Broadview®
www.broadview.com.cn

WILEY

数据库索引设计与优化

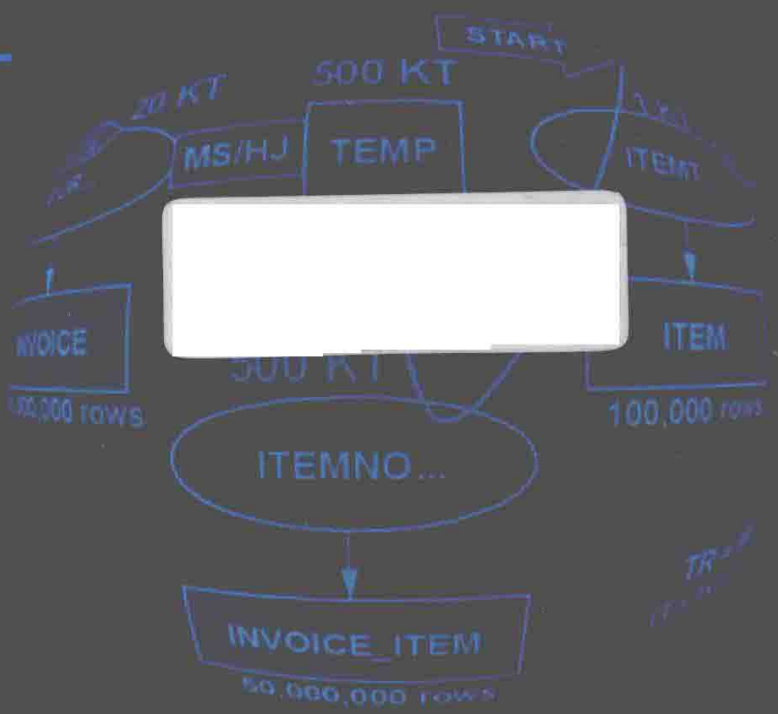
Relational Database Index Design and the Optimizers

[美] Tapio Lahdenmäki
Michael Leach 著

曹怡倩 赵建伟 译

MySQL

Oracle



SQL Server

et.al.

中国工信出版集团

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

数据库索引设计与优化

Relational Database Index Design and the Optimizers

[美] Tapio Lahdenmäki
Michael Leach 著

曹怡倩 赵建伟 译

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

内 容 简 介

本书提供了一种简单、高效、通用的关系型数据库索引设计方法。作者通过系统的讲解及大量的案例清晰地阐释了关系型数据库的访问路径选择原理，以及表和索引的扫描方式，详尽地讲解了如何快速地估算 SQL 运行的 CPU 时间及执行时间，帮助读者从原理上理解 SQL、表及索引结构、访问方式等对关系型数据库造成的影响，并能够运用量化的方法进行判断和优化，指导关系型数据库的索引设计。

本书适用于已经具备了 SQL 这一关系型语言相关知识，希望通过理解 SQL 性能相关的内容，或者希望通过了解如何有效地设计表和索引而从中获益的人员。另外，本书也同样适用于希望对新硬件的引入所可能带来的变化做出更好判断的资深人士。

Relational Database Index Design and the Optimizers, 9780471719991, Tapio Lahdenmäki, Michael Leach.
Copyright © 2005 by John Wiley & Sons, Inc.
All rights reserved. This translation published under license.
No part of this book may be reproduced in any form without the written permission of John Wiley & Sons, Inc. Copies of this book sold without a Wiley sticker on the back cover are unauthorized and illegal.

本书简体中文字版专有翻译出版权由美国 John Wiley & Sons, Inc.公司授予电子工业出版社。未经许可，不得以任何手段和形式复制或抄袭本书内容。本书封底贴有 John Wiley &Sons,Inc.防伪标签，无标签者不得销售。

版权贸易合同登记号　图字：01-2014-4721

图书在版编目（CIP）数据

数据库索引设计与优化 /（美）拉赫登迈奇，（美）利奇（Leach,M.）著；曹怡倩，赵建伟译. —北京：电子工业出版社，2015.6
书名原文: Relational Database Index Design and the Optimizers
ISBN 978-7-121-26054-4

Ⅰ. ①数… Ⅱ. ①拉… ②利… ③曹… ④赵… Ⅲ. ①数据库系统－研究 Ⅳ. ①TP311.13

中国版本图书馆 CIP 数据核字（2015）第 098122 号

策划编辑：张春雨
责任编辑：徐津平
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
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：720×1000　1/16　　印张：20　　字数：380 千字
版　　次：2015 年 6 月第 1 版
印　　次：2015 年 6 月第 1 次印刷
定　　价：79.00 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到 dbqq@phei.com.cn。

服务热线：（010）88258888。

推荐序一

2015 年，我很欣喜地看到这本书终于被真正的技术粉们翻译成了中文。

这本写作于 2005 年的著作，是少数能够穿越十年仍然历久弥新的经典著作之一。译者曹怡倩和赵建伟是支付宝的资深数据库专家，也是童家旺的得意门生，他们的经验和理解为本书增色良多。我在学习过程中也曾经从本书的英文版中获益，所以当旺旺委托我写一段文字时，我欣然受命，但是其实他才是这本书的知音，我曾经在他的多次公开演讲中听到他郑重推荐此书，并引用书中的方法达成实践。今天，两位作者的真知灼见跨越时间和空间，在中文世界里找到喜爱的读者们，这是作者们的欣慰，更是读者们的幸运。

在 RDBMS 的世界里，直到今天，索引仍然是无与伦比的重要环节——走过千山万水，优化仍然要回归到和索引纠缠不清的本质上来。作者在第 5 章提出“前瞻性的索引设计”，其中的观点之一是“设计索引，使所有查询语句都运行得足够快”，这让我想到 2011 年我们翻译的韩国著作《海量数据库解决方案》一书，李华植先生在书中同样提出了“战略性索引设计方案”，实在是具有异曲同工之妙，大家都着眼于如何使用更合理的索引去满足尽可能多的 SQL 访问需求。而为了解决开发环节中的不良 SQL 性能问题，云和恩墨甚至开发了一款专门的软件用于 SQL 审核，其中的多个维度指标仍然是考量索引的。

大道至简，从最基本的起点出发，才有可能从最根本上解决我们面对的复杂终极问题。当然，今天我们对于索引的关注已经不仅仅是有无，而已扩展到索引的各种访问方式，索引全扫描、索引跳跃式扫描等，细微之处更见功夫。

技术的另外一大迷人之处在于其无穷的细节变化和无限的可能。针对作者在第 1 章中提到的种种误区，我们在阅读时都可以细细地思量并引发探讨。比如作者提到的“误区 2 单表的索引数不要超过 6 个”，在很长一段时间内，我对此并没有深刻的认识和经验，直到 2010 年在一个客户的环境中遇到了非常棘手的问题，我的确史无前例地分析了索引个数对于 Oracle SQL 解析性能的影响。要知道，在硬解析的过程中，CBO 需要对所有可能发挥作用的索引进行非常详尽的成本计算和评估，这在 Oracle 中涉及了 ROW

CACHE 的算法管理，在串行 Latch 的守护下，如果索引的数量增多，无疑会降低解析的速度。我还意外地发现在 dc_object_ids 的使用上，效率不佳——你可能也注意到了，在 Oracle 11g 中，这个资源消失不见了。就是这么好玩，你总会发现例外。而且这一切发生在 SQL 开始执行之前，我们更多时候关注的是索引好不好、快不快，可能我们忽略了在 SQL 开始执行之前的解析环节中就可能存在无法规避的问题。

在 2008 年 Oracle 推出的 Exadata 一体机上，Oracle 更加喜欢在全扫描的过程中通过智能扫描（Smart Scan）进行数据卸载，而存储索引更是一个让人目瞪口呆的设计，你可能觉得这个基于内存的设计根本就不是索引！再然后，在 2014 年 Oracle 推出了 IMO 之后，Larry 居然宣布，有了这个特性赶紧去干掉一堆索引吧，这会让 OLTP 更流畅。

各种好玩啊！技术世界就是如此变化多端，曲曲折折！

你也必须觉得好玩，才能在技术的世界里找到快乐！

在这本书中，你一定能够找到点燃你技术热情的火花！

——盖国强（Oracle ACE 总监，云和恩墨创始人）

2015 年 4 月于北京

推荐序二

经过多年的发展，关系型数据库已经成为一个企业 IT 架构的核心，诞生了包含 Oracle、MySQL、DB2、Microsoft SQL Server、Sybase、PostgreSQL 这样的数据库。上述这些传统的数据库都强烈地依赖于索引的设计与规划。不夸张地说，一个应用是否合格的关键就在于数据库索引的设计与优化。

在工程领域，很多数据库工程师们都积累了对于索引设计与优化的经验，可惜很少有书籍系统地介绍这部分内容，这导致网上存在很多错误的观念。可喜的是本书详细介绍了一种简单、高效的数据库索引设计方法，可让读者对数据库的索引设计快速进阶。此外，本书通过大量的实例分析了关系型数据库的访问路径选择原理，对实践有着实际的指导意义。最为关键的是，这些原理与知识对于所有关系型数据库都是通用的，甚至对于目前出现的 NoSQL 数据库也是具有指导意义的。

本书对于想要理解数据库索引设计方法，以及数据库优化器访问路径选择原理的读者都具有现实的指导意义。希望将本书推荐给所有从事数据库相关工作的人员，它的内容不局限于具体的数据库类型，是一本修炼内功的好书，看完之后定能解决在工程中遇到的各种索引设计与优化问题。

姜承尧

网易杭州研究院数据库技术组负责人

推荐序三

一直以来，我们都将数据库行业鼻祖萨师煊老师主编的《数据库系统概论（第三版）》作为葵花宝典，每年都要对其核心章节读上 1~2 遍，也曾读过他人翻译的《数据库系统面向应用的方法》。总体而言这两本书在关系型数据库的基础知识和技术原理等方面基本一致，也有互补的技术内容，但关于索引设计与优化方面的内容尚有不足，国内外也未发现这方面较好的书籍。好友童家旺找我为其徒弟曹怡倩和赵建伟翻译的书籍作序，我看到书名《数据库索引设计与优化》时便欣然答应了，因为此书能填补国内的空白，为众多技术人员带去福音。

在阅读书籍全文内容的过程中，我深刻地体会到国际友人写书风格非常值得学习，全篇使用示例与公式一步一步分析、计算和验证，不仅得出经得起推敲的方法论，而且深入浅出地讲述了相关的技术原理，既体现出作者的严谨风格，又便于读者学习、记忆和理解。阅读过程中能发现本书的专业术语翻译得非常精湛，对原理、算法、方法论、技术常识等内容的描述，更让人读起来有读原著作品的口感，这体现了译者深厚的数据库理论功底，以及对本书内容的深刻理解，甚至已有超越原著的水准。

虽然近二十年中数据库理论知识未有任何质的变化，但计算机软硬件技术的飞速发展促进了多种数据库管理系统实现算法的革新。随着软硬件技术算法实现的变革，早期关于索引创建与优化的部分准则已不适应当下，故本书籍从索引相关的基础知识和早期建立准则误区出发，介绍了硬盘处理器内存与索引的关系、其中的技术原理和索引原理，以及不同数据库产品之间的异同、SQL 处理过程与索引的关联性、如何创建理想和前瞻性的索引、索引设计需要关注和平衡的点、不同应用场景下的索引设计技巧与方法论、如何规避索引的弊端、数据库管理系统的索引限制、如何让优化器运行更佳，总结了当下设计优秀索引的准则。

本书按照由浅入深的方式安排内容，清楚阐释了原著作者想要解释的两个问题：如何量化索引和优化器的成本，如何从 CPU 和执行时间的角度量化运行过程。

书中使用了 Oracle、DB2 和 SQL Server 这三种数据库的大量案例，介绍了它们的异同与各自的特性，绝大部分内容也同样适合于 MySQL 和 PostgreSQL 数据库产品。个人认为本书同萨老师主编的书籍一样可奉为葵花宝典，是值得 DBA 们人人收藏的一本书，也适合作为云计算平台研发自动化 SQL 审核与索引创建组件研发工程师的工具书。

金官丁

书于 2015 年 4 月 20 日

推荐序四

本书原著是数据库领域的重要理论大作，几年前童家旺先生就推荐过此书，在《高性能 MySQL》一书中也推荐过它，据说他还要求每位徒弟都熟读此书，可见其重要性。本书花了大量篇幅讲解索引基础的各种理论知识，以及如何综合评估索引使用的代价，而非简单介绍 SQL 如何编写，或者如何进行 SQL 优化等内容。认真通读本书之后，相信各位读者对数据库的索引设计、代价评估等方面就基本可以驾轻就熟了。本书经知含（曹怡倩）等人的悉心翻译，终于有了中文版，是国内广大 DBA 同行的福音，强烈建议购买阅读。

叶金荣

译者序一

记得第一次接触到这本书是在 2011 年的时候，那时我刚毕业参加工作不久，在公司里还处于学习知识和熟悉业务的阶段，当时带我的师傅童家旺先生经常会介绍一些他认为不错的书籍给团队的同事们阅读，这本 *Relational Database Index Design and the Optimizers* 就在他推荐给我们几位入职不久的新人的书单之中。我最初阅读的是书中第 3 章至 5 章的内容，读完后感到非常地痛快，只看一遍，所有的原理就都能清晰地呈现在脑海之中，于是随即将全书的其余章节也通读完成。从这本书中所学到的知识对我之后的工作带来了极大的帮助，使我在应用系统的索引设计及 SQL 调优上节省了大量的精力且取得了不错的效果。之所以能有如此成效，完全得益于 Tapio Lahdenmäki 和 Michael Leach 两位作者的贡献，是他们将宝贵的经验与智慧与广大读者分享。本书从表和索引结构开始，层层深入，通过大量的案例系统性地阐释了关系型数据库的访问路径选择原理，以及表和索引的结构、扫描方式、硬件等对性能的影响；详尽地讲解了如何快速地估算 SQL 运行的 CPU 及执行时间的方法，使读者能够运用量化的方法进行判断和优化，指导关系型数据库的索引设计，真正做到知其所以然，进而能够举一反三，灵活运用于各类场景之中。

当得知有幸能够参与本书中文版的翻译工作时，我们感到非常兴奋，迫切地想把它翻译出来与更多的朋友分享。但当真正开始着手翻译时，才体会到这其中的艰难以及身负的责任，以至于有段时间愈翻译愈感到焦虑，深恐译本无法传达作者之原意。对于翻译中遇到的一些疑难问题，有时会反反复复阅读相关内容数遍，查阅各种资料以尽力获得较为准确的译法。加之我和赵建伟工作都较为繁忙，最终的交稿时间远远超出了原定计划。在此特别感谢王中英编辑在此过程中的督促、鼓励以及对翻译疏漏之处的校正。

本书由蚂蚁 DBA 团队的多位同事共同翻译完成，其中第 4、7、10、12、14、16 章由曹怡倩翻译，第 3、9、13、15 章由赵建伟翻译，第 1、2、5、6、8、11 章分别由刘晓军、叶莹佼、曹栖锋、柯翔宇、蔺瑞超、樊振华翻译。全书校稿及前言、术语表部分的翻译由曹怡倩完成。感谢恩师童家旺先生在全书翻译过程中的解惑和帮助。

本书原文通俗晓畅，但因我们个人水平及时间的限制，译文中难免存在

不准确或是不通顺之处，这全是我们译者的责任，与作者及帮忙的朋友无关；如有识者，望不吝赐教。

感谢电子工业出版社编辑张春雨、王中英以及其他所有工作人员在各方面的鼎力支持与几近无限的耐心。感谢在本书翻译过程中给予了我们如此多帮助与支持的朋友、同事和家人们。

感谢阿里巴巴 DBA 二部团队同事和领导给予的支持与帮助。感谢恩师童家旺先生在全书翻译过程中的解惑和帮助。感谢父母一直以来的关心照顾和体谅，使我能够有更多的精力专注于本书的翻译工作。

曹怡倩

译者序二

记得 2010 年 7 月毕业进入支付宝数据库管理团队，担任开发 DBA 一职，阿里系各子公司是国内当时为数不多的在数据库团队划分开发 DBA 和运维 DBA 两种角色的公司，开发 DBA 的主要职责包括评估业务开发人员数据库使用的申请，SQL 审核，上线评估，运行监控，容量规划等，贯穿整个产品的生命周期，其中 SQL 审核尤为烦琐和重要，在业务迅猛发展和小机+存储架构流行的年代，一条烂 SQL 足以搞垮多条业务产品线，修复的代价比较大，周期也比较长。

接触了一段工作时间后，我的师傅童家旺先生推荐我阅读本书的英文原版 *Relational Database Index Design and the Optimizers*，当时拿到这本书的时候不以为然，觉得在关系数据库领域遇到 SQL 性能问题时，无非就是添加 B 树索引。但当我真正翻开这本书的时候，真的是爱不释手，一口气把它读完，仍意犹未尽。这本书不仅讲述了如何建立三星索引的方法论，更重要的是给出了基于硬件和软件环境下索引设计的量化评估的方法和实践，掌握了这些方法后，你将能够提前量化业务 SQL 上线运行情况，并指导后期的容量评估。这本书并没有限定具体的商业或者开源的关系型数据库产品，而是讲述通用的理论和方法。所以，强烈建议数据库从业人员，使用关系型数据库的开发人员阅读本书，它会给大家带来体系化的理论和方法。

在此，感谢原支付宝数据库管理团队的领导和同事们合力完成本书的翻译工作，同时要感谢我的家人，是你们不辞辛苦地照顾家庭，我才能够在工作之余专注于本书的翻译。希望大家享受这本书。

赵建伟

2015 年 5 月于杭州

前言

关系型数据库至今已存在了三十多年。在其发展早期，由于硬件资源限制及优化器成熟度的不足，性能问题非常普遍，因此性能成为了人们优先考虑的事项。但现在情况已经不同了，硬件及软件以超出人们想象的速度发展了起来，系统已经能够自己关心自己的性能了，这在之前看来是不可思议的！但比这些资源增长速度更快的是随之产生的大量信息以及这些信息所衍生出的活动。另外，有一个重要的硬件还没有跟上整体的发展速度：虽然磁盘已变得更大且异常廉价，但它们的访问速度仍相对较慢。因此，许多老问题其实并没有消失——它们只是变换了形式。这其中的有些问题可能会造成巨大的影响——那些所谓的应该只需运行不到一秒的“简单”查询实际却运行了几分钟或更久，尽管所有的书中都写了如何正确编写查询、如何组织表，以及应当按照什么规则来决定将哪些列添加至索引上。所以，很明显，我们需要有一本能够突破常规的书，真正开始思考为何现今仍有这么多人还会遇到如此多的问题。

为了满足这一需求，我们认为必须关注两个问题。第一个必须关注的对象是关系型系统中用于确定如何以最高效的方式查询所需数据的部分（我们称其为 SQL 优化器）。第二个必须关注的是索引及表是以何种方式被扫描的。我们试着把自己放在优化器的角度思考问题，也许当我们理解为什么可能存在问题时，我们就能够做出改变。幸运的是，我们需要知道的有关优化器的内容其实非常少，但非常重要。本书与其他同领域的书籍的一个很重要的区别在于，我们不会提供大量的用于指导 SQL 编写以及表和索引设计的规则和语法。这不是一本告诉你在各种场景下应当使用哪一个 SQL WHERE 语句的书，也不是一本告诉你应当使用什么语法的书。如果我们努力遵循一大堆复杂、模糊甚至可能不完整的指导原则，那么我们就是在走前人走过的老路。相反，如果我们能够理解 SQL 请求对关系型系统造成的潜在影响，并知道如何控制这一影响，那么我们就能够理解、控制、最小化甚至避免这些问题。

本书的第二个目的是展示如何使用这些知识从 CPU 和执行时间的角度

量化运行过程。只有这样，我们才能真正判断我们设计的表和索引是否合适，我们需要用真实的数字来展示优化器是如何思考的、扫描将耗费多少时间，以及需要进行哪些改动以提供满意的性能。不过，最重要的是，我们必须能够方便且快速地完成这一评估过程，这就要求我们必须将关注点放在少数几个真正重要的问题上，而不是将关注点放在那些不那么重要的细节上（许多人都被这些细节问题困扰过）。所以，关键就是要关注少数核心领域，并能够说出这需要花费多少时间或成本。

同样是由于我们专注于核心问题，所以我们还能提供另一个优势。对于那些可能使用多个关系型产品（即便来自相同的供应商）的人，由于我们在本书中所使用的是一种适用于所有关系型产品的通用方法，所以使用者就不需要阅读和掌握多套截然不同的规则和建议。所有“真正的”关系型系统的优化器都有一个相同的任务：它们都必须要扫描索引和表。它们都使用异常相似的方式来处理这些操作（虽然他们对其有各自不同的描述方式）。当然，它们之间的确存在着一些差异，但是我们可以毫不费力地处理这些不同。

也正是由于相同的原因，本书的读者对象包括：认为了解 SQL 性能方面的知识或如何有效设计索引的知识能给自己带来益处的人，直接负责索引设计的人，编写 SQL 语句用于查询或作为应用程序一部分的人，以及那些负责维护关系型数据和关系型环境的人。只要你觉得需要对自己所做的事情的性能影响负责，那么你都将不同程度地从本书中受益。

最后，用一句话概括本书目标读者所需具备的背景知识：我们假定读者已经具备了 SQL 这一关系型语言相关的知识。考虑阅读本书的人应该已经具备了对计算机系统的大体理解。除此以外，能帮到读者的最重要的品质也许就是对事物运行原理的好奇和兴趣了，还有想把事情做得更好的渴望。另一方面，在众多拥有几十年的关系型系统经验的人中，有两类人也会从本书受益：第一类是那些根据详细的规则手册良好地管理了系统很多年的人，他们想通过理解这些规则适用的原因来使自己的工作更轻松一些；第二类是那些已经使用了本书中所描述的技术很多年，但对于新硬件所带来的改善并不赞赏的人。

本书中的绝大部分观点及使用的技术都是原创的，因此很少有对外部出版物及其他作者成果的引用。在本书的创作过程中，我们非常感谢给予了我们如此多帮助和鼓励的朋友及同事们。感谢 Matti Ståhl 在全书撰写过程中所给予的详细指点及批判性但极其有用的建议。感谢 Lennart Henäng、Ari Hovi、Marja Kärmeniemi 和 Timo Raitalaakso 的帮助和校对，也感谢 Akira Shibamiya

在关系型性能公式上的原创工作。另外，还要感谢许许多多的学生和数据库顾问们，感谢他们提供的对于实际问题及其解决方案的深入见解。最后，特别感谢 Meta 和 Lyn，没有他们的鼓励与支持，本书不可能完成，Meta 还特别为本书设计了封面，与全书的主旨非常契合。

Tapio Lahdenmäki（斯姆勒尼科，斯洛文尼亚）
Michael Leach（什鲁斯伯里，英格兰）

目录

第 1 章　概述 1

关于 SQL 性能的另一本书 1
不合适的索引 3
误区和误解 4
　误区 1：索引层级不要超过 5 层 5
　误区 2：单表的索引数不要超过 6 个 6
　误区 3：不应该索引不稳定的列 6
　示例 7
　磁盘驱动器使用率 7
系统化的索引设计 8

第 2 章　表和索引结构 10

介绍 10
　索引页和表页 11
索引行 11
索引结构 12
表行 12
缓冲池和磁盘 I/O 12
　从 DBMS 缓冲池进行的读取 13
　从磁盘驱动器进行的随机 I/O 13
　从磁盘服务器缓存进行的读取 14
　从磁盘驱动器进行的顺序读取 15
　辅助式随机读 15
　辅助式顺序读 18
　同步 I/O 和异步 I/O 18
硬件特性 19
DBMS 特性 20
　页 20

表聚簇.....21
索引行.....21
表行.....22
索引组织表.....22
页邻接.....23
B 树索引的替代品.....24
聚簇的许多含义.....25

第 3 章 SQL 处理过程.....27

简介.....27
谓词.....27
评注.....28
优化器及访问路径.....28
索引片及匹配列.....29
索引过滤及过滤列.....29
访问路径术语.....31
监控优化器.....32
帮助优化器（统计信息）.....32
帮助优化器（FETCH 调用的次数）.....32
何时确定访问路径.....33
过滤因子.....34
组合谓词的过滤因子.....35
过滤因子对索引设计的影响.....37
物化结果集.....39
游标回顾.....39
方式 1：一次 FETCH 调用物化一条记录.....40
方式 2：提前物化.....41
数据库设计人员必须牢记.....41
练习.....41

第 4 章 为 SELETE 语句创建理想的索引.....43

简介.....43
磁盘及 CPU 时间的基础假设.....44
不合适的索引.....44

三星索引——查询语句的理想索引 45
星级是如何给定的 46
范围谓词和三星索引 48
为查询语句设计最佳索引的算法 49
候选 A 50
候选 B 50
现今排序速度很快——为什么我们还需要候选 B 51
需要为所有查询语句都设计理想索引吗 52
完全多余的索引 52
近乎多余的索引 53
可能多余的索引 53
新增一个索引的代价 54
响应时间 54
磁盘负载 55
磁盘空间 56
一些建议 57
练习 58

第 5 章 前瞻性的索引设计 59

发现不合适的索引 59
基本问题法（BQ） 59
注意 60
快速上限估算法（QUBE） 61
服务时间 62
排队时间 62
基本概念：访问 63
计算访问次数 65
FETCH 处理 66
主要访问路径的 QUBE 示例 67
使用满足需求的成本最低的索引还是所能达到的最优索引：示例 1 72
该事务的基本问题 73
对该事务上限的快速估算 73
使用满足需求的成本最低的索引还是所能达到的最优索引 74
该事务的最佳索引 75

半宽索引（最大化索引过滤）.....75
宽索引（只需访问索引）.....76
使用满足需求的成本最低的索引还是所能达到的最优索引：示例 2....77
范围事务的 BQ 及 QUBE.....78
该事务的最佳索引.....79
半宽索引（最大化索引过滤）.....80
宽索引（只需访问索引）.....81
何时使用 QUBE.....82

第 6 章 影响索引设计过程的因素.....83

I/O 时间估算的验证.....83
多个窄索引片.....84
简单就是美（和安全）.....86
困难谓词.....87
LIKE 谓词.....87
OR 操作符和布尔谓词.....88
IN 谓词.....89
过滤因子隐患.....90
过滤因子隐患的例子.....92
最佳索引.....95
半宽索引（最大化索引过滤）.....96
宽索引（只需访问索引）.....97
总结.....97
练习.....99

第 7 章 被动式索引设计.....100

简介.....100
EXPLAIN 描述了所选择的访问路径.....101
全表扫描或全索引扫描.....101
对结果集排序.....101
成本估算.....102
数据库管理系统特定的 EXPLAIN 选项及限制.....102
监视揭示现实.....103
性能监视器的演进.....104

LRT 级别的异常监视 106
程序粒度的均值是不够的 106
异常报告举例：每个尖刺一行 106
问题制造者和受害者 108
有优化空间的问题制造者和无优化空间的问题制造者 108
有优化空间的问题制造者 109
调优的潜在空间 111
无优化空间的问题制造者 114
受害者 115
查找慢的 SQL 调用 117
调用级别的异常监视 118
Oracle 举例 121
SQL Server 举例 123
结论 125
数据库管理系统特定的监视问题 126
尖刺报告 127
练习 127

第 8 章 为表连接设计索引 129

简介 129
两个简单的表连接 131
例 8.1：CUST 表作为外层表 131
例 8.2：INVOICE 表作为外层表 132
表访问顺序对索引设计的影响 133
案例研究 133
现有索引 136
理想索引 142
理想索引，每事务物化一屏结果集 146
理想索引，每事务物化一屏结果集且遇到 FF 缺陷 149
基本连接的问题（BJQ） 151
结论：嵌套循环连接 153
预测表的访问顺序 153
合并扫描连接和哈希连接 155
合并扫描连接 155

例 8.3：合并扫描连接 155
哈希连接 157
程序 C：由优化器选择 MS/HJ（在现有索引条件下） 158
理想索引 159
嵌套循环连接 VS. MS/HJ 及理想索引 161
嵌套循环连接 VS. MS/HJ 161
嵌套循环连接 VS.理想索引 162
连接两张以上的表 163
为什么连接的性能表现较差 166
模糊的索引设计 166
优化器可能选择错误的表访问路径 166
乐观的表设计 166
为子查询设计索引 167
为 UNION 语句设计索引 167
对于表设计的思考 167
冗余数据 167
无意识的表设计 171
练习 173

第 9 章 星型连接 175

介绍 175
维度表的索引设计 177
表访问顺序的影响 178
事实表的索引 179
汇总表 182

第 10 章 多索引访问 184

简介 184
索引与 184
与查询表一同使用索引与 186
多索引访问和事实数据表 187
用位图索引进行多索引访问 187
索引或 188
索引连接 189
练习 190

第 11 章 索引和索引重组 191

B 树索引的物理结构 191
DBMS 如何查找索引行 192
插入一行时会发生什么 193
叶子页的分裂严重吗 194
什么时候应该对索引进行重组 196
　　插入模式 196
索引列的稳定性 205
长索引行 207
举例：对顺序敏感的批处理任务 208
　　表乱序（存在聚簇索引） 211
　　表乱序（没有以 CNO 开头的聚簇索引） 212
存储在叶子页中的表行 212
　　SQL Server 212
　　Oracle 213
索引重组的代价 214
分裂的监控 215
总结 216

第 12 章 数据库管理系统相关的索引限制 219

简介 219
索引列的数量 219
索引列的总长度 220
变长列 220
单表索引数量上限 220
索引大小上限 220
索引锁定 221
索引行压缩 221
数据库管理系统索引创建举例 222

第 13 章 数据库索引选项 224

简介 224
索引行压缩 224

索引键以外的其他索引列 225
唯一约束 227
从不同的方向扫描数据库索引 227
索引键截断 228
基于函数的索引 228
索引跳跃式扫描 229
块索引 230
数据分区的二级索引 230
练习 231

第 14 章 优化器不是完美的 232

简介 232
优化器并不总能看见最佳方案 234
匹配及过滤问题 234
非 BT 谓词 234
无法避免的排序 237
不必要的表访问 238
优化器的成本估算可能错得离谱 239
使用绑定变量的范围谓词 239
偏斜分布 241
相关列 242
部分索引键的警示故事 243
成本估算公式 246
估算 I/O 时间 247
估算 CPU 时间 248
协助优化器处理估算相关的问题 249
优化器的问题是否会影响索引设计 252
练习 253

第 15 章 其他评估事项 254

QUBE 公式背后的假设条件 254
内存中的非叶子索引页 255
例子 255
磁盘服务器读缓存的影响 256

缓冲子池 258
长记录 259
慢速顺序读 259
实际的响应时间可能比 QUBE 评估值短得多 259
叶子页和表页缓存在缓冲池中 260
识别低成本的随机访问 262
辅助式随机读取 262
辅助式顺序读 265
评估 CPU 时间（CQUBE） 265
单次顺序访问的 CPU 时间 265
单次随机访问的 CPU 时间 267
单次 FETCH 调用的 CPU 时间 269
每排序一行的平均 CPU 时间 270
CPU 评估举例 270
宽索引还是理想索引 270
嵌套循环（及反范式化）还是 MS/HJ 271
合并扫描与哈希连接的比较 274
跳跃式顺序扫描 275
CPU 时间仍然不可忽视 276

第 16 章 组织索引设计过程 277

简介 277
计算机辅助式索引设计 278
设计出色索引的 9 个步骤 280

参考文献 282

术语表 283

索引 291

第 1 章

概述

- 了解 SQL 优化器是如何决定应该执行何种表及索引扫描的，以尽可能高效地处理 SQL 语句
- 能够量化这些扫描中所涉及的操作，从而实现令人满意的索引设计
- 本书目标读者的类型和背景
- 对于索引不合适的主要原因的初步想法
- 系统化的索引设计

关于 SQL 性能的另一本书

关系型数据库至今已存在了三十多年，性能问题也随之存在了三十多年——本书是另一本关于这个话题的书。本书侧重于性能领域的索引设计方面，还有其他一些书在更大或更小的程度上对这一领域进行过思考。虽然这类书已经存在了三十多年，但性能问题却仍然继续出现。所以，也许这里需要一本书，能够超越常规的范畴并思考为什么这么多人仍然遇到了那么多的性能问题。

当然，关系型数据库系统的世界是非常复杂的——如果我们思考一下我们需要做哪些事情才能满足 SQL 语句的查询需求，就能意识到这种复杂是必然的。但具有讽刺意味的是，书写 SQL 是如此简单，表、行与列的概念也非常容易理解。我们可以从世界各地的巨大数据来源中查找大量的信息——甚至不需要知道它在哪里或者它是如何被查到的。我们既不需要担心这将花费多少时间，也不需要担心这将花费多少钱。这一切看起来就像是魔术。也许这正是问题的一部分——它太简单了，但当然，它也应该很简单。

如今我们仍旧意识到将会有问题出现，并且是巨大的问题。预期只需花费几分之一秒的“简单”查询似乎更倾向于花费几分钟或者更长的时间。于是，我们就有了所有这些书，它们告诉我们如何正确地编写查询语句，如何组织表，以及遵循何种规则把正确的列放进索引——这通常是有效的。尽管

这些书大都非常好，而且这些书的作者确实很清楚自己在讲些什么，但我们似乎仍然在不断地遇到各种性能问题。

在本书中，我们尤其感兴趣的是关系型系统（我们称其为 SQL 优化器），正是它决定了如何以最高效的方式找到我们所需的所有信息。在理想情况下，我们不需要知道它的存在，并且事实上大多数人都乐意如此！在做完决策之后，优化器会接着指挥表和索引进行扫描来找到我们所需的数据。为了理解优化器在这一过程中是如何思考的，我们同样需要理解这些扫描涉及了什么。

所以我们在本书中想做的第一件事就是试着把我们自己放在优化器的位置，思考它如何决定应该执行什么样的表和索引扫描，以尽可能高效地执行 SQL 语句。如果我们理解了它为什么可能有问题，也许我们就能用不同的方式去处理性能问题。我们要试图理解它正试图做什么，而不是简单地跟从那堆复杂得难以置信的规则，即使我们能够理解这些规则是否适用。

读到这儿，读者最大的担心可能就是这看起来太复杂了，甚至是不可能的。但实际上，我们真正需要理解的东西出乎意料得少，不过，极其重要。

同样，或许这本书与同领域中其他书籍的首要区别就在于，我们不会提供一个庞大的针对 SQL 编写、表或索引设计的规则和语法列表。这不是一本精确描述所有情况（比如应该使用哪一个 SQL WHERE 子句，或应使用什么语法）的参考书。如果我们试图遵循一长串复杂的、模糊的甚至可能不完整的操作指南来进行设计，那么我们将走上别人已经走过的道路。相反地，如果我们理解了我们的 SQL 指令会对关系系统产生什么样的影响，以及如何改变这一影响，那么我们将能理解、避免、最小化并控制我们所遇到的问题。

本书的第二个目的是展示如何使用这些知识来量化系统运行所涉及的工作，只有这样我们才能真正判断出我们的索引设计是否是成功的。我们需要能够使用实际的数字来描述优化器将思考什么，扫描将会花费多长时间，以及为了提供令人满意的性能我们还需要做哪些改动。但最重要的是，我们必须能够快速轻松地完成这些，这就意味着专注于核心问题而非次要细节是极为重要的，许多人都是被这些次要的细节所困住的。这就是关键——专注于少数核心问题，并能说出它需要花费多长时间或多少成本。

专注于关键问题，这使得我们还能够提供一个更大的优势。对于那些可能使用多个关系型产品（即便来自相同的供应商）的人，并不需要阅读和掌握多套截然不同的规则和建议，因为我们在本书中所使用的是一个适用于所有关系型产品的通用方法。所有“真正的”关系型系统都有一个有着相同任务的优化器，它们都必须要扫描索引和表。而且它们对这些任务的处理方式都异常相似（虽然描述方式各不相同）。当然，它们之间的确存在着一些差

异，但是我们可以毫不费力地处理这些不同。

本书的目标读者，从字面上说，是那些认为了解 SQL 性能方面的知识，或了解如何有效地设计索引能给自己带来益处的人。对于直接负责索引设计的人、编写 SQL 语句用于查询或作为应用程序一部分的人，以及负责维护关系型数据和关系型环境的人，如果他们觉得需要对自己所做的事情的性能影响负责，那么都将不同程度地从本书中受益。

最后，我们来概括一下本书目标读者所需具备的背景知识。我们假定读者已经具备了 SQL 这一关系型语言相关的知识，幸运的是，当今这种知识可以很容易地从大量资料中获得；另外读者需要大体理解计算机系统；除此以外，能帮到读者的最重要的品质也许就是对事物运作原理的好奇和兴趣了，还有想把事情做得更好的渴望。另一方面，在众多拥有几十年的关系系统经验的人中，有两类人也会从本书受益：第一类，是那些根据详细的规则手册良好地管理系统很多年的人，他们想通过理解这些规则适用的原因来使自己的工作更轻松；第二类，是对本书中所描述的技术已经使用了很多年的人。他们之所以会对本书产生感兴趣，可能是因为这些年来硬件的发展超越了所有人的认知，过去的问题在今天已不再是问题，但即便如此，性能问题却仍然不断地出现！

我们将从反思这一问题来开始我们的讨论：为什么索引依旧是如此之多问题的来源？

不合适的索引

重要提示

下面的讨论将使用一些用于关系型数据库系统和磁盘子系统的概念和术语。如果读者在这些领域只有很少或没有背景知识，可能只能粗浅地阅读这一章，略过一些用于证明论述和结论的技术性细节。相关技术领域的更多细节将会在第 2 章至第 4 章中详述。

多年来，不合适的索引是性能低下的最常见原因。最普遍的问题似乎是 4
没有索引足够多的列来支持 WHERE 子句中的所有谓词。普遍的情况包括表上没有足够多的索引；一些 SELECT 语句可能没有有效的索引；有时索引上包含了正确的列，但列的顺序却不对。

在关系型数据库中改进索引是相对容易的，因为不需要进行任何程序上的修改。然而，对生产系统的改动总是会带来一些风险。此外，当一个索引正在被创建的时候，更新程序可能会经历长时间的等待，例如无法更新为了

创建索引而正在被扫描的表。由于这些原因，再加上为了让新应用从投产第一天开始就达到可接受的性能，所以在投产之前索引就应处于相当良好的状态。索引应当在系统投产后不久就被固定下来，而不再需要大量的实验。

数据库索引已经存在了数十年，那么为什么索引平均质量却仍然那么差？其中的一个原因也许是由于很多人认为，在当前巨大的处理和存储能力下，已经不再需要为这些看起来如此简单的 SQL 担心性能问题了。另一个原因可能是已经没有几个人还在考虑这个问题了。而对于那些考虑过这个问题的人，他们往往在众多关系数据库教材和教育课程入门之处就犯下了错误。浏览各种关系数据库管理系统（DBMS）的书籍将很可能得出以下判断：

- 索引设计的话题很简短，也许只有寥寥数页。
- 索引的副作用被强调了，索引会消耗磁盘空间，而且它们会使插入、更新和删除操作变慢。
- 索引设计的准则是模糊的，且有时是有问题的。一些作者推荐对所有的限制性列都创建索引；另一些人则主张索引设计是一门艺术，只能通过练习和试错才能掌握。
- 很少或没有书籍尝试给索引设计的整个过程提供一种简单而有效的方法。

许多关于索引成本的警告 20 世纪 80 年代遗留下来的，在当时，存储、磁盘和半导体的价格比现在昂贵得多。

误区和误解

即便是 21 世纪之后出版的书，比如 2002 年出版的一本书（1），竟也认为通常只有 B 树索引的根页会留在内存中。在三十年前，这是一个合理的假设，因为当时系统的内存通常很小，可能小到不及 1MB 字节，以至于数据库缓冲池只能包含几百个页。而今天，数据库缓冲池能够包含成千
5 上万个页，大小变为千兆字节（GB）或更多，而磁盘服务器的读缓存则通常更大——如 64 GB。尽管随着磁盘存储越来越便宜，数据库的规模也在变大，但是在现今我们完全可以假设 B 树索引的所有非叶子页通常都会留在内存或读缓存中。通常只有叶子页需要从磁盘驱动器读取，这显然使得索引维护的速度比以前更快了。

只有根页留在内存中的假设导致了很多过时的和危险的建议，以下只是其中的几个例子。

误区 1：索引层级不要超过 5 层

这个建议常常在关系型文献中出现，通常基于的假设就是只有根页是留在内存中的。在当前的处理器条件下，即使所有非叶子页都在数据库缓冲池中，每个索引层级也将向索引扫描过程贡献约 50 微秒（μs）的中央处理单元（CPU）时间。如果非叶子页不在数据库缓冲池中而在磁盘服务器的读缓存中，那么读取索引页可能会花费 1 毫秒（ms）。应该将这些值与一次磁盘随机读取所花费的时间（大约是 10 ms）相比较。为了了解这实际上意味着什么，我们将进行一个简单的说明。

图 1.1 所示的是一个拥有 1 亿行数据的表的索引。该索引有 1 亿索引行且行平均长度为 100 字节。考虑到离散的空闲空间，大约每个叶子页会包含 35 个索引行。如果 DBMS 在非叶子页中不截断索引键，那么这些非叶子页中索引条目的数量也是 35。



图 1.1　有 6 个层级的大索引

图 1.1 展示了这些页可能的分布以及它们的大小，由此我们可以推导出如下结论：

- 该索引上总共有大约 3 000 000 个 4KB 大小的页，共需要 12GB 的磁盘空间。 6
- 叶子页的总大小为 2 900 000 × 4KB，几乎占 12GB 空间。比较合理的假设是这些页通常会从磁盘读取（10 ms）。
- 下一个层级的大小为 83 000 × 4KB，占 332 兆字节（MB）。如果该索引被频繁使用，即使这些索引页不在数据库缓冲池中（比如用于存放索引页的 4GB 内存），那么也可能会留在磁盘服务器的读

缓存中（可能是 64 GB）。

- 再上面的层级大小约为 2500 × 4 KB ≈ 10 MB，几乎肯定会留在数据库缓冲池中。

访问这 6 层索引的 100 000 000 个索引行中的任意一个索引行，将花费 10 ms ~ 20 ms。即使许多索引行被杂乱无章地添加到索引中，这一数字也是对的，关于这方面的更多信息将在第 11 章中讲解。因此，对索引的层数做强制限制是没有什么意义的。

误区 2：单表的索引数不要超过 6 个

Mark Gurry 早期创作的 *Oracle SQL Tuning Pocket Reference*（2）不赞同这一观点，可以说是针对这个观点的一个令人欣慰的例外，这从书中关于索引的正面态度可以看出。正如书名所指，这本书专注于协助 Oracle 9i 的优化器，但它在书中第 63 页也对设置单表索引数上限的标准进行了批判：

> 我访问过一些网站，这些网站都制定了单表不得超过 6 个索引的标准。这往往会导致绝大部分 SQL 语句都运行良好，但少数语句运行糟糕，且不能添加新索引，因为表上已经有 6 个索引了。
>
> ……
>
> 我建议不要给表的索引数目设置上限。
>
> ……
>
> 保证所有的 SQL 语句都能够流畅运行是设计的底线。我们总能找到一种方法来达到这一点。如果为了达到这一点需要在表上创建 10 个索引，那么你就应该在表上建立 10 个索引。

误区 3：不应该索引不稳定的列

索引行是按索引键的顺序存储的，所以当索引键中有一列被更新时，DBMS 可能不得不把相应的行从旧的索引位置移到新的位置来保持这一顺序。这个新的位置可能与旧的位置位于相同的叶子页上，在这种情况下，只有一个页会受到影响。然而，如果被修改的键是第一列或唯一的列，那么新的索引行可能必须被迁移到一个不同的叶子页上，即 DBMS 必须更新两个叶子页。三十年前，如果这个索引为一个 4 层索引，这也许需要 6 次磁盘随
7 机读取：3 次常规读取，即 2 次非叶子页读取和 1 次叶子页读取，加上新的位置所涉及的 3 次随机读取。当一次随机读取耗时 30 ms 时，迁移一个索引行可能会给该更新操作额外增加 6 × 30 ms = 180 ms 的响应时间。因此，不稳定的列很少被索引就不足为奇了。

现在，当四层索引中三个层级的非叶子页保留在内存中时，一次磁盘随

机读取需要 10 ms，响应的时间变成了 2 × 10 ms = 20 ms。此外，许多索引为多列索引，也称作复合或组合索引，它通常包含多列，以使得索引键值唯一。当不稳定的列为复合索引的尾列，更新这个不稳定的列绝不会导致其迁移到新的叶子页。因此，在当前的磁盘条件下，更新一个不稳定的列只会对该更新操作增加 10 ms 的响应时间。

示例

几年前，一个经过很好调优的 DB2 数据库出现了本地平均响应时间为 0.2 秒的情况，DBA 开始对其进行事务级别的异常诊断。很快，他们注意到一个简单的浏览事务经常耗时超过 30 秒，观察到的最长的一次本地响应时间为几分钟。他们很快追踪到问题的原因是一张 200 万行的数据表没有合适的索引。诊断出的两个问题是：

- 一个每秒更新两次的不稳定列 STATUS 并不在索引中，虽然它显然是查询的必备条件。用此 STATUS 列进行匹配的谓词与 WHERE 条件中的其他 5 个谓词一同进行 AND 运算。
- ORDER BY 操作需要对结果集进行一次排序。

这两个索引设计决策是基于广泛运用的建议而有意识地制定的。STATUS 列比这个数据库中的其他大多数列都更不稳定。这就是 DBA 不敢把它添加到索引中的原因。他们还担心一个额外的能消除排序的索引，会因为表的写入速度相当高而引起 INSERT 性能问题。他们尤其担心磁盘驱动器的负载。

在意识到这两个问题的严重程度后，DBA 对新建索引将带来的额外负载进行了粗略的估算，随后他们决定创建一个额外的包含这 5 个列的索引，并在索引末尾加上 STATUS 列。这个新的索引解决了这两个问题。DBA 观察到的最长响应时间从几分钟降至不到一秒。更新和插入事务并没有因此而受牵连，索引所在的磁盘也并未超载。

磁盘驱动器使用率

磁盘驱动器的负载及 INSERT、UPDATE 和 DELETE 的性能需求仍然决定了表上索引数目的上限。然而，这个上限比三十年前要高得多。一个合理的添加新索引的需求并不应该被直接拒绝。在当前的磁盘条件下，只有在更新频率多于 10 次/秒的情况下，不稳定列才可能成为问题，但是这样的列并不常见。

系统化的索引设计

索引设计方法的首次尝试源于20世纪60年代。当时，教材推荐人们使用一种矩阵模型的方法，用于预测每个字段（列）读取和更新的频率，以及包含这些字段的记录（行）的插入和删除频率。这一模型能帮助人们得出有待被索引的列的列表。通常这些索引都被假定为只包含一个列，以减少高峰期磁盘输入/输出（I/O）的数量。令人惊讶的是，在最近的一些书中，这种方法仍然被提及，尽管有些较现实的作者承认这一矩阵模型应当只应用于最普通的事务。

这个*列活动矩阵模型方法*可以用于解释为什么在最近的教科书和数据库课程中还能找到面向列的调整思路，比如，*考虑索引具有这些属性的列、避免索引具有那些属性的列*。

在20世纪80年代，*面向列的索引设计方式*开始落后于*面向响应时间的索引设计方式*。与时俱进的DBA们开始意识到，创建索引的目的应该是在硬件容量限制的前提下保证所有的数据库调用运行得足够快。IBM S/38（后来的AS/400，再后来的iSeries）这种伪关系型DBMS就是这种思想的先驱。它会自动为每一个数据库调用创建一个好的索引。对于简单的应用程序，这一方式运作良好。今天，有许多产品会对每个SQL调用提供索引建议，但除了主键索引及外键索引外，它们不自动创建索引。

随着应用变得越来越复杂，数据库变得越来越大，索引设计的重要性和复杂性也越来越明显了。一些项目开始着手开发自动化设计工具。基本的思想就是采集生产环境的工作负载样本，然后在该工作负载下为SELECT语句生成一组候选索引。随后，用一些简单的评估公式或一个基于成本的优化器来决定哪些索引是最有价值的。这类产品已经推向市场很久了，但传播速度比预期慢得多。我们将在第16章中讨论其可能的原因。

系统化的索引设计包含如图1.2所示的两个过程。首先，必须找到在当前索引条件下运行得非常慢或将会运行得非常慢的查询，至少找到在最差输入的条件下（比如，“最大的客户”或“最老的日期”）运行非常慢的查询。然后，必须设计索引以使缓慢的查询变得足够快，并且不导致其他SQL调用明显变慢。这两个步骤缺一不可。

9 在*设计阶段*，检测不合适索引的首次尝试是基于无比复杂的预测公式进行的，有时也会基于成本优化器所使用的简化版公式来进行。用程序及图形用户界面取代计算器并没有大幅降低工作量。后来，一些极其简单的公式，比如20世纪80年代后期于IBM芬兰开发的QUBE公式和一个估算随机I/O数量的简易算法，被发现在真实项目中是有用的。Ari Hovi提出的基础问题

是索引设计过程的下一个，也可能是最重要的步骤。这两个观点将在第 5 章中讨论，且会在全书中被广泛应用。

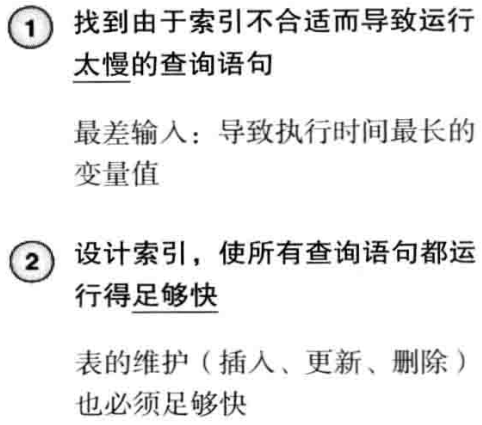


图 1.2　系统化的索引设计

20 世纪 90 年代，系统上线之后的索引改进方法显著地发展了起来。先进的监控软件形成了这一技术的必要基础，但是，一个将大量测量数据有效利用起来的方法也是必不可少的。

很长一段时间以来，系统化索引设计的第二步都未被认可。教材和课程资料中的 SELECT 都异常简单，以至于很容易就能设计出最佳索引。然而，实际应用程序的经验告诉我们，即便是看起来无害的 SELECT，尤其是表连接，通常也会有大量合理的索引设计方式。估算每一种方式需要花费太多精力，而测量所花费的时间则更多。另一方面，即便是经验丰富的数据库设计者，当其凭借直觉来设计索引时也会犯很多错误。

这就是我们需要一个算法来为给定的 SELECT 设计最佳索引的原因。第 4 章中所讨论的三星索引及其相关候选索引的概念，被证明是有效的方法。

这些简单的手工索引设计算法已经在许多场景中被成功地运用了。运用这一方法后，SELECT 调用的运行时间降低两个数量级的情况并不罕见。例如，从远大于 1 分钟降低至远小于 1 秒，使用第 4 章、第 5 章、第 7 章、第 8 章中所建议的方法，我们能够几乎不费吹灰之力（也许仅需 5 分钟或 10 分钟）地做到这一点。

第 2 章

表和索引结构

- 表和索引的物理结构
- 索引及表页、索引及表行、缓冲池，以及磁盘缓存的结构和使用
- 随机和顺序磁盘 I/O 的特性
- 一些辅助式随机读和顺序读：跳跃式顺序读、列表预读及数据块预读
- 同步和异步 I/O 的意义
- 各种数据库管理系统之间的异同
- 页和表聚簇、索引行、索引组织表，以及页邻接
- 关于术语聚簇的易混淆但非常重要的问题
- B 树索引的替代品
- 位图索引和散列

介绍

在我们讨论索引设计之前，我们必须先理解索引和表是如何组织和使用的。当然，这很大程度上取决于具体的关系型 DBMS。然而，它们都依赖于大致相似的结构和原则，尽管在处理中它们使用的术语差异很大。

在这一节中，我们将会研究关系对象的基本结构，我们将会讨论在使用它时与性能相关的一些问题，例如缓冲池的作用、磁盘和磁盘服务器，以及怎样使用它们以使 SQL 处理过程能够获取到这些数据。

一旦我们对于这些基本概念熟悉了之后，我们就可以进入下一节，去讨论怎样使用这些关系型的对象来满足 SQL 调用。

本章仅仅是一个介绍，更多的细节将会在本书合适的地方给出。在本书的最后提供了一个本书中使用过的所有术语的词汇表。

索引页和表页

表和索引行都被存储在页中。页的大小一般为 4KB，这是一个可以满足大多数需求的大小，不过也可以使用其他大小。幸好在索引设计中这并不是一个很重要的考虑点，页的大小仅仅决定了一个页可以存储多少个索引行、表行，以及一共需要多少个页来存储表或者索引。当表和索引被加载或重组时，每个页都会预留出一定比例的空闲空间，以满足向其添加新的表行或索引行的需求。我们稍后会讨论这一点。

缓冲池和 I/O 活动（接下来讨论）都是基于页的，例如一次将一个完整的页进读取到缓冲池。这意味着一次 I/O 会读入多条记录到缓冲池，而不仅仅是一条。我们还将看到，一次 I/O 可以读入多个页到缓冲池中。

索引行

索引行在评估访问路径的时候是一个非常有用的概念。对于一个唯一索引，例如表 CUST 上的主键索引 CNO，一个索引行等同于叶子页中的一个索引条目（参见图 2.1）。字段的值从表中复制到索引上，并加上一个指向表中记录的指针。通常，表页的编号是这个指针的组成部分，我们需要牢记这一点，以便为后续的讨论做准备。对于一个非唯一索引，例如表 CUST 上的索引 CITY，一个特定的索引值所对应的索引行应该被想象成独立的索引条目，每一个都含有相同的 CITY 值，但是却有不同的指针。大多数情况下，非唯一索引的实际存储方式是一个 CITY 值后带着多个指针。将这些索引记录想象成独立的索引条目的好处将在后文中讲述。

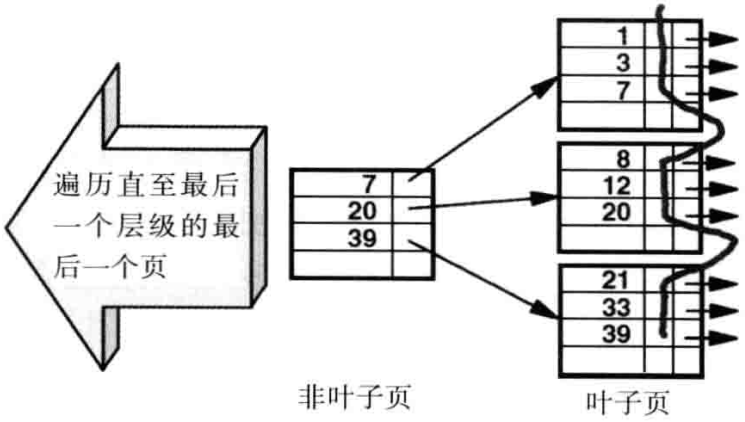


图 2.1　非常小的索引

13

索引结构

非叶子页通常包含着一个（可能被截断的）键值，以及一个指向下一层级页的指针，该键值是下一层级页中的最大键值，如图 2.1 中所示。多个索引层级按照这一方式逐层建立，直到只剩下一个页，我们把这个页叫作根页，它位于索引结构的最上层。这种组织方式的索引被称为 B 树索引（一种平衡树索引），因为通过这种索引来查找任何一条索引记录都需要访问相同数量的非叶子页。

表行

图 2.1 中的每一个索引行都指向表中相对应的一行记录，指针通常标识了记录所存放的页及它在页中的位置。表中的每一行除了存储行的字段之外，还包含了一些控制信息用于定义行并帮助 DBMS 处理插入或删除操作。

当加载表或者向表中插入记录的时候，表中记录的顺序可以被定义成和它的某一个索引记录相同的顺序。在这种情况下，当索引行被按顺序处理时，对应的表行也将依照相同的顺序被逐个处理。索引和表都按相同的顺序被访问，我们很快就会看到，这是一个效率很高的处理过程。

显然，表中记录的顺序只能按照表上某一个索引的顺序来组织。如果通过表上其他的索引来访问这张表，那么表中相应的记录将不会按照与索引条目相同的顺序存储。例如，第一条索引记录有可能指向页 17，下一条索引记录有可能指向页 2，再下一条可能指向页 85，等等。如此一来，虽然索引的处理仍然是顺序且高效的，但是表的处理却是随机且低效的。

缓冲池和磁盘 I/O

关系型数据库管理系统最重要的一个目标是确保表或者索引中的数据是随时可用的。为了尽可能地实现这个目标，我们使用内存中的缓冲池来最小化磁盘活动。每个 DBMS 都会根据对象类型（表或索引）及页的大小拥有多个缓冲池。每一个缓冲池都足够大，大到可以存放许多页，可能是成千上万的页。缓冲池管理器将尽力确保经常使用的数据被保存在池中，以避免一些不必要的磁盘读。这一策略的有效性对于 SQL 语句的性能表现至关重要，因此对于本书的目标也同样重要。我们会在必要时回来讨论这个话题。现在我

14 们需要简单地意识到，索引或表页在或不在缓冲池中，访问的成本是不同的。

从 DBMS 缓冲池进行的读取

如果一个索引或者表页在缓冲池中被找到，那么唯一的成本就是去处理这些索引或者表的记录。成本的具体值高度依赖于这些记录是否是 DBMS 所需要的，若不是则只需要非常少的处理，若是则需要相对较多的处理，我们会在适当的地方讲述这一点。

从磁盘驱动器进行的随机 I/O

图 2.2 展示了等待一个页从磁盘驱动器读取至缓冲池所需的巨大成本。

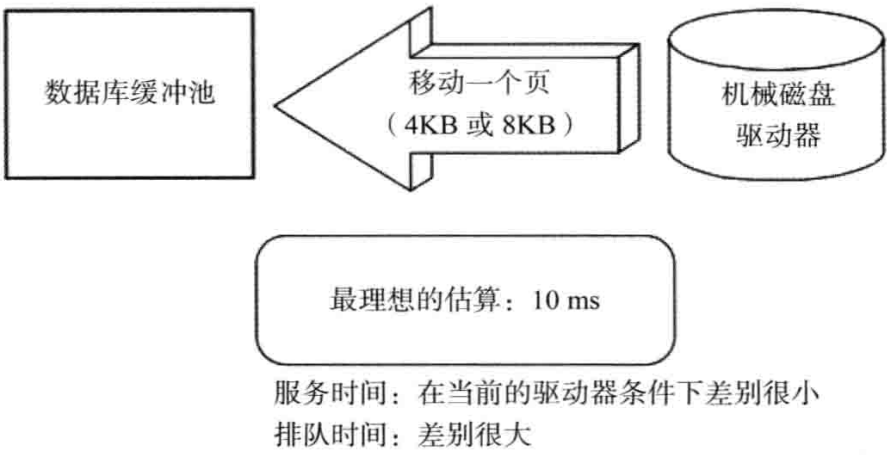
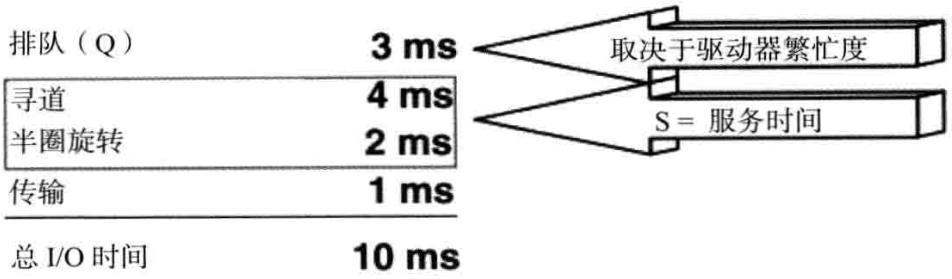


图 2.2　从磁盘驱动器进行的随机 I/O-1

我们必须记住一个页包含了多条记录，我们可能需要该页上所有的行，或者其中一部分行，又或者只需要其中某一行——但所花的成本都是相同的，约 10 ms。如果磁盘驱动器被重度使用，那么这一速度将大幅降低，因为需要等待磁盘空闲下来。在计算领域中，10 ms 是一个很长的时间，这也是我们贯穿全书来讨论这一过程的原因。

其实我们并不一定要知道这 10 ms 是怎么得出的，不过如果有读者希望知道这一数字是如何得出的，可以看图 2.3 对其组成部分的分解。从图中可以看到，我们假设在这 10 ms 中磁盘的实际繁忙时间为 6 ms。1 ms 左右的传输时间是将页从磁盘服务器缓冲区移动至数据库缓冲池所花费的。其余 3 ms 是对于可能发生的排队时间的估计值，这是基于每秒 50 次读取的磁盘活动情况得出的。这组数字对于直接安装在主机上的磁盘也是适用的，当然，所有这些数字在实际情况中都可能有变化，但是我们只需记住 10 ms 这一粗略但合理的数字即可。

15



一次随机读使得磁盘驱动器运行 6 ms

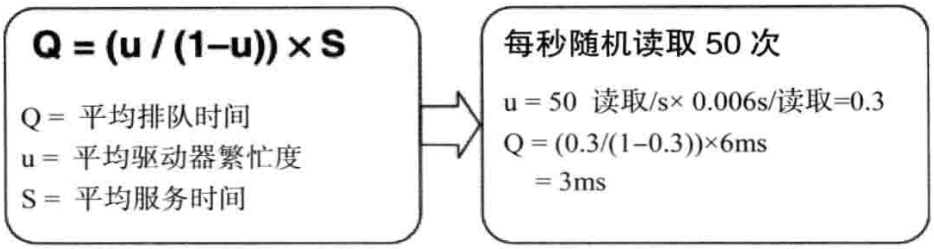


图 2.3　从磁盘驱动器进行的随机 I/O-2

从磁盘服务器缓存进行的读取

幸运的是，现今的磁盘服务器都提供自己的内存（或缓冲区）以降低响应时间上的巨大成本。图 2.4 展示了从磁盘服务器的缓冲区读取一个表或索引页（等同于读取多个表或索引行）的过程。就像从数据库缓冲池读取一样，磁盘服务器试图将频繁使用的数据保留在内存（缓冲区）中，以降低高昂的磁盘读成本。若 DBMS 所需的页不在数据库缓冲池中，继而会向磁盘服务器发起读请求，磁盘服务器会判断该页是否在服务器缓冲区中，只有当它发现页不在缓冲区中时才从磁盘驱动器上读取该页。如果该页在磁盘服务器的读缓冲区中，那么所花费的时间将从 10 ms 大幅降低至 1 ms。

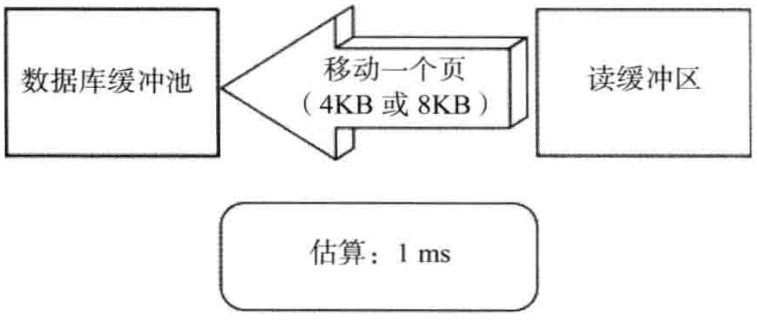


图 2.4　从磁盘服务器缓冲区读取

所以总而言之，当一个索引或表页被请求时，它的理想位置是在数据库缓冲池中。如果它不在那儿，那么下一个最佳的位置是在磁盘服务器的读缓冲区中。如果它也不在那儿，那么就必须从磁盘进行一次很慢的读取，这一

过程中可能要花费很长的时间等待磁盘设备空闲下来。

从磁盘驱动器进行的顺序读取

到目前为止，我们只考虑了将一个索引或表页读取到缓冲池中的情况。但在许多实际场景下，我们需要将多个页读取到缓冲池中，并按顺序处理它们。图 2.5 展示了 4 种这样的场景。DBMS 会意识到多个索引或表页需要被顺序地读取，且能识别出那些不在缓冲池中的页。随后，它将发出多页 I/O 请求，每次请求的页的数量由 DBMS 决定。只有那些不在缓冲池中的页会被从磁盘服务器上读取，因为那些已经在缓冲池中的页中可能包含了尚未被写入磁盘的更新数据。

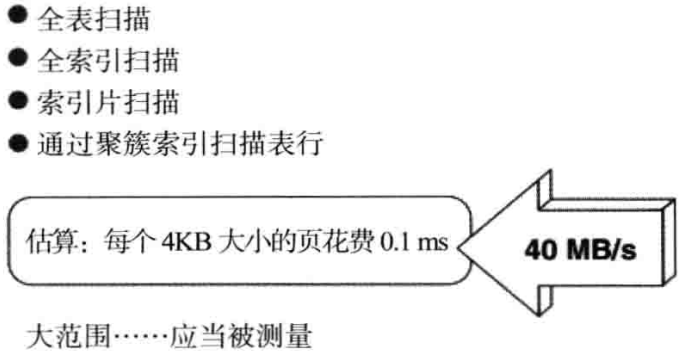


图 2.5　从磁盘驱动器顺序读取

顺序地读取页有两个非常重要的优势：

- 同时读取多个页意味着平均读取每个页的时间将会减少。在当前的磁盘服务器条件下，对于 4KB 大小的页而言，这一值可能会低至 0.1 ms（40 MB/s）。
- 由于 DBMS 事先知道需要读取哪些页，所以可以在页被真正请求之前就提前将其读取进来，我们称其为预读。

图 2.5 中使用的术语*索引片*和*聚簇索引*将在后续讨论。用于指代上述所描述的顺序读取的术语包括：*顺序预读*、*多块 I/O* 和*多重顺序前读*。

辅助式随机读

我们已经看到了随机读取的成本有多高，也看到了缓冲池和磁盘缓冲区是如何帮助将成本降至最低的。还有其他一些场景也能降低这一成本，有时这是自然发生的，有时这是优化器有意为之的。从传授知识的角度出发，用一个简单的术语来代表这些机制有利于接下来的讲述，我们把它们统称为*辅助式随机读*。请注意，这一术语并未被任何一个 DBMS 使用。

17 自动跳跃式顺序读

从定义上看，如果一系列不连续的行被按照同一个方向扫描，那么访问模式将会是跳跃式顺序的。于是，每行的平均 I/O 时间自然就比随机访问时间短，跳跃的距离越短则节省的时间越多。比如，当表行是通过一个聚簇索引读取时，访问模式即为跳跃式顺序的，我们将在适当的地方再讲这一点。跳跃式顺序读的好处能够在两种情况下被放大：

1. 磁盘服务器可能注意到对某一驱动器的访问是顺序的（或者几乎是顺序的），于是服务器开始向前提前读取多个页。
2. DBMS 可能注意到 SELECT 语句正以顺序的或几乎顺序的方式访问索引或表页，于是 DBMS 便开始向前提前读取多个页，这在 DB2 for z/OS 中被称为动态预读。

列表预读

在上一个例子中，由于表和索引行被访问的顺序是一致的，所以很方便地获得了成本降低的收益。事实上 DB2 for z/OS 能够在表和索引行顺序不一致的情况下主动创造跳跃式顺序访问。为了做到这一点，它需要访问所有满足条件的索引行，然后按照表页的顺序对其进行排序后再访问表行。图 2.6 和图 2.7 对使用和不使用列表预读进行了对比，图中的数字代表了操作的顺序。

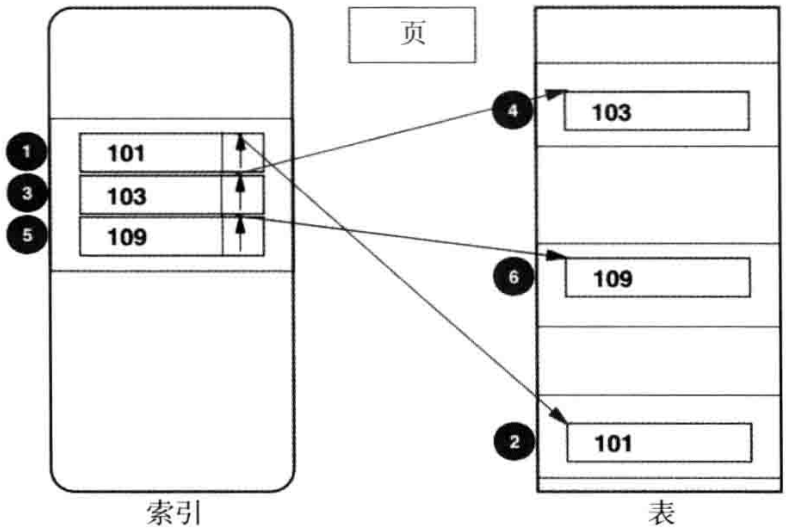


图 2.6　传统的索引扫描

18 数据块预读

当表行和索引行的访问顺序不一致时，Oracle 会使用数据块预读这一特性。在这种方式下，如图 2.8 所示，DBMS 先从索引片上收集指针，然后再进行多重随机 I/O 来并行地读取表行。如果第 4、5、6 步所代表的表行分别位于三个不同的磁盘驱动器上，那么这三次随机 I/O 将被并行执行。就像列

表预读一样，我们可以用图 2.6 和图 2.8 来对比在访问路径中使用和不使用数据块预读的情况。

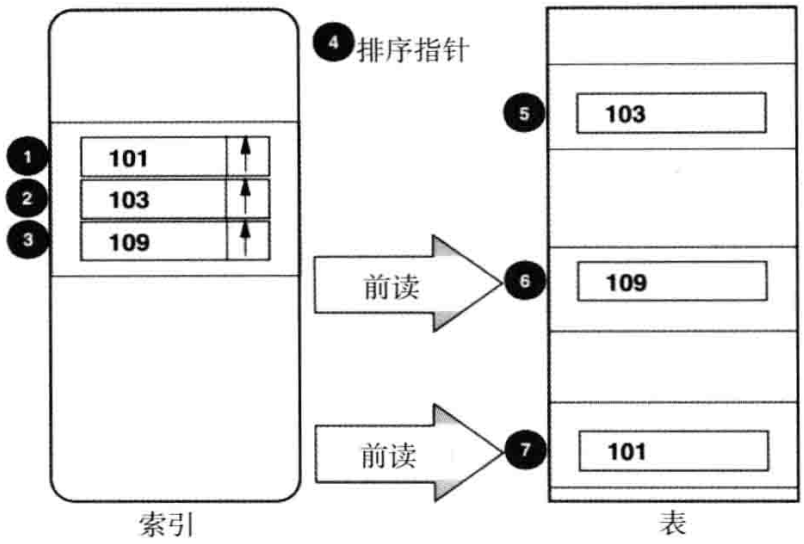


图 2.7　DB2 的列表预读

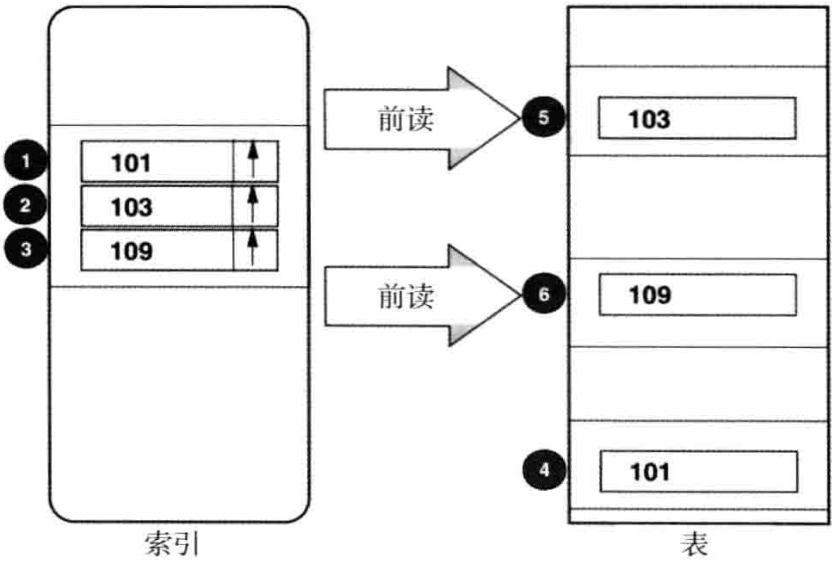


图 2.8　Oracle 数据块预读

在结束辅助式随机读的话题之前，我们还需要考虑一下结果集被获取的顺序。一个索引也许能够自动提供正确的顺序，但上述所讨论的机制也许会在访问表行之前就破坏了这一顺序，因此，也就需要对结果集进行一次排序了。

评注

贯穿整本书，我们将提到三种类型的读 I/O 操作：同步读、顺序读和辅助式随机读。为了保证估算过程的有效性，前期我们将只讨论前两种类型，

有关辅助式随机读取的估算将在第 15 章中详细讨论。

注意，SQL Server 使用术语*索引前读*，而 Oracle 使用术语*索引跳跃扫描*。前者是指提前向前读取下一组叶子页，而后者是指读取多个索引片而不进行全索引扫描。

19 辅助式顺序读

当要扫描一张大表时，优化器可能会选择开启并行机制。例如，它可能会将一个游标拆分为多个用范围谓词限定的游标，每一个游标扫描一个索引片。当有多个处理器和磁盘驱动器可用时，所花费的时间将相应地减少。我们将在第 15 章讨论这个话题。请注意，*辅助式顺序读*这一术语同样也未被任何一个 DBMS 使用过。

同步 I/O 和异步 I/O

在讨论过这些不同的访问技术之后，是时候讨论最后一个关键点了，同步 I/O 和异步 I/O，如图 2.9 所示。

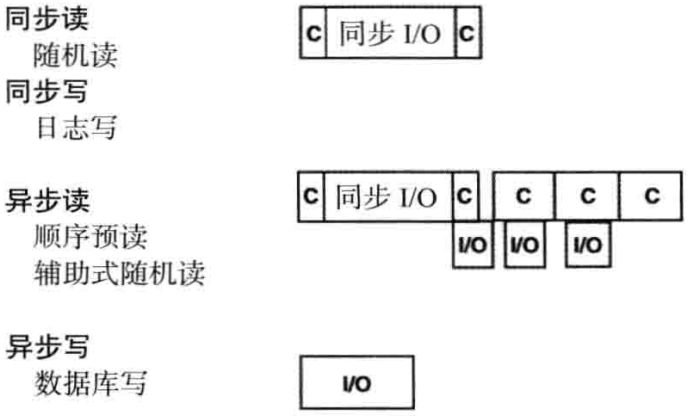


图 2.9　同步 I/O 和异步 I/O

术语同步 I/O 是指在进行 I/O 操作时，DBMS 不能继续进行其他操作，它必须等待，直至 I/O 操作完成。例如，在一次同步读操作中，我们必须先定位到我们所需的行（在图中用“C”表示起始部分的 CPU 时间），随后访问该页并处理该行（在图中展示为第二部分的 CPU 时间），每一步都必须等待，直至上一步完成。

而另一方面，异步读是当前一步的页尚在处理时就被提前发起了，这一处理时间和 I/O 时间之间可能有很大一部分重叠。理想情况下，在这些页被
20 实际处理之前，异步 I/O 就已经完成了。每一组页都以这种方式被预读然后再处理，图 2.9 展示了这一过程。注意，整个预读过程从一次同步读开始，然后才开始预读过程，以此来最小化首次读取的等待时间。

当 DBMS 请求一个页时，磁盘系统可能也会将接下来的一组页加载至磁盘缓冲区中（它预测这些页可能很快就会被请求到）。这一组页可能是条带中的剩余部分，可能是磁轨的剩余部分，也可能是多个条带（我们将很快讨论到条带）。我们将这种机制称为*磁盘前读*。

大部分的数据库写是异步的，以达到对性能的影响最小化。这一方式导致的最主要的影响是加重了磁盘环境的负载，而这反过来会影响读 I/O 的性能。

硬件特性

在本书写作时，数据库服务器所使用的磁盘驱动器在性能表现上相差并不大。它们的转速为 10 000 转/min ~ 15 000 转/min，平均寻道时间为 3 ms 或 4 ms。我们所建议的从磁盘驱动器随机读取的平均时间估算值（10 ms）——包括驱动器排队以及从服务器缓冲区移至数据库缓冲池的传输时间——对于所有当前的磁盘系统都是适用的。

而另一方面，顺序读取的时间则由于配置的不同而差别很大。它不仅仅取决于连接（和最后的争用）的带宽，还取决于并行度。RAID 条带为一个单线程提供了潜在的并行预读能力。*强烈建议*在使用我们所建议的 0.1ms 每 4KB 大小的页（参考第 6 章）之前，对环境的顺序读进行速度测试。

除了 I/O 时间估值以外，磁盘空间和内存的成本也会对索引设计造成影响。 21

本地磁盘只能提供物理数据存储能力，无法提供磁盘服务器所能提供的其他额外功能（比如容错、读写缓冲区、条带等），因此它的价格非常低。

磁盘服务器是具备了多个处理器和大量内存的计算机。最先进的磁盘服务器有容错能力：所有必要的组件都有一份冗余，并且软件能够支持将操作快速转移至另一个备用单元上。一个几百万兆大小的高性能容错磁盘服务器可能价值 200 万美元。于是，每千兆大小空间的成本将会是 500 美元（购买价）或者是每月 50 美元（外包的硬件）。

本地磁盘和磁盘服务器用的都是标准的磁盘驱动器。磁盘驱动器越大，平均每千兆的成本越低。例如，一个 145 GB 的驱动器比 8 个 18 GB 的驱动器更便宜。然而不幸的是，在相同的访问密度下（平均每千兆空间每秒的 I/O 次数），这也同时意味着更久的排队时间。

内存的成本在过去的几年中大幅降低。Intel 服务器（Windows 和 Linux）上平均每 100 MB 大小的随机存取内存（RAM）的成本约为 500 美元，而 RISC（专门品牌的 UNIX 和 Linux）和大型服务器（z/OS 和 Linux）上的成本则为 10 000 美元。在 32 位寻址系统中，数据库缓冲池的大小最大可达 1000 MB

（如 Windows 服务器），而大型服务器的数据库缓冲池的大小则最大可达几千 MB，因为它拥有多个地址空间以支持多个缓冲池。在接下来的几年中，64 位寻址将越来越普遍，这意味着能够支持比现在大得多的缓冲池。如果内存（RAM）的价格继续下降，那么届时大小为 10 万 MB 的数据库缓冲池将很常见。

磁盘服务器的读缓冲的价格与 RISC 服务器的内存的价格相当。购买 64 GB 的读缓冲区（而非 64 GB 的服务器内存）的一个主要原因是，32 位的软件无法利用 64 GB 作为缓冲池。

贯穿本书，我们将使用如下成本假设：

CPU 时间	每小时 1000 美元，基于每个处理器 250mips 的能力
内存	每月每百万兆大小 1000 美元
磁盘空间	每月每百万兆大小 50 美元

这些是当前外包大型服务器的可能值。当然，每一个索引设计者都应该弄清自己系统的值，也许会比上述所假设的值低得多。

DBMS 特性

页

表页的大小限定了表行的最大长度。通常，一个表行必须能够存入一
22 个表页中，一个索引行也必须能够存入一个叶子页中。如果一张表的平均行长大于表页大小的三分之一，那么空间利用率将很糟。例如，一个 4KB 大小的页只能存储一个长度为 2100 字节的行。无法利用的空间问题对于索引更为严重。由于新的索引行必须根据索引键的值放在相应的某个叶子页中，因此在加载和重组后，许多索引的叶子页都应该有能够容纳多个索引行的空闲空间。因此，如果索引行的长度大于叶子页大小的 20%，就有可能造成糟糕的空间利用率和频繁的叶子页分裂。我们会在第 11 章中讨论更多有关这一问题的内容。

在当前的磁盘条件下，磁盘的一次旋转需要花费 4 ms（15 000 rpm）或 6 ms（10 000 rpm）。由于一个磁轨的容量通常大于 100 千字节（KB），因此 2KB、4KB 和 8KB 大小的页的随机读取时间大致是相等的。不过，RAID 磁盘的条带大小必须大到足以存放一个页，否则，为了读取一个页就可能需要访问不止一个磁盘驱动器。

在如今的大部分环境下，顺序处理过程中一次 I/O 操作会同时将多个页

加载至缓冲池中，例如，多个页会在一次磁盘转动中被传输进来。所以，页的大小对于顺序读的性能影响并不大。

SQL Server 2000 对表页和索引页只支持一种大小：8KB。索引行的最大长度为 900 字节。

Oracle 使用术语块来代表术语页。它所支持的块大小（BLOCK_SIZE）为 2KB、4KB、8KB、16KB、32KB 和 64KB，但有些操作系统可能对所能支持的块大小有所限制。索引行的最大长度为 BLOCK_SIZE 的 40%。为了尽可能地保持简单，我们将在本书中使用术语页而非术语块，希望读者不要介意。

DB2 for z/OS 支持 4KB、8KB、16KB 和 32KB 大小的表页，但只支持 4KB 大小的索引页。在 V7 版本中，索引行的长度上限为 255 字节，不过在 V8 版本中，这一上限变为 2000 字节了。

DB2 for LUW 支持 4KB、8KB、16KB 和 32KB 大小的表页或索引页。索引行的长度上限为 1024 字节。

表聚簇

通常情况下，一个表页中只包含一张表中的数据。Oracle 提供了一个选项以支持在一个表页中交错存储多个相关表的数据，这与存储多种不同段类型的分层 IMS 数据库记录类似。例如，一份保单可能在 5 张表中存有相关的数据。单据号可能是其中一张表的主键和其余 4 张表的外键。当把所有与这个保单相关的数据交错存储在一张表中时，这些数据可能都能够保存在同一页中。于是，读取某个保单相关的所有数据可能就只需要进行 1 次 I/O 操作而非 5 次。而另一方面，年长的读者可能知道，交错存储多张表的数据可能会在其他方面引起一些问题。

索引行

23

在一个索引上，当前 DBMS 所能支持的列的数量上限分别为：SQL Server 支持 16 个，Oracle 支持 32 个，DB2 for z/OS 支持 64 个，DB2 for LUW 支持 16 个（更详细的内容请见第 12 章）。

在一些产品中，索引变长的列会有一些限制。如果只支持定长的索引行，那么 DBMS 会将变长列扩展至最大长度，然后再存储在索引中。由于变长的列变得越来越普遍（原因有很多，比如 Java 的发展）——即便在过去很少使用变长列的环境也是如此——对于变长索引列（及索引行）的支持在新

的数据库版本中已经很常见了。例如，DB2 for z/OS 在 V8 版本中已经完全支持变长的索引列了。

通常，索引键由所有被复制到索引上的列组成，它决定了索引条目的顺序。在唯一索引中，索引条目等同于索引行。在非唯一索引中，对于每一个唯一的索引键值，都存在一个索引条目，以及指向每一个满足该索引键的表行的指针。例如，DB2 for LUW 允许在索引行的最后存储非键值列。除此以外，每一个索引条目还需要一些特定数量的控制信息，用以将其按键值顺序串联起来。在本书中，为了计算每页所能存储的索引行数量，我们假设这一控制信息所占用的空间约为 10 字节。

表行

我们知道有些 DBMS（如 DB2 for z/OS、DB2 for LUW、Informix 和 Ingres）支持聚簇索引，这会对表行的插入位置造成影响。使用聚簇索引的目的是为了使得表行的存储顺序尽可能地与索引行的顺序保持一致。如果表上没有聚簇索引，那么新插入的表行将会被放置在表的最后一个页上，或者被放置在任何一个有足够空闲空间的表页上。

有些 DBMS，如 Oracle 和 SQL Server，并不支持这种会对新加行的表页选择造成影响的聚簇索引。然而，无论何种 DBMS，都可以通过频繁地重组表来使得表行按照所需的顺序存储——通过在重载前经由某个特定的索引读取表行来实现，或者通过在重载前对数据进行排序来实现。

Oracle 和 SQL Server 支持一个在索引中存储表行的选项，我们将在下一小节中看到。更详细的信息将在第 12 章中讨论。

索引组织表

如果一个表的行不是特别长，那么可以考虑将表上所有的列复制到索引
24 上，以加快 SELECT 的执行速度。如此一来，表就变得冗余了。有些 DBMS 有去除多余表的选项。若使用这一选项，那么其中一个索引的叶子页将用于存储表行。

在 Oracle 中，这一选项被称为索引组织表，包含表行的索引被称为主键索引。在 SQL Server 中，可以用选项 CLUSTERED 创建一个存储表行的索引。在这两种环境中，其余的索引（在 Oracle 中被称为次级索引，在 SQL Server 中被称为非聚集索引）都指向包含表行的那个索引。

索引组织表的最明显的好处就是节省磁盘空间。另外，INSERT、UPDATE 和 DELETE 操作的速度也更快，因为少了一个需要更新的页。

然而，这给其余的索引带来了不利。如果这些索引使用的是直接指向表行的指针（指针中包含页号），那么主键（聚集）索引的一次叶子页分裂将导致其余索引上大量的磁盘 I/O。任何对于主键索引键的更新操作，由于需要移动索引行，都会导致 DBMS 去更新那些指向这一索引行的其他索引行。这也就是为什么 SQL Server 会把主键索引的索引键作为指向聚集索引的指针值。这么做能避免叶子页分裂所带来的额外负载，但如果聚集索引的键值很长，那么在这种方式下，那些非聚集索引将变得更大。此外，任何经由非聚集索引的访问都需要涉及两组非叶子页的读取，第一组是非聚集索引的非叶子页，第二组是聚集索引的非叶子页。不过，这一额外负载不是主要问题，因为非叶子页是被缓存在缓冲池中的。

本书中所介绍的技术对于索引组织表也同样适用，虽然在图表中总是会画出单独的表。如果使用了索引组织表，那么主键表应当被视为一个满足所有 SELECT 语句的宽聚簇索引——我们将在第 4 章中讨论这一点。索引行的顺序是由索引键决定的。其余的列则为非键列。

注意，在 SQL Server 中聚集索引并非必须为主键索引。然而，为了降低指针维护的成本，通常会选择一个键值不会被更新的索引作为聚集索引，如主键索引或候选键索引。在大部分索引中（非键值列选项将在后续讨论），所有的索引列一起构成了索引键，因此，在其余的索引中很难找到键值不会被更新的索引。

页邻接

逻辑上相邻的页（如叶子页 1 和叶子页 2）在磁盘上是物理相邻的吗？如果是物理相邻的，那么顺序读将会非常快（图 2.10 中的级别 2）。

三个级别：
读取一个页，得到许多行
读取一个磁轨，得到许多页
磁盘服务器提前从驱动器上将数据读取至读缓冲区中

- **级别 1 是自动的**
  如果每个 4KB 大小的页包含 10 行记录，那么 I/O 时间 = 1 ms 每行
- **级别 2 是由 DBMS 或磁盘系统支持的**
  可能将顺序 I/O 的时间降至 0.1 ms 每行
- **级别 3 是由磁盘服务器支持的**
  可能将顺序 I/O 的时间降至 0.01 ms 每行

图 2.10　页邻接

在某些较早的 DBMS（如 SQL/DS 和 SQL Server 的早期版本）中，一个索引或表的页可以分布在一个大文件的各个地方。在这种情况下，随机读取和顺序读取在性能方面的唯一区别就是，一些逻辑上相邻的行被保存在同一个页中（图 2.10 中的级别 1）。读取下一个页将需要进行一次随机 I/O。
25 如果每页包含 10 行数据且一次随机 I/O 需耗费 10 ms，那么顺序读取的 I/O 时间将会是 1 ms 每行。

SQL Server 在为索引和表分配空间时，一次性分配 8 个 8KB 的页。DB2 for z/OS 以区间的粒度分配空间，一个区间的大小可能有许多 MB，通常一个中等大小的索引或表的所有页都能保存在一个区间中。于是，逻辑上相邻的页在物理上也互相挨着了。在 Oracle（和其他一些系统）中，页的放置位置取决于所选择的文件选项。

现今许多数据库都存储在 RAID 5 或 RAID 10 的磁盘上。RAID 5 提供了具有冗余能力的条带。RAID 10，即 RAID 1 + RAID 0，提供了具有镜像能力的条带。

术语冗余和镜像的定义可以在词汇表中找到。RAID 条带是指将一张表或一个索引的第一个条带存在驱动器 1 上，将第二个条带存在驱动器 2 上，以此类推。很明显，这能够将负载平均分配在一组驱动器上，但这会对顺序读的性能产生怎样的影响呢？令人高兴的是，这一影响可能是积极的。

让我们考虑一次全表扫描，被扫描的表页在 7 个驱动器上做条带。于是，磁盘服务器可以从 7 个驱动器上进行并发前读了。当 DBMS 请求下一组页时，它们很可能已经在磁盘服务器的读缓冲区中了。这一预读操作可能将 I/O 时间降至 0.1 ms 每 4KB 大小的页（图 2.10 中的级别 3）。当拥有快速通道且磁盘服务器能够发现一个文件正在被顺序访问时，就能使时间降到 0.1 ms。

B 树索引的替代品

位图索引

位图索引由针对各个不同列值的位图（位向量）组成。在每一个位图中，表中的每一行对应一个位。若该行满足位图条件，则该位被置为 1。

对于复杂且不可预测的组合谓词的大表查询，适合用位图索引。因为用位图索引进行与和或计算（在第 6 章和第 10 章中讨论）的速度非常快，即
26 便表行数量达亿级也是如此。而若使用 B 树索引进行同样的操作则需要收集大量指针并进行排序。

另一方面，一个包含合适的列的 B 树索引能够避免表访问。这一点很重要，因为对一张大表进行随机 I/O 是非常慢的（约 10 ms）。而若使用位

图索引，那么就必须访问表行，除非 SELECT 列表只包含 COUNT。因此，使用位图索引可能比使用一个合适的（宽）B 树索引的总执行时间长得多。

位图索引应当在满足以下条件的情况下使用：

1. 可能的谓词组合数量太多了，以至于设计足够的 B 树索引是不可行的。
2. 单个谓词具有很高的过滤因子（将在第 3 章中讨论），但组合起来之后（WHERE 分句）具有很低的过滤因子，或者 SELECT 列表中只包含 COUNT。
3. 更新操作是批量进行的（不存在锁争用）。

散列

散列，或者说随机化，是在已知主键值的情况下读取一个表行的最快方式。当存储一行数据时，表页是由一个随机选择器选择的，该选择器将主键值转换为 1 至 *N* 之间的某个页号。如果该页已满，那么该行将被放在另一个串联至这个主页的页中。当发起一个 SELECT...WHERE PK = :PK 查询时，随机发生器将被再次用来决定主页的页号。该行要么能在该页中被找到，要么能通过遍历从该页开始的页链表中找到。

随机发生器在一些非关系型的 DBMS 中经常被用到，如 IMS 和 IDMS。当区间大小（*N*）设置得合适（对应于 70%的空间利用率）时，读取一行记录所需的 I/O 次数可以低至 1.1，这相比索引要低得多（一个 3 层索引需要 3 次 I/O，其中第 3 次用于读取记录本身）。然而，使用随机发生器的方式需要不断地监视并调整空间利用率。当添加了许多记录后，页链表的长度会增长，而 I/O 的次数也会随之大幅增加。此外，随机发生器还无法支持范围谓词。Oracle 提供了一个将主键值通过散列转换为数据库页号的选项。

聚簇的许多含义

聚簇是一个在关系型文献中被广泛使用的术语。它也是许多混淆的产生源头，因为它在不同产品中的含义各不相同。

在 DB2（z/OS、LUW、VM 和 VSE）中，聚簇索引是指定义了新插入的表行所在表页的索引。如果索引行的顺序和表行的顺序之间具有强关联性，那么就可以说该索引是聚集的。一张表上只能有一个聚簇索引，但是，在某个特定的时间，可能有多个索引是聚集的。索引的聚簇比例（CLUSTERRATIO）是指索引行和表行顺序之间关联度的一个量度。优化器会使用这一测量值来估算 I/O 时间。

27

DB2 的表上通常都有一个聚簇索引。

在 SQL Server 中，存储表行的索引被称为是聚集的，只有当需要一张索引组织表时才定义一个聚集索引。其余的索引（在 SQL Server 中的术语为非聚簇索引）都指向这一聚集索引。

在 Oracle 中，聚簇一词被用于代表将多个表的行交错存储（聚簇的表）的选项。该词与我们之前所讨论的限定表行顺序的聚簇索引毫不相干。

DB2 for LUW V8 版本有一个称为多维度聚簇的选项，它使得相关的行能够被放在一起。更多细节请见第 13 章。

重要说明

在本书所有的图表中，C 用于标记定义了新添加表行的存储位置的索引。在我们的计算中，我们假定表行的存储顺序和这个索引一致。对于不支持聚簇索引的产品，表行的顺序在重组或重新加载表时被指定。

第3章

SQL处理过程

- 如何有效使用索引和表结构来处理SQL语句
- SQL处理的一些概念
- 谓词
- 优化器及访问路径
- 索引片、匹配索引扫描、匹配列、索引过滤和过滤列
- 优化器何时确定访问路径
- 监控优化器
- 使用统计信息和所需FETCH调用的次数来指导优化器的行为
- 过滤因子、选择性和基数的概念及其对索引设计的影响
- 结果集物化及其影响

简介

现在我们对表及索引的结构已经有了初步的理解，我们还知道了它们如何与缓冲池、磁盘和磁盘服务器等物理结构进行关联，以及如何使用后者来实现——让SQL处理过程访问到数据。现在，是时候讨论SQL调用的处理过程了。

早期，大部分SQL语句的处理过程依赖于所使用的不同的关系型DBMS。它们之间有许多相似点，但也有许多差别。为了避免将基础问题与数据库特性混淆，我们首先会概括地讨论基础的处理过程，而各DBMS之间的不同将会在合适的时机进行阐述。关于这些过程更详细的细节讨论将在本书中合适的地方给出。另外，本章中使用的所有术语都能在本书末尾的词汇表中找到。

谓词

WHERE子句由一个或者多个谓词（搜索参数）组成。SQL 3.1中有三

个简单谓词，它们是：

- SEX = 'M'
- WEIGHT > 90
- HEIGHT > 190

SQL 3.1

```
WHERE   SEX = 'M'
        AND
        (WEIGHT > 90
        OR
        HEIGHT > 190)
```

同样，它们也可以被认为是两个组合谓词：

- WEIGHT > 90 OR HEIGHT > 190
- SEX = 'M' AND (WEIGHT > 90 OR HEIGHT > 190)

谓词表达式是索引设计的主要入手点。如果一个索引能够满足 SELECT 查询语句的所有谓词表达式，那么优化器就很有可能建立起一个高效的访问路径。

评注

Chris Date 在他的很多著作中都极不情愿使用谓词这一术语：

> 本书中，我们沿用了传统的数据库的使用习惯，统一使用“谓词”来指代条件表达式。但严格来讲，这种使用方式是不严谨的。更为正确的表述应该是条件表达式或者真值表达式（第 79 页）。

优化器及访问路径

关系型数据库的一大优势就是，用户无须关心数据的访问方式。其访问路径是由 DBMS 的一个组件，即优化器来确定的。虽然不同的关系型系统的优化器各不相同，但它们都是在系统收集的统计信息的基础上，尽可能以最高效的方式访问数据。显然，优化器是 SQL 处理过程的核心，也是本书讨论的重点。

在 SQL 语句能够被真正执行之前，优化器必须首先确定如何访问数据。这包括：应该使用哪一个索引，索引的访问方式如何，是否要使用辅助式随机读，等等。所有的这些细节都包含在访问路径中。正如我们在第 2 章所使
31 用的例子，如图 3.1 所示，该访问路径定义了一次索引片段的顺序扫描，以及对每一个所需的表页进行的一次同步读。

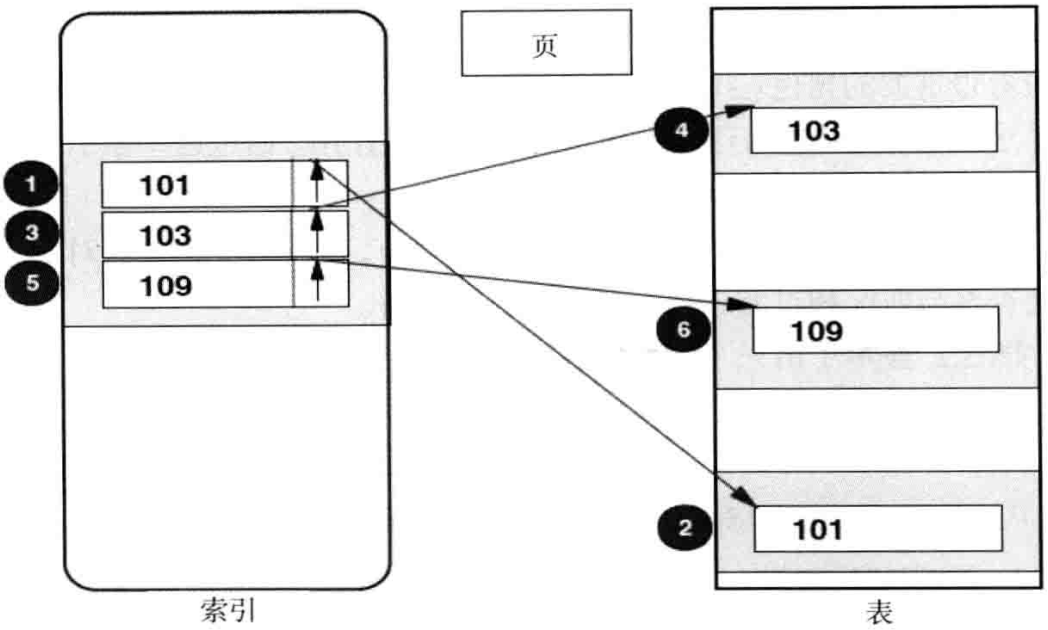


图 3.1　传统的索引扫描

索引片及匹配列

如图 3.1 所示，索引的一个窄的片段将会被顺序扫描，其上索引行的值在 100 至 110 之间，相应的表行将通过同步读从表中读取，除非该页已在缓冲池中。所以访问路径的成本很大程度上取决于索引片的厚度，即谓词表达式确定的值域范围。索引片越厚，需要顺序扫描的索引页就越多，需要处理的索引记录也就越多，而最大开销还是来自于增加的对表的同步读操作，每次表页读取需要 10 ms。相应的，如果索引片比较窄，就会显著减少索引访问的那部分开销，但主要的成本节省还是在更少的对表的同步读取上。

并不是所有的关系数据库系统都使用索引片这种描述方式，不同的产品有自己的术语，但我们认为索引片是独立于数据库产品的更具描述性的术语。另外一种使用较为广泛的描述索引片的方法是定义索引匹配列的数量。在上面的例子中，仅仅只有一个匹配的列，其值域范围从 100 到 110。这个匹配列实际上定义了索引片的大小。如果 WHERE 子句中有第二个列，而这 32
个列也在索引上，从而使得这两个列能够一同定义一个更窄的索引片，那么我们将有两个匹配谓词。这样不仅显著减少了索引的处理量，更为重要的是，减少了对表进行同步读的次数。

索引过滤及过滤列

有时候，列可能既存在于 WHERE 子句中，也存在于索引中，但这个列却不能参与索引片的定义（这其中的原因非常复杂，我们将在本书中持续讨论这个问题，现在我们只需要知道并不是所有的索引列都能够定义索引片

的大小）。不过这些列仍然能够减少回表进行同步读的次数，所以这些列仍扮演着很重要的角色。我们称这些列为过滤列，有些关系型数据库系统也确实是这么命名的，因为这些列确实起到了过滤作用。通过这些索引上的过滤列能够避免对表行的访问。

在这个阶段我们并不准备讨论过多的细节，仅仅为了让大家对于判断谓词能否参与匹配和过滤过程有一个初步的理解。

图 3.2 显示了由三个谓词组成的 WHERE 子句，每一个谓词的列都是索引的一部分。列 D，即索引中的最后一列，并不包含在谓词表达式中。

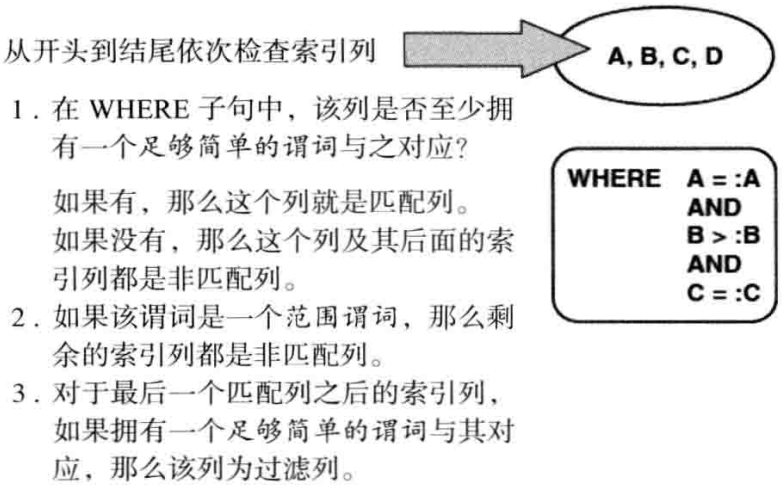


图 3.2　谓词匹配和过滤字段

首先，我们来看第一个索引列 A，这个列出现在一个等值谓词中，这也是所有谓词中最简单的一个。很明显，列 A 是匹配列，将被用于定义索引片。

33 接下来，我们来看第二个索引列 B。就像图 3.1 中的 BETWEEN 谓词一样，列 B 也简单到足以成为匹配列，它也将被用于定义索引片。

图 3.2 中的第 2 条表明，由于列 B 是一个范围谓词，所以剩下的列 C 将不能参与匹配过程——它不能定义索引片。不过第 3 条随之指出，列 C 能够避免不必要的表访问，因为它参与了索引片的过滤过程。从效果上而言，列 C 的作用与列 A 和列 B 几乎同等重要，列 C 无法参与匹配过程只是使得索引片更厚了一点而已。

这里最困难的问题是简单与复杂的谓词的判定标准是什么？不同的 DBMS 有不同的标准，这部分将放到第 6 章再进行讨论。

总结起来，表 3.2 所示的 WHERE 子句有两个匹配列，列 A 和列 B，它们定义了扫描的索引片。除此之外，还有一个列 C 作为过滤列。所以，只有当一行同时满足这三个谓词时才会访问表中的数据。假如没有列 B 的谓词表达式，那么就只有列 A 一个匹配列，但是列 C 仍然可以用来过滤。如

果列 B 的谓词表达式是等值谓词，那么这三个列都可以作为匹配列，来显著减少索引片段的访问。最后，假如取消列 A 的谓词表达式，那么索引片段就是整个索引的大小，列 B 和列 C 都仅仅只能用来过滤。

访问路径术语

很遗憾，描述访问路径的专业术语并没有统一的标准，甚至访问路径这一术语本身也是如此，另一个经常被用来指代访问路径的术语是执行计划。在本书中，我们使用访问路径来描述数据访问的方式，而使用执行计划来描述 DBMS 提供的 EXPLAIN 工具所产生的输出结果。这两个术语只有细微的差别，将其互换使用也不会有什么问题。接下来，进入我们关注的重点。

匹配谓词有时也称作范围限定谓词。当有合适的索引存在时，如果优化器能够识别到某谓词为匹配谓词，那么我们就称这个谓词是可索引的（DB2 for z/OS）或者是可搜索的（SQL Server、DB2 for VM 和 VSE）。反之就是不可索引的或者不可搜索的。在一些 Oracle 的书籍中，当讨论到一些困难得无法参与匹配过程的谓词时，会用到索引抑制这一术语。

SQL Server 使用表查找这一术语来描述使用索引并且需要读取表行的访问路径。这不同于仅仅使用索引的访问路径。消除表访问的最显而易见的方式就是将缺失的列添加至索引上。许多 SQL Server 的书籍将这种能够避免某个 SELECT 调用的表访问的索引称为覆盖索引。使用覆盖索引的 SELECT 语句有时会被称为覆盖 SELECT。

在DB2中，这种对于优化器过于复杂而无法参与索引过滤的谓词被称为二阶段谓词。与其相对的就是一阶段谓词。在许多数据库产品的手册中根本就没有关于索引过滤的讨论。但愿这意味着这些产品在逻辑上可行时会自动进行索引过滤。不过，在做出这一假定之前，有必要做一个如下的简单实验。

34

创建一张拥有 4 个字段的表并插入 100 000 条记录，前三个字段分别是 A、B 和 C，每一列分别有 100 个不同的值。添加列 D 以确保该表会被访问。行长必须足够长，以保证每个页中只有一行记录。创建一个索引(A, B, C)并运行查询语句 SELECT D FROM TABLE WHERE A = :A AND C = :C 来确定访问表行的次数。我们假设使用了索引结构并且获取了所有满足条件的表行记录。因为一行占用一页的空间，所以该表有 100 000 个页。如果访问的表页数量接近 10 个（0.01 × 0.01 × 100 000；第一个 0.01 用于匹配，第二个 0.01 用于过滤），那么谓词 C = :C 肯定被用于进行索引过滤了。如果访问的表页接近 1000 个（0.01 × 100 000；只有一个 0.01，用于匹配），那么就代表没有使用索引过滤。还可以使用复杂的谓词来做这个实验，用以验证谓词对于优化器而言是否太困难了。

第 6 章将对困难和非常困难的谓词进行详细讨论，包括与其相关的匹配和过滤。

监控优化器

当发现一个慢 SQL 的时候，通常首先被怀疑的对象就是优化器，可能是优化器选择了错误的访问路径。关系型 DBMS 通常都提供一个工具，用于解释优化器决定使用某个访问路径的原因，这一工具被称为 EXPLAIN、SHOW PLAN 或者 EXPLAIN PLAN，我们将在第 7 章中对其进行讨论。从现在开始，本书中所有提及 EXPLAIN 之处都应当被认为适用于所有这三种术语。

帮助优化器（统计信息）

如果优化器进行成本估算所使用的统计信息不完整，那么优化器就很容易做出错误的决定，可能是收集统计信息时所用的选项太有限，也可能是统计信息已经过时了。这些统计信息通常是在需要时由专门的程序进行收集的，比如 DB2 for z/OS 中的 RUNSTATS 程序。

正常情况下，默认收集的统计信息包括了基础的信息，如每张表的记录数和表页数、叶子页数、每个索引的聚簇率、某些列或者列组的不同值个数（被称为基数），以及某些列的最大值、最小值（或者第二大值和第二小值）等。其他可选的统计信息的选项能够提供更多关于列和列组的值分布情况，
35 比如最常见的 *N* 个值连同其对应的表行数量。许多数据库产品（比如 Oracle、SQL Server 和 DB2 for LUW）还支持以直方图的形式采集列值分布（即在用户所定义的值域范围内表行所占的比例 *N*%）。

帮助优化器（FETCH 调用的次数）

基于成本的优化器在评估不同访问路径的成本时会FETCH所有满足条件的记录——除非有特别说明。如果我们不需要所有的记录集合，那么可以设置仅FETCH第前*n*行。SQL Server数据库可以在SELECT语句的最后添加：

```
OPTIONS (FAST n)
```

对于Oracle，访问路径提示的使用方法为：

```
SELECT /*+ FIRST_ROWS(n)*/
```

[(n)选项只能在Oracle 9i及以上版本中使用]。

DB2 for z/OS的语法是：

```
OPTIMIZE FOR n ROWS
```

SQL 3.2S、SQL 3.2O和SQL 3.2D分别举例如下：

SQL 3.2S

```
DECLARE LASTINV CURSOR FOR
SELECT      INO, IDATE, IEUR
FROM        INVOICE
WHERE       CNO = :CNO
ORDER BY    INO DESC
OPTIONS (FAST 1)
```

SQL 3.2O

```
DECLARE LASTINV CURSOR FOR
SELECT /*+ FIRST_ROWS(1)*/
            INO, IDATE, IEUR
FROM        INVOICE
WHERE       CNO = :CNO
ORDER BY    INO DESC
```

36

SQL 3.2D

```
DECLARE LASTINV CURSOR FOR
SELECT      INO, IDATE, IEUR
FROM        INVOICE
WHERE       CNO = :CNO
ORDER BY    INO DESC
OPTIMIZE FOR 1 ROW
```

请注意，这些选项并不限制可能发起的 FETCH 调用的次数。如果需要限制次数，则 SQL Server 提供了 SELECT TOP (n)，DB2 for z/OS 提供了 FETCH FIRST n ROW(S) ONLY，都可以用来达到这一目的。

何时确定访问路径

现阶段，我们必须理解的优化器相关的问题就是图3.3所描述的内容。很明显，在每次SQL语句执行时都进行一次访问路径选择要比仅做一次消耗更多的资源，所以在应用程序开发的过程中，基于成本的优化器进行访问路径选择的处理成本不容忽视。

不太明显但有时很重要的一点是，在每次 SQL 语句执行时进行访问路径选择，这种方式可能会为优化器提供一个选择更好的访问路径的机会，因为这种情况下我们使用的是确定的值而不是绑定的变量，比如谓词 WHERE

SALARY > 1 000 000 要比 WHERE SALARY > :SALARY 更明确。

图3.3显示了优化器是如何支持这一选择时，后面将继续对比进行讨论。



图 3.3　优化器何时选择访问路径

37 过滤因子

过滤因子描述了谓词的选择性，即表中满足谓词条件的记录行数所占的比例，它主要依赖于列值的分布情况。因此，当一名女会员加入到会员表中时，谓词 SEX = 'F'的过滤因子就会变大。

谓词 CITY = :CITY 使用平均过滤因子（1/不同 CITY 的个数），类似于特定值的过滤因子（CITY = 'HELSINKI'）。不过这两者之间的不同点对索引的设计和访问路径的选择至关重要。图 3.4 描述了一张拥有百万行记录的用户表，表上 CITY 列的过滤因子是 0.2%，这意味着查询的结果集将包含 2000 行记录。

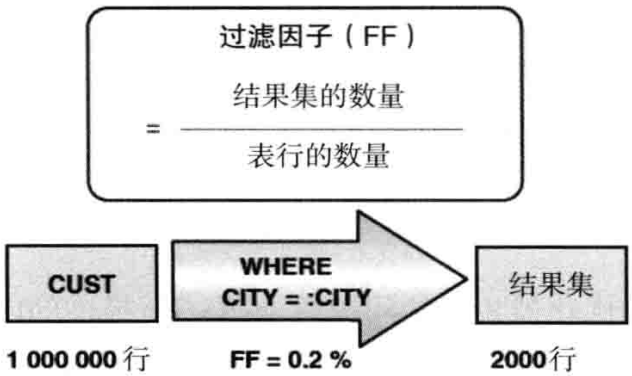


图 3.4　过滤因子是谓词的一个属性

在评估一个索引是否合适时，最差情况下的过滤因子比平均过滤因子更重要，因为最差情况与最差输入相关，即在该输入条件下，基于特定索引的查询将消耗最长的时间。

组合谓词的过滤因子

如果组成谓词的列之间非相关，那么组合谓词的过滤因子可以从单个谓词的过滤因子推导出来。比如组合谓词 CITY = :CITY AND LNAME = :LNAME 的过滤因子，如果列 CITY 和列 LNAME 没有相关性，那么组合谓词的过滤因子就等于谓词 CITY = :CITY 和谓词 LNAME = :LNAME 的过滤因子乘积。如果列 CITY 有 500 个不同的值，列 LNAME 有 10 000 个不同的值，那么组合谓词的过滤因子就是 1/500 × 1/10 000。这意味着列组合 CITY,LNAME 有 5 000 000 个不同的值。然而，在大多数的用户表中，这两列是有相关性的，比如哥本哈根的 Andersens 用户要比伦敦的 Andersens 用户多很多；再比如，英国的某些城市根本没有以 Danish 作为姓的用户。因此，组合谓词的过滤因子可能比两个谓词过滤因子的乘积要低得多。 38

列 CITY 和 BD（生日，mmdd）可能没有相关性。因此，这两列的组合谓词的过滤因子可以按照图 3.5 所描述的按照乘法的方法进行评估。

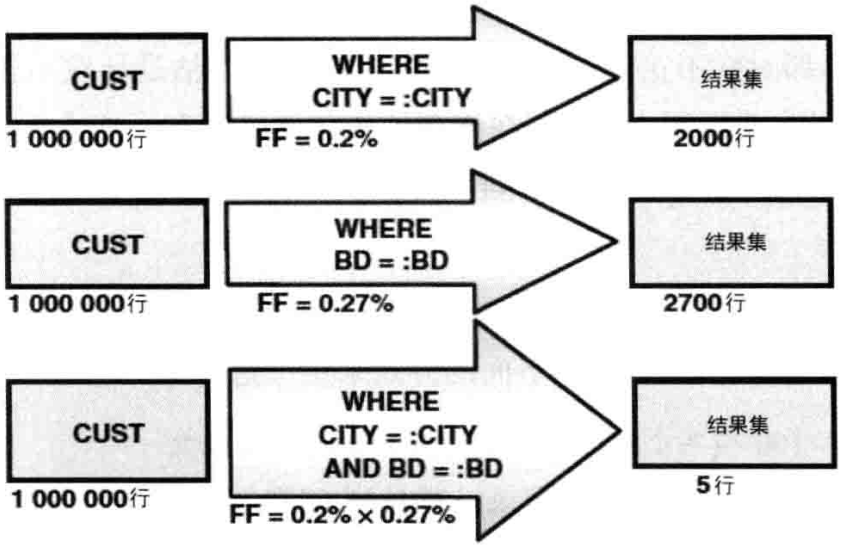


图 3.5　无相关性

来看一个列相关性的极端例子。如图 3.6 所示的 CAR 表，它有列 MAKE 和列 MODEL。谓词 MAKE = :MAKE 和 MODEL = :MODEL 的过滤因子可能分别是 1/100 和 1/1000。组合谓词 MAKE = :MAKE AND MODEL = :MODEL 的组合过滤因子却远小于 1/100 000，可能低至 1/1000，因为，举个例子，我们都知道，只有 Ford 公司制造 Model 7 型车。

在设计索引结构的时候，需要将组合谓词看作一个整体来评估过滤因子，而不能仅仅基于零相关性进行评估。幸运的是，最坏情况下的过滤因子评估通常都比较简单，比如，如果谓词 CITY = :CITY 和 LNAME = :LNAME 的最大过滤因子分别是 10%和 1%，那么组合谓词 CITY = :CITY AND 39

LNAME = :LNAME 的过滤因子最大为 0.1 × 0.01 = 0.001 = 0.1%，因为它与某个具体城市的 LNAME 分布情况有关。

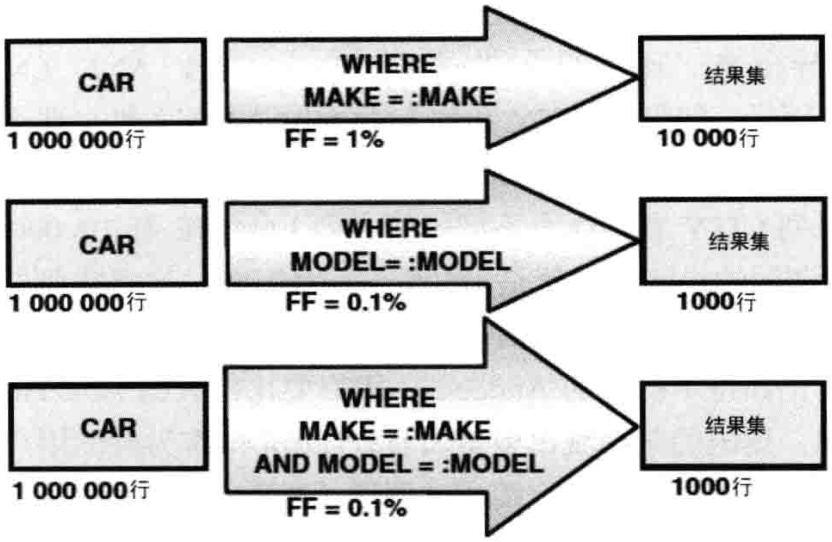


图 3.6　强相关性

优化器在评估可选访问路径的成本时，必须先评估过滤因子，过去的算法远比现在简单，那时候不正确的组合谓词的过滤因子评估是导致不合适的访问路径选择的最主要原因之一。现在许多的优化器已经有了可选项，用于计算或者采样索引列组合的基数（不同值的数量）。

评注

在一些关系型数据库文献中，经常讨论的一个与过滤因子相似的概念是选择率，Vipul Minocha （3）按照下面的方式来定义选择率：

选择率 = 100 ×(某个键值对应的行数) /表的总记录数

如果 CUST 表有 1 000 000 行记录，并且列 CITY 的值等于 HELSINKI 的行数，为 200 000 行。那么值 HELSINKI 的选择率就是 100 × 200 000/1 000 000 = 20%，等于谓词 CITY= ‘HELSINKI’ 的过滤因子，因此，该索引的索引选择性并不好，尤其对于值 HELSINKI。所以 CITY 索引并不是一个好的索引。

选择性是一个容易造成困惑的概念，因为它本身是索引的一个属性。不过情况会变得更糟，如下面参考文献 1 中的一段话所述。

> 厂商的手册或者其他文档中关于选择性的定义，大多是不准确的或者含混的：“不同值的行数除以表中的记录数”（Oracle）；“索引中重复的键值的比例”（Sybase）。对于词语“高选择性”，不同的人有不同的解释，有人认为是“一个大的选择性数字”，也有人认为是“一个低选择性数字”（第 552 页）。

这里，我们不希望增加更多的困扰，过滤因子是一个对于索引设计更有用的概念，且对于性能预测是必不可少的。本书将广泛使用过滤因子这个概念。

过滤因子对索引设计的影响

需要扫描的索引片的大小对访问路径的性能影响至关重要。在当前的硬件条件下，索引片大小的最重要的量度就是需要扫描的索引记录数，即匹配组合谓词的过滤因子与总行数的乘积。按照定义，匹配谓词能够参与定义索引片的大小，即从哪开始，到哪结束。通常索引行的记录数等同于表的记录数，但这其中也存在例外情况，如 Oracle 的索引并不为 NULL 值创建索引行。考虑下面的一个查询（参见 SQL 3.3）：

40

SQL 3.3

```
SELECT      PRICE, COLOR, DEALERNO
FROM        CAR
WHERE       MAKE = :MAKE
            AND
            MODEL = :MODEL
ORDER BY    PRICE
```

如果使用图 3.7 中所示的索引，那么这两个谓词都是匹配谓词。如果组合谓词的过滤因子是 0.1%，那么所需访问的索引片大小将为整个索引的 0.1%。列 MAKE 和列 MODEL 都是匹配列。图 3.7 所示的索引设计对于 SQL 3.3 的 SELECT 语句相对比较合适，尽管它还远远算不上最好的索引，但至少扫描的索引片已经很小了。

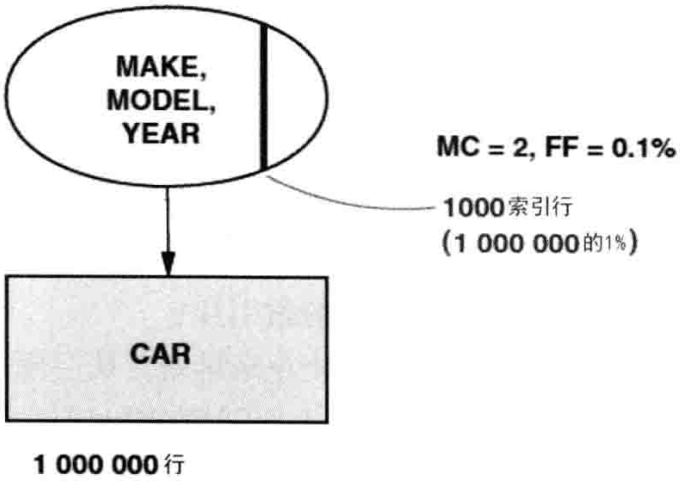


图 3.7　两个匹配列——一个窄索引片

对于 SQL 3.4 中的 SELECT 语句来说，该索引并不太好，因为只有一个匹配列，如图 3.8 所示。

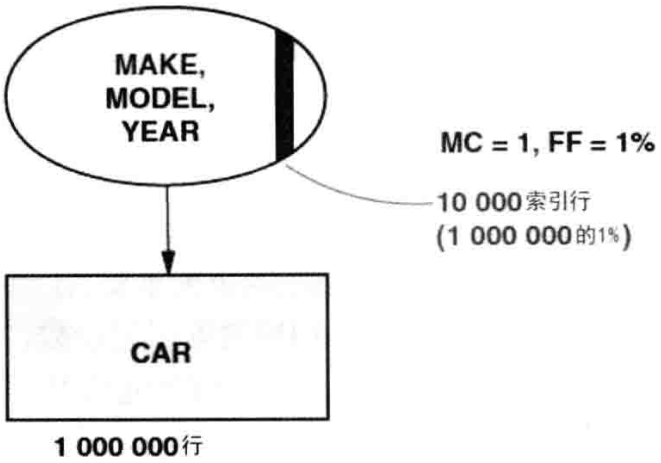


图 3.8　一个匹配列——一个厚索引片

SQL 3.4

```
SELECT    PRICE, COLOR, DEALERNO
FROM      AUTO
WHERE     MAKE = :MAKE
          AND
          YEAR = :YEAR
```

如果 SQL 3.5 中的 SELECT 语句利用索引 SEX,HEIGHT,WEIGHT,CNO 来查找高大的男士，那么只有一个匹配谓词 SEX，其过滤因子正常情况下是 0.5。只有一个匹配谓词的原因将在后面讨论。

SQL 3.5

```
SELECT      LNAME, FNAME, CNO
FROM        CUST
WHERE       SEX = 'M'
            AND
            (WEIGHT > 90
            OR
            HEIGHT > 190)
ORDER BY    LNAME, FNAME
```

结果，如果 CUST 表有 1 000 000 行记录，那么需要扫描的索引片将包含 0.5 × 1 000 000 行 = 500 000 行，一个相当厚的索引片！

41 一些教科书中建议索引列的顺序按照基数的降序来排列。从字面理解，这一建议可能导致错误的索引设计，如索引(CNO,HEIGHT,WEIGHT,SEX)。但在特定的场景下，假设列的顺序不影响 SELECT 语句的查询性能（比如 WHERE A = :A AND B = :B，索引(A,B)和(B,A)是等价的），也不影响更新操作的性能——那么这会是一个合理的建议，它增加了索引被其他 SELECT 语句复用的可能性。

物化结果集

42

物化结果集意味着执行必要的数据库访问来构建结果集。在最好的情况下，这只需要简单地从数据库缓冲池向应用程序返回一条记录。在最坏的情况下，数据库管理系统需要发起大量的磁盘读取。

当一个 SELECT 语句只查询出一条记录时，优化器必须在 SELECT 请求被执行的时候就物化结果记录。而另一方面，当结果集可能有多条记录而需要使用游标时，有两种可供选择的方式（参见图 3.9）：

1. 数据库管理系统在 OPEN CURSOR 时物化整个结果集（或者至少在第一次 FETCH 的时候）。
2. 每次 FETCH 物化一条记录。

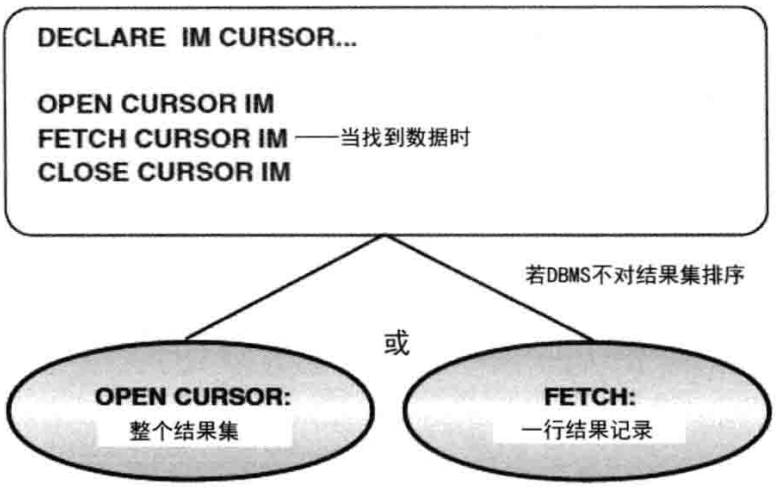


图 3.9　物化结果集

我们接下来将逐个讨论这两种方式。

游标回顾

一次 FETCH 调用按照 DECLARE CURSOR 语句的定义向应用程序返回结果集的一条记录。如果 DELCARE CURSOR 包含了绑定变量，那么在执行时，应用程序会在 OPEN CURSOR 调用之前，将实际的值传入绑定变量。如果应用程序期望使用同一个游标获取多个结果集，那么首先需要执行一次 CLOSE CURSOR 调用，然后向绑定变量传入新的值，再重新执行一次 OPEN CURSOR 调用来打开游标。

在一般的事务隔离级别下（ANSI SQL-92：READ COMMITTED，第 1 级，如 DBZ 中 CURSOR STABILITY），CLOSE CURSOR 调用会释放最后一次 FETCH 操作所持有的锁。如果应用程序没有调用 CLOSE CURSOR 或

43

者使用了更高的事务隔离级别，那么数据库系统会在提交的时刻释放持有的锁。

一些查询工具可以根据 SELECT 语句生成 DECLARE CURSOR、OPEN CURSOR、一个 FETCH 循环和 CLOSE CURSOR 共 4 个步骤。

众所周知，SQL 应用程序的课程中一般都这样描述：DCLARE CURSOR 所定义的 SQL 语句总是在 OPEN CURSOR 调用的时刻传值并组合结果集。然而，事实并非如此。为了避免不必要的工作，数据库管理系统总是尽可能晚地进行结果集物化。假设数据库系统先物化结果集，那么在 FETCH 调用的时候，系统将从临时表中检索记录，而在数据库更新的时候，并不会更新临时表中存放的结果集。

为了说明进行物化结果集的时间点的重要性，我们将使用 SQL 3.6 所示的程序做介绍，这条 SQL 的逻辑是读取某客户的最新的一张发票的记录。

SQL 3.6

```
DECLARE LASTINV CURSOR FOR
SELECT      INO, IDATE, IEUR
FROM        INVOICE
WHERE       CNO = :CNO
ORDER BY    INO DESC
WE WANT 1 ROW PLEASE

OPEN CURSOR LASTINV
   FETCH CURSOR LASTINV
CLOSE CURSOR LASTINV
```

这条查询语句将总是返回一条记录。应用程序只需要执行一次 FETCH 调用即可，因此，应用程序所需要的是获取一条记录的最有效的访问路径。如前面章节所描述，不同 DBMS 的具体语法各有不同，在本书中，我们将使用 SQL 3.6 中这种抽象的描述方法来避免提供代码的多个不同版本。尽管只是请求一条记录，但数据库系统究竟读取了哪些数据？

1. 是读取了 CURSOR 所定义的整个结果集（该客户的所有发票记录）？
2. 还是只读取了满足条件的那一条记录？

从性能的角度，这可能是一个非常重要的问题。我们将逐个讨论这两种可能的方式。

方式 1：一次 FETCH 调用物化一条记录

如果满足下面的条件，数据库系统将倾向于这种方案。

- 没有排序的需求（ORDER BY、GROUP BY 等）或者
- 虽然需要排序，但同时满足下面的两个条件：
  - 存在一个索引满足结果集的 ORDER BY 排序需求，比如 (CNO,IDATA DESC)。
  - 优化器决定以传统的方式使用这个索引，即访问第一条满足条件的索引行并读取相应的表行，然后再访问第二条满足条件的索引行并读取相应的表行，以此类推。如果优化器假定程序将会提取整个结果集，那么它可能会选择错误的访问路径——也许是一次全表扫描。

44

方式 2：提前物化

到目前为止，选择方案 2 的最主要的原因就是需要对结果集进行排序，这可以通过 EXPLAIN 或类似工具的报告看出。在一些特殊情况下，尽管结果集没有排序，但整个结果集还是被物化了，这些场景中有些是 DBMS 相关的。这些场景同样也可以通过 EXPLAIN 的输出看到。

在这个例子中，如果没有以 CNO 列和 IDATE 列开头，或者单独以 DATE 列开头的索引存在，那么 DBMS 就必须对结果集进行排序。如果数据库不能按照索引的反序进行读取，那么索引列 IDATE 必须按倒序进行组织：IDATE DESC。

如果选择了提前物化，一些数据库系统会在 OPEN CURSOR 时物化结果集，另一些则会在第一次 FETCH 调用时物化结果集。只有在查看 SQL trace 时才需要了解这一点，否则这两种方式的区别并不大，因为不太可能发生应用程序在 OPEN CURSOR 后未发起 FETCH 调用的情况。

数据库设计人员必须牢记

对结果集排序意味着，即使只需要提取条记录，也必须物化整个结果集合。

练习

3.1. 为 SQL 3.7 中所示的查询设计尽可能好的索引：

SQL 3.7

```
SELECT      LNAME, FNAME, CNO
FROM        CUST
WHERE       SEX = 'M'
            AND
            HEIGHT > 190
ORDER BY    LNAME, FNAME
```

45

3.2. 为 SQL 3.8 中所示的查询设计尽可能好的索引：

SQL 3.8

```
SELECT      LNAME, FNAME, CNO
FROM        CUST
WHERE       SEX = 'M'
            AND
            (WEIGHT > 90
            OR
            HEIGHT > 190)
ORDER BY    LNAME, FNAME
```

第4章

为 SELETE 语句创建理想的索引

- 影响表及索引扫描性能的主要因素列表
- 随机/顺序读时间及 CPU 成本
- 根据三个最重要的需求为查询语句的索引指定星级
- 三星级索引的设计——最理想的索引设计
- 宽索引
- 为查询语句设计最好索引的算法
- 根据现存索引情况设计最实用的索引，将 CPU 时间、磁盘读时间和耗费时间考虑在内
- 从维护开销来看，对现有索引进行所建议的改变可能产生的结果
- 响应时间、驱动负载和磁盘代价
- 一些建议

简介

当程序中所有的 SQL 都使用到了一个或多个索引时，许多 DBA 就会对此表示满意，认为一切都看起来很正常，EXPLAIN 没有异常。但是，使用一个不合适的索引有可能会导致比全表扫描更差的性能。

Mark Gurry（2）同意这一观点：

> 很多调优人员（尽管没经验）认为，如果一个 SQL 语句使用了索引，那这个 SQL 就是被很好地优化过的，我对此感到很惊讶。你应该总是问自己，“这是不是可用的最好的索引？”或“再添加另外一个索引能否提升响应性能？”，又或者“全表扫描会不会更快地返回结果？”（第 47 页）

在这一章中，我们将会非常详细地考虑这些极其重要的问题。不过，首先需要定义我们接下来的分析所依赖的前提假设。

48 磁盘及 CPU 时间的基础假设

在第 2 章中我们已经对磁盘 I/O 进行了详细的讨论，根据这些信息，我们现在给出一些贯穿全书的基础假设。关于 CPU 时间的更多信息将在第 15 章中讲述。

磁盘及CPU时间的基础假设

	I/O 时间
随机读	**10 ms** （4KB或8KB的页）
顺序读	**40 MB/s**
	顺序扫描的CPU时间
检查一行记录	**5 μs**
FETCH	**100 μs**

不合适的索引

对于下面这个简单 SQL 查询（参见 SQL 4.1 与图 4.1），仅有的两个合理的访问路径：

1. 索引扫描(LNAME,FNAME)
2. 全表扫描

SQL 4.1

```
DECLARE CURSOR41 CURSOR FOR
SELECT      CNO, FNAME
FROM        CUST
WHERE       LNAME = :LNAME
            AND
            CITY = :CITY
ORDER BY    FNAME
```

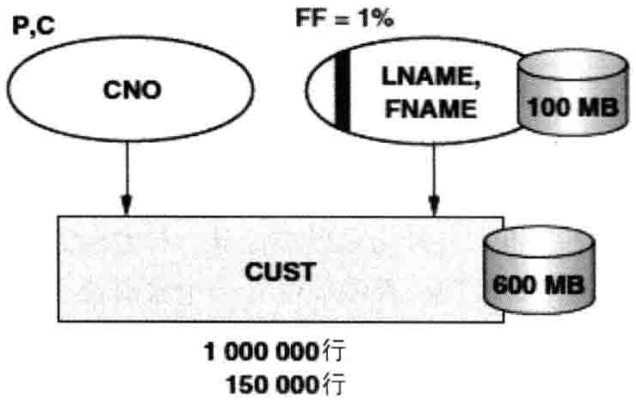


图 4.1　是进行全表扫描还是使用一个不合适的索引？

那么，即使对于最普遍的姓氏（过滤因子 1%），这两种选择是否能够 <49> 提供可接受的响应时间？

对于第一种选择来说，数据库管理系统会根据谓词条件 LNAME = :LNAME 扫描索引片。对于索引片中的每一个索引行，数据库管理系统都必须回到表中校验 CITY 字段的值。由于表中的行是根据 CNO 字段而不是 LNAME 字段来聚簇的，所以这个校验操作需要做一次磁盘的随机读。对于最普遍的姓氏来说，在不考虑 CITY 字段的过滤因子的情况下，获取完整的结果集意味着，需比对 10 000 个索引行和 10 000 个表行。那么，这个过程会耗时多久？

让我们假设索引(LNAME,FNAME)的大小是 1 000 000×100 byte≈100MB，包括数据及分散的空闲空间。另外，再假设顺序读的速度是 40 MB/s。读取一个宽度为 1%的索引片，即 1MB，需花费 10 ms + 1 MB/40 MB/s≈35 ms，这显然没有问题，但是 10 000 次随机读将花费 10 000 × 10 ms = 100 s，这使得这种方式太慢了。

对于第二种选择来说，只有第一个页需要随机读。如果表的大小为 1 000 000 × 600 byte≈600MB，包括分散的空闲空间，那么花费的 I/O 时间将会是 10 ms + 600 MB/40 MB/s≈15 s，仍旧太慢了。

第二种方案的 CPU 时间将会比第一种方案的 CPU 时间长得多，因为数据库管理系统必须比对 1 000 000 行而不是 20 000 行，并且还需要对这些行进行排序。从另一个方面来说，由于是顺序读，CPU 时间可以与 I/O 时间交叠。在这个场景下，全表扫描比在不合适的索引上扫描要快，但是这还不够快，需要有一个更好的索引。

三星索引——查询语句的理想索引

前一小节我们讨论了 CURSOR41 的一个非常不合适的索引，这一小节我们来讨论另一个极端，*三星索引*，即对于一个查询语句可能的*最好索引*。类似图 4.2 中的查询语句，如果使用了三星索引的话，一次查询通常只需要进行一次磁盘随机读及一次窄索引片的扫描。因此，其响应时间通常会比使用一个普通索引的响应时间少几个*数量级*。

即使返回的结果集有 1000 行，CURSOR41（见 SQL 4.2）的响应时间也只有不到一秒。这是怎么做到的呢？图 4.2 展示了索引的最低一层叶子页的情况。索引较高层（非叶子页）的内容将会在以后进行讨论。

50

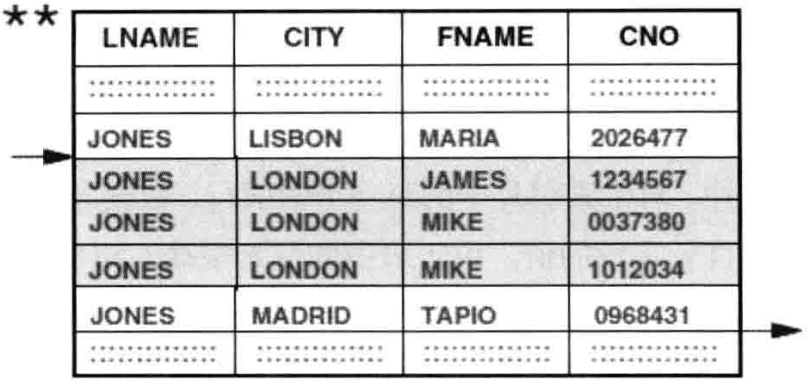


图 4.2　CURSOR41 的三星索引

SQL 4.2

```
DECLARE CURSOR41 CURSOR FOR
SELECT      CNO, FNAME
FROM        CUST
WHERE       LNAME = :LNAME
            AND
            CITY = :CITY
ORDER BY    FNAME
```

如果结果集有 1000 行的话，那么组合谓词 LNAME = :LNAME AND CITY = :CITY 的过滤因子就是 0.1%。被扫描的索引片现在就只包含 1000 行，因为索引片的宽度完全由 LNAME 和 CITY 这两个匹配谓词所决定（这也就是为什么在图 4.2 中索引片那样画的原因）。这个匹配过程在 EXPLAIN 中有所展示：匹配列有两个（MC = 2）。这种情况下，查询将花费 1×10 ms + 1000×0.1 ms≈0.1 s 来扫描这个索引片。在这过程中，表根本就没有被访问过，因为所有所需的列值都已经复制到索引中了。

星级是如何给定的

如果与一个查询相关的索引行是相邻的（如图 4.2 所示），或者至少相距足够靠近的话，那这个索引就可以被标记上第一颗星。这最小化了必须扫描的索引片的宽度。

如果索引行的顺序与查询语句的需求一致，则索引可以被标记上第二颗星，如图 4.2 所示。这排除了排序操作。

51　如果索引行包含查询语句中的所有列——就像图 4.2 中的那样——那么索引就可以被标记上第三颗星。这避免了访问表的操作：仅访问索引就可以了。

对于这三颗星，第三颗通常是最重要的。将一个列排除在索引之外可能会导致许多速度较慢的磁盘随机读。我们把一个至少包含第三颗星的索引称

为对应查询语句的宽索引。

宽索引

宽索引是指一个至少满足第三颗星的索引。该索引包含了SELECT语句所涉及的所有列，因此能够使得查询只需访问索引而无须访问表。

在类似 CURSOR41 这样的简单场景下，一个三星索引的构造是很简单的。

为了满足第一颗星

取出所有等值谓词的列（WHERE COL = ...）。把这些列作为索引最开头的列——以任意顺序都可以。对于 CURSOR41 来说，三星索引可以以 LNAME, CITY 或者以 CITY, LNAME 开头。在这两种情况下，必须扫描的索引片宽度将被缩减至最窄。

为了满足第二颗星

将 ORDER BY 列加入到索引中。不要改变这些列的顺序，但是忽略那些在第一步中已经加入索引的列。例如，如果 CURSOR41 在 ORDER BY 中有重复的列，如 ORDER BY LNAME,FNAME 或者是 ORDER BY FNAME,CITY，只有 FNAME 需要在这步中被加入到索引中去。当 FNAME 是索引的第三列时，结果集中的记录无须排序就已经是以正确的顺序排列的了。第一次读取操作将返回 FNAME 值最小的那一行。

为了满足第三颗星

将查询语句中剩余的列加到索引中去，列在索引中添加的顺序对查询语句的性能没有影响，但是将易变的列放在最后能够降低更新的成本。现在，索引已包含了满足无须回表的访问路径所需的所有列。

最终三星索引将会是：

(LNAME, CITY, FNAME, CNO) 或 (CITY, LNAME, FNAME, CNO)

CURSOR41 在以下三方面是较为挑剔的：

- WHERE 条件不包含范围谓词（BETWEEN、>、>=等）。
- FROM 语句只涉及单表。
- 所有谓词对于优化器来说都足够简单。

52

第一点将在本章中进行简要讨论，并在第 6 章中进行更深入的探讨；第二点将在第 8 章中阐述；第三点将在第 6 章中阐述。

范围谓词和三星索引

下面的 SQL 4.3 需要的信息与之前相同，只是现在顾客是在一个范围内。

SQL 4.3

```
DECLARE CURSOR43 CURSOR FOR
SELECT   CNO, FNAME
FROM     CUST
WHERE    LNAME BETWEEN :LNAME1 AND :LNAME2
         AND
         CITY = :CITY
         ORDER BY FNAME
```

让我们尝试为这个 CURSOR 设计一个三星索引。大部分的推论与 CURSOR41 相同，但是“BETWEEN 谓词”将“=谓词”替代后将会有很大的影响。我们将会以相反的顺序依次考虑三颗星，按理说，这代表了理解的难度。

首先是最简单的星（虽然非常重要），第三颗星。按照先前所述，确保查询语句中的所有列都在索引中就能满足第三颗星。这样不需要访问表，那么同步读也就不会造成问题。

添加 ORDER BY 列能使索引满足第二颗星，但是这个仅在将其放在 BETWEEN 谓词列 LNAME 之前的情况下才成立，如索引(CITY,FNAME,LNAME)。由于 CITY 的值只有 1 个（=谓词），所以使用这个索引可以使结果集以 FNAME 的顺序排列，而不需要额外的排序。但是如果 ORDER BY 字段加在 BETWEEN 谓词列 LNAME 后面，如索引(CITY,LNAME,FNAME)，那么索引行不是按 FNAME 顺序排列的，因而就需要进行排序操作，这个可以从图 4.3 中看出。因此，为了满足第二颗星，FNAME 必须在 BETWEEN
53 谓词列 LNAME 的前面，如索引(FNAME,...)或索引(CITY,FNAME,...)。

CITY	LNAME	FNAME	CNO
.....	.....	.....	.....
LISBON	JONES	MARIA	2026477
LONDON	JOHNS	TONY	7477470
LONDON	JONES	MIKE	0037380
LONDON	JONES	MIKE	1012034
MADRID	JONES	TAPIO	0968431
.....	.....	.....	.....

图 4.3　使用范围谓词按顺序进行扫描的效果图

再考虑第一颗星，如果 CITY 是索引的第一个列，那我们将会有一个相对较窄的索引片需要扫描（MC=1），这取决于 CITY 的过滤因子。但是如果用索引(CITY,LNAME,...)的话，索引片会更窄，这样在有两个匹配列的情况下我们只需访问真正需要的索引行。但是，为了做到这样，并从一个很窄的索引片中获益，其他列（如 FNAME）就不能放在这两列之间。

所以我们的理想索引会有几颗星呢？首先它一定能有第三颗星，但是，正如我们刚才所说，我们只能有第一颗星或者第二颗星，而不能同时拥有两者！换句话说，我们只能二选一：

- 避免排序——拥有第二颗星。

或

- 拥有可能的最窄索引片，不仅将需要处理的索引行数降至最低，而且将后续处理量，特别是表中数据行的同步读，减小到最少——拥有第一颗星。

在这个例子中，BETWEEN 谓词或者任何其他范围谓词的出现，意味着我们不能同时拥有第一颗星和第二颗星。也就是说我们不能拥有一个三星索引。

这就意味着我们需要在第一颗星和第二颗星中做出选择。通常这不是一个困难的选择，因为根据第 6 章的内容，第一颗星一般比第二颗星更重要，虽然并不总是这样。

让我们考虑一下图 4.4 中的索引(LNAME,CITY,...)，LNAME 是范围谓词，如第 3 章中我们看到的，这意味着 LNAME 是参与索引匹配过程的最后一个列。等值谓词 CITY 不会在匹配过程中被使用。这样做将会导致只有一个匹配列——索引片将会比使用索引(CITY, LNAME, ...)更宽。

54

LNAME	CITY	FNAME	CNO
.....	.....	.....	.....
JOHNS	ZURICH	BERNHARD	9696969
JONES	LONDON	JAMES	1234567
JONES	LONDON	MIKE	0037380
JONES	LONDON	MIKE	1012034
JONES	MADRID	TAPIO	0968431
.....	.....	.....	.....

图 4.4　对于 CURSOR43，只满足了第三颗星且 MC = 1

为查询语句设计最佳索引的算法

根据以上的讨论，理想的索引是一个三星索引。然而，正如我们所见，

当存在范围谓词时，这是不可能实现的。我们（也许）不得不牺牲第二颗星来满足一个更窄的索引片（第一颗星），这样，最佳索引就只拥有两颗星。这也就是为什么我们需要仔细区分理想和最佳。在这个例子中理想索引是不可实现的。将这层因素考虑在内，我们可以对所有情况下创建最佳索引（也许不是理想索引）的过程公式化。创建出的索引将拥有三颗星或者两颗星。

首先设计一个索引片尽可能窄（第一颗星）的宽索引（第三颗星）。如果查询使用这个索引时不需要排序（第二颗星），那这个索引就是三星索引。否则这个索引只能是二星索引，牺牲第二颗星。或者采用另一种选择，避免排序，牺牲第一颗星保留第二颗星。这两种二星索引中的一个将会是相应查询语句的最佳索引。

下面内容阐述了为查询语句创建最佳索引的算法。

候选 A

1. 取出对于优化器来说不过分复杂（在第 6 章中讨论）的等值谓词列。将这些列作为索引的前导列——以任意顺序皆可。
2. 将选择性最好的范围谓词作为索引的下一个列，如果存在的话。最好的选择性是指对于最差的输入值有最低的过滤因子。只考虑对于优化器来说不过分复杂的范围谓词（在第 6 章中讨论）即可。
3. 以正确的顺序添加 ORDER BY 列（如果 ORDER BY 列有 DESC 的话，加上 DESC）。忽略在第 1 步或第 2 步中已经添加的列。
4. 以任意顺序将 SELECT 语句中其余的列添加至索引中（但是需要以不易变的列开始）。

55

举例：CURSOR43

候选 A 为(CITY,LNAME,FNAME,CNO)。

由于 FNAME 在范围谓词列 LNAME 的后面，候选 A 引起了 CURSOR43 的一次排序操作。

候选 B

如果候选 A 引起了所给查询语句的一次排序操作，那么还可以设计候选 B。根据定义，对于候选 B 来说第二颗星比第一颗星更重要。

1. 取出对于优化器来说不过分复杂的等值谓词列。将这些列作为索引的前导列——以任意顺序皆可。

2. 以正确的顺序添加 ORDER BY 列（如果 ORDER BY 列有 DESC 的话，加上 DESC）。忽略在第 1 步中已经添加的列。
3. 以任意顺序将 SELECT 语句中其余的列添加至索引中（但是需要以不易变的列开始）。

举例：CURSOR43

候选 B 为(CITY,FNAME,LNAME,CNO)。

现在我们有两个最佳索引的候选对象，一个有第一颗星，一个有第二颗星。为了判断哪一个是最佳索引，我们按照本章最开始所讲的那样分析使用每个索引的性能。这将会比较耗时，第 5 章提供了一个更简单的方法（QUBE）来估算哪个候选对象将会提供对于这个查询语句更快的访问路径。

有一点需要注意的是，到目前为止，我们所做的只是设计理想索引或是最佳索引。但是这是否是实际可行的，我们在这个阶段还不好说。

现今排序速度很快——为什么我们还需要候选 B

近几年来，排序速度已经提升很多。现在大多数的排序过程都在内存中进行，用当下最快的处理器每排序一行花费的 CPU 时间大约在 10 μs 左右。因此，排序 50 000 行记录所耗费的时间只有 0.5 s，这对于一次事务操作来说也许是可接受的，但这对于 CPU 时间来说已经是一个比较大的开销了。

由于在现在的硬件条件下排序速度很快，所以如果一个程序取出结果集的所有行，那么候选 A 可能和候选 B 一样快，甚至比候选 B 更快。对于程序员来说，这是最方便的解决方案。许多环境都提供灵活的命令来浏览结果集。

然而，如果一个程序只需获取能够填充满一个屏幕的数据量，如 SQL 4.4 中的 CURSOR44，那么候选 B 可能会比候选 A 快很多。正如第 3 章中讨论的，如果访问路径中没有排序的话，数据库管理系统只要一次一次地读取数据行就能对结果集进行物化。这也是为什么有些时候避免排序非常重要（通过采用候选 B）。如果结果集很大的话，为了产生第一屏的数据，二星索引候选 A（需要进行排序）可能会花费非常长的时间。我们需要时刻记着，终端用户的一次错误输入可能会使得结果集变得非常大。

56

如果访问路径中没有排序的话，使用 CURSOR44 的程序将会非常快（假设 LNAME 和 CITY 两列是索引中的前两列——不管顺序如何），即使结果集包含数以百万级的数据行。每个事务永远都不会使数据库管理系统物化大于 20 行的数据。我们后面会讨论如何实现高效查找接下来的 20 行结果数据。

SQL 4.4

```
DECLARE CURSOR44 CURSOR FOR
SELECT      CNO, FNAME
FROM        CUST
WHERE       LNAME = :LNAME
            AND
            CITY = :CITY
ORDER BY    FNAME
WE WANT 20 ROWS PLEASE

OPEN CURSOR CURSOR4
FETCH CURSOR CURSOR4 ----- 最多20次
CLOSE CURSOR CURSOR4
```

需要为所有查询语句都设计理想索引吗

为每一个查询设计最佳索引的过程是很简单的(除非已经存在相同的索引了)。这个设计过程，也就是上文所说的两种候选方案算法，是机械式的，只要给出下面这些内容就可以自动完成整个过程：

1. 查询语句
2. 数据库统计信息（行数、页数、列值分布等）
3. 对于每一个简单谓词或组合谓词最差情况下的过滤因子
4. 已经存在的索引

在这种最简单的过程中，当前存在的索引信息只是用来避免生成重复的索引。在一个使用了这种索引设计方法的 AS/400 系统中，数据库管理系统
57 在一个表上创建了 60 个索引。其中的许多索引实际上都是可以删除以提升插入速度的，且删除后不会使得查询速度明显下降。

在为查询语句设计了一个最佳索引后，去看一下已经存在的索引是很有必要的。有可能某一个已经存在的索引几乎和理想索引差不多好用，特别是打算在这个已有索引的最后添加一些列的情况下。

当分析一个已经存在的索引对一个新的查询语句有多大用处时，需要记住，多余的索引分为三种：完全多余的索引，近乎多余的索引，以及可能多余的索引。

完全多余的索引

在许多年前，AS/400 系统用户发现，如果一个查询包含 WHERE A = :A AND B = :B 而另一个查询包含 WHERE B = :B AND A = :A 的话，数据库管理系统就会创建两个索引：(A,B)和(B,A)。如果没有查询包含 A 列或者 B 列

上的范围谓词的话，那么这两个索引中的一个就是完全多余的。我们不需要两本电话薄，一本根据 LNAME,FNAME 排序而另一本根据 FNAME,LNAME 排序（如果姓氏和名字都已知的话）。AS/400 系统的用户很快就学会了规范化他们所写的 WHERE 语句，但当开始使用程序生成器后这个问题又出现了。

近乎多余的索引

假设索引(LNAME,CITY,FNAME,CNO)已经存在。为一个新的查询语句设计的理想索引包含了以这个索引的 4 列为开头的 14 个列。那么，在创建了新的索引之后，原来的索引是不是应该删除呢？一些 DBA 可能会犹豫要不要这么做，因为这个已经存在的索引是唯一索引。但是，这个索引并不是主键索引也不是候选键索引，只是恰好这个索引包含了主键列 CNO。把其他的列加到这个索引上不会有完整性问题。如果数据库管理系统支持非键值索引列，或者有约束来保证唯一性的话，数据列甚至可以加到主键索引或者任何键值必须唯一的索引上。这样一来，问题就成了一个纯粹的性能问题：一个原来使用 4 列索引的查询现在使用新的 14 列索引，速度是否会明显变慢？

假设索引行的大小从原先的 50 字节增长为 200 字节，那么扫描 10 000 行索引片并从中取出 1000 个索引项会多花费多少时间？CPU 时间增长不多，但是 I/O 时间是和需要访问的页数成比例的。

CPU 时间 = 1000 × 0.1 ms + 10 000 × 0.005 ms = 150 ms
（两种情况下都是 1000 次 FETCH 调用和 10 000 个索引行）
4KB 大小的叶子页的数量（4 列）　1.5 × 10 000 × 50/4000≈200
（1.5 为空闲空间系数）
4KB 大小的叶子页的数量（14 列）　1.5 × 10 000 × 200/4000≈800　58
顺序读时间（4 列）= 200 × 0.1 ms = 20 ms
顺序读时间（14 列）= 800 × 0.1 ms = 80 ms

由于顺序读的处理过程使得响应时间还是受 CPU 时间的限制，所以查询语句使用这两个索引的响应时间没有明显的不同。在新的 14 列索引创建之后，现存的 4 列索引就变成多余的了。第 15 章会对 CPU 时间的估算进行详细讨论。

可能多余的索引

一个普遍的场景是这样的：一个新的查询语句的理想索引是

(A,B,C,D,E,F)，而表上已经存在的索引是(A,B,F,C)。那么如果把已经存在的索引替换成(A,B,F,C,D,E)的话，新的索引是不是就多余了？换句话说，如果把 D 和 E 两列加到现有的索引上使得访问路径仅限于索引，这样对于新的查询语句是否就已经足够了？

理想索引可能在两方面比索引(A,B,F,C,D,E)要好：

1. 可能使得查询有更多的匹配列。
2. 可能可以避免排序。

这两个优势都受需要在索引片上扫描的行数的影响。两个索引的差异可以如本章所述转换为毫秒值进行比较，或者更简单一些，通过第 5 章中讨论的快速上限估算法（QUBE）进行估算。估算结果往往会显示，新的索引是不需要的，在现有索引后面加上新的列对于新的 SELECT 语句就已经足够了。

新增一个索引的代价

如果一个表上有 100 个不同的查询，且为每一个查询语句都设计了最佳索引的话，那么即使没有重复的索引，该表上最终也可能有非常多的索引。这样一来表的插入、更新和删除操作就会变得很慢。

响应时间

当数据库管理系统向表中添加一行时，它必须在每一个索引上都添加相应的行。在当前的硬件条件下，在一个索引上添加一行，插入操作所花费的时间就增加 10 ms，因为必须从磁盘上读取一个叶子页。当一个事务向一张有 10 个索引的表里插入 1 行数据时，索引的维护就会使响应时间增加 10 × 10 ms = 100 ms，这可能是可接受的。然而，如果一个事务向一张有 10 个索引的表里插入 20 行数据的话，索引的维护就会需要 181 次随机读，即耗费 1.8 s。这个估算基于的前提假设是，新的索引行会把表上其
59 中一个索引（一直增大的键值上的索引）添加到同一个叶子页上，而会把其余 9 个索引添加到 20 个不同的叶子页上。从响应时间的角度来看，在一个有 10 个索引的大表上进行大的事务操作（每个事务中有许多插入或删除操作）可能是无法忍受的。另外，从磁盘负载的角度来看，要在一个大表上进行每秒多于 10 行的插入操作可能不容许表上有 10 个索引。让我们来深入分析下第二个问题。

磁盘负载

被修改过的叶子页是迟早会被写到磁盘上去的。由于数据库的写是异步的，所以这些写不会影响到事务的响应时间。但是，这些写会增加磁盘负载。RAID 5 会放大这种影响，因为每一次页的随机更新都会引来两个磁盘的访问。每一次访问都会耗费 12ms，因为整个 Raid 条带都需要被读取和写回：一次寻道（4ms）和两次旋转（2 × 4 ms）。因此，向磁盘写一个被修改的页所带来的整体磁盘繁忙度的增加为 24ms。RAID 10（条带化和镜像）相应的增量为 2 × 6 ms = 12 ms。

如果一张表的插入频率较高的话，磁盘负载可能会变成主要的问题，限制了表上索引的数量。由于删除操作和插入操作所带来的磁盘负载是相同的，所以大量的删除任务是另外一个重要的考虑事项。更新操作只会影响列值被修改了的索引。

我们假设一个 RAID 5 磁盘服务器有 128 块盘：112 块活动盘，16 块空闲盘。数据库（表和索引或者它们的分区）被条带化到这些活动盘上。读缓存为 64GB，写缓存为 2GB。

在 TRANS 表中（如图 4.5 所示），新插入的行保存在表及其聚簇索引的末尾。在页被写到磁盘上之前，许多行会被写到这个页上，所以这些操作不会造成大量的磁盘读和写。会带来问题的是在 4 个索引上的随机插入操作，每一个新的索引行可能都会导致一次磁盘读和磁盘写，即每秒一共 80 次随机读和 80 次随机写（4 × 20）。 60

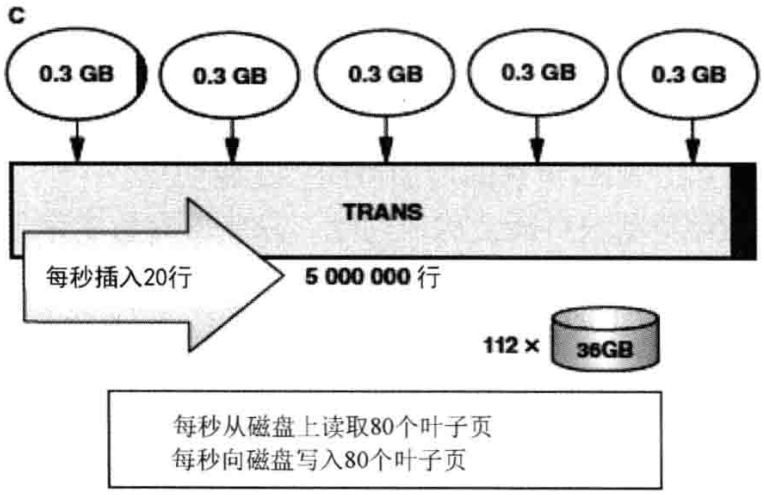


图 4.5　对磁盘服务器有压力

我们先忽略读缓存和写缓存。在最差的情况下，4 个索引造成的磁盘繁忙度为 80 × (6 ms + 24 ms) =2400 ms，相应的磁盘负载为 2400 ms/s = 2.4 = 240%，如果这 4 个索引是在 112 块盘上做条带的话，它们对于平均磁盘负

载的贡献是 240%/112≈2%。需要将这个值和磁盘的过载值相比较。理想情况下，平均磁盘负载不应超过 25%。根据排队理论，平均磁盘排队时间将会是 3ms。由此，增加 2%的磁盘繁忙度可能是可以忍受且不明显的。

使用读缓存和写缓存能在多大程度上减轻磁盘负载?

64GB 的读缓存和 4 个索引的大小（1.2GB）相比较似乎很大，但是如果访问模式是完全随机的话，当插入频率为 20 行每秒时，对于每一个索引上 75 000 个叶子页中的任何一个叶子页，对同一个叶子页的相邻两次访问的时间间隔是基本平均的，间隔时间为 75 000 × 50 ms = 3750 s≈1 h，如果读缓存的平均保留时间是 30 分钟，那么就不会有很多磁盘读命中读缓存。如果写缓存的保留时间比读缓存的保留时间短，比如 10 分钟，那读缓存的作用将会更小：只有当一个叶子页在 10 分钟内被更新 1 次以上，才能通过写缓存节省一次磁盘写。因此，由这 4 个索引导致的平均磁盘繁忙度几乎依然是 2%。如果访问模式不是随机的，缓存将会节省更多磁盘负载上的开销。每小时更新次数多于 10 次的叶子页能在读缓存和写缓存中长时间保留。

RAID 10，镜像并条带但是没有校验位，能将每个被修改页所带来的磁盘繁忙度从 24 ms 降至 12 ms，但是这样需要增加磁盘数量。有 256 块盘的 RAID 5 基本上也能带来相同的效果。

从这个例子中可以引申出一个经验法则，其中指示符 L 表示一个表上的索引对 RAID 5 下磁盘平均负载的贡献：

$$L = N \times I/D$$

其中 N= 随机插入涉及的索引数量

I= 插入频率（表每秒插入的行数）

D= 磁盘数量（空闲盘除外）

如果 $L < 1$，那么磁盘负载的增加不构成问题；负载增量很可能低于 2%。

如果 L 在 1 和 10 之间，那么负载的增加可能会比较明显。

如果 L>10，那么磁盘负载很可能会构成问题，除非缓存命中率很高。

在上面的例子中，$L = 4 \times 20/112 = 0.7$。

61 从磁盘负载的角度来看，一个插入频率低的表，比如一个典型的 CUST 表，能够容忍在表上建许多索引，索引数量的上限取决于插入事务对响应时间的要求。

如果磁盘负载是一个问题，较明显的解决办法是尝试合并索引。一个有 10 个列的索引比两个各有 6 个列的索引所引起的磁盘负载要小。

磁盘空间

如果一个表中有一千万行以上的数据的话，索引磁盘空间的成本可能会

成为一个问题。外购硬件的价格主要取决于两个因素：花费的 CPU 时间和分配的磁盘空间。在本书写作时，一个高性能、能容错的磁盘服务器每月磁盘空间的费用可能在 50 美元/GB。对于那些有自己的硬件的系统，每 GB 磁盘空间的费率有自己的内部价格。数据库设计人员需要关心当前磁盘空间的成本。举例来说，在一个索引上添加一个长度为 100 字节的列的需求，常常可能会因为磁盘空间浪费而被否决。

下面是数据库专家 Jim Gray（微软研究院）在最近一次采访中提到的（5）：

> 实际上，我很仔细地跟进磁盘价格的情况。从真空吸尘器的想法失败（1996 年）以来，磁盘价格已经下降了 100 倍。磁臂的价格也下降了 10 倍左右。这些比例一定会改变一些事情的……（第 5 页）

举个例子，有人建议在表上创建一个每行包含 400 字节用户数据的索引，我们是否需要考虑磁盘空间？

这个索引（去除 RAID 带来的额外开销）需要 1.5 × 10 000 000 × 400 byte ≈6 GB 左右的磁盘空间。成本可能是每月 300 美元。不过如果将这些列添加到现有索引上并不能提供可接受的响应时间的话，磁盘空间可能不是问题的关键。

随着索引变大，缓冲池或磁盘缓存也应该随之增大，否则非叶子页的 I/O 量会增加。在当前硬件价格情况下，对于我们例子中的非叶子页，内存开销会在每月 300 美元左右。

宽索引的额外开销系数主要取决于分布的空闲空间的量。1.5 倍对于每个叶子页保留 25%的空闲空间已经足够大了。对于一个 8KB 的叶子页来说，这意味着在分裂前，还可以向叶子页上添加 5 个 400 字节的索引行。在第 11 章中会根据索引行的长度和插入模式，对索引空闲空间的规范和重建频率做推荐。

指向表数据行的指针长度取决于数据库管理系统。直接指针长度较短，通常小于 10 个字节。符号指针可能较长，这些通常在索引中用来指向持有表数据行的索引。

一些建议

62

即使在目前磁盘空间成本较低的情况下，机械性地为每一个查询设计最佳索引也是不明智的，因为索引的维护可能会使得一些程序速度太慢或者使磁盘负载超负荷（这会影响所有程序）。最佳索引（根据两个候选索引的方法设计或者使用索引工具设计）是一个好的开端，但是，在决定为一个新的

查询创建理想索引前需要先考虑一下三种多余的索引。

即使有可能为每一个新的查询都设计最佳索引，但在实际中更常见的情况是，只对那些由于不合适的索引而导致速度太慢（通过估算或通过测量）的查询语句进行索引设计。但是如何发现这些语句呢？第 5 章推荐了两个在编写或生成 SQL 语句的早期发现慢访问路径的简单方法。第 7 章讨论了一种异常监控方法，通过它能够在生产系统中发现严重的访问路径问题和其他性能问题。

练习

4.1. 为 SQL 4.5 中的查询语句设计候选索引 A 和候选索引 B。

SQL 4.5

```
SELECT      A, B, D, E
FROM        ORDERITEM
WHERE       B BETWEEN :B1 AND :B2    (FF = 1...10%)
            AND
            C = 1                    (FF = 2%)
            AND
            E > 0                    (FF = 50%)
            AND
            F = :F                   (FF = 0.1...1%)
ORDER BY    A, B, C, F
WE WANT 20 ROWS PLEASE
```

4.2. 对每一个候选索引，计算在最差情况下一个事务必须访问的索引行数。ORDERITEM 表有 100 000 000 行。

第 5 章

前瞻性的索引设计

- 介绍两个快速、易用的技术：基本问题法（BQ）和快速上限估算法（QUBE），这两个技术将在本书中被广泛运用
- 第 4 章中所讨论的原则在系统 SQL 设计过程中的应用
- 使用这些技术来确定候选索引设计方案的优缺点，并确定必要的改进以提供足够的 SQL 性能
- 评估考虑到的所有改进点的合适性
- 关于 QUBE 应在何时使用的讨论

发现不合适的索引

一旦一个应用的明细方案确定下来，我们就应该确认当前的索引对新的应用来说是否合适。为了完成这个工作，我们将考虑两种简单、快速并且可行的方法：

1. 基本问题法（Basic Question，BQ）（8）
2. 快速上限估算法（Quick Upper-Bound Estimate，QUBE）

基本问题法（BQ）

即使是最忙的程序员也有时间去做这个评估。对每个 SELECT 语句，以下问题的答案都必须按照下述步骤来考虑。

是否有一个已存在的或者计划中的索引包含了 WHERE 子句所引用的所有列（一个半宽索引）？

- 如果答案是否，那么我们应当首先考虑将缺少的谓词列加到一个现有的索引上去。这将产生一个半宽索引，尽管索引的等值匹配过程并不令人满意（一星），但是索引过滤可以确保回表访问只发

64

生在所有查询条件都满足的时候。

- 如果这还没有达到足够的性能，那么下一个选择就是将所有涉及的列都加到索引上，以使访问路径只需访问索引。这将产生一个避免所有表访问的宽索引。
- 如果SELECT仍然很慢，就应当使用第4章所介绍的两个候选索引算法来设计一个新的索引。根据定义，这将是所能实现的最佳索引。

如何确定第一个方案（半宽索引）或第二个（宽索引）方案能否让SELECT在最差输入的情况下仍然运行得足够快？

如果可以访问生产库或者类似生产库的测试库，我们可以每次创建一个索引，用最差输入来测试响应时间。为了确保测得的响应时间接近于在正常生产库上运行的性能表现，我们必须把缓冲池考虑进来，并观察每个事务的缓冲池命中率。测试的第一个事务很可能在缓冲池中没有发现相应的缓存页，所以最差输入下的磁盘读的指标会比正常环境的要高。此外，第一个访问索引的事务必须要等待文件被打开。而另一方面，如果变量的输入值保持不变，那么第二个访问该索引的事务将很可能会获得100%的缓冲池命中率（没有磁盘读）。为了获得具有代表性的响应时间，每个索引方案都应在进行过预热事务之后再开始测试，可以通过传入一些典型的输入值（但不是最差情况的值）来打开文件，并将大部分非叶子索引页加载到数据库的缓冲池中。这样，使用最差输入值的事务就会有一个比较有代表性的响应时间了，至少我们已经把CPU和磁盘读的服务时间考虑进来了。

实际上，使用第二种方法，QUBE，来评估索引方案将不会那么乏味，我们很快会详细阐述这一方法。

如果模拟测试和QUBE方法都没有实施，那么我们就应当选择使用宽索引的方案，并在应用切换到生产环境后立即启用异常报告来发现那些连宽索引都无法满足性能要求的场景。如果必要的话，我们需要为那些运行缓慢的查询设计我们所能达到的最佳（新的）索引。

注意

对于BQ的一个肯定的回答并不能保证足够的性能。记住，BQ的目的只是确保我们至少可以通过索引过滤来最小化对表的访问——除此以外没有其他的作用。

举个例子，假设有一个SELECT，谓词是WHERE B = :B AND C = :C，

唯一有用的索引是(A,B,C)。这个索引的确包含了该 SELECT 的所有的谓词列，因此用 BQ 方法检查这个 SELECT 并不会产生告警。然而，这个查询的访问路径将会是全索引扫描——如果该表有超过 100 000 条记录，那么查询 65 将运行得非常慢。索引过滤本身并不意味着这个索引有三颗星中的任何一颗星。

然而，根据我们的经验，许多发布后才发现的索引问题都是可以通过 BQ 方法来提早发现的。

将 BQ 方法应用到单表查询上是很容易的。但对于连接查询，我们就必须先在脑海中将其拆分成多个单表游标，然后再应用 BQ 方法。这是一个相当复杂的过程，我们会在第 8 章中详细讨论。

快速上限估算法（QUBE）

在最初的评估阶段（当前索引是否让 SELECT 足够快？），QUBE 比 BQ 更耗时，但它能揭示所有与索引或者表设计相关的性能问题——假设对每个谓词所用的最差过滤因子都非常接近于实际的最差情况过滤因子值。根据定义，QUBE 方法是悲观的（上限），它有时会有告警误报，但它不会像 BQ 那样漏掉发现某些问题。

QUBE 的目的是在一个非常早的阶段（程序在设计、编写或者完成的时候）将潜在的慢访问路径问题暴露出来。为了能在实际项目中使用，任何语句级别的预测公式都必须足够简单，这样才能将评估过程的额外开销保持在一个可以接受的程度。几年前，一家芬兰公司使用了一个老版本的 QUBE 来为一个新的应用评估所有事务和批处理程序。根据记录，评估过程只给整体实现过程增加了 5%的额外开销，但却最终帮助其发现了许多索引的优化点和程序设计的改进点，并且最终被实施了。与没有使用 QUBE 的类似项目相比，该应用发布后的优化工作减少了 90%。本书中所讨论的 QUBE 版本比在这家公司中所使用的版本耗时更少。

这个快速估算法的输出结果是本地响应时间（LRT），即在数据库服务器中的耗时。在单层环境（指发出 SQL 调用的应用与数据库部署在同一台机器上）中，一个传统事务的 LRT 是指用户和数据库服务器之间一次交互的响应时间，不包括所有的网络延迟。在多层环境（客户端/服务器）中，各层之间的通信时间也被排除在外。批量任务的 LRT 是指执行任务的耗时。任何任务队列中的排队时间也被排除在外。

图 5.1 展示了 LRT 的组成部分，以及 QUBE 算法会计算进来的部分。

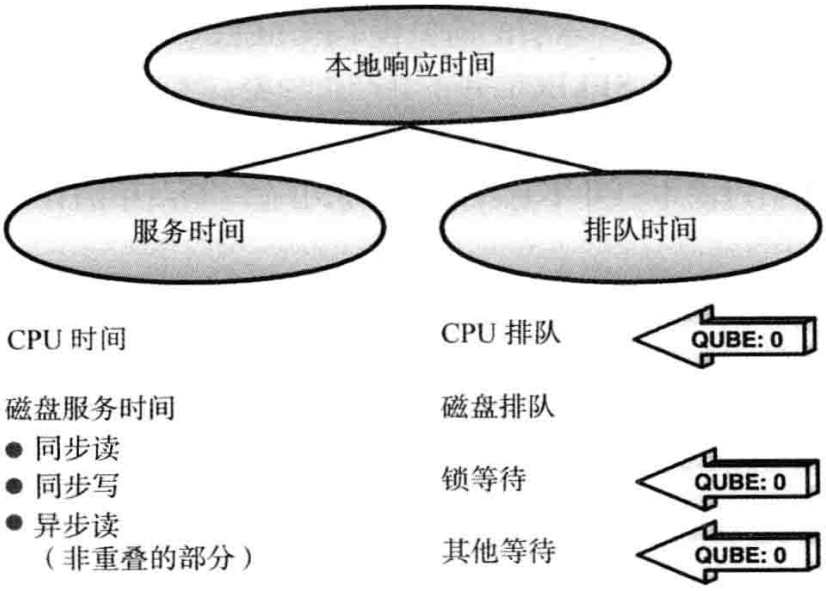


图 5.1　本地响应时间等于服务时间和排队时间的总和

66 服务时间

在简单的场景下（I/O 时间和 CPU 时间不重叠），服务时间等于 CPU 时间加上排除了磁盘驱动排队的随机读时间。如果没有资源竞争，则本地响应时间等于服务时间。

排队时间

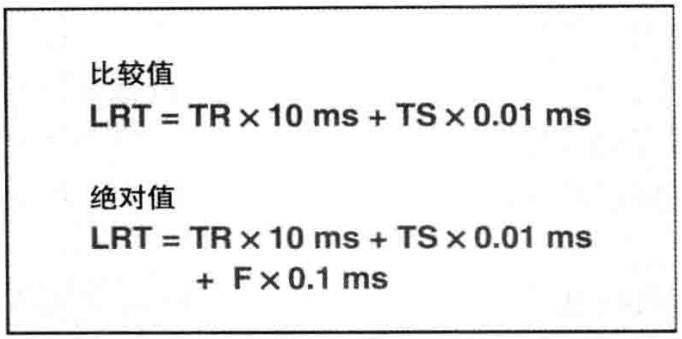
在一个常规的多用户环境中，程序的并发会导致对所需资源的各种竞争，因此这些并发的程序不得不排队来获取这些资源。以下资源在 LRT 的范畴内：

- CPU 排队时间（所有处理器都忙着处理更高优先级的任务）。
- 磁盘驱动器排队（包含请求页的驱动器处于繁忙状态）。
- 锁等待（请求的表或行被锁定在一个不兼容的级别）。
- 多道编程的级别、线程数或其他方面已经达到了上限（这些都是为了防止资源过载而设计的系统值，对于理解本书而言没有必要了解这些值）。

QUBE 会忽略除磁盘驱动排队以外的其他所有类型的排队，以提供一个简单的评估过程，用于评估那些对性能影响特别大的方面。我们稍后将详细讨论这些方面，以及 QUBE 方法背后的简单假设。

在排除上述因素之后，我们得到的是一个非常简单的估算过程，仅需两个输入变量，TR 和 TS，必要时还有第三个输入变量，F。这些就可以将 SQL

的处理及 I/O 成本考虑进来了，并且它们是影响索引设计的主要因素。图 5.2 展示了这些变量及 QUBE 中使用到的值。如果是使用 QUBE 来比较多个可选访问路径之间的性能差异（图 5.2 中的比较值），由于 FETCH 调用的次数是相同的，所以可以忽略 FETCH 调用次数带来的影响。如果是使用 QUBE 来确定响应时间（图 5.2 中的绝对值），那就需要包含 FETCH 调用的次数。 67
为了完整起见，本书中的示例通常会包含 FETCH 的成本。



LRT = 本地响应时间
TR = 随机访问的数量
TS = 顺序访问的数量
F = 有效FETCH的数量

图 5.2　快速上限估算法（QUBE）

基本概念：访问

根据定义，DBMS 读取一个索引行或一个表行的成本称为一次访问：索引访问或表访问。如果 DBMS 扫描索引或表的一个片段（被读取的行在物理上是彼此相邻的），那么第一行的读取即为一次随机访问。对于后续行的读取，每行都是一次顺序访问。在当前的硬件条件下，顺序访问的成本比随机访问要低得多。一次索引访问的成本与一次表访问的成本基本上是相同的。

现在，我们将更深入地讨论索引及表的顺序访问。

读取一组连续的索引行

物理上彼此相邻是什么意思？索引中的所有行都通过指针链接在一起，链接的先后顺序由索引的键值严格定义。当几个索引行的键值相同时，就根据索引行存储的指针值进行链接。在传统的索引设计（从某个角度看，是理想化的）中，链表从 LP1（叶子页 1）开始，随后链接 LP2，以此类推。这样（假设每个磁道可以放 12 个叶子页——当前的硬件通常可以容纳更多），叶子页就组成了一个连续的文件，LP1 至 LP12 存储在磁盘柱面的第一个磁道，LP13 至 LP24 存储在下一个磁道，如此继续。当第一个柱面存满后，下一组 LP 就会被存储在下一个柱面的首个磁道上。换句话说，就是叶子页之

间没有其他的页。

现在，读取一组连续的索引行（即一个索引片，或者包含了单个键值或者一个范围的键值所对应的索引行）就非常快了。一次磁盘旋转会将多个叶子页读取进内存中，而且只有在磁盘指针移到下一个柱面时才需要进行一次短暂的寻址。

68 不过这个完美的顺序还是会被打破的，至少有以下三个影响因素：

1. 如果一个叶子页没有足够的空间存储新插入的索引行，那么叶子页就必须被分裂。之后链表仍会按照正确的顺序链接索引行，但是这与底层的物理存储顺序就不再一致了，一些“按道理”应该是顺序的访问就变成随机访问了。不过索引的重组可以再次恢复最理想的顺序。
2. 意想不到的数据增长可能会填满原本连续的空间（区或者类似的概念）。操作系统于是就会寻找另外一个连续的空间，并将它链接到原来空间的后面。这时候从第一个区跨到第二个区访问就会产生一次随机访问，不过这种情况的影响不大。
3. RAID 5 条带会将前几个叶子页存储在一个驱动器上，将后面的叶子页存储在另外的驱动器上。这就会产生额外的随机读，但实际上条带的积极作用要大过随机读带来的性能恶化。一个智能的磁盘服务器可以将后续的叶子页并行地从多个驱动器上读取至磁盘缓存中，从而大大降低了单个叶子页的 I/O 时间。此外，在 RAID 5 条带策略下，一个被频繁访问的索引不太可能导致某一个磁盘负载过高，因为 I/O 请求会被均匀地分布到 RAID 5 阵列内的多个磁盘驱动器上。

忽略上述情况，我们仍然假设，如果两个索引行在链表上彼此相邻（或者在非唯一索引中，相同键值的行指针彼此相邻），那么我们就认为这两行在物理上也相邻。这就意味着 QUBE 认为所有的索引都有最理想的顺序，我们将在第 11 章中再讨论这一点。

读取一组连续的表行

读取一组连续的表行有如下两种情况。

1. *全表扫描*　从 TP1（表页 1）开始，读取该页上所有的记录，然后再访问 TP2，以此类推。按照记录在表页中存储的顺序进行读取，没有其他特殊的顺序。
2. *聚簇索引扫描*　读取索引片上的第一个索引行，然后获取相应的表行，再访问第二个索引行，以此类推。如果索引行与对应的表

行记录顺序完全一致（聚簇率为 100%），那么除了第一次之外的所有表访问就都是顺序访问。表记录的链接方式跟索引不一样。单个表页中记录的顺序无关紧要，只要访问的下一个表记录在同一个表页或者相邻的下一个表页内就可以了。

同索引一样，存储表的传统方式也是将所有表页保留在一个连续的空间内。引起顺序杂乱或碎片化的因素也与索引中的相似，但有两个地方不同：

1. 如果往表中插入的记录在聚簇索引所定义的主页中装不下，则通常不会移动现有的行，而是会将新插入的记录存储到离主页尽可能近的表页中。对第二个页的额外的随机 I/O 会使聚簇索引扫描变得更慢，但是如果这条记录离“主页”很近，这些额外的开销就可以被避免，因为顺序预读功能会一次性将多个表页装载到数据库缓存中。即使顺序预读功能没有被使用，也只有当该页在数据库缓存中被覆盖的情况下才会发生额外的随机 I/O。 69
2. 一条记录被更新之后，可能会因为表行过长而导致其无法再存储于当前的表页中。这时 DBMS 就必须将该行记录迁移到另外一个表页中，同时在原有的表页中存储指向新表页的指针。当该行被访问的时候，这将引入额外的随机访问。

表可以通过重组来还原行记录的顺序，从而减少不必要的随机访问。因此，如果行记录存储在同一个页或者相邻的页当中，QUBE 就认为它们在物理上彼此相邻。换句话说，QUBE 假设所有的表、索引都是以最理想的顺序组织的。

之前我们做过一些最差场景的假设，在这种假设下，QUBE 足够简单，能够快速且方便地使用。同样的原因，我们也有必要做一些乐观的假设。尽管存在上述问题，根据 QUBE 的定义，扫描索引或表的一个片段只需进行一次随机访问。数据库专家们需要对重组的必要性进行监控，以保证我们所做的这些乐观假设都是合理的。

计算访问次数

既然访问对于 QUBE 而言如此重要，我们现在就解释一下如何确定索引及表的访问次数，包括随机访问和顺序访问。

随机访问

我们将首先思考一下磁盘读与访问的区别。一次磁盘读所访问的对象是一个页，而一次访问的访问对象则是一行。一次随机磁盘读会将一整页（通

常包含很多行）读取至数据库的缓冲池中，但是根据定义，前后两次连续的随机读不太可能会访问到同一个页。因此，QUBE 中单次随机访问所消耗的时间与一次磁盘随机读的平均耗时是一样的，都是 10 ms。虽然随机读是同步的，但是由于现在的处理器速度非常快，所以在估算单次随机访问的开销时可以忽略 CPU 时间，这一 CPU 时间通常小于 0.1 ms。

70

顺序访问

一次顺序访问是指读取物理上连续的下一行，这一行要么存储在同一页中，要么在下一页中。由于顺序读的 CPU 时间与 I/O 时间是重叠的（DBMS 和磁盘控制器通常都会预读一些页），因此顺序访问的消耗时间就是两者中较大的那个。在 QUBE 中，一次顺序读所消耗的时间是 0.01 ms。

当计算访问次数时，为了简单起见，我们会遵循以下规则：

1. 忽略索引的非叶子节点页。我们假定它们都在数据库的缓冲池中，或者至少在磁盘服务器的读缓存中。
2. 假设 DBMS 能够直接定位到索引片的第一行（忽略为了定位索引行的位置而使用二分查找或者其他技术所耗费的时间）。
3. 忽略跳跃使顺序读所节省的任何时间。这可能是一个非常悲观的假设，我们将在后面进行讨论。
4. 假设所有的索引和表都处于理想的组织顺序。如上所述，这可能是个很乐观的假设，除非索引和表都被很好地监控着。

当计算索引访问次数的时候，可以将索引看成一个微表，它的行数与其指向的表包含的行数相同，且按照索引键值排列。

当计算表访问次数时，我们假设表行在表页中是按理想顺序排序的，这个顺序依赖于表的组织方式。这样我们就可以假设一次全表扫描（N 行数据）将需要一次随机访问和 $N-1$ 次顺序访问。

图 5.2 中所展示的顺序访问的时间（0.01 ms）是指数据库在判断是否接受一行时所做的必要处理的时间——不管是在索引中还是表中，所有的行都必须进行这一步检测。被拒绝的行将不再需要进一步处理。

FETCH 处理

被接受的行的数量可以通过 FETCH 调用的次数来确定（除非多行 FETCH 可用），这些行将经历更多的处理程序。请记住，TS 不包含这个额外的处理过程。现在，我们需要将第三个输入变量，LRT 组成中的 F 考虑进来了。如图 5.2 所示，F 的成本比 TS 大一个数量级，而另一方面，它比

71

TR 的成本要小很多。

如果 QUBE 被用来比较可选的访问路径，比如比较全表扫描与使用特定索引，那么 F 参数是无关紧要的，因为它的值在两种情况下相等，我们只需要考虑 TR 和 TS。但如果使用 QUBE 来确定 LRT，F 参数就可能会比较重要，这取决于 FETCH 调用的次数。

在处理被接受了的行时，可能会涉及排序操作，排序的整体开销通常与 FETCH 的行数成正比。大多数情况下，相对于在所接受的行上进行的其他处理过程，排序的开销非常小，所以它将被“隐藏”在 F 的开销内。然而有些场景却不是这样的，我们将在第 8 章中讨论。除了这些例外场景外，我们都假定排序开销包含在 F 参数中。

请注意，在计算 FETCH 调用次数时，为了避免混淆，我们将忽略“判断没有更多符合条件的行”的那次调用，这纯粹是为了简化计算。例如，一张有 10 000 条记录的表上的一个过滤因子为 1%，期望的返回行数是 100，那么我们就假设 F 为 100 而不是 101。

主要访问路径的 QUBE 示例

示例 5.1：主键索引访问

SQL 5.1

```
SELECT   CNO, LNAME, FNAME
FROM     CUST
WHERE    CNO = :CNO
```

这是最简单的例子，图 5.3 很清楚地展示了索引和表上发生的访问。索引两侧的箭头展示了需要被扫描的索引片（在本示例中是单个索引行）。

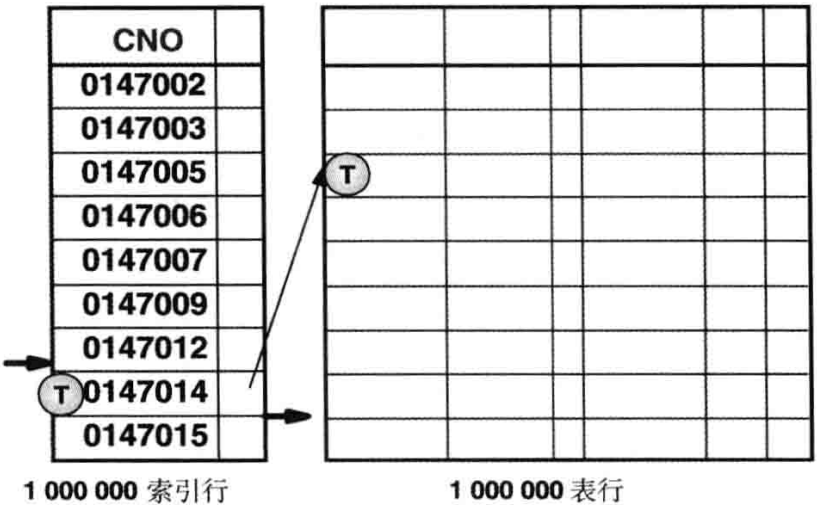


图 5.3　主键索引访问

索引列

我们已经看到索引匹配和索引过滤对于性能的重要性，我们也已经讨论了如何用通过使用索引来避免排序。为了充分理解索引列在这三种方式中是如何被使用的，请确保你已经很清楚如下内容的含义：

匹配字段
过滤字段
避免排序的字段

为了便于比较，我们会在每个 SQL 示例中展示相应的本地响应时间（LRT）。

72 通过主键索引读取一个表行需要随机访问一次表和随机访问一次索引。

```
索引  CNO                TR = 1
表    CUST               TR = 2
提取  1×0.1 ms

LRT                      TR = 2
                         2×10 ms + 0.1 ms
                         ≈ 20 ms
```

示例 5.2：聚簇索引访问

SQL 5.2

```
DECLARE CURSOR52 CURSOR FOR
SELECT      CNO, LNAME, FNAME
FROM        CUST
WHERE       ZIP = :ZIP
            AND
            LNAME = :LNAME
ORDER BY    FNAME
```

在图 5.4 中，我们假设区号为(ZIP)30103 的地区内有 1000 位名为 Joneses 的客户，CURSOR52 通过这个二星索引的两个匹配列访问了一个薄的索引片。这里不需要排序，因为索引已经提供了所需的顺序。当然，只访问索引而不回表是不可能的。

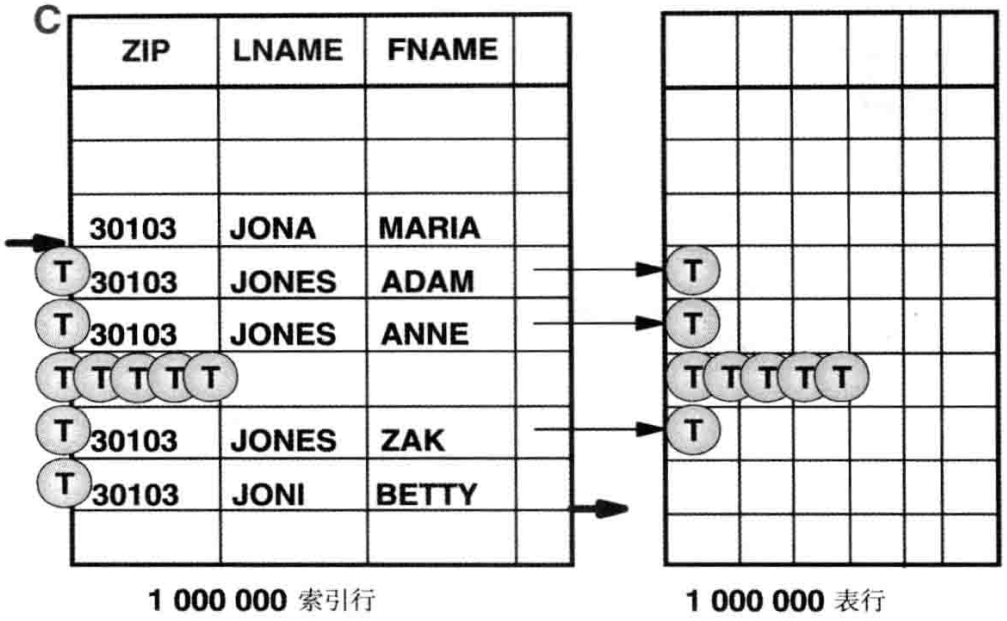


图 5.4　聚簇索引访问

扫描 1000 行的索引片需要多长时间？首先，需要进行一次随机访问来找到索引片上的第一条符合条件的索引行，然后 DBMS 继续向前读取，直到找到列 ZIP 和 LNAME 与输入值不匹配的行为止。包括访问的最后一个不匹配的行，索引的顺序访问数是 1000。因此，根据 QUBE，扫描索引片需要 1 × 10 ms + 1000 × 0.01 ms = 20 ms。

73

因为列 CNO 不在索引中，所以必须从表中读取该列。由于索引是聚簇的，而且我们假设表中的 1000 行是相邻的，于是扫描表片段的成本将会是 1 × 10 ms + 999 × 0.01 ms≈20 ms（本章前面已经描述了如何计算访问次数）。此外，还有 FETCH 调用的时间，根据 QUBE，总的响应时间为 140 ms。由于顺序访问的成本非常低，因此表访问的成本是非常低的。最大的开销来自处理 FETCH 调用的过程。

```
索引 ZIP, LNAME, FNAME          TR = 1        TS = 1000
表   CUST                       TR = 1        TS = 999
提取 1000 × 0.1 ms

LRT                             TR = 2        TS = 1999
                                2 × 10 ms     1999 × 0.01 ms
                                20 ms + 20 ms + 100 ms = 140 ms
```

上表中表的顺序访问次数可能会让人对 QUBE 的精确度产生一个错误的印象。我们将 TS 写成 999 仅仅是为了展示计算访问次数的方法。后面我们将不再这么精确，将它四舍五入为 1000 就可以了。

74 ## 示例 5.3：非聚簇的索引访问

SQL 5.3

```
DECLARE CURSOR53 CURSOR FOR
SELECT      CNO, LNAME, FNAME
FROM        CUST
WHERE       ZIP = :ZIP
            AND
            LNAME = :LNAME
ORDER BY    FNAME
```

如图 5.5 所示，如果表的记录不是通过聚簇索引访问的，那么表的 1000 次访问就会变成随机访问。

```
索引  ZIP, LNAME, FNAME          TR = 1              TS = 1000
表    CUST                       TR = 1000           TS = 0
提取  1000 × 0.1 ms

LRT                              TR = 1001           TS = 1000
                                 1001 × 10 ms        1000 × 0.01 ms
                                 10 s + 10 ms + 100 ms ≈ 10 s
```

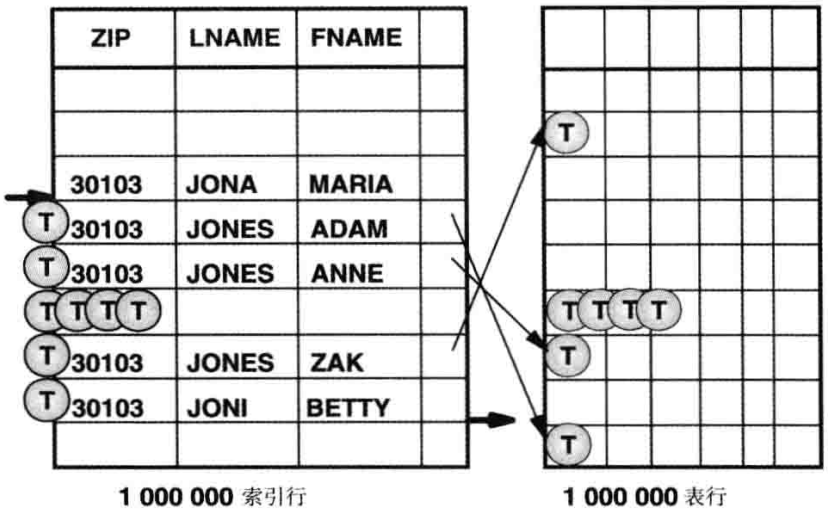


图 5.5　非聚簇索引访问

在这种情况下，将列 CNO 加到索引中让索引成为三星索引会是个好方案，因为这样能使本地响应时间从 10 s 降低至 0.1 s。

75

```
索引  ZIP, LNAME, FNAME, CNO     TR = 1          TS = 1000
表    CUST                       TR = 0          TS = 0
提取  1000 × 0.1 ms

LRT                              TR = 1          TS = 1000
                                 1 × 10 ms       1000 × 0.01 ms
                                 10 ms + 10 ms + 100 ms = 120 ms
```

示例 5.3 展示了非聚簇索引访问可能的危险性，同时也展示了三星索引的重要性。

示例 5.4：使用聚簇索引进行跳跃式顺序表访问

SQL 5.4

```
SELECT      STREET, NUMBER, ZIP, BORN
FROM        CUST
WHERE       LNAME = 'JONES'
            AND
            FNAME = 'ADAM'
            AND
            CITY = 'LONDON'
ORDER BY    BORN
```

示例 5.4 的查询将使用聚簇索引(LNAME,FNAME,BORN,CITY)。在这一索引条件下，要找到住在伦敦的所有名为 Adam Joneses 的客户，需要多少次随机访问和顺序访问呢?

索引扫描将用到两个匹配列。根据图 5.6 中的数据，可以看到只会有两次表访问，而不是四次，因为 CITY 列将被用于索引过滤。这两次表访问将是跳跃式顺序的（请允许我们对仅有两次的表访问使用这一术语），因为该表中的行与聚簇索引的顺序是一致的。由于索引中 BORN 列在 CITY 列的前面，所以这两行不会是连续的。在 QUBE 中，跳跃式顺序访问被算作随机读取。我们将在第 15 章中讨论忽略跳跃式顺序读的影响。

```
索引  LNAME, FNAME, BORN, CITY    TR = 1          TS = 4
表    CUST                       TR = 2          TS = 0
提取  2 × 0.1 ms

LRT                              TR = 3          TS = 4
                                 3 × 10 ms       4 × 0.01 ms
                                 30 ms + 0.04 ms + 0.2 ms ≈ 30 ms
```

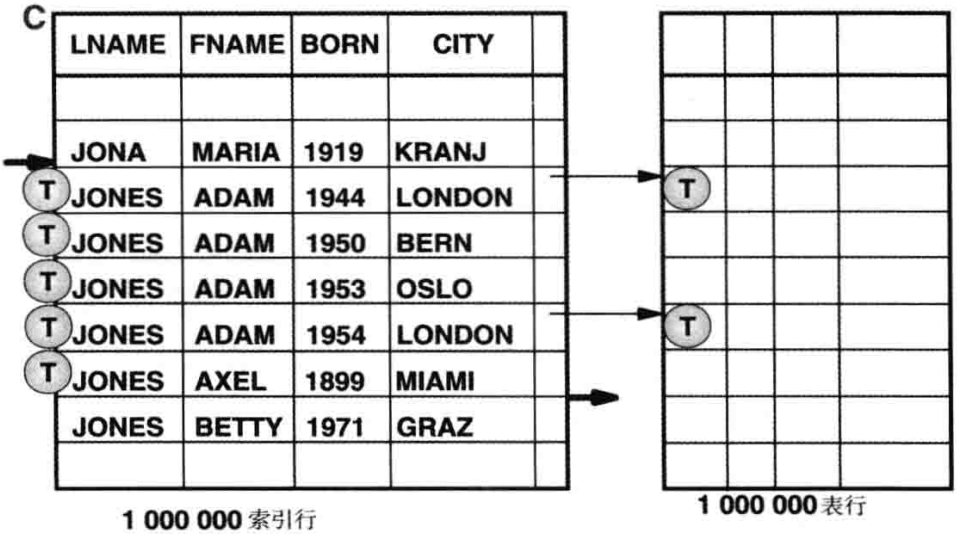


图 5.6　使用聚簇索引的跳跃式顺序表访问

使用满足需求的成本最低的索引还是所能达到的最优索引：示例 1

我们已经通过一些例子展示了如何运用 QUBE 来方便地评估一些比较
76 重要的访问方式，现在我们将展示如何使用这两种技术（BQ 和 QUBE）来完成第 4 章中讨论的整个索引设计过程。

图 5.7 展示了客户表的当前索引。用户会先从弹出窗口中选择姓（LNAME）和城市名称（CITY），然后再开启事务，这会触发程序打开 CURSOR55（如 SQL 5.5 所示）。

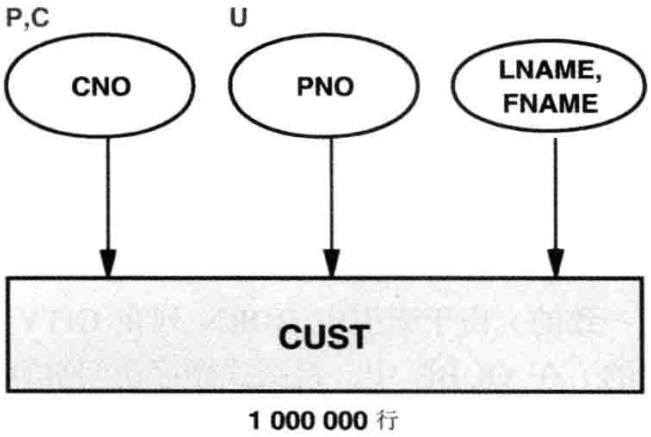


图 5.7　使用满足需求的成本最低的索引还是所能达到的最优索引：示例 1A

SQL 5.5

```
DECLARE CURSOR55 CURSOR FOR
SELECT      CNO, FNAME
FROM        CUST
WHERE       LNAME = :LNAME
            AND
            CITY = :CITY
ORDER BY    FNAME
```

看起来(LNAME,FNAME)是唯一有用的索引。下面我们就来分析一下 SQL 游标使用此索引时，性能是否满足需求。

77 我们将使用 BQ 方法来确定索引(LNAME,FNAME)能否给 CURSOR55（参见 SQL 5.5 和图 5.8）提供足够的性能。若有必要，我们将接着评估一个 FETCH 完整结果集的事务的本地响应时间。评估过程基于以下假设：

1. 表中有 100 万条记录。
2. (LNAME,FNAME)是唯一合适的索引。
3. 谓词 LNAME = :LNAME 的最大过滤因子是 1%。
4. 谓词 CITY = :CITY 的最大过滤因子是 10%。

5. 结果集最多包含 1000 行记录（1 000 000 × 1% × 10%）。
6. 表中的记录按照 CNO 列的大小顺序存储［主键索引（CNO）是聚簇索引，或者表经常按照 CNO 进行排序重组］。

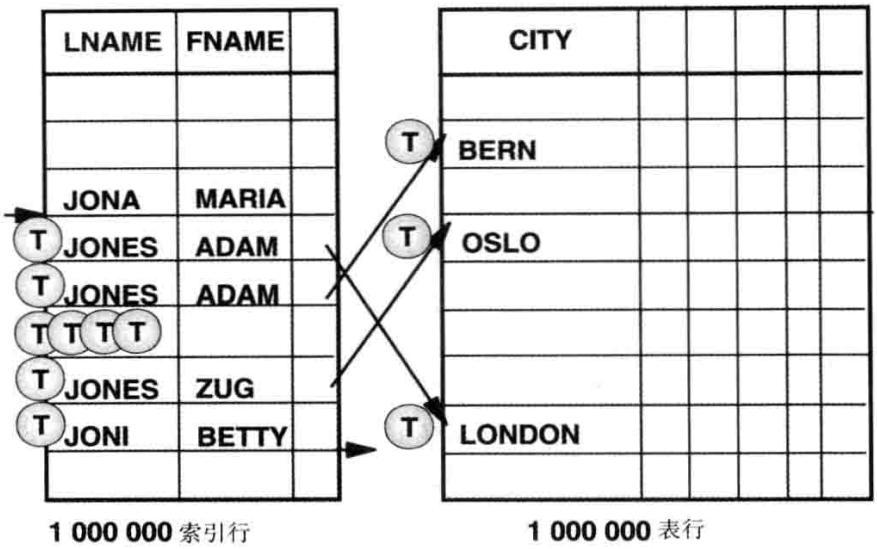


图 5.8　使用满足需求的成本最低的索引还是所能达到的最优索引：示例 1B

该事务的基本问题

78

BQ 方法所提出的问题是：

> 是否有一个现有的或者计划中的索引包含了 WHERE 子句中所涉及的所有列？

换句话说，我们是否现在或者即将有一个半宽索引？答案显然是没有，至少从现有的索引情况看是这样。这并不一定意味着存在问题，性能还是有可能令人满意的。但至少这是一个警示信号，我们应该去思考是否真的存在问题，如果是并且我们决定忽略它，那么它会变得多严重？我们是否需要将那些不在索引中的谓词列加到索引中？这么做将给 BQ 一个肯定的答案，但我们应该牢记这仍然并不能确保良好的性能。

对该事务上限的快速估算

当 DBMS 选择使用索引(LNAME,FNAME)的时候，结果行是按照所请求的顺序（ORDER BY FNAME）被获取的，即不需要排序。因此，DBMS 将不会进行早期物化：OPEN CURSOR 动作不会产生对表或索引的访问，而是会在每次 FETCH 的时候访问索引和表，其中的一些访问会将记录行返回给 FETCH，另一些（大部分）则不会。整个过程将引起一次索引扫描，匹配列只有一个，该列的过滤因子是 1%。根据 QUBE，执行如下 SQL：

```
OPEN CURSOR    1001 × FETCH    CLOSE CURSOR
```

则事务的本地响应时间将会是：

```
索引  LNAME, FNAME      TR = 1              TS = 1% × 1 000 000
表    CUST              TR = 10 000         TS = 0
提取  1000 × 0.1 ms

LRT                     TR = 10 001         TS = 10 000
                        10 001 × 10 ms      10 000 × 0.01 ms
                        100 s + 0.1 s + 0.1 s ≈ 100 s
```

很明显，这里有一个很大的问题，即数据库必须读取 10 000 条不相邻的表记录。这需要耗费很长的时间，将近 2 分钟。

QUBE 是一个对上限的估算法：结果为 100 s 意味着本地响应时间最多会达到 100 s。真实的 LRT 可能比这个值小，特别是当内存和读缓存的大小对数据库来说非常充足时。这种情况下，许多随机访问将不再需要从磁盘驱动器上读取，随机访问的平均响应时间也可能比 10 ms 低。不管怎么说，这不是一个错误的告警。对表的随机访问次数太高了，这个话题我们会在第 15 章展开讨论。

79

注

由于 QUBE 是一个非常粗略的公式，因此在结果中显示多于一位的有效数字是具有误导性的，尽管我们在本书中为了清楚起见会这么做。在真实的报告中，应当这样阐述结论：根据快速估算，被提议的索引能将最差情况的响应时间从 2 min 降至 1 s。

使用满足需求的成本最低的索引还是所能达到的最优索引

设计索引并不只是单纯地为了最小化磁盘 I/O 的总量，而是为了设法让所有程序都运行得足够快，同时还能做到不使用过量的磁盘存储、内存和读缓存，且不使磁盘超载。所有的这些问题已经在第 4 章中详细讨论了。

我们甚至可以将这种决策以数字的方式来表达，比如这样问："将一个每月运行 50 000 次的事务的响应时间从 2s ~ 10s 降低至 0.1s ~ 0.5s，同时节省 1000 秒的 CPU 运算时间，为达到这样的目的每个月支付 10 美元是否值得？"

通过 BQ、QUBE（或者其他方法），我们已经发现了这样一个事实：CURSOR55 会由于没有合适的索引而执行得非常慢。现在，我们面临着一

个困难的问题：我们应该给这个语句设计一个所能达到的最优索引，还是为其设计一个成本最低的满足性能需求的索引？又或者是介于两者之间的某种索引？我们并没有一个硬性的公式去解答这个问题，但我们在第 4 章中讨论的思路应该能够帮助我们设计一个合适的索引。

该事务的最佳索引

使用第 4 章中描述的算法，我们可以发现针对候选方案 A，有两个三星的索引：

```
(LNAME, CITY, FNAME CNO)或(CITY, LNAME, FNAME, CNO)
```

这两个索引都有三颗星，因为它们会扫描一个非常薄的索引片（2 MC）；ORDER BY 列跟在匹配列（它们都使用等值条件）的后面，从而规避了排序；此外，该查询只需访问索引而无须回表。

因为不需要排序，所以没必要考虑候选方案 B 了。

```
索引  LNAME, CITY, FNAME, CNO     TR = 1        TS = 1% × 10% × 1 000 000
 或   CITY, LNAME, FNAME, CNO
提取  1000 × 0.1 ms

LRT                               TR = 1        TS = 1000
                                  1 × 10 ms     1000 × 0.01 ms
                                  10 ms + 10 ms + 100 ms = 120 ms
```

在性能方面，这两个索引并没有什么区别，它们都极大地降低了成本。 80
不幸的是，三星索引将意味着额外增加一个索引，因为将列（CITY）添加到现有索引的 LNAME 和 FNAME 之间可能会对现有程序造成不利的影响。我们在第 4 章已经讨论过这个问题了。这就是为什么我们需要先考虑一下开销更低的可选方案。

半宽索引（最大化索引过滤）

将谓词列 CITY 添加至现有索引(LNAME,FNAME)的末端，可以消除大部分的表访问，因为会引入索引过滤过程。表中的行只有在确定包含所请求的 CITY 值的情况下才会被访问。

该索引(LNAME,FNAME,CITY)是满足 BQ 的，所有的谓词列都在索引中。然而，它只有一颗星，所请求的索引行不是连续的（我们只有一个 MC），当然，它也不是宽索引。但是，这种开销很低的方案是否满足需求了呢？

```
索引  LNAME, FNAME, CITY      TR = 1              TS = 1% × 1 000 000
表    CUST                    TR = 10% × 10 000   TS = 0
提取  1000 × 0.1 ms

LRT                           TR = 1001           TS = 10 000
                              1001 × 10 ms        10 000 × 0.01 ms
                              10 s + 0.1 s + 0.1 s ≈ 10 s
```

借助于这个半宽索引，LRT 由原来的近 2 min 降低到了 10 s，这是一个巨大的进步，但还不够（别忘了最佳索引下的运行时间仅为 120ms）。我们仍然有太多成本很高的 TR，我们需要一个宽索引，以取得更好的性能。

宽索引（只需访问索引）

将一个列添加到半宽索引上，将使该索引变为一个宽索引：(LNAME,FNAME,CITY,CNO)。现在，我们有了两颗星，第二颗和第三颗，同时 LRT 变成了 1s。

```
索引  LNAME, FNAME, CITY, CNO   TR = 1          TS = 1% × 1 000 000
表    CUST                      TR = 0          TS = 0
提取  1000 × 0.1 ms

LRT                             TR = 1          TS = 10 000
                                1 × 10 ms       10 000 × 0.01 ms
                                10 ms + 0.1 s + 0.1 s ≈ 0.2 s
```

表 5.1 提供了各种索引的性能比较，最后一列显示的是由此引起的额外的维护成本。需要注意的是，三星索引将是一个新的索引。对于这些成本提醒如下两点事项：

81

1. 插入和删除意味着修改一个叶子页。如果该叶子页不在缓冲池或者读缓存中，那么整个叶子页就必须从磁盘驱动器上读取。无论所添加或删除的索引行的长度是多少，这都将耗费 10 ms 的时间。
2. 更新 CITY 列会导致响应时间增加 10ms 或 20 ms，具体的值取决于是否需要将索引行迁移至另一个叶子页。通过将 CUST 列作为最后一个索引列的方式可以避免这一移动，详见下述说明。

表中的数字显示，现有索引和半宽索引都是不合适的。另外，尽管使用三星索引比使用宽索引要快一倍，但后者已经够用了，且后者不会引起新索引带来的额外性能负载。

表5.1　示例1在最差输入条件下的各种索引比较

类型	索引	LRT	维护成本
现有索引	LNAME,FNAME	100 s	—
半宽索引	LNAME,FNAME,CITY	10 s	U CITY + 10 – 20 ms
宽索引	LNAME,FNAME,CITY,CNO	0.2 s	U CITY + 10 – 20 ms
三星索引	LNAME,CITY,FNAME,CNO	0.1 s	I/D + 10 ms U + 10 – 20 ms

LRT是最差输入下的QUBE值，I = 插入，D = 删除，U = 更新

重要说明

在这个阶段，我们需要将所有变更的维护成本也考虑进来。比如，在将索引升级为宽索引时，理论上是将 CNO 添加至索引的末尾，但实际上，将它放在 CITY 列的前面会更好，即(LNAME,FNAME,CNO,CITY)。对于这个 SELECT 而言，由于 CITY 列参与的是索引过滤过程，所以它的位置并不会对运行结果造成影响。

同样，当有多个等值谓词作为匹配列时，我们需要考虑这些列在索引上的先后顺序。经常变化的列应当被尽可能地排在后面（相对 LNAME，客户更改地址的可能性更大，因此 LNAME,CITY 可能是比较合适的顺序）。另一方面，我们需要考虑新索引对于其他 SELECT 的帮助有多大。在本例中，我们已经有一个 LNAME 开头的索引了，因此从这个角度看，CITY,LNAME 可能更合适。显然，我们需要在这两者之间进行权衡，对于这个案例而言，后者可能更重要，因为 CITY 并不是一个频繁更新的列。

82

警告

更改现有索引列的顺序与在现有索引列之间添加新列同样危险。在这两种情况下，现有的 SELECT 的执行速度都可能会急剧下降，因为匹配列的数量减少了，或者引入了排序（导致过早产生结果集）。

使用满足需求的成本最低的索引还是所能达到的最优索引：示例 2

我们将用另一个例子再次展示在索引设计过程中如何使用这两种技术，这次是一个稍微复杂一些的 SELECT。

范围事务的 BQ 及 QUBE

在这个例子中，我们像之前一样假设 CITY 的最大过滤因子是 10%，但现在 LNAME 列在一个范围谓词中，我们假设该列有一个比之前更大的过滤因子，比如 10%。现在，结果集的大小最大将会是 1 000 000×10%×10%，即 10 000。

83 图 5.9 展示了客户表的现有索引。现在有两个索引应当被考虑即(CITY)和(LNAME,FNAME)，所以 CURSOR56 被展示了两次（见 SQL 5.6A 和 SQL 5.6B），分别对应一个索引。需要注意的是，这两个索引都不能避免排序。

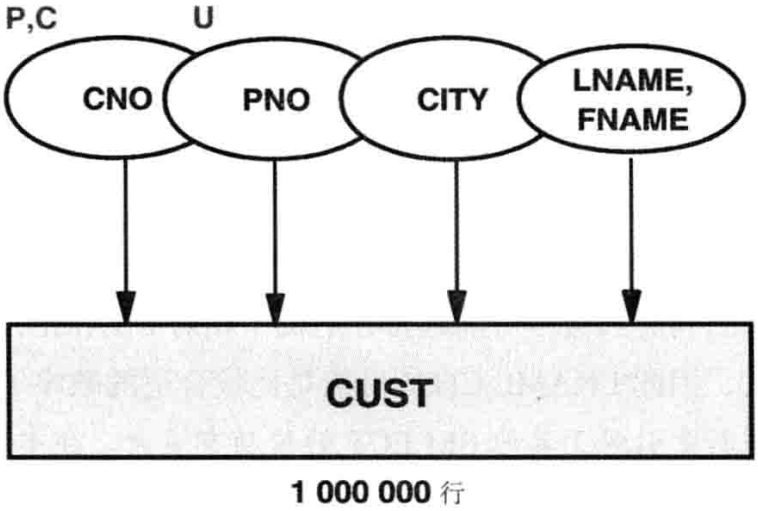


图 5.9　使用满足需求的成本最低的索引还是所能达到的最佳索引：示例 2

索引 CITY—SQL 5.6A

```
DECLARE CURSOR56 CURSOR FOR
SELECT      CNO, FNAME
FROM        CUST
WHERE       CITY = :CITY
            AND
            LNAME BETWEEN :LNAME1 AND :LNAME2
ORDER BY    FNAME
```

索引 LNAME, FNAME—SQL 5.6B

```
DECLARE CURSOR56 CURSOR FOR
SELECT      CNO, FNAME
FROM        CUST
WHERE       CITY = :CITY
            AND
            LNAME BETWEEN :LNAME1 AND :LNAME2
ORDER BY    FNAME
```

这两个索引都不是半宽索引，所以两者都不满足 BQ。QUBE 的结果显示出二者的性能都非常差，无论使用哪个索引其结果都是相同的：两个索引的匹配扫描过程都只能使用 1 个 MC，都需要进行排序，并且都做不到只需访问索引。一颗星也没有，相应地，性能也相当差。

```
索引 LNAME, FNAME      TR = 1              TS = 10% × 1 000 000
或   CITY
表   CUST              TR = 100 000        TS = 0
提取 10 000 × 0.1 ms

LRT                    TR = 100 001        TS = 100 000
                       100 001 × 10 ms     100 000 × 0.01 ms
                       1000 s + 1 s + 1 s ≈ 17 min
```

在这样的索引条件下，很多优化器实际上会选择多索引访问的方式（将在第 10 章中讨论），在访问表之前会读取两个索引片，(LNAME,FNAME)和(CITY)。这一方式的时间开销会比 17min 少得多，但仍然很慢。

该事务的最佳索引

最佳索引很容易得到，候选方案 A 是：

```
(CITY, LNAME, FNAME, CNO)
```

该索引只有两颗星，因为需要排序（ORDER BY 列跟在范围谓词列的后面）。84
因此，候选方案 B 一定是：

```
(CITY, FNAME, LNAME, CNO)
```

它也只有两颗星，不过它具有的是第二颗星而不是第一颗。通过 QUBE 很容易判断出哪一个才是最佳索引。

```
索引 CITY, LNAME, FNAME, CNO     TR = 1     TS = 10% × 10% × 1 000 000
提取 10 000 × 0.1 ms

候选方案 A 的 LRT                TR = 1     TS = 10 000
                                 1 × 10 ms  10 000 × 0.01 ms
                                 10 ms + 0.1 s + 1 s ≈ 1 s

索引 CITY, FNAME, LNAME, CNO     TR = 1       TS = 10% × 1 000 000
提取 10 000 × 0.1 ms

候选方案 B 的 LRT                TR = 1       TS = 100 000
                                 1 × 10 ms    100 000 × 0.01 ms
                                 10 ms + 1 s + 1 s ≈ 2 s
```

最佳索引显然是候选方案 A，(CITY,LNAME,FNAME,CNO)，LRT 为

1s，但即便是最佳索引，事务的性能也一般。

我们早先提出过，如果只是为了比较不同的访问路径，FETCH 的处理可以被忽略，因为它在所有的情况下都是相同的。但是这么做之后，在决定选取哪个索引时要小心一点。例如，在这个案例中，如果忽略 FETCH 的处理，那么候选方案 A（LRT 0.1s）的成本仅为候选方案 B（LRT 1s）成本的 1/10，而实际的优势比例要小得多（2 比 1）。

在这个例子中，候选方案 A 所需要的排序成本与候选方案 B 所节省的 1 秒相比是微不足道的，如前所述，事实上排序成本已经被包含在 FETCH 的开销中了。候选方案 B 的问题在于我们使用的索引片更厚（10%），从而产生了大量的 TS。

半宽索引（最大化索引过滤）

在两个现有索引的末端添加缺少的谓词列可以消除大量的随机表访问，因为这样做能够引入索引过滤过程。只有在确定索引行中同时包含所需的 CITY 和 LNAME 值时，表中的行才会被访问。

在现有索引的基础上，我们可以使用的两个半宽索引是：

```
(CITY, LNAME) 或 (LNAME, FNAME, CITY)
```

我们再一次用 QUBE 来判断哪个是最好的。

85

```
索引  CITY, LNAME        TR = 1              TS = 10% × 10% × 1 000 000
表    CUST               TR = 10 000         TS = 0
提取  10 000 × 0.1 ms

LRT                      TR = 10 001         TS = 10 000
                         10 001 × 10 ms      10 000 × 0.01 ms
                         100 s + 0.1 s + 1 s ≈ 101 s

索引  LNAME, FNAME, CITY     TR = 1                  TS = 10% × 1 000 000
表    CUST                   TR = 10% × 100 000      TS = 0
提取  10 000 × 0.1 ms

LRT                          TR = 10 001             TS = 100 000
                             10 001 × 10 ms          100 000 × 0.01 ms
                             100 s + 1 s + 1 s = 102 s
```

虽然使用第二个索引会有一个更厚的索引片，但这个因素在很大程度上被大量的表访问掩盖了（尽管已经通过过滤的方式将表访问的次数大大减少了）。看来我们可能需要设计一个宽索引来消除对表的 10 000 次 TR 了。

宽索引（只需访问索引）

我们在上面评估的第一个半宽索引上再多加两个列，在第二个索引上再多加一个列，两者就都变成了宽索引：(CITY,LNAME,FNAME,CNO)或(LNAME,FNAME,CITY,CNO)。

```
索引 CITY, LNAME, FNAME, CNO     TR = 1          TS = 10% × 10% × 1 000 000
提取 10 000 × 0.1 ms

LRT                              TR = 1          TS = 10 000
                                 1 × 10 ms       10 000 × 0.01 ms
                                 10 ms + 0.1 s + 1 s ≈ 1 s

索引 LNAME, FNAME, CITY, CNO     TR = 1          TS = 10% × 1 000 000
提取 10 000 × 0.1 ms

LRT                              TR = 1          TS = 100 000
                                 1 × 10 ms       100 000 × 0.01 ms
                                 10 ms + 1 s + 1 s ≈ 2 s
```

现在表访问已经被消除了，两个索引的 LRT 之间的区别就变得很明显了。第一个索引满足第一颗星，即提供了一个薄的索引片，而第二个不满足，所以使用了一个较厚的索引片。

第一个索引当然是候选方案 A 的最佳索引（我们之前已经设计过了），因为它所基于的原始索引只包含 CITY 列，这是在 SELECT 中唯一的等值条件。正因为如此，我们才能将它升级成最佳索引，首先添加 BETWEEN 谓词列（使其变成半宽索引，但仍不够），然后再添加 SELECT 中引用的其他列（使其变成宽索引）。

表 5.2 提供了各种索引的性能比较。

表5.2　示例2在最差输入条件下的各种索引比较　86

类型	索引	LRT	维护成本
现有索引	LNAME,FNAME或CITY	17 min	—
半宽索引	CITY,LNAME	101 s	U LNAME + 10–20 ms
半宽索引	LNAME,FNAME,CITY	102 s	U CITY + 10–20 ms
宽索引	CITY, LNAME, FNAME, CNO	1 s	U L & FNAME + 10–20 ms
宽索引	LNAME, FNAME, CITY, CNO	2 s	U CITY + 10–20 ms
二星索引A	CITY, LNAME, FNAME, CNO	1 s	U L & FNAME + 10–20 ms
二星索引B	CITY, FNAME, LNAME, CNO	2s	U L & FNAME + 10–20 ms

LRT是最差输入下的QUBE值，I = 插入，D = 删除，U = 更新

何时使用 QUBE

理想情况下，QUBE 应当在新方案的设计过程中使用。如果在当前或者计划设计的索引条件下最差输入的本地响应时间过长，比如大于 2s，那么就应当考虑进行索引优化了。

尽管批处理任务的告警阀值视具体的任务而定，但检查一下所有批处理程序相邻提交点之间的最大时间间隔是很重要的。这个值也许应当小于 1s 或 2s，否则该批处理任务有可能会导致事务池和其他批处理任务中的大量锁等待。

QUBE 甚至可以在设计程序之前就开始使用，只需要简单地知道数据库一定会处理的最差输入场景即可。例如，读取 A 表中 10 行不相邻的记录，向 B 表中增加 2 行相邻的记录，等等。事实上，直到估算结果令人满意时再开始设计程序是非常明智的做法。有时，我们可能还必须设计较为复杂的程序结构以满足性能的要求。

第 7 章将讨论如何使用 QUBE 来解决程序上线之后产生的问题。

第 6 章

影响索引设计过程的因素

- 影响第 4 章和第 5 章中所讨论的索引设计过程的重要因素
- 基础估算的验证
- 多个窄索引片
- 数据库管理系统的特性
- 对优化器而言较为困难的谓词
- 布尔谓词
- 过滤因子的问题及缺陷

I/O 时间估算的验证

在第 4 章中，我们引入了以下 I/O 时间：

随机读取	10 ms（页的大小为 4KB 或 8KB）
顺序读取	40 MB/s

这些数据是假定系统使用当前硬件并在一个合理的负载下运行时的值。一些系统可能运行得更慢或处于超负荷状态，所以，进行下面这些检验可能是有用的。为此，我们需要建一张表，该表有 100 万行记录，每行记录的平均长度约为 400 字节。根据假定的顺序 I/O 时间计算，以顺序 I/O 读取 400 字节长度的行所花费的时间为 400 byte/40 MB/s = 0.01ms。

于是，进行如下扫描所消耗的时长便是确定的（括号中的数据是根据我们在第 4 章中使用的估算数据来预测的）：

- 由一个所有行都不满足条件的 SELECT 查询引起的一次全表扫描（$1 \times 10\ ms + 1\ 000\ 000 \times 0.01\ ms \approx 10\ s$）。
- 一次涉及 1000 次 FETCH 调用的索引扫描，使用如(LNAME,FNAME)这样的索引，导致 1000 行的索引片扫描和 1000 次对表的同步读（$1000 \times 0.01\ ms + 1000 \times 10\ ms = 10\ ms + 10\ s \approx 10\ s$）。

88 我们有必要以这样的方式进行测量，即保证在程序刚开始的时候，查询所需要的页不在内存或者读缓存中。如果系统运行相当繁忙，那么在两次测量之间间隔几个小时可能就达到这一目的了。

如果测量出的时间比预计的时间长很多，那么便需要明确造成这一结果的主要原因：磁盘设备或路径较慢？还是磁盘设备超负载运行？

无论这些测出的数字是乐观或悲观，作为相对性能指标，它们是有用的。

多个窄索引片

通常情况下，我们为一个 SELECT 语句设计一个索引（而不是反过来），但有时候为了让数据库管理系统能够以最高效的方式使用一个已经存在的索引，对一个 SQL 语句进行重写也是可取的。因为优化器并不知道一切！

当 WHERE 子句包含两个范围谓词时，一个完美的优化器能够构造出一个由多个窄索引片组成的访问路径，如 CURSOR61（参见 SQL 6.1 和图 6.1）。在这个例子中，我们假设 LNAME 和 CITY 的输入值中都包含缩减值，如 JO 和 LO，程序由此生成相应的范围值。

SQL 6.1

```
DECLARE CURSOR61 CURSOR FOR
SELECT      CNO, FNAME
FROM        CUST
WHERE       LNAME BETWEEN :LNAME1 AND :LNAME2
            AND
            CITY BETWEEN :CITY1 AND :CITY2
ORDER BY    FNAME
```

LNAME	CITY	FNAME	CNO
.....	.....	.....	.....
JONES	LISBON	MARIA	2026477
JONES	LONDON	DAVID	5643234
JONES	LONDON	MIKE	1234567
JONES	LONDON	TED	0037378
JONES	MADRID	ADAM	0968431
.....	.....	.....	.....
JONSON	BRUSSELS	INGA	3620551
JONSON	LONDON	DAVID	6643234
JONSON	MILAN	SOPHIA	2937633
.....	.....	.....	.....

图 6.1　MC = 2 情况下的多个窄索引片（CURSOR62）

因为一个范围谓词字段是索引匹配过程中的最后一个匹配字段，所以让数据库管理系统读取多个窄索引片（由 LNAME 和 CITY 共同定义）需要额外的程序逻辑。谓词条件 LNAME BETWEEN :LNAME1 AND :LNAME2 必须被修改为 LNAME = :LNAME。这可以通过一个由触发器维护的辅助表来完成，该表包含客户表中 LNAME 列的所有不同的值。当用户输入字符串 JO 时，程序便从辅助表中每次读取一个满足条件的 LNAME，并用每个 LNAME 值打开 CURSOR62（参见 SQL 6.2）。

SQL 6.2

```
DECLARE CURSOR62 CURSOR FOR
SELECT      CNO, FNAME
FROM        CUST
WHERE       LNAME =:LNAME
            AND
            CITY BETWEEN :CITY1 AND :CITY2
ORDER BY    FNAME
```

现在，即便是一个没有完美优化器的数据库管理系统也将会从索引 89 (LNAME,CITY,FNAME,CNO) 中读取多个窄片。EXPLAIN 将报告 CURSOR62 的匹配列为 MC = 2。响应时间将取决于索引片的数量以及每一个索引片中的行数。

让我们使用如下假设对比一下这两种选择：

- customer 表中有 100 万行记录
- LNAME 最坏情况下的过滤因子为 1%
- CITY 最坏情况下的过滤因子为 10%

被 CURSOR61 扫描的索引片由 10 000 行索引记录组成(MC = 1; 1 000 000 的 1%为 10 000），其中只有 10%的 CITY 值（1000 行）是满足需要的。

我们同时假设此索引片中有 50 个不同的 LNAME 值。用 CURSOR62 90 优化过的程序将会读取 50 个索引片（平均每个索引片有 20 行记录），这 1000 行中的每一行都是满足查询条件的 CITY。

根据 QUBE（鉴于只是单纯地做比较，我们忽略 FETCH 所带来的时间开销），两个游标的时间开销为：

$$\text{CURSOR 61} \quad 1 \times 10\ \text{ms} + 10\,000 \times 0.01\ \text{ms} \approx 0.1\ \text{s}$$
$$\text{CURSOR 62} \quad 50 \times (1 \times 10\ \text{ms} + 20 \times 0.01\ \text{ms}) \approx 0.5\ \text{s}$$

这有些让人吃惊，但却是合理的。顺序处理的确有时比跳跃处理要来得高效，原因是每一次的跳跃都可能意味着一次随机访问。当数字不同时，用 CURSOR62 优化后的程序有可能会比单纯使用 CURSOR61 的程序快很多。

为了证明最后说的这个观点，我们来考虑一个场景。该场景有一张更大的表（100 000 000 行记录），且第一个索引列拥有较高的过滤因子，如 SQL 6.3 中所示。

SQL 6.3

```
SELECT      CNO, FNAME
FROM        CUST
WHERE       LNAME BETWEEN :LNAME1 AND :LNAME2    FF max 10%
            AND
            CITY BETWEEN :CITY1 AND :CITY2       FF max 0.01%
ORDER BY    FNAME
```

结果集的大小将会是 0.01 % × 10% × 100 000 000 = 1000 行。让我们假设有一个索引(LNAME,CITY,FNAME,CNO)且 LNAME 在 1000 行结果集的表中有 50 个不同的值。现在，简单程序（两个 BETWEEN）的时间花费估算结果（一个宽索引片）为

$$1 \times 10\ \text{ms} + 10\,000\,000 \times 0.01\ \text{ms} \approx 100\ \text{s}$$

然而复杂程序（50 个窄索引片）的估算结果仍旧为

$$50 \times (1 \times 10\ \text{ms} + 20 \times 0.01\ \text{ms}) \approx 0.5\ \text{s}$$

Oracle 9i 能够生成一种跳跃式顺序的访问路径来替代全索引扫描——*索引跳跃扫描*（index skip scan)。在我们的例子中，这将意味着对每一个 LNAME 的值扫描一个索引片。这些索引片有两个匹配字段，LNAME 和 CITY。

简单就是美（和安全）

使用辅助表来保存 LNAME 不同值的解决方案，是一个典型的将数据完整性和性能进行折中的例子。理论上，使用一个触发器来维护辅助表是没有数据完整性的风险的。然而，触发器是应用程序，是程序就不会是完美的。
91 当 ALAN JONES 的记录从客户表删除且表中还有另一个 JONES 的记录时，是否能完全确定触发器不会同时把 JONES 的记录从辅助表中删除呢？只有当量化估算表明不这样做性能就无法接受时，我们才考虑采用这样的调整手段。有时，这样的调整方式反而会对性能产生负面影响，正如我们之前所看到的那样。

困难谓词

定义：如果一个谓词字段不能参与定义索引片，即它无法成为一个匹配谓词，那么我们就说它对优化器而言太困难了。

用户友好型的数据库管理系统供应商会提供一个对自己的优化器而言的困难谓词列表。这些谓词有时被称为*不可索引化*（nonindexable）。典型的例子包括字段函数，如

```
COL 1 CONCAT COL2 > :hv
```

以及否定函数，如

```
COL1 NOT BETWEEN :hv1 AND :hv2
```

随着新版本的发布，数据库管理系统的困难谓词缺陷列表的长度在逐步减小。然而，一些缺陷虽然可能已经被移除，但移除得并不彻底。举个例子，一个函数或者算术表达式对优化器而言已经变得足够简单，但并不是它的所有变化形式都是如此。至于这个谓词对当前使用的优化器而言是否是困难的，无论自己是否清楚，用 EXPLAIN 检查一下总是明智的。

我们接下来要讨论的谓词尤其重要。

LIKE 谓词

在 CURSOR61 和 CURSOR62 中，LIKE 也许能起到和 BETWEEN 同样的作用，具体还要取决于绑定变量的值。LIKE 实际上更方便，但若是与绑定变量同时使用，那么这对优化器而言就太困难了。当然，最终得到的是正确的结果集，但如果数据库管理系统因为 LIKE 谓词不参与匹配过程而扫描整个索引，那么访问路径将会变得很慢。

这些问题也取决于数据库管理系统。例如，DB2 for z/OS 会为 COL LIKE :COL 创建两个访问路径，并当 DB2 for z/OS 知道传入绑定变量:COL 的值后，便会在执行时从这两个路径中选择一个。如果值是%XX，那么它将选择全索引扫描；如果值是 XX%，则会选择索引片扫描——除非在 LNAME 字段上有一个字段处理程序。在那些受“national characters”困扰的国家，经常会使用一些影响排序顺序的字段处理程序。

只有在已知优化器不会造成问题的前提下，才应该使用谓词 COL LIKE :COL。

92 OR 操作符和布尔谓词

即便是简单的谓词，如果它们与其他谓词之间为 OR 操作，那么对于优化器而言它们也可能是困难的。这样的谓词只有在多索引访问下才有可能参与定义一个索引片。我们将在第 10 章中对此进行讨论。

下面的例子(SQL 6.4A 和 SQL 6.4B)将明确这一点。我们假设在 TABLE 表上有一个索引(A,B)。

SQL 6.4A

```
SELECT   A, B, C
FROM     TABLE
WHERE    A > :A
         AND
         B > :B
```

SQL 6.4B

```
SELECT   A, B, C
FROM     TABLE
WHERE    A > :A
         OR
         B > :B
```

在 SQL 6.4A 中，单个索引片将会被扫描，由谓词 A > :A 定义，因为它是个范围谓词，所以它是唯一的一个匹配字段。B 字段因为索引筛选将会在此次扫描中被检查，如果谓词 B > :B 不满足条件，则索引条目将会被忽略。

在 SQL 6.4B 中，由于 OR 操作符的存在，数据库管理系统不能只读这一个索引片，因为即使不满足谓词 A > :A 条件，谓词 B > :B 的条件也可能会满足。可行的替代方案有全索引扫描、全表扫描，以及多索引访问（如果存在另一个以字段 B 开始的索引的话）。

在这个例子中，我们不能责怪优化器。有时候一个拥有复杂 WHERE 子句的游标可能必须被拆成多个带有简单 WHERE 子句的游标来避免此类问题。大体上，假设一个谓词的判定结果为 false，而这时如果不检查其他谓词就不能确定地将一行记录排除在外，那么这类谓词对优化器而言就是太过困难的。在 SQL 6.4B 中，仅仅因为不满足谓词 A > :A 或仅仅因为不满足谓词 B > :B 是无法排除一行记录的。在 SQL 6.4A 中，如果一行记录不满足谓词 A > :A，那么该条记录就可以被排除而不必再检查它是否满足谓词 B > :B。

这样的谓词被称为非布尔谓词（non-Boolean term）或被称为非 BT 谓词

（non-BT）。如果像 SQL 6.4A 那样在 WHERE 子句中只有 AND 操作符，那么所有的简单谓词都是 BT 谓词（布尔谓词），也就是“好”的谓词，因为只要任何一个简单谓词被判定为不满足条件，那么这一行就可以被排除。

总结

93

除非访问路径是一个多索引扫描，否则只有 BT 谓词可以参与定义索引片。

我们来考虑一个定义身材高大的男人的 WHERE 子句（参见 SQL 6.5）：

SQL 6.5

```
WHERE   SEX = 'M'
        AND
        (WEIGHT > 90
        OR
        HEIGHT > 190)
```

只有第一个简单谓词 SEX = ‘M’是 BT 谓词。如果 WEIGHT 谓词判定为 false，HEIGHT 谓词的判定结果可能为 true 也可能相反。这两个谓词都无法仅通过自己的判定结果来排除一行记录。

这就是为什么我们在第 4 章用理想索引设计方法来设计候选索引 A 和 B 时提出了这样的结论：只将那些对优化器而言足够简单的谓词添加至索引上。对于 SQL 6.5 而言，在设计方法的第一步中，只有 SEX 字段可以被用到。

IN 谓词

谓词 COL IN (:hv1, :hv2 …)可能会给优化器制造一些麻烦。例如，在 DB2 for z/OS V7 中，只有一个 IN 谓词能够被用来做匹配，即能够参与定义索引片，不过参与匹配的 IN 谓词中的字段不一定是最后一个匹配字段。因此，当理想候选索引 A 被选中时，选择性最高的 IN 谓词的字段应当在第 1 步就被包含进索引。SQL 6.6 提供了一个例子。

SQL 6.6

```
DECLARE CURSOR66 CURSOR FOR
SELECT   A, B, C, D, E
FROM     TX
WHERE    A = :A
         AND
         B IN (:B1, :B2, :B3)   FF = 0...10%
         AND
         C > 1
         AND
         E IN (:E1, :E2, :E3)   FF = 0...1%
```

94　下面候选 A 中的数字对应于我们在第 4 章中描述的步骤。

候选 A

1. 挑选出字段 A 和 E 作为索引的起始字段。
2. 加上字段 C。
3. 无字段。
4. 加上字段 B 和 D。

候选索引 A 为(A,E,C,B,D)或者以下变化形式中的任何一个：

- (A,E,C,D,B)
- (E,A,C,B,D)
- (E,A,C,D,B)

对于 CURSOR66 而言，所有这些候选索引都满足 MC = 3。数据库管理系统读三个索引片(A,:E1,C)、(A,:E2,C)和(A,:E3,C)，而无须使用其他任何特殊的方式。字段 B 的 IN 谓词将参与筛选过程。如果需要有 4 个匹配字段(即 9 个索引片)，那么我们很可能需要把 SELECT 语句拆分成三个，每个拆分出的语句中只有 IN 谓词字段列表中的一个字段。第一个 SELECT 语句包含谓词 B = :B1，第二个为 B = :B2，第三个为 B = :B3。这三个 SELECT 语句可以通过一个 UNION ALL 再结合为一个游标。对这个游标进行 EXPLAIN 将显示三次 MC = 4。

候选 B

IN 谓词的字段不应该在第 1 步中就被选择出来，否则这将破坏结果集的顺序要求，从而必须进行一次排序操作了。

过滤因子隐患

贯穿本书，我们关注于*最糟糕的场景*，这些场景都是在输入值对应于最大过滤因子时出现的。然而，情况并不总是如此，就像 SQL 6.7 所示的那样。

SQL 6.7

```
SELECT      B
FROM        TABLE
WHERE       A = :A (FF = 1%)
            AND
            C > :C (FF = 0...10%)
ORDER BY    B
WE WANT 20 ROWS PLEASE
```

在这个 SELECT 语句中，可以从 WE WANT n ROWS PLEASE 语句推 95
断出，我们并不关心整个结果集。一旦程序获取了 20 行记录，屏幕上便会将这 20 行结果集展示给用户，而不管整个结果集的大小是多少。

为了解释过滤因子对性能的隐式影响，我们来比较一下过滤因子处于两种极端情况下的 QUBE 结果。第一种极端情况提供可能的最小的结果集，即 0 行记录；第二种则提供很大的结果集（1000 行记录）。图 6.2 展示了在这两种场景下使用索引(A,B,C)的情况。每一种场景下，匹配字段 A 都拥有相同的过滤因子，结果集大小的不同完全取决于字段 C。

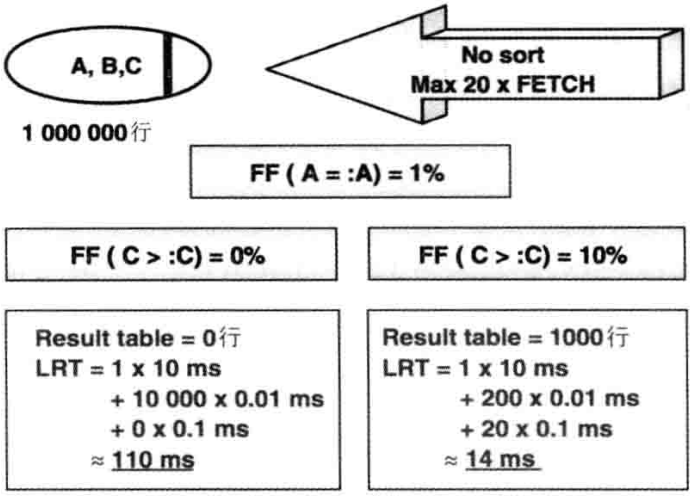


图 6.2　过滤因子缺陷

让我们首先考虑第二种超大结果集的场景，通常我们认为这将带来最糟糕的性能。一次索引匹配扫描（1 MC，只需扫描索引而无须访问表）将访问一个包含 20 × 10 行记录的索引片，因为必须扫描 10 个索引行才能提供出结果集中的一行记录（字段 C 的 FF 为 10%）。

```
索引 A, B, C        TR = 1              TS = 20 × 10
提取 20 × 0.1 ms

超大结果集的 LRT    TR = 1              TS = 200
                    1 × 10 ms           200 × 0.01 ms
                    10 ms + 2 ms + 2 ms = 14 ms
```

在第一种场景下，没有记录满足字段 C 的谓词条件，所以过滤因子为 0%。当然，数据库管理系统必须扫描索引片中的每一行，即 1 000 000 的 1%，才会知道这一情况。

```
索引 A, B, C        TR = 1              TS = 1% × 1 000 000
提取 0 × 0.1 ms

空结果集的 LRT      TR = 1              TS = 10 000
                    1 × 10 ms           10 000 × 0.01 ms
                    10 ms + 100 ms + 0 ms = 110 ms
```

96

很明显，对于这个事务而言，最差场景是结果集为空的场景，或者需要扫描整个索引片才能得出结果集的场景。更糟糕的是，如果这个案例不能做到只扫描索引或者对应的索引不是聚簇索引，那么就需要进行额外的 10 000 次随机读取，而这将消耗将近 2 分钟的时间，结果却是发现没有一行满足条件的数据！

如果索引匹配扫描过程中扫描的是一个更厚的索引片，比如字段 A 的过滤因子为 20%，那么即便只需扫描一个索引，LRT 也将达到 2 s（200 000 × 0.01 ms）！当查询结果只需要一屏幕的数据时，满足第一颗星（最小化待扫描的索引片厚度）比满足第二颗星（避免排序）更为重要。

当以下三个条件同时满足时，这种过滤因子隐患可能会产生：

1. 访问路径中没有排序。
2. 第一屏幕结果一建立就回应。
3. 不是所有的谓词字段都参与定义待扫描的索引片——换句话说就是，不是所有字段都为匹配字段。

这个例子满足了所有这三个条件：

- 访问路径中没有排序，因为字段 B（Order By 字段）紧接在等值谓词字段 A 后面，并且字段 A 是索引的第一个字段。
- 第一屏幕结果一建立事务就响应。
- 一个谓词字段（字段 C）不参与定义待扫描的索引片。

我们应该充分理解这三个条件背后的逻辑：

- 如果条件 1 和条件 2 不为真，那么完整的结果集表总是会被数据库管理系统或者应用程序（通过进行 FETCH 操作来物化）物化。
- 如果条件 3 不为真，就不存在索引过滤（因为所有的谓词字段都是匹配字段），索引片中的每一个索引行都是满足条件的。即使结果集为空，数据库管理系统也只需通过一次访问就能判断出这一情况。

过滤因子隐患的例子

这一节中我们将再一次演示如何使用第 4 章和第 5 章中讨论过的索引设计方法中的两种估算技巧。

97 在这个例子中，用户在开始查询前首先在一个弹出窗口中选择姓（LNAME）和城市（CITY），随后便打开 CURSOR68（见 SQL 6.8）。我们之前分析这个例子时需要的是整个结果集。而在这个例子中，程序发起的

FETCH 调用不会超过 20 次，也许刚好够填满一个屏幕。下一个查询事务将显示接下来的 20 条结果，以此类推。这种方式使程序变得复杂，但当结果集很大的时候，这种方式提升了第一屏结果的响应速度。和之前一样，我们将最大的过滤因子假设为 1%（LNAME）及 10%（CITY）。

SQL 6.8

```
DECLARE CURSOR68 CURSOR FOR
SELECT      CNO, FNAME
FROM        CUST
WHERE       LNAME = :LNAME
            AND
            CITY = :CITY
ORDER BY    FNAME
WE WANT 20 ROWS PLEASE

OPEN CURSOR CURSOR610
FETCH CURSOR CURSOR610 最多20行
CLOSE CURSOR CURSOR610
```

图 6.3 展示了 CUSTOMER 表上当前的索引，它们都无法通过 BQ 测试：

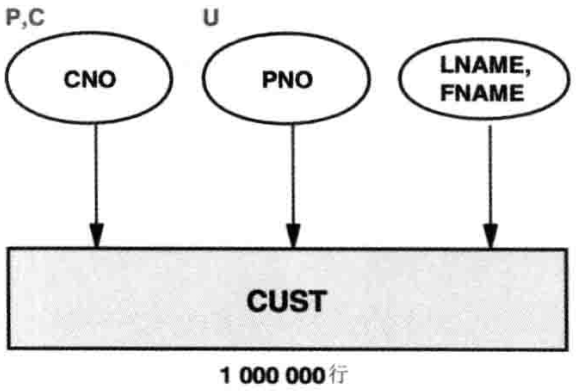


图 6.3　过滤因子缺陷举例——索引

是否有一个已经存在的或者计划创建的索引，包含了 WHERE 子句中涉及的所有字段（一个半宽索引）？

如果我们决定使用索引(LNAME,FNAME)，那么问题将严重到何种程度？最糟情况下的 QUBE 结果将回答这个问题。

当数据库管理系统选择索引(LNAME,FNAME)时，结果行是按照所要求 98
的顺序获取的（ORDER BY FNAME），也就不需要进行排序操作。因此，数据库管理系统不会做早期的物化：OPEN CURSOR 不会引起表访问，不过 FETCH 操作将同时对索引和表进行访问，其中的一些 FETCH 操作会返回结果行，而另外大部分的 FETCH 操作则不会。

如图 6.4 所示，最先发生的两次访问（一次对索引，一次对表）产生了

一行结果行记录，因为第一个 JONES 生活在 LONDON。在第一次 FETCH 调用被满足后程序便发起了第二次 FETCH 调用。下一个 JONES 并不生活在 LONDON，所以没有产生结果行，于是会继续进行更多的对索引及表的访问，直到生活在 LONDON 的 JONES 被找到以满足 FETCH 调用。因为只有 10%的用户生活在 LONDON，所以平均一次 FETCH 调用需要检查 10 个 JONES 条目——10 次索引访问及 10 次表访问——才能将一行记录加入到结果表中。

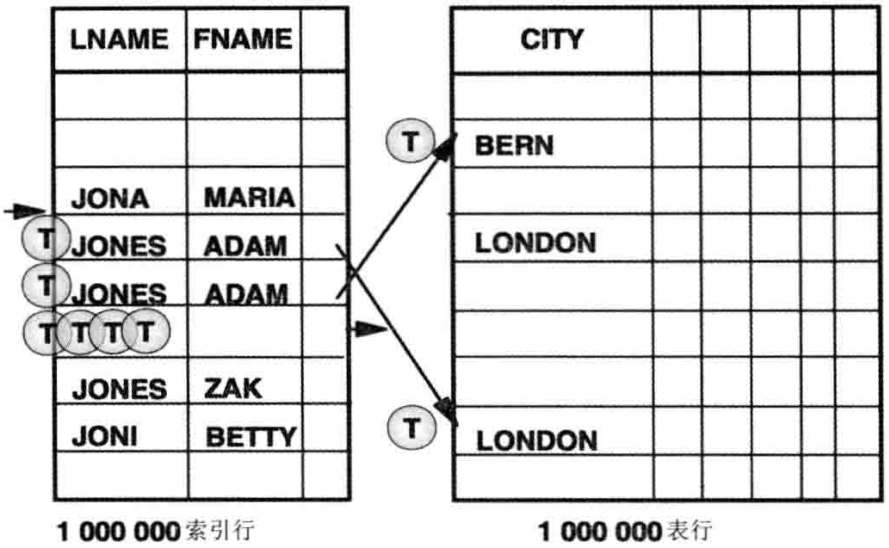


图 6.4　过滤因子缺陷举例——索引条目

因此，创建由 20 个客户组成的一屏幕结果集平均需要进行 20 × (10 + 10) = 400 次访问。

索引 LNAME, FNAME	TR = 1	TS = 20 × 10
表 CUST	TR = 20 × 10	TS = 0
提取 20 × 0.1 ms		
超大结果集的 LRT	TR = 201	TS = 200
	201 × 10 ms	200 × 0.01 ms
	2 s + 2 ms + 2 ms ≈ 2 s	

根据 QUBE 得出的本地响应时间是 2s，也许还是可以接受的。但这就是最糟糕的情况了么？假定的过滤因子 CITY = :CITY 为 10%、LNAME = :LNAME 为 1%已经是最大值了。然而，在这个例子中，产生最大结果集的过滤因子值并不是最糟糕的情况——本例是过滤因子陷阱的另一个例子。过滤因子陷阱的三个条件都适用于用索引(LNAME,FNAME)进行处理的这个游标：

99

- 访问路径上没有排序。

- 第一屏幕结果一建立事务就响应。
- 不是所有的谓词字段都参与了定义待扫描的索引片——换句话说，不是所有字段都为匹配字段。

因此，为了创建一条“不存在符合条件的用户”的消息，所需花费的时间可能比返回一个大结果集的第一页数据所花费的时间更长。

让我们考虑一种极端的情况，数据库管理系统必须扫描一个最大的索引片，但最终只产生了一个空的结果集——一个城市中，没有任何一个客户的姓是常见的姓，此时数据库管理系统仍旧必须扫描整个索引片，并对每一个索引行进行相应表行的检查。

```
索引 LNAME, FNAME        TR = 1              TS = 1% × 1 000 000
表   CUST                 TR = 10 000         TS = 0
提取 0×0.1 ms

空结果集的 LRT            TR = 10 001         TS = 10 000
                         10 001 × 10 ms      10 000 × 0.01 ms
                         100 s + 100 ms + 0 ms ≈ 100 s
```

将近 2 分钟而不是 2 秒钟！索引必须要改善，此时我们将再一次面对那个难题：是选择刚好够用的最低成本的改造，还是设计一个最佳索引，又或是设计一个折中的方案？

最佳索引

我们能够很容易地设计出理想索引。候选 A 为

```
(LNAME,CITY,FNAME,CNO)
```

因为这个索引满足三星索引的要求，所以没必要再去设计候选 B 了。该索引的 QUBE 为：

```
索引 LNAME,CITY,FNAME,CNO      TR = 1              TS = 20
数据提取 20 × 0.1 ms

LRT                            TR = 1              TS = 20
                               1 × 10 ms           20 × 0.01 ms
                               10 ms + 0.2 ms + 2 ms ≈ 12 ms
```

以上对应的是在最糟糕的场景下使用候选 A 索引的情况：至少有 20 行结果记录。在使用理想索引的情况下，当整个结果集无法全部显示在一个屏幕上时，建立一个有 20 行记录的结果表需要 21 次索引访问。使用索引(LNAME,FNAME)与使用索引(LNAME,CITY,FNAME,CNO)的最糟输入响

应时间相差约 4 个数量级。

100 如果结果集表为空，那么使用这个三星索引只需要一次索引访问就可以判定出来。从定义上能看出，三星索引不满足陷阱发生条件列表中的第三条。

然而不幸的是，一个三星索引意味着额外增加一个索引，因为在现有的索引中，LNAME 和 FNAME 之间增加一个 CITY 的字段可能会对现有程序造成负面影响。我们已经在第 4 章和第 5 章中讨论过这个问题。这也就是为什么我们需要先考虑廉价替代方案的原因。

半宽索引（最大化索引过滤）

在现有索引(LNAME,FNAME)的最后增加谓词字段 CITY 可以消除绝大多数的表访问，因为在这个简单的场景中，优化器可以进行索引过滤，仅当 CITY 值符合需求时才读取表行。

索引(LNAME,FNAME,CITY)能够通过 BQ 测试：所有的谓词字段都在索引中。然而，它仅仅是一星索引。符合条件的索引行在物理上并不挨在一起（MC 仅为 1），并且索引并不宽。那么这个廉价的解决方案是否能提供可接受的响应时间呢?

现在使用这个索引的游标落入了过滤因子缺陷的范畴。因此我们必须估算使用索引(LNAME,NAME,CITY)时过滤因子的极限值。其中一个过滤因子将导致这种访问路径下最糟糕的性能。

最大结果集（例如 1000 行）

```
FF (LNAME = :LNAME) = 1%
FF (CITY = :CITY) = 10%
FF (LNAME = :LNAME AND CITY = :CITY) = 0.1%
```

因为 CITY 的过滤因子为 10%，所以每 10 个索引行满足一次谓词判定 CITY = :CITY。所以为了找出一行满足条件的结果记录，平均需要访问 10 次索引。

```
索引 LNAME,FNAME,CITY    TR = 1              TS = 20 × 10
表                       TR = 20             TS = 0
提取 20 × 0.1 ms

1000 行结果集的 LRT      TR = 21             TS = 200
                         21 × 10 ms          200 × 0.01 ms
                         210 ms + 2 ms + 2 ms = 214 ms
```

空结果集

```
FF (LNAME = :LNAME) = 1%
FF (CITY = :CITY) = 0%
FF (LNAME = :LNAME AND CITY = :CITY) = 0%
```

因为结果集为空，所以在以过滤因子为 1%的谓词 LNAME = :LNAME 定义的索引片中，每一行都需要被校验，CITY 字段参与过滤的结果清晰地反映在了表访问的数字上。 101

```
索引 LNAME, FNAME, CITY      TR = 1              TS = 10 000
表                           TR = 0              TS = 0
提取 0 × 0.1 ms

空结果集的 LRT               TR = 1              TS = 10 000
                             1 × 10ms            10 000 × 0.01ms
                             10 ms + 100 ms + 0 ms = 110 ms
```

宽索引（只需访问索引）

向上述半宽索引上再添加一个字段，使得索引变宽：(LNAME,FNAME,CITY,CNO)。

```
索引 LNAME, FNAME, CITY, CNO  TR = 1             TS = 20 × 10
提取 20 × 0.1 ms

1000 行结果集的 LRT           TR = 1             TS = 200
                              1 × 10 ms          200 × 0.01 ms
                              10 ms + 2 ms + 2 ms = 14 ms
索引 LNAME, FNAME, CITY, CNO  TR = 1             TS = 10 000
提取 0 × 0.1 ms

空结果集的 LRT                TR = 1             TS = 10 000
                              1 × 10 ms          10 000 × 0.01 ms
                              10 ms + 100 ms + 0 ms = 110 ms
```

在空结果集的场景下，本地响应时间保持在 0.1s。从一个多屏结果集中产生一个屏幕的结果集的事务现在变得非常快，因为不再需要对表进行 20 次 TR 了。

总结

1. 性能最差的场景取决于所使用的索引：
   - 对于原始索引而言，在结果集表为空时性能最差（最常见的 LNAME 加上最不常见的 CITY）。
   - 对于半宽索引而言，虽然我们的例子中是结果集表最大时性能最差，但性能最差的场景可能是这两者（结果集表为空和结果集表最大）中的任何一个，这取决于表访问和索引片扫描的

相对成本。（根据 QUBE）20 次 TR 所耗费的时间（200ms）等于 20 000 次 TS 所耗费的时间（200ms）。若每屏的行数减少，或索引片更厚，那么结果集为空就可能成为性能最差的场景。

- 对于宽索引而言，结果集为空时性能最差。
- 对于三星索引而言，结果集最大时性能最差（结果集为空时只需要进行一次索引访问）。

102

2. 原始索引将在结果集为空时（而不是结果集最大时）成为一个灾难。这是因为它需要进行大量的表访问来校验 CITY 字段。
3. 半宽索引及宽索引都能提供还不错的性能，即便是在结果集为空的情况下。两个索引都避免了不必要的表访问，并且索引片扫描的代价也相对较小。随着由字段 LNAME 定义的索引片的数据增长，本地响应时间的增长也相对较缓慢。如果一个公司拥有了很多来自 South Korea 的顾客，那么有一天 LEE 的索引片可能包含 100 000 索引行。在结果集这么大的情况下，这两个索引依旧能够提供比较不错的响应时间；最差的输入值所带来的响应时间将会是 1s（100 000 × 0.01 ms）。
4. 相比使用半宽索引，宽索引的优势在于避免了前 20 行结果的表访问。
5. 尽管使用三星索引可能会得到一些好处，尤其是在结果集为空的情况下，但这一好处并不明显，并且这会引入由于添加了一个新索引而产生的额外负载（不过需要额外注意一下接下来的第 6 点）。
6. 如果 LEE 的索引片变得非常厚，一个三星索引可能更合适。不过存在一个上限，大约是 100 000 次 TS，这时扫描索引片将耗费太长时间以至我们需要创建一个新的索引。对于本例而言这个新的索引就是三星索引。

对本次分析中我们所考量的各种索引的总结如表 6.1 所示。

表 6.1　最差输入条件下的 QUBE 对比

类型	索引	LRT	维护
现有索引	LNAME, FNAME	100 s	—
半宽索引	LNAME, FNAME, CITY	0.2 s	U CITY + 10–20 ms
宽索引	LNAME, FNAME, CITY,CNO	0.1 s	U CITY + 10–20 ms
三星索引	LNAME, CITY, FNAME, CNO	0.01 s	I/D + 10 ms U + 10–20 ms

LRT是最差输入条件下的QUBE计算值；I = 插入，D = 删除，U = 更新

在之前的例子中，我们一直心照不宣地假定结果集为空的情况是导致性能最糟糕的场景。然而，我们应该意识到，一个几乎为空的结果集将导致更糟糕的性能，因为在使用半宽索引的情况下，这还将引起不超过 20 次的表访问。

练习

6.1. 图 6.5 中的 SELECT 查询在当前索引条件下需要花费 1 分钟。请用两种方案设计可能的最佳索引：（1）不增加额外的第三个索引，（2）增加第三个索引。

6.2. 假设 1 秒是我们可接受的执行时间，并且我们不愿增加一个新索引。对于上一题中的（1）方案，你需要了解哪些信息来预计它所能带来的性能提升？ 103

```
SELECT C, E, F
FROM   TABLE
WHERE  B = :B
         AND
       C > :C
         AND
       D = 1
ORDER BY C
```

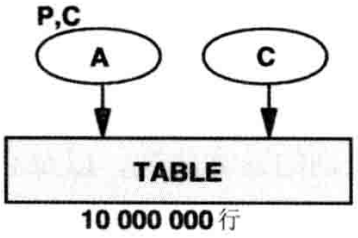


图 6.5　习题 6.1 和习题 6.2

第7章

被动式索引设计

- EXPLAIN 函数的角色以及如何在索引设计过程中使用它来描述被选中的访问路径
- 如何借助能够识别个体问题的尖刺报告，通过 LRT 级别的异常监视来识别事务的 SQL 问题
- 使用气泡图识别和分析问题制造者及受害者
- 辨别有优化空间的问题制造者和无优化空间的问题制造者，以关注于能够取得的主要潜在收益——调优的潜在空间
- 如何使用调用级别的异常监视，以及它与 LRT 级别异常监视的比较
- 发现慢的 SQL 调用
- 数据库管理系统特定的监视问题

简介

到目前为止，我们讨论的都是相对简单的 SQL 语句。在我们讨论如关联查询、子查询、合并查询、星型关联查询和多索引访问等更复杂的活动前，有必要先讨论一下如何监视 SQL，并从被动式响应的角度考虑索引设计中需要考虑的问题。

Amy Anderson 和 Michael Cain 生动地描述了被动式查询方法（6，第 26 页）：

> 被动式的方法与莱特兄弟创造首架飞机的经历非常相似。本质上就是把查询放在一起，推下悬崖，然后看它能否起飞。换句话说，就是为应用设计一个没有索引的原型，然后开始运行一些查询。又或者，创建原始索引集，然后通过运行应用来看哪些索引被用到、哪些没有被用到。即便是一个小型数据库系统，运行速度慢的查询也会很快凸显出来。
>
> 被动式调优方法也被用来理解和调优一个性能没有满足预期的已有应用。

在产品投入大规模使用后才对应用的大部分索引进行调优工作，就像是 106
将一个满载乘客的大型客机推下悬崖。这可能会有一个好的结果，但是数据库员工必须做好快速响应的准备。如果没有人在应用开发阶段关注索引设计的话，一些程序可能会在应用投入使用后变得非常慢。

当然，我们不推荐莱特兄弟的方法。我们认为应用索引设计需要在投入使用之前就被调优好。然而，即使对所有程序都进行了快速估算和快速 EXPLAIN 复查，一些性能异常还是会在投入使用后产生。估算永远都不是完美的，比如在估算最差情况下的过滤因子时就很容易忽略一些东西。

在本章中，我们将会讨论在应用投入生产环境后的最初几天需要用到的性能工具和技术。这些工具和技术对于之后能够轻松地调优也是有用的，以在用户觉察之前发现性能问题，或至少在问题变得令人无法忍受之前发现它们。在这种场景下，我们实际上是主动的而非被动的。

EXPLAIN 描述了所选择的访问路径

识别可疑的访问路径是相当容易的，尤其是当 EXPLAIN 的输出结果被存储在表中的情况下，因为这使得获取结果变得容易。这就是为什么对于优化器选择了可疑访问路的 SQL，分析过程通常从索引优化开始。通过 EXPLAIN 能够快速发现以下这些性能警示信号（我们之前研究过它们）。

全表扫描或全索引扫描

如果在系统升级前还未检查最可疑的访问路径，即扫描整个索引或表，那么现在是时候去做了。

对结果集排序

除了全表扫描和全索引扫描，结果集的排序就是最有用的警示信号了。引起排序的原因可能有以下两种：

1. 没有可使查询语句避免排序的索引。
2. 优化器所选择的访问路径包含了一次多余的排序。

对于第一种情况，排序可以通过优化索引来避免（参考第 4 章和第 5 章）。第二种情况将在第 14 章中讨论。

通常，排序是没有害处的。例如，某应用可能有一千个不同的查询语句，其中几百个在它们的访问路径中都有一次排序，这几百个语句中的 90%可能 107
都有非常出色的性能，排序对其响应时间只贡献了很不明显的一部分。因此，

大量错误的警告可能会使这种检查方法有些无用。

有许多数据库顾问将排序视为敌人。我们认为，那些强调随机 I/O 带来致命影响的顾问更值得信任。

成本估算

一些数据库管理系统的EXPLAIN功能显示了优化器对所选访问路径的本地响应时间的估算，或至少显示了对 CPU 时间的估算。有些产品，如 SQL Server 2000，显示了对访问路径中每一步的 CPU 及 I/O 时间估值。在一个简单的场景中，可能会有如下几步：

1. 检索并读取索引行。
2. 对指针进行排序。
3. 检索并读取表行。
4. 对结果行进行排序。

经验显示，过分依赖优化器的成本估算是危险的。毕竟优化器生成成本估算仅仅是为了选择最快的访问路径。优化器生成的估值有时会给出错误的警示信号，而且还不会显示所有的警示信号，但这是一个用起来简单且快速的工具，它使得早期检查成为可能。因此，那些成本估值异常高的 SQL 语句——可能是一个新应用中估值最高的前 20 个语句——应该被检查一下，很可能不合适的索引使用或优化器问题就能被发现。

不幸的是，以下两个严重问题限制了使用成本估算方法的价值：

1. 优化器所做出的本地响应时间估算可能与实际值相差很大（参考第 14 章）。
2. 当谓词使用绑定变量时（显然这是很普遍的），优化器对过滤因子的估算是基于平均输入值的，或更差情况下，基于默认值。为了获取更有价值的最差情况估值，EXPLAIN 中的绑定变量必须用最差情况下的输入值来替代。这是一个需要应用知识的累人操作。

EXPLAIN 是分析优化器问题的一个不可或缺的工具。因为这正是它存在的理由，而不仅仅被用于索引设计。就像机场安检人员不仅是在机场检查可疑乘客。

108 数据库管理系统特定的 EXPLAIN 选项及限制

许多数据库管理系统的EXPLAIN功能在过去二十年中有了很明显的增

强。除了每行一个执行步骤的紧凑型报告之外，还提供了会话级别的图表。可疑的访问路径甚至会在一些 EXPLAIN 的输出中被标记出来。对于每个所访问的表，优化器决策所基于的统计信息，连同当前的索引列表及其特征，都会在报告中提供。这是极其有用的。如果这些内容没有在报告中给出的话，就必须从系统表中分别获取这部分信息。最好的 EXPLAIN 还集成了另外一个有价值的步骤：提供索引优化建议。

另一方面，目前的 EXPLAIN 仍存在一些恼人的缺点。z/OS V7 上运行的 DB2 数据库不报告索引筛选信息（通过比对那些被复制到索引行上的列值来排除一行）。虽然这个缺点不直接影响索引检查，但是这意味着一些优化器问题只能通过测量表访问的次数来揭露。Oracle 9i 数据库不报告参与索引筛选的索引列个数。实际上，它甚至不报告匹配列的个数（Oracle 9i 中 PLAN_TABLE 表的 SEARCH_COLUMN 列目前未被使用）。但 Oracle 9i 会报告索引扫描的类型：唯一扫描（一行）、范围扫描、全索引扫描或快速全索引扫描。如此一来，最可疑的访问路径就能被揭露，但 MC 值的缺乏使得分析那些由复杂谓词（这在 Oracle 术语中有时被称为*索引禁用*）导致的优化器问题变得更困难。

监视揭示现实

幸运的是，现在大部分数据库管理系统都提供记录了本地响应时间各个组成部分的跟踪信息，或者至少提供事件发生的次数信息，如物理读事件。这种跟踪活动所需的开销，曾一度有些负担不起，但现在已趋于可接受：比如在 z/OS V7 上运行的 DB2 中，这种跟踪活动在每次 SQL 调用中只花费几毫秒 CPU 时间。对于一次典型的操作型事务，这意味着 CPU 时间有几个百分点的增长，对于总体耗费时间则意味着更小比例的增长。然而，对于一个几乎只发起简单读取调用并引起顺序扫描的批量任务来说，耗费时间可能会增长 20%之多。

相较 EXPLAIN 来说，测量是分析一个慢程序的更好的着手点，它除了包含访问路径问题外，也包含所有其他可能导致长运算时间的原因。然而，看监视报告可能会花费大量的时间，尤其是在没有实现一个好的监视策略的情况下。

测量一个计算机系统的性能是一件复杂的任务——因为这可以从许多不同的角度来考虑。本章后面介绍的简易尖刺报告是一个长期演进的产物。对其历史的了解将帮助我们更容易地理解该方法的实质。

109 性能监视器的演进

20 世纪 60 年代

最初的监视器有两个主要目标：基于耗费的 CPU 时间进行计费，以及监视硬件负载。这时候的监视器通常是操作系统的一部分。类似 CPU 繁忙度、信道繁忙度及卷繁忙度这些负载因子都是通过采样测量的。

至于调优，最有用的报告是提供磁盘 I/O 响应时间的报告，这种报告还包含了按磁盘卷分解的磁盘 I/O 时间各个组成部分的信息。文件经常被从一个卷移动到另一个卷以平衡各个磁盘的负载。在极端情况下，活跃的文件甚至会被移动到相互靠近的区域以降低平均寻道时间。

20 世纪 70 年代

在 20 世纪 70 年代，人们对最初的数据库管理系统进行了增强，使其能够通过程序对耗费时间进行监视。这时，跟踪平均本地响应时间的走势以及识别慢程序就成为了可能。例如，IMS 提供了一个能够通过 DL/I 调用及返回码展示耗费时间的监视报告。尽管这个报告可能会很长，但通过它能容易地发现那些异常慢的调用。最常见的导致运行慢的原因是不正确地划定根寻址区大小，该区域包含了随机发生器的根锚点。通过这个报告可以发现许多程序错误，比如，一个 DL/I 调用可能使用了错误的索引。在前关系型数据库管理系统中还不存在优化器。

20 世纪 80 年代

当人们开始转用关系型数据库时，调优的重点转向了优化器。EXPLAIN 是调优过程中最重要的工具。虽然性能优化课程也强调详细的 SQL 跟踪报告的重要性，但是使用者很快发现，他们只需使用类似 DB2 Accounting Trace 这样的报告检查一下程序计数器和响应时间的组成部分，就能解决几乎所有的性能问题。

随着数据库管理系统供应商在报告中加入越来越多的跟踪记录，这些性能报告变得越来越长，且开始看起来有些复杂了。与此同时，应用也比以前变得更为复杂。阅读监视报告变得过于耗时了。数据库管理系统供应商和第三方公司（专业开发性能监视和性能工具的公司）都投入了大量资源用于开发对用户更友好的总结报告、异常报告和图表。实时监视在许多生产环境中变得十分流行，因为它们显示了阻塞以及其他系统级问题的原因。然而，基于长期的测量数据，批量报告被认为是发现重大应用性能问题的唯一可靠的方式。在最流行的报告中，有一个报告提供了按程序名或事务代码统计的关键指标的平均值。尽管报告可能多达几百页，但是那些持续运行缓慢的程序

很容易就能被发现。如果 CPU 时间或磁盘读取量较高，那么 EXPLAIN 会显示出可疑的查询语句。早在 20 世纪 80 年代，不合适的索引就已经是导致性能低下的最主要原因了，紧接其后的第二大原因就是优化器问题。虽然那时的优化器不如现今的这么智能，但是导致错误的访问路径选择（比如在有一个完美索引的情况下选择了全表扫描）的一个更重要的原因是人们普遍缺乏对于优化器的正确理解。在 20 世纪 90 年代之前，优化器相关的注意事项列表在性能调优指南及课程资料中都不常见。 110

20 世纪 90 年代

到 20 世纪 90 年代，应用变得愈发复杂的同时，用户的忍耐度也变得越来越低。许多用户不再满足于不到一秒的平均响应时间，他们觉得没有哪个交互型事务应该花去几秒时间。举个例子，一个申请座位预定系统的客户坚持认为，在特定的高峰负载下，每一次单笔事务的响应时间都必须少于 5 秒。在产品升级以后，只有在大量压力测试和测量中证明能达到这一性能要求之后，系统才会被接受。

此时，异常报告就变得有必要了。人们的关注点从事务维度的均值转向了单次慢事务的概况。在 20 世纪 90 年代，许多性能监视系统都包含了这个选项，异常的判断标准由生产系统来定义。但是由此产生的每一个异常事务的概况报告可能会多达 10 页，且包含数以千计的数字。

在这种情况下，下一步自然就是将 10 页的报告压缩成一个汇总报告，其中每个事务只用一行来描述，从原始的长报告中选择最有价值的数字和比例进行展示。在芬兰，首个这样的报告是在 20 世纪 90 年代中期被实现的，称为*尖刺报告*。有了这种方式的报告之后，我们就能把测量周期从原先的分钟级延长至小时级，用以生成统计意义上更可靠的结果。如此一来，我们便能通过报告自信地指出最普遍的问题了。从另一个角度来说，即便是偶尔执行的低性能程序也能被这种方式发现。而且，这还大大降低了分析的工作量。当一个事务的表现情况被展示在一行中时，理解这些信息就变得很容易了。一个根据事务代码和结束时间排序的尖刺报告很快便能显示出那些经常运行得很慢的程序。如果每天都生成这种报告，那么就能容易地看出程序从什么时候开始变慢，以及什么时候通过类似辅助优化器或向索引添加列等调优措施解决了这个问题。一个根据结束时间排序的尖刺报告对于排查长时间锁等待，或分析在数据库同步读期间磁盘进行集中写入所带来的影响都很有价值。

除了显示慢事务的尖刺报告外，查看最慢的单次 SQL 调用也是有用的。针对此目的的监视器在 20 世纪 90 年代进入了市场。

111 2000 年至 2005 年[1]

如果没有人花时间去分析并做出正确的结论，那么尖刺报告就几乎没有多少价值。在这几年期间，可能是受到全球性 DBA 严重短缺的影响，对于有自我调优能力的数据库管理系统的需求变得越来越明显。为了实现这个宏伟的目标，最先发展出的是能够对表及索引的组织方式或缓冲池大小提出建议的工具。在不久的将来可能会出现能对优化器更新统计信息提出建议的工具，该工具可能还会包含对每个索引及表统计信息的推荐值（如直方图）。此类工具的下一步发展可能是基于运行时所测量的过滤因数，对访问路径进行再评估，从而用改进后的访问路径进行再次处理。这也是聪明的人在许多情况下所选择的方式。

一些 DB2 生产系统中所使用的尖刺报告能将人们的注意力吸引至可能的访问路径问题制造者或锁等待牺牲者上去。这些指示器有助于忙碌的数据库专家快速方便地发现新的性能问题，例如由程序维护相关的变化、数据库增长或使用模式的变化所引起的新性能问题。

LRT 级别的异常监视

程序粒度的均值是不够的

一个展示每个程序平均响应时间的报告所揭示的是那些长时间运行较慢的程序。然而，均值的用处是有限的，因为一个程序通常都包含不同的函数。即便是同一个函数，执行时间也会由于输入的不同而相差很大，比如这是对于平常客户的操作还是对于大客户的操作。

对于单个事务的监视应该尽早开始，最好在测试阶段就开始，用以捕捉最差输入下的响应时间。而且，分析单个事务的概况比分析同一个程序下许多不同事务的平均概况要简单得多，后者的复杂度就好比让我们比较一些卡车和许多自行车的平均特征。

异常报告举例：每个尖刺一行

以下所展示的就是我们之前推荐的每行一个异常事务的异常报告，即尖刺报告。在本章中，我们将使用术语尖刺来代表被异常监视工具监测到的一

[1] 2006 年至今的情况补充如下。近些年来，数据库管理系统的自动调优能力又有了进一步的发展，在一些系统中已经有了能对优化器重新收集统计信息提出建议的工具，该工具还能提供其他许多建议，比如修改 SQL 语句结构的建议、创建索引的建议、硬件配置调整的建议等。此外，有些工具还能够通过分析找到更优的访问路径，并自动用其替换旧的访问路径——译者注

次事务。可能相同的事务在其他的多次运行中性能表现都令人非常满意。

标识

- 程序名
- 主要模块（对 LRT 贡献最大）的名称
- 主要模块对 LRT 的贡献（%）
- 日期
- 结束时间（hh.mm.ss）

112

事务概述

- 本地响应时间（s）　LRT
- SQL 调用的总时间（s）　SQL
- SQL 调用的 CPU 时间（s）　CPU 时间
- 同步读时间（s）　同步读
- 等待预读取的时间（s）　预读等待
- 锁等待时间（s）　锁等待
- 其他等待时间（s）　其他等待
- 平均同步读时间（ms）
- 同步读次数（表页）
- 同步读次数（索引页）
- SQL 调用次数
- 访问的表页数量
- 访问的索引页数量
- 顺序预读取请求数
- 提交次数
- 快速诊断（由以上数字得出）

事务概述中前 7 项右边突出显示的术语（如 LRT）对应于本章中气泡图中的术语。

图 7.1 展示了一个会被异常监视器识别到的事务的事务概况构成信息。通过这种方式展示异常报告中的信息使我们能够很容易地发现任何问题。

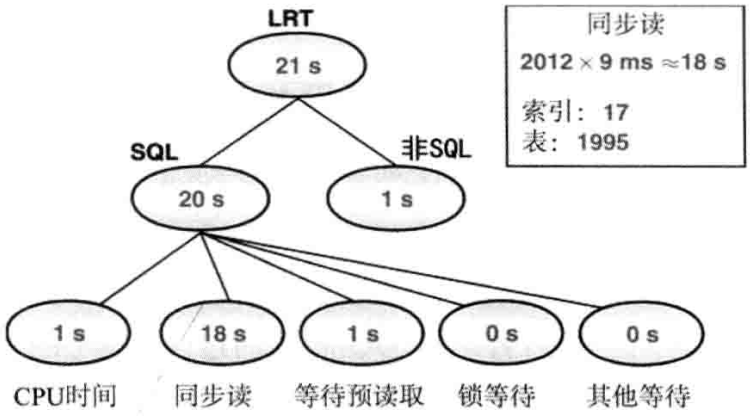


图 7.1　一个尖刺的例子

对这些数字的含义、解释及任何随之而来的调优将在后面讨论。

问题制造者和受害者

一个涵盖一周中高峰时段的尖刺报告可能包含 100 多个不同领域模块的数千个尖刺。在应用开发者的些许帮助下，一个数据库专家也许每周能够分析并修复 5 至 10 个模块。由于程序变更或不同的用户输入，下一周的尖刺报告可能会显示新的问题模块。为了高效地利用有限的时间，仔细地对所分
113 析的模块排定优先级是很重要的，事务的响应时间及执行频率并不是唯一的优先级评定标准。

如图 7.2 所示，我们首先需要区分的是问题的制造者及受害者。如果一个事务独占了资源（也许是因为使用了不合适的索引），那么就会对其他事务造成明显的负面影响，进而导致这些事务也与独占资源的事务一同出现在异常报告中。

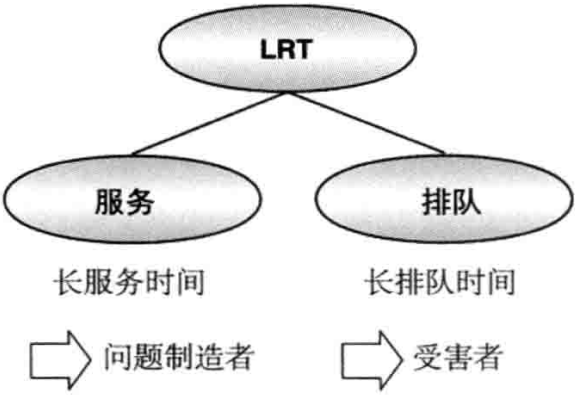


图 7.2　问题制造者和受害者

问题制造者所使用的资源会被包含进服务时间，而问题受害者会因为需要等待这些资源而受到影响，即排队时间受影响。

着手调优的合理方式是考虑如何解决问题制造者导致的问题，而非受害者的问题。

114 有优化空间的问题制造者和无优化空间的问题制造者

在决定将注意力从受害者转移到问题制造者上之后，我们需要区分的第二项内容就是有优化空间的问题制造者和无优化空间的问题制造者，如图 7.3 所示，不过我们可能没有足够的时间去分析所有的问题制造者。有优化空间的问题制造者是指那些能够通过改进索引来获得大幅性能提升的事务。

有优化空间的问题制造者有两个特征：

1. 磁盘服务时间长。

2. 磁盘读大多是对表页的读取。

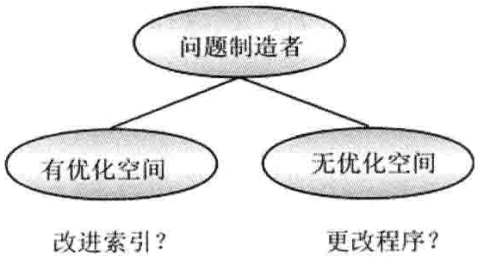


图 7.3　有优化空间的问题制造者和无优化空间的问题制造者

若在同步读上耗费的时间占了本地响应时间的绝大部分，且有超过 100 个表页是从磁盘服务器上读取的，那么索引改进很可能会带来不错的效果。同样，如果对于索引页的同步读大于 100 次也不是一个好消息。

为了减少对索引的随机读次数，可以减少所需访问的索引数量，也可以减少所需访问的表的数量，这可以通过向表添加冗余字段的方式实现。这些改变并不比创建一个新索引或向现有索引上添加列更容易或可控。

关于随机索引访问有一个例外，就是对一个厚索引片的扫描可能会由于叶子页的分裂而变慢。这个问题很容易解决也相对容易避免，我们将在第 11 章中讨论。

有优化空间的问题制造者

图 7.4 展示了一个由于研发阶段未被发现的不合适索引而可能产生的尖刺。如果在系统投入使用时只对最可疑的访问路径（全表或全索引扫描）进行了检查，并且开发人员在编写 SELECT 语句时连基础问题（BQ）都没有应用过，那么这种情况就有可能发生。

即便对应用没有任何了解，我们也能很快确定图 7.4 中的问题制造者是有优化空间的。我们得到以下几个主要的观察结果：

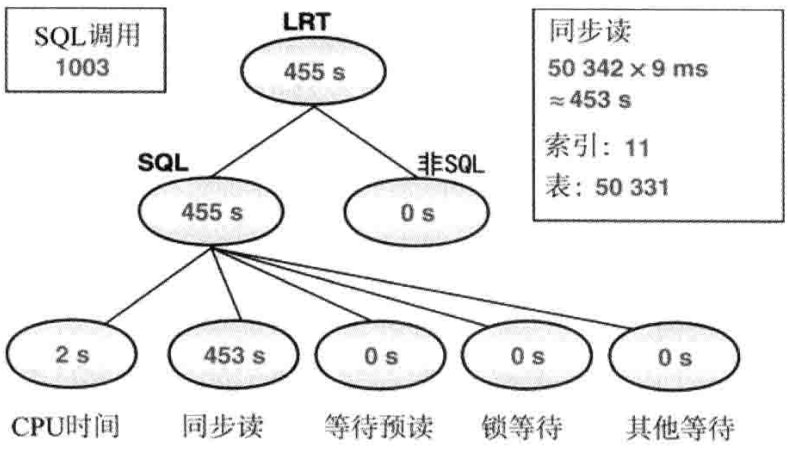


图 7.4　有优化空间的问题制造者

115

- 同步读（453 s）是导致过长本地响应时间（455 s）的主要原因。
- 一次同步读的平均时间是相当短的（9 ms）；磁盘排队时间可能并不明显。
- 几乎所有的同步读所访问的都是表页（50 331）。

一次同步读的期望平均时长依赖于读缓存的命中率。如果读缓存比数据库缓冲池要大得多，如100∶1，那么缓冲读和磁盘读的期望均值可以低至3 ms。假设本例（9 ms 的同步读时间）满足这种情况，那么磁盘排队时间将会是很明显的，不过减少同步读的次数还是能够改变许多的。如果磁盘已经严重超负荷了，可能是由大量的集中写入导致的，那么一个事物的平均同步读耗时可能会达到几百 ms。这样一来，该事务显然就是一个受害者了。

分析有优化空间的问题制造者

一个有优化空间的问题制造者通常会读取许多表页。我们知道有三种读取表页的方式：

1. 事务中的随机读（同步读，SR）
2. 跳跃式顺序读
3. 顺序读（顺序预读）

通过分析尖刺报告，我们能够推断出有优化空间的问题制造者主要是用这三种方式中的哪种方式读取表页的。

116

同步读（方式 1）不与 CPU 时间重叠，应用程序会停下来，等待所请求的页从磁盘服务器返回。尖刺报告会将该事务同步读取的页数连同这些同步读的平均等待时间一起显示出来。

异步读（方式 2 和方式 3）是预读取，它的 I/O 时间与程序所花费的 CPU 时间是重叠的。在数据库管理系统向磁盘服务器请求后几页的同时，程序仍然在处理数据库缓冲池中的页。程序可能会遇到没有页可供处理的情况，也可能不会遇到。如果发生了这种情况，那么这个等待时间会被记录进预读等待的计数器中。如果程序从未遇到需要等待预读取页的情况，那么 CPU 时间便决定了程序的性能。于是，在这种情况下，预读等待的值为零。顺序扫描所花费的时间一部分被记录进 SQL 的 CPU 时间，一部分被记录进其他等待（对于任何 CPU 排队来说）。

在当前的硬件条件下，以跳跃式顺序读（方式 2）的方式访问表页时，响应时间可能还是取决于所花费的I/O时间。如果大部分的跳跃跨度都很大，那么预读等待可能是本地响应时间中最主要的组成部分，因为这意味着包含

满足条件的页彼此相隔很远。

而以顺序读的方式（方式 3）访问表页时，在当前硬件条件下，CPU 或 I/O 时间的主导地位基本各占一半，有时访问每页所花费的 CPU 时间比 I/O 时间长一些，有时又短一些。QUBE 算法假设，I/O 时间与 CPU 时间中较大者的上限为 0.01 ms 每行。如果 I/O 时间决定了顺序读的时间，那么顺序扫描所花费的时间就等于磁盘 I/O 的时间，反过来，则是顺序扫描所花费的时间等于尖刺报告中 CPU 时间与预读等待部分的总和。

许多程序使用上述方式中的两种或三种来访问表页。此时理解尖刺报告就没有那么简单了，不过主要的方法通常也不是那么难推理。访问的表页总数也许能够帮助我们理解报告。访问的表页总数除了同步读次数外，还包括了通过预读读取的页数（方式 2 和方式 3）和缓冲池的命中数（在数据库缓冲池中被找到的表页数量）。

目前许多数据库管理系统能进行并行的顺序操作，比如将一个指针拆分成几个内部指针，每个指针使用一个处理器并创建自己单独的预读取流。如果开启了这个选项（CPU 及 I/O 并行机制），实际的并行度通常是由优化器来决定的。如果有足够的处理器、磁盘服务器和缓冲空间，那么并行级别可能达到 10。理论上讲，这样所耗费的时间可能低至无并行情况下所耗费时间的 10%。当扫描大量表及其索引时，如数据仓库的表，这个选项是极其有价值的。但是大部分运行传统事务和批量任务的生产环境都关闭了并行，因为一个同时占用了几个处理器和磁盘驱动的程序可能会引起其他并发用户无法接受的排队时间。

调优的潜在空间

在投入大量时间分析一个异常事务之前，值得花时间先去评估一下在索引优化之后本地响应时间能降低多少。调优的潜在空间就是指可实现的降低值的上限。 117

随机读

只读事务的调优潜在空间等于同步读取的表页数和同步读的平均时长的乘积。如果通过使用宽索引避免全部表页的读取，那么本地响应时间就能减少这个量。由此，若图 7.5 中的程序是只读的，则其调优的潜在空间为 50 331 × 9 ms≈453 s。包含插入、更新或删除操作的程序的调优潜在空间较小，因为无论索引多宽，表页都必须被更新。

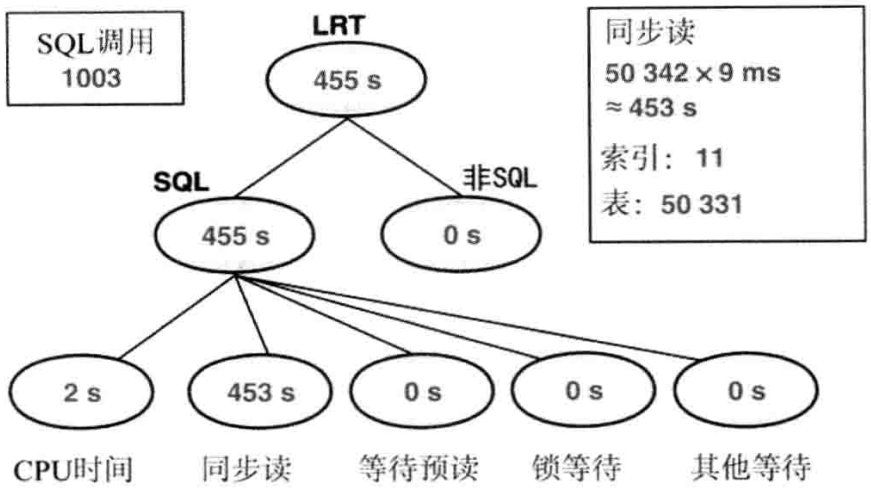


图 7.5　有优化空间的问题制造者——随机读

跳跃式顺序读

在上述例子中，有些优化器可能会选择跳跃式顺序读。那么数据库管理系统将首先从所有满足条件的索引行中获取指针，然后再根据表页号对指针进行排序。此时，数据库管理系统就有了所有表页排序后的列表，每个表页都至少包含一条符合条件的记录。只有到这时，数据库管理系统才会开始访问表。在这种方式下，访问每个表页的 I/O 时间自然相对较短，尤其是当一些满足条件的行恰好都在同一个磁轨上时。另外，I/O 时间将会与 CPU 时间重叠，因为数据库管理系统有条件进行预读了。相比同步读，这种方式所节省的时间是非常不确定的，具体取决于读取的页之间的平均距离以及条带的实现方式。这种方式的调优潜在空间即为当前预读等待的值（假设程序是只读的，且所有预读取都是跳跃式顺序读的，如在 z/OS 系统上运行的 DB2 中使用的列表预读）。由此可知，图 7.6 中事务的调优潜在空间为 102 s。

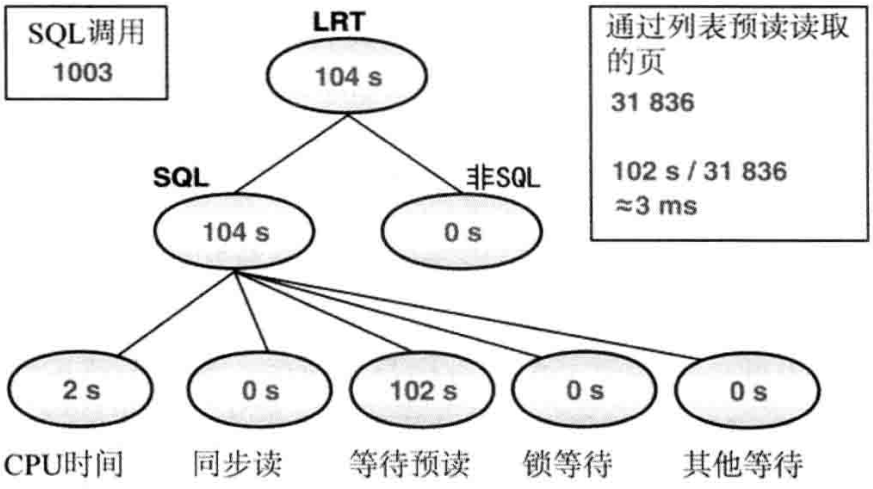


图 7.6　有优化空间的问题制造者——跳跃式顺序读

顺序读

118

图 7.7 中的尖刺报告展示了一个扫描全表的应用的情况（通过顺序预读方式读取了 112 721 个页）。由于没有 I/O 等待时间，程序的性能取决于 CPU 时间。图中 2 s 的其他等待可能是由 CPU 排队引起的。

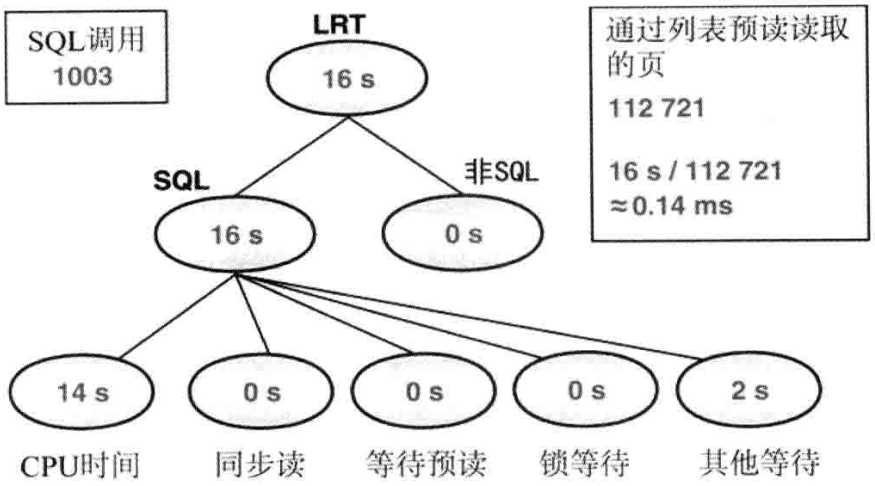


图 7.7　有优化空间的问题制造者——顺序读

图 7.7 中的尖刺乍看起来像是一个无优化空间的问题制造者。我们很容易能想出即便在好的访问路径下 1000 次调用花去 16 秒的例子，例如，1000 次单独查询导致平均每次查询需要两次随机读。然而，在图 7.7 中，CPU 时间（而非 I/O 时间）是响应时间最大的组成部分。这显示出 SQL 处理过程几乎是顺序的。顺序扫描的响应时间通常取决于 CPU 时间。而对于随机读，只有在所有页都能通过内存访问到的情况下，其响应时间才取决于 CPU。

119

另外，报告显示 100 000 个左右的页是通过顺序预读的方式读取的。这些页可能是叶子页，也可能是表页，又或者两者都有。有可能程序读取了一个非常宽的索引段和一个非常宽的表段，且索引及表行的顺序是一致的。然而，大部分行都不满足条件被拒绝了。如果我们假设一次顺序读花费大约 10 μs（这将在第 15 章中详细讨论），且随机访问的量比顺序访问的量少得多，那么总的访问次数大约为

$$\frac{14\ 000\ 000\ \mu s}{10\ \mu s/次} = 1\ 400\ 000次$$

通过这种方式，我们能很容易地找出那些造成大量顺序扫描的查询，这些查询可能扫描了 100 000 多页，而且可能扫描了 1 000 000 多行。

于是，此类问题的调优潜在空间大约为 16 s（假设 2 s 的其他等待是由 CPU 排队引起的）。我们应该能够找到一个读取更窄的索引段或表段的访问路径，以达到调优的目的。

如果该慢访问路径包含全索引扫描或全表扫描，那么难道这在快速EXPLAIN复查过程中没有被发现？情况并不一定如此。因为访问路径可能是用一个范围谓词作为匹配列来使用索引的。当输入的值为最差输入时，实际被扫描的索引片可能就是整个索引了。图7.8展示了此类情况的一个例子。

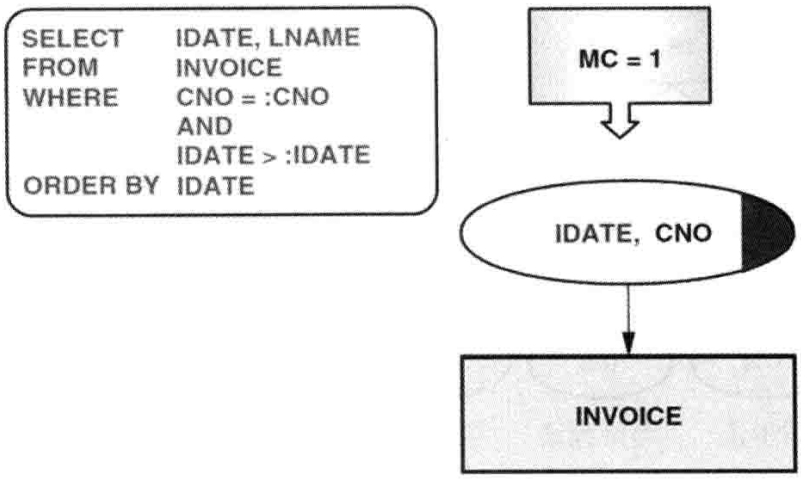


图7.8　即便MC = 1，也可能导致扫描整个索引

如果优化器假设谓词IDATE > :IDATE是一个较小的过滤因子，那么它可能会选择索引(IDATE,CNO)。由于IDATE是唯一的匹配列（因为第一列是范围谓词，所以CNO不能参与匹配过程），因此如果用户输入了一个很久以前的日期，那么数据库管理系统可能就会扫描整个索引。

120 无优化空间的问题制造者

图7.9所描述的尖刺表明这是一个响应时间主要取决于CPU的顺序读操作（只有4 s的预读等待）。大量的SQL调用（935 206次）是一个坏消息。理想情况下，这个数量的SQL调用可能意味着相近数量的待处理行数，每行的处理花销为0.1 ms（没有行被拒绝的前提下）。在这种情况下，时间成本(将顺序访问的成本包含在读取过程中)将会是935 206 × 0.1 ms≈93 s。实际上，93 s比测量出的53 s CPU时间还长，显然这个查询的评估标准与0.1 ms的性能原则所基于的标准有些不同。虽然如此，53 s仍是这些SQL调用的最小时间花费了，我们没办法让它们变得更快。因此，唯一降低SQL CPU时间的方法就是减少SQL调用的数量。这样一来顺序读的响应时间可能就会从取决于CPU时间变为取决于I/O时间，所以即便SQL调用量下降90%，响应时间也可能降低不了90%。

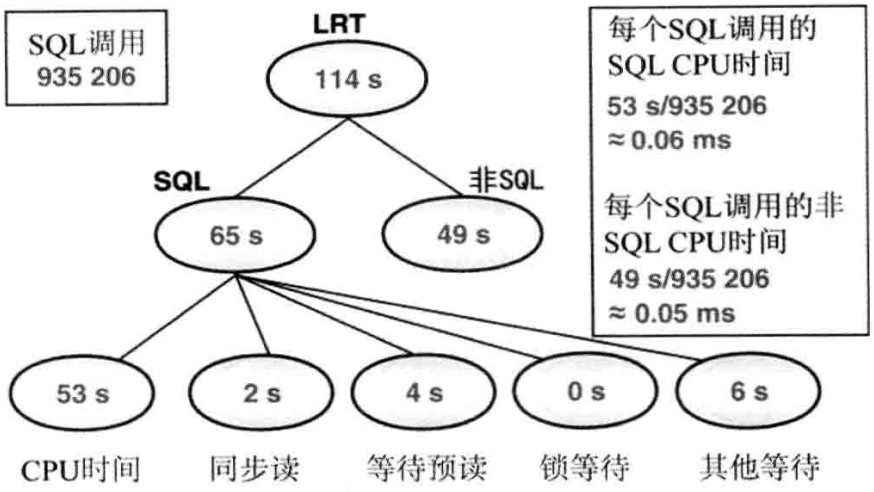


图 7.9　无优化空间的问题制造者

很大的 SQL 调用量也解释了大量的非 SQL 时间（49 s）。在许多平台上，每个 SQL 调用的一部分时间被报告为非 SQL 时间。相邻两次 SQL 调用之间的 50 μs（49 s/935 206）很可能是由于这种消耗（发送一次 SQL 调用以及从应用程序接受相应结果）引起的。相较于此，本例中用 PL/I 编写的应用程序本身所花费的 CPU 时间要少得多。

实际中发现过一个类似这样的尖刺，是由某个用户发起的每日一次的数据库完整性检查。对于编程者来说，能够通过在 where 分句中添加谓词来减少 SQL 调用量。但这样做是不值得的，因为作为该事务的唯一用户，较长的响应时间并不是问题。为了避免给其他同时运行的事务带来不必要的排队时间，我们决定该事务应该在凌晨运行！ 121

通常来说，对有大量 SQL 调用（如 10 000 多次）的事务来说，如果 CPU 时间（SQL CPU 时间及非 SQL 时间）是本地响应时间的主要组成部分，那么这些事务是很难被优化的。

对于此类事务，调优潜在空间即为能够去除的 SQL 调用次数乘以每次 SQL 调用的基本 CPU 时间（如 0.1 ms）。

受害者

图 7.10 展示了一个导致尖刺的小表。该表中包含了一些用于收集统计数据的计数器。这些行被许多事务更新。这些更新操作通常在程序的较早阶段就被执行了（通常都是不必要的早），也就是说，这些更新不是在接近提交点的时候执行的。有时更新这些计数器的事务可能会由于磁盘排队等原因运行较慢。结果，某行数据就会被锁上几秒钟，导致其他需要更新该行的事务不得不进行等待。

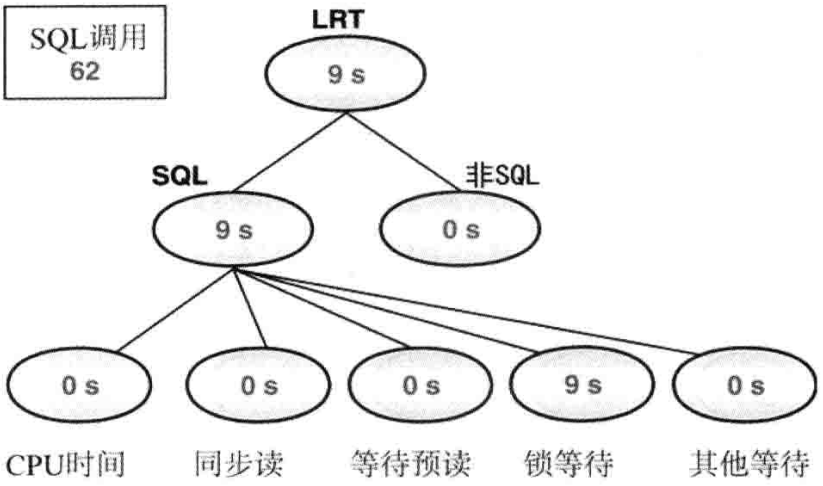


图 7.10　受害者

不同数据库管理系统在锁的实现上差异很大，但是锁等待一般都是应用设计的问题。首先要做的是找出热点资源——造成锁等待的行或页。如果数据库管理系统提供了锁等待的跟踪功能，那么就很容易对受锁等待影响的程
122 序执行一次跟踪，来定位热点资源。另外一个替代的方法是去判断这个受长时间锁等待影响的组件访问了哪些表。假设索引页或索引行没有被数据库管理系统锁住，那么热点资源必定是这些表中某一个表的页或行——可能是一个小表中访问最频繁的行，或者是一个大表中最后的表页（在新行都插入到表的末尾的情况下）。

基础的排队理论在分析锁等待问题上很有帮助。

$$Q = u/(1 - u)S$$

其中 Q = 平均排队时间

u = 利用率（资源繁忙度）

S = 平均服务时长（每次请求引起的资源繁忙度）

这个简单公式假设服务器数量为一台，且请求到达时间和服务时间满足泊松分布。

在数据库锁定过程中，一个对象的繁忙（锁定）时间等于所请求的频率与平均锁定时间的乘积：

$$\text{每秒 2 次请求} \times \text{每次请求 0.2 s} = 0.4$$

经验显示，资源繁忙度（u）的告警阀值不应大于 0.1，因为这意味着在高峰期对象有 10%的时间是被锁住的。这样一来，平均排队时间将会是资源锁定时间的 $u/(1 - u) \approx 11\%$。由于变化的随机性，某些个别请求的排队时间可能会比平均排队时间长得多，显然这种特性很可能会导致明显的问题。

如果一个对象 50%的时间是被锁住的，那么公式显示平均锁等待时间将会与锁定时间相等。某些锁等待时间将会比均值高很多，在这种情况下，问题很可能会变得非常严重。

索引的改进能够减少锁定时间（S），从而减少锁等待时间（Q）。这就是为什么我们建议从分析有优化空间的问题制造者开始调优，分析锁的牺牲者应该是下一步。一旦找到瓶颈，锁的问题通常相对来说就容易解决了。

其他牺牲者，如 CPU 时间较长的事务或磁盘排队时间较长的事务，通常较难优化。长时间的排队意味着较高的利用率。假设已对主要的问题制造者进行了调优，那么唯一的解决方式可能只有硬件升级。那些需要等待打开或扩展文件等事件的事务也被归类为牺牲者。类似这些延迟通常是系统级调优的问题。对于不常用的文件，这些问题可能无法避免，或者如果周末系统被重启过（文件被关闭），周一早上也会发生此类无法避免的问题。

这些不常见的等待被记录进其他等待时间（CPU 排队时间也被包含在内）。更详细的异常报告可能会识别出这些事件或等待的类型。基于这个原因，保留一至两周的详细报告是明智的。

对于锁的牺牲者来说，潜在的调优空间等于锁等待的时间，对于图 7.10 中的例子，潜在的调优空间为 9 s。对于其他类型牺牲者来说，潜在的调优空间等于其他等待时间中相关部分的时间。 123

查找慢的 SQL 调用

在识别出 SQL 执行时间的主要组成部分后（SQL CPU 时间、同步读、预读等待、锁等待或其他等待），接下来的一步是找出对该部分贡献最多的那些 SQL 调用（我们应该意识到，气泡图中所示的数字很可能涉及许多不同的 SQL 调用）。可以简单地通过阅读主要模块的 SQL 语句来找出这些 SQL 调用。如果问题是由过多的 CPU 时间或 I/O 时间导致的，那么许多语句能被很快地排除，如那些通过主键（WHERE PK = :PK）访问一行的语句。另外一个方法是对该程序进行一次 SQL 跟踪，如果数据库管理系统提供该功能的话。跟踪报告展示了每个 SQL 语句的耗费时间及其主要组成部分。

最普遍被使用的（且负担得起的）跟踪是模块级别的。在这种情况下，模块越小就越容易识别慢 SQL 调用。从调优的角度来看，一个内置大模块的应用是一个噩梦。另一个极端情况是，一些模块可能只包含一个 SQL 语句。于是，对本地响应时间贡献显著的 SQL 语句就能直接从异常报告中被识别出了。

识别出有问题的 SQL 调用后，如果发现问题是由不合适的索引导致的，我们就需要根据第 4 章和第 5 章中所述的技术对其进行改进。

调用级别的异常监视

除了 LRT 级别的异常监视之外，最有用的东西就是一个监视时间段内最慢 SQL 调用的报告了。这些报告对于在一个有大量不同 SQL 语句的慢程序中找出较慢的 SQL 调用很有帮助。

假设我们执行了一次高峰时段的跟踪，且基于该时间段内耗费时间的长短生成了最差 SQL 调用列表。现在，我们将着手处理单次调用而非均值。进一步假设我们所使用的工具对每次调用只提供了 4 个重要测算，它们是：耗费时间（ET）、CPU 时间（CPU）、从磁盘读取的页数（READS）及数据库页的请求数（PR）。数据库页请求在 DB2 中被称为读取页，在 SQL Server 中被称为逻辑读，在 Oracle 中被称为读取或 LIO。为了保持所有事物尽可能简单，我们使用一个能反映所有这些含义的简单术语——页请求（PR），参考图 7.11。

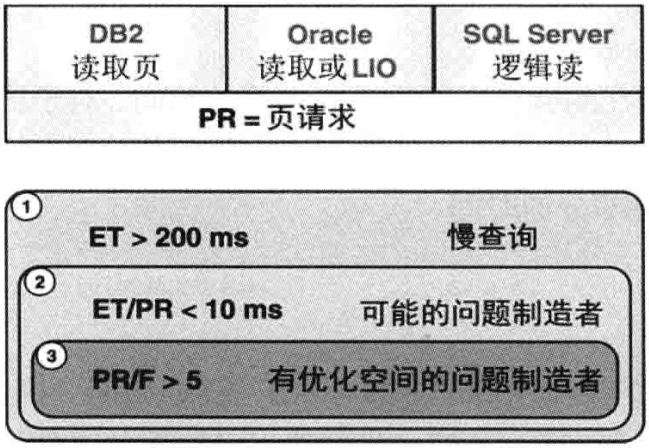


图 7.11　过滤运行较慢的 SQL 调用

124

从耗费时间（ET）最长的 SQL 调用开始，为了通过访问路径调优的方式改进响应时间，我们同样应该试着回答之前在 LRT 级别讨论过的两个问题。

首先，该 SQL 调用看起来是不是问题制造者？换句话说，过长的耗费时间是不是由于服务时间过长导致的？有几个方法可以回答这个问题。较高的 CPU 时间或较高的 READS 值都意味着较长的服务时间，但是，实际上，PR 似乎是服务时间最有用的指示器，因为它与 CPU 和 READS 都相关，而且在很大程度上是经得起反复检验的，它不受系统负载或预热的影响。

为了识别出那些可能是问题制造者的 SQL 调用，我们应该忽略那些 PR 值低的调用。如果一个查询耗费了 1 s 且发送了少于 100 次页请求，那么它可能就是一个受害者。在可能的最差情况下，每一次页请求都会造成一次随

机读，如果磁盘排队情况不过分的话，这将花去 100 × 10 ms = 1 s。而在实际情况中，许多页请求在数据库缓冲池中就会被满足，另一些则会在磁盘服务器的读缓存中被满足。请记住，报告的 PR 值包含了非叶子索引页，另外，许多页请求是顺序的。100 个页请求的期望 I/O 时间远比 1 s 短得多，CPU 时间的期望值也是如此。总而言之，如果一个包含 100 次页请求的 SQL 调用被测得的耗费时间是 1 s，那么很可能它所耗费的排队时间（磁盘驱动排队、CPU 排队、锁等待或其他等待）比服务时间长。通常，如果一个慢查询所耗费的时间比“页请求数×10 ms”要多，那么这个查询很可能是受害者。它属于图 7.11 中的子集①，但不属于子集②。

子集②是可能的问题制造者，它包含了那些 ET/PR 时间不是很长的 SQL 调用，它们不是很可能的或者明显的受害者。然而，子集②可能也包含了许多受害者，比如进行顺序操作的 SQL 调用（在每页上所耗费的服务时间较短）会由于过度的 CPU 排序时间而变得较慢。如果我们像对待 LRT 级别的尖刺报告那样关注所有的锁等待或磁盘、CPU 排队时间，那么我们是能够过滤掉这些受害者中的绝大部分的。但实际情况是我们只有 4 个值，所以很不幸，子集②可能会很大。这也就是为什么第二个问题很重要：该 SQL 调用是不是一个有优化空间的问题制造者？ 125

为了找出有优化空间的问题制造者，即子集③，我们需要判断结果集的行数。如果读取（F）的数量包含在了异常报告中（或者该值很容易被取得），那么我们可以把它作为结果行数的近似值。如果 PR/F 大于 5，那么这个查询很可能是一个有优化空间的问题制造者，因为在好的访问路径下，对每张表的一次典型读取只需一或二次以内的页请求。造成 PR/F 大于 5 的原因很可能是有大量无用的随机访问，即许多行被访问后就被丢弃掉了。

子集③包含了由于没有半宽索引而引起大量随机读的查询，这些是优化回报率最高的调用。子集③同样包含一些有好的访问路径的 SQL 调用（如使用宽索引对一个级联进行 COUNT 查询），但子集中的绝大部分查询可能都存在有趣的访问路径问题。

这些查询应该先用 Basic Question 方法（或者用第 8 章中讨论的 Basic Join Question 方法）分析一下。这样可能立刻揭露出不合适的索引。紧接着下一步是 EXPLAIN 一下这些 SQL 调用，看优化器选择了什么访问路径。这样可能会揭露出一个优化器问题，如一个复杂的谓词（匹配列太少）或者一个较差的过滤因子估算，进而导致错误的索引选择或错误的表访问顺序。

CPU 和 READS 值提供了对访问路径的有效指示器。例如，如果 CPU 接近 ET，那么处理过程大部分是顺序的；如果 CPU 很低，ET/READS 通常与每页的平均 I/O 时间接近；如果该值明显小于 1 ms，那么大部分的 I/O 是

顺序的；如果该值为几毫秒，那么大部分 I/O 可能是随机的。这能对执行计划所提供的信息进行补充。另外，任何高 CPU 值的 SQL 调用（如大于 500 ms），都可以被认为是一个有趣的问题制造者。

分析过子集③中排在前几位的调用（至少有一次很长 ET 的调用和那些经常很慢的调用）后，我们再看一下可能的问题制造者子集②中最慢且调用最频繁的 SQL 调用。我们可能会找出这样的查询：即使使用了半宽索引却仍旧太慢，但能够通过一个宽索引大大提升性能的查询。此类查询的概况类似下面这样：

ET = 8 s
CPU = 0.06 s
READS = 900
PR = 1050
F = 1000

当使用更好的索引或调整优化器的方式优化了一些 SQL 调用后，应该生成一个新的异常报告。索引的改善可能会帮助许多 SQL 调用、问题制造者及受害者，但是也有可能在新生成的问题列表中出现一些新的访问路径问题——主要是由于系统负载的改变，或者有时是由于索引的改变（幸运的是这不经常发生）。

126

按 ET 值排序，我们从上往下依次处理列表中的问题，那么要往下处理到第几个问题才能停下来呢？在一个操作型系统中，响应时间极少超过 1 s。那么理想情况下，我们应该检查任何响应时间大于 200 ms 的 SQL 调用。不幸的是，在实际生活中，我们可能永远都做不到如此。有太多的 SQL 调用需要处理，而可用的时间太少了！

图 7.11 总结了上文中我们讨论的术语及标准。为了演示这些术语及标准如何使用，我们马上会举两个调用级别监视的例子：一个 Oracle 的例子和一个 SQL Server 的例子。

调用级别的异常监视方法将用最少的努力发现许多严重的访问路径问题。若是如此，那为什么我们要说 LRT 级别的异常监视比调用级别的异常监视更好呢？

首先，展示最差 SQL 调用的报告不能发现那些有较快调用速度的问题，在此类问题中，每个事务执行许多次这样的快调用。例如，考虑一个单笔查询在一个循环中执行了 500 次的情况。如果访问路径不是仅限于索引的，那么由于两次随机读，每次调用的平均时间可能为 20 ms。很少有分析人员会关注一个耗费 20 ms 的 SQL 调用，但大多数人都会很关注一个耗费 10 s（500 × 20 ms）的事务！将所有没有匹配的列添加到索引上，这会使得该查询对

整体 LRT 的贡献从 10 s 降低至 5 s。

其次，LRT 级别的异常监视揭示了所有的响应时间问题，而不仅仅是那些与慢访问路径相关的问题。另外，它还能对随着数据库增长或事务频率上涨而开始变严重的问题做出早期警示。

Oracle 举例

在 Oracle 9i 的异常报告中，如下所示的数字会将每一个慢 SQL 调用显示在报告中——比如在一个测量时间段内的前 *N* 个最慢的调用。

```
Statistics
------------------------------
  0      db block gets
1021 consistent gets
1017 physical reads
CPU time            0.031250 s
Elapsed time   6.985000 s
```

consistent gets 是指在一致读模式下所请求的块数，而 db block gets 是指在当前模式下所请求的块数。为了能进行更新操作，块必须在当前模式下被请求。这两个值加在一起构成了查询的页请求（PR）总数。

physical reads 是指数据库实例向操作系统请求从磁盘读取的数据库块数。

该 SQL 调用总共耗费了 37 s，其中 CPU 时间耗费了 31 ms。它访问了 1021 个数据库页（块），几乎所有的这些页（1017 个）都是从磁盘读取的。 127

此 SQL 调用是受害者吗？可能不是，因为 ET/PR = 7000 ms/1021 ≈ 7 ms，小于 10 ms，此 SQL 调用是一个有优化空间的问题制造者吗？

当我们发现结果集只有 5 行数据（实际测得的结果）时，我们认为这个查询很可能是一个有优化空间的问题制造者：PR/F = 1021/5 ≈ 200。我们认为其中很可能存在大量无用的随机访问。

如果结果集大得多，PR/F 可能小于 5，除了 CPU 有些高以外，其他值大体上都相差不多。那么，这个 SQL 调用可能被发现是图 7.11 子集②中最慢的调用之一。无论是哪种情况，我们都可以进行如下观察。

从磁盘读取 1017 个页意味着较长的服务时间。由于 CPU 时间较短，I/O 时间可能与总体耗费的时间接近，约 7 s。那么平均读取每个页的 I/O 时间将大约是 7 ms，大部分的读取可能都是随机的。即使是获取 1000 行，我们也不希望每次获取都是通过随机读的方式获取的。如果最终获取的行数很少，那么我们就更应感到奇怪了。我们需要看一下这个 SQL 调用，并且判断它

使用了什么访问路径、涉及哪些索引。这些信息不仅能够解释为什么该查询会这样表现，而且也能显示出应采取何种改进措施。

CUST 表上唯一有用的索引是 CUST-X1(LNAME, FNAME)。该表有 1 000 000 行。SQL 7.1 中字段的过滤因子显示在了谓词旁边。

SQL 7.1

```
SELECT   CNO, FNAME, ADDRESS, PHONE
FROM     CUST
WHERE    LNAME = 'LNAME287'    FF = 0.1%
         AND
         CITY = 'TRURO'        FF = 0.5%
ORDER BY FNAME
```

```
Execution Plan
----------------------------------
 0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1020
        Card=5 Bytes=545)
 1    0  SORT (ORDER BY) (Cost=1020 Card=5 Bytes=545)
 2    1   TABLE ACCESS (BY INDEX ROWID) OF 'CUST'
          (Cost=1016 Card=5 Bytes=545)
 3    2    INDEX (RANGE SCAN) OF 'CUST_X1' (NON-UNIQUE)
          (Cost=16 Card=5)
```

执行计划显示访问路径使用了索引(LNAME, FNAME)，索引片(RANGE
128 SCAN)（而不是整个索引）被读取。表被访问了，且唯一令我们感到意外的是结果集被排序了。然而，排序不是问题，因为结果集只有 5 行。这正好是优化器所假设的（Card = 5），因为输入的值代表了平均情况。

现在，所测得的时间就容易理解了：1000 次对索引的顺序访问引起了相对少量的磁盘 I/O。如果每个叶子页有 40 个索引行，那么顺序访问的叶子页数将是 25。这只会花去几毫秒的时间。每次随机读的平均时间大概是 7 ms，这个时间也是较短的。Oracle 可能使用了并行数据块预读的方式来做随机读（这是一个非官方的访问路径，没有在执行计划中被报告出）。这就能解释为什么执行计划中会令人意外地出现排序了。

对于以上输入，在现有索引条件下，用 QUBE 算法计算出的时间为 $1001 \times 10\ \text{ms} + 1000 \times 0.01\ \text{ms} + 5 \times 0.1\ \text{ms} \approx 10\ \text{s}$，真实测量值为 7 s。

现在我们可以通过对比查询语句与索引确认导致 7 s 较长响应时间的原因了：CITY 列不在索引中。甚至索引都不是半宽的！由于谓词 LNAME = 'LNAME287'的过滤因子是 0.1%，所以 Oracle 必须对表做 1000 次随机访问来匹配谓词 CITY = 'TRURO'，从而找出所需的 5 行数据。至此，解决方法就很明显了：至少将 CITY 列加到索引上去，使索引成为半宽索引，以排除

对表的访问。

在我们迫不及待地设计一个解决方案之前，我们应该想到 5 行数据是个很小的结果集。因此，这可能不是事实上的最差情况。任何解决方案都应该对最差情况也适用。一个半宽的索引可能对于最差输入是不合适的。最差情况输入的过滤因子为 LNAME 1%和 CITY 10%。

对于现有索引，在最差输入条件下，用 QUBE 算法计算出的值为 10 001 × 10 ms + 10 000 × 0.01 ms + 1000 × 0.1 ms ≈ 100 s，实际测量值为 72 s。

而对于半宽索引，在上述均值输入条件下，用 QUBE 算法计算出的值将会是 6 × 10 ms + 1000 × 0.01 ms + 5 × 0.1 ms ≈ 0.07 s。

另外，对于半宽索引，在最差输入条件下，用 QUBE 算法计算出的值将会是 1001 × 10 ms + 1000 × 0.01 ms + 1000 × 0.1 ms ≈ 10 s。

由此看来，检查一下最差输入情况下的表现是很有必要的（见 SQL 7.2），半宽索引在这种情况下将是完全不可接受的。解决方案是必须使用一个宽索引，此时（在最差输入条件下）用 QUBE 算法计算出的值为 1 × 10 ms + 1000 × 0.01 ms + 1000 × 0.1 ms ≈ 0.1 s。

最差输入情况 SQL 7.2

```
SELECT   CNO, FNAME, ADDRESS, PHONE
FROM     CUST
WHERE    LNAME = 'SMITH'    FF = 1%
         AND
         CITY = 'LONDON'    FF = 10%
ORDER BY FNAME
```

于是一个宽索引适时地被创建了：

129

```
CREATE INDEX Cust X4 ON Cust (LNAME, CITY, CNO, FNAME, ADDRESS, PHONE)
```

且执行计划证实访问路径是仅限于索引的：

```
Execution Plan
----------------------------------
 0      SELECT STATEMENT Optimizer=CHOOSE (Cost=8
        Card=5 Bytes=545)
 1    0 SORT (ORDER BY) (Cost=8 Card=5 Bytes=545)
 2    1   INDEX (RANGE SCAN) OF 'CUST_X4' (NON-UNIQUE)
          (Cost=4 Card=5 Bytes = 545)
```

SQL Server 举例

SQL Server 2000 对于 SQL 调用也报告了类似的信息：

```
物理读                    2
预先读取              50 160
逻辑读                50 002
总时间                34 057(ms)
CPU 时间               1 933(ms)
```

此查询耗费了 34 s，其中 CPU 时间 1.9 s，页请求量（逻辑读）为 50 002，并且从磁盘读取了 50 160 + 2 = 50 162 个页。

该 SQL 调用是 Oracle 例子中的同一个，所使用的索引也是 CUST_X1(LNAME, FNAME)。但是，不同的是，这里的过滤因子代表了最差输入（LNAME 1%、CITY 10%——结果集有 1000 行）。另外，表行是在聚簇索引（CNO）中存储的。

在最差输入情况下（LNAME 的过滤因子为 1%），当唯一相关的索引为(LNAME, FNAME)时，优化器选择了全表扫描（如果访问路径的成本花费能够在每次执行时进行估算，那么这是一个好的选择——10 000 次随机读将会耗费更长时间）。

执行计划显示，整个聚簇索引（CNO）被扫描（无索引探寻），且对结果集进行了排序。

```
SHOWPLAN
-----------------------------------------------------
|--Sort(ORDER  BY([Cust].[Fname]ASC))
  |--Clustered Index
     Scan(OBJECT:([Invoices].[dbo].[Cust].[Cust_PK]),
     WHERE:( [Cust].[LNAME] = 'Adams'
      AND [CUST].[CITY]='BigCity').
```

130

此查询一定不是受害者（ET/PR = 34 s/50 002 ≈ 0.7 ms，这比 10 ms 小得多），它看起来更像是一个有优化空间的问题制造者（PR/F = 50 002/1000 ≈ 50，这比 5 大得多）。

CUST 表的大小为 400 MB（大约有 50 000 个 8KB 的页）。因此，顺序读的速度将会是 12 MB/s（34 s 扫描了 400 MB）。这表示这个小型服务器只有一个较慢的无条带的磁盘驱动器。

在使用了一个宽索引后，在最差输入情况下，性能优化的程度几乎与所期望的一样：

```
物理读                    3
预先读取                175
逻辑读                  176
总时间                  244 (ms)——从 34 s 下降至该值
CPU 时间                 40 (ms)——从 1.9 s 下降至该值
```

不过该查询仍将被报告为一个慢查询（ET > 200 ms），当然，它不是

一个受害者（ET/PR = 244 ms/176 ≈ 1.4 ms，比 10 ms 小得多），但它也不是个有优化空间的问题制造者（PR/F = 176/1000 ≈ 0.2，比 5 小得多）。

对于宽索引，在最差输入下，用 QUBE 算法计算出的值与 Oracle 例子中的一样：1 × 10 ms + 1000 × 0.01 ms + 1000 × 0.1 ms ≈ 0.1 s，实际测量值为 0.2 s。

注意：以上所有的测量都是在一个冷系统上进行的（数据库池或磁盘缓存中没有页）。由于预热的影响，测得的服务时间（包括 CPU 及 I/O）将会比在正常的生产环境中的服务时间长。从另一个角度来说，在生产环境中会有更多的争用及排队时间。

我们来看一下 SQL Server 是如何报告一个使用了非宽索引的访问路径的：

```
SHOWPLAN
--------------------------------
|--Filter(WHERE:( [CUST].[CITY]= 'Sunderland'))
     |--Bookmark Lookup(BOOKMARK:([Bmk1000]),
        OBJECT([Invoices].[dbo].[Cust]))
         |--Index
            Seek(OBJECT([Invoices].[dbo].[Cust_X1]),
            SEEK: ([Cust].[Lname]='Lname265')
              ORDERED FORWARD)
```

该查询语句及所使用的索引与索引优化前的相同，只是这次的输入对应平均的过滤因子。CUST 表同样仍存储在聚簇索引（CNO）中。

该执行计划显示，优化器扫描了 CUST_X1 索引的一个索引片，索引片 131
的宽度由谓词 Lname = 'Lname265'决定，该索引维持了索引行的顺序（ORDERED）从而避免了排序。对于索引片中的每一个索引行，优化器都在 CUST 表中检查相应的表行，并只取出那些 CITY 列匹配的行（过滤器）。

结论

基于以上所假设的基础信息，调用级别的异常监视能花费最少的努力揭示出许多索引问题。但是，受许多错误告警的影响，可能需要花费很长时间才能找出所有导致重大响应时间问题的索引问题，这远比通过 LRT 级别异常监视来寻找所花费的时间长得多。

如果报告中能展示的信息比例子中所展示的 4 个值的信息更多，那么识别那些可能有优化空间的问题制造者会变得更容易。结果集的行数和表的随机读的量可能是最有价值的两个信息。理想情况下应该有我们之前在气泡图中所展示的所有信息。

数据库管理系统特定的监视问题

从索引设计的角度来说，不同环境间（数据库管理系统和操作系统）最主要的区别是那些与性能监视相关的差异。许多其他方面的限制是可以被克服的，例如，如果最大索引键长度太短了，可以通过创建一个表，用 trigger 维护的方式来替代索引。而如果系统不提供合适的跟踪记录，或者提供的跟踪记录精度不合适，那么显然我们就无法测量出本地响应时间的各个组成部分。

如果数据库管理系统无法生成本章所描述的尖刺报告、气泡图或者 SQL 调用异常报告，那么索引设计就必须主要依赖前期预测、类似 QUBE 的快速估算，或者依赖优化器所做的估算。我们能够通过记录事务及 SQL 调用的时间，或者通过用户的抱怨发现尖刺问题。随后才可能去跟踪程序的运行来确认每一次 SQL 调用的时间花费和页请求数。然后再对较慢的 SQL 调用进行 EXPLAIN 及相应的分析。

对于这样的调用，根据 EXPLAIN 报告的访问路径进行 QUBE 计算，这可能对于判断较长响应时间的原因是有用的。

Mark Gurry 在他杰出的 Oracle 调优指南中展示了一些 SQL*Plus 命令，这些命令根据优化器的估算揭示了将会运行较慢的 SQL 语句，如下。

识别较差的 SQL

> 本小节中的 SQL 语句演示了如何识别出响应时间期望值大于 10 s
> 132 的 SQL 语句。假设每秒能够进行 300 次磁盘 I/O 及 4000 次缓存页请求。这些次数值是中高端机器的典型值。
>
> 因此，对于那些将会导致大于 3000 次磁盘 I/O 或者访问大于 40 000 次（根据优化器的估算）数据库页的 SQL 语句，所推荐的这些查询都能够报告出来。
>
> （2，第 67 页）

Mark Gurry（2，第 68 页）接下来提出了一些对于监视的看法：

> 在 Oracle 8i 及以后的版本中有一个很好的功能，在 V$SESSION_LONGOTS 中存储了当前处于活跃状态的长时间运行的查询。
>
> ……
>
> 也许 DBA 应该打电话给 HROA 的用户询问该语句，如果该语句将会执行太长时间的话，甚至也许会取消该语句的执行。

对于运行小型非关键应用的数据仓库环境，在线监视可能是合适的。但是在有许多用户的操作系统中，一个运行越来越慢的程序可能会使整个数据库服务器挂住。一个覆盖所有交互事务的异常报告能够对于所有类型的性能问题给出早期的警告，从而最小化发生性能危机的风险。

由于现在许多数据库管理系统供应商都投入大量资源进行研发，以使得系统的自我调优能力更好，所以很可能在接下来的几年许多产品中的监视功能将被改进。

尖刺报告

本章所示的尖刺报告在 z/OS 生产环境的 DB2 数据库中有两种实现方式。

一种是 DB2 Accounting Trace 记录，类别 1、2、3、7 和 8。通过 DB2PM（DB2 性能监视器），一个过滤命令查询出那些本地响应时间或 SQL 执行时间超出生产环境所设告警阀值的交互事务的相关记录。这些事务的概要信息会被自动地生成，不过要生成一个简明的报告（即每个尖刺一行），就需要一些程序代码并根据数据库缓冲池的设置进行少量定制（哪些对象与哪些池相关联）。可以通过写一个 REXX 程序读取标准报告记录，或通过保存三张表（BASE、PACKAGE 和 POOL）中的尖刺跟踪数据以供程序进行读取并生成尖刺报告的方式来实现。第二种方式对于生成长期尖刺报告是一个更好的选择，因为它对于跟踪记录的变化没有那么敏感。一旦生成尖刺报告，报告中的记录行就会被保存在一张历史表中，以方便将来的对比。即使是连续产生异常报告，它对磁盘空间的占用也是可以忽略不计的。

练习

7.1. 你将如何归类图 7.12 中的尖刺：有优化空间的问题制造者，无优化空间的问题制造者，还是受害者？

133

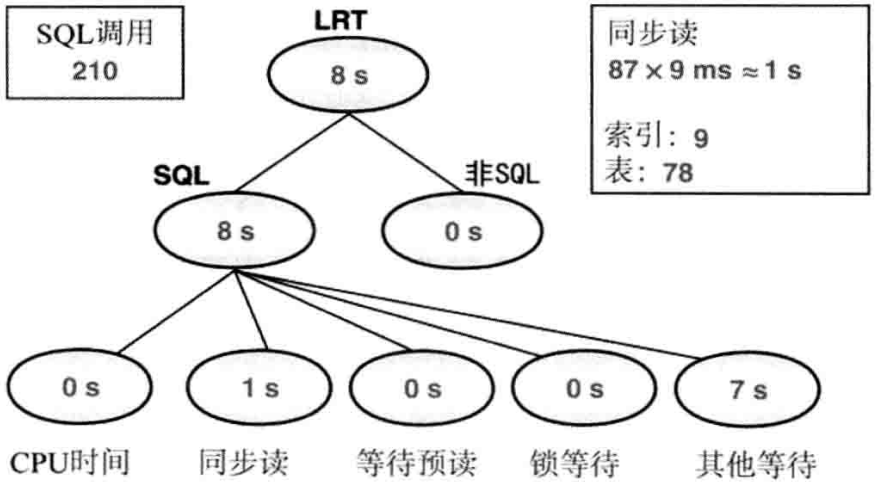


图 7.12　一个非典型尖刺

7.2. 如何减少本地响应时间？

第8章

为表连接设计索引

- 常见连接技术的介绍及其相关术语
- 用简单的表连接事例来说明嵌套循环这种最普通的连接方式的处理流程
- 两表连接的 SELECT 与用程序实现类似查询的对比
- 连接处理过程中所涉及的因素及其影响
- 本地谓词和连接谓词
- 表的访问顺序
- 内层表和外层表
- 通过研究案例来说明表的访问顺序在索引设计中的重要性，使用 QUBE 方法来说明三种不同程序实现方式的相对性能
- 合并扫描连接和哈希连接
- 与嵌套循环连接方式的对比
- 为什么这些连接技术在当前的硬件水平下得到了越来越广泛的应用
- 将 BQ 方法应用到表连接中，得出基本的连接问题，即 BJQ 方法
- 在多于两张表做连接时，以及使用子查询和联合的情况下，索引设计需要考虑的因素
- 为什么连接往往性能表现不佳
- 在设计表结构时需要为以后做连接查询提前做一些考虑
- 向上和向下反范式化
- 反范式化的成本
- NLJ 和 MS/HJ 与反范式化的对比
- 无意识的表设计

简介

目前我们提到的所有例子都是单表查询，现在可以讨论那些涉及多表的

136 查询了，即*表连接*。为连接查询设计合适的索引比为单表查询设计索引更困难，因为表的连接方式及表的访问顺序对索引影响很大。不过，在前面章节中所讨论的索引设计方法仍然适用于连接查询，只是需要进行一些额外的考量。

在一个连接查询中有两类谓词：*本地谓词*和*连接谓词*。只用于访问一张表的谓词称为本地谓词，定义了表和表之间的连接关系的谓词称为连接谓词。大部分优化器是通过评估各种可行方案的本地响应时间来决定采用哪种连接方式和表访问顺序的。尽管随着顺序读速度的提高，*哈希连接*和*合并扫描连接*（Oracle：*排序合并连接*；SQL Server：*合并连接*）变得越来越流行，但*嵌套循环连接*（SQL Server：循环连接）依旧是最常用的连接方式。在嵌套循环中，DBMS 首先在*外层表*中找到一行满足本地谓词的记录，然后再从*内层表*中查找与这一行数据相关的记录，并检查其中哪些符合内层表的本地谓词条件，以此类推。

一个两表连接查询可以被两个单表的游标以及在程序中编写的嵌套循环代码来代替。在这种方式中，表的访问顺序——哪张表在外层循环——是由程序员来决定的。若使用连接查询的方式，则表的访问顺序是由优化器来做决定的，虽然这可能被提示或其他技巧所重写的。

连接谓词大部分是基于*主键 = 外键*这一条件。假设外键上有合适的索引（以外键的列作为前导列的索引）并且结果集并不是特别的大，那么嵌套循环可能是最快的连接方式了，至少当所有的本地谓词均指向同一张表时是如此。

如果嵌套循环不合适，那么合并扫描或者哈希连接可能会更快。如果使用前者，在必要的情况下（在用本地谓词筛选之后），一张或者多张表将会按一致的方式进行排序，随后表或文件中符合条件的记录会被合并。哈希连接本质上是使用了哈希而非排序的一种合并扫描——稍后我们再详细讨论。使用这些方法时，所有表页被访问的次数都不会超过一次。在 Oracle、SQL Server 和 DB2 for LUW 中都倾向于选择哈希连接而非合并扫描。

DB2 for z/OS 不使用哈希连接，在一些较为少见的场景中可能会倾向于选择使用*混合连接*——一个带有列表预读的嵌套循环连接。

注：

- 在本章中，我们将首先集中阐述嵌套循环连接，然后再将其与合并扫描、哈希连接做比较。
- 在本章中，当数据需要从多张表中获取时，列名将由表名作为前缀，以此表明数据来自哪张表或哪个索引列。

两个简单的表连接

我们首先来分析两个简单的表连接的例子，以此说明嵌套循环连接方式所涉及的处理过程，以及如何计算连接所耗费的成本。

例 8.1：CUST 表作为外层表

137

SQL 8.1

```
DECLARE CURSOR81 CURSOR FOR
SELECT      CNAME, CTYPE, INO, IEUR
FROM        CUST, INVOICE
WHERE       CUST.CNO = :CNO
            AND
            CUST.CNO = INVOICE.CNO
WE WANT 20 ROWS PLEASE
```

这是一个从两个不同的表中获取数据的非常简单的例子(参见图 8.1)。查询从通过主键 CNO 索引访问主键列开始，所以在 CUST 表中将只访问一条记录便可获取两个列的值。

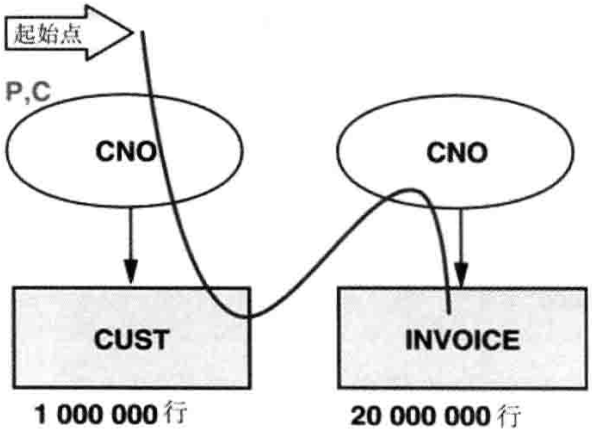


图 8.1　CUST 表作为外层表

连接谓词显示客户号也被用来通过外键 CNO 索引访问 INVOICE 表，从而获取另外两个列的数据。每一个客户平均会有 20 张发票（1 000 000 行客户记录和 20 000 000 行发票记录），当然有些客户会有更多的发票。

我们假设在 INVOICE 表上的 CNO 索引不是聚簇索引。

按照 QUBE 方法（基于 20 行的结果集）计算，将得出一个非常不错的响应时间，如下所示：

```
索引 CNO          TR = 1
表   CUST         TR = 1
```

```
索引 CNO              TR = 1              TS = 20
表  INVOICE           TR = 20
提取 20×0.1 ms
LRT                   TR=23               TS=20
                      23×10 ms            20×0.01 ms
                      230ms+0.2ms+2ms≈232ms
```

在这个连接查询中，CUST 表无疑是外层表。WHERE 语句中包含了主键的谓词条件，所以 DBMS 就会知道只有一条记录是满足查询需求的，因此也就只需对内层表进行一次简单的扫描。在 INVOICE 表上面没有本地谓词，所以该表必定是内层表。

用两个单独的查询语句代替单个连接查询语句所需要进行的处理过程也是同样的。第一步，在 CUST 表上使用主键进行查询；第二步，在第二个
138 SELECT 语句中使用同一个客户号通过一个游标进行查询，获取 20 条记录（平均而言）。同理，QUBE 的计算结果也是相同的（FETCH 调用量除外）。

例 8.2：INVOICE 表作为外层表

SQL 8.2

```
DECLARE CURSOR82 CURSOR FOR
SELECT      CNAME, CTYPE, INO, IEUR
FROM        CUST, INVOICE
WHERE       IDATE = :IDATE
            AND
            CUST.CNO = INVOICE.CNO
WE WANT 20 ROWS PLEASE
```

这是又一个从两张不同的表查询数据的简单例子（参见图 8.2）。在这个例子中，查询将从 INVOICE 表中 date 字段上的 IDATE 索引开始，因为查询语句中未提供有关客户的信息。假设我们只需要前 20 行发票数据，为此需要提取出其中所需的两个列的数据以及每一个发票的客户号，然后才能通过 CNO 上的索引来访问 CUST 表，即现在的内层表。CUST 表的访问像之前一样是通过主键索引进行的，每张发票对应 CUST 表中的一行记录。

每一张发票很可能对应一个不同的客户，所以会有较大数量的随机读。
139 因此这个查询的响应时间将比前一个查询的响应时间长，其 QUBE 结果如下：

```
索引 IDATE            TR = 1        TS = 19
表  INVOICE           TR = 20
索引 CNO              TR = 20
表  CUST              TR = 20
```

```
提取 20×0.1 ms

LRT                          TR=61          TS=19
                             61×10 ms       19×0.01 ms
                             610ms+0.2ms+2ms≈612ms
```

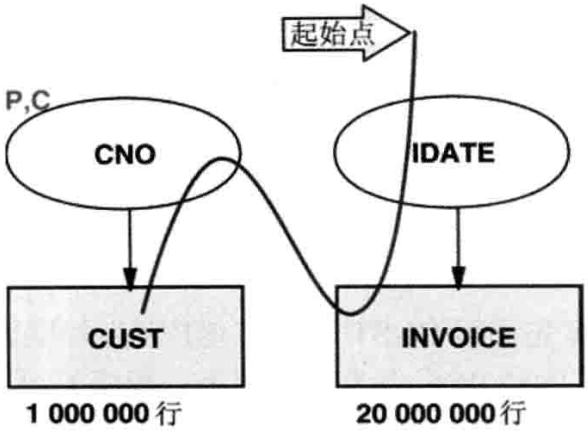


图 8.2　INVOICE 表作为外层表

同之前一样，哪张表是外层表，即起始点，是确定的。在 CUST 表上没有本地谓词，所以它是内层表。

当使用两个分开的查询代替单个连接语句时，处理过程也是同样的。第一步，通过 INVOICE 表的游标进行查询；第二步，通过第一步查询的客户号来查询 CUST 表，从而获取对应列的数据。同样，QUBE 结果是相同的（FETCH 调用量除外）。

表访问顺序对索引设计的影响

在前面两个例子中，本地谓词连同可用的索引现状使得我们在起始点方面别无他选。然而，许多人却发现实际情况并不总是如此。接下来我们将通过一个案例来讨论其中一种不那么简单的情况。

在案例研究的第一部分，我们将专注于讨论嵌套循环。当我们对其中涉及的问题都比较熟悉，特别是对于这种连接方式相关的潜在问题熟悉后，我们会将案例研究的重点转移至合并扫描连接和哈希连接。最后，我们将从索引设计的角度来比较这两种技术的优点和缺点。

140

案例研究

一家跨国公司拥有 80 万国内客户以及分布在其他 100 多个国家的 20 万客户。这 100 万客户的数据都存储在 CUST 表中。INVOICE 表有 20 万条记录。不论国内客户还是国外客户，平均每人都拥有 20 张发票。

现在需要实现一个新的查询，该查询需要通过 INVOICE 表上的 IEUR 列查出某个外国国家的高额发票，并将结果按降序排列。用户的输入包括一个发票总额的下限值和一个国家的代码。

这个查询可以用一个两表连接的 SELECT 语句来实现，还有一种实现方式是使用两个单表 SELECT 语句，并用程序实现表连接的逻辑。程序可以建立一个嵌套循环结构，例如程序 A 可以从表 CUST 开始（CUST 表作为外层表），程序 B 可以将 INVOICE 表作为外层表。由于这个查询操作是一个内连接，因此我们就能够对连接方式进行选择，我们只关心那些至少有一张大额发票的客户。如果我们的查询需求是获取客户信息和他们的发票信息（如果存在），那么这将会是一个外连接，在这种情况下，程序就必须先访问 CUST 表了。

在接下来的分析当中，程序 A 先访问 CUST 表（CUST 表为外层表），程序 B 从表 INVOICE 开始访问（INVOICE 表为外层表）。程序 C 采用折中的方式，使用两表连接的 SELECT 语句。若选择此方案，则我们是把表的连接方式和表的访问顺序都交由优化器选择了。当然，表这一层面的决策，如索引的选择等，在三个程序中都是由优化器来做决定的。

在分析这个三个程序之前，先花一点时间来考虑一下你会选择哪一个。一些程序员可能会下意识地选择程序 A，觉得以表 CUST 作为开始显得比较自然，因为一个客户有多张发票。另一方面，我们必须考虑到查询的需求是为了找出高额发票并按降序排列，在程序 B 中加入 ORDER BY IEUR DESC 可以满足这一需求。如果表 INVOICE 上的索引无法避免排序，那么 DBMS 将会进行排序。从这一点来看，程序 B 似乎更合适。

141 程序 A：CUST 表作为外层表

SQL 8.3

```
DECLARE CURSORC CURSOR FOR                          程序A
SELECT    CNO, CNAME, CTYPE
FROM      CUST
WHERE     CCTRY = :CCTRY

DECLARE CURSORI CURSOR FOR
SELECT    INO, IEUR
FROM      INVOICE
WHERE     IEUR > :IEUR
          AND
          CNO = :CNO

OPEN CURSORC
     FETCH CURSORC          while CCTRY = :CCTRY
     OPEN CURSORI
          FETCH CURSORI     while IEUR > :IEUR
     CLOSE CURSORI
CLOSE CURSORC

 用IEUR DESC对结果集排序
```

程序 B：INVOICE 表作为外层表

SQL 8.4

```
                                                    程序 B
DECLARE CURSORI CURSOR FOR
SELECT   CNO, INO, IEUR
FROM     INVOICE
WHERE    IEUR > :IEUR
ORDER BY IEUR DESC

OPEN CURSORI
     DO
          FETCH CURSORI    while IEUR > :IEUR
          SELECT      CNAME, CTYPE
          FROM        CUST
          WHERE       CNO = :CNO
                      AND
                      CCTRY = :CCTRY
     DO END
CLOSE CURSORI
```

程序 C：由优化器选择外层表

142

SQL 8.5

```
                                                    程序 C
DECLARE CURSORJ CURSOR FOR
SELECT     CNAME, CTYPE, INO, IEUR
FROM       CUST, INVOICE
WHERE      IEUR > :IEUR
           AND
           CCTRY = :CCTRY
           AND
           CUST.CNO = INVOICE.CNO

ORDER BY   IEUR DESC

OPEN CURSORJ
  FETCH CURSORJ  while IEUR > :IEUR and CCTRY = :CCTRY
CLOSE CURSORJ
```

就像我们在 SQL 8.1 和 SQL 8.2 的 SQL 中所看到的那样，当只有一张表上有本地谓词时，从性能的角度来说，从有本地谓词的表开始查询是一个好主意。现在两张表上都有本地谓词，所以程序运行的快慢差别就不那么明显了——除非我们知道表上有哪些索引。

出于案例研究的目的，我们假设除了主键和外键索引外，唯一一个额外的索引在表 CUST 上（CCTRY 字段），如图 8.3 所示。CCTRY 代表国家的代码。这种情形下，大部分读者很可能都会选择程序 A，因为看起来 CUST 表上有一个合适的索引，而 INVOICE 表上却没有。

同往常一样，当我们开发一个新程序时，我们需要检查现有索引是否够用，包括在最差输入的情况下。若不够用，那么我们就需要设计更适合的索引，可能以设计和评估理想索引作为开始。

143

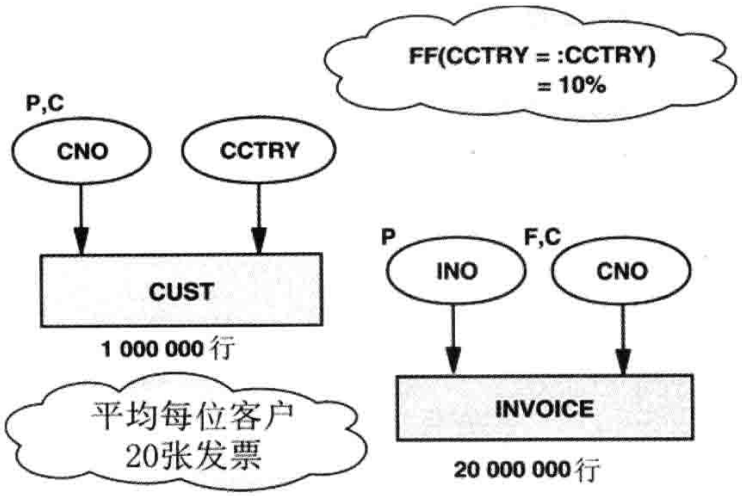


图 8.3　现有索引

为了对这三个程序之间存在的实际性能差异有一个清晰的理解，我们来比较一下用 QUBE 对它们评估出的结果，从而了解在当前的索引条件下，这些程序将表现如何。

在评估中我们将使用最高的过滤因子，假设这代表了最差的情况：

```
FF (CCTRY = :CCTRY)          最高过滤因子  10%
FF (IEUR > :IEUR)            最高过滤因子  0.1%
```

现有索引

程序 A：CUST 表作为外层表

SQL 8.6

```
DECLARE CURSORC CURSOR FOR                                    程序 A
SELECT      CNO, CNAME, CTYPE
FROM        CUST
WHERE       CCTRY = :CCTRY

DECLARE CURSORI CURSOR FOR
SELECT      INO, IEUR
FROM        INVOICE
WHERE       IEUR > :IEUR
            AND
            CNO = :CNO

OPEN CURSORC
     FETCH CURSORC          while  CCTRY = :CCTRY
     OPEN CURSORI
          FETCH CURSORI     while  IEUR > :IEUR
     CLOSE CURSORI
CLOSE CURSORC

 用 IEUR DESC 对结果集排序
```

图 8.3 展示了现有的索引情况。该图再一次展示了这三个程序连同其对应的访问路径对每个索引及表所需的访问次数。

第 1 步：通过非聚簇索引 CCTRY 访问 CUST 表　DBMS 必须找到所有来自特定国家的客户。这可以通过一次全表扫描或者通过索引 CCTRY 来实现（如图 8.4 所示）。当谓词 CCTRY = :CCTRY 具有很高的过滤因子时，全表扫描会更快，当过滤因子较低时，索引扫描会更快。如果优化器为多次查询只做一次访问路径的选择，那么它会基于 1%的过滤因子（通过 1 除以列 CCTRY 的不同值个数得出）而选择索引扫描。若优化器基于用户每次输入的实际值进行访问路径的选择，那么只要它知道真正的过滤因子值，它就会毫不犹豫地选择全表扫描。

144

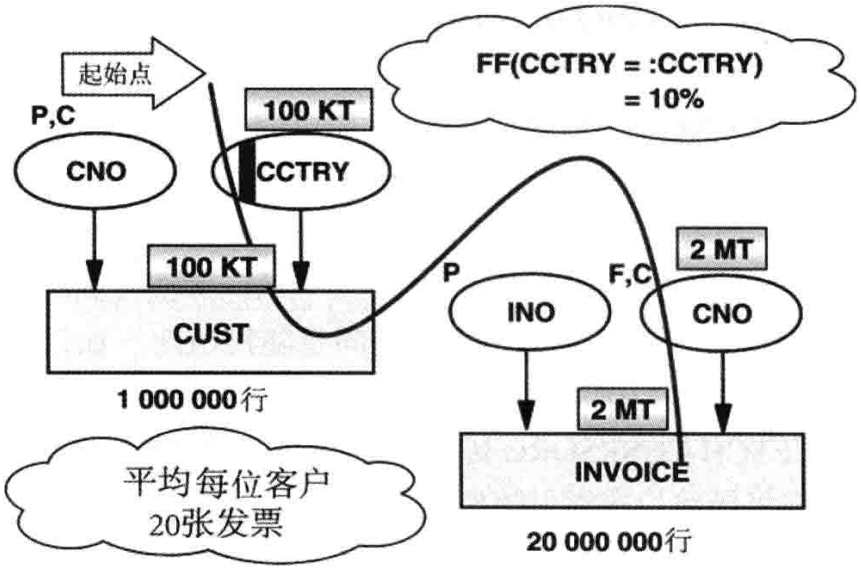


图 8.4　在现有索引条件下运行程序 A

```
索引 CCTRY          TR = 1                  TS = 10% x 1 000 000
表   CUST           TR = 100 000            TS = 0

提取 10% x 1 000 000 = 100 000 x 0.1 ms

LRT                 TR = 100 001            TS = 100 000
                    100 001 x 10 ms         100 000 x 0.01 ms
                    1000 s + 1 s + 10 s ≈ 1000 s
```

根据 QUBE 方法，这一步为响应时间贡献了 1000 s！由于 CCREY 索引不是聚簇索引，所以使得 CUST 表上的随机访问量很高。

使用这一“想当然”的访问路径所需的时间太多了。在过滤因子为 10%的情况下，100 000 次随机读（约为 17min）显然是不可接受的。即使过滤因子变为 1%，也需要 10 000 次随机读（1.7min）。而另一种方法，全表扫描，只需要：

$$1 \times 10\ \text{ms} + 1\ \text{m} \times 0.01\ \text{ms} = 10\ \text{s}$$

当过滤因子为 0.1%时，索引扫描需要 1000 次随机读，与全表扫描消耗

的时间一致。当然，这是因为在 QUBE 方法中，一次随机访问所需时间是一次顺序访问的 1000 倍。在过滤因子大于 0.1%的情况下，一次全表扫描查询比一次非宽非聚簇的索引扫描要快很多。然而，在得出类似这样的笼统结论时，我们需要非常小心，因为我们的结论是完全基于 QUBE 的数字假设而得出的。我们还需要考虑到表扫描在 CPU 时间上的耗费会高得多。无论如何，我们应该意识到，在过滤因子很低的情况下，进行非宽非聚簇索引的扫描比进行表扫描更好。

145 连同数据提取的时间成本一起，这一步的 LRT 将会是 20s。

第 2 步：通过聚簇索引 CNO 访问 INVOICE 表　当从 CUST 表找出一位来自指定国家的客户后，该客户的所有发票都必须被检查以找到大额发票。由于 INVOICE 表上的 CNO 索引是聚簇索引，所以同表行一样，这些记录在索引上是相邻的（平均每位客户有 20 行）。对于类似 CUST 和 INOVICE 这样互相关联的大表，拥有一致聚簇方式是非常重要的。对一位能代表平均情况的用户而言，需要进行 21 次索引访问和 20 次表访问才能找到所有大额发票，但是只有第一次索引访问和第一次表访问是随机访问。如果处理了 100 000 位客户（最差输入：10%的客户），那么访问次数将如下所示。需要注意，程序中从 FETCH CUSRSORC 传递至 OPEN CURSORI 的客户号是不连续的，每一次都将导致一次索引随机访问和表随机访问。

```
索引 CNO                TR = 100 000        TS = 100 000×20
表   INVOICE            TR = 100 000        TS = 100 000×19

提取 100 000×0.1ms

LRT                     TR = 200 000        TS = 3 900 000
                        200 000×10 ms       3 900 000 ×0.01 ms
                        2000s+39s+10s≈2050 s
```

根据 QUBE 算法，这一步骤的开销成本是第一步的两倍多，所以第一步的表扫描与此次内层表的处理成本相比就显得不那么明显了。

第 3 步：结果集排序　在最差的输入情况下，结果集将包含 0.1 × 0.001 × 20 000 000 = 2000 行大额发票数据。这些结果必须按照发票总额（IEUR）的降序排列。排序操作将会是在最后额外增加的一步操作，可能会使用到临时表。根据我们的估算，这一排序的 CPU 时间大约是每行 0.01ms，2000 千行将是 0.02s。这么小的一个排序操作很可能不会产生磁盘 I/O 操作，所以总的时间消耗也就是大约为 0.02s。QUBE 方法假设排序时间被提取操作的时间所囊括了，因为它非常小以至于可以忽略不计。

本地响应时间　在最差输入情况下，这一访问路径的本地响应时间由以下三部分组成

$$20\ s + 2050\ s + 0\ s = 2070\ s \approx 35\ min$$

为了等待第一屏，用户要等待半个多小时！即便你告诉用户在输入国家号码后应该去休息一下，但由于磁盘流量剧增，可能使随机读在半小时内上升至 200 000 次之多，这可能会导致其他正在使用的用户处于很明显的排队状态。

在图 8.5 所示的索引条件下，数据库管理系统必须扫描整个 INVOICE 表来寻找大额发票。当找到一个之后，通过 CUST 表上的主键索引来访问对应的 CUST 表记录，以此方式来检验该客户的国籍。 146

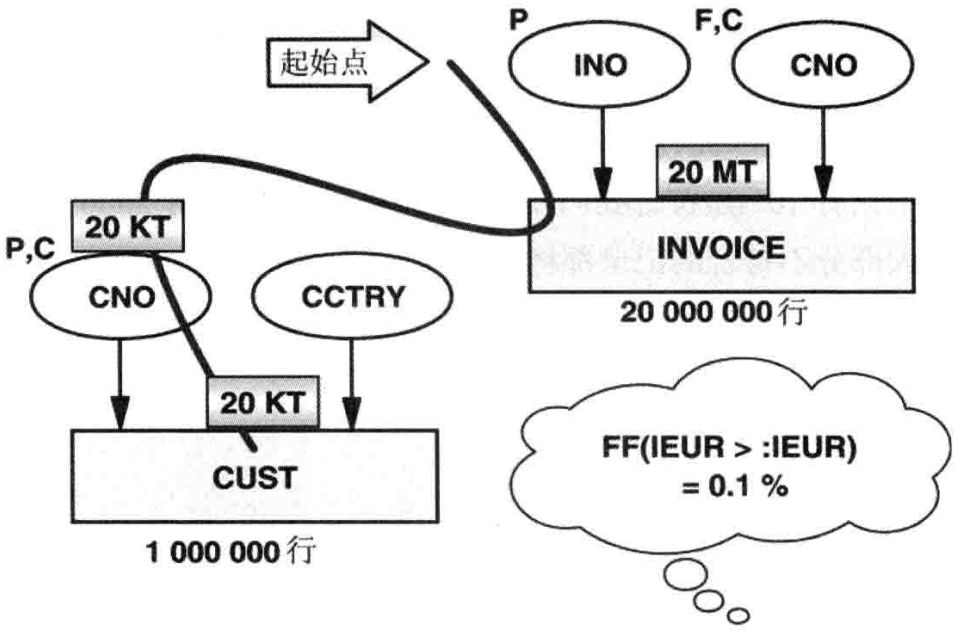


图 8.5　在现有索引下的程序 B

程序 B：INVOICE 表作为外层表

SQL 8.7

程序 B

```
DECLARE CURSORI CURSOR FOR
SELECT      CNO, INO, IEUR
FROM        INVOICE
WHERE       IEUR > :IEUR
ORDER BY    IEUR DESC

OPEN CURSORI
      DO
            FETCH CURSORI        while  IEUR > :IEUR
            SELECT       CNAME, CTYPE
            FROM         CUST
            WHERE        CNO = :CNO
                         AND
                         CCTRY = :CCTRY
      DO END
CLOSE CURSORI
```

147 **第 1 步：INVOICE 表全表扫描**　INVOICE 表有 20 000 000 行，要把每一行都检查一遍需要 20 000 001 次访问，其中包含了文件末尾标志的访问。只有第一次访问是随机访问。

```
表   INVOICE        TR = 1          TS = 20 000 000
提取 0.1%×20 000 000 = 20 000×0.1 ms
LRT                 TR = 1          TS = 20 000 000
                    1×10 ms         20 000 000 ×0.01 ms
                    10ms+200s+2s≈200s
```

第 2 步：通过聚簇索引 CNO 来读取大额发票所对应的 CUST 表行　用我们所假定的最差情况下的过滤因子，INVOICE 表中有 0.001 × 20 000 000 = 20 000 张都是大额的。因此，DBMS 将通过 CNO 索引来读取 20 000 行 CUST 表记录。其中只有 10%是符合我们要查找的国家的，即只有 2000 行是符合条件的，而大部分不符合的记录都将被抛弃。

```
索引 CNO            TR = 20 000
表   CUST           TR = 20 000
提取 20 000×0.1 ms
LRT                 TR = 40 000
                    40 000×10ms+2s≈400s
```

第 3 步：结果集排序　在第 1 步中从 CURSORI 中取到的 20 000 条数据必须按发票总额 IEUR 进行降序排列。排序 20 000 条记录所花费的时间（约200ms）是可以忽略的，同样假设这部分开销被FETCH的开销所囊括了。

本地响应时间　在最差输入的情况下，采用这种访问路径的本地响应时间是以下三部分的总和

$$200\ \text{s} + 400\ \text{s} + 0\ \text{s} = 600\ \text{s} = 10\ \text{min}$$

我们惊讶地发现，程序 B 比程序 A 快很多，即便是在当前索引的条件下——IEUR 上面没有索引！怎么会如此呢？这里有如下两个原因。

1. 在当前的硬件条件下，顺序读是非常快的：一张有 20 000 000 行记录的表可能在大约 3 min 内就能被全部扫描一遍了（基于 QUBE 的假定数字）。
2. 由于谓词 IEUR < :IEUR 的最大过滤因子仅为 0.1%，因此使用程序 B 时对内层表的随机访问次数比用程序 A 时要少。

程序 C: 由优化器选择外层表

148

SQL 8.8

程序 C

```
DECLARE CURSORJ CURSOR FOR
SELECT      CNAME, CTYPE, INO, IEUR
FROM        CUST, INVOICE
WHERE       IEUR > :IEUR
            AND
            CCTRY = :CCTRY
            AND
            CUST.CNO =  INVOICE.CNO
ORDER BY    IEUR DESC

OPEN CURSORJ
 FETCH CURSORJ  while  IEUR > :IEUR and  CCTRY = :CCTRY
CLOSE CURSORJ
```

这是一个介于前面两者之间的程序。由优化器来评估谓词 IEUR > :IEUR 和 CCTRY = :CCTRY 的过滤因子(如我们在第 3 章中看到的那样)，然后再相应地选择表的访问顺序。

对优化器而言，IEUR > :IEUR 并不是一个易于评估的谓词，因为优化器并不知道用户对主机变量:IEUR 将会输入的值的范围。优化器可能会用一个默认值，如 33%或者 50%。接下来如果它选择嵌套循环的方式，那就是选择了与程序 A 一样的访问路径，即 CUST 表为外层表，如图 8.6 所示。这可能导致在最差输入下，本地响应时间长达 35 min 而不是 10 min。

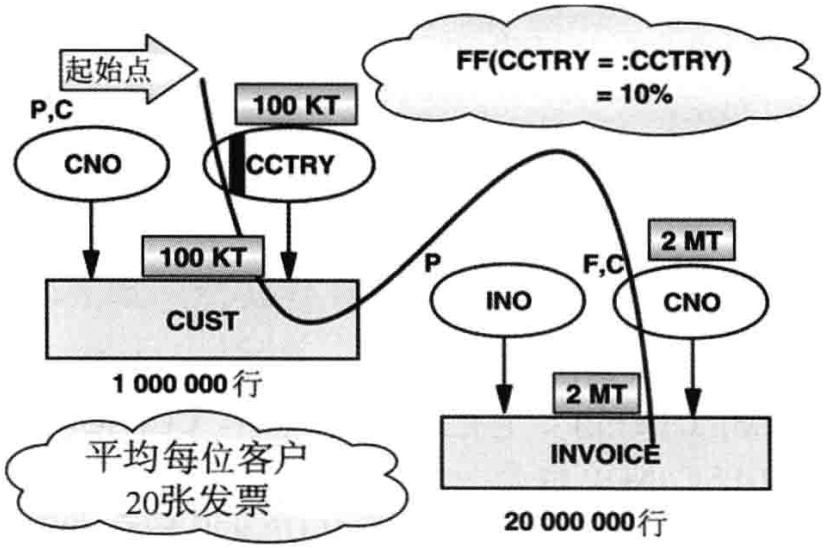


图 8.6　在现有索引下的程序 C

除了对外层表的选择以外，程序 A、程序 B 与程序 C 的 QUBE 评估结果的唯一区别就是 FETCH 调用的次数：

149

程序 A	200 000 × 0.1 ms = 20 s
程序 B	40 000 × 0.1 ms = 4 s
程序 C	2000 × 0.1 ms = 0.2 s

从整体耗时上看，这一收益非常小。当然，在其他场景下，这一收益可能是很显著的。

结论：现有索引

很显然，现有索引对于最差输入情况并不够用。我们现在应当开始考虑设计最合适的索引了，我们以设计和评估理想索引作为开始。

理想索引

程序 A：CUST 表作为外层表

SQL 8.9

程序 A

```
DECLARE CURSORC CURSOR FOR
SELECT      CNO, CNAME, CTYPE
FROM        CUST
WHERE       CCTRY = :CCTRY

DECLARE CURSORI CURSOR FOR
SELECT      INO, IEUR
FROM        INVOICE
WHERE       IEUR > :IEUR
            AND
            CNO = :CNO

OPEN CURSORC
      FETCH CURSORC              while  CCTRY = :CCTRY
      OPEN CURSORI
            FETCH CURSORI        while  IEUR > :IEUR
      CLOSE CURSORI
CLOSE CURSORC

用 IEUR DESC 对结果集排序
```

根据第 4 章所讨论的设计最佳索引的方法，CURSORC 的候选索引 A 是(CCTRY,CNO,CNAME,CTYPE) ，它是一个三星索引。CURSORI 的候选索引 A(CNO,IEUR DESC,INO) 也是一个三星索引。当然 CNO 来自 CURSORC。现在我们有了一个基于列 CNO 和列 IEUR 的比较窄的索引片。由于这两个索引都是三星索引，所以就不必考虑候选索引 B 了。这两个索引如图 8.7 所示。严格地说，对于程序 A 而言，CURSORI 并不需要 IEUR 在理想索引中做降序排列。但为了简化程序 A 与程序 B 的比较，我们仍将假设 IEUR 是一个降序的列。

150

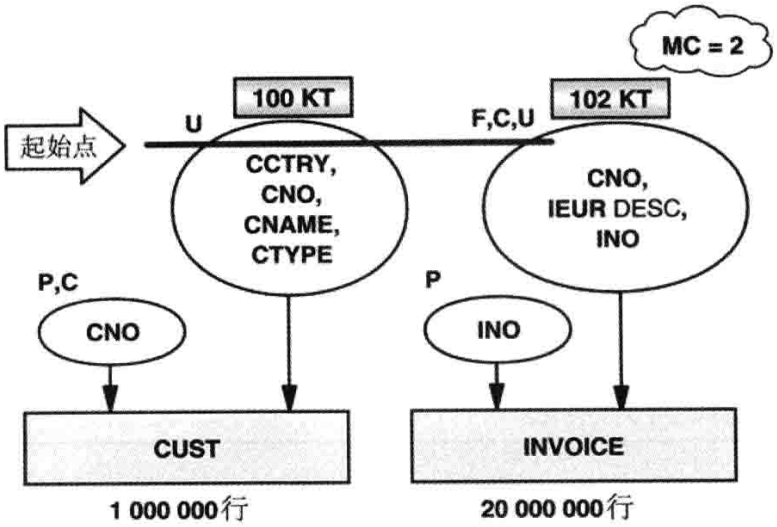


图 8.7　理想索引条件下的程序 A

第 1 步：从索引(CCTRY,CNO,CNAME,CTYPE)上读取索引片(CCTRY = :CCTRY)　由于 CCTRY = :CCTRY 的过滤因子为 10%，所以索引片包含 0.1 × 1 000 000 的索引行。由于我们使用了一个三星索引，所以不需要对表进行访问。

现在这一步中的两个主要开销变成了索引的扫描和数据的提取调用。索引的扫描已做到了尽可能窄，因而数据提取调用成为了更关键的一个开销，大量的提取调用将耗费 10 s!

```
索引 CCTRY,CNO,CNAME,CTYPE     TR = 1          TS = 10% × 1 000 000
提取 10% × 1 000 000 = 100 000 × 0.1 ms
LRT                             TR = 1          TS = 100 000
                                1 × 10 ms       100 000 × 0.01 ms
                                10 ms + 1 s + 10 s ≈ 11 s
```

第 2 步：对于每一个 CNO，通过索引(CNO,IEUR DESC,INO)来读取所有的大额发票　现在每一个 CNO 对应的索引片由列 CNO 和 IEUR 来定义。当一个客户没有大额发票时，DBMS 只需要对索引执行一次随机访问即可。如果索引的第一行是大额发票，那么 DBMS 需要继续读取下一行，如此继续。由此，随机访问的次数是 100 000 次（一个具体 CCTRY 值对应的客户数量），顺序读取的次数是 2000 次（一个国家对应的大额发票的数量）。

151

```
索引 CNO,IEUR DESC,INO          TR = 100 000        TS = 2000
提取 100 000 × 0.1 ms
LRT                             TR = 100 000        TS = 2000
                                100 000 × 10 ms     2000 × 0.01 ms
                                1000 s + 0.02 s + 10 s ≈ 1000 s
```

FETCH 调用将会耗费额外的 10 s，与对索引的随机读取的时间相比，这 10 s 是微不足道的。

本地响应时间　像以前一样，我们忽略排序的开销，本地响应时间为

$$11\text{ s} + 1000\text{ s} = 1011\text{ s} \approx 17\text{ min}$$

我们必须承认一个很重要的事实，即在这个案例中，理想索引所带来的好处被最小化了，因为有大量对内层表索引的随机访问。

程序 B：INVOICE 表作为外层表

SQL 8.10

程序 B

```
DECLARE CURSORI CURSOR FOR
SELECT      CNO, INO, IEUR
FROM        INVOICE
WHERE       IEUR > :IEUR
ORDER BY    IEUR DESC

OPEN CURSORI
     DO
          FETCH CURSORI          while  IEUR > :IEUR
          SELECT      CNAME, CTYPE
          FROM        CUST
          WHERE       CNO = :CNO
                      AND
                      CCTRY = :CCTRY
     DO END
CLOSE CURSORI
```

程序 B 的理想索引和程序 A 的理想索引是不同的，因为现在用于访问两张表的信息发生了变化。

CURSORI 的候选索引 A 现在变为了(IEUR DESC,INO,CNO)。客户表的候选索引 A 是(CCTRY,CNO,CNAME,CTYPE)。CNO 字段来自 CURSORI。它们都是三星索引，如图 8.8 所示，因此不需要考虑候选索引 B 了。

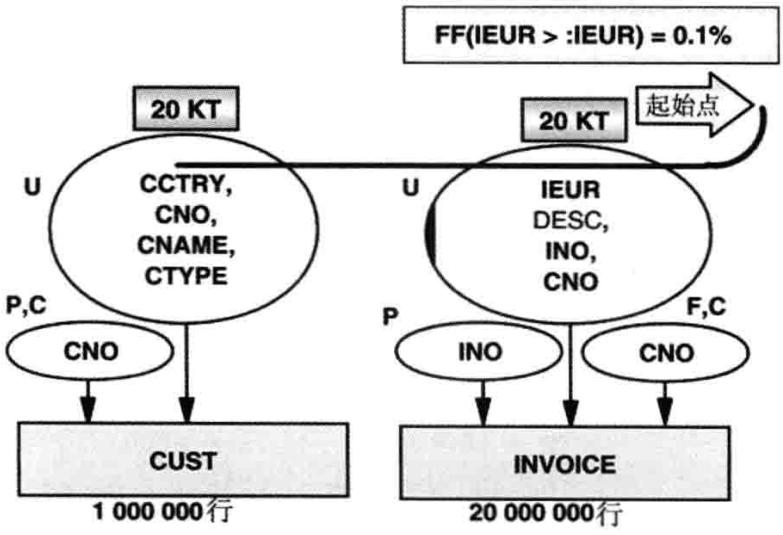


图 8.8　在理想索引条件下的程序 B

第 1 步：从索引(IEUR DESC,INO,CNO)上读取 IEUR > :IEUR 对应的索引片　该索引片包含 0.001 × 20 000 000 = 20 000 索引行。因此，DBMS 需要进行一次随机访问和 20 000 次顺序访问。现在，这一步的主要开销变成了数据提取的处理。开销不会像程序 A 中的开销那么大，但依旧有 2s 之多。索引按照所要求的顺序提供结果集，因此无须再进行排序操作。 152

```
索引 IEUR DESC,INO,CNO          TR = 1        TS = 20 000
提取 0.1% × 20 000 000 = 20 000 × 0.1 ms
LRT                             TR = 1        TS = 20 000
                                1 × 10 ms     20 000 × 0.01 ms
                                10 ms + 0.2 s + 2 s ≈ 2 s
```

第 2 步：对每一张大额发票读取一个索引行(CCTRY,CNO,CNAME,CTYPE)　由于查询的客户号并不是连续的，所以 DBMS 需要做 20 000 次随机访问。其中只有 10%的记录符合所指定的国家。

```
索引 CCTRY,CNO,CNAME,CTYPE      TR = 20 000
提取 20 000 × 0.1 ms
LRT                             TR = 20 000
                                20 000 × 10 ms
                                200 s + 2 s = 202 s
```

本地响应时间

$$2\text{ s} + 202\text{ s} \approx 3.5\text{ min}$$

从程序 A 的 17 min 到现在的 3.5min 是一个巨大的缩减，但不幸的是，我们还是要强调与程序 A 中相同的一点：使用理想索引的好处被最小化了，因为存在大量的对内层表索引的随机访问。 153

程序 C：由优化器选择外层表

在使用理想索引的情况下，根据我们在评估程序 A 和程序 B 中得出的结论，如果优化器选择了嵌套循环的连接方式，那么表 INVOICE 应当作为外层表。因此，所需要的理想索引应当如图 8.8 所示。由于数据提取的调用次数有所降低，所以本地响应时间相对程序 B 将有所下降，不过如我们所见，这一降低是很有限的。

SQL 的连接方式和程序的连接方式对比　人们通常说使用 SQL 连接比使用多游标的程序更高效，这是真的吗？

决定这一推测的最关键的因素是访问路径的选择，也就是连接方式和表访问顺序。程序有时可能会选择比优化器更好的访问路径，有时则不会。最终的访问路径应该是相同的，就像程序 B 和程序 C 一样。磁盘 I/O 的数量

也应当是一样的，唯一的不同就是 SQL 调用的次数。上文中的数字清楚地显示了在这一方面使用连接查询更好。

由于程序 A 和程序 B 中存在大量的随机 I/O，CPU 时间相对总的消耗时间占比很低。然而，在有些情况下，CPU 时间在本地响应时间中占据了最主要的部分，在这种情况下，使用连接游标的程序将比使用多个单游标的程序快得多。

结论：理想索引

虽然使用三星索引后，最差输入的本地响应时间从 35min 减少到了 3.5min，但这一响应时间仍旧太长了。下一步必须考虑更改程序，以使它只产生一屏的数据，比如每个事务 20 行记录。请注意，我们迄今为止只关注了嵌套循环连接。我们目前得出的结论很可能会受到一些尚未进行讨论的因素的影响。

理想索引，每事务物化一屏结果集

程序 B+：物化单屏事务

我们很容易修改程序B，使它只发起20次数据提取调用。只需要做到：

- 确保访问路径不需要进行排序操作。
- 修改程序，使其对所发起调用次数进行计数。
- 将位置信息保存在一张辅助表中，以使下一个事务从该点继续执行。

154

首个事务所需的SQL变更如SQL 8.11所示。请关注其中的20 ROWS PLEASE语句和SAVE语句所保存的值。另外，CURESORI游标中发票编号被加到了ORDER BY语句上，因为这对于定位起点是必需的。

当需要请求下一屏数据时，程序必须从所保存的值处开始执行。让我们将这两个值命名为 IEURLAST 和 INOLAST。后续事务所需的 SQL 变更如 SQL 8.12 所示。很可能存在两张或更多的发票拥有相同的总额（列 IEUR）。因此，程序必须打开一个游标 CURSORI1 用于查询那些总额与 IEURLAST 相等但仍未被展示的发票记录。当该值所对应的所有记录都被提取后，第一个游标被关闭，同时第二个游标 CURSORI2 被打开，用于查询下一批大额发票记录。

SQL 8.11

程序 B+ 首个事务

```
DECLARE CURSORI CURSOR FOR
SELECT    CNO, INO, IEUR
FROM      INVOICE
WHERE     IEUR > :IEUR
ORDER BY  IEUR DESC, INO
WE WANT 20 ROWS PLEASE

OPEN CURSORI
     DO
          FETCH CURSORI              max 20 times
          SELECT        CNAME, CTYPE
          FROM          CUST
          WHERE         CNO = :CNO
                        AND
                        CCTRY = :CCTRY
     DO END
CLOSE CURSORI

SAVE / INSERT IEUR, INO  the values of the last line
displayed to the user
```

现在，在使用理想索引的情况下，建立一屏数据需要耗费多少时间？让我们再次假设查询对应最大的过滤因子分别为10%和0.1%，如图8.9和SQL 8.12所示。

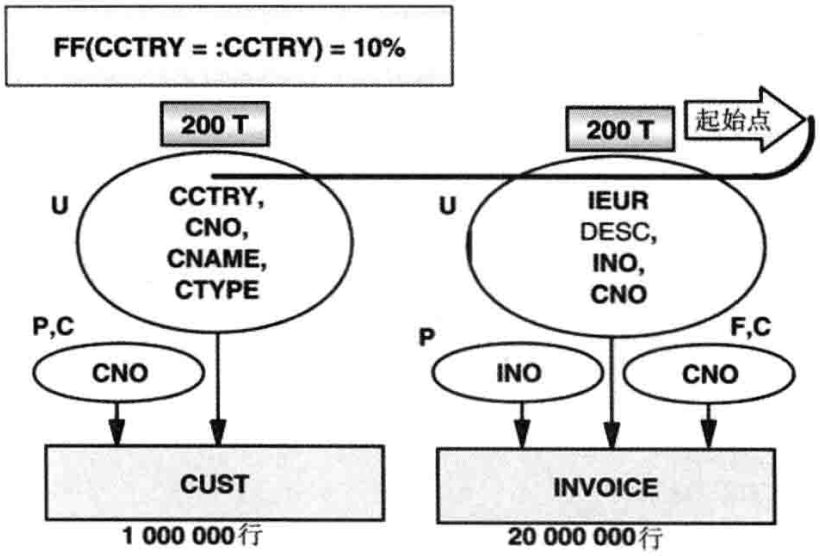


图 8.9　在理想索引条件下且每个事务物化一屏结果集的程序 B+

第1步：从索引(IEUR DESC,INO,CNO)上查询20张大额发票　响应时间与通过游标CURSORI1和CURSORI2查找到多少行无关。程序无论使用哪个游标，都将扫描一个有200行数据的索引片，因为若过滤因子FF(CCTRY = :CCTRY)为10%，那么平均每10张发票中有一张是来自该特定国家的。这一步骤大约需进行20/0.1 = 200次数据访问，其中只有第一次访问是随机访问。

155

SQL 8.12

程序 B+ 后续事务

```
DECLARE  CURSORI1 CURSOR FOR
SELECT   CNO, INO, IEUR
FROM     INVOICE
WHERE    IEUR = :IEURLAST
         AND
         INO > :INOLAST
         AND
         IEUR > :IEUR
ORDER BY IEUR DESC, INO
WE WANT 20 ROWS PLEASE

DECLARE CURSORI2 CURSOR FOR
SELECT    CNO, INO, IEUR
FROM      INVOICE
WHERE     IEUR < :IEURLAST
          AND
          IEUR > :IEUR
ORDER BY  IEUR DESC, INO
WE WANT 20 ROWS PLEASE

READ IEURLAST, INOLAST

OPEN CURSORI1
     DO
          FETCH CURSORI1                max 20 times
          SELECT          CNAME, CTYPE
          FROM            CUST
          WHERE           CNO = :CNO
                          AND
                          CCTRY = :CCTRY
     DO END
CLOSE CURSORI1

OPEN CURSORI2
     DO
           FETCH CURSORI2    until 20 result rows
           SELECT       CNAME, CTYPE
           FROM         CUST
           WHERE        CNO = :CNO
                        AND
                        CCTRY = :CCTRY
     DO END
CLOSE CURSORI2

SAVE / INSERT IEUR, INO the values of the last line
displayed to the user
```

156

索引 IEUR DESC,INO,CNO　　TR = 1　　TS = 199

提取 20 × 0.1 ms

LRT　　TR = 1　　TS = 199

1 × 10 ms　　199 × 0.01 ms

10 ms + 2 ms + 2 ms = 14 ms

第2步：对每一笔大额发票读取一个索引行(CCTRY,CNO,CNAME,CTYPE)　由于客户编号是不连续的，所以DBMS必须进行200次随机访问。

```
索引 CCTRY,CNO,CNAME,CTYPE    TR = 200
提取 200 × 0.1 ms
LRT                           TR = 200
                              200 × 10 ms
                              2 s + 20 ms ≈ 2 s
```

本地响应时间

$$14\ \text{ms} + 2\ \text{s} \approx 2\ \text{s}$$

结论：使用理想索引且每个事务返回一屏数据　157

在最高过滤因子的情况下，本地响应时间是2 s，这是比较能接受的。

理想索引，每事务物化一屏结果集且遇到 FF 缺陷

以上的评估是在最大过滤因子（10%和 0.1%）的情况下做出的。但这不是最差输入情况，因为这个案例符合过滤因子（FF）缺陷的标准（如图 8.10 所示）。当该特定国家只包含 19 张大额发票的情况时才是最差情况。在这种情况下，整个包含大额发票的索引片都必须被扫描，并且每张发票所对应的 CCTRY 值都需要被校验。于是谓词 CCTRY = :CCTRY 的过滤因子变为了 0.1%。此时用程序 B+来构建单屏结果集的响应时间又是多少呢?

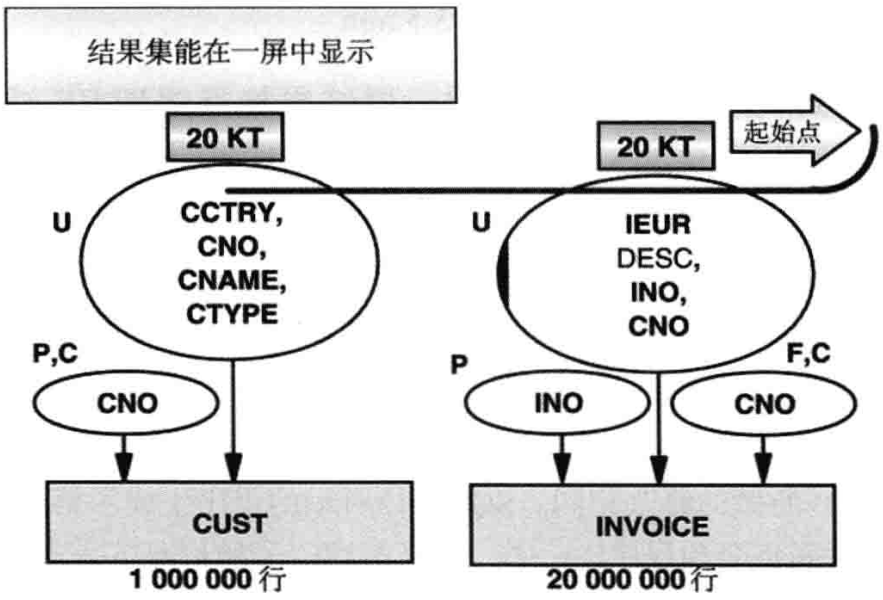


图 8.10　在理想索引条件下遇到过滤因子缺陷的程序 B+

程序 B+：每个事务物化一屏数据

第1步：从索引(IEUR DESC,INO,CNO)上找出所有的大额发票　为了找到某特定国家的所有大额发票，若所有满足条件的数据能在一屏中显示，那么DBMS就必须扫描整个包含大额发票的索引片。让我们把这种情况作为最坏情况的假设——结果集只需一屏即可显示完整。

158

```
索引 IEUR DESC, INO, CNO      TR = 1          TS = 0.1% × 20 000 000
提取 20 000 × 0.1 ms
LRT                           TR = 1          TS = 20 000
                              1 × 10 ms       20 000 × 0.01 ms
                              10 ms + 0.2 s + 2 s = 2 s
```

现在我们失去了限制结果集大小的优势，不过幸运的是损失并不是太大。

第2步：对每张大额发票读取一个索引行（CCTRY,CNO,CNAME,CTYPE）　因为每一张大额发票都必须被检查，所以该步骤需要对索引进行大量的随机访问。

```
索引 CCTRY,CNO,CNAME,CTYPE        TR = 20 000
提取 20 000 × 0.1 ms
LRT                               TR = 20 000
                                  20 000 × 10 ms
                                  = 200 s + 2 s = 202 s
```

本地响应时间

$$2\text{ s} + 202\text{ s} \approx 3.5\text{ min}$$

结论：使用理想索引，每个事务物化一屏结果集且遇到 FF 缺陷

这一情况不免有点让人失望，当结果集能在一屏中显示时，即使采用改进的程序B+，响应时间仍有3.5 min。这和先前的多屏结果集中第一屏的响应时间形成了鲜明的对比，之前的响应时间仅为2 s。现在，理想索引的收益又被内层表索引上大量的TR最小化了，之前遇到的情况又来困扰我们了！

程序 C+：每个事务物化一屏数据

为了使程序C的每个事务只物化一屏数据，我们需要对程序进行修改，这与之前对程序B+所做的修改相同。SQL 8.13所示的程序生成了第一屏的数据。本地响应时间将会和程序B+一样，在高过滤因子的情况下表现良好，在低过滤因子的最差情况下表现非常糟糕。后者通过减少FETCH调用次数节省下了一些时间，但由此带来的收益很小。

SQL 8.13

```
                                        程序 C+ 首个事务
DECLARE CURSORJ CURSOR FOR
SELECT      CNAME, CTYPE, INO, IEUR
FROM        CUST, INVOICE
WHERE       IEUR > :IEUR
            AND
            CCTRY = :CCTRY
            AND
            CUST.CNO = INVOICE.CNO
ORDER BY    IEUR DESC,  INO
WE WANT 20 ROWS PLEASE

OPEN CURSORJ
      FETCH CURSORJ while IEUR > :IEUR and CCTRY =
                                  :CCTRY, max 20
CLOSE CURSORJ

SAVE / INSERT IEUR, INO          the values of the last
     line displayed to user
```

基本连接的问题（BJQ）

这个简单的案例研究表明，即使为最优访问路径使用理想索引，也有可能产生不可接受的响应时间。主要问题在于潜在的对内层表（或其索引）的大量随机访问。包含至少有一张大额发票的客户的索引片在索引(CCTRY, CNO,CNAME,CTYPE)上都是不连续的。 159

在第5章中，我们引入了基本问题BQ：是否存在或者计划设计一个索引，使它包含所有被WHERE子句引用的列。我们观察到：

> 根据我们的经验，在我们遇到的索引问题中，很大一部分发布后才发现的问题可以通过分析基本问题来提早发现。在单表的 SELECT 查询中应用 BQ 方法是很简单直接的。而对于一个连接查询，为了运用 BQ 方法，必须先将其分解成几个单表游标。这是一个相对更复杂的过程，我们将在第 8 章详细讨论此问题。

当然，BQ 背后的原理其实就是，保证只有知道表中的行是必须被访问时，才去访问该表，也许是通过使用索引过滤的方式。我们可以将这个论断扩展到嵌套循环连接中，也就是仅当我们确认内层表（或者索引）的数据是我们所需要的数据时，我们才去访问它。我们的基本连接问题 BJQ 于是变为了：是否有一个已经存在的或者计划添加的索引，包含了所有本地谓词对应的列?这里是指包含了涉及的所有表的本地谓词。

于是，能够消除对内层表或者索引的大量随机访问的唯一方法就是，依

照这一原则添加冗余的列，使一张表上包含所有的本地谓词。本例中，需往INVOICE表上添加一个CCTRY列，用INVOICE.CCTRY = :CCTRY替换CCTRY = :CCTRY。现在就可以创建一个包含所有本地谓词的索引了，如图8.11所示。

160

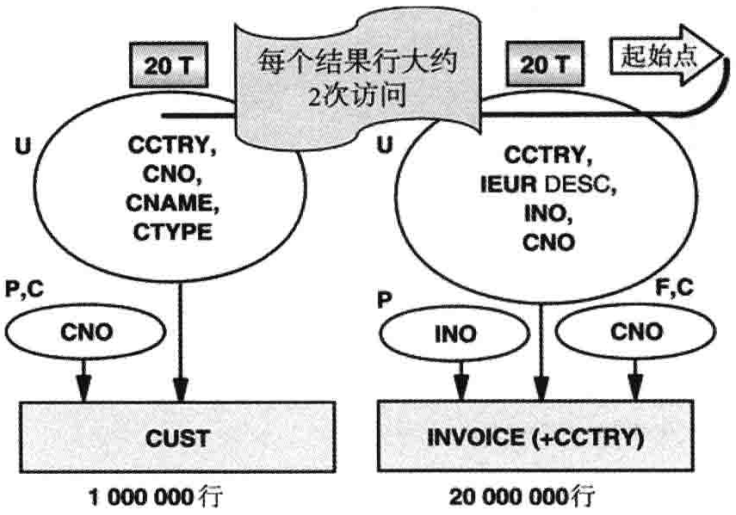


图 8.11　理想索引条件下满足基础连接问题的程序 B+

在一个连接查询被写完或者被生成之后，实施必要的反范式化，同时创建用来维护冗余表数据的触发器（INVOICE表的CCTRY列），从而尽早地应用BJQ或许是一个好主意。

现在，我们的事务无论在什么样的输入下都将运行得非常快了，因为再也不会对索引(CCTRY,CNO,CNAME,CTYPE)有多于20次的随机访问了。

```
索引 CCTRY, IEUR DESC, INO, CNO   TR = 1              TS = 19
索引 CCTRY, CNO, CNAME, CTYPE     TR = 20

提取 20 × 0.1 ms

LRT                               TR = 21             TS = 19
                                  21 × 10 ms          19 × 0.01 ms
                                  = 210 ms + 0.2 ms + 2 ms ≈ 0.2 s
```

给INVOICE表及其索引添加字段CCTRY可能需要180 MB（1.5 × 40 000 000 × 3字节）的磁盘空间，如果一个客户移居到了另一个国家（这样的概率极低），那么一个表行和一个索引行就必须被更新。平均每位用户20行大额发票可能意味着对表的20次随机访问和对索引的40次随机访问，共耗费60×10ms = 0.6 s。

对于从连接角度进行表设计的更深入地讨论，请参考本章最后的“对表设计的思考”小节。

结论：嵌套循环连接

表8.1对本例在不同索引及程序情况下的本地响应时间做了一个总结。

表 8.1　本地响应时间小结—1　161

类型	程序A	程序B
现有索引	35 min	10 min
理想索引	17 min	3.5 min
理想索引，理想程序（+），多屏结果数据		2 s
理想索引，理想程序（+），单屏结果数据（FF缺陷)		3.5 min
BJQ理想索引，理想程序（+），单屏结果数据（FF缺陷）		0.2 s

我们必须再次强调，到目前为止，我们关于连接方法的讨论还远没有完成。

预测表的访问顺序

我们可以看到使用嵌套循环的连接方式时，表的访问顺序可能会对性能产生巨大的影响，索引类似。而在未确定表的最佳访问顺序之前，是无法设计理想索引的。

在大部分情况下，可以使用以下经验法则来预测最佳的表访问顺序：

将包含最低数量本地行的表作为外层表。

本地行的数量是指最大过滤因子过滤本地谓词之后所剩余的行数，如图8.12 所示。

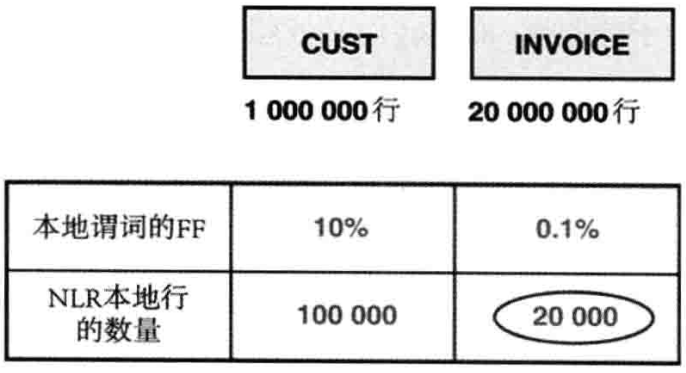


图 8.12　预测嵌套循环连接的表访问顺序

经验法则忽略了以下这些因素。

1. **排序**。在我们的案例中，ORDER BY IEUR DESC 所带来的排序只能够通过把 INVOICE 表作为外层表来避免。而 INVOICE 表正

162

巧含有最低数量的本地行，但是显然，情况并不会总是如此。

2. **很小的表**。非常小的表及其索引可能长期存在于数据库的缓冲池中，至少在一个连接查询中，没有页会被读取多次。在这样的表和索引上进行随机读取所耗费的时间小于 0.1ms，至少在页被第一次读取之后是这样的。所以，当这样的表不是外层表时，对其大量的随机读取也不会成为问题。
3. **聚簇比例**。索引中行的顺序和表中行的顺序的关联性（对于聚簇索引而言，该关联性在表重组后为 100%），可能会影响对最佳访问顺序的选择，当然，除非索引是宽索引。

最好的基于成本的优化器在进行路径选择时会把这些因素考虑进来。因此，它找出的访问顺序可能比我们基于本地行数量的经验法则所得出的结果更优。

读者可能会疑惑，为什么在基于本地行数量的经验法则中使用最大的过滤因子。这样做只是因为如果把过滤因子陷阱的情况包含进来，那么评估过程就太过耗时了。如果有一个可选方法避免排序，且程序不能在一个事务中读取所有的行，那么这种简化无疑增加了不恰当建议的风险。但无论如何，基于假设的访问顺序进行索引设计有可能能够提供足够的性能，即便该访问顺序不是最优的。

当我们为一个连接查询设计索引的时候，我们应该从哪里开始？理论上讲，我们应该为所有可能的访问路径设计最好的索引，然后让优化器去选择。然后，我们再删除无用的索引和索引列。然而这样太耗时间了，尤其是在连接涉及的表多于两张时——另外，优化器所做的选择仍然会依赖于输入的值。基于优化器的索引设计工具解决了第一个问题，但没有解决第二个。它们的方案的质量取决于对过滤因子的假设，而这转而又依赖于谓词中绑定变量的输入值。

幸运的是，在大部分情况下，经验法则能够给出令人满意的结果——而且通常情况下，由于最佳的表访问顺序非常明显，我们甚至无须计算本地行的数量。

关键在于，在为连接查询设计索引的时候，要把所假设的连接方法和表访问顺序记在心里（或者记录下来）。然后，我们就能设计索引了，就像是已经编写好了使用单游标来取代连接游标的程序一样。一种常见的错误是在没有涉及连接方式和表访问顺序的确定的访问路径假设的情况下进行索引设计。仅仅在连接谓词和本地谓词上有索引是不够的，这往往会导致一些冗余的索引。另外，在一个嵌套循环连接中，内层表通常需要有好的宽索引，且以连接谓词列作为前导列。

合并扫描连接和哈希连接

163

当我们分析用例学习中的程序 A 和程序 B 的时候，我们发现，对内层表的大量随机访问导致使用理想索引带来的好处非常有限。合并扫描连接和哈希连接的一个主要的好处就是能够避免这个问题。

合并扫描连接

合并扫描连接的执行过程如下：

- 执行表或索引扫描以找出满足本地谓词的所有行。
- 随后可能会进行排序，如果这些扫描未按所要求的顺序提供结果集。
- 对前两步生成的临时表进行合并。

在以下情况中，合并扫描连接会比嵌套循环快。

1. 用于连接的字段上没有可用的索引。这种情况下，若使用嵌套循环，那么内层表可能需要被扫描很多次。在实际情况中，用于连接的列上面没有索引的情况很少见，因为大部分连接谓词都是基于“主键等于外键”这一条件的，就像我们的案例中的一样——CUST.CNO = INVOICE.CNO。
2. 结果表很大。在这种情况下，若使用嵌套循环连接，可能会导致相同的页被不断地重复访问。
3. 连接查询中不止一张表的本地谓词的过滤因子很低。如我们所见，嵌套循环可能导致对内层表（或者内层表索引）的大量的随机访问。

在通过案例对比嵌套循环连接和合并扫描连接之前，我们先通过一个简单的例子来看看合并扫描连接具体需要执行哪些步骤，以及我们如何使用 QUBE 方法来计算本地响应时间。

例 8.3：合并扫描连接

SQL 8.14

```
DECLARE CURSOR81 CURSOR FOR
SELECT   CNAME, CTYPE, INO, IEUR
FROM     CUST, INVOICE
WHERE    CUST.CTYPE = :CTYPE
         AND
         IDATE > :IDATE
         AND
         CUST.CNO = INVOICE.CNO
```

现在，内层表和外层表的选择变得不那么重要了，尽管术语仍在被使用（如图 8.13 所示）。每一个本地谓词都被依次用来生成包含所需行的临时表。所使用的两个谓词的过滤因子都是 0.1%。对这些访问使用 QUBE 方法分析如下。

164

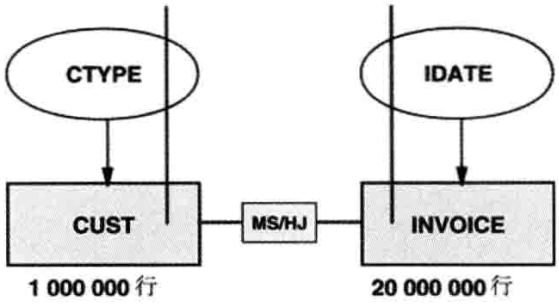


图 8.13　合并扫描连接举例

第 1 步：通过非聚簇索引访问 CUST 表

```
索引 CTYPE                TR = 1              TS = 1000
表 CUST                   TR = 1000

LRT                       TR = 1000           TS = 1000
                          1000 × 10 ms        1000 × 0.01 ms
                          10 s + 0.01 s ≈ 10 s
```

根据 QUBE 方法，进行表扫描也将花费 10s 的时间，同时会消耗更多的 CPU 时间。

第 2 步：通过非聚簇索引 IDATE 访问 INVOICE 表

```
索引 IDATE                TR = 1              TS = 20 000
表 INVOICE                TR = 20 000

LRT                       TR = 20 000         TS = 20 000
                          20 000 × 10 ms      20 000 × 0.01 ms
                          200 s + 0.2 s ≈ 200 s
```

根据 QUBE 方法，进行表扫描也将花费大概 200 s 的时间，同时会消耗更多的 CPU 时间。

第 3 步：合并和 FETCH 结果行　在读取行的过程中，所需要的行会被拷贝至两张临时表中。这一过程的开销很容易被一开始的访问过程所吸收。两张临时表都不是要按照连接列的顺序排序的，因此将不得不对其进行排序。排序的成本依旧是每行 0.01 ms。最终两表将被合并，时间成本同样是每行 0.01 ms。因此：

165

```
为 CUST 表和 INVOICE 表
创建临时表                0 ms
对客户数据排序            1000 × 0.01 ms = 0.01 s
对发票数据排序            20 000 × 0.01 ms = 0.2 s
合并                      (1000 + 20 000) × 0.01 ms ≈ 0.2 s
```

或者简单点来描述：

$$\text{排序和合并：} 2(1000 + 20\,000) \times 0.01\ \text{ms} \approx 0.4\ \text{s}$$
$$\text{提取 } 1000 \times 20 = 20\,000 \times 0.1\ \text{ms} = 2\ \text{s}$$

本地响应时间

$$10\ \text{s} + 200\ \text{s} + 2\ \text{s} \approx 3.5\ \text{min}$$

在DB2 for z/OS数据库中，如果外层表已经是按照连接字段的顺序访问了，那么就没有必要创建和排序临时表了。然而，对于内层表，则总是会被放到一个工作文件中。于是，随着外层表的访问，从外层表上提取的记录将会与内层表的临时表进行合并。

在SQL Server中，合并连接可以是一个常规的或者是一个多对多的操作。后者使用临时表来存储行。如果多张临时表中都存在重复的数据，那么在处理其余临时表中的重复数据时，其中一张临时表上的指针会不停地绕回到重复数据的起始点。

哈希连接

如我们之前所提到的，Oracle、SQL Server及DB2 for LUW都倾向于使用哈希连接（HJ）而非合并扫描连接（MS）。哈希连接本质上是用哈希算法代替排序算法的合并扫描连接。首先，对较小的结果集用哈希算法计算其连接字段，并将其保存在一个临时表中；然后，再扫描其他的表（或索引片），并通过（计算得到的）哈希值将满足本地谓词条件的每一行记录与临时表中相应的行进行匹配。

若结果行集已在索引中满足了所要求的顺序，那么合并扫描的速度将更快。若合并扫描需要进行排序，那么哈希连接的速度可能更快，尤其是当其中一个行集能够全部保留在内存中时（对一个内存中的哈希表进行一次随机访问所花费的CPU时间，通常会比排序和合并一行所花费的时间少）。如果两个行集都很大，那么哈希表会根据可用内存的大小对哈希表进行分区（同一时间内存中只有一个分区），而另一个行集则会被扫描多次。

根据QUBE，无论采用合并扫描还是哈希连接的方式，最初的表或索引扫描都是相同的。任何从哈希计算中获得的时间收益都很难量化，因此，为了避免不必要的复杂性，我们将使用在合并扫描中所用的方法来计算哈希连接的成本开销。

因此，在将来我们没有必要区分合并扫描连接和哈希连接，除非你希望在最后一步（使用散列过程）中节省时间。后续我们将使用术语 MS/HJ 来

166

指代这两者中的任何一个过程。

现在，让我们回到案例学习中，并重新考虑一下程序C。

程序 C：由优化器选择 MS/HJ（在现有索引条件下）

SQL 8.15

程序 C

```
DECLARE CURSORJ CURSOR FOR
SELECT    CNAME, CTYPE,  INO, IEUR
FROM      CUST, INVOICE
WHERE     IEUR > :IEUR
          AND
          CCTRY = :CCTRY
          AND
          CUST.CNO = INVOICE.CNO
ORDER BY  IEUR DESC

OPEN CURSORJ
      FETCH CURSORJ     while  IEUR > :IEUR and
                               CCTRY = :CCTRY
CLOSE CURSORJ
```

若对游标 CURSORJ 中的本地谓词 IEUR > :IEUR 和 CCTRY =:CCTRY 使用我们之前的过滤因子，优化器很可能会选择 MS/HJ，如我们在嵌套循环的讨论中所得出的数字，在现有索引条件下这也许并不是一个坏主意（参见图 8.14）。为了使哈希连接和嵌套循环的对比更易于理解，我们将假设优化器所使用的过滤因子与我们之前所使用的相同。

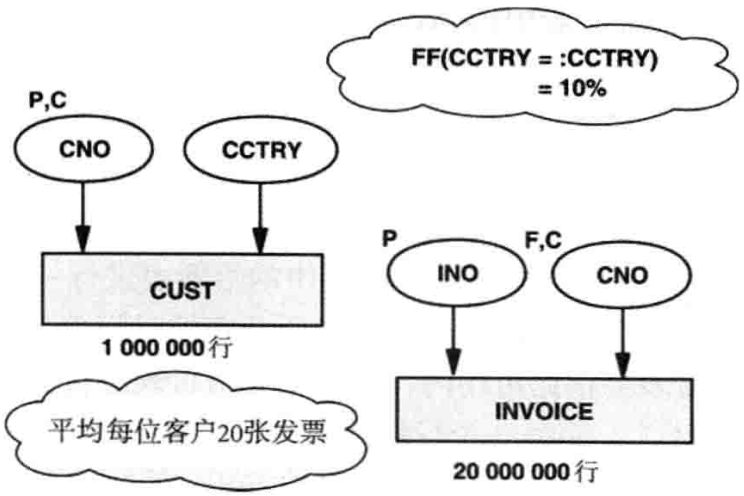


图 8.14　在现有索引下的案例学习

第1步：全表扫描CUST表

表CUST	TR = 1	TS = 1 000 000
LRT	1 × 10 ms 10 ms + 10 s ≈ 10 s	1 000 000 × 0.01 ms

第2步：全表扫描INVOICE表

表 INVOICE	TR = 1	TS = 20 000 000
LRT	1 × 10 ms	20 000 000 × 0.01 ms
	10 ms + 200 s ≈ 200 s	

第3步：进行合并/哈希操作并FETCH结果行　首先，必须对从第1步 167 中得到的100 000行结果数据按CNO进行排序。同样，也必须对第2步中得到的20 000行结果数据按CNO进行排序。然后，再将这两个排好序的表进行合并。在排序和合并的过程中，在每一行结果数据上所花费的时间成本大致是0.01 ms：

排序和合并/哈希	2(100 000 + 20 000) × 0.01 ms ≈ 2.4 s
提取	2000 × 0.1 ms = 0.2 s

本地响应时间　在最差输入情况下，在该访问路径下的响应时间是三部分的和：

$$10\text{ s} + 200\text{ s} + 2.6\text{ s} \approx 200\text{ s} \approx 3.5\text{ min}$$

结论：在现有索引条件下进行 MS/HJ

在现有索引条件下，使用MS/HJ所花费的时间仅为嵌套循环查询的三分之一，当然，在最差输入下这还不够快。同往常一样，现在我们可以开始考虑设计最佳索引了，也许该从设计和评估理想索引开始。

理想索引

SQL 8.16

程序 C

```
DECLARE CURSORJ CURSOR FOR
SELECT    CNAME, CTYPE,  INO, IEUR
FROM      CUST, INVOICE
WHERE     IEUR > :IEUR
          AND
          CCTRY = :CCTRY
          AND
          CUST.CNO = INVOICE.CNO
ORDER BY  IEUR DESC

OPEN CURSORJ
  FETCH CURSORJ  while  IEUR > :IEUR and  CCTRY = :CCTRY
CLOSE CURSORJ
```

对于 CUST 表的索引访问过程而言，理想索引候选 A 为(CCTRY,CNO,CNAME,CTYPE)。这是一个三星索引，这一访问路径保证了结果行是按连接字段排序的，因此结果行不需要再进行排序。对于 INVOICE 表的访问的候选索引 A 为(IEUR DESC,CNO,INO)。这个索引只有两个星，因为范围谓

168 词意味着无法按 CNO 字段的顺序提供结果行。候选索引 B 是 CNO,IEUR DESC,INO，但如此一来，整张 INVOICE 表都将参与合并/哈希过程了，这将导致对如此大量的索引行（FF 为 0.1%）扫描 1000 次。图 8.15 展示了两张表上的候选索引 A。严格地说，在理想索引上，IEUR 列并不是必须按降序排列的。与之前提到的原因相同，我们还是假定该列在索引上是一个降序键。

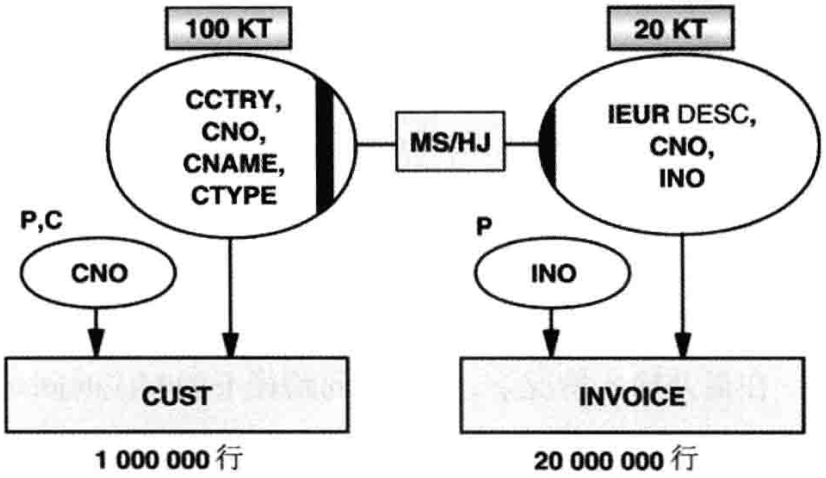


图 8.15　在理想索引条件下的 MS/HJ

第1步：访问索引(CCTRY,CNO,CNAME,CTYPE)

```
索引 CCTRY,CNO,0CNAME,CTYPE      TR = 1          TS = 10% ×1 000 000
LRT                              TR = 1          TS = 100 000
                                 1×10 ms         100 000 ×0.01 ms
                                 10ms+1s≈1s
```

169 **第2步：访问索引(IEUR DESC,CNO,INO)**

```
索引 IEUR DESC,CNO,INO      TR = 1          TS = 0.1%×20 000 000
LRT                         TR = 1          TS = 20,000
                            1×10 ms         20 000×0.01 ms
                            10 ms + 0.2 s ≈ 0.2 s
```

第3步：进行合并/哈希并FETCH结果行　若第1步和第2步需要将结果集按CNO的顺序进行排序，那么第3步的时间成本将会是前2步扫描时间的两倍：

$$\text{排序及合并/哈希}\ \ 2\times(100\,000+20\,000)\times 0.01\ \text{ms}\ \approx\ 2.4\ \text{s}$$

然而，外层扫描的确是按所要求的顺序提供的结果行，因此最终的成本被大大降低了：

$$\text{排序及合并/哈希}\ \ 2\times 20\,000\times 0.01\ \text{ms} = 0.4\ \text{s}$$
$$\text{提取}\ \ 2000\times 0.1\ \text{ms} = 0.2\ \text{s}$$

注意，正是因为第一列CCTRY是等值谓词才能降低这一成本，若第一列为范围谓词，则索引扫描将无法按CNO的顺序访问行。

本地响应时间　在最差输入情况下，此访问路径的本地响应时间是这三者的总和：

$$1s+0.2s+0.6s=1.8s$$

结论：在理想索引条件下采用 MS/HJ

我们将合并扫描的结果也添加到了嵌套循环的对比结果中，见表8.2。

表 8.2　本地响应时间小结—2

类型	程序A	程序B/C NLJ	程序C MS/HJ
现有索引	35 min	10 min	3.5 min
理想索引	17 min	3.5 min	1.8 s
理想索引，理想程序（+），多屏结果数据		2 s	
理想索引，理想程序（+），单屏结果数据（FF缺陷)		3.5 min	
BJQ理想索引，理想程序（+），单屏结果数据（FF缺陷）		0.2 s	

结果很清楚地显示，在理想索引条件下使用 MS/HJ 是最好的访问路径，它不需要我们将策略改为只显示一屏数据或使用 BJQ 方法。采用这一方式能获得很好性能的原因有两个：

170

1. 它避免了我们在嵌套循环连接中遇到的大问题——对内层表或其索引的随机访问。
2. 在当前的硬件条件下，顺序访问的速度已经非常快了。

尽管性能有了巨大的提升，但近2s的响应时间对于一个在线查询而言也许仍旧有点高。若响应时间不可接受，那么就需要设计一个满足BJQ的嵌套循环连接了。

嵌套循环连接 VS. MS/HJ 及理想索引

嵌套循环连接 VS. MS/HJ

若合适的索引已经存在且结果集不是特别大，则我们比较倾向于选择嵌套循环连接（NLJ）这一技术。当然，所谓的“合适的索引”是指相应的半

宽索引、宽索引或理想索引，我们对此已经非常熟悉了。然而，如我们在本章中所见，即便是这些设计得相当完美的索引也可能无法提供好的性能表现，对于内层表（或索引）的大量的随机访问可能会导致大问题。在过去，为了解决这一问题，我们不得不限制结果集的数量，或者运用基础连接问题（BJQ）来保证所有本地谓词都指向一张表，但也有可能这两者都无法令人满意。

在过去的几年中，正如我们在本书中多次提到的，硬件对于顺序访问的性能支持越来越到位。随机访问的性能也有所提升（如更快的磁盘及更大的内存），但它的提升与前者不在一个量级上。通过对比QUBE中的0.01 ms TS与10 ms TR，我们能很清晰地看到这一变化带来的结果。如果将随机访问一行所花费的时间用于顺序访问，则能够访问1000行（忽略额外的CPU开销和其他问题，比如倒推，这将在第15章中讨论）。

这一变化的一个副产品是，如今优化器更倾向于选择顺序扫描（相对于随机访问）了。我们已经看到，即便是在过滤因子低至0.1%的条件下，优化器也可能优先选择进行表扫描，而不是非宽、非聚簇的索引扫描。同样的道理，优化器可能比以前更愿意选择MS/HJ。对于那些长时间从事与关系型数据库相关工作，且并未意识到这些变化意味着什么的人，这些问题尤为重要。比如，我们可能不应该急于强制优化器转而使用索引扫描或NLJ。

当然，如何访问一张表或一个索引，以及应该使用哪一种连接技术，这
171 些是优化器的工作，但是，这些问题同样也会对索引设计过程造成影响，因此，我们需要意识到这一逐渐偏向表扫描及MS/HJ的趋势。

嵌套循环连接 VS.理想索引

不同类型连接的理想索引不会差别太大。NLJ中外层表上的理想索引与MS/HJ中表上的索引是相同的——一个能够提供窄索引片的宽索引。唯一的区别可能存在于NLJ中的内层表上。连接列应当是索引的一部分，且索引最好是宽索引，从而使其能够提供一个非常窄的索引片且不会引入过多的TR。

另外，若使用MS，如果访问路径已按连接的顺序提供表行，那么对外层表的排序可能就不需要了，因此，连接谓词应当包含在索引中，放在任一等值谓词的后面。无须创建、排序并访问工作文件，由此所节省的时间也许会相当可观。

我们知道，在 HJ 中较小的结果集会被保留在一张临时表中，根据连接列进行哈希。随后外层表（或索引片）被扫描，并对每个满足本地谓词的记录进行哈希计算，与临时表中的记录做匹配。

连接两张以上的表

当连接所涉及的表多于两张时，有些连接可能并不是基于常规列（如主键及外键）进行连接的，这并不罕见。在图8.16中，表INVOICE和表ITEM只能通过笛卡儿连接的方式进行连接，这意味着创建一张包含这两表中所有本地行的组合的临时表。在假设的过滤因子条件下，临时表中将会有20 000×1000=20 000 000行（本地谓词没有在图中展示）。由于这是一个代价较高的过程，根据NLR（本地行的数量）所得出的表访问顺序——ITEM,INVOICE,INVOICE_ITEM——可能并不是效率最高的。 172

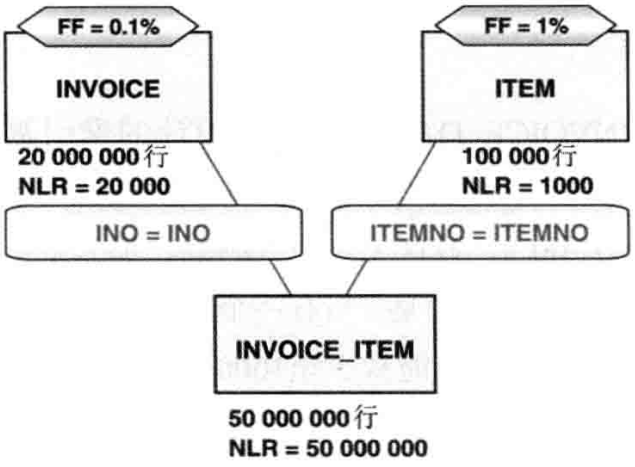


图 8.16　连接两张以上的表

唯一可能的避免笛卡儿连接的方法如图8.17及图8.18所示。图中每个索引上方所显示的访问次数是对该索引的访问次数的总和；每张图的右下角显示了总的TR数——它是响应时间的主要组成部分。所有的访问操作都仅需访问索引。 173

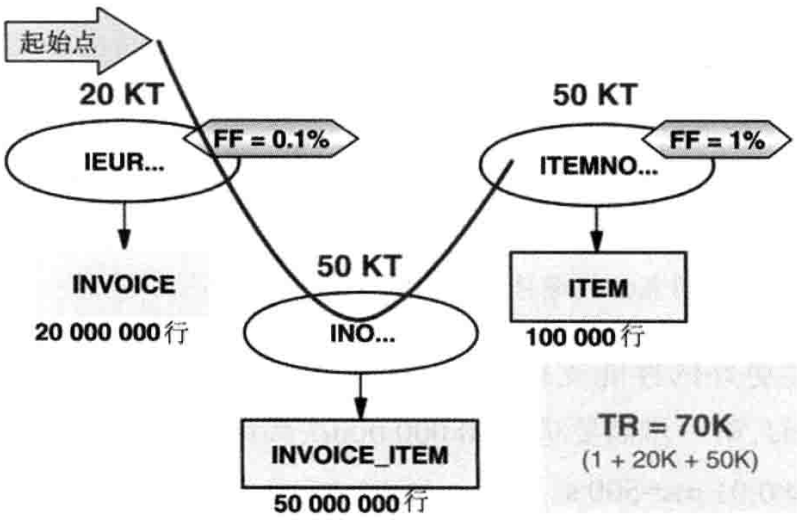


图 8.17　预测表访问顺序—1

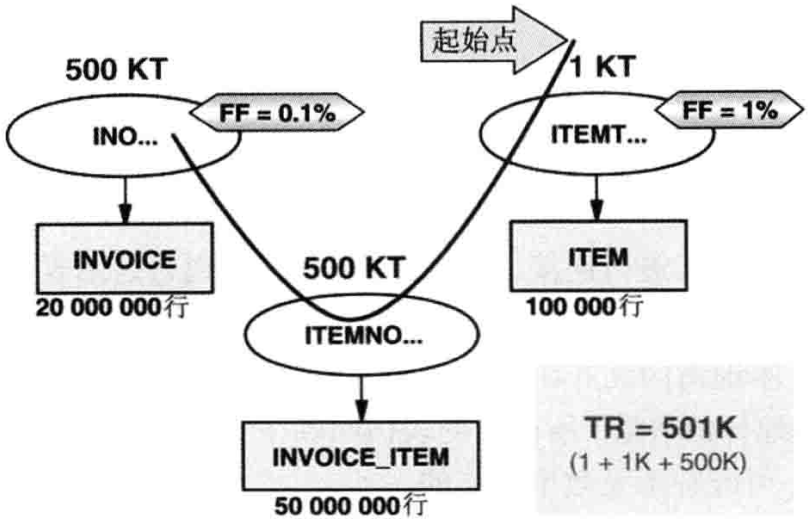


图 8.18　预测表访问顺序—2

为表访问顺序(INVOICE,INVOICE_ITEM,ITEM)所设计的索引提供了一个比原方案更好的访问路径。然而，根据QUBE，70 000次随机访问（参见图8.17）将花费70 000×10 ms=700 s。若这无法令人满意，那么就必须将这些谓词列拷贝至INVOICE_ITEM表上。于是，当有一个包含所有谓词列的索引时，如图8.19所示，随机访问的次数可能减少至1000——这是另一个说明了BJQ重要性的例子。

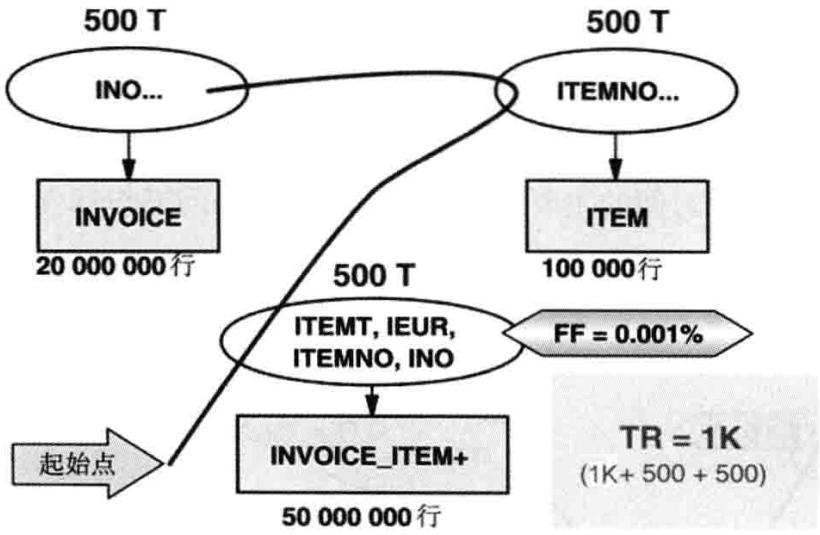


图 8.19　预测嵌套循环的表访问顺序

MS/HJ还能提供更好的性能支持吗？INVOICE_ITEM表上没有本地谓词,因此若使用MS/HJ,第一步需要进行50 000 000次顺序访问,根据QUBE,这将耗费50 000 000×0.01 ms=500 s。

现在似乎采用这样的方式更好：首先使用嵌套循环的方式连接ITEM表

和INVOICE_ITEM表，然后再使用MS/HJ的方式连接中间结果集和INVOICE表（如图8.20所示）。于是，INVOICE表上的理想索引将变为由本地谓词列（IEUR...）作为前导列了。

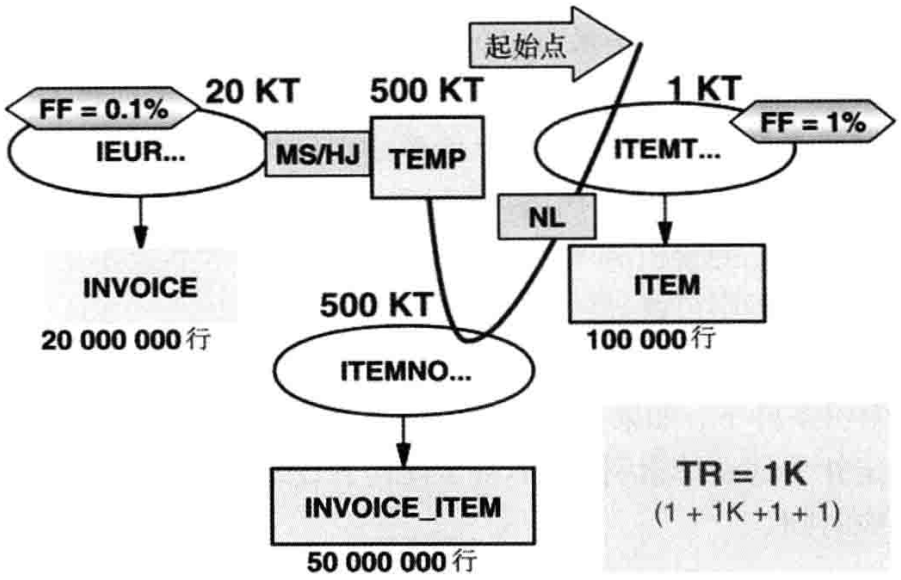


图 8.20　预测嵌套循环连接和合并扫描/哈希连接中的表访问顺序

第1步：连接 ITEM表和INVOICE_ITEM表（NL）

```
索引（ITEMT,…）      TR = 1        TS = 0.01×100 000 = 1000
索引（ITEMNO,…）     TR = 1000     TS = 0.01×50 000 000 = 500 000
```

$$LRT=1001\times 10\ ms+501\ 000\times 0.01\ ms\approx 15\ s$$

中间结果集包含500 000行。

第2步：连接中间结果集和INVOICE表（MS/HJ）

174

```
中间结果集          TR = 1        TS = 500 000
索引（IEUR,…）      TR = 1        TS = 0.001×20 000 000= 20 000
```

$$LRT=2\times 10\ ms+520\ 000\times 0.01\ ms\approx 5s$$

由于使用的是合并扫描的方式，所以必须对520 000行进行排序和合并：

$$2\times 520\ 000\times 0.01\ ms\approx 10\ s$$

哈希连接可能更快，因为20 000行这一较小的结果集能够保留在内存中。

本地响应时间　在这些索引条件下，合并扫描的总响应时间为30 s，哈希连接的总响应时间在20 s ~ 30 s之间——相较嵌套循环（12 min），这是一个很大的改善。

为什么连接的性能表现较差

模糊的索引设计

“在连接字段上建索引”是最古老的索引建议之一。事实上，这是基本建议的一个扩展：“为主键创建一个索引，并且为每一个外键创建一个由此外键作为前导列的索引”。在连接谓词上创建索引使得嵌套循环成为了一个
175 可行的方案，但包含连接谓词列的窄索引并不一定能够提供完全可接受的响应时间。连接谓词列上的索引和本地谓词列上的索引通常都需要是宽索引。而且，不同的表访问顺序可能导致完全不同的索引需求。

根据假定的访问路径创建索引能够增加优化器选择这一访问路径的可能性。在理想索引条件下，如果另一个访问路径能够有更好的性能表现，那么保持原有假定并为其设计索引，比不基于任何假设地随机设计索引更有可能实现短的响应时间。

优化器可能选择错误的表访问路径

一个涉及多张表的SELECT语句比一个仅涉及一张表的语句有更多的可选访问路径。由于优化器错误的过滤因子估算导致的错误访问顺序及错误连接方法并不罕见。另外，对于单表SELECT，访问路径可能对于大多数平均输入是最优的，而对于最差输入是性能较差的。但对于连接查询，不合适的访问路径通常会导致巨大的影响。不过幸运的是，现今已经有很多方法可以协助优化器进行访问路径的选择了——比如通过使用提示。在DBMS发展早期，开发人员有时不得不用多个单表游标替代连接游标，来实现好的表访问顺序或连接方式。

乐观的表设计

本章一直在讨论的程序更像是一个数据仓库的查询，而实际上，由两张大表上的本地谓词导致的此类问题在操作型应用中也很普遍。在最差输入下，即便是最佳访问路径，所耗费的时间也太长了。对于这类问题，索引设计者、应用程序开发者及优化器可能都没有错。甚至将该连接转换为两个游标都无法解决这类问题，而且CPU时间可能还会因此而上涨，因为SQL调用的次数变多了。这类问题可能应当归咎于那些坚信在表中不应该有冗余数据的专家。无论在设计表还是编写连接语句时都应该时刻想到基础连接问题。关于冗余

数据带来的问题将在本章“对于表设计的思考”一节中进行讨论。

为子查询设计索引

从性能的角度看，子查询与连接十分相似。实际上，现今的优化器通常会在进行访问路径的选择之前，先将子查询重写为一个连接。若优化器没有进行重写，那么子查询的类型本身可能就决定了表访问顺序。内外层无关联的子查询通常会从最内层的SELECT开始执行。结果集被保存在一张临时表中，等待下一个SELECT的访问。内外层有关联的子查询通常会从最内层的SELECT开始执行。无论是何种情况，同连接一样，应当基于 176 能够形成最快访问路径的表访问顺序进行索引设计。若最佳的表访问顺序未被选中，那么程序开发人员可能需要对语句进行重写，在某些情况下还可能要使用连接。

为 UNION 语句设计索引

通过UNION或UNION ALL连接的SELECT语句是逐个分别进行优化和执行的。因此，应该为每一个独立的SELECT设计合适的索引。需要注意一点，带ORDER BY的UNION可能会导致提前物化。

对于表设计的思考

在上文中我们提到，许多数据库专家认为在表中有冗余数据总是不合适的。我们还看到，在设计表时需要牢记那些与BJQ相关的问题。现在，让我们来详细讨论一下那些会影响SQL性能的表的设计问题。

冗余数据

有两种通过冗余数据优化连接速度的方法：

1. 将某列拷贝至依赖表（向下反范式化）。
2. 将汇总数据添加至父表（向上反范式化）。

向下反范式化

在我们的案例研究中，我们引入向下反范式化来消除大量对内层表（CUST表）索引的TR。将列CCTRY添加至INVOICE表是一个看起来不错的方案，如图8.21所示。

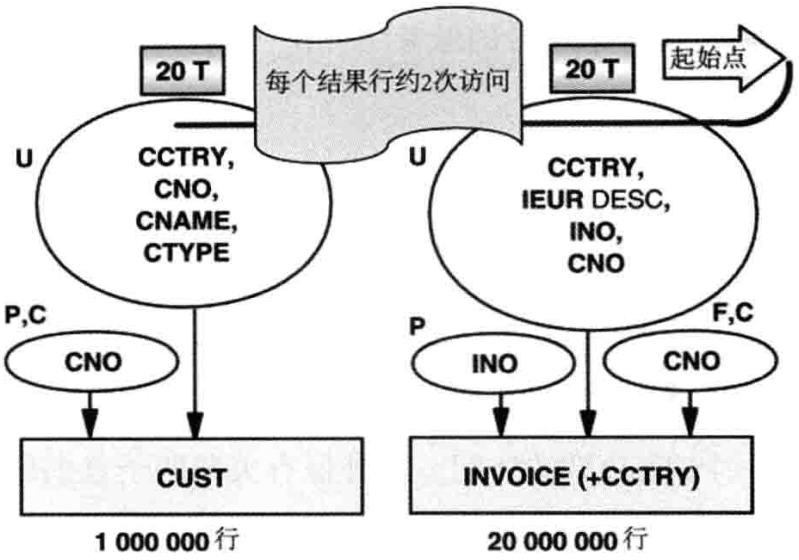


图 8.21　一个满足基本连接问题的嵌套循环连接的理想索引

由此带来的额外的存储成本是较低的，而且当一个顾客搬去另一个国家时，更新INVOICE表的CCTRY列也不是一个大问题。如果一个客户的INVOICE表行是互相临近的（索引(CNO)是聚簇索引），那么这一更新将只涉及一次随机表访问，以及几次索引访问，用于移动索引(CCTRY, IEUR,INO,CNO)上的一些行。当然，如果一个大客户有1000张发票，那么这些索引行移动将导致2000次随机索引访问，花费近20s时间。鉴于这不会经常发生（大客户不会经常搬去其他国家），所以这也许是可以接受的。

177 不过，总体而言，当我们考虑引入向下反范式化时，需要预测一下冗余字段（在我们的例子中是CCTRY）更新时可能导致的最差情况下的索引随机访问次数。

向上反范式化

让我们假设案例研究中的排序需求为ORDER BY CNAME,CNO而非ORDER BY IEUR DESC。此时，若将CUST表作为嵌套循环的外层表便能避免一次排序了。于是，这一访问路径就有可能使一个只做20次FETCH的事务（只提取一屏数据）变得很快了。然而，如果没有对数据进行冗余，假设在这一百万客户中只有0.1%的客户有至少一张大额发票，那么为了在INVOICE表上找到一行满足条件的记录，可能就要涉及1000次索引随机访问——根据QUBE，这需要花费10s。于是，我们需要考虑向上反范式化，以减少为了判断客户是否有大额发票而进行的无用的随机访问次数。

178 最直接的方法是在 CUST 表上添加一个新的列 CLARGE，并在 SQL 上添加相应的谓词条件，如 SQL 8.17L 所示，若 CLARGE=1，则该客户拥有至少一张大额发票。显然，这是一个笨拙的方法，原因有二：

1. 当对 INVOICE 表的数据进行删除或插入操作时，维护 CLARGE 列的工作并不简单。
2. 修改 CLARGE 的定义将带来大规模的批处理任务。

SQL 8.17L

```
WHERE
      CUST.CNO = INVOICE.CNO
AND   CCTRY = :CCTRY
AND   CLARGE = 1
AND   IEUR > :IEUR
```

我们可以往CUST表上添加CTOTAL_IEUR列，即每个客户的发票总额，并在SQL上添加相应的谓词条件，如SQL 8.17T所示。这一方式能够避免上述两个问题，但同时也引入了更高的维护代价：INVOICE表每次INSERT及DELETE操作都将引入两次随机访问，一次是对于CUST表的，另一次是对于包含CTOTAL_IEUR列的索引的。这一方案能够避免一部分对INVOICE表上索引的无用访问，即那些发票总额小于等于:IEUR的客户；而对于那些发票总额大于:IEUR的客户，他们可能有一张或多张大额发票，但也可能只有一些小额发票。

SQL 8.17T

```
WHERE
      CUST.CNO = INVOICE.CNO
AND   CCTRY = :CCTRY
AND   CTOTAL_IEUR > :IEUR
AND   IEUR > :IEUR
```

我们还可以向CUST表添加CMAX_IEUR列，即每个客户的发票中最大的IEUR值（见SQL 8.17M），这样可以完全消除对INVOICE表的索引的无用访问。维护成本在这一方案中也将有所降低，因为只有那些会影响CMAX_IEUR值的发票的插入、删除操作才会导致该列的更新。再次需要注意的是，对CUST表上索引的访问还是无法避免的，因为需要获取该客户最大的IEUR值。

SQL 8.17M

```
WHERE
      CUST.CNO = INVOICE.CNO
AND   CCTRY = :CCTRY
AND   CMAX_IEUR > :IEUR
AND   IEUR > :IEUR
```

179 最佳的方案（见图8.22及SQL 8.17H中所示的SQL语句）是添加CHWM_IEUR列，一个高水位标记，一个客户最近拥有过的最大额的IEUR，但不一定仍然拥有。在此，我们尝试在维护的成本和收益之间达到一种平衡。CHWM_IEUR不一定要非常精确，但它需要足够大，大到能够帮助我们找到所有的大额发票。而如果它的值过高，那么INVOICE表的索引就会被不必要地访问过多次。频繁地更新CHWM_IEUR值将导致高昂的成本，所以对于删除操作而言，只需要通过一个CHWM_IEUR的更新程序，定期对高水位标记进行更新即可。对于插入操作而言，仅当客户的新发票的IEUR值大于其余发票时，CHWM_IEUR列才需要被更新，要完成这一过程，只需在每次INVOICE表INSERT时，对CUST表上的索引进行一次随机访问以比较CHWM_IEUR的值即可。

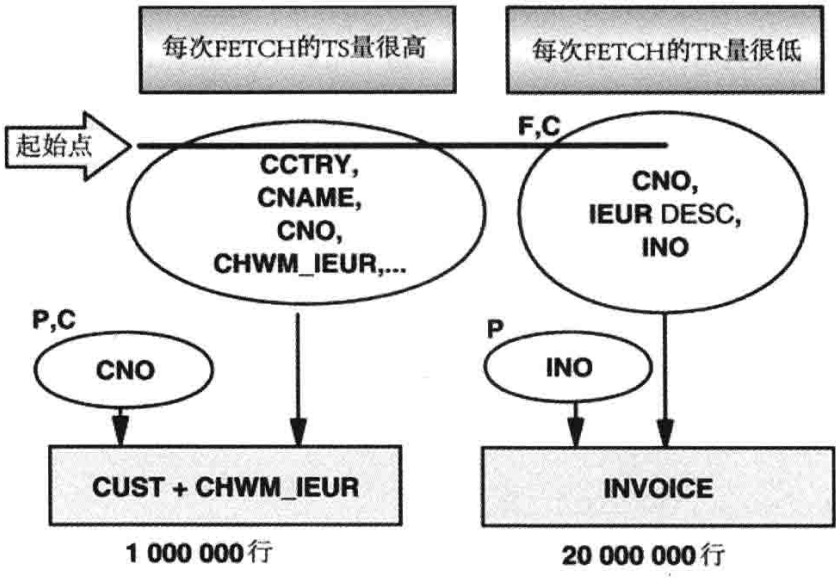


图 8.22　几乎满足基础连接问题的理想索引

SQL 8.17H

```
WHERE
      CUST.CNO = INVOICE.CNO
AND   CCTRY = :CCTRY
AND   CHWM_IEUR > :IEUR
AND   IEUR > :IEUR
```

180 当更新完所有的高水位标记后，平均每次FETCH操作将需要对第一个索引进行1000次顺序访问，再对第二个索引只进行一次访问，通常是随机访问。于是，对于一个只提取20行的事务而言，根据QUBE算法，响应时间应为：

$$20\times10\ \text{ms}+20\ 000\times0.01\ \text{ms}+20\times0.1\text{ms}\approx0.4\text{s}$$

反范式化的成本

考虑性能时最令人关注的通常是，为了更新表及索引上冗余字段所带来的I/O时间。在向下反范式化中，这可能需要移动大量的索引行，从而导致一个简单的UPDATE运行得很慢。向上反范式化不太可能因为一次简单的更新操作而引发I/O剧增，不过INSERT、UPDATE和DELETE可能导致父表及其索引上的一些额外I/O。在极端情况下，如每秒10次以上的INSERT或UPDATE，由这些I/O带来的磁盘负载可能会成为问题。

嵌套循环连接和 MS/HJ VS.反范式化

许多数据库专家不愿意将冗余列添加至事务型表上，这是可以理解的。反范式化不仅仅是查询速度和更新速度之间的一个权衡，在某种程度上，它还是性能和数据完整性之间的一种权衡，即使在使用触发器来维护冗余数据的情况下。然而，当嵌套循环引入了过多的随机访问，且MS/HJ耗费了过多的CPU时间时，反范式化可能成为唯一的选择。尽管如此，在决定采用这一极端的方案之前，我们必须确保所有能够避免这一方案的方法都已经考虑过了。这意味着我们需要确保考虑过为NLJ或MS/HJ设计最佳的宽索引，且将所有可能导致问题的索引保留在内存中；购买更多的内存来缓存索引，或购买能够提供更快顺序读取速度的磁盘，这可能会将使用反范式化的临界值抬得更高。

这毫无疑问是一个艰难的权衡。我们不希望给大家这样一种印象，即非BJQ的查询通常能够通过更快的顺序扫描来解决。在我们的案例研究中，性能数据可能的确反映出这样的情况，但在实际生活中，数据表通常更大，通常包含1亿行以上的数据，且伴有很高的事务频率，另外，CPU时间也是一个非常重要的考量点。

无意识的表设计

从性能的角度看，我们非常难以理解为何有那么多的数据库中存在具有1：1或1：C（C=有条件地；即0或1）关系的表，如图8.23所示。

为何要建四张表而非只建一张CUST表？只要关系永远不会变为1：M，那么灵活性就不会成为问题。在本例中，客户要么是公司，要么是个人，且不会有客户死亡两次！将这四张表合成一张部分字段为空的表（对于每一行，要么公司相关的字段为空，要么个人相关的字段为空；同样，所有活着的客户的死亡相关的字段为空），这是存储空间和性能（随机访问的次数）之间的权衡。为空的数据并不违反范式。

181

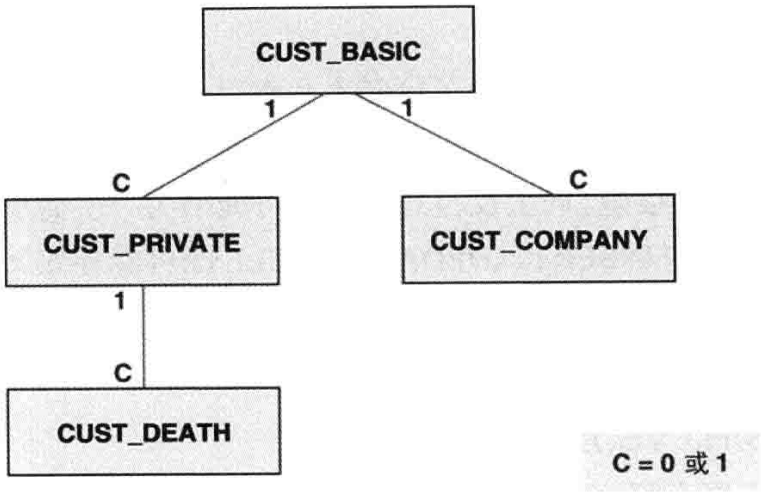


图 8.23　比一张表更好吗?

考虑一个根据某CNO值来查询一些基础字段和公司相关字段的语句。若只有一张CUST表，那么这可以通过一次随机访问来做到；若有多张表，那么至少也要进行两次随机访问。二者的差别可能不只这些，考虑SQL 8.18中所示的查询：

SQL 8.18

```
SELECT      FNAME, CNO
FROM        CUST
WHERE       LNAME = :LNAME
            AND
            CITY = :CITY
ORDER BY    FNAME
```

索引(CITY,LNAME)可能无法提供足够的性能表现，不过我们知道如何仅通过在索引上添加列来使该查询变得很快。图8.24所示的两表解决方案需要进行连接，且它是非BJQ连接。我们已经知道这将导致什么：要么导致大量对内层表（或索引）的无用随机访问，要么由于使用MS/HJ而导致高CPU
182 时间（可能还有排序）。又或者，我们可能需要引入向下反范式化，例如，通过将列CITY拷贝至CUST_PRIVATE表。

在决定拆分一张表之前，需要先衡量一下收益。CUST表所需的磁盘空间为1.5×1 000 000 行×1000 字节/行=1.5 GB。若磁盘空间每月的成本为50美元每GB，那么拆分CUST表所节省的存储成本将小于每月75美元。

有时创建一张独立的表，如 CUST_DEATH，是由于一些程序只对死亡数据感兴趣。处理这样一张表确实会非常快，但这同样可以通过在 CUST 表上设计一个好的索引来实现。

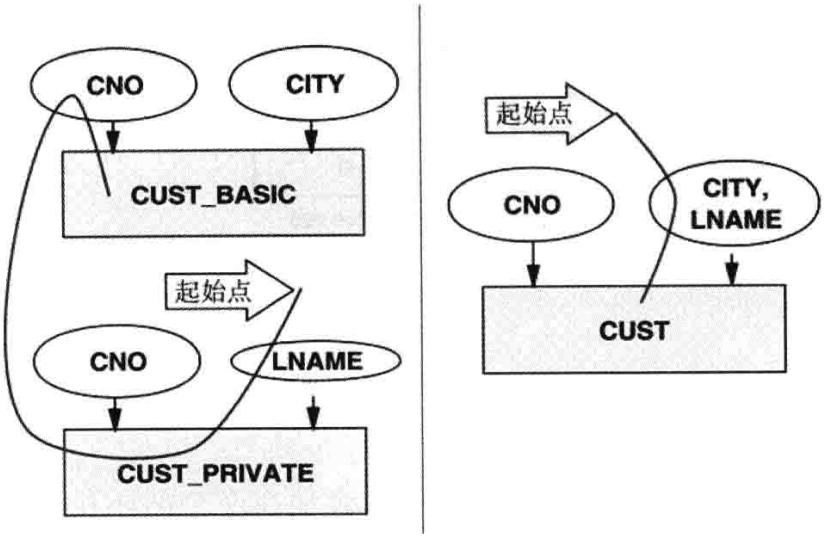


图 8.24　哪种方式性能更好呢?

通常来说，一个数据库包含太多的表仅仅是因为这样看起来更合理，或者仅仅是简单地把对象转换为表而没有进行任何性能上的考虑。一旦应用投入生产，这类错误就很难被纠正了。我最近就遇到了这样一个案例，由于事务会同时访问十几张表，因此即便使用理想索引也无法满足性能要求。最终不得不采用这样的解决方案：将一些数据同步复制到一张新的组合表中供关键事务访问，而其余程序则仍旧继续访问小表。然而，另外一个应用的性能差到了无法正常运行的地步。优化器似乎在支持多于 5 张表的连接时有些问题（但这是相当常见的，事实上有些应用甚至同时访问 20 张以上的表）。而实际上，表结构可以设计得很简单，用更少的表，但这将需要大量的代码重写工作。

在不考虑硬件性能的影响下设计表可能会有如下问题：

1. 即便是在最佳索引条件下，随机访问的次数仍可能会很高。 183
2. 复杂连接可能使得索引设计变得非常困难。
3. 优化器可能对复杂连接做出错误的访问路径选择。

有一个简单的方法可以避免这些问题——对于任何有1：1或1：C关系的表设计，应当由提出该设计的人评估一下其合理性。有时将表拆分是个不错的方案，甚至有时必须这么做，但在当前的硬件条件下这种情况并不常见。主张拆分的人有义务给出足够的理由来支撑拆分方案。

练习

8.1. 评估图 8.25 中所示连接的响应时间，过滤因子使用给定的值。

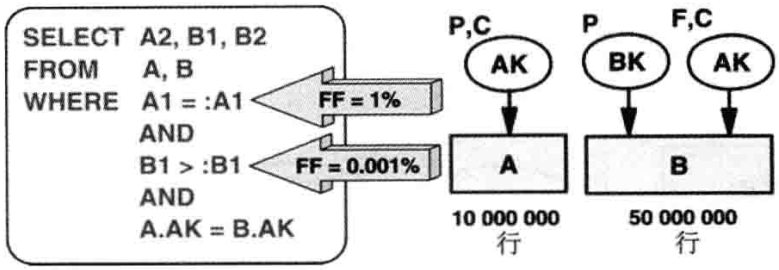


图 8.25　慢连接

8.2. 在不添加冗余字段的前提下，为该连接设计最佳索引并评估响应时间。

8.3. CUST 表中有三个指向代码表的外键。评估在嵌套循环和最佳表访问顺序下，下述这四表连接的本地响应时间（参见图 8.26）。

184

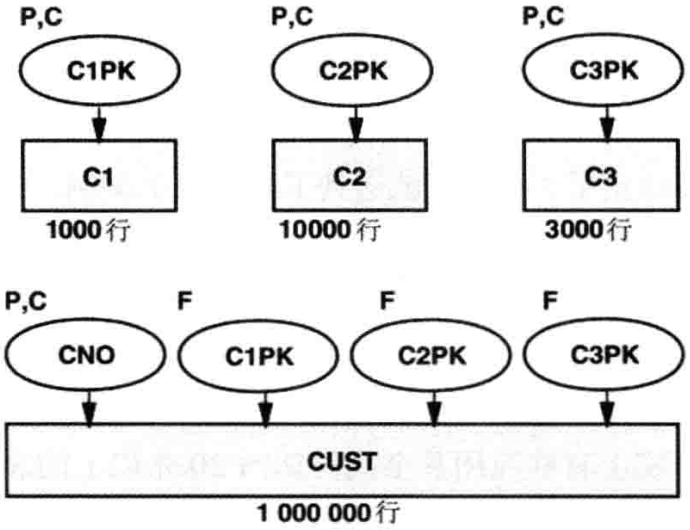


图 8.26　四表连接

8.4. 假设 SQL 8.19 在一个批任务中会被执行一百万次。我们需要改进索引吗？调优的空间有多大？有其他方法可以用来提升该 SELECT 语句的性能吗？

SQL 8.19

```
SELECT      CNAME, C1TEXT, C2TEXT, C3TEXT
FROM        CUST, C1, C2, C3
WHERE       CUST.CNO = :CNO
            AND
            CUST.C1PK = C1.C1PK
            AND
            CUST.C2PK = C2.C2PK
            AND
            CUST.C3PK = C3.C3PK
```

第 9 章

星型连接

- 通过实例介绍星型连接
- 笛卡儿积、维度表和事实表的作用
- 星型连接与普通连接的比较
- 维度表与事实表的索引设计
- 进行笛卡儿连接的表，其不同的访问顺序所产生的巨大影响
- 使用 QUBE 进行的比较
- 星型连接的局限，以及汇总或查询表的使用

介绍

星型连接与普通连接的差别主要有两个方面：

1. 如图 9.1 所示，位于星型结构中心位置的表称为事实表，它的数据量远大于它周围的表——维度表。
2. 最佳的访问路径通常是包含维度表的笛卡儿积，这意味着它们没有相同的冗余列，满足本地谓词的维表数据行都会参与连接。

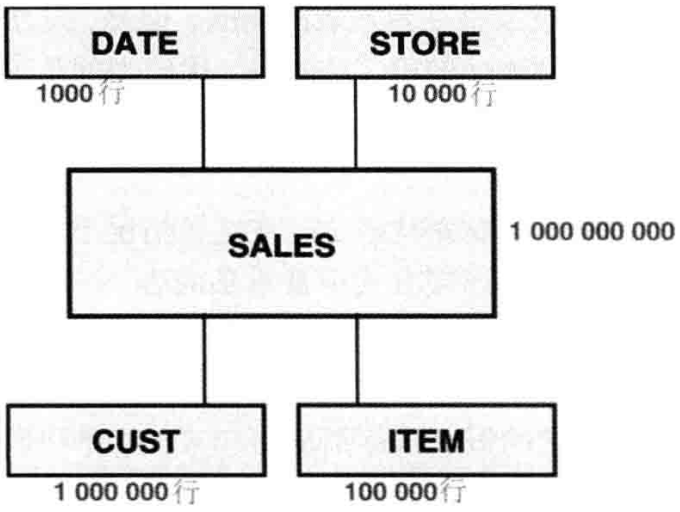


图 9.1　星型模式——一张大的事实表和几张小的维度表

Kevin Viers 评价这种笛卡儿积“听起来比较怪，但却是有道理的”（3，第 611 页）。

图 9.1 所示的事实表 SALES 有 100 万行记录，每行代表一条销售记录。4 张维度表提供了每条销售记录的详细信息：哪家店的哪件商品在何时销售给了哪位客户。为了更好地理解，我们使用 SQL 9.1 展示一个示例，这个示例简化了场景，只使用了除客户信息表以外的三个维度表信息（我们假设并不需要获取客户的信息）。

SQL 9.1

```
SELECT   SUM (EUR)
FROM     SALES, DATE, ITEM, STORE
WHERE    ITEM.GROUP = 901
         AND
         DATE.WEEK = 327
         AND
         STORE.REGION = 101
         AND
         SALES.ITEMPK = ITEM.ITEMPK
         AND
         SALES.DATEPK = DATE.DATEPK
         AND
         SALES.STOREPK = STORE.STOREPK
```

186 这个星型连接非常简单：某种商品在某个地区的周销售总额。ITEM 表中包含产品码，DATE 表中包含日期，STORE 表中包含地区码。

在数据仓库环境中，星型模型（和星型连接）比较普遍，因为终端用户只需要一条简单的 SELECT 语句，就可以从 n 个维度的数据立方体中获取期望的结果集。

传统的连接与星型连接的边界并非很清晰，图 8.26 所示的 4 表连接看起来像是星型连接，尤其是代码表被放置在客户表的周围。但是，这个连接并不是星型连接，因为它不满足星型连接的第二个特征，代码表的笛卡儿积没有任何的意义。按照第 8 章所述的经验法则，CUST 表应该作为最外层的表，剩余其他表的访问顺序影响并不大。

187 在星型连接中，事实表的数据量通常都比较大。在这种情况下，至少依据经验法则，事实表应该作为嵌套循环连接方式中最内层的表。一般情况下维度表都没有共同的列，所以这种连接顺序就意味着是笛卡儿连接。

接下来，我们将讨论影响连接性能的两个主要因素：索引设计和表访问顺序。随着讨论的展开，笛卡尔连接的概念将逐渐清晰起来。这里我们将首先讨论维度表的索引设计，然后讨论表的访问顺序，最后讨论事实表的索引设计。

维度表的索引设计

如果终端用户可以毫无限制地使用谓词，那么维度表上的索引就需要能够满足任意的本地谓词的组合。图 9.2 展示了 4 张维度表之一的客户表。第一版的索引设计可能会是由每一个搜索列组成的宽索引。若统计信息较为完备，比如包含了直方图信息，那么优化器将很可能会使用选择性最好的索引，但在大多数情况下，也只会有一个匹配列。

索引的顺序访问次数等于维度表的行数乘以匹配谓词的过滤因子（FF），比如：

```
WHERE      DOB BETWEEN :BDATE1 AND :BDATE2      FF = 10%
           AND
           SEX = :SEX                           FF = 50%
```

维度表 CUST 的这些本地谓词会产生 0.1×1 000 000=100 000 次对索引(DOB, …)的顺序访问，如图 9.2 所示，因为只有一个匹配列；根据 QUBE 规则，范围谓词 DOB 是最后一个匹配列。根据 QUBE 算法，这一步将花费 1 s 的时间。

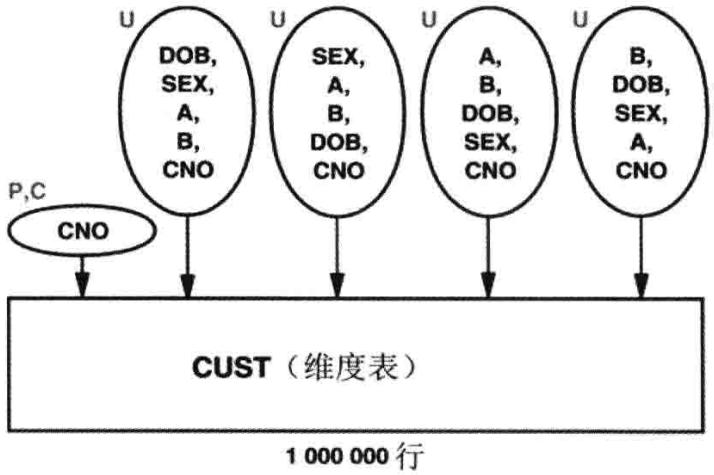


图 9.2　起始点：至少一个匹配列

为了最大化任意 WHERE 条件的匹配列数量，针对 4 个列 DOB、SEX、A 和 B 的查询大约需要 4×3×2×1=24 个宽索引。但事实是，通过对终端用户的调查或通过跟踪用户的查询语句，对于大多数的查询，我们有可能做到使用几个索引就为大部分查询实现多个匹配列。举个例子：一个用户可能对不同年龄段（FF = 10%）的不同性别的客户的销售情况感兴趣，那么索引(SEX,DOB, A, B, CNO)就要优于图 9.2 所示的其他索引，因为相同年龄段相同性别的客户，比如三十多岁的男性，在索引结构上是相邻的。按照 QUBE 算法，两个匹配列的过滤因子是 0.05（不是 0.1），所以扫描这个索引片段只需要 0.5 s。

188

表访问顺序的影响

表的访问顺序在星型连接中比在普通的表连接中更为重要。假设我们现在从 DATE 维度表开始查询，在该表中通过给定的星期号，来确定属于这个星期的所有日期。在 DATE 表中，我们有跨度为三年的记录，每天对应一条记录。我们只对其中的一个星期感兴趣，所以 DATE 表访问的过滤因子就是 7/1000=0.7%——只需访问表中的 7 行数据。按常理推断，如图 9.3 所示，需要访问 0.7%的 SALES 表行，这意味着将进行 7 000 000 次的 SALES 表访问和索引访问（如果索引设计良好，那么将只需进行索引访问）。

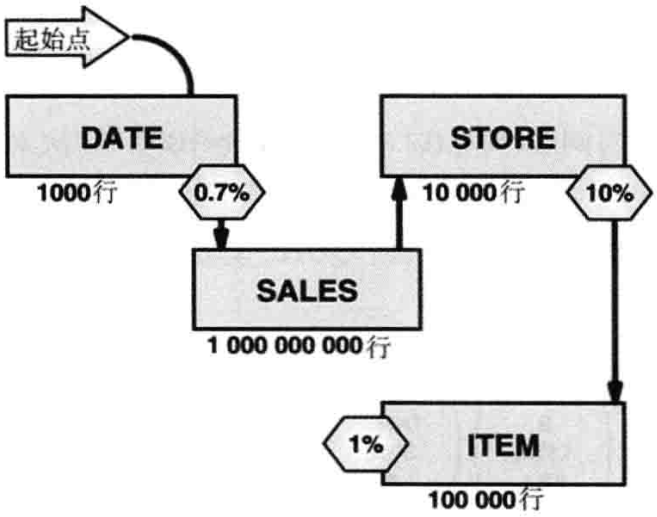


图 9.3　可能的表访问顺序

如果使用 STORE 或者 ITEMS 表来作为驱动表，则相应的表或者索引访问的计数可能更为庞大，所以现在我们来考虑一下在访问事实表 SALES 之前访问所有维度表的可能性，如图 9.4 所示。尽管每个维度表都有本地谓词，但维度表之间并没有关联谓词。

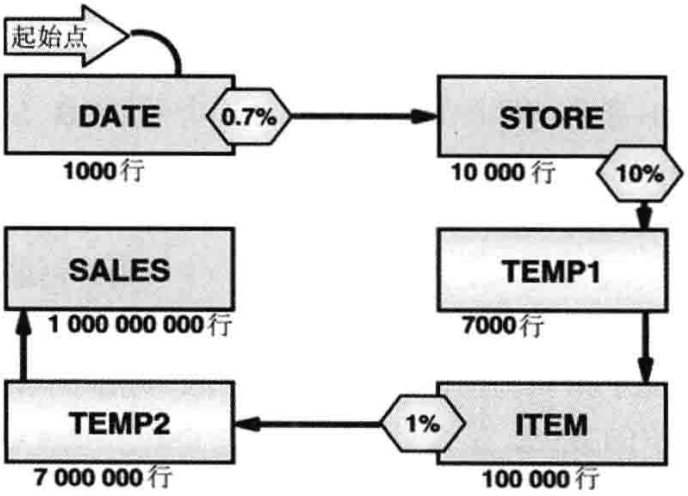


图 9.4　比较好的表访问顺序

如果事实表作为星型连接中最内层的表，那么数据库为这个查询必须建立一张包含所有的有效外键组合的中间表。这个中间表是三个维度表中满足本地谓词的主键的笛卡儿积，如果主键比较短，那么上述例子中的中间表大小可能不超过 100MB，也许能够被直接保存在数据库的缓冲池中。 189

第一张临时表的行数将会是：

$$(0.7\%\times 1000)\times(10\%\times 10\,000)=7\times 1000=7000$$

第二张临时表的行数将会是：

$$7000\times(1\%\times 100\,000)=7000\times 1000=7\,000\,000$$

图表 9.5 显示了第二张临时表的部分片段，即三张维度表 DATE、SORTE 和 ITEM 的笛卡儿积的主键。如之前所计算的临时表，将包含 7 000 000 条记录。

DATENO	STORENO	ITEMNO
.....	.....	.....
764	235	1002
764	235	2890
764	235	4553
764	235	6179
.....	.....	.....

7 000 000条短行

图 9.5　笛卡儿积中间表 2

事实表的索引 190

创建完图 9.4 所示的第二个中间临时表后，数据库系统将根据关联外键组合来读取所有与其匹配的事实表记录。一些情况下，一个组合键会对应多条事实表记录（例如一个商店在一天当中销售了多个同款商品），也可能没有关联的事实表记录。所以，这里不能准确地估算事实表的记录访问次数，但该值可能高达百万。

如图 9.6 所示，我们需要一个设计良好的索引，它将选择性最好的列作为起始列，来满足用户对响应时间的要求。图 9.7 显示了两种理论上可行的扫描方法，通过该图我们可以比较容易地估算出响应时间。

第一种，优化器选择扫描一个单一的索引片段，索引片段的大小只取决

于 DATENO 这一个匹配字段。选择这种方式的原因是我们需要扫描一个时间段的数据，即某一周的七天，比如 DATENO 列的值从 764 到 770。我们之前已经计算过这个索引片段包含 0.7%×1 000 000 000=7 000 000 条记录。

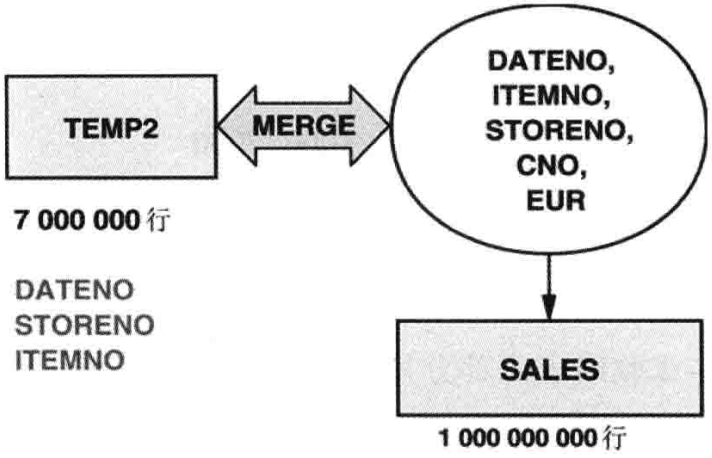


图 9.6　事实表上的宽索引



图 9.7　事实表上的扫描

第二种，“聪明的”优化器可能会把它分成多个子扫描，对于每一个日期（大约有 1 000 000 000/1000= 1 000 000 条记录），基于 ITEMNO 列将有 1000 个（100 000 的 1%）子片段。所以，一个星期将有 7000 个不同的索引片段（两个匹配列），每个索引片段平均为 1 000 000/100 000 = 10 条记录。在这一场景下，厚度为 10 行的 7000 个索引子片意味着扫描 70 000 条记录。

这两种方式的比较结果将是：

1 MC　　1 ×10 ms+ 7 000 000 ×0.01 ms≈ 70 s

2 MC　　7000 ×10 ms+ (7000 ×10 ×0.01)1 ms= 70 s + 0.7 s ≈ 71 s

191　访问事实表之前，中间临时表将通过宽索引进行排序，如果内存有足够

的空间，那么这个操作将非常快。

在给定的过滤条件和假设的满足条件的事实表记录数下，只要优化器选择了相同的访问路径，星型连接的响应时间很可能就是几分钟。这个时间估计包括了访问维度表、构建和读取两个中间表所花费的总时间。

如果只有 4 张维度表，如图 9.8 所示，我们可以使用任意一个维度表的外键组成的索引与事实表进行星型连接。然而，一张拥有 10 亿条记录的表上的 4 个宽索引将非常大，4 个索引的总磁盘空间需求大约为上百 GB 大小。

$$4\times1.5\times1\ 000\ 000\ 000\times40\ \text{byte}=240\ \text{GB}$$

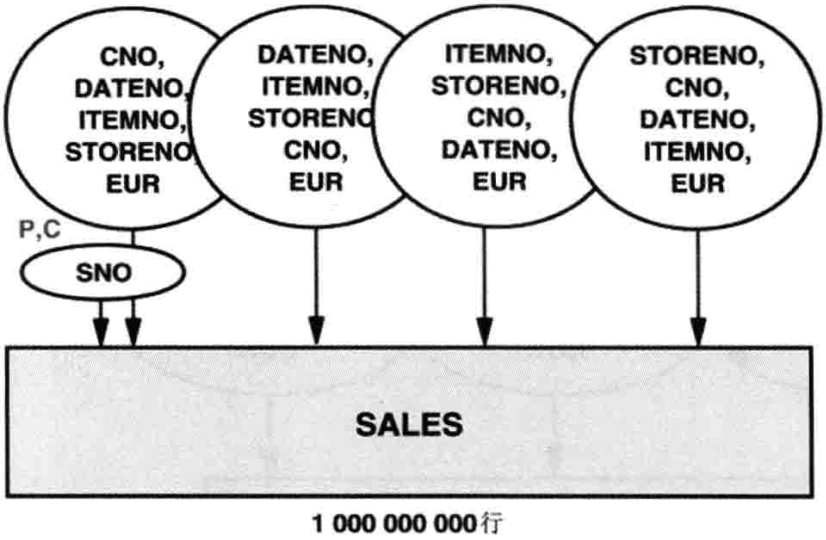


图 9.8　事实表上的 4 个宽索引

假设租用磁盘空间的费用是 50 美元/GB/月，那么 4 个索引每个月共需要 12 000 美元，事实上，宽索引通常比事实表还要大，原因有两个：

1. 表通常都进行了压缩处理，而索引没有。
2. 新增一条记录通常都追加到表的尾部，因此事实表并不需要分散的空闲空间。但插入到索引（除了聚簇索引）上的位置是随机的。为了避免频繁的索引重组，在当前的硬件条件下，通常 10 亿行的表将花费几个小时，大多数的索引在叶子节点上都需要留出足够的空闲空间，可能为 30%~40%。 192

然而，在性能问题上，最大的挑战还是过滤因子和匹配结果集大小的不可预知性。如果针对 ITEM 的查询覆盖了 10%（而不是 1%）的销售记录，那么临时表将会有 7 千万条记录，最终响应时间也将增长到不可接受的长度。如果事实表的结果集大小是 1 亿而不是 1 千万，那么即使索引行只分散在少数几个索引片上，大量的对表的顺序访问也将大大增加响应时间。

汇总表

即便是在理想索引的情况下，一些针对 10 亿条记录的事实表进行的查询也会导致大量的 I/O 访问。提高这类查询的性能的唯一方式就是使用汇总表（查询表）。这类表是反范式化的事实表。如果表不是特别大（比如只包含几百万行数据），那么这是一个比较可行的方案，因为反模式化查询只需针对汇总表，而不再需要多表关联。

如果频繁查询每周的销售情况，那么可以针对每周的消费记录建立一张汇总表。比较好的汇总表设计是根据周、商品、商店汇总一条记录，如图 9.9 所示。在三个宽索引的情况下，查询响应时间通常小于 1 分钟，而汇总表根据周和商品进行汇总后，数据量可以大幅度减小，如图 9.10 所示，在这种情况下，查询的响应时间可能降低到不到 1 秒钟。

193

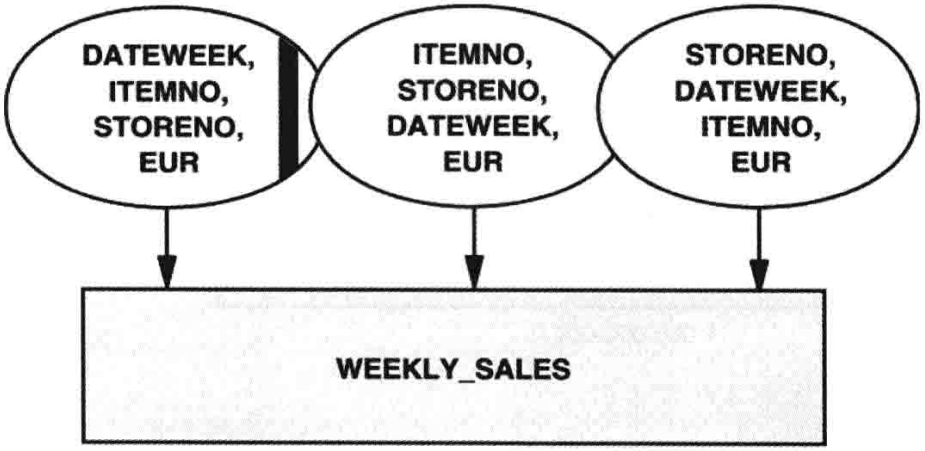


图 9.9　汇总表 Weekly_Sales

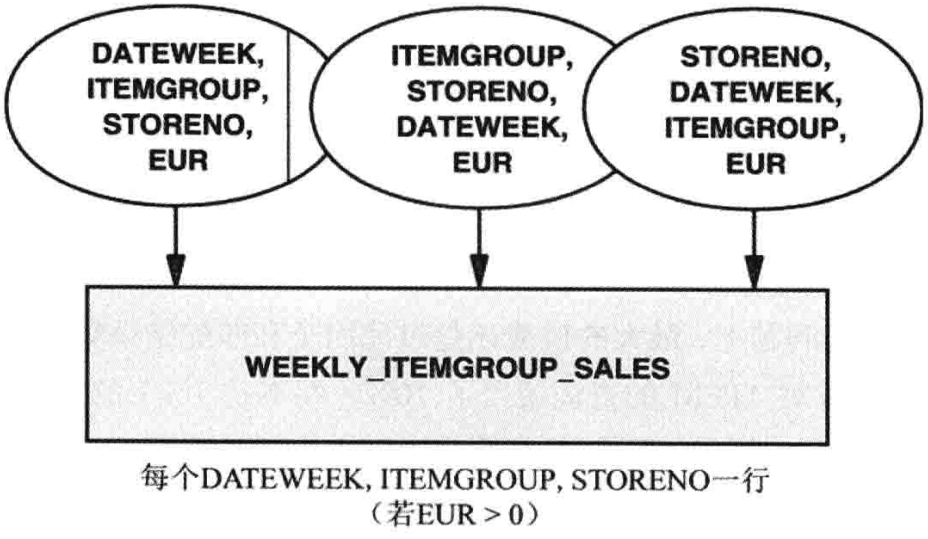


图 9.10　汇总表 Weekly_Itemgroup_Sales

汇总表上的索引通常会比较小，它唯一的限制因素就是刷新此表所需的时间。如同所有的新事物一样，汇总表的方案也会带来新的问题。

1. 如果用户的查询需求多样，汇总表的设计会比索引的设计更困难，不过，已经有一些工具基于查询日志来协助汇总表的设计。
2. 如果优化器不能选择正确的汇总表，那么汇总表的意义就不大；我们不能指望用户来指定查询使用某个合适的汇总表。最好的优化器已经在试着访问合适的汇总表了，虽然 SELECT 语句实际上查询的是事实表。如果优化器不具备这个能力，或者它的正确率不够高，那么可能就不得不强制指定每个用户只能访问一个汇总表。用户将不得不自己选择要使用的汇总表。优化器能够自动识别的汇总表通常被称为*自动汇总表*或者物化视图。

194

第 10 章

多索引访问

- 索引与（AND）
- 缺陷及与使用单个索引的比较
- 与查询表和事实数据表一同使用
- 位图索引
- 索引或（OR）
- 布尔谓词
- 与使用单个索引的情况比较
- SQL Server 索引连接

简介

许多数据库管理系统支持从一张表的多个索引处收集指针，或是从单个索引的几个索引片处收集，然后比较这些指针集并访问满足 WHERE 语句中所有谓词条件的数据行。这一能力被称为*多索引访问*，或被称为*索引与*（索引交集）和*索引或*（索引并集）。本章中我们将思考何时使用这些特性是有益的，以及它们是如何影响索引设计的。

索引与

让我们考虑一下如何使用图 10.1 所示的索引来完成 SQL 10.1 的查询。

SQL 10.1

```
SELECT      B, C
FROM        TX
WHERE       A = :A
            AND
            B BETWEEN :B1 AND :B2
ORDER BY    B
```

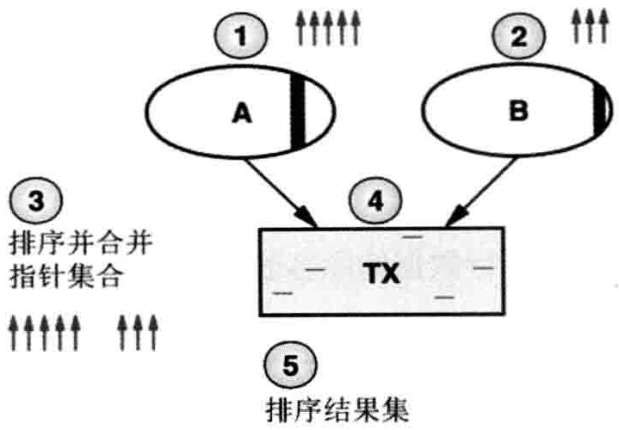


图 10.1　两个窄索引和多索引访问

在高昂的磁盘空间成本导致宽索引不太普遍时，索引与是很重要的。对于必须在索引 A 和索引 B 之间做出选择，而后读取表行以校验其余谓词条件的场景，该特性为其提供了另外一种替代方案。

显然，按如下步骤进行操作，查询的效率将会更高 （其中的数字指代图 10.1 中相应的数字）：

- 从索引片①和索引片②上收集所有满足相应谓词条件的索引行指针。
- 按页号的顺序对两个指针集合进行排序③。
- 合并这两个已排序的指针集合③。
- 只针对那些同时满足两个 WHERE 子句的结果行进行表访问④——每个结果行进行一次表访问；这些表访问将会比传统随机读的方式访问得更快，因为由于指针已经按照页号进行了排序，所以对表页的扫描将通过跳跃式顺序读的方式完成。

那么这一访问路径与单个组合索引(A,B)相比哪个更好？相比多索引访问我们更倾向于后者，因为使用后者对索引的访问次数更少。即便如此，优化器仍然有可能会选择跳跃式顺序读的方式，如果这么做看起来性能不错的话。当然，对该查询来说，一个宽索引(A,B,C)是最高效的方案。

在现今的操作型应用中，索引与并不是一种普遍的访问路径。如果通过 EXPLAIN 发现了索引与，则应当考虑用一个宽索引将其替代，因为使用上述机制进行索引与操作有三个严重的缺陷：

1. 当一个简单谓词有一个较高的过滤因子时，顺序访问的量可能会过多。
2. 即便多个索引片都是从一个符合查询排序顺序的索引上读取的，ORDER BY 子句仍会引起一次排序，因为对指针集的排序操作 197

破坏了从索引继承而来的原始顺序。

3. 如果从索引片上仅仅收集指向表行的指针，那么一个只需访问索引的访问路径是无法实现的。这一数据库管理系统的实现相关的问题将会在后续章节中进行讨论。

让我们回去再看那个从有 100 万行数据的顾客表中寻找身材高大的男士的查询语句，见 SQL 10.2。

SQL 10.2

```
SELECT      LNAME, FNAME, CNO
FROM        CUST
WHERE       SEX = 'M'
            AND
            (WEIGHT > 90
            OR
            HEIGHT > 190)
ORDER BY    LNAME, FNAME
```

如果 CUST 表有三个单列索引（一个在 SEX 列上，一个在 WEIGHT 列上，还有一个在 HEIGHT 列上），并且优化器决定从这三个索引上收集匹配的指针，那么数据库管理系统就必须访问 50 万的 SEX 索引行。并且，针对每一位身材高大的男士都将进行一次表访问。若数据库管理系统依照上述方式进行多索引访问，那么即使将 SEX 索引变宽也无法避免表访问。

与查询表一同使用索引与

如果应用系统会对一个表生成 SELECT 语句，且这些 SELECT 语句带有许多不同且不可预测的 WHERE 子句，那么此时索引与是一个很有用的功能。

正如我们在第 9 章中所见，查询表是对查询进行优化了的事实数据表。它们在表的行数不是特别多时是可行的，比如几百万行数据。正是由于对查询进行了优化，所以不再需要进行表连接，所有查询语句所需的列都在一个表中了。图 10.2 中的 CUST 表即是一个典型的例子。该表可能有几十个列，即便压缩之后行长仍有几百字节。

图 10.2 展示了 5 个搜索列，即最终的 WHERE 列，假设顾客号不作为搜索列。这 6 个窄索引连同多索引访问方式一起，能为任何使用这 5 个搜索列的复合谓词查询提供足够的性能表现。如果扫描的索引片不多于几十万行，那么收集并对指针进行排序所花费的时间是可以承受的。访问表行往往是响应时间中最大的组成部分：数据库管理系统可能需要读取彼此之间有一定距

离的上千行表行。而宽索引能避免这些表访问。虽然设计宽索引与为维度表创建索引类似——我们在第 9 章中讨论过(最小量为每个搜索列一个索引), 不过宽索引的索引行长度可能会是个问题。 198

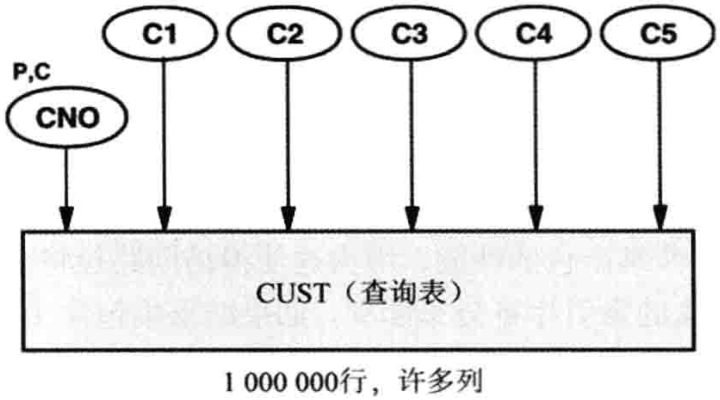


图 10.2　6 个窄索引——对于任何 WHERE 子句都能提供足够的性能

多索引访问和事实数据表

也许在一个大的事实数据表上创建多个窄索引、主键索引及外键索引的方案看起来不错。这样将会比第 9 章中推荐的宽索引耗费更少的磁盘空间。然而，不幸的是，如果该表有 10 亿行数据，那么一个基于 B 树索引的多索引扫描，连同大量的指针扫描和排序一起，在当前的硬件条件下可能会运行得太慢。

用位图索引进行多索引访问

利用位图索引进行索引与和索引或操作速度更快。对于在超大表上进行不可预测的不回表查询（如 SELECT COUNT），或是只需少量表访问的查询而言，位图索引比传统的 B 树索引效率更高。按照 Sasha 和 Bonnet（7，第 281 页）所说，CIA 是位图索引的首批用户之一：

> 位图索引的比较优势在于能够很容易地使用多个位图索引来满足单个查询。考虑一个有多个谓词条件的查询，每个谓词上都一个索引。虽然有些系统可能尝试对多个索引的记录标识进行交集操作，但是传统的数据库可能会只使用其中的一个索引。位图索引在此种情况下工作得更好，因为它们更紧凑，而且计算几个位图的交集比计算几个记录集合的交集更快。在最好的情况下，性能的提升空间与机器的字长成比例，因为同一时间两个位图能够进行一个字长的位的交集计算。最佳的使用场景是，每一个单独谓词的选择性不好，但是所有谓词一起进行索引与后的选择性很好。例如，考虑如下查询，“找出有棕色头 199

发，戴眼镜，年龄在 30 岁至 40 岁之间，从事计算机行业并居住在加利福尼亚的人”。这意味着对棕色头发位图、佩戴眼镜的位图、年龄在 30 岁至 40 岁间的位图等进行交集计算。

在当前的磁盘条件下，只要查询中没有太多的范围谓词，使用一个半宽 B 树索引是性能最佳的方案，即便对于像 CIA 那样的应用来说也是如此。对于上文中的例子，一个用 HAIRCOLOUR、GLASSES、EYECOLOUR、INDUSTRY 和 STATE 的任意排序序列作为开头，并以 DATE_OF_BIRTH 作为第 6 列的索引将提供非常出色的性能，因为这使得访问路径将会有 6 个匹配列：包含目标结果集的索引片将会非常窄。如果结果集包含 100 行，那么根据 QUBE 计算出的本地响应时间将会是 101 × 10 ms + 100 × 0.01 ms + 100 × 0.1 ms≈1 s。

读取又长又高压缩的位图向量可能会消耗更多的 CPU 时间。另外，单行的插入、更新及删除操作将会由于需要锁住位图索引而变得很慢且具有破坏性。但从另一方面来说，当潜在的可能用到的 WHERE 子句数量变大时，为每一个 WHERE 子句都设计一个好的 B 树索引将变得不可能。

索引或

索引或是比索引与更重要的一个特性。

在多索引访问功能被引入作为优化器的一部分之前，对于 SQL 10.3 中所示的 SELECT 语句，优化器将会选择哪种访问路径呢？如果该表有一个单列索引 A 和一个单列索引 B（参考图 10.3），那么优化器只有一个合理的选择，即全表扫描，因为这两个谓词都是非布尔的——如第 6 章中所讨论的，数据库管理系统并不能因为其中一个谓词被检查出不满足条件而排除该行。

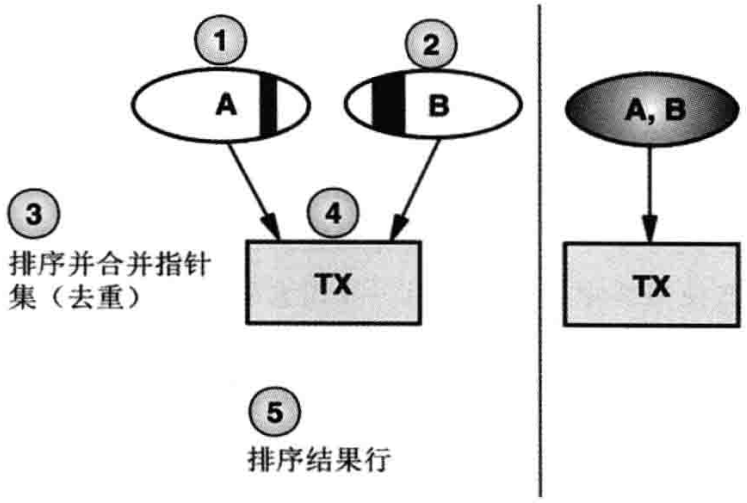


图 10.3　索引或是个好消息

200

SQL 10.3

```
SELECT      B, C
FROM        TX                              1,000,000 rows
WHERE       A = :A                          FF = 0 ... 0.01%
            OR
            B BETWEEN :B1 AND :B2           FF = 0 ... 0.05%
ORDER BY    B
```

即使使用一个宽索引(A,B,C)也不会有太大的帮助，因为数据库管理系统必须扫描整个索引，这涉及一百万次的顺序访问。

在使用索引或时，会进行如下操作：

- 采集两个指针集①和②——100 次顺序访问 A，500 次顺序访问 B，在最差输入条件下合计共 600 次访问。
- 对指针进行排序③——非常快。
- 去除重复的指针③——非常快。
- 读取满足条件的表行④——在最差输入且没有重复指针的情况下，共 600 次随机访问，600×10 ms = 6 s；相较之下，全索引扫描花费的时长为 1 000 000×0.01 ms = 10 s。

之前某段时间，一个银行在其编程规范中禁止在 WHERE 子句中使用 OR，尽管优化器在那时已能够支持多索引访问。当然，一个程序员可能并不知道非布尔谓词的影响，但是也存在 WHERE 子句中有多个 AND 和 OR，却在好的索引条件下运行良好的场景：没有多索引访问，没有排序，无须表访问。在快速 EXPLAIN 复查的过程中，我们应当对 SELECT 语句进行多索引访问检查；不合适的索引或者有害的 OR 可能会被识别出来，这些识别出的问题必须用 UNION 来替代，或者对游标进行拆分。

索引连接

SQL Server 的优化器能够生成一种名为索引连接的访问路径，该种访问路径与上文所述的实现方式有所不同。

以下是 Kevin Viers（3，第 611 页）关于 SQL Server 索引连接的描述。

> 另一个在单表上使用多个索引的方法是连接两个或多个索引以创建一个覆盖式索引。提醒一下，覆盖式索引是指涵盖给定查询中所有列的索引。考虑如下示例：

```
SELECT OrderDate, ShippedDate, count(*)
FROM Orders
```

```
GROUP BY OrderDate, ShippedDate
```

201

再提一下，Orders 表在 OrderDate 列和 ShippedDate 列上都有索引。在本例中，优化器将使用合并连接的方式连接这两个索引从而返回合适的结果集。通过连接这两个索引，SQL Server 达到了与拥有一个覆盖索引相同的效果。

由于从索引片上收集的不仅有指针，所以索引连接的实现方式避免了多索引访问的第三个缺陷——不必要的表访问。然而，第一个缺陷，宽索引片，以及可能发生的第二个缺陷，不必要的结果集排序，依然存在。为了同时避免这三个缺陷，需要有一个真正的宽索引，对于这个查询而言，宽索引将会是(OrderDate, ShippedDate)。

练习

10.1. 假设多索引访问一节中所描述的拥有位图索引的 CIA 表包含 200 000 000 行数据。请评估（a）位图索引和（b）半宽 B 树索引所需的磁盘空间。

假设一个字节占 8 位。请将磁盘空间的差异转化为每月需要支付的美元金额。

第 11 章

索引和索引重组

- 索引的物理结构
- 数据库管理系统如何访问索引
- 数据插入对索引的影响
- 索引叶子页的分裂
- 索引重组
- 索引分裂对索引扫描的影响
- 索引叶子页的分裂比例及其二项分布预测方法
- 插入模式
- 索引末页、随机插入和热点问题
- 空闲空间的推荐值，以及如何确定重组频率
- 一些特殊案例
- 变更频繁的索引列及较长的索引行
- 一个顺序混乱的索引和表对大的批处理程序的影响
- 在 SQL Server 和 Oracle 中，当表行存储在叶子页中时，叶子页分裂的影响
- 索引重组的成本
- 索引分裂的监控

B 树索引的物理结构

在前面的章节中，索引被描述为一个表，该表包含了其所属表的列的子集，连同指向表行的指针一起。我们假设 DBMS 可以直接定位到匹配列所定义的索引片的第一行，然后继续扫描，直到发现一行不满足匹配列条件的索引行时停止。之前的图表中总是显示一个索引行对应一个表行，即便当索引键值不唯一时也是如此。而且，索引行总是以索引列所定义的顺序来显示的。当通过计算访问次数来评估索引时，在脑海中有这么一个图是非常有用

204 的。不过，现在是时候讨论 B 树索引真实的物理结构了。

在 20 世纪 60 年代，索引文件和数据库系统的索引均被构造成平衡树。图 11.1 中所示的是一棵被横着放置的平衡树，其中根页被放置在左边，叶子页被放置在右边。

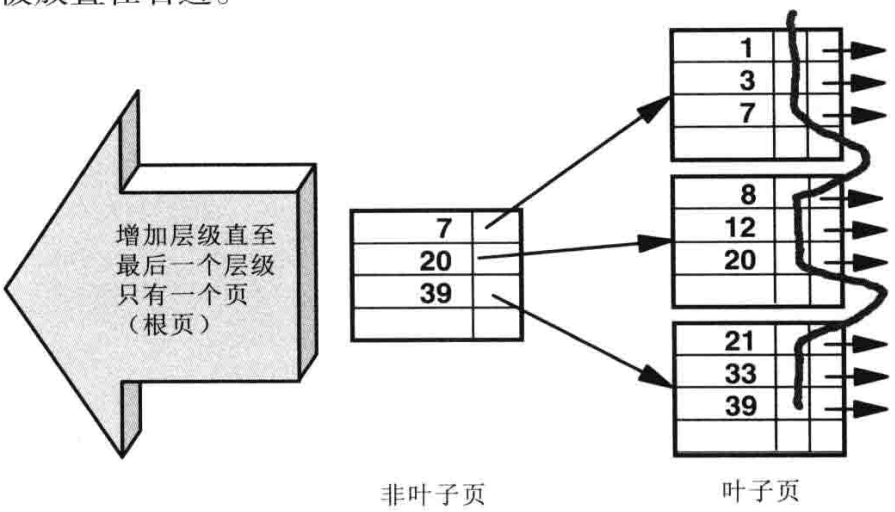


图 11.1　一个二层的 B 树索引

索引行被存储在索引的叶子页上。如今，典型的索引页大小为 4 KB 或 8 KB。索引行的长度一般为索引列的总长度外加约 10 个字节的控制信息。如果索引行的长度为 200 字节，并且索引为唯一索引，那么一个 8KB 的叶子页大约可以存放 40 个索引行。如果是非唯一索引，每个索引键值后可能会存有多个指针，在许多 DBMS 产品中，这些指针会以其值的顺序来存储。这样，即使同一个索引键值对应 100 万条记录，DBMS 也能够快速地找到那个需要删除的指针。我们在这里特别提及这一点是因为一些 DBA 会担心（这种担心是由于历史原因）最大指针链的长度对删除性能的影响。

DBMS 如何查找索引行

假设优化器决定使用图 11.1 中的索引来执行 SQL 11.1 中的 SELECT 语句。

SQL 11.1

```
SELECT   COLX
FROM     TABLE
WHERE    COLI = 12
```

205 DBMS 首先通过内部系统表找到指向根页的指针，在读取根页之后，DBMS 知道需要读取第二个叶子页来查找与键值 12 相关联的指针。

一次随机读取会从磁盘上读取一个页。如果系统表中那些用于定位根页的表页已经在数据库缓冲池中，那么这个查询可能需要进行三次随机读：一次根页、一次叶子页及一次表页。然而，在当前的硬件条件下，非叶子页很有可能已被缓存在数据库的缓冲池中，或者至少在磁盘的读缓存中，因为它们经常被频繁地访问。所以，很有可能只会产生两次随机读，该查询的同步 I/O 时间约为 2 × 10 ms = 20 ms，在当前的处理器条件下，CPU 时间相对小得多，所以该查询的耗时很可能就在 20ms 左右。

当数据库需要读取一个索引片时，如上所述它会先读取第一个索引行。第一个索引行有一个指针指向下一个索引行，DBMS 会一直沿着这个指针链访问，直到遇到一个不满足匹配谓词的索引行为止。所有的索引行都通过索引键值链在一起。

在SQL 11.2中，SELECT语句所要查找的是LNAME以字符串“JO”开始，且CITY以字符串“LO”开始的记录。在这一场景下，DBMS会首先在键值JO后追加字母“A”至最大长度，如JOAA...AAA，尝试用该值在索引(LNAME,CITY)上找到一行记录。

SQL 11.2

```
DECLARE CURSOR112 CURSOR FOR
SELECT      CNO, FNAME
FROM        CUST
WHERE       LNAME BETWEEN :LNAME1 AND :LNAME2
            AND
            CITY BETWEEN :CITY1 AND :CITY2
ORDER BY    FNAME
```

当找到该键值在索引上的位置后，DBMS 随之通过链表读取后续的索引行，直至找到一个不以“JO”开始的索引行为止。

插入一行时会发生什么

如果一张表有一个聚簇索引，那么 DBMS 会根据聚簇索引的键值尝试将插入的记录放在它所属的表页(主页)中。如果这行记录在主页里放不下，或者当前页被锁住，那么 DBMS 将会检查邻近的页。在最坏的情况下，新的行会被插入到表的最后一页。依赖于 DBMS 和表的类型，已经插入的行通常都不会被移动，否则这将意味着更新表上已经建立的所有索引上的相关 206
指针。当有许多表行未能存放在主页中时，如果表行的顺序很重要，则需要对这个表进行重组——对于那些涉及多张大表的大规模批处理任务而言，通常需要这么做。

如图 11.2 所示，当在表中插入一行数据时，DBMS 会尝试将索引行添

加至其索引键所属的叶子页上，但是该索引页可能没有足够的空闲空间来存放这个索引行，在这种情况下，DBMS 将会分裂该叶子页。其中一半的行将被移动到一个新的叶子页上，并尽可能地靠近被分裂的页，但是在最坏的情况下，这个索引页可能会被放置在索引的末尾。除了在每个叶子页上预留部分比例的空闲空间外，也许可以在索引被创建或者重组时，每 *n* 个页面预留一个空页——当索引分裂无法避免时，这会是一个不错的办法。

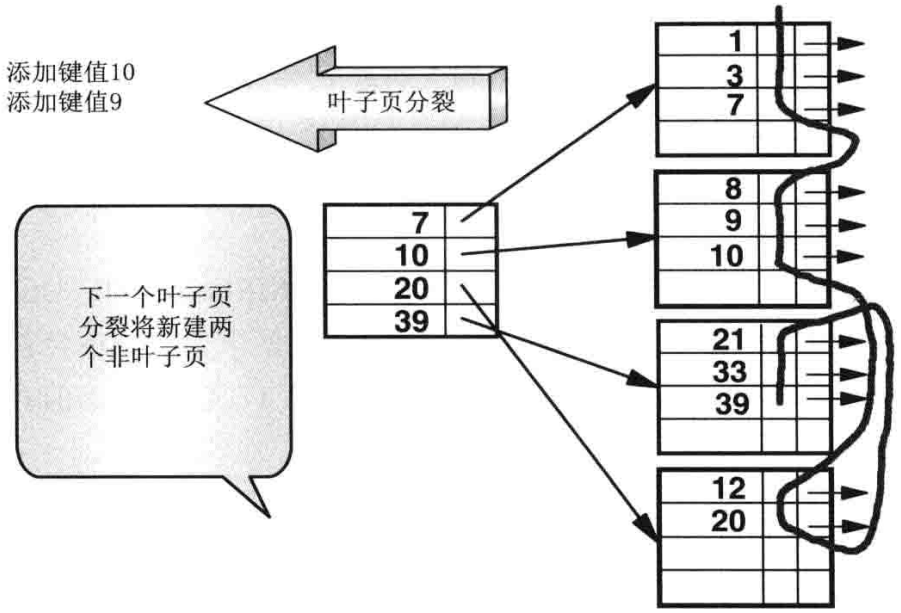


图 11.2　叶子页分裂是一个很快的操作

当一个索引有一个不断增长的键值时，新行将被添加到索引页的最后，索引页可能永远也不会进行分裂。这样的索引可能不需要任何空闲空间。

图 11.2 显示了一个在增加了 2 行记录以后的索引。其中第一个索引键值 10 被存放在了合适的索引页中，此时还有可用的空闲空间。而第二个索引键值 9 则导致了索引叶子页的分裂。任何下一次的叶子页分裂都将需要创建第二个非叶子页，随之也将需要创建一个层级为三的索引页。

叶子页的分裂严重吗

分裂一个索引页只需一次额外的同步读，约 10 ms。除了两个叶子页以外，DBMS 通常还必须更新一个非叶子页，而它很可能已经在内存或者读缓存中了。

207 在叶子页分裂后，查询任何一条索引行的速度很可能同之前一样快。在最坏情况下，一次分裂会创建一个新的索引层级，但是如果不是从磁盘读取非叶子页的话，这只会增加很少的 CPU 时间。

然而，叶子页分裂会导致读一个索引片变得更慢。目前为止，我们在所有的场景中都假设串连索引行的链指针总是指向同一页或者下一页，这些页可能已被 DBMS 预读取。在索引被创建或者重组后，这种假设是接近真实情况的，但是索引片上的每处叶子页分裂都可能会增加额外的两次随机访问——一次是为了查找索引行被移动至的索引页，一次是为了返回到扫描的原始位置。其中第一次随机访问很可能会导致一次磁盘的随机读取（10 ms）。

假设我们需要扫描 100 000 个索引行，根据 QUBE，这将花费

$$1 \times 10\ \text{ms} + 100\,000 \times 0.01\ \text{ms} \approx 1\ \text{s}$$

当每个叶子页包含 20 个索引行（不存在叶子页分裂）时，DBMS 必需读取 5000 个叶子页。如果这些页是相邻的，并且顺序读的速度为 0.1 ms 每页（40MB/s，页大小为 4KB），那么 I/O 时间为：

$$10\ \text{ms} + 5000 \times 0.1\ \text{ms} = 510\ \text{ms}$$

如果 1%的叶子页发生过分裂，扫描 100 000 个索引行可能需要进行 50 次随机 I/O（5000 的 1%）。即便是这样很小比例的分裂，I/O 的时间也增加了一倍：

$$51 \times 10\ \text{ms} + 5000 \times 0.1\ \text{ms} = 1010\ \text{ms}$$

在叶子页分裂后，计算索引片扫描的 I/O 时间的通用公式为：

$$(1 + (\text{LPSR} \times A/B)) \times \text{ORIG}$$

其中 LPSR = 叶子页的分裂比率（分裂的数量/叶子页的总数量）

A = 单次随机读的 I/O 时间

B = 单次顺序读的 I/O 时间

ORIG = 无叶子页分裂时的 I/O 时间

这个公式基于两个假设：

1. 顺序 I/O 时间为 NLP × B（访问第一个叶子页的随机 I/O 忽略不计）。
2. 随机 I/O 时间为 LPSR × NLP × A（每个叶子页分裂涉及一次随机 I/O）。

随着 A/B 的增加（这是当前的技术趋势），叶子页分裂比例的预警阀值在不断地降低。当 A/B = 100（我们假设）时，若 LPSR 达到 1%，那么 I/O 时间就翻倍了。

208

将这个公式应用于我们上面的例子：

$$\begin{aligned}\text{LPSR} &= 0.01\\ A &= 10\text{ ms}\\ B &= 0.1\text{ ms}\\ \text{ORIG} &= 0.5\text{ s}\end{aligned}$$

在叶子页分裂后，索引片扫描的 I/O 时间变成了：

$$(1 + (0.01 \times 10\text{ ms}/0.1\text{ ms})) \times 0.5\text{ s} = 2 \times 0.5\text{ s} = 1\text{ s}$$

什么时候应该对索引进行重组

插入模式

索引重组是为了恢复索引行正确的物理位置，它对于索引片扫描和全索引扫描的性能而言很重要。因为插入模式的不同，增加的索引行可能会以无序的方式来创建。我们需要记住，更新一个列意味着需要删除旧的索引行，并增加一个新的索引行，新索引行的位置由新的索引键值来确定。我们将在第 13 章中看到，一些产品支持非键值索引列，更新这些列对索引行的存储位置没有影响。

下文对三种基本插入模式的讨论基于如下两个假设：

1. 索引是唯一索引。
2. 被删除的索引行所腾出的空间在重组之前可以被新的索引行重用。

新索引行将被添加至索引的末尾（永远递增的键）

假设插入了一个索引行，其索引键值比任何已经存在的索引键值都要大，则 DBMS 就不会分裂最后的叶子页，那么就不需要空闲的空间或者进行索引重组了。然而，如果在索引前面的索引行被定期地删除，那么为了回收空闲的空间，索引可能不得不进行重组（一个“爬行”的索引）。

有一个重要的例外场景：如果索引行是变长的，那么就需要有空闲的空间去适应任何索引行的增长。

随机插入模式

我们稍后将会看到，尽管考虑了空闲空间和重组，对于不同的索引行长度（短、中、长）的处理也是不同的，如图 11.3 所示。越长的索引行越难

于处理，越短的索引行越容易处理。我们将以中等长度的索引行为例来解释其中的原因。

209

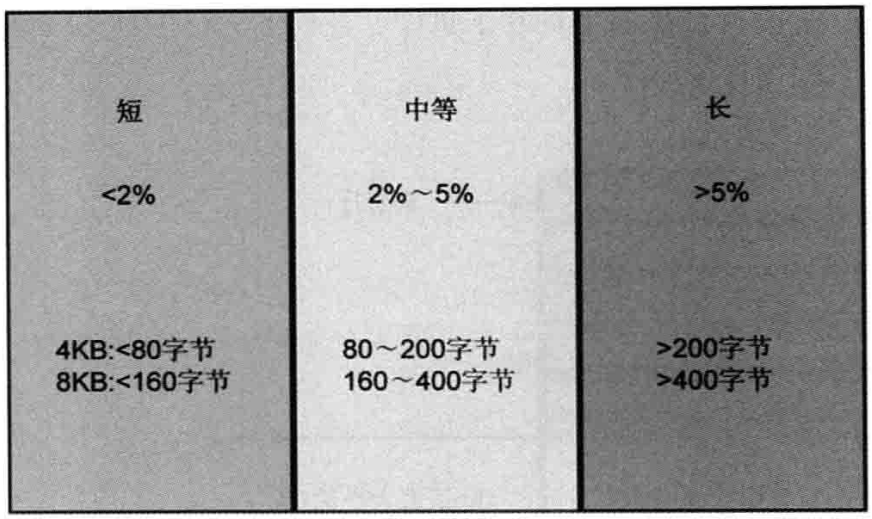


图 11.3　三种不同的空闲空间方案

短索引行的选择捷径

当索引行比较短时，参考图 11.3，为一个存在随机插入情况的索引选择 P 值是相对简单的。

基本建议：P = 10%，当索引已经增长超过 5%时重组索引。因此，P = 所预测增长值 × 2。

对于短的索引行，这个规则保证了叶子页的分裂数量维持在一个较低的水平。LPSR 将小于 1%。下面讨论的第二个场景会对 2 这一系数进行解释。此外，我们还会看到为什么当一个索引行变得更长时，叶子页的分裂将更有可能发生，且 P 值也更加难以选择。

如果希望有一个更长的索引重组间隔期，那么 P 应该被设置成 20%，并在索引增长了 12%时进行索引重组。后面我们也会讨论如果需要更长的重组间隔时间，将需要什么条件。

长索引行的困难选择

当一个索引行变得很长时，参见图 11.3，为一个存在随机插入的索引选择 P 值将是非常棘手的事。下面我们先讨论中等长度的索引行长，在理解中等长度的索引行长之后，再来探讨这个难题。

为中等长度的索引行所做的选择

索引叶子页的分裂值，可以通过两个值来进行预测，P 和 X，这两个值用于描述索引重组后的状态：

P = 创建索引时需要的空闲空间

当需要 25%的空闲空间时，在 Oracle 和 DB2 中，应当将 PCTFREE 设置为 25；在 SQL Server 中，应当将 FILLFACTOR 设置为 75。在重组（重建）和加载期间，索引行会被添加到叶子页中，直到下一行的插入无法满足 P%这个值为止，如图 11.4 所示。

210

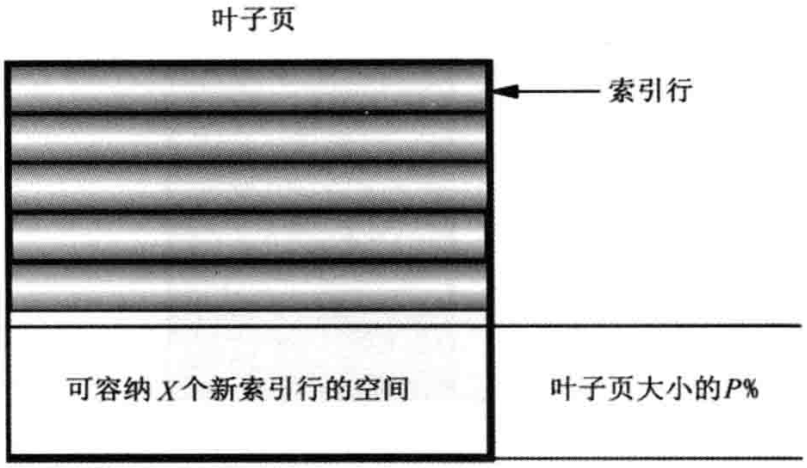


图 11.4 叶子页分裂——*P* 和 *X*

X= 在重组之后，可以被添加到叶子页上的新索引行的数量。我们的建议是将 *X* 设置成 5，稍后我们将看到这一点。

如果 *P* = 10%，叶子页大小 = 4KB，索引行的长度为 200 字节，那么 18 个索引行将填充至 *P*%（4KB × 90%），因此 *X* 将被设置成 2。

除了 *P* 和 *X*，我们还需要两个变量来描述索引的增长情况：

Y= 上一次索引重组以来索引行增加的数量
Z= 上一次索引重组以来增加的叶子页数

假设插入是随机的，那么现在我们可以通过一个二项分布来预测叶子页分裂的数量。Excel 中的函数

```
BINOMDIST(X,Y,1/Z,TRUE)
```

确定了一个叶子页未被分裂的概率。我们假设下面的场景，一个重组后的索引有 1 000 000 行，包含 50 000 个叶子页（*Z*），*P*=20%。每天新增 10 000 个索引行（增长速率约为每天 1%），以随机的方式插入，并且在可预见的未来原有的行不会被删除。

当块的填充达到 80%时，每个叶子页中有 20 个索引行，每个叶子页中预留了 5 个新索引行的空间，也就是说 *X*=5。

在重组后的最初 5 天内是否会有叶子页的分裂[50 000 次插入（*Y*）]？

```
BINOMDIST(5,50000,1/50000,TRUE) = 0.9994
```

也就是说，在 5 天内将会有 0.06%的叶子页发生分裂，即 0.06% × 50 000 = 30。在这种情况下，一次索引扫描的 I/O 时间将变为(1 + 0.06) × ORIG，这是可以接受的。

在 10 天后（100 000 次插入后），没有分裂的叶子页的比例变成了：

```
BINOMDIST(5,100000,1/50000,TRUE) = 0.9834
```

1.7%的叶子页将会发生分裂，索引扫描的 I/O 时间将变为(1 + 1.7) × ORIG = 2.7 × ORIG。平均每个叶子页中会有 2 个新的索引行（即 100 000/50 000），也就是 40%（即 2/5）的空闲空间将会被使用。 211

当只有 40%的空闲空间被使用时，就有这么多的叶子页已经被分裂了（1.7% × 50 000 = 850）——这可能看起来非常奇怪。图 11.5 说明了为什么会发生这种情况，它展示了在随机插入的模式下，新的索引行命中叶子页的情况是如何的。

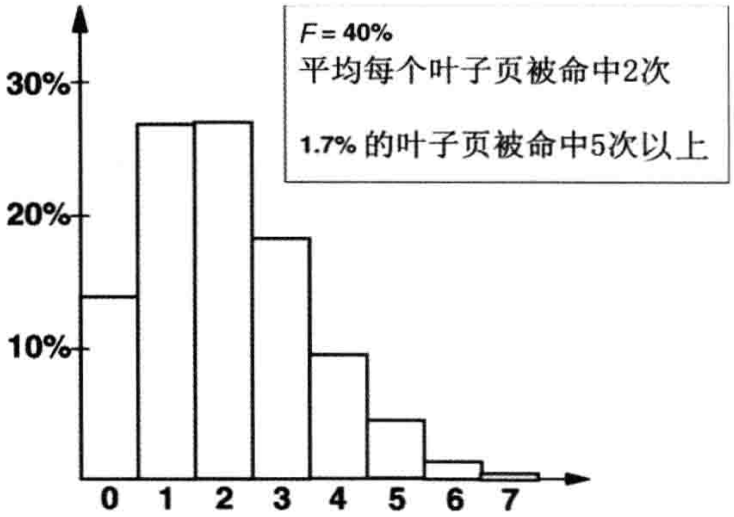


图 11.5 随机插入的影响

- 15%的叶子页将不会被命中。
- 超过 25%的叶子页只会被命中一次。
- 1.2%的叶子页将会被命中 6 次，这些叶子页都会发生分裂。

我们将会看到空闲空间的填充因子（40%以上）在 LPSR 预测中有着非常重要的作用。我们将这个值称为 F：

F = 在某个特定的时间点，空闲空间被使用的百分比

当 Y 个新行被增加到 Z 个叶子页（重组时每个节点有可以存储 X 个新行的空闲空间）中时，

$$F = 100 \times Y / (X \times Z)\%$$

在上面的例子中，

10 天后的 F = 100 000 行/(5 行每页 × 50 000 页) = 40%

LPSR 可以用图 11.6 中的一个 X 和 F 的函数来表达。

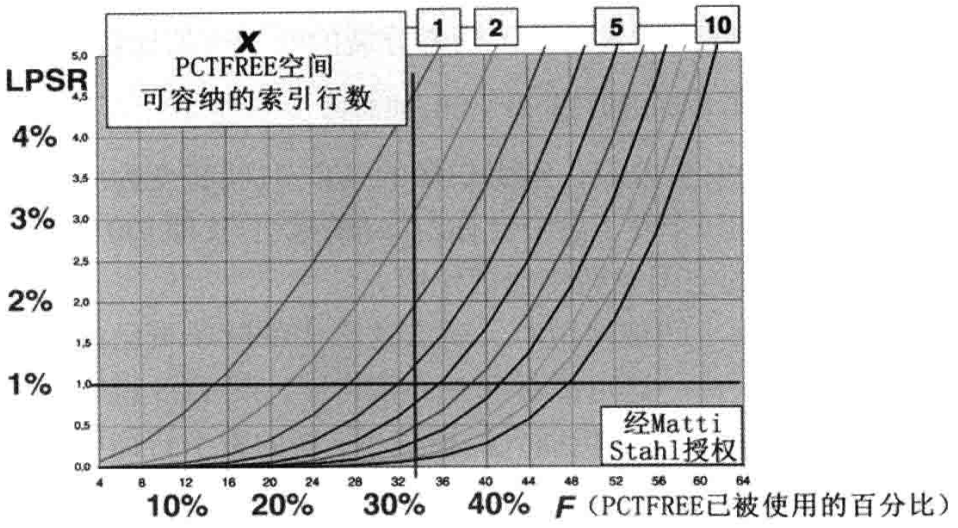


图 11.6　LPSR 作为已使用的空闲空间的函数

这个模型给出了当 LPSR 比较低时，对叶子页分裂的准确预测。在大量的叶子页分裂后，比如 LPSR > 10%，这个预测就开始变得悲观了。因为增加了大量的叶子页——每个分裂会创建两个叶子页，每个叶子页有将近 50%的空闲空间。

212 那么什么时候一个索引应该被重组？让我们假设我们希望索引扫描的 I/O 时间保持在 2 × ORIG 以下。为了达到这个目的，如果 A/B = 100，则 LPSR 必须保持在 1%以下：1 + (0.01 × 100) = 2。

参考图 11.6，当 X = 1，F 低到 15%时，LPSR 变为 1%，也就是说，只能有 15%的空闲空间被使用，这可能意味着频繁的索引重组。

当 X = 10 时，LPSR 可以保持在 1%以下，直至 F 变为 48%。这种情况看起来似乎更加合适。然而，如果索引行的长度为叶子页大小的 5%（4KB 的页大概为 200 字节，8KB 的页大概为 400 字节），那么 P 必需要达到 50%才能使 X = 10。如果所有页的空间利用率只有一半，那么顺序读将会变得相对较慢。

对于当前最常用的页大小（一般为 4KB 和 8KB），在频繁的重组和大量分配空闲空间之间，X = 5 看起来是一个合理的折中值。我们会将 X = 5 作为一般情况的建议值。当一个索引行特别长的时候，我们将需要考虑使用一个更小的 X 值，我们稍后将会看到。

图 11.6 表明了如果 X = 5，为了使 LPSR 小于 1%，当将近 36%的空闲空间已经被使用时，我们就应该重组索引了。为了简单起见，我们使用三分之一这个数字，即 33%。但是何时会达到这一状态呢？在经过多少时间之后

空闲空间的占用比例会达到三分之一？在哪个时间点索引需要被重组以满足我们的既定目标呢？

F 代表了空闲空间的占用百分比，我们需要将这个值转换为表中实际数据量的一个百分比值，即非空闲空间。在我们的例子中（$P = 20\%, F = 33\%$），重组发生的时间点应当是在索引进一步增长了（我们后续将会把这个称为重组前的增长，用 G 表示）8.3%时进行：

$$F \times (P/(100 - P)) = 33 \times (20/(100 - 20)) \approx 8.3\%$$

8.3%这一数值是索引被重组前的一个合理的增长值（为了满足 1%的 LPSR，同时保持索引扫描的 I/O 时间不超过 ORIG 的 2 倍）。图 11.7 展示了不同 P 值所对应的 G 值。 213

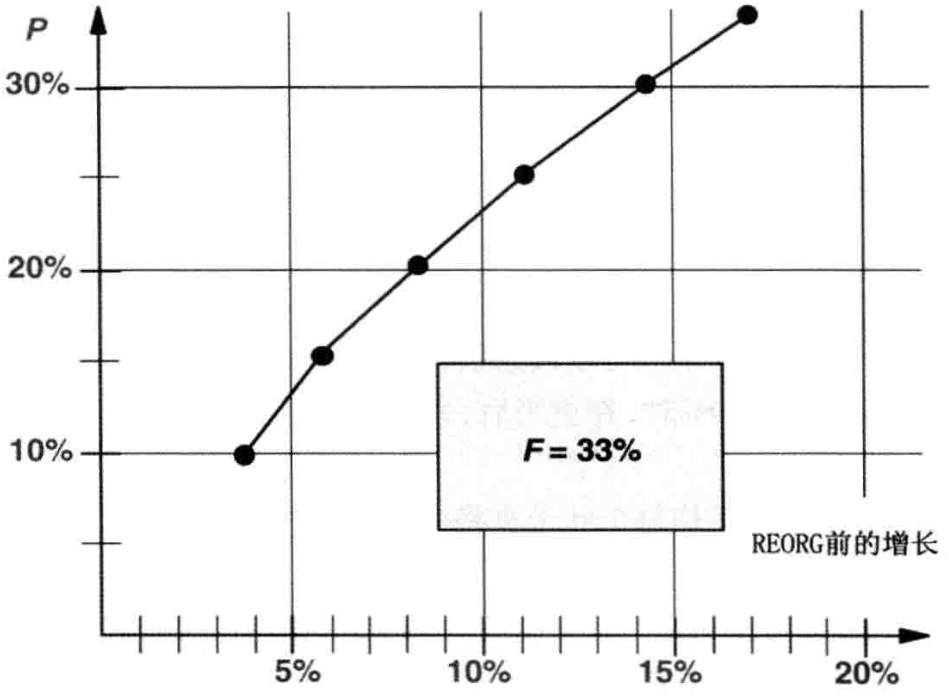


图 11.7　重组参考标准

因此，对于一个给定的 P，可以使用 G 的公式或者图 11.7 中的曲线图来预测索引重组的频率。反过来，如果重组的间隔已经确定，那么可以通过它们来确定 P 的值。假设在我们的例子中，索引可以每两周重组一次（每天的增长率为 1%）。由于 $G = 14\%$（即 14 × 1%），根据图 11.7 的建议，P 应该被设置为 30%。

许多表最开始是空表，然后保持增长，最初的增长速度可能非常快。类似这类表上的索引在最初的几周中需要特别对待：它的 P 值会非常高（甚至达 99%），且每天晚上可能都需要进行一次重组。伴随着表的增长和 P 的递减，图 11.7 可被用于预测重组的间隔可以如何延长。如果上面讨论的

索引每天新增 10 000 行，那么 3 年内表中的记录将超过 10 000 000 行。此时，若 $P = 20\%$，那么索引重组的间隔可能长达 80 天：从图 11.7 可得，与 $P = 20\%$对应的 $G = 8\%$，$10\,000\,000 \times 8\% = 800\,000$ 行，约对应 80 天。

这个例子中，索引行的长度为页大小的 4%（每页总共 25 行），对于 4KB 大小的页大约为 160 字节，对于 8KB 大小的页大约 320 字节。在这个场景中，P 必须至少为 20%以满足空闲空间中有五行记录的规则。通常这个不会成为问题，但是如果索引行的长度为现在的 3 倍（每个页的 12%）呢？在重组后每个页留有 60%空闲空间不是一个很好的方式，每个页只存储 3
214 个索引行将会导致索引片扫描的 I/O 时间变得很长，即便是在发生分裂前。这个问题的分析及解决方法将会在后面进行讨论。

推荐值的依据

短的索引行　基本建议：

$$P=10\%$$

当索引增长为5%时开始进行重组。

为了解释这个建议值，我们考虑一下临界场景：页大小为 4KB，索引行的长度为 80 字节。

如果 4096-10 = 4086 字节可以被索引行使用，那么这个索引页可以放得下 51 个索引行。当 $P = 10\%$时，在重组后，每个叶子页存放了 45 个索引行，即 $X = 6$。

当索引增长了 5%，平均每个叶子页将会有 2.25 个新索引行（45 的 5%），于是 $F = 2.25/6 \approx 38\%$。

从 $X = 6$ 的曲线可以得出（图 11.6），当 $F = 38\%$时，LPSR = 1%。

这是最坏的场景，当索引行的长度小于页大小的 2%时，LPSR 将保持在一个更低的值。

如果需要一个长的重组间隔，P 应该被设置成 20%，且在索引增长达到 12%时对其进行重组，证明如下。

当 $P = 20\%$时，重组后每个叶子页上有 40 个索引行，于是 $X = 11$。当索引增长了 12%时，平均每个叶子页会有 4.8（40 的 12%）个新的索引行，于是 $F = 4.8/11 \approx 47\%$。从 $X = 11$ 的曲线可以得出（图 11.6），当 $F = 47\%$时，LPSR = 1%。

对于其他更长的索引重组间隔，假设当 $F = 50\%$时，LPSR = 1%，则有如下公式：

$$G = 50 \times (P/(100 - P))$$

中等长度的索引行　基本建议：选择一个 P，使 $X=5$，当 $G=33\times(P/(100-P))$时，重组。

我们还是考虑临界值的场景：页大小为 4KB，索引行的长度为 200 字节。

P 必须为(5 × 200 字节)/4086 字节 = 25%，以使得 $X=5$。在重组时，每个叶子页中将存放 15 个索引行。

当索引增长率为 11%（$G=33\times25/75$）时，平均每个叶子页将会有 1.65 个新的索引行（15 的 11%）。因此 F=1.65/5 ≈ 33%。

从 $X=5$ 的曲线可以得出（图 11.6），当 $F=33\%$时，LPSR = 0.7%。

这仍旧是一种最坏的场景，当索引行的长度小于页大小的 5%时，LPSR 将保持在一个更低的值。

长的索引行　选择一个合适的 X 值和 P 值（尽量使 X 大于等于 2），然后再用常规方法找出 F 和 G 的值。我们将会在长索引行这一节中讨论关于长索引的更多细节问题。

决定重组频率的步骤总结　215

1. 对短索引行来说，P=10%。当索引的增长达到 5%（G=5%）时重组索引。或者，使用 P=20%和 G=12%。如果想要延长重组的间隔，那么选择 P>20%，$G=50\times(P/(100-P))\%$。
2. 对于长索引行，请参考后面的章节。
3. 对于中等长度的索引行：
   a.选择能够使 $X=5$ 的 P 值，$P=5\times$索引行长度/页大小。
   b.使用图 11.7 判定 G 的值和重组的间隔。

指定重组频率的步骤总结

1. 对短索引行来说，选择下面的其中一种：（$G=5\%$，$P=10\%$），（$G=12\%$，$P=20\%$），或者（$G=33\%$，$P=40\%$）。当 $G>33\%$ 时，使用公式 $P=100\times G/(G+50)$。
2. 对于长索引行，请参考后面的章节。
3. 对于中等索引行：
   a.假定 1%的 LPSR，$X=5$，$F=33\%$。
   b.针对给定的 G 的值，使用图 11.7 得到 P 的值。

大部分的新索引行都集中写入到了同一块小区域（热点——不是索引的末尾）

不幸的是，管理插入模式是困难的。在索引(BO,DATE)中，例如 BO 代表分公司，DATE 是一个不断递增的值，新的索引行都会插入到每个 BO 分片的最后。在这种情况下，索引的重组间隔必需根据叶子页的分裂数量和它

们对索引片扫描性能的影响来为每一个索引量身定制。

在索引(BO,DATE)中，包含最大 BO 的最新一行的页是最热的页。它在重组后很快就会分裂。在第一次分裂后，最热的页填充了一半的空间。不久之后，最热的页将只包含最大 BO 的行。此时该 BO 所对应的最新插入的行还是可以被快速查询出的。当许多分公司的最新页被分裂时，情况会变得糟糕。最终，每个 BO 的每个新索引行，都可能会被存放到不同的叶子页中。为了获取每个 BO 最近 20 行的记录，可能会消耗 20 次的随机索引访问。

在类似这样的场景下，有一个简单的解决方案可以很好地缓解这个问题。如果老记录（比较早的 DATE）会被定期删除，索引不应该在删除后进行重组。在每个热点附近有空的叶子页是很合适的，在末尾腾出的空间正好能给头部的增长使用。

随着内存价格不断下降，以及 64 位寻址空间使更大的缓冲池成为可能，将一个插入频率高且存在热点的索引缓存在内存中成为了越来越可行的一个方案。这样做的一个明显好处是没有读 I/O，此外一个常驻内存中的索引
216 对叶子页的分裂是不敏感的。由于随机读取一个内存中的叶子页的时间约为 0.1 ms（而非原来的 10 ms），所以一个常驻内存的索引可能不需要重组，即使它的某一小块区域有很大的并发插入。如果将一个宽索引常驻于内存的代价太大，那么将一个半宽索引常驻于内存可能是一个不错的替代方案。

特殊场景

在某些场景下，上面的一个或两个假设可能并不成立，此时可以考虑下面的建议。

1. 如果索引是非唯一索引，每一个索引键后会有多个指针，每个指针对应一行满足该重复键值的记录，那么每个索引行的物理存储空间可能很短，如 10 字节左右（一个指针和少量的控制字符）。如果所有的索引键值都对应有大量的指针（如性别的索引），那么空闲空间的需求将会大大减少。当每个页中 67%（相比 33%）的空闲空间已经被使用时，索引可能还在一个很好的状态。如果部分索引键值只有一个指针，那么安全起见应当使用与唯一索引相同的计算标准。
2. 如果关系型数据库一直不回收已删除的索引行的空闲空间，直到索引重组后才回收，那么，在决定索引重组频率的公式中，应该使用总增长值（插入量）来代替净增长值（插入量–删除量）。

索引列的稳定性

我们考虑一个以极其不稳定的列作为起始列的索引(INV_THIS_M,CNO, CNAME)，其中列 INV_THIS_M 指本月的单据金额。使用这个索引，假设 DBMS 可以逆向读取索引，下面的查询将会非常快：

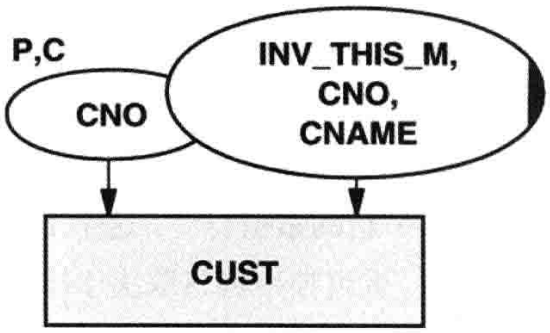
```
TR = 1, TS = 500, F = 500, LRT = 65 ms
```

图 11.8 展示了这个月刚开始时的索引情况，此时索引行是按 CNO 的顺序存储的。当第一个单据被创建，DBMS 必需将相关的索引行移动到索引的最后。接着会对第二个单据相关的客户记录进行移动，根据单据的金额决定将它移动至先前被移动的索引之前或者之后。对于每一个新产生的单据，索引行都可能会被移动，并且移动总是向前的。于是在索引的末尾附近，将会有大量的叶子页分裂。这是一个热点问题。

SQL 11.3

```
SELECT      CNAME, CNO, INV_THIS_M
FROM        CUST
ORDER BY    INV_THIS_M DESC
WE WANT 500 ROWS PLEASE
```

217



INV_THIS_M	CNO
0	4
0	10
0	14
0	15
0	19
0	22
...	...

图 11.8 一个“边界分裂”的索引——索引行的移动不是随机的

如果索引重组一个月只发生一次，LPSR 将会变得很高。理论上，这个 500 行的查询可能需要多达 500 次的随机读取（如果每次索引访问都是随机的），虽然实际的 TR 次数可能会少很多。真正的问题会发生在那些具有以 INV_THIS_M 列开头的非三星索引的 SELECT 上。在这种情况下，一次

FETCH 可能需要数千次的索引访问，如果 LPSR 很高，那么其中大部分的访问可能都是随机的。

对于这个热点问题，至少有下面三种可行的解决方案。

1. 使索引缓存在内存中，$P = 0$，不进行索引重组。如果有 1 000 000 个用户，索引的大小可能为 60 MB。如果内存的价格为每月 1000 美元/GB，那么这个缓存在内存中的索引的成本就会是 60 美元/月。然而，在 32 位系统中，虚拟存储的限制可能会使这种解决方案不可行。
2. 创建一个长期缓存在内存中的半宽索引(INV_THIS_MONTH)，$P = 0$，不进行索引重组。这个索引不会消耗太多的内存——可能小于 10 MB，因为它是非唯一的。由于是半宽索引，所以每次 FETCH 都需要进行一次表访问。因此，这个替代方案只有当每一次的 FETCH（而不是许多次的 FETCH）都会涉及许多索引访问的场景下才有意义：无论 LPSR 多高，索引访问都可以保持很快。
3. 定期重组这个宽索引，以使 LPSR 保持在较低值，可能每个晚上都需要进行一次。如果有 1 000 000 个用户，索引重组所需的时间可能小于 1 分钟。

如果索引行的移动是随机的，那么变动频繁的索引列并不会引起索引分裂问题。考虑如下场景：在一个 1 000 000 行的表 CUST 上有一个以 ZIPCODE 开头的索引。每年有 100 000 名客户需要移动到新的 ZIPCODE，而新增客
218 户的比例却只有 2%。在依照基本建议值分配了空闲空间之后（P 至少 10%，并且每个叶子页的预留空间至少可以容纳 5 个新的索引行），这个 LPSR 很可能一年以上都可以保持在一个非常低的值。重组频率将只取决于净增长值。

一般情况下，我们应当将变更频繁的列尽可能地放在索引的右边。对于 WHERE STATUS = :STATUS AND ITEMNO = ITEMNO，ORDER 表上的索引应该是(ITEMNO,STATUS . . .)，而不是(STATUS,ITEMNO . . .)。在第一个索引条件下，由列 STATUS 改变所引起的移动距离将更短。即使这些移动不是随机的，LPSR 也可能保持在一个较低的水平。

在主键之后的不稳定的定长索引列，如索引(CNAME,CNO,INV_THIS_M)中的 INV_THIS_M 列，不会造成索引分裂，因为更新这个列不会引起索引行的移动。

长索引行

如果为了使 P 能够容纳 5 个新索引行（$X = 5$）会造成大量的空闲空间，那么我们认为这个索引的长度是长的。当 P 不足以存放 5 行时，公式

```
G = 33 × P / (100% - P)
```

将不再有效，因为当 $F = 33\%$时，LPSR > 1%。举例来说，如果空闲的空间只能容纳一行，则此时 LPSR = 4.5%——见图 11.6 中 $X = 1$ 时的曲线。

假设一个叶子页中只能容纳 8 个索引行。为了让每个叶子页能够存放 5 个新索引行，P 应该为 63%（即 5/8）。这会导致索引扫描的速度变得很慢，即使没有发生分裂。$P = 25\%$（重组后存放 6 行，预留 2 行的空间）看起来更加合理，让我们来看看在这种情况下 LPSR 会如何增长。

根据图 11.6 中 $X = 2$ 时的曲线，我们看到当 $F = 22\%$时，LPSR 将达到 1%。在那个点上，每页的平均新索引行数为 $0.22 \times 2 = 0.44$，所以索引增长率为 7%（即 $100\% \times 0.44/6 \approx 7\%$）。如果允许以这样频率重组索引，那么 LPSR 将保持在 1%以下，并且 $P = 25\%$。

假设索引行变得更长，可能一页只能存储 4 行，则合理的替换方案是 $P = 25\%$（$X = 1$）和 $P = 50\%$（$X = 2$）。在第一种方案中，为了保持 LPSR 在 1%以下，索引必须在 $F = 15\%$时重组（图 11.6）。那就是说，平均每页的新索引行数为 $0.15 \times 1 = 0.15$。索引的增长率为 5%（$100\% \times 0.15/3$）。这个解决方案比 $P = 50\%$时更好，因为在重组后，每个叶子页中的索引行有 3 个，而不是 2 个。此时访问每个索引行的 I/O 时间是 0.03 ms，而 $P = 50\%$时，这一时间为 0.05 ms。如果索引不能以这样的频率被重组，那么就必须从如下替代方案中选择一个：

1. 设置 $P = 50\%$，当索引增长达到 22%时，进行重组。此时，22% 的空闲空间已经被使用，并且每个叶子页的平均新索引行数为 $0.22 \times 2 = 0.44$。索引已经增长了 $100\% \times 0.44/2$，LPSR = 1%。 219
2. 如果有可用选项的话（如 DB2 for z/OS 中的 FREEPAGE 选项），在重组时应为每 N 个叶子页预留一个空页。如此一来，大部分的叶子页分裂的影响将不那么大。当叶子页的另一半被移动到临近的一些新页上时，这个分裂一般不会引起随机 I/O。至少硬盘系统通常会使用预读功能，这样附近的页通常可以在缓冲池中被发现。如果 $P = 25\%$（足够存放一个新行），并且每四个页预留一

个空页，那么每 32 个页组将会有 56 个新行（即(32 × 1) + (8 × 3)= 56）的空间。即使当 $F = 50\%$时，发生远距离分裂的概率也是非常低的。为了避免索引扫描时间翻倍而所需调整的索引重组间隔时间是很难预测的，但是它可能是第一种方案的 2 倍以上（第一种方案中的空闲空间更多一些，为 50%，第二种方案的空闲空间为 56/128 = 44%）。

在这个场景中，第二种替代方案看起来更好。然而在一般情况下，当索引行没有这么长时，通过使用空页来减轻叶子页分裂的影响，不如通过高的 P 值来避免叶子页的分裂。空页会使顺序读变得更慢，就像每个叶子页中大量的空闲空间一样。

这两个例子可能看起来有点不切实际。是否有这么长的索引行，使一个叶子页中只能存储 4 行或 8 行？如今这样的索引是很少见的，但是随着硬件的演变和 DBMS 限制的放宽，这样的场景可能会变得越来越常见。从索引分裂的角度，对索引行太长的情况有所思考是很明智的。此外，一些产品也提供了在索引页中存储表行的选项。表行通常较长——500 字节以上是很常见的。

半宽及宽索引所能带来的好处依赖于当前硬件情况下索引片扫描的速度，根据 QUBE，该值为 0.01 ms/TS。当索引很长时，这个优势就减少了。如果每个 4KB 大小的叶子页只能存放两行，每行的 I/O 时间可能变为 0.1 ms/2 = 0.05 ms。更严重的是，当 $X < 5$ 时，随着插入量的增加，索引随机访问的数量将急剧增加。

举例：对顺序敏感的批处理任务

如图 11.9 和 SQL 11.4 所示，根据 QUBE，一个连接 CUST 表和 POLICY 表涉及 1200 万次访问的批处理程序只需要 10 分钟——至少当 QUBE 所基于的假设都成立时，运行速度是这样的——即索引和表是有序的，就像它被重组过一样。（注：因为我们想要 500 行，优化器将会选择嵌套循环的方式，
220 这样可以避免排序。程序会在大致 500 次 FETCH 后创建一个提交点。出于同样的原因，按照物理顺序来读取索引行——如 Oracle 中的 FAST FULLINDEX SCAN——并不是一个合适的选择）。

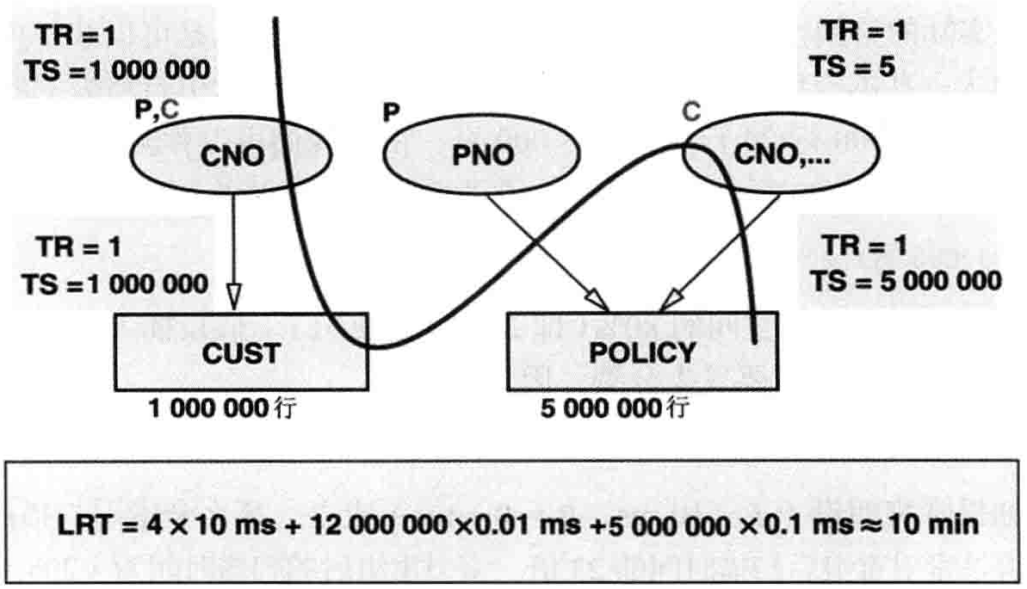


图 11.9　插入可能导致大量批处理程序变慢

SQL 11.4

```
DECLARE CURSOR113 CURSOR FOR
SELECT     many columns
FROM       CUST, POLICY
WHERE      CUST.CNO = POLICY.CNO
ORDER BY   CNO

OPEN CURSOR113
FETCH while...
CLOSE CURSOR113
WE WANT 500 ROWS PLEASE

大约每500次FETCH一个提交点
```

我们假设 6 个月内，保险单增加了 10%（也许是因为一个成功的营销活动）。在此之前，每个叶子页的空闲空间被设置为了 10%，因为当业务增长缓慢的时候，这足以支持 6 个月的业务了。现在的结果是，由于每个叶子页的平均增长率为 10%,所以推测其中一半的叶子页发生了 10%以上的增长，从而导致了许多页发生了分裂。

我们再详细看一下在插入 500 000 行记录后 POLICY 表的(CNO,...)索引 221 的状态。假设索引行的长度是固定的 150 字节，每个叶子页的可用空间为 4086 字节（4096 −10）。当 P = 10%时，在重组时使用 3677 字节（即 0.9 × 4086≈3677）可以存放 24 个索引行（24 × 150 字节 = 3600 字节），但存不下 25 个索引行。由于 4086 字节的空间最多可以存放 27 个索引行（27 × 150 字节 = 4050 字节），所以每页预留了 3 行的空间给新插入的行使用。

因此，实际的空闲空间离散值是 3/27≈11%。我们几乎总是可以获得比所需空间稍大一些的空闲空间。现在为了存储 5 000 000 个索引行所需的叶子页的数为 5 000 000 行/24 行/页≈208 000 页，由此我们可以预测插入的效果。

在插入 500 000 行之后，每个叶子页需要平均存储的新索引行为：

```
500 000 行/ 208 000 页≈2.4 行/页
```

因此，离散空闲空间的 80%（即 2.4/3）被使用了。假设插入是随机的，那么 22%的叶子页需要发生分裂，因为

```
BINOMDIST(3 500 000,1/208 000, TRUE) = 0.78
```

如果常规假设（A = 10 ms，B = 0.1 ms）成立，那么全索引扫描的 I/O 时间将是索引重组后扫描时间的 23 倍。索引重组后的扫描时间为（208 000 × 0.1 ms ≈ 21 s），而不重组的扫描时间为：

```
(1 + (LPSR × A/B)) × ORIG = (1 + (0.22 × 100)) × 21 s
= 23 × 21 s
≈ 8 min
```

实际的 I/O 时间会比 8 分钟少，因为二项分布模型在发生大量索引分裂的情况下是较为悲观的。无论在何种情况下，高的 LPSR 都应该被每周的异常报告展示出来（在本章后面会详细讲解），以免索引变得非常无序。

现在让我们使用前面的模型来分析一下，假设我们期望索引重组前的增长率为 10%。图 11.7 建议当 G = 10%时，P 应该被设置为 23%。现在只能在 0.77 × 4086 字节≈3146 字节的空间中存储 20 个以内的索引行（21 × 150 字节 =3150 字节，已大于 3146 字节），预留出了 7 个索引行的空间。即 X=7，F=29%（2/7≈29%），从图 11.6 中可以看出，LPSR = 0.1%，因此在 10%的索引增长后，全索引扫描的 I/O 时间将会变得很短：

```
(1 + ( LPSR × A/B)) × ORIG = (1 + (0.001 × 100)) ×25 s
= 1.1 × 25 s
≈ 28 s
```

这个结果好得难以置信，推荐值不应是基于 LPSR = 1%得出的吗？其实我们的假设是 LPSR < 1%，但真实的 LPSR 通常远小于 1%，主要有以下两个原因：

1. 基本建议是通过设置使每个空闲页面填充 5 个新的索引行 $P(X=5)$，在我们的例子中 X=7。
2. 在重组时，对 P 是取整的。在我们的例子中，当 P = 23%时，约 21 个索引行可以填充到一个叶子页中。

222

在这个时候重组索引不是必需的，但是做了也没问题。在图 11.6 所示的曲线图中，当 $X=7$ 时，性能开始恶化——在这一个点上，曲线开始变陡了。

这个例子说明了，如果索引重组时间延后，会导致索引全扫描的 I/O 时间显著增加。这种情况可能会在表的增长超出预期或者监控不足的时候发生。由于 500 万次的 FETCH 调用需要很高的 CPU 时间（QUBE：500 s），所以批处理程序的运行时间并不会以相同的比例增长。受高 LPSR 值影响最大的是那些从厚索引片中找出一些索引行的 SELECT 调用，这类调用的运行时间大都来自 I/O 时间。

表乱序（存在聚簇索引）

POLICY 表增长了 10%，P = 10%，和它的索引一样。这会对批处理程序的运行时间造成怎样的影响呢?

在 500 000 次（10%的增长率）插入后，几乎所有的 POLICY 表页都满了。因为（真正的）表页不会发生分裂，所以 P 等于 10%意味着当表增长了

```
P / (100% - P) ≈ 11%
```

之后，分配的空闲空间就全部被用完了。那些在主页上（本来应该被插入到这儿的）放不下的表行通常会被存放到主页附近的一个页中。存储在叶子页的表行将在下一小节中进行讨论。如果当主页已满时 DBMS 把表行放到了主页的附近，那么访问表的性能一开始并不会急剧恶化。这是因为最近访问的页面会被缓存在缓冲池中或者磁盘的缓冲区中。但是，当所有主页附近的页面都被写满时，新行可能会插入到 POLICY 表的末尾。离主页很远的每一行记录可能都会增加一次随机 I/O。如果每个页面的空闲空间（10%）中都只够存放一条新的记录，那么在 500 000 次插入后，表可能会处于一个非常可怕的状态。额外的随机 I/O 次数可能会超过 100 000（假设有超过 100 000 个表行距离主页很远），于是程序的响应时间将增加 15 分钟以上。在这种情况下，这张表应该分配更多的空闲空间，至少为 20%，以适应 10%的增长量。

如果 DBMS 不支持聚簇索引，那么 POLICY 表就必须频繁地进行重组，因为新的 POLICY 行会增加到表的末尾，并且每一个 INSERT 会给批处理程序增加 10 ms 的响应时间（一次随机访问）。在插入 10 000 个新行后，POLICY 表只会增长 0.2%，但是通过外键索引的表扫描的时间将增长 50 s（即 $1 \times 10\,\text{ms} + 5\,000\,000 \times 0.01\,\text{ms} \approx 50\text{s}$）至 150 s（即 $10\,000 \times 10\,\text{ms} + 5\,000\,000 \times 0.01\,\text{ms} = 150\text{s}$）。为了避免 POLICY 表的频繁重组，应当将列增加到索引 223

(CNO,...)上以避免表访问，或者在 DBMS 支持的前提下将表行存储到那个索引中，更多的细节后续会再做讨论。

表乱序（没有以 CNO 开头的聚簇索引）

图 11.10 展示了如果 POLICY 表的行并没有按照 CNO 的顺序存储，会发生什么样的情况。图中展示是聚簇索引的情况（表上没有聚簇索引，并且在重新加载时表行并没有按 CNO 排序的情况与之类似）。

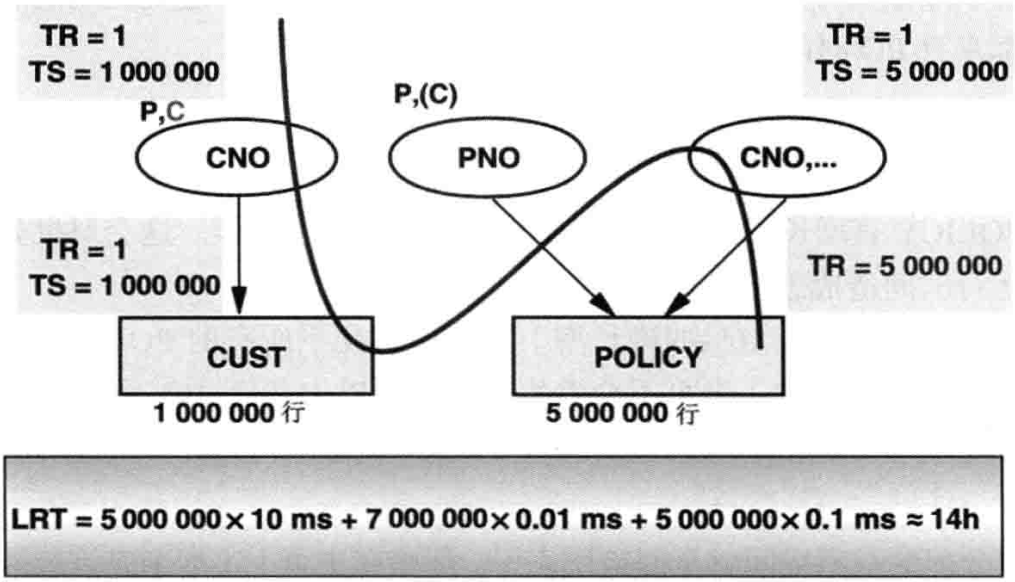


图 11.10　在重组后大规模批处理任务就已经运行得很慢了

即便是在营销活动前，这 1200 万次的访问也需要花费 14 h。此时，将外键索引(CNO,...)针对该批处理程序变更为宽索引将产生巨大的差异。在一般情况下，只要访问路径是仅限于索引而无须回表的，则表的行顺序就不重要。

存储在叶子页中的表行

在真实场景中，表页从来不会发生分裂，但是对于存储在叶子页中的表行，当叶子页分裂时，这些行却会被移动。我们需要担心这种情况吗？

SQL Server

在 SQL Server 2000 中，如果一张表上有一个聚集索引，则表行将被存储在聚集索引的叶子页中。让我们假设主键索引(CNO)是 CUST 表上的聚集索引，并且外键索引(CNO,...)是 POLICY 表的聚集索引——参考图 11.11。这样就避免了 6 000 000 次访问——就像批处理程序中用到的宽索引一样——同

224

时也节省了磁盘的空间。由于没有表页需要被更新，因此对 CUST 表和 POLICY 表的插入操作会缩短 10 ms 左右。

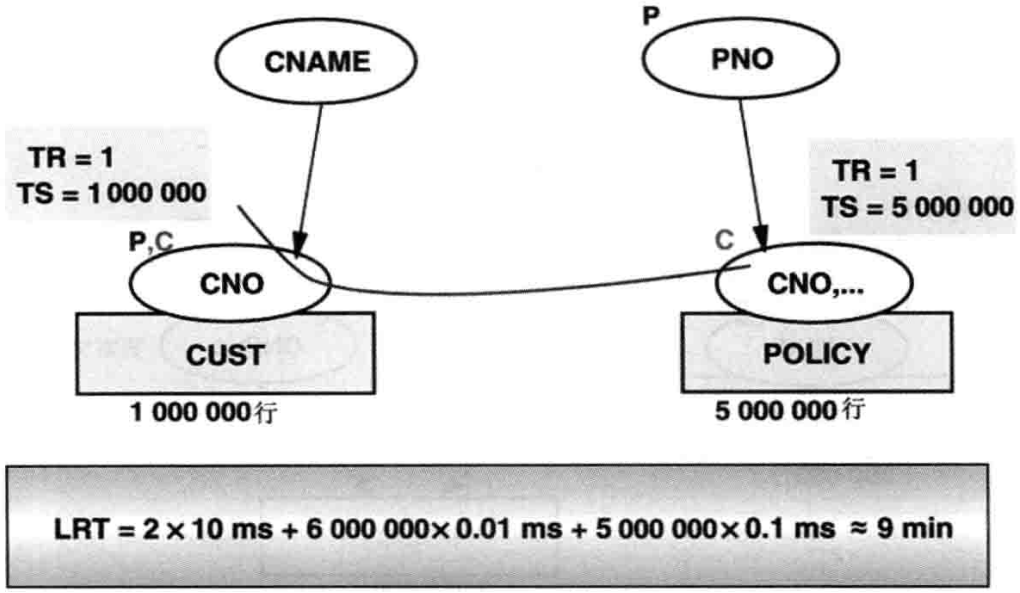


图 11.11　在叶子页中存储表行——SQL Server

非聚集索引通过聚集索引键指向表行。这些间接的指针增加了少量额外的开销（例如，就 SELECT. . .WHERE PNO =:PNO 这一 SQL 而言，就必须访问两组非叶子页），但是这样防止了在叶子页分裂移动表行时所带来的大量的指针更新。聚集索引叶子页分裂的影响和其他索引一样：索引片读取的时间变长了。如果聚集索引的键值并非是永远递增的，那么当表行很长时，LPSR 可能会急剧增长。

Oracle

在 Oracle 9i 中，将表行存储在叶子页（索引组织表）中的实现方式与 SQL Server 有以下三处不同。

1. 只有主键索引可以被用来存储表行。
2. 在其他的索引上，需要有两个指针：一个直接指针，一个主键。如果因为分裂引起了表行的迁移，直接指针不会更新。直接指针会被作为优先的查询方案，如果表行已不在那个地址，那么会访问主键所对应的非叶子页来查询表行。
3. 有一个选项可以用于在叶子页中选择只存储表行的前 *N* 个字节，将剩余的内容存储到溢出区域中。这减少了每个叶子页的空闲空间需求，但是访问溢出区域中的列会变慢。

225

图 11.12 显示了 CUST 表如何被存储为一个索引组织表。这么做能为我

们的批处理程序省却 100 万次读取，并且对于任何 WHERE CNO = :CNO 的查询条件，索引都是宽的。为了减少批处理程序的表访问，同时防止频繁地对 POLICY 进行重组，POLICY 表上的外键索引应该被建为宽索引。因为索引(CNO,...)不是主键索引，所以表行不能存储在这个索引上。

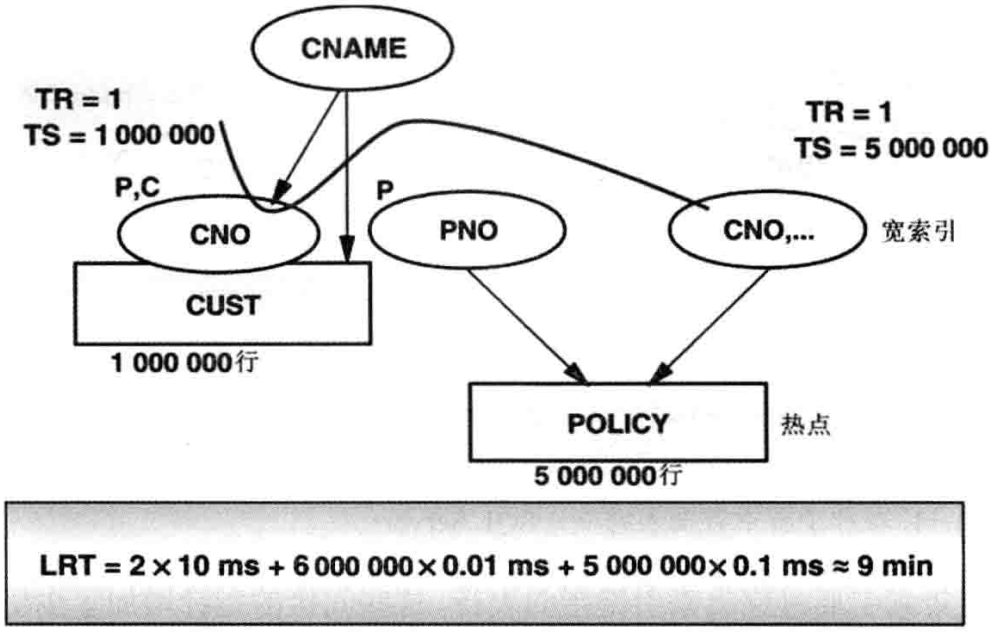


图 11.12　在叶子页中存储表行——Oracle

让我们假设页（块）的大小为 8KB，并且对 CUST 表的插入是随机的。不使用溢出区域的情况下，每个叶子页可以存储 6 个 CUST 表行，并预留 2 个新行的空间（$X=2$）。为了使 LPSR 小于 1%，当空闲空间的 20%被使用时（即图 11.6 中 F=20%时），CNO 索引就必须进行重组。在 F=20%时，平均每个页已经增加了 0.4 个 CUST 行($F \times X = 0.2 \times 2 = 0.4$)，这相当于 CUST 表增长了 7%（0.4/6≈7%）。为了使直接指针的查询成功率接近 100%，可能需要更加频繁的重组操作。

索引重组的代价

一个索引可以以多种方式进行重组：

1. 全索引扫描（随机访问且无须排序，或者顺序访问并排序）。
2. 全表扫描（类似 CREATE INDEX；顺序访问及一次排序）。

226 根据 QUBE，对于第二种方案，每 100 万行记录的查询时间是 4 分钟（200 万次顺序访问，200 万次 SQL 调用，100 万行排序），但是实际情况可能会比这个更快，因为许多工具会绕过应用程序接口。排序一般都是最耗时的部分，会消耗大量的 CPU 时间（例如 100 万行会消耗 10 s）。

由索引重组产生的锁等待依赖于具体的数据库和选项。如果使用的是简易的工具，那么当表或者索引正在被扫描时，整个表可能被加上一个 S 锁（更新操作会被阻塞）。如果工具能在扫描期间将更新操作保存下来，并在排序前将它们应用到数据行上，那么锁等待的时间将会缩短很多。

有些时候，大的索引可能不得不在不合适的时间被重组。在最坏情况下，锁的问题可能会导致频繁的重组无法实现。在这种情况下，一些易变的索引可能必需被强制缓存在内存中（将其固定在内存中）。

分裂的监控

对于某些索引的重组特性，我们应该提前就能够预测到，至少对于那些有大量插入和更新操作的大表上的索引应当如此，正如我们前面所说的那样。对于其他大多数的索引而言，可以基于监控来决定是否需要重组。

每个索引的叶子页分裂数可以被准确地监控，或者可以简单地通过观察叶子页的数量来确定。若插入是随机的，并且重组时不会留有空白页，那么 LPSR 的值（叶子页分裂的数量除以叶子页的总数）就足以确定哪些索引可能需要被重组。长距离的叶子页分裂的数量除以叶子页总数被命名为 LLPSR，它是一个更好的指标（可以拥有更少的误报），因为短距离的叶子页分裂很少会在索引片扫描时产生随机读。在 DB2 for z/OS 的实时统计报告中有关于 LLPSR 的报告，例如：

```
(100 × REORGLEAFFAR / NACTIVE)
```

当一个新索引行被增加到索引的末尾时，如果 DBMS 会分裂最后的那个叶子页，并且无法获取 LLSPR 的值，那么一个键值递增的索引应该从重组的候选索引列表中排除。

我们假设这样一个场景：其中有 2000 张表和 5000 个索引，大部分索引以 $P = 10\%$被创建（这是 Oracle 和 DB2 的默认值），只读的表 $P = 0$，快速增长的表的索引拥有一个更高的定制的 P 值和重组计划。其余的索引则一年进行一次重组。

如果有许多的插入和更新操作，那么应当每周对索引的状态进行一次检查。在最重要的报告中应该展示所有 LPSR > 1%的索引（或者 LLPSR > 1%的索引，如果有这个指标的话）。但是如果这个报告包含了数百个索引，应该如何处理呢？报告中的索引仅仅是被怀疑可能有问题的，它们可能需要重组，但是也很有可能大部分不需要重组。这依赖于它们是如何被使用的，例如：

227

- 如果索引总是被用于查询一行记录，那么一个高的 LPSR 值并不会造成问题。
- 如果被扫描的索引片总是很窄，比如不超过 1000 行记录，那么一个高的 LPSR 值可能也不会有太大问题。

如果索引重组的影响很大，除非我们能够证明 SQL 性能不佳确定是索引分裂导致的，否则就不要对它进行重组。可以通过一个展示了很慢的事务或者 SQL 调用的异常报告来找到那些对分裂敏感的索引，如果一个查询因为大量的索引页随机读取而花费了很长时间，那么原因可能如下：

1. 没有以嵌套循环的方式执行一个非 BJQ 连接。
2. 有一个较高的 LPSR 值并进行了索引片扫描。

于是，我们可以对所有可能有问题的索引进行重组以获得更大范围的收益，或者也可以仅仅对那些已经确认能够获得小范围收益的索引进行重组，如图 11.13 所示。如果重组不会引起显著的争用（指锁等待、CPU 排队和磁盘排队），并且服务器是公司自购的因而不会产生额外的 CPU 时间费用，那么第一种方法可能是更加合理的。

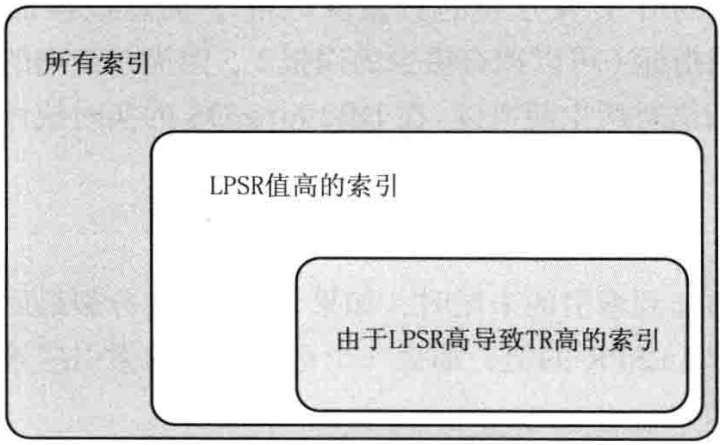


图 11.13　重组候选索引

每周的分裂报告应包括自最近一次重组以来的插入次数，如果该信息可用的话。DB2 for z/OS 中的 RTS:REORGINSERTS 是一个很好的指标，它可以被用来帮助改进重组的计划。如果 LPSR 的增长相对于插入次数较高，且高于二项分布的预测值，那么这个索引很可能存在热点。于是，我们可能需要采取相应的措施，比如使索引常驻于内存中。

总结

索引重组管理是一个有争议的话题。Donald K. Burleson（9）在最近刚

发布的一篇文章中提出了以下想法。

> **辩论仍在继续**
>
> 如今正在进行的一个争论是：有些学者认为如果没有很深入的研究，索引就不应该被重组；而实用主义者则支持有计划地重组索引，因为他们从最终用户那获得了重建索引可以获得更快响应速度的反馈。 228
>
> 目前为止，世界上没有一个 Oracle 专家可以确定一个可靠的索引重组规则，也没有专家能够证明索引重组的好处“很少”。从一个不稳定的生产系统获取统计学上的有效“证明”是一个很大的挑战。在一个大型的生产系统中，跟踪某查询在某索引重建前后的 LIO 是非常困难的。
>
> - **学术的角度：**许多 Oracle 专家声称，能从索引重建中获得好处的场景很少，但他们却没有给出这种情况的经验和证据，也没有给出实际的例子说明在哪种“罕见”的逻辑 I/O 场景中，索引能从重建中受益。
> - **实践的角度：**因为有最终用户反馈在重建后获得了更快的响应速度，所以许多 IT 经理要求他们的 Oracle DBA 周期性地重建索引。这些“实用主义者”对如何“证明”重建的好处并不感兴趣，他们只是跟着用户的反馈走。即使索引的重建被证明是无用的，但对最终用户的心理安慰作用就足以证明这个工作的意义了。
>
> 很显然，所有这 70 个索引的度量都是以可预见的方式相互作用的，一些科学家应该根据这些数据对索引重建的内部规则进行反向推理——如果确实存在所谓的规则的话。目前而言，Oracle 专家所能做的就是探索他们的索引，并学习软件是如何管理 B 树结构的。

索引重建的实际效果不能被否认，我们相信世界上有许多索引应当被周期性地重组（重建），以使顺序读的收益最大化。许多索引都在增长，还有一些索引是以变更频繁且递增的列作为起始列的。读取厚索引片正变得越来越常见，就像我们在全书中所看到的那样。

像 LPSR 和 LLPSR 这些静态指标只能将索引分为两个集合：一部分不需要重组，另一部分可能需要重组。而真正是否需要重组，只能够通过估算或者监控每次 SQL 调用中叶子页的随机读数量来判定。当发现一个读取了厚索引片的 SQL 调用时，索引重组的效果就可以很容易地确定下来。

在设计索引时，就应该预测索引的增长。此外，一定要记住，若索引行的长度相对叶子页的大小较长，则当只有一小部分的离散空闲空间被使用时，叶子页分裂可能就已经发生了。这可能限制了一个索引中列的数量，有时创

建一个新的索引可能更好。叶子页的分裂率与索引行的长度相关，这一事实降低了将表行存储在索引中的吸力。

图 11.14 总结了关于空闲空间和索引重组的最主要的关键点。假设行是随机插入的，对于较短和中等长度的索引行的建议如图所示。我们之前已经看到，对于长的索引行，无法给出特定的建议。

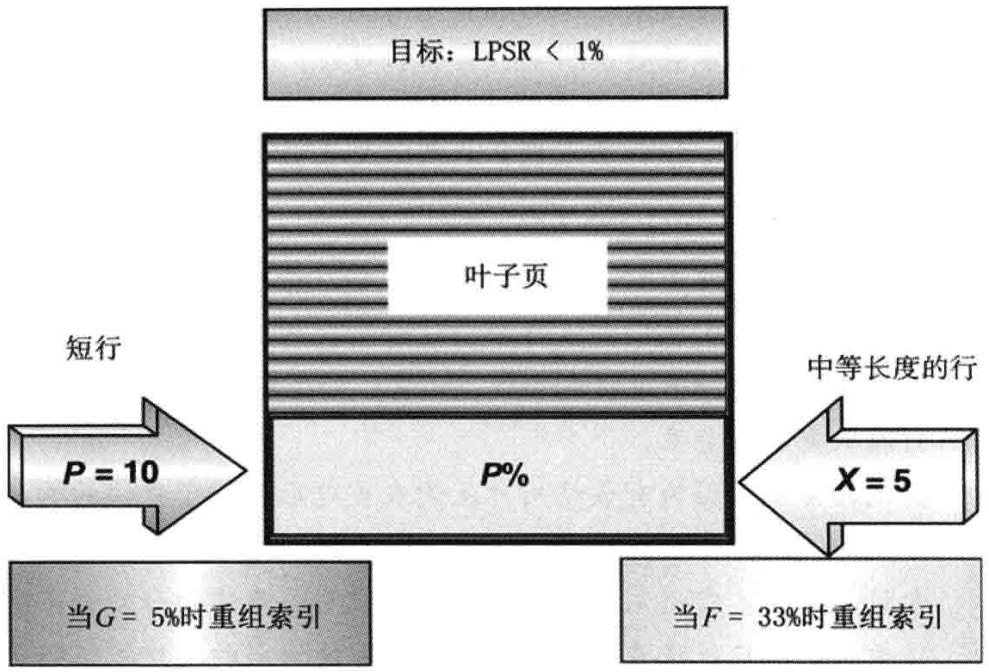


图 11.14　空闲空间以及重组的总结

229

图 11.14 中包括了影响这些建议最重要的两个特征变量。理解 LPSR 的意义是至关重要的，一个非常小的 LPSR 值就会影响索引扫描的性能，图中 1%的标准无疑会使人感到惊讶，甚至包括许多有经验的 DBA 在内。同样会令人感到惊讶的是，当相对较少的空闲空间被占用时，这个阀值就会被突破，而且可能在空闲空间只被使用了三分之一时就建议进行重组了。

第 12 章

数据库管理系统相关的索引限制

- 关系型数据库管理系统对索引施加的限制
- 索引列的个数上限及其最大长度
- 变长列
- 单表的索引数量上限
- 索引大小的上限
- 索引锁定
- 索引行空值压缩

简介

索引设计在很大程度上与数据库管理系统无关，毕竟最重要的是确定哪些表列应当被复制到索引中。尽管如此，客观上仍然存在一些数据库管理系统及版本相关的限制。

索引列的数量

能够复制到索引上的列的个数上限在 16 至 64 之间。并非每个人都将这视为一个问题。Gulutzan 和 Pelzer（1，第 231 页）提出了一个出人意料的建议，如下。

> 针对所有数据库管理系统的总体建议为：在一个复合索引中最多使用 5 列。虽然你能够确定数据库管理系统至少能够支持 16 列，但是 5 列是一些专家所认为的合理上限。

232 索引列的总长度

复制到索引的列的总长度存在一个上限，该上限的值取决于数据库管理系统。随着宽索引变得越来越流行，这一上限在数据库管理系统的新版本中似乎在变大。例如，在 DB2 数据库的 z/OS V7 版本中，该上限为 255 个字节；而在 2004 年推出的 V8 版本中，该上限被增大为 2000 个字节。另一个实际的限制是索引页的大小。如第 11 章中所讨论的，如果每个叶子页的空闲空间不够存放两个索引行，那么叶子页的分裂可能在重整后不久就又开始了。如果插入操作并非插入至索引末端，那么比叶子页的长度大 20%（对于 4KB 大小的页而言是 800 个字节，对于 8KB 大小的页而言是 1600 字节）的索引行可能意味着频繁的索引重整。

变长列

如果数据库管理系统将变长的列按照其实际长度紧挨着填满索引，那么一个宽索引中的索引行可能很容易就超出限制了。例如，在 DB2 数据库的 z/OS V7 版本中，由于这个原因，基于 VARCHAR 列的宽索引不常被使用；不过在 V8 版本中，CREATE INDEX 语句中有了一个 NOT PADDED 选项。

许多数据库对于非数值型的列都使用 VARCHAR 列，SAP 就是其中的一个知名的例子。如果当列被复制到索引时仍保持变长，那么这类数据库的索引调优空间就被根本性地提升了。

单表索引数量上限

在单表索引数量限制方面，许多数据库产品要么没有上限，要么上限太高以至于无关紧要。例如，SQL Server 2000 允许每张表上最多有 249 个非唯一非聚簇索引或约束，以及 1 个聚簇索引。随着索引数量的增加，访问路径选择过程所花费的时间将会增加，不过这一花费只有在每次执行都需进行一次路径成本估算的情况下才会变得显著。

索引大小上限

典型的索引大小上限为几 GB，而且这一上限正在持续增大。就像大表一样，大索引通常是分区的，这样能够使执行维护程序的成本最小化，并且能将索引分散到多个磁盘驱动器或 RAID 组上。

索引锁定

从更新的时间点到提交的时间点内，如果数据库管理系统给一个索引页或者一个索引页的一部分（如一个子页）加了锁，那么该索引页或子页很可能会成为瓶颈，因为插入操作将会变为顺序的。例如，SQL Server 2000 就 233 是这样做的，但如果上锁的粒度仅为一行，那么这可能不会成为一个问题。

在 DB2 数据库的 z/OS 版本中，使用闩锁来保证索引页的物理完整性。当用闩锁对一个缓冲池中的页加锁时，实际上是在数据库缓冲池中进行了一次置位操作，当释放闩锁时再进行重置。一个页只有在读取或修改期间才会被加上闩锁，在当前的处理器条件下耗费时间不到一微秒。而数据完整性是通过对索引行所指向的表页或表行加上普通锁来保证的（仅对数据上锁）。当程序修改一个表行或表页，这些锁一直不会释放，直至修改被提交。

几年前，DB2 采取的方式是使用普通锁锁住被修改的页或子页，直至提交点再释放。为了避免由此带来的长时间锁等待，普遍采取的策略是避免永久增长的键，如通过将 hh.mm.ss 格式的时间戳转变为 ss.mm.hh 格式。在使用了闩锁（或索引键锁定）的情况下，在索引的末端进行插入操作不太可能引起排队，除非插入事务的频率达到每秒几千次。

避免永久增长的索引键是另一个正在慢慢消失的神话。在当前的实现方式下，一个在永久增长的键上的索引对性能是有好处的：通常这样能够避免同步读，因为最后的那个叶子页很可能还在数据库缓冲池中。另外，如果索引行不移动或其大小不增长，那么就不需要分配任何空闲空间，也不需要进行结构重整了。

索引行压缩

Oracle 数据库的索引并不总是对表中的每一行数据都存有相应的索引行。那些索引列全部为空值的数据行在索引中不存在对应的行。由于这是一个需要谨慎对待的问题，我们将再次引用 Gulutzan 和 Pelzer 的阐述（1，第 251 页），如下。

> 对于该定律的一个首要的臭名昭著的异常是，Oracle——仅仅只是 Oracle——拒绝存储 NULL 值。所以，如果你有一张表（Table1），表中包含两列(column1,column2)，且在 column2 上建了索引，那么

- Oracle 能够非常快地执行

```
INSERT INTO Table1 VALUES (5, NULL)
```

因为数据库管理系统根本无须更新索引行。

- Oracle 能够较快地执行

```
SELECT * FROM Table1 WHERE column2 < 5
```

因为 column2 上的索引有更少的键值，从而索引层级可能更低。

- Oracle 无法快速执行

```
SELECT * FROM Table1 WHERE column2 IS NULL
```

234

因为在 column2 为 NULL 的条件可能为真的任何场景下，数据库管理系统都无法使用索引。

这是一个好的权衡。然而不幸的是，这造成了 Oracle 程序员和其他程序员之间的看法差异：Oracle 程序员会认为可以为空的列是好的，而其他数据库管理系统的程序员则会认为 NOT NULL 能够提升性能。

在某些场景下，Oracle 的实现方式确实能够以较低的维护成本支持很小的索引，我们将在下一章中讨论一个这样的例子。然而，这也可能制造出一些无法如我们所期望的那样利用上索引的场景。幸运的是，我们现在能够通过创建一个基于内置函数 NVL 的函数索引来阻止索引行压缩：

```
create index cust3 on cust (nvl(fname, 'null'),
 lname, cno, city)
```

这意味着索引列 FNAME 永远都不会有 null 值——当表列为 NULL 时，相应的索引列将包含一个字符串‘null’。现在，谓词条件 FNAME=‘null’将能够读取索引片了。

数据库管理系统索引创建举例

在各种常用数据库管理系统之间比较索引所支持的各种特性是一件有趣的事情。以下列举了不同数据库所允许的索引参数值和限制，以及索引创建举例，另外还列举了一些其他的索引选项，这将在接下来的章节中讨论。

DB2 for z/OS

```
CREATE [UNIQUE] INDEX index_name [CLUSTER]
ON table (column1 [DESC], column2 [DESC]...)
PCTFREE n, FREEPAGE n
```

页大小4KB
最多64个索引列
长度最多2000个字节（V7版本：255个字节）

DB2 for LUW

235

```
CREATE [UNIQUE] INDEX index_name [CLUSTER]
ON table (column1 [DESC], column2 [DESC]...)
[INCLUDE (columna, columnb...)]
PCTFREE n
```

页大小4KB、8KB、16KB或32KB
最多16个索引列
长度最多1024个字节

Oracle

```
CREATE [UNIQUE] INDEX index_name
ON table (column1, column2...)
PCTFREE n
```

或者函数

选项：索引组织表
所有叶子页 = 表

页大小（DB_BLOCK_SIZE）4KB、8KB、16KB、32KB或64KB
最多32个索引列
长度最多为DB_BLOCK_SIZE的40%

Microsoft SQL Server

```
CREATE [UNIQUE]
[CLUSTERED|NONCLUSTERED]
INDEX index_name
ON table (column1, column2...)
WITH  FILLFACTOR = fillfactor
```

所有叶子页 = 表

100 - PCTFREE

页大小8KB
最多16个索引列
长度最多900个字节

第 13 章

数据库索引选项

- 索引行压缩与异常情况
- 索引键以外的其他索引列
- 唯一性约束
- 从两个方向扫描索引
- 索引键截断
- 基于函数的索引
- 索引跳跃式扫描
- 块索引
- 数据分区的二级索引

简介

随着时间的推移，索引的选项也越来越多。本章覆盖了对于传统索引而言比较重要的选项。除此以外，一些与应用类型相关的索引结构也越来越多，比如那些针对地理数据库的索引结构。

在上一章中，我们已对常见的几个数据库系统的索引选项进行了讨论。

索引行压缩

在非关系型 DBMS 中，索引行压缩是一个普遍且常用的选项。如果被索引的字段的值是固定的（典型的比如空字符串或者 0），那么就可以避免创建索引段或者索引行。此外，还可以通过编写退出程序来进行一些更复杂的处理。或许在目前大家并不会投入更多的精力来减少索引的大小，因为在许多关系型产品中并没有实现这个功能。

事实上，这一选项对于那些为了发现异常情况而创建的索引很有用，不是为了节省磁盘空间，而是为了减少不必要的索引维护代价。一种比较常见

的自己实现索引行压缩的方法是，使用触发器来维护一个小型的类索引表，如图 13.1 所示。

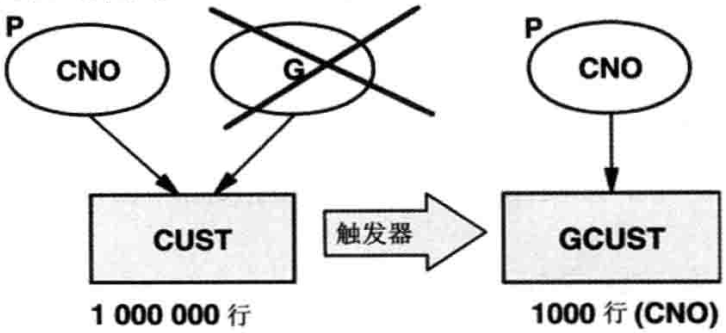


图 13.1　当 DBMS 不支持索引行压缩时的处理方法

金牌客户，占客户总数的千分之一，该类客户 G 列的值为 1，而普通客户 G 列的值为 0。如果 DBMS 支持索引行压缩，那么或许就应该在 G 索引上使用这一选项。如果 DBMS 不支持，那么为了达到减少维护索引代价的目的，我们需要建一张独立的类索引表 GCUST，用于存储所有金牌客户记录的主键。

正如在支持索引压缩时为 G = 1 的客户所创建的索引一样，这个类索引表只有在以下事件发生时才需要进行维护：

- 一条金牌客户记录写入 CUST 表。
- 一条金牌客户记录从 CUST 表删除。
- 一个普通客户升级成为金牌客户。
- 一个金牌客户降级成为普通客户。

如第 12 章所述，Oracle 使用 NULL 值来实现索引行压缩。在上述例子中，普通客户 G 列的值可以使用 NULL 来代替数值 0。这样整体的性能会比维护一个类索引表要好（不需要进行连接），而且还不需要触发器来维护 GCUST 表。然而，我们不推荐在实践中使用 NULL 来代替一个特定的值，因为从长远来看，这可能会导致应用系统错误。

索引键以外的其他索引列

*索引键*决定了这一索引行在索引结构中的位置。当索引键被修改后，DBMS 会删除原来的索引行，并将其插入到新的位置上。在最差情况下，索引行会被移动到其他的叶子页上。

DB2 for LUW 允许将 CREATE INDEX 语句中的列分为两组：*索引键列*和*非索引键列*（CREATE INDEX 中的 INCLUDE 选项，见第 12 章）。举个例子，在索引(A,B,C,D)中，索引键列可能是(A,B)，而(C,D)是非索引键列。

239

于是，更新 C、D 列的值并不会移动索引行的位置。这样或许能为每一个更新的行节省一次磁盘驱动器的随机读取（10 ms）。然而，需要注意的是，如果列 A 和列 B 能够保证索引的唯一性，那么即便 C 列和 D 列是索引键列，更新 C 列或者 D 列也不太可能导致索引行被移至另一个叶子页上。

区分索引键列的另一个好处是减少非叶子页的数量：非叶子页只会保存索引键列。这样一来，将非叶子页缓存在内存中所需的数据库缓冲池的空间也会更少。

在非键值索引列提升更新操作的响应速度的同时，那些在 WHERE 和 ORDER BY 子句中涉及非键值索引列的 SELECT 查询可能会因此受到影响，SQL 13.1 与图 13.2 展示了一个这样的例子。由于列 C 和列 D 在索引中的位置不再是确保有序的，访问路径将只有两个匹配列且需要进行一次排序，这是不满足性能要求的。

SQL 13.1

```
SELECT      D
FROM        TABLE
WHERE       A = :A
              AND
            B = :B
              AND
            C = :C
ORDER BY    D
```

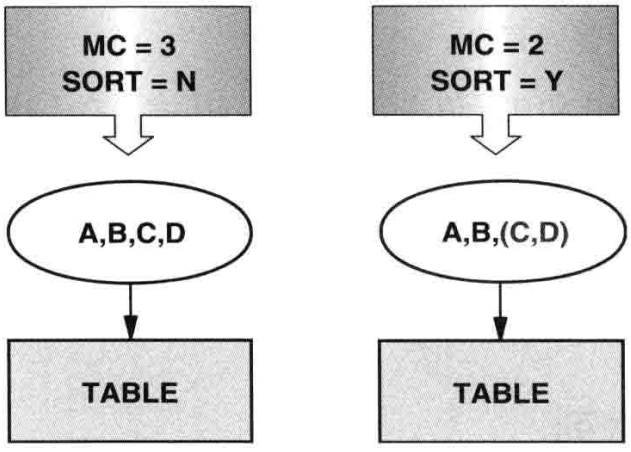


图 13.2　非键值索引列可能导致 SELECT 调用运行得更慢

非键值索引列最大的收益来源于能够把非键值索引列添加到主键索引中。

如果 DBMS 不支持非键值索引列（或者如下所述，部分索引列的唯一性可以通过约束来保证）那么就不能将其他列添加到主键索引中了，因为数据库管理系统只支持整个索引列的唯一性。在这种情况下，如果将列

240

CNAME 添加到 CNO 的主键索引中，那么 CUST 表中就有可能存在两条拥有相同 CNO 值的记录。在一个不支持非键值索引的 DBMS 中，为了针对 SQL 13.2 中的 SELECT 语句提供宽索引，我们就不得不创建一个额外的索引(CNO,CNAME)，如图 13.3 所示。

SQL 13.2

```
SELECT      CNAME
FROM        CUST
WHERE       CNO = :CNO
```

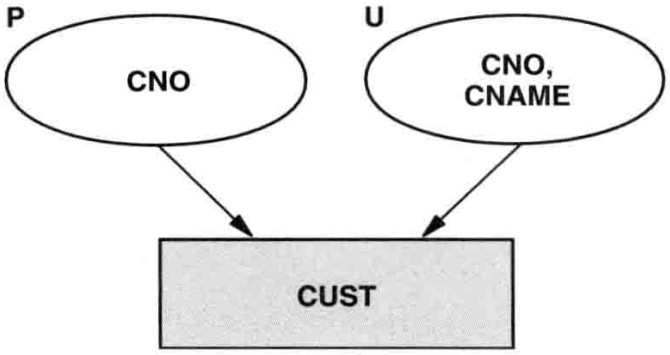


图 13.3　当 DBMS 不支持非键值索引列时的处理方法

唯一约束

Oracle 数据库系统提供了不同的方案来解决图 13.3 所描述的问题。在 Oracle 中并不需要索引 CNO，因为索引(CNO,CNAME)可以被用来约束主键 CNO 的唯一性。这可以通过引入 PRIMARY KEY 约束来做到。当一个候选键必须保证唯一时，可以使用 UNIQUE 约束连同以候选键作为前导列的索引来确保唯一性。

需要注意的是，在 Oracle 中，创建索引(CNO,CNAME)时不能使用关键字 UNIQUE，尽管它实际上确实是唯一的（它包含了主键）。

从不同的方向扫描数据库索引

如果 DBMS 能够逆向扫描索引（这已变得越来越普遍，Oracle、SQL Server 和 DB2 for LUW 在部分场景下提供了这样的功能，DB2 for z/OS 从 V8 版本也开始支持这一功能），那么从不同的方向扫描就可以在同一个索引上实现了，不再需要通过排序来完成。否则，就需要创建两个索引来避免排序操作，如图 13.4 所示。

241

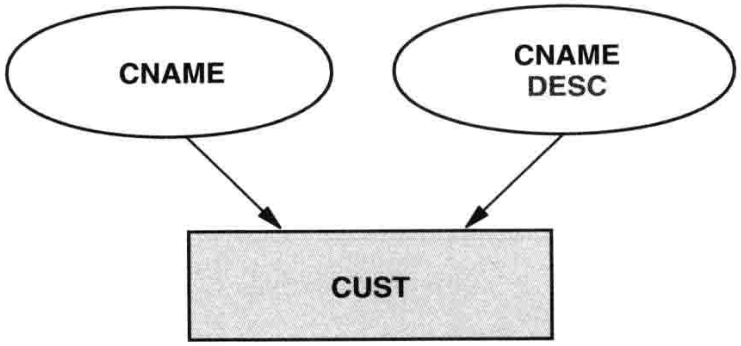


图 13.4　当 DBMS 无法逆向扫描索引时的处理方法

避免这类冗余索引的一个常用的方法就是，将那些已经给用户展示过的索引键值保存在辅助表中。不过这种方式有一个限制，就是用户不能逆向扫描超出辅助表中的起始点。

注：逆向扫描的功能可能需要显式地指定该选项，比如在 DB2 for LUW 中，需要在 CREATE INDEX 子句中指定 ALLOW REVERSE SCANS 选项。

索引键截断

索引键截断（比如在 DB2 for z/OS 中的实现）是指在非叶子页上 DBMS 只保存了部分索引键列，这部分索引键能够决定下一层级的索引节点。正如上文所讨论的非键值索引列，这样能够有效地减少非叶子页的数量。

基于函数的索引

假设 CUST 表中 CNAME 列的值是大小写混合的，但 SELECT 查询语句的绑定变量传入值是大写的（如图 13.5 所示）。比较常见的解决方案是在 CUST 表存储两份 CNAME 的值，一个是大小写混合的原始值，一个是大写值。然后在后面的这个列上建立一个索引，这两列的值必须保持同步，通过使用触发器或通过生成列的方式。

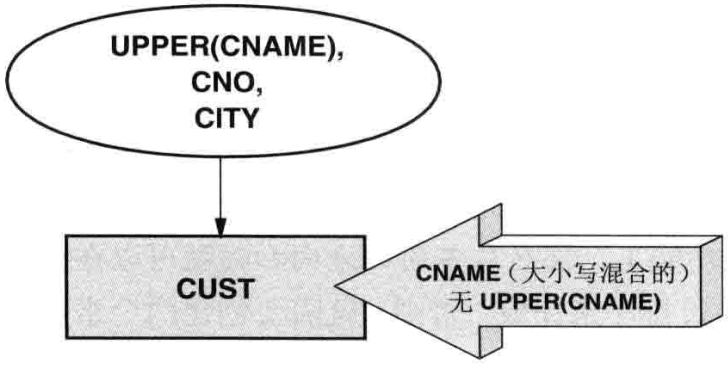


图 13.5　基于函数的索引避免了在表上创建冗余列的需要

在 Oracle 数据库中，可以建立一个函数索引：

```
CREATE INDEX .... ON CUST (UPPER (CNAME), CNO, CITY)
```

当谓词包含了相同的函数时，如 SQL 13.3 所示，优化器就会考虑使用这个基于函数的索引。这样就避免了在 CUST 表上创建冗余列的需要。

242

SQL 13.3

```
SELECT      UPPER (CNAME), CITY, CNO
FROM        CUST
WHERE       UPPER (CNAME) BETWEEN :hv1 AND :hv2
ORDER BY UPPER (CNAME)
```

注：如果结果集需要 CNAME 字段的原始值，那么需要将 CNAME 列追加到索引中才能避免回表扫描。

```
CREATE INDEX .... ON CUST (UPPER (CNAME), CNAME, CNO, CITY)
```

Oracle 9i 支持在 CREATE INDEX 子句中使用表达式或者函数（甚至是用户自定义的表达式或函数），SQL Server 2000 支持在计算列上添加索引。

索引跳跃式扫描

假设 CUST 表上有一个索引(CCTRY,BDATE,CNAME)，但没有以 BDATE 为前导列的索引。大多数的优化器可能会为 SQL 13.4 选择进行全索引扫描，但 Oracle 有可能会选择使用*索引跳跃式扫描*，多个由 CCTRY 和 DBATE 所定义的索引片段（每个国家一个索引片段）会被并行地读取。跳跃式扫描所跳跃的是第一个索引列。如果 CCTRY 的基数很低，那么新建一个以 BDATE 和 CCTRY 开头的索引将会是多余的，比如这样的场景：分布在 20 个不同国家的一百万名客户。

SQL 13.4

```
SELECT CCTRY, CNAME
FROM   CUST
WHERE  BDATE = :BDATE                FF =  0.01%
```

243

全索引扫描(CCTRY,BDATE,CNAME)：

$$1 \times 10\ \text{ms} + 1\,000\,000 \times 0.01\ \text{ms} \approx 10\ \text{s}$$

理想索引扫描(BDATE,CCTRY,CNAME)：

$$1 \times 10\ \text{ms} + 100 \times 0.01\ \text{ms} = 11\ \text{ms}$$

索引跳跃式扫描(CCTRY,BDATE,CNAME)：

$$20 \times 10\ \text{ms} + 100 \times 0.01\ \text{ms} = 201\ \text{ms}$$

块索引

DB2 for LUW V8 版本提供了一个多维聚簇的功能：即互相关联的表记录存储在同一个数据块上。尽管这是一个表设计相关的可选项，但这并不妨碍从索引设计的角度去考虑这一选项，因为在某些情况下，它可以被用来替代宽索引。

假设有一张 ORDER 表，该表保存着某公司在 50 个国家销售 100 种商品的订单数据。为了使用多维聚簇，我们在 CREATE TALBE 子句中添加了下面的语句：

```
ORGANIZE BY DIMENSIONS PRODUCT AND COUNTRY
```

于是，该表将由块组成，每个块由连续的页组成（2 ~ 256 个页）。每个块中只包含了在所有维度上有相同值的数据记录，比如一些块包含了国家 1 和商品 1 的记录。一个 PRODUCT 列上的块索引拥有指向包含某商品相关记录的所有块的指针；同样，一个 COUNTRY 列的块索引拥有指向包含某国家相关记录的所有块的指针。

当 SELECT 语句包含 WHERE PRODUCT = :PRODUCT AND COUNTRY = :COUNTRY 谓词时，优化器很可能会选择在两个块索引上进行与（AND）操作，然后再读取所有符合条件的记录。

传统的方式是使用一个以 PRODUCT 和 COUNTRY 为开头的聚簇索引或宽索引。虽然重组之后所有的扫描都是顺序的，但随机访问的次数会由于插入而增加。多维度聚簇的好处就是插入操作并不会降低聚簇因子，在大量插入后并不需要进行表重组。而明显的缺点是，在维度的基数比较大而一些列值又比较罕见的情况下，这将导致较低的磁盘利用率，因为可能有许多块只包含一条记录。

数据分区的二级索引

对于实施了分区的大表，出于可用性的考虑，很可能会将表上的索引也进行分区。能够并行地重组分区及其索引是主要的好处之一。DB2 for z/OS V8 版本提供了这一选项。

244　假设 INVOICE 表有 50 个分区，如图 13.6 所示。

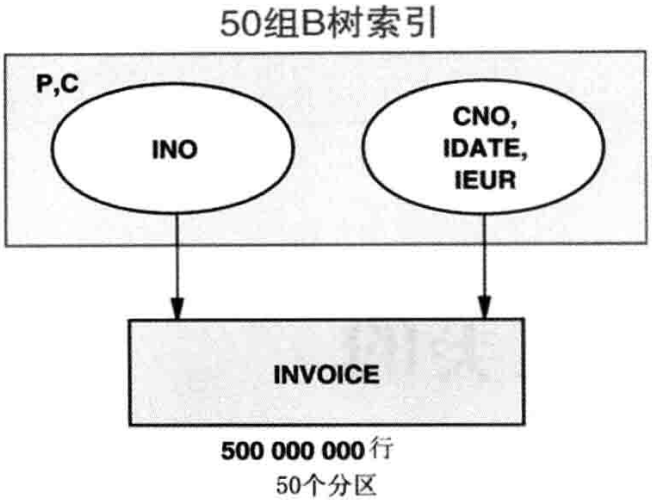


图 13.6　数据分区的二级索引

```
CREATE TABLE INVOICE
              PARTITION BY (INO)
              (PARTITION1 ENDING AT (10000000),
              PARTITION2 ENDING AT (20000000)...
```

如果索引(CNO,IDATE,IEUR)是一个分区索引，那么将会创建 50 个 B 树索引（索引分区），每个表分区上一个 B 树索引。开发人员需要特别注意这一点，因为如果查询语句没有针对 INO 字段的谓词，那么查询的随机索引访问次数将会是 50。SQL 13.5 所示的 SELECT 语句将扫描 50 个 B 树索引。

SQL 13.5

```
SELECT      INO, IEUR
FROM        INVOICE
WHERE       IDATE > :IDATE
ORDER BY IDATE
```

为了减少被扫描的 B 树索引的个数，需要在 SQL 语句中指定包含 INO 字段的谓词，参见 SQL 13.6。

SQL 13.6

```
SELECT      INO, IEUR
FROM        INVOICE
WHERE       IDATE > :IDATE AND INO > :INO
ORDER BY IDATE
```

练习

13.1. 调查你当前使用的 DBMS 版本关于索引的限制和高级选项。

第 14 章

优化器不是完美的

- 优化器并不总能看见最佳方案
- 匹配及过滤问题
- 非 BT 谓词
- 不必要的排序
- 不必要的表访问
- 优化器的成本估算有时是错误的，甚至可能是灾难性的（我们该如何协助优化器）
- 范围谓词
- 偏斜分布
- 相关列
- 部分索引键的特性
- 优化器成本估算公式的思考
- I/O 估算
- CPU 估算
- 工具及缓存的影响
- 在估算相关的问题上协助优化器
- 每次都用真实值进行优化
- 访问路径提示
- 冗余谓词
- 伪造统计信息
- 修改索引
- 优化器问题对索引设计的影响

简介

有时候，一个 SELECT 调用在合适索引已创建的情况下仍然运行得难

以置信得慢。如果耗时的主要部分是服务时间而非排队时间，那么可能是因为优化器选择了一个较差的访问路径。关系型数据库的优化器有一个很困难的任务，即它必须基于仅有的代表表及索引特征的统计数据来选择正确的访问路径，而不能进行任何实际测量。现今的优化器在选择访问路径时并不向数据库发起任何 SELECT 调用，它们会以一种有限的方式在执行时验证它们的假设。例如，它们可能在观察了页访问模式后为一个索引或表开启顺序预读。将来，最好的优化器也许还能在执行时确定实际的过滤因子。

246

对于一个简单的 SELECT 操作，需要做的最重要的决定是索引的选择（或是在多索引访问下的多个索引的选择）及其使用方式（匹配列、过滤列、顺序预读）。对于连接来说，连接的方式及表访问顺序也同样重要。

所有重要的关系型数据库管理系统现在都有一个基于成本的优化器，它们能识别出一些合理的方案，然后对其进行成本估算，大部分是对于本地响应时间的预估——一个 CPU 时间以及同步 I/O 时间的加权平均值。该估算是基于一个工具组件所收集的统计信息，如表的大小、表上索引的大小以及索引列值的分布。许多这些分布值是可选的，对每个列或列组合的统计数据集最多可能包含以下数据：

- 不同值的数量（基数）
- 第二大和第二小的值
- *N* 个最普遍的值及其出现频率（一个给定值对应的行数）
- 有 *N* 个柱的直方图（2%的值低于 10，5%的值低于 20 等；或百分比值 2%、4%、6%等）

最好的优化器的成本计算公式包含许多变量（而在 QUBE 中只有两个变量——TR 和 TS）。这些变量中可能还包括硬件信息，比如处理器速度、磁盘驱动速度及数据库缓冲池的大小。最老的基于成本的优化器已经存在 31 年了，开发人员一直在持续地优化它们。然而，它们有时还是会选择一个完全错误的访问路径，即使是在处理一个看起来完全无辜的 SQL 语句时。现在我们将讨论为什么会如此。

本质上，我们需要学着去承认两个基本的问题（我们将逐个处理它们）：

1. 优化器并不总能看见最佳方案。
2. 优化器的成本估算可能大错特错。

优化器并不总能看见最佳方案

匹配及过滤问题

第 6 章中定义的困难谓词，如那些*无法参与索引片的定义的列*，曾经是导致优化问题的最普遍的原因。这类问题的发生率在近几年已有所降低，原因有两个：

247

- 在很长一段时间内，优化器学会了以一种高效的方式处理更多的谓词，通常是通过将这些谓词转换为一种对优化器更友好的形式，然后再进行访问路径选择的方式做到的。除此之外，还有一些为特定优化器提供查询重写建议的独立工具。
- 在优秀的公司里，每一名 SQL 程序员都拥有一份缺陷列表，该列表包含了对于当前的优化器版本而言最常见的困难谓词以及相应的规避建议。

每一个数据库管理系统似乎对于困难谓词有各自不同的称谓，在 SQL Server 的书中称其为*非搜索参数*；在一些 Oracle 的书籍中称其为*索引抑制*；而在 DB2 for z/OS 中则称其为*无法索引谓词*。

除了匹配问题之外，优化器可能还有过滤问题。对于*真正的困难谓词*，数据库管理系统*甚至不能做索引过滤*。这意味着即便该谓词列或这些谓词列已经复制到索引上了，数据库管理系统仍必须读取表行以校验真正困难的谓词条件。在 DB2 for z/OS 中，真正困难的谓词被称作*二阶段谓词*。

最著名的例子之一便是 :hv BETWEEN COL1 AND COL2。例如，通常当主机变量为 CURRENT DATE 并且列 COL1 和列 COL2 是一段时间的起始和结束日期时，这么写很方便。在一些场景下，程序员必须将这一真正困难的谓词重写为 COL1 <= :hv AND COL2 >= :hv。某些优化器可能已经自动做这层转换了。

非 BT 谓词

WHERE 语句中的布尔操作符 OR 经常会导致令人不愉快的意外，这是由于它可能制造一个对于优化器来说太难的组合谓词。如第 6 章中所描述的，当校验结果为假时，非 BT 谓词并不能用来排除一行。因此，它们强加了一个非常严重的限制，即*它们只有在采用多索引访问的方式时才能参与定义索引片的过程*。如果当一个谓词为假时能排除一行，那么该谓词便是 BT 谓词。由此，如果一个 WHERE 语句中仅仅包含 AND（无 OR），那么所有的谓

词都将是 BT 谓词。

在许多浏览类事务及批任务重定位中，很容易使用类似 SQL 14.1 中的游标来表达对尚未展示的结果集进行展示的需求。至少有一种程序生成器会为浏览类事务生成类似的游标。

SQL 14.1

```
DECLARE CURSOR141 CURSOR FOR
SELECT      LNAME, FNAME, CNO
FROM        CUST
WHERE       (LNAME = :LNAMEPREV
            AND
            CNO > :CNOPREV)
            OR
            (LNAME > :LNAMEPREV
            AND
            LNAME <= :LNAMEMAX)
ORDER BY    LNAME, CNO
WE WANT 20 ROWS PLEASE
```

用户输入 LNAME 的前三个字母，如 JON，应用程序会将以下值传递至主机变量：

```
LNAMEPREV
```

在首次事务中为 JONAAA...，在随后的事务中为所展示的最后一条记录的 LNAME 值：

```
CNOPREV
```

在首次事务中该变量的值为 0，在随后的事务中该变量的值为已展示的最后一条记录的 CNO 值： 248

```
LNAMEMAX
该变量的值为 JONZZZ...
```

当索引(LNAME,CNO,FNAME)可用时，最佳的访问路径自然是 MC=2，仅需访问索引，且无排序。于是每次事务产生的访问次数降至最低，对于一个每屏显示 20 行的浏览类事务而言，TR = 1，TS = 19。

但不幸的是，该 WHERE 语句中没有 BT 谓词。如果优化器无法通过重写去除 OR，那么优化器必须在*全索引扫描*（MC = 0）和*多索引访问*之间选择一个。对于后者，如果数据库管理系统从索引片上仅收集了指针，那么访问路径便无法做到只访问索引。除此以外，整个结果集都将被物化，因为需要对指针进行排序以满足 ORDER BY 的要求。如果结果集包含 1000 行，那这就意味着 1000 次不必要的随机表访问。本章末尾的习题 14.1 考虑了对 SQL 14.1 进行重写以去除 OR。

下一个讨论的 WHERE 条件没有 SQL 14.1 中的 SQL 那么糟。谓词 P1 和 P2 是 BT 谓词，假设谓词 P1 和 P2 都不困难，那么访问路径至少为 MC = 2。

```
WHERE P1 AND P2 AND (P3 OR P4)
```

若优化器没有发现这个最好的方案，那么我们如何协助它呢？不幸的是，该 SQL 语句必须被重构。甚至需要考虑将一个复杂的游标拆分为几个游标。优化器现在已经正在学着处理如下这类转换：

```
COLX = :hv1 OR COLX = :hv2    转换为
COLX IN (:hv1, :hv2)
```

然而，拆分游标是一个更复杂的任务，因为需要在每个 SQL 调用之间加入应用程序代码。

249 SQL 14.1 是一个典型的以拆分游标为最佳解决办法的例子。用 UNION ALL 替换 OR 可能会造成非必要的排序以满足 ORDER BY，但是省略 ORDER BY 可能是很危险的。许多事情可能会在将来改变。为了避免排序以及所有多余的访问，需要编写如 SQL 14.2 中所展示的两个游标。

SQL 14.2

```
DECLARE CURSOR142A CURSOR FOR
SELECT      LNAME, FNAME, CNO
FROM        CUST
WHERE       LNAME = :LNAMEPREV
            AND
            CNO > :CNOPREV
ORDER BY    LNAME, CNO
WE WANT 20 ROWS PLEASE

 EXPLAIN: MC = 2, index only, no sort

DECLARE CURSOR142B CURSOR FOR
SELECT      LNAME, FNAME, CNO
FROM        CUST
WHERE       LNAME > :LNAMEPREV
            AND
            LNAME <= :LNAMEMAX
ORDER BY    LNAME, CNO
WE WANT 20 ROWS PLEASE

 EXPLAIN: MC = 1, index only, no sort
```

应用程序首先打开游标 CURSOR142A 并发起最多 20 次数据提取调用。如果游标已经到了结果集末尾，那么游标将被关闭且游标 CURSOR142B 将被打开。程序将一直提取直至游标到达结果集末尾或者数据填满一屏。在结束之前，程序会保存最后一行数据的 LNAME 和 CNO 值以备下一个事务使用。

此程序不会进行不必要的访问，它永远从上一次事务退出时所对应的索引位置开始执行。根据 QUBE 算法，本地响应时间将为：

$$1 \times 10\ \text{ms} + 19 \times 0.01\ \text{ms} \approx 10\ \text{ms}$$

让我们最后一次回到那个搜索身材高大的男士的查询。

在这个场景中，如果应用程序是在一个事务中提取出整个结果集，那么用 UNION 替换 OR 将会是一个好的解决方案，看起来似乎确实是这样，因为并不存在“WE WANT n ROWS PLEASE”语句。现在，为这两个 SELECT 创建理想索引就很容易了，参见 SQL 14.3 和 SQL 14.4：

250

SQL 14.3

```
SELECT     LNAME, FNAME, CNO
FROM       CUST
WHERE      SEX = 'M'
           AND
           (WEIGHT > 90 OR HEIGHT > 190)
ORDER BY   LNAME, FNAME
```

SQL 14.4

```
SELECT       LNAME, FNAME, CNO
FROM         CUST
WHERE        SEX = 'M'
             AND
             WEIGHT > 90

UNION

SELECT       LNAME, FNAME, CNO
FROM         CUST
WHERE        SEX = 'M'
             AND
             HEIGHT > 190
ORDER BY     LNAME, FNAME
```

无法避免的排序

对于许多优化器而言，有些包含 ORDER BY 语句的游标太困难了，以至于即便通过所选索引访问的结果行的顺序是正确的，优化器仍会对结果行进行排序。以下两个场景就在基于 DB2 for z/OS V7 的缺陷列表中：

1. 一个包含 UNION 和 ORDER BY 的游标总是会导致排序，无论索引多么好。

2. 如果 ORDER BY 语句涉及非最外层表（优化器选择访问的第一张表）中的列，那么，一个包含 ORDER BY 的连接 SELECT 会导致一次排序。

一些著名的数据库大拿建议在那些无须 ORDER BY 仍能使结果集排序正确的语句中省去 ORDER BY。他们于是假定 DBA 总是会注意到访问路径上的任何变化。最安全的做法是通过拆分游标来排除不必要的排序，不过这
251 可能很费时间。而如果应用使用的是一个付费包，那么将无法做到这一点。一种可以容忍的折中方案是使用宽索引或理想索引，从而使整个结果集的物化过程足够快。

不必要的表访问

由 Gulutzan 和 Pelzer 最近所著的一本书（1）中有一个有趣的方法。书中描述了许多普遍使用到的调优技巧，并在 8 个不同的数据库管理系统中进行了尝试。若本地响应时间发生了大于 5%的改善，则认为所使用的方法有效。

宽索引（*覆盖索引*）当然是这些技巧中的一个。令人惊讶的是，方法有效的比例仅为 6/8。这 8 个产品中有两个没有选择只访问索引的访问路径。于是作者便提出了以下这些与仅访问索引相关的警示。

> 通常情况下，使用一个覆盖索引能够节省一次磁盘访问（有效比例为 6/8）。这一好处是能自动获得的，但只有当 select 列表中的列与该覆盖索引中的列完全匹配时才能生效。如果 select 语句中有函数或者文字，又或者列的顺序与索引中的不一致，那么你便会失去这一好处。
>
> ……

```
SELECT name FROM Table1
ORDER BY name
```

> 对以上语句，数据库管理系统是否会“聪明”地选择扫描覆盖索引，而非扫描整个表并排序呢？也许会。但是 Cloudscape 不会使用覆盖索引，除非为了 WHERE-语句而需要它，另外，如果结果集可能包含 Null，那么 Oracle 也不会使用索引。因此，如果你假设值为 NULL 的 name 列无用，并将该查询修改为如下语句，那么便能从覆盖索引中获益：

```
SELECT    name
FROM      Table1
WHERE     name > ''
ORDER BY  name
```

> 有效比例 7/8

SELECT 列表中的文字是一个众所周知的缺陷，Oracle 的 NULL 问题也一样。然而，人们对 SELECT 列表中列的顺序的抱怨更大。上文所引用段落的总结部分中的如下命题更是令人感到惊讶：

> 当语句中有连接或者分组操作时，数据库管理系统从不使用覆盖索引。

人们对于连接语句无法做到仅访问索引的抱怨，与我们在 DB2、Oracle 和 SQL Server 上的实际经验并不一致，也不符合在第 8 章中提出的建议。也许一些产品确实有这样的奇怪限制，但是这也可能是对于 EXPLAIN 的一个误解。如果 EXPLAIN 没有显示所期望的访问路径，那么很可能就会做出优化器没有看到最好方案（仅访问索引）的结论。但其实有可能优化器的确看到了这一“最佳”方案，只是由于评估值与真实值之间相差太多以至于优化器选择了错误的访问路径，因为根据优化器的估算，这一错误的访问方式是最快的。本章的第二部分将讨论此种类型的问题。 252

尽管如此，我们仍推荐你阅读上述引文的原著书籍（1），尤其当数据库管理系统手册或性能指南并未充分覆盖它们当前优化器的限制时。

优化器的成本估算可能错得离谱

使用绑定变量的范围谓词

```
WHERE COL1 > :COL1
```

如果优化器不在执行过程中估算过滤因子，即不在得知绑定变量的实际值后进行估算，那么优化器就必须赋予过滤因子一个默认的值。该默认值可能是一个常量，如 1/3；或者也可能以某种方式依赖于列值分布情况，例如，DB2 for z/OS 7 假定一个范围谓词的过滤因子随着列值分布的增大而减小。无论何种情况，得出的过滤因子值都可能是非常错误的。这可能导致优化器选择错误的索引或是错误的表访问顺序，还可能出现其他一些索引问题。

虽然在每次 SQL 调用执行的时候进行估算会耗费大量的 CPU 时间，但是取默认的过滤因子值经常会导致非常糟糕的成本估算。许多产品现在都提供一种折中的可选方案：在 SQL 语句第一次被执行时，优化器会被要求进行成本估算，然后将所选择的执行计划存储下来，以便复用。这一技术可能会比使用默认值做出更好的成本估算，但它在某种意义上也是无法预测的，因为首次执行时传入的绑定变量值可能是非典型的。

图 14.1 展示了一个可能导致非常糟糕的过滤因子估算的场景。两个谓词都使用了绑定变量的方式。对于范围谓词 A < :A，优化器必须使用一个基

于列分布情况的默认过滤因子值。举个例子，如果列 A 有一个较高的分布值，优化器可能会选择一个较低的过滤因子值，如 0.01%。

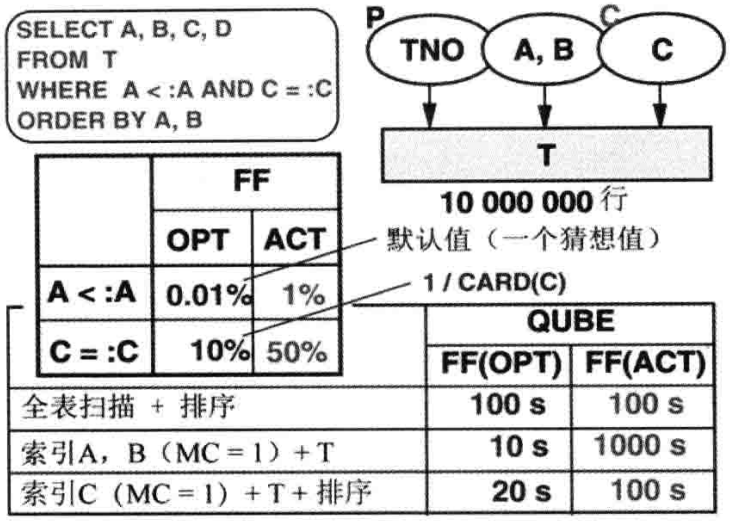


图 14.1　可能导致非常糟糕的过滤因子估算的场景

253　对于优化器而言，等值谓词 C = :C 更容易。若列 C 的基数值为 10，则优化器假定该过滤因子为 10%，该值对于平均场景的确是一个正确的值。然而，在最差输入情况下，C = :C 的过滤因子可能为 50%。

在图 14.1 中，第一个 FF 索引估值非常低，第二个也较低。基于这些糟糕的估算结果，优化器很可能会做出错误的选择。

如果在执行时能够得知绑定变量的实际值，我们就可以通过强制在每次执行时选择访问路径来帮助优化器。这种方式有可能能够解决上述问题，至少在优化器拥有一个展示了列 A 值分布的柱状图时是可以的。若列 A 的值分布如图 14.2 所示那样偏斜，那么知道列 A 的最小值和最大值也许还不足以解决问题。

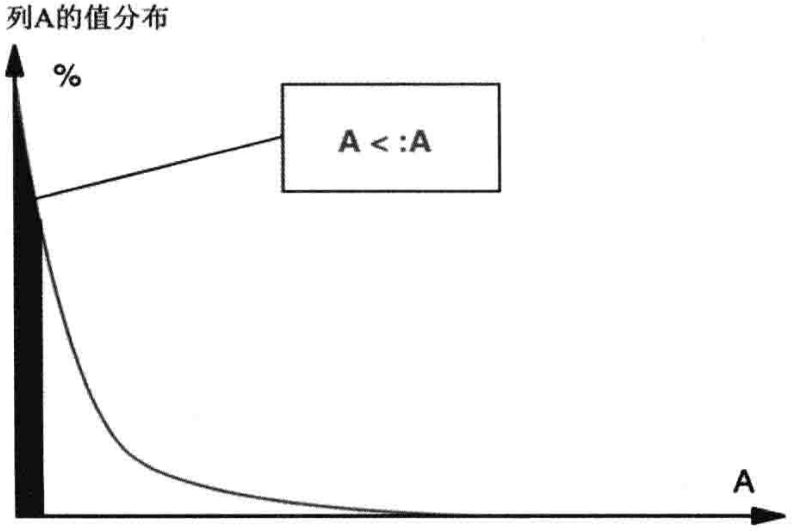


图 14.2　展示列 A 值分布的直方图

使用访问路径提示可以避免每次执行时进行成本估算带来的额外负载。若索引 C 确实对于所有输入值均为最佳选择，那么选择这一方式就是很合理的。

然而，在许多现实场景下，此例的最佳解决方案是创建比现有最佳索引更好的索引，如图 14.3 所示。这一索引改进不仅提升了 LRT，而且还排除了使用上述技巧的必要性。一个真正好的索引是很难被基于成本的优化器忽视的。

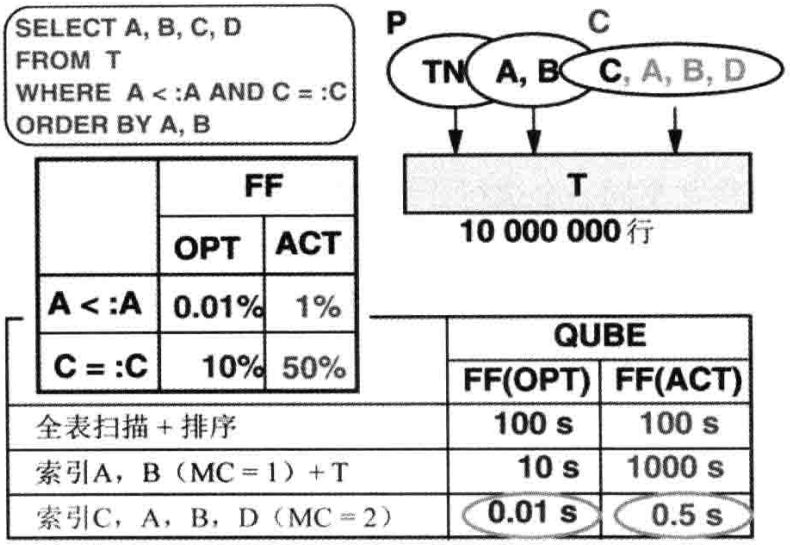


图 14.3　改进现有最佳索引

偏斜分布

在 SQL 14.5 所示的场景中，我们假设存在一个 SEX 列上的单列索引，且 99.9%的相扑选手为男性。

SQL 14.5

```
SELECT      AVG (WEIGHT)
FROM        SUMO
WHERE       SEX = :SEX
```

若访问路径并非是在每次执行时重新选择的，那么即便列 SEX 的值分布情况存储在系统表中，优化器仍无法利用该信息。当访问路径是在谓词为绑定变量的情况下被选出的，优化器只能假设过滤因子为 1/基数值 = 0.5。于是，优化器便会选择全表扫描。 254

若使用该索引的唯一目的是找出特例，那么解决该问题的方法很简单——即用文字替代绑定变量。对于谓词 SEX = 'F'，优化器能对 FF（过滤因子）做出正确的假定，即 0.1%。然后便很可能选择使用该索引。

若用户有时需要计算女性相扑选手的平均体重，有时又需要计算男性相扑选手的平均体重，也存在使得两种场景都使用最佳访问路径的解决方案，而且无须反复进行成本估算：

- 写两个 SELECT 语句，一个的谓词为 SEX = 'F'，另一个的谓词为 SEX = 'M'，程序根据用户的输入从中选择正确的 SELECT 语句。
- 如果 DBMS 没有静态 SQL 绑定的功能，那么可以使用一个能在缓冲池中存储语句及其访问路径（以便复用）的选项。注意保证优化器不会将文字转化为绑定变量，否则两个游标将共用同一个访问路径。在 Oracle 中，该选项被称为*游标共享*。若初始化参数文件 INIT.ORA 中 CURSOR_SHARING = FORCE，那么所有只在文字部分不同的语句将共享同一个执行计划。设置 CURSOR_SHARING = EXACT 能禁用游标共享，每一个文字值都将有一个单独的执行计划。若 CURSOR_SHARING = SIMILAR，则优化器可能根据统计值使用不同的执行计划。

结果，另一个旧的建议变成了传说：

不要为一个基数值低的列创建索引。

另外——你可能会想到以下问题：

255 *若某一单列索引对应列的基数值为 1，那么该索引是否应当被删除？*

例如，假设我们扫描了系统表以评估现有的索引，发现有一个单列索引上列的基数值为 1，那么该索引应当被删除吗？

这一场景当然是值得花时间思考的。我们的建议可以在本书的 ftp 网址上找到，详见前言部分。

在缓冲池中寻找执行计划比反复进行成本估算来选择访问路径要快得多。当前的硬件条件能够保证我们有几 GB 的 SQL 缓冲池可用。若每个 SQL 语句连同其访问路径占十几 KB，那么这样一个 SQL 池可以容纳一百万个访问路径。这对于一个类似 SAP 或 Peoplesoft 的系统而言很重要，这不是由于偏斜分布的问题，而是因为若无法复用访问路径，那么每次 SELECT 可能都会引起为估算其他可选访问路径的成本而带来的额外负载。

相关列

```
WHERE MAKE = :MAKE AND MODEL = :MODEL
```

仅当优化器知道列组合 MAKE,MODEL 的基数值时，优化器才能对此组合谓词做出好的估算。一些产品（如 DB2 for z/OS）在工具组件中提供了能够确定索引前 *N* 列的基数值的选项，用以更新优化器的统计信息，如索引(MAKE,MODEL,YEAR)前两列的基数值 CARD(MAKE,MODEL)。另一些产品（如 SQL Server 2000）会自动收集这些信息（在 SQL Server 中该信息被称为密度）。因此，若列 MAKE 和 MODEL 是一个多列索引上的前两列，优化器可能能够基于高度相关的列值分布信息来确定该组合谓词的正确过滤因子。请记住，只有 Ford 创造了 Model T!

若优化器不知道 CARD(MAKE,MODEL)，即索引前两列的组合基数值，那么优化器将假定一个非常小的基数值：

```
1 / CARD(MAKE)× 1 / CARD(MODEL)
```

若优化器不知道 CARD(MODEL)的值，它可能会使用一个通用的默认值（如 25），又或者，如果它知道 CARD(fullkey)，即全体索引键的基数值，那么可能会选择一个位于 CARD(MAKE)和 CARD(fullkey)之间的值；例如，若 CARD(MAKE)= 50，CARD(MAKE,MODEL,YEAR)= 2000，那么优化器可能会使用 1025（50 和 2000 的平均值）作为 CARD(MAKE,MODEL)的估值。上述两种方式都有可能会导致一个非常错误的过滤因子估值。

Mark Gurry（2，第 36 页）说，以他在 Oracle 方面的经验，这是一个普遍的问题：

> 总的来说，为什么基于成本的优化器会做出如此糟糕的决定呢？ 256
> 首先，我必须指出，糟糕的决策判断属于异常而非定律。本节中的例子表明，列是以个体而非整体来对待的。若列是以整体来看待的，那么基于成本的优化器将会意识到第一个例子中的被查询的每一行都是唯一的，无须 DBA 将索引重建为唯一索引。而在第二个例子中，若其中几个列的唯一值较少，且 SQL 请求的是这些唯一值中的大部分，那么基于成本的优化器通常会忽略该索引。尽管这些列组合在一起之后只会返回很少的行数，但忽略索引的情况还是会发生。

部分索引键的警示故事

以下警示故事是基于 Scandinavian 公司的一次实际经历的，不过有些细节被简化了，且其中包含一些假设的数字。

该公司买了一个监视工具用来发现异常慢的 SQL。在最初生成的报告

中，DBA 发现一个频繁被执行的 SELECT 语句运行得非常慢且耗费了大量的 CPU 时间。该 SQL 语句是一个简单语句，导致性能低下的问题很明显——没有合适的索引。该 SQL，即 SQL 14.6 中所示的 SQLA，运行时使用的是主键索引，该索引对于此调用而言甚至都不是半宽的。而且，匹配列只有一个（C1），在最差输入下，由该列所定义的索引片很厚。于是，一个包含三个匹配列的新索引被创建了，如图 14.4 所示，这一又慢又耗资源的 SELECT 语句性能得到了极大的提升。这是好消息。

SQL 14.6

SQLA

```
SELECT    many columns
FROM      SALES
WHERE     C1  = :C1
          AND
          C3  = :C3
          AND
          C10 = :C10
```

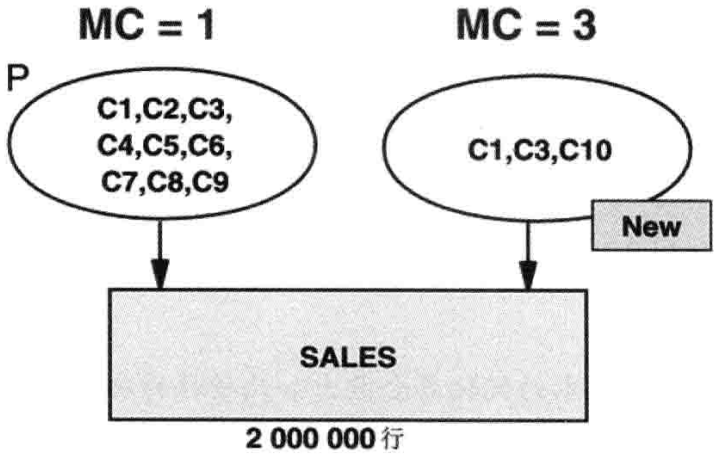


图 14.4　新索引使 SQLA 运行得更快

现在来看坏消息。在这一新索引创建后不久，一个销售经理正试图用一个 SQL 报表工具创建一份报表，他需要第二天一早带着这份报表赴德国开会。正常情况下，该报表很快便能生成完毕，但这一次花费了非常长的时间。
257 幸运的是，销售经理赶上了航班，但他向 IT 部门抱怨了这件事。DBA 非常震惊地发现，这一报表工具生成的 SELECT 语句，即 SQL 14.7 中所示的 SQLB，使用了新的索引而非主键索引。主键索引看起来更合适，因为使用它能有 8 个匹配列，而使用新索引只能有 2 个匹配列，如图 14.5 所示。当 DBA 发现了优化器的这一行为时，只能将新索引删除。

SQL 14.7

```
                                                              SQLB
SELECT     many columns and sums
FROM       SALES
WHERE      C1 = :C1 AND C2 = :C2 AND C3 = :C3 AND C4 = :C4 AND
           C5 = :C5 AND C6 = :C6 AND C7 = :C7 AND C8 = :C8
ORDER BY...
GROUP BY...
```

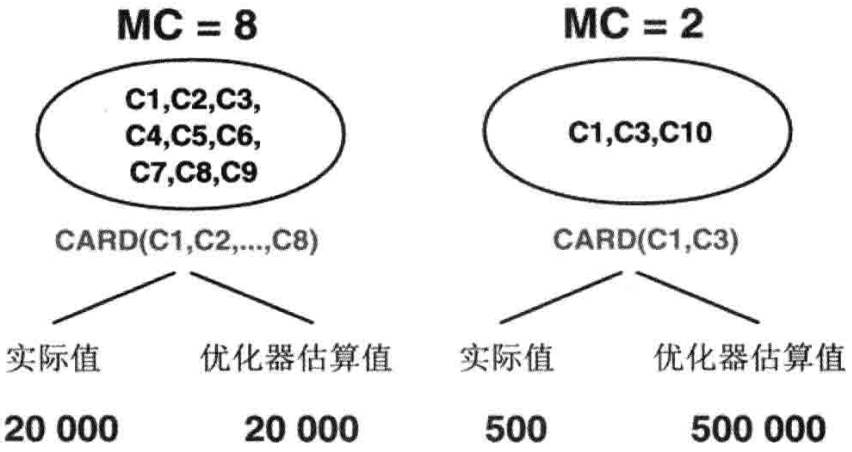


图 14.5　优化器好像什么都不知道

一个基于成本的优化器并不基于匹配列的数量来做决策，而是对几个可行的方案进行成本估算。在本例中，对于 SQLB，当考虑到新索引时，优化器需要知道组合谓词 C1 = :C1 AND C3 = :C3 的过滤因子。 258

如果列组(C1,C3)的实际基数值为 500，那么匹配谓词的平均过滤因子将会是 1/500 = 0.2%。每次索引及表访问的次数便为 0.002 × 2 000 000 = 4000。不过最差输入可能会导致比这多得多的访问次数。同样，若列组(C1,C2,C3,C4,C5,C6,C7,C8)（即主键索引的前八个列）的实际基数为 20 000，那么整个 WHERE 语句的平均过滤因子便为 1/20 000，表及索引的访问次数仅为 100 + 100。如果优化器知道这所有的信息，它就不会使用新的索引了。

那么优化器知道些什么呢？这取决于产品及其选项。不过通常默认情况下，优化器只知道第一个索引键的基数（对于新索引而言为 C1）和整个索引键的基数（对于新索引而言为 C1,C3,C10）的基数。当优化器需要知道(C1,C3)的基数时，它必须使用一个在这两个已知值之间的中间值计算方式。中间值的计算公式通常是保密的，然而我们并不需要知道这些细节就能证明优化器的估算在类似这样的场景下是非常错误的。优化器需要我们的帮助。

许多数据库管理系统产品都在那些为优化器创建统计信息的工具包（RUNSTATS、ANALYZE 或类似的功能）中提供了能够收集部分索引键信息的功能选项，支持采集的信息有基数、直方图、最常见的值及最少见的值等。至少，为所有索引采集其所有部分键的基数值可能是明智的。只为某

些索引、列或者列组采集统计数据有时会使情况变得更糟，因为对某些对象知道很多信息的同时对另一些对象采用默认值，这种方式还不如让优化器以一种一致的方式发生错误。例如，我们曾经就遇到过这样一个优化器选择了错误索引的场景，虽然优化器知道外键 CNO 的基数值为 5 000 000，但它不知道 CNAME 的基数值，从而使用了默认的基数值 25。如果优化器对 CNO 和 CNAME 都采用默认基数值，那么优化器也许能够做出正确的选择。

然而在实际生活中，依照上述建议，对于可选的统计信息做到同等处理不是一件容易的事。我们经常会由于默认的过滤因子值导致了错误的访问路径而对一个列收集更详细的统计信息。那么有可能做到同时对于其他相关列也做同样的操作吗？也许不能。我们更倾向于让数据库管理系统来对可选的统计信息做出智能建议。毕竟，优化器知道何时它应该使用默认的过滤因子值。优化器还能监视实际的过滤因子并将其与假设进行比较。那些最先进的产品（例如 DB2 for LUW 8.2 和 Oracle 10g）已经在朝这一方向前进了。这一领域的任何改进都将会受到欢迎，因为这将使得需要使用提示的场景减少。治本总比治标更好。

259 回到我们的例子，当定位到优化器奇怪行为的原因之后，判断如何使优化器对 SQLA 使用新的索引，同时对 SQLB 使用原来的索引就相当容易了。在这个特定的场景中，一个简单的方案是创建一个新的索引(C10,C1,C3)替代(C1,C3,C10)。现在对于 SQLB，优化器可能甚至不会对新索引进行估算，因为无匹配列。如果它做了估算，使用新索引的成本可能比使用旧索引要高得多。

成本估算公式

上文所讨论的优化器的过滤因子估算是成本估算过程中最薄弱的环节。如果一个过滤因子与实际值偏离很远（许多因素都会导致这一偏离），那么成本估算出的结果将是非常错误的。如果这些错误的成本估算并不以一种一致的方式发生，那么优化器将可能做出错误的选择。然而，也存在许多优化器做了完美的过滤因子估算的场景，在这些场景中，优化器清楚地知道，执行计划的每一步中有多少行将被访问。于是，成本估算的质量将取决于成本估算公式的精确性。

那么优化器所估算的成本到底是指什么呢？在简单的场景中（只有同步读，没有并行），成本可能是指耗费的时间（不包括争用），即服务时间。但是有很多理由可以用来解释为什么成本估算的结果可能与实际的服务时间差别很大，即便有完美的过滤因子估值也是如此。也许这就是为什么优化

器的开发人员坚持使用术语“成本”的原因。无论如何，我们真的希望这个成本就是响应时间吗？考虑对于一个 SELECT 的如下两个方案：

- 响应时间为 3s，100 000 个页的异步读，以并行的方式从 6 个磁盘读取，CPU 时间为 2s。
- 响应时间为 5s，1000 个页的同步读，CPU 时间为 0.1s。

对于这两种方案而言成本估算是公平的吗？你希望优化器在一个有 1000 并发用户的操作环境中选择第一个方案吗？

当前主流产品（至少在当前版本中）使用的优化器会同时评估 I/O 和 CPU 的成本。有时可以预见的并行也会被考虑其中，以降低成本估算的结果值，但有时不会。总的成本是这些组成部分的加权和。所以，估算出的成本的确与服务时间相关，尽管这一相关性并不是特别高。

估算 I/O 时间

这可能看起来相当容易（如果忽略并行），我们知道如何做，而且能做得很快。如果优化器对过滤因子做出了好的估算，并且统计值是最新的，那么优化器的确能够准确地预测随机读的数量，以及顺序读取的页的数量。然而，某些优化器的确在评估预读所带来的影响方面有困难，尤其是在使用参数对预读的页的数量进行了限制时。 260

于是优化器就只需要两种类型的 I/O 系数，如每次随机读 10 ms 以及顺序读 40MB/s。其中第一个是对现今所使用的驱动器的一个平均估算，但是顺序读的速度会相差很大。对于很老的磁盘服务器而言顺序读可能为 2 MB/s，对于最新的磁盘服务器而言顺序读可能为 80 MB/s。这里有三种替代方案：

1. 优化器开发人员为每一个数据库管理系统版本决定一个典型的磁盘系统，并相应地选择系数。DB2 for z/OS（仅在 mainframes 上被使用）和 SQL Server 2000（仅在 Windows 上被使用）目前似乎使用的是这种方案。
2. 优化器提供可以用来指定磁盘系统特征的外部参数。DB2 UDB for LUW（在许多不同的平台上被使用）采用的是这种方式。
3. 收集统计信息的工具包会测量 I/O 速度并将结果保存在系统统计信息库中。Oracle 10g 中采用的是这种方式（也在许多不同的平台上被使用）。

这不是一个微不足道的点。如果优化器假设顺序读速度为 4 MB/s，而实际情况为 40 MB/s，那么扫描一个 400MB 的表的 I/O 时间估算为 100 s，

而实际耗时仅为 10 s。假设忽略 CPU 时间，那么最终优化器会将判断选择全表扫描还是非宽非聚簇索引扫描的临界点设定为 FF = 1%（10 000 次随机读），而实际的分界点应该为 FF = 0.1%（1000 次随机读）。

缓冲池及缓存命中率同样也是使估算 I/O 时间变得困难的重要因素。我们的快速估算方法（QUBE）通常是悲观的，因为我们假设每一次随机访问都需要耗时 10 ms。随着内存变得越来越便宜，一次表请求或叶子页请求在数据库缓冲池或读缓存中命中的可能性更大了。对于我们熟悉的应用而言，我们也许能够评估出哪些页会由于访问十分频繁而留在缓冲池中。于是，我们也许可以在 QUBE 公式中加入第三种访问参数类型——*一次低成本的随机访问*，仅耗费 0.1 ms。

然而估算缓冲池或缓存的影响是令人生厌的过程，对于优化器而言也非常困难。当前的优化器可能假设那些确实很小的索引和表是留在缓冲池中的，但是优化器对于缓冲池行为的假设几乎同 QUBE 的假设一样粗糙。实际的算法是保密的而且有时会悄悄地改变，不过最主要的假设似乎是只有根叶是保存在内存里的。这解释了一个很常见的问题，即当合适的索引（通常是一个宽索引）恰好比另一个错误的索引多一个层级时，优化器可能选择这个错误的索引。

261 估算 CPU 时间

这需要对一次 SQL 调用所需的指令数或周期数进行预测，然后再将这一值乘以处理器相关的系数。如果优化器知道随机访问的数量、顺序访问的数量、排序的行数以及结果集的行数，那么我们可以认为优化器能做出一个相当好的 CPU 时间估算。实际上优化器可能还会将许多其他因素考虑在内，如谓词的数量和复杂度，但是有另外一个因素会降低 CPU 时间估算的精确性：内存中的高速缓存。如果一个指令或一行不在这个缓存中，那么它就必须从主内存移入。这一操作需要花费比处理器周期时间长得多的等待时间。当前处理器速度和内存速度之间的差距一直在增大。因此，高速缓存的命中率对实际 CPU 时间有着非常显著的影响。这一命中率取决于 SQL 调用的复杂程度及争用程度。这也就是为什么高峰期的 CPU 时间比独立基准场景下更长。需要注意的是，数据库管理系统监视器测量的 CPU 时间包含*内存等待时间*——虽然我们通常认为测量出的 CPU 时间是服务时间而非 CPU 排队时间（等待一个可用的处理器的时间）。

优化器是如何选择处理器相关系数来进行 CPU 时间估算的呢？DB2 for z/OS 会判断成本估算所使用的 CPU 模型并选择一个合适的值。Oracle 8 压根不估算 CPU 成本，Oracle 9 使用外部参数，而 Oracle 10g 使用上文提到的

系统统计信息库。DB2 for LUW 同样有外部参数。

协助优化器处理估算相关的问题

每次执行时进行优化——将真实值赋予谓词变量

当绑定变量的值已知时，优化器可以被设定为在每次执行时选择访问路径。然而，这对于操作型应用而言 CPU 时间的额外负载太高了——随着成本估算公式变得愈发复杂，这一额外负载正在增多而非减少。因此，仅当最佳访问路径取决于绑定变量实际传值时（如相扑案例中的 SEX = :SEX），才应该选择这一方式。

这一选项的实现方式因产品的不同而不同。

- 在 DB2 for z/OS 中，包含 SELECT … WHERE SEX = :SEX 语句的程序包必须与参数 REOPT(VARS)绑定。于是优化器便会在每次执行时使用绑定变量:SEX 的实际传值重新进行成本估算。
- 为了在 Oracle 中使用两个不同的访问路径，必须在代码中写两个 SELECT 语句，一个为 WHERE SEX = 'M'，另一个为 WHERE SEX = 'F'。应用程序必须负责选择合适的 SELECT 语句。除此之外，INIT.ORA 中的参数必须为 EXACT 或 SIMILAR。 262
- 在一些环境中，可能必须使用动态 SQL，同时禁用 SQL 缓存。

*每次执行时优化*的方式不应与*首次优化*的方式混淆。Oracle 9i 会在 SQL 调用被第一次执行时进行成本估算，并随即将该 SQL 与执行计划保存起来以便复用，这一方式被称为*绑定变量窥视*。DB2 for z/OS V8 通过对动态 SQL 使用 REOPT(FIRST)实现同样的功能。在使用了这一选项的情况下，若语句 SELECT AVG(WEIGHT) FROM SUMO WHERE SEX = :SEX 首次执行时查询的是男性，那么优化器将始终使用全表扫描的访问路径。

如果过滤因子是通过常量或绑定变量值来计算得出的，那么优化器当然需要足够的关于列值分布的信息：一个直方图或首/尾 *N* 个值。否则，优化器怎能知道类似“99.9%相扑选手是男性”的信息？

访问路径提示

当最佳访问路径与输入无关（如不受偏斜分布的影响）时，访问路径提示提供了一种比*每次执行时都进行优化*更好的解决方案。访问路径提示告知优化器所示的访问路径是成本最低的路径。若优化器能够生成这一访问路径，那么它甚至都不会进行成本估算。

有三种类型的提示：

1. 指定型的（指定了完整的访问路径）。
2. 限制型的（将一些选择排除在外）。
3. 信息型的（给优化器提供一些有助于做出更好的成本估算的信息）。

Oracle 现在已经拥有 100 多种不同的提示且被广泛使用，其中部分是由于历史原因。SQL Server 2000 所拥有的提示（连接方式、索引的选择等）数目不多。DB2 for z/OS 提供了一种可以对任何优化器指定所能做到的访问路径的功能，这在下文中有所阐述。DB2 for LUW V8 没有提示功能。

为了帮助理解访问路径提示的涵盖范围，我们将举一些简单的 Oracle 的例子，不过请记住，这仅仅是少数样例：

```
SELECT        /*+ORDER*/ C1, C2, C3
FROM          A, B, C
WHERE         …
```

这指定了表的访问顺序必须是 A,B,C。如果用 LEADING 替换此提示，
263 将指定一个连接中的最外层的表，即驱动表，必须是 FROM 语句中的第一张表。其余表的顺序由优化器决定：

```
/*+FULL(table)*/                  选择全表扫描
/*+INDEX(table index)*/           选择索引扫描
/*+NO_INDEX(table index)*/        不使用指定的索引
/*+INDEX_FFS*/                    选择快速全索引扫描（1）
/*+USE_NL(table)*/                选择嵌套循环连接（2）
/*+USE_HASH(table1 table2)*/      选择哈希连接
/*+USE_MERGE(table1 table2)*/     选择排序合并连接
/*+CARDINALITY(card)*/            结果集的大小（3）
```

（1）快速全索引扫描（FFS）按物理顺序读取所有的叶子页，并允许并行。结果集不保持索引键的原始顺序。

（2）指定的表是嵌套循环内层的表。

（3）可以在提示中加入一个表名来指定一个中间结果集。

在 SQL Server 2000 中，查询提示通常通过使用 OPTION 语句来表示，如下例中，OPTION 语句意味着我们需要查询 20 行：

```
SELECT … FROM … WHERE … ORDER BY … OPTION(FAST 20)
```

或者提示也可以被添加至 JOIN 语句中，比如指定进行哈希连接可以这样：

```
SELECT … FROM table1 INNER HASH JOIN table2 ON …
```

SQL Server 的提示包含以下这些，其中第一个用于指定使用某个索引，或是用 ANDing 指定几个索引，其余的提示用于指定连接方式和表访问顺序：

```
INDEX =
HASH
MERGE
LOOP
FORCE ORDER
```

在 DB2 for z/OS 中，访问路径提示是脱离 SQL 代码之外实现的。优化器所选的访问路径特征被存储在 PLAN_TABLE 表中，该表有许多列。这些列中的大部分都可以被更新，来描述我们希望优化器选择的访问路径。于是 SQL 调用便与指向特定访问路径的参数绑定了。如果该访问路径能够被优化器识别，那么优化器就将选择该访问路径而不进行任何成本估算。于是我们便可以指定另一个索引、另一种连接方式或另一种不同的表访问顺序。然而，若其中某个谓词对于优化器而言太困难了，以至于导致 MC = 1（一个匹配列），那么将 MC 值修改为 2 是没有任何帮助的！ 264

相较 Oracle 与 SQL Server 的实现方式，DB2 的提示使用起来有些烦琐。开发者建议主要将其用于快速回退至前一个（更好的）访问路径。只要旧的访问路径保存在 PLAN_TABLE 表中，这便是容易做到的。另外，DB2 的方式对于改进已生成的程序（如 SAP 和 Peoplesoft）更容易，因为源程序无法修改。当 SQL 调用无法修改时，可以使用带提示的视图方式或者将 SQL 调用连同提示一起保存在系统表中（在 Oracle 中称为*存储概要*）。

冗余谓词

让另一个访问路径看起来成本很高，这样的方式可以影响优化器的决定，这是一种聪明但粗暴的影响优化器的方式。考虑图 14.1 中所示的 WHERE A < :A AND C = :C 的情况，假设优化器由于错误的过滤因子估值而选择了索引 A，可以通过添加一个并不影响查询结果的冗余谓词，如 0 = 1，来使得索引 A 看起来并不合适：

```
WHERE (A < :A OR 0 = 1) AND C = :C
```

冗余谓词会使谓词 A < :A 变为非 BT 的。现在，索引 A 无匹配列了。即便过滤因子估算是错误的，优化器仍将估算出较好的方式（即成本更低的方式）是使用索引 C。

使用这种方式的风险是显而易见的：

1. 也许有一天优化器将智能到能够去除冗余谓词。

2. 把 A 添加至索引 C 上后，将会只有一个匹配列了（参见图 14.3）。
3. 如果我们的后辈发现我们写了这样的代码，他们会做何感想？

伪造统计信息

优化器的统计信息通常被保存在可以用 SQL 进行更新的表中。这样做可以将测试库模拟为生产库，从而有助于提前发现优化器问题。我们也可以通过在生产系统中伪造统计信息来影响优化器。例如，我们可以通过将一张表的行数从 10 000 修改为 100 000 来影响连接的表访问顺序。当然，这比使用冗余谓词的风险更大，因为这样的变更可能会影响到许多 SQL 调用。

这一技巧被多家公司使用过，这些公司的一个共同点是，都出现过优化器选择了错误索引的情况，可能是由于更合适的索引恰好多了一个层级导致的（优化器假定了只有根页是待在缓冲池中的）。其中一家受此问题影响的公司为了使优化器选择正确的索引，在统计信息中修改了某一特定索引的层
265 级信息。这似乎是一个无害的谎言——直至某一天该公司必须做一个复杂的恢复程序时，问题才暴露出来。在此之后，一个关键事务在几个小时内运行得非常慢，原因就是系统表回到了正确的层级。DBA 决定以后再也不伪造统计信息了。

修改索引

通过将一个 SELECT 语句的正确索引修改得更合适，或将 SELECT 语句的错误索引修改得更不合适，都可以用来协助优化器。在本章中我们看到过这两种方式的例子。通常情况下修改索引是一个好的方式。

优化器的问题是否会影响索引设计

对此问题的简短回答是“有所保留的不会”。对于基于成本的优化器而言，拒绝一个真正好的索引是不容易的。因此，当优化器由于错误的过滤因子估算而选择了错误的索引时，最好的方法可能是将合适的索引修改得更好。必须牢记我们在前文关于部分索引键的章节中所讨论的限制，即便是基于成本的优化器，在原有索引非常合适的情况下，也可能为一个 SELECT 语句选择使用一个新建的索引。我们不得不接受一个现实，那就是无论我们采取何种监控异常索引使用的方法，这一风险都是存在的。若只是将列添加至现有索引，这一风险可能会降低很多。

完全避免由索引改进导致性能下降的唯一方法是，对每一个 SQL 调用都使用提示。这么做是极其烦琐的，而且这还违背了将访问路径的选择权交给优化器的理念。另外，这么做也是相当冒险的，因为我们也不是完美的。

同样，这也将否定优化器的任何正面影响。当一个索引被优化以使某个 SQL 调用变得更快时，通常许多其他的 SQL 调用也会变得更快。我们推荐的方式是尽早设计出非常好的索引，并为优化器提供足够多的关于列及列组值分布的信息，至少应提供所有部分索引键的基数值。与直方图和首/末 *N* 列不同，基数值对于等值谓词很有用，即使当估算是基于绑定变量进行时也是如此。可以想一下索引(A,B,C)以及 WHERE A = :A AND B = :B 的情况。

索引改进的同时需要冒着可能引起负面影响的风险，这与优化器开发人员面临的两难选择类似——每当他们改进了成本估算公式，世界上许多 SQL 调用将因此而变得更快，但不幸的是，也有一些 SQL 调用将因此而变得更慢。

练习

14.1. 重写 SQL 14.8 中的游标，使得新游标的访问路径满足：

- MC = 1
- 仅需访问索引
- 无排序

266

SQL 14.8

```
DECLARE CURSOR141 CURSOR FOR
SELECT   LNAME, FNAME, CNO
FROM     CUST
WHERE    (LNAME = :LNAMEPREV
         AND
         CNO > :CNOPREV)
         OR
         (LNAME > :LNAMEPREV
         AND
         LNAME <= :LNAMEMAX)
ORDER BY LNAME, CNO
WE WANT 20 ROWS PLEASE
```

要求：不能去除 ORDER BY。

提示：WHERE 语句可以包含操作符 NOT，不过 NOT 将使该谓词对于优化器而言太过困难（无匹配列）。

14.2. 列出你正在使用的优化器具有的最常见的缺点。

第 15 章

其他评估事项

- 再论 QUBE 背后的假设条件
- 缓冲池、子池及磁盘读缓存中所缓存的索引叶子页和非叶子页
- 响应时间远小于 QUBE 值的场景
- 内存中的叶子页和表页
- 小表、热点及倒推
- 跳跃式顺序扫描
- 计算 CPU 时间
- 默认 CPU 系数
- 测量和注意事项
- 索引设计中所使用的 CPU 评估

QUBE 公式背后的假设条件

简单的 QUBE 公式是基于以下几个主要假设的。

1. CPU 排队时间微不足道——系统有多个处理器，且处理器的负载较为合理。
2. 磁盘利用率（繁忙度）在 15% ~ 35%之间。
3. 其他排队情况也微不足道。
4. CPU 时间是排序操作所花费时间的主要组成部分。
5. 索引的非叶子页缓存在内存中或者磁盘读缓存中，但是索引的叶子页和表页总是从磁盘上被读取的。
6. 处理器和磁盘驱动器的速度为：
   - 每个处理器的速度至少为 200 mips 或 1 GHz。
   - 磁盘驱动器的转速为 10 000 rpm 或 15 000 rpm，平均寻道时间小于 5ms。
   - 磁盘顺序读取的速度为 40 MB/s。

7. 行长不超过 500 字节。 268
8. 非顺序访问都被认为是随机的。

在本书的各个地方其实已经讨论过这些假设条件，现在我们将进一步详细讨论上述用楷体字标注出的部分。

内存中的非叶子索引页

目前为止，我们仅讨论了 B 树索引中的最底层，即叶子页。在此之前，我们假设在当前的硬件条件下，非叶子页的访问时间可以忽略不计，但实际情况是这样吗？下面我们来看一个与典型的宽索引有关的例子。

例子

图 15.1 展示了一个操作型数据库中的宽索引。该表包含一个保险公司的保单索赔信息。该索引究竟有几层，而非叶子页又需要多少空间呢？

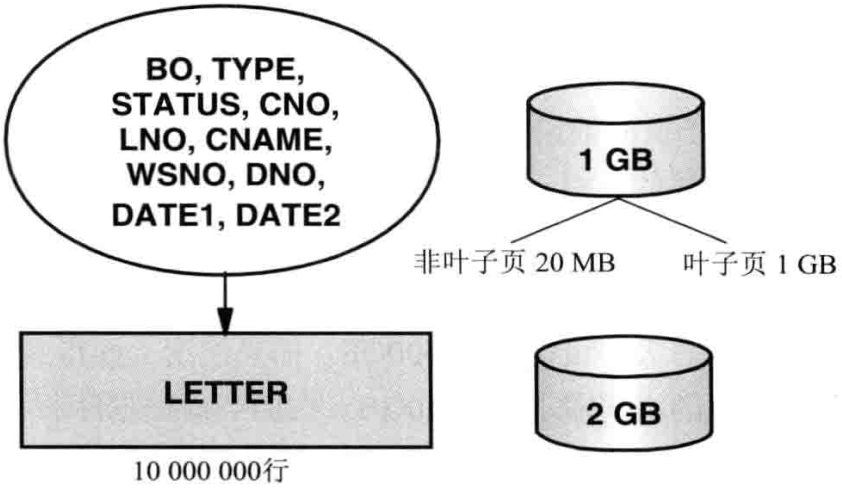


图 15.1　一个宽索引中有许多非叶子页

假设

- 索引包含 10 个列，加上系统开销，总行长为 80 字节。
- 所有的索引页，包括叶子页和非叶子页，在重组后将预留 10%的空闲空间。
- 重组时会回收所有的空叶子页。
- 索引页的大小是 4KB。如果实际情况比这个值大，那么下文中的计算可以重新进行一次，主要是索引的层级可能变小，但非叶子页所使用的空间并不会有很大的不同。
- 数据库管理系统不会对非叶子页中的键值进行截断。

269 **索引页的数量**

索引重组后，叶子页上存储了大约 45 个索引行。因此，叶子页的数量就是 10 000 000/45 = 220 000，第二层包含 220 000/45 ≈ 4900 页，第三层包含 4900/45 ≈ 110 页，第四层包含 110/45 ≈ 3 页，第五层只有 1 页，即根页。

因此，总共大约有 5000 个非叶子页，空间需求约为 20 MB，索引的总页数约为 225 000。因此，磁盘的空间总需求约为 1 GB。

关于索引键截断

如果非叶子页只存储了路由下一层节点页所必需的键值列，而且这些键又是可以截断的，那么非叶子页的数量会大大降低。在我们的例子中，LNO 是主键。索引列中的前 5 列就保证了索引行的唯一性，所以 LNO 后面的索引列就可以不被存储在非叶子页上。因为索引中最长的列 CNAME 在 LNO 的后面，这样非叶子页的数量就可以从 5000 降至 2000。但不幸的是，并非所有的索引键截断都能像本例一样有如此好的效果。

磁盘的空间需求可能并不是一个重大的问题（根据经验，每月的开销约为 50 美元），但是非叶子节点页真的会如我们假设的那样都缓存在内存中吗？在这个索引中，非叶子页的数量是叶子页的 2%（5000/220 000≈2%）。我们假设这是一个有代表性的索引。

那么接下来的问题就是该数据库中所有索引所占空间的总大小。对于一个拥有几百万客户的中等规模的保险公司而言，操作类表的大小可能达到 500 GB，这些表上的索引总大小可能也有 500GB。如果使用上面得到的 2% 的系数，那么非叶子页总的空间需求就是 10GB。按照当前数据库服务器的典型配置（16 GB 或者 64 GB），这一内存需求有些过多了。另外，如果系统不支持 64 位寻址，那么虚拟内存管理会限制数据库缓冲池的大小。

磁盘服务器读缓存的影响

磁盘服务器读缓存的快速增长为我们提供了另外一种解决方案。在撰写本书时，典型的磁盘服务器已经配置了 64GB 大小的读缓存，而且这一半导体的内存并不会增加太多费用。此方案的主要优势在于数据从读缓存到内存的传输时间：在当前的磁盘服务器配置和光纤通道条件下，4KB 大小的页所需的传输时间小于 1ms。尽管这个时间远大于直接从内存中读取数据的时间，但磁盘服务器的读缓存空间远大于数据库的缓冲池，这能够帮助系统将非叶子页的平均访问耗时维持在一个较低的值。

如果数据库缓冲池与磁盘读缓冲的大小只有图 15.2 中所示值的 10%，

而操作型数据库的大小仍保持 1TB，那么检索一个索引行通常需要 2 次磁盘随机读取，即第二层的非叶子页和叶子页的读取，共需 20 ms 时间。这将大幅增加涉及多索引维护的更新事务的响应时间。 270

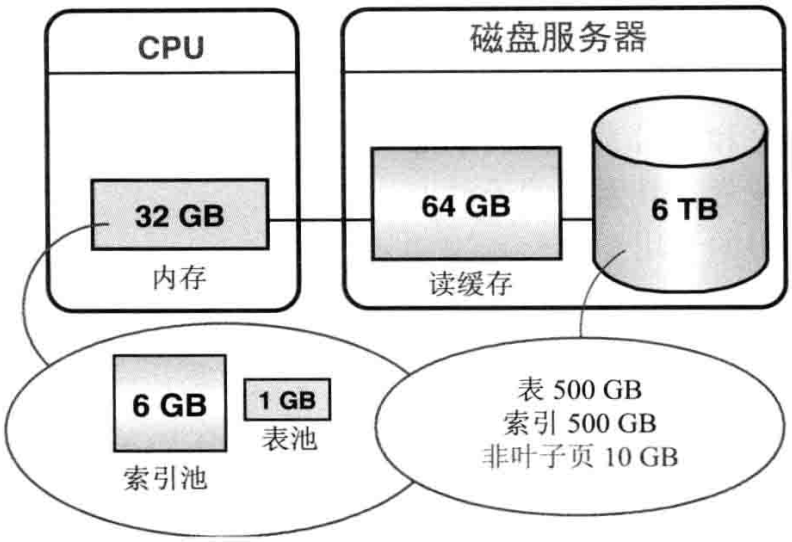


图 15.2　必须以何种频率从磁盘驱动器读取一个非叶子页？

事实上，许多数据库的大小要远小于保险公司的数据库。假如一个数据库中所有表和索引的总大小为 32GB，那么这部分数据就很可能一直缓存在磁盘服务器的读缓存中，或者至少命中率接近 100%。在这类场景中，随机读取的响应时间将从 10ms 降为 1ms。然而，不幸的是，只有当其他共享此 64 GB 读缓存的文件是友好的共享者时，才能实现这一性能提升。

如果一个文件从不占用大量的读缓存，那么该文件就是友好的共享者。磁盘服务器的读缓存管理策略是优先缓存随机读取的页，所以，如果一个文件总是被顺序访问，那么这个文件的数据页将不会被长时间缓存在缓存中。如果针对一个大文件的访问是完全随机的，并且访问的频率不高，那么被读取进缓存的页将留在缓存中，直至它成为最近最少被访问的页。这种页会被保留在读缓存中一段时间，如 10min。如果一个页被随机且频繁地访问——比如不到 10min 就会被随机访问一次——那么该页就会被长时间保存在读缓存中。

对于一个配置了 64GB 读缓存的磁盘服务器，其中大概会有 64 或者 128 块磁盘共享该缓存，总空间达几 TB。该例中所假设的 32GB 大小的数据库会有大量不友好的文件共享读缓存，除非这个服务器是数据库专用的（由于成本原因，这是不太现实的），否则我们所假设的 1ms 传输延时未免过于乐观了。

缓冲子池

许多数据库管理系统将数据库缓冲池分成了 5 个子池：

1. 用于缓冲应用索引的子池
271
2. 用于缓冲应用表的子池
3. 用于缓冲排序和临时表数据的子池
4. 用于缓冲系统表及其索引的子池
5. 内存表及索引的专用子池

使用多个子池有两个原因，一是监控，二是优先级。

监控

如果一个事务需要 500 次磁盘随机读，那么了解应用表和应用索引分别对应多少随机读的信息是有意义的。监控缓冲子池的开销比监控数据对象的开销要低得多。

优先级

让特定的表和索引常驻于专用的数据库缓冲子池中，这在技术和成本层面上都是可行的。在数据库启动后，这类表和索引会被从磁盘服务器中读取并装载到子池中，而且仅有一次这样的操作。通常这样的子池空间可达几 GB 大小。在不久的将来，除了 CPU 时间和分配的磁盘空间外，硬件外包的计费标准还会出现第三个维度：*常驻内存的数据量*。按照目前的价格，1GB 的成本可达 1000 美元每月，不过随着时间的推移，该价格仍在不断地下降。

当前，系统程序和应用程序占用了很大一部分内存（16GB 至 64 GB），留给数据库缓冲池的内存通常只有几个 GB。不过随着时间的推移，64 位寻址技术和更廉价的内存将使得 100 GB 大小的缓冲池成为可能。

在数据库管理系统启动的时刻，各个子池都是空的。从磁盘服务器读取数据并将最大的子池填满，这一过程可能需要耗费半个小时。预热满缓冲池之后，从磁盘读取的新页就必须覆盖缓冲池中的旧页。如果这些旧页被更新过，那么在覆盖之前，这些页应当已经被 DBMS 持久化到磁盘服务器了。缓冲池所使用的覆盖算法主要是*最近最少使用算法*（LRU）。通常顺序访问的页要比随机读取的页在内存中停留的时间短。许多数据库管理系统也开放了管理接口，允许系统管理员在子池级别上调整覆盖算法。

在数据库调优中，核心任务之一就是调整缓冲池的大小。基本的目标就是在系统允许的情况下（不会发生页交换）尽可能地增加缓冲池的大小。将缓冲池的空间分配给 DBMS 的不同实例（比如操作型应用的实例、数据仓库的实例、应用测试的实例及系统测试的实例等），是需要进行复杂权衡的。

通过设置池的大小，可以有效地控制不同实例使用系统资源的优先级。

一些平台（比如 IBM iSeries 和 AS/400 平台）并不支持多个缓冲池的设置，这些平台的操作系统无差别地对待所有的页。缓存的核心算法仍然是 LRU。这样免去了系统调优的任务，但同时也无法设置优先级了。

在没有缓冲池的情况下，磁盘服务器读缓存的表现更像是内存，它决定 272 了哪个页将被新页所覆盖。毫无疑问，缓存的核心算法仍然是 LRU 算法，主要的不同在于读缓存对于顺序读取的页所保留的时间更短。通常情况下，系统管理员无法调整该算法，但有可能可以强制指定某具体的对象保留更短的时间。

如果磁盘服务器包含多个系统的数据，那么读缓存无差别地对待所有数据页可能会成为问题。在一个操作型的数据库中，缓存在服务器读缓存的 4KB 大小的页的数量可能从 1 至 1 百万个不等，具体的缓存数量依赖于其他的系统负载。如果数据库如图 15.2 所示，配置了相对比较大的缓冲池，那么就可以大大减少事务响应时间的抖动。

长记录

在顺序读的情况下，读取一行的 I/O 时间取决于行长。如果顺序读的速度为 40 MB/s，那么读取 4KB 大小的页所需的平均 I/O 时间就是 100 μs。因此，如果每页存储 10 行记录，那么每行记录的平均 I/O 时间就是 10 μs，这就是我们在 QUBE 中所采用的默认值。

当行长超过 400 字节时，尤其是当存在大量的空闲空间时，实际的 I/O 时间“*平均每页的 I/O 时间/平均每页的行数*”可能会远远超过 10 μs。

慢速顺序读

传输时间的系数需要根据高峰时期所测量的读取速度进行校准。如果测量出的速度仅仅是 10 MB/s（在旧的磁盘驱动器条件下，这个值是很有可能的）而不是 40 MB/s，那么该系数将会是 0.4 ms 而非 0.1 ms。

实际的响应时间可能比 QUBE 评估值短得多

QUBE 的主要目的是期望使用*最少的努力*来找出潜在的慢访问路径。因为公式很简单（只使用了两个*必不可少的*变量——随机读次数和顺序读次数），所以在系统设计早期就可以对所有的 SQL 调用进行检查了。

当使用 QUBE 进行索引设计的时候，需要对所建议的索引调整保持谨慎的态度。比如，在一个半宽索引和一个宽索引之间选择（可能需要额外的

10 GB 空间）时，或者在一个宽索引和一个理想索引之间选择时（高峰期间可能有超过 10 000 行记录被写入该表）。知道 QUBE 可能对于成本较低的
273 方案略显悲观是有必要的。

接下来，我们就讨论一下实际响应时间远短于 QUBE 评估值的三个最重要的场景。

叶子页和表页缓存在缓冲池中

在 QUBE 评估方法中，随机读取的响应时间是 10 ms。如果所有的索引页和表页都完全按照随机的方式进行访问，那么这个值是接近真实情况的。如果一个数据库中存储了 100 万个叶子页和表页，页的大小按 4KB 计算，即占用的总空间为 400 GB，且访问频率为每秒 10 000 页，那么每个页的平均访问间隔时间是 100 000 000/10 000 = 10 000 s ≈ 3 h，当然，在实际的应用场景中，其中一些数据页的访问频率要远高于平均水平。

为了简化讨论，这里我们将内存中的数据库缓冲池和磁盘服务器的读缓存统称为缓冲池。如果一个随机访问的页未被再次访问，则该页将在缓冲池中停留 10 min。也就是说，若一个页每过不到 10 min 就被访问一次，那么该页将一直停留在缓冲池中。

叶子页和表页在什么时候能达到这个访问频率？

小表

如果一张表有 100 个同等热度的数据页，且该表每秒被访问一次，那么每个数据页的平均访问间隔时间即为 100 s，如果缓冲池保留数据页的时间是 10 min，那么可以合理地认为，所有的表页和活跃索引的叶子页将持续停留在缓冲池中。

热点

假设某火车座位预定系统保留 90 天的数据，其业务主表按出发日期进行聚簇。由于大部分的人都是在出发当天预订座位的[1]，所以最后 1%的表页和最后 1%的聚簇索引都倾向于缓存在池中。这减少了大量的随机读。

那么热点何时才足够热呢？如果缓冲池的保留周期是 10 min，那么数据页被访问的时间间隔则不应超过 5 min。因此，如果火车座位主表的访问频率是 10 页/s，那么热点的数据就不应该超过 300 s × 10 页/s = 3000 页。

---

[1] 这里所述是美国的情况——编者注

倒推

应用程序可能随机访问某个数据页数次。当访问该页时，这个数据页还缓存在池中吗？如果访问的间隔时间小于缓冲池的保留时间，那么答案就是肯定的，这也是为什么缓冲池的保留时间对于涉及大量随机访问的批量任务而言非常关键的原因。 274

我们再来看一下第 11 章节中所讨论的批量任务。如果表 POLICY 的聚簇索引是如图 15.3 所示的索引 PNO，那么就会有针对 POLICY 表的 5 000 000 次随机访问。假设每次访问都是从磁盘驱动器进行的随机读取，那么 5 000 000 次随机访问将耗费 5 000 000 × 10 ms = 50 000 s，约为 14h。我们假设 POLICY 表的大小是 1 GB。若缓冲池远大于 2 GB，且 POLICY 表中的每个数据页仅从磁盘服务器读取一次，那么一共将耗费 200 000 × 10 ms = 2000 s，大约为 34 min。而余下的随机访问，因为数据页已经缓存在缓冲池中，所以每次访问将只需 0.1ms，总共耗费 4 800 000 × 0.1 ms = 480 s，即 8 min。这样一来，5 000 000 次的随机访问时间就从 14 小时降到了 42 分钟！

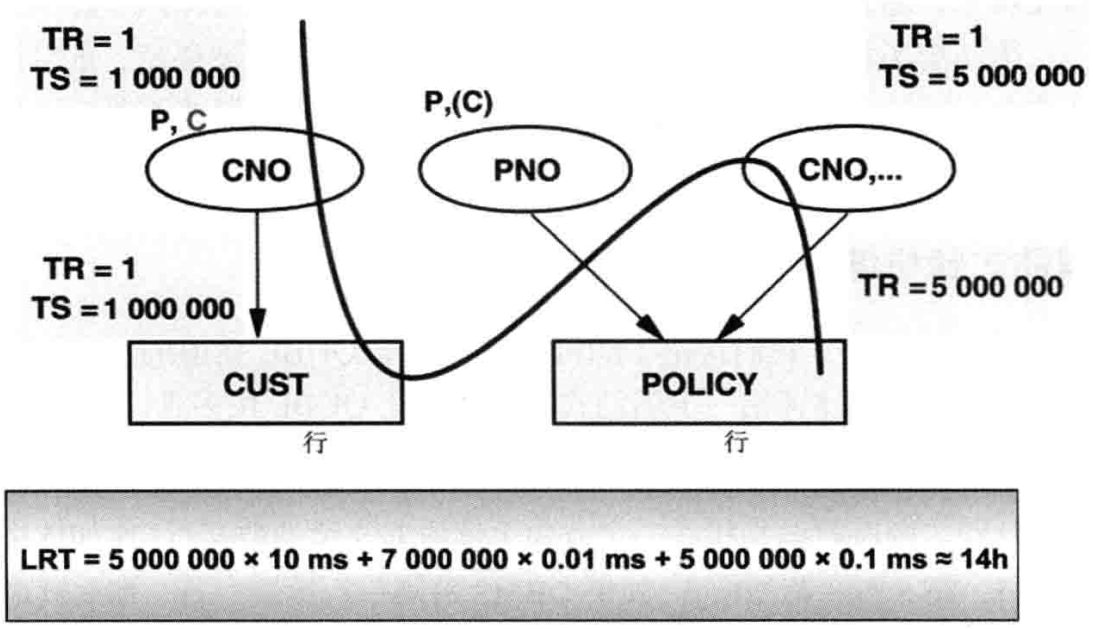


图 15.3　批处理程序——反推

POLICY 表共有 250 000 个数据页，平均每页访问 20 次。如果应用程序的运行时间是 40 min，那么每个数据页被访问的平均间隔时间是 2 min。由此，缓冲池的保留周期必须大于 2 × 2 min = 4 min，才能保证数据页仅仅从磁盘上读取一次。

常驻缓存的表和索引

系统重新启动后，这类表及索引的数据页会从磁盘驱动器上读取一次。如果缓冲池的大小设置得合理，那么这些保存在特殊池中的页就永远不会被

覆盖。只要系统还在运行，这些数据页就会常驻在内存中。

275 识别低成本的随机访问

对于前面两种场景（小表和热点），了解缓冲池的保留时间至关重要。这可以通过应用程序来测量，该程序需要每 *N* 秒钟访问一次只有单个页的某张表。从同步读的次数可以判断出该数据页是否缓存在内存中，而通过同步读的响应时间可以判断数据页是否缓存在磁盘服务器的读缓存中。在不同的工作负载压力下，缓冲池的保留时间也存在很大的不同，尤其是在大表上进行涉及大量随机访问的批量任务时，这一值会变化很大。尽管如此，在特定的负载下，判断缓存保留时间是否在一个合适的范围还是可能的，比如如果该值小于 1 分钟，那么缓冲池的设置就过小了。

对于第三个场景（倒推），应当首先评估索引和表的大小与缓冲池大小的差异：差值是否足够大，大到能够承载其他同一时间的系统负载？如果差值比较大，那么第二个需要考虑的因素就是访问同一个页的平均间隔时间。

第四个场景（常驻缓存的表和索引）并不普遍，但要注意，重启系统后会清空缓冲池，必须意识到这将带来的影响，因为优化器并不善于*评估缓存命中率*。

辅助式随机读取

在第 2 章中，我们讨论了随机读成本低于 QUBE 估值的几类场景。接下来，我们将具体评估一下辅助式随机读会比 QUBE 快多少。

跳跃式顺序扫描

跳跃式顺序扫描是指在一个方向上读取非连续的数据行，比如依次读取第 5 行、第 8 行、第 20 行。如果 DBMS 每隔一行读取一行，那么读取单行的响应时间几乎等同于顺序读的响应时间。而另一方面，如果从有 100 万行记录的表中读取三行相互距离很远的记录，那么读取单行的响应时间将几乎等同于随机读的响应时间。

有两种不同的跳跃式顺序扫描，我们称其为*自然形成的*和*优化器生成的*。对于前者，一个例子为一个被反复执行的 SELECT 语句，其绑定变量按顺序排列但不连续；另外一个例子来自第 8 章，我们将在下文中详细讨论。第二种类型的跳跃式顺序扫描通常发生的场景是，优化器决定首先从索引中读取一些表的记录指针并对其进行排序，然后再访问表的记录。当使用非聚簇索引对单表进行查询时，这一方式可能会出现——如列表预读；或者当在连

接查询中使用混合连接时，也可能使用这一方式。这两种场景都会在 DB2 for z/OS 中出现。

对于跳跃式顺序扫描而言，最为重要的是平均每个数据页中满足条件的记录数。在跳跃式顺序扫描的方式下，只有每页的第一条记录会引起物理读，对页上剩余记录的扫描都等同于顺序扫描。 276

评估跳跃式顺序扫描的 I/O 时间

当跳跃扫描的访问次数（T）大于叶子页或者表页的个数（P）时，可以使用以下的公式来评估扫描的耗时：

$$P \times 10\ \text{ms} + (T - P) \times 0.01\ \text{ms}$$

P 代表被扫描的索引或表片段中的页数。这个公式基于一个假设，即同一时间只会读取一个数据页——10 ms/页。但这可能是一个非常悲观的假设，实际情况可能是，在跳跃扫描了几个数据页后，DBMS 或磁盘系统开始对数据页进行预读。在这种情况下，平均每页的读延时可能低至 0.1ms 或 0.2 ms（具体取决于页的大小为 4KB 还是 8KB）。

让我们将这一公式用于表 CUST 上的主键索引，如图 15.4 所示。当表 INVOICE 的行按 CNO 排序时，将需要 20 000 次跳跃式顺序访问。使用以上公式，这一过程将花费：

$$2500 \times 10\ \text{ms} + (20\,000 - 2500) \times 0.01\ \text{ms} \approx 25\ \text{s}$$

（QUBE：$20\,000 \times 10\ \text{ms} \approx 200\ \text{s}$）

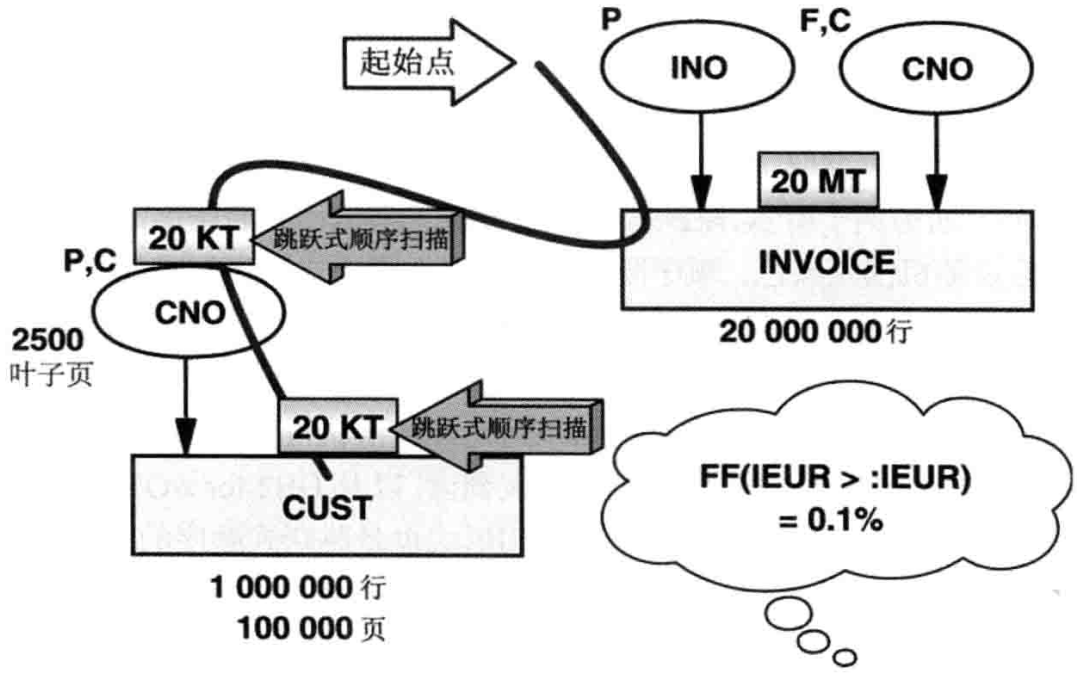


图 15.4　自然形成的跳跃式顺序扫描

如果扫描伴有辅助式随机读取，那么实际的延时可能更低。我们通过在不同的平台上进行连接场景的测试来观察到这一点。比如，在 DB2 for z/OS 平台上进行了一些单页 I/O 后，DBMS 开始一次读取 32 页（动态预读）。
277 这就是为什么对有 20 000 个叶子页的索引进行 100 000 次跳跃扫描访问只需要花费 13s，而使用上面的公式计算出的结果却是：

$$20\,000 \times 10\ \text{ms} + (100\,000 - 20\,000) \times 0.01\ \text{ms} \approx 201\ \text{s}$$

CUST 表上的记录和索引记录都按照 CNO 列的顺序进行组织，所以表记录的访问是跳跃式顺序的，平均每 5 页读取一次，即总共 100 000 页，共 20 000 次访问，也可以看作是每页进行 0.2 次访问。当把表访问考虑进来时，收益就少得多了，而且该收益严重依赖于数据页在磁盘上存储的连续性，以及磁盘服务器缓存数据页的策略。

如果条带大小是 32KB（8 个 4KB 大小的页），且磁盘服务器会将完整的条带一次读取至该缓存中，那么单次操作将平均读入 1.6（即 0.2×8）条满足条件的记录。每个条带的第一次访问耗时 10 ms，而其余的访问耗时则是 1 ms，单次访问的平均耗时为：

$$\frac{1 \times 10\ \text{ms} + 0.6 \times 1\ \text{ms}}{1.6} \approx 7\ \text{ms}$$

于是，20 000 次的跳跃式顺序访问的耗时即为：

$$20\,000 \times 7\ \text{ms} = 140\ \text{s}$$

而 QUBE 方法所计算出的耗时是 $20\,000 \times 10\ \text{ms} = 200\ \text{s}$。

在这一案例中，如果磁盘服务器提前预读多个条带，则跳跃式顺序扫描的收益会更大（类似于在顺序扫描中所做的处理）。

如上面的例子所示，跳跃式顺序读覆盖了随机读和顺序读之间的整个范围。假设随机读是黑色，顺序读是白色，那么跳跃式顺序扫描就是整个灰色渐变区域。这在评估诸如嵌套循环连接等查询时特别有帮助。在 QUBE 方法论中，1000 次的非连续访问会按照随机读进行计算，但如果这些访问是跳跃式顺序的，那么实际的运行结果将比评估值快得多。

以上所讨论的原则同样适用于列表预读（以及 DB2 for z/OS 中的混合连接），因为在这些方法中，表行的访问形式也是跳跃式顺序的。

数据块预读

如第 2 章中所述，当 Oracle 数据库扫描索引片段时，会一次发起多个同步读操作。如果数据已经被条带化（如 Oracle 所推荐的那样），那么这

多个读操作就会并行处理，n 次随机访问的耗时将小于 10 ms。假设数据库表被条带到三个磁盘驱动器上，对 n 次随机访问的耗时最乐观的估计是(n × 10 ms)/3，而实际的值将会介于该值与 QUBE 估值之间。

评论

从上面的讨论可以看出，通过辅助式随机读的方法可以有效地降低 n 次随机访问的响应时间。不过，尽管如此，我们仍然建议尽可能地避免随机访问，就像我们在全书中不断强调的那样。否则，由此导致的磁盘负载将一直很高。

278

辅助式顺序读

除了条带带来的并发度，还可以使用多个处理器来加速顺序读。但不同平台的实现方式不同，比如 DB2 for z/OS 使用的是共享内存和存储的方案，Oracle 集群使用的是处理器专用内存和存储的方式。如果单个处理器的运行耗时是 T，那么 n 个处理器并行处理单次顺序读的耗时则可能接近于 T/n。

评估 CPU 时间（CQUBE）

我们可以简单地借鉴 QUBE 的方法来评估 SQL 执行的 CPU 时间（CQUBE），用 RS 代表排序的记录数：

$$\text{CPU 时间} = 100\ \mu s \times TR + 5\ \mu s \times TS + 100\ \mu s \times F + 10\ \mu s \times RS$$

我们将该公式中的数值称为默认 CPU 系数。

接下来我们将逐个讨论如何测得这些默认 CPU 系数，并着重展示这些数值在各种因素的作用下所产生的巨大变化，比如平台、处理器速度、缓存、DBMS、缓冲池、SQL 语句的特征、列设置、页大小及锁策略，等等。

单次顺序访问的 CPU 时间

按照 QUBE 方法中的定义，一次顺序访问是指在顺序扫描过程中读取一条记录，然后使用非匹配谓词进行过滤（匹配谓词已被用于定义所扫描片段的厚度）。当然，如果是全表或者全索引扫描，那么所有的谓词都是非匹配谓词。

单次顺序访问的 CPU 时间主要跟单条记录的长度有关，因为 CPU 时间主要耗费在数据页的处理上。另外，非匹配谓词的个数和复杂度是另一个主要因素，还有锁机制和压缩等其他因素。

在具体的平台上，评估单次顺序访问的 CPU 时间相对简单，接下来我们就展示三个具体的测量的例子。

在第一个例子中有两个 WHERE 条件恒为非真的单表 SELECT 语句。两个语句的 WHERE 条件中都只有一个谓词，且谓词为非索引列。这样，优
279 化器不得不为这两个 SELECT 语句选择使用全表扫描的方式。其中的 CUST 表有 111 000 个 4KB 大小的数据页，共 1 000 000 条长度为 400 字节的记录。另外的 INVOICE 表有 77 000 个数据页，共 4 000 000 条长度为 80 字节的记录。

表没有进行压缩，所有的列都是固定长度，两个 WHERE 条件也足够简单。锁的粒度是数据页级别的，隔离级别是 CS。一些额外的谓词，特别是复杂的谓词会增加 CPU 时间，另外，数据记录长度的增加、记录变长、使用压缩也同样会增加 CPU 时间。

使用一台旧的繁忙的服务器（单处理器 100mips）所测得的运行时间如下：

全表扫描的 SQL 的 CPU 时间

CUST（1 000 000 行，111 000 个 4KB 大小的页）2.5 s，2.5 μs 每行

INVOICE（4 000 000 行，77 000 个 4KB 大小的页）5 s，1.25 μs 每行

假设记录数和页数是唯一的两个重要因素，那么我们可以计算出下面两个变量的系数：

$$1\,000\,000X + 111\,000Y = 2.5\text{ s}$$
$$4\,000\,000X + 77\,000Y = 5\text{ s}$$

X=每行的 CPU 时间= 1 μs

Y= 每页的 CPU 时间= 13 μs

X 和 Y 的值清楚地表明了列的长度和个数对 CPU 时间的影响。

在第二个例子中采用了一个大型的繁忙的服务器（单处理器 400mips）；扫描一张有 222 000 个 4KB 大小的页、共 13 000 000 行短记录的表，结果集同样是 0。该表经过了压缩，而且记录是变长的，但条件谓词依然是简单谓词。

最终测得的 CPU 时间是 9 s，平均每行 0.7 μs。

在第三个例子中使用了一个小型的服务器（单处理器 750MHz），扫描的 CUST 表有 1 000 000 行 400 字节的长记录，结果集共提取 1078 行记录，并进行排序操作。在 WHERE 条件中，有两个简单的谓词。最终测得的 CPU 时间是 10.4 s。

TS 的 CPU 时间一共是 10.4 s – X，X 代表 1078 行记录的 FETCH 和排序时间，利用默认的 CPU 系数进行计算：

$$X = 1078 \times 110\ \mu s \approx 0.1\ s$$

于是，长记录的单次顺序访问的 CPU 时间就变成了 10 μs。

看起来 5 μs 可以作为一个合理的系数值来使用，但是最好能在具体的平台上，使用长记录和短记录实测一下 CPU 时间。

单次随机访问的 CPU 时间

有以下几个原因会导致随机访问的CPU时间高于顺序访问的CPU时间。

1. 几乎每次随机读取都会引起一次数据页请求。 280
2. 如果数据页没有在缓冲池中，I/O 相关的 CPU 时间会比较多。而由于顺序读取每次会读取多个数据页，所以平均每个数据页的 I/O 代价就相对比较低。另外，平均每页的 CPU 成本能够平摊至许多顺序访问上。
3. 由于随机访问是不可预测的，所以 CPU 无法将数据页预读到高速 CPU 缓存中，从而可能会导致明显的内存等待。
4. 按照 QUBE 的定义，一次对索引的随机访问忽略了访问非叶子页的成本，因为 QUBE 假定它们是位于缓冲池中的。然而，从计算 CPU 时间的角度上看，随机访问一个三层索引上的记录将引起三次数据页请求。这也是为什么索引上的随机访问会比表上的随机访问耗时更多。

计算单次随机访问的 CPU 时间的一个简单方法是，对比使用半宽或宽索引（以避免不必要的随机读取）前后的 CPU 时间差异。

另外一个方法是，消除掉低成本的操作（比如顺序访问和排序）。我们通过一个实例来描述这种方法：一个嵌套循环连接的 SELECT 查询语句，有 813 次对三层索引的随机访问，在一台 100mips 的机器上运行，各项指标如下：

```
        TR =  814  （中等长度的索引行）
磁盘读 =  845  （磁盘随机读取）
        TS =  8130  （较短长度的索引行）
         F =  814
        RS =  0
CPU 时间 =  401 ms
```

单次随机访问的CPU时间可以通过实测的TS系数及F系数计算得出：

$$\frac{401\ \text{ms} - [(8130 \times 1\ \mu\text{s}) + (814 \times 50\ \mu\text{s})]}{814} \approx 434\ \mu\text{s}$$

$$\text{转化成 250mips 的机器} \quad \frac{434\ \mu\text{s}}{2.5} \approx 173\ \mu\text{s}$$

这个CPU时间是在一个缓存命中率为0的测试环境得出的（数据库缓冲池和磁盘缓冲区都没有命中，即845次随机读取都来自磁盘）。在第一次测量完成后不久再次执行同样的事务时，消耗的CPU时间是165 ms。在这个例子中，假设其他部分的CPU时间开销是48ms，那么单次随机访问的CPU时间就是：

$$\frac{165\ \text{ms} - 48\ \text{ms}}{814} \approx 144\ \mu\text{s}$$

$$\text{转化成 250mips 的机器} \quad \frac{144\ \mu\text{s}}{2.5} \approx 58\ \mu\text{s}$$

281 在第一个例子中，平均一次随机读取的磁盘物理读个数是845/814≈1.04，而在第二个例子中完全没有磁盘I/O。这一结果验证了一条重要的原则：减少磁盘读缓存或者物理磁盘驱动器上的随机读取可以大大地节省CPU时间。同时也解释了影响单次随机读取的CPU时间的一个重要因素。

根据我们的经验，在当前的硬件条件下，单次随机访问的CPU时间通常在以下这一范围区间里：

$$\text{表访问} \quad : 10 \sim 100\ \mu\text{s}$$
$$\text{索引访问} : 20 \sim 300\ \mu\text{s}$$

影响随机访问CPU时间的最大因素还是CPU的高速缓存，如果大量地随机访问内存中的少量数据页，那么单次随机访问的CPU时间可能会小于10 μs，实际上可能接近于顺序访问的耗时（几微秒）。

结论

单次随机访问的CPU时间100 μs仍然可以用来进行快速评估。但是，按照之前的讨论，CPU时间会受到许多因素的影响，所以，当做出重要决定时（例如基于CPU时间来评估表的反范式化，或添加一个新索引等方案时），需要使用一个合理的时间范围进行评估。比如，对于一个大索引，其随机访问的CPU时间可以假设在100～300 μs之间。

然而，在评估CPU时间的潜在收益时，这可能会导致问题。比如，如

果一个涉及 10 000 次随机访问的 SQL 语句消耗了 600 ms 的 CPU 时间，但是消除 9000 次的随机访问并不会减少 900 ms 的 CPU 时间。我们需要评估 600 ms 中有多少消耗在 TR 上。

单次 FETCH 调用的 CPU 时间

单次 FETCH 调用的 CPU 时间消耗（除去访问的时间）依赖于平台以及应用程序与 DBMS 的连接方式。在一个具有多层结构和拥有事务管理器的应用设计下，发送 SQL 语句到数据库管理系统（以及将结果集传输回来）的 CPU 时间大概是 50 μs ~ 100 μs。在其他不同的环境下，尤其是当 SQL 语句嵌入在数据库服务器上的批处理程序中时，单次 FETCH 操作的 CPU 时间大概在 10 μs ~ 20 μs 之间。 282

图 7.9 显示了一个具有高效的访问路径的事务，该事务发起了约 100 万次 SQL 请求，其中大部分的请求都只涉及一次顺序访问。将事务管理器（CICS）消耗的 CPU 时间考虑进来，CPU 总时间是 100 s，平均每次 FETCH 操作耗时约 100 μs。

由批处理程序发起的 FETCH 访问的代价会相对比较低。在一些场景下，100 μs 这一默认系数可能是非常悲观的，如下述场景所测得的那样。

例如，在之前的例子中，在一台大型的繁忙的机器上（单核 440 mips）借助全表扫描，结果集是空，得到的 TS 的 CPU 时间是 0.7 μs。然后又执行了一次该 SELECT 语句，只是这次所有记录都满足条件。该查询涉及 222 000 个 4KB 大小的页，共 13 000 000 条记录，且 SELECT 列表只包含一个列。

测得的总 CPU 时间是 115 s，单次 FETCH 操作平均耗时 8.8 μs（115 s/13 000 000≈8.8 μs）。因此，单次 FETCH 操作的 CPU 时间就是 8 μs（8.8 μs − 0.7 μs）。这是一个下限值，如果查询的列比较多，那么 CPU 时间会相对更长。对于中型的服务器（单核 250mips），只查询一个列的 SELECT 语句的单次 FETCH 调用的 CPU 时间下限约为(440 mips/250 mips) × 8 μs≈14 μs。

如果读者希望在自己的平台上进行这些测量，应当注意在 FETCH 相关的 CPU 时间中，有一部分会出现在事务的监控报表中，而不出现在 DBMS 的监控报表中。

有些数据库系统支持多行 FETCH：

```
FETCH NEXT ROWSET FROM cursor_x FOR 10 ROWS
  INTO :hv1, :hv2, ... :hv10
```

如果只查询少量的列，那么使用这一方式可以使得应用程序接口（API）调用的 CPU 时间降低 50%以上。于是，上述获取 10 行记录的 FETCH 操作

的 CPU 时间可以估算为 10 × 0.1 ms/2 = 1 ms/2。

每排序一行的平均 CPU 时间

在内存充足且没有引起磁盘 I/O 的情况下，排序操作的 CPU 时间基本和记录数呈线性关系。平台相关的系数很容易测试：首先执行一条对至少 10 000 行记录排序的 SELECT 语句，一段时间之后，再执行同样一条没有 ORDER BY 的 SQL 语句。平均排序一行的 CPU 时间经验值是 10 μs，所以这两条语句的 CPU 时间差异的期望值为 100 ms。

CPU 评估举例

在许多决策支持过程中，具备 CPU 需求的评估能力是有用的。在本书中，我们对比过单纯基于响应时间的多种场景下的差异。其中一个例子就是优化器选择使用扫描表而非扫描索引的场景。根据 QUBE 的参数，即单次 TR 10 ms、单次 TS 0.1 ms，优化器做出选择的拐点为过滤因子等于 0.1%。本章节开头讨论的问题，比如倒推和跳跃式顺序扫描，降低了这一拐点的确
283 定性。这其中所涉及的 CPU 时间也是一个重要的考虑因素。下述几个对比的方面是我们之前所讨论内容中较为重要的一些，下面我们就来评估一下它们与 CPU 相关的问题。

宽索引还是理想索引

在第 6 章的例子中，单纯从响应时间来看，理想的索引并非具有完全的优势，我们给出了如下结论：

> 虽然三星索引有一定的优势，尤其是在结果集为空的情况下，但是这个优势并不特别明显，而且会带来与新增索引相关的一些额外开销。

宽索引和最佳索引的 QUBE 如下。

```
宽索引：结果集为空

索引 LNAME, FNAME, CITY, CNO      TR = 1          TS = 10 000
提取 0×0.1 ms
LRT                               TR = 1          TS = 10 000
                                  1×10 ms         10 000×0.01 ms
                                  10ms+100ms+0ms=110ms

最佳索引：20 行结果记录 (1 屏)

索引 LNAME, CITY, FNAME, CNO      TR = 1          TS = 20
```

```
提取 20×0.1 ms
LRT                                TR = 1          TS = 20
                                   1×10 ms         20×0.01 ms
                                   10ms+0.2ms+2ms≈12ms
```

现在来评估一下宽索引和理想索引的 CQUBE 值：

$$\text{宽索引}\ 100\,\mu s \times 1 + 5\,\mu s \times 10\,000 + 100\,\mu s \times 0 + 10\,\mu s \times 0 \approx 50\,ms$$
$$\text{最佳索引}\ 100\,\mu s \times 1 + 5\,\mu s \times 20 + 100\,\mu s \times 20 + 10\,\mu s \times 0 \approx 2\,ms$$

从 CPU 的角度我们可以清楚地看到，使用我们之前得出的 CPU 系数，理想索引比宽索引的收益要高 25 倍。图 15.5 对比了例子中所有索引的 LRT 和 CPU 指标。

①半宽索引　②宽索引　③最佳索引 ★★★

	索引	QUBE（最差输入条件下）	
		LRT	CPU
	LNAME, FNAME	100 s	1s
①	LNAME, FNAME, CITY	0.2 s	54 ms
②	LNAME, FNAME, CITY, CNO	0.1 s	50 ms
③	LNAME, CITY, FNAME, CNO	0.01 s	2 ms

图 15.5　通过使用三星索引来节省 CPU 时间

增加一个索引，对于 CUST 表的插入和删除只会增加几毫秒的 CPU 时间（一次随机访问以及写相关的 CPU 时间）。因此，如 15.5 所示，在 SELECT 语句执行得比较频繁的情况下，使用理想索引（图 15.5 中的方案③）将会带来相当可观的 CPU 收益。

嵌套循环（及反范式化）还是 MS/HJ

在第 8 章的连接查询的例子中，最终有两种可选方案：

1. 使用嵌套循环式的 BJQ 连接（使用反范式化）。
2. 使用理想索引进行合并扫描（MS）或哈希连接（无反范式化）。

第二种方案的性能表现很好：在最差输入条件的情况下，即返回 1000 行结果集的情况下，耗时为 1.8s。但这个方案的唯一问题就是 CPU 时间。毕竟，扫描两个索引片，需要 120 000 次随机访问以及对 20 000+100 000 行

284

记录的合并排序或哈希连接，如图 15.6 所示。所以，在确定选择第二个方案之前，还需要慎重评估一下 CPU 时间。

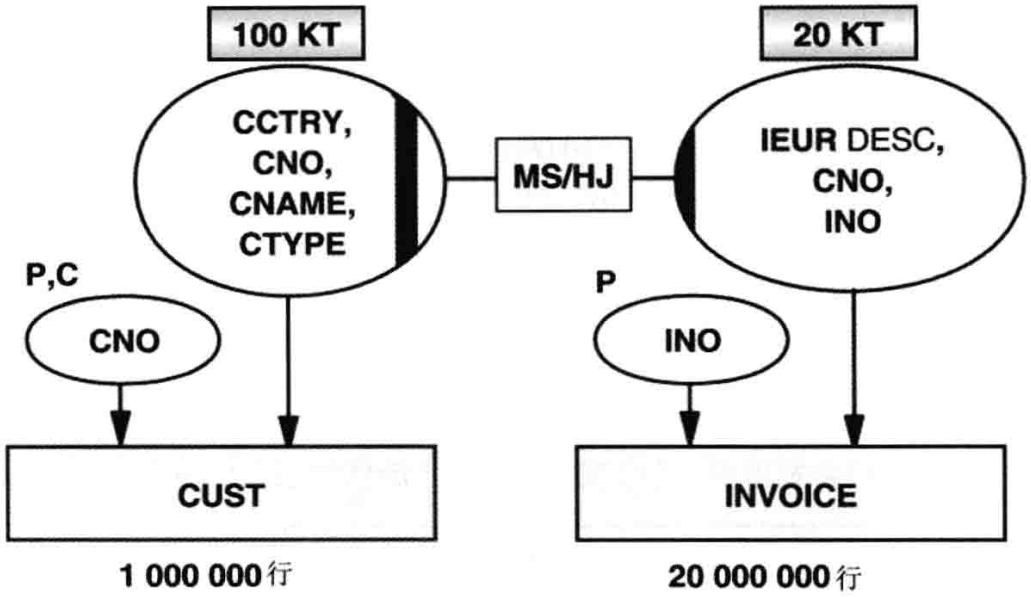


图 15.6　MS/HJ 的 CPU 时间

与之形成对比的是，用第一种方案获取一屏结果集的 CPU 时间非常短，且无排序：

$$TR = 1 + 20 = 21,\ TS = 20 + 0 = 20,\ F = 20$$

所以，CPU 时间= $21 \times 100\ \mu s + 20 \times 5\ \mu s + 20 \times 100\ \mu s \approx 4$ ms（每屏结果集）。

285 **方案 2（使用合并扫描）**

第 1 步： 访问索引 (CCTRY,CNO,CNAME,CTYPE)

```
TR = 1 TS = 100 000
CPU 时间 = 100 μs x 1 + 5 μs x 100 000 ≈ 500 ms
```

第 2 步： 访问索引 (IEUR DESC,CNO,INO)

```
TR = 1 TS = 20 000
CPU 时间 = 100 μs x 1 + 5 μs x 20 000 ≈ 100 ms
```

第 3 步： 合并排序并 FETCH 结果行

将 CUST 作为外层扫描使得结果行已按照所需的顺序排列。因此，最终的 CPU 时间如下。

$$\text{排序及合并的 CPU 时间为：}\ 2 \times 20\,000 \times 0.01\ \text{ms} = 0.4\ \text{s}$$

$$\text{提取的 CPU 时间为：}\ 2000 \times 0.1\ \text{ms} = 0.2\ \text{s}$$

CPU 时间　在最差输入下，这一访问路径下的 CPU 时间是上述三部分的总和：

$$0.5\ s + 0.1\ s + 0.6\ s = 1.2\ s$$

方案 2（使用哈希连接）

这种方式下，第 1 步和第 2 步与合并扫描的前两步一样——CPU 时间 = 0.5 s + 0.1 s =0.6 s

第 3 步：使用哈希函数进行匹配并 FETCH 结果集

优化器优先选择为来自 INVOICE 表的 20 000 行短记录建立哈希表，表大小可能为 20 000 × 30 字节 = 0.6 MB。在生产环境中，该哈希表可以一次性全部缓存在内存当中，但不太可能全部缓存在只有 1MB 大小的 CPU 缓存中。

建立好哈希表后，开始扫描 CUST 表的索引片段。扫描的匹配过程需要随机访问哈希表 100 000 次，单次访问的 CPU 时间依赖于 CPU 缓存的命中率。若给定的是一个 1MB 大小的 CPU 缓存，那么我们假设单次哈希表访问的 CPU 时间是 1 μs ~ 50 μs，于是，100 000 次哈希表访问的 CPU 时间就是

$$100\ 000 \times (1 \ldots 50\ \mu s) = 0.1\ s, \ldots, 5\ s$$

最后，提取结果集，提取操作的 CPU 时间为：

$$2000 \times 0.1\ ms = 0.2\ s$$

如果哈希连接的成本比较高，由于合并排序连接的 CPU 时间估值为 0.4s，那么优化器很可能会选择使用合并扫描。在必要的情况下，可以使用相应的 MERGE 提示来指定该访问路径。

CPU 时间　在最差输入条件下，使用这一访问路径的 CPU 时间是上述三部分的总和，至少为：

286

$$0.5\ s + 0.1\ s + 0.3\ s = 0.9\ s$$

结论

第一种方案需要大约 4ms 的 CPU 时间来提取一屏结果集，第二种方案需要 1 s 的 CPU 时间来提取整个结果集（1000 条记录，50 屏）。对于比较小的结果集，合并扫描的 CPU 时间是 0.6 s（扫描，排序，合并 20 000 条 INVOICE 索引记录），而哈希连接的 CPU 时间则是 0.2 s（扫描 20 000 条 INVOICE 索引记录）。

所以，如果 SELECT 语句执行得不是非常频繁，第二种方案很可能是

可接受的。在极端情况下，可以创建两个游标，一个使用合并连接（针对大结果集），一个使用哈希连接（针对小结果集），从而用最小的代价避免表的反范式化改造。

合并扫描与哈希连接的比较

我们在第 8 章已经对它们做过对比，但现在有必要从 CPU 的角度再回顾一下。

之前，我们曾在一个实验室的平台上（单核 933MHZ，没有并发负载）测量 CUST 表和 INVOICE 表的哈希连接：通过索引（FF CUST 0.5%）随机读取 5000 行 CUST 表记录，在内存中建立哈希表，然后使用全表扫描的方式读取 4 000 000 行 INVOICE 表记录。最终测得的 CPU 时间是 4.5 s。因此，顺序访问单行记录的平均 CPU 时间少于 1.1 μs。

在这个例子中，CPU 时间主要由三部分组成：

4 000 000 次对 INVOICE 表的顺序访问

5000 次对 CUST 表的随机访问

8000 次对哈希表的随机访问（INVOICE 表 FF = 0.2%，0.2% × 4 000 000 行= 8000 行）

对于每一条满足条件的 INVOICE 记录，数据库必须使用 CNO 列的哈希值来访问哈希表（连接谓词是 CUST.CNO = INVOICE.CNO）。哈希表的大小是 5000 × 100 字节 = 0.5 MB。由于测试是在一个独立的环境中进行的，所以该哈希表可以在 INVOCIE 表被扫描的过程中一直缓存在 CPU 高速缓存中。在这种情况下，哈希表的单次随机访问 CPU 时间将低于 100 μs。若 CUST 表单次随机访问的 CPU 时间是 100 μs，INVOICE 表单次顺序访问的 CPU 时间是 1μs，那么总的 CPU 时间则为：

4 000 000 次对 INVOICE 表的顺序访问	4 000 000 × 1 μs = 4 s
5000 次对 CUST 表的随机访问	5000 × 100 μs = 0.5 s
8000 次对哈希表的随机访问	8000 × Xμs = 8X ms
40 次提取操作	40 × 100 μs =4 ms

287 因为哈希表的操作（建立哈希表并对其进行 8000 次随机访问）所花费的 CPU 时间，远远小于 400 万次顺序扫描所花费的 CPU 时间，所以在测量的过程中，我们无法准确地评估出哈希表操作的 CPU 时间。但基于前面对于 CPU 缓存的讨论，我们可以推测出单次哈希表随机访问的时间是低至几微秒的。

若使用合并扫描，那么取代 8000 次哈希表随机访问的将是 13 000 行记录的排序操作（CPU 时间估值是 13 000 × 10 μs ≈ 0.1 s），以及合并扫描 8000 行和 5000 行记录的操作（CPU 时间估值是 0.1 s）。如果哈希表的单次随机访问 CPU 时间小于 25 μs（$X < 25$），那么在这个例子中，合并扫描的 CPU 时间就会高于哈希连接的 CPU 时间。

如果哈希表非常大，而且 CPU 高速缓存被许多应用程序共享，那么情况就会变得不同。我们假设 CUST 表和 INVOCIE 表上本地谓词都有 10%的过滤因子，那么哈希表的大小就变成了 100 000 × 100 字节 = 10 MB。对于一个 1MB 的共享的 CPU 缓存来说，哈希表访问的 CPU 缓存命中率就变得非常低。在高峰期间，随机访问的 CPU 时间可能会高达 50 μs（类似于一次不涉及 I/O 的随机表访问）。那么 400 000 次对哈希表的随机访问将花费 400 000 × 50 μs = 20 s。此时，合并连接是更好的选择，因为它一共需要对 100 000 + 400 000 行进行排序和合并扫描，CPU 时间共 2 × 500 000 × 10 μs = 10 s。如果哈希表更大一些，以至于优化器决定不将其全部缓存在内存中，而是将其拆分成多个分区，比如若拆分成 10 个分区，那么和前面所述的情况差别就更大了，匹配 INVOCIE 行记录的时候需要访问哈希表 10 次。

优化器必须要在哈希连接和合并扫描之间做出选择，但 CPU 高速缓存增加了优化器预测哈希连接 CPU 时间的难度。

这也是为什么数据库系统提供了提示功能，允许用户自由选择连接方式的原因。但按第 8 章提到的，从索引设计的角度来看，HJ 和 MS 唯一的不同就是：MS 可以有效地使用索引来避免排序操作。

另一方面，在 NLJ 和 MS/HJ 设计出的索引之间进行选择是另一种不同的考量，尤其是对于非 BJQ 查询而言，MS/HJ 的连接方式通常都有更低的响应时间，但却有更高的 CPU 时间。

跳跃式顺序扫描

我们之前建议过，评估单次随机访问的 CPU 时间的简单方法是，比较一下在使用宽索引或者半宽索引消除了大量随机访问前后的 CPU 差异情况。

在本章前文中，我们已经讨论了跳跃式顺序扫描的情况，并做了以下描述：

> 如果随机读取是黑色，顺序读是白色，那么跳跃式顺序扫描就是它们之间整个灰色渐变区域。

下面的测试就是一个“灰色”的例子，在一个处理器为 100mips 的平台 288

上，在非半宽索引条件下，一条 SELECT 语句的执行情况为：

$$\begin{aligned} TR &= 10\,000 \\ TS &= 10\,000 \\ F &= 1000 \\ RS &= 1000 \\ \text{CPU 时间} &= 275\ \text{ms} \end{aligned}$$

若使用半宽索引，由于大量的随机访问被转换为了跳跃式顺序访问，所以 TR 减少至 1000，不过 TS、F 和 RS 仍保持不变。总的 CPU 时间减到 124 ms，由此计算出单次跳跃式扫描访问的 CPU 时间是(275 −124)ms/9000 ≈ 17 μs。

CPU 时间仍然不可忽视

减少访问的次数——指所有类型的访问，不仅仅是成本较高的随机访问——可以大大地减少 CPU 时间。由于通常情况下，随机访问的时间是响应时间的最主要部分，所以减少 CPU 时间在大部分的平台上仍然很重要。大型机的 CPU 成本高达 1000 美元每小时——因为许多软件授权会根据 CPU 的处理能力来收费。虽然在 Linux、UNIX 和 Windows 平台上，CPU 的成本会稍微低一些，但为了避免达到单机 CPU 处理能力的瓶颈，节省 CPU 时间仍然很关键。虽然增加服务器可以有效地分担负载，但与此同时这也同样会带来 CPU 和存储的额外负载，而且还会增加锁等待的风险。

第 16 章

组织索引设计过程

- 从相关人员的职责角度看如何组织索引设计过程
- 使用计算机辅助式的索引设计工具
- 它们的目标及优缺点
- 对于基础评估过程的总结
- 设计出色索引的 9 个步骤

简介

在数据库发展的早期，索引设计在专家组中被认为是一种核心职责。当时的应用非常简单，只需要一位数据库专家就能够熟悉所有的处理需求。而在今天，这几乎不可能了。

现今，每位应用程序开发者在编写一个新的 SQL 语句时都会对现有的索引进行评估，至少会用基础问题来进行评估，理想情况下，他们还应使用 QUBE 来检查最差输入下的性能。若性能不可接受，那么他们将考虑进行必要的改进。如有必要，可将该语句连同建议（譬如向一个索引上加一个列，或创建一个新索引）一起向数据库专家征求意见。最终的决策可能是集中的，至少在公司层面被综合应用。专家是评估并控制整体成本及副作用的最佳人选。

当然，将处理索引相关问题的职责委派给开发人员才能取得最好的效果。然而，许多公司认为所需的培训和指导（约 1 至 2 周）是一笔过于庞大的开销。

在大公司里，一个合理的折中方案是聘用 50/50 的专家，他们会将 50% 的时间用于应用程序开发，另外 50%的时间用于协助同事进行索引评估及其他性能问题的处理（比如根据 EXPLAIN 的输出内容解决某个优化器问题）。 290
经验显示，为每 5 至 10 位应用开发者配一名 50/50 专家的方式效果很好。

索引设计需要同时掌握技术技能以及应用系统知识。相比让数据库专家熟悉应用系统的细节而言，教会开发人员索引技能是更容易的。

计算机辅助式索引设计

令人意外的是，索引设计工具直至 20 世纪 90 年代才在市面上出现。第一批工具是基于规则的，而许多当前的工具则是使用优化器来评估各个可供选择的索引。这些工具的目的是找出能将系统在一个特定负载下运行的平均响应时间降至最低的索引集（见图 16.1）。

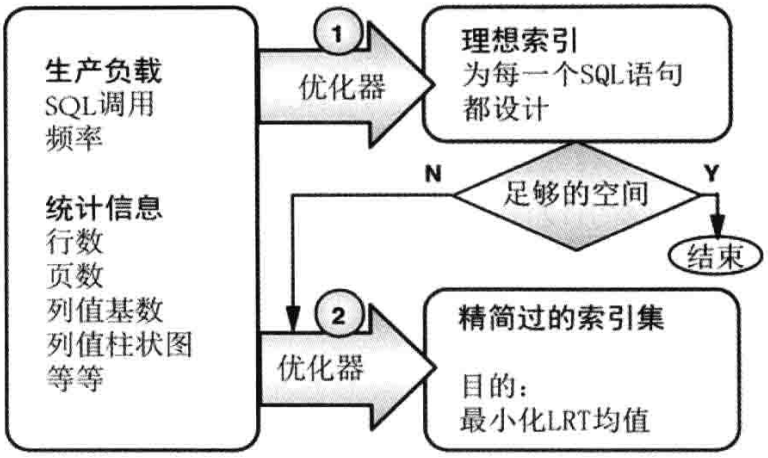
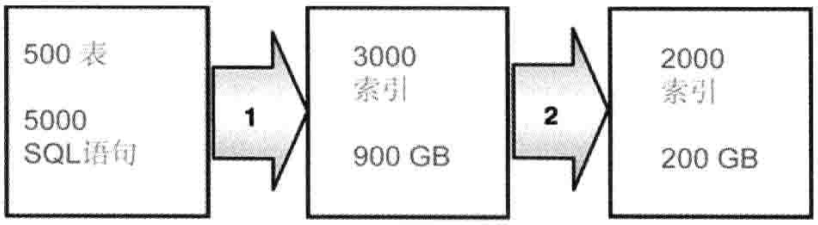


图 16.1　计算机辅助式索引设计

通常情况下，为一个给定的 SELECT 语句找出最佳索引是很快的，但这可能会导致表上有过多的索引。于是，工具必须从中找出价值最低的索引。显然，当有许多 SELECT 语句时，这是一个非常耗时的任务。想象一下，为了评估删除一个索引带来的影响，工具需要进行多少次成本估算。据工具的开发者说，这是一个庞大的搜索范围。有必要使用启发式的快捷方式将执行时间控制在可接受的范围内。当然，这会影响目标的完成质量，如图 16.2 所述。另外，最佳索引集并不一定必须是理想索引集的子集。



方案质量	多半是好的	多变的
执行时间	多半是可接受的	非常长

图 16.2　精简索引集可能是极度耗时的

现今的许多工具都与象棋计算机类似——可以限定计算时间的上限。于是，最终产出的结果即为给定时间内所能得出的最好方案。

目前，一个实际的方法似乎是每次只向索引设计工具提供一个 SELECT 语句（参见图 16.3）。工具于是便会生成出所能达到的最佳索引，即使是对于一个涉及多表连接的查询语句也是如此——只要我们确保使用了最差输入值而非绑定变量。当然，这不是一个可有可无的事情——请不要忘了前文提到过的过滤因子缺陷。我们接下来要做的是基于每张表所能容许的索引数量来做出一个折中的方案，并为每张表挑选所需的索引，可能还会需要对一些索引进行合并。 291

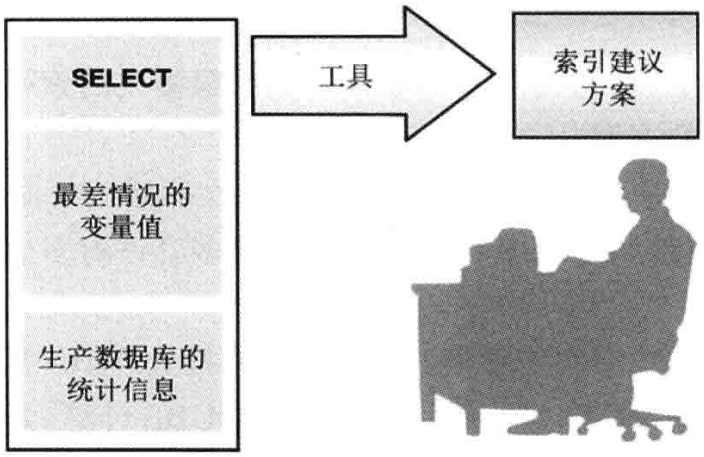


图 16.3　切实的方法

索引设计的基本方法在图 16.4 中进行了小结。

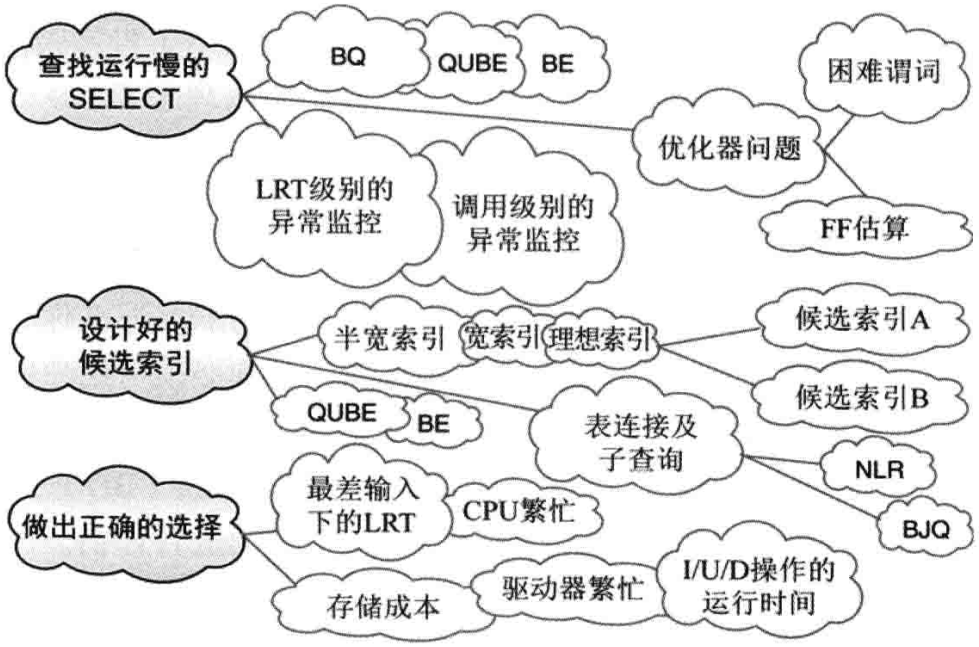


图 16.4　索引设计基本方法总结

292 设计出色索引的 9 个步骤

1. 当表结构第 1 版设计（主键、外键、表行顺序）完成时，就开始创建第 0 版的索引：主键索引、外键索引及候选键索引（如果有的话）。
2. 对第 1 版表结构设计的性能表现进行检查：使用 QUBE 评估一些重负载事务和批处理程序在理想索引下的响应时间。若评估结果无法满足要求，则将那些具有 1：1 或 1：C（1 对 0 或 1）关系的表进行合并，同时将冗余数据添加至有 1：M（一对多）关系的依赖表中。
3. 当表结构基本稳定后，你可以开始添加一些明显需要的索引——基于对应用系统的理解。
4. 若一个表的变化频率很高（如每秒有大于 50 次的插入、更新或删除），那么你应该用 QUBE 评估一下该表最多容许有多少个索引。
5. 当知道一个程序的数据库处理模式（事务型或批处理型）后，就需要用最新的数据库版本进行最坏输入下的 QUBE 计算。若评估出一个事务的本地响应时间超出了应用的警戒值（如 2 s），则
293 表明当前的数据库版本无法满足该程序。对于一个批处理程序而言，对响应延时的接受度必须针对具体情况逐个评估。不过，为了避免长时间锁等待，相邻两个事务提交点之间耗时的告警阈值应该与本地响应时间的告警阈值相同。一旦超出了告警阈值，你就应当对索引进行改进（半宽索引、宽索引或理想索引）。若在使用了理想索引的情况下评估结果（QUBE）仍不令人满意，或者所需的索引数量超出了第 4 步中评估的表上所容许的最大索引数，那么你需要根据第 15 章中所讨论的问题对这些慢查询进行更精确的评估。若评估出的响应时间仍旧过长，那么你必须像第 2 步中那样修改表的设计。最差情况下，你必须与用户协商调整需求，或者与管理人员协商调整硬件配置。
6. SQL 语句被编写后，开发人员就应使用基础问题（BQ），或者如果可行的话，用基础连接问题（BJQ）对其进行评估。
7. 当应用程序发布至生产环境后，有必要进行一次快速的 EXPLAIN 检查：对所有引起全表扫描或全索引扫描的 SQL 调用进行分析。这一检查过程也许能发现不合适的索引或优化器问题。

8. 当生产系统正式投入使用后，需要针对首个高峰时段生成一个 LRT 级别的异常报告（尖刺报告或类似的报告）。若一个长响应时间问题并非由排队或优化器问题引起，那么你应该用第 5 步中的方法进行处理。
9. 至少每周生成一个 LRT 级别的异常报告。

变体 1

若没有时间对所有的程序都进行 QUBE 计算，那么可以仅对那些可能出问题的程序进行评估（例如，浏览类事务及大规模批量处理程序）。

变体 2

若无法实现 LRT 级别的异常报告，那么可以用一个较低的阈值（例如，100 ms 的响应时间）生成一个调用级别的异常报告。另外，若有运行明显较慢的事务未被调用级别的异常报告捕获，那么也应当用 QUBE 对其进行分析。

备注

在第 2 步中，一个表结构设计图就够用了，并不需要完整的 CREATE TABLE 语句。由此，一旦第一个程序被定义完成，索引设计就可以开始了。随后定义的程序可能需要额外追加表和索引列。

备注 2

294

在第 2 步至第 5 步中，对于数据库处理的描述必须包含相应 SQL 语句的相关信息，不过 SQL 的具体结构（如连接或子查询）是不需要的。在这个阶段，我们应当假定优化器会选择（或能被强制指定选择）使用我们所设想的访问路径。

参考文献

1. Peter Gulutzan and Trudy Pelzer, *SQL Performance Tuning*, Addison-Wesley, Reading, MA, 2002.
2. Mark Gurry, *Oracle SQL Tuning Pocket Reference*, O'Reilly, Sebastopol, CA, 2002.
3. Sharon Bjeletch, Greg Mable, Vipul Minocha, and Kevin Viers, *Microsoft SQL Server* 7.0 *Unleashed*, SAMS Publishing, Indianapolis, IN, 1999.
4. C. J. Date with Colin J. White, *A Guide to DB2*, Addison-Wesley, Reading, MA, 1990.
5. Marianne Winslett, Jim Gray Speaks Out, *www.acm.org/sigmod/record/issues/0303/Gray_SIGMOD_Interview_Final.*
6. Amy Anderson and Michael Cain, *www-1.ibm.com/servers/enable/site/bi/strategy/index.html.*
7. Dennis Sasha and Philippe Bonnet, *Database Tuning—Principles, Experiments, and Troubleshooting Techniques*, Morgan Kaufmann Publishers, San Francisco, CA, 2003.
8. Ari Hovi, *Data Warehousing*, Tietovarastotekniikka, Talentum, Helsinki, 1997.
9. Donald K. Burleson, *www.dba-oracle.com/art_dbazine_idx_rebuild.htm.*

术语表

索引设计方法中所涉及的术语

以下是我们在讨论索引设计方法时所使用到的术语。这些术语并不是现今人们正在使用的任何一个数据库管理系统的官方术语。

设计最佳索引的算法 首先设计一个宽索引（第三颗星）以使所需扫描的索引片尽可能地窄（第一颗星）。若该索引能避免排序（第二颗星），那么这个索引即为三星索引。若无法避免排序，那么它将只是二星索引，即牺牲了第二颗星。于是，我们便需要设计另外一个能避免排序的候选索引，即拥有第二颗星，牺牲第一颗星。这两个二星索引中的一个最终将成为相应 SELECT 语句的最佳索引。

辅助式随机读 本书中用于指代自动跳跃式扫描、列表预读及数据块预读的术语。

辅助式顺序读 本书中用于指代将一个游标拆分为多个范围谓词游标，每个游标分别扫描一个索引片。当多个处理器和磁盘驱动器可用时，响应时间将相应地降低。

BJQ，基础连接问题 是否存在或计划设计一个包含所有本地谓词列的索引（包含语句涉及的所有表的本地谓词）？

BQ，基础问题 是否存在或计划设计一个包含所有 WHERE 语句涉及的列的索引（一个半宽索引）？

最佳索引 为一个 SELECT 语句所能设计出的最好的索引。这个索引可能拥有三颗星，即理想索引；也可能只有两颗星，由于 ORDER BY 和范围谓词同时存在而不得不牺牲第一颗星或第二颗星。

调用级别的异常监控 生成异常报告，该报告展示了某监视时段中运行得最慢的 SQL 调用。如果需要从一个有大量不同 SQL 语句的慢程序中找出那些运行得很慢的 SQL 调用，这个报告是非常重要的。

候选索引 A 和 B 最佳索引设计算法中所使用到的两个索引。

CPU 系数 用于计算随机访问、顺序访问、FETCH 和排序的 CPU 时间的值。

CQUBE，CPU 时间的快速上限估算　快速、粗略地估算 SQL 的 CPU 时间需要用到四个变量：TR、TS、F 和 RS。

298 **问题制造者**　一个独占了资源的事务，可能是由不合适的索引导致的。

DB2 for LUW　一个运行在 Linux、UNIX 和 Windows 平台上的 DBMS。

困难谓词　无法参与定义索引片的谓词——即该谓词无法成为匹配谓词；有时我们称此类谓词不可索引。

宽索引　一个包含了 SELECT 语句所涉及的所有字段的索引，在使用这类索引的情况下，不需要进行表访问。

过滤因子缺陷　指这样一种场景：性能最差的情况并非发生在过滤因子值最高时，可能是由于访问路径中没有排序，一旦取得第一屏数据就返回，且所有的谓词列都未参与定义索引片。

第一颗星　SELECT 语句所需的索引行都是相邻的，或至少尽可能地相互靠近。这颗星尽可能地将所需扫描的索引片的厚度降至最低。

理想索引　拥有三颗星的索引。

索引片　索引被扫描的部分。索引片的厚度会影响对表的同步读的数量。

LRT 级别的异常监控　生成异常报告，该报告展示了本地响应时间或 SQL 运行耗时特别长的操作型事务。

NLR，本地行的数量　在最大过滤因子条件下，使用本地谓词筛选后剩余的本地行数。该指标被用于预测连接的最佳表访问顺序——将具有最低 NLR 的表作为最外层表。

QUBE，快速上限估算　一个只使用两个变量（TR 和 TS）的简易的本地响应时间评估方法，通常用于在早期发现可能运行得很慢的访问路径。若估算所使用的最差情况的过滤因子与真实值相近，那么就能够揭示索引及表设计相关的性能问题。从定义可以看出 QUBE 是悲观的，它有时会做出错误的警告。

确实困难的谓词　无法参与索引过滤过程的谓词。我们因此需要访问索引片上的每一个表行来匹配谓词，即便谓词列已经被拷贝至索引上也是如此。

第二颗星　索引行的顺序满足 SELECT 语句中 ORDER BY 的要求。满足这颗星的查询不需要对结果集进行排序。

半宽索引　一个包含 WHERE 子句中所有列的索引，使用半宽索引将使得访问路径仅在必要时才访问表。

尖刺　一个本地响应时间或 SQL 运行时间特别长的操作型事务。

第三颗星　包含 SELECT 语句中所有列的索引。满足这颗星意味着避免了表访问——访问路径只需访问索引，第三颗星通常是最重要的一颗星。

299 **三星索引**　一个针对给定 SELECT 语句的理想索引。

访问 DBMS 读取一个索引行或表行的成本。如果 DBMS 扫描一个索引或表的片段（被读取的表在物理上是相邻的），读取片段的第一行将引起一次随机访问，读取剩余的行将引起每行一次的顺序访问。一次索引访问和一次表访问的成本相同。

调优的潜在空间 计划中的索引改进所能实现的本地响应时间提升上限。

受害者 由于需要等待资源而被问题制造者所影响的事务。

通用术语

访问路径 优化器为了获取某个 SQL 语句的结果集所使用的方法。对于单表 SELECT 而言，这是指以某种方式使用某个索引或者进行全表扫描；对于连接查询而言，这是指在前者的基础上增加表的访问顺序和连接方式。

异步读 在前一组页还在处理中时就开始下一组页的读取，I/O 时间和处理时间之间可能有很大一部分的重叠，理想情况下，这些页的异步 I/O 在被请求处理之前就已经完成了。这一过程被称为预读。

位图索引 用于替代 B 树索引，索引列的每一个值都有一个位向量。适用于基数很小且变更频率极低的列。位图索引能使某类有复杂 WHERE 子句的查询运行得很快，尤其是在数据仓库系统中。有些 DBMS 产品并不支持此类索引。

块 参见术语“页”的解释。

布尔谓词 如果当一个谓词被校验为假时就能排除一行，那么该谓词就被称为布尔谓词（BT），否则它就是非布尔的。非 BT 谓词可能使得 WHERE 子句对于优化器而言太困难了——访问路径不是最优的。若一个 WHERE 子句中没有 OR 操作符，则所有的谓词都是 BT 的。

B 树索引 这是最常见的索引类型。索引上的列（通常）是从单个表中拷贝的。索引的最底层（叶子页）包含了指向每一个表行的指针。叶子页层拥有其自己的索引树，该树的最顶层被称为根页。

缓冲池 计算机内存的一个区域，从磁盘上读取的索引或表页被存放在这里。缓冲池可能被划分为多个子池，并将其分配给单独的索引和表。缓冲池的管理者试图将频繁使用的数据保留在池中以避免从磁盘上进行额外的读取。

聚集索引 在 SQL Server 中是指一个包含表行的索引，在 DB2 中是指任何一个索引行顺序与表行顺序相同或计划相同的索引。

聚簇索引 这类索引使得 DBMS 在向表中添加记录时，将新记录添加至由聚簇索引键所定义的主页（或接近主页的页）上。一张表上只能有一个聚

簇索引。有些 DBMS 产品并不支持聚簇索引，但我们可以通过对表进行重组或重新加载来使表行按照所需的顺序存储。

覆盖索引　在 SQL Server 中是指一个包含了 SELECT 语句所涉及的所有列的索引，它能够避免表访问。这与 SQL Server 中的术语表查询相对，表查询是指访问路径使用了索引但仍需从表中读取记录。

300

CPU 缓存　CPU 芯片的高速内存，用于存储运行最频繁的程序指令和数据。

游标　嵌入式 SQL 中的一个结构。用于通过 FETCH 调用向应用程序传递结果集，每次传递一行。

数据库　逻辑上相关的数据，一个 DBMS 相关的表的集合。

按数据分区的二级索引　将超大索引进行分区，每个表分区对应一个索引分区，从而降低不可用率。

数据块预读　Oracle 中用于表达这样一个过程的术语：从索引片上收集多个指针，从而使得能够发起多重随机 I/O 来并行读取表行。

DBMS，数据库管理系统　用于支持、管理及控制所有数据库的使用行为的软件。现在网络和分层系统已经几乎被关系型系统所取代了。

数据仓库　一个提供持续的时间相关数据的业务环境，用于决策支持。被设计用来支持大量各种类型的不可预测的查询和报告。通常这些数据是从多个操作型系统加载过来并转换成统一格式的。它能够支持趋势分析，比如每月销售情况的对比。

默认值　在没有特别给定值的情况下 DBMS 所使用的假定值。

反范式化　向一个表中添加冗余数据，这在数据仓库类环境中非常常见，在操作型数据库中有时也需要使用这一方式以提升 SELECT 语句的性能。

磁盘驱动器　一个存储设备，该设备能够在同一时间支持一次读取或一次写入操作。

执行计划　DBMS 用于描述所使用的访问路径的输出信息。

事实表　包含详细的事务数据，如包括销售数据、价格或优惠之类的销售或支付条目。这些数据在维度数据的辅助下被汇总或分组。

宽表　一张包含其他表中部分字段或由几张表合并而成的表。

FETCH　在游标操作中用于提取数据的 SQL 调用，每次提取一行记录。

过滤因子　被用来定义谓词的选择性——满足谓词条件的记录数占表行总数的比例。该值取决于列值的分布情况。当评估一个索引是否合适时，最差情况的过滤因子比平均的过滤因子更重要。

外键　一个指向另一张表或本表的主键的列或组合列。外键上通常有引用完整性约束来保证数据的完整性。

空闲空间　为了支持新添加到表及索引中的行，当页被加载或重组时，页中

预留出一定比例的空闲空间。

哈希连接 一种连接方式。这种连接方式首先对一张表上的连接列进行哈希计算，再将其存储在一张临时表中（本地谓词已参与数据过滤）；对于另一张表上满足本地谓词的记录，则通过其哈希值对临时表中的记录进行校验。 301

提示 在 SQL 调用或绑定选项中所指定的规范，用以影响优化器，提示的语法与具体的产品相关。我们应当只在优化器由于不合理的成本估算而无法选择最佳访问路径时才使用这一技术。

绑定变量 WHERE 语句中所使用的程序变量，如 WHERE SALARY>:SALARY 中的:SALARY。

索引匹配 通过谓词限定待扫描的索引片的大小。一个或多个列（有时称为匹配列）确定了索引片的起始位置，并使用 B 树索引的结构找到第一个所需的索引条目。同样地，这些列也确定了扫描的结束位置。匹配谓词有时被称为范围限定谓词.

只需访问索引 一个不需要访问表就可以提供所有被请求的数据的访问路径。

索引前读 SQL Server 的一个术语。用于指代向前读取下一组叶子页。

索引过滤 那些无法参与匹配过程的索引列仍然能够被用来与谓词的输入值相比较，如此一来，表访问将只在必要的时候才发生。

索引跳跃式扫描 一个 Oracle 的术语，用于指代读取多个索引片而不是进行全索引扫描。

完整性 数据库的一个状态，在该状态下，所有数据的约束和规则都是有效的。

I/O 处理器发起的一个从磁盘读取一个页或向磁盘写入一个页的请求，该写入伴随着一次更新。

连接方式 优化器做出的关于如何连接表的决定，通常优化器会选择嵌套循环，虽然其他方式也是可用的。

叶子页 索引最下层的页，该页包含了键值及其对应表行的指针，索引行是按键值的顺序存储的。

最近最少使用算法 一个通常被缓存管理器和磁盘服务器缓冲管理器所使用的算法，用于定位那些为了满足新请求而应当被覆盖的页。

列表预读 DB2 for z/OS 中所使用的一种机制，对指向表行的索引指针按照表行所属页的顺序进行排序，从而使得 DBMS 可以通过跳跃式顺序读的方式访问表行。

本地响应时间 排除工作站及服务器之间的传输及等待时间之后，一个事务的运行时间，即服务器响应时间。

锁　一个针对顺序处理所设计的数据结构，用于保证逻辑一致性，通常关联至一张表、页或行。

物化结果行　执行所需的数据库访问来构建结果集。在最佳情况下，这一过程只需将一行记录从数据库缓冲池传递给应用程序。在最差情况下，DBMS 需要发起大量的磁盘读请求。

302

合并扫描　一种连接方式。在此种连接方式下，一张或多张表按统一的顺序被排序（在使用本地谓词过滤过之后），然后再将表或工作文件中满足条件的行进行合并（Oracle：*排序合并连接*）。

镜像　将所有的页同时写入两个磁盘驱动器。

多块 I/O　Oracle 顺序读。

多索引访问　从多个索引或同一索引的多个索引片上采集指针，对其进行比较，然后再访问所需的行。也被称为*索引与*（索引交集）和*索引或*（索引并集）。

多重顺序前读　SQL Server 顺序读。

多行 FETCH　在游标操作中用一次 SQL 调用同时请求多行数据。

嵌套循环　一种连接方式。在这种连接方式下，DBMS 首先从外层表上找出一行满足本地谓词的数据，然后再在下一个表（即内层表）上查找相关的行，并校验哪些是满足内层表的本地谓词的，以此类推。

非叶子页　叶子页以外的索引页，这类页包含了一个（可能被截断了的）键值，该键值包含了指向下一层级的页的指针及该页上的最大键值。

null　一个空值或未知值。当存储一个表行时，若表行中有一列未指定值，那么 DBMS 会用一个特殊的标识来指代 null。

优化器　关系型数据库管理系统的一个组件，该组件负责为每一个 SQL 语句选择访问路径。它会评估每一个合适的访问路径的成本，通常这是基于 I/O 时间和 CPU 时间的权重之和来得到的。

页　索引行和表行是按页组织在一起的（Oracle 使用“块”这一术语）。通常，页的大小为 4KB，但也有使用其他大小的页的情况。页的大小决定了能在页中存储的索引行和表行的数量。在读取数据时，DBMS 会从磁盘上读取一整页到缓冲池中，因此单个 I/O 能够一次读取多行。

谓词　SQL 语句的 WHERE 子句中的一个搜索参数。

主键　唯一确定一个表行的一个或一组列。

查询　一个以 SQL 的形式表达的数据请求，该请求提供了满足搜索参数的数据行。

RAID 5　一个由廉价的磁盘所组成的 5 级冗余阵列——一个常见的存储数据的方法——逻辑卷被条带至多个构成一个 RAID 组的磁盘驱动器上。例

如第一个 32 KB 的条带被写至磁盘驱动器 1，第二个条带被写至磁盘驱动器 2，以此类推。

RAID 10 适用于频繁随机写入、更新或删除的数据库，实际上它就是 RAID 0 镜像 + RAID 1 条带。与奇偶校验冗余数据不同，一个更新了的页会被同时写入两个磁盘驱动器，一个页也可以从任何一个磁盘驱动器上读取。由随机写导致的 RAID 10 的磁盘负载（磁盘繁忙度）比 RAID 5 低，但前者需要更多的磁盘驱动器。

读缓存 是磁盘服务器的半导体内存（RAM）中的一块区域，用于存储最近从磁盘驱动器上读取过的页。读缓存的目的是为了减少从磁盘驱动器读取页的次数。通常，读缓存比数据库的内存缓冲池要大得多，能保存在最近 20 分钟左右的时长内被随机读取过的页。 303

冗余 出于安全或性能的考虑，在磁盘驱动器或表中存储额外的数据拷贝。RAID 5 冗余是指为一个条带组（如 7 个条带，每个条带 32 KB）上的每一个位都存储一个奇偶校验位。有了这些奇偶校验位后，当磁盘损坏时，这些条带中的任何一个都能够被重建。

关系 在表的关系模型中所使用的术语。

关系型数据库 一个根据关系型模型用关系型 DBMS 构建的数据库。

重组 索引被重组以还原其正确的物理顺序，这对于索引片扫描和全索引扫描很重要。表被重组以还原空闲空间及表行的顺序。

根页 B 树索引结构中最顶端的页。

行 一个表行必须能被单个表页容纳，一个索引行必须能被单个叶子页容纳。

顺序预读 DB2 顺序读。

顺序读 将多个索引或叶子页从磁盘加载到缓冲池中。由于 DBMS 提前就知道需要请求哪些页，所以可以在页被真正请求之前先执行读取操作，这类操作包括顺序预读、多块 I/O 和多重连续预读。

服务时间 通常指 CPU 时间和同步磁盘 I/O 时间的总和。

跳跃式顺序 按一个方向扫描一组不连续的行。

条带 RAID 条带是指将索引或表的第一个条带（如 32 KB）存储在磁盘驱动器 1 上，将第二个条带存储在磁盘驱动器 2 上，以此类推，从而将负载平均分布在一组磁盘上。由于磁盘服务器可能提前从多个磁盘上并行读取数据，因此当请求下一组页时，这些页很可能已经被保存在磁盘服务器的读缓存中了。

汇总表 反范式化的事实表。由于是反范式化的，所以不需要连接，所有所需的列都在一张表里。

同步 I/O 当进行 I/O 操作时，DBMS 无法再进行其他操作，即它必须等待，

直至 I/O 操作完成。

表　关系的实现，由包含列值的行组成。

事务　用户和程序之间的一次与本地响应时间相关的交互，它由一个或多个 SQL 调用构成，有可能只是读取，也有可能涉及更新。

触发器　一个存储在数据库中的程序模块，当一个 SQL 调用访问表时，该程序模块会自动开始执行。不同的 SQL 调用都有各自专门针对插入、删除和更新操作的触发器。它们通常由 SQL 和程式化扩展（例如 SQL Server 中的 T-SQL 或 Oracle 中的 PL/SQL Oracle）编写而成。

视图　一个虚拟表，虽然它本身并不存储任何数据，但它提供了表列的一个子集，由一个包含 SELECT 语句的 create view 语句所定义。

写缓存　磁盘服务器的半导体内存（RAM）中的一个区域，该区域用于存储数据，并由电池电源保护（非易失性存储，NVS）。DBMS 每秒可能会
304 进行多次将被修改的页写至磁盘驱动器上的操作，这些页首先就会被存储在写缓存中。而缓存中的 LRU 页则会被写至磁盘驱动器上。写缓存可能包含了那些在最近几分钟内被更新过的页，若某个页被频繁地更新，那么该页可能会长久地停留在写缓存中。缓存越大，更新引起的磁盘负载（磁盘繁忙度）就越低。

索引

A

access path, 31
  hint, 35, 253, 262
access pattern, 60
alarm limit, 86, 122, 292
AND operator, 92
arithmetic expression, 91
AS/400 system, 56
assisted random read, 16, 275
asynchronous read, 19, 59, 116
audience, 3
automating, 8
auxiliary table, 88

B

B-tree index, 4, 26, 199, 204, 268
background, 3
balanced tree index, 13
basic estimates, 145
basic join question, 159, 173, 175, 293, 297
basic question, 9, 63, 125, 159
batch job, 65, 86, 108, 247, 274
batch program, 65, 86, 219, 292
best index, 54, 64, 79, 83, 84, 99, 253, 254
binary search, 70
binomial distribution, 210
bit vector, 25
bitmap index, 25, 198
block, 22
BQ, 64, 77, 100, 293
  verification, 87
browsing, 7, 247
bubble chart, 112, 131
buffer pool, 4, 13, 64, 69, 70, 260
  hit ratio, 64
  size, 271
  subpool, 270, 274
buffer pool hits, 260

C

cache hits, 260
candidate A, 55, 79, 83, 93, 99, 149, 167
candidate B, 55, 84, 99, 149
candidate key index, 24
cardinality, 39, 41, 246, 252, 255
Cartesian product, 185
Cloudscape, 251
cluster, 26
clustered, 27
  index, 24
  index scan, 68
clustering
  index, 23
  ratio, 162
column
  correlation, 38, 255
  fixed length, 23
  non-key, 238
  restrictive, 4
  variable length, 23, 232
  volatile, 7, 216
comebacks, 273
commit point, 86, 121
comparison
  performance, 80
computer-assisted index design, 290
control information, 13
cost, 21
  access selection, 36
  additional index, 58
  assumptions, 21
  CPU time, 21
  denormalization, 180
  disk servers, 21
  disk space, 21
  maintenance, 80

cost (*continued*)
memory, 21
sort, 84
storage, 21
counting touches, 70
covering index, 33
CPU
assumptions, 48
cache, 281
coefficients, 278
queuing, 122, 267
time, 21, 65, 112, 123, 303
time estimation, 278
CQUBE, 278
culprit, 112, 120, 124
cursor, 42, 92
split, 92, 248

D
data block prefetching, 18, 277
data integrity, 90
data warehouse, 186, 271
data-partitioned secondary indexes, 243
DB2, 7, 22, 23, 25–27, 33, 34, 42, 91, 93, 108, 111, 123, 132, 232, 233, 247, 250, 252, 275
DB2 for LUW, 23
DB2 for z/OS, 17
denormalization, 160, 192, 197, 283, 303
downward, 176
upward, 176
vs join, 180
dimension table, 185
disk
assumptions, 48
cache, 68
load, 7
space, 61
storage, 5
disk drive, 20
utilization, 267
disk read ahead, 20
disk server, 5, 70, 269
cache, 14
disorganization, 6, 68
drive load, 59
drive queuing, 20, 121

E
early materialization, 78, 176
education, 4
elapsed time, 123
exception
monitor, 111
monitoring, 7
exception report, 110, 112
call-level, 123, 293
LRT-level, 293
EXPLAIN, 33, 34, 44, 91, 94, 106, 108, 119
EXPLAIN PLAN, 34
extent, 25

F
fact table, 185, 192, 197
fat index, 51, 54, 61, 64, 190, 251, 272, 297
fault tolerance, 21, 61
FETCH
multi-row, 70, 282
field procedure, 91
files
open, 64
filter factor
pitfall, 94
filter factor, 37, 56, 65, 78, 89, 95, 107, 143, 252
first star, 50, 64, 298
FIRST_ROWS(n), 35
foreign key, 190
free space
distributed, 61
index calculation, 208
frequently used data, 15
full index scan, 87, 92, 106, 119
full table scan, 68, 70, 92, 106, 119
function, 91
fuzzy indexing, 174

G
get page, 123
guidelines, 4

H
hardware, 3
hardware capacity, 8
hashing, 26
histogram, 35, 111, 253

home page, 26, 69
host variable, 107
hot spots, 215, 273

I
IBM iSeries, 271
ideal index, 54, 62, 93, 143, 149, 167, 272, 292, 298
  join, 170
IN, 94
in-storage tables, 271
inadequate indexing, 4, 7, 9, 79
index
  ANDing, 195, 302
  backwards, 44
  block, 243
  both directions, 240
  candidates, 9
  clustering, 69, 74
  columns, 231
  composite, 7
  covering, 251
  creation examples, 234
  creeping, 208
  design, 56
  design method, 8
  design tool, 62
  function-based, 241
  hot spots, 217
  join, 200
  key truncation, 241
  length, 218, 232
  levels, 13
  locking, 232
  maintenance, 5, 58, 62
  multiple slices, 88
  non-clustering, 75
  non-leaf pages, 70
  non-unique, 12
  number, 232
  only, 64, 72, 79
  options, 237
  ORing, 195, 302
  pointer, 117, 195, 302
  resident, 274
  restrictions, 231
  row, 12
  row suppression, 233, 237
  screening, 64, 92, 108
  size, 232
  slice, 39, 54, 67, 68, 72, 297
  suppression, 33
  unique, 12, 240
index design summary, 291
index read-ahead, 18
index skip scan, 18, 90, 242
index-organized table, 24
inner table, 136, 302
insert
  pattern, 61, 208
  random, 208
  rate, 7
integrity check, 121
Intel servers, 21
interleave, 22
intermediate table, 190

J
join, 9
  comparison, 170
  hash, 165
  hash join, 285
  hybrid, 136
  ideal index, 170
  inner, 140
  merge scan, 136, 163, 285
  method, 136
  multiple, 171
  nested loop, 136
  nested loop, 140
  outer, 140

K
key
  descending, 150, 168
  foreign, 8
  modified, 6
  partial, 256
  primary, 8
  sequence, 6
  truncation, 269

L
leaf page, 5, 49, 67, 70, 204, 267
leaf page split, 24, 206, 226
  ratio, 207
LIKE, 91
LIO, 123
list prefetch, 17, 277
local disk drives, 21

local response time, 7, 112
lock wait, 66, 86, 112, 121
lock wait trace, 121
logical I/O, 123
LRT, 65
LRT-level exception monitoring, 126
LRU algorithm, 271

M
mainframe servers, 21
matching
  columns, 31, 72, 79
  index, 64
materialization, 42, 56, 301
  early, 44
matrix, 8
memory, 4, 55
mirroring, 25
misconceptions, 4
monitoring, 108, 271
  software, 9
  splits, 226
multi-block I/O, 16
multi-clustered indexes, 27
multiple index access, 92, 195
multiple serial read-ahead, 16
multitier environment, 65
myths, 4

N
network delays, 65
nine steps, 292
non-BT, 247
nonclustered indexes, 27
nonindexable, 33
nonleaf page, 5, 49, 64, 267
nonsargable, 33
nonSQL time, 120
NULL, 233, 251

O
one screen transaction, 153
optimizer, 30, 92, 148, 246
  cost based, 9, 162, 246, 253
  cost estimates, 107, 252
  cost formulae, 259
  CPU time, 261
  helping, 248, 261
  I/O time, 259
  transformation, 248
OPTIONS (FAST n), 35
OR, 92, 247
Oracle, 6, 18, 22, 23, 25, 26, 33, 35, 40, 90, 108, 123, 126, 131, 224, 233, 238, 247, 277
  cursor sharing, 254
  hints, 262
other waits, 112, 118, 122
outer table, 136, 139, 302

P
page, 12, 21, 302
  adjacency, 24
  number, 12
  request, 123, 125
  size, 22
parallelism, 19, 116
Peoplesoft, 255
perfect order, 70
performance
  monitor, 110
  problems, 2
  tools, 106
pitfall list, 91
pointer
  direct, 61
  length, 61
  symbolic, 61
pool, 273
predicate, 30
  compound, 30, 37, 255
  difficult, 33, 91, 246
  join, 136
  local, 136, 139, 142, 159, 185, 297
  matching, 33, 301
  non-boolean, 92
  nonindexable, 91, 298
  range, 53, 82, 88, 92, 252
  really difficult, 34, 247, 298
  redundant, 264
  simple, 30, 92
  stage 1, 34
  stage 2, 34
  suspicious, 91
prediction formulae, 9
prefetch, 16, 303
production, 4
  cutover, 4, 9
  workload, 8
promising culprits, 125

Q
QUBE, 9, 55, 58, 63, 65, 138, 246, 267, 292
  accuracy, 73
  assumptions, 66, 267
query table, 197
queuing, 121
  disk drive, 66
  theory, 122

R
RAID 10, 25, 59
RAID 5, 25, 59, 68
RAM, 21
random I/O, 69, 107
random read, 20
random touch, 78, 138, 260, 279, 299
  cheap, 67, 260, 275
  unproductive, 125, 179
randomizer, 26
reactive approach, 105
read cache, 5, 60, 70, 78, 88, 115, 272
  neighbors, 267, 270
redundancy, 25
redundant data, 159, 176
relational database, 1
reorganization, 23, 69, 70, 162, 206
  cost, 125
  frequency, 215
  interval, 215
  summary, 228
RISC, 21
root page, 4, 260
row, 21
  extended, 69
  inserted, 205
  length, 268
  long, 272
RUNSTATS, 34

S
SAP, 255
sargable, 33
screening, 32
second star, 50, 298
seek time, 20
selectivity ratio, 39
semifat index, 63, 78, 272
sequential
  prefetch, 16, 69
  read, 24, 116, 272
  touch, 67, 299
server
  cache, 15
service time, 65, 303
SHOW PLAN, 34
single-tier environment, 65
sixty four bit, 21, 269, 271
skewed distribution, 253
skip-sequential, 17, 90, 116, 275, 287
  benefit, 75, 277
  estimation, 276
  read, 70
sort, 55, 72, 96, 106, 145, 267, 282
  unnecessary, 250
spike report, 108, 110, 111, 112, 131, 132
split monitoring, 226
SQL call exception report, 131
SQL optimizer, 2
SQL pool, 255
SQL Server, 18, 22–24, 27, 35, 107, 123, 129, 200, 223, 232, 247
  hints, 263
stage 2 predicates, 247
standards, 6
star join, 185
statistics, 30, 56, 246
  falsifying, 264
stripe, 20
striping, 21, 25
subqueries, 175
  correlated, 176
  non-correlated, 175
summary table, 192, 303
superfluous index, 57, 62
  possibly, 58
  practically, 57
  totally, 57
suspicious access path, 106, 108
synchronous read, 19, 112, 114, 115, 116
system tables, 271
systematic index design, 9

T
table
  access order, 136, 140, 153, 161, 175, 187
  join, 136
  lookup, 33, 300
  resident, 274

table (*continued*)
small, 273
temporary, 163
touches, 70
unnecessary touches, 251
very small, 162
table design
optimistic, 175
unconscious, 180
textbooks, 4
thick slice, 31, 84
thin slice, 31, 53, 72, 79, 88
third star, 51, 75, 298
three star
algorithm, 54
three star index, 9, 49, 51, 79, 80, 100, 151
touch, 67, 69, 268
trace, 108
trigger, 88
tuning potential, 119
two candidate algorithm, 56, 62, 64

U
UNION, 176
UNION ALL, 94, 176, 249
UNIX, 21

V
victim, 112

W
wait for prefetch, 112, 120
Windows servers, 21
workfile, 271
write cache, 60

Broadview
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

通过设计适用于现今硬件的索引来提升关系型数据库的性能

在过去的几年中，硬件和软件的发展速度超出了所有人的想象，因此关系型数据库的性能也不如以前那么受人关注了。然而，事实却是硬件的发展速度并没有跟上日益增长的数据处理量。尽管磁盘压缩密度的大幅增加使得存储的成本很低且顺序读的速度很快，但是随机读的速度却仍旧很慢。于是，许多原有的设计建议已无法适用于现今的情况——索引的最优化点已经有了很大的提升。但许多原有的问题其实并没有消失——它们只是以另一种形式呈现而已。

本书提供了一种简单高效的设计索引和表的方法。作者通过大量的举例及案例研究描述了DB2、Oracle和SQL Server优化器是如何决定以何种方式访问数据库的，同时还阐述了快速估算所选择的访问路径的CPU及响应时间的方法。这使得对比不同设计方案的优劣成为了可能，且能帮助你在众多方案中选出最合适的那一个。

本书的目标读者是那些希望理解SQL性能相关内容，并希望了解如何有效设计表和索引的人。通过本书，拥有多年关系型系统经验的读者能够更好地判断新硬件的引入所可能带来的变化。

Tapio Lahdenmäki，数据库性能顾问，教授通用索引设计课程。他在IBM公司工作了三十多年，是公司全球课程中有关DB2 (for z/OS)性能相关课程的主要作者。

Michael Leach，关系型数据库顾问，已从IBM公司退休，他拥有二十年的应用系统及数据库课程的教授经验。两位作者的文章均被翻译成了多国语言广为传播。他们有关索引设计的方法被成功应用于许多核心系统。

上架建议：数据库

ISBN 978-7-121-26054-4

9 787121 260544

定价：79.00元

WILEY

策划编辑：张春雨
责任编辑：徐津平
封面设计：李　玲