

内容简介

本书荣获 2009 年 Jolt 图书大奖，是不可多得的分享 MySQL 实用经验的图书。它不但可以帮助 MySQL 初学者提高使用技巧，更为有经验的 MySQL DBA 指出了开发高性能 MySQL 应用的途径。全书包含 14 章和 4 个附录，内容覆盖 MySQL 系统架构、设计应用技巧、SQL 语句优化、服务器性能调优、系统配置管理和安全设置、监控分析，以及复制、扩展和备份 / 还原等主题，每一章的内容自成体系，适合各领域技术人员作选择性的阅读。

978-0-596-10171-8 High Performance MySQL, Second Edition © 2008 by O'Reilly Media, Inc. Simplified Chinese edition, jointly published by O'Reilly Media, Inc. and Publishing House of Electronics Industry, 2010. Authorized translation of the English edition, 2008 O'Reilly Media, Inc., the owner of all rights to publish and sell the same. All rights reserved including the rights of reproduction in whole or in part in any form.

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

版权贸易合同登记号 图字：01-2008-4026

图书在版编目 (CIP) 数据

高性能 MySQL：第 2 版 / 施瓦茨 (Schwartz, B.) 等著；王小东，李军，康建勋译. —北京：电子工业出版社，2010.1
书名原文 : High Performance MySQL, 2nd Edition
ISBN 978-7-121-10245-5

I. 高… II. ①施…②王…③李…④康… III. 关系数据库 - 数据库管理系统, MySQL IV. TP311.138

中国版本图书馆 CIP 数据核字 (2010) 第 010252 号

策划编辑：徐定翔

责任编辑：周 筠

项目管理：梁 晶

封面设计：Karen Montgomery，张 健

印 刷：北京天宇星印刷厂

装 订：三河市皇庄路通装订厂

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：860 × 1092 1/16 印张：35 字数：950千字

印 次：2010年1月第1次印刷

定 价：99.00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zts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：(010) 88258888。

O'REILLY®

高性能MySQL (第二版)

High Performance MySQL, 2nd Edition



Baron Schwartz Peter Zaitsev
Vadim Tkachenko Jeremy D.Zawodny 著
Arjen Lentz Derek J.Balling

王小东 李军 康建勋 译

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

目录

Contents

推荐序.....	I
前言.....	III
第 1 章 MySQL 架构.....	1
1.1 MySQL 的逻辑架构	1
1.2 并发控制	3
1.3 事务	4
1.4 多版本并发控制	10
1.5 MySQL 的存储引擎	11
第 2 章 寻找瓶颈：基准测试与性能分析	25
2.1 为什么要进行基准测试	25
2.2 基准测试策略	26
2.3 基准测试工具	32
2.4 基准测试样例	34
2.5 性能分析（Profiling）	43
2.6 分析操作系统	60
第 3 章 架构优化和索引	63
3.1 选择优化的数据类型	63
3.2 索引基础知识	74
3.3 高性能索引策略	83
3.4 索引实例研究	102
3.5 索引和表维护	105
3.6 正则化和非正则化	108
3.7 加速 ALTER TABLE	113
3.8 对存储引擎的说明	115
第 4 章 查询性能优化	118
4.1 基本原则：优化数据访问.....	118
4.2 重构查询的方式	122
4.3 查询执行基础知识	124

4.4	MySQL 查询优化器的限制	139
4.5	优化特定类型的查询	146
4.6	查询优化提示	151
4.7	用户定义变量	154
第 5 章	MySQL 高级特性	159
5.1	MySQL 查询缓存	159
5.2	在 MySQL 中存储代码	168
5.3	游标	173
5.4	准备语句	174
5.5	用户自定义函数	177
5.6	视图	179
5.7	字符集和排序规则	182
5.8	全文搜索	188
5.9	外键约束	194
5.10	合并表和分区	194
5.11	分布式 (XA) 事务	201
第 6 章	优化服务器设置	203
6.1	配置基础知识	203
6.2	通用调优原则	207
6.3	MySQL I/O 调优	214
6.4	MySQL 并发调优	224
6.5	基于工作负载调优	226
6.6	每连接 (Per-Connection) 设置调优	231
第 7 章	操作系统和硬件优化	232
7.1	什么限制了 MySQL 的性能	232
7.2	如何为 MySQL 选择 CPU	233
7.3	平衡内存和磁盘资源	235
7.4	为从服务器选择硬件	240
7.5	RAID 性能优化	240
7.6	存储区域网络和网络附加存储	246
7.7	使用多个磁盘卷	247
7.8	网络配置	248
7.9	选择操作系统	250
7.10	选择文件系统	250
7.11	线程处理	252
7.12	交换	252
7.13	操作系统状态	254

第 8 章	复制	259
8.1	复制概述	259
8.2	创建复制	262
8.3	揭示复制的真相	268
8.4	复制拓扑	273
8.5	复制和容量规划	284
8.6	复制管理和维护	285
8.7	复制问题和解决方案	292
8.8	复制有多快	305
8.9	MySQL 复制的未来	307
第 9 章	伸缩性与高可用性	308
9.1	术语	308
9.2	MySQL 的伸缩性	310
9.3	负载均衡	328
9.4	高可用性	336
第 10 章	应用层面的优化	344
10.1	应用程序性能概述	344
10.2	Web 服务器的议题	346
10.3	缓存	349
10.4	扩展 MySQL	354
10.5	可替代的 MySQL	354
第 11 章	备份与还原	356
11.1	概况	356
11.2	要权衡的事项	360
11.3	管理和备份二进制日志	367
11.4	数据备份	369
11.5	从备份中还原	377
11.6	备份和还原的速度	386
11.7	备份工具	387
11.8	脚本化备份	392
第 12 章	安全	395
12.1	术语	395
12.2	账号的基本知识	396
12.3	操作系统安全	411
12.4	网络安全	412
12.5	数据加密	418
12.6	在 Chroot 环境里使用 MySQL	421

第 13 章	MySQL 服务器的状态	423
13.1	系统变量	423
13.2	SHOW STATUS	423
13.3	SHOW INNODB STATUS	429
13.4	SHOW PROCESSLIST	440
13.5	SHOW MUTEX STATUS	441
13.6	复制的状态	442
13.7	INFORMATION_SCHEMA	442
第 14 章	用于高性能 MySQL 的工具	444
14.1	带界面的工具	444
14.2	监控工具	446
14.3	分析工具	453
14.4	MySQL 的辅助工具	455
14.5	更多的信息来源	458
附录 A	大文件传输	459
附录 B	使用 EXPLAIN	463
附录 C	在 MySQL 里使用 Sphinx	476
附录 D	锁的调试	497
索引	505

本书豆瓣主页：<http://www.douban.com/subject/4241826/>

互动网热卖链接：<http://www.china-pub.com/196341>

推荐序

Foreword

我认识 Peter、Vadim 和 Arjen 已经有很长一段时间，见证了他们长久以来在自己项目上使用 MySQL 和为各类高标准客户调优 MySQL 服务器的历史。另一方面，Baron 为增强 MySQL 的功能编写了许多客户端软件。

作者们的专业背景清晰地反映在了彻底重写《High Performance MySQL: Optimizations, Replication, Backups, and More》第二版的工作里。这本书不只告诉你如何优化工作，从而能比以前更好地使用 MySQL，作者们还做了大量额外的工作，亲自编制执行基准测试，并将结果发布出来以佐证他们的观点。这些信息让读者可以借此获悉许多很有价值的 MySQL 内部工作机制——这在其他书中是难以得到的；同样，这些信息也能帮助读者避开那些在将来会引发糟糕性能的错误。

我不但要向刚刚接触 MySQL 服务器，正准备编写第一个 MySQL 应用的初学者推荐这本书，还要向富有经验的用户推荐这本书，他们已经对基于 MySQL 的应用作过一些调优的工作，现在正需要在这个方向上再前进“一小步”。

——Michael Widenius

2008 年 3 月

前言

Preface

对于这本书，我们在头脑里有好几个目标。其中的大多数源于我们一直想要有一本在书架上寻找却总是找不到的神话般完美的 MySQL 书，其他几个目标来自于想把我们的经验分享给那些把 MySQL 用在他们环境中的用户。

我们不想让这本书只是一本 SQL 入门书，不想让这本书的书名随意使用一些时限词语来开始或结尾（例如“……只需 30 天”，“7 天内提高……”），也不想说服读者什么。最主要的，我们希望这本书能帮助你把技能提高一个层次，用 MySQL 构建出快速、可用的系统——它能解答类似这样的问题：“我怎样才能搭建起一个 MySQL 服务器集群，它能处理数以百万计的请求，哪怕有几台服务器宕机时，它仍然能正常提供服务？”

我们编写本书的着眼点不仅在于迎合 MySQL 应用开发人员的需求，还在于满足 MySQL 管理员的严格要求，管理员需要不管开发人员和用户怎么折腾，服务器都能挂在线上正常运行。如前所述，我们假定你已经具备了一些 MySQL 的相关经验，比较理想的就是你已经读过一本 MySQL 方面的入门书。我们同样也假定你具备一些常用的系统管理、网络和 Unix 风格操作系统等方面的经验。

经过修订、扩充后的第二版对于第一版里的所有主题都作了更深入的讲解，并增加了一些新的主题。这也部分地反映了自本书首次出版之后，MySQL 世界发生的一些变化：MySQL 现在已经成为软件中更大更复杂的一部分。如同其重要性一样，它的普及度也提高了：MySQL 社区变得更加庞大，更多的大企业把 MySQL 应用到他们的关键业务系统中。自本书第一版发布以后，MySQL 已被广泛认同可作为企业级应用（注 1）。人们也越来越多地把 MySQL 用在互联网应用上，这些应用若发生故障和其他问题都无法被掩饰过去，也不能被容忍。

作为我们努力的结果，第二版的内容着重点跟第一版略有不同。我们会像强调性能一样，强调可用性和准确性，这部分由于我们自己也把 MySQL 用在那些运作着巨大金额的业务系统里。我们对 Web 应用也有着切身体验，MySQL 在这方面正变得越来越普及。第二版里会谈论到在 MySQL 周边扩展的世界，而这个世界在第一版编写时还不存在。

本书是如何组织的

How This Book Is Organized

我们把许多复杂的主题放在一本书里，所以，在这里我们要解释一下它们的编排次序，使读者能更易于学习它们。

注 1：我们觉得这段话更像是市场营销的说辞，但是，它大概传达了这样一个意思：MySQL 对于许多人而言显得很重要了。

内容广泛的概述

A Broad Overview

第 1 章，MySQL 架构，用于讲述基础知识——这些知识在你做更深入挖掘之前必须加以熟悉。你需要在高效利用 MySQL 前理解整个框架是如何被组织起来的。这一章解释了 MySQL 的架构和它存储引擎的关键方面。如果你还不熟悉关系数据库的一些基本概念及事务，它就能帮你更快地进入角色。如果本书就是你的 MySQL 入门书，这一章也非常有用，不过，你最好还是事先熟悉了另外一种数据库，例如 Oracle。

构建一个坚实的基础

Building a Solid Foundation

接下来的 4 章涉及了你在使用 MySQL 时会几次三番来查阅的资料。

第 2 章，寻找瓶颈：基准测试与性能分析，讨论了基准测试和获取系统概况的基础。它们决定了你的系统能处理哪一类的工作负荷、执行某些任务时它能运行得多快等。你会希望在做重要更改的前后都能对你的应用做一次基准测试，这样就可以判断出这些更改产生了多大的效果。有些看似正面的更改在真实世界的负载压力下可能会变成负面的影响，除非你能精确地对其进行衡量，否则你是永远都不会知道到底是什么导致糟糕的系统性能。

第 3 章，架构优化和索引，我们会介绍各数据类型的细微差别、表的设计和索引的创建。一个设计良好的数据库能有助于 MySQL 获得更佳的性能表现。在接下来的那些章节里，我们讲到的很多内容的关键点都在于你的应用是怎么使用 MySQL 索引的。深刻认识索引及如何巧妙地运用它们是高效使用 MySQL 的核心所在，所以，你可能会经常地回过来重新阅读这一章。

第 4 章，查询性能优化，解释了 MySQL 是怎样执行查询的，以及怎么才能利用查询优化器的能力。深入领会查询优化器的工作方法能帮你在编写查询时创造奇迹，也能帮你更好地理解索引（索引和查询优化的次序就像“先有鸡还是先有蛋”的问题，读完第 4 章后再回过去读第 3 章可能会让你受益匪浅。）该一章里还特别展示了那些常见的查询示例，用来说明 MySQL 所擅长的是哪方面的工作，怎么把查询转换成能够利用查询优化器强大能力的形式。

为了更好地做到这一点，我们已经讲述过对任何数据库都适用的一些基本概念：表、索引、数据和查询。第 5 章，MySQL 高级特性，将在上述基础之上再进一步，向你展示 MySQL 内部那些更高层次的框架是如何运作的。我们会介绍查询缓存、存储过程、触发器、字符集等内容。MySQL 实现这些功能特性的方法跟其他数据库有点不一样，因此，对这些特性的深入理解能够帮你创造一个性能优化的新机会，这在以前你是不会想到的。

调优你的应用

Tuning Your Application

接下来的两章讨论如何修改你的基于 MySQL 的应用，使它能在性能上得到提升。

第 6 章，优化服务器设置，我们讨论的是如何调优 MySQL，使它能在最大程度上让硬件特性为你的特定应用服务。第 7 章，操作系统和硬件优化，我们解释了如何充分利用你的操作系统和硬件配置，同时为大规模应用提供了某些能提高性能的硬件配置建议。

配置更改之后的向上扩展

Scaling Upward After Making Changes

一台服务器往往是不够用的。第 8 章，复制，介绍如何把将数据自动地复制到多台服务器上。第 9 章，“伸缩性与高可用性”，讲述如何将伸缩性、负载均衡和高可用性综合起来运用，为应用伸展到你所需要的程度提供基础性工作。

当应用运行在一个大规模的 MySQL 后端之上时，它本身就蕴含了意义非凡的优化机会。设计一个大型应用有更好的途径，也有更坏的途径，但这不是本书的着重点，我们不希望你把所有的时间都专注于 MySQL 之上。第 10 章，“应用层面的优化”，帮助你发现那些悬挂在靠近地面枝头上的柿子，特别是对于 Web 应用。

增强应用的可靠性

Making Your Application Reliable

哪怕是世界上设计得最好、伸缩性最强的架构，如果它不能在掉电、恶意攻击、程序 Bug、程序员的过失，以及其他灾难中幸存下来，那它也算不上是好的架构。

第 11 章，“备份和还原”，我们会讨论到不同的 MySQL 数据库备份和还原策略。这些策略都有助于在系统遭受到不可避免的硬件错误时最小化故障停机时间，遭遇到各种灾难时确保你的数据安全。

第 12 章，“安全”，能让你对运行 MySQL 服务器涉及的安全因素有深入的认识。最重要的是，我们给你提供了很多建议，防止来自外部的攻击威胁你苦心优化、配置过的服务器。我们还会指出几个很少见的暴露出数据库安全问题的地方，并展示不同的实施方法的好处及对性能的影响。通常情况下，就性能方面而言，保持安全策略简单化是值得的。

其他有用的主题

Miscellaneous Useful Topics

最后的几个章节和附录里，我们深入研究了几个既不“适合”放入前面任何一个章节中，又被多个章节反复引用的内容，它们值得特别关注。

第 13 章，“MySQL 服务器的状态”，展示的是如何检查 MySQL 服务器运行情况。知道如何获取服务器的状态信息很重要，知道那些信息包含的意思更加重要。我们针对 SHOW INNODB STATUS 作了特别具体的讲解，它能提供关于 InnoDB 事务存储引擎的更深层次的操作信息。

第 14 章，“用于高性能的 MySQL 工具”，介绍了一些能帮你更有效管理 MySQL 的工具。这些工具包括监控和分析工具，以及能帮你编写查询语句的工具等。其中提到的 Maatkit 是由 Baron 创建的，它能够增强 MySQL 的功能性，使你的数据库管理员的生活更加轻松。在该章里也演示了一个名叫 innotop 的程序，这个程序是 Baron 写的，其目的是提供一个易于使用的查看 MySQL 正在做什么的用户接口，它的功能与 Unix 的 top 命令类似。在调优 MySQL 各阶段里，你若要监控 MySQL 和它的存储引擎里发生的情况，它就是一个很有价值的工具。

附录 A，“大文件传输”，展示如何高效地把很大的文件从一个地方复制到另一个地方——这在大数据量管理时肯定会用到。附录 B，“使用 EXPLAIN”，展示如何真正理解和使用那个重要的 EXPLAIN 命令。附录 C，“在 MySQL 里使用 Sphinx”是对 Sphinx 的一个介绍，这个高性能全文索引系统是对 MySQL 自有功能的一个补充。

最后的附录 D，“锁的调试”，向你展示的是当几个查询在请求锁时相互妨碍时，该如何去破译其中的缘由。

软件的版本和有效性

Software Versions and Availability

MySQL 就像个移动的目标。在 Jeremy 写出第一版提纲之后的几年里，有大量的不同版本 MySQL 发布出来。在本书第一版发行的时候，MySQL 4.1 和 5.0 还都是 alpha 版，但如今它们已经作为正式产品很多年了，并成为今天许多大型在线应用的后台支撑。当我们完成本书第二版时，MySQL 5.1 和 6.0 也处于这样的边缘。（MySQL 5.1 是 candidate 版，而 6.0 是 alpha 版。）

在本书里，我们没有依赖于某个特定的 MySQL 版本。相反，我们讲述的是基于真实世界里各版本 MySQL 的更广阔的知识。本书涉及的核心版本是 MySQL 5.0，因为我们把它看作是“当前”版本。书中的大多数示例都假定你运行的是 MySQL 5.0 的某个相对比较稳定的版本，例如 MySQL 5.0.40 或更新的。我们会特意标注出哪些框架或功能在那些老版本里不存在，或者会出现在即将到来的 5.1 版本系列里。然而，明确的功能特性与版本的对应关系只有在 MySQL 的文档里才能找到，所以，我们希望你阅读本书的时候能够经常访问带有注解的在线文档（<http://dev.mysql.com/doc/>）。

MySQL 另一个伟大之处在于它能运行在当今所有流行的平台上：Mac OS X、Windows、GNU/Linux、Solaris、FreeBSD，只要你能想到的都行！但是，我们偏向于 GNU/Linux（注 2）和其他 Unix 风格的操作系统。Windows 用户可能会有所不同，例如，文件路径就会完全不一样。书中也会用到一些标准的 Unix 命令行功能，我们假定你知道它们在 Windows 上的对应命令。（注 3）

Perl 也是 MySQL 在 Windows 上运行时的麻烦之一。MySQL 自带的几个很有用的辅助功能都是用 Perl 写的，本书某些章节里展示的 Perl 脚本就是构建更复杂工具的基础，像 Maatkit 也是用 Perl 写的。可是，Perl 并没有包含在 Windows 里。为了能使用这些脚本，你需要访问 ActiveState 下载一个 Windows 版的 Perl，然后安装一个必需的插件模块（DBI 和 DBD::mysql）好让 MySQL 能够访问到它。

本书使用的书写约定

Conventions Used in This Book

本书使用了以下这些书写约定：

等宽字体（Constant width）

用于表示代码、配置选项、数据库和表名、变量和它们的值、函数、模块、文件内容，以及命令的输出结果。

等宽粗体（Constant width Bold）

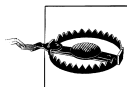
用于表示命令或要用户自己输入的内容，也用于强调命令的输出结果里的某些内容。

注 2：为了避免混淆，当我们写到关于内核的内容时，就以 Linux 称呼；当我们讲到支持应用的整个操作系统架构时，就以 GNU/Linux 称呼。

注 3：你可以在 <http://unxutils.sourceforge.net> 或 <http://gnuwin32.sourceforge.net> 下载到与 Windows 兼容的 Unix 辅助工具。



这个图标表示提示、建议或一般性注解。



这个图标表示的是提醒或警告。

使用本书示例代码

Using Code Examples

本书的目的是帮你把事情做好。一般来说，你无需特地联系我们就可以在你的程序和文档里任意使用本书的代码，除非你要把其中的关键代码以你的名义重新发布。举例来说，你的程序中使用到了书中的几段代码，不需要获得许可；出售或发布 O'Reilly 的随书光盘，需要获得许可；引用本书内容和示例代码去解答一个问题，不需要获得许可；把本书中大量代码合并到你的产品文档里时，需要获得许可。

本书的示例代码在 <http://www.highperformmysql.com> 上可以获取到，并经常会有更新。但是，我们不能保证会为所有次要版本的 MySQL 更新和测试这些代码。

我们会感谢，但是不要求写上代码所属权的声明。这个所有权声明一般包括书名、作者、出版商和 ISBN，例如“High Performance MySQL: Optimization, Backups, Replication, and More, Second Edition, by Baron Schwartz et al. Copyright 2008 O'Reilly Media, Inc., 9780596101718.”

如果你觉得你对示例代码的使用超过了正当使用范围或如上所述的授权使用的范围了，请跟我们联系：permissions@oreilly.com。

如何联系我们

How to Contact Us

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。

奥莱理软件（北京）有限公司
北京市 西城区 西直门 南大街2号 成铭大厦C座807室
邮政编码：100080
网页：<http://www.oreilly.com.cn>
E-mail：info@mail.oreilly.com.cn

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

与本书有关的在线信息如下所示。

<http://www.oreilly.com/catalog/9780596101718> (原书)

<http://www.oreilly.com.cn/book.php?bn=978-7-121-10245-5> (中文版)

北京博文视点资讯有限公司 (武汉分部)

湖北省 武汉市 洪山区 吴家湾 邮科院路特1号 湖北信息产业科技大厦1402室

邮政编码: 430074

电话: (027)87690813 传真: (027)87690813转817

读者服务网页: <http://bv.csdn.net>

E-mail:

reader@broadview.com.cn (读者信箱)

bvtougao@gmail.com (投稿信箱)

本书第二版的致谢

Acknowledgments for the Second Edition

Sphinx 的开发者 Andrew Aksyonoff 编写了附录 C, “在 MySQL 里使用 Sphinx”。我们非常感谢他首次对此进行深入讨论。

在本书编写的时候, 我们也得到了来自于许多人的无私帮助, 在这里我们不可能把他们都一一列举出来——我们真地非常感谢 MySQL 社区和 MySQL AB 公司的每一个人。下面是对本书作出了直接贡献的人, 如果我们遗漏了任何一个人, 还请原谅, 他们是: Tobias Asplund、Igor Babaev、Pascal Borghino、Roland Bouman、Ronald Bradford、Mark Callaghan、Jeremy Cole、Britt Crawford 和他的 HiveDB 项目、Vasil Dimov、Harrison Fisk、Florian Haas、Dmitri Joukovski、Zmanda (感谢他为解释 LVM 快照配上的图表)、Alan Kasindorf、Sheeri Kritzer Cabral、Marko Makela、Giuseppe Maxia、Paul McCullagh、B. Keith Murphy、Dhiren Patel、Sergey Petrunia、Alexander Rubin、Paul Tuckfield、Heikki Tuuri 和 Michael “Monty” Widenius。

有一份特别的感谢要送给 O'Reilly 的编辑 Andy Oram 和助理编辑 Isabel Kunkle, 以及审稿人 Rachel Wheeler, 同时也感谢 O'Reilly 团队里的其他成员。

来自 Baron

From Baron

我要感谢我的妻子 Lynn Rainville 和我们的狗狗 Carbo。如果你也曾写过一本书, 我确信你可体会到我有多么感谢他们。我也非常感谢 Alan Rimm-Kaufman 和我在 Rimm-Kaufman 集团的同事们, 在写书的过程中, 他们给了我支持和鼓励。我要感谢 Peter、Vadim 和 Arjen, 是他们给了我这个机会让梦想成真。最后, 我要感谢 Jeremy 和 Derek 为我们开了个好头。

来自 Peter

From Peter

我从事 MySQL 性能和伸缩性方面的讲演、培训和咨询已经很多年了，我一直想把它们扩大到更多的受众，因此，当 Andy Oram 邀请我加入本书编写中时，我感到非常兴奋。此前我没写过书，所以，我对所需要的时间和精力都毫无把握。我们先谈到只对第一版做一些更新，以跟上 MySQL 最近的版本升级，但我们想把很多新素材加到书里去，结果就几乎重写了整本书。

这本书是真正的团队合作的结晶。因为我忙于 Percona 的事情——我和 Vadim 的咨询公司，又因为英语并非我的第一语言，所以我们有不同的角色。我负责提供大纲和技术性内容，然后，我把素材都过一遍，在写作的时候再对它进行修订和扩展。当 Arjen（MySQL 文档团队的前任负责人）加入之后，我们就开始勾划出整个提纲。在 Baron 到来后，一切才真正开动起来，他能够以不可思议的速度编写出高质量的内容。Vadim 在深入检查 MySQL 源代码和提供基准测试或其他探索来巩固我们的论点时发挥了很大的作用。

当我们编写这本书时，我们发现越来越多的领域需要刨根问底。本书主题里的大多数，例如复制、查询优化、InnoDB、架构和设计都可以分别轻易地写成一本书，因此，我们不得不在某一个阶段时停止，把余下的材料用在将来可能要出的新版上、我们的博客上、我们的讲演上，以及我们的技术文章里。

本书的评审者给予了我们巨大的帮助，无论是来自 MySQL AB 公司内部的还是外部的，他们都是这个世界上最顶级的 MySQL 专家，他们包括 MySQL 的创建者 Michael Widenius、InnoDB 的创建者 Heikki Tuuri、MySQL 优化器团队的负责人 Igor Babaev，以及其他人士。

我还要感谢我的妻子 Katya Zaytseva、我的孩子 Ivan 和 Nadezhda，他们容许了我把家庭时间花在本书写作上。我也要感谢 Percona 的雇员们，当我在公司里“人间蒸发”去写书的时候，是他们处理了日常的事务。当然，我也要感谢 O'Reilly 和 Andy Oram 让这一切成为了可能。

来自 Vadim

From Vadim

我要感谢 Peter，能在本书中与他合作，我感到十分开心，期望在其他项目中能继续共事；我也要感谢 Baron，他在本书写作过程中起了很大的作用；还有 Arjen，跟他一起工作非常好玩。我还要感谢我们的编辑 Andy Oram，他抱着十二万分的耐心与我们一起工作。还要感谢 MySQL 团队，是他们创造了这个伟大的软件；我还要感谢我们的客户给予我调优 MySQL 的机会。最后，我要特别感谢我的妻子 Valerie 及我们的儿子 Myroslav 和 Timur，他们一直支持我，帮助我一步步前进。

来自 Arjen

From Arjen

我要感谢 Andy 的睿智、指导和耐心，感谢 Baron 中途加入到我们当中来，感谢 Peter 和 Vadim 坚实的背景信息和基准测试。也要感谢 Jeremy 和 Derek 在第一版里打下的基础，在我的书上，Derek 题写着：“要诚实——这就是我所有的要求。”

我也要感谢所有我在 MySQL AB 时的同事们，在那里我获得了关于本书主题的大多数知识。在此，我还要特别提到 Monty，我一直认为他是令人自豪的 MySQL 之父，尽管他的公司如今已成为 SUN 公司的一部分。我要感

谢全球 MySQL 社区里的每一个人。

最后但同样重要的是，我要感谢我的女儿 Phoebe，在她尚年少的生活舞台上，不用关心什么叫 MySQL，也不用考虑 Wiggles 所指的到底是何物。从某些方面来讲，无知就是福，它能给予我们一个全新的视角来看清生命中真正重要的是什么。对于读者，祝愿你们的书架上又增添了一本有用的书，还有，不要忘记你的生活。

本书第一版的致谢

Acknowledgments for the First Edition

像这样一本书的写成离不开许许多多人的帮助。没有他们的无私援助，你手上的这本书可能仍然是我们显示器屏幕四周的那一堆小贴纸。这是本书的一部分，在这里，我们可以感谢每一个曾经帮我们脱离困境的人，而无须担心突然奏响的背景音乐催促我们闭上嘴巴赶快走掉——如同你在电视里看到的颁奖晚会那样。

如果没有编辑 Andy Oram 坚决的督促、请求、央求和支持，我们就无法完成这个项目。如果要找出本书最负责一个人，那就是 Andy。我们真地非常感激每周一次的唠唠叨叨的会议。

其实，Andy 也不是孤独的，在 O'Reilly 里，还有一批人参与了把那些小贴纸转换成一本已装订好的你正要阅读的图书的工作，所以，我们也要感谢那些在生产、插画和销售环节的人们，感谢你们把本书合在一起。当然，还要感谢 Tim O'Reilly，是他持久不变的承诺为广大的开源软件出版了一批行业里最好的文档。

最后，我们要把感谢给予那些同意审阅本书不同阶段版本，并告诉我们哪里有错误的人们：我们的评审者。他们把 2003 年假期的一部分时间用在了审阅这些格式粗糙，充满了打字符号、误导性的语句和彻底的数学错误的文本上。我们要感谢（排名不分先后次序）：Brian “Krow” Aker、Mark “JDBC” Matthews、Jeremy “the other Jeremy” Cole、Mike “VBMySQL.com” Hillyer、Raymond “Rainman” De Roo、Jeffrey “Regex Master” Friedl、Jason DeHaan、Dan Nelson、Steve “Unix Wiz” Friedl，最后还有 Kasia “Unix Girl” Trapszo。

来自 Jeremy

From Jeremy

我要再次感谢 Andy，是他同意接纳这个项目，并持续不断鞭策我们加入更多的章节内容。Derek 的帮助非常关键，本书最后的 20%~30% 内容都是他来完成的，这使得我们不再错失下一个目标日期。感谢他同意中途加入进来，代替我只能零星爆发一下的生产力，完成了关于 XML 的繁琐工作、第 10 章、附录 C，以及我丢给他的其他那些活儿。

我也要感谢我的父母，在多年以前他们就给我买了 Commodore 64 电脑，他们不仅在前 10 年里容忍了我那如同一辈子漫长的对电子和计算机技术的沉迷，在之后还成为我不懈学习和探索的支持者。

接下来，我要感谢在过去几年里在 Yahoo! 推广 MySQL 信仰时遇到的那一群人，跟他们共事，我感到非常愉快。在本书的筹备阶段，Jeffrey Friedl 和 Ray Goldberger 给了我鼓励和反馈意见。在他们之后就是 Steve Morris、James Harvey 和 Sergey Kolychev 容忍了我在 Yahoo! Finance MySQL 服务器上做着看似固定不变的实验，即使打扰到了他们的重要工作。我也要感谢 Yahoo! 的其他成员，是他们帮我发现了 MySQL 上的那些有趣的问题和解决方法。还有，最重要地是要感谢他们对我有足够的信任和信念，让我把 MySQL 用在 Yahoo!'s 业务的重要和可见的那一部分上。

Adam Goodman，出版家和 Linux Magazine 的所有者，他帮助我轻装上阵开始为技术受众撰写文章，并在 2001 年后半年第一次出版了我的长篇 MySQL 文章。自那以后，他教授给我更多他所能认识到的关于编辑和出版的技能，还鼓励我通过在杂志上开设月度专栏在这条路上继续走下去。谢谢你，Adam。

我要感谢 Monty 和 David 与这个世界分享 MySQL。说到 MySQL AB，也要感谢在那里的其他伟大的人们，是他们鼓励我写成这本书：Kerry、Larry、Joe、Marten、Brian、Paul、Jeremy、Mark、Harrison、Matt 和团队的其他那些人。他们真的非常棒！

我要感谢我 Weblog 的所有读者，是他们鼓励我撰写基于日常工作的非正式的 MySQL 及其他技术文章。最后但同样重要的是，感谢 Goon Squad。

来自 Derek

From Derek

就像 Jeremy 一样，因为太多相同的原因，我也要感谢我的家庭。我要感谢我的父母，是他们不停地鼓动我去写一本书，哪怕他们头脑中都没任何跟它相关的东西。我的祖父母给我上了两堂很有价值的课：美金的含义，以及我跟电脑相爱有多深，他们还借钱给我去购买了我平生第一台电脑：Commodore VIC-20。

我万分感谢 Jeremy 邀请我加入他那旋风般的写作过山车（bookwriting roller coaster）中来。这是一个很棒的体验，我希望将来还能跟他一起工作。

我要特别感谢 Raymond De Roo、Brian Wohlgemuth、David Calafrancesco、Tera Doty、Jay Rubin、Bill Catlan、Anthony Howe、Mark O'Neal、George Montgomery、George Barber，以及其他无数耐心听我抱怨的人，我从他们那里了解到我所努力讲述的是否能让门外汉也能理解，或者仅仅得到一个我迫切希望的笑脸。没有他们，这本书可能也会写出来，但是，我几乎可以肯定我在这过程中会疯掉。

在进入 MySQL 世界之前，先照例介绍一下 MySQL 的历史（况且本书里也没提到这些）。

真正以 MySQL 为名的数据库是从 1994 年开始开发的，并于 1995 年第一次呈现在小范围的用户面前，它的开发者刚好不是美国人，而是两个瑞典人 Michael Widenius 和 David Axmark。那时的 MySQL 还非常简陋，除了在一个表上做一些 Insert、Update、Delete 和 Select 操作，恐怕没有更多的功能给用户使用。这种情况直到 2001 年左右发布 3.23 版的时候，才有了显著的进步——它支持大多数的基本 SQL 操作了，而且还集成了我们现在熟识的 MyISAM 和 InnoDB 存储引擎。然后又是几年不断完善的过程，到了 2004 年 10 月，这个夯实基础的过程到达了顶峰——4.1 这个经典版本发布了。次年 10 月，又一里程碑式的 MySQL 版本发布了，在新出的 MySQL 5.0 里加入了游标、存储过程、触发器、视图和事务的支持，准备进入中高端应用领域。在 5.0 之后的版本里，MySQL 明确地表现出迈向高性能数据库的发展步伐。

到今天，MySQL 已经上升到了 600 多万的装机量，著名的 WordPress、phpBB 都以 MySQL 为后台数据库，很多大型的 WWW 应用例如 Wikipedia、Google 和 Facebook，也都采用了 MySQL 作为它们的数据存储系统。

反观国内，鉴于心照不宣的原因，MySQL 的普及程度还不如 SQL Server。就我这些年来的所见所闻而言，一直作为 MySQL 黄金搭档的 PHP 都常常使用别的数据库，更别提其他开发语言了。好在那些上规模企业，尤其是外资企业里，多数明智的 IT 负责人在项目前期都会提议使用 MySQL，原因之一是它是免费的，一般不会产生授权费用问题，原因之二是它足够用了，不是吗？你想要的增、删、改、联接（Joint）、嵌套查询它都有；你想要的视图、存储过程、触发器、事务它也有；如果你要集群，它也能提供。


但是，使用 MySQL 是一回事，用好 MySQL 又是另外一回事。市面上更多的是关于 MySQL 开发的书籍，这些书籍的很多篇幅都花费在 SQL 语句的学习上。若要获得关于 MySQL 性能提高方面的资料，我们只能在网上的各个论坛或博客上披沙拣金了，而本书则系统性地从各个方面讲述一个高性能 MySQL 应用应该怎么来做。作者们都是这方面的行家里手，所以内容也是全面、充实，无论是架构师设计师、程序开发人员，还是系统管理员都能找到感兴趣的方面。在阅读正文前，最好能够先读一下作者精心编写的前言部分，通过它把握整本书内容的构成方式和相互关联，之后，带有目的地阅读本书会更富有成效。

本书由李军、王小东、康建勋三人合作翻译完成，其中，康建勋翻译第 1 章和第 2 章前 31 页；王小东翻译第 2 章的后 17 页，以及第 3 章至第 8 章；李军翻译了序言、前言、第 9 章至第 14 章、所有附录，以及作者介绍、封面、封底等内容，并撰写了内容简介。翻译的过程也是译者与编辑、审阅人员之间交互的过程，在这个过程中，编辑徐定翔老师，审阅人金照林老师、柳安意老师给予了我们很大的帮助。如果说译者是生产毛坯的工匠，那么他们就是把毛坯打磨成精品呈现给读者的人，在此十分感谢他们！

同样地，我们也要感谢家人和朋友。我们把那些本来应该陪伴家人出游，或者参加朋友聚会的时间，都“自私”地用在翻译本书上了。他们都比较宽容，一句“到时要请客哦”就原谅了我们，谢谢他们的支持！

最后，得向读者们说声抱歉，由于术业专攻不同、识见浅深有别之故，译文中难免会有诘屈聱牙、词不达意甚至疏误之处，还请读者不吝指正（译注 1）。

译者
2009 年 12 月

译注 1：由于本书篇幅较大，为了节约成本和便于读者阅读，我们将原书版式作了压缩，原书页码用“”表示，供读者对照。本书的索引（包括正文中的交叉索引）所列页码为原英文版页码。

MySQL 高级特性

Advanced MySQL Features

MySQL5.0 和 5.1 引入了许多特性，例如存储过程、视图和触发器。对于使用过其他数据库产品的用户来说，它们都是很熟悉的概念。这些新增特性也吸引了很多 MySQL 的新用户。但是只有在人们大规模地使用这些特性之后，才能知道它们的性能。

本章讨论了这些新增特性及其他的高级主题，包括一些在 MySQL4.1 或更老版本中就存在的特性。性能是讨论的重点，但是也展示了如何从这些特性中得到最大的益处。

5.1 MySQL 查询缓存

The MySQL Query Cache

许多数据库系统都可以缓存查询计划，服务器可以根据缓存对相同的查询跳过解析和优化阶段。在某些情况下，MySQL 也能做到这一点。但是它还有一种不同的缓存机制，叫做“查询缓存（Query Cache）”。这种缓存保存了 SELECT 语句的完整结果集（Complete Result Set）。本节的主题就是这种缓存。

MySQL 查询缓存保留了查询返回给客户端的完整结果。当缓存命中的时候，服务器马上返回保存的结果，并跳过解析、优化和执行步骤。

查询缓存保留了查询使用过的表，如果表发生了改变，那么缓存就失效了。这种失效的方法比较粗糙，看上去也不够高效，因为某些表的改变不会导致查询结果的改变。但是这种简单方式的开销比较小，而这对于繁忙的系统是很重要的。

查询缓存对应用程序完全透明。程序不用知道 MySQL 是从缓存中返回结果还是通过实际计算返回结果。两种方式返回的结果是一样的。换句话说，查询缓存不会改变语义。不管缓存是打开的还是关闭的，服务器的行为都一样（注 1）。

5.1.1 MySQL 如何检查缓存命中

How MySQL Checks for a Cache Hit

MySQL 检查缓存命中的方式相当简单快捷。缓存就是一个查找表（Lookup Table）。查找的键就是查询文本、当前数据库、客户端协议的版本，以及其他少数会影响实际查询结果的因素之哈希值。

注 1：查询缓存实际以一种微妙的方式改变了语义。在默认情况下，一个查询中使用的表即使被 LOCK TABLES 命令锁住了，查询也能被缓存下来。可以通过设置 query_cache_wlock_invalidate 来关闭这个功能。

在检查缓存的时候，MySQL 不会对语句进行解析、正则化或者参数化，它精确地使用客户端传来的查询语句和其他数据。只要字符大小写、空格或者注释有一点点不同，查询缓存就认为这是一个不同的查询。这是书写查询语句时要注意的一点。不管怎么说，使用一致的格式和风格是一个好习惯，而且在这种情况下还能得到更高的性能。

另外一件值得注意的事情就是查询缓存不会存储有不确定结果的查询。因此，任何一个包含不确定函数（比如 NOW() 或 CURRENT_DATE()）的查询不会被缓存。同样地，CURRENT_USER() 或 CONNECTION_ID() 这些由不同用户执行，将会产生不同的结果的查询也不会被缓存。事实上，查询缓存不会缓存引用了用户自定义函数、存储函数、用户自定义变量、临时表、mysql 数据库中的表或者任何一个有列级权限的表的查询。请参阅 MySQL 手册了解所有不会被缓存的查询类型。

经常可以听到“如果查询包含不确定函数，MySQL 就不会检查缓存”这样的说法。其实这是不对的。MySQL 只有在解析查询的时候才知道里面是否有不确定函数。缓存查找发生在解析之前。服务器会执行一次不区分大小写的检查来验证查询是否以字母 SEL 打头。这就是服务器在进行缓存查找前所做的所有事情。

但是，“如果查询包含 NOW() 这样的函数，服务器就不会在缓存中找到结果”这种说法是正确的。因为即使服务器在早些时候执行了同样的查询，服务器也不会有缓存的结果。MySQL 一旦发现有阻止缓存的元素存在，它里面就把查询标记为不可缓存，并且产生的结果也不会被保持下来。

对于引用了当天日期的查询，如果想让它被缓存下来，一个有用的技巧就是用一个字面常量来代替函数，比如下面的例子：

206

```
... DATE_SUB(CURRENT_DATE, INTERVAL 1 DAY) -- 不可缓存!  
... DATE_SUB('2007-07-14', INTERVAL 1 DAY) -- 可缓存
```

因为查询缓存只针对服务器第一次收到的完整 SELECT 语句，所以查询里面的子查询或视图不能使用缓存，存储过程中的查询也不能使用缓存。MySQL 5.1 之前的准备语句（Prepared Statement）也不能使用缓存。

MySQL 查询缓存可以改善性能，但是在使用的时候有一些问题值得注意。首先，开启查询缓存对于读写都增加了某些额外的开销。

- 读取查询在开始之前必须要检查缓存。
- 如果查询是可以被缓存的，但是不在缓存中，那么在产生结果之后进行保存会带来一些额外的开销。
- 最后，写入数据的查询也会有额外的开销，因为它必须使缓存中相关的数据表失效。

这些开销相对来说较小，所以查询缓存还是很有好处的。但是，稍后你会看到，额外的开销有可能也会增加。

对于 InnoDB 的用户，另外的问题就是事务限制了查询缓存的失效。当事务内部的语句更改了表，即使 InnoDB 的多版本机制应当对其他语句隐藏事务的变化，服务器也会使所有引用了该表的查询缓存失效。直到事务提交之前，该表会全局地不可缓存。所以不会有任何引用了该表的查询，不管它是在事务的内部还是外部，在事务提交之前都能被缓存。因此，长期运行的事务可以增加查询缓存未命中（Cache Miss）的数量。

失效对于大型查询缓存也会是一个问题。如果缓存中有许多查询，缓存失效就会需要很长的时间并且延缓整个系统的工作。这是因为查询缓存有一个全局锁，它会阻塞所有访问缓存的查询。在检查查询是否命中，以及是否有查询失效的时候都会发生访问动作。

5.1.2 缓存如何使用内存

How the Cache Uses Memory

MySQL 将查询缓存完全存储在内存中，所以在对它进行调优之前需要了解它如何使用内存。缓存不仅仅存储了查询结果，它在某种程度上像一个文件系统，它保持了自身的结构，而这些结构有助于它了解哪块内存是空闲的、表和查询之间的映射关系、查询文本、查询结果。

除了用于自身的 40KB 内存，查询缓存的内存池被分为大小可变的块。每一块都知道自己的类型、大小、数据量和指向前一个和后一个逻辑块和物理块的指针。内存块可以分为存储查询结果、查询使用的表的列表、查询文本等类型。然而，不同类型的块的处理方式都一样，所以对查询缓存调优的时候不用区分它们的类型。

服务器启动的时候会初始化查询缓存使用的内存。内存池最开始只有一个块。它的大小是被配置用于缓存的内存大小减去自身需要的 40KB。

在缓存查询结果时，服务器会为查询分配一块空间。如果服务器知道正在缓存一个较大的结果，这个块就会大一些。但是它至少等于 `query_cache_min_res_unit` 的值。不幸的是，服务器不能精确地分配大小，因为分配发生在结果产生之前。服务器不会在内存中生成最终的结果然后发送到客户端，而是每产生一行数据，就发送一行，因为这种方式效率更高。这造成的结果就是：当服务器开始缓存结果的时候，它无法知道结果最终会有多大。

分配内存块的速度相对较慢，因为服务器需要查看可用内存的列表并且找到大小合适的块。因此，服务器会尽量减少分配的次数。当需要缓存结果时，它会创建一个大小至少为最小值的块，并且把结果放入到块中。如果块已经满了，但是还有数据没有保存，服务器就会产生一个新块并且继续存储数据。在保存完成后，如果数据块还有剩余空间，服务器就会裁剪该块，并且把空间并入剩下的空闲空间中。图 5-1 显示了该过程（注 2）。

所谓的服务器“分配块”，并不是意味着向操作系统使用 `malloc()` 这样的函数请求内存空间。`Malloc` 过程只会在创建查询缓存的时候发生一次。这儿的意思是服务器检查数据块的列表并且选择一个最佳的位置放置新块。如果有需要，还会移除最旧的查询以腾出空间。MySQL 服务器管理自己的内存，不依赖于操作系统。

到目前为止，这些过程都非常简单。但是，实际过程比图 5-1 要复杂一些。假设平均结果都很小，服务器同时把结果发送到两个客户端。对数据块的裁剪结果就可能留下比 `query_cache_min_res_unit` 还小的空间，这些空间将来不能用于存储结果。

注 2：出于演示的目的，本节的图形已经进行了简化。服务器实际分配块的过程比这儿介绍的要复杂得多。如果想了解详情，`sql/sql_cache.cc` 这个文件头部的注释将该过程解释得很清楚。

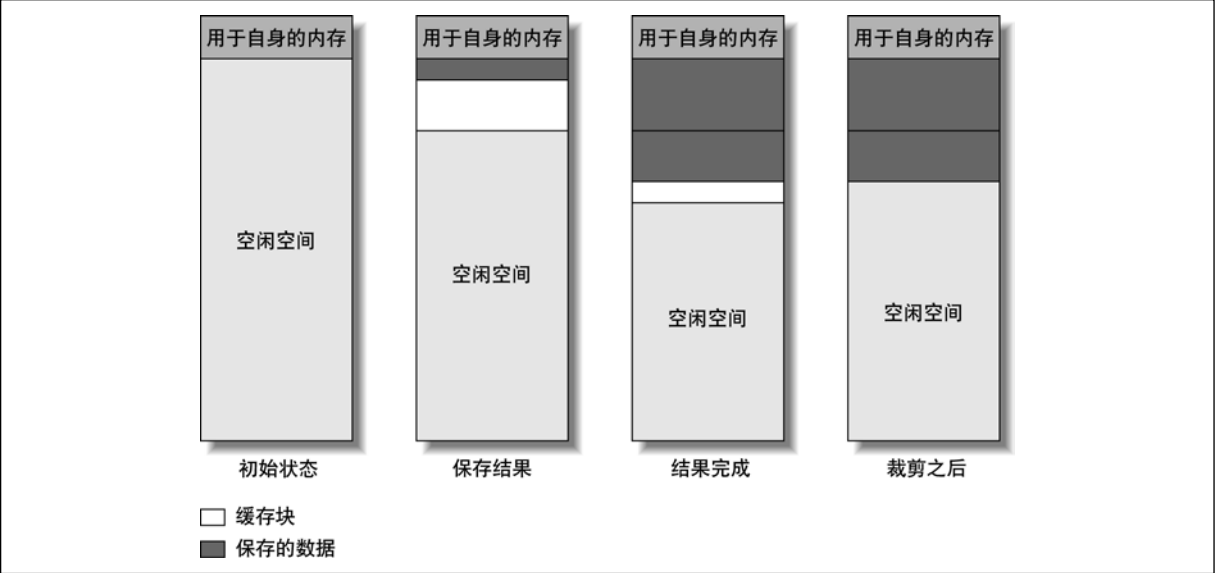


图 5-1：查询缓存如何分配内存块以存储数据

这时，数据块的分配看上去就像图 5-2。

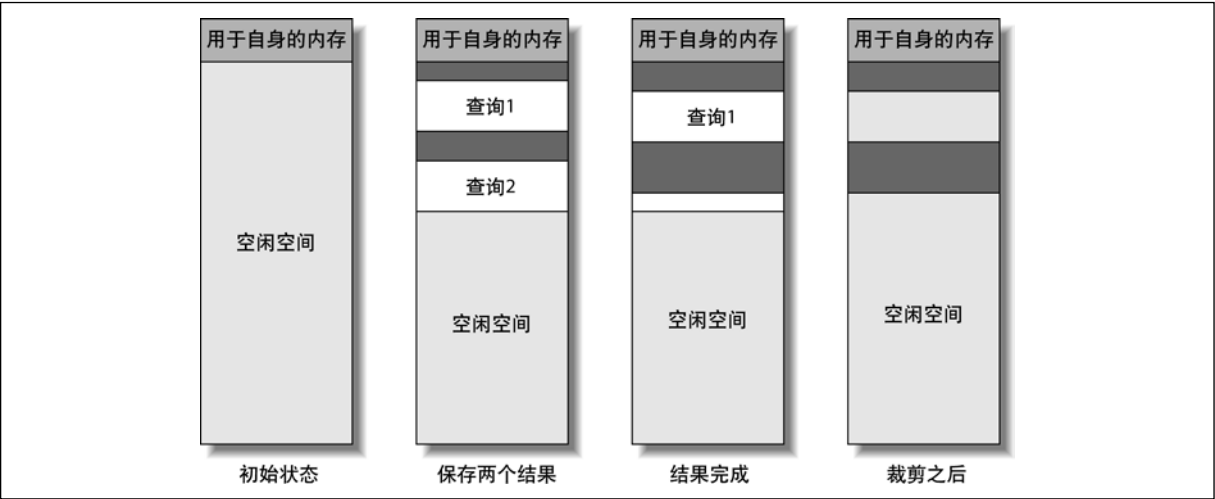


图 5-2：向查询缓存中存储结果引起的碎片

对第一个结果进行裁剪后，在两个结果之间留下了一块很小的空间，它不能存储结果，这就是碎片。它也是内存和文件系统空间分配的经典问题。导致碎片的原因是多样的，比如缓存失效会留下不能复用的小块空间。

5.1.3 查询缓存何时有帮助

When the Query Cache Is Helpful

缓存并不会自动地比非缓存高效。缓存也需要开销，只有在节省的资源大于开销的时候，缓存才是真正有效率的，这和服务器的负载相关。

在理论上,可以通过对比在缓存开启和关闭时服务器需要做的工作来了解缓存是否有帮助。在缓存关闭的时候,读取操作需要执行查询语句并且把结果返回给客户,写入操作需要执行查询。在缓存启用的时候,读取首先会检查查询缓存,然后要么直接返回结果,要么执行查询语句,保存结果,再返回结果。每一个写入操作都需要执行查询语句并且检查是否有缓存过的查询要失效。

尽管这听上去很直接,但其实不是。实际上很难精确地计算或者预测查询缓存的好处。必须考虑外部因素。例如,查询缓存可以减少产生结果的时间,但它不会减少将结果发送到客户端的时间,而这有可能是主要因素。

从缓存中受益最多的查询可能是需要很多资源来产生结果,但是不需要很多空间来保存的类型。所以用于存储、返回和失效的代价都较小。聚集查询,比如从大表中利用 `COUNT()` 产生较小的结果,就符合这个范畴。但是也有其他很多种查询值得缓存。

检查是否从查询缓存中受益的最简单的办法就是检查缓存命中率。它是缓存提供的查询结果的数量,而不是服务器执行的数量。当服务器收到 `SELECT` 语句的时候, `Qcache_hits` 和 `Com_select` 这两个变量会根据查询缓存的情况进行递增,查询缓存命中率的计算公式是: $Qcache_hits / (Qcache_hits + Com_select)$ 。

命中率要多少才好?这视情况而定,即使 30% 的命中率也可能很有帮助。因为对于每一个查询,不执行它所节约的资源远大于在缓存中保存结果及让查询失效的开销。知道哪个查询被缓存了也很重要。如果缓存命中代表了开销最大的查询,那么即使是很低的命中率也有很大的好处。

任何一个不在缓存中存在的查询都是**缓存未命中**。缓存未命中可能是因为下面的原因:

- 查询不可缓存。原因可能是含有不确定函数,比如 `CURRENT_DATE`,也有可能是结果太大,无法缓存。状态变量 `Qcache_not_cached` 会因为这两种无法缓存的查询而增加。
- 服务器以前从来没见过这个缓存,所以它根本就没有机会缓存自身结果。
- 查询的结果以前被缓存过,但是服务器把它移除了。发生移除的原因可能是内存空间不够,所以被人从服务器上把它删除了,也可能是缓存失效了。

如果服务器有很多缓存未命中,但是不能缓存的查询却很少,那么原因应该是下面之一:

- 查询缓存未被激活,也就是说服务器根本就没有机会将结果存储到缓存中。
- 服务器看到了以前未曾见过的查询。如果没有很多重复的查询,即使缓存被激活了,也有可能见到这种情况。
- 有很多缓存失效。

缓存可能会因为碎片、内存不足或数据改变而失效。如果已经给缓存分配了足够的内存,并且把 `query_cache_min_res_unit` 调整到了合适的值,那么大部分缓存失效都应该是由数据改变引起的。可以通过检查 `Com_*` (`Com_update`, `Com_delete` 等) 的值知道有多少查询修改了数据。也可以通过检查 `Qcache_lowmem_prunes` 的值了解有多少查询因为内存不足而失效。

一个不错的想法就是把缓存失效的开销和命中率分开考虑。举一个极端的例子,假设有一个表只发生读取动作并且命中率是 100%,另外一张表只发生更新。如果只简单地从状态变量计算命中率,它始终都是 100%。然而,查询缓存在这种情况下仍然有可能效率不高,因为更新被减慢了。所有的更新查询在完成后都不得不访问缓存,检查是否有其他的查询会因为数据变化而失效。但实际上答案始终是“否”。所以这些工作就全部浪费了。如果不在检查命中率的同时检查不可缓存的查询数量,那么就不可能发现这些问题。

如果对同一张表进行大致平均的读写，服务器也可能不会从查询缓存中得益。写入数据会不停地让缓存失效，而读取数据会不停地把新结果插入缓存中。在这种情况下，只有发生后续读取，这种缓存才是有益的。

211

如果在服务器收到同一个查询语句之前，缓存就失效了，那么保持结果就只能是浪费时间和内存。检查 `Com_select` 和 `Qcache_inserts` 的相对大小可以确认这种情况是否发生。如果差不多所有的 `SELECT` 语句都是缓存未命中（`Com_select` 会因此增加），并且接下来把结果保存到了缓存中，那么 `Qcache_inserts` 就会和 `Com_select` 差不多大小。因此，`Qcache_inserts` 一般比 `Com_select` 小得多，至少在缓存被正确地激活后是这样的。

每个应用程序都有确定的潜在缓存大小，即使没有写入查询也是如此。潜在缓存用于保存应用程序所有可缓存查询的内存数量。从理论上说，它对于大部分应用程序是极大的数字。在实际中，由于失效的数量，许多应用程序可用的缓存比期望的小得多。即使把查询缓存设置得非常小，实际也不可能超过潜在缓存的大小。

你应该监视服务器实际使用的缓存数量。如果它没有用到分配的内存，那么就应该把分配给它的内存减少一点。如果由于内存限制引起了缓存失效，那么就应该多分配一些内存。但是不用太在意缓存的大小，它比有实际影响的稍大一点或小一点都没有问题。只有在内存有严重浪费或者缓存失效太多的时候才需要去考虑它的大小。

还应该在服务器其他缓存和查询缓存之间找到某种平衡，例如 InnoDB 的缓存池或 MyISAM 的键缓存。它们之间没有简单的公式或固定的比例，因为这取决于应用程序。

5.1.4 如何对查询缓存进行维护和调优

How to Tune and Maintain the Query Cache

一旦了解查询缓存的工作机制，对它进行调优就是一件容易的工作。它只有几个“活动部分”。

Query_cache_type

这个选项表示缓存是否被激活。具体选项是 `OFF`、`ON` 或 `DEMAND`。`DEMAND` 的意思是只有包含了 `SQL_CACHE` 选项的查询才能被缓存。它既是会话级变量，也是全局性变量（更多关于会话变量和全局变量的话题请参阅第 6 章）。

Query_cache_size

分配给查询的总内存，以字节为单位。它必须是 1024 的倍数。所以 MySQL 实际使用的值可能和定义的值稍有不同。

Query_cache_min_res_unit

分配缓存块的最小值。第 206 页的“缓存如何使用内存”解释过该设置，下节会对它做进一步讨论。

212

Query_cache_limit

这个选项限制了 MySQL 存储的最大结果。如果查询的结果比这个值大，那么就不会被缓存。要知道的是服务器在产生结果的同时进行缓存，它无法预先知道结果是否会超过这一限制。如果在缓存的过程中发现已经超过了限制，MySQL 会增加 `Qcache_not_cached` 的值，并且丢掉已经缓存过的值。如果知道会发生这样的事，那么给查询加上 `SQL_NO_CACHE`，可以避免这种开销。

这个选项指是否缓存其他联接已经锁定了的表。默认值是 OFF，可以让你从其他联接已经锁定了的表中读取缓存过的数据，这改变了服务器的语义。因为这种读取通常是不被允许的。把它改成 ON 会阻止读取数据，但有可能增加锁等待。它对于大多数程序都没有影响，所以通常保持默认值就可以了。

在原则上，对缓存进行调优很简单，但是理解自己所做的改变的影响则要复杂得多。接下来的章节展示了如何对查询缓存进行推理，以做出正确的决定。

减少碎片

没有办法避免所有的碎片，但是仔细地选择 query_cache_min_res_unit 可以避免在查询缓存中造成大量的内存浪费。关键在于每一个新块和服务已分配给存储结果的块的数量之间找到平衡。如果值过小，服务器将会浪费较少的内存，但会更频繁地分配块，这对服务器意味着更多的工作。如果值过大，碎片将会很多。合适的折中是在浪费内存和增加 CPU 处理时间上取得平衡。

最佳设置根据典型查询结果而定。可以用使用的内存（大致等于 query_cache_size - Qcache_free_memory）除以 Qcache_queries_in_cache 得到查询的平均大小。如果缓存由大结果和小结果混合而成，那么就很难找到一个合适的大小，既能避免碎片，也能避免过多的内存分配。但是，有理由相信缓存大结果没有太大的益处（这通常是真的）。可以通过降低 query_cache_limit 的值阻止缓存大结果，它有时有助于在碎片和在缓存中保存结果的开销中得到平衡。

可以通过检查 Qcache_free_blocks 的值来探测缓存中碎片的情况，它可以显示缓存中有多少内存块处于 FREE 状态。图 5-2 中最后一步显示了两个处于 FREE 的块。碎片最严重的情况就是在每两个存储了数据的块之间都有一个比最小值稍小的可用块。这样的话，每隔一个存储块就有一个自由块。因此，如果 Qcache_free_blocks 大致等于 Qcache_total_blocks/2，则说明碎片非常严重。如果 Qcache_lowmem_prunes 的值正在增加，并且有大量的自由块，这意味着碎片导致查询正被从缓存中永久删除。

可以使用 FLUSH QUERY CACHE 命令移除碎片。这个命令会把所有的存储块向上移动，并把自由块移到底部。当它运行的时候，它会阻止访问查询缓存，这锁定了整个服务器。但它通常都很快，除非缓存非常大。和名字相反，它不会从缓存中移除查询，RESET QUERY CACHE 才会这么做。

提高查询缓存的可用性

如果缓存没有碎片，但是命中率却不高，那么就应该给缓存分配较少的内存。如果服务器找不到足够大小的块来存储结果，那么就应该从缓存中清理掉一些查询。

当服务器清理查询的时候，Qcache_lowmem_prunes 的值会增加。如果它的值增加得很快，那么可能有两个原因：

- 如果有很多自由块，那么问题可能是由碎片引起的（参阅前一节）。
- 如果自由块比较少，那么这可能意味着工作负载使用的内存大小超过了所分配的内存。可以检查 Qcache_free_memory 知道未使用的内存数量。

如果有很多自由块，碎片很少，由于内存不足引起的清理工作也很少，但是命中率仍然不高，这说明工作负载也许不能从缓存中受益。肯定有什么东西阻止查询使用缓存，很多 update 语句可能是原因，另外一个可能的原因是查询是不可缓存的。

如果已经估算过缓存命中率，但是还不确定服务器是否从缓存中受益，此时可以禁用缓存并且监控性能，然后重新开启缓存并观察性能变化。为了禁用缓存，可以将 `query_cache_size` 设置为 0（改变 `query_cache_type` 不会从全局上影响已经打开了的连接，而且不会把内存归还给服务器）。也可以做基准测试，但是有时候很难得到包含了可缓存的查询、不可缓存的查询，以及更新语句的测试样例。

图 5-3 用一个基本例子显示了分析和调整查询缓存的流程。

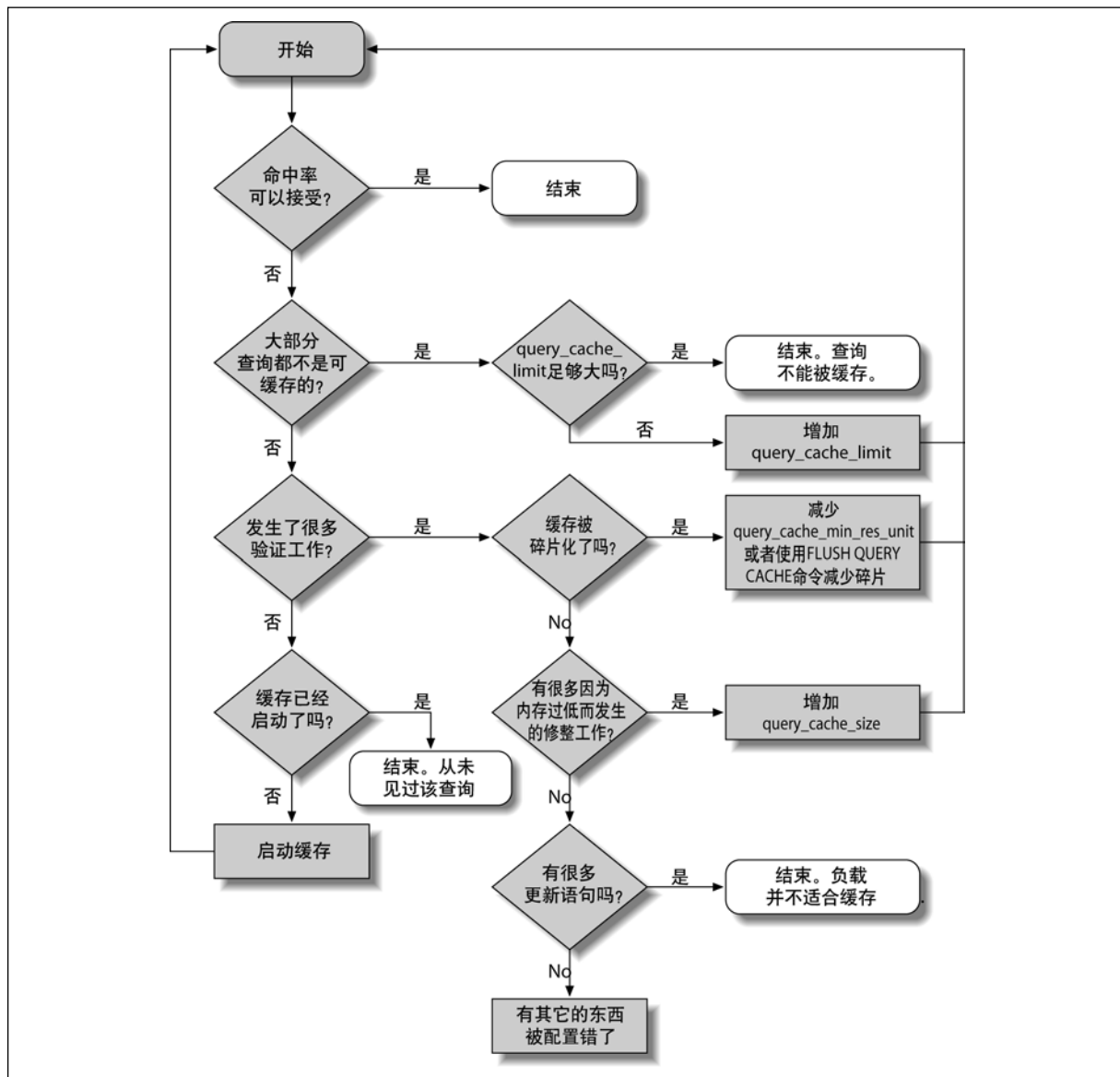


图 5-3：如何分析调整查询缓存

5.1.5 InnoDB 和查询缓存

InnoDB and the Query Cache

InnoDB 由于其 MVCC 架构，和查询缓存的交互比其他存储引擎要复杂得多。在 MySQL 4.0 中，查询缓存在事务内部是被完全禁止的，但是在 MySQL 4.1 及其新版本中，InnoDB 会针对每一个表请示服务器一个事务是否可以访问查询缓存。它控制了读（从缓存中获取数据）、写（向缓存中保存数据）操作对缓存的访问。

决定访问的因素是事务 ID 及表上是否有锁。每个表在 InnoDB 的内存内部数据字典中都有一个对应的事务 ID 计数器。ID 小于计数器值的事务会被禁止从缓存读取或写入数据。表上的任何锁都会导致使用该表的查询变得不可缓存。例如，如果事务在表上面执行了 SELECT FOR UPDATE 查询，那么直到锁解除为止，不会有任何其他的事务能从缓存对它进行读写操作。

当事务提交的时候，InnoDB 会在锁定的基础上更新计数器的值。锁可以作为粗略的提示，用来表示事务是否修改了表。有可能事务锁住了表却没有更新它，但是不可能在没有获取锁的情况下修改表的内容。InnoDB 把每个表的计数器设置为系统的事务 ID，它是已有的事务 ID 的最大值。

这会造成下面的结果：

- 使用查询缓存的事务中的表的计数器位于绝对下限。如果系统事务 ID 是 5，并且一个事务请求了表的行锁，然后提交了，事务 1 到 4 中引用的这个表的查询将再也不能对缓存进行读写操作。
- 表的计数器没有按照使用表的事务 ID 进行更新，却按照系统事务 ID 进行了更新。这样的后果就是这个事务发现自己以后也不能在缓存中操作引用了这个表的查询。

查询缓存的存储、获取及失效都在服务器级进行，并且 InnoDB 不能绕过或延迟这种行为。但是，InnoDB 能显式地告诉服务器让使用了某些特定的表的查询失效。在外键约束，比如 ON DELETE CASCADE，修改了查询中未出现的表的内容时，这是有用的。

从原则上说，如果对表的改动不会影响其他事务看到的连续读取视图，InnoDB 的 MVCC 架构就可以让缓存为查询服务。但是，实现这种方式是很复杂的。InnoDB 的算法出于简单性的考虑走了一些捷径，其代价就是在实际并不需要的时候锁住了查询缓存之外的事务。

5.1.6 通用查询缓存优化方案

General Query Cache Optimizations

架构、查询以及应用程序设计会影响查询缓存。除了上节讨论的内容，下面还有一些需要注意的观点：

- 使用多个较小的表，而不是用一个表，对查询缓存有帮助。这种设计方式使失效策略工作在一个较好的颗粒度上。但是不要让这个想法过度影响架构设计，因为其他的因素能轻易抵消它的好处。
- 成批地进行写入操作，而不是逐个执行，会有效率得多。因为这种方法只会引起一次失效操作。
- 我们已经注意到在让缓存失效或清理一个大型缓存的时候，服务器可能会挂起相当长时间。至少在 MySQL 5.1 之前的版本中是这样。一个容易的解决办法就是不要让 query_cache_size 太大，256MB 已经太大了。
- 不能在数据库或表的基础上控制查询缓存，但是可以使用 SQL_CACHE 和 SQL_NO_CACHE 决定是否缓存查询。也可以基于某个连接来运行或禁止缓存，可以通过用适当的值设定 query_cache_size 来开启或关闭对某个连接的缓存。

- 对于很多写入任务的应用程序，关闭查询缓存也许能改进性能。这样做可以消除缓存那些很快就会失效的查询所带来的开销。要记住在禁用时需要把 `query_cache_size` 设置到 0，这样就不会消耗任何内存。

如果想让大多数查询都不使用缓存，但是有少部分查询能从缓存中极大地受益，这时可以将全局变量 `query_cache_type` 设置为 `DEMAND`，然后在想使用缓存的查询后面添加 `SQL_CACHE`。尽管这会造成更多的工作，但是可以细粒度地控制缓存。相应地，如果想缓存大部分查询，只排除其中一小部分，就可以使用 `SQL_NO_CACHE`。

5.1.7 替代查询缓存的方法

Alternatives to the Query Cache

MySQL 查询缓存的原则就是最快的查询不需要执行，但是仍然需要发起查询，服务器也需要做一点工作。如果某些特殊的查询完全不想和服务器沟通，那该怎么办？客户端缓存可以进一步减轻 MySQL 服务器的负载。第 10 章提供了更多的细节。

217

5.2 在 MySQL 中存储代码

Storing Code Inside MySQL

MySQL 可以用触发器、存储过程和存储函数把代码保存在服务器内部。MySQL 5.1 中还可以把代码保存为名为“事件 (Event)”的周期性任务。存储过程和存储函数被统称为“存储例程 (Stored Routines)”。

这 4 种存储代码使用了一种特殊的 SQL 扩展语言，它包括了循环，以及条件判断等过程性结构（注 3）。这些存储代码之间最大的区别就是他们操作的上下文，也就是输入和输出。存储过程和存储函数能接受参数并返回结果，触发器和事件则不能。

从原则上说，存储代码是共享和复用代码的好方式。Giuseppe Maxia 和其他人在 <http://mysql-sr-lib.sourceforge.net> 创建了通用的存储例程库。但是很难在其他数据库系统中重用代码，因为它们都包含了语言（DB2 是一个例外，它和 MySQL 基于同一个标准，语言很相似）（注 4）。

本书主要集中在存储代码的性能，而不是具体语法上。O'Reilly 公司的《MySQL Stored Procedure Programming》（Guy Harrison 和 Steven Feuerstein 著）对在 MySQL 中书写存储过程很有用。

存储代码的支持者和反对者随处可见。我们不会偏向任意一方，下面列出了它的优点和缺点。首先，它有如下优点：

- 它在代码存在的地方运行，所以可以在服务器内部运行任务，从而节约带宽和减少延迟。
- 它是一种代码复用的方式。它能集中商业逻辑，这可以加强程序行为的一致性并且让人更加省心。
- 它可以减少发布和维护的开销。
- 它提供了安全性方面的优势，以及更好地控制权限的方式。一个通常的例子就是用于转账的存储过程。过程可以在事务内部完成转账，并且把所有的操作都记录下来。可以让应用程序调用存储过程而不用授权

注 3：这种语言是 SQL/PSM (Persistent Stored Modules) 的子集。它定义于 ISO/IEC 9075-4:2003(E)。

注 4：一些移植工具，比如 `tsql2mysql` (<http://sourceforge.Net/projects/tsql2mysql>) 可以把 Microsoft SQL Server 转换为 MySQL。

访问特定的表。

- 服务器缓存了存储过程的执行计划，这可以降低重复调用的开销。
- 存储代码被保存在服务器里面，可以和服务器一起发布、备份和维护。它很适合维护任务。它不依赖于任何外部的组件，比如 Perl 库和其他你不想放在服务器上的软件。
- 它让应用程序程序员和数据库程序员的工作可以分开。数据库专家肯定更适合写存储过程，因为并不是应用程序方面的每一个程序员都善于写高效的查询语句。

它有下面的缺点：

- MySQL 没有提供很好的开发和调试工具，所以在 MySQL 里面写存储代码比在其他数据库服务器要难。
- MySQL 语言比应用程序语言慢而且原始。可用的函数是有限的，并且很难写出复杂的字符串操作和逻辑。
- 存储代码实际增加了部署应用程序的复杂性。除了程序代码和数据库架构的改变，还需要部署服务器内部的代码。
- 因为存储例程和数据库保存在一起，所以它可能变成安全隐患。比如存储例程使用了非标准化的加密函数，那么在数据库受到攻击的时候就无法保护数据。如果加密函数在代码里面，攻击者将危及数据库和代码。
- 存储例程把负载转移到了数据库服务器，它通常更难扩展并且比应用程序和网页服务器更昂贵。
- 你不能通过 MySQL 控制存储代码分配的资源，因此一个错误就能导致服务器宕机。
- MySQL 对存储代码的执行有很多限制，比如执行计划缓存基于单个连接，游标是用临时表实现的，等等（我们在讨论各种特性的时候会涉及限制）。
- 在 MySQL 里面很难剖析存储过程的代码。当缓慢查询显示 `CALL XYZ('A')` 的时候，对它的分析就很难。因为你不得不找到该存储过程并且调查代码。
- 存储代码是一种隐藏复杂性的方式，它简化了开发但是常常没有好的性能。

在质疑是否该使用存储代码的时候，要问问自己业务逻辑会被放在什么地方。是应用程序代码中？还是数据库里面？这两种方式都很普遍，要知道的是，当使用存储代码的时候，业务逻辑就在数据库里面了。

5.2.1 存储过程和函数

Stored Procedures and Functions

MySQL 的架构和查询优化器对使用存储例程（Stored Routines）和它们的效率有一些限制。在写本书时，有如下限制：

- 优化器不能使用存储函数内部的 `DETERMINISTIC` 修饰符把重复调用优化掉。
- 优化器无法估计执行存储函数的开销。
- 每个连接都有自己的存储过程执行计划缓存。如果许多连接调用同一个存储过程，它们将会浪费资源去缓存同样的执行计划（如果使用连接池或持续连接，执行计划将会被缓存得更久一些）。
- 存储例程和复制（Replication）以令人困惑的方式交织在一起。如果不想复制对存储例程的调用，而想复制数据集上发生的实实在在的改变，那么就应该使用 MySQL 5.1 引入的基于行复制的机制。如果在 MySQL 5.0 中开启了二进制日志记录，服务器就会强制要求要么把所有的存储过程定义为 `DETERMINISTIC`，要么开启服务器上的 `log_bin_trust_function_creators` 选项。

最好把存储例程做得既小又简单。最好用面向过程的语言把复杂的逻辑放在数据库外面，这会更有表达性和通用性。这也可以访问更多的计算资源并且有可能使用不同类型的缓存。

但是，存储过程对于某些类型的操作是很快，尤其是对于小型查询。如果一个查询足够小，解析和网络通信的开销就会成为主要因素。作为展示，下面有一个简单的存储过程，它向一个表插入特定数目的行，代码如下：

```

1 DROP PROCEDURE IF EXISTS insert_many_rows;
2
3 delimiter //
4
5 CREATE PROCEDURE insert_many_rows (IN loops INT)
6 BEGIN
7     DECLARE v1 INT;
8     SET v1=loops;
9     WHILE v1 > 0 DO
10         INSERT INTO test_table values(NULL,0,
11             'qqqqqqqqqqwwwwwwwwwEEEEEEEEEErrrrrrrrrrttttttttt',
12             'qqqqqqqqqqwwwwwwwwwEEEEEEEEEErrrrrrrrrrttttttttt');
13         SET v1 = v1 - 1;
14     END WHILE;
15 END;
16 //
17
18 delimiter ;
```

然后用该存储过程插入 100 万行数据，并和使用客户端程序每次插入一行相对比。表的结果和使用的硬件没有太大关系，重要的是它们之间的相对速度。出于好玩的目的，我们也测试了通过 MySQL 代理连接服务器的时间。为了保持测试的简单性，在同一台服务器上运行所有的测试。表 5-1 显示了结果。

表 5-1：一次性插入 100 万行数据耗费的时间

方法	总时间
存储过程	101 秒
客户端程序	279 秒
使用了 MySQL 代理的客户端程序	307 秒

存储过程要快得多，主要原因是它避免了网络通信、解析、优化等的开销。

在第 227 页中的“准备语句的 SQL 接口”，有一个典型的用于维护工作的存储过程示例。

5.2.2 触发器

触发器 (Trigger) 可以在执行 INSERT、UPDATE 或 DELETE 的时候运行代码。触发器可以在这些语句之前或之后运行。触发器不会返回结果, 但是它可以读取或修改数据。因此, 可以使用触发器代替客户端代码来强制约束 (Constraint) 或保证商业逻辑。一个不错的例子就是在类似于 MyISAM 这样的不支持外键的存储引擎上模拟外键。

触发器可以简化程序逻辑并提高效率。因为它节约了数据服务器和客户端之间数据往来的开销。它对自动更新非正则化表和汇总表很有帮助。例如，Sakila 数据库就使用它来维护 `film_text` 表。

在写作本书的时候，MySQL 触发器还未被完整地实现。如果你习惯于在其他数据库产品中广泛地使用触发器，那么就应该假设它在 MySQL 中不会按照同样的方式工作，尤其是：

- 对于每个事件，在每个表上只能有一个触发器。换句话说，不会有两个触发器启动 AFTER INSERT 事件。
- MySQL 只支持行级触发器，也就是说，服务器总是针对每一行进行操作，而不是把它们看成一个整体。这使得处理大型数据集的效率很差。

下面这些针对触发器的通用警示条款也适用于 MySQL：

- 它们让服务器正在做的事情变得模糊，因为一个简单的语句也可能导致很多“不可见”的工作。例如，如果一个触发器更新了一个相关的表，那么受影响的行的数量就会加倍。
- 触发器很难调试，并且它也不利于分析性能。
- 触发器会引起不明显的死锁和锁等待。如果触发器失败了，原始查询也会失败，如果没意识到触发器的存在，就很难准确地解释错误码。

从性能上说，MySQL 触发器实现上最严重的问题就是按行操作数据。这有时让使用触发器维护汇总表和缓存表变得不可能，因为它们会变得非常慢。使用触发器代替周期性的大量更新语句的主要原因是它们能让数据随时都保持一致。

触发器也不能保持原子性。例如，一个更新 MyISAM 表的触发器发生了错误，但却无法回滚。假设在一个 MyISAM 表中使用 AFTER UPDATE 触发器更新另外一个 MyISAM 表，如果触发器发生错误，导致第二个表更新失败，但第一个表的更新却不会被回滚。

InnoDB 表上的触发器的所有操作都在一个事务中完成，所以触发器和引发它的操作是原子操作。然而，如果做约束验证的时候使用 InnoDB 上的触发器去检查另外一个表的数据，那么就要当心 MVCC 架构，因为如果不小心的话，就不会得到正确的结果。例如，假设使用 BEFORE INSERT 触发器校验另外一个表中是否有匹配的数据，但是在读取数据的时候如果没有在触发器中使用 SELECT FOR UPDATE，对表的并发更新就会导致不正确的结果。

我们并不是让你害怕触发器，相反地，它们非常有用，特别是对于约束、系统维护任务，以及在同步中保持非正则化数据。

也可以使用触发器记录每一行的改变。它对于定制的复制器的配置非常方便，可以用于断开连接、改变数据，然后再把改变合并在一起。一个简单的例子就是一群带着笔记本去上班的用户，他们的改变需要被同步到主服务器上，然后主服务器需要把这些改变拷回各自的机器。这需要双向同步。触发器是构建这种系统的好方式。每台笔记本使用触发器把数据改变全部都记录到一个表中，然后使用定制的工具把这些改变同步到主服务器中。最后，普通的 MySQL 复制机制可以让笔记本和主服务器保持同步，服务器此时已经有了所有的改变。

有时甚至可以为 FOR EACH ROW 设计找到变通的办法。罗兰德·波曼 (Roland Bouman) 发现触发器里面 ROW_COUNT() 始终返回 1，只有 BEFORE 触发器的第一行除外。可以使用这种办法防止触发器代码对每一行都执行一遍，并且只对每个语句运行一次。这和每个语句的触发器不同。但是它对于模拟每个语句的 BEFORE 触发器是一种有用的技巧。这种行为其实是一个缺陷，并且会在某个时候被修复。所以应该小心地使用它，并且在升级服务器的时候应该验证它是否还能继续工作。下面的例子展示了如何使用这种方式：

```
CREATE TRIGGER fake_statement_trigger
BEFORE INSERT ON sometable
FOR EACH ROW
BEGIN
    DECLARE v_row_count INT DEFAULT ROW_COUNT( );
```

```

        IF v_row_count <> 1 THEN
            -- Your code here
        END IF;
    END;

```

5.2.3 事件

Events

MySQL 5.1 提供了一种新的存储代码：事件（Event）。它类似于定时任务（Cron Job），但内部机制却完全不同。你可以创建事件，它会在某个特定时间或时间间隔执行一次预先写好的 SQL 代码。通常的方式就是将复杂的 SQL 语句包装到一个存储过程中，然后调用一下即可。

事件和连接完全无关，它运行在一个独立的定时器线程上。事件不接受参数，也不返回值，没有任何连接让它们可以得到输入或返回输出。如果激活了服务器日志，就可以在日志中看到它们执行的命令，但是很难分辨哪些命令是从事件中执行的。可以查看 INFORMATION_SCHEMA_EVENTS 表了解事件的状态，比如上次执行时间。

事件有和存储过程相同的顾虑，那就是让服务器做了多余的工作。事件自身的开销是很小的，但是它调用的 SQL 语句可能会对性能产生严重的影响。适合事件的任务包括周期性的维护工作、重新建立缓存和汇总表以模拟物化视图，或者保存用于监视和诊断的状态值。

223

下面是一个事件的例子，它每周都会针对特定的数据库运行一次某个存储过程（注 5）。

```

CREATE EVENT optimize_somedb ON SCHEDULE EVERY 1 WEEK
DO
CALL optimize_tables('somedb');

```

可以定义事件是否应该被复制到从服务器。在某些情况下，复制是合适的，但是对另外一些情况则未必。以前一个例子为例，比如想对所有从服务器运行 OPTIMIZE TABLE 操作，但是要知道如果所有从服务器同时执行这种操作的话，性能就会受到某些因素的影响（比如表锁）。

最后，如果一个周期性事件需要较长的时间才能完成，那么有可能在前一个操作还没完成的时候后一次操作又发生了。MySQL 不会为这种行为提供保护，所以需要自己定义互斥的代码。可以使用 GET_LOCK() 来保证每次只有一个事件在运行：

```

CREATE EVENT optimize_somedb ON SCHEDULE EVERY 1 WEEK
DO
BEGIN
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
        BEGIN END;
    IF GET_LOCK('somedb', 0) THEN
        DO CALL optimize_tables('somedb');
    END IF;
    DO RELEASE_LOCK('somedb');
END

```

上面代码中的 CONTINUE 保证了即使存储过程抛出异常，锁也能得到释放。

尽管事件和连接无关，但它和线程有关。服务器有一个主调度线程，必须用配置文件或利用命令把它激活：

```
mysql> SET GLOBAL event_scheduler := 1;
```

注 5：稍后会介绍如何创建存储过程。

一旦激活了它，每执行一个事件，都会创建一个新线程。在事件代码内部，调用 `CONNECTION_ID()` 会返回一个唯一的值，这个值其实是线程 ID。可以通过查看服务器错误日志了解事件执行情况。

5.2.4 保存存储代码的注释

Preserving Comments in Stored Code

存储过程、存储函数、触发器和事件都含有大量的代码，有适当的注释会很有帮助。但是注释不会被保持在服务器内部，因为命令行的客户端把它们都去掉了（命令行客户端的这个特性很不好，但这就是现实）。

一个保存注释的有用技巧就是使用特定版本的注释方式，这时服务器会把它们看做潜在的可执行代码，比方说它们只能在某个版本或更高版本的服务器上执行。服务器和客户端程序知道它们不是普通的注释，所以就不会把它们丢掉。为了防止这些“代码”被执行，可以使用一个很高的版本数字，比如 99999。例如，下面就是一个给触发器加注释的例子：

```
CREATE TRIGGER fake_statement_trigger
BEFORE INSERT ON sometable
FOR EACH ROW
BEGIN
    DECLARE v_row_count INT DEFAULT ROW_COUNT( );
    /*!99999
       ROW_COUNT( ) is 1 except for the first row, so this executes
       only once per statement.
    */
    IF v_row_count <> 1 THEN
        -- Your code here
    END IF;
END;
```

5.3 游标

Cursors

MySQL 现在提供了只能向前的只读游标（Cursor），它只能在存储过程中使用。它可以迭代每一行结果，并且把结果提取到变量中以做进一步处理。一个存储过程在同一时刻可以使用多个游标，也可以在循环中嵌套游标。

MySQL 以后也许会提供用于更新的游标，但现在没有。因为游标读取的是临时表，而不是产生数据的原始表，所以它是只读的。

一不小心就会掉入游标的陷阱，它是在临时表上执行的，容易让开发人员产生对性能的误解。最重要的事情是要知道当打开游标的时候，它执行的是整个查询。看看下面的存储过程：

```
1 CREATE PROCEDURE bad_cursor( )
2 BEGIN
3     DECLARE film_id INT;
4     DECLARE f CURSOR FOR SELECT film_id FROM sakila.film;
5     OPEN f;
6     FETCH f INTO film_id;
7     CLOSE f;
8 END
```

这个例子说明了可以在对所有的结果进行迭代前关闭游标。习惯了 Oracle 和 Microsoft SQL Server 的用户不会觉得这个过程有什么错误，但它引起了很多不必要的工作。使用 `SHOW STATUS` 命令分析这个存储过程，发现它

进行了 1000 次索引读取和 1000 次插入。其原因是 `sakila.film` 表有 1000 行数据。所有这 1000 次操作都发生在代码的第 5 行。

这个例子的价值就是告诉你早早关闭一个从大型结果中读取数据的游标，不会节约任何的工作。如果只需要几行数据，就应该使用 `LIMIT`。

游标也会导致 MySQL 执行额外的 I/O 操作，这会非常慢。临时表位于内存中，它不支持 `BLOB` 和 `TEXT` 数据类型，MySQL 不得不在磁盘上为包含这些类型的游标创建临时表。如果没有这种情况，当临时表大于 `tmp_table_size` 时，MySQL 也会在磁盘上创建临时表。

MySQL 不支持客户端游标，但是客户端 API 通过把结果提取到内存中，可以模拟游标操作。这和在应用程序中把结果放入数组没什么区别。第 161 页的“MySQL 客户端/服务器协议”有更多把结果放到客户端内存对性能影响的内容。

5.4 准备语句

Prepared Statements

MySQL 4.1 及其更高版本支持服务器端准备语句（Prepared Statements），它使用增强的二进制客户端/服务器协议在客户端和服务端之间高效地发送数据。可以通过支持这种新协议的编程库来访问准备语句，例如 MySQL C API。MySQL Connector/J 和 MySQL Connector/NET 为 Java 和 .NET 提供了同样的访问接口。它也有 SQL 语言的访问接口，稍后再讨论这一接口。

创建准备语句时，客户端库会向服务器发送一个实际查询的原型，然后服务器对该原型进行解析和处理，将部分优化过的原型保存起来，并且给客户端返回一个状态句柄（State Handle）。客户端可以通过定义状态句柄重复地执行查询。

准备语句可以有参数，它用问号（?）代表执行时的具体参数。比如，可以按下面的方式准备查询：

```
mysql> INSERT INTO tbl1(col1, col2, col3) VALUES (?, ?, ?) ;
```

接下来可以把状态句柄和每个问号对应的值发送到服务器执行查询。这个过程可以重复任意次。发送状态句柄的具体方式取决于编程语言。其中一种方式是利用 MySQL 为 Java 和 .NET 定制的连接器（Connector）。许多其他客户端类库也有类似的接口，实际使用时请查看具体文档。

使用准备语句会比多次执行查询效率高得多，具体原因如下：

- 服务器只需要解析一次查询，这节约了解析和其他的开销。
- 因为服务器缓存了一部分执行计划，所以它只需要执行某些优化步骤一次。
- 通过二进制发送参数比通过 ASCII 码要快得多。比如，通过二进制发送 `DATE` 类型的参数只需要 3 个字节，但通过 ASCII 码发送需要 10 个字节。节约的效果对于 `BLOB` 和 `TEXT` 类型最为显著，因为它们可以成块地发送，而不是一个个地发送。二进制协议也帮客户端节约了内存，同时减少了网络开销和数据从本身的类型转换为非二进制协议的开销。
- 整个查询不会被发送到服务器，只有参数才会被发送，这减少了网络流量。
- MySQL 直接把参数保存在服务器的缓冲区内，不需要在内存中到处拷贝数据。

准备语句对安全性也有好处。它不需要在应用程序中对值进行转义和加引号，这更加方便，并且减少了 SQL 遭受注入攻击和其他攻击的可能（永远也不能信任用户的输入，即使使用准备语句也不能）。

只有准备语句能使用二进制协议。使用普通的 `mysql_query()` 函数提交查询不会使用二进制协议。许多客户端类库能用问号“准备”查询，然后定义问号的值以执行查询，但是这些类库通常都是在客户端模拟准备语句的，而实际上是把查询用 `mysql_query()` 发送到服务器。

5.4.1 准备语句优化

Prepared Statement Optimization

MySQL 缓存了准备语句的部分执行计划，但是不会预先计算和缓存一些依赖于具体值的优化过程。按照优化执行的时机来划分，它们可以被分为 3 类。下面的列表在写作本书的时候是可用的，但是将来可能会有所不同。

准备时期

服务器解析查询文本、消除否定条件并且重写子查询。

第一次执行时

服务器简化嵌套联接并且把可以转化的外联接转化为内联接。

每次执行时

服务器此时会进行：

- 修整分区。
- 消除可以消除的 `COUNT()`、`MIN()` 和 `MAX()`。
- 移除常量子表达式。
- 探测常量表。
- 传递相等性。
- 分析和优化 `ref`、`range` 和 `index_merge` 等访问方法。
- 优化联接顺序。

要知道更多有关优化的内容，请查阅第 4 章。

5.4.2 准备语句的 SQL 接口

The SQL Interface to Prepared Statements

MySQL 4.1 及以上版本为准备语句提供了 SQL 语言接口。下面是一个具体的例子：

```
mysql> SET @sql := 'SELECT actor_id, first_name, last_name
-> FROM sakila.actor WHERE first_name = ?';
mysql> PREPARE stmt_fetch_actor FROM @sql;
mysql> SET @actor_name := 'Penelope';
mysql> EXECUTE stmt_fetch_actor USING @actor_name;
```

actor_id	first_name	last_name
1	PENELOPE	GUINNESS
54	PENELOPE	PINKETT

104	PENELOPE	CRONYN
120	PENELOPE	MONROE

```
mysql> DEALLOCATE PREPARE stmt_fetch_actor;
```

228 服务器在接收到这些语句时，会对它们进行翻译，最终效果和通过客户端类库执行完全一样。这说明不需要使用特殊的二进制协议去创建和执行准备语句。

如上可见，准备语句的语法比起一般的 SELECT 语句要笨拙一些。那么按照这种方式使用准备语句的好处是什么呢？

它主要用于存储过程。在 MySQL 5.0 中，可以在存储过程中使用准备语句，并且语法和 SQL 接口类似。这意味着可以通过连接字符串在存储过程内部执行“动态 SQL”，增加灵活性。例如，下面是一个可以在特定的数据库中针对每一个表都调用 OPTIMIZE TABLE 函数的存储过程：

```
DROP PROCEDURE IF EXISTS optimize_tables;
DELIMITER //
CREATE PROCEDURE optimize_tables(db_name VARCHAR(64))
BEGIN
    DECLARE t VARCHAR(64);
    DECLARE done INT DEFAULT 0;
    DECLARE c CURSOR FOR
        SELECT table_name FROM INFORMATION_SCHEMA.TABLES
        WHERE TABLE_SCHEMA = db_name AND TABLE_TYPE = 'BASE TABLE';
    DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;
    OPEN c;
    tables_loop: LOOP
        FETCH c INTO t;
        IF done THEN
            CLOSE c;
            LEAVE tables_loop;
        END IF;
        SET @stmt_text := CONCAT("OPTIMIZE TABLE ", db_name, ".", t);
        PREPARE stmt FROM @stmt_text;
        EXECUTE stmt;
        DEALLOCATE PREPARE stmt;
    END LOOP;
    CLOSE c;
END//
DELIMITER ;
```

可以按照下面的方式调用该存储过程：

```
mysql> CALL optimize_tables('sakila');
```

另外一种在存储过程进行循环的方式如下：

```
REPEAT
    FETCH c INTO t;
    IF NOT done THEN
        SET @stmt_text := CONCAT("OPTIMIZE TABLE ", db_name, ".", t);
        PREPARE stmt FROM @stmt_text;
        EXECUTE stmt;
        DEALLOCATE PREPARE stmt;
    END IF;
UNTIL done END REPEAT;
```

229 这两种循环结构之间有重要的区别。REPEAT 为每次循环检查了两次循环条件。在这个例子中，因为它仅仅只检查了一个整型值，所以不会有太大的性能影响，但是对于更复杂的检查，代价可能会比较大。

对于准备语句使用 SQL 接口而言,通过连接字符串来引用表和数据库是一种较好的方式,这样就无须使用参数。其实也不能对数据库和表的名字进行参数化,因为它们本身就是固定的标识符。另外一种情况就是动态设定 LIMIT 子句,它也不能使用参数。

SQL 接口对于手工测试准备语句是有用的,但是在存储过程之外它就不那么有用了。因为这个接口通过 SQL,没有使用二进制协议,并且它不能真正地减少网络流量,当有参数的时候,就不得不启动另外的查询给变量赋值。在某些特殊的情况下,可以通过这个接口得到好处,例如要准备一个巨大的 SQL 语句,它没有参数,需要执行很多次。然而,如果认为准备语句使用 SQL 接口会节约工作的话,应该做实际的评测。

5.4.3 准备语句的局限

Limitations of Prepared Statements

准备语句有下面的一些局限:

- 准备语句只针对一个连接,所以另外的连接不能使用同样的句柄。出于同样的原因,一个先断开再重新连接的客户端会丢失句柄(连接池或持续连接会减轻这个问题)。
- 准备语句不能使用 MySQL 5.0 以前版本的缓存。
- 使用准备语句并不总是高效的。如果只使用一次准备语句,那么准备它花费的时间可能比执行一次平常的 SQL 语句更长。准备语句也需要在服务器和客户端之间进行额外的信息交互。
- 现在不能在存储函数内部使用准备语句,但是可以在存储过程中使用准备语句。
- 如果忘记销毁准备语句,那么就有可能引起资源泄漏。这会消耗相当多的服务器资源。同样,因为对存储语句的数量有一个全局性的限制,所以一个错误可能会干扰其他使用准备语句的连接。

5.5 用户自定义函数

User-Defined Functions

MySQL 支持用户自定义函数(UDF, User-Defined Functions)已经有很长时间了。UDF 和用 SQL 语句写的存储函数不同,它可以用任何支持 C 调用方式的编程语言来写。

用户自定义函数必须被编译,然后动态地和服务器进行链接,使之和平台相关,并且给予你极强的功能。UDF 运行速度很快,并且能访问很多操作系统提供的功能和类库。SQL 存储函数很适合简单操作,比如计算球面上两点之间的最远圆周距离。但是如果发送网络包,就需要使用 UDF。同样,现在不能使用 SQL 语句构建聚集函数,但用 UDF 可以轻易地做到这一点。

UDF 的能力越强,使用它的责任就越重大。UDF 中的一个错误可以使服务器完全崩溃,破坏内存和数据。它就像那些有错误的 C 代码一样危害巨大。



提示: 和 SQL 语言写成的存储函数不同,UDF 现在不能读写表,至少在调用上下文中不行。这意味着它对纯计算,或者是和外部的交互非常有帮助。MySQL 和外部资源的交互正在增加。布莱恩·阿克尔(Brian Aker)和帕特里克·加尔布雷思(Patrick Galbraith)创建的 UDF 是一个极好的例子,它可以用于和数据库缓存服务器(Memcached)进行沟通,网址:http://tangent.org/586/Memcached_Functions_for_MySQL.html。

如果使用了 UDF，升级 MySQL 的时候要仔细地检查版本间的不同，也许需要对它进行重新编译，甚至得做出改变才能和新版本兼容。同时要保证 UDF 是绝对线程安全的，它们在 MySQL 服务器进程里面运行，那是纯粹的多线程环境。

网上有很多已经预先编译好的 UDF 类库，也有很多示例说明了如何创建自己的 UDF。最大的 UDF 存储库在：<http://www.mysqludf.org>。

下面是用于计算复制速度的 NOW_USEC() 函数（请查看第 405 页的“复制有多快”）。

```
#include <my_global.h>
#include <my_sys.h>
#include <mysql.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>

extern "C" {
    my_bool now_usec_init(UDF_INIT *initid, UDF_ARGS *args, char *message);
    char *now_usec(
        UDF_INIT *initid,
        UDF_ARGS *args,
        char *result,
        unsigned long *length,
        char *is_null,
        Char *error);
}

my_bool now_usec_init(UDF_INIT *initid, UDF_ARGS *args, char *message) {
    return 0;
}

char *now_usec(UDF_INIT *initid, UDF_ARGS *args, char *result,
               unsigned long *length, char *is_null, char *error) {

    struct timeval tv;
    struct tm* ptm;
    char time_string[20]; /* e.g. "2006-04-27 17:10:52" */
    char *usec_time_string = result;
    time_t t;

    /* Obtain the time of day, and convert it to a tm struct. */
    gettimeofday (&tv, NULL);
    t = (time_t)tv.tv_sec;
    ptm = localtime (&t);

    /* Format the date and time, down to a single second. */
    strftime (time_string, sizeof (time_string), "%Y-%m-%d %H:%M:%S", ptm);

    /* Print the formatted time, in seconds, followed by a decimal point
     * and the microseconds. */
    sprintf(usec_time_string, "%s.%06ld\n", time_string, tv.tv_usec);

    *length = 26;

    return(usec_time_string);
}
```

5.6 视图

Views

视图（View）是很普通的概念，MySQL 5.0 包含了这一特性。MySQL 视图就像一张表，但其自身不会存储任何数据，它的数据来自 SQL 查询语句。

本书不会介绍如何创建或使用视图，MySQL 手册包含了创建视图的细节，而其他很多文档也解释了如何使用它。出于很多考虑，MySQL 把视图当成表来看待，并且两者共享了同样的命名空间。但是 MySQL 处理它们的方式并不完全一样。例如，不能在视图上创建触发器，也不能用 DROP TABLE 命令删除视图。

重要的是理解视图的内部机制，以及和查询优化器交互的方式，否则就别指望通过它得到好的性能。下面通过 world 数据库展示一下如何使用视图：

```
mysql> CREATE VIEW Oceania AS
->     SELECT * FROM Country WHERE Continent = 'Oceania'
->     WITH CHECK OPTION;
```

执行视图最简单的方式就是执行其中的 SQL 语句，然后把结果放到临时表里面。在查询中使用视图的时候，就可以引用临时表里面的数据。为了解它如何工作，请看下面的例子：

```
mysql> SELECT Code, Name FROM Oceania WHERE Name = 'Australia';
```

下面是服务器的执行过程，临时表的名字只是出于演示的目的：

```
mysql> CREATE TEMPORARY TABLE TMP_Oceania_123 AS
->     SELECT * FROM Country WHERE Continent = 'Oceania';
mysql> SELECT Code, Name FROM TMP_Oceania_123 WHERE Name = 'Australia';
```

这种方式很明显有性能和优化方面的问题。一种较好的使用方式是重写查询，将视图的 SQL 语句与查询语句合并。下面是合并后的例子：

```
mysql> SELECT Code, Name FROM Country
->     WHERE Continent = 'Oceania' AND Name = 'Australia';
```

MySQL 可以使用这两种执行方式，它们分别调用了合并算法（MERGE Algorithm）和临时表算法（TEMPTABLE（注 6）Algorithm）。MySQL 会优先考虑使用合并算法。它甚至可以合并嵌套的视图。可以通过 EXPLAIN EXTENDED 命令，加上 SHOW WARNINGS 参数查看重写后的查询。

如果使用临时表算法，解释器通常会把它显示为 DERIVED 表。图 5-4 显示了这两种执行方式。

在视图包含 GROUP BY、DISTINCT、聚集函数、UNION、子查询或其他无法保持视图返回的行和待查询的行之间一对一关系的结构的时候，MySQL 就会使用临时表算法。其实并非只有在上面的情况下才会使用临时表算法，具体情况将来也许还会发生改变。如果想知道一个视图使用的是合并算法还是临时表算法，使用解释器解释一下即可：

```
mysql> EXPLAIN SELECT * FROM <view_name>;
+----+-----+
| id | select_type |
+----+-----+
| 1  | PRIMARY    |
| 2  | DERIVED     |
+----+-----+
```

注 6：它指临时表（Temp Table），而不是指有诱惑力（Can be tempted）。

选择类型是 DERIVED，这表明视图将会使用临时表算法。

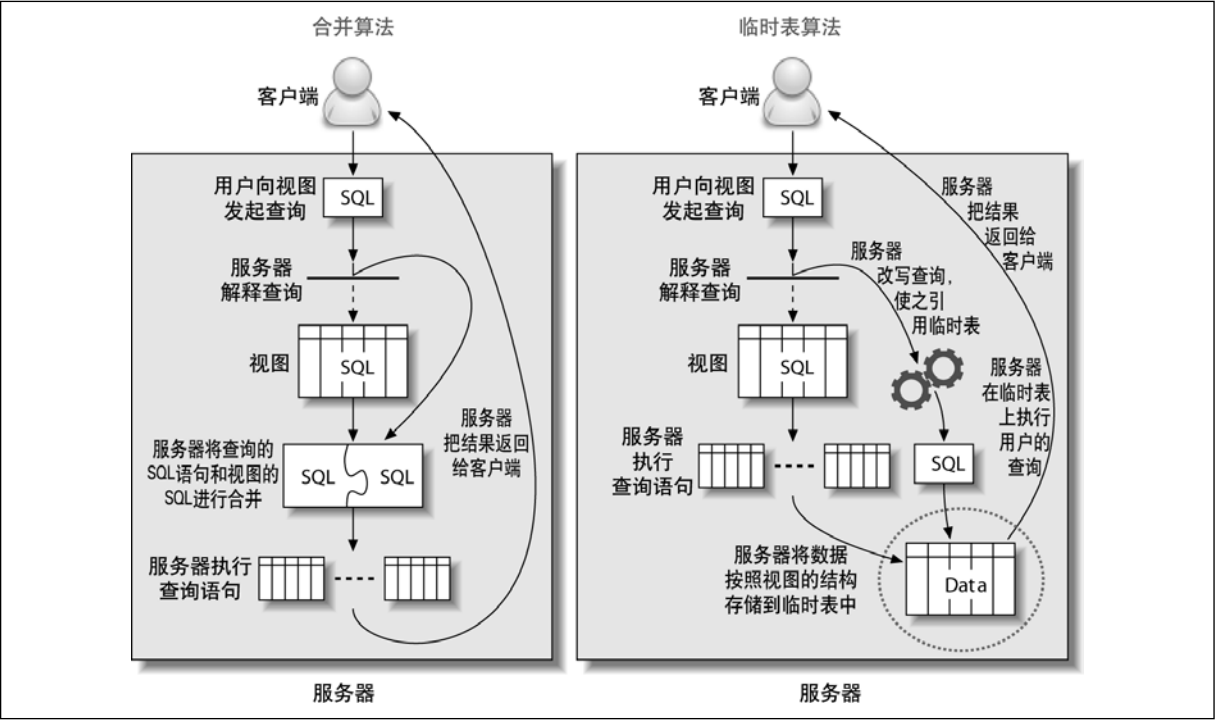


图 5-4：视图的两种执行方式

5.6.1 可更新视图

Updatable Views

可更新视图可以通过视图更新它所引用的表。在某些特定的条件下，可以对视图使用 UPDATE、DELETE，甚至 INSERT 语句。下面的例子是一个有效的操作：

```
mysql> UPDATE Oceania SET Population = Population * 1.1 WHERE Name = 'Australia';
```

234 视图如果含有 GROUP BY、UNION、聚集函数或任何其他的异常，就不是可更新的。改变数据的查询也许包含了联接，但是被改变的列必须在同一个表里面。使用 TEMPTABLE 算法的视图是不可更新的。

上一节创建视图时使用的 CHECK OPTION 子句，保证了通过视图改变的行和视图中 WHERE 子句始终匹配。所以，不能改变 Continent 列，也不能插入有不同 Continent 的行，任何一种情况都会导致服务器报告下面的错误：

```
mysql> UPDATE Oceania SET Continent = 'Atlantis';
ERROR 1369 (HY000): CHECK OPTION failed 'world.Oceania'
```

一些数据库运行在视图上面使用 INSTEAD OF 触发器，因此可以精确地定义当语句修改视图数据时会发生什么。但是 MySQL 不支持在视图上面使用触发器。一些关于可更新视图的限制在将来会被解决掉，一种可能性是在表上面用不同的存储引擎构造合并表。这也许会成为使用视图既高效又有用的方式。

5.6.2 视图对性能的影响

Performance Implications of Views

许多人不认为视图可以改进性能，但是它确实可以提高性能，也可以用它来支持其他提高性能的方式。例如，利用视图重构数据库架构的某一阶段，可以在更改它访问的表的同时，使代码继续工作。

一些应用程序为每一个用户使用一个表，这通常是为了实现行级别安全性。一个和前面例子类似的视图能够在表内实现类似的安全性，并且打开表的次数会更少，这有助于提高性能。被很多用户使用的许多开源工程已经累积了上百万张表，它们可以从这种方式得益。下面有一个假想的博客系统数据库：

```
CREATE VIEW blog_posts_for_user_1234 AS
  SELECT * FROM blog_posts WHERE user_id = 1234
  WITH CHECK OPTION;
```

可以使用视图实现列级权限，但是却没有实际创建这些权限的开销。列级权限也会阻止查询被缓存。视图可以限制访问想要的列，但却不会导致这些问题：

```
CREATE VIEW public.employeeinfo AS
  SELECT firstname, lastname -- but not socialsecuritynumber
  FROM private.employeeinfo;
GRANT SELECT ON public.* TO public_user;
```

有时可以使用伪临时视图取得好的效果。这不用创建一个只在当前连接中存在的真正临时视图，而是用一种特殊的方式，比如在特定的数据库中，创建一个视图，在使用完之后就可以把它删除。接下来就可以在 FROM 子句中使用这个视图，就像在 FROM 子句中使用子查询一样。这两种方式在理论上是同样的，但是 MySQL 对视图有不同的代码库（Codebase），所以通过临时视图可能会得到更好的性能。下面是一个例子：

```
-- Assuming 1234 is the result of CONNECTION_ID( )
CREATE VIEW temp.cost_per_day_1234 AS
  SELECT DATE(ts) AS day, sum(cost) AS cost
  FROM logs.cost
  GROUP BY day;

SELECT c.day, c.cost, s.sales
FROM temp.cost_per_day_1234 AS c
  INNER JOIN sales.sales_per_day AS s USING(day);

DROP VIEW temp.cost_per_day_1234;
```

要注意到使用了连接 ID 作为唯一的后缀，以避免名字冲突。这种方式在应用程序崩溃并无法删除临时视图时可以比较方便地进行清理。请查看第 394 页的“丢失的临时表”了解更多细节。

使用了临时表算法的视图性能可能会很差（尽管它们比没有使用视图的同样查询性能会好一些）。MySQL 在优化外部查询的时候就使用递归的步骤执行了它，这时外部查询的优化还没有完成，所以它没有得到完全的优化。构造临时表的查询不会从外部查询中得到 WHERE 条件，并且临时表没有索引。下面仍然使用 temp.cost_per_day_1234 举一个例子：

```
mysql> SELECT c.day, c.cost, s.sales
-> FROM temp.cost_per_day_1234 AS c
->   INNER JOIN sales.sales_per_day AS s USING(day)
->   WHERE day BETWEEN '2007-01-01' AND '2007-01-31';
```

在服务器执行这个视图，把结果放到临时表，然后联接 sales_per_day 表的过程中，到底发生了什么呢？WHERE 子句中的 BETWEEN 限制没有被“推送”到视图中，所以视图会为表中的所有日期创建结果，这不会有什么问题，

因为服务器将会把临时表排在联接的第一位，所以联接就可以使用 `sales_per_day` 表的索引。但是，如果将这样的两个视图进行联接，那么就不会有任何索引的优化了。

236 如果想用视图来提高性能，那么总是要进行测试，至少也应该进行分析。甚至合并视图也会增加开销，并且很难预测视图如何影响性能。如果有性能问题，永远也不要猜测，一定要进行测量。

视图引入了一些并非只有 MySQL 才有的问题。视图会误导开发人员，认为它非常简单，但实际上它非常复杂。某些查询看上去是反复查询一个表，但实际是查询一个代价很高的视图，一个不理解它潜在复杂性的开发人员会对这个现象无动于衷。我们曾经看到过一个很简单的查询在解释器中输出了上百条记录，因为其中某些表其实是视图，而它们又引用了很多其他的表和视图。

5.6.3 视图的局限

Limitations of Views

MySQL 不支持物化视图 (Materialized View)。物化视图通常把结果存储在一个不可见的表里面，然后周期性地从原始数据对不可见表进行刷新。MySQL 也不支持索引视图 (Indexed View)，可以通过创建缓存表和汇总表模拟物化视图和索引视图。但是在 MySQL 5.1 中，可以使用事件来调度这些任务。

MySQL 对视图的实现有一些让人烦恼的地方。最大的问题是 MySQL 不会保持视图的原始 SQL 语句。如果试着使用 `SHOW CREATE VIEW` 命令把视图显示出来，然后对它进行编辑，你会非常惊讶。视图被展开成了规范化的内部格式，不能对它进行格式化、添加注释和缩进。

如果想按照创建视图时的格式来编辑视图，可以打开视图的 `.frm` 文件，最后一行就是视图的文本。如果有 `FILE` 权限，并且 `.frm` 文件是可读取的，那么就可以通过 `LOAD_FILE()` 函数加载文件的内容。加载完毕后，进行一些字符串的操作就可以把语句原封不动地提取出来了，这儿要感谢 Roland Bouman 的创新性工作：

```
mysql> SELECT
-> REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(
-> REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(
-> SUBSTRING_INDEX(LOAD_FILE('/var/lib/mysql/world/Oceania.frm'),
-> '\nsource=', -1),
-> '\\_', '\\_'), '\\%', '\\%'), '\\\\', '\\\\'), '\\z', '\\z'), '\\t', '\\t'),
-> '\\r', '\\r'), '\\n', '\\n'), '\\b', '\\b'), '\\\\', '\\\\'), '\\\\', '\\\\'),
-> '\\0', '\\0')
-> AS source;
```

```
237 +-----+
| source |
+-----+
| SELECT * FROM Country WHERE continent = 'Oceania'
| WITH CHECK OPTION
|
+-----+
```

5.7 字符集和排序规则

Character Sets and Collations

字符集 (Character Sets) 是从二进制编码到规定符号集的映射。可以把它看作如何用位来代表特定的字符。排序规则 (Collations) 是字符集排序原则的集合。在 MySQL 4.1 及其后续版本中，每个基于字符的值都有字符集

和排序规则（注 7）。MySQL 对字符集和排序规则的支持是世界级的，但是它也会增加复杂性，在某些情况下还会有一些性能开销。

本节讨论了在大多数情况下所需的设置和功能。如果需要了解更多的细节，请查阅 MySQL 手册。

5.7.1 MySQL 如何使用字符集

How MySQL Uses Character Sets

字符集有几种排序规则，并且每种字符集都有默认的排序规则。某个字符集特定的排序规则不能被用于其他字符集。可以将字符集和排序规则合在一起使用，所以从现在开始，我们使用字符集代指它们。

MySQL 有不同的选项控制字符集。这些选项和字符集很容易让人迷惑，所以记住它们的区别：只有基于字符的值才有字符集。其余的任何东西都只是规定使用何种字符集来进行比较或其他操作。基于字符的值可以是存储在列中的值、查询中使用的字面常量、表达式的结果、用户变量等。

MySQL 的设置可以分为两类：创建对象时的默认设置，以及用于控制服务器和客户端沟通的设置。

创建对象时的默认设置

MySQL 对服务器的每个数据库、每个表都有默认的字符集和排序规则。这形成了创建列时影响其字符集的默认值的继承关系。反过来说，它也告诉了服务器列存储的值使用了何种字符集。

在继承的每一个层次，都可以显式地定义字符集或让服务器使用默认值：

- 当创建一个数据库的时候，它从服务器继承了 `character_set_server` 设置。
- 当创建表的时候，它从数据库继承字符集。
- 当创建列的时候，它从表继承字符集。

记住，MySQL 存储数据的唯一地方就是列，所以继承的较高层次只有默认值。表的默认字符集不会影响到表里面存储的数据，它只是告诉 MySQL 在创建列的时候如果没有指定字符集，那么就使用该默认值。

用于客户端/服务器沟通的设置

当服务器和客户端彼此进行沟通的时候，它们也许会使用不同的字符集来回传递数据。服务器将会按需进行翻译：

- 服务器假设客户端正在按照 `character_set_client` 设置的字符集发送数据。
- 服务器从客户端收到语句后，它按照 `character_set_connection` 设置的字符集对数据进行翻译，它也会用这个字符集决定如何把数字转换为字符串。
- 当服务器把结果或错误信息返回给客户端时，它会按照 `character_set_result` 定义的字符集进行翻译。

图 5-5 展示了这一过程。

注 7：MySQL 4.0 及之前的版本对整个服务器采用了全局设置，并且只能从几个 8 位字符集中进行选择。

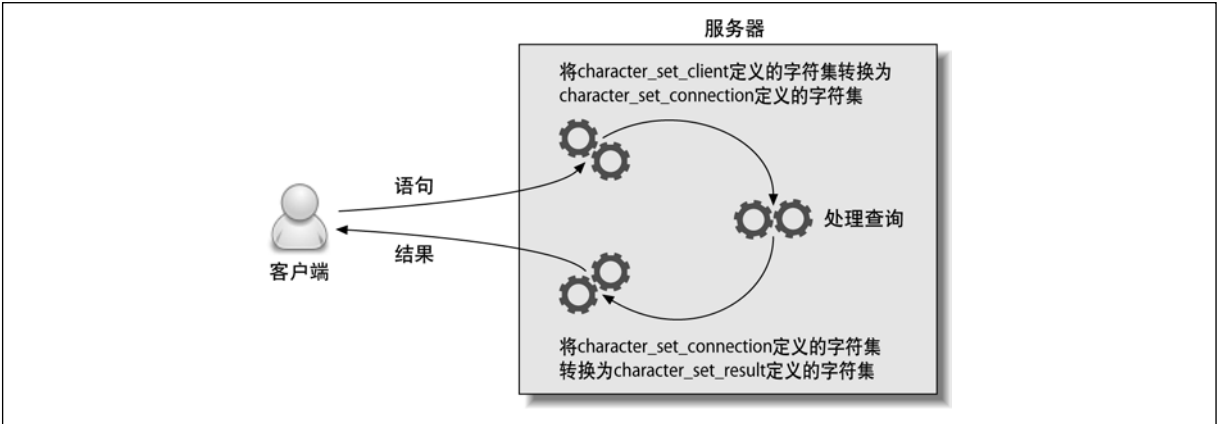


图 5-5：客户端和服务端字符集

可以使用 `SET NAMES` 命令，或者 `SET CHARACTER SET` 命令按照自己的需要改变上面的 3 个设置。但是，这两个命令只会影响服务器的设置。客户端程序和 API 也需要正确地进行设置，以避免和服务器的沟通问题。

239 假设使用 `latin1` 字符集打开了一个连接（这是默认的字符集，除非用 `mysql_options()` 改变了它），然后使用 `SET NAMES utf8` 告诉服务器客户端将会用 `utf8` 字符集发送数据。这样的话，就造成了字符集不匹配的问题，它会导致错误，甚至安全性问题。此时应该设置客户端字符集并在需要转义的时候使用 `mysql_real_escape_string()`。在 PHP 中，可以使用 `mysql_set_charset()` 改变客户端的字符集。

MySQL 如何比较值

当 MySQL 比较两个使用了不同字符集的值时，它必须把它们转换为同一字符集。如果字符集不兼容，就会引发错误，例如“错误 1267 (HY000)：不合法的排序规则混合。(ERROR 1267 (HY000): Illegal mix of collations)”在这种情况下，通常需要使用 `CONVERT()` 函数显式地把值转换为兼容的字符集格式。这个错误在 MySQL 4.1 中更为常见。

MySQL 也能对值采取强制措施。它决定了值采用的字符集的优先级并且影响了 MySQL 会隐式转换的值。可以使用 `CHARSET()`、`COLLATION()` 和 `COERCIBILITY()` 函数来调试和字符集及排序规则有关的问题。

可以使用导引符 (Introducer) 和排序规则子句 (Collate Clauses) 来定义字面常量的字符集和排序规则，如下例：

```
mysql> SELECT _utf8 'hello world' COLLATE utf8_bin;
+-----+
| _utf8 'hello world' COLLATE utf8_bin |
+-----+
| hello world                          |
+-----+
```

特殊行为

MySQL 字符集的行为有一些出人意料的地方。下面是一些值得注意的事项：

难以捉摸的 `character_set_database` 设置

`character_set_database` 的默认值是默认数据库的值。当改变默认数据库的时候，它也会跟着改变。如果没有默认数据库，它的默认值就是 `character_set_server`。

LOAD DATA INFILE 按照当前 character_set_database 的设置解释接收到的数据。一些版本的 MySQL 在 LOAD DATA INFILE 语句中接受可选的 CHARACTER SET 子句，但是不应该依赖于它。

取得可靠结果的最佳方式是用 USE 命令使用需要的数据库，并执行 SET NAMES 选择一个字符集，最后再加载数据。不管列的字符集是什么，MySQL 都会按照同一种字符集解释数据。

SELECT INTO OUTFILE

MySQL 利用 SELECT INTO OUTFILES 无转换地写入数据。如果不用 CONVERT 函数包装列，就无法设置数据的字符集。

嵌入的转义序列

MySQL 按照 character_set_client 的设置解释转义序列，即使有导引符或者排序规则子句也是这样。因为解析器是以字面常量的方式解析转义序列的。到目前为止，解析器没有排序的意识，它不会把导引符看成一种指令，而仅仅是一个标识。

5.7.2 选择字符集和排序规则

Choosing a Character Set and Collation

MySQL 4.1 及其以上版本支持大量的字符集和排序规则，包括使用 Unicode 字符集的 UTF-8 编码支持多字节字符（MySQL 支持完整的 UTF-8 三字节子集，它能存储世界上的绝大多数语言）。可以使用 SHOW CHARACTER SET 及 SHOW COLLATION 命令查看 MySQL 支持的字符集。

最常见的排序规则是利用大小写或字母的二进制编码进行排序。排序规则通常用 _cs、_ci 或_bin 结尾，这样就能很轻易地分辨它们。

在显式地定义字符集的时候，并不需要同时定义字符集和排序规则。如果忽略了其中一个，或者两个全部都被忽略了，MySQL 就会使用应用程序的默认值补上。表 5-2 显示了 MySQL 如何决定使用的字符集和排序规则。

表 5-2: MySQL 如何决定字符集和排序规则的默认值

如果定义	结果字符集	结果排序规则
字符集和排序规则	按照定义	按照定义
只有字符集	按照定义	字符集的默认排序规则
只有排序规则	拥有该排序规则的字符集	按照定义
都没有	可用的默认值	可用的默认值

下面的命令显示了如何利用特定的字符集和排序规则创建数据库、表和列。

```
CREATE DATABASE d CHARSET latin1;
CREATE TABLE d.t(
  col1 CHAR(1),
  col2 CHAR(1) CHARSET utf8,
  col3 CHAR(1) COLLATE latin1_bin
) DEFAULT CHARSET=cp1251;
```

结果表的列有下面的排序规则：

```
mysql> SHOW FULL COLUMNS FROM d.t;
+-----+-----+-----+
| Field | Type   | Collation          |
+-----+-----+-----+
| col1  | char(1) | cp1251_general_ci |
| col2  | char(1) | utf8_general_ci   |
| col3  | char(1) | latin1_bin         |
+-----+-----+-----+
```

保持简单

在数据库中混合多种字符集确实是很糟糕的事情。不兼容的字符集会让人非常困惑。它们甚至只会在某些字符出现的时候才会出问题，到那时，所有的操作都会出错（比如表联接）。只能使用 ALTER TABLE 命令将列转换为兼容的字符集，或者在 SQL 语句中使用引导符或排序规则子句将值转化为需要的字符集。

为了让你的头脑保持清醒，最好的选择就是在服务器级选择合适的默认值，也许在数据库级也可以，然后就可以在列这一级具体问题具体分析。

5.7.3 字符集和排序规则如何影响查询

How Character Sets and Collations Affect Queries

一些字符集需要更多的 CPU 操作、消耗更多的内存和存储空间，甚至会使索引失效。因此要仔细地选择字符集和排序规则。

在字符集和排序规则之间做转化会增加某些操作的开销。例如，sakila.film 表在 title 列上有索引，它可以用来加速 ORDER BY 查询：

```
mysql> EXPLAIN SELECT title, release_year FROM sakila.film ORDER BY title\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: film
         type: index
possible_keys: NULL
         key: idx_title0
        key_len: 767
         ref: NULL
         rows: 953
       Extra:
```

但是，只有在索引的排序规则和查询定义的规则一致的情况下服务器才能使用索引。索引是按照列的排序规则进行排序的，在本例中它是 utf8_general_ci。如果想使用另外一种排序规则，服务器就不得不进行文件排序：

```
mysql> EXPLAIN SELECT title, release_year
-> FROM sakila.film ORDER BY title COLLATE utf8_bin\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: NULL
         key: NULL
        key_len: NULL
         ref: NULL
         rows: 953
       Extra: Using filesort
```

除了适应联接的默认字符集和在查询中显式定义的字符集之外, MySQL 还需要转换字符集, 以进行比较。例如, 使用字符集不同的列联接两个表, MySQL 就不得不在它们之间做转换。这种转换使联接无法利用索引, 因为它就像使用一个函数对列进行封装一样。

UTF-8 多字节字符集以可变字节数 (1 字节到 3 字节) 来保存每个字符。MySQL 内部许多字符串操作使用了固定大小的缓冲区, 所以它必须分配足够的空间, 以容纳最大的可能长度。例如, CHAR(10) 使用 UTF-8 进行编码, 即使实际的字符串没有所谓的宽字符, 也需要 30 个字节。可变长度字段 (VARCHAR, TEXT) 保存在磁盘上时不会有这样的问题。但是内存中的临时表总是分配需要的最大长度。

在多字节字符集中, 一个字母不再是一个字节。因此, MySQL 有 LENGTH() 和 CHAR_LENGTH() 函数, 它们不会对多字节字符返回同样的结果。在使用多字节字符集的时候, 要确保在统计字符 (比如执行 SUBSTRING() 操作) 的时候使用 CHAR_LENGTH() 函数。这个问题同样也存在于应用程序语言中。

另外一个出人意料的地方就是索引限制。如果索引了一个使用了 UTF-8 字符集的列, MySQL 就会假设每个字符都会是 3 字节, 所以平常的长度限制突然就减少为三分之一:

```
mysql> CREATE TABLE big_string(str VARCHAR(500), KEY(str)) DEFAULT CHARSET=utf8;
Query OK, 0 rows affected, 1 warning (0.06 sec)
mysql> SHOW WARNINGS;
+-----+-----+-----+
| Level | Code | Message |
+-----+-----+-----+
| Warning | 1071 | Specified key was too long; max key length is 999 bytes |
+-----+-----+-----+
```

注意到 MySQL 自动地将索引的长度减少到 333 个字符。

```
mysql> SHOW CREATE TABLE big_string\G
***** 1. row *****
      Table: big_string
Create Table: CREATE TABLE `big_string` (
  str` varchar(500) default NULL,
  KEY `str` (`str`(333))
) ENGINE=MyISAM DEFAULT CHARSET=utf8
```

如果没有注意到警告信息并且检查表的定义, 就不会认识到索引仅仅是列的前缀。这会有一些副作用, 例如会禁止覆盖索引。

某些人推荐在所有的地方都使用 UTF-8。但是, 如果在意性能的话, 这不是一个好主意。许多应用程序根本不需要 UTF-8, 并且使用何种字符集依赖于数据。UTF-8 使用更多的磁盘空间。

在决定字符集的时候, 重要的是考虑将要存储的数据。例如, 如果存储英文文本, UTF-8 不会有问题, 因为大部分英文字符在 UTF-8 里面都是用一个字节存储的。在另一方面, 如果保存阿拉伯文或俄文这样的非拉丁语言, 情况就会很不一样。一个只需要存储阿拉伯语的应用程序语言可以使用 cp1256 字符集, 它可以用一个字节包含所有的阿拉伯字符。但是如果程序需要保存许多不同的语言并且选择了 UTF-8, 那么同样的阿拉伯字符就会使用更多的存储空间。同样地, 如果将某个国家的字符集转换为 UTF-8, 存储空间将会极大地上升。如果使用 InnoDB, 就有可能把数据大小增加到超出页面限制的地步, 并且需要外部存储空间。这会造成存储空间大量浪费并且导致碎片。更多细节请参阅第 298 页的“优化 BLOB 和 TEXT 工作负载”。

有时根本不需要使用字符集。字符集对于不区分大小写比较、排序, 以及 SUBSTRING() 这样的字符串操作非常有用。如果不需要数据库服务器处理字符, 就可以把任何数据, 包括 UTF-8 数据, 存储在 BINARY 列中。如果

这么做的话，就需要加一列标明用来编码的字符集。尽管有些人使用这种方法已经很久了，但是还是需要更小心一点。如果你忘记了一个字节并不一定是一个字符，它就能导致很难捕获的错误，比如和 `SUBSTRING()` 和 `LENGTH` 相关的错误。在实际工作中，我们建议你最好避免这种方式。

5.8 全文搜索

Full-Text Searching

大部分的查询都可能包含 `WHERE` 子句，用于比较相等性、过滤数据等。但是，也许还需要执行关键字搜索，它基于数据的关联性，而不是相互比较。全文搜索系统就是为这个目的而设计的。

全文搜索（Full-Text Searching）需要特殊的查询语法。无论有没有索引，它都能工作，但是索引可以加速匹配的过程。全文搜索使用的索引有特殊的结构，可以帮助发现含有所要关键字的文档。

你也许不知道全文索引，但是你至少知道一种全文索引引擎：因特网搜索引擎。尽管它们规模很大，而且后端并不使用关系数据库，但是基本原则是一样的。

在 MySQL 中，只有 MyISAM 存储引擎支持全文索引。可以在上面搜索基于字符的内容（`CHAR`、`VARCHAR` 和 `TEXT` 列），并且它支持自然语言搜索和布尔搜索。全文搜索的实现有很多限制和缺陷（注 8），并且它非常复杂，但是它仍然被广泛地使用，因为它被包含在服务器中，而且能满足许多应用程序的要求。本节广泛地讨论了如何使用它，以及如何在有全文搜索的情况下，为得到好的性能而进行设计。

MyISAM 全文索引操作了一个全文集合（Full-text Collection），它由单个表中的一个或多个字符列组成。实际上，MySQL 在集合中通过联接列构造索引，并且把它们当成很长的字符串进行索引。

MyISAM 全文索引是一种特殊的具有两层结构的 B 树。第 1 层保存了关键字，然后，对每个关键字，第 2 层包含了一个列表，它由相关的文档指针组成，这些指针指向包含该关键字的全文集合。索引不会包含集合中的每一个词，它按照下面的方式进行调整：

- 一个停用字（Stopword）清单把某些无意义的词剔除了，这样它们就不会被索引。停用字列表基于常用的英语用法，但是可以使用 `ft_stopword_file` 选项用一个外部列表替换掉它。
- 除非一个词长度大于 `ft_min_word_len` 并且小于 `ft_max_word_len`，否则它就会被忽略掉。

全文索引没有存储关键字发生的列信息，所以如果要对组合的列进行搜索，就要创建多个索引。

这也意味着不能使用 `MATCH AGAINST` 子句来指定某些列里面的词比另外列里面的词更重要。这在构建网站的搜索引擎时其实是很常见的。比如，我们可能会需要当关键字出现在标题里面时，结果就显示在前面。如果需要这种特性，就不得不写一些更复杂的查询（稍后会有示例）。

5.8.1 自然语言全文搜索

Natural-Language Full-Text Searches

自然语言搜索查询决定了每个文档和查询的关联性。关联性基于匹配的单词数量，以及它们在文档中出现的频率。

注 8：也许会发现 MySQL 的全文搜索引擎的限制让它在应用程序中根本就不可用。在附录 C 中讨论了使用 Sphinx 作为外部的全文搜索引擎。

在所有索引中出现得越少的单词，说明搜索的关联性越高。相反，极其常见的单词根本就不值得搜索。对于那些在多于一半的行里面出现的单词，即使它们没有被列在停用词列表（Stop Words List）里面，自然语言全文搜索也会把它们排除掉（注 9）。

全文搜索的语法和其他类型的查询有一点点不同。可以通过使用带有 `MATCH AGAINST` 的 `WHERE` 子句告诉 MySQL 执行全文搜索。下面是一个例子。在标准的 `sakila` 数据库里面，`film_text` 表在 `title` 和 `description` 列上面有全文索引：

```
mysql> SHOW INDEX FROM sakila.film_text;
```

Table	Key_name	Column_name	Index_type
...			
film_text	idx_title_description	title	FULLTEXT
film_text	idx_title_description	description	FULLTEXT

下面是使用自然语言全文搜索查询的例子：

```
mysql> SELECT film_id, title, RIGHT(description, 25),
-> MATCH(title, description) AGAINST('factory casualties') AS relevance
-> FROM sakila.film_text
-> WHERE MATCH(title, description) AGAINST('factory casualties');
```

film_id	title	RIGHT(description, 25)	relevance
831	SPIRITED CASUALTIES	a Car in A Baloon Factory	8.4692449569702
126	CASUALTIES ENCINO	Face a Boy in A Monastery	5.2615661621094
193	CROSSROADS CASUALTIES	a Composer in The Outback	5.2072987556458
369	GOODFELLAS SALUTE	d Cow in A Baloon Factory	3.1522686481476
451	IGBY MAKER	a Dog in A Baloon Factory	3.1522686481476

MySQL 会将搜索字符串拆成单词，然后逐个地和 `title` 及 `description` 字段进行匹配，这两个字段已经按照自己的索引被加入到了全文集合中。要注意到只有一个结果包含了这两个单词，并且 3 个包含“casualties”的结果被首先列了出来（整个表中也只有这 3 个含 casualties 的结果）。这是因为索引是按照相关性从高到低排列的。



提示：和普通的查询不同，全文搜索结果自动地按相关性进行了排序。MySQL 不能在全文搜索的时候使用索引来排序。因此，如果想避免文件排序，就不要使用 `ORDER BY` 子句。

就像上例中显示的那样，`MATCH()` 函数实际返回表示相关性的浮点数。可以用这个值按照相关性进行过滤或者在用户界面中显示。如果定义了 `MATCH()` 函数两次，也不会有额外的开销，MySQL 知道它们是同样的函数，只会执行一次。但是，如果在 `ORDER BY` 子句中使用了 `MATCH()`，MySQL 会使用文件排序来对结果进行排序。

必须在 `MATCH` 子句中明确地表示列，不管这些列是否在全文索引中被定义，或者 MySQL 根本就没使用索引。这是因为索引没有记录关键字在哪个列中出现过。

正如前文，这也意味着不能使用全文搜索定义某个关键字在特定的列中出现。但是也有变通的办法。可以在某些列上添加多个索引，然后利用索引的组合进行自定义排序。假设有想让 `title` 列更重要，那么可以在这个列上再

注 9：在测试中常见的错误就是只在全文搜索索引中加入很少的数据行，这样查询就不会找到任何结果。这是因为每个单词都在超过了一半的行中出现过。

加一个索引：

```
mysql> ALTER TABLE film_text ADD FULLTEXT KEY(title) ;
```

现在可以两次使用 title 来进行想要的排序：

```
mysql> SELECT film_id, title, RIGHT(description, 25),
-> ROUND(MATCH(title, description) AGAINST('factory casualties'), 3)
-> AS full_rel,
-> ROUND(MATCH(title) AGAINST('factory casualties'), 3) AS title_rel
-> FROM sakila.film_text
-> WHERE MATCH(title, description) AGAINST('factory casualties')
-> ORDER BY (2 * MATCH(title) AGAINST('factory casualties'))
-> + MATCH(title, description) AGAINST('factory casualties') DESC;
```

title	RIGHT(description, 25)	full_rel	title_rel
831	a Car in A Baloon Factory	8.469	5.676
126	Face a Boy in A Monastery	5.262	5.676
299	jack in The Sahara Desert	3.056	6.751
193	a Composer in The Outback	5.207	5.676
369	d Cow in A Baloon Factory	3.152	0.000
451	a Dog in A Baloon Factory	3.152	0.000
595	a Cat in A Baloon Factory	3.152	0.000
649	nizer in A Baloon Factory	3.152	0.000

但是这通常效率不高，因为它使用了文件排序。

5.8.2 布尔全文搜索

Boolean Full-Text Searches

在布尔搜索中，查询自身定义了匹配单词的相对相关性。布尔搜索使用了停用词表（Stopword List）来过滤无用的单词，但是要禁用单词的长度必须大于 ft_min_word_len 且小于 ft_max_word_len 这一选项。布尔搜索的结果是没有排序的。

在构造一个布尔搜索查询的时候，可以使用前缀来修改搜索字符串中每个关键词的相对排名。最常用的修饰符在表 5-3 中。

表 5-3：布尔全文搜索常用修饰符

示例	含义
dinosaur	含有“dinosaur”的行排名较高
~dinosaur	含有“dinosaur”的行排名较低
+dinosaur	行必须含有“dinosaur”
-dinosaur	行不能含有“dinosaur”
dino*	含有以“dino”打头的单词的行排名较高

也可以使用其他的操作符，比如使用括号进行分组。可以用这种方式构造复杂的搜索。

还是举一个例子，仍然搜索 sakila.film_text 表，找到含有“factory”和“casualties”的电影。自然语言搜索会返回含有其中一个单词或包含了这两个单词的结果。但是，这儿使用的布尔搜索要求结果同时包含这两个单词：

```
mysql> SELECT film_id, title, RIGHT(description, 25)
-> FROM sakila.film_text
-> WHERE MATCH(title, description)
-> AGAINST('+factory +casualties' IN BOOLEAN MODE);
```

film_id	title	right(description, 25)
831	SPIRITED CASUALTIES	a Car in A Baloon Factory

也可以把单词用引号引起来，执行短语搜索，这要求精确匹配该短语。

```
mysql> SELECT film_id, title, RIGHT(description, 25)
-> FROM sakila.film_text
-> WHERE MATCH(title, description)
-> AGAINST('"spirited casualties"' IN BOOLEAN MODE);
```

film_id	title	right(description, 25)
831	SPIRITED CASUALTIES	a Car in A Baloon Factory

短语搜索很慢。只靠全文索引无法响应这种搜索，因为索引没有在原始的全文集合中记录单词之间的相对位置。这样造成的结果就是服务器不得不到行内部去执行单词搜索。

为了执行这种搜索，服务器将会查找所有含有“spirited”和“casualties”的文档。然后它会从这些文档中提取行，并且精确地匹配该短语。因为它使用了索引查找最开始使用的文档，所以你可能认为这会很快，至少比 LIKE 操作快得多。实际上，只要该短语并不常见，而且不会返回很多结果，它确实很快。如果短语非常常见，LIKE 实际会快一些，因为它会顺序读取数据，而不会使用索引排序的二次算法，并且它根本就不需要读取全文索引。

布尔全文搜索实际不需要全文索引。如果有全文索引的话，它就会使用索引，如果没有的话，它就会扫描整个表。甚至可以对多个表使用布尔全文搜索，例如对联接的结果进行搜索。但是在所有的情况下，它都很慢。

5.8.3 MySQL 5.1 及以上版本中全文搜索的变化

Full-Text Changes in MySQL 5.0 and Beyond

MySQL 5.1 引入了好些关于全文搜索的新变化。其中包括性能改进和构建用于改进内建功能的可插拔解析器。例如，插件可以改变索引的工作方式。它们可以比默认设置更灵活地把文本分割成单词，例如可以把 C++ 当成一个单词，还可以做预处理，索引不同类型的内容（比如 PDF），或者进行单词填充。插件还可以影响搜索工作的方式，比如通过填充搜索词条的方式。

InnoDB 的开发人员正在改进全文索引，但是不知道什么时候才能完成。

5.8.4 全文搜索的折中和变通方式

Full-Text Tradeoffs and Workarounds

MySQL 对全文搜索的实现有一些设计上的局限。这些局限可能和特定的目的相冲突，但是也有很多办法可以绕过这些局限。

例如, MySQL 全文索引只会按照频率进行相关性排序。索引不会记录被索引的单词在字符串中的位置, 所以即使单词相邻, 也不会对相关性有贡献。尽管这对大多数应用都没有问题, 尤其是数据量较小的时候, 但是这仍然有可能无法达到需求, 并且 MySQL 全文索引没有提供自由选择排名算法的可能。它甚至没有把用于相邻性排名的数据保存起来。

数据大小是另外一个问题。MySQL 全文索引在数据大小和内存相匹配的时候工作得很好, 但是如果索引没有在内存中, 性能就会非常差, 对于很大的字段尤其如此。在使用短语搜索的时候, 数据和索引都必须和内存相匹配, 才能得到良好的性能。和其他索引类型比起来, 对全文索引进行添加、更新和删除都很慢。

- 修改有 100 个单词的文本需要的不是 1 次索引操作, 而是 100 次。
- 字段的长度通常对其他索引类型没什么影响, 但是对于全文索引, 有 3 个单词的文本和 1 万个单词的文本性能的差异可以相差几个数量级。
- 全文搜索索引更容易引起碎片, 并且要求更频繁地使用 `OPTIMIZE TABLE`。

全文索引也会影响服务器优化查询的方式。索引选择、WHERE 子句和 ORDER BY 工作的方式和我们期望的不一样:

- 如果有全文索引并且查询的 `MATCH AGAINST` 子句可以使用它, MySQL 会使用全文索引来处理查询。它不会在索引之间做比较, 也许有其他的索引比全文索引更好, 但是 MySQL 不会考虑它们。
- 全文搜索索引只能做全文匹配。查询中的任何其他准则, 比如 WHERE 子句, 都必须在 MySQL 读取了行之后才能使用。而其他类型的索引可以一次性地检查 WHERE 子句的几个部分, 这也是一种区别。
- 全文索引不会存储索引的实际文本。因此, 不能像使用覆盖索引那样使用它。
- 全文索引不能被用于排序, 只能在使用自然语言搜索的时候按照相关性进行排序。如果想使用相关性以外的排序方式, MySQL 会使用文件排序。

250 让我们看看这些限制是如何影响查询的。假设有 100 万个文档, 文档的作者有普通索引, 内容有全文索引。现在想对文档内容做全文搜索, 但是只限于作者 123 的作品, 那么可能会写出下面的查询:

```
... WHERE MATCH(content) AGAINST ('High Performance MySQL')
      AND author = 123;
```

但是这个查询的性能会很差。MySQL 将会首先搜索开始的 100 万个文档, 因为它会首先应用全文索引。然后它再使用 WHERE 子句过滤掉其他作者的作品, 但是过滤的时候不能使用对作者的索引。

一个变通的办法就是在全文索引中包含作者的 ID。可以选择一个很不可能出现在文本中的前缀, 然后把作者的 ID 拼接到你后面, 并且把这个单词放到一个用于过滤的列中, 单独进行维护 (比如使用触发器)。

然后可以让全文索引包含这个过滤列, 并且重写查询如下:

```
... WHERE MATCH(content, filters)
      AGAINST ('High Performance MySQL +author_id_123' IN BOOLEAN MODE);
```

如果作者的 ID 非常具有选择性的话, 这种方式效率很高, 因为 MySQL 能很快地在全文索引中查找 “author_id_123”, 并且缩小查找范围。但是如果 ID 没有选择性, 性能可能会变得更差。要小心地使用这种方法。

有时可以使用全文索引来进行有边界的查询。比如, 想搜索固定范围内的座标, 就可以把座标编码到全文集合中。假定某行的座标是 X=123, Y=456。可以交错地把最重要的数字放在前面, 比如 XY142536, 然后把它放到一个列里面并用全文索引包含这个列。现在如果想进行一次限定了 X 和 Y 的矩形式搜索, 比如 X 在 100 到 199

之间, Y 在 400 到 499 之间, 那么这时候就可以在查询中加上 “+XY14”。这比使用 WHERE 子句过滤要快得多。

有种技巧有时能很好地利用全文索引, 特别是对于分页显示, 就是通过全文索引选择主键的列表并且缓存结果。当应用程序准备好了渲染结果时, 它可以用另外一个查询通过主键提取想要的列。第二个查询可以包括更多负责的判断条件或联接, 它们可以更好地使用其他的索引。

尽管只有 MyISAM 支持全文索引, 如果想使用 InnoDB 或其他存储引擎, 也不用担心, 因为你可以自己做全文索引。一个通常的办法就是把表复制到从服务器上, 它可以使用 MyISAM 存储引擎, 然后使用从服务器进行全文搜索。如果不想在不同的服务器上处理查询, 可以把表垂直地分为两部分, 一部分保留文本列, 另一部分保留其余的数据。

也可以把某些列复制到被全文索引了的表中。Sakila.film 表使用了这种技巧, 那些列是用触发器维护的。但是另外一种替代的办法就是使用外部的全文引擎, 例如 Lucene 和 Sphinx。附录 C 有 Sphinx 的更多内容。

使用了全文搜索的 GROUP BY 查询简直就是性能杀手。这是因为全文搜索会找到大量的匹配数据, 引起了随机磁盘读取, 然后使用临时表或文件排序来进行分组。因为这些查询通常都是为了寻找每个分组中靠前的数据, 一种比较好的优化方式就是对数据进行抽样, 而不是完全精确地匹配。例如, 选择最开始的 1 000 行, 放入临时表中, 然后为每个分组返回靠前的数据。

5.8.5 全文调优和优化

Full-Text Tuning and Optimization

对全文索引进行常规的维护是提高性能的最重要方式。全文索引使用的是双重平衡树结构 (Double B-Tree), 并且还有大量的关键词, 这意味着它比普通的索引更容易出现碎片。常常要使用 OPTIMIZE TABLE 来去除索引的碎片。如果服务器是 IO 密集型的, 那么这会比周期性删除和重建索引要快得多。

对于需要键缓冲区 (Key Buffers) 的全文搜索来说, 如果缓存能容纳全文索引, 那么服务器的性能就会更好, 因为全文索引都在内存中的话, 工作效率就会更高。实际上可以使用专用的键缓冲区, 以免别的索引把全文索引冲掉。请参阅第 274 页的 “MyISAM 键缓存” 以了解更多细节。

提供好的停用词表也非常重要。默认的列表对普通的英语文本效果不错, 但是它未必适用于别的语言或专门的技术文档。例如, 在索引有关 MySQL 的文档时, mysql 应该就是停止字, 因为它出现的实在太频繁了, 对搜索没什么帮助。

跳过短的单词常常能提高性能。长度可以使用 ft_min_word_len 进行配置。增加长度会跳过更多的单词, 使索引变得更小和更快, 但是精确性也会降低。在某些特别的情况下, 也许需要很短的单词。例如, 一个在消费电器文档中搜索 “cd player” 的查询可能会产生大量无关的结果, 除非索引允许短的单词。搜索 “cd player” 的用户不想在结果中看到 MP3 和 DVD 的信息, 但是最短长度默认是 4, 所以搜索只会查找 “player”, 这样会返回大量的无关的播放器信息。

停止词表和最短长度可以把某些单词排除在搜索之外, 但是搜索的质量也会受到它们的影响。正确的平衡依赖于具体的应用。如果需要好的性能和好的搜索结果, 那么就要按照程序的要求自己定制这两个参数。一个较好的方式就是用日志进行记录, 然后调查常见的搜索、不常见的搜索、不会返回结果的搜索以及会返回许多结果的搜索。可以用这种方式了解用户需求和搜索的内容, 然后就可以利用这些信息来改进搜索的性能和质量。



提示：要知道如果改变了单词的最小长度，那么可能就需要使用 `OPTIMIZE TABLE` 重建索引，以使其生效。一个相关的参数是 `ft_max_word_len`，它通常可以用来防止索引很长的关键词。

如果正在向数据库导入大量的数据并且希望全文索引某些列，那么在导入之前就应该使用 `DISABLE KEYS` 禁止全文索引，然后在导入后再用 `ENABLE KEYS` 重新启用它。这通常会更快，因为为插入的每一行更新索引需要很多时间，并且这样还可以避免碎片产生。

对于大型的数据集，可能需要手工地对它们进行分区，然后并行地进行搜索。这是很困难的任务，这时使用外部的全文搜索引擎可能会更好，例如 `Lucene` 和 `Sphinx`。经验表明它们的效率要高几个数量级。

5.9 外键约束

Foreign Key Constraints

InnoDB 现在是主要的支持外键的存储引擎。支持外键的选择目前看来比较有限（注 10）。MySQL 承诺过服务器自身会在某天提供和存储引擎无关的外键支持，但是目前看来，主要还是依靠 InnoDB 支持外键。所以我们这儿的讨论集中在 InnoDB 上。

外键是有开销的。在更改数据的时候，它们通常都会要求服务器查看另外一个表。尽管 InnoDB 有索引可以让这个操作更快，但是这通常不会消除这种检查的影响。它甚至能导致很大的索引，但是却毫无选择性。例如，假设一个巨大的表中有一个 `status` 列，并且想约束 `status` 的值必须是有效的，但是它只有 3 个可能的值。在这种情况下，尽管列本身很小，外键要求的额外索引也能极大地增加表的总大小，尤其是如果主键很大的话，更容易发生这种情况。这些索引除了进行外键检查之外没有任何用处。

但是，外键在某些情况下能改进性能。如果必须保证两个表有连续的数据，外键能让服务器进行更有效的检查。外键对于级联删除及更新也很有用。然而它们是逐行操作的，所以比起多表删除或批处理要慢一些。

外键使查询可以“到达”其他的表，这意味着需要锁。例如，向一个子表插入行，外键约束就会要求 InnoDB 检查父表中相关的内容。这肯定会把父表中的行锁住，以保证在事务完成前数据不会被删除掉。这导致了出乎意料的锁等待，甚至会引起死锁。这种问题很难进行调试。

有时可以用触发器来代替外键。外键对于级联更新这样的任务比触发器更好，但是外键只用于一种约束，就像前面的 `status` 列的例子，可以用触发器进行重写，并且显式地列出允许的值（可以使用 `ENUM` 数据类型），以达到较好的效率。

如果不把外键看成一种约束，那么用它来限制应用程序里的值通常是个好主意。

5.10 合并表和分区

Merge Tables and Partitioning

合并表和分区是相关的概念，并且它们之间的区别会让人很迷惑。合并表是 MySQL 的一种特性，它可以把多个 MyISAM 表合并成一个虚表，就像对表使用了 `UNION` 的视图一样。可以使用合并存储引擎创建合并表。合并

注 10: PBXT 也支持外键。

表其实并不是一个真正的表，它更像一个用于放置相似表的容器。

相反的是，分区表是一个正常的表，它包含了一些特殊的指令，告诉 MySQL 物理数据被存放在什么地方。一个秘密就是分区表使用的存储字和合并表使用的极其相似。事实上，每个分区实际都是有索引的独立表，分区表其实包装了很多句柄对象。分区表看上去像一个单独的表，但它实际上是一大堆独立的表，但是无法访问分区表下面的独立表，不过合并表可以。

分区是 MySQL 5.1 的一种新特性。但是合并表已经有很长的历史了。两个特性都会带来同样的好处，它们可以让你做到下面这些事情：

- 分离静态的和变化的数据。
- 使用相关数据的物理相邻性来优化查询。
- 设计表以便查询访问较少的数据。
- 更容易地维护非常多的数据。（合并表在这个领域上比分区表更有优势）

因为 MySQL 在对分区表和合并表的实现上有很多共通之处，它们也有同样的限制。例如，分区表或合并表都有实际的限制，限制它们可以使用多少子表。在大部分情况下，几百张表就能造成性能下降。当我们具体分析某个系统的时候，会说明它的具体限制。

5.10.1 合并表

Merge Tables

如果愿意的话，可以把合并表看成一种较老的、有更多限制的分区表，但是它们也有自己的用处，并且能提供一些分区表不能提供的功能。

合并表实际是容纳真正的表的容器。可以使用特殊的 UNION 语法来 CREATE TABLE。下面是一个合并表的例子：

```
mysql> CREATE TABLE t1(a INT NOT NULL PRIMARY KEY)ENGINE=MyISAM;
mysql> CREATE TABLE t2(a INT NOT NULL PRIMARY KEY)ENGINE=MyISAM;
mysql> INSERT INTO t1(a) VALUES(1),(2);
mysql> INSERT INTO t2(a) VALUES(1),(2);
mysql> CREATE TABLE mrg(a INT NOT NULL PRIMARY KEY)
-> ENGINE=MERGE UNION=(t1, t2) INSERT_METHOD=LAST;
mysql> SELECT a FROM mrg;
```

a
1
1
2
2

注意到合并表包含的表列的数量和类型都是一样的，并且合并表上的索引也会在下属表上存在。这是创建合并表的要求。也要注意在每个表的独有列上有主键，这会导致合并表有重复的行。这是合并表的一个局限：合并表内的每个表行为都很正常，但是它不会对下面的所有表进行强制约束。

INSERT_METHOD=LAST 指令告诉 MySQL 把所有的 INSERT 语句都发送到合并表的最后一个表上。定义 FIRST 或 LAST 是控制插入数据位置的唯一方式（但是也可以直接插入到下属表中）。

分区表可以更多地控制数据存放的位置。

下面的 INSERT 语句对合并表和下属表都可见：

```
mysql> INSERT INTO mrg(a) VALUES(3);
mysql> SELECT a FROM t2;
```

a
1
2
3

合并表还有其他有趣的特性和限制，比如删除合并表或它的某个下属表。删除合并表让所有的“子表”都变得不可访问，但是删除其中的某个子表有不同的影响，它的行为和操作系统有关。例如，在 GNU/Linux 上，子表的文件描述符还保持开启的状态，并且表还继续存在，但是只能从合并表中访问。

```
mysql> DROP TABLE t1, t2;
mysql> SELECT a FROM mrg;
```

a
1
1
2
2
3

还有一些另外的局限性和特殊行为。最好的办法是阅读手册，但是在这儿要说的是 REPLACE 并不能在所有的合并表上工作，并且 AUTO_INCREMENT 不会像你期望的那样工作。

合并表对性能的影响

MySQL 对合并表的实现对性能有一些重要的影响。和其他 MySQL 特性一样，它在某些条件下性能会更好。下面是关于它的一些注意事项：

- 合并表比含有同样数据的非合并表需要更多的文件描述符。尽管合并表看上去是一个表，它实际是逐个打开了下属表。这样的结果就是单个表的缓存可以创建许多文件描述符。因此，即使已经配置了表的缓存，让服务器线程的文件描述符数量不要超过操作系统的限制，合并表仍然有可能导致超过这一限制。
- 创建合并表的 CREATE 语句不会检查下属表是否是兼容的。如果下属表的定义有轻微的不一样，MySQL 会创建合并表，但是却无法使用。同样，如果在创建了一个有效的合并表之后对某个下属表进行了改变，它也会无法工作，并且会显示下面的错误信息：“ERROR 1168 (HY000)：无法打开定义不同的下属表，或者非 MyISAM 表，或者不存在的表”。
- 访问合并表的查询访问了每一个下属表。这也许会使单行键查找比单个表慢。在合并表中限制下属表是一个好主意，尤其是它是联接中的第二个或以后的表。每次操作访问的数据越少，那么访问每个表的开销相对于整个操作而言就越重要。下面是一些如何使用合并表的注意事项：
 - 范围查找受访问所有下属表的开销的影响小于单个查找。
 - 对索引表的表扫描和对单个表一样快。

- 一旦唯一键和主键查询成功，它们就立即停止。在这种情况下，服务器会挨个访问下属表，一旦查找到了值，就不会再查找更多的表。
- 下属表读取的顺序和 `CREAT TABLE` 语句中定义的一致。如果经常需要按照特定的顺序取得数据，可以利用这种特性使合并排序操作更快。

合并表的长处

合并表在处理数据方面既有积极的一面，也有消极的一面。经典的例子就是日志记录。日志是只追加的，所以可以每天用一个表。每天创建新的表并把它加入到合并表中。也可以把以前的表从合并表中移除掉，把它转化为压缩的 `MyISAM` 表，再把它们加回到合并表中。

这并不是合并表的唯一用途。它们通常都被用于数据仓库程序，因为它的另一个长处就是管理大量的数据。在实际中不太可能管理一个 `TB` 级别的表，但是如果是由单个 `50GB` 的表组成的合并表，任务就会简单很多。

当管理极其巨大的数据库时，考虑的绝不仅仅是常规操作。还要考虑崩溃与恢复。使用小表是很好的主意。检查和修复一系列的小表比起一个大表要快得多，尤其是大表和内存不匹配的时候。还可以并行地检查和修复多个小表。

数据仓库中另外一个顾虑就是如何清理掉老的数据。对巨型表使用 `DELETE` 语句最佳状况下效率不高，而在最坏情况下则是一场灾难。但是更改合并表的定义是很简单的，可以使用 `DROP TABLE` 命令删除老的数据。这可以轻易地实现自动化。

合并表并非只对日志和大量数据有效。它可以方便地按需创建繁忙的表。创建和删除合并表的代价是很低的。索引可以像对视图使用 `UNION ALL` 命令那样使用合并表。但它的开销更低，因为服务器不会把结果放到临时表中然后再传递给客户端。这使得它对于报告和仓库化数据非常有用。例如，要创建一个每晚都会运行的任务，它会把昨天的数据和 8 天前、15 天前、以及之前的每一周的数据进行合并。使用合并表就可以创建无须修改的查询，并且自动地访问合适的数据库。甚至还可以创建临时合并表，这是视图无法做到的。

因为合并表没有隐藏下属的 `MyISAM` 表，所以它提供了一些分区表无法提供的特性：

- 一个 `MyISAM` 表可以包含很多合并表。
- 可以通过拷贝 `.frm`、`.MYI`、`.MYD` 文件在服务器之间拷贝下属表。
- 可以轻易地把更多的表添加到合并表中。这只需要创建一个新表并且更改合并定义即可。
- 可以创建只包含想要的数据的临时合并表，例如某个特定时间段的数据。这是分区表无法做到的。
- 如果想对某个表进行备份、恢复、更改、修复，或者其他的操作，可以把它从合并表中移除，完成所有的工作之后再把它加回来。
- 可以使用 `mysampack` 压缩某些或所有的下属表。

分区表正好相反，`MySQL` 隐藏了分区表的分区，并只能通过分区表访问所有的分区。

5.10.2 分区表

Partitioned Tables

`MySQL` 分区表的实现在本质上和合并表非常相像。但是，它紧密地和服务器结合在一起，并且和合并表有一个重大的区别：任何一个给定的数据行只会被存储在一个合适的分区上。表的定义基于分区函数（`Partitioning`

Function)，它约定了行和分区之间的映射关系，我们稍后再讲解这一点。这意味着主键和唯一键是对整个表起作用的。并且 MySQL 优化器可以更智能地优化分区表。

258

下面是分区表的一些重要的益处：

- 可以把某些行放在一个分区中，这可以减少服务器检查数据的数量并且使查询更快。例如，如果按照日期进行分区，然后对某个日期范围内数据的查询就可以只访问一个分区。
- 分区数据比非分区数据更好维护，并且可以通过删除分区来移除老的数据。
- 分区数据可以被分布到不同的物理位置，这样服务器可以更有效地使用多个硬盘驱动器。

MySQL 对分区的实现还在不停地变化，它非常复杂，我们不会在这儿讨论所有的细节。我们讨论主要集中在性能上，关于基础知识，最好去查阅 MySQL 手册。最好能把分区那一章通读一遍，并且仔细了解 CREATE TABLE、SHOW CREATE TABLE、ALTER TABLE、INFORMATION_ SCHEMA、PARTITIONS 和 EXPLAIN。分区使 CREATE TABLE 和 ALTER TABLE 更复杂了。

和合并表一样，分区表实际也是在存储引擎层由有独立索引的单个的表（分区）组成的。这意味着分区表的内存和文件描述符的要求和合并表类似。但是，分区不能从表中独立访问，并且每个分区只能属于一张表。

如同前文所说，MySQL 使用分区函数来决定行到底会被保存到哪个分区中。该函数会返回一个可变的、但是确定的整数。一共有几种分区方式。按范围（Range）分区对每个分区设定了范围值，然后把行基于其范围放入分区中。MySQL 也支持键（Key）分区、哈希（Hash）分区和列表（List）分区。每种类型都有其优势和劣势，尤其是在处理主键的时候。

分区为何可以工作

MySQL 设计分区表的一个关键就是把分区看作一个粗糙的索引。假设一个表有 10 亿行对每一天，每一种物品的销售数据，并且每一行都比较大，假设是 500 字节。只会在表中插入数据，却永远不会更新数据。在大部分情况下，都是按照日期来检索数据。对该表进行查询的主要问题就是它的大小：大约 500GB 没有任何索引的数据。

259

一种加快查询的办法就是在（day，itemno）上添加主键，并且使用 InnoDB。这会把每天的数据物理上绑定在一起，所以范围查询就可以检索较少的数据。另外，还可以使用 MyISAM 并且按照想要的顺序插入数据，这样索引扫描就不会引起大量的随机 I/O 读取。

另外一种选择就是不使用主键，而把数据按日期进行分区。每次访问某个日期段的数据将会扫描整个分区，但是这会比在巨型表上使用索引查找要好得多。分区有一点点像索引，它大致告诉 MySQL 在什么地方去查找数据。但是，它实际上没有使用内存或磁盘空间，精确地说来，这是因为分区没有像索引那样指向具体的行。

但是不要同时加上主键和对表进行分区，这有可能会降低性能。尤其是要对所有分区进行扫描的时候。在考虑分区时，要仔细地做性能评测，因为分区表并不总是能提高性能。

分区示例

下面用两个简短的示例说明分区的好处。首先看看如何设计分区表来存储基于日期的数据。假设已经按照产品对订单和销售数据进行了统计。因为常常需要按照日期范围运行查询，所以会把订单日期放在主键的第一位，并且使用 InnoDB 按照日期聚集（Cluster）数据。现在可以对日期进行分区，进行较高层次的聚集。下面是表

的基本定义，没有使用任何分区策略：

```
CREATE TABLE sales_by_day (
    day DATE NOT NULL,
    product INT NOT NULL,
    sales DECIMAL(10, 2) NOT NULL,
    returns DECIMAL(10, 2) NOT NULL,
    PRIMARY KEY(day, product)
) ENGINE=InnoDB;
```

通常会对基于日期的数据按年或按天进行分区。YEAR()和 TO_DAYS()函数可以用于数据分区。通常情况下，进行范围分区的好的函数可以在想创建分区的值之间形成线性关系，下面按年进行分区：

```
mysql> ALTER TABLE sales_by_day
-> PARTITION BY RANGE(YEAR(day)) (
->     PARTITION p_2006 VALUES LESS THAN (2007),
->     PARTITION p_2007 VALUES LESS THAN (2008),
->     PARTITION p_2008 VALUES LESS THAN (2009),
->     PARTITION p_catchall VALUES LESS THAN MAXVALUE );
```

现在插入的行会根据 day 的值被放入到合适的分区。

```
mysql> INSERT INTO sales_by_day(day, product, sales, returns) VALUES
-> ('2007-01-15', 19, 50.00, 52.00),
-> ('2008-09-23', 11, 41.00, 42.00);
```

稍后会使用这个例子中的数据。但是在继续之前，我们要在这儿指出一个重要的局限：添加更多的年份会改变表，如果表很大的话，代价会很高（这儿已经假设表很大了，如果不大的话，也没有必要使用分区）。一个比较好的办法就是预先定义好更多的年份，即使在很长时间内你都用到这些年份，但是预先把它们包含进来也不会影响性能。

分区表另外一个常用的用途就是分布大表中的行。假设对一个大表运行很多查询。如果想用不同的磁盘为查询服务，那么就会要求 MySQL 将数据行分布到不同的磁盘上。这时不用考虑把相关的数据放在一起，只须简单地把数据平均地分布到磁盘上。下面的例子可以让 MySQL 按照主键的模式分摊数据。它是一种在分区中均匀分布数据的好办法：

```
mysql> ALTER TABLE mydb.very_big_table
-> PARTITION BY KEY(<primary key columns>) (
->     PARTITION p0 DATA DIRECTORY='/data/mydb/big_table_p0/',
->     PARTITION p1 DATA DIRECTORY='/data/mydb/big_table_p1/');
```

可以用磁盘阵列（RAID）控制器实现同样的目标。而且有时候效果会更好，因为磁盘阵列是用硬件执行的，它隐藏了工作的细节，所以不会给数据库的结构和查询带来额外的复杂性。如果目标仅仅是将数据进行物理分布的话，它能提供更好、更一致的性能。

分区表的局限

分区表并不是“银弹”。它现在有如下的局限：

- 当前，所有的分区都要使用同样的存储引擎。例如，不能像合并表那样只压缩部分分区。
- 分区表上的每一个唯一索引必须包含由分区函数引用的列。这样的后果就是很多指导性的示例都避免使用主键。尽管这对于包含没有主键或唯一索引的表的数据仓库而言，是很普遍的现象，但是它对于联机事务处理（OLTP）系统并不常见。相应地，对数据如何分区的选择也会受到限制。

- 尽管 MySQL 能避免分区表的查询访问所有的分区，但是它仍然锁定了所有的分区。
- 分区函数中能使用的函数和表达式有很多限制。
- 一些存储引擎不支持分区。
- 分区不支持外键。
- 不能使用 `LOAD INDEX INTO CACHE`。

还有很多其他的限制（至少在写本书的时候是这样的，MySQL5.1 现在还没有发布）。分区表的灵活程度在某种程度上比合并表要小一些。例如想给分区表添加索引，这个操作并不能在一小段时间内完成，因为 `ALTER` 会将表锁住，并且重新构建整个表。合并表有更多的灵活性，比如可以给一个下属表添加索引。同样地，不能一次只备份或恢复一个分区，但是合并表没有这个限制。

表是否能从分区中得到好处取决于许多因素，应该在应用程序中做实际的评测，以确认它是否是一个好的选择。

利用分区表优化查询

分区引入了一种新的优化查询的方式（当然，也有相应的缺点）。优化器可以使用分区函数修整（Prune）分区，或者把分区从查询中完全移除掉。它通过推断是否可以在特定的分区上找到数据来达成这种优化。因此在最好的情况下，修整可以让查询访问更少的数据。

重要的是要在 `WHERE` 子句中定义分区键，即使它看上去像是多余的。通过分区键，优化器就可以去掉不用的分区，否则的话，执行引擎就会像合并表那样访问表的所有分区，这在大表上会非常慢。

可以使用 `EXPLAIN PARTITIONS` 检查优化器是否去除了分区，还是使用前面例子的数据：

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales_by_day\G
***** 1. row *****
      id: 1
    select_type: SIMPLE
      table: sales_by_day
    partitions: p_2006,p_2007,p_2008
         type: ALL
possible_keys: NULL
          key: NULL
        key_len: NULL
          ref: NULL
         rows: 3
       Extra:
```

正如所见，查询访问了所有的分区。现在看看在 `WHERE` 子句添加一个约束之后有什么不同：

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales_by_day WHERE day > '2007-01-01'\G
***** 1. row *****
      id: 1
    select_type: SIMPLE
      table: sales_by_day
    partitions: p_2007,p_2008
```

优化器很聪明，知道如何去除分区。它甚至把范围转换成了一个离散的列表并修整了列表中的每一项。但是，它不是全知全能的，例如，下面的 `WHERE` 子句在理论上是可以修整的，但是 MySQL 却做不到：

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales_by_day WHERE YEAR(day) = 2007\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: sales_by_day
    partitions: p_2006,p_2007,p_2008
```

在现在的设计中，MySQL 只能通过对分区函数中的列进行修整。它不能修整表达式的结果，即使是表达式和分区函数一样也不行。可以把查询转换成相等的形式，如下：

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales_by_day
-> WHERE day BETWEEN '2007-01-01' AND '2007-12-31'\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: sales_by_day
    partitions: p_2007
```

WHERE 子句直接引用了分区列，优化器现在就可以使用修整了。

优化器也能在处理查询的过程中修整分区。例如，如果一个分区表联接的是第二个表，并且联接条件是分区键，MySQL 就会在相关的分区中搜索匹配的行。这和合并表很不一样，它总是查找所有的下属表。

5.11 分布式 (XA) 事务

Distributed(XA) Transactions

存储引擎事务在存储引擎内部被赋予了 ACID (译注 1) 属性，分布式 (XA) 事务是一种高层次事务，它可以利用两段提交的方式将 ACID 属性扩展到存储引擎外部，甚至数据库外部。MySQL 5.0 及其以上的版本部分支持 XA 事务。

XA 事务需要事务协调员，它会通知所有的参与者准备提交事务 (阶段一)。当协调员从所有参与者那里收到“就绪 (Ready)”信号时，它会通知所有参与者进行真正的提交 (阶段二)。MySQL 可以是 XA 事务的参与者，但不能是协调员。

263

MySQL 内部其实有两种 XA 事务。MySQL 服务器能参与由外部管理的分布式事务，但它内部使用了 XA 事务来协调存储引擎和二进制日志。

5.11.1 内部 XA 事务

Internal XA Transactions

MySQL 内部使用 XA 事务的原因是服务器和存储引擎之间是隔离的。存储引擎之间是完全独立的，彼此不知道对方的存在，所以任何跨引擎的事务本质上都是分布的，并且要求第三方来进行协调。MySQL 就是第三方。假如没有 XA 事务，跨引擎事务提交需要顺序地要求每个引擎进行提交。这样就会引入一种可能，就是在某个引擎提交之后发生了崩溃，但是另外一个引擎还未提交。这就打破了事务的原则。

如果把记录事件的二进制日志看成一个“存储引擎”，那么就能理解为什么即使是单个事务性引擎也需要 XA 事

译注 1: ACID, 是指在数据库管理系统 (DBMS) 中事务所具有的四个特性: 原子性 (Atomicity)、一致性 (Consistency)、隔离性 (Isolation, 又称独立性)、持久性 (Durability)。

务。存储引擎把事件提交给二进制日志时，需要和服务器进行同步，因为是服务器，而不是存储引擎处理二进制日志。

当前的 XA 在性能上有些进退两难。它打破了 InnoDB 从 MySQL 5.0 以来的对**群体提交**（Group Commit）（一种使用单次 I/O 提交多个事务的技术）的支持，所以会导致了很多 `fsync()` 调用。如果二进制日志处于激活状态，那么每个事务都会需要等待日志同步，并且每次事务提交都要求两次日志重写，而不是一次。换句话说，如果想让事务和二进制日志安全地同步，就会要求至少三次 `fsync()` 调用。防止其发生的唯一办法就是禁用二进制日志并把 `innodb_support_xa` 设置为 0。

这样设置无法兼容复制。复制需要二进制日志和 XA 支持，并且为了尽可能地安全，还须要把 `sync_binlog` 设置成 1，这样设置就能对存储引擎和二进制日志进行同步。（否则的话，XA 支持就没有必要了，因为二进制日志不会被提交到磁盘上）。这是强烈推荐使用带有备用电池的写入缓存的磁盘阵列控制器的一个原因，它能加快 `fsync()` 调用并且恢复性能。

下一章将会详细讲解如何配置事务日志及二进制日志。

5.11.2 外部 XA 事务

External XA Transactions

MySQL 可以参与，但不能管理外部分布式事务。它不支持完整的 XA 规范。例如，XA 规范允许连接运行单个事务中的连接，但是 MySQL 现在还不能做到这一点。

外部 XA 事务的开销比内部 XA 事务更高，这是因为延迟会增加，并且参与者失败的可能性更大。在 WAN、甚至是因特网上使用 XA，一个常见的问题就是网络性能不可预测。当有不可预测的部分，比如较慢的网络或一个有可能很久都不点击“保存”按钮的用户，最好的选择就是避免 XA 事务。任何耽搁提交的因素都会有很高的代价，因为它导致的不是单个系统延迟，而是许多系统。

可以用另外的方式设计分布式事务。例如，可以在本地把数据插入队列，然后把它自动地分布成小而快的事务。也可以使用 MySQL 复制把数据从一个地方搬运到另外一个地方。我们也发现某些使用了分布式事务的应用程序其实根本没必要使用事务。

总的说来，XA 事务是一种在服务器之间同步数据的有用的方式。如果因为某些原因，比如不能使用复制或数据更新的性能并不是关键因素，它的效果会不错。

应用层面的优化

Application-Level Optimization

本书若不讲解一章关于连接到 MySQL 的应用程序优化的内容，那就不能算完整，因为人们常常把一些性能方面的问题都归咎到 MySQL 身上。书里面我们更多地是讲到 MySQL 的优化，但是，我们不想让你错过这个更大的图景。一个糟糕的应用设计会使你无论怎么优化 MySQL 也弥补不了它带来的损失。实际上，有时候对于这类问题的答案是把它们从 MySQL 上脱离开来，让应用自己或其他工具来做这些事情，这样或许会有较好的性能表现。

本章不是构建高性能应用的参考书，我们只是希望通过阅读这一章让你避免那些常见的会伤及 MySQL 性能的小错误。下文中我们以 Web 应用为主要讲解对象，因为 MySQL 主要是用在 Web 应用上的。

10.1 应用程序性能概述

Application Performance Overview

对于更快性能的追求开始时很简单：应用响应请求花费了太长的时间，你总要为此做点什么吧。然而，真正的问题是什么呢？通常的瓶颈是缓慢的查询、锁、CPU 饱和、网络延时和文件 I/O。如果应用配置错误，或者不恰当地使用资源，以上任何一个因素都会引出一个大问题。

10.1.1 找出问题的根源

Find the Source of the Problem

第一个任务是找出“肇事者”。如果你的应用具备了显示系统运行概况的功能，这做起来就简单了。如果你已经做到了这一步，但还是没法找出引起性能低下的原因，那你就需要增加更多的概况信息的调用，去找出那些要么缓慢要么被多次调用的资源。

如果你的应用因为 CPU 高占用率而一直等待，并且应用里有高并发性，那我们在第 55 页的“分析应用程序”所提到过的“丢失的时间”可能就成了问题了。鉴于此，有些时候在有限的并发条件下生成应用的概况信息是很有用的。

网络延时会占用大块的时间，哪怕是在局域网里。应用层面的概况信息已经包括了网络延时，因此，你应该在概况系统里看到网络往返延时带来的影响了。举例来说，如果一个页面执行了 1 000 个查询，即使每次只有 1 毫秒的延时，那累加起来也有 0.5 秒的响应时间，这对高性能应用来说已经是个很大的数目了。

如果应用层面概况信息收集得很充分，那就不难找出问题的根源。如果还没有内置概况功能，那就尽可能地加上它。如果你无法添加这个功能，那也可以试试第 76 页的“当你无法加入概况信息代码时”里提供的那些建议。这个总比钻研像“什么引起应用变慢”那样没头绪的理论设想要更快更容易。

10.1.2 寻找常见问题

Look for Common Problems

同样的问题我在应用里一次又一次地遇到，其原因往往是人们使用了设计糟糕的原有系统，或者采用了简化开发的通用框架。虽然这在某些时候能让你在开发一些功能时变得方便又快速，但它们也给应用增加了风险，因为你不知道它们底下是怎么工作的。这里有一张清单你应该逐个检查一下：

- 在各个机器上的 CPU、磁盘、网络 and 内存资源的使用情况如何？使用率对你而言是否合理？如果不合理，就检查那些影响资源使用的应用的程序基础。配置文件有时就是解决问题的最简单方法，举例来说，如果 Apache 耗光了内存，那是因为它创建 1 000 个工作者进程，每个工作进程需要 50MB 内存，这样，你可通过配置文件配置这个应用能申请的 Apache 工作者进程数。你也可以配置系统，使之创建进程时少用些内存。
- 应用是否真正使用了它所取得的数据？一个常见的错误是：读取了 1 000 行数据，却只要显示 10 行就够了，其他 990 行就丢弃了（然而，如果应用缓存了余下的 990 条记录供以后使用，那么这可能是特意做的优化）。
- 应用里是否做了本该由数据库来做的处理？反之亦然。有个对应的例子是：读取了所有行的数据，然后在应用里计算它们的总数；以及在数据库里做复杂的字符串处理。数据库擅长于计数，而应用的编程语言擅长于正则表达式。你该使用正确的工具去干正确的活。
- 应用里执行了太多的查询？那些号称能“把程序员从 SQL 代码里解救出来”的 ORM（Object-Relational Mapping）就因此常被人们责备。数据库服务器是被设计用来匹配多表数据的，因为要移除那些嵌套循环，代之以联接（Join）来做同样的查询。
- 应用里执行的查询太少了？我们只知道执行了太多的查询会成为问题。但是，有时“手工的联接”和与其相似的查询是个好主意，因为它们可以更加有效地利用缓存，更少的锁（尤其是 MyISAM），有时当你在应用的代码里使用一个散列联接时（MySQL 的嵌套循环的联接方法往往是低效的），查询的执行速度会更快。
- 应用是不是在毫无必要的时候还连到 MySQL 上去了？如果你能从缓存里读取数据，就不要去连数据库了。
- 应用连接到同一个 MySQL 实例的次数是不是太多了？这可能是因为应用的各个部分都各自开启了自己的数据库连接。有个建议在通常情况下都很对：从头到尾都重用同一个数据库连接。
- 应用是不是做了太多的“垃圾”查询？一个常见的例子是在做查询前才去选择需要的数据库。一个较好的做法是连接到名称明确的数据库，并使用表的全名做查询。（这样做，也便于通过日志或 SHOW PROCESSLIST 去查询情况，因为你可以直接执行这些查询语句，无需再更改数据库）。“准备”数据库连接又是另一个常见的问题，特别是 Java 写的数据库驱动程序，它在准备连接时会做大量的工作，它们中的大多数你都可以关闭。另一种垃圾查询是 SET NAMES UTF8，这纯粹是多此一举（它无法改变客户端连接库的字符集，它只对服务器有影响）。如果你的应用已确定在多数任务下使用的是某一个字符集，那你就避免这样无谓的字符集设置命令。
- 应用使用连接池了吗？这既是好事情也是坏事情。它限制了连接的数量，这在连接上查询数不多的情况下（Ajax 应用就是个典型的例子）是有利的；然而，它的不好的一面是，应用会受限于使用事务、临时表、连接指定的设置和定义用户变量。

- 应用使用了持久性连接吗？这样做的直接结果是会产生太多的数据库连接连到 MySQL 上。通常情况下，这是个坏主意，除了一种情形：由于慢速的网络导致 MySQL 的连接成本很高，如果每条连接上只执行一两个快速的查询，或者频繁地连接到 MySQL，那样你会很快用完客户端的所有本地端口（更多内容请查看第 328 页的“网络配置”）。如果你正确地配置了 MySQL，根本不需要持久性连接，可以使用“跳过名称解析”来防止 DNS 的查找，并确认该线程的优先级足够高。
- 即使没有使用，应用是不是还打开着连接？如果是，特别是当这些连接连向多台服务器时，它们可能占用了其他进程需要的连接。举例来说，假设你连接到 10 台 MySQL 服务器。由一个 Apache 进程占用 10 个连接数，这不是个问题，但是它们中只有一条连接是在任何指定时间里做着一些操作，而其他 9 条连接绝大多数时间都处于睡眠状态。如果有一台服务器响应变得迟缓，或者网络延时变长，那其他几台服务器就遭殃了，因为它们根本没连接可用。对于这个问题的解决办法是控制应用使用数据库连接的方式。

400

举例来说，你可以在各个 MySQL 实例中依次做批量操作，在向下一个 MySQL 发起查询前，关闭当前的所有连接。如果你要的是时间消耗很大的操作，比如调用一个 Web Service，可以先关闭与 MySQL 的连接，等这个耗时的调用完成后，再打开 MySQL 的连接，完成剩余的需要在数据库上操作的任务。

持久性连接与连接池的不同点比较模糊。持久性连接有与连接池相同的副作用，因为在各种情况下重新使用的连接往往都带有状态。

然而，连接池并不总是导致许多连接到服务器的联接，因为它们是队列化的，并在各进程间共享这些连接。在另一方面，持久化连接是基于每个进程来创建的，无法被其他进程所使用。

与持久性连接相比，连接池在连接策略上有更多的控制。你可以把一个连接池配置成自动扩充的，但是通常的做法还是当连接池满的时候，新的连接请求都被放在队列里等待。这使得这些请求都在应用服务器上等待，总好过 MySQL 因为连接太多而超载。

有太多的方法使查询和连接更加快速，一般性准则是避免把它们放在一起，胜于试着把它们加速。

10.2 Web 服务器的议题

Web Server Issues

Apache 是 Web 应用中使用最广泛的服务器软件。在各种用途下，它都能运行良好，但如果使用得不恰当，它也会占用大量的资源。最常见的一个情况是让它的进程活动了太长的时间，并把它用在各种不同类型的任务下却没有做相应的优化。

Apache 经常在 prefork 配置项里使用 mod_php、mod_perl、mod_python。预分叉（Prefork）是为每个请求分配一个进程。因为 PHP、Perl 和 Python 等脚本语言运行起来很费资源，每个进程占用 50MB 或 100MB 内存的情形也不罕见。当一个请求处理完后，它会把绝大多数内存归还给操作系统，但不会是全部。Apache 会让这个进程保持在运行状态，以处理将要到来的请求。这就意味着如果这个新来的请求只是为了获得一个静态文件，比如一个 CSS 文件或一张图片，你都需要重新启用那个又“肥”又“大”的进程来处理这个简单请求。这也是为什么把 Apache 用作多用途 Web 服务器是件危险的事情。它是多用途的，若你对它进行了有针对性的配置，它才会有更好的性能表现。

另外有个主要的问题是如果你打开了 Keep-Alive 参数项，进程就会长时间地保持忙碌状态。即使你不这么做，

有些进程也会这样。如果内容是像“填鸭”一样传给客户端的，那这个读取数据的过程也会很漫长（注 1）。

人们也经常犯这样的错误：按 Apache 默认开启的模块来运行。你可以按照 Apache 使用手册里的说明，把你不需要的模块都关闭掉，做法也很简单：查看 Apache 的配置文件，把不需要的模块都注释掉，然后重启 Apache。你可以从 `php.ini` 文件中把不需要的 PHP 模块都移除。

如果你创建了一个多用途 Apache 才需要的配置当作 Web 服务器来用，你最后可能会被众多繁重的 Apache 进程所拖垮，这些进程纯粹浪费你的 Web 服务器上的资源。而且，它们会占用大量与 MySQL 的连接，以至于也浪费了 MySQL 的资源。这里有一些方法能给你的服务器“减负”（注 2）：

- 不要把 Apache 用作静态内容的服务，如果一定要用，那也至少要换个另外的 Apache 实例来处理这些事情。常见的替代品有 `lighttpd` 和 `nginx`。
- 使用一个缓存代理服务器，比如 `Squid` 或 `Varnish`，使用户请求无须抵达 Web 服务器后才能被响应。即使在这个层面上你无法缓存所有的页面，你也能缓存大部分页面，并通过 `Edge Side Includes (ESI)`，<http://www.esi.org> 技术把页面上的小块动态部分放到缓存的静态部分里。
- 对动态内容和静态内容都设置过期策略。你可以使用缓存代理软件，像 `Squid`，去验证内容的明确性。`Wikipedia` 就是用这样的技术在缓存里移除内容已发生变化的文档。
- 有时你可能需要改变一下应用，使它能使用更长的超期时间。举例来说，如果你告诉浏览器要永久缓存 `CSS` 和 `JavaScript` 文件，然后又对这个网站静态 `HTML` 文件做了一些修改，这样这些页面的显示效果可能会变得很糟。对此，你需要使用一个唯一的文件名对每次修订后的页面文件都作一个明确的版本标记。举例来说，你可以自定义你的网站发布脚本，把 `CSS` 文件复制到 `css/123_frontpage.css` 目录下，这里的 123 就是 `Subversion` 里的修订号。你也可以用同样的方法来处理图片文件——不要重用原来的文件名，否则，即使你更新了文件内容，页面不会再被更新，不管浏览器要将原来的页面缓存多久。
- 不要让 Apache 与客户端做“填鸭”式通信。这不仅仅是慢，而且很容易招致拒绝性服务攻击。典型地，硬件化的负载均衡器会处理好缓存，Apache 就能很快地结束响应，然后让负载均衡器从缓存里读出数据去“喂”客户端。你也可以使用 `lighttpd`、`Squid`，或者设为事件驱动模式下的 Apache 作为应用的前端。
- 开启 `gzip` 压缩。现在的 CPU 很廉价，它可以用来节省大量的网络流量。如果你想节省 CPU 周期，那可以使用轻量级的 Web 服务器，比如 `lighttpd`，来缓存和提供压缩过的页面。
- 不要将 Apache 上的长距离连接配置为“保活”（`Keep-Alive`）模式，因为它会使 Apache 上臃肿的进程长时间处于运行状态。代替的方案是，用一个服务端的代理来处理“保活”的连接，使服务器免受这类客户端的伤害。如果将 Apache 与代理之间的连接方式设为“保活”，那是不错的主意，因为代理仅使用几个连接从服务器上读取数据。图 10-1 说明了以上两者的差异。

注 1：这种“填鸭”式过程发生在当一个客户端发起一个 HTTP 请求，但无法立即得到请求结果时。直到得到全部数据之前，这个 HTTP 连接及对应的 Apache 进程都将保持忙碌状态。

注 2：有一本关于如何优化 Web 应用的好书，名叫《*High Performance Web Sites*》，作者是 `Steve Sounders (O'Reilly)`。虽然它里面的大多数内容是从客户端的角度来讲怎样使网站运行得更快，但是他倡导的实践案例也适用于你的服务器。

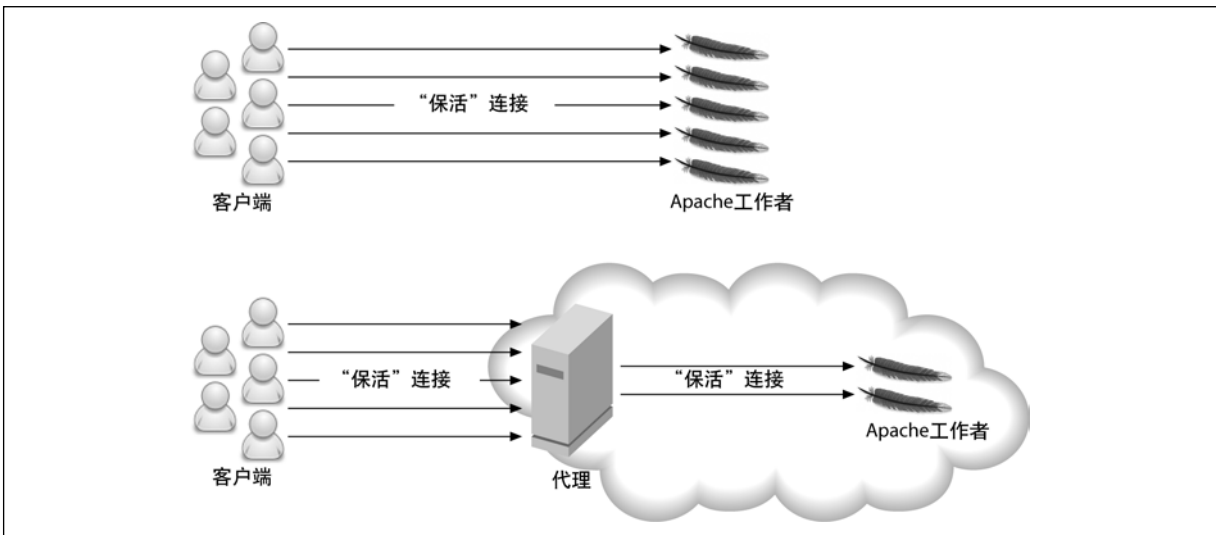


图 10-1：一个代理可以让 Apache 免受长久保持的“保活”连接的负担，从而可以使用更少的 Apache 工作者进程

以上这些策略应该可以帮助 Apache 减少进程的使用数，使你的服务器不会因为太多的进程而崩溃。然而，有些具体的操作仍然会引起 Apache 的进程长时间地运行，吞掉大量的系统资源。有一个例子就是查询外部资源时具有很高的延迟，比如访问一个远程 Web 服务器。这样的问题还是无法用上述那些方法来解决。

10.2.1 找到最佳的并发数

Finding the Optimal Concurrency

每个 Web 服务器都有它的一个**最佳并发数**——它的含义是服务器能同时处理的并发连接数目，它们既能尽可能快地处理客户端请求，又不会使服务器过载。这个“神奇的数目”需要做多次的尝试—失败的反复才能得到，相比于它能带来的好处，这还是值得一做。

对于大流量的网站而言，Web 服务器同时处理几千个连接是件很平常的事情。然而，这些连接中只有很少的一部分需要主动地去处理请求，而其他那些都是读取请求、文件上传、“喂”内容，或者仅仅等待客户端的下一步请求。

463 当并发数增加时，服务器会在某一点上达到它的吞吐量顶峰，在此之后，吞吐量会变得平稳，往往还会开始下降。更重要的是，系统的响应时间（延迟）开始增加。

想要知道究竟，就要设想如果你只有一颗 CPU，而服务器同时接收到 100 个请求，接下来会发生什么？假如一个 CPU 秒只能处理一个请求，而且你使用了一个完美的操作系统，没有任务调度的开销，也没有上下文切换的开销，那么这些请求总共需要 100 个 CPU 秒才能完成。

那么，怎样去做才是处理这些请求的最好办法？你可以把它们一个接一个放进队列里，或者对它们进行并行处理，每个请求在每一个轮回中都获得一样多的处理时间。这两种方式里，吞吐量都是每一秒一个请求。然而，如果使用队列，平均延迟有 50 秒（并发数=1），如果并行处理，那延迟有 100 秒（并发数=100）。在实际环境下，并发处理方法的平均延迟还会更高，因为其中还有个切换开销。

对于高 CPU 占有率的工作负载而言，其最佳并发数就是 CPU（或者是 CPU 里的核）的数目。然而，进程不总是可以运行的，因为它们会执行阻塞式调用，比如 I/O、数据库查询和网络请求等。因此，最佳并发数往往会多于 CPU 数目。

你可以估计最佳并发数，但是这需要精确的分析模型。通常情况下，还是通过实验的方法比较容易，你尝试着不同的并发数，然后观察系统在降低响应时间前，能达到多大的顶峰吞吐量。

10.3 缓存

Caching

缓存对于高负载的应用而言极其重要。一个典型 Web 应用里，直接提供服务要比使用缓存（包括缓存校验、作废）多生成很多内容，所以，缓存能够将应用的性能提高好几个数量级。这个技巧的关键在于找出缓存粒度和作废策略的最佳结合点。同时，你需要决定缓存哪些内容，在哪里缓存。

一个典型的高负载应用有许多层的缓存。缓存不仅仅发生在你的服务器上：它出现在整个流程的每一个步骤上，包括用户的 Web 浏览器里（这就是网页头部的有关作废设置内容的用途）。通常而言，缓存越靠近客户端，就越能节省更多的资源，更加高效。一副图片从浏览器缓存里读出要好于从 Web 服务器的内存里读取，而后者又好于从服务器的磁盘上读取。每一种缓存都有其独有的特性，比如尺寸、延时等，在接下来的章节里我们将对它们逐一进行叙述。

你可以把缓存想象成两大类：**被动缓存**和**主动缓存**。被动缓存除了保存和返回数据不做其他事情。当你从被动缓存那里请求一些内容时，它要么给你需要的结果，要么告诉你“你要的数据不存在”。一个被动缓存的例子就是 memcached。

相反地，主动缓存在找不到请求的数据时，它会做点别的事情。一般就是把你的请求传递给应用的某一部分——它能生成请求所需要的内容，然后主动缓存就会存储这部分内容，并返回给客户端。Squid 缓存代理服务器就是一个主动缓存。

当设计应用时，你总希望你的缓存是主动型（也叫**透明型**）的，因为对于应用，它们可以隐藏“检查—生成—存储”这个逻辑。你可以在被动缓存之上构建你的主动缓存。

缓存并不总是有用

你需要确定缓存是不是真地提高了系统的性能，因为它可能一点用处也没有。举例来说，在实际应用中，从 lighttpd 的内存中读取内容要比从缓存代理那里读取快一些。如果那个代理的缓存是建于磁盘上的，那结论会更明显。

这个原因很简单：缓存也有自己的运行开销，它们主要检查缓存的开销和提供被命中缓存内容的开销，另外还有将缓存内容作废和保存数据的开销。只有当这些开销的总和小于服务器生成和提供数据所要的开销时，缓存才有用。

如果你知道所有这些操作的总开销，你就能计算缓存能起多大的作用。没有缓存时的开销就是服务器为每个请求生成数据所需要的总开销。有缓存时的开销就是检查缓存的开销，加上缓存没命中的可能性乘以生成这些数据的开销，再加上缓存命中的可能性乘以从缓存里取出这些数据的开销。

如果有缓存时的开销小于没缓存的时候的开销，那使用缓存就可以提高系统性能，但是也不能保证肯定是这样。记在脑子里的一个例子就是从 `lighttpd` 内存里读取内容的开销要比代理从磁盘缓存上读取的开销要小，一些缓存总会比另外一些便宜。

10.3.1 在应用之下的缓存

Caching Below the Application

MySQL 服务器有它自己的内部缓存，你也可以构建你自己的缓存和汇总表。你可以自定义缓存表，以便于更好地将它用于过滤、排序、与其他表做联接、计数，以及其他用途。缓存表比其他应用层的缓存更加持久，因为它们服务器重启后还会继续存在。

我们在第 3 章、第 4 章里讲到过这些缓存策略，因此在本章里，我们的篇幅主要集中在应用层面和应用之上的缓存。

465

10.3.2 应用层面的缓存

Application-Level Caching

典型的应用层面的缓存一般都是将数据放在本机内存里，或者放在网络上的另外一台机器的内存里。

应用层面的缓存一般要比更低层面的缓存有更高的效率，因为应用可以把部分计算结果存放在缓存里。因而，缓存对两类工作很有帮助：读取数据和在这些读取数据之上做计算。一个很好的例子是 HTML 文本的各个分块。应用能够产生 HTML 段落，比如头条新闻，然后将它们缓存起来。随后打开的页面里就能将这些被缓存起来的头条新闻直接放到页面上。通常来讲，缓存之前处理的数据越多，使用缓存之后能节省的工作量也越多。

这里有个不足之处就是缓存的命中率越多，要提高它而花费的钱就越多。假如你需要 50 个不同版本的头条新闻，能根据用户所在的不同地域来显示不同的头条。你需要有足够的内存来保存这全部 50 个版本的头条新闻，任何一个给定版本的头条被请求得越少，那它的作废操作也会越复杂。

应用缓存有许多种类型，以下是其中的一部分：

本地缓存

这种缓存一般都比较小，只存在于请求处理时的进程内存空间里。它们可用于避免对同一资源的多次请求。因此，它也没什么精彩之处：它往往只是应用程序代码里的一个变量或一个散列表。举例来说，如果需要显示用户名，而你只知道用户 ID，于是就设计一个函数叫 `get_name_from_id`，把缓存功能放在这个函数里，具体代码如下：

```
<?php
function get_name_from_id($user_id) {
    static $name; // static makes the variable persist
    if ( !$name ) {
        // Fetch name from database
    }
    return $name;
}
?>
```

如果你使用的是 Perl，那么 Memoize 模块就是缓存函数调用结果的标准办法：

```
use Memoize qw(memoize);
memoize 'get_name_from_id';
sub get_name_from_id {
    my ( $user_id ) = @_;
    my $name = # get name from database
    return $name;
}
```

这类技术都比较简单，但是它们能帮你节省大量工作。

本地共享内存式缓存

这种缓存大小中等（几个 GB）、访问快速，同时，难于在各机器间同步。它们适用于小型的、半静态的数据存储。举例来说，像每个州的城市列表、共享数据存储里的分块函数（使用映射表），或者应用了存活时间（Time-to-live，TTL）策略的数据。共享内存的最大好处是访问时非常快速——一般要比任何一种远程缓存要快很多。

分布式内存缓存

分布式内存缓存的最著名的例子是 memcached。分布式缓存比本地共享缓存要大，增长也容易。每一份缓存的数据只被创建一次，因为不会浪费你的内存，当同一份数据在各处缓存时也不会引起数据一致性问题。分布式内存擅长于对共享对象的排序，比如用户信息文件、评论和 HTML 片段。

这种缓存比本地共享缓存有更高的延迟，因此最有效的使用它们的方法是“多取”操作（比如在一次往返时，读取多个对象数据）。它们也要事先规划好怎么加入更多的节点，以及当一个节点崩溃时该怎么做。在这两种情形下，应用都要决定如何在各节点间分布或重新分布缓存对象。

当你在缓存集群里增加或减少一台服务器时，一致性的缓存对于性能问题就显得尤为重要。这里有一个用于 memcached 的一致性缓存库：<http://www.audioscrobbler.net/development/ketama/>。

磁盘缓存

磁盘是慢速的，所以，持久性对象最适合做磁盘缓存。对象往往不适合放在内存里，静态内容也是（比如预生成的自定义图片）。

非常有效地使用磁盘缓存和 Web 服务器的技巧是用 404 错误处理过程来捕捉没命中的缓存。加入你的 Web 应用要在页面的头部显示一个用户自定义的图片，暂且将这个图片命名为/images/welcomeback/john.jpg。如果这个图片不存在，它就会产生一个 404 错误，同时触发错误处理过程。接着，错误处理过程就生成这个图片，并存放在磁盘上，然后再启动一个重定向，或者仅仅把这个图片“回填”到浏览器里，那么，以后的访问都可以直接从文件里返回这个图片了。

你可以将这项技巧用于许多类型的内容，举例来说，你用不着再缓存那块用来显示最新头条新闻的 HTML 代码了，而把它们放入一个 JavaScript 文件里，然后在页面的头部插入指向这个 js 文件的引用。

缓存失效的操作也很简单：删除这个文件就可以了。你可以通过运行一个周期性的任务，将 N 分钟前创建的文件都删除掉，来实现 TTL 失效策略。

如果想对缓存的尺寸做限制，那你可以实现一个最近最少使用（Least Recently Used，LRU）的失效策略，根据缓存内容的创建时间来删除内容。

这个失效策略需要你在文件系统的挂载（Mount）选项上开启“访问时间”这个开关项。（实际操作时忽略 `noatime` 挂载选项来达到这个目的）。如果这么做了，你就应该使用内存文件系统来避免大量的磁盘操作。更多内容请查看第 331 页的“选择文件系统”。

10.3.3 缓存控制策略

Cache Control Policies

缓存引出的问题跟你数据库设计时违背了基本范式一样：它们包含了重复数据，这意味更新数据时要更新多个地方，还要避免读到过期的“坏”数据。以下是几个常用的缓存控制策略：

存活时间

每个缓存的对象都带有一个作废日期，用一个删除进程定时检查该数据的作废时间是否到达，如果是就立即删除它，你也可以暂时不理睬它，直到下一次访问它时，如果已经超过作废时间，那才用一个更新的版本来替换它。这种作废策略最适用于很少变动或几乎不用刷新的数据。

显式作废

如果缓存里的数据过于“陈旧”而无法被接受，那么更新缓存数据的进程就立即将该旧版本的数据作废。这个策略里有两个变体类型：写一作废和写一更新。写一作废策略非常简单：直接将该数据标志为作废（也可以有从缓存里把它删除掉的选择）。写一更新策略就有更多的工作要做，因为你还要用最新的数据来替换旧缓存数据。但是，这个策略非常有用。特别是当生成缓存数据的代价很昂贵时（这个功能在写的进程里可能已经具备）。更新了缓存之后，将来的请求就不用不着再等应用来生成这份数据了。如果你是在后台执行作废过程的，比如是基于 TTL 的作废过程，你可以在一个独立于任何用户请求的进程里生成最新版本的数据去替换缓存里已作废的数据。

读时作废

相对于在改变源数据时使缓存里对应的旧数据作废，有一个替代性的方法是保存一些信息来帮你判断从缓存里读出的数据是否已经作废。它有个比显式作废更显著的优点：随着时间的增长，它开销是固定的。假设你要将一个对象作废，而缓存里有 100 万个对象依赖于它。如果在写时将它作废，你就不得不将缓存里的相关 100 万个对象都作废。而 100 万次读的延迟是相当小的，这样就可以摊薄作废操作的时间成本，避免了加载时的长时间延迟。

468

采用写时作废策略的最简单的方法是实行对象版本化管理。在这个方法里，当把对象保存到缓存里时，你同时要保存该数据所依赖的版本号或时间戳。举例来说，假设你将一个用户在博客发表的文章的统计信息保存到缓存里，这些信息包括了发表文章的数量。当将它作为 `blog_stats` 对象缓存时，你同时也要把该用户当前的版本号也保存起来，因为这个统计信息依赖于具体某个用户。

无论什么时候你更新了依赖于用户的数据，也要随之改变用户的版本号。假设用户版本初始为 0，你生成并缓存这些统计信息。当用户发表了一篇文章后，你就将用户版本号改为 1（最好将这个版本号与文章存放在一起，尽管这个例子我们不必这么做）。那么，当你需要显示统计信息时，就先比较缓存的 `blog_stats` 对象的版本和缓存的用户版本，因为这时用户的版本比这个对象的版本要高，这样你就知道这份统计信息里的数据已经陈旧，须要更新了。

这种用于内容作废的方法相当粗糙，因为它预先假设了缓存里的依赖于用户的数据也跟其他数据进行互动。这个条件并不总是成立。举例来说，如果用户编辑了一篇文章，你也会去增加用户的版本号，这使得缓存里的统

计数据都要作废了，哪怕真正的统计信息（文章的数目）实际上根本没发生变化。折中的方案是朴素的，一个简单的缓存作废策略不仅要易于实现，还要有更高的效率。

对象版本化管理是**标签式缓存**的一个简化形式，后者可以处理更复杂的依赖关系。一个标签化缓存了解不同类别的依赖关系，并能单独追踪每一个对象的版本号。在上一章的图书俱乐部的例子里，你可以这样给评论做缓存：用用户版本号和书本版本号一起给评论做标签，具体像 `user_ver=1234` 和 `book_ver=5678` 这样。如果其中一个版本发生了变化，你就要刷新缓存。

10.3.4 缓存对象的层次化

Cache Object Hierarchies

把对象按层次结构存放在缓存中，有助于读取、作废和内存使用的操作。你不仅要将对对象本身缓存起来，还要缓存它们的 ID 和对象分组的 ID，这样就能方便成组地读取它们。

电子商务网站上的搜索结果就是这种技术很好的例子。一次搜索可能返回一个匹配的产品清单，清单里包含了产品的名称、描述、缩略图和价格。如果把整个列表存放到缓存里，那读取时的效率是低下的，因为其他的搜索可能也会包含了同样的某几个产品，这样做的结果就是数据重复、浪费内存。这个策略也难以在产品价格发生变化时到缓存里找到对应的产品并使其作废，因为必须逐个清单地去查看是否存在这个价格变化了的产品。

一个可以代替缓存整个清单的方法是把搜索结果里尽量少的信息缓存起来，比如搜索的结果数目和结果清单里的产品 ID，这样你就可以单独缓存每一个产品资料了。这个方法解决了两个问题：一是消除了重复数据；二是更容易在单独产品的粒度上将缓存数据作废。

这个方法的缺点是你不得不从缓存里读取多个对象数据，而不是立即读取到整个搜索结果。然而，另一方面这也让你能更快地按照产品 ID 对搜索结果进行排序。现在，一次缓存命中就返回一个 ID 列表，如果缓存允许一次调用返回多个对象（Memcached 有一个 `mget()` 调用支持这个功能），你就可以用这些 ID 再到缓存里去读取对应的产品资料。

如果你使用不当，这个方法也会产生古怪的结果。假设你使用 TTL 策略来作废搜索结果，当产品资料发生变化时，明确地将缓存里对应的单个产品资料作废。现在试着想象一个产品的描述发生了变化，它不再包含跟缓存里搜索结果匹配的关键字，而搜索结果还没到作废时间。于是，你的用户就会看到“陈旧”的搜索结果，因为缓存里的这个搜索结果仍然引用了那个描述已经发生变化的产品。

对于多数应用来说，这一般不成为问题。如果你的应用无法容忍这个问题，那么就可以使用以版本为基础的缓存策略，在搜索之后，把产品版本号和搜索结果放在一起。在缓存里找到一个搜索结果后，把结果里的每个产品的版本号跟当前产品的版本号（也是在缓存里的）进行比较，如果发现有版本不符的，就通过重新搜索来获取新的搜索结果。

10.3.5 内容的预生成

Pregenerating Content

除了应用层面上缓存数据之外，你还可以使用后台进程向服务器预先请求一些页面，然后将它们转换为静态页面保存在服务器上。如果页面是动态变化的，那你可以预生成页面中的一部分，然后使用一种技术，比如服务端整合，来生成最终页面。这样有助于减少预生成内容的大小和开销，因为本来你要为了各个最终页面上的

细微差别而不得不重复存储大量的内容。

缓存预生成的内容会占用大量空间，也不可能总是去预生成所有东西。无论哪种形式的缓存，预生成内容里的最重要部分就是请求最多的那些内容。因此，像我们在本章的前面提到过的那样，你可以通过 404 错误处理程序来对内容作“按需生成”。这些预生成的内容一般都放在内存文件系统中，避免放在磁盘上。

470

10.4 扩展 MySQL

Extending MySQL

如果 MySQL 完不成你所需要的任务，有一种可能性就是扩展它的能力。在这里，我们不是打算告诉你怎么做扩展，而是要提一下这个可能性里的一些具体途径。如果你有兴趣去深究其中的任何一条途径，那么网上有很多资源可供使用，也有很多关于这个主题的书可以参考。

当我们说“MySQL 完不成你所需要的任务”时，其中包含了两个含义：一是 MySQL 根本做不到，二是 MySQL 能做到，但是使用的办法不够好。无论哪个含义都是我们要扩展 MySQL 的理由。一个好消息是 MySQL 现在变得越来越模块化、多用途了。举例来说，MySQL 5.1 有大量可用的功能插件，它甚至允许存储引擎也是插件形式的，这样你就用不着把它们编译到 MySQL 服务器里了。

使用存储引擎将 MySQL 扩展为特定用途的数据库服务器是个伟大的想法。Brian Aker 已经编写了一个存储引擎的框架和一系列的文章、幻灯片来指导用户如何开发自己的存储引擎。这已经构成了一些主要的第三方存储引擎的基础。如果跟踪 MySQL 的内部邮件列表，你会发现现在有许多公司正在编写他们自己的内置存储引擎。举例来说，Friendster 使用一个特别的存储引擎来做社交图操作，另外，我们还知道有一家公司正在做一个用来做模糊搜索的引擎。编写一个简单的自定义引擎一点也不难。

你也可以把存储引擎直接用作软件某一部分的接口。Sphinx 就是个很好的例子，它直接与 Sphinx 全文检索软件通信（请查看附录 C）。

MySQL 5.1 也允许全文检索解析器插件，如果你能编写 UDF（请查看第 5 章），它擅长处理 CPU 密集的任务，这些任务必须在服务器线程环境下运行，对于 SQL 而言又太慢太笨重。因此，你可以用它们完成系统管理、服务集成、读取操作系统信息、调用 Web 服务、同步数据，以及其他更多相类似的任务。

MySQL 代理另外有一个很棒的选项，可以让你向 MySQL 协议增加你自己的功能。Paul McCullagh 的可扩展大二进制流框架项目（<http://www.blobstreaming.org>）为你打通了在 MySQL 里存储大型对象的道路。

因为 MySQL 是免费的、开源的软件，所以当你感觉它功能不够用时，你还可以去查看服务器代码。我们知道一些公司已经扩展了 MySQL 内部解析器的语法。近年来，还有第三方提交的许多有趣的 MySQL 扩展，涵盖了性能概要、扩展及其他新奇的应用。当人们想扩展 MySQL，MySQL 的开发者们总是反应积极，并乐于提供帮助。你可以通过邮件列表 internals@lists.mysql.com（注册用户请访问 <http://lists.mysql.com>）、MySQL 论坛和 IRC 频道 #mysql-dev 跟他们取得联系。

471

10.5 可替代的 MySQL

Alternatives to MySQL

MySQL 不是一个能适用于所有需要的万能解决方案。有些工作全部放到 MySQL 之外会更好，即使 MySQL 在

理论上也能做到。

一个很明显的例子是在传统的文件系统里对数据进行排序而不是在表里。图像文件是又一个经典的案例：你可以把它们都放在 BLOB 字段里，但是这在多数时候都不是个好主意（注 3）。通常的做法是把图像文件或其他大型二进制文件存在文件系统里，然后把文件名放在 MySQL 里。这样，应用就可以在 MySQL 之外读取文件了。在 Web 应用里，你可以把文件名放在 元素的 src 属性里。

全文检索也是应该放在 MySQL 之外处理的任务之一——MySQL 不像 Lucene 或 Sphinx（请查看附录 3）那样擅长于这类检索。

NDB API 可以被用于某一类型的任务。比如，虽然 MySQL 的 NDB Cluster 存储引擎不适合在高性能要求的 Web 应用中作排序操作，但是可以通过直接使用 NDB API 来存储网站的 session 数据或用户注册信息。关于 NDB API，你可以访问 <http://dev.mysql.com/doc/ndbapi/en/index.html> 来获取更多信息。Apache 上也有相应的 NDB 模块，你可以从 <http://code.google.com/p/mod-ndb/> 下载。

最后，对于有些操作，比如图形化的关系、树的遍历，关系数据库并不擅长做这些。MySQL 也不擅长分布式数据处理，因为它缺少并行查询的执行能力。你可能需要使用别的工具（与 MySQL 一起使用）来达到这一目的。

注 3：使用 MySQL 复制功能能快速地将图像文件发布到其他机器上。据我们所知，一些应用使用了这项技术。

作者简介

Baron Schwartz 是一名软件工程师，他住在弗吉尼亚州的 Charlottesville，在网上用的名字是 Xaprb，这是他名字的第一部分按 QWERTY 键盘的顺序打在 Dvorak 键盘上时显示出来的名字。当他不忙于解决有趣的编程挑战时，Baron 就会和他的妻子 Lynn、狗 Carbon 一起享受闲暇时光。他的关于软件工程的博客地址是 <http://www.xaprb.com/blog>。

Peter Zaitsev，MySQL AB 公司高性能组的前任经理，现正运作着 mysqlperformanceblog.com 网站。他擅长于帮助管理员为每天有着数以百万计访问量的网站修补漏洞，使用数百台服务器来处理 TB 级的数据。他常常为了找到一个解决方案而修改和升级软硬件（比如查询优化）。Peter 还经常在讨论会上发表演讲。

Vadim Tkachenko，Percona 公司的合伙人，该公司是一家专业的 MySQL 性能咨询公司。他过去是 MySQL AB 公司的性能工程师。作为一名在多线程编程和同步领域里的专家，他的主要工作是基准测试、特征分析和找出系统瓶颈。他还在性能监控和调优方面做着一些工作，使 MySQL 在多个 CPU 上更具有伸缩性。

Jeremy D. Zawodny 和他的两只猫在 1999 年底从俄亥俄州的西北部搬到了硅谷，这样他就能为 Yahoo! 工作了——那时他刚好亲眼见证了 .com 泡沫的破灭。他在 Yahoo! 工作了八年半，将 MySQL 和其他开源技术组合起来使用，找到有趣的、令人兴奋的用途，而它们往往也是很大的用途。

近段时间，他重新发掘出了对飞行的热爱。其实，早在 2003 年年初，他就已经取得了私人滑翔机飞行员的执照，2005 年获得商业飞行员的定级。从那时起，他花了大量的空闲时间驾驶滑翔机，飞翔在 Hollister、加利福尼亚和 Tahoe 湖地区上空。他偶尔还会驾驶单引擎轻型飞机，和别人共同拥有一架 Citabria 7KCAB 和一架 Cessna 182。临时的咨询工作可以帮助他支付飞行账单。

Jeremy 和他可人的妻子及四只猫生活在加州的旧金山湾区。他的博客地址是 jeremy.zawodny.com/blog。

Arjen Lentz 出生在阿姆斯特丹，但从千禧年以来他和他美丽的女儿 Phoebe、黑猫 Figaro 一直生活在澳大利亚的 Queensland。Arjen 最初是 C 程序员，在 MySQL AB 公司(2001-2007)里是第 25 号职员。在 2007 年短暂的休息之后，Arjen 创建了 Open Query (<http://openquery.com.au>)，该公司致力于在亚太及临近地区开发和提供数据管理培训和咨询服务。Arjen 也经常在讨论会和用户群中发表讲演。在充裕的闲暇时间里，Arjen 热衷于烹饪、园艺、阅读、露营，以及研究 RepRap。他的博客地址是 <http://arjen-lentz.livejournal.com>。

Derek J. Balling 自 1996 年以来就一直一直是 Linux 系统管理员。他协助 Yahoo! 那样的公司和 Vassar 学院那样的机构建立和维护服务器基础设施，也曾为 Perl 杂志和其他一些在线杂志撰写文章，并一直为 LISA (Large Installation System Administration) 会议的编程委员会服务。目前，他作为数据中心经理受雇于 Answers.com。

当不做与计算机有关的事情时，Derek 喜欢和他的妻子 Debbie 及他们的动物群（四只猫和一只狗）在一起。在博客 <http://blog.megacity.org> 上，他也会对当前热点发出评论或写些近来惹恼他的事情。

封面说明

High Performance MySQL 的封面动物是一只雀鹰(*Accipiter nisus*)，它是猎鹰家族的一员，生活在欧亚大陆和北非的林地周围。雀鹰有一条长长的尾巴和一双短翅膀；雄鸟是蓝灰色的，有一个浅棕色的胸部；雌鸟大多是棕灰色的，胸部几乎全白。雄鸟（28 厘米）通常要比雌鸟（38 厘米）小一些。

雀鹰生活在针叶林里，以小型哺乳动物、昆虫和鸟类为食。它们的巢一般筑在树上，有时甚至在悬崖峭壁上。每年夏初，在最高一棵树的主干上的巢里，雌鸟产下 4 至 6 个白色的，略带红色和棕色斑点的蛋。而雄鸟会给雌鸟和孩子们喂食。

像所有的老鹰一样，雀鹰具有在飞行时突然高速俯冲的能力。无论是高飞还是滑翔，雀鹰都会有带着明显特征的拍翅－拍翅－滑行的动作；它的大尾巴使它能够扭身，轻松地出入树林。

封面图片是一幅 19 世纪雕版画，来自于 Dover Pictorial Archive。