SQL语言艺术

内容介绍

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

资深 SQL 专家 Stéphane Faroult倾力打造
《软件架构设计》作者温昱最新译作
巧妙借鉴《孙子兵法》的智慧结晶
传授25年的SQL性能与调校经验
深入探讨九种常见查询方案及其性能

前言

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

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

往往就是这样，那些所谓的“常识”和“可靠方法”最终却是极端有害的。先写低效的代码、后由专家调优，这种做法实际上是自找麻烦。本书认为，首先要关注性能的就是开发者，而且SQL问题绝不仅仅只包含正确编写几个查询这么简单。开发者角度看到的性能问题和DBA从调优角度看到的大相径庭。对DBA而言，他尽量从现有的硬件（如处理器和存储子系统）和特定版本的DBMS获得最高性能，他可能有些SQL技能并能调优一个性能极差的SQL语句。但对开发者而言，他编写的代码可能要运行5到10年，这些代码将经历一代代的硬件，以及DBMS各种重要版

本升级（例如支持互联网访问、支持网格，不一而足）。所以，代码必须从一开始就快速、健全。很多开发者仅仅是“知道”SQL而已，他们没有深刻理解SQL及关系理论，实在令人遗憾。

为何写作本书

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

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

战争的艺术和其他艺术一样，有它的历史和原则——否则，就不能成其为艺术。

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

目标读者

本书的目标读者是：

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

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

本书假定

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

本书内容

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

第1章，制定计划：为性能而设计
讨论如何设计高性能数据库
第2章，发动战争：高效访问数据库
解释如何进行程序设计才能高效访问数据库
第3章，战术部署：建立索引
揭示为何建立索引，如何建立索引
第4章，机动灵活：思考SQL语句
解释如何设计SQL语句
第5章，了如指掌：理解物理实现
揭示物理实现如何影响性能
第6章，锦囊妙计：认识经典SQL模式
包括经典的SQL模式、以及如何处理
第7章，变换战术：处理层次结构
说明如何处理层次数据
第8章，孰优孰劣：认识困难，处理困难
指出如何认识和处理比较棘手的情况
第9章，多条战线：处理并发
讲解如何处理并发
第10章，集中兵力：应付大数据量
讲解如何应付大数据量
第11章，精于计谋：挽救响应时间
分享一些技巧，以挽救设计糟糕的数据库的性能
第12章，明察秋毫：监控性能
收尾，解释如何定义和监控性能

本书约定

本书使用了如下印刷惯例：
等宽（Courier）
表示SQL及编程语言的关键字，或表示table、索引或字段的名称，抑或表示函数、代码及命令

输出。
等宽黑体（Courier）
表示必须由用户逐字键入的命令等文本。此风格仅用于同时包含输入、输出的代码示例。
等宽斜体（Courier）
表示这些文本，应该被用户提供的值代替。
总结：箴言，概括重要的SQL原则。

注意

提示、建议、一般性注解。为相关主题提供有用的附加信息。

代码示例

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

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

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

评论与提问

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

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

致谢

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

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

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

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

特别感谢Yann-Arzel Durelle-Marc慷慨提供第12章用到的图片。感谢Paul McWhorter授权我们将他的战争图用于第6章。

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

作者介绍

Stéphane Faroult从1983年开始接触关系数据库。Oracle法国成立早期他即加入（此前是短暂的IBM经历和渥太华大学任教生涯），并在不久之后对性能和调优产生了兴趣。1988年他离开了Oracle，此后一年间，他进行调整，并研究过运筹学。之后，他重操旧业，一直从事数据库咨询工作，并于1998年创办了RoughSea公司（http://www. roughsea.com）。

Stéphane Faroult出版了Fortran Structuré et Méthodes Numériques一书（法语，Dunod出版社，1986,与Didier Simon合作），并在Oracle Scene和Select（分别为英国和北美Oracle用户组杂志）以及Oracle杂志在线版上发表了许多文章。他还是美国、英国、挪威等众多用户组大会的演讲者。

Peter Robson毕业于达拉谟大学地质专业（1968年），然后在爱丁堡大学任教，并于1975年获得地质学研究型硕士学位。在希腊度过了一段地质学家生涯之后，他开始在纽卡斯尔大学专攻地质和医学数据库。

他使用数据库始于1977年，1981年开始使用关系数据库，1985年开始使用Oracle，这期间担任过开发工程师、数据架构师、数据库管理员等角色。1980年，Peter参加了英国地质普查，负责指导使用关系数据库管理系统。他擅长SQL系统，以及从组织级到部门级的数据建模。Peter多次出席英国、欧洲、北美的Oracle数据库大会，在许多数据库专业杂志上发表过文章。他现任英国Oracle用户组委员会主任，可通过peter.robson@justsql.com联系他。

查询的识别

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

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

查询的识别

Query Identification

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

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

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

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

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

总结：易识别的语句有助于定位性能问题。

保持数据库连接稳定

Stable Database Connections

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

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

处理每个文件的代码如下：

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

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

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

注意

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

表2-1：连接 / 中断性能测试结果

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

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

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

为何有如此大的性能提升？

第一个原因，也是最大的原因，在于数据库连接是很"重"的操作，消耗资源很多。

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

除了这些系统操作（创建进程或线程并开始执行）之外，数据库系统还必须为每

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

第二个原因，你的程序（甚至包括存储过程）和数据库之间的交互也有开销。

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

总结：数据库连接和交互好似万里长城——长度越长，传递消息越耗时。

战略优先于战术

Strategy Before Tactics

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

我们错失最有效的解决方案。

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

此存储过程的目的，是根据现有库存和各地订单，计算出总厂需要订购的原料数量。大体上，它就是把不同数据源的几个相同的表聚合（aggregate）到一个主表（master table）中。首先，将每个数据源的数据插入主表；接着，对主表中的各项数据进行合计并更新；最后，将与合计结果无关的数据从表中删除。针对每个数据源，重复执行上述步骤。所有 SQL 语句都不是特别复杂，也没有哪个单独的SQL语句特别低效。

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

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

总结：考虑解决方案的细节之前，先站得远一些，把握大局。

先定义问题，再解决问题

Problem Definition Before Solution

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

现成的解决方案中，非规范化设计引人注目。设计伊始，非规范化设计的拥护者就提出此方案，为了寻求“性能”而无视最终将会面临的升级恶魔——而事实上，在开发周期早期，改进设计（或学习如何使用join）也是一个不错的选择。作为非规范化设计的一种手段，物化视图（materialized view）常被认为是灵丹妙药。物化视图有时被称为快照（snapshot），这个更加平常的词更形象地反映了可悲的事实：物化视图是某时间点的数据副本。在没有其他办法时，这个理论上遭到质疑的技术也未尝不值得一试，借用卡夫卡（Franz Kafka）的一句名言：“逻辑诚可贵，生存价更高。”

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

所有技术方案，都只是我们达到目标的手段。没有经验的开发者误把新技术本身当成了目标。

对于热衷于技术、过于看重技术的人来说，此问题就更为严重。

总结：先打基础，再赶时髦：摆弄新工具之前，先把手艺学好。

直接操作实际数据

Operations Against Actual Data

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

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

总结：暂时工作表意味着以不太合理的方式存储更多信息。

用SQL处理集合

Set Processing in SQL

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

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

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

(2)万一修改失败，回滚消耗过长的实践。

许多人认为，进行大规模修改操作时，应在操作数据的代码中有规律地多安排些commit命令。其实，严格从实践角度来讲，“从头开始重做”比“确定失败发生的时间和位置，接着已提交部分重做”要容易得多、简单得多、也快得多。

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

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

动作丰富的SQL语句

Action-Packed SQL Statements

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

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

数据库访问，总会跨多个软件层，甚至包括网络访问。

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

在SQL中引入过程逻辑，意味着性能和维护问题应由你的程序承担。

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

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

总结：尽可能多地把事情交给数据库优化器来处理。

充分利用每次数据库访问

Profitable Database Accesses

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

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

一次操作全部完成。

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

总结：在合理范围内，利用每次数据库访问完成尽量多的工作。

接近DBMS核心

Closeness to the DBMS Kernel

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

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

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

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

```
create or replace function squeeze1(p_string in varchar2)
return varchar2
is
v_string varchar2(512) := '';
c_char    char(1);
n_len     number := length(p_string);
i         binary_integer := 1;
j         binary_integer;
begin
while (i <= n_len)
```

```
loop
c_char := substr(p_string, i, 1);
v_string := v_string || c_char;
if (c_char = ' ')
then
j := i + 1;
while (substr(p_string || 'X', j, 1) = ' ')
loop
j := j + 1;
end loop;
i := j;
else
i := i + 1;
end if;
end loop;
return v_string;
end;
/
```

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

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

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

还有第三种方法：

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

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

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

Elapsed: 00:00:00.00

SQL> select squeeze2('azeryt   hgfrdt    r')
2    from dual
3    /
azeryt hgfrdt r

Elapsed: 00:00:00.01

SQL> select squeeze3('azeryt   hgfrdt    r')
2    from dual
3    /
azeryt hgfrdt r

Elapsed: 00:00:00.00
```

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

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

declare

i         binary_integer;


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

```
commit;
end;
/
```

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

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

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

表2-2显示了在测试机上的运行结果。

表2-2：处理10 000条记录中空格所花的时间

函数	机制	时间
squeeze1	用 PL/SQL 循环处理字符	0.86 秒

squeeze2	Instr() + ltrim()	0.48 秒
squeeze3	循环调用 replace()	0.39 秒

尽管都在1秒内完成了10 000次调用，但 squeeze2的速度是squeeze1的1.8 倍，而 squeeze3则是它的2.2 倍。为什么呢？原因很简单，因为SQL 函数比PL/SQL"离核心更近'。当函数只偶尔执行一次时，性能差异微乎其微，但在批处理程序或高负载的 OLTP 服务器中性能差异就非常明显。

总结：代码喜欢SQL内核——离核心越近，它就运行得越快。

只做必须做的

Doing Only What Is Required

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

如果存在满足某条件的记录

那么处理这些记录

用代码直接实现就是：

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

if (counter > 0) then
```

当然，在 90% 的情况下，count(*) 是完全不必要的，正如上面的例子。要对多项记录进行操作，直接做即可，不必用count(*)。即使一个操作对任何记录都没有影响，也没有关系，不用count(*)没有什么不好。而且，即使要对未知的记录进行复杂处理，也能通过第一个操作就确定并返回受影响的记录——要么通过特殊的 API （例如 PHP 中的 mysql_affected_rows()），要么采用系统变量（Transact-SQL 中为@@ROWCOUNT，PL/SQL 中为SQL%ROWCOUNT），若使用内嵌式 SQL，则使用SQL通讯区（SQL Communication Area，SQLCA）的特殊字段。有时，

可以通过函数访问数据库然后直接返回要处理的记录数，例如 JDBC 的executeUpdate()方法。总之，统计记录数极可能意味着重复全部搜索，因为它对相同数据处理了两次。

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

总结：没必要编程实现那些数据库隐含实现的功能。

SQL语句反映业务逻辑

SQL Statements Mirror Business Logic

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

总结：检查数据库活动，看它是否与当时正进行的业务活动保持合理的一致性。

把逻辑放到查询中

Program Logic into Queries

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

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

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

```
END
```

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

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

一次完成多个更新

Multiple Updates at Once

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

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

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

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

```
update tbo_invoice_extractor
set pga_status = (case pga_status
when 1 then 0
when 3 then 0
else pga_status
```

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

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

总结：有可能的话，用一个语句处理多个更新；尽量减少对同一个表的重复访问。

慎用自定义函数

Careful Use of User-Written Functions

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

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

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

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

```
create or replace function airport_city(iata_code in char)
return varchar2
is
```

```
city_name   varchar2(50);
begin
select city
into city_name
from iata_airport_codes
where code = iata_code;
return(city_name);
end;
/
```

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

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

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

303个数据块，53个是从磁盘读出的——而且每行数据有个递归调用）。

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

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

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

总结：优化器对自定义函数的代码无能为力。

简洁的SQL

Succinct SQL

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

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

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

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

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

```
'9' || to_char(Param_dtAcc,'MM'))
order by rslt_period;
```

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

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

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

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

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

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

SQL的进攻式编程

Offensive Coding with SQL

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

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

有检查，支付操作才继续进行。

为了完成上述功能，不熟练的开发者会写出下列语句，并检查其返回结果：

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

接下来，他会做类似的工作，并再一次检查错误代码：

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

之后，他才会处理金融交易。

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

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

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

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

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

注意

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

“进攻式编程”的本质特征是：以合理的可能性（reasonable probabilities）为基础。例如，检查

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

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

简洁的SQL

Succinct SQL

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

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

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

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

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

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

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

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

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

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

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

SQL的进攻式编程

Offensive Coding with SQL

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

有个很好的例子：进行一连串检查，每当其中一个检查所要求的条件不符时就产生异常。信用

卡付款的处理中就涉及类似步骤。例如，检查所提交的客户身份和卡号是否有效，以及两者是否匹配；检查信用卡是否过期；最后，检查当前的支付额是否超过了信用额度。如果通过了所有检查，支付操作才继续进行。

为了完成上述功能，不熟练的开发者会写出下列语句，并检查其返回结果：

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

接下来，他会做类似的工作，并再一次检查错误代码：

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

之后，他才会处理金融交易。

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

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

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

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

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

注意

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

佳例子。

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

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

精明地使用异常（Exceptions）

Discerning Use of Exceptions

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

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

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

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

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

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

```
SQL> insert into employees(emp_num, emp_name,
emp_firstname, emp_email)
2   select emp_num,
```

```
3         emp_name,
4         emp_firstname,
5         substr(substr(EMP_FIRSTNAME, 1, 1)
6              ||translate(EMP_NAME, ' "', '_   _'), 1, 12)
7   from employees_old;

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


Elapsed: 00:00:00.85
```

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

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

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

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

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

```
SQL> insert into employees(emp_num, emp_firstname,
2                          emp_name, emp_email)
3   select emp_num,
4         emp_firstname,
5         emp_name,
6         decode(rn, 1, emp_email,
7                      substr(emp_email,
```

```
8                    1, 12 - length(ltrim(to_char(rn))))
9                    || ltrim(to_char(rn)))
10  from (select emp_num,
11               emp_firstname,
12               emp_name,
13               substr(substr(emp_firstname, 1, 1)
14                 ||translate(emp_name, ' '', '_   _'), 1, 12)
15                       emp_email,
16               row_number()
17                 over (partition by
18                       substr(substr(emp_firstname, 1, 1)
19                        ||translate(emp_name,' '','_   _'),1,12)
20                       order by emp_num) rn
21        from employees_old)
22  /

3000 rows created.

Elapsed: 00:00:11.68
```

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

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

SQL的本质

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

SQL的本质

The Nature of SQL

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

SQL与数据库

SQL and Databases

关系数据库的出现，要归功于E.F. Codd的关系理论开创性研究成果。Codd的研究成果为数据

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

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

正如其名，结构化查询语言（Structured Query Language，SQL）无非是一种语言，虽然它与数据库关系密切。将SQL语言和关系数据库等同视之，或者更糟——与关系理论等同视之，都是错误的。这种错误就好比将掌握了电子表软件或文字处理软件视为掌握了“信息技术”。实际上，有些软件产品并非数据库，但它们也支持SQL（注1）。另外，SQL在成为标准之前也不得不与诸如RDO或QUEL等其他语言竞争，这些语言曾被许多理论家认为优于SQL。

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

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

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

初步总结一下，我们可以将SQL查询表示为一个两层的操作，如图4-2所示。第一层是一个关系操作的“核”，它负责找出我们要操作的数据集；第二层是“非关系操作层（non-relational layer）”，

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

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

SQL与优化器

SQL and the Optimizer

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

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

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

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

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

一些SQL方言会误导用户，使他们认为自己仍在关系世界中——但其实早就不是关系操作了。举个简单的例子：不是经理的员工当中，哪五个人收入最高？这是个现实生活中很合理的问题，

但它包含了明显的非关系描述。“找出不是经理的员工”是其中的关系操作部分，由此获得一个有限的员工集合，然后排序。有些SQL方言通过在select语句中增加特殊子句来限制返回的记录数，很显然，排序和限制记录数都是非关系操作。其他SQL方言（这里主要是指 Oracle）则采用另外的机制，即用一个名为rownum的虚拟字段（dummy column）为查询结果编号——这意味着编号工作发生在关系操作阶段。如果查询语句如下：

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

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

现在分析一下上面的代码。查询的关系操作部分仅从employees表中，以完全不可知的顺序，取出最先发现的五位非经理人员（只包含empname和salary字段）。别忘了关系理论指出，关系（以及描述关系的表）是无序的，关系中的元组（即记录）可以被存储或检索。上面的查询执行后，收入最高的非经理人员或许在查询结果中，或许不在，无从知道查询结果是否满足查询条件。

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

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

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

其实，正确的理解应该如图4-4所示。

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

上例虽然简单，但说明一旦查询中的关系操作结束，就再也回不去了。解决该问题最好的办法是：把查询结果传给一个外部查询的关系操作。例如：五个收入最高的非经理人员属于哪些个

部门？然而，需重点强调的是，此时无论优化器有多聪明，它都不会合并两个查询，而是按顺序分别执行它们。此外，中间查询的结果集将暂时放在临时存储设备中，可以是内存或硬盘，于是可供优化器选择的访问方法就受到了限制。一旦离开了纯关系操作层，查询语句的编写对性能影响重大，因为SQL引擎将严格执行它规定的执行路径。

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

总结：为了取得好的优化效果，应将大部分工作安排在关系层。

优化器的有效范围

Limits of the Optimizer

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

优化器需借助在数据库中找到的信息。

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

能够进行数学意义上的等价变换，优化效果才能最佳。

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

优化器考虑整体响应时间。

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

优化器改善的是独立的查询。

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

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

掌握SQL艺术的五大要素

Five Factors Governing the Art of SQL

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

获得结果集所需访问的数据量

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

数据总量

Total Quantity of Data

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

定义结果集的查询条件

Criteria Defining the Result Set

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

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

结果集的大小

Size of the Result Set

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

从技术的角度来看，查询结果集的大小并不重要，重要的是最终用户的感觉。用户的耐心，在很大的程度上和预期返回的记录条数有关：用户只检索一条记录，则他期望非常快，他不会关心整个数据库有多大。更极端的例子是，查询之后并未返回任何结果：好的开发者都会努力使

返回少量记录或不返回记录的查询尽量快，因为对用户而言，最令人沮丧的事莫过于等待了数分钟后，看到“无相符数据”的结果；若是按下回车键后马上察觉查询语句有误，而又无法终止查询，等待就更为恼人。最终用户情愿等待的，是预期返回大量数据时。如果把每个过滤条件定义的特定结果集看作中间结果，而最终结果是它们的交集（在条件中用and相连）或并集（在条件中用or相连），那么小型中间结果集的交集很可能为空。换言之，更精确的条件经常是零结果集产生的主要原因。无论何时，只要查询有可能返回零结果集时，都应该先检查那个最大可能导致空结果集的条件——尤其是在该检查执行非常快捷时。不用说，条件的顺序与条件所在上下文的关系十分密切，这在稍后“过滤”一节中讲述。

总结：熟练的开发者应该努力使响应时间与返回的记录数成比例。

表的数量

Number of Tables

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

连接（Join）

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

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

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

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

复杂查询与复杂视图

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

总结：表明简单的查询背后，可能隐藏着复杂的视图。

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

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

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

当视图返回的数据远多于上级查询所需时，放弃使用该视图（或改用一个较简单的视图），通常可使效率大为改善。首先，用SQL 查询取代主查询中用到的视图。对视图的组成部分有了整体的了解之后，要去除严格意义上不必要的部分就容易多了。改用较简单视图的效果也不错，从查询中去除了不必要部分，执行速度快多了。

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

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

并发用户数

Number of other Users

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

注意

第9章将更详细地探讨并发性。

过滤

Filtering

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

过滤条件的含义

Meaning of Filtering Conditions

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

下面是个常见的连接（join）的例子：

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

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

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

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

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

```
select * from parameters
```

```
where parameter_name like '%size'
and parameter_type = 'NUMBER'
```

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

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

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

总结：查询条件是有差异的，有的好，有的差。

过滤条件的好坏

Evaluation of Filtering Conditions

编写 SQL 语句时，应首先考虑的问题是：

哪些数据是最终需要的，这些数据来自哪些表？

哪些输入值会传递到 DBMS 引擎？

哪些过滤条件能滤掉不想要的记录？

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

在编写查询之前，我们甚至应该对过滤条件进行排序，真正高效的条件（可能有多个，涉到不同的表）是查询的主要驱动力，低效条件只起辅助作用。那么定义高效过滤条件的准则是什么呢？首先，要看过滤条件能否尽快减少必须处理的数据量。所以，我们必须倍加关注条件的编写方式，下面通过例子说明这一点。

蝙蝠车买主

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

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

```
select distinct c.custname
from customers c
join orders o
on o.custid = c.custid
join orderdetail od
on od.ordid = o.ordid
```

```
join articles a
on a.artid = od.artid
where c.city = 'GOTHAM'
and a.artname = 'BATMOBILE'
and o.ordered >= somefunc
```

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

暂不考虑优化器将如何改写此查询，我们先看一下这段代码的含义。首先，来自customers表的数据应只保留城市名为 Gotham 的记录。接着，搜索orders表，这意味着custid字段最好有索引，否则只有通过排序、合并或扫描orders表建立一个哈希表才能保证查询速度。对orders表，还要针对订单日期进行过滤：如果优化器比较聪明，它会在连接（join）前先过滤掉一些数据，从而减少后面要处理的数据量；不太聪明的优化器则可能会先做连接，再作过滤，这时在连接中指定过滤条件利于提高性能，例如：

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

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

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

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

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

本性难移，我偏爱这种较古老的方式。原因只有一个：从逻辑的角度来看，旧方法突显出数据处理顺序无足轻重这一事实；无论以什么顺序查询表，返回结果都是一样的。customers 表非常重要，因为最终所需数据都来自该表，在此例中，其他表只起辅助作用。注意，没有适用于

所有问题的解决方案，表连接的方式会因情况不同而异，而决定连接方式取决于待处理数据的特点。

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

蝙蝠车买主的进一步讨论

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

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

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

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

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

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

我们还可以这样编写查询：

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

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

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

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

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

或者：

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

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

注意
每当对大量记录做存在性检查时，选择in还是exists须斟酌。
利于多数 SQL 方言，非关联子查询可以被改写成from 子句中的内嵌视图。然而，一定要记住的是，in 会隐式地剔除重复项目，当子查询改写为 from 子句中的内嵌视图时，必须要显式地消除重复项目。例如：

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

编写功能等价的查询时，不同的编写方式就好像同义词。在书面语和口语中，同义词的意思虽

然大致相同，但又有细微差异，因此某个词在特定语境中更合适。同样，数据和处理的具体实现细节可以决定选择哪种查询方式。

蝙蝠车买主案例总结

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

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

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

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

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

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

总结：保证SQL 语句返回正确结果，只是建立最佳 SQL语句的第一步。

大数据量查询

Querying Large Quantities of Data

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

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

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

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

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

另一个“查询条件用错了地方”的经典例子，和在含有 group by 子句的查询中进行过滤操作有关。你可以过滤分了组的字段，也可以过滤聚合（aggregate）结果（例如检查 count() 的结果是否小于某阈值），或者同时过滤两者；SQL 允许在 having 子句中使用这类条件，但应该在 group by 完成后才进行过滤（比如排序之后再进行聚合操作）。任何影响聚合函数（aggregate

function）结果的条件都应放在 having 子句中，因为在 group by 之前无从知道聚合函数的结果。任何与聚合无关的条件都应放在 where 子句中，从而减少为进行group by而必须执行的排序操作所处理的数据量。

现在回过头来看客户与订单的例子，我承认先前处理订单的方法比较复杂。在订单完成之前，必须经历几个阶段，这些都记录在表orderstatus中，该表的主要字段有：ordid（订单ID）、status、statusdate（时间戳）等，主键由ordid和statusdate组成。我们的需求是列出所有尚未标记为完成状态的订单（假设所有交易都已终止）的下列字段：订单号、客户名、订单的最后状态，以及设置状态的时间。最终，我们写出下列查询，滤掉已完成的订单，并找出订单当前状态：

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

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

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

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

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

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

这个子查询会对 orderestatus 作'全扫描'，但未必是坏事，下面会对此加以解释。
重写的子查询条件中，等号左端的'字段对'有点别扭，因为这两个字段来自不同的表，其实不必这样。我们想让orders和orderstatus的订单ID相等，但优化器能感知这一点吗？答案是不一

定。所以优化器可能依然先执行子查询，依然要把orders和orderstatus这两个表连接起来。我们应该将查询稍加修改，使优化器更容易明白我们的描述，最终按照“先获得子查询的结果，然后再连接orders和orderstatus表”的顺序工作：

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

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

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

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

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

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

```
and o.custid = c.custid
```

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

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

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

总结：尽早过滤掉不需要的数据。

取出数据在表中的比例

The Proportions of Retrieved Data

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

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

录所要访问的数据页数量可能差异很大；此外，有的访问路径将以串行方式执行，有的则以大规模并行（parallelized）方式执行……。因此，再别拿"10%的记录"这根鸡毛当令箭了。

总结：当查询的结果集很大时，索引未必必要。

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

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

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

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

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

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

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

小结果集，直接条件

Small Result Set, Direct Specific Criteria

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

一般而言，通过一个表或通过两个表的连接查询较少记录，只要确保查询有适当的索引支持即

可。然而，当很多表连接在一起，并且查询条件要参照不同的表时（例如 TA 和 TB），会面临连接顺序的问题。连接顺序的选择，取决于如何更快地过滤不想要的记录。如果统计数据足够精确地反映了表的内容，优化器有可能对连接顺序做出适当选择。
当查询仅返回少量记录，且过滤条件直接针对源表时，我们必须保证这些过滤条件高效；对于非常重要的条件，必须事先为相应字段加上索引，以便查询时使用。

索引可用性

Index Usability

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

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

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

使用唯一性索引（unique index）检索单条记录

非唯一性索引（non-unique index）或基于唯一性索引的范围扫描（range scan）

查询的效率与索引的使用

Query Efficiency and Index Usage

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

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

数据散布

Data Dispersion

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

```
where customer_id between . .. and ...
```
或：
```
where supplier_name like 'SOMENAME%'
```

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

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

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

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

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

下面读取c3 ，使用如下范围条件：

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

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

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

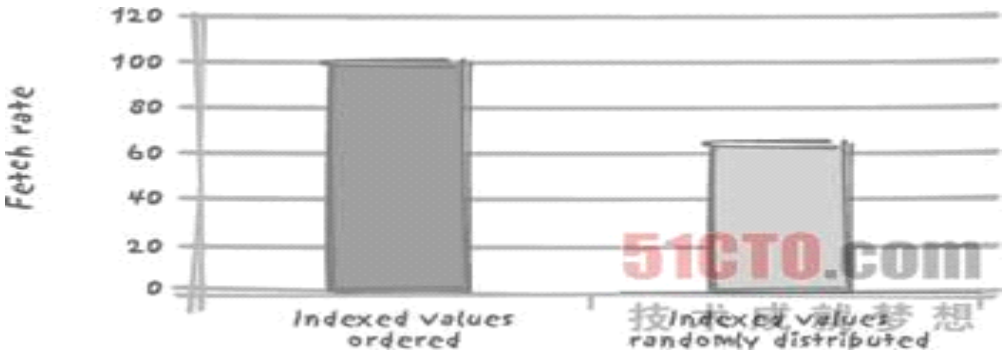

图6-1：“索引项顺序与表中记录顺序是否一致”对性能的影响

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

总结：类似的索引，性能却不同，这可能是物理数据的散布引起的。

条件的“可索引性”

Criterion Indexability

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

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

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

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

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

```
select
total.deptnum,
total.accounting_period,
total.ledger,
total.cnt,
error.err_cnt,
cpt_error.bad_acct_count
from
-- First in-line view
(select
deptnum,
accounting_period,
ledger,
```

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

group by
deptnum,
ledger,
accounting_period) error,
-- Third in-line view
(select
deptnum,
accounting_period,
ledger,
count(distinct account) bad_acct_count
from
glreport
where
amount_diff <> 0
group by
deptnum,
ledger,
accounting_period
) cpt_error
where
total.deptnum = error.deptnum(+) and
total.accounting_period = error.accounting_period(+) and
total.ledger = error.ledger(+) and
```

total.deptnum = cpt_error.deptnum(+) and
total.accounting_period = cpt_error.accounting_period(+) and
total.ledger = cpt_error.ledger(+)
order by
total.deptnum,
total.accounting_period,
total.ledger
外层查询where子句中的“(+)”是Oracle 特有的语法，代表外连接（outer join）。换言之：
select whatever
from ta,
tb
where ta.id = tb.id (+)
相当于：
select whatever
from ta
outer join tb
on tb.id = ta.id
下列SQL*Plus输出显示了该查询的执行计划：
10:16:57 SQL> set autotrace traceonly
10:17:02 SQL> /

37 rows selected.

Elapsed: 00:30:00.06

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

12 11 SORT (GROUP BY) (Cost=593908 Card=154 Bytes=5698)
13 12 TABLE ACCESS (FULL) OF 'GLREPORT'
(Cost=584519 Card=4370885 Bytes=161722745)

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

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

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

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

总结：如果必须进行全表扫描，表上的索引就没用了。

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

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

So this is the suggestion that I returned to the developer:

```
select  deptnum,
accounting_period,
ledger,
```

```
count(account) nb,
sum(decode(amount_diff, 0, 0, 1)) err_cnt,
count(distinct decode(amount_diff, 0, chr(1), account)) - 1
bad_acct_count
from
glreport
group by
deptnum,
ledger,
accounting_period
```

这个新的查询，执行速度是原先的四倍。这丝毫不令人意外，因为三次的完整扫描变成了一次。注意，查询中不再有where子句：amount_diff上的条件已被“迁移”到了select列表中decode()函数执行的逻辑，以及由group by子句执行的聚合（aggregation）中。

使用聚合代替过滤条件有点特殊，这正是我们要说明的“九种典型情况”中的另一种—— 以聚合函数为基础获得结果集。
总结：内嵌查询可以简化查询，但若使用不慎，可能造成重复处理。

小结果集，间接条件

Small Result Set, Indirect Criteria

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

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

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

取用第4章中例子的简化版本，找出订购蝙蝠车的客户，典型实现如下：

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

依我看，明确使用子查询来检查客户订单是否包含某项商品，才是较好的方式，而且也比较容易理解。但应该采用“关联子查询”还是“非关联子查询”呢？由于我们没有其他条件，所以答案

应该很清楚：非关联子查询。否则，就必须扫描orders表，并针对每条记录执行子查询——当orders表规模小时通常不会查觉其中问题，但随着orders表越来越大，它的性能就逐渐让我们如坐针毡了。

非关联子查询可以用如下的经典风格编写：

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

或采用from子句中的子查询：

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

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

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

多个宽泛条件的交集

Small Intersection of Broad Criteria

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

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

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

```
select distinct orders.custid
from orders
join orderdetail
```

```
on (orderdetail.ordid = orders.ordid)
join articles
on (articles.artid = orderdetail.artid)
where articles.artname = 'SOAP'
and
```

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

要知道哪些客户在上星期六买了肥皂，最明显而自然的选择是使用关联子查询：

```
select distinct orders.custid
from orders
where
and exists (select 1
from orderdetail
join articles
on (articles.artid = orderdetail.artid)
where articles.artname = 'SOAP'
and orderdetails.ordid = orders.ordid)
```

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

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

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

where articles.artid = orderdetail.artid
and articles.artname = 'SOAP') as sub_q
where sub_q.ordid = orders.ordid
and
```

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

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

多个间接宽泛条件的交集

Small Intersection, Indirect Broad Criteria

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

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

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

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

```
select (data from ttex_a,
ttex_b,
ttraoma,
topeoma,
ttypobj,
ttrcap_a,
ttrcap_b,
trgppdt,
tstg_a)
from ttrcapp ttrcap_a,
ttrcapp ttrcap_b,
tstg tstg_a,
```

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

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

TABLE_NAME	NUM_ROWS
ttypobj	186
trgppdt	366
tpdt	5370
topeoma	12118
ttraoma	12118
tbooks	12268

ttex	102554
ttrcapp	187759
tstg	702403

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

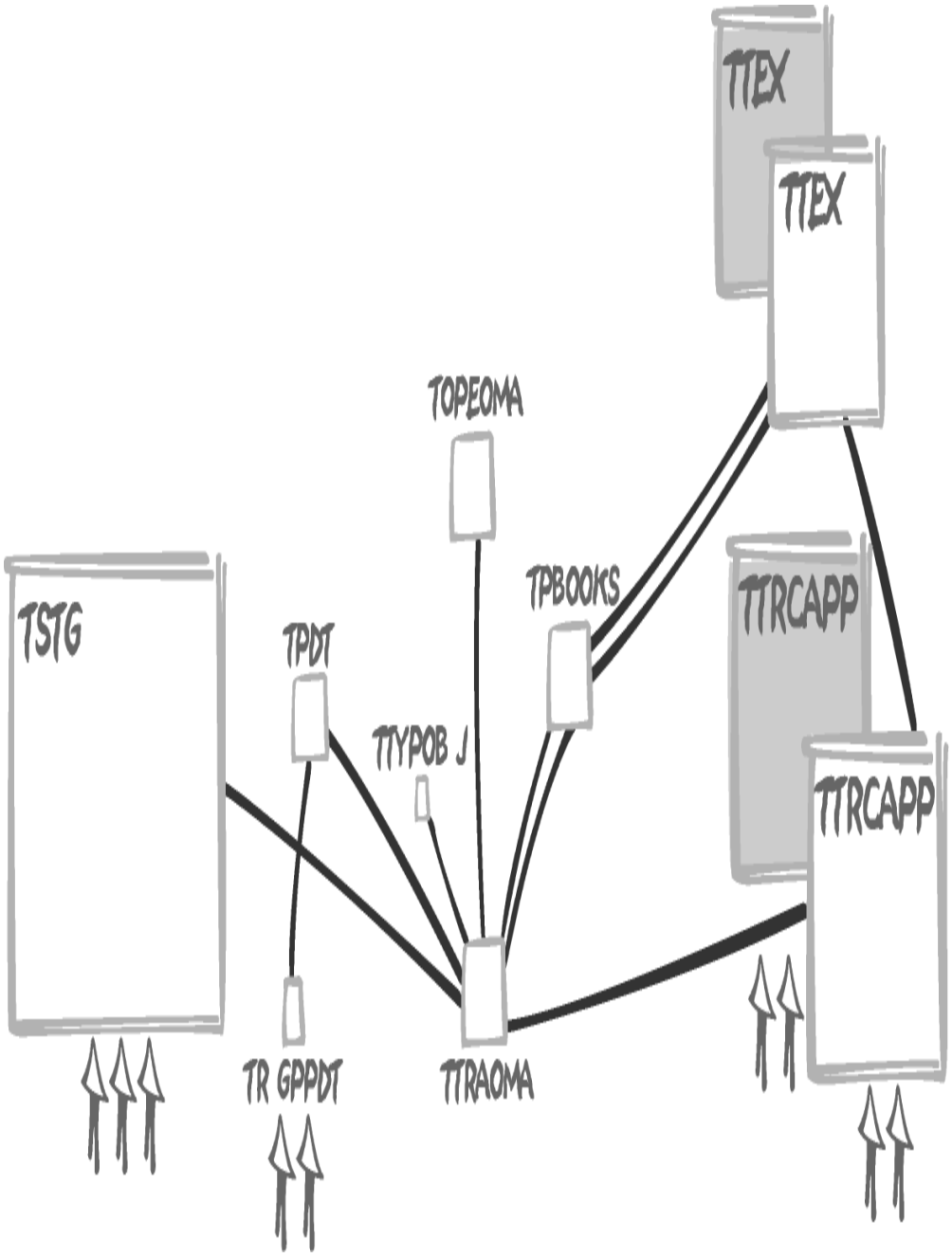

图6-2：数据的位置关系

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

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

优化器发现：tstg表有“大量敌军”，而查询中针对此表的选择条件比较弱，所以难以对它形成“正

面攻击”；而ttrcapp表在查询的from子句中出现两次，但基于该表的判断条件也较弱，所以也不会带来查询效率的提升；但是，ttraoma表的位置显然很关键，且该表比较小，适合作为“第一攻击点”——优化器会毫不犹豫地这么做。

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

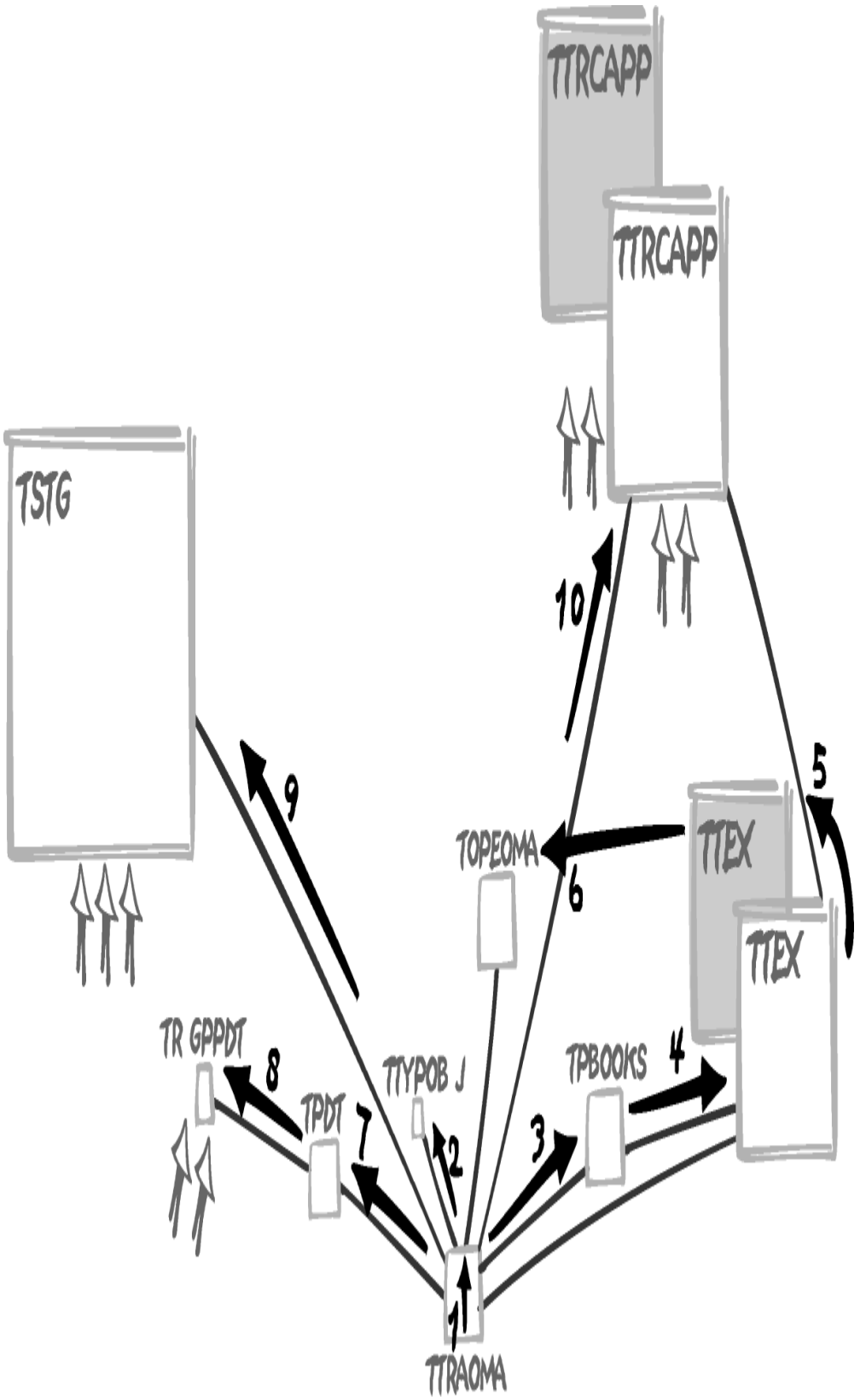

图6-3：优化器选择的执行路径

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

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

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

总结：记住，你应该详细说明所有强迫 DBMS 做的事。

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

```
select (select list)
from (select ttraoma.txnum,
ttraoma.bkcod,
ttraoma.trscod,
ttraoma.pdtcod,
ttraoma.objtyp,
...
from ttraoma,
tstg tstg_a,
ttrcapp ttrcap_a
where tstg_a.chncod = :7
```

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

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

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

大结果集

Large Result Set

无论结果集是如何获得的，只要结果集“很大”，就符合我们下面要讨论的“大结果集”的情况。

批处理环境下，产生大结果集是明智的。当需要返回大量记录时，只要查询条件的可选择性不高，那么即使结果集只占表中数据量的一小部分，也会引起DBMS引擎执行全表扫描；只有某些数据仓库例外，我们将在第10章中讨论之。

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

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

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

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

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

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

基于一个表的自连接

Self-Joins on One Table

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

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

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

```
select f.flight_number,
a.airport_name departure_airport,
```

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

此时，一般规则仍然适用：重点保证索引访问的高效。但是，如果此时索引访问不太高效怎么办呢？首当其冲地，应避免“第一轮处理丢弃了第二轮处理所需的记录”。应该通过一次处理收集所有感兴趣的记录，再使用诸如case 语句等结构分别显示记录，第11章将详细说明这种方法。非常微妙的是，有些情况看似与“机场的例子”很像，但其实不然。例如，如何利用一个保存“定期累计值”（注3）的表，显示每个时间段内累计值的增量？此时，该表内的两个不同记录间虽然有关联，但这种关联很弱：两个记录之所以相关，是因为它们的时间戳之间有前后关系。而连接两个flights表是通过airports表进行的，这种关联很强。

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

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

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

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

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

我们还有更安全的方法，就是使用基于“记录窗口（windows of rows）”的OLAP函数（OLAP function）。难以想象，这种本质上不太符合关系理论的技术可以显著提升性能，但应作为查询优化的最后手段使用。借助partition 子句，OLAP函数支持“分别处理结果集的不同子集”，比如

分别对它们进行排序、总计等处理。借助OLAP 函数row_number()，可以根据 statistic_id 建立子集，然后按时间戳增大的顺序为不同统计赋予连续整数编号，接下来，就可以连接statistic_id和两个序号了，如下例子所示：

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

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

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

很多时候，我们的数据并不像航班案例中那样具有对称性。通常，当需要查找和最小、最大、最早、或最近的值相关联的数据时，首先必须找到这些值本身（此为第一遍扫描，需比较记录），

接下来的用这些值作为第二遍扫描的搜索条件。而以滑动窗口（sliding window）为基础的OLAP函数，可以将两遍扫描合而为一（至少表面上如此）。基于时间戳或日期的数据查询，非常特殊也非常重要，本章在稍后的“基于日期的简单搜索或范围搜索”中专门讨论。
总结：当多个选取条件用于同一个表的不同记录时，可以使用基于滑动窗口工作的函数。

基于一个表的自连接

Self-Joins on One Table

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

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

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

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

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

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

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

```
select a.timestamp,
a.statistic_id,
(b.counter - a.counter)/5 hits_per_minute
```

```
from hit_counter a,
hit_counter b
where b.timestamp = a.timestamp + 300
and b.statistic_id = a.statistic_id
order by a.timestamp, a.statistic_id
```

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

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

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

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

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

```
order by timestamp) rn
from hit_counter) b
where b.rn = a.rn + 1
and a.statistic_id = b.statistic_id
order by a.timestamp, a.statistic_id
```

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

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

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

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

基于日期的简单搜索或范围搜索

Simple or Range Searching on Dates

搜索条件有多种，其中日期（和时间）占有特殊地位。日期极为常见，而且比其他数据类型更可能成为范围搜索的条件，范围搜索可以是有界的（如“在某两天之间”），也可以是部分有界（“在某天之前”）。通常，为了获得这种结果集，查询需要使用当前日期（如“前六个月”）。

上一节“通过聚合获得结果集”所举的例子，用到了sales_history 表。当时，条件位于amount 上，其实对于sales_history这种表更常见的是日期条件，尤其是读取特定日期的数据、或读取两个日期之间的数据。在保存历史数据的表中查找特定日期（或其对应值）时，必须特别注意确定当前日期的方法，它可能成为聚合条件的基础。

第1章已指出，设计保存历史数据的表颇为困难，而且没有现成的简单解决方案。无论你对当前数据、还是历史数据感兴趣，设计历史数据的存储方案都要根据如何使用数据决定，同时还要看数据多快会过时。例如，零售系统中价格的变动速度比较慢（除非正在经受严重的通货膨胀），而网络流量或财务设备的价格改变速度比较快，甚至快很多。

从宏观角度来看，关键是各项历史数据的数量：是“少量数据项、大量历史数据”，还是“大量数据项、少量历史数据”，或是介于两者之间？其重点是：数据项的可选择性取决于数据项的总数、取样频率（“每天一次”还是“每次改变时”）、时间长短（“永久”还是“一年”等）。因此，本节将首先讨论“大量数据项、少量历史数据”的情况，接着讨论“少量数据项、大量历史数据”的情况，最后讨论当前值问题。

大量数据项、少量历史数据

Many Items, Few Historical Values

既然没有为每个数据项保留大量历史数据，那么各项的ID可选择性很高。说明要查询哪些项，限定参与查询的少数历史记录，就可确定特定日期（当前日期或以前日期）对应的值。这种情况需要我们再次处理聚合值（aggregate value）。

除非建立了代理键（本情况不需要代理键），否则主键通常是复合键，由item_id和record_date组成。为了查询特定日期的值，可采用两种方法：子查询和 OLAP 函数。

使用子查询

查找某数据项在特定日期的值相对简单，但实际上，这种简单只是假象。通常你会遇到这样的代码：

```
select whatever
from hist_data as outer
where outer.item_id = somevalue
and outer.record_date = (select max(inner.record_date)
from hist_data as inner
where inner.item_id = outer.item_id
and inner.record_date <= reference_date)
```

考察这个查询的执行路径，我们发现：首先，内层查询与外层查询是有关联的（correlated），因为内层查询参照了item_id的值，该值是由外层查询返回的当前记录一个字段。下面，先来分析外层查询。

理论上，复合键中的字段顺序不会有太大影响，但实际上它们非常重要。如果我们误把主键定义为 (record_date, item_id)，而不是 (item_id, record_date)，前例的内层查询就非常依赖item_id字段的索引，否则无法高效地向下访问树状结构索引。但我们知道，额外增加一个索引的代价很高。

外层查询找到了保存 item_id 历史的各条记录，接着使用当前 item_id 值逐次执行子查询。注

意，内层查询只依赖 item_id，这与外层查询处理的记录相同，这意味着我们执行相同的查询、返回相同的结果。优化器会注意到查询总是返回相同的值吗？无法确定。所以最好不要冒这个险。

在使用关联子查询时，如果它处理不同的记录后总是返回相同的值，就没有意义了。所以，应该改用无关联子查询：

```
select whatever
from hist_data as outer
where outer.item_id = somevalue
and outer.record_date = (select max(inner.record_date)
from hist_data as inner
where inner.item_id = somevalue
and inner.record_date <= reference_date)
```

现在子查询的执行不需要访问表，只需访问主键索引就够了。

个人习惯各有不同，但如果 DBMS支持将“子查询的输出”与多个字段进行比较（这个特性不是所有产品都支持的），则应优先考虑基于主键比较：

```
select whatever
from hist_data as outer
where (outer.item_id, outer.record_date) in
(select inner.item_id, max(inner.record_date)
from hist_data as inner
where inner.item_id = somevalue
and inner.record_date <= reference_date
group by inner.item_id)
```

让子查询返回的字段，完全与复合主键的字段相符，有一定道理。如果必须返回“数据项值的列表”（例如是另一个查询的结果），则上述查询语句建议的执行路径非常合适。

只要每个数据项的历史信息数量都较少，以 in() 列表或子查询取代内层查询中的 somevalue，会使整个查询执行更高效。也可以用 in 子句取代“相等性条件”，在多数情况下没有什么不同；但偶有例外，例如，如果用户输错了item_id，采用in()时会返回未发现数据，而采用“相等性条件”时会返回错误数据。

使用OLAP函数

我们在自连接（self-join）情况下，使用了诸如row_number()等OLAP函数，它们在查询“特定日期某数据项的值”时也同样有用甚至高效。（但记住，OLAP函数会带来非关系的处理模式（注5）。）

注意

OLAP 函数属于 SQL 的非关系层。这类函数的作用是：在查询中做最后（或几乎是最后）处理。因为它们在过滤已完成后对结果集进行处理。

运用row_number()等函数，可以通过日期排序判断数据的“新旧程度（degree of freshness）”（也就是距离现在有多久）：

```
select row_number( ) over (partition by item_id
order by record_date desc) as freshness,
whatever
from hist_data
where item_id = somevalue
and record_date <= reference_date
```

选取最新数据，只需保留 freshness 值为 1 的记录：

```
select x.<suitable_columns>
from (select row_number( ) over (partition by item_id
order by record_date desc) as freshness,
whatever
from hist_data
where item_id = somevalue
and record_date <= reference_date) as x
where x.freshness = 1
```

理论上，使用 OLAP 函数方法和子查询几乎没有差异。实际上，OLAP 函数只访问一次表，即使需要为此而进行排序操作也不例外。OLAP函数对表不需要做额外的访问，甚至在使用主键快速存取时也是如此。因此，采用OLAP 函数速度会比较快（尽管只是快一点点）。

少量数据项、大量历史数据

Many Historical Values Per Item

当存在大量历史数据时，情况有所不同 —— 例如，监控系统中采集“度量值”的频率很高。这里的困难在于，必须根据对极大量的数据进行排序，才能找到特定日期或最接近特定日期的值。

排序是代价很高的操作：如果我们应用第4章的原则，降低非关系层厚度的唯一方法，就是在关系层多做一些工作，增加过滤条件的数量。此时，针对所需数据更精确地归类日期（或时间）以缩小范围，便非常重要。如果我们只提供上限，就必须扫描并排序所有历史数据。所以如果数据的采集频率很高，提供下限是有必要的。如果我们成功地把记录的“工作集”控制在可管理的大小，就相当于回到了“少量历史记录”的情况。如果无法同时指定上限（例如当前日期）和下限，我们的唯一希望就是根据数据项分区；我们只需在单一分区上操作，这比较接近“大结果集”的情况。

结果集和别的数据存在与否有关

Result Set Predicated on Absence of Data

一个表中的哪些记录和另一个表中的数据不匹配？这种“识别例外”的需求经常出现。人们最常想到的解决方案有两个：not in ()搭配非关联子查询，或者not exists()

搭配关联子查询。一般认为应该使用 not exists。在子查询出现在高效搜索条件之后，使用not exists是对的，因为高效过滤条件已清除大量无关数据，关联子查询当然会很高效。但当子查询恰好是唯一条件时，使用not in比较好。

查找在另一个表无对应数据的记录时，会碰到一些奇特的解决方案。以下为实际例子，显示哪些数据库查询代价最高。注意，问号是占位符（placeholder）或称为绑定变量（bind variable），它们的具体值在后续执行中传递给查询：

```
insert into ttmpout(custcode,
suistrcod,
cempdtcod,
bkgareacod,
mgtareacod,
risktyp,
riskflg,
usr,
seq,
country,
rating,
sigsecsui)
select distinct custcode,
?,
?,
?,
mgtareacod,
?,
?,
usr,
seq,
country,
rating,
sigsecsui
from ttmpout a
where a.seq = ?
and 0 = (select count(*)
from ttmpout b
where b.suistrcod = ?
and b.cempdtcod = ?
and b.bkgareacod = ?
and b.risktyp = ?
and b.riskflg = ?
and b.seq = ?)
```

此例并非暗示我们无条件地认可临时表的使用。另外，我怀疑这个insert语句会被循环执行，通

过消除循环可以适当改善性能。

例子中出现了自参照（self-reference）很不常见的用法：对一个表的插入操作，是以同一个表上的 select 为基础的。当前存在哪些记录？要创建的记录是否不存在？要插入的记录是由上述两个问题决定的。

使用 count(*) 测试某些数据是否存在是个糟糕的主意：为此DBMS 必须搜索并找出所有相符的记录。其实，此时应该使用 exists，它会在遇到第一个相符数据时就停止。当然，如果过滤条件是主键，使用count或exists的差别不大，否则差异极大——无论如何，从语义角度讲，若想表达：

and not exists (select 1 ...)

不能换成：

and 0 = (select count(*) ...)

使用count(*)时，优化器“可能”会进行合理的优化——但未必一定如此。记录的数量若通过独立步骤被计入某个变量，优化器肯定不会优化，因为优化器再聪明也无法猜测计数的用途：count()的结果可能是极重要的值，而且必须显示给最终用户！

然而，当我们只想建立一条新记录，且新记录要从已存在于表中的记录推导出来时，正确的做法是使用诸如except（有时称为 minus）这样的集合操作符（set operator）。

```
insert into ttmpout(custcode,
suistrcod,
cempdtcod,
bkgareacod,
mgtareacod,
risktyp,
riskflg,
usr,
seq,
country,
rating,
sigsecsui)
(select custcode,
?,
?,
?,
mgtareacod,
?,
?,
usr,
seq,

country,
rating,
```

```
sigsecsui
from ttmpout
where seq = ?
except
select custcode,
?,
?,
?,
mgtareacod,
?,
?,
usr,
seq,
country,
rating,
sigsecsui
from ttmpout
where suistrcod = ?
and cempdtcod = ?
and bkgareacod = ?
and risktyp = ?
and riskflg = ?
and seq = ?)
```

集合操作符的重大优点是彻底打破了“子查询强加的时间限制”，无论子查询是关联子查询还是非关联子查询。打破“时间限制”是什么意思？当存在关联子查询时，就必须执行外层查询，接着对所有通过过滤条件的记录，执行内层查询。外层查询和内层查询相互依赖，因为外层查询会把数据传递给内层查询。

使用非关联子查询时情况要好得多，但也不是完全乐观：必须先完成内层查询之后，外层查询才能介入。即使优化器选择把整个查询作为哈希连接（hash join）执行——这是聪明的方法——也不例外，因为要进行哈希连接，SQL 引擎必须先进行表扫描以建立哈希数组（hash array）。

相比之下，使用集合操作符union、intersect或except时，查询中的这些组成部分不会彼此依赖，从而不同部分的查询可以并行执行。当然，如果有个步骤非常慢，而其他步骤非常快，则并行化意义不大；另外，如果查询的两个部分工作完全相同，并行化就没有好处，因为不同进程的工作是重复的，而不是分工负责。一般而言，在最后步骤之前，让所有部分并行执行会很高效，最后步骤把不完整的结果集组合起来——这就是分而治之。

集合操作符的使用有个额外的问题：各部分查询必须返回兼容的字段 —— 字段的类型和数量都要相同。下例（实际案例，来自账单程序）通常不适合集合操作符：

```
select whatever, sum(d.tax)
from invoice_detail d,
```

```
invoice_extractor e
where (e.pga_status = 0
or e.rd_status = 0)
and suitable_join_condition
and (d.type_code in (3, 7, 2)
or (d.type_code = 4
and d.subtype_code not in
(select trans_code
from trans_description
where trans_category in (6, 7))))
group by what_is_required
having sum(d.tax) != 0
```

最后一个条件有问题（它使我想起了《绿野仙踪》里的黄砖路，甚至使我做起了“负税率”的白日梦）：

```
sum(d.tax) != 0
```

如前所述，换成下列条件更加合理：

```
and d.tax > 0
```

上述的例子中，使用集合操作符会相当笨拙，因为必须访问invoice_detail表好几次——如你所料，那不是个轻量级的表。当然，还要看每个条件的可选择性，如果 type_ code=4很少见，那么它就是个可选择性很高的条件，exists或许会比not in ()更适合。另外，如果trans_description正好是个小型表（或者相对较小），尝试通过单独操作测试存在性，并起不到改善性能的效果。

另一个表达非存在性的方法很有趣——而且通常相当高效——是使用外连接（outer join）。外连接的主要目的是，返回来自一个表的所有信息及连接表中的对应信息。无对应信息的记录也需返回——查找另一个表中无对应信息的数据时，这些记录正好是我们的兴趣所在，可通过检查连接表的字段值是否为null找出它们。

例如：

```
select whatever
from invoice_detail
where type_code = 4

and subtype_code not in
(select trans_code
from trans_description
where trans_category in (6, 7))
```

或重写为：

```
select whatever
from invoice_detail
outer join trans_description
on trans_description.trans_category in (6, 7)
and trans_description.trans_code = invoice_detail.subtype_code
```

where trans_description.trans_code is null
我故意在join子句中加上trans_category的条件。有人认为它应该出现在where 子句中，实际上，在连接之前或在连接之后过滤都不影响结果（当然，根据这个条件和连接条件本身的可选择性不同，会有不同的性能表现）。然而，在使用空值上的条件时，我们别无选择，只有在连接后才能做检查。

外连接有时需要加 distinct。实际上，通过外连接或not in()非关联子查询，来检查数据是否存在的差异很小，因为连接所使用的字段，正好与比较子查询结果集的字段完全相同。不过，众所周知的是，SQL 语言的“查询表达式风格”对“执行模式”影响很大，尽管理论上不是这么说的。这取决于优化器的复杂程度，以及它是否会以类似方法处理这两类查询。换言之，SQL 不是真正的声明性语言（SQL is not a truly declarative language），尽管优化器不断推陈出新改善SQL的可靠性（reliability）。

最后提醒一下，应密切注意null，这个舞会扫兴者（party-poopers）经常出现。虽然在in ()子查询中，null与大量非空值连接不会对外层查询造成影响，但在使用not in ()子查询时，由内层查询返回的null会造成not in()条件不成立。要确保子查询不会返回null并不需要太高的代价，而且这么做可以避免许多灾难。
总结：数据集可以通过各种技巧进行比较，但一般而言，使用外连接和子集合操作符更高效。

当前值

Current Values

当我们只对最近或当前值感兴趣时，如何避免使用嵌套子查询或 OLAP 函数（两者都引起排序）而直接找到适当值，是非常吸引人的设计。如第1章所述，解决该问题的方法之一，就是把每个值与某个“截止日期”相关联 —— 就像麦片外盒上的“保质期（best before）”一样——并让当前值的“截止日期”是遥远的未来（例如公元 2999 年 12 月 31 日）。这种设计存在一些与实际相关的问题，下面讨论这些问题。

使用“固定日期”，确定当前值变得非常容易。查询如下所示：

```
select whatever
from hist_data
where item_id = somevalue
and record_date = fixed_date_in_the future
```

接着，通过主键找到正确的记录。（当然，要参照的日期如果不是当前日期，就必须使用子查询或 OLAP 函数了。）然而，这种方法有两个主要缺点。

较明显的缺点：插入新的历史数据之前，先要更新“当前值”（例如今天），接着，将最新“当前值”和历史数据一起插入表中。这个过程导致工作量加倍。更糟的是，关系理论中的主键用于识别记录，但具有唯一性的(item_id, record_date)却不能作为主键，因为我们会对它做“部分更新（partially update）”。因此，必须有一个能让外键参照的代理键（ID字段或序列号），结果程序变得更加复杂。大型历史表的麻烦就是，通常它们也经历过高频率的数据插入，所以数据量才

会这么大。快速查询的好处，能抵销缓慢插入的缺点吗？这很难说，但绝对是个值得考虑的问题。

还有个微妙的缺点与优化器有关。优化器使用各种详细程度不同的统计数据，检查字段的最低值和最高值，尝试评估值的分布情况。假设历史表包含了自 2000 年 1 月 1 日开始的历史数据。于是，我们的数据组成是“散布在几年间的99.9% 的历史数据”加上“2999 年 12 月 31 日的0.1% 的‘当前数据’”。因此，优化器会认为数据散布在一千年的范围内。优化器在数据范围上的偏差是由于查询中出现的上限日期的误导（即“and record_date = fixed_date_in_the future”）。此时的问题就是，如果你当查询的不是当前值（例如，你要统计不同时段的数据变化），优化器可能错误地做出“使用索引”的决定——因为你访问的只是千年中的极小部分——但实际上需要的是对数据进行扫描。是优化器的评估偏差导致它做出完全错误的执行计划决定，这很难修正。

总结：要理解优化器如何看待你的系统，就必须理解你的数据和数据分布方式。

通过聚合获得结果集

Result Set Obtained by Aggregation

本节讨论一类极常见的情况：对一个或多个主表（main table）中的详细数据进行汇总，动态计算出结果集。换言之，我们面临数据聚合（aggregation of data）的问题。此时，结果集大小取决于group by的字段的基数，而不是查询条件的精确性。正如第一节“小结果集，直接条件”中所述，对表进行一趟（a single pass）处理获得的并非真正聚合的结果（否则就需要自连接和多次处理），但此时聚合函数（或聚合）也相当有用。实际上，最让人感兴趣的SQL聚合使用技巧，不是明显需要sum或avg的情况，而是如何将过程性处理转化为以聚合为基础的纯 SQL替代方案。

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

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

注意：

聚合函数非常有用，可以解决不少SQL问题（第11章会再次讨论）。然而，我发现开发者通常只使用最平常的聚合函数count()，它对大多数程序是否真的有用值得怀疑。
第2章说明了使用count(*)判定是否要更新记录（插入新记录）是很浪费的。你可能在报表中误

用了count(*)。测试存在性有时会以模仿布尔值的方式实现：

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

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

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

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

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

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

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

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

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

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

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

索引：

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

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

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

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

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

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

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

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

因此，查询可写成：

```
select shipment_id
```

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

甚至可以写得更简洁：

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

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

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

这个方法利用了“Y”大于“N”，而没有考虑标志转换为数值。本方法更高效。

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

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

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

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

必须指出：对视图（即聚合的结果）应用条件时，如果优化器不够聪明，没有在聚合前再次注入过滤条件，我们就会遇到完全相同的问题。
有些过滤条件生效太晚，应该提前，可做如下修改：

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

在这个查询中，以下 having 的条件乍看起来相当合理：

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

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

```
where amount > 0
```

此例中，group by的使用分两种情况。首先：

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

我们注意到，group by对聚合计算是不必要的，可以用 distinct 取代它，并执行相同的排序和消除重复项目的工作：

```
select distinct customer_id
from orders
where order_date < add_months(sysdate, -1)
and amount > 0
```

把条件放在 wher e 子句中，能让多余的记录尽早被过滤掉，因而更高效。

总结：聚合操作的数据应尽量少。