/THEORY/IN/PRACTICE

The Art of SQL

SQL语言艺术

Stéphane Faroult &
Peter Robson 著
温昱 靳向阳 译

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

O'REILLY®

Database/SQL

“一部SQL力作。对于在其他书中止步于理论的难点，本书提供了实际的经验技巧，效果卓然。”

——Sean Hull，资深咨询顾问，Heavyweight互联网集团

“一本真正‘聪明’的SQL书，注重实践而非课堂上的理论，解决数据库专业人员遇到的问题。”

——Anthony Molinaro，首席数据库工程师，Wireless Generation公司

你是否把数据库应用看作一场战役？

你是否把数据看作一排排需要筛选并打退的敌军纵队？

SQL专家Stéphane Faroult就是这么做的。

《SQL语言艺术》的作者Stéphane Faroult借用《孙子兵法》的智慧处理 SQL性能方面的问题。开发数据库应用好比一场战役，设计即战略，而每次数据库访问就是与敌军在战术上的对决。

Stéphane对SQL性能问题的论述，视角独特，一丝不苟。如何才能更好地使用SQL？本书结合一线实践的案例，强调了：

应从最开始就综合设计数据库和应用的性能。“只强调某个最重要方面的设计是错误的”。

应避免只考虑单个SQL语句。“关注总的目标，SQL语句再好也挽救不了糟糕的处理流程”。

必须明白为哪些字段建立索引、为何建立索引。“索引过多是设计不确定的表现”。

Stéphane还对你的实际工作提出了具体建议。九种SQL经典查询方案，以及对其性能影响的讨论，非常便于实践。书中有一章专门讨论树状结构，可帮助你解决层次结构数据的问题。另外，当你必须基于别人的数据库设计进行工作时，“精于计谋：挽救响应时间”这一章会助你成功。

Stéphane Faroult经营着RoughSea公司，从事数据库咨询业务，致力于帮助客户从数据库投资中获得最佳性能。他的SQL经验开始于1983年。Oracle法国的第一个性能及调优课程就是他1987年编写的。

译者简介：温昱 资深咨询顾问，CSAI特聘高级顾问，软件架构专家，软件架构思想的传播者和积极推动者。十年系统规划、架构设计和研发管理经验，在金融、航空、多媒体、网络管理、中间件平台等领域负责和参与多个大型系统的规划、设计、开发与管理。著有《软件架构设计》，译著有《应用框架的设计与实现——.NET平台》等。

图书分类：数据库

责任编辑：梁晶

O'REILLY

www.oreilly.com

Broadview®
WWW.BROADVIEW.COM.CN

www.phei.com.cn

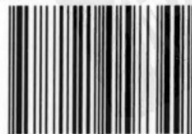
网上订购：www.dearbook.com.cn
第二书店 · 第一服务

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

O'Reilly Media, Inc.授权电子工业出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-121-05834-9



9 787121 058349 >

定价：58.00元

O'REILLY®

SQL 语言艺术

The Art of SQL

[美] Stéphane Faroult,Peter Robson 著
温 昱 靳向阳 译

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内 容 简 介

本书分为 12 章，每一章包含许多原则或准则，并通过举例的方式对原则进行解释说明。这些例子大多来自于实际案例，对九种 SQL 经典查询场景以及其性能影响讨论，非常便于实践，为你的实际工作提出了具体建议。本书适合 SQL 数据库开发者、软件架构师，也适合 DBA，尤其是数据库应用维护人员阅读。

0596008945 The Art of SQL© 2006 by O'Reilly Media, Inc.
Simplified Chinese edition, jointly published by O'Reilly Media，Inc. and Publishing House of Electronics Industry, 2008.Authorized translation of the English edition, 2006 O'Reilly Media Inc., the owner of all rights to publish and sell the same.
All rights reserved including the rights of reproduction in whole or in part in any form..

本书中文简体版专有出版权由 O'Reilly Media, Inc.授予电子工业出版社，未经许可，不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号　图字：01-2006-5602

图书在版编目（CIP）数据

SQL 语言艺术 /（美）法禾（Faroult,S.），（美）罗伯森（Robson,P.）著；温昱，靳向阳译.—北京：电子工业出版社，2008.3
书名原文:The Art of SQL
ISBN 978-7-121-05834-9

I. S… Ⅱ.①法…②罗…③温…④靳… Ⅲ.关系数据库—数据库管理系统 Ⅳ.TP311.138

中国版本图书馆 CIP 数据核字（2008）第 009405 号

责任编辑：梁　晶
印　　刷：北京市天竺颖华印刷厂
装　　订：三河市金马印装有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×980　1/16　印张：23.5　字数：450 千字
印　　次：2008 年 3 月第 1 次印刷
定　　价：58.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。
质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。
服务热线：（010）88258888。

O’Reilly Media, Inc.介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构 O’Reilly Media, Inc. 授权电子工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O’Reilly Media, Inc. 是世界上在UNIX、X、Internet和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的《The Whole Internet User’s Guide & Catalog》(被纽约公共图书馆评为20世纪最重要的50本书之一）到GNN（最早的Internet 门户和商业网站)，再到 WebSite(第一个桌面 PC 的Web服务器软件)，O’Reilly Media, Inc. 一直处于Internet发展的最前沿。

许多书店的反馈表明，O’Reilly Media, Inc. 是最稳定的计算机图书出版商 —— 每一本书都一版再版。与大多数计算机图书出版商相比，O’Reilly Media, Inc. 具有深厚的计算机专业背景，这使得O’Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O’Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O’Reilly Media, Inc. 还有许多固定的作者群体 —— 他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O’Reilly Media, Inc. 依靠他们及时地推出图书。因为 O’Reilly Media, Inc. 紧密地与计算机业界联系着，所以O’Reilly Media, Inc. 知道市场上真正需要什么图书。

计利以听，乃为之势，以佐其外。

While heeding the profit of my counsel,avail yourself also of
any helpful circumstances over and beyond the ordinary rules.

——*Sun Tzu,The Art of War*

The French humorist Alphonse Allais(1854-1905),once dedicated one of his short storied as follows:

To the only woman I love and who knows it well.

...with the following footnote:

This is a very convenient dedication that I cannot recommend too warmly to my fellow writers.It costs nothing,and can,all at once,please five or six persons.

I can take a piece of wise advice when I meet one.

Stéphane Faroult

前言

过去，“信息技术（IT）”的名字还不如今天这般耀眼，被称为“电子数据处理”。其实，尽管当今新潮技术层出不穷，数据处理依然处于我们系统的核心地位，而且需管理的数据量的增长速度似乎比处理器的增长速度还快。今天，最重要的集团数据都被保存在数据库中，通过 SQL 语言来访问。SQL 语言虽有缺点，但非常流行，它从 1980 年代早期开始被广泛接受，随后就所向无敌了。

如今，年轻开发者在接受面试时，没有谁不宣称自己能熟练应用 SQL 的。SQL 作为数据库访问语言，已成为任何基础 IT 课程的必备部分。开发者宣传自己熟练掌握 SQL，其实前提是“熟练掌握”的定义是“能够获得功能上正确的结果”。然而，全世界的企业如今都面临数据量的爆炸式增长，所以仅做到“功能正确”是不够的，还必须足够快，所以数据库性能成了许多公司头疼的问题。有趣的是，尽管每个人都认可性能问题源自代码，但普遍接受的事实则是开发者的首要关注点应该是功能正确。人们认为：为了便于维护，代码中的数据库访问部分应该尽量简单；“拙劣的 SQL”应该交给资深的 DBA 去摆弄，他们还会调整几个“有魔力”的数据库参数，于是速度就快了——如果数据库还不够快，似乎就该升级硬件了。

往往就是这样，那些所谓的“常识”和“可靠方法”最终却是极端有害的。先写低效的代码、后由专家调优，这种做法实际上是自找麻烦。本书认为，首先要关注性能的就是开发者，而且 SQL 问题绝不仅仅只包含正确编写几个查询这么简单。开发者角度看到的性能问题和 DBA 从调优角度看到的大相径庭。对 DBA 而言，他尽量从现有的硬件（如处理器和存储子系统）和特定版本的 DBMS 获得最高性能，他可能有些 SQL 技能并能调优一个性能极差的 SQL 语句。但对开发者而言，他编写的代码可能要运行 5 到 10 年，这些代码将经历一代代的硬件，以及 DBMS 各种重要版本升级（例如支持互联网访问、支持网格，不一而足）。所以，代码必须从一开始就快速、健全。很多开发者仅仅是“知道”SQL 而已，他们没有深刻理解 SQL 及关系理论，实在令人遗憾。

为何写作本书

SQL 书主要分为三种类型：讲授具体 SQL 方言的逻辑和语法的书、讲授高级技术及解决问题方法的书、专家与资深 DBA 所需的性能和调优的书。一方面，书籍要讲述如何写 SQL 代码；另一方面，要讲如何诊断和修改拙劣的 SQL 代码。在本书中，我不再为新手从头讲解如何写出优秀的 SQL 代码，而是以超越单个 SQL 语句的方式看待 SQL 代码，无疑这更加重要。

教授语言使用就够难了，那么本书是怎样讲述如何高效使用 SQL 语言的呢？SQL 的简单性具有欺骗性，它能支持的情况组合的数目几乎是无限的。最初，我觉得 SQL 和国际象棋很相似，后来，我悟到发明国际象棋是为了教授战争之道。于是，每当出现 SQL 性能难题的时候，我都自然而然地将之视为要和一行行数据组成的军队作战。最终，我找到了向开发者传授如何有效使用数据库的方法，这就像教军官如何指挥战争。知识、技能、天赋缺一不可。天赋不能传授，只能培养。从写就了《孙子兵法》的孙子到如今的将军，绝大多数战略家都相信这一点，于是他们尽量以简单的格言或规则的方式表达沙场经验，并希望这样能指导真实的战争。我将这种方法用于战争之外的许多领域，本书借鉴了孙子兵法的方法和书的题目。许多知名 IT 专家冠以科学家称号，而我认为“艺术”比“科学”更能反映 IT 活动所需的才能、经验和创造力（注 1）。很可能是由于我偏爱“艺术”的原因，“科学”派并不赞成我的观点，他们声称每个 SQL 问题都可通过严格分析和参考丰富的经验数据来解决。然而，我不认为这两种观点有什么不一致。明确的科学方法有助于摆脱单个具体问题的限制，毕竟，SQL 开发必须考虑数据的变化，其中有很大的不确定性。某些表的数据量出乎意料地增长将会如何？同时，用户数量也倍增了，又将会如何？希望数据在线保存好几年将会如何？如此一来，运行在硬件之上的这些程序的行为是否会完全不同？架构级的选择是在赌未来，当然需要明确可靠的理论知识——但这是进一步运用艺术的先决条件。第一次世界大战联军总司令 Ferdinand Foch，在 1900 年 French Ecole Supérieure de Guerre 的一次演讲中说：

> 战争的艺术和其他艺术一样，有它的历史和原则——否则，就不能成其为艺术。

本书不是 cookbook，不会列出一串问题然后给出“处方”。本书的目标重在帮助开发者（和他们的经理）提出犀利的问题。阅读和理解了本书之后，你并不是永不再写出丑陋缓慢的查询了——有时这是必须的——但希望你是故意而为之、且有充足的理由。

注 1：我最喜欢的计算机书，恰好是 D.E. Knuth 的经典著作 *Art of Computer Programming*（Addison Wesley 出版社）。

目标读者

本书的目标读者是：

- 有丰富经验的 SQL 数据库开发者
- 他们的经理
- 数据库占重要地位的系统的软件架构师

我希望一些 DBA、尤其是数据库应用维护人员也能喜欢本书。不过，他们不是本书的主要目标读者。

本书假定

本书假定你已精通 SQL 语言。这里所说的“精通”不是指在你大学里学了 SQL 101 并拿来 A+的成绩，当然也并非指你是国际公认的 SQL 专家，而是指你必须具有使用 SQL 开发数据库应用的经验、必须考虑索引、必须不把 5000 行的表当大表。本书的目标不是讲解连接、外连接、索引的基础知识，阅读本书过程中，如果你觉得某个 SQL 结构还显神秘,并影响了整段代码的理解，可先阅读几本其他 SQL 书。另外，我假定读者至少熟悉一种编程语言,并了解计算机程序设计的基本原则。性能已很差、用户已抱怨、你已在解决性能问题的前线，这就是本书的假定。

本书内容

我发现 SQL 和战争如此相像，以至于我几乎沿用了《孙子兵法》的大纲,并保持了大部分题目名称（注 2）。本书分为 12 章，每一章包含许多原则或准则，并通过举例的方式对原则进行解释说明，这些例子大多来自于实际案例。

第 1 章，制定计划：为性能而设计

讨论如何设计高性能数据库

注 2：另一些题目名称我借鉴了 Clausewitz 的 *On War*。

第 2 章，发动战争：高效访问数据库

解释如何进行程序设计才能高效访问数据库

第 3 章，战术部署：建立索引

揭示为何建立索引，如何建立索引

第 4 章，机动灵活：思考 SQL 语句

解释如何设计 SQL 语句

第 5 章，了如指掌：理解物理实现

揭示物理实现如何影响性能

第 6 章，锦囊妙计：认识经典 SQL 模式

包括经典的 SQL 模式、以及如何处理

第 7 章，变换战术：处理层次结构

说明如何处理层次数据

第 8 章，孰优孰劣：认识困难，处理困难

指出如何认识和处理比较棘手的情况

第 9 章，多条战线：处理并发

讲解如何处理并发

第 10 章，集中兵力：应付大数据量

讲解如何应付大数据量

第 11 章，精于计谋：挽救响应时间

分享一些技巧，以挽救设计糟糕的数据库的性能

第 12 章，明察秋毫：监控性能

收尾，解释如何定义和监控性能

本书约定

本书使用了如下印刷惯例：

`等宽（Courier）`

　　表示 SQL 及编程语言的关键字，或表示 table、索引或字段的名称，抑或表示函数、代码及命令输出。

等宽黑体（Courier）

　　表示必须由用户逐字键入的命令等文本。此风格仅用于同时包含输入、输出的代码示例。

等宽斜体（Courier）

　　表示这些文本，应该被用户提供的值代替。

总结：箴言，概括重要的 SQL 原则。

注意

提示、建议、一般性注解。为相关主题提供有用的附加信息。

代码示例

本书是为了帮助你完成工作的。总的来说，你可以将本书的代码用于你的程序和文档，但是，若要大规模复制代码，则必须联系 O'Reilly 申请授权。例如：编程当中用了本书的几段代码，无需授权；但出售或分发 O'Reilly 书籍中案例的 CD-ROM 光盘，需要授权。再如：回答问题时，引用了本书或其中的代码示例，无需授权；但在你的产品文档中大量使用本书代码，需要授权。

O'Reilly 公司感谢但不强制归属声明。归属声明通常包括书名、作者、出版商、ISBN。例如“The Art of SQL by Stéphane Faroult with Peter Robson. Copyright © 2006 O'Reilly Media, 0-596-00894-5”。

如果你对代码示例的使用超出了上述范围，请通过 *permissions@oreilly.com* 联系出版商。

评论与提问

我们已尽力核验本书所提供的信息，尽管如此，仍不能保证本书完全没有瑕疵，而网络世界的变化之快，也使得本书永不过时的保证成为不可能。如果读者发现本书内容上的错误，不管是赘字、错字、语意不清，甚至是技术错误，我们都竭诚虚心接受读者指教。如果您有任何问题，请按照以下的联系方式与我们联系。

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the U.S. or Canada)
(707) 829-0515 (international or local)
(707) 829-0104 (fax)

奥莱理软件（北京）有限公司
北京市 海淀区 知春路 49 号 希格玛公寓 B 座 809 室
邮政编码：100080
网页：*http://www.oreilly.com.cn*
E-mail：*info@mail.oreilly.com.cn*

北京博文视点资讯有限公司（武汉分部）
湖北省 武汉市 洪山区 吴家湾 邮科院路特 1 号 湖北信息产业科技大厦 1402 室
邮政编码：430074
电话：(027)87690813　　传真：(027) 87690595
读者服务网页：*http://bv.csdn.net*
E-mail：
reader@broadview.com.cn（读者信箱）
bvtougao@gmail.com（投稿信箱）

出版商为本书建立了网页，来提供勘误表、案例、其他附加信息，网址如下：

http://www.oreilly.com/catalog/artofsql（原书）

http://www.oreilly.com.cn/book.php?bn=978-7-121-05834-9（中文版）

评论本书或讨论本书相关的技术问题，可发邮件至：

bookquestions@oreilly.com

欲了解更多 O'Reilly 书籍、会议、资源中心、网络等信息，请访问 O'Reilly 网站：

http://www.oreilly.com

本书作者公司的网站为：

http://www.roughsea.com

致谢

本书原版用英语写作，英语既不是我的家乡话，又不是我所在国家的语言，所以写这样一本书要求极度乐观（回想起来几近疯狂）。幸运的是，Peter Robson 不仅为本书贡献了他在 SQL 和数据库设计方面的知识，也贡献了持续的热情来修改我冗长的句子、调整副词位置、斟酌替换词汇。Peter Robson 和我在好几个大会上都碰过面，我们都是演讲者。

Jonathan Gennick 担任本书编辑，这有点让人受宠若惊，Jonathan Gennick 是 O'Reilly 出版的 *SQL Pocket Guide* 等畅销名著的作者。Jonathan 是个非常尊重作者的编辑。由于他的专业、他对细节的关注、他的犀利视角，使本书的质量大大提升。同时，Jonathan 也使本书的语言更具“中大西洋”风味（Peter 和我发现，虽然我们保证按美国英语拼写，但还远远不够）。

我还要感谢很多人，他们来自三个不同的大陆，阅读了本书全部或部分草稿并坦诚地提出意见。他们是：Philippe Bertolino、Rachel Carmichael、Sunil CS、Larry Elkins、Tim Gorman、Jean-Paul Martin、Sanjay Mishra、Anthony Molinaro、Tiong Soo Hua。我特别感激 Larry，因为本书的思想最初来自于我们的 E-mail 讨论。

我也要感谢 O'Reilly 的许多人，他们使本书得以出版。他们是：Marcia Friedman、Rob Romano、Jamie Peppard、Mike Kohnke、Ron Bilodeau、Jessamyn Read、Andrew Savikas。感谢 Nancy Reinhardt 卓越的手稿编辑工作。

特别感谢 Yann-Arzel Durelle-Marc 慷慨提供第 12 章用到的图片。感谢 Paul McWhorter

授权我们将他的战争图用于第6章。

最后，感谢 Roger Manser 和 Steel Business Briefing 的职员，他们为 Peter 和我提供了位于伦敦的办公室（还有大量咖啡）。感谢 Qian Lena (Ashley)提供了本书开始引用的《孙子兵法》的中文原文。

目录

Contents

前言 ······ I

1 制定计划：为性能而设计 ······ 1

数据的关系视图 ······ 2
规范化的重要性 ······ 4
有值、无值、空值 ······ 11
限用 Boolean 型字段 ······ 14
理解子类型（Subtype） ······ 15
约束应明确声明 ······ 17
过于灵活的危险性 ······ 18
历史数据的难题 ······ 19
设计与性能 ······ 21
处理流程 ······ 22
数据集中化（Centralizing） ······ 23
系统复杂性 ······ 24
小结 ······ 25

2 发动战争：高效访问数据库 ······ 27

查询的识别 ······ 28
保持数据库连接稳定 ······ 29
战略优先于战术 ······ 31
先定义问题，再解决问题 ······ 32
保持数据库 Schema 稳定 ······ 33
直接操作实际数据 ······ 34
用 SQL 处理集合 ······ 34
动作丰富的 SQL 语句 ······ 35
充分利用每次数据库访问 ······ 36
接近 DBMS 核心 ······ 37
只做必须做的 ······ 41

SQL 语句反映业务逻辑 …… 42
把逻辑放到查询中 …… 42
一次完成多个更新 …… 43
慎用自定义函数 …… 44
简洁的 SQL …… 46
SQL 的进攻式编程 …… 48
精明地使用异常（Exceptions） …… 50

3 战术部署：建立索引 …… 55
找到“切入点” …… 56
索引与目录 …… 59
让索引发挥作用 …… 60
函数和类型转换对索引的影响 …… 62
索引与外键 …… 67
同一字段，多个索引 …… 69
系统生成键 …… 70
索引访问的不同特点 …… 72

4 机动灵活：思考 SQL 语句 …… 75
SQL 的本质 …… 76
掌握 SQL 艺术的五大要素 …… 84
过滤 …… 89

5 了如指掌：理解物理实现 …… 105
物理结构的类型 …… 106
冲突的目标 …… 108
把索引当成数据仓库 …… 109
记录强制排序 …… 113
数据自动分组（Grouping） …… 115
分区是双刃剑 …… 119
分区与数据分布 …… 120
数据分区的最佳方法 …… 121
预连接表 …… 123
神圣的简单性 …… 124

6 锦囊妙计：认识经典 SQL 模式 127

小结果集，直接条件 129

小结果集，间接条件 137

多个宽泛条件的交集 138

多个间接宽泛条件的交集 140

大结果集 146

基于一个表的自连接 147

通过聚合获得结果集 150

基于日期的简单搜索或范围搜索 156

结果集和别的数据存在与否有关 161

7 变换战术：处理层次结构 167

小结果集，直接条件 129

小结果集，间接条件 137

多个宽泛条件的交集 138

多个间接宽泛条件的交集 140

大结果集 146

基于一个表的自连接 147

通过聚合获得结果集 150

基于日期的简单搜索或范围搜索 156

结果集和别的数据存在与否有关 161

8 孰优孰劣：认识困难，处理困难 199

看似高效的查询条件 200

抽象层 202

分布式系统 205

动态定义的搜索条件 208

9 多条战线：处理并发 225

数据库引擎作为服务提供者 226

并发修改数据 231

10 集中兵力：应付大数据量 247

增长的数据量 248

数据仓库 264

11 精于计谋：挽救响应时间 ······ 279

数据的行列转换 ······ 280

基于变量列表的查询 ······ 294

基于范围的聚合 ······ 297

一般规则，最后使用 ······ 299

查询与列表中多个项目相符的记录 ······ 301

最佳匹配查询 ······ 304

优化器指令 ······ 305

12 明察秋毫：监控性能 ······ 307

数据库速度缓慢 ······ 308

服务器负载因素 ······ 310

何谓“性能优良” ······ 311

从业务任务角度思考 ······ 317

执行计划 ······ 319

合理运用执行计划 ······ 328

总结：影响性能的重要因素 ······ 330

Photo Credits ······ 333

索引 ······ 335

第 1 章

制定计划：为性能而设计

Laying Plans: Designing Databases for Performance

在所有战争中，显示军事才华的第一步是战略规划。

——Joseph de Maistre（1754—1821）

19 世纪伟大的德国战略家克劳塞维茨（Clausewitz）有句名言：战争是政治的延续。同样，对于一个组织而言，计算机程序是其综合活动的延续，目的是多（more）、快（faster）、好（better）、省（cheaper）地运营。计算机程序不是从数据库中取出数据加以处理这么简单，而是应为特定目标处理数据——勿将手段与目标混淆。

计算机程序的首要目标是：满足业务需求（business requirement）（注 1）。这也许是老生常谈了，但实际上，人们经常在遇到技术挑战的刺激时不知不觉地忽视目标、而重视手段，忽视记录商业活动的数据的质量、而仅重视按期交付能完成预期功能的程序。我们必须像指挥军队的将军一样，明确目标、坚持不懈、即便改变计划也不放弃目标。涉及 SQL 语言时，我们必须确保数据忠实地、一致地反映商业活动，这关系到数据库模型的质量。设计 SQL 的初衷就是要支持关系模型，所以怎么强调良好的模型和合理的数据库设计都不为过，毕竟这是任何信息系统的基础。

数据的关系视图
The Relational View of Data

数据库只是一小部分现实状况的模型而已。和任何其他描述一样，数据库也不是精确模型，而仅是对纷繁复杂的现实世界的有限描述。对特定业务活动的描述往往不止一种方法，这些方法从技术上讲都是语义正确的，然而，其中通常只有一种描述方法是最符合业务需求的。

之所以称为“关系模型”，不是因为不同表之间建立了“关系”（这是很常见的误解），而是因为表内不同字段之间存在“关系”。也就是说，如果有几个值属于表的同一行，它们之间就存在关系。关系就是表（更确切地说，表描述了关系），不同字段联系在一起的方式定义了关系。

要在多大范围（scope）内为现实状况建模，这是由业务需求决定的。一旦明确了这个范围，就可以确定记载业务活动需要哪些数据。假设，你是个二手车经销商，要为待售

注 1：业务需求应包括经营活动和非经营活动。

的汽车建模——例如你要在网站上登出出售二手车的广告，那么品牌、型号、版本、款式（轿车、双门小轿车、敞篷车……）、出产年份，总里程数、以及价格等信息会首先被想到。但潜在的买家可能还想了解更多信息，例如：

- 车况（谁都希望车况良好）
- 安全设备
- 手动档还是自动档
- 车身颜色、车内颜色、是否金属漆、内饰、软顶车还是硬顶车、车的照片
- 座位数、后备箱大小、几开门
- 动力方向盘、空调、音响
- 发动机排气量、汽缸数、马力和最高时速、刹车（只有狂热的汽车爱好者单凭汽车型号就能知道这些技术规格）
- 燃料、油耗、油箱容量
- 该车目前所在位置（如果网站提供来自不同地方的二手车，买家则会关心这项信息）
- ……

如果把待售二手车的信息存入数据库，那么表的每一行都是对事实的具体描述，例如，“有一部待售二手车，1964 年产，粉红色，凯迪拉克双门小轿车 DeVille，行驶里程相当于绕地球二十圈”。

可以通过多种方法获得新的“对事实的描述”：例如通过 Join 等关系操作，再例如通过条件过滤、属性选择，或者对属性进行特定计算等（根据油箱容量和油耗来计算还可行驶多少公里就是一个例子）。只要原来的描述是正确的，那么获得的新描述就是正确的。

我们处理知识的依据是“被接受为真理而不需证明的事实”，数学中称它们为“公理（axiom）”，而其他学科称它们为“原理（principle）”。推衍出的新事实（在数学中就是证明定理）本身又成为进一步推衍的基础。

关系数据库的工作方式完全相同。显然，关系模型是以数学为基础的。我们定义的关系（再次强调，关系就是我们创建的表）代表我们接受的事实，而视图（View）和查询就是新事实。

注意

> 关系模型的一致性（coherence）是个必须掌握的重要概念。关系模型的数学理论基础具有固有的稳定性，所以只要遵守关系理论，就可保证基于数据库的任何查询结果与原始数据具有同样的有效性。关系理论的关键原理是：关系不包含重复数据，且记录之间没有顺序。正如在第 4 章中将讲述的，SQL 的使用非常灵活，拥有强大的查询能力，但使用不当会导致查询优化失败。

然而，我们可以自由选择用于推衍新事实的基本事实，选择不当则非常糟糕。例如，如果每次有人要买苹果时，杂货商都必须证明所有牛顿物理学说，岂不是太麻烦？同样，如果程序大部分基本操作都需要 25 路关联（25-way join），岂不是自寻烦恼？

一个组织所使用的数据，可能和它的供应商、客户所使用的数据是一样的。然而，只要不是直接竞争者，他们的立足点就不同，他们看待相同数据的角度也不一样。例如，即使都在使用相同的数据，但我们的业务需求和我们的供应商及客户的业务需求不同。没有哪个设计是放之四海而皆准的，但基于高质量的设计便于编写查询。

总结：建模是业务需求（business requirement）具体化的过程。

规范化的重要性

The Importance of Being Normal

规范化（Normalization）——尤其是满足第三范式（third normal form ，3NF）的规范化，是关系理论课程的一部分。如同许多课本知识一样，规范化被认为枯燥、沉闷、且与实际脱节（我想到了古典文学）。直到多年以后，我们有了经验，才认识到规范化的重要性，同时也领悟到原理和古典主义的精髓都是永恒的。

规范化的原理是：按照严格的逻辑要求，将不同的数据项组织在一起，使它们成为结构化的信息。这种严格性是通过不同级别的范式（normal form）来界定的。尽管纯化论者

（purist）认为对数据的分析应该超越 3NF，满足 BCNF（Boyce-Codd 范式）（注 2）甚至 5NF，但最具代表性的范式是 1NF、2NF 和 3NF。别担心，本书只会讨论前三个范式，在绝大多数的情况下，符合 3NF 的数据库也符合 BCNF 和 5NF。

规范化为何如此重要？答案在于：规范化使混沌变得有序。一场战役之后，我们犯下的错误变得显而易见，而成功行动则像“本应如此”的举动。同样，数据库中的表经过规范化之后，结构非常自然，所以有人认为规范化只是“常识”的美称。但我们真的有足够丰富的常识吗？我们处理复杂数据时不会手忙脚乱吗？而基于严格逻辑规则的三个范式，能为我们清晰思考提供清单式指南（checklist）。

表结构没有规范化就会面临天大的风险吗？通常不会（个人观点，未经证实）。其实真正的风险是数据的不一致性、难于编写的数据输入控制代码、拙劣的性能、庞大应用中存在的错误，以及模型无法演进（evolve）等问题。但没有规范化会使上述风险出现的几率大大提高，本书稍后将讨论其中的原因。

那么，如何将一些松散的、不同类型的信息组织在一起，成为便于运用的数据模型呢？方法本身并不复杂，我们只须遵循一定的步骤，下面结合案例说明之。

第 1 步：确保原子性（Atomicity）

Step 1: Ensure Atomicity

首先，必须确保要处理的特征（characteristic）或属性（attribute）具有原子性。原子性看似简单，但其完整概念相当难懂。原子（atom）一词源自公元前五世纪的希腊哲学家 Leucippus，是指“无法分割的东西”（其实，原子可以裂变）。判断数据是否具有原子性有个尺度问题：例如对将军来说，一个团就是具有原子性的战斗单位；但指挥那个团的上校要负责更小一级的营或连，所以团对他来说不具有原子性。同样，对于汽车经销商来说，一辆车是具有原子性的信息单元；但对于汽车修理厂的技工来说，一辆汽车根本不具有原子性，而是由更低一级的大堆零件组成的，这些零件才是他们眼里具有原子性的数据项。

注 2：满足 3NF 但不满足 BCNF 是指表有多组候选键，但这多组候选键共享了某字段。这种情况不太常见。

纯粹从实用角度而言，应该把原子属性定义为表中的字段，以便在 where 子句中充分利用它。尽管在 select 选出项列表（属性被返回到这里）中，可以对一个属性进行进一步的拆解，但如果在 where 子句中必须引用某属性的一部分，则说明这个属性没有达到所需的原子性级别。例如，二手车的“安全配备”包含下列几项信息：防锁死刹车系统（ABS）、安全气囊（乘客专用气囊、乘客驾驶员共用气囊、前方气囊、侧向气囊等）、集中式锁控等。若定义一个名为 `safety_equipment` 的字段来描述所有这些安全特性，则会丧失两个主要优点。

高效搜索能力

> 经常在湿滑路面上行驶的驾车者认为 ABS 非常重要，他们会以“ABS”为条件进行搜索（search），但由于需要判断每一行数据的 `safety_equipment` 字段是否包含“ABS”子串，因此搜索非常慢。正如将在第 3 章中要讲述的，常规索引（regular index）应以具有原子性的值作为键。有时，可以采用其他手段（例如建立全文索引）加速查询，但又会带来不能实时更新等缺点。另外，全文搜索的结果可能包含错误。例如，如果 color 字段同时包含了车身和内饰颜色的描述，而你因为欲购买蓝色车子而搜寻“blue”，则搜索结果中也会包含内饰为蓝色的灰色车子。相信在进行 Web 搜索时，大家都经历过得到毫不相干结果的事情吧。

由数据库保证的数据正确性

> 数据输入很容易出错。比漫长的搜索时间更糟的是：如果输入到描述性字符串中的是“ASB”而不是“ABS”，那么 DBMS 是无法发现此错误的，结果，查询不到需要的数据。于是，有些查询的结果将是错误的（例如统计配备 ABS 的汽车的数量时，结果要么不准确要么根本就是错的）。此时，为了保证数据的正确性，除了要求输入者在打字时仔细检查之外，只能编写复杂的函数检查 safety_equipment 字符串输入或更新的正确性——但这样做同样糟不可言，检查函数很难维护，且开销增大性能下降。相比而言，定义取值非“Y”即“N”的 has_ABS 字段虽不能保证“Y”和“N”不会被搞反，但至少可以让 DBMS 拒绝其他值。

除了上述两个主要优点的丧失，必须精通很多字符串函数也是一个问题。所以，应避免将多个值塞进同一字符串中。

定义数据原子单元有时并不简单。例如，处理地址时就常常出现原子性的问题。必须把地址定义成大的、不可分的字符串吗？相反，必须把地址拆分成多个部分吗？还有，地址要拆分到什么程度呢？请回忆先前关于原子性和业务需求的一些结论，“如何描述地址”实际上取决于“要如何使用地址”。如果需要利用地址进行统计、或是根据邮政编码和所在城镇进行搜索，那么从地址中拆分出相应属性即可。接下来的问题是，地址的分解应该进行到什么程度？

原则上，为了决定将地址拆分到什么程度，必须考虑业务需求，因为业务需求决定了原子级的地址属性。拆分不能凭空预测，照搬其他组织采用的地址格式也很危险，除非我们根据自己的业务需求分析了其适用性。

注意，细节之中潜藏着危险。过分“精益求精”会使我们精力分散，甚至关注不相干的问题。例如，如果将地址拆分为街道、门牌号等原子数据项，那么遇到“ACME 大厦”这样的地址描述，我们就无法处理。不要人为地制造问题，因为有些信息无需处理。合理把握处理数据的层次非常重要，要突破数据操作在决策支持层面考虑问题。

一旦确定了所有原子数据项，且数据项间的相互联系也已明确，那么清晰的关系就呈现出来。下一步是确定能唯一标识各记录的主键（primary key）。在当前这个阶段，主键很可能是复合的，由两个或更多属性组成。二手车的品牌、型号、版本、款式、出产年份、总里程数合在一起构成了主键，客户根据它来区分二手车，而不是根据车牌号。正确定义键并非易事，经典的“客户（customer）”的例子能很好地说明这个问题。用“客户名”作为“客户”的主键不是最佳选择，“客户”可能是一家公司，则名称是“RSI”、“Relational Software”，还是“Relational Software Inc”呢？“Relational Software”之后有没有逗号呢？“Inc”之后有没有点号呢？名称是大写还是小写呢？首字母要不要大写呢？看来以客户名称作为主键值得商榷，为了避免混淆必须有严格的命名标准。其实，

比较可取的方式是以标准的简化名称或唯一代码作为主键，同时留意公司名变更（例如由“Relational Software Inc.”变更为“Oracle Corporation”）对其他数据的影响；如果必须保留业务关系的历史，还必须能识别出两个名称代表“不同时期的同一家公司”。

一般而言，应尽量使用具有实际意义的主键，而不是晦涩的递增整数。我必须强调，主键应能够刻画出数据的特色，而分配给新记录的序列号不具此能力。“注册地+注册号”的方式使用起来比较麻烦，专门增加一个序列号 `company_id` 用起来很方便，这个序列号作为主键时其实是真正键的技术代用品（或速记符），和“使用表的别名简化查询”具有一样的效果：

```
where a.id =  b.id
```

取代：

```
where table_with_a_long_name.id = table_even_worse_than_the_other.id
```

记住，仅有技术上的序列号，还不意味着有了真正的主键，千万不要把它误认为实际的主键。一旦所有属性都具有原子性、且确定了键，我们的数据就符合 1NF 了。

第 2 步：检查对键的完全依赖性
Step 2: Check Dependence on the Whole Key

前例中，表 used_cars 保存的某些信息可供二手车买主参考，但这些信息对汽车发烧友来说是多余的。事实上，许多特性也并不是某部车特有的，例如，一旦品牌、型号、版本、款式确定了，车的座位数和载货量也就确定了，根本不必考虑复合键中的其他两个属性——出产年份和总里程数。换言之，表 `used_cars` 有些属性只部分依赖于键，这会造成什么影响呢？

数据冗余

> 如果有许多品牌、型号、版本及款式都相同的车待售（这组特性可统称为“车型”），则这些公共属性会被重复保存，重复的次数与同车型的车辆数相同。数据冗余有两个问题：第一，冗余的数据容易引起信息不一致（修改它们也更费时）；第二，造成了存储空间浪费。如今存储设备很便宜，看似不必太在意空间问题，但其实不然：首先，需要存储的实际数据量本身在增长；另外，数据常需要镜像；而且，数据可能要备份到灾难恢复站点的磁盘上，在那里需再次镜像；还有，生产数据

库会有许多个复本作为开发用数据库。所以，看似一个字节的冗余，通常也都是四五倍的浪费，浪费量非常惊人。除了上述存储空间的代价之外，数据冗余还会给数据恢复带来更严重的问题。当“系统非计划停机”时，唯一的解决方案就是根据备份来恢复数据库，而此时双倍的数据量意味着双倍的恢复时间，这在某些场合成本非常高——若是医院数据库，甚至会危及生命。

查询性能

扫描一个大数据量的表（记录数很多）要花更多时间。如本书所述，全表扫描（full table scan）在许多情况下是最佳方案，不值得像初学者那样“大惊小怪”。不过，是每行数据的字节数越多，此表所需的存储页就越多，扫描此表所花的时间就越长。如果表是非规范化的，要显示汽车款式的选择列表就必须对所有待售汽车数据进行 `select distinct` 操作；和将 `car_model` 表独立出来相比，这不仅意味着扫描更多条记录，还意味着必须对记录排序以剔除重复数据。通过数据的细化分割能让 DBMS 引擎只对数据子集进行操作，而性能也会高得多。

为了消除键的部分依赖性，必须建立新的表，此例中为表 `car_mode`。新表的键都是原始表的键的一部分，例如表 `car_model` 的键为品牌、型号、版本和款式。接下来，必须把依赖于新键的所有属性转移到新表 `car_model` 中，同时原先的表中保留品牌、型号、版本和款式这几个属性。由于发动机及其特性不依赖于车的款式，不必将它们转移到新表中。在去除了只依赖部分键的属性后，表就符合 2NF 了。

第 3 步：检查属性独立性

Step 3: Check Attribute Independence

在所有数据都满足 2NF 后，我们可以开始确认它是否满足 3NF。其实，满足 2NF 的数据集通常也满足 3NF，然而我们还是应该检查一下。当前，数据集的每个属性都已完全依赖唯一键（unique key），如果除了唯一键所包含的属性之外，不能根据任何其他属性确定一个属性的值，则这个数据集就满足 3NF。例如，我们必须检查：“属性 A 的值确定之后，属性 B 的值是否就确定了呢？”

国际联系信息（International contact information）是个绝佳的例子，说明一个属性可能依赖另一个非键属性（non-key attribute）。只要知道国家，就知道国际区号；反之则不然，因为美国和加拿大区号相同。当同时需要国家、国际区号这两项信息时，该怎么办呢？可以创建独立的 `country_info` 表，然后通过 ISO 国家代码（如 IT 表示意大利）作为外键，将联系信息表和 `country_info` 表关联；`country_info` 表以国家代码为主键，并记录业务所需的国家信息——例如“意大利，国际区号 39，货币为欧元”，诸如此类。为了确定是否存在依赖关系，必须对满足 2NF 的数据集中的每一对属性进行检查，最终才能确定该数据集是否满足 3NF。那么，不按照 3NF 的要求进行数据建模会有何风险呢？其实，基本上与不符合 2NF 的风险相同。

将数据规范为 3NF 非常重要。（注意，有时数据库设计者故意选择不遵守 3NF，第 10 章介绍的维度模型就是这样的例子。当然，在决定不按规则建模之前必须先了解规则，并权衡其中的风险。）3NF 重要的部分原因如下。

合理规范化的模型可应对需求变更

> 第 10 章会讲到，诸如维度模型等非规范化模型的前提是，维护和查询数据的方式非常特殊（第 5 章将讲述的数据的物理结构，也应根据维护和查询数据的特定方式来决定，数据的物理结构对程序性能影响很大，但不影响程序逻辑）。如果有一天，发现非规范化模型的前提是错误的，那就必须推倒重来。相反，3NF 模型有足够的灵活性以适应变化。

规范化使数据重复降至最少

> 如前所述，重复数据会造成磁盘空间和处理能力方面的浪费，而且还会增加数据不一致的可能性。当数据值的一个实例（instance）被修改，而保持在数据库中的另一个相同值未被同时（且同样）修改，就会出现数据不一致。信息丢失（Losing information）不仅指数据被删除；如果数据库的一部分说“白”、另一部分说“黑”，也是一种信息丢失。数据不一致的问题可由 DBMS 预防，前提是模型要给予支持——你是否为具有原子性的属性定义了字段约束（column constraints），你是否声明了参考完整性约束（referential integrity constraints）。否则，就只能通过编程来预防了：你若选择采用触发器或存储过程，不仅增加了复杂性，且开销巨大；你若选择修改程序的方式，会为程序带来不必要的复杂性，并因而增加了维护成本。

触发器和存储过程的编写必须非常小心，在程序中保证数据一致性的做法相当于将数据完整性（data integrity）的保护从数据库移到了应用层，于是，所有需要访问这些数据的程序都要重复保护工作，否则就会轻易破坏其他程序精心维护的数据一致性。

总结：规范化的重要基础是，模型必须具有原子性。

有值、无值、空值

To Be of Not to Be, or to Be Null

有个建模错误非常普遍，就是将大量特性都“塞”到一个表中，于是导致这个表有太多字段。尽管一些科学领域要求非常详细地描述研究对象会导致这种现象，但商业应用很少如此。无论如何，下述情况是数据库设计有缺陷的明确征兆：表的字段大多为空值（null），特别是某两个字段不可能同时有值（如果字段 A 有值则字段 B 一定为空值，反之亦然）。毫无疑问，这违背了 2NF 或 3NF 。

表中的每一条记录都应该是特定“事物”的状态描述，如果大部分特性信息都显示“我们不知道”，无疑大大降低了信息的可信程度。若是单纯以保存信息为目的，这样做的危害并不大，但要基于这些数据进行处理获得结果集，问题就严重了——这意味着模型有缺陷。即使业务流程各种数据的信息源和输入时间都不同，每条记录的所有字段最终也应该有值。集邮者有时也会为邮票预留位置，但无论如何，按照最大尺寸预留位置会浪费空间并严重影响性能。

存在空值意味着关系模型存在严重问题，动摇了查询优化的基础。关系模型的完备性（completeness）是以二值逻辑（two-valued logic）为基础的，记录要么存在、要么不存在。where 子句中的条件必须明确，但任何中间状态或空值都是不确定的。条件非“真”即“假”，从而决定记录返回与否，诸如“或许是的但不确定”这样的条件无法判断是

否返回记录。包含空值的模型属于三值逻辑（真、假、不确定），将它转换成结果集要求的二值逻辑非常危险，会造成看似正确的 SQL 查询返回错误结果集，因为遇到了空值。例如，color 字段包含 RED、GREEN、BLACK 等值，则：

```
where color not in ('BLUE', 'BLACK', null)
```

就不会返回任何结果，因为 SQL 引擎无法确定 null 代表什么——既可能代表 RED 也可能代表 GREEN。而：

```
where color in ('BLUE', 'BLACK', null)
```

只返回所有颜色为 BLACK 的记录（别忘了我们的表中不含 BLUE），因为 SQL 引擎认为 null 可能既不是 RED 也不是 GREEN。由此可见，SQL 引擎比银行家还不愿冒险。虽说 in() 列表中显式出现“null”并不常见，但其中出现“返回空值的子查询”却是可能的。

处理遗漏信息（missing information）具有内在的困难性，如何描述客户（customer）就是个好例子。每个客户都有地址，正常情况下该地址将出现在发货单上，但“客户地址”与“送货地址”不同时我们必须“另行输入送货地址”吗？但这又不是殡仪馆信息系统（客户只“光顾”一次），所以“另行输入送货地址”不满足业务要求，多次重复输入相同的地址不仅浪费了时间，还增加了出错风险，货物送错了地址会引起客户不满而失去客户。很明显，送货地址应该是客户的特性，而不应该是订单的特性，这个问题在模型设计初期做依赖性分析（analysis of dependencies）时就应该解决。

如果客户是一家公司，会计部门和办公部门可能在不同地点，送货地址会不同。所以，每个客户应该有一个办公地址、一个账单寄送地址，以及一个送货地址。于是，我们经常看到表 customer 中有三组字段，每组描述一个地址。

然而，最常见的情况不是这样。在 90% 的情况下，只需一个地址，即办公地址。那么其他两组地址字段怎么办？有两种不同的设计策略。

账单寄送地址和送货地址为 null

> 此策略不太合理，需要程序使用一些隐含规则，例如“如果账单地址为 null 则把货物送到办公地址”，这使程序逻辑更为复杂，程序代码中出现错误的风险也会增加。

复制地址信息，账单寄送地址没有指定则复制公司办公地址

> 此方法要求数据输入时进行特殊处理，可以通过触发器来完成。不过，有时数据复制的开销很大。此外，信息更新时也必须进行数据复制——每次公司地址更新时必须复制到相应字段，否则会造成信息不一致性。

上述两种设计策略都说明建模者对问题理解的严重不足。使用空值和隐含规则，是采用三值逻辑的典型表现。空值的使用必然导致三值逻辑，从而造成语义不一致，程序设计再高明也无法消除语义问题。而数据需要复制说明依赖性分析没有做到位。

解决地址难题的方案之一，是将地址信息从 `customer` 表中分离出来。例如，建立一个 `address` 表，它除了保存每个地址之外，还包含客户 ID 及一个“地址种类（role of the address）”字段（可能是屏蔽位）。这未必是最佳方案，因为到底如何使用地址的问题往往到程序仓促上线之后才会出现，而后续版本中难以改变模型。

到现在为止，我们一直假设每个客户只有一个送货地址，该地址与公司地址相同或不同。下面，再来考虑一种情况：同一张发票包含不同的送货要求，货物需要送到不同的分公司，但发票只送到某一个地址。这也很常见啊！`customer` 表只有一个（且常为空的）“送货地址”的设计行不通了。有讽刺意义的是，我们回到了“送货地址是订单特性”的老路上来，这意味着要引用订单中的地址（特别是多次引用）就必须通过特定 ID 与 `address` 表关联，该 `address` 表避免了每个订单中重复出现送货地址的问题（规范化起作用了）。或者，考虑引入 `shipments` 表。

对于客户/地址的难题，没有十全十美的设计。上面讨论了可能的问题和一些解决方案，

但适应具体情况的总是只有其中一种，而其他方案并不适合。若是选择了不合理的设计，不但程序代码更复杂，而且性能也会大打折扣。

空值问题或许是关系模型中最让人头痛的问题。关系模型之父 E.F. Codd 博士很早就引入了空值，并在 1985 年发表的“12 条规则”的第 3 条明确提出了空值的系统化处理方法。（“12 条规则”简明地定义了关系数据库必备的特性）。然而，争论仍在理论家之间激烈进行着，因为“不知道（not known）”可能包含许多种不同的情况：以著名作家清单为例，每个人都有出生日期和死亡日期，空的出生日期显然表示“生日未知”，但空的死亡日期代表“健在”、“卒年不详”、还是“生死不明”呢？

这不禁让我想起了美国前国防部长 Donald Rumsfeld 在 2002 年 2 月国防部新闻简报中的名言 ：

> 如我们所知，存在已知的已知。有些事情，我们知道自己知道。有些事情，我们知道自己不知道。也就是说，我们知道有些事情我们是不知道的。但也存在未知的未知，这些事，我们并不知道自己不知道。(As we know, there are known knowns. There are things we know we know. We also know there are known unknowns. That is to say we know there are some things we do not know. But there are also unknown unknowns, the ones we don't know we don't know.)

含有空值的模型经常出现，按 Donald Rumsfeld 的说法叫“已知的未知”，我们知道有些属性是存在的，但它的值我们现在还不知道。至于其他，推测无用。奇怪的是，空值最有趣的用法恰恰和“每行每列都有值的表”有关：通过外连接（outer join），有可能产生空值。第 6 章将讨论的“有效检查是否缺少特定值的技术”都是基于外连接和空值检查的。

总结：空值对程序逻辑是危险的；必须使用空值的话，一定要清楚它在特定情况下的影响。

限用 Boolean 型字段

Qualifying Boolean Columns

SQL 中并不存在 Boolean 类型，但很多人认为有必要实现标志位（flag）来表示 Boolean 型的真／假状态（例如表示订单是否完成的 `order_completed`）。我们应以提高“数据密度”为设计目标，`order_completed` 字段虽然有用，不过同时保存“订单何时完成”、

“订单由谁完成”这些信息比较好。于是，无论 completion_date 字段还是 completed_by 字段，都可以代替 completed 这个“Y/N”字段，而且所包含的信息量更大（当然，为了避免在订单未完成的情况下出现空值 null，我们可以使用不同的表来跟踪订单从建立到完成的各个阶段）。如前所述，我们还必须根据业务需求检查依赖性，仅当业务的成功操作需要时才引入这些额外字段。

另外，一组 Boolean 型属性有时可以组合成一个单独的状态属性。例如，如果我们有四个属性其取值都是“真/假”，则我们可以用 0 到 15 这 16 个数值代表四个属性的所有组合“状态”。但要注意，这个技巧可能违反原子性基本原则，所以应谨慎使用。

总结：为数据而数据，是通向灾难之路。

理解子类型（Subtype）

Understanding Subtypes

表过“宽”（有太多属性）的另一个原因，是对数据项之间的关系了解不够深入，子类型就是绝佳的例子。公司可能有不同类型的员工，既有固定工、又有合同工，他们有相同的特性（姓名、出生年份、部门、房间、电话号码等），但有些特性则为某类员工独有的（例如固定工的到职日期与薪水、合同工的费用与合同）。相同的属性可以共用，但独有的属性要确保分开，这就引出了子类型的话题。

在这种情况下，可以定义三个表。首先，employee 表包含了每个员工共有的属性信息，无论他们是固定工、合同工或其他工种。当然，必须有个属性说明每个员工的类型，每种员工类型对应一个值，例如“P”代表固定工（permanent employee）、“C”代表合同工（contract employee）。该表以员工号为主键。

接下来，为每种类型的员工建立一个表。在此例中为两个表：permanent 表和 contract 表。每个固定工或合同工，从 employee 表继承相应特性，同时还拥有在各自表中定义的独有特性。

下面看看这些表的主键是如何定义的，正是依靠主键才实现了“子类型”。对于上述 employee 、permanent 和 contract 三个表，主键都是员工号。 employee 表中所有主键的集合，是所有子类型表中主键的并集（union），而所有子类型表中主键的交集（intersection）必为空，因为本例中每个员工只能属于一种类型。子类型表的主键同时还是外键，它建立了与 employee 主键的参照关系。

注意，给子类型表指定完全独立于父表主键的主键，是极其错误的，现实中这种错误时有发生。还要注意，子类型不同于主从关系（master-detail relationship），从它们各自的主键就能很快区分出这一点。有人认为上述讨论过于理论化了（他们对“理论”一词多少有些贬低的意思），但恰恰相反，如果子类型主键不是父表主键的子集的话，在很多方面都会导致性能降低。

高效访问数据库的主要原则之一，应归功于亚历山大大帝之父、马其顿国王菲利浦二世的一个原理：分而治之（Divide and Rule）。例如，HR 部门用到的绝大部分查询方法有两种：一种是大致查询组织中的所有员工，另一种是查询某种人员的详细信息。只要正确利用子类型（注 3），上述两种查询都可以高效地查到期望的结果，而不会在无关的数据中浪费时间。相反，如果所有信息放在一个表中，最简单的查询也会触发大量数据的搜索，而其中大部分数据对查询结果并无用处。

总结：表中有些字段出现了空值，表明需要引入子类型。

注 3：子类型可能被误用。正如一个专家评论的那样，大量查询频繁引用父表不是高效的做法。如果父表保存着查询必需的信息，则所有查询都会来访问父表。子类型之间必须有逻辑差异性，千万不要受到 OO 技术的启发，而设计出继承性很强的 schema。

约束应明确声明

Stating the Obvious

数据中存在隐含约束是一种不良设计，例如“某种业务线（business line）ID 为数字，其他业务线 ID 为字符串”，或者“型号 T 的颜色必为黑色”。有时，这些常识对高效过滤数据很有用，但如果只有我们“人”知道而 DBMS 引擎不知道，依然不能帮助优化器（optimizer）实现最高效的数据库访问。最糟糕的情况是，隐含约束会导致运行时错误，例如不慎要求数据库引擎对字符串进行算术处理时——当定义了字符串型字段来存储数字数据，而其中不慎包含了非数字字符时，就会出现这样的错误。

顺便指出，若字符串型的 ID 有时包含字符有时包含数字，则说明最初数据库设计时对领域的定义比较混乱。显然，该字段的性质（nature）随环境变化而改变，这是合理数据库设计所不允许的。如果需要保存数值型、布尔型、字符型等各种类型的设置参数，不应使用单独一张表 configuration（parameter_name, parameter_value），而应使用更通用的表 configuration（parameter_id, parameter_name, parameter_type），并为每种参数类型建立子类型。例如数字型参数建立表 configuration_numeric ）（parameter_id, parameter_value），其中 parameter_value 是数字型字段；改变配置时 DBMS 会做检查，能及时发现“数字 0”写成“字母 O”等错误，避免了参数被使用时造成的运行时错误。

尽可能多地定义约束。当然，主键是必不可少的。备用键（alternate key）可以标志数据及各种唯一性约束。通过外键（Foreign key）可关联到主表（master table），从而保证数据一致性，所以外键是全面表达数据模型内涵不可或缺的。约束有助于控制被输入值的范围，也一样很有价值。约束有两个主要作用。

- 有助于保证数据完整性（integrity）。只要定义的规则是正确的，约束能保证所有数据都符合规则。
- 为 DBMS 核心——确切地说是为优化器，提供关于数据的重要信息。即使目前优化器没有充分利用所有可用的约束，但在未来的版本中，DBMS 核心很可能会将那些约束应用到更复杂的处理上。

回顾前面那个“多个送货地址和账单寄送地址”的例子，同样也是因设计不足而造成数

据库得不到足够的语义信息（semantic information），于是，这些信息就不得不出现在所有应用程序中。例如“如果账单寄送地址是 null 就使用总部地址”这条规则 DBMS 并不知道，所以必须由许多程序来处理——注意是“许多”程序！再次强调，数据库中的信息只需定义一次，所有程序就可以一致地使用。关于何种地址优先考虑的隐含规则，必须被编码到每个存取数据的程序中；这些隐含规则过于随意，账单寄送地址参照送货地址、而非总部地址，也不是不可能。

总结：数据语义属于 DBMS，别放在应用程序中。

过于灵活的危险性

The Dangers of Excess Flexibility

通常，将推理运用到极限（并且经常超越极限），会导致疯狂之举。第三方软件编辑器（third-party software editors）非常推崇“自认为更灵活（more-flexible-than-thou）”的思想，即将大多数据保存在为通用目的而设计的表中，表中包含诸如 `entity_id`、`attribute_id`、`attribute_value` 等通用属性。这种将所有信息当成字符串保存在 `attribute_value` 中的做法，的确避免了空值的出现，但同时也要求那些必填属性（mandatory attributes）保存在 `attribute_value` 中。顺便说一下，支持者认为这种设计便于随时增加新属性。对于这个“视增加属性为必然”的设计，我们姑且不谈它的的设计质量，假设它保存数据一切正常，但通常总需要读取并处理这些属于 `attribute_value` 字段的数据（不规划数据读取是严重的错误）；与通过编程进行信息处理相比，增加新字段其实并不那么重要（关于这一点，那些高估 XML 灵活性的人也必须明白）。

这种不切实际的灵活性，使数据库成本急剧升高。数据类型丧失殆尽，所以数据库完整性也完全丧失。同时，参照完整性（referential integrity）也完全丧失。事实上，也没有任何类型的声明式约束（declarative constraints）。最简单的查询也变成了怪异的 join，“值表（value table）”本身被 join 10 次、15 次，甚至 20 次（视要 select 的属性数量而定）；不言而喻，即使最聪明的优化器对这样的查询也不知所措，查询性能自然令人失望。（如

何优化这种查询的性能请参考第 11 章，但 SQL 语句有点乱。）其实，军事史上最差的战役，也看似战略计划的杰作。

总结：真正的设计灵活性来自合理的数据建模。

历史数据的难题

The Difficulties of Historical Data

我们经常需要处理历史数据——例如商品或服务在某时间点的价格，都是以历史数据为基础的。关系设计的真正难题，正是如何处理某一时间段相关的数据（而不是某一时点的数据）。

为历史数据建模有几种方法。假设要连续记录货物的价格变化，键为 article_id。一种显而易见的建模方法为：

```
(article_id, effective_from_date, price)
```

其中 effective_from_date 代表新价格生效的日期，该表的主键为（article_id, effective_from_date）。

该模型虽然逻辑正确，但在获取当前价格时比较笨拙，而当前价格在大多情况下恰恰是我们的主要关注点。为了确定当前价格，需要找到与最近的 effective_from_date 关联在一起的价格值，查询语句如下：

```
select a.article_name, h.price
from articles a,
     price_history h
where a.article_name = some_name
  and h.article_id = a.article_id
  and h.effective_from_date =
     (select max(b.effective_from_date)
      from price_history b
      where b.article_id = h.article_id)
```

执行这个查询会两次扫描 price_history 表：一次在查询语句内层，目的是为了确定特定商品的最新定价日期；另一次在查询语句外层，根据内层查询返回的那一条记录找到当前价格（第 6 章将讲述的某些 DBMS 提供的特殊函数，可以在某种程度上避免多次扫描）。可以证明，不断执行这种模式的查询代价很高。

当然，可以采取其他方式记录“价格有效期”。既然可以保存价格生效的开始日期，为什么不能保存价格有效的“终止日期”呢？看似不错的解决方案。其中，要定义当前价格的“终止日期”有两种方法——不定义终止日期（看似巧妙其实是个坏主意），或者设定为 3000/12/31 之类比较遥远的日期。

显然，要查询商品在 3000/12/31 之前的价格，只需扫描一次。这很好，但此方案完美吗？一方面，第 6 章将讨论优化器在这种情况下会遇到的问题；另一方面，此方案在逻辑上有严重问题：任何消费者都知道，价格很少保持不变，而且价格变化通常不会立即生效（金融市场或许有所不同）。例如，在 10 月时已决定下年度的新价格，而且必须记录到数据库中，会发生什么？

在价格表中，对于不同商品：有一条记录记载当前价格、有效日期至 12 月 31 日，还有一条记录记载着下一年 1 月 1 日起有效的价格。如果采用“记录价格生效起始日期”的方案，应有一条记录的 effective_from_date 字段是诸如“当年 1 月 1 日”之类过去的日期，而另一条记录的 effective_from_date 字段则是诸如“下一年 1 月 1 日”之类未来的日期。事实上，当前价格不是由最大日期来决定，而是由当前日期（在 Oracle 中可由系统函数 sysdate 获得）之前的最大日期来决定的。这样，先前的查询只需稍加修改：

```
select a.article_name, h.price
from articles a,
     price_history h
where a.article_name = some_name
  and h.article_id = a.article_id
  and h.effective_from_date =
     (select max(b.effective_from_date)
      from price_history b
      where b.article_id = h.article_id
        and b.effective_from_date <= sysdate)
```

相反，如果采用“记录价格有效终止日期”的方案，会有一条记录的 end_date 字段为“12 月 31 日”，而另一条记录的 end_date 字段值为 null 或某遥远的日期。查询“价格的 end_date 是大于今天的最小日期”的表达式，性能上并没有明显改善。

非规范化建模当然也是一种选择 ——可以同时保存价格生效和失效日期，或同时保存价格的生效日期和有效天数。这样就可以根据查询需要，随意使用有效期的开始或终止日期。

非规范化总会为数据完整性（data integrity）带来隐患——如果没有定义价格，一个很小的日期输入错误都会后患无穷。当然，我们可以在数据插入或更新时，增加更多的检查来防范风险，代价是查询性能的降低。

另一种方案是分别创建当前价格表和历史价格表，并定义一个操作，当价格改变时将数据从当前价格表移到历史价格表。此方案适合某类应用但难于维护，而且，更难处理"预先记录了未来价格"的情况。

在实际应用当中，可以运用分区（partitioning，见第 5 章）等存储技术，来改善对 `effective_from_date` 方式定义的数据的操作，对大数据量处理尤其有用。

在决定方案之前，我们必须意识到价格表可以以各种形式定义。例如，电信公司的数据量虽大，但价格表较小，且价格不会经常变动；相比而言，投资银行要保存所有证券、期货，以及各类金融商品的新价格，且这些价格会不断变化。在一种情况下优良的解决方案，在另一种情况下未必优良。

总结：要处理不断变化且需保存变化历史的数据，必须根据数据改变的快慢程度，非常小心地决定设计策略。

设计与性能

Design and Performance

一些开发者信奉性能专家，这有点"言过其实（flattering）"（也有点令人担心）。恕我罗嗦，我在本书引言中强调多了：调优（tuning）仅是在目前情况下优化性能至最佳；开发时总想着"先把程序写出来，之后再让专家在生产环境中调优"是错误的。调优几乎从不修改程序结构，但随着重大错误的修正，调整查询带来的性能提升会变得很小。性能调优其实包括两个方面。

- 一方面，为了调优系统整体性能，可根据当前 CPU 能力、可用内存、I/O 子系统等资源情况来设置相应参数，有时也可借助 DBMS 的物理实现来改善性能。这是个高技术含量的任务，有可能在某些方面提高性能，但除非之前的方案有重大错误，否则性能提升很少会超过 20%或 30%。

- 另一方面，可通过修改具体查询来调优。不幸的是，这种方法会受到优化器及不同 DBMS 版本的限制。

事情就是这样。

在我看来，增加索引并不属于数据库调优（虽然调优有时就是要评价与纠正索引方案）。绝大多数索引可以且必须在设计一开始就被正确定义，含糊不清的情况应该靠性能测试来明确。

性能不仅仅是调优几个查询，正如战争不仅仅是打赢几场战役一样。赢得一场战役，仍然可能输掉整场战争。同样，查询调优得再好，整个应用程序的性能可能依然很差，差到无人想用。所以，必须合理地设计数据库、应用，以及 SQL 查询。

设计仅做到功能正确是不够的，设计必须考虑性能。至于后续的调优工作，可作为解决性能问题的补充手段。

总结：性能拙劣的罪魁祸首，是错误的设计。

处理流程

Processing Flow

除了本章已讨论过的所有问题之外，操作模式（operating mode）也可能对系统造成重大影响。这里的"操作模式"，是指数据应该以异步模式处理（例如批处理系统）、还是同步模式处理（例如典型的交易系统）。

批处理（Batch）方式是所有数据处理方式的先驱，现在已不太流行，但仍被广泛使用——毕竟同步处理（synchronous processing）不可或缺的程度没有我们想象的那么高。然而，网络的改进和带宽的增加导致"全球可访问的"应用程序数量急剧增加，于是部署在美国中西部的联机交易处理（online transaction processing ，OLTP）系统不能轻易关闭，因为东亚用户和欧洲用户会在不同时段使用该系统。这样一来，就不能再假设批处理程序可在空闲机器上执行。另外，日渐增加的数据量可能要求立即处理新数据，否

则累积大量数据会难以处理，此时对数据进行同步处理是最有效的方法。

处理数据的方式，会影响我们看待系统的方式，尤其是物理结构方面更是如此，详见第5章的讨论。采用大规模批处理时，我们最关心吞吐量（throughput）——即充分利用硬件资源提高处理效率。从数据处理角度看，批处理是一种“强力处理（brute force）”的方式，被重复执行的小查询居多，这些查询执行速度“还马马虎虎（moderately well）”是不够的——必须做到“尽量快”。采用批处理这种异步方式时，很容易发现问题（如果未能修正）：花的时间太长。而采用同步处理方式时，问题不易发现，因为往往在最糟时才出现性能问题，即有大量的数据处理活动时；如果不能及早发现问题，系统要处理的业务达到峰值时就不正常工作了。

总结：数据模型设计必须考虑数据流。

数据集中化（Centralizing）

Centralizing Your Data

将数据分布到多台服务器上，大大增加了系统复杂性，对网格（grid）、集群（cluster）等皆是如此。无论哪种结构，结构越复杂，健壮性（robust）越低。当然，技术的进步会影响可接受的程度，例如18世纪时人们不大信任能指示分钟的钟表，倒是指示小时的钟表最受信任。尽管如此，还是将数据操作限制在严格要求的范围内为好。

对远程数据的透明引用，是性能的杀手。原因有二。

第一，无论看起来有多“透明”，跨多个软件层或网络的代价都很高。如果不信，可试着执行一个在本地数据表中插入数千行数据的程序，再执行另一个跨机器的相同程序——例如通过指向Oracle的数据库连结进行操作，哪怕此Oracle数据库就是本地数据库——结果就会发现后者比前者至少慢5/6，正如将在第8章看到的例子一样。

第二，来自不同数据源的数据结合极为困难。要比较数据源 A 和数据源 B 的数据时，必须拷贝A的数据到 B，或者拷贝B的数据到A，这种数据传递开销很大。而且，脱

离了数据源环境的数据不能从数据规划（精心设计的物理配置、索引等）中获益，取而代之的是某种临时存储方式（数据量小为内存，数据量大则为磁盘）。临时存储方式的管理是另一个主要开销。理论上，查询本地数据最有效的方式是嵌套的循环。但当一部分数据位于远程时，优化器只有如下两种选择，且都效果不佳：

- 使用套嵌的循环，每次循环开销很大
- 建立远程数据的本地副本，然后操作该副本，但此副本没有任何索引

在这些情况下，性能不佳不能怪优化器。

那么，主要数据库到底该如何部署呢？答案是折衷。如果是家全球性的公司，那么把所有数据保存在一个地点未必受欢迎。这和浏览 Internet 时连接远程服务器不同，频繁访问应用系统完全是另外一回事，这不是带宽问题，而是速度问题；无论怎么做，只要是来自另一块大陆的查询，其响应时间都要再慢上 1/4 秒或半秒，而且这还算是最佳情况。如果希望每个人都能快速访问全球数据，那就应考虑数据复制（replication）解决方案及相关产品。就像玩棋时，棋盘要让每组参赛者触手可及，而无须伸长手臂才够得着。

总结：离数据越近，访问速度越快！

系统复杂性

System Complexity

还有一些问题，是数据库设计中必须时时留意的：如果出现硬件故障（如磁盘控制器故障）或发生了错误操作（如同一个批处理程序被执行了两次），会出现什么情况？即使管理员是个高手，有能力通过夜里加班将系统在天亮前恢复正常，但传输速率依然是个限制——庞大的数据库的恢复工作要花很长时间。或许，采用同步方式（或许有少许延时）建立的“备用”后援数据库有助于解决这一问题，但此方法对某程序不小心执行了

两次的情况不适用，特别是对同步延时小于程序执行时间的情况更是如此。一个数据库发生上述情况已经够棘手的了，若是多个相关联的数据库则情况更糟，因为恢复必须确保所有数据库都正确同步，以免出现数据不一致。

数据恢复往往是开发者和数据库管理员争论的焦点：开发者认为备份和恢复是管理员的工作，这并非完全不合理；而管理员则认为，从理论上讲，即使他们能够保证数据库本身工作正常，依然不知道其中的数据是否正确。的确，开发者不应忘记数据库恢复后要进行所有功能性的检查。整个设计越复杂，开发者就越应记住操作数据时的诸多约束。

总结：数据库系统好似合资企业，需要用户、管理员和开发者积极协作。

小结

The Completed Plans

至此，我们讨论了进行数据库规划的基础知识：包括数据建模的基本原则，特别是对数据进行规范化使其满足 3NF 的主要步骤；以及在不同情况下，错误设计是如何导致灾难性后果的。

本章的多数例子，都是我在一些大公司亲身经历的情况，或是由那些情况得到的启发。忽视基本设计原则会导致性能问题，解决这些问题会浪费惊人的精力与智慧。这样的性能问题非常普遍，而且在已经出现问题的设计中，更进一步打着“改善性能”的旗号进行非规范化处理，经常使性能问题变得更糟。诚然，经过上面的改动，某个查询也许执行得更快了，但不幸的是，夜间的批处理程序要花费两倍的时间。如此这般，整个信息系统在不知不觉中建在了沙基之上。

总结：成功的数据建模应严格遵守基本的设计原则。

第2章

发动战争：高效访问数据库

Waging War: Accessing Databases Efficiently

战争中有一些基本的原则，无视这些原则非常危险，而遵循这些原则可能胜利。

——Général Antoine-Henri de Jomini（1779—1869）

有经验的朋友都知道，把关键系统从开发环境切换到生产环境是一场战役，一场甚嚣尘上的战役。通常，在“攻击发起日（D-Day）”的前几周，性能测试会显示新系统达不到预期要求。于是，找专家，调优 SQL 语句，召集数据库管理员和系统管理员不断开会讨论对策。最后，性能总算与以前的系统大致相当了（尽管新系统用的是价格翻倍的硬件）。

人们常常使用战术，而忽略了战略。战略要求从大局上把握整个架构与设计。和战争一样，战略的基本原则并不多，且经常被忽视。架构错误的代价非常高，SQL 程序员必须准备充分，明确目标，了解如何实现目标。在本章中，我们讨论编写高效访问数据库的程序需要实现哪些关键目标。

查询的识别
Query Identification

数个世纪以来，将军通过辨别军装颜色和旗帜等来判断各部队的位置，以此检查激战中部队行进情况。同样，当一些进程消耗了过多的 CPU 资源时，通常也可以确定是由哪些正被执行的 SQL 语句造成的。但是，要确定是应用的哪部分提交了这些 SQL 语句却困难得多，特别是复杂的大型系统包含动态建立的查询的时候。尽管许多产品提供良好的监控工具，但要确定一小段 SQL 语句与整个系统的关系，有时却非常困难。因此，要养成为程序和关键模块加注释的习惯，在 SQL 中插入注释有助于辨别查询在程序中的位置。例如：

```
/* CUSTOMER REGISTRATION */ select blah ...
```

这些注释在查错时非常有用。另外，注释也有助于判断单独应用对服务器造成的负载有多大；例如我们希望本地应用承担更多工作，需要判断当前硬件是否能承受突发高负载，这时注释特别有用。

有些产品还提供了专门的记录功能（registration facilities），将你从“为每个语句加注释”的乏味工作中解放出来。例如 Oracle 的 `dbms_application_info` 包，它支持 48 个字

符的模块名称（module name）、32 个字符的动作名称（action name）和 64 个 字符的客户信息，这些字段的内容可由我们定制。在 Oracle 环境下，你可以利用这个程序包记录哪个应用正在执行，以及它在何时正在做什么。因为应用是通过“Oracle V$ 动态视图”（能显示目前内存中发生的情况）向程序包传递信息的，于是我们可以轻易地掌握这些信息。

总结：易识别的语句有助于定位性能问题。

保持数据库连接稳定

Stable Database Connections

建立一个新的数据库连接，既快又方便，但这其中往往掩藏着重复建立数据库连接带来的巨大开销。所以，管理数据库连接必须非常小心。允许多重连接——可能就藏在你的应用中——的后果可能很严重，下面即是一例。

不久前，我遇到一个应用，要处理很多小的文本文件。这些文本文件最大的也不超过一百行，每一行包含要加载的数据及数据库等信息。此例中固然只有一个数据库实例，但即使有上百个，这里所说明的原理也是适用的。

处理每个文件的代码如下：

```
Open the file
Until the end of file is reached
      Read a row
      Connect to the server specified by the row
      Insert the data
      Disconnect
Close the file
```

上述处理工作令人满意，但当大量小文件都在极短的时间内到达时，可能应用程序来不及处理，于是积压大量待处理文件，花费时间相当可观。

我用 C 语言编了个简单的程序来模拟上述情况，以说明频繁的数据库连接和中断所造成的系统性能下降问题。表 2-1 列出了模拟的结果。

注意

产生表 2-1 结果的程序使用了常规的 `insert` 语句。顺便提一下，直接加载（direct-loading）的技术会更快。

表 2-1：连接／中断性能测试结果

测 试	结 果
依次对每一行作连接／中断	7.4 行/秒
连接一次，所有行逐个插入	1 681 行/秒
连接一次，以 10 行为一数组插入	5 914 行/秒
连接一次，以 100 行为一数组插入	9 190 行/秒

此例说明了尽量减少分别连接数据库次数的重要性。对比表中前后两次针对相同数据库的插入操作，明显发现性能有显著提升。其实还可以做进一步的优化。因为数据库实例的数量势必有限，所以可以建立一组处理程序（handler）分别负责一个数据库连接，每个数据库只连接一次，使性能进一步提高。正如表 2-1 所示，仅连接数据库一次（或很少次）的简单技巧，再加上一点额外工作，就能让效率提升 200 倍以上。

当然，在上述改进的基础上，再将欲更新的数据填入数组，这样就尽可能减少了程序和数据库核心间的交互次数，从而使性能产生了另一次飞跃。这种每次插入几行数据的做法，可以使数据的总处理能力又增加了 5 倍。表 2-1 中的结果显示改进后的性能几乎是最初的 1 200 倍。

为何有如此大的性能提升？

第一个原因，也是最大的原因，在于数据库连接是很“重”的操作，消耗资源很多。

在常见的客户/服务器模式中（现在仍广为使用），简单的连接操作背后潜藏着如下事实：首先，客户端与远程服务器的监听程序（listener program）建立联系；接着，监听程序要么创建一个进程或线程来执行数据库核心程序，要么直接或间接地把客户请求传递给已存在的服务器进程，这取决于此服务器是否为共享服务器。

除了这些系统操作（创建进程或线程并开始执行）之外，数据库系统还必须为每

次 session 建立新环境，以跟踪它的行为。建立新 session 前，DBMS 还要检查密码是否与保存的加密的账户密码相符。或许，DBMS 还要执行登录触发器（logon trigger），还要初始化存储过程和程序包（如果它们是第一次被调用）。上面这些还不包括客户端进程和服务器进程之间要完成的握手协议。正因为如此，连接池（connection pooling）等保持永久数据库连接的技术对性能才如此重要。

第二个原因，你的程序（甚至包括存储过程）和数据库之间的交互也有开销。

即使数据库连结已经建立且仍未中断，程序和 DBMS 核心之间的上下文切换（context switch）也有代价。因此，如果 DBMS 支持数据通过数组传递，应毫不犹豫地使用它。如果该数组接口是隐式的（API 内部使用，但你不能使用），那么明智的做法是检查它的默认大小并根据具体需要修改它。当然，任何逐行处理的方式都面临上下文切换的问题，并对性能产生严重影响——本章后面还会多次涉及此问题。

总结：数据库连接和交互好似万里长城——长度越长，传递消息越耗时。

战略优先于战术

Strategy Before Tactics

战略决定战术，反之则谬也。思考如何处理数据时，有经验的开发者不会着眼于细微步骤，而是着眼于最终结果。要获得想要的结果，最显而易见的方法是按照业务规则规定的顺序按部就班地处理，但这不是最有效的方法——接下来的例子将显示，刻意关注业务处理流程可能会使我们错失最有效的解决方案。

几年前，有人给了我一个存储过程，让我“尝试”着进行一下优化。为什么说是“尝试”呢？因为该存储过程已经被优化两次了，一次是由原开发者，另一次是由一个自称 Oracle 专家的人。但尽管如此，这个存储过程的执行仍会花上 20 分钟，使用者无法接受。

此存储过程的目的，是根据现有库存和各地订单，计算出总厂需要订购的原料数量。大体上，它就是把不同数据源的几个相同的表聚合（aggregate）到一个主表（master table）

中。首先，将每个数据源的数据插入主表；接着，对主表中的各项数据进行合计并更新；最后，将与合计结果无关的数据从表中删除。针对每个数据源，重复执行上述步骤。所有 SQL 语句都不是特别复杂，也没有哪个单独的 SQL 语句特别低效。

为了理解这个存储过程，我花了大半天时间，终于发现了问题：为什么该过程要用这么多步骤呢？在 from 子句中加上包含 union 的子查询，就能得到所有数据源的聚合（aggregation）。一条 select 语句，只需一步就得到了结果集，而之前要通过插入目标表（target table）得到结果集。优化后，性能的提升非常惊人——从 20 分钟减至 20 秒；当然，之后我花了一些时间验证了结果集，与未优化前完全相同。

想要获得上述的大幅提高性能，无需特别技能，仅要求站在局外思考（think outside the box）的能力。之前两次优化因“太关注问题本身”而收到了干扰。我们需要大胆的思维，站得远一些，试着从大局的角度看待问题。要问自己一些关键的问题：写存储过程之前，我们已有哪些数据？我们希望存储过程返回什么结果？再辅以大胆的思维，思考这些问题的答案，就能得到一个性能大幅提升的处理方式了。

总结：考虑解决方案的细节之前，先站得远一些，把握大局。

先定义问题，再解决问题

Problem Definition Before Solution

一知半解是危险的。人们常在听说了新技术或特殊技术之后——有时的确很吸引人——试图采用它作为新的解决方案。普通开发者和设计师通常会立即采纳这些新“解决方案”，直到后来才发现它们会产生许多后续问题。

现成的解决方案中，非规范化设计引人注目。设计伊始，非规范化设计的拥护者就提出此方案，为了寻求“性能”而无视最终将会面临的升级恶魔——而事实上，在开发周期

早期，改进设计（或学习如何使用 join）也是一个不错的选择。作为非规范化设计的一种手段，物化视图（materialized view）常被认为是灵丹妙药。物化视图有时被称为快照（snapshot），这个更加平常的词更形象地反映了可悲的事实：物化视图是某时间点的数据副本。在没有其他办法时，这个理论上遭到质疑的技术也未尝不值得一试，借用卡夫卡（Franz Kafka）的一句名言："逻辑诚可贵，生存价更高。"

然而，绝大部分问题都可借助传统技术巧妙解决。首先，应学会充分利用简单、传统的技术。只有完全掌握了这些技术，才能正确评价它们的局限性，最终发现它相当于新技术的潜在优势（如果有的话）。

所有技术方案，都只是我们达到目标的手段。没有经验的开发者误把新技术本身当成了目标。对于热衷于技术、过于看重技术的人来说，此问题就更为严重。

总结：先打基础，再赶时髦：摆弄新工具之前，先把手艺学好。

保持数据库 Schema 稳定

Stable Database Schema

在应用程序中使用数据定义语言（data definition language ，DDL）建立、修改或删除数据库对象，是很差的方式，在多数情况下应禁止。除了分区（第 5 章讨论）和 DBMS 已知的临时表之外，根本没有必要动态建立、修改或删除数据库对象。（第 10 章还会讨论此规则的另一个重大例外。）

DDL 的作用是以核心数据库的数据字典为基础的，数据字典是所有数据库操作的中心，所以任何对数据字典的操作都会引起全局加锁，对系统性能影响巨大。唯一可接受的 DDL 操作是对表的 truncate（清空）操作，它能极快地清空表的所有数据（记住，`truncate table` 不可通过回滚恢复!）。

总结：不应在程序中进行数据库对象的建立、修改及删除等操作，虽然设计应用时要考虑这些问题。

直接操作实际数据
Operations Against Actual Data

许多开发者喜欢建立临时工作表（temporary work table），把后续处理使用的大量数据放入其中，然后开始“正式”工作。这种方法广受质疑，反映了“跳出业务流程细节考虑问题”的能力不足。记住，永久表（permanent table）可以设置非常复杂的存储选项（在第 5 章将讨论一些存储选项的设置），而临时表不能。临时表的索引（如果有的话）可能不是最优的，因此，查询临时表的语句效率比永久表的差。另外，查询之前必然先为临时表填入数据，这自然也多了一笔额外的开销。

就算使用临时表有充足理由，若数据量大，也绝不能把永久表当作临时工作表来用。问题之一在于统计信息的自动收集：若没有实时收集要求，DBMS 通常会在不活动或活动少时进行统计信息收集，而这时作为临时工作表可能为空，从而使优化器收到了完全错误的信息。这些不正确且有偏差的统计信息可能造成执行计划（execution plan）完全不合理，导致性能下降。所以，如果一定要用临时表，应确保数据库知道哪些表是临时的。

总结：暂时工作表意味着以不太合理的方式存储更多信息。

用 SQL 处理集合
Set Processing in SQL

SQL 完全基于集合（Set）来处理数据。对大部分更新或删除操作而言 —— 如果不是针对整个表的话 —— 你必须先精确定义出要处理的记录的集合。这定义了该处理的粒度（granularity），可能是对大量记录的粗粒度操作，有可能是只影响少数记录的细粒度操作。

将一次“大批量数据的处理”分割成多次“小块处理”是个坏主意，除非对数据库的修改太昂贵，否则不要使用，因为这种方法极其低效：

(1)占用过多的空间保存原始数据，以备事务（transaction）回滚（rollback）之需；

(2)万一修改失败，回滚消耗过长的实践。

许多人认为，进行大规模修改操作时，应在操作数据的代码中有规律地多安排些 `commit`

命令。其实，严格从实践角度来讲，“从头开始重做”比“确定失败发生的时间和位置，接着已提交部分重做”要容易得多、简单得多、也快得多。

处理数据时，应适应数据库的物理实现。考虑事务失败时回滚所需日志的大小，如果要为 undo 保存的数据量确实巨大，或许应该考虑数据修改的频率问题。也就是说，将大规模的“每月更新”，改为规模不大的“每周更新”，甚至改为规模更小的“每日更新”，或许是个有效方案。

总结：几千个语句，借助游标（cursor）不断循环，很慢。换成几个语句，处理同样的数据，还是较慢。换成一个语句，解决上述问题，最好。

动作丰富的 SQL 语句
Action-Packed SQL Statements

SQL 不是过程性语言（procedural language），尽管也可以将过程逻辑（procedural logic）用于 SQL，但必须小心。混淆声明性处理（declarative processing）和过程逻辑，最常见的例子出现在需要从数据库中提取数据、然后处理数据、然后再插入到数据库时。在一个程序（或程序中的一个函数）接收到特定输入值后，如下情况太常见了：用输入值从数据库中检索到一个或多个另外的数据值，然后，借助循环或条件逻辑（通常是 if ... then ... else）将一些语句组织起来，对数据库进行操作。大多数情况下，造成上述错误做法的原因有三：根深蒂固的坏习惯、SQL 知识的缺乏、盲从功能需求规格说明。其实，许多复杂操作往往可由一条 SQL 语句完成。因此，如果用户提供了一些数据值，尽量不要将操作分解为多条提取中间结果的语句。

避免在 SQL 中引入“过程逻辑（procedural logic）”的主要原因有二。

数据库访问，总会跨多个软件层，甚至包括网络访问。

即使没有网络访问，也会涉及进程间通讯；额外的存取访问意味着更多的函数调用、更大的带宽，以及更长的等待时间。一旦这些调用要重复多次，其对性能的影响就非常可观了。

在 SQL 中引入过程逻辑，意味着性能和维护问题应由你的程序承担。

> 大多数据库系统都提供了成熟的算法，来处理 join 等操作，来优化查询以获得更高的效率。基于开销的优化器（cost-based optimizer，CBO）是很复杂的软件，它早已不像刚推出时那样没什么用了，而在大部分情况下都是非常出色的成熟产品了，优秀的 CBO 查询优化的效率极高。然而，CBO 所能改变的只有 SQL 语句。如果在一条单独的 SQL 语句中完成尽可能多的操作，那么性能优化可以还由 DBMS 核心负责，你的程序可以充分利用 DBMS 的所有升级。也就是说，未来大部分维护工作从程序间接转移给了 DBMS 供货商。

当然，“避免在 SQL 中引入过程逻辑”规则也有例外。有时过程逻辑确实能加快处理速度，庞大的 SQL 语句未必总是高效。然而，过程逻辑及其之后的处理相同数据的语句，可以编写到一个单独的 SQL 语句中，CBO 就是这么做的，从而获得最高效的执行方式。

总结：尽可能多地把事情交给数据库优化器来处理。

充分利用每次数据库访问

Profitable Database Accesses

如果计划逛好几家商店，你会首先决定在每家店买哪些东西。从这一刻起，就要计划按何种顺序购物才能少走冤枉路。每逛一家店，计划东西购买完毕，才逛下一家。这是常识，但其中蕴含的道理许多数据库应用却不懂得。

要从一个表中提取多段信息时，采用多次数据库访问的做法非常糟糕，即使多段信息看似“无关”（但事实上往往并非如此）。例如，如果需要多个字段的数据，千万不要逐个字段地提取，而应一次操作全部完成。

很不幸，面向对象（OO）的最佳实践提倡为每个属性定义一个 get 方法。不要把 OO 方法与关系数据库处理混为一谈。混淆关系和面向对象的概念，以及将表等同于类、字段等同于属性，都是致命的错误。

总结：在合理范围内，利用每次数据库访问完成尽量多的工作。

接近 DBMS 核心

Closeness to the DBMS Kernel

代码的执行越接近 DBMS 核心，则执行速度越快。数据库真正强大之处就在于此，例如，有些数据库管理产品支持扩展，你可以用 C 等较底层的语言为它编写新功能。用含有指针操作的底层语言有个缺点，即一旦指针处理出错会影响内存。仅影响到一个用户已很糟糕，何况数据库服务器（就像"服务器"名字所指的一样）出了问题会影响众多"用户"——服务器内存出了问题，所有使用这些数据的无辜的应用程序都会受影响。因此，DBMS 核心采取了负责任的做法，在沙箱（sandbox）环境中执行程序代码，这样，即使出了问题也不会影响到数据。例如，Oracle 在外部函数（external function）和它自身之间实现了一套复杂的通信机制，此机制在某些方面很像控制数据库连结的方法，以管理两个（或多个）服务器上的数据库实例之间的通信。到底采用 PL/SQL 存储过程还是外部 C 函数，应综合比较后决定。如果精心编写外部 C 函数获得的好处超过了建立外部环境和上下文切换（context-switching）的成本，就应采用外部函数。但需要处理一个大数据量的表的每一行时，不要使用外部函数。这需要平衡考虑，解决问题时应完全了解备选策略的后果。

如要使用函数，始终应首选 DBMS 自带的函数。这不仅仅是为了避免无谓的重复劳动，还因为自带函数在执行时比任何第三方开发的代码更接近数据库核心，相应地其效率也会高出许多。

下面这个简单例子是用 Oracle SQL 编写的，显示了 使用 Oracle 函数所获得的效率。假设手工输入的文本数据可能包含多个相邻的"空格"，我们需要一个函数将多个空格

替换为一个空格。如果不采用 Oracle Database 10g 开始提供的正规表达式（regular expression），函数代码将会是这样：

```
create or replace function squeeze1(p_string in varchar2)
return varchar2
is
  v_string varchar2(512) := '';
  c_char   char(1);
  n_len    number := length(p_string);
  i        binary_integer := 1;
  j        binary_integer;
begin
  while (i <= n_len)
  loop
    c_char := substr(p_string, i, 1);
    v_string := v_string || c_char;
    if (c_char = ' ')
    then
      j := i + 1;
      while (substr(p_string || 'X', j, 1) = ' ')
      loop
        j := j + 1;
      end loop;
      i := j;
    else
      i := i + 1;
    end if;
  end loop;
  return v_string;
end;
/
```

上述代码中的 'X' 在内层循环中被串接到字符串上，以避免超出字符串长度的测试。

还有别的方法消除多个空格，可以使用 Oracle 提供的字符串函数。以下为替代方案：

```
create or replace function squeeze2(p_string in varchar2)
return varchar2
is
  v_string varchar2(512) := p_string;
  i        binary_integer := 1;
begin
  i := instr(v_string, '  ');
  while (i > 0)
  loop
    v_string := substr(v_string, 1, i)
             || ltrim(substr(v_string, i + 1));
    i := instr(v_string, '  ');
  end loop;
  return v_string;
end;
/
```

还有第三种方法：

```
create or replace function squeeze3(p_string in varchar2)
return varchar2
is
  v_string varchar2(512) := p_string;
  len1     number;
  len2     number;
begin
  len1 := length(p_string);
  v_string := replace(p_string, '  ', ' ');
  len2 :=  length(v_string);
  while (len2 < len1)
  loop
    len1 := len2;
    v_string := replace(v_string, '  ', ' ');
    len2 :=  length(v_string);
  end loop;
  return v_string;
 end;
/
```

用一个简单的例子对上述三种方法进行测试，每个函数都能正确工作，且没有明显的性能差异：

```
SQL> select squeeze1('azeryt  hgfrdt   r')
  2  from dual
  3  /
azeryt hgfrdt r

Elapsed: 00:00:00.00
SQL> select squeeze2('azeryt  hgfrdt   r')
  2  from dual
  3  /
azeryt hgfrdt r

Elapsed: 00:00:00.01
SQL> select squeeze3('azeryt  hgfrdt   r')
  2  from dual
  3  /
azeryt hgfrdt r

Elapsed: 00:00:00.00
```

那么，如果每天要调用该空格替换操作几千次呢？我们构造一个接近现实负载的环境，下面的代码将建立一个用于测试的表并填入随机数据，已检测上面三个函数是否有性能差异：

```
create table squeezable(random_text  varchar2(50))
/

declare
   i        binary_integer;
```

```
    j         binary_integer;
    k         binary_integer;
    v_string  varchar2(50);
begin
  for i in 1 .. 10000
  loop
    j := dbms_random.value(1, 100);
    v_string := dbms_random.string('U', 50);
    while (j < length(v_string))
    loop
      k := dbms_random.value(1, 3);
      v_string := substr(substr(v_string, 1, j) || rpad(' ', k)
                  || substr(v_string, j + 1), 1, 50);
      j := dbms_random.value(1, 100);
    end loop;
    insert into squeezable
    values(v_string);
  end loop;
  commit;
end;
/
```

上面的脚本在测试表中建立了10 000条记录（决定 SQL 语句要执行多少次时，这是数字比较适中）。要执行该测试，运行下列语句：

```
select squeeze_func(random_text)
from squeezable;
```

我运行这个测试时，关闭了所有头信息（headers）和屏幕的显示。禁止输出可确保结果反映的是替换空格算法所花费的时间，而不是显示结果所花费的时间。这些语句会执行多次，以确保不受缓存（caching）的影响。

表2-2显示了在测试机上的运行结果。

表2-2：处理10 000条记录中空格所花的时间

函数	机制	时间
squeeze1	用 PL/SQL 循环处理字符	0.86 秒
squeeze2	Instr() + ltrim()	0.48 秒
squeeze3	循环调用 replace()	0.39 秒

尽管都在1秒内完成了10000次调用，但 squeeze2 的速度是 squeeze1 的 1.8 倍，而 squeeze3 则是它的 2.2 倍。为什么呢？原因很简单，因为 SQL 函数比 PL/SQL“离核心更近”。当函数只偶尔执行一次时，性能差异微乎其微，但在批处理程序或高负载的 OLTP 服务器中性能差异就非常明显。

总结：代码喜欢 SQL 内核——离核心越近，它就运行得越快。

只做必须做的

Doing Only What Is Required

开发者使用 count(*)往往只是为了测试“是否存在”。这通常是由以下的需求说明引起的：

```
如果存在满足某条件的记录
那么处理这些记录
```

用代码直接实现就是：

```
select count(*)
into counter
from table_name
where <certain_condition>

if (counter > 0) then
```

当然，在 90% 的情况下，count(*) 是完全不必要的，正如上面的例子。要对多项记录进行操作，直接做即可，不必用 count(*)。即使一个操作对任何记录都没有影响，也没有关系，不用 count(*)没有什么不好。而且，即使要对未知的记录进行复杂处理，也能通过第一个操作就确定并返回受影响的记录——要么通过特殊的 API（例如 PHP 中的 mysql_affected_rows()），要么采用系统变量（Transact-SQL 中为@@ROWCOUNT，PL/SQL 中为 SQL%ROWCOUNT），若使用内嵌式 SQL，则使用 SQL 通讯区（SQL Communication Area，SQLCA）的特殊字段。有时，可以通过函数访问数据库然后直接返回要处理的记录数，例如 JDBC 的 executeUpdate()方法。总之，统计记录数极可能意味着重复全部搜索，因为它对相同数据处理了两次。

此外，如果是为了更新或插入记录（常使用 count 检查键是否已经存在），一些数据库系统会提供专用的语句（例如 Oracle 9i 提供 MERGE 语句），其执行效率要比使用 count 高得多。

总结：没必要编程实现那些数据库隐含实现的功能。

SQL 语句反映业务逻辑

SQL Statements Mirror Business Logic

大多数数据库系统都提供监控功能，我们可以借此查看当前正在执行的语句及其执行的次数。同时，必须对有多少个"业务单元（business units）"正在执行心里有数——例如待处理的订单、需处理的请求、需结账的客户，或者业务管理者了解的任何事情。我们应检查上述语句活动和业务活动的数量关系是否合理（并不要求绝对精确）。换言之，如果客户数量一定，那么数据库初始化活动的数量是否与之相同？如果查询 customers 表的次数比同一时间正在处理的客户量多 20 倍，那一定是某个地方出了问题，或许该查询对表中相同记录做了重复（而且多余）的访问，而不是一次就从表中找出了所需信息。

总结：检查数据库活动，看它是否与当时正进行的业务活动保持合理的一致性。

把逻辑放到查询中

Program Logic into Queries

在数据库应用程序中实现过程逻辑（procedural logic）的方法有几种。SQL 语句内部可实现某种程度上的过程逻辑（尽管 SQL 语句应该说明做什么，而不是怎么做）。即便内嵌式 SQL 的宿主语言（host language）非常完善，依然推荐尽量将上述过程逻辑放在 SQL 语句当中，而不是宿主语言当中，因为前一种做法效率更高。过程性语言（Procedural language）的特点在于拥有执行迭代（循环）和条件（if ... then ... else 结构）逻辑的能力。SQL 不需要循环能力，因为它本质上是在操作集合，SQL 只需要执行条件逻辑的能力。

条件逻辑包含两部分——IF 和 ELSE。要实现 IF 的效果相当容易——where 子句可以胜任，困难的是实现 ELSE 逻辑。例如，要取出一些记录，然后对其分组，每组进行不同的转换。case 表达式（Oracle 早已在 decode()（注 1）中提供了功能等效的操作符）可以容易地模拟 ELSE 逻辑：根据每条记录值的不同，返回具有不同值的结果集。下面用伪代码（pseudocode）表达 case 结构的使用（注 2）：

注 1：decode() 没有 case 的能力强，所以需要 sign() 等函数的帮助才能达到 case 的效果。

注 2：Case 语句有两种变体，本例使用最成熟的变体。

```
CASE
WHEN condition THEN <return something to the result set>
   WHEN condition THEN <return something else>
...
   WHEN condition THEN <return still something else>
   ELSE <fall back on this value>
END
```

数值或日期的比较则简单明了。操作字符串可以用 Oracle 的 greatest()或 least()，或者 MySQL 的 strcmp()。有时，可以为 insert 语句增加过程逻辑，具体办法是多重 insert 及条件 insert（注 3），并借助 merge 语句。如果 DBMS 提供了这样语句，毫不犹豫地使用它。也就是说，有许多逻辑可以放入 SQL 语句中；虽然仅执行多条语句中的一条这种逻辑价值不大，但如果设法利用 case、merge 或类似功能将多条语句合并成一条，价值可就大了。

总结：只要有可能，应尽量把条件逻辑放到 SQL 语句中，而不是 SQL 的宿主语言中。

一次完成多个更新
Multiple Updates at Once

我的基本主张是：如果每次更新的是彼此无关的记录，对一张表连续进行多次 update 操作还可以接受；否则，就应该把它们合并成一个 update 操作。例如，下面是来自实际应用的一些代码（注 4）：

```
update tbo_invoice_extractor
set pga_status = 0
where pga_status in (1,3)
  and inv_type = 0;
 update tbo_invoice_extractor
   set rd_status = 0
 where rd_status in (1,3)
   and inv_type = 0;
```

两个连续的更新是对同一个表进行的。但它们是否将访问相同的记录呢？不得而知。问题是，搜索条件的效率有多高？任何名为 type 或 status 的字段，其值的分布通常是杂乱无章的，所以上面两个 update 语句极可能对同一个表连续进行两次完整扫描：一个 update 有效地利用了索引，而第二个 update 不可避免地进行全表扫描；或者，幸运

注 3：例如，Oracle 9.2 支持条件 insert。
注 4：表名已改变。

的话，两次 update 都有效地利用了索引。无论如何，把这两个 update 合并到一起，几乎不会有损失，只会有好处：

```
update tbo_invoice_extractor
set pga_status = (case pga_status
                  when 1 then 0
                  when 3 then 0
                  else pga_status
                 end),
    rd_status = (case rd_status
                  when 1 then 0
                  when 3 then 0
                  else rd_status
                 end)
where (pga_status in (1,3)
      or rd_status in (1, 3))
  and inv_type = 0;
```

上例中，可能出现重复更新相同字段为相同内容的情况，这的确增加了一小点儿开销。但在多数情况下，一个 update 会比多个 update 快得多。注意上例中的"逻辑（logic）"，我们通过 case 语句实现了隐式的条件逻辑（implicit conditional logic），来处理那些符合更新条件的数据记录，并且更新条件可以有多条。

总结：有可能的话，用一个语句处理多个更新；尽量减少对同一个表的重复访问。

慎用自定义函数

Careful Use of User-Written Functions

将自定义函数（User-Written Function）嵌到 SQL 语句后，它可能被调用相当多次。如果在 select 语句的选出项列表中使用自定义函数，则每返回一行数据就会调用一次该函数。如果自定义函数出现在 where 子句中，则每一行数据要成功通过过滤条件都会调用一次该函数；如果此时其他过滤条件的筛选能力不够强，自定义函数被调用的次数就非常可观了。

如果自定义函数内部还要执行一个查询，会发生什么情况呢？每次函数调用都将执行此内部查询。实际上，这和关联子查询（correlated subquery）效果相同，只不过自定义函数的方式阻碍了基于开销的优化器（cost-based optimizer，CBO）对整个查询的优化效果，因为"子查询"隐藏在函数中，数据库优化器鞭长莫及。

下面举例说明将 SQL 语句隐藏在自定义函数中的危险性。表 `flights` 描述商务航班，有航班号、起飞时间、到达时间及机场 IATA 代码（注 5）等字段。IATA 代码均为三个字母，有 9 000 多个，它们的解释保存在参照表中，包含城市名称（若一个城市有多个机场则应为机场名称）、国家名称等。显然，显示航班信息时，应该包含目的城市的机场名称，而不是简单的 IATA 代码。

在此就遇到了现代软件工程中的矛盾之一。被认为是“优良传统”的模块化编程一般情况下非常适用，但对数据库编程而言，代码是开发者和数据库引擎的共享活动（shared activity），模块化要求并不明确。例如，我们可以遵循模块化原则编写一个小函数来查找 IATA 代码，并返回完整的机场名称：

```
create or replace function airport_city(iata_code in char)
return varchar2
is
  city_name  varchar2(50);
begin
  select city
  into city_name
  from iata_airport_codes
  where code = iata_code;
  return(city_name);
end;
/
```

对于不熟悉 Oracle 语法的读者，在此做个说明，以下查询中 `trunc(sysdate)`的返回值为“今天的 00:00 a.m.”，日期计算以天为单位；所以起飞时间的条件是指今天 8:30 a.m. 至 4:00 p.m. 之间。调用 `airport_city` 函数的查询可以非常简单，例如：

```
select flight_number,
       to_char(departure_time, 'HH24:MI') DEPARTURE,
       airport_city(arrival) "TO"
from flights
where departure_time between trunc(sysdate) + 17/48
                         and trunc(sysdate) + 16/24
order by departure_time
/
```

这个查询的执行速度令人满意；在我机器上的随机样本中，返回 77 行数据只用了 0.18 秒（多次执行的平均值），用户对这样的速度肯定满意（统计数据表明，此处理访问了

注 5：国际航空运输协会（International Air Transport Association）。

303 个数据块，53 个是从磁盘读出的——而且每行数据有个递归调用）。

我们还可以用 join 来重写这段代码，作为查找函数的替代方案，当然它看起来会稍微复杂些：

```
select f.flight_number,
      to_char(f.departure_time, 'HH24:MI') DEPARTURE,
      a.city "TO"
from flights f,
    iata_airport_codes a
where a.code = f.arrival
  and departure_time between trunc(sysdate) + 17/48
                         and trunc(sysdate) + 16/24
order by departure_time
/
```

这个查询只用了 0.05 秒（统计数据同前，但没有递归调用）。对于执行时间不到 0.2 秒的查询来说，速度快了 3 倍似乎无关紧要，但在大型系统中，这些查询每天经常执行数十万次——假设以上查询每天只执行五万次，于是查询的总耗时为 2.5 小时。若不使用上述查找函数（lookup function）则只需要不到 42 分钟，速度提高超过 300%，这对大数据量的系统意义重大，最终带来经济上的节约。通常，使用查找函数会使批处理程序的性能极差。而且查询时间的增加，会使同一台机器支持的并发用户数减少，我们将在第 9 章对此展开讨论。

总结：优化器对自定义函数的代码无能为力。

简洁的 SQL
Succinct SQL

熟练的开发者使用尽可能少的 SQL 语句完成尽可能多的事情。相反，拙劣的开发者则倾向于严格遵循已制订好的各功能步骤，下面是个真实的例子：

```
-- Get the start of the accounting period
select closure_date
into dtPerSta
from tperrslt
where fiscal_year=to_char(Param_dtAcc,'YYYY')
  and rslt_period='1' || to_char(Param_dtAcc,'MM');
```

```
-- Get the end of the period out of closure
select closure_date
into dtPerClosure
from tperrslt
where fiscal_year=to_char(Param_dtAcc,'YYYY')
  and rslt_period='9' || to_char(Param_dtAcc,'MM');
```

就算速度可以接受，这也是段极糟的代码。很不幸，性能专家经常遇到这种糟糕的代码。既然两个值来自于同一表，为什么要分别用两个不同的语句呢？下面用 Oracle 的 bulk collect 子句，一次性将两个值放到数组中，这很容易实现，关键在于对 rslt_period 进行 order by 操作，如下所示：

```
select closure_date
bulk collect into dtPerStaArray
from tperrslt
where fiscal_year=to_char(Param_dtAcc,'YYYY')
  and rslt_period in ('1' || to_char(Param_dtAcc,'MM'),
                      '9' || to_char(Param_dtAcc,'MM'))
order by rslt_period;
```

于是，这两个日期被分别保存在数组的第一个和第二个位置。其中，bulk collect 是 PL/SQL 语言特有的，但任何支持显式或隐式数组提取的语言都可如法炮制。

其实甚至数组都是不必要的，用以下的小技巧（注 6），这两个值就可以被提取到两个变量中：

```
select max(decode(substr(rslt_period, 1, 1), -- Check the first character
                '1', closure_date,
                    -- If it's '1' return the date we want
                   to_date('14/10/1066', 'DD/MM/YYYY'))),
                    -- Otherwise something old
      max(decode(substr(rslt_period, 1, 1),
                '9', closure_date, -- The date we want
                   to_date('14/10/1066', 'DD/MM/YYYY'))),
into dtPerSta, dtPerClosure
from tperrslt
where fiscal_year=to_char(Param_dtAcc,'YYYY')
  and rslt_period in ('1' || to_char(Param_dtAcc,'MM'),
                      '9' || to_char(Param_dtAcc,'MM'));
```

在这个例子中，预期返回值为两行数据，所以问题是：如何把原本属于一个字段的两行数据，以一行数据两个字段的方式检索出来（正如数组提取的例子一样）。为此，我们

注 6：Oracle 的 decode() 函数类似 case 语句。如果第一个参数等于第二个参数，则返回第三个参数。没有第五个参数的情况下，第四个参数就是 else 返回值；否则，如果第一个参数等于第四个参数，则返回第五个参数……其余参数对的含义依次类推。

检查 rslt_period 字段，两行数据的 rslt_period 字段有不同值；如果找到需要的记录，就返回要找的日期；否则，就返回一个在任何情况下都远比我们所需日期要早的日期（此处选了哈斯丁之役（battle of Hastings）的日期）。只要每次取出最大值，就可以确保获得需要的日期。这是个非常实用的技巧，也可以应用在字符或数值数据，第 11 章会有更详细的说明。

总结：SQL 是声明性语言（declarative language），所以设法使你的代码超越业务过程的规格说明。

SQL 的进攻式编程

Offensive Coding with SQL

一般的建议是进行防御式编程（code defensively），在开始处理之前先检查所有参数的合法性。但实际上，对数据库编程而言，尽量同时做几件事情的进攻式编程有切实的优势。

有个很好的例子：进行一连串检查，每当其中一个检查所要求的条件不符时就产生异常。信用卡付款的处理中就涉及类似步骤。例如，检查所提交的客户身份和卡号是否有效，以及两者是否匹配；检查信用卡是否过期；最后，检查当前的支付额是否超过了信用额度。如果通过了所有检查，支付操作才继续进行。

为了完成上述功能，不熟练的开发者会写出下列语句，并检查其返回结果：

```
select count(*)
from customers
where customer_id = provided_id
```

接下来，他会做类似的工作，并再一次检查错误代码：

```
select card_num, expiry_date, credit_limit
from accounts
where customer_id = provided_id
```

之后，他才会处理金融交易。

相反，熟练的开发者更喜欢像下面这样编写代码（假设 today()返当前日期）：

```
update accounts
set balance = balance - purchased_amount
where balance >= purchased_amount
  and credit_limit >= purchased_amount
  and expiry_date > today()
  and customer_id = provided_id
  and card_num = provided_cardnum
```

接着，检查被更新的行数。如果结果为 0，只需执行下面的一个操作即可判断出错原因：

```
select c.customer_id, a.card_num, a.expiry_date,
      a.credit_limit, a.balance
from customers c
    left outer join accounts a
        on a.customer_id = c.customer_id
        and a.card_num = provided_cardnum
where c.customer_id = provided_id
```

如果此查询没有返回数据，则可断定 customer_id 的值是错的；如果 card_num 是 null，则可断定卡号是错的；等等。其实，多数情况下此查询无需被执行。

注意

你是否注意到，上述第一段代码中使用了 count(*)呢？这是个 count(*)被误用于存在性检测的绝佳例子。

“进攻式编程”的本质特征是：以合理的可能性（reasonable probabilities）为基础。例如，检查客户是否存在是毫无意义的——因为既然该客户不存在，那么他的记录根本就不在数据库中！所以，应该先假设没有事情会出错；但如果出错了，就在出错的地方（而且只在那个地方）采取相应措施。有趣的是，这种方法很像一些数据库系统中采用的“乐观并发控制（optimistic concurrency control）”，后者会假设 update 冲突不会发生，只在冲突真的发生时才进行控制处理。结果，乐观方法比悲观方法的吞吐量高得多。

总结：以概论为基础进行编程。假设最可能的结果；不是的确必要，不要采用异常捕捉的处理方式。

精明地使用异常（Exceptions）
Discerning Use of Exceptions

勇敢与鲁莽的界线很模糊，我建议进攻式编程，但并不是要你模仿轻步兵旅在 Balaclava 的自杀性冲锋（注 7）。针对异常编程，最终可能落得虚张声势的愚蠢结果，但自负的开发者还是对它“推崇备至（go for it）”，并坚信检查和处理异常能使他们完成任务。

正如其名字所暗示的，异常应该是那些例外情况。对数据库编程的具体情况而言，不是所有异常都要求同样的处理方式——这是理解异常的使用是否明智的关键点。有些是“好”异常，应预先抛出；有些是“坏”异常，仅当真正的灾害发生时才抛出。

例如，以主键为条件进行查询时，如果没有结果返回则开销极少，因为只需检查索引即可判断。然而，如果查询无法使用索引，就必须搜索整个表——当此表数据量很大，所在机器又正在接近满负荷工作时，可能造成灾难。

有些异常的处理代价高昂，即使是在最佳情况下也不例外，例如重复键（duplicate key）的探测。“唯一性（uniqueness）”如何保证呢？我们几乎总是建立一个唯一性索引，每次向该索引增加一个键时，都要检查是否违反了该唯一性索引的约束。然而，建立索引项需要记录物理地址，于是就要求先将记录插入表，后将索引项插入索引。如果违反此约束，数据库会取消不完全的插入，并返回违反约束的错误信息。上述这些操作开销巨大。但最大的问题是，整个处理必须围绕个别异常展开，于是我们必须“从个别记录的角度进行思考”，而不是“从数据集出发进行思考”，这与关系数据库理论完全背道而驰。多次违反此约束会导致性能严重下降。

来看一个 Oracle 的例子。假设在两家公司合并后，电子邮件地址定为<Initial><Name>的标准格式，最多 12 个字符，所有空格或引号以下划线代替。

注 7：1845 年的克里米亚战争中，英国、法国、土耳其与俄国作战，由于命令错误和某些指挥官之间的个人恩怨，导致 600 名英国骑兵顶着俄军密集的火力冲进山谷。大约 120 人和 60 匹战马阵亡。Tennyson 的诗歌“The bravery of the men”，以及后来好莱坞拍的电影，将愚蠢的军事行动变成了一个神话。

如果新的 employee 表已经建好，并包含 3 000 条从 employee_old 表中提取并进行标准化处理的电子邮件地址。我们希望每个员工的电子邮件地址具有唯一性，于是 Fernando Lopez 的地址为 flopez，而 Francisco Lopez 的地址为 flopez2。实际上，我们实际测试的数据中有 33 个潜在的重复项，所以我们需要做如下测试：

```
SQL> insert into employees(emp_num, emp_name,
                           emp_firstname, emp_email)
  2  select emp_num,
  3         emp_name,
  4         emp_firstname,
  5         substr(substr(EMP_FIRSTNAME, 1, 1)
  6              ||translate(EMP_NAME, ' ''', '_ _'), 1, 12)
  7  from employees_old;

insert into employees(emp_num, emp_name, emp_firstname, emp_email)
*
ERROR at line 1:
ORA-00001: unique constraint (EMP_EMAIL_UQ) violated


Elapsed: 00:00:00.85
```

3 000 条数据中重复 33 条，比率大约是 1%，所以，或许可以心安理得地处理符合标准的 99%，并用异常来处理其余部分。毕竟，1% 的不符标准数据带来的异常处理开销应该不大。以下是采用该“乐观方法”的代码：

```
SQL> declare
  2    v_counter    varchar2(12);
  3    b_ok        boolean;
  4    n_counter    number;
  5    cursor c is  select emp_num,
  6                        emp_name,
  7                        emp_firstname
  8                 from employees_old;
  9  begin
 10    for rec in c
 11    loop
 12      begin
 13        insert into employees(emp_num, emp_name,
 14                              emp_firstname, emp_email)
 15        values (rec.emp_num,
 16                rec.emp_name,
 17                rec.emp_firstname,
 18                substr(substr(rec.emp_firstname, 1, 1)
 19                ||translate(rec.emp_name, ' ''', '_ _'), 1, 12));
 20      exception
 21       when dup_val_on_index then
 22         b_ok := FALSE;
 23         n_counter := 1;
 24         begin
 25           v_counter := ltrim(to_char(n_counter));
```

```
26          insert into employees(emp_num, emp_name,
27                                emp_firstname, emp_email)
28          values (rec.emp_num,
29                 rec.emp_name,
30                 rec.emp_firstname,
31                 substr(substr(rec.emp_firstname, 1, 1)
32                   ||translate(rec.emp_name, ' ''', '_ _'), 1,
33                   12 - length(v_counter)) || v_counter);
34          b_ok := TRUE;
35        exception
36         when dup_val_on_index then
37           n_counter := n_counter + 1;
38        end;
39      end;
40   end loop;
41  end;
40  /

PL/SQL procedure successfully completed.

Elapsed: 00:00:18.41
```

但这个异常处理的开销到底在哪里呢？让我们先从测试数据中剔除“问题记录”，然后再执行相同的测试，比较发现：这次测试的总运行时间，与上次几乎相同，都是 18 秒。然而，从测试数据中剔除“问题记录”之后再执行前面第一段 insert...select 语句时，速度明显比循环快：最终发现采用“一次处理一行”的方式导致耗时增加了近 50%。那么，在此例中可以不用“一次处理一行”的方式吗？可以，但要首先避免使用异常。正是这个通过异常处理解决“问题记录”问题决定，迫使我们采用循序方式的。

另外，由于发生冲突的电子邮件地址可能不止一个，可以为它们指定某个数字获得唯一性。

很容易判断有多少个数据记录发生了冲突，增加 一个 group by 子句就可以了。但在分配数字时，如果不使用主数据库系统提供的分析功能，恐怕比较困难。(Oracle 称为分析功能（analytical function），DB2 则称在线分析处理（online analytical processing，OLAP），SQL Server 称之为排名功能（ranking function）。）纯粹从 SQL 角度来看，探索此问题的解决方案很有意义。

重复的电子邮件地址都可以被赋予一个具唯一性的数字：1 赋给年纪最大的员工，2 赋给年纪次之的的员工……依次类推。为此，可以编写一个子查询，如果是 group 中的第一个电子邮件地址就不作操作，而该 group 中的后续电子邮件地址则加上序号。代码如下：

```
SQL> insert into employees(emp_num, emp_firstname,
  2                        emp_name, emp_email)
  3  select emp_num,
  4         emp_firstname,
  5         emp_name,
  6         decode(rn, 1, emp_email,
  7                       substr(emp_email,
  8                       1, 12 - length(ltrim(to_char(rn))))
  9                        || ltrim(to_char(rn)))
 10  from (select emp_num,
 11               emp_firstname,
 12               emp_name,
 13               substr(substr(emp_firstname, 1, 1)
 14                 ||translate(emp_name, ' ''', '_ _'), 1, 12)
 15                       emp_email,
 16               row_number()
 17                  over (partition by
 18                        substr(substr(emp_firstname, 1, 1)
 19                         ||translate(emp_name,' ''','_ _'),1,12)
 20                        order by emp_num) rn
 21        from employees_old)
 22  /

3000 rows created.

Elapsed: 00:00:11.68
```

上面的代码避免了一次一行的处理，而且该解决方案的执行时间仅是先前方案的 60%。

总结：异常处理会迫使我们采用过程式逻辑。应始终使用声明式 SQL，尽量预测可能的异常情况。

第3章

战术部署：建立索引

Tactical Dispositions: Indexing

无论谁想运筹帷幄，都要遵循罗马的战略战术：首先是速决战。

——Niccolò Machiavelli（1469—1527）

一旦战场布署已定，将军必须明确敌方要害，作为重点攻击目标。对信息系统也是如此，我们必须关注关键数据的读取，为它们提供最高效的访问路径。对此，基本策略就是建立索引。建立索引是个复杂的话题，比如必须解决优先级的问题。本章中，我们将讨论索引（index）的各种问题和建立索引（indexing）的策略，提供数据库访问策略的基本方针。

找到“切入点”

The Identification of “Entry Points”

你必须对搜索条件有所了解，否则不要着手编写任何 SQL 语句，因为搜索条件对用户很重要。程序的输入值为哪些，以及定义的数据子集的大小，是建立索引的基础。从本质上讲，索引是一种尽快访问“特定数据”的技术。注意我这里说的是“特定数据”，因为索引不是万能药，靠索引并不能实现对所有数据的快速存取。事实上，如果索引策略和数据检索需求严重不符的话，建立索引反而会降低查询性能。

可以把索引看成是指向数据的快捷方式，但它与图形化桌面环境中的快捷方式意义不同。因为索引的开销相当高，包括磁盘空间的开销及处理（processing）开销等，其中处理开销往往更高。例如，有时索引的数据量远远超过数据表本身的大小，这种情况并不罕见。第 1 章对表中冗余数据的负面分析也同样适用于索引：索引通常会被镜像、备份到其他磁盘等；它们庞大的数据量也会带来很多开销，而且不止是存储设备的开销，系统停机后从备份中恢复开销也很大。

图 3-1 显示了一个实际案例：某大银行的主要账户表。数据表和索引共占用 33GB 空间，而索引占了其中 75%还多。

暂且不提存储开销，下面讨论一下处理开销。每当在表中插入或删除记录时，该表的所有索引都必须进行相应调整。每当对已建立索引的字段（indexed column）进行更新时，这种调整（或“维护”）也会发生；例如，被更新的字段本身被建立了索引，或者该字段是一个复合索引（多个字段一同被建立了索引）的一部分。在实际当中，这个“维护”活动意味着：占用大量 CPU 资源以扫描内存数据块、启动 I/O 操作将变动记入日志文件、可能引起更多的数据库文件 I/O 操作，并可能引起数据库系统进行维护存储设备空间分配的操作。

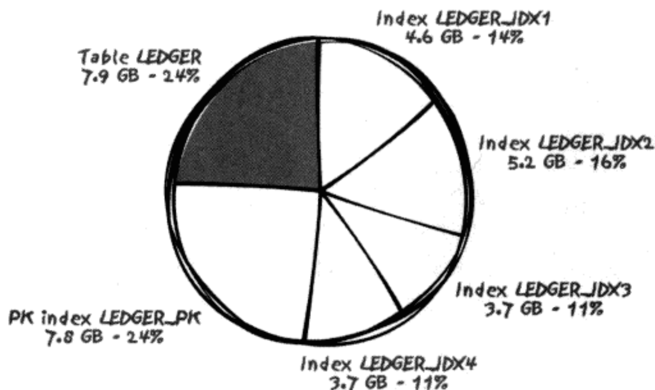


图 3-1：实际案例，数据表和索引共占用 33GB 空间

通过测试，可以量化维护一个表的索引所需开销。例如，如果在未建立索引的表中插入数据需 100 个单位时间（这里单位是秒、分钟还是小时并不重要），那么每增加一个索引就会增加 100 到 250 个单位时间。

总结：维护一个索引的开销，可能比维护一张表的开销还要大。

尽管索引的具体实现因 DBMS 产品不同而异，但其高昂的维护成本，对所有产品都是一样的，图 3-2 和 3-3 分别显示了使用 Oracle 和 MySQL 时的情况。

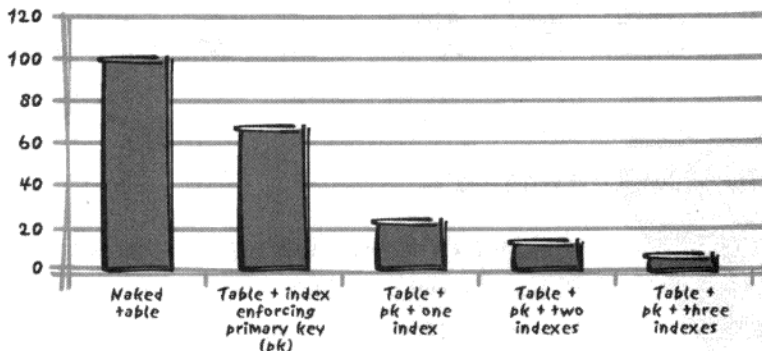


图 3-2：索引对插入操作的影响（采用 Oracle）

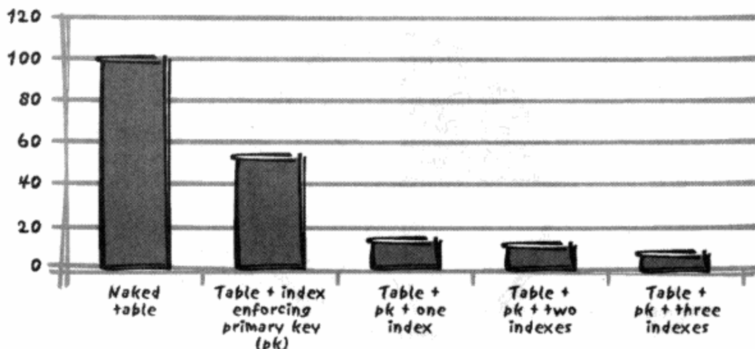


图 3-3：索引对插入操作的影响（采用 MySQL）

有趣的是，维护索引的开销与简单触发器带来的开销大致相当。例如，创建一个触发器，将每行数据的键、用户名和时间戳记录到日志表中——这是一个典型的审计跟踪（audit trail）功能，人们料定它会影响性能。但是，简单触发器对性能影响的程度，竟和增加两个索引相当（如图 3-4 所示）！还记得“为了提升性能而避免使用触发器”的呼声是何等强烈吗？当不愿意用触发器的人们使用索引的时候，殊不知两者造成的性能影响不相上下。

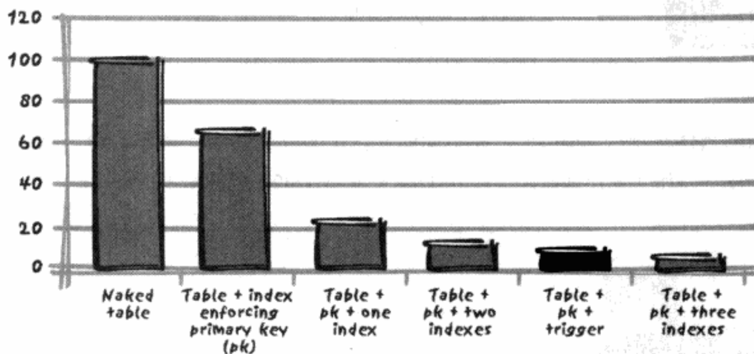


图 3-4：比较索引与触发器对性能的影响

索引产生了额外的开销，但这还不是索引影响性能的唯一方式。在大规模并发访问的环境中，过多索引会造成资源竞争（contention）和加锁（locking）。从本质上讲，索引的结构比表更紧凑——只要比较一下本书索引的页数与书的总页数即可理解这一点。更新带索引的表时，既要更新表中数据，又要更新索引数据。于是，并发更新可能涉及的区

域散布于表的各个部分，更改实际数据意味着一连串的冲击影响；另外，更新索引时能用的空间非常有限，正如先前解释的，索引被定义为“更紧凑的”数据集合。

无论存储和处理的开销多么大，我都要强调：索引始终是数据库中极重要的组成部分。正如第 6 章中将讨论的，对支持事务处理的数据库而言，大部分 SQL 语句要么查出少数几行记录，要么处理少数几行记录，此时索引对提升性能价值很大。第 10 章还将说明，决策支持系统（decision support system）的性能也在很大程度上依赖索引。然而，如果事务处理型数据库（transactional database）的表都符合范式的要求（这里再次提到设计的重要性），需要加索引的字段应该很少；这些表肯定有主键，在声明了主键字段（若为复合键则指主键中所有字段）后，它们会被自动加上索引；同样，具有唯一性的字段极可能在实现完整性约束时被加上索引；对于不具有唯一性、但接近唯一性的字段——换言之，该字段的值变化跨度比较大——也应考虑加上索引。

经验告诉我们，对于通用目的（general purpose）或事务处理型数据库而言，大部分表不需要加索引，因为许多表的查找是根据一组非常有限的条件来进行的。但在第 10 章中会看到，决策支持系统中建立索引的准则与上面的经验相去甚远。我开始对索引很多的表产生怀疑，特别是那些非常庞大且经常更新的表。不排除某些情况下增加大量索引很有必要，但我们应该根据设计的初衷判断大量增加索引的合理性。

总结：在事务处理型数据库中，“太多索引”往往意味着设计不够稳定。

索引与目录

Indexes and Content Lists

以书作为比喻，有助于更好地理解索引在 DBMS 中的作用。认清目录和索引这两种机制的区别很重要。两者虽然都提供快速访问数据的机制，但它们的粒度（granularity）差异很大。书的索引和数据库的索引相似，而目录提供整书的结构概览，起到与索引互补的作用。

在书中精确定位具体信息时，可求助于索引。我们常常借助索引查找两三项信息，如果要找 20 项信息，在索引页和正文间多次地翻来翻去会很低效。和书的索引一样，数据库索引帮助我们精确定位一条或多条记录的值（这里我忽略了范围搜寻也可以使用索引）。

为了在书中找到具体信息，可采用不同方法：既可以通过索引找到目标主题的第一个索引项，然后开始阅读；又可以利用目录，找到与主题最相关的章节。目录和索引之间的关键区别是：目录项指示正文块，可能是一章或是一节。第 5 章讲到的机制与目录方式类似，通过组织一个表来实现。

正如数据设计的初衷所规定的，索引是一种以原子粒度（atomic level of granularity）访问数据的手段，而不是为了检索大量数据的；否则，就严重误解了索引的作用。索引常被当作救命稻草，用于挽回已定的败局。此时的指挥官已开始恐慌，并下令全面突围，他希望以纯粹的数量优势来弥补由于失策带来的损失。当然，这于事无补。

总结：*你一定要非常清楚为哪些字段加索引，以及为什么为它们加索引。*

让索引发挥作用

Making Indexes Work

索引的使用是否合理，取决于它是否有用。就像前面提到的书的比喻，如果需要的是某个数据项中非常具体的信息，就可以使用索引；但如果是想阅读整个主题的内容，则应使用目录而不是索引。

有时，很难在使用索引还是目录之间做出选择。此时，考虑检索比率（retrieval ratios）有助于我们做出决定。对于许多 IT 和数据库的工作人员来说，检索比率有很大误导性，因为它太明确、太容易、太科学了！

长久以来，判断索引适用性的依据，就是用键值作唯一条件检索出数据的百分比。依惯例，通常是 10%。匹配某索引键（index key）的记录所占的平均百分比反映了索引的可

选择性（selectivity），百分比越低，索引的可选择性就越高。我们在文献中也常会发现这类规则。检索比率及其同类比率，都是以一些古老的假设作为前提的，比如磁盘访问和内存访问的相关性能。就算这些 1980 年代中期就开始使用的性能指标没有过时（理想的百分比把问题过分简单化了），使用索引时依然应该考虑多得多的因素。

当初设定这个“神奇”比率（例如上面提到的 10%）的时候，含有 500 000 行数据的表算是非常大了，通常它的 10%大致就是几万行数据。但当表中的数据有数亿行、甚至数十亿行时，用可选择性为 10% 的索引检索出的数据，常常比当初预想的“大表”的数据量还要大。

接下来讨论配备了大容量缓存的现代硬盘系统所起的作用。DBMS 所进行的“物理 I/O”很可能是指内存访问；此外，由于数据库核心通常会根据存取的不同类型（表或索引），分别把数量不等的数据搬到内存中来，所以使用索引后的性能提升会令人大吃一惊。仅仅考虑这些当然还不够，磁盘操作的次数也是需要考虑的因素之一，尽管它们确实能以并行方式执行。索引键值相关记录的物理位置是否邻近很重要，例如一个针对插入日期而建立的索引，只要没有特殊设置（如第 5 章中讲到的特殊存储设备选项发生变化），对特定范围内的插入日期进行查询总会返回物理上连续的一块空间。该日期范围中第一个键所指的数据块或存储页，很可能也包含了那些紧跟其后的键值所指的记录。因此，利用索引查询时所返回的表中任何一部分数据，都可能包含许多目标数据，从而由索引找到的任何一个数据块，对提高查询的性能都会有很大帮助。

建立了索引之后，如果索引键（index key）所指向的记录散布于整个表中（例如订单表中某订单项数据），情况就大不一样了。尽管这些记录在表中占的比率很小，但因为它们散布在整个磁盘上，所以索引的性能就会大打折扣。这可以由图 3-5 来说明：两个唯一性索引，在检索单独一行数据时性能完全一样，但检索一组数据时则效率相差很大，这种情况在处理日期时经常会发生。

上面提到的众多因素错综复杂，使我们决定如何使用索引变得困难。

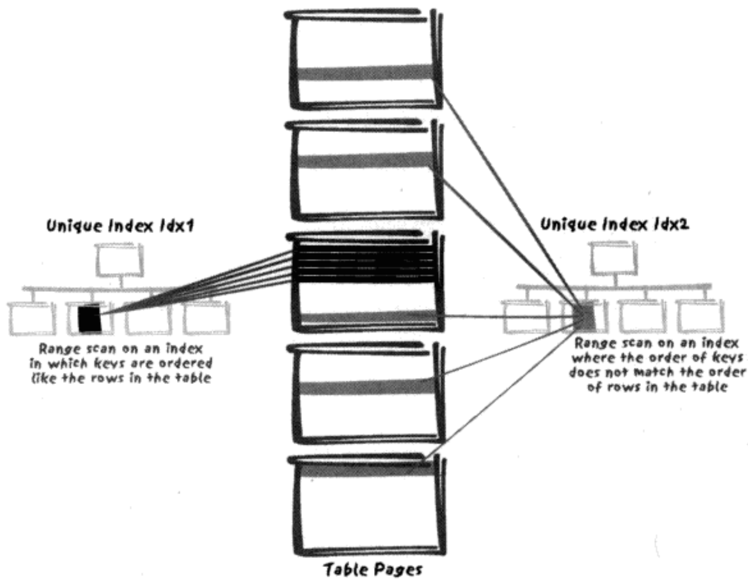


图 3-5：何时两个可选择性很高的索引的性能不同

总结：记录的排列顺序与索引键相同，可以提高范围（range）查询的速度。

函数和类型转换对索引的影响

Indexes with Functions and Conversions

索引通常使用树（tree）结构来实现——通常是复杂的树——以避免对表进行大量插入、更新及删除操作之后，索引快速退化（decay）。要找到一条数据记录的物理地址（该地址保持在索引中），就必须比较键值与保存在当前树节点中的值，以判断要递归搜索哪个子树。下面，让我们假设 SQL 语句中直接提供的不是要搜索的值，而是某字段的函数 `f()`。此时，表达查询条件的语句将是：

```
where f(indexed_column) = 'some value'
```

这种方式的搜索条件很典型，会使索引无法发挥作用。原因是，无法保证函数 `f()` 值的顺序与索引数据的相同——事实上，大多数情况下它们都不会相同。例如，假设索引的树结构如图 3-6 所示。

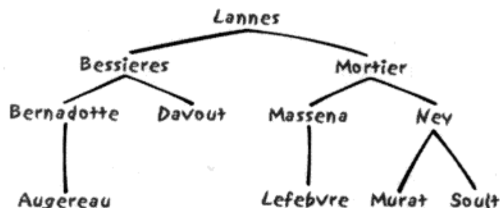


图 3-6：索引如何保存名字的简单示例

（图中的名字看起来很熟悉吧？他们都是拿破仑麾下的将领。）图 3-6 当然是个简化的示意图，实际的索引可能不会与之完全相同。假设用下面的条件来搜索键值 MASSENA：

```
where name = 'MASSENA'
```

索引的检索过程很简单。先在树的根节点找到 LANNES，并将它和 MASSENA 按字母序比较，结果前者比较大，于是开始递归搜索 LANNES 的右子树。右子树的根节点为 MORTIER，大于条件值 MASSENA，于是向下搜索其左子树。接下来，便找到了目标值 MASSEMA。大功告成。

但是，假设搜索条件如下：

```
where substr(name, 3, 1) = 'R'
```

此条件表示“名字的第三个字母是大写的‘R’”，返回结果应该是 BERNADOTTE、MORTIER 和 MURAT。访问索引时，首先找到的根节点 LANNES 不符合条件，接下来就出现了问题，当前树节点的值无法指示我们应该要递归搜索哪个分枝。我们不知所措：条件“第三个字母是 R”不能帮助我们判断该搜索左子树还是右子树（事实上两个子树都包含符合条件的数据）；我们也无法采用惯用的、比较条件值和当前树节点值的方式选择要继续搜索的分枝。

单就图 3-6 所示的索引来说，要找出所有第三个字母是 R 的名字必须顺序扫描全部数据，这会导致另一个问题：如果优化器设计非常完善，它能够判断最有效的方式是扫描实际的表、还是扫描为目标字段所建的索引。若为后者，将导致搜索必须根据索引提取数据，这并不是模型设计的初衷，因为这种使用索引的方式很低效。

回想一下第 1 章中讨论过的原子性（atomicity），不难发现此处的性能问题源自一个简单的事实：如果需要对字段使用函数，就意味着表中原子性的数据不符合业务需求。此时连 1NF 都不符合！

当然，原子性这个概念并不简单。日期的搜寻条件即为典型的例子。例如，Oracle 中的日期类型（即大多数数据库的 datetime），不但保存了日期信息，还存储了精确至秒的时间信息。然而，或许为了考验粗心的人，默认的日期格式并不显示时间信息。所以如果输入如下条件：

```
where date_entered = to_date('18-JUN-1815',
                             'DD-MON-YYYY')
```

则只有 datetime 为 1815 年 6 月 18 日 00:00（即午夜）的记录被返回！每个人在第一次查询 datetime 时，都会遭遇上面这个问题。很自然的，新手们的第一个想法就是屏蔽 data_entered 的时间信息，代码如下：

```
where trunc(date_entered) = to_date('18-JUN-1815',
                                    'DD-MON-YYYY')
```

这样，查询可以正常工作了，但许多人（在性能出现问题之前）没有意识到这种方式无法使用 `date_entered` 上的索引（假设有索引的话）。那么是否就不可能符合 1NF 了呢？很幸运，有办法符合 1NF。在第 1 章，我们在 `where` 子句中完整引用原子属性。现在，我们使用范围作为条件就可以完整引用日期字段，从而利用 `date_entered` 上的索引，代码如下：

```
where date_entered >= to_date('18-JUN-1815',
                              'DD-MON-YYYY')
  and date_entered < to_date('19-JUN-1815',
                             'DD-MON-YYYY')
```

这种方式可以利用 `date_entered` 上的索引，因为第一个条件能让检索沿索引树向下，最终到达位于索引树结构底端的链的有序队列（我们可以把索引看作由键及关联地址组成的有序队列，且该队列还专门配备了树结构，以支持对队列中每一项的直接访问）。因此，一旦第一个条件把我们带到了索引树的底层，且在有序队列中找到了第一个目标项时，只要该项也符合第二个条件，剩下的只是扫描该队列了。这类访问就是所谓的“索引范围扫描（index range scan）”。

有时，“函数导致索引失效”的影响会更大。例如，在 `where` 条件中，某类型的字段要与另一类型的常数作比较，而引起 DBMS 引擎进行隐式的数据类型转换时——虽然

SQL 允许这么做，但这么做逻辑上有问题。Oracle 再次提供了绝佳的例子：当字符型字段与一个数字进行比较时就很危险，Oracle 不是立即报运行时错误，而是先隐式地把字符转换成数值并继续比较操作；如果那个“数字字符串”中含有字母，转换操作才报“运行时错误”。但在许多情况下，数字字符串字段中保存的值并不代表真正的数值含义（如社会保险号码，或者以 mmddyy 或 ddmmyy 格式显示的生日，后者中两种格式代表同一日期，但转换成数值后差异不言而喻），所以虽然转换和比较操作都会“继续进行”，但那个字符串字段上的索引几乎毫无价值。

上例中，Oracle 的设计者选择了对字段而不是对那个常数进行类型转换。这似乎令人惊讶，但的确有些道理。首先，将马铃薯和红萝卜进行比较是个逻辑上的错误。DBMS 对字段进行转换时遇到不适合转换的值继而产生运行时错误的可能性较大(因执行方式而异)，而在此阶段发生的错误对开发者是个有益的提醒，告诉我们某些实际数据需要修正，数据质量还存在着恼人的问题。第二，假设没有报错，接下来我们最不想看到的就是返回不符要求的数据。如果遇到：

```
where account_number = 12345
```

写上面代码的人希望返回账号 0000012345。如果检索前 account_number（字符型字段）被转换为数值，而不是 12345 被转换为不具有任何特殊格式的字符串，结果就会如愿以偿。

或许，有人认为隐式类型转换在 DBMS 中很少发生，就像 Bug 很少发生一样。后者是事实，但隐式类型转换实际上却相当常见。例如，表 parameters 中有个名为 parameter_value 的字段，它以字符串表示数值、日期、文件名或其他任何常规字符组成的字符串值，在处理中用到这个字段时隐式转换将频繁发生。所以，为了避免隐患，一定要显式地用转换函数进行转换。

有时，为操作一个或多个字段的函数所产生的结果建立索引也是可能的。大部分数据库产品中都有此功能，名称各不相同——例如“函数索引（fnctional index)”、“基于函数的索引（function-based index)”、“索引扩展（index extension)”，甚至更直接的“计算字段上的索引（index on a computed column)”，等等。我个人认为，这类功能应该慎用，而且只作为程序代码无法修改时的备选方案。

前面已经提到，索引的存在会导致数据修改的开销增大。而调用函数将进一步导致索引无法改善性能——其实只会增加索引的总维护成本。正如前面那个 date_entered 的例子，建立基于函数的索引只是个偷懒的解决办法，应该以另一种方式重写查询。而且，“将函数用于字段”不能保证其精确度与“对原始字段进行等效查询”的精确度相同。例如，销售数据在线保存五年，且 sales_date 字段有索引。表面上看起来这样的索引效率还行，但不太值得为一个“提取日期字段中月份信息的函数”的结果建立索引，尤其是当每年有大量销售记录发生在圣诞期间时。必须经过非常仔细的评估，才能确定基于函数结果建立的索引是否有价值。

纯粹从设计角度看，函数暗中助长了某字段保存两个或多个分离数据项的做法。多数情况下使用函数索引（functional index）是为了提取字段中的部分数据。正如前面指出的，这样做违反了著名的 1NF，即违反了数据必须具有“原子性”的要求。在 select 选出项列表中使用 “原子性”不强的数据，危害还不算太大，但搜寻条件中使用这样的数据，其危害是致命的。

然而，有时采用基于函数的索引（function-based index）是合理的。不区分大小写的搜索是最好的例子。对转换为大写或小写的字段加上索引，能提高对该字段进行“大小写不敏感搜索”的效率。另外，在插入和更新期间强制要求大小写，也是个不错的解决方案。但不管怎样，数据以小写保存而后来却需要大写，则是最初数据库设计没有考虑周全。

另一个难对付的问题是“时间段”，因为缺少专门表示时间间隔的数据类型。假设有三个时间字段，分别为开始日期、完成日期及持续时间，其中任一个值都可以由另两个已存在的值决定 —— 但只有借助函数索引或不惜保存冗余数据来实现。无论使用哪个解决方案，冗余是不可避免的，确定方案之前必须权衡使用函数索引的各种利弊，以做出明智的决定。

总结：函数索引的使用通常暗示设计存在问题，数据模型甚至连基本的数据原子性问题都没有解决。

索引与外键

Indexes and Foreign Keys

系统地对表的外键加上索引的做法非常普遍，已被认为是常识。事实上，一些设计工具和 DBMS 也会这么做。然而，我力劝大家慎重！基于索引整体开销的考虑，为外键加上不必要的索引可能不妥，尤其是对具有很多外键的表。

注意

当然，如果 DBMS 自动为外键加索引，那就没有办法了。对于索引带来的不必要开销，你也只能听天由命。

考虑为外键加上索引，是因为表 A 的外键参照了表 B 的主键，于是两个表需要同时修改。以图 3-7 所示的模型为例。

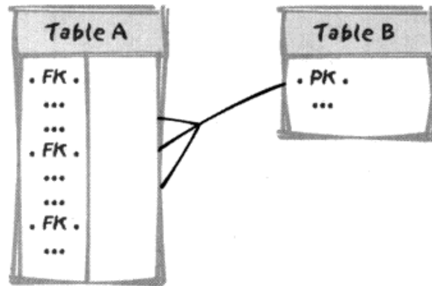


图 3-7：一个简单的主-从结构

想象一下表 A 非常大会发生什么情况。如果用户 U1 仅想从表 B 中删除一行记录，由于 B 的主键被 A 以外键参照，所以 DBMS 为了避免表间依赖的不一致性必须检查 A 是否包含参照了 B 中待删除记录的子记录（child row）。如果包含的话，删除操作将失败，否则会造成 A 中的子记录成为“孤儿”，破坏数据的一致性。当 A 的外键有索引时，上述检查执行得非常快，反之就会花费很长时间来扫描整个 A 表。

另一个问题是，数据库是为多用户服务的，扫描 A 表时还有很多工作要做。例如，用户 U1 刚开始搜寻表 A，用户 U2 就开始向表 A 中插入一条记录，而这条记录恰好与 B 中待删记录相参照。如图 3-8 所示。

（1）用户 U1 先访问表 B，检查待删记录的键。

（2）接下来，搜寻表 A 中的相关数据。

（3）与此同时，U2 检查表 B 中是否存在待插入记录的父记录（parent row）。由于 B 上有主键索引，所以不像 U1 顺序搜寻表 A 的外键那么慢，于是 U2 立即从表 B 中获得结果，并很快将新记录插入表 A。

（4）U2 提交改变时，U1 可能已完成检查，并得出错误结论——即没有找到子记录，于是删除 B 中记录的操作被“审核通过了”。

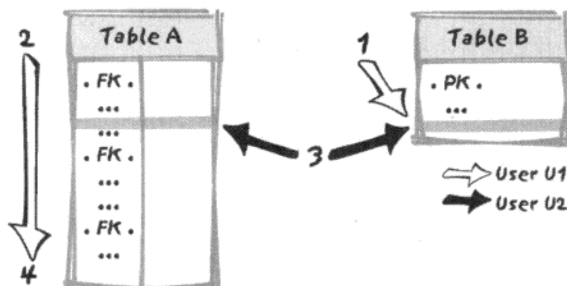


图 3-8：主键访问冲突

采用加锁（locking）的方法可以避免上述情况发生，否则将导致数据的不一致（inconsistent）。数据本应具有完整性（integrity），这也是企业级 DBMS 关注的基本问题之一。我们必须始终关注数据的完整性，每当要删除表 B 的数据时，应保证在查找子记录（child row）期间不向任何表中插入参照待删记录的数据。为此，可以采取两种方法：

- 将所有参照表（referencing tables）加锁。（较笨的方法）
- 将表 B 加锁，并让向参照表插入新数据的另一个进程（例如 U2）等待，直到解锁为止。可以对表、页或记录加锁，这取决于 DBMS 操作的粒度。（多数 DBMS 使用这个方法）

总之，如果外键没有索引，查找子记录就很慢，且参照表被锁的时间会很长，进而使很多更改操作阻塞。如果采用上面提到的“笨方法”，最糟时甚至发生死锁（deadlock）：两个进程互锁了对方需要访问的表，而且任一进程都不愿先解锁。在这种情况下，DBMS 通常会杀掉其中一个进程来解决死锁问题。

对数据进行并发更新时，外键加索引尤为必要，以避免加锁时间过长。于是，我们常听到“外键一定要加上索引”的说法。这样做的好处是每个进程所需的时间大幅缩短，从而也缩短了“为确保数据完整性而锁定表”的时间。

但人们忘了，“一定要对外键加索引”是在“特殊情况”下才成立的规则。有趣的是，这种“特殊情况”常因人们怪癖的设计而出现，例如“主从关系（master/detail relationship）”中的主表经常包含总计（summary）或聚合（aggregate）之类非规范化字段。要同时更新存在“参考完整性约束（referential integrity constraint）”的两个表，可以作为引入索引的充足理由。但我们必须注意，事务型数据库（transactional database）的被参照表常常是“只读的”——例如字典（dictionary）和查找表（look-up table）很少被修改，或在午夜没有其他活动时被修改——此时是否为外键加索引的判断标准是，增加索引是否真正为性能带来好处。切记索引的维护会对性能造成严重影响。其实，很多情况下无需为外键建立索引。

总结： 建立索引必须有理由。无论是对外键，或是其他字段，都是如此。

同一字段，多个索引

Multiple Indexing of the Same Columns

系统地为外键增加索引，常导致同一字段属于多个索引的情况。下面是个经典的例子：在订单系统中，表 `order_details` 记录了每张订单（用 `order_id` 字段标识，它是表 `orders` 的外键）中所包含的不同商品（用 `article_id` 标志，它是表 `articles` 的外键）及数量。关系表 `order_details` 建立了 `orders` 和 `articles` 表之间的多对多关系。如图 3-9 所示。

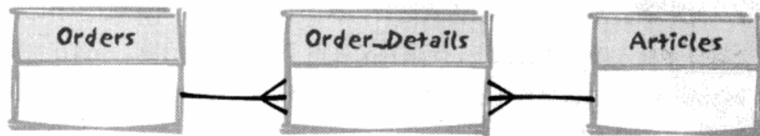


图 3-9：订单与商品的例子

通常，order_details 的主键是个复合键，由两个外键（order_id, article_id）组成。被参照表和参照表很可能同时更改——订单录入就是这种情况，所以必须为表 order_details 的外键 order_id 加上索引。但是，外键 order_id 作为表 order_details 复合主键的一部分，已经建立过索引了，并且它是复合主键的第一个字段。所以，无须另建索引，order_id 就可以享受到“建立专门索引”的所有好处。如果不是基于复合键的全部字段建立复合索引（composite index），但只要是复合键的开头部分，所建复合索引就是非常有用的。

向下搜寻索引树时（如本章前面的例子），只要能将键的头几个字符和索引节点做比较，就已经能够决定下一个要搜寻的分枝了。所以，没有必要单独对 order_id 加索引，操作表 orders 时 DBMS 会使用(order_id, article_id)上的索引来寻找子记录(child row)。从而，这两个表都不需要加锁。再次提醒，上述做法完全是因为 order_id 正好是复合主键的第一个字段。如果主键被定义为（article_id, order_id），我们就必须专门建立 order_id 上的索引，反而不用再为外键 artilce_id 专门建立索引了。

总结：为每个外键建立索引，可能会造成多余的索引。

系统生成键
System-Generated Keys

对于系统生成键（可以是一个自动递增的序列号，或像 Oracle 那样由系统计数器产生）要特别小心。即使能选择有实际意义的主键，一些经验不足的设计者还是偏爱使用系统生成键。由系统产生序列号，当然远比寻找当前最大值并加上 1 容易得多（但在并发程度较大的环境中会产生重复值），也远比用一个专用的表保存“下一个值”且加锁更新来得方便（此机制会使访问以串行方式进行，从而大大降低访问速度）。不过，如果同一个表会有许多插入操作并发执行，而该表又采用了自动生成的键，那么在主键索引（primary key index）的创建操作上会发生十分严重的资源竞争（contention）。主键索引的主要用途，就是确保主键的唯一性。

这里通常会有个问题，如果仅有一个生成器（而不是生成器的数量和并发进程数一样多，分别生成属于不同范围的值），那么生成的序列号就彼此非常接近。于是，将键值插入到主键索引中时所有进程将争用同一索引页，而 DBMS 引擎就必须串行化各进程——利用锁（lock）、闩（latch）、信号量（semaphore）或任何合适的锁定机制 —— 以避免任一进程覆写（overwrite）其他进程正在写入的数据。资源竞争会导致服务器对硬件的利用率非常低，这就是个典型例子。本可以并行执行（work in parallel）的进程，现在必须以串行方式执行。若在并行操作为主的多处理器服务器上，此瓶颈可相当严重。

某些数据库系统提供了一些手段，来降低系统生成键可能造成的影响。例如，Oracle 允许建立逆序索引（reverse index），保存索引之前先将组成键的 bit 序列（sequence of bits）进行逆向排列。为大致演示上述上述思想，下面以前面的将士名字为例，不过，我们没有对 bit 序列逆序，而是改为对字母序列逆序，即可得到图 3-10。

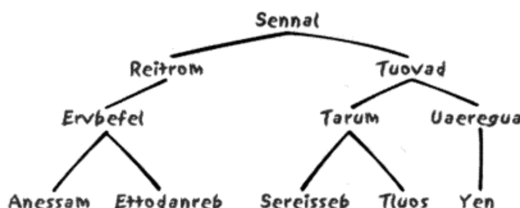


图 3-10：一个简单的逆向索引的例子

不难理解，即使插入按字母序彼此非常接近的名字，它们也会散布到索引树的各个分枝，例如图中 MASSENA（变成 ANESSAM）、MORTIER（REITROM）和 MURAT（TARUM）的位置。于是，插入时会找到索引中的不同位置，资源竞争也会比使用常规索引时更少。当然，执行搜索操作之前，Oracle 会同样逆序化你的待搜索条件，接下来就可以一如往常地在索引树中遍历了。

不过，是玉有瑕，如果搜索条件使用如下通配符：

```
where name like 'M%'
```

这是个典型的范围搜索（range search），它使逆序索引完全失效；相反，此时常规索引可以快速找出以特定字符串开头的值。当然，逆序索引对范围搜索无效，仅是使用系统

生成键的小小不足，所以大家一般都不知道。但是，如果是时间戳（timestamp）字段，问题就严重了：一方面，含时间戳的记录密集地接连出现，此时为时间戳字段建立逆序索引非常合适；但另一方面，又非常有可能对时间戳字段进行范围搜寻。

一些数据库系统使用哈希索引（hash indexing）避免“更新同一索引页”瓶颈。此时，系统根据索引字段的值，将实际键转换成一个系统产生的、无意义的随机数值键。尽管转换后的值还有可能相似，但原本“值接近”的键转换之后往往“相距甚远”。同样，哈希索引作为一种避免索引树内出现资源竞争点（hot spot）的手段，有优势也有劣势：哈希索引是以“相等与否”作为工作的基础，换言之，范围搜索及类似上述使用通配符的查询，哈希索引都不支持；然而，如果只是直接访问某一个健值，哈希索引非常快。

即使有办法降低资源竞争的风险，也不应过多使用系统生成键。我曾发现每个表都有系统生成键的情况（比如前例中表 order_details 的键改用 detail_id 字段，而不是 order_id 和订单号的组合）。这种草率的做法，无论如何都有点不合理，对不被任何其他表参照的表尤其不合理。

总结：正确使用系统生成键大有裨益，但切忌滥用！

索引访问的不同特点
Variability of Index Accesses

许多人认为，如果查询使用了索引就万事大吉了。这是个严重的误解，因为索引的效果要视索引访问的具体特点而定。显然，最高效的索引访问类型是唯一索引（unique index），此时最多只有一条记录与搜索值相符，通常这样的搜索是基于主键的搜索。但是，正如第 2 章所述，主键访问不适合需要循环检查一个表的所有键值的情况，这就像用小茶匙移走大沙堆，其实此时应使用全表扫描（full scan）这把大铁铲。所以，虽然唯一索引是战术层面最高效的索引，但其最终效果还要视综合情况而定，有时使用唯一索引是个代价不菲的错误。

当存在多条记录与“非唯一索引（non-unique index)”中的一个键相符时（或基于唯一索引搜索特定范围的值时），就会发生范围搜索。此时，从索引中检索出一组记录地址，它们对应的每个记录都包含我们要查找的键值。这个索引可能是“准唯一索引（near-unique index)”，即除了少数键值对应多条记录外，其他每个键值都只对应一条记录。极端情况下的非唯一索引，所有记录的索引字段（indexed column）值都相同，当前的一些软件包就是如此，大部分字段都加上了索引以防万一。别忘了，“通过索引定位记录”只是工作的一部分，除非：

1. 需要的就是索引键中的数据；

2. 索引没有压缩,否则，必须从表中找到实际的值才能确认。

除了上述情况外，我们的查询工作都才做了一半，还必须根据索引提供的地址访问每个数据块（或页）。重申一次，符合要求的记录是聚集在同一磁盘区域、还是散布在不同地方，对性能的影响较大，即使其他条件都相同。

上述描述适用于“常规”索引访问。然而，聪明的查询优化器可以选择另一种方式使用索引：

优化器可以操作多个索引，把它们结合起来，并在查找数据之前做某种“预过滤”操作；

优化器还可以决定对某个索引执行完整扫描，如果完整扫描是该查询最有效的执行方式的话（在此暂不展开讨论“最有效”的具体含义）；

优化器还决定是否系统地从索引中收集记录地址，免去每次向下搜寻索引树的麻烦。

所以，索引不是“能运行就万事大吉”。一些索引访问速度非常快，而另一些则极慢。即使某个查询的速度已经很快了，但可能另一个查询更快。此外，即便优化器聪明地绕过了一个无用的索引，但表的内容修改后该索引仍需立即更新，这种更新在大规模批处理式上载（upload）或清除（purge）操作时尤为重要。无论有用与否，索引的维护是必须的。

总结：*索引不是万灵药。充分理解要处理的数据，做出合理的判断，才能获得高效方案。*

第 4 章

机动灵活：思考 SQL 语句

Maneuvering:
Thinking SQL Statements

战争之道在于：出其不意，攻其要害，又快又狠。

——威廉·斯利姆元帅（1891—1970）
引自某军士长

本章我们将深入讨论 SQL 查询，并研究如何根据不同情况的具体要求，来编写 SQL 语句。我们会分析复杂的 SQL 查询语句，将它们拆解成小的语句片断，并讲解这些语句片断如何共同促成了最终查询结果的产生。

SQL 的本质
The Nature of SQL

在深入讨论如何编写 SQL 查询之前，我们有必要首先了解一些 SQL 自身的基本特性：SQL 与数据库引擎（database engine）和优化器（optimizer）是什么关系？哪些因素可能限制优化效率？

SQL 与数据库
SQL and Databases

关系数据库的出现，要归功于 E.F. Codd 的关系理论开创性研究成果。Codd 的研究成果为数据库学科提供了坚实的数学基础——而在此之前的很长一个时期数据库学科主要是凭经验。这和造桥的历史很相似：几千年前我们就开始建造跨江大桥，但是由于当时的营造商并不完全了解造桥材料和桥梁强度之间的关系，桥梁的设计往往会大大超出实际的要求；后来土木工程学的材料强度理论完善了，更先进更安全的桥梁也就随之出现，这表明造桥使用的各种建筑材料得到了充分利用。的确，如今的一些桥梁工程非常浩大，与此类似，现代 DBMS 软件能够处理的数据量之大也是今非昔比了。关系理论之于数据库，正如土木工程学之于桥梁。

SQL 语言、数据库和关系模型三者经常被混淆。数据库的功能主要是存储数据，这些数据符合对现实世界一部分所建立的特定模型。相应地，数据库必须提供可靠的基础设施（infrastructure），无论何时都能够让多个用户使用同一些数据，且在数据被修改时不破坏数据完整性。这要求数据库能够处理来自不同用户的“资源争用（contention）”，并能在事务（transaction）处理过程中遇到机器故障等极端情况下也保持数据一致性。当然，数据库还有很多其他的功能，本书并未涵盖。

正如其名，结构化查询语言（Structured Query Language，SQL）无非是一种语言，虽然它与数据库关系密切。将 SQL 语言和关系数据库等同视之，或者更糟——与关系理论等同视之，都是错误的。这种错误就好比将掌握了电子表软件或文字处理软件视为掌握

了“信息技术”。实际上，有些软件产品并非数据库，但它们也支持 SQL（注 1）。另外，SQL 在成为标准之前也不得不与诸如 RDO 或 QUEL 等其他语言竞争，这些语言曾被许多理论家认为优于 SQL。

为了解决所谓的“SQL 问题”，你必须了解两个相关部分：SQL 查询表达式和数据库优化器。如图 4-1 所示，这两部分在三个不同区域里协同工作。图的中央是关系理论，这是数学家们尽情发挥的区域。简而言之，关系理论支持我们通过一组关系运算符来搜寻满足某些条件的数据，这些关系运算符几乎支持任何基本查询。关键在于，关系理论有严格的数学基础，我们完全可以相信同一结果可由不同的关系表达式来获得，正如在算术中 246/369 完全等于 2/3 一样。

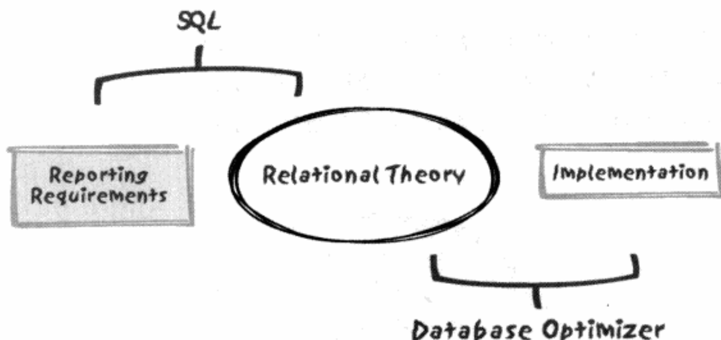


图 4-1：DBMS 的相关部分

然而，尽管关系理论有至关重要的理论价值，但一些有重要实践意义的方面它并未涉及，这些方面属于图中所示的“报告需求（reporting requirements）”的范围。其中最明显的例子就是结果集的排序：关系理论只关心如何根据查询条件取得正确的数据集；而对我们这些实践者（而非理论家）而言，关系操作阶段只负责准确无误地找出属于最终数据集的记录，而不同行的相同字段的关系并不是在这个阶段处理，而是完全属于排序操作。另外，关系理论并不涉及各种统计功能（例如百分位数等），而这些统计功能经常出

注 1：Sqlite 就是一例，它是著名的存储引擎，支持用 SQL 管理文件中的数据。Sqlite 不是数据库服务器。

现在不同的“SQL 方言（dialect）”当中。关系理论所研究的是集合（set），但并不涉及如何为这些集合排序。尽管有许多关于排序的数学理论，但它们都与关系理论无关。

必须说明的是，关系操作与上述“报告需求”的不同在于关系操作适用于理论上无限大的、数学意义上的集，无论是操作含有十行数据的表、一万行数据的表、还是一亿行数据的表，我们都能以相同的方式对其施以任何过滤条件。再次强调：当我们只关心找出并返回符合查询条件的数据时，关系理论是完全适用的；然而，当我们需要进行记录排序，或者执行一个大多数人错误地认为它是关系操作的 group 操作时，却已不再是针对可以无限大的数据集进行操作了，而必须是一个有限数据集，于是这个结果数据集不再是数学意义上的“关系（relation）”了，至此我们已经超出了关系操作层。当然，我们仍然可以利用 SQL 对该数据集进行一些有用的操作。

初步总结一下，我们可以将 SQL 查询表示为一个两层的操作，如图 4-2 所示。第一层是一个关系操作的“核”，它负责找出我们要操作的数据集；第二层是“非关系操作层（non-relational layer）”，它对有限的数据结果集进行“精雕细刻”从而产生用户期望的最终结果。

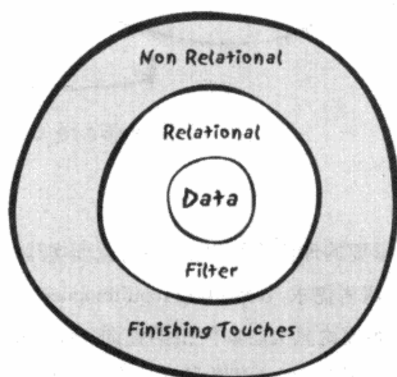


图 4-2：SQL 查询的不同层

尽管图 4-2 简要地表达了 SQL 在数据处理环境中的位置，但 SQL 查询在大多数情况下都比这要复杂得多，图 4-2 仅仅展示了一个总体的描述。关系操作中的过滤器（filter）有可能只是一个代名词，其背后是几个独立过滤器的组合，例如通过 union 结构或子查询来实现；最终，SQL 语句的构成可以很复杂。稍后还会讨论编写 SQL 语句的问题，但我们接下来首先要讨论的是数据物理实现和数据库优化器的关系。

总结：千万别把SQL查询的执行过程中真正的关系操作和附加的展现层（presentation layer）功能混为一谈。

SQL与优化器

SQL and the Optimizer

当SQL引擎处理查询时，会用优化器找出执行查询最高效的方式。此时关系理论又可以大有作为了——优化器借助关系理论，对开发者提供的语义无误的原始查询进行有效的等价变换，即使原始查询编写得相当笨拙。

优化是在数据处理真正被执行时发生的。经过变换的查询在执行时可能比语义上等效的其他查询快得多，这因是否存在索引，以及变换与查询是否适应而不同。在第5章我们将介绍各种数据存储模型；有时，特定存储模型决定了查询优化的方式。优化器会检查下列因素：定义了哪些索引、数据的物理布局、可用内存大小，以及可用于执行查询任务的处理器数。优化器还很重视查询直接或间接涉及的表和索引的数据量。最终，优化器根据数据库的实际实现情况对理论上等价的不同优化方案做出权衡，产生有可能是最优的查询执行方案。

然而，要记住的关键一点是，尽管优化器在SQL查询的“非关系操作层”也偶有用途，但以关系理论为支柱的优化器主要用于关系操作层。SQL查询的等价变换还提醒我们：SQL原本就是一种声明性语言（declarative language）。换言之，SQL应该是用来表达“要做什么”、而非“如何来做”的。理论上讲，从“要做什么”到“如何来做”的任务就是由优化器来完成的。

在第1章、第2章中讨论的SQL查询比较简单，但即使从编写技巧层面来说，拙劣的查询语句也会影响优化器的效率。切记，关系理论的数学基础为数据处理提供了非常严谨的逻辑支持，因此SQL艺术本应注重减小“非关系操作层”的厚度，即尽量在关系操作层完成大部分处理，否则优化器在“非关系操作层”难以保证返回的结果数据和原始查询执行的结果一样。

另外，在执行非关系操作时（这里非关系操作不严格地定义为针对已知结果集的操作），应专注于操作那些解决问题所必需的数据，不要画蛇添足。和当前记录不同，有限数据集必须以某种方式进行临时存储（内存或硬盘），这会带来惊人的开销。随着结果集数据量的增大，这种开销会急剧加大，尤其是在主存所剩无几的时候。主存不足会引发硬盘数据交换等开销很高的活动。而且，别忘了"索引所指的是硬盘地址，并非临时存储地址"，所以数据一旦进行临时存储，就意味着我们向最快的数据访问方式说再见了（哈希方式可能例外）。

一些 SQL 方言会误导用户，使他们认为自己仍在关系世界中——但其实早就不是关系操作了。举个简单的例子：不是经理的员工当中，哪五个人收入最高？这是个现实生活中很合理的问题，但它包含了明显的非关系描述。"找出不是经理的员工"是其中的关系操作部分，由此获得一个有限的员工集合，然后排序。有些 SQL 方言通过在 select 语句中增加特殊子句来限制返回的记录数，很显然，排序和限制记录数都是非关系操作。其他 SQL 方言（这里主要是指 Oracle）则采用另外的机制，即用一个名为 rownum 的虚拟字段（dummy column）为查询结果编号——这意味着编号工作发生在关系操作阶段。如果查询语句如下：

```
select empname, salary
from employees
where status != 'EXECUTIVE'
  and rownum <= 5
order by salary desc
```

乍一看好像没问题，但输出结果却不符合要求，并没有返回不是经理的人中"收入最高的五位"，而是返回不是经理的人中"最先被查到的五位"，以收入递减序返回。他们可能恰好是"收入最低的五位"！（这是 Oracle 实践者都知道的陷阱，大家都中过招。）

现在分析一下上面的代码。查询的关系操作部分仅从 employees 表中，以完全不可知的顺序，取出最先发现的五位非经理人员（只包含 empname 和 salary 字段）。别忘了关系理论指出，关系（以及描述关系的表）是无序的，关系中的元组（即记录）可以被存储

或检索。上面的查询执行后，收入最高的非经理人员或许在查询结果中，或许不在，无从知道查询结果是否满足查询条件。

正确的操作是：找出所有的非经理人员，以收入递减排序，然后返回前五条记录。代码如下：

```
select *
from (select empname, salary
      from employees
      where status != 'EXECUTIVE'
      order by salary desc)
where rownum <= 5
```

那么，这个查询是如何分层执行的呢？很多人错误地认为对一个排序的结果集进行过滤，如图 4-3 所示。

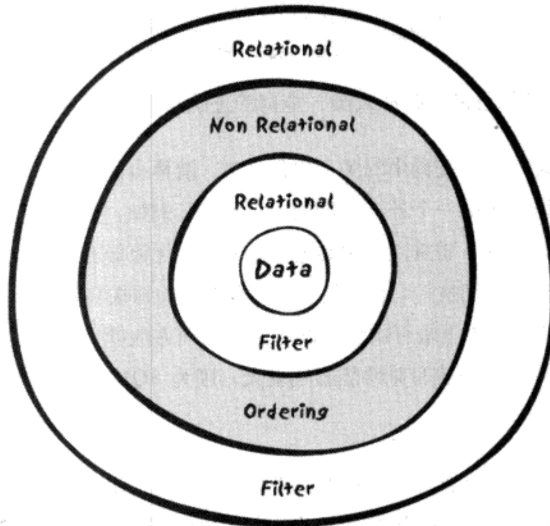


图 4-3：“收入最高的前五位非经理人员”查询的错误理解

其实，正确的理解应该如图 4-4 所示。

看来，有些看似关系的概念其实并不属于关系操作的范畴，因为关系操作必须要有关系操作符的参与。上面的子查询用了 order by 为结果集排序，而一旦用了排序操作，该数据集就已经不是关系了（关系是无序的）。于是外层的 select 看似关系操作，但其实是对一个内嵌视图的结果集进行操作，其中的 order by 子句早已不是关系操作了。

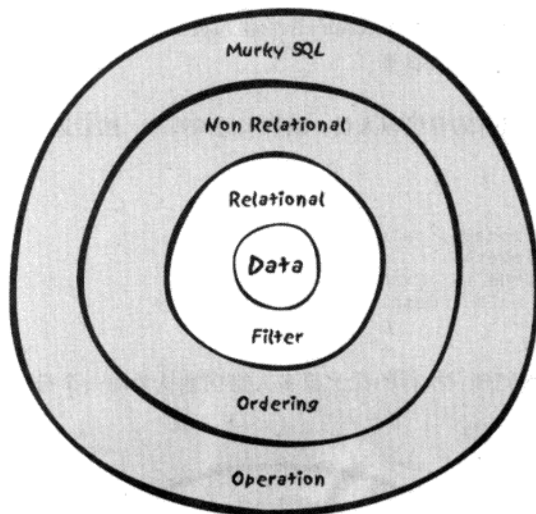


图 4-4：“收入最高的前五位非经理人员”查询的正确理解

上例虽然简单，但说明一旦查询中的关系操作结束，就再也回不去了。解决该问题最好的办法是：把查询结果传给一个外部查询的关系操作。例如：五个收入最高的非经理人员属于哪些个部门？然而，需重点强调的是，此时无论优化器有多聪明，它都不会合并两个查询，而是按顺序分别执行它们。此外，中间查询的结果集将暂时放在临时存储设备中，可以是内存或硬盘，于是可供优化器选择的访问方法就受到了限制。一旦离开了纯关系操作层，查询语句的编写对性能影响重大，因为 SQL 引擎将严格执行它规定的执行路径。

总而言之，最稳妥的办法就是在关系操作层完成尽量多的工作，因为关系操作层的执行可以优化。对于不完全是关系操作的 SQL 部分，应加倍留意查询的编写。掌握 SQL 语言的关键就是要懂得它有双重特性。如果你只把 SQL 看作是一把“单刃剑”，说明你太重视具体技巧了，无法深入理解高难度的 SQL 问题是如何解决的。

总结：为了取得好的优化效果，应将大部分工作安排在关系层。

优化器的有效范围
Limits of the Optimizer

优秀的 SQL 引擎非常强调优化器的作用，以确保性能优化方面的出色表现。然而，应牢记优化器在工作方式方面的一些特点。

优化器需借助在数据库中找到的信息。

这样的信息有两种类型：普通统计数据（须确保数据合适）、数据定义中重要的声明性信息。如果错误地将反映数据关系的重要语义信息写在触发器程序中，甚至写在应用代码中，会导致优化器无法利用这些重要信息，势必影响到优化器的优化效果。

能够进行数学意义上的等价变换，优化效果才能最佳。

对于查询中的非关系部分，优化器可借助的理论基础不多，优化结果和原始语句有意无意指定的方式相差无几。

优化器考虑整体响应时间。

比较大量执行方式的备选方案要花时间。最终用户只看到总共花费的时间，并不知道优化处理和查询执行各花了多少时间。优化器很聪明，对于预期耗时很长的查询执行，优化器会多花一些时间（当然有个上限）来改善性能。如果是个 20 路关联（20-way join）——这在一些应用中并不稀奇——就比较麻烦，优化器必须让步，因为要考虑的组合情况太多了，何况还要同时考虑合成视图和子查询。所以，一个独立执行的查询，优化效果可能非常好；但若把它嵌入到一个更复杂的查询内部时，其优化效果可能不佳。

优化器改善的是独立的查询。

然而，优化器无法使独立的查询联系起来。所以，通过过程性编程提取数据，而后将数据传递给后续查询，优化器就无法进行优化了。

总结：如果是若干个小查询，优化器将个个优化；如果是一个大的查询，优化器会将它作为一个整体优化。

掌握 SQL 艺术的五大要素

Five Factors Governing the Art of SQL

本章的第一节已详细讨论了 SQL 包含的关系和非关系特性，及其对优化器有效工作的影响。带着来自第一节的经验教训，接下来我们将集中讨论使用 SQL 时必须考虑的关键因素。依我看来，有五大要素：

- 获得结果集所需访问的数据量
- 定义结果集所需的查询条件
- 结果集的大小
- 获得结果集所涉及的表的数量
- 多少用户会同时修改这些数据

数据总量

Total Quantity of Data

必须访问的数据总量，是要考虑的最重要因素。一个查询方案，用于只有 14 行数据的 `emp` 表和 4 行数据的 `dept` 表时表现非常出色，但它可能完全不适用于有 1 500 万行数据的 `financial_flows` 表与有 500 万行数据的 `products` 表的 join 操作。注意，以许多公司的标准来看，1 500 万行的表并不算特别大。所以结论是，没有确定目标容量之前，很难断定查询执行的效率。

定义结果集的查询条件

Criteria Defining the Result Set

在编写 SQL 语句时，多数情况下会涉及 `where` 子句的条件，而在子查询或视图（普通视图或内嵌视图）中可能有多个 `where` 子句。然而，过滤条件的效率有高有低，这会受到其他因素的极大影响，例如物理实现（将在第 5 章中讨论）及要访问的数据量等因素。

为了定义结果集，必须从几个方面来考虑，包括过滤、主要 SQL 语句，以及庞大的数据量对查询的影响等。这是个复杂的问题，须做深度探讨，详见本章“过滤”一节。

结果集的大小

Size of the Result Set

查询所返回的数据量（或是 SQL 语句改动的数据量），是个重要且常被忽略的因素。一般而言，这取决于表的大小和过滤条件的细节，但不都是这样。典型的情况是，若干个独立使用时效率不高的条件，结合起来使用时会产生极高的效率；例如，以“是否获得理工科或文科学位”作为查询学生姓名的条件，结果集会非常大，但如果同时使用这两个条件（获得这两个学位），则产生的结果集就会大幅缩小。

从技术的角度来看，查询结果集的大小并不重要，重要的是最终用户的感觉。用户的耐心，在很大的程度上和预期返回的记录条数有关：用户只检索一条记录，则他期望非常快，他不会关心整个数据库有多大。更极端的例子是，查询之后并未返回任何结果：好的开发者都会努力使返回少量记录或不返回记录的查询尽量快，因为对用户而言，最令人沮丧的事莫过于等待了数分钟后，看到“无相符数据”的结果；若是按下回车键后马上察觉查询语句有误，而又无法终止查询，等待就更为恼人。最终用户情愿等待的，是预期返回大量数据时。如果把每个过滤条件定义的特定结果集看作中间结果，而最终结果是它们的交集（在条件中用 and 相连）或并集（在条件中用 or 相连），那么小型中间结果集的交集很可能为空。换言之，更精确的条件经常是零结果集产生的主要原因。无论何时，只要查询有可能返回零结果集时，都应该先检查那个最大可能导致空结果集的条件——尤其是在该检查执行非常快捷时。不用说，条件的顺序与条件所在上下文的关系十分密切，这在稍后“过滤”一节中讲述。

总结：熟练的开发者应该努力使响应时间与返回的记录数成比例。

表的数量

Number of Tables

查询中涉及的表的数量，自然会对性能有所影响。这不是因为 DBMS 引擎不能很好地执行连接操作——恰恰相反，现代的 DBMS 都能非常高效地连接很多表。

认为连接效率不高的想法，来自另一个对关系数据库的成见。通常的说法是不该连接太多表，建议的上限是 5 个。事实上，连接 15 个表也一样可以极高效地执行。但在连接大量表时，会产生一些额外的问题。

- 当需要连接多个表时（例如 15 个），按常理你就应该质疑设计的正确性。回忆一下第 1 章的内容—— 表的一条记录陈述了某个事实，而且可以将它比作数学的公理，通过连接表的操作，可衍生出其他事实。但要清楚一点，即哪些是显而易见的事实，可以称为公理；哪些是较不明显的事实，必须推衍得到。如果我们需要花大量时间来推衍事实，或许最初选择的公理就不合适。

- 对于优化器来说，随着表数量的增加，复杂度将呈指数增长。再次提醒，统计优化器通常有出色的表现，但同时其耗时在查询总响应时间中的比例也很高，尤其是在查询第一次执行时。如果表比较多，让优化器分析所有可能的查询路径，是非常不切实际的。除非查询语句是为方便优化器刻意编写的，否则，查询越复杂，优化器越容易“押错宝（bet on the wrong horse）”。

- 编写涉及许多表的复杂查询时，若可以用好几种截然不同的方式进行连接，最终选择失误的几率很高。如果我们连接表 A、B、C 和 D，优化器可能没有足够的信息判断出 A 直接与 D 连接的效率会很高。想以 distinct 解决记录重复问题的开发者，也常会遗漏连接条件。

复杂查询与复杂视图

我们必须明白，表面上看到的参与查询的表的数量可能不真实，有些表实际上是视图，它们有时很复杂。和查询一样，视图的复杂程度也差异极大。视图可以屏蔽字段、记录、甚至是字段和记录的组合，只让少数有权限的用户可以访问。视图从特定视角反映数据，从表的现存关系中推衍出新的关系。此时，视图可以看作查询的简略表达方式，这是视图最常见的用途之一。随着查询复杂度的增加，似乎应该把查询拆成一系列独立视图，每个视图代表复杂查询的一部分。

总结：表明简单的查询背后，可能隐藏着复杂的视图。

不要走极端，完全不使用视图也不合理，一般它们并无坏处。然而，将视图用在复杂查询中时，我们多半只对视图返回数据中的一小部分感兴趣——可能是几十个字段中的几个字段——这时，优化器会试图将简单视图重新并入一段更大的查询语句中。但是，一旦查询复杂到一定程度，此方法就太复杂了，以至于难以保证效率。

在某些情况下，视图的编写方式，能有效地预防优化器把它并入上级语句中。我已提过 `rownum`，那是 Oracle 使用的虚拟字段，用来显示记录最初被查到时的顺序。如果在视图中使用 `rownum`，复杂性会进一步增加。任何想把参照了 `rownum` 的视图并入上级查询中的尝试，都会改变后续 `rownum` 的顺序，所以此时不允许优化器改写查询。于是，复杂查询中这种视图将独立执行。DBMS 优化器常把视图原样并入语句中，把它当成语句执行的一步来运行（注 2），而且只使用视图执行结果中所需要的部分。

视图中执行的操作（典型的例子是通过 join 获取 ID 号对应的描述信息），往往与其所属查询的上下文无关；或者，查询条件很特殊，会淘汰组成视图的一些表。例如，对若干个表进行 union 得到的视图，代表了多个子类型，而上级查询的过滤器只针对其中一个子类型，所以 unio 其实是不必要的。将“视图”与“视图中出现的表”进行 join 也有危险，这会强制多次扫描该表并多次访问相同记录，但其实只扫描一次就足够了。

当视图返回的数据远多于上级查询所需时，放弃使用该视图（或改用一个较简单的视图），通常可使效率大为改善。首先，用 SQL 查询取代主查询中用到的视图。对视图

注 2：有时，优化器也会将查询条件“下推”到视图中执行。

的组成部分有了整体的了解之后，要去除严格意义上不必要的部分就容易多了。改用较简单视图的效果也不错，从查询中去除了不必要部分，执行速度快多了。

许多开发者不愿在复杂查询中，再引入复杂的视图，他们认为这会使情况更为复杂。推导与分解复杂的 SQL 表达式的确有点令人生畏，不过，和高中时常做的数学表达式推导也差不多。在我看来，这有助于形成良好的编程风格，值得花些时间去掌握。对于渴望提高编程技巧的开发者来说，研究上述技巧有利于对查询内部工作原理的深入了解，常常使你受益匪浅。

总结：当视图返回不必要的元素时，别把视图内嵌在查询中，而是应将视图分解，将其组成部分加到查询主体中。

并发用户数

Number of other Users

最后，在设计 SQL 程序时，并发性（concurrency）是个必须认真对待的因素。写数据库时需要关注并发性：是否存在数据块访问争用（block-access contention）、阻塞（locking）、或闩定（latching）（DBMS 内部资源阻塞）等重要问题；甚至有时，为保证读取一致性（read consistency）也会导致某种程度的资源争用。任何服务器的处理能力都是有限的，不管其说明书有多令人震撼。在机器相同的情况下，很少并发或没有并发操作时设计可能是完美的，但对有大量并发操作的情况未必完美。排序操作可能没有足够内存可用，于是转而求助于磁盘，引发了新的资源争用……一些计算密集型（CPU-intensive）操作——例如负责复杂计算的函数、索引区块的重复扫描，均可引起计算机负荷过多。我遇到过一些案例，增加物理 I/O 会使任务执行效率更高，因为其中计算密集操作的并发执行程度很高，一个进程刚因等待 I/O 而阻塞，被释放的 CPU 就被另一个进程占用了，这样一来 CPU 资源就被充分利用了。一般而言，我们必须考虑特定商业任务的整体吞吐量（throughput），而不是个别用户的响应时间（response-time）。

注意

第 9 章将更详细地探讨并发性。

过滤
Filtering

如何限定结果集是最关键的因素之一，有助于你在编写 SQL 语句时判断该用哪些技巧。用来过滤数据的所有准则，通常被视为是 where 子句中各种各样的条件，我们必须认真研究各种 where 子句（及 having 子句）。

过滤条件的含义
Meaning of Filtering Conditions

若从 SQL 语法来看，where 子句中表达的所有过滤条件当然大同小异。但事实并非如此。有些过滤条件通过关系理论直接作用于 select 运算符：where 子句检查记录中的字段是否与指定条件相符。但其实，where 子句中的条件还可以使用另一个关系运算符 join。自从 SQL92 出现 join 语法后，人们就试图将“join 过滤条件”（位在主 from 子句和 where 子句之间）和“select 过滤条件”（位于 where 子句内）区分开来。从逻辑上讲，连接两个（或多个）表建立了新的关系。

下面是个常见的连接（join）的例子：

```
select .....
from t1
   inner join t2
      on t1.join1 = t2.joind2
where ...
```

假设表 t2 中有一字段 c2，该不该把 c2 上的条件当作 inner join 的额外条件呢？即是否应认为参与连接的不是“t2 表”而是“t2 表的子集”呢？或者，假设 where 子句中有一些关于 t1 字段的条件，那么这些条件是否会应用到 t1 连接 t2 的结果呢？连接条件放在何处应该都一样，然而其运行效率却会因优化器不同而异。

除了连接条件和简单的过滤条件之外，还有其他种类的条件。例如，规定返回的记录集为某种子类型的条件，以及检查另一个表内是否存在特定数据的条件。虽然从 SQL 语法上看它们相似，但在语义上却未必完全相同。有时条件的计算顺序无足轻重，但有时却影响重大。

下面的例子说明了条件计算顺序的重要性，实际上，在许多商用软件包中都能找到这样的例子。假设有个 parameters 表，它包含字段：parameter_name、parameter_type、

parameter_value，无论由 parameter_type 定义了什么参数属性，其中 parameter_value 均以字符串表示。（从逻辑上来说，上述情况堪比罗密欧与茱莉叶的悲剧，因为属性 parameter_value 所表示的领域类型非常多，所以违反了关系理论的主要规则。）假设进行如下查询：

```
select * from parameters
where parameter_name like '%size'
  and parameter_type = 'NUMBER'
```

在这个查询中，无论先计算两个条件中的哪一个，都无关紧要。然而，如果增加了以下条件，计算的顺序就变得非常重要了，其中 int()是将字符转换为整数值的函数：

```
and int(parameter_value) > 1000
```

这时，parameter_type 上的条件必须先计算，而 parameter_value 上的条件后计算，否则会因为试图把非数字字符串转换为整数，而造成运行时错误（假设 parameter_type 字段的类型定义为 char）。如果你无法向数据库说明这一点，那么优化器也无从知道哪个条件应该有较高的优先权。

总结：查询条件是有差异的，有的好，有的差。

过滤条件的好坏
Evaluation of Filtering Conditions

编写 SQL 语句时，应首先考虑的问题是：

- 哪些数据是最终需要的，这些数据来自哪些表？
- 哪些输入值会传递到 DBMS 引擎？
- 哪些过滤条件能滤掉不想要的记录？

然而要清楚的是，有些数据（主要是用来连接表的数据）可能冗余地存储在几个表中。所以，即使所需的返回值是某表的主键，也不代表这个表必须出现在 from 子句中，这个主键可能会以外键的形式出现在另一个表中。

在编写查询之前，我们甚至应该对过滤条件进行排序，真正高效的条件（可能有多个，涉到不同的表）是查询的主要驱动力，低效条件只起辅助作用。那么定义高效过滤条

件的准则是什么呢？首先，要看过滤条件能否尽快减少必须处理的数据量。所以，我们必须倍加关注条件的编写方式，下面通过例子说明这一点。

蝙蝠车买主

假设有四个表：customers、orders、orderdetail、articles，如图 4-5 所示。注意，图中各表的方框大小不同，这代表各表的数据量大小，而不代表字段数量；加下划线的字段为主键。

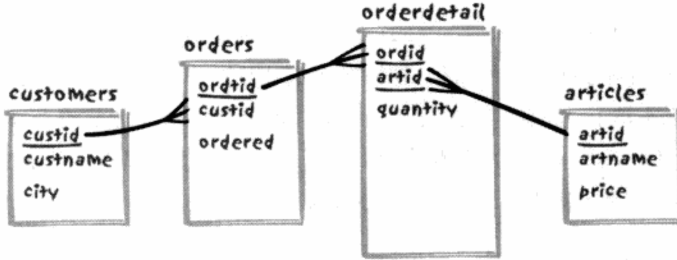


图 4-5：经典的订单 Schema

现在假设 SQL 要处理的问题是：找出最近六个月内居住在 Gotham 市、订购了蝙蝠车的所有客户。当然，编写这个查询有多种方法，ANSI SQL 的推崇者可能写出下列语句：

```
select distinct c.custname
from customers c
    join orders o
      on o.custid = c.custid
    join orderdetail od
      on od.ordid = o.ordid
    join articles a
      on a.artid = od.artid
where c.city = 'GOTHAM'
  and a.artname = 'BATMOBILE'
  and o.ordered >= somefunc
```

其中，somefunc 是个函数，返回距今六个月前的具体日期。注意上面用了 distinct，因为考虑到某个客户可以是大买家，最近订购了好几台蝙蝠车。

暂不考虑优化器将如何改写此查询，我们先看一下这段代码的含义。首先，来自 customers 表的数据应只保留城市名为 Gotham 的记录。接着，搜索 orders 表，这意味着 custid 字段最好有索引，否则只有通过排序、合并或扫描 orders 表建立一个哈

希表才能保证查询速度。对 orders 表，还要针对订单日期进行过滤：如果优化器比较聪明，它会在连接（join）前先过滤掉一些数据，从而减少后面要处理的数据量；不太聪明的优化器则可能会先做连接，再作过滤，这时在连接中指定过滤条件利于提高性能，例如：

```
join orders o
  on o.custid = c.custid
  and a.ordered >= somefunc
```

即使过滤条件与连接（join）无关，优化器也会受到过滤条件的影响。例如，若 orderdetail 的主键为（ordid, artid），即 ordid 为索引的第一个属性，那么我们可以利用索引找到与订单相关的记录，就和第 3 章中讲的一样。但如果主键是（artid, ordid）就太不幸了（注意，就关系理论而言，无论哪个版本都是完全一样），此时的访问效率比（ordid, artid）作为索引时要差，甚至一些数据库产品无法使用该索引（注 3），唯一的希望就是在 ordid 上加独立索引了。

连接了表 orderdetail 和 orders 之后，来看 articles 表，这不会有问题，因为表 orderdetail 主键包括 artid 字段。最后，检查 articles 中的值是否为 Batmobile。查询就这样结束了吗？未必结束，因为用了 distinct ，通过层层筛选的客户名还必须要排序，以剔除重复项目。

分析至此，可以看出这个查询有多种编写方式。下面的语句采用了古老的 join 语法：

```
select distinct c.custname
from customers c,
     orders o,
     orderdetail od,
     articles a
where c.city = 'GOTHAM'
  and c.custid = o.custid
  and o.ordid = od.ordid
  and od.artid = a.artid
  and a.artname = 'BATMOBILE'
  and o.ordered >= somefunc
```

注 3：一种名为“跳跃式扫描（skip-scan）”的特性支持索引搜索。

本性难移，我偏爱这种较古老的方式。原因只有一个：从逻辑的角度来看，旧方法突显出数据处理顺序无足轻重这一事实；无论以什么顺序查询表，返回结果都是一样的。customers 表非常重要，因为最终所需数据都来自该表，在此例中，其他表只起辅助作用。注意，没有适用于所有问题的解决方案，表连接的方式会因情况不同而异，而决定连接方式取决于待处理数据的特点。

特定的 SQL 查询解决特定的问题，而未必适用于另一些问题。这就像药，它能治好这个病人，却能将另一个病人医死。

蝙蝠车买主的进一步讨论

下面看看查询蝙蝠车买家的其他方法。我认为，避免在最高层使用 `distinct` 应该是一条基本规则。原因在于，即使我们遗漏了连接的某个条件，`distinct` 也会使查询“看似正确”地执行——无可否认，较旧的 SQL 语法在此方面问题较大，但 ANSI/SQL92 在通过多个字段进行表的连接时也可能出现问题。发现重复数据容易，但发现数据不准确很难，所以避免在最高层使用 `distinct` 应该是一条基本规则。

发现结果不正确更难，这很容易证明。前面使用 `distinct` 返回客户名的两个查询，都可能返回不正确结果。例如，如果恰巧有多位客户都叫“Wayne”，`distinct` 不但会剔除由同个客户的多张订单产生的重复项目，也会剔除由名字相同的不同客户产生的重复项目。事实上，应该同时返回具唯一性的客户 ID 和客户名，以保证得到蝙蝠车买家的完整清单。在实际中，发现这个问题可不容易。

要摆脱 `distinct`，可考虑以下思路：客户在 Gohtam 市，而且满足存在性测试，即在最近六个月订购过蝙蝠车。注意，多数（但非全部） SQL 方言支持以下语法：

```
select c.custname
from customers c
where c.city = 'GOTHAM'
  and exists (select null
              from orders o,
                   orderdetail od,
                   articles a
              where a.artname = 'BATMOBILE'
                and a.artid = od.artid
                and od.ordid = o.ordid
                and o.custid = c.custid
                and o.ordered >= somefunc )
```

上例的存在性测试，同一个名字可能出现多次，但每个客户只出现一次，不管他有多少订单。有人认为我对 ANSI SQL 语法的挑剔有点苛刻（指“蝙蝠车买主”的例子），因为上面代码中 customers 表的地位并没有降低。其实，关键区别在于，新查询中 customers 表是查询结果的唯一来源（嵌套的子查询会负责找出客户子集），而先前的查询却用了 join。

这个嵌套的子查询与外层的 select 关系十分密切。如代码第 11 行所示（粗体部分），子查询参照了外层查询的当前记录，因此，内层子查询就是所谓的关联子查询(correlated subquery)。此类子查询有个弱点，它无法在确定当前客户之前执行。如果优化器不改写此查询，就必须先找出每个客户，然后逐一检查是否满足存在性测试，当来自 Gotham 市的客户非常少时执行效率倒是很高，否则情况会很糟（此时，优秀的优化器应尝试其他执行查询的方式）。

我们还可以这样编写查询：

```
select custname
from customers
where city = 'GOTHAM'
  and custid in
           (select o.custid
            from orders o,
                 orderdetail od,
                 articles a
            where a.artname = 'BATMOBILE'
              and a.artid = od.artid
              and od.ordid = o.ordid
              and o.ordered >= somefunc)
```

在这个例子中，内层查询不再依赖外层查询，它已变成了非关联子查询（uncorrelated subquery），只须执行一次。很显然，这段代码采用了原有的执行流程。在本节的前一个例子中，必须先搜寻符合地点条件的客户（如均来自 GOTHAM），接着依次检查各个订单。而现在，订购了蝙蝠车的客户，可以通过内层查询获得。

不过，如果更仔细地分析一下，前后两个版本的代码还有些更微妙的差异。含关联子查询的代码中，至关重要的是 orders 表中的 custid 字段要有索引，而这对另一段代码并不重要，因为这时要用到的索引（如果有的话）是表 customers 的主键索引。

你或许注意到，新版的查询中执行了隐式的 distinct。的确，由于连接操作，子查询可能会返回有关一个客户的多条记录。但重复项目不会有影响，因为 in 条件只检查该项目是否出现在子查询返回的列表中，且 in 不在乎某值在列表中出现了一次还是一百次。但为了一致性，作为整体，应该对子查询和主查询应用相同的规则，也就是在子查询中也加入存在性测试：

```
select custname
from customers
where city = 'GOTHAM'
  and custid in
           (select o.custid
            from orders o
            where o.ordered >= somefunc
              and exists (select null
                          from orderdetail od,
                               articles a
                          where a.artname = 'BATMOBILE'
                            and a.artid = od.artid
                            and od.ordid = o.ordid))
```

或者：

```
select custname
from customers
where city = 'GOTHAM'
  and custid in
           (select custid
            from orders
            where ordered >= somefunc
              and ordid in (select od.ordid
                            from orderdetail od,
                                 articles a
                            where a.artname = 'BATMOBILE'
                              and a.artid = od.artid)
```

尽管嵌套变得更深、也更难懂了，但子查询内应选择 exists 还是 in 的选择规则相同：此选择取决于日期与商品条件的有效性。除非过去六个月的生意非常清淡，否则商品名称应为最有效的过滤条件，因此子查询中用 in 比 exists 好，这是因为，先找出所有蝙蝠车的订单、再检查销售是否发生在最近六个月，比反过来操作要快。如果表 orderdetail 的 artid 字段有索引，这个方法会更快，否则，这个聪明巧妙的举措就会黯然失色。

注意

每当对大量记录做存在性检查时，选择 in 还是 exists 须斟酌。

利于多数 SQL 方言，非关联子查询可以被改写成 from 子句中的内嵌视图。然而，一定要记住的是，in 会隐式地剔除重复项目，当子查询改写为 from 子句中的内嵌视图时，必须要显式地消除重复项目。例如：

```
select custname
from customers
where city = 'GOTHAM'
  and custid in
           (select o.custid
            from orders o,
                 (select distinct od.ordid
                  from orderdetail od,
                       articles a
                  where a.artname = 'BATMOBILE'
                    and a.artid = od.artid) x
            where o.ordered >= somefunc
              and x.ordid = o.ordid)
```

编写功能等价的查询时，不同的编写方式就好像同义词。在书面语和口语中，同义词的意思虽然大致相同，但又有细微差异，因此某个词在特定语境中更合适。同样，数据和处理的具体实现细节可以决定选择哪种查询方式。

蝙蝠车买主案例总结

前面讨论的各段 SQL 语句，看似意义不大的编程技巧练习，实则不然。关键是“擒获（attack）”数据的方法有很多，不必按照先 customers、然后 orders、接着 orderdetail 和 articles 的方式来编写查询。

现在以箭头表示搜索条件的强度——条件分辨力越强，箭头就越大。假设 Gotham 市的客户非常少，但过去六个月的销售业绩不错，卖出了很多蝙蝠车，此时规划图如图 4-6 所示。虽然商品名称之上有个过滤条件，但图中的中等大小的箭头指向了表 orderdetail，因为该表是真正重要的表。待售商品可能很少，反映出销售收入的百分比；也可能待售商品很多，最畅销的商品之一就是蝙蝠车。

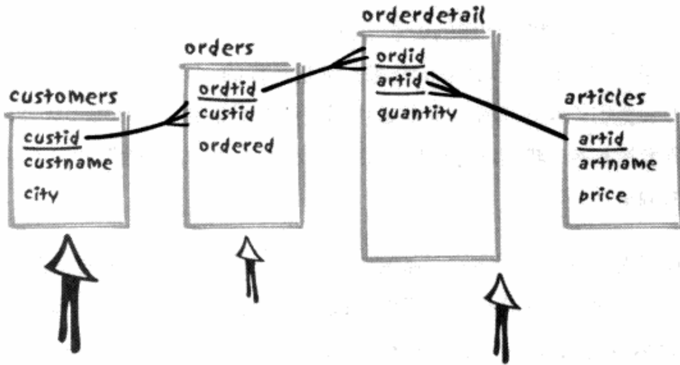


图 4-6：查询的分辨力主要受位置的影响

相反，如果我们假设多数客户在 Gotham 市，但其中很少的客户买了蝙蝠车，则规划图如图 4-7 所示。很显然，此时表 orderdetail 是最大的目标。来自这个表的数据的数据量缩减速度越快，查询执行得就越快。

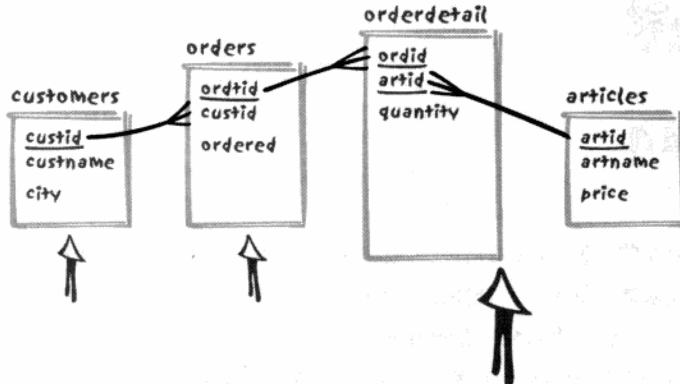


图 4-7：查询的分辨力主要受 orderdetail 表的影响

还要注意的非常重要的一点是，“过去六个月”并不是个非常精确的条件。但如果我们把条件改为过去两个月，而库中有十年的销售记录，会发生什么呢？在这种情况下，如果能先访问到近期的订单（借助第 5 章中描述的一些技术，这些数据或许就聚集在一起），查询的效率就会更高些；找出近期订单后，一方面选取 Gotham 的客户，另一方面则选取蝙蝠车订单。所以，换个角度来看，最好的执行计划并不只相依于数据值，还应该随着时间而不断进化。

好了，总结一下。首先，解决问题的方法不只一种……而且查询的编写方式经常会与数据隐含的假设相关。殊途同归，最终的结果集都是一样的，但执行速度可能有极大差异。查询的编写方式会影响执行路径，尤其是应用无法在真正的关系环境中表达的条件时。若想让优化器发挥极致，我们就必须扩大关系处理的工作量，并确保非关系的部分对最后结果集的影响最小。

本章前面一直假设代码的执行方式与编写方式一样，但其实，优化器可能改写查询——有时改动还很大。你或许认为优化器所做的改写无关紧要，因为 SQL 本是一种声明性语言（declarative language），用它来说明想要什么，并让 DBMS 予以执行。然而，你也看到了，每次用不同方式改写查询时，都必须更新关于数据分布和已有索引的假设。因此有一点非常重要：应预先考虑优化器的工作，以确定它能找到所需数据——这可能是索引，也可能是数据相关的详细统计信息。

总结：保证 SQL 语句返回正确结果，只是建立最佳 SQL 语句的第一步。

大数据量查询

Querying Large Quantities of Data

越快剔除不需要的数据，查询的后续阶段必须处理的数据量就越少，自然查询的效率就越高，这听起来显而易见。集合操作符（set operator）是这一原理的绝佳应用，其中的 union 使用最为广泛，我们经常看到通过 union 操作将几个表“粘”在一起。中等复杂程度的 union 语句较为常见，大多数被连接的表都会同时出现在 union 两端的 select 语句中。例如下面这段代码：

```
select ...
from A,
     B,
     C,
     D,
     E1
  where (condition on E1)
    and (joins and other conditions)
```

```
union
select ...
from A,
     B,
     C,
     D,
     E2
  where (condition on E2)
    and (joins and other conditions)
```

这类查询是典型的“照搬式”编程。为了提高效率，可以仅对代码中非共用的表（本例中即 E1 和 E2）使用 union，然后配合筛选条件，把 union 语句降级为内嵌视图。代码如下：

```
select ...
from A,
     B,
     C,
     D,
     (select ...
      from E1
      where (condition on E1)
      union
      select ...
      from E2
      where (condition on E2)) E
where (joins and other conditions)
```

另一个“查询条件用错了地方”的经典例子，和在含有 group by 子句的查询中进行过滤操作有关。你可以过滤分了组的字段，也可以过滤聚合（aggregate）结果（例如检查 count() 的结果是否小于某阈值），或者同时过滤两者；SQL 允许在 having 子句中使用这类条件，但应该在 group by 完成后才进行过滤（比如排序之后再进行聚合操作）。任何影响聚合函数（aggregate function）结果的条件都应放在 having 子句中，因为在 group by 之前无从知道聚合函数的结果。任何与聚合无关的条件都应放在 where 子句中，从而减少为进行 group by 而必须执行的排序操作所处理的数据量。

现在回过头来看客户与订单的例子，我承认先前处理订单的方法比较复杂。在订单完成之前，必须经历几个阶段，这些都记录在表 orderstatus 中，该表的主要字段有：ordid（订单 ID）、status、statusdate（时间戳）等，主键由 ordid 和 statusdate 组成。我们的需求是列出所有尚未标记为完成状态的订单（假设所有交易都已终止）的下列字

段：订单号、客户名、订单的最后状态，以及设置状态的时间。最终，我们写出下列查询，滤掉已完成的订单，并找出订单当前状态：

```
select c.custname, o.ordid, os.status, os.statusdate
from customers c,
     orders o,
     orderstatus os
where o.ordid = os.ordid
  and not exists (select null
                  from orderstatus os2
                  where os2.status = 'COMPLETE'
                  and os2.ordid = o.ordid)
  and os.statusdate = (select max(statusdate)
                       from orderstatus os3
                       where os3.ordid = o.ordid)
  and o.custid = c.custid
```

乍一看，这个查询很合理，但事实上，它让人非常担心。首先，上面代码中有两个子查询，但它们嵌入的方式和前一个例子的方式不同，它们只是彼此间接相关的。最让人担心的是，这两个子查询访问相同的表，而且该表在外层已经被访问过。我们编写的过滤条件质量如何呢？因为只检查了订单是否完成，所以它不是非常精确。

这个查询如何执行的呢？很显然，可以扫描 orders 表，检查每一条订单记录是否为已完成状态——注意，仅通过表 orders 即可找出所要信息似乎令人高兴，但实际情况并非如此，因为只有上述活动之后，才能检查最新状态的日期，即必须按照子查询编写的顺序来执行。

上述两个子查询是关联子查询，这很不好。因为必须要扫描 orders 表，这意味着我们必须检查 orders 的每条订单记录状态是否为 “COMPLETE”，虽然检查状态的子查询执行很快，但多次重复执行就不那么快了。而且，若第一个子查询没找到 “COMPLETE” 状态时，还必须执行第二个子查询。那么，何不试试非关联子查询呢？

要编写非关联子查询，最简单的办法是在第二个子查询上做文章。事实上，在某些 SQL 方言中，我们可以这么写：

```
and (o.ordid, os.statusdate) = (select ordid, max(statusdate)
                                from orderstatus
                                group by ordid)
```

这个子查询会对 orderestatus 作“全扫描”，但未必是坏事，下面会对此加以解释。

重写的子查询条件中，等号左端的“字段对”有点别扭，因为这两个字段来自不同的表，其实不必这样。我们想让 orders 和 orderstatus 的订单 ID 相等，但优化器能感知这一点吗？答案是不一定。所以优化器可能依然先执行子查询，依然要把 orders 和 orderstatus 这两个表连接起来。我们应该将查询稍加修改，使优化器更容易明白我们的描述，最终按照“先获得子查询的结果，然后再连接 orders 和 orderstatus 表”的顺序工作：

```
and (os.ordid, os.statusdate) = (select ordid, max(statusdate)
                                 from orderstatus
                                 group by ordid)
```

这次，等号左端的字段来自相同的表，从而不必连接 orders 和 orderstatus 这两个表了。尽管好的优化器可能会帮我们做到这一点，但保险起见，一开始就指定这两个字段来自相同的表是更明智的选择。为优化器保留最大的自由度总是上策。

前面已经看到了，非关联子查询可以变成内嵌视图，且改动不大。下面，我们写出“列出待办订单”的整个查询语句：

```
select c.custname, o.ordid, os.status, os.statusdate
from customers c,
     orders o,
     orderstatus os,
     (select ordid, max(statusdate) laststatusdate
      from orderstatus
      group by ordid) x
where o.ordid = os.ordid
  and not exists (select null
                  from orderstatus os2
                  where os2.status = 'COMPLETE'
                  and os2.ordid = o.ordid)
  and os.statusdate = x.laststatusdate
  and os.ordid = x.ordid
  and o.custid = c.custid
```

但还有问题，如果最终状态确实是“COMPLETE”，我们就没有必要用子查询检查其最新状态了。内嵌视图能帮我们找出最后状态，无论它是不是“COMPLETE”。所以我们把查询改为“检查已知的最新状态”，这个过滤条件非常令人满意：

```
select c.custname, o.ordid, os.status, os.statusdate
from customers c,
     orders o,
     orderstatus os,
     (select ordid, max(statusdate) laststatusdate
      from orderstatus
      group by ordid) x
where o.ordid = os.ordid
  and os.statusdate = x.laststatusdate
  and os.ordid = x.ordid
  and os.status != 'COMPLETE'
  and o.custid = c.custid
```

如果进一步利用 OLAP 或 SQL 引擎提供的分析功能，还可以避免对 orderstatus 的重复参照。不过就此打住，来思考一下我们是如何修改查询的，更重要的是“执行路径（execution path）”为何。基本上，正常路径是先扫描 orders 表，接着利用 orderstatus 表上预计非常高效的索引进行访问。在最后一版的代码中，我们改用完整扫描 orderstatus 的方法，这是为了执行 group by。orderstatus 中的记录条数一定会比 orders 中的大好几倍，然而，只以要扫描的数据量来看，估计前者比较小（而且可能小很多），这取决于为每张订单保存了多少信息。

无法确定哪种方法一定更好，这一切都取决于实际数据。补充说明一点，最好别在预期会增大的表上做全表扫描操作（若能把搜索限制在最近一个月或几个月的数据上则会好些）。不过，最后一版的代码肯定比第一版的（在 where 子句用子查询）要好。

在结束“大数据量查询”的话题之前，有个特殊情况值得一提。当查询要返回非常大量的数据时，该查询很可能不是某个用户坐在电脑前敲入的命令，而是来自于某个批处理操作。即便“预备阶段”稍长，只要整个处理能达到令人满意的结果，就是可以接受的。当然，不要忘了，无论是不是预备阶段，都会需要资源——CPU、内存，可能还有临时磁盘空间。即使最基本的查询完全相同，优化器在返回大量数据时所选择的路径，仍可能会与返回少量数据时完全不同，了解这一点是有用的。

总结：尽早过滤掉不需要的数据。

取出数据在表中的比例
The Proportions of Retrieved Data

有个典型的说法：当查询返回的记录数超过表中数据总量的 10% 时，就不要使用索引。这种说法暗示，当（常规）索引的键指向表中不足 10%的记录时，它是高效的。正如第 3 章中所指出的，这个经验法则建立于许多公司仍对关系数据库有所怀疑的年代，那时，关系数据库一般用于部门级数据库，包含十万行数据的表就被认为是大型表。与含有五亿行数据的表相比，十万行的 10% 不值一提。所以，执行计划"佳者恒佳"仅是个美好的愿望罢了。

就算不考虑"10%的记录"这条"经验法则（rule of thumb）"产生的年代（现在的表大小早已今非昔比了），要知道，返回的记录数除了与期望响应时间有关之外，它本身并无意义。例如，计算十亿行数据的某字段的平均值，虽然返回结果只有一行，但 DBMS 要做大量工作。甚至没有任何聚合处理，DBMS 要访问的数据页的数量也会造成影响。因为要访问的数据页并非只依赖索引：第 3 章曾指出，表中记录的物理顺序与索引顺序是否一致，对要访问的页数有极大影响；第 5 章将讨论的一些物理实现也会造成影响，由于数据的物理存储方式不同，检索出相同数量的记录所要访问的数据页数量可能差异很大；此外，有的访问路径将以串行方式执行，有的则以大规模并行（parallelized）方式执行……。因此，再别拿"10%的记录"这根鸡毛当令箭了。

总结：当查询的结果集很大时，索引未必必要。

第 5 章

了如指掌：理解物理实现

Terrain:
Understanding Physical Implementation

地形地貌与战争的关系十分密切，是战役的决定性因素。

——Carl von Clausewitz（1780—1831）
《战略谈》，第五卷，第 17 章

从程序的角度而言，表不仅包括 table，还包括 view，只不过 table 的存储参数为优化某类操作而进行了精心设置。本章将探讨表中数据不同的存储方式，以及该存储方式有利于哪类操作。

首先强调，本章的主题不是磁盘空间分配、也不是日志和数据文件如何放置，系统工程师和数据库管理员关心这些问题，但其他人不会感兴趣。数据库的组织方式，远不止这些字节级的考虑，数据的实际特点是选择存储方式的最主要因素。

系统工程师和数据库管理员知道需要多少存储空间，也知道保存这些数据可以选择不同技术——无论是磁盘“条带化（stripe）”等低级的数据容器，还是表（table）等高级的数据容器。但是，他们对这些数据容器背后所使用的技术往往缺乏了解。有时，地形的选择对战斗很重要。就像将军会与工程兵讨论战术一样，应用系统的架构师也会与数据库管理员一起研究数据的物理布局问题。不过，有时我们必须在自己无法控制的地形中战斗，或者更糟——数据的物理布局优化完全是针对其他目的而设计的。

物理结构的类型
Structural Types

尽管数据库的物理结构和 SQL 没有直接关系，但如何使用 SQL 却受到这些底层结构的影响。现今知名的数据库，大概可归纳为两种结构类型。

固定型

> 总有些时候，我们别无选择。我们必须使用现存的数据库结构，无论它造成性能不佳的可能性有多大（只要不是真正原因）。不管是开发新的应用，还是改进已有系统，SQL 语句都将受到底层结构的制约。我们别无选择，所以必须尽量了解系统内部的工作情况，并加以利用。

有一定的灵活性，有时可更改数据的物理布局（在不改变其逻辑模型的前提下）。但一定要明白，这样做是有风险的，且数据库管理员不情愿这样做并不是因为懒惰。为了解决性能问题，尽管重新安排数据的物理布局可能造成服务中断，但许多人仍对此抱以最后的希望。可惜此招并非总是灵验，它在某些情况下可能相当有用，有时却不怎么管用，有时甚至有害。所以，重要的是，判断这种强力措施是否能为我们带来好处。

从某种意义上来说，如果设计本身有缺陷，这两种类型的物理结构都于事无补。“因设计不正确而放弃希望”或许有些夸张，但仍需强调：越早发现设计问题越好。

在很多方面，具体数据库实现选择和一级方程式赛车（F1）中选择轮胎很像：你必须预测比赛情况，选错轮胎可能代价惨重，选对了则有助于获胜，但即使是最佳选择也不能保证胜利。

本章不讨论如何编写 SQL 语句，也不深入讨论具体实现细节——这些受数据库产品本身的影响太大。然而，实践中必须了解设计所要面对的各种情况（包括好坏两方面），否则难于设计出可靠的架构。同时，必须了解特殊的物理实现会对性能有多大影响，会使性能更好还是更糟。所以，一方面，实践中的问题有时需由 DBMS 实现者解决，帮助改善查询和修改数据库的性能（在第 9 章会作更多讨论）；另一方面，实践中的有些问题已有答案。本章讨论的某些特性，并不是所有数据库系统都支持的，或者需独立授权才支持。

在开始进一步讨论之前，还有最后一点要提醒。针对本章主题，我在各种商业产品中发现了一些可比之处，并会给出一些比较数据，但我们绝不想在各种数据库产品中“选美”，何况不同版本间的比较结果可能不同。同样，绝对值并没有意义，因为那完全取决于硬件和数据库的设计，所以我们的比较结果都是相对值，而且对比是在同一 DBMS 的不同变化间进行（仅有一个例外）。

冲突的目标

The Conflicting Goals

在并发用户数很大的系统中，有人在读数据，有人在写数据，此时试图优化数据的物理布局常碰到两个冲突的目标。一个目标是，尽量以紧凑的方式存储数据，这有助于查询尽快找到所有数据。另一个目标则是，尽量将数据分散存储，以便多个进程可以并发写入，不会造成资源争用而互相妨碍。

甚至没有并发操作的情况下，设计数据的物理布局也面临尴尬：数据查询和数据更新都快，这是冲突的目标。显然，索引是个恰当的例子：人们为字段加上索引，期待着查询时建立索引的字段作为 select 的条件；然而，正如第 3 章所指出的，维护索引的开销很高，索引的插入操作通常比表的插入操作开销高得多。

资源争用问题会影响所有必须保存的数据，尤其是对“修改密集型（change-heavy）”的事务处理系统更是如此（“修改”一词泛指所有插入、删除和更新操作）。存储设备和操作系统可处理资源争用问题。数据库的数据文件可能被分区、镜像并散布各处，以确保硬件失败时的数据完整性，并有助于解决资源争用。

不幸的是，只依赖操作系统处理资源争用问题还不够。从数据库的角度来看，DBMS 所处理的数据基本单元（即数据页或数据块，视产品而定）通常是不可分割的（甚至在最低层也是），尤其是最终到内存中被扫描时。甚至在系统工程师看来一切完美时，也可能依然存在 DBMS 性能问题。

为了尽量缩短响应时间，就必须减少数据库引擎访问数据页的数量，重要手段有两种：

- 努力使每页的数据密度更高
- 将一个检索处理最可能访问的数据聚集在一起

然而，用尽量少的页来存储数据，不利于解决一个页被多个进程同时读写的问题，因为资源争用问题的解决会变得更加复杂。

许多人认为，数据库的物理布局完全是数据库管理员的职责，但其实，他虽是主要负责人，但并非负全责。数据的物理布局与数据的特征及预期用途密切相关。例如，在优化物理布局的设计时，分区（partitioning）会很有帮助，但这个方法不应随意使用。正因为处理需求和物理层的设计关系密切，所以当两个或多个业务处理过程共享数据时，常出现数据设计方案强烈冲突的情况。这就像是将军在战场上遇到了进退两难的困境，他必须根据具体地形权衡各种兵团（步兵、骑兵、炮兵）的部署。所以，表和索引的物理设计，需要数据库管理员和开发者合作，根据业务需求，结合既有 DBMS 的特性，使物理层设计尽可能最佳。

以下章节将介绍一些不同的策略，然后从单进程（通常批处理程序是单进程）的角度来说明各策略对查询和更新的影响。

总结：读与写不会和睦相处：读操作希望数据聚集在一起，而并发的写入操作希望数据是分散保存的。

把索引当成数据仓库

Considering Indexes as Data Repositories

根据键，利用索引可以快速找到记录地址（指向持久化存储设备的某存储区域，通常是文件标识及文件内偏移量）。操作系统使用该记录地址——如果幸运的话，操作系统直接将它对应到数据所在缓存的真实内存地址，否则索引搜索会引起 IO 操作，将所需数据读入内存加以使用。

就如第 3 章讨论的，当待查的键值指向大量记录时，通常效率较高的做法是从头到尾扫描表并忽略索引。所以，当一个字段包含较多重复值时（即低区别度），为该字段建索引没什么用（至少对事务处理型数据库如此），除非某个值可选择性很高且频繁出现在 `where` 子句中。作为复合索引的第一个字段，也没有必要单独加索引，无论如何都不需

要对一个字段建立两个索引。即使没有完整的键值，只要确保前面几个字节足以分辨数据，就能高效地搜寻常见的树状或层式索引。

利用头部几个字节而不是完整键值来查询索引，导致了一种有趣的优化。如果在（c1, c2, c3）上加了索引，即使只指定 c1 的值，也可以使用该索引。此外，如果键值没有压缩，索引中所含的值组(c1, c2, c3) 会与相应表中的数据一样。因为，如果想由 c1 获得相应的 c2 值，或再由 c2 找出相应的 c3，就能完全在索引内完成，而毋需进一步访问实际数据表。做个非常简单的类比，就像利用任何网络搜索引擎搜索莎士比亚（Wiliam Shakespeare）的出生年份，在提交字符串 “William Shakespeare” 后将返回类似图 5-1 的信息。

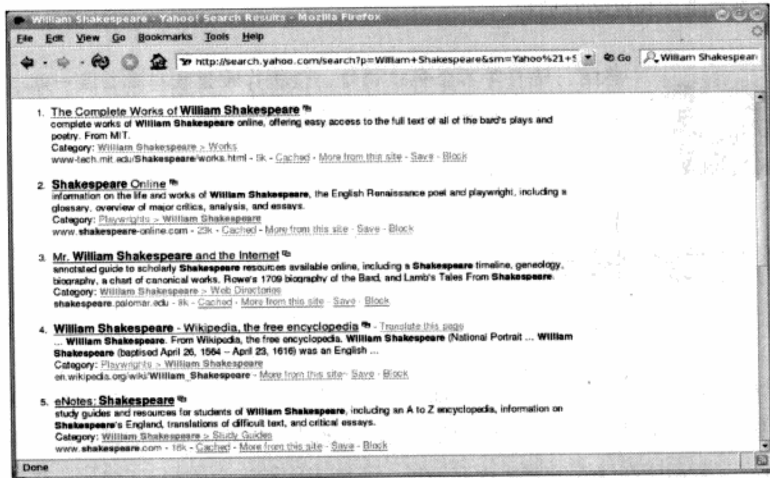


图 5-1：通过 Web 搜索 “William Shakespeare”

无须访问任何被搜索网站（这可能恰巧会引起问题），答案已经出现在了搜索引擎返回的索引中。第四条搜索结果显示，莎士比亚出生于 1564 年。

当索引已含有所需信息时，就没必要再访问它所指的原始数据，这与常用优化策略的基本思想如出一辙。在索引中加入额外的字段（一个或多个，它们本身与实际搜寻条件无关，但却含有查询所需的关键数据），就能提高某个频繁运行的查询的速度，这是因为

查询所需要的数据可以全部从索引中获得，而完全不必再访问原始数据源。有些产品（例如 DB2）支持为唯一性索引指定可包含哪些其他字段，并支持只检查复合键部分字段的唯一性。Oracle 也能做到这些，方式不太直接，它使用非唯一性索引，并强制实施一个唯一性约束或主键约束。

但这时会出现反常现象：对查询“稍作修改”后，批处理程序的执行时间较修改前大幅增长。这里“稍作修改”是指，在 `select` 的返回字段中增加一个复合索引中没有的字段。在修改之前，整个查询仅从索引返回的数据中就可能得到满足，新字段的加入，却迫使数据库访问相应的表，从而导致处理器工作量大增。

下面深入讨论“只查索引”与“查索引和表”的性能差异。图 5-2 比较了在三种主流数据库系统中，当查询中增加一个不在复合索引中的字段时，性能上会有多大影响。测试表均含 12 个字段和 250 000 行数据，主键是三个字段的复合键：整数字段，其值为平均散布在 1 到 5 000 之间的随机数；接着是一个 8 到 10 个字符的字符串型字段；最后是 一个 datetime 型字段。除了主键上有唯一性索引之外，该表没有其他的索引。用来对比的参照查询是以第一个字段中 1 到 5 000 之间的随机数为条件，获取主键中第二个和第三个字段。此测试反映多获取一个不在索引中的字段时（是个数值型字段，所以相对而言修改很小），性能会受到多大影响。图 5-2 中的结果已做标准化处理，仅获取两个索引字段的性能为 100%，获取额外字段的性能则一定小于 100%。

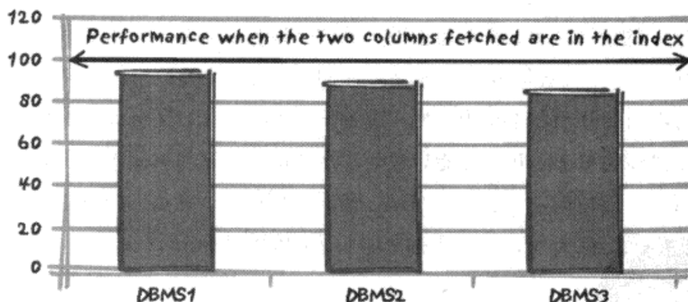


图 5-2：“第三个字段必须从 table 获得”对性能的影响

如图 5-2 所示，与直接从索引中获取数据相比，一定要访问原始表对性能的影响并非很大（大约 5% 到 10 %），但并不是没有影响，而且有些数据库产品受到的影响较大。当然，确切的数字会因具体情况而异，如果访问表需要额外的物理 I/O 操作，影响会更严重，但此例不涉及该情况。

把“尽量在索引中多存储数据”的原则扩展到极限，有些数据库管理系统（如 Oracle）允许在主键索引中存储表中所有数据，于是不再需要表。此方法能节省存储空间，也可能节省时间。表即是索引，相对于常规的堆结构（heap structure）而言，这就是所谓的“索引组织表（index-organized table，IOT）”。

第 3 章讨论了插入索引带来的额外开销，你或许会希望数据插入“索引组织表”的开销低于只有主键索引的表，但事实上，在某些情况下正好相反，如图 5-3 所示。这个测试用了四个表，来比较插入普通表与插入“索引组织表”的速率。首先建立两种表样式：第一种表是小型表，由主键字段加上一个额外字段组成；第二种表则由主键字段加上另外九个数值字段组成。这两种表的（复合）主键均被定义为一个数值字段、一个长度为 10 的字符串字段，以及一个时间戳字段。两种表均有普通堆结构的版本及“索引组织表”版本，所以一共四个表。针对四个表要进行两组测试：在第一个测试中，插入随机顺序的主键值；第二个测试，则在主键首字段依升序排序插入数据。

除了定义为主键的字段，“索引组织表”中只有少量字段，所以插入速度比较快。然而，如果该表中字段的数量很多，所有与主键无关的字段也必须被存储在索引结构中（有时存入额外区域）。由于表就是索引，所以它存储的信息要比其他情况下还多。第 3 章也提过了，插入索引的开销在本质上比插入普通表还高，那么，在更复杂的结构中插入更多数据时，数据移动带来的开销可能导致效率严重降低，除非被插入记录的顺序与主键索引相同或近似。若数据中含有长字符串，情况会更严重。在很多情况下，不访问表而从主键索引获取数据的效益，会小于插入时带来的额外开销（注 1）。

注 1：有专家认为：除了主键索引之外，其他索引“在索引组织表上”的效率比“在常规表上”要低。这是由于实现原理（超出本书讨论范围）的原因。

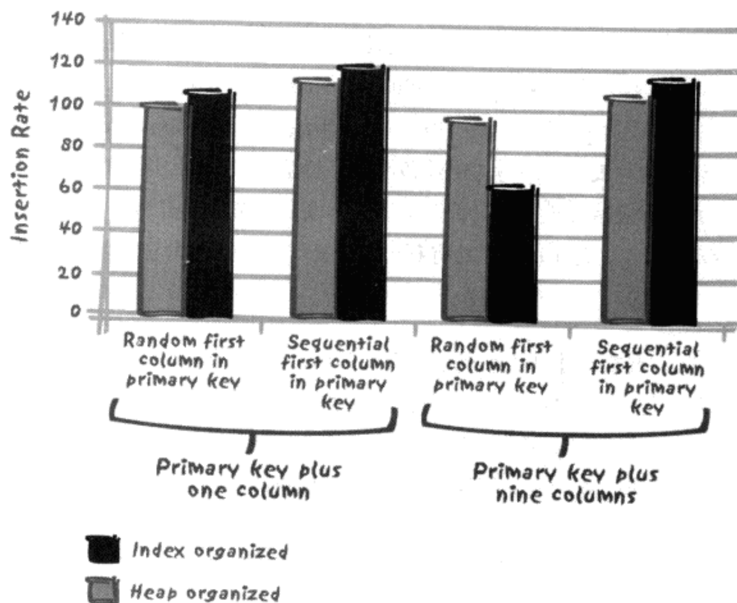


图 5-3：插入 Oracle 的索引组织表和常规表（基于堆组织）的成本比较

索引具有强大的内部排序功能，其潜在益处我们稍后讨论。

总结：有些查询，仅通过索引数据即可完成。

记录强制排序

Forcing Row Ordering

除了无须访问表即可找到全部数据之外，“索引组织表”还有一个特点：因为本身即是索引，是严格排序的结构，所以记录在其内部已排好序了。虽然关系理论中根本没有顺序的概念，但实践中，每当进行范围搜索时，有序结构有助于集中发现范围数据，而不必四处搜集。这类应用中最常见例子，就是基于时间顺序的范围搜寻，例如查找两个特定日期之间的记录。

为了使表中记录有序，多数数据库系统会使用索引来定义记录顺序。SQL Server 和 Sybase 中称这种索引为“聚集索引（clustered index）”，DB2 则称之为“聚集索引（clustering index）”，它们和 Oracle 中的“索引组织表（index-organized table）”非常类似。有些查询从这种表的组织方式中获益极大。但是，和“索引组织表”类似，更新定义顺序的索引字段时开销明显，因为数据更新会导致新的“排名”，继而导致数据的物理位置变动。所以，有序记录必然有利于范围查询，代价是非范围搜索也要进行范围扫描。

为了更加安全，“聚集索引（clustering index）”可以像“索引组织表”一样，根据主键来定义，因为主键绝不会被更新（如果应用需要修改主键，说明设计上有严重的问题，很快就会导致严重的数据完整性问题）。但与“索引组织表”不同，“聚集索引（clustering index）”也可以是非主键索引。但要记住，任何有序数据便于某些处理的同时，必将对其他处理不利。如果主键的值有业务含义（不是生成键），则它的索引比其他索引（包括唯一性索引）更有用，更可能在物理实现层面需要倍加关注。

图 5-4 显示了聚集与非聚集索引的实际差异。如果采用图 5-3 相同的索引组织表（三个字段的主键加上九个数值字段），且以完全随机的方式插入数据，在主键聚集的表中进行插入操作开销很高，测试结果显示其效率仅为非聚集主键表的一半。但在大约 50 000 行的数据中执行范围扫描时，此聚集索引极大地提高了性能，此例中它比非聚集方式约快 20 倍。当然，只查询一条记录时，它们之间没有差别。

诸如“聚集索引”和“索引组织表”这些结构优化，必然存在缺点。首先，表被改造成了严格的树状结构，从而达到分层排序（hierarchical ordering）的效果。但层次型的树状数据库在业界已被关系数据库取代，这个优化自然也就带来许多树状数据库的缺点。任一种层次结构，都仅有利于一类数据操作和一条访问路径，但多数其他访问路径常比较糟糕。更新数据的开销也更高。原本库文件内的数据物理分布非常整齐，但现在由于加链接、增加溢出页等改造，数据的物理分布不再整齐，从而也降低了性能。聚集结构

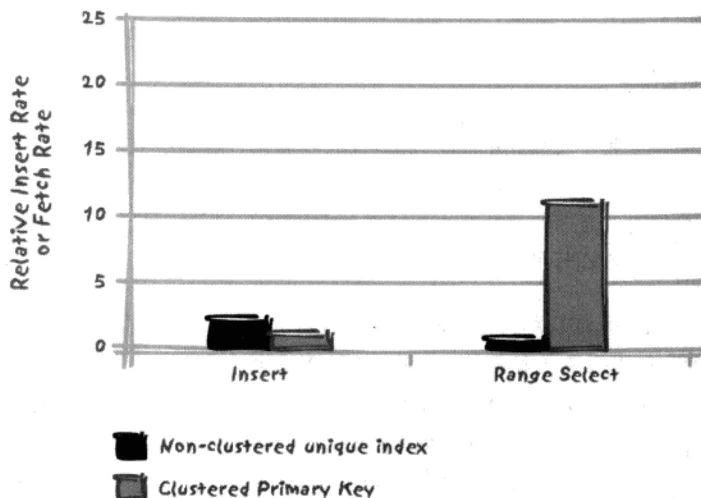


图 5-4：聚集索引性能如何

在某些情况下能成倍提高效率，但一定要经过仔细测试，因为它们很可能使其他操作效率降低。所以，“聚集索引”和“索引组织表”的适用性必须从整体角度考虑，而不是仅考虑一个具体的查询。

总结：对聚集数据进行范围查询效率惊人，但其他查询的效率会因此大打折扣。

数据自动分组（Grouping）

Automatically Grouping Data

范围扫描时若能同时发现多条记录，则可大大提高效率。事实上，处理使用“聚集索引”和“索引组织表”之外，还可以通过其他手段达到数据分组的效果。所有的数据库管理系统，都支持对表进行分区—— 这利用了古老的“分而治之”原则——将大型表切割为多个部分以便于管理。此外，从处理的角度来看，分区可以提高并发性（concurrency）和并行性（parallelism），从而使系统架构的可伸缩性（scalable）增强，这在第 9 章和第 10 章会加以说明。

首先需明确，“分区（partition）”这个词的含义会因 DBMS 不同（甚至 DBMS 版本不同）而不同。现在 Oracle 中“表空间 （tablespace）”，有段时间就曾被称为“分区”。

循环分区

Round-Robin Partitioning

循环分区完全是一种不受数据影响的内部机制。我们大可随意地把分区定义为各个磁盘的存储区域，通常会和数据存储设备的数量一样多。一个表可能会存储在一个或多个分区中。插入数据时，数据会按照某方法循环（round-robin）地加载到各个分区，以保持插入引起的磁盘 I/O 操作的平衡。

顺便说明，循环分区对并发的随机查询很有帮助，此机制与磁盘阵列上的文件条带化（file striping）技术很像。事实上，已采用文件条带化技术的情况下，循环分区的价值就不大了。循环分区可看做是随意散布数据的机制，它并不根据实际的逻辑数据相关性来分组数据。然而，在某些产品中（如 Sybase），一个事务（transaction）总是在同一分区进行写操作，这在一定程度上实现了符合业务处理的数据分组。

数据驱动分区

Data-Driven Partitioning

然而，有种更有趣的分区类型，就是所谓的数据驱动分区（data-driven partitioning）。它是根据一个或数个字段中的值来定义分区，每条记录插入相应的分区。当然，此时 DBMS 对数据及其存储方式知道得越多越好。

多数非常大的表之所以大，是因为它们包含了历史数据。然而，随着新数据的不断加入，对过去的兴趣会逐渐消退，所以我们有理由假设，最常被查询的是那些较新的数据。因此，很自然地，我们可以根据日期进行分区，去芜存菁，把常用数据和不常用数据分别放在不同的分区中。

例如，可以以手工方式，按日期将数据量很大的 `figures` 表（包含最近十二个月的数据）分区为 12 个独立表，按月份名称分别命名为 `jan_figures`、`feb_figures`，……，`dec_figures`。如果要查询全年数据，只要对十二个表做 `union` 操作即可。这样的 `union` 操作受到数据库厂商的"官方认可（official endorsement）"，一般叫"分区视图（partitioned view）"，MySQL 则称之为"合并表（merge table）"。在三月份，数据会插入 `mar_figures` 表，到了四月份会切换到 `apr_figures`。视图可作为一组结构类似的表的综合对象，这好像是个不错的方法，但也有一些缺点：

- 最大的缺点就是，这种视图的基本设计思想就有问题。设计者知道视图背后那些表在逻辑上的关联性，但 DBMS 却无从知道这些（除了分区视图的一点点定义）。这种将一个大表分成若干个小表的设计不利于正确定义完整性约束，很难让那些表都保持唯一性，而且还必须建立多个外键来参照这“组”表，这既复杂又不合理。就完整性（integrity）而言，唯一能做的是在决定分区的字段上增加一个 check 约束，例如，可以对 sales_date 字段增加 check 约束，以确保 jun_sales 表中的 sales_date 不会落在 6 月 1 日到 6 月 30 日的范围之外。

- 如果 DBMS 不支持分区视图，就更不方便，因为每个月都必须插入不同的表，所以就要对这一组表编程，动态建立 insert 语句以适应变化的表名，这使程序更复杂。例如，在前例中，程序一定要先获得日期（当前日期或某个设定日期），然后检查并确定表名，最后产生相应的 SQL 语句。当然，如果 DBMS 支持分区视图，情况会好得多，因为插入会直接通过视图执行，而 DBMS 会确定记录插入的位置。

  然而无论如何，上述设计问题带来的直接后果是，在执行了一些不连贯的插入操作之后，必须编程实现“参照完整性（referential integrity）”的检查，这使开发工作量增加（计算机的负载也增加），也使本来就有问题的设计变得更复杂。其实，这个参照完整性的检查工作本可以由 DBMS 核心来完成，而现在要靠触发器和存储过程实现（已算不错了），甚至靠应用程序来实现（再糟不过了）。

- 使用综合视图对查询性能有一定影响。如果对 figures 表中某月份的数据有兴趣，可以查询该月的表；如果对最近 30 天的数据有兴趣，最多只须查两个表；但最简单且较易维护的编写查询的方式是使用查询视图，而不是视图背后的表。如果有分区视图，而且查询条件中含有辨别记录所在分区的字段，DBMS 优化器倒是能把查询范围缩小到需要的几个表中；但如果没有分区视图，查询就一定比基于普通表的查询复杂，尤其是查询中包含子查询（subquery）或聚合（aggregate）处理时。随着 union 涉及的表增多，查询的复杂度也会持续增加。查询大型 union 视图的开销比查询单一表要大，重复执行类似语句很快就会影响性能。

回顾过去，多数数据库管理系统在实现分区功能时所采取的第一步，就是支持“分区视图（partitioned views）”。接下来必然的一步，就是支持真正的“数据驱动分区（data-driven partitioning）”。真正的分区要求逻辑上有一个单独的表，且它具有能被其他表参照的真正主键。此外，指定一个或多个字段作为“分区键（partition key）”，该键决定记录将被插入哪个分区。这样，我们就能利用分区视图的所有优势，即在操作表时具透明性，而且可以把保证数据完整性的任务交回给 DBMS 引擎，那是 DBMS 的主要功能之一。由于 DBMS 核心知道分区情况，从而知道如何利用这种物理结构进行优化，把操作范围缩小到几个分区上（即所谓的分区截断（partition pruning）），或并行操作几个分区。

分区的具体实现可采用多少方式，则因产品而异，每种方法分别适合不同情况。

哈希分区（Hash-partitioning）

对分区键（partition key）进行运算，根据结果进行分区。这个随意的安排完全基于算术运算的结果，而且完全不考虑数据值的分布。尽管如此，哈希分区能保证根据分区键快速找到记录。哈希分区对范围搜索没有优势，因为哈希函数会把连续的键值转换为非连续的哈希值，再根据哈希值确定物理地址。

注意

DB2 增加了一种称为范围聚集（range-clustering）的机制，虽然它与分区不同，但仍然是由键值来决定数据的物理位置。与哈希方式不同，它是通过保留数据项顺序的机制来实现的，这样，无论是访问个别数据，还是范围扫描，效率都得到了保证。

范围分区（Range-partitioning）

根据连续数据的范围对数据进行分组。范围分区非常适合处理历史性数据。范围分区与先前讨论的分区视图概念最为接近：分区专门用来存储特定范围内的数据。系统会设定一个 “else 分区”来存储所有可能“漏网”的数据。虽然范围分区最常用于按时间范围进行的分区，例如小时、年等，但这类分区并不局限于特定数

据类型。例如，在一本多卷的百科全书中，每一卷里的文章，都必然属于那一册按字母序所限定的范围，除此之外就没有其他特别的顺序了。

列表分区（List-partitioning）

这是“最具手工风格（most manual type of）”的分区类型，适合定制解决方案（tailor-made solutions）。正如该方法的名字一样：为了说明特定分区可包含哪些记录，你必须明确指定分区键（通常为一个字段）一组可能的取值。当值的分布不均匀时，链表分区比较有用。

分区处理有时会因为建立子分区（subpartition）而重复进行。子分区是分区内的分区，它无非就是一种二维分区处理，例如，在一个范围分区内建立哈希子分区。

总结：以数据值本身作为分区的基础，此时数据分区最有价值。

分区是双刃剑

The Double-Edged Sword of Partitioning

数据驱动分区能将一个表中的数据分散到多个稍独立的分区中，但这并不意味着并发问题就全解决了。例如，一个表按日期进行分区，每星期的数据一个分区。若把一年的数据写入五十二个逻辑上不同的分区，这种分区方式的确很有效。但问题是，每个星期所有人都会拥入同一分区插入新数据——更糟的是，如果分区键是当前系统日期，所有这些并发操作都会针对同一个数据块（除非利用一些结构上的实现技巧，例如维持几个可供插入的页或数据块列表），结果将导致非常棘手的内存争用。这个大型表中的大部分都无人访问，但存储最新数据分区的访问却过于集中，当许多进程同时插入数据时，这样的分区方式显然不够理想。

注意

如果所有数据都通过单一进程进行插入（数据仓库环境中会出现这种情况），“访问过于集中”问题就不存在了，而五十二个星期的分区方案也不会导致并发性问题。

另一方面，假设根据订单的地理位置进行分区（如果产品在某地热销而在另一些地方销路不畅，则应谨慎采用此分区方式）。指定某个时间，由于销售不太可能全都来自同一

区域，所以插入的数据大体会随机分布到各分区。但要基于时间生成报表时，这种分区对性能影响较为明显，因为分区是按地理位置划分的，效率肯定比按时间划分的分区要低。然而，基于地理位置的分区也有可能对基于时间的查询有利，因为在多处理器的机器上，对各分区的搜寻可以并行执行，随后将结果合并。

因此，分区是把双刃剑。一方面，它通过分区键将数据聚集在一起，利于高速检索。另一方面，对并发执行的插入操作，分散数据可避免出现“访问过于集中”的问题。但在实践中，这两个目标可能彼此矛盾，所以首要问题是搞清楚要解决的主要问题是什么，并针对主要问题设计分区。而且，两方面的得失都要检查，看结果是否能够接受。理想的情况是为 `select` 而设计的数据聚集方式，与为 `insert` 而设计的数据散布方式是一致的，只可惜这并常见。

总结：到底用数据分区来分散数据、还是聚集数据，完全取决于你的需求。

分区与数据分布

Partitioning and Data Distribution

或许你认为，只要表非常大，且希望避免数据的并发写入操作发生资源争用，就一定会采用某种方法对数据进行分区。但事实并非如此。

假设有个很大的客户订单明细表，如果该表中大部分数据都来自于同一个客户（该情况时有发生），那么按客户 ID 对数据进行分区，就不会有太大帮助。我们可以非常粗略地把查询分为两大类：与大客户相关的查询及与较小客户相关的查询。当查询小客户的数据时，在客户 ID 上的索引可选择性会很高，因而查询效率也很高，这时完全不需要分区。聪明的优化器在获得关于键值分布的适当统计数据后，即可侦测到分布的不均匀并转而使用索引。这种把小客户存储在较小的分区，紧靠在含有主要客户的大分区旁的做法，好处并不大。

相反，当查询大客户的相关数据时，优化器也知道扫描表是效率最高的处理方式。在此情况下，由于该大客户的数据占全表很大比例——假设为 80%，仅扫描该客户数据所在分区并不比全表扫描快很多，所以性能提升不大，但购买 DBMS 的分区功能（该功能单独定价）却需要额外开销。

总结：对分区表进行查询，当数据按分区键均匀分布时，收益最大。

数据分区的最佳方法
The Best Way to Partition Data

绝对不要忘记，整体改善业务处理的操作，才是选择非标准的存储选项（例如分区）的目标。这意味着，改善不合理的业务流程是重中之重。例如，牺牲晚上的时间进行批处理工作，优化白天的事务型处理，是合理的；也可以优化批处理工作，如果能将关键的上载时间减至最短（期间数据对用户仍不可用），我们可以接受交易处理速度稍慢一点。这是平衡问题。

一般而言，当有多个处理在类似条件下执行时，不应过度偏袒其中一个。就这点上，只要是根据数据值决定物理位置的存储方式（例如聚集型索引和分区），更新数据时代价都非常高。对普通表的更新，几乎是在物理地址不变的情况下做更改，最多只修改和移动表中的一些字节而已；但如果采用了上述那些存储方式，更新必将导致一系列删除、插入操作和相关索引维护工作。

更新分区键会引起移动数据，似乎应避免这么做，但奇怪的是，根据可更新的键进行分区，有时效果更好。例如，有个表，用作服务队列，某进程会在该表中插入不同类型（例如 T1 型到 Tn 型）的服务请求。新出现的服务请求，初始状态会被设为 W——表示“等待处理（waiting to be processed）”。服务器进程 S1 到 Sp 会定期检查表中是否有 W 状态的请求，若有则将其状态更改为 P——表示“处理中（being processed）”，接着，在完成每个请求后，其状态被置为 D——表示“已完成（done）”。

让我们进一步假设服务器进程的数量和请求的种类一样多，而且每个服务器进程专门用来处理特定类型的请求。图 5-5 显示了服务队列和进程。当然，表中不能塞满了“已完成”状态的请求，所以必须有回收进程（图中未显示）每隔一段时间移走已完成请求。

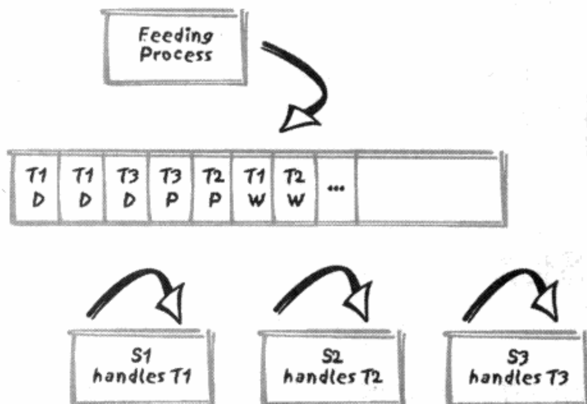


图 5-5：服务队列

每个服务器进程会定期执行一个 select 语句（事实上是 select ... for update），这个 select 语句的一个条件是“请求类型”（取决于服务器），另一个是“请求状态”：

```
and status = 'W'
```

下面讨论对服务队列表进行分区的几种方法。按请求类型进行分区可能是最显而易见的方法，如果服务器进程可能崩溃或变慢，该分区方法就很有优势：可以在不影响其他种类进程的情况下，增加出问题进程的进程数。

根据请求类型分区的另一优点是，可避免同一类型的大量请求“淹没”系统：在没有分区的情况下，每类请求的轮询进程（polling processes）扫描队列一般会发现少量需要处理的记录，若此时突然出现大量同一类型和状态的待处理请求，每个轮询进程都要检查所有请求，造成各进程的等待时间加长；如果队列按类型进行了分区，则可避免不同类型的处理相互影响。

当然，还可以根据状态（status）对请求队列表进行分区。其缺点很明显：任何状态的改变，都会使请求从一个分区移动到另一个分区。这样的迁移有好处吗？事实上，确实

有。在分区 W 中的所有请求都已准备好并等待接受处理，而对于另一服务器进程负责的请求队列、或已处理请求的队列，就无须再扫描了，因此，轮询（polling）的开销可能会大幅降低。另一个优点是，回收进程将在独立的分区上进行操作，不会干扰服务器进程。

我们不能断言“必须按类型分区”或“必须按状态分区”，这取决于服务器进程的数量、轮询的频率、数据的相对流量、各类型请求的处理时间，以及已完成请求的移除频率，等等。我们必须小心测试各种假定，并从整体的角度加以考虑。但是，有时彻底牺牲某种操作的性能，可提高其他频繁运行的处理的效率，从而整个系统性能也得以提升。

总结：对表进行分区的方法有多种，显而易见的分区方法未必是最有效的，一定要从整体来考虑。

预连接表
Pre-Joining Tables

前面已经看到，物理上分组在一起的记录对范围扫描有很大好处，因为此时我们感兴趣的是逻辑上相邻的数据。但到目前为止，我们只讨论了如何从一个表获取数据。除非数据库设计非常、非常、非常糟，否则大部分查询都会涉及多个表，因此，我们仅关注“如何把来自同一个表的数据分组到某一物理区域”似乎是不够的，我们需要关注“如何把至少两个表的数据分组到相同的物理区域”。

答案就是预连接表（pre-joining table），某些数据库系统支持这种技术。预连接表与汇总表（summary table）或物化视图（materialized view）不同，后两者只是冗余数据，一定程度上是自动更新的预处理结果。

预连接表是物理上存储在一起的表，依据的准则通常是连接条件。Oracle 把这样一组预连接表称为“簇（cluster）”，这与本章前面的“索引聚集（index clustering）”无关，也与 MySQL 的“数据库群集（clusters of databases）”无关——那是指为访问相同的表而设的多个数据库服务器。

一般的表，其基本存储单元（页或表）通常只存放来自一个表的数据；当对表进行预连接存储时，基本单元（页或块）会存放来自两个或多个表的数据，它们以共同的连接键（common join key）为基础而放在一起。这样的安排，对某个具体的连接非常有益，对其他操作却是灾难。以下为预连接表的缺点。

- 一旦来自两个或多个表的数据开始共享一个页（或块），页能容纳的来自同一个表的数据量就会减少，因为多个表的数据要共用这一固定大小的空间。所以，为了存放所有来自该表的数据，所需页的数量会增加，执行全表扫描所需的 I/O 操作也会增加。
- 不仅分享数据需要更多的页，页的有效容量也比数据库建立时所判定的最佳容量少了，还要面对溢出（overflow）和链接（chaining）等问题。当这种情况发生时，要获得数据所需的访问次数也会增加。
- 此外，曾经分租过房子的人都会知道，某个人扩大居住空间时，其他人的空间就会变小。数据库表也是如此！但如果为簇中每个表的每个页分配完全相同的空间，结果往往是空间浪费、导致耗费更多页。

这种特殊的存储方式应该谨慎使用，仅应由数据库管理员用它来解决十分特殊的问题。开发者应忘掉这种技术。

总结：预连接表（Pre-joining table）是改进查询的一种专门技术，但几乎会影响所有其他数据库操作。

神圣的简单性

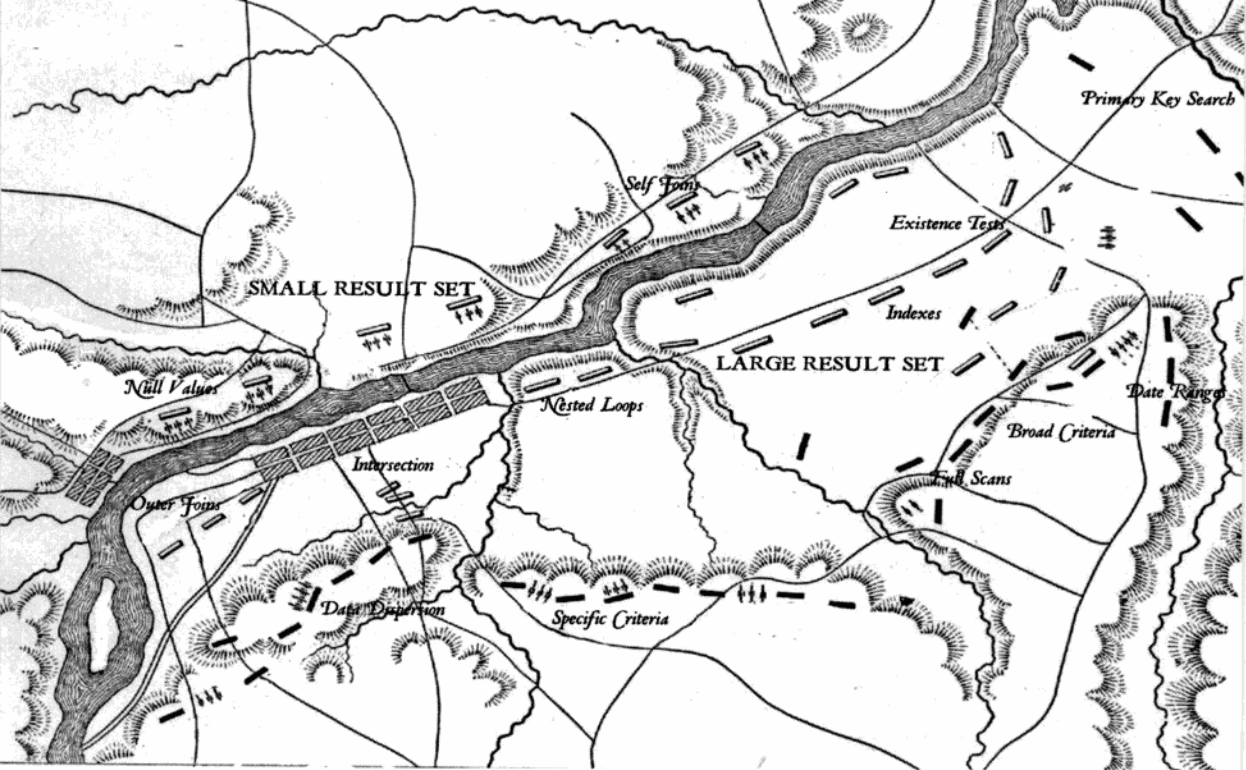
Holy Simplicity

我们有理由假定，除了默认存储方式之外的任何存储方式，无论看起来有多好用，都会带来一定程度的复杂性，且该复杂性远远超出了采用这种存储方式带来的好处（也可能没有任何好处）。最糟时，选错存储方式会大幅降低性能。军事史上出现过很多难以攻陷的堡垒，但都建在了完全错误的地方，因而无法满足任何实际需要。万里长城就从未

能阻止任何侵略，因为“没有风度”的敌人根本不从长城正面进攻。所有组织都会经历分分合合的变化，商业计划和流程也会改变，因此周详的数据库规划随之可能推倒重建。

伴随组织数据的特殊方式，是特殊的处理过程带来的麻烦。关系模型的优点之一就是“灵活性（flexibility）”，而在物理层对数据组织进行“强力干预”，很可能会在不知不觉中失去灵活性。当然，有些数据组织方式的限制少些，且数据分区几乎是数据量增大时的必然手段。但在小心测试的同时一定要记住：因为最初设计不佳而调整大型数据库的物理结构，往往要花费数天、甚至数周的时间才能完成。

总结：同样的物理存储组织方式，现在可能是有益的，但未来可能变得有害。



第 6 章

锦囊妙计：认识经典 SQL 模式

The Nine Situations: Recognizing Classic SQL Patterns

为了保证军事行动的决策清晰，只要报告对决策有影响的事实就够了。

——Général Baron de Marbot（1782—1854）

SQL 语句为了返回结果集或更改数据，必须访问一定数量的数据。“战斗”的环境和条件，决定了我们“进攻”那些数据的方法。就如第 4 章所讨论的，“进攻”取决于：结果集的数据量、必须访问的数据量、可动用的“部队”（过滤条件）。

任何大型的、复杂的查询，都可以被分成一连串较简单的步骤，其中一些步骤可以并行执行，就像综合战役通常要面对敌军的不同部队。每次战斗的结果差异可能很大，但关键是最后的综合结果。

当我们分析查询的每个步骤时可能不会深入执行细节，但这些步骤可能的组合数量跟国际象棋不相上下，可以非常复杂。

本章讨论存取经过适当规范化的数据时，经常遇到的情况。虽然本章主要讨论查询，但也适用于更新和删除操作，只要它们也有 where 子句，毕竟要先读取数据才能修改数据。无论是单纯为了查询、还是更新或删除记录，过滤数据会遇到的最典型情况有九种：

- 小结果集，源表较少，查询条件直接针对源表
- 小结果集，查询条件涉及源表之外的表
- 小结果集，多个宽泛条件，结果取交集
- 小结果集，一个源表，查询条件宽泛且涉及多个源表之外的表
- 大结果集
- 结果集来自基于一个表的自连接
- 结果集以聚合函数为基础获得
- 结果集通过简单搜索或基于日期的范围搜索获得
- 结果集和别的数据存在与否有关

本章将依次讨论上述各种情况。至于例子，有的简单明了，有的较为复杂（来自实际案例）。虽然案例大小存在差异，但解决问题的模式是相通的。

通常，在执行查询时，应过滤掉所有不属于结果集的数据，这意味着应尽量采用最高效的搜索条件。决定先执行哪个条件，通常是优化器的工作。但是，正如第 4 章所述，优化器必须考虑大量不同情况——例如表的物理结构、查询编写方式等，所以优化器未必总能“理解正确”。因此，提高性能还有很多事情可做，下面对九种模式的讨论中，每种模式均是如此。

小结果集，直接条件
Small Result Set, Direct Specific Criteria

对于典型的在线交易处理，多为返回小结果集的查询，源表数量较少，查询条件也是“直接”针对源表的。当我们要通过一组条件查询出少许记录时，首先要注意的就是索引。

一般而言，通过一个表或通过两个表的连接查询较少记录，只要确保查询有适当的索引支持即可。然而，当很多表连接在一起，并且查询条件要参照不同的表时（例如 TA 和 TB），会面临连接顺序的问题。连接顺序的选择，取决于如何更快地过滤不想要的记录。如果统计数据足够精确地反映了表的内容，优化器有可能对连接顺序做出适当选择。

当查询仅返回少量记录，且过滤条件直接针对源表时，我们必须保证这些过滤条件高效；对于非常重要的条件，必须事先为相应字段加上索引，以便查询时使用。

索引可用性
Index Usability

如第 3 章所述，对某字段使用函数时，则该字段上的索引并不能起作用。当然，你可以建立函数索引（functional index），这意味着要对函数的结果加索引，而不是为字段加索引。

注意，“函数调用”不光是指“显式函数调用”。如果你将某类型的字段与一个不同类型的字段或常量进行比较，则 DBMS 会执行“隐式类型转换”（隐式调用一个转换函数），如你所料，这会对性能造成影响。

一旦确定重要的搜索条件上有索引，而查询编写方式也的确能因索引而提高性能，我们还须进一步区别如下两种情况：

- 使用唯一性索引（unique index）检索单条记录
- 非唯一性索引（non-unique index）或基于唯一性索引的范围扫描（range scan）

查询的效率与索引的使用

Query Efficiency and Index Usage

需要连接（join）表时，唯一性索引非常有用。然而，当程序获得的原始输入（primitive input）不是查询语句需要的主键值时，必须通过编程来解决转换问题。

这里的“原始输入”指程序接受的数据，可能由使用者输入，也可能从文件中读入。如果查询语句需要的主键值本身，就是根据原始输入利用另一个查询所获得的结果，则说明设计不合理。因为这意味着一个查询的输出被用作另一个查询的输入，应该考虑合并这两个查询。

总结：优秀的查询未必来自优秀的程序。

数据散布

Data Dispersion

当条件是“非唯一性”的，或者条件以唯一性索引上的范围来表达时，DBMS 就必须执行范围扫描。例如：

```
where customer_id between ... and ...
```

或：

```
where supplier_name like 'SOMENAME%'
```

键对应的记录很可能散布在整个表中，而基于成本的优化器知道这一点。所以，索引范围扫描会使 DBMS 核心逐一读取表的存储页，此时，优化器会决定 DBMS 核心忽略索引对表进行扫描。

如第 5 章所述，许多数据库系统提供了诸如分区（partition）和聚集索引（clustered index）等功能，直接将可能一并读取的数据存储在一起。其实，数据插入处理也常造成数据丛聚（clumping）保存的现象：如果每条记录插入表时都要加时间戳（timestamp），则相继插入的记录会彼此紧邻（除非我们采取特殊手段避免资源竞争，见第 9 章的讨论）。这其实没有必要，而且关系理论中也没有“顺序”的概念，但在实际中却很可能发生。

因此，当我们在时间戳字段的索引上执行范围扫描、查询时间上接近的索引项时，这些记录可能彼此紧邻——如果特意为此设置了存储选项参数，就更是如此了。

现在做一个假定：键值与特定插入环境无关、与存储设置无关，与键值（或键值范围）对应的记录可能存储在磁盘的任何位置。索引仅以特定顺序来存储键值，而对应的记录随机散落在表中。此时，若既不分区、也不采用聚集索引，则需访问的存储区会更多。于是，可能出现下列情况：同一个表上有两个可选择性完全相同的索引，但一个索引性能好、一个索引性能差。这种情况在第 3 章已提到过，下面来分析一下。

为了说明上述情况，先创建一个具有 1 000 000 条记录的表，这个表有 c1、c2 和 c3 三个字段，c1 保存序号（1 到 1 000 000），c2 保存从 1 到 2 000 000 不等的随机数，c3 保存可重复、且经常重复的随机值。表面看来，c1 和 c2 都具唯一性，因此具有完全相同的可选择性。索引建在 c1 上，则表中字段的顺序，与索引中的顺序相符——当然，实际上，对表的删除操作会留下“空洞”，随后又有新的插入记录填入，所以记录顺序会被打乱。相比之下，索引建在 c2 上，则表中记录顺序与索引中的顺序无关。

下面读取 c3 ，使用如下范围条件：

```
where column_name between some_value and some_value + 10
```

如图 6-1 所示，使用 c1 索引（有序索引，索引中键的顺序与表中记录顺序相同）和 c2 索引（随机索引）的性能差异很大。别忘了造成这种差异的原因：为了读取 c3 的值，除了访问索引，还要访问表。如果我们有两个复合索引，分别在 (c1, c3) 和 (c2, c3) 上，就不会有上述差异了，因为这时不必访问表，从索引中即可获得要返回的内容。

图 6-1 说明的这种性能差异，也解释了下述情况的原因：有时性能会随时间而降低，尤其是在新系统刚投入生产环境并导入旧系统的大量数据时。最初加载的数据的物理排序，可能是有利于特定查询的；但随后几个月的各种活动破坏了这种顺序，于是性能“神秘”降低 30%～40%。

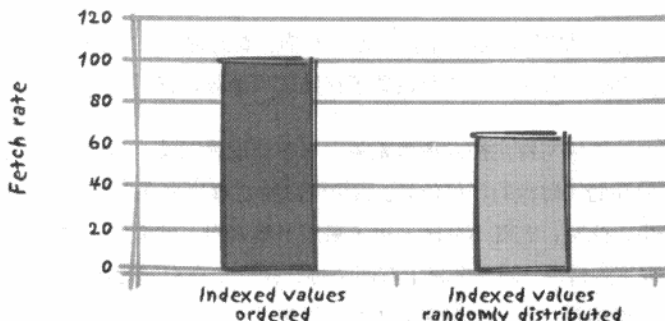


图 6-1：“索引项顺序与表中记录顺序是否一致”对性能的影响

现在很清楚了，“DBA 可以随时重新组织数据库”其实是错误的。数据库的重新组织曾一度流行；但不断增加的数据量及 99 999 9% 正常运行等要求，使得重新组织数据库变得不再适合。如果物理存储方式很重要，则应考虑第 5 章讨论过的“自组织结构（self-organizing structure）”之一，例如聚集索引（clustered indexe）或索引组织表（index-organized table）。但要记住，对某种类型的查询有利，可能对另一种类型的查询不利，鱼与熊掌不可得兼。

总结：类似的索引，性能却不同，这可能是物理数据的散布引起的。

条件的“可索引性”

Criterion Indexability

对“小结果集，直接条件”的情况而言，适当的索引非常重要。但是，其中也有不适合加索引的例外情况：以下案例，用来判断会计账目是否存在“金额不平”的情况，虽然可选择性很高，但不适合加索引。

此例中，有个表 `glreport`，该表包含一个应为 0 的字段 `amount_diff`。此查询的目的是要追踪会计错误，并找出 `amount_diff` 不是 0 的记录。既然使用了现代的 DBMS，直接把账目对应成表，并应用从前“纸笔记账”的逻辑，实在有点问题；但很不幸，我们经常遇到这种有问题的数据库。无论设计的质量如何，像 `amount_diff` 这样的字段

通常不应加索引，因为在理想情况下每条记录的 amount_diff 字段都是 0。此外，amount_diff 字段明显是“非规范化”设计的结果，大量计算要操作该字段。维护一个计算字段上的索引，代价要高于静态字段上的索引，因为被修改的键会在索引内“移动”，于是索引要承受的开销比简单节点增／删要高。

总结：并非所有明确的条件都适合加索引。特别是，频繁更新的字段会增加索引维护的成本。

回到例子。开发者有天来找我，说他已最佳化了以下 Oracle 查询，并询问过专家建议：

```
select
   total.deptnum,
   total.accounting_period,
   total.ledger,
   total.cnt,
   error.err_cnt,
   cpt_error.bad_acct_count
from
 -- First in-line view
 (select
     deptnum,
     accounting_period,
     ledger,
     count(account) cnt
 from
     glreport
 group by
     deptnum,
     ledger,
     accounting_period) total,
 -- Second in-line view
 (select
    deptnum,
    accounting_period,
    ledger,
    count(account) err_cnt
 from
    glreport
 where
    amount_diff <> 0
```

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
                   (Cost=1779554 Card=154 Bytes=16170)
   1    0   MERGE JOIN (OUTER) (Cost=1779554 Card=154 Bytes=16170)
   2    1     MERGE JOIN (OUTER) (Cost=1185645 Card=154 Bytes=10780)
   3    2       VIEW (Cost=591736 Card=154 Bytes=5390)
   4    3         SORT (GROUP BY) (Cost=591736 Card=154 Bytes=3388)
   5    4           TABLE ACCESS (FULL) OF 'GLREPORT'
                           (Cost=582346 Card=4370894 Bytes=96159668)
   6    2       SORT (JOIN) (Cost=593910 Card=154 Bytes=5390)
   7    6         VIEW (Cost=593908 Card=154 Bytes=5390)
   8    7           SORT (GROUP BY) (Cost=593908 Card=154 Bytes=4004)
   9    8             TABLE ACCESS (FULL) OF 'GLREPORT'
                             (Cost=584519 Card=4370885 Bytes=113643010)
  10    1     SORT (JOIN) (Cost=593910 Card=154 Bytes=5390)
  11   10       VIEW (Cost=593908 Card=154 Bytes=5390)
  12   11         SORT (GROUP BY) (Cost=593908 Card=154 Bytes=5698)
  13   12           TABLE ACCESS (FULL) OF 'GLREPORT'
                           (Cost=584519 Card=4370885 Bytes=161722745)


Statistics
----------------------------------------------------------
        193  recursive calls
          0  db block gets
    3803355  consistent gets
    3794172  physical reads
       1620  redo size
       2219  bytes sent via SQL*Net to client
        677  bytes received via SQL*Net from client
          4  SQL*Net roundtrips to/from client
         17  sorts (memory)
          0  sorts (disk)
         37  rows processed
```

在此说明，我没有浪费太多时间在执行计划上，因为查询本身的文字描述已显示了查询的最大特点：只有四~五百万条记录的 glreport 表，被访问了三次；每个子查询存取一次，而且每次都是完全扫描。

编写复杂查询时，嵌套查询通常很有用，尤其是你计划将查询划分为多个步骤，每个步骤对应一个子查询。但是，嵌套查询不是银弹，上述例子就属于“滥用嵌套查询”。

查询中的第一个内嵌视图，计算每个部门的账目数、会计期、分类账，这不可避免地要进行全表扫描。面对现实吧！我们必须完整扫描 glreport 表，因为检查有多少个账目涉及所有记录。但是，有必要扫描第二次甚至第三次吗？

总结：如果必须进行全表扫描，表上的索引就没用了。

不要单从“分析（analytic）”的观点看待处理，还要退一步，从整体角度考虑。除了在 amount_diff 值上的条件之外，第二个内嵌视图所做的计算，与第一个视图完全相同。我们没有必要使用 count() 计算总数，可以在 amount_diif 不是 0 时加 1，否则加 0，通过 Oracle 特有的 decode(u, v w, x) 函数，或使用标准语法 case when u = v then w else x end，即可轻松实现这项计算。

第三个内嵌视图所过滤的记录与第一个视图相同，但要计算不同账目数。把这个计数合并到第一个子查询中并不难：用 chr(1) 代表 amount_diff 为 0 时的“账户编号（account number）”，就很容易统计有多少个不同的账户编号了，当然，记住减 1 去掉 chr(1) 这个虚拟的账户编号。其中，账户编号字段的类型为 varchar2（注 1），而 chr(1) 在 Oracle 中代表 ASCII 码值为 1 的字符——在使用 Oracle 这类用 C 语言编写的系统时，我总是不敢安心使用 chr(0)，因为 C 语言 以 chr(0) 作为字符串终止符。

```
So this is the suggestion that I returned to the developer:
    select  deptnum,
            accounting_period,
            ledger,
            count(account) nb,
            sum(decode(amount_diff, 0, 0, 1)) err_cnt,
            count(distinct decode(amount_diff, 0, chr(1), account)) - 1
                                    bad_acct_count
    from
         glreport
    group by
          deptnum,
          ledger,
          accounting_period
```

这个新的查询，执行速度是原先的四倍。这丝毫不令人意外，因为三次的完整扫描变成了一次。

注意，查询中不再有 where 子句：amount_diff 上的条件已被“迁移”到了 select 列表中 decode() 函数执行的逻辑，以及由 group by 子句执行的聚合（aggregation）中。

注 1：对非 Oracle 用户而言，varchar2 实际上与 varchar 类型相同。

使用聚合代替过滤条件有点特殊，这正是我们要说明的“九种典型情况”中的另一种——以聚合函数为基础获得结果集。

总结：内嵌查询可以简化查询，但若使用不慎，可能造成重复处理。

小结果集，间接条件

Small Result Set, Indirect Criteria

与上一节类似，这一节也是要获取小结果集，只是查询条件不再针对源表，而是针对其他表。我们想要的数据来自一个表，但查询条件是针对其他表的，且不需要从这些表返回任何数据。典型的例子是在第 4 章讨论过的“哪些客户订购了特定商品”问题。如第 4 章所述，这类查询可用两种方法表达：

- 使用连接，加上 distinct 去除结果中的重复记录，因为有的客户会多次订购相同商品
- 使用关联或非关联子查询

如果可以使用作用于源表的条件，请参考前一节“小结果集，直接条件”中的方法。但如果找不到这样的条件，就必须多加小心了。

取用第 4 章中例子的简化版本，找出订购蝙蝠车的客户，典型实现如下：

```
select distinct orders.custid
from orders
     join orderdetail
       on (orderdetail.ordid = orders.ordid)
     join articles
       on (articles.artid = orderdetail.artid)
where articles.artname = 'BATMOBILE'
```

依我看，明确使用子查询来检查客户订单是否包含某项商品，才是较好的方式，而且也比较容易理解。但应该采用“关联子查询”还是“非关联子查询”呢？由于我们没有其他条件，所以答案应该很清楚：非关联子查询。否则，就必须扫描 orders 表，并针对每条记录执行子查询——当 orders 表规模小时通常不会查觉其中问题，但随着 orders 表越来越大，它的性能就逐渐让我们如坐针毡了。

非关联子查询可以用如下的经典风格编写：

```
select distinct orders.custid
from orders
where ordid in (select orderdetails.ordid
                from orderdetail
                     join articles
                       on (articles.artid = orderdetail.artid)
                where articles.artname = 'BATMOBILE')
```

或采用 from 子句中的子查询：

```
select distinct orders.custid
from orders,
     (select orderdetails.ordid
      from orderdetail
           join articles
             on (articles.artid = orderdetail.artid)
      where articles.artname = 'BATMOBILE') as sub_q
where sub_q.ordid = orders.ordid
```

我认为第一个查询较为易读，当然这取决于个人喜好。别忘了，在子查询结果上的 in() 条件暗含了 distinct 处理，会引起排序，而排序把我们带到了关系模型的边缘。

总结：如果要使用子查询，在选择关联子查询、还是非关联子查询的问题上，应仔细考虑。

多个宽泛条件的交集

Small Intersection of Broad Criteria

本节讨论对多个宽泛条件取交集获得较小结果集的情况。在分别使用各个条件时，会产生大型数据集，但最终各个大型数据集的交集却是小结果集。

继续上一节的例子。如果“判断订购的商品是否存在”可选择性较差，就必须考虑其他条件（否则结果集就不是小结果集）。在这种情况下，使用正规连接、关联子查询，还是非关联子查询，要根据不同条件的过滤能力和已存在哪些索引而定。

例如，由于不太畅销，我们不再检索订购蝙蝠车的人，而是查找上周六购买某种肥皂的客户。此时，我们的查询语句为：

```
select distinct orders.custid
from orders
    join orderdetail
      on (orderdetail.ordid = orders.ordid)
    join articles
      on (articles.artid = orderdetail.artid)
where articles.artname = 'SOAP'
  and <selective criterion on the date in the orders table>
```

这个处理流程很合逻辑，该逻辑和商品具有高可选择性时相反：先取得商品，再取得包含商品的明细订单，最后处理订单。对目前讨论的肥皂订单的情况而言，我们应该先取得在较短期间内下的少量订单，再检查哪些订单涉及肥皂。从实践角度来看，我们将使用完全不同的索引：第一个例子需要 orderdetail 表的商品名称、商品 ID 这两个字段上的索引，以及 orders 表的主键 orderid 上的索引；而此肥皂订单的例子需要 orders 表日期字段的索引、orderdetail 表的订单 ID 字段的索引，以及 articles 表的主键 orderid 上的索引。当然，我们首先假设索引对上述两例都是最佳方式。

要知道哪些客户在上星期六买了肥皂，最明显而自然的选择是使用关联子查询：

```
select distinct orders.custid
from orders
where <selective criterion on the date in the orders table>
  and exists (select 1
              from orderdetail
                 join articles
                   on (articles.artid = orderdetail.artid)
               where articles.artname = 'SOAP'
                 and orderdetails.ordid = orders.ordid)
```

在这个方法中，为了使关联子查询速度较快，需要 orderdetail 表的 ordid 字段上有索引（就可以通过主键 artid 取得商品，无需其他索引）。

第 3 章已提到，事务处理型数据库（transactional database）的索引是种奢侈，因为它处在经常更改的环境中，维护的成本很高。于是选择“次佳”解决方案：当表 orderdetail 上的索引并不重要，而且也有充足理由不再另建索引时，我们考虑以下方式：

```
select distinct orders.custid
from orders,
     (select orderdetails.ordid
      from orderdetail,
           articles
```

```
      where articles.artid = orderdetail.artid
        and articles.artname = 'SOAP') as sub_q
  where sub_q.ordid = orders.ordid
    and <selective criterion on the date in the orders table>
```

这第二个方法对索引的要求有所不同：如果商品数量不超过数百万项，即使 artname 字段上没有索引，基于商品名称条件的查询性能也不错。表 orderdetail 的 artid 字段可能也不需索引：如果商品很畅销，出现在许多订单中，则表 orderdetail 和 articles 之间的连接通过哈希或合并连接（merge join）更高效，而 artid 字段上的索引会引起嵌套的循环。与第一种方法相比，第二种方法属于索引较少的解决方案。一方面，我们无法承受为表的每个字段建立索引；另一方面，应用中都有一些“次要的”查询，它们不太重要，对响应时间要求也不苛刻，索引较少的解决方案完全满足它们的要求。

总结：为现存的查询增加搜索条件，可能彻底改变先前的构想：修改过的查询成了新查询。

多个间接宽泛条件的交集

Small Intersection, Indirect Broad Criteria

为了构造查询条件，需要连接（join）源表之外的表，并在条件中使用该表的字段，就叫间接条件（indirect criterion）。正如上一节“多个宽泛条件的交集”的情况，通过两个或多个宽泛条件的交集处理获取小结果集，是项艰难的工作；若是涉及多次 join 操作，或者对中心表（central table）进行 join 操作，则会更加困难——这是典型的“星形 schema（star schema）”（第 10 章详细讨论），实际的数据库系统中经常遇到。对于多个可选择性差的条件，一些罕见的组合要求我们预测哪些地方会执行完整扫描。当牵涉到多个表时，这种情况颇值得研究。

DBMS 引擎的执行始于一个表、一个索引或一个分区，就算 DBMS 引擎能并行处理数据也是如此。虽然由多个大型数据集合的交集所定义的结果集非常小，但前期的全表扫描、两次扫描等问题依然存在，还可能在结果上执行嵌套循环（nested loop）、哈希连接

（hash join）或合并连接（merge join）。此时，困难在于确定结果集的哪种表组合产生的记录数最少。这就好比，找到防线最弱的环节，然后利用它获得最终结果。

下面通过一个实际的 Oracle 案例说明这种情况。原始查询相当复杂，有两个表在 from 子句中都出现了两次，虽然表本身不太庞大（大的包含 700 000 行数据），但传递给查询的九个参数可选择性都太差：

```
select (data from ttex_a,
                  ttex_b,
                  ttraoma,
                  topeoma,
                  ttypobj,
                  ttrcap_a,
                  ttrcap_b,
                  trgppdt,
                  tstg_a)
from ttrcapp ttrcap_a,
     ttrcapp ttrcap_b,
     tstg tstg_a,
     topeoma,
     ttraoma,
     ttex ttex_a,
     ttex ttex_b,
     tbooks,
     tpdt,
     trgppdt,
     ttypobj
where ( ttraoma.txnum = topeoma.txnum )
  and ( ttraoma.bkcod = tbooks.trscod )
  and ( ttex_b.trscod = tbooks.permor )
  and ( ttraoma.trscod = ttrcap_a.valnumcod )
  and ( ttex_a.nttcod = ttrcap_b.valnumcod )
  and ( ttypobj.objtyp = ttraoma.objtyp )
  and ( ttraoma.trscod = ttex_a.trscod )
  and ( ttrcap_a.colcod = :0 ) -- not selective
  and ( ttrcap_b.colcod = :1 ) -- not selective
  and ( ttraoma.pdtcod = tpdt.pdtcod )
  and ( tpdt.risktyp = trgppdt.risktyp )
  and ( tpdt.riskflg = trgppdt.riskflg )
  and ( tpdt.pdtcod = trgppdt.pdtcod )
  and ( trgppdt.risktyp = :2 ) -- not selective
  and ( trgppdt.riskflg = :3 ) -- not selective
  and ( ttraoma.txnum = tstg_a.txnum )
  and ( ttrcap_a.refcod = :5 ) -- not selective
  and ( ttrcap_b.refcod = :6 ) -- not selective
  and ( tstg_a.risktyp = :4 ) -- not selective
  and ( tstg_a.chncod = :7) -- not selective
  and ( tstg_a.stgnum = :8 ) -- not selective
```

我们提供适当的参数（这里以 :0 到 :8 代表）执行此查询：耗时超过 25 秒，返回记录不到 20 条，做了 3 000 次物理 I/O，访问数据块 3 000 000 次。上述统计数据反映了实际执行的情况，这是必须首先明确的。下面，通过查询数据字典，得到表记录数情况：

```
TABLE_NAME                    NUM_ROWS
-------------------------- ----------
ttypobj                          186
trgppdt                          366
tpdt                            5370
topeoma                        12118
ttraoma                        12118
tbooks                         12268
ttex                          102554
ttrcapp                       187759
tstg                          702403
```

认真研究表及表的关联情况，得到图 6-2 所示的分析图：小箭头代表较弱的选择条件，方块为表，方块的大小代表记录数多少。注意：在中心位置的 `tTRaoma` 表，几乎和其他所有表有关联关系，但很不幸，选择条件都不在 `tTRaoma` 表。另一个有趣的事实是：上述的查询语句中，我们必须提供 `TRgppdt` 表的 `risktyp` 字段 和 `riskflg` 字段的值作为条件——为了连接（join）`TRgppdt` 表和 `tpdt` 表要使用这两个字段和 `pdtcod` 字段。在这种情况下，应该思考倒转此流程——例如把 `tpdt` 表的字段与所提供的常数做比较，然后只从 `trgppdt` 表取得数据。

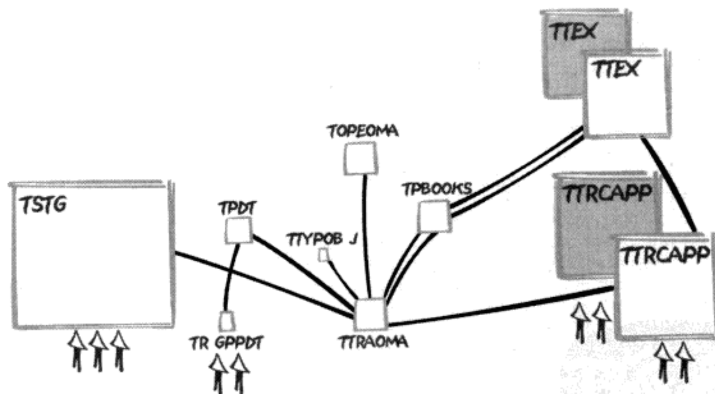


图 6-2：数据的位置关系

多数 DBMS 提供“检查优化器选择的执行计划”这一功能，比如通过 `explain` 命令直接检查内存中执行的项目。上述查询花了 25 秒（虽然不是特别糟），通常是先完整扫描 `tTRaoma` 表，接着进行一连串的嵌套循环，使用了各种高效的索引（详述这些索引

很乏味，我们假设所有字段都建立了合适的索引)。速度慢的原因是完整扫描吗？当然不是。为了证明完整扫描所花时间占的比例甚微，只需做如下简单的测试：读取 tTRaoma 表的所有记录；为了避免受到字符显示时间的干扰，这些记录无需显示。

优化器发现：tstg 表有“大量敌军”，而查询中针对此表的选择条件比较弱，所以难以对它形成“正面攻击”；而 ttrcapp 表在查询的 from 子句中出现两次，但基于该表的判断条件也较弱，所以也不会带来查询效率的提升；但是，ttraoma 表的位置显然很关键，且该表比较小，适合作为“第一攻击点”——优化器会毫不犹豫地这么做。

那么，既然对 tTRaoma 表的完整扫描无可厚非，优化器到底错在哪里呢？请看图 6-3 所示的查询执行情况。

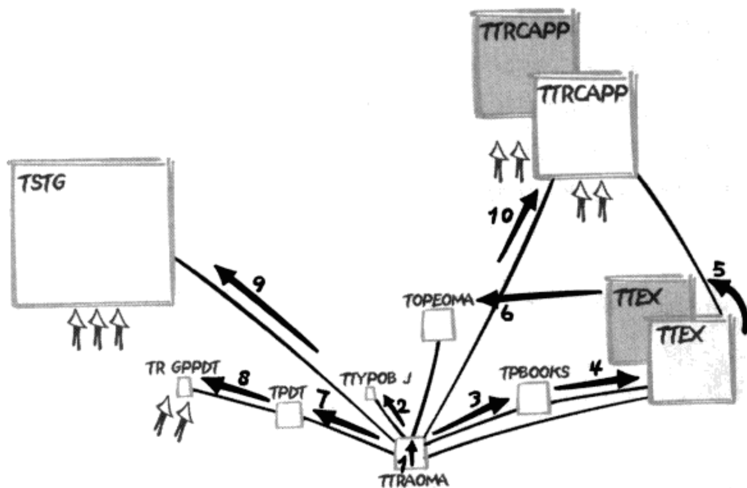


图 6-3：优化器选择的执行路径

注意观察图中所示的操作执行顺序，查询速度慢的原因显露无遗：我们的查询条件很糟糕，优化器选择完全忽略它们。优化器决定先对 ttraoma 表进行完整扫描；接着，访问和表 ttraoma 关联的所有小型表；最后，对其他表运用我们的过滤条件。这样执行是错误的：虽然优化器决定首先访问表 ttraoma 有道理（该表的索引可能非常高效，每个键平均对应的记录数较少，或者索引与记录的顺序有较好的对应关系），但将我们提供的查询条件推迟执行，不利于减少要处理的数据量。

既然已访问了 ttraoma 这个关键表，应该紧接着执行语句中的查询条件，这样可以借助这些表与 ttraoma 表之间的连接（join）先去除 ttraoma 表中无用的记录——甚至在结果集更大时，如此执行的效率仍比较高。但是上述信息我们知道，“优化器”却无从知道。

怎样才能迫使 DBMS 依我们所要求的方式执行查询呢？要依靠 SQL 方言（SQL dialect）。正如你将在第 11 章看到的，多数 SQL 方言都支持针对优化器的指示或提示（hint），虽然各种方言所用语法不同；例如，告诉优化器按表名在 from 子句中出现的顺序依次访问各表。不过，“提示”的实际影响远比它的名字暗示的要大得多，采用“提示”的问题在于，每个提示都是在“赌未来”——我们已强制规定了执行路径，所以环境、数据量、数据库算法、硬件等因素的发展变化即使不能绝对适合我们的执行路径，也应该基本适合。例如，既然索引的嵌套循环是最高效选择，并且嵌套循环不会因并行化而受益，那么命令优化器按照表的排列顺序访问它们几乎没什么风险。明确指定表的访问顺序，就是这个案例中实际采用的方法，最终查询不到 1 秒即可完成，不过物理 I/O 次数减少并不明显（原来 3 000 次，现在 2 340 次，因为我们仍以 ttraoma 表的完整扫描开始），但逻辑 I/O 次数的大幅降低（从 3 000 000 次降到 16 500 次）使总体响应时间显著缩短，因为我们“建议”了更高效的执行路径。

总结：记住，你应该详细说明所有强迫 DBMS 做的事。

显式地通过优化器指令，指定表的访问顺序，是个笨拙的方法。更优雅的方法是在 from 子句中采用嵌套查询，在数值表达式中建议连接关系，这样不必大幅修改 SQL 子句：

```
select (select list)
from (select ttraoma.txnum,
             ttraoma.bkcod,
             ttraoma.trscod,
             ttraoma.pdtcod,
             ttraoma.objtyp,
            ...
      from ttraoma,
           tstg tstg_a,
           ttrcapp ttrcap_a
     where tstg_a.chncod = :7
       and tstg_a.stgnum = :8
       and tstg_a.risktyp = :4
       and ttraoma.txnum = tstg_a.txnum
       and ttrcap_a.colcod = :0
       and ttrcap_a.refcod = :5
       and ttraoma.trscod = ttrcap_a.valnumcod) a,
     ttex ttex_a,
     ttrcapp ttrcap_b,
     tbooks,
     topeoma,
     ttex ttex_b,
     ttypobj,
     tpdt,
     trgppdt
where ( a.txnum = topeoma.txnum )
 and ( a.bkcod = tbooks.trscod )
 and ( ttex_b.trscod = tbooks.permor )
 and ( ttex_a.nttcod = ttrcap_b.valnumcod )
 and ( ttypobj.objtyp = a.objtyp )
 and ( a.trscod = ttex_a.trscod )
 and ( ttrcap_b.colcod = :1 )
 and ( a.pdtcod = tpdt.pdtcod )
 and ( tpdt.risktyp = trgppdt.risktyp )
 and ( tpdt.riskflg = trgppdt.riskflg )
 and ( tpdt.pdtcod = trgppdt.pdtcod )
 and ( tpdt.risktyp = :2 )
 and ( tpdt.riskflg = :3 )
 and ( ttrcap_b.refcod = :6 )
```

通常，没有必要采用非常具体的方式和难以理解的提示，其实，提供正确的最初指导就可使优化器找到正确的执行路径。嵌套查询是个不错的选择，它使表的关联变得明确，而 SQL 语句的阅读也相当容易。

总结：混乱的查询会让优化器困惑。结构清晰的查询及合理的连接建议，通常足以帮助优化器提升性能。

大结果集

Large Result Set

无论结果集是如何获得的，只要结果集“很大”，就符合我们下面要讨论的“大结果集”的情况。批处理环境下，产生大结果集是明智的。当需要返回大量记录时，只要查询条件的可选择性不高，那么即使结果集只占表中数据量的一小部分，也会引起 DBMS 引擎执行全表扫描；只有某些数据仓库例外，我们将在第 10 章中讨论之。

如果查询返回几万条记录，那么使用索引是没有意义的，无论索引用于产生最终结果，还是用于复杂查询的中间步骤。相比而言，借助哈希或合并连接进行全表扫描是合适的。当然，强力手段背后也必须有智慧：我们必须尽量扫描数据返回比例最高的表、索引，或者这两者的分区；扫描时的过滤条件必须是粗粒度的，从而返回的数据量比较大，使扫描更有价值；扫描显然违背了“尽快去除不必要数据”这一原则，但一旦扫描结束应立即重新贯彻该原则。

相反，采取扫描方式不合适的情况下，应尽量减少要访问数据的块数。为此，最常用的手段就是使用索引（而不是表），尽管所有索引的总数据量经常比表还大，但单个索引则远比表要小。如果索引包含了所有需要的信息，则扫描索引而不扫描表是相当合理的，可以利用诸如聚集索引等避免访问表的技术。

无论是要返回大量记录，还是要对大量记录进行检查，每条记录的处理都需小心。例如，一个性能不佳的用户自定义函数的调用，如果发生在“返回小结果集的 `select` 列表”中或在“可选择性很高的 `where` 子句”中，则影响不大；但返回大数据集的查询可能会调用这个函数几十万次，DBMS 服务器就不堪重负了，这时必须优化代码。

还要重点关注子查询的使用。处理大量记录时，关联子查询（Correlated subquery）是性能杀手。当一个查询包含多个子查询时，必须让它们操作各不相同、自给自足的数据子集，以避免子查询相互依赖；到查询执行的最后阶段，多个子查询分别得到的不同数据集经过哈希连接或集合操作得到结果集。

查询执行的并行化（parallelism）也是个好主意，不过只应在"并发活动会话数（concurrently active sessions）"很少（典型情况为批处理操作）时才这么做。并行化是由 DBMS 实现的，如果有可能，DBMS 把一个查询分割为多个并行运行的子任务，并由另一个专门的任务来协调。并发用户数很大时，并行化反而会影响处理能力。一般而言，并发用户数又多、要处理的信息量又大的情况下，最好做好战斗准备，因为这经常靠投入更多硬件来解决。

除了处理过程中由资源争用引起的等待之外，查询必须访问的数据量是影响"响应时间"的主要因素。但正如第 4 章讲过的，最终用户并不关心客观的数据量分析，他们只关心查询获得的数据。

基于一个表的自连接

Self-Joins on One Table

利用卓越的、广为流行的范式（注 2），有助于我们设计正确的关系数据库（至少满足 3NF）。所有非键字段均与键相关、并完整依赖于键，非键字段之间没有任何依赖。每条记录具有逻辑一致性，同一个表中没有重复记录。于是，才能够建立同一个表之间的连接关系：使用同一查询从同一表中选择不同记录的集合（可以相交），然后连接它们，就好像它们来自不同表一样。本节将讨论简单的自连接。本节不讨论较复杂的嵌套层次结构，这一主题在第 7 章中讨论。

自连接，指表与自身的连接，这种情况比分层查询更常见。自连接用于"从不同角度看

注 2：1983 年，William Kent 写过一篇关于范式的论文，在 *http://www.bkent.net* 可以找到。

待相同数据”的情况，例如，查询航班会两次用到 airports 表，一次找到“出发机场”的名称，另一次找出“到达机场”的名称：

```
select f.flight_number,
       a.airport_name departure_airport,
       b.airport_name arrival_airport
from flights f,
     airports a,
     airports b
where f.dep_iata_code = a.iata_code
  and f.arr_iata_code = b.iata_code
```

此时，一般规则仍然适用：重点保证索引访问的高效。但是，如果此时索引访问不太高效怎么办呢？首当其冲地，应避免“第一轮处理丢弃了第二轮处理所需的记录”。应该通过一次处理收集所有感兴趣的记录，再使用诸如 case 语句等结构分别显示记录，第 11 章将详细说明这种方法。

非常微妙的是，有些情况看似与“机场的例子”很像，但其实不然。例如，如何利用一个保存“定期累计值”（注 3）的表，显示每个时间段内累计值的增量？此时，该表内的两个不同记录间虽然有关联，但这种关联很弱：两个记录之所以相关，是因为它们的时间戳之间有前后关系。而连接两个 flights 表是通过 airports 表进行的，这种关联很强。

例如，时间段为 5 分钟，时间戳以“距参照日期多少秒（seconds elapsed since a reference date）”表示，则查询如下：

```
select a.timestamp,
       a.statistic_id,
       (b.counter - a.counter)/5 hits_per_minute
from hit_counter a,
     hit_counter b
where b.timestamp = a.timestamp + 300
  and b.statistic_id = a.statistic_id
order by a.timestamp, a.statistic_id
```

注 3：从 Oracle 的 V$视图中收集数据就是如此，其中包含监控信息。

上述脚本有重大缺陷：如果第二个累计值不是正好在第一个累计值之后5分钟取得的，那么就无法连接这两条记录。于是，我们改以“范围条件”定义连接。查询如下：

```
select a.timestamp,
       a.statistic_id,
      (b.counter - a.counter) * 60 /
         (b.timestamp - a.timestamp) hits_per_minute
from  hit_counter a,
      hit_counter b
where b.timestamp between a.timestamp + 200
                      and a.timestamp + 400
  and b.statistic_id = a.statistic_id
order by a.timestamp, a.statistic_id
```

这个方法还是有缺陷：前后两次计算累计值的时间间隔，如果不介于 200 到 400 秒之间（例如取样频率改变了），如此之大的时间跨度就会引起风险。

我们还有更安全的方法，就是使用基于“记录窗口（windows of rows）”的 OLAP 函数（OLAP function）。难以想象，这种本质上不太符合关系理论的技术可以显著提升性能，但应作为查询优化的最后手段使用。借助 partition 子句，OLAP 函数支持“分别处理结果集的不同子集”，比如分别对它们进行排序、总计等处理。借助 OLAP 函数 row_number()，可以根据 statistic_id 建立子集，然后按时间戳增大的顺序为不同统计赋予连续整数编号，接下来，就可以连接 statistic_id 和两个序号了，如下例子所示：

```
select a.timestamp,
      a.statistic_id,
      (b.counter - a.counter) * 60 /
            (b.timestamp - a.timestamp)
from (select timestamp,
            statistic_id,
            counter,
            row_number( ) over (partition by statistic_id
                                order by timestamp) rn
      from hit_counter) a,
      (select timestamp,
             statistic_id,
             counter,
             row_number( ) over (partition by statistic_id
                                order by timestamp) rn
      from hit_counter) b
where b.rn = a.rn + 1
  and a.statistic_id = b.statistic_id
order by a.timestamp, a.statistic_id
```

Oracle 等 DBMS 支持 OLAP 函数 lag(column_name, n)。该函数借助分区()和排序()，返回 column_name 之前的第 n 个值。如果使用 lag()函数，我们的查询甚至执行得更快——比先前的查询大约快 25%。

```
select timestamp,
       statistic_id,
       (counter - prev_counter) * 60 /
       (timestamp - prev_timestamp)
from (select timestamp,
             statistic_id,
             counter,
             lag(counter, 1) over (partition by statistic_id
                                    order by timestamp) prev_counter,
             lag(timestamp, 1) over (partition by statistic_id
                                      order by timestamp) prev_timestamp
      from hit_counter) a
order by a.timestamp, a.statistic_id
```

很多时候，我们的数据并不像航班案例中那样具有对称性。通常，当需要查找和最小、最大、最早、或最近的值相关联的数据时，首先必须找到这些值本身（此为第一遍扫描，需比较记录），接下来的用这些值作为第二遍扫描的搜索条件。而以滑动窗口（sliding window）为基础的 OLAP 函数，可以将两遍扫描合而为一（至少表面上如此）。基于时间戳或日期的数据查询，非常特殊也非常重要，本章在稍后的“基于日期的简单搜索或范围搜索”中专门讨论。

总结：当多个选取条件用于同一个表的不同记录时，可以使用基于滑动窗口工作的函数。

通过聚合获得结果集

Result Set Obtained by Aggregation

本节讨论一类极常见的情况：对一个或多个主表（main table）中的详细数据进行汇总，动态计算出结果集。换言之，我们面临数据聚合（aggregation of data）的问题。此时，结果集大小取决于 group by 的字段的基数，而不是查询条件的精确性。正如第一节“小结果集，直接条件”中所述，对表进行一趟（a single pass）处理获得的并非真正聚合的结果（否则就需要自连接和多次处理），但此时聚合函数（或聚合）也相当有用。实际

上，最让人感兴趣的 SQL 聚合使用技巧，不是明显需要 sum 或 avg 的情况，而是如何将过程性处理转化为以聚合为基础的纯 SQL 替代方案。

如第 2 章所强调的，编写高效 SQL 代码的关键，第一是“勇往直前”，即不要预先检查，而是查询完成后测试是否成功 —— 毕竟，蹑手蹑脚地用脚趾试水赢不了游泳比赛。第二是尽量把更多“动作”放到 SQL 查询中，此时聚合函数特别有用。

优秀 SQL 编程的困难，多半在于解决问题的方式：不要将“一个问题”转换成对数据库的“一系列查询”，而是要转换成“少数查询”。程序用大量中间变量保存从数据库读出的值，然后根据变量进行简单判断，最后再把它们作为其他查询的输入……这样做是错误的。糟糕的 SQL 编程有个显著特点，就是在 SQL 查询之外存在大量代码，以循环的方式对返回数据进行些加、减、乘、除之类的处理。这样做毫无价值、效率低下，这里工作应该交给 SQL 的聚合函数。

注意

> 聚合函数非常有用，可以解决不少 SQL 问题（第 11 章会再次讨论）。然而，我发现开发者通常只使用最平常的聚合函数 `count()`，它对大多数程序是否真的有用值得怀疑。

第 2 章说明了使用 `count(*)` 判定是否要更新记录（插入新记录）是很浪费的。你可能在报表中误用了 `count(*)`。测试存在性有时会以模仿布尔值的方式实现：

```
case count(*)
when 0 then 'N'
else 'Y'
end
```

对于上述实现，只要存在与条件相符的记录，就会读取其中每条记录。其实，只需找到一条记录就足以判断要显示 Y 还是 N，通过测试存在性或限制返回记录数可以写出更高效的语句，一旦发现条件相符就停止处理即可。

当要解决的问题与最多、最少、最大、第一、最后有关时，聚合函数（可能会当成 OLAP 函数使用）很可能是最佳选择。也就是说，不要认为聚合函数仅支持 count、sum、max、min、avg 等功能，否则就说明你还没有充分理解聚合函数。

有趣的是，聚合函数在作用范围上非常狭窄。除了计算最大值和最小值，它们唯一能做的就是简单的算术运算：count()每遇到的一行加 1；avg()一方面将字段值累加，另一方面不断加 1 计数，最后进行除法运算。

聚合函数有时可取得令人吃惊的效果，比如通过 sum 就可以做很多事情。喜欢数学的朋友知道，通过对数和次方函数，要在 sum 和乘积（product）之间转换有多简单。喜欢逻辑的朋友也会知道 OR 很依赖 sum，而 AND 很依赖乘积。

下面通过简单的例子说明聚合的强大作用。假设要进行装运（shipment）处理，一次装运由一些不同的订单组成，每张订单都必须分别做准备；只有装运涉及的每张订单都完成时装运才准备就绪。问题就是，如何判断装运涉及的所有订单都已完成。

这样的情况常会发生，有多种方法可以判定装运是否就绪。最糟的方法是逐一判断每批装运，而每批装运内部进行第二个循环，查看有多少张订单的 order_complete 字段值为“N”，并返回计数为 0 的装运 ID。更好的解决方案是理解“‘N’值的不存在性测试”的意图，并用子查询（无论是关系还是非关系）完成：

```
select shipment_id
from shipments
where not exists (select null from orders
                  where order_complete = 'N'
                    and orders.shipment_id = shipments.shipment_id)
```

如果表 shipments 上没有其他条件了，则上述方法很糟糕，当 shipments 表数据量大时（而且未完成订单占少数），换成以下查询会更高效：

```
select shipment_id
from shipments
where shipment_id not in (select shipment_id
                          from orders
                          where order_complete = 'N')
```

上述查询也可以稍作变形，优化器比较喜欢这个变形，但要求 orders 表 的 shipment_id 字段上有索引：

```
select shipments.shipment_id
from shipments
   left outer join orders
        on orders.shipment_id = shipments.shipment_id
        and orders.order_complete = 'N'
where orders.shipment_id is null
```

另一个替代方案是借助集合操作，该集合操作会使用 shipments 主键索引，且对 orders 表进行全表扫描：

```
select shipment_id
from shipments
except
select shipment_id
from orders
where order_complete = 'N'
```

注意，并非所有 DBMS 都实现了 except 操作符，有的 DBMS 称之为 minus。

还有一种方法。主要是对装运中所有订单执行逻辑 AND 操作，将 order_complete 为 TRUE 的订单的 ID 返回。这类操作在现实中很常见。如前所述， AND 和乘法、OR 和加法之间关系密切。关键是把诸如“Y” 和 “N” 的 flag 值转换为 0 和 1，使用 case 结构即可。要把 order_complete 转成 0 或 1 的值可以这样写：

```
select shipment_id,
      case when order_complete = 'Y' then 1
                                     else 0
      end flag
from orders
```

到目前为止，一切顺利。如果每批装运包含的订单数固定的话，则很容易对适当字段进行 sum 后检查是否为预期订单数。然而，实际上希望每批装运的 flag 值相乘，并检查结果是 0 或是 1。这个方法是可行的，因为只要有一张以 0 表示的未完成订单，乘法的最后结果就是 0。乘法运算可由对数运行协助完成（虽然在以对数处理时，0 不是最简单的值），但我们这个例子要做的甚至更简单。

我们想要的是“第一张订单已完成、且第二张订单已完成……且第 n 张订单已完成”。德摩根定律（laws of de Morgan）（注 4）告诉我们，这等价于“第一张订单未完成、或第二张订单未完成……或第 n 张订单未完成”的情况“不成立”。由于使用聚合时，OR 比 AND 更容易处理。检查由 OR 连结的一连串条件是否不成立，比检查由 AND 连结的一连串条件是否成立，要容易得多。我们要考虑的真正“谓词（predicate）”是“订单未完成”，并对 order_complete 标志作转换，如果是 N 就转换为 1，如果是 Y 就转换为 0。之后，通过加总 flag 值，就可检查是否所有订单的 flag 值都是 0（都已完成）——如果总和是 0，所有订单都已完成。

因此，查询可写成：

```
select shipment_id
from (select shipment_id,
             case when order_complete = 'N' then 1
                                        else 0
             end flag
      from orders) s
group by shipment_id
having sum(flag) =0
```

甚至可以写得更简洁：

```
select shipment_id
from orders
group by shipment_id
having sum(case when order_complete = 'N' then 1
                                      else 0
           end) =0
```

还有更简单的方法。使用另一个聚合函数，而不必转换任何的 flag 值。注意，从字母的顺序来看，“Y” 大于 “N”，如果所有的值都是 “Y”，则最小值就是 “Y”。于是：

```
select shipment_id
from orders
group by shipment_id
having min(order_complete) = 'Y'
```

注 4：Augustus de Morgan (1806—1871)，生于印度，英国数学家，在数学的许多领域都有贡献，尤以逻辑领域贡献最大。德摩根定律规定：任意数量集合的交集的补集，等价于这些集合的并集；任意数量集合的并集的补集，等价于这些集合的交集（the complement of the intersection of any number of sets equals the union of their complements and that the complement of the union of any number of sets equals the intersection of their complements.）。如果你记得 SQL 是关于集合的，并记得“条件取否返回的结果集，即为原始条件返回的结果集的补集”，你就可以理解为什么德摩根定律对 SQL 实践者特别有用了。

这个方法利用了“Y”大于 “N”，而没有考虑标志转换为数值。本方法更高效。

上例使用了 group by，并以 order_complete 值最小作为查询条件，那么，其中不同的子查询（或作为子查询替代品的聚集函数）之间是如何比较的呢？如果先做 sum 操作而后检查总和是否为 0，必然导致整个 orders 表排序。而上例中使用了不太常见的聚合函数 min，一般比其他查询快，其他查询因访问两个表（shipments 和 orders）而速度较慢。

先前的例子大量使用了 having 子句。如第 4 章所述，“粗心的 SQL 语句”往往和在聚合语句中使用 having 子句有关。下面这个查询（Oracle）就是一例，它要查询过去一个月内每个产品的每周销售情况：

```
select product_id,
       trunc(sale_date, 'WEEK'),
       sum(sold_qty)
from sales_history
group by product_id, trunc(sale_date, 'WEEK')
having trunc(sale_date, 'WEEK') >= add_month(sysdate, -1)
```

这里的错误在于，having 子句中的条件没有使用聚合。于是，DBMS 必须处理 sales_history 中的每条记录，进行排序操作、进行聚合操作……然后过滤掉过时的数值，最后返回结果。这类错误并不引人注意，直到 sales_history 表数据量变得非常大为止。当然，正确的方法是把条件放在 where 子句中，确保过滤会发生在早期阶段，而之后要处理的数据集已大为减小。

必须指出：对视图（即聚合的结果）应用条件时，如果优化器不够聪明，没有在聚合前再次注入过滤条件，我们就会遇到完全相同的问题。

有些过滤条件生效太晚，应该提前，可做如下修改：

```
select customer_id
from orders
where order_date < add_months(sysdate, -1)
group by customer_id
having sum(amount) > 0
```

在这个查询中，以下 having 的条件乍看起来相当合理：

```
having sum(amount) > 0
```

然而，如果 amount 只能是正数或零，这种 having 用法就不合理。最好改为：

```
where amount > 0
```

此例中，group by 的使用分两种情况。首先：

```
select customer_id
from orders
where order_date < add_months(sysdate, -1)
  and amount > 0
group by customer_id
```

我们注意到，group by 对聚合计算是不必要的，可以用 distinct 取代它，并执行相同的排序和消除重复项目的工作：

```
select distinct customer_id
from orders
where order_date < add_months(sysdate, -1)
  and amount > 0
```

把条件放在 wher e 子句中，能让多余的记录尽早被过滤掉，因而更高效。

总结：聚合操作的数据应尽量少。

基于日期的简单搜索或范围搜索
Simple or Range Searching on Dates

搜索条件有多种，其中日期（和时间）占有特殊地位。日期极为常见，而且比其他数据类型更可能成为范围搜索的条件，范围搜索可以是有界的（如“在某两天之间”），也可以是部分有界（“在某天之前”）。通常，为了获得这种结果集，查询需要使用当前日期（如“前六个月”）。

上一节“通过聚合获得结果集”所举的例子，用到了 sales_history 表。当时，条件位于 amount 上，其实对于 sales_history 这种表更常见的是日期条件，尤其是读取特定日期的数据、或读取两个日期之间的数据。在保存历史数据的表中查找特定日期(或其对应值）时，必须特别注意确定当前日期的方法，它可能成为聚合条件的基础。

第 1 章已指出，设计保存历史数据的表颇为困难，而且没有现成的简单解决方案。无论你对当前数据、还是历史数据感兴趣，设计历史数据的存储方案都要根据如何使用数据决定，同时还要看数据多快会过时。例如，零售系统中价格的变动速度比较慢（除非正在经受严重的通货膨胀），而网络流量或财务设备的价格改变速度比较快，甚至快很多。

从宏观角度来看，关键是各项历史数据的数量：是“少量数据项、大量历史数据”，还是“大量数据项、少量历史数据”，或是介于两者之间？其重点是：数据项的可选择性取决于数据项的总数、取样频率（“每天一次”还是“每次改变时”）、时间长短（“永久”还是“一年”等）。因此，本节将首先讨论“大量数据项、少量历史数据”的情况，接着讨论“少量数据项、大量历史数据”的情况，最后讨论当前值问题。

大量数据项、少量历史数据

Many Items, Few Historical Values

既然没有为每个数据项保留大量历史数据，那么各项的 ID 可选择性很高。说明要查询哪些项，限定参与查询的少数历史记录，就可确定特定日期（当前日期或以前日期）对应的值。这种情况需要我们再次处理聚合值（aggregate value）。

除非建立了代理键（本情况不需要代理键），否则主键通常是复合键，由 `item_id` 和 `record_date` 组成。为了查询特定日期的值，可采用两种方法：子查询和 OLAP 函数。

使用子查询

查找某数据项在特定日期的值相对简单，但实际上，这种简单只是假象。通常你会遇到这样的代码：

```
select whatever
from hist_data as outer
where outer.item_id = somevalue
  and outer.record_date = (select max(inner.record_date)
                           from hist_data as inner
                           where inner.item_id = outer.item_id
                             and inner.record_date <= reference_date)
```

考察这个查询的执行路径，我们发现：首先，内层查询与外层查询是有关联的（correlated），因为内层查询参照了 item_id 的值，该值是由外层查询返回的当前记录一个字段。下面，先来分析外层查询。

理论上，复合键中的字段顺序不会有太大影响，但实际上它们非常重要。如果我们误把主键定义为 (record_date, item_id)，而不是 (item_id, record_date)，前例的内层查询就非常依赖 item_id 字段的索引，否则无法高效地向下访问树状结构索引。但我们知道，额外增加一个索引的代价很高。

外层查询找到了保存 item_id 历史的各条记录，接着使用当前 item_id 值逐次执行子查询。注意，内层查询只依赖 item_id，这与外层查询处理的记录相同，这意味着我们执行相同的查询、返回相同的结果。优化器会注意到查询总是返回相同的值吗？无法确定。所以最好不要冒这个险。

在使用关联子查询时，如果它处理不同的记录后总是返回相同的值，就没有意义了。所以，应该改用无关联子查询：

```
select whatever
from hist_data as outer
where outer.item_id = somevalue
  and outer.record_date = (select max(inner.record_date)
                           from hist_data as inner
                           where inner.item_id = somevalue
                             and inner.record_date <= reference_date)
```

现在子查询的执行不需要访问表，只需访问主键索引就够了。

个人习惯各有不同，但如果 DBMS 支持将"子查询的输出"与多个字段进行比较（这个特性不是所有产品都支持的），则应优先考虑基于主键比较：

```
select whatever
from hist_data as outer
where (outer.item_id, outer.record_date) in
                          (select inner.item_id, max(inner.record_date)
                           from hist_data as inner
                           where inner.item_id = somevalue
                             and inner.record_date <= reference_date
                           group by inner.item_id)
```

让子查询返回的字段，完全与复合主键的字段相符，有一定道理。如果必须返回"数据项值的列表"（例如是另一个查询的结果），则上述查询语句建议的执行路径非常合适。

只要每个数据项的历史信息数量都较少，以 in() 列表或子查询取代内层查询中的somevalue，会使整个查询执行更高效。也可以用 in 子句取代“相等性条件”，在多数情况下没有什么不同；但偶有例外，例如，如果用户输错了 item_id，采用 in()时会返回未发现数据，而采用“相等性条件”时会返回错误数据。

使用 OLAP 函数

我们在自连接（self-join）情况下，使用了诸如 row_number()等 OLAP 函数，它们在查询“特定日期某数据项的值”时也同样有用甚至高效。（但记住，OLAP 函数会带来非关系的处理模式（注 5）。）

注意

OLAP 函数属于 SQL 的非关系层。这类函数的作用是：在查询中做最后（或几乎是最后）处理。因为它们在过滤已完成后对结果集进行处理。

运用 row_number()等函数，可以通过日期排序判断数据的“新旧程度（degree of freshness）”（也就是距离现在有多久）：

```
select row_number( ) over (partition by item_id
                           order by record_date desc) as freshness,
       whatever
from hist_data
where item_id = somevalue
  and record_date <= reference_date
```

选取最新数据，只需保留 freshness 值为 1 的记录：

```
select x.<suitable_columns>
from (select row_number( ) over (partition by item_id
                                 order by record_date desc) as freshness,
             whatever
      from hist_data
      where item_id = somevalue
        and record_date <= reference_date) as x
where x.freshness = 1
```

注 5：……尽管“OLAP”一词是 E.F. Codd 博士本人在 1993 的论文里提出的。

理论上，使用 OLAP 函数方法和子查询几乎没有差异。实际上，OLAP 函数只访问一次表，即使需要为此而进行排序操作也不例外。OLAP 函数对表不需要做额外的访问，甚至在使用主键快速存取时也是如此。因此，采用 OLAP 函数速度会比较快（尽管只是快一点点）。

少量数据项、大量历史数据

Many Historical Values Per Item

当存在大量历史数据时，情况有所不同 —— 例如，监控系统中采集“度量值”的频率很高。这里的困难在于，必须根据对极大量的数据进行排序，才能找到特定日期或最接近特定日期的值。

排序是代价很高的操作：如果我们应用第 4 章的原则，降低非关系层厚度的唯一方法，就是在关系层多做一些工作，增加过滤条件的数量。此时，针对所需数据更精确地归类日期（或时间）以缩小范围，便非常重要。如果我们只提供上限，就必须扫描并排序所有历史数据。所以如果数据的采集频率很高，提供下限是有必要的。如果我们成功地把记录的“工作集”控制在可管理的大小，就相当于回到了“少量历史记录”的情况。如果无法同时指定上限（例如当前日期）和下限，我们的唯一希望就是根据数据项分区；我们只需在单一分区上操作，这比较接近“大结果集”的情况。

当前值

Current Values

当我们只对最近或当前值感兴趣时，如何避免使用嵌套子查询或 OLAP 函数（两者都引起排序）而直接找到适当值，是非常吸引人的设计。如第 1 章所述，解决该问题的方法之一，就是把每个值与某个“截止日期”相关联 —— 就像麦片外盒上的“保质期(best before)”一样——并让当前值的“截止日期”是遥远的未来（例如公元 2999 年 12 月 31 日）。这种设计存在一些与实际相关的问题，下面讨论这些问题。

使用“固定日期”，确定当前值变得非常容易。查询如下所示：

```
select whatever
from hist_data
where item_id = somevalue
  and record_date = fixed_date_in_the future
```

接着，通过主键找到正确的记录。（当然，要参照的日期如果不是当前日期，就必须使用子查询或 OLAP 函数了。）然而，这种方法有两个主要缺点。

- 较明显的缺点：插入新的历史数据之前，先要更新“当前值”（例如今天），接着，将最新“当前值”和历史数据一起插入表中。这个过程导致工作量加倍。更糟的是，关系理论中的主键用于识别记录，但具有唯一性的(item_id, record_date)却不能作为主键，因为我们会对它做“部分更新（partially update）”。因此，必须有一个能让外键参照的代理键（ID 字段或序列号），结果程序变得更加复杂。大型历史表的麻烦就是，通常它们也经历过高频率的数据插入，所以数据量才会这么大。快速查询的好处，能抵销缓慢插入的缺点吗？这很难说，但绝对是个值得考虑的问题。

- 还有个微妙的缺点与优化器有关。优化器使用各种详细程度不同的统计数据，检查字段的最低值和最高值，尝试评估值的分布情况。假设历史表包含了自 2000 年 1 月 1 日开始的历史数据。于是，我们的数据组成是“散布在几年间的 99.9% 的历史数据”加上“2999 年 12 月 31 日的 0.1% 的‘当前数据’”。因此，优化器会认为数据散布在一千年的范围内。优化器在数据范围上的偏差是由于查询中出现的上限日期的误导（即“and record_date = fixed_date_in_the future”）。此时的问题就是，如果你当查询的不是当前值（例如，你要统计不同时段的数据变化），优化器可能错误地做出“使用索引”的决定——因为你访问的只是千年中的极小部分——但实际上需要的是对数据进行扫描。是优化器的评估偏差导致它做出完全错误的执行计划决定，这很难修正。

总结：要理解优化器如何看待你的系统，就必须理解你的数据和数据分布方式。

结果集和别的数据存在与否有关

Result Set Predicated on Absence of Data

一个表中的哪些记录和另一个表中的数据不匹配？这种“识别例外”的需求经常出现。人们最常想到的解决方案有两个：not in ()搭配非关联子查询，或者 not exists()

搭配关联子查询。一般认为应该使用 not exists。在子查询出现在高效搜索条件之后，使用 not exists 是对的，因为高效过滤条件已清除大量无关数据，关联子查询当然会很高效。但当子查询恰好是唯一条件时，使用 not in 比较好。

查找在另一个表无对应数据的记录时，会碰到一些奇特的解决方案。以下为实际例子，显示哪些数据库查询代价最高。注意，问号是占位符(placeholder)或称为绑定变量(bind variable)，它们的具体值在后续执行中传递给查询：

```
insert into ttmpout(custcode,
                    suistrcod,
                    cempdtcod,
                    bkgareacod,
                    mgtareacod,
                    risktyp,
                    riskflg,
                    usr,
                    seq,
                    country,
                    rating,
                    sigsecsui)
select distinct custcode,
                ?,
                ?,
                ?,
                mgtareacod,
                ?,
                ?,
                usr,
                seq,
                country,
                rating,
                sigsecsui
from ttmpout a
where a.seq = ?
  and 0 = (select count(*)
           from ttmpout b
           where b.suistrcod = ?
           and b.cempdtcod = ?
           and b.bkgareacod = ?
           and b.risktyp = ?
           and b.riskflg = ?
           and b.seq = ?)
```

此例并非暗示我们无条件地认可临时表的使用。另外，我怀疑这个 insert 语句会被循环执行，通过消除循环可以适当改善性能。

例子中出现了自参照（self-reference）很不常见的用法：对一个表的插入操作，是以同一个表上的 select 为基础的。当前存在哪些记录？要创建的记录是否不存在？要插入的记录是由上述两个问题决定的。

使用 count(*) 测试某些数据是否存在是个糟糕的主意：为此 DBMS 必须搜索并找出所有相符的记录。其实，此时应该使用 exists，它会在遇到第一个相符数据时就停止。当然，如果过滤条件是主键，使用 count 或 exists 的差别不大，否则差异极大——无论如何，从语义角度讲，若想表达：

```
and not exists (select 1 ...)
```

不能换成：

```
and 0 = (select count(*) ...)
```

使用 count(*)时，优化器“可能”会进行合理的优化——但未必一定如此。记录的数量若通过独立步骤被计入某个变量，优化器肯定不会优化，因为优化器再聪明也无法猜测计数的用途：count()的结果可能是极重要的值，而且必须显示给最终用户！

然而，当我们只想建立一条新记录，且新记录要从已存在于表中的记录推导出来时，正确的做法是使用诸如 except（有时称为 minus）这样的集合操作符（set operator）。

```
insert into ttmpout(custcode,
                    suistrcod,
                    cempdtcod,
                    bkgareacod,
                    mgtareacod,
                    risktyp,
                    riskflg,
                    usr,
                    seq,
                    country,
                    rating,
                    sigsecsui)
(select custcode,
        ?,
        ?,
        ?,
        mgtareacod,
        ?,
        ?,
        usr,
        seq,
```

```
          country,
        rating,
        sigsecsui
from ttmpout
where seq = ?
except
select custcode,
        ?,
        ?,
        ?,
        mgtareacod,
        ?,
        ?,
        usr,
        seq,
        country,
        rating,
        sigsecsui
from ttmpout
where suistrcod = ?
  and cempdtcod = ?
  and bkgareacod = ?
  and risktyp = ?
  and riskflg = ?
  and seq = ?)
```

集合操作符的重大优点是彻底打破了“子查询强加的时间限制”，无论子查询是关联子查询还是非关联子查询。打破“时间限制”是什么意思？当存在关联子查询时，就必须执行外层查询，接着对所有通过过滤条件的记录，执行内层查询。外层查询和内层查询相互依赖，因为外层查询会把数据传递给内层查询。

使用非关联子查询时情况要好得多，但也不是完全乐观：必须先完成内层查询之后，外层查询才能介入。即使优化器选择把整个查询作为哈希连接（hash join）执行——这是聪明的方法——也不例外，因为要进行哈希连接，SQL 引擎必须先进行表扫描以建立哈希数组（hash array）。

相比之下，使用集合操作符 `union`、`intersect` 或 `except` 时，查询中的这些组成部分不会彼此依赖，从而不同部分的查询可以并行执行。当然，如果有个步骤非常慢，而其他步骤非常快，则并行化意义不大；另外，如果查询的两个部分工作完全相同，并行化就没有好处，因为不同进程的工作是重复的，而不是分工负责。一般而言，在最后步骤之前，让所有部分并行执行会很高效，最后步骤把不完整的结果集组合起来——这就是分而治之。

集合操作符的使用有个额外的问题：各部分查询必须返回兼容的字段 —— 字段的类型和数量都要相同。下例（实际案例，来自账单程序）通常不适合集合操作符：

```
select whatever, sum(d.tax)
from invoice_detail d,
     invoice_extractor e
where (e.pga_status = 0
       or e.rd_status = 0)
 and suitable_join_condition
 and (d.type_code in (3, 7, 2)
      or (d.type_code = 4
         and d.subtype_code not in
             (select trans_code
              from trans_description
              where trans_category in (6, 7))))
group by what_is_required
having sum(d.tax) != 0
```

最后一个条件有问题（它使我想起了《绿野仙踪》里的黄砖路，甚至使我做起了"负税率"的白日梦）：

```
sum(d.tax) != 0
```

如前所述，换成下列条件更加合理：

```
and d.tax > 0
```

上述的例子中，使用集合操作符会相当笨拙，因为必须访问 invoice_detail 表好几次——如你所料，那不是个轻量级的表。当然，还要看每个条件的可选择性，如果 type_code=4 很少见，那么它就是个可选择性很高的条件，exists 或许会比 not in () 更适合。另外，如果 trans_description 正好是个小型表（或者相对较小），尝试通过单独操作测试存在性，并起不到改善性能的效果。

另一个表达非存在性的方法很有趣——而且通常相当高效——是使用外连接（outer join）。外连接的主要目的是，返回来自一个表的所有信息及连接表中的对应信息。无对应信息的记录也需返回——查找另一个表中无对应信息的数据时，这些记录正好是我们的兴趣所在，可通过检查连接表的字段值是否为 null 找出它们。

例如：

```
select whatever
from invoice_detail
where type_code = 4
```

```
and subtype_code not in
          (select trans_code
           from trans_description
           where trans_category in (6, 7))
```

或重写为：

```
select whatever
from invoice_detail
    outer join trans_description
          on trans_description.trans_category in (6, 7)
         and trans_description.trans_code = invoice_detail.subtype_code
where trans_description.trans_code is null
```

我故意在 join 子句中加上 trans_category 的条件。有人认为它应该出现在 where 子句中，实际上，在连接之前或在连接之后过滤都不影响结果（当然，根据这个条件和连接条件本身的可选择性不同，会有不同的性能表现）。然而，在使用空值上的条件时，我们别无选择，只有在连接后才能做检查。

外连接有时需要加 distinct。实际上，通过外连接或 not in()非关联子查询，来检查数据是否存在的差异很小，因为连接所使用的字段，正好与比较子查询结果集的字段完全相同。不过，众所周知的是，SQL 语言的“查询表达式风格”对“执行模式”影响很大，尽管理论上不是这么说的。这取决于优化器的复杂程度，以及它是否会以类似方法处理这两类查询。换言之，SQL 不是真正的声明性语言（SQL is not a truly declarative language），尽管优化器不断推陈出新改善 SQL 的可靠性（reliability）。

最后提醒一下，应密切注意 null，这个舞会扫兴者（party-poopers）经常出现。虽然在 in ()子查询中，null 与大量非空值连接不会对外层查询造成影响，但在使用 not in () 子查询时，由内层查询返回的 null 会造成 not in()条件不成立。要确保子查询不会返回 null 并不需要太高的代价，而且这么做可以避免许多灾难。

总结：数据集可以通过各种技巧进行比较，但一般而言，使用外连接和子集合操作符更高效。

第 7 章

变换战术：处理层次结构

Variations in Tactics: Dealing with Hierarchical Data

没有什么金科玉律，就是金科玉律。

——乔治 · 萧伯纳（1856—1950）

上一章讨论了查询参照同一个表多次，或者结果通过连接同一个表中的不同记录获得等情况。还有一种非常重要的情况：A 记录依赖于 B 记录，B 记录又依赖于 C 记录……依次类推。这就是层次（hierarchy）结构。

树状结构

Tree Structures

在结构化数据的存储方面，关系理论给层次数据库（hierarchical database）造成了决定性的打击。在历史上，层次数据库最早出现，将数据保存为文件中的记录（record），各种记录在逻辑上被嵌套在一起，而没有将同样的记录按线性排列。层次数据库对某些查询非常适合，但过强的结构限制了人们对数据的自由操控。后来出现了网络（或 CODASYL）数据库，虽然灵活多了，但数据操控仍较困难。直到关系理论出现，才证明数据库设计是“科学（science）”不是“工艺（craft）”。然而，因为层次模型很有弹性，所以层次结构依然极为常见（至少层次式描述非常常见），例如 XML 和 LDAP（Lightweight Directory Access Protocol，轻量级目录访问协议）等层次技术非常活跃。

层次式数据——例如广为人知的材料单（Bill of Materials，BOM）问题——不太容易理解。层次结构之所以复杂，主要原因不是“组件之间关系的表达”，而是“访问树的方式”。我们访问树的全部或部分节点，通常按特定顺序返回这些节点。访问树通常由 DBMS 引擎以过程性（procedural）方式实现，而过程性操作正是违背关系理论的主要表现之一。

树状结构 vs. 主从关系

Tree Structures Versus Master/Detail Relationship

大量设计者认为“父／子关系（parent/child link）”与“主／从关系（master/detail relationship）”没有太大不同——典型的 `orders/order_detail` 关系中，`order_detail` 表保存（作为键的一部分）相关订单的参照数据。然而，描述“父／子关系”的树状结构至少在下列四个方面与主／从关系不同。

树状结构的保存只需一个表

第一个差异是，代表层次结构的树，其所有节点类型完全相同。叶节点（没有子节点的节点）的类型有时可能不同——例如文件系统的文件夹（普通节点）和文

件（叶节点），我们暂不考虑这种情况。既然所有节点类型完全相同，我们可以用相同方法描述，而且用同一个表来代表节点。换句话说，表与它本身之间有种主／从关系，而不是两个类型不同的表的关系。

深度

第二个差异是，层次结构中，与根节点的距离本身是重要信息。而在主／从关系中，不是主表、就是明细表。

所有权

第三个差异是，主／从关系中，可以有明确的外键完整性约束，例如，每个 `order_detail` 表中的订单 ID，必须与 `orders` 表中已存在的 ID 对应。但层次结构不同，例如，虽然经理标志号必会参照已存在的员工标志号，但总经理向董事会汇报，他不是向员工报告。这给我们造成了困难，也就是 NULL 的问题。可能会有多个“特例”记录，因而需要在相同表中描述多个独立的树，每个树都有自己的根——这种情况称为森林（forest）。

多重父节点

以“父节点”的识别数据结合“子节点”，是假设子节点只会有一个父节点。实际上，无论是投资、药方组成、机械零件的螺丝，许多现实生活中的情况并非如此。以数学的观念来看，当子节点有多个父节点时，那并不是树；很不幸的，许多现实生活中的树（包含家谱的树）比简单的父／子关系更复杂，而且甚至会需要处理特例（那不在本书的范围内），例如一连串关系中的循环。

Fabian Pascal 在他的杰出著作 *Practical Issues in Database Management*（Addison Wesley）中，阐述了如何从关系理论的角度理解树结构：数有两种实体类型（entity type），一个是节点（可能包含信息量更大的叶节点的子类型），另一个是节点之间的连结（link）。必须指出，这样的设计解决了完整性约束的问题，因为只有实际存在的连结才能被描述。Fabian Pascal 的方法也支持一个“子节点”是多个“父节点”的后代的情况，这种情况在业界很常见，但教科书中涉及不多，教科书常使用员工／经理的例子。

Fabian Pascal 延续了 Chris Date 的思想，主张应该有个 `explode()` 操作符用于将层次结构“扁平化（flatten）”，即把节点间的隐式连结显式化。但问题是 `explode()` 操作符本

应由厂商实现，但他们没有这样做，最终不得不由开发者实现：DBMS 厂商常常实现空间数据处理或全文索引等特殊功能，但对层次结构的支持很薄弱或根本没有。

我已提到，处理层次结构的主要困难在于树的访问。当然，仅仅为了在图形用户界面中显示树状结构，每次用户点击时将节点展开，并不会有什么问题：以节点 ID 为参数，查询该节点的所有子节点，是直截了当的方法。

层次结构的实际案例
Practical Examples of Hierarchies

在现实生活中常会遇到层次数据，它们支持的应用一般都比较复杂。下面三个层次结构的例子来自不同行业。

风险暴露（或风险敞口，Risk exposure）

> 要计算金融机构（financial structure）面临的风险，如对冲基金(Hedge Fund)，就呈现出层次式的复杂性。这些金融机构会投资基金，而基金本身可能又投资了其他基金。

档案位置

> 商业零售银行（retail bank）的经营，需要从档案中找到 John 七年前签名的借款文件可不是件容易事，因为档案放在档案夹中，档案夹在箱子中，箱子在架子上，架子在某栋建筑的某个楼层的某个房间的某排柜子里……层层“容器”（档案夹、箱子、架子等）形成了层次结构。

原料使用

> 以制药行业为例，已证实某种原料有了便宜的替代品，如何找到所有包含此原料的药品？这种情况说明 SQL 要面临非关系领域的难题。

理解上述三个层次结构案例有着不同的基本特征非常重要。例如，确定文件在档案中的位置，意味着从底端到顶端访问树（即位置的综合程度越来越高），因为你从单个文件开始找到保持该文件的档案夹，进而从档案夹找到特定箱子，再确定所在建筑物的房间……要确定包含特定原料的所有产品，同样也是自底向上的访问，只不过我们的起始点数量较多，而且每次都重复访问动作。相反，风险暴露分析则是自顶向下访问，以找出所有的投资，并逐步计算至顶端；这是种聚合操作，只是复杂得多罢了。

通常，树中的叶节点数量偏少。实际上，这是树的主要优点，利于高效搜索。如果叶节点的数量固定，我们只需把表自连接多次，连接的次数与层数相同。以档案位置为例：假设 inventory 表保持贷款文件所在的档案夹，档案夹的标志数据指向 location 表，比表保持档案夹所在的箱子、箱子所在的架子、架子所属的柜子、柜子所在的通道、通道所在的房间，以及房间所在的建筑等信息。如果 location 表把档案夹、箱子、架子等当成一般“位置”，则返回文件物理位置的查询将如：

```
select building.name building,
       floor.name floor,
       room.name room,
       alley.name alley,
       cabinet.name cabinet,
       shelf.name shelf,
       box.name box,
       folder.name folder
from inventory,
     location folder,
     location box,
     location shelf,
     location cabinet,
     location alley,
     location room,
     location floor,
     location building
where inventory.id = 'AZE087564609'
  and inventory.folder = folder.id
  and folder.located_in = box.id
  and box.located_in = shelf.id
  and shelf.located_in = cabinet.id
  and cabinet.located_in = alley.id
  and alley.located_in = room.id
  and room.located_in = floor.id
  and floor.located_in = building.id
```

虽然连接的次数很多，但这类查询执行速度很快，因为后面的每个连接都使用 location 上的唯一性索引（为 id 字段建立的索引，id 可能就是主键）。不过，也有问题，即层次结构的层数很少会是常数。例如档案管理工作虽然比较平稳，但有时档案的位置也会调整，从这个容器移到那个容器，甚至取消了“架子”而直接用特殊的箱子，这样层次结构的层数就发生了变化。当我们不知道层数时，该怎么查询呢？最终查询的性能会怎样？要使用 union 吗？要使用外连接（outer-join）吗？

总结：只要对象的类型相同、而对象之间的层数可变，其关系就应该被建模为树结构。

用 SQL 数据库描述树结构

Representing Trees in an SQL Database

在 SQL 的世界中，树通常可用三种模型描述。

邻接模型（Adjacency model）

称为邻接模型(adjacency model)的原因在于：层次中的父记录 ID 作为子记录(child row）的属性，树中两个相邻的节点因而被明确关联在一起。经理的员工编号被指定为被管理员工的属性，就是邻接模型常见的例子。（经理与员工的直接关联，实际上是不良的设计，因为经理的识别数据，应该是他所管理的结构（例如部门）的属性。当更换部门经理时，不应该更新所有部门员工的记录。）为了处理这类模型，有些产品实现了特殊的操作符，例如 Oracle 的 `connect by`（大约 20 世纪 80 年代中期的 Oracle 版本 4 引入的），再如 DB2 和 SQL Server 采用的较新的“递归的 `with` 语句”。如果没有这种操作符，邻接模型是非常难处理的。

物化路径模型（Materialized path model）

此模型的思想是，把树中的每个节点、与它在树中位置的描述数据结合。此描述数据所采用的形式，是此节点所有祖先的识别数据的串接列表（从树根往下直到与它的父节点），或是指出在同代兄弟节点中的排名（家谱学者常使用这个方法）。列表通常以分隔字符串的形式保存。例如，'1.2.3.2' 代表此节点是它父节点（其路径为 '1.2.3'）的第 2 个子节点，父节点是祖父节点（'1.2'）的第 3 个子节点，依次类推。

嵌套集合模型（Nested set model）

此模型由 Joe Celko（注 1）发明，每个节点被赋予一对数字（即 left number 和 right number），最终，父节点的两个数字定义的间隔总是将其所有子孙所定义的间隔包含在其中。我们稍后在“树的实际实现”中的“嵌套集合模型（来自 Celko）”一节中讨论，并附实例。

注 1：在 *DBMS Magazine* 杂志的一篇文章中首次提出，大约是 1996 年，后在 *Trees and Hierarchies in SQL for Smarties* (Morgan-Kauffman 出版社)一书中论述。

其实，还存在第四种模型，名为“嵌套间隔模型（nested interval model）”（注 2），名气不大，提出者为 Vadim Tropashko。简要而言，该模型的思想是以两个数字为特定节点的路径编码，这两个数字被解释成有理数（即分数）的分子和分母。很不幸，这个模型的计算量大、对存储程序要求高，虽然未来可能在 DBMS 的树操作函数（如 `explode()` 操作符）上很有前途，但现在还不易实现，而且也不是最快的方式，所以我们重点关注前面提到的三个模型。

为了与本书的主题保持一致，并产生合理的数据量，我为 1815 年滑铁卢（注 3）战役的参战军队建立了测试数据库，描述了英荷联军（Anglo-Dutch）、普鲁士军队及法国军队的结构，包括军、师、旅、团等。我使用这些数据（多半是各部队描述或指挥官名字）作为本章中许多例子的基础。

必须说明，本章后续的重点是说明访问层次结构的各种方法，至于表的设计，未必十分严谨。通常，部队的主键应该是可理解和标准化的代码，而不可能是文字描述，否则数据输入常会出错，毕竟，代理键实际上是对隐式的真正主键的快捷方式。

层次结构的主要困难是不存在“最佳描述（best representation）”。当我们只对少数元素的祖先感兴趣时（自底向上访问），无论使用 `connect by` 或递归 with，在功能和性能上还能令人满意。然而，深入分析发现 `connect by` 这种不太优雅的实现方式是非关系的、过程性的，从某种意义来说只能逐条记录向下移动；当数据量大时，用 `connect by` 的方式返回自底向上（bottom-up）的层次结构，或者通过自顶向下（top-down）的遍历返回大量后代时，效果很差。和使用 SQL 时经常遇到的情况一样，处理 14 条记录的表时问题可能不会暴露，但处理数百万条记录（更别说是数十亿条了）时问题就很明显了，原本的 SQL 技巧现在成了性能瓶颈。

注 2：最初发表在 *http://www.dbazine.com*。

注 3：数据来源 *http://www.kessler-web.co.uk*，由 Peter Kessler 编译，使用得到其授权。

我们的例子表包含八百多行，比一般例子稍大些，但比业界常见的情况要小。不过，这已经足以表现各种模型的优缺点了。

总结：树结构的 SQL 实现依赖于 DBMS，即要借助 DBMS 提供的手段才能实现。

树的实际实现

Practical Implementation of Trees

下面讨论上述三个层次模型的实例。在每个例子中，记录会以相同顺序（依 commander 排序）插入表，以对比记录的物理顺序与预期结果。注意，此设计存在问题，我们只是以尽可能简单的方法来说明如何处理树。

邻接模型

Adjacency Model

下表描述了军队的层次结构，使用邻接模型。表名为 ADJACENCY_MODEL，表的每一行描述一个部队，parent_id 指向树中的上级部队：

```
Name                              Null?    Type
--------------------------------  -------- --------------
ID                                NOT NULL NUMBER
PARENT_ID                                  NUMBER
DESCRIPTION                       NOT NULL VARCHAR2(120)
COMMANDER                                  VARCHAR2(120)
```

ADJACENCY_MODEL 表有三个索引：在 id （主键）上的唯一性索引、在 parent_id 上的索引、在 commander 上的索引。假设 ADJACENCY_MODEL 表中有如下记录：

```
ID  PARENT_ID DESCRIPTION                    COMMANDER
--- --------- ----------------------------   ----------------------------
435         0 French Armée du Nord of 1815   Emperor Napoleon Bonaparte
619       435 III Corps                      Général de Division Dominique
                                             Vandamme
620       619 8th Infantry Division          Général de Division Baron
                                             Etienne-Nicolas Lefol
621       620 1st Brigade                    Général de Brigade Billard
                                             (d.15th)
622       621 15th Rgmt Léger                Colonel Brice
623       621 23rd Rgmt de Ligne             Colonel Baron Vernier
```

```
624       620 2nd Brigade                 Général de Brigade Baron
                                          Corsin
625       624 37th Rgmt de Ligne          Colonel Cornebise
626       620 Division Artillery
627       626 7/6th Foot Artillery        Captain Chauveau
```

物化路径模型

Materialized Path Model

MATERIALIZED_PATH_MODEL 表保存了与 ADJACENCY_MODEL 相同的层次，但描述方式不同。原先的两个字段(id, parent_id)被单一的 materialized_path 字段取代，其中记录了当前记录的完整“祖先（ancestry)”：

```
Name                                 Null?    Type
------------------------------------ -------- ----------------
MATERIALIZED_PATH                    NOT NULL VARCHAR2(25)
DESCRIPTION                          NOT NULL VARCHAR2(120)
COMMANDER                                     VARCHAR2(120)
```

MATERIALIZED_PATH_MODEL 表有两个索引，在 materialized_path 上的唯一性索引（主键）及在 commander 上的索引。在实际中以路径为主键是很糟糕的设计，因为人或物在层次结构中的位置很少是不变的。正确的设计应该有某种 id，就像 ADJACENCY_MODEL 表中的一样；上面的 schema 中没出现 id 字段，因为我们的例子没有涉及该字段。

然而，MATERIALIZED_PATH_MODEL 表为何以 materialized_path 作为主键呢？第 5 章讨论了数据的特殊物理结构的意义，我们以 materialized_path 作为主键正是出于这种考虑。本例中，“描述树结构的 table”恰恰反映了“索引的树结构”，再做额外的映射（mapping）就没有意义了。

下面是表的内容示例，记录值与邻接模型中的相同，但采用物化路径模型：

```
MATERIALIZED_PATH  DESCRIPTION                  COMMANDER
-----------------  ---------------------------  --------------------------
F                  French Armée du Nord of 1815 Emperor Napoleon Bonaparte
F.3                III Corps                    Général de Division
                                                Dominique Vandamme
F.3.1              8th Infantry Division        Général de Division Baron
                                                Etienne-Nicolas Lefol
F.3.1.1            1st Brigade                  Général de Brigade Billard
                                                (d.15th)
F.3.1.1.1          15th Rgmt Léger              Colonel Brice
F.3.1.1.2          23rd Rgmt de Ligne           Colonel Baron Vernier
F.3.1.2            2nd Brigade                  Général de Brigade Baron
                                                Corsin
F.3.1.2.1          37th Rgmt de Ligne           Colonel Cornebise
F.3.1.3            Division Artillery
F.3.1.3.1          7/6th Foot Artillery         Captain Chauveau
```

嵌套集合模型（来自 Celko）

Nested Sets Model(After Celko)

使用嵌套集合模型时，会涉及 left_num 和 right_num 两个字段，它们描述每条记录与层次中其他记录的关系。下面简要说明如何用这两个数字来表示在层次中所处的位置：

```
Name                                   Null      Type
-------------------------------------- --------  -------------
DESCRIPTION                            VARCHAR2(120)
COMMANDER                                        VARCHAR2(120)
LEFT_NUM                               NOT NULL  NUMBER
RIGHT_NUM                              NOT NULL  NUMBER
```

NESTED_SETS_MODEL 表有个复合主键(left_num, right_num)，同时，在 commander 字段 上有索引。这种主键设计方式，和上一节“物化路径模型”中的主键设计一样，都可进一步改进，不过对于我们测试不同模型的表现已足够了。

下面说明如何获得神秘数值 left_num 和 right_num 。从树根开始，把 1 指定给根节点的 left_num。接着，所有的子节点会被递归地访问，如图 7-1 所示，而计数器会在每次调用时递增。图中显示了计数器增长的路线，根节点为 1，下一个被访问的节点加 1。

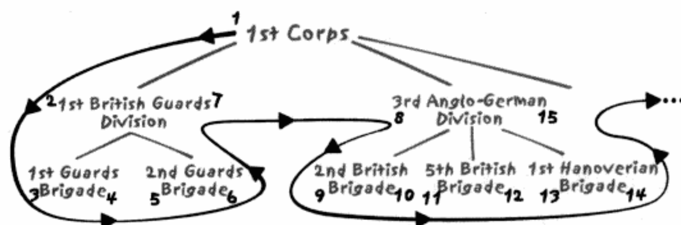


图 7-1：嵌套集合是如何编号的

假设我们第一次访问某个节点，例如图 7-1 所示把整数 1 赋给“1st Corps（第一军）”节点的 left_num 值之后，我们会遇到 “1st British Guards Division（第一英军护卫师）”节点。增加计数器的值，把 2 赋给 left_num。接着，访问此节点的子节点，第一次遇到 “1st Guards Brigade（第一护卫旅）”，并把计数器的值（此时是 3）赋给 left_num。这个节点没有子节点，所以递增计数器后将 4 赋给 right_num。接着，移到这个节点的兄弟节点“2nd Guards Brigade（第二护卫旅）”，处理这个兄弟节点的方式相同。最后，回到父节点 “1st British Guards Division（第一英军护卫师）”并把新的计数器值（现在已为 7）赋给它的 right_num。接下来，处理“3rd Anglo-German Division（第三英德师）”等其他兄弟节点，依次类推。

如前所述，节点的 [left_num, right_num] 值必定介于其任一个祖先的 [left_num, right_num] 值之间，这就是“嵌套集合（nested set）”的由来。然而，由于我们有三

个独立的树（英荷、普鲁士和法国军队），术语称之为森林（forest），所以必须建立名为“Armies of 1815（1815 年的军队）”的人造顶端层次。在其他的模型中，不需要这类人造的根元素。

在计算过所有的数值之后，最终得到的结果如下：

```
DESCRIPTION                    COMMANDER                    LEFT_NUM  RIGHT_NUM
----------------------------   --------------------------   --------  ----------
Armies of 1815                                                  1       1622
French Armée du Nord of 1815   Emperor Napoleon Bonaparte     870       1621
III Corps                      Général de Division           1237       1316
                               Dominique Vandamme
8th Infantry Division          Général de Division Baron     1238       1253
                               Etienne-Nicolas Lefol
1st Brigade                    Général de Brigade Billard    1239       1244
                               (d.15th)
15th Rgmt Léger                Colonel Brice                 1240       1241
23rd Rgmt de Ligne             Colonel Baron Vernier         1242       1243
2nd Brigade                    Général de Brigade Baron      1245       1248
                               Corsin
37th Rgmt de Ligne             Colonel Cornebise             1246       1247
Division Artillery                                           1249       1252
7/6th Foot Artillery           Captain Chauveau              1250       1251
```

叶子元素是那些 `right_num` 等于 `left_num + 1` 的元素。

原创者声称这个聪明的方法优于邻接模型，理由是：它基于集合进行操作，而 SQL 完全就是关于集合的。这个理由倒是“完全正确（perfectly true）”，只不过 SQL 关心无限集合，而嵌套集合模型关心有限集合，因为只有知道所有节点的数量之后才能为根节点的 `right_num` 赋值。显然，每次插入新节点时，必须重编其后访问的所有节点的 `left_num` 和 `right_num`，也需重编该节点所有祖先的 `right_num` 值。插入新项目却要修改许多其他项目，就像是把新项目插入到有序数组一样，平均要移动一半数组元素。不容置疑，嵌套集合模型很有想象力，但对关系模型却是梦魇，很难想象在非规范化的情况下会糟到什么程度。实际上，嵌套集合模型是基于指针的解决方案，而设计关系方法的目标正是要逃离指针的沼泽。

用 SQL 访问树结构

Walking a Tree with SQL

为了检查效率和性能，分别用不同模型解决如下两个问题。

1. 法国将军 Dominique Vandamme 指挥哪些部队（自顶向下的查询），以缩排方式（必须记录树中的深度）或简单列表的方式显示它们。注意，所有例子中，指挥官名字字段上都建立了索引。我们把这个问题称为“Vandamme 查询”。

2. 苏格兰高地联队（Scottish Highlanders）的每个团各属于哪个部队（自底向上的查询），以适当方式显示。在部队的名称（description 字段）上没有索引，而找出苏格兰高地联队的唯一方法就是在部队名称中查找 “Highland” 字符串，在没有任何全文索引的情况下，这当然意味着完整扫描。我们把这个问题称为“高地联队查询”。

为了确保各测试只有模型不同，我们使用相同的 DBMS，即 Oracle。

自顶向下访问：Vandamme 查询

Top-Down Walk: The Vandamme Query

Vandamme 查询从 French Third Corps（法国第三军）的指挥官 Vandamme 将军开始，有秩序地显示他指挥的所有部队。我们没有采用简单列表，因为军队的结构必须清楚，军（corp）是由师组成，师（division）由旅组成，旅（brigade）通常是由两个团（regiment）组成。

邻接模型

当使用 Oracle 的 connect by 操作符时，以邻接模型编写 Vandamme 查询是相当容易的。只要指定开始的节点及两个连续的记录如何返回彼此的关系（“connect by < 当前记录的字段 > = prior < 前一个记录的字段 >”，“或 connect by < 前一个记录的字段 > = prior < 当前记录的字段 >”，这取决于向下访问树还是向上访问树）。至于缩排，Oracle 维护了名称为 level 的虚拟字段，它会告知离起点有多少层。我使用了这个虚拟字段（pseudo-column），在显示结果左边填补“与当前层次值一样多的空格”。查询如下：

```
select lpad(description, length(description) + level) description,
       commander
from adjacency_model
connect by parent_id = prior id
start with commander = 'Général de Division Dominique Vandamme'
```

结果如下：

```
DESCRIPTION                    COMMANDER
------------------------------ ---------------------------------------------
 III Corps                     Général de Division Dominique Vandamme
  8th Infantry Division        Général de Division Baron Etienne-Nicolas Lefol
   2nd Brigade                 Général de Brigade Baron Corsin
    37th Rgmt de Ligne         Colonel Cornebise
   1st Brigade                 Général de Brigade Billard (d.15th)
    23rd Rgmt de Ligne         Colonel Baron Vernier
    15th Rgmt Léger            Colonel Brice
      ...
```

```
  10th Infantry Division        Général de Division Baron Pierre-Joseph Habert
   2nd Brigade                  Général de Brigade Baron Dupeyroux
    70th Rgmt de Ligne          Colonel Baron Maury
    22nd Rgmt de Ligne          Colonel Fantin des Odoards
    2nd (Swiss) Infantry Rgmt   Colonel Stoffel
   1st Brigade                  Général de Brigade Baron Gengoult
    88th Rgmt de Ligne          Colonel Baillon
    34th Rgmt de Ligne          Colonel Mouton
   Division Artillery
    18/2nd Foot Artillery       Captain Guérin

40 rows selected.
```

那么，其他邻接成员呢，用递归 with 语句（注 4）吗？递归分解语句（recursive-factorized statement）可支持邻接模型，它对两个 select 语句进行了 union 操作（精确地说是 union all）：

- 定义了我们起点的 select，它在这个案例中应这样写：

  ```
  select 1 level,
         id,
         description,
         commander
  from adjacency_model
  where commander = 'Général de Division Dominique Vandamme'
  ```

  这里单独出现的 1 有什么作用呢？就如别名所指出的，它代表树中的深度。与 Oracle 的 connect by 实现相比，这个 DB2 实现没有系统虚拟变量能告知树中位置。然而，我们可以计算层次，随后尚有更多说明。

- 另一个 select 定义子记录与父记录的关系。子记录和父记录其实是同一个查询返回的，我们将该查询命名为“recursive_query（递归查询）”：

  ```
  select parent.level + 1,
         child.id,
         child.description,
         child.comander
  from recursive_query parent,
       adjacency_model child
  where parent.id = child.parent_id
  ```

  注意，我们在查询中对 parent.level 加 1。每执行一次这个查询代表向树的下方前进一步，于是 level 的值增加 1，用来跟踪访问深度。

前述步骤只是为了用函数巧妙地缩排结果，下面是我们最终的查询：

```
with recursive_query(level, id, description, commander)
as (select 1 level,
           id,
           description,
```

注 4：由 DB2 最早实现。

```
        commander
    from adjacency_model
    where commander = 'Général de Division Dominique Vandamme'
    union all
    select parent.level + 1,
           child.id,
           child.description,
           child.commander
    from recursive_query parent,
         adjacency_model child
    where parent.id = child.parent_id)
select char(concat(repeat(' ', level), description), 60) description,
       commander
from recursive_query
```

只要不是递归 `with` 的狂热爱好者，就会认为这里的语法不太自然易懂。然而，一旦写好之后，要理解并不困难；而且它相当令人满意，先返回 Vandamme 将军，接着返回所有直接向 Vandamme 报告的军官，然后依次返回每个军官的下属……依次类推。此时，查询结果还不像 `connect by` 那样精确显示谁向谁报告。需立即说明的是，排序不属于关系理论，使用排序虽没有错，但的确引起了严重的问题。实践当中，我们应如何对层次模型查询的结果排序呢？

使用递归 `with` 对层次模型查询的结果排序是可能的。根据合理的假设，一个父节点的子节点数绝不会超过 99 个，而且树也不会太深。所以，可以把每个节点与指出节点层次位置的数字结合——例如 “1.030801” 代表 “根节点的第三个子节点（.03）的第八个子节点（08）的第一个子节点（01）”。当然，这里假设兄弟节点是有序的，但实际上未必如此。使用诸如 `row_number()` 的 OLAP 函数，可以先对每个兄弟节点指定顺序。

于是，对先前查询稍作修改，任意指定兄弟节点的顺序，并使用刚才说明的技巧对结果记录排序：

```
with recursive_query(level, id, rank, description, commander)
as (select 1,
           id,
           cast(1 as double),
           description,
           commander
    from adjacency_model
    where commander = 'Général de Division Dominique Vandamme'
    union all
```

```
    select parent.level + 1,
           child.id,
           parent.rank + ranking.sn / power(100.0, parent.level),
           child.description,
           child.commander
    from recursive_query parent,
         (select id,
                 row_number( ) over (partition by parent_id
                                     order by description) sn
          from adjacency_model) ranking,
         adjacency_model child
    where parent.id =child.parent_id
      and child.id = ranking.id)
select char(concat(repeat(' ', level), description), 60) description,
       commander
from recursive_query
order by rank
```

我们可能担心，排名查询作为整个查询的递归部分出现，会对树中每个节点执行且每次返回相同结果。事实并非如此。很幸运，优化器足够聪明，不会执行不必要的排名查询。最终查询结果如下：

```
DESCRIPTION                      COMMANDER
------------------------------   ------------------------------------------------
 III Corps                       Général de Division Dominique Vandamme
  10th Infantry Division         Général de Division Baron Pierre-Joseph Habert
   1st Brigade                   Général de Brigade Baron Gengoult
    34th Rgmt de Ligne           Colonel Mouton
    88th Rgmt de Ligne           Colonel Baillon
   2nd Brigade                   Général de Brigade Baron Dupeyroux
    22nd Rgmt de Ligne           Colonel Fantin des Odoards
    2nd (Swiss) Infantry Rgmt    Colonel Stoffel
    70th Rgmt de Ligne           Colonel Baron Maury
   Division Artillery
    18/2nd Foot Artillery        Captain Guérin
  11th Infantry Division         Général de Division Baron Pierre Berthézène
   ...
    23rd Rgmt de Ligne           Colonel Baron Vernier
   2nd Brigade                   Général de Brigade Baron Corsin
    37th Rgmt de Ligne           Colonel Cornebise
   Division Artillery
    7/6th Foot Artillery         Captain Chauveau
  Reserve Artillery              Général de Division Baron Jérôme Doguereau
   1/2nd Foot Artillery          Captain Vollée
   2/2nd Rgmt du Génie
```

上述结果与 connect by 查询返回的结果不完全相同，这是因为在 description 字段上按照字母序对兄弟节点进行了排序，但使用 connect by 时没有排序（我们也可以通过增加特殊的子句对它们排序）。但除此之外，所显示的层次完全相同。

虽然查询结果等价，但 with 查询非常复杂，而 connect by 仅有非常精致的 5 行语句。本例性能尚可接受，但递归 with 在数据量极大的表上表现如何呢？我们必须使用 connect by 吗？递归 with 只是 connect by 的糟糕的、不合标准的替代实现吗？结论在本章结尾给出。

我们在递归查询中建立的排名编号，只不过是物化路径的数值表达方式。所以，接下来可以使用简单的物化路径实现“Vandamme 查询”。

物化路径模型

使用物化路径模型时，编写查询不会太困难，但计算由路径导出的层次时不太方便。举例来说，假设我们手边有个名称为 mp_depth() 的函数，它会返回当前节点的深度：

```
select lpad(a.description, length(a.description)
       + mp_depth(...)) description,
      a.commander
from materialized_path_model a,
    materialized_path_model b
where a.materialized_path like b.materialized_path || '%'
  and b.commander = 'Général de Division Dominique Vandamme')
order by a.materialized_path
```

在解释 mp_depth() 函数之前，先说明其中几个技巧。首先，我选择从代表英荷联军的 A、代表普鲁士军队的 P 和代表法国军队的 F 开始物化路径。第一个字母后面会接着以点号分隔的数字。因此，Bagelaar 上校指挥的第 12 荷兰步兵营是 “A.1.4.2.3”，而 Courtier 上校（Colonel Courtier）指挥的第 11 团是 “F.9.1.2.2”。以物化路径排序会导致数字字符串常见的排序问题，即“10.2” 会出现在 “2.3”之前；不过，我要强调，由于可以选用比 0 低的代码作为分隔符（至少在 ASCII 中是这样），所以可以保证层的次序正确。然而，基于物化路径的排序无法保证兄弟节点的顺序正确，这重要吗？我认为不重要，因为兄弟节点的顺序可由物化路径以外的条件导出（例如，兄弟姐妹可以按出生日期排序）。要注意我这里使用的排序方法，数据库所使用的字符集编码可能对此有影响。

下面讨论 mp_depth() 函数。包括 Vandamme 将军在内的所有指挥官处在不同的层次，每个指挥官的“绝对层次”即该节点和树根的距离。我们如何计算绝对层次呢？嗯，计算点号“.”的数量即可。

要计算点号的数量，最简单的方式是：用 replace() 函数（主要产品的 SQL 方言都支持）把它们去掉，接下来用具有点号的字符串长度减去去掉点号的字符串长度即可。

```
length((materialized_path) - length(replace(materialized_path, '.', ''))
```

以第 6 章开始的引用语作者为例（当时是骑兵上校），检查点号计数算法得到：

```
SQL> select materialized_path,
  2         length(materialized_path) len_w_dots,
  3         length(replace(materialized_path, '.', '')) len_wo_dots,
  4         length(materialized_path) -
  5             length(replace(materialized_path, '.', '')) depth,
  6         commander
  7  from materialized_path_model
  8  where commander = 'Colonel de Marbot'
  9  /

MATERIALIZED_PATH LEN_W_DOTS LEN_WO_DOTS      DEPTH COMMANDER
----------------- ---------- ----------- ---------- -----------------
F.1.5.1.1                  9           5          4 Colonel de Marbot
```

Et voilà.

嵌套集合模型

采用嵌套集合模型时，解决“Vandamme 查询”的问题很简单，因为此模型要求对节点编号，而且某节点后代的 left_num 和 right_num 都会在该节点的 left_num 至 right_num 范围内。查询如下：

```
select a.description,
       a.commander
from nested_sets_model a,
     nested_sets_model b
where a.left_num between b.left_num and b.right_num
  and b.commander = 'Général de Division Dominique Vandamme'
```

这就够了吗？不太够，还少了缩排。我们要如何知道在第几层呢？很不幸，取得节点深度（由此进行缩排）的唯一方法，就是计算那个节点和根节点之间的有多少个节点，这是无法从 left_num 和 right_num 导出的（与物化路径模型形成鲜明对比）。

采用嵌套集合模型时要进行缩排显示，就必须第三次连接 nested_sets_model，唯一目的就是计算深度：

```
select lpad(description, length(description) + depth) description,
       commander
from (select count(c.left_num) depth,
             a.description,
             a.commander,
             a.left_num
      from nested_sets_model a,
           nested_sets_model b,
           nested_sets_model c
      where a.left_num between c.left_num and c.right_num
        and c.left_num between b.left_num and b.right_num
        and b.commander = 'Général de Division Dominique Vandamme'
      group by a.description,
               a.commander,
               a.left_num)
order by left_num
```

仅是增加了缩排功能就使查询变得如此难读，这点和使用递归 with() 相似。

比较各种模型下的 Vandamme 查询

上面，讨论了各种查询返回 40 条记录并缩排显示的情况。下面，在循环中执行每个查询 5 000 次（共返回 200 000 条记录）。我比较了每秒返回的记录数，图 7-2 显示了比较结果，其中比例（rate）的显示以邻接模型作为参考值（100 分）。

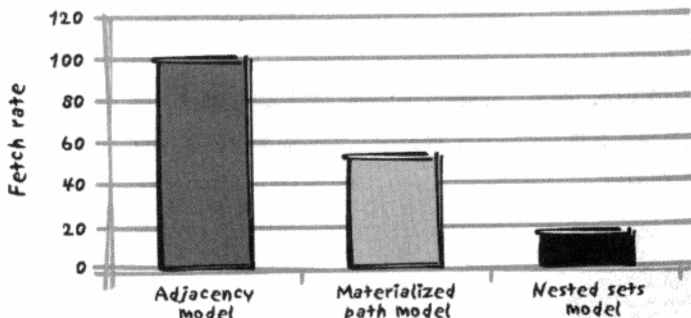


图 7-2：Vandamme 查询的性能比较

如图 7-2 所示，“Vandamme 查询”使用邻接模型时性能最好，虽然 connect by 访问树具有过程性本质。物化路径模型的表现也还不错，可能计算深度和缩排的函数影响了总体表现。嵌套集合模型为缩排显示付出了更大代价，性能降低显然和额外连接，以及

group by 所做的深度计算有关。有人可能不以为然，认为既然嵌套集合模型是“硬连线的（hard-wired)”、静态的且非关系的，那么就应该彻底忽略关系设计原则，直接保存每个节点相对于根节点的深度。这种做法必然会改善查询的性能，但要为维护付出的代价高得可怕。

自底向上访问：高地联队查询
Bottom-Up Walk: The Highlanders Query

如前所述，在 description 属性内查找 “Highland” 字符串，必然导致完整的表扫描。下面，我们依序基于不同模型编写高地联队查询，最后分析它们的性能差异说明了什么。

邻接模型

使用 connect by 很容易实现高地联队查询，这时会再次使用动态计算的 level 虚拟字段来支持缩排显示。注意，前一节的 level 字段代表深度（depth)，而今代表高度（height)，因为前一节的访问顺序是自顶向下，而本节的访问顺序是自底向上：

```
select lpad(description, length(description) + level) description,
      commander
from adjacency_model
connect by id = prior parent_id
start with description like '%Highland%'
```

查询结果如下：

```
DESCRIPTION                          COMMANDER
-----------------------------------  ----------------------------------------
 2/73rd (Highland) Rgmt of Foot      Lt-Colonel William George Harris
  5th British Brigade                Major-General Sir Colin Halkett
   3rd Anglo-German Division         Lt-General Count Charles von Alten
    I Corps                          Prince William of Orange
     The Anglo-Allied Army of 1815   Field Marshal Arthur Wellesley, Duke of
                                     Wellington
 1/71st (Highland) Rgmt of Foot      Lt-Colonel Thomas Reynell
  British Light Brigade              Major-General Frederick Adam
   2nd Anglo-German Division         Lt-General Sir Henry Clinton
    II Corps                         Lieutenant-General Lord Rowland Hill
     The Anglo-Allied Army of 1815   Field Marshal Arthur Wellesley, Duke of
                                     Wellington
 1/79th (Highland) Rgmt of Foot      Lt-Colonel Neil Douglas
  8th British Brigade                Lt-General Sir James Kempt
   5th Anglo-German Division         Lt-General Sir Thomas Picton (d.18th)
    General Reserve                  Duke of Wellington
     The Anglo-Allied Army of 1815   Field Marshal Arthur Wellesley, Duke of
                                     Wellington
 1/42nd (Highland) Rgmt of Foot      Colonel Sir Robert Macara (d.16th)
  9th British Brigade                Major-General Sir Denis Pack
   5th Anglo-German Division         Lt-General Sir Thomas Picton (d.18th)
    General Reserve                  Duke of Wellington
```

```
    The Anglo-Allied Army of 1815   Field Marshal Arthur Wellesley, Duke of
                                    Wellington
1/92nd (Highland) Rgmt of Foot      Lt-Colonel John Cameron
 9th British Brigade                Major-General Sir Denis Pack
  5th Anglo-German Division         Lt-General Sir Thomas Picton (d.18th)
   General Reserve                  Duke of Wellington
    The Anglo-Allied Army of 1815   Field Marshal Arthur Wellesley, Duke of
                                    Wellington

25 rows selected.
```

这充分体现了 connect by 的非关系本质：查询结果并不是关系，因为有重复的项目出现。Wellington 公爵的名字涉及两个不同职位、共出现八次，其中五次是最高统帅（和高地团出现次数一样多），另外三次是预备队指挥官。其实出现两次就足够了——一次是一般预备队的指挥官，一次是最高统帅。我们能轻易消除重复项目吗？不能，至少不是很容易。如果使用 distinct，DBMS 会排序结果，在消除重复记录的同时也打破了层次顺序，你根据具体要求决定是否采用吧。

物化路径模型

采用物化路径模型，要编写高地联队查询稍微困难些。仅找出适当的记录并缩排显示，尚算容易：

```
select lpad(a.description, length(a.description)
                      + mp_depth(b.materialized_path)
                       - mp_depth(a.materialized_path)) description,
       a.commander
from materialized_path_model a,
     materialized_path_model b
where b.materialized_path like a.materialized_path || '%'
  and b.description like '%Highland%')
```

然而，还有两个问题需要解决：

- 重复项目的出现（和邻接模型的情况一样）；
- 记录的顺序不满足我们的要求。

矛盾的是，正是由于没有严格要求顺序问题，第一个问题的解决才如此轻易。我们必须找到一种排序方法，而不受 distinct 的任何影响。我们如何正确排序呢？一如往常，使用物化路径作为排序键。于是，加入这两个元素，并把查询推入 from 子句中，以便能通过 materialized_path 排序，而且不显示这个字段。我们得到：

```
select description, commander
from (select distinct lpad(a.description, length(a.description)
                     + mp_depth(b.materialized_path)
                       - mp_depth(a.materialized_path)) description,
                     a.commander,
                     a.materialized_path
      from materialized_path_model a,
          materialized_path_model b
      where b.materialized_path like a.materialized_path || '%'
        and b.description like '%Highland%')
order by materialized_path desc
```

查询结果如下：

```
DESCRIPTION                          COMMANDER
----------------------------------   ------------------------------------------
1/92nd (Highland) Rgmt of Foot       Lt-Colonel John Cameron
1/42nd (Highland) Rgmt of Foot       Colonel Sir Robert Macara (d.16th)
 9th British Brigade                 Major-General Sir Denis Pack
1/79th (Highland) Rgmt of Foot       Lt-Colonel Neil Douglas
 8th British Brigade                 Lt-General Sir James Kempt
  5th Anglo-German Division          Lt-General Sir Thomas Picton (d.18th)
   General Reserve                   Duke of Wellington
1/71st (Highland) Rgmt of Foot       Lt-Colonel Thomas Reynell
 British Light Brigade               Major-General Frederick Adam
  2nd Anglo-German Division          Lt-General Sir Henry Clinton
   II Corps                          Lieutenant-General Lord Rowland Hill
2/73rd (Highland) Rgmt of Foot       Lt-Colonel William George Harris
 5th British Brigade                 Major-General Sir Colin Halkett
  3rd Anglo-German Division          Lt-General Count Charles von Alten
   I Corps                           Prince William of Orange
    The Anglo-Allied Army of 1815    Field Marshal Arthur Wellesley, Duke of
                                     Wellington

16 rows selected.
```

与邻接模型相比，上述查询结果比较简洁紧凑。然而，我们发现：

```
where b.materialized_path like a.materialized_path || '%'
```

上述查询条件查找表 a 的记录，并在查询表 b 时使用这些记录，一般说来，这种做法速度很慢，因为无法高效地利用索引。我们其实想用已知的 a.materialized_path 查找 b.materialized_path。有一些方法可以把节点的物化路径分解为路径列表（请参阅第 11 章），但这些方法代价较大。对我们例子中的数据而言，当前查询的效果远胜过“分解物化路径，再使用每个祖先节点的物化路径执行高效连接”的做法，当然，数据量达几百行时，情况可能不同。

嵌套集合模型

采用嵌套集合模型时，动态计算深度是个问题，计算深度的操作很“重”。由于高地联队查询是自底向上的查询，所以我们必须小心，不要显示人造根节点(可以用 left_num = 1 轻易地识别)。此外，为了支持缩排显示，我硬编码了最大深度 6。顶层会比底层缩排更多，即填补空格的多少与深度成反比。由于深度难以取得，所以把缩排定义为 6 ，这样做是最简单的方式。

和使用物化路径模型一样，无论如何都必须重排序，因此可以毫无顾忌地使用 distinct 以消除重复记录。此查询如下：

```
select lpad(description, length(description) + 6 - depth) description,
       commander
from (select distinct b.description,
                      b.commander,
                      b.left_num,
                      (select count(c.left_num)
                      from nested_sets_model c
                      where b.left_num between c.left_num
                                       and c.right_num) depth
      from nested_sets_model a,
          nested_sets_model b
      where a.description like '%Highland%'
        and a.left_num between b.left_num and b.right_num
        and b.left_num > 1)
order by left_num desc
```

这个查询所显示的结果，与先前章节中物化路径查询的结果完全相同。

比较各种模型下的高地联队查询

和“Vandamme 查询”时一样，我们测试“高地联队查询”执行五千次的情况，稍有差异的是，这次邻接模型会返回重复记录。最终，邻接模型会返回五千次 25 条记录，其他模型则返回五千次 16 条记录。如果纯粹以“单位时间返回记录数”来衡量性能，使用邻接模型会把许多无效记录计算在内，因此增加“调整型邻接模型”，以每单位时间中有效记录数（即 16 条）来衡量，如图 7-3 所示。

从图 7-3 明显看出，调整前的邻接模型远远胜过另外两种模型，调整后的邻接模型仍明显胜出。另外，物化路径模型仍比嵌套集合模型快，但只是小幅领先。

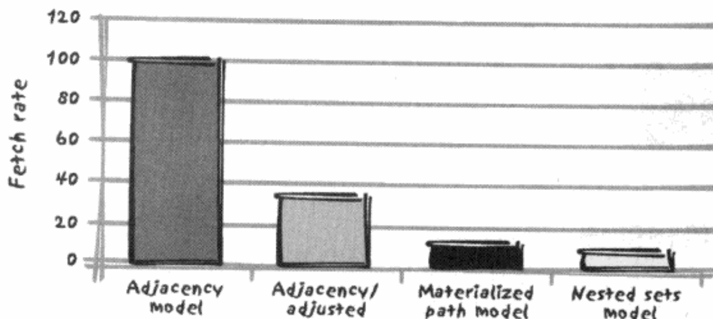


图 7-3：高地联队查询的性能比较

因此不难看出，尽管 connect by 本质上是过程性的，但自底向上、自顶向下的查询都表现卓越，当然，应为字段建立适当的索引。然而，多个起点的情况下自底向上的查询会返回重复记录，确实是件麻烦事。

当 connect by 或递归 with 不可用时，物化路径模型是良好的替代品。令人感兴趣的是它的性能比“硬连线的（hard-wired）”的嵌套集合高。

设计保存层次数据的表时，应避免一些错误。下列问题已出现在我们的例子当中。

物化路径绝不该是键，即使它们具唯一性。

的确如此。动态环境无法造就牢固的层次结构，人在层次结构中的位置也不是一成不变的。

物化路径不该暗示任何兄弟节点的排序。

排序不属于关系模型，而只与数据的展现有关。当插入新的节点或删除已存在的节点时，不必更改其他记录的内容（暂不考虑理论原因，不使用嵌套集合模型的最大原因就在于此）。在父节点最后一个儿子的位置上插入子节点总很容易。对整个树结构排序只需如下两步：首先，基于父节点的物化路径排序；之后，根据任何需要的属性对兄弟节点排序。

所选择的编码方式不见得完全中立。

无论以物化路径排序，还是以父节点的物化路径排序，都必须以路径为排序键，因此选择不会是中立的。最安全的方法，可能是在数字左边补 0，例如 001.003.004.005（注意，如果每个数值都是三位数，就可以省去分隔符）。你或许担心物化路径的长度；但假设每个父节点的子节点数量不超过 100 个，并从 0 到 99 编号，那么 20 个字符就能保存深达 10 个层次的物化路径，这意味着树的节点数高达 100 的 10 次方个——或许超过实际需要的数量。

总结：无论是自顶向下、还是自底向上，树的访问本质上都是顺序操作，所以比较慢。

聚合来自树的值
Aggregating Values from Trees

现在你已知道如何处理树，下面讨论如何聚合树状结构中保存的值，可大致分为两种情况：对保存于叶节点中的值做聚合，以及计算某个值散布在树中各层次的百分比。

对保存于叶节点中的值做聚合
Aggregation of Values Stored in Leaf Nodes

现实的案例与 Vandamme 和高地联队查询的案例相比，节点包含更多信息 —— 尤其是叶节点。例如，“团”节点应包含士兵数量，从中反应其战斗力。

为人数建模

依然采用先前的例子，并限定在 Vandamme 将军的法国第三军，我们从“旅”节点层往下访问，合理的建模（至少目前合理）如下。

表 UNITS. 每条记录描述一个团、一个师或一个旅。和表 adjacency_model、materialized_ path_models 或 nested_sets_model 很像，但没有包含指向所述的高一级隶属部队的字段。

```
ID NAME                       COMMANDER
-- -------------------------- ---------------------------------------------
 1 III Corps                  Général de Division Dominique Vandamme
 2 8th Infantry Division      Général de Division Baron Etienne-Nicolas Lefol
 3 1st Brigade                Général de Brigade Billard
 4 2nd Brigade                Général de Brigade Baron Corsin
 5 10th Infantry Division     Général de Division Baron Pierre-Joseph Habert
 6 1st Brigade                Général de Brigade Baron Gengoult
 7 2nd Brigade                Général de Brigade Baron Dupeyroux
 8 11th Infantry Division     Général de Division Baron Pierre Berthézène
 9 1st Brigade                Général de Brigade Baron Dufour
10 2nd Brigade                Général de Brigade Baron Logarde
11 3rd Light Cavalry Division Général de Division Baron Jean-Simon Domont
12 1st Brigade                Général de Brigade Baron Dommanget
13 2nd Brigade                Général de Brigade Baron Vinot
14 Reserve Artillery          Général de Division Baron Jérôme Doguereau
```

因为不同作战单元之间的关联没有保持在表 UNITS 中，所以需要另外的表来描述。

表 UNIT_LINKS_ADJACENCY. 我们再次使用邻接模型，这次单独保存部队之间的隶属关系。我们采用一个邻接列表，保持 UNITS 表每条记录的 id 与其父记录的 id 的对应关系。该 unit_links_adjacency 表隔离了结构信息：

```
        ID  PARENT_ID
---------- ----------
         2          1
         3          2
         4          2
         5          1
         6          5
         7          5
         8          1
         9          8
        10          8
        11          1
        12         11
        13         11
        14          1
```

表 UNIT_LINKS_PATH. 当然，除了邻接表，我们还可以用物化路径，如下面 unit_links_path 表所示：

```
        ID PATH
---------- -----------------
         1 1
         2 1.1
         3 1.1.1
         4 1.1.2
         5 1.2
         6 1.2.1
         7 1.2.2
         8 1.3
         9 1.3.1
        10 1.3.2
        11 1.4
        12 1.4.1
        13 1.4.2
        14 1.5
```

表 UNIT_STRENGTH. 最终，每个旅的人数保持在 unit_strength 表中：

```
        ID        MEN
---------- ----------
         3       2952
         4       2107
         6       2761
```

```
 7        2823
 9        2488
10        2050
12         699
13         318
14         152
```

计算每一层的人数

对于邻接模型，编写如下简单查询可取得第三军的总人数：

```
select sum(men)
from unit_strength
where id in (select id
             from unit_links_adjacency
             connect by prior id = parent_id
             start with parent_id = 1)
```

但是，如何获得每级战斗单元的人数呢？例如，各师人数，师可能由两个旅组成。显然，只要修改上述查询的“起点”就可以了——改用各个师的 id 代替法国第三军的 id。

现在面临一个选择：在应用中采用过程性编码方式，逐一处理所有部队，最后做合计；或者，通过一个完整的 SQL 语句解决，计算要返回的每项人数。为了直接返回实际人数，我们必须对上述查询稍加修改：

```
select u.name,
       u.commander,
       (select sum(men)
       from unit_strength
       where id in (select id
                    from unit_links_adjacency
                    connect by parent_id = prior id
                    start with parent_id = u.id)
       or id = u.id) men
from units u
```

从不同“起点”多次向下访问同一棵树，相同的记录会被多次读取，如果数据量大会严重影响性能。这体现了 connect by 的过程性本质，它使我们“无键可用(leaves us without a key to operate on)”——前面提到的“不破坏顺序就无法消除重复项”，情景何其相似。当性能问题非常严重时，我们依然别无选择，只能采用过程性处理方式。这使人想起“凡动刀的，必死于刀下”这句话，不过这里的“刀”变成了“过程（procedure)”。

如果允许使用第 11 章讲述的“技巧（black magic）”，使用物化路径的情况会稍好一些，这种技巧就是“链接分解（explosion of links）”。我们可以编写一个查询来分解 unit_links_path，虽然这个查询并不优雅。我把这个查询作为视图并命名为 exploded_links_path，其内容显示如下：

```
SQL> select * from exploded_links_path;

        ID   ANCESTOR      DEPTH
---------- ---------- ----------
        14          1          1
        13          1          2
        12          1          2
        11          1          1
        10          1          2
         9          1          2
         8          1          1
         7          1          2
         6          1          2
         5          1          1
         4          1          2
         3          1          2
         2          1          1
         4          2          1
         3          2          1
         7          5          1
         6          5          1
        10          8          1
         9          8          1
        13         11          1
        12         11          1
```

随着 depth 的取值不同，我们看到 id 和 ancestor 被隔离开了。

既然有了这个视图，对所有层次（此例不考虑底层）做 sum 操作就很容易了：

```
select u.name, u.commander, sum(s.men) men
from units u,
    exploded_links_path el,
    unit_strength s
where u.id = el.ancestor
  and el.id = s.id
group by u.name, u.commander
```

上述语句返回：

```
NAME                       COMMANDER                              MEN
-------------------------- -------------------------------------- -----
III Corps                  Général de Division Dominique Vandamme 16350
8th Infantry Division      Général de Division Baron Etienne-      5059
                           Nicolas Lefol
10th Infantry Division     Général de Division Baron Pierre        5584
                           Joseph Habert
```

11th Infantry Division	Général de Division Baron Pierre Berthézène	4538
3rd Light Cavalry Division	Général de Division Baron Jean-Simon Domont	1017

（可以通过 union，在 units 和 unit_strength 之间加上连接，以查看所显示的部队，不需要任何计算。）

同样，我们执行查询五千次，接着比较单位时间返回的记录数。不出所料，一直表现卓越的邻接模型这次大败，如图 7-4 所示。

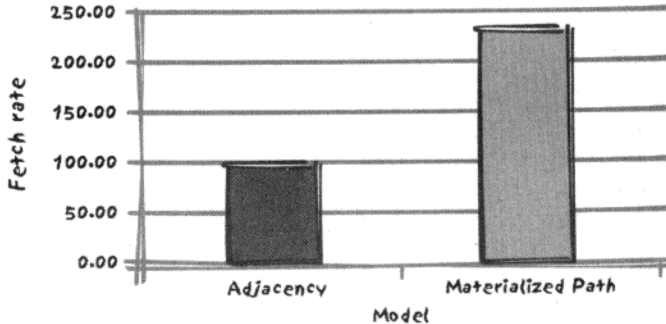


图 7-4：计算每个部队人数的性能比较

总结：需要进行聚合计算时，较简单的树实现方式有时性能很好。

散布在各层的百分比
Propagation of Percentages Across Different Levels

使用物化路径和少量可用的邻接性，就能优美高效地解决任何问题吗？很不幸，不能。我们前一个例子实际说明了，用 SQL 处理树结构仍有一些局限性。

下面，采用完全不同的例子说明此问题。假设我们经营魔药。每种魔药由多种成分（ingredient）组成，处方（recipe）列出成分及百分比。这是层次模型吗？处方可以共享某种“基础魔药”，以复合成分（compound ingredient）的形式表示，如图 7-5 所示。

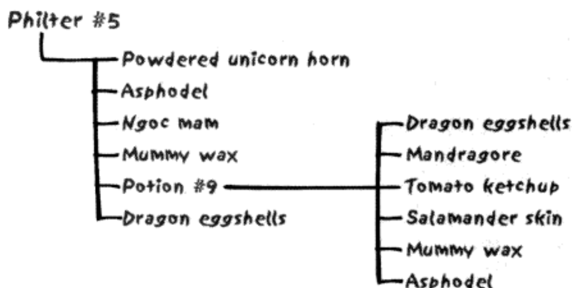


图 7-5：请勿尝试

为了满足当前的规则，我们的目标是展示整组第 5 号魔药所有基本成分的名称和比例。首先，考虑如何建立层次模型。此时物化路径很不适合，因为与在军队层次中具有单一、明确定义位置的部队不同，每种成分（包括例如第 9 号药水这样的复合成分）可以用在多种魔药中，因此路径不能作为成分的属性。另外，“扁平化”的保存魔药、并用另一个表保存魔药与各种基础成分的关联也不行，因为第 9 号药水的处方改变会影响数百个其他处方，风险太大。

因此，表示上述结构的最自然方法，是同时说明魔药包含的基本成分和复合成分。图 7-6 说明了一种建模方法。components 表为通用类型，它有 recipes 和 basic_ingredients 两种子类型，composition 表保存处方成分（可以是处方或基本成分）及其数量。

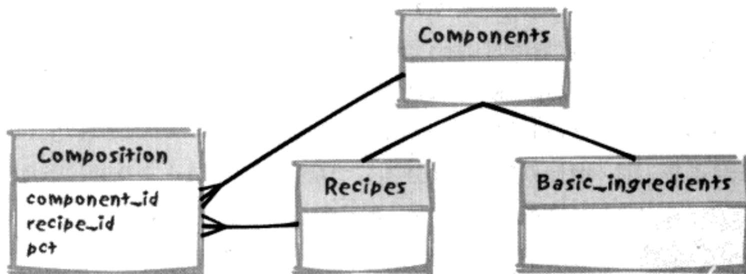


图 7-6：处方模型

然而，图 7-6 的设计使 connect by 等方法难以使用。由于 connect by 操作符的过程性本质，我们只能包含两个层次，这虽然对于图 7-5 所示的情况已足够，但不适用于一般情况。只有两层意味着（当前层为根除外）使用 connect by 可以一次"看到"两个层，即当前层和父层。例如：

```
SQL> select connect_by_root recipe_id root_recipe,
  2         recipe_id,
  3         prior pct,
  4         pct
  5         component_id
  6  from composition
  7  connect by recipe_id = prior component_id
  8  /

ROOT_RECIPE  RECIPE_ID   PRIORPCT          PCT  COMPONENT_ID
-----------  ---------- ---------- ------------  ------------
         14          14                       5             3
         14          14                      20             7
         14          14                      15             8
         14          14                      30             9
         14          14                      20            10
         14          14                      10             2
         15          15                      30            14
         15          14         30            5             3
         15          14         30           20             7
         15          14         30           15             8
         15          14         30           30             9
...
```

root_recipe 字段参照了树根，访问树的不同层次时可同时处理当前记录和先前记录的百分比；但按自顶向下的方式进行跨层合计、跨层乘法比较困难。

然而，跨层次计算百分比正是递归 with 语句的用武之地。为什么？还记得吗，在尝试显示 Vandamme 将军的下属部队时，我们必须计算层次以知道在树中的深度，并在访问中逐层取得结果。那个方法当时似乎有点罗嗦，但此方法现支持一个重要技巧。connect by 的重大缺点是，同一时间只知道两个层次：当前记录（子节点）及其父节点。如果只有两个层次，其中第 9 号药水包含 15% 的曼陀罗草，而第 5 号魔药包含 30% 的第 9 号药水，通过同时存取子节点（第 9 号药水）和父节点（第 5 号魔药）可轻易算出曼陀罗草使用的百分比是 30% *15%。但如果多于两个层次呢？我们或许会借助程

序手段（或数据库存取程序，或用户定义函数），来计算各种成分在最终产品中的比例。但我们无法只用 SQL 就作出这样的计算。

“处方中包含的各种成分，百分比是多少？”是个复杂的问题。要回答这个问题，用递归 with 是轻而易举的事。只需把当前百分比（曼陀罗草在第 9 号药水中的份量）乘上父节点的百分比（第 9 号药水在第 5 号魔药中的份量），而不是把当前层次计算成父节点层次加 1。假设各成分的名称保存在 components 表中，可按如下方式编写递归查询：

```
with recursive_composition(actual_pct, component_id)
as (select a.pct,
           a.component_id
    from composition a,
         components b
    where b.component_id = a.recipe_id
      and b.component_name = 'Philter #5'
    union all
    select parent.pct * child.pct,
           child.component_id
    from recursive_composition parent,
         composition child
    where child.recipe_id = parent.component_id)
```

假设 components 表有个 component_type 字段，包含了代表基本成分的 “I” 和代表处方的 “R”。最终，查询中应把处方过滤掉，而且，由于同样的基本成分可以出现在不同层次，所以要以成分作聚合：

```
select x.component_name, sum(y.actual_pct)
from recursive_composition y,
     components x
where x.component_id = y.component_id
  and x.component_type = 'I'
group by x.component_name
```

碰巧，虽然邻接模型表示层次结构非常自然，但它的两个实现完全不同，却有点互补。connect by 看上去比较简单（在你已了解优先级时），而且便于以缩排方式展现；而稍微难用的递归 with 支持以相对简单的方式处理极为复杂的问题——在现实项目中，更可能遇到复杂问题。看看麦片外盒或牙膏软管上打印的小字，其实和组成成分案例相似的例子还有很多。

在其他情况下（就算 DBMS 支持 connect by），要实现“单一 SQL 语句”产生结果，恐怕只能编写用户自定函数了，如果 DBMS 无法访问树，此时就必须递归。

总结：访问树的语法越复杂，越容易以纯 SQL 来解决较复杂的问题。

本章描述的方法，在数据量很少时效果令人满意，但对大数据量的处理“像老爷车一样慢”。此时，你或许会考虑非规范化模型、或基于触发器的扁平化数据模型。许多人（包括我自己）会对非规范化皱眉头，我不建议对关系模型“屡遭诟病的缓慢本性（oft-cited inherent slowness）”反规范化，这很容易遮掩程序设计中的问题。不过，SQL 确实缺乏处理树结构的强大的、可伸缩的手段。

第 8 章

孰优孰劣：认识困难，处理困难

Weaknesses and Strengths: Recognizing and Handling Difficult Cases

没有人可以保证战争的胜利，只能等待它的回报。

——温斯顿· 丘吉尔（1874—1965）

棘手的情况经常出现。对作战而言，经常要面对不利的战场条件；对 SQL 而言，经常要用较弱的手段处理大量数据。本章中，我们将讨论会面临的许多困难及解决策略，更为重要的是，辨别哪些策略其实是陷阱。在机械制造领域，运动部件越多，出问题的几率就越大，其实复杂的架构也是如此。很不幸，令人兴奋的新技术（或改头换面的旧技术）常使人们忘记一项重要原则：保持简单（keep things simple）。通常，更简单意味着更快速、更健壮（robust）。但是，数据库简单，不等于开发简单，反而常常需要更多的开发技巧。

本章中，我们首先讨论如何改进"看似高效，其实不然"的查询条件；接着，讨论抽象"持久层（persistency layer）"与分布式系统的危险因素；最后，通过一些 PHP/MySQL 例子，说明如何选择查询条件才能兼顾灵活性（flexibility）与效率（efficiency）。

看似高效的查询条件
Deceiving Criteria

正如第 6 章所述，有些查询条件组合的可选择性颇高，但各条件单独使用效果却不佳，很难获得高性能。

另外，有些选择条件乍一看很高效，但需要稍加改进才能发挥潜力。信用卡验证过程是个很好的例子。信用卡号包含好几段信息，如信用卡类型、发卡行等。下面看例子：某个西欧国家，旅客如云，如何在收费道路上进行付费控制。我们必须检查世界各地的发卡行发行的信用卡，数量巨大，而且每种信用卡都有自己的卡号规则。

信用卡号最多由 19 位数字（也有例外）组成，例如万事达卡（MasterCard）为 16 位、威萨卡（Visa）为 16 位或 13 位、美国运通卡（American Express）为 15 位……。对所有信用卡而言，前六位数字代表发卡机构，而最后一位数字代表校验和（checksum），校验和可以检查输入错误。为了进行支付控制，首先可以粗略检查发卡机构是否已知、校验和是否正确——不过，众所皆知的校验和算法（可以在因特网上找到）很容易被造

假。较精细的控制，还会根据发卡机构检查卡号的位数是否正确，卡号的前缀部分是否在有效取值范围。在我们的例子中，卡号的前缀部分的有效列表约由 200 000 组各种长度的数字所组成。

如何检查卡号是否在信用卡发卡机构的有效值范围内呢？以下的查询够简单了：

```
select count(*)
from credit_card_check
where ? like prefix + '%'
```

语句中的“where ?”指出要检查的卡号，而“ + ”表示字符串串接——也常由 `||` 或 `concat()` 完成。必须为 `prefix` 字段加上索引，每次需进行全表扫描。

为何会发生全表扫描？当我们只处理键的最左边时，不是已利用索引了吗？没错，但“要检查的值属于键值的最左边部分”，与“键是欲检查值的最左边”完全不同。差异很细微，两种情况是彼此的镜像。

假设要验证的信用卡号是 4000 0012 3456 7899（注 1）。如果 `credit_card_checks` 表保存有 312345、3456、40001 等值，这 3 个值是已排序的前置码。但是，如果它们以字符串形式保存，则为升序排序，以数值方式保存则不是升序。不仅如此，还有更多事情令人担心。

当向下访问树（我们的索引）时，会有个值与保存在树中的键做比较。如果这个值与保存在当前节点中的键相等，则完成，否则还要依据这个值小于或大于节点继续搜索子树。当前置码的长度固定时不会有问题，但前置码的长度可变时（我们例中的 312345、3456、40001 即如此），就必须每次比较不同数量的字符，这使 SQL 引擎不知如何执行索引搜索。

那么如何解决呢？很庆幸，可以利用 like 这类可选取一段范围的操作符。例如形式为 4000% 的 16 位数维萨卡号，实际上代表从 4000000000000000 到 4000999999999999

注 1：无效卡号，请勿吃惊。

之间的所有卡号，如果上下限的数字上有复合索引，就可以通过检查索引轻易地查找卡号。这是卡号为 16 位数的情况，其实位数可变的情况也容易解决。所有信用卡的最大位数是 19 位，在维萨卡号右边补三个 0 让其成为 19 位即可。

相反，如果不保存前置码，可以用 lower_bound 和 upper_bound 两个字段解决。lower_bound，前置码右边补 0 至 19 位；upper_bound，前置码右边补 9 至 19 位。这种设计，属于非规范化设计，不过，这是只读的参照表，可稍微放宽标准。再对（lower_bound, upper_bound）加上索引，并按下列方式编写查询，就可看到查询速度如飞了：

```
where substring(? + '0000000000000000000', 1, 19) between lower_bound
and upper_bound
```

很多 DBMS 产品直接实现了 rpad() 函数，提供"从右填补"的功能。当检查不定长的前置码时，解决方案就是常见的索引范围扫描（index range scan）。

总结：遇到"前置码比较"或"基于键的一部分比较"等特殊情况时，可使用范围条件（range condition）来表达，并尽量使用下限值和上限值。

抽象层
Abstract Layers

通过建立层层抽象可以增强可维护性和可重用性。这很有价值，但很不幸，抽象常被滥用，尤其是数据库访问常通过抽象层封装。当然，业界的这种现象与面向对象语言有关。

下面以实际的程序代码，说明如何不封装数据库访问。既然本书名为《SQL 语言艺术》，下列 C#代码就只保留 SQL 语句部分：

```
1    public string Info_ReturnValueUS(DataTable dt,
2                                     string   codeForm,
3                                     string   infoTxt)
4    {
5      string returnValue = String.Empty ;
```

```
6    try
7    {
8      infoTxt = infoTxt.Replace("'","''");
9      string expression = ComparisonDataSet.FRM_CD
10                         + " = '" + codeForm
11                         + "' and " + ComparisonDataSet.TXT_US
12                         + " = '" + infoTxt + "'" ;
13        DataRow[] drsAttr = dt.Select(expression);
14
15        foreach (DataRow dr in drsAttr)
16        {
17         if (dr[ComparisonDataSet.VALUE_US].ToString().ToUpper().Trim()
18                                 != String.Empty)
19           {
20            returnValue = dr[ComparisonDataSet.VALUE_US].ToString() ;
21            break;
22           }
23        }
24      }
25      catch (MyException myex)
26      {
27        throw myex ;
28      }
29      catch (Exception ex)
30      {
31        throw new MyException("Info_ReturnValueUS " + ex.Message) ;
32      }
33      return returnValue ;
34    }
```

无需成为C#专家，即可理解这段程序。这段程序返回与“消息代码（message code）”相关的文字，但这些文字必须以特定语言返回（本例中的“US”指美式英语）。这段程序代码来自多语言系统，系统里还有其他功能完全相同的函数，只是“US”变成了其他字母。毫无疑问，相同的程序代码将被复制多次，只不过用其他的 ISO 代码代替“US”而已。每次改变程序，都必须重复更改许多份完全相同的函数（……只有 ISO 代码无需修改），易维护性何在？！尽管我相信“令人激动的管理表现预示着现代语言能拯救什么（what exciting management presentations promise modern languages to deliver）”，但对上面代码的质疑是无可厚非的。

下面更仔细地研究这个程序。第 9~12 行的字符串表达式，是糟糕的硬编码（hardcoding），用来构造 Select()将执行的查询。实际上，有两种不同类型的元素被硬编码：“要传递给查询的实际值”（codeForm 和 infoTxt），以及“字段名称”（保存在 ComparisonDataSet.FRM_CD 和 ComparisonDataSet.TXT_US 属性中——很明显，每种被支持的语言各占一个字段，这种设计有些问题）。字段名称只能被硬编码，但不该有太多组字

段名称，否则要编写的不同查询数量太多。实际值的传递也是一样：要查询的值与记录数量一样多，甚至更多（不过查询不到内容）。从 codeForm 和 infoTxt 到 SQL 语句的硬编码传值方式，会严重影响性能，因为这类“给我相关标签（give me the associated label）”的查询可能被调用非常多次，而每次调用都会触发完整的解释机制，判断最佳执行计划……但却没有带来好处。这些值应该当成绑定变量传递给查询——就像传递给函数的自变量一样。

第 15-23 行的循环也值得质疑。要查找数据集中第一个“非空值”，为何不直接利用 SQL、而写到外部应用程序中呢？ SQL 语言可以完美解决此问题。为何要从服务器返回不必要的记录，然后把它们抛弃呢？未免太多此一举了吧。首先，这加重了数据库服务器的负担：即使在第一次迭代就跳出循环，但为了优化网络通信量常需要预读（pre-fetch），在应用开始第一个循环之前服务器已准备好了要返回的成千上万条纪录。其次，应用服务器也作了更多工作，因为它必须过滤掉数据库辛苦返回的大部分数据。最后，不用说，开发者要写更多代码。其实，只要对表达式增加适当条件，就可轻易阻止返回不需要的记录。当 C# 程序代码产生查询时，服务器不会知道我们只对第一条非空记录感兴趣，所以只能“照章办事”。如果尝试在数据库端查找不良程序代码的线索，可能发现的线索无非是许多几乎完全相同的硬编码语句。然而，这只是“冰山一角”。

用任何编程语言都能写出糟糕的代码，从朴素古老的 COBOL 到最酷的面向对象语言皆是如此。但软件各层之间的独立程度越大，每层的编码质量就必须越高，因为多层调用可能会影响性能：无论如何，最终性能会收到编写最差的那一层的影响。

如果有些遗留的库设计不佳，而你又必须使用它，那么上述问题会更严重，因为你不能修改这些库。无论从进度安排（schedule）还是从预算（budget）来讲，几乎都不容许重写效率低下的底层程序。有个例子使我获得了经验教训，其中，编程语言的某个操作符（operator）被“复载（overloaded）”，而开发者在不知情的情况下用这个操作符进

行数据库读写！要发现这种问题很难，因为查询语句看起来再平常不过了，不像要扫描几百万条记录并引起性能下降的糟糕查询那样容易识别。

总结：很酷的“数据库读写库（database access libraries）”未必高效。

分布式系统
Distributed Systems

无论你所谓的“分布式系统”指联合系统（federated systems）、所连接的服务器（linked server），还是数据库连接（database link），分布式查询的原理都是一样的：你要查询的数据，并不是存储在你连接的服务器中（对 Oracle 用户而言，还可能是数据并不存储在你连接的数据库中）。分布式查询的执行机制很复杂，对远程更新更是如此，因为必须保证事务完整性（transaction integrity）。这种复杂性带来了很大代价，很多人没有充分意识到这一点。

下面我们将通过例子进行对比测试（数据库采用 Oracle）：首先，对本地表执行大量 insert 和 select 操作；然后，通过如下三种数据库连接进行相同操作：

进程间（Inter-process）

通过进程间通信的方式进行互操作，通常数据库就在同一主机上（注 2）。这种情况不涉及网络通信。

回环（loop-back）

通过 TCP 连接，但指定了回环地址（127.0.0.1）将访问限制在网络层。

IP 地址（IP address）

指定了机器的实际 IP 地址，但同样没有真正使用网络，所以不必考虑网络响应时间。

如图 8-1 所示，测试结果很有启发。虽然进程间通信、TCP 回环、或正规 IP 模式的连接之间的确有差异，但性能差异主要来自是否使用了数据库链接。基于数据库链接进行插入操作时，每秒插入的记录数是本地操作的 20%，而读取操作每秒返回的记录数是本地操作的 40%（操作以“逐条记录”方式进行）。

注 2：别忘了 Oracle 所称的“数据库”就是大多数数据库系统所称的“服务器”。

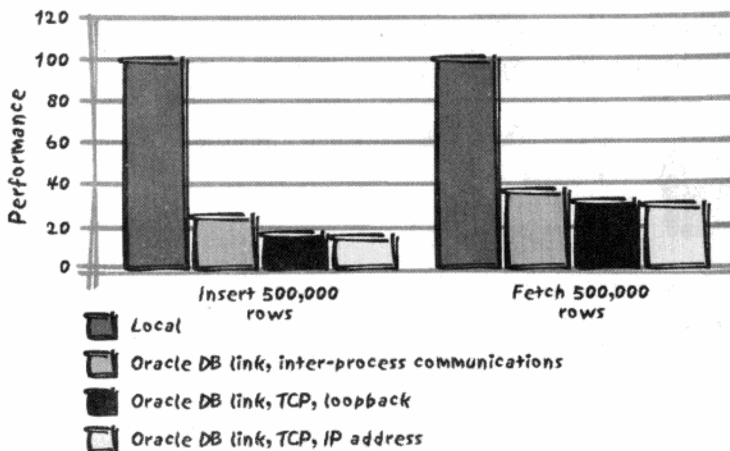


图 8-1：远程数据的代价

必须跨异质系统执行交易时，除了使用数据库链接或类似技术外，别无选择。只要需求，无论成本多高，都要借助相关机制保证数据的完整性。然而，在许多情况中，专用服务器（dedicated server）是架构上的选择，目的是为了访问数据。对于临时性的远程数据访问，性能损失是可以接受的。例如，连接时可能根据远程服务器核对身份，只要远程服务器处在工作状态，人们往往注意不到这一活动。然而，如果需要向本地数据库加载大量数据，而每一行数据的加载需要访问远程服务器进行验证时，则性能必然很差。逐一验证记录，本身就是糟糕的主意（设计合理的数据库，所有验证都应通过完整性约束执行），而远程检查又比本地检查慢两到三倍。

涉及不同服务器的分布式查询也相当麻烦。首先，查询的优化由 DBMS 核心负责，优化器会决定如何划分查询、分配工作、协调远程和本地的活动，最后把不同片段组合起来。当这一切都发生在本地服务器时，确定执行路径已很复杂。而“分布”的概念是物理的更是逻辑的：一部分性能损失是由于远程数据库的字典信息在本地无法访问造成的。使用放在同一台机器、但彼此无关的两个数据库，其成本损失明显高于放在两台不同的服务器、但共享联合数据库且分享数据字典信息的两个数据库。

在分布式与并行查询（当查询被划分为可并行执行的独立区块时）之间，有很多共通之处。如前所述，网络层增加了额外的困难，严重影响了某些操作的速度；而在某某无法取得所有数据字典信息里，也使并行执行的区块划分更冒险。另外，若数据源是异质的（例如某查询的数据来自 Oracle 和 SQL Server），优化器经常无法获得所需信息。当然，多数产品会收集同类信息以便查询最佳化，但出于多种原因它们无法互相合作：第一，优化器到达是怎么工作的，每个开发商都严格保密。第二，产品版本不同，优化器的工作方式会发生变化。第三，Oracle 优化器绝对无法完全利用 SQL Server 的细节功能，反之亦然。最终，在不同产品的优化器之间，只有“最大公约数”才能有效分享。

甚至在使用多个同质数据源时，处理过程也受到很大限制。例如，跨网络读取的代价，要比所有处理都在本地完成时高。不难理解，优化器不应处理服务器间来回切换的路径问题，而是应尽量在靠近数据的地方进行过滤操作，而接下来，由 SQL 引擎负责传递产生的数据，供下一个处理步骤使用。如第 4 章和第 6 章所述，当没有其他搜索条件时，采用关联子查询（correlated subquery）测试存在性是很糟糕的，例如：

```
select customer_name
from customers
where exists (select null
              from orders,
                   orderdetails
              where orders.customer_id = customers.customer_id
                and orderdetails.order_id = orders.order_id
                and orderdetails.article_id = 'ANVIL023')
```

在表 customers 中扫描到的每一行，都要调用针对 orders 和 orderdetails 的子查询。

如果表 customers 在甲机器上，而表 orders 和 orderdetails 在乙机器上，则情况更糟。考虑到读取每条记录的成本很高，合理的解决方案应把关联子查询转换为非关联子查询：

```
select customer_name
from customers
where customer_id in (select orders.customer_id
```

```
                from orders,
                     orderdetails
                where orderdetails.article_id = 'ANVIL023'
                  and orderdetails.order_id = orders.order_id)
```

而且，子查询会在远程机器上执行，即使编写如下查询也不例外：

```
select distinct customer_name
from customers,
     orders,
     orderdetails
where orders.customer_id = customers.customer_id
  and orderdetails.article_id = 'ANVIL023'
  and orders.order_id = orderdetails.order_id
```

另一个问题是，现在优化器能做出正确选择吗？最好不要冒险。显然，引入远程数据来源后，减少了最高效查询方式的选择。此外，别忘了外层查询介入之前，子查询必须执行完毕并返回数据。执行时间必会增加，因为没有任何操作能与另一个操作并发执行。

如何确保两个远程表的 join 操作在远程执行呢？最安全的方法是在远程机器查询 join 操作定义的视图。例如，在先前的例子中，定义如下的 vorders 视图是个好主意：

```
select orders.customer_id, orderdetails.article_id
from orders,
     orderdetails
where orderdetails.order_id = orders.order_id
```

查询视图 vorders，避免了 DBMS 分别读取两个远程表的数据，然后在本地完成 join 操作。不用说，如果前例中的表 customers 和 orderdetails 在同一个服务器中，而表 orders 在其他服务器里，则非常危险。

总结：优化器只能适当处理它清楚的本地数据。与远程数据的频繁交互会影响性能。

动态定义的搜索条件

Dynamically Defined Search Criteria

有的性能问题比较容易发现（批处理程序糟糕的性能却可能暂时发现不了），其最常见的原因之一是使用动态定义的搜索条件。实际上，动态条件背后是一连串可怕的需求

（dreaded requirement）：支持用户通过界面输入搜索条件和排序条件。

这类应用一般表现为：开始查询执行得相当好，但随着时间的推移，查询变得非常非常慢。此类问题很难修正，因为所有东西都是动态的。

需要进行“动态搜索”的应用，查询通常被设计成两个步骤，如图 8-2 所示。首先，屏幕显示宽泛的条件选项和一系列可能条件，例如“除去（exclude）”或“时间介于（date between ... and ...）”等；然后，使用这些条件动态建立查询，以返回特定数据的列表；当选取列表中具体项之后，才显示其详细信息。

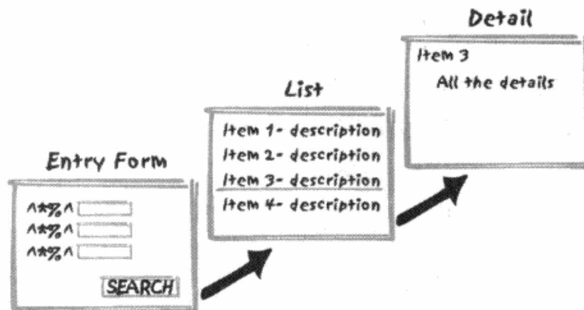


图 8-2：一个典型的多条件搜索

要查询的是相同表的相同字段，而搜索条件会发生变化，此时，成功的关键是如何聪明地产生 SQL 查询。下面将以电影数据库为例进行说明，要解决的问题是如何根据不同条件返回电影列表。此例使用的环境是广为流行的组合：PHP 加 MySQL。不用说，此例说明的技术并不局限于 PHP、MySQL 或电影数据库。

设计简单的电影数据库及主要查询

Designing a Simple Movie Database and the Main Query

Movies 表定义如下：

```
Table MOVIES
  movie_id          int(10) (auto-increment)
  movie_title       varchar(50)
  movie_country     char(2)
  movie_year        year(4)
  movie_category    int(10)
  movie_summary     varchar(250)
```

我们还需要一个 categories 表（会被 movie_category 上的外键参照），用来保存电影类型，例如动作片（Action）、戏剧（Drama）、喜剧（Comedy）、音乐剧（Musical）等。或许有人会说，一部电影可以属于多个类型，所以更好的设计是引入额外的表以支持电影类型与电影之间的多对多关系。为简单起见，我们的例子假设每部电影属于一种类型。

我们需要保存演员及导演信息的表吗？分别建立这两个表是错误的设计，因为演员后来当导演的现象很常见，而重复保存个人信息是不必要的，何况导演经常就是主要演员之一。

因此，我们还需要 3 个表：people 表用来保存名字、姓氏、性别、出生年份等信息；roles 表定义这部电影需要多少角色参与（演员、导演、作曲者、摄影指导等）；而 movie_credits 表说明电影参与者们的工作。图 8-3 所示为完整的 schema。

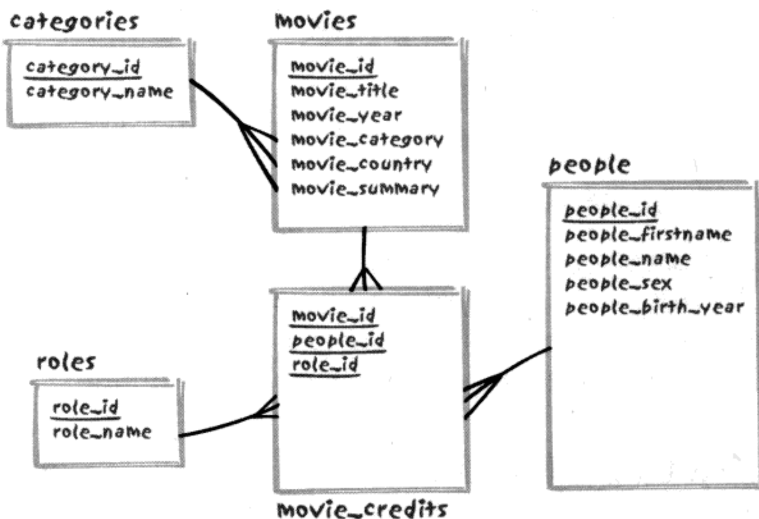


图 8-3：电影数据库 Schema

我们假设，人们可以在数据库中搜索电影，所指定的条件为：电影名称中的关键词、导演或任意 3 名演员。以下为页面原型的源码，用 HTML 编写：

```
<html>
<head>
  <title>Movie Database</title>
```

```
</head>
<body>
<CENTER>
    <HR>
    <BR>
    Please fill the form to query our database and click on <b>Search</b> when you
are done...
    <BR>
    <HR>
    <BR>
<form action="display_query.php" method="post">
    <TABLE WIDTH="75%">
    <TR>
      <TD>Movie Title :</TD>
      <TD><input type="text" name="title"></TD>
    </TR>
    <TR>
      <TD>Director    :</TD>
      <TD><input type="text" name="director"></TD>
    </TR>
    <TR>
      <TD>Actor       :</TD>
      <TD><input type="text" name="actor1"></TD>
    </TR>
    <TR>
      <TD>Actor       :</TD>
      <TD><input type="text" name="actor2"></TD>
    </TR>
    <TR>
      <TD>Actor       :</TD>
      <TD><input type="text" name="actor3"></TD>
    </TR>
    <TR>
      <TD COLSPAN="2" ALIGN="CENTER">
      <HR>
      <input type="Submit" value="Search">
      <HR>
      </TD>
    </TR>
    </TABLE>
</form>
</CENTER>
</body>
</html>
```

页面原型显示如图 8-4 所示。

首先，先做几点说明：

- 虽然我们把名和姓分别保存在数据库中（产生依姓排序的列表就比较方便），但用户会看到单一的姓名字段，界面搞得像重新申请护照一样就太复杂了。
- 查询的输入值不区分大小写。

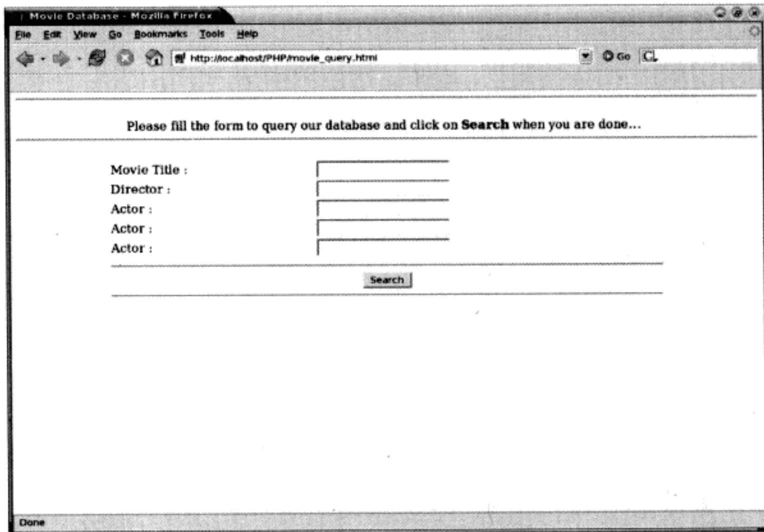


图 8-4：电影数据库搜索页面

绝对不要编写包含如下条件的查询：

```
and upper(<value entered for actor1>) =
        concat(upper(people_firstname), ' ', upper(people_name))
```

就如第 3 章中所述，这种等式的右半部会阻止我们使用该字段上的正规索引。有的产品支持建立函数索引（即为表达式的结果加上索引），但最简单、也是最好的解决方案是：

1. 系统地以大写来保存可能被查询的所有字符字段（输出之前可以通过函数美化）。

2. 把名字传递给查询之前，将输入内容划分为名和姓。

为了满足上述第 1 点，只需每次插入 upper（*string*）而不是 *string* 即可，这很简单。上述第 2 点稍后讨论。

如果用户填入所有字段，则最后产生的查询如下：

```
select movie_title, movie_year
  from movies
      inner join movie_credits mc1
        on mc1.movie_id = movies.movie_id
      inner join people actor1
        on   mc1.people_id = actor1.people_id
      inner join roles actor_role
        on   mc1.role_id = actor_role.role_id
        and  mc2.role_id = actor_role.role_id
        and  mc3.role_id = actor_role.role_id
```

```
        inner join movie_credits mc2
          on mc2.movie_id = movies.movie_id
        inner join people actor2
          on  mc2.people_id = actor2.people_id
        inner join movie_credits mc3
          on mc3.movie_id = movies.movie_id
        inner join people actor3
          on  mc3.people_id = actor3.people_id
        inner join movie_credits mc4
          on mc4.movie_id = movies.movie_id
        inner join people director
          on  mc4.people_id = director.people_id
        inner join roles director_role
          on mc4.role_id = director_role.role_id
  where actor_role.role_name = 'ACTOR'
    and director_role.role_name = 'DIRECTOR'
    and movies.movie_title like 'CHARULATA%'
    and actor1.people_firstname = 'SOUMITRA'
    and actor1.people_name = 'CHATTERJEE'
    and actor2.people_firstname = 'MADHABI'
    and actor2.people_name = 'MUKHERJEE'
    and actor3.people_firstname = 'SAILEN'
    and actor3.people_name = 'MUKHERJEE'
    and director.people_name = 'RAY'
    and director.people_firstname = 'SATYAJIT'
```

很不幸，能同时说出影片名称、导演和 3 名主要演员的人通常是电影迷，他们不太可能需要我们的电影数据库。所以，填写一个或两个字段后进行搜索的情况较为常见。于是，我们必须预先考虑空字段的处理问题：没有传值怎么办？

为此，最常见的编程手段是：保持选取列表不变；之后，使用适当的 join 条件，把可能会以某种方法相互影响的所有表连接在一起；然后，用一长串如下所示的项目取代先前的简单条件：

```
and column_name = coalesce(?, column_name)
```

其中 “?” 将与来自输入字段的值相结合，而函数 coalesce() 返回第一个非空的自变量。每提供一个值，就应用一个过滤条件；否则，“所有值”都会通过检测。

“所有值”？实际上并非如此；如果有字段包含 null，针对那个字段的条件就返回 false。我们无法假定未知的两个项目是否相等。如果一长串由 and 连结的条件中，有个条件返回 false，查询就会什么都不返回，这当然不是我们要的。解决如下：

```
and coalesce(column_name, constant) = coalesce(?, column_name, constant)
```

只要利用了任何 column_name 字段的索引、并且至少为一个参数提供了值，这个解决方案绝对是完美的。性能和结果的正确性，必须牺牲其一吗？第二个解决方案应该更合理，但不幸的是，这两种方案都可能砸了我们的饭碗（sacrifice our job），真是讨厌。

无论如何，很难写出在所有情况下都能正常工作的查询。常被采用的解决方案，就是动态建立查询。例子中，我们依据 where 和角色名称上的固定条件，把字符串保存的所有内容与程序同用户输入的条件串接——只串接那些条件。

总结：搜索条件的数量不确定时，需要动态建立查询。

假设用户搜索电影明星成龙（JACKIE CHAN），最后产生的动态查询可能如下：

```
select distinct movie_title, movie_year
    from movies
         inner join movie_credits mc1
           on mc1.movie_id = movies.movie_id
         inner join people actor1
           on  mc1.people_id = actor1.people_id
         inner join roles actor_role
           on  mc1.role_id = actor_role.role_id
           and mc2.role_id = actor_role.role_id
           and mc3.role_id = actor_role.role_id
         inner join movie_credits mc2
           on mc2.movie_id = movies.movie_id
         inner join people actor2
           on  mc2.people_id = actor2.people_id
         inner join movie_credits mc3
           on mc3.movie_id = movies.movie_id
         inner join people actor3
           on  mc3.people_id = actor3.people_id
         inner join movie_credits mc4
           on mc4.movie_id = movies.movie_id
         inner join people director
           on  mc4.people_id = director.people_id
         inner join roles director_role
           on mc4.role_id = director_role.role_id
    where actor_role.role_name = 'ACTOR'
      and director_role.role_name = 'DIRECTOR'
      and actor1.people_firstname = 'AMITABH'
      and actor1.people_name = 'BACHCHAN'
    order by movie_title, movie_year
```

首先，需要说明两点：

- 必须使用 select distinct，因为在没有任何额外条件的情况下也要保持连接。否则，针对每部电影所返回的记录数，会和电影所记录的演员和导演数一样。

- 把接收的值与需要构建的 SQL 文本串接起来，以获得上文所述的查询语句，是很不错的想法。但实际上，这不是我们应该做的。我们曾提到过“绑定变量（bind variable）”的话题，现在是说明其工作方式的时候了。正确的做法是：以“?”作为占位符（具体字符因语言而异）建立查询，然后调用特殊函数，将实际值与占位符绑定。这看似是为开发者着想，但实际上，是为 DBMS 着想，使 DBMS 引擎做较少的事。这样一来，即使每次重建此查询，DBMS 也可通过 cache 找到语句，而此时的语句已经被 DBMS 解释、并优化过了；由于使用了占位符，所有以相同样式建立的查询（例如搜索由特定演员主演的电影）对应于 cache 中的同一条 SQL 语句，而不是演员姓名的不同会产生新的 SQL 语句；之后，当所有设置都已完成，查询就可以立即执行了，而最终用户得到的是更快的响应速度。

除了性能之外，动态建立的硬编码查询还存在验证的“安全性”问题：这样的查询会对所谓的“SQL 注入（SQL injection）”技术大开门户。什么是 SQL 注入呢？假定只有订阅者才被允许查询我们的电影数据库，不过，每个人都可以访问 1960 年之前的电影。如果有个非订阅者，恶意在 `movie_title` 字段输入以下内容：

```
X' or 1=1 or 'X' like 'X
```

当把输入字段与查询语句串接时，就会产生如下条件：

```
where movie_title like 'X' or 1=1 or 'X' like 'X%'
  and movie_year < 1960
```

这组条件永远都返回 true，显然不会过滤任何东西！把输入字段与 SQL 语句串接后，意味着：任何非订阅者都可查看整个数据库——当然，你的应用系统中的数据可能比电影数据库更加机密。绑定变量可防止 SQL 注入。对任何在线数据库，SQL 注入都是非常现实的安全威胁，应特别小心，以避免恶意使用。

总结：基于性能和安全性（SQL 注入）的考虑，动态建立查询应使用参数标记（parameter marker），并以绑定变量的方式传值。

当表已建立了合适的索引，基于 join 和动态串接的过滤条件的查询执行得非常快。但仍有一些事情让人烦恼：先前作为例子的查询非常复杂，而输出结果和输入条件却非常简单。

合理精简查询大小

Right-Sizing Queries

实践上，查询的复杂度只是问题的一部分。在前一节最后的查询中，如果没有把导演的名字记录在数据库中，或是只知道两个主要演员的名字，会如何呢？答案是不会返回任何记录。那么，此时无法使用外连接吗？（外连接在有对应数据时返回对应数据，否则返回 null。）

使用外连接是种解决方案，除非不知道将被查询的数据。如果数据库中只有导演的名字，又会怎么样？实际上，每处都需要外连接，但理论上这是不可能的。于是，出现了有趣的情况：即使所有属性都必要，而且数据库中绝对没有 NULL 值，但我们却查询不到信息；一切都因为，查询一直采用不能令人满意的 join。

实际上，只提供一个演员的名字，则查询不会很复杂：

```
select movie_title, movie_year
from movies
     inner join movie_credits mc1
       on mc1.movie_id = movies.movie_id
     inner join people actor1
       on  mc1.people_id = actor1.people_id
     inner join roles actor_role
       on  mc1.role_id = actor_role.role_id
where actor_role.role_name = 'ACTOR'
  and actor1.people_firstname = 'AMITABH'
  and actor1.people_name = 'BACHCHAN'
order by movie_title, movie_year
```

这个“量身定做”的查询假设我们不知道导演的名字，也不晓得其他演员，因此，无需外连接。既然是动态建立查询，为何不尝试注入更多智慧，以“按单定制(built to order)”的方式建立查询呢？程序会更复杂，但这样的复杂性值得吗？首先，提供了演员名字，即使不知道他参与的影片由谁执导，也会返回所有可用的信息；这个简单事实足以成为无条件赞成“按单定制”查询的理由。另外，性能更优也是“按单定制”查询的理由。

要说明两种方法之间的差异，最有说服力的做法就是在循环中执行足够多次查询："量身定做"查询比"单一尺寸"查询快5倍。先不考虑其他因素，如果查询要花0.001秒、而不是0.005秒，会有何影响？偶尔查询不会有太大影响，但某天查询的频率高过了服务能力时，我们就有麻烦了。查询会排队，队列的长度快速增加——抱怨性能的电话数量也以同样的速度增长。掐指一算，"量身定做"查询的速度快了5倍，意味着能让同一台硬件处理5倍的查询量……（第9章会详细讨论这一问题。）

总结：以"量身定做"方式动态创建的查询，通过尽量减少join次数、消除遗漏值问题，最终提高了性能。

用PHP封装SQL

Wrapping SQL in PHP

编写PHP页面之前，先看看常规HTML：

```
<html>
     <head>
        <title>Query result</title>
    </head>
    <body>
    <CENTER>
      <table width="80%">
        <TR><TH>Title</TH><TH>Year</TH><TR>


    <?php
     ...
```

（如果使用样式表，我们的页面会更美观……）

成功连接数据库之后，第一件事就是取得提交给登录界面（entry screen）的数据。由于所有东西都以大写保存入数据库，所以可把用户输入直接转换为大写，这当然可以在SQL 程序代码中完成，但用 PHP 并不会花什么代价：

```
$title=strtoupper($_POST['title']);
$director=strtoupper($_POST['director']);
$actor1=strtoupper($_POST['actor1']);
$actor2=strtoupper($_POST['actor2']);
$actor3=strtoupper($_POST['actor3']);
```

我们现在有个技术上的问题，与 PHP 绑定的实现有关。以下是用 PHP 绑定变量的过程：

1. 先在要传递给查询的每个参数的位置上写一个“ ?”。

2. 然后，调用 bind_param()，它的第一个变量是字符串，其中每个字符代表一个要绑定的值，同时字符也说明了要传递参数的类型（本例为代表字符串类型的 “s”），接着是不定数量的参数——每个要绑定的值各为一个参数。

所有的参数都以位置来识别（使用 JDBC 时也是如此，但不是所有的数据库系统和语言都是这样，例如，SQLJ 程序会通过名称引用绑定变量）。但主要问题是对 bind_param() 的单一调用：当确切知道有多少个参数要绑定时没有问题，但本例无法预知用户会输入几个值。如果有办法能逐一对值做绑定，就方便多了。

有个绑定不定数量值的方法（或许不是最好的方法）：逐一查看从窗体收到的所有变量，检查哪些变量有值，依次把每个值保存在数组中。在我们的例子中可以这么做，因为所有值的类型都是字符串。其他值呢？例如电影的首映年份，可以在 PHP 程序中把年份当成字符串处理，并在 SQL 程序中把它转换为数字或日期类型。

可以使用$paramcnt 变量累计窗体的用户提供了多少个参数，并把那些值保存在$params 数组里：

```
$paramcnt=0;

if ($title != "") {
   $params[$paramcnt] = $title;
   $paramcnt++;
  }
```

人名的处理稍微复杂些。我们已决定提供单一字段输入名字，这比分别输入名和姓的“用户友好性”更高。然而，对姓名字符串和 people 表中名和姓的串接做比较时，会阻碍查询使用“姓（last name）”字段上的索引，而且还可能产生错误的结果（例如，如果用户在名和姓之间误输了两个空格，而不是一个空格，我们就不会找到那个人）。

因此，我们要把输入字段分解为名和姓，假设姓是最后一个词，而名可能由零到多个词

组成，出现在姓之前。在 PHP 中，可以轻易地编写这样的函数，假设以引用的方式传递参数：

```
function split_name($string, &$firstname, &$lastname)
{
/*
 *   We assume that the last name is the last element of the string,
 *   and that we may have several first names
 */
 $pieces = explode(" ", $string);
 $parts = count($pieces);
 $firstnames = array_slice($pieces, 0, $parts - 1);
 $firstname = implode(" ", $firstnames);
 $lastname = $pieces[$parts - 1];
}
```

此函数将$director 分解成$dfn 和$dn，把$actor1 分解成$a1fn 和$a1n，依此类推，所有项都以相同模式处理：

```
if ($director != "") {
   /* Split firstname / name */
      split_name($director, $dfn, $dln);
   if ($dfn != "")
     {
      $params[$paramcnt] = $dfn;
      $paramcnt++;
     }
    $params[$paramcnt] = $dln;
   $paramcnt++;
  }
```

检查参数之后，就可以建立查询了。插入绑定变量的参数时要非常小心，其顺序应与它们出现在$params 数组中的顺序完全相同：

```
$query = "select movie_title, movie_year "
       ."from movies";
/* Director was specified ? */
if ($director != "")
   {
    $query = $query." inner join movie_credits mcd"
                    ."  on mcd.movie_id = movies.movie_id"
                    ." inner join people director"
                    ."  on  mcd.people_id = director.people_id"
                    ." inner join roles director_role"
                    ."  on mcd.role_id = director_role.role_id";
   }
/* Any actor was specified ? */
if ($actor1.$actor2.$actor3 != "")
   {
   /*
    *   First the join on the ROLES table
    */
    $query = $query." inner join roles actor_role";
```

```
/*
 *  Even if only one actor was specified, we may
 *  not necessarily find the name in $actor1 so careful
 */
 $actcnt = 0;
 if ($actor1 != "")
    {
     if ($actcnt == 0)
        {
         $query = $query." on";
        }
      else
        {
         $query = $query." and";
        }
     $query = $query." mc1.role_id = actor_role.role_id";
    }
 if ($actor2 != "")
    {
     ...
    }
 if ($actor3 != "")
    {
     ...
    }
/*
 *   Then join on MOVIE_CREDITS and PEOPLE
 */
 if ($actor1 != "")
    {
     $query = $query." inner join movie_credits mc1"
                   ." on mc1.movie_id = movies.movie_id"
                   ." inner join people actor1"
                   ." on actor1.people_id = mc1.people_id";
    }
 if ($actor2 != "")
    {
     ...
    }
 if ($actor3 != "")
    {
     ...
    }
}
/*
 *   We are done with the FROM clause; we are using the old 1=1
 *   trick to avoid checking each time whether it is the very
 *   first condition or not - the latter case requires an 'and'.
 */
$query = $query." where 1=1";
/*
 * Be VERY careful to add parameters in the same order they were
 * stored into the $params array
 */
if ($title != "")
```

```
        {
         $query = $query." and movies.movie_title like concat(?, '%')";
        }
     /* Director was specified ? */
     if ($director != "")
        {
         $query = $query."  and director_role.role_name = 'DIRECTOR'";
         if ($dfn != "")
            {
            /*
             * Use like instead of regular equality for the first name, it will
             * work with some abbreviations or initials.
             */
             $query = $query
                   ." and director.people_firstname like concat(?, '%')";
            }
         $query = $query." and director.people_name = ?";
        }
     if ($actor1.$actor2.$actor3 != "")
        {
         $query = $query."  and actor_role.role_name = 'ACTOR'";
         if ($actor1 != "")
            {
             ...
            }
         if ($actor2 != "")
            {
             ...
            }
         if ($actor3 != "")
            {
             ...
            }
        }
```

查询准备之后，就可以调用 prepare()方法了，接着对变量作绑定，这里的程序代码不太漂亮，因为有 1 到 9 个变量要绑定和处理，而每个变量都必须分开处理：

```
     /* create a prepared statement */
     if ($stmt = $mysqli->prepare($query)) {
        /*
         *  Bind parameters for markers
         *
         *  This is the messiest part.
         *  We can have anything between 1 and 9 parameters in all (all strings)
         */
         switch ($paramcnt)
            {
             case 1 :
                  $stmt->bind_param("s", $params[0]);
                  break;
             case ...
                      ...
                  break;
```

```
        case 9 :
            $stmt->bind_param("sssssssss", $params[0],
                                               $params[1],
                                                     $params[2],
                                                     $params[3],
                                                     $params[4],
                                                     $params[5],
                                                     $params[6],
                                                     $params[7],
                                                     $params[8]);
                break;
          default :
                break;
        }
```

就这样！我们已经完成了，可以执行此查询并显示结果了：

```
    /* execute query */
      $stmt->execute();
      /* fetch values */
      $stmt->bind_result($mt, $my);
      while ($row = $stmt->fetch())
         {
          printf ("<tr><TD>%s</TD><TD>%d</TD></TR>\n", $mt, $my);
         }
      /* close statement */
      $stmt->close();
     }
   else
     {
      printf("Error: %s\n", $mysqli->sqlstate);
     }
?>
</TABLE>
</CENTER>
```

很明显，此时的程序代码比只用单一查询时复杂许多。

先前我们建议尽量把工作推入 DBMS 端进行，现在却使用复杂的程序代码而尽量简化 SQL 语句，如何解释呢？必须执行的工作尽量放在 SQL 端进行，是很合理的，但查询中动辄做 3 次 join、其中有些 join 还根本无用，却是不合理的。

以更聪明的方式建立查询，可以更好地控制安全性、结果正确性和性能，而更为简单的解决方案可能牺牲某一方面的质量。

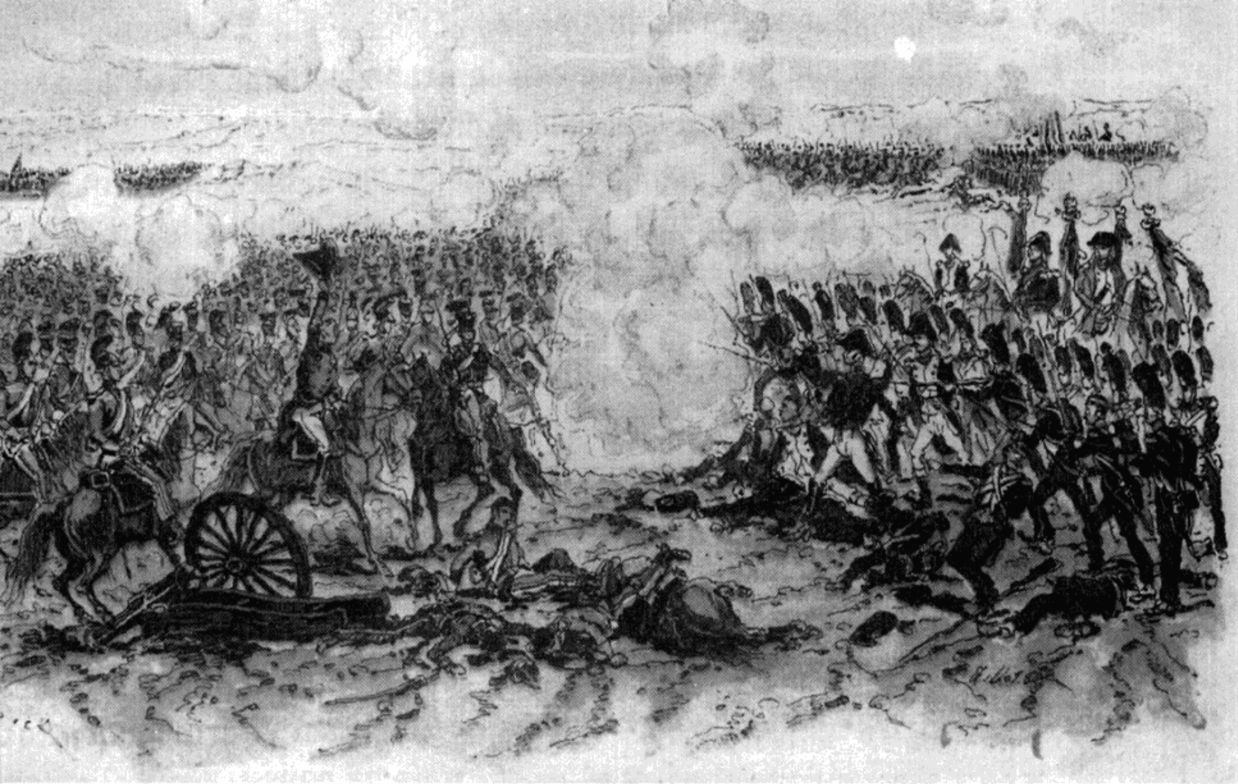
总结一下，在搜索条件数量不定的查询中，至少有 3 种极为常见的错误：

- 首先，我们经常把要与表中字段做比较的值，直接与要建立的语句串接，最终产生一个庞大的、硬编码的语句。甚至理应无法预测的情况，也被组织成了动态查询，而其中可以变化的只有几个常数。有些常数容易受到频繁变化的影响（例如实体 ID 易受变化影响，而日期格式和状态码不易受变化影响）。其实，用参数标记取代这些常数并不麻烦，集体语法取决于语言（例如 '?'），接着再把实际值与参数标记绑定。这会减少服务器的工作量，每次调用查询时无需重新分析语句，尤其是无需每次重新决定最佳执行计划（计划一定都相同）。另外，采用绑定变量的方式，没有用户能绕过为查询增加的约束，这意味着杜绝了严重的安全问题。

- 第二个错误，就是把可能有影响的所有 table 加到查询中。一个表出现在 `from` 子句中的原因，不是因为搜索条件会参照表中的数据。前面几章提到过，`from` 子句只应该包含源表，以及能把源表连接起来的其他表，而存在性测试应该由子查询解决（如第 6 章所述）——动态产生子查询不会比 `where` 子句中的正规条件更困难。

- 最严重的错误，就是“单一尺寸”哲学。在每一个通用的查询背后，常常隐藏着三到四类查询。通常，输入数据包括识别数据、状态值、或日期范围，它们将成为强大高效的条件，还是较弱的条件？或是介于两者之间呢？（有时可增加额外条件来缩小范围、强化较弱的条件。）以此为着眼点，试着更聪明地建立数个不同的查询（就如第 6 章的各种情况），是唯一合理的途径，尽管这看起来较为复杂。

总结：动态构造 SQL 语句时，运用越多智慧，SQL 语句就越高效。



第9章

多条战线：处理并发

Multiple Fronts: Tackling Concurrency

但将士们仍义无反顾地服从了将军的命令。

——约翰·弥尔顿（1608—1674）
《失乐园》，第一卷

当许多访问同一数据库的 Session 并发执行时，会遇到一些问题，这些问题进行单用户测试时是不会暴露的。资源竞争（Contention）时有发生，加锁时间无法预知……。本章讨论如何应付大量并发用户的情况。

大量并发用户会涉及多种问题。最明显的是更新（有时是读取）数据时的资源争用问题，以及随后产生的采用何种级别的锁的问题。这可看作“用户不受他人干扰地修改数据的权力”问题，因为不同用户在争用 CPU 处理能力、磁盘存取、内存空间和网络带宽等资源，所以用户数的增加会引起一些问题——突发性的用户数暴增可能是因为公司合并或收购引起的。

数据库引擎作为服务提供者
The Database Engine as a Service Provider

有人将 DBMS 比作聪明敬业的仆人，快速处理我们任何要求，准时送上我们需要的数据。现实的 DBMS 没有这么完美，比作餐厅中忙碌的服务生较为合适，如果你点菜较慢，服务生会告诉你“请先看一下菜单，我待会回来”然后消失。DBMS 是个服务提供者，更精确地说，是服务提供者的集合。服务是对数据的操作，或读取或更新，并且许多并发 Session 可能在同一时间请求服务。只有每个 Session 都高效执行时，DBMS 才是高效的。

索引的优点
The Virtues of Indexes

首先，对一个有 3 个字段的表进行基本测试。前两个字段为整数（填入从 1 到 50 000 的不同值），第一个字段为主键，第二个字段没有索引。第三个名为 `label` 的字段是字符串型，由长度为 30 到 50 的随机字符串组成。如果使用上述 1 到 50 000 的随机数作为查询条件来检索 label 字段，我们惊讶地发现，在能力一般的机器上，以下查询几乎总能立即返回结果：

```
select label
from test_table
where indexed_column = random value
```

或：

```
select label
from test_table
where unindexed_column = random value
```

怎么可能呢？没有索引时应该慢很多，不是吗？但其实，50,000 条记录的表太小了，而且字段数量也少（我们的例子只有 3 个字段），所以要扫描的字节数并不多，现在的机器可以快速完成扫描。所以，虽然一个是主键查询，另一个是全表扫描，但最终存取效率差异很小，以致察觉不到。

为了真正测试出索引的价值，我们持续执行查询一分钟，然后看看两例每秒各进行了多少条查询。结果完全在意料之中：有索引时，每秒可执行 5 000 次查询，没有索引时，每秒只能执行 25 次查询。只进行单用户测试，或许注意不到差异，但差异确实存在，而且差异非常大。

总结：响应时间不到一秒，仍然可能隐藏着重大的性能问题。不要相信单独某次测试。

有例为证
A Just-So Story

继续前例，来看看实际上发生了什么。假设表的键不是数值，而是字符串。开发期间就有人注意到，查询会出乎意料地返回错误的结果。开发者快速研究了一下，发现键同时包含大小写，于是急于修正他曾修改了的查询中的 `where` 子句，对键使用 `upper()` 函数——从而丧失了使用索引的可能性。开发者再次执行查询，结果正确，除了来自氪星球的超人之外，没有人注意到响应时间上的重大差异。似乎一切良好，于是把代码提交给生产环境。

于是，大量用户重复执行着上述查询。第 2 章曾指出，程序的循环中不应执行查询——无论是基于游标（cursor）的循环、还是传统的 for 循环或 while 循环。遗憾的是，人们经常这么做，所以查询可能被频繁执行，甚至在没有大量并发用户时也是如此。假如

每单位时间内以随机间隔执行若干次查询，会发生什么？以每分钟 500 次相对低的频率执行查询时，无论是否使用索引，似乎都一切正常。如图 9-1 所示，所有查询都在 0.2 秒内完成，没有人抱怨。

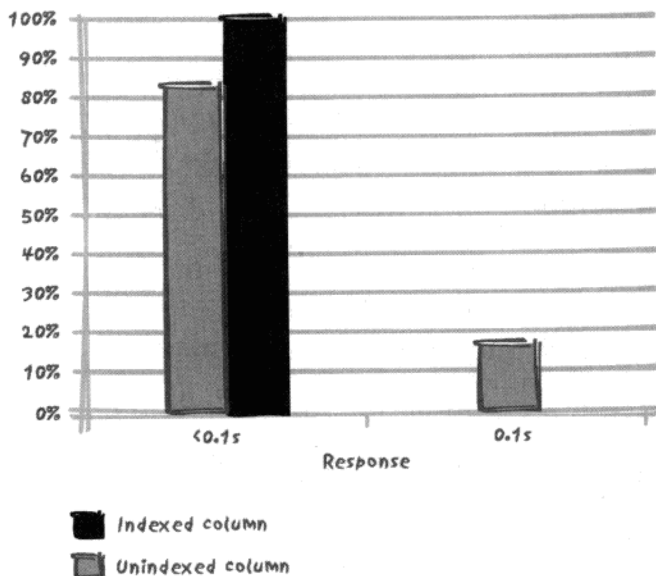


图 9-1：以“较低”频率对 50 000 的表进行简单查询的响应时间

若执行频率增加十倍，每分钟执行 5 000 次，将会出现如图 9-2 所示的情况：若没有为键建立索引，偶尔会比较慢，但只影响比例较少的查询。事实上，97%的情况下只需不到 0.3 秒就完成了。

但是，每分钟执行 5 000 次查询时，我们并没有意识到已濒临灾难。继续提高频率，每分钟执行 10 000 次查询时，就会出现如图 9-3 所示的情况——绝大部分查询都明显变慢，有的长达 4 秒。而另一个使用了索引的测试中，仍然是同样的执行频率，所有查询都不到 0.1 秒。

当然，原本执行得很快的查询，现在开始变慢，用户就会抱怨；即使没有发现异常的用户也会跟着抱怨起来。数据库好慢呀，难道不能调整吗？于是数据库管理员和系统管理员开始调整参数，又会撑上几个星期，最后“证据充足”地得出简单的结论：我们需要更强大的服务器？

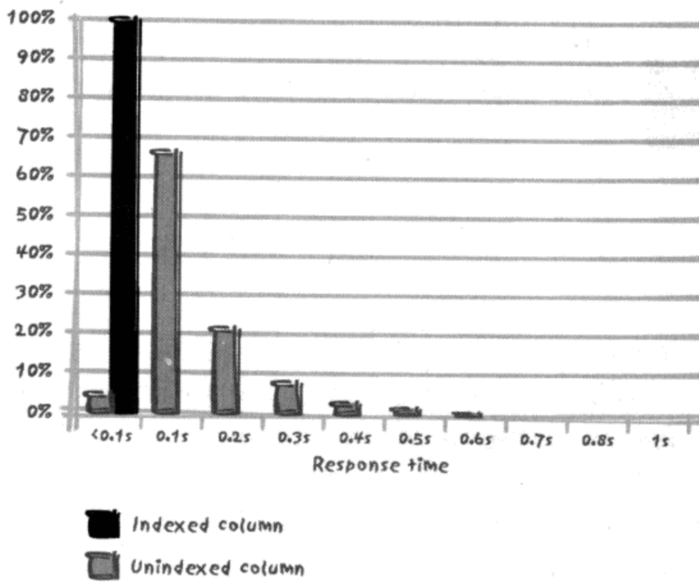


图 9-2：以“较高”频率对 50 000 的表进行简单查询的响应时间

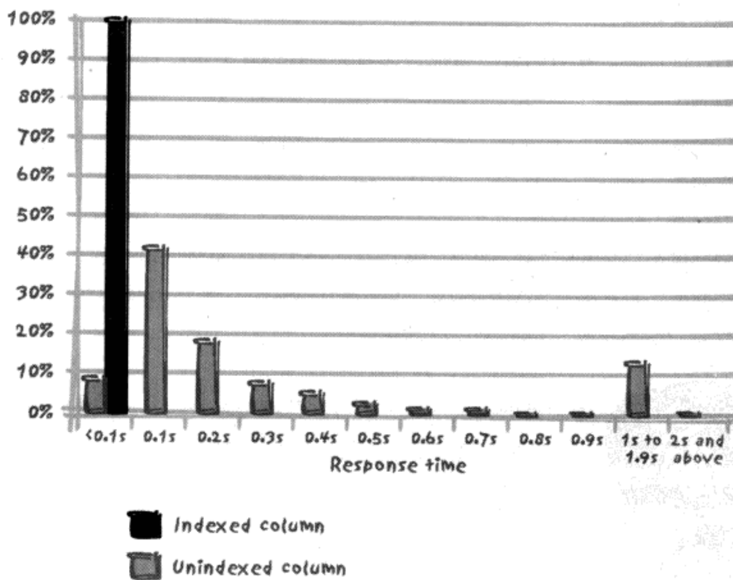


图 9-3：以“极高”频率对 50 000 的表进行简单查询的响应时间

总结：负载增加未必是造成性能问题的原因，它只不过使性能问题暴露出来了而已。此时，建议改善程序，而不是升级硬件。

排队

Get in Line

可把 DBMS 引擎比作邮局，雇佣很多雇员，为客户各式各样的请求（也就是我们的查询）提供服务。很明显，大邮局多个柜台同时服务，同时享受服务的不止一个客户。可以想象，爱喝咖啡的年轻职员速度快，爱喝茶的老职员速度慢。不过，影响速度的最大因素（尤其是在高峰时段）源自客户提出的请求，有人已准备好零钱要买集邮簿，有人却要详细询问寄包裹到国外的费率和报关手续……不一而足。最耗时的，当然是有人提出比较复杂的请求后，要花上好几分钟翻钱包找钱。还算庆幸，邮局中绝不会出现寄 20 封信的人占 20 个位置的情况，“类似情况”在数据库应用中却时有发生。一个人在邮局柜台是否能获得快速服务，有两个决定性要素：

- 职员的“性能”。在数据库应用中，这会受到数据库引擎、硬件、I/O 子系统等因素的影响。
- 请求本身的复杂度，以及请求阐述得是否清楚易懂，以便职员了解请求并快速而完整地回答。

在数据库的世界中，上述的“职员因素”变成了系统工程师和数据库管理员的领域，“第二个因素”变成了业务需求和开发活动。当你想从硬件和软件投资得到最大效益时，整个系统越复杂，不同团队之间的合作就越重要。

借助邮局的比喻，可以理解查询测试中发生的事：关键是客户数（或查询的执行频率）和查询所需平均时间之间的比例。只要足以让每个人找到“空柜台”，就不会有人抱怨；然而，一旦客户到达后不能立即得到服务，队伍就开始拉长，速度快的查询和速度慢的查询都需排队。

这就是“阈值效应（threshold effect）”，正如狄更斯的《大卫·科波菲尔》中一个角色所说的：

> 年收入 20 镑，年支出 19.6 镑，结果很幸福。年收入 20 镑，年支出 20.6 镑，结果很悲惨。

对数据库查询也是如此。下面并发执行上述两个查询（一个使用了加索引的字段，另一个使用索引），依然每秒钟执行 5 000 次，如图 9-4 所示。图 9-2 显示了没有采用并发执行的情况，而图 9-4 所示的并发执行情况和它大为不同，由于受并发执行的慢速查询的影响，快速查询也慢了下来。

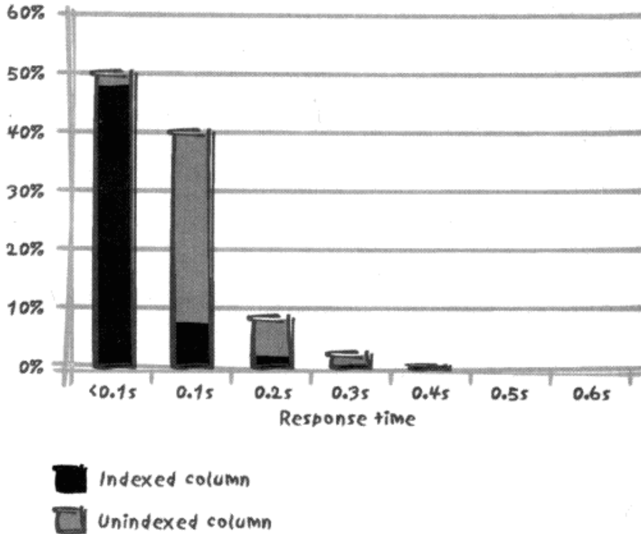


图 9-4：以“较高”频率，同时执行快速查询和慢速查询

总结：当 SQL 语句提交的速度比服务速度要快时，系统性能就会出现严重问题；所有查询都会受到影响，而不仅仅是慢速查询。

并发修改数据

Concurrent Data Changes

修改数据操作越频繁，维持良好性能的难度就越大。修改操作本质上比查询代价更高，

不仅牵涉取得数据，还涉及写回数据（当然 insert 操作只涉及写入），所以无论是 update、delete 或 insert，往往比查询要花更长时间。另外，加锁机制和资源争用会引起上述情况恶化。

加锁
Locking

多个用户想同时修改相同数据时——例如要预订航班的最后一个位置——DBMS 必须只为第一个提出请求的用户服务，而阻塞所有其他用户。对关键资源的访问做列表化处理是必要的，这一问题自多用户系统（multiuser systems）产生之日起就产生了。在数据库系统出现之前，文件（file）与记录（record）访问的列表化问题已经存在很久了。一个用户获得了资源上的锁，所有其他欲使用该资源的用户必须排队等候锁被释放，否则会报错。在许多方面，这与多个邮局客户使用同一台影印件类似——大家必须耐心等候轮到自己（或暂时离开过后再来）。

锁的粒度

要改变数据库内容时，最重要的问题之一，就是要决定使用锁的准确位置。锁会影响以下的任一项或全部：

- 整个数据库
- 存储被修改的表的那部分物理单元
- 要修改的表
- 包含目标数据的块或页
- 包含受影响数据的记录
- 记录中的字段

如你所知，到底有多少用户之间会产生相互影响，与锁的粒度有关。不同的 DBMS 支持不同的锁类型。支持何种粒度的锁，在“大型产品”（针对大型信息系统而设计）与“小型产品”（所追求的目标较少）之间有极大差异。

加锁的数据量小（细粒度锁），则多个并发进程就可能同时修改同一表中的数据，而不会造成阻塞。不同进程之间可以有一定重叠，不一定非要等到一个进程执行的事务结束——从硬件的角度看，这意味着处理器有更多工作要做，从而提高硬件资源利用率。细

粒度加锁的好处可以从图 9-5 清楚地看出，图中对比了多个并发 Session 更新表时的总吞吐量，一种采用表级锁（table-locking），一种采用行级锁（row-locking），两者使用的 DBMS 服务器相同。

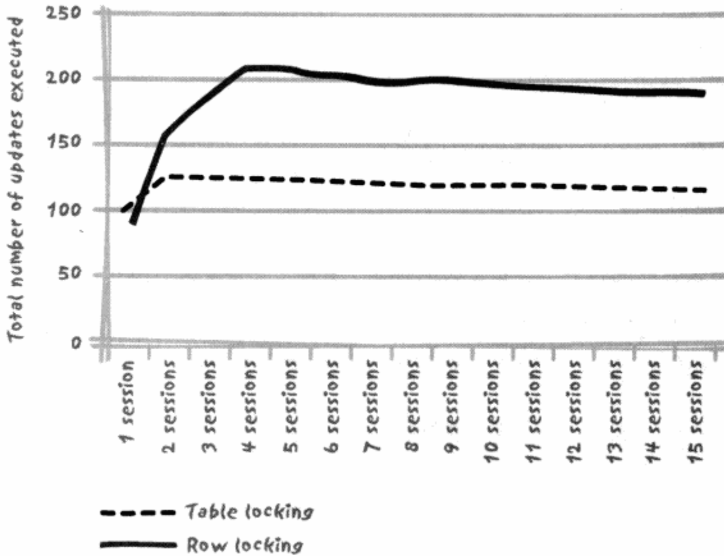


图 9-5：基于表级锁和行级锁 update 操作的性能对比

如图所示，使用表级锁时，增加到两个 Session 时吞吐量稍有增加，因为服务器——作为服务的提供者——的工作还未饱和。但两个 Session 已是单位时间能列表处理的最大 Session 数，之后曲线就是平的——事实上，因为更多 Session 要耗费更多的系统资源，所以曲线实际上是缓慢下降的，这反而对最终的数据更新工作不利。行级锁则不同，只要更新的不是同一行记录，都可以并发执行。最终，行级锁达到服务器饱和点比表级锁更晚，支持的并发更新数也更多。

如果由于加锁的影响，DBMS 变得非常缓慢，那么对付突然增加的访问活动的唯一办法（假设所有其他方法都已试过）就是购买更好的硬件了。“更好的硬件”能否提升性能也是有条件的，如果是锁造成的性能瓶颈，更多的处理器也于事无补，因为关键资源（critical resource）是数据存取。当然，较快的处理器可以加速执行，降低实际加锁时间，因此单位时间内能处理的更新数将增加。

加锁处理

加锁机制是实现 DBMS 时的重要部分，DBMS 的用户无需操心。我们在处理加锁问题时，要注意两个问题：

不要随便使用表级锁

> 不用说，当许多用户正对同一个表执行一些短的更新事务时，我们不应该对同一个表的数百万条记录进行更新的程序。

尽量缩短加锁时间

> 多个用户并发访问无法被共享的资源时，不只一个事务会出现速度问题。更新操作快的事务，也照样要等待速度慢的更新操作将锁释放。所以必须所有事务都快，否则性能依然不佳，正所谓“最差的一环决定锁链的强度”。

大多数的 `update` 和 `delete` 语句都包含 `where` 子句，通过改写 `where` 子句能加快 `select` 的执行，也能加快 `update` 和 `delete` 的执行。如果 `delete` 语句没有 `where` 子句（级删除整个表），使用 `truncate` 语句可能更合适，因为它清空表（或分区）更高效。

千万别忘了，索引也需维护，而更新加了索引的字段代价很高，所以我们必须在读取速度与更新速度之间权衡。对 `where` 子句有帮助的索引，可能为修改记录带来麻烦。至于 `insert` 语句，不少实际上是 `insert...select` 结构，`select` 的性能与 `insert` 的性能的关系是很明显的。

第 2 章中讲过，语句性能高，未必程序性能就高。当修改数据时，还应考虑一个问题：事务（transaction），也就是说，逻辑工作单元需要运行多长时间。大多数（也许不是全部）事务要求保留数据库特定部分的锁，所以不需要在事务中完成的工作，尤其是耗时的活动，应该排除在事务之外。事务的开始有时由发出的第一个数据操作语言（data manipulation language，DML）的语句隐含表示了，但事务的结束一定很明显，由 `commit` 或 `rollback` 明确标出。有了这些背景知识之后，一些实际做法只是常识而已。在事务内，应该：

- 尽可能避免 SQL 语句上的循环处理。

- 无论是以应用程序服务器或客户端执行，应尽量减少程序和数据库之间的交互次数，因为这增加了网络访问的开销。

- 充分利用 DBMS 提供的机制，使跨机器交互的次数降至最少（例如，运用存储程序或数组读取）。

- 把所有不重要的、不必须的 SQL 语句放在逻辑工作单元之外。例如，从表中读取错误信息即相当常见，尤其是在本地化的应用程序中。如果遇到错误，就应该先以 rollback 结束事务，然后再查询错误信息表，而不是反过来做，因为先结束事务会提早释放锁，有助于提高吞吐量。

像同时在 master 表和 slave 表中插入一条新记录这样简单的例子，就非常容易在加锁处理方面犯错误。例如，创建一条新的客户订单（在 master 表中），同时在购物篮（`order_detail` 表）中创建第一项订单项。问题常出在“由系统产生的订单标识”上。

首先应避免的错误，是把“下一个要使用的值”保存在表中。所有要更新这个表的并发进程都必须把它加锁，于是造成了整个应用的瓶颈。借助 DBMS 可解决此问题：可以用自增字段（auto-incremented colmun）——新插入记录的该字段的值自增 1；也可以用诸如 sequence 这种的数据库对象（database object）的下一个值，它本质上与自增字段类似，但无需显式引用现存表的字段。然后，直接把这些由系统产生的值用于每一笔新订单即可。当然，我们必须知道这个值，才能把购物篮里的项目与订单联系起来，换言之，必须把这个值插入 `order_detail` 表和 master 表中。

有些支持自增字段的 DBMS 产品，提供了系统变量（例如 Transact-SQL 的`@@IDENTITY`）或函数（例如 MySQL 的 `last_insert_id()`），以获取最新值。如果不使用 DBMS 提供的这些手段，在事务中自己实现类似功能，不仅徒劳无功，而且降低性能。使用自增字段对语句的执行顺序有一定要求，尤其在同时操作多个自增字段时。

不知何故，使用 sequence 的开发者倾向于先调用`<sequence name>.nextval`，并将获取的值保存在变量中备用。其实，可直接调用`<sequence name>.currval`（或 DB2 的 previous value for `<sequence name>`），正如其名称所暗示的，它的用途是返回特定 sequence 产生的最新值。所以，在多数情况下，无需用变量暂存当前值，甚至无需事先获取 sequence 值。在最糟的情况下，可以借助有些 DBMS 提供的扩充功能。例如 Oracle（和 PL/SQL）的用户可以用 `insert` 和 `update` 语句的 `returning ... into ...` 子句返回由系统产生的值，这避免了额外增加一次对服务器的调用。为了取得下一个 sequence 值而执行专门的语句，会增加一次与数据库的交互，如果此事务简单且经常执行，则开销的比例很大。

总结：事务型活动密集时，无需加锁的操作一定不要加锁，这一点至关重要。

加锁与提交

想要使加锁时间最短，就必须进行频繁的提交（commit）。提交是代价极高的处理，因为它意味着对持久存储（日志文件）的写操作，涉及物理 I/O 操作。如果每个逻辑工作单元完成后就提交改变，会增加大量额外开销。如图 9-6 所示，在一台空的测试机上，由单一用户进程进行频繁更新，每 1、2、3……12 条记录就提交一次。不同提交语句影响的记录数不同，性能当然有所变化，但反映的趋势是：相比低频提交而言，每个事务都做提交的方式要花两到三倍的时间。

对于批处理程序，并发控制不是问题，所以避免频繁提交才是明智的做法。不过，延迟很多改变的提交会延长加锁时间，且系统要为 undo 操作记录更多改变前数据映像（pre-change data image），这会消耗不少资源。如果处理因某种原因失败了，回滚（rolling back）也要花更长时间。在这个问题上，意见分为两派。一派赞成定期提交改变，缓和资源压力，并降低数据库修改失败时的工作量。坦白讲，另一派比较有说服力，他们认

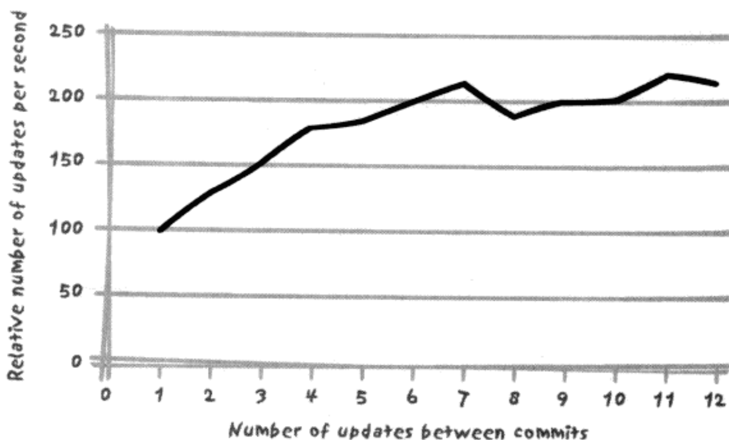


图 9-6：提交对性能的影响

为系统资源本应为业务过程服务，而不是业务过程为资源服务。如果降低提交频率能提高吞吐量——也能承受偶尔的失败，较快地对失败进行适当处理——那么降低提交频率就是有好处的。如果在 DBMS 提供“通过或中止（pass or break）”方式来避免产生 undo 数据的情况下，低频度提交就更具优势了。“完成时一次提交（commit-once-when-we're-done）”的方法认为：当有事情彻底失败时，“所有事情重做一次”通常比“修正部分完成的事情”简单。总之，这两种方案各有优缺点，最终选择常与操作限制（operational constraints）相关——甚至与政治（politics）相关。

无论如何，批处理程序的低频率提交方式可能会阻塞（block）用户交互，同样，用户交互也可能阻塞批处理程序。就算使用行级锁，某些数据库系统的“锁升级（lock escalation）”机制（即粗粒度锁代替细粒度锁），也可能导致系统挂起（hung）。甚至在没有锁升级的情况下，一个未提交的更改，也可能阻塞大量更新操作。显然：并发和批处理并不是幸福的组合，必须根据事务是交互式的还是批处理的，以不同方式加以解决。

总结：并发用户数越多，提交的间隔就应该越短。

与表级锁相比，行级锁能产生更佳的吞吐量。然而，和表级锁一样，行级锁的性能曲线很快到达极限（性能无法再改善），之后曲线相当平坦。所有产品都是如此吗？实际上并非如此，如图 9-7 所示。

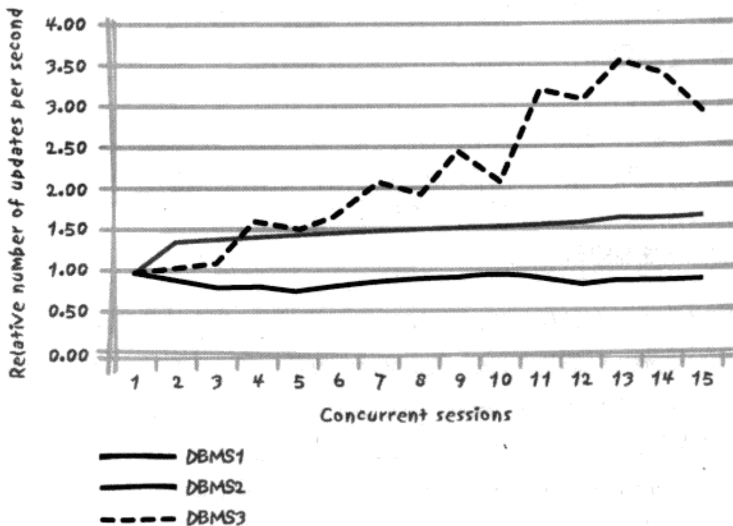


图 9-7：三种数据库系统采用行级锁时的并发性表现

当并发性增强时，为了比较不同数据库系统的表现，并排除具体例子的影响，我对大型表执行了两组更新：第一组，使用主键上的条件做“快速更新”；第二组，使用无索引字段上的条件做“慢速更新”。这些更新操作以不同数量的并发数重复执行，每次记录完成的更新操作的数量。

在使用“快速更新”时，不同数据库产品的吞吐量都没有受到 Session 数的很大影响（译注 1），因为对硬件资源的利用已达到饱和。但是，我们关心的是以并行方式执行大量 Session 是否有价值。并发性增强可以提升速度吗？这个问题直接影响“选择具有少量较快处理器的服务器，还是选择具有大量较慢处理器的服务器”。

译注 1：请注意，这句话是理解图 9-7 的前提。

图 9-7 显示，当 Session 数增加时，“慢速更新”与“快速更新”可完成的更新操作数量之间的比例（the ratio of the number of slow updates to the number of fast updates）。DBMS1 的表现源于两个原因：

- “慢速更新”比“快速更新”慢得不是很多（Slow updates are not that slow relatively to fast ones）。
- 并发 Session 数从 1 个 增加到 3 个时曲线陡峭下降，说明慢速更新受到了并发数增加的影响。

然而，最值得关注的还不是 DBMS1，而是 DBMS3，因为虽然所有系统都使用了行级锁，但惟有 DBMS3 表现卓越，随着越来越多的并发 Session 加入，比例缓慢但显著地提高了。这份测试报告会对我们进行硬件选择和架构设计产生重大影响，例如 DBMS1 和 DBMS2 更能从较快的处理器获益，而不是更多处理器，以软件的观点来看，基于少量 Session 建立“查询池（query pooling）”机制也对它们比较适合。另一方面，类似 DBMS3 的产品能从更多相同速度的处理器获益，在某种程度上，较大数量的并发 Session 数比较合适。

当如何解释 DBMS3 和其他产品之间的这种差异呢？最主要是两个因素：

硬件资源的饱和

硬件资源的利用已达饱和，这是例子中 DBMS1 情况出现的重要原因：在特定硬件情况下，吞吐量已达到最优，无法再进一步改善了。

资源竞争

三种产品的测试使用了相同粒度的锁（行级锁），所执行和提交的语句也完全相同。事实上，除了锁之外，还有其他因素制约着并行 Session 数。以机械学作为比喻， DBMS1 和 DBMS2 的情况中摩擦力比 DBMS3 中的大，这种“摩擦力”即资源竞争（contention）。

总结：并发性取决于完整性保护机制（integrity protection mechanisms），其中包括锁和其他控制，后者因产品不同而不同。

资源竞争

Contention

表中的记录，不是唯一不能被共享的资源。例如，有值更新时，undo 数据就必须被保存，以备回滚时使用；当两个或多个进程尝试将 undo 数据写入相同位置时，就会产生资源竞争，尽管此时进程操作的表和记录都不同。这种情况需要对事件进行顺序化（serialization）控制。同样，提交修改会涉及写日志文件，或许还需先写入内存缓冲区最终一起写入日志文件，此时也要避免各进程彼此覆盖数据。

上面举了竞争的例子。加锁比竞争更能体现 DBMS 架构的特点，除了尽量缩短加锁时间之外，我们基本别无选择。相反，竞争与具体实现有关，可以通过不同手段来协调竞争——有时由系统工程师进行，例如更精细地设置事务日志文件（transaction log file）；有时由数据库管理员进行，调整数据库参数和存储选项；有时由开发者进行，通过编程解决问题。下面，通过一个最典型的案例，说明如何通过编程解决竞争问题，此案例即同时进行多个并发的插入操作。

插入与竞争

我们以 14 个字段、具有两个唯一性索引的表为例。主键为系统产生的编号，而真正的键是由短字符串和日期值组成的复合键，必须满足唯一性约束（并通过唯一性索引强制实施）。下面，我们以并发 Session 数渐增的方式，执行一系列插入操作。

如图 9-8 所示，我们采用了行级锁，但随着进程数的增加，单位时间插入的记录数并没有增加。图中显示，采用不同进程数运行 10 次，每次运行 1 分钟，其执行结果的中间值、最小值和最大值的测试结果上下起伏——但最佳结果是在 4 个并发进程时获得的（这会受到机器的处理器数量的影响）。

可以断定硬件资源已饱和了吗？是的。但真正的问题是“是否可以更好地利用这些资

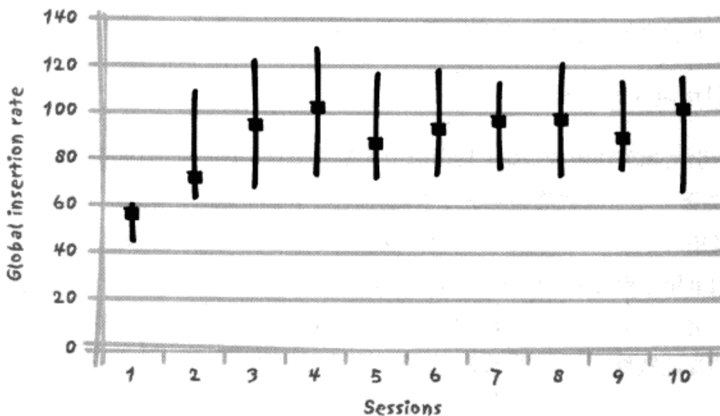


图 9-8：通过并发 session 对常规表进行 insert 操作

源”。此时，对加锁难有太多调整，因为不存在两个进程存取相同记录的情况。然而，当尝试访问数据容器（data container）时，会发生资源竞争——竞争会发生在两个地方："表"和"索引"。在系统层次（system level）也可能发生资源竞争，但它们通常由数据库层次（database level）的竞争引起。资源竞争消耗 CPU，因为处理竞争问题要执行一些代码，在等待另一个处理器上执行的进程释放资源时，还可能出现"活动等待（active wait）"或"闲置循环（idle loop）"。是否可以降低竞争，使插入操作获得更多 CPU 周期呢？

产生图 9-8 的例子运行在 Oracle 上，Oracle 等产品为减少资源竞争提供了丰富的手段。以数据库为中心来解决竞争问题基本上有以下三类解决方案：

- DBA 解决方案
- 架构解决方案
- 开发解决方案

接下来的几节将评价每种类型的解决方案。

DBA 解决方案

数据库管理员通常缺乏业务流程知识。我们所谓的"DBA 解决方案"，是针对容器本身的改变，它们与应用程序无关（虽然要做到绝对与应用程序无关几乎是不可能的，但较精确地说，除了要改善的插入操作之外，对其他操作的影响很小）。

Oracle DBA 通过两种途径改善竞争问题，并且使抱怨降至最少：

事务空间（Transaction space）

表或索引由物理块组成，每个物理块中为事务条目（transaction entry）预留了空间。DBA 的第一个手段就是调整事务条目所占空间的大小。每项事务条目可理解为一个低级锁（low-level lock），当多个 session 竞争同一个物理块的写权限时，事务条目的争用是资源冲突的重要原因。DBA 可以通过增加分配给事务条目的空间来缓解冲突，对应用其余部分的唯一影响是表或索引的数据可用的空间变少了，于是，同样的数据量所占的物理块增多了，完整扫描操作和索引搜索操作（影响较轻）必须访问更多的物理块。

可用列表（Free lists）

另外，可以让 `insert` 操作分散到不同的物理块，有时可以借助存储管理拥有的能力做到这一点。例如，Oracle 为每个表维护一到多个“块的列表”，默认情况下只有一个列表，但多个列表可支持 `insert` 操作循环（round-robin）进行。但这种解决方案不像增加“事务空间”那样中立，因为数据分区在提高插入性能的同时，会对查询性能有较大影响。

架构解决方案

架构解决方案，即借助 DBMS 工具，修改数据的物理部署（physical disposition）。这样做的影响比较复杂，可能对其他处理不利。最明显的三个架构解决方案是：

分区

如果我们的目的是将更新活动分布到整个表，那么范围分区（Range partitioning）反而适得其反，除非我们让不同进程分别负责某月数据的 insert，而每个进程分别对应一个分区，但这不是我们当前的例子所讨论的情况。相反，哈希分区（Hash partitioning）适合上述情况，但如果哈希值是根据系统生产键（序列号）计算出来的，连续值又会不恰当地被分配到不同分区中，很不幸的是，我们就会受限于强制实行约束的索引，于是，不得不借助表级锁解决此问题。此外，分区解决方案会是数据分散，这对其他查询的性能可能造成影响。

逆序索引（Reverse index）

如第 3 章所述，将组成索引键的字节顺序反转，可把键原本紧邻的项目分散到不同的叶节点中，这是尽量避免索引竞争的良好方法（虽然表竞争并无改善）。这样做的缺点是，不利于索引上的范围扫描，会严重影响性能。

索引组织表（Index organized table）

以索引方式组织表，就少了一个“竞争源（sources of contention）”。这样做免去了从一个竞争点（表区块）移向另一个竞争点（索引区块）的问题，现在只有一个“竞争源”了。

开发解决方案

开发解决方案完全掌控在开发者手中，而且不需要改变数据库的物理结构。下面是两个例子：

调节并发数

前面调整并发进程数的尝试明确表明，4 个并发 session 性能最佳，继续增加 session 数量并无帮助。正如可由 4 人完美完成的任务交由 10 人完成没有好处一样，反而使协调更复杂。最佳 session 数量是和硬件相关的，图 9-8 显示，再增加额外的 Session 毫无作用。因此，去除多余的 session，能减轻系统压力。

不使用系统产生值

我们真的需要连续值作为代理键吗？未必。如果处理范围，则连续值有用，因为能支持诸如“ >”和“`between`”等操作符的使用。但仅需要能在其他表中作外键的唯一性 ID，则可考虑其他方案——即只使用随机数值，遇到已使用过的值重新产生即可。

结果

在 10 个并发 Session 的情况下，分别运用上述改善竞争的方法，最终插入比率如图 9-9 所示。跟之前一样，每个测试执行 10 次。

结果的波动较大。我们无法判断在这里表现良好的技术，是否在其他情况下也表现良好；也无法推断在此令人失望的技术，未来是否有出乎意料的表现。不过，结果仍然很值得研究。

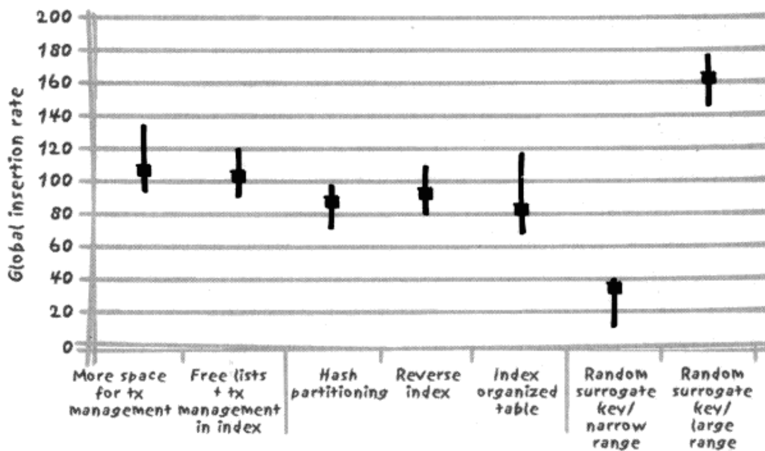


图 9-9：限制 insert 操作之间竞争的技术

首先，DBA 技术带来了积极效果，但不是特别显著。而在这个例子中，架构解决方案效果很差。值得一提的是，我们的两个索引会强制实施约束，这会限制索引选项的数量。因此，当多数竞争发生在索引时，有些技术倒能改善表层次的竞争，索引组织表就是如此，其中表的竞争会因表不复存在而消除；不幸的是，因为现在有更多数据存储在索引内，所以索引竞争会增加，并抵销了表不复存在的好处。在这种情况下，CPU 成了最需要的系统资源，于是让所有需要更多 CPU 资源的技术处于不利地位——例如计算哈希值或逆序索引键。

最后，最糟和最好的测试结果都出自采用“随机值”之时。在最糟的情况下，会产生 1 到特定数值之间的（整数）随机值，特定数值大约是预期在测试中插入记录数的两倍。所以，多次产生重要数字值可能违反主键约束，须再次产生新的随机数字，这浪费了时间，消耗了额外资源。此外，违反主键约束必须到插入主键值之时才可发现，而且索引保存的是物理地址，此时记录已被插入表中，所以需要 undo，这又是额外成本。

在最佳情况下，产生的随机数的范围比最糟情况大一百倍时，性能才显著改善。但由于 10 个并发 Session 并不比 4 个更高效，如果只用 4 个 Session，会是什么结果？图 9-10 提供了答案。

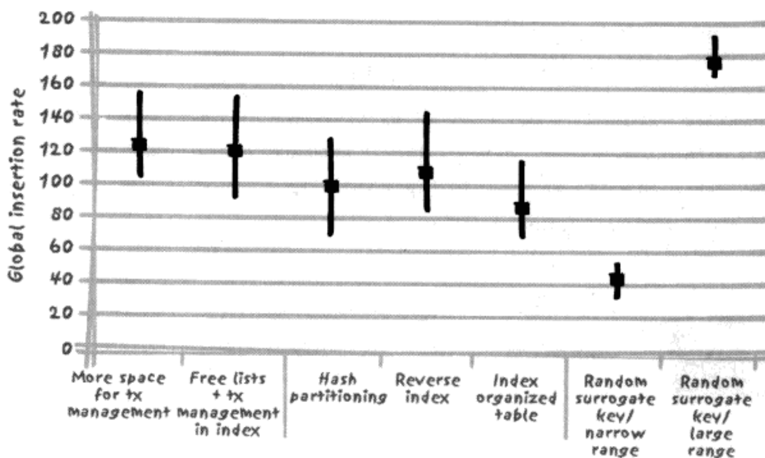


图 9-10：session 数较少时竞争限制技术的表现

非常有趣的是，所有技术的效果都明显提升了，不过从吞吐量的改善程度来看，它们的排名仍完全相同（相对性能基本类似）。比较图 9-9 和 9-10 结果的不同能够说明：

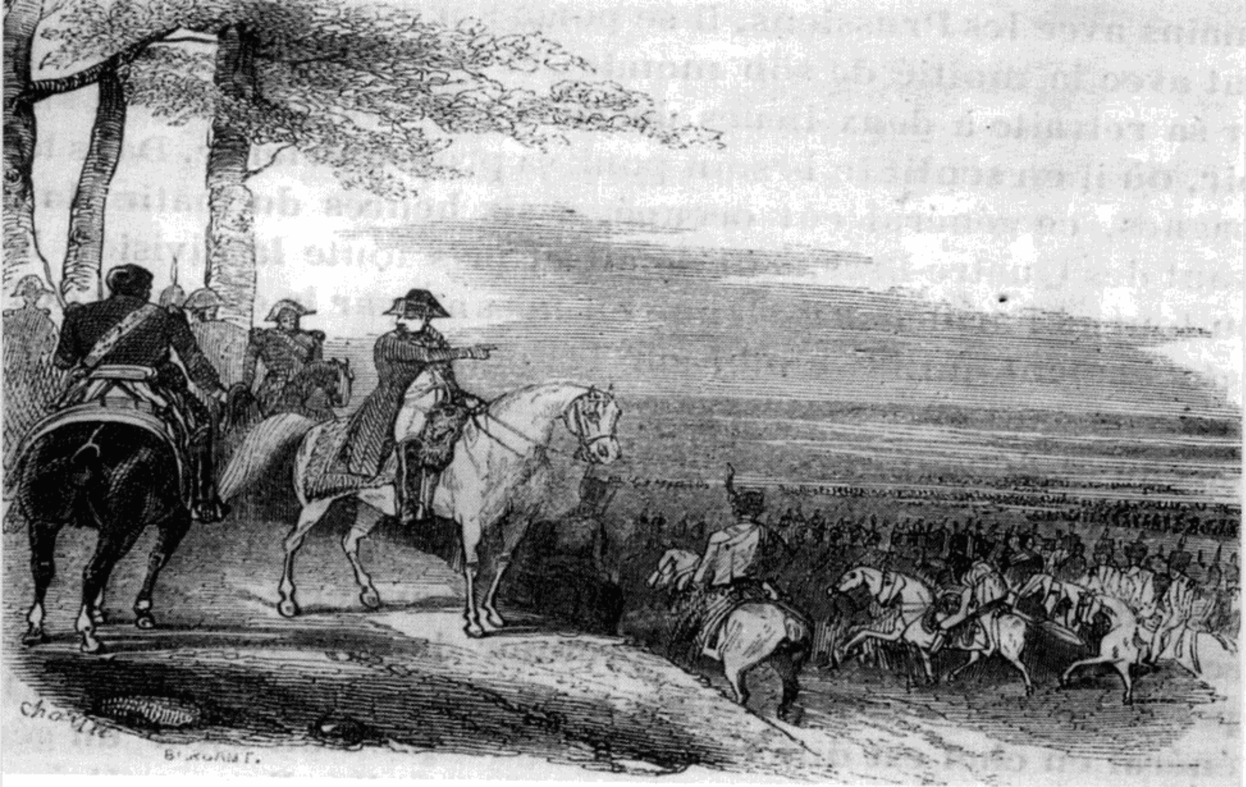
- 此案例的瓶颈是主键索引。本应较好解决表级资源冲突的技术（哈希分区、索引组织表）而没有达到效果；事实上，本例的索引组织表比配以主键索引的正规表的性能更糟。相反，能降低表和索引竞争的技术（例如为事务管理分配更多空间）和改善索引级资源冲突的技术（逆序索引、随机代理键），都提升了性能。

- 10 个 Session 与 4 个 Session 的差异说明，有些技术需要已处在饱和状态的机器提供（珍贵）的 CPU 资源，所以不能带来性能的改善。

- 避免竞争的最佳方法，是避免使用顺序产生的代理键！不要考虑采用各种度量结果能获得多少性能，而是要考虑“顺序键引起的主键索引竞争”造成了多少性能损失（不慎造成的？）。就是因为主键索引上的竞争，`insert` 操作的速度从每秒钟 180 个降至每秒钟 100 个，就是说几乎降低了一半！此教训很清楚：若不必要（例如该表未被其他表参照，或该表本来的主键并不长），最好不要采用自增字段。

何时推荐使用随机产生的代理键呢？在很大区间内产生代理键，还是在很小区间内产生代理键？这二者之间的性能差异说明：如果预期数据量大于整个区间的百分之一，就很危险，而且最终并不高效。产生 1 到 20 亿（这是常见的整数值范围）之间的随机整数值，对于大型表是有风险的；很不幸，执行大量插入操作的表的数据量往往增长相当快。然而，如果你的系统支持"很大很大的整数"，随机产生的代理键就是很好的解决方案（如果你真的需要代理键）。

总结：与加锁不同，数据库竞争可以改善。架构师、开发者、以及管理员都可以从各自的角度改善竞争。



第 10 章

集中兵力：应付大数据量

Assembly of Forces: Coping with Large Volumes of Data

这时巨人 Lubance 出现了，他作战骁勇，大挫了敌军的军心。

——托马斯 · 马洛礼爵士（? —1471）
《亚瑟王之死》，第五卷

本章要讨论数据量增加时面对的特殊挑战。这包括如何高效搜索庞大的表，如何避免数据量稍有增长就可能出现的性能下降。首先，我们讨论一般情况下，数据量增加及 SQL 查询大量记录的影响。接着，我们讨论在数据仓库（data warehousing）和决策支持系统（decision-support systems）的特殊环境下的情况。

增长的数据量

Increasing Volumes

有些应用面临"要处理的数据量大幅度增长"的问题。尤其是，出于管理或业务分析目的，应用需要在线保存数月、甚至数年不经常被用到的数据。这些应用会面临危机的考验——批处理程序的耗时慢慢超过了它们可用的时间范围，干扰了人们的其他正常活动。

项目的不同阶段，数据量经常会改变，如图 10-1 所示。一开始，除了相对少量的参考数据被加入数据库之外，几乎没有任何东西。后来，新系统取代旧系统，需要花大力气将旧系统的数据移植到新系统中。首先，由于重新规划了新系统，所以从旧系统转换到新系统会伴随一些困难。最后期限（deadlines）的压力常使非关键任务被推迟，这其中首当其冲的就是旧数据的移植工作，这些工作常在新系统已上线及"初期问题（teething problem）"已解决后才开始进行。其次，旧系统的功能通常比新系统逊色很多（否则，新项目的成本从成本效益角度难以接受），所以旧系统的数据量比新系统要少得多，几个月的旧数据最多与几周的新数据量相当。

同时，新系统的工作数据在不断增加。

在达到目标数据量（target volume）——数据库以"巡航速度（cruising speed）"运行时能处理的数据量——之前，第一次严重的性能问题通常会出现。当数据量很少或中等时，从最终用户角度看不出糟糕的查询和糟糕的算法的影响。硬件本身的处理能力会隐藏巨大错误，完整扫描几十万条记录的表也不到一秒时间。我们可能严重地滥用硬件能力来弥补程序的错误——直到数据量变得相当可观时，真正的错误才被发现。

当项目第一次出现性能危机时，通常会进行"专家调优（expert tuning）"，增加几个

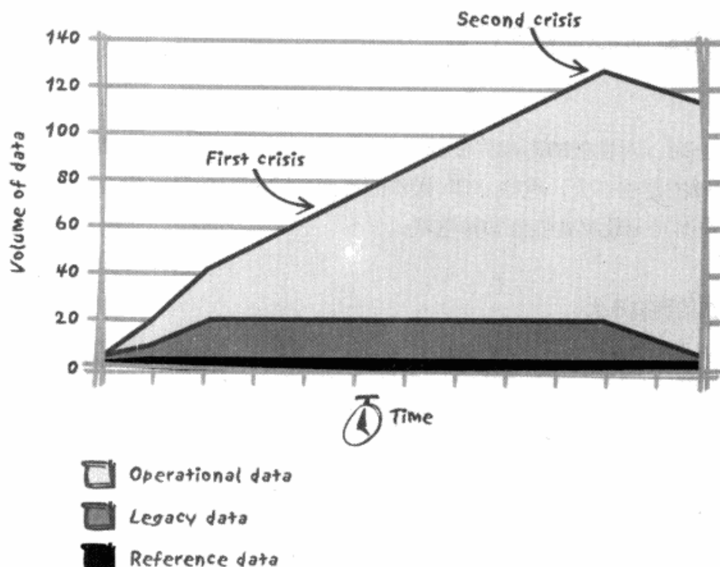


图 10-1：新系统的数据量如何增长

本应一开始就有的索引。接下来的系统可能不太稳定，直到数据量达到目标量。通常会有两个“目标量”：计划目标量（系统设计应管理的数据量，通常会被高估）和实际目标量(系统实际处理的数据量,通常稍有超出,因为遗留数据在项目最后会被考虑进来)。第二个性能危机（而且是较严重的危机）通常会随着实际目标量的到来而出现。当系统投入生产环境，架构设计的弱点被重新评估，一些关键处理被积极重写，系统最后达到“巡航速度”。随着业务的自然增长——平稳成长和指数成长都有可能，数据量也会随之增长。

上面描述了一个新的数据库应用生命周期的头几个月，颇有几分讽刺意味，但这比“应有情况”更接近现实，因为导致这些问题的简单错误还是时常出现。压力、没时间进行充分测试、含糊不清的规格说明……无论多努力工作，还是会有错误。不过，造成无法挽回失败的，是数据库设计错误和架构选择错误——这两个主题密切相关，而且是系统的基础。如果地基不够坚固，就必须把整栋建筑物推倒重建，而其他错误需要某种程度上的检修。当然，不是所有危险都必定发生，编程时我们必须预先考虑数据量的增加，而且面对逐渐增加的数据量时，必须尽快找出使性能快速恶化的查询并改写它们。

操作对数据量增加的敏感程度

Sensitivity of Operations to Volume Increases

当数据量增加时，对性能的影响程度因 SQL 操作不同而不同。有些 SQL 操作的性能受数据量增加的影响不大，有些 SQL 操作的性能则随着数据量增加而线性下降，还有些 SQL 操作面对大数据量性能非常糟糕。

受数据量增加的影响不大

通常，基于主键的搜索受数据量的影响不大，无论是 1 000 笔还在 1 000 000 笔不会有显著差异。常见的 B 树索引，其结构趋于扁平，效率很高，基于主键搜索单笔记录的性能不会受到表大小的影响。

但是，除了基于主键搜索单笔记录，还会查询大量记录的时候。看下面两个 Oracle 的例子，都会产生范围扫描：

```
SQL> declare
  2     n_id            number;
  3     cursor c is select customer_id
  4                 from orders
  5                 where order_id between 10000 and 20000;
  6  begin
  7     open c;
  8     loop
  9        fetch c into n_id;
 10        exit when c%notfound;
 11     end loop;
 12     close c;
 13  end;
 14  /

PL/SQL procedure successfully completed.

Elapsed: 00:00:00.27

SQL> declare
  2      n_id     number;
  3    begin
  4      for i in 10000 .. 20000
  5      loop
  6         select customer_id
  7         into n_id
  8         from orders
  9         where order_id = i;
 10     end loop;
 11  end;
 12  /
```

```
PL/SQL procedure successfully completed.

Elapsed: 00:00:00.63
```

第一个例子中用到了 cursor，进行显式的范围扫描（explicit range scan），它的速度比迭代处理单笔记录快两倍。为什么？其中有多个技术上的原因——“软解析”（soft parsing）是原因之一，DBMS 引擎无需重复解析每个语句——但最重要的原因在于第一个例子只要向下搜索 B 树一次，接着扫描“键的有序列表”即可找出记录地址并访问表；但在第二个例子中，每次搜索 order_id 字段时，都要向下访问 B 树。处理大量记录的最高效方法，就是不要反复，并应用单一记录处理。

受到数据量增加的线性影响

最终用户通常认为，要返回的记录数量为原来的两倍，则查询就会花更多的时间。但实际上，许多 SQL 操作花了两倍的执行时间，用户却并没有意识到，就像通过全表扫描逐一返回记录时发生的情况一样。考虑聚合函数（aggregate function），例如计算 max() 一定只返回单一记录，但 DBMS 所操作的记录数可能很多，不过最终用户只看到单一记录被返回，所以他们会抱怨性能随着时间而下降了。确保情况不会变糟的唯一方法，就是使用另一个条件（例如日期范围），对要处理的记录数量设定上限。设定上限能让数据量保持在控制范围内。对于 max() 的例子，可以查询给定日期之后的最大值，而不是所有值中的最大值。增加查询条件不是单纯的技术问题，还依赖于业务需求，但限定查询范围作为可选手段，值得在设计时进行讨论。

受到数据量增加的非线性影响

数据量增加时，排序操作所受的影响比扫描操作还大，因为排序是复杂操作，一般而言需要多遍处理。对 100 条随机顺序的记录排序，所需成本并不是 10 条数据的 10 倍，而是大约 20 倍。排序 1000 条记录所需成本，比排序 10 条记录平均高 300 倍。

然而，在实际中，记录很少是随机存储的，即使没有使用“聚集索引（clustering index）”（第 5 章）这类技术也是如此。DBMS 有时使用有序索引，以预期的顺序读取记录，而不是先读取后排序——读取较大的有序集合时，性能降低一点并不奇怪。但注意，排序性能降低常是间歇发生的，因为较小型的排序将全部在内存中执行，而较大型的排序（涉及多个有序子集的合并）则需要将有序子集临时存储到慢得多的硬盘中。所以，通过调整分配给排序的内存数量来改善排序密集型操作（sort-heavy operation）的性能，是常见且有效的调优技巧。

作为例子，图 10-2 显示了表增大时，查询大量记录时的读取速率（单位时间读取的记录数）。这个测试使用非常简单的 orders 表，定义如下：

```
order_id          bigint(20)  (primary key )
customer_id       bigint(20)
order_date        datetime
order_shipping    char(1)
order_comment     varchar(50)
```

用作测试的查询操作共 5 种。

第一种，基于主键的简单查询：

```
select order_date
from orders
where order_id = ?
```

第二种，简单排序：

```
select customer_id
from orders
order by order_date
```

第三种，涉及 group 操作：

```
select customer_id, count(*)
from orders
group by customer_id
having count(*) > 3
```

第四种，对无索引的字段计算最大值：

```
select max(order_date)
from orders
```

最后一种，查询订单数量最多的 5 个客户:

```
select customer_id
from (select customer_id, count(*)
      from orders
```

```
        group by customer_id
        order by 2 desc) as sorted_customers
  limit 5
```

（代码中的 limit 5，在 SQL Server 中换成 select top 5 并放在语句开头，在 Oracle 中换成 where rownum <= 5。）

作为例子的表中，记录数大约在 8 000 到 1 000 000 行之间，而不同的 customer_id 值大约 3000 个。如图 10-2 所示，执行主键搜索时，处理 8000 到 1000000 条记录差不多。在记录数较多时，性能似乎有轻微下降，但查询速度仍很快，几乎不会察觉；但此时排序性能会变差。当记录数从 8 000 增加到 1 000 000 时，排序的性能会减少 40%（以单位时间返回的记录衡量，跟实际读取的记录数无关）。

一些查询需要访问大量数据返回较少几条记录，如果返回的记录是聚合记录（aggregated row），则性能较一般情况明显要低，通常会引起用户抱怨。注意，DBMS 表现并不是很糟：性能降低与记录数增加是成比例的，甚至对两个需要排序的查询（图 10-2 中标为"Group by"和"Top"的查询）也是如此。但最终用户只看到返回相同数量数据的速度慢多了。

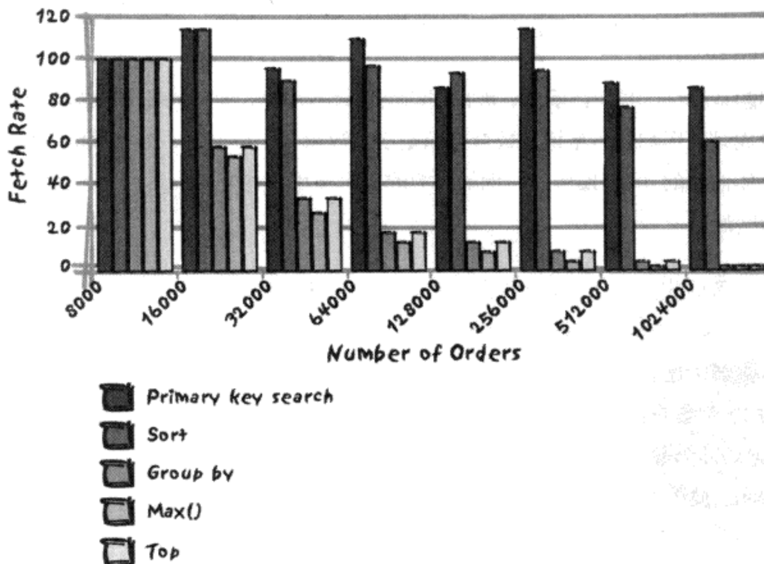


图 10-2：表的数据量增加时简单查询的性能表现

总结：不同的数据库操作对数据量增加的敏感程度不同。要预先考虑查询对不同数据量的执行方式。

综合考虑

数据量增加时，预估查询行为的主要困难在于：隐藏在查询背后的对数据量的高敏感度。以查询商品“当前价格”为例，如果使用子查询找到价格更改的所有时间，然后通过 max() 确定最近一次更改，则此查询对数据量是高度敏感的。随着价格更改的历史记录不断增多，子查询的性能会缓慢降低，外层查询会受影响。如果使用“非关联子查询（uncorrelated subquery）”，则数据量增加对性能的影响小得多，因为子查询只执行一次，而“关联子查询（correlated subquery）”每次都要执行。这种性能降低，对单项目操作几乎无法察觉，但在批处理程序中会较为明显。

注意

贸易系统中，跟踪订单当前状态就完全不同了，因为 max()只用于少数可能状态。即使订单的数量加倍，max()操作的记录数依然和一张订单时相同。

另一个问题是排序。我们已看到，被排序的记录数增加时，会导致显著的性能降低。事实上，造成较大影响的不是记录数量，而是字节数量——换言之，被排序的总数据量。所以和具体信息的 join 操作——例如晦涩的代码与用户友好的标签——应该延后到查询的最后阶段。

我们来看个简单的例子，可说明有些 join 为何应延迟到查询的最后阶段。要查询过去一年内前 10 大客户的名称和地址，需要连接 orders 和 order_details 表以获得每个客户的订单金额，还要连接 customers 表以取得每个客户的详细数据。如果我们只想查询最大的 10 个客户，就必须取得过去一年中有购买活动的客户，并依金额做降序排序，前 10 条记录即为结果。如果一开始就连接所有信息，过去一年所有客户的名称和地址都要被排序，操作如此大量的数据是不必要的。我们必须做的，就是对尽量少的数据进

行排序——就是客户ID和金额，然后将留下的 10 个 customer_id 与 customers 表连接，以返回所需信息。换言之，我们不应如此编写查询：

```
select *
from (select c.customer_name,
             c.customer_address,
             c.customer_postal_code,
             c.customer_state,
             c.customer_country
             sum(d.amount)
      from customers c,
           orders_o,
           order_detail d
      where c.customer_id = o.customer_id
        and o.order_date >= some date expression
        and o.order_id = d.order_id
      group by c.customer_name,
               c.customer_address,
               c.customer_postal_code,
               c.customer_state,
               c.customer_country
       order by 6 desc) as A
limit 10
```

查询应该这样编写：

```
select c.customer_name,
       c.customer_address,
       c.customer_postal_code,
       c.customer_state,
       c.customer_country
       b.amount
from (select a.customer_id,
             a.amount
      from (select o.customer_id,
                   sum(d.amount) as amount
            from orders_o,
                 order_detail d
            where o.order_date >= some date expression
              and o.order_id = d.order_id
            group by o.customer_id
            order by 2 desc) as a
      limit 10) as b,
      customers c
where c.customer_id = b.customer_id
order by b.amount desc
```

第二个排序是为了避免连接修改内层子查询产生的记录的顺序（切记，关系理论对排序一无所知，DBMS 引擎借助优化器找到最有效的处理连接的方式）。现在有两个排序，而不是一个，但内层排序是对“小范围”记录进行操作，而外层排序只对 10 个记录进行操作。

第 4 章中讲过：我们必须限制 SQL 查询中非关系层（non-relational layer）的“厚度”。非功能层的厚度与操作数量、操作复杂性、操作的数据量等因素有关。由于排序（sort）和各种限制（limit）都是非关系层操作，所以优化器可能不会改写这个“先过滤客户 ID 再连接”的查询。虽然仔细阅读这两个查询后，就知道它们返回相同结果（数学原理保证了这一点），但优化器从不冒险，DBMS 也无法承受返回错误结果的改写，尤其是在对语义完全无知的情况下。因此，对于我们的例子，优化器会比较保守，仅只尽量优化内层查询中的连接，排序和聚集操作阻止了优化器将内层查询与外层查询混合，最终，查询基本上按原始的编写方式执行。毫无疑问，在连接之前对金额排序非常“丑陋”，但若想在客户和订单数量激增时保持较快的速度，就必须这样编写 SQL 语句。

总结：为了降低查询对数据量增加的敏感度，应该在较深层的查询中只操作绝对必要的数据，将辅助性的 join 操作留在外层。

消除关联子查询

本书已说过多次，“关联子查询”在计算每一条返回记录时都被调用一次。当数据量增加时，如何把分散的调用集中化，是个重要问题。在这一节中，会以实际例子说明“关联子查询”使用不当的影响以及如何解决。

例子来自 Oracle 环境，我们的问题是如何每小时以批处理的形式更新安全管理表。注意，如何加速检查本身就是个问题，随着时间的推移，生产服务器（production server）最终可能要花 15 分钟来做此检查——每小时一次的活动要花 15 分钟也太多了——这会使应用的可用性（availability）受到影响，引发全面问题。红色警报！

处理速度为何缓慢呢？其原因已定位在以下语句：

```
insert /*+ append */ into fast_scrty
 ( emplid,
   rowsecclass,
   access_cd,
   empl_rcd,
   name,
```

```
   last_name_srch,
   setid_dept,
   deptid,
   name_ac,
   per_status,
   scrty_ovrd_type)
select distinct
       emplid,
       rowsecclass,
       access_cd,
       empl_rcd,
       name,
       last_name_srch,
       setid_dept,
       deptid,
       name_ac,
       per_status,
     'N'
from pers_search_fast
```

统计数据总是最新的，所以我们必须注意调整查询。我们发现，蹩脚的命名 pers_search_fast 其实是以下查询定义的视图：

```
 1 select a.emplid,
 2        sec.rowsecclass,
 3        sec.access_cd,
 4        job.empl_rcd,
 5        b.name,
 6        b.last_name_srch,
 7        job.setid_dept,
 8        job.deptid,
 9        b.name_ac,
10        a.per_status
11 from person a,
12      person_name b,
13      job,
14     scrty_tbl_dept sec
15 where a.emplid = b.emplid
16   and b.emplid = job.emplid
17   and (job.effdt=
18         ( select max(job2.effdt)
19           from job job2
20           where job.emplid = job2.emplid
21             and job.empl-rcd = job2.empl_rcd
22             and job2.effdt <= to_date(to_char(sysdate,
23                                        'YYYY-MM-DD'),'YYYY-MM-DD'))
24      and job.effseq =
25           ( select max(job3.effseq)
26             from job job3
27             where job.emplid = job3.emplid
28               and job.empl_rcd = job3.empl_rcd
29               and job.effdt = job3.effdt ) )
30  and sec.access_cd = 'Y'
31  and exists
32         ( select 'X'
33           from treenode tn
```

```
34           where tn.setid = sec.setid
35             and tn.setid = job.setid_dept
36             and tn.tree_name = 'DEPT_SECURITY'
37             and tn.effdt = sec.tree_effdt
38             and tn.tree_node = job.deptid
39             and tn.tree_node_num between sec.tree_node_num
40                                      and sec.tree_node_num_end
41             and not exists
42                ( select 'X'
43                  from scrty_tbl_dept sec2
44                  where sec.rowsecclass = sec2.rowsecclass
45                    and sec.setid = sec2.setid
46                    and sec.tree_node_num <> sec2.tree_node_num
47                    and tn.tree_node_num between sec2.tree_node_num
48                                             and sec2.tree_node_num_end
49                    and sec2.tree_node_num between sec.tree_node_num
50                                                 and sec.tree_node_num_end ))
```

当然，这类“死亡查询”太复杂，难以一看就懂！就当是个练习吧，研读一下也很有趣。请认真考虑此查询，发现它的特点，找出可能的性能障碍……

好了，我们来交换一下意见，你或许已注意到一些有趣的模式：

- 大量子查询。有个子查询甚至是嵌套的（nested），且所有子查询都是关联子查询。
- 几乎所有条件的可选择性都很差。唯一的常量表达式出现在第 22 行，和当前日期进行比较，很可能没有过滤掉任何东西；第 30 行对 Y/N 字段的比较，以及第 36 行在 `tree_name` 字段上的条件，看起来只是粗略的分类。`insert` 语句没有 `where` 子句，这引起了我的注意，所以这个查询会处理非常多的记录。
- 类似 `between sec.tree_node_num and sec.tree_node_num_end` 这种表达式，是个危险的信号。这很像第 7 章所述的情形， Celko 嵌套集合。在 Oracle 中很难发现嵌套集合，但现货商品（commercial off-the-shelf，COTS）程序包为了具有可移植性（并不总会成功），没有使用好用的、但是特定 DBMS 才有的功能。
- 更微妙的是，外层 `from` 子句中的 4 个表（实际上其中的 `person_name` 是个视图），只有 3 个（`person`、`person_name`、`job`）被明确地连接。在 `scrty_tbl_dept` 上有个条件，但连接被隐藏在第 34 行到 38 行之间的一个子查询中了。这是个低效率的“处方”。

首先要做的是，对涉及多大数据量有个概念。`person_name` 是个视图，对它的查询没有性能问题。数据字典会告诉我们有多少记录：

```
TABLE_NAME                          NUM_ROWS
------------------------------      ----------
TREENODE                            107831
JOB                                  67660
PERSON                               13884
SCRTY_TBL_DEPT                         568
```

这些表都不大，不需要处理数亿条记录就能发现性能降低，这非常值得研究。致命的因素是“如何访问表”。在开发服务器上，查询视图返回的记录数并不困难，但需要勇气（这种查询很耗时）：

```
SQL> select count(*) from PERS_SEARCH_FAST;

 COUNT(*)
----------
 264185

Elapsed: 01:35:36.88
```

快速查看一下索引，显示出 `treenode` 和 `job` 的问题都是“过度索引（over-indexed）”——这是 COTS 程序包的常见缺点——而不是“明显遗漏索引”。

那么为什么查询如此缓慢呢？大量记录及关联子查询同时出现时，通常应该警惕。层叠出现的 `exists / not exists` 更值得考虑。

注意

在实际工作当中，做下列分析所花的时间，远比你现在阅读这部分的时间要多。实际上，以下分析归纳了好几个小时的工作成果，灵感并没有像闪光一样出现！

请仔细分析 `exists / not exists` 表达式。第一层子查询引进了 `treenode` 表，第二层子查询再度访问 `scrty_tbl_dept` 表（这个表在外层查询中已存在），而且将之与两个“当前行”进行比较——第一层子查询（第 47 和 48 行）的当前行和外层子查询（第 44、45、46、49、和 50 行）的当前行。想要提高性能，我们别无选择，必须将这些紧扣的查询“拆开”。

我们理解了这个查询做了什么吗？尽管表命名为“`treenode`”易令人误解，但该表并不是保存“嵌套集合”。涉及数量范围时，全都与 `scrty_tbl_dept` 表有关，而 `treenode` 表更像由 `scrty_tbl_dept` 中描述的“节点”的非规范化扁平列表（在关系理论中这“很糟糕”）。还记得树状结构的嵌套集合实现吗？每个节点与两个值相关，而值的计

算方法是“子节点的值介于父节点的值之间”。如果这两个值相差1则为叶节点（反之则未必，因为子树可能会被修剪，而基于性能原因未重新计算值）。可以这样描述第 31 到 50 行的含义：

> 表 treenode 中有一条具有特殊 tree_name 的记录：通过 setid_dept 和 deptid 字段，该记录可对应至表 job；通过 setid 和 tree_effdt 字段，该记录对应至表 scrty_tbl_dept；该记录还指向 表 scrty_tbl_dept 中的当前“节点”或它的一个子孙节点。表 treenode 的当前记录不会指向表 scrty_tbl_dept 的其他节点（或子孙节点），只会指向 setid 字段和 rowsecclass 字段相符的节点或其子孙节点。

上面一段话很难懂，尤其是不知道表的用途时。可以将 SQL 改写得更易理解更高效吗？关键就在于“没有其他节点”的部分。没有子孙节点意味着表 treenode 中 tree_node_num 属性值确定的节点位于树的底端。原先的子查询无法与外层查询混合，我们可以编写单独的查询，它不关心表 treenode 和表 job 的连结，而是针对表 scrty_tbl_dept（这是个不到600行的小型表）计算与 setid 字段值和 rowsecclass 字段值相符的子节点的数量：

```
select s1.rowsecclass,
       s1.setid,
       s1.tree_node_num,
       tn.tree_node,
       count(*) - 1 children
from scrty_tbl_dept s1,
     scrty_tbl_dept s2,
     treenode tn
where s1.rowsecclass = s2.rowsecclass
  and s1.setid = s2.setid
  and s1.access_cd = 'Y'
  and tn.tree_name = 'DEPT_SECURITY'
  and tn.setid = s1.setid
  and tn.effdt = s1.tree_effdt
  and s2.tree_node_num between s1.tree_node_num
                           and s1.tree_node_num_end
  and tn.tree_node_num between s2.tree_node_num
                           and s2.tree_node_num_end
group by s1.rowsecclass,
         s1.setid,
         s1.tree_node_num,
         tn.tree_node
```

（count(*) - 1 意味着不将当前行计算在内。）当然，最后产生的集合很小，最多几百行。我们应该使用先前的查询作为内嵌视图，并应用过滤器筛除非叶节点（在上下文中）：

```
and children = 0
```

这样，也只有这样，我们才可以连接 job 表，得到结果集。最终代码中的视图部分不是很有趣，下面只列出 exists 的第一段：

```
and (job.effdt=
       ( select max(job2.effdt)
         from job job2
         where job.emplid = job2.emplid
           and job.empl-rcd = job2.empl_rcd
           and job2.effdt <= to_date(to_char(sysdate,'YYYY-MM-DD'),
                                            'YYYY-MM-DD'))
    and job.effseq =
          ( select max(job3.effseq)
            from job job3
            where job.emplid = job3.emplid
              and job.empl_rcd = job3.empl_rcd
              and job.effdt = job3.effdt ) )
```

上述代码的意图是，找到 job 表中满足如下条件的记录：其 effdt 值是当前（emplid, empl_rcd）对应的所有 effdt 值中最近的一个，其 effseq 值是当前（emplid, empl_rcd）对应的所有 effseq 值中最高的一个。相比其他嵌套子查询而言，这个条件并不可怕。不过，当有 OLAP 功能（Oracle 中叫作分析功能）可用时，可以更高效地处理这类“前几名”的问题。例如：

```
select emplid,
       empl_rcd,
       effdt,
       effseq
from (select emplid,
             empl_rcd,
             effdt,
             effseq
             row_number() over (partition by emplid, empl_rcd
                                order by effdt desc, effseq desc) rn
      from job
      where effdt <= to_date(to_char(sysdate,'YYYY-MM-DD'),'YYYY-MM-DD'))
where rn = 1
```

这个查询能轻易地返回我们真正感兴趣的（emplid, empl_rcd）值，并再以内嵌视图的方式传递给主查询。实际上，改写这个查询之后，每小时进行的例行批处理所需的时间，从 15 分钟降到了 2 分钟。

总结：尽量减小关联子查询对外层查询元素的依赖性。

通过分区（Partitioning）提高性能

Partitioning to the Rescue

随着记录数日渐增加，对小数据量有效的索引搜索（index search）逐渐变得低效。典型的主键搜索需要向下寻访索引，接着访问表的存储页，一般会引起 DBMS 引擎访问三四个存储页。范围扫描（range scan）相当高效，尤其是采用聚集索引（clustering index）时，此时记录按照索引键的顺序存储。然而，一旦数据量超过临界点，在索引存储页和表存储页间来回“穿梭”的开销将比普遍线性搜索（linear search）更高；因为线性搜索可以利用操作系统和硬件提供的并行化及预读能力，而依靠比较键的索引搜索本质上更注重顺序执行。当要检查大量记录时：应该采用类似“大扫荡”的存取方式、而不是“各个击破”的方式；join 也会通过哈希或合并方式执行、而不是以循环方式执行（这些都在第 6 章中讨论过）。

当结果集在被检查记录中占的比例很高时，表扫描会较高效。若可使用第 5 章介绍的“数据驱动分区”划分表，搜索条件就可以只操作表的一部分物理单元，从而使扫描效率极大提升。此时，对合理分区的部分进行大范围扫描，反而会比使用索引协助检查范围边界值更高效。

当然，数据驱动分区并不能奇迹般地解决所有大数据量的问题：

- 首先，分区键的再分配必须均衡。如果 90% 的记录都对应同一个分区键值，那么针对这个键扫描分区和扫描表不会有任何差异；而对其余 10%记录而言，通过索引存取会更高效。对可选择性较高的值而言，实施分区之后又使用索引，其效果微乎其微。分布均匀性是日期适合作为分区的原因、也是以日期进行范围分区广受欢迎的原因。

- 第二，范围的边界（包括下限值和上限值）必须已明确定义。这或许不太明显，但重要性毫不逊色。这不是分区的独有特点，因为索引范围扫描也是如此。除非我们查找的值只需大于“表中最大值接近的值”、或小于“表中最小值接近的值”，否则只单独限定范围的上限值（或下限值）不利于大幅降低必须检查的记录数。所以，定义如下的范围：

```
where date_column_1 >= some value
  and date_column_2 <= some other value
```

不会比只使用其中一个条件更高效。使用 between（或相同语义的其他操作符）定位少数分区，能充分利用分区的好处。

总结：单边范围条件（Half-bounded condition）不能充分利用索引和分区。

数据清除

Data Purges

备存并清除数据常被看作辅助手段，为挽回 6 个月前的良好性能而做最后努力。事实上，备存并清除数据是极敏感的操作，处理不好会造成许多问题，把不稳定状况推向无法收拾的边缘。

理想情况是，表已被分区（真正的分区或分区视图），然后在分区上进行备存和清除。如果分区可以通过某种途径进行分离，备存和清除操作就很简单了，另建一个新的分区代替备存分区即可。否则，通过 select 查询选取备存记录，之后对分区进行 truncate 操作（truncate 是清空表或分区的方法，它会略过常见的大多数机制，因而比正规 delete 快许多）。

注意

因为 delete 操作的许多工作都被 truncate 略过，所以使用时要小心。使用 truncate 也可能影响备份，还可能有使某些索引无效等副作用。使用 truncate 前一定要和 DBA 讨论。

次理想情况（但很常见）是，根据时间和其他条件触发备存。例如，会计师通常不愿备存未付款的发票，即使那些发票已经相当久了。于是，简单优雅的分区变得混乱，直接 truncate 也太武断。我们必须求助于单调但值得信任的 delete 吗？

此时，应关注不同数据操作（inserts、updates、delete）整体成本的高低。我们已看到 insert 的代价相当高，因为插入新记录时还必须维护“表上的索引”。而 update 只需要维护“更新字段上的索引”。但与 insert 相比，update 的弱点是双重的：首先，update 和搜索（where 子句）密不可分，这可能与使用 select 一样引发灾难，加

锁期间状况恶化。第二，先前的值（insert 的情况下并不存在）必须保存在某个地方，以备回滚（rollback）。Delete 则涵盖了所有缺点，既会影响所有索引，又常与缓慢的 where 子句相伴相随，还必须保存删除值以备回滚时使用。

总结：*在所有更改数据的操作中，删除最有可能造成麻烦。*

所以，如果能避免进行 delete（尽管其他操作还要做），就很可能取得不错的效果。如果表已分区，而备存和清除主要根据提供的日期字符串进行时，可考虑分 3 阶段清除（three stage purge）：

1. 把需保留的值 insert 到临时表。
2. 对分区进行 Truncate 操作。
3. 把需保留的记录从临时表重新 insert 到分区。

若没有进行分区，将更困难些。为了减少加锁持续的时间，我们可以考虑两步操作——前提是，一旦记录达到“准备备存”状态就无法回到“待定”状态。由于查找需备存记录较慢，所以这种两步骤操作较有优势：

1. 将所有需备存记录的 ID 放入列表。
2. 进行备存及进行清除时，都会使用该列表，而无需两次执行缓慢的 where 查询：一次在 select 语句中，另一次在 delete 语句中。

注意：*采用临时表的主要理由，是用来支持大量面向表的操作，而避免逐行操作。*

数据仓库

Data Warehousing

本书并不想花一半篇幅讨论数据仓库这个复杂主题，这方面已有许多书籍问世，有的较通用（其中最著名的两本恐怕要数 Ralph Kimball 的《The Data Warehouse Toolkit》和 Bill Inmon 的《Building the Data Warehouse》了，均由 John Wiley & Sons 出版），还

有一些则专门讨论数据仓库 DBMS 引擎。在 Inmon 的追随者和 Kimball 的支持者之间，有一些观念之争：Inmon 提倡完全按照 3NF 的要求设计“决策支持系统使用的数据仓库”；而 Kimball 认为应该用“维度模型”取代 3NF 模型，即不少参照数据一定程度上是“非规范化的（denormalized）”，因为数据仓库和“操作型（operational）系统”完全不同。

既然本书已花了很大篇幅讨论 3NF 设计，下面将着重讨论维度模型，研究其优点，发现其不足。我们将深入探讨“操作型数据的存储”和“决策支持系统”之间的交互，因为后者的维度模型的内容来自于前者——数据不是从天上掉下来的（你若在 NASA 或负责卫星运营的公司上班，数据倒是“从天上掉下来的”）。注意，使用 SQL 语言操作“表”，并不一定意味着处于关系世界。

事实表与维度表：星型 Schema

Facts and Dimensions: the Star Schema

维度模型的原则是保存度量值（measurement value），无论它们是数值量、统计值或任何其他需放入大型事实表（fact table）中的信息。而参考数据保存在维度表（dimension table）中，这里通常还包含自解释的（self-explanatory）标签，维度表明显是非规范化的。典型的维度模型具有 5 到 15 个维度表，每个维度表都采用系统产生的主键，而事实表包含所有维度表的外键。通常，事实表中的众多度量值和日期相关，但此日期不会以 `date` 字段的形式保存在事实表中，而是专门有一个名为 `date_dimension` 的、以系统生产序列号为主键的维度表，而事实表中保存此序列号为外键。表 `date_dimension` 将日期分解为各种形式，例如 Unix 社区传统的开始日期 1970 年 1 月 1 日会被存储为：

date_key	date_value	date_description	Day	month	year	quarter	holiday
12345	01/01/1970	January 1, 1970	Thursday	January	1970	Q1 1970	Holiday

而事实表中“参照 1970 年 1 月 1 日的每一条记录”都保存键值 12345。这种方式明显是非规范化的，其依据是：规范化对“数据修改”操作很重要，因为它是保证数据完整性的唯一方法；相反，数据仓库中保存冗余信息的危害不大，因为较之规范化的事实表，维度表中的信息量非常少。例如，`date_dimension` 保存一个世纪的信息也不过 36525 行。此外，Kimball 博士提出，如图 10-3 所示的维度表围绕事实表的结构（“星型 schema”由此而得名）使数据查询极为简单，所需的连接（join）非常少，所以执行得非常快。

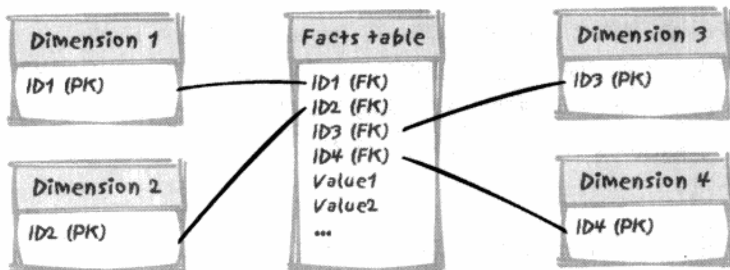


表 10-3：一个简单的星型 Schema，显示了主键与外键

稍有 SQL 知识的人都对“连接越少、查询越快”的言外之意感到惊讶，并为连接辩护：你动辄就连接几十个表当然慢！其实，如果你曾经有过在大型的、无索引的表上进行嵌套循环处理的痛苦经历，就会理解连接就是查询缓慢的原因。速度缓慢来自于查询的编写方式，从这个角度来看，维度模型非常合理，读完本章你会明白其中缘由。

总结：维度模型要求专门面向读操作而设计，所以经常忽略关系设计的规则。

查询工具
Query Tools

决策支持系统有个问题，即用户不懂 SQL 编程，就连糟糕的 SQL 代码也写不出来。因此，必须借助查询工具，提供友好的界面，帮助用户建立 SQL 查询。如第 8 章所述，根据一组固定的条件动态产生高效查询是很困难的，必须仔细思考，小心编程。所以不难理解，只有在复杂度比较低时，查询工具才能产生“像样的”查询。

以下是查询工具实际产生的代码片断（它显示了由 `from` 子句中的子查询返回的一些字段）：

```
...
FROM (SELECT ((((((((((t2."FOREIGN_CURRENCY"
                           || CASE
                                WHEN 'tfp' = 'div' THEN t2."CODDIV"
                                WHEN 'tfp' = 'ac' THEN t2."CODACT"
                                WHEN 'tfp' = 'gsd' THEN t2."GSD_MNE"
                                WHEN 'tfp' = 'tfp' THEN t2."TFP_MNE"
                                ELSE NULL
```

```
            END
            )
          || CASE
              WHEN 'Y' = 'Y' THEN TO_CHAR (
                      TRUNC (
                            t2."ACC_PCI"
                            )
                      )
              ELSE NULL
            END
           )
         || CASE
             WHEN 'N' = 'Y' THEN t2."ACC_E2K"
             ELSE NULL
           END
          )
        || CASE
            WHEN 'N' = 'Y' THEN t2."ACC_EXT"
            ELSE NULL
           END
          )
        || CASE ...
```

从上例不难发现，一些“商业智能（BI）”工具在业务方面投入了过多精力，而对产生SQL 查询关心不够。既然单纯使用 where 子句已无能为力，干脆就另想办法！此时，为了性能最好避免大量连接，实际上，越接近“在文件中进行文本搜索”（又名 grep）越好。还应合理利用“日期维度”，事实表含日期字段，而查询工具会把“第一季度（Q1）”转换成“1 月 1 日至 3 月 31 日之间”进行索引范围扫描——我们一过孩童时代就不敢相信还有这么简单的方法了。显式地列举最终用户可能使用的所有日期格式，分别加上索引，就能控制风险。非规范化维度和简单的连接，以及全面加上索引，有助于提供多数查询都可接受的速度。

总结：对维度模型施以设计较差的查询，性能也可以接受，因为维度模型比事务型模型简单。

数据抽取、数据转换、数据加载

Extraction, Transformation, and Loading

为了让业务用户主动运用“战略成本（strategic cost）”产生机会（如果数据仓库的数据是可信的），少不了一些数据输入的琐碎工作。即使有工具支持，此项工作也不轻松。

数据抽取

通常，数据抽取不是使用 SQL 语句，而是使用专用工具：不是 DBMS 提供的公用程序或特殊功能，就是合作厂商的专用产品。自编 SQL 查询来做这件事情不太现实，因为信息量很大，而全表扫描是最安全的策略。必须尽量以数组的方式操作（如果你的 DBMS 支持数组接口，就读取到数组内，或把多个值传递到单一数组内），这样可以减少 DBMS 核心和程序之间的来回调用。

数据转换

数据转换手段的选择受下列因素影响：SQL 技术、数据来源、对生产环境的影响程度、转换程度的大小。我们可以用 SQL 语言执行复杂的 select，返回准备上传的数据；也可以使用 SQL 修改暂存区域的数据；或者，在数据上传到数据仓库的同时，使用 SQL 执行转换。

转换经常涉及聚合（aggregate）操作，因为决策支持系统比生产环境所需的信息粒度大。通常，值可能以“天”为聚合单位。如果转换不比聚合复杂，就没有理由把它当成单独的操作——写数据库的代价比读数据库高，所以在更新数据仓库之前更新暂存数据会降低性能。

但是，当要组合来自数个不同营运系统的数据时，上述额外步骤将不可避免。从不同且不相关的数据来源获取数据，可能的原因有：

- 公司内的派系之分
- 最近合并的部门仍使用原先的信息系统
- 数据的分批迁移（migration）。例如某区域的运营仍基于旧的信息系统，而新系统已在总部应用并正在向全公司推行

把数个来源的数据集合在一起，应尽量一步完成，这看通过诸如 `union` 和内嵌视图（在 `from` 子句中的子查询）等集合操作符（set operator）实现。多步处理会带来一些风险，不应该直接应用在目标数据仓库。无论是表的多步更新，还是突然为空字段赋值，都会在物理层对数据库造成影响。当数据以不定长度存储时——如字符、数值等信息的保存

都经常如此，Oracle 就是这种存储策略——会导致部分数据被放入“溢出存储页（overflow page）”，于是完整扫描和索引访问的效率都会收到影响。这是因为，任何对溢出区域的指针，都会引起访问更多存储页，代价很高。而直接将准备好的数据插入目标数据仓库表，数据会被适当地重新组织。

多个更新依次应用到相同表的不同字段上，也相当常见，应尽可能在一个语句中更新所有字段，或通过 case 结构来做。

总结：对同一个表多次进行大规模更新，对物理层影响巨大。

数据加载

如果根据维度模型的规则建立数据仓库（或数据集市，有些人偏好这种说法），所有维度都会使用系统生成键来“跟踪”每种逻辑上相同、技术上不同的数据项。例如，在消费电子的制造领域，具有新参照值的新型号取代已停产的旧型号，对这两个型号使用相同的系统生成键，就可以把它们当成单一逻辑实体。

有个问题，操作型数据库（operational database）中的主键值通常与决策支持系统的维度表的主键值不同，最终，这不会影响维度表、反到会影响事实表。在操作型系统中，没有必要使用系统生成键，也未必需要记录电子设备的某个型号是哪一型号的后继产品。大多数情况下，维度表只加载一次，而且很少更新。“快速改变的值会被保存在事实表中”，维度模型有赖于这一假设，所以，插入事实表中的每一行都必须（从操作型系统主键）读取相对应的代理值，即每个维度的系统生成键。决策支持系统的查询只需较少连接，但“数据加载”工作要负责从操作型到维度表的键的映射，所以需要进行更多连接操作。

总结：基于维度模型的查询更加简单，这一优点要以复杂的数据准备和加载工作为代价。

完整性约束（Integrity constraint）与索引

“数据加载”时，令 DBMS 暂停检查参考完整性（referential integrity）比较明智，否则，DBMS 引擎必须对每条记录检查外键是否存在，工作量将增加两倍，因为任何一个加载事实表的语句必须寻找父表的代理键。另外，也可以先舍弃大部分索引，到“数据加载”之后再重建它们，以简化数据加载时的工作量，但必须保证“加载记录数”只占“目标表记录数”的很小比例，否则重建索引太耗资源和时间了。但应注意，暂停所有约束检查，尤其是暂停主键约束检查，可能造成致命错误：要加载的数据丢失了并不要紧，问题是很容易错误地重复加载相同数据，要知道之后删除重复项非常困难。

总结：临时修改 schema 一般是不容许的，但决策支持系统“数据加载”时例外。

查询维度表与事实表：即席报表
Querying Dimensions and Facts: Ad Hoc Reports

如果不考虑“麻烦的连接（evil join）”和“复杂的子查询（sophisticated subquery）”，查询工具无疑会轻松不少，但我们不能这样做，否则总有一天过于简单的 schema 将无法提供业务用户所要求的答案。于是乎，维度模型似乎有理由“打扮一番（embellished）”了，采用迷你维度（mini-dimension）、外部触发器（outrigger）、桥接表（bridge table），以及各种花哨的技术，直到它像个十足的 3NFschema 为止。殊不知，查询工具的灾难从此开始——有一天，高级用户大胆尝试了某些动作，第二天，问题立即反映到了查询工具开发者这里，因为昨天工具产生的查询今天还没有执行完呢…… 这就是即席查询的速度？这就是令人震惊的 SQL？

当必须编写即席查询时，需要回到维度模型并合理运用 SQL。基本上，维度代表了报表的不同部分。如果最终用户需要知道产品、商店、月份的销售情况，就分别需要 3 个维度：date、product、store。例如，可以采用非规范化的 product 表，其中包含产品所述的产品线、产品品牌、产品所属分类，甚至包含区域、产品外观等任何重要信息。显然，销售金额应属于事实表。

星型 schema 的主要特点是，要通过维度查询事实表，如图 10-4 所示。在先前的例子中，我们想查看第三季度美国西南部商店中的乳制品的销售情况。维度表不但是非规范化的，还增加了大量索引，这与规范化的事务型系统恰恰相反。对所有字段加上索引，意味着任何查询细节都涉及索引——例如位置维度中的城市（city）、州（state）、区域（region）、国家（country）、地区（area）等。毕竟维度表（或参考表）很少被更新，所以不会有维护大量索引的成本。如果所有条件都参照了维度表数据，而且任何类型的搜索都加上了索引，在逻辑上，就应该先访问维度表，而后找出事实表中的相关值。

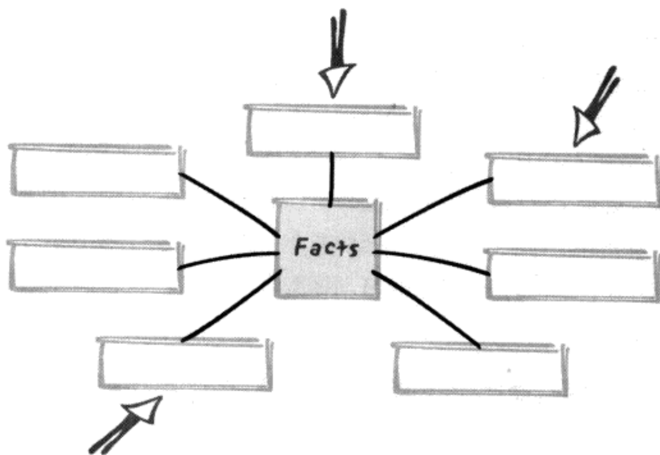


图 10-4：在维度模型中查询表的一般方式

先访问维度表强烈暗示，通常会使用搜索条件访问表中的特定记录，找出其中的外键，然后使用外键读取事实表数据。举个简单的例子，要查询 Owens 所属部门助理的电话号码，就应先在 employees 表中搜索“Owens”，找出部门编号，然后从 departments 表中找到电话号码。这是典型的嵌套循环连接（nested-loop join）。

使用维度模型时，情况完全不同。不再是从 employees 表到 departments 表，而是从事

实表到维度表，因为维度表中不含事实表的外键，而事实表包含维度表的外键。当从事实表连接至维度表时，DBMS 引擎必须采用与一般嵌套循环不同的机制，例如哈希连接（hash join）。

维度模型查询的另一个特点是：查询条件不是很明确，通过取交集的方式获得不太大的结果集。对此，优化器可以使用一些手段，例如：

判定在这些可选择性不高的条件中，哪个条件最具可选择性，把相关的维度表与事实表连接，然后检查其他维度表

这种战术伴随着一些困难。首先，维度表的建立方式，可能误导优化器。以 date 维度为例，它被用作不同日期的参照——如销售日期和商店开业日期，可支持各个商店一段时间内营业状况的比较。date 维度表绝不会很大，所以我们决定一开始就填入 70 年的信息，一方面满足“历史”查询所需，另一方面足够的未来日期避免了频繁维护工作。显然，作为去年第三季度销售额的参照，优化器会高估此条件的可选择性：如果在事实表中真的有个“销售日期”，要判定有用的日期范围就很直接了；如果只是有个指向日期维度的“日期参照”，评估的起点是维度表，而不是事实表。

扫描事实表，并丢弃未能满足其中任何一个条件的记录

由于事实表包含了所有的测量值，所以非常庞大。如果将它们分区，只能根据某一维度进行（如果使用了子分区就是两个维度），而涉及其他维度的查询则要对数百万条记录做完整扫描。扫描事实表不是最佳选择。

在这种情况下，在早期阶段（在第一个维度之后）访问事实表或许是个错误。有些产品实现了令人感兴趣的算法，在 Oracle 中就是所谓的“星型转换（star transformation）”。下面先详细讨论星型转换，之后讨论如何在其他环境中模拟星型转换。

总结：维度模型有个前提：先访问维度表，后访问事实表。

星型转换（The star transformation）

星型转换的原理是，首先将事实表与每个条件中出现的维度表分别连接。这岂不是要对事实表做多次连接？不要被表象欺骗。实际上，只需从事实表中获取满足维度条件的记录的地址，这些地址即所谓的 rowid（在 Oracle 中可以当成虚拟字段存取，Postgres 的等价功能叫 oid），它们存储在索引中。因此，我们只需连接 3 个对象：

- 出现在过滤条件中的维度表字段（例如表 `date_dimension` 中的 `quarters` 字段）上的索引
- 表 `date_dimension`，其中包含系统产生键 `date_key`
- 事实表中外键字段上的索引，此外键参照了 `date_key`（为事实表中的外键建立索引有利于星型转换更好地工作）

虽然事实表在星型查询中出现了多次，但不必重复访问相同的数据或索引存储页。不同连接涉及不同索引，而且全都保存了参照到同一个表的 rowid——但除此之外，那些索引完全是不同的对象。

只要有两个连接的结果，就可以将产生的两个 rowid 集合结合——对 and 条件丢弃交集之外的项目，对 or 条件则保留所有项目。当使用位图索引（bitmap index）时这一步骤更加简化，因为只需单纯的位操作即可。一旦获得了相对小的 rowid 集合，就可以从事实表读取对应的记录。

正如其名，位图索引通过位图（Bitmap）表示每条记录中是否保存了特定值。为低区别度的字段（即字段中只有少数不同的值）建立索引时，宜采用位图索引。那么为什么之前的章节没有提到位图索引呢？很简单，它们不适合一般数据库操作。当更新数据时，如果同时需要更新位图，则必须加锁。位图索引是针对具有少量不同值的字段设计的，

不宜更新太多记录，“位图加锁”介于“存储页加锁”和“表加锁”之间，但更接近“表加锁”。只读数据库适合采用位图索引，它在大量加载期间快速建立，而所需存储空间低于正规索引。

模拟星型转换

尽管自动进行的星型转换，甚至已经能使拙劣的查询高效执行了，但仍可以编写类似于（也许不完全相同）星型转换执行风格的查询。我必须承认，不应该编写如此独特的 SQL 语句；从关系的观点来看，这是极坏的典型。另一方面，维度模型与关系理论无关，我就硬着头皮以非关系的方式使用一回 SQL。

假设维度表分别名为 dim1、dim2...dimn，这些维度表围绕着名为 facts 的事实表。表 facts 的每条记录都包含外键 key1、key2...keyn，分别指向不同维度表。表 facts 还包含一些值字段（事实字段）val1、val2...valp。表 facts 的主键为复合键，由 key1 到 keyn 组成。

进一步假设，要执行的查询必须满足来自 dim1、dim2 和 dim3 上的 col1、col2、col3 字段的条件（例如产品类别、商店地点、以及时间区间）。我们将忽略任何转换、聚合等处理，重点练习如何高效地返回适当的记录集合。

星型转换主要致力于高效地从事实表获得结果集所属记录的 ID，结果集既可以是最终的结果集，也可以是需进一步处理的中间结果。下面，从连接 dim2 和 facts 开始，例如：

```
select ...
from dim2,
     facts
where dim2.key2 = facts.key2
  and dim2.col2 = some value
```

如果没有类似 Oracle 的 rowid 机制的支持，会产生严重的问题：因为来自 facts 的满足条件记录的 ID，正好是我们需要的返回项目。必须从 facts 返回主键才能识别记录吗？如果是，我们不但要访问 facts（key2）上的索引，还要访问 facts 本身，这违

背初衷。避免额外访问表的常用技术，就是把要返回的字段加入索引，将所需信息保存在索引中。于是，我们要把 facts (key2) 上的索引转成 facts (key2, key3, ... keyn) 上的索引吗？这意味着要对所有外键做同样处理！最后会有 n 个索引，每个索引的大小都与 facts 相同，这是无法被接受的，而且引起索引扫描读取大量数据，从而影响性能。

表 facts 需要相对小的记录 ID——代理键，不妨叫做 fact_id。虽然事实表有极佳的主键，而且虽然它不会被任何其他的表参照，但我们仍然需要精简的代理键——不是用在其他表中，而是用在索引中。

在使用由系统产生的 fact_id 字段时，就可以利用 (key1, fact_id)、(key2, fact_id) ... (keyn, fact_id) 上的索引。现在可以把先前的查询完整地写成：

```
select facts.fact_id
   from dim2,
        facts
   where dim2.key2 = facts.key2
     and dim2.col2 = some value
```

这个版本的查询，除了维度表 dim2、col2 上的索引，以及 (key2, fact_id) 上的事实索引之外，不再需要 DBMS 引擎访问任何其他对象。注意，对 dim2（当然还有其他的维度表）应用相同的技巧，可以把这个键附加到每个字段上的索引，执行查询时就只会访问索引。

对 dim1 和 dim3 重复此查询，即可获得满足维度条件的事实的 ID，再通过连接即可获得满足所有条件的 ID 的集合：

```
select facts1.fact_id
from (select facts.fact_id
       from dim1,
            facts
      where dim1.key1 = facts.key1
        and dim1.col1 = some value ) facts1,
      (select facts.fact_id
       from dim2,
            facts
       where dim2.key2 = facts.key2
         and dim2.col2 = some other value ) facts2,
       (select facts.fact_id
        from dim3,
             facts
        where dim3.key3 = facts.key3
          and dim3.col3 = still another value ) facts3
where facts1.fact_id = facts2.fact_id
   and facts2.fact_id = facts3.fact_id
```

之后，只要从事实表收集这些记录即可。

当然，刚才说明的技术并不局限于决策支持系统。但必须指出，我们假设拥有大量建立了索引的、只读的数据库，这是数据集市的标准配置，此时把更多的信息放到索引，并增加代理键，可视为“没有影响”。在正常的环境中（强调关系理论），难以修改 schema 来配合查询。

同时参照事实和维度查询星型 schema

正如我们已讨论的，设计维度模型的初衷是通过维度进行查询。那么，当输入的条件同时参照了某些事实（例如，销售金额大于特定值）和维度时（如图 10-5 所示），会如何呢？

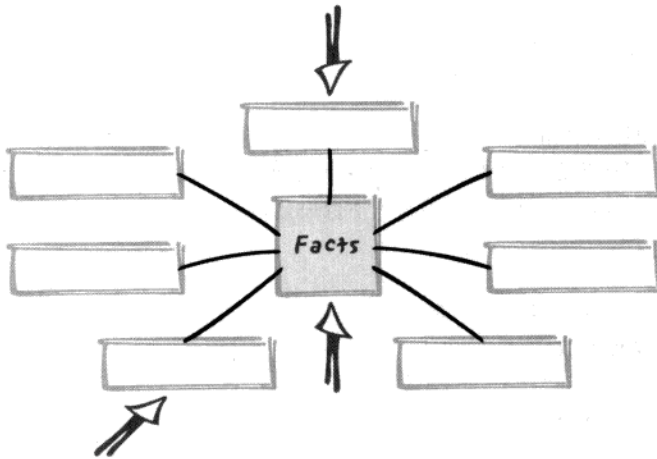


图 10-5：维度模型的不常见用法

可将这种情况与使用 group by 作比较。如果事实表上的条件用于聚合（通常是 sum 或 avg），则和使用 having 子句相似：在处理完所有的数据之前，无法提供结果，而事实表上的条件只不过是维度模型处理的额外步骤。所以，情况看似不同，其实不然。

与此相反，如果条件用于事实表的记录，我们就应该考虑：处理的早期阶段就丢弃不需要的事实记录，是否会更高效；使用类似 where 子句中滤除无用记录的方法在 group by

之前判断，而不是像 having 子句那样在 group by 之后判断。在这种情况下，必须仔细研究处理方式。除非条件中使用的事实字段上建立了索引——这不太可能也不太合理——否则仍然会通过其中一个维度开始访问。那么，选择哪个维度表作为访问起点呢？这取决于多个因素。可选择性是其中之一，但未必是最重要的一个。还记得聚集式存储的索引吗？它的性能远胜过其他索引，因为索引顺序与记录的物理顺序相符（或许正巧是数据输入的顺序，或者是专门定义为聚集索引）。相同的现象会发生在事实表和维度表之间。首先，事实表记录的顺序可能正好与日期相符。如果新的事实记录正好是以每天为基础，那么这些记录明显倾向于日期维度。其次，如果数据由大量站点提供且依次为每个站点进行处理和加载的话，事实记录的顺序可能与“位置维度”明显相关。所以，从顺序无关的关系模型角度看，星型 schema 好像是对称的，但实际上星型 schema 往往不对称。于是，尽量利用这种隐藏的不对称性很重要。

在搜索条件涉及的所有维度表中，如果某一个维度表与事实表较为符合，则最好将这个维度表与事实表连接；尤其是当搜索条件涉及事实表时，这种做法的可选择性更高。注意，必须通过外键连接“实际的事实表”，而不是像“星型转换”节所述的那样只访问约束。这样一来，根据事实表的记录可直接取得目标集合的超集，而且访问存储页的成本最小，并能尽早检查直接应用至事实记录的条件（其他条件之后检查）。

总结：*数据加载到星型 schema 的方式较倾向某一个维度。*

强烈警告

A (Strong) Word of Caution

维度模型是种技术而不是理论，它受欢迎是因为它很适合作为决策支持系统的工具（以 SQL（注 1）观点看它并不完美），也因为只读系统——指加载后为只读——可以容忍地毯式索引（就像“地毯式轰炸”）。问题是，当具有 10 到 15 个维度时，在事实表

注 1：基于某种原因，有些人会说，SQL 本身也不是没有缺点。

中就会有 10 到 15 个外键，而且要获得可接受的执行速度就必须为外键加上索引。你已看到，维度多数是静态的，而且数量不多，所以这样做不会有问题。但是，为事实表的所有字段加上索引令人烦恼，因为事实表会非常大——只要想象一下“大型连锁店每卖出一件商品就会记录一条事实”即可。新记录必须极有规律地插入事实表。在第 3 章中已看到，对索引进行插入操作代价极高，而此时的 15 个索引则会使加载非常之慢。加速加载的常见技巧是先舍弃索引，在数据加载后再重建索引（也可能并行运行）。这个技巧暂时行得通，但随着基础表越来越大，重建索引会花更多的时间。建立索引需要排序，而当记录数增加时，排序操作会严重影响性能。你迟早会发现重建索引会花太多时间，而你却接到通知说“用户在遥远的地方要在深夜访问数据仓库”。

如果有的用户需要晚上访问数据库，则意味着允许的决策支持系统的加载时间更短。但是，随着数据量的累积，重建索引会花更长时间，加载时间会越来越长。那么，不是每晚加载一次，而是让数据连续不断地从业务系统流向数据仓库可以吗？此时，非规范化成了问题，因为越需要连续处理，离事务型模型越接近，只有适当规范化才能避免完整性问题。在精心控制的特殊环境下，非规范化是可以接受的。但司令部离战场太近时，最好服从规范化原则。

第 11 章

精于计谋：挽救响应时间

Stratagems: Trying to Salvage Response Times

不过我已经跟我的教义越来越疏远。

——托马斯·哈代（1840—1928）
《无名的裘德》，第 4 部，第 2 节

我们通过第 1 章和第 2 章了解到，性能首要依赖于合理的数据库设计，其次依赖于清晰的策略和良好的程序设计。但实际情况令人堪忧。尽管你是公认的 SQL 调优高手，但人们直到发现性能问题之后才会寻求你的建议，这时至少也是验收测试的最后阶段了，已进行了数个人月的无序开发。在表设计、程序架构、甚至需求都不合理的情况下，人们期待调优高手创造奇迹。有些最敏感的问题和遗留系统的互操作有关——换言之，就是要读取遗留数据库的信息、或往遗留文件中写入数据。

如果本书中哪一章对你印象至深的话，应该就是这一章，我希望读者不仅记住本章的“处方”（那些机智的、有趣的、关键的 SQL 查询），还要记住每个解决方法背后的原理，我会尽量深入浅出地进行讲解。好的开始是成功的一半，但最糟情况下的力挽狂澜也不是没有效果。

我们发现，一些常见问题的解决方法令人费解，导致开发者进行扭曲的程序开发。这些程序不但效率极差，通常还极为含糊不清，甚至比复杂的 SQL 语句还难维护。

在本章最后，我们将讨论一些常用的策略，它们是和 SQL 间接相关的优化器指令（optimizer directives）。

下面开始讨论“困难的核心（heart of darkness）”。

数据的行列转换

Turning Data Around

解决 SQL 问题时，我们最常碰到的困难是：必须基于“非传统设计（unconventional design）”编程。性能卓越的查询很难编写。然而，我必须强调，由于设计不合理而强加给开发者的复杂 SQL 查询的编写，仅是程序复杂性（包括触发器和存储过程）的反映——由于设计不合理，为了完成诸如完整性检查等基本操作不得不进行复杂的编程。相反，合理的设计支持我们声明约束（constraint）让 DBMS 帮我们检查，毕竟确保数据完整性是 DBMS 的设计目标，最终避免了许多与复杂性相关的风险。很不幸，设计不合理时，我们要被迫花好几天来编写应用程序的控制手段，从而出现 bug 的机率也非常高。通用软件（popular software）每天会有数百万用户使用，所以 bug 会被较快地发

现并修正，而我们这里的定制软件（home-grown software），bug 往往要几个星期或几个月后才会被发现。

行转成列

Rows That Should Have Been Columns

“不可思议的四属性设计”只有 entity_id、attribute_name、attribute_type、attribute_value 四个字段，人们指望通过这种“设计”解决所有 schema 演进的问题，这就是“行本应是列”的最常见情况。令人恐惧的是，许多支持这种模型的人真的相信，它是复杂的规范化“归于简单”的表现。伴随这种设计出现的是各种讨人喜欢的名词——例如元设计（meta-design），或数据仓库中的事实维度（fact dimension）。

“不可思议的四属性设计”的支持者赞扬这种模型有“灵活性”，这种观点混淆了“灵活性”和“薄弱”：能立即增加“属性”，但那些属性不能被有针对性地读取和处理，就不能称为灵活性。一开始不苦心设计数据库，日后的编程工作量会增加（第一，要处理那些新记录，第二，要保证一定的完整性和数据一致性），比起这些，方便增加属性显得微不足道。解决属性数量可变问题的合理手段，就是定义子类型（subtype），如第 1 章所述。子类型支持你定义明确的参考完整性约束，开发者不必再为编写与维护完整性检查而劳神。数据库不仅仅是数据的存储库，还要考虑语义完整性（semantic integrity）。

“不可思议的四属性设计”有时被称为元设计，查询这种表的主要特点是相同的表在 from 子句中被引用非常多次。典型的的查询如下所示：

```
select emp_last_name.entity_id        employee_id,
       emp_last_name.attribute_value  last_name,
       emp_first_name.attribute_value first_name,
       emp_job.attribute_value        job_description,
       emp_dept.attribute_value       department,
       emp_sal.attribute_value        salary
from employee_attributes emp_last_name,
     employee_attributes emp_first_name,
     employee_attributes emp_job,
     employee_attributes emp_dept,
     employee_attributes emp_sal
where emp_last_name.entity_id = emp_first_name.entity_id
  and emp_last_name.entity_id = emp_job.entity_id
  and emp_last_name.entity_id = emp_dept.entity_id
  and emp_last_name.entity_id = emp_sal.entity_id
  and emp_last_name.attribute_name = 'LASTNAME'
  and emp_first_name.attribute_name = 'FIRSTNAME'
  and emp_job.attribute_name = 'JOB'
  and emp_dept.attribute_name = 'DEPARTMENT'
  and emp_sal.attribute_sal = 'SALARY'
order by emp_last_name.attribute_value
```

注意，同一个表在 from 子句中被引用 5 次，自连接的情况下则会更多。此外，这种查询经常涉及外连接（outer join）。

具有大量自连接的查询，在数据量大时非常慢。显然，在 where 子句中使用这么多条件，仅仅是因为要把各种“属性”组合在一起。所以，若把表定义为更具逻辑性的 employees (employee_id, last_name, first_name, job_description, department, salary)，查询会很简单：

```
select *
from employees
order by last_name
```

对于 employee_attributes 这样的表，多重连接和基于索引访问都会严重影响性能。所以执行上述查询的最佳做法，很明显是单纯的表扫描。

设计极差时，其上的查询不可能像合理设计上的查询运行得那么快，对 SQL 查询的改写再聪明也不过是亡羊补牢，只能在一定程度上挽回一点灵活性。然而，如果能避免多重连接，仅访问属性表一次，通常可以获得可观的性能提升。

我们的基本想法是，在单一记录中得到多个属性，这些属性反映设计表的初衷；而不是多条记录，每条记录只包含一个令人感兴趣的属性。我们知道，通过聚合函数即可把多条记录合并成一条记录。因此，如图 11-1 所示，可进行如下两个步骤：

1. 查询出所有记录，每条记录除了包含一个我们感兴趣的字段之外，还包含多个虚拟字段（我们希望有多少个属性，这里就有多少个虚拟字段）。

2. 聚合不同记录，每条记录只保留一个感兴趣的字段。适用于多种数据类型的 max() 等函数，此时非常有用。

为了使 max() 返回正确的最大值，我们使用的虚拟值（dummy value）一定要比该列的任何一个合法值都小。就算 max() 按照标准应忽略 null，但显式使用虚拟值还是更好些。

将图 11-1 所示的方法用于先前的例子，就可避免使用许多连接：

```
select employee_id,
       max(last_name)        last_name,
       max(first_name)       first_name,
```

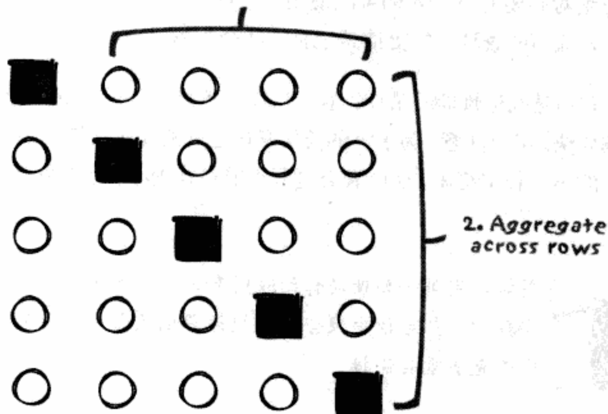


图 11-1：多条记录如何变为一条记录

```
        max(job_description) job_desription,
        max(department)      department,
        max(salary)          salary
   from -- select all the rows of interest, returning
        -- as many columns as we have rows, one column
        -- of interest per row and values smaller
        -- than any value of interest everywhere else
      (select entity_id                employee_id,
              case attribute_name
                  when 'LASTNAME' then attribute_value
                  else ''
              end                      last_name,
              case attribute_name
                  when 'FIRSTNAME' then attribute_value
                  else ''
              end                      first_name,
              case attribute_name
                  when 'JOB' then attribute_value
                  else ''
              end                      job_description,
              case attribute_name
                  when 'DEPARTMENT' then attribute_value
                  else -1
              end                      department,
              case attribute_name
                  when 'SALARY' then attribute_value
                  else -1
              end                      salary
      from employee_attributes
      where attribute_name in ('LASTNAME',
                               'FIRSTNAME',
                               'JOB',
                               'DEPARTMENT',
                               'SALARY')) as inner
   group by inner.employee_id
   order by 2
```

内层查询并不是绝对必要的——我们可以使用一连串的 `max (case when ... end)` ——但以这种方式编写的查询，能更清楚地呈现两个步骤。

如你所料，聚合并不是改进性能的最佳途径，但比上不足，比下有余，毫无疑问，比大量自连接的查询要快。但要注意：为了容纳任何意料之外的冗长属性，attribute_value 字段通常是相当大的不定长字符串，所以聚合过程可能极耗内存，若有好几十个属性就会出问题。

总结：用单一处理动作读取所有记录，把那些值散布在不同字段中，然后使用聚合函数把许多记录合并成一条，通常可以避免多重自连接。

列转成行

Columns That Should Have Been Rows

在前面的设计中，每个属性被定义为不同记录；另一个不良设计的例子正好相反，数据未建立为不同记录而建立成不同字段。许多初学者所犯的经典设计错误，就是为一些变量预先定义固定数量的字段，其中一些字段在没有值时就设定为 null。图 11-2 所示为典型的例子，这是个设计糟糕的电影数据库（请与第 8 章图 8-3 的正确设计作个比较）。

图 11-2：设计糟糕的电影数据库

图 11-2 中的糟糕设计，假设我们只需要记录固定数量的主角和一个导演。第一个假设明显是错的，而第二个假设也可能不对，因为许多喜剧短片有多个导演。这个模型对现实情况的描述存在严重缺陷，应该抛弃。更糟的是，在做报表输出时，该设计中的字段

要以记录方式呈现，这种混淆使查询很难编写。很不幸，为糟糕的数据库设计查询，在 SQL 开发的世界中就像缴税和死亡一样不可避免。

要把不同字段以行的方式显示时，就需要枢轴表（pivot table）。枢轴表用在想把字段看成记录之时，对表做旋转或转动。在 SQL 数据库的环境中，枢轴表是个工具表，它只包含一个字段，被填入了从 1 到任意所需数值的递增值。它既可以是 table，也可以是 view，甚至可以是内嵌在 `from` 子句中的查询。有经验的 SQL 开发者特别喜欢工具表，下面说明如何创建和使用枢轴表。

创建枢轴表

第 7 章讨论了访问树结构的一些方法，利用这些方法可以方便地产生枢轴表的值，例如，如果数据库系统支持的话可以采用递归 `with`。下面是个 DB2 的例子，产生从 1 到 50 的数值：

```
with pivot(row_num)
   -- Generate 50 values
   -- 1 to 50, one value per row
   as (select 1 row_num
       from sysibm.sysdummy1
       union all
       select row_num + 1
       from pivot
       where row_num < 50)
select row_num
from pivot;
```

Oracle 当然也支持类似技巧，即使用 `connect by`（注 1）。例如：

```
select level
from dual
connect by level <= 50
```

上述两种方法若在查询的 `from` 子句中使用，则查询很难看懂，因此，建议以正规表作为枢轴。但为枢轴表填入值时，递归查询倒是很有用（或者对现存表进行笛卡尔连接以便为枢轴表填入值）。典型地，枢轴表大约有上千条记录。

注 1：注意，Oracle 的一些旧版本不支持 connect by。

使用枢轴表使记录倍增

我们已创建了枢轴表，能用它作什么呢？它的用途之一，是用于记录倍增。将枢轴表和我们想转置的表结合，就能复制欲转换表的各个记录。要指定每一行的重复次数，只需对枢轴表上的连接增加限制条件就可以了，例如：

```
where pivot.row_num <= multiplying value
```

于是，可以用非常简单的方法让测试表 employees 中的 3 个记录倍增。首先，我们看到 employees 表有 3 条记录：

```
SQL> select name, job
  2  from employees;
NAME        JOB
----------  ------------------------------
Tom         Manager
Dick        Software engineer
Harry       Software engineer
```

下面使 3 条记录倍增：

```
SQL> select e.name, e.job, p.row_num
  2  from employees e,
  3       pivot    p
  4  where p.row_num <= 3;

NAME        JOB                               ROW_NUM
----------  ------------------------------  ----------
Tom         Manager                                  1
Dick        Software engineer                        1
Harry       Software engineer                        1
Tom         Manager                                  2
Dick        Software engineer                        2
Harry       Software engineer                        2
Tom         Manager                                  3
Dick        Software engineer                        3
Harry       Software engineer                        3

9 rows selected.
```

最好为枢轴表中具唯一性的记录加上索引，当访问极少记录时可免去完整扫描。

使用枢轴表中的值

笛卡尔连接除了支持倍增效果之外，还支持为要转置的表的每条记录副本分别赋值，值的范围从 1 到“倍增倍数”，这个值来自枢轴表的 row_num 字段。图 11-3 说明了在单一

记录情况下，源记录倍增和赋值的全过程。如果希望原先的记录以单一字段出现（这要求 col1 ... coln 的数据类型一致），则只能挑选一个字段参与笛卡尔乘积。之后，借助枢轴表的编号就可以对每个字段使用 case，以判断要显示的字段并排除其他字段。例如枢轴表中值为 1，就显示 col1，如果是 2 就显示 col2 ……依此类推。

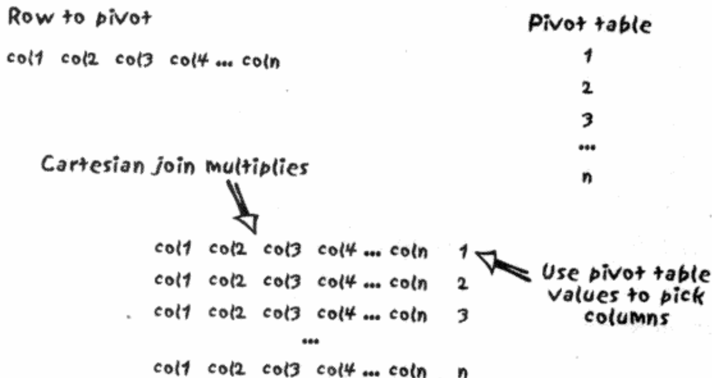


图 11-3：对一行记录进行转置操作

不用说，先让记录倍增，再丢弃大部分正在处理的字段，并不是最高效的处理数据方式，要记住，逆水行舟，事倍功半。理想的数据库设计，必须避免这种无谓的要求。

有趣的是，若数据库设计很糟糕，枢轴表有时利用提高性能。假设在本例的电影数据库中，要查询记录了多少位不同的演员（注意，actor_ ... 字段都没有索引，必须从表读取那些值）。于是，编写这个查询的方法之一是使用 union：

```
select count(*)
from (select actor_1
      from movies
      union
      select actor_2
      from movies
      union
      select actor_3
      from movies) as m
```

但我们也可以转置此表，此时 select 子句和设计正确时很像：

```
select count(distinct actor_id)
from -- Use a 3-row pivot to multiply
     -- the number of rows by 3
     -- and return actor_1 the first row in each
     -- set of 3, actor_2 for the second one
     -- and actor_3 for the third one
    (select case pv.row_num
                 when 1 then actor_1
                 when 2 then actor_2
                 else actor_3
              end actor_id
       from movies as m,
            pivot as pv
      where pv.row_num <= 3) as m
```

第二个版本的查询，速度是第一个的两倍——快多了。

pivot 与 unpivot 操作符

或许是不得不接受“一般数据库设计都比较糟”的现实，SQL Server 2005 引入了“pivot”和“unpivot”操作符，分别支持行转为列、列转为行的操作。先前的 employee_attributes 例子，可以使用 pivot 操作符编写如下：

```
select entity_id as employee_id,
       [lastname],
       [firstname],
       [job],
       [department],
       [salary]
from employee_attributes as employees
     pivot (max(attribute_value)
            for attribute_name in ([lastname],
                                   [firstname],
                                   [job],
                                   [department],
                                   [salary]))
               as pivoted_employees
order by 2
```

我们希望 attribute_name 字段的不同值，分别成为不同的字段，于是，我们用 for ... in 子句分别处理 attribute_name 字段的不同值，采用特殊的语法将字符数据转换成字段标识符。虽然 pivot 子句中没有出现，但 group by 会隐式地被应用于所有 empoyee_attributes 值；如果我们拥有 entity_id 之外的字段（例如 attribute_type），就必须小心，可能需要额外的聚合层。

unpivot 操作符会执行反向操作，将电影和演员以 (movie_id, actor_id) 的方式展现，可通过下列语句：

```
select movie_id, actor_type, actor_id
from movies
     unpivot (actor_id for actor_type in ([actor_1],
                                          [actor_2],
                                          [actor_3])) as movie_actors
```

注意，这个查询并没有精确产生我们要的结果，因为它引入了原始字段的名称，作为虚拟的 actor_type 字段。没有必要把演员限定为 actor_1、actor_2、actor_3，而且查询有可能会再被包入另一个只返回 movice_id 和 actor_id 的查询中。

使用枢轴表或 pivot 和 unpivot 操作符，是非常有趣的技巧，可以帮助我们解决很多棘手的困难。主流数据库系统虽然支持这两个转置操作符，但不该被理解为支持不良的设计，这是“现实政治（realpolitik）”的一个体现。

总结：枢轴表和 pivot / unpivot 操作符本身是很有用的技术，但它们不该作为掩盖糟糕设计的手段。

复杂字段的解析

Single Columns That Should Have Been Something Else

电影数据库这个案例的设计师对“一部电影的演员数量有限”这一事实过于敏感，于是积极创造性地使用了不恰当的技术：把演员 ID 以逗号分隔的字符串的形式，保存在一个很宽的 actors 字段中：

```
first actor id, second actor id, ...
```

这违反 1NF 也太过明显了……此处的重大设计错误，就是把需要处理的多项数据，一个接一个地保存在单一字段内。如果 DBMS 只需把复合字符串（例如冗长的 XML 讯息）当成不透明体，并且当成单元项目来处理，就不会有问题。但情况并非如此。现在一个字段中有好几个值，而且我们的确需要分别对待并操作每一个值。这可是个大麻烦。

对待这种“极具创意”的设计只有两种方法：

- 推倒重来。显然，这是最佳解决方案。
- 当延期、成本、以及政策要求快速的解决方案时，唯一的办法是应用极具创意的 SQL 解决方案。必须强调，此时“解决方案（solution）”的说法并不合适，叫“修改（fix）”较为合适。

另外指出，“XML 类型”字段犯的是相同错误，只不过稍微复杂些罢了——此时要使用 XML 解析函数，而我们的例子中使用单纯的字符串操作函数。

注意

警告：“极具创意的 SQL”通常是“丑陋的 SQL” 的婉转说法！

毫不犹豫地满足 1NF

我们的问题，就是抽取字符串的不同组成部分，然后以单独记录的形式依次返回这些不同部分。某些数据库系统中这么做比较容易（例如 Oracle 有非常丰富的字符串函数，能显著减轻此工作），而全部以逗号作分隔符的字符串更便于这样处理。真正的 SQL 开发者敢于面对最糟的情况：

1. 首先，假设 ID 列表形式如下：

   id1,id2,id3, ..., idn

2. 另外，还假设只能使用主流数据库系统共有的函数。对于这个例子，我们使用 Transact-SQL，而且只使用内建函数。正如你将看到的，设计良好的用户定义函数也对编写查询和提高性能有所帮助。

我们从（非常小的）movies 表开始，其中演员 ID 列表被（错误地）保存为电影的属性：

```
1> select movie_id, actors
2> from movies
3> go
 movie_id             actors
 -------------------- ---------------------------------------
                    1 123,456,78,96
                    2 23,67,97
                    3 67,456

(3 rows affected)
```

第一步，`actors` 字段的字符串最大为多长（假设最大长度 50 字符），就要使用枢轴表的多少行。最终，`movies` 表的记录数将倍增相应的倍数（此例即 50 倍）。自然，如果有百万条记录，我们不会这样做。顺便说明，有了返回字符串第 n 个分隔字符、或第 n 项位置的函数，记录数倍增的倍数就无需是最大字符串长度、而只需是演员 ID 个数的最大值了。

接下来，使用 substring() 函数连续取出 actors 的子集合（可以是空集合），从第一个字符开始，接着移到第二个字符，依此类推，直到最后一个字符（最多 50 个字符）。我们必须使用来自枢轴表的 row_num 值，以确定每个子字符串的开始位置。以 movie_id 值 为 1 的电影记录的 actors 字段为例，会得到如下结果：

```
123,456,78,96  associated to the row_num value 1
23,456,78,96   associated to the row_num value 2
3,456,78,96    associated to the row_num value 3
,456,78,96      ....
456,78,96
56,78,96
6,78,96
,78,96
78,96
....

....
```

我们在 substring1 字段中计算这些子集合。有了这些连续的子字符串，即可检查第一个逗号的位置。接下来，把 substring1 的内容搬移一个位置，并作为 substring2 字段返回。我们也会找出 substring2 中第一个逗号的位置。具体操作参见图 11-4 。我们只对下一个演员 ID 的起始位置感兴趣：第一条记录、以及所有逗号出现在 substring1 第一个位置的记录，这些记录的 row_num 值均赋为 1。对于这些记录，substring2 中逗号的位置代表了将分离的演员 ID 的长度。

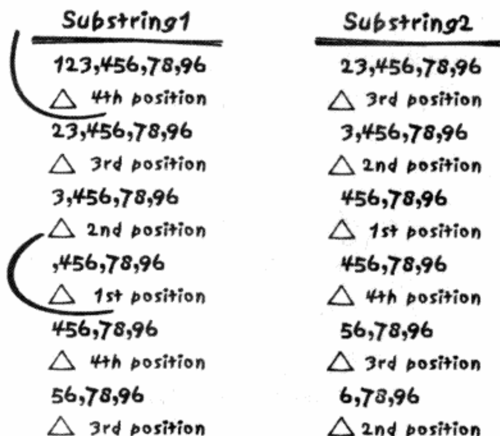


图 11-4：如何分割“以逗号作为分隔符的列表”

转换为 SQL 代码即：

```
1> select row_num,
2>        movie_id,
3>        actors,
4>        first_sep,
5>        next_sep
6> from (select row_num,
7>              movie_id,
8>              actors,
9>              charindex(',', substring(actors, row_num,
10>                                      char_length(actors))) first_sep,
11>             charindex(',', substring(actors, row_num + 1,
12>                                      char_length(actors))) + 1 next_sep
13>      from movies,
14>           pivot
15>     where row_num <= 50) as q
16> where row_num = 1
17>    or first_sep = 1
18> go
 row_num     movie_id      actors                       first_sep    next_sep
 ----------- ------------- ----------------- ----------- -----------
           1             1 123,456,78,96                      4            4
           4             1 123,456,78,96                      1            5
           8             1 123,456,78,96                      1            4
          11             1 123,456,78,96                      1            1
           1             2 23,67,97                           3            3
           3             2 23,67,97                           1            4
           6             2 23,67,97                           1            1
           1             3 67,456                             3            3
           3             3 67,456                             1            1

(9 rows affected)
```

如果愿意多花点时间去除逗号，也满足 ID 列表至少包含 2 个演员这一条件的话，要取得各个演员 ID 就相当简单了，只是最后产生的代码有点儿吓人：

```
1> select movie_id,
2>        actors,
3>        substring(actors,
4>                  case row_num
5>                    when 1 then 1
6>                    else row_num + 1
7>                  end,
8>                  case next_sep
9>                    when 1 then char_length(actors)
10>                   else
11>                     case row_num
12>                       when 1 then next_sep - 1
13>                       else next_sep - 2
14>                     end
15>                 end) as id
16> from (select row_num,
```

```
17>            movie_id,
18>            actors,
19>            first_sep,
20>            next_sep
21>      from (select row_num,
22>                   movie_id,
23>                   actors,
24>                   charindex(',', substring(actors, row_num,
25>                             char_length(actors))) first_sep,
26>                   charindex(',', substring(actors, row_num + 1,
27>                             char_length(actors))) + 1 next_sep
28>           from movies,
29>                pivot
30>           where row_num <= 50) as q
31>      where row_num = 1
32>         or first_sep = 1) as q2
33> go
 movie_id        actors                         id
 --------------- ------------------------------ ----------------
               1 123,456,78,96                  123
               1 123,456,78,96                  456
               1 123,456,78,96                  78
               1 123,456,78,96                  96
               2 23,67,97                       23
               2 23,67,97                       67
               2 23,67,97                       97
               3 67,456                         67
               3 67,456                         456

(9 rows affected)
```

通过对 actors 字段添加或附加逗号，可以让代码稍微简单些，有兴趣的读者可以试试。注意，为什么 ID 以左对齐方式显示呢？因为 ID 字段是字符串类型。所以为了和保存演员名字的表进行 join，必须先转换为数值。

上述例子很有趣，用了不少查询来解决问题。同时它也是个有益的警告，说明不良的表设计会造成 SQL 编程工作量激增。

掀开第 7 章路径分解的神秘面纱

还记得吗，第 7 章讨论了物化路径模型，它是一种描述树结构的方式。当时提到，如果可以把物化路径分解（path explosion）为其所有祖先节点的物化路径，就非常方便。这样做的优点是，自底向上的访问可以有效利用物化路径的索引。如果不进行物化路径的分解，要找到指定记录的祖先节点的唯一方法就是使用如下条件：

```
and offspring.materialized_path
    like concat(ancestor.materialized_path, '%')
```

可惜，这样做导致无法使用索引（原因类似于第 8 章的信用卡前置码问题）。

那么，如何分解物化路径呢？现在是说明原委的时候了。节点一般有多个祖先，所以首先要使记录倍增，倍增的倍数与祖先的层数相同。以此从最初记录（例如，代表在 Colonel de Marbot 指挥下的 Hussar 团的记录）的物化路径中抽取各祖先节点的路径。一如往常，记录的倍增借助枢轴表实现。MySQL 有个 substring_index() 函数，它能非常方便地返回第一个自变量的子字符串；子串开始于第一个字符，截止于第二个参数的第 N 次出现——N 由第三个自变量指定。需要从枢轴表读取多少条记录呢？只需计算路径中有多少个元素即可，具体方法与第 7 章计算深度的方法完全相同，即对路径的长度和除去分隔符后的路径长度作比较。查询及查询结果如下：

```
mysql> select mp.materialized_path,
    ->        substring_index( mp.materialized_path, '.', p.row_num )
    ->                                     as ancestor_path
    -> from materialized_path_model as mp,
    ->      pivot as p
    -> where mp.commander = 'Colonel de Marbot'
    ->   and p.row_num <= 1 + length( mp.materialized_path )
    ->                       - length(replace(mp.materialized_path, '.', ''));
+-------------------+---------------+
| materialized_path | ancestor_path |
+-------------------+---------------+
| F.1.5.1.1         | F             |
| F.1.5.1.1         | F.1           |
| F.1.5.1.1         | F.1.5         |
| F.1.5.1.1         | F.1.5.1       |
| F.1.5.1.1         | F.1.5.1.1     |
+-------------------+---------------+
5 rows in set (0.00 sec)
```

基于变量列表的查询

Querying with a Variable in List

下面说明枢轴表的另一个重要用途。在前几章曾强调过如何绑定到变量，即传递参数给 SQL 查询。变量只需被 DBMS 解释（parse）一次，之后变量已绑定，所以 DBMS 核心就可以跳过解释阶段直接使用变量。注意，解释可能包含搜寻最佳执行路径之类的高成本步骤。如第 8 章所述，甚至在动态创建 SQL 语句时，也可以给绑定的变量传值。

然而，还有个困难：最终用户可以通过组合框（combo box）作多重选择，并传递可变数量的参数的列表。与此相关有几个严重的问题：

- 有的语言不支持动态绑定可变数量的参数（通常必须一次绑定所有变量，而不是一个接一个），而且无论如何，都会相当难编写。
- 如果每次调用的参数数量都不同，当一个语句绑定的变量数量不同时，将被 DBMS 当成不同语句，这样就丧失了变量绑定的好处。

使用枢轴表提供的分割字符串的能力，可把值的列表当成单一字符串传递给语句，而不管那些值的实际数量。下面的 Oracle 案例使用的就是这种方法。

以下例子说明，当值的列表被包含在单一字符串中时，开发者如何传递值的列表到 `in` 列表中。这个字符串就是 `v_list`，而开发者将包括 `v_list` 在内的多个字符串串接成完整的 `select` 语句：

```
v_statement := 'select count(order_id)'
                    || ' from order_detail'
                    || ' where article_id in ('
                    || v_list || ')';
execute immediate v_statement into n_count;
```

这个例子看似是动态的，但事实上，对 DBMS 来说它完全是硬编码的。两个连续的执行都是独立的语句，在执行之前都必须被解释。可以把 `v_list` 当成参数传递给语句而不是把它串接到语句中吗？可以。在“毫不犹豫地满足 1NF”一节的例子中，我们对 actors 中以逗号分隔的值所采用的技术，同样可用于变量 `v_list` 中以逗号分隔的值。下列复杂的 SQL 语句用了枢轴表：

```
select count(od.order_id)
into n_count
from order_detail od,
     (  -- Return at many rows as we have items in the list
        -- and use character functions to return the nth item
        -- on the nth row
      select to_number(substr(v_list,
                                 case row_num
                                      when 1 then 1
                                      else 1 + instr(v_list, ',', 1, row_num - 1)
                                 end,
                                 case instr(v_list, ',', 1, row_num)
                                      when 0 then length(v_list)
                                      else
```

```
                                    case row_num
                                      when 1  then instr(v_list, ',',
                                                        1, row_num) - 1
                                      else instr(v_list, ',', 1, row_num) - 1
                                           - instr(v_list, ',',
                                                   1, row_num - 1)
                                    end
                                  end)) article_id
          from pivot
          where instr(v_list||',', ',', 1, row_num) > 0
            and row_num <= 250) x
    where od.article_id = x.article_id;
```

如果需要，你可以深入研究它的工作方式。此中的机制是以重复使用 Oracle 的 `instr()` 函数为基础的。函数 `instr` (*haystack*, *needle*, *from_pos*, *count*) 会返回 *haystack* 中从 *from_pos*（若没有指定就返回 0）开始的第 *count* 个 *needle* 项目，但其逻辑与先前例子中使用的完全相同。

我连续执行了这个查询的“枢轴版”和“硬编码版”，分别执行 1 次、10 次、100 次、1 000 次、10 000 次、100 000 次。每次都随机产生 1～250 个 `v_list` 值组成列表。结果如图 11-5 所示。在查询被重复执行时，“枢轴版”快 30%。

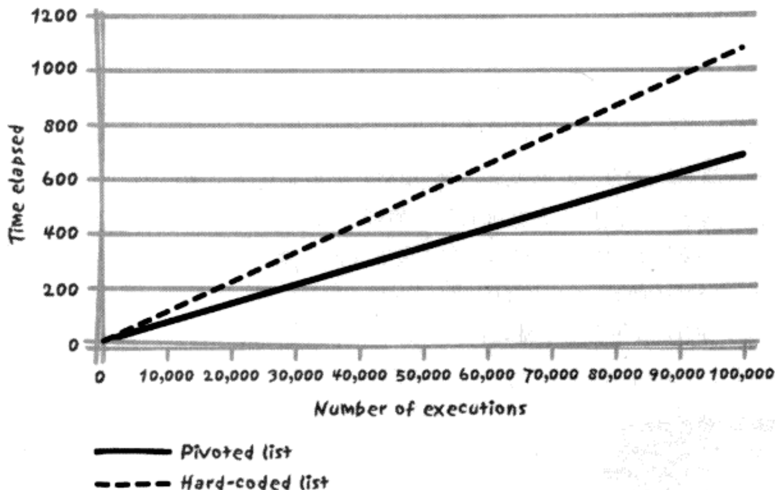


图 11-5：“枢轴版”和“硬编码版”的性能对比

注意，硬编码查询在每次执行时，都需要先解释再执行；而取用参数（绑定变量）的查询可以连续执行，只在第一次执行时有“边际成本（marginal cost）”。虽然后者明显比较复杂，但其性能方面仍然胜出。

实际上，还有两个好处在图 11-5 中没有体现：

- SQL 语句的解释是 CPU 密集型的操作，如果 CPU 正好是瓶颈，硬编码查询就会比较慢。
- SQL 语句会被缓存（cache），无论是参数方式还是硬编码方式，因为 SQL 引擎希望硬编码查询也被重复执行。以电影数据库作为例子，即使演员的名字被硬编码，但极受欢迎的演员经常被查询，所以有可能被执行多次，于是 SQL 引擎可以从 cache 中读取这些语句（注 2）。很不幸，能重复执行的硬编码语句仅是少数，一般来说硬编码语句不会重复，最终，为硬编码语句做 cache 的管理开销不小，意义却不大。

基于范围的聚合

Aggregating by Range (Bands)

有些人不太擅长编写针对范围作聚合的 SQL 查询。其实，运用 case 的思想，这种查询非常容易编写。下面我们分别显示记录数不同的表。例如，有多少个表的记录数少于 100 行？有多少个表的记录数在 100 到 10 000 行之间？ 数据量在 10 000 到 1 000 000 行、以及大于 1 000 000 行的表又分别有多少？

与表有关的信息，通常可以通过数据字典视图存取：例如，INFORMATION_SCHEMA.TABLES、pg_statistic、pg_tables、dba_tables、syscat.tables、sysobjects、systabstats 等。下面假设名为 table_info 的视图包含 table_name、row_count 等字段。根据这个表，使用 case 和 group by，就可以计算 row_count 的分布情况：

```
select case
         when row_count < 100
              then 'Under 100 rows'
         when row_count >= 100 and row_count < 10000
              then '100 to 10000'
         when row_count >= 10000 and row_count < 1000000
              then '10000 to 1000000'
         else
             'Over 1000000 rows'
       end as range,
       count(*) as table_count
from table_info
```

注 2：其实，这种特殊情况下，最佳优化策略是对查询结果进行缓存，而不是对查询进行缓存。

```
where row_count is not null
group by case
           when row_count < 100
                then 'Under 100 rows'
           when row_count >= 100 and row_count < 10000
                then '100 to 10000'
           when row_count >= 10000 and row_count < 1000000
                then '10000 to 1000000'
           else
                'Over 1000000 rows'
         end
```

上例只有一个问题：在聚合数据之前，group by 会执行排序。由于我们为每个聚合赋予了一个标签，查询结果会（默认地）按照字母序进行排序：

```
RANGE             TABLE_COUNT
----------------- ------------
100 to 10000                18
10000 to 1000000            15
Over 1000000 rows            6
Under 100 rows              24
```

从使用角度来看，应先显示记录数最少的表的数量，依次类推……为此，不要把精力花在标签的处理上，而是通过如下两个步骤来完成：

1. 在两个（而不是一个）字段上执行 group by，并为虚拟字段赋予标签，这么做唯一的目的是使用虚拟字段作为排序键。

2. 把这个查询包装为 from 子句中的查询，以便隐藏排序键，从而保证只创建和返回我们感兴趣的数据。

以下是应用此方法的查询：

```
select row_range, table_count
from ( -- Build a sort key to have bands suitably ordered
       -- and hide it inside a subquery
       select case
                when row_count < 100
                   then 1
                when row_count >= 100 and row_count < 10000
                   then 2
                when row_count >= 10000 and row_count < 1000000
                   then 3
                else
                   4
            end as sortkey,
            case
              when row_count < 100
                 then 'Under 100 rows'
              when row_count >= 100 and row_count < 10000
                 then '100 to 10000'
```

```
            when row_count >= 10000 and row_count < 1000000
              then '10000 to 1000000'
            else
              'Over 1000000 rows'
          end  as row_range,
          count(*) as table_count
    from table_info
    where row_count is not null
    group by case
                when row_count < 100
                   then 'Under 100 rows'
                when row_count >= 100 and row_count < 10000
                   then '100 to 10000'
                when row_count >= 10000 and row_count < 1000000
                   then '10000 to 1000000'
                else
                   'Over 1000000 rows'
             end,
             case
               when row_count < 100
                  then 1
               when row_count >= 100 and row_count < 10000
                  then 2
               when row_count >= 10000 and row_count < 1000000
                  then 3
               else
                  4
             end) dummy
order by sortkey;
```

查询执行结果如下：

```
ROW_RANGE          TABLE_COUNT
----------------- -----------
Under 100 rows              24
100 to 10000                18
10000 to 1000000            15
Over 1000000 rows            6
```

总结：基于范围做聚合操作时，需要建立人造排序键，以便按需要的顺序显示结果。

一般规则，最后使用

Superseding a General Case

上一节用过的技巧“把排序键隐藏在 from 子句中”，对其他情况也很有帮助。有种情况特别重要，即包含一般规则的表要延后使用，而包含特殊规则的表要优先使用。下面举例说明。

如第 1 章所述，各种地址的处理是个难题。以在线零售商为例，每个客户最多有两个地址：一个账单地址和一个送货地址。在多数情况中，这两个地址会相同。零售商决定把必需的账单地址保存在 customers 表中，而把 customer_id 和地址的各个组成部分（line_1、line_2、city、state、postal_code、country）保存在独立的 shipping_addresses 表中。

知道客户 ID，需要查询送货地址，下列两步查询是错误的：

1. 查找 shipping_addresses 中的记录。

2. 如果没有相符项目，就查询 customers。

正确处理这个问题的一种方法是，对 shipping_addresses 和 customers 进行外连接（outer join）。此时会得到两个地址，其中一个一般为空。我们可以通过程序检查有效的送货地址是否存在，不过这种解决方案不太好；其实我们可以使用返回第一个非空值自变量的 coalesce() 函数：

```
select ... coalesce(shipping_address.line_1, customers.line_1), ...
```

这样使用 coalesce()非常危险，因为它假设所有地址的非空的组成部分数量都相同。如果你认为真的会有不同的送货地址，但它的 line_2 组成部分是空的，而账单地址的 line_2 组成部分不是空的，就可能会产生无效的地址，该地址同时采用了来自送货和账单地址的组件。正确的方法，是使用 case 检查地址中的必要组成部分——难免形成较难阅读的查询。更好的解决方案是使用“隐藏排序键”技巧，结合对返回记录数的限制（select top 1 ...，limit 1，where rownum = 1，或类似的语句，这取决于 DBMS），编写查询如下：

```
select *
from (select 1 as sortkey,
             line_1,
             line_2,
             city,
             state,
             postal_code,
             country
      from shipping_addresses
      where customer_id = ?
      union
      select 2 as sortkey,
             line_1,
```

```
            line_2,
            city,
            state,
            postal_code,
            country
      from customers
      where customer_id = ?
      order by 1) actual_shipping_address
limit 1
```

主要思想是使用排序键标识优先级。因为结果集是有限集（limit set），所以能确保得到“最佳答案”（注意，当可用 OLAP 函数 `row_number()`时，类似概念也可应用到多个记录上）。这个方法会大幅简化应用程序端的处理，因为 DBMS 返回的是“保证正确”的数据。

上述技巧，也可用在多语言应用中，支持部分项目不打算转译成所有语种的情况。为了支持消息显示，只需定义默认语言，并确保能得到消息，从而避免了应用程序端的额外编程工作。

查询与列表中多个项目相符的记录

Selecting Rows That Match Several Items in a List

有个有趣的问题：如何编写“条件中要使用变量列表”的查询？使用图 11-6 中的 3 个表，查找具有特定技能的雇员，就是此情况的最佳说明。`skillset` 表建立了 `employees` 表和 skills 表的关系，并用 `skill_level` 值的 1、2、3 分别表示“可以胜任”、“经验丰富”、“非常精通”3 个档次。

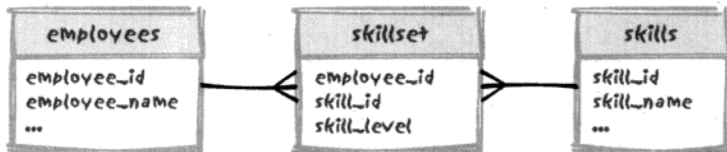


图 11-6：用于查询雇员技能的表

要查询 `skill_level` 是 2 和 3 的具有 SQL 技能的雇员，极为简单：

```
select e.employee_name
from employees e
where e.employee_id in
      (select ss.employee_id
       from skillset ss,
            skills s
       where s.skill_id = ss.skill_id
          and s.skill_name = 'SQL'
          and ss.skill_level >= 2)
order by e.employee_name
```

（或通过简单的 join 来实现上述查询。）如果希望查询掌握 Oracle 或 DB2 的雇员，可以这样编写查询：

```
select e.employee_name, s.skill_name, ss.skill_level
from employees e,
     skillset ss,
     skills s
where e.employee_id = ss.employee_id
  and s.skill_id = ss.skill_id
  and s.skill_name in ('ORACLE', 'DB2')
order by e.employee_name
```

无需检查 skill_level，因为我们接受所有的级别。然而，我们却需要显示技能名称，否则无法理解雇员满足要求的具体原因。我们将遇到的第一个难题是，同时掌握 Oracle 和 DB2 者会出现两次。可以尝试根据 employee 来聚合 skills，但可惜不是所有 SQL 方言都提供串接字符串的聚合函数（有时可以编写用户定义的聚合函数）。最终，可以通过使用单纯的双重转换策略（double conversion）对 skills 进行聚合。首先，将字符串转换为数值，然后对值进行聚合，最后将数值转换回字符串。

skill_level 的取值范围是从 1 到 3。所以，我们有信心采用两位数代表 Oracle 和 DB2 技能的任何组合，例如第一个数字代表 DB2 技能，第二个数字代表 Oracle 技能，如下所示：

```
select e.employee_name,
       (case s.skill_name
          when 'DB2' then 10
          else 1
        end) * ss.skill_level as computed_skill_level
from employees e,
     skillset ss,
     skills s
where e.employee_id = ss.employee_id
  and s.skill_id = ss.skill_id
  and s.skill_name in ('ORACLE', 'DB2')
```

根据掌握 DB2 的程度不同，computed_skill_level 会分别以 10、20、和 30 表示，而 Oracle 技能程度仍用 1、2、3 表示。此时，就可以轻松地对 computed_skill_level 进行聚合操作，并最终转换为更友好的描述：

```
select employee_name,
        -- Decode the numerically encoded skill + skill level combination
        -- Tens are DB2 skill levels, and units Oracle skill levels
       case
          when aggr_skill_level >= 10
             then 'DB2:' + str(round(aggr_skill_level/10,0)) + ' '
       end
       + case
           when aggr_skill_level % 10 > 0
              then 'Oracle:' + str(aggr_skill_level % 10)
         end as skills
```

```
from (select e.employee_name,
             -- Numerically encode skill + skill level
             -- so that we can aggregate them
             sum((case s.skill_name
                     when 'DB2' then 10
                     else 1
                  end) * ss.skill_level)  as aggr_skill_level
      from employees e,
           skillset ss,
           skills s
      where e.employee_id = ss.employee_id
        and s.skill_id = ss.skill_id
        and s.skill_name in ('ORACLE', 'DB2')
     group by e.employee_name) as encoded_skills
order by employee_name
```

还有另一个更困难的问题。如果需要人手的项目涉及 DBMS 迁移，我们就需要查询同时了解 Oracle 和 DB2 的雇员，而不是掌握其一即可。

此时，方法有多种。如果 SQL 方言支持 `intersect` 操作符（交集操作符），就可借此来解决：一方面找出精通 Oracle 的人，另一方面找出精通 DB2 的人，并保留同时属于这两个集合的人。当然，也可以用 in() 编写完全相同的查询：

```
select e.employee_name
from employees e,
     skillset ss,
     skills s
where s.skill_name = 'ORACLE'
  and s.skill_id = ss.skill_id
  and ss.employee_id = e.employee_id
  and e.employee_id in (select ss2.employee_id
                        from skillset ss2,
                             skills s2
                        where s2.skill_name = 'DB2'
                              and s2.skill_id = ss2.skill_id)
```

另外，还可以使用双重转换，并采用类似上述 `encoded_skills` 计算型字段的表达式对数值聚合结果进行过滤。这种做法还有其他优点：

- 只对表进行一次访问。
- 处理复杂问题时更容易，例如“同时了解 Oracle 和 Java，或同时了解 MySQL 和 PHP 的员工”。
- 由于只使用技能列表，所以可使用枢轴表并绑定技能列表，从而使此查询经常执行时的性能提升。如果列表比较短，可以这样来编码：`skill_level` 值分别乘以 10 的 0 到（`row_num` - 1）次方，并累加。如果查询不关心 `skill_level` 的具体值，而你的 DBMS 又实现了基于位（bit-wise）的聚合函数，你甚至可以动态构建位图（bit-map）。

最佳匹配查询

Finding the Best Match

最后，结合本章中说明的几个技巧，基于一些复杂而模糊的条件来查询雇员。我们想在雇员中找到一个最佳人选，他的技能跨多种环境（例如，Java、.NET、PHP、和 SQL Server）。完美的候选人是所有环境中的权威，但现实往往是没有哪个人在所有方面的技能都是最高的。所以，我们需求找出在尽量多的环境中能力最佳者——例如，雇员中 Java 权威是个世界级专家，但他对 PHP 一窍不通，则他不太可能被选上。

“最适合”是指在各雇员之间比较，或换个说法，是作排序。由于我们只想要一个优胜者，所以必须保证排序之后候选人是第一条记录。你应该已经开始规划查询，或许是 `select ... from (select ... order by) limit 1`，或许是 SQL 方言……。

当然，最大的问题就是如何对雇员排序。如果有个人在三个指定的主题中，有着还不错的知识，而另一个人在两个主题上是公认的权威，谁应该排在前面？本例认为，知识的广度比知识的深度重要。于是，首先判断雇员精通几项要求的技能，以此作为主要排序键（major sort key）。然后，根据雇员各项技能的 `skill_level` 值，计算出其总和，作为次要排序键（minor sort key）。自然，出现了内层查询：

```
select e.employee_name,
       count(ss.skill_id) as major_key,
       sum(ss.skill_level) as minor_key
from employees e,
     skillset ss,
     skills s
where s.skill_name in ('JAVA', '.NET', 'PHP', 'SQL SERVER')
  and s.skill_id = ss.skill_id
  and ss.employee_id = e.employee_id
group by e.employee_name
order by 2, 3
```

然而，关于最佳候选人的实际技能程度，这个查询不会告诉我们任何事。因此，应该通过转换，返回技能编号对应的技能描述。假设你还没被搞迷糊，可将此作为练习。

从性能角度而言，你应注意到了内层查询不需要参照 employees 表。雇员名字这项信息，只有在显示最后结果时才会需要。因此，我们应该只处理 employee_id 值，并使用 skills 和 skillset 表进行大量的处理。也应考虑特殊情况，例如两个候选人具有完全相同的技能——你肯定不能把他们通过一条记录输出吧？

注意

杜撰 Robert E. Lee 将军的一句名言：SQL 是如此的可怕，要不我们就会喜欢上它了。

优化器指令
Optimizer Directives

我将以优化器指令的警示，来结束这一章。SQL 优化器，可以比作自动相机中计算快门速度与曝光时间的程序。“自动”模式在有些情况中不再适用——例如，主体逆光或要拍摄夜景时。同样，所有数据库系统都提供了某些方法，可以推翻优化路径，或至少指示优化器如何寻找最佳执行路径。基本上，有两个技巧可以限制优化器：

- session 环境中的特殊设置。这会影响 session 中执行的所有查询，直到设置改变为止。
- 显式写入语句的本地指令（Local directives）。

本地指令的语法因产品而异，可能是 SQL 语句的一部分（例如 MySQL 的 `force index(...)`，或 Transact-SQL 的 `option loop join`），有可能是特殊语法的批注（例如 Oracle 的 `/*+ all_rows */`）。

到目前为止，本书还没有出现过优化器指令，这是有充足理由的。对生产数据重复执行查询，类似在一天的不同时间对相同主题拍照：早上也许逆光，但到了下午就顺光了。优化器指令注定会影响优化器的行为，最好不要干预。最可被接受的指令，就是明确指定预期输出的那些指令，例如 MySQL 的 `sql_small_result` 或 `sql_big_result`；或者快速响应特别重要时(如交易处理)，可以使用诸如 SQL Server 的 `option fast 100` 或 Oracle 的 `/*+ first_rows (100) */`。上述指令类似相机的“风景”或“运动”模式，提供了优化器无法以其他方式捕获的信息，并且和数据量及数据分布无关，因此，

它们很稳定，且提供了附加价值。总之，除非有必要，否则甚至连提供附加价值的指令也不应使用。一开始的查询编写合理的话，优化器多半能找到最佳执行路径。优化器秉承的最有用也最简单的一条指导原则，就是根据不同环境来决定使用关联子查询，还是非关联子查询最终功能上效果完全相同。

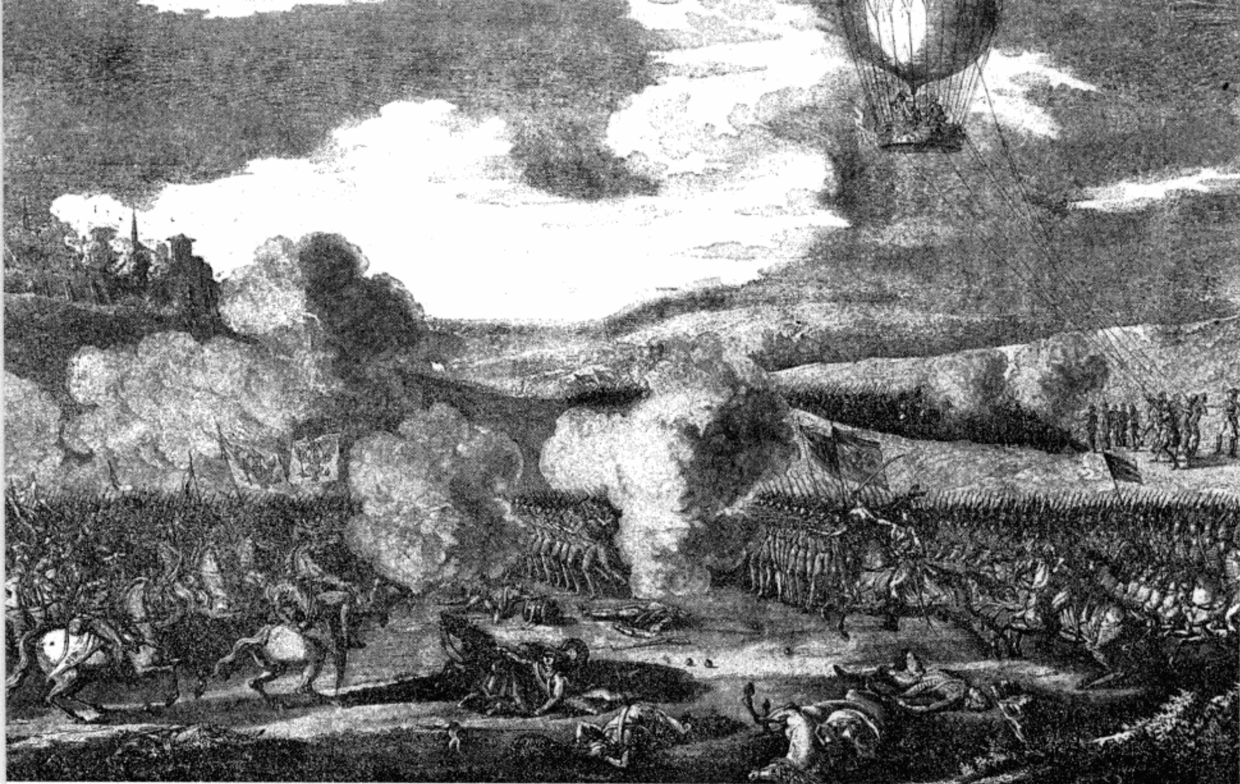
数据库优化器最棒的特性之一，就是适应环境变化的能力。使用具限制性的指令冻结它们的行为，预示着设计者的考虑不够长远，留下潜在的性能问题。有些指令是名副其实的“定时炸弹”，例如以名称指定索引：如果 DBA 基于某种原因，更改了指令中的索引名称，结果就会是场灾难。另外，如果指令指定了复合索引，而这个索引在某一天以不同的字段顺序重建时，也会造成灾难性的影响。

注意

优化器指令必须被当成 DBA 的私有领地。DBA 应该使用它们解决特定 DBMS 版本所带来的缺点，并在下次升级后，尽可能将它们去除。

补充说明：我们经常看到，经验不足的开发者根据已存在的查询改写新的查询。当原始的查询包含指令时，初学者很少质疑这些指令是否适合新情况。对于选择列表和搜索条件，初学者只会进行他们认为次要的修改。所以，最后得到的查询看起来仅被微调，但却遵循完全不同的执行路径。

总结：今天强加给查询的优良计划，明天可能是场灾难。



第 12 章

明察秋毫：监控性能

Employment of Spies: Monitoring Performance

那在黑暗里行走的，不知道往何处去。

——《新约 · 约翰福音》，第 12 章

情报收集一直是战争不可或缺的部分。所有的数据库系统也都包含监控工具，这些工具能力各不相同，有时还有来自第三方的工具。监控工具主要是给数据库管理员（DBA）使用的，然而，既然这些工具能像间谍一样让我们看到 SQL 引擎实际做的事情，那么它们就可以为重视性能的开发者所用。注意，当监控工具缺乏所需细节时，通常可通过开启日志或追踪功能以获得额外信息。因此,虽然日志或追踪功能在已十分忙碌的生产环境服务器中成为额外负担，但在性能测试期间却能够提供大量有用信息。

要详述现有的各种工具，不但冗长乏味，而且涉及的产品可能会很快过时。所以，本节把注意力放在"监控什么"和"为何监控"上，这无疑也是复习本书前面所讲的关键思想的好机会。

数据库速度缓慢

The Database Is Slow

在生产环境中常遇到一些性能问题，我们首先将它们分类。最开始，通常是 DBA 接到电话，并被告知"数据库很慢"（对于管理数百个数据库服务器的 DBA 而言，还真是有用的信息……）。若管理有条不紊，DBA 会检查监控工具是否报了异常，如果有，他会很有信心地回答"我们知道，正在处理该问题"。若管理混乱，DBA 可能提供相同的回答，只不过是"外交谎言"罢了。

无论如何，接到电话之后，就要开始疯狂收集线索。

人们之所以打电话说"数据库很慢"，通常由 5 种原因引起：

非数据库问题

网络不稳，或主机因某种原因超过负荷。谢谢您的来电。

整体性能突然下降

对于所有用户，所有的工作突然间慢下来。这有两个情况要考虑：

- 性能真的突然降低。通常是因为一些系统或 DBMS 的改变（软件升级、参数调整、或硬件配置修改）。
- 突然涌入的查询。

第一种情况不属开发问题，系统工程师或 DBA 的生活会变得紧张刺激起来。第二种情况是开发或设计问题。回忆第 9 章中邮局的比喻，当客户抵达的速度高于提供服务的速度时，队伍就会拉长，性能会急剧降低。如果原先的设计为负载增长留的余地太小，就会出现上述问题；要么就是应用未经过充分的压力测试。在许多情况中，改善某些关键查询可以大幅降低平均服务时间，并可稍稍缓解硬件升级的成本压力。整体性能突然下降常常表现为，第一个电话之后，别的电话就接二连三地打过来。

局部性能突然下降

如果某个任务突然变慢，应该考虑是否是“加锁（locking）”的问题。数据库管理员可以监控锁，并确认正在竞争相同资源的多个工作。这种情况属于开发和任务调度（task-scheduling）问题，应通过更快地释放锁来改善。

性能达到临界值而缓慢下降

临界值首先被敏感用户感觉到。遭遇临界值是“灾难迫近（an impending catastrophe）”的信号，它可能与三种因素有关：

- 第一，如果负载随着时间呈现规律性增长，则此时系统达到临界值可能是因为服务队列的整体性能突然下降。
- 第二，如果负载随着表大小的不断增加而不断增长，则是因为没有合适建立索引造成了临界值的出现。
- 第三，如果负载在大量删除/更新操作之后大幅增长，则临界值是因为物理存储的质量下降造成的。（表的实际大小先膨胀而后缩小，会造成申请的大量物理空间不能被释放，这种瑞士干酪效应造成许多存储页或区块并未存储数据，或存在过多溢出区域。）

如果问题与索引或物理存储有关（或过时的统计数据造成了优化器使用错误的执行路径），DBA 可以帮忙；但如果需要 DBA 的“定期救援”，则是设计不良的征兆。

某查询很慢

如果应用经过了适当的测试，但后来发现某查询很慢，则应注意动态建立的查询，可能是一些很不常见的条件组合在作怪。这纯粹是开发问题。

上述 5 种情况大多可以预见到并能及早预防。如果已清楚服务器负载增加的原因，那么理清“数据库活动”与“业务活动”的联系，就能找到应用中最弱的环节。接着，进行性能测试，专注于那些弱点的改善。

总结：为了预测实际应用的性能，必须密切关注压力测试和用户验收测试。

服务器负载因素

The Components of Server Load

信息技术中的“负载（load）”最终可归结为 CPU 过度使用、输入／输入操作太多、网络速度慢、网络带宽不足等因素的组合。这很像项目管理中的的“关键任务（critical task）”，任何瓶颈——即使并未完全停止——都会使整个系统慢下来。一个进程必须等待其他进程释放 CPU，或相反，无事可做的 CPU 等待网络或持久存储设备返回数据，都使系统处于“过载（overloaded）”状态。

但是，“过载”不是绝对的概念，负载和工作量不成正比。在这方面，可以将“系统”与“人类”作个比较，正如帕金森（C. Northcote Parkinson）在《帕金森定律》一书中对官僚制度的著名讽刺：

> 因此，上了年纪、有空闲的女士，可以在编写和发送明信片上花上一整天……只需占据大忙人三分钟工作的事件，却可能在同一层次上，让别人在一整天的怀疑、焦虑和辛苦后感到疲惫。

糟糕的 SQL 应用很容易引起服务器瘫痪，而且没实现什么价值，这样的例子很多，以下所列仅为三种：

硬编码所有查询

为了执行这种查询， DBMS 不得不每次进行解释和优化的工作，对 CPU 资源的浪费非常可观。

执行无用的查询

这种情况比一般人认为的更加常见。首先是完全无用的查询，例如，在每个语句前都有一个“虚拟查询（dummy query）”判断 DBMS 是否已启动（这可是真实的故事）；或者，调用 `count(*)`判断记录是否应被更新或插入。无用查询还包括：重复读取 Session 期间不变的信息；一天进行 40 万次货币汇率查询，而汇率只在每天晚上更新一次。

交互过于频繁

逐条处理记录，滥用光标循环，未用存储过程，都会增加应用与 SQL 引擎之间的

“交互”工作量，最终网络传输的数据包数量势必成倍增长，大量时间浪费在通讯协议上。另外，绝大部分数据操作散布在应用中，不利于优化器进行优化。

要强调的是，许多人认为一般是“糟糕的 SQL 查询”引起了性能问题，但上述三种降低性能的情况没有明确包含“糟糕的 SQL 查询”。其实，上述三种查询通常执行得很快，但这些无用的查询会使处理高峰时资源短期加剧。

有两个因素会影响数据库服务器的负载。首先，是显而易见的因素，即速度很慢的“糟糕的 SQL 查询”，大多数人会孤注一掷地进行这部分的调优。其次，是称为“背景噪音（background noise）”的不显眼因素，即那些需要被重复不断执行的、数量巨大的查询，每个查询的执行速度都可以接受，甚至很快；“背景噪音”对性能的影响积少成多，最终会影响那些大型的糟糕查询的执行。正如柯南道尔爵士（Sir Arthur Conan Doyle）笔下的福尔摩斯（Sherlock Holmes）所说的：

> 长久以来，我的格言是：小地方最重要。

“背景噪音”无时不在，它不会突然爆发，所以易被忽视；但是，它会极大地降低系统的“备用处理能力（power reserve）”，即降低了特殊场合出现的业务激增的处理能力。

总结：反复出现的、运行时间不长的、普普通通的查询语句对服务器负载的影响，往往比耗时很长的、大型的糟糕 SQL 查询的影响还大。

何谓“性能优良”

Defining Good Performance

负载是一回事，性能是另一回事。性能优良很难定义。使用 CPU 或执行大量 I/O 操作本身并不是错误，一般而言，公司都不希望购买强大的硬件却让它闲置。

当评估性能时，数据库领域和公司财务领域有显著的相似性：一方面，都对“关键性能指标”和投入产出比有较高渴望；另一方面，笼统的指标都有极大误解，例如优良的平均值常隐藏了高峰时段令人痛苦的结果，再如负载过高的批处理程序的性能远非最佳，但由于在夜间执行就没人注意了。要反映真实情况，必须“深入细节”。

宏观而言，“深入细节”与管理学中“作业成本法（activity-based costing）”相类似。公司很容易了解支出细节，然而，将成本与效益联系起来就会困难重重，对于像 IT 运营这样的横切方面就更不用说了。要判断究竟花了多少钱在硬件、软件、工作人员等方面，并把握这一切之间的软性联系是非常困难的，何况 IT 部门并不直接赚钱，实际带来收入的人是 IT 部门的“客户”。

可借助 3 个必要条件评估是否把钱花在了刀刃上：

- 知道花费了什么
- 知道获得了什么
- 知道投资回报率比公认标准是高还是低

下面将结合数据库系统的相关环境，依次讨论这 3 点。

知道花费了什么
Knowing What You Spend

在数据库性能上的“花费”，首先要看访问了多少数据存储页。有些人倾向于专注物理 I/O 上，其实这不是关键。既然访问了大量数据存储页，势必会引起相应的 I/O 活动（除非数据库完全放在内存中），除此之外访问存储页还造成 CPU 负载。但是，降低存取的数据存储页数量并不是万能药，因为有时访问存储页的次数略高于严格所需时，反而整体性能更高。不过，数据存储页绝对是单一指标中最重要的一项。另一个要注意的成本，是对 SQL 语句的多次解释（parse），这很耗 CPU 资源（大量硬编码的插入操作，动辄就需要 75% CPU 使用率）。

总结：两个最重要的数据库负载指标是：CPU 花费在语句解释上的总时间，以及执行查询需访问的数据存储页数量。

知道获得了什么
Knowing What You Get

做广告是大家经常引用一句话，是来自 19 世纪美国零售商华纳梅克（John Wanamaker）的妙语：

> 我花在广告上的钱，有一半被浪费掉；问题是，我不知道是哪一半。

这种情况在数据库应用中稍微好些，但也好不到哪儿去。我们根据 select 语句返回的记录数（或字节）衡量结果，但记录数同样受操作方式影响。这种基于外部事实的评估，很难反映 SQL 引擎所执行的工作，原因如下：

- 第一，事实上，所有产品都没有提供该项统计数据。
- 第二，获得结果集的开销可能与结果集的大小不成正比。通常返回少量记录，却需访问大量数据存储页。另外，对聚合操作而言“不成正比”是合理的。所以，就这个指标提供严格的规则不太可能。
- 第三，从数据库返回数据只是为了作为其他查询的输入，这有用吗？既然大部分记录中的值都将是 N，那么不用 where 子句而直接系统地把字段更新为 N 如何？对上述两种情况，虽然都可以通过返回或改变的字节数衡量 DBMS 引擎的工作量，但很不幸，其中很多工作量可以避免。

有时，扫描大型表，或执行耗时较长的查询，是绝对情有可原的（或无法避免的）。例如，利用大量数据生成综合报表就不会很快。如果需要立即得到答案，数据模型（数据库对现实的描述）极可能是不对称的（inappropriate），决策支持数据库就是典型案例，因为它们不比深入生产数据库的细节。如第 1 章所述：建模（modeling）是否正确，取决于“数据”和“想用数据做的事”。我们可能与供货商或客户共享数据，但我们有着完全不同的数据库模型。当然，要把数据从生产系统导入决策支持系统，两者都要面临冗长且高代价的操作。

如何处理数据的影响甚大，所以，必须将产生的负载与特定 SQL 语句对应起来，否则就无法评断性能。如果不能将每个语句与之所造成的负载对应起来，通过监控工具（最常见的是累积计数器）获得的性能整体数据也就没多大用处了。

因此，负载分析的第一个阶段必须找到所有 SQL 语句，并判断每个语句对整体性能的影响。是否一个不漏地找到了每个 SQL 语句并不重要，因为数据库领域非常适用 80/20 原则：经验表明，80% 的结果是由 20% 的原因引起的。通常，多数负载来自于少数

SQL 语句。注意，硬编码的 SQL 语句可能影响你的总体认识：使用硬编码语句时，DBMS 会记录几千个不同的语句；而编写合理的查询只被参照一次，通过使用不同参数即可完成几千次的调用。上述情况通过分析大量 SQL 语句可轻易发现，有时可借助整体统计数据——例如，借助 Transact-SQL 的 sp_trace_setevent 等过程可获得 cursor 被执行的精确次数，以及已准备好的 cursor 再执行的次数等。

如果没有工具可用，但你可以访问 SQL 引擎的 cache，则以每几分钟一次的较低频率对 cache 进行快照也颇有帮助。这种方法一般不会遗漏大型的糟糕查询，一分钟执行几十次的查询也不会遗漏。注意，此时必须注意总的开销，以便证明我们遗漏的因素对整体性能的影响较小。不过，这种快照的方法对识别硬编码的 SQL 语句的影响，效果不太好——此时应得到更全面的负载信息，可以通过日志或侵扰性不太强的 sniffer 工具。必须指出，找到所有硬编码的语句之后，必须将这些语句中的常量值从语句中去除，使之恢复"软编码（soft-coded）"；然后，才能评估每种模式的 SQL 语句——而不是每个单独的 SQL 语句——在整体负载中占了多大比重。

虽然找到让 DBMS 持续忙碌的语句，就是唯一的重点，但如果不能将 SQL 活动与本质的业务活动联系起来，就会遗漏许多东西。对 SQL 性能而言，了解每处理一个客户订单平均要调用多少个 SQL 语句，比知道在标准温度和压力下的磁盘传输速率或 CPU 速度更重要。首先，这有助于我们预测下一次广告活动会对系统造成多大影响；第二，如果每处理一个客户订单平均要调用几百个 SQL 语句，有助于我们发现程序的问题（是否误将 SQL 语句放到了读取其他语句结果的循环中了？是否存在只需执行一次却被执行多次的语句？）。同样，对表中两个字段的大规模更新，分作两次分别进行（而且具有类似 where 子句），立即会引起质疑：为什么不一次处理完成？

总结：负载大小必然和 SQL 语句相关，SQL 语句必然和业务活动相关，业务活动必然和业务需求相关。

根据公认标准进行检查

Checking Against Acknowledged Standards

找到 SQL 语句，评估它们的成本，并建立 SQL 语句与公司或机构事项之间的大致关系，通常能帮我们明确哪些代码是需要认真检查的。可疑的代码可能是 SQL 语句、算法、或包含两者。但是，如何改善，可以改善多少，则是 SQL 专家技艺中的难点。此时经验很有用，但仍有一定程度上的不准确性。

了解基准值很有帮助，例如，通过简单的插入测试对硬件能支撑的插入速率有个了解。同样，要了解查询速率，这个速率可以通过对最大表执行可怕的完整扫描来获得。基础速率是多少？你的应用要达到的速率有是多少？二者比较通常会有所启发，它们甚至可能并不在同一个数量级。

总结：要了解环境的限制，请在机器上量测每单元时间能插入、读取、更新、删除多少条记录。

一旦定义了基准值，就可以找到“最佳改进点”，即业务回报较高，而技术可行性也较高的需改进部分。接着，专注这些部分，进行相应的改善。

有些实践者认为：只要最终用户不抱怨性能，就没有问题，也不必浪费时间改善性能。这种态度既聪明，又缺乏远见，理由有二：

- 第一，最终用户对不良性能的容忍度之高，常出人意料。或者更应该这么说，相比了解工作机制的人而言，最终用户对速度慢有着不同感受，他们会抱怨无法更快的处理，却较能容忍另外一些处理。抱怨不强烈，未必代表一切都很好；口头上的不满，也未必代表应用就有问题，但或许需要改进。

- 第二，服务器负载的轻微增长，可能意味着性能开始恶化，迅速从性能良好变成性能糟糕。如果环境非常稳定，就完全不必担心负载轻微增长。但系统在 1 年中的 11 个月都性能良好，只在某个特定月份活动记录奇高，则可能造成突然混乱。此时，背景噪声影响极大。当活动增加时，已过载的机器无法继续提供相同水平的服务，一定是超过阈值使性能突然下降。因此，关键在于活动激增前要先研究整个系统，看是否可通过改善代码来降低负载。如果改善代码不足以保证可接受的性能，或许就要更换或升级硬件了。

别忘了，“改善性能的回报”不只是技术部门的事。最终用户的感受应该被赋与最高优先权，即使有时存在偏见或与技术难题不符。用户才是使用程序的人，而且人类工程学亦应列入考虑范围。经常有人出于好意全力提升统计数据而非程序的吞吐量，最终用户自然不会满意。而巨大的技术努力换来的仅是最终用户不冷不热的态度，开发人员可能会感到受挫与不解，其实我们只为用户带来了局部改善。有个 18 世纪的作家写道，一个人对医生说：“嗯，尽管你许诺会治好他，但他却死了。”医生的回答很妙：“你当时不在场，不知道治疗的过程，他是被治好后死的。”

从最终用户角度来看，统计数据极佳，但数据库性能仍不足，就像为病人看病，但病人却因为另一个疾病死了一样。改善性能意味着，要实现最终用户高度可见的改善，即使改善的查询每个月只执行一次，但却是对用户极关键的工作。改善也要立足长远，进行优先级稍低的降低背景噪声等工作，以确保服务器在必要时可以提升处理能力。

总结：最终用户能感觉到的性能改善是最重要的，但绝不要忘记，在饱和运行的环境中，性能满意和性能糟糕相隔不远。

定义性能目标

Defining Performance Goals

性能目标通常用“消耗的时间”来定义，例如“本程序必须在两小时内执行完毕”。但最好以单位时间所处理的业务量来定义，例如“每小时 50 000 张发票”或“每分钟 100 笔贷款”，这是因为：

- 围绕程序实际提供的服务进行描述。
- 当业务量升高时，使最终用户更容易理解性能为何降低，开会时压力就会小些。
- 从心理上讲，提高吞吐量比缩短响应时间更令人兴奋。管理图表上也会反映更多向上的曲线，而不是向下的曲线。

总结：性能改善最首要的含义是：第一，相同时间内完成了更多工作；第二，花费的总时间甚至有所减少。

从业务任务角度思考
Thinking in Business Tasks

在专注于特定查询之前，不要忽略上下文。在循环中执行的查询是糟糕代码的标志，因为程序变量只保存从数据库返回、还未传递给另一个查询的信息。数据库访问的代价很高，应该使其保持最低。细想某些程序的编写方式，给人的印象是作者正在购物：他们跳进车子，开到超市，把车停下，在通道上来回走动，挑了几瓶牛奶，走向柜台，排好队，付款，把牛奶放进车中，开车回家，把牛奶放进冰箱；接着，检查购物清单的下一项，再开车去超市……。当他太太抱怨购物太花时间时，所得到的理由是路上车多，食品区的标识不清，收银员太少等等。这些理由虽很充分，且多少也增加了购物时间，但并不是修正的重点。

我遇到过一些开发者，他们真的认为多写简单查询能提高性能，我毫不费力地向他们说明了相反的做法才是正确的。同样，人们还说不含 join 的 SQL 语句能提高可维护性。事实上，这样做的唯一好处是便于经验不足（薪水低）的开发者来维护。于是，程序中到处是最基本的、看似高效的 SQL 语句；若某个语句性能糟糕，则草率判断“此 SQL 语句需要调优”……这太常见了，但有些“很慢”的语句（可能真的很慢）其实只是性能问题的部分原因。

总结：精心调优的语句、劣质的程序、设计欠佳的数据库，会和战术对头而战略不力一样糟糕，最多只能推迟惩罚来临的时间。

不懂得 SQL 语言适用于整个数据管理子系统，并且不懂得它不是简单的在硬盘和内存之间搬运数据的一组原语，就无法设计出高效的程序。数据库访问通常是程序中对性能最关键的因素，必须在总体设计（overall design）中考虑。

多写简单的 SQL 语句可以使程序更简单，这是危险的幻想。复杂性的根源不是语言，而是业务需求。只使用简单 SQL 语句，复杂性并未消失，只是从 SQL 端移到了应用端。而且，本应属于 DBMS 端的逻辑被编写到应用中，会增加许多数据不一致的风险，并使极大部分处理被排除在了 DBMS 优化范围之外。

我并提倡不加选择地采用冗长、复杂的 SQL 语句（即“单语句”方针）。例如，下面的例子本应该用多个语句来实现：

```
insert into custdet (custcode, custcodedet, usr, seq, inddet)
select case ?
         when 'GRP' then b.codgrp
         when 'GSR' then b.codgsr
         when 'NIT' then b.codnit
         when 'GLB' then 'GLOBAL'
         else b.codetb
       end,
       b.custcode,
       ?,
       ?,
       'O'
from edic00 a,
     clidet bT
where ((b.codgrp = a.custcode
        and ? = 'GRP')
      or (b.codgsr = a.custcode
         and ? = 'GSR')
      or (b.codnit = a.custcode
         and ? = 'NIT')
      or (a.custcode = 'GLOBAL'
         and ? = 'GLB'))
  and a.seq = ?
  and b.custlvl = ?
  and b.histdat = ?
```

要比较运行时参数与常数的语句，通常应该分割为多个较简单的语句。上例中，敢于 case 执行的值，和后续 where 句中与 GRP、GSR、NIT、GLB 比较的值相同。进行一些排他性测试，以挑选出一种匹配情况，明显应该放在应用端，强迫 SQL 引擎做这些事情很不合理。此时，应该用 if ... elsif ... elsif 结构（以降低发生机率）配合 4 个 insert ... select 语句使用。

当复杂的 SQL 语句能以更少的 DBMS 访问次数、更快地返回所需的数据时，就与上述情况完全不同了。冗长、复杂的查询未必缓慢，这完全取决于编写的方式。开发者显然不应不顾自己的 SQL 技能程度，无谓地编写 300 行的语句；但把尽可能多的行为放入一个 SQL 语句中，应该是改善语句性能的先决条件。

总结：*在改善程序、减少数据库访问次数之前，先调优 SQL 语句，意味着你忽略了改善性能的主要手段。*

执行计划

Execution Plans

当“间谍”（无论是用户或监控工具）已指引我们关注特定 SQL 语句时，就必须更仔细地检查。许多调优者喜欢细察执行计划，论坛或邮件列表中有大量帖子讨论这种问题，例如“我有个查询非常慢，执行计划如下……”。

执行计划通常会以缩排列表方式显示（通常比较复杂的）SQL 语句的执行步骤，或以图形方式显示，如图 12-1 所示。这张图显示了第 7 章中一个查询的执行计划。文字方式的执行计划不太吸引人，但较容易在论坛中张贴，所以一直以来都比较流行。无论执行计划是以图形还是文字方式呈现，正确阅读和理解执行计划是有价值的技能。

本书到目前为止，除了稍稍提及的几个例子之外，还未讨论执行计划这一主题。执行计

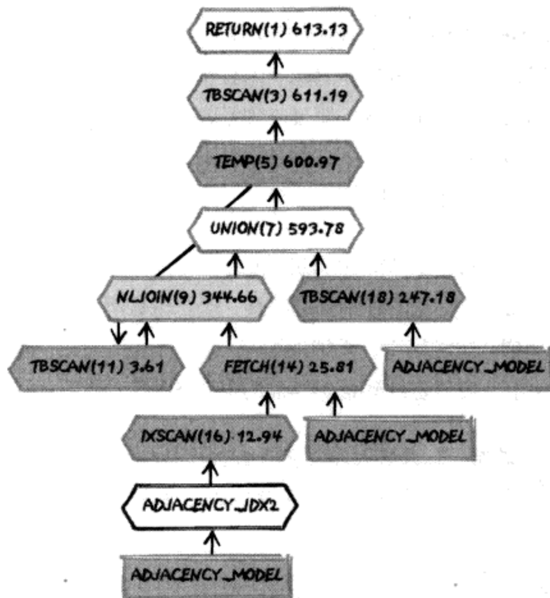


图 12-1：一个 DB2 执行计划

划是工具，大家有不同的喜好；你可能有不同看法，但我认为执行计划对性能“二等重要”。有些开发者认为它们“极为关键”，但下面两个实际例子将说明，执行计划的作用并没有那么大。

识别最快的执行计划

Identifying the Fastest Execution Plan

本节来测试一下你理解执行计划的能力。我们将展示 3 个执行计划，并从中选出最快的那个。准备好了吗？出发！祝好运！

备选执行计划

下列为相同查询在执行时，3 种不同的执行计划：

Plan 1

```
Execution Plan
----------------------------------------------------------------
   0      SELECT STATEMENT
   1    0   SORT (ORDER BY)
   2    1     CONCATENATION
   3    2       NESTED LOOPS
```

```
   4    3          HASH JOIN
   5    4            HASH JOIN
   6    5              TABLE ACCESS (FULL) OF 'TCTRP'
   7    5              TABLE ACCESS (BY INDEX ROWID) OF 'TTRAN'
   8    7                INDEX (RANGE SCAN) OF 'TTRANTRADE_DATE' (NON-UNIQUE)
   9    4            TABLE ACCESS (BY INDEX ROWID) OF 'TMMKT'
  10    9              INDEX (RANGE SCAN) OF 'TMMKTCCY_NAME' (NON-UNIQUE) ...
  11    3          TABLE ACCESS (BY INDEX ROWID) OF 'TFLOW'
  12   11            INDEX (RANGE SCAN) OF 'TFLOWMAIN' (UNIQUE)
  13    2        NESTED LOOPS
  14   13          HASH JOIN
  15   14            HASH JOIN
  16   15              TABLE ACCESS (FULL) OF 'TCTRP'
  17   15              TABLE ACCESS (BY INDEX ROWID) OF 'TTRAN'
  18   17                INDEX (RANGE SCAN) OF 'TTRANLAST_UPDATED' (NON-UNIQUE)
  19   14            TABLE ACCESS (BY INDEX ROWID) OF 'TMMKT'
  20   19              INDEX (RANGE SCAN) OF 'TMMKTCCY_NAME' (NON-UNIQUE)
  21   13          TABLE ACCESS (BY INDEX ROWID) OF 'TFLOW'
  22   21            INDEX (RANGE SCAN) OF 'TFLOWMAIN' (UNIQUE)
```

Plan 2

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT
   1    0   SORT (ORDER BY)
   2    1     CONCATENATION
   3    2       NESTED LOOPS
   4    3         NESTED LOOPS
   5    4           NESTED LOOPS
   6    5             TABLE ACCESS (BY INDEX ROWID) OF 'TTRAN'
   7    6               INDEX (RANGE SCAN) OF 'TTRANTRADE_DATE' (NON-UNIQUE)
   8    5             TABLE ACCESS (BY INDEX ROWID) OF 'TMMKT'
   9    8               INDEX (UNIQUE SCAN) OF 'TMMKTMAIN' (UNIQUE)
  10    4           TABLE ACCESS (BY INDEX ROWID) OF 'TFLOW'
  11   10             INDEX (RANGE SCAN) OF 'TFLOWMAIN' (UNIQUE)
  12    3         TABLE ACCESS (BY INDEX ROWID) OF 'TCTRP'
  13   12           INDEX (UNIQUE SCAN) OF 'TCTRPMAIN' (UNIQUE)
  14    2       NESTED LOOPS
  15   14         NESTED LOOPS
  16   15           NESTED LOOPS
  17   16             TABLE ACCESS (BY INDEX ROWID) OF 'TTRAN'
  18   17               INDEX (RANGE SCAN) OF 'TTRANLAST_UPDATED' (NON-UNIQUE)
  19   16             TABLE ACCESS (BY INDEX ROWID) OF 'TMMKT'
  20   19               INDEX (UNIQUE SCAN) OF 'TMMKTMAIN' (UNIQUE)
  21   15           TABLE ACCESS (BY INDEX ROWID) OF 'TFLOW'
  22   21             INDEX (RANGE SCAN) OF 'TFLOWMAIN' (UNIQUE)
  23   14         TABLE ACCESS (BY INDEX ROWID) OF 'TCTRP'
  24   23           INDEX (UNIQUE SCAN) OF 'TCTRPMAIN' (UNIQUE)
```

Plan 3

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT
   1    0   SORT (ORDER BY)
   2    1     NESTED LOOPS
   3    2       NESTED LOOPS
   4    3         NESTED LOOPS
```

```
  5    4          TABLE ACCESS (BY INDEX ROWID) OF 'TMMKT'
  6    5            INDEX (RANGE SCAN) OF 'TMMKTCCY_NAME' (NON-UNIQUE)
  7    4          TABLE ACCESS (BY INDEX ROWID) OF 'TTRAN'
  8    7            INDEX (UNIQUE SCAN) OF 'TTRANMAIN' (UNIQUE)
  9    3        TABLE ACCESS (BY INDEX ROWID) OF 'TCTRP'
 10    9          INDEX (UNIQUE SCAN) OF 'TCTRPMAIN' (UNIQUE)
 11    2      TABLE ACCESS (BY INDEX ROWID) OF 'TFLOW'
 12   11        INDEX (RANGE SCAN) OF 'TFLOWMAIN' (UNIQUE)
```

查询“战场”介绍

查询的结果集包含 860 条记录，涉及以下 4 个表：

表名	记录数（已四舍五入）
tctrp	18 000
ttran	1 500 000
tmmkt	1 400 000
tflow	5 400 000

所有表都加了索引，无需 create、drop 或重建索引，数据结构也都一样。3 个执行计划之间，只有查询语句不同，有的用了优化器指令。

考虑这 3 个执行计划，尝试给出速度排名，还可提出改善意见。

哪个是最佳执行计划

答案是：执行计划一花了 27 秒，计划二花了 1 秒，而计划三（此查询最初的执行计划）花了 1 分 12 秒。如果你选择了错误的计划也可以原谅，事实上，如果能根据现有信息选择出最快的执行计划（或有根据地怀疑结果）真是运气很好。我们注意到，最慢的执行计划是其中最简短的，没有包含索引之外的任何参照。而计划一说明，即使有两次相同表的完整扫描，也能让执行速度比简短计划（例如计划三这种只利用索引的计划）快三倍。

通过这个练习，我们要重点说明的是执行计划的长度并不重要，而且基于索引的独占性访问无法保证最佳性能。的确，如果返回 19 行的查询的执行计划长达 300 行虽然很麻烦，但你不能假设执行计划越短越好。

强制执行正确的计划

Forcing the Right Execution Plan

第二个例子有点不可思议，该查询来自某商业软件包。采用同样的 DBMS 的相同版本（硬件也接近），将该查询用于某数据库时，花 4 分钟、返回 40,000 条记录，用于查询另一个数据库时，所有都小得多，但要花 11 分钟。通过比较执行计划，发现前后两次查询的执行计划差异很大，但前后两个数据库的统计数据都是最新的，优化器也都发挥了作用。于是，问题成了如何强迫查询在较小的数据库上采用正确的执行路径。DBA 被要求尽力在这两个数据库上执行相同的计划，厂商的技术团队与客户的团队紧密配合，试图解决此问题。

棘手的查询

以下是此查询语句（注 1），后面列出了较快的执行计划。注意，优良的执行计划只访问索引，而不是表：

```
select o.id_outstanding,
       ap.cde_portfolio,
       ap.cde_expense,
       ap.branch_code,
       to_char(sum(ap.amt_book_round
           + ap.amt_book_acr_ad - ap.amt_acr_nt_pst)),
       to_char(sum(ap.amt_mnl_bk_adj)),
       o.cde_outstd_typ
from accrual_port ap,
     accrual_cycle ac,
     outstanding o,
     deal d,
     facility f,
     branch b
where ac.id_owner = o.id_outstandng
   and ac.id_acr_cycle = ap.id_owner
   and o.cde_outstd_typ in ('LOAN', 'DCTLN', 'ITRLN',
                            'DEPOS', 'SLOAN', 'REPOL')
   and d.id_deal = o.id_deal
   and d.acct_enabl_ind = 'Y'
   and (o.cde_ob_st_ctg = 'ACTUA'
        or o.id_outstanding in (select id_owner
                                from subledger))
   and o.id_facility = f.id_facility
   and f.branch_code = b.branch_code
   and b.cde_tme_region = 'ZONE2'
group by o.id_outstanding,
```

注 1：对象名稍有改动，以保护客户和厂商双方。

```
        ap.cde_portfolio,
        ap.cde_expense,
        ap.branch_code,
        o.cde_outstd_typ
having sum(ap.amt_book_round
          + ap.amt_book_acr_ad - ap.amt_acr_nt_pst) <> 0
    or (sum(ap.amt_mnl_bk_adj) is not null
        and sum(ap.amt_mnl_bk_adj) <> 0)

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   FILTER
   2    1     SORT (GROUP BY)
   3    2       FILTER
   4    3         HASH JOIN
   5    4           HASH JOIN
   6    5             HASH JOIN
   7    6               INDEX (FAST FULL SCAN) OF 'XDEAUN08' (UNIQUE)
   8    6               HASH JOIN
   9    8                 NESTED LOOPS
  10    9                   INDEX (FAST FULL SCAN) OF 'XBRNNN02' (NON-UNIQUE)
  11    9                   INDEX (RANGE SCAN) OF 'XFACNN05' (NON-UNIQUE)
  12    8                 INDEX (FAST FULL SCAN) OF 'XOSTNN06' (NON-UNIQUE)
  13    5             INDEX (FAST FULL SCAN) OF 'XACCNN05' (NON-UNIQUE)
  14    4           INDEX (FAST FULL SCAN) OF 'XAPONN05' (NON-UNIQUE)
  15    3         INDEX (SKIP SCAN) OF 'XBSGNN03' (NON-UNIQUE)
```

对较小的数据库加索引毫无意义。在前后两个数据库上的索引完全相同，而在较小的数据库上建立不同的索引也不会改变执行计划。在问题提出三周后，注意力已转到磁盘条带化，但希望不大。使用优化器指令逐渐成了唯一的出路。

使用优化器指令之前，最好先充分了解应如何查询。正如第 4 章和第 6 章所述，要确定合适的角度，需要评估各种查询条件的相对精确度，即使本例的结果集比较大（较大的数据库返回 40 000 条记录，较小的数据库返回 3 000 条），也使确定关键条件希望渺茫。

研究搜索条件

先以时区为唯一条件。此时，查询返回的结果集比使用所有过滤条件时多 17%，但它确实非常快：

```
SQL> select count(*) "FAC"
  2  from outstanding
  3  where id_facility in (select f.id_facility
```

```
  4                         from facility f,
  5                              branch b
  6                         where f.branch_code = b.branch_code
  7                           and b.cde_tme_region = 'ZONE2');

      FAC
----------
    55797
Elapsed: 00:00:00.66
```

单独使用标志作为过滤条件，返回的结果集是使用所有过滤条件时的 3 倍，但也非常快：

```
SQL> select count(*) "DEA"
  2  from outstanding
  3  where id_deal in (select id_deal
  4                    from deal
  5                    where acct_enabl_ind = 'Y');

      DEA
----------
   123970

Elapsed: 00:00:00.63
```

表 outstanding 上的 or 条件怎么样？结果如下：

```
SQL> select count(*) "ACTUA/SUBLEDGER"
  2  from outstanding
  3  where (cde_ob_st_ctg = 'ACTUA'
  4         or id_outstanding in (select id_owner
  5                               from subledger));

ACTUA/SUBLEDGER
---------------
          32757

Elapsed: 00:15:00.64
```

上述结果显示，我们已查明了问题所在。这个 or 条件正是造成查询执行缓慢的原因。

第一次查询的执行计划只显示索引访问：

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   SORT (AGGREGATE)
   2    1     FILTER
   3    2       INDEX (FAST FULL SCAN) OF 'XOSTNN06' (NON-UNIQUE)
   4    2       INDEX (SKIP SCAN) OF 'XBSGNN03' (NON-UNIQUE)
```

注意，这两个索引访问不完全是常见的索引向下访问类型。此处无需深究晦涩难解的细节，但 FAST FULL SCAN 这个选项，实际上选择了使用较小的索引而不是较大的表进行扫描，而 SKIP SCAN 这个选项是来自优化器的类似评估。换言之，选择访问方法并没有完全以最佳路径为根据，而是采用了“大体上比较好”的优化器评估。如果执行时间是可信的，那么 SKIP SCAN 不是最佳选择。

我们来看看 outstanding 表的索引（索引键和字段值的数量是估计值，它们说明了数字不协调）。以粗体显示的索引，就是出现在执行计划中的索引：

INDEX_NAME	DIST KEYS	COLUMN_NAME	DIST VAL
XOSTNC03	25378	ID_DEAL	1253
		ID_FACILITY	1507
XOSTNN05	134875	ID_OUTSTANDING	126657
		ID_DEAL	1253
		IND_AUTO_EXTND	2
		CDE_OUTSTD_TYP	5
		ID_FACILITY	1507
		UID_REC_CREATE	161
		NME_ALIAS	126657
XOSTNN06		**ID_OUTSTANDING**	**126657**
		CDE_OUTSTD_TYP	**5**
		ID_DEAL	**1253**
		CDE_OB_ST_CTG	**3**
		ID_FACILITY	**1507**
XOSTUN01 (U)	121939	ID_OUTSTANDING	126657
XOSTUN02 (U)	111055	NME_ALIAS	126657

另一个索引（xbsgnn03）和 subledger 有关：

INDEX_NAME	DIST KEYS	COLUMN_NAME	DIST VAL
XBSGNN03	**101298**	**BRANCH_CODE**	**8**
		CDE_PORTFOLIO	**5**
		CDE_EXPENSE	**56**
		ID_OWNER	**52664**
		CID_CUSTOMER	**171**
XBSGNN04		59542 ID_DEAL	4205
		ID_FACILITY	4608
		ID_OWNER	52664
XBSGNN05		49694 BRANCH_CODE	8
		ID_FACILITY	4608
		ID_OWNER	52664
XBSGUC02 (U)		147034 CDE_GL_ACCOUNT	9
		CDE_GL_SHTNAME	9
		BRANCH_CODE	8
		CDE_PORTFOLIO	5
		CDE_EXPENSE	56
		ID_OWNER	52664
		CID_CUSTOMER	171
XBSGUN01 (U)	134581	ID_SUBLEDGER	154362

这是使用 COTS 程序包经常遇到的情况，是地毯式索引 (carpet-indexing) 的极佳例子。

表 outstanding 上的索引会引发一些问题：

- 为何 outstanding 表的主键 id_outstnading 也出现在另两个索引的“排头字段（lead column）”呢？这必须有令人信服的理由。即使这样可以用于获取所有那些字段的值而无需访问表，但 id_oustanding 也不应该处在“排头字段”的位置；另一方面，不同取值的数量很高的字段不多，某些索引的必要性值得商榷。

- Subledger 也有问题。可选择性最高的值之一，正好是 id_owner。为何 id_owner 在 5 个索引中出现 4 次但都没有作为“排头字段”？对于常被参照的可选择性较高的字段，这很令人惊讶。顺便说明，发现应以 id_owner 为索引的排头字段，有助于解决查询存在的问题。

修改索引是个棘手的工作，需要小心研究所有可能的副作用。我们发现了一些可疑的索引，但还有亟待解决的问题。因此，我们先不对现存索引做任何修改，而是专注于 SQL 代码。

正如例中唯一性索引的不同键的数量所暗示的，这里不是在处理大型表；事实上，我们用于表 outstanding 的另两个条件虽然比较弱，但都提供了极佳的响应时间。在使用 or 结构时，之所以性能差，是因为合并来自两个索引的数据造成的，我们试试其他做法：

```
SQL> select count(*) "ACTUA/SUBLEDGER"
  2  from (select id_outstanding
  3        from outstanding
  4        where cde_ob_st_ctg = 'ACTUA'
  5        union
  6        select o.id_outstanding
  7        from outstanding o,
  8             subledger sl
  9        where o.id_outstanding = sl.id_owner)
 10  /

ACTUA/SUBLEDGER
---------------
          32757

Elapsed: 00:00:01.82
```

索引没有改变，但即使我们访问了 outstanding 表两次，优化器却突然领悟了。现在执行速度快多了。

取代“有问题的条件”，并稍微重组其他条件，原先花 4 分钟（据说还是“较好情况”）的查询现在只要 13 秒；而在另一个数据库，原先要花 11 分钟返回 3 200 条记录，现在只要 3.4 秒。

此案例的结论

如果进一步研究，并在索引方面做些改进，查询速度很可能比 13 秒还快。不过，似乎每个人都对 4 分钟很满意，13 秒也就足够了。

这个真实案例（以及本书的其他许多真实案例）的吸引人之处在于，我们如何才能（连续几周）专注于错误问题。上例较小的数据库上的确存在明显的问题。两个执行计划的对比，似乎说明较慢的执行计划是错误的（的确如此），那么较快的执行计划就是正确的（并非如此）？这样的逻辑是个大大的错误，而且误导了好几个人，于是他们专注于重现糟糕的执行计划，而不是改善查询。

我们必须总结一下这个案例的结论。查询被改写之后，在两个（数据量不同的）数据库上的执行计划仍然不同，这恰恰证明优化器在工作。

总结：判断查询性能的唯一标准，是花了多长时间执行，而不是执行计划是否符合偏见。

合理运用执行计划
Using Execution Plans Properly

执行计划很有用，但通常用于检查 DBMS 引擎是否按预期的方式工作。执行计划描述了来自战场的报告，有助于发现战术规划和实际执行是否一致，并能发现战术缺点和被忽略的细节。

如何干预查询的执行
How Not to Execute a Query

就算对合理的执行计划完全不了解，执行计划依然很有用。原因是，根据定义，有问题的查询的执行计划也很糟糕。知道执行计划糟糕，就可以通过形式逻辑（formal logic）中最久经世故的工具之一——三段论（syllogism）来发现改善查询的方法。三段论的推理由两个前提和一个结论构成。

推理过程如下：

（前提 1）这个查询很慢。

（前提 2）执行计划大都显示同类型行动，例如全表扫描、哈希连接、索引访问、嵌套循环等。

（结论）应该改写查询，且可能要改变索引，以便为优化器提供其他建议。

设法使优化器采用完全不同的执行计划，可通过如下手段：

- 当返回少量记录时，应增加索引，或重建复合索引并调整其中字段的顺序；此时，将非关联子查询转换为关联子查询，也会有帮助。
- 当返回大量记录时，做法相反，并在 `from` 子句中使用子查询，以建议表连接以不同顺序进行。
- 如果没有把握，则还有很多其他选择。例如用 `union` 或 `with` 子句分解查询。两个复杂查询的 `union`，有时可以转换成 `from` 子句内的较简单的 `union`。尽量使每个条件不依赖其他条件，通常也有帮助。一般来说，在干预优化器的执行顺序之前，应该首先尽量去除查询中强制的处理顺序，使优化器尽量自由。只有在别无选择的时候，才考虑干预优化器这一手段。
- 最后一招，就是优化器指令，应小心使用。

隐藏的复杂性

Hidden Complexity

在揭示隐藏的复杂性方面，执行计划也堪称有价值的间谍。查询的执行未必与表面理解的完全相同，某些参与查询的数据库对象可能导致额外工作，而执行计划可以发现这些额外工作。导致额外工作的数据库对象主要包括：

视图

看似简单的查询可能有欺骗性。有时，看起来是一个简单的 table，最终却成了负责查询所定义的视图，并牵涉到另外多个视图。视图的名称未必有启发，即使有，也无法说明视图复杂性的高低。而执行计划会毫无遗漏地检查 SQL 代码的执行，最重要的是，它会告诉我们哪些表被重复访问了。

触发器

有时，对数据库的修改会使系统工作反常，最终仅仅是因为触发器。触发器可能是速度极慢的代码，甚至会导致加锁问题。触发器很容易被忽视，而执行计划会揭示它们。

总结：执行计划之所以不可或缺，是因为它为调查性能提供了起点，并能揭示复杂视图和触发器引起的隐藏的数据库操作。

总结：影响性能的重要因素

What Really Matters

总结前面各章的讨论，影响查询性能的真正重要的因素包括：

- 表的数据量
- 表有哪些索引
- 存储特性（例如分区），和索引同样重要
- 查询条件的质量
- 结果集的大小

有了上述知识作为坚实基础，就可以超越执行计划本身，研究更有价值的“查询性能”问题了。首先了解你的上下文和你的数据，然后开始行动。查询时一定要尽量“尽快去除多余数据”，一定要尽量保持优化器的自由，避免语句内部存在依赖性而限制了表的访问顺序。

我要提醒大家，优化器（通常能提供高效优化）在以下情况下无法高效工作：

- 通过很多语句，分别读取数据片段。从应用角度看来这些 SQL 语句当然是相关的，然而，SQL 引擎绝不会“知道”这些，从而无法跨越语句的界限进行优化。SQL 引擎可以分别优化单个语句，但无法对整体处理优化。
- 随便使用 SQL 方言提供的各种非关系特性（虽然有时相当有用）。

记住，到大量的数据读取工作已完成时，才应用非关系特性（于较广的读取可接受度中；在作更新或删除之前，必须先读取数据）。非关系特性是基于有限集合的操作（换言之，就是数组），不同于关系理论中的无限集合。

曾几何时，能找到遗漏的索引、并改写语句以避免在函数中使用索引字段，就能赢得 SQL 专家的名声。这样的时代基本上已经过去了。许多数据库都被“过度索引”，有时是加了不合适的索引。使用索引字段的函数仍会出现，但函数索引提供了对特殊问题的“快速修正”。然而，对效率糟糕的查询的改写，依然只是重组条件或做些表面修改。

真正的挑战越来越需要整体思考，并且必须承认，需保存的数据量的增长速度比硬件性的增长更快，所以数据处理很关键。无论如何，数据处理需要 SQL。和所有语言一样，SQL 有自己的特点和优点，还有许多缺点。和所有语言一样，要精通 SQL 需要时间、经验和个人天赋。希望在精通 SQL 的漫长道路上，本书能真正对你有所帮助。

总结：建立性能最佳的 SQL，能带来强烈的满足感。祝君愉快！

作者简介

Stéphane Faroult 从 1983 年开始接触关系数据库。Oracle 法国成立早期他即加入（此前是短暂的 IBM 经历和渥太华大学任教生涯），并在不久之后对性能和调优产生了兴趣。1988 年他离开了 Oracle，此后一年间，他进行调整，并研究过运筹学。之后，他重操旧业，一直从事数据库咨询工作，并于 1998 年创办了 RoughSea 公司（*http://www.roughsea.com*）。

Stéphane Faroult 出版了 *Fortran Structuré et Méthodes Numériques* 一书（法语，Dunod 出版社，1986，与 Didier Simon 合作），并在 *Oracle Scene* 和 *Select*（分别为英国和北美 Oracle 用户组杂志）以及 Oracle 杂志在线版上发表了许多文章。他还是美国、英国、挪威等众多用户组大会的演讲者。

Peter Robson 毕业于达拉谟大学地质专业（1968 年），然后在爱丁堡大学任教，并于 1975 年获得地质学研究型硕士学位。在希腊度过了一段地质学家生涯之后，他开始在纽卡斯尔大学专攻地质和医学数据库。

他使用数据库始于 1977 年，1981 年开始使用关系数据库，1985 年开始使用 Oracle，这期间担任过开发工程师、数据架构师、数据库管理员等角色。1980 年，Peter 参加了英国地质普查，负责指导使用关系数据库管理系统。他擅长 SQL 系统，以及从组织级到部门级的数据建模。Peter 多次出席英国、欧洲、北美的 Oracle 数据库大会，在许多数据库专业杂志上发表过文章。他现任英国 Oracle 用户组委员会主任，可通过 *peter.robson@justsql.com* 联系他。

作者简介

Stephane Faroult [illegible] 1983 [illegible] Oracle [illegible]

[illegible]

Stephane Faroult [illegible] The Art of SQL [illegible]

[illegible] Oracle [illegible]

Peter Robson [illegible] (1994 年) [illegible]

[illegible]

[illegible] 1977 年 [illegible]

[illegible] Oracle [illegible]

索引

Index

Symbols

@@IDENTITY system variable (Transact-SQL), 235
10% of rows rule of thumb, 103
1NF (first normal form), 8
2NF (second normal form), 9
3NF (third normal form), 4, 9
    data warehousing and, 265
5NF (fifth normal form), 5

A

absence of data, result sets predicated on, 161–166
abstract layers, 202–205
accesses to the database (see database accesses)
ad hoc queries, 270
addresses, 109, 300–301
    atomicity and, 7
adjacency model (SQL trees), 172, 174, 197
    aggregating values stored in leaf nodes, 191
        computing head counts at every level, 192
    bottom-up tree walk, 185
    top-down walk, 178–182
aggregation
    consolidating multiple rows into one, 282–284
    double conversion, using, 302
    on dates, 156
    by range (bands), 297–299
    result sets obtained from, 150–156
    in transformations, 268
    values from trees, 190–198
        propagating percentages across levels, 194–198
        values stored in leaf nodes, 190–194
aggressive coding, 49
analytical functions (Oracle), 52
ANSI SQL query (example), 91
architectural solutions for contention, 242
architecture (global), choice of, 249
archival data
    purging, 263
    putting into production, 249
archives, location of, 170, 171
array interface, communicating between program and DBMS kernel, 31
art of SQL, governing factors, 84–88
    number of tables, 85
    number of users, 88
    result set criteria, 84
    result set size, 85
    total quantity of data, 84
art, science vs., xi
associative table, resolving many-to-many relationship between tables, 69
asynchronous processing, 22
atomic attributes, 6, 64
atomicity, 5
    business requirements and, 7
    function applied to a column, 64
attributes
    atomic, 6, 64
    dealing with varying numbers of, 281
    excessive flexibility in, 18
    independence of, 9
auto-incremented columns, 235
    not using in order to limit contention, 245
axioms, 3

B

backup databases, 24
bad SQL queries, 311
bands, aggregating by, 297–299
batch programs, 22
  queries returning large amounts of data, 102
  queries satisfied by data from an index, 111
BCNF (Boyce-Codd normal form), 5
Bill of Materials (BOM) problem, 168
bind variables, 162
bind_param( ) method, 218
binding variables, 294, 296
  PHP, 218–222
bitmap indexes, 273
blanket views, performance impact on queries, 117
blocks, 108
  contention for access, 88
  locking, 232
  minimizing accesses to, 146
  pre-joined tables and, 124
BOM (Bill of Materials) problem, 168
book indexes, table of contents vs., 59
Boolean columns, qualifying, 14
bottom-up tree walk, 178, 185–189
  adjacency model, 185
  materialized path model, 186
  nested set model, 188
  performance, comparing for various models, 188
boundaries of ranges, defining well, 262
Boyce-Codd normal form (BCNF), 5
bridge tables, 270
Building the Data Warehouse, 264
business logic, mirrored by SQL statements, 42
business processes, physical design and, 109
business requirements, 2
  atomicity and, 7
  database modeling and, 2
business tasks, focusing on, 317–319

C

C# code, 202
cache, SQL engine, 314
cardinality (low), 109
Cartesian joins, 285, 286
case expression, 42
case-insensitive searches with function-based index, 66
CBOs (cost-based optimizers), 36
Celko, Joe, 172, 176
centralizing data, 23
changing data, concurrency and, 231–246
  contention, 240–246
    architectural solutions, 242
    DBA solutions for, 241
    developmental solutions, 243
    insertion and, 240
    results from measures limiting, 243–246
  locking, 232–239
    committing and, 236
    granularity of, 232
    lock handling, 234–236
    scalability and, 238
child with multiple parents, 169
classic SQL patterns, 128–166
  large result set, 146
  nine common situations, listed, 128
  result set obtained by aggregation, 150–156
  result set predicated on absence of data, 161–166
  self-joins on one table, 147–150
  simple or range searching on dates, 156–161
  small intersection of broad criteria, 138–140
  small intersection, indirect broad criteria, 140–145
  small result set, direct specific criteria, 129–137
    criterion indexability, 132–137
    data dispersion, 130–132
    index usability, 129
    query efficiency and index usage, 130
  small result set, indirect criteria, 137
client/server environment, database connections, 30
clustered indexes, 114, 130
  drawbacks of, 114
clustering data with partitioning, 120
clustering index, 114
coalesce( ) function, 300
coarse (granularity), 34
Codd, E.F., 76
coding offensively with SQL, 48
columns, 2
  auto-incremented, 235
    effects on contention, 245
  Boolean, qualifying, 14

locking, 232
rows that should have been, 281–284
single, that should have been something else, 289–294
that should have been rows, 284–289
comments, identifying programs and critical modules, 28
commercial off-the-shelf (COTS) software package, 323
commit statements, 34
committing, locking and, 236
comparisons, 43
complexity
degree for the request, performance and, 230
introduced by storage options other than the default, 124
sources of hidden complexity, 329
composite primary keys, 70
order of columns in, 158
concurrency, 226–246
considering in SQL code design, 88
data modifications, 231–246
contention, 240–246
locking, 232–239
database engine as service provider, 226–231
increasing load revealing performance problems, 227
indexes, virtues of, 226
data-driven partitioning and, 119
increased, with partitioning, 115
concurrent updates, foreign key indexing for, 69
conditional logic, 42
conditions
applied at the wrong place, 98
order of evaluation, 89
(see also criteria; filtering conditions)
connect by operator (Oracle), 172, 178, 181
propagating percentages across different tree levels, 196
substituting materialized path model for, 189
constraints
implicit, unsoundness of, 17
major impact of, 17
violation of, 50
containers, contention when trying to access, 241
content lists, indexes and, 59
contention, 88, 240–246
architectural solutions, 242
DBA solutions for, 241
developmental solutions, 243
indexing system-generated primary keys, 71
insertion and, 240
physical layout of data and, 108
results from measures limiting, 243–246
correctness of data, 6
correlated subqueries, 94, 100
determining when to use, 137
looking for rows with no matching data, 162
performance effects when processing huge numbers of rows, 147
testing for existence without other search criteria, 207
un-correlating, 100, 158
volume increases and, 256–261
corruption of data, 10
(see also data corruption)
cost-based optimizers (CBOs), 36
COTS (commercial off-the-shelf) software package, 323
counts
redundant, 41, 49, 310
using as test for existence, 163
CPU, excessive use of, 312
CPU-intensive operations, high level of concurrency for, 88
credit card validation procedures, 200–202
criteria
defining result sets, 84
dynamic search criteria, 208–223
quality of, 330
(see also classic SQL patterns; conditions; filtering conditions), 223
current table and historical table, using, 21
(see also tables)
current values, 160
cursor loops, 310
customer, defining, 7

D

data containers, contention when trying to access, 241
data corruption, 10
data definition language (DDL), 33
data duplication
detection of duplicate primary keys, 50
minimizing with normalization, 10
data entry errors, 6
data flow, 22
data manipulation language (DML), commit statements, 34

data manipulation operations, ranking in terms of overall cost, 263
data modeling, 4, 25
historical data, 19
data pages
accessed by database engine, keeping as low as possible, 108
number visited by DBMS during a query, 103
number you are hitting, 312
data purges, 263
data redundancy, 8
data volumes (large), coping with, 248–278
data warehousing, 264–278
increasing volumes, 248–264
partitioning as solution, 262
sensitivity of operations to, 250–261
sudden increase in volume, 309
Data Warehouse Toolkit, The, 264
data warehousing, 264–278
cautions about, 277
data extraction, 268
integrity constraints and indexes, 270
loading data, 269
querying dimensions and facts, 270–273
star transformation, 273
emulating, 274–276
transformation, 268
database access libraries, 202–205
database accesses
maximizing usefulness of, 36
minimizing, 317–319
multiplying, 310
database connections, managing use of, 29
database links, 205
performance and, 205
database optimizers (see optimizers)
databases
conflicting goals in optimizing physical layout of data, 108
locking entire database, 232
reorganizations of, 132
SQL and, 76–79
structural types, 106–107
data-driven partitioning, 116, 262
concurrency problems and, 119
true partitioning, 118
date arithmetic, 45
date type, 64
dates and times, 64
comparing dates, 43
partitioning, 262
partitioning historical data tables by date, 116
simple or range searching on dates, 156–161
datetime type, 64
DB2
call to get new sequence value, 236
clustering index, 114
range-clustering, 118
recursive with statement, 172, 179
DBA (database administrator), solutions to contention, 241, 244
DBMS
closeness to kernel, 37–40
partition, different meanings of, 115
dbms_application_info package (Oracle), 28
DDL (data definition language), 33
decision support systems (DSS)
interaction with production databases, 265
query tools, 266
(see also data warehousing)
declarative language, 79
declarative processing, procedural vs., 35
decode( ) function, 42
delete operations, 263
against a database, 34
ranking in terms of overall cost, 263
denormalization, 20
caused by ready-made solutions, 32
dependencies, analyzing, 8–11
Boolean columns, 14
checking attribute independence, 9
checking dependence on whole key, 8
data replications and, 13
depth, hierarchical data, 169, 179, 294
design
irredeemable failure caused by, 249
performance and, 21
developmental solutions to contention, 243
dimension tables, 265
joining dimensions to fact tables, 273
querying, 270
dimensional modeling, 265
caution with, 277
facts and dimensions, 265
querying dimensions and facts, 270–273
SQL implications, 270
(see also data warehousing)
directives to the optimizer, 144–145, 305, 329
disk addresses, indexes referring to, 80
distinct, 91
avoiding at the top level, 93
implicit, 95
regular join with, 137

distributed systems, 205–208
DML (data manipulation language), commit statements, 34
double conversion, 302, 303
DSS (see decision support systems)
duplicate data
    detection of duplicate primary keys, 50
    minimizing with normalization, 10
duration, determining without dedicated interval data type, 66
dynamic queries, 117, 309
dynamic search criteria, 208–223
    defining movie database and main query, 209–216
    mistakes common in queries with, 223
    redesigning main query to fit criteria tightly, 216
    wrapping SQL in PHP, 217–222

E

efficiency
    of filtering conditions, 84, 90
    of searches, descriptions and, 6
    use of SQL, x
    (see also performance)
ELSE logic, obtaining, 42
encapsulation of database accesses, how not to, 202–205
entry points, identifying, 56–59
errors, data entry, 6
evaluating filtering conditions, 90–98
evolutionary database model, 107
except operator, 163, 164
exception handling
    cost of, 52
    forcing use of procedural logic, 53
exceptions, judicious use of, 50–53
excessive flexibility, dangers of, 18
execution plans, 319–330
    forcing the right plan, 323–328
    identifying the fastest, 320–322
    using properly, 328–330
existence test, 93
    correlated subquery without other search criteria, 207
    within subquery, 95
explain command, 142
explode( ) operator, 169
exploding a materialized path, 293
explosion of links, 193
expressions, complex SQL expressions, 88
extending DBMS products, 37
extraction of data, 268

F

fact dimension, 281
fact tables, 265
    joining to dimensions, 273
    querying, 270
    querying star schema through, 276
federated systems, 205
fifth normal form (5NF), 5
filtering conditions, 84, 89–103
    dynamically concatenated, 216
    evaluation of, 90–98
    large quantities of data, 98–102
    meaning of, 89
    proportions of retrieved data, 103
    queries returning a few rows from direct, specific criteria, 129
financial structures, risk exposure calculations, 170
fine (granularity), 34
first normal form (1NF), 8
fixed, inflexible database model, 106
flexibility (excessive), dangers of, 18
foreign keys, 17
    indexes and, 67–69
    integrity constraint in master/detail relationship, 169
    multiple indexing of the same columns, 69
    referencing underlying tables in partitioned view, 117
    (see also primary keys)
free lists, 242
from clauses
    nested queries in, 144
    uncorrelated subqueries rewritten as inline views, 96
    uncorrelated subquery in, 138
full table scans, 135, 146
    indexes vs., 109
    on tables expected to grow, 102
functions
    added to DBMS products, 37
    aggregate, 150–156
    built-in, advantages over external functions, 37
    indexes with, 62–66, 129
        appropriate use of, 66
        implicit conversions and, 64
    OLAP, operating on sliding windows, 149–150
    user-defined, 44, 146

G

global architecture, choice of, 249
good performance, defining, 311–317
  checking against standards, 315–316
  knowing what you get, 312–314
  knowing what you spend, 312
  performance goals, 316
granularity, 34
  of locking, 232, 239
  required by decision support systems, 268
  table of contents vs. index, 59
greatest( ) function, 43
group by clauses
  filtering and, 99
  use in aggregate statements, 156

H

hardcoding, 203, 310
hardware
  balancing programming mistakes with power, 248
  failures of, 24
hash indexes, 72
hash joins, 140, 147, 164, 262, 272
hash-partitioning, 118
having clause
  filtering after group by clause, 99
  use in aggregate statements, 155
head counts, modeling, 190
  computing head counts at every level, 192–194
heap-organized table, index-organized table vs., 112
hierarchical data
  depth (see depth, hierarchical data)
  practical example of hierarchies, 170
  suggested explode( ) operator, 169
  walking a tree in SQL, 177–189
    aggregating values from trees, 190–198
    bottom-up walk, 185–189
    top-down walk, 178–185
  walking hierarchies, 173
hierarchical databases, 168
hierarchical ordering of tables with IOTs and clustered indexes, 114
hints to the optimizer, 144–145
historical data, 156–161
  current values, obtaining, 160
  design of tables storing, 157
  difficulties of working with, 19–21
  many historical values per item, 160
  many items with few historical values, 157
  partitioning tables by date, 116
historical table, using, 21
  (see also tables)
host language, SQL statements embedded in, 42
hot spot in an index tree, 72

I

I/Os (increased), relieving over-worked CPUs, 88
implicit constraints on data, 17
implicit rules, 13
implicit type conversions, 129
  indexes and, 64
inconsistency of data, 10, 13
incorrect results, difficulty of spotting, 93
increasing data volumes (see large data volumes, coping with)
indeterminate condition, 11
index range scan, 64
indexes, 330
  adding to DMBS to tune it, 248
  bitmap, 273
  clustered vs. non-clustered index performance, 114
  completing queries with data returned from, 146
  content lists and, 59
  contention within, improving, 244
  correlated subqueries and, 94
  costs of, 56
  costs of maintaining, 108
  as data repositories, 109–113
    full table scans vs., 109
    storing maximum data possible, 110
  data access at atomic level of granularity, 60
  data loading and, 270
  data warehousing, 276
  defining row order in tables, 114
  dimension tables, 271
  dimensional model, 277
  finding physical location of a row, 62
  foreign keys and, 67–69
  with functions, 62–66, 129
    appropriate situations for use, 66
    implicit conversions and, 64
  maintenance costs, 57

making them work, 60
location of rows associated with index key, 61
multiple indexing of a column, 69
necessity of using, despite costs, 59
optimizer and, 79
partitioning, 115
performance and, 309
physical design of, 109
production database tuning and, 22
proportions of retrieved data, 103
queries returning small result set from very specific criteria, 129–137
criterion indexability, 132–137
data dispersion, 130–132
index usability, 129
query efficiency and index usage, 130
reference to disk addresses, 80
reverse, 71
solving contention problems, 243
searching, 92
system-generated keys, 70
temporary tables, 34
transactional databases, 59
variability of accesses, 72
virtues of, 226
index-organized tables (IOTs), 112
drawbacks of, 114
forcing row ordering, 113
solving contention problems with, 243
indirect criterion, 140
ingredients, identifying, 170
names and proportions in various products, 195–198
inline views, 135
rewriting uncorrelated subqueries as, 96
rewriting uncorrelated subqueries as joins in, 101
Inmon, Bill, 264
inner queries, 94
input, primitive, 130
insensitivity to volume increases, 250
insert statements
dynamic, built for varying table names, 117
returning ... into ... clause, 236
insertion rates
obtained with contention-limiting measures, 243
regular table vs. index-organized table, 112
insertions
contention and, 240
into referencing tables, preventing, 68
ranking in terms of overall cost, 263
integrity checking, taken out of DBMS kernel, 117
integrity constraints
data loading and, 270
foreign key constraint in master/detail relationship, 169
partitioned view and, 117
validation performed through, 206
inter-process communications, caused by use of procedural logic, 35
inter-process database link, 205
intersect operator, 164
intersection of result sets
final result set, small intersection of broad criteria, 138–140
intermediate result sets, 85
interval data type, 66
IOTs (see index-organized tables)
IP address database link, 205

J

joins
Cartesian, 285, 286
delaying till query end, 254–256
filtering conditions, 89, 92
older join syntax, 92
overlooked join condition, hidden by distinct, 93
pattern of, deciding factor, 93
performance and, 266
pre-joining tables, 123
problems with, 86
regular join with a distinct, 137
rewriting uncorrelated subqueries as joins in inline views, 101
self-joins on one table, 147–150
of two remote tables, 208
unions of complex joins, 98
(see also hash joins; nested loops)

K

Kent, William, 147
key values (index), querying indexes without full key, 110
Kimball, Ralph, 264

L

lag( ) OLAP function, 150
large data volumes, coping with, 248–278
  data warehousing, 264–278
  increasing volumes, 248–264
    partitioning as solution, 262
    sensitivity of operations to, 250–261
  sudden increase in volume, 309
last_insert_id( ) (MySQL), 235
latching, 88
  (see also locking)
LDAP (Lightweight Directory Access Protocol), 168
leading bytes, using for index queries, 110
leaf nodes, 168
  aggregation of values stored in, 190–194
    computing head counts at all levels, 192–194
    modeling head counts, 190
least( ) function, 43
legacy system data, 248
  putting into production, 249
levels in trees, 171
Lightweight Directory Access Protocol (LDAP), 168
like operator, 201
limitation criteria, 80
linear sensitivity to data volume increases, 251
linked server, 205
listener program, contacting in database connections, 30
list-partitioning, 119
lists
  content lists, indexes and, 59
  querying a list variable, 294–297
  selecting rows matching several items, 301–303
load, 310
  database load, main indicators of, 312
  increase in server load, 316
  relating to execution of SQL statements, 313
loading data, 269
locking, 88, 232–239
  causing sudden localized slowness, 309
  committing and, 236
  granularity of, 232
  lock handling, 234–236
  scalability and, 238
logic, programming into queries, 42
loop-back database link, 205
loops, 42
  not executing queries in, 227
  SQL statements executed in, 314
low cardinality, 109

M

many-to-many relationship resolving between two tables, 69
master/detail relationships, 168
matching, finding the best match, 304
materialized path model (SQL trees), 172, 175
  aggregating values stored in leaf nodes, 191
    computing head counts at every level, 193
  bottom-up tree walk, 186
  path explosion, 293
  substituting for connect by or recursive with, 189
  top-down walk, 182–183
materialized views, 33
  pre-joining vs., 123
measurement values, storage of, 265
memory
  addresses for data storage, 109
  corrupting by mishandling pointers, 37
merge statement, Oracle 9*i*, 41
merge table (MySQL), 116
meta-design, 281
mini-dimensions, 270
modeling, 4
  head counts, 190
    computing at every level, 192–194
  (see also data modeling; dimensional model; relational model)
modifying data (see changing data, concurrency and)
modularity, database programming and, 45
monitoring performance, 308–331
  defining good performance, 311–317
    checking against standards, 315–316
    defining performance goals, 316
    knowing what you get, 312–314
    knowing what you spend, 312
  execution plans, 319–330
    forcing the right plan, 323–328
    identifying the fastest, 320–322
    using properly, 328–330
  server load, 310

slow database, 308–310
statements currently being executed, 42
thinking in business tasks, 317–319
what really matters in improving queries, 330
"more-flexible-than-thou" construct, 18
movie database, 209–223
designing database and main search query, 209–216
redesigning main search query for tight fit, 216
wrapping SQL in PHP, 217–222
MySQL
last_insert_id( ), 235
merge table, 116
PHP, using with, 209

N

nested "containers", 170
nested interval model (SQL trees), 173
nested loops, 24, 140, 142, 144
nested queries, 135
in from clause, 144
nested set model (SQL trees), 172, 176–177
aggregating values stored in leaf nodes, 190
bottom-up tree walk, 188
top-down walk, 183
network problems, 308
insufficient speed or bandwidth, 310
"next value to use" table, 235
nine situations, 128
(see also classic SQL patterns)
nodes
relational view of a tree, 169
tree representing a hierarchy, 168
(see also leaf nodes)
non-linear sensitivity to data volume increases, 251–254
non-relational layer of SQL
applying last, 331
limiting thickness of, 256
OLAP functions, 159
normalization, 4–11
atomicity, 5
checking attribute independence, 9
checking dependence on the whole key, 8
data warehousing and 3NF design, 265
ensuring atomicity, 5
not exists ( ), using with a correlated subquery, 162
not in ( ), using with uncorrelated subquery, 161, 165, 166
null returns, 85
null values, 11–14, 166
indicating need for subtypes, 16
numerical values, comparing, 43

O

object-oriented (OO) practice, relational database processing vs., 37
offensive coding with SQL, 48
OLAP functions
current value for an item at a given date, 159
operating on sliding windows, 149–150
row_number( ), 180
"one size fits all" philosophy, 223
online analytical processing (OLAP), DB2, 52
online transaction processing (OLTP), 22
operating mode, 22
operating systems, contention issues and, 108
operational data stores, 265
operations (data manipulation), ranking in terms of overall cost, 263
operations, sensitivity to data volume increases, 250–261
disentangling subqueries, 256–261
insensitivity to, 250
linear sensitivity to, 251
non-linear sensitivity to, 251–254
optimistic concurrency control method, 49
optimizers, 77, 79
causing to take a different course, 329
checking execution plan, 142
circumstances not allowing efficient working of, 330
data distributions and, 161
directives, 305
directives or hints to, 144–145
heterogeneous, on distributed systems, 207
join filtering conditions and, 89
joins and filtering conditions, 92
limits of, 83
queries and, 79
rewriting of queries, 98
views and, 87

Oracle
connect by operator, 172, 178, 181
propagating percentages across tree levels, 196
substituting materialized path model for, 189
dbms_application_info package, 28
rownums, 80, 87
tablespace referred to as a partition, 115
Oracle 9*i* Database, merge statement, 41
order of evaluation, filtering conditions, 89
ordering criteria, 80
ordering information, relations vs., 81
(see also sorts)
orders/order_detail relationship, 168
outer joins
expressing nonexistence with, 165
using in dynamically defined search query, 216
outer queries, 94
outriggers, 270

P

pages, 108
locking, 232
pre-joined tables and, 124
parallelism
adjusting to solve contention problems, 243
increased, with partitioning, 115
processing very large volumes of information, 147
parallelized queries, 207
parent/child link, master/detail relationship vs., 168
parents, multiple, 169
Parkinson's Law, 310
partition clause, 149
partition key, 118
partition pruning, 118
partitioned view, 116
partitioning, 115–123, 330
archival and data purges, 263
data distribution and, 120
data-driven, 116
determining best way to partition data, 121–123
methods of, 118
round-robin, 116
scattering or clustering data, 119
solution for contention, 242
solving data volume problems, 262
partitions, 130
varying meanings among DBMS systems, 115
Pascal, Fabian, 169
patterns (see classic SQL patterns)
percentages, propagating across different tree levels, 194–198
performance
bottom-up tree walk, comparing for various models, 188
clustered vs. non-clustered indexes, 114
committing and, 237
computing head counts from aggregation of information in leaf nodes, 194
costs of excess flexibility, 18
database connections, minimizing, 29
database engine, hardware, and I/O subsystems, 230
database links, costs of, 205
design and, 21
improving queries, what really matters, 330
joins and, 266
monitoring (see monitoring performance)
non-relational layer of queries, 82
procedural logic and, 36
request complexity and, 230
results from contention-limiting measures, 243–246
solving problems with, 280–306
aggregating by range (bands), 297–299
columns that should have been rows, 284–289
columns that should have been something else, 289–294
finding the best match, 304
optimizer directives, 305
querying a list variable, 294–297
rows that should have been columns, 281–284
selecting rows matching several list items, 301–303
superseding a general case, 299–301
statements arriving faster than they are serviced, 231
top-down query, comparing various models, 184
transparent references to remote data, 23
tuning vs., x

persistency layers, abstract, 202–205
PHP, 209
 wrapping SQL in, 217–222
physical layout of data
 conflicting goals in optimization attempts, 108
 forcing row ordering, 113
 index performance and, 130–132
 indexes as data repositories, 109–113
 partitioning, 115–123
  best way to partition, determining, 121–123
  data distribution and, 120
  scattering or clustering data, 119
 pre-joining tables, 123
 process requirements and physical design, 109
 sacrificing simplicity with strong structuring, 124
pivot operator, 288
pivot tables
 binding a list, 303
 creating, 285
 multiplying rows, 286
 passing list of values as single string to a statement, 294–297
 using values, 286
place-holders (bind variables), 162
pointers, manipulation of, 37
Practical Issues in Database Management, 169
primary key index
 clustering index, using as, 114
 insertion rate for IOT vs. regular table, 112
 limiting contention within, 245
primary keys
 composite, 70
  order of columns, 158
 defining, 7
 detection of duplicates, 50
 indexing of, 59
 subtype relationships, 16
 system-generated, indexing of, 70
 values in operational database vs. dimension identifiers, 269
 whole key dependence and, 8
 (see also foreign keys)
primitive input, 130
principles, 3
probabilistic basis, coding on, 48
problems, defining before solution, 32
procedural logic
 achieving in database applications, 42
 exception handling and, 53
 in SQL, reasons for shunning, 35
procedural processes within a business, paying too much attention to, 31
procedural processing, declarative vs., 35
processes
 contention between (see contention)
 physical database layout and, 35
 spawned by listener in database connections, 30
processing flow, 22
production databases, 265
program variable for sequence value, 236
purging data, 263

Q

queries, 3, 76–103
 answered by returning index data, 110
 blanket views, performance impact of, 117
 complex, and complex views, 86
 distributed, 206
 distributed and parallelized, 207
 dynamic, 309
  for variable number of search criteria, 214–216
  mistakes commonly made in, 223
 estimating behavior with increased data volumes, 254
 expression of, association with implicit assumptions about data, 98
 functionally equivalent, comparison to synonyms, 96
 identifying, 28
 improving, what really matters, 330
 limits of the optimizer, 83
 making as fast as possible, 108
 nested, 135
 non-relational aspects of, 81
 number of data page hits by DBMS during performance of, 103
 order of evaluation of conditions, 90
 performance, whole key dependence and, 9
 programming logic into, 42
 relational component, 80
 relational layer, doing maximum work in, 82
 returning a very large amount of data, 102
 rewriting by the optimizer, 98
 size of result set, 85
 SQL expression of, 77
 tuning, 22
 various layers of, 78
query tools, 266

R

random numbers, using instead of system-generated values, 243, 244
range scans, 130
  on clustered data, 113
  converting variable-length comparison to common case, 200–202
  reverse indexes and, 71
  simple or range searching on dates, 156–161
range-clustering (DB2), 118
range-partitioning, 118
ranges
  aggregating by range (bands), 297–299
  importance of well-defined boundaries, 262
ranking functions (SQL Server), 52
recovering databases, 24
recursive with statement, 172, 179
  adjacency model, top-down tree walk, 180
  propagating percentages across different tree levels, 196
  substituting materialized path model for, 189
redundant data, 8
reference data in dimension tables, 265
referencing tables, preventing insertions into, 68
relational databases, 76
  hierarchical databases vs., 114
  processing, confusing with object-oriented methods, 37
  SQL and, 76
relational model, 2
  coherence of, 3
  flexibility of, sacrificing by strongly structured data, 125
  two-valued logic, 11
  view of a tree, 169
relational operations, reporting requirements vs., 78
relational theory, 77
relations, 3
  associating large numbers of possible characteristics in, 11
  ordering information vs., 81
remote data
  querying, 24
  transparent references to, 23
remote data sources, 205–208
remote validation checks, 206
reorganizations of databases, 132
reporting requirements, 77
request type, partitioning by, 122
requirements, evolution of, 10
response times, 147
  (see also performance)
result sets
  criteria defining, 84
  difficulty of spotting incorrect data, 93
  filtering conditions, 89–103
    evaluation of, 90–98
    large quantities of data, 98–102
    meaning of, 89
    proportions of retrieved data, 103
  large, 146
  obtained by aggregation, 150–156
  predicated on absence of data, 161–166
  size of, 85, 330
  small intersection of broad criteria, 138–140
  small intersection, indirect broad criteria, 140–145
  small result set, indirect criteria, 137
  small, from direct, specific criteria, 129–137
retrieval ratios, 60
returning ... into ... clause, 236
reverse indexes, 71
  solving contention problems, 243
right-padding function (rpad( )), 202
risk exposure in a financial structure, 170
round-robin partitioning, 116
row_number( ) OLAP function, 149, 159, 180
rownums (Oracle), 80, 87
rows, 330
  associated with index key, physical closeness of, 61
  columns that should have been rows, 284–289
  emptying a table of all rows, 33
  locking, 232, 238
  matching several list items, selecting, 301–303
  ordering of, forcing, 113
  physical location, finding with an index, 62
  primary key, defining, 7
  proportions of retrieved data, 103
  that should have been columns, 281–284
  updating and inserting, dedicated statements for, 41
rpad( ) function (right-padding), 202

S

scalability, locking and, 238
scattering data with partitioning, 120
schemas
  classical order schema, 91
  movie database (example), 210
  (see also star schema)
science, art vs., xi
searches
  dynamically defined criteria, 208–223
    designing movie database and main query, 209–216
    mistakes common in queries, 223
    redesigning query for tight fit with criteria, 216
    wrapping SQL in PHP, 217–222
  efficiency, descriptions and, 6
second normal form (2NF), 9
select distinct queries, 9
select operator, filtering conditions, 89
selectivity of an index, 61
self-joins, 147, 282
  performance and, 282, 284
semantic inconsistency, 13
sensitivity of operations to volume increases, 250–261
  disentangling subqueries, 256–261
  insensitivity to, 250
  linear sensitivity to, 251
  non-linear sensitivity to, 251–254
sequences
  call to database for new value, 236
  not using in order to limit contention, 245
server load, 310
  increase in, 316
servers, 205
set operators, 163–165
  assembling data from several sources, 268
  getting rid of unwanted data quickly, 98
sets
  nested set model, SQL trees, 172, 176–177
    bottom-up walk, 188
    top-down walk, 183
  processing in SQL, 34
  relational theory and, 78
slow database, 308–310
  it's not the database, 308
  particularly slow query, 309
  slow performance degradation reaching a threshold, 309
  sudden global sluggishness, 308
  sudden localized slowness, 309
snapshots, 33, 314
solutions (ready-made), problems caused by, 32
sorts, 80
  volume increases and, 251–253
    delaying joins to end of query, 254–256
spreading data across many servers, 23
SQL
  art of, governing factors, 84–88
    number of tables, 85
    number of users, 88
    result set criteria, 84
    result set size, 85
    total quantity of data, 84
  classic patterns (see classic SQL patterns)
  efficient use of, x
  general characteristics of, 76–83
    relational and non-relational aspects, 80
    SQL and databases, 76–79
    SQL and the optimizer, 79
  wrapping in PHP, 217–222
SQL Communication Area (SQLCA), 41
SQL engine cache, 314
SQL Server
  clustered index, 114
  pivot and unpivot operators, 288
  recursive with statement, 172
star schema, 265
  querying tables, 271
  querying through facts and dimensions, 276
star transformation, 273
  emulating, 274–276
statements
  action-packed, 35
  first questions to consider when writing, 90
  mirroring business logic, 42
  relating load to execution of, 313
  succinct, 46
statistical functions, 77
statistics, automated collection of, 34
status, partitioning by, 122
storage
  options other than default, introducing complexity with, 124
  peculiarities in, 330
  temporary, 82
stored procedures, 10, 310

strategy, defining tactics with, 31
strings
comparing, 43
extracting individual characters and returning them on separate rows, 290–293
structural types, databases, 106–107
subqueries
correlated or uncorrelated, deciding between, 137
sensitivity of operations to data volume increases, 254
use in processing massive numbers of rows, 147
value of an item on a given date, 157
where clauses, 84
(see also correlated subqueries; uncorrelated subqueries)
subtypes, 15
defining to deal with varying numbers of attributes, 281
succinct statements, 46
summary tables, pre-joining vs., 123
Sybase, clustered index, 114
syllogisms, 328
synchronization of databases after recovery, 25
synchronous processing, 22
synonyms, comparison to functionally equivalent queries, 96
system
changes in, causing sudden global slowness, 309
complexity of, 24
database connections, 30
distributed, 205–208
tuning, 21
system-generated keys, 70
system-generated values, 235
not using in order to prevent contention, 243, 244

T

table of contents, index vs., 59
tables, 2
current table and historical table, using, 21
enabling data retrieval as with table of contents, 60
forcing row ordering, 113
improving contention within indexes, 244
index-organized table (IOT), 112
locking, 232, 238
number involved in a query, 85
partitioning, 115
physical design of, 109
pre-joining, 123
remote, joins, 208
single table in hierarchical tree structure, 168
tables that are views, 106
valuation, 19–21
tablespace (Oracle), referred to as a partition, 115
tactics, defined by strategy, 31
target volumes for database systems, 249
temporary storage, 82
temporary tables, 264
disadvantages of using, 34
third normal form (3NF), 4, 9
data warehousing and, 265
threads spawned by listener in database connections, 30
three-valued logic (implied by nulls), 12
"tight-fit" query, 216
time information, 64
timestamps, 148
reverse indexing and, 72
top-down tree walk, 178–185
adjacency model, 178–182
aggregating values stored in leaf nodes, 190
materialized path model, 182–183
nested set model, 183
performance for various models, 184
transaction space, 242
transactional databases
indexes, 109
indexing requirements, 59
transactions
good practices in, 234
across heterogeneous systems, 206
locking and committing, 236
Transact-SQL, @@IDENTITY system variable, 235
transformations, 268
mathematical equivalence of, 83
star transformation, 273
emulating, 274–276
transparent references to remote data, 23
tree structures, 168–171
aggregating values from trees, 190–198
propagating percentages across levels, 194–198
values stored in leaf nodes, 190–194

hierarchies, practical examples of, 170
of indexes, 62
master/detail relationships vs., 168
materialized path model, exploding, 293
practical implementation of trees, 174–177
adjacency model, 174
materialized path model, 175
nested set model, 176–177
representing trees in SQL database, 172–174
walking a tree in SQL, 177–189
bottom-up walk, 185–189
top-down walk, 178–185
triggers, 10, 330
index maintenance costs vs., 58
Tropashko, Vadim, 173
truncate operations, 33
delete vs., 263
truths, 3
freedom in the choice of, 4
tuning, 21
adding indexes, 248
performance vs., x
two-valued logic, 11
types
implicit conversions, 129
partitioning by, 122

U

uncorrelated subqueries, 94, 158
in classic style or in from clause, 138
not in ( ), using with, 162
rewriting as a join in an inline view, 101
rewriting as inline views in from clause, 96
sensitivity of operations to data volume increase, 254
testing for existence without other search criteria, 207
union operator, 164
querying large quantities of data, 98
unions
of complex joins, 98
of intermediate result sets, 85
overhead of querying large union view, 117
partitioned tables, 116
uniqueness, enforcement of, 50, 117
unnecessary coding, avoiding, 41
unpivot operator, 288
update statement, returning ... into ... clause, 236
updates
against a database, 34
combining multiple into one, 43
concurrent
foreign key indexing, 69
costly massive updates, 314
locking and scalability, 238
making as fast as possible, 108
multiple massive updates to a table, 268
optimistic concurrency control, 49
ranking in terms of overall cost, 263
(see changing data, concurrency and)
useless queries, 310
user-defined functions, 44, 146
users
not complaining about performance, 315
number of, concurrency and, 88
perception of performance improvement, 316

V

valuation, 19
valuation tables, 19–21
variable number of search criteria, mistakes in queries containing, 223
variables, binding, 294, 296
PHP, 218–222
views, 3, 329
complex, 86
partitioned view, 116
tables as, 106
where clauses, 84
volume of data (see large data volumes, coping with)

W

warehousing data (see data warehousing)
where clauses
in aggregate statements, 156
atomic attributes, 6, 64
filtering conditions, 84, 89
filtering conditions independent of the aggregate, 99
join operator, 89
two-valued logic, 11
with statement, recursive (see recursive with statement)

X

XML, 168, 290

[General Information]
书名=SQL语言艺术_12022766
SS号=11805268

封面
书名
版权
前言
目录
前言
制定计划：为性能而设计
数据的关系视图
规范化的重要性
有值、无值、空值
限用Ｂｏｏｌｅａｎ型字段
理解子类型（Ｓｕｂｔｙｐｅ）
约束应明确声明
过于灵活的危险性
历史数据的难题
设计与性能
处理流程
数据集中化（Ｃｅｎｔｒａｌｉｚｉｎｇ）
系统复杂性
小结
发动战争：高效访问数据库
查询的识别
保持数据库连接稳定
战略优先于战术
先定义问题，再解决问题
保持数据库Ｓｃｈｅｍａ稳定
直接操作实际数据
用ＳＱＬ处理集合
动作丰富的ＳＱＬ语句
充分利用每次数据库访问
接近ＤＢＭＳ核心
只做必须做的
ＳＱＬ语句反映业务逻辑
把逻辑放到查询中
一次完成多个更新
慎用自定义函数
简洁的ＳＱＬ
ＳＱＬ的进攻式编程
精明地使用异常（　Ｅｘｃｅｐｔｉｏｎｓ）
战术部署：建立索引
找到“切入点”
索引与目录
让索引发挥作用
函数和类型转换对索引的影响
索引与外键
同一字段，多个索引
系统生成键
索引访问的不同特点
机动灵活：思考ＳＱＬ语句
ＳＱＬ的本质
掌握ＳＱＬ艺术的五大要素
过滤
了如指掌：理解物理实现
物理结构的类型
冲突的目标
把索引当成数据仓库
记录强制排序
数据自动分组（Ｇｒｏｕｐｉｎｇ）
分区是双刃剑
分区与数据分布
数据分区的最佳方法
预连接表
神圣的简单性
锦囊妙计：认识经典ＳＱＬ模式

小结果集，直接条件
小结果集，间接条件
多个宽泛条件的交集
多个间接宽泛条件的交集
大结果集
基于一个表的自连接
通过聚合获得结果集
基于日期的简单搜索或范围搜索
结果集和别的数据存在与否有关
变换战术：处理层次结构
小结果集，直接条件
小结果集，间接条件
多个宽泛条件的交集
多个间接宽泛条件的交集
大结果集
基于一个表的自连接
通过聚合获得结果集
基于日期的简单搜索或范围搜索
结果集和别的数据存在与否有关
孰优孰劣：认识困难，处理困难
看似高效的查询条件
抽象层
分布式系统
动态定义的搜索条件
多条战线：处理并发
数据库引擎作为服务提供者
并发修改数据
集中兵力：应付大数据量
增长的数据量
数据仓库
精于计谋：挽救响应时间
数据的行列转换
基于变量列表的查询
基于范围的聚合
一般规则，最后使用
查询与列表中多个项目相符的记录
最佳匹配查询
优化器指令
明察秋毫：监控性能
数据库速度缓慢
服务器负载因素
何谓“ 性能优良 ”
从业务任务角度思考
执行计划
合理运用执行计划
总结：影响性能的重要因素
Photo Credits
索引