INNER JOIN table2
 ON join_conditions
```

*columns*是一个或多个逗号分隔的表达式或者*table1*、*table2*的列名。*table1*和*table2*是要联结的表名。如果表有同名的列，加上表名限定这些列名。

*join\_conditions*定义一个或多个联结条件以判定每一对联结的行。联结条件采用这种形式：

[*table1*.] *column* *op* [*table2*.] *column*

*op*通常是=，也可以是任意的比较操作符：=、<、<=、>或>=（见第4章中的表4-2）。可以使用AND或OR组合多个联结条件，参见4.6节。

### ✓ 提示

- 要使用JOIN语法创建3个以上的表的内联结，输入：

```
SELECT columns
 FROM table1
 INNER JOIN table2
 ON join_condition1
 INNER JOIN table3
 ON join_condition2
 ...
 ...
```

使用WHERE语法，输入：

```
SELECT columns
 FROM table1, table2, ...
 WHERE join_condition1
 AND join_condition2
 ...
 ...
```

- 如果使用WHERE语法并且错误地省略了联结条件，将创建一个交叉联结。如果涉及的是一个巨大的表，将产生失控的查询，最终不得不去请数据库管理员终止该查询。
- 默认情况下，JOIN（没有CROSS、NATURAL、OUTER或其他限定符）等价于INNER JOIN。
- **DBMS** 可以在Microsoft Access中使用WHERE语法或JOIN语法，但是如果在联结中使用包含3个以上的表的JOIN语法，Access需要使用下面的一般形式嵌套联结。

```
SELECT columns
 FROM table1
 INNER JOIN (table2
 INNER JOIN (table3
 INNER JOIN (table4
 INNER JOIN ...))
 ON table3.column3 op table4.column4
 ON table2.column2 op table3.column3)
 ON table1.column1 op table2.column2;
```

7

210

(其他的DBMS也可以使用圆括号嵌套联结, 但Access则必须使用。)

Oracle 8i不支持JOIN语法, 改为使用WHERE联结。Oracle 9i和之后的版本支持JOIN语法。

代码7-7在列au\_id上联结两个表, 列出每一位作者写(或合写)的书。表authors中每一位作者的au\_id匹配表title\_authors中零行或多行, 结果见图7-7。请注意结果中未包含作者A07(Paddy O'Furniture), 因为他没有写书, 因此在title\_authors中没有匹配的行。

### 代码7-7 列出每一位作者写(或合写)的书。结果见图7-7

```
SELECT
 a.au_id,
 a.au_fname,
 a.au_lname,
 ta.title_id
FROM authors a
INNER JOIN title_authors ta
 ON a.au_id = ta.au_id
ORDER BY a.au_id ASC, ta.title_id ASC;
```

#### ✓ 提示

□ 使用WHERE语法, 代码7-7等价于:

```
SELECT a.au_id, a.au_fname,
 a.au_lname, ta.title_id
FROM authors a, title_authors ta
WHERE a.au_id = ta.au_id
ORDER BY a.au_id ASC,
 ta.title_id ASC;
```

211

代码7-8在列pub\_id上联结了两个表, 列出了每一本书的书名和ID, 以及每一本书的出版社名和ID。只对检索出版社名(结果中的第4列)来说, 联结是必需的。注意, 其他的3列可从表title中得到, 结果见图7-8。

#### ✓ 提示

□ 使用WHERE语法, 代码7-8等价于:

```
SELECT t.title_id, t.title_name,
 t.pub_id, p.pub_name
FROM titles t, publishers p
WHERE p.pub_id = t.pub_id
ORDER BY t.title_name ASC;
```

212

### 代码7-8 列出每一本书的书名和ID, 以及书的出版社名和ID。结果见图7-8

```
SELECT
 t.title_id,
 t.title_name,
 t.pub_id,
 p.pub_name
FROM titles t
INNER JOIN publishers p
```

| au_id | au_fname  | au_lname  | title_id |
|-------|-----------|-----------|----------|
| A01   | Sarah     | Buchman   | T01      |
| A01   | Sarah     | Buchman   | T02      |
| A01   | Sarah     | Buchman   | T13      |
| A02   | Wendy     | Heydemark | T06      |
| A02   | Wendy     | Heydemark | T07      |
| A02   | Wendy     | Heydemark | T10      |
| A02   | Wendy     | Heydemark | T12      |
| A03   | Hallie    | Hull      | T04      |
| A03   | Hallie    | Hull      | T11      |
| A04   | Klee      | Hull      | T04      |
| A04   | Klee      | Hull      | T05      |
| A04   | Klee      | Hull      | T07      |
| A04   | Klee      | Hull      | T11      |
| A05   | Christian | Kells     | T03      |
| A06   |           | Kellsey   | T08      |
| A06   |           | Kellsey   | T09      |
| A06   |           | Kellsey   | T11      |

图7-7 运行代码7-7的结果

```
ON p.pub_id = t.pub_id
ORDER BY t.title_name ASC;
```

| title_id | title_name                          | pub_id | pub_name            |
|----------|-------------------------------------|--------|---------------------|
| T01      | 1977!                               | P01    | Abatis Publishers   |
| T02      | 200 Years of German Humor           | P03    | Schadenfreude Press |
| T03      | Ask Your System Administrator       | P02    | Core Dump Books     |
| T04      | But I Did It Unconsciously          | P04    | Tenterhooks Press   |
| T05      | Exchange of Platitudes              | P04    | Tenterhooks Press   |
| T06      | How About Never?                    | P01    | Abatis Publishers   |
| T07      | I Blame My Mother                   | P03    | Schadenfreude Press |
| T08      | Just Wait Until After School        | P04    | Tenterhooks Press   |
| T09      | Kiss My Boo-Boo                     | P04    | Tenterhooks Press   |
| T10      | Not Without My Faberge Egg          | P01    | Abatis Publishers   |
| T11      | Perhaps It's a Glandular Problem    | P04    | Tenterhooks Press   |
| T12      | Spontaneous, Not Annoying           | P01    | Abatis Publishers   |
| T13      | What Are The Civilian Applications? | P03    | Schadenfreude Press |

图7-8 运行代码7-8的结果

7

代码7-9使用两个联结条件列出居住在出版社所在城市和州的作者，结果见图7-9。注意，这个查询是在两个表中名称相同、非键列city和state上的自然联结（见7.6节）。等价的查询如下：

```
SELECT a.au_id, a.au_fname,
 a.au_lname, a.city, a.state
 FROM authors a
 NATURAL JOIN publishers p
 ORDER BY a.au_id ASC;
```

代码7-9 列出居住在出版社所在城市和州的作者。结果见图7-9

```
SELECT
 a.au_id,
 a.au_fname,
 a.au_lname,
 a.city,
 a.state
 FROM authors a
 INNER JOIN publishers p
 ON a.city = p.city
 AND a.state = p.state
 ORDER BY a.au_id;
```

✓ 提示

□ 使用WHERE语法，代码7-9等价于：

```
SELECT a.au_id, a.au_fname,
 a.au_lname, a.city, a.state
 FROM authors a, publishers p
 WHERE a.city = p.city
```

| au_id | au_fname  | au_lname | city          | state |
|-------|-----------|----------|---------------|-------|
| A03   | Hallie    | Hull     | San Francisco | CA    |
| A04   | Klee      | Hull     | San Francisco | CA    |
| A05   | Christian | Kells    | New York      | NY    |

图7-9 运行代码7-9的结果

213

```
 AND a.state = p.state
 ORDER BY a.au_id ASC;
```

代码7-10组合使用WHERE条件和内联结，列出出版于加利福尼亚或北美地区以外国家的图书（见4.5节），结果见图7-10。

#### 代码7-10 列出出版于加利福尼亚或北美地区以外国家的图书。结果见图7-10

```
SELECT
 t.title_id,
 t.title_name,
 p.state,
 p.country
FROM titles t
INNER JOIN publishers p
 ON t.pub_id = p.pub_id
WHERE p.state = 'CA'
 OR p.country NOT IN ('USA', 'Canada', 'Mexico')
ORDER BY t.title_id ASC;
```

| title_id | title_name                          | state | country |
|----------|-------------------------------------|-------|---------|
| T02      | 200 Years of German Humor           | NULL  | Germany |
| T03      | Ask Your System Administrator       | CA    | USA     |
| T04      | But I Did It Unconsciously          | CA    | USA     |
| T05      | Exchange of Platitudes              | CA    | USA     |
| T07      | I Blame My Mother                   | NULL  | Germany |
| T08      | Just Wait Until After School        | CA    | USA     |
| T09      | Kiss My Boo-Boo                     | CA    | USA     |
| T11      | Perhaps It's a Glandular Problem    | CA    | USA     |
| T13      | What Are The Civilian Applications? | NULL  | Germany |

图7-10 运行代码7-10的结果

#### ✓ 提示

□ 使用WHERE语法，代码7-10等价于：

```
SELECT t.title_id, t.title_name,
 p.state, p.country
FROM titles t, publishers p
WHERE t.pub_id = p.pub_id
 AND (p.state = 'CA'
 OR p.country NOT IN
 ('USA', 'Canada', 'Mexico'))
ORDER BY t.title_id ASC;
```

214

代码7-11组合使用带有聚合函数COUNT()的内联结和GROUP BY子句，列出每一位作者写或合写的书的数量。（关于聚合函数和GROUP BY子句的介绍，见第6章。）结果见图7-11。注意，与图7-7一样，结果中未包含作者A07（Paddy O'Furniture），因为他没有写书，在表title\_authors中没有匹配的行。7.8节中的代码7-30用来查询没有写过书的人。

**代码7-11** 列出每一位作者写或合写书的数量。结果见图7-11

```
SELECT
 a.au_id,
 COUNT(ta.title_id) AS "Num books"
FROM authors a
INNER JOIN title_authors ta
 ON a.au_id = ta.au_id
GROUP BY a.au_id
ORDER BY a.au_id ASC;
```

✓ 提示

□ 使用WHERE语法，代码7-11等价于：

```
SELECT a.au_id,
 COUNT(ta.title_id)
 AS "Num books"
FROM authors a, title_authors ta
WHERE a.au_id = ta.au_id
GROUP BY a.au_id
ORDER BY a.au_id ASC;
```

| au_id | Num books |
|-------|-----------|
| A01   | 3         |
| A02   | 4         |
| A03   | 2         |
| A04   | 4         |
| A05   | 1         |
| A06   | 3         |

图7-11 运行代码7-11的结果

代码7-12使用WHERE条件列出每一本传记的预付款，结果见图7-12。

**代码7-12** 列出每一本传记的预付款。结果见图7-12

```
SELECT
 t.title_id,
 t.title_name,
 r.advance
FROM royalties r
INNER JOIN titles t
 ON r.title_id = t.title_id
WHERE t.type = 'biography'
 AND r.advance IS NOT NULL
ORDER BY r.advance DESC;
```

✓ 提示

□ 使用WHERE语法，代码7-12等价于：

```
SELECT t.title_id, t.title_name,
 r.advance
FROM royalties r, titles t
WHERE r.title_id = t.title_id
 AND t.type = 'biography'
 AND r.advance IS NOT NULL
ORDER BY r.advance DESC;
```

| title_id | title_name                | advance    |
|----------|---------------------------|------------|
| T07      | I Blame My Mother         | 1000000.00 |
| T12      | Spontaneous, Not Annoying | 50000.00   |
| T06      | How About Never?          | 20000.00   |

图7-12 运行代码7-12的结果

代码7-13使用聚合函数和GROUP BY子句列出每一种类型书的数量和总的预付款，结果见图7-13。

**代码7-13** 列出每一种类型图书的数量和总的预付款。结果见图7-13

```
SELECT
 t.type,
 COUNT(r.advance)
 AS "COUNT(r.advance)",
```

215

7

216

```

SUM(r.advance)
AS "SUM(r.advance)"
FROM royalties r
INNER JOIN titles t
ON r.title_id = t.title_id
WHERE r.advance IS NOT NULL
GROUP BY t.type
ORDER BY t.type ASC;

```

## ✓ 提示

□ 使用WHERE语法，代码7-13等价于：

```

SELECT t.type,
COUNT(r.advance)
AS "COUNT(r.advance)",
SUM(r.advance)
AS "SUM(r.advance)"
FROM royalties r, titles t
WHERE r.title_id = t.title_id
AND r.advance IS NOT NULL
GROUP BY t.type
ORDER BY t.type ASC;

```

217

| type       | COUNT(r.advance) | SUM(r.advance) |
|------------|------------------|----------------|
| biography  | 3                | 1070000.00     |
| children   | 2                | 0.00           |
| computer   | 1                | 15000.00       |
| history    | 3                | 31000.00       |
| psychology | 3                | 220000.00      |

图7-13 运行代码7-13的结果

代码7-14类似于代码7-13，但它使用了一个附加的分组列按出版社列出每一种类型图书的数量和总的预付款，结果见图7-14。

## 代码7-14 按出版社列出每一种类型图书的数量和总的预付款。结果见图7-14

```

SELECT
t.type,
t.pub_id,
COUNT(r.advance) AS "COUNT(r.advance)",
SUM(r.advance) AS "SUM(r.advance)"
FROM royalties r
INNER JOIN titles t
ON r.title_id = t.title_id
WHERE r.advance IS NOT NULL
GROUP BY t.type, t.pub_id
ORDER BY t.type ASC, t.pub_id ASC;

```

218

| type       | pub_id | COUNT(r.advance) | SUM(r.advance) |
|------------|--------|------------------|----------------|
| biography  | P01    | 2                | 70000.00       |
| biography  | P03    | 1                | 1000000.00     |
| children   | P04    | 2                | 0.00           |
| computer   | P02    | 1                | 15000.00       |
| history    | P01    | 1                | 10000.00       |
| history    | P03    | 2                | 21000.00       |
| psychology | P04    | 3                | 220000.00      |

图7-14 运行代码7-14的结果

## ✓ 提示

- 使用 WHERE 语法，代码 7-14 等价于：

```
SELECT t.type, t.pub_id,
 COUNT(r.advance)
 AS "COUNT(r.advance)",
 SUM(r.advance)
 AS "SUM(r.advance)"
FROM royalties r, titles t
WHERE r.title_id = t.title_id
 AND r.advance IS NOT NULL
GROUP BY t.type, t.pub_id
ORDER BY t.type ASC, t.pub_id ASC;
```

代码 7-15 使用 HAVING 子句列出多作者合著的书的数量。(关于 HAVING 的知识，参见 6.10 节。) 结果见图 7-15。

代码 7-15 列出有多作者合著的书的数量。结果见图 7-15

```
SELECT
 ta.title_id,
 COUNT(ta.au_id) AS "Num authors"
FROM authors a
INNER JOIN title_authors ta
 ON a.au_id = ta.au_id
GROUP BY ta.title_id
HAVING COUNT(ta.au_id) > 1
ORDER BY ta.title_id ASC;
```

| title_id | Num authors |
|----------|-------------|
| T04      | 2           |
| T07      | 2           |
| T11      | 3           |

图 7-15 运行代码 7-15 的结果

7

## ✓ 提示

- 使用 WHERE 语法，代码 7-15 等价于：

```
SELECT ta.title_id,
 COUNT(ta.au_id) AS "Num authors"
FROM authors a, title_authors ta.
WHERE a.au_id = ta.au_id
GROUP BY ta.title_id
HAVING COUNT(ta.au_id) > 1
ORDER BY ta.title_id ASC;
```

也可联结两个列中不相等的值。代码 7-16 使用大于号(>) 联结查找每一本书的收入(=price×sales) 比给作者的预付款大 10 倍以上的书，结果见图 7-16。使用 <、<=、>、>= 的联结很常见，但是不等联结(<>) 则很少用到。通常，不等联结仅当用于自联结时才有意义，参见 7.9 节。

219

代码 7-16 列出每一本收入(=price×sales) 比给作者的预付款大 10 倍以上的书。结果见图 7-16

```
SELECT
 t.title_id,
 t.title_name,
 r.advance,
 t.price * t.sales AS "Revenue"
FROM titles t
INNER JOIN royalties r
 ON t.price * t.sales > r.advance * 10
 AND t.title_id = r.title_id
ORDER BY t.price * t.sales DESC;
```

220

|     |                                     | advance    | Revenue     |
|-----|-------------------------------------|------------|-------------|
| T07 | I Blame My Mother                   | 1000000.00 | 35929790.00 |
| T05 | Exchange of Platitudes              | 100000.00  | 1400008.00  |
| T12 | Spontaneous, Not Annoying           | 50000.00   | 1299012.99  |
| T03 | Ask Your System Administrator       | 15000.00   | 1025396.65  |
| T13 | What Are The Civilian Applications? | 20000.00   | 313905.33   |
| T06 | How About Never?                    | 20000.00   | 225834.00   |
| T02 | 200 Years of German Humor           | 1000.00    | 190841.70   |
| T09 | Kiss My Boo-Boo                     | .00        | 69750.00    |
| T08 | Just Wait Until After School        | .00        | 40950.00    |

图7-16 运行代码7-16的结果

**✓ 提示**

- 使用WHERE语法，代码7-16等价于：

```
SELECT t.title_id, t.title_name,
 r.advance,
 t.price * t.sales AS "Revenue"
 FROM titles t, royalties r
 WHERE t.price * t.sales >
 r.advance * 10
 AND t.title_id = r.title_id
 ORDER BY t.price * t.sales DESC;
```

有时简单的问题需要用复杂的查询来解决。在代码7-17中，必须联结3个表才能列出作者名和每一位作者写或合写的书名，结果见图7-17。

**✓ 提示**

- 使用WHERE语法，代码7-17等价于：

```
SELECT a.au_fname, a.au_lname,
 t.title_name
 FROM authors a, title_authors ta,
 titles t
 WHERE a.au_id = ta.au_id
 AND t.title_id = ta.title_id
 ORDER BY a.au_lname ASC,
 a.au_fname ASC,
 t.title_name ASC;
```

- DBMS** 要在Microsoft Access中运行代码7-17，输入：

```
SELECT a.au_fname, a.au_lname,
 t.title_name
 FROM titles AS t
 INNER JOIN authors AS a
 INNER JOIN title_authors AS ta
 ON a.au_id = ta.au_id)
 ON t.title_id = ta.title_id
```

```
ORDER BY a.au_lname ASC,
a.au_fname ASC,
t.title_name ASC;
```

**代码7-17** 列出作者名和每一位作者写或合写的书名。结果见图7-17

```
SELECT
a.au_fname,
a.au_lname,
t.title_name
FROM authors a
INNER JOIN title_authors ta
ON a.au_id = ta.au_id
INNER JOIN titles t
ON t.title_id = ta.title_id
ORDER BY a.au_lname ASC, a.au_fname ASC,
t.title_name ASC;
```

222

|                 | au_fname   | au_lname                            | title_name |
|-----------------|------------|-------------------------------------|------------|
| Sarah           | Buchman    | 1977!                               |            |
| Sarah           | Buchman    | 200 Years of German Humor           |            |
| Sarah           | Buchman    | What Are The Civilian Applications? |            |
| Wendy           | Heydermark | How About Never?                    |            |
| Wendy           | Heydermark | I Blame My Mother                   |            |
| Wendy           | Heydermark | Not Without My Faberge Egg          |            |
| Wendy           | Heydermark | Spontaneous, Not Annoying           |            |
| Hallie          | Hull       | But I Did It Unconsciously          |            |
| Hallie          | Hull       | Perhaps It's a Glandular Problem    |            |
| Klee            | Hull       | But I Did It Unconsciously          |            |
| Klee            | Hull       | Exchange of Platitudes              |            |
| Klee            | Hull       | I Blame My Mother                   |            |
| Klee            | Hull       | Perhaps It's a Glandular Problem    |            |
| Christian Kells |            | Ask Your System Administrator       |            |
| Kellsey         |            | Just Wait Until After School        |            |
| Kellsey         |            | Kiss My Boo-Boo                     |            |
| Kellsey         |            | Perhaps It's a Glandular Problem    |            |

7

图7-17 运行代码7-17的结果

作为代码7-17的扩展，代码7-18需要联结4个表，才能列出出版社名、作者名和书名，结果见图7-18。

**代码7-18** 列出作者名、每一位作者写或合写的书名和出版社名。结果见图7-18

223

```
SELECT
a.au_fname,
a.au_lname,
t.title_name,
p.pub_name
FROM authors a
INNER JOIN title_authors ta
ON a.au_id = ta.au_id
```

```

INNER JOIN titles t
 ON t.title_id = ta.title_id
INNER JOIN publishers p
 ON p.pub_id = t.pub_id
ORDER BY a.au_lname ASC, a.au_fname ASC,
t.title_name ASC;

```

| au_fname        | au_lname  | title_name                          | pub_name            |
|-----------------|-----------|-------------------------------------|---------------------|
| Sarah           | Buchman   | 1977!                               | Abatis Publishers   |
| Sarah           | Buchman   | 200 Years of German Humor           | Schadenfreude Press |
| Sarah           | Buchman   | What Are The Civilian Applications? | Schadenfreude Press |
| Wendy           | Heydemark | How About Never?                    | Abatis Publishers   |
| Wendy           | Heydemark | I Blame My Mother                   | Schadenfreude Press |
| Wendy           | Heydemark | Not Without My Faberge Egg          | Abatis Publishers   |
| Wendy           | Heydemark | Spontaneous, Not Annoying           | Abatis Publishers   |
| Hallie          | Hull      | But I Did It Unconsciously          | Tenterhooks Press   |
| Hallie          | Hull      | Perhaps It's a Glandular Problem    | Tenterhooks Press   |
| Klee            | Hull      | But I Did It Unconsciously          | Tenterhooks Press   |
| Klee            | Hull      | Exchange of Platitudes              | Tenterhooks Press   |
| Klee            | Hull      | I Blame My Mother                   | Schadenfreude Press |
| Klee            | Hull      | Perhaps It's a Glandular Problem    | Tenterhooks Press   |
| Christian Kells |           | Ask Your System Administrator       | Core Dump Books     |
| Kellsey         |           | Just Wait Until After School        | Tenterhooks Press   |
| Kellsey         |           | Kiss My Boo-Boo                     | Tenterhooks Press   |
| Kellsey         |           | Perhaps It's a Glandular Problem    | Tenterhooks Press   |

图7-18 运行代码7-18的结果

### ✓ 提示

□ 使用**WHERE**语法，代码7-18等价于：

```

SELECT a.au_fname, a.au_lname,
 t.title_name, p.pub_name
 FROM authors a, title_authors ta,
 titles t, publishers p
 WHERE a.au_id = ta.au_id
 AND t.title_id = ta.title_id
 AND p.pub_id = t.pub_id
 ORDER BY a.au_lname ASC,
 a.au_fname ASC,
 t.title_name ASC;

```

□ **DBMS** 要在Microsoft Access中运行代码7-18，输入：

```

SELECT a.au_fname, a.au_lname,
 t.title_name, p.pub_name
 FROM (publishers AS p
 INNER JOIN titles AS t
 ON p.pub_id = t.pub_id)
 INNER JOIN (authors AS a
 INNER JOIN title_authors AS ta

```

```

 ON a.au_id = ta.au_id)
 ON t.title_id = ta.title_id
ORDER BY a.au_lname ASC,
 a.au_fname ASC,
 t.title_name ASC;

```

224

代码7-19计算所有图书的稿酬总和。一本书的稿酬是书的收入乘以版税率（收入中应该支付给作者的百分比）。在大多数情况下，作者要先收一笔预付款。出版社从总稿酬中减去预付款得出净稿酬。如果净稿酬是正值，出版社就必须向作者继续付酬。如果净稿酬是负值或零，作者就没有新收入，因为他们还没有“挣回”预付款。结果见图7-19。总稿酬标记为“Total royalties”，总预付款标记为“Total advances”，净稿酬标记为“Total due to authors”。

代码7-19计算所有图书的总稿酬。本部分后面的例子将显示如何将稿酬按作者、图书、出版社和其他条件分组。

#### ✓ 提示

使用WHERE语法，代码7-19等价于：

```

SELECT
 SUM(t.sales * t.price *
 r.royalty_rate)
 AS "Total royalties",
 SUM(r.advance)
 AS "Total advances",
 SUM(t.sales * t.price *
 r.royalty_rate) - r.advance
 AS "Total due to authors"
FROM titles t, royalties r
WHERE r.title_id = t.title_id
AND t.sales IS NOT NULL;

```

7

225

#### 代码7-19 计算所有书的总稿酬。结果见图7-19

```

SELECT
 SUM(t.sales * t.price * r.royalty_rate) AS "Total royalties",
 SUM(r.advance) AS "Total advances",
 SUM(t.sales * t.price * r.royalty_rate) - r.advance AS "Total due to authors"
FROM titles t
INNER JOIN royalties r
 ON r.title_id = t.title_id
WHERE t.sales IS NOT NULL;

```

| Total royalties | Total advances | Total due to authors |
|-----------------|----------------|----------------------|
| 4387219.55      | 1336000.00     | 3051219.55           |

图7-19 运行代码7-19的结果

代码7-20使用了三表联结来计算每个作者从写（或合写）的每本书得到的稿酬。因为一本书可能有多个作者，每个作者的稿酬计算涉及该书全体作者的稿酬（及预付款）的分配。表title\_authors的列royalty\_share给出了作者每本书的稿酬份额。如果一本书只有唯一作者，royalty\_share（稿酬份额）是1.0（100%）。对于多个作者的书，每个作者的royalty\_share是一个介于0和1之间的小数。一本书所有的royalty\_share值合计必定是1.0（100%），结果见图7-20。结果中的最后3列值每一列的

合计等于图7-19所示的相应合计。

**代码7-20 计算每个作者从写（或合写）的每本书中得到的稿酬。结果见图7-20**

```

SELECT
 ta.au_id,
 t.title_id,
 t.pub_id,
 t.sales * t.price * r.royalty_rate * ta.royalty_share AS "Royalty share",
 r.advance * ta.royalty_share AS "Advance share",
 (t.sales * t.price * r.royalty_rate * ta.royalty_share) -
 (r.advance * ta.royalty_share) AS "Due to author"
FROM title_authors ta
INNER JOIN titles t
ON t.title_id = ta.title_id
INNER JOIN royalties r
ON r.title_id = t.title_id
WHERE t.sales IS NOT NULL
ORDER BY ta.au_id ASC, t.title_id ASC;

```

| au_id | title_id | pub_id | Royalty share | Advance share | Due to author |
|-------|----------|--------|---------------|---------------|---------------|
| A01   | T01      | P01    | 622.32        | 10000.00      | -9377.68      |
| A01   | T02      | P03    | 11450.50      | 1000.00       | 10450.50      |
| A01   | T13      | P03    | 18834.32      | 20000.00      | -1165.68      |
| A02   | T06      | P01    | 18066.72      | 20000.00      | -1933.28      |
| A02   | T07      | P03    | 1976138.45    | 500000.00     | 1476138.45    |
| A02   | T12      | P01    | 116911.17     | 50000.00      | 66911.17      |
| A03   | T04      | P04    | 8106.38       | 12000.00      | -3893.62      |
| A03   | T11      | P04    | 15792.90      | 30000.00      | -14207.10     |
| A04   | T04      | P04    | 5404.26       | 8000.00       | -2595.74      |
| A04   | T05      | P04    | 126000.72     | 100000.00     | 26000.72      |
| A04   | T07      | P03    | 1976138.45    | 500000.00     | 1476138.45    |
| A04   | T11      | P04    | 15792.90      | 30000.00      | -14207.10     |
| A05   | T03      | P02    | 71777.77      | 15000.00      | 56777.77      |
| A06   | T08      | P04    | 1638.00       | .00           | 1638.00       |
| A06   | T09      | P04    | 3487.50       | .00           | 3487.50       |
| A06   | T11      | P04    | 21057.20      | 40000.00      | -18942.80     |

图7-20 运行代码7-20的结果

### ✓ 提示

□ 使用**WHERE**语法，代码7-20等价于：

```

SELECT ta.au_id, t.title_id,
 t.pub_id,
 t.sales * t.price *
 r.royalty_rate *
 ta.royalty_share
 AS "Royalty share",

```

```

r.advance * ta.royalty_share
AS "Advance share",
(t.sales * t.price *
r.royalty_rate *
ta.royalty_share) -
(r.advance *
ta.royalty_share)
AS "Due to author"
FROM title_authors ta,
titles t, royalties r
WHERE t.title_id = ta.title_id
AND r.title_id = t.title_id
AND t.sales IS NOT NULL
ORDER BY ta.au_id ASC,
t.title_id ASC;

```

227

□ **DBMS** 为了在Microsoft Access中运行代码7-20，输入：

```

SELECT ta.au_id, t.title_id,
t.pub_id,
t.sales * t.price *
r.royalty_rate *
ta.royalty_share
AS "Royalty share",
r.advance * ta.royalty_share
AS "Advance share",
(t.sales * t.price *
r.royalty_rate *
ta.royalty_share) -
(r.advance * ta.royalty_share)
AS "Due to author"
FROM (titles AS t
INNER JOIN royalties AS r
ON t.title_id = r.title_id)
INNER JOIN title_authors AS ta
ON t.title_id = ta.title_id
WHERE t.sales IS NOT NULL
ORDER BY ta.au_id ASC,
t.title_id ASC;

```

7

代码7-21类似于代码7-20，不同之处在于它加上了一个与表authors的联结来打印作者名字，还包含了WHERE条件来检索稿酬为正数的行，结果见图7-21。

#### 代码7-21 只列出每本书作者得到的正数稿酬

```

SELECT
a.au_id,
a.au_fname,
a.au_lname,
t.title_name,
(t.sales * t.price * r.royalty_rate * ta.royalty_share) -
(r.advance * ta.royalty_share) AS "Due to author"
FROM authors a
INNER JOIN title_authors ta
ON a.au_id = ta.au_id
INNER JOIN titles t
ON t.title_id = ta.title_id

```

```

INNER JOIN royalties r
 ON r.title_id = t.title_id
 WHERE t.sales IS NOT NULL
 AND (t.sales * t.price * r.royalty_rate * ta.royalty_share) -
 (r.advance * ta.royalty_share) > 0
 ORDER BY a.au_id ASC, t.title_id ASC;

```

### ✓ 提示

- 使用**WHERE**语法，代码7-21等价于：

```

SELECT a.au_id, a.au_fname,
 a.au_lname, t.title_name,
 (t.sales * t.price *
 r.royalty_rate *
 ta.royalty_share) -
 (r.advance * ta.royalty_share)
 AS "Due to author"
FROM authors a, title_authors ta,
 titles t, royalties r
 WHERE a.au_id = ta.au_id
 AND t.title_id = ta.title_id
 AND r.title_id = t.title_id
 AND t.sales IS NOT NULL
 AND (t.sales * t.price *
 r.royalty_rate *
 ta.royalty_share) -
 (r.advance *
 ta.royalty_share) > 0
 ORDER BY a.au_id ASC,
 t.title_id ASC;

```

228

- **DBMS** 为了在Microsoft Access运行代码7-21，输入：

```

SELECT a.au_id, a.au_fname,
 a.au_lname, t.title_name,
 (t.sales * t.price *
 r.royalty_rate *
 ta.royalty_share) -
 (r.advance * ta.royalty_share)
 AS "Due to author"
FROM (titles AS t
 INNER JOIN royalties AS r
 ON t.title_id = r.title_id)

INNER JOIN (authors AS a
 INNER JOIN title_authors AS ta
 ON a.au_id = ta.au_id)
 ON t.title_id = ta.title_id
 WHERE t.sales IS NOT NULL
 AND (t.sales * t.price *
 r.royalty_rate *
 ta.royalty_share) -
 (r.advance * ta.royalty_share)
 > 0
 ORDER BY a.au_id ASC,
 t.title_id ASC;

```

229

| au_id | au_fname  | au_lname  | title_name                    | Due to author |
|-------|-----------|-----------|-------------------------------|---------------|
| A01   | Sarah     | Buchman   | 200 Years of German Humor     | 10450.50      |
| A02   | Wendy     | Heydemark | I Blame My Mother             | 1476138.45    |
| A02   | Wendy     | Heydemark | Spontaneous, Not Annoying     | 66911.17      |
| A04   | Klee      | Hull      | Exchange of Platitudes        | 26000.72      |
| A04   | Klee      | Hull      | I Blame My Mother             | 1476138.45    |
| A05   | Christian | Kells     | Ask Your System Administrator | 56777.77      |
| A06   |           | Kellsey   | Just Wait Until After School  | 1638.00       |
| A06   |           | Kellsey   | Kiss My Boo-Boo               | 3487.50       |

图7-21 运行代码7-21的结果

代码7-22使用GROUP BY子句来计算每家出版社支付的总稿酬。聚合函数COUNT()计算每家出版社支付稿酬的图书总数。注意，在这里每个作者的稿酬份额是不需要的，因为不涉及每个作者的稿酬计算。结果见图7-22。结果中最后3列值每一列的总计等于图7-19所示相应的总计。

### 代码7-22 计算每个出版社支付的总稿酬。结果见图7-22

```

SELECT
 t.pub_id,
 COUNT(t.sales) AS "Num books",
 SUM(t.sales * t.price * r.royalty_rate) AS "Total royalties",
 SUM(r.advance) AS "Total advances",
 SUM((t.sales * t.price * r.royalty_rate) - r.advance) AS "Total due to authors"
FROM titles t
INNER JOIN royalties r
 ON r.title_id = t.title_id
WHERE t.sales IS NOT NULL
GROUP BY t.pub_id
ORDER BY t.pub_id ASC;

```

| pub_id | Num books | Total royalties | Total advances | Total due to authors |
|--------|-----------|-----------------|----------------|----------------------|
| P01    | 3         | 135600.21       | 80000.00       | 55600.21             |
| P02    | 1         | 71777.77        | 15000.00       | 56777.77             |
| P03    | 3         | 3982561.72      | 1021000.00     | 2961561.72           |
| P04    | 5         | 197279.85       | 220000.00      | -22720.15            |

图7-22 运行代码7-22的结果

### ✓ 提示

□ 使用WHERE语法，代码7-22等价于：

```

SELECT t.pub_id,
 COUNT(t.sales) AS "Num books",
 SUM(t.sales * t.price *
 r.royalty_rate)

```

```

 AS "Total royalties",
 SUM(r.advance)
 AS "Total advances",
 SUM((t.sales * t.price *
 r.royalty_rate) -
 r.advance)
 AS "Total due to authors"
FROM titles t, royalties r
WHERE r.title_id = t.title_id
 AND t.sales IS NOT NULL
GROUP BY t.pub_id
ORDER BY t.pub_id ASC;

```

230

除了计算每个作者从写(或合写)的书中得到的总稿酬,代码7-23类似于代码7-22,结果见图7-23。

231 结果中最后3列值的合计等于图7-19所示的总计。

### 代码7-23 计算每个作者从写(或合写)的书中得到的总稿酬。结果见图7-23

```

SELECT
 ta.au_id,
 COUNT(sales) AS "Num books",
 SUM(t.sales * t.price * r.royalty_rate * ta.royalty_share) AS "Total royalties share",
 SUM(r.advance * ta.royalty_share) AS "Total advances share",
 SUM((t.sales * t.price * r.royalty_rate * ta.royalty_share) -
 (r.advance * ta.royalty_share)) AS "Total due to author"
FROM title_authors ta
INNER JOIN titles t
 ON t.title_id = ta.title_id
INNER JOIN royalties r
 ON r.title_id = t.title_id
WHERE t.sales IS NOT NULL
GROUP BY ta.au_id
ORDER BY ta.au_id ASC;

```

| au_id | Num books | Total royalties share | Total advances share | Total due to author |
|-------|-----------|-----------------------|----------------------|---------------------|
| A01   | 3         | 30907.14              | 31000.00             | -92.86              |
| A02   | 3         | 2111116.34            | 570000.00            | 1541116.34          |
| A03   | 2         | 23899.28              | 42000.00             | -18100.72           |
| A04   | 4         | 2123336.32            | 638000.00            | 1485336.32          |
| A05   | 1         | 71777.77              | 15000.00             | 56777.77            |
| A06   | 3         | 26182.70              | 40000.00             | -13817.30           |

图7-23 运行代码7-23的结果

#### ✓ 提示

□ 使用WHERE语法,代码7-23等价于:

```

SELECT
 ta.au_id,
 COUNT(sales) AS "Num books",
 SUM(t.sales * t.price *
 r.royalty_rate *

```

```

 ta.royalty_share)
 AS "Total royalties share",
SUM(r.advance *
 ta.royalty_share)
 AS "Total advances share",
SUM((t.sales * t.price *
 r.royalty_rate *
 ta.royalty_share) -
 (r.advance *
 ta.royalty_share))
 AS "Total due to author"
FROM title_authors ta, titles t,
 royalties r
WHERE t.title_id = ta.title_id
 AND r.title_id = t.title_id
 AND t.sales IS NOT NULL
GROUP BY ta.au_id
ORDER BY ta.au_id ASC;

```

**DBMS** 为了在Microsoft Access运行代码7-23，输入:

```

SELECT ta.au_id,
 COUNT(sales) AS "Num books",
 SUM(t.sales * t.price *
 r.royalty_rate *
 ta.royalty_share)
 AS "Total royalties share",
 SUM(r.advance *
 ta.royalty_share)
 AS "Total advances share",
 SUM((t.sales * t.price *
 r.royalty_rate *
 ta.royalty_share) -
 (r.advance *
 ta.royalty_share))
 AS "Total due to author"
FROM (title_authors AS ta
 INNER JOIN titles AS t
 ON t.title_id = ta.title_id)
 INNER JOIN royalties AS r
 ON r.title_id = t.title_id
WHERE t.sales IS NOT NULL
GROUP BY ta.au_id
ORDER BY ta.au_id ASC;

```

7

232

233

代码7-24使用两个列分组来计算每家美国的出版社支付给作者写或合写的书的总稿酬。HAVING条件检索返回拥有正数净稿酬的行，WHERE条件检索美国的作者，结果见图 7-24。

**代码7-24 计算每家美国的出版社支付给作者写（或合写）书的正数净稿酬。结果见图7-24**

```

SELECT
 t.pub_id,
 ta.au_id,
 COUNT(*) AS "Num books",
 SUM(t.sales * t.price * r.royalty_rate * ta.royalty_share) AS "Total royalties share",
 SUM(r.advance * ta.royalty_share) AS "Total advances share",
 SUM((t.sales * t.price * r.royalty_rate * ta.royalty_share) -
 (SUM(r.advance * ta.royalty_share)))
 AS "Total due to author"
 HAVING "Total due to author" > 0
 WHERE ta.au_id IN (SELECT au_id
 FROM title_authors
 WHERE country_id = 1)
 GROUP BY t.pub_id, ta.au_id
 ORDER BY t.pub_id ASC;

```

```

(r.advance * ta.royalty_share)) AS "Total due to author"
FROM title_authors ta
INNER JOIN titles t
 ON t.title_id = ta.title_id
INNER JOIN royalties r
 ON r.title_id = t.title_id
INNER JOIN publishers p
 ON p.pub_id = t.pub_id
WHERE t.sales IS NOT NULL
 AND p.country IN ('USA')
GROUP BY t.pub_id, ta.au_id
HAVING SUM((t.sales * t.price * r.royalty_rate * ta.royalty_share) -
(r.advance * ta.royalty_share)) > 0
ORDER BY t.pub_id ASC, ta.au_id ASC;

```

| pub_id | au_id | Num books | Total royalties share | Total advances share | Total due to author |
|--------|-------|-----------|-----------------------|----------------------|---------------------|
| P01    | A02   | 2         | 134977.89             | 70000.00             | 64977.89            |
| P02    | A05   | 1         | 71777.77              | 15000.00             | 56777.77            |
| P04    | A04   | 3         | 147197.87             | 138000.00            | 9197.87             |

图7-24 运行代码7-24的结果

**✓ 提示**

□ 使用**WHERE**语法，代码7-24等价于：

```

SELECT t.pub_id, ta.au_id,
 COUNT(*) AS "Num books",
 SUM(t.sales * t.price *
 r.royalty_rate *
 ta.royalty_share)
 AS "Total royalties share",
 SUM(r.advance *
 ta.royalty_share)
 AS "Total advances share",
 SUM((t.sales * t.price *
 r.royalty_rate *
 ta.royalty_share) -
 (r.advance *
 ta.royalty_share))
 AS "Total due to author"
FROM title_authors ta, titles t,
 royalties r, publishers p
WHERE t.title_id = ta.title_id
 AND r.title_id = t.title_id
 AND p.pub_id = t.pub_id
 AND t.sales IS NOT NULL
 AND p.country IN ('USA')
GROUP BY t.pub_id, ta.au_id
HAVING SUM((t.sales * t.price *
 r.royalty_rate *
 ta.royalty_share) -
 (r.advance * ta.royalty_share))
 > 0

```

```
ORDER BY t.pub_id ASC,
ta.au_id ASC;
```

□ **DBMS** 为了在Microsoft Access中运行代码7-24，输入：

```
SELECT t.pub_id,
ta.au_id,
COUNT(*) AS "Num books",
SUM(t.sales * t.price *
r.royalty_rate *
ta.royalty_share)
AS "Total royalties share",
SUM(r.advance *
ta.royalty_share)
AS "Total advances share",
SUM((t.sales * t.price *
r.royalty_rate *
ta.royalty_share) -
(r.advance *
ta.royalty_share))
AS "Total due to author"
FROM ((publishers AS p
INNER JOIN titles AS t
ON p.pub_id = t.pub_id)
INNER JOIN royalties AS r
ON t.title_id =
r.title_id)
INNER JOIN title_authors AS ta
ON t.title_id = ta.title_id
WHERE t.sales IS NOT NULL
AND p.country IN ('USA')
GROUP BY t.pub_id, ta.au_id
HAVING SUM((t.sales * t.price *
r.royalty_rate *
ta.royalty_share) -
(r.advance * ta.royalty_share)) > 0
ORDER BY t.pub_id ASC,
ta.au_id ASC;
```

7

234

## 7.8 使用 OUTER JOIN 创建外联结

上一节，学习了两表中至少有一行满足联结条件才返回行的内联结。内联结删除在另一表中没有匹配的行，但外联结至少返回其中一个表的所有行（假定这些行满足WHERE或HAVING查询条件）。

外联结在回答包含缺失数量的问题时是有用的，例如，没有写书的作者或没有学生注册报名的课程。要对一个表所有行与另一个表匹配行创建报告，外联结也是有用的。例如，销售量超过某个给定数字的所有作者和所有图书；或订购的所有产品，包括没有订单的产品。

不同于其他联结，在外联结中指明的表的顺序是重要的，两个联结操作数被称作左表和右表。外联结有以下3类。

□ 左外联结。左外联结的结果包括LEFT OUTER JOIN子句中指明的左表中的所有行，不止是联结列匹配的行。如果左表中的行没有匹配右表中的行，结果中相关行对于SELECT子句中来自右表的列显示空值。

- 右外联结。右外联结与左外联结相反，返回来自右表的所有行。如果右表中没有行与左表中的行相匹配，左表返回空值。
- 全外联结。全外联结是左外联结和右外联结的结合，返回左表和右表中的所有行。如果行在另一表中没有匹配，另一表中SELECT子句的列显示空值。如果有匹配，结果中的整个行包含两个表的数据值。

总而言之，左外联结引用左表检索所有行，右外联结引用右表检索所有行，全外联结检索两个表的所有行。在所有这些情况中，没有匹配的行用空值填充。在结果中，我们无法判别空值（如果有）是最初被输入表的还是外联结操作插入的。记住，条件`NULL = NULL`和`NULL = any_value`是不可知且无匹配的，见3.14节。

235

### ⇒ 创建左外联结

输入：

```
SELECT columns
 FROM left_table
 LEFT [OUTER] JOIN right_table
 ON join_conditions
```

*columns*是来自*left\_table*或*right\_table*的一个或多个用逗号分隔的表达式或列名。*left\_table*和*right\_table*是联结表的名称。如果表有名称相同的列，用表名限定列名。*join\_conditions*指明一个或几个被每对联结行判断的联结条件。联结条件采用这种形式：

```
[left_table.]column op
 → [right_table.]column
```

*op*通常是`=`，但也可以是其他比较操作符：`=`、`<`、`<=`、`>`或`>=`（见第4章表4-2）。可以使用`AND`或`OR`结合多个联结条件，参见4.6节。

关键字`OUTER`是可选的。

### ⇒ 创建右外联结

输入：

```
SELECT columns
 FROM left_table
 RIGHT [OUTER] JOIN right_table
 ON join_conditions
```

*columns*、*left\_table*、*right\_table*和*join\_conditions*与7.8.1节中的参数具有相同含义。

关键字`OUTER`是可选的。

### ⇒ 创建全外联结

输入：

```
SELECT columns
 FROM left_table
 FULL [OUTER] JOIN right_table
 ON join_conditions
```

*columns*、*left\_table*、*right\_table*和*join\_conditions*与7.8.1节有相同的含义。

关键字`OUTER`是可选的。

✓ 提示

- 对于外联结，因为 JOIN 语法更精确，只要有可能就应该使用 JOIN 而不用 WHERE。SQL 对外联结缺少标准化的 WHERE 语法，于是语法因 DBMS 而异。DBMS 也可以在 WHERE 外联结加上在 JOIN 外联结不存在的限制，见本节后面 DBMS 提示的具体内容。
- 注意外联结中表出现的顺序。不像其他联结，外联结中表的顺序不可互换，也就是说，外联结查询的结果依赖于表被分组和联结（关联）的顺序。以下两个三表内联结是等价的（除了结果中列的顺序）。

```
SELECT * FROM table1
 INNER JOIN table2
 INNER JOIN table3
```

和

```
SELECT * FROM table2
 INNER JOIN table3
 INNER JOIN table1
```

但以下三表外联结产生了不同的结果。

```
SELECT * FROM table1
 LEFT OUTER JOIN table2
 LEFT OUTER JOIN table3
```

和

```
SELECT * FROM table2
 LEFT OUTER JOIN table3
 LEFT OUTER JOIN table1
```

- 在 SQL:2003 标准之前，SQL 有联合联结（union join）。它并非真正返回两个表的匹配行，而是返回删除匹配行的全外联结。在联合联结中每行是一个表的列与另一个表为空值的列。语句 `t1 UNION JOIN t2` 的结果看上去像右表：

|                      |                      |
|----------------------|----------------------|
| <code>t1</code> 所有的行 | 空值                   |
| 空值                   | <code>t2</code> 所有的行 |

UNION JOIN 很少实际应用，很多 DBMS 不支持。可以使用全外联结来模仿联合联结。

`t1 UNION JOIN t2`

等价于：

`t1 FULL OUTER JOIN t2 ON 1 = 2`

`t1` 和 `t2` 是表，`1 = 2` 表示永远为假的条件。注意，UNION JOIN 不同于 UNION。UNION 是集合操作，而不是一个联结，参见 9.1 节。

- **DBMS** Microsoft SQL Server 支持标准的 OUTER JOIN 语法，但在使用 WHERE 语法创建外联结时也使用（非标准的）外联结操作符\*。在比较操作符的左边或右边加上\* 来创建左外联结或右外联结。对于外联结，WHERE 语法没有 OUTER JOIN 精确，可能产生模糊不清的查询。未来的 SQL Server 版本也许不再支持\*= 和 \*= 操作符。

Oracle 8i 和之前的版本不支持 JOIN 语法，应使用 WHERE 联结。Oracle 9i 和之后的版本支持标准的 OUTER JOIN 语法。在 WHERE 语法中，Oracle 使用（非标准的）外联结操作符(+) 来创建外联结。在必须扩展的表后加(+) 会用空值填充，见本节后面的例子。

236

7

237

下面来看4个例子。代码7-25和图7-25显示了每个作者和出版社所在的城市。

代码7-26显示了表authors和publishers以他们的city列为条件的内联结。图7-26的结果列出了居住在出版社所在城市的作者。可以将内联结的结果和后面3个外联结例子的结果进行比较。

#### 代码7-25 列出有作者和出版社的城市。结果见图7-25

```
SELECT a.au_fname, a.au_lname, a.city
FROM authors a;

SELECT p.pub_name, p.city
FROM publishers p;
```

#### 代码7-26 列出所居住在出版社所在城市的作者。结果见图7-26

```
SELECT a.au_fname, a.au_lname, p.pub_name
FROM authors a
INNER JOIN publishers p
ON a.city = p.city;
```

238

**代码7-27** 这个左外联结包含了表authors的所有行，不管在表publishers是否有city列匹配城市列。结果见图7-27

```
SELECT a.au_fname, a.au_lname, p.pub_name
FROM authors a
LEFT OUTER JOIN publishers p
ON a.city = p.city
ORDER BY p.pub_name ASC,
a.au_lname ASC, a.au_fname ASC;
```

代码7-27使用了左外联结来包括所有作者，不管是否有出版社和他在同一城市，结果见图7-27。

#### ✓ 提示

使用WHERE语法，代码7-26等价于：

```
SELECT a.au_fname, a.au_lname,
p.pub_name
FROM authors a, publishers p
WHERE a.city = p.city;
```

#### ✓ 提示

- **DBMS** 为了在Microsoft SQL Server运行代码7-27，使用WHERE语法，输入：

```
SELECT a.au_fname, a.au_lname,
p.pub_name
```

| au_fname            | au_lname      | city          |
|---------------------|---------------|---------------|
| Sarah               | Buchman       | Bronx         |
| Wendy               | Heydemark     | Boulder       |
| Hallie              | Hull          | San Francisco |
| Klee                | Hull          | San Francisco |
| Christian           | Kells         | New York      |
|                     | Kellsey       | Palo Alto     |
| Paddy               | O'Furniture   | Sarasota      |
| pub_name            | city          |               |
| Abatis Publishers   | New York      |               |
| Core Dump Books     | San Francisco |               |
| Schadenfreude Press | Hamburg       |               |
| Tenterhooks Press   | Berkeley      |               |

图7-25 运行代码7-25的结果

| au_fname  | au_lname | pub_name          |
|-----------|----------|-------------------|
| Hallie    | Hull     | Core Dump Books   |
| Klee      | Hull     | Core Dump Books   |
| Christian | Kells    | Abatis Publishers |

图7-26 运行代码7-26的结果

| au_fname  | au_lname    | pub_name          |
|-----------|-------------|-------------------|
| Sarah     | Buchman     | NULL              |
| Wendy     | Heydemark   | NULL              |
|           | Kellsey     | NULL              |
| Paddy     | O'Furniture | NULL              |
| Christian | Kells       | Abatis Publishers |
| Hallie    | Hull        | Core Dump Books   |
| Klee      | Hull        | Core Dump Books   |

图7-27 运行代码7-27的结果。注意，这里有4个作者无匹配数据，这些行在列pub\_name包含了空值

```
FROM authors a, publishers p
WHERE a.city *= p.city
ORDER BY p.pub_name ASC,
a.au_lname ASC, a.au_fname ASC;
```

为了在Oracle 8i中运行代码7-27，输入：

```
SELECT a.au_fname, a.au_lname,
p.pub_name
FROM authors a, publishers p
WHERE a.city = p.city (+)
ORDER BY p.pub_name ASC,
a.au_lname ASC, a.au_fname ASC;
```

239

7

代码7-28 使用结果中包含所有出版社的右外联结，不管作者居住城市是否有出版社，结果见图7-28。

**代码7-28** 该右外联结在结果中包含表publishers的所有行，不管表authors中是否有城市列匹配

```
SELECT a.au_fname, a.au_lname, p.pub_name
FROM authors a
RIGHT OUTER JOIN publishers p
ON a.city = p.city
ORDER BY p.pub_name ASC,
a.au_lname ASC, a.au_fname ASC;
```

✓ 提示

□ **DBMS** 为了在Microsoft SQL Server中运行代码7-28，使用WHERE语法，输入：

```
SELECT a.au_fname, a.au_lname,
p.pub_name
FROM authors a, publishers p
WHERE a.city =* p.city
ORDER BY p.pub_name ASC,
a.au_lname ASC, a.au_fname ASC;
```

为了在Oracle 8i中运行代码7-28，输入：

```
SELECT a.au_fname, a.au_lname,
p.pub_name
FROM authors a, publishers p
WHERE a.city (+) = p.city
ORDER BY p.pub_name ASC,
a.au_lname ASC, a.au_fname ASC;
```

代码7-29 使用全外联结在结果中包含所有出版社和所有作者，不管他们是否在同一城市，结果见图7-29。

**代码7-29** 这个全外联结在结果中包含表authors和publishers的所有行，不论是否匹配city列。结果见图7-29

```
SELECT a.au_fname, a.au_lname, p.pub_name
```

| au_fname  | au_lname | pub_name            |
|-----------|----------|---------------------|
| Christian | Kells    | Abatis Publishers   |
| Hallie    | Hull     | Core Dump Books     |
| Klee      | Hull     | Core Dump Books     |
| NULL      | NULL     | Schadenfreude Press |
| NULL      | NULL     | Tenterhooks Press   |

图7-28 运行代码7-28的结果。注意，列出的两个出版社没有匹配数据，列au\_fname和au\_lname的相应行包含空值

| au_fname  | au_lname    | pub_name            |
|-----------|-------------|---------------------|
| Sarah     | Buchman     | NULL                |
| Wendy     | Heydermark  | NULL                |
|           | Kelsey      | NULL                |
| Paddy     | O'Furniture | NULL                |
| Christian | Kells       | Abatis Publishers   |
| Hallie    | Hull        | Core Dump Books     |
| Klee      | Hull        | Core Dump Books     |
| NULL      | NULL        | Schadenfreude Press |
| NULL      | NULL        | Tenterhooks Press   |

图7-29 运行代码7-29的结果。结果包含9行，其中有4行的作者在publishers表中没有匹配行，有3行在同一城市有作者和出版社，有2行的出版社在authors表没有匹配行

```
FROM authors a
FULL OUTER JOIN publishers p
ON a.city = p.city
ORDER BY p.pub_name ASC,
a.au_lname ASC, a.au_fname ASC;
```

✓ 提示

- **DBMS** 在Microsoft SQL Server中，不能在比较操作符两边加上\*操作符来创建一个全外联结，而是要用一个左外联结和一个右外联结的合并来构成，参见9.1节。为了运行代码7-29，使用WHERE语法，输入：

```
SELECT a.au_fname, a.au_lname,
p.pub_name
FROM authors a, publishers p
WHERE a.city *= p.city
UNION ALL
SELECT a.au_fname, a.au_lname,
p.pub_name
FROM authors a, publishers p
WHERE a.city =* p.city
AND a.city IS NULL;
```

241

在Oracle中，不能在比较操作符的两边加上(+)操作符构成全外联结，而要合并一个左外联结和一个右外联结构成，见9.1节。为了在Oracle 8i运行代码7-29，输入：

```
SELECT a.au_fname, a.au_lname,
p.pub_name
FROM authors a, publishers p
WHERE a.city = p.city (+)
UNION ALL
SELECT a.au_fname, a.au_lname,
p.pub_name
FROM authors a, publishers p
WHERE a.city (+) = p.city
AND a.city IS NULL;
```

Microsoft Access和MySQL不支持全外联结，但可以采取合并左外联结和右外联结来实现，见9.1节。在以下例子中，第一个UNION表是返回表authors的所有行，以及匹配表publishers的列city的左外联结。第二个UNION表是返回没有匹配表publishers的右外联结。为了运行代码7-29，输入：

```
SELECT a.au_fname, a.au_lname,
p.pub_name
FROM authors a
LEFT OUTER JOIN publishers p
ON a.city = p.city
UNION ALL
SELECT a.au_fname, a.au_lname,
p.pub_name
FROM authors a
RIGHT OUTER JOIN publishers p
ON a.city = p.city
WHERE a.city IS NULL;
```

242

代码7-30使用了左外联结列出每个作者写或合写的图书数量。结果见图7-30。注意，与7.7节的代码7-11对比，作者A07(Paddy O'Furniture)尽管没有写书也出现在结果中。

**代码7-30** 列出每个作者写或合写的图书数量，包含没有写过书的作者。结果见图7-30

```
SELECT
 a.au_id,
 COUNT(ta.title_id) AS "Num books"
FROM authors a
LEFT OUTER JOIN title_authors ta
 ON a.au_id = ta.au_id
GROUP BY a.au_id
ORDER BY a.au_id ASC;
```

✓ 提示

- **DBMS** 为了在Oracle 8i中运行代码7-30，输入：

```
SELECT a.au_id,
 COUNT(ta.title_id)
 AS "Num books"
 FROM authors a, title_authors ta
 WHERE a.au_id = ta.au_id (+)
 GROUP BY a.au_id
 ORDER BY a.au_id ASC;
```

| au_id | Num books |
|-------|-----------|
| A01   | 3         |
| A02   | 4         |
| A03   | 2         |
| A04   | 4         |
| A05   | 1         |
| A06   | 3         |
| A07   | 0         |

图7-30 运行代码7-30的结果

243

7

代码7-31使用WHERE条件来测试空值，只列出已经写过书的作者，结果见图7-31。

**代码7-31** 列出没有写（或合写）过书的作者。结果见图7-31

```
SELECT a.au_id, a.au_fname, a.au_lname
 FROM authors a
 LEFT OUTER JOIN title_authors ta
 ON a.au_id = ta.au_id
 WHERE ta.au_id IS NULL;
```

✓ 提示

- **DBMS** 为了在Oracle 8i中运行代码7-31，输入：

```
SELECT a.au_id, a.au_fname,
 a.au_lname
 FROM authors a, title_authors ta
 WHERE a.au_id = ta.au_id (+)
 AND ta.au_id IS NULL;
```

| au_id | au_fname | au_lname    |
|-------|----------|-------------|
| A07   | Paddy    | O'Furniture |

图7-31 运行代码7-31的结果

244

245

代码7-32结合内联结和左外联结来列出所有作者和销量超过100 000本的图书。在这个例子中，首先创建了过滤过的INNER JOIN结果，然后将它和表authors外联结，这样就得到所需的所有行，结果见图7-32。

**代码7-32** 列出所有作者和销量超过100 000本的图书。结果见图7-32

```
SELECT a.au_id, a.au_fname, a.au_lname,
 tta.title_id, tta.title_name, tta.sales
 FROM authors a
 LEFT OUTER JOIN
 (SELECT ta.au_id, t.title_id,
```

```
t.title_name, t.sales
FROM title_authors ta
INNER JOIN titles t
 ON t.title_id = ta.title_id
WHERE sales > 100000) tta
ON a.au_id = tta.au_id
ORDER BY a.au_id ASC, tta.title_id ASC;
```

|     | au_id     | au_fname    | au_lname | title_id                  | title_name             | sales   |
|-----|-----------|-------------|----------|---------------------------|------------------------|---------|
| A01 | Sarah     | Buchman     |          | NULL                      | NULL                   | NULL    |
| A02 | Wendy     | Heydemark   | T07      | I Blame My Mother         | 1500200                |         |
| A02 | Wendy     | Heydemark   | T12      | Spontaneous, Not Annoying | 100001                 |         |
| A03 | Hallie    | Hull        |          | NULL                      | NULL                   | NULL    |
| A04 | Klee      | Hull        |          | T05                       | Exchange of Platitudes | 201440  |
| A04 | Klee      | Hull        |          | T07                       | I Blame My Mother      | 1500200 |
| A05 | Christian | Kells       |          | NULL                      | NULL                   | NULL    |
| A06 |           | Kellsey     |          | NULL                      | NULL                   | NULL    |
| A07 | Paddy     | O'Furniture |          | NULL                      | NULL                   | NULL    |

图7-32 运行代码7-32的结果

## ✓ 提示

- **DBMS** 为了在Oracle 8i中运行代码7-32，输入：

```
SELECT a.au_id, a.au_fname,
 a.au_lname,
 tta.title_id, tta.title_name,
 tta.sales
 FROM authors a,
 (SELECT ta.au_id, t.title_id,
 t.title_name, t.sales
 FROM title_authors ta,
 titles t
 WHERE t.title_id =
 ta.title_id
 AND sales > 100000) tta
 WHERE a.au_id = tta.au_id (+)
 ORDER BY a.au_id ASC,
 tta.title_id ASC;
```

MySQL 4.1及之后版本可以运行代码7-32，但之前版本不支持子查询，见8.1节的DBMS提示。对于复杂的查询，经常要创建临时表来存储子查询结果，见11.10节。为了在MySQL 4.0及之前版本运行代码7-32，输入：

```
CREATE TEMPORARY TABLE tta
SELECT ta.au_id, t.title_id,
 t.title_name, t.sales
 FROM title_authors ta
INNER JOIN titles t
 ON t.title_id = ta.title_id
 WHERE sales > 100000;
```

```

SELECT a.au_id, a.au_fname,
 a.au_lname, tta.title_id,
 tta.title_name, tta.sales
 FROM authors a
 LEFT OUTER JOIN tta
 ON a.au_id = tta.au_id
 ORDER BY a.au_id ASC,
 tta.title_id ASC;

DROP TABLE tta;

```

## 7.9 创建自联结

246

自联结是一个表和自身的联结，并通过比较同一个表的一列或多列值，从表中检索行的常规SQL联结。自联结经常被使用在有反身联系（reflexive relationship）的表。反身联系是指主键/外键同为一个表中的列或列的组合。要了解更多有关键的知识，见2.2节和2.3节。假定有表employees：

| emp_id | emp_name          | boss_id |
|--------|-------------------|---------|
| E01    | Lord Copper       | NULL    |
| E02    | Jocelyn Hitchcock | E01     |
| E03    | Mr. Salter        | E01     |
| E04    | William Boot      | E03     |
| E05    | Mr. Corker        | E03     |

`emp_id`是唯一标识雇员的主键，`boss_id`是标识雇员管理者的雇员ID。每一个管理者也是一个雇员，于是为了保证添加到表的每个管理者ID匹配已经存在的雇员ID，`boss_id`被定义为`emp_id`的外键。

代码7-33使用反身联系来比较表中的行，并检索每个雇员的管理者的名字，结果见图7-33。（不必只是为了得到管理者ID而创建联结。）结果见图7-33。

7

**代码7-33 列出每个雇员和他的经理的名字。结果见图7-33**

```

SELECT
 e1.emp_name AS "Employee name",
 e2.emp_name AS "Boss name"
 FROM employees e1
 INNER JOIN employees e2
 ON e1.boss_id = e2.emp_id;

```

同一个表(`employees`)在代码7-33中出现了两次。在联结条件`e1.boss_id = e2.emp_id`中分别使用两个别名(`e1`和`e2`)来限定列名。

像任何联结一样，自联结需要两个表，但不是向联结添加另一个表，而是添加同一个表的另一个实例。这样，就可以比较第一个实例中的列和第二个实例中的列。像任何联结一样，DBMS组合并返回满足联结条件的行。不需要创建表的另一个副本，而是让表和自身联结，但是将它们想象成两个表就很容易理解。

| Employee name     | Boss name   |
|-------------------|-------------|
| Jocelyn Hitchcock | Lord Copper |
| Mr. Salter        | Lord Copper |
| William Boot      | Mr. Salter  |
| Mr. Corker        | Mr. Salter  |

图7-33 运行代码7-33的结果。注意，Lord Copper没有管理者。因为他的boss\_id为空值不满足联结条件，所以结果中不包含

⇒ 创建自联结

247

输入：

```
SELECT columns
 FROM table [AS] alias1
 INNER JOIN table [AS] alias2
 ON join_conditions
```

*columns*是来自*table*的一个或多个逗号分隔的表达式或列名。*alias1*和*alias2*是在*join\_conditions*中引用*table*的不同别名，参见7.2节。

*join\_conditions*指明一个或多个用于每对联结行判断的联结条件。联结条件采用以下形式：

*alias1.column op alias2.column*

*op*是任何操作符=、<>、<、<=、>或>=（见第4章中的表4-2）。可以使用AND或OR组合多个联结条件，见4.6节。

#### ✓ 提示

□ 甚至可以联结没有反身联系的表与自身。普通自联结是比较第一个实例中的列与第二个实例中的相同列。联结条件比较该列与另一列的值，如本节后面给出的例子所示。

□ 参见15.16节。

□ **DBMS** Oracle 8i和之前版本不支持JOIN语法，应该使用WHERE联结。Oracle 9i和之后版本支持JOIN语法。

代码7-34使用WHERE查询条件和从state列到自身的自联结找到所有和作者A04 (Klee Hull) 住在同一州的作者，结果见图7-34。

代码7-34 列出与作者A04 (Klee Hull) 住在同一个州的作者。结果见图7-34

```
SELECT a1.au_id, a1.au_fname,
 a1.au_lname, a1.state
 FROM authors a1
 INNER JOIN authors a2
 ON a1.state = a2.state
 WHERE a2.au_id = 'A04';
```

#### ✓ 提示

□ 使用WHERE语法，代码7-34等价于：

```
SELECT a1.au_id, a1.au_fname,
 a1.au_lname, a1.state
 FROM authors a1, authors a2
 WHERE a1.state = a2.state
 AND a2.au_id = 'A04';
```

□ 自联结经常被改写成子查询（见第8章）。使用子查询，代码7-34等价于：

```
SELECT au_id, au_fname,
 au_lname, state
 FROM authors
 WHERE state IN
 (SELECT state
 FROM authors
 WHERE au_id = 'A04');
```

| au_id | au_fname | au_lname | state |
|-------|----------|----------|-------|
| A03   | Hallie   | Hull     | CA    |
| A04   | Klee     | Hull     | CA    |
| A06   |          | Kellsey  | CA    |

图7-34 运行代码7-34的结果

248

249

代码7-35对于每本传记列出销量超过它的其他传记。注意，因为联结条件认为*type*列是两个独立的列，WHERE查询条件对于表t1和t2需要*type = 'biography'*，结果见图7-35。

**代码7-35** 对于每本传记列出销售超过它的其他传记的ID和销量。结果见图7-35

```

SELECT t1.title_id, t1.sales,
 t2.title_id AS "Better seller",
 t2.sales AS "Higher sales"
 FROM titles t1
 INNER JOIN titles t2
 ON t1.sales < t2.sales
 WHERE t1.type = 'biography'
 AND t2.type = 'biography'
 ORDER BY t1.title_id ASC, t2.sales ASC;

```

|     | title_id | sales | Better seller | Higher sales |
|-----|----------|-------|---------------|--------------|
| T06 | 11320    | T12   |               | 100001       |
| T06 | 11320    | T07   |               | 1500200      |
| T12 | 100001   | T07   |               | 1500200      |

图7-35 运行代码7-35的结果

**✓ 提示**

- 使用WHERE语法，代码7-35等价于：

```

SELECT t1.title_id, t1.sales,
 t2.title_id AS "Better seller",
 t2.sales AS "Higher sales"
 FROM titles t1, titles t2
 WHERE t1.sales < t2.sales
 AND t1.type = 'biography'
 AND t2.type = 'biography'
 ORDER BY t1.title_id ASC,
 t2.sales ASC;

```

代码7-36是找到同住在New York州的作者的自联结，结果见图7-36。

7

250

**代码7-36** 列出同住在纽约州的作者。结果见图7-36

```

SELECT
 a1.au_fname, a1.au_lname,
 a2.au_fname, a2.au_lname
 FROM authors a1
 INNER JOIN authors a2
 ON a1.state = a2.state
 WHERE a1.state = 'NY'
 ORDER BY a1.au_id ASC, a2.au_id ASC;

```

| au_fname  | au_lname | au_fname  | au_lname |
|-----------|----------|-----------|----------|
| Sarah     | Buchman  | Sarah     | Buchman  |
| Sarah     | Buchman  | Christian | Kells    |
| Christian | Kells    | Sarah     | Buchman  |
| Christian | Kells    | Christian | Kells    |

图7-36 运行代码7-36的结果

**✓ 提示**

- 使用WHERE语法，代码7-36等价于：

```

SELECT
 a1.au_fname, a1.au_lname,
 a2.au_fname, a2.au_lname
 FROM authors a1, authors a2
 WHERE a1.state = a2.state
 AND a1.state = 'NY'
 ORDER BY a1.au_id ASC,
 a2.au_id ASC;

```

图7-36中第一行和第四行是不必要的，因为它们无非是说明Sarah Buchman和Sarah Buchman居住在同一个州，Christian Kells也和他自己住在相同的州。创建一个联结条件只保留两个不同作者的行（代码7-37和图7-37）。

251

**代码7-37** 列出同住在纽约州的不同作者。结果见图7-37

```

SELECT
 a1.au_fname, a1.au_lname,
 a2.au_fname, a2.au_lname
FROM authors a1
INNER JOIN authors a2
 ON a1.state = a2.state
 AND a1.au_id <> a2.au_id
WHERE a1.state = 'NY'
ORDER BY a1.au_id ASC, a2.au_id ASC;

```

| au_fname  | au_lname | au_fname  | au_lname |
|-----------|----------|-----------|----------|
| Sarah     | Buchman  | Christian | Kells    |
| Christian | Kells    | Sarah     | Buchman  |

图7-37 运行代码7-37的结果

**✓ 提示**

- 使用**WHERE**语法，代码7-37等价于：

```

SELECT
 a1.au_fname, a1.au_lname,
 a2.au_fname, a2.au_lname
FROM authors a1, authors a2
WHERE a1.state = a2.state
 AND a1.au_id <> a2.au_id
 AND a1.state = 'NY'
ORDER BY a1.au_id ASC,
 a2.au_id ASC;

```

图7-37的结果仍然不是很理想，因为这两个结果行内容重复。第一行说明Sarah Buchman和Christian Kells住在同一个州，第二行给出了相同的信息。为了消除冗余，改变第二个联结条件的比较操作符，将不等改为小于（见代码7-38和图7-38）。

**代码7-38** 无冗余地列出同住在纽约州的不同作者。结果见图7-38

252

```

SELECT
 a1.au_fname, a1.au_lname,
 a2.au_fname, a2.au_lname
FROM authors a1
INNER JOIN authors a2
 ON a1.state = a2.state
 AND a1.au_id < a2.au_id
WHERE a1.state = 'NY'
ORDER BY a1.au_id ASC, a2.au_id ASC;

```

| au_fname | au_lname | au_fname  | au_lname |
|----------|----------|-----------|----------|
| Sarah    | Buchman  | Christian | Kells    |

图7-38 运行代码7-38的结果

**✓ 提示**

- 使用**WHERE**语法，代码7-38等价于：

```

SELECT
 a1.au_fname, a1.au_lname,
 a2.au_fname, a2.au_lname
FROM authors a1, authors a2
WHERE a1.state = a2.state
 AND a1.au_id < a2.au_id
 AND a1.state = 'NY'
ORDER BY a1.au_id ASC,
 a2.au_id ASC;

```

**到**这里，本书只是使用SELECT语句从一个或几个表中检索数据。本章将介绍能从其他的查询结果中检索或修改数据的嵌套查询。*subquery*或*subselect*是嵌套在另一SQL语句中的SELECT语句。可以将子查询嵌套在：

- SELECT语句的SELECT、FROM、WHERE或HAVING子句中。
- 其他子查询中。
- INSERT、UPDATE或DELETE语句中。

大体上，子查询可以用在允许有表达式的任何地方，但具体的各个DBMS可能有不同限制。本章介绍嵌套在SELECT语句或其他子查询中的子查询，第10章介绍嵌套在INSERT、UPDATE和DELETE语句中的子查询。

253

8

## 8.1 理解子查询

本节给出一些术语的定义，并通过包含简单子查询的SELECT语句的例子介绍子查询。后面的小节解释各类子查询及它们的语法和语义。

如果要列出传记出版社的名称，最直接的做法是用两个查询：第一个查询检索出所有传记出版社的ID（代码8-1和图8-1），第二个查询使用第一个查询的结果列出出版社的名称（代码8-2和图8-2）。

**代码8-1** 列出传记出版社。结果见图8-1

```
SELECT pub_id
 FROM titles
 WHERE type = 'biography';
```

| pub_id |
|--------|
| -----  |
| P01    |
| P03    |
| P01    |
| P01    |

图8-1 运行代码8-1的结果。可以在代码8-1的SELECT子句上加入DISTINCT，让每个出版社只出现一次，参见4.3节

**代码8-2** 这个查询使用了代码8-1的结果列出传记出版社的名称。结果见图8-2

```
SELECT pub_name
 FROM publishers
 WHERE pub_id IN ('P01', 'P03');
```

更好的方法是用内联结(代码8-3和图8-3), 参见7.7节。

还有一种方法是使用子查询(代码8-4和图8-4)。代码8-4中的子查询用粗代码体显示出来。子查询也被称作内查询, 包含子查询的语句被称作外查询。或者说, 被包含的子查询是外部查询的内部查询。因为一个子查询可以被嵌套进另一个子查询, 所以对于多重嵌套子查询语句而言, 内部和外部是相对的。

**代码8-3** 使用内联结列出传记出版社的名称。结果见图8-3

```
SELECT DISTINCT pub_name
 FROM publishers p
 INNER JOIN titles t
 ON p.pub_id = t.pub_id
 WHERE t.type = 'biography';
```

| pub_name            |
|---------------------|
| Abatis Publishers   |
| Schadenfreude Press |

图8-2 运行代码8-2的结果

| pub_name            |
|---------------------|
| Abatis Publishers   |
| Schadenfreude Press |

图8-3 运行代码8-3的结果

**代码8-4** 通过子查询列出传记出版社的名称。结果见图8-4

```
SELECT pub_name
 FROM publishers
 WHERE pub_id IN
 (SELECT pub_id
 FROM titles
 WHERE type = 'biography');
```

| pub_name            |
|---------------------|
| Abatis Publishers   |
| Schadenfreude Press |

图8-4 运行代码8-4的结果

在8.4节将会解释DBMS如何执行子查询, 但现在所要理解的就是代码8-4中DBMS首先处理内部查询(粗代码体部分), 然后基于中间结果来执行外部查询(代码体部分), 得到最终结果。关键字IN引出子查询来测试成员资格, 就像4.9节中的IN那样运行。注意代码8-4中的内部查询和代码8-1中的查询相同, 外部查询和代码8-2中的查询相同。

### ✓ 提示

- 子查询这个术语有时用来指包含了一个或多个子查询的整个SQL语句。为了避免这种混淆, 本书没有这样使用这个术语。
- MySQL 4.1及之后的版本支持子查询, 但之前的版本不支持。如果使用的是MySQL 4.0或之前版本, 就无法运行本章的例子, 但可以选择以下方法(按先后顺序列出)。
  - 升级到MySQL的最新版本([www.mysql.com](http://www.mysql.com))。
  - 将子查询改写为联结(见8.3节)。
  - 创建一个临时表来保存子查询的结果(参见11.10节和7.8节DBMS提示中的临时表例子, 代码7-32)。
  - 在过程化宿主语言(比如PHP或Java)中模拟子查询(本书不包含这些内容)。

## 8.2 子查询语法

除了以下的不同点，子查询的语法和普通的SELECT语句相同（见第4章至第7章）。

- 可以在SELECT、FROM、WHERE、HAVING子句或另外的子查询中嵌套子查询。
- 子查询应包含在括号中。
- 不要用分号结束子查询（包含子查询的语句仍需用分号结束）。
- 不要在子查询中使用ORDER BY子句（子查询返回的中间结果是看不到的，对子查询排序没有意义）。
- 子查询是单个SELECT语句（不能使用UNION连接多个SELECT语句作为子查询）。
- 子查询可以使用它自身的FROM子句或外部查询的FROM子句中表的列。
- 如果表只出现在内部查询，没有出现在外部查询中，那么在最终结果中（或者说，在外部查询的SELECT子句中）无法包含这个表的列。
- 依据使用的具体情况，可能要求子查询只返回有限的行或列。SQL标准依据子查询返回的行或列对其分类（表8-1）。在任何情况下，子查询都可以返回一个空表（零行）。

在实际工作中，子查询通常出现在如下形式的WHERE子句中。

- WHERE *test\_expr op (subquery)*
- WHERE *test\_expr [NOT] IN (subquery)*
- WHERE *test\_expr op ALL (subquery)*
- WHERE *test\_expr op ANY (subquery)*
- WHERE [NOT] EXISTS (*subquery*)

*test\_expr*是字面量、列名、表达式或标量子查询，*op*是比较操作符（=、<>、<、<=、>或>=），*Subquery*是简单子查询或相关子查询。在本章后面将介绍这些形式的子查询。它们在HAVING子句中也可以使用。

### ✓ 提示

- DBMS** SQL标准没有规定子查询嵌套的最大数量，最高限制由具体的DBMS来决定。人们通常不能构造出达到内置限制的嵌套查询，例如Microsoft SQL Server允许32层嵌套。

8

256

表8-1 子查询结果的大小

| 子查询   | 行  | 列  |
|-------|----|----|
| 标量子查询 | 1  | 1  |
| 行子查询  | 1  | ≥1 |
| 表子查询  | ≥1 | ≥1 |

## 8.3 子查询和联结

在8.1节中，代码8-3和代码8-4显示了两个等价的查询：一个使用联结，而另一个使用子查询。某些子查询可以用联结替换。事实上，子查询是不使用联结将一个表与另一个表建立关联的方法。

因为子查询难于使用和查错，有人更愿意使用联结，但有些问题只能通过子查询来解决。在既可以使用子查询，又可以使用联结的情况下，可以在DBMS上测试二者的性能是否有差异。例如，查询：

```
SELECT MAX(table1.col1)
 FROM table1
 WHERE table1.col1 IN
```

```
(SELECT table2.col1
 FROM table2);
```

通常会比下面的快：

```
SELECT MAX(table1.col1)
 FROM table1
 INNER JOIN table2
 ON table1.col1 = table2.col1;
```

257

要了解更多相关知识，参见8.13节。

下面这两个语句是等价的，一个用IN子查询：

```
SELECT *
 FROM table1
 WHERE id IN
 (SELECT id FROM table2);
```

一个用内联结：

```
SELECT DISTINCT table1.*
 FROM table1
 INNER JOIN table2
 ON table1.id = table2.id;
```

258

具体例子见代码8-5a、代码8-5b及图8-5。

**代码8-5a** 这个语句使用子查询列出那些居住在有出版社的城市的作者。结果见图8-5

```
SELECT au_id, city
 FROM authors
 WHERE city IN
 (SELECT city FROM publishers);
```

**代码8-5b** 这个语句等价于代码8-5a，但使用内联结而不是子查询。结果见图8-5

```
SELECT DISTINCT a.au_id, a.city
 FROM authors a
 INNER JOIN publishers p
 ON a.city = p.city;
```

以下3条语句是等价的，一个用NOT IN子查询：

```
SELECT *
 FROM table1
 WHERE id NOT IN
 (SELECT id FROM table2);
```

另一个用NOT EXISTS子查询：

```
SELECT *
 FROM table1
 WHERE NOT EXISTS
 (SELECT *
 FROM table2
 WHERE table1.id = table2.id);
```

| au_id | city          |
|-------|---------------|
| A03   | San Francisco |
| A04   | San Francisco |
| A05   | New York      |

图8-5 运行代码8-5a和代码8-5b的结果

第三个用左外联结：

```
SELECT table1.*
FROM table1
LEFT OUTER JOIN table2
ON table1.id = table2.id
WHERE table2.id IS NULL;
```

具体例子见代码8-6a、代码8-6b、代码8-6c及图8-6。IN和EXISTS子查询将在本章后面介绍。

**代码8-6a** 这个语句使用IN子查询来列出那些没有写过书的作者。结果见图8-6

```
SELECT au_id, au_fname, au_lname
FROM authors
WHERE au_id NOT IN
(SELECT au_id FROM title_authors);
```

| au_id | au_fname | au_lname    |
|-------|----------|-------------|
| A07   | Paddy    | O'Furniture |

**代码8-6b** 这个语句等价于代码8-6a，但使用 EXISTS子查询而不是IN子查询。结果见图8-6

图8-6 运行代码8-6a、代码8-6b和代码8-6c的结果

```
SELECT au_id, au_fname, au_lname
FROM authors a
WHERE NOT EXISTS
(SELECT *
FROM title_authors ta
WHERE a.au_id = ta.au_id);
```

**代码8-6c** 这个语句等价于代码8-6a和代码8-6b，但使用左外联结而不是子查询。结果见图8-6

```
SELECT a.au_id, a.au_fname, a.au_lname
FROM authors a
LEFT OUTER JOIN title_authors ta
ON a.au_id = ta.au_id
WHERE ta.au_id IS NULL;
```

### ✓ 提示

- 也可以将子查询写成自联结（代码8-7a、代码8-7b及图8-7），要了解更多自联结的知识，参见7.9节。
- 内联结都可以写作子查询，但反过来就不行。这种不对称现象是因为内联结是可交换的，不论将表A和表B按什么顺序联结，得到的结果是相同的。子查询没有这个特点（外联结都可以写成子查询，尽管外联结不是可交换的）。

**代码8-7a** 这条语句使用子查询来列出和作者A04 (Klee Hull) 居住在同一个州的作者。

结果见图8-7

```
SELECT au_id, au_fname, au_lname, state
FROM authors
WHERE state IN
(SELECT state
FROM authors
WHERE au_id = 'A04');
```

| au_id | au_fname | au_lname | state |
|-------|----------|----------|-------|
| A03   | Hallie   | Hull     | CA    |
| A04   | Klee     | Hull     | CA    |
| A06   |          | Kellsey  | CA    |

图8-7 运行代码8-7a和代码8-7b的结果

**代码8-7b** 这条语句和代码8-7a等价，但使用内联结而不是子查询。结果见图8-7

```
SELECT a1.au_id, a1.au_fname,
 a1.au_lname, a1.state
 FROM authors a1
 INNER JOIN authors a2
 ON a1.state = a2.state
 WHERE a2.au_id = 'A04';
```

- 如果要将聚合值和其他值比较，用子查询更可取（代码8-8和图8-8）。不使用子查询，就需要两条SELECT语句来列示价格等于最高价格的图书：一个查询找出最高价格，另一个查询列出所有以这个价格销售的图书。要了解更多关于聚合函数的知识，见第6章。

**代码8-8** 列出价格等于最高价格的图书。结果见图8-8

```
SELECT title_id, price
 FROM titles
 WHERE price =
 (SELECT MAX(price)
 FROM titles);
```

| title_id | price |
|----------|-------|
| T03      | 39.95 |

图8-8 运行代码8-8的结果

- 在结果包含来自多个表的列的情况下，使用联结。代码8-5b使用联结来检索那些居住在有出版社的城市的作者。为了在结果中包含出版社的ID，只需在SELECT子句加上列pub\_id（代码8-9和图8-9）。

使用子查询无法完成同样的任务，因为在外部查询子句中包含仅出现在内部查询表的列是非法的。

```
SELECT a.au_id, a.city, p.pub_id
 FROM authors a
 WHERE a.city IN
 (SELECT p.city
 FROM publishers p); --Illegal
```

**代码8-9** 列出那些居住在有出版社的城市的作者，在结果中包含出版社。结果见图8-9

```
SELECT a.au_id, a.city, p.pub_id
 FROM authors a
 INNER JOIN publishers p
 ON a.city = p.city;
```

| au_id | city          | pub_id |
|-------|---------------|--------|
| A03   | San Francisco | P02    |
| A04   | San Francisco | P02    |
| A05   | New York      | P01    |

图8-9 运行代码8-9的结果

- **DBMS** MySQL 4.0和之前的版本不支持子查询，参见8.1节中的DBMS提示。

## 8.4 简单子查询和相关子查询

可以使用两种类型的子查询。

- 简单子查询
- 相关子查询

简单子查询（simple subquery）也叫非相关子查询（noncorrelated subquery），是指能够独立于外

部查询的子查询，它在整个语句中只运行一次。到目前为止，本章中的所有子查询例子（除了代码8-6b）都是简单子查询。

相关子查询（correlated subquery）无法独立于外部查询，它是依赖于外部查询结果的内部查询。如果语句需要针对外部查询的每一行在内部查询中处理一个表，就需要用到相关子查询。

相关子查询比简单子查询的语法更复杂，执行过程更曲折，但可以被用来解决无法用简单子查询和联结处理的问题。本节给出简单子查询和相关子查询的例子，然后描述DBMS如何运行它们。本章后面的小节包含更多每种子查询的例子。

### 8.4.1 简单子查询

DBMS通过执行一次内部查询来得出结果，并将结果提交到外部查询。简单子查询先于并独立于它的外部查询执行。

回顾本章前面的代码8-5a。代码8-10（它和代码8-5a是等价的）使用简单子查询列出那些居住在有出版社的城市的作者，结果见图8-10。从概念上讲，DBMS处理这个查询分成两个步骤，是两个独立的SELECT语句。

**代码8-10** 列出居住在有出版社的城市的作者。结果见8-10

```
SELECT au_id, city
 FROM authors
 WHERE city IN
 (SELECT city
 FROM publishers);
```

(1) 内部查询（简单子查询）返回所有出版社所在的城市（代码8-11和图8-11）。

(2) DBMS将步骤(1)中内部查询返回的值替换到外部查询，找出那些对应于出版社所在城市的作者ID（代码8-12和图8-12）。

**代码8-11** 列出有出版社的城市。结果见图8-11

```
SELECT city
 FROM publishers;
```

**代码8-12** 列出居住在代码8-11返回的有出版社的城市的作者。结果见图8-12

```
SELECT au_id, city
 FROM authors
 WHERE city IN
 ('New York', 'San Francisco',
 'Hamburg', 'Berkeley');
```

| au_id | city          |
|-------|---------------|
| A03   | San Francisco |
| A04   | San Francisco |
| A05   | New York      |

图8-10 运行代码8-10的结果

| city          |
|---------------|
| New York      |
| San Francisco |
| Hamburg       |
| Berkeley      |

图8-11 运行代码8-11的结果

| au_id | city          |
|-------|---------------|
| A03   | San Francisco |
| A04   | San Francisco |
| A05   | New York      |

图8-12 运行代码8-12的结果

### 8.4.2 相关子查询

相关子查询提供了比简单子查询更有力的数据检索机制。

相关子查询的重要特点如下。

- 它与简单子查询的执行顺序和执行次数不同。
- 因为它依赖外部查询得到值，所以无法独立于外部查询执行。
- 它重复执行——为外部查询选择的每一候选行执行一次。
- 它总是引用外部查询FROM子句指定的表。
- 它使用限定列名来引用外部查询确定的值。在相关子查询中，这些限定的列名被称作相关变量。要了解更多有关限定列名和表别名的知识，参见7.1节和7.2节。
- 包含相关子查询的查询的基本语法如下。

```
SELECT outer_columns
 FROM outer_table
 WHERE outer_column_value IN
 (SELECT inner_column
 FROM inner_table
 WHERE inner_column = outer_column)
```

执行总是开始于外部查询（代码体）。外部查询选择*outer\_table*表的每一行作为候选行。对于每一候选行，DBMS执行相关内部查询（粗代码体）一次，对于*outer\_column*值满足内部WHERE条件的*inner\_table*行加标记。DBMS针对被标记的*inner\_table*行测试外部WHERE条件，显示那些满足条件的标记行。这个过程持续到所有候选行被处理完为止。

代码8-13使用相关子查询列出那些销量超过或等于其所属类型平均销量的图书，结果见图8-13。  
candidate（在外部查询titles之后）和average（在内部查询titles之后）是表titles的别名，于是信息就能够从两个不同的表获得（见7.9节）。

**代码8-13** 列出那些销量超过或等于所属类型平均销售的图书。相关变量candidate.type定义了内部表average行需要满足的初始条件。外部WHERE条件(sales >=)定义了内部表average必须满足的最终测试。结果见图8-13

```
SELECT
 candidate.title_id,
 candidate.type,
 candidate.sales
 FROM titles candidate
 WHERE sales >=
 (SELECT AVG(sales)
 FROM titles average
 WHERE average.type = candidate.type);
```

|     | title_id   | type    | sales |
|-----|------------|---------|-------|
| T02 | history    | 9566    |       |
| T03 | computer   | 25667   |       |
| T05 | psychology | 201440  |       |
| T07 | biography  | 1500200 |       |
| T09 | children   | 5000    |       |
| T13 | history    | 10467   |       |

图8-13 运行代码8-13的结果

在代码8-13中，子查询无法独立于外部查询运行。它需要一个candidate.type值，但这个值是相关变量，会随着DBMS检测candidate表不同的行而变化。列average.type和外部查询candidate.type相关。通过使用外部查询(candidate)中每本书的类型，子查询计算每种类型图书的平均销量。子查询计算出这个类型的平均销量，然后将candidate表中的行与它比较。如果candidate表中的销售超过或等于这个图书类型的平均销售，这本书就显示在结果中。DBMS按照如下顺序执行这个查询。

(1) 表candidate第一行的图书类型被用在子查询中计算平均销售。

选取图书T01所在行，因为它的类型是历史，于是在表candidate第一行中列type的值是历史。实际上，子查询变为：

```
SELECT AVG(sales)
 FROM titles average
 WHERE average.type = 'history';
```

这个值通过子查询产生值6 866——历史书的平均销量。在外部查询中，图书T01的销售量566与历史书的平均销量进行比较。T01的销量低于平均销量，于是不显示在结果中。

(2) 判断表candidate中图书T02行。

T02也是历史书，于是子查询的计算和步骤(1)相同。

```
SELECT AVG(sales)
 FROM titles average
 WHERE average.type = 'history';
```

通过子查询再一次产生历史书的平均销量6 866。书 T02的销量为9 566，高于平均数，于是T02出现在结果中。

(3) 判断表candidate中图书T03行。

T03是计算机书，于是这次子查询的计算如下。

```
SELECT AVG(sales)
 FROM titles average
 WHERE average.type = 'computer';
```

通过子查询得到的结果是计算机类图书的平均销量为25 667。因为图书T03的销量25 667等于平均数（它是唯一的计算机书），所以T03出现在结果中。

(4) DBMS重复这个过程直到外部表candidate的每一行都被检测完。

### ✓ 提示

□ 如果使用简单子查询或者相关子查询都可以得到结果，就应使用简单子查询，因为它运行得快。代码8-14a和8-14b显示了两个等价的查询，列出获得图书100%稿酬的作者。使用简单子查询的代码8-14a比使用相关子查询的8-14b的执行效率要高。在简单子查询中，DBMS读取内部表title\_authors一次。在相关子查询中，DBMS必须循环读取表title\_authors 5次——对于外部表authors每行都需一次。结果见图8-14。

当相关子查询显然需要做更多的工作时，为什么只说简单子查询有可能会比等价的相关子查询快。这是因为DBMS的优化器也许能够在执行语句前识别出相关子查询，并自动将其转化为语义等价的简单子查询。要了解更多相关知识，参见8.13节。

□ **DBMS** MySQL 4.0和之前的版本不支持子查询，参见8.1节中的DBMS提示。

在较早的PostgreSQL版本中，需转换代码8-14a和8-14b的浮点数为DECIMAL，见5.13节。为了运行代码8-14a和8-14b，须转换代码中的浮点字面量为：CAST(1.0 AS DECIMAL)。

**代码8-14a** 这条语句使用简单子查询列出所有获得图书100%稿酬的作者。结果见图8-14

```
SELECT au_id, au_fname, au_lname
 FROM authors
 WHERE au_id IN
 (SELECT au_id
 FROM title_authors
 WHERE royalty_share = 1.0);
```

**代码8-14b** 这条语句等价于代码8-14a。它使用了相关子查询，而不是简单子查询。这个查询可能会比代码8-14a运行得慢。结果见图8-14

```
SELECT au_id, au_fname, au_lname
 FROM authors
 WHERE 1.0 IN
 (SELECT royalty_share
 FROM title_authors
 WHERE title_authors.au_id =
 authors.au_id);
```

| au_id | au_fname  | au_lname  |
|-------|-----------|-----------|
| A01   | Sarah     | Buchman   |
| A02   | Wendy     | Heydemark |
| A04   | Klee      | Hull      |
| A05   | Christian | Kells     |
| A06   |           | Kellsey   |

图8-14 运行代码8-14a和8-14b的结果

## 8.5 在子查询中限定列名

7.1节介绍了使用表名限定列名以避免混淆。在包含子查询的语句中，列名隐式地限定为那个同一嵌套层次FROM子句所引用的表。

代码8-15a中，列出传记出版社的名称，列名是隐式限定的，意味着：

- 在外部查询WHERE子句中的pub\_id列，由外部查询FROM子句中的表publishers隐式限定。
- 在子查询SELECT子句中，列pub\_id由子查询FROM子句的表titles隐式限定。

代码8-15b表示代码8-15a的显式限定，结果见图8-15。

**代码8-15a** 表publishers与表titles都包含名为pub\_id的列，但没有必要限定pub\_id，因为在这个查询中隐含着SQL要使用的表名。结果见图8-15

```
SELECT pub_name
 FROM publishers
 WHERE pub_id IN
 (SELECT pub_id
 FROM titles
 WHERE type = 'biography');
```

| pub_name            |
|---------------------|
| Abatis Publishers   |
| Schadenfreude Press |

图8-15 运行代码8-15a和8-15b的结果

**代码8-15b** 这条查询等价于代码8-15a，但显式限定了pub\_id。结果见图8-15

```
SELECT pub_name
 FROM publishers
 WHERE publishers.pub_id IN
 (SELECT titles.pub_id
 FROM titles
 WHERE type = 'biography');
```

### ✓ 提示

- 显式说明表名，就不会出错。
- 为列限定一个与它不在同一嵌套层次的表名，可以重写SQL默认的表名。
- 如果列名在同一嵌套层次上对应多个表，列名就是模糊的，这时就必须限定表名或表的别名。
- DBMS MySQL 4.0及之前的版本不支持子查询，参见8.1节的DBMS提示。

## 8.6 子查询中的空值

要小心空值，因为它们会使子查询变得复杂。当空值出现的时候，如果不删除它们就有可能出现意外的结果。

子查询可能会隐藏空值的比较。回顾3.14节，空值互不相等，因此无法判断空值与其他任何值是否相等。后面例子包含NOT IN子查询（见8.9节）。考虑下面的两张表，每个表各有一列。第一个表命名为table1：

```
col

1
2
```

第二个表名为table2：

```
col

1
2
3
```

运行代码8-16列出表table2中存在而table1中不存在的值。结果见图8-16a所示。

图8-16a 当table1不包含空值时，运行代码8-16的结果

| col |
|-----|
| 3   |

代码8-16 列出表table2中存在而表table1中不存在的值。结果见图8-16

```
SELECT col
 FROM table2
 WHERE col NOT IN
 (SELECT col
 FROM table1);
```

现在在table1加入一个空值。

```
col

1
2
NULL
```

如果运行代码8-16，得到结果图8-16b（一张空表），这个结果合乎逻辑但出乎意料。为什么这次的结果是空表？回答这个问题要用到一些代数知识。将NOT移动到子查询条件之外并不会改变代码8-16的含义。

8

268

图8-16b 当table1包含一个空值时，运行代码8-16的结果。结果是空表，符合逻辑但出乎意料

| col |
|-----|
|     |

```
SELECT col
 FROM table2
 WHERE NOT col IN
 (SELECT col FROM table1);
```

IN子句决定了table2中的值是否与table1中的某个值匹配，于是将子查询重写为组合条件。

```
SELECT col
 FROM table2
 WHERE NOT ((col = 1)
 OR (col = 2))
```

```
OR (col = NULL));
```

如果应用De Morgan's法则（见第4章中的表4-6），这个查询变为：

```
SELECT col
 FROM table2
 WHERE (col <> 1)
 AND (col <> 2)
 AND (col <> NULL);
```

最后的表达式`col <> NULL`总是未知的，引用AND真值表（第4章中的表4-3），将看到整个WHERE搜索条件变成未知的，并且总会被WHERE拒绝。

为了修改代码8-16，使其不检查table1中的空值，对子查询加上IS NOT NULL条件（见4.10节）。

```
SELECT col
 FROM table2
 WHERE col NOT IN
 (SELECT col
 FROM table1
 WHERE col IS NOT NULL);
```

#### ✓ 提示

269

**DBMS** MySQL 4.0和之前的版本不支持子查询，参见8.1节中的DBMS提示。

## 8.7 使用子查询作为列表达式

在第4章至第6章，已经说明SELECT子句列示的项可以是字面量、列名或较复杂的表达式。SQL允许在SELECT子句中嵌入一个子查询。

用来作为列表达式的子查询必须是标量子查询。8.2节中的表8-1指出，一个标量子查询返回单一值（即一行一列的结果）。在大多数情况下，必须在子查询中使用聚合函数或限制性WHERE条件，以保证该子查询只返回一行。

SELECT子句的语法和前面一直使用的相同，只是在列表中可以指定一个括号内的子查询作为列表达式，如后面举例所示。

270

代码8-17使用两个简单子查询作为列表达式列出每本传记及其价格、所有图书的平均价格（不只是传记），以及每本传记的价格和所有图书平均价格的差价。聚合函数AVG()保证每个子查询返回唯一值，结果见图8-17。AVG()计算平均值时忽略空值，参见6.6节。

**代码8-17** 列出每本传记及其价格，所有图书的平均价格，每本传记的价格与所有图书平均价格的差价。结果见图8-17

```
SELECT title_id,
 price,
 (SELECT AVG(price) FROM titles)
 AS "AVG(price)",
 price - (SELECT AVG(price) FROM titles)
 AS "Difference"
 FROM titles
 WHERE type='biography';
```

|     | title_id | price   | AVG(price) | Difference |
|-----|----------|---------|------------|------------|
| T06 | 19.95    | 18.3875 | 1.5625     |            |
| T07 | 23.95    | 18.3875 | 5.5625     |            |
| T10 | NULL     | 18.3875 | NULL       |            |
| T12 | 12.99    | 18.3875 | -5.3975    |            |

图8-17 运行代码8-17的结果

代码8-18应用相关子查询在一行里显示每本书的所有作者，就像在报告或电子表格中看到的那样。结果见图8-18。注意在每个WHERE子句中，SQL在子查询的FROM子句中引用表的别名ta隐式限定title\_id，见8.5节。要了解实现这个查询的更高效方法，查看本节提示。代码15-8为这个查询的改写。

### 代码8-18 在一行中列出每本书的所有作者。结果见图8-18

```
SELECT title_id,
 (SELECT au_id
 FROM title_authors ta
 WHERE au_order = 1
 AND title_id = t.title_id)
 AS "Author 1",
 (SELECT au_id
 FROM title_authors ta
 WHERE au_order = 2
 AND title_id = t.title_id)
 AS "Author 2",
 (SELECT au_id
 FROM title_authors ta
 WHERE au_order = 3
 AND title_id = t.title_id)
 AS "Author 3"
 FROM titles t;
```

代码8-19改写7.8节的代码7-30，但这次使用了相关子查询，而非外联结列出作者写（或合写）的图书数量，结果见图8-19。

### 代码8-19 列出每个作者写（或合写）的图书数量，包括那些没还有写过书的作者。结果见图8-19

```
SELECT au_id,
 (SELECT COUNT(*)
 FROM title_authors ta
 WHERE ta.au_id = a.au_id)
 AS "Num books"
 FROM authors a
 ORDER BY au_id;
```

代码8-20使用了相关子查询列出每个作者及其最近出版图书的日期。应该在包含了联结的子查询中显式指出每列的名称，清楚表明引用了哪张表（即便限定符不是必需的），结果见图8-20。

### 代码8-20 列出每个作者和他（或她）最新图书的出版日期。结果见图8-20

```
SELECT au_id,
 (SELECT MAX(pubdate)
```

| title_id | Author 1 | Author 2 | Author 3 |
|----------|----------|----------|----------|
| T01      | A01      | NULL     | NULL     |
| T02      | A01      | NULL     | NULL     |
| T03      | A05      | NULL     | NULL     |
| T04      | A03      | A04      | NULL     |
| T05      | A04      | NULL     | NULL     |
| T06      | A02      | NULL     | NULL     |
| T07      | A02      | A04      | NULL     |
| T08      | A06      | NULL     | NULL     |
| T09      | A06      | NULL     | NULL     |
| T10      | A02      | NULL     | NULL     |
| T11      | A06      | A03      | A04      |
| T12      | A02      | NULL     | NULL     |
| T13      | A01      | NULL     | NULL     |

图8-18 运行代码8-18的结果

| au_id | Num books |
|-------|-----------|
| A01   | 3         |
| A02   | 4         |
| A03   | 2         |
| A04   | 4         |
| A05   | 1         |
| A06   | 3         |
| A07   | 0         |

图8-19 运行代码8-19的结果

| au_id | Latest pub date |
|-------|-----------------|
| A01   | 2000-08-01      |
| A02   | 2000-08-31      |
| A03   | 2000-11-30      |
| A04   | 2001-01-01      |
| A05   | 2000-09-01      |
| A06   | 2002-05-31      |
| A07   | NULL            |

图8-20 运行代码8-20的结果

```

FROM titles t
INNER JOIN title_authors ta
 ON ta.title_id = t.title_id
WHERE ta.au_id = a.au_id)
 AS "Latest pub date"
FROM authors a;

```

代码8-21应用相关子查询来计算所有图书的累加值。累加值，或者说累积总计，是一种普通的运算：对于每一本书都要计算所有在它之前图书销量的汇总数。在这里，按title\_id字母顺序来对书排序。注意表别名在上下两段中指同一个表。子查询返回在这本书之前的销售汇总，用t1.title\_id指明，结果见图8-21，参见15.1节。

#### 代码8-21 计算所有图书累计销售总额。结果见图8-21

```

SELECT t1.title_id, t1.sales,
 (SELECT SUM(t2.sales)
 FROM titles t2
 WHERE t2.title_id <= t1.title_id)
 AS "Running total"
 FROM titles t1;

```

#### ✓ 提示

- 可以在FROM子句中使用子查询。在6.8节的提示中，使用FROM子查询取代不重复的聚合函数。
- 代码8-22使用FROM子查询计算所有作者中写（或合写）书最多的数量，结果见图8-22。注意外部查询使用表别名（ta）和列标签（count\_titles）来引用内部查询的结果。参见4.5节的“列的别名和WHERE”提要栏。

#### 代码8-22 计算出所有作者中所写（或合写）图书最多的数据量。结果见图8-22

```

SELECT MAX(ta.count_titles) AS "Max titles"
 FROM (SELECT COUNT(*) AS count_titles
 FROM title_authors
 GROUP BY au_id) ta;

```

- 在UPDATE、INSERT和DELETE语句中可以使用子查询作为列表达式（见第10章），但不能用在ORDER BY列表中。
- 使用CASE表达式而不是相关子查询，可以更高效地实现代码8-18（见5.14节）。

```

SELECT title_id,
 MIN(CASE au_order WHEN 1
 THEN au_id
 END)
 AS "Author 1",
 MIN(CASE au_order WHEN 2
 THEN au_id
 END)
 AS "Author 2",
 MIN(CASE au_order WHEN 3

```

|     | title_id | sales   | Running total |
|-----|----------|---------|---------------|
| T01 | 566      | 566     |               |
| T02 | 9566     | 10132   |               |
| T03 | 25667    | 35799   |               |
| T04 | 13001    | 48800   |               |
| T05 | 201440   | 250240  |               |
| T06 | 11320    | 261560  |               |
| T07 | 1500200  | 1761760 |               |
| T08 | 4095     | 1765855 |               |
| T09 | 5000     | 1770855 |               |
| T10 | NULL     | 1770855 |               |
| T11 | 94123    | 1864978 |               |
| T12 | 100001   | 1964979 |               |
| T13 | 10467    | 1975446 |               |

图8-21 运行代码8-21的结果

#### Max titles

-----  
4

图8-22 运行代码8-22的结果

```

 THEN au_id
 END)
 AS "Author 3"
FROM title_authors
GROUP BY title_id
ORDER BY title_id ASC;

```

- DBMS** MySQL 4.0和之前版本不支持子查询，参见8.1节中DBMS提示。

在Microsoft Access中，必须提高代码8-17中平均价格的精确度。使用类型转换函数Cdbl()强制平均价格转为双精度浮点数，参见5.13节中的DBMS提示。为了运行代码8-17，变所有AVG(price)为Cdbl(AVG(price))。

274

## 8.8 使用比较操作符比较子查询的值

在WHERE或HAVING子句中可以使用比较操作符（=、 $\neq$ 、 $<$ 、 $\leq$ 、 $>$ 或 $\geq$ ）将子查询作为过滤器。子查询比较的重要特点如下。

- 比较操作符和它们在其他比较中的作用相同（见第4章中的表4-2）。
- 子查询可以是简单的或相关的（见8.4节）。
- 子查询的SELECT子句仅可以包括一个表达式或列名。
- 被比较的值必须是同一数据类型或隐式转换为同一类型（见5.13节）。
- 串比较是否区分大小写取决于具体的DBMS，见4.5节的DBMS提示。
- 子查询必须只能返回一个值（一行一列的值）。子查询返回多个值将引发错误。
- 如果子查询的结果包含零行，比较判断值为假。

275

写这条语句的困难在于让子查询返回一个值。这可以用几种方式来确保实现。

- 对未分组的表使用聚合函数总是可以得到唯一返回值（见第6章）。
- 在总是返回唯一值的键上使用外联结。

8

### » 比较子查询的值

在SELECT语句的WHERE子句中，输入：

WHERE *test\_expr op (subquery)*

*test\_expr*是字面量、列名、表达式或返回唯一值的子查询，*op*是比较操作符（=、 $\neq$ 、 $<$ 、 $\leq$ 、 $>$ 或 $\geq$ ），*subquery*是仅能返回一列且一行或零行的标量子查询。

如果子查询返回的值满足同*test\_expr*的比较，对比条件的值为真。如果子查询的值不能满足条件，如子查询的值为空值，或子查询的结果为空（零行），则对比条件的值为假。

同样的语法适用于HAVING子句：

HAVING *test\_expr op (subquery)*

代码8-23测试简单子查询的结果等价于列出那些和Tenterhooks出版社同处一个州的作者。只有一个名为Tenterhooks的出版社，于是内部的WHERE条件保证内部查询返回唯一值的结果，结果见图8-23。

276

**代码8-23** 列出那些和Tenterhooks 出版社同处一个州的作者。结果见图8-23

```

SELECT au_id, au_fname, au_lname, state
 FROM authors
 WHERE state =

```

```
SELECT state
FROM publishers
WHERE pub_name = 'Tenterhooks Press');
```

代码8-24和代码8-23除了出版社的名称以外都是相同的。不存在名为XXX的出版社，于是子查询返回了一个空表（零行）。与空值进行比较，结果是空表，结果见图8-24。

**代码8-24** 列出那些和出版社XXX同处一州的作者。结果见图8-24

```
SELECT au_id, au_fname, au_lname, state
FROM authors
WHERE state =
 (SELECT state
 FROM publishers
 WHERE pub_name = 'XXX');
```

代码8-25列出超过平均销售量的图书。比较操作符引出的子查询经常使用聚合函数来返回唯一值，  
结果见图8-25。

277

**代码8-25** 列出超过平均销量的图书。结果见图  
8-25

```
SELECT title_id, sales
FROM titles
WHERE sales >
 (SELECT AVG(sales)
 FROM titles);
```

为列出作品销量超过图书平均销量的作者，代码8-25加上了一个内联结（代码8-26和图8-26）。

**代码8-26** 使用联结和子查询列出超过平均销量图书的作者。结果见图8-26

```
SELECT ta.au_id, ta.title_id
FROM titles t
INNER JOIN title_authors ta
 ON ta.title_id = t.title_id
WHERE sales >
 (SELECT AVG(sales)
 FROM titles)
ORDER BY ta.au_id ASC, ta.title_id ASC;
```

回顾本章的内容，几乎所有可以使用表达式的地方都可以使用子查询，这样的语法是有效的：

**WHERE (subquery) op (subquery)**

左边的子查询必须返回唯一值。代码8-27等同于代码8-26，但在比较操作符的左边用相关子查询取代了内联结，结果见图8-27。

**代码8-27** 用两个子查询列出超过平均销量图书的作者。结果见图8-27

```
SELECT au_id, title_id
```

| au_id | au_fname | au_lname | state |
|-------|----------|----------|-------|
| A03   | Hallie   | Hull     | CA    |
| A04   | KLee     | Hull     | CA    |
| A06   |          | Kellsey  | CA    |

图8-23 运行代码8-23的结果

| au_id | au_fname | au_lname | state |
|-------|----------|----------|-------|
|       |          |          |       |

图8-24 运行代码8-24的结果（空表）

| title_id | sales   |
|----------|---------|
| T05      | 201440  |
| T07      | 1500200 |

图8-25 运行代码8-25的结果

| au_id | title_id |
|-------|----------|
| A02   | T07      |
| A04   | T05      |
| A04   | T07      |

图8-26 运行代码8-26的结果

```

FROM title_authors ta
WHERE
 (SELECT AVG(sales)
 FROM titles t
 WHERE ta.title_id = t.title_id)
 >
 (SELECT AVG(sales)
 FROM titles)
ORDER BY au_id ASC, title_id ASC;

```

| au_id | title_id |
|-------|----------|
| A02   | T07      |
| A04   | T05      |
| A04   | T07      |

图8-27 运行代码8-27的结果

如果能确定GROUP BY或HAVING子句返回唯一值，那么在子查询中可以包含GROUPBY或HAVING子句。代码8-28列出价格高于传记最高价格的图书，结果见图8-28。

278

**代码8-28** 列出价格高于传记最高价格的图书。结果见图8-28

```

SELECT title_id, price
FROM titles
WHERE price >
 (SELECT MAX(price)
 FROM titles
 GROUP BY type
 HAVING type = 'biography');

```

| title_id | price |
|----------|-------|
| T03      | 39.95 |
| T13      | 29.99 |

图8-28 运行代码8-28的结果

代码8-29在HAVING子句中使用子查询列出图书平均销量超过总平均销量的出版社。子查询返回唯一值（所有图书总平均销量），结果见图8-29。

8

**代码8-29** 列出其平均销量超过总体平均销量的出版社。结果见图8-29

```

SELECT pub_id, AVG(sales) AS "AVG(sales)"
FROM titles
GROUP BY pub_id
HAVING AVG(sales) >
 (SELECT AVG(sales)
 FROM titles);

```

| pub_id | AVG(sales) |
|--------|------------|
| P03    | 506744.33  |

图8-29 运行代码8-29的结果

代码8-30使用相关子查询列出合著者稿酬比例低于最高稿酬比例的图书作者。外部查询在title\_authors（也就是ta1）中一行接一行地进行。子查询得出要被用于外部查询的每本书的最高稿酬比例。对于ta1的每一个可能的值，DBMS判断子查询，如果低于稿酬的最大比例，则将正在检测的行放入结果中，结果见图8-30。

279

**代码8-30** 列出合著者稿酬比例低于最高比例的图书作者。结果见8-30

```

SELECT ta1.au_id, ta1.title_id,
 ta1.royalty_share
 FROM title_authors ta1
 WHERE ta1.royalty_share <
 (SELECT MAX(ta2.royalty_share)
 FROM title_authors ta2
 WHERE ta1.title_id = ta2.title_id);

```

| au_id | title_id | royalty_share |
|-------|----------|---------------|
| A04   | T04      | 0.40          |
| A03   | T11      | 0.30          |
| A04   | T11      | 0.30          |

图8-30 运行代码8-30的结果

代码8-31使用了相关子查询来模仿GROUP BY子句，列出价格超过它所属类型的平均价格的图书。

对于t1每一个可能的值，DBMS计算子查询，如果其价格超过计算得出的平均价格，则在结果中包含这一行。没有必要用类型做显式分组，因为子查询的WHERE子句限制计算平均价格的行，结果见图8-31。

### 代码8-31 列出那些价格超过所属类型图书平均价的图书。结果见图8-31

```
SELECT type, title_id, price
 FROM titles t1
 WHERE price >
 (SELECT AVG(t2.price)
 FROM titles t2
 WHERE t1.type = t2.type)
 ORDER BY type ASC, title_id ASC;
```

代码8-32使用与代码8-31同样的结构，列出销量小于其所属类型的最畅销书的图书，结果见图8-32。

### 代码8-32 列出销量小于所属类型的最畅销书的图书。结果见图8-32

```
SELECT type, title_id, sales
 FROM titles t1
 WHERE sales <
 (SELECT MAX(sales)
 FROM titles t2
 WHERE t1.type = t2.type
 AND sales IS NOT NULL)
 ORDER BY type ASC, title_id ASC;
```

#### ✓ 提示

- 如果子查询返回不止一行，可以使用ALL或ANY来限制比较操作符，或用IN来引入子查询（本章后面的内容将介绍ALL、ANY和IN）。
- **DBMS MySQL 4.0**和之前版本不支持子查询，参见8.1节中的DBMS提示。

## 8.9 使用 IN 测试集合成员资格

4.9节说明如何在WHERE子句中使用关键字IN将字面量、列值或更复杂的表达式和值列表进行比较。可以使用子查询来产生列表。

子查询集合成员资格测试最重要的特点如下。

- IN关键字对于子查询的运行结果和括号内的一组值，其运行方式是相同的（见4.9节）。
- 子查询可以是简单子查询或相关子查询（见8.4节）。
- 子查询的SELECT子句只可以包含一个表达式或列名。
- 被比较的值必须具有相同的数据类型，或是可以被隐式转换成同样的类型（见5.13）。
- 串比较是否区分大小写由DBMS决定，参见4.5节的DBMS提示。
- 子查询必须返回一列零行或多行，子查询返回超过一列会引发错误。
- 使用NOT IN来实现IN的反向测试。使用NOT IN，如果没有匹配子查询中的结果，那么DBMS执行SQL语句指定的动作。

| type       | title_id | price |
|------------|----------|-------|
| biography  | T06      | 19.95 |
| biography  | T07      | 23.95 |
| children   | T09      | 13.95 |
| history    | T13      | 29.99 |
| psychology | T04      | 12.99 |

图8-31 运行代码8-31的结果

| type       | title_id | sales  |
|------------|----------|--------|
| biography  | T06      | 11320  |
| biography  | T12      | 100001 |
| children   | T08      | 4095   |
| history    | T01      | 566    |
| history    | T02      | 9566   |
| psychology | T04      | 13001  |
| psychology | T11      | 94123  |

图8-32 运行代码8-32的结果

### ⇒ 检测集合成员资格

在SELECT语句的WHERE子句中，输入：

```
WHERE test_expr [NOT] IN (subquery)
```

*test\_expr*是一个字面量、列名、表达式或返回唯一值的子查询，子查询应返回一列零行或多行。

如果*test\_expr*等于子查询返回的任一值，IN条件结果为真。如果子查询返回为空，或子查询结果没有任何一行与*test\_expr*匹配，所有子查询返回结果全为空值，那么IN条件结果为假。指明NOT对条件结果取反。

这些语法规则同样适用于HAVING子句。

```
HAVING test_expr [NOT] IN (subquery)
```

代码8-33列出出版过传记的出版社的名称。DBMS分两步得出结果。第一步，内部查询返回出版过传记的出版社(P01和P03)。第二步，DBMS将这些值提交给外部查询，外部查询查找与表publishers中ID匹配的出版社，结果见图8-33。

代码8-33 列出出版过传记的出版社。结果见图8-33

```
SELECT pub_name
 FROM publishers
 WHERE pub_id IN
 (SELECT pub_id
 FROM titles
 WHERE type = 'biography');
```

| pub_name            |
|---------------------|
| -----               |
| Abatis Publishers   |
| Schadenfreude Press |

282

图8-33 运行代码8-33的结果

这里使用联结对代码8-33进行改写。

```
SELECT DISTINCT pub_name
 FROM publishers p
 INNER JOIN titles t
 ON p.pub_id = t.pub_id
 AND type = 'biography';
```

代码8-34与代码8-33基本相同，只是使用NOT IN列出没有出版过传记的出版社，结果见图8-34。这个语句不能用联结改写。类似的不等联结有不同的意义：它列出已经出版过一些图书但不是传记的出版社名字。

代码8-34 列出没有出版过传记的出版社的名称。

结果见图8-34

```
SELECT pub_name
 FROM publishers
 WHERE pub_id NOT IN
 (SELECT pub_id
 FROM titles
 WHERE type = 'biography');
```

| pub_name          |
|-------------------|
| -----             |
| Core Dump Books   |
| Tenterhooks Press |

图8-34 运行代码8-34的结果

8

代码8-35与代码7-31等价，除了使用子查询而不是外联结列出没有写（或合写）过图书的作者，结果见图8-35。

代码8-35 列出没有写（或合写）过图书的作者。结果见图8-35

```
SELECT au_id, au_fname, au_lname
```

```
FROM authors
WHERE au_id NOT IN
 (SELECT au_id
 FROM title_authors);
```

代码8-36列出在出版社P03 (Schadenfreude Press) 出版过书的作者。为了在结果中显示作者的名字(不只是他们的ID), 联结表authors是必要的, 结果见图8-36。

283

**代码8-36** 列出已经在出版社P03出版过图书的作者名字。结果见图8-36

```
SELECT DISTINCT a.au_id, au_fname, au_lname
 FROM title_authors ta
 INNER JOIN authors a
 ON ta.au_id = a.au_id
 WHERE title_id IN
 (SELECT title_id
 FROM titles
 WHERE pub_id = 'P03');
```

子查询本身也可以包含一个或多个子查询。代码8-37列出至少参与了一部传记写作的作者。最内层的查询返回了书名ID——T06、T07、T10和T12。DBMS在下一层的子查询中使用这些书名ID返回作者ID。最后, 最外层的查询用作者ID查询出作者姓名, 结果见图8-37。

**代码8-37** 列出至少参与了一部传记写作的作者。结果见图8-37

```
SELECT au_id, au_fname, au_lname
 FROM authors
 WHERE au_id IN
 (SELECT au_id
 FROM title_authors
 WHERE title_id IN
 (SELECT title_id
 FROM titles
 WHERE type = 'biography'));
```

过多的子查询嵌套会让语句难以理解。在一般情况下, 用联结重写这样的查询是很容易的。下面对代码8-37使用联结改写为:

```
SELECT DISTINCT a.au_id, au_fname,
 au_lname
 FROM authors a
 INNER JOIN title_authors ta
 ON a.au_id = ta.au_id
 INNER JOIN titles t
 ON t.title_id = ta.title_id
 WHERE type = 'biography';
```

代码8-38列出生活在California且收到的稿酬比例不超过50%的非主要作者(*au\_order>1*), 结果见图8-38。

| au_id | au_fname | au_lname    |
|-------|----------|-------------|
| A07   | Paddy    | O'Furniture |

图3-35 运行代码8-35的结果

| au_id | au_fname | au_lname  |
|-------|----------|-----------|
| A01   | Sarah    | Buchman   |
| A02   | Wendy    | Heydemark |
| A04   | Klee     | Hull      |

图3-36 运行代码8-36的结果

| au_id | au_fname | au_lname  |
|-------|----------|-----------|
| A02   | Wendy    | Heydemark |
| A04   | Klee     | Hull      |

图3-37 运行代码8-37的结果

**代码8-38** 列出生活在加州且收到的稿酬比例少于50%的非主要作者。结果见图8-38

```
SELECT au_id, au_fname, au_lname
 FROM authors
 WHERE state = 'CA'
 AND au_id IN
 (SELECT au_id
 FROM title_authors
 WHERE royalty_share < 0.5
 AND au_order > 1);
```

| au_id | au_fname | au_lname |
|-------|----------|----------|
| A03   | Hallie   | Hull     |
| A04   | Klee     | Hull     |

图8-38 运行代码8-38的结果

下面对代码8-38使用联结改写为：

```
SELECT DISTINCT a.au_id, au_fname,
 au_lname
 FROM authors a
 INNER JOIN title_authors ta
 ON a.au_id = ta.au_id
 WHERE state = 'CA'
 AND royalty_share < 0.5
 AND au_order > 1;
```

284

代码8-39列出合写过书的作者。为了确定某个作者是一本书的唯一作者还是合写者，检查他这本书的稿酬比例。如果稿酬比例小于100%（1.0），这个作者就是合作者，否则就是唯一作者，结果见图8-39。

**代码8-39** 列出合写过书的作者名字。结果见图8-39

```
SELECT au_id, au_fname, au_lname
 FROM authors a
 WHERE au_id IN
 (SELECT au_id
 FROM title_authors
 WHERE royalty_share < 1.0);
```

| au_id | au_fname | au_lname  |
|-------|----------|-----------|
| A02   | Wendy    | Heydemark |
| A03   | Hallie   | Hull      |
| A04   | Klee     | Hull      |
| A06   |          | Kellsey   |

8

图8-39 运行代码8-39的结果

代码8-40用相关子查询列出那些独立写过书的作者的名字——也就是拿到了100%（1.0）稿酬的作者，结果见图8-40。DBMS将外部查询表authors的每行都视为结果的候选行。当DBMS检查表authors的第一候选行时，它设置相关变量a.au\_id等于A01(Sarah Buchman)，并将这个提交到内部查询。

```
SELECT royalty_share
 FROM title_authors ta
 WHERE ta.au_id = 'A01';
```

内部查询返回1.0，于是外部查询判断：

```
SELECT a.au_id, au_fname, au_lname
 FROM authors a
 WHERE 1.0 IN (1.0)
```

WHERE条件为真，于是作者A01被包含在结果中。DBMS对每个作者重复这个过程，参见8.4节。

285

**代码8-40** 列出独立写过书的作者。结果见图8-40

```
SELECT a.au_id, au_fname, au_lname
```

```
FROM authors a
WHERE 1.0 IN
 (SELECT royalty_share
 FROM title_authors ta
 WHERE ta.au_id = a.au_id);
```

代码8-41列出既合写过书又独立写过书的作者名字。内部查询返回唯一作者的作者ID，外部查询比较这些ID与合作者的ID，结果见图8-41。

**代码8-41** 列出既独立写书又合写过书的作者名字。结果见图8-41

```
SELECT DISTINCT a.au_id, au_fname, au_lname
 FROM authors a
 INNER JOIN title_authors ta
 ON a.au_id = ta.au_id
 WHERE ta.royalty_share < 1.0
 AND a.au_id IN
 (SELECT a.au_id
 FROM authors a
 INNER JOIN title_authors ta
 ON a.au_id = ta.au_id
 AND ta.royalty_share = 1.0);
```

你可以用联结或交集重写代码8-41。下面对代码8-41使用联结改写为：

```
SELECT DISTINCT a.au_id, au_fname,
 au_lname
 FROM authors a
 INNER JOIN title_authors ta1
 ON a.au_id = ta1.au_id
 INNER JOIN title_authors ta2
 ON a.au_id = ta2.au_id
 WHERE ta1.royalty_share < 1.0
 AND ta2.royalty_share = 1.0;
```

下面对代码8-41使用交集改写为（见9.2节）：

```
SELECT DISTINCT a.au_id, au_fname,
 au_lname
 FROM authors a
 INNER JOIN title_authors ta
 ON a.au_id = ta.au_id
 WHERE ta.royalty_share < 1.0
INTERSECT
SELECT DISTINCT a.au_id, au_fname,
 au_lname
 FROM authors a
 INNER JOIN title_authors ta
 ON a.au_id = ta.au_id
 WHERE ta.royalty_share = 1.0;
```

代码8-42使用相关子查询列出被多个出版社出版的图书类型，结果见图8-42。

**代码8-42** 列出多个出版社共同出版的图书类型。结果见图8-42

```
SELECT DISTINCT t1.type
```

| au_id | au_fname  | au_lname  |
|-------|-----------|-----------|
| A01   | Sarah     | Buchman   |
| A02   | Wendy     | Heydemark |
| A04   | Klee      | Hull      |
| A05   | Christian | Kells     |
| A06   |           | Kellsey   |

图8-40 运行代码8-40的结果

| au_id | au_fname | au_lname  |
|-------|----------|-----------|
| A02   | Wendy    | Heydemark |
| A04   | Klee     | Hull      |
| A06   |          | Kellsey   |

图8-41 运行代码8-41的结果

```
FROM titles t1
WHERE t1.type IN
 (SELECT t2.type
 FROM titles t2
 WHERE t1.pub_id <> t2.pub_id);
```

下面代码8-42使用自联结改写为：

```
SELECT DISTINCT t1.type
 FROM titles t1
 INNER JOIN titles t2
 ON t1.type = t2.type
 AND t1.pub_id <> t2.pub_id;
```

|           |
|-----------|
| type      |
| -----     |
| biography |
| history   |

图8-42 运行代码8-42的结果

### ✓ 提示

- IN等于= ANY，见8.11节。
- NOT IN等于<> ALL (not <> ANY)，见8.11节。

- **DBMS** 为了在Microsoft Access中运行代码8-42，输入：

```
SELECT DISTINCT a.au_id, au_fname,
 au_lname
 FROM Authors AS a
 INNER JOIN title_authors AS ta1
 ON a.au_id = ta1.au_id
 INNER JOIN title_authors AS ta2
 ON a.au_id = ta2.au_id
 WHERE ta1.royalty_share < 1.0
 AND ta2.royalty_share = 1.0;
```

8

MySQL 4.0及之前的版本不支持子查询，参见8.1节中的DBMS提示。

在PostgreSQL之前的版本中，转换代码8-38至代码8-41中的浮点数为DECIMAL，见5.13节。为了运行代码8-38至代码8-41，应如下转换浮点字面量，对于代码8-38：

CAST(0.5 AS DECIMAL)

对于代码8-39：

CAST(1.0 AS DECIMAL)

对于代码8-40：

CAST(1.0 AS DECIMAL)

对于代码8-41：

CAST(1.0 AS DECIMAL) (在两处)

一些DBMS可以使用下面的语法同时检测多个值。

```
SELECT columns
 FROM table1
 WHERE (col1, col2, ..., colN) IN
 (SELECT colA, colB, ..., colN
 FROM table2);
```

检测表达式（IN的左边）是一个带括号的table1列。子查询返回数量相同的列。DBMS比较相对应列的值。后面的查询举例，可以运行在Oracle、DB2、MySQL和PostgreSQL中。

```
SELECT au_id, city, state
 FROM authors
 WHERE (city, state) IN
 (SELECT city, state
 FROM publishers);
```

结果列出所在州和城市有出版社的作者。

| au_id | city          | state |
|-------|---------------|-------|
| A03   | San Francisco | CA    |
| A04   | San Francisco | CA    |
| A05   | New York      | NY    |

287

## 8.10 使用 ALL 比较所有子查询的值

可以用关键字ALL来确定一个值是否小于或大于子查询的所有值。

使用ALL子查询比较的重要特点如下。

- 在子查询比较测试中，ALL修改了比较操作符，并且跟在=、<>、<、<=、>或>=之后，见8.8节。
- 比较操作符和ALL结合在一起，决定DBMS如何对子查询返回的值进行比较。例如< ALL，意味着小于每个子查询的返回值，而> ALL意味着大于每个子查询的返回值。
- 当ALL和<、<=、>或>=一起使用时，比较等价于找出子查询结果的最小值或最大值。< ALL意味着小于每个子查询的返回值，换句话说就是小于最小值；> ALL意味着大于每个子查询的返回值，或者说大于最大值。

表8-2所示为等价的ALL表达式和列函数。代码8-45显示如何用MAX()取代> ALL。

表8-2 ALL等值表达

| ALL表达式                   | 列 函 数                           |
|--------------------------|---------------------------------|
| < ALL( <i>subquery</i> ) | < MIN( <i>subquery values</i> ) |
| > ALL( <i>subquery</i> ) | > MAX( <i>subquery values</i> ) |

- 语义上的等价并不意味着两个查询的运行速度相同。例如查询：

```
SELECT * FROM table1
 WHERE col1 > ANY
 (SELECT MAX(col1) FROM table2);
```

通常要比以下的查询快：

```
SELECT * FROM table1
 WHERE col1 > ALL
 (SELECT col1 FROM table2);
```

要了解更多相关知识，参见8.13节。

- = ALL比较是有效的，但很少用。除非子查询返回同一个值（同时等于测试值），= ALL永远返回假。
- 子查询可以是简单或相关子查询（见8.4节）。
- 子查询SELECT子句的列表可能仅包含一个表达式或列名。
- 被比较的值必须为相同的数据类型，或隐式转换为同一种类型（见5.13节）。
- 串比较是否区分大小写取决于使用的DBMS，参见4.5节的DBMS提示。

- 子查询必须返回一列（零行或多行）值。子查询返回多于一列的值将引发错误。
- 如果子查询返回零行，ALL条件为真（这个结果与直觉相反）。

288

### ⇒ 比较所有子查询的值

在SELECT语句的WHERE子句中，输入：

`WHERE test_expr op ALL (subquery)`

`test_expr`是字面量、列名、表达式或返回唯一值的子查询，`op`是比较操作符（=、<>、<、<=、>或>=），子查询能够返回一列（零行或多行）。

如果子查询的值都满足ALL条件，或者子查询为空（零行），ALL条件的值为真。如果子查询的某个值（至少一个）不满足ALL，或任何一个值为空，ALL条件为假。

HAVING子句适用同样的语法。

`HAVING test_expr op ALL (subquery)`

代码8-43列出居住在没有出版社的城市的作者。内部查询找出所有有出版社的城市，外部查询比较每一位作者的居住城市与这些有出版社的城市，结果见图8-43。

#### 代码8-43 列出所居住城市没有出版社的作者。结果见图8-43

```
SELECT au_id, au_lname, au_fname, city
 FROM authors
 WHERE city <> ALL
 (SELECT city
 FROM publishers);
```

可以用NOT IN改写代码8-43。

```
SELECT au_id, au_lname, au_fname, city
 FROM authors
 WHERE city NOT IN
 (SELECT city FROM publishers);
```

| au_id | au_lname    | au_fname | city      |
|-------|-------------|----------|-----------|
| A01   | Buchman     | Sarah    | Bronx     |
| A02   | Heydermark  | Wendy    | Boulder   |
| A06   | Kellsey     |          | Palo Alto |
| A07   | O'Furniture | Paddy    | Sarasota  |

图8-43 运行代码8-43的结果

8

代码8-44列出价格小于所有传记的非传记图书。内部查询找出所有传记的价格。外部查询找出传记列表中的最低价格，然后查找比这个价格低的非传记，结果见图8-44。“price IS NOT NULL”这个条件是必须的，因为传记T10的价格是空值。没有这个条件，整个查询会返回零行，因为判断价格是否低于空值是不可能的（见3.14节）。

289

#### 代码8-44 列出比所有传记价格低的非传记图书。结果见图8-44

```
SELECT title_id, title_name
 FROM titles
 WHERE type <> 'biography'
 AND price < ALL
 (SELECT price
 FROM titles
 WHERE type = 'biography'
 AND price IS NOT NULL);
```

| title_id | title_name                       |
|----------|----------------------------------|
| T05      | Exchange of Platitudes           |
| T08      | Just Wait Until After School     |
| T11      | Perhaps It's a Glandular Problem |

图8-44 运行代码8-44的结果

代码8-45列出销量大于作者A06所写（或合写）图书销量的图书。内部查询使用联结来找出作者A06每本书的销量。外部查询确定这个列表的最高销量，然后确定所有超过这个销量的图书，结果见

图8-45。以防作者A06写的书的sales为空值，IS NOT NULL条件是必须的。

#### 代码8-45 列出销量超过作者A06写（或合写）图书的图书作者。结果见图8-45

```
SELECT title_id, title_name
 FROM titles
 WHERE sales > ALL
 (SELECT sales
 FROM title_authors ta
 INNER JOIN titles t
 ON t.title_id = ta.title_id
 WHERE ta.au_id = 'A06'
 AND sales IS NOT NULL);
```

可以用GROUP BY、HAVING和MAX()（而不是ALL）改写代码8-45。

```
SELECT title_id
 FROM titles
 GROUP BY title_id
 HAVING MAX(sales) >
 (SELECT MAX(sales)
 FROM title_authors ta
 INNER JOIN titles t
 ON t.title_id = ta.title_id
 WHERE ta.au_id = 'A06');
```

代码8-46在外部查询的HAVING子句使用了相关子查询，列出最高销量是该类图书平均销量两倍以上的图书类型。内部查询对外部查询定义的每个分组计算一次（每类图书一次），结果见图8-46。

#### ✓ 提示

□ ◇ ALL等价于NOT IN，见8.9节。

□ MySQL 4.0及之前版本不支持子查询，参见8.1节的DBMS提示。

**DBMS** 在较早的PostgreSQL版本，需转换代码8-46中的浮点数为DECIMAL，参见5.13节。为了运行代码8-46，转换代码中的浮点数为CAST(2.0 AS DECIMAL)。

#### 代码8-46 列出最高销量是该类图书平均销量两倍以上的图书类型。结果见图8-46

```
SELECT t1.type
 FROM titles t1
 GROUP BY t1.type
 HAVING MAX(t1.sales) >= ALL
 (SELECT 2.0 * AVG(t2.sales)
 FROM titles t2
 WHERE t1.type = t2.type);
```

| title_id | title_name                |
|----------|---------------------------|
| T05      | Exchange of Platitudes    |
| T07      | I Blame My Mother         |
| T12      | Spontaneous, Not Annoying |

图8-45 运行代码8-45的结果

| type      |
|-----------|
| biography |

图8-46 运行代码8-46的结果

## 8.11 使用 ANY 比较某些子查询的值

ANY的功能类似于ALL（见上一节），确定是否一个值等于、小于、大于子查询的结果中的某个（至少一个）值。

使用ANY子查询比较的重要特点如下。

□ 在子查询比较测试中，ANY限定了比较操作符，并且跟在=、<>、<、<=、>或>=之后，参见8.8

节。

- 比较操作符和ANY结合在一起决定DBMS如何对子查询返回的值进行比较。例如`< ANY`, 意味着小于至少一个子查询返回值, 而`> ANY`意味着大于至少一个子查询的返回值。
- 当ANY和`<`、`<=`、`>`或`>=`一起使用时, 比较等价于找出子查询结果的最大值或最小值。`< ANY`意味着小于至少一个子查询的返回值, 或者说, 至少小于最大值。`> ANY`意味着大于至少一个子查询的返回值, 或者说大于最小值; 表8-3所示为等价的ANY表达式和列函数。本章后面的代码8-49显示如何使用`MIN()`代替`> ANY`。
- `= ANY`比较等价于IN, 参见8.9节。
- 子查询可以是简单或相关子查询(见8.4节)。
- 子查询SELECT子句只能列出一个表达式或列名。
- 被比较的值必须是同一数据类型, 或隐式转换为同一类型(见5.13节)。
- 字符串比较是否区分大小写取决于使用的DBMS, 参见4.5节DBMS提示。
- 子查询必须返回一列(零行或多行)。子查询返回多于一列的值将引发错误。
- 如果子查询返回零行, ANY条件为假。

表8-3 ANY等价表达

8

| ANY表达式                          | 列 函 数                                  |
|---------------------------------|----------------------------------------|
| <code>&lt; ANY(subquery)</code> | <code>&lt; MAX(subquery values)</code> |
| <code>&gt; ANY(subquery)</code> | <code>&gt; MIN(subquery values)</code> |

### ⇒ 比较某些子查询的值

在SELECT语句的WHERE子句中, 输入:

`WHERE test_expr op ANY (subquery)`

`test_expr`是字面量、列名、表达式或是返回唯一值的子查询, `op`是比较操作符(`=`、`≠`、`<`、`<=`、`>`或`>=`), 子查询是一个返回一列(零行或多行)的子查询。如果子查询中的某个(至少一个)值满足ANY条件, 条件值为真。如果子查询中没有值满足条件、子查询为空(零行)或只包含空值, ANY条件为假。

这个语法规则同样适用于HAVING子句。

`HAVING test_expr op ANY (subquery)`

代码8-47列出居住城市有出版社的作者。内部查询找到出版社所在的所有城市, 外部查询比较每一个作者的居住城市与出版社的所在城市, 结果见图8-47。

代码8-47 列出居住在有出版社的城市的作者。结果  
见图8-47

```
SELECT au_id, au_lname, au_fname, city
 FROM authors
 WHERE city = ANY
 (SELECT city
 FROM publishers);
```

可以用IN改写代码8-47。

| au_id | au_lname | au_fname  | city          |
|-------|----------|-----------|---------------|
| A03   | Hull     | Hallie    | San Francisco |
| A04   | Hull     | Klee      | San Francisco |
| A05   | Kells    | Christian | New York      |

图8-47 运行代码8-47的结果

```
SELECT au_id, au_lname, au_fname, city
 FROM authors
 WHERE city IN
 (SELECT city FROM publishers);
```

代码8-48列出价格至少低于一部传记的非传记图书。内层查询找出所有传记价格。外部查询找到其中的最高价格，判断每本非传记图书是否低于此价格，结果见图8-48。

**代码8-48** 列出价格至少低于一部传记的非传记图书。结果见图8-48

```
SELECT title_id, title_name
 FROM titles
 WHERE type <> 'biography'
 AND price < ANY
 (SELECT price
 FROM titles
 WHERE type = 'biography');
```

不像上一节代码8-44中的ALL比较，在这里Price IS NOT NULL条件不是必需的，即便传记T10的价格为空值。DBMS并不需要判断所有价格比较为真，只要有一个比较为真即可，于是忽略了与空值的比较。

292

□ 在SQL标准中，关键字ANY和SOME是同义词。在一些DBMS中，可以用SOME代替ANY。

□ MySQL 4.0和之前的版本不支持子查询，参见8.1节中的DBMS提示。

代码8-49列出销量至少超过作者A06写（或合写）的一本书销量的图书。内部查询使用联结找出作者A06每本图书的销量。外部查询找出列表中的最低销量，然后确定每一本书的销量是否超过了这个数字，结果见图8-49。不像上一节代码8-45中的ALL比较，IS NOT NULL条件并不是必须的。

**代码8-49** 列出销量至少超过作者A06所写（或合写）的一本书销量的图书。结果见图8-49

```
SELECT title_id, title_name
 FROM titles
 WHERE sales > ANY
 (SELECT sales
 FROM title_authors ta
 INNER JOIN titles t
 ON t.title_id = ta.title_id
 WHERE ta.au_id = 'A06');
```

可以使用GROUP BY、HAVING和MIN()（而不是ANY）改写代码8-49。

```
SELECT title_id
 FROM titles
 GROUP BY title_id
 HAVING MIN(sales) >
 (SELECT MIN(sales)
 FROM title_authors ta
 INNER JOIN titles t
```

| title_id | title_name                       |
|----------|----------------------------------|
| T01      | 1977!                            |
| T02      | 200 Years of German Humor        |
| T04      | But I Did It Unconsciously       |
| T05      | Exchange of Platitudes           |
| T08      | Just Wait Until After School     |
| T09      | Kiss My Boo-Boo                  |
| T11      | Perhaps It's a Glandular Problem |

图8-48 运行代码8-48的结果

| title_id | title_name                          |
|----------|-------------------------------------|
| T02      | 200 Years of German Humor           |
| T03      | Ask Your System Administrator       |
| T04      | But I Did It Unconsciously          |
| T05      | Exchange of Platitudes              |
| T06      | How About Never?                    |
| T07      | I Blame My Mother                   |
| T09      | Kiss My Boo-Boo                     |
| T11      | Perhaps It's a Glandular Problem    |
| T12      | Spontaneous, Not Annoying           |
| T13      | What Are The Civilian Applications? |

图8-49 运行代码8-49的结果

```
ON t.title_id = ta.title_id
WHERE ta.au_id = 'A06');
```

### ✓ 提示

- = ANY 等价于 IN, 但 < ANY 不等价于 NOT IN, 如果子查询返回值 x、y 和 z:

*test\_expr* < ANY (*subquery*)

等价于:

*test\_expr* < x OR  
*test\_expr* < y OR  
*test\_expr* < z

但:

*test\_expr* NOT IN (*subquery*)

等价于:

*test\_expr* < x AND  
*test\_expr* < y AND  
*test\_expr* < z

(NOT IN 实际上等价于 < ALL。)

293

8

## 8.12 使用 EXISTS 检测存在性

到目前为止, 本章已经使用比较操作符 IN、ALL 和 ANY 对一个特定的检测值与子查询结果进行比较。 EXISTS 和 NOT EXISTS 不比较值, 而是在子查询结果中确定存在或不存在行。

存在性检测的重要特点如下。

- 存在性检测不比较值, 因此它不在表达式之前。
- 子查询可以是简单子查询或相关子查询, 但通常是后者 (见 8.4 节)。
- 子查询可以返回任意数量的行和列。
- 按照惯例, 子查询中的 SELECT 子句是用 SELECT \* 来检索出所有列。因为 EXISTS 只是简单检测满足子查询条件的行是否存在, 列出具体列名是没有必要的, 与行中的实际值是多少也并无关系。
- 所有 IN、ALL 和 ANY 查询可以用 EXISTS 或 NOT EXISTS 替代。在本节后面的例子中给出了等价的查询。
- 如果子查询返回至少一行, EXISTS 检测为真, NOT EXISTS 检测为假。
- 如果子查询返回零行, EXISTS 检测为假, NOT EXISTS 检测为真。
- 行中只包含空值的子查询算作一行。(EXISTS 检测为真, NOT EXISTS 检测为假)。
- EXISTS 检测不进行比较, 就不存在使用 IN、ALL 或 ANY 时的空值问题, 参见 8.6 节。

294

### ⇒ 存在性测试

在 SELECT 语句的 WHERE 子句中, 输入:

WHERE [NOT] EXISTS (*subquery*)

*subquery* 是返回任何行数和列数的子查询。

如果子查询返回一行或多行, EXISTS 测试的结果就为真。如果子查询返回零行, EXISTS 测试的结果就为假。加 NOT 对结果取反。

这些语法规则适用于 HAVING 子句。

HAVING [NOT] EXISTS (*subquery*)

代码8-50列出出版过传记的出版社的名称。这个查询逐个判断出版社的ID在存在性测试中的值是否为真。在这里，第一个出版社是P01 (Abatis Publishers)。DBMS查明在表titles中是否存在pub\_id等于P01，且type是传记。如果存在，Abatis Publishers就会包括在最后的结果中。DBMS针对每一个出版社的ID重复这个过程，结果见图8-50。如果想得到没有出版过传记的出版社，只需将EXISTS换成NOT EXISTS。8.9节中的代码8-33是使用IN的等价查询。

**代码8-50** 列出出版过传记的出版社的名称。结果见图8-50

```
SELECT pub_name
 FROM publishers p
 WHERE EXISTS
 (SELECT *
 FROM titles t
 WHERE t.pub_id = p.pub_id
 AND type = 'biography');
```

295

代码8-51列出没有写过书的作者，结果见图8-51。8.9节中的代码8-35是使用NOT IN的等价查询。

**代码8-51** 列出还没有写（或合写）过图书的作者。  
结果见图8-51

```
SELECT au_id, au_fname, au_lname
 FROM authors a
 WHERE NOT EXISTS
 (SELECT *
 FROM title_authors ta
 WHERE ta.au_id = a.au_id);
```

代码8-52列出所居住城市有出版社的作者，结果见图8-52。8.11节中的代码8-47是使用=ANY的等价查询。9.2节说明了如何使用INTERSECT检索两个表中相同的行，也可以用EXISTS找出交集。代码8-53列出既有作者又有出版社的城市，结果见图8-53。9.2节中的代码9-8是使用INTERSECT的等价查询。

**代码8-52** 列出所居住城市有出版社的作者。结果见图8-52

```
SELECT au_id, au_lname, au_fname, city
 FROM authors a
 WHERE EXISTS
 (SELECT *
 FROM publishers p
 WHERE p.city = a.city);
```

**代码8-53** 列出既有作者又有出版社的城市。结果见图8-53

```
SELECT DISTINCT city
 FROM authors a
 WHERE EXISTS
 (SELECT *
```

| pub_name            |
|---------------------|
| -----               |
| Abatis Publishers   |
| Schadenfreude Press |

图8-50 运行代码8-50的结果

| au_id | au_fname | au_lname    |
|-------|----------|-------------|
| ----- | -----    | -----       |
| A07   | Paddy    | O'Furniture |

图8-51 运行代码8-51的结果

| au_id | au_lname | au_fname  | city          |
|-------|----------|-----------|---------------|
| ----- | -----    | -----     | -----         |
| A03   | Hull     | Hallie    | San Francisco |
| A04   | Hull     | Klee      | San Francisco |
| A05   | Kells    | Christian | New York      |

图8-52 运行代码8-52的结果

| city          |
|---------------|
| -----         |
| New York      |
| San Francisco |

图8-53 运行代码8-53的结果

```
FROM publishers p
WHERE p.city = a.city);
```

可以使用内联结来改写这个查询。

```
SELECT DISTINCT a.city
 FROM authors a
 INNER JOIN publishers p
 ON a.city = p.city;
```

9.3节介绍如何使用EXCEPT检索存在于一个表，但不存在于另一个表的行；也可以使用NOT EXISTS发现不同的行。代码8-54列出有作者而没有出版社的城市，结果见图8-54。9.3节中的代码9-9是一个使用EXCEPT的等价查询。

代码8-54 列出有作者但没有出版社的城市。结果见图8-54

```
SELECT DISTINCT city
 FROM authors a
 WHERE NOT EXISTS
 (SELECT *
 FROM publishers p
 WHERE p.city = a.city);
```

也可以使用NOT IN改写这个查询。

```
SELECT DISTINCT city
 FROM authors
 WHERE city NOT IN
 (SELECT city
 FROM publishers);
```

或使用外联结：

```
SELECT DISTINCT a.city
 FROM authors a
 LEFT OUTER JOIN publishers p
 ON a.city = p.city
 WHERE p.city IS NULL;
```

代码8-55列出写过3本以上图书的作者，结果见图8-55。

代码8-55 列出写过3本以上图书的作者。结果见图8-55

```
SELECT au_id, au_fname, au_lname
 FROM authors a
 WHERE EXISTS
 (SELECT *
 FROM title_authors ta
 WHERE ta.au_id = a.au_id
 HAVING COUNT(*) >= 3);
```

代码8-56使用两个存在性检测列出既写过儿童图书又写过心理学图书的作者，结果见图8-56。

| city      |
|-----------|
| Boulder   |
| Bronx     |
| Palo Alto |
| Sarasota  |

图8-54 运行代码8-54的结果

| au_id | au_fname | au_lname  |
|-------|----------|-----------|
| A01   | Sarah    | Buchman   |
| A02   | Wendy    | Heydemark |
| A04   | Klee     | Hull      |
| A06   |          | Kellsey   |

图8-55 运行代码8-55的结果

**代码8-56** 列出既写过儿童图书和又写过心理学图书的作者。结果见图8-56

```

SELECT au_id, au_fname, au_lname
FROM authors a
WHERE EXISTS
 (SELECT *
 FROM title_authors ta
 INNER JOIN titles t
 ON t.title_id = ta.title_id
 WHERE ta.au_id = a.au_id
 AND t.type = 'children')
AND EXISTS
 (SELECT *
 FROM title_authors ta
 INNER JOIN titles t
 ON t.title_id = ta.title_id
 WHERE ta.au_id = a.au_id
 AND t.type = 'psychology');

```

| au_id | au_fname | au_lname |
|-------|----------|----------|
| A06   |          | Kellsey  |

图8-56 运行代码8-56的结果

298

代码8-57对表authors中列au\_id是否有重复进行了检测。如果列au\_id的值有重复，查询打印出Yes；否则，返回结果为空，结果见图8-57。au\_id是表authors的主键，显然它不重复。

**代码8-57** 表authors的au\_id列是否包含重复的值？结果见图8-57

```

SELECT DISTINCT 'Yes' AS "Duplicates?"
WHERE EXISTS
 (SELECT *
 FROM authors
 GROUP BY au_id
 HAVING COUNT(*) > 1);

```

| Duplicates? |
|-------------|
|-------------|

图8-57 运行代码8-57的结果

代码8-58是对表title\_authors使用同样的查询，它包含了重复的au\_id值，结果见图8-58。可以增加GROUP BY子句的分组列来判断是否存在多列重复。

**代码8-58** 表title\_authors的au\_id列是否包含重复的值？结果见图8-58

```

SELECT DISTINCT 'Yes' AS "Duplicates?"
WHERE EXISTS
 (SELECT *
 FROM title_authors
 GROUP BY au_id
 HAVING COUNT(*) > 1);

```

| Duplicates? |
|-------------|
| Yes         |

图8-58 运行代码8-58的结果

**✓ 提示**

- 可以使用COUNT(\*)来判断子查询是否返回至少一行，但COUNT(\*)（通常情况下）没有EXISTS效率高。一旦判断出子查询是否返回了行，DBMS就退出EXISTS子查询的处理过程，但COUNT(\*)要求DBMS完成整个子查询。这个查询等价于代码8-52，但速度较慢。

```

SELECT au_id, au_lname, au_fname,
 city
 FROM authors a
 WHERE
 (SELECT COUNT(*)

```

```
FROM publishers p
WHERE p.city = a.city) > 0;
```

- 尽管在EXISTS子查询中，SELECT \*是最普通的SELECT子句形式，但如果DBMS优化器认为需要为EXISTS子查询构建完整的临时表，这时可以使用SELECT column或SELECT constant\_value来加速查询的运行。
- 尽管在本节的DBMS提示中，某些DBMS子查询可以使用SELECT COUNT(\*)，但在子查询的SELECT子句中使用聚合函数要特别小心。例如，在代码8-59中的存在性测试，因为COUNT(\*)总是返回一行（值为零），于是结果总是为真。因为并没有ID为XXX的出版社存在，所以图8-59所示的结果是错误的。
- **DBMS** 为了在Microsoft Access中运行代码8-55、代码8-57和代码8-58，将SELECT \*变为SELECT 1。另外，在代码8-57的外部查询中加入FROM authors子句。

为了在Oracle中运行代码8-57和代码8-58，在外部查询中加入FROM DUAL，参见5.1节的DBMS提示。要在DB2中运行代码8-55、代码8-57和代码8-58，将SELECT \*变为SELECT 1。另外，在代码8-57和代码8-58的外部查询中加入子句FROM SYSIBM.SYSDUMMY1，参见5.1节。例如，将代码8-57变为：

```
SELECT DISTINCT 'Yes' AS
 "Duplicates?"
 FROM SYSIBM.SYSDUMMY1
 WHERE EXISTS
 (SELECT 1
 FROM title_authors
 GROUP BY au_id
 HAVING COUNT(*) > 1);
```

为了在MySQL中运行代码8-57，在外部清单中加入子句FROM authors，在代码8-58的外部查询中加入子句FROM title\_authors。MySQL 4.0和之前的版本不支持子查询，参见8.1节中的DBMS提示。

为了在PostgreSQL中运行代码8-55、代码8-57和代码8-58，将SELECT \*变为SELECT 1。

**代码8-59 在子查询的SELECT子句中使用聚合函数要小心。结果见图8-59**

```
SELECT pub_id
 FROM publishers
 WHERE EXISTS
 (SELECT COUNT(*)
 FROM titles
 WHERE pub_id = 'XXX');
```

| pub_id |
|--------|
| -----  |
| P01    |
| P02    |
| P03    |
| P04    |

图8-59 运行代码8-59的结果

## 8.13 比较等价查询

看完本章及上一章，读者可以使用各种方法实现等价查询（语法不同，语义相同）。为了说明这点，本书已经给出6种语义相同的方法。代码8-60的每条语句都列出至少写（或合写）过一本图书的作者，结果见图8-60。

**代码8-60** 这6个查询是语义等价的，它们都列出至少写（或合写）过一本图书的作者。结果见图8-60

```

SELECT DISTINCT a.au_id
 FROM authors a
 INNER JOIN title_authors ta
 ON a.au_id = ta.au_id;

SELECT DISTINCT a.au_id
 FROM authors a, title_authors ta
 WHERE a.au_id = ta.au_id;

SELECT au_id
 FROM authors a
 WHERE au_id IN
 (SELECT au_id
 FROM title_authors);

SELECT au_id
 FROM authors a
 WHERE au_id = ANY
 (SELECT au_id
 FROM title_authors);

SELECT au_id
 FROM authors a
 WHERE EXISTS
 (SELECT *
 FROM title_authors ta
 WHERE a.au_id = ta.au_id);

SELECT au_id
 FROM authors a
 WHERE 0 <
 (SELECT COUNT(*)
 FROM title_authors ta
 WHERE a.au_id = ta.au_id);

```

| au_id |
|-------|
| ----- |
| A01   |
| A02   |
| A03   |
| A04   |
| A05   |
| A06   |

图8-60 代码的6个查询都返回这个结果

前两个查询（内联结）的运行速度相同。第3个到第6个查询使用了子查询，最后一个可能是速度最慢的。因为一旦遇到满足条件的值，DBMS会停止执行别的子查询。但最后一个语句的子查询在返回真假值前，需要和所有行进行比较。DBMS优化器运行内联结的速度和最快的子查询语句相同。

你可能很喜欢这类编程的灵活性，但设计DBMS优化器的人却未必，他们要考虑表达一个查询所有可能的方式，搞清楚哪个是效率最高的，在内部将查询修改为最优形式。（他们的毕生事业就是解决优化问题）。如果DBMS的优化器没有缺陷，那它运行代码8-60中6个程序的速度是相同的。但情况往往不是这样，必须在DBMS上测试来看哪个程序运行得最快。

301

#### ✓ 提示

- 要使用巨大的表（超过10 000或100 000行）来比较查询运行速度，这样才能明显地发现速度和存储的差异。
- DBMS提供了计量查询效率差异的工具。表8-4和表8-5列出时间查询命令和它们的执行计划。

表8-4 时间查询

| DBMS       | 命 令                    |
|------------|------------------------|
| Access     | 无                      |
| SQL Server | SET STATISTICS TIME ON |
| Oracle     | SET TIMING ON          |
| DB2        | db2batch               |
| MySQL      | mysql命令行工具默认状态下打印执行时间  |
| PostgreSQL | \timing                |

表8-5 显示查询执行计划

| DBMS       | 命 令                  |
|------------|----------------------|
| Access     | 无                    |
| SQL Server | SET SHOWPLAN_TEXT ON |
| Oracle     | EXPLAIN PLAN         |
| DB2        | EXPLAIN 或 db2expln   |
| MySQL      | EXPLAIN              |
| PostgreSQL | EXPLAIN              |

### SQL调优

学习了基本的SQL，下一步就是调优SQL语句以便它们能高效运行，这意味着要了解DBMS优化器。执行调优包含了一些与平台无关的原则，但最有效的调优方法要考虑具体DBMS的特点。调优问题不是本书的讨论范围，互联网上有大量讨论组和文章——搜索“调优”（或“执行”、“优化”）和所用的DBMS的名字。

Peter Gulutzan 和 Trudy Pelzer写的*SQL Performance Tuning* ( Addison-Wesley ) 是一本入门的好书，它包含了8种DBMS；Dan Tow的*SQL Tuning* ( O'Reilly ) 包含了Microsoft SQL Server、Oracle和DB2。在amazon.com上查找这两本书，点击“类似书目”(Similar Items)链接就能发现其他调优书籍。

 顾第2章，其中介绍的集合理论是关系模型的基础。数学集合是不改变的，但数据库集合是动态的——随着时间增长、收缩和改变。本章包含以下SQL集合操作符，它们将两个SELECT语句的结果合并成一个。

- UNION返回两个查询返回的所有行，但会删掉重复行。
- INTERSECT返回两个查询返回的所有共同行（也就是说，两个查询都检索到的行）。
- EXCEPT返回第一个查询中存在，但第二个查询中不存在的所有行，删掉重复行。

303

## 9.1 使用 UNION 合并行

UNION操作将两个查询返回的结果合并成一个结果。（这个操作不同于将两个表的列合并的联结）。UNION从结果中去掉重复的行，UNION ALL不去掉重复的行。

联合操作是简单的，但它们有一些限制。

- 两个查询的SELECT子句列出的列（列名、计算表达式、聚合函数等）必须数量相同。
- 两个查询对应的列必须有相同的顺序。
- 对应的列必须是相同的数据类型，或隐式转换为相同的类型。
- 如果对应列的名称相同，结果中使用列名。如果对应列的名称不同，要由DBMS决定结果中的列名。大多数DBMS采用UNION语句第一个查询的列名作为结果的列名。如果想对结果中的列重命名，在第一个查询中使用AS子句，参见4.2节。
- ORDER BY子句只能出现在UNION语句最后的查询中。对最后合并的结果进行排序，因为结果的列名要由DBMS决定，最简单的方法是使用相对列位置来确定排序，参见4.4节。
- GROUP BY和HAVING只能用于单独的查询，它们不能用于影响最终结果。

304

### ⇒ 合并行

输入：

```
select_statement1
UNION [ALL]
select_statement2;
```

*select\_statement1*和*select\_statement2*都是SELECT语句。在两个语句中列的数量和顺序必须一致，对应列的数据类型必须是兼容的。除非用ALL明确，否则重复的行会被去掉。

代码9-1列出作者和出版社所处同一州的州名。默认情况下，UNION去掉结果中重复的行，结果见图9-1。

除了包含关键字ALL以外，代码9-2和代码9-1是相同的，于是结果中包含了所有的行；重复行没有被去掉，结果见图9-2。

代码9-3列出所有作者的名字和出版社的名称。第一个查询中的AS子句命名结果中的列名。ORDER BY子句使用相对的列名位置而不是列名来排序结果，结果见图9-3。

**代码9-1** 列出作者和出版社所处同一州的州名。结果见图9-1

```
SELECT state FROM authors
UNION
SELECT state FROM publishers;
```

**代码9-2** 列出作者和出版社所处同一州的州名（包括重复）。结果见图9-2

```
SELECT state FROM authors
UNION ALL
SELECT state FROM publishers;
```

**代码9-3** 列出所有作者的名字和出版社的名称。结果见图9-3

```
SELECT au_fname || ' ' || au_lname AS "Name"
FROM authors
UNION
SELECT pub_name
FROM publishers
ORDER BY 1 ASC;
```

| state |
|-------|
| ----- |
| NULL  |
| CA    |
| CO    |
| FL    |
| NY    |

图9-1 运行代码9-1的结果

| state |
|-------|
| ----- |
| NY    |
| CO    |
| CA    |
| CA    |
| NY    |
| CA    |
| FL    |
| NY    |
| CA    |
| NULL  |
| CA    |

图9-2 运行代码9-2的结果

| Name                |
|---------------------|
| -----               |
| Kellsey             |
| Abatis Publishers   |
| Christian Kells     |
| Core Dump Books     |
| Hallie Hull         |
| Klee Hull           |
| Paddy O'Furniture   |
| Sarah Buchman       |
| Schadenfreude Press |
| Tenterhooks Press   |
| Wendy Heydemark     |

图9-3 运行代码9-3的结果

代码9-4扩展了代码9-3，且定义了一个额外的列Type来明确行来自那个表。WHERE条件用来检索纽约州的作者和出版社，结果见图9-4。

306

代码9-5对代码9-4增加了第三个查询，来检索在纽约州出版的图书，结果见图9-5。

除了列出作者、出版社、图书的数量而不是它们的名字，代码9-6与代码9-5类似，结果见图9-6。

#### 代码9-4 以类型及名字排序，列出纽约州所有作者和出版社的名字。结果见图9-4

```

SELECT
 'author' AS "Type",
 au_fname || ' ' || au_lname AS "Name",
 state
FROM authors
WHERE state = 'NY'
UNION
SELECT
 'publisher',
 pub_name,
 state
FROM publishers
WHERE state = 'NY'
ORDER BY 1 ASC, 2 ASC;

```

| Type      | Name              | state |
|-----------|-------------------|-------|
| author    | Christian Kells   | NY    |
| author    | Sarah Buchman     | NY    |
| publisher | Abatis Publishers | NY    |

图9-4 运行代码9-4的结果

#### 代码9-5 以类型及名字排序，列出纽约州所有作者和出版社的名字，以及在纽约州出版的图书。结果见图9-5

```

SELECT
 'author' AS "Type",
 au_fname || ' ' || au_lname AS "Name"
FROM authors
WHERE state = 'NY'
UNION
SELECT
 'publisher',
 pub_name
FROM publishers
WHERE state = 'NY'
UNION
SELECT
 'title',
 title_name
FROM titles t
INNER JOIN publishers p
ON t.pub_id = p.pub_id
WHERE p.state = 'NY'
ORDER BY 1 ASC, 2 ASC;

```

| Type      | Name                       |
|-----------|----------------------------|
| author    | Christian Kells            |
| author    | Sarah Buchman              |
| publisher | Abatis Publishers          |
| title     | 1977!                      |
| title     | How About Never?           |
| title     | Not Without My Faberge Egg |
| title     | Spontaneous, Not Annoying  |

图9-5 运行代码9-5的结果

#### 代码9-6 以类型排序，列出纽约州作者和出版社的数量，以及在纽约州出版图书的数量。结果见图9-6

```

SELECT
 'author' AS "Type",
 COUNT(au_id) AS "Count"
FROM authors
WHERE state = 'NY'
UNION

```

```

SELECT
 'publisher',
 COUNT(pub_id)
FROM publishers
WHERE state = 'NY'
UNION
SELECT
 'title',
 COUNT(title_id)
FROM titles t
INNER JOIN publishers p
 ON t.pub_id = p.pub_id
WHERE p.state = 'NY'
ORDER BY 1 ASC;

```

| Type      | Count |
|-----------|-------|
| author    | 2     |
| publisher | 1     |
| title     | 4     |

图9-6 运行代码9-6的结果

307

308

在代码9-7中，再一次用到了5.14节中的代码5-30，但没有用CASE来改变图书价格和模拟if-then逻辑，这次使用了多重UNION查询，结果见图9-7。

### UNION交换律

按照理论，SELECT语句（表）在UNION中出现的顺序对于运行速度没有影响，但在实际应用中，DBMS可能运行左边的代码要快于右边的代码：

|                                                              |                                                              |
|--------------------------------------------------------------|--------------------------------------------------------------|
| small_table1<br>UNION<br>small_table2<br>UNION<br>big_table; | small_table1<br>UNION<br>big_table<br>UNION<br>small_table2; |
|--------------------------------------------------------------|--------------------------------------------------------------|

原因是优化器合并中间结果和除去重复行的方式不同。不同的DBMS结果不同，需要测试。

9

**代码9-7** 将历史书的价格提高10%，心理学书的价格提高20%，其他图书的价格保持不变。结果见图9-7

```

SELECT title_id, type, price,
 price * 1.10 AS "New price"
 FROM titles
 WHERE type = 'history'
UNION
SELECT title_id, type, price, price * 1.20
 FROM titles
 WHERE type = 'psychology'
UNION
SELECT title_id, type, price, price
 FROM titles
 WHERE type NOT IN ('psychology', 'history')
 ORDER BY type ASC, title_id ASC;

```

#### ✓ 提示

- UNION是可交换的操作。A UNION B和B UNION A是相同的。

|     | title_id   | type  | price | New price |
|-----|------------|-------|-------|-----------|
| T06 | biography  | 19.95 | 19.95 |           |
| T07 | biography  | 23.95 | 23.95 |           |
| T10 | biography  | NULL  | NULL  |           |
| T12 | biography  | 12.99 | 12.99 |           |
| T08 | children   | 10.00 | 10.00 |           |
| T09 | children   | 13.95 | 13.95 |           |
| T03 | computer   | 39.95 | 39.95 |           |
| T01 | history    | 21.99 | 24.19 |           |
| T02 | history    | 19.95 | 21.95 |           |
| T13 | history    | 29.99 | 32.99 |           |
| T04 | psychology | 12.99 | 15.59 |           |
| T05 | psychology | 6.95  | 8.34  |           |
| T11 | psychology | 7.99  | 9.59  |           |

图9-7 运行代码9-7的结果

□ 在SQL标准中，INTERSECT的优先级别要比UNION和EXCEPT高，但不同DBMS可能采用不同的优先顺序。使用括号来明确含有多个集合操作查询的运算顺序，参见5.3节。

□ 在组合条件可以实现的情况下，不要使用UNION。

```
SELECT DISTINCT * FROM mytable
 WHERE col1 = 1 OR col2 = 2;
```

通常快于：

```
SELECT * FROM mytable
 WHERE col1 = 1;
UNION
SELECT * FROM mytable
 WHERE col2 = 2;
```

□ 如果在一条语句中将UNION和UNION ALL混合使用，要使用括号来明确运算顺序。例如，下面的两个语句：

```
SELECT * FROM table1
UNION ALL
(SELECT * FROM table2
UNION
SELECT * FROM table3);
```

和

```
(SELECT * FROM table1
UNION ALL
SELECT * FROM table2)
UNION
SELECT * FROM table3;
```

第一条语句去掉table2和table3合并后的重复行，但不去掉这个结果与table1合并后的重复行。第二个语句包含了table1和table2合并后的重复行，但在与table3合并后，去掉结果中的重复行，于是ALL对最后的结果没有影响。

□ 对于UNION操作，DBMS按内部顺序来确定和去掉重复行；因此即便没有使用ORDER BY子句，UNION的结果可能也会排序。UNION ALL因为不需要去掉重复行，所以不排序。排序计算的消耗很大——若使用UNION ALL能实现，就不要使用UNION。

□ **DBMS** 在Microsoft Access和Microsoft SQL Server环境中，使用+来连接字符串（见5.4节）。  
要运行代码9-3至代码9-5，将连接表达式变为：

```
au_fname + '' + au_lname
```

在MySQL环境中，使用CONCAT()来连接串（见5.4节）。要运行代码9-3至代码9-5，将连接表达式变为：

```
CONCAT(au_fname, ' ', au_lname)
```

在较早的PostgreSQL版本环境中，将代码9-7中的浮点数转换为DECIMAL，参见5.13节。要运行代码9-7，将价格计算变为：

```
price * CAST((1.10) AS DECIMAL)
price * CAST((1.20) AS DECIMAL)
```

## 9.2 使用 INTERSECT 查找相同行

INTERSECT操作将两个查询的结果中相同的行并成一个结果。交集与合并一样有限制，参见9.1节。

### » 查找相同行

输入：

```
select_statement1
INTERSECT
select_statement2;
```

*select\_statement1*和*select\_statement2*都是SELECT语句。两个查询中列的数量和顺序必须相同，对应列的数据类型必须兼容。结果中重复的行被去掉。

代码9-8使用INTERSECT列出既有作者又有出版社的城市，结果见图9-8。

**代码9-8 使用 INTERSECT列出既有作者又有出版社的城市。结果见图9-8**

```
SELECT city
 FROM authors
INTERSECT
SELECT city
 FROM publishers;
```

| city          |
|---------------|
| New York      |
| San Francisco |

图9-8 运行代码9-8的结果

310

9

### ✓ 提示

- INTERSECT是可交换的操作。A INTERSECT B 和B INTERSECT A是相同的。
- 在SQL标准中INTERSECT的优先级别要比UNION和EXCEPT高，但不同的DBMS可能采用不同的优先顺序。使用括号来明确含有混合集合操作查询的运算顺序，参见5.3节。
- 可以认为UNION是逻辑的OR，INTERSECTION是逻辑的AND，参见4.6节。例如想知道供应商A或B都供应哪些产品，输入：

```
SELECT product_id
 FROM vendor_a_product_list
UNION
SELECT product_id
 FROM vendor_b_product_list;
```

如果想知道哪些产品供应商A和B都供应。输入：

```
SELECT product_id
 FROM vendor_a_product_list
INTERSECT
SELECT product_id
 FROM vendor_b_product_list;
```

- 如果DBMS不支持INTERSECT，可以用INNER JOIN或EXISTS子查询。以下这些语句和代码9-8（内联结）是等价的。

```
SELECT DISTINCT authors.city
 FROM authors
INNER JOIN publishers
 ON authors.city =
 publishers.city;
```

或 (EXISTS子查询):

```
SELECT DISTINCT city
 FROM authors
 WHERE EXISTS
 (SELECT *
 FROM publishers
 WHERE authors.city =
 publishers.city;)
```

- DBMS** Microsoft Access、Microsoft SQL Server 2000和MySQL不支持INTERSECT。要运行代码9-8，用前面提示给出的等价查询。(Microsoft SQL Server 2005和之后的版本支持INTERSECT)。

311

### 9.3 使用 EXCEPT 查找不同行

EXCEPT操作，也称作差操作，将两个查询中只属于第一个查询的行作为结果。对比INTERSECT 和 EXCEPT，A INTERSECT B包含表A和表B重复的行；A EXCEPT B包含属于表A但在表B中不重复的行。差操作和合并操作一样有限制，参见9.1节。

⇒ 查找不同的行

输入：

```
select_statement1
EXCEPT
select_statement2;
```

*select\_statement1*和*select\_statement2*是SELECT语句。两条语句的列数量与顺序必须相同，对应列的数据类型必须兼容。结果中去掉重复的行。

312

代码9-9使用EXCEPT列出有作者但没有出版社的城市，结果见图9-9。

代码9-9 列出有作者但没有出版社的城市。结果见图  
9-9

```
SELECT city
 FROM authors
EXCEPT
SELECT city
 FROM publishers;
```

✓ 提示

与UNION和INTERSECT不同，EXCEPT不可交换项：

A EXCEPT B与B EXCEPT A是不同的。

在SQL标准中INTERSECT的优先级别要比UNION和EXCEPT高，但不同的DBMS可能采用不同的优先顺序。使用括号来明确含有混合集合操作查询的运算顺序，参见5.3节。

如果组合条件可以实现，就不要使用EXCEPT。

```
SELECT * FROM mytable
 WHERE col1 = 1 AND NOT col2 = 2;
```

通常要比下面代码运行得快：

| city      |
|-----------|
| Boulder   |
| Bronx     |
| Palo Alto |
| Sarasota  |

图9-9 运行代码9-9的结果

```
SELECT * FROM mytable
 WHERE col1 = 1;
EXCEPT
SELECT * FROM mytable
 WHERE col2 = 2;
```

- 如果DBMS不支持EXCEPT，可以使用外联结、NOT EXISTS子查询、NOT IN子查询来取代它。以下语句和代码9-9（外联结）是等价的。

```
SELECT DISTINCT a.city
 FROM authors a
 LEFT OUTER JOIN publishers p
 ON a.city = p.city
 WHERE p.city IS NULL;
```

或者是用NOT EXISTS子查询：

```
SELECT DISTINCT city
 FROM authors
 WHERE NOT EXISTS
 (SELECT *
 FROM publishers
 WHERE authors.city =
 publishers.city);
```

或者是用NOT IN子查询：

```
SELECT DISTINCT city
 FROM authors
 WHERE city NOT IN
 (SELECT city
 FROM publishers);
```

- **DBMS** Microsoft Access、Microsoft SQL Server 2000和MySQL不支持EXCEPT。要运行代码9-9，用前面提示给出的等价查询。（Microsoft SQL Server 2005和之后版本支持EXCEPT）。在Oracle中，EXCEPT操作符是MINUS。要运行代码9-9，输入：

```
SELECT city FROM authors
MINUS
SELECT city FROM publishers;
```

**本**书已经介绍了如何使用SELECT检索和分析表中的数据。本章将介绍如何使用SQL语句来修改表中的数据。

- INSERT语句用来向表中增加行。
- UPDATE语句用来改变表中现有行的值。
- DELETE语句用来删除表中的行。

这些语句不返回结果，DBMS通常会打印一条消息，说明语句是否运行成功及修改涉及的行数。

要了解语句对表的影响，使用SELECT语句，如SELECT \* FROM *table*。

与SELECT只能访问数据不同，这些语句能修改数据，所以只有在数据库管理员授权的情况下，这些命令才能运行。

315

## 10.1 显示表结构

要使用INSERT、UPDATE或DELETE，就必须了解所要修改表的列，包括：

- 表中列的顺序。
- 每列的名称。
- 每列的数据类型。
- 列是否为主键（或主键的一部分）。
- 列中的值是否必须唯一。
- 列是否允许空值。
- 每列的默认值（如果存在）。
- 表和列的约束（见第11章）。

表2-3至表2-7给出了示例数据库的表结构，可以使用说明数据库对象的DBMS工具来获得这些信息。本部分介绍如何使用这些工具来显示当前数据库的表结构。

### » 在Microsoft Access中显示表结构

在Access 2003或之前的版本，按F11键显示Database窗口，单击Tables（在Objects下面），单击列表中的某个表，然后单击工具栏中的Design，在Design View中打开表（图10-1）。  
或者

在Access 2007或之后版本，按F11键来显示Navigation面板（在左边），然后用鼠标右键单击表名，从弹出的快捷菜单中选择Design View。（如果在Navigation面板中看不到表，单击面板顶部的菜单并选

择Object Type，然后再一次单击菜单并选择Tables)。

#### ⇒ 在Microsoft SQL Server中显示表结构

(1) 在SQL Server 2000中，启动SQL Query Analyzer或交互式osql命令行工具(见1.3节)。

或者在SQL Server 2005和之后版本中，启动SQL Server Management Studio或交互式sqlcmd命令行工具(见1.3节)。

osql和sqlcmd命令显示一闪而过的几页内容。选择Query→Results in Grid(或Results to Grid)，使用图形工具则更加容易。

(2) 输入sp\_help table。

table是表名。

(3) 在SQL Query Analyzer或SQL Server Management Studio中，选择Query→Execute或按F5键(图10-2)。

或者在osql或sqlcmd中，按回车键；然后输入go，再按回车键。

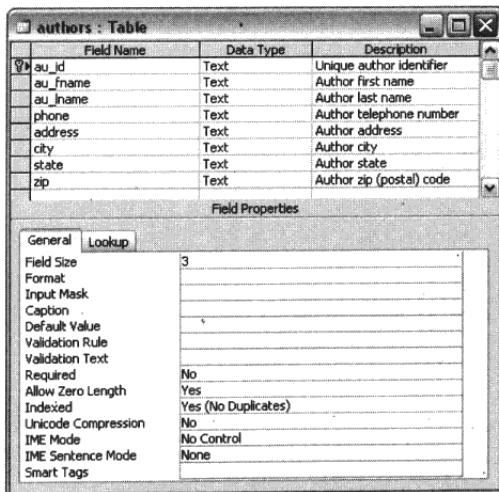


图10-1 在Microsoft Access中显示表结构

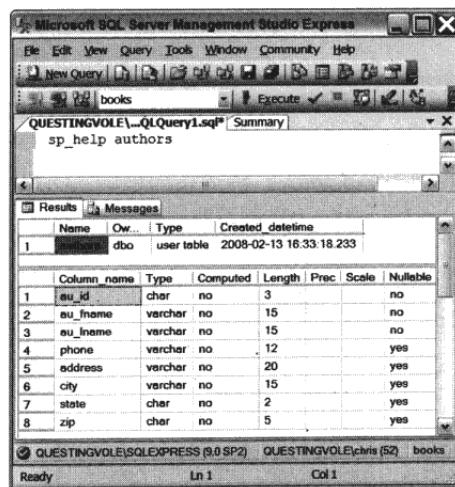


图10-2 在Microsoft SQL Server中显示表结构

#### ⇒ 在Oracle中显示表结构

(1) 启动交互式sqlplus命令行工具(见1.4节)。

(2) 输入describe table，然后按回车键(图10-3)。

table是表名。

#### ⇒ 在IBM DB2中显示表结构

(1) 启动Command Center或db2命令行处理器(见1.5节)。

(2) 输入describe table table，然后按回车键(或在Command Center按Ctrl+Enter快捷键)(图10-4)。

table是表名。

```
SQL> describe authors;
Name Null? Type
----- -----
AU_ID NOT NULL CHAR(3)
AU_FNAME NOT NULL VARCHAR2(15)
AU_LNAME NOT NULL VARCHAR2(15)
PHONE VARCHAR2(12)
ADDRESS VARCHAR2(20)
CITY VARCHAR2(15)
STATE CHAR(2)
ZIP CHAR(5)
```

图10-3 在Oracle中显示表结构

```
db2 => describe table authors;
 Data type
Column name Schema Data type name Column Length Scale Nulls
----- -----
AU_ID SYS10M CHARACTER 3 0 No
AU_FNAME SYS10M VARCHAR 15 0 No
AU_LNAME SYS10M VARCHAR 15 0 No
PHONE SYS10M VARCHAR 12 0 Yes
ADDRESS SYS10M VARCHAR 20 0 Yes
CITY SYS10M VARCHAR 15 0 Yes
STATE SYS10M CHARACTER 2 0 Yes
ZIP SYS10M CHARACTER 5 0 Yes

8 record(s) selected.
```

图10-4 在IBM DB2中显示表结构

**⇒ 在MySQL中显示表结构**

- (1) 启动交互式mysql命令行工具（见1.6节）。
  - (2) 输入describe table, 然后按回车键（图10-5）。
- table*是表名。

**⇒ 在PostgreSQL中显示表结构**

- (1) 启动交互式psql命令行工具（见1.7节）
  - (2) 输入\dt *table*, 然后按回车键（图10-6）。
- table*是表名。注意不能用分号结束这个命令。

```
mysql> describe authors;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
au_id	char(3)	NO	PRI		
au_fname	varchar(15)	NO			
au_lname	varchar(15)	NO			
phone	varchar(12)	YES		NULL	
address	varchar(20)	YES		NULL	
city	varchar(15)	YES		NULL	
state	char(2)	YES		NULL	
zip	char(5)	YES		NULL	
+-----+-----+-----+-----+-----+-----+
8 rows in set (0.00 sec)
```

图10-5 在MySQL中显示表结构

```
books=# \d authors
Table "public.authors"
 Column | Type | Modifiers
-----+-----+-----+
au_id | character(3) | not null
au_fname | character varying(15) | not null
au_lname | character varying(15) | not null
phone | character varying(12) |
address | character varying(20) |
city | character varying(15) |
state | character(2) |
zip | character(5) |

Indexes:
"pk_authors" PRIMARY KEY, btree (au_id)
```

图10-6 在PostgreSQL中显示表结构

**✓ 提示**

- 列出表的列名及它们出现的顺序，但不包含表中的数据，输入：

```
SELECT * FROM table WHERE 1 = 2;
```

*table*是表名，1 = 2意味着条件永远为假。

- 要全面了解列的知识，参见2.1节。

关于键，参见2.2节和2.3节。

关于数据类型，参见3.4节。

修改表的结构，参见第11章。

- DBMS 表10-1所示为当前数据库中表的命令和查询。

表10-1 列出数据库表

| DBMS       | 命令或查询              |
|------------|--------------------|
| Access     | Database窗口(按[F11]) |
| SQL Serve  | sp_tables          |
| Oracle     | SELECT * FROM TAB; |
| DB2        | LIST TABLES;       |
| MySQL      | SHOW TABLES;       |
| PostgreSQL | \d                 |

## 10.2 使用 INSERT 插入行

INSERT语句向表中插入新行。本节介绍如何使用各种形式的INSERT来进行如下操作。

- 按列位置插入行（INSERT VALUES）。
- 按列名插入行（INSERT VALUE）。
- 将一个表中的行插入到另一个表（INSERT SELECT）。

INSERT的主要特点如下。

- 按位置插入行的情况下，按照列在表中的先后顺序将有序的值插入到新行（见10.2.1节）。按列名插入行的情况下，指定每个值要插入到新行的列名（见10.2.2节）。若按指定列名来插入行，即便改变了表列顺序或添加了新列，SQL代码仍然可以运行。
- 使用INSERT VALUES可以向表中插入确定的值。使用INSERT SELECT可以向表中插入另一个表中的行。
- INSERT VALUES向表中插入一行。INSERT SELECT向表中插入零行或多行。
- 每一个被插入的值必须与相应的列具有相同的数据类型，或可被隐式转换为相同的数据类型（见5.13节）。
- 要保证引用完整性，一个插入的外键值必须包含空值（如果允许）或被外键引用的主键或唯一键列中的键值，参见2.2节和2.3节。
- 插入的值不能违反检查约束，参见11.9节。
- 表达式不能引发计算错误（如溢出或被零除的错误）。
- 2.1节曾提到表中行的顺序并不重要，且无法控制它的物理位置，新添加的行可能出现在表的任何位置。

### ⇒ 按列位置插入行

输入：

```
INSERT INTO table
VALUES(value1, value2, ..., valueN);
```

table是插入行的表名。value1,value2,...,valueN是用带有括号、逗号分隔的值或表达式，它们提供新行每列的值。值的数量必须等于table中列的数量，值的排列顺序必须和table中列名的排列顺序一致。DBMS依据table中列的位置关系向列中插入值。在table中新行的第一列的位置插入value1,

第二列的位置插入 *value2*, 以此类推。

320 该语句向 *table*插入一行 (代码10-1)。

**代码10-1** 这条INSERT语句通过列出与表authors的列顺序一致的值, 来向表authors添加一个新行。

结果见图10-7

```
INSERT INTO authors
VALUES(
 'A08',
 'Michael',
 'Polk',
 '512-953-1231',
 '4028 Guadalupe St',
 'Austin',
 'TX',
 '78701');
```

| au_id | au_fname  | au_lname    | phone        | address              | city             | state | zip   |
|-------|-----------|-------------|--------------|----------------------|------------------|-------|-------|
| A01   | Sarah     | Buchman     | 718-496-7223 | 75 West 205 St       | Bronx            | NY    | 10468 |
| A02   | Wendy     | Heydemark   | 303-986-7020 | 2922 Baseline Rd     | Boulder          | CO    | 80303 |
| A03   | Hallie    | Hull        | 415-549-4278 | 3800 Waldo Ave, #14F | San Francisco CA | CA    | 94123 |
| A04   | Klee      | Hull        | 415-549-4278 | 3800 Waldo Ave, #14F | San Francisco CA | CA    | 94123 |
| A05   | Christian | Kells       | 212-771-4680 | 114 Horatio St       | New York         | NY    | 10014 |
| A06   |           | Kellsey     | 650-836-7128 | 390 Serra Mall       | Palo Alto        | CA    | 94305 |
| A07   | Paddy     | O'Furniture | 941-925-0752 | 1442 Main St         | Sarasota         | FL    | 34236 |
| A08   | Michael   | Polk        | 512-953-1231 | 4028 Guadalupe St    | Austin           | TX    | 78701 |
| A09   | Irene     | Bell        | 415-225-4689 | 810 Throckmorton Ave | Mill Valley      | CA    | 94941 |
| A10   | Dianne    | Weston      | 312-998-0020 | 1937 N. Clark St     | Chicago          | IL    | 60614 |
| A11   | Max       | Allard      | 212-502-0955 | NULL                 | NULL             | NULL  | NULL  |

图10-7 authors表在运行了代码10-1至代码10-4后有了4个新行

### » 按列名插入行

输入:

```
INSERT INTO table
(column1, column2, ..., columnN)
VALUES(value1, value2, ..., valueN);
```

*table*是插入行的表的名称。*column1, column2, ..., columnN*是表*table*中带有括号、用逗号分隔的列名列表。*value1, value2, ..., valueN*是带有括号、用逗号分隔的值或提供值的表达式，并在行中按列名顺序排列。

值的数量必须与列表中列的数量相同，值的排列必须与列名的顺序一致。DBMS依据对应的位置将值添加到列中。*value1*被插入新行的*column1*, *value2*被插入新行的*column2*, 以此类推。被忽略的列被指定为默认值或空值。

该语句向表*table*中插入一行。

按表中顺序列出列名就更清楚了（代码10-2），但按任何顺序列出都是允许的（代码10-3）。在任何情况下，VALUES子句中值的顺序必须与列名的顺序一致。

如果只是向一些指明的列提供值，可以忽略其他的列名（代码10-4）。

**代码10-2** 这条INSERT语句通过列出与列名列表中列顺序一致的值，来向authors表添加一个新行。结果见图10-7

```
INSERT INTO authors(
 au_id,
 au_fname,
 au_lname,
 phone,
 address,
 city,
 state,
 zip)
VALUES(
 'A09',
 'Irene',
 'Bell',
 '415-225-4689',
 '810 Throckmorton Ave',
 'Mill Valley',
 'CA',
 '94941');
```

**代码10-3** 如果与表中的列顺序一致，就没有必要列出列名。这里，重新排列列名和其对应的值。结果见图10-7

```
INSERT INTO authors(
 zip,
 phone,
 address,
 au_lname,
 au_fname,
 state,
 au_id,
 city)
VALUES(
 '60614',
 '312-998-0020',
 '1937 N. Clark St',
 'Weston',
 'Dianne',
 'IL',
 'A10',
 'Chicago');
```

**代码10-4** 忽略作者地址信息的列名和值，增加一个新作者行。DBMS向忽略的列自动加入空值。结果见图10-7

```
INSERT INTO authors(
 au_id,
 au_fname,
 au_lname,
 phone)
VALUES(
 'A11',
 'Max',
 'Allard',
 '212-502-0955');
```

如果忽略一列，DBMS能够依据列的定义给出一个值。DBMS将向这些列插入默认值（如果已定义）或空值（如果允许）。如果忽略的列既没有默认值也不允许空值，DBMS将无法插入行，并显示错误信息。在这种情况下，VALUES子句等价于VALUES('A11', 'Max', 'Allard', '212-502-0955', NULL, NULL, NULL)。要了解更多有关指定默认值和允许空值的知识，参见11.4节和11.5节。

图10-7显示代码10-1至代码10-4运行后，表authors中的新行。

⇒ 将一个表中的行插入到另一个表

输入：

```
INSERT INTO table
[(column1, column2, ..., columnN)]
subquery;
```

*table*是要插入行的表的名称。*column1, column2, ..., columnN*是表*table*可选的带有括号、用逗号分隔的列的名称。*subquery*是返回一个SELECT语句，返回插入到表*table*中的行。

*subquery*结果中列的数量必须等于表*table*中列的数量或列出的列的数量。DBMS忽略*subquery*结果中的列名而使用列的位置。*subquery*结果中的第一列用于填入表*table*的第一列或*column1*，以此类推。忽略的列被指定为默认值或空值。

该语句向表*table*插入零行或多行。

323 本节其余的例子使用new\_publishers表（图10-8），它用来展示INSERT SELECT如何运行。new\_publishers与publishers的表结构相同，只是作为新行的来源，它本身不会被INSERT操作改变。

| pub_id | pub_name             | city        | state | country       |
|--------|----------------------|-------------|-------|---------------|
| P05    | This is Pizza? Press | New York    | NY    | USA           |
| P06    | This is Beer? Press  | Toronto     | ON    | Canada        |
| P07    | This is Irony? Press | London      | NULL  | United Kindom |
| P08    | This is Fame? Press  | Los Angeles | CA    | USA           |

图10-8 这个名为new\_publishers的表被用于代码10-5至代码10-7。new\_publishers和publishers的表结构相同

代码10-5将表new\_publishers中位于Los Angeles的出版社添加到表publishers。清单忽略了列的列表，于是DBMS使用publishers中列的位置而不是列名插入值。这个语句向publishers插入一行，结果见图10-9。

代码10-6将表new\_publishers中非美国的出版社添加到表publishers。INSERT和SELECT子句的列名相同，但没有必要匹配，因为DBMS依据的是列的位置而不是SELECT返回的列名。该语句向publishers插入两行，结果见图10-9。

324 SELECT返回的结果为空（零行）是合法的。代码10-7将new\_publishers中名为XXX的出版社添加到publishers。因为new\_publishers和publishers的结构相同，可以使用SELECT \*而不是列名列表。因为没有名为XXX的出版社，该语句没有向publishers插入行，结果见图10-9。

**代码10-5** 将表 new\_publishers 中位于 Los Angeles 的出版社插入到表 publishers。结果见图10-9

```
INSERT INTO publishers
SELECT
 pub_id,
 pub_name,
 city,
 state,
 country
FROM new_publishers
WHERE city = 'Los Angeles';
```

**代码10-6** 将表 new\_publishers 中非美国的出版社添加到表 publishers。结果见图 10-9

```
INSERT INTO publishers(
 pub_id,
 pub_name,
 city,
 state,
 country)
SELECT
 pub_id,
 pub_name,
 city,
 state,
 country
FROM new_publishers
WHERE country <> 'USA';
```

**代码10-7 将表new\_publishers中名为XXX的出版社添加到表publishers。该语句对表publishers没有影响。结果见图10-9**

```
INSERT INTO publishers(
 pub_id,
 pub_name,
 city,
 state,
 country)
SELECT *
FROM new_publishers
WHERE pub_name = 'XXX';
```

图10-9显示代码10-5至代码10-7运行后的表publishers。

#### ✓ 提示

- 第一次向表中插入行叫做填充表。
- 为了谨慎起见，插入行前可以在目标表的临时副本表里测试INSERT语句，参见11.10节和11.11节。
- 可以通过视图插入行，参见13.3节。

| pub_id | pub_name             | city          | state | country        |
|--------|----------------------|---------------|-------|----------------|
| P01    | Abatis Publishers    | New York      | NY    | USA            |
| P02    | Core Dump Books      | San Francisco | CA    | USA            |
| P03    | Schadenfreude Press  | Hamburg       | NULL  | Germany        |
| P04    | Tenterhooks Press    | Berkeley      | CA    | USA            |
| P06    | This is Beer? Press  | Toronto       | ON    | Canada         |
| P07    | This is Irony? Press | London        | NULL  | United Kingdom |
| P08    | This is Fame? Press  | Los Angeles   | CA    | USA            |

10

图10-9 代码10-5至代码10-7运行后的表publishers

- 使用事务，必须在最后的INSERT语句修改完表之后，使用COMMIT语句。要了解更多有关事务的知识，参见第14章。

如果table1和table2具有兼容的结构，可以使用以下语句将table2中所有的行插入到table1。

```
INSERT INTO table1
SELECT * FROM table2;
```

- **DBMS** 对于一些DBMS，INSERT语句中的关键字INTO是可选的，但为了代码的可移植性还是应该加上。

默认情况下，MySQL（不恰当地）对INSERT或UPDATE的一些无效值进行转换并发出警告，而不是引发错误，除非使用事务并能够取消操作。例如插入9/0，MySQL会试着插入空值，而不是发出被零除的错误，即便此列禁止空值。如果在SMALLINT列中插入超出范围的值999 999，MySQL将插入32 767（最大的SMALLINT值）并发出警告。MySQL提供ERROR\_FOR\_DIVISION\_BY\_ZERO、STRICT\_ALL\_TABLES、STRICT\_TRANS\_TABLES等，来正确处理无效或缺失的值时。

325

326

对于所有DBMS，查阅文档来了解DBMS如何对插入到自动产生唯一行标识数据类型的列值进行处理（见3.12节）。

### 10.3 使用UPDATE更新行

UPDATE语句改变表中已存在的行的值。可以使用UPDATE改变：

- 表中的所有行；
- 表中的特定行。

更新行，需要明确：

- 要更新的表；
- 要更新的列名及它们的新值；
- 确定要更新的行的可选搜索条件。

UPDATE的重要特点如下。

- UPDATE采用可选的WHERE子句来明确要更新的行。若没有WHERE子句，UPDATE将改变表中所有的行。
- UPDATE是不安全的，因为它很容易错误地忽略WHERE子句（更新所有行）或将WHERE搜索条件搞错（更新错了行）。在运行实际的UPDATE前，运行使用同样WHERE子句的SELECT子句是明智的。使用SELECT \*列出运行UPDATE语句时DBMS将要改变的所有行，或使用SELECT COUNT(\*)列出将要改变的行数。
- 每个被更新的值必须和它的列具有相同的数据类型，或能够隐式转换为同一数据类型（见5.13节）。
- 为了保持引用完整性，当试图UPDATE一个被外键指向的键值时，必须定义DBMS的默认行为，参见11.7节中的提示。
- 更新的值不能违反检查约束，参见11.9节。
- 表达式不能引发算术错误（如溢出或被零除的错误）。
- 2.1节曾提到表中行的顺序并不重要及无法控制行的物理位置，因此更新行可能改变行在表中的位置。

327

#### ⇒ 更新行

输入：

```
UPDATE table
 SET column = expr
 [WHERE search_condition];
```

*table*是要更新的表的名字。*column*是表*table*中包含的要修改行的列名。*expr*是字面量、表达式或返回唯一值的子查询。*expr*返回的值取代*column*中已存在的值。要修改多个列中的值，在SET子句中输入一个逗号分隔的*column = expr*表达式列表。可以采用任何顺序排列*column = expr*表达式。

*search\_condition*明确要更新行应满足的条件。*search\_condition*条件可以是WHERE条件（比较操作符LIKE、BETWEEN、IN和IS NULL，见第4章）或子查询条件（比较操作符IN、ALL、ANY和EXISTS，见第8章），及它们使用AND、OR和NOT的结合。如果WHERE子句被忽略，*table*中的所有行都要更新。

代码10-8将titles中所有行的contract值修改为零。没有WHERE子句，就会指示DBMS更新所有行

的列contract。该语句更新了13行，结果见图10-10。

代码10-9使用算术表达式和WHERE条件来使历史书的价格翻倍。该语句更新了3行，结果见图10-10。

#### 代码10-8 将每行contract的值改为零。结果见图10-10

```
UPDATE titles
SET contract = 0;
```

#### 代码10-9 将历史书的价格翻倍。结果见图10-10

```
UPDATE titles
SET price = price * 2.0
WHERE type = 'history';
```

##### ✓ 提示

- 使用CASE更新价格的技巧如下。

```
UPDATE titles
SET price = price * CASE type
 WHEN 'history' THEN 1.10
 WHEN 'psychology' THEN 1.20
 ELSE 1
END;
```

代码10-10更新心理学图书的type和pages列的值。使用逗号分隔的`column = expr`表达式，一条SET语句就可以更新多个列。(不要在最后一个表达式的后面使用逗号)。这个语句更新了3行，结果见图10-10。

代码10-11使用子查询和聚合函数将超过平均销量的图书销量减半。该语句更新了两行，结果见图10-10。

可以依据存储在别的表里的数据更新指定表。代码10-12使用嵌套子查询更新Sarah Buchman写(或合写)的所有书的出版日期。该语句更新了3行，结果见图10-10。

#### 代码10-10 将心理学图书的类型修改为自学，并将页数改为空值。结果见图10-10

```
UPDATE titles
SET type = 'self help',
 pages = NULL
WHERE type = 'psychology';
```

#### 代码10-11 将超过平均销量的图书的销量减半。结果见图 10-10

```
UPDATE titles
SET sales = sales * 0.5
WHERE sales >
 (SELECT AVG(sales)
 FROM titles);
```

#### 代码10-12 将Sarah Buchman所有书的出版时间更新为2003-01-01。结果见图10-10

```
UPDATE titles
SET pubdate = DATE '2003-01-01'
WHERE title_id IN
 (SELECT title_id
 FROM title_authors
 WHERE au_id IN
```

```
(SELECT au_id
 FROM authors
 WHERE au_fname = 'Sarah'
 AND au_lname = 'Buchman'));
```

假定出版社Abatis Publishers(P01)兼并了Tenterhooks Press(P04), 于是所有Tenterhooks Press的图书都由Abatis Publishers出版。代码10-13按照自下而上方式将titles中的出版社ID从P04改为P01。WHERE子查询检索出Tenterhooks Press的pub\_id。DBMS使用这个pub\_id从表titles检索出版社是Tenterhooks Press的图书。最后, DBMS使用SET子查询返回的值来更新表titles中的特定行。因为子查询和未加限制的比较操作符一起使用, 它们必须是返回唯一值的标量子查询(也就是一行一列的结果), 见8.8节。代码10-13更新了5行, 结果见图10-10。

**330** 图10-10显示表titles运行了代码10-8至代码10-13后的情况。每个代码更新了与其他代码不同列(或多个列)的值。每列中被更新的值用粗体显示。

**代码10-13 将所有出版社为Tenterhooks Press的图书更新为Abatis Publishers出版社。结果见图10-10**

```
UPDATE titles
 SET pub_id =
 (SELECT pub_id
 FROM publishers
 WHERE pub_name = 'Abatis Publishers')
 WHERE pub_id =
 (SELECT pub_id
 FROM publishers
 WHERE pub_name = 'Tenterhooks Press');
```

| title_id | title_name                          | type      | pub_id | pages | price | sales  | pubdate    | contract |
|----------|-------------------------------------|-----------|--------|-------|-------|--------|------------|----------|
| T01      | 1977!                               | history   | P01    | 107   | 43.98 | 566    | 2003-01-01 | 0        |
| T02      | 200 Years of German Humor           | history   | P03    | 14    | 39.90 | 9566   | 2003-01-01 | 0        |
| T03      | Ask Your System Administrator       | computer  | P02    | 1226  | 39.95 | 25667  | 2000-09-01 | 0        |
| T04      | But I Did It Unconsciously          | self help | P01    | NULL  | 12.99 | 13001  | 1999-05-31 | 0        |
| T05      | Exchange of Platitudes              | self help | P01    | NULL  | 6.95  | 100720 | 2001-01-01 | 0        |
| T06      | How About Never?                    | biography | P01    | 473   | 19.95 | 11320  | 2000-07-31 | 0        |
| T07      | I Blame My Mother                   | biography | P03    | 333   | 23.95 | 750100 | 1999-10-01 | 0        |
| T08      | Just Wait Until After School        | children  | P01    | 86    | 10.00 | 4095   | 2001-06-01 | 0        |
| T09      | Kiss My Boo-Boo                     | children  | P01    | 22    | 13.95 | 5000   | 2002-05-31 | 0        |
| T10      | Not Without My Faberge Egg          | biography | P01    | NULL  | NULL  | NULL   | NULL       | 0        |
| T11      | Perhaps It's a Glandular Problem    | self help | P01    | NULL  | 7.99  | 94123  | 2000-11-30 | 0        |
| T12      | Spontaneous, Not Annoying           | biography | P01    | 507   | 12.99 | 100001 | 2000-08-31 | 0        |
| T13      | What Are The Civilian Applications? | history   | P03    | 802   | 59.98 | 10467  | 2003-01-01 | 0        |

图10-10 运行代码10-8至代码10-13之后的表titles。更新的值用粗体显示

### ✓ 提示

❑ DBMS使用引用列更新之前的值求值SET或WHERE子句中的表达式。

```
UPDATE mytable
 SET col1 = col1 * 2,
 col2 = col1 * 4,
 col3 = col2 * 8
 WHERE col1 = 1
 AND col2 = 2;
```

对于满足WHERE条件的行，DBMS将col1修改为2，col2修改为4（ $1 \times 4$ ，不是 $2 \times 4$ ），及col3修改为16（ $2 \times 8$ ，不是 $4 \times 8$ ）。

这个求值模式可交换兼容列的值。

```
UPDATE mytable
 SET col1 = col2,
 col2 = col1;
```

（这个技巧无法在MySQL中应用。）

- 在更新行之前要保持谨慎，可在目标表的临时副本中测试UPDATE语句，参见11.10节和11.11节。
- 可以通过视图更新行，参见13.3节。
- 如果使用事务，必须在最后的UPDATE语句更新完表之后，使用COMMIT语句。要了解更多有关事务的知识，参见第14章。
- SQL:2003引入了MERGE语句作为在一条语句中结合运用多个UPDATE和INSERT操作的简便方法。这些操作非正式地叫做upserts。Oracle和DB2支持MERGE。MySQL有简化的MERGE的变型REPLACE INTO和INSERT...ON DUPLICATE KEY UPDATE。
- Microsoft Access中的日期值忽略关键字DATE，以#而不是括号包围值。运行代码10-12，要将日期值变为#2003-01-01#。Microsoft Access的SET子句不支持标量子查询。运行代码10-13，要将UPDATE语句分为两条语句：一条语句从表publishers用SELECT列出出版社Abatis Publishers的pub\_id，另一条语句用这个pub\_id来更新表titles中所有Tenterhooks Press图书的pub\_id。然后连续地运行这些语句（在宿主语言中，如Visual Basic或C#），使用第一条语句的结果作为第二条语句的输入。  
Microsoft SQL Server和DB2中的日期值忽略关键字DATE。为了运行代码10-12，将日期值变为'2003-01-01'。

MySQL 4.1和之后的版本支持子查询，但MySQL不允许在子查询的FROM子句和更新目标中都使用同一个表（在这个例子中，是表titles），因此无法运行代码10-11。之前的MySQL版本不支持子查询，无法运行代码10-11至代码10-13，要运行这些代码，参见8.1节中的DBMS提示。对于MySQL，还可参见10.2节中的DBMS提示。

为了在PostgreSQL的较早版本上运行代码10-9和代码10-11，将浮点数转变为DECIMAL（见5.13节）。

转变采用，对于代码10-9：

```
CAST(2.0 AS DECIMAL)
```

对于代码10-11：

```
CAST(0.5 AS DECIMAL)
```

对于所有DBMS，要查阅文档来了解DBMS如何更新相应数据类型支持自动产生唯一行标识符的列的值，参见3.12节。

331

10

332

## 10.4 使用DELETE删除行

DELETE语句从表中删除行。可以使用DELETE删除：

表中所有行；

表中特定的行。

要删除行，需明确：

删除行所属的表；

删除行的可选搜索条件。

DELETE的重要特点如下。

与INSERT和UPDATE不同，DELETE删除整个行，不需要指出列名。

DELETE删除表中的行，但不会删除表结构。即便删除了表中所有的行，表还是存在的。如果要删除表结构（及所有相关数据、索引等），参见11.3节。

DELETE用可选的WHERE子句来明确要删除的行。没有WHERE子句，DELETE删除表中所有行。

DELETE是有风险的，因为错误地忽略WHERE子句（删除所有行）或搞错了WHERE搜索条件（删错行）是很有可能发生的。在实际运行DELETE语句前，运行有相同的WHERE子句的SELECT语句是好办法。使用SELECT \*列出运行DELETE，DBMS将删除的所有行，或使用SELECT COUNT(\*)显示将被删除的行的数量。

为了保证引用完整性，当试图DELETE一个外键值指向的键值时，可以定义DBMS的自动行为，参见11.7节中的提示。

表达式不可引发算术错误（如溢出或被零除的错误）。

2.1节介绍了表中行的顺序并不重要，无法控制行的物理位置，所以删除行会使表中行的顺序发生变化。

### ⇒ 删除行

输入：

```
DELETE FROM table
[WHERE search_condition];
```

*table*是删除行的所属表。

*search\_condition*明确删除行所需要满足的条件。*search\_condition*可以是WHERE条件（比较操作符LIKE、BETWEEN、IN和IS NULL，见第4章）或子查询条件（比较操作符IN、ALL、ANY和EXISTS，见第8章），及它们使用AND、OR和NOT的组合。如果忽略WHERE子句，表*table*中所有行被删除。

在下面的例子中，忽略了引用完整性约束——在实际工作中，当然不会这样做。

代码10-14删除表royalties中所有行。没有WHERE子句，DBMS将删除所有行。该语句删除了13行，结果见图10-11。

代码10-14 删除表royalties中所有行。结果见图  
10-11

```
DELETE FROM royalties;
```

| <i>title_id</i> | <i>advance</i> | <i>royalty_rate</i> |
|-----------------|----------------|---------------------|
| -----           | -----          | -----               |

图10-11 运行代码10-14的结果

代码10-15中的WHERE子句指示DBMS从authors表删除姓为Hull的作者。该语句删除了两行，结果见图10-12。

**代码10-15** 从表authors删除姓为Hull的作者。结果见图10-12

```
DELETE FROM authors
WHERE au_lname = 'Hull';
```

| au_id | au_fname  | au_lname    | phone        | address          | city      | state | zip   |
|-------|-----------|-------------|--------------|------------------|-----------|-------|-------|
| A01   | Sarah     | Buchman     | 718-496-7223 | 75 West 205 St   | Bronx     | NY    | 10468 |
| A02   | Wendy     | Heydemark   | 303-986-7020 | 2922 Baseline Rd | Boulder   | CO    | 80303 |
| A05   | Christian | Kells       | 212-771-4680 | 114 Horatio St   | New York  | NY    | 10014 |
| A06   |           | Kellsey     | 650-836-7128 | 390 Serra Mall   | Palo Alto | CA    | 94305 |
| A07   | Paddy     | O'Furniture | 941-925-0752 | 1442 Main St     | Sarasota  | FL    | 34236 |

图10-12 运行代码10-15的结果

可以删除依据其他表中数据确定的行。代码10-16使用子查询从表title\_authors中删除出版社P01或P04出版的图书。该语句删除了12行，结果见图10-13。

**代码10-16** 从表title\_authors中删除出版社为P01或P04的图书。结果见图10-13

```
DELETE FROM title_authors
WHERE title_id IN
(SELECT title_id
FROM titles
WHERE pub_id IN ('P01', 'P04'));
```

### ✓ 提示

- 在删除行前要保持格外谨慎，可以用目标表的临时副本来自测试DELETE语句，参见11.10节和11.11节。
- 可以通过视图DELETE行，参见13.13节。
- 如果使用事务，必须在最后的DELETE语句执行完之后使用COMMIT语句。要了解更多有关事务的知识，参见第14章。
- 可以使用NOT EXISTS或NOT IN子查询引用另一个表中不存在的行来从表中删除行。（删除没有引用或违背引用完整性的行很有用）。以下的语句从titles表中删除所有其出版社不在表publishers中的行。

```
DELETE FROM titles
WHERE NOT EXISTS
(SELECT * FROM publishers
WHERE publishers.pub_id =
titles.pub_id);
```

或

```
DELETE FROM titles
WHERE pub_id NOT IN
(SELECT pub_id FROM publishers);
```

| title_id | au_id | au_order | royalty_share |
|----------|-------|----------|---------------|
| T02      | A01   | 1        | 1.00          |
| T03      | A05   | 1        | 1.00          |
| T07      | A02   | 1        | 0.50          |
| T07      | A04   | 2        | 0.50          |
| T13      | A01   | 1        | 1.00          |

图10-13 运行代码10-16的结果

- **DBMS** 在一些DBMS中，DELETE语句中的FROM关键字是可选的，但为了可移植性应该包含。MySQL 4.1和之后的版本支持子查询，能够运行代码10-16。MySQL较早的版本不支持子查询，无法运行。要运行这些代码，参见8.1节的DBMS提示。  
对于MySQL，还要参见10.2节的DBMS提示。

### 删除表中的行

如果想删除表中的所有行，TRUNCATE语句比DELETE要快。TRUNCATE不是SQL标准的一部分，但Microsoft SQL Server、Oracle、MySQL和PostgreSQL都支持它。TRUNCATE的运行像是没有WHERE子句的DELETE语句，删除表中的所有行。TRUNCATE比DELETE的速度快且使用的系统资源少，因为TRUNCATE并不扫描整个表，也不记录变化的事务日志（见第14章）。如果使用TRUNCATE，代价是有错误就无法恢复到变化前（回滚）。语法是：

```
TRUNCATE TABLE table;
```

*table*是要删除行的表。要了解TRUNCATE，在DBMS文档中搜索*truncate*。

要在DB2中使用TRUNCATE删除表，运行带有REPLACE选项的LOAD，使用零字节文件作为输入。

# 创建、更改和删除表



**在**一些DBMS中，可以利用互动式、图形化工具来创建、管理表及表的属性，如列定义和约束。本章将介绍如何使用SQL程序来完成这些任务。

- 使用CREATE TABLE语句创建表。
- 使用ALTER TABLE语句更改现存表的结构。
- 使用DROP TABLE删除表和表中的数据。
- 使用CREATE TEMPORARY TABLE语句创建不再使用时会被DBMS自动删除的表。
- 使用CREATE TABLE AS语句利用已存在的表创建新表。

这些语句不会返回结果，但DBMS会打印一条说明语句是否成功执行的信息。要了解语句对表的实际影响，使用10.1节介绍的命令查看表结构。

这些语句修改数据库对象和数据，所以只有得到数据库管理员的授权才能执行。

337

## 11.1 创建表

在写SQL代码之前，数据库设计者要花费很多时间来规范表和定义关系及约束。要创建用于实际工作的数据库中的表，学习比第2章介绍的数据库设计和关系模型理论更多的知识。

2.1节所讲的数据库是围绕表来组织的。对于用户或SQL编程人员，数据库就是一个或多个表的集合（除了表没别的）。要创建表，需定义以下内容。

- 表名
- 列名
- 列的数据类型
- 列的默认值
- 约束

11

表名或列名的命名必须符合SQL的标识符规则，参见3.3节。列的数据类型是字符型、数字型、日期型或其他数据类型，参见3.4节。默认值是指在没有明确给定值的情况下，列将采用的值。约束定义是否允许空值、键、允许的值等属性。

可以用CREATE TABLE语句创建新表，一般的语法如下。

```
CREATE TABLE table
(
 column1 data_type1 [col_constraints1],
 column2 data_type2 [col_constraints2],
```

```

...

columnN data_typeN [col_constraintsN]

[, table_constraint1]

[, table_constraint2]

...

[, table_constraintM]

);

```

**[338]** 每个列定义包含列名、数据类型，以及可选的一个或多个列约束的列表。可选的表约束列表放在最后的列定义之后。按照惯例，本书将每个列定义和表约束写在它们自己的行里。

表11-1 约束

| 约 束         | 约 束 说 明             |
|-------------|---------------------|
| NOT NULL    | 阻止向列中插入空值           |
| PRIMARY KEY | 设置表的主键列             |
| FOREIGN KEY | 设置表的外键列             |
| UNIQUE      | 阻止向列中插入重复的值         |
| CHECK       | 使用逻辑（布尔）表达式限制插入列中的值 |

## 11.2 理解约束

约束定义了列允许值的规则（表11-1）。DBMS使用这些规则自动强制性地保证数据库信息的完整性。

约束有两个优点。

- 列约束是列定义的一部分，它设置作用于列的条件。
- 表约束是有别于列定义并加强于表中多个列的条件。可以在一个表约束中包含多个列。

依据使用环境，可以将约束区分为列约束或表约束。例如，如果主键包含一列，那么既可以定义为列约束也可以定义为表约束。如果主键有两个或更多列，就必须使用表约束。

为约束指定名称可方便管理。可以使用ALTER TABLE语句修改或删除约束。约束名是可选的，但很多SQL编程人员和数据库设计人员对所有约束命名。尽管不给NOT NULL约束命名很常见，但还是应该给其他类型的约束命名（尽管本书在一些例子里没有这样做）。

如果不显式命名约束，DBMS将自动产生并分配约束的名称。系统分配的名称经常包含随机的字符串并且使用起来很麻烦，所以应该用CONSTRAINT子句分配自定义的约束名。约束名还会出现在警告、错误信息和日志里，这也是自主给约束命名的原因。

### » 定义约束

在定义约束之前，输入：

CONSTRAINT constraint\_name

**[340]** constraint\_name是约束名并且是有效的SQL标识符。约束名在一个表中必须是唯一的。

## 11.3 使用CREATE TABLE创建新表

本节讲述如何使用最简洁的CREATE TABLE语句创建新表，随后讲述如何向CREATE TABLE加入列和

表约束。

### » 创建新表

输入：

```
CREATE TABLE table
(
 column1 data_type1,
 column2 data_type2,
 ...
 columnN data_typeN
);
```

*table*是要创建的新表名称。*column1*,*column2*,...,*columnN*是表*table*中的列名。至少要创建一个列。

*data\_type1*,*data\_type2*,...,*data\_typeN*明确每个相应列的SQL数据类型。数据类型明确列的适用长度、范围或精确度，见3.4节和第3章随后的部分。

表名在数据库中必须唯一，列名在表中必须唯一。

代码11-1创建简单数据库表titles。

代码11-2创建简单数据库表title\_authors。

341

#### 代码11-1 创建简单数据库表titles

```
CREATE TABLE titles
(
 title_id CHAR(3) ,
 title_name VARCHAR(40) ,
 type VARCHAR(10) ,
 pub_id CHAR(3) ,
 pages INTEGER ,
 price DECIMAL(5,2),
 sales INTEGER ,
 pubdate DATE ,
 contract SMALLINT
);
```

#### 代码11-2 创建简单数据库表title\_authors

```
CREATE TABLE title_authors
(
 title_id CHAR(3) ,
 au_id CHAR(3) ,
 au_order SMALLINT ,
 royalty_share DECIMAL(5,2)
);
```

### ✓ 提示

11

- 要了解CREATE TABLE语句的运行结果，并查看表的结构，可以使用10.1节中介绍的命令。
- 如果要创建的表的名称在数据库中已经存在，DBMS会产生错误。为了防止错误地覆盖表，SQL需要在创建新表时，使用DROP TABLE显式地删除同名的表，参见11.13节。
- 新创建的表是空的（零行）。为了在表中添加数据，使用INSERT语句，参见10.2节。
- 默认情况下，列允许空值。如果不允许空值，参见11.4节。
- 修改已存在的表结构，参见11.12节。
- 要利用已存在表的结构和数据创建表，参见11.11节。
- **DBMS** Microsoft SQL Server不支持DATE型数据。为了运行代码11-1，要将列pubdate的数据类型改为DATETIME。

当创建表时，MySQL会自动改变CHAR和VARCHAR的类型。例如，长度不超过4的VARCHAR类型列会被改为CHAR类型。

342

## 11.4 使用 NOT NULL 禁止空值

列的空值属性决定行能否包含空值——或者说，列的值是必须的还是可选的。3.14节说明了空值及其影响，在这里回顾其基本内容。

- 空值不是值，意味着没有输入值。
- 空值表示缺少、不知道、不适用的值。列price中的空值并不意味着该项没有价格或价格为零，它意味着不知道价格或还没有输入。
- 空值和零(0)、空格或空串('')不同。
- 空值没有数据类型且可以被插入到任何允许空值的列。
- 在SQL语句中，关键字NULL代表空值。

在定义是否允许空值的约束时，需要重点考虑如下方面。

- 空值约束通常是列约束不是表约束，参见11.2节。
- 在CREATE TABLE列定义中使用关键字NOT NULL来定义列的空值约束。
- 通常情况下，空值会将查询、插入和更新变得复杂，要避免允许空值。
- 列禁止空值可以确保用户必须在列中输入数据，这有助于保持数据完整性。如果不允许空值的列包含了空值，DBMS无法插入或更新行。
- 其他约束，比如PRIMARY KEY约束，不能用于允许空值的列。
- 空值影响外键的引用完整性检查，参见11.7节。
- 如果INSERT插入行在允许空值的列没有值，那么DBMS提供空值（除非有DEFAULT子句存在）；参见10.2节和11.5节；
- 不论列定义为何种数据类型或默认值，只要列允许空值，用户都可以显式输入NULL。
- 如果没有明确NOT NULL约束，列默认可以接受空值。

### ⇒ 确定列可否允许空值

向CREATE TABLE的列定义添加列约束：

```
[CONSTRAINT constraint_name]
 NOT NULL
```

NOT NULL禁止列中出现空值。如果列没有空值约束，列可以为空值。要了解CREATE TABLE的一般语法，参见11.1节。

CONSTRAINT子句是可选的，*constraint\_name*是列的空值约束名，参见11.2节。

代码11-3创建示例数据库表authors，在一些列禁止空值。没有地址和电话是常见的，于是允许这些列包含空值。

### 代码11-3 创建示例数据库表authors。忽略空值约束默认允许空值

```
CREATE TABLE authors
(
 au_id CHAR(3) NOT NULL,
 au_fname VARCHAR(15) NOT NULL,
 au_lname VARCHAR(15) NOT NULL,
 phone VARCHAR(12) ,
 address VARCHAR(20) ,
 city VARCHAR(15) ,
```

```
state CHAR(2)
zip CHAR(5)
);
```

注意在姓和名的列禁止空值。如果作者的名只有一个词（如作者A06, Kellsey），程序在au\_lname插入名字，在au\_fname插入空串（''）。或者列au\_fname允许空值，名字只有一个词的作者在au\_fname列插入空值；或者列au\_fname和au\_lname都允许空值，同时增加一个检查约束，要求两列中至少有一列包含非空值或非空串。数据库设计者要在创建表前考虑这类问题。

大多数DBMS可以用NULL关键字（没有NOT）来确定允许空值。代码11-4创建示例表titles。

344

#### 代码11-4 创建示例数据库表titles并显式地为每列分配空值约束

```
CREATE TABLE titles
(
 title_id CHAR(3) NOT NULL,
 title_name VARCHAR(40) NOT NULL,
 type VARCHAR(10) NULL ,
 pub_id CHAR(3) NOT NULL,
 pages INTEGER NULL ,
 price DECIMAL(5,2) NULL ,
 sales INTEGER NULL ,
 pubdate DATE NULL ,
 contract SMALLINT NOT NULL
);
```

#### ✓ 提示

- 要了解CREATE TABLE语句运行结果，使用10.1节介绍的命令来查看表结构。
- 向表插入行时，必须显式地提供非空列的值（如果没有默认值）。例如，对于代码11-3产生的表authors，最简洁的INSERT语句如下。

```
INSERT INTO authors(
 au_id,
 au_fname,
 au_lname)
VALUES(
 'A08',
 'Michael',
 'Polk');
```

11

DBMS自动向表authors中那些没有出现在INSERT列表的列（phone、address等）分配空值，参见10.2节。

- 当向行插入空值时，不要将关键字NULL放入引号里；DBMS会将其解释为插入字符串'NULL'，而不是空值。

□ 参见4.10节、5.15节和5.16节。

- **DBMS** Microsoft SQL Server不支持DATE数据类型。为了运行代码11-4，将列pubdate的数据类型变为DATETIME。

Oracle将空串（''）作为空值，参见3.14节的DBMS提示。

DB2在空值约束中不接受单独的关键字NULL（没有NOT）。为了运行代码11-4，除去所有不是NOT NULL的空值约束。

DB2 和 MySQL 不接受命名的非空约束。要在NOT NULL列定义中略去CONSTRAINT

345

*constraint\_name*。

本书涉及的DBMS的空值约束是可选的（默认情况下允许空值），但其他的DBMS或许不同。

对于所有的DBMS，查阅文档来了解DBMS如何处理相应数据类型支持自动产生唯一行标识符的空值约束，参见3.13节。

## 11.5 使用 DEFAULT 确定默认值

如果在插入行时缺少列的值，DBMS给列分配的值默认确定，参见10.2节。定义默认值时，要重点考虑以下方面。

- 默认作用于一个列。
- 在CREATE TABLE列定义中使用关键字DEFAULT定义默认值。
- 默认值可以是基于常数的任何表达式。
- 默认值必须和它的列具有相同的数据类型，或隐式转换成列的数据类型，参见5.13节。
- 列的长度必须足够存储默认值。
- 如果插入行没有给出列的值，应用列的默认值。如果列定义中没有DEFAULT，默认值是空值。
- 如果列没有默认值，同时声明NOT NULL，插入行时需要显式提供列的值（见10.2节）。如果在这种情况下没有提供显式的值，一些DBMS会拒绝插入行，而另一些则会依据列的数据类型自动分配默认值。例如，MySQL给数字型，不能为空且缺少显式默认值的列分配零。

346

### ⇒ 指定列的默认值

添加以下子句到CREATE TABLE的列定义：

DEFAULT *expr*

*expr*是一个基于常数的表达式，比如字面量、内置的函数、数学表达式或NULL。如果没有指定默认值，就认为是NULL。要全面了解CREATE TABLE语法，参见11.1节。

代码11-5向示例数据库表titles的一些列分配默认值。列title\_id和pub\_id是NOT NULL且没有默认值，于是必须在INSERT语句中为它们提供显式的值。pages子句的DEFAULT NULL和忽略DEFAULT是相同的。pubdate和contract默认值可采用比字面量更为复杂的表达式。

代码11-5 为示例数据库表titles的一些列设置默认值

```
CREATE TABLE titles
(
 title_id CHAR(3) NOT NULL ,
 title_name VARCHAR(40) NOT NULL DEFAULT '' ,
 type VARCHAR(10) DEFAULT 'undefined' ,
 pub_id CHAR(3) NOT NULL ,
 pages INTEGER DEFAULT NULL ,
 price DECIMAL(5,2) NOT NULL DEFAULT 0.00 ,
 sales INTEGER DEFAULT NULL ,
 pubdate DATE DEFAULT CURRENT_DATE,
 contract SMALLINT NOT NULL DEFAULT (3*7)-21
);
```

347

代码11-6显示能够向表titles插入行的最简洁INSERT语句（如代码11-5所创建的）。图11-1显示插入的行，默认值特别标出。title\_name的默认值是空串（''），看不到。

**代码11-6** DBMS向这个INSERT语句没有给定值的列插入默认值。对于没有指定默认值的列，DBMS插入空值，结果见图11-1

```
INSERT INTO titles(title_id, pub_id) VALUES('T14','P01');
```

| title_id | title_name | type      | pub_id | pages | price | sales | pubdate    | contract |
|----------|------------|-----------|--------|-------|-------|-------|------------|----------|
| T14      |            | undefined | P01    | NULL  | 0.00  | NULL  | 2005-02-21 | 0        |

图11-1 代码11-6向表titles插入这个行

### ✓ 提示

- 要了解CREATE TABLE语句的运行结果，通过10.1节介绍的命令查看表结构。
- **DBMS** Microsoft Access不允许DEFAULT子句中有算数表达式，要使用数字值。使用Date()而不是CURRENT\_DATE返回系统日期（见5.11节中的DBMS提示）。为了运行代码11-5，改列pubdate的默认子句为DEFAULT Date()，列contract的默认子句为DEFAULT 0。

Microsoft SQL Server不支持数据类型DATE，而是使用DATETIME。使用GETDATE()而不是CURRENT\_DATE来返回系统时间，参见5.11节的DBMS提示。要运行代码11-5，将列pubdate的数据类型变为DATETIME，将它的默认子句改为DEFAULT GETDATE()。

在Oracle中，DEFAULT子句跟在数据类型之后，在包括空值约束的所有列约束之前。Oracle 9i及之后版本支持CURRENT\_DATE，在Oracle 8i及之前版本使用SYSDATE而不是CURRENT\_DATE，参见5.11节中的DBMS提示。Oracle将空字符串('')视为空值，所以本书已将title\_name默认设置变为空格字符(' ')，参见3.14节的DBMS提示。代码11-7是代码11-5的Oracle版本。

DB2不支持以算数表达式作为默认值。为了运行代码11-5，将列contract的默认子句变为DEFAULT 0。

在MySQL中，默认值必须为字面量，不能是函数或表达式。这个限制意味着不能将列date的默认值设为CURRENT\_DATE。为了运行代码11-5，删除列pubdate默认子句（或将默认表达式变为一个时间值），同时将列contract的默认子句变为DEFAULT 0。（例外，可以将一个TIMESTAMP列的默认值设为CURRENT\_TIMESTAMP）。

对于所有DBMS，查阅文档来了解DBMS如何处理相应数据类型支持自动产生唯一行标识符的默认子句，参见3.13节。

**代码11-7** 在Oracle中，默认子句必须居于所有列约束之前

```
CREATE TABLE titles
(
 title_id CHAR(3) NOT NULL,
 title_name VARCHAR(40) DEFAULT '' NOT NULL,
 type VARCHAR(10) DEFAULT 'undefined',
 pub_id CHAR(3) NOT NULL,
 pages INTEGER DEFAULT NULL ,
 price DECIMAL(5,2) DEFAULT 0.00 NOT NULL,
 sales INTEGER , ,
 pubdate DATE DEFAULT SYSDATE,
```

```
contract SMALLINT DEFAULT (3*7)-21 NOT NULL
);
```

## 11.6 使用 PRIMARY KEY 指定主键

2.2节介绍了主键，这里回顾其基本的内容。

- 主键唯一标识出表的每一行。
- 没有两行拥有相同主键。
- 主键不允许为空值。
- 每个表只能有一个主键。
- 单列键是简单键，多列键是组合键。
- 在组合键中，一列中的值可以重复，但所有组合键列的值的组合必须唯一。
- 表可以有不止一种列的组合唯一标识行，每一种组合是候选键。数据库设计者从中选择一个作为主键。

在定义主键约束时，要重点考虑以下方面。

- 简单键可以作为列约束或表约束，组合键只能作为表约束，参见11.2节。
- 在CREATE TABLE的定义中使用关键字PRIMARY KEY定义主键约束。
- 作为表约束，PRIMARY KEY要求显式指定列名。作为列约束，PRIMARY KEY作用于它定义的列。
- SQL标准允许创建没有主键的表（违反了关系模型）。实际工作中，每个表总有一个主键。
- 一个表不能有一个以上的主键约束。
- 在实际工作中，主键约束总是显式命名，使用CONSTRAINT子句来实现，参见11.2节。
- 所有主键列都是NOT NULL的。如果没有指明空值约束，DBMS对主键列隐式指定为NOT NULL，参见11.4节。
- 除非列的数据类型能自动产生唯一的行标识符，否则插入行时必须显式指明主键值，参见3.13节。要了解更多有关插入行的知识，参见10.2节。
- 主键值通常在插入之后不再改变。
- 有关被外键引用的主键插入修改和删除所要考虑的问题，参见11.7节。
- DBMS自动为主键创建唯一索引（见第12章）。

350

### ⇒ 指定简单主键

要指定简单主键作为列约束，在CREATE TABLE的列定义中加入以下的列约束。

```
[CONSTRAINT constraint_name]
 PRIMARY KEY
```

或

要指定简单主键作为表约束，向CREATE TABLE定义加入以下表约束。

```
[CONSTRAINT constraint_name]
 . PRIMARY KEY (key_column)
```

*key\_column*是主键列的名称。一个表的主键约束不能多于一个。要全面了解CREATE TABLE的语法，参见11.1节。CONSTRAINT子句是可选的，*constraint\_name*是主键约束的名字，参见11.2节。

代码11-8a、11-8b和11-8c显示了3个等价的为示例数据库表publishers定义简单主键的方法。

代码11-8a使用列约束指定主键列。这种语法是创建简单主键的最简单方法。

代码11-8b使用了无名的表约束来指定主键。列pub\_id显式添加了列约束NOT NULL，但是这并不必要，因为DBMS会自动、隐式地设置这个约束（除了DB2，参见本节后面的DBMS提示）。

代码11-8c使用了有名的表约束指明主键。这种语法显示了创建主键的更好方法，如果以后决定要改变或删除这个键可以使用名publishers\_pk，见11.12节。

#### 代码11-8a 使用列约束为示例数据库表 publishers 定义简单主键

```
CREATE TABLE publishers
(
 pub_id CHAR(3) PRIMARY KEY,
 pub_name VARCHAR(20) NOT NULL ,
 city VARCHAR(15) NOT NULL ,
 state CHAR(2) ,
 country VARCHAR(15) NOT NULL
);
```

#### 代码11-8b 使用无名表约束为示例数据库表 publishers 定义简单主键

```
CREATE TABLE publishers
(
 pub_id CHAR(3) NOT NULL,
 pub_name VARCHAR(20) NOT NULL,
 city VARCHAR(15) NOT NULL,
 state CHAR(2) ,
 country VARCHAR(15) NOT NULL,
 PRIMARY KEY (pub_id)
);
```

#### 代码11-8c 使用有名表约束为示例数据库表 publishers 定义简单主键

```
CREATE TABLE publishers
(
 pub_id CHAR(3) NOT NULL,
 pub_name VARCHAR(20) NOT NULL,
 city VARCHAR(15) NOT NULL,
 state CHAR(2) ,
 country VARCHAR(15) NOT NULL,
 CONSTRAINT publishers_pk
 PRIMARY KEY (pub_id)
);
```

#### ⇒ 指定组合主键

在CREATE TABLE定义中加入表约束。

[CONSTRAINT *constraint\_name*]  
PRIMARY KEY (*key\_columns*)

11

*key\_columns*是逗号分隔的主键列名列表。一个表的主键约束不能超过一个。要全面了解CREATE TABLE语法，参见11.1节。

CONSTRAINT子句是可选的，*constraint\_name*是主键约束名，参见11.2节。

代码11-9为示例数据库表title\_authors定义组合主键。主键列是title\_id和au\_id，键名为authors\_pk。

#### 代码11-9 使用有名表约束为示例数据库表 title\_authors 定义组合主键

```
CREATE TABLE title_authors
(
 title_id CHAR(3) NOT NULL,
 au_id CHAR(3) NOT NULL,
 au_order SMALLINT NOT NULL,
 royalty_share DECIMAL(5,2) NOT NULL,
```

```
CONSTRAINT title_authors_pk
 PRIMARY KEY (title_id, au_id)
);
```

✓ 提示

- 要了解CREATE TABLE语句运行的结果，使用10.1节介绍的命令查看表结构。
- 除主键外要定义包含唯一值的列，参见11.8节。
- 在一个表指定两个或更多主键列约束是非法的。例如，不能使用以下语句为表title\_authors指定组合主键。

```
CREATE TABLE title_authors(
 title_id CHAR(3) PRIMARY KEY,
 au_id CHAR(3) PRIMARY KEY,
 au_order SMALLINT NOT NULL,
 ...
); --Illegal
```

- **DBMS** DB2要求为主键列显式设置空值约束NOT NULL，参见11.4节。为了运行代码11-8a，为列pub\_id加上列约束NOT NULL。  
Oracle将空字符串('')视为空值，参见3.14节的DBMS提示。

352

## 11.7 使用 FOREIGN KEY 指定外键

第2章介绍了外键，这里回顾其中基本内容。

- 外键是联系两个表的一种机制。
- 外键是表的一列（或一组列），它的值联系或引用其他表的值。
- 外键确保表中的行，在另外被称作引用表或父表的表中有相对应的行。
- 外键与引用表的主键或候选主键建立直接关系，于是外键的值被限于已经存在的父键值。这个约束被称作完整性。
- 外键不像主键，允许空值。
- 表可以有零个或多个外键。
- 外键值在表中通常不是唯一的。
- 不同表的外键列可以引用父表的同一列。
- 单列键是简单键，多列键是组合键。

在定义外键约束时，要重点考虑以下方面。

- 简单键可以作为列约束或表约束，组合键只能作为表约束，参见11.2节。
- 在CREATE TABLE的定义中使用关键字FOREIGN KEY或REFERENCES来定义外键约束。
- 外键和它的父键可以有不同的列名。
- 外键的数据类型必须和父键有相同的数据类型，或必须能够隐式转换为相同的数据类型，参见5.13节。
- FOREIGN KEY列不一定非要引用另一个表的PRIMARY KEY列，它也可以引用另一个表的UNIQUE列，参见11.8节。
- 一个表可以有任意个外键约束（或根本没有）。
- 实际工作中，外键约束总是显式命名。使用CONSTRAINT子句命名一个约束，参见11.2节。

□ 外键约束简化了修改和删除，并使数据库变得易于保持一致，但即便一个中型的数据库，关系结构都会变得非常复杂。糟糕的设计会让日常查询变得很费时，规则重复、备份和恢复操作复杂，还会产生错乱地删除。

353

为了保证引用完整性，DBMS不允许创建孤立的行或使行成为孤立的（有外键约束表的行在父表中没有关联的行）。在引用父表主键列的外键列INSERT、UPDATE或DELETE行时，DBMS执行引用完整性检查。

**向包含外键的表插入行。**DBMS检查新的外键值是否匹配父表中的主键值。如果没有匹配，DBMS无法插入这个行。

**有外键的表更新行。**DBMS检查修改的外键值是否匹配父表中主键的一个值。如果没有匹配，DBMS无法更新行。

在有外键的表中删除行。引用完整性检查是没有必要的。

在父表中插入行。引用完整性检查是没有必要的。

**更新父表中的行。**DBMS检查有没有外键值与要修改的主键值匹配。如果匹配存在，DBMS就不允许更新这一行。

**从父表中删除一行。**DBMS检查有没有外键值与要删除的主键值匹配。如果匹配存在，DBMS就不允许删除这一行。

354

### ⇒ 创建简单外键

创建简单外键作为列约束，在CREATE TABLE的列定义中加入以下列约束。

```
[CONSTRAINT constraint_name]
 REFERENCES ref_table(ref_column)
```

或

创建简单外键作为表约束，在CREATE TABLE定义中加入以下表约束。

```
[CONSTRAINT constraint_name]
 FOREIGN KEY (key_column)
 REFERENCES ref_table(ref_column)
```

*key\_column*是外键列的名称。*ref\_table*是被外键约束引用的父表名称。*ref\_column*是*ref\_table*中被引用键的列名。一个表允许零个或多个外键约束。要全面了解CREATE TABLE的语法，参见11.1节。*CONSTRAINT*子句是可选的，*constraint\_name*是外键约束的名称，参见11.2节。

代码11-10使用列约束为表titles创建一个外键列。这个语法显示了创建简单外键的最简单方法。在运行语句后，DBMS将保证插入到表titles的列pub\_id中的值已经存在于表publishers的列pub\_id中。这个外键列不允许空值，所以每本书必有一个出版社。

11

355

### 代码11-10 使用列约束为示例数据库表title定义简单外键

```
CREATE TABLE titles
(
 title_id CHAR(3) NOT NULL
 PRIMARY KEY ,
 title_name VARCHAR(40) NOT NULL,
 type VARCHAR(10) ,
 pub_id CHAR(3) NOT NULL
 REFERENCES publishers(pub_id) ,
```

```

pages INTEGER ,
price DECIMAL(5,2) ,
sales INTEGER ,
pubdate DATE ,
contract SMALLINT NOT NULL
);

```

对外键列为空值的行，DBMS忽略引用完整性检查。

表**royalties**和表**titles**有一对一的联系，于是代码11-11将列**title\_id**定义为主键，同时又是引用**titles**中**title\_id**列的外键。要了解更多关于联系的知识，参见2.4节。

#### 代码11-11 使用有名的表约束为示例数据库表**royalties**定义简单外键

```

CREATE TABLE royalties
(
 title_id CHAR(3) NOT NULL,
 advance DECIMAL(9,2) ,
 royalty_rate DECIMAL(5,2) ,
 CONSTRAINT royalties_pk
 PRIMARY KEY (title_id),
 CONSTRAINT royalties_title_id_fk
 FOREIGN KEY (title_id)
 REFERENCES titles(title_id)
);

```

代码11-12使用命名表约束来创建两个外键。这个语法表明了创建外键的最好方法：如果以后决定改变或删除键可以使用这些名称（见11.12节）。每个外键列都是独立的键，不是单个组合键的一部分。然而，外键一起组成了这个表的组合主键。

#### 代码11-12 使用有名的表约束为示例数据库表**title\_author**定义简单外键

```

CREATE TABLE title_authors
(
 title_id CHAR(3) NOT NULL,
 au_id CHAR(3) NOT NULL,
 au_order SMALLINT NOT NULL,
 royalty_share DECIMAL(5,2) NOT NULL,
 CONSTRAINT title_authors_pk
 PRIMARY KEY (title_id, au_id),
 CONSTRAINT title_authors_fk1
 FOREIGN KEY (title_id)
 REFERENCES titles(title_id),
 CONSTRAINT title_authors_fk2
 FOREIGN KEY (au_id)
 REFERENCES authors(au_id)
);

```

#### ⇒ 创建组合外键

向CREATE TABLE定义中加入以下表约束。

```
[CONSTRAINT constraint_name]
FOREIGN KEY (key_columns)
REFERENCES ref_table(ref_columns)
```

*key\_columns*是逗号分隔的外键列名列表。*ref\_table*是被外键约束引用的父表名。*ref\_columns*逗号分隔、被引用的表*ref\_table*中的列名列表。*key\_columns*和*ref\_column*的列数量必须相同，按照对应的顺序排列。在一个表中可以有零个或多个外键约束。要全面了解CREATE TABLE的语法，参见11.1节。

CONSTRAINT子句是可选的，*constraint\_name*是外键约束的名称，参见11.2节。

示例数据库不包含组合外键，假定创建名为out\_of\_print的表来存储关于每位作者的绝版书信息。表title\_authors包含组合外键。这个约束说明表out\_of\_print如何引用这个键。

```
CONSTRAINT out_of_print_fk
 FOREIGN KEY
 (title_id, au_id)
 REFERENCES
 title_authors(title_id, au_id)
```

### ✓ 提示

- 要了解CREATE TABLE语句的运行结果，使用10.1节中介绍的命令检查表结构。
  - 如果被引用的列是表*ref\_table*的主键，可以省略REFERENCES子句中的(*ref\_column*)或(*ref\_columns*)表达式。
  - 外键约束可以引用同一个表的其他列（自引用），7.9节介绍过表employees是自引用的。（为了说明问题，创建表employees，它不是示例数据库的一部分）。
- 表employees有3列：emp\_id、emp\_name和boss\_id。emp\_id是唯一标识雇员的主键，boss\_id是雇员管理者的雇员ID。每一个管理者也是一个雇员，于是要确保添加到表的每一个管理者ID匹配已存在的雇员ID。设置boss\_id的外键约束emp\_id。

```
CREATE TABLE employees
(
 emp_id CHAR(3) NOT NULL,
 emp_name CHAR(20) NOT NULL,
 boss_id CHAR(3) NULL,
 CONSTRAINT employees_pk
 PRIMARY KEY (emp_id),
 CONSTRAINT employees_fk
 FOREIGN KEY (boss_id)
 REFERENCES employees(emp_id)
);
```

- 11
- 357
- 当试图更新或删除外键值所引用的键值（在父表中）时，SQL允许定义DBMS要采取的行为。要触发一个引用行为，在FOREIGN KEY约束中使用ON UPDATE或ON DELETE子句。不同DBMS对这些子句的支持不同，查阅DBMS文档了解foreign key或referential integrity内容。以下两条提示说明了SQL标准对这类子句的定义。
  - ON UPDATE *action*子句指明，当试图UPDATE一个行中被其他表的外键所引用的键值（在父表）时，DBMS将如何处理。行为采用下述4个值中的一个：
    - CASCADE，更新依赖的外键值为新的父表值；
    - SET NULL，将依赖的外键值改为空值；
    - SET DEFAULT，将依赖的外键值改为默认值，参见11.5节；
    - NO ACTION，当违反外键约束时产生一个错误提示，这是默认行为。
  - ON DELETE *action*子句指明，当试图DELETE一个行中被其他表的外键所引用的键值（在父表）

时, DBMS将如何处理。行为采用下述4个值中的一个:

- CASCADE, 删除所包含的外键值与要删除的主键值匹配的行;
- SET NULL, 将依赖的外键值改为空值;
- SET DEFAULT, 将依赖的外键值改为默认值, 参见11.5节;
- NO ACTION, 当违反外键约束时产生一个错误信息, 这是默认行为。

□ **DBMS** Microsoft SQL Server不支持数据类型DATE。为了运行代码11-10, 将列pubdate的数据类型改为DATETIME。

Oracle将空字符串('')视为空值, 参见3.14节的DBMS提示。

MySQL通过InnoDB表加强了外键约束。查阅MySQL有关foreign key的文档。InnoDB FOREIGN KEY语法比标准的CREATE TABLE语法更严格。

358

## 11.8 使用 UNIQUE 确保值唯一

唯一约束确保列(列集)没有重复的值。除了唯一列可以包含空值和表可以包含多个唯一列, 唯一约束和主键约束没有别的区别。(要了解更多有关主键约束的知识, 参见11.6节。)

假定要向表titles加入列isbn来存储图书的ISBN。ISBN是区分每种图书的唯一、标准化的识别号。表titles已经有了一个主键(title\_id), 为了保证ISBN值唯一, 可以为列isbn创建唯一约束。

在创建唯一约束时, 要重点考虑以下方面。

- 单列键是简单约束, 多列键是组合约束。
- 组合约束的值在一列内可以重复, 但这些列值的每一个组合必须是唯一的。
- 简单唯一约束可以作为列约束或表约束, 组合唯一约束只能作为表约束, 参见11.2节。
- 在CREATE TABLE定义中使用关键字UNIQUE创建唯一约束。
- 作为表约束, UNIQUE需要指明列名; 作为列约束, UNIQUE用于它被定义的列。
- 表可以没有或有多个唯一约束。
- 在实际工作中, 唯一约束总是显式命名的。可使用CONSTRAINT子句命名约束, 参见11.2节。
- 唯一约束列可以禁止空值, 参见11.4节。

### ⇒ 创建简单的唯一约束

创建简单唯一约束作为列约束, 在CREATE TABLE列定义中加入以下列约束。

```
[CONSTRAINT constraint_name]
 UNIQUE
```

或

要创建简单唯一约束作为表约束, 在CREATE TABLE的定义中加入以下表约束。

```
[CONSTRAINT constraint_name]
 UNIQUE (unique_column)
```

*unique\_column*是要限制重复值的列名。一个表可以有零个或多个唯一约束。要全面了解CREATE TABLE的语法, 参见11.1节。

CONSTRAINT子句是可选的, *constraint\_name*是唯一约束的名称, 参见11.2节。

代码11-13a和代码11-13b给出了为示例数据库表titles定义简单唯一约束的两个等价的方法。

359

代码11-13a使用列约束来指定唯一列。这种语法是创建简单唯一约束的最简单方法。

代码11-13b使用了有名表约束来创建唯一列。如果将来要修改或删除列约束，这个语法是创建唯一约束的更好方法，参见11.12节。

### 代码11-13a 使用列约束为示例数据库表titles在title\_name列创建简单唯一约束

```
CREATE TABLE titles
(
 title_id CHAR(3) PRIMARY KEY ,
 title_name VARCHAR(40) NOT NULL UNIQUE,
 type VARCHAR(10) ,
 pub_id CHAR(3) NOT NULL ,
 pages INTEGER ,
 price DECIMAL(5,2) ,
 sales INTEGER ,
 pubdate DATE ,
 contract SMALLINT NOT NULL
);
```

### 代码11-13b 使用有名表约束来为示例数据库表titles在列title\_name上创建简单唯一约束

```
CREATE TABLE titles
(
 title_id CHAR(3) NOT NULL,
 title_name VARCHAR(40) NOT NULL,
 type VARCHAR(10) ,
 pub_id CHAR(3) NOT NULL,
 pages INTEGER ,
 price DECIMAL(5,2) ,
 sales INTEGER ,
 pubdate DATE ,
 contract SMALLINT NOT NULL,
 CONSTRAINT titles_pk
 PRIMARY KEY (title_id),
 CONSTRAINT titles_unique1
 UNIQUE (title_name)
);
```

#### ⇒ 创建组合唯一约束

在CREATE TABLE的定义中加入以下表约束。

[CONSTRAINT *constraint\_name*]  
UNIQUE (*unique\_columns*)

*unique\_columns*是逗号分隔禁止重复值的列名列表。在一个表中可以有零个或多个唯一约束。要全面了解CREATE TABLE语法，参见11.1节。CONSTRAINT子句是可选的，*constraint\_name*是唯一约束的名称，参见11.2节。

代码11-14为示例数据库表authors定义了多列唯一约束。这个约束要求每一位作者姓与名的结合是唯一的。

### 代码11-14 使用有名表约束为示例数据库表authors在au\_fname和au\_lname列创建一个组合唯一约束

```
CREATE TABLE authors
(
 au_id CHAR(3) NOT NULL,
```

```

au_fname VARCHAR(15) NOT NULL,
au_lname VARCHAR(15) NOT NULL,
phone VARCHAR(12) ,
address VARCHAR(20) ,
city VARCHAR(15) ,
state CHAR(2) ,
zip CHAR(5) ,
CONSTRAINT authors_pk
 PRIMARY KEY (au_id),
CONSTRAINT authors_unique1
 UNIQUE (au_fname, au_lname)
);

```

#### ✓ 提示

- 要了解CREATE TABLE语句的运行结果，使用10.1节介绍的命令来查看表结果。
- 外键列可以引用唯一列，参见11.7节。
- 可以为候选主键指定非空唯一约束，参见2.2节。
- 可以创建唯一索引而不是唯一约束，参见12.1节。要了解所用的DBMS更倾向索引还是约束，查看DBMS有关唯一、索引和约束的文档。
- **DBMS** Microsoft SQL Server不支持数据类型DATE。为了运行代码11-13a和代码11-13b，将列pubdate的数据类型变为DATETIME。

362

Oracle将空串('')视为空值，参见3.14节的DBMS提示。

DB2要求对主键列显式设置空值约束为NOT NULL。为了运行代码11-13a，在title\_id的列约束中加入NOT NULL。对于允许有空值的唯一列，SQL标准允许有任何数量的空值。

Microsoft SQL Server在这样的列只允许一个空值，DB2一个也不允许。

## 11.9 使用 CHECK 创建检查约束

到这里，对插入值的约束仅是它所在列的数据类型、长度、范围。可以加入检查约束来进一步限制列（列集）可以接受的值。检查约束通常适用于以下方面的检查。

**最大值或最小值。**例如，阻止小于零的销售数量。

**具体值。**例如，在列science只允许'biology'、'chemistry'或'physics'。

**一定范围的值。**例如，确保作者的稿酬比率在2%和20%之间。

检查约束类似外键约束那样限制插入列的值（见11.7节）。它们不同之处在于如何判断所允许的值。外键约束从另一个表得到有效值的列表，而检查约束通过逻辑（布尔）表达式来判断有效值。例如，以下检查约束，确保没有雇员的工资超过50 000美元：

```
CHECK (salary <= 50000)
```

创建检查约束时，要重点考虑以下方面。

- 作用于单列的检查约束可以是列约束或表约束，作用于多列的检查约束只能是表约束，参见11.2节。
- 在CREATE TABLE中使用关键字CHECK创建检查约束。
- 列可以有零个或多个与其相关的检查约束。
- 如果为一个列创建多个检查约束，要防止它们相互冲突。不要认为DBMS会对约束进行排序或

对约束分别验证。

- 在实际工作中，检查约束总是显式命名。使用CONSTRAINT命名约束，参见11.2节。
- 几乎所有有效的WHERE条件都可以作检查约束的条件，例如比较(=、<、<=、>、>=)、LIKE、BETWEEN、IN或IS NULL条件。（大多数DBMS在检查约束中不允许使用子查询。）可使用AND、OR和NOT连接多个条件。要了解更多关于条件的知识，参见4.5节及其后各节内容。
- 检查约束的条件可以参考表中的任何列，但不能参考其他表中的列。
- 尽管在表存入数据之后可以创建检查约束，但最好还是在表存入数据之前创建约束，这样可以尽早发现输入错误。363

### ⇒ 创建检查约束

要创建作为列约束或表约束的检查约束，应在CREATE TABLE定义中添加以下约束。

```
[CONSTRAINT constraint_name]
 CHECK (condition)
```

*condition*是当每次INSERT、UPDATE或DELETE语句修改表的内容时，DBMS需要判断的逻辑（布尔）条件。如果在修改之后*condition*条件为真或未知（因为空值），那么DBMS允许修改。如果*condition*条件为假，DBMS取消修改并返回错误。要全面了解CREATE TABLE的语法，参见11.1节。CONSTRAINT子句是可选的，*constraint\_name*是检查约束的名称，参见11.2节。

代码11-15显示了示例数据库表titles的各种列约束和表约束。约束title\_id\_chk确保每个主键值的形式为'Tnn'，其中的nn代表00和99之间的整数，也包括00和99。364

### 代码11-15 为示例数据库表titles创建一些约束

```
CREATE TABLE titles
(
 title_id CHAR(3) NOT NULL,
 title_name VARCHAR(40) NOT NULL,
 type VARCHAR(10)
 CONSTRAINT type_chk
 CHECK (type IN ('biography',
 'children', 'computer',
 'history', 'psychology')) ,
 pub_id CHAR(3) NOT NULL,
 pages INTEGER
 CHECK (pages > 0) ,
 price DECIMAL(5,2) ,
 sales INTEGER ,
 pubdate DATE ,
 contract SMALLINT NOT NULL,
 CONSTRAINT titles_pk
 PRIMARY KEY (title_id),
 CONSTRAINT titles_pub_id_fk
 FOREIGN KEY (pub_id)
 REFERENCES publishers(pub_id),
 CONSTRAINT title_id_chk
 CHECK (
 SUBSTRING(title_id FROM 1 FOR 1) = 'T')
 AND
 (CAST(SUBSTRING(title_id FROM 2 FOR 2)
 AS INTEGER) BETWEEN 0 AND 99),
```

```

CONSTRAINT price_chk
 CHECK (price >= 0.00
 AND price < 100.00),
CONSTRAINT sales_chk
 CHECK (sales >= 0),
CONSTRAINT pubdate_chk
 CHECK (pubdate >= DATE '1950-01-01'),
CONSTRAINT title_name_chk
 CHECK (title_name <> ''
 AND contract >= 0),
CONSTRAINT revenue_chk
 CHECK (price * sales >= 0.00)
);

```

#### ✓ 提示

- 要了解CREATE TABLE语句的运行结果，使用10.1节介绍的命令查看表结构。
- SQL标准规定检查条件不能引用日期和时间，以及用户函数(CURRENT\_TIMESTAMP、CURRENT\_USER等)，但有一些DBMS，比如Microsoft Access、Microsoft SQL Server和PostgreSQL，是允许这么做的，例如：

`CHECK(ship_time >= CURRENT_TIMESTAMP)`

这些函数在5.11节和5.12节中有说明。

- **DBMS** 要在Microsoft Access中运行代码11-15，需将两个列约束（在type列和pages列上）移到最后的列定义之后，把它们变为表约束。将第一个子串表达式变为`Mid(title_id, 1, 1)`；将CAST表达式变为`CInt(Mid(title_id, 2, 2))`；从日期值除去关键字DATE同时用字符#包围而不要用引号（即#1950-01-01#）。

要在Microsoft SQL Server中运行代码11-15，需将列pubdate的数据类型变为DATETIME；将两个子串表达式变为`SUBSTRING(title_id, 1, 1)`和`SUBSTRING(title_id, 2, 2)`；同时日期值除去关键字DATE变为('1950-01-01')。

要在Oracle中运行代码11-15，需将两个子串表达式变为`SUBSTR(title_id, 1, 1)`和`SUBSTR(title_id, 2, 2)`。

为了在DB2中运行代码11-15，要将两个子串表达式变为`SUBSTR(title_id, 1, 1)`和`SUBSTR(title_id, 2, 2)`，同时日期值去掉关键字DATE变为('1950-01-01')。

MySQL不支持命名的检查列约束，也不强制进行约束检查。要运行代码11-15，必须除去type\_chk约束。同时，将CAST的数据类型从INTEGER变为SIGNED。

为了在较早的PostgreSQL版本上运行代码11-15，将浮点数0.00和100.00变为`CAST(0.00 AS DECIMAL)`和`CAST(100.00 AS DECIMAL)`，参见5.13节。

在Microsoft SQL Server中，将检查约束title\_id\_chk变为`CHECK (title_id LIKE '[T][0-9][0-9] ')`；查阅SQL Server帮助文档的pattern或wildcard部分。

Oracle将空串('')视为空值，参见3.14节的DBMS提示。

365

## 11.10 使用CREATE TEMPORARY TABLE创建临时表

到目前为止，本书创建的表都是永久表，称作基础表(base table)，它们持久保存数据直到显式删除(DROP)表为止。SQL也允许创建临时表来存储中间结果。临时表通常用于如下操作。

- 将复杂耗时的查询结果存储起来，在随后的查询中重复使用这个结果，可以极大地改变运行效率。
- 在特定时刻及时创建表的图像或快照。（可以加入默认值为CURRENT\_TIMESTAMP的列来记录这个时刻。）
- 保存子查询的结果。
- 保存费时或复杂计算的中间结果。

临时表（temporary table）是在会话或事务结束时DBMS能自动清空的表。（表中数据将随表被删除。）会话（session）是登录与注销之间连接DBMS的时间，在会话期间，DBMS接受和执行命令。

在创建临时表时，要重点考虑以下方面。

- 有关表名、列名、数据类型等方面，临时表和基本表的规则相同。
- 使用标准的CREATE TABLE加上一点额外的语法就可以创建临时表。在关键字TABLE之前加上关键字GLOBAL TEMPORARY或LOCAL TEMPORARY。
- 临时表最初没有行。可以像在基本表那样（见第10章）插入、更新和删除行。
- 如果创建了巨大的临时表，可以自己删除而不是等DBMS来释放内存，参见11.13节。
- CREATE TEMPORARY TABLE使得数据库管理员可以给予用户工作存储空间而无需给他们CREATE TABLE、ALTER TABLE或DROP TABLE的授权，后者会带来潜在的灾难性后果。

366

### » 创建临时表

输入：

```
CREATE {LOCAL | GLOBAL} TEMPORARY TABLE table
(
 column1 data_type1 [constraints1],
 column2.data_type2 [constraints2],
 ...
 columnN data_typeN [constraintsN]
 [, table_constraints]
);
```

*table*是要创建的临时表的名字。LOCAL表明*table*是局部临时表。GLOBAL表明*table*是全局临时表（代码11-6和代码11-7）。*column1*, *column2*, ..., *columnN*是表*table*中的列名。

*data\_type1*, *data\_type2*, ..., *data\_typeN*表明相应列的SQL数据类型。

不同的DBMS对临时表的列约束和表约束的支持是不同的。查阅DBMS文档的temporary table部分。要全面了解约束，参见11.2节。

11

367

**代码11-16** 局部临时表仅用户自己可用。当DBMS进程结束时就会消失

```
CREATE LOCAL TEMPORARY TABLE editors
(
 ed_id CHAR(3) ,
 ed_fname VARCHAR(15),
 ed_lname VARCHAR(15),
 phone VARCHAR(12),
 pub_id CHAR(3)
);
```

**代码11-17** 全局临时表可以被用户自己和其他用户访问。当DBMS会话和其他引用它的任务结束时，它就会消失

```
CREATE GLOBAL TEMPORARY TABLE editors
(
 ed_id CHAR(3) ,
 ed_fname VARCHAR(15),
 ed_lname VARCHAR(15),
 phone VARCHAR(12),
 pub_id CHAR(3)
);
```

### ✓ 提示

- 要了解CREATE TEMPORARY TABLE语句的执行结果，使用10.1节介绍的命令来查看表结构。
- 要修改临时表，参见11.12节。
- 要创建已存在表的临时备份，参见11.11节。
- **DBMS** Microsoft Access不支持临时表。

Microsoft SQL Server创建临时表时，若要创建局部表，使用语法：

`CREATE TABLE #table (...);`

若创建全局表则使用语法：

`CREATE TABLE ##table (...);`

当使用表名引用临时表时必须包含#字符。

Oracle创建临时表的语法是：

`CREATE GLOBAL TEMPORARY TABLE table (...);`

DB2创建临时表的语法是：

`DECLARE GLOBAL TEMPORARY TABLE table (...);`

MySQL不区分局部临时表和全局临时表，去掉关键字LOCAL或GLOBAL。

PostgreSQL支持（但忽略）GLOBAL和LOCAL关键字，只创建一种临时表。

你使用的DBMS或许支持可选的、SQL标准定义的ON COMMIT子句。ON COMMIT PRESERVE ROWS保留COMMIT事务中对临时表的所有数据进行的修改，而ON COMMIT DELETE ROWS则在COMMIT事务结束之后清空表。要了解更多有关COMMIT的知识，参见第14章。

对于所有DBMS，查阅文档了解DBMS如何处理与基本表名字相同的临时表。在某些情况下，临时表会在被删除之前，隐藏或封闭同名的基本表。

SQL标准有关临时表行为的定义普遍被忽略。各种DBMS在实现临时表的持久性、可见性、约束、外键（引用完整性）、索引和视图等方面各不相同，请查阅有关DBMS文档的temporary table部分。

368

## 11.11 使用CREATE TABLE AS利用已存在表创建新表

CREATE TABLE AS语句创建新表且向其中填充SELECT的结果。它类似于使用CREATE TABLE创建空表，然后使用INSERT SELECT填充表（见10.2节）。CREATE TABLE AS通常被用于如下操作。

- 对某些行存档。
- 为表做备份。
- 在特定时刻为表创建快照。
- 快速复制表结构但不包含数据。
- 创建测试数据。
- 在修改实际数据前，复制表来测试INSERT、UPDATE和DELETE操作。

当使用CREATE TABLE AS时，要重点考虑以下方面。

- 通过使用标准的SELECT子句WHERE、JOIN、GROUP BY和HAVING，或第4章到第9章介绍的任何SELECT选项为新表选择行。
- 不管SELECT引用的数据表有多少，CREATE TABLE AS只向一个表中插入数据。

- 列的属性和SELECT子句列表中的表达式决定新表的结构。
- 如在SELECT子句的列表里包含导出（计算得到）的列，新表中对应列的值为CREATE TABLE AS 执行时计算得到的结果，参见5.1节。
- 新表不能和已存在的表重名。
- 必须从数据库管理员获得CREATE TABLE的授权。

369

### ⇒ 利用已存在表创建新表

输入：

```
CREATE TABLE new_table
AS subquery;
```

*new\_table*是要创建的表的名称。*subquery*是一个返回插入到*new\_table*表行的SELECT语句。DBMS 使用*subquery*的结果决定*new\_table*的结构和列的顺序、名称、数据类型和列值。

代码11-18将已存在表authors的结构和数据复制到名称为authors2的新表。

代码11-19使用总是为假的条件WHERE来仅将已存在的表publishers的结构（不包括数据）复制到名称为publishers2的新表。

代码11-20创建名称为titles2、包含出版社P01所出版书的名称与销量的全局临时表，参见11.10节。

代码11-21使用连接创建名称为author\_title\_names、包含非纽约或非加利福尼亚州的作者和他们图书的新表。

370

#### 代码11-18 将已存在表authors的结构和数据复制到名称为authors2的新表

```
CREATE TABLE authors2 AS
SELECT *
FROM authors;
```

#### 代码11-19 将已存在表publishers的结构（不包括数据）复制到名称为publishers2的新表

```
CREATE TABLE publishers2 AS
SELECT *
FROM publishers
WHERE 1 = 2;
```

11

#### 代码11-20 创建名称为titles2、包含出版社P01所出版书的名称和销量的全局临时表

```
CREATE GLOBAL TEMPORARY TABLE titles2 AS
SELECT title_name, sales
FROM titles
WHERE pub_id = 'P01';
```

#### 代码11-21 创建名称为author\_title\_names、包含非纽约或加利福尼亚州的作者和他们图书的新表

```
CREATE TABLE author_title_names AS
SELECT a.au_fname, a.au_lname,
t.title_name
FROM authors a
INNER JOIN title_authors ta
ON a.au_id = ta.au_id
INNER JOIN titles t
```

```
ON ta.title_id = t.title_id
WHERE a.state NOT IN ('CA', 'NY');
```

✓ 提示

- 要检查新表结构，使用10.1节中介绍的命令。
- 创建临时表经常用来保存当前数据。

```
CREATE GLOBAL TEMPORARY TABLE
 sales_today AS
SELECT *
 FROM orders
 WHERE order_date = CURRENT_DATE;
```

- **DBMS** SQL:2003引入了CREATE TABLE AS，但Microsoft Access和Microsoft SQL Server使用SELECT INTO从已存在的表创建新表：

```
SELECT columns
 INTO new_table
 FROM existing_table
 [WHERE search_condition];
```

SQL标准中SELECT INTO与此不同，它选择一个值赋给宿主程序的标量变量而不是创建新表。Oracle、DB2和MySQL按照标准方法使用SELECT INTO。为了可移植性，不要使用CREATE TABLE AS或SELECT INTO。应该使用CREATE TABLE创建新的空表，然后使用INSERT SELECT添加值。

为了在Microsoft Access中运行代码11-18至代码11-21，对于代码11-18输入：

```
371
SELECT *
 INTO authors2
 FROM authors;
```

对于代码11-19：

```
SELECT *
 INTO publishers2
 FROM publishers
 WHERE 1=2;
```

对于代码11-20：

```
SELECT title_name, sales
 INTO titles2
 FROM titles
 WHERE pub_id = 'P01';
```

对于代码11-21：

```
SELECT a.au_fname, a.au_lname,
 t.title_name
 INTO author_title_names
 FROM titles t
 INNER JOIN authors a
 INNER JOIN title_authors ta
 ON a.au_id = ta.au_id)
 ON t.title_id = ta.title_id
 WHERE a.state NOT IN ('NY', 'CA');
```

为了在Microsoft SQL Server中运行代码11-18至代码11-21，对于代码11-18输入：

```
SELECT *
 INTO authors2
 FROM authors;
```

对于代码11-19:

```
SELECT *
 INTO publishers2
 FROM publishers
 WHERE 1=2;
```

对于代码11-20:

```
SELECT title_name, sales
 INTO ##titles2
 FROM titles
 WHERE pub_id = 'P01';
```

对于代码11-21:

```
SELECT a.au_fname, a.au_lname,
 t.title_name
 INTO author_title_names
 FROM authors a
 INNER JOIN title_authors ta
 ON a.au_id = ta.au_id
 INNER JOIN titles t
 ON ta.title_id = t.title_id
 WHERE a.state NOT IN ('CA', 'NY');
```

在Oracle 8i中，代码11-21要使用WHERE语法而不是JOIN语法:

```
CREATE TABLE author_title_names AS
SELECT a.au_fname, a.au_lname,
 t.title_name
 FROM authors a, title_authors ta,
 titles t
 WHERE a.au_id = ta.au_id
 AND ta.title_id = t.title_id
 AND a.state NOT IN ('CA', 'NY');
```

DB2的CREATE TABLE AS语法是:

```
CREATE TABLE new_table AS
 (subquery) options;
```

DB2文档对可用的选项有说明。例如，为了运行代码11-19，输入:

```
CREATE TABLE publishers2 AS
 (SELECT * FROM publishers)
 WITH NO DATA;
```

DB2也支持CREATE TABLE new\_table LIKE existing\_table将一个表作为模板创建另一个表。

为了在MySQL中运行代码11-20，删除关键字GLOBAL。

PostgreSQL也允许使用SELECT INTO从查询结果来定义新表，但推荐使用CREATE TABLE AS。CREATE TABLE AS与一些供应商所谓的物化表或物化视图类似，只是标准语句无法创建新表与旧表之间的联系。

## 11.12 使用ALTER TABLE修改表

使用ALTER TABLE语句来增加、修改、删除列和约束可以修改表。

**DBMS** 不管SQL标准如何，不同的DBMS的ALTER TABLE实现方式有很大差异。要了解可以修改什么及在什么条件下可以修改，查看DBMS文档的ALTER TABLE说明。依据不同的DBMS，使用ALTER TABLE可以：

- 增加或删除列；
- 修改列的数据类型；
- 增加、修改或删除列的默认值或空值约束；
- 增加、修改或删除列约束或表约束，如主键、外键、唯一和检查约束；
- 重命名列；
- 重命名表。

373

### ⇒ 修改表

输入：

```
ALTER TABLE table alter_table_action;
```

*table*是要修改的表名。*alter\_table\_action*是以关键字ADD、ALTER或DROP开始表明要采取行为的子句。如：

```
ADD COLUMN column type [constraints]
ALTER COLUMN column SET DEFAULT expr
DROP COLUMN column [RESTRICT|CASCADE]
ADD table_constraint
DROP CONSTRAINT constraint_name
```

代码11-22或代码11-23从表authors增加和删除列email\_address。

如果DBMS的ALTER TABLE语句不支持所需要的行为（如删除或重命名列或约束），了解DBMS是否提供了其他SQL语句，或通过命令行的单独（非SQL）命令，或图形用户界面实现这种行为。如果都不行，就手工重建和填充表到所要求的状态。

#### 代码11-22 向表authors添加列email\_address

```
ALTER TABLE authors
 ADD email_address CHAR(25);
```

#### 代码11-23 从表authors删除列email\_address

```
ALTER TABLE authors
 DROP COLUMN email_address;
```

### ⇒ 再创建和再填充表

- (1) 使用CREATE TABLE创建有新的列定义、列约束和表约束的新表，参见11.3节及其后各节内容。
- (2) 使用INSERT SELECT从旧表复制行（恰当的列）到新表，参见10.2节。
- (3) 使用SELECT \* FROM new\_table来确认新表包含了恰当的行，参见4.1节。
- (4) 使用DROP TABLE删除旧表，参见11.13节。
- (5) 将新表重命名为旧表名，参见本节的DBMS提示。

374

(6) 如有必要就重建索引，参见12.1节。

也需要重建所有其他已从旧表中删除的属性，比如授权和触发器。

#### ✓ 提示

- 要了解ALTER TABLE语句的运行结果，使用10.1节中介绍的命令检查表结构。
- 不能删除表仅有的一列。
- 要修改或删除约束，使用CONSTRAINT子句创建约束时指定的约束名，参见11.2节。如果对约束没有命名，使用DBMS自动产生的约束名。
- 通常对空表的强制约束比有数据的表的约束要少。例如，当向已有一行或多行数据的表添加新列时，这个列不能有NOT NULL约束，空表可以添加非空的列。
- **DBMS** DB2不允许使用ALTER TABLE删除列，因此代码11-23无法运行。

表11-2列出了当前数据库中重命名表的命令和查询。

375

表11-2 重命名表

| DBMS       | 命令或查询                                    |
|------------|------------------------------------------|
| Access     | 右击数据库窗口中的表，然后单击Rename（重命名）               |
| SQL Server | EXEC sp_rename 'old_name','new_name'     |
| Oracle     | RENAME old_name TO new_name;             |
| DB2        | RENAME TABLE old_name TO new_name;       |
| MySQL      | RENAME TABLE old_name TO new_name;       |
| PostgreSQL | ALTER TABLE old_name RENAME TO new_name; |

## 11.13 使用 DROP TABLE 删除表

使用DROP TABLE语句从数据库删除表。删除表，要重点考虑以下方面。

- 可以删除基本表或临时表。
- 一些DBMS允许通过回滚事务恢复删除了的表（见第14章）。如果删除的表不属于事务，可以从最近的备份恢复表。（尽管可能已经过时。）
- 删除表就意味着删除了表的结构、数据、索引、约束、授权等。
- 删除表和删除表中所有的行不同。可以使用DELETE FROM *table*清空表，但并不删除表，参见10.4节。
- 删除表并未删除引用这个表的视图，参见第13章。
- 除非也将引用表修改或删除，否则会出现外键或视图引用到的表被删除的问题。

11

#### ⇒ 删除表

输入：

DROP TABLE *table*;

*table*是要删除的表的名称。（代码11-24）

#### 代码11-24 删除表royalties

```
DROP TABLE royalties;
```

✓ 提示

- **DBMS** 一些DBMS要求在删除表前，删除或改变某些属性。如Microsoft SQL Server，无法使用DROP TABLE删除一个被外键约束引用的表，直到引用外键约束或引用表被首先删除。  
标准的SQL允许指明RESTRICT（限制）或CASCADE（级联）删除行为。RESTRICT（安全的）防止删除被视图或别的约束引用的表。CASCADE（不安全的）引发引用对象随着表一同被删除。  
要了解DBMS对这个功能或相似功能的支持，查看DBMS文档的*DROP TABLE*部分。

**在** 2.1节中介绍过存储在表里的行，如关系模型所要求的那样，是没有顺序的。这种无顺序使得DBMS很容易快速地插入、更新和删除行，但不幸的是这使得查询和排序的效率降低。假设运行下面的查询。

```
SELECT *
 FROM authors
 WHERE au_lname = 'Hull';
```

为了执行这个查询，DBMS必须一行接一行地查询整个表authors，比较每行au\_lname列的值与串Hull。在一个小数据库里搜索整个表很简单，但实际使用的表可能有几百万行。

DBMS提供了一种称作索引的机制，它的目的与图书和图书馆索引相同：加快数据检索。简而言之，索引是排序的列表，在这个列表中索引列（或列集）的每个不同值和包含该值的行的硬盘地址（物理地址）存储在一起。DBMS无需检索整个表来定位行，而仅需扫描索引中的地址就可直接访问相应的行。索引搜索通常要比顺序搜索快许多，但这样也有代价，本章将对此说明。

377

## 12.1 使用 CREATE INDEX 创建索引

索引是复杂的，它们的设计和其对于效率的影响取决于特定数据库优化器的特性。本章尽管提供了指导，但读者还是要查询具体DBMS文档的索引部分来了解DBMS是如何实现和使用索引的。大体上，对于以下列创建索引是恰当的，这些列经常被：

- 查询 (WHERE)。
- 排序 (ORDER BY)。
- 分组 (GROUP BY)。
- 联结 (JOIN)。
- 用来计算顺序统计 (比如, MIN()、MAX()或中值)。

大体上，对于以下列创建索引是不恰当的。

- 仅接受很少的不同值 (如性别、婚姻状态或状况)。
- 很少被用于查询。
- 只有几行的小表的一部分。

当创建索引时，要重点考虑以下方面。

- SQL的索引语句改变数据库对象，所以需要数据库管理员的授权。
- 索引不会改变数据，仅仅是快速访问数据的途径。

12

- 表可以没有或有多个索引。
- 理想的情况是在创建表时就创建表的索引。在实际工作中，索引管理是互动的过程。通常只有非常关键的索引才和表一同创建。别的索引要在执行效率问题产生、消失及用户访问模式改变时，才添加、删除。DBMS提供测试和对比工具来确定索引的效果。
- 不要超出需求创建索引。DBMS在表插入、更新或删除行之后必须更新索引（见第10章）。随着表索引的增长，DBMS要花费越来越多的时间维护索引，行的修改速度会越来越慢，因此不要在一个表上创建超过10个索引。
- 在索引被创建后，DBMS自动维护和使用索引。不需要用户或SQL程序员做额外事情在相关的索引里反映数据变化。
- 索引对于用户和SQL程序员是透明的，索引的有无并不会给编写SQL语句带来变化。
- 索引可以引用表中的一列或多列。引用一列的索引称作简单索引；引用多列的索引称作组合索引。组合索引中的列在表中不一定是相邻的。简单索引无法跨多个表。
- 列的顺序在组合索引中是重要的。组合索引只作用于定义它的那组列，并非分别作用于每个列或相同列的其他顺序。
- 可以使用相同的列、不同的结合方式创建多个组合索引。例如，以下两条语句为同一个表创建了有效的索引组合：

```
CREATE INDEX au_name_idx1
 ON authors (au_fname, au_lname);
CREATE INDEX au_name_idx2
 ON authors (au_lname, au_fname);
```

- 为了高效地排序和检索，索引要确保唯一性。唯一索引强制表索引中列（或多列）的值唯一。如果要在有重复值的列上创建唯一索引，DBMS将产生错误并拒绝创建。当创建主键约束和唯一约束时，DBMS自动创建唯一索引。
- DBMS或许自动创建外键索引，或许没有。如果没有，就要自己创建，因为大多数联结包含外键。
- 尽管索引不是关系模型的组成部分，但所有的DBMS都支持索引（且没有违反模型的任何规则）。尽管最简单的CREATEINDEX语句的语法在这本书所涉及的DBMS中是相同的，但索引不是SQL标准的一部分，所以与索引有联系的SQL语句因DBMS不同而有差异。

#### ⇒ 要创建索引

输入：

```
CREATE [UNIQUE] INDEX index
 ON table (index_columns);
```

*index*是要创建的索引的名称，且是有效的SQL标识符。在一个表中索引名称必须唯一。在Oracle、DB2和PostgreSQL中，一个数据库中的索引名称必须唯一。

*table*是要创建索引的表，*index\_columns*是要创建索引的一个或多个用逗号分隔的列名。

指明UNIQUE创建唯一索引。UNIQUE确保DBMS检查*index\_columns*中的重复值。如果*table*在*index\_columns*中已经包含了重复的行，DBMS就不能创建这个索引。如果试图在唯一索引*index\_columns*中插入或更新重复的值，DBMS会产生错误并取消操作。

代码12-1在表titles的列pub\_id上创建名为pub\_id\_idx的简单索引。pub\_id是外键，也是创建索

引的很好候选列。因为：

- 主键约束的变化将被相关表的外键约束检查。
- 当表中的数据通过匹配表中的外键列和另一个表的主键或唯一约束列结合查询时，外键列经常被用做联结标准。380

#### 代码12-1 在表titles的列pub\_id上创建简单索引

```
CREATE INDEX pub_id_idx
ON titles (pub_id);
```

代码12-2在表titles的列title\_name上创建一个名为title\_name\_idx的简单唯一索引。只有在title\_name列没有重复的值时，DBMS才创建这个索引。这个索引禁止相同书名的插入和更新。

#### 代码12-2 在表titles列title\_name上创建简单唯一索引

```
CREATE UNIQUE INDEX title_name_idx
ON titles (title_name);
```

代码12-3在表authors的列state和列city上创建名为state\_city\_idx的组合索引。当按照state加city对行排序时，DBMS使用这个索引。当按照state、city或city加state排序和检索时，这个索引是没用的。若要实现这些目的，必须分别创建索引。

#### 代码12-3 在表authors的列state和列city上创建组合索引

```
CREATE INDEX state_city_idx
ON authors (state, city);
```

##### ✓ 提示

- 不要将*index*和*key*这两个术语互换使用（尽管某些书中这样使用了）。索引是DBMS改善执行效率的物理机制（硬件相关）。键是DBMS用来强制引用完整性和通过视图更新的逻辑（基于数据）概念。
- 可以使用唯一约束来防止列中出现重复的值，参见11.8节。
- 索引是存储在硬盘上的文件，因此占用空间（可能是很大的空间）。如果使用得当，索引可以成为避开连续读取大表、减少硬盘存取的主要手段。当创建索引时，它使用的空间是相关表占用空间的1.5倍（要确保有足够的空间）。当索引完成时，大多数空间将被释放。381
- 连续地搜索表（因为没有索引）被称作表扫描。
- 聚集索引是键值逻辑顺序决定表中相应行的物理顺序的索引。而在非聚集索引中，索引的逻辑顺序和行存储在硬盘上的物理顺序是不同的。表只可以有一个聚集索引。聚集索引通常能改进运行效率，但有些情况下，它们会让搜索变得很快，而让插入、删除和修改变得很慢。
- 大多数索引是采用平衡树，或B树实现的。B树是一种高级数据结构，可以使硬盘的读写尽可能地少。有些DBMS允许在创建索引时指明数据结构。
- DBMS** Microsoft SQL Server和DB2在唯一约束被指明时，将多个空值视为重复，在唯一索引列中不允许有一个以上的空值。Microsoft Access、Oracle、MySQL和PostgreSQL允许在这样的列中有多个空值。382
- 有些DBMS允许在视图中（见第13章）像在表中一样创建索引。

## 12.2 使用DROP INDEX删除索引

使用DROP INDEX语句删除索引。因为索引逻辑上独立于相关表中的数据，在任何时候删除索引都不会影响表（或其他索引）。如果删除了索引，所有SQL程序和应用会继续正常运行，但访问先前有索引的数据会变慢。

删除索引通常是因为：

- 相关表很小（或已被删除）或用户不再访问索引列，因此不再需要索引；
- 在插入、更新和删除操作后DBMS维护索引所需的时间超过了索引加速检索而节约的时间。

SQL标准未包含索引，因此不同DBMS的SQL索引语句有所差异。本节介绍本书涉及的DBMS如何删除索引。对于不同的DBMS，请查阅文档的index部分了解如何删除索引。

对于Oracle、DB2和PostgreSQL，一个数据库中的索引名唯一，因此当删除索引时不必指明表名。对于Microsoft Access、Microsoft SQL Server和MySQL，一个表中索引名唯一，但在不同的表中可以使用相同的索引名，因此必须指明要删除的索引的表名。本节的例子删除代码12-1创建的索引。

### ⇒ 在Microsoft Access或MySQL中删除索引

输入：

```
DROP INDEX index
ON table;
```

*index*是要删除的索引名，*table*是索引相关表的名称（代码12-4a）。

### 代码12-4a 删除索引pub\_id\_idx (Microsoft Access或MySQL)

```
DROP INDEX pub_id_idx
ON titles;
```

### ⇒ 在Microsoft SQL Server中删除索引

输入：

```
DROP INDEX table.index;
```

*index*是要删除的索引名，*table*是索引相关表的名称（代码12-4b）。

### 代码12-4b 删除索引pub\_id\_idx (Microsoft SQL Server)

```
DROP INDEX titles.pub_id_idx
```

### ⇒ 在Oracle、DB2或PostgreSQL中删除索引

输入：

```
DROP INDEX index;
```

*index*是要删除的索引名（代码12-4c）。

### 代码12-4c 删除索引pub\_id\_idx (Oracle、DB2或PostgreSQL)

```
DROP INDEX pub_id_idx;
```

#### ✓ 提示

- 无法删除DBMS为主键约束和唯一约束自动创建的索引（见第11章）。

**视图** 图是存储的SELECT语句，它能返回基于一个或多个表（被称作基础表）检索得到的数据表。视图的重要特点如下。

- 视图的基础表可以是基本表、临时表或其他视图。
- 视图是虚拟表或导出表，有别于基本表或临时表。
- DBMS只是将视图存储为SELECT语句，而不是数据值的集合，从而防止数据冗余。
- 在SQL语句中用到时，视图才被动态创建。语句运行则存在，语句结束则消失。
- 视图是指定的数据列和数据行的集合，能使用实际表的地方几乎都可以使用视图。
- 通过视图查询没有限制。在某些情况下，视图更新引起的数据的变化可以被传递到基本表。
- 因为封闭，不管视图引用了多少基础表，也不管这些表是如何结合的，视图永远只是一个表，见2.1节。

385

### 13.1 使用CREATE VIEW 创建视图

可以将视图看作裁剪合适的、查看一个或多个基本表的窗口。这个窗口可以展示整个基本表、部分基本表或基本表的结合（或结合的一部分）。视图也可以通过其他视图查看基本表的数据——窗口之上的窗口。SQL程序员通常使用视图为数据库应用程序的终端用户展示数据。视图具有以下好处。

**简化数据访问。**视图隐藏了数据的复杂性并简化了语句，于是用户在视图上可以比直接在基本表上更容易地执行操作。如果创建了复杂视图——包含了多个基本表、联结和子查询——用户可以查询这个视图而不必理解其中复杂的关系，甚至不关心视图涉及哪些基本表。

**自动更新。**当基本表被更新后，所有引用这个表的视图都自动改变。例如，如果在表authors插入表示一个新作者的行，则所有在表authors上定义的视图将自动显示这个新作者。没有视图，DBMS将不得不存储检索来的数据以保持同步，所以这种模式节约硬盘空间并防止冗余。

**增强安全性。**视图最有用的是通过对基础表的过滤来对用户隐藏数据。假定表employees包含列salary和列commission。如果创建基于employees的视图不含这两列而包含其他无关紧要的列（如email\_address），数据库管理员就可以授权用户查看视图而不查看基础表，这样就避免了人们因好奇而查看工资数据。

**逻辑上数据独立。**基本表提供数据库的真实视图。当使用SQL创建数据库应用时，如果不向最终用户展示真实视图，就可以使用针对应用的虚拟视图。虚拟视图隐藏了数据库与应用无关的部分（整个表或某些行与列）。因此，用户在与虚拟视图交互。它尽管独立，却是从基本表的真实视图导出的。

13

虚拟视图使得应用免受到数据库设计逻辑变化的影响。假定一些应用访问表titles。有的图书已经绝版一段时间了，于是数据库设计者决定通过分离绝版书而减轻系统负担。他将表titles分成两个表——in\_print\_titles和out\_of\_print\_titles。因为无法访问titles表，相关的应用失效。

但如果那些应用访问表titles的视图而不是真实表，这个视图可以被重新定义为in\_print\_titles和out\_of\_print\_titles的合并（见9.1节）。这些视图仍能将两个新表视作它们最初的一个表，并继续工作，好像没有分离一样。（使用视图并不能使应用免受所有变化的影响。例如，视图无法避免因为表或列的删除带来的影响。）

在创建视图时，要重点考虑以下方面。

- 视图相关的SQL语句能修改数据库对象和数据，所以需要得到数据库管理员的授权才能运行。
- 视图命名遵循表命名的规则。
- 视图名称在一个模式（或数据库）中必须唯一。它们不能和别的表名或视图重名。
- 视图中的列名默认继承自基础表。可以使用AS来给视图列不同的名称，见4.2节。
- 如果与视图中的另一列同名（通常是因为视图定义包含了联结，其中两个或多个基础表的列名相同），则必须为该列指定新名。
- 视图中定义的列可以是简单列引用、字面量或包含计算或聚合函数的表达式。
- 在一些DBMS中，如果列来自数学表达式、内置函数或字面量，则必须显式指明视图中的列名。
- 视图列的数据类型继承它所源自的列或表达式。
- 可以创建的视图数量没有限制。通常，可以将对某些用户有用的数据子集创建为视图。
- 一些DBMS不允许在临时表上创建视图。
- 除了ORDER BY子句通常是被禁止的，几乎所有有效的SELECT语句都可以定义视图。
- 可以嵌套视图——或者说，视图的SELECT语句可以从其他视图中检索数据。嵌套的视图最终必须指向基本表（否则，什么也看不到）。嵌套的最大数量因DBMS而异。
- 使用视图可以方便地存储复杂的查询。将包含大量运算的查询存储为视图，就可以通过查询视图重复这些运算。
- 用视图可表示一些别无他法运行的查询。例如，可以定义联结GROUP BY视图和基础表的视图，或定义联结UNION视图和基本表的视图。
- 视图不能引用自身，因为它还不存在。
- 视图可以按不同于基础表中的格式展示数据。
- 有别于基础表，视图不支持约束。有些DBMS允许对视图建立索引。
- 通过使用SELECT \*可以定义视图，SQL自动将\*转化为所有列的列表。转化发生在视图创建时（而不是执行时），这种转化只发生一次。因此如果在基础表中加入了一列（通过使用ALTER TABLE），视图的定义不会改变。
- 因为视图不存储数据，DBMS必须在每次引用时执行它们。复杂的视图，尤其是嵌套视图，可能严重地降低效率。

#### ⇒ 创建视图

输入：

```
CREATE VIEW view [(view_columns)]
 AS select_statement;
```

*view*是要创建的视图的名称。数据库中视图的名称必须唯一。

*view\_columns*是可选项，带括号的一个或（逗号分隔的）多个列名。在*view\_columns*中的列数量必须与*select\_statement*中的SELECT子句中的列数量相同。（如果这样命名一列，就必须将列全部这样命名。）当*select\_statement*语句中的列源自一个算数表达式、函数或字面量时，就要指明*view\_columns*。两个或多个视图列可能重名（通常因为联结）时，或视图中的列名不同于它源自的基础表中的列名时，也要指明*view\_columns*。如果忽略*view\_columns*，视图就从*select\_statement*继承列名。列名也可以通过*select\_statement*的AS子句指定。在一个视图中列名必须唯一。*select\_statement*是指明视图所依据的行与列的SELECT语句。*select\_statement*可以任意复杂且使用多个表和视图。ORDER BY子句通常是被禁止的。要了解更多有关SELECT语句的知识，请查看第4章至第9章。要了解所用的DBMS对于视图中SELECT的限制，请查阅DBMS文档的CREATE VIEW部分（代码13-1至代码13-5）。 388

#### 代码13-1 创建隐藏作者个人信息（电话号码和地址）的视图

```
CREATE VIEW au_names
AS
SELECT au_id, au_fname, au_lname
FROM authors;
```

#### 代码13-2 创建视图列出那些居住在有出版社的城市的作者。注意视图使用了列名*au\_city*和*pub\_city*。重命名这些列避免了两列从基础表继承相同的列名*city*

```
CREATE VIEW cities
(au_id, au_city, pub_id, pub_city)
AS
SELECT a.au_id, a.city, p.pub_id, p.city
FROM authors a
INNER JOIN publishers p
ON a.city = p.city;
```

#### 代码13-3 创建视图，按某一出版社图书的不同类型列出总收入（=价格×销量）。因为对数学表达式显式命名而不是让DBMS分配默认名，这个视图在以后会很容易查询

```
CREATE VIEW revenues
(Publisher, BookType, Revenue)
AS
SELECT pub_id, type, SUM(price * sales)
FROM titles
GROUP BY pub_id, type;
```

#### 代码13-4 为使打印作者邮寄标签变得容易而创建视图。注意程序在SELECT子句中而不是CREATE VIEW子句中指定列名

```
CREATE VIEW mailing_labels
AS
SELECT
 TRIM(au_fname || ' ' || au_lname) AS "address1",
 TRIM(address) AS "address2",
 TRIM(city) || ',' || TRIM(state) ||
 ' ' || TRIM(zip)
```

```
AS "address3"
FROM authors;
```

**代码13-5** 创建列出作者A02和A05的姓及两个人所写（或合写）图书的视图。注意这个语句使用了嵌套视图，它引用代码13-1创建的视图au\_names。

```
CREATE VIEW au_titles (LastName, Title)
AS
SELECT an.au_lname, t.title_name
FROM title_authors ta
INNER JOIN au_names an
ON ta.au_id = an.au_id
INNER JOIN titles t
ON t.title_id = ta.title_id
WHERE an.au_id IN ('A02', 'A05');
```

### ✓ 提示

- 不能创建临时视图。视图和临时表的持久性是不同的。视图存在于SQL语句的生存期，临时表存在于进程的生存期，参见11.10节。
- 标准SQL没有ALTER VIEW语句。如果在视图创建之后，基础表或视图发生了改变，就要删除并重建这个视图。然而，Microsoft SQL Server、Oracle、DB2、MySQL和PostgreSQL都支持非标准的ALTER VIEW语句。
- **DBMS** 在Microsoft Access中运行CREATE VIEW语句时，视图将作为一个查询对象出现在数据库窗口。为了运行代码13-4，将每个||变为+，参见5.4节的DBMS提示。为了运行代码13-5，输入：

```
CREATE VIEW au_titles
(LastName, Title)
AS
SELECT an.au_lname, t.title_name
FROM au_names an
INNER JOIN (titles t
INNER JOIN title_authors ta
ON t.title_id = ta.title_id)
ON an.au_id = ta.au_id
WHERE an.au_id IN ('A02', 'A05');
```

为了在Microsoft SQL Server中运行代码13-1至代码13-5，要删除每个语句的结束分号。另外，为了运行代码13-4，要将每一个||变为+，将每一个TRIM(x)变为LTRIM(RTRIM(x))，参见5.4节和5.7节中的DBMS提示。

为了在Oracle 8i和之前的版本运行代码13-2和代码13-5，要使用WHERE语法而不是JOIN语法。对于代码13-2输入：

```
CREATE VIEW cities
(au_id, au_city, pub_id, pub_city)
AS
SELECT a.au_id, a.city,
p.pub_id, p.city
FROM authors a, publishers p
WHERE a.city = p.city;
```

对于代码13-5：

```

CREATE VIEW au_titles
 (LastName, Title)
AS
 SELECT an.au_lname, t.title_name
 FROM title_authors ta,
 au_names an, titles t
 WHERE ta.au_id = an.au_id
 AND t.title_id = ta.title_id
 AND an.au_id IN ('A02', 'A05');

```

为了在DB2中运行代码13-4，应将所有`TRIM(x)`变为`LTRIM(RTRIM(x))`，参见5.7节的DBMS提示。为了在MySQL中运行代码13-4，应使用`CONCAT()`函数而不是联接操作符`||`，参见5.4节的DBMS提示。MySQL 5.0和之后版本支持视图，之前版本不能运行本节的代码。（为了在之前版本隐藏数据，可使用MySQL的权限系统来限制列的访问。）

在Microsoft SQL Server、Oracle、DB2、MySQL和PostgreSQL中，创建视图时可以加入可选的`[CASCADED | LOCAL] CHECK OPTION`子句。这个子句只用于可修改的视图，并确保只有能够被视图读取的数据可以被插入、更新或删除，参见13.3节。如果视图显示来自纽约州的作者，就不可能通过视图插入、更新或删除非纽约州的作者。`CASCDED`和`LOCAL`选项只能用于嵌套视图。`CASCDED`执行对当前视图和所有引用视图的检查。`LOCAL`只执行对当前视图的检查。

390

## 13.2 通过视图检索数据

创建视图并不会有任何显示。`CREATE VIEW`所做的是让DBMS用一个命名的`SELECT`语句存储视图。要通过视图查看数据，就要像查询表一样使用`SELECT`查询这个视图。可以：

- 对`SELECT`子句显示的列重新排序。
- 使用操作符和函数来执行运算。
- 使用`AS`改变列标题。
- 使用`WHERE`过滤行。
- 使用`GROUP BY`对行分组。
- 使用`HAVING`对分组的行过滤。
- 使用`JOIN`联结视图和别的视图、表或临时表。
- 使用`ORDER BY`对结果排序。

391

### ⇒ 要通过视图检索数据

输入：

```

SELECT columns
 FROM view
 [JOIN joins]
 [WHERE search_condition]
 [GROUP BY group_columns]
 [HAVING search_condition]
 [ORDER BY sort_columns];

```

13

`view`是要查询的视图的名称。涉及这些视图的子句所遵循的规则和表相同，详见第4章至第9章。代码13-6至13-11和图13-1至13-6显示如何通过13.1节中的代码13-1至13-5创建的视图检索数据。

**代码13-6** 列出视图au\_titles的所有行和列，结果见图13-1

```
SELECT *
 FROM au_titles;
```

| LastName  | Title                         |
|-----------|-------------------------------|
| Kells     | Ask Your System Administrator |
| Heydemark | How About Never?              |
| Heydemark | I Blame My Mother             |
| Heydemark | Not Without My Faberge Egg    |
| Heydemark | Spontaneous, Not Annoying     |

图13-1 运行代码13-6的结果

**代码13-7** 列出视图cities中不同的城市，结果见图13-2

```
SELECT DISTINCT au_city
 FROM cities;
```

| au_city       |
|---------------|
| New York      |
| San Francisco |

图13-2 运行代码13-7的结果

**代码13-8** 列出平均销售收入超过一百万美元的图书类型，结果见图13-3

```
SELECT BookType,
 AVG(Revenue) AS "AVG(Revenue)"
 FROM revenues
 GROUP BY BookType
 HAVING AVG(Revenue) > 1000000;
```

| BookType   | AVG(Revenue) |
|------------|--------------|
| biography  | 18727318.50  |
| computer   | 1025396.65   |
| psychology | 2320933.76   |

图13-3 运行代码13-8的结果

**代码13-9** 列出名字中包含Kell的作者邮寄地址的第三行，结果见图13-4

```
SELECT address3
 FROM mailing_labels
 WHERE address1 LIKE '%Kell%';
```

| address3            |
|---------------------|
| New York, NY 10014  |
| Palo Alto, CA 94305 |

图13-4 运行代码13-9的结果

**代码13-10** 列出至少不是一本书主要作者的名字，结果见图13-5

```
SELECT DISTINCT an.au_fname, an.au_lname
 FROM au_names an
 INNER JOIN title_authors ta
 ON an.au_id = ta.au_id
 WHERE ta.au_order > 1;
```

| au_fname | au_lname |
|----------|----------|
| Hallie   | Hull     |
| Klee     | Hull     |

图13-5 运行代码13-10的结果

**代码13-11** 列出来自加利福尼亚州的所有作者的名字，结果见图13-6

```
SELECT au_fname, au_lname
 FROM au_names
```

```
WHERE state = 'CA';
```

ERROR: Invalid column name 'state'.

图13-6 运行代码13-11的结果。视图au\_names引用表authors，但隐藏了列state。于是通过视图引用state引发了错误

### ✓ 提示

- DBMS** 要在Microsoft Access中运行代码13-9，要使用双括号和方括号括起视图的列名。

```
SELECT ["address3"]
FROM mailing_labels
WHERE ["address1"] LIKE '%Kell%';
```

要在Oracle和DB2中运行代码13-9，应使用双引号括起视图列名。

```
SELECT "address3"
FROM mailing_labels
WHERE "address1" LIKE '%Kell%';
```

MySQL 5.0和之后的版本支持视图，之前版本无法运行本节的代码。

392  
393

## 13.3 通过视图修改数据

可修改视图是指可以使用插入、更新和删除操作来改变基础表中数据的视图。对可修改表做的任何变化都将明确地传递到基础表。视图的插入、更新和删除语法与表的相同，参见第10章。

不可修改（只读）视图是不支持插入、更新和删除操作的视图，因为对数据的改动可能是含糊不清的。要改变出现在只读视图的数据，必须直接修改基础表（或通过其他可修改视图）。

可修改视图的每一行都和基础表的一行相关联。如果视图的SELECT语句使用了GROUP BY、HAVING、DISTINCT或聚合函数等，视图就成为不可更新的。

SQL-92规定可更新视图只能定义在一张表上，这样过于严格但很安全。SQL:1999放宽了限制，因为存在样式繁多的可更新视图。随着标准的发布，DBMS供应商扩展了可更新视图。单一表视图总是可更新的。DBMS检查基础表的联结和多表视图的引用完整性约束来决定视图是否可更新。以下这些类型的查询可以定义为可更新的视图。

- 一对内联结
- 一对外联结
- 一对多内联结
- 一对多外联结
- 多对多联结
- UNION和EXCEPT查询

本节的例子使用仅引用一个基础表的可更新视图。请查阅DBMS文档了解哪种多表视图是可更新的，以及它们的更新如何影响到每个基础表。

13

### 13.3.1 通过视图插入行

视图ny\_authors由来自纽约州作者的ID、名字和州名组成（代码13-12和图13-7）。ny\_authors仅

394

引用基础表authors。

**代码13-12 创建和显示视图ny\_authors，列出来自纽约州作者的ID、名字和州，结果见图13-7**

```
CREATE VIEW ny_authors
AS
SELECT au_id, au_fname, au_lname, state
FROM authors
WHERE state = 'NY';

SELECT *
FROM ny_authors;
```

| au_id | au_fname  | au_lname | state |
|-------|-----------|----------|-------|
| A01   | Sarah     | Buchman  | NY    |
| A05   | Christian | Kells    | NY    |

图13-7 运行代码13-12的结果：视图ny\_authors

代码13-13通过视图插入一个新行。DBMS向表authors插入一个新行。行的au\_id列是A08, au\_fname是Don, au\_lname是Dawson, state是NY。行的其他列——phone、address、city和zip设置为空值（或它们的默认值，如果默认值存在）。

**代码13-13 通过视图ny\_authors插入一个新行**

```
INSERT INTO ny_authors
VALUES('A08', 'Don', 'Dawson', 'NY');
```

代码13-14类似于代码13-13，通过视图插入行。但这次新作者来自加州，而不是纽约，它违反了视图定义的WHERE条件。DBMS是插入行还是取消操作？这取决于视图是如何创建的。在本例中，因为CREATE VIEW语句（见代码13-12）没有WITH CHECK OPTION子句，于是DBMS并不强制保持与视图最初定义的一致性，插入是允许的。要了解更多WITH CHECK OPTION的知识，请参见13.1节的DBMS提示。

395

如果ny\_authors如下定义所示，DBMS将取消插入。

```
CREATE VIEW ny_authors
AS
SELECT au_id, au_fname, au_lname,
 state
 FROM authors
 WHERE state = 'NY'
 WITH CHECK OPTION;
```

**代码13-14 通过视图ny\_authors插入一个新行。如果创建ny\_authors时使用了WITH CHECK OPTION，DBMS将取消插入**

```
INSERT INTO ny_authors
VALUES('A09', 'Jill', 'LeFlore', 'CA');
```

### 13.3.2 通过视图更新行

代码13-15通过视图更新了已存在的行。DBMS将作者名字由Sarah Buchman变为Yasmin Howcomely，更改了表中authors作者A01所在行。这行的其他列值——au\_id、phone、address、city、state和zip并无变化。

**代码13-15 通过视图ny\_authors更新已存在的行**

```
UPDATE ny_authors
SET au_fname = 'Yasmin',
```

```
au_lname = 'Howcomely'
WHERE au_id = 'A01';
```

假定代码13-15如下。

```
UPDATE ny_authors
SET au_fname = 'Yasmin',
 au_lname = 'Howcomely',
 state = 'CA'
WHERE au_id = 'A01';
```

这个语句与代码13-14有同样的问题：所需要的变化会使Yasmin行不再满足视图成员资格的条件，DBMS是接受还是取消修改，取决于视图创建时是否指明WITH CHECK OPTION子句。如果使用了WITH CHECK OPTION，行无法以使它们从视图上消失的方法修改。

396

- 有些计算导出列可以更新（理论上）。例如，视图包含列bonus， $bonus = 0.1 * salary$ ，应该能够修改bonus而使SQL应用反函数（ $bonus/0.1$ ）来更新基础表中的salary。其实这是不可行的，因为SQL无法反向更新导出列。
- 对于复杂的可更新视图，一种操作可以包含另一种操作。例如，视图更新可能包含插入新的基本表行。
- 为了在Microsoft SQL Server中运行代码13-12，请删除CREATE VIEW语句结束的分  
**DBMS**号，并分别运行两条语句。

MySQL 5.0及之后版本支持视图。之前版本无法运行本节的代码。

PostgreSQL没有提供可更新视图，但可以利用它的规则系统通过定义ON INSERT、ON UPDATE和ON DELETE的规则来创建可更新视图的影像（illusion）。查阅PostgreSQL文档的CREATE RULE或rule system部分。

对于所有的DBMS，查阅文档了解DBMS如何处理可更新视图中自动产生唯一行标识符的列，参见3.12节。

### 13.3.3 通过视图删除行

代码13-16通过视图删除一行。DBMS删除了表authors中作者A05所在的行。（行的每一列都删除了，不只是出现在视图中的列）。接下来，行从视图ny\_authors中消失了。

#### 代码13-16 通过视图ny\_authors删除行

```
DELETE FROM ny_authors
WHERE au_id = 'A05';
```

视图更新会影响完整性。如果删除行影响到完整性约束，DBMS不允许删除，参见11.7节。如果删除行后，所有相关基础表的外键约束仍满足，则删除成功。一些更新能够在外键约束有CASCADE选项（如果指明）的情况下执行，而不是通过视图定义。

13

例如，在代码13-16中，如果没有首先改变或删除表title\_authors中指向的表authors中作者A05的外键值，DBMS就会取消删除。

#### ✓ 提示

- 可更新的视图必须包含基础表的键来保证每个视图行指向基本表的一行。
- 可更新视图不包含的列必须可以为空或在基础表有一个默认值，以便DBMS能够在插入行时构

建整行。

- 397  更新值必须符合基础表列的限制，如数据类型、可否为空及其他约束。

## 13.4 使用DROP VIEW删除视图

使用DROP VIEW语句删除视图。因为视图物理上独立于它引用的表，在任何时候删除视图都不影响表。然而，所有SQL程序、应用和引用到这个删除视图的其他视图将无法正常运行。

### » 删除视图

输入：

```
DROP VIEW view;
```

*view*是要删除的视图名称（见代码13-17）。

#### 代码13-17 删除视图ny\_authors

```
DROP VIEW ny_authors;
```

#### ✓ 提示

删除表不会删除引用这个表的视图，所以必须使用DROP VIEW显式删除视图，参见11.13节。

**DBMS** MySQL 5.0和之后版本支持视图。之前版本无法运行本节的代码。

**事 务** (transaction) 是一个或多个接连在一起作为一个逻辑单位运行的SQL语句。DBMS认为事务是不可分割的，要么全部执行，要么全不执行。

用银行的例子来说明事务的重要性是很经典的。假定某个客户从他的储蓄账户向支票账户转账500美元。这个操作包括连续执行的两个独立步骤。

- (1) 储蓄账户减500美元。
- (2) 支票账户增500美元。

图14-1显示了这个事务的两条SQL语句。设想，如果DBMS在执行了第一条语句，但没有执行第二条时突然发生故障——断电、系统崩溃、硬件出问题，账户在神不知鬼不觉的情况下出现了不平衡。渎职控告和牢狱之灾就会接踵而至。

```
UPDATE savings_accounts
SET balance = balance - 500.00
WHERE account_number = 1009;

UPDATE checking_accounts
SET balance = balance + 500.00
WHERE account_number = 6482;
```

图14-1 当银行客户从储蓄账户向支票账户转账时，必须有两条SQL语句

为了避免违法记录，应该使用事务来保证两条SQL语句都被执行，以维持账户平衡。如果事务中的一条语句无法执行时，DBMS将撤销（回滚）事务中其他语句。如果一切顺利，变化将被持久化（提交）。

399

## 执行事务

要了解事务如何工作，就要了解一些术语。

**提交**。提交（committing）事务是使自事务开始后修改的所有数据持久化在数据库中。在事务提交后，即使发生崩溃或其他故障，事务带来的所有变化仍然对其他用户可见并能够保证持久化。

**回滚**。回滚（rolling back）事务是撤销事务中SQL语句带来的所有变化。事务回滚后，此前影响到的数据回到原状，就好像SQL语句从未执行一样。

14

**事务日志。**事务日志文件（transaction logfile）或日志（log）是有关事务对数据库进行修改的一系列记录。事务日志记录了每个事务开始、数据的变化以及撤销或重新执行事务（如果将来需要）的足够信息。日志随着数据库事务的执行不断增长。

尽管保证每个事务本身的完整性是DBMS的职责，但依据组织或公司规章来开始和结束事务以保证数据逻辑的一致性则是数据库开发人员的责任。事务应该仅包含能做出一致修改的必要的SQL语句——不多不少。所有引用表中的数据在事务开始前和事务结束后必须保持一致。

在设计和执行事务时，要重点考虑以下方面。

- 事务相关的SQL语句会修改数据，所以执行事务要得到数据库管理员的授权。
- 事务过程应用于那些改变数据和数据库对象的语句（INSERT、UPDATE、DELETE、CREATE、ALTER、DROP——因不同DBMS而异）。对于工作中用到的数据库，每一条这样的语句都应该作为事务的一部分执行。
- 提交了的事务被称作持久化，意味着永久性改变，即便系统发生故障仍能保持。
- DBMS的数据恢复机制依赖于事务。当DBMS在故障之后被在线复原，DBMS检查事务日志确认是否所有事务都提交给了数据库。如发现没有提交（部分执行）的事务，依据日志将它们回滚。必须重新提交回滚的事务（尽管一些DBMS能够自动完成没有结束的事务）。
- DBMS的备份/恢复设备依赖于事务。备份设备获得例行的数据库快照并将它们和随后的事务日志存储在备份盘上。假定使用的硬盘发生故障使得数据和事务日志不可读。可以借助于恢复设备，它将采用最近的数据库备份并执行，或前滚所有从快照到故障前最后执行并在日志中提交的事务。这个恢复操作使数据库恢复到故障发生前的正确状态（注意，要再次提交没有提交的事务）。
- 显然，应该将数据库和它的事务日志存储于不同的物理硬盘。

400

### 并发控制

对人来说，计算机好像在同一时间运行着两个或更多进程。实际上，计算机操作并非同时发生，而是连续的。同时发生的印象是因为微处理器在人们难以察觉的很短的时间段内工作。在DBMS里，并发控制是在两个或更多用户同时访问或修改相同的数据时为防止数据失去完整性的一组策略。

DBMS使用锁定策略来保证事务完整性和数据库的一致性。在读写操作时，锁定限制数据的访问；于是，它阻止用户读那些正在被其他用户修改的数据，并防止多用户同时对同一数据修改。如果没有锁定，数据可能发生逻辑错误，针对这些数据执行的语句将返回不可预料的结果。偶尔会出现两个用户都锁定了对方事务所需的数据并尝试去得到对方的解锁，这时发生死锁问题。大多数DBMS能够侦测和解决死锁问题，通过回滚一个用户的事务让另一个事务可以运行（否则，两个用户都要永远等对方解锁）。锁定机制非常复杂，请查阅DBMS文档了解锁定。

并发透明性是从事务的角度看数据库上运行唯一事务的现象。DBMS分离事务变化与任何其他并发事务的变化。当然，事务永远见不到数据的中间状态；或在其他并发事务之前访问，或在其他并发事务结束以后访问。分离的事务允许重载开始数据并再次执行（前滚）一系列事务来达到它们在最初的事物被执行之后的状态。

401

因为事务按照要么全部，要么全不方式被执行，事务的边界（开始点和结束点）必须清晰。边界使DBMS作为一个原子单元来执行这些语句。事务隐式开始于第一个可执行的SQL语句或显式使用

START TRANSACTION语句。事务显式结束于COMMIT或ROLLBACK语句（无法隐式结束），且无法在提交之后回滚事务。

- **DBMS** Oracle和DB2的事务总是隐式开始，这些DBMS没有用来开始事务的语句。在Microsoft Access、Microsoft SQL Server、MySQL和PostgreSQL中，可以使用BEGIN语句显式开始事务。SQL:1999引入START TRANSACTION语句——由于这发生在DBMS使用BEGIN开始事务很久以后，因此不同DBMS扩展BEGIN的语法也各不相同。MySQL和PostgreSQL支持START TRANSACTION（作为BEGIN同义词）。

#### » 显式开始一个事务

在Microsoft Access或Microsoft SQL Server中，输入：

```
BEGIN TRANSACTION;
```

或

在MySQL or PostgreSQL，输入：

```
START TRANSACTION;
```

#### » 提交事务

输入：

```
COMMIT;
```

#### » 回滚事务

输入：

```
ROLLBACK;
```

代码14-1中的SELECT语句显示UPDATE操作被DBMS执行后又被ROLLBACK语句取消。结果见图14-2。

**代码14-1** 在一个事务内，更新操作（像插入和删除操作那样）永远不是在最后出现。结果见图14-2

```
SELECT SUM(pages), AVG(price) FROM titles;

BEGIN TRANSACTION;
 UPDATE titles SET pages = 0;
 UPDATE titles SET price = price * 2;
 SELECT SUM(pages), AVG(price) FROM titles;
ROLLBACK;

SELECT SUM(pages), AVG(price) FROM titles;
```

代码14-2显示更实用的事务例子。要从表publishers删除出版社P04而不产生引用完整性错误。因为表titles的有些外键值指向表publishers的出版社P04，所以要先删除表titles、titles\_authors、和royalties中相关的行。应该使用事务保证所有DELETE语句都被执行。如果只有一些语句执行成功，数据将无法保持一致（要了解更多有关引用完整性检查的信息，参见11.7节）。

| SUM(pages) AVG(price) |         |
|-----------------------|---------|
| -----                 | -----   |
| 5107                  | 18.3875 |
| SUM(pages) AVG(price) |         |
| -----                 | -----   |
| 0                     | 36.7750 |
| SUM(pages) AVG(price) |         |
| -----                 | -----   |
| 5107                  | 18.3875 |

图14-2 运行代码14-1的结果。SELECT语句的结果显示DBMS取消了操作

## 代码14-2 使用事务从表publishers中删除出版社P04，及删除其他表中与P04相关的行

```

BEGIN TRANSACTION;

DELETE FROM title_authors
WHERE title_id IN
(SELECT title_id
FROM titles
WHERE pub_id = 'P04');

DELETE FROM royalties
WHERE title_id IN
(SELECT title_id
FROM titles
WHERE pub_id = 'P04');

DELETE FROM titles
WHERE pub_id = 'P04';

DELETE FROM publishers
WHERE pub_id = 'P04';

COMMIT;

```

## ACID

ACID是首字母缩写，它概括了事务的特点：

**原子性 (Atomicity)**。要么事务中所有的数据修改都执行，要么都撤销。

**一致性 (Consistency)**。完全的事务应让数据保持一致来保证数据完整。一致状态要保证满足所有数据约束。(注意，并不要求在任何事务的中间点保持一致性)。

**隔离性 (Isolation)**。事务的影响独立(或隐藏)于其他事务，参见14.1节“并发控制”提要栏。

**持久性 (Durability)**。在事务完成后，它的影响是永久和持续的——即便是系统崩溃。

事务理论是独立于关系模型的重大问题。由Jim Gray和Andreas Reuter所著的*Transaction Processing: Concepts and Techniques* (Morgan Kaufmann) 是一本很好的参考书。<sup>①</sup>

403

## ✓ 提示

- 不要忘记使用COMMIT或ROLLBACK显式结束事务。没有结束点将导致回滚的最后未提交事务巨大，可能带来意外的数据变化或程序异常中止。因为事务在生存期锁定行、整个表、索引和其他资源，所以要让事务尽可能小。COMMIT或ROLLBACK为其他事务释放资源。

- 可以嵌套事务。最大嵌套数因DBMS而异。

使用一条SET语句的UPDATE更新多个列比使用多个UPDATE快。例如，查询：

```

UPDATE mytable
SET col1 = 1
 col2 = 2
 col3 = 3
WHERE col1 <> 1
 OR col2 <> 2

```

① 该书影印版《事务处理》由人民邮电出版社出版。——编者注

OR col3 < 3;

比3个UPDATE语句好，因为它减少了日志记录（尽管带来锁定）。

- 默认情况下，DBMS运行在自动提交模式（autocommit mode），除非被其他显式或隐式事务重写（或被系统设置关闭）。在这种模式下，每一条语句作为一个事务执行。如果语句执行成功，DBMS就将它提交；如果DBMS遇到错误，就回滚这条语句。
- 对于长的事务，可以设置称为存储点（savepoints）的中间标志，将事务分割为小段。存储点允许回滚从事务当前点到事务靠前的时间点之间的变化（假定事务还没有提交）。如果在一系列复杂的插入、更新、删除操作未提交时，意识到最后的变化是不正确的或不必要的，使用存储点就可以避免重新提交所有语句。Microsoft Access不支持存储点。在Oracle、DB2、MySQL和PostgreSQL中，使用语句：

`SAVEPOINT savepoint_name`

对于Microsoft SQL Server，使用：

`SAVE TRANSACTION savepoint_name;`

查阅DBMS文档来了解有关存储点锁定的细节知识及如何COMMIT或ROLLBACK到特定的存储点。

- Microsoft Access中，通过SQL视图窗口或DAO无法执行事务，必须使用Microsoft Jet DBMS OLE DB Provider和ADO。

Oracle和DB2是隐式开始事务的。为了在Oracle 和 DB2中运行代码14-1和代码14-2，要删除语句：BEGIN TRANSACTION；

为了在MySQL中运行代码14-1和代码14-2，将语句BEGIN TRANSACTION;变为START TRANSACTION;（或BEGIN）。

MySQL通过InnoDB和BDB表支持事务，请查阅MySQL文档了解事务。Microsoft SQL Server、Oracle、MySQL和PostgreSQL支持SET TRANSACTION语句设置事务特征。DB2通过服务器层和连接初始化设置控制事务特征。

404

14

本章介绍一些解决常见问题的SQL程序，这些SQL程序涉及：

- 使用复杂或明智的SQL元素结合；
- 使用非标准的（特定DBMS）SQL元素来取代标准SQL曲折费事的方案。

本书称这些查询为技巧，但其实任何有经验的程序员应该掌握的。本章介绍的查询技术，在下

405 面“高级SQL图书”提要栏所列图书中均有详细介绍，读者可以进一步参考这些书籍。

#### 高级SQL图书

*Inside Microsoft SQL Server 2005: T-SQL Querying* by Itzik Ben-Gan, et al. (Microsoft Press)

*Joe Celko's SQL for Smarties* by Joe Celko (Morgan Kaufmann)

*SQL Hacks* by Andrew Cumming, Gordon Russell (O'Reilly)

*MySQL Cookbook* by Paul DuBois (O'Reilly)

*The Guru's Guide to Transact-SQL* by Ken Henderson (Addison-Wesley)

*SQL Cookbook* by Anthony Molinaro (O'Reilly)

*The Essence of SQL* by David Rozenshtein (Coriolis)

*Optimizing Transact-SQL* by David Rozenshtein, et al. (SQL Forum Press)

*Developing Time-Oriented Database Applications in SQL* by Richard T. Snodgrass (Morgan Kaufmann)

*Transact-SQL Cookbook* by Ales Spetic, Jonathan Gennick (O'Reilly)

## 15.1 动态统计

动态统计（running statistic）是使用不断增多的值，一行接一行地进行计算，开始是对一个值（第一个值），接下来是按照它们被提交的顺序对更多的值，最后是对所有值进行计算。动态合计（running sum）和动态平均（running average）（均值）是最常见的动态统计。

代码15-1按照不断增加的数据项对图书销量计算动态合计和动态平均。这个查询将两个titles表的实例交叉联结，按照第一个表（t1）中图书的ID分组，并限定第二张表（t2）行的ID值小于或等于联结表t1行的值。应用SUM()、AVG()和COUNT()函数的中间交叉联结表，像如下这样：

```
t1.id t1.sales t2.id t2.sales
----- -----
T01 566 T01 566
T02 9566 T01 566
```

```

T02 9566 T02 9566
T03 25667 T01 566
T03 25667 T02 9566
T03 25667 T03 25667
T04 13001 T01 566
T04 13001 T02 9566
T04 13001 T03 25667
T04 13001 T04 13001
T05 201440 T01 566
...

```

注意，动态统计对于T10没有变化，因为它的销量是空值。因为没有指明，GROUP BY不会自动对结果排序，所以ORDER BY子句是必要的，结果见图15-1。

406

**代码15-1 计算动态合计、均值和图书销量。结果见图15-1**

```

SELECT
 t1.title_id,
 SUM(t2.sales) AS RunSum,
 AVG(t2.sales) AS RunAvg,
 COUNT(t2.sales) AS RunCount
FROM titles t1, titles t2
WHERE t1.title_id >= t2.title_id
GROUP BY t1.title_id
ORDER BY t1.title_id;

```

移动平均（moving average）是一种时间序列平滑法（如一段时间内的股票价格列表），它将每一个值重置为这个值与邻近其他值的平均值。如果包含有像time\_series表一样的整数或日期序列，那么计算移动平均值就很容易。

| seq | price |
|-----|-------|
| 1   | 10.0  |
| 2   | 10.5  |
| 3   | 11.0  |
| 4   | 11.0  |
| 5   | 10.5  |
| 6   | 11.5  |
| 7   | 12.0  |
| 8   | 13.0  |
| 9   | 15.0  |
| 10  | 13.5  |
| 11  | 13.0  |
| 12  | 12.5  |
| 13  | 12.0  |
| 14  | 12.5  |
| 15  | 11.0  |

| title_id | RunSum  | RunAvg | RunCount |
|----------|---------|--------|----------|
| T01      | 566     | 566    | 1        |
| T02      | 10132   | 5066   | 2        |
| T03      | 35799   | 11933  | 3        |
| T04      | 48800   | 12200  | 4        |
| T05      | 250240  | 50048  | 5        |
| T06      | 261560  | 43593  | 6        |
| T07      | 1761760 | 251680 | 7        |
| T08      | 1765855 | 220731 | 8        |
| T09      | 1770855 | 196761 | 9        |
| T10      | 1770855 | 196761 | 9        |
| T11      | 1864978 | 186497 | 10       |
| T12      | 1964979 | 178634 | 11       |
| T13      | 1975446 | 164620 | 12       |

图15-1 运行代码15-1的结果

代码15-2计算价格的移动平均值，结果见图15-2。结果中移动平均列里的每个值是5个值（当前行的价格和它前面4行的价格（按照seq列的排序））的平均值。因为最前面4行的前面没有必要值，它们被忽略了。可以调整WHERE子句里的值来包括任何跨度的平均区间。例如，要让代码15-2使用每个价格和它前面的两个价格以及它后面的两个价格来计算五点移动平均值，可将WHERE子句改为：

```
WHERE t1.seq >= 3
AND t1.seq <= 13
AND t1.seq BETWEEN t2.seq - 2 AND
t2.seq + 2
```

**代码15-2 计算五点移动平均值。结果见图15-2**

```
SELECT t1.seq, AVG(t2.price) AS MovingAvg
FROM time_series t1, time_series t2
WHERE t1.seq >= 5
AND t1.seq BETWEEN t2.seq AND
t2.seq + 4
GROUP BY t1.seq
ORDER BY t1.seq;
```

如果表中已经有了动态总计，就可以计算前后两行总计数之间的差值。代码15-3从以下被称作roadtrip的表得出城市间距离。这个表包含从Washington州的Seattle到California州的San Diego每一段旅程的累计距离，结果见图15-3。

| seq | city              | miles |
|-----|-------------------|-------|
| 1   | Seattle, WA       | 0     |
| 2   | Portland, OR      | 174   |
| 3   | San Francisco, CA | 808   |
| 4   | Monterey, CA      | 926   |
| 5   | Los Angeles, CA   | 1251  |
| 6   | San Diego, CA     | 1372  |

| seq | MovingAvg |
|-----|-----------|
| 5   | 10.6      |
| 6   | 10.9      |
| 7   | 11.2      |
| 8   | 11.6      |
| 9   | 12.4      |
| 10  | 13.0      |
| 11  | 13.3      |
| 12  | 13.4      |
| 13  | 13.2      |
| 14  | 12.7      |
| 15  | 12.2      |

图15-2 运行代码15-2的结果

**代码15-3 依据累计距离计算城市间的距离。结果见图15-3**

```
SELECT
t1.seq AS seq1,
t2.seq AS seq2,
t1.city AS city1,
t2.city AS city2,
t1.miles AS miles1,
t2.miles AS miles2,
t2.miles - t1.miles AS dist
FROM roadtrip t1, roadtrip t2
WHERE t1.seq + 1 = t2.seq
ORDER BY t1.seq;
```

| seq1 | seq2 | city1             | city2             | miles1 | miles2 | dist |
|------|------|-------------------|-------------------|--------|--------|------|
| 1    | 2    | Seattle, WA       | Portland, OR      | 0      | 174    | 174  |
| 2    | 3    | Portland, OR      | San Francisco, CA | 174    | 808    | 634  |
| 3    | 4    | San Francisco, CA | Monterey, CA      | 808    | 926    | 118  |
| 4    | 5    | Monterey, CA      | Los Angeles, CA   | 926    | 1251   | 325  |
| 5    | 6    | Los Angeles, CA   | San Diego, CA     | 1251   | 1372   | 121  |

图15-3 运行代码15-3的结果

✓ 提示

- 如果分组列包含有重复值，代码15-1和代码15-2将给出不准确的结果。
- 通过第8章的代码8-21可以了解计算动态统计的另一种方法。
- **DBMS** 在Oracle和DB2中，可以使用窗口函数来计算动态统计，例如：

```
SELECT title_id, sales,
 SUM(sales) OVER (ORDER BY title_id)
 AS RunSum
 FROM titles
 ORDER BY title_id;
```

408

## 15.2 产生序列

在3.12节中介绍了用自动产生的整数序列来创建标识列（主键是典型的应用）。SQL标准提供序列发生器来创建这样的序列。

### ⇒ 创建序列发生器

输入：

```
CREATE SEQUENCE seq_name
 [INCREMENT [BY] increment]
 [MINVALUE min | NO MINVALUE]
 [MAXVALUE max | NO MAXVALUE]
 [START [WITH] start]
 [[NO] CYCLE];
```

*seq\_name*是要创建的序列名称（唯一标识符）。

*increment*指明在当前序列值上加上多少来创建一个新值。正数会创建递增的序列，负数会创建递减的序列。*increment*的值不能为零。如果子句INCREMENT BY被省略，那么默认递增值是1。

*min*指明序列能够创建的最小值。如果子句省略MINVALUE或指明NO MINVALUE，使用默认最小值。默认最小值取决于DBMS，但对于递增数列它们通常为1，对于递减数列通常是一个非常大的数字。

*max (> min)* 指明序列能产生的最大值。如果省略子句MAXVALUE或指明NO MAXVALUE，将使用默认最大值。默认最大值取决于DBMS，对于递增序列它们通常是一个非常大的数，对于递减数列它们通常是-1。

*start*指明序列的第一个值。如果省略子句START WITH，默认开始值对于递增序列是*min*，对于递减序列是*max*。

· CYCLE表示在达到最大或最小值时序列继续产生值。当递增数列达到最大值时，序列产生最小值。当递减数列达到最小值时，序列产生最大值。

NO CYCLE（默认）表示在达到最大或最小值后，序列不再产生值。

409

代码15-4定义如图15-4所示的序列。可以用几种方式应用序列发生器。SQL标准提供了内置函数NEXT VALUE FOR来增加序列的值，如：

```
INSERT INTO shipment(
 part_num,
 desc,
 quantity)
```

15

```
VALUESC
NEXT VALUE FOR part_seq,
'motherboard',
5);
```

如果要创建值唯一的列，可以直接在CREATE TABLE语句中使用关键字IDENTITY来定义序列。

```
CREATE TABLE parts (
part_num INTEGER AS
IDENTITY(INCREMENT BY 1
MINVALUE 1
MAXVALUE 10000
START WITH 1
NO CYCLE),
desc AS VARCHAR(100),
quantity INTEGER;
```

这个表定义允许在插入行时忽略NEXT VALUE FOR。

```
INSERT INTO shipment(
desc,
quantity)
VALUESC
'motherboard',
5);
```

SQL还提供了ALTER SEQUENCE和DROP SEQUENCE来改变和删除序列发生器。

#### 代码15-4 创建产生从1到10 000连续整数的序列发生器。结果见图15-4

```
CREATE SEQUENCE part_seq
INCREMENT BY 1
MINVALUE 1
MAXVALUE 10000
START WITH 1
NO CYCLE;
```

#### ✓ 提示

**DBMS** Oracle、DB2 和 PostgreSQL 支持 CREATE SEQUENCE、ALTER SEQUENCE 和 DROP SEQUENCE。在Oracle中，使用NOCYCLE而不是NO CYCLE。读者可以查阅自己使用DBMS的文档来了解系统如何使用序列。大多数DBMS因为有其他（前SQL:2003）定义列值唯一的方法，它们不支持IDENTITY列，参见3.12节中的表3-18。PostgreSQL的generate\_series()函数提供了一种快速产生数字标识行的方法。

包含连续整数序列的一列表使原本因SQL计算能力有限而很复杂的问题变得简单。序列表不是数据模型的组成部分——它们是辅助表，它们依赖于查询和别的“真实”表。

可以使用任何一种前面介绍过的方法来创建序列表。代码15-5也可以用来创建序列表，它使用中间表temp09和自身的交叉联结来创建序列表seq。CAST表达式连接数字字符为连续数字，然后将它们转为整数。在表seq创建后，可以删除表temp09。图15-5显示运行结果。表seq包含整数序列0, 1, 2, …, 9 999。可以通过改变INSERT INTO seq语句的SELECT和FROM表达式对这个序列收缩或放大。

|       |
|-------|
| 1     |
| 2     |
| 3     |
| ...   |
| 9998  |
| 9999  |
| 10000 |

图15-4 代码15-4产生的序列

**代码15-5** 创建只包含连续整数的一列表。结果见图15-5

```

CREATE TABLE temp09 (
 i CHAR(1) NOT NULL PRIMARY KEY
);

INSERT INTO temp09 VALUES('0');
INSERT INTO temp09 VALUES('1');
INSERT INTO temp09 VALUES('2');
INSERT INTO temp09 VALUES('3');
INSERT INTO temp09 VALUES('4');
INSERT INTO temp09 VALUES('5');
INSERT INTO temp09 VALUES('6');
INSERT INTO temp09 VALUES('7');
INSERT INTO temp09 VALUES('8');
INSERT INTO temp09 VALUES('9');

```

```

CREATE TABLE seq (
 i INTEGER NOT NULL PRIMARY KEY
);

```

```

INSERT INTO seq
SELECT CAST(t1.i || t2.i ||
 t3.i || t4.i AS INTEGER)
FROM temp09 t1, temp09 t2,
 temp09 t3, temp09 t4;

```

```
DROP TABLE temp09;
```

序列表对于日期和时间函数及枚举函数尤其有用。  
代码15-6列出ASCII字符集的95个可打印字符(如果它是在用的字符集)。结果见图15-6。

**代码15-6** 列出与字符集相关的字符。结果见图15-6

```

SELECT
 i AS CharCode,
 CHR(i) AS Ch
FROM seq
WHERE i BETWEEN 32 AND 126;

```

代码15-7对于今天(07-03-2005)加上6个月的时间间隔,结果见图15-7。这个例子可以运行在Microsoft SQL Server上;其他DBMS有类似的递增日期功能。

**代码15-7** 在今天日期上按1个月时间间隔递增到6个月后。结果见图15-7

```

SELECT
 i AS MonthsAhead,
 DATEADD("m", i, CURRENT_TIMESTAMP)
 AS FutureDate
FROM seq

```

| i    |
|------|
| 0    |
| 1    |
| 2    |
| 3    |
| 4    |
| ...  |
| 9996 |
| 9997 |
| 9998 |
| 9999 |

图15-5 运行代码15-5的结果

| CharCode | Ch |
|----------|----|
| 32       |    |
| 33       | !  |
| 34       | "  |
| 35       | #  |
| 36       | \$ |
| 37       | %  |
| 38       | &  |
| 39       | '  |
| 40       | (  |
| 41       | )  |
| 42       | *  |
| 43       | +  |
| 44       | ,  |
| 45       | -  |
| 46       | .  |
| 47       | /  |
| 48       | 0  |
| 49       | 1  |
| 50       | 2  |
| 51       | 3  |
| 52       | 4  |
| ...      |    |

图15-6 运行代码15-6的结果

```
WHERE i BETWEEN 1 AND 6;
```

| MonthsAhead | FutureDate |
|-------------|------------|
| 1           | 2005-04-07 |
| 2           | 2005-05-07 |
| 3           | 2005-06-07 |
| 4           | 2005-07-07 |
| 5           | 2005-08-07 |
| 6           | 2005-09-07 |

图15-7 运行代码15-7的结果

序列表对于数据规范化是非常有用的，它可以从非关系型环境（如电子表格）导入数据。假定有如下不规范的表au\_orders，显示每本图书封面上的作者名字。

|     | title_id | author1 | author2 | author3 |
|-----|----------|---------|---------|---------|
| T01 | A01      | NULL    | NULL    |         |
| T02 | A01      | NULL    | NULL    |         |
| T03 | A05      | NULL    | NULL    |         |
| T04 | A03      | A04     | NULL    |         |
| T05 | A04      | NULL    | NULL    |         |
| T06 | A02      | NULL    | NULL    |         |
| T07 | A02      | A04     | NULL    |         |
| T08 | A06      | NULL    | NULL    |         |
| T09 | A06      | NULL    | NULL    |         |
| T10 | A02      | NULL    | NULL    |         |
| T11 | A06      | A03     | A04     |         |
| T12 | A02      | NULL    | NULL    |         |
| T13 | A01      | NULL    | NULL    |         |

代码15-8交叉联结au\_orders和seq，结果如图15-8。可以删除结果中列au\_id为空的行，让结果集像示例数据库的表title\_authors那样。注意，代码15-8由代码8-18改写而成。

#### 代码15-8 规范化表au\_orders。结果见图15-8

```

SELECT title_id,
 (CASE WHEN i=1 THEN '1'
 WHEN i=2 THEN '2'
 WHEN i=3 THEN '3'
 END) AS au_order,
 (CASE WHEN i=1 THEN author1
 WHEN i=2 THEN author2
 WHEN i=3 THEN author3
 END) AS au_id
FROM au_orders, seq
WHERE i BETWEEN 1 AND 3
ORDER BY title_id, i;

```

```
title_id au_order au_id
```

|     |   |      |
|-----|---|------|
| T01 | 1 | A01  |
| T01 | 2 | NULL |
| T01 | 3 | NULL |
| T02 | 1 | A01  |
| T02 | 2 | NULL |
| T02 | 3 | NULL |
| T03 | 1 | A05  |
| T03 | 2 | NULL |
| T03 | 3 | NULL |
| T04 | 1 | A03  |
| T04 | 2 | A04  |
| T04 | 3 | NULL |
| T05 | 1 | A04  |
| T05 | 2 | NULL |
| T05 | 3 | NULL |
| T06 | 1 | A02  |
| T06 | 2 | NULL |
| T06 | 3 | NULL |
| T07 | 1 | A02  |
| T07 | 2 | A04  |
| T07 | 3 | NULL |
| T08 | 1 | A06  |
| T08 | 2 | NULL |
| T08 | 3 | NULL |
| T09 | 1 | A06  |
| T09 | 2 | NULL |
| T09 | 3 | NULL |
| T10 | 1 | A02  |
| T10 | 2 | NULL |
| T10 | 3 | NULL |
| T11 | 1 | A06  |
| T11 | 2 | A03  |
| T11 | 3 | A04  |
| T12 | 1 | A02  |
| T12 | 2 | NULL |
| T12 | 3 | NULL |
| T13 | 1 | A01  |
| T13 | 2 | NULL |
| T13 | 3 | NULL |

图15-8 运行代码15-8的结果

### 日历表

另一个有用的辅助表是日历表。有一类日历表有主键列，它的行包含了日历日期（过去和将来）和其他表示这个日期属性的列——工作日、假日、国际假日、财务月底、财务年底、儒略历日期、工作日补偿等。另一类日历表存储事件开始和结束的日期（如在event\_id、start\_date和end\_date列）。电子表格的日期计算函数比DBMS丰富，在电子表格中建立日期表然后将它导入数据表更简便。

即便DBMS有大量日期计算函数，在日历表中查找数据也要比在查询中使用这些函数要快。

#### ✓ 提示

- 如果有一个连续的整数列，其中缺少一些数字，可以用连续序列列减去这个列来填充空缺，参见9.3节。
- **DBMS** 为了在Microsoft Access和Microsoft SQL Server中运行代码15-5，将CAST表达式变为：

```
t1.i + t2.i + t3.i + t4.i
```

为了在MySQL中运行代码15-5，将CAST表达式变为：

```
CONCAT(t1.i, t2.i, t3.i, t4.i)
```

为了在Microsoft SQL Server和MySQL中运行代码15-6，将CHR(i)变为CHAR(i)。

为了在Microsoft Access中运行代码15-8，将CASE表达式变为Switch()函数调用（参见5.14节DBMS提示）：

```
(Switch(i=1, '1', i=2, '2', i=3, '3'))
AS au_order,
(Switch(i=1, author1, i=2, author2, i=3,
author3)) AS au_id
```

414

## 15.3 发现等差数列、递增数列和等值数列

连续数列（sequence）是一系列没有间隔的值。递增数列（run）类似于连续序列，但它的值未必是连续的，只是递增（或者说间隔是允许的）。等值数列（region）是没有间断且都相等的数值列。

发现这样的数列需要至少包含两列值的表：一列包含连续的整数主键列，另一列包含相关值。表temps（代码15-9和图15-9）显示了15天的最高温度序列。

**代码15-9** 列出表temps中所有列。结果见图15-9

```
SELECT *
FROM temps;
```

作为面向集合的语言，SQL并不善于发现数列。以下查询运行得不是非常快，如果有很多数据要分析，

| id | hi_temp |
|----|---------|
| 1  | 49      |
| 2  | 46      |
| 3  | 48      |
| 4  | 50      |
| 5  | 50      |
| 6  | 50      |
| 7  | 51      |
| 8  | 52      |
| 9  | 53      |
| 10 | 50      |
| 11 | 50      |
| 12 | 47      |
| 13 | 50      |
| 14 | 51      |
| 15 | 52      |

图15-9 运行代码15-9的结果

15

就应该输出到统计软件，或使用过程型主机语言。

### ✓ 提示

- 这些查询基于David Rozenshtein、Anatoly Abramovich和Eugene Birger所著*Optimizing Transact-SQL: Advanced Programming Techniques*(SQL Forum Press)中的思路。可以使用这些查询的共同框架来创建发现别的数列值的类似查询。

代码15-10发现表temps中所有序列并列出每个序列的开始位置、结束位置和长度。结果见图15-10。这个查询初看很复杂，但只要一点点地看下去是容易理解的。在这个基础上就能理解本节中的其他查询。

**代码15-10** 列出起点、终点和表temps每个序列的长度。结果见图15-10

```

SELECT
 t1.id AS StartSeq,
 t2.id AS EndSeq,
 t2.id - t1.id + 1 AS SeqLen
FROM temps t1, temps t2
WHERE (t1.id < t2.id)
 AND NOT EXISTS(
 SELECT *
 FROM temps t3
 WHERE (t3.hi_temp - t3.id <>
 t1.hi_temp - t1.id
 AND t3.id BETWEEN
 t1.id AND t2.id)
 OR (t3.id = t1.id - 1
 AND t3.hi_temp - t3.id =
 t1.hi_temp - t1.id)
 OR (t3.id = t2.id + 1
 AND t3.hi_temp - t3.id =
 t1.hi_temp - t1.id)
);

```

这个查询的子查询WHERE子句用hi\_temp减去id产生（在内部）如下。

id hi\_temp diff

| 1  | 49 | 48 |
|----|----|----|
| 2  | 46 | 44 |
| 3  | 48 | 45 |
| 4  | 50 | 46 |
| 5  | 50 | 45 |
| 6  | 50 | 44 |
| 7  | 51 | 44 |
| 8  | 52 | 44 |
| 9  | 53 | 44 |
| 10 | 50 | 40 |
| 11 | 50 | 39 |
| 12 | 47 | 35 |
| 13 | 50 | 37 |
| 14 | 51 | 37 |
| 15 | 52 | 37 |

StartSeq EndSeq SeqSize

| StartSeq | EndSeq | SeqSize |
|----------|--------|---------|
| 6        | 9      | 4       |
| 13       | 15     | 3       |

图15-10 运行代码15-10的结果

注意，在diff列中，连续的差是常数序列（50–6=44，51–7=44，诸如此类）。如同15.1节那样，为了寻找邻近的行，外层查询将同一个表的两个实例（t1和t2）交叉联结，参见15.1节。条件：

`WHERE (t1.id < t2.id)`

确保任何t1行代表一个小于相应t2行索引（id）的元素。

416

子查询使用条件：

`t3.hi_temp - t3.id < t1.hi_temp - t1.id`

侦测序列中断。

子查询中的第三个实例`temps`（t3）用于确定在候选序列（t3）中是否所有行与序列的第一行（t1）拥有同样的差。如果是这样，它就是序列数字。如果不是这样，候选组（t1和t2）被否定。

最后两个OR条件决定候选序列的边界是否可以延伸。满足这些条件的行意味着当前候选列可以延伸，如果不满足，那么序列就不再延伸。

### ✓ 提示

□ 为了仅发现大于n行的序列，要给`WHERE`条件加上：

`AND (t2.id - t1.id) >= n - 1`

为了改变代码15-10来发现所有4行或更多行的序列，例如，将

`WHERE (t1.id < t2.id)`

改为：

`WHERE (t1.id < t2.id)  
AND (t2.id - t1.id) >= 3`

结果是：

| StartSeq | EndSeq | SeqSize |
|----------|--------|---------|
| 6        | 9      | 4       |

417

代码15-11寻找表`temps`中所有的递增，并列出每个递增的起点、终点和长度。结果见图15-11。

这个查询的逻辑类似于前面的例子，但只需增加就看作是递增值，不必连续。因为在`id`和`hi_temp`之间没有差值相同的必要，`temps`第四个实例（t4）是需要的。子查询交叉联结t3和t4来在候选递增行中进行检查，它的边界是t1和t2。对于t1和t2（使用`BETWEEN`限定）之间的每一个元素，用t3及其前面的t4比较来判断它们的值是否增加了。

418

**代码15-11 列出起点、终点和`temps`表中每个递增的长度。结果见图15-11**

```
SELECT
 t1.id AS StartRun,
 t2.id AS EndRun,
 t2.id - t1.id + 1 AS RunLen
FROM temps t1, temps t2
WHERE (t1.id < t2.id)
 AND NOT EXISTS(
 SELECT *
 FROM temps t3, temps t4
 WHERE (t3.hi_temp <= t4.hi_temp
```

15

```

 AND t4.id = t3.id - 1
 AND t3.id BETWEEN
 t1.id + 1 AND t2.id)
 OR (t3.id = t1.id - 1
 AND t3.hi_temp <
 t1.hi_temp)
 OR (t3.id = t2.id + 1
 AND t3.hi_temp >
 t2.hi_temp)
);

```

代码15-12在表temps中使用50这个温度发现所有等值序列，并列出每个范围的起点、终点、和长度。结果见图15-12。

**代码15-12 在temps表中列出每个等值序列的起点、终点和长度（使用值50）。结果见图15-12**

```

SELECT
 t1.id AS StartReg,
 t2.id AS EndReg,
 t2.id - t1.id + 1 AS RegLen
FROM temps t1, temps t2
WHERE (t1.id < t2.id)
 AND NOT EXISTS(
 SELECT *
 FROM temps t3
 WHERE (t3.hi_temp <> 50
 AND t3.id BETWEEN
 t1.id AND t2.id)
 OR (t3.id = t1.id - 1
 AND t3.hi_temp = 50)
 OR (t3.id = t2.id + 1
 AND t3.hi_temp = 50)
);

```

#### ✓ 提示

□ 为了列出等值的长度，向外部查询添加ORDER BY子句：

```
ORDER BY t2.id - t1.id DESC
```

□ 为了列出等值中（值50）的单个id，输入：

```

SELECT DISTINCT t1.id
FROM temps t1, temps t2
WHERE t1.hi_temp = 50
 AND t2.hi_temp = 50
 AND ABS(t1.id - t2.id) = 1;

```

所有DBMS都支持的标准函数ABS()返回参数的绝对值。结果是：

|    |
|----|
| id |
| —  |
| 4  |
| 5  |
| 6  |
| 10 |
| 11 |

| StartRun | EndRun | RunLen |
|----------|--------|--------|
| 2        | 4      | 3      |
| 6        | 9      | 4      |
| 12       | 15     | 4      |

图15-11 运行代码15-11的结果

| StartReg | EndReg | RegLen |
|----------|--------|--------|
| 4        | 6      | 3      |
| 10       | 11     | 2      |

图15-12 运行代码15-12的结果

代码15-13是代码15-12的另一种实现方式，用于发现所有长度为2的等值序列。结果见图15-13。注意重叠的子序列被列出。要返回长度 $n$ 的等值序列，改变WHERE子句的第二个条件为：

AND t2.id=t1.id+1

420

#### 代码15-13 列出长度为2的等值序列。结果见图15-13

```
SELECT
 t1.id AS StartReg,
 t2.id AS EndReg,
 t2.id - t1.id + 1 AS RegLen
FROM temps t1, temps t2
WHERE (t1.id < t2.id)
 AND t2.id - t1.id = 1
 AND NOT EXISTS(
 SELECT *
 FROM temps t3
 WHERE (t3.hi_temp <> 50
 AND t3.id BETWEEN
 t1.id AND t2.id)
);
;
```

| StartReg | EndReg | RegLen |
|----------|--------|--------|
| 4        | 5      | 2      |
| 5        | 6      | 2      |
| 10       | 11     | 2      |

图15-13 运行代码15-13的结果

#### 代码15-14 按销量的降序列出雇员。结果见图15-14

```
SELECT emp_id, sales
 FROM empsales
 ORDER BY sales DESC;
```

##### ✓ 提示

- 也可以使用这些查询来限定一条UPDATE或DELETE语句影响的行数（见第10章）。
- 其中的一些查询在某些运行环境中可能是非法的（例如在子查询或视图中），查阅所用DBMS的文档。
- 上面例子中的一部分是基于Troels Arvin的文章“Comparison of Different SQL Implementations”中的想法（<http://troels.arvin.dk/db/rdbms>）。

| emp_id | sales |
|--------|-------|
| E09    | 900   |
| E02    | 800   |
| E10    | 700   |
| E05    | 700   |
| E01    | 600   |
| E04    | 500   |
| E03    | 500   |
| E06    | 500   |
| E08    | 400   |
| E07    | 300   |

图15-14 运行代码15-14的结果

## 15.4 限定返回行的数量

在实际工作中，使用查询返回首尾范围由ORDER BY子句确定的一定数量( $n$ )的行是很常见的。SQL可以没有ORDER BY子句，但如果省略它，查询会返回任意顺序的行集合（因为SQL没有承诺在不使用ORDER BY子句时以某种顺序返回结果）。

本节的例子使用了表empsales（代码15-14和图15-14）。它按照雇员列出销售数字。注意，一些雇员有相同的销售数字。3个最高销量销售员的正确查询从表empsales实际返回4行：雇员E09、E02、E10和E05。并列不会强制查询在相同值（此例中的E10和E05）中选择某个。返回最“如何”的 $n$ 行（不管并列）的查询有时被称作限定查询，对此没有标准的术语。包括连带的查询可能返回超过 $n$ 行，被称作top- $n$ 查询或比例查询。

15

- DBMS** SQL:2003标准引入了ROW\_NUMBER()和RANK()函数用于限定查询和top-n查询。Microsoft SQL Server 2005及之后版本, Oracle和DB2都支持这两个函数。使用SQL: 2003前的标准查询是复杂、不直观的, 且运行缓慢(见本节结尾处对于SQL-92例子的提示)。SQL标准已经滞后于DBMS, DBMS已经提供了非标准的扩展来创建这些类型的查询。有些DBMS也允许返回一定比例的行(不是固定的n)或跨过一定数量的起始行(例如, 返回3~8行而不是1~5)返回分支。本节介绍了各种DBMS的扩展。

421

### 15.4.1 Microsoft Access

代码15-15列出业绩最佳的3个销售员, 包括并列的。结果见图15-15。这个查询是按最高到最低, 要反转这个顺序, 在ORDER BY子句中将DESC改为ASC。

**代码15-15** 列出业绩最佳的3个销售员, 包括并列的。  
结果见图15-15

```
SELECT TOP 3 emp_id, sales
 FROM empsales
 ORDER BY sales DESC;
```

代码15-16列出业绩属于最差40%的销售员, 包括并列的。结果见图15-16。这个查询是按从最低到最高顺序; 要反转这个顺序, 在ORDER BY子句中将ASC变为DESC。

**代码15-16** 列出业绩最差的40%的销售员, 包括并列的。结果见图15-16

```
SELECT TOP 40 PERCENT emp_id, sales
 FROM empsales
 ORDER BY sales ASC;
```

TOP子句总是包含并列。它的语法是:  
TOP n [PERCENT]

#### ✓ 提示

- 下面分支查询返回结果中不包括最前面行的n

行。这个查询是按从最高到最低, 要反转顺序, 将每个ORDER BY子句中的ASC变为DESC, 同时将DESC变为ASC。

```
SELECT *
 FROM C
 SELECT TOP n *
 FROM C
 SELECT TOP n + skip *
 FROM table
 ORDER BY sort_col DESC)
 ORDER BY sort_col ASC)
 ORDER BY sort_col DESC;
```

422

| emp_id | sales |
|--------|-------|
| E09    | 900   |
| E02    | 800   |
| E10    | 700   |
| E05    | 700   |

图15-15 运行代码15-15的结果

| emp_id | sales |
|--------|-------|
| E07    | 300   |
| E08    | 400   |
| E06    | 500   |
| E04    | 500   |
| E03    | 500   |

图15-16 运行代码15-16的结果

### 15.4.2 Microsoft SQL Server

代码15-17列出销售业绩最佳的3个销售员，不包括并列的，结果见图15-17。注意，这个查询的结果和有并列情况存在时是不一致的。返回的结果是E10还是E05，取决于ORDER BY如何对表排序。这个查询顺序是从最高到最低；要反转顺序，将ORDER BY子句中的DESC改为ASC。

**代码15-17** 列出业绩最佳的3个销售员，不包含并列的。结果见图15-17

```
SELECT TOP 3 emp_id, sales
FROM empsales
ORDER BY sales DESC;
```

代码15-18列出业绩最佳的3个销售员，包括并列的，结果见图15-18。这个查询顺序是从最高到最低，要反转顺序，将ORDER BY子句里的DESC改为ASC。

**代码15-18** 列出业绩最佳的3个销售员，包括并列的。结果见图15-18

```
SELECT TOP 3 WITH TIES emp_id, sales
FROM empsales
ORDER BY sales DESC;
```

代码15-19列出了业绩最差的40%销售员，包括并列的，结果见图15-19。这个查询是按照从最低到最高顺序，要反转顺序，将ORDER BY子句中的ASC改为DESC。

**代码15-19** 列出业绩最差的40%销售人员，包括并列的。结果见图15-19

```
SELECT TOP 40 PERCENT WITH TIES
emp_id, sales
FROM empsales
ORDER BY sales ASC;
```

TOP子句的语法是：

TOP *n* [PERCENT] [WITH TIES]

#### ✓ 提示

□ SET ROWCOUNT *n*语句提供了返回*n*行的另一种方法。

□ 为了检索有序行的特定子集，可以使用游标（本书没有介绍）。以下分支查询返回结果中不包括起始跨越行的*n*行。查询顺序是从最高到最低；要反转顺序，将每个ORDER BY子句中的ASC改为DESC，DESC改为ASC。

```
SELECT *
FROM C
SELECT TOP n *
FROM C
SELECT TOP n + skip *
FROM table
```

| emp_id | sales |
|--------|-------|
| E09    | 900   |
| E02    | 800   |
| E05    | 700   |

图15-17 运行代码15-17的结果

| emp_id | sales |
|--------|-------|
| E09    | 900   |
| E02    | 800   |
| E05    | 700   |
| E10    | 700   |

图15-18 运行代码15-18的结果

| emp_id | sales |
|--------|-------|
| E07    | 300   |
| E08    | 400   |
| E06    | 500   |
| E03    | 500   |
| E04    | 500   |

图15-19 运行代码15-19的结果

424

```

 ORDER BY sort_col DESC)
 AS any_name1
 ORDER BY sort_col ASC)
 AS any_name2
 ORDER BY sort_col DESC;

```

### 15.4.3 Oracle

使用内置的ROWNUM伪列来限制数量或返回行的数量。选择的第一行为1，第二行为2，依此类推。使用窗口函数RANK()来包括并列情况。

代码15-20列出业绩最好的3个销售员，不包括并列的，结果见图15-20。注意当并列情况存在时，这个查询的结果是不一致的。再次运行将返回E10还是E05，取决于ORDER BY如何排序表。这个查询顺序是从最高到最低的，要反转顺序，将ORDER BY子句中的DESC改为ASC。

**代码15-20** 列出业绩最好的3个销售员，不包括并列的。结果见图15-20

```

SELECT emp_id, sales
FROM (
 SELECT *
 FROM empsales
 ORDER BY sales DESC)
WHERE ROWNUM <= 3;

```

| emp_id | sales |
|--------|-------|
| E09    | 900   |
| E02    | 800   |
| E05    | 700   |

图15-20 运行代码15-20的结果

代码15-21列出业绩最佳的3个销售员，包括并列的，结果见图15-21。这个查询顺序是从最高到最低的，要反转顺序，将ORDER BY子句中的DESC改为ASC。

**代码15-21** 列出业绩最好的3个销售员，包括并列的。结果见图15-21

```

SELECT emp_id, sales
FROM (
 SELECT
 RANK() OVER
 (ORDER BY sales DESC)
 AS sales_rank,
 emp_id,
 sales
 FROM empsales)
WHERE sales_rank <= 3;

```

| emp_id | sales |
|--------|-------|
| E09    | 900   |
| E02    | 800   |
| E05    | 700   |
| E10    | 700   |

图15-21 运行代码15-21的结果

#### ✓ 提示

- 函数ROW\_NUMBER()提供了为行分配唯一数字的另一种方法。
- 为了检索有序行的特定子集，可以使用游标（本书没有介绍）。以下分支查询返回不包括结果中起始跨越的行的n行。查询顺序是从最高到最低的，要反转顺序，将每个ORDER BY子句中的DESC改为ASC。

```

SELECT *
FROM (
 SELECT
 ROW_NUMBER() OVER
 (ORDER BY sort_col DESC)

```

```

 AS rnum,
 columns
 FROM table)
WHERE rnum > .skip
 AND rnum <= (n + skip);

```

425

#### 15.4.4 IBM DB2

代码15-22列出业绩最好的3个销售员，不包括并列的，结果见图15-22。注意当并列情况存在时，这个查询结果是不一致的。再次运行将返回E10还是E05取决于ORDER BY如何排序表。这个查询顺序是从最高到最低的，要反转顺序，将ORDER BY子句中的DESC改为ASC。

**代码15-22** 列出业绩最好的3个销售人员，不包括并列的。结果见图15-22

```

SELECT emp_id, sales
 FROM empsales
 ORDER BY sales DESC
 FETCH FIRST 3 ROWS ONLY;

```

代码15-23列出业绩最佳的3个销售员，包括并列的，结果见图15-23。这个查询顺序是从最高到最低的，要反转顺序，将ORDER BY子句中的DESC改为ASC。

| emp_id | sales |
|--------|-------|
| E09    | 900   |
| E02    | 800   |
| E05    | 700   |

图15-22 运行代码15-22的结果

**代码15-23** 列出业绩最好的3个销售员，包括并列的。结果见图15-23

```

SELECT emp_id, sales
 FROM (
 SELECT
 RANK() OVER
 (ORDER BY sales DESC)
 AS sales_rank,
 emp_id,
 sales
 FROM empsales
 AS any_name
 WHERE sales_rank <= 3;

```

| emp_id | sales |
|--------|-------|
| E09    | 900   |
| E02    | 800   |
| E05    | 700   |
| E10    | 700   |

图15-23 运行代码15-23的结果

FETCH子句的语法是：

FETCH FIRST *n* ROW[S] ONLY

#### ✓ 提示

- 为了检索有序行的特定子集，可以使用游标（本书没有介绍）。以下分支查询返回结果中不包括起始跨越行的*n*行。这个查询的顺序是从最高到最低的，要反转顺序，将每个ORDER BY子句中的DESC改为ASC。

```

SELECT *
 FROM (
 SELECT
 ROW_NUMBER() OVER
 (ORDER BY sort_col DESC)

```

15

426

```

 AS rnum,
 columns
 FROM table)
 AS any_name
 WHERE rnum > skip
 AND rnum <= n + skip;

```

### 15.4.5 MySQL

代码15-24列出业绩最好的3个销售员，不包括并列的，结果见图15-24。注意当并列情况存在时，查询的结果是不一致的。再次运行将返回E10还是E05取决于ORDER BY如何排序表。这个查询结果按照从最高到最低排列，要反转顺序，将ORDER BY子句中的DESC改为ASC。

**代码15-24** 列出业绩最好的3个销售员，不包括并列的。结果见图15-24

```

SELECT emp_id, sales
 FROM empsales
 ORDER BY sales DESC
 LIMIT 3;

```

| emp_id | sales |
|--------|-------|
| E09    | 900   |
| E02    | 800   |
| E10    | 700   |

图15-24 运行代码15-24的结果

代码15-25列出业绩最好的3个销售员，包括并列的。OFFSET值是 $n-1=2$ 。COALESCE()的第二个自变量使得表的行数小于n时查询可以运行，参见5.15节。结果见图15-25。这个查询结果按照从最高到最低排列，要反转顺序，将比较中的 $>$ 改为 $\leq$ ，将第二个子查询中的MIN()改为MAX()，并且将每个ORDER BY子句中的DESC改为ASC。

**代码15-25** 列出业绩最好的3个销售员，包括并列的。结果见图15-25

```

SELECT emp_id, sales
 FROM empsales
 WHERE sales >= COALESCE(
 (SELECT sales
 FROM empsales
 ORDER BY sales DESC
 LIMIT 1 OFFSET 2),
 (SELECT MIN(sales)
 FROM empsales))
 ORDER BY sales DESC;

```

| emp_id | sales |
|--------|-------|
| E09    | 900   |
| E02    | 800   |
| E05    | 700   |
| E10    | 700   |

图15-25 运行代码15-25的结果

代码15-26列出业绩最佳的3个销售员，跨过最初的4行，结果见图15-26。注意当存在并列情况时，查询结果是不一致的。这个查询结果顺序是按照从最高到最低的，要反转顺序，将ORDER BY子句中的DESC改为ASC。

**代码15-26** 列出业绩最佳的3个销售员，越过最初的4行。结果见图15-26

```

SELECT emp_id, sales
 FROM empsales

```

| emp_id | sales |
|--------|-------|
| E01    | 600   |
| E04    | 500   |
| E03    | 500   |

图15-26 运行代码15-26的结果

```
ORDER BY sales DESC
LIMIT 3 OFFSET 4;
```

LIMIT子句的语法是：

```
LIMIT n [OFFSET skip]
```

或

```
LIMIT [skip,] n
```

初始行的偏移值是0（不是1）。

#### 15.4.6 PostgreSQL

代码15-27列出业绩最好的3个销售员，不包括并列的，结果见图15-27。注意当并列情况存在时，查询的结果不一致。再次运行将返回E10还是E05，取决于ORDER BY如何排序表。这个查询结果按照从最高到最低排列，要反转顺序，将ORDER BY子句中的DESC改为ASC。

**代码15-27** 列出业绩最佳的3个销售员，不包括并列的。结果见图15-27

```
SELECT emp_id, sales
FROM empsales
ORDER BY sales DESC
LIMIT 3;
```

代码15-28列出业绩最好的3个销售员，包括并列的。OFFSET值是*n*-1=2，结果见图15-28。这个查询结果是按从最高到最低的，要反转顺序，将比较中的 $>=$ 改为 $<=$ ，并且将每个ORDER BY子句中的DESC改为ASC。

| emp_id | sales |
|--------|-------|
| E09    | 900   |
| E02    | 800   |
| E05    | 700   |

图15-27 运行代码15-27的结果

**代码15-28** 列出业绩最佳的3个销售员，包括并列的。结果见图15-28

```
SELECT emp_id, sales
FROM empsales
WHERE (
 sales >= (
 SELECT sales
 FROM empsales
 ORDER BY sales DESC
 LIMIT 1 OFFSET 2)
) IS NOT FALSE
ORDER BY sales DESC;
```

| emp_id | sales |
|--------|-------|
| E09    | 900   |
| E02    | 800   |
| E10    | 700   |
| E05    | 700   |

图15-28 运行代码15-28的结果

代码15-29列出业绩最佳的3个销售员，跨过最初的4行，结果见图15-29。注意当存在并列情况时，查询结果不一致。这个查询结果按照从最高到最低排列，要反转顺序，将ORDER BY子句中的DESC改为ASC。

**代码15-29** 列出业绩最佳的3个销售员，跨过最初的4行。结果见图15-29

```
SELECT emp_id, sales
FROM empsales
```

| emp_id | sales |
|--------|-------|
| E01    | 600   |
| E06    | 500   |
| E03    | 500   |

图15-29 运行代码15-29的结果

```
ORDER BY sales DESC
LIMIT 3 OFFSET 4;
```

LIMIT子句的语法是：

```
LIMIT n [OFFSET skip]
```

初始行的偏移值是0（不是1）。

428

### ✓ 提示

- 当使用的查询显示给最终用户的结果不一致时，最好使用一个不包括并列情况的ORDER BY列，这样用户从多个查询看到的结果一致。例如，在本节这个查询的ORDER BY子句的sales列后加入emp\_id（在最外面），确保有相同销售量的雇员总是按照相同的方式排列。
- Fabian Pascal所著的*Practical Issues in Database Management* (Addison-Wesley)讨论了比例查询。他的SQL-92方案（在实际使用中太慢）列出业绩最佳的3个销售员，包括并列的，如下所示。

```
SELECT emp_id, sales
 FROM empsales e1
 WHERE (
 SELECT COUNT(*)
 FROM empsales e2
 WHERE e2.sales > e1.sales
) < 3;
```

429

这个查询结果按照从最高到最低排列，要反转顺序，将最内层WHERE子句的>改为<。

## 15.5 分配排名

排名（分配数字1, 2, 3, …给排序出来的值）和top-n查询有关，同时也存在怎样处理并列情况的问题。以下查询对上一节empsales表里的销售量的计算结果排名。

代码15-30a到代码15-30e按销售量对雇员排名。前面两个查询显示了最为常用的排名方法。其他查询是其表达方式的变形。图15-30显示了每一种方法的结果，a到e简单地结合在一起为的是容易比较。这些查询结果按照从最高到最低排列；要反转顺序，将WHERE比较中的>（或 $\geq$ ）改为<（或 $\leq$ ）。

### 代码15-30a 按销售量对雇员排名（方式a）， 结果见图15-30

```
SELECT e1.emp_id, e1.sales,
 (SELECT COUNT(sales)
 FROM empsales e2
 WHERE e2.sales >= e1.sales)
 AS ranking
 FROM empsales e1;
```

### 代码15-30c 按销售量对雇员排名（方法c）， 结果见图15-30

```
SELECT e1.emp_id, e1.sales,
 (SELECT COUNT(sales)
 FROM empsales e2
 WHERE e2.sales > e1.sales)
 AS ranking
 FROM empsales e1;
```

### 代码15-30b 按销售量对雇员排名（方法b）， 结果见图15-30

```
SELECT e1.emp_id, e1.sales,
 (SELECT COUNT(sales)
 FROM empsales e2
 WHERE e2.sales > e1.sales) + 1
 AS ranking
 FROM empsales e1;
```

### 代码15-30d 按销售量对雇员排名（方法d）， 结果见图15-30

```
SELECT e1.emp_id, e1.sales,
 (SELECT COUNT(DISTINCT sales)
 FROM empsales e2
 WHERE e2.sales >= e1.sales)
 AS ranking
 FROM empsales e1;
```

**代码15-30e 按销售量对雇员排名 (方法e), 结果见图15-30**

```
SELECT e1.emp_id, e1.sales,
 (SELECT COUNT(DISTINCT sales)
 FROM empsales e2
 WHERE e2.sales > e1.sales)
 AS ranking
 FROM empsales e1;
```

- **DBMS** 很慢。要对大量项目排名，如果可能应该使用内置的排名函数或OLAP组件。SQL:2003引入了函数RANK()和DENSE\_RANK()，Microsoft SQL Server 2005和之后的版本，Oracle和DB2都支持。对于Microsoft SQL Server 2000，可以借助分析服务(OLAP)函数RANK()。另一个方法是使用DBMS SQL扩展来高效处理排名。例如，下面的MySQL脚本和代码15-30b是等价的。

```
SET @rownum = 0;
SET @rank = 0;
SET @prev_val = NULL;
SELECT
 @rownum := @rownum + 1 AS row,
 @rank := IF(@prev_val <> sales,
 @rownum, @rank) AS rank,
 @prev_val := sales AS sales
 FROM empsales
 ORDER BY sales DESC;
```

✓ 提示

- 可以在查询外部的SELECT中添加ORDER BY ranking ASC子句来对结果进行排名。
- **DBMS** Microsoft Access不支持COUNT(DISTINCT)，不能运行代码15-30d和代码15-30e。为了解决这个问题，参见6.8节。[431]

## 15.6 计算修整均值

修整均值是一种粗略的顺序统计，它将最大值和最小值忽略掉，然后求平均值。这样做是为了避免受到极值的影响。

代码15-31通过忽略最高的3个销售数字和最低的3个销售数字，计算示例数据库中图书销售量的修整平均值，结果见图15-31。这12个排序的销售值是566, 4 095, 5 000, 9 566, 10 467, 11 320, 13 001, 25 667, 94 123, 100 001, 201 440和1 500 200。查询忽略566, 4 095, 5 000, 100 001, 201 440和1 500 200，使用剩下的6个中间值按通常的方法计算平均值，空值被忽略。重复值要么都忽略要么都保留。(例如，如果销售量是相同的，不论k是多少，一个值也不忽略。)

**代码15-31 当k=3时，计算修整均值，结果见图15-31**

```
SELECT AVG(sales) AS TrimmedMean
```

| emp_id | sales | a  | b  | c | d | e |
|--------|-------|----|----|---|---|---|
| E09    | 900   | 1  | 1  | 0 | 1 | 0 |
| E02    | 800   | 2  | 2  | 1 | 2 | 1 |
| E10    | 700   | 4  | 3  | 2 | 3 | 2 |
| E05    | 700   | 4  | 3  | 2 | 3 | 2 |
| E01    | 600   | 5  | 5  | 4 | 4 | 3 |
| E04    | 500   | 8  | 6  | 5 | 5 | 4 |
| E03    | 500   | 8  | 6  | 5 | 5 | 4 |
| E06    | 500   | 8  | 6  | 5 | 5 | 4 |
| E08    | 400   | 9  | 9  | 8 | 6 | 5 |
| E07    | 300   | 10 | 10 | 9 | 7 | 6 |

图15-30 代码15-30a到代码15-30e的共同结果

```

FROM titles t1
WHERE
 (SELECT COUNT(*)
 FROM titles t2
 WHERE t2.sales <= t1.sales) > 3
AND
 (SELECT COUNT(*)
 FROM titles t3
 WHERE t3.sales >= t1.sales) > 3;

```

|             |
|-------------|
| TrimmedMean |
| -----       |
| 27357.3333  |

图15-31 运行代码15-31的结果

代码15-32和代码15-40类似，但它排除固定比率而不是固定数量的极值。比如，排除0.25（25%），忽略最高和最低销售量的四分之一，然后对剩下的值求平均值，结果见图15-32。

**代码15-32 忽略最低和最高值的25%，计算修整均值，结果见图15-32**

```

SELECT AVG(sales) AS TrimmedMean
 FROM titles t1
 WHERE
 (SELECT COUNT(*)
 FROM titles t2
 WHERE t2.sales <= t1.sales) >=
 (SELECT 0.25 * COUNT(*)
 FROM titles)
AND
 (SELECT COUNT(*)
 FROM titles t3
 WHERE t3.sales >= t1.sales) >=
 (SELECT 0.25 * COUNT(*)
 FROM titles);

```

|             |
|-------------|
| TrimmedMean |
| -----       |
| 27357.3333  |

图15-32 运行代码15-32的结果

### ✓ 提示

- **DBMS** 因为列sales被定义为整数，Microsoft SQL Server和DB2返回一个整数的修整均值。  
要得到一个浮点数值，将AVG(sales)变为AVG(CAST(sales AS FLOAT))。要了解更多相关知识，参见5.13节。

432

## 15.7 随机选取行

有些数据库很大，查询起来很复杂，检索所有与查询相关的数据往往是不现实的（也不必要）。比如，若想发现总体趋势或模式，那么带有一些误差的近似回答也可以满足要求。加快这类查询的方法之一是选择随机样本行。高效的样本可以极大地改善运行效率，同时得到相对准确的结果。

SQL标准的TABLESAMPLE子句返回行的随机子集。DB2和SQL Server 2005及其之后版本支持TABLESAMPLE，Oracle也有类似功能。对于其他DBMS，可以使用能均匀返回0~1间随机数的（非标准的）函数（表15-1）。

表15-1 随机功能

| DBMS                 | 子句或函数         |
|----------------------|---------------|
| Access               | RND()函数       |
| SQL Server 2000      | RAND()函数      |
| SQL Server 2005/2008 | TABLESAMPLE子句 |

433

(续)

| DBMS       | 子句或函数                 |
|------------|-----------------------|
| Oracle     | SAMPLE子句或DBMS_RANDOM包 |
| DB2        | TABLESAMPLE子句         |
| MySQL      | RAND()函数              |
| PostgreSQL | RANDOM()函数            |

代码15-33a从示例数据库表titles中随机选取25%（0.25）的行。

根据情况，可将RAND()变为表15-1中相应DBMS的函数。对于Oracle，使用代码15-33b。对于SQL Server 2005及其之后版本和DB2，使用代码15-33c。

图15-33显示一个随机选择的可能结果。每次运行查询，返回的行和行的数量都不一样。如果需要精确数量的随机行，可以提高采样比例，同时使用15.4节介绍的方法之一。

**代码15-33a 在表titles中随机选择25%的行。  
图15-33是一个可能的结果**

```
SELECT title_id, type, sales
 FROM titles
 WHERE RAND() < 0.25;
```

**代码15-33c 在表titles中随机选择25%的行  
(仅限SQL Server 2005及其之后版本和DB2)。图15-33是一个可能的结果**

```
SELECT title_id, type, sales
 FROM titles
 TABLESAMPLE SYSTEM (25);
```

**代码15-33b 在表titles(仅限Oracle)中随机选择25%的行。图15-33是一个可能的结果**

```
SELECT title_id, type, sales
 FROM titles
 SAMPLE (25);
```

| title_id | type       | sales |
|----------|------------|-------|
| T03      | computer   | 25667 |
| T04      | psychology | 13001 |
| T11      | psychology | 94123 |

图15-33 代码15-33a至代码15-33c的一个可能结果

### ✓ 提示

- 随机发生器采用可选的种子(seed)变量或设定一个随机数序列的起始值。同一种子产生同样的结果(测试很容易)。默认情况下，DBMS每次基于系统时间设置种子来产生不同的序列。
- 因为随机数函数对于每一选择行返回同一“随机”数，代码15-33a无法在Microsoft DBMS Access或Microsoft SQL Server 2000上运行。在Access中，可以使用Visual Basic或C#方法来选择随机行。对于SQL Server 2000，请自行查阅www.sqlteam.com上的文章“Returning Rows in Random Order”。

在Microsoft SQL Server中使用NEWID()函数随机选择n行：

```
SELECT TOP n title_id, type, sales
 FROM titles
 ORDER BY NEWID();
```

在Oracle中使用DBMS\_RANDOM包的函数VALUE()来随机选择n行：

```
SELECT * FROM
 (SELECT title_id, type, sales
```

```
FROM titles
ORDER BY DBMS_RANDOM.VALUE()
WHERE ROWNUM <= n;
```

### 选择每个第n行

也可以不随机选取行，而是通过使用一个模表达式选取每个第n行。

- $m \bmod n$  (Microsoft Access)
- $m \% n$  (Microsoft SQL Server)
- $\text{MOD}(m, n)$  (其他DBMSs)

以上表达式都返回 $m$ 除以 $n$ 的余数。例如，因为 $20$ 等于 $(3 \times 6) + 2$ ， $\text{MOD}(20, 6)$ 是 $2$ 。如果 $a$ 是一个偶数， $\text{MOD}(a, 2)$ 是 $0$ 。

条件 $\text{MOD}(\text{rownumber}, n) = 0$ 选取每个第n行，其中 $\text{rownumber}$ 是连续整数或行标识符。以下Oracle查询选取表的每个第3行，例如：

```
SELECT *
 FROM table
 WHERE (ROWID,0) IN
 (SELECT ROWID, MOD(ROWNUM,3)
 FROM table);
```

注意， $\text{rownumber}$ 要求遵循关系型数据库表中并不显式存在的行序。

## 15.8 处理重复值

通常情况下，使用SQL的PRIMARY KEY或UNIQUE约束（见第11章），可以在实际应用的表中防止出现重复行。当偶然输入同样的数据两次或从非关系环境（如电子表格或会计软件）中输入数据时，冗余信息是很泛滥的，因此需要了解如何处理出现的重复值。本节介绍如何查找、计数和删除重复值。

假定在插入数据到实际使用表之前，将行输入到过渡表来发现和删除所有重复值（代码15-34和图15-34）。要识别和选择重复的值，主要应借助列id。如果输入的行还不存在标识列，可以自己添加，见3.12节和15.2节。即便对于存在时间短的工作表，加入一个标识列也是好办法，它能使删除重复行变得简单。输入的数据也可能包含其他列，不管行中其他列的值，但我们决定只使用书名、图书类型和价格来判断行是否重复。在识别或删除重复值前，必须明确定义两行“重复”的含义。

代码15-34 列出输入行，结果见图15-34

```
SELECT id, title_name, type, price
 FROM dups;
```

代码15-35通过计算列title\_name、type和price唯一结合出现的次数，来列出重复行，结果见图15-35。如果这个查询返回空结果，表不包含重复。要列出非重复的行，将 $\text{COUNT(*)} > 1$ 改为 $\text{COUNT(*)} = 1$ 。

| id | title_name   | type      | price |
|----|--------------|-----------|-------|
| 1  | Book Title 5 | children  | 15.00 |
| 2  | Book Title 3 | biography | 7.00  |
| 3  | Book Title 1 | history   | 10.00 |
| 4  | Book Title 2 | children  | 20.00 |
| 5  | Book Title 4 | history   | 15.00 |
| 6  | Book Title 1 | history   | 10.00 |
| 7  | Book Title 3 | biography | 7.00  |
| 8  | Book Title 1 | history   | 10.00 |

图15-34 运行代码15-34的结果

**代码15-35 列出重复值，结果见图15-35**

```
SELECT title_name, type, price
 FROM dups
 GROUP BY title_name, type, price
 HAVING COUNT(*) > 1;
```

| title_name   | type      | price |
|--------------|-----------|-------|
| Book Title 1 | history   | 10.00 |
| Book Title 3 | biography | 7.00  |

图15-35 运行代码15-35的结果

代码15-36使用类似的技术来列出每一行和它的重复数，结果见图15-36。只列出重复的行，将COUNT(\*) >= 1改为COUNT(\*) > 1。

**代码15-36 列出每一行和它的重复数，结果见图15-36**

```
SELECT title_name, type, price,
 COUNT(*) AS NumDups
 FROM dups
 GROUP BY title_name, type, price
 HAVING COUNT(*) >= 1
 ORDER BY COUNT(*) DESC;
```

代码15-37从表dups的合适位置删除重复行。这个语句使用列id来准确保留每次重复出现中的一个(拥有最大ID的行)。图15-37显示在运行这个语句之后的表dups，参见10.4节。

**代码15-37 删除适当位置的冗余重复行，结果见图15-37**

```
DELETE FROM dups
 WHERE id < (
 SELECT MAX(d.id)
 FROM dups d
 WHERE dups.title_name = d.title_name
 AND dups.type = d.type
 AND dups.price = d.price);
```

| title_name   | type      | price | NumDups |
|--------------|-----------|-------|---------|
| Book Title 1 | history   | 10.00 | 3       |
| Book Title 3 | biography | 7.00  | 2       |
| Book Title 4 | history   | 15.00 | 1       |
| Book Title 2 | children  | 20.00 | 1       |
| Book Title 5 | children  | 15.00 | 1       |

图15-36 运行代码15-36的结果

436

| id | title_name   | type      | price |
|----|--------------|-----------|-------|
| 1  | Book Title 5 | children  | 15.00 |
| 4  | Book Title 2 | children  | 20.00 |
| 5  | Book Title 4 | history   | 15.00 |
| 7  | Book Title 3 | biography | 7.00  |
| 8  | Book Title 1 | history   | 10.00 |

图15-37 运行代码15-37的结果

**杂乱数据**

数据越杂乱，删除重复的工作就越困难。购买来的邮寄名录经常会出现类似以下的条目。

| name        | address1           |
|-------------|--------------------|
| John Smith  | 123 Main St        |
| John Smith  | 123 Main St, Apt 1 |
| Jack Smiht  | 121 Main Rd        |
| John Symthe | 123 Main St.       |
| Jon Smith   | 123 Mian Street    |

尽管DBMS提供非标准的工具（如Soundex (phonetic)函数）来删掉拼写变形，但创建一个能够运行于成千上万行的自动删除程序，仍然是一项艰巨的任务。

15

**✓ 提示**

- 如果以行的所有列定义重复（不只是以列的子集定义重复），可以删除id列，并使用SELECT DISTINCT \* FROM table删除重复行，参见4.3节。
- **DBMS** 如果所使用的DBMS提供了内置的唯一行标识符，那么不借助列id也同样可以删除重值。例如，在Oracle中运行代码15-37，可以使用ROWID伪列取代id，将外部WHERE子句改为：

```
WHERE ROWID < (SELECT
 MAX(d.ROWID)...
```

要在MySQL中运行代码15-45，将ORDER BY COUNT(\*) DESC改写为ORDER BY NumDups DESC。因为437 MySQL对于子查询的FROM子句和DELETE目标不允许使用同一个表，所以无法使用代码15-37来执行删除。

## 15.9 创建电话列表

可以使用函数COALESCE()和左外联结，从规范的电话号码表创建一个方便的电话号码列表。假定示例数据库有一个存储作者的工作和家庭电话号码、称作telephones的附加表。

| au_id | tel_type | tel_no       |
|-------|----------|--------------|
| A01   | H        | 111-111-1111 |
| A01   | W        | 222-222-2222 |
| A02   | W        | 333-333-3333 |
| A04   | H        | 444-444-4444 |
| A04   | W        | 555-555-5555 |
| A05   | H        | 666-666-6666 |

表的组合主键是(au\_id, tel\_type)，这里tel\_type指的是tel\_no是工作(W)还是家庭(H)电话号码。代码15-38列出作者名字和电话号码。如果作者只有一个号码，列出号码。如果作者既有工作又有家庭电话号码，只列出工作电话号码。没有号码的作者不列出，结果见图15-38。

第一个左联结找出工作号码，第二个找出家庭号码。WHERE子句过滤掉没有电话号码的作者（可以加上移动电话和其他电话来扩展这个查询）。

**✓ 提示**

- 要了解更多有关COALESCE()的知识，参见5.15节的内容。要了解更多有关左外联结的知识，参见7.8节。
- **DBMS** 因为Access对联结表达式有限制，Microsoft Access无法运行代码15-38。

**代码15-38** 列出作者的名字和电话号码，工作电话优先于家庭电话，结果见图15-38

```
SELECT
 a.au_id AS "ID",
 a.au_fname AS "FirstName",
 a.au_lname AS "LastName",
 COALESCE(twork.tel_no, thome.tel_no)
 AS "TelNo",
 COALESCE(twork.tel_type, thome.tel_type)
 AS "TelType"
 FROM authors a
```

```

LEFT OUTER JOIN telephones twork
ON a.au_id = twork.au_id
AND twork.tel_type = 'W'
LEFT OUTER JOIN telephones thome
ON a.au_id = thome.au_id
AND thome.tel_type = 'H'
WHERE COALESCE(twork.tel_no, thome.tel_no)
IS NOT NULL
ORDER BY a.au_fname ASC, a.au_lname ASC;

```

| ID  | FirstName | LastName  | TelNo        | TelType |
|-----|-----------|-----------|--------------|---------|
| A05 | Christian | Kells     | 666-666-6666 | H       |
| A04 | Klee      | Hull      | 555-555-5555 | W       |
| A01 | Sarah     | Buchman   | 222-222-2222 | W       |
| A02 | Wendy     | Heydemark | 333-333-3333 | W       |

图15-38 运行代码15-38的结果

## 15.10 检索元数据

元数据 (metadata) 是关于数据的数据。在DBMS中元数据包括模式、数据库、用户、表、列等方面的信息。在5.12节和10.1节中我们已经见过元数据了。面对一个新数据库，首先要做就是检查它的元数据：数据库里有什么？它有多大？表是如何组织的？

像其他数据一样，元数据也存储在表中，因此就可以通过SELECT语句访问。使用命令行和图形工具访问元数据往往更方便。以下列表显示特定DBMS查看元数据的例子。DBMS自动维护元数据——可以查看但不要修改。

### ✓ 提示

- **DBMS** SQL标准称一个元数据集为一个目录 (catalog)，并指明它可以通过模式INFORMATION\_SCHEMA访问。不是所有的DBMS都用同样的术语和实现这个模式。例如，在Microsoft SQL Server里和目录等价的词是数据库，和模式等价的词是所有者。在Oracle里，这个存储元数据的资料库被称作数据字典。439

### 15.10.1 Microsoft Access

Access的元数据可以通过每一个数据库对象的设计视图形象地显示，或通过Visual Basic for Applications (VBA) 或C#语言以编程方式查得。Access为每个数据库创建和维护隐藏的数据系统表。

#### ⇒ 显示系统表

在Access 2003或之前版本中，选择Tools→Options命令，在打开的对话框中选择View选项卡，在其中查看系统对象。在Access 2007及之后版本中，单击Microsoft Office按钮，选择Access选项，在左面板中选择Current Database，滚动到Navigation，单击Navigation Options后，查看系统对象。

系统表名以MSys开始并和其他数据库表混在一起。可以像打开和查询普通表一样打开和查询它们。最有趣的系统表是MSysObjects，它对数据库的所有对象分类。代码15-39列出当前数据库中的所有表。注意系统表并非可见才可在查询中使用。

**代码15-39** 列出当前Access数据库中的表。要列出查询，可将Type = 1改为Type = 5

```

SELECT Name
FROM MSysObjects
WHERE Type = 1;

```

### 15.10.2 Microsoft SQL Server

SQL Server元数据可以通过模式INFORMATION\_SCHEMA和系统存储过程访问（代码15-40）。

440

15

**代码15-40 Microsoft SQL Server的元数据语句和命令**

```
-- List the databases.
exec sp_helpdb;

-- List the schemas.
SELECT schema_name
 FROM information_schema.schemata;

-- List the tables (Method 1).
SELECT *
 FROM information_schema.tables
 WHERE table_type = 'BASE TABLE'
 AND table_schema = 'schema_name';

-- List the tables (Method 2).
exec sp_tables;

-- Describe a table (Method 1).
SELECT *
 FROM information_schema.columns
 WHERE table_catalog = 'db_name'
 AND table_schema = 'schema_name'
 AND table_name = 'table_name';

-- Describe a table (Method 2).
exec sp_help table_name;
```

**15.10.3 Oracle**

Oracle的元数据可以通过数据字典视图或通过sqlplus（代码15-41）访问。要列出数据字典视图，在sqlplus中运行以下查询。

```
SELECT table_name, comments
 FROM dictionary
 ORDER BY table_name;
```

**代码15-41 Oracle的元数据语句和命令**

```
-- List the schemas (users).
SELECT *
 FROM all_users;

-- List the tables.
SELECT table_name
 FROM all_tables
 WHERE owner = 'user_name';

-- Describe a table (Method 1).
SELECT *
 FROM all_tab_columns
 WHERE owner = 'user_name'
 AND table_name = 'table_name';

-- Describe a table (Method 2, in sqlplus).
DESCRIBE table_name;
```

要在Unix或Linux中列出Oracle数据库实例，查看位于目录/etc或/var/opt/oracle中的文件oratab。在Windows的命令行提示符中，运行这个命令：

```
net start | find /i "OracleService"
```

441

或选择Start→Run（Windows标志键+R键），输入services.msc，按回车键，然后查看服务列表条目中以*OracleService*开头的条目。

#### 15.10.4 IBM DB2

DB2元数据可以通过系统目录SYSCAT和db2访问（代码15-42）。

442

##### 代码15-42 IBM DB2的元数据语句和命令

```
-- List the databases (in db2).
LIST DATABASE DIRECTORY;

-- List the schemas.
SELECT schemaname
 FROM syscat.schemata;

-- List the tables (Method 1).
SELECT tablename
 FROM syscat.tables
 WHERE tabschema = 'schema_name';

-- List the tables (Method 2, in db2).
LIST TABLES;

-- List the tables (Method 3, in db2).
LIST TABLES FOR SCHEMA schema_name;

-- Describe a table (Method 1).
SELECT *
 FROM syscat.columns
 WHERE tablename = 'table_name'
 AND tabschema = 'schema_name';

-- Describe a table (Method 2, in db2).
DESCRIBE TABLE table_name SHOW DETAIL;
```

#### 15.10.5 MySQL

MySQL的元数据可以通过模式INFORMATION\_SCHEMA和通过mysql访问（代码15-43）。

443

##### 代码15-43 MySQL的元数据语句与命令

```
-- List the databases (Method 1).
SELECT schema_name
 FROM information_schema.schemata;

-- List the databases (Method 2, in mysql).
SHOW DATABASES;

-- List the tables (Method 1).
```

15

```

SELECT table_name
 FROM information_schema.tables
 WHERE table_schema = 'db_name';

-- List the tables (Method 2, in mysql).
SHOW TABLES;

-- Describe a table (Method 1).
SELECT *
 FROM information_schema.columns
 WHERE table_schema = 'db_name'
 AND table_name = 'table_name';

-- Describe a table (Method 2, in mysql).
DESCRIBE table_name;

```

### 15.10.6 PostgreSQL

**444** PostgreSQL的元数据通过模式INFORMATION\_SCHEMA或通过psql访问（代码15-44）。

#### 代码15-44 PostgreSQL的元数据语句和命令

```

-- List the databases (Method 1).
psql --list

-- List the databases (Method 2, in psql).
\l

-- List the schemas.
SELECT schema_name
 FROM information_schema.schemata;

-- List the tables (Method 1).
SELECT table_name
 FROM information_schema.tables
 WHERE table_schema = 'schema_name';

-- List the tables (Method 2, in psql).
\dt

-- Describe a table (Method 1).
SELECT *
 FROM information_schema.columns
 WHERE table_schema = 'schema_name'
 AND table_name = 'table_name';

-- Describe a table (Method 2, in psql).
\d table_name;

```

### 15.11 处理日期

正如5.10节和5.11节中介绍的，DBMS提供它们自己扩展的（非标准的）函数来操作日期和时间。

**445** 本节介绍如何使用内置函数来处理简单的日期计算。每个代码中的查询如下。

- 提取部分（小时、天、月等）当前（系统）日期和时间并作为数字返回。

- 从一个日期加或减年、月或日。
- 计算同列不同行的两个日期之间相差的天数。结果是正数、零还是负数取决于第一个日期晚于、等于还是早于第二个日期。
- 计算同一列上最早日期和最晚日期间隔的月份数。

### 15.11.1 Microsoft Access

函数`datepart()`取得日期和时间的特定部分。`now()`返回当前（系统）日期和时间。`dateadd()`为日期加上特定时间间隔。`datediff()`返回两个日期之间明确的时间间隔（代码15-45）。

#### 代码15-45 在Microsoft Access中处理日期

```
-- Extract parts of the current datetime.
SELECT
 datepart("s", now()) AS sec_pt,
 datepart("n", now()) AS min_pt,
 datepart("h", now()) AS hr_pt,
 datepart("d", now()) AS day_pt,
 datepart("m", now()) AS mon_pt,
 datepart("yyyy", now()) AS yr_pt;

-- Add or subtract days, months, and years.
SELECT
 dateadd("d", 2, pubdate) AS p2d,
 dateadd("d", -2, pubdate) AS m2d,
 dateadd("m", 2, pubdate) AS p2m,
 dateadd("m", -2, pubdate) AS m2m,
 dateadd("yyyy", 2, pubdate) AS p2y,
 dateadd("yyyy", -2, pubdate) AS m2y
FROM titles
WHERE title_id = 'T05';

-- Count the days between two dates.
SELECT datediff("d", date1, date2) AS days
FROM
 (SELECT pubdate as date1
 FROM titles
 WHERE title_id = 'T05') t1,
 (SELECT pubdate as date2
 FROM titles
 WHERE title_id = 'T06') t2;

-- Count the months between two dates.
SELECT datediff("m", date1, date2) AS months
FROM
 (SELECT
 MIN(pubdate) AS date1,
 MAX(pubdate) AS date2
 FROM titles) t1;
```

#### ✓ 提示

- `datepart()`的一个替代是抽取函数`second()`、`day()`、`month()`等。

### 15.11.2 Microsoft SQL Server

函数`datepart()`提取日期和时间的特定部分。`getdate()`返回当前（系统）日期和时间。`dateadd()`为日期加上明确的时间间隔。`datediff()`返回两个日期之间明确的时间间隔（代码15-46）。

#### 代码15-46 在Microsoft SQL Server上处理日期

```
-- Extract parts of the current datetime.
SELECT
 datepart("s", getdate()) AS sec_pt,
 datepart("n", getdate()) AS min_pt,
 datepart("hh", getdate()) AS hr_pt,
 datepart("d", getdate()) AS day_pt,
 datepart("m", getdate()) AS mon_pt,
 datepart("yyyy", getdate()) AS yr_pt;

-- Add or subtract days, months, and years.
SELECT
 dateadd("d", 2, pubdate) AS p2d,
 dateadd("d", -2, pubdate) AS m2d,
 dateadd("m", 2, pubdate) AS p2m,
 dateadd("m", -2, pubdate) AS m2m,
 dateadd("yyyy", 2, pubdate) AS p2y,
 dateadd("yyyy", -2, pubdate) AS m2y
FROM titles
WHERE title_id = 'T05';

-- Count the days between two dates.
SELECT datediff("d", date1, date2) AS days
FROM
 (SELECT pubdate as date1
 FROM titles
 WHERE title_id = 'T05') t1,
 (SELECT pubdate as date2
 FROM titles
 WHERE title_id = 'T06') t2;

-- Count the months between two dates.
SELECT datediff("m", date1, date2) AS months
FROM
 (SELECT
 MIN(pubdate) AS date1,
 MAX(pubdate) AS date2
 FROM titles) t1;
```

#### ✓ 提示

`datepart()`函数的一个替代是抽取函数`day()`、`month()`和`year()`。

### 15.11.3 Oracle

函数`to_char()`将日期和时间转换为既定格式的字符串。`to_number()`转换它的参数为数字。`sysdate`返回当前（系统）的日期和时间。标准的加和减操作符用于从一个日期加和减一定天数。`add_months()`对一个日期加一定的月数。从一个日期减另一个日期得到的是两个日期之间的天数。

`months_between()`返回两个日期之间的月份数（代码15-47）。

#### 代码15-47 在Oracle中处理日期

```
-- Extract parts of the current datetime.
SELECT
 to_number(to_char(sysdate,'ss')) AS sec_pt,
 to_number(to_char(sysdate,'mi')) AS min_pt,
 to_number(to_char(sysdate,'hh24')) AS hr_pt,
 to_number(to_char(sysdate,'dd')) AS day_pt,
 to_number(to_char(sysdate,'mm')) AS mon_pt,
 to_number(to_char(sysdate,'yyyy')) AS yr_pt
FROM dual;

-- Add or subtract days, months, and years.
SELECT
 pubdate+2 AS p2d,
 pubdate-2 AS m2d,
 add_months(pubdate,+2) AS p2m,
 add_months(pubdate,-2) AS m2m,
 add_months(pubdate,+24) AS p2y,
 add_months(pubdate,-24) AS m2y
FROM titles
WHERE title_id = 'T05';

-- Count the days between two dates.
SELECT date2 - date1 AS days
FROM
 (SELECT pubdate as date1
 FROM titles
 WHERE title_id = 'T05') t1,
 (SELECT pubdate as date2
 FROM titles
 WHERE title_id = 'T06') t2;

-- Count the months between two dates.
SELECT months_between(date2,date1) AS months
FROM
 (SELECT
 MIN(pubdate) AS date1,
 MAX(pubdate) AS date2
 FROM titles) t1;
```

#### 15.11.4 IBM DB2

函数`second()`、`day()`、`month()`等用于获取日期和时间中的各个组成部分。`current_timestamp`返回当前（系统）的日期和时间。标准的增和减操作符用于从一个日期加和减一定的时间间隔。`days()`将一个日期转换为整数数字序列（代码15-48）。

**代码15-48 在IBM DB2中处理日期**

```
-- Extract parts of the current datetime.
SELECT
 second(current_timestamp) AS sec_pt,
 minute(current_timestamp) AS min_pt,
 hour(current_timestamp) AS hr_pt,
 day(current_timestamp) AS day_pt,
 month(current_timestamp) AS mon_pt,
 year(current_timestamp) AS yr_pt
FROM SYSIBM.SYSDUMMY1;

-- Add or subtract days, months, and years.
SELECT
 pubdate + 2 DAY AS p2d,
 pubdate - 2 DAY AS m2d,
 pubdate + 2 MONTH AS p2m,
 pubdate - 2 MONTH AS m2m,
 pubdate + 2 YEAR AS p2y,
 pubdate - 2 YEAR AS m2y
FROM titles
WHERE title_id = 'T05';

-- Count the days between two dates.
SELECT days(date2) - days(date1) AS days
FROM
 (SELECT pubdate as date1
 FROM titles
 WHERE title_id = 'T05') t1,
 (SELECT pubdate as date2
 FROM titles
 WHERE title_id = 'T06') t2;

-- Count the months between two dates.
SELECT
 (year(date2)*12 + month(date2)) -
 (year(date1)*12 + month(date1))
 AS months
FROM
 (SELECT
 MIN(pubdate) AS date1,
 MAX(pubdate) AS date2
 FROM titles) t1;
```

**15.11.5 MySQL**

函数`date_format()`将日期和时间按照指定的格式格式化。`current_timestamp`返回当前（系统）

**449** 日期和时间。标准的加和减操作符用于从一个日期加和减一定的时间间隔。`datediff()`返回两个日期之间相隔的天数（代码15-49）。

**代码15-49 在MySQL中处理日期**

```
-- Extract parts of the current datetime.
SELECT
```

```

date_format(current_timestamp, '%s')
 AS sec_pt,
date_format(current_timestamp, '%i')
 AS min_pt,
date_format(current_timestamp, '%k')
 AS hr_pt,
date_format(current_timestamp, '%d')
 AS day_pt,
date_format(current_timestamp, '%m')
 AS mon_pt,
date_format(current_timestamp, '%Y')
 AS yr_pt;

-- Add or subtract days, months, and years.
SELECT
 pubdate + INTERVAL 2 DAY AS p2d,
 pubdate - INTERVAL 2 DAY AS m2d,
 pubdate + INTERVAL 2 MONTH AS p2m,
 pubdate - INTERVAL 2 MONTH AS m2m,
 pubdate + INTERVAL 2 YEAR AS p2y,
 pubdate - INTERVAL 2 YEAR AS m2y
FROM titles
WHERE title_id = 'T05';

-- Count the days between two dates.
SELECT datediff(date2,date1) AS days
 FROM
 (SELECT pubdate as date1
 FROM titles
 WHERE title_id = 'T05') t1,
 (SELECT pubdate as date2
 FROM titles
 WHERE title_id = 'T06') t2;

-- Count the months between two dates.
SELECT
 (year(date2)*12 + month(date2)) -
 (year(date1)*12 + month(date1))
 AS months
 FROM
 (SELECT
 MIN(pubdate) AS date1,
 MAX(pubdate) AS date2
 FROM titles) t1;

```

### ✓ 提示

实现`date_format()`功能的其他方法是提取函数`extract()`、`second()`、`day()`、`month()`等。

## 15.11.6 PostgreSQL

函数`date_part()`提取日期和时间的特定部分。`current_timestamp`返回当前（系统）日期和时间。标准的加和减操作符用于从一个日期加和减一定的时间间隔。一个日期减去另一个日期得到的是它们之间相差的天数（代码15-50）。

450

15

**代码15-50 在PostgreSQL中处理日期**

```
-- Extract parts of the current datetime.
SELECT
 date_part('second',current_timestamp)
 AS sec_pt,
 date_part('minute',current_timestamp)
 AS min_pt,
 date_part('hour',current_timestamp)
 AS hr_pt,
 date_part('day',current_timestamp)
 AS day_pt,
 date_part('month',current_timestamp)
 AS mon_pt,
 date_part('year',current_timestamp)
 AS yr_pt;

-- Add or subtract days, months, and years.
SELECT
 pubdate + INTERVAL '2 DAY' AS p2d,
 pubdate - INTERVAL '2 DAY' AS m2d,
 pubdate + INTERVAL '2 MONTH' AS p2m,
 pubdate - INTERVAL '2 MONTH' AS m2m,
 pubdate + INTERVAL '2 YEAR' AS p2y,
 pubdate - INTERVAL '2 YEAR' AS m2y
FROM titles
WHERE title_id = 'T05';

-- Count the days between two dates.
SELECT date2 - date1 AS days
FROM
 (SELECT pubdate as date1
 FROM titles
 WHERE title_id = 'T05') t1,
 (SELECT pubdate as date2
 FROM titles
 WHERE title_id = 'T06') t2;

-- Count the months between two dates.
SELECT
 (date_part('year', date2)*12 +
 date_part('month',date2)) -
 (date_part('year', date1)*12 +
 date_part('month',date1))
 AS months
FROM
 (SELECT
 MIN(pubdate) AS date1,
 MAX(pubdate) AS date2
 FROM titles) t1;
```

**✓提示**

- 实现函数date\_part()功能的另一方式是extract()。

## 15.12 计算中值

中值是指作为（排序后的） $n$ 个值的数据中心点的值。如果 $n$ 是奇数，中值是第 $(n+1)/2$ 个数的值。如果 $n$ 是偶数，中值是 $n/2$ 和 $n/2+1$ 的中点（平均）值。本节例子计算表`empsales`（图15-39）中列`sales`的中值。中值是550——排序列表的中间两个数字，500和600的平均数。

网上和一些高级SQL图书中，能够找到一些计算中值的标准方法和特定方法。

代码15-51介绍了一种方法。该方法使用自联结和`GROUP BY`来创建一个没有重复的笛卡尔积（`e1`和`e2`），然后使用`HAVING`和`SUM`来发现`e1.sales=e2.sales`的次数等于（或超过）`e1.sales > e2.sales`的次数的行（包含中值）。像其他所有使用标准（或接近于标准的）SQL的方法一样，上述方法很麻烦、很难理解且运行得很慢。因为SQL是针对无序的集合，取得有序集合的中间值很困难。

### 代码15-51 用标准SQL计算销售的中值

```
SELECT AVG(sales) AS median
 FROM
 (SELECT e1.sales
 FROM empsales e1, empsales e2
 GROUP BY e1.sales
 HAVING
 SUM(CASE WHEN e1.sales = e2.sales
 THEN 1 ELSE 0 END) >=
 ABS(SUM(SIGN(e1.sales -
e2.sales)))) t1;
```

### 中值和均值

中值是常用的统计指标。因为中值是粗略的，意味着它不受极大数或极小数的严重影响，不论这些影响是正常的还是错误引起的。而算术平均（平均数）非常敏感，添加或删除一个极值就能够严重地影响它。因此中值常应用于不对称（偏向的）属性，如健康、房子价格、军事预算和基因表达式。中值也称作第50个百分位数（*50th percentile*）或第二个四分位数（*second quartile*），参见15.13节。

**DBMS** 如果可以使用DBMS特定的函数计算中值，将既快速又高效。代码15-52在Microsoft SQL Server中计算中值。代码15-53在Oracle中计算中值。代码15-52的第二个查询可以在DB2中计算中值。*DB2 SQL Reference, Vol. 2*介绍了如何使用游标创建中值的过程（在行中移动的滚动标志，本书没有介绍）。

| emp_id | sales |
|--------|-------|
| E07    | 300   |
| E08    | 400   |
| E03    | 500   |
| E04    | 500   |
| E06    | 500   |
| E01    | 600   |
| E05    | 700   |
| E10    | 700   |
| E02    | 800   |
| E09    | 900   |

图15-39 表`empsales`按销售升序排列

451

**代码15-52 在Microsoft SQL Server中计算中值的两种方法。第二种方法（在DB2中也可以运行）要比第一种快得多**

```
-- Works in SQL Server 2000 and later.
SELECT
(
 (SELECT MAX(sales) FROM
 (SELECT TOP 50 PERCENT sales
 FROM empsales
 ORDER BY sales ASC) AS t1)
 +
 (SELECT MIN(sales) FROM
 (SELECT TOP 50 PERCENT sales
 FROM empsales
 ORDER BY sales DESC) AS t2)
)/2 AS median;

-- Works in SQL Server 2005 and later.
-- Works in DB2.
SELECT AVG(sales) AS median
FROM
(
 SELECT
 sales,
 ROW_NUMBER() OVER (ORDER BY sales)
 AS rownum,
 COUNT(*) OVER () AS cnt
 FROM empsales) t1
WHERE rownum IN ((cnt+1)/2, (cnt+2)/2);
```

**代码15-53 在Oracle中计算中值的两种方法**

```
-- Works in Oracle 9i and later.
SELECT
 percentile_cont(0.5)
 WITHIN GROUP (ORDER BY sales)
 AS median
 FROM empsales;

-- Works in Oracle 10g and later.
SELECT median(sales) AS median
 FROM empsales;
```

### ✓提示

- 如果使用其他方法来计算中值，要确保在计算期间没有删除重复的值，并且如果n是偶数，要计算两个中位数的平均值（不是仅仅简单地选择其中之一作为中值）。
- 参见6.6节中的“SQL中的统计”提要栏。
- **DBMS** 要在Microsoft Access中运行代码15-51，将CASE表达式变为iif(e1.sales = e2.sales, 1, 0)，且将SIGN变为SGN。

## 15.13 查询极值

代码15-54在表royalties列advance中查找最高和最低值（包括并列的）所在的行。图15-40显示结果。

**代码15-54** 列出最高和最低预付款的图书，结果见图15-40

```
SELECT title_id, advance
 FROM royalties
 WHERE advance IN (
 (SELECT MIN(advance) FROM royalties),
 (SELECT MAX(advance) FROM royalties));
```

## title\_id advance

|     |            |
|-----|------------|
| T07 | 1000000.00 |
| T08 | 0.00       |
| T09 | 0.00       |

图15-40 运行代码15-54的结果

## ✓ 提示

- 也可以使用15.4节的查询来发现极值，但不会在同一个查询中发现最高和最低值。
- **DBMS** 在Microsoft SQL Server、Oracle和DB2中，可以使用窗口函数MIN OVER和MAX OVER改写代码15-54（代码15-55）。

453

**代码15-55** 使用窗口函数列出最高和最低预付款的图书

```
SELECT title_id, advance
 FROM
 (SELECT title_id, advance,
 MIN(advance) OVER () min_adv,
 MAX(advance) OVER () max_adv
 FROM royalties) t1
 WHERE advance IN (min_adv, max_adv);
```

## 15.14 改变动态统计的中流

可以依据另一列来修改正在进行动态统计的值。首先，请回顾15.1节的代码15-1。

代码15-56计算忽略生物类的图书销售量的动态总计。内部的CASE表达式标识出生物类并将其销售值变为空值，这样它们将被聚合函数SUM()忽略，而标量子查询计算动态总计。（外层的CASE表达式仅创建结果中的标签列，它和动态总计没有关系。）图15-41显示结果。

454

**代码15-56** 计算忽略生物类的图书动态总计，结果见图15-41

```
SELECT
 t1.title_id,
 CASE WHEN t1.type = 'biography'
 THEN '*IGNORED*'
 ELSE t1.type END
 AS title_type,
 t1.sales,
 (SELECT
 SUM(CASE WHEN t2.type = 'biography'
 THEN NULL
 ELSE t2.sales END)
 FROM titles t2
 WHERE t1.title_id >= t2.title_id)
 AS RunSum
 FROM titles t1;
```

| title_id | title_type | sales   | RunSum |
|----------|------------|---------|--------|
| T01      | history    | 566     | 566    |
| T02      | history    | 9566    | 10132  |
| T03      | computer   | 25667   | 35799  |
| T04      | psychology | 13001   | 48800  |
| T05      | psychology | 201440  | 250240 |
| T06      | *IGNORED*  | 11320   | 250240 |
| T07      | *IGNORED*  | 1500200 | 250240 |
| T08      | children   | 4095    | 254335 |
| T09      | children   | 5000    | 259335 |
| T10      | *IGNORED*  | NULL    | 259335 |
| T11      | psychology | 94123   | 353458 |
| T12      | *IGNORED*  | 100001  | 353458 |
| T13      | history    | 10467   | 363925 |

图15-41 运行代码15-56的结果

15

## ✓ 提示

- 在内部的CASE表达式中，可以设置值为任何数而不只是空值来求总计。例如，对银行事务求和，可以将储蓄设为正，将取款设为负。
  - **DBMS** = 'biography', '\*IGNORED\*', t1.type) 和 iif (t2.type = 'biography', NULL, t2.sales)。
- 在Oracle和DB2中，可以通过窗口函数SUM OVER改写代码15-56（代码15-57）

**代码15-57 忽略生物类并使用窗口函数计算图书销售量动态总计**

```

SELECT
 title_id,
 CASE WHEN type = 'biography'
 THEN '*IGNORED*'
 ELSE type END
 AS title_type,
 sales,
 SUM(CASE WHEN type = 'biography'
 THEN NULL
 ELSE sales END)
 OVER (ORDER BY title_id, sales)
 AS RunSum
FROM titles;

```

## 15.15 旋转结果

旋转表（交换其行与列的位置）的典型应用是在报告中以简洁的格式显示数据。

代码15-58使用SUM函数和CASE表达式列出每个作者写（或合写）过的图书数量，但不用通常的方式显示结果（例如代码6-9）。

| au_id | num_books |
|-------|-----------|
| A01   | 3         |
| A02   | 4         |
| A03   | 2         |
| A04   | 4         |
| A05   | 1         |
| A06   | 3         |
| A07   | 0         |

**代码15-58 列出每个作者写（或合写）过的图书数量，旋转结果**

```

SELECT
 SUM(CASE WHEN au_id='A01'
 THEN 1 ELSE 0 END) AS A01,
 SUM(CASE WHEN au_id='A02'
 THEN 1 ELSE 0 END) AS A02,
 SUM(CASE WHEN au_id='A03'
 THEN 1 ELSE 0 END) AS A03,
 SUM(CASE WHEN au_id='A04'
 THEN 1 ELSE 0 END) AS A04,
 SUM(CASE WHEN au_id='A05'

```

```

 THEN 1 ELSE 0 END) AS A05,
SUM(CASE WHEN au_id='A06'
 THEN 1 ELSE 0 END) AS A06,
SUM(CASE WHEN au_id='A07'
 THEN 1 ELSE 0 END) AS A07
FROM title_authors;

```

代码15-58产生一个旋转的结果：

```

A01 A02 A03 A04 A05 A06 A07

3 4 2 4 1 3 0

```

代码15-59反向进行旋转。FROM子句的第一个查询返回了作者唯一ID。第二个子查询再次得到代码15-58的结果。

456

#### 代码15-59 列出每个作者写（或合写）过的图书数量，反向旋转结果

```

SELECT
 au_ids.au_id,
 CASE au_ids.au_id
 WHEN 'A01' THEN num_books.A01
 WHEN 'A02' THEN num_books.A02
 WHEN 'A03' THEN num_books.A03
 WHEN 'A04' THEN num_books.A04
 WHEN 'A05' THEN num_books.A05
 WHEN 'A06' THEN num_books.A06
 WHEN 'A07' THEN num_books.A07
 END
 AS num_books
FROM
 (SELECT au_id FROM authors) au_ids,
 (SELECT
 SUM(CASE WHEN au_id='A01'
 THEN 1 ELSE 0 END) AS A01,
 SUM(CASE WHEN au_id='A02'
 THEN 1 ELSE 0 END) AS A02,
 SUM(CASE WHEN au_id='A03'
 THEN 1 ELSE 0 END) AS A03,
 SUM(CASE WHEN au_id='A04'
 THEN 1 ELSE 0 END) AS A04,
 SUM(CASE WHEN au_id='A05'
 THEN 1 ELSE 0 END) AS A05,
 SUM(CASE WHEN au_id='A06'
 THEN 1 ELSE 0 END) AS A06,
 SUM(CASE WHEN au_id='A07'
 THEN 1 ELSE 0 END) AS A07,
 FROM title_authors) num_books;

```

#### ✓ 提示

- DBMS** 在Microsoft Access中运行代码15-58和代码15-59，将简单的CASE表达式改为iff函数（例如，将代码15-58中的第一个CASE表达式改为`if(au_id = 'A01', 1, 0)`），将CASE搜索表达式改为switch()函数（参见5.14节的DBMS提示）。

457

15

## 15.16 处理层次结构

层次结构(hierarchy)是对组织中的人或系统中的事物的排列。每一个元素(除了最顶层的一个)是唯一一个元素的下级。图15-42是表示一个公司等级顺序的树状图，首席执行官(CEO)处在最顶端，在副总经理(VP)、主任(DIR)和雇员(WS)之上。

层级树有自己的术语。树上每个元素是一个节点。节点被树枝联结。两个联结的节点形成了父子关系(3个联结的节点形成祖-父-子关系等等)。金字塔的顶部是根节点(例如CEO)。节点没有子节点则被称作末节点或叶节点(DIR2和所有的WS)。枝节点联结到叶节点或其他枝节点(VP1、VP2、DIR1和DIR3——中层管理)。

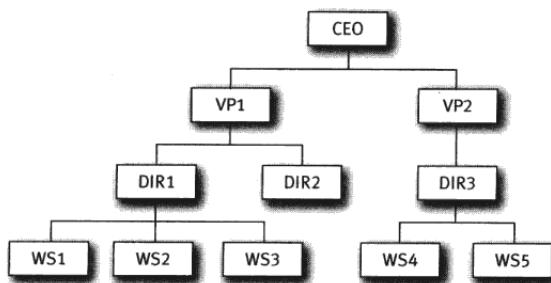


图15-42 显示了一个简单公司的层级组织图

表hier(图15-43)表示了图15-42中的树。表hier和7.9节中的表employees有相同的结构。自联结的知识是解决层次结构问题的基础。

| emp_id | emp_title | boss_id |
|--------|-----------|---------|
| E01    | CEO       | NULL    |
| E02    | VP1       | E01     |
| E03    | VP2       | E01     |
| E04    | DIR1      | E02     |
| E05    | DIR2      | E02     |
| E06    | DIR3      | E03     |
| E07    | WS1       | E04     |
| E08    | WS2       | E04     |
| E09    | WS3       | E04     |
| E10    | WS4       | E06     |
| E11    | WS5       | E06     |

图15-43 SELECT \* FROM hier的查询结果。表hier表示了图15-42中的组织图

### ✓ 提示

- 层次在现实和数据库中都很常见。在本章开始的提要栏“高级SQL图书”中的大多数图书都介

绍了层次结构，比本书更详细。要了解高级处理方法，请阅读Joe Celko所著的*Trees and Hierarchies in SQL for Smarties* (Morgan Kaufmann)。

代码15-60使用自联结列出谁为谁工作，结果见图15-44。

458

#### 代码15-60 列出父子关系，结果见图15-44

```
SELECT h1.emp_title ||
 ' obeys ' ||
 h2.emp_title
 AS power_structure
 FROM hier h1, hier h2
 WHERE h1.boss_id = h2.emp_id;
```

##### ✓ 提示

- DBMS** 在Microsoft Access和Microsoft SQL Server中运行代码15-60，将每一个||改为+。在MySQL中，使用CONCAT()连接字符串，参见5.4节。

代码15-61使用多个自联结来追踪从雇员WS3到树顶层的指挥链，遍历了各个层级，结果见图15-45。但是在写这个查询前必须知道层级的深度；如果可能，可以使用提示中介绍的方法。

459

460

| power_structure |
|-----------------|
| -----           |
| VP1 obeys CEO   |
| VP2 obeys CEO   |
| DIR1 obeys VP1  |
| DIR2 obeys VP1  |
| DIR3 obeys VP2  |
| WS1 obeys DIR1  |
| WS2 obeys DIR1  |
| WS3 obeys DIR1  |
| WS4 obeys DIR3  |
| WS5 obeys DIR3  |

图15-44 运行代码15-60的结果

#### 代码15-61 显示雇员WS3的完整的层级关系，结果见图15-45

```
SELECT
 h1.emp_title || ' < ' ||
 h2.emp_title || ' < ' ||
 h3.emp_title || ' < ' ||
 h4.emp_title
 AS chain_of_command
 FROM hier h1, hier h2, hier h3, hier h4
 WHERE h1.emp_title = 'WS3'
 AND h1.boss_id = h2.emp_id
 AND h2.boss_id = h3.emp_id
 AND h3.boss_id = h4.emp_id;
```

##### ✓ 提示

- DBMS** 在Microsoft Access和Microsoft SQL Server中运行代码15-61，将||改为+。在MySQL中，使用CONCAT()连接字符串，参见5.4节。

在Microsoft SQL Server和DB2中，使用（标准的）递归WITH子句来遍历一个层级结构，以下查询与代码15-61等价（在SQL Server中将每个||变为+）。

| chain_of_command       |
|------------------------|
| -----                  |
| WS3 < DIR1 < VP1 < CEO |

图15-45 运行代码15-61的结果

```
WITH recurse (chain, emp_level,
 boss_id) AS
(SELECT
 CAST(emp_title
 AS VARCHAR(50)),
 0,
```

15

```

boss_id
FROM hier
WHERE emp_title = 'WS3'
UNION ALL
SELECT
 CAST(recurse.chain || ' < ' ||
 hier.emp_title
 AS VARCHAR(50)),
 recurse.emp_level + 1,
 hier.boss_id
FROM hier, recurse
WHERE recurse.boss_id =
 hier.emp_id
)
SELECT chain AS chain_of_command
FROM recurse
WHERE emp_level = 3;

```

在Microsoft SQL Server和DB2中，列出向某个雇员（在这个例子中是VP1）直接或间接报告的所有人（通过上级的上级），使用以下查询。

```

WITH recurse (emp_title, emp_id) AS
(SELECT emp_title,emp_id
 FROM hier
 WHERE emp_title = 'VP1'
UNION ALL
SELECT hier.emp_title, hier.emp_id
 FROM hier, recurse
 WHERE recurse.emp_id =
 hier.boss_id
)
SELECT emp_title AS "Works for VP1"
 FROM recurse
 WHERE emp_title <> 'VP1';

```

在Oracle 10g或之后的版本中，使用（非标准的）CONNECT BY语法来遍历一个层级。以下查询等价于代码15-61。

```

SELECT LTRIM(SYS_CONNECT_BY_PATH(
 emp_title, '< '),
 AS chain_of_command
)
FROM hier
WHERE LEVEL = 4
START WITH emp_title = 'WS3'
CONNECT BY PRIOR boss_id = emp_id;

```

在Oracle 10g或之后版本中，列出向某个雇员（在这个例子中是VP1）直接或间接报告的所有人（通过上级的上级），使用以下查询。

```

SELECT emp_title AS "Works for VP1"
 FROM hier
 WHERE emp_title <> 'VP1'
START WITH emp_title = 'VP1'
CONNECT BY PRIOR emp_id = boss_id;

```

代码15-62使用多个UNION和自联结来追踪每个雇员的指挥链来遍历层级结构，结果见图15-46。不幸的是，在写查询之前必须知道层级深度。如果可能，应使用提示中的方法之一。

✓ 提示

□ **DBMS** 因为Access对联结表达式的限制，Microsoft Access无法执行代码15-62。

在Microsoft SQL Server中运行代码15-62，将每个||变为+。

在MySQL中运行代码15-62，使用CONCAT()而不是||连接字符串。

在Microsoft SQL Server和DB2中，使用（标准的）递归WITH子句来遍历一个层级结构。以下查询与代码15-62等价（在SQL Server中将每个||变为+）。

```
WITH recurse (emp_title, emp_id) AS
 (SELECT
 CAST(emp_title
 AS VARCHAR(50)),
 emp_id
 FROM hier
 WHERE boss_id IS NULL
 UNION ALL
 SELECT
 CAST(recurse.emp_title ||
 ' > ' ||
 h1.emp_title
 AS VARCHAR(50)),
 h1.emp_id
 FROM hier h1, recurse
 WHERE h1.boss_id =
 recurse.emp_id
)
SELECT emp_title emp_tree
 FROM recurse;
```

462

代码15-62 显示每个雇员的完整层次关系，结果见图15-46

```
SELECT chain AS chains_of_command
 FROM
 (SELECT emp_title as chain
 FROM hier
 WHERE boss_id IS NULL
 UNION
 SELECT
 h1.emp_title || ' > ' ||
 h2.emp_title
 FROM hier h1
 INNER JOIN hier h2
 ON (h1.emp_id = h2.boss_id)
 WHERE h1.boss_id IS NULL
 UNION
 SELECT
 h1.emp_title || ' > ' ||
 h2.emp_title || ' > ' ||
 h3.emp_title
 FROM hier h1
 INNER JOIN hier h2
 ON (h1.emp_id = h2.boss_id)
 LEFT OUTER JOIN hier h3
 ON (h2.emp_id = h3.boss_id)
 WHERE h1.emp_title = 'CEO'
```

| chains_of_command      |
|------------------------|
| -----                  |
| CEO                    |
| CEO > VP1              |
| CEO > VP1 > DIR1       |
| CEO > VP1 > DIR1 > WS1 |
| CEO > VP1 > DIR1 > WS2 |
| CEO > VP1 > DIR1 > WS3 |
| CEO > VP1 > DIR2       |
| CEO > VP2              |
| CEO > VP2 > DIR3       |
| CEO > VP2 > DIR3 > WS4 |
| CEO > VP2 > DIR3 > WS5 |

图15-46 运行代码15-62的结果

15

```

UNION
SELECT
 h1.emp_title || ' > ' ||
 h2.emp_title || ' > ' ||
 h3.emp_title || ' > ' ||
 h4.emp_title
FROM hier h1
INNER JOIN hier h2
 ON (h1.emp_id = h2.boss_id)
INNER JOIN hier h3
 ON (h2.emp_id = h3.boss_id)
LEFT OUTER JOIN hier h4
 ON (h3.emp_id = h4.boss_id)
WHERE h1.emp_title = 'CEO'
) chains
WHERE chain IS NOT NULL
ORDER BY chain;

```

在Oracle 10g或之后的版本中，使用（非标准的）CONNECT BY语法来遍历一个层次结构。以下查询等价于代码15-62。

```

SELECT ltrim(SYS_CONNECT_BY_PATH(
 emp_title, ' > '), ' > ')
 AS chains_of_command
FROM hier
START WITH boss_id IS NULL
CONNECT BY PRIOR emp_id = boss_id;

```

463

代码15-63使用标量子查询来确定每个节点在层级结构中是根，是枝，还是叶节点，结果见图15-47。在结果中，零意味着True，非零意味着False。

### ✓ 提示

- 在Microsoft Access中运行代码15-63，将每个SIGN变为SGN。
- **DBMS** 在Oracle 10g或之后的版本中，使用（非标准的）CONNECT BY语法来遍历一个层级。以下查询等价于代码15-63。

```

SELECT
 emp_title,
 (CASE CONNECT_BY_ROOT(emp_title)
 WHEN emp_title THEN 1
 ELSE 0 END)
 AS root_node,
 (SELECT COUNT(*)
 FROM hier h1
 WHERE h1.boss_id = hier.emp_id
 AND hier.boss_id IS NOT NULL
 AND rownum = 1)
 AS branch_node,
 CONNECT_BY_ISLEAF AS leaf_node
FROM hier
START WITH boss_id IS NULL
CONNECT BY PRIOR emp_id = boss_id
ORDER BY root_node DESC,
 branch_node DESC;

```

464

**代码15-63 确定每个节点是根，是枝，还是叶节点，结果见图15-47**

```

SELECT h1.emp_title,
 (SELECT SIGN(COUNT(*))
 FROM hier h2
 WHERE h1.emp_id = h2.emp_id
 AND h2.boss_id IS NULL)
 AS root_node,
 (SELECT SIGN(COUNT(*))
 FROM hier h2
 WHERE h1.emp_id = h2.boss_id
 AND h1.boss_id IS NOT NULL)
 AS branch_node,
 (SELECT SIGN(COUNT(*))
 FROM hier h2
 WHERE 0 =
 (SELECT COUNT(*)
 FROM hier h3
 WHERE h1.emp_id = h3.boss_id))
 AS leaf_node
FROM hier h1;

```

|      | emp_title | root_node | branch_node | leaf_node |
|------|-----------|-----------|-------------|-----------|
| CEO  | 1         | 0         | 0           | 0         |
| VP1  | 0         | 1         | 0           | 0         |
| VP2  | 0         | 1         | 0           | 0         |
| DIR1 | 0         | 1         | 0           | 0         |
| DIR2 | 0         | 0         | 1           | 0         |
| DIR3 | 0         | 1         | 0           | 0         |
| WS1  | 0         | 0         | 1           | 0         |
| WS2  | 0         | 0         | 1           | 0         |
| WS3  | 0         | 0         | 1           | 0         |
| WS4  | 0         | 0         | 1           | 0         |
| WS5  | 0         | 0         | 1           | 0         |

图15-47 运行代码15-63的结果

# 索引

索引中页码为英文原书页码，与书中边栏页码一致。

## 符号

+ (加号), 130, 131–132  
\ (反斜杠), 3  
[] (方括号), 67, 117  
/\* \*/ (注释符), 64  
^ (插入符号), 117  
/ (除操作符), 130, 131–132  
\ (双反斜杠), 3  
" (双引号), 63  
= (等于操作符), 101, 110  
> (大于操作符), 101  
>= (大于等于操作符), 101  
< (小于操作符), 101  
<= (小于等于操作符), 101  
\* (乘操作符), 130, 131–132  
<> (不等于操作符), 101  
() (括号), 63, 106  
% (百分号操作符), 114  
; (分号), 62, 63  
' (单引号), 70, 71  
/ (斜线), 3  
- (减操作符), 130, 131–132  
\_ (下划线操作符), 114  
|| (连接操作符), 134–136

## A

absolute pathnames (绝对路径名), 3  
Access. 参见 Microsoft Access  
ACID acronym (ACID首字母缩写), 403  
addition operator (加号操作符, +), 130, 131–132  
aggregate functions (聚合函数), 169–192  
    AVG(), 170, 175–176  
    COUNT(), 170, 175, 176, 177, 178, 185  
    creating (创建), 171  
    DISTINCT(), 179–182  
    filtering groups with HAVING (使用 HAVING 分组), 169, 190–192  
GROUP BY clauses with (GROUP BY子句), 169, 183–189

inner join combined with GROUP BY and (内联结与 GROUP BY 结合), 215, 217  
listing of (列表), 170  
MAX(), 170, 173  
MIN(), 170, 172  
returning single values with subqueries (使用子查询返回唯一值), 276  
SUM(), 170, 174  
ALL keyword (ALL关键字), 94, 288–290  
ALTER TABLE statement (ALTER TABLE语句), 337, 373–375  
ALTER VIEW statement (ALTER VIEW语句), 389

- books written (by publisher) [图书(按出版社)], 223–224  
 books written (by title) [图书(按书名)], 221  
 calculating greatest number of titles written by (计算所写数量最多的图书), 273  
 comparing values in subqueries with ALL (使用ALL比较子查询中的值), 289–290  
 creating table of unpublished (创建没有出版的表), 244, 295  
 earned royalties by book and (书获得的稿酬), 226–229  
 finding number of books by (查询图书数量), 243, 272  
 finding pairs by location (按地址查询成对的), 251–252  
 grouping names of coauthors and sole (按合写作者和唯一作者分组), 286  
 join listing cities of publisher and (联结出版社所在的城市), 238  
 listed by above-average sales (超过平均销售), 278  
 listing books by (列出图书), 211  
 listing by latest date of publications (按最新出版日期列出), 272  
 listing by volume of book sales and (按图书销量列出), 245  
 living in different city from publisher (和出版社不在同一城市), 297, 312  
 living in publisher's city (和出版社在同一城市), 296, 310  
 names of sole (唯一作者的名字), 285  
 outer joins listing all rows with nulls (列出包含空值行的外联结), 239  
 querying names of co-authors (查询合写者的名字), 284, 285  
 residing in publisher's city/state (居住在出版社所在城市/州), 213  
 residing same state as publisher (和出版社同在一州), 276–277  
 royalty comparisons with subqueries (使用子查询比较稿酬), 279  
 sorting by genre of writing (按写作类型排序), 284  
 sorting by specific location (按地址排序), 249  
 subquery comparisons using ANY (使用ANY的子查询比较), 292  
 total book royalties paid by publisher (出版社支付的图书总稿酬), 233–234  
 unpublished (未出版的), 282, 283  
 using equivalent queries on (使用等价的查询), 301  
 writing different genres of books (写作不同类型的图书), 298  
 writing three or more books (写作3本或更多图书), 297  
 authors table (表authors)  
 adding new rows to and listing (插入新行并列出), 321–322  
 creating views of (创建视图), 388, 389  
 deleting rows from (删除行), 334  
 structure of (结构), 51, 52  
 author\_title\_names table (表author\_title\_names), 370  
 autocommit mode (自动提交模式), 404  
 averages (平均数)  
 moving (移动), 407  
 running (累计), 406  
 AVG() aggregate function (AVG()聚合函数), 170, 175–176
- 
- B**
- backslash () (反斜杠), 3
  - balanced trees (B-trees) (平衡树(B树)), 382
  - batch files (批文件), 2
  - batch mode (批模式), 2
  - BETWEEN condition (BETWEEN条件), 118–120
  - binary table operations (二进制表操作), 36
  - BLOB (binary large object) data types (二进制大对象数据类型), 72
  - books (图书)
    - advances by genre (按类型的预付款), 216–217
    - authors writing three or more (写过3本或更多书的作者), 297
    - by author and listed by publisher (按作者和出版社列出), 223–224
    - calculating running sum and average for (计算累计总计和平均), 406
    - changing prices by genre (按类型改变价格), 328
    - comparing values in subqueries with ALL (使用ALL比较子查询的值), 289–290
    - computing running sum of sales (计算销售的累计总计), 273
    - filtering books written by author's name (按图书的作者名筛选图书), 221
    - finding authors who haven't written (查找没写过书的作者), 282, 283
    - genres listed by greater sales volume (最大销量图书的类型), 250
    - greatest number written by authors (作者所写图书的最大数量), 273
    - having above-average sales (高于平均销量), 277
    - listed by sales volume and author (按销量和作者列出), 245
    - listing authors by latest published (按最近出版日期列出作者), 272
    - listing by authors (按作者列出), 211
    - listing number by author (按作者列出数量), 243, 272
    - names and IDs of publisher and (出版社的名称和ID), 212

place of publication (出版地点), 214  
 priced greater than highest price genre (价格高于类型最高价格), 278  
 revenues greater than advance (收入高于预付), 220  
 sale prices compared by genre (按类型比较售价), 280  
 subqueries listing publishers by genre (按类型列出出版社的子查询), 254–255  
 subquery comparisons using ANY (使用ANY的比较子查询), 292, 293  
 total royalties for all (总稿酬), 225  
 types of published by several publishers (多家出版社都出版的类型), 286, 287  
 updating table values for (修改表的值), 329–330  
**books sample database** (books示例数据库).另见specific tables  
 about (关于), 51  
 authors table for (表authors), 51, 52  
 creating sample (创建示例), 57  
 listing of books\_standard.sql script  
   (books\_standard.sql脚本的代码), 57–60  
 publishers table (表publishers), 51, 53  
 royalties table of (表royalties), 56  
 title\_authors table (表title\_authors), 55  
 titles table of (表titles), 54  
 books\_standard.sql script (books\_standard.sql脚本), 57–60  
 Boolean types, (布尔类型) 76  
 Boyce-Codd normal form (BCNF, Boyce-Codd范式), 50  
 bracketed comments /\* \*/, (注释符), 64  
 brackets ([ ]), (方括号)  
   filter patterns using (筛选模式), 117  
   using around identifiers (使用通用标识符), 67  
 branch nodes (支节点), 458

**C**

calculating statistics (计算统计)  
 mode (节点), 177  
 medians (中值), 451–452  
 overview (概览), 177  
 running statistics (累计统计), 406–408, 454–455  
 sum of set's values (集合值的汇总), 174, 179–182  
 trimmed mean (修整均值), 432  
 calendar tables (日历表), 414  
 candidate keys (候选键), 39  
 caret (^), 117  
**CASE expression** (CASE表达式)  
   correlated subqueries vs. (相关子查询), 274  
   evaluating conditional values with (判断条件的值), 161–164

case sensitivity (区分大小写)  
   changing (改变), 140–141  
   comparisons and (比较), 140, 173  
   keyword and identifier (关键字和标识符), 63  
**CAST()**, 157–160  
**catalogs** (目录), 439  
**character strings**.另见 substrings  
   case sensitivity of comparisons (区分大小写比较), 140–141, 173  
   comparison operators with (比较操作), 101  
   concatenating (连接), 134–136  
   example of string operations (串运算的例子), 127  
   extracting substrings (减子串), 137–139  
   finding length of (确定长度), 147–148  
   trimming characters from (除去字符), 142–146  
   types of (类型), 70–71  
**CHARACTER\_LENGTH() function** (CHARACTER\_LENGTH()函数), 147–148  
**CHECK constraints** (检查约束), 339, 363–365  
**clauses** (子句), 另见specific clauses  
   about (关于), 62, 63, 64  
 closure property for tables (表的封闭属性), 36  
 clustered indexes (聚集索引), 382  
**COALESCE() expression** (COALESCE()表达式), 161, 165, 170, 427, 438  
 Codd, E. F., 33, 34, 38  
 collating sequence (检查顺序), 96  
 column aliases (列别名)  
   creating (创建), 91–92  
   sorting by (排序), 99  
   WHERE clauses and (WHERE子句和), 104  
**columns** 列  
   about (关于), 34, 35, 37  
   adding UNIQUE constraints to (创建唯一约束), 359–362  
   comparing from similar domains (类似域比较), 199  
   constraints for (约束), 339–340  
   creating aliases with AS clauses (使用AS子句创建别名), 91–92  
   DEFAULT values for (默认值), 346–349  
   defining constraints for (创建约束), 363–365  
   derived (导出), 128–129  
   displaying table definitions for (显示表结构), 316–318  
   grouping, (分组), 171, 183, 184  
   inserting rows in (插入行), 320–322  
   joining unequal values in (联结不等的值), 220  
   modifying with ALTER TABLE (使用ALTER TABLE修改), 373–374  
   nullability in (可否为空值), 343–345  
   order in composite indexes (组合索引的顺序), 379

- qualifying column names (限定列名), 194–195, 267  
 retrieving from SELECT and FROM clauses (用SELECT和FROM子句检索数据), 88–90  
 self-joins within (自联结), 247–252  
 sorting rows by (对行排序), 95–96  
 specifying relative position for sorting (指明排序的相对位置), 97  
 subqueries vs. joins for working with (子查询和联结), 261  
 two tables on (两个表), 211–212  
 unordered (无序的), 35  
 using simple FOREIGN KEY constraints (使用简单的外键约束) 355–356  
 when to use indexes for (使用索引), 378  
**Command Center (IBM DB2 8.x)**, 21  
**Command Editor (IBM DB2 9.x)**, 22  
**command-line tools** (命令行工具),  
 IBM DB2 db2, 23–26  
 MySQL mysql, 27–29  
 Oracle sqlplus, 17–19  
 PostgreSQL psql, 30–32  
 SQL Server osql, 11, 12–13  
 SQL Server sqlcmd, 10, 15  
 using SQL with (使用SQL), 2, 3  
**comments** (注释), 62, 64  
**committing transactions** (提交事务)  
 about (关于), 400  
 using COMMIT statement (使用COMMIT语句), 335, 404  
**comparison operators** (比较操作符)  
 ALL modifications to (ALL限制), 288  
 ANY modifications to (ANY限制), 291  
 listing of (列表), 101  
 using in subqueries (在子查询中使用), 275–280  
**comparisons** (比较),  
 case sensitivity of (区分大小写), 140, 173  
 changing string case (改变字符串大小写), 140–141  
**composite constraints** (组合约束)  
 foreign key (外键), 356–357  
 primary key (主键), 352  
 unique (唯一), 361–362  
**composite indexes** (组合索引), 378, 379  
**concatenate operator** (||, 连接符), 134–136  
**concurrency transparency** (并发透明性), 401  
**conditions** (条件)  
 combining and negating (结合和取消), 105–109, 111–112  
 equivalent (等价), 112–113  
 filtering lists with IN (使用IN筛选列表), 121–123  
 filtering with BETWEEN (使用BETWEEN筛选), 118–120  
 matching row patterns with LIKE (LIKE匹配行模式)
- 114–117  
 re-expressing (再表达), 113  
 types of search (搜索类型), 101  
**consistency in transactions** (事务一致性), 403  
**constant expressions** (常量表达式), 128  
**constants** (常量), 参见literal values  
**constraints** (约束)  
 altering or dropping (修改或删除), 375  
 check (检查), 339, 363–365  
 column and table (列和表), 339–340  
 foreign key (外键), 339, 353  
 nullability (可否为空), 343  
 primary key (主键), 339, 350  
 unique (唯一), 359–362  
 using CONSTRAINT clauses (使用CONSTRAINT子句), 339–340  
**converting data types** (转换数据类型), 157–160  
**correlated subqueries** (相关子查询), 另见subqueries  
 comparing author's royalties with subqueries (使用子查询比较作者的稿酬), 279  
 computing running sum of book sales (计算图书销售的累计总计), 273  
 GROUP BY clauses vs. (GROUP BY子句和), 280  
 including null values in list of books and authors (在图书和作者列表中包含空值), 272  
 listing data in spreadsheet fashion with (在电子表格中列出数据), 271  
 qualifying column names in (限定列名), 272  
 simple vs. (简单), 266  
**correlation variables** (相关变量), 263  
**COUNT()** aggregate function (COUNT()聚合函数)  
 about (关于), 170  
 comparing equivalent subqueries using (比较等价的子查询), 301–302  
 COUNT (expr) vs. COUNT(\*), 185  
 creating inner joins with GROUP BY clause and (创建带 GROUP BY子句内联结), 215, 230–232  
**DISTINCT()** with, 179  
**forms of** (形式), 178  
 listing with duplicate rows with (列出重复行), 436  
 nulls and (空值和), 170, 176  
 statistics using (统计), 177  
**CREATE INDEX statement** (CREATE INDEX语句),  
 creating index with (创建索引), 378–382  
 unique indexes vs. unique constraints (唯一索引和唯一约束), 362  
**CREATE TABLE AS statement** (CREATE TABLE AS语句), 337, 369–372  
**CREATE TABLE statement** (CREATE TABLE语句)  
 adding UNIQUE constraints to columns (向列添加唯一约

束), 360  
defined (定义), 337  
defining foreign-key constraint in (创建外键约束), 353  
table creation with (创建表), 337, 338, 341–342  
CREATE TEMPORARY TABLE statement (CREATE TEMPORARY TABLE语句), 337, 366–368  
CREATE VIEW statement (CREATE VIEW语句), 386–390  
cross joins (交叉联结)  
accidentally turning inner to (意外地创建), 210  
creating (创建), 204–205  
defined (定义), 198  
current date and time (当前日期和时间), 154–155  
CURRENT\_DATE() function (CURRENT\_DATE()函数), 154–155  
CURRENT\_TIME() function (CURRENT\_TIME()函数), 154–155  
CURRENT\_USER() function (CURRENT\_USER()函数), 156  
cursors (游标), 99

**D**

data types (数据类型)  
approximate numeric (近似数据类型), 75  
characteristics of (特点), 68–69  
comparing subquery values of same (比较子查询的值), 275  
compatibility of for join columns (列联结的兼容性), 199  
converting with CAST() (使用CAST()转换), 157–160  
datetime (日期和时间), 77–79  
exact numeric (精确数字), 73–74  
interval (间隔), 80–81  
database management systems (数据库管理系统), 参见  
DBMSs;DBMS-specific SQL features  
databases (数据库) 另见books sample database  
books sample (books示例), 51  
command and queries listing tables in (列出表的命令和查询), 318  
denormalizing (反规范化), 50  
learning to design (为设计学习), 38  
picking random rows (随机选取行), 433–434  
providing views of data (提供数据视图), 386  
recovering and restoring data (恢复数据), 401  
renaming tables of (重命名表), 375  
rolling back transactions (回滚事务), 400, 403, 404  
dates (日期)  
listing author and publications by latest (按最新的出版日期列出作者和出版日期), 272  
sequence tables for incrementing (顺序表增加), 413  
using in queries (查询中使用), 445  
datetime operations (日期和时间操作)  
data types for (数据类型), 77–79  
example of (例子), 127

extracting part of (选取部分), 152–153  
formatting and (格式化), 90  
sequence tables for (顺序表), 412, 413  
DB2 CLP windows (DB2 CLP窗口), 23  
db2 command-line tool (db2命令行工具)  
exiting (存在), 26  
interactive mode for (互动模式), 24  
script mode for (脚本模式), 25  
showing options for (列出选项), 26  
starting (开始), 23  
DBMSs (database management systems) 另见  
DBMS-specific SQL features  
ANSI-89 vs. ANSI-92 syntax mode for (Access), 5  
command line for (命令行), 2  
determining SQL Server version running (确定SQL Server 版本), 10  
running in autocommit mode (在自动提交模式下), 404  
transactions in (事务), 399–404  
using SQL on (使用SQL), 2–3  
working with indexes (使用索引), 378–382  
DBMS-specific SQL features 另见 *specific database programs*  
Access versions (Access版本), 5  
aggregate functions (聚合函数), 171  
altering tables (修改表), 373, 375  
approximate numeric types (近似数据类型), 75  
BLOB data types (二进制大对象数据类型), 72  
Boolean types (布尔数据类型), 76  
calculating medians (计算中值), 452  
changing case of character strings (改变串的大小写), 141  
character string types (字符串类型), 71  
column aliases (列的别名), 91  
comparing values in subqueries with ALL (带有ALL比较子查询的值), 290  
concatenating strings (连接串), 136, 309  
converting data types (转换数据类型), 159–160  
CREATE TEMPORARY TABLE support (CREATE TEMPORARY TABLE支持), 368  
CREATE VIEW statement (CREATE VIEW语句), 390  
creating aliases with AS clauses (使用AS子句创建别名), 91–92  
cross joins (交叉联结), 205  
dates (日期), 445–450  
datetime data types (日期和时间类型), 79  
defining check constraints (创建检查约束), 365  
DELETE and TRUNCATE statements (DELETE和TRUNCATE语句), 336  
derived columns (导出列), 129  
determining order of operator evaluation (确定操作符的

顺序), 133  
**displaying table definitions** (显示表结构), 316–318  
**DISTINCT()** aggregate functions (**DISTINCT()**聚合函数), 181  
**dropped tables** (删除表), 376  
**dropping views** (删除视图), 398  
**duplicate rows** (重复的行), 437  
**evaluating conditional values with CASE expression** (使用CASE表达式判断条件值), 164  
**exact numeric data types** (精确数字类型), 73–74  
**EXCEPT operations** (EXCEPT操作), 313  
**extracting part of datetime or interval** (选取部分日期和时间或间隔), 153  
**extracting substrings** (选取子串), 138–139  
**filtering ranges** (过滤范围), 120  
**finding rows with extreme values** (查找含有极值的行), 453  
**generating sequences** (产生序列), 410, 414  
**getting current date and time** (得到当前日期和时间), 154–155  
**handling sorting** (处理排序), 98  
**hierarchies** (层次), 459, 461, 462–463, 464  
**IBM DB2 versions** (IBM DB2版本), 20  
**identifiers** (标识符), 66, 67  
**indexes** (索引), 380, 382  
**INSERT statements** (INSERT语句), 326  
**inserting column values with DEFAULT clause** (使用DEFAULT子句插入默认值), 348–349  
**INTERSECT operations** (INTERSECT操作), 311  
**interval data types** (间隔数据类型), 80–81  
**joins with WHERE and JOIN clauses** (使用WHERE和JOIN子句联结), 202  
**length of character strings** (字符串的长度), 148  
**limiting number of returned rows** (限制返回行的数量), 421–429  
**lists filtered with IN condition** (使用条件IN筛选), 121–123  
**making new table from existing one** (利用已存在表创建新表), 371–372  
**metadata retrieval** (元数据检索), 439–444  
**Microsoft SQL Server versions** (Microsoft SQL Server版本), 10  
**midstream changes to running statistics** (改变累计统计的中流), 455  
**modifications with CREATE TABLE listings** (使用CREATE TABLE修改), 342  
**modifying row values with UPDATE** (使用UPDATE修改行的值), 331–332  
**nulls** (空值), 84–85, 98, 126, 167, 345

**number and datetime formatting** (数字及日期和时间格式), 90  
**operators and functions for** (操作符和函数), 131, 132  
**Oracle versions** (Oracle版本), 17  
**ORDER BY clause for columns in** (ORDER BY子句), 100  
**OUTER JOIN syntax** (OUTER JOIN语法), 237, 239, 240, 241–242, 246  
**pattern matching** (模式匹配), 114–117  
**pivoting tables** (旋转表), 456–457  
**POSITION() function for finding substrings** (POSITION()函数发现子串), 151  
**PostgreSQL versions** (PostgreSQL 版本), 30  
**PRIMARY KEY constraints** (主键约束), 352  
**printing current user** (打印当前用户), 156  
**qualifying column names** (限定列名), 195, 267  
**randomization** (随机化), 433–434  
**ranking data** (对数据排名), 431  
**retrieving data through views** (通过视图检索数据), 392  
**row comparisons** (行比较), 104  
**running statistics calculations** (累计统计计算), 408  
**SELECT \* with EXISTS subqueries** (带有EXISTS子查询的SELECT \*), 299–300  
**specifying foreign key constraints** (创建外键约束), 358  
**string comparisons** (字比较), 172  
**subqueries and** (子查询和), 255, 261, 266, 269, 274, 287  
**SUM()** aggregate functions (**SUM()**聚合函数), 174  
**table alias creation with AS clauses** (使用AS子句创建表别名), 197  
**timing and measuring query execution** (查询执行的计时和计量), 302  
**transactions** (事务), 402, 404  
**trimmed mean calculations** (计算修整均值), 432  
**trimming function** (修整函数), 145–146  
**UNION operations** (合并操作), 309  
**unique identifiers** (唯一标识符), 82  
**user-defined types** (用户定义类型), 83  
**using inner joins** (使用内联结), 211, 222, 224, 226, 229, 232, 234  
**working with UNIQUE constraints** (使用唯一约束), 362  
**debugging WHERE clauses** (调试WHERE子句), 110  
**DEFAULT clause** (DEFAULT子句), 346–349  
**DELETE statement** (DELETE语句)  
    **limiting number of rows affected by** (限制影响行数), 421  
    **removing rows with** (删除行), 315, 333–336  
    **using updateable views in** (使用可修改的视图), 394  
**deleting** (删除)  
    **duplicate rows** (重复行), 93–94, 436–437  
    **row through view** (通过视图), 397

delimited identifiers (定界标识符), 66  
 denormalization (反规范化), 50  
   dependencies (依赖)  
   dependency-preserving decomposition (依赖-保持分解), 45  
   fully functional (完全函数的), 47–48  
   multivalued (多值), 50  
   partial functional (部分函数), 47–48  
   transitive (传递), 49  
 DESC clause (DESC子句), 422–428  
 directories (目录), 3  
 DISTINCT() aggregate function (DISTINCT()聚合函数)  
   aggregating values with (聚合值), 178, 179–182  
   eliminating duplicate values (删除重复值), 93  
 division (/) (除以)  
   arithmetic operator for (数学操作符), 130  
   avoiding divide-by-zero errors (避免被零除错误), 166  
   performing (执行), 131–132  
 domains (域)  
   column's (列的), 35  
   comparing columns of similar (比较相似的列), 199  
 double backslash (\) (双反斜杠), 3  
 double quotes ("") (双引号),  
   common errors with (常见的错误), 63  
   straight (直接), xvi  
 DROP INDEX statement (DROP INDEX语句), 383–384  
 DROP TABLE statement (DROP TABLE语句)  
   defined (定义), 337  
   deleting base tables with (删除基础表), 366  
   using (使用), 342, 376  
 DROP VIEW statement (DROP VIEW语句), 398  
 duplicate rows (重复的行)  
   deleting (删除), 93–94, 436–437  
   handling (处理), 435–437  
   listing (列表), 435–436  
 duplicate value tests (重复值的检测), 298–299  
 durability in transactions (事务的持久性), 403

**E**

employees table (表employees), 458  
 empty strings (空串), 71, 84, 126  
   in Oracle (在Oracle中), 85  
 encoding (嵌入), 71  
 end nodes (末节点), 458  
 equal to operator (=) (相等操作符), 101, 110  
 equivalent conditions (相等条件), 112–113  
 escaped and unescaped patterns (转义和非转义模式), 116  
 exact numeric types (精确数字类型), 73–74  
 EXCEPT operation (EXCEPT操作)

finding different rows (发现不同的行), 312–313  
 function of (函数), 303  
 precedence of INTERSECT with (INTERSECT的优先顺序), 311  
 updateable views with (修改视图), 394  
 EXISTS keyword (EXISTS关键字)  
   comparing equivalent subqueries using (比较等价的子查询), 301–302  
   testing subqueries with (测试子查询), 294–300  
 exiting (存在)  
   db2 command-line tool (db2命令行工具), 26  
   mysql command-line tool (mysql命令行工具), 27–29  
   osql command-line tool (osql命令行工具), 13  
   psql command-line tool (psql命令行工具), 31  
   sqlcmd command-line tool (sqlcmd命令行工具), 15  
   sqlplus command-line tool (sqlplus命令行工具), 19  
 expressions (表达式)  
   constant (常数), 128  
   operands as (运算域), 127  
   returning null for equivalent (相等则返回空值), 167  
   sorting by query results of (按查询结果排序), 100  
   types of SQL (SQL类型), 64  
   using subqueries as (使用子查询), 253, 278

**F**

fields, datetime (字段, 日期和时间), 78  
 fifth normal form (5NF) (第五范式), 50  
 FileMaker Pro, 4  
 files, script (文件, 脚本), 2  
 filtering (过滤), 另见INNER JOIN clause; joins  
   books by place of publication (按出版地点), 214  
   groups with HAVING (带有HAVING分组), 190–192  
   lists with IN condition (使用IN条件列出), 121–123  
   patterns (模式), 117  
   ranges (范围), 118–120  
   rows by matching patterns (匹配模式的行), 115  
   rows with WHERE clause (带WHERE子句的行), 101–104  
 Firebird, 4  
 first normal form (1NF) (第一范式), 45, 46  
 FOREIGN KEY constraint (外键约束), 353–358  
   composite (组合), 356–357  
   defining in CREATE TABLE statements (在CREATE TABLE语句中定义), 353  
   function of (函数), 339  
   simple (简单), 355–356  
 foreign keys. 另见FOREIGN KEY constraint  
   characteristics of (特点), 40–41  
   joins and (联结), 198  
   one-to-many relationships and (一对多联系), 43

one-to-one relationships and (一对-一联系), 42  
 primary and (主键和), 41  
 self-joins and (自联结和), 247  
 fourth normal form (4NF) (第四范式), 50  
 FROM clause (FROM子句), 88–90  
 FROM keyword (FROM关键字), 336  
 full outer joins (完全外联结)  
     creating (创建), 236  
     defined (定义), 198, 235  
     including all rows in table regardless of column match (包括表中所有行不管列匹配), 241  
 fully functional dependency (完全函数依赖), 47–48  
 functions. 参见 aggregate functions; operators and  
     functions; and specific functions  
 further reading (进一步阅读)  
     advanced SQL books (高级SQL图书), 405, 458  
     database design (数据库设计), 38  
     relational model (关系模型), 33  
     on runs, regions, and sequences (递增、等值和数列), 415  
     transaction theory (事务理论), 403

## G

global temporary tables (全局临时表), 366, 367, 370  
 greater than operator (>) (大于操作符), 101  
 greater than or equal to operator (>=) (大于等于操作符), 101  
 GROUP BY clause (GROUP BY子句)  
     adding to INNER JOIN syntax (内联结语法) 215, 217  
     aggregate functions with (聚合函数) 169  
     calculating royalty totals by publisher with (按出版社计算总稿酬), 230–232  
     comparing subquery values using (比较子查询的值), 278  
     correlated subquery vs. (相关子查询), 280  
     COUNT (*expr*) vs. COUNT(\*) in, 185  
     execution sequences with (执行顺序), 201  
     grouping rows with (行分组), 183–189  
     sequence with WHERE and HAVING clauses (WHERE和HAVING子句的顺序), 190  
     using with UNION operations (使用UNION合并操作), 304  
 grouping columns (列分组)  
     inner joins using (内联结), 233  
     using (使用), 171, 183, 184  
 GUID (Globally Unique Identifier (全局唯一标识符)), 82

## H

HAVING clause (HAVING子句)  
     comparing subquery values using (比较子查询的值), 278, 279

creating inner joins with (创建内联结), 219, 233–234  
 execution sequences with (执行顺序), 201  
 filtering groups with (过滤组), 169, 190–192  
 logical operators with (逻辑操作符), 191  
 sequence with WHERE and GROUP BY clauses (WHERE和GROUP BY子句的顺序), 190  
 using with UNION operations (UNION合并操作), 304  
 hexadecimal format (十六进制格式), 72  
 hier table (表hier), 458  
 hierarchies (层级), 458–464  
     about (关于), 458  
     listing parent-child relationships (列出父-子关系), 459  
 hosts (主机), 1

## I

IBM DB2, 20–26 另见 DBMS-specific SQL features  
 Command Center (v. 8), 21  
 Command Editor (v. 9), 22  
 db2 command-line tool for (db2命令行工具), 23–26  
 displaying table definitions for (显示表结构), 317  
 dropping indexes (删除索引), 383, 384  
 limiting number of returned rows (限制返回行的数量), 426  
 retrieving metadata (检索元数据), 439, 442  
 running SQL with, (运行SQL), 2–3  
 versions of (版本), 20  
 working with dates in (使用日期), 448  
 IBM DB2 Express-C, 20  
 identifiers (标识符)  
     about (关于) 62, 66–67  
     case for (大小写), 63  
 IN keyword (IN关键字)  
     comparing equivalent subqueries using (比较等价的子查询), 301–302  
     filtering lists with (过滤), 121–123  
     testing set membership with (检测集合成员资格), 281–287  
 incrementing dates (递增日期), 413  
 indexes (索引), 377–384  
     clustered and nonclustered (聚集和非聚集), 382  
     composite (组合), 378, 379  
     creating for columns grouped frequently (为经常用来分组的列创建), 188  
     creating with CREATE INDEX (使用CREATE INDEX创建), 378–382  
     dropping (删除), 383–384  
     function of (函数), 377  
     keys vs. (键), 381  
     simple (简单), 380

unique (唯一), 379, 381  
**INNER JOIN clause (INNER JOIN子句)**  
 calculating total royalties for all books with (计算所有书的稿酬总计), 225  
 comparing results of OUTER JOIN and (比较外联结的结果), 238–241  
 creating (创建), 210–234  
 creating self-joins (创建自联结), 248  
 defined (定义), 198, 210  
 equivalent subquery comparisons using (比较等价子查询), 301–302  
 four-table joins with (四表联结), 223  
 HAVING clause with (HAVING子句), 219, 233–234  
 joining unequal values in columns (联结不等值的列), 220  
**OUTER JOIN clause with (OUTER JOIN子句), 245–246**  
 subqueries vs. (子查询和), 257–258  
 three-table joins (三表联结), 210, 221–222, 226  
 using aggregate functions with GROUP BY (使用GROUP BY聚合函数), 215, 217, 230–232  
 using subqueries with (使用子查询), 254, 255  
 WHERE conditions with (WHERE条件), 214, 216  
 inner queries. 参见subqueries  
**INSERT statement (INSERT语句)**  
 adding default values to table in (向表添加默认值), 347  
 adding rows with (添加行), 315, 319–326  
 displaying table definitions when using (显示表结构), 316–318  
 populating new table with (填充新表), 342  
 specifying column's default values in (设置列的默认值), 346  
 using updateable views in (使用可更新的视图), 394  
 instances (实例), 37  
 interactive mode (互动模式)  
 db2 command-line tool (db2命令行工具), 24  
 defined (定义), 2  
 mysql command-line tool (mysql命令行工具), 27  
 osql, 12  
 psql command-line tool (psql命令行工具), 30  
 sqlcmd command-line tool, (sqlcmd命令行工具), 15  
 sqlplus command-line tool, (sqlplus命令行工具), 18  
**INTERSECT operation (INTERSECT操作), 303, 310–311**  
 interval operations (间隔操作), 80–81, 152–153  
 ISO (International Organization for Standardization (国际标准化组织)), xii  
 isolation in transactions (事务分离), 403

**J**

**JOIN clause (JOIN子句)**

creating joins with (创建联结), 200–202  
 execution sequences with (执行顺序), 201  
 replicating natural joins with (改写自然联结), 208  
 joins, 另见 *specific join clauses*, 193–252  
 creating cross (创建交叉), 204–205  
 defined (定义), 193  
 inner vs. outer (内部与外部), 235  
 natural (自然), 206–209  
 query execution sequences for (查询执行顺序), 201  
 self (自身), 198, 247–252  
 subqueries vs. (子查询和), 257–261  
 types defining updateable views (定义可更新的视图类型), 394  
 using (使用), 198–199  
 with USING clause (使用USING子句), 203  
 using JOIN syntax for (使用JOIN语法), 200, 202  
 WHERE syntax for (WHERE语法), 201, 202  
 junction tables (联结表), 44

**K**

keys. 另见foreign keys; primary keys  
 candidate and alternate (候选的和可更换的), 39  
 foreign (外部的), 40–41  
 indexes vs. (索引和), 381  
 joins and (联结和), 198  
 primary (主键), 36, 38–39  
**keywords. 另见 specific keywords**  
 case for (大小写), 63  
 defined (定义), 62  
**keywords (关键字)**  
 defining foreign-key constraint with (定义外键约束), 353  
 reserved and non-reserved (保留与非保留), 66, 67  
 SELECT statements using reserved (SELECT语句使用保留), 92

**L**

leaf nodes (叶节点), 458  
**LEFT [OUTER] JOIN clauses (左外联结语句)**  
 creating (创建) 236  
 creating phone list with COALESCE(), (使用COALESCE()创建电话列表), 438  
 defined (定义), 198, 235  
 including all rows of table in results (在结果中包含表中所有行), 239  
 listing number of books by author (按作者列出图书的数量), 243, 272  
 subqueries vs. (子查询和), 259  
 less than operator (<) (小于操作符), 101  
 less than or equal to operator (<=) (小于等于操作符), 101

level of conformance (一致性水平), 65  
**L**  
**LIKE** condition (**LIKE**条件), 114–117  
**LIMIT** clause (**LIMIT**子句), 428, 429  
**Listing** (列出)  
  **duplicate rows** (列出重复行) 435–436  
  **phone numbers with COALESCE()** (使用**COALESCE()**的电话号码), 438  
**literal values** (字面量)  
  **datetime** (日期和时间) 78  
  **interval** (间隔), 81  
  **storing** (存储), 69  
**local temporary tables** (局部临时表), 366, 367  
**logical operators** (逻辑操作符),  
  applying **HAVING** clause with (应用**HAVING**子句), 191  
  combining in compound conditions (复合条件的组合), 109  
  defined (定义), 105  
**lossless decomposition** (无损分解), 45  
**LOWER()** function (**LOWER()**函数), 140–141

**M**

**many-to-many relationships** (多对多联系), 44, 55  
**MAX()** aggregate function (**MAX()**聚合函数), 170, 173, 177  
**means** (均值)  
  calculating trimmed (计算修整的), 432  
  finding arithmetic (发现数学的), 175–176  
  medians vs. (中值和), 451  
  medians (中值), 451–452  
**metadata** (元数据)  
  defined (定义), 36  
  retrieving (检索), 439–444  
**Microsoft Access**. 另见 DBMS-specific SQL features  
  displaying existing queries in (显示存在的查询), 9  
  dropping indexes (删除索引), 383–384  
  limiting number of returned rows (限制返回行的数量), 422  
  pivoting tables (旋转表), 457  
  retrieving metadata (检索元数据), 439, 440  
  running SQL with (运行SQL), 2–3, 6–7, 8  
  table definitions for (表结构), 316  
  turning on ANSI-92 SQL syntax for (使用ANSI-92 SQL语法), 5–6  
  using statements with 2007 version (使用2007版本语句), 8  
  version used in book (本书使用的版本), 5  
  working with dates in (使用日期), 445  
**Microsoft SQL Server**. 另见DBMS-specific SQL features  
  calculating medians (计算中值), 451–452  
  conformance to SQL standards (SQL标准的一致性), 65

connecting to remote computer (连接远程计算机), 13, 16  
**D**  
  displaying table definitions for (显示表结构), 317  
  dropping indexes (删除索引), 383, 384  
  limiting number of returned rows (限制返回行的数量), 423–424  
**osql** command-line tool for, (**osql**命令行工具), 12–13  
  retrieving metadata (检索元数据), 439, 440  
  running programs for 2000 version (运行2000版程序), 11–12  
  running SQL with (运行SQL), 2–3  
**sqlcmd** command-line tool for (**sqlcmd**命令行工具), 15–16  
**SQL Server Management Studio** for 2005/2008 versions (2005/2008版SQL Server Management Studio), 14–16  
  using Access as front-end for (使用Access作为前台), 5  
  versions of (版本), 10  
  working with dates in (使用日期), 446  
**MIN()** aggregate function (**MIN()**聚合函数), 170, 172, 177  
**moving averages** (移动平均值), 407  
**multiple column selection** (多列选择), 88  
**multiplication operator (\*)** (乘法操作符), 130, 131–132  
**multivalued dependencies (MVD)** (多值依赖), 50  
**MySQL**. 另见 DBMS-specific SQL features  
  about (关于), 27  
  command-line tool in interactive mode (交互模式的命令行工具), 27  
  conformance to SQL standards (SQL标准的一致性), 65  
  displaying table definitions for (显示表结构), 318  
  dropping indexes (删除索引), 383–384  
  illustrated (演示), 27  
  limiting number of returned rows (限制返回行的数量), 427–428  
  mysql command-line tool for (**mysql**命令行工具), 27–29  
  retrieving metadata (检索元数据), 439, 443  
  running SQL with (运行SQL), 2–3  
  terminating statements with semicolon (使用分号结束语句), 27  
  working with dates in (使用日期), 449  
**mysql** command-line tool, (**mysql**命令行工具), 27–29

**N**

**names** (名称)  
  inserting rows using column (插入行), 321–322  
  naming constraints (命名约束), 340  
  qualifying column (限定列), 194–195, 267, 272  
  rules for identifier (标识符规则), 66, 67  
**natural joins** (自然联结)

creating inner join using (创建内联结), 213  
 defined (定义), 198  
 Navigation pane (Access) (导航面板 (Access)), 9  
 nesting (嵌套)  
 subqueries (子查询), 284, 329  
 views (视图), 387  
**new\_publishers table (new\_publishers表)**, 323–326  
**nodes (节点)**, 458  
 nonclustered indexes (非聚集索引), 382  
 normalization 另见 normal forms  
     avoiding table anomalies with (避免表异常), 45  
     defined (定义), 45  
     first normal form (第一范式), 45, 46  
     other types of normal forms (其他类型的范式), 50  
     second normal form (第二范式), 45, 47–48  
     sequence tables for (顺序表), 412, 413  
     third normal form (第三范式), 45, 49  
**not equal to operator ( $\neq$ ) (不等操作符)**, 101  
**NOT EXISTS keyword (NOT EXISTS关键字)**, 294–300  
**NOT IN keyword (NOT IN关键字)**, 281, 283  
**NOT NULL constraint (NOT NULL约束)**, 339, 343–345  
**NOT operator (NOT操作符)**, 105, 108, 109  
**NULLIF() expression (NULLIF()表达式)**, 161, 166–167, 176  
**nulls (空值)**  
     aggregate functions and (聚合函数), 176  
     allowing in foreign-key column (外键列允许), 41  
     characteristics of (特点), 84–85  
     checking for with COALESCE() expression (COALESCE()检查表达式), 165, 170  
     comparing (比较), 102  
     counting rows including (统计行包括), 178  
     eliminating in tables (在表中删除), 85  
     empty strings vs. (清空串和), 84, 85  
     forbidding in columns (列中禁止), 343–345  
     joins and (联结), 199  
     OR operator with (OR操作符), 107  
     outer joins listing results with (外联结列出结果), 239, 240, 244  
     returning for equivalent expressions (等价表达式), 167  
     sorting and (排序和), 98  
     subqueries and (子查询和), 268–269  
     testing for in SELECT statements (SELECT语句测试), 124–126  
     UNIQUE columns and (唯一列), 359  
**numbers (数字)**  
     changing sign of (改变标记), 130  
     formatting in specific databases (在特定的数据库格式化), 90

## O

**object references (对象引用)**, 37  
**ON DELETE clause (ON DELETE子句)**, 357, 358  
**ON UPDATE clause (ON UPDATE子句)**, 357, 358  
 one-to-many relationships (一对多联系), 43  
 one-to-one relationships (一对一联系) 42, 56  
**operands (域)** 127  
**operators and functions (操作符和函数)**, 127–167 另见  
*specific operators and functions*  
**about (关于)**, 127  
**arithmetic operations (算术操作符)**, 130–132  
**built-in SQL functions for statistics (内置SQL统计函数)**, 177  
**checking for nulls (检查空值)**, 165  
**comparing expressions (比较表达式)**, 166–167  
**comparison operators (比较操作符)**, 101  
**concatenating strings (连接字符串)**, 134–136  
**converting data types (转换数据类型)**, 157–160  
**creating derived columns with (创建导出列)**, 128–129  
**evaluating conditional values with CASE (使用CASE判断条件值)**, 161–164  
**operators and functions (操作符与函数)**  
     extracting and finding substrings (选取和发现子串), 137–139, 149–151  
     finding string lengths (确定串长度), 147–148  
     operator evaluation order (操作符优先顺序), 133  
     overloading (重载) 152  
     performing datetime and interval (执行日期和时间及间隔)  
         arithmetic (算术的), 152–153  
         printing current user (打印当前用户), 156  
         retrieving current date and time (检索当前日期和时间), 154–155  
         trimming characters (修整字符串), 142–146  
         wildcard operators (通配符), 114, 116  
         working with dates (使用日期), 445–450  
**options for command-line tool (命令行工具选项)**  
     db2, 26  
     mysql, 29  
     osql, 13  
     psql, 32  
     sqlcmd, 15  
     sqlplus, 19  
**OR operator (OR操作符)**  
     combining with NOT and AND operators (组合NOT和AND操作符), 109  
     using (使用), 105, 107  
**Oracle** 另见 DBMS-specific SQL features

- about (关于), 17  
 displaying table definitions for (显示表结构), 317  
 dropping indexes (删除索引), 383, 384  
 empty strings and nulls in (清空串和空值), 85, 126  
 limiting number of returned rows (限制返回行的数量), 425  
 retrieving metadata (检索元数据), 439, 441  
 running SQL with (运行SQL), 2–3  
**sqlplus** command-line tool for (**sqlplus**命令行工具), 18–19  
 versions of (版本), 17  
 working with dates in (使用日期), 447  
**Oracle Express Edition** (Oracle Express版本), 17  
**ORDER BY** clause (**ORDER BY**子句)  
 execution sequences with (执行顺序), 201  
 limiting number of rows returned (限制返回行的数量), 421–429  
 restricted use in UNION operations (UNION操作运用限制), 304  
 sorting rows with (行排序), 95–97, 99–100  
 speed of sorting and (排序速度), 98  
 in statements with GROUP BY clause (带有GROUP BY子句的语句), 184 186, 188  
**osql** command-line tool (**osql**命令行工具), 11, 12–13  
**OUTER JOIN** clause (**OUTER JOIN**子句), 235–246  
 combining with INNER JOIN (与INNER JOIN结合), 245–246  
 comparing results of INNER JOIN and (INNER JOIN比较结果), 238–241  
 creating left, right, and full outer joins (创建左、右、全外联结), 236–237, 239–241, 243  
**INNER JOIN** vs., 235  
 returning single values with subqueries (返回唯一值的子查询), 276  
 testing and listing nulls (测试和列出空值), 244  
 types of (类型), 198, 235  
 using subqueries vs. (使用子查询和), 259  
**outer queries** (外查询),  
 about (关于), 254  
 correlated subqueries and (相关子查询), 262, 265  
 overloading operators and functions (重载操作符和函数), 152
- P**
- parent tables (父表), 353, 354  
 parent-child relationships (父-子关系), 458, 459  
 parentheses (), 63, 106  
 partial functional dependency (部分函数依赖) 47–48  
 paths and pathnames (路径与路径名), 3  
 percent sign (%) operator (百分比操作符), 114  
 performance (表现)  
 combining conditions for query (查询的组合条件), 111  
 creating fastest comparisons (创建最快的比较), 103  
 queries ranking data (数据查询排行), 431  
 simple subqueries and (简单查询), 266  
 speed of sorting (排序速度), 98  
 tuning statements for (语句调优), 302  
**POSITION()** function (**POSITION()**函数), 149–151  
**PostgreSQL**, 另见DBMS-specific SQL features . 20–26  
 about (关于), 30  
 conformance to SQL standards (SQL标准一致性), 65  
 displaying table definitions for (显示表结构), 318  
 dropping indexes (删除索引), 383, 384  
 limiting number of returned rows (限制返回行的数量), 428–429  
**psql** command-line tool for, (**psql**命令行工具), 30–32  
 retrieving metadata (检索元数据), 439, 444  
 running SQL with (运行SQL), 2–3  
 working with dates in (使用日期), 450  
**precedence rules** (优先法则)  
 arithmetic operators (算数操作符), 133  
 combining logical operators in compound conditions (复合条件中的逻辑操作符的组合), 109  
 INTERSECT with EXCEPT and UNION operations (带有EXCEPT和UNION操作符的INTERSECT), 311  
**PRIMARY KEY** constraint (主键约束)  
 about (关于), 339, 350  
 can't be used with nullable columns (不能用于可为空值的列), 343  
 composite (组合的), 352  
 simple (简单), 351  
 specifying for table (限定于表), 339, 350–352  
 UNIQUE vs., 359  
**primary keys**.另见PRIMARY KEY constraint  
 characteristics of (特点), 36, 38–39  
 joins and (联结), 198  
 many-to-many relationships and (多对多联系), 44  
 one-to-many relationships and (一对多联系), 43  
 one-to-one relationships and (一对一联系), 42  
 optional vs. mandatory (可选的和必须的), 62  
 self-joins and (自联结), 247  
 simple (简单), 351  
 using in tables (在表中使用), 41  
**programming language** (编程语言)  
 making common errors in (犯常见错误), 63  
 projections (投影), 90  
 properties, closure (属性, 封闭), 36  
**psql** command-line tool, (**psql**命令行工具), 30–32

existing (存在), 31  
 interactive mode for (互动模式), 30  
 options for (可选), 32  
 script mode for (脚本模式), 31  
 publishers (出版社)  
     author/publisher queries using UNION operations (对作者/出版社使用合并操作), 304–307  
 authors in same city/state as (在同一城市/州的作者), 213  
 authors living in different city from (在不同城市的作者), 297, 312  
 authors living in same city as (在同一城市的作者) 296, 310  
 books by author and listed by (按作者列出图书), 223–224  
 calculating royalty totals by (计算总稿酬), 230–232  
 inner join listing cities of authors and (内联结列出作者所在城市), 238  
 listing book types published by more than one (列出不止一个出版社出版的类型), 286, 287  
 listing name and ID of books and (列出图书的ID和名称), 212  
 merging titles of two (合并两个书名), 330  
 outer joins listing all results with nulls (外联结列出所有带空值的结果), 240  
 with sales exceeding overall average (超过总平均数量), 279  
 subqueries listing by genre (按类型列出的子查询), 254–255, 295  
 total royalties paid authors for all books (支付给作者的总稿酬), 233–234  
 publishers table (表publishers), 51, 53, 324–325, 370  
 publishers2 table (表publishers2), 370

## Q

qualifying column names in tables (限定列名), 194–195  
     using in correlated subqueries (在相关子查询中使用), 267, 272  
 queries. 另见 subqueries  
     aggregate functions in (聚合函数), 169  
     calculating arithmetic mean (计算算数中值), 175–176  
     calculating sum of set (求总计), 174  
     combining conditions for performance (组合条件的效率), 111  
     counting rows (统计行), 178  
     defining sequence generators (定义顺序操作符), 409–414  
     displaying list of existing (显示存在列表), 9  
     execution sequences for joins (联结的执行顺序), 201  
     finding rows with extreme values (寻找包含极值的行), 453

finding trimmed mean (寻找修整中值), 432  
 functions for dates in (日期函数), 445–450  
 GROUP BY, 186, 188  
 handling duplicate rows (处理重复的行), 435–437  
 hierarchies and (层次), 458–464  
 limiting number of rows returned (限制返回行的数量), 421–429  
 MAX(), 173  
 median calculations with (计算中值), 451–452  
 MIN(), 172  
 nesting subqueries in (嵌套子查询), 284  
 performance with denormalization (反规范化), 50  
 picking random rows (随机选择行), 433–434  
 pivoting tables with (旋转表), 456–457  
 ranking data (排名数据), 431  
 retrieving metadata (检索元数据), 439–444  
 running statistics calculations with (移动统计计算), 406–408  
 sorting results of expressions (排序表达式的结果), 100  
 timing and measuring execution of (执行计时和计量), 302  
 types defining updateable views (定义可修改视图), 394  
 UNION for combining resulting rows (UNION合并结果行), 304–309  
     using with views (使用视图), 385, 391–393  
 Query Design (Access) (查询设计器), 8  
 quoted identifiers (带引号的标识符), 66, 67

## R

randomization (随机化), 433–434  
 range filtering (过滤范围), 118–120  
 ranking data (排名数据), 430–431  
 RDBMS. 见 DBMSs  
 recovering data (恢复数据), 401  
 re-creating and repopulating tables (再创建和再填充表), 374–375  
 re-expressing conditions (再表达条件), 113  
 referenced tables (引用的表), 353  
 referential integrity (引用完整性), 353, 354  
 reflexive relationships (反身联系), 247, 248  
 regions (界限), 415–420  
 relational model (关系模型), 33–60  
     columns (列), 34, 35  
     denormalization (反规范化), 50  
     many-to-many (多对多), 44  
     natural joins in SQL standard vs. (SQL标准中的自然联结), 209  
     normalization (规范化), 45–50  
     one-to-many (一对多), 43

one-to-one relationships (一对一联系), 42  
 primary keys (主键), 36, 38–39, 41  
 rows (行) 34, 36  
 set theory and (集合理论), 33  
 SQL differences from (SQL与……不同), 61  
 tables (表), 34  
 relationships (联系)  
   defined (定义), 193  
   many-to-many (多对多), 44, 55  
   one-to-many (一对多), 43  
   one-to-one (一对一), 42, 56  
   reflexive (反身的), 247, 248  
 relative pathnames (相对路径名), 3  
 remote computers (远程计算机), 13, 16  
 renaming tables (重命名表), 375  
 repeating groups (重复组), 46  
 restoring data (恢复收据), 401  
 restrictions (限制), 104  
 retrieving metadata (检索元数据), 439–444  
 right outer joins (右外联结)  
   creating (创建), 236  
   defined (定义) 198, 235  
   including all rows in table regardless of column match (不论列是否匹配包括表中所有行), 240  
 rolling back transactions (回滚事务), 400, 403, 404  
 row subqueries (行子查询), 256  
 rows (行)  
   about (关于), 34, 36–37  
   adding (增加), 315, 319–326  
   applying aggregate functions to (应用聚合函数), 169  
   calculating differences between successive (计算相邻差异), 408  
   changing values of (改变值), 315, 327–332  
   combining (组合), 304–309  
   counting (计算), 178  
   deleting through view (通过视图删除), 397  
   duplicate (重复), 93–94, 298–299, 435–437  
   filtering (过滤), 101–104, 115, 190–192  
   finding common (寻找共同), 310–311  
   grouping (分组), 183–189  
   having highest and lowest values (有最大和最小值), 453  
   inserting through view (通过视图插入), 395  
   limiting number returned (限制返回数量), 421–429  
   listing with outer joins (使用外联结列出), 235, 239, 240, 241  
   locating different (定位不同的), 312–313  
   matching patterns with LIKE conditions (LIKE条件的匹配模式), 114–117  
   nulls inserted into (插入空值), 345

orphan (孤儿), 354  
 picking random (随机选取), 433–434  
 removing (删除), 315, 333–336  
 skipping table (跨过表), 428, 429  
 sorting (排序), 95–97, 99–100  
 unordered (无序的), 35  
 updating (更新), 396  
 royalties (稿酬, 版税)  
   calculating total book (计算所有书) 225  
   earned by author and book (按作者和书得到的), 226–229  
   listing advances by genre (按类型列出预付款), 216–217  
   total paid by publisher for all author's books (出版社支付给所有作者所写书的合计), 233–234  
 using inner join to calculate total (内联结计算合计), 230–232  
 royalties table (表royalties), 56, 334  
 running statistics (移动统计), 406–408  
 runs (递增), 415–420

## S

SAS, 4  
 scalar aggregates (标量聚合), 188  
 scalar subqueries (标量子查询), 256  
 schemas (概要), 37  
 scope of identifier (标识符范围), 66  
 script files (脚本文件), 2  
 script mode (脚本模式)  
   db2 command-line tool (db2命令行工具), 25  
   defined (定义), 2  
   mysql command-line tool (mysql命令行工具), 28  
   osql command-line tool (osql命令行工具), 12  
   psql command-line tool (psql命令行工具), 31  
   sqlcmd command-line tool (sqlcmd命令行工具), 15  
   sqlplus command-line tool (sqlplus命令行工具), 18  
 search conditions (搜索条件), 101  
 searched CASE expression (CASE表达式搜索), 163–164  
 second normal form (2NF) (第二范式), 45, 47–48  
 security of table views (表视图的安全性), 386  
 SELECT clause (SELECT子句)  
   GROUP BY expressions in (GROUP BY表达式), 184  
   results of DISTINCT() in aggregate functions and (DISTINCT()聚合函数的结果), 180, 181–182  
   retrieving columns with (检索列), 88–90  
   using aggregate functions in (使用聚合函数), 171  
   using CREATE TABLE AS with (使用CREATE TABLE AS), 369  
   using SELECT \* with EXISTS subqueries (使用带EXISTS子查询的SELECT \*), 299–300  
 SELECT INTO statement (SELECT INTO语句), 371  
 SELECT statement (SELECT语句)

- ALL keyword in (ALL关键字), 94  
 combining and negating conditions in (结合和反条件), 105–109, 111–112  
 creating column aliases (创建列别名), 91–92  
 debugging WHERE clauses (调试WHERE子句), 110  
 deleting duplicate rows (删除重复的行), 93–94  
 displaying (显示), 9  
 filtering rows (过滤行), 101–104  
 list filtering (过滤列表), 121–123  
 matching patterns with LIKE (LIKE匹配模式), 114–117  
 occurrence order in UNION operations (合并操作的顺序), 308  
 range filtering in (过滤范围), 118–120  
 set operations with (集合操作), 303  
 simple subquery processing in (简单子查询处理), 262–263  
 sorting rows (排序行), 95–97, 99–100  
 syntax for (语法), 87  
 testing for nulls in (测试空值), 124–126  
 views as (视图), 385  
 selecting *n*th row (选择第*n*行), 434  
**self-joins** (自联结)  
 creating (创建), 248  
 defined (定义), 198, 247  
 listing parent-child relationships (列出父-子关系), 459, 460  
 using (使用), 247–252  
 writing subqueries as (写子查询), 260  
**self-referencing tables** (自引用表), 40  
**semicolon (;)** (分号), 62, 63  
**sequence tables** (顺序表), 411–412  
**sequences** (顺序)  
 defining sequence generators (定义序列发生器), 409–414  
 detecting breaks in (检查中断), 417  
 finding start/end positions and length of (发现开始/结束位置和长度), 416  
 runs, regions, and (递增、等值和), 415–420  
**sets** (集合), 304–313  
 about (关于), 33  
 combining rows (结合行), 304–309  
 determining maximum value (确定最大值), 170, 173  
 finding different rows (发现不同的行), 312–313  
 finding minimum value (发现最小值), 172  
 locating common rows (定位共同的行), 310–311  
 queries calculating sum of (汇总查询), 174  
 summing distinct values (不同值求和), 179–182  
 testing membership for (测试成员资格), 281–287  
**Show Table dialog** (Show Table对话框), 8  
**simple primary keys** (简单主键), 351  
**simple subqueries** (简单子查询)
- about (关于), 262–263  
 correlated vs. (相关的和), 266  
**single quotes (')** (单引号), 70, 71  
**slash (/)** (斜杠), 3  
**sorting** (排序)  
 based on conditional logic (基于条件逻辑), 99  
 collation and (核对), 96  
 nulls and (空值和), 98  
 rows with ORDER BY clause (ORDER BY子句), 95–97, 99–100  
 speed of (速度), 98  
**Soundex**, 437  
**Spreadsheets** (电子表格), 37  
**SQL**. 另见 further reading; SQL statements  
 approximate numeric types (近似数字类型), 75  
 based on relational model (基于关系型模型), 61  
 BLOB data types (二进制大对象数据类型), 72  
 books on advanced (预付款的书), 405, 458  
 Boolean types (布尔类型), 76  
 character string types (字符串类型), 70–71  
 common programming errors in (常见语法错误), 63  
 comparison operators (比较操作符), 101  
 creating legal aggregate functions (创建合法聚合函数), 171  
 data types (日期类型), 68–69  
 datetime types (日期和时间类型), 77–79  
 exact numeric types (精确数据类型), 73–74  
 identifiers (标识符), 62, 66–67  
 interval types (时间间隔型), 80–81  
 nulls (空值), 84–85  
 other data types in (别的数据类型), 83  
 performance tuning for statements (语句调优), 302  
 programs for (程序), 2–3  
 standards for (标准), 65  
 syntax of (语法), 62–64  
 types of expressions (表达式类型), 64  
 unique identifiers (唯一标识符), 82  
 user-defined types (用户定义类型), 83  
 working with statistics in (使用统计), 177  
**SQL Query Analyzer**, 10, 11  
**SQL server**. 另见 Microsoft SQL Server  
**SQL Server Express Edition**, 10, 14  
**SQL Server Management Studio**, 14–16  
**SQL Server Management Studio Express**, 10, 14  
**SQL Server Studio Query Editor**, about, 10  
**SQL statements** (SQL语句) 另见 *specific clauses*  
 clauses of (子句), 62, 63, 64  
 defined (定义), 62  
 syntax of (语法), 62–64  
 transactions (事务), 399–404

## T

- SQL View (Access) (SQL视图), 8  
 sqlcmd command-line tool (sqlcmd命令行工具), 10, 15  
 SQLite, 4  
 sqlplus command-line tool (sqlplus命令行工具), 18, 19  
 standards (标准)  
   natural joins in relational model vs. (关系模型中的自然联结), 209  
 statistics in SQL (SQL中的统计)  
   built-in SQL functions for (SQL内置函数), 177  
   changing in midstream (中流的改变), 454–455  
   using running statistics (使用移动统计), 406–408  
 string comparisons (串比较), 275  
 strings. 参见 character strings  
 subqueries (子查询) 253–302  
   categories of (目录), 256  
   comparing subquery values (比较子查询的值), 275–280  
   comparing values with ALL (带有ALL比较值), 288–290  
   comparisons using ANY (使用ANY比较), 291–293  
   correlated (相关的), 262, 263–266  
   defined (定义), 253  
   existence tests with (存在测试), 294–300  
   inner and outer (内部和外部), 254  
   joins vs. (联结和), 257–261  
   nesting (嵌套), 284, 329  
   nulls in (空值), 268–269  
   qualifying column names in (限定列名), 267  
   simple (简单), 262–263, 266  
   syntax for (语法), 256  
   terminology for (术语), 254, 255  
   testing set membership with IN (使用IN检测集合成员), 281–287  
   types of (类型), 262  
   using as column expression (列表达式), 270–274  
   using equivalent (使用等价的子查询), 301–302  
 substrings (子串)  
   extracting with SUBSTRING(), (使用SUBSTRING()取子串), 137–139  
   finding with POSITION() (使用POSITION()定位), 149–151  
   sorting data by (排序数据), 97  
 subtraction operator (−) (减号操作符), 130, 131–132  
 SUM() aggregate function (SUM()聚合函数), 170, 174  
 Sybase, 4  
 syntax语法  
   ANSI-89 vs. ANSI-92 mode (Access) (ANSI-89和ANSI-92模式), 5  
   SQL statement (SQL语句), 62–64  
 system catalog (系统目录), 36  
 system tables (系统表), 36, 440
- table aliases (表别名), 196–197  
 table definitions (表结构), 316–318  
 table scan (表扫描), 382  
 table subqueries (表的子查询), 256  
 tables 另见 joins; normalization; temporary tables; views;  
   and specific tables  
   about (关于), 34  
   adding rows with INSERT (使用INSERT添加行), 315, 319–326  
   altering (修改), 373–375  
   base (基础), 366  
   calendar (日历), 414  
   changing values in existing rows (改变既有行的值), 327–332  
   check constraints for (检查约束), 339, 363–365  
   column aliases (列别名), 91–92  
   constraining (约束), 339–340  
   creating sequence of consecutive integers (创建连续整数序列), 411  
   creating table aliases (创建表别名), 196–197  
   DEFAULT column values specified for (指定默认列值), 346–349  
   deleting duplicate rows (删除重复行), 93–94  
   dropping (删除) 337, 366, 376  
   dropping views (删除视图), 398  
   eliminating nulls in (删除空值), 85  
   empty (清空), 34  
   filtering rows with WHERE clause (使用WHERE子句筛选行), 101–104  
   forbidding nulls in columns (在列中禁止空值), 343–345  
   foreign key constraints in (外键约束), 353–358  
   foreign keys (外键), 40–41  
   fully and partially functionally dependent (全部或部分依赖), 47–48  
   inserting rows from one table to another (从别的表插入行), 323–326  
   joining three or more (联结3个或更多), 210  
   joining two tables on column (联结两个表), 211–212  
   junction (连接), 44  
   making new tables from existing (从既有表创建新表), 369–372  
   many-to-many relationships between (多对多联系), 44, 55  
   matching row patterns in (匹配行模式), 114–117  
   modifying data with SQL statements (使用SQL语句修改数据), 315  
   one-to-many relationships between (一对多联系), 43

one-to-one relationships between (一对一面联系), 42  
 pivoting (旋转), 456–457  
 populating with INSERT statement (使用INSERT语句填充), 342  
 primary key for (主键), 350–352  
 qualified column names in (限定列名), 194–195  
 reason for indexing (索引的原因), 377  
 re-creating and repopulating (再创建和再填充), 374–375  
 renaming (重命名), 375  
 selecting columns from (选择列), 88–90  
 self-referencing (自引用), 40  
 sequence (顺序), 411–412  
 skipping rows with LIMIT clause (使用LIMIT子句跨过行), 428, 429  
 specifying in outer joins (在外联结中限定), 235  
 spreadsheets vs. (电子表格和), 37  
 statements creating (创建语句), 337, 338, 341–342  
 temporary (临时的), 366–368  
 transitive dependency of (可传递的依赖性), 49  
 truncating (删除), 336  
 user and system (用户和系统), 36  
 using primary keys in (使用主键), 38–39, 41  
 temporary tables (临时表), 366–368  
 defined (定义), 366  
 global (全局), 367, 370  
 local (局部), 367  
 views vs. (视图) 389  
 temps table (表temps), 416–418  
 Teradata, 4  
 testing (检测)  
 before inserting rows in table (在表中插入行前), 325  
 existence for subqueries (对于子查询的存在性), 294–300  
 for nulls (空值), 124–126  
 set membership with IN (使用IN检测集合成员), 281–287  
 SQL code against standards (相对于标准的SQL代码), 65  
 theta joins, (theta 联结), 198  
 third normal form (3NF) (第三范式), 45, 49  
 three-value logic (3VL) (三值逻辑), 105  
 timestamps (时间戳), 78  
 title\_authors table (表title\_authors), 55, 335, 341  
 title\_name table (表title\_name), 360  
 titles2 table (表titles2), 370  
 titles table (表titles), 54, 330, 341, 347, 364, 388–389  
 tools suppressing data variation (删除数据变型的工具), 437  
 transaction logs (事务日志), 400, 401  
 transactions (事务), 399–404  
 ACID acronym (首字母缩写), 403  
 committing (提交) 335, 400, 404

implicit or explicit starts to (显式或隐式开始), 402  
 rolling back (回滚), 400, 403, 404  
 transitive dependency (传递的依赖性), 49  
 TRIM() function (TRIM()函数), 135, 142–146  
 trimmed mean (修整中值), 432  
 TRUNCATE statement (TRUNCATE语句), 336  
 two-value logic (2VL) (两值逻辑), 105  
 typographic conventions (排版约定), 87

**U**

unary table operations (一元表操作), 36  
 underscore (\_) operator (下划线操作符), 114  
 unicode, 71  
 union joins (union联结), 237  
 UNION operations (UNION操作)  
 combining rows with (结合行), 305–309  
 function of (函数), 303  
 order of SELECT statements in (SELECT语句顺序), 308  
 precedence of INTERSECT with (在INTERSECT之前), 311  
 restrictions on (限制), 304  
 updateable views with (可更新视图), 394  
 UNIQUE constraints (唯一约束), 359–362  
 unique indexes (唯一索引)  
 unique constraints vs. (唯一约束和) 362  
 using (使用), 379, 381  
 uniqueness tests (唯一测试), 298  
 UPDATE statement (UPDATE语句)  
 changing values in rows with (改变行的值), 315, 327–332  
 limiting number of rows affected by (限制影响到行的数量), 421  
 performing and rolling back (执行与回滚), 402, 403  
 updating tables (修改表)  
 avoiding anomalies with normalization (规范化避免异常), 45  
 using views for (使用视图), 386, 394, 396  
 UPPER() function (UPPER()函数), 140–141  
 user tables (用户表), 36  
 USING clause (USING子句)  
 joins with (联结), 203  
 replicating natural joins with (改写自然联结), 208  
 UUID (Universally Unique Identifier (全局唯一标识符)), 82

**V**

Values (值)  
 calculating sum of set (计算集合总计), 174, 179–182  
 changing in table with UPDATE (使用UPDATE更新表), 327–332  
 comparing subquery (比较子查询), 275–280

defined (定义), 34  
 finding rows with highest and lowest (发现最大和最小行), 453  
 joining unequal column (联结不等的列), 220  
 maximum set of (最大集合), 170, 173  
 minimum set of (最小集合), 172  
 subqueries comparing with ALL (带有ALL的比较子查询), 288–290  
 testing for duplicates with EXISTS (使用EXISTS测试重复值), 298–299  
 vector aggregates (向量聚合), 188  
 Venn diagram (韦恩图), 33  
 viewing path (视图路径), 3  
 views (视图), 385–398  
     considerations before creating (创建前的考虑), 387  
     creating (创建), 386–390  
     data updates through (数据修改), 394  
     defined (定义), 385  
     deleting row through (删除行), 397  
     dropping (删除), 398  
     inserting row through (插入行), 395  
     retrieving data through (检索数据), 391–393  
     temporary tables vs. (临时表和), 389  
     updating rows through (更新行), 396

**W**

WHERE clause (WHERE子句) aggregate expressions and (聚合表达式), 171  
 comparing equivalent subqueries using (比较等价的子查询), 301–302

comparing subquery values using (比较子查询的值), 276–278  
 comparing values in subqueries with ALL (带有ALL的子查询的值比较), 289  
 creating joins with (创建联结), 200–202  
 debugging (调试), 110  
 execution sequences with (执行顺序), 201  
 filtering rows with (过滤行), 101–104  
 joining tables using (联结表), 210–220  
 list filtering with IN condition (使用IN条件筛选), 121–123  
 misspecifying or omitting for DELETE (错误或忽略删除), 333  
 in query containing GROUP BY clause (包含GROUP BY子句), 184, 186, 188  
 replicating natural joins with (改写自然联结), 207  
 sequence with GROUP BY and HAVING clause (GROUP BY和HAVING子句的顺序), 190  
 sorting authors by specific location (按位置排序作者), 249  
 specifying for UPDATE statement (UPDATE语句的限定), 327  
 testing subqueries with EXISTS (使用EXISTS子查询测试), 294–300  
 using for joins (联结) 201  
 using with column aliases (使用列别名), 104  
 WHERE conditions (WHERE条件), 214, 216  
 wildcard operators (通配符)  
     listing of (列表), 114  
     matching (匹配), 116



书号：978-7-115-19256-1

## SQL Server 编程必知必会

- 《SQL必知必会》作者新作
- Amazon全五星评价
- T-SQL学习与使用必备图书

### 内容简介

现今 Microsoft SQL Server 是已经成为世界上应用最广受欢迎的数据库管理系统之一。

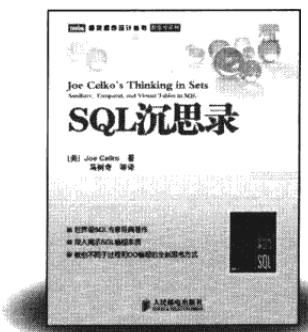
本书是作者继经典畅销书《SQL 必知必会》之后，作者应众多读者的请求编写的，专门针对 SQL Server T-SQL 用户。书中继承了《SQL 必知必会》的优点，没有过多在精炼然而透彻的阐述了数据库基础理论之后，而是紧贴实战需要，直接很快转向从数据检索开始，逐步深入各种复杂的内容，包括联结的使用、子查询、基于全文本的搜索、函数和存储过程、游标、触发器、表约束、XML，等等。对于每个知识点，都给出了实用的代码及其解析，并有丰富的技巧和常犯错误警示。通过本书，读者能够掌握扎实的基本功，迅速成为 SQL Server 编程高手。

作者为本书专门开设了网站：<http://www.forta.com/books/0672328674/>，提供下载、勘误和答疑论坛。

### 评论

盼望这本书已久已经很久了……虽然我是一个经验丰富的数据库管理员和数据库开发人员，但这本书仍使我受益匪浅。

——Pinal Dave，微软 SQL – MVP，SQLAuthority.com 的创办人者



书号：978-7-115-21395-2

## SQL沉思录

- 世界级SQL专家经典著作
- 深入揭示SQL编程本质
- 教你如何以数据集的方式思考

### 内容简介

SQL 是数据库的标准数据查询语言，在广大程序员的日常工作中已必不可少。但是，大部分的 SQL 程序员都是由过程式编程人员和面向对象编程人员转变而来的，他们已经习惯了原有的思维方式，若要充分利用 SQL，必须学会以数据集的方式思考。

本书中，世界级 SQL 专家 Joe Celko 通过展示实用技术及思想方法，教给大家如何完成这个转变。书中通过大量实例详细介绍了各种 SQL 编程技术，内容涉及查找表、辅助表、虚拟表、时态表、视图、SQL 的思考方式等多个方面，值得所有 SQL 程序员仔细研读和思考。

### 读者评论

Joe Celko 写的所有的书我都买了，这本书是其中写得最好的一本……强烈推荐！

——Amazon.com 读者评论