O'REILLY®

Broadview
www.broadview.com.cn

第4版

高性能 MySQL

经过大规模运维验证的策略

High Performance MySQL, Fourth Edition

[美] Silvia Botros
[美] Jeremy Tinley 著
宁海元 周振兴 张新铭 译
[美] Jeremy Cole 作序

中国工信出版集团
电子工业出版社

O'REILLY®
高性能 MySQL 第4版
[美] Silvia Botros
[美] Jeremy Tinley 著

版权信息

COPYRIGHT

书名：高性能MySQL：第4版

作者：（美）西尔维亚·博特罗斯（Silvia Botros），（美）杰里米·廷利（Jeremy Tinley）

译者：宁海元，周振兴，张新铭

出版社：电子工业出版社

出版时间：2022年10月

ISBN：9787121442575

本书由“满屋书香”整理，如果你不知道读什么书或者想获得更多免费电子书请加小编微信：sisijuan2012或QQ：151680600 小编也可以结交一些喜欢读书的朋友。或者关注小编个人微信公众号名称：满屋书香。 公众号中可以获得书香君所有分享的电子书！

内容简介

《高性能MySQL》一直是MySQL领域的经典之作，影响了一代又一代的DBA和技术人员，从第3版出版到第4版出版过去了近十年，MySQL也从5.5版本更新到了8.0版本。第4版中增加了大量对MySQL 5.7和8.0版本新特性的介绍，删除了一些在新版本中已经废弃或者不再常用的功能，还增加了对云数据库的介绍，减少了在官方文档中已有的基础使用和配置相关的内容。这些年，MySQL经过在大量大规模互联网场景中的应用验证，使得本书在继续关注高性能之外，还用了较多的篇幅来介绍如何实现MySQL的大规模可扩展应用和合规性问题，这是相比第3版最大的不同，也是本书封面上所写的“经过大规模运维验证的策略”的体现。

本书适合数据库管理员（DBA）阅读，也适合系统运维和开发人员参考学习。不管你是数据库新手还是专家，相信都能从本书中有所收获。

O'Reilly Media, Inc.介绍

O'Reilly以“分享创新知识、改变世界”为己任。40多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O'Reilly业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、互动学习、认证体验、图书、视频等等，使客户更容易获取业务成功所需的专业知识。几十年来O'Reilly图书-直被视为学习开创未来之技术的权威资料。我们所做的一切是为了帮助各领域的专业人士学习最佳实践，发现并塑造科技行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O'Reilly Radar博客有口皆碑。”

—Wired

“O'Reilly凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

—Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

—Irish Times

“Tim是位特.立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按.照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

—Linux Journal

本书赢得的赞誉

我喜欢这个新版本，它将重点转向了现代、务实的通过团队成员传递商业价值的思维模式。它改变了前几版只关注复杂的内部结构和理论从而导致收益缩减的短视，转向了更全面的视角。本书仍然覆盖关于“数据库如何工作”的知识，但现在是以一个全新的、人性化的观点呈现的，这是非常有必要的。

—Baron Schwartz，《高性能MySQL》第2版和第3版的主要作者

自17年前第1版面世以来，《高性能MySQL》一直是MySQL世界的重要组成部分。MySQL一直在向前发展，Silvia和Jeremy做了一项伟大的工作，使这本书的内容与现代MySQL保持同步。

—Jeremy Cole

本书第4版更新了现代实践经验，为MySQL管理员和开发者提供了丰富、可靠的建议。

—Shlomi Noach，PlanetScale数据库工程师

《高性能MySQL》（第4版）有了新的关注点，它不再只是深挖MySQL的每一点性能，我们现在拥有一个由工具和供应商组成的大型生态系统。Silvia和Jeremy完美地详述了MySQL如何适应新的局面。不管你用什么方式使用MySQL，这本书都是必备的。

—Sugu Sougoumarane，Vitess联合创始人，PlanetScale CTO

Silvia和Jeremy做了一项了不起的工作，他们既保留了本书的核心内容，又对其进行了更新，以覆盖快速变化的MySQL世界。

—Peter Zaitsev，Percona创始人兼CEO，《高性能MySQL》（第3版）作者之一

《高性能MySQL》从第2版引入国内以来一直是业界经典。宁海元翻译的第3版让大部分MySQL从业人员受益匪浅，十年后的今天我们又迎来了《高性能MySQL》的第4版。十年来，中国的MySQL发展从部分互联网场景发展到全领域场景，本书也新增了大量云化、规模化、可扩展等特性的内容，通俗易懂又专业深入，是时候更新一下你的书架了。

陈栋，沃趣科技创始人&CEO

岁月洗练，历久弥新。高性能是一个永恒的话题和不懈的追求。在MySQL领域，本书无疑是最佳的表率之一。从互联网到企业级应用，从独立部署到虚拟化和云计算，十年之间，天翻地覆，然而MySQL的魅力有增无减。数据库性能涉及很多复杂的技术领域，而本书娓娓道来深入浅出的风格让人着迷，使得系统性的理解水到渠成。当我读到Performance Schema常因性能影响而被建议关闭的主题时，仿佛看到与当年Oracle同样的演进历程，MySQL逐渐丰富的可观测性，已经使其性能诊断能力接近了商业数据库的水准，臻至大成。

盖国强，云和恩墨创始人，鲲鹏MVP

《高性能MySQL》已经是MySQL DBA的案前必备图书了，这本书不仅教会了大家很多MySQL的最佳实践，同时还从内核实现层面深入地解释原理，做到了知其然更知其所以然。所以这不仅是一本很好的MySQL工具书，同时也是有志于从事数据库内核研发的朋友很好的参考资料。

黄东旭，PingCAP联合创始人兼CTO

无论是在中国还是全世界，MySQL都是当下最流行、使用最广泛的开源关系数据库。《高性能MySQL》是MySQL社区非常有影响力的一本书，第4版使得这本书与近十年MySQL的发展和实践保持了高度同步。

阳振坤，OceanBase数据库创始人

从2009年开始，在过去的十几年里，《高性能MySQL》一直陪伴着我。它也是我向MySQL爱好者推荐最多的一本书，甚至超过了我自己写的《MySQL运维内参》。它可以帮助MySQL相关人员把自己的数据库管理和运维能力快速提升到相对较高的水平并解决工作中碰到的实际问题，是一本不可多得的经典之作。

近几年来，MySQL的发展很快，在内核层的变化也很大，同时伴随着云原生的发展，MySQL也是云上数据库的主流产品。本书第4版恰逢其时地增加了相应的内容，在技术深度和知识广度上做到了很好的平衡。非常感谢原作者和译者所做的努力，期待更多的人通过本书获得更多的MySQL知识。通过从本书中学到的知识，在工作中解决实际问题，这是对为本书付出劳动的人的最好回馈。

周彦伟，极数云舟创始人，Oracle MySQL ACE Director，中国计算机行业协会数据库专委会会长

推荐序

随着互联网行业以及云计算产业的高速发展，MySQL成为世界范围内以及中国数据库领域最流行的开源数据库。在几乎所有大型互联网业务场景中，MySQL都是业务架构的核心组件之一。广泛的应用也推动了MySQL在过去十年的高速发展，MySQL社区相继推出了5.6、5.7、8.0版本，从性能、可扩展性、安全性、稳定性、可维护性、易用性等维度都有了非常大的发展。《高性能MySQL》（第3版）是2012年出版的，最新版本的《高性能MySQL》（第4版）在上一版内容的基础上延续了之前的经典内容，包括架构设计、优化、高可用等内容，同时新增了云数据库、扩展性等过去十年发展的相关内容，另外还增加了MySQL过去十年里的新版本（包括5.7、8.0版本）的新特性。

MySQL是当下最流行的开源数据库之一，本书从实践的角度涵盖了数据库系统的架构设计、锁、性能管理、高可用等内容，除了可作为MySQL的参考书之外，也可以作为数据库系统原理和设计的一个实现参考。云数据库已流行多年，本书最新版对云数据库的内容也做了相应的调整。例如，对数据库的安装、配置、监控搭建等基础操作内容进行了大幅缩减，增加了对云端具体操作的介绍。因此，本书也非常适合作为面向云数据库系统开发者的一本MySQL参考图书。如本书的名字所述，本书在内核设计、性能优化方面，依旧是着墨最多的部分，深入介绍了锁管理、并发控制、Performanc Schema使用、索引优化等内核机制，可以帮助企业的DBA、想深入了解MySQL优化的开发者，以及云数据库开发者更高效地使用和拓展MySQL。

本书的译者是云数据库领域和MySQL数据库的资深专家，有着很强的技术能力、行业实践以及业务洞察，同时具备非常出色的业务架构设计和商业化经验。在深入理解原著的基础上，译者们结合自己的洞察和经验提供了出色的专业化中文版本，是MySQL领域不可多得的一本必读书目。

李飞飞（飞刀），ACM/IEEE Fellow，阿里云数据库负责人

译者序

十年时光，转瞬即逝；重译经典，心境迥异。

2012年，电子工业出版社博文视点公司的张春雨找我来组织翻译《高性能MySQL》的第3版；2022年，还是张春雨找我来组织翻译《高性能MySQL》的第4版。2012年我还在淘宝DBA团队，淘宝核心交易系统刚刚完成“去IOE”。十年过去，国内的数据库江湖已经换了模样，云数据库成为很多公司的选择，国产数据库创业公司也此起彼伏，而MySQL依然高居DB-Engines排行榜亚军，还是最受欢迎的开源数据库。根据中国信息通信研究院的《数据库发展研究报告（2021年）》，截至2021年6月，国产关系数据库产品共有81个，其中23个是基于MySQL二次开发的，占比为28.4%。

从2012年开始，我从淘宝DBA团队到手机淘宝数据团队，再到阿里云数加团队，直到2015年年底离职出来联合创立袋鼠云。至今在数据领域创业也有6年多了，目前主要管理袋鼠云集团旗下的两家子公司，分别是聚焦数字孪生的易知微和聚焦可观测业务运维的云掣。当年淘宝DBA团队的很多成员如今也在创业的路上，他们的工作也和MySQL数据库有或多或少的关系，比如本书的另外两位译者周振兴和张新铭。

在本书第3版出版的时候，MySQL还是5.5版本。此后在2015年发布的5.7版和2018年发布的8.0版是两个重要的版本，目前的最新版本是2022年4月发布的8.0.29。本书的最新版本中增加了大量对MySQL 5.7和8.0版本新特性的介绍，删除了一些在新版本中已经废弃或者不再常用的功能。另外，本书还增加了对云数据库的介绍，减少了在官方文档中已有的基础使用和配置相关的内容。这些年，MySQL在互联网的大量场景中被大规模地应用和验证，使得本书第4版在继续关注高性能之外，还使用了更多的篇幅来介绍如何实现MySQL的大规模可扩展应用，这是相比第3版最大的不同，所以本书还增加了新卖点“经过大规模运维验证的策略”。

本书的翻译依然是团队合作的结晶，其中我负责第1、2、3、4、5、6、13章和附录的翻译，周振兴负责第7、8、9、10章的翻译，张新铭负责第11、12章的翻译，另外一位同事刘开阳也参与了部分章节的翻译和审稿工作。在第3版的基础之上，这次整体工作的进度还比较顺利。但创业公司各种事务繁忙，译者们的时间精力都有限，难免存在疏漏之处，恳请读者不吝指出。

我要感谢易知微和云掣团队的同事，让我能够在工作之余抽出时间完成本书的翻译。当然更要感谢我的妻子Lalla，在我以创业和翻译为借口之际，毫无怨言地照顾我和我们的孩子则一的生活。是的，十年之后，我们有了新的家庭成员，这个月他已经三岁啦！

宁海元，易知微&云掣CEO

2022年6月于未来科技城

序

自近20年前《高性能MySQL》（第1版）问世以来，它已经成为每一位被新聘成为数据库管理员、系统工程师或数据库开发人员桌上的第一本书。

最初当Jeremy Zawodny和Derek Balling着手写一本关于大规模运行MySQL的书，为多年的谜题带来清晰的思路和结构时，它注定会成为MySQL世界的经典。经过十几年的多版更新，有些原始内容和后续更新一直有效，有些则不然。

MySQL本身在进步，MySQL社区改变了很多，我们使用MySQL的方式也改变了。现在，在第4版中，Silvia和Jeremy承担了艰巨的任务，要把这本经典图书更新到当今时代，他们是完成这项任务的最佳搭档。

我在MySQL社区工作的这段时间（现在已经超过20年了！），始终不变的一件事情，就是变化。每个人使用MySQL（或者说通用数据库）的方式都略有不同，对它有不同的期望。每个人都会做出一些好的决定，也会做出一些善意但值得怀疑的决定。当然，也总会做出一些糟糕的决定。有时，进步是容易的，但有时需要明智的建议，以及直接从专家那里学到思考问题的新方法。

Silvia和Jeremy就是这样的专家。从体系结构、优化、复制、备份等各个方面，我们都将受益于他们在MySQL领域分享的丰富经验。在新的第4版中，许多主题得到了新的处理，许多过时的知识被删除，许多错误被纠正，这些工作最终为本书带来了新的内容和新的风格。

就像最初的第1版一样（现在来看它的内容已经很老了，而且内容较少），第4版承诺将帮助最新一代的开发人员、数据库管理人员和他们的老板进入MySQL的新世界；这个世界有时很令人兴奋，但有时也可能让人抓狂。

谢谢Silvia和Jeremy，感谢你们为培养下一代MySQL极客所做的辛勤工作，这些人将确保这个世界上数据的安全，也将确保全球顶级网站和其他数据驱动的系统在其业务峰值时性能的卓越。

祝贺你们，在新冠肺炎病毒肆虐和其他各种事情干扰的情况下，依然完成了这项工作。我们其他人将确保为所有新的数据库管理员提供这本书。

Jeremy Cole

于内华达州里诺附近

2021年10月

前言

由Oracle维护的官方文档为你提供了安装、配置和与MySQL交互所需的基础知识。本书可以作为官方文档的配套图书，帮助你了解如何更好地利用MySQL作为某个应用场景的强大数据平台。

第4版扩展了在操作数据库过程中的合规性和安全性方面的内容，这两方面所扮演的角色日益重要。隐私法和数据主权等新的现实问题改变了公司构建产品的方式，这自然会给技术架构的发展带来新的复杂性。

这本书是写给谁的

这本书首先是为那些希望在运行MySQL方面提高专业技能的工程师准备的。这个版本假设读者熟悉关系数据库管理系统（RDBMS）的基本原理。我们还假设读者有常规系统管理、网络和操作系统维护方面的经验。

我们将为你提供经过验证的策略，让你可以使用现代体系结构及更先进的工具和实践来大规模运行MySQL。

最后，我们希望你能从本书中获得MySQL内部原理和扩展策略的相关知识，能够帮助你在组织中扩展数据存储层。我们还希望你从书中新发现的见解，可以帮助你学习和实践一种系统的方法，用来设计、维护和检测基于MySQL的系统架构。

第**4**版有什么不同

《高性能MySQL》多年来一直是数据库工程社区的一部分，从第1版到第3版的3本书分别发布于2004年、2008年和2012年。在以前的版本中，我们专注于深入剖析内部设计，解释各种调优设置的含义，并提供有效更改设置的知识，以此来达到我们的目标，告诉开发者和管理员如何优化MySQL的每一点性能。这个版本保持了相同的目标，但有不同的侧重点。

自第3版出版以来，MySQL发布了3个新的大版本，其生态系统发生了很大变化。它在工具范围上大大扩展，突破了Perl和Bash脚本，并发展出成熟的工具解决方案。全新的开源项目已经建立，这改变了企业管理MySQL的方式。

传统的数据库管理员（DBA）的角色也在不断演变。业内有一个古老的笑话，DBA的全称是“不要费心去问”（Don't Bother Asking）。DBA在软件开发生命周期（SDLC）中以“减速带”而闻名，这并不是因为他们有一种守旧的态度，而仅仅是因为数据库的发展速度不如SDLC的其他部分快。

有了《数据库可靠性工程：数据库系统设计与运维指南》（莱恩·坎贝尔、夏丽蒂·梅杰斯著，O'Reilly出版）等书，这已经成为一个新的现实：技术组织更多地将数据库工程师视为业务增长的促进因素，而不是所有数据库的唯一操作员。DBA的主要日常工作曾经涉及schema设计和查询优化，现在他们负责向开发人员传授这些技能，并管理系统，使开发人员能够快速、安全地部署自己的模式变更。

基于这些变化，我们认为不应该再将本书的重点放在优化MySQL以将性能提高几个百分点上，而应当是为人们提供他们所需要的信息，以便就如何最好地使用MySQL做出明智

的决定。这首先要理解MySQL是如何设计的，也就是要理解MySQL擅长做什么，不擅长做什么。[1]MySQL的现代版本提供了合理的默认值，除非你遇到非常具体的扩展问题，否则几乎不需要进行任何调整。现代团队需要更多地处理模式更改、合规性问题和分片。我们希望本书能够全面指导当下的公司如何更好地大规模运行MySQL。

本书使用的约定

本书使用如下排版约定：

斜体（*Italic*）

用于表示对应的内容是新词汇、URL、邮箱地址、文件名和文件扩展名。

等宽字体（Constant width）

用于程序清单（包括段落中的），表示程序元素，如变量名或函数名、数据库、数据类型、环境变量、语句和关键字。

等宽加粗字体（Constant width bold）

用于显示命令或用户输入的文本。

等宽斜体（*Constant width italic*）

用于显示应该由用户提供或根据上下文确定的值。

这个图标表示提示或建议。

这个图标表示一般注释。

这个图标表示警告或者提醒。

O'Reilly在线学习平台（**O'Reilly Online Learning**）

近40年来，O'Reilly Media致力于提供技术和商业培训、知识和卓越见解，来帮助众多公司取得成功。

我们拥有独一无二的专家和革新者组成的庞大网络，他们通过图书、文章、会议和我们的在线学习平台分享他们的知识和经验。O'Reilly的在线学习平台允许你按需访问现场培训课程、深入的学习路径、交互式编程环境，以及O'Reilly和200多家其他出版商提供的大量文本和视频资源。有关的更多信息，请访问*http://oreilly.com*。

如何联系我们

请将对本书的评价和存在的问题通过如下地址告知出版者。

美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）

奥莱利技术咨询（北京）有限公司

O'Reilly的每一本书都有专属网站，你可以在那里找到图书的相关信息，包括勘误列表、示例代码及其他信息。本书的网址是：

https://oreil.ly/hiperfmysql_2e

关于本书的评论和技术性的问题，请发送电子邮件到：

bookquestions@oreilly.com

关于我们的图书、课程、会议和新闻的更多信息，请参阅我们的网站 *http://www.oreilly.com*。

在Facebook上找到我们：*http://facebook.com/oreilly*

在Twitter上关注我们：*http://twitter.com/oreillymedia*

在YouTube上观看我们：*http://www.youtube.com/oreillymedia*

---

[1] 有一种众所周知的情况，经常有人把**MySQL**作为队列来使用，然后还抱怨它为什么不好。最常被吐槽的原因是轮询新队列操作的开销、处理消息时锁定记录的成本，以及随着时间的推移数据量不断增长，队列表变得越来越笨重。

本书第4版的致谢

来自**Silvia**

首先，我要感谢我的家人。我的父母放弃了在埃及的稳定工作和生活，把我和我的兄弟带到了美国。还有我的丈夫Armea，感谢他在我职业生涯中一直支持我，帮助我迎接一个又一个挑战，并最终取得了这一成就。

我结束在中东的大学生活，从一名技术移民开始，实现了移居美国的梦想。在加利福尼亚州的一所州立大学获得学位后，我在纽约找到了一份工作。我还记得，本书的第2版是我用自己的钱买的大学教科书之外的第一本科技图书。我要感谢前几版的作者，他们教会我很多基础知识，让我在职业生涯中做好了管理数据库的准备。

我还要感谢与我共事过的很多同事。他们的鼓励让我写了本书的第4版，这本书在我职业生涯的早期教会了我很多东西。我要感谢SendGrid的前首席技术官Tim Jenkins，他雇用了我，从而使我有了这份可以做一辈子的工作。尽管我在面试时告诉他，他使用MySQL复制的方式不对，但他对我的信任是一艘驱动我前进的宇宙飞船。

我要感谢所有在技术领域的杰出女性，她们一直是我的支持者和啦啦队。特别感谢Camille Fournier和Nicole Forsgren博士，她们写的两本书影响了我过去几年的职业发展，改变了我对日常工作的看法。

感谢我在Twilio的团队。感谢Sean Kilgore使我成为一名出色的而不是只关心数据库的工程师。感谢John Martin，他是与我共事过的最乐观的人。感谢Laine Campbell和她的PalominoDB团队（后来被Pythian收购），他们在最艰难的岁月里给予我支持，教会了我很多东西，也感谢Baron Schwartz鼓励我写下自己的经历。

最后，感谢优秀的编辑Virginia Wilson，帮助我把源源不断的想法变成有意义的文字，并在这一过程中给予我很多的支持和恩惠。

来自**Jeremy**

Silvia找我一起写这本书的时候，正值大多数人生活中一个非常紧张的时期——从2020年开始的新冠肺炎病毒疫情全球大流行。我不确定是否想给自己的生活增加更多压力。我的妻子Selena告诉我，如果我不接受，我会后悔的，我知道最好不要和她争论。她一直支持我，鼓励我成为最好的自己。我将永远爱她，因为她愿意为我做一切事情。

致我的家人、同事和社区里的朋友们：没有你们，我无法取得今天的成就。你们教会了我如何成为今天的自己。我的事业是我和你们所有人的经历的汇总。你们教会我如何接受批评，如何以身作则，如何接受失败和重新振作，最重要的是，你们让我明白团队大于个人。

最后，我想感谢Silvia，感谢你相信我能为这本书带来共识之外的不同视角。我希望达到了你的期望。

致谢技术审阅者

我们还要在这里感谢所有的技术审阅者，是你们帮助这本书得以最终完成，谢谢大家付出的时间和努力：Aisha Imran、Andrew Regner、Baron Schwartz、Daniel Nichter、Hayley

Anderson、Ivan Mora Perez、Jam Leoni、Jaryd Remillard、Jennifer Davis、Jeremy Cole、Keith Wells、Kris Hamoud、Nick Vyzas、Shubheksha Jalan、Tom Krouper和Will Gunty。

第1章 MySQL架构

MySQL的架构特点使其可以被应用在很多场景中。尽管它并不完美，但足够灵活，从小型的个人网站到大型的企业应用它都可以工作得很好。为了最大限度地使用MySQL，你需要了解它的设计，以便能够用其所长，避其所短。

本章概述了MySQL服务端的架构、各种存储引擎之间的主要区别，以及这些区别的重要性。我们试图通过简化细节和演示案例来解释MySQL的原理。这些讨论无论是对数据库一无所知的新手，还是熟知其他数据库的专家，都十分有用。

MySQL的逻辑架构

如果能在脑海中构建出一幅MySQL各组件之间协同工作的架构图，那么这将有助于你深入理解MySQL服务器。图1-1展示了MySQL架构的逻辑视图。

最上层的客户端所包含的服务并不是MySQL独有的，大多数基于网络的客户端/服务器工具或服务器都有类似的服务，包括连接处理、身份验证、确保安全性等。

第二层是比较有意思的部分。大多数MySQL的核心功能都在这一层，包括查询解析、分析、优化、以及所有的内置函数（例如，日期、时间、数学和加密函数），所有跨存储引擎的功能也都在这一层实现：存储过程、触发器、视图等。

第三层是存储引擎层。存储引擎负责MySQL中数据的存储和提取。和GNU/Linux下的各种文件系统一样，每种存储引擎都有其优势和劣势。服务器通过存储引擎API进行通信。这些API屏蔽了不同存储引擎之间的差异，使得它们对上面的查询层基本上是透明的。存储引擎层还包含几十个底层函数，用于执行诸如“开始一个事务”或者“根据主键提取一行记录”等操作。但存储引擎不会去解析SQL[1]，不同存储引擎之间也不会相互通信，而只是简单地响应服务器的请求。

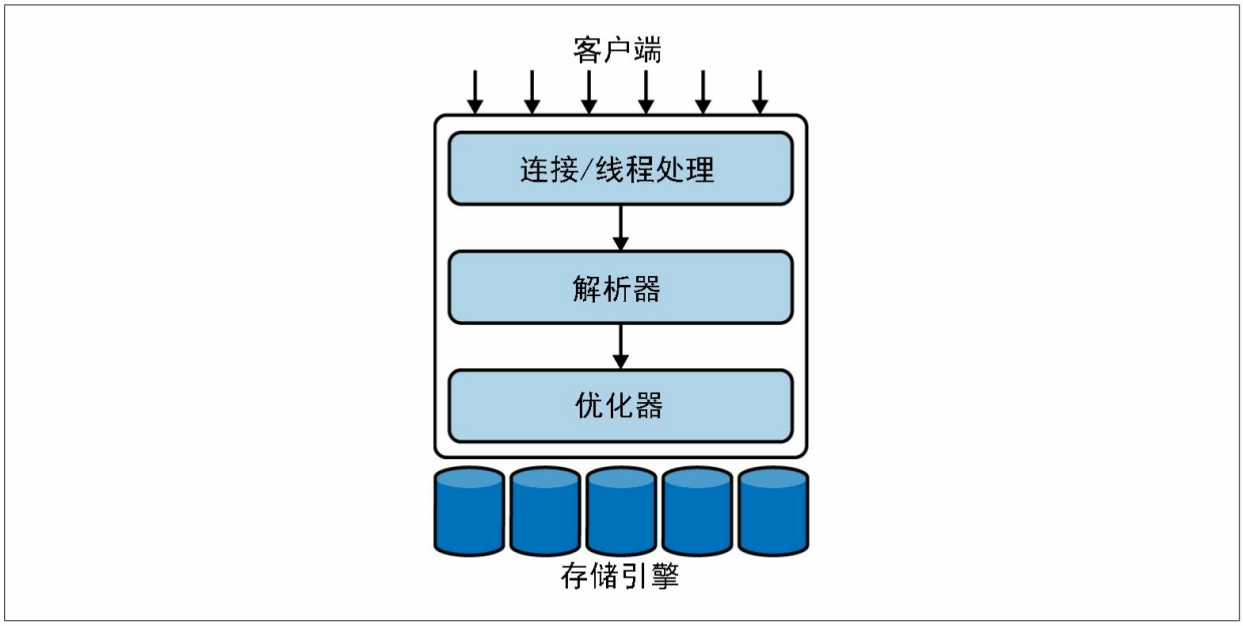

图1-1：MySQL服务器架构的逻辑视图

连接管理与安全性

默认情况下，每个客户端连接都会在服务器进程中拥有一个线程，该连接的查询只会在这个单独的线程中执行，该线程驻留在一个内核或者CPU上。服务器维护了一个缓存区，用于存放已就绪的线程，因此不需要为每个新的连接创建或者销毁线程。[2]

当客户端（应用）连接到MySQL服务器时，服务器需要对其进行身份验证。身份验证基于用户名、发起的主机名和密码。如果以跨传输层安全（TLS）的方式连接，还可以使用

X.509证书认证。客户端连接成功后，服务器会继续验证该客户端是否具有其发出的每个查询的权限（例如，是否允许客户端对world数据库中的Country表执行SELECT语句）。

优化与执行

MySQL解析查询以创建内部数据结构（解析树），然后对其进行各种优化，包括重写查询、决定表的读取顺序，以及选择合适的索引等。用户可以通过特殊关键字向优化器传递提示，从而影响优化器的决策过程。也可以请求服务器解释优化过程的各个方面，使用户可以知道服务器是如何进行优化决策的，并提供一个参考点，便于用户重构查询和schema、修改相关配置，使应用尽可能高效地运行。我们将在第8章介绍更多关于优化器的细节。

优化器并不关心表使用的是什么存储引擎，但存储引擎对于查询优化是有影响的。优化器会向存储引擎询问它的一些功能、某个具体操作的成本，以及表数据的统计信息。例如，一些存储引擎支持对某些查询有帮助的特定索引类型。更多关于索引与schema优化的内容，请参见第6章和第7章。

在旧版本中，MySQL可以使用内部查询缓存（query cache）来查看是否可以直接提供结果。但是，随着并发性的增加，查询缓存成为一个让人诟病的瓶颈。从MySQL 5.7.20版本开始，查询缓存已经被官方标注为被弃用的特性，并在8.0版本中被完全移除。尽管查询缓存不再是MySQL服务器的核心部分，但缓存被频繁请求的结果集依然是一个很好的实践。在本书的范围之外，一个流行的设计模式是在memcached或Redis中缓存数据。

并发控制

无论何时，只要有多个查询需要同时修改数据，就会产生并发控制问题。本章的目的是讨论MySQL在两个级别的并发控制：服务器级别与存储引擎级别。本章只简要地介绍MySQL如何控制并发读写，作为读者应理解本章其余内容的背景知识。

我们用一个传统的电子表格文件作为示例，来说明MySQL如何处理同一组数据上的并发工作。电子表格由行和列组成，很像数据库中的表。假设文件在只有你可以访问的笔记本电脑上，没有潜在的冲突，因为只有你可以对该文件进行更改。现在，想象一下你需要与一位同事合作制作电子表格，文件存放在一个你们都可以访问的共享服务器上。当你们需要同时对该文件进行更改时，会发生什么情况？如果还有一整个团队的人积极地尝试编辑、添加和删除这个电子表格中的单元格内容，那会怎么样呢？我们可以说他们应该轮流修改，但这样效率是极低的。我们需要一种允许并发访问大容量电子表格的方法。

读写锁

从电子表格中读取数据不会有什么麻烦，即使很多人同时读取也不会有问题。因为读取操作不会修改数据，所以不太可能出错。如果有人试图删除A25单元格中的内容，而其他人正在读取电子表格，会发生什么情况？这要视情况而定，读取者可能遇到报错退出，也可能得到不一致的数据视图。为安全起见，即使是从电子表格中读取数据也需要特别小心。

如果把上述电子表格看作数据库中的表，很容易发现也会有同样的问题。从很多方面来说，电子表格就是一张简单的数据库表。修改数据库表中的记录，和删除或者修改电子表格文件中的单元格内容十分类似。

并发控制这一经典问题的解决方案相当简单。处理并发读/写访问的系统通常实现一个由两种锁类型组成的锁系统。这两种锁通常被称为共享锁（shared lock）和排他锁（exclusive lock），也叫读锁（read lock）和写锁（write lock）。

先不考虑具体的锁定机制，锁的概念可以如下描述：资源上的读锁是共享的，或者说是相互不阻塞的。多个客户端可以同时读取同一个资源而互不干扰。写锁则是排他的，也就是说，一个写锁既会阻塞读锁也会阻塞其他的写锁，这是出于安全策略的考虑，只有这样才能确保在特定的时间点只有一个客户端能执行写入，并防止其他客户端读取正在写入的资源。

在实际的数据库系统中，每时每刻都在发生锁定：当某个客户端在修改某一部分数据时，MySQL会通过锁定防止其他客户端读取同一数据。如果数据库服务器以可接受的方式执行，锁的管理速度足够快，那么不会引起客户端的感知。我们将在第8章中讨论如何调整查询以避免锁引起的性能问题。

锁的粒度

一种提高共享资源并发性的方式就是让锁定对象更有选择性。尽量只锁定包含需要修改的部分数据，而不是所有的资源。更理想的方式是，只对需要修改的数据片段进行精确的锁定。任何时候，让锁定的数据量最小化，理论上就能保证在给定资源上同时进行更改操

作，只要被修改的数据彼此不冲突即可。

问题是加锁也需要消耗资源。锁的各种操作，包括获取锁、检查锁是否空闲、释放锁等，都会增加系统的开销。如果系统花费大量的时间来管理锁，而不是存取数据，那么系统的性能可能会受影响。

锁定策略是锁开销和数据安全性之间的平衡，这种平衡会影响性能。大多数商业数据库系统没有提供太多的选择，一般都是在表中施加行级锁（row level lock），为了在锁比较多的情况下尽可能地提供更好的性能，锁的实现方式非常复杂。锁是数据库实现一致性保证的方法。数据库操作专家必须深入源代码，才能确定合适的配置，以优化速度与数据安全之间的平衡。

而MySQL则提供了多种选择。每种MySQL存储引擎都可以实现自己的锁策略和锁粒度。在设计存储引擎时，锁的管理是一个非常重要的决定。将锁粒度固定在某个级别，可以提高某些应用场景下的性能，但同时会使其不适合另外一些应用场景。好在MySQL提供了多种存储引擎，而不是单一的通用解决方案。下面让我们来看两种最重要的锁策略。

表锁

表锁（table lock）是MySQL中最基本也是开销最小的锁策略。表锁非常类似于前文描述的电子表格的锁机制：它会锁定整张表。当客户端想对表进行写操作（插入、删除、更新等）时，需要先获得一个写锁，这会阻塞其他客户端对该表的所有读写操作。只有没有人执行写操作时，其他读取的客户端才能获得读锁，读锁之间不会相互阻塞。

表锁有一些变体，可以在特定情况下提高性能。例如，READ LOCAL表锁支持某些类型的并发写操作。写锁队列和读锁队列是分开的，但写锁队列的优先级绝对高于读队列。[3]

行级锁

使用行级锁（row lock）可以最大程度地支持并发处理（也带来了最大的锁开销）。回到电子表格的类比，行级锁等同于锁定电子表格中的某一行。这种策略允许多人同时编辑不同的行，而不会阻塞彼此。这使得服务器可以执行更多的并发写操作，带来的代价则是需要承担更多开销，以跟踪谁拥有这些行级锁、已经锁定了多长时间、行级锁的类型，以及何时该清理不再需要的行级锁。

行级锁是在存储引擎而不是服务器中实现的。服务器通常[4]不清楚存储引擎中锁的实现方式。在本章的后续内容甚至整本书中都可以看到，每种存储引擎都以自己的方式来实现锁。

事务

在理解事务的概念之前，接触数据库系统的其他高级特性还言之过早。事务就是一组SQL语句，作为一个工作单元以原子方式进行处理。如果数据库引擎能够成功地对数据库应用整组语句，那么就执行该组语句。如果其中有任何一条语句因为崩溃或其他原因无法执行，那么整组语句都不执行。也就是说，作为事务的一组语句，要么全部执行成功，要么全部执行失败。

本节的内容并非专属于MySQL，如果读者已经熟悉ACID事务的概念，可以直接跳到本章后面的“MySQL中的事务”一节。

银行应用是解释事务必要性的经典例子。[5]假设一个银行的数据库有两张表：支票表（checking）和储蓄表（savings）。现在要从用户Jane的支票账户转移200美元到她的储蓄账户，那么需要至少三个步骤：

1. 确保支票账户的余额高于200美元。
2. 从支票账户的余额中减去200美元。
3. 在储蓄账户的余额中增加200美元。

以上三步操作必须打包在一个事务中，以保证一旦其中任何一步失败，都能够回滚所有的操作。

可以用START TRANSACTION语句启动事务，然后要么使用COMMIT提交事务将修改的数据持久保留，要么使用ROLLBACK撤销所有的修改。本例中的事务的SQL语句可能如下：

```
1  START  TRANSACTION;
2  SELECT balance FROM checking WHERE customer_id = 10233276;
3  UPDATE checking SET balance = balance - 200.00 WHERE customer_id = 10233276;
4  UPDATE savings SET balance = balance + 200.00 WHERE customer_id = 10233276;
5  COMMIT;
```

单纯的事务概念并不是故事的全部。试想一下，如果数据库在执行第4条语句时崩溃了，会发生什么？天知道，客户可能会损失200美元。再假如，在执行到第3条语句和第4条语句之间时，另外一个进程要删除支票账户的所有余额会发生什么？结果可能是银行不知不觉白白给了Jane 200美元。

在这一系列操作中，有更多的失败可能性。连接可能会断开、会超时，甚至数据库服务器在操作执行过程中会崩溃。这就是为什么存在高度复杂且缓慢的两阶段提交系统的典型原因：为了应对各种失败场景。

除非系统通过严格的ACID测试，否则空谈事务的概念是不够的。ACID代表原子性（atomicity）、一致性（consistency）、隔离性（isolation）和持久性（durability）。一个确保数据安全的事务处理系统，必须满足这些密切相关的标准。

原子性（atomicity）

一个事务必须被视为一个不可分割的工作单元，整个事务中的所有操作要么全部提交

成功，要么全部失败回滚。对于一个事务来说，不可能只执行其中的一部分操作，这就是事务的原子性。

一致性（consistency）

数据库总是从一个一致性状态转换到下一个一致性状态。在前面的例子中，一致性确保了，即使在执行第3、4条语句之间时系统崩溃，支票账户中也不会损失200美元。如果事务最终没有提交，该事务所做的任何修改都不会被保存到数据库中。

隔离性（isolation）

通常来说，一个事务所做的修改在最终提交以前，对其他事务是不可见的，这就是隔离性带来的结果。在前面的例子中，当执行完第3条语句、第4条语句还未开始时，此时有另外一个账户汇总程序开始运行，其看到的支票账户的余额并没有被减去200美元。后面我们讨论隔离级别（isolation level）的时候，会发现为什么我们要说“通常来说”是不可见的。

持久性（durability）

一旦提交，事务所做的修改就会被永久保存到数据库中。此时即使系统崩溃，数据也不会丢失。持久性是一个有点模糊的概念，实际上持久性也分很多不同的级别。有些持久性策略能够提供非常强的安全保障，而有些则未必。而且不可能有100%的持久性保障（如果数据库本身就能做到真正的持久性，那么备份又怎么能增加持久性呢？）。

ACID事务和InnoDB引擎提供的保证是MySQL中最强大、最成熟的特性之一。虽然它们在吞吐量方面做了一定的权衡，但如果应用得当，就可以避免在应用层实现大量复杂逻辑。

隔离级别

隔离性在实际操作中比看起来复杂得多。ANSI SQL标准定义了4种隔离级别。如果你是数据库领域的新手，我们强烈建议你在阅读特定的MySQL实现之前先熟悉ANSI SQL[6]的通用标准。这个通用标准的目标是定义在事务内外可见和不可见的更改的规则。较低的隔离级别通常允许更高的并发性，并且开销也更低。

每种存储引擎实现的隔离级别都不尽相同。如果你熟悉其他的数据库产品，可能会发现某些特性和你期望的会有些不一样（因此本节不打算讨论更详细的内容）。读者可以根据所选择的存储引擎，查阅相关的手册。

下面简单地介绍一下这4种隔离级别。

READ UNCOMMITTED（未提交读）

在READ UNCOMMITTED级别，在事务中可以查看其他事务中还没有提交的修改。这个隔离级别会导致很多问题，从性能上来说，READ UNCOMMITTED不会比其他级别好太多，却缺乏其他级别的很多好处，除非有非常必要的理由，在实际应用中一般很少使用。

读取未提交的数据，也称为脏读（dirty read）。

READ COMMITTED（提交读）

大多数数据库系统的默认隔离级别是READ COMMITTED（但MySQL不是）。READ COMMITTED满足前面提到的隔离性的简单定义：一个事务可以看到其他事务在它开始之后提交的修改，但在该事务提交之前，其所做的任何修改对其他事务都是不可见的。这个级别仍然允许不可重复读（nonrepeatable read），这意味着同一事务中两次执行相同语句，可能会看到不同的数据结果。

REPEATABLE READ（可重复读）

REPEATABLE READ解决了READ COMMITTED[7]级别的不可重复读问题，保证了在同一个事务中多次读取相同行数据的结果是一样的。但是理论上，可重复读隔离级别还是无法解决另外一个幻读（phantom read）的问题。所谓幻读，指的是当某个事务在读取某个范围内的记录时，另外一个事务又在该范围内插入了新的记录，当之前的事务再次读取该范围的记录时，会产生幻行（phantom row）。InnoDB和XtraDB存储引擎通过多版本并发控制（MVCC，Multiversion Concurrency Control）解决了幻读的问题。本章稍后会对此做进一步讨论。

REPEATABLE READ是MySQL默认的事务隔离级别。

SERIALIZABLE（可串行化）

SERIALIZABLE是最高的隔离级别。该级别通过强制事务按序执行，使不同事务之间不可能产生冲突，从而解决了前面说的幻读问题。简单来说，SERIALIZABLE会在读取的每一行数据上都加锁，所以可能导致大量的超时和锁争用的问题。实际应用中很少用到这个隔离级别，除非需要严格确保数据安全且可以接受并发性能下降的结果。

表1-1概述了这几种隔离级别之间的利与弊。

表1-1：ANSI SQL的隔离级别

隔离级别	脏读	不可重复读	幻读	加锁读
READ UNCOMMITTED	是	是	是	否
READ COMMITTED	否	是	是	否
REPEATABLE READ	否	否	是	否
SERIALIZABLE	否	否	否	是

死锁

死锁是指两个或多个事务相互持有和请求相同资源上的锁，产生了循环依赖。当多个事务试图以不同的顺序锁定资源时会导致死锁。当多个事务锁定相同的资源时，也可能会发生死锁。例如，设想运行下面两个针对主键为（stock_id，date）的StockPrice表的事务：

事务1

```
START TRANSACTION;
UPDATE StockPrice SET close = 45.50 WHERE stock_id = 4 and date = '2020-05-01';
UPDATE StockPrice SET close = 19.80 WHERE stock_id = 3 and date = '2020-05-02';
COMMIT;
```

事务2

```
START TRANSACTION;
UPDATE StockPrice SET high = 20.12 WHERE stock_id = 3 and date = '2020-05-02';
UPDATE StockPrice SET high = 47.20 WHERE stock_id = 4 and date = '2020-05-01';
COMMIT;
```

每个事务都开始执行第一个查询，在处理过程中会更新一行数据，同时在主键索引和其他唯一索引中将该行锁定。然后，每个事务将在第二个查询中尝试更新第二行数据，却发现该行已经被锁定。这两个事务将永远等待对方完成，除非有其他因素介入解除死锁。我们将在第7章进一步介绍索引如何随着模式的发展而影响或破坏查询的性能。

为了解决这个问题，数据库系统实现了各种死锁检测和锁超时机制。更复杂的系统，比如InnoDB存储引擎，检测到循环依赖后会立即返回一个错误信息。这可能是一件好事——否则，死锁将表现为非常缓慢的查询。还有一种方式，当超过锁等待超时的时间限制后直接终止查询，这样做通常来说不太好。InnoDB目前处理死锁的方式是将持有最少行级排他锁的事务回滚（这是一种最容易回滚的近似算法）。

锁的行为和顺序是和存储引擎相关的。同样的一系列查询语句，有些存储引擎会产生死锁，有些则不会。死锁的产生有双重原因：有些是因为真正的数据冲突，这种情况通常很难避免，但有些则完全是由于存储引擎的实现方式导致的。[8]

一旦发生死锁，如果不回滚其中一个事务（部分或全部），就无法打破死锁。对于事务型的系统，这是无法避免的，所以应用程序在设计时必须考虑如何处理死锁。大多数情况下只需要重新从头开始执行被回滚的事务即可，除非又遇到另一个死锁。

事务日志

事务日志有助于提高事务的效率。存储引擎只需要更改内存中的数据副本，而不用每次修改磁盘中的表，这会非常快。然后再把更改的记录写入事务日志中，事务日志会被持久化保存在硬盘上。因为事务日志采用的是追加写操作，是在硬盘中一小块区域内的顺序I/O，而不是需要写多个地方的随机I/O，所以写入事务日志是一种相对较快的操作。最后会有一个后台进程在某个时间去更新硬盘中的表。因此，大多数使用这种技术（write-ahead logging，预写式日志）的存储引擎修改数据最终需要写入磁盘两次。

如果修改操作已经写入事务日志，那么即使系统在数据本身写入硬盘之前发生崩溃，存储引擎仍可在重新启动时恢复更改。具体的恢复方法则因存储引擎而异。

MySQL中的事务

存储引擎是驱动如何从硬盘中存储和检索数据的软件。虽然MySQL传统上提供了许多支持事务的存储引擎，但InnoDB现在已经成为金标准，是推荐使用的引擎。这里描述的事

务原语将基于InnoDB引擎中的事务。

理解**AUTOCOMMIT**

默认情况下，单个INSERT、UPDATE或DELETE语句会被隐式包装在一个事务中并在执行成功后立即提交，这称为自动提交（AUTOCOMMIT）模式。通过禁用此模式，可以在事务中执行一系列语句，并在结束时执行COMMIT提交事务或ROLLBACK回滚事务。

在当前连接中，可以使用SET命令设置AUTOCOMMIT变量来启用或禁用自动提交模式。启用可以设置为1或者ON，禁用可以设置为0或者OFF。如果设置了AUTOCOMMIT=0，则当前连接总是会处于某个事务中，直到发出COMMIT或者ROLLBACK，然后MySQL会立即启动一个新的事务。此外，当启用AUTOCOMMIT时，也可以使用关键字BEGIN或者START TRANSACTION来开始一个多语句的事务。修改AUTOCOMMIT的值对非事务型的表不会有任何影响，这些表没有COMMIT或者ROLLBACK的概念。

还有一些命令，当在活动的事务中发出时，会导致MySQL在事务的所有语句执行完毕前提交当前事务。这些通常是进行重大更改的DDL命令，如ALTER TABLE，但LOCK TABLES和其他一些语句也具有同样的效果。有关会导致自动提交事务的完整命令列表，请查看对应版本的官方文档。

MySQL可以通过执行SET TRANSACTION ISOLATION LEVEL命令来设置隔离级别。新的隔离级别会在下一个事务开始的时候生效。可以在配置文件中设置整个服务器的隔离级别，也可以只改变当前会话的隔离级别：

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

建议最好在服务器级别设置最常用的隔离，并且只在显式情况下修改。MySQL可以识别所有4个ANSI标准的隔离级别，InnoDB也支持这些隔离级别。

在事务中混合使用存储引擎

MySQL不在服务器层管理事务，事务是由下层的存储引擎实现的。所以在同一个事务中，混合使用多种存储引擎是不可靠的。

假设在事务中混合使用事务表和非事务表（例如，InnoDB和MyISAM表），如果一切顺利，事务将正常工作。如果需要回滚，则无法撤销对非事务表的更改。这会使数据库处于不一致的状态，可能难以恢复，并使整个事务问题变得毫无意义。所以，为每张表选择合适的存储引擎，并不惜一切代价避免在应用中混合使用存储引擎是非常重要的。

在非事务表中执行事务相关操作的时候，MySQL通常不会发出提醒，也不会报错。有时候只有回滚的时候才会发出一个警告：某些非事务表中的变更无法回滚。但在大多数情况下，对非事务表的操作都不会有提示。

最好不要在应用程序中混合使用存储引擎。失败的事务可能导致不一致的结果，因为某些部分可以回滚，而其他部分不能回滚。

隐式锁定和显式锁定

InnoDB使用两阶段锁定协议（two-phase locking protocol）。在事务执行期间，随时都可

以获取锁，但锁只有在提交或回滚后才会释放，并且所有的锁会同时释放。前面描述的锁定机制都是隐式的。InnoDB会根据隔离级别自动处理锁。

另外，InnoDB还支持通过特定的语句进行显式锁定，这些语句不属于SQL规范：[9]，[10]

```
SELECT ... FOR SHARE
SELECT ... FOR UPDATE
```

MySQL还支持LOCK TABLES和UNLOCK TABLES命令，这些命令在服务器级别而不在存储引擎中实现。如果需要事务，应该使用支持事务的存储引擎。因为InnoDB支持行级锁，所以没必要使用LOCK TABLES。

LOCK TABLES命令和事务之间的交互非常复杂，并且在一些服务器版本中存在意想不到的行为。因此，本书建议，除了在禁用AUTOCOMMIT的事务中可以使用之外，其他任何时候都不要显式地执行LOCK TABLES，不管使用的是什么存储引擎。

多版本并发控制

MySQL的大多数事务型存储引擎使用的都不是简单的行级锁机制。它们会将行级锁和可以提高并发性能的多版本并发控制（MVCC）技术结合使用。不仅是MySQL，包括Oracle、PostgreSQL以及其他一些数据库系统也都使用了MVCC，但各自的实现机制不尽相同，因为MVCC如何工作没有统一的标准。

可以认为MVCC是行级锁的一个变种，但是它在很多情况下避免了加锁操作，因此开销更低。根据其实现方式，不仅实现了非阻塞的读操作，写操作也只锁定必要的行。

MVCC的工作原理是使用数据在某个时间点的快照来实现的。这意味着，无论事务运行多长时间，都可以看到数据的一致视图，也意味着不同的事务可以在同一时间看到同一张表中的不同数据！如果你之前没有这方面的概念，这句话听起来可能有点让人迷惑，熟悉了以后你会发现还是很容易理解的。

每个存储引擎实现MVCC的方式都不同。其中一些变体包括乐观并发控制和悲观并发控制。我们可以通过图1-2所示的序列图解释InnoDB的行为[11]，以此来展示MVCC的一种实现方式。

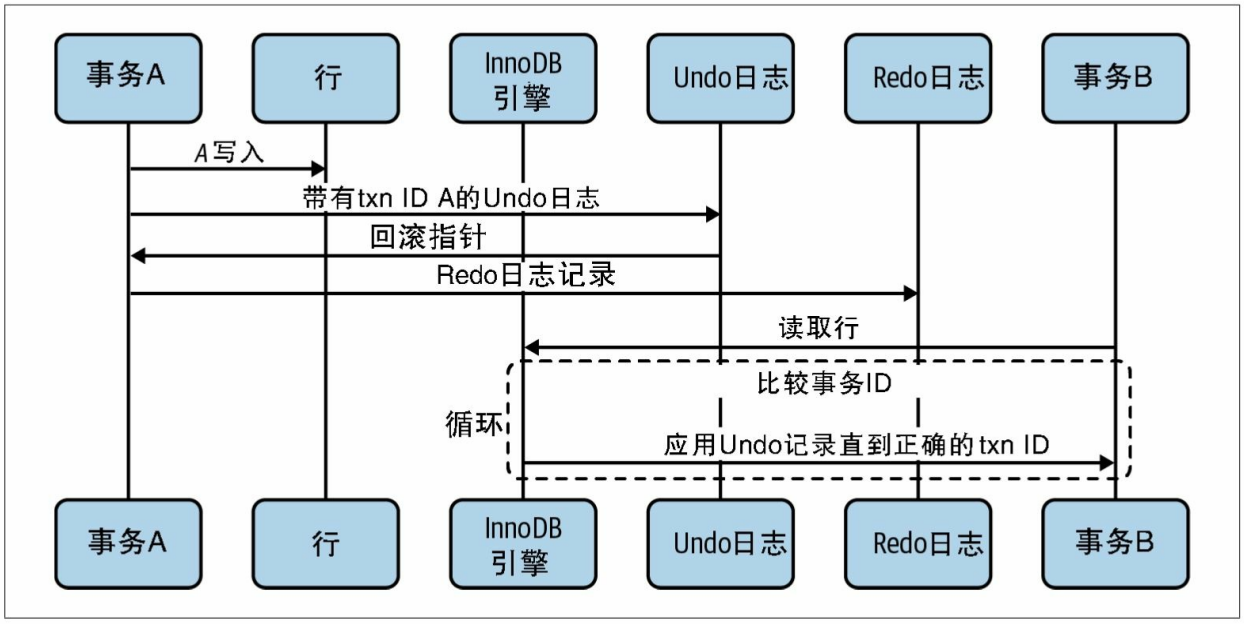

图1-2：跨不同事务处理同一行多个版本的序列图

InnoDB通过为每个事务在启动时分配一个事务ID来实现MVCC。该ID在事务首次读取任何数据时分配。在该事务中修改记录时，将向Undo日志写入一条说明如何恢复该更改的Undo记录，并且事务的回滚指针指向该Undo日志记录。这就是事务如何在需要时执行回滚的方法。

当不同的会话读取聚簇主键索引记录时，InnoDB会将该记录的事务ID与该会话的读取视图进行比较。如果当前状态下的记录不应可见（更改它的事务尚未提交），那么Undo日志记录将被跟踪并应用，直到会话达到一个符合可见条件的事务ID。这个过程可以一直循环到完全删除这一行的Undo记录，然后向读取视图发出这一行不存在的信号。

事务中的记录可以通过在记录的“info flags”中设置“deleted”位来删除。这在Undo日志中也被作为“删除标记”进行跟踪。

值得注意的是，所有Undo日志写入也都会写入Redo日志，因为Undo日志写入是服务器崩溃恢复过程的一部分，并且是事务性的。[12]这些Redo日志和Undo日志的大小也是高并发事务工作机制中的重要影响因素。我们将在第5章更详细地介绍它们的配置。

在记录中保留这些额外信息带来的结果是，大多数读取查询都不再需要获取锁。它们只是尽可能快地读取数据，确保仅查询符合条件的行即可。缺点是存储引擎必须在每一行中存储更多的数据，在检查行时需要做更多的工作，并处理一些额外的内部操作。

MVCC仅适用于REPEATABLE READ和READ COMMITTED隔离级别。READ UNCOMMITTED与MVCC不兼容[13]，是因为查询不会读取适合其事务版本的行版本，而是不管怎样都读最新版本。SERIALIZABLE与MVCC也不兼容，是因为读取会锁定它们返回的每一行。

复制

MySQL被设计用于在任何给定时间只在一个节点上接受写操作。这在管理一致性方面具有优势，但在需要将数据写入多台服务器或多个地区时，会导致需要做出取舍。MySQL提供了一种原生方式来将一个节点执行的写操作分发到其他节点，这被称为复制。在MySQL中，源节点为每个副本节点提供一个线程，该线程作为复制客户端登录，当写入发生时会被唤醒，发送新数据。在图1-3中，我们展示了此设置的一个简单示例，通常将它称为一主多副的多个MySQL服务器拓扑树。

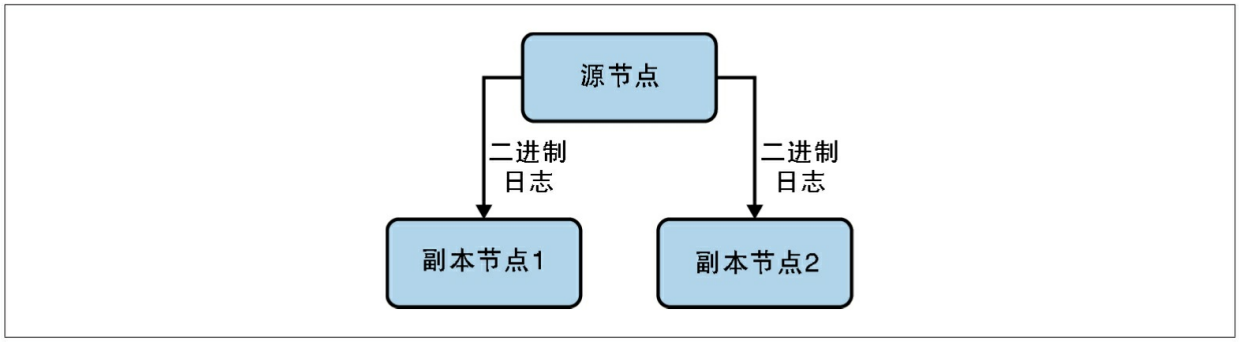

图1-3：MySQL服务器复制拓扑的简化视图

对于在生产环境中运行的任何数据，都应该使用复制并至少有三个以上的副本，理想情况下应该分布在不同的地区（在云托管环境中，称为region）用于灾难恢复计划。

多年来，MySQL中的复制变得十分复杂。全局事务标识符、多源复制、副本上的并行复制和半同步复制是一些主要的更新。我们将在第9章详细讨论复制。

数据文件结构

在8.0版本中，MySQL将表的元数据重新设计为一种数据字典，包含在表的*.ibd*文件中。这使得表结构上的信息支持事务和原子级数据定义更改。在操作期间，我们不再仅仅依赖information_schema来检索表定义和元数据，而是引入了字典对象缓存，这是一种基于最近最少使用（LRU）的内存缓存，包括分区定义、表定义、存储程序定义、字符集和排序信息。服务器访问表的元数据的方式的这一重大变化减少了I/O，非常高效。特别是当前访问最活跃的那些表，在缓存中最常出现。每个表的*.ibd*和*.frm*文件被替换为已经被序列化的字典信息（*.sdi*）。

InnoDB引擎

InnoDB是MySQL的默认事务型存储引擎，也是最重要、使用最广泛的引擎。它是为处理大量短期事务而设计的，这些事务通常是正常提交的，很少会被回滚。InnoDB的性能和自动崩溃恢复特性，使得它在非事务型存储需求中也很流行。如果想研究存储引擎，比起花同样时间学习所有的存储引擎，深入研究以尽可能多地了解InnoDB是更值得的选择。

最佳实践是使用InnoDB存储引擎作为所有应用程序的默认引擎。在好几个大版本之前，MySQL已经将InnoDB作为默认引擎，这让事情变得简单了。

InnoDB是MySQL默认的通用存储引擎。默认情况下，InnoDB将数据存储在一系列的数据文件中，这些文件统被称为表空间（tablespace）。表空间本质上是一个由InnoDB自己管理的黑盒。

InnoDB使用MVCC来实现高并发性，并实现了所有4个SQL标准隔离级别。InnoDB默认为REPEATABLE READ隔离级别，并且通过间隙锁（next-key locking）策略来防止在这个隔离级别上的幻读：InnoDB不只锁定在查询中涉及的行，还会对索引结构中的间隙进行锁定，以防止幻行被插入。

InnoDB表是基于聚簇索引构建的，我们将在第8章讨论schema设计时详细介绍聚簇索引。InnoDB的索引结构与MySQL其他大部分存储引擎有很大的不同。聚簇索引提供了非常快速的主键查找。但是，因为二级索引（secondary index，非主键索引）需要包含主键列，如果主键较大，则其他索引也会很大。如果表中的索引较多，主键应当尽量小。

InnoDB内部做了很多优化。其中包括从磁盘预取数据的可预测性预读、能够自动在内存中构建哈希索引以进行快速查找的自适应哈希索引（adaptive hash index），以及用于加速插入操作的插入缓冲区（insert buffer）。我们将在本书第4章讨论这些内容。

InnoDB的行为是非常复杂的，如果你正在使用InnoDB，强烈建议阅读MySQL官方手册中“InnoDB锁定和事务模型”（参见链接5）一节。由于其MVCC架构，在使用InnoDB构建应用程序之前，有许多微妙之处需要注意。存储引擎要为所有用户甚至包括修改数据的用户维持一致性的视图，这是非常复杂的工作。

作为事务型存储引擎，InnoDB可以通过一些机制和工具支持真正的在线“热”备份，包括Oracle专有的MySQL Enterprise Backup和开源的Percona XtraBackup。我们将在第10章详细介绍备份和恢复。

从MySQL 5.6开始，InnoDB引入了在线DDL，它最初只支持有限的使用场景，但在5.7和8.0版本中进行了扩充。就地（in-place）更改schema的机制允许在不使用完整表锁和外部工具的情况下进行特定的表更改操作，这大大提高了MySQL InnoDB表的可操作性。在第6章中，我们将介绍在线更改schema的选项，包括本机工具和外部工具。

JSON文档支持

JSON类型在5.7版本被首次引入InnoDB，它实现了JSON文档的自动验证，并优化了存储以允许快速读取，这是对旧版本只能使用BLOB类型来处理JSON文档作为折中的重大改进。除了支持新的数据类型，InnoDB还引入了SQL函数来支持在JSON文档上的丰富操作。MySQL 8.0.7的进一步改进增加了在JSON数组上定义多值索引的能力。将常用访问模式匹配到可以映射JSON文档值的函数这一特性可以进一步加快对JSON类型的读取访问查询。我们将在第6章的“JSON数据类型”一节中讨论JSON数据类型的使用和性能影响。

数据字典的变化

MySQL 8.0的另一个主要变化是删除了基于文件的表元数据存储，并将其转移到使用InnoDB表存储的数据字典中。这给所有类似修改表结构这样的操作带来了InnoDB的崩溃恢复事务的好处。

这一更改虽然大大改进了MySQL中数据定义的管理，但也需要我们对MySQL服务器的操作方式做出很大改变。最显著的变化是，以前依赖于表元数据文件的备份程序，现在必须查询新的数据字典以提取表定义。

原子**DDL**

最后，MySQL 8.0引入了原子数据定义更改。这意味着数据定义语句现在要么全部成功完成，要么全部失败回滚。这是通过创建DDL特定的Undo日志和Redo日志来实现的，InnoDB便依赖这两种日志来跟踪变更——这是InnoDB经过验证的设计，已经扩展到MySQL服务器的操作中。

小结

MySQL拥有分层的架构，上层是服务器级别的服务和查询执行层，下层则是存储引擎。虽然有很多不同作用的插件API，但存储引擎API是最重要的。MySQL通过API来与存储引擎交互需要处理的数据行，理解了这一点，就掌握了MySQL服务器架构的基本原理。

在过去的几个主要版本中，MySQL主要的改进核心在于InnoDB的演进。表元数据、用户认证、身份鉴权这些内部统计信息的管理也已经调整为使用InnoDB表来实现，而在几年前，使用的都是MyISAM引擎。Oracle官方在InnoDB引擎上加大了投入，这使得MySQL有了一些重大改进，例如原子DDL，更完善的online DDL，更强大的崩溃恢复能力，以及更安全的部署操作。

InnoDB是MySQL的默认存储引擎，它几乎能覆盖每一种使用场景。后面的章节我们将重点介绍InnoDB存储引擎，包括它的特性、性能及限制。从现在开始，我们将很少提到InnoDB以外的其他存储引擎了。

---

[1] **InnoDB**是一个例外，它会解析外键定义，因为**MySQL**服务端没有实现该功能。

[2] **MySQL 5.5**或者更新的版本提供了一个支持线程池（**Thread-Pooling**）插件的**API**，但并不常用到。线程池的常见实践是在访问层实现的，这些知识我们将在第**5**章中讨论。

[3] 强烈建议读者阅读官网上的文档中关于排他锁、共享锁、意图锁和记录锁的内容。

[4] 还有一种元数据锁，用于修改表名或者**schema**，另外，在**8.0**中还引入了**“**应用程序级别的锁功能**”**。在日常的数据更改过程中，内部锁则留给**InnoDB**引擎处理。

[5] 这个例子仅作为一个常规学术练习，实际上大多数银行更多依靠日常对账，而不是白天严格的交易操作。

[6] 欲了解更多信息，请阅读由**Adrian Coyler**编写的**ANSI SQL**摘要（参见链接**1**）和**Kyle Kingsbury**编写的一致性模型说明（参见链接**2**）。

[7] 原文这里是**READ UNCOMMITTED**，但结合上下文，此处应该是指**READ COMMITTED**。——译者注

[8] 正如你将在本章后面看到的，一些存储引擎会锁定整个表，而其他存储引擎则实现了更复杂的基于行的锁定。这些逻辑大部分都是在存储引擎层实现的。

[9] 这些锁定提示经常被滥用，应当尽量避免使用。

[10] **SELECT...FOR SHARE**是**MySQL 8.0**的新语句，取代了以前版本的**SELECT...LOCK IN SHARE MODE**。

[11] 建议阅读**Jeremy Cole**的这篇博客（参见链接**3**），以深入了解**InnoDB**中的记录结构。

[12] 关于**InnoDB**如何处理多个版本的记录的更多细节，请看**Jeremy Cole**的一篇博客（参见链接**4**）。

[13] **MVCC**没有正式的标准定义，不同的存储引擎和数据库实现的方式都极不相同，没有人能说任何一个是错的。

第2章 可靠性工程世界中的监控

监控系统是一个广泛的主题，在过去的几年中，这个主题很大程度上是由两本开创性的著作所塑造的，它们是《SRE：Google运维解密》及其后续著作《Google SRE工作手册》。自从这两本书出版以来，网站可靠性工程师（SRE）的公开招聘已经成为一个流行趋势。一些公司甚至将现有员工重新命名为某种“可靠性工程师”。

网站可靠性工程改变了团队对运营工作的思考方式。这是因为它包含了一系列的原则，使我们能够更容易地回答诸如下面这样的问题：

- 我们提供的是可接受的客户体验吗？
- 我们应该关注可靠性和可恢复性工作吗？
- 我们如何平衡新功能和工作量？

本章期望读者理解这些原则是什么。如果你还没有读过上述任何一本书，建议将《Google SRE工作手册》的这几章作为速成课程：

- 第1章，帮助我们更深入地理解生产中迈向服务水平性能管理背后的哲学。
- 第2章，介绍了如何实现服务水平目标（SLO）。
- 第5章，介绍了服务水平目标中的警报。

有些人可能会争辩说，严格来说，SRE实现并不是高性能MySQL的一部分，但我们不同意。Nicole Forsgren博士在她的*Accelerate*[1]一书中说：“我们的衡量方法应该关注结果，而不是输出”。有效管理MySQL的一个关键点在于对数据库的健康状况进行良好的监控。传统的监控是一条相对平坦的道路。由于SRE是一个新的领域，人们不太了解如何在MySQL中实现SRE原则。随着SRE原则持续得到认可，DBA的传统角色将不断改变，这包括DBA对监控系统的思维方式。

可靠性工程对**DBA**团队的影响

多年来，监控数据库性能依赖于对单服务器性能的深入研究。这么做仍然有很大的价值，但更多的是对被动响应的度量，比如对性能较差的服务器进行分析。在以前DBA团队作为守门人的年代，这是标准的操作程序，当时是不允许其他人知道数据库是如何操作的。

自谷歌引入可靠性工程以来，DBA的角色变得更加复杂，更像是网站可靠性工程师（SRE）或数据库可靠性工程师（DBRE）。团队必须优化他们的时间。服务级别帮助你定义客户满意的程度和标准，以便你在解决性能、可扩展性挑战等事情与开发内部工具之间做出时间权衡。让我们讨论一下监视MySQL以确保成功的客户体验所需的不同方法。

定义服务水平目标

在讨论如何衡量客户对数据库集群的性能是否满意之前，我们必须首先知道目标是什么，并使用通俗的说法来表述这些目标。这里有一些问题，可以作为组织中确定这些目标的引导话题：

- 衡量成功的合理指标是什么？
- 客户和我们的业务需求可以接受这些指标的哪些值？
- 在什么情况下会被视为处于降级状态？
- 什么时候处于完全失败的状态，需要尽快补救？

在某些情况下这些问题的答案是显而易见的（例如，源数据库故障，此时无法进行任何写入操作，这会导致业务暂停运作）。在某些情况下则不那么明显，比如周期性任务有时会占用所有数据库磁盘I/O，这会让其他任务突然变慢。在整个组织中，对正在测量的内容和原因有一个共同的理解，可以帮助指导关于团队优先事项的讨论。在整个组织中通过持续讨论达成共识，有助于指导你是否应在新特性上花费精力，或者是否需要在性能改进或稳定性方面投入更多。

在SRE实践中，这些关于客户满意度的讨论将使团队在服务水平指标（SLI）、服务水平目标（SLO）和服务水平协议（SLA）方面就什么对业务有益达成一致。让我们从定义这些术语的含义开始。

服务水平指标（SLI）

> 简单地说，SLI回答了“如何衡量客户是否满意”的问题。从用户的角度来看，答案代表了一个健康的系统。SLI可以是业务级别的指标，如“面向客户的API的响应时间”或是最基本的“服务已启动”。根据数据的上下文及其与产品的关系，你可能会发现需要不同的指标。

服务水平目标（SLO）

> SLO回答了“为了确保客户满意，能允许SLI达到的最低限度是多少”的问题。SLO是我们希望将特定的SLI视为健康服务的目标范围。如果SLI的指标是服务正常运行的时间，那么在给定的时间范围内，运行时间达到几个9就是SLO。SLO必须定义为给定时间范围内的一个具体值，以确保每个人都对SLO的含义保持一致的理解。SLI加上SLO构成了了解客户是否满意的基本方程式。

服务水平协议（SLA）

> SLA回答了“我同意的SLO会产生什么后果”的问题。SLA是与一个或多个业务客户（付费客户，而非内部利益相关者）签订的协议中包含的SLO，如果未满足该SLA，将受到财务或其他处罚。请务必注意，SLA是可选的。
>
> 本章不会详细介绍SLA，因为它们往往需要更多的业务讨论而不是工程讨论。这类决策主要取决于如果业务部门在合同中承诺SLA，他们希望获得什么样的销售业绩，以及如果SLA被破坏，是否值得冒收入损失的风险。希望通过在这里介绍的关于选择SLI和匹配SLO的内容，可以帮助你做出类似的决定。

定义这些SLI、SLO和SLA不仅可以引导业务健康状况，还可以指导工程团队的规划。如果团队没有达到其承诺的SLO，则不应继续进行新特性的工作。数据库工程团队也是如此。如果我们在本章中讨论的某个潜在的SLO没有得到满足，那么应该引发关于为什么不满足的讨论。当有了数据来解释为什么客户体验不是最理想的，便可以就团队的优先事项进行更有意义的讨论。

怎样才能让客户满意

选择一组指标作为SLI后，可能有人会倾向于将目标设置为100%，但你必须克制这种冲动。请记住，选择指标和目标的目的是随时使用客观指标评估团队是否能够利用新功能进行创新，或者稳定性是否有可能降至客户可接受的水平以下，因此需要更多的关注和资源。我们的目标是定义确保客户满意的最低标准。如果客户对于两秒内加载页面感到满意，则无须设置750毫秒内加载页面的目标，这会给工程团队带来不合理的负担。

以正常运行时间作为一个指标和目标值为例，我们可以宣称“将不会有任何停机时间”，但在实现和跟踪我们是否达到目标时，这意味着什么？达到3个9的可用性就是一个不小的壮举。一年中的3个9相当于8个多小时的停机时间，转换到一周则只有10分钟。你承诺的“9”越多，就越难实现，团队为实现这一承诺所花费的工程时间也就越昂贵。表2-1是来自Amazon Web Services的一个有用的图表，以纯数字显示了挑战。

表2-1：几个9的可用时间

可用性	一年内的宕机时间	一个月内的宕机时间	一周内的宕机时间	一天内的宕机时间
99.999%	5 分 15.36 秒	26.28 秒	6.06 秒	0.14 秒
99.995%	26 分 16.8 秒	2 分 11.4 秒	30.3 秒	4.32 秒
99.990%	52 分 33.6 秒	4 分 22.8 秒	1 分 0.66 秒	8.64 秒
99.950%	4 小时 22 分 48 秒	31 分 54 秒	5 分 3 秒	43 秒
99.900%	8 小时 45 分 36 秒	43 分 53 秒	10 分 6 秒	1 分 26 秒
99.500%	43 小时 48 分 36 秒	3 小时 39 分	50 分 24 秒	7 分 12 秒
99.250%	65 小时 42 分	5 小时 34 分 30 秒	1 小时 15 分 48 秒	10 分 48 秒
99.000%	3 天 15 小时 54 分	7 小时 18 分	1 小时 41 分 5 秒	14 分 24 秒

工程时间是有限的资源，所以在选择SLO时必须注意不要过于追求完美。不是产品中的所有特性都需要这么多个9才能让客户满意。你会发现，随着产品特性集的增长，将会有不同的SLI和SLO，具体取决于特定功能的影响或其带来的收入。这是意料之中的，也是一个深思熟虑的过程的标志。你还有一项关键任务：检测数据集何时成为不同用户的不同查询概要文件（query profile）的瓶颈，从而影响性能。这也意味着要找到一种方法来区分不同用户的需求，以便可以为他们提供合理的SLI和SLO。

这些指标和目标也是在产品和工程之间达成一致标准的有效方法，以指导在“将工程时间花在新功能上”与“将时间花在可恢复性和修复问题上”之间做出决策。这也是一种从需要做的事情列表中决定什么是最重要事情的方法，关键是基于用户体验来判断。你可以使用SLI和SLO来指导那些难以协调的关于工作优先级的讨论。

用什么来度量

假设有一家公司的产品是在线商店。由于在线购物的发展，该公司未来会有更多的流量，因此需要基础设施组来确保数据库层能够处理增加的需求。在本节中，我们将讨论如果我们是虚构的基础设施团队，那么应该度量什么。

定义**SLI**和**SLO**

定义好的SLI和匹配的SLO，是简洁地解释如何为客户提供愉快的用户体验的核心。我们不会花太多时间抽象地解释如何创建有意义的SLI和SLO。[2]在MySQL的使用环境中，需要定义三大重要主题：可用性、延迟和关键错误缺失。

以上面的在线商店为例，这种场景下的页面加载速度很快，在一个月内至少有99.5%的时间是快于几百毫秒的。这也意味着一个可靠的检查过程，在一个指定的自然月内，间歇性故障只允许1%的时间出现。注意这些指标和目标是如何定义的。我们没有要求必须做到100%，因为我们在一个不可避免失败的世界中运作。我们定义了一个时间跨度，以便团队能够在新功能和可恢复性之间准确地平衡工作。

“我希望99.5%的数据库请求在不到两毫秒的时间内正常执行”是一个有明确SLO的充分SLI，但并不是一个简单的SLI。不应该用一个指标来确定所有要求。这只是一条简单的描述语句，表示你希望数据库层如何运行，以提供可接受的客户体验。

那么，在在线商店的例子中，建立这种客户体验监控图的指标有什么好例子呢？从模拟测试开始，例如，在生产环境中对页面的加载速率进行取样。这是一个很有用的信号，表明“一切正常”。但这仅仅是一个开始，接下来让我们讨论通过跟踪不同方面的信号来构建一张完整的监控图。当浏览这些示例时，把它与在线商店联系起来，以直观地了解这些不同的指标如何构成一张良好的客户体验监控图。首先，让我们讨论一下跟踪查询响应时间。

监控解决方案

在SLI和SLO的背景中，查询分析和查询延迟监控都需要关注客户体验。这意味着需要依赖一些工具，当查询响应时间超过商定的阈值时，能够第一时间发出警报。下面讨论几种实现这种监控级别的方法。

商业选项

这是其中一个例子。如果一个供应商的竞争优势是分析MySQL性能的特定任务，那么付费是可以给组织带来回报的。诸如SolarWinds数据库性能管理工具（参见链接6）可以大大提高剖析查询性能的自动化程度，并且让工程组织中的大部分人都能比较容易地理解。

开源选项

Percona监控和管理工具是一个成熟的开源选项（参见链接7），称为PMM。它以客户端/服务器的方式运行。在数据库实例上安装一个客户端，该客户端收集指标并将其发送到服务器端。服务器端还有一组仪表盘，可以查看与性能相关的图表。PMM的一大好处是，仪表盘的组织是由Percona社区在监控MySQL性能方面的长期经验指导的。这让它成为一

个很好的资源，可以让刚接触MySQL的工程师熟悉如何监控MySQL性能。另一种方法是将数据库慢查询日志和MySQL Performance Schema的输出信息发送到一个集中的位置，然后可以使用像*pt-query-digest*（Percona Toolkit包中的工具）这样的成熟工具来分析日志，并更深入地了解数据库实例在哪些方面花费了时间。这种方法虽然有效，但此过程可能会很慢，如果使用不当，可能会影响客户。理想情况下，还是希望在客户注意到问题之前发现问题。在问题发生后再被动地检查日志，将面临客户信任下降的风险，因为发现性能倒退然后挖掘各种事后现场以确定发生了什么可能会需要很长的时间。

最后，使用Performance Schema来分析MySQL的性能对我们会有很大帮助，在第3章你将了解到关于Performance Schema的更多细节。通过Performance Schema可以发现性能瓶颈，从而让数据库实例能在相同规格下承载更多业务压力，并节省基础设施成本，或者回答诸如“为什么要花这么长时间”这样的问题。Performance Schema并不是一个仅用于确定是否满足服务可用性承诺的工具，因为它涉及MySQL内部深层的机制。为了做好服务水平性能评估，我们现在需要一种思考性能的新方法。

关于“在生产中测试”的说明

经常会听到“生产中测试”的鼓声，这让很多人畏缩。实际情况是，在生产中进行测试具有很大的价值。在生产中，你可以发现这种变化是如何影响系统的其他部分、规模和实际客户流量的。也可以查看对相邻系统的影响。

通过使用基本的“客户是否满意”的问题，可以看到：

- 当来自生产的反馈循环很快，并且和变更紧密相关时，回滚变更并重新检查正在部署的特定变更的速度也会快得多。
- 这种方法促进了功能团队和数据库工程师之间更强大的协作。当所有相关方都在观察特定的具体指标和它们应该是什么值时，测量性能的任务就变成了一项团队工作。
- 在性能倒退的情况下，花费生产之外的精力去调查“发生了什么”远比试图重新创建一个模拟更大代码路径的基准测试套件要具体得多。用于调试的工程时间变得更有针对性。

现在，让我们深入了解其他指标，以进一步理解在线商店的客户体验。你应该考虑从MySQL中获得的结果，而不是输出。我们还将介绍一些无法仅通过MySQL指标来测量的情况。

监控可用性

间歇性不可用的在线商店会降低购物者的信任。这就是为什么将可用性作为一个独立的指标，以及作为客户体验中如此重要的一部分。

可用性是指能够无错误地响应客户的请求。要用标准HTTP术语来描述这一点，要么是一个明确成功的响应，如200响应代码，要么是一个成功接受请求并承诺异步完成相关工作的响应，如202已接受。在单机系统的时代，可用性曾经是一个简单的指标。如今，大多数架构都十分复杂，可用性的概念已经演变为分布式系统如何失效的更细微的反映。当试图将可用性转换为数据库架构的SLI和SLO时，请考虑进一步的细节（结合在线商店的示

例），例如：

- 在处理不可避免的灾难性故障时，哪些功能是不可协商的，哪些功能是“最好拥有的”（例如，客户是否可以继续使用现有购物车并下单，只是在此故障期间不能添加新商品）？
- 将哪些类型的失败定义为“灾难性的”（例如，列表搜索失败可能不是灾难性的，但下单操作失败将是灾难性的）？
- “降级功能”是什么样子的（例如，是否可以在需要时加载通用推荐，而不是基于过去的购买历史加载定制化推荐）？
- 给定一组可能的故障场景（例如，如果支持购物车下单系统的数据库写入失败，可以以多快的速度安全地转换到新的主节点），可以为核心功能承诺的最短平均恢复时间（MTTR）是多少？

当选择一组指标来表示可用性时，如果你想要与客户支持团队一起设定“100%正常运行时间”的期望，这是不合理的，这里的重点是，在理解并接受组件故障不可避免的情况下，尽可能地提供最好的客户体验。

验证可用性的首选方法是从客户端或远程端点来进行访问。如果可以访问客户端记录的数据库访问日志，则可以被动地执行此操作。明确地说，这意味着如果是PHP应用程序，并且在Apache下运行，那么需要访问Apache日志，以确定PHP在连接到数据库时是否发出任何错误。你也可以主动验证可用性。如果环境是隔离的，无法访问客户端日志，那么可以考虑设置远程代码并在数据库执行操作，以确保数据库可用。这可能很简单，比如SELECT 1这样的查询，可以验证MySQL是否正在接收和解析查询，但不需要访问存储层。也可能更复杂一点，比如从表中读取实际数据，或者执行写操作然后再读取以验证写操作是否成功。这种来自异地网络访问的模拟事务可以让你了解应用程序是否可用。

可用性的远程验证对于跟踪可用性目标非常有用，但它不能让你在问题出现之前就发现。MySQL中有一个Threads_running状态计数器可以作为可用性问题的关键指标，这个计数器跟踪的是给定数据库主机上当前正在运行的查询数量。当运行的线程快速增长且没有任何下降迹象时，说明查询完成得不够快，因此正在堆积和消耗资源。如果允许这个指标增长，通常会导致数据库主机出现完全的CPU锁定或严重的内存负载，从而导致操作系统关闭整个MySQL进程。如果这种情况发生在主节点上，显然会导致重大的中断，所以应该将其视为关键指标。要监控这一点，首先要检查有多少个CPU核，如果Threads_running超过了CPU核数，则可能表明服务器正处于不稳定状态。与此相结合，你还可以监控Thread_running与max_connections的差距，将此差距作为另一个数据点，以检查正在进行的工作是否过载。

在第5章的“安全设置”一节中，我们将深入了解如何遏制失控的MySQL线程。

监控查询延迟

MySQL引入了许多长期需要的增强功能来跟踪查询运行所需的时间（参见链接8），当更改应用程序代码时，你一定要使用监控堆栈来跟踪这些趋势。然而，这仍然不是客户体验的全貌，特别是考虑到现代软件架构的设计方式。除了内部跟踪的延迟，你还需要了解应用程序如何感知延迟，以及当感知到的延迟增加时会发生什么。这意味着，除了直接从数

据库服务器跟踪查询延迟，还可以通过客户端工具来及时报告查询完成情况，从而尽可能提升客户体验。也可以使用诸如DataDog或SolarWinds数据库性能监控之类的付费工具，或者使用诸如PMM之类的开源工具，从客户端（尤其是当基础设施不断扩张时）收集所有的样本指标。在这里，与组织的应用程序开发人员密切协作至关重要。你需要了解应用程序团队如何从应用程序的角度来衡量这一点，并使用诸如Honeycomb或Lightstep之类的跟踪工具增加对异常值的关注。

监控报错

每次报错是否都需要跟踪和提醒？这要视情况而定。

MySQL客户端在访问运行着的服务的过程中出现错误并不一定意味着服务遭到破坏。在分布式系统的世界中，在很多情况下，客户端可能会遇到间歇性错误，通过简单地重试失败的查询就可以解决这些错误。不过，在基础架构中，对于处理数据库查询的服务而言，错误发生的频率可能是潜在问题的关键指标。以下是一些客户端错误的示例，这些错误通常可能只是噪声，但如果报错的频率加快，则是将要出现问题的迹象。

Lock wait timeout

> 如果客户端中该报错急剧增加，可能是主节点上的行级锁争用在不断扩大，即事务不断重试但仍然失败。这可能是无法写入的前兆。

Aborted connections

> 如果客户端中该报错突然激增，可能表明客户端和数据库实例之间的某个访问层出现了问题。不跟踪这一点会导致大量客户端重试，这会消耗资源。

MySQL服务器跟踪的名为Connection_errors_×××的服务器变量集（参见链接9）非常有用，其中×××是不同类型的连接错误。这些计数器的突然增加可能是一个强有力的指示器，告诉你出现了一些新的问题。

是否存在只要在单个实例上出现即意味着麻烦并需要处理的错误呢？答案是肯定的。

举个例子，如果MySQL实例运行在只读模式，即使这种错误不是经常发生，也是出现问题的标志。这可能意味着你已经将一个副本提升为主节点，但仍然运行在只读模式下（副本通常运行在只读模式，不是吗？），对于集群的写操作而言，这就是宕机了。或者，这可能意味着访问层出现了问题，从而将写操作发送到了副本节点。在上述任何一种情况下，这都不是通过重试就能解决的间歇性问题的标志。

另一个作为重大问题标志的服务器端错误是“too many connections”或操作系统级别的“cannot create new thread”。这些迹象表明，应用程序创建和打开的连接数超过了数据库服务器配置中允许的连接数，这个限制可能来自服务器的max_connections变量或者MySQL进程被允许打开的线程数。这些错误会立即转化为5××错误返回给应用程序，这会对客户产生影响，具体取决于应用程序的设计。

如你所见，衡量性能和选择哪些错误来构建SLI既是一个技术问题，也是一个沟通和社交问题，你应该为此做好准备。

主动监控

正如我们所说，SLO监控的重点是客户是否满意。这有助于你在客户不满意的时候专注于改善他们的体验，在客户满意的时候可以专注于其他任务，比如减少工作量。但这忽略了一个关键点：主动监控。

回到在线商店的例子，设想一下我们要如何监控客户的体验。假设没有遇到任何组件的重大故障，但注意到反馈“缓慢”或偶尔出现错误的客户支持工单正在上升。你如何跟踪这种行为？如果不知道许多信号的基线性能是什么，那么这可能是一项非常困难的任务。用于触发随叫随到（on-call）告警的仪表盘和脚本被称为稳态监控。无论是否有变更，这些告警会让你知道某个系统发生了意外情况，也是在客户体验失败之前提供前导指标的重要工具。

监控方面需要达到的平衡是，它既要具有可操作性，又要成为真正的前导指标。数据库磁盘100%满时的空间告警已经太迟了，因为服务已经停止。但如果磁盘空间占用的增长速度没那么快，设置磁盘空间超过80%就发出告警的话，可能还不需要急着采取行动，告警可能会被忽略。

让我们来讨论一下可以监控的有用信号，这些信号与实际的客户影响没有直接关系。

磁盘空间使用率增长

在出现问题之前，跟踪磁盘空间使用率的增长可能是一类不太会被考虑的指标。而这个问题真的出现时，解决起来会耗费时间并影响业务。最好还是了解如何跟踪该指标，制订缓解计划，并知道告警阈值设置成多少是合适的。

有许多策略可用于监控磁盘空间使用率的增长。让我们从最理想到最简单的方式来拆解一下。

如果监控工具允许，那么跟踪磁盘空间使用率的增长速度将非常有用。总会有这样的场景，可用磁盘空间可能被快速耗尽，从而使可用性面临风险。诸如伴随大量Undo日志的长时间运行的事务或alter table之类的操作是导致磁盘空间快速耗尽的原因。有很多这样的事故，在数据库耗尽磁盘空间之前，过度的日志记录或者对某些数据集的插入模式的变化这样的情况都没有被检测到。直到磁盘耗尽之后，各种告警才发出来。

如果跟踪增长率不可行（并非所有监控工具都提供此功能），则可以设置多个阈值，其中较低的警告可以设置为仅在工作时间触发，而较高的、更严重的值则作为对非工作时间随叫随到的告警。这使得团队可以在工作时间提前处理告警，以免事情变得糟糕到半夜把人惊醒。

如果既不能监控增长率，也不能为同一指标定义多个阈值，那么至少需要确定用于呼叫随叫随到（on-call）工程师的单个磁盘空间阈值。这个阈值需要足够低，以便在团队评估触发的原因并考虑长期缓解措施时，有时间执行释放磁盘空间和其他一些操作。可以考虑计算磁盘可以写入的最大吞吐量（M B/s），并用它来计算在最大吞吐量下填满磁盘需要多长时间，你需要这么长的准备时间来避免发生事故。

我们将在第4章讨论操作系统和硬件配置，会涉及MySQL如何使用磁盘空间以及在与磁盘空间增长相关的决策中应考虑哪些取舍。可以预料的是，在某个时候，业务可能会增长到无法将所有数据存储在一个服务器集群中。即使运行在可以扩展卷的云环境中，仍然需要对此进行规划，设置一个可用磁盘空间的阈值，以便有足够的时间来计划和执行所需的扩展，而不会感到恐慌。

这里的要点是确保有一些对于磁盘空间增长的监控，即使你认为现在还为时过早，不需要它。这是几乎让所有人措手不及的增长率之一。

连接数增长

随着业务的增长，普遍现象是应用程序层的线性增长。你将需要更多的应用实例来支持登录、购物车、处理请求或产品的任何功能。添加的这些实例开始打开越来越多到数据库主机的连接。你可以通过添加副本、使用复制作为扩展措施，甚至使用ProxySQL之类的中间件层，将前端的增长与数据库上的连接负载直接解耦，从而在一段时间内缓解这种增长。

当流量不断增长时，数据库服务器可以支持有限的连接池，这被配置为服务器参数max_connections。一旦与服务器的连接总数达到最大值，数据库将不允许任何新的连接，这是导致无法与数据库建立新连接的常见原因，从而会导致用户感知的错误增加。

监控连接数增长是为了确保资源不会耗尽到危及数据库可用性的程度。这种风险可能以两种不同的方式出现：

- 应用程序层打开了大量未使用的连接，导致产生了毫无理由的连接过多的风险。一个明显的迹象是连接的线程数（threads_connected）很高，但运行的线程数（threads_running）仍然很低。
- 应用程序层正在积极地使用大量的连接，并有导致数据库过载的风险。可以通过查看连接的线程数（threads_connected）和运行的线程数（threads_running）都处于高值（成百上千？）并持续增加来识别这种状态。

在设置连接数监控时，要考虑的一个有用的技巧是依赖百分比而不是绝对值。threads_connected/max_connections的比值显示了应用程序节点数量的增长与数据库允许的最大连接池有多接近。这有助于监控连接数增长问题的第一个状态。

另外，还需要对数据库主机的繁忙程度进行跟踪并设置告警，正如前面解释的，这可以从threads_running的值中看出。通常，如果这个值增长到100以上，就会开始看到CPU使用率和内存使用率的增加，这是数据库主机上高负载的普遍迹象。对于数据库可用性来说，这是需要立即关注的问题，因为它可能导致MySQL进程被操作系统杀死。一个常见的快速解决方案是使用杀死进程命令或自动化使用该命令的工具，如*pt kill*，从战术上减轻负载，然后使用查询分析（前面已经描述过）研究数据库为什么会进入这种状态。

连接风暴是指在生产系统中，应用程序层感知到查询延迟增加，并通过打开更多到数据库层的连接进行响应的情况。这可能会导致数据库负载显著增加，因为它要处理大量涌入的新连接，这会占用执行查询请求的资源。连接风暴可能导致max_connections中的可用连接突然减少，并增加数据库可用性的风险。

复制延迟

MySQL有原生的复制功能，可以将数据从源服务器发送到一个或多个副本服务器。数据从写入源服务器到在副本服务器上可读取之间的延迟称为复制延迟。如果应用程序从副本读取数据，当你向副本发送读取命令，而副本还没有赶上所有更改时，复制延迟可能会使

数据看起来不一致。在社交媒体示例中，用户可能会对其他人发布的内容发表评论。此数据写入源节点，然后被复制到所有副本节点。当用户试图查看其回复时，如果应用程序将请求发送到延迟的副本服务器，可能读取不到数据。这可能会让用户感到困惑，认为他们的评论没有被保存。我们将在第9章更详细地介绍对抗复制延迟的策略。

复制延迟能够被视为一种重要的SLI指标，它能引起异常事故。同时它也是一个表示架构需要做出调整的长期趋势。长期来看，即使复制延迟从未达到影响客户体验的程度，如果偶然出现，这仍然是一个比较明显的迹象，说明当前配置下源节点写入设备的性能要强于副本节点，这可能预示系统写容量出现不足。如果你听从我们在这里的建议，那么能够全面预防未来可能发生的事故。

设置复制延迟告警需要谨慎。对于复制延迟，可立即采取行动的补救措施并不总是可行的。另外，假如你没有从副本读取数据，考虑一下监控系统将复制延迟告警发送给某人是否会过于激进。告警，尤其是非工作时间的告警，对于接收人来说应该是需要立即采取行动的。

复制延迟是一种可能会影响到决策的指标，无论这些决策是即刻决定的还是经过深思熟虑的。并且，如果长期关注其发展趋势，会帮助我们省去不少会影响到大型业务的麻烦，并能在曲线增长前发现问题。

I/O使用率

数据库工程师不断努力的目标之一是“尽可能多地在内存中工作，因为这样更快”。虽然这么说肯定是正确的，但我们也知道，不可能100%地做到这一点，因为这意味着数据完全可以被存储在内存中，在这种情况下，“规模”还不是我们需要花费精力的事情。

随着数据库基础设施的扩展，数据再也无法完全放入内存中，你会逐渐意识到，下一个最好的方法是不要从磁盘读取太多的数据，否则查询就只能等待那些宝贵的I/O周期。即使在这个几乎所有数据都在固态硬盘上运行的时代，这一点仍然是正确的。随着数据量的增长，查询需要扫描更多的数据来满足请求，你会发现I/O等待可能成为流量增长的瓶颈。

监控磁盘I/O活动有助于在性能下降成为客户面临的问题之前解决问题。为了实现这一目标，可以监控一些事情。像*iostat*这样的工具可以监控I/O等待。如果数据库服务器有很多线程位于IOwait状态，则需要监控发出告警，这表示它们正在队列中等待某些磁盘资源可用。可以通过将IOutil作为一个运行图跟踪一段有意义的时间（例如一天、两天甚至一周）来发现这一点。IOutil报告的是占整个系统磁盘访问容量的百分比。如果IOutil在未运行备份的主机上持续接近100%，则表明存在全表扫描和低效查询。还需要以百分比形式监控磁盘I/O容量的总体使用率，因为这可以预先警告磁盘访问将会成为数据库性能的瓶颈。

自增键空间

在使用MySQL时，一个不太为人所知的“地雷”是，自动递增主键在默认情况下被创建为有符号整数，并且可能会耗尽键空间。当插入了足够多的数据行，使自动递增键达到其数据类型的最大可能值时，就会发生这种情况。在规划需要长期监控的指标时，为所有使用

自增主键的表监控剩余整数空间是一个简单的操作，但几乎可以肯定的一点是，它会在将来为你避免一些重大事故，因为可以提前预测需要更大的键空间。

如何监控自增键空间？有几个选择。如果使用了PMM及其Prometheus导出器（exporter），那么已经自带监控方法，你需要做的就是开启collect.auto_increment.columns设置。如果没有使用Prometheus，也可以使用下面的查询，它可以作为指标生成器或告警来进行修改，以告诉你的表何时会接近可能的最大键空间（参见链接10）。这个查询依赖于information_schema，其中包含有关数据库实例中表的所有元数据：

```
SELECT
    t.TABLE_SCHEMA AS `schema`,
    t.TABLE_NAME AS `table`,
    t.AUTO_INCREMENT AS `auto_increment`,
    c.DATA_TYPE AS `pk_type`,
    (
        t.AUTO_INCREMENT /
        (CASE DATA_TYPE
            WHEN 'tinyint'
                THEN IF(COLUMN_TYPE LIKE '%unsigned',
                    255,
                    127
                )
            WHEN 'smallint'
                THEN IF(COLUMN_TYPE LIKE '%unsigned',
                    65535,
                    32767
                )
            WHEN 'mediumint'
                THEN IF(COLUMN_TYPE LIKE '%unsigned',
                    16777215,
                    8388607
                )
            WHEN 'int'
                THEN IF(COLUMN_TYPE LIKE '%unsigned',
                    4294967295,
```

```
                    2147483647
                )
            WHEN 'bigint'
                THEN IF(COLUMN_TYPE LIKE '%unsigned',
                    18446744073709551615,
                    9223372036854775807
                )
        END / 100)
    ) AS `max_value`
    FROM information_schema.TABLES t
    INNER JOIN information_schema.COLUMNS c
        ON t.TABLE_SCHEMA = c.TABLE_SCHEMA
        AND t.TABLE_NAME = c.TABLE_NAME
    WHERE
        t.AUTO_INCREMENT IS NOT NULL
        AND c.COLUMN_KEY = 'PRI'
        AND c.DATA_TYPE LIKE '%int'
;
```

一般来说，在选择主键和管理自动增量时，必须考虑很多细微差别和事件背景，我们将在第6章讨论这一点。

创建备份/恢复时间

长期规划不仅要在业务正常运行时实现增长，还要在可接受的时间范围内实现灾难恢复。我们将在第10章讨论如何更深入地思考灾难恢复，以及在第13章讨论如何将其作为法规合规性控制职责的一部分，在这里提出这一点是为了指出，一个好的灾难恢复计划只有在重新审视并调整其目标时才起作用。

> 功能分片与水平分片
>
> 在本章和本书的其他章节中，你将看到我们将分片或分区作为在单独的实例上分割数据的不同方法，以实现可伸缩性。我们想要定义它们的含义以及它们的不同之处，以避免你在阅读本书的其余部分时产生混淆。
>
> 功能分片（Functional sharding）是指将服务于特定业务功能的特定表分割到一个专用的集群中，以便单独管理该数据集的正常运行时间、性能甚至访问控制。
>
> 水平分片（Horizontal sharding）是指当数据集的大小超过了可以在单个集群中可靠地提供服务的规模时，将它拆分为多个集群，并从多个节点提供数据，这依赖于某种查找机制来定位所需的数据子集。

如果你的数据库达到一定规模，从备份恢复所需的时间将超过恢复业务关键功能所需的时间，那么即使其他一切运行正常，也需要检查调整的MTTR目标，更改“关键功能”的定义，或找到缩短备份还原时间的方法。在规划灾难恢复时，需要考虑以下几点：

- 要非常具体地说明哪些功能属于这个恢复目标，如果需要的话，要查看支持该功能子集的数据是否需要在一个单独的集群中，以实际实现该期望。
- 如果无法按功能将数据划分为多个更小的实例，那么整个数据集都要处于通过备份

进行恢复的目标之下。从备份中恢复所需时间最长的数据集将是此恢复过程完成时间的决定因素。

- 确保有自动化的测试方法（我们将在第10章中介绍一些例子）。监控将备份从文件恢复到运行的数据库（该数据库自创建备份以来还一直在复制所有更改）需要多长时间，并将该指标存储在具有足够的保留能力的地方，以查看长期（至少一年）趋势。如果没有自动化的监控，这是一个可能会被忽略的指标，并且时间会变得非常长。

可以看到，在我们简短描述的许多长期指标示例中，几乎总是指出需要对数据进行功能分片或水平分片。这里的目标是明确指出这样一个事实：如果你已经遇到了主要由容量问题导致的事故，这时再考虑分片，可能已经太晚了。将数据分解为可管理的小块，这项工作不是在数据对于一个集群来说太大的时候才开始的，而是在此之前，当你还在为提供成功的客户体验确定目标时开始的。

了解恢复数据所需的时间有助于设定在实际灾难中应该做什么。它还可以让你意识到，什么时候花费的时间可能比业务希望的要长。这是需要分片的前兆。

度量长期性能

为日常操作选择SLI和SLO只是一个开始。你需要确保不会把整片森林错当成树，并将重点放在特定的主机指标上，而不是检查总体系统性能和客户体验结果。在这一节，我们将介绍可以用来思考系统整体长期健康状况的策略。

了解业务节奏

了解业务的流量节奏非常重要，因为这将让你有足够的时间，来确保所有SLO都是经过最严格测试的，并且受到最重要客户最严格的审查。业务节奏可能意味着峰值流量时间比“平均值”大几个数量级，如果数据库基础架构没有准备好，将产生很多不良结果。在数据库基础架构的环境中，这可以转化为每秒要完成的请求数量级的增加、来自应用程序服务器的连接负载的增加，或者如果写入操作出现间歇性故障，则会对收入产生更大的影响。以下是一些业务节奏的例子，有助于你了解贵公司的经营周期。

电子商务网站

: 对许多国家来说，11月下旬到年底是最忙的时候，从在网上商店中可以看到呈数量级的销量增长。这意味着更多的购物车订单，更多的并发售卖，以及比一年中任何其他时间相同故障对收入的影响更大。

人力资源软件

: 在美国，11月通常是许多员工进行福利选举的时候，这段时间被称为“开放登记”，这将产生更多的流量。

网上鲜花供应商

: 对网上的鲜花供应商而言，情人节是一年中最忙的时候，因为会有更多的人订购鲜花。

如上所见，这些业务周期可能会因业务所满足的客户需求而大不相同。了解业务周期以及对业务收入、声誉的影响至关重要，因此，在不影响运行系统的稳定性的情况下，你应该为满足需求做好足够准备。

当度量支撑业务的数据库基础设施的性能时，重要的是，不要脱离工程组织正在跟踪的其他重要指标。

数据库性能应该被视为关于技术栈性能讨论话题中的一部分，而不是作为特例来处理。首先，应该尽可能地使用与工程组织中的其他成员相同的工具。用于确定数据库执行情况所依赖的指标和仪表盘应与应用程序的指标使用相同方式访问，或者干脆使用同一仪表盘。无论使用何种技术或供应商，这个观点都将有助于创建一个人人都为整个堆栈性能而努力的环境，并减少众所周知的隔阂（工程师可能在开发的业务功能与支撑这些功能的数据库之间感受到这种隔阂）。

有效地跟踪指标

当涉及业务的长期规划时，有很多事情需要关注，包括但不限于：

- 为未来的容量做规划。
- 预见何时需要重大改进，何时增量修改就够了。
- 为运行基础架构增加的成本做规划。

不仅需要能够在某个时间点度量数据存储基础架构的运行状况，还需要能够长期预测性能的改善或下降趋势。这意味着不仅要识别SLI和SLO，还要找出哪些SLI和SLO仍然是有价值的、严重等级较高的长期趋势指标。你可能会发现，并非所有可用于短期随叫随到（on-call）决策的指标也适用于长期业务规划。

在深入探讨哪些指标对长期规划很重要之前，先来讨论一些能够实现长期趋势监控的工具。

使用监控工具检查性能

度量性能在即时感知“我们目前是否处于事故中”以及长期跟踪和趋势感知上都很重要。持有你所关心的指标的测量工具与指标本身一样重要。如果无法以与组织其他指标相关的方式正确地看到其随时间变化的趋势，那么选择一个好的SLI又有什么用？

监控工具正在迅速发展，对于如何发展及发展方向有很多强烈的观点。这里的目标是提高透明度，重点是跟踪结果而不是输出。在使基础架构堆栈成功的领域中，跟踪成功是一项团队运动。

在这里，我们不讨论具体的工具，而是列出一些重要的特性和方面，以便在考虑工具是否适合这种长期趋势时加以考虑。

对平均值说不

无论是作为工程组织自行管理度量解决方案，还是使用软件即服务（SaaS），都要注意度量解决方案如何规范数据以进行长期存储。许多解决方案在默认情况下将长期数据整合为平均值（Graphite是第一个这样做的方案之一），这是一个大问题。如果要查看一个指标在几周以上的时间内的趋势，平均值将使峰值平缓下来，这意味着如果你想查看磁盘I/O使用率是否会在下一年翻一番，那么平均值的数据点图很可能会让你产生错误的安全感。在对月份数据进行趋势分析时，请始终关注峰值，这样能够在视图中保持偶尔尖刺的逼真度。

与百分位为友

百分位依赖于在给定的时间范围内对数据点进行排序，并根据目标百分位移除最高值的数据点（例如，如果要寻找95百分位，则移除最顶端的5%）。这是一种极好的方法，可以使你查看的数据在视觉上更接近于查看SLI和SLO的方式。如果可以让显示查询响应时间的图表显示95百分位，你可以更轻松地将其与希望实现的应用程序请求完成SLO进行匹配，并使数据库指标对客户支持团队和工程师等人员（而不仅仅是数据库工程团队）更有意义。

长保留期和性能

这似乎是显而易见的，但是当试图显示长时间跨度时，监控工具的性能非常重要。如果你正在评估业务指标趋势的解决方案，则需要确保在要求越来越长的数据时间跨度时，测试用户体验如何变化。衡量指标的解决方案需要尽可能地保证数据的可用性，而不仅仅是获

取数据的速度或者保存时间。

既然我们已经描述了长期监控工具应该是什么样的，那么让我们来讨论一下到目前为止在选择SLI和SLO时所涉及的内容是如何指导数据架构的。

使用**SLO**来指导整体架构

在业务增长的同时保持良好一致的客户体验不是一件容易的事。随着业务规模的增长，保持相同的SLO都将变得越来越困难，更不用说制定更雄心勃勃的SLO了。以可用性为例：每个人都希望其所有数据的读写正常运行时间有尽可能多的9。但是，你希望实现的SLO越严格，工作成本就越高，因为每秒的数据库事务数峰值或数据量也会呈数量级的方式增长。

使用我们已经讨论过的SLI和SLO，你可以找到需要开始将数据进行分片或分区的增长点。我们将在第11章更详细地讨论使用分片来扩展MySQL，但在这里要指出的重要一点是，告诉你系统目前运行情况的SLI和SLO也可以引导你知道何时该投入成本扩展MySQL，以便各个集群在SLO的范围内保持可管理性，从而保留客户的体验。

拥有一个能够处理短期和长期指标并能以有用的方式预测变化趋势的指标解决方案，是跟踪战术性能指标以及影响数据库基础架构运行的长期业务趋势的一个非常重要的部分。

小结

在将可靠性工程概念应用于监控数据库基础架构的过程中，不断改进并重新审视指标和目标非常重要。在第一次定义一些SLI和SLO之后，并不意味着一成不变。随着业务的发展，你将对客户体验有更深入的了解，这将推动SLI和SLO的改进。

在选择指标并为其分配目标时，要意识到需要始终专注于表达客户体验。此外，不要将所有精力都集中在显示事故发生的指标上，而是应花一些时间来监控可以帮助预防事故的事情。这都是为保护客户体验的主动行为。

建议在三个关键方面预先设定目标：延迟、可用性和错误。这三个方面可以提供一个很好的信号，表明客户是否满意。除此之外，还要确保在连接数增长、磁盘空间使用率增长、磁盘I/O使用率和延迟方面进行主动监控。

希望本章能够帮助你了解如何在公司规模不断扩大的情况下，成功地将可靠性工程应用于监控MySQL。

---

[1] **Nicole Forsgren**，***Accelerate*：*The Science of Lean Software and DevOps***（**IT Revolution Press**，**2018**）.

[2] 强烈推荐**Alex Hidalgo**编著的***Implementing Service Level Objectives***一书（**O'reilly**出版）。

第3章Performance Schema

本章由**Sveta Smirnova**撰稿

在高负载下调优数据库性能是一个迭代循环的过程。每次进行更改以调优数据库的性能时，都需要了解更改是否有什么影响。查询速度比以前快吗？锁是否会减慢应用程序的速度，或者是否已经完全消失了？内存使用情况改变了吗？等待磁盘的时间改变了吗？一旦理解了如何回答这些问题，你将能够更快地评估和应对日常情况，并更有信心。

Performance Schema是一个存储回答上述问题所需数据的数据库。本章将帮助你了解Performance Schema的工作原理、局限性，以及如何更好地使用它和sys Schema来搞清楚MySQL内部的运行细节。

Performance Schema介绍

Performance Schema提供了有关MySQL服务器内部运行的操作上的底层指标。为了解释清楚Performance Schema的工作机制，先介绍两个概念。

第一个概念是程序插桩（instrument）。程序插桩在MySQL代码中插入探测代码，以获取我们想了解的信息。例如，如果想收集关于元数据锁的使用情况，需要启用wait/lock/meta-data/sql/mdl这个插桩。

第二个概念是消费者表（consumer），指的是存储关于程序插桩代码信息的表。如果我们为查询模块添加插桩，相应的消费者表将记录诸如执行总数、未使用索引的次数、花费的时间等信息。大多数人都将消费者表与Performance Schema紧密联系在一起。

Performance Schema的一般功能如图3-1所示。

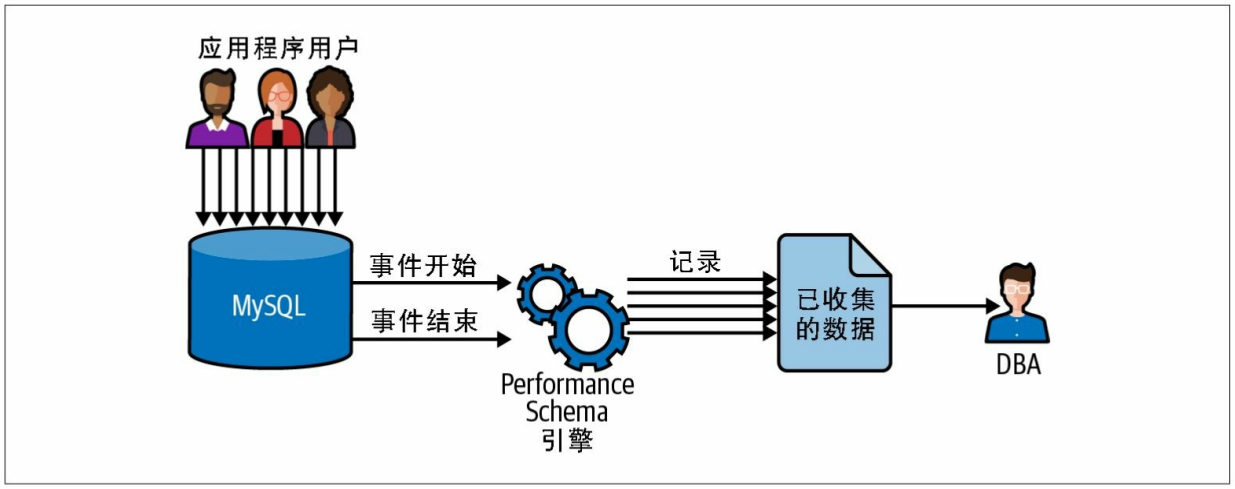

图3-1：在数据库上运行的查询流，显示Performance schema如何收集和聚合数据，然后呈现给DBA

当应用程序用户连接到MySQL并执行被测量的插桩指令时，performance_schema将每个检查的调用封装到两个宏中，然后将结果记录在相应的消费者表中。这里的要点是，启用插桩会调用额外的代码，这意味着插桩会消耗CPU资源。

插桩元件

在performance_schema中，setup_instruments表包含所有支持的插桩的列表。所有插桩的名称都由用斜杠分隔的部件组成。下面的例子展示了插桩的命名规则：

- statement/sql/select
- wait/synch/mutex/innodb/autoinc_mutex

插桩名称的最左边部分表示插桩的类型。因此，statement表示插桩类型是statement，wait表示插桩类型是wait，以此类推。

名称字段中的其余部分从左至右依次表示从通用到特定的子系统。在前面的示例中，select是sql子系统的一部分，属于statement类型。或者autoinc_mutex属于innodb，它是更通用的插桩类mutex的一部分，而mutex又是更通用的插桩类型wait的sync插桩的一部分。

大多数插桩名称是自描述型的。与示例中一样，statement/sql/select是一个SELECT查询，wait/synch/mutex/innodb/autoinc_mutex是InnoDB在自增列上设置的一个互斥体。setup_instruments表中还有一个DOCUMENTATION列，其中包含更多详细信息：

```
mysql> SELECT * FROM performance_schema.setup_instruments
    -> WHERE DOCUMENTATION IS NOT NULL LIMIT 5, 5\G
*************************** 1. row ***************************
 NAME: statement/sql/error
 ENABLED: YES
 TIMED: YES
 PROPERTIES:
 VOLATILITY: 0
 DOCUMENTATION: Invalid SQL queries (syntax error).
*************************** 2. row ***************************
 NAME: statement/abstract/Query
 ENABLED: YES
 TIMED: YES
 PROPERTIES: mutable
 VOLATILITY: 0
 DOCUMENTATION: SQL query just received from the network. At this point, the
 real statement type is unknown, the type will be refined after SQL parsing.
*************************** 3. row ***************************
 NAME: statement/abstract/new_packet
 ENABLED: YES
 TIMED: YES
 PROPERTIES: mutable
 VOLATILITY: 0
 DOCUMENTATION: New packet just received from the network. At this point,
the real command type is unknown, the type will be refined after reading
the packet header.
*************************** 4. row ***************************
 NAME: statement/abstract/relay_log
 ENABLED: YES
 TIMED: YES
 PROPERTIES: mutable
 VOLATILITY: 0
 DOCUMENTATION: New event just read from the relay log. At this point, the
real statement type is unknown, the type will be refined after parsing the event.
*************************** 5. row ***************************
 NAME: memory/performance_schema/mutex_instances
 ENABLED: YES
 TIMED: NULL
 PROPERTIES: global_statistics
 VOLATILITY: 1
 DOCUMENTATION: Memory used for table performance_schema.mutex_instances
5 rows in set (0,00 sec)
```

不幸的是，对于许多插桩而言，DOCUMENTATION列可能为空，因此需要根据插桩的名称、你的直觉以及对MySQL源代码的认识来理解特定插桩检查的内容。

消费者表的组织

正如前面提到的，消费者表是插桩发送信息的目的地。测量结果存储在Performance Schema数据库的多个表中；事实上，MySQL 8.0.25社区版的performance_schema中包含110个表。基于它们的用途，可分为以下几个类别。

当前和历史数据

存放事件的表名包含如下结尾：

*_current

当前服务器上进行中的事件。

*_history

每个线程最近完成的10个事件。

*_history_long

从全局来看，每个线程最近完成的10000个事件。

*_history和*_history_long表的大小是可配置的。

以下是当前和历史数据：

events_waits

底层服务器等待，例如获取互斥对象。

events_statements

SQL查询语句。

events_stages

配置文件信息，例如创建临时表或发送数据。

events_transactions

事务。

汇总表和摘要

汇总表保存有关该表所建议的内容的聚合信息。例如，memory_summary_by_thread_by_event_name表保存了用户连接或任何后台线程的每个MySQL线程的聚合内存使用情况。

摘要是一种通过删除查询中的变量来聚合查询的方法。例如以下查询：

```
SELECT user,birthdate FROM users WHERE user_id=19;
SELECT user,birthdate FROM users WHERE user_id=13;
SELECT user,birthdate FROM users WHERE user_id=27;
```

该查询的摘要是：

```
SELECT user,birthdate FROM users WHERE user_id=?
```

这允许Performance Schema跟踪摘要的延迟等指标，而无须单独保留查询的每个变体。

实例表（**Instance**）

实例是指对象实例，用于MySQL安装程序。例如，file_instances表包含文件名和访问这些文件的线程数。

设置表（**Setup**）

设置表用于performance_schema的运行时设置。

其他表

还有一些表的名称没有遵循严格的模式。例如，metadata_locks表保存关于元数据锁的数据。在本章后面讨论performance_schema可以解决的问题时，我将介绍其中几个表。

资源消耗

Performance Schema收集的数据保存在内存中。可以通过设置消费者表的最大大小来限制其使用的内存量。performance_schema中的一些表支持自动伸缩，这意味着它们在启动时分配最小数量的内存，并根据需要调整其大小。然而，一旦分配了内存，即使禁用了特定的插桩并截断了表，也不会再释放该内存。

如前所述，每个插桩指令的调用都会再添加两个宏调用，以将数据存储在performance_schema中。这意味着插桩越多，CPU的使用率就越高。对CPU使用率的实际影响取决于特定的插桩。例如，与statement相关的插桩在查询过程中只能被调用一次，而wait类插桩的被调用频率要高得多。例如，要扫描一个有一百万行的InnoDB表，引擎需要设置和释放一百万行锁。如果对锁使用wait类插桩，CPU使用率可能会显著增加。但是，同一查询本来就需要一次调用来确定是否是statement/sql/select。因此，如果启用statement类插桩，你不会注意到CPU负载的任何增加。内存或元数据锁类型的插桩也是如此。

局限性

在讨论如何设置和使用performance_schema之前，了解其局限性非常重要。

它必须得到MySQL组件的支持。

> 例如，假设使用内存插桩来计算哪个MySQL组件或线程使用了大部分内存。你发现使用最多内存的组件是一个不支持内存插桩的存储引擎。在这种情况下，你将无法找到内存的去向。

它只在特定的插桩和用户启用后才收集数据。

> 例如，如果在禁用所有插桩的情况下启动服务器，然后决定检测内存使用情况，则无法知道全局缓冲区（如InnoDB缓冲池）分配的确切数量，因为在启用内存插桩之前已分配了该缓冲区。

它很难释放内存

> 可以在启动时限制消费者表的大小，也可以让其自动调整大小。在后一种情况下，它们不会在启动时分配内存，而是仅在收集启用的数据时分配内存。但是，即使以后禁用了特定的插桩或消费者表，也不会释放内存，除非重新启动服务器。

在本章的其余部分，我将默认你已经理解这些局限性，因此不会再特别强调它们。

sys Schema

自5.7版以来，标准MySQL发行版包括一个和performance_schema数据配套使用的sys schema，它全部基于performance_schema上的视图和存储例程组成。它的设计目的是让performance_schema体验更加流畅，它本身并不存储任何数据。

sys schema的使用非常方便，但需要记住，它只访问存储在performance_schema表中的数据。如果在sys schema中找不到你想看的数据，可尝试在performance_schema的基表中查找。

理解线程

MySQL服务端是多线程软件。它的每个组件都使用线程。可以是后台线程，例如，由主线程或存储引擎创建的，也可以是为用户连接创建的前台线程。每个线程至少有两个唯一标识符：一个是操作系统线程ID，另一个是MySQL内部线程ID。操作系统线程ID可以通过相关工具查看，如在Linux系统中可使用`ps-eLf`命令查看。而MySQL内部线程ID在大多数performance_schema表中以THREAD_ID命名。此外，每个前台线程都有一个指定的PROCESSLIST_ID：连接标识符，在SHOW PROCESSLIST命令输出中或在MySQL命令行客户端连接时在“Your MySQLconnection id is”字符串中可以看到。

THREAD_ID不等于PROCESSLIST_ID！

performance_schema中的threads表包含了服务器中存在的所有线程：

```
mysql> SELECT NAME, THREAD_ID, PROCESSLIST_ID, THREAD_OS_ID
    -> FROM performance_schema.threads;
+------------------------+-----------+----------------+--------------+
| NAME                   | THREAD_ID | PROCESSLIST_ID | THREAD_OS_ID |
+------------------------+-----------+----------------+--------------+
| thread/sql/main        |         1 |           NULL |       797580 |
| thread/innodb/io_ib... |         3 |           NULL |       797583 |
| thread/innodb/io_lo... |         4 |           NULL |       797584 |
...
| thread/sql/slave_io    |        42 |              5 |       797618 |
| thread/sql/slave_sql   |        43 |              6 |       797619 |
| thread/sql/event_sc... |        44 |              7 |       797620 |
| thread/sql/signal_h... |        45 |           NULL |       797621 |
| thread/mysqlx/accep... |        46 |           NULL |       797623 |
| thread/sql/one_conn... |     27823 |          27784 |       797695 |
| thread/sql/compress... |        48 |              9 |       797624 |
+------------------------+-----------+----------------+--------------+
44 rows in set (0.00 sec)
```

除了线程号信息外，threads表还包含与SHOW PROCESSLIST输出相同的数据和一些附加列，如RESOURCE_GROUP或PARENT_THREAD_ID。

Performance Schema到处使用THREAD_ID，而PROCESSLIST_ID只在threads表中可用。如果需要获取PROCESSLIST_ID，例如，要杀死持有锁的连接，则需要查询threads表来获取。

threads表可以连接到许多其他表，以提供有关正在运行的查询的附加信息（例如，查询数据、锁、互斥体或打开的表实例）。

在本章的其余部分，希望你熟悉这个表和THREAD_ID的含义。

配置

Performance Schema的部分设置只能在服务器启动时更改：比如启用或禁用Performance Schema本身以及与内存使用和数据收集的限制相关的变量。Performance Schema插桩和消费者表则可以被动态启用或禁用。

可以在禁用所有消费者表和插桩的情况下启动Performance Schema，所以建议只启用那些解决特定问题所需的插桩。这样就不会在Performance Schema上白白消耗额外的资源，也不会因为过度检测消耗资源而导致系统饿死。

启用或禁用**Performance Schema**

要启用或禁用Performance Schema，可以将变量performance_schema设置为ON或OFF。这是一个只读变量，要么在配置文件中更改，要么在MySQL服务器启动时通过命令行参数更改。

启用或禁用插桩

插桩也可以被启用或禁用。可以通过setup_instruments表查看插桩的状态：

```
mysql> SELECT * FROM performance_schema.setup_instruments
    -> WHERE NAME='statement/sql/select'\G
*************************** 1. row ***************************
 NAME: statement/sql/select
 ENABLED: NO
 TIMED: YES
 PROPERTIES:
 VOLATILITY: 0
 DOCUMENTATION: NULL
1 row in set (0.01 sec)
```

可以看到，在上面的查询结果中，ENABLED的值是NO，这说明目前没有针对SELECT语句执行模块启用插桩。

有三个方法可用于启用或禁用performance_schema插桩：

- 使用setup_instruments表。
- 调用sys schema中的ps_setup_enable_instrument存储过程。
- 使用performance-schema-instrument启动参数。

方法一：**UPDATE**语句

第一种方法是使用UPDATE语句更改instruments表的对应列值：

```
mysql> UPDATE performance_schema.setup_instruments
    -> SET ENABLED='YES' WHERE NAME='statement/sql/select';
Query OK, 1 rows affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

这是标准的SQL语句，所以也可以使用通配符来启用所有的SQL语句的插桩：

```
mysql> UPDATE performance_schema.setup_instruments
    -> SET ENABLED='YES' WHERE NAME LIKE statement/sql/%';
Query OK, 167 rows affected (0.00 sec)
Rows matched: 167 Changed: 167 Warnings: 0
```

使用这种方法设置的参数，在数据库重启后就会失效。

方法二：**sys**存储过程

sys schema提供了两个存储过程——ps_setup_enable_instrument和ps_setup_disable_instrument——用于启用和禁用其参数所对应的插桩。这两个存储过程都支持通配符。如果要启用或禁用所有受支持的插桩，请使用通配符“%”：

```
mysql> CALL sys.ps_setup_enable_instrument('statement/sql/select');
+----------------------+
| summary              |
+----------------------+
| Enabled 1 instrument |
+----------------------+
1 row in set (0.01 sec)
```

这个方法和前面的方法实际上完全一样，在重启之后设置也会失效。

方法三：启动选项

如前所述，两种方法都允许在线更改performance_schema配置，但是数据库重启之后配置就会失效。如果要在重启之后保留特定插桩的配置，需要使用performance-schema-instrument配置参数。

这个变量支持performance-schema-instrument='instrument_name=value'这样的语法。其中，instrument_name是插桩名称，值为ON、TRUE或1表示启用，值为OFF、FALSE或0表示禁用。对于那些计数统计而不是计时统计的插桩，可以多次指定此选项以启用或禁用不同的插桩。该选项还支持通配符：

performance-schema-instrument='statement/sql/select=ON'

如果指定了多个选项，则无论顺序如何，较长的插桩字符串优先于较短的插桩字符串。

启用或禁用消费者表

与插桩一样，消费者表也可以通过以下三种方式启用或禁用：

- 使用Performance Schema中的setup_consumers表。
- 调用sys schema中的ps_setup_enable_consumer或ps_setup_disable_consuper存储过程。
- 使用performance-schema-consumer启动参数。

一共有15张消费者表。其中一些名字已经不言自明，但也有一些消费者表的名字需要更多解释，如表3-1所示。

表3-1：消费者表及其目的

消费者表	描述
events_stages_[current|history|history_long]	剖析的细节，如"Creating tmp table"、"Statistics"或"buffer pool load"
events_statements_[current|history|history_long]	语句统计
events_transactions_[current|history|history_long]	事务
events_waits_[current|history|history_long]	等待
global_instrumentation	启用或禁用全局插桩。如果禁用，则不会再检查单个参数，不会维护全局或者每个线程的数据，也不会收集单个事件的数据
thread_instrumentation	线程级插桩。仅检查是否启用了全局插桩。如果禁用，则不会收集每个线程或单个事件的数据
statements_digest	语句摘要

上述例子中配置插桩的方法，也可用于配置消费者表。

优化特定对象的监控

Performance Schema可以针对特定对象类型、schema和对象名称启用或禁用监控。这在setup_objects表中完成。

对象类型（OBJECT_TYPE列）可以是下面的五个值之一：EVENT、FUNCTION、PROCEDURE、TABLE和TRIGGER。此外，还可以指定OBJECT_SCHEMA和OBJECT_NAME，并且支持通配符。

例如，要关闭test数据库中触发器的performance_schema信息采集，可以使用以下语句：

```
mysql> INSERT INTO performance_schema.setup_objects
    -> (OBJECT_TYPE, OBJECT_SCHEMA, OBJECT_NAME, ENABLED)
    -> VALUES ('TRIGGER', 'test', '%', 'NO');
```

如果要保留名为my_trigger的触发器的信息采集，可通过如下语句进行配置：

```
mysql> INSERT INTO performance_schema.setup_objects
    -> (OBJECT_TYPE, OBJECT_SCHEMA, OBJECT_NAME, ENABLED)
    -> VALUES ('TRIGGER', 'test', 'my_trigger', 'YES');
```

当performance_schema决定是否需要监测特定对象时，它首先搜索更具体的规则，然后再退回到一般的规则。例如，如果用户对一张表进行查询，并且触发了test.my_trigger，它将会检查触发器所触发的语句。但是，如果用户在表中运行的查询触发的是名为test.some_other_trigger的触发器，将不会检查该触发器。

这些对象没有配置文件的选项。如果需要在重新启动后持久化更改，那么需要将这些INSERT语句写入SQL文件中，并在启动时使用init_file选项加载该SQL文件。

优化线程的监控

setup_threads表包含可以监控的后台线程列表。ENABLED列指定是否启用了特定线程的监测。HISTORY列指定特定线程的检测事件是否也应存储在_history和_history_long表中。

例如，要禁用事件调度程序（thread/sql/event_scheduler）的历史日志记录，可以运行：

```
mysql> UPDATE performance_schema.setup_threads SET HISTORY='NO'
    -> WHERE NAME='thread/sql/event_scheduler';
```

用户线程的设置不在setup_threads表中，而是在setup_actors表中，它包含表3-2中描述的列。

表3-2：setup actors表包含的列

列名	描述
HOST	主机名，如 localhost、%、my.domain.com，或 199.27.145.65
USER	用户名，如 sveta 或 %
ROLE	未使用
ENABLED	线程是否启用
HISTORY	是否在 _history 和 _history_long 中保存数据

要为特定账户指定规则，请使用如下命令：

```
mysql> INSERT INTO performance_schema.setup_actors
    -> (HOST, USER, ENABLED, HISTORY)
    -> VALUES ('localhost', 'sveta', 'YES', 'NO'),
    -> ('example.com', 'sveta', 'YES', 'YES'),
    -> ('localhost', '%', 'NO', 'NO');
```

该语句启用了sveta@localhost和sveta@example.com的监测，禁用sveta@localhost的历史记录，并禁用从localhost连接的所有其他用户的监测和历史记录。

与对象监控一样，线程和actor都没有配置文件的选项。如果需要在重新启动后保留表中的更改，需要将这些INSERT语句写入SQL文件，并使用init_file选项在启动时加载该SQL文件。

调整**Performance Schema**的内存大小

Performance Schema将数据存储在使用PERFORMANCE_SCHEMA引擎的表中，该引擎将数据存储在内存中。默认情况下，某些performance_schema表会自动调整大小，其他的则有固定数量的行。可以通过更改启动变量来调整这些选项。变量的名称遵循performance_schema_object_[size|instances|classes|length|handles]的模式，其中对象要么是消费者表，要么是设置表，要么是特定事件的插桩实例。例如，配置变量performance_schema_events_stages_history_size定义了performance_schema_events_stages_history表将存储的每个线程的阶段数。变量performance_schema_max_memory_classes定义了可以使用的最大内存插桩数量。

默认值

MySQL不同部分的默认值会随着版本的不同而改变。因此，在参考这里描述的值之前，最好阅读官方的用户参考手册。但是，对于Performance Schema来说，它们还会影响服务器的整体性能，因此我想介绍其中一些重要的默认值。

从5.7版开始，Performance Schema在默认情况下是启用的。大多数插桩默认是禁用的，只启用了全局、线程、语句和事务插桩。从8.0版本开始，默认情况下还启用了元数据锁和内存插桩。

mysql、information_schema和performance_schema数据库没有启用插桩，但所有其他对象、线程和actor都启用了插桩。

大多数实例、句柄和设置表都是自动调整大小的。_history表会存储每个线程的最后10个事件，_history_long表则存储每个线程的最后10000个事件。存储的SQL文本的最大长度为1024字节。SQL摘要的最大长度也是1024字节。超出部分会被截断（right-trimmed）。

使用**Performance Schema**

至此已经介绍了Performance Schema是如何配置的。接下来将通过一些示例来演示如何使用Performance Schema解决常见的故障案例。

检查**SQL**语句

正如在本章前面的“插桩元件”一节中提到的，Performance Schema提供了一组丰富的插桩来检查SQL语句的性能。你可以找到用于标准预处理语句和存储例程的插桩。使用performance_schema，很容易找到引起性能问题的查询以及原因。

要启用语句检测，需要启用statement类型的插桩，如表3-3所述。

表3-3：statement类型的插桩及其描述

插桩类	描述
statement/sql	SQL 语句，如 SELECT，或者 CREATE TABLE
statement/sp	存储过程控制
statement/scheduler	事件调度器
statement/com	命令，如 quit、kill、DROP DATABASE，或者 Binlog Dump。有些命令是用户不可用的，只能由 mysqld 进程调用
statement/abstract	包括四类命令：clone、Query、new_packet 和 relay_log

常规**SQL**语句

Performance Schema将语句指标存储在events_statements_current、events_statements_history和events_statements_history_long表中。这三个表具有相同的结构。

直接使用performance_schema。下面是一个event_statement_history条目的例子：

```
  THREAD_ID: 3200
  EVENT_ID: 22
  END_EVENT_ID: 23
  EVENT_NAME: statement/sql/select
  SOURCE: init_net_server_extension.cc:94
  TIMER_START: 878753511280779000
  TIMER_END: 878753544491277000
  TIMER_WAIT: 33210498000
  LOCK_TIME: 657000000
  SQL_TEXT: SELECT film.film_id, film.description FROM sakila.film INNER JOIN
 ( SELECT film_id FROM sakila.film ORDER BY title LIMIT 50, 5 )
 AS lim USING(film_id)
  DIGEST: 2fdac27c4a9434806da3b216b9fa71aca738f70f1e8888a581c4fb00a349224f
  DIGEST_TEXT: SELECT `film` . `film_id` , `film` . `description` FROM `sakila` .
 `film` INNER JOIN ( SELECT `film_id` FROM `sakila` . `film` ORDER BY
 `title` LIMIT?, ... ) AS `lim` USING ( `film_id` )
  CURRENT_SCHEMA: sakila
  OBJECT_TYPE: NULL
  OBJECT_SCHEMA: NULL
  OBJECT_NAME: NULL
  OBJECT_INSTANCE_BEGIN: NULL
  MYSQL_ERRNO: 0
  RETURNED_SQLSTATE: NULL
  MESSAGE_TEXT: NULL
  ERRORS: 0
  WARNINGS: 0
  ROWS_AFFECTED: 0
  ROWS_SENT: 5
  ROWS_EXAMINED: 10
  CREATED_TMP_DISK_TABLES: 0

CREATED_TMP_TABLES: 1
SELECT_FULL_JOIN: 0
SELECT_FULL_RANGE_JOIN: 0
SELECT_RANGE: 0
SELECT_RANGE_CHECK: 0
SELECT_SCAN: 2
SORT_MERGE_PASSES: 0
SORT_RANGE: 0
SORT_ROWS: 0
SORT_SCAN: 0
NO_INDEX_USED: 1
NO_GOOD_INDEX_USED: 0
NESTING_EVENT_ID: NULL
NESTING_EVENT_TYPE: NULL
NESTING_EVENT_LEVEL: 0
STATEMENT_ID: 25
```

这些列的含义在官方文档中都有解释（参见链接11），这里就不一一介绍了。表3-4列出

了可以用于标识需要优化查询的指标的列。并非所有这些列都是一样的。例如，CREATED_TMP_DISK_TABLES在大多数情况下是查询优化不良的标志，而4个与排序相关的列可能只是表明查询结果需要排序。重要性这一列表示该指标的重要程度。

表3-4：event statement history表中可以作为优化指标的列

列	描述	重要性
CREATED_TMP_DISK_TABLES	查询创建的磁盘临时表的数量。有两个选项可以解决这个问题：优化查询或增加内存临时表的最大大小	高
CREATED_TMP_TABLES	查询创建的内存临时表的数量。使用内存临时表本身并不是坏事。但是，如果基础表数量增加，超过了内存临时表的空间，可能会转换为磁盘临时表。最好能提前为这种情况做好准备	中
SELECT_FULL_JOIN	因为没有合适的索引，所以导致JOIN执行了全表扫描。除非表非常小，否则需要重新考虑索引的设计	高
SELECT_FULL_RANGE_JOIN	JOIN操作是否使用了被引用表的范围搜索	中
SELECT_RANGE	JOIN操作是否使用了范围搜索来解析第一个表中的行。这个通常不是什么问题	低
SELECT_RANGE_CHECK	如果JOIN操作没有索引，则会检查每一行之后的键。这是一种非常糟糕的症状，如果该值大于零，则需要重新考虑表中的索引设计	高
SELECT_SCAN	JOIN操作是否对第一个表执行了全表扫描。如果第一个表很大则会是一个问题	中

续表

列	描述	重要性
SORT_MERGE_PASSES	排序必须执行的合并过程数。如果该值大于零且查询性能较低，则可能需要增加sort_buffer_size的值	低
SORT_RANGE	是否使用的是范围排序	低
SORT_ROWS	排序的行数。如果排序的行数比返回的行数多，则可能需要优化查询	中 （参考“描述”列）
SORT_SCAN	排序是否是通过扫描表完成的。这是一个非常糟糕的迹象，除非有意选择表中的所有行而不使用索引	高
NO_INDEX_USED	查询没有使用索引	高，除非表非常小
NO_GOOD_INDEX_USED	查询所用的索引不是最合适的。如果该值大于零，则需要重新考虑表的索引设计	高

要找出哪些语句需要优化，可以选择上述列中的任何一列，并将其与0进行比较。例如，要找到所有没有使用合适索引的查询，可运行以下命令：

```
SELECT THREAD_ID, SQL_TEXT, ROWS_SENT, ROWS_EXAMINED, CREATED_TMP_TABLES,
NO_INDEX_USED, NO_GOOD_INDEX_USED
FROM performance_schema.events_statements_history_long
WHERE NO_INDEX_USED > 0 OR NO_GOOD_INDEX_USED > 0;
```

要查询所有创建了临时表的查询，可运行：

```
SELECT THREAD_ID, SQL_TEXT, ROWS_SENT, ROWS_EXAMINED, CREATED_TMP_TABLES,
CREATED_TMP_DISK_TABLES
FROM performance_schema.events_statements_history_long
WHERE CREATED_TMP_TABLES > 0 OR CREATED_TMP_DISK_TABLES > 0;
```

可以使用这些列值单独显示潜在的问题。例如，要查找所有返回错误的查询，可以使用条件WHERE errors > 0；要找到所有执行时间超过5秒的查询，可以使用条件WHERE TIMER_WAIT > 5000000000，等等。

或者也可以创建一个查询，使用一系列条件查找所有有问题的语句，如下所示：

```
WHERE ROWS_EXAMINED > ROWS_SENT
OR ROWS_EXAMINED > ROWS_AFFECTED
OR ERRORS > 0
OR CREATED_TMP_DISK_TABLES > 0
OR CREATED_TMP_TABLES > 0
OR SELECT_FULL_JOIN > 0
OR SELECT_FULL_RANGE_JOIN > 0
OR SELECT_RANGE > 0
OR SELECT_RANGE_CHECK > 0
```

```
    OR SELECT_SCAN > 0
    OR SORT_MERGE_PASSES > 0
    OR SORT_RANGE > 0
    OR SORT_ROWS > 0
    OR SORT_SCAN > 0
    OR NO_INDEX_USED > 0
    OR NO_GOOD_INDEX_USED > 0
```

使用sys schema。sys schema提供了可用于查找有问题语句的视图。例如，statements_with_errors_or_warnings列出了带有错误和警告的所有语句，statements_with_full_table_scans列出了需要全表扫描的所有语句。sys schema使用摘要文本而不是查询文本，因此你将获得查询文本的摘要，而不是像直接访问Performance Schema那样获得的要么是SQL语句要么是摘要文本：

```
mysql> SELECT query, total_latency, no_index_used_count, rows_sent,
    -> rows_examined
    -> FROM sys.statements_with_full_table_scans
    -> WHERE db='employees' AND
    -> query NOT LIKE '%performance_schema%'\G
*********************** 1. row ***********************
 query: SELECT COUNT ( 'emp_no' ) FROM ... 'emp_no' )
 WHERE 'title' = ?
 total_latency: 805.37 ms
 no_index_used_count: 1
 rows_sent: 1
 rows_examined: 397774
 …
```

其他可以用来查找需要优化的语句的视图如表3-5所示。

表3-5：可用于查找需要优化的语句的视图

视图	描述
statement_analysis	具有聚合统计信息的规范化语句视图，按每个规范化语句的总执行时间排序。类似于events_statements_summary_by_digest表，但没有那么详细
statements_with_errors_or_warnings	所有引起错误或警告的规范化语句
statements_with_full_table_scans	所有执行了全表扫描的规范化语句
statements_with_runtimes_in_95th_percentile	所有平均执行时间在前95%的规范化语句
statements_with_sorting	所有执行了排序的规范化语句。该视图包括各种类型的排序
statements_with_temp_tables	所有使用了临时表的规范化语句

预处理语句

prepared_statements_instances表包含服务器中存在的所有预处理语句。它和events_statements_[current|history|history_long]表有相同的统计数据，此外还有关于预处理语句所属的线程以及该语句被执行了多少次的信息。和events_statements_[current|history|history_long]表不同的是，统计数据是累加的，这个表包含所有语句执行的总量。

COUNT_EXECUTE列包含语句的执行次数，因此可以用总数除以该列的数字来获得每个语句的平均统计信息。然而，请注意，任何平均统计数据都可能不准确。例如，如果一条语句执行了10次，并且SUM_SELECT_FULL_JOIN列的值是10，那么平均每个语句将有一个FULL JOIN。如果在表中添加一个合适的索引并再次执行该语句，SUM_SELECT_FULL_JOIN将保持10，因此平均值将是10/11=0.9。这并不表明问题已经解决。

要启用预处理语句检测，需要启用表3-6所示的插桩。

表3-6：启用预处理语句检测的插桩

插桩类	描述
statement/sql/prepare_sql	文本协议中的 PREPARE 语句（通过 MySQL CLI 运行）
statement/sql/execute_sql	文本协议中的 EXECUTE 语句（通过 MySQL CLI 运行）
statement/com/Prepare	二进制协议中的 PREPARE 语句（通过 MySQL C API 访问）
statement/com/Execute	二进制协议中的 EXECUTE 语句（通过 MySQL C API 访问）

一旦启用预处理语句功能，一个预处理好的语句就可以多次执行：

```
mysql> PREPARE stmt FROM
    -> 'SELECT COUNT(*) FROM employees WHERE hire_date > ?';
Query OK, 0 rows affected (0.00 sec)
Statement prepared

mysql1> SET @hd='1995-01-01';
Query OK, 0 rows affected (0.00 sec)

mysql1> EXECUTE stmt USING @hd;
+----------+
| count(*) |
+----------+
| 34004    |
+----------+
1 row in set (1.44 sec)

-- Execute a few more times with different values
```

然后查看诊断结果：

```
mysql2> SELECT statement_name, sql_text, owner_thread_id,
     -> count_reprepare, count_execute, sum_timer_execute
     -> FROM prepared_statements_instances\G
*************************** 1. row ***************************
 statement_name: stmt
 sql_text: select count(*) from employees where hire_date > ?
 owner_thread_id: 22
 count_reprepare: 0
 count_execute: 3
 sum_timer_execute: 4156561368000
1 row in set (0.00 sec)
```

注意，只有在服务器中存在的预处理语句才能在prepared_statements_instances表中查到统计信息。一旦预处理语句被删除，就不能再访问这些统计信息了：

```
mysql1> DROP PREPARE stmt;
Query OK, 0 rows affected (0.00 sec)

mysql2> SELECT * FROM prepared_statements_instances\G
Empty set (0.00 sec)
```

存储例程

使用performance_schema可以检索有关存储例程如何执行的信息：例如，IF...ELSE流控制语句的哪个分支被选择了，或者是否调用了错误处理程序。

要启用存储例程检测，需要启用匹配'statement/sp/%'模式的插桩。statement/sp/stmt插桩负责例程内部调用的语句，而其他插桩则负责跟踪事件，例如进入或离开过程、循环或任何其他控制指令。

可以使用如下存储例程来演示存储例程探测的工作原理：

```
CREATE DEFINER='root'@'localhost' PROCEDURE 'sp_test'(val int)
BEGIN
  DECLARE CONTINUE HANDLER FOR 1364, 1048, 1366
  BEGIN
    INSERT IGNORE INTO t1 VALUES('Some string');
    GET STACKED DIAGNOSTICS CONDITION 1 @stacked_state = RETURNED_SQLSTATE;
    GET STACKED DIAGNOSTICS CONDITION 1 @stacked_msg = MESSAGE_TEXT;
  END;
  INSERT INTO t1 VALUES(val);
END
```

接下来使用不同的值来调用以上存储过程：

```
mysql> CALL sp_test(1);
Query OK, 1 row affected (0.07 sec)

mysql> SELECT THREAD_ID, EVENT_NAME, SQL_TEXT
```

```
    -> FROM EVENTS_STATEMENTS_HISTORY
    -> WHERE EVENT_NAME LIKE 'statement/sp%';
+-----------+-------------------------+----------------------------+
| THREAD_ID | EVENT_NAME              | SQL_TEXT                   |
+-----------+-------------------------+----------------------------+
|        24 | statement/sp/hpush_jump | NULL                       |
|        24 | statement/sp/stmt       | INSERT INTO t1 VALUES(val) |
|        24 | statement/sp/hpop       | NULL                       |
+-----------+-------------------------+----------------------------+
3 rows in set (0.00 sec)
```

在本例中，没有调用错误处理程序，过程将参数值（1）插入表中：

```
mysql> CALL sp_test(NULL);
Query OK, 1 row affected (0.07 sec)

mysql> SELECT THREAD_ID, EVENT_NAME, SQL_TEXT
    -> FROM EVENTS_STATEMENTS_HISTORY
    -> WHERE EVENT_NAME LIKE 'statement/sp%';
+-----------+-------------------------+-----------------------------+
| THREAD_ID | EVENT_NAME              | SQL_TEXT                    |
+-----------+-------------------------+-----------------------------+
|        24 | statement/sp/hpush_jump | NULL                        |
|        24 | statement/sp/stmt       | INSERT INTO t1 VALUES(val)  |
|        24 | statement/sp/stmt       | INSERT IGNORE INTO t1
                                                VALUES(‘Some str... |
|        24 | statement/sp/stmt       | GET STACKED DIAGNOSTICS
                                                CONDITION 1 @s...  |
|        24 | statement/sp/stmt       | GET STACKED DIAGNOSTICS
                                                CONDITION 1 @s...  |
|        24 | statement/sp/hreturn    | NULL                        |
|        24 | statement/sp/hpop       | NULL                        |
+-----------+-------------------------+-----------------------------+
7 rows in set (0.00 sec)
```

然而，在第二次调用中，events_statements_history表的内容是不同的：包含了来自错误处理程序和替换错误的SQL语句的调用。

虽然过程本身的返回值没有改变，但我们清楚地看到它的执行方式不同。理解例程执行流中的这些差异有助于理解为什么同一个例程在某次调用后几乎可以立即完成，而在再次调用时则需要更长的时间。

语句剖析

events_stages_[current|history|history_long]表包含剖析信息，例如MySQL在创建临时表、更新或等待锁时花费了多少时间。要启用剖析，需要启用上述消费者表以及匹配'stage/%'模式的插桩。启用后可以找到类似“查询执行的哪个阶段花费了非常长的时间”等问题的答案。下面的例子进入搜索耗时超过1秒的阶段：

```
mysql> SELECT eshl.event_name, sql_text,
    ->        eshl.timer_wait/10000000000 w_s
    -> FROM performance_schema.events_stages_history_long eshl
    -> JOIN performance_schema.events_statements_history_long esthl
    -> ON (eshl.nesting_event_id = esthl.event_id)
    -> WHERE eshl.timer_wait > 1*10000000000\G
*************************** 1. row ***************************
 event_name: stage/sql/Sending data
 sql_text: SELECT COUNT(emp_no) FROM employees JOIN salaries
 USING(emp_no) WHERE hire_date=from_date
 w_s: 81.7
1 row in set (0.00 sec)
```

使用events_stages_[current|history|history_long]表的另一种技术是注意那些在已知阶段中花费了超过某个阈值的语句，这些语句会导致性能问题。表3-7列出了这些阶段。

表3-7：代表不同性能问题的阶段

阶段类	描述
stage/sql/%tmp%	所有与临时表相关的内容
stage/sql/%lock%	所有与锁相关的内容
stage/%/Waiting for%	所有与等待资源相关的内容
stage/sql/Sending data	这个阶段应该与语句统计中的 ROWS_SENT 进行比较。如果 ROWS_SENT 很小，那么在这个阶段花费大量时间的语句可能意味着必须创建一个临时文件或表来解析中间结果。在向客户端发送数据之前，通常会对行进行过滤。这通常是查询优化不良的症状
stage/sql/freeing items stage/sql/cleaning up stage/sql/closing tables stage/sql/end	这些是释放资源的阶段。不幸的是，它们不够详细，每个阶段都包含不止一个任务。如果发现查询在这些阶段花费了很长时间，那么很可能由于高并发性而遇到资源争用。你需要检查 CPU、I/O 和内存使用情况，以及硬件和 MySQL 选项设置是否足够处理应用程序创建的并发性

非常重要的一点是，只有通用服务模块支持剖析。存储引擎不支持使用performance_schema进行剖析。因此，类似stage/sql/update等阶段意味着任务在存储引擎中运行，不仅包含更新自身的统计，还会包含等待存储引擎特定的锁或其他争用问题。

检查读写性能

Performance Schema中的statement类型的插桩对于理解工作负载是受读还是受写限制非常有用。可以从统计各类型语句的执行量入手：

```
mysql> SELECT EVENT_NAME, COUNT(EVENT_NAME)
    -> FROM events_statements_history_long
    -> GROUP BY EVENT_NAME;
+----------------------+-------------------+
| EVENT_NAME           | COUNT(EVENT_NAME) |
+----------------------+-------------------+
| statement/sql/insert |               504 |
| statement/sql/delete |               502 |
| statement/sql/select |              6987 |
| statement/sql/update |              1007 |
| statement/sql/commit |               500 |
| statement/sql/begin  |               500 |
+----------------------+-------------------+
6 rows in set (0.03 sec)
```

在本例中，SELECT查询的数量大于任何其他查询的数量。这表明该场景中的大多数查询都是读查询。

如果想知道语句的延迟情况，可以按LOCK_TIME列进行聚合：

```
mysql> SELECT EVENT_NAME, COUNT(EVENT_NAME),
    -> SUM(LOCK_TIME/1000000) AS latency_ms
    -> FROM events_statements_history
    -> GROUP BY EVENT_NAME ORDER BY latency_ms DESC;
+---------------------------------+-------------------+------------+
| EVENT_NAME                      | COUNT(EVENT_NAME) | latency_ms |
+---------------------------------+-------------------+------------+
| statement/sql/select            |               194 |  7362.0000 |
| statement/sql/update            |                33 |  1276.0000 |
| statement/sql/insert            |                16 |   599.0000 |
| statement/sql/delete            |                16 |   470.0000 |
| statement/sql/show_status       |                 2 |   176.0000 |
| statement/sql/begin             |                 4 |     0.0000 |
| statement/sql/commit            |                 2 |     0.0000 |
| statement/com/Ping              |                 2 |     0.0000 |
| statement/sql/show_engine_status |                1 |     0.0000 |
+---------------------------------+-------------------+------------+
9 rows in set (0.01 sec)
```

如果还想知道读取和写入的字节数和行数，可以使用全局状态变量Handler_*：

```
mysql> WITH rows_read AS (SELECT SUM(VARIABLE_VALUE) AS rows_read
    -> FROM global_status
    -> WHERE VARIABLE_NAME IN ('Handler_read_first', 'Handler_read_key',
    -> 'Handler_read_next', 'Handler_read_last', 'Handler_read_prev',
    -> 'Handler_read_rnd', 'Handler_read_rnd_next')),
    -> rows_written AS (SELECT SUM(VARIABLE_VALUE) AS rows_written
    -> FROM global_status
    -> WHERE VARIABLE_NAME IN ('Handler_write'))
    -> SELECT * FROM rows_read, rows_written\G
*************************** 1. row ***************************
rows_read: 169358114082
rows_written: 33038251685
1 row in set (0.00 sec)
```

检查元数据锁

元数据锁用于保护数据库对象定义不被修改。执行任何SQL语句都需要获取共享元数据锁：SELECT、UPDATE等，这不会影响其他需要获取共享元数据锁的语句。但是，共享元数据锁会阻止那些更改数据库对象定义的语句，比如ALTER TABLE或CREATE INDEX，直到锁被释放为止。虽然大多数元数据锁冲突由表引起，但元数据锁本身是可以在各种数据库对象上设置的，比如SCHEMA、EVENT、TABLESPACE等。

事务执行期间会一直持有元数据锁。多语句事务的使用会使故障排除变得更加困难。很容易搞清楚哪个语句在等待元数据锁：DDL语句会隐式提交事务，因此它们是新事务中唯一的语句，并且可以在进程列表中发现它们处于“waiting for a metadata lock”状态。但是在进程列表中可能找不到持有元数据锁的语句，这些语句已经执行完成，但是包含这些语句的事务还没有提交。

performance_schema中的metadata_locks表包含关于当前由不同线程设置的锁的信息，以及处于等待状态的锁请求信息。通过这种方式，可以轻松确定哪个线程阻塞了DDL请求，你可以决定是终止该语句还是等待它完成执行。

要启用元数据锁监测，需要启用wait/lock/meta-data/sql/mdl插桩。

从以下示例中可以看到，在ID为5的进程列表中可见的线程持有processlist_id=4的线程正在等待的锁：

```
mysql> SELECT processlist_id, object_type,
    -> lock_type, lock_status, source
    -> FROM metadata_locks JOIN threads ON (owner_thread_id=thread_id)
    -> WHERE object_schema='employees' AND object_name='titles'\G
*************************** 1. row ***************************
 processlist_id: 4
 object_type: TABLE
 lock_type: EXCLUSIVE
 lock_status: PENDING -- waits
 source: mdl.cc:3263
*************************** 2. row ***************************
 processlist_id: 5
 object_type: TABLE
 lock_type: SHARED_READ
 lock_status: GRANTED -- holds
 source: sql_parse.cc:5707
```

检查内存使用情况

要在performance_schema中启用内存监测，请启用memory类的插桩。启用后就可以查看MySQL内部结构如何使用内存的详细信息。

直接使用**performance schema**

Performance Schema将内存使用统计信息存储在摘要表中，摘要表的名称以memory_summary_前缀开头。内存使用聚合统计，其参数如表3-8所示。

表3-8：内存使用的聚合参数

聚合参数	描述
global	按事件名全局聚合
thread	按线程聚合：包括后台线程和用户线程
account	按用户账号聚合
host	按主机聚合
user	按用户名聚合

例如，要找到占用大部分内存的InnoDB结构，可以执行以下查询：

```
mysql> SELECT EVENT_NAME,
    -> CURRENT_NUMBER_OF_BYTES_USED/1024/1024 AS CURRENT_MB,
    -> HIGH_NUMBER_OF_BYTES_USED/1024/1024 AS HIGH_MB
    -> FROM performance_schema.memory_summary_global_by_event_name
    -> WHERE EVENT_NAME LIKE 'memory/innodb/%'
    -> ORDER BY CURRENT_NUMBER_OF_BYTES_USED DESC LIMIT 10;
+---------------------------+-------------+-------------+
| EVENT_NAME                | CURRENT_MB  | HIGH_MB     |
+---------------------------+-------------+-------------+
| memory/innodb/buf_buf_pool | 130.68750000 | 130.68750000 |
| memory/innodb/ut0link_buf  |  24.00006104 |  24.00006104 |
| memory/innodb/buf0dblwr    |  17.07897949 |  24.96951294 |
| memory/innodb/ut0new       |  16.07891273 |  16.07891273 |
| memory/innodb/sync0arr     |   6.25006866 |   6.25006866 |
| memory/innodb/lock0lock    |   4.85086060 |   4.85086060 |
| memory/innodb/ut0pool      |   4.00003052 |   4.00003052 |
| memory/innodb/hash0hash    |   3.69776917 |   3.69776917 |
| memory/innodb/os0file      |   2.60422516 |   3.61988068 |
| memory/innodb/memory       |   1.23812866 |   1.42373657 |
+---------------------------+-------------+-------------+
10 rows in set (0,00 sec)
```

使用sys schema

使用Sys schema中的视图可以更好地获取内存统计信息，可以按host、user、thread或global进行聚合。memory_global_total视图包含一个单独的值，显示被监测内存的总量：

```
mysql> SELECT * FROM sys.memory_global_total;
+-----------------+
| total_allocated |
+-----------------+
| 441.84 MiB      |
+-----------------+
1 row in set (0,09 sec)
```

聚合视图根据需要将字节转换为KB、MB和GB。memory_by_thread_by_current_bytes视图有一个user列，该列可以采用以下值之一。

NAME@HOST

普通用户账户，例如sveta@oreilly.com。

系统用户，如**sql/main**或**innodb/***

此类“usernames”的数据取自threads表，当需要了解特定线程正在做什么时，这些数据使用起来非常方便。

视图memory_by_thread_by_current_bytes中的行是按照当前分配的内存降序排序的，所以很容易就能找到哪个线程占用了大部分内存：

```
mysql> SELECT thread_id tid, user,
    -> current_allocated ca, total_allocated
    -> FROM sys.memory_by_thread_by_current_bytes LIMIT 9;
+-----+---------------------------+-----------+-----------------+
| tid | user                      | ca        | total_allocated |
+-----+---------------------------+-----------+-----------------+
|  52 | sveta@localhost           | 1.36 MiB  | 10.18 MiB       |
|   1 | sql/main                  | 1.02 MiB  | 4.95 MiB        |
|  33 | innodb/clone_gtid_thread  | 525.36 KiB| 24.04 MiB       |
|  44 | sql/event_scheduler       | 145.72 KiB| 4.23 MiB        |
|  43 | sql/slave_sql             | 48.74 KiB | 142.46 KiB      |
|  42 | sql/slave_io              | 20.03 KiB | 232.23 KiB      |
|  48 | sql/compress_gtid_table   | 13.91 KiB | 17.06 KiB       |
|  25 | innodb/fts_optimize_thread| 1.92 KiB  | 2.00 KiB        |
|  34 | innodb/srv_purge_thread   | 1.56 KiB  | 1.64 KiB        |
+-----+---------------------------+-----------+-----------------+
9 rows in set (0,03 sec)
```

上面的例子是在一台笔记本电脑中运行的，因此，数据不能用来做生产服务器参考。显然，本地连接使用了大部分内存，其次是主服务器进程。

当需要找到占用最多内存的用户线程时，内存监测非常方便。在下面的示例中，一个用户连接被分配了36 GB的RAM，即使在现代高内存系统中也算是相当大的了：

```
mysql> SELECT * FROM sys.memory_by_thread_by_current_bytes
    -> ORDER BY current_allocated desc\G
*************************** 1. row ***************************
 thread_id: 152
 user: lj@127.0.0.1
 current_count_used: 325
 current_allocated: 36.00 GiB
 current_avg_alloc: 113.43 MiB
 current_max_alloc: 36.00 GiB
 total_allocated: 37.95 GiB
...
```

检查变量

Performance Schema将变量监测提升到了一个新的水平。它为以下方面提供了工具：

- 服务器变量
  —全局级
  —会话级，针对当前所有打开的会话
  —源，所有当前变量值的来源
- 状态变量
  —全局级
  —会话级，针对当前所有打开的会话

—聚合维度

—主机

—用户名

—账号

—线程

- 用户变量

在5.7版本之前，服务器和状态变量是在information_schema中配置的。这种配置是有限制的：只允许跟踪全局和当前会话值。其他会话中关于变量和状态的信息，以及关于用户变量的信息，都是不可访问的。但是，出于向后兼容的考虑，MySQL 5.7还是使用information_schema来跟踪变量。要启用performance_schema对变量跟踪的支持，需要将配置变量show_compatibility_56设置为0。这一要求以及information_schema中的变量表在8.0版中都不再存在。

全局变量值被存储在表global_variables中。当前会话的会话变量被存储在session_variables表中。这两个表中只有两列具有自解释的名称：VARIABLE_NAME和VARIABLE_VALUE。

variables_by_thread表中有一个额外的列THREAD_ID，表示变量所属的线程。这允许你查找将会话变量值设置为不同于默认值的线程。

在下面的例子中，THREAD_ID=84的线程将变量tx_isolation设置为SERIALIZABLE，这可能会导致事务获得比使用默认锁级别更多的锁的情况：

```
mysql> SELECT * FROM variables_by_thread
    -> WHERE VARIABLE_NAME='tx_isolation';
+-----------+---------------+-----------------+
| THREAD_ID | VARIABLE_NAME |  VARIABLE_VALUE |
+-----------+---------------+-----------------+
|        71 |  tx_isolation | REPEATABLE-READ |
|        83 |  tx_isolation | REPEATABLE-READ |
|        84 |  tx_isolation | SERIALIZABLE    |
+-----------+---------------+-----------------+
3 rows in set, 3 warnings (0.00 sec)
```

下面的例子查找所有和当前会话变量值不同的线程和会话变量：

```
mysql> SELECT vt2.THREAD_ID AS TID, vt2.VARIABLE_NAME,
    -> vt1.VARIABLE_VALUE AS MY_VALUE,
    -> vt2.VARIABLE_VALUE AS OTHER_VALUE
    -> FROM performance_schema.variables_by_thread vt1
    -> JOIN performance_schema.threads t USING(THREAD_ID)
    -> JOIN performance_schema.variables_by_thread vt2
    -> USING(VARIABLE_NAME)
    -> WHERE vt1.VARIABLE_VALUE != vt2.VARIABLE_VALUE
    -> AND t.PROCESSLIST_ID=@@pseudo_thread_id;
+-----+--------------------+------------------+--------------------+
| TID | VARIABLE_NAME      | MY_VALUE         | OTHER_VALUE        |
+-----+--------------------+------------------+--------------------+
|  42 | max_allowed_packet | 67108864         | 1073741824         |
|  42 | pseudo_thread_id   | 22715            | 5                  |
|  42 | timestamp          | 1626650242.678049 | 1626567255.695062  |
|  43 | gtid_next          | AUTOMATIC        | NOT_YET_DETERMINED |
|  43 | pseudo_thread_id   | 22715            | 6                  |
|  43 | timestamp          | 1626650242.678049 | 1626567255.707031  |
+-----+--------------------+------------------+--------------------+
6 rows in set (0,01 sec)
```

全局和当前会话状态值分别被存储在表global_status和session_status中。它们也只有两列：VARIABLE_NAME和VARIABLE_VALUE。

状态变量可以按用户账户、主机、用户和线程聚合。在我看来，最有趣的是线程聚合，因为它允许你快速识别哪个连接在服务器上造成了大部分资源压力。例如，下面的代码片段清楚地显示了THREAD_ID=83的连接正在执行大部分写入操作：

```
mysql> SELECT * FROM status_by_thread
    -> WHERE VARIABLE_NAME='Handler_write';
+-----------+---------------+----------------+
| THREAD_ID | VARIABLE_NAME | VARIABLE_VALUE |
+-----------+---------------+----------------+
|        71 | Handler_write | 94             |
|        83 | Handler_write | 4777777777     | --大部分写入操作
|        84 | Handler_write | 101            |
+-----------+---------------+----------------+
3 rows in set (0.00 sec)
```

用户定义的变量可以通过SET@my_var='foo'创建，并在表user_variable_by_thread中跟踪：

```
mysql> SELECT * FROM user_variables_by_thread;
+-----------+---------------+----------------+
| THREAD_ID | VARIABLE_NAME | VARIABLE_VALUE |
+-----------+---------------+----------------+
|        71 | baz           | boo            |
|        84 | foo           | bar            |
+-----------+---------------+----------------+
2 rows in set (0.00 sec)
```

当需要查找内存消耗源时，这种检测非常有用，因为每个变量都需要字节来保存其值。还可以使用用户定义的变量来解决持久性连接的棘手问题。最后同样重要的是，这个表是找出你在自己的会话中定义了哪些变量的唯一方法。

表variables_info中不包含任何变量值，但它有关于服务器变量起源的信息和其他文档，例如，变量默认的最小值和最大值。SET_TIME列包含最新变量更改的时间戳。SET_HOST和SET_USER列标识设置变量的用户账号。例如，要查找自服务器启动以来动态更改的所有变量，请运行：

```
mysql> SELECT * FROM performance_schema.variables_info
    -> WHERE VARIABLE_SOURCE = 'DYNAMIC'\G
*************************** 1. row ***************************
 VARIABLE_NAME: foreign_key_checks
 VARIABLE_SOURCE: DYNAMIC
 VARIABLE_PATH:
    MIN_VALUE: 0
    MAX_VALUE: 0
    SET_TIME: 2021-07-18 03:14:15.560745
    SET_USER: NULL
    SET_HOST: NULL
*************************** 2. row ***************************
 VARIABLE_NAME: sort_buffer_size
 VARIABLE_SOURCE: DYNAMIC
 VARIABLE_PATH:
    MIN_VALUE: 32768
    MAX_VALUE: 18446744073709551615
    SET_TIME: 2021-07-19 02:37:11.948190
    SET_USER: sveta
    SET_HOST: localhost
2 rows in set (0,00 sec)
```

VARIABLE_SOURCE的值可设置为如下几项。

COMMAND_LINE

在命令行中设置的变量。

COMPILED

编译的默认值。

PERSISTED

在服务器指定的*mysqld-auto.cnf*选项文件中设置。

变量也有许多选项，设置在不同的选项文件中。我不会全部讨论：它们要么是自描述性的，要么很容易在用户参考手册中查看。每个版本的细节数量也在增加。

检查最常见的错误

除了特定错误信息，performance_schema还提供摘要表，可以按用户、主机、账户、线程和错误号聚合错误信息。所有的聚合表都有类似于events_errors_summary_global_by_error表的结构：

```
mysql> USE performance_schema;
mysql> SHOW CREATE TABLE events_errors_summary_global_by_error\G
*************************** 1. row ***************************
       Table: events_errors_summary_global_by_error
Create Table: CREATE TABLE `events_errors_summary_global_by_error` (
  `ERROR_NUMBER` int DEFAULT NULL,
  `ERROR_NAME` varchar(64) DEFAULT NULL,
  `SQL_STATE` varchar(5) DEFAULT NULL,
  `SUM_ERROR_RAISED` bigint unsigned NOT NULL,
  `SUM_ERROR_HANDLED` bigint unsigned NOT NULL,
  `FIRST_SEEN` timestamp NULL DEFAULT '0000-00-00 00:00:00',
  `LAST_SEEN` timestamp NULL DEFAULT '0000-00-00 00:00:00',
  UNIQUE KEY `ERROR_NUMBER` (`ERROR_NUMBER`)
) ENGINE=PERFORMANCE_SCHEMA DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
1 row in set (0,00 sec)
```

可通过列ERROR_NUMBER、ERROR_NAME和SQL_STATE来识别错误。SUM_ERROR_RAISED是错误发生的次数。SUM_ERROR_HANDLED是错误被处理的次数。FIRST_SEEN和LAST_SEEN是错误第一次发生和最后一次发生的时间戳。

有些聚合表有额外的列。表events_errors_summary_by_thread_by_error有一个名为THREAD_ID的列，标识了引发错误的线程，表events_errors_summary_by_host_by_error有一个名为HOST的列，依此类推。

例如，要查找所有运行语句发生错误超过10次的账户，可以运行：

```
mysql> SELECT * FROM
    -> performance_schema.events_errors_summary_by_account_by_error
    -> WHERE SUM_ERROR_RAISED > 10 AND USER IS NOT NULL
    -> ORDER BY SUM_ERROR_RAISED DESC\G
*************************** 1. row ***************************
   USER: sveta
   HOST: localhost
   ERROR_NUMBER: 3554
   ERROR_NAME: ER_NO_SYSTEM_TABLE_ACCESS
   SQL_STATE: HY000
   SUM_ERROR_RAISED: 60
   SUM_ERROR_HANDLED: 0
   FIRST_SEEN: 2021-07-18 03:14:59
   LAST_SEEN: 2021-07-19 02:50:13
1 row in set (0,01 sec)
```

错误摘要表可用于找出哪些账号、主机、用户或线程发送错误最多的查询并执行操作。还可以帮助解决诸如ER_DEPRECATED_UTF8_ALIAS之类的错误，这可能表明一些常用的查询是为以前的MySQL版本编写的，需要更新。

检查**Performance Schema**自身

可以使用相同的插桩和消费者表来检查Performance Schema本身。请注意，默认情况下，

如果Performance_schema被设为默认数据库，则不会跟踪对它的查询。如果需要检查对performance_schema的查询，则需要首先更新setup_actors表。

一旦setup_actors表被更新，就可以使用所有的插桩。例如，要找到performance_schema中消耗内存最多的10个表，可以运行：

```
mysql> SELECT SUBSTRING_INDEX(EVENT_NAME, '/', -1) AS EVENT,
    -> CURRENT_NUMBER_OF_BYTES_USED/1024/1024 AS CURRENT_MB,
    -> HIGH_NUMBER_OF_BYTES_USED/1024/1024 AS HIGH_MB
    -> FROM performance_schema.memory_summary_global_by_event_name
    -> WHERE EVENT_NAME LIKE 'memory/performance_schema/%'
    -> ORDER BY CURRENT_NUMBER_OF_BYTES_USED DESC LIMIT 10;
+----------------------------------------+-------------+-------------+
| EVENT                                  | CURRENT_MB  | HIGH_MB     |
+----------------------------------------+-------------+-------------+
| events_statements_summary_by_digest    | 39.67285156 | 39.67285156 |
| events_statements_history_long         | 13.88549805 | 13.88549805 |
| events_errors_summary_by_thread_by_... | 11.81640625 | 11.81640625 |
| events_statements_summary_by_thread... |  9.79296875 |  9.79296875 |
| events_statements_history_long.dige... |  9.76562500 |  9.76562500 |
| events_statements_summary_by_digest... |  9.76562500 |  9.76562500 |
| events_statements_history_long.sql_... |  9.76562500 |  9.76562500 |
| memory_summary_by_thread_by_event_name |  7.91015625 |  7.91015625 |
| events_errors_summary_by_host_by_error |  5.90820313 |  5.90820313 |
| events_errors_summary_by_account_by... |  5.90820313 |  5.90820313 |
+----------------------------------------+-------------+-------------+
10 rows in set (0,00 sec)
```

使用sys schema可以获取同样的信息：

```
mysql> SELECT SUBSTRING_INDEX(event_name, '/', -1), current_alloc
    -> FROM sys.memory_global_by_current_bytes
    -> WHERE event_name LIKE 'memory/performance_schema/%' LIMIT 10;
+-----------------------------------------------+--------------+
| SUBSTRING_INDEX(event_name, '/', -1)          | current_alloc |
+-----------------------------------------------+--------------+
| events_statements_summary_by_digest           | 39.67 MiB     |
| events_statements_history_long                | 13.89 MiB     |
| events_errors_summary_by_thread_by_error      | 11.82 MiB     |
| events_statements_summary_by_thread_by_event_name | 9.79 MiB      |
| events_statements_history_long.digest_text    | 9.77 MiB      |
| events_statements_summary_by_digest.digest_text | 9.77 MiB      |
| events_statements_history_long.sql_text       | 9.77 MiB      |
| memory_summary_by_thread_by_event_name        | 7.91 MiB      |
| events_errors_summary_by_host_by_error        | 5.91 MiB      |
| events_errors_summary_by_account_by_error     | 5.91 MiB      |
+-----------------------------------------------+--------------+
10 rows in set (0,00 sec)
```

还可以使用SHOW ENGINE PERFORMANCE_SCHEMA STATUS语句获取

performance_schema的相关信息：

```
mysql> SHOW ENGINE PERFORMANCE_SCHEMA STATUS\G
*************************** 1. row ***************************
 Type: performance_schema
 Name: events_waits_current.size
 Status: 176
*************************** 2. row ***************************
 Type: performance_schema
 Name: events_waits_current.count
 Status: 1536
*************************** 3. row ***************************
 Type: performance_schema
 Name: events_waits_history.size
 Status: 176
*************************** 4. row ***************************
 Type: performance_schema
 Name: events_waits_history.count
 Status: 2560
…
*************************** 244. row ***************************
 Type: performance_schema
 Name: (pfs_buffer_scalable_container).count
 Status: 17
*************************** 245. row ***************************
 Type: performance_schema
 Name: (pfs_buffer_scalable_container).memory
 Status: 1904
*************************** 246. row ***************************
 Type: performance_schema
 Name: (max_global_server_errors).count
 Status: 4890


*************************** 247. row ***************************
 Type: performance_schema
 Name: (max_session_server_errors).count
 Status: 1512
*************************** 248. row ***************************
 Type: performance_schema
 Name: performance_schema.memory
 Status: 218456400
248 rows in set (0,00 sec)
```

在输出中可以发现一些细节，比如消费者表中存储了多少特定事件，或者特定度量的最大值。最后一行包含Performance Schema当前占用的字节数。

小结

Performance Schema是一个经常受到批评的特性。早期版本的MySQL对其的实现不够理想，导致资源消耗较高。通常的建议是干脆关掉它。

这也被认为是难以理解的。只是启用了一些插桩代码，这些代码用于记录数据并将其提交给消费者表。消费者表是一些内存表，需要使用标准SQL语句查询数据，获取信息。了解了Performance Schema如何管理自己的内存后，你就能认识到MySQL并没有泄漏内存，它只是将消费者数据保存在内存中，这些内存只有在MySQL重启时才会释放。

我的建议很简单：应该启用Performance Schema，按需动态地启用插桩和消费者表，通过它们提供的数据可以解决可能存在的任何问题——查询性能、锁定、磁盘I/O、错误等。充分利用sys schema是解决常见问题的捷径。这样做将为你提供一种可以直接从MySQL中测量性能的方法。

第4章 操作系统和硬件优化

MySQL服务器的性能受限于整个系统最薄弱的环节，而承载它的操作系统和硬件往往是限制因素。磁盘空间大小、可用内存和CPU资源、网络以及连接它们的组件都限制了系统的最终容量。因此，你需要谨慎地选择硬件，并对硬件和操作系统进行适当的配置。例如，如果工作负载受I/O限制，一种方法是设计应用程序以最小化MySQL的I/O工作负载。然而更明智的做法往往是升级I/O子系统、安装更多内存或重新配置现有磁盘。即便是在云托管环境中，本章中的信息仍然非常有用，特别是对于理解文件系统限制和Linux I/O调度器。

什么限制了**MySQL**的性能

许多不同的硬件组件都会影响MySQL的性能，我们看到的最常见的瓶颈是CPU耗尽。当MySQL尝试并行执行太多的查询，或者当少量的查询在CPU上运行太长时间时，就可能会发生CPU饱和。

I/O饱和也会发生，但比CPU耗尽的频率低得多。这主要是因为普遍使用了固态硬盘（SSD）。过去，由于内存不足，数据库必须到机械硬盘（HDD）中获取数据，性能开销是非常大的。SSD通常比HDD[1]快10到20倍。现在，即使查询需要访问磁盘，也仍然有良好的性能。

内存耗尽的情况也会发生，但通常只在你试图将太多内存分配给MySQL时才会发生。我们在5.4节中会讨论防止这种情况发生的最佳配置。

如何为**MySQL**选择**CPU**

在升级当前硬件或购买新硬件时，应该考虑工作负载是否受CPU限制。可以通过检查CPU使用率来确定工作负载是否受CPU限制，但不要只查看CPU的总体负载，而是要查看最重要查询的CPU使用率和I/O之间的平衡，并注意CPU负载是否均匀。

一般来说，你的服务器要达到两个目标。

低延迟（快速响应时间）

为此需要更快的CPU，因为每个查询将只使用一个CPU。

高吞吐量

如果可以同时运行多个查询，那么可以使用多个CPU为查询提供服务。

如果工作负载没有用完所有的CPU资源，MySQL可以使用剩余的CPU来执行后台任务，比如清理InnoDB缓冲区、执行网络操作等。然而，与执行查询相比，这些任务通常是次要的。

平衡内存和磁盘资源

配置大内存的主要原因并不是为了在内存中保存大量数据，而是为了避免磁盘I/O，因为磁盘I/O比访问内存中的数据要慢几个数量级。重要的是平衡内存和磁盘空间大小、速度、成本和其他因素，以便让工作负载获得良好的性能。

缓存、读取和写入

如果有足够的内存，可以完全避开磁盘读取操作。如果所有数据都能装入内存，那么一旦服务器的缓存预热完成，每次读取都将是一次缓存命中。在这种情况下，仍然会从内存中进行逻辑读取，但不会从磁盘中进行物理读取。然而，写入是另一回事。写入可以像读取一样在内存中执行，但迟早必须被写入磁盘，才能持久保留数据。换句话说，缓存可以延迟写操作，但缓存不能像消除读操作那样消除写操作。

事实上，除了允许写操作延迟之外，缓存还允许它们以两种重要的方式组合在一起。

多次写操作，一次刷新

> 一个数据片段可以在内存中被多次更改，而无须每一次都将新值写入磁盘。当数据被最终刷新到磁盘时，自上次物理写入以来发生的所有修改都将被持久化。例如，许多语句可以更新内存中的计数器。如果计数器被更新了100次，然后写入磁盘，则100次内存修改被合并为一次磁盘写入。

I/O合并

> 许多不同的数据片段可以在内存中被修改，这些修改可以被收集在一起，因此物理写可以作为单个磁盘操作执行。

这就是为什么许多事务系统使用提前写日志（write-ahead logging）策略的原因。提前写日志允许在内存中更改页面，而不用将更改刷新到磁盘，这通常涉及随机I/O，速度非常慢。相反，它们将更改的记录写入顺序日志文件，这样要快得多。后台线程可以稍后将修改过的页面刷新到磁盘，这样做可以优化写操作的性能。

写操作从缓冲中获益，因为可以将随机I/O转换为顺序I/O。异步（缓冲）写操作通常由操作系统处理，并且是被成批处理的，因此可以更优地被刷新到磁盘。同步（无缓冲）写入必须等待数据落盘。这就是为什么廉价磁盘冗余阵列（RAID）控制器的电池保护回写缓存能提升写入性能（我们稍后讨论RAID）的原因。

你的工作集是什么

每个应用程序都有一个数据“工作集”，即它真正需要的数据。许多数据库也有大量不在工作集中的数据。你可以将数据库想象成一个带有存档抽屉的桌子。工作集由你需要放在桌面上处理的文件组成。在这个类比中，桌面表示主内存，而文件抽屉则表示硬盘。就像你不需要把每一张纸都放在桌面上一样，也不需要把整个数据库放进内存来获得最佳性能——仅仅将工作集放入内存即可。

在使用HDD硬盘时，最好尝试找到一个有效的内存/磁盘比率。这主要是由于HDD延迟较

高、IOPS较低。使用SSD时，内存对磁盘的比率就变得不那么重要了。

固态存储

固态（闪存）存储已经是大多数数据库系统的标配，尤其是在线事务处理（OLTP）。通常，只有在大规模的数据仓库系统或老旧的系统上才能找到HDD。这种转变发生在2015年左右。当时SSD的价格大幅下降。

固态存储设备使用非易失性闪存芯片组成的单元，而不是磁碟，它们也被称为非易失随机访问内存（NVRAM）。固态存储设备没有可移动的机械部件，这使得其行为与硬盘驱动器非常不同。

下面是闪存性能的简要总结。高质量的闪存设备具有如下性能。

比硬盘驱动器明显更好的随机读写性能

: 闪存设备读的能力通常略好于写的能力。

比硬盘驱动器更好的顺序读写性能

: 然而并没有像随机I/O那样有显著的改进，因为硬盘驱动器在随机I/O时比在顺序I/O时慢得多。

比硬盘驱动器更好的并发支持

: 闪存设备可以支持更多的并发操作，事实上，只有在拥有大量并发性的情况下，闪存设备才能真正实现最高吞吐量。

最重要的是同时提升了随机I/O和并发性。闪存在高并发性下提供了非常好的随机I/O性能。

闪存概述

带有旋转盘片和摆动磁头的硬盘驱动器有其固有的局限性和特性，这是所涉及的物理现象的后果。固态存储也是如此，它建立在闪存之上。不要以为固态存储很简单。在某些方面，它实际上比硬盘更复杂。闪存的局限性非常严重，很难克服，因此典型的固态设备具有复杂的体系结构，包含大量抽象、缓存和专有的“魔法”。

闪存最重要的特点是，它可以被快速地多次读取，而且以很小的单元读取数据，但写入则要困难得多。只有进行特殊的擦除操作之后，存储单元才能被重新写入数据，并且每次擦除的块大小较大（例如512K B）。擦除的过程非常缓慢，而且最终会导致块损耗。一个块能够承受的擦除次数取决于它所使用的底层技术，稍后详细介绍。

写操作的限制是固态存储复杂性的原因。这就是为什么有些SSD设备能提供稳定、一致的性能，而有些则不能。其中的关键区别在于专有的固件、驱动程序和其他能让固态设备运行的部件。为了使写操作能够很好地执行，并避免过早地耗尽闪存块写寿命，设备必须能够重新定位页面，并执行垃圾收集和所谓的损耗均衡。写入放大用来描述由于部分块写而导致的数据从一处移动到另一处、多次写入数据和元数据所产生的额外写操作。

垃圾收集

理解垃圾收集很重要。为了使某些块保持新鲜并为新的写入做好准备，设备会回收块。这

需要设备上有可用空间。要么设备内部有一些不可见的预留空间，要么需要用户预留空间，不要把空间填满；具体因设备而异。无论使用哪种方式，当设备被填满时，垃圾收集器必须更加努力地保持某些块的清洁，因此写放大因子会增加。

因此，许多设备在空间快满时会变慢。具体慢多少取决于不同的厂商和型号，这取决于设备的架构。有些设备即使非常满，也能实现高性能，但一般来说，100G B文件在160GB的SSD上的性能与在320GB的SSD上的性能是不同的。当没有空闲块时，必须等待擦除操作完成，从而导致速度减慢。写入空闲块需要几百微秒，但擦除速度要慢得多，通常需要几毫秒。

RAID性能优化

存储引擎通常将数据/索引保存在单个大文件中，这意味着RAID通常是存储大量数据的最佳选项。RAID在数据冗余、存储空间大小、缓存和速度方面能提供一些帮助。但与我们一直在研究的其他优化一样，RAID有很多不同的配置选项，选择一种适合的配置非常重要。

这里我们不会介绍所有RAID级别，也不会详细介绍不同RAID级别存储数据的具体方式。我们关注的是RAID配置如何满足数据库服务器的需求。以下是最重要的RAID级别。

RAID 0

RAID 0是最便宜和性能最好的RAID配置，至少在简单地衡量成本和性能时是这样的（如果要考虑数据恢复，它开始变得昂贵）。因为它不提供冗余，所以我们认为RAID 0永远不适合用于生产数据库，但是如果希望节省成本，那么在开发环境中，当服务器完全失效也不会变成突发事件时，可以选择RAID 0。

再次重申，虽然“RAID”中的R表示“冗余”，但是RAID 0不提供任何冗余。事实上，RAID 0阵列的故障概率高于任何单磁盘的故障概率，而不是更低！

RAID 1

RAID 1为许多场景提供了良好的读性能，并且可以跨磁盘复制数据，因此有良好的冗余。RAID 1的读取速度略高于RAID 0。这对处理日志记录和类似工作负载的服务器很好，因为顺序写入不需要许多底层磁盘就能执行得很好（与随机写入相反，随机写入可以从并行化中受益）。它是需要冗余但只有两个硬盘驱动器的低端服务器的典型选择。

RAID 0和RAID 1非常简单，通常可以在软件中被很好地实现。大多数操作系统都允许你轻松地创建软件RAID 0和RAID 1卷。

RAID 5

RAID 5过去对数据库系统来说非常可怕，这主要是因为它对性能的影响。随着SSD的普通应用，现在RAID 5也是一个可行的选择。RAID 5将数据分布在具有分布式奇偶校验块的多块磁盘上，以便在任何一块磁盘出现故障时，可以从奇偶校验块重建数据。如果两块磁盘同时出现故障，整个卷将无法恢复。就每单位存储成本而言，它是最经济的冗余配置，因为在整个阵列中，只会损失一块磁盘的存储空间。

RAID 5最让人捉摸不透的是，如果一块磁盘出现故障，阵列的表现如何。这是因为数据必须通过读取所有其他磁盘来重建。如果使用了HDD，这会严重影响性能，这就是为什么RAID 5通常不受欢迎。如果有很多块磁盘，情况会更糟。如果你试图在重建期间保持服务器在线，则不要期望重建操作或阵列自身的性能很好。其他性能成本包括可伸缩性的限制，因为奇偶校验块（RAID 5的可伸缩性不能超过10块磁盘）和缓存问题。良好的RAID 5的性能在很大程度上依赖于RAID控制器的缓存，这可能与数据库服务器的需求相冲突。正如我们在前面提到的，SSD在IOPS和吞吐量方面提供了显著的性能改进，而且还消除了性能较差的随机读写性能问题。

RAID 5比较受欢迎，RAID控制器经常对RAID 5进行高度优化，尽管存在理论上的限

制，但在某些工作负载下，RIAD 5智能控制器能够很好地使用缓存，其性能有时几乎可以达到RAID 10控制器的水平。这实际上可能反映了RAID 10控制器的优化程度较低，但不管什么原因，这就是我们所看到的。

RAID 6

RAID 5最大的问题是丢失两块磁盘时会产生灾难性的后果。阵列中的磁盘越多，磁盘出现故障的概率就越高。RAID 6通过增加第二块校验盘来降低故障的可能性。这允许你在承受两次磁盘故障的情况下仍然能重建阵列。它的缺点是计算额外的奇偶校验会使写操作比RAID 5慢。

RAID 10

RAID 10是一个非常好的数据存储选择。它由条带化的镜像对组成，因此在读写方面都能很好地被扩展。与RAID 5相比，它的重建速度快且容易。RAID 10也可以很好地在软件中被实现。

当一个硬盘驱动器失效时，性能损失仍然可能很大，因为该条带可能会成为瓶颈。根据工作负载的不同，性能可能会降低50%。需要注意的是，有些RAID 10的控制器使用的是“串联镜像”的方式实现的。这不是最优的，因为没有条带化：最常访问的数据可能只放在一对磁盘上，而不是分散在多块磁盘上，因此性能会很差。

RAID 50

RAID 50由条带化的RAID 5阵列组成，如果有很多磁盘，它可以很好地兼顾RAID 5的经济性和RAID 10的性能。这主要用于非常大的数据集，如数据仓库或非常大的OLTP系统。

表4-1为各种RAID级别的比较。

表4-1：不同RAID级别的比较

级别	概述	冗余	所需的磁盘	更快地读取	更快地写入
RAID 0	便宜、快速、危险	否	N	是	是
RAID 1	读取快速、简单、安全	是	通常是 2 块	是	否
RAID 5	便宜、使用 SSD 时速度快	是	N+1	是	视情况而定
RAID 6	与 RAID 5 类似，但容错能力更强	是	N+2	是	视情况而定
RAID 10	昂贵、快速、安全	是	2N	是	是
RAID 50	用于非常大的数据存储	是	2(N+1)	是	是

RAID故障、恢复和监控

RAID配置（RAID 0除外）提供冗余。这很重要，但也很容易低估多块磁盘同时出现故障的可能性。不要认为RAID是数据安全方面的强有力的保证。

RAID并不能消除，甚至不能减少备份的需要。出现问题时，恢复时间将取决于控制器、RAID级别、阵列大小、磁盘速度，以及重建阵列时是否需要保持服务器在线。

多块磁盘有可能同时发生故障。例如，电源尖峰或过热很容易损坏两块或多块磁盘。然而，更常见的是，两块磁盘同时发生故障。许多这样的问题可能会被忽视。一个常见的原因是保存在损坏的物理媒介中的数据很少被访问，可能几个月都不会被发现，直到尝试读

取数据，或者当另一个驱动器出现故障，RAID控制器尝试使用损坏的数据重建阵列时才发现问题。硬盘空间越大，出现这种情况的可能性就越大。

这就是为什么监控RAID阵列很重要的原因。大多数控制器都提供了一些软件来报告阵列的状态，你需要对此进行跟踪，否则可能完全不知道驱动器已出现故障。这可能会错过恢复数据和发现问题的机会，到第二个驱动器出现故障时就太晚了。你应该配置监控系统，以便在驱动器或卷变为降级或故障状态时发出警报。

通过定期主动检查阵列的一致性，可以降低潜在的损坏风险。后台巡检读取（Background Patrol Read）功能也有助于避免此类问题。后台巡检读取是某些控制器的一项功能，可以在所有驱动器都处于联机状态时检查损坏的介质并进行修复。与恢复一样，超大阵列的检查速度可能较慢，因此在创建大型阵列时，请确保制订相应的检查计划。

你还可以添加一个未使用的热备盘，将其配置为控制器的备用盘，以便控制器自动用于恢复。如果你依赖于每台服务器，这是一个好主意。对于只有几块硬盘驱动器的服务器来说成本很高，因为拥有一块空闲磁盘的成本相对较高，但是如果有很多磁盘，不使用热备盘几乎是愚蠢的。请记住，磁盘越多，驱动器发生故障的概率就越高。

除了监控驱动器的故障外，你还应该监控RAID控制器的电池备份单元和写缓存策略。如果电池发生故障，默认情况下，大多数控制器会将缓存策略从回写（write back）改为透写（write through）来禁用写缓存。这可能会导致性能严重下降。很多控制器也会通过一个学习过程周期性地循环电池，在此期间缓存也会被禁用。RAID控制器的管理实用程序应该允许你查看和配置学习周期的计划时间，以免让你措手不及。较新的RAID控制器通过使用闪回缓存（flashback cache）来避免这种情况，该缓存使用NVRAM而不是电池保护缓存来存储未提交的写操作，这避免了整个学习周期的痛苦。

你可能还希望使用设置为透写的缓存策略对系统进行基准测试，这样就知道会发生什么了。首选方法是在低流量时段安排电池学习周期，通常是在晚上或周末。如果性能在任何时候都受到透写的严重影响，还可以在学习周期开始之前手工执行故障切换，切换到另一台服务器。作为最后的手段，你可以通过更改innodb_flush_log_at_trx_commit和sync_binlog变量来重新配置服务器，以降低持久性设置。这将降低透写期间的磁盘使用率，并可能提供可接受的性能；然而，这真的应该作为最后的手段。降低持久性对数据库崩溃期间可能丢失的数据量以及恢复数据的能力有很大影响。

RAID配置和缓存

通常可以通过在机器的启动引导过程中进入其设置工具或从命令提示符运行它来配置RAID控制器本身。尽管大多数控制器都提供了很多选项，但我们应关注的两个选项是阵列的条带块大小和控制器上的缓存（也称为RAID缓存，可以互换使用这两个术语）。

RAID条带块大小

最佳的条带块大小取决于工作负载和硬件。理论上，对于随机I/O来说，拥有较大的块大小是很好的，因为这意味着可以从单个驱动器中满足更多的读取需求。

要了解原因，请考虑工作负载中典型随机I/O操作的大小。如果块的大小设置为尽量大，并且数据不会跨越块之间的边界，那么只需要一个驱动器参与读取。但是，如果块大小小于要读取的数据量，就无法避免在读取过程中涉及多个驱动器。

理论到此为止。实际上，很多RAID控制器不能很好地处理大数据块。例如，控制器可能使用块大小作为其缓存中的缓存单元，这可能导致浪费。控制器还可以匹配块大小、缓存大小和读取单元大小（它在单个操作中读取的数据量）。如果读取单元太大，其缓存的效率可能会降低，最终可能会读取比实际需要更多的数据，即使对于很小的请求也是如此。

也很难知道给定的数据是否会跨越多个驱动器。即使块大小为16KB（与InnoDB的页面大小相匹配），也无法确定所有读取都将在16KB的边界上对齐。文件系统可能会对文件进行分段，它通常会根据文件系统块大小（通常为4K B）对齐这些分段。有些文件系统可能更智能，但你不应该指望它。

RAID缓存

RAID缓存是物理安装在硬件RAID控制器上的（相对）少量内存。当数据在磁盘和主机系统之间传输时，RAID缓存可以用来缓冲数据。以下是RAID卡使用缓存的一些可能原因。

缓存读操作

> 当控制器从硬盘中读取一些数据并将其发送给主机系统后，控制器就可以存储这些数据；这将使控制器能够满足未来对相同数据的请求，而无须再次访问磁盘。
>
> 这通常是非常糟糕的RAID缓存使用方式。为什么？因为操作系统和数据库服务器都有更大的缓存。如果在其中一个缓存中发生缓存命中，则不会使用RAID缓存中的数据。相反，如果某个更高级别的缓存中有一个未命中，那么在RAID缓存中命中的可能性会非常小。因为RAID缓存要小得多，所以几乎可以肯定它也会被刷新并填充其他数据。无论从哪个角度看，在RAID缓存中缓存读取都是浪费内存。

缓存预读取数据

> 如果RAID控制器注意到顺序的数据请求，它可能会决定进行预读取，即预取它预测不久将需要的数据。不过，在收到请求之前，它必须有地方存放数据，可以使用RAID缓存来实现。这对性能的影响可能差别很大，你应该进行检查以确保它确实有帮助。如果数据库服务器正在执行自己的智能预读取（就像InnoDB所做的），RAID的预读操作可能没有帮助，而且可能会干扰最重要的同步写操作的缓冲。

缓存写操作

> RAID控制器可以将写操作缓冲到缓存中，并在以后调度。这样做有两个好处：首先，可以以比实际在物理磁盘上执行写操作更快的速度给主机系统返回“成功”；其次，可以累积写操作并更有效地执行写操作。

内部操作

> 有些RAID操作非常复杂，尤其是RAID 5的写操作，它必须计算奇偶校验位，以便在出现故障时可重建数据。控制器需要为这种内部操作使用一些内存。这就是为什么RAID 5在某些控制器上表现不佳的原因之一：它需要将大量数据读入缓存以获得良好的性能。有些控制器不能很好地平衡缓存写操作和缓存RAID 5奇偶校验操作。

一般来说，RAID控制器的内存是一种稀缺资源，应该明智地使用它。将其用于读操作通常是一种浪费，将其用于写操作是提高I/O性能的一种重要方式。很多控制器允许你选择如何分配内存。例如，可以选择其中有多少用于缓存写操作，有多少用于读操作。对于RAID 0、RAID 1和RAID 10，应该为缓存写操作分配100%的控制器内存。对于RAID 5，应该为其内部操作保留一些控制器内存。这通常是一个很好的建议，但并不总是适用，不

同的RAID卡需要不同的配置。

当使用RAID缓存进行写缓存时，很多控制器都允许配置可接受的写入延迟时间（1秒、5秒等）。更长的延迟意味着更多的写入可以组合在一起，并以最佳方式刷新到磁盘。不利的一面是，写操作将更加“间歇性”，这不是一件坏事，除非应用程序恰好在控制器缓存填满时发出了一堆写请求，而此时RAID缓存正要被刷新到磁盘上。如果没有足够的空间容纳应用程序的写请求，它将不得不等待。保持较短的延迟意味着将有更多的写操作，缓存的效率也会降低，但它会平滑尖峰现象，并有助于保持更多的缓存空闲以处理来自应用程序的突发。（我们在这里做了简化，控制器通常有复杂的、特定于供应商的平衡算法，所以这里只是尝试介绍基本原理。）

写缓存对于同步写操作非常有用，例如，对事务日志发出fsync()调用，并在启用sync_binlog的情况下创建二进制日志，但除非控制器有备用电池单元（BBU）或其他非易失性存储，否则不应启用写缓存。如果断电，在没有BBU的情况下，缓存写操作可能会损坏数据库，甚至会损坏事务文件系统。但是，如果有BBU，启用写缓存可以将执行大量日志刷新（例如，在事务提交时刷新事务日志）的工作负载的性能提高20倍以上。

最后要考虑的是，很多硬盘驱动器都有自己的写缓存，通过欺骗控制器数据已经写入物理媒介，实际执行了“假”fsync()操作。直接连接（而不是连接到RAID控制器）的硬盘驱动器有时可以让操作系统管理它们的缓存，但这也并不总是有效。对于fsync()，这些缓存通常被刷新，对于同步I/O，这些缓存会被绕过，但同样地，硬盘也可能会撒谎。应该确保在fsync()时刷新缓存，或者干脆在没有备用电池时禁用它们。没有被操作系统或RAID固件正确管理的硬盘驱动器导致了许多数据丢失的案例。

由于种种原因，当安装新硬件时，最好进行真正的暴力碰撞测试（例如把电源插头拔出来）。这通常是发现细微的错误配置或偷偷摸摸的硬盘行为的唯一方法。可以在网上找到方便的脚本（参见链接12）。

要测试RAID控制器的BBU是否真的可靠，请确保将电源线拔下一段时间。有些BBU设备在没电的情况下无法维持足够的时间（将RAID缓存的数据刷新到磁盘）。在这里，一个环节出问题可能会导致整个存储系统链变得无用。

网络配置

正如延迟和吞吐量是硬盘驱动器的限制因素一样，延迟和带宽也是网络连接的限制因素。对于大多数应用程序来说，最大的问题是延迟；一个典型的应用程序会进行很多次小的网络传输，每次传输都会有轻微的延迟。

网络运行不正常也是主要的性能瓶颈之一。数据包丢失是一个常见的问题。即使是1%的丢失也足以导致性能显著下降，因为协议栈中的各个层都会尝试通过等待一段时间然后重新发送数据包等策略来解决问题，这会增加额外的响应时间。另一个常见问题是DNS解析异常中断或缓慢。[2]

DNS已经成为一个致命弱点，因此对于生产服务器来说，启用skip_name_resolve是一个好主意。DNS解析中断或缓慢对于许多应用程序来说都是问题，对于MySQL来说尤其严重。当MySQL收到连接请求时，它会同时进行正向和反向DNS查找。可能出错的原因有很多。DNS解析出错会导致连接被拒绝，连接到服务器的速度会变慢，总之它会引起各种问题，严重时可导致拒绝服务攻击。如果启用skip_name_resolve选项，MySQL不会进行任何DNS查找。但是，这也意味着用户账号在host列中只能有IP地址、“localhost”或IP地址通配符。host列中包含主机名的任何用户账号都将无法登录。

不过，通常更重要的是，调整设置以有效地处理大量连接和小查询。一个常见的调整是更改本地端口范围。Linux系统有一系列可以使用的本地端口。当连接返回到调用方法时，需要使用本地端口。如果同时有多个连接，则可能会耗尽本地端口。

下面是一个配置为默认值的系统：

```
$ cat /proc/sys/net/ipv4/ip_local_port_range
32768 61000
```

有时可能需要将这些值更改为更大的范围。例如：

```
$ echo 1024 65535 > /proc/sys/net/ipv4/ip_local_port_range
```

TCP允许系统对接收到的连接请求进行排队，就像一个桶。如果桶被填满，客户端将无法连接。你可以允许更多的连接排队，如下所示：

```
$ echo 4096 > /proc/sys/net/ipv4/tcp_max_syn_backlog
```

对于仅在本地使用的数据库服务器，如果服务器中断后未关闭其连接端时，可以缩短关闭套接字后的超时时间。大多数系统的默认值为1分钟，这相当长：

```
$ echo <value> > /proc/sys/net/ipv4/tcp_fin_timeout
```

大多数时候，可以保留这些设置的默认值。通常只有在发生异常情况时才需要更改它们，例如，极其糟糕的网络性能或大量连接。在互联网上搜索“TCP参数变量”会找到很多关于这些配置的好材料。

选择文件系统

文件系统的选择在很大程度上取决于操作系统。很多系统，比如Windows，实际上只有一两个选择，而且只有一个（NTFS）是真正可行的。GNU/Linux支持很多文件系统。

很多人想知道在GNU/Linux下，哪个文件系统对MySQL性能最好，或者更具体地说，哪个文件系统对InnoDB性能最好。一些基准测试表明，大多数文件系统在很多方面都非常接近，单纯为性能寻找文件系统实际上是一种干扰。文件系统的性能跟特定的工作负载相关，没有哪个文件系统是万能的。大多数时候，给定的文件系统跟其他文件系统相比不会有明显的差别。只有在某些情况下，达到文件系统的处理极限时，例如，需要处理高并发性、处理许多文件、碎片，等等，不同文件系统的差异才会体现出来。

总的来说，最好使用日志型文件系统，如ext4、XFS或ZFS。否则，系统崩溃后，检查文件系统可能需要很长时间。

如果使用ext3或其后续版本ext4，日志级别可设置为3个，你可以在*/etc/fstab*挂载选项中设置。

data=writeback

> 这个选项意味着只记录元数据的写入，这会导致对元数据的写入与对数据的写入不同步。这是最快的配置，通常与InnoDB一起使用是安全的，因为InnoDB有自己的事务日志。例外情况是，在某些特定的时间发生崩溃可能会导致MySQL 8.0之前版本的*frm*文件损坏。
>
> 下面的示例说明了这种配置可能会导致的问题。假设一个程序决定扩展一个文件。元数据（文件的大小）将在数据实际写入文件之前被记录和写入。带来的结果是，文件尾部新扩展区域中包含垃圾数据。

data=ordered

> 这个选项也只记录元数据，但会在写元数据之前写入数据，这能保证一定程度的一致性。它只比writeback选项稍微慢一点，而且在发生崩溃时要安全得多。在这个配置中，如果我们再次假设一个程序想要扩展一个文件，那么在写入新扩展区域中的数据之前，日志中不会记录反映文件新大小的元数据。

data=journal

> 这个选项提供原子日志行为，即在将数据写入最终位置之前将数据先写入日志。此选项通常是不必要的，并且比其他两个选项有更高的开销。然而，在某些情况下，它可以提高性能，因为日志记录允许文件系统延迟对数据最终位置的写操作。

不管使用什么文件系统，都有一些特定的选项是最好禁用的，因为它们没有提供任何好处，并且可能会增加相当多的开销。最著名的是记录访问时间，其在读取文件或目录时也需要写入。要禁用此选项，请将noatime、nodiratime挂载选项添加到*/etc/fstab*；这有时可以将性能提高5%～10%，具体取决于工作负载和文件系统（尽管在其他情况下可能没有多大区别）。下面是我们提到的ext3选项的示例*/etc/fstab*行：

```
/dev/sda2 /usr/lib/mysql ext3 noatime,nodiratime,data=writeback 0 1
```

还可以调整文件系统的预读行为，因为它可能是多余的。例如，InnoDB有自己的预读机制。禁用或限制预读对Solaris的UFS尤其有利。使用innodb_flush_method=O_DIRECT会自动禁用文件系统的预读。

有些文件系统可能不支持你需要的功能。例如，如果使用InnoDB的O_DIRECT刷新方法，就需要文件系统支持直接I/O。此外，有些文件系统可以更好地处理大量底层驱动器；例如，XFS在这方面通常比ext3好得多。最后，如果计划使用逻辑卷管理器（LVM）快照来初始化副本或进行备份，则应验证所选文件系统和LV M版本是否能很好地协同工作。

表4-2总结了一些常见文件系统的特性。

表4-2：常见文件系统的特性

文件系统	操作系统	日志支持	大目录支持
ext3	GNU/Linux	可选	可选 / 部分支持
ext4	GNU/Linux	支持	支持
JFS	GNU/Linux	支持	不支持
NTFS	Windows	支持	支持
ReiserFS	GNU/Linux	支持	支持
UFS(Solaris)	Solaris	支持	可调整
UFS(FreeBSD)	FreeBSD	不支持	可选 / 部分支持
UFS2	FreeBSD	不支持	可选 / 部分支持
XFS	GNU/Linux	支持	支持
ZFS	GNU/Linux、Solaris、FreeBSD	支持	支持

我们通常建议使用XFS文件系统。ext3文件系统有太多太严格的限制，比如每个inode只有一个互斥锁，还有一些不好的行为，比如在fsync()时刷新整个文件系统中的所有脏块，而不是某个文件的脏块。尽管ext4文件系统在特定的内核版本中存在性能瓶颈，但它是一个可以接受的选择，你应该在确认选择之前进行调查。

当考虑数据库所使用的文件系统时，最好考虑它已经使用了多长时间、有多成熟，以及是否在生产环境中被实际验证过。文件系统是在数据库中保证数据完整性的最低层。

选择磁盘队列调度器

在GNU/Linux上，队列调度器决定了对块设备的请求实际发送到底层设备的顺序。默认设置为完全公平排队，即CFQ（Complete Fair Queuing）。这在笔记本电脑和台式机上偶尔使用是可以的，因为它有助于防止I/O饥渴，但对于服务器来说很糟糕。在MySQL的工作负载类型下，CFQ会导致非常糟糕的响应时间，因为会不必要地阻塞队列中的一些请求。

可以使用以下命令查看哪些调度器可用，哪些处于活动状态：

```
$ cat /sys/block/sda/queue/scheduler
noop deadline [cfq]
```

执行上述命令时应该用想查看的磁盘的设备名替换其中的*sda*。在这个示例中，方括号表示该设备正在使用哪个调度器。另外两种选择适用于服务器级硬件，在大多数情况下，它

们的表现差不多。noop调度器适合于在后台执行自身调度器的设备，比如硬件RAID控制器和存储区域网络（SAN），而deadline则适用于RAID控制器和直接连接的磁盘。我们的基准测试显示，这两者之间的差别非常小。最重要的还是不要使用CFQ，它可能会导致严重的性能问题。

内存和交换

给MySQL分配大量内存后，它的表现最好。正如我们在第1章中了解到的，InnoDB使用内存作为缓存来避免磁盘访问。这意味着内存系统的性能会直接影响查询的速度。即使在今天，确保更快的内存访问的最佳方法之一仍然是用外部内存分配器（如tcmalloc或jemalloc）来替换内置的内存分配器（glibc）。大量基准测试[3]表明，与glibc相比，这两种方法都能提高性能并减少内存碎片。

当操作系统因为没有足够的物理内存来容纳虚拟内存而将一些虚拟内存写入磁盘时，就会发生交换。交换对操作系统上运行的进程是透明的。只有操作系统知道特定的虚拟内存地址是在物理内存中还是在磁盘上。

当使用SSD时，性能损失不像使用HDD时那样明显。但仍然应该积极地避免交换，即使只是为了避免不必要的写操作，因为写操作可能会缩短磁盘整体寿命。也可以考虑关闭交换，这会完全消除潜在的交换影响，但在内存耗尽时可能导致进程被终止的情况。

在GNU/Linux上，可以使用*vmstat*来监控交换（我们将在下一节展示一些示例）。你需要查看si和so列中报告的交换I/O活动，而不是swpd列中报告的交换使用情况。swpd列可以显示已加载但未使用的进程，这并不是真正的问题。我们希望si和so列的值为0，它们肯定应该小于每秒10个块。

在极端情况下，过多的内存分配可能会导致操作系统耗尽交换空间。如果发生这种情况，虚拟内存不足可能会导致MySQL崩溃。但即使没有耗尽交换空间，非常活跃的交换也会导致整个操作系统没有响应，甚至无法登录并终止MySQL进程。有时候，当交换空间用完时，Linux内核甚至会完全挂起。我们建议在运行数据库时完全不使用交换空间。磁盘会比RAM慢一个数量级，但这可避免这里提到的所有让人头痛的问题。

在极端的虚拟内存压力下经常发生的另一件事是，OOM Killer进程将启动并终止某些进程，通常会是MySQL，也可能是其他进程，比如被终止的是SSH进程，则会使系统无法通过网络访问。可以通过设置SSH进程的oom_adj或oom_score_adj值来防止这种情况发生。在使用专用数据库服务器时，我们强烈建议识别所有关键进程，如MySQL和SSH，并主动调整OOM Killer分值，以防止这些进程被首先终止。

通过正确配置MySQL缓冲区，可以解决大多数交换问题，但有时操作系统的虚拟内存系统还是会交换MySQL，有时与Linux中NUMA的工作方式[4]有关。这通常发生在操作系统看到大量来自MySQL的I/O时，因此它会尝试增加文件缓存以容纳更多数据。如果没有足够的内存，就必须换出一些东西，而这可能是MySQL本身。一些较旧的Linux内核版本具有会产生相反效果的优先级，在不应该交换的时候进行交换，但在较新的内核中这个问题已经有所缓解。

操作系统通常允许对虚拟内存和I/O进行控制。我们提到了在GNU/Linux上控制它们的几种方法。最基本的方法是将*/proc/sys/vm/swappiness*的值更改为较低的值，例如0或1。这告

诉内核，除非对虚拟内存的需求非常大，否则不要进行交换。例如，下面的代码可以检查当前值：

```
$ cat /proc/sys/vm/swappiness
60
```

交换设置默认为60（范围从0到100）。这个默认值只适用于笔记本电脑，对于服务器来说是一个非常糟糕的设置。对于服务器应设置为0：

```
$ echo 0 > /proc/sys/vm/swappiness
```

另一种选项是更改存储引擎读写数据的方式。例如，设置innodb_flush_method=O_DIRECT可以减轻I/O压力。直接I/O没有被缓存，因此操作系统不会将其视为增加文件缓存大小的原因。该参数仅对InnoDB有效。

还有一个选项是使用MySQL的memlock配置项，它将MySQL锁定在内存中。这将避免交换，但可能很危险：如果没有足够的可锁定内存，MySQL在尝试分配更多内存时可能会崩溃。如果锁定了太多内存，导致操作系统没有足够的内存，也可能会出现问题。

很多技巧都是特定于内核版本的，所以要小心，特别是在升级时。在某些工作负载中，很难让操作系统合理地运行，唯一的办法可能是将缓冲区大小降低到次优值。

操作系统状态

操作系统提供了一些工具来帮助了解操作系统和硬件正在做什么。在本节中，我们将向你展示如何使用两个广泛可用的工具——*iostat*和*vmstat*的示例。如果你的系统不提供这两种工具，很可能会提供类似的工具。因此，我们的目标不是让你成为使用*iostat*或*vmstat*的专家，而是简单地展示当试图使用此类工具诊断问题时应该关注什么。

除了这些工具之外，操作系统还可能提供其他工具，例如*mpstat*或*sar*。如果你对系统的其他部分感兴趣，比如网络，也可以使用*ifconfig*（显示发生了多少网络错误）或*netstat*等工具。

默认情况下，*vmstat*和*iostat*只生成一个报告，显示自服务器启动以来各种计数器的平均值，这不是很有用。但你可以为这两种工具提供一个时间间隔参数，使它们生成增量报告，显示服务器当前正在做什么，这更为相关。（第一行依然会显示自系统启动以来的统计数据，所以可以忽略这一行。）

如何阅读**vmstat**的输出结果

先来看一个*vmstat*的例子。可以使用以下命令，每5秒输出一行新的信息，并以MB为单位展示数据：

```
$ vmstat -SM 5
procs -------memory------- -swap- -----io---- ---system---- ------cpu-----
 r  b swpd free buff cache  si so    bi    bo     in     cs us sy id wa st
11  0    0 2410    4 57223   0  0  9902 35594 122585 150834 10  3 85  1  0
10  2    0 2361    4 57273   0  0 23998 35391 124187 149530 11  3 84  2  0
```

可以使用Ctrl+C组合键停止*vmstat*的输出。具体的输出取决于操作系统，因此你可能需要

阅读操作手册来了解具体的信息。

如前所述，尽管我们要求增量输出，但第一行的值依然显示的是自服务器启动以来的平均值。第二行显示当前正在发生的事情，接下来将以5秒的间隔显示正在发生的事情。列按以下标题之一分组。

procs

r列显示有多少进程在等待CPU时间。b列显示了处于不可中断的休眠状态的进程数，这通常意味着它们正在等待I/O（磁盘、网络、用户输入，等等）。

memory

swpd列显示了有多少个块被交换到磁盘（paged）。剩下的三列分别显示有多少个块是空闲的（unused），有多少个块用于缓冲区（buff），还有多少个块用于操作系统的缓存（cache）。

swap

这些列显示交换活动：操作系统每秒（从磁盘）换入和换出（到磁盘）的块数。它们比swpd列更重要。我们大多数时候都希望看到si和so列的值为0，绝对不希望看到每秒超过10个块。当然这两个值突然增加也是不好的。

io

这些列显示每秒从块设备中读入（b i）和写入块设备（b o）的块数。这通常反映了磁盘I/O。

system

这些列显示每秒中断数（in）和每秒上下文切换数（cs）。

cpu

这些列分别显示运行用户（非内核）代码、运行系统（内核）代码、空闲和等待I/O所花费的CPU总时间百分比。如果使用了虚拟化，可能会有第5列（st），显示从虚拟机“窃取”的百分比。这是指虚拟机上可以运行某些内容，但虚拟机管理程序（hypervisor）却选择运行其他内容的时间。如果虚拟机不想运行任何内容，而虚拟机管理程序运行其他内容，则不算“窃取”时间。

*vmstat*的输出结果是和系统相关的，所以如果你的*vmstat*操作手册与这里展示的示例不同，那么应该阅读系统的*vmstat*操作手册。

如何阅读**iostat**的输出结果

接下来看一下*iostat*。默认情况下，它显示与*vmstat*相同的CPU使用信息。不过，我们通常只对I/O统计数据感兴趣，可以使用以下命令仅显示扩展的设备统计数据：

```
$ iostat -dxk 5
Device: rrqm/s wrqm/s r/s w/s rkB/s wkB/s
sda 0.00 0.00 1060.40 3915.00 8483.20 42395.20

avgrq-sz avgqu-sz await r_await w_await svctm %util
 20.45 3.68 0.74 0.57 0.78 0.20 98.22
```

与*vmstat*一样，第一个报告显示自服务器启动以来的平均值（为了节省空间，例子中通常

会省略掉），随后的报告显示增量平均值。每个设备一行数据。

有各种选项可控制显示或隐藏*iostat*的输出列。官方文档有点混乱，我们必须深入源代码才能找出真正显示的内容。以下是每一列显示的内容。

rrqm/s和wrqm/s

每秒进入队列的合并读写请求数。合并意味着操作系统从队列中获取多个逻辑请求，并将它们组合成对实际设备的单个请求。

r/s和w/s

每秒发送到设备的读写请求数。

rkB/s和wkB/s

以KB为单位每秒读写的吞吐量。

avgrq-sz

以扇区为单位的请求大小。

avgqu-sz

设备队列中等待的请求数。

await

在磁盘队列中花费的毫秒数。

r_await和w_await

向要服务的设备发出读取和写入请求的平均时间（毫秒）。包括请求在队列中等待的时间以及为它们提供服务所花费的时间。

svctm

服务请求所花费的毫秒数，不包括队列时间。

%util [5]

至少有一个请求处于活动状态的时间百分比。这是一个令人困惑的名字。如果你熟悉排队理论中使用率的标准定义，那么这不是设备的使用率。具有多个硬盘驱动器的设备（如RAID控制器）应该能够支持高于1的并发性，但是%util永远不会超过100%，除非用于计算它的数学中存在舍入误差。因此，它并不是一个很好的设备饱和的指示，这与文档中所说的不同，除非是在查看单个物理硬盘驱动器的特殊情况下。

可以使用*iostat*的输出来推断有关机器I/O子系统的一些运行情况。一个重要指标是服务的并发请求数。由于读写速率的时间单位为秒，而服务时间的单位为毫秒，因此可以使用利特尔法则推导出设备正在服务的并发请求数的公式，如下所示：

```
并发度 = (r/s + w/s) * (svctm/1000)
```

将前面的样本数据代入上述公式可以得到大约0.995的并发度。这意味着平均而言，在采样间隔期间，设备每次服务的请求少于一个。

其他有用的工具

前面已经展示了*vmstat*和*iostat*工具，它们是广泛可用的，而且*vmstat*通常默认安装在很多

类UNIX操作系统上。然而，这些工具都有其局限性，例如，度量单位混乱，采样间隔与操作系统更新统计数据的时间间隔不一致，以及无法同时查看所有指标等。如果这些工具无法满足需求，你可能会对*dstat*（参见链接16）或*collectl*（参见链接17）感兴趣。

也可以使用*mpstat*来查看CPU的统计数据，它能输出每个CPU的统计数据，而不是只计算一个汇总数据，这样能更好地观察到每个CPU的运行情况。有时，在诊断问题时这一点非常重要。在检查磁盘I/O使用情况时，你可能会发现*blktrace*很有用。

Percona编写了自己的*iostat*替代品，名为*pt-diskstats*。这是Percona工具箱的一部分。它解决了一些关于*iostat*的抱怨，比如聚合显示读取和写入的方式，以及缺乏并发性的可见性。另外，*pt-diskstats*是交互式和按键驱动的，因此可以放大和缩小、更改聚合、过滤设备以及显示和隐藏列。这是一种对磁盘统计数据进行切片和切分的好方法，即使没有安装该工具，也可以使用简单的shell脚本收集这些数据。你可以捕获磁盘活动的样本数据，通过电子邮件或将其保存以供以后分析。

最后，Linux剖析器*perf*是一个非常有用的工具，可以用它来检查操作在系统级别发生的事情。你可以使用*perf*检查操作系统的常规信息，比如为什么内核如此频繁地使用CPU。还可以检查特定的进程ID，允许你查看MySQL是如何与操作系统交互的。检查系统性能是一个非常深入的研究，因此我们推荐Brendan Gregg（Pearson）编写的《性能之巅：系统、企业与云可观测性》（第2版）作为优秀的后续读物。

小结

为MySQL选择和配置硬件，并为硬件配置MySQL，并不是一门神秘的艺术。一般来说，你需要的技能和知识与大多数其他目的一样。然而，你还是应该知道一些MySQL特有的东西。

我们通常建议在性能和成本之间找到一个良好的平衡。首先，出于许多原因，我们喜欢使用廉价的服务器。例如，如果服务器出现问题，需要在尝试诊断时将其停用，或者如果只是想尝试将其与另一台服务器交换作为诊断的一种形式，那么使用一台5000美元的服务器要比使用一台50000美元或更贵的服务器容易得多。从软件本身和它运行的典型工作负载来看，MySQL通常也更适合运行在廉价硬件上。

MySQL需要的4种基本资源是CPU、内存、磁盘和网络资源。网络通常不会成为严重的瓶颈，但CPU、内存和磁盘经常会成为瓶颈。速度和数量的平衡实际上取决于工作负载，你应该在预算允许的范围内努力实现更快更多的平衡。你期望的并发性越高，就越应该使用更多的CPU来支撑工作负载。

CPU、内存和磁盘之间的关系错综复杂，一个领域的问题往往会在其他领域出现。在你为一个问题投入资源之前，问问自己是否应该为另一个问题投入资源。如果I/O受限，是需要更多的I/O容量，还是只需要更多内存？答案取决于工作集的大小，即在给定的时间内最经常需要的数据集。

固态设备对于提高服务器整体性能非常有用，现在通常应该成为数据库的标准配置，尤其对于OLTP工作负载。只有在预算极为有限的系统中，或者在需要惊人的高达数PB的磁盘空间的数据仓库场景下，才考虑继续使用HDD。

就操作系统而言，只需要正确掌握几件重要的事情，大部分与存储、网络和虚拟内存管理有关。如果你像大多数MySQL用户一样使用GNU/Linux，我们建议使用XFS文件系统，并将交换率和磁盘队列调度器设置为适合于服务器的值。你可能需要更改一些网络参数，并且调整一些其他内容（例如，禁用SELinux），但这些更改只是个人偏好问题。

---

[1] 原文这里是**SSH**，有误，按照上下文应当是指**HDD**。——译者注

[2] 流行俳句：不是**DNS**。不可能是**DNS**。是**DNS**。

[3] 请参阅博客文章**“Impact of Memory Allocators on MySQL Performance”**（参见链接**13**）和**“MySQL（or Percona）Memory Usage Tests”**（参见链接**14**），并进行比较。

[4] 更多信息请参阅博客文章（参见链接**15**）。

[5] 软件**RAID**（如**MD/RAID**）可能不会显示**RAID**阵列本身的使用率。

第5章 优化服务器设置

本章我们将解释如何为MySQL服务器创建合适的配置文件。这是一个迂回的旅程，有许多兴趣点和可以俯瞰风景的短途旅程。这些短途旅程是必要的。确定合适配置的最短路径并不是从研究配置选项并询问应该设置哪些配置或如何更改它们开始的，也不是从检查服务器行为并询问是否有任何配置选项可以改善它开始的。最好从了解MySQL的内部结构和行为开始。然后，你可以使用这些知识作为如何配置MySQL的指南。最后，你可以将期望的配置与当前配置进行比较，并纠正任何重要和值得修改的差异。

人们经常会问："我的服务器有32 GB的RAM和12个CPU核，最佳配置文件是什么？"不幸的是，事情并没有那么简单。你应该根据工作负载、数据和应用程序需求来配置服务器，而不仅仅是根据硬件来配置。MySQL有许多可以更改但不应该更改的设置。通常更好的做法是正确地配置基本设置（在大多数情况下，只有少数设置是重要的），并将更多的时间花在schema优化、索引和查询设计上。在正确设置MySQL的基本配置选项之后，从进一步的更改中获得的潜在收益通常很小。

另外，修改配置的潜在缺点可能是巨大的。MySQL的默认设置是有充分理由的。在不了解其影响的情况下进行更改可能会导致崩溃、卡顿或性能下降。因此，你永远不应该盲目地相信来自热门帮助网站（如MySQL论坛或Stack overflow）的某些人所报告的最佳配置。[1]应该始终通过阅读相关的官方手册来检查任何更改并仔细测试。

那该怎么办？首先应该确保InnoDB缓冲池和日志文件大小等基本设置是合适的。然后，如果你想防止不希望的行为出现，应该设置一些安全选项（但请注意，这些通常不会提高性能，只会避免问题），然后保持其他的设置不变。如果遇到问题，首先要仔细诊断。

如果问题是由服务器的某个部分引起的，而该部分的行为可以通过配置选项进行纠正，那么可能需要对其进行更改。

有时还需要设置特定的配置选项，这些选项在特殊情况下可能会对性能产生重大影响。但是，这些不应该是基本服务器配置文件的一部分。应该只在发现它们解决的特定性能问题时，才设置它们。这就是为什么我们不建议通过寻找不好的方面来改进配置选项的原因。如果需要改进配置，应该会在查询响应时间中体现出来。最好从查询及其响应时间开始分析，而不是从配置选项开始。这可以节省很多时间，避免很多问题。

另一个节省时间和避免麻烦的好方法是使用默认设置，除非你明确知道不应该使用默认设置。很多默认设置都是安全的，很多人都会直接使用。这使默认设置成为测试最彻底的设置。当没必要改变这些设置而改变它们时，可能会引起意想不到的错误。

MySQL的配置是如何工作的

介绍在MySQL中应该配置什么之前，我们将首先解释MySQL的配置机制。MySQL通常对其配置很宽容，但遵循这些建议可以为你节省大量的工作和时间。

首先要知道的是，MySQL从何处获取配置信息：命令行参数和配置文件中的设置项。在类UNIX系统上，配置文件通常位于*/etc/my.cnf*或*/etc/mysql/my.cnf*。如果使用操作系统的启动脚本，这通常是配置设置的唯一位置。如果手动启动MySQL（在运行测试安装时可能会这样做），还可以在命令行上指定设置。服务器实际上会读取配置文件中的内容，删除其中的注释行和换行符，然后将其与命令行选项一起处理。

> 术语说明
>
> 因为MySQL的很多命令行选项和服务器变量是对应的，所以我们有时会交替使用选项（option）和变量（variable）这两个术语。大多数变量与其对应的命令行选项具有相同的名称，但也有一些例外。例如，--memlock命令行选项可以设置locked_in_memory变量。

需要永久使用的任何设置都应该写入全局配置文件，而不是在命令行中指定。否则会有风险，可能会在没有指定命令行选项的情况下意外启动服务器。将所有配置文件保存在一个地方也是一个好主意，这样可以方便地检查它们。

一定要知道服务器的配置文件在哪里！我们已经看到有人尝试用一个不能读取的文件来配置服务器，但没有成功，比如，Debian服务器上默认不存在*/etc/my.cnf*，而是会在*/etc/mysql/my.cnf*中查找配置。有时文件位于多个位置，这可能是因为以前的系统管理员也搞混了。如果不知道服务器会读取哪些文件，可以通过如下命令查询：

```
$ which mysqld
/usr/sbin/mysqld
$ /usr/sbin/mysqld --verbose --help | grep -A 1 'Default options'
Default options are read from the following files in the given order:
/etc/mysql/my.cnf ~/.my.cnf /usr/etc/my.cnf
```

配置文件采用标准INI格式，被分为多个部分，每个部分都以一行包含在方括号中的该部分名称开头。MySQL程序通常会读取与该程序同名的部分，很多客户端程序也会读取client部分，这为你提供了放置公共设置的位置。服务器通常读取mysqld部分。确保将设置放在文件的正确部分，否则它们将不起作用。

语法、作用域和动态性

配置设置全部用小写字母书写，单词之间以下画线或短横线分隔。这两种写法是等效的，你可能会在命令行和配置文件中看到下面的写法：

```
/usr/sbin/mysqld --auto-increment-offset=5
/usr/sbin/mysqld --auto_increment_offset=5
```

我们建议选择一种风格并始终如一地使用它。这使得在文件中搜索设置更容易。

配置设置可以有多个作用域。有些设置是服务器范围的（全局作用域），有些设置对于每个连接都不同（会话作用域），有些设置是基于每个对象的。许多会话作用域的变量都有相应的全局变量，可以将相应的全局变量的值视为会话变量的默认值。如果更改会话作用域的变量，它只会影响更改该变量的连接，在连接关闭时更改将丢失。以下是一些例子，需要注意各个例子中的不同行为：

- max_connections变量是全局作用域的。
- sort_buffer_size变量有一个全局默认值，但是也可以在每个会话中设置。
- join_buffer_size变量具有全局默认值，可以在每个会话中设置，但是多表联接查询可以为每个联接操作分配一个连接缓冲区，所以每个查询可能有多个联接缓冲区。

除了在配置文件中进行设置外，很多变量（但不是全部）还可以在服务器运行时进行更改。MySQL将这些称为动态配置变量。以下语句显示了动态更改sort_buffer_size的会话和全局值的不同方法：

```
SET sort_buffer_size = <value>;
SET GLOBAL sort_buffer_size = <value>;
SET @@sort_buffer_size := <value>;

SET @@session.sort_buffer_size := <value>;
SET @@global.sort_buffer_size := <value>;
```

请注意，动态设置的变量在MySQL重启后会失效。如果要保留设置，必须更新配置文件。

如果在服务器运行时设置变量的全局值，则当前会话和其他现有会话的值将不受影响。如果客户端依赖数据库长连接，请务必记住这一点。这是因为在创建连接时，会话值是从全局值初始化的。在每次更改后应该检查SHOW GLOBAL VARIABLES的输出，以确保其达到预期效果。

还可以使用SET命令为变量指定一个特殊值：关键字DEFAULT。将会话作用域变量设置为DEFAULT会将该变量设置为相应的全局作用域变量的值。如果想将会话作用域的变量重置为创建连接时的值，这么设置非常方便。建议不要将它用于全局变量，因为它可能不会实现想要的效果，也就是说，它不会将值设置为启动服务器时的值，甚至不会设置为配置文件中指定的值，它会将变量设置为默认编译的值。

持久化系统变量

了解所有这些变量作用域和配置方法还不够，你还必须知道，如果重新启动MySQL，即使使用了SET GLOBAL来更改全局变量，它也将恢复到配置文件中的状态。这意味着必须同时管理MySQL的配置文件和运行时配置，并确保它们保持同步。如果要增加服务器的最大连接数max_connections，必须在每个正在运行的实例上执行SET GLOBAL max_connections命令，然后编辑配置文件以反映新配置。

MySQL 8.0引入了一个名为持久化系统变量的新功能（参见链接18），这有助于简化这个问题。新的语法SET PERSIST允许在运行时设置一次值，MySQL将把这个设置写入磁盘，以便在下次重启后继续使用该值。

设置变量的副作用

动态设置变量可能会产生意想不到的副作用，例如，引起缓冲区刷新脏块。在线更改设置时要小心，可能会导致服务器执行大量工作。

有时可以从变量的名称推断变量的行为。例如，max_heap_table_size指定了内存中隐式临时表允许增长到的最大大小。然而，命名约定并不完全一致，因此不能总是通过查看变量名来猜测它会做什么。

让我们看一些常用的变量以及动态更改这些变量的效果。

table_open_cache

: 设置此变量不会立即生效：下一次线程打开表时，MySQL会检查变量的值。如果该值大于缓存中的表的数目，线程可以将新打开的表插入缓存。如果该值小于缓存中的表的数目，MySQL将从缓存中删除未使用的表。

thread_cache_size

: 设置此变量不会立即生效：下一次关闭连接时，MySQL会检查缓存中是否有空间来存储线程。如果有，则缓存线程以供其他连接将来重用。如果没有，则将线程终止而不是缓存它。在这种情况下，缓存中的线程数量，以及线程缓存使用的内存量不会立即减少；只有当一个新连接从缓存中删除一个线程以使用它时，内存量才会降低。（MySQL仅在连接关闭时将线程添加到缓存中，并仅在创建新连接时将其从缓存中删除。）

read_buffer_size

: 只有当查询需要时，MySQL才会为该缓冲区分配内存，而且会立即分配此变量指定的整块内存。

read_rnd_buffer_size

: 在查询需要之前，MySQL不会为该缓冲区分配任何内存。即时查询需要也只会分配所需要的内存（max_read_rnd_buffer_size这样的名字才能更准确地描述该变量）。

MySQL的官方文档详细解释了这些变量的作用，但并不是一个详尽的列表。我们在这里的目标只是展示更改一些常见变量时会出现什么行为。

除非知道这样做是正确的，否则不应该全局地提高每个连接设置的值。有些缓冲区是一次性分配的，即使它们并不需要，所以一个大的全局设置可能是一个巨大的浪费。相反，可以在查询需要时提高该值。

规划变量的更改

设置变量时要小心。并不总是越多越好，如果将值设置得太高，则很容易导致问题：可能会耗尽内存或导致服务器使用交换区。

回到第2章，监控你的SLO，以确保更改不会影响客户体验。只做基准测试是不够的，因为这不是真实的生产环境。如果你不测量服务器的实际性能，可能会在不知不觉中损害性能。我们看到过很多情况，有人改变了服务器的配置，并认为它提高了性能，而实际上，由于一天或一周中不同时间的不同工作负载，服务器的性能总体上有所下降。

理想情况下，应该使用版本控制系统来跟踪配置文件的更改。这种策略可以非常有效地将性能变化或SLO不达标与特定配置更改关联起来。请注意，在默认情况下，只更改配置文件实际上不会做任何事情，还必须更改运行时设置。

在开始更改配置之前，应该优化查询和schema，至少解决一些显而易见的问题，比如添加索引。如果深入调整了配置，但之后又更改了查询或schema，则可能需要重新评估配置。请记住，除非你的硬件、工作负载和数据完全是静态的，否则需要在变化后重新审视配置。事实上，大多数人的服务器在一天中甚至没有稳定的工作负载，这意味着早上八九点的“完美”配置可能不适合下午三点时的情况！显然，追求神话般的“完美”配置是完全不切实际的。因此，不需要去榨干服务器的每一点性能；事实上，这种时间投资的回报可能很小。我们建议专注于优化峰值工作负载，然后在“足够好”的时候就可以停止优化，除非你有理由相信正在放弃显著的性能改进。

什么不该做

在开始服务器配置之前，我们建议你避免一些常见的做法，这些做法有风险，或者实际上不值得这么做。警告：前方有咆哮声！

你可能会被期望（或相信你会被期望）建立一个基准测试套件，并通过迭代修改配置来“调优”服务器，以寻找最佳设置。我们通常不建议大家去做这样的事情。它需要大量的工作和研究，而且大多时候，潜在的回报非常小，因此是巨大的时间浪费。最好把这些时间花在其他事情上，比如检查备份、监控查询计划的变化等。

你不应该“按比率调优”。经典的“调优比率”是一个经验法则，比如，InnoDB缓冲池命中率应该高于某个百分比，如果命中率过低，应该增加缓存大小。这是非常错误的建议。不管别人怎么说，缓存命中率与缓存是太大还是太小无关。首先，命中率取决于工作负载——不管缓存有多大，有些工作负载根本不能被缓存——其次，缓存命中是没有意义的，原因我们将在后面解释。有时会出现缓存太小，出现命中率很低的情况，增加缓存大小会增加命中率。然而，这是一种偶然的关联，并不表示缓存的性能或大小是适当的。

相关性有时看似正确，但问题在于人们容易相信它们永远是正确的。Oracle DBA在几年前就放弃了基于比率的调优，我们希望MySQL DBA也能跟随他们的脚步。[2]我们甚至更强烈地希望人们不要编写“调优脚本”来将这些危险的实践编入法典，并将它们传授给成千上万的人。这就引出了我们“什么不该做”的下一个建议：不要使用调优脚本！可以在网上找到一些非常流行的脚本，但最好还是忽略它们。

我们还建议你避免使用调优这个词，在前几段文字中大量使用了这个词。我们更倾向于用配置或优化（只要这是你实际在做的）。一提到调优，人们就会联想到一名不守纪律的新手，他调整了服务器，然后看看会发生什么。我们在前一节中建议，这个实践最好留给那些研究服务器内部结构的人。“调优”服务器可能是一种惊人的时间浪费。

在互联网上搜索配置建议并不总是一个好主意，你会在博客、论坛等找到很多糟糕的建议。尽管许多专家在网上贡献了他们所知道的，但很难判断谁是真正的专家。当然，对于去哪里找真正的专家，我们无法给出公正的建议。但是我们可以说，可靠的、有信誉的MySQL服务提供商通常比简单的互联网搜索结果更安全，因为那些需要拥有满意的客户的人可能正在做正确的事情。然而，即使是他们的建议，在没有经过测试和理解的情况下进行应用也可能是危险的，因为它可能针对的是一种与你不同的情况，而你却没有理解。

最后，不要相信流行的内存消耗公式——是的，MySQL本身在崩溃时会输出的那个公式。（这里我们就不重复了。）这个公式是很早之前的。这不是一个可靠的，甚至不是一个有用的方法来了解MySQL在最坏的情况下可以使用多少内存。你也可以在网上看到这个公式的一些变体。这些公式都有类似的缺陷，尽管它们添加了更多原公式所没有的因素。事实是，你不能给MySQL的内存消耗设定上限。MySQL并不是一个严格控制内存分配的数据库服务器。

创建**MySQL**配置文件

正如我们在本章开头提到的，没有一个适合所有人的“最佳配置文件”，比如一个针对4 CPU、16 GB内存和12块硬盘驱动器的服务器配置。你确实需要开发自己的配置，因为根据你使用服务器的方式，即使是一个良好的基础配置也会有很大的差异。

最小化配置

我们为本书创建了一个最小化的示例配置文件，你可以将其用作自己服务器的良好起点。[3]你必须为一些设置选择值，我们将在本章后面对此进行解释。我们的基础文件是围绕MySQL 8.0构建的，如下所示：

```
[mysqld]
# GENERAL
datadir                                 = /var/lib/mysql
socket                                  = /var/lib/mysql/mysql.sock
pid_file                                = /var/lib/mysql/mysql.pid

user                                    = mysql
port                                    = 3306
# INNODB
innodb_buffer_pool_size                 = <value>
innodb_log_file_size                    = <value>
innodb_file_per_table                   = 1
innodb_flush_method                     = O_DIRECT
# LOGGING
log_error                               = /var/lib/mysql/mysql-error.log
log_slow_queries                        = /var/lib/mysql/mysql-slow.log
# OTHER
tmp_table_size                          = 32M
max_heap_table_size                     = 32M
max_connections                         = <value>
thread_cache_size                       = <value>
table_open_cache                        = <value>
open_files_limit                        = 65535
[client]
socket                                  = /var/lib/mysql/mysql.sock
port                                    = 3306
```

与经常看到的配置文件相比，这可能看起来内容太少了，但实际上已经超过了很多人的需要。你可能还会使用一些其他类型的配置选项，例如，二进制日志记录，我们将在本章的后面及其他章节中介绍这些内容。

我们配置的第一个变量是数据存放的位置。我们将其设置为*/var/lib/mysql*，因为它在许多UNIX变体中都很流行。选择另外的位置也没有问题，这由你决定。我们把*.pid*文件也放置在相同的位置，但很多操作系统默认将其放在*/var/run*，这也没有问题。我们只需要为这些设置配置一些东西。顺便说一句，不要让socket和*.pid*文件使用服务器编译的默认

值；在不同的MySQL版本中有一些bug，使用默认值可能会导致问题，所以最好显式设置。（我们不是建议选择不同的地点，只是建议确保*my.cnf*文件明确地设置了这些位置，这样当升级服务器时它们就不会改变或引起问题。）

我们还指定了在操作系统上以*mysql*账号运行*mysqld*。你需要确保这个账号存在，并且确保数据目录和其中的所有文件的属主是这个账号。端口设置为默认的3306，但有时也需要更改。

在MySQL 8.0中引入了一个新的配置选项，innodb_dedicated_server。这个选项检查服务器上可用的内存，并为专用的数据库服务器配置了4个额外的变量（innodb_buffer_pool_size、innodb_log_file_size、innodb_log_files_in_group和innodb_flush_method），从而简化这些值的计算和更改。这在云环境中特别有用，在云环境中，你可以运行一个128G B内存的虚拟机（V M），然后重新启动以扩展到256G B内存。这种情况下MySQL是自配置的，不需要管理配置文件中的值的修改。这通常是管理这4个设置的最佳方式。

在我们的示例文件中，大多数其他设置都是不言自明的，其中许多都是判断问题。我们将在本章的其余部分探讨其中的几个。我们还将在本章的后面讨论一些安全设置，这些设置有助于提高服务器的健壮性，并有助于防止错误数据和其他问题。这里我们先暂且略过。

还要解释的一个设置是open_files_limit选项。在典型的Linux系统中，我们将其设置得尽可能大。在现代操作系统中，打开文件句柄的成本很低。如果这个设置不够大，就会看到经典的24号错误，“too many open files”。

跳到最后，配置文件中的最后一部分是针对*mysql*和*mysqladmin*等客户端程序的，让它们知道如何连接到服务器。应该为客户端程序设置与服务器相匹配的值。

检查**MySQL**服务器的状态变量

有时，你可以使用SHOW GLOBAL STATUS的输出作为配置的输入，以更好地为工作负载定制设置。为了获得最佳结果，请查看绝对值以及值如何随时间变化，最好在峰值和非峰值时间做几次快照。可以使用以下命令查看状态变量每60秒的增量变化：

```
$ mysqladmin extended-status -ri60
```

在解释各种配置设置时，我们会经常提到状态变量随时间的变化。我们通常希望你检查一个命令的输出，比如刚才展示的命令。其他一些有用的工具如Percona Toolkit的*pt-mext*或*pt-mysql-summary*可以提供状态计数器变化的紧凑显示。

我们已经向你展示了一些初步内容，接下来将带你查看一些服务器内部组件，并附带一些配置建议。这将为你提供必要的背景知识，以便为配置选项选择适当的值，稍后将返回示例配置文件进行配置。

配置内存使用

使用innodb_dedicated_server通常会占用50%～75%的内存。这样，至少有25%的内存可用于每个连接的内存分配、操作系统开销和其他内存设置。我们将在下面逐一介绍这些内容，然后更详细地了解各种MySQL缓存的需求。

每个连接的内存需求

MySQL只需要少量的内存就能保持一个连接（通常是一个相关的专用线程）打开。它还需要基本内存量来执行任何给定的查询。你需要为MySQL留出足够的内存，以便在负载高峰期执行查询，否则查询将因内存不足而无法正常运行或失败。

了解MySQL在峰值使用期间将消耗多少内存是很有用的，但是一些使用模式可能会意外地消耗大量内存，这使得内存消耗很难预测。预处理语句就是一个例子，因为可以同时打开多个预处理语句。另一个例子是InnoDB数据字典（稍后将详细介绍）。

在试图预测内存消耗峰值时，不需要假设最坏的情况。例如，如果将MySQL配置为最多允许100个连接，理论上可以同时在所有100个连接上运行大型查询，但实际上这不太可能会发生。使用许多大型临时表或复杂存储过程的查询最有可能占用大量内存。

为操作系统保留内存

与查询一样，需要为操作系统保留足够的内存以完成其工作。这包括运行任何本地监控软件、配置管理工具、计划作业等。操作系统有足够内存的最佳判断依据是，它没有主动将虚拟内存交换（分页）到磁盘。

InnoDB缓冲池

InnoDB缓冲池需要的内存比其他任何组件都多，就性能而言，InnoDB缓冲池大小通常是最重要的变量。InnoDB缓冲池不仅缓存索引，还缓存行数据、自适应哈希索引、更改缓冲区、锁和其他内部结构等。InnoDB还使用缓冲池来实现延迟写操作，从而可以将多个写操作合并在一起并按顺序执行。简而言之，InnoDB严重依赖缓冲池，应该确保为其分配足够的内存。你可以使用SHOW命令或*innotop*等工具中的变量来监控InnoDB缓冲池的内存使用情况。

如果数据不多，并且不会快速增长，那么也不需要将内存过度分配给缓冲池。让缓冲池大大超过其所容纳的表和索引的大小并不会有额外的好处。当然，为一个快速增长的数据库提前规划，设置更大的缓冲池并没有错，但有时我们会看到巨大的缓冲池中只有少量数据，这是没有必要的。

大型缓冲池会带来一些挑战，比如更长的关闭时间和预热时间。如果缓冲池中有很多脏（修改过的）页，InnoDB可能需要很长时间才能关闭，因为它会在关闭时将脏页写到数据文件中。当然也可以强制快速关闭，但在重新启动时，InnoDB需要做更多的恢复工作，因此实际上不能加快关闭和重新启动周期时间。如果提前知道什么时候需要关闭，可以在运行时将innodb_max_dirty_pages_pct变量更改为较低的值，等待刷新线程清理缓冲

池，然后在脏页数量变少时关闭。可以通过监控innodb_buffer_pool_pages_dirty服务器状态变量或使用*innotop*监控SHOW INNODB STATUS来查看脏页数量。还可以使用变量innodb_fast_shutdown来调整关闭InnoDB的方式。

降低innodb_max_dirty_pages_pct变量的值并不能保证InnoDB在缓冲池中保留更少的脏页。相反，它控制的是InnoDB停止“lazy”行为的阈值。InnoDB默认使用同一个后台线程来刷新脏页，以及合并写操作并按顺序执行以提高效率。这种行为被称为“lazy”，因为它允许InnoDB延迟刷新缓冲池中的脏页，除非需要为其他数据提供空间。当脏页的百分比超过阈值时，InnoDB会尽可能快地刷新页面，以尽量降低脏页的数量。与之前的行为相比，这些页面清理操作已经得到了极大的优化（参见链接19），包括能够配置多个线程来执行刷新。

当MySQL再次启动时，缓冲池缓存是空的，也称为冷缓存。在内存中保存行和页的所有好处现在都没有了。值得庆幸的是，默认情况下，innodb_buffer_pool_dump_at_shutdown和innodb_buffer_pool_load_at_startup这两个配置可以配合使用，以在启动时预热缓存池。启动时的加载需要时间，但它可以比等待服务器自然填充缓冲池更快地提高性能。

线程缓存

线程缓存保存了当前没有与连接关联但已准备好为新连接提供服务的线程。创建新连接时，如果缓存中有一个线程，MySQL会从缓存中取出该线程并将其提供给新连接。当连接关闭时，如果缓存中还有空间，MySQL会将线程放回缓存中。如果缓存中已经没有空间，MySQL会销毁线程。只要MySQL在缓存中有空闲线程，它就可以快速响应连接请求，因为不必为每个新连接创建新线程。

变量thread_cache_size指定了MySQL可以保存在缓存中的线程数。其默认值为-1或auto-sized，通常不需要更改这个变量，除非服务器会收到很多连接请求。要检查线程缓存是否足够大，请查看Threads_created状态变量。应该尽量保持线程缓存足够大，以使每秒创建的新线程数少于10个，但通常很容易使这个数字低于每秒1个。

一个好的方法是观察Threads_connected变量，并尝试将thread_cache_size设置得足够大，以处理工作负载中的典型波动。例如，如果Threads_connected通常保持在100到120之间，那么可以将缓存大小设置为20。如果停留在500到700之间，那么将缓存大小设置为200也足够大了。可以这样想：当同时有700个连接时，缓存中线程全部用光，当只有500个连接时，将有200个缓存线程作为备用，即便工作负载随后再增加到700也够用了。

对于大多数使用场景来说，不需要将线程缓存设置得非常大，但是将线程缓存设置得很小也不会节省太多内存，所以这样做没有什么好处。每个处于线程缓存或休眠状态的线程通常使用大约256KB内存。这与活动连接在处理查询时线程可以使用的内存量相比并不多。通常应该保持线程缓存足够大，这样Threads_created就不会经常增加。但是，如果这是一个非常大的数字（例如，成千上万个线程），可能需要将其设置得更低一些，因为一些操作系统不能很好地处理大量线程，即使这些线程大部分处于休眠状态。

配置MySQL的I/O行为

一些配置选项会影响MySQL将数据同步到磁盘和执行恢复的方式。这会涉及I/O操作，因此会极大地影响性能。这些选项还代表了性能和数据安全之间的权衡。一般来说，确保数据立即且一致地写入磁盘的代价是很高的。如果愿意冒磁盘写入操作没有真正写入持久存储的风险，是可以增加并发性和/或减少I/O等待的，但你必须自己决定可以承受多大风险。

InnoDB不仅允许你控制其恢复方式，还允许控制其打开和刷新数据的方式，这将极大地影响恢复和总体性能。尽管可以通过配置影响其采取的行动，但InnoDB的恢复过程是自动的，并且总是在InnoDB启动时运行。撇开恢复不谈，假设没有任何崩溃或出错，InnoDB还有很多需要配置的地方。它有复杂的缓冲区和文件的链，旨在提高性能和保证ACID属性，链的每个部分都是可配置的。图5-1显示了这些文件和缓冲区。

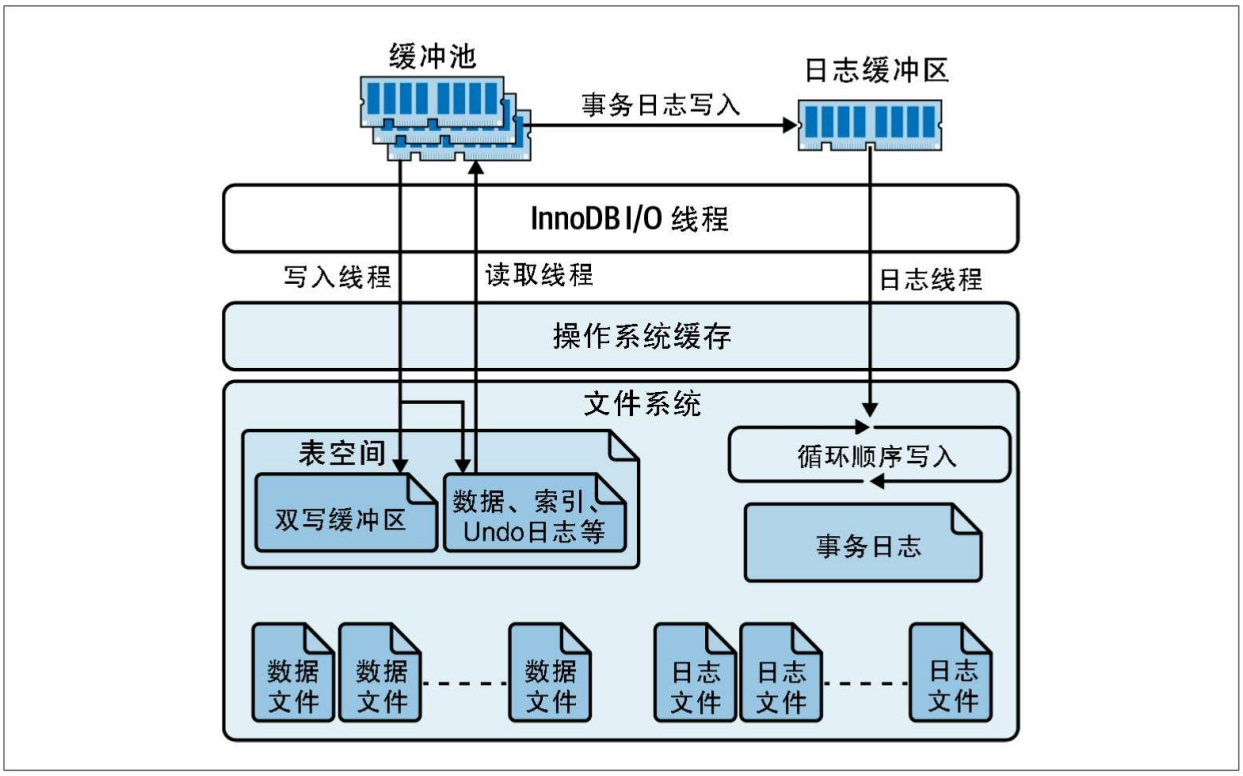

图5-1：InnoDB的缓冲区和文件

为了正常使用，需要更改的几个最重要的参数包括InnoDB日志文件大小、InnoDB如何刷新其日志缓冲区，以及InnoDB如何执行I/O。

InnoDB事务日志

InnoDB使用日志来降低提交事务的成本。它不会在每个事务提交时将缓冲池刷新到磁盘，而是将事务记录到日志中。事务对数据和索引所做的更改通常映射到表空间中的随机位置，因此将这些更改刷新到磁盘将需要随机I/O。InnoDB假定它使用的是传统的磁盘，

随机I/O比顺序I/O的开销要大很多，因为随机I/O需要在磁盘上寻找正确的位置，并等待将所需的磁盘部分旋转到磁头下。

使用日志，InnoDB可以将随机磁盘I/O转换为顺序I/O。一旦日志被安全地保存在磁盘中，即使更改的数据尚未写入数据文件，事务仍将是持久的。如果发生故障（例如停电），InnoDB可以重放日志并恢复已提交的事务。

当然，InnoDB最终必须将更改的数据写入数据文件，因为日志的大小固定，采取的是循环写入的方式：当到达日志的末尾时，它会环绕到日志的开头。如果日志记录中包含的更改尚未应用于数据文件，则无法覆盖日志记录，因为这将删除已提交事务的唯一永久记录。

InnoDB使用后台线程智能地刷新对数据文件的更改。该线程可以将写入分组，并使数据写入顺序化，以提高效率。实际上，事务日志可以将随机数据文件I/O转换为顺序日志文件I/O和顺序数据文件I/O。将刷新移到后台可以更快地完成查询，并有助于缓冲I/O系统的查询负载峰值。

日志文件的总大小由innodb_log_file_size和innodb_log_files_in_group控制，这对写入性能非常重要。如果你采纳了我们之前的建议，使用innodb_dedicated_server，日志文件的大小将根据系统内存量来自动管理。

日志缓冲区

InnoDB修改数据时会将修改记录写入日志缓冲区，并将其保存在内存中。当缓冲区满了、事务提交时，或者每秒1次（这三个条件以先满足者为准），InnoDB会将缓冲区刷新到磁盘上的日志文件中。如果有大型事务，增加缓冲区大小（默认为1MB）有助于减少I/O。控制缓冲区大小的变量是innodb_log_buffer_size。

通常不需要将缓冲区设置得太大。建议的范围是1～8M B，一般来说足够了，除非写入很多大的BLOB记录。与InnoDB的普通数据相比，日志条目非常紧凑。它们不是基于页面的，所以不会浪费空间一次存储整个页面。InnoDB也会让日志条目尽量短，有时甚至只用几个整数来表示记录的操作类型和该操作所需的任何参数！

InnoDB如何刷新日志缓冲区

当InnoDB将日志缓冲区刷新到磁盘上的日志文件时，会使用互斥锁锁定缓冲区，将其刷新到所需的位置，然后将剩余的条目移动到缓冲区的前面。当释放互斥锁时，可能会有多个事务准备刷新其日志条目。InnoDB使用了一个组提交特性，可以在单次I/O操作中将一组日志全部提交。

日志缓冲区必须被刷新到持久存储中，以确保提交的事务完全持久。如果你更关心性能而不是持久性，可以更改innodb_flush_log_at_trx_commit来控制日志缓冲区的刷新位置和刷新频率。

可能的设置如下。

0

每秒定时将日志缓冲区写入日志文件，并刷新日志文件，但在事务提交时不做任何操作。

1

每次事务提交时，将日志缓冲区写入日志文件，并将其刷新到持久存储中。这是默认的（也是最安全的）设置；它保证你不会丢失任何已提交的事务，除非磁盘或操作系统“假装”进行刷新操作（没有将数据真正写入磁盘）。

2

每次事务提交时都将日志缓冲区写入日志文件，但不执行刷新。InnoDB按计划每秒刷新1次。与0设置最重要的区别是，如果只是MySQL进程崩溃，设置为2不会丢失任何事务。但是，如果整个服务器崩溃或断电，仍然可能丢失事务。

了解将日志缓冲区写入日志文件和将日志刷新到持久存储之间的区别很重要。在大多数操作系统中，将缓冲区写入日志只是将数据从InnoDB的内存缓冲区移动到操作系统的缓存中，依然还是在内存中。它实际上不会将数据写入持久存储。因此，如果发生崩溃或断电，设置为0和2通常会导致最多1秒的数据丢失，因为数据可能只存在于操作系统的缓存中。我们之所以说“通常”，是因为InnoDB会以每秒1次的速度将日志文件刷新到磁盘上，但在某些情况下，例如刷新暂停时，可能会丢失超过1秒的事务。

有时，硬盘控制器或操作系统通过将数据放入另一个缓存（如硬盘自身的缓存）来“假装”进行刷新。这样做速度会更快，但非常危险，如果驱动器断电，数据仍可能丢失。这甚至比将innodb_flush_log_at_trx_commit设置为1之外的其他值更糟糕，可能导致数据损坏，而不仅仅是事务丢失。

将innodb_flush_log_at_trx_commit设置为1之外的其他值有可能导致事务丢失。然而，如果不担心持久性（ACID中的D），你会发现其他设置也很有用。也许你只是想要InnoDB的一些其他功能，比如聚簇索引、防止数据损坏和行级锁定。

高性能事务需求的最佳配置是将innodb_flush_log_at_trx_commit设置为1，并将日志文件放在具有备用电池的写缓存和SSD的RAID卷上，这既安全又非常快。事实上，我们敢说，任何需要处理重要工作负载的生产数据库服务器都需要这种硬件。

InnoDB如何打开和刷新日志文件和数据文件

innodb_flush_method选项允许配置InnoDB与文件系统的实际交互方式。这个名字有一点误导，实际上该选项还会影响InnoDB读取数据的方式，而不仅仅是写入数据的方式。

改变InnoDB执行I/O操作的方式会极大地影响性能，所以在改变任何东西之前，一定要理解你在做什么！

这是一个有点令人困惑的选项，因为它同时影响日志文件和数据文件，而且有时对每种文件执行不同的操作。最好为日志文件和数据文件分别提供一个配置选项，但目前是组合在一起的。

如果你使用的是类UNIX操作系统，并且RAID控制器有备用电池的写缓存，我们建议使用O_DIRECT。如果不是，则default或O_DIRECT都可能是最佳选择，具体取决于应用程序。如果你选择使用innodb_dedicated_server，正如我们前面提到的，这个选项会自动设置。

InnoDB表空间

InnoDB将数据保存在表空间中，表空间本质上是一个虚拟文件系统，由磁盘上的一个或多个文件组成。InnoDB将表空间用于多种用途，而不仅仅是存储表和索引。表空间中还包含了Undo日志（重新创建旧行版本所需的信息）、修改缓冲区、双写缓冲区和其他内部结构。

配置表空间

可以使用innodb_data_file_path配置选项指定表空间文件。这些文件都包含在innodb_data_home_dir指定的目录中。下面是一个例子：

```
innodb_data_home_dir = /var/lib/mysql/
innodb_data_file_path = ibdata1:1G;ibdata2:1G;ibdata3:1G
```

上述配置将创建一个3GB的表空间，跨越了3个文件。有时人们想知道是否可以使用多个文件来将负载分散到多个驱动器上，例如：

```
innodb_data_file_path = /disk1/ibdata1:1G;/disk2/ibdata2:1G;...
```

虽然这确实将文件放在了不同的目录中，并且在这个例子中，这些目录还在不同的磁盘上，但InnoDB会将这些文件端到端串起来使用，因此通常不会获得太多性能提升。InnoDB会先填满第一个文件，然后当第一个文件填满时再填第二个，以此类推；负载并没有以更高性能所需的方式分散。RAID控制器是分散负载的一种更智能的方式。

如果允许表空间在空间不足时增长，可以按如下方式自动扩展最后一个文件：

```
...ibdata3:1G:autoextend
```

默认行为是创建一个10M B的自动扩展文件。如果让文件自动扩展，最好对表空间的大小设置一个上限，以防止其变得过大，因为一旦增长，就无法再收缩。例如，以下配置会将自动扩展文件限制为2 GB：

```
...ibdata3:1G:autoextend:max:2G
```

管理单个表空间可能很麻烦，特别是当它自动扩展而你又想回收空间时（出于这个原因，我们建议禁用自动扩展特性，或者至少为空间设置一个合理的上限）。回收空间的唯一方法是将数据导出，然后关闭MySQL并删除所有文件，再修改配置，重启，让InnoDB创建新的空文件，最后再恢复数据。InnoDB对表空间是完全不宽容的：你不能简单地删除文件或者改变其大小。如果损坏了表空间，InnoDB将无法启动。同样，InnoDB对日志文件也非常严格。如果你习惯像使用MyISAM那样随意地移动文件，请务必注意！

innodb_file_per_table选项允许你将InnoDB配置为每个表使用单独的文件。它将数据存储在数据库目录下的*tablename.ibd*文件中。这使得删除表时更容易回收空间。然而，将数据放在多个文件中实际上会导致更多的空间浪费，因为跟InnoDB单个表空间中的内部碎片相比，每个*.ibd*文件中都会有一些浪费的空间。

即使启用innodb_file_per_table选项，你也仍然需要用于Undo日志和其他系统数据的主表空间文件。如果不把数据都存储在里面，主表空间文件会更小。

有些人喜欢使用innodb_file_per_table，因为它提供了额外的可管理性和可视性。例如，通过检查单个文件来查找表的大小要比使用SHOW TABLE STATUS快得多，而SHOW TABLE STATUS必须执行更复杂的工作来确定为一个表分配了空间。

innodb_file_per_table也有不好的一面：会使DROP TABLE性能变差。严重时可能导致服务器范围内明显的停顿，原因有二。

删除表将在文件系统级别解除（删除）文件的链接，这在某些文件系统（ext3）中可能非常慢。可以使用文件系统中的技巧来缩短这个过程的持续时间：先将*.ibd*文件链接到一个大小为零的文件，然后手动删除该文件，而不是等待MySQL来删除。

当启用该选项时，每个表在InnoDB中都有自己的表空间。事实证明，删除表空间需要InnoDB在查找属于该表空间的页面时锁定和扫描缓冲池，这在服务器的缓冲池很大时是非常慢的。如果使用innodb_buffer_pool_instances将缓冲池分解为多个部分，这一点会有所改善。

MySQL的各种版本对此进行了一些修复。从8.0.23版本开始，这应该不再是一个问题了。

最后的建议是什么？我们建议使用innodb_file_per_table并限制共享表空间的大小，这样会使你的生活更轻松。如前所述，如果你遇到任何使此过程痛苦的情况，请考虑我们建议的修复方法。

行的旧版本和表空间

InnoDB的表空间在写操作频繁的环境中可能会变得非常大。如果事务长时间保持打开状态（即使没有做任何工作），并且使用默认的可重复读取事务隔离级别，InnoDB将无法删除行的旧版本，因为未提交的事务仍需要能够看到它们。InnoDB将旧版本存储在表空间中，因此随着更多数据的更新，它将继续增长。清除过程是多线程的，但如果遇到清除延迟问题（innodb_purge_threads和innodb_purge_batch_size），则可能需要针对工作负载进行调优。

SHOW INNODB STATUS可以帮助定位问题。可以查看TRANSACTIONS部分中的历史列表长度（History list length），其显示了Undo日志的大小：

```
------------
TRANSACTIONS
------------
Trx id counter 1081043769321
Purge done for trx's n:o < 1081041974531 undo n:o < 0 state: running but idle
History list length 697068
```

如果Undo日志很大，并且表空间因此而增长，你可以强制MySQL放慢速度来让InnoDB的清理线程跟上。这听起来可能不太吸引人，但别无选择。否则，InnoDB会不断地写入数据并填充磁盘，直到磁盘空间耗尽或者表空间达到所定义的上限。

要限制写操作，请将innodb_max_purge_lag变量设置为0以外的值。这表示在InnoDB开始

延迟更多修改数据的查询之前，可以等待清除的最大事务数。你必须了解你的工作负载，才能决定合适的值。例如，如果平均事务影响1KB的行，并且可以在表空间中容忍100 MB未清除的行，那么可以将该值设置为100000。

请记住，未清除的行版本会影响所有查询，因为它们会使表和索引变大。如果清除线程不能跟上进度，性能就会下降。设置innodb_max_purge_lag变量也会降低性能，但这是两害相权取其轻。

其他**I/O**配置选项

sync_binlog选项控制MySQL如何将二进制日志刷新到磁盘，默认值是1，意味着MySQL将执行刷新并保持二进制日志的持久性和安全性。强烈推荐将其设置为1，不建议设置为任何其他值。

如果不将sync_binlog设置为1，发生崩溃时可能会导致二进制日志与事务数据不同步。这很容易破坏复制且不可恢复，尤其是当数据库使用全局事务ID时（更多信息，请参阅第9章）。将其设置为1所提供的安全性远远超过由此产生的I/O性能损失。

我们在第4章中深入地讨论了RAID，在这里值得重复的是，高质量的RAID控制器、使用有备用电池的写缓存、同时使用回写（write backup）策略，可以每秒处理数千次写操作，并仍然能提供持久的存储。数据被写入有电池的快速缓存，即使系统断电，也能被保存下来。当电源恢复时，RAID控制器将数据从缓存写入硬盘后再使硬盘可用。因此，具有足够大备用电池的写缓存的优秀RAID控制器可以显著提高性能，是一项非常好的投资。当然，固态存储也是目前推荐的解决方案，可以极大地提高I/O性能。

配置**MySQL**并发

当在高并发工作负载下运行MySQL时，可能会遇到其他情况下不会遇到的瓶颈。本节将解释如何检测这些问题，以及如何在高并发工作负载下获得最佳性能。

如果遇到InnoDB并发问题，并且运行的MySQL版本低于5.7，解决方案通常是升级服务器。旧版本仍然面临许多高并发可伸缩性的挑战。所有的东西都在诸如缓冲池互斥锁之类的全局互斥锁上排队，导致服务器几乎停止运行。如果升级到较新版本的MySQL，在大多数情况下不需要限制并发性。

如果你发现自己遇到了并发性瓶颈，最好的选择是对数据进行分片。如果分片不可行，那么可能需要限制并发性。InnoDB有自己的“线程调度器”，它控制线程如何进入内核访问数据，以及进入内核后可以做什么。限制并发性的最基本方法是使用innodb_thread_concurrency变量，该变量限制了内核中同时可以有多少线程。值为0表示对线程的数量没有限制。如果是在老版本的MySQL中遇到InnoDB并发问题，这个变量是最重要的配置变量。

MySQL的在线文档（参见链接20）提供了最佳的配置指南。你必须通过实验来找到适合系统的最佳值，我们建议首先将innodb_thread_concurrency设置为与可用CPU核数相同的值，然后根据需要调整大小。

如果内核中已经有超过允许数量的线程，则新的线程不能进入内核。InnoDB使用一个两阶段的过程来尝试让线程尽可能高效地进入内核。两阶段策略减少了操作系统调度器导致的上下文切换开销。线程首先休眠innodb_thread_sleep_delay指定的微秒数，然后再重试。如果仍然不能进入，它将进入一个等待线程队列，将控制权交给操作系统。

第一阶段的默认睡眠时间为10000微秒。在高并发性环境中，当CPU未充分利用且许多线程处于“进入队列前的睡眠”状态时，更改此值会有所帮助。如果有很多小查询，默认值也可能太大，因为这会增加查询延迟。

一旦线程进入内核，InnoDB就有一定数量的“门票”，可以“免费”返回内核，而无须任何并发性检查。这限制了它在返回到其他等待的线程队列之前可以完成的工作量。innodb_concurrency_tickets选项控制“门票”的数量。除非有很多非常长时间运行的查询，否则很少需要更改这个选项。“门票”是根据查询而不是事务授予的。一旦查询完成，未使用的门票将被丢弃。

除了缓冲池和其他结构中的瓶颈之外，在提交阶段还有另一个并发瓶颈，主要是由于刷新操作造成的I/O限制。innodb_commit_concurrency变量控制着可以同时提交的线程数。如果在innodb_thread_concurrency设置为较低的值时仍存在大量线程抖动，配置此选项可能会有所帮助。

安全设置

在基本配置设置就绪后，你可能还需要启用一些使服务器更安全、更可靠的设置。其中一些会影响性能，因为安全性和可靠性的保障成本往往更高。然而，有些设置是合理的：它们可以防止愚蠢的错误，比如在服务器中插入无意义的数据。有些并不会对日常运维产生影响，但可以防止在极端案例中发生不好的事情。

让我们先来看一组关于服务器行为的有用选项。

max_connect_errors

如果网络暂时出现问题、出现应用程序或配置错误，或者存在另一个问题导致连接无法在短时间内成功完成，则客户端可能会被阻止连接，并且在刷新主机缓存之前无法再次连接。此选项的默认设置（100）非常小，因此该问题很容易发生。你可能想要增加它，事实上，如果你知道服务器对暴力攻击具有足够的安全性，可以将其设置得非常大，以便有效地禁用由于连接错误而阻塞主机。但是，如果启用了skip_name_resolve，则max_connect_errors选项将无效，因为其行为取决于主机缓存，而主机缓存被skip_name_resolve禁用。

max_connections

此设置类似于紧急刹车，以防止服务器被来自应用程序的大量连接压垮。如果应用程序出现错误或服务器遇到问题（如暂停），可能会打开大量的新连接。但是，如果不能执行查询，那么打开连接就没有什么好处，因此被“too many connections”错误拒绝是一种快速失败且失败成本较低的方法。

可以将max_connections设置得足够高，以满足你认为将要经历的正常负载的连接需求，并且额外保留一些连接以便管理服务器时可以登录。例如，如果在正常操作中会有大约300个连接，那么可以将其设置为500。如果不知道会有多少个连接，500也并非一个不合理的起点。默认值为151，这对于很多应用程序来说都不够。

同时还要小心那些可能会让你达到连接极限的意外情况。例如，如果重新启动应用程序服务器，它可能不会干净地关闭连接，MySQL可能没有意识到它们已经被关闭。当应用服务器恢复并试图打开到数据库的连接时，可能会因为尚未超时的死连接而被拒绝。如果没有使用持久连接，但是应用程序没有正常断开连接，也会出现服务器连接占满的情况。服务器将保持连接，直到达到TCP超时，或者在最坏的情况下，直到wait_timeout配置的秒数。

注意观察随时间变化的max_used_connections状态变量。这是一个高水位线，可以显示服务器是否在某个时刻出现了连接高峰。如果到达max_connections，则客户端可能至少被拒绝了一次。

skip_name_resolve

此设置禁用另一个与网络和身份验证相关的陷阱：DNS查找。DNS是MySQL连接过程中的一个薄弱环节。当连接到服务器时，默认情况下，它会尝试确定正在连接的主机名，并将其作为身份验证凭据的一部分（也就是说，你的凭据是用户名、主机名和密码，而不仅仅是用户名和密码）。但是要验证主机名，服务器需要执行一个正向确

认的反向DNS查找（或“双反向DNS查找”），这在接受连接之前涉及反向和正向DNS查找。如果DNS没有问题，那这一切都很好，但在某个时间点出现DNS故障的概率几乎是确定性的。当DNS出现问题时，所有东西都会堆积起来，最终连接超时。为了防止出现这种情况，我们强烈建议设置此选项，该选项将在身份验证期间禁用DNS查找。但如果你这样配置了，则需要将所有基于主机名的授权转换为使用IP地址、通配符或特殊主机名“localhost”，因为基于主机名的账户将被禁用。

sql_mode

此设置可以接受各种选项，会影响服务器的行为。我们不建议仅仅为了好玩而修改该参数。在大多数情况下，最好让MySQL在大多数方面保持原样，而不要试图让它像其他数据库服务器那样运行。（例如，许多客户端和GUI工具都希望MySQL有自己的SQL风格，因此，如果你将其修改为更符合ANSI标准的SQL，可能会出现一些问题。）然而，其中有些设置非常有用，在特定情况下可能值得考虑。在早期，MySQL通常对sql_mode非常宽松，但在后来的版本中则严格得多。

但是，请注意，更改现有应用程序的这些设置可能不是一个好主意，这可能会使服务器与应用程序的预期不兼容。例如，人们在编写查询时无意中引用GROUP　BY子句以外的列或使用聚合函数是很常见的，因此如果你想启用ONLY_FULL_GROUP_BY选项，最好先在开发环境或预发布环境的服务器中进行，并在确定一切正常后再在生产环境中部署。

另外，在计划升级数据库时，一定要检查对默认sql_mode的更改。对该变量的更改可能与现有的应用程序不兼容，因此需要事先进行测试。我们将在附录A中讨论更多关于升级的内容。

sysdate_is_now

这是另一个可能与应用程序的期望向后不兼容的设置。但是，如果你不是明确地希望SYSDATE()函数具有不确定性行为（这会破坏复制并使从备份中进行的时间点恢复变得不可靠），那么可以启用该选项并使其行为具有确定性。

read_only和super_read_only

read_only选项可防止未经授权的用户对副本进行更改，副本应仅通过复制而不是从应用程序接收更改。我们强烈建议将副本设置为只读模式。

还有一个更严格的只读选项super_read_only，它甚至可阻止拥有SUPER权限的用户写入数据。启用此功能后，唯一可以将更改写入数据库的就是复制。我们也强烈建议启用super_read_only。它将防止你意外地使用管理员账户将数据写入只读副本，从而引起数据不同步。

高级**InnoDB**设置

还有一些InnoDB选项对服务器性能非常重要，其中包含一些安全选项。

innodb_autoinc_lock_mode

此选项控制InnoDB如何生成自动递增的主键值，在某些情况比如高并发插入时，这可能是一个瓶颈。如果有很多事务在等待自动增量锁（可以在SHOW ENGINE INNODB STATUS中看到），则应该调查此设置。这里不再重复手册中对选项及其行为的解释。

innodb_buffer_pool_instances

在MySQL 5.5及更新的版本中，此设置将缓冲池划分为多个段，这可能是提高多核机器上MySQL在高并发工作负载下可伸缩性最重要的方法之一。多个缓冲池对工作负载进行分区，这样一些全局互斥体就不会成为争用热点。

innodb_io_capacity

InnoDB过去是硬编码的，并且假设在一个每秒可以执行100次I/O操作的硬盘上运行。这是一个糟糕的假设。该选项可以告知InnoDB有多少I/O容量可供其使用。InnoDB有时需要很高的设置（在速度极快的存储设备，如PCIe闪存设备上，可能有上万个I/O）才能稳定地刷新脏页，原因比较复杂。[4]

innodb_read_io_threads和innodb_write_io_threads

这些选项控制有多少后台线程可用于I/O操作。MySQL最新的版本默认有4个读线程和4个写线程，这对很多服务器来说已经足够了，特别是在MySQL 5.5之后提供了可用的本地异步I/O。如果你有很多硬盘和高并发工作负载，并且发现线程很难跟上，那么可以增加线程的数量，或者简单地将它们设置为执行I/O操作的物理磁盘数量（即使它们位于RAID控制器后面）。

innodb_strict_mode

此设置使InnoDB在某些情况下抛出错误而不是警告，尤其是无效或可能导致危险的CREATE TABLE选项。如果启用此选项，一定要检查所有的CREATE TABLE选项，因为它可能不允许创建一些以前可以创建的表。有时它有点悲观和过度限制。你不会希望在恢复备份时发现这个问题。

innodb_old_blocks_time

InnoDB有一个由两部分组成的缓冲池LRU列表，设计目的是防止临时查询将长期多次使用的页面驱逐出去。一次性查询，如*mysqldump*发出的查询，通常会将一个页面带到缓冲池的LRU列表中，从其中读取行，然后移到下一个页面。理论上，两部分的LRU列表可以防止这个页面将需要很长一段时间的页面替换为“年轻”子列表，只有在被多次访问后才会移动到“年老”子列表。但是InnoDB在默认情况下不会阻止这种情况的发生，因为这个页面有多行，因此多次访问从这个页面读取的行会导致它立即被移动到“年老”的子列表中，给需要长生命周期的页面带来压力。这个变量指定一个页面从LRU列表的“年轻”部分移动到“年老”部分所需的毫秒数。默认情况下，它被设置为0。将其设置为一个较小的值如1000（1秒），这在我们的基准测试中被证明是非常

有效的。

小结

读完本章后，你应该拥有一个比默认配置好得多的服务器配置。你的服务器应该又快又稳定，除非遇到异常情况，否则不需要调整配置。

回顾一下，我们建议从示例配置文件开始，基于服务器和工作负载设置基本选项，并根据需要添加安全选项。这就是你真正需要做的。

如果运行的是专用数据库服务器，那么可以设置的最佳选项是innodb_dedicated_server，它可以处理90%的性能配置。如果无法使用此选项，那么最重要的两个选项是：

- innodb_buffer_pool_size
- innodb_log_file_size

恭喜你，已经解决了我们所看到的绝大多数实际配置问题！

我们还提出了很多关于不要做什么的建议。其中最重要的一点是不要“调优”服务器，不要使用比率、公式或“调优脚本”作为设置配置变量的基础。

---

[1] 例如，如果关闭数据持久性设置，**MySQL**可以运行得非常快，但它也会让你的数据在崩溃时很容易丢失。

[2] 如果你不相信**“**按比率调优**”**是不好的，请阅读**Cary Millsap**和**Jeff Holt**编写的***Optimizing OraclePerformance***。他们甚至专门为这个主题写了一个附录，里面有一个工具，可以人为地生成你想要的任何缓存命中率，无论你的系统性能有多差！当然，这一切都是为了说明这个比率是多么无用。

[3] 请注意，**MySQL**的不同版本会删除、弃用和更改一些选项，欲了解详细信息请查看相关文档。

[4] 进一步阅读可以参阅**Percona**的博客文章**“Give Love to Your SSDs Reduce innodb_io_capacity_max”**（参见链接**21**），**“InnoDB Flushing in Action for Percona Server for MySQL”**（参见链接**22**），以及**“Tuning MySQL/InnoDB Flushing for a Write-Intensive Workload”**（参见链接**23**）。

第6章 schema设计与管理

良好的逻辑设计和物理设计是高性能的基石，应该根据系统将要运行的特定查询设计schema。这通常需要权衡各种因素。例如，反范式的schema可以加速某些类型的查询，但同时可能减慢其他类型的查询。添加计数器和汇总表是一个优化查询的好方法，但它们的维护成本可能很高。MySQL的某些独有的特性和实现细节对性能的影响也很大。

同样，schema也会随着时间的推移而变化——这是你了解如何存储和访问数据以及业务需求如何随时间而变化的结果，这意味着应该将修改schema作为一个常见事件来规划。在本章的后面部分，我们将介绍如何避免此活动成为组织的运维瓶颈。

本章和聚焦在索引优化的下一章，涵盖了特定于MySQL的schema设计。我们假设你已经知道如何设计数据库，所以本章既不会介绍如何入门数据库设计，也不会讲解数据库设计方面的深入内容。这一章关注的是MySQL数据库的设计，主要介绍的是MySQL数据库设计与其他关系数据库管理系统的区别。如果你需要学习数据库设计的基础知识，我们推荐Clare Churcher的*Beginning Database Design*（Apress出版社出版）一书。

本章内容是为接下来的两章做铺垫的。在这三章中，我们将讨论逻辑设计、物理设计和查询执行，以及它们之间的相互作用。这既需要关注全局，又需要关注细节，还需要理解整个系统以便弄清楚各个部分如何相互影响。如果在阅读完关于索引的第7章和关于查询优化的第8章之后再来回顾这一章，也许你会发现本章很有用。很多讨论的主题都不能孤立地考虑。

选择优化的数据类型

MySQL支持的数据类型非常多，选择正确的数据类型对于获得高性能至关重要。不管存储哪种类型的数据，下面几个简单的原则都有助于你做出更好的选择。

更小的通常更好

一般来说，尽量使用能够正确存储和表示数据的最小数据类型。更小的数据类型通常更快，因为它们占用的磁盘、内存和CPU缓存的空间更少，并且处理时需要的CPU周期也更少。

但也要确保没有低估需要存储的值的范围，因为在schema中的多个地方增加数据类型范围是一个痛苦且耗时的操作。如果无法确定哪个数据类型是最好的，请选择你认为不会超过的最小数据类型。（如果系统不是很忙或存储的数据量不大，或者是在设计过程的早期阶段，也可以在之后轻松地修改数据类型）。

简单为好

简单数据类型的操作通常需要更少的CPU周期。例如，整型数据比字符型数据的比较操作代价更低，因为字符集和排序规则（collation）使字符型数据的比较更复杂。这里有两个例子：一个是应该将日期和时间存储为MySQL的内置类型而不是字符串类型，另外一个是应该用整型数据存储IP地址。稍后我们将专门讨论这个话题。

尽量避免存储NULL

即使应用程序本身并不需要存储NULL（缺失值），很多表也包含可为NULL的列，这是因为NULL可以是列的默认属性。通常情况下最好指定列为NOT NULL，除非明确需要存储NULL值。如果查询中包含可为NULL的列，对MySQL来说更难优化，因为可为NULL的列使得索引、索引统计和值比较都更复杂。可为NULL的列会使用更多的存储空间，在MySQL里也需要特殊处理。通常把可为NULL的列改为NOT NULL带来的性能提升比较小，所以（调优时）没有必要首先在现有schema中查找并修改这种情况，除非确定这会导致问题。

在为列选择数据类型时，第一步需要确定合适的大类型：数字、字符串、时间等。这通常是很简单的，但是我们会提到一些特殊的不是那么直观的案例。

下一步是选择具体类型。很多MySQL数据类型可以存储相同类型的数据，但在存储的值范围、表示的精度或者需要的物理空间（磁盘和内存）上存在着差异。相同大类型的不同子数据类型有时也有一些特殊的行为和属性。

例如，DATETIME和TIMESAMP列可以存储相同类型的数据：时间和日期，精确到秒。然而TIMESTAMP只使用DATETIME一半的存储空间，还会根据时区变化，而且具有特殊的自动更新能力。另一方面，TIMESTAMP允许的时间范围要小得多，有时候它的特殊能力会成为障碍。

本章只讨论基本的数据类型。MySQL为了兼容性支持很多别名，例如，INTEGER（映射到INT）、BOOL（映射到TINYINT）和NUMERIC（映射到DECIMAL）。它们都只是别名。这些别名可能令人不解，但不会影响性能。如果建表时采用数据类型的别名，然后用SHOW CREATE TABLE检查，会发现MySQL报告的是基本类型，而不是别名。

整数类型

有两种类型的数字：整数（whole number）和实数（real number，带有小数部分的数字）。如果存储整数，可以使用这几种整数类型：TINYINT、SMALLINT、MEDIUMINT、INT或BIGINT。它们分别使用8、16、24、32和64位存储空间。可以存储的值的范围从-2$^{(N-1)}$到2$^{(N-1)}$-1，其中*N*是存储空间的位数。

整数类型有可选的UNSIGNED属性，表示不允许负值，这大致可以使正数的上限提高一倍。例如，TINYINT UNSIGNED可以存储的值的范围是0～255，而TINYINT的值的存储范围是-128～127。

有符号和无符号类型使用相同的存储空间，并具有相同的性能，因此可以根据数据实际范围选择合适的类型。

你的选择决定了MySQL在内存和磁盘中保存数据的方式。然而，整数计算通常使用64位的BIGINT整数。（一些聚合函数是例外，它们使用DECIMAL或DOUBLE进行计算。）

MySQL可以为整数类型指定宽度，例如，INT（11），这对大多数应用毫无意义：它不会限制值的合法范围，只是规定了MySQL的一些交互工具（例如，MySQL命令行客户端）用来显示字符的个数。对于存储和计算来说，INT（1）和INT（20）是相同的。

实数类型

实数是带有小数部分的数字。然而，它们不仅适用于带小数的数字，也可以使用DECIMAL存储比BIGINT还大的整数。MySQL既支持精确类型，也支持不精确类型。

FLOAT和DOUBLE类型支持使用标准的浮点运算进行近似计算。如果你需要知道浮点运算是怎么计算的，则需要研究平台的浮点数的具体实现方式。

有两种方式可以指定浮点列所需的精度，这可能会导致MySQL以静默方式选择不同的数据类型，或者在存储值时对其进行近似处理。这些精度说明符是非标准的，因此我们建议只指定数据类型，不指定精度。

浮点类型通常比DECIMAL使用更少的空间来存储相同范围的值。FLOAT列使用4字节的存储空间。DOUBLE占用8字节，比FLOAT具有更高的精度和更大的值范围。与整数类型一样，你只能选择存储类型；MySQL会使用DOUBLE进行浮点类型的内部计算。

由于额外的空间需求和计算成本，应该尽量只在对小数进行精确计算时才使用DECIMAL——例如，存储财务数据。但在一些大容量的场景，可以考虑使用BIGINT代替DECIMAL，将需要存储的货币单位根据小数的位数乘以相应的倍数即可。假设要存储财务数据并精确到万分之一分，则可以把所有金额乘以一百万，然后将结果存储在BIGINT里，这样可以同时避免浮点存储计算不精确和DECIMAL精确计算代价高的问题。

字符串类型

MySQL支持多种字符串数据类型，每种类型还有许多变体。每个字符串列可以有自己的字符集和该字符集的排序规则集。

VARCHAR和**CHAR**类型

VARCHAR和CHAR是两种最主要的字符串类型。不幸的是，很难精确地解释这些值是如何存储在磁盘和内存中的，因为这跟存储引擎的具体实现有关。下面的描述假设使用的存储引擎是InnoDB。如果不是InnoDB，请参考所使用的存储引擎的文档。

先来看看VARCHAR和CHAR值通常是如何存储在磁盘上的。请注意，存储引擎在内存中存储CHAR或VARCHAR值的方式可能与在磁盘上存储该值的方式不同，并且服务器在从存储引擎检索该值时可能会将其转换为另一种存储格式。下面是关于两种类型的一些比较。

VARCHAR

VARCHAR用于存储可变长度的字符串，是最常见的字符串数据类型。它比固定长度的类型更节省空间，因为它仅使用必要的空间（即，更少的空间用于存储更短的值）。

VARCHAR需要额外使用1或2字节记录字符串的长度：如果列的最大长度小于或等于255字节，则只使用1字节表示，否则使用2字节。假设采用latin1字符集，一个VARCHAR（10）的列需要11字节的存储空间。VARCHAR（1000）的列则需要1002个字节，因为需要2字节存储长度信息。

VARCHAR节省了存储空间，所以对性能也有帮助。但是，由于行是可变长度的，在更新时可能会增长，这会导致额外的工作。如果行的增长使得原位置无法容纳更多内容，则处理行为取决于所使用的存储引擎。例如，InnoDB可能需要分割页面来容纳行。其他一些存储引擎也许不在原数据位置更新数据。

下面这些情况使用VARCHAR是合适的：字符串列的最大长度远大于平均长度；列的更新很少，所以碎片不是问题；使用了像UTF-8这样复杂的字符集，每个字符都使用不同的字节数进行存储。

InnoDB更为复杂，它可以将过长的VARCHAR值存储为BLOB。我们稍后再讨论。

CHAR

CHAR是固定长度的：MySQL总是为定义的字符串长度分配足够的空间。当存储CHAR值时，MySQL删除所有尾随空格。如果需要进行比较，值会用空格填充。

CHAR适合存储非常短的字符串，或者适用于所有值的长度都几乎相同的情况。例如，对于用户密码的MD5值，CHAR是一个很好的选择，它们的长度总是相同的。对于经常修改的数据，CHAR也比VARCHAR更好，因为固定长度的行不容易出现碎片。对于非常短的列，CHAR也比VARCHAR更高效；设计为只保存Y和N的值的CHAR（1）在单字节字符集[1]中只使用1字节，但VARCHAR（1）需要2字节，因为还有一个记录长度的额外字节。

CHAR类型的这些行为可能有一点让人难以理解，下面通过一个具体的例子来说明。首先，我们创建一张只有一个CHAR（10）列的表并且往里面插入一些值：

```
mysql> CREATE TABLE char_test( char_col CHAR(10));
mysql> INSERT INTO char_test(char_col) VALUES
    -> ('string1'), (' string2'), ('string3 ');
```

当检索这些值的时候，会发现末尾的空格被截断了：

```
mysql> SELECT CONCAT("'", char_col, "'") FROM char_test;
+---------------------------+
| CONCAT("'", char_col, "'") |
+---------------------------+
| 'string1'                 |
| ' string2'                |
| 'string3'                 |
+---------------------------+
```

如果用VARCHAR（10）字段存储相同的值，检索时会得到以下结果，其中string3末尾的空格并没有被删除：

```
mysql> SELECT CONCAT("'", varchar_col, "'") FROM varchar_test;
+------------------------------+
| CONCAT("'", varchar_col, "'") |
+------------------------------+
| 'string1'                    |
| ' string2'                   |
| 'string3 '                   |
+------------------------------+
```

与CHAR和VARCHAR类似的类型还有BINARY和VARBINARY，它们存储的是二进制字符串。二进制字符串与常规字符串非常相似，但它们存储的是字节而不是字符。填充也不同：MySQL填充BINANRY用的是\0（零字节）而不是空格，并且在检索时不会去除填充值。[2]

当需要存储二进制数据，并且希望MySQL将值作为字节而不是字符进行比较时，这些类型非常有用。字节比较的优势不仅仅是大小写不敏感。MySQL比较BINARY字符串时，每次按一个字节，并且根据该字节的数值进行比较。因此，二进制比较比字符比较简单得多，因此速度更快。

> 慷慨是不明智的
>
> 使用VARCHAR（5）和VARCHAR（200）存储'hello'的空间开销是一样的。那么使用更短的列有什么优势吗？
>
> 事实证明有很大的优势。较大的列会使用更多的内存，因为MySQL通常会在内部分配固定大小的内存块来保存值。这对于使用内存临时表的排序或操作来说尤其糟糕。在利用磁盘临时表进行文件排序时也同样糟糕。
>
> 最好的策略是只分配真正需要的空间。

BLOB和**TEXT**类型

BLOB和TEXT都是为存储很大的数据而设计的字符串数据类型，分别采用二进制和字符方式存储。

实际上，它们分别属于两组不同的数据类型家族：字符类型是TINYTEXT、SMALLTEXT、TEXT、MEDIUMTEXT和LONGTEXT；二进制类型是TINYBLOB、SMALLBLOB、BLOB、MEDIUMBLOB、LONGBLOB。BLOB是SMALLBLOB的同义词，TEXT是SMALLTEXT的同义词。

与其他数据类型不同，MySQL把每个BLOB和TEXT值当作一个具有自己标识的对象来处理。存储引擎通常会专门存储它们。当BLOB和TEXT值太大时，InnoDB会使用独立的“外部”存储区域，此时每个值在行内需要1～4字节的存储空间，然后在外部存储区域需要足够的空间来存储实际的值。

BLOB和TEXT家族之间的唯一区别是，BLOB类型存储的是二进制数据，没有排序规则或字符集，但TEXT类型有字符集和排序规则。

MySQL对BLOB和TEXT列的排序与其他类型不同：它只对这些列的最前max_sort_length字节而不是整个字符串做排序。如果只需要按前面少数几个字符排序，可以减小max_sort_length服务器变量的值。

MySQL不能将BLOB和TEXT数据类型的完整字符串放入索引，也不能使用索引进行排序。

> 在数据库中存储图像？
>
> 在过去，某些应用程序接受上传的图像并将其作为BLOB数据存储在MySQL数据库中，这是很常见的。这种方法便于将应用程序的数据保存在一起；但是，随着数据大小的增长，修改schema等操作会由于BLOB数据的大小而变得越来越慢。
>
> 如果可以避免的话，不要在数据库中存储像图像这样的数据。相反，应该将它们写入单独的对象数据存储，并使用该表来跟踪图像的位置或文件名。

使用枚举代替字符串类型

有时可以使用ENUM（枚举）列代替常规的字符串类型。ENUM列可以存储一组预定义的不同字符串值。MySQL在存储枚举时非常紧凑，会根据列表值的数量压缩到1或者2字节中。在内部会将每个值在列表中的位置保存为整数。这里有一个例子：

```
mysql> CREATE TABLE enum_test(
    -> e ENUM('fish', 'apple', 'dog') NOT NULL
    -> );
mysql> INSERT INTO enum_test(e) VALUES('fish'), ('dog'), ('apple');
```

上面三行实际上存储的是整数，而不是字符串。可以通过在数值上下文中检索看到这个双重属性：

```
mysql> SELECT e + 0 FROM enum_test;
+-------+
| e + 0 |
+-------+
|     1 |
|     3 |
|     2 |
+-------+
```

如果使用数字作为ENUM常量，这种双重属性很容易导致混乱，例如，
ENUM（'1'，'2'，'3'）。尽量避免这么做。

另一个令人惊讶的事情是，ENUM字段是根据内部整数值排序的，而不是根据字符串本

身：

```
mysql> SELECT e FROM enum_test ORDER BY e;
+-------+
| e     |
+-------+
| fish  |
| apple |
| dog   |
+-------+
```

可以通过按照需要的顺序指定ENUM成员来解决这个问题。也可以在查询中使用FIELD()函数显式地指定排序顺序，但这会导致MySQL无法利用索引消除排序：

```
mysql> SELECT e FROM enum_test ORDER BY FIELD(e, 'apple', 'dog', 'fish');
+-------+
| e     |
+-------+
| apple |
| dog   |
| fish  |
+-------+
```

如果是按字母顺序定义的值，就没有必要这么做了。

MySQL将每个枚举值存储为整数，并且必须进行查找以将其转换为字符串表示，因此ENUM列有一些开销。这些开销通常可以被ENUM列的小尺寸所抵消，但并不总是如此。特别是，将CHAR/VARCHAR列联接到ENUM列可能比联接到另一个CHAR/VARCHAR列更慢。

为了说明这一点，我们对一个应用程序中MySQL执行表联接的速度进行了基准测试。该表有一个相当大的主键：

```
CREATE TABLE webservicecalls (
 day date NOT NULL,
 account smallint NOT NULL,
 service varchar(10) NOT NULL,
 method varchar(50) NOT NULL,
 calls int NOT NULL,
 items int NOT NULL,
 time float NOT NULL,
 cost decimal(9,5) NOT NULL,
 updated datetime,
 PRIMARY KEY (day, account, service, method)
) ENGINE=InnoDB;
```

这个表大约有11万行，只有10MB大小，所以可以完全载入内存。service列包含5个不同的值，平均长度为4个字符，method列包含71个值，平均长度为2 0个字符。

我们复制一下这个表，并将service列和method列转换为ENUM类型，如下所示：

```
CREATE TABLE webservicecalls_enum (
 ... omitted ...
 service ENUM(...values omitted...) NOT NULL,
 method ENUM(...values omitted...) NOT NULL,
 ... omitted ...
) ENGINE=InnoDB;
```

然后我们测试用主键列进行联接操作的性能，下面是所使用的查询语句：

```
mysql> SELECT SQL_NO_CACHE COUNT(*)
    -> FROM webservicecalls
    -> JOIN webservicecalls USING(day, account, service, method);
```

我们用VARCHAR和ENUM列以不同的组合进行联接，结果如表6-1所示。[3]

表6-1：联接VARCHAR和ENUM列的速度

测试	QPS
VARCHAR 联接 VARCHAR	2.6
VARCHAR 联接 ENUM	1.7
ENUM 联接 VARCHAR	1.8
ENUM 联接 ENUM	3.5

从上面的结果可以看到，当把列都转换成ENUM以后，联接变得很快。但是当VARCHAR列和ENUM列进行联接时则慢很多。在本例中，只要不是必须让ENUM和VARCHAR列进行联接，那么将这些列转换为ENUM就是一个好主意。通常的设计实践是使用带有整数主键的“查找表”，以避免在联接中使用字符串。

然而，将列转换为ENUM类型还有另一个好处：根据SHOW TABLE STATUS输出结果中的Data_length列，发现将这两列转换为ENUM会使表变小约三分之一。在某些情况下，即使必须将ENUM列联接到VARCHAR列，这也可能是有益的。而且，转换后主键也只有原来的一半大小了，因为这是InnoDB表，如果表中有其他索引，减少主键大小也会使这些非主键索引小得多。

虽然ENUM类型在存储值的方式上非常有效，但更改ENUM中的有效值会导致需要做schema变更。如果你没有一个健壮的系统来支持自动schema变更（本章后面会进行描述），那么如果ENUM经常更改，这种操作需求可能会带来很大的不便。在后面的schema设计中，我们还会提到“枚举值过多”的反例。

日期和时间类型

MySQL中有很多数据类型用以支持各种各样的日期和时间值，比如YEAR和DATE。MySQL可以存储的最小时间粒度是微秒。大多数时间类型都没有其他选择，因此不存在哪一种是最佳选择的问题。唯一的问题是，当需要同时存储日期和时间时该怎么做。MySQL提供了两种非常相似的数据类型来实现这一需求：DATETIME和TIMESTAMP。

对于许多应用程序来说，两者都可以，但在某些场景，一个比另一个更好。我们来看一下。

DATETIME

这种类型可以保存大范围的数值，从1000年到9999年，精度为1微秒。它以YYYYMMDDHHMMSS格式存储压缩成整数的日期和时间，且与时区无关。这需要8字节的存储空间。

默认情况下，MySQL以可排序、无歧义的格式显示DATETIME值，例如，2008-01-16 22：37：08。这是ANSI表示日期和时间的标准方式。

TIMESTAMP

顾名思义，TIMESTAMP类型存储自1970年1月1日格林尼治标准时间（GMT）午夜以来经过的秒数——与UNIX时间戳相同。TIMESTAMP只使用4字节的存储空间，所以它的范围比DATETIME小得多：只能表示从1970年到2038年1月19日。MySQL提供FROM_UNIXTIME()函数来将UNIX时间戳转换为日期，并提供了UNIX_TIMESTAMP()函数将日期转换为UNIX时间戳。

时间戳显示的值依赖于时区。MySQL服务器、操作系统和客户端连接都有时区设置。

因此，存储值0的TIMESTAMP在美国东部标准时间（EST）中显示为1969-12-31 19：00：00，与格林尼治标准时间（GMT）差5小时。有必要强调一下这个区别：如果存储或访问多个时区的数据，TIMESTAMP和DATETIME的行为将很不一样。前者保留与所使用时区相关的值，而后者保留日期和时间的文本表示。

TIMESTAMP还有DATETIME没有的特殊属性。默认情况下，当插入一行记录时没有指定第一个TIMESTAMP列的值，MySQL会将该列的值设置为当前时间。[4]当更新一行记录时没有指定第一个TIMESTAMP列的值，MySQL默认也会将该列的值更新为当前时间。可以为任何TIMESTAMP列配置插入和更新行为。最后，TIMESTAMP列在默认情况下为NOT NULL，这也和其他的数据类型不一样。

将日期和时间存储为整数？

DATETIME和TIMESTAMP都迫使你处理服务器和客户端上的时区，虽然TIMESTAMP比DATETIME更节省空间（4字节与8字节的区别，忽略分数秒支持），但它也会遇到2038年的问题。

最终，存储日期和时间归结为以下几件事：

- 需要支持前后多大范围的日期和时间？
- 存储空间对这些数据有多重要？
- 需要支持分数秒吗？
- 在MySQL中处理日期、时间和时区，还是在代码中处理？

通过将日期和时间存储为UNIX纪元（即自1970年1月1日以来的秒数），以协调世界时（UTC）的形式，可避免MySQL处理的复杂性，这一做法越来越流行。使用带符号的32位INT，可以表达直到2038年的时间。使用无符号的32位INT，可以表达直到

2106年的时间。如果使用64位，还可以超出这些范围。

就像关于操作系统、编辑器和标签与空间的流行讨论一样，如何存储这组特定的数据可能更多的是一种观点，而不是最佳实践。需要考虑的是，这对于你的用例来说是否可行。

位压缩数据类型

MySQL有几种使用值中的单个位来紧凑地存储数据的类型。所有这些位压缩类型，不管底层存储和处理方式如何，从技术上来说都是字符串类型。

BIT

可以使用BIT列存储一个或多个true/false值。BIT（1）定义一个包含1位的字段，BIT（2）存储2位的字段，依此类推；BIT列的最大长度为64位。InnoDB将每一列存储为足够容纳这些位的最小整数类型，所以使用BIT列不会节省任何存储空间。

MySQL在处理时会将BIT视为字符串类型，而不是数字类型。当检索BIT（1）的值时，结果是一个包含二进制值0或1的字符串，而不是ASCII码的“0”或“1”。但是，如果在数字上下文中检索该值，则会将BIT字符串转换为数字。如果需要将结果与另外的值进行比较，一定要记得这一点。例如，如果将值b'00111001'（二进制数相当于57）存储到BIT（8）列中并检索它，则将得到包含字符码为57的字符串。这恰好是“9”的ASCII字符代码。但在数字上下文场景中，得到的将会是数字57：

```
mysql> CREATE TABLE bittest(a bit(8));
mysql> INSERT INTO bittest VALUES(b'00111001');
mysql> SELECT a, a + 0 FROM bittest;
+------+-------+
| a    | a + 0 |
+------+-------+
| 9    |    57 |
+------+-------+
```

这可能会让人非常困惑，因此我们建议谨慎使用BIT类型。对于大多数应用来说，最好避免使用这种类型。

如果想在1位的存储空间中存储true/false值，另一个方法是创建一个可为空的CHAR（0）列。该列可以存储空值（NULL）或长度为零的值（空字符串）。这在实践中是可行的，但可能对使用数据库中该数据的其他人来说是难以理解的，并且使编写查询变得困难。除非你非常注重节省空间，否则我们仍然建议使用TINYINT。

SET

如果需要存储多个true/false值，可以考虑使用MySQL原生的SET数据类型，可以将多列组合成一列，这在MySQL内部是以一组打包的位的集合来表示的。这样可以更有效地利用存储空间，MySQL具有FIND_IN_SET()和FIELD()等函数，使其易于在查询中使用。

整数列上的位操作

SET的另一种替代方法是使用整数作为二进制位的打包集合。例如，可以在TINYINT中打包8位，并使用逐位操作符对它们进行操作。可以在应用程序代码中为每个位定义命名常量来简化这一过程。

与SET相比，这种方法的主要优点是可以在不使用ALTER TABLE的情况下更改字段表示的“枚举”。缺点是查询更难编写和理解（当设置第5位时是什么意思）。有些人喜欢位操作，有些人则不喜欢，所以是否想尝试这种技术很大程度上取决于个人的偏好。

一个封装位的应用示例是保存权限的访问控制列表（ACL）。每个位或SET元素代表一个值，例如CAN_READ、CAN_WRITE或CAN_DELETE。如果使用SET列，可以让MySQL在列定义中存储位到值的映射；如果使用整数列，则可以在应用程序代码中存储这个映射。下面是使用SET列的查询：

```
mysql> CREATE TABLE acl (
    -> perms SET('CAN_READ', 'CAN_WRITE', 'CAN_DELETE') NOT NULL
    -> );
mysql> INSERT INTO acl(perms) VALUES ('CAN_READ,CAN_DELETE');

mysql> SELECT perms FROM acl WHERE FIND_IN_SET('CAN_READ', perms);
+--------------------+
| perms              |
+--------------------+
| CAN_READ,CAN_DELETE |
+--------------------+
```

如果使用整数列，则可以参考下面的例子：

```
mysql> SET @CAN_READ := 1 << 0,
    -> @CAN_WRITE := 1 << 1,
    -> @CAN_DELETE := 1 << 2;
mysql> CREATE TABLE acl (
    -> perms TINYINT UNSIGNED NOT NULL DEFAULT 0
    -> );
mysql> INSERT INTO acl(perms) VALUES(@CAN_READ + @CAN_DELETE);
mysql> SELECT perms FROM acl WHERE perms & @CAN_READ;
+-------+
| perms |
+-------+
| 5     |
+-------+
```

这里我们使用了MySQL变量来定义值，也可以在代码中使用常量来代替。

JSON数据类型

使用JSON作为系统之间交换数据的格式正变得越来越普遍。MySQL有原生的JSON数据类型，可以方便地直接在表中的JSON结构部分进行操作。纯粹主义者可能会认为，在数据库中存储原始JSON是一种反范式，因为理想情况下，schema应该是JSON中具体字段的表示。新手在查看JSON数据类型时，可能会发现这是避免创建和管理独立字段的捷径。

哪种方法更好在很大程度上是主观的，但我们将客观地展示一个使用示例，并比较查询速度和数据大小。

我们的样本数据是由NASA提供的202颗近地小行星和彗星的发现清单（参见链接24）。测试的版本是MySQL 8.0.22，运行在一个4核、16GB内存的虚拟机上。数据示例如下：


```
[
  {
     "designation":"419880 (2011 AH37)",
     "discovery_date":"2011-01-07T00:00:00.000",
     "h_mag":"19.7",
     "moid_au":"0.035",
     "q_au_1":"0.84",
     "q_au_2":"4.26",
     "period_yr":"4.06",

    "i_deg":"9.65",
    "pha":"Y",
    "orbit_class":"Apollo"
  }
]
```


这份数据是关于名称、发现日期及收集的有关实体的数据，包括数字和文本字段。

首先，我们以JSON格式来获取数据集，并将其转换为每个条目一行。这是一个看起来相对简单的schema：

```
mysql> DESC asteroids_json;
+-----------+------+------+-----+---------+-------+
| Field     | Type | Null | Key | Default | Extra |
+-----------+------+------+-----+---------+-------+
| json_data | json | YES  |     | NULL    |       |
+-----------+------+------+-----+---------+-------+
```

其次，我们将该JSON用合适的数据类型将字段转换为列。可以得到以下schema：

```
mysql> DESC asteroids_sql;
+----------------+-------------+------+-----+---------+-------+
| Field          | Type        | Null | Key | Default | Extra |
+----------------+-------------+------+-----+---------+-------+
| designation    | varchar(30) | YES  |     | NULL    |       |
| discovery_date | date        | YES  |     | NULL    |       |
| h_mag          | float       | YES  |     | NULL    |       |
| moid_au        | float       | YES  |     | NULL    |       |
| q_au_1         | float       | YES  |     | NULL    |       |
| q_au_2         | float       | YES  |     | NULL    |       |
| period_yr      | float       | YES  |     | NULL    |       |
| i_deg          | float       | YES  |     | NULL    |       |
| pha            | char(3)     | YES  |     | NULL    |       |
| orbit_class    | varchar(30) | YES  |     | NULL    |       |
+----------------+-------------+------+-----+---------+-------+
```

下面来比较数据大小：

```
mysql> SHOW TABLE STATUS\G
*************************** 1. row ***************************
 Name: asteroids_json
 Engine: InnoDB
 Version: 10
 Row_format: Dynamic
 Rows: 202
 Avg_row_length: 405
 Data_length: 81920
 Max_data_length: 0
 Index_length: 0

*************************** 2. row ***************************
 Name: asteroids_sql
 Engine: InnoDB
 Version: 10
 Row_format: Dynamic
 Rows: 202
 Avg_row_length: 243
 Data_length: 49152
 Max_data_length: 0
 Index_length: 0
```

可以看到，这个例子中的SQL版本使用了3个16KB的页来存储，JSON版本则使用了5个16KB的页。这并不令人惊讶。JSON数据类型将使用更多空间来存储用于定义JSON的额外字符（大括号、方括号、冒号等）以及空格。在这个小例子中，可以通过将JSON转换为特定的数据类型来优化数据存储的大小。

在一些有效的用例中，数据大小可能不是那么重要。接下来看看如何衡量两者之间的查询延迟？

要使用SQL语句选择单列的所有内容，语法很简单：

```
SELECT designation FROM asteroids_sql;
```

在第一次运行这个查询时，InnoDB的缓冲池中没有缓存，我们得到了1.14毫秒的结果。第二次执行时，已经在内存中进行了缓存，则只要0.44毫秒。

对于JSON，可以用如下方式访问JSON结构中的字段：

```
SELECT json_data->'$.designation' FROM asteroids_json
```

类似地，第一次未缓存时需要1.13毫秒，缓存后的执行时间约为0.80毫秒。在这个执行速度下，我们预计会有一个合理的变化——我们讨论的是VM环境中数百微秒的差异。在我们看来，这两个查询执行得相当快，值得注意的是，JSON版本的查询仍然是SQL版本的两倍长。

即便如此，访问特定行将会怎么样呢？对于单行查找，可以利用索引的优势：

```
ALTER TABLE asteroids_sql ADD INDEX ( designation );
```

当进行单行查找时，SQL版本的运行时间为0.33毫秒，JSON版本的运行时间为0.58毫秒，SQL版本具有优势。这很容易解释：使用索引使得InnoDB只返回1行，而不是202行。

不过，将索引查询与全表扫描进行比较是不公平的。为了公平竞争，我们可以使用虚拟列的特性来提取designation值，然后针对虚拟列创建索引：

```
ALTER TABLE asteroids_json ADD COLUMN designation VARCHAR(30) GENERATED ALWAYS AS
(json_data->"$.designation"), ADD INDEX ( designation );
```

这给了我们一个JSON表的schema，看起来像下面这样：

```
mysql> DESC asteroids_json;
+-------------+-------------+------+-----+---------+-------------------+
| Field       | Type        | Null | Key | Default | Extra             |
+-------------+-------------+------+-----+---------+-------------------+
| json_data   | json        | YES  |     | NULL    |                   |
| designation | varchar(30) | YES  | MUL | NULL    | VIRTUAL GENERATED |
+-------------+-------------+------+-----+---------+-------------------+
```

这个schema从json_data列生成一个虚拟列并创建了索引。现在，我们重新运行单行查找，以使用索引列而不是JSON列路径操作符（->）。由于字段数据在JSON中被引用，所以我们还需要搜索SQL中引用的数据：

```
SELECT * FROM asteroids_json WHERE designation='"(2010 GW62)"';
```

这个查询在0.4毫秒内执行完成，非常接近SQL版本的0.33毫秒。

从前面的简单测试用例来看，我们使用的表空间总量似乎是使用SQL列而不是存储原始JSON文档的主要驱动因素。使用SQL列的速度仍然更好。总的来说，决定使用原生SQL还是JSON取决于在数据库中存储JSON的便捷性是否大于性能。如果每天访问这些数据数百万次或数十亿次，速度差异就会累加起来。

选择标识符

一般来说，标识符是引用行及通常使其唯一的方式。例如，如果你有一个关于用户的表，可能希望为每个用户分配一个数字ID或唯一的用户名。此字段可能是主键中的部分或全部。

为标识符列选择合适的数据类型非常重要。与其他列相比，更有可能将标识符列与其他值（例如，在联接中）进行比较，并使用它们进行查找。标识符列也可能在其他表中作为外键，因此为标识符列选择数据类型时，应该与联接表中的对应列保持一致。（正如我们在本章前面演示的，在关联表中使用相同的数据类型是一个好主意，因为这些列很可能在联接中使用。）

在为标识符列选择类型时，不仅需要考虑存储类型，还需要考虑MySQL如何对该类型执行计算和比较。例如，MySQL在内部将ENUM和SET类型存储为整数，但在字符串上下文中进行比较时，会将它们转换为字符串。

选择类型后，要确保在所有相关表中使用相同的类型。类型应该完全匹配，包括UNSIGNED等属性。[5]混合不同的数据类型可能导致性能问题，即使没有性能影响，在进行比较操作时，隐式类型转换也可能会产生难以发现的错误。甚至在很久以后，当你忘记正在比较不同类型的数据时，这些问题可能会突然出现。

在可以满足值的范围的需求，并且预留未来增长空间的前提下，应该选择最小的数据类型。例如，如果有一个state_id列存储美国各州的名字，则不需要数千或数百万个值，因此不要使用INT，TINYINT就足够了，它比INT少3字节。如果在其他表中使用此值作为外键，3字节可能会产生很大的性能差异。这里有一些小建议。

整数类型

整数通常是标识符的最佳选择，因为它们速度快，并且可以自动递增。AUTO_INCREMENT是一个列属性，可以为新的行自动生成一个整数类型的值。例如，计费系统可能需要为每个客户生成新发票，使用AUTO_INCREMENT意味着生成的第一张发票是1，第二张是2，依此类推。请注意，应该确保选择适合预期数据增长的整数大小，与整数意外耗尽有关的系统停机事故可不止发生一次。

ENUM和**SET**类型

对于标识符来说，ENUM和SET类型通常是糟糕的选择，尽管对某些只包含固定状态或者类型的静态“定义表”来说可能是没有问题的。ENUM和SET列适用于保存订单状态或产品类型等信息。

举个例子，如果使用ENUM字段来定义产品类型，可能会设计一张以这个ENUM字段为主键的查找表。（可以在查找表中添加描述性文本的列，以生成术语表，或者在网站上的下拉菜单中提供有意义的标签。）在这种情况下，使用ENUM类型作为标识符是可行的，但是大部分情况下都要避免这么做。

字符串类型

如果可能，应避免使用字符串类型作为标识符的数据类型，因为它们很消耗空间，而且通常比整数类型慢。

对于完全“随机”的字符串要非常小心，如MD5()、SHA1()或UUID()生成的字符串。这些函数生成的新值会任意分布在很大的空间内，这会减慢INSERT和某些类型的SELECT查询的速度：[6]

- 因为插入的值会写到索引的随机位置，所以会使得INSERT查询变慢。这会导致页分裂、磁盘随机访问，以及对于聚簇存储引擎产生聚簇索引碎片。
- SELECT查询也会变慢，因为逻辑上相邻的行会广泛分布在磁盘和内存中。
- 对于所有类型的查询，随机值都会导致缓存的性能低下，因为它们会破坏引用的局部性，而这正是缓存的工作原理。如果整个数据集都是“热的”，那么将任何特定部分的数据缓存到内存中都没有任何好处，而且如果工作集比内存大，缓存就会出现大量刷新和不命中。

如果存储通用唯一标识符（UUID）值，则应该删除破折号，或者更好的做法是，使用UNHEX()函数将UUID值转换为16字节的数字，并将其存储在一个BINARY（16）列中。可以使用HEX()函数以十六进制格式检索值。

当心自动生成的**schema**

我们已经讨论了最重要的有关数据类型的注意事项（一些涉及严重的性能问题，另一些则影响较小），但我们还没有提到自动生成schema的坏处。

写得不好的schema迁移程序和自动生成schema的程序可能会导致严重的性能问题。有些程序存储任何数据都会使用很大的VARCHAR字段，或者对将在联接中进行比较的列使用不同的数据类型。如果schema是自动生成的，一定要反复检查确认没有问题。

对象关系映射（ORM）系统（以及使用它们的“框架”）通常是另一个性能“噩梦”。其中一些ORM系统会将任意类型的数据存储到任意类型的后端数据存储中，这通常意味着其没有设计使用更优的数据存储。有时ORM系统会为每个对象的每个属性使用单独的行来存储，甚至使用基于时间戳的版本控制，导致每个属性有多个版本存在！

这种设计对开发者很有吸引力，因为这使得他们可以用面向对象的方式工作，而不需要考虑数据是如何存储的。然而，“对开发人员隐藏复杂性”的应用程序通常不能很好地扩展。我们建议在用性能交换开发人员的效率之前仔细考虑，并始终在实际的大型数据集上进行测试，这样就不会太晚才发现性能问题。

特殊数据类型

某些类型的数据并不直接对应于可用的内置类型。IPv4地址就是一个很好的例子。人们通常使用VARCHAR（15）列来存储IP地址。然而，它们实际上是32位无符号整数，而不是字符串。用小数点将地址分成四段的表示方法只是为了让人们阅读容易，所以应该将I P地址存储为无符号整数。MySQL提供了INET_ATON()和INET_NTOA()函数来在这两种表示形式之间进行转换。使用的空间从VARCHAR（15）的约16字节缩减到无符号32位整数的4字节。如果你担心数据库的可读性，不想继续使用函数查看行数据，请记住MySQL有视图，可以使用视图来简化数据查看的复杂性。

MySQL schema设计中的陷阱

尽管设计原则有好有坏，但MySQL的实现方式会带来一些问题，这意味着你也可能会犯MySQL特有的错误。本节讨论我们在MySQL schema设计中观察到的问题。它可能会帮助你避免这些错误，并让你选择更适合MySQL具体实现的替代方案。

太多的列

MySQL的存储引擎API通过在服务器和存储引擎之间以行缓冲区格式复制行来工作；然后，服务器将缓冲区解码为列。将行缓冲区转换为具有解码列的行数据结构的操作代价是非常高的。InnoDB的行格式总是需要转换的。这种转换的成本取决于列数。当调查一个具有非常宽的表（数百列）的客户的高CPU消耗问题时，我们发现这种转换代价可能会变得非常昂贵，尽管实际上只使用了几列。如果计划使用数百列，请注意服务器的性能特征会有所不同。

太多的联接

所谓的实体属性值（entity attribute value，EAV）设计模式是一种被普遍认为糟糕的设计模式的典型案例，尤其是在MySQL中效果不佳。MySQL限制每个联接有61个表，而E AV模式设计的数据库需要许多自联接。我们已经看到不少E AV模式设计的数据库最终超过了这个限制。然而，即使联接数远小于61，规划和优化查询的成本对MySQL来说也会成为问题。一个粗略的经验法则是，如果需要以高并发性快速执行查询，那么每个查询最好少于十几个的表。

全能的枚举

要小心过度使用ENUM。下面是我们看到的一个例子：

```
CREATE TABLE ... (
 country enum(',0','1','2',...,'31')
```

schema中大量地散布着这种模式。在任何具有枚举值类型的数据库中，这可能是一个值得商榷的设计决策，因为它实际上应该是一个整数，会被设计为“字典”或“查找”表的外健。

变相的枚举

ENUM列允许在列中保存一组已定义值中的单个值。SET列则允许在列中保存一组已定义值中的一个或多个值。有时很容易混淆。这里有一个例子：

```
CREATE TABLE ...(
 is_default set('Y','N') NOT NULL default 'N'
```

如果这里真和假两种情况不会同时出现，那么毫无疑问应该使用ENUM列而不是SET列。

NULL不是虚拟值

我们之前说过避免使用NULL的好处，并且建议尽可能考虑其他选择。即使需要在表中存储事实上的“空值”，也可能不需要使用NULL。也许可以使用0、特殊值或空字符串作为代替。

但是遵循这个原则也不要走极端。当需要表示未知值时，不要太害怕使用NULL。在某些情况下，使用NULL比使用某个虚拟常数更好。从受约束类型的域中选择一个值，例如使用-1表示一个未知的整数，可能会使代码复杂化，容易引入bug，并通常会把事情搞得一团糟。处理NULL并不容易，但通常比其他替代方案更好。

下面是我们经常看到的一个例子：

```
CREATE TABLE ... (
 dt DATETIME NOT NULL DEFAULT '0000-00-00 00:00:00'
```

这个虚假的全0值可能会导致很多问题（可以将MySQL的SQL_MODE配置为不允许使用无意义的日期，对于尚未创建满是坏数据的数据库的新应用程序来说，这是一个特别好的实践）。

还有一个相关的细节，MySQL会对NULL值进行索引，而Oracle则不会。

到这里我们已经讨论了很多关于数据类型、如何选择数据类型以及不做什么的实用建议，接下来让我们继续讨论另一个好的、迭代式的设计：schema管理。

schema管理

修改schema是数据库工程师必须承担的最常见任务之一。当达到运行数十个或数百个数据库实例的阶段，而这些实例又具有不同的业务场景和不断变化的功能时，需要注意，不要让修改schema成为整个组织的瓶颈，而是仍然可以安全地进行且不会中断操作。本节将介绍如何将schema变更管理视为“数据存储平台”的一部分，该战略应该以什么核心价值观为指导，可以引入什么工具来实现该战略，以及如何将其整合到更大的软件交付生命周期中。

作为数据存储平台一部分的**schema**管理

如果你与一个快速发展的组织中的任何一位工程主管交谈，你会发现工程师的效率和从功能设计到投入生产的时间是他们最需要优化的事情。在这种情况下，在计划大规模管理schema时，你的任务是不允许schema管理成为一个手动过程，从而因为一个或几个人阻碍整个工程组织的进度。

建立合作伙伴团队以获得成功

随着组织中依赖MySQL实例的团队数量的增长，你希望始终成为这些团队成功的赋能者，而不是他们完成工作所需要通过的难关。这也适用于schema修改，这意味着你希望创建一个不依赖于“仅由数据库团队执行”的部署schema修改的路径。

将**schema**管理与持续集成相整合

在介绍完一些支持大规模schema管理的工具之后，我们将讨论如何将它们与CI管道集成。但现在我们想强调的是，如果你从schema修改将由功能开发团队而不仅仅是数据库团队管理的前提开始，那么你需要尽可能地接近工作流，以了解这些团队如何部署代码更改。科学已经证明，以对待代码部署的方式对待schema管理的团队会经历一个更积极的功能交付过程，并看到团队效率的提升。我们将在考虑软件交付实践的情况下讨论支持该迭代的工具。

schema修改的源代码控制

我们部署的代码都会使用源代码控制，对吗？那么为什么不考虑数据库的schema应该是什么样子的呢？大规模schema管理的第一步是确保源代码管理支持并跟踪所做的更改。这不仅是一件值得做的好事，在很多情况下，这是友好的合规团队所要求的，如第13章所示。让我们介绍一些能够在数据库schema上进行迭代的工具。

为了使你的组织获得最大的价值，请使用与代码部署相同的CI工具。

付费方案。在过去的几年中，数据库schema管理作为一种企业工具的前景急剧增长，特别是增加了对MySQL安装的支持之后。如果你正在寻找一个现成的解决方案来帮助组织管理schema更改，如下是一些应该考虑的事情。

成本

成本模型各不相同，所以如果你选择的解决方案是按目标（要管理的schema）收费，那么应该小心，因为成本可能会很快累加到非常高。

在线schema管理

在撰写本文时，Flyway等付费解决方案（参见链接25）没有一条清晰的路径来以非阻塞的方式为你运行schema更改，尽管它的竞争对手Liquibase正在推出（参见链接26）一个支持Percona的在线schema更改的插件。你需要了解每个供应商代替你做出的权衡，以及这些权衡对可用性意味着什么，尤其是如果你计划使用这些供应商来管理大型数据库（磁盘上有多TB）的schema更改。

开箱即用的集成

这些工具中的大多数都假设了你的内部软件是用什么语言编写的，因此提供了对应的挂钩接口（Hook）来与现有软件交付过程相集成。如果你的应用由多种语言编写，或者正在改变主要的软件语言，这可能会将其中一些供应商排除在外。在下一节中，我们将介绍在实现schema的源代码管理时，如果需要“自己动手”该怎么做。

开源方案。如果购买付费工具遥不可及，或者如果有正当理由认为当前的解决方案都不适合你的组织，那么可以使用现有的开源工具和组织的CI管道来实现相同的结果。

Skeema（参见链接27）是一个在跨多个环境的版本控制中管理schema更改的杰出开源解决方案。Skeema本身不会在生产环境中为你运行schema更改（我们将很快介绍如何做到这一点），但是它是一个很好的工具，可以在每个数据库集群和跨多个环境中跟踪源代码控制存储库中的更改。当与选择的CI解决方案集成时，它的CLI实现提供了很大的灵活性。如何将Skeema直接与CI解决方案集成，需要考虑CI解决方案所具有的功能。这篇由Twilio Sendgrid团队撰写的博客文章（参见链接28）解释了他们如何将Skeema与Buildkite整合在一起，从而为那些希望管理数据库变更的特性开发团队实现自治。

请注意，无论此解决方案如何与CI集成，它都需要访问权限才能对所有环境（包括生产环境）运行schema更改。这意味着还需要与安全团队合作，确保创建正确的访问控制，以获得使用持续集成进行自动化schema部署的好处。

如果你已经在使用Vitess扩展数据库基础设施，那么应该知道Vitess还可以为你管理schema更改。请确保检查文档的相关部分。

在过去的几年中，使用自动化和合规性思维方式管理跨环境schema变更的领域得到了显著的发展。在你做出选择时，以下是最后的一些建议：

- 尽可能靠近现有的软件部署工具和工作流程。当工程组织变得更大时，你会希望熟悉这一点。
- 应该使用能够集成针对schema更改的基本检测的工具，以确保满足一些基线需求。如果新表没有使用正确的字符集，或者有已决定不允许使用的外键，那么你的解决方案应该自动使代码合并请求失败。

- 如果所在的组织使用多种编程语言且发展迅速，请确保不会意外地引入人为瓶颈，例如，将所有数据库和所有schema更改都放在一个代码存储库中。记住，这里的目标是工程团队的效率。

在生产环境中运行**schema**更改

我们已经讨论了跟踪和管理组织部署schema更改的选项，接下来讨论如何在不影响数据库或依赖它们的服务的正常运行时间的情况下在生产环境中运行这些更改。

原生DDL语句。MySQL在5.6版中引入了非阻塞的schema更改，但在这个大版本中，该特性附带了一些警告，使得仅限于非常特定的schema更改类型才能使用。

当8.0版本正式发布时，MySQL对原生DDL的支持得到了极大的扩展，尽管仍然不是通用的。对主键的更改、对字符集的更改、开启每个表的加密以及添加或删除外键，这些都是仍无法通过就地变更（INPLACEalter）的方式进行schema更改的例子。[7]我们强烈建议通过文档熟悉使用就地（INPLACE）或即时（INSTANT）算法可以进行哪些更改，这是在MySQL中进行schema更改而不停机的首选原生方法。

然而，即使需要的更改在8.0及更高版本中已经在技术上有原生DDL的支持，如果要更改的表非常大，而此时如果InnoDB内部保留的记录表更改的日志文件太大，也可能会遇到回滚，导致白费了数小时或数天的工作。你可能需要使用外部工具的另一个原因是，你强烈希望使用节流机制控制表更改的速度。这是可以使用我们将要讨论的外部工具来管理的。

使用外部工具来运行schema更改。即使你还不能运行具有就地schema更改的所有灵活性的最新和最好的MySQL版本，也仍然可以将CI工具与可用的开源工具结合起来，在不影响服务的情况下自动运行schema更改。实现这一目标的两个主要选择是Percona的pt-online-schema-change和GitHub的gh-ost。这两个工具都有文档来学习如何安装和使用，所以我们将集中讨论如何对二者进行选择，你应该考虑的主要优缺点是什么，以及如何提高使用这两种工具作为生产中自动化schema部署管道的安全性。

需要注意的一点是：任何运行schema更改的外部工具都需要对正在更改的表进行完整的复制。这些工具只会降低更改过程的影响，并且不需要破坏性的写锁，但只有MySQL中的原生DDL可以在没有完整表拷贝的情况下更改表schema。

pt-online-schema-change的主要吸引力在于它的稳定性以及在MySQL社区中的长期使用。当切换到新的表版本时，它主要利用触发器来为各种规格的表支持schema更改，而对数据库可用性的影响很小。但其核心设计也有权衡。在学习使用pt-online-schema-change来支撑schema部署管道时，请记住以下几点。

触发器的局限性

在MySQL 8.0之前，同一个表中不能有多个具有相同操作的触发器。这是什么意思？如果有一个名为sales的表，并且你已经在其上维护一个insert定时触发器，那么MySQL 8.0之前的版本不允许在该表中使用另一个insert触发器。如果试图使用pt-online-schema-change对其运行schema更改，则该工具在尝试添加其运行所需的触发器

时将产生错误。尽管我们通常极不鼓励将表触发器作为业务逻辑的一部分使用，但仍然存在遗留选择创建约束的情况，在选择schema更改机制时，这将成为权衡计算的一部分。

触发器对性能的影响

Percona（参见链接30）提供了一些出色的基准测试，展示了在表中定义触发器对性能的影响。大多数情况下这种性能损失可能是无形的，但是如果你恰巧运行的是一个每秒事务吞吐量非常高的数据库实例，那么可能需要更仔细地观察通过pt-online-schema-change引入的触发器的影响，并对其进行更保守的调整以及时中止。

执行并发迁移

由于在8.0之前的MySQL中使用触发器具有局限性，你会发现不能使用pt-online-schema-change在同一个表中运行多个schema更改。这在一开始可能是一个小麻烦，但如果将该工具集成到一个完整的自动化schema迁移管道中，就可能会成为团队的瓶颈。

外键约束

尽管该工具对有外键的schema更改提供了一定程度的支持，但仍需要仔细阅读文档，并权衡如何操作才能对数据和事务吞吐量的影响最小。

gh-ost是由GitHub的数据工程团队创建的，专门作为一种管理schema更改过程的解决方案，既不影响服务，也不使用触发器。它不使用触发器在表复制阶段跟踪更改，而是以副本的形式连接到集群中，并将基于行的复制日志作为更改日志使用。

在使用gh-ost进行schema更改时，需要仔细考虑的一件事是，现有的数据库是否使用了外键。虽然pt-online-schema-change已经尝试支持外键关系中的父表或子表的schema更改，但这是一个复杂的选择，充满了权衡（是牺牲正常运行时间来保持一致性，还是冒着可能出现不一致的风险）。另外，gh-ost通常会做出选择，如果要更改的表中存在外键，它会完全退出。作为gh-ost的主要贡献者，Shlomi Noach在一篇很长但非常有用的博客文章（参见链接31）中解释说，在线schema更改工具终究是工作在数据库引擎之外的，同时使用外键和在线schema更改工具会创建一个难以权衡的环境，他还建议，如果需要在线schema更改，最好完全不要使用外键。

如果你和团队是这项任务的新手，并且正在为组织中的schema更改的CI铺平道路，只要能够严格遵守不引入外键的原则，那么我们相信gh-ost是更好的解决方案。鉴于其使用的是二进制日志而不是触发跟踪变化，我们认为这是更安全的选择，所以不必担心触发器的性能影响，也不必担心运行的是哪个MySQL版本（甚至能以基于语句的复制模式工作，但会抛出一些警告）。这已经在大规模部署中被证明。

那么什么时候pt-online-schema-change会是首选项？如果你正在运行许多已存在外键的旧数据库，并且删除外键是很困难的，那么会发现pt-online-schema-change已经尝试对外键提供更广泛的支持，但你必须承担为数据完整性和正常运行时间选择最安全选项的认知负担。此外，gh-ost利用二进制日志来完成其工作，因此如果由于某种原因无法访问这些日志，pt-online-schema-change仍然是一个可行的选项。

理想的情况是，有一天我们都可以在MySQL中原生进行在线schema更改，但这一天还没有到来。在此之前，开源生态系统在使schema更改成为更容易自动化的过程方面已经走了

很长的路。让我们讨论一下如何将所有这些工具组合在一起，形成一个用于schema更改的完整的CI/CD管道。

用于**schema**更改的**CI/CD**管道

至此，我们已经介绍了许多工具，从帮助管理schema定义版本的工具，到在生产中以最短的停机时间进行更改的工具，可以看到，我们已经具备了对schema更改进行全面持续集成和部署的能力，可以消除组织中工程师生产力的巨大瓶颈。让我们把下面列出的这一切放在一起。

对schema源代码控制进行组织

首先最重要的是，必须在代码存储库中单独分离每个数据库集群的schema定义。如果这里的目标是为不同的团队以不同的速度运行他们的更改提供灵活性，那么将所有数据库的所有schema定义合并到一个代码存储库中是没有意义的。这种分离还允许每个团队在代码存储库中定义不同的代码校验。一些团队可能需要非常特别的字符集和校验集，而其他团队可能可以接受默认设置。对于你的合作伙伴团队来说，灵活性是关键。

确保记录了特性开发团队的工程师如何从笔记本电脑上的schema更改到在所有环境中运行并在投入生产前运行测试的工作流程。这里的“pull-request”模型非常有用，可以帮助每个团队在更多环境或生产环境中升级和运行更改之前且以自动化的方式定义在请求schema更改时要运行的测试。

安全的基线配置

为选择的在线schema更改工具定义基线配置。假设你是一个团队，为依赖你提供灵活、可伸缩且安全的解决方案的合作伙伴团队提供工具。当你考虑如何实现在线schema更改工具时，需要分析schema设计的一些考虑因素，它们需要作为测试整体schema更改的代码合并请求的一部分。例如，如果你更喜欢gh-ost的安全性及其无触发器设计，这意味着你必须拥有一个没有外键的数据库平台。不考虑权衡的选择，如果最终决定使用“死亡外键”，那么应该确保在预提交hook或在Skeema代码存储库中测试schema更改的方式中对其进行了编码，这样就可以避免意外地将不希望的schema更改引入生产环境。类似地，你应该为在线schema更改工具确定一个基本配置，该工具为更改如何在生产环境中运行提供了一个基本的安全网。在这样的配置中，你可能需要引入的例子包括运行的最大MySQL线程或允许的最大系统负载。当任何一个特性开发团队创建一个新的数据库，并希望用一个代码存储库来跟踪和管理schema更改时，代码存储库模板可以成为一个强大的工具，使正确的事情变得容易。

每个团队的管道灵活性

在每个数据库的代码存储库中组织schema定义时，为每个拥有该数据库的团队提供了最大的灵活性，以决定如何自动化或人工管理其管道。一个团队可能处于新产品的迭代阶段，只要定义的测试通过，就可以自动升级schema的代码合并请求。另一个团队可能拥有更关键的任务数据库，需要更谨慎的方法，它们倾向于在CI系统将其推进到下一个环境之前，由操作员批准代码合并请求。

在设计组织如何实现规模化的schema更改部署时，请关注最终目标：为不断增长的工程组织提供速度和安全性，而不让数据库工程团队成为公司在生产中从想法转变为功能的瓶

颈。

小结

好的schema设计是非常普遍的，但是MySQL有特殊的实现细节要考虑。简而言之，让事情尽可能小而简单是一个好主意。MySQL喜欢简单，需要使用数据库的人也喜欢简单。请记住以下指导原则：

- 尽量避免在设计中出现极端情况，例如，强制执行非常复杂的查询或者包含很多列的表设计（很多的意思是介于有点多和非常多之间）。
- 使用小的、简单的、适当的数据类型，并避免使用NULL，除非确实是对真实数据进行建模的正确方法。
- 尝试使用相同的数据类型来存储相似或相关的值，尤其是在联接条件中使用这些值时。
- 注意可变长度字符串，它可能会导致临时表和排序的全长内存分配不乐观。
- 如果可能的话，尝试使用整数作为标识符。
- 避免使用一些传统的MySQL技巧，例如，指定浮点数的精度或整数的显示宽度。
- 小心使用ENUM和SET类型。它们很方便，但也可能被滥用，有时还很棘手。另外最好避免使用BIT类型。

数据库设计是一门科学。如果你非常关注数据库设计，可考虑使用专用的源材料。[8]

还请记住，你的schema将随着业务需求和用户数据而发展，这意味着拥有一个强大的软件生命周期来管理schema更改是使这种发展在组织中安全且可扩展的关键部分。

---

[1] 请记住，字符串长度定义的不是字节数，是字符数。多字节字符集可能需要多个字节来存储**1**个字符。

[2] 如果需要在检索后保持值不变，请小心使用**BINARY**类型，**MySQL**会使用**\0**将其填充到需要的长度。

[3] 这里显示的速度是相对的，因为**CPU**、内存和其他硬件的速度会随着时间的变化而变化。

[4] **TIMESTAMP**的行为规则很复杂，并且在不同的**MySQL**版本中会发生变化，因此你应该验证数据库的行为是否符合需要。在对**TIMESTAMP**列进行更改后，通常最好检查**SHOW CREATE TABLE**命令的输出。

[5] 如果使用**InnoDB**存储引擎，除非数据类型完全匹配，否则可能无法创建外键，对应的错误消息是**“ERROR 1005（HY000）：Can't create table”**。这个信息可能让人困惑，具体取决于上下文，**MySQL**邮件列表中经常会出现相关问题。（奇怪的是，可以在不同长度的**VARCHAR**列之间创建外键。）

[6] 另一方面，对于一些有很多写入的非常大的表，这种伪随机值实际上可以帮助消除**“**热点**”**。

[7] 更多信息请参阅**MySQL**文档（参见链接**29**）。

[8] 要进行深入学习，请参阅**Michael J.Hernandez**（**Pearson**）的***Database Design for Mere Mortals***。

第7章 创建高性能的索引

索引，在MySQL中也叫作键（key），是存储引擎用于快速找到记录的一种数据结构。本章将讨论索引的一些有用的特性。

要想获得好的性能，索引至关重要。尤其是当表中的数据量越来越大时，索引对性能的影响愈发重要。在数据量较小且负载较低时，缺少合适的索引对性能的影响可能还不明显，但当数据量逐渐增大时，性能会急剧下降。[1]不过，索引却经常被忽略，有时候甚至被误解，所以在实际案例中，经常会遇到由糟糕索引导致的问题。这也是为什么我们把本章放在了全书靠前的位置，甚至比查询优化还要靠前。

索引优化应该是对查询性能优化最有效的手段了。索引能够轻易将查询性能提高几个数量级，“最优”的索引有时比一个“好的”索引性能要好两个数量级。创建一个真正“最优”的索引经常需要重写查询，所以，本章和下一章的关系非常紧密。

本章使用了示例数据库Sakila（参见链接32），其可以在MySQL的官方站点获取。Sakila是一个基于出租商店模型的示例数据库，包含演员、电影、顾客等信息。

索引基础

要理解MySQL的索引原理，最简单的方法就是去看一本书的“索引”部分：如果想在一本书中找到某个特定主题，一般会先看书的索引，找到对应的页码。

在MySQL中，存储引擎用类似的方法使用索引。先在索引结构中找到对应值，这样就可以找到包含这个值的记录。假如要运行下面的查询：

```
SELECT first_name FROM sakila.actor WHERE actor_id = 5;
```

如果在actor_id列上建了索引，则MySQL将使用该索引找到actor_id为5的记录。也就是说，MySQL先在索引上按值进行查找，然后返回所有包含该值的记录。

索引可以包含一列或多列的值。如果索引包含多列，那么列的顺序也十分重要，因为MySQL只能有效地使用索引的最左前缀列。创建一个包含两列的索引，和创建两个只包含一列的索引是大不相同的，下面将详细介绍。

> 如果使用了**ORM**工具，是否还需要关注索引？
>
> 简而言之：是，仍然需要理解索引，即使是使用对象关系映射（ORM）工具。
>
> ORM工具能够生成符合逻辑的、合法的查询（多数时候），除非只是生成非常基本的查询（例如，仅是根据主键查询），否则它很难生成适合索引的查询。在精妙和复杂的索引面前，无论ORM工具多么精巧，都不要对其抱太大希望。读完本章后面的内容，你就会同意这个观点！很多时候，即使是查询优化技术专家也很难兼顾到各种情况，更别说ORM了。

索引的类型

索引有很多种类型，可以为不同的场景提供更好的性能。在MySQL中，索引是在存储引擎层而不是服务器层实现的。所以，并没有统一的索引标准：不同存储引擎的索引的工作方式并不一样，也不是所有的存储引擎都支持所有类型的索引。即使多个存储引擎支持同一种类型的索引，其底层的实现也可能不同。本书假设所有的表都使用InnoDB引擎，并会重点介绍InnoDB引擎的索引实现。

下面，先来看两种MySQL中最常用的索引类型，看看它们的优点和缺点。

B-tree索引

当人们谈论索引的时候，如果没有特别指明类型，那么多半说的是*B-tree*索引，它使用B-tree数据结构来存储数据。[2]大多数MySQL引擎都支持这种索引。

本书使用术语*B-tree*来代表这类索引结构，是因为，MySQL在CREATE TABLE和其他语句中也是使用该关键字的。不过，底层的存储引擎在内部实现上可能会略有不同。例如，NDB集群存储引擎虽然依然使用了BTREE标识，但在其内部实际上使用了T-tree结构存储这种索引，InnoDB则使用的是B+tree。这种数据结构和算法的变种不在本书的讨论范围之内。

B-tree通常意味着所有的值都是按顺序存储的，并且每一个叶子页到根的距离相同。图7-1所示的是B-tree索引的抽象表示，大致反映了InnoDB索引是如何工作的。

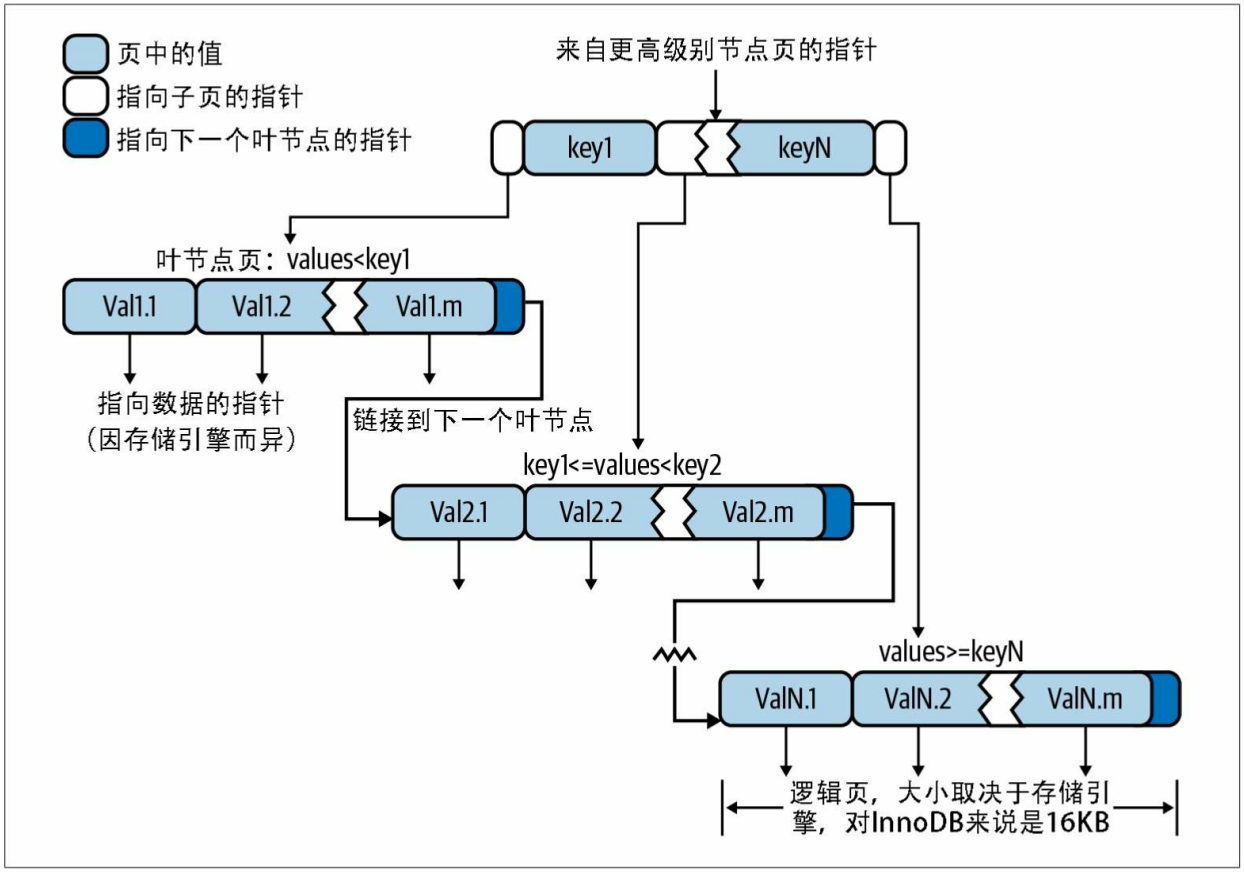

图7-1：建立在B-tree结构（准确地说是B+tree）上的索引

B-tree索引能够加快数据访问的速度，这是因为有了索引，在查询某些条件的数据时，存储引擎不再需要进行全表扫描。而是从索引的根节点（图中并未画出）开始进行搜索，根节点的槽中存放了指向子节点的指针，存储引擎根据这些指针向下层查找。通过比较节点页的值和要查找的值可以找到合适的指针进入下层子节点，这些指针实际上定义了子节点页中值的上限和下限。最终存储引擎要么找到对应的值，要么该记录不存在。

叶子节点比较特殊，它们的指针指向的是被索引的数据，而不是其他的节点页（不同存储引擎的“指针”类型不同）。图7-1中仅绘制了一个节点和其对应的叶子节点，其实在根节点和叶子节点之间可能有很多层节点页。树的深度和表的大小直接相关。

B-tree是按照索引列中的数据大小顺序存储的，所以很适合按照范围来查询。例如，在一个基于文本列的索引树上遍历，按字母顺序传递连续的值进行范围查找是非常合适的，所以，像“找出所有以I到K开头的名字”这样的查找效率会非常高。

假设有如下数据表：

```
CREATE TABLE People (
 last_name varchar(50) not null,
 first_name varchar(50) not null,
 dob date not null,
 key(last_name, first_name, dob)
);
```

对于表中的每一行数据，索引中都包含对应的last_name、first_name和dob列的值，图7-2显示了该索引是如何组织数据存储的。

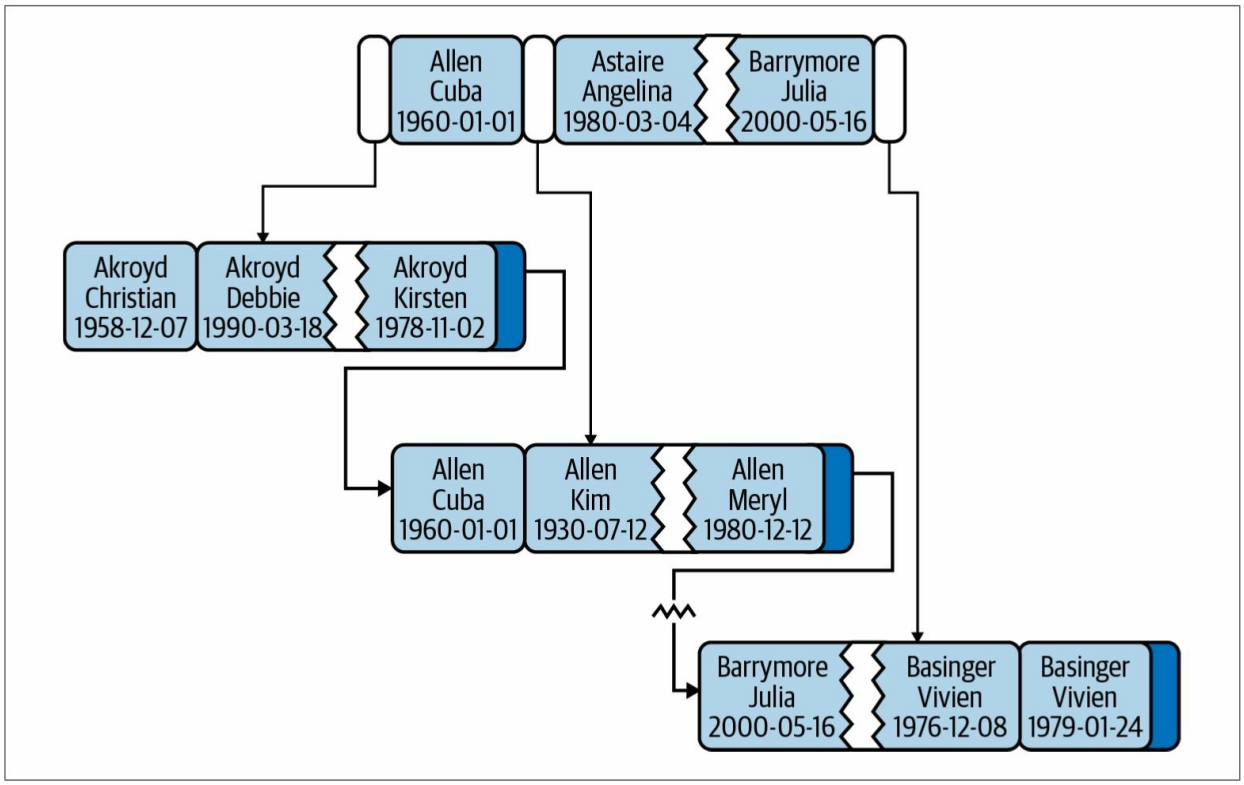

图7-2：B-tree（从技术上来说是B+tree）索引树中的部分条目示例

请注意，索引对多个值进行排序的依据是CREATE TABLE语句中定义索引时列的顺序。看一下最后两个条目，两个人的姓和名都一样，这时，则根据他们的出生日期来排列。

自适应哈希索引。InnoDB存储引擎有一个被称为自适应哈希索引的特性。当InnoDB发现某些索引值被非常频繁地被访问时，它会在原有的B-tree索引之上，在内存中再构建一个哈希索引。这就让B-tree索引也具备了一些哈希索引的优势，例如，可以实现非常快速的哈希查找。这个过程是完全自动化的，用户无法进行控制或者配置。不过，可以通过参数彻底关闭自适应哈希索引这个特性。

可以使用B-tree索引的查询类型。B-tree索引适用于全键值、键值范围或键前缀查找。其中键前缀查找只适用于根据最左前缀的查找。[3]前面所述的索引对如下类型的查询有效。

全值匹配

全值匹配指的是和索引中的所有列匹配，例如，前面提到的索引可用于查找姓名为Cuba Allen、出生于1960-01-01的人。

匹配最左前缀

例如，前面提到的索引可用于查找所有姓为Allen的人，即只使用索引的第一列。

匹配列前缀

也可以只匹配某一列的值的开头部分。例如，前面提到的索引可用于查找所有姓以J开头的人。这里也只使用了索引的第一列。

匹配范围值

例如，前面提到的索引可用于查找姓在Allen和Barrymore之间的人。这里也只使用了索引的第一列。

精确匹配某一列而范围匹配另外一列

前面提到的索引也可用于查找所有姓为Allen，并且名字是字母K开头（比如Kim、Karl等）的人，即第一列last_name是全匹配，第二列first_name是范围匹配。

只访问索引的查询

B-tree索引通常可以支持“只访问索引的查询”，即查询只需要访问索引，而无须访问数据行。后面我们将单独讨论这种“覆盖索引”的优化。

因为索引树中的节点是有序的，所以除了按值查找，索引还可以用于查询中的ORDER BY操作（按顺序查找）。一般来说，如果B-tree可以按照某种方式查找到值，那么也可以按照这种方式去排序。所以，如果ORDER BY子句满足前面列出的几种查询类型，则这个索引也可以用于这类排序场景。

下面是一些关于B-tree索引的限制。

- 如果不是按照索引的最左列开始查找，则无法使用索引。例如，上面例子中的索引无法用于查找名字为Bill的人，也无法查找某个特定生日的人，因为这两列都不是最左数据列。类似地，也无法查找姓氏以某个字母结尾的人。
- 不能跳过索引中的列。也就是说，前面所述的索引无法用于查找姓为Smith并且在某个特定日期出生的人。如果不指定名（first_name），则MySQL只能使用索引的第一列。
- 如果查询中有某列的范围查询，则其右边所有列都无法使用索引优化查找。例如，有查询WHERE last_name='Smith'AND first_name LIKE'J%'AND dob='1976-12-23'，这个查询只能使用索引的前两列，因为这里LIKE是一个范围条件（不过，MySQL可以把其余列用于其他目的）。如果范围查询列值的数量有限，那么可以通过使用多个等于条件来代替范围条件。

现在可以看到前面提到的索引列的顺序是多么重要：这些限制都和索引列的顺序有关。在优化性能的时候，可能需要使用相同的列但顺序不同的索引来满足不同类型的查询需求。

也有些限制并不是B-tree本身导致的，而是MySQL查询优化器和存储引擎使用索引的方式导致的，这部分限制在未来的版本中可能就不再是限制了。

全文索引

FULLTEXT是一种特殊类型的索引，它查找的是文本中的关键词，而不是直接比较索引中的值。全文索引和其他几类索引的匹配方式完全不一样。它有许多需要注意的细节，如停用词、词干、复数、布尔搜索等。全文索引更类似于搜索引擎做的事情，而不是简单的

WHERE条件匹配。

在相同的列上同时创建全文索引和基于值的B-tree索引并不会有冲突，全文索引适用于MATCH AGAINST操作，而不是普通的WHERE条件操作。

使用索引的优点

索引可以让服务器快速地定位到表的指定位置。但是这并不是索引的唯一作用，到目前为止可以看到，根据创建索引的数据结构不同，索引也有一些其他的附加作用。

最常见的B-tree索引，按照顺序存储数据，所以MySQL可以用来做ORDER BY和GROUP BY操作。因为数据是有序的，所以B-tree也就会将相关的列值都存储在一起。最后，因为索引中存储了实际的列值，所以某些查询只使用索引就能够完成全部查询。据此特性，总结下来索引有如下三个优点：

- 索引大大减少了服务器需要扫描的数据量。
- 索引可以帮助服务器避免排序和临时表。
- 索引可以将随机I/O变为顺序I/O。

“索引”这个主题完全值得单独写一本书，如果想深入理解这部分内容，强烈建议阅读由Tapio Lahdenmaki和Mike Leach编写的*Relational Database Index Design and the Optimizers*（Wiley出版社出版）一书，该书详细介绍了如何计算索引的成本和作用、如何评估查询速度、如何分析维护索引的代价和其带来的好处等。

Lahdenmaki和Leach在书中还引入了一个“三星系统”（three-star system）评价体系，用以判断一个索引是不是适合某个查询语句：索引将相关的记录放到一起则获得“一星”；如果索引中的数据顺序和查找中的排列顺序一致则获得“二星”；如果索引中的列包含了查询中需要的全部列则获得“三星”。后面我们将会介绍这些原则。

高性能的索引策略

正确地创建和使用索引是实现高性能查询的基础。前面已经介绍了各种类型的索引及其对应的优缺点，现在我们一起来看看如何真正地发挥这些索引的优势。

高效地选择和使用索引有很多种方式，其中有些是针对特殊案例的优化方法，有些则是针对特定行为的优化。[4]使用哪个索引，以及如何评估选择不同索引对性能的影响，则需要持续不断地学习。接下来的几节将帮助读者理解如何高效地使用索引。

前缀索引和索引的选择性

有时候为了提升索引的性能，同时也节省索引空间，可以只对字段的前一部分字符进行索引，这样做的缺点是，会降低索引的选择性。索引的选择性是指，不重复的索引值（也称为基数，cardinality）和数据表的记录总数（＃T）的比值，范围从1/＃T到1之间。索引的选择性越高则查询效率越高，因为选择性高的索引可以让MySQL在查找时过滤掉更多的行。唯一索引的选择性是1，这是最好的索引选择性，性能也是最好的。

一般情况下，列前缀的选择性也是足够高的，足以满足查询性能。对于BLOB、TEXT或者很长的VARCHAR类型的列，必须使用前缀索引，因为MySQL并不支持对这些列的完整内容进行索引。

这里的关键点在于，既要选择足够长的前缀以保证较高的选择性，同时又不能太长（以便节约空间）。前缀应该足够长，以使得前缀索引的选择性接近于索引整列。换句话说，前缀的“基数”应该接近于完整列的“基数”。

为了确定前缀的合适长度，需要找到最常见的值的列表，然后和最常见的前缀列表进行比较。在示例数据库Sakila中并没有合适的例子，所以我们从表city中生成一个示例表，这样就有足够的数据进行演示了：

```
CREATE TABLE sakila.city_demo(city VARCHAR(50) NOT NULL);
INSERT INTO sakila.city_demo(city) SELECT city FROM sakila.city;
-- Repeat the next statement five times:
INSERT INTO sakila.city_demo(city) SELECT city FROM sakila.city_demo;
-- Now randomize the distribution (inefficiently but conveniently):
UPDATE sakila.city_demo
 SET city = (SELECT city FROM sakila.city ORDER BY RAND() LIMIT 1);
```

现在我们有了示例数据集。数据分布当然不是真实的分布，由于我们使用了RAND()函数，所以你的结果会与此不同，但对这个练习来说这并不重要。首先，找到最常见的城市列表：

```
mysql> SELECT COUNT(*) AS c, city
    -> FROM sakila.city_demo
    -> GROUP BY city ORDER BY c DESC LIMIT 10;
+-----+---------------+
| c   | city          |
+-----+---------------+
| 65 | London         |
| 49 | Hiroshima      |
| 48 | Teboksary      |
| 48 | Pak Kret       |
| 48 | Yaound         |
| 47 | Tel Aviv-Jaffa |
| 47 | Shimoga        |
| 45 | Cabuyao        |
| 45 | Callao         |
| 45 | Bislig         |
+-----+---------------+
```

注意，上面的每个值都出现了45～65次。现在查找到最频繁出现的城市前缀，先从3个前缀字母开始：

```
mysql> SELECT COUNT(*) AS c, LEFT(city, 3) AS pref
    -> FROM sakila.city_demo GROUP BY pref ORDER BY cc DESC LIMIT 10;
+-----+------+
| c   | pref |
+-----+------+
| 483 | San  |
| 195 | Cha  |
| 177 | Tan  |
| 167 | Sou  |
| 163 | al-  |
| 163 | Sal  |
| 146 | Shi  |
| 136 | Hal  |

| 130 | Val  |
| 129 | Bat  |
+-----+------+
```

每个前缀都比原来的城市名出现的次数更多，因此唯一前缀比唯一城市名要少得多。然后，我们增加前缀长度，直到这个前缀的选择性接近完整列的选择性。经过实验后发现，前缀长度为7时比较合适：

```
mysql> SELECT COUNT(*) AS c, LEFT(city, 7) AS pref
    -> FROM sakila.city_demo GROUP BY pref ORDER BY c DESC LIMIT 10;
+-----+---------+
| c   | pref    |
+-----+---------+
| 70  | Santiag |
| 68  | San Fel |
| 65  | London  |
| 61  | Valle d |
| 49  | Hiroshi |
| 48  | Teboksa |
| 48  | Pak Kre |
| 48  | Yaound  |
| 47  | Tel Avi |
| 47  | Shimoga |
+-----+---------+
```

计算合适的前缀长度的另外一个办法就是计算完整列的选择性，并使前缀的选择性接近完整列的选择性。下面显示如何计算完整列的选择性：

```
mysql> SELECT COUNT(DISTINCT city)/COUNT(*) FROM sakila.city_demo;
+------------------------------+
| COUNT(DISTINCT city)/COUNT(*) |
+------------------------------+
|                       0.0312 |
+------------------------------+
```

通常来说（尽管也有例外情况），在这个例子中，如果前缀的选择性能够接近0.031，基本上就可以用了。可以在一个查询中针对不同前缀长度进行计算，这对于大表非常有用。下面给出了如何在同一个查询中计算不同前缀长度的选择性：

```
mysql> SELECT COUNT(DISTINCT LEFT(city, 3))/COUNT(*) AS sel3,
    -> COUNT(DISTINCT LEFT(city, 4))/COUNT(*) AS sel4,
    -> COUNT(DISTINCT LEFT(city, 5))/COUNT(*) AS sel5,
    -> COUNT(DISTINCT LEFT(city, 6))/COUNT(*) AS sel6,
    -> COUNT(DISTINCT LEFT(city, 7))/COUNT(*) AS sel7
    -> FROM sakila.city_demo;
+--------+--------+--------+--------+--------+
| sel3   | sel4   | sel5   | sel6   | sel7   |
+--------+--------+--------+--------+--------+
| 0.0239 | 0.0293 | 0.0305 | 0.0309 | 0.0310 |
+--------+--------+--------+--------+--------+
```

查询显示，当前缀长度到达7的时候，再增加前缀长度，选择性提升的幅度已经很小了。

只看平均选择性是不够的，还有例外的情况，需要考虑最坏情况下的选择性。平均选择性会让你认为前缀长度为4或者5的索引已经足够了，但如果数据分布很不均匀，可能就会有陷阱。观察前缀为4的最常出现城市的次数，可以看到明显不均匀：

```
mysql> SELECT COUNT(*) AS c, LEFT(city, 4) AS pref
    -> FROM sakila.city_demo GROUP BY pref ORDER BY c DESC LIMIT 5;
+-----+------+
| c   | pref |
+-----+------+
| 205 | San  |
| 200 | Sant |
| 135 | Sout |
| 104 | Chan |
|  91 | Toul |
+-----+------+
```

如果前缀是4字节，则最常出现的前缀出现的次数比最常出现的城市出现的次数要多很多，即这些值的选择性比平均选择性要低。如果有比这个随机生成的示例更真实的数据，就更有可能看到这种现象。例如，在真实的城市名上建一个长度为4的前缀索引，对于以“San”和“New”开头的城市的选择性就会非常糟糕，因为很多城市都以这两个词开头。

在上面的示例中，已经找到了合适的前缀长度，下面演示一下如何创建前缀索引：

```
ALTER TABLE sakila.city_demo ADD KEY (city(7));
```

前缀索引是一种能使索引更小、更快的有效办法，但它也有缺点：MySQL无法使用前缀索引做ORDER BY和GROUP BY操作，也无法使用前缀索引做覆盖扫描。

一个常见的场景是针对很长的十六进制唯一ID使用前缀索引。在前面的章节中已经讨论了很多有效的技术来存储这类ID信息，但如果业务系统使用了某个整体的解决方案，因而无法修改存储结构，那该怎么办？例如，使用vBulletin或者其他基于MySQL的应用在存储网站的会话（SESSION）时，需要在一个很长的十六进制字符串上创建索引。此时，如果采用长度为8的前缀索引通常能显著地提升性能，并且这种方法对上层应用完全透明。

多列索引

很多人对多列索引的理解都不够。一个常见的错误就是，为每列创建独立的索引，或者按照错误的顺序创建多列索引。

我们会在接下来的章节中单独讨论索引列的顺序问题。先来看第一个常见问题：为每列创建独立的索引。从SHOW CREATE TABLE中很容易看到这种情况：

```
CREATE TABLE t (
 c1 INT,
 c2 INT,
 c3 INT,
 KEY(c1),
 KEY(c2),
 KEY(c3)
);
```

这种索引策略，一般是由于大家会听到一些“专家”这样模糊建议：“把WHERE条件里面的

列都建上索引”。实际上这种建议是错误的，这样一来，在最好的情况下也只能是“一星”索引，其性能比起真正最优的索引可能差几个数量级。有时如果无法设计一个“三星”索引，那么不如忽略掉WHERE子句，集中精力优化索引列的顺序，或者创建一个全覆盖索引。

在多列上独立地创建多个单列索引，在大部分情况下并不能提高MySQL的查询性能。MySQL引入了一种叫“索引合并”（index merge）的策略，它在一定程度上可以使用表中的多个单列索引来定位指定的行。在这种情况下，查询能够同时使用两个单列索引进行扫描，并将结果进行合并。这种算法有三个变种：OR条件的联合（union），AND条件的相交（intersection），组合前两种情况的联合及相交。下面的查询就使用了两个索引扫描的联合，通过EXPLAIN中的Extra列可以看到这点：

```
mysql> EXPLAIN SELECT film_id, actor_id FROM sakila.film_actor
    -> WHERE actor_id = 1 OR film_id = 1\G
*************************** 1. row ***************************
 id: 1
 select_type: SIMPLE
 table: film_actor
 partitions: NULL
 type: index_merge
 possible_keys: PRIMARY,idx_fk_film_id
 key: PRIMARY,idx_fk_film_id
 key_len: 2,2
 ref: NULL
 rows: 29
 filtered: 100.00
 Extra: Using union(PRIMARY,idx_fk_film_id); Using where
```

MySQL会使用这类技术来优化复杂查询，所以在某些语句的Extra列中还可以看到嵌套操作。

索引合并策略有时候效果非常不错，但更多的时候，它说明了表中的索引建得很糟糕：

- 当优化器需要对多个索引做相交（相交操作是使用“索引合并”的一种情况，另一种是做联合操作）操作时（通常有多个AND条件），通常意味着需要一个包含所有相关列的多列索引，而不是多个独立的单列索引。
- 当优化器需要对多个索引做联合操作时（通常有多个OR条件），通常需要在算法的缓存、排序和合并操作上耗费大量CPU和内存资源，尤其是当其中有些索引的选择性不高，需要合并扫描返回的大量数据的时候。
- 更重要的是，优化器不会把这些操作计算到“查询成本”（cost）中，优化器只关心随机页面读取。这会使得查询的成本被“低估”，导致该执行计划还不如直接进行全表扫描。这样做不但会消耗更多的CPU和内存资源，还可能会影响并发的查询，但如果单独运行这样的查询则往往会忽略对并发性的影响。通常来说，使用UNION改写查询，往往是最好的办法。

如果在EXPLAIN中看到有索引合并，那么就应该好好检查一下查询语句的写法和表的结构，看是不是已经是最优的。也可以通过参数optimizer_switch来关闭索引合并功能，还可

以使用IGNORE INDEX语法让优化器强制忽略掉某些索引，从而避免优化器使用包含索引合并的执行计划。

选择合适的索引列顺序

最容易让人感到困惑的问题之一就是索引列的顺序。正确的顺序依赖于使用该索引的查询语句，同时，还需要考虑如何更好地满足排序和分组操作的需要。

在一个多列B-tree索引中，索引列的顺序意味着索引首先按照最左列进行排序，其次是第二列，等等。所以索引可以按照升序或者降序进行扫描，以满足精确符合列顺序的ORDER BY、GROUP BY和DISTINCT等子句的查询需求。

所以，多列索引的列顺序至关重要。在Lahdenmaki和Leach的“三星索引”系统中，列顺序还决定了一个索引是否能够成为一个真正的“三星索引”，在本章的后续部分我们将通过更多的例子来说明这一点。

对于如何选择索引的列顺序有一个重要的经验法则：将选择性最高的列放到索引最前列。这个建议准确吗？在很多场景中可能有帮助，但是要全面地考虑各种场景的话，考虑如何避免大量随机I/O和排序可能更重要。（场景不同则选择不同，没有一个放之四海皆准的法则。这里只是说明，这个经验法则可能没有你想象中那么重要。）

当不需要考虑排序和分组时，将选择性最高的列放在前面通常是很好的。这时索引的作用只是优化查询语句中的WHERE条件。在这种情况下，按这个原则设计的索引确实能够最快地过滤出需要的行，对于在WHERE子句中只使用了索引部分前缀列的查询来说，选择性也更高。然而，性能不只依赖于所有索引列的选择性（整体基数），也和查询条件的具体值有关，也就是和值的分布有关。这和前面介绍的选择前缀的长度需要考虑的因素一样。可能需要根据那些运行频率最高的查询来调整索引列的顺序，让这种情况下索引的选择性最高。

以下面的查询为例：

```
SELECT * FROM payment WHERE staff_id = 2 AND customer_id = 584;
```

是应该创建一个（staff_id、customer_id）索引还是应该颠倒一下顺序？这时，可以通过运行某些查询来确定在这个表中值的分布情况，并确定哪列的选择性更高。先用下面的查询预测一下，看看各个WHERE条件的分支对应的数据基数有多大：

```
mysql> SELECT SUM(staff_id = 2), SUM(customer_id = 584) FROM payment\G
*************************** 1. row ***************************
 SUM(staff_id = 2): 7992
SUM(customer_id = 584): 30
```

根据前面的经验法则，应该将索引列customer_id放到前面，因为对应条件值的customer_id数量更小。我们再来看看对于这个customer_id的条件值，对应的staff_id列的选择性如何：

```
mysql> SELECT SUM(staff_id = 2) FROM payment WHERE customer_id = 584\G
*************************** 1. row ***************************
SUM(staff_id = 2): 17
```

这样做有一个地方需要注意，查询的结果非常依赖于选定的具体值。如果按上述办法优化，可能对其他一些条件值的查询不公平，服务器的整体性能可能会变得更糟，或者其他某些查询的运行变得不如预期。

如果是从诸如*pt-query-digest*这样的工具的报告中提取“最差”查询，那么再按上述办法选定索引顺序，往往可以获得更好的性能。如果没有运行类似的查询来确认实际的情况，那么最好还是按经验法则来做，因为经验法则考虑的是全局基数和选择性，而不是某个具体查询：

```
mysql> SELECT COUNT(DISTINCT staff_id)/COUNT(*) AS staff_id_selectivity,
    -> COUNT(DISTINCT customer_id)/COUNT(*) AS customer_id_selectivity,
    -> COUNT(*)
    -> FROM payment\G
*************************** 1. row ***************************
 staff_id_selectivity: 0.0001
 customer_id_selectivity: 0.0373
 COUNT(*): 16049
```

customer_id的选择性更高，所以答案是将其作为索引的第一列：

```
ALTER TABLE payment ADD KEY(customer_id, staff_id);
```

和前缀索引例子中描述的情况一样，当使用前缀索引的时候，在某些条件值的基数比正常值高的时候，问题就来了。例如，在某些应用程序中，对于没有登录的用户，都将其用户名记录为“guset”，在记录用户行为的会话（session）表和其他记录用户活动的表中，“guest”就成为一个特殊用户ID，一旦查询涉及这个用户，那么和对于正常用户的查询就大不相同了，因为通常有很多会话都是没有登录的。有些系统账号也会导致类似的问题。一个应用通常都有一个特殊的管理员账号，和普通账号不同，它并不是一个具体的用户，系统中所有的其他用户都是这个用户的好友，所以系统往往通过它向网站的所有用户发送状态通知和其他消息。这个账号的巨大的好友列表很容易导致网站出现服务器性能问题。

这在实际中是一个非常典型的问题。对于任何异常用户，不仅那些用于管理应用的设计糟糕的账号会有同样的问题，那些拥有大量好友、图片、状态、收藏的用户，也会有前面提到的与系统账号一样的问题。

下面是一个我们遇到的真实案例。在一个商品购买和经验分享的用户论坛系统中，这个特殊表中的查询运行得非常慢：

```
SELECT COUNT(DISTINCT threadId) AS COUNT_VALUE
FROM Message
WHERE (groupId = 10137) AND (userId = 1288826) AND (anonymous = 0)
ORDER BY priority DESC, modifiedDate DESC
```

这个查询看似没有建立合适的索引，所以客户咨询我们是否可以优化。EXPLAIN的结果如下：

```
id: 1
select_type: SIMPLE
table: Message
type: ref
key: ix_groupId_userId
key_len: 18
ref: const,const
rows: 1251162
Extra: Using where
```

MySQL为这个查询选择了索引（groupId，userId），如果不考虑列的基数，这看起来是一个非常合理的选择。但如果考虑一下user ID和group ID条件匹配的行数，你可能就会有不同的想法了：

```
mysql> SELECT COUNT(*), SUM(groupId = 10137),
    -> SUM(userId = 1288826), SUM(anonymous = 0)
    -> FROM Message\G
*************************** 1. row ***************************
 count(*): 4142217
 sum(groupId = 10137): 4092654
 sum(userId = 1288826): 1288496
 sum(anonymous = 0): 4141934
```

从上面的结果来看，符合组（groupId）条件的几乎满足表中的所有行，符合用户（userId）条件的有130万条记录，也就是说，索引基本上没什么用。因为这些数据是从其他应用中迁移过来的，迁移的时候把所有的消息都赋予了管理员组的用户。这个案例的解决办法是修改应用程序代码，区分这类特殊用户和组，禁止针对这类用户和组执行这个查询。

从这个小案例可以看到，经验法则和推论在多数情况下是有用的，但要注意，不要假设平均情况下的性能也能代表特殊情况下的性能，特殊情况可能会摧毁整个应用的性能。

最后，尽管关于选择性和基数的经验法则值得去研究和分析，但一定别忘了查询子句中的排序、分组和范围条件等其他因素，这些因素可能会对查询的性能造成非常大的影响。

聚簇索引

聚簇索引[5]并不是一种单独的索引类型，而是一种数据存储方式。具体的细节依赖于其实现方式，但InnoDB的聚簇索引实际上在同一个结构中保存了B-tree索引和数据行。

当表有聚簇索引时，它的数据行实际上存放在索引的叶子页（leaf page）中。术语“聚簇”表示数据行和相邻的键值紧凑地存储在一起。[6]因为无法同时把数据行存放在两个不同的地方，所以一个表只能有一个聚簇索引（不过，覆盖索引可以模拟多个聚簇索引的情况，本章后面将详细介绍）。

因为是存储引擎负责实现索引，因此，不是所有的存储引擎都支持聚簇索引。本节我们主要关注InnoDB，但是这里讨论的原理对于任何支持聚簇索引的存储引擎都是适用的。

图7-3展示了聚簇索引中的记录是如何存放的。注意，叶子页包含了一条记录的全部数

据，但是节点页只包含了索引列。在这个案例中，索引列包含的是整数值。

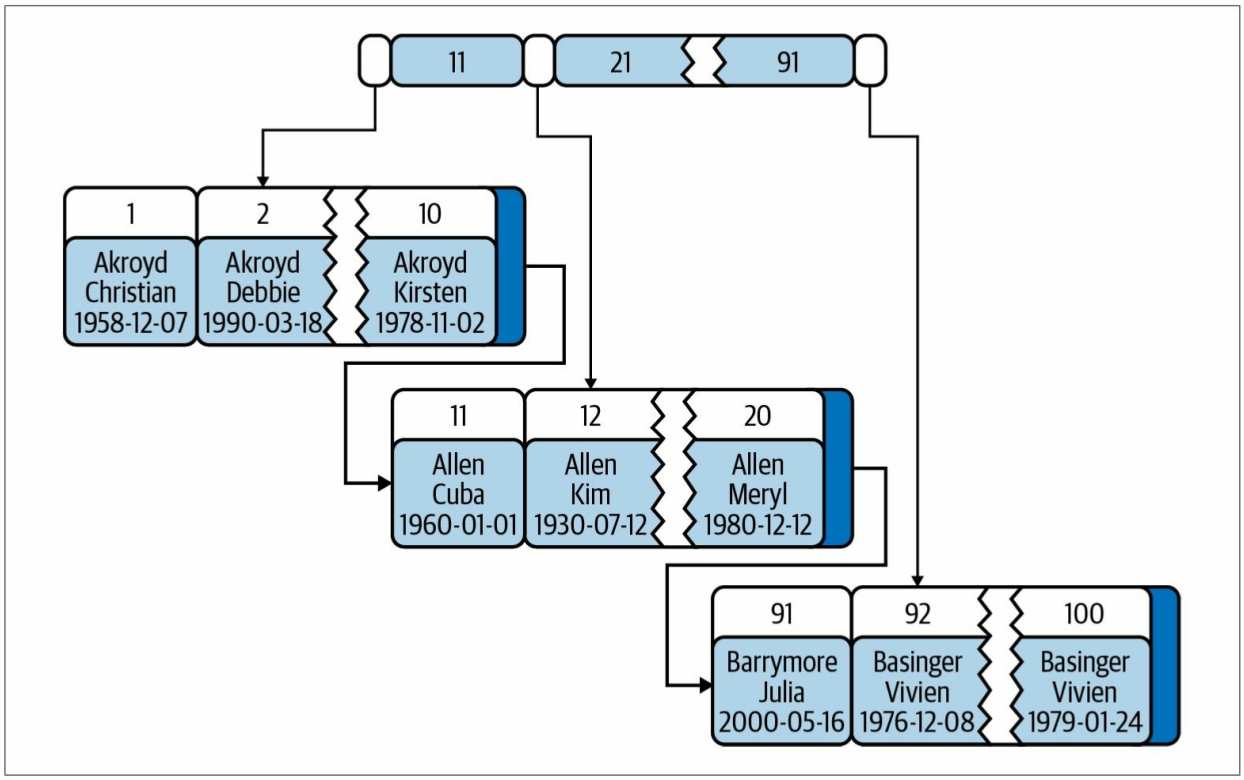

图7-3：聚簇索引的数据分布

有些数据库服务器允许你选择用于聚簇的索引，但是MySQL内置的存储引擎都不支持这个特性。InnoDB根据主键聚簇数据。这意味着图7-3中所示的“索引列”就是主键列。

如果你没有定义主键，InnoDB会选择一个唯一的非空索引代替。如果没有这样的索引，InnoDB会隐式定义一个主键来作为聚簇索引。这样做的缺点在于，所有需要使用这种隐藏主键的表都依赖一个单点的“自增值”，这可能会导致非常高的锁竞争，从而出现性能问题。

聚集的数据有一些重要的优点：

- 你可以把相互关联的数据保存在一起。例如，在实现电子邮箱应用时，可以根据用户ID来聚集数据，这样只需要从磁盘读取少数的数据页就能获取某个用户的全部邮件。如果没有使用聚簇索引，则每封邮件都可能导致一次磁盘I/O。
- 数据访问更快。聚簇索引将索引和数据保存在同一个B-tree中，因此从聚簇索引中获取数据通常比在非聚簇索引中查找要快。
- 使用覆盖索引扫描的查询可以直接使用页节点中的主键值。

如果在设计表和查询时能充分利用上面的优点，那么就能极大地提升性能。同时，聚簇索引也有一些缺点：

- 聚簇数据最大限度地提高了I/O密集型应用的性能，但如果数据全部都放在内存中，则访问的顺序就没那么重要了，聚簇索引也就没什么优势了。
- 插入速度严重依赖于插入顺序。按照主键的顺序插入行是将数据加载到InnoDB表

中最快的方式。但如果不是按照主键的顺序加载数据，那么在加载完成后最好使用OPTIMIZE TABLE命令重新组织一下表。

- 更新聚簇索引列的代价很高，因为它会强制InnoDB将每个被更新的行移动到新的位置。
- 基于聚簇索引的表在插入新行，或者主键被更新导致需要移动行的时候，可能面临页分裂（page split）的问题。当行的主键值要求必须将这一行插入某个已满的页中时，存储引擎会将该页分裂成两个页面来容纳该行，这就是一次页分裂操作。页分裂会导致表占用更多的磁盘空间。
- 聚簇索引可能导致全表扫描变慢，尤其是行比较稀疏，或者由于页分裂导致数据存储不连续的时候。
- 二级索引（非聚簇索引）可能比想象中的要更大，因为二级索引的叶子节点包含了引用行的主键列。
- 二级索引访问需要两次索引查找，而不是一次。

最后一点可能会让人有些疑惑，为什么二级索引需要两次索引查找？答案是，二级索引中保存的是“行指针”。要记住，二级索引叶子节点保存的不是指向行的物理位置的指针，而是行的主键值。

这意味着通过二级索引查找行，存储引擎需要找到二级索引的叶子节点，以获得对应的主键值，然后根据这个值去聚簇索引中查找对应的行。这里做了双倍工作：两次B-tree查找而不是一次。[7]对于InnoDB，自适应哈希索引（参考本章前面的“B-tree索引”一节）能够减少这样的重复工作。

InnoDB的数据分布

为了更好地理解聚簇索引，我们来看看下表的InnoDB的布局：

```
CREATE TABLE layout_test (
 col1 int NOT NULL,
 col2 int NOT NULL,

 PRIMARY KEY(col1),
 KEY(col2)
);
```

假设该表的主键取值为1～10000，按照随机顺序插入并使用OPTIMIZE TABLE命令做了优化。换句话说，数据在磁盘上的存储方式已经最优，但行的顺序是随机的。列col2的值是从1～100之间随机赋值的，所以有很多重复的值。

InnoDB以如图7-4所示的方式存储数据。

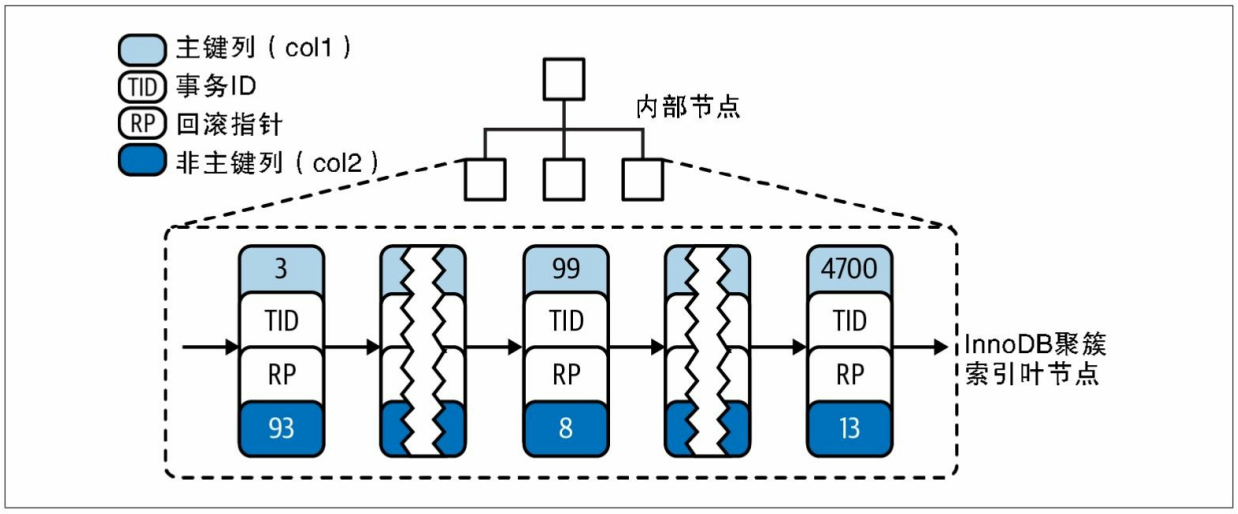

图7-4：InnoDB表layout test的主键物理存储分布

聚簇索引的每一个叶子节点都包含了主键值、事务ID、用于事务和MVCC的回滚指针，以及所有的剩余列（在这个例子中是col2）。如果主键是一个列前缀索引，InnoDB也会包含完整的主键列和剩下的其他列。

InnoDB的二级索引的叶子节点中存储的是主键值，并以此作为指向行的“指针”。这样的策略减少了当出现行移动或者数据页分裂时二级索引的维护工作。使用主键值作为指针会让二级索引占用更多的空间，换来的好处是，InnoDB在移动行时无须更新二级索引中的这个“指针”。

图7-5展示了示例表中的col2索引。每一个叶子节点都包含了索引列（这里是col2），紧接着是主键值（col1）。

图7-5展示了B-tree的叶子节点结构，但这里故意省略了非叶子节点这样的细节。InnoDB的非叶子节点包含了索引列和一个指向下级节点的指针（下一级节点可以是非叶子节点，也可以是叶子节点）。这对聚簇索引和二级索引都适用。

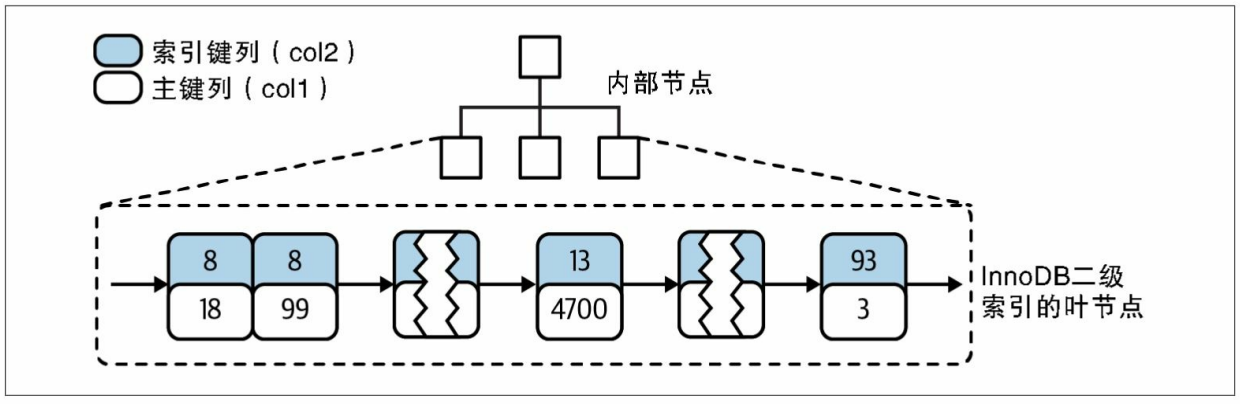

图7-5：InnoDB表layout test的二级索引分布

在**InnoDB**表中按主键顺序插入行

如果你正在使用InnoDB表并且没有什么数据需要聚集，那么可以定义一个代理键（surrogate key）作为主键，这种主键的值和应用无关，最简单的方法是使用

AUTO_INCREMENT自增列。这样可以保证数据行是按顺序写入的，对于根据主键做联接操作的性能也会更好。

最好避免随机的（不连续且值的分布范围非常大）聚簇索引，特别是对于I/O密集型的应用。例如，从性能的角度考虑，使用UUID作为聚簇索引会很糟糕：它使得聚簇索引的插入变得完全随机，这就是最糟糕的情况，数据本身没有任何聚集特性。

为了演示这一点，我们来做如下两个基准测试。第一个使用整数ID插入userinfo表：

```
CREATE TABLE userinfo (
 id int unsigned NOT NULL AUTO_INCREMENT,
 name varchar(64) NOT NULL DEFAULT '',
 email varchar(64) NOT NULL DEFAULT '',
 password varchar(64) NOT NULL DEFAULT '',
 dob date DEFAULT NULL,
 address varchar(255) NOT NULL DEFAULT '',
 city varchar(64) NOT NULL DEFAULT '',
 state_id tinyint unsigned NOT NULL DEFAULT '0',
 zip varchar(8) NOT NULL DEFAULT '',
 country_id smallint unsigned NOT NULL DEFAULT '0',
 gender ('M','F')NOT NULL DEFAULT 'M',
 account_type varchar(32) NOT NULL DEFAULT '',
 verified tinyint NOT NULL DEFAULT '0',
 allow_mail tinyint unsigned NOT NULL DEFAULT '0',
 parrent_account int unsigned NOT NULL DEFAULT '0',
 closest_airport varchar(3) NOT NULL DEFAULT '',
 PRIMARY KEY (id),
 UNIQUE KEY email (email),
 KEY country_id (country_id),
 KEY state_id (state_id),

 KEY state_id_2 (state_id,city,address)
) ENGINE=InnoDB
```

注意，使用了自增的整数ID作为主键。[8]

第二个例子是userinfo_uuid表。除了将主键改为UUID，其余的和前面的userinfo表完全相同：

```
CREATE TABLE userinfo_uuid (
 uuid varchar(36) NOT NULL,
 ...
```

我们测试了这两个表的设计。首先，在一个有足够内存容纳索引的服务器上向这两个表中各插入100万条记录。然后向这两个表继续插入300万条记录，使索引的大小超过服务器的内存容量。表7-1对测试结果做了比较。

表7-1：向InnoDB表中插入数据的测试结果

表	行数	时间（秒）	索引大小（MB）
userinfo	1,000,000	137	342
userinfo_uuid	1,000,000	180	544
userinfo	3,000,000	1233	1036
userinfo_uuid	3,000,000	4525	1707

注意，向UUID主键插入行不仅花费的时间更长，而且索引占用的空间也更大。一方面是由于主键字段更长；另一方面，无疑是由于页分裂和碎片导致的。

为了更加明白地说明为什么会这样，来看看向第一个表中插入数据时，索引发生了什么变化。图7-6展示了插满一个页面后继续插入相邻的下一个页面的场景。

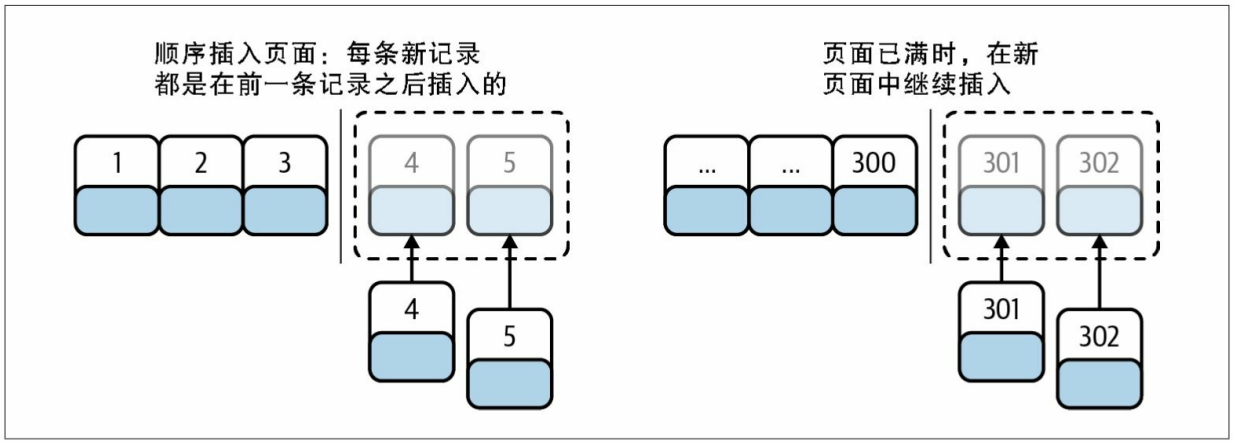

图7-6：向聚簇索引顺序地插入索引值

如图7-6所示，因为主键的值是顺序的，所以InnoDB把每一条记录都存储在上一条记录的后面。当达到页的最大填充因子时（InnoDB默认的最大填充因子是页大小的15/16，留出部分空间用于以后修改），下一条记录就会被写入新的页中。一旦数据按照这种顺序写入，主键页就会近似于被顺序的记录填满，这也正是所期望的结果（然而，二级索引页可能有所不同）。

对比一下向第二个使用了UUID聚簇索引的表中插入数据的情况，看看有什么不同，如图7-7所示。

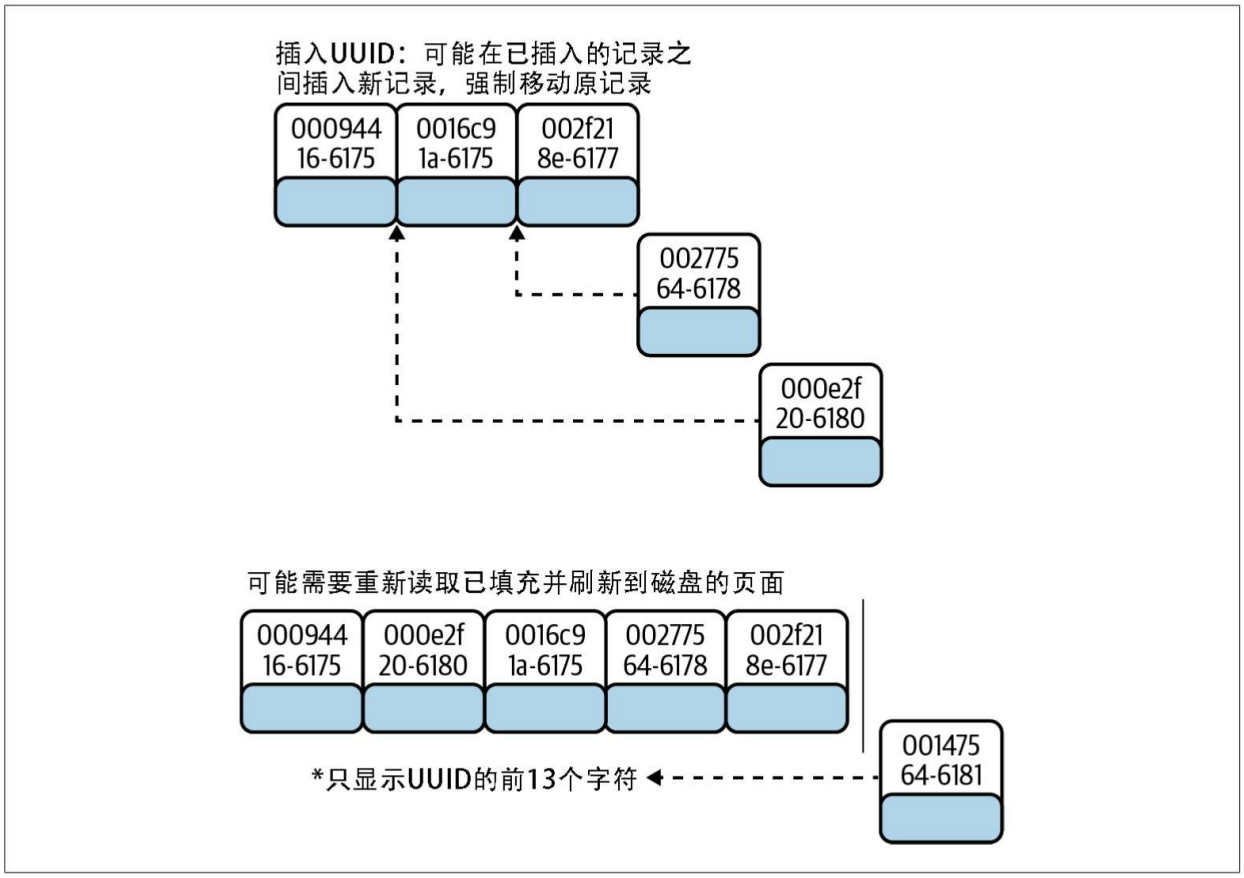

图7-7：向聚簇索引中插入无序的记录

因为新写入的记录的主键值不一定比之前插入的大，所以InnoDB无法简单地总是把新记录插到索引的最后，而是需要为新记录寻找合适的位置——通常是已有数据的中间位置——并且分配空间。这会增加很多额外工作，并导致数据分布不够优化。下面是总结的一些缺点：

- 写入的目标页可能已经刷到磁盘上并从缓存中移除，或者还没有被加载到缓存中，InnoDB在插入之前不得不先找到，并从磁盘将目标页读取到内存中。这将导致大量的随机I/O。
- 因为写入是乱序的，所以InnoDB不得不频繁地做页分裂操作，以便为新记录分配空间。页分裂会导致移动大量数据，一次插入最少需要修改三个页而不是一个。
- 由于频繁的页分裂，页会变得稀疏并被不规则地填充，所以最终数据会有碎片。

在把这些随机值载入聚簇索引后，最好做一次OPTIMIZE TABLE来重建表并优化页的填充情况。

从这个案例可以看出，使用InnoDB时应该尽可能地按主键顺序插入数据，并且尽可能地按照单调增加的聚簇键的值顺序插入新记录。

什么时候按主键顺序插入反而会更糟

对于高并发的工作负载，在InnoDB中按主键顺序插入可能会造成明显的写入竞争。主键的上界会成为“热点”。因为所有的插入都发生在这里，所以并发插入可能导致间

隙锁竞争。另一个热点可能是AUTO_INCREMENT锁机制；如果遇到这个问题，则可能需要考虑重新设计表或者应用，或者更改innodb_autoinc_lock_mode配置。如果你的服务器版本还不支持innodb_autoinc_lock_mode参数，可以将其升级到新版本的InnoDB，该版本对这种场景会适应得更好。

覆盖索引

大家通常都会根据查询的WHERE条件来创建合适的索引，不过这只是索引优化的一个方面。设计优秀的索引应该考虑到整个查询，而不单是WHERE条件部分。索引的确是一种高效找到数据的方式，但是如果MySQL还可以使用索引直接获取列的数据，这样就不再需要读取数据行了。如果索引的叶子节点中已经包含要查询的数据，那么还有什么必要再回表查询呢？如果一个索引包含（或者说覆盖）所有需要查询的字段的值，我们就称之为覆盖索引。需要注意的是，只有B-tree索引可以用于覆盖索引。

覆盖索引是非常有用的工具，能够极大地提高性能。试想一下，如果查询只需要扫描索引而无须回表，会带来多少好处：

- 索引条目通常远小于数据行大小，所以如果只需要读取索引，那么MySQL就会极大地减少数据访问量。这对缓存型的应用负载非常重要，因为在这种情况下，响应时间大部分花费在数据拷贝上。覆盖索引对于I/O密集型的应用也有帮助，因为索引比数据更小，更容易全部放入内存中。
- 因为索引是按照列值的顺序存储的（至少在单页内如此），所以对于I/O密集型的范围查询会比随机从磁盘读取每一行数据的I/O要少得多。可以通过OPTIMIZE命令使得索引完全实现顺序排列，这让简单的范围查询能使用完全顺序的索引访问。
- 由于InnoDB的聚簇索引的特点，覆盖索引对InnoDB表特别有用。InnoDB的二级索引在叶子节点中保存了记录的主键值，所以如果二级索引能够覆盖查询，则可以避免对主键索引的二次查询。

在所有这些场景中，在索引中满足查询的成本一般比查询记录本身要小得多。

当执行一个被索引覆盖的查询（也叫作索引覆盖查询）时，在EXPLAIN的Extra列可以看到“Using index”的信息。[9]例如，表sakila.inventory有一个多列索引（store_id，film_id）。MySQL如果只需访问这两列，就可以使用这个索引做覆盖索引，如下所示：

```
mysql> EXPLAIN SELECT store_id, film_id FROM sakila.inventory\G
*************************** 1. row ***************************
 id: 1
 select_type: SIMPLE
 table: inventory
 partitions: NULL
 type: index
 possible_keys: NULL
 key: idx_store_id_film_id
 key_len: 3
 ref: NULL
 rows: 4581
 filtered: 100.00
 Extra: Using index
```

在大多数存储引擎中，索引只能覆盖那些只访问索引中部分列的查询，不过，可以更进一步优化InnoDB。回想一下，InnoDB的二级索引的叶子节点都包含了主键的值，这意味着InnoDB的二级索引可以有效地利用这些“额外”的主键列来覆盖查询。

例如，sakila.actor表使用InnoDB存储引擎，并在last_name字段有二级索引，虽然该索引的列不包括主键列actor_id，但下面的查询也能够使用索引来对actor_id列做覆盖查询：

```
mysql> EXPLAIN SELECT actor_id, last_name
    -> FROM sakila.actor WHERE last_name = 'HOPPER'\G
*************************** 1. row ***************************
 id: 1
 select_type: SIMPLE
 table: actor
 partitions: NULL
 type: ref
 possible_keys: idx_actor_last_name
 key: idx_actor_last_name
 key_len: 182
 ref: const
 rows: 2
 filtered: 100.00
 Extra: Using index
```

使用索引扫描来做排序

MySQL有两种方式可以生成有序的结果：通过排序操作，或者按索引顺序扫描。如果在EXPLAIN的输出结果中，type列的值为“index”，则说明MySQL使用了索引扫描来做排序（注意，不要和Extra列的“Using index”搞混）。

扫描索引本身是很快的，因为只需要从一条索引记录移动到紧接着的下一条记录。但如果索引不能覆盖查询所需的全部列，那么就不得不每扫描一条索引记录都回表查询一次对应

的记录。这基本上都是随机I/O，因此按索引顺序读取数据的速度通常要比顺序地全表扫描慢，尤其是在I/O密集型的应用负载上。

MySQL可以使用同一个索引既满足排序，又用于查找行。因此，如果可能，设计索引时应该尽可能地同时满足这两项任务，这样是最好的。

只有当索引的顺序和ORDER BY子句的顺序完全一致，并且所有列的排序方向（倒序或正序）都一样时，MySQL才能使用索引来对结果做排序。[10]如果查询需要联接多张表，则只有当ORDER BY子句引用的字段全部在第一个表中时，才能使用索引做排序。ORDER BY子句和查找型查询的限制是一样的：需要满足索引的最左前缀的要求，否则，MySQL需要执行排序操作，而无法利用索引排序。

有一种特殊情况，如果前导列为常量的时候，ORDER BY子句中的列也可以不满足索引的最左前缀的要求。如果在WHERE子句或者JOIN子句中将这些列指定为了常量，就可以“填补”索引字段的间隙了。

例如，Sakila示例数据库的表rental在列（rental_date，inventory_id，customer_id）上建有名称为rental_date的索引：

```
CREATE TABLE rental (
 ...
 PRIMARY KEY (rental_id),
 UNIQUE KEY rental_date (rental_date,inventory_id,customer_id),
 KEY idx_fk_inventory_id (inventory_id),
 KEY idx_fk_customer_id (customer_id),
 KEY idx_fk_staff_id (staff_id),

 ...
);
```

MySQL可以使用rental_date索引为下面的查询做排序，从EXPLAIN中可以看到没有出现文件排序（filesort）操作：[11]

```
mysql> EXPLAIN SELECT rental_id, staff_id FROM sakila.rental
    -> WHERE rental_date = '2005-05-25'
    -> ORDER BY inventory_id, customer_id\G
*************************** 1. row ***************************
 type: ref
 possible_keys: rental_date
 key: rental_date
 rows: 1
 Extra: Using where
```

即使ORDER BY子句不满足索引的最左前缀的要求，也可以用于查询排序，这正是因为索引的第一列被指定为了一个常数。

还有很多可以使用索引做排序的查询示例。下面这个查询可以利用索引排序，是因为查询为索引的第一列提供了常量条件，而使用第二列进行排序，将两列组合在一起，就形成了索引的最左前缀。

```
... WHERE rental_date = '2005-05-25' ORDER BY inventory_id DESC;
```

下面这个查询也没有问题[12]，因为ORDER BY使用的两列就是索引的最左前缀：

```
... WHERE rental_date > '2005-05-25' ORDER BY rental_date, inventory_id;
```

下面是一些不能使用索引做排序的查询。
下面这个查询使用了两种不同的排序方向，但是索引中的列都是按正序排序的：

```
... WHERE rental_date = '2005-05-25' ORDER BY inventory_id DESC, customer_id ASC;
```

在下面这个查询的ORDER BY子句中，引用了一个不在索引中的列：

```
... WHERE rental_date = '2005-05-25' ORDER BY inventory_id, staff_id;
```

下面这个查询的WHERE和ORDER BY中的列无法组合成索引的最左前缀：

```
... WHERE rental_date = '2005-05-25' ORDER BY customer_id;
```

下面这个查询在索引列的第一列上是范围条件，所以MySQL无法使用索引的其余列：

```
... WHERE rental_date > '2005-05-25' ORDER BY inventory_id, customer_id;
```

这个查询在inventory_id列上有多个等于条件。对于排序来说，这也是一种范围查询：

```
... WHERE rental_date = '2005-05-25' AND inventory_id IN(1,2) ORDER BY
customer_id;
```

在下面这个例子中，理论上是可以使用索引进行联接排序的，但由于优化器在优化时将film_actor表当作联接的第二张表，所以实际上无法使用索引：

```
mysql> EXPLAIN SELECT actor_id, title FROM sakila.film_actor
    -> INNER JOIN sakila.film USING(film_id) ORDER BY actor_id\G
+------------+----------------------------------------------+
| table      | Extra                                        |
+------------+----------------------------------------------+
| film       | Using index; Using temporary; Using filesort |
| film_actor | Using index                                  |
+------------+----------------------------------------------+
```

使用索引做排序的另一个最重要的场景是，查询语句中同时有ORDER BY和LIMIT子句的情况。

冗余和重复索引

不幸的是，MySQL允许在相同列上创建多个相同的索引。虽然MySQL会抛出一个警告，但是并不会阻止你这么做。MySQL需要单独维护重复的索引，优化器在优化查询的时候也需要逐个地进行评估，这会影响性能，同时也浪费磁盘空间。
重复索引是指在相同的列上按照相同顺序创建的相同类型的索引。应该避免创建这样的重复索引，发现以后应该立即移除。
有时，还是会在不经意间创建重复索引，例如下面的代码：

```
CREATE TABLE test (
 ID INT NOT NULL PRIMARY KEY,
 A INT NOT NULL,
 B INT NOT NULL,
 UNIQUE(ID),
 INDEX(ID)
) ENGINE=InnoDB;
```

一个经验不足的用户可能是想创建一个主键，先加上唯一限制，然后再加上索引以供查询使用。事实上，MySQL的唯一限制和主键限制都是通过索引实现的，因此，上面的写法实际上在相同的列上创建了三个重复的索引。通常，这样做是完全没有必要的，除非是在同一列上创建不同类型的索引来满足不同类型查询的需求。[13]

冗余索引和重复索引有一些不同。如果创建了索引（A，B），再创建索引（A）就是冗余索引，因为这只是前一个索引的前缀索引，因此，索引（A，B）也可以当作索引（A）来使用（这种冗余只是对B-tree索引来说的）。但是如果再创建索引（B，A），则不是冗余索引，索引（B）也不是，因为B不是索引（A，B）的最左前缀列。另外，如果新建的是其他不同类型的索引（例如，哈希索引或者全文索引），那么无论覆盖了哪些索引列，也不会是B-tree索引的冗余索引。

冗余索引通常发生在为表添加新索引的时候。例如，有人可能会增加一个新的索引（A，B）而不是扩展已有的索引（A）。还有一种情况是，将一个索引扩展为（A，ID），其中ID是主键，因为主键列已经包含在二级索引中了，所以这也是冗余的。

大多数情况下都不需要冗余索引，应该尽量扩展已有的索引而不是创建新的索引。但有时候出于性能方面的考虑也需要冗余索引，因为扩展已有的索引会导致其变得太大，从而影响其他使用该索引的查询的性能。

例如，如果在整数列上有一个索引，现在需要额外增加一个很长的VARCHAR列来扩展该索引，那么性能可能会急剧下降。特别是当有查询把这个索引当作覆盖索引使用的时候。

考虑如下的userinfo表：

```
CREATE TABLE userinfo (
 id int unsigned NOT NULL AUTO_INCREMENT,
 name varchar(64) NOT NULL DEFAULT '',
 email varchar(64) NOT NULL DEFAULT '',
 password varchar(64) NOT NULL DEFAULT '',
 dob date DEFAULT NULL,
 address varchar(255) NOT NULL DEFAULT '',
 city varchar(64) NOT NULL DEFAULT '',
 state_id tinyint unsigned NOT NULL DEFAULT '0',
 zip varchar(8) NOT NULL DEFAULT '',
 country_id smallint unsigned NOT NULL DEFAULT '0',
 account_type varchar(32) NOT NULL DEFAULT '',
 verified tinyint NOT NULL DEFAULT '0',
 allow_mail tinyint unsigned NOT NULL DEFAULT '0',
 parrent_account int unsigned NOT NULL DEFAULT '0',
```

```
 closest_airport varchar(3) NOT NULL DEFAULT '',
 PRIMARY KEY (id),
 UNIQUE KEY email (email),
 KEY country_id (country_id),
 KEY state_id (state_id)
) ENGINE=InnoDB
```

这个表有1，000，000行，每个state_id值大概有20，000条记录。在state_id列有一个索引，可以用于优化如下的查询，假设该查询名为Q1：

```
SELECT count(*) FROM userinfo WHERE state_id=5;
```

一个简单的基准测试表明，该查询的执行速度大概是每秒115次（QPS）。还有一个相关查询需要检索几列的值，而不是只统计行数，假设名为Q2：

```
SELECT state_id, city, address FROM userinfo WHERE state_id=5;
```

对于这个查询，测试结果QPS小于10。[14]提升该查询性能的最简单办法就是扩展索引为（state_id，city，address），让索引能覆盖查询：

```
ALTER TABLE userinfo DROP KEY state_id,
ADD KEY state_id_2 (state_id, city, address);
```

索引扩展后，Q2运行得更快了，但是Q1却变慢了。如果我们想让两个查询都变得更快，就需要两个索引，但这样一来原来的单列索引就是冗余的了。表7-2显示了这两个查询在不同的索引策略下的详细结果。

表7-2：不同索引策略下SELECT查询的QPS测试结果

	只有 state_id	只有 state_id_2	同时有 state_id 和 state_id_2
Query 1	108.55	100.33	107.97
Query 2	12.12	28.04	28.06

建两个索引的缺点是，会带来一定的维护成本。表7-3展示了向表中插入100万行数据所需要的时间。

表7-3：使用不同索引策略插入100万行数据的速度

	只有 state_id	同时有 state_id 和 state_id_2
InnoDB，两个索引都有足够的内存	80 秒	136 秒

可以看到，表中的索引越多，插入的速度越慢。一般来说，增加新索引会导致INSERT、UPDATE、DELETE等操作的速度变慢，特别是当新增索引后达到了内存瓶颈的时候。

解决冗余索引和重复索引的方法很简单，删除这些索引就可以了，但首先要做的是找出这样的索引。可以针对INFORMATION_SCHEMA表编写各种复杂的查询来识别这类索引，也有更简单的技术，比如可以使用Percona工具箱中的*pt-duplicate-key-checker*，该工具通过分析表结构来找出冗余和重复索引。

在删除或扩展索引的时候要非常小心。回忆一下，在前面的InnoDB的示例表中，因为二级索引的叶子节点包含了主键值，所以在列（A）上的索引就相当于在（A，ID）上的索引。如果有像WHERE A=5 ORDER BY ID这样的查询，这个索引会很有用。但如果将索

引扩展为（A，B），则实际上就变成了（A，B，ID），那么上面查询的ORDER BY子句就无法使用该索引做排序，而只能用文件排序了。所以，建议使用Percona工具箱中的*pt-upgrade*工具来仔细检查计划中的索引变更。

对于上述的两种情况，都可以考虑使用MySQL 8.0的不可见索引特性，而不是直接删除索引。要使用这个特性，可以通过ALTER TABLE语句，改变索引的一个标志位，使得优化器在确定执行计划时，忽略该索引。如果你发现计划删除的索引依旧有非常重要的作用，可以直接把索引改成可见，而不需要重新构建该索引。

未使用的索引

除了冗余索引和重复索引，可能还会有一些服务器永远不用的索引。这样的索引完全是累赘，建议删除。[15]

如本书第3章中所述，找到未使用索引的最好办法就是使用系统数据库performance_schema和sys。在sys数据库中，在table_io_waits_summary_by_index_usage视图中可以非常简单地知道哪些索引从来没有被使用过：

```
mysql> SELECT * FROM sys.schema_unused_indexes;
+---------------+---------------+-----------------------------+
| object_schema | object_name   | index_name                  |
+---------------+---------------+-----------------------------+
| sakila        | actor         | idx_actor_last_name         |
| sakila        | address       | idx_fk_city_id              |
| sakila        | address       | idx_location                |
| sakila        | payment       | fk_payment_rental           |
.. trimmed for brevity ..
```

维护索引和表

即使用正确的数据类型创建了表并加上了合适的索引，工作也没有结束：还需要维护表和索引来确保它们都能正常工作。维护表有三个主要目的：找到并修复损坏的表，维护准确的索引统计信息，减少碎片。

找到并修复损坏的表

对于数据表来说，最糟糕的情况就是表被损坏了。另外，其他任何存储引擎都可能由于硬件问题、MySQL本身的缺陷或者操作系统的问题导致索引损坏，当然InnoDB很少出现这样的问题。

损坏的索引会导致查询返回错误的结果或者出现莫须有的主键冲突等问题，严重时甚至还会导致数据库的崩溃。如果你遇到了古怪的问题——例如一些不应该发生的错误——可以尝试运行CHECK TABLE来检查是否发生了表损坏（注意，有些存储引擎不支持该命令；而有些存储引擎则支持以不同的选项来控制检查表的强度）。CHECK TABLE通常能够找出大多数的表和索引的错误。

可以使用REPAIR TABLE命令来修复损坏的表，但同样不是所有的存储引擎都支持该命令。如果存储引擎不支持，可通过一个不做任何操作（n o-o p）的ALTER操作来重建表，例如，将表的存储引擎修改为当前的引擎。下面是一个针对InnoDB表的例子：

```
ALTER TABLE <table> ENGINE=INNODB;
```

此外，还可以将数据导出再导入一次。不过，如果损坏的是系统区域，或者是表的“行数据”区域，而不是索引，那么上面的办法就没有用了。在这种情况下，可以从备份中恢复表，或者尝试从损坏的数据文件中尽可能地恢复数据。

如果是InnoDB存储引擎的表发生了损坏，那么一定是发生了严重的错误，需要立刻调查一下原因。InnoDB一般不会出现损坏，它的设计保证了它并不容易被损坏。如果发生了，一般要么是数据库的硬件问题，例如，内存或者磁盘问题（有可能），要么是由于数据库管理员的误操作，例如，在MySQL外部操作了数据文件（有可能），抑或是InnoDB本身的缺陷（不太可能）。常见的类似错误通常是由于尝试使用*rsync*备份InnoDB导致的。不存在什么查询能够让InnoDB表损坏，也不用担心暗处有“陷阱”。如果某条查询导致InnoDB数据的损坏，那么一定是遇到了bug，而不是查询的问题。

如果遇到数据损坏，最重要的是找出是什么导致了损坏，而不只是简单地修复，否则很有可能还会不断地出现数据损坏的情况。可以通过设置innodb_force_recovery参数进入InnoDB的强制恢复模式来修复数据，更多细节可以参考MySQL手册。

更新索引统计信息

如果存储引擎向优化器提供的扫描行数信息不准确，或者执行计划本身太复杂以致无法准确地获取各个阶段匹配的行数，那么优化器会使用索引统计信息来估算扫描行数。MySQL的优化器使用的是基于成本的模型，而衡量成本的主要指标就是一个查询需要扫

描多少行。如果表没有统计信息，或者统计信息不准确，优化器就很有可能做出错误的决定。可以通过运行ANALYZE TABLE来重新生成统计信息，以解决这个问题。

可以使用SHOW INDEX FROM命令来查看索引的基数（cardinality）。例如：

```
mysql> SHOW INDEX FROM sakila.actor\G
*************************** 1. row ***************************
 Table: actor
 Non_unique: 0
 Key_name: PRIMARY
 Seq_in_index: 1
 Column_name: actor_id
 Collation: A
 Cardinality: 200
 Sub_part: NULL
 Packed: NULL
 Null:
 Index_type: BTREE
 Comment:
*************************** 2. row ***************************
 Table: actor
 Non_unique: 1
 Key_name: idx_actor_last_name
 Seq_in_index: 1
 Column_name: last_name
 Collation: A
 Cardinality: 200
 Sub_part: NULL
 Packed: NULL
 Null:
 Index_type: BTREE
 Comment:
```

这个命令输出了很多关于索引的信息，在MySQL手册中对上面每个字段的含义都有详细的解释。这里需要特别提及的是索引列的基数，其显示了存储引擎估算索引列有多少个不同的取值。还可以通过INFORMATION_SCHEMA.STATISTICS表很方便地查询到这些信息。例如，基于INFORMATION_SCHEMA的表，可以编写一个查询给出当前选择性比较低的索引。需要注意的是，如果服务器上的库表非常多，则从这里获取元数据的速度可能会非常慢，而且会给MySQL带来额外的压力。

InnoDB的统计信息值得深入研究。InnoDB引擎通过抽样的方式来计算统计信息，首先随机地读取少量的索引页面，然后以此为样本计算索引的统计信息。在旧InnoDB版本中，样本页面数是8，新版本的InnoDB可以通过参数innodb_stats_sample_pages来设置样本页的数量。设置更大的值，理论上来说可以帮助生成更准确的索引信息，特别是对于某些超大的数据表来说，但具体设置多大合适依赖于具体的环境。

InnoDB会在表首次打开，或者执行ANALYZE TABLE，或者表的大小发生非常大的变化时计算索引的统计信息。

InnoDB在打开某些INFORMATION_SCHEMA表，或者使用SHOW TABLE STATUS和

SHOW INDEX，或者在MySQL客户端开启自动补全功能的时候，会触发索引统计信息的更新。如果服务器上有大量的数据表，这可能会带来严重的问题，尤其是当I/O比较慢的时候。客户端程序或者监控工具触发索引信息采样更新时可能会导致大量的锁，并给服务器带来很多额外的压力，这会让用户因为启动时间漫长而感到沮丧。只要使用SHOW INDEX查看索引统计信息，就一定会触发统计信息的更新。可以关闭innodb_stats_on_metadata参数来避免上面提到的问题。

减少索引和数据的碎片

B-tree索引可能会产生碎片化，这会降低查询的效率。碎片化的索引可能会以很差或者无序的方式存储在磁盘上。

根据设计，B-tree索引需要随机磁盘访问才能定位到叶子页，所以随机访问总是不可避免的。然而，如果叶子页在物理分布上是顺序且紧密的，那么查询的性能就会更好。否则，对于范围查询、索引覆盖扫描等操作来说，速度可能会降低很多；对于索引覆盖扫描，这一点会表现得更加明显。

表的数据存储也可能发生碎片化。然而，数据存储的碎片化比索引更加复杂。有三种类型的数据碎片。

行碎片（*Row fragmentation*）

> 这种碎片指的是数据行被存储在多个地方的多个片段中。即使查询只从索引中访问一行记录，行碎片也会导致性能下降。

行间碎片（*Intra-row fragmentation*）

> 行间碎片是指逻辑上顺序的页或者行，在磁盘上不是顺序存储的。行间碎片对诸如全表扫描和聚簇索引扫描之类的操作有很大的影响，因为这些操作原本能够从磁盘上顺序存储的数据中获益。

剩余空间碎片（*Free space fragmentation*）

> 剩余空间碎片是指数据页中有大量的空余空间。这会导致服务器读取大量不需要的数据，从而造成浪费。

可以通过执行OPTIMIZE TABLE或者导出再导入的方式来重新整理数据。这对多数存储引擎都是有效的。对于那些不支持OPTIMIZE TABLE的存储引擎，可以通过一个不做任何操作（n o-o p）的ALTER TABLE操作来重建表。只需将表的存储引擎修改为当前的引擎即可：

```
ALTER TABLE <table> ENGINE=<engine>;
```

小结

通过本章可以看到，索引是一个非常复杂的话题！MySQL和存储引擎访问数据的方式，加上索引的特性，使得索引成为一个影响数据访问的有力而灵活的工具（无论数据是存放在磁盘中还是在内存中）。

在MySQL中，大多数情况下都会使用B-tree索引。其他类型的索引大多只适用于特殊的目的。如果在合适的场景中使用索引，将大大缩短查询的响应时间。本章将不再介绍更多这方面的内容了，最后值得总体回顾一下这些特性以及如何使用B-tree索引。

在选择索引和编写利用这些索引的查询时，有如下三个原则始终需要记住：

- 单行访问是很慢的，特别是在机械硬盘中存储（SSD的随机I/O要快很多，不过这一点仍然成立）。如果服务器从存储中读取一个数据块只是为了获取其中一行，那么就浪费了很多工作。最好读取的块中能包含尽可能多的所需要的行。
- 按顺序访问范围数据是很快的，有两个原因。第一，顺序I/O不需要多次磁盘寻道，所以比随机I/O要快很多（特别是对于机械硬盘）。第二，如果服务器能够按需顺序读取数据，那么就不再需要额外的排序操作，并且GROUP BY查询也无须再做排序和将行按组进行聚合计算了。
- 索引覆盖查询是很快的。如果一个索引包含了查询需要的所有列，那么存储引擎就不需要再回表查找行。这避免了大量的单行访问，而上面的第一点已经写明单行访问是很慢的。

总的来说，编写查询语句时应该尽可能选择合适的索引以避免单行查找，尽可能地使用数据内部顺序从而避免额外的排序操作，并尽可能地使用索引覆盖查询。这与本章开头提到的Lahdenmaki和Leach的书中介绍的“三星”评价系统是一致的。

如果表中的每一个查询都能有一个完美的索引来满足当然是最好的。但不幸的是，要这么做有时可能需要创建大量的索引。还有一些时候对某些查询是不可能创建一个达到“三星”级别的索引的（例如，查询要按照两列排序，其中一列为正序，另一列为倒序）。这时必须有所取舍以创建最合适的索引，或者寻求替代策略（例如，反范式化或者提前计算汇总表等）。

理解索引是如何工作的非常重要，应该根据这些理解来创建最合适的索引，而不是根据一些诸如“在多列索引中将选择性最高的列放在第一列”或“应该为WHERE子句中出现的所有列创建索引”之类的经验法则及其推论来创建。

那么如何判断一个系统创建的索引是合理的呢？一般来说，我们建议按响应时间来对查询进行分析。找出那些消耗最长时间的查询或者那些给服务器带来最大压力的查询，然后检查这些查询的schema、SQL语句和索引结构，判断是否有查询扫描了太多的行，是否做了很多额外的排序或者使用了临时表，是否使用随机I/O访问数据，或者是否有太多回表查询查询那些不在索引中的列的操作。

如果一个查询无法从所有可能的索引中获益，则应该看看是否可以创建一个更合适的索引来提升性能。如果不行，还可以看看是否可以重写该查询，将其转化成一个能够高效利用现有索引或者新创建索引的查询。这也是下一章要介绍的内容。

[1] 在本书的第**4**章介绍过，各种固态硬盘在性能上有一些不一样的特点。索引的原则依然成立，相比传统硬盘，在**SSD**设备上，糟糕的索引带来的负面影响要稍微小一些。

[2] 实际上，很多存储引擎使用的是**B+tree**索引，即每一个叶子节点都包含指向下一个叶子节点的指针，从而方便遍历叶子节点的范围。对于**B-tree**索引更详细的细节信息可以参考计算机科学方面的相关书籍。

[3] 这是**MySQL**的特性，具体的版本之间也有一些差异。虽然使用完全前缀的效率会更高，但其他一些数据库可以使用索引的非前缀部分。**MySQL**未来也可能会提供这个特性，本章后面会介绍一些绕过限制的方法。

[4] **MySQL**优化器是一个非常神秘且强大的**“**装置**”**，不过还好，它的**“**强大**”**应该略胜于**“**神秘**”**一筹。鉴于它查找最优执行计划的方式，你应该更多依靠**EXPLAIN**和具体业务负载情况，来决定最优的执行策略。

[5] **Oracle**用户可能更熟悉索引组织表（**index-organized table**）的说法，它们实际上是一样的意思。

[6] 这并非总成立，很快就可以看到。

[7] 顺便提一下，并不是所有的非聚簇索引都能做到一次索引查询就找到行。当行更新的时候，它可能无法被存储在原来的位置，这会导致表中出现行的碎片化或者移动行并在原位置保存**“**向前指针**”**，这两种情况都会导致在查找行时需要做更多的工作。

[8] 值得注意的是，这是一个真实案例中的表，有很多二级索引和列。如果删除这些二级索引只测试主键，那么性能差异将会更明显。

[9] 很容易把**Extra**列的**“Using index”**和**type**列的**“index”**搞混。其实这两者完全不同，**type**列和覆盖索引毫无关系，它只表示这个查询访问数据的方式，或者说是**MySQL**查找记录的方式。**MySQL**手册中称之为联接类型。

[10] 如果需要按不同方向做排序，一个技巧是存储该列值的反转串或者相反数。

[11] **MySQL**在这里称其为文件排序（**filesort**），其实并不一定使用磁盘文件，只有无法在内存中完成排序时才会用到磁盘文件。

[12] 这里需要注意，虽然这里可以使用索引进行排序，但是在实际使用**8.0.25**版本测试的时候，除非强制使用**FORCE INDEX FOR ORDER BY**，否则，优化器不会使用索引进行排序**——**所以，需要特别注意，有时候优化器可能会与预期的行为不相同，总是需要通过**EXPLAIN**语句来确认实际的执行计划。

[13] 如果索引类型不同，则不算是重复索引。例如，实际情况中经常会创建**KEY**（**col**）和**FULLTEXTKEY**（**col**）这两种索引。

[14] 这里使用了全内存的案例，如果当表逐渐变大，导致工作负载变成**I/O**密集型时，性能测试结果差距会更大。对于**COUNT()**查询，覆盖索引性能提升**100**倍也是很有可能的。

[15] 有些索引的功能相当于唯一约束，虽然该索引一直没有被查询使用，却可能是用于避免产生重复数据的。

第8章 查询性能优化

在前面的章节中，我们介绍了如何设计最优的库表结构、如何建立最好的索引，这些对于提高性能来说是必不可少的。但这些还不够——还需要合理地设计查询。如果查询写得很糟糕，即使库表结构再合理、索引再合适，也无法实现高性能。

查询优化、索引优化、库表结构优化需要齐头并进，一个不落。在获得编写MySQL查询的经验的同时，你还将学习到如何为高效的查询设计表和索引。同样地，你也可以学习到在优化库表结构时会影响到哪些类型的查询。这个过程需要时间，建议大家在学习后面章节的时候多回头看看这些章节的内容。

本章将从查询设计的一些基本原则开始——这也是在发现查询效率不高的时候首先需要考虑的因素。然后会介绍一些更深的查询优化的技巧，并会介绍一些MySQL优化器内部的机制。我们将展示MySQL是如何执行查询的，你也将学会如何去改变一个查询的执行计划。最后，我们要看一下MySQL优化器在哪些方面做得还不够，并探索查询优化的模式，以帮助MySQL更有效地执行查询。

本章的目标是帮助大家更深刻地理解MySQL如何真正地执行查询，并明白高效和低效的原因何在，这样才能充分发挥MySQL的优势，并避开它的弱点。

为什么查询速度会慢

在尝试编写快速的查询之前，需要清楚一点，真正重要的是响应时间。如果把查询看作一个任务，那么它由一系列子任务组成，每个子任务都会消耗一定的时间。如果要优化查询，实际上要优化其子任务，要么消除其中一些子任务，要么减少子任务的执行次数，要么让子任务运行得更快。

通常来说，查询的生命周期大致可以按照如下顺序来看：从客户端到服务器，然后在服务器上进行语法解析，生成执行计划，执行，并给客户端返回结果。其中，“执行”可以被认为是整个生命周期中最重要的阶段，这其中包括大量为了检索数据对存储引擎的调用以及调用后的数据处理，包括排序、分组等。

在完成这些任务的时候，查询需要在不同的地方花费时间，包括网络、CPU计算、生成统计信息和执行计划、锁等待（互斥等待）等操作，尤其是向底层存储引擎检索数据的调用操作，这些调用需要在内存操作、CPU操作和内存不足时导致的I/O操作上消耗时间。根据存储引擎不同，可能还会产生大量的上下文切换以及系统调用。

在每一个消耗大量时间的查询案例中，我们都能看到一些不必要的操作、某些操作被额外地重复了很多次、某些操作执行得太慢等。优化查询的目的就是减少和消除这些操作所花费的时间。

再次声明一点，对于一个查询的全部生命周期，上面列得并不完整。这里我们只是想说明：了解查询的生命周期和清楚查询的时间消耗情况对于优化查询有很大意义。有了这些概念，我们再一起来看看如何优化查询。

慢查询基础：优化数据访问

一条查询，如果性能很差，最常见的原因是访问的数据太多。某些查询可能不可避免地需要筛选大量数据，但这并不常见。大部分性能低下的查询都可以通过减少访问的数据量的方式进行优化。对于低效的查询，我们发现通过下面两个步骤来分析总是很有效：

1. 确认应用程序是否在检索大量且不必要的数据。这通常意味着访问了太多的行，但有时候也可能是访问了太多的列。
2. 确认MySQL服务器层是否在分析大量不需要的数据行。

是否向数据库请求了不需要的数据

有些查询会请求超过实际需要的数据，然后这些多余的数据会被应用程序丢弃。这会给MySQL服务器带来额外的负担，并增加网络开销[1]，另外，这也会消耗应用服务器的CPU和内存资源。

以下是一些典型案例。

查询了不需要的记录

一个常见的错误是，常常会误以为MySQL只会返回需要的数据，实际上MySQL却是先返回全部结果集再进行计算。我们经常会看到一些了解其他数据库系统的人会设计出这类应用程序。这些开发者习惯使用这样的技术，先使用SELECT语句查询大量的结果，然后获取前面的*N*行后关闭结果集（例如，在新闻网站中取出100条记录，但是只是在页面上显示前面10条）。他们认为MySQL会执行查询，并只返回他们需要的10条数据，然后停止查询。实际情况是，MySQL会查询出全部的结果集，客户端的应用程序会接收全部的结果集数据，然后抛弃其中大部分数据。最简单有效的解决方法就是在这样的查询后面加上LIMIT子句。

多表联接时返回全部列

如果你想查询所有在电影*Academy Dinosaur*中出现的演员，千万不要按下面的写法编写查询：

```
SELECT * FROM sakila.actor
INNER JOIN sakila.film_actor USING(actor_id)
INNER JOIN sakila.film USING(film_id)
WHERE sakila.film.title = 'Academy Dinosaur';
```

这将返回这三个表的全部数据列。正确的方式应该是像下面这样只取需要的列：

```
SELECT sakila.actor.* FROM sakila.actor...;
```

总是取出全部列

每次看到SELECT*的时候都需要用怀疑的眼光审视，是不是真的需要返回全部的列，很可能不是必需的。取出全部列，会让优化器无法完成索引覆盖扫描这类优化，还会为服务器带来额外的I/O、内存和CPU的消耗。因此，一些DBA严格禁止

SELECT*的写法，这样做有时候还能避免某些列被修改而带来的问题。

当然，查询返回超过需要的数据也不总是坏事。在我们研究过的许多案例中，人们会告诉我们，这种有点浪费数据库资源的方式可以简化开发，因为能提高相同代码片段的复用性，如果清楚这样做对性能的影响，那么这种做法也是值得考虑的。如果应用程序使用了某种缓存机制，或者有其他考虑，获取超过需要的数据也可能有其好处，但不要忘记这样做的代价是什么。获取并缓存所有的列的查询，相比多个独立的只获取部分列的查询可能更有好处。

重复查询相同的数据

如果你不够小心，很容易出现这样的错误——不断地重复执行相同的查询，然后每次都返回完全相同的数据。例如，在用户评论的地方需要查询用户头像的URL，那么在用户多次评论的时候，可能就会反复查询这个数据。比较好的方案是，当初次查询的时候将这个数据缓存起来，需要的时候从缓存中取出，这样性能显然会更好。

MySQL是否在扫描额外的记录

在确定查询只返回需要的数据以后，接下来应该看看查询为了返回结果是否扫描了过多的数据。对于MySQL，最简单的衡量查询开销的三个指标如下：

- 响应时间
- 扫描的行数
- 返回的行数

没有哪个指标能够完美地衡量查询的开销，但它们大致反映了MySQL在内部执行查询时需要访问多少数据，并可以大概推算出查询运行的时间。这三个指标都会被记录到MySQL的慢日志中，所以检查慢日志记录是找出扫描行数过多的查询的好办法。

响应时间

要记住，响应时间只是一个表面上的值。这样说可能看起来和前面关于响应时间的说法有矛盾，其实并不矛盾，响应时间仍然是最重要的指标，这有一点复杂，后面细细道来。

响应时间是两部分之和：服务时间和排队时间。服务时间是指数据库处理这个查询真正花了多长时间。排队时间是指服务器因为等待某些资源而没有真正执行查询的时间——可能是等I/O操作完成，也可能是等待行锁，等等。遗憾的是，我们无法把响应时间细分到上面这些部分，除非有什么办法能够逐个测量这些消耗，这很难做到。最常见和重要的是I/O等待和锁等待，但是实际情况更加复杂。实际上，I/O等待和锁等待非常重要，因为它们对于性能有着至关重要的影响。

所以在不同类型的应用压力下，响应时间并没有一致的规律或者公式。诸如存储引擎的锁（表锁、行锁）、高并发资源竞争、硬件响应等诸多因素都会影响响应时间。所以，响应时间既可能是一个问题的结果也可能是一个问题的原因，不同案例情况不同。

当你看到一个查询的响应时间时，首先需要问问自己，这个响应时间是否是一个合理的值。实际上可以使用“快速上限估计”法来估算查询的响应时间，这是在Tapio Lahdenmaki和Mike Leach编写的*Relational Database Index Design and the Optimizers*（Wiley出版社出版）一书中提到的技术，限于篇幅，在这里不会详细展开。概括地说，了解这个查询需要

哪些索引以及它的执行计划是什么，然后计算大概需要多少个顺序和随机I/O，再用其乘以在具体硬件条件下一次I/O的消耗时间。最后把这些消耗都加起来，就可以获得一个大概参考值来判断当前响应时间是不是一个合理的值。

扫描的行数和返回的行数

分析查询时，查看该查询扫描的行数是非常有帮助的。这在一定程度上能够说明该查询找到需要的数据的效率高不高。对于找出那些“糟糕”的查询，这个指标可能还不够完美，因为并不是所有行的访问代价都是相同的。较短的行的访问速度更快，内存中的行比磁盘中的行的访问速度要快得多。

理想情况下扫描的行数和返回的行数应该是相同的，但实际中这种“美事”并不多。例如，在做一个联接查询时，服务器必须要扫描多行才能生成结果集中的一行。扫描的行数与返回的行数的比率通常很低，一般在1：1到10：1之间，不过有时候这个值也可能非常非常大。

扫描的行数和访问类型

在评估查询开销的时候，需要考虑从表中找到某一行数据的成本。MySQL有好几种访问方式可以查找并返回一行结果。有些访问方式可能需要扫描很多行才能返回一行结果，也有些访问方式可能无须扫描就能返回结果。

EXPLAIN语句中的type列反映了访问类型。访问类型有很多种，从全表扫描到索引扫描、范围扫描、唯一索引查询、常数引用等。这里列出的这些，速度从慢到快，扫描的行数从多到少。你不需要记住这些访问类型，但需要明白扫描表、扫描索引、范围访问和单值访问的概念。

如果你没办法找到合适的访问类型，那么最好的解决办法通常就是增加一个合适的索引，这也正是我们前一章讨论过的问题。现在应该明白为什么索引对于查询优化如此重要了吧。索引让MySQL以最高效、扫描行数最少的方式找到需要的记录。

例如，我们看一下示例数据库Sakila中的一个查询案例：

```
SELECT * FROM sakila.film_actor WHERE film_id = 1;
```

这个查询将返回10行数据，从EXPLAIN的结果可以看到，MySQL在索引idx_fk_film_id上使用了ref访问类型来执行查询：

```
mysql> EXPLAIN SELECT * FROM sakila.film_actor WHERE film_id = 1\G
*************************** 1. row ***************************
 id: 1
 select_type: SIMPLE
 table: film_actor
 partitions: NULL
 type: ref
 possible_keys: idx_fk_film_id
 key: idx_fk_film_id
```

```
 key_len: 2
 ref: const
 rows: 10
 filtered: 100.00
 Extra: NULL
```

EXPLAIN的结果还显示MySQL预估需要访问10行数据。换句话说，查询优化器认为这种访问类型可以高效地完成查询。如果没有合适的索引会怎样呢？MySQL就不得不使用一种糟糕的访问类型，下面来看看如果删除对应的索引再来运行这个查询会发生什么情况：

```
mysql> ALTER TABLE sakila.film_actor DROP FOREIGN KEY fk_film_actor_film;
mysql> ALTER TABLE sakila.film_actor DROP KEY idx_fk_film_id;
mysql> EXPLAIN SELECT * FROM sakila.film_actor WHERE film_id = 1\G
*************************** 1. row ***************************
 id: 1
 select_type: SIMPLE
 table: film_actor
 partitions: NULL
 type: ALL
 possible_keys: NULL
 key: NULL
 key_len: NULL
 ref: NULL
 rows: 5462
 filtered: 10.00
 Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

正如我们预测的，访问类型变成了一个全表扫描（ALL），现在MySQL预估需要扫描5462条记录来完成这个查询。这里的“Using where”表示MySQL将通过WHERE条件来筛选存储引擎返回的记录。

一般地，MySQL能够使用如下三种方式应用WHERE条件，从好到坏依次为：

- 在索引中使用WHERE条件来过滤不匹配的记录。这是在存储引擎层完成的。
- 使用索引覆盖扫描（在Extra列中出现了Using index）来返回记录，直接从索引中过滤不需要的记录并返回命中的结果。这是在MySQL服务器层完成的，但无须再回表查询记录。
- 从数据表中返回数据，然后过滤不满足条件的记录（在Extra列中出现Using where）。这在MySQL服务器层完成，MySQL需要先从数据表中读出记录然后过滤。

上面这个例子说明了好的索引多么重要。好的索引可以让查询使用合适的访问类型，尽可能地只扫描需要的数据行。但也不是说增加索引就能让扫描的行数等于返回的行数。例如，下面是使用聚合函数COUNT()[2]的查询：

```
mysql> SELECT actor_id, COUNT(*)
    -> FROM sakila.film_actor GROUP BY actor_id;
+----------+----------+
| actor_id | COUNT(*) |
+----------+----------+
|        1 |       19 |
|        2 |       25 |
|        3 |       22 |
.. omitted..
|      200 |       20 |
+----------+----------+
200 rows in set (0.01 sec)
```

这条查询语句仅需返回200条记录，但是，它实际读取了多少条记录呢？我们通过前面章节介绍的EXPLAIN语句来查看一下：

```
mysql> EXPLAIN SELECT actor_id, COUNT(*)
    -> FROM sakila.film_actor GROUP BY actor_id\G
*************************** 1. row ***************************
 id: 1
 select_type: SIMPLE
 table: film_actor
 partitions: NULL
 type: index
 possible_keys: PRIMARY
 key: PRIMARY
 key_len: 4
 ref: NULL
 rows: 5462
 filtered: 100.00
 Extra: Using index
```

哇！获取200条记录却需要读取几千行记录，这就意味着，读取了太多不必要的记录。因为WHERE子句中没有过滤掉对应的记录，所以，在这个案例中，索引并不能减少需要扫描的记录行数。

不幸的是，MySQL不会告诉我们生成结果实际上需要扫描多少行数据，而只会告诉我们生成结果时一共扫描了多少行数据。扫描的行中的大部分都很可能是被WHERE条件过滤掉的，对最终的结果集并没有贡献。在上面的例子中，我们删除索引后，看到MySQL需要扫描所有记录然后根据WHERE条件过滤，最终只返回10行结果。理解一个查询需要扫描多少行和实际需要使用的行数需要先去理解这个查询背后的逻辑和思想。

如果发现查询需要扫描大量的数据但只返回少数行，那么通常可以尝试下面的技巧去优化它：

- 使用索引覆盖扫描，把所有需要用的列都放到索引中，这样存储引擎无须回表获取对应行就可以返回结果了（在第7章中我们已经讨论过了）。
- 改变库表结构。例如，使用单独的汇总表（这是我们在第6章中讨论的办法）。
- 重写这个复杂的查询，让MySQL优化器能够以更优化的方式执行这个查询（这是

本章后续需要讨论的问题）。

重构查询的方式

在优化有问题的查询时，目标应该是找到获得实际需要的结果的替代方法——但这并不一定意味着从MySQL返回完全相同的结果集。有时候，可以将查询转换为返回相同结果的等价形式，以获得更好的性能。但是，如果可以获得更好的效率，还应该考虑重写查询以检索不同的结果。通过修改应用代码和查询，最终达到一样的目的。这一节我们将介绍如何通过这种方式来重构查询，并展示何时需要使用这样的技巧。

一个复杂查询还是多个简单查询

设计查询的时候，一个需要考虑的重要问题是，是否需要将一个复杂的查询分成多个简单的查询。在传统实现中，总是强调需要数据库层完成尽可能多的工作，这样做的逻辑在于以前人们总是认为网络通信、查询解析和优化是一件代价很高的事情。

但是这样的想法对于MySQL并不适用，因为MySQL从设计上让连接和断开连接都很轻量，在返回一个小的查询结果方面很高效。现代的网络速度比以前要快很多，能在很大程度上降低延迟。在某些版本的MySQL中，即使在一台通用服务器上，也能够运行每秒超过10万次的简单查询，即使是一个千兆网卡也能轻松满足每秒超过2000次的查询。所以运行多个小查询现在已经不是大问题了。

在MySQL内部，每秒能够扫描内存中上百万行的数据，相比之下，MySQL响应数据给客户端就慢得多了。在其他条件都相同的时候，使用尽可能少的查询当然是更好的。但是有时候，将一个大查询分解为多个小查询是很有必要的。别害怕这样做，好好衡量一下这样做是不是会减少工作量。稍后我们将通过一个示例来展示这个技巧的优势。

不过，在设计应用的时候，如果在一个查询能够胜任时还将其写成多个独立的查询是不明智的。例如，我们看到有些应用对一个数据表做10次独立的查询来返回10行数据，每个查询返回一条结果，查询10次，这时可以使用单个查询获取10行数据。有的应用甚至每次只查询一个字段，获取一行数据就需要执行多次查询。

切分查询

有时候对于一个大查询，我们需要“分而治之”，将大查询切分成小查询，每个查询的功能完全一样，只完成一小部分，每次只返回一小部分查询结果。

删除旧的数据就是一个很好的例子。定期清除大量数据时，如果用一个大的语句一次性完成的话，则可能需要一次锁住很多数据、占满整个事务日志、耗尽系统资源、阻塞很多小的但重要的查询。将一个大的DELETE语句切分成多个较小的查询可以尽可能小地影响MySQL的性能，同时还可以降低MySQL复制的延迟。例如，我们需要每个月运行一次下面的查询：

```
DELETE FROM messages
WHERE created < DATE_SUB(NOW(),INTERVAL 3 MONTH);
```

那么可以用类似下面的办法来完成同样的工作：

```
rows_affected = 0
do {
 rows_affected = do_query(
 "DELETE FROM messages WHERE created < DATE_SUB(NOW(),INTERVAL 3 MONTH)
 LIMIT 10000")
} while rows_affected > 0
```

一次删除一万行数据一般来说是一个比较高效而且对服务器[3]影响最小的做法（如果是事务型引擎，很多时候小事务能够更高效）。同时，需要注意的是，如果每次删除数据后，都暂停一会儿再做下一次删除，也可以将服务器上原本一次性的压力分散到一个很长的时间段中，可以大大降低对服务器的影响，还可以大大减少删除时锁的持有时间。

分解联接查询

很多高性能的应用都会对联接查询进行分解。简单地说，可以对每一个表进行一次单表查询，然后将结果在应用程序中进行联接。例如，下面这个查询：

```
SELECT * FROM tag
JOIN tag_post ON tag_post.tag_id=tag.id
JOIN post ON tag_post.post_id=post.id
WHERE tag.tag='mysql';
```

可以分解成下面这些查询来代替：

```
SELECT * FROM tag WHERE tag='mysql';
SELECT * FROM tag_post WHERE tag_id=1234;
SELECT * FROM post WHERE post.id in (123,456,567,9098,8904);
```

到底为什么要这样做？乍一看，这样做并没有什么好处，原本一条查询，这里却变成多条查询，返回的结果又是一模一样的。事实上，用分解联接查询的方式重构查询有如下优势：

- 让缓存的效率更高。许多应用程序可以方便地缓存单表查询对应的结果对象。例如，上面查询中的tag mysql已经被缓存了，那么应用就可以跳过第一个查询。再例如，应用中已经缓存了ID为123、567、9098的内容，那么第三个查询的IN()中就可以少几个ID。
- 将查询分解后，执行单个查询可以减少锁的竞争。
- 在应用层做联接，可以更容易对数据库进行拆分，更容易做到高性能和可扩展。
- 查询本身的效率也可能会有所提升。在这个例子中，使用IN()代替联接查询，可以让MySQL按照ID顺序进行查询，这可能比随机的联接要更高效。
- 可以减少对冗余记录的访问。在应用层做联接查询，意味着对于某条记录应用只需要查询一次，而在数据库中做联接查询，则可能需要重复地访问一部分数据。从这点看，这样的重构还可能会减少网络和内存的消耗。

在有些场景下，在应用程序中执行联接操作会更加有效。比如，当可以缓存和重用之前查询结果中的数据时、当在多台服务器上分发数据时、当能够使用IN()列表替代联接查询大

型表时、当一次联接查询中多次引用同一张表时。

查询执行的基础

当希望MySQL能够以更高的性能运行查询时，最好的办法就是弄清楚MySQL是如何优化和执行查询的。一旦理解了这一点，很多查询优化工作实际上就是遵循一些原则让优化器能够按照预想的合理的方式运行。

在这里，是时候回头看看我们前面讨论的内容了：MySQL执行一个查询的过程。根据图8-1可以看到，当向MySQL发送一个请求的时候，MySQL到底做了些什么：

1. 客户端给服务器发送一条SQL查询语句。
2. 服务器端进行SQL语句解析、预处理，再由优化器生成对应的执行计划。
3. MySQL根据优化器生成的执行计划，调用存储引擎的API来执行查询。
4. 将结果返回给客户端。

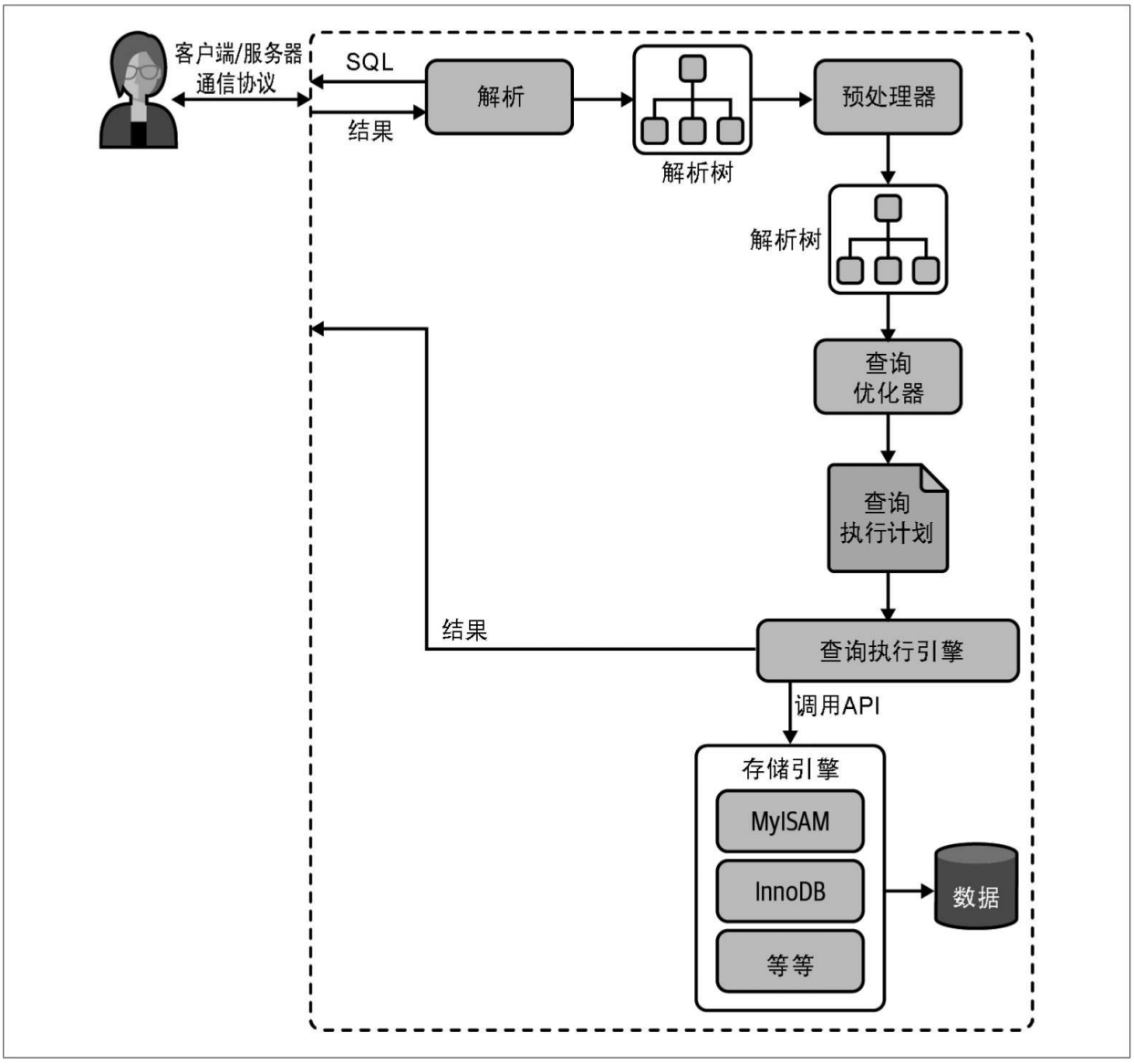

图8-1：查询执行路径

上面的每一步都比想象的要复杂，我们在后续章节中将继续讨论。我们会看到，在每一个阶段，查询处于何种状态。查询优化器是其中特别复杂也特别难理解的部分。还有很多的例外情况，例如，当查询使用绑定变量后，执行路径会有所不同，我们将在下一章讨论这点。

MySQL的客户端/服务器通信协议

一般来说，不需要去理解MySQL通信协议的内部实现细节，只需要大致理解通信协议是如何工作的。MySQL的客户端和服务器之间的通信协议是“半双工”的，这意味着，在任何时刻，要么是由服务器向客户端发送数据，要么是由客户端向服务器发送数据，这两个动作不能同时发生。所以，我们无法也无须将一个消息切成小块来独立发送。

这种协议让MySQL通信变得简单快速，但是也从很多地方限制了MySQL。一个明显的限制是，这意味着没法进行流量控制。一旦一端开始发送消息，另一端要接收完整个消息才能响应它。这就像来回抛球的游戏：在任何时刻，只有一个人能控制球，而且只有控制球的人才能将球抛回去（发送消息）。

客户端用一个单独的数据包将查询传给服务器。这也是为什么当查询的语句很长的时候，参数max_allowed_packet就特别重要了。[4]一旦客户端发送了请求，它能做的事情就只是等待结果了。

然而，一般的服务器响应给用户的数据通常很多，由多个数据包组成。当服务器开始响应客户端请求时，客户端必须完整地接收整个返回结果，而不能简单地只取前面几条结果，然后让服务器停止发送数据。在这种情况下，客户端若接收完整的结果，然后取前面几条需要的结果，或者接收完几条结果后就“粗暴”地断开连接，都不是好主意。这也是在必要的时候一定要在查询中加上LIMIT限制的原因。

换一种方式解释这种行为：当客户端从服务器取数据时，看起来是一个拉数据的过程，但实际上是MySQL在向客户端推送数据的过程。客户端不断地接收从服务器推送的数据，客户端也没法让服务器停下来。客户端像是“从消防水管喝水”（这是一个术语）。

多数连接MySQL的库函数都可以获得全部结果集并将结果缓存到内存里，还可以逐行获取需要的数据。默认一般是获得全部结果集并将它们缓存到内存中。MySQL通常需要等所有的数据都已经发送给客户端才能释放这条查询所占用的资源，所以接收全部结果并缓存通常可以减少服务器的压力，让查询能够早点结束、早点释放相应的资源。

当使用多数连接MySQL的库函数从MySQL获取数据时，其结果看起来都像是从MySQL服务器获取数据，而实际上都是从这个库函数的缓存获取数据。多数情况下这没什么问题，但是在需要返回一个很大的结果集的时候，这样做并不好，因为库函数会花很多时间和内存来存储所有的结果集。如果能够尽早开始处理这些结果集，就能大大减少内存的消耗，在这种情况下可以不使用缓存来记录结果而是直接处理。这样做的缺点是，对于服务器来说，需要查询完成后才能释放资源，所以在和客户端交互的整个过程中，服务器的资源都是被这个查询所占用的。[5]

我们看看当使用PHP的时候是什么情况。下面是我们连接MySQL的通常写法：

```
<?php
$link = mysql_connect('localhost', 'user', 'p4ssword');
$result = mysql_query('SELECT * FROM HUGE_TABLE', $link);

while ( $row = mysql_fetch_array($result) ) {
 // Do something with result
}
?>
```

这段代码看起来像是只有当你需要的时候，才通过循环从服务器端取出数据。而实际上，在上面的代码中，在调用mysql_query()的时候，PHP就已经将整个结果集缓存到内存中了。while循环只是从这个缓存中逐行取出数据。然而，如果使用下面的查询，用mysql_unbuffered_query()代替mysql_query()，PHP则不会缓存结果：

```
<?php
$link = mysql_connect('localhost', 'user', 'p4ssword');
$result = mysql_unbuffered_query('SELECT * FROM HUGE_TABLE', $link);
while ( $row = mysql_fetch_array($result) ) {
 // Do something with result
}
?>
```

不同的编程语言处理缓存的方式不同。例如，在Perl的DBD：mysql驱动中需要指定C客户端库的mysql_use_result属性（默认是mysql_buffer_result）。下面是一个例子：

```
#!/usr/bin/perl
use DBI;
my $dbh = DBI->connect('DBI:mysql:;host=localhost', 'user', 'p4ssword');
my $sth = $dbh->prepare('SELECT * FROM HUGE_TABLE', { mysql_use_result => 1 });
$sth->execute();
while ( my $row = $sth->fetchrow_array() ) {
 # Do something with result
}
```

注意，上面的prepare()调用指定了mysql_use_result属性为1，所以应用将直接使用返回的结果集而不会将其缓存。也可以在连接MySQL的时候指定这个属性，这会让整个连接都使用不缓存的方式处理结果集：

```
my $dbh = DBI->connect('DBI:mysql:;mysql_use_result=1', 'user', 'p4ssword');
```

查询状态

对于一个MySQL连接，或者一个线程，任何时刻都有一个状态，该状态表示了MySQL当前正在做什么。有很多种方式能查看当前的状态，最简单的是使用SHOW FULL PROCESSLIST命令（该命令返回结果中的Command列，其就表示当前的状态）。在一个查询的生命周期中，状态会变化很多次。MySQL官方手册中对这些状态值的含义有最权威的解释，下面将这些状态列出来，并做一个简单的解释。

Sleep

线程正在等待客户端发送新的请求。

Query

线程正在执行查询或者正在将结果发送给客户端。

Locked

在MySQL服务器层，该线程正在等待表锁。在存储引擎级别实现的锁，例如InnoDB的行锁，并不会体现在线程状态中。

Analyzing and statistics

线程正在检查存储引擎的统计信息，并优化查询。

Copying to tmp table [on disk]

线程正在执行查询，并且将其结果集复制到一个临时表中，这种状态一般要么是在做GROUP　BY操作，要么是在进行文件排序操作，或者是在进行UNION操作。如果这个状态后面还有“on disk”标记，那表示MySQL正在将一个内存临时表放到磁盘上。

Sorting result

线程正在对结果集进行排序。

了解这些状态的基本含义非常有用，这可以让你很快地了解当前“谁正在持球”。在一个繁忙的服务器上，可能会看到大量的不正常的状态，例如，statistics正占用大量的时间。这通常表示，某个地方有异常了。

查询优化处理

查询的生命周期的下一步是将一个SQL查询转换成一个执行计划，MySQL再依照这个执行计划和存储引擎进行交互。这包括多个子阶段：解析SQL、预处理、优化SQL执行计划。这个过程中产生的任何错误（例如，语法错误）都可能终止查询。这里不打算详细介绍MySQL的内部实现，而只是选择性地介绍其中几个独立的部分，在实际执行中，这几部分可能一起执行也可能单独执行。我们的目的是帮助大家理解MySQL如何执行查询，以便写出更优秀的查询。

语法解析器和预处理

首先，MySQL通过关键字将SQL语句进行解析，并生成一棵对应的“解析树”。MySQL解析器将使用MySQL语法规则验证和解析查询。例如，它将验证是否使用了错误的关键字，使用关键字的顺序是否正确，或者它还会验证引号是否能前后正确匹配。

然后，预处理器检查生成的解析树，以查找解析器无法解析的其他语义，例如，这里将检查数据表和数据列是否存在，还会解析名字和别名，看看它们是否有歧义。

下一步预处理器会验证权限。这通常很快，除非服务器上有非常多的权限配置。

查询优化器

现在解析树被认为是合法的了，并且由优化器将其转化成查询执行计划。一条查询可以有很多种执行方式，最后都返回相同的结果。优化器的作用就是找到这其中最好的执行计划。

MySQL使用基于成本的优化器，它将尝试预测一个查询使用某种执行计划时的成本，并选择其中成本最小的一个。最初，成本的最小单位是随机读取一个4KB数据页的成本，后来成本计算公式变得更加复杂，并且引入了一些“因子”来估算某些操作的代价，如执行一次WHERE条件比较的成本。可以通过查询当前会话的Last_query_cost的值来得知MySQL计算的当前查询的成本：

```
mysql> SELECT SQL_NO_CACHE COUNT(*) FROM sakila.film_actor;
+----------+
| count(*) |
+----------+
|     5462 |
+----------+

mysql> SHOW STATUS LIKE 'Last_query_cost';
+-----------------+-------------+
| Variable_name   | Value       |
+-----------------+-------------+
| Last_query_cost | 1040.599000 |
+-----------------+-------------+
```

这个结果表示，MySQL的优化器认为大概需要做1040个数据页的随机查找才能完成上面的查询。这是根据一系列的统计信息计算得来的：每个表或者索引的页面个数、索引的基数（索引中不同值的数量）、索引和数据行的长度、索引分布情况。优化器在评估成本的时候并不考虑任何层面的缓存带来的影响，它假设读取任何数据都需要一次磁盘I/O。

有很多种原因会导致MySQL优化器选择错误的执行计划，如下所示：

- 统计信息不准确。MySQL服务器依赖存储引擎提供的统计信息来评估成本，但是有的存储引擎提供的信息是准确的，有的偏差可能非常大。例如，InnoDB因为其MVCC的架构，并不能维护一个数据表的行数的精确统计信息。
- 成本指标并不完全等同于运行查询的实际成本，因此即使统计数据是准确的，查询的成本也可能超过或者低于MySQL估算的近似值。例如，有时候某个执行计划虽然需要读取更多的页面，但是它的成本却更低。因为如果这些页面都是顺序读或者这些页面都已经在内存中的话，那么它的访问成本将很低。MySQL并不知道哪些页面在内存中、哪些在磁盘中，所以查询在实际执行过程中到底需要多少次物理I/O是无法得知的。
- MySQL的最优可能和你想的最优不一样。你可能希望执行时间尽可能短，但是MySQL只是基于其成本模型选择最优的执行计划，而有些时候这并不是最快的执行方式。所以，这里我们看到的根据执行成本来选择执行计划并不是完美的模型。
- MySQL从不考虑其他并发执行的查询，这可能会影响到当前查询的速度。
- MySQL也并不是任何时候都是基于成本的优化。它有时也会基于一些固定的规则，例如，如果存在全文搜索的MATCH()子句，则在存在FULLTEXT索引的时候就使用全文索引。即使有时候使用其他索引和WHERE条件可以远比这种方式要快，MySQL也仍然会使用对应的全文索引。
- MySQL不会考虑不受其控制的操作的成本，例如，执行存储函数或者用户自定义

函数的成本。

- 后面我们还会看到，优化器有时候无法估算所有可能的执行计划，所以它可能错过实际上最优的执行计划。

MySQL的查询优化器是一个非常复杂的软件，它使用了很多优化策略来生成一个最优的执行计划。优化策略可以简单地分为两种，一种是静态优化，一种是动态优化。静态优化可以直接对解析树进行分析，并完成优化。例如，优化器可以通过一些简单的代数变换将WHERE条件转换成另一种等价形式。静态优化不依赖于特别的数值，如WHERE条件中带入的一些常数等。静态优化在第一次完成后就一直有效，即使使用不同的参数重复执行查询也不会发生变化，可以认为这是一种“编译时优化”。

然而，动态优化则和查询的上下文有关，也可能和很多其他因素有关，例如WHERE条件中的取值、索引中条目对应的数据行数等。这需要在每次查询的时候都重新评估，可以认为这是“运行时优化”。

在执行绑定变量和存储过程的时候，动态优化和静态优化的区别非常重要。MySQL对查询的静态优化只需要做一次，但对查询的动态优化则在每次执行时都需要重新评估。有时候甚至在查询的执行过程中也会重新优化。[6]

下面是一些MySQL能够处理的优化类型。

重新定义联接表的顺序

数据表的联接并不总是按照在查询中指定的顺序进行。决定联接的顺序是优化器很重要的一个功能，本章后面将深入介绍这一点。

将外联接转化成内联接

并不是所有的OUTER JOIN语句都必须以外联接的方式执行。诸多因素，例如WHERE条件、库表结构都可能会让外联接等价于一个内联接。MySQL能够识别这一点并重写查询，让其可以调整联接顺序。

使用代数等价变换规则

MySQL可以使用一些代数等价变换规则来简化并规范表达式。它可以合并和减少一些比较，还可以移除一些恒成立和一些恒不成立的判断。例如，（5=5 AND a>5）将被改写为a>5。类似地，如果有（a<b AND b=c）AND a=5则会改写为b>5 AND b=c AND a=5。这些规则对于编写条件语句很有用，我们将在本章后面的内容中继续讨论。

优化COUNT()、MIN()和MAX()

索引和列是否可为空通常可以帮助MySQL优化这类表达式。例如，要找到某一列的最小值，只需要查询对应B-tree索引最左端的记录，MySQL可以直接获取索引的第一行记录。在优化器生成执行计划的时候就可以利用这一点，在B-tree索引中，优化器会将这个表达式作为一个常数对待。类似地，如果要查找一个最大值，也只需读取B-tree索引的最后一条记录。如果MySQL使用了这种类型的优化，那么在EXPLAIN中就可以看到“Select tables optimized away”。从字面意思可以看出，它表示优化器已经从执行计划中移除了该表，并以一个常数代替。

预估并转化为常数表达式

当MySQL检测到一个表达式可以转化为常数的时候，就会一直把该表达式作为常数进行优化处理。例如，一个用户自定义变量在查询中没有发生变化时就可以将其转换为一个常数。数学表达式则是另一种典型的例子。

让人惊讶的是，在优化阶段，有时候一个查询也能够转化为一个常数。一个例子是在索引列上执行MIN()函数。甚至是主键或者唯一键查找语句也可以被转换为常数表达式。如果WHERE子句中使用了该类索引的常数条件，MySQL可以在查询开始阶段就先查找到这些值，这样优化器就能够知道并将其转换为常数表达式。下面是一个例子：

```
mysql> EXPLAIN SELECT film.film_id, film_actor.actor_id
    -> FROM sakila.film
    -> INNER JOIN sakila.film_actor USING(film_id)
    -> WHERE film.film_id = 1\G
*************************** 1. row ***************************
 id: 1
 select_type: SIMPLE
 table: film
 partitions: NULL
 type: const
 possible_keys: PRIMARY
 key: PRIMARY
 key_len: 2
 ref: const
 rows: 1
 filtered: 100.00
 Extra: Using index
*************************** 2. row ***************************
 id: 1
 select_type: SIMPLE
 table: film_actor
 partitions: NULL
 type: index
 possible_keys: NULL
 key: PRIMARY
 key_len: 4
 ref: const
 rows: 5462
 filtered: 10.00
 Extra: Using where; Using index
```

MySQL分两步来执行这个查询，也就是上面执行计划的两行输出。第一步先从film表找到需要的行。因为在film_id列上有主键索引，所以MySQL优化器知道这只会返回一行数据，优化器在生成执行计划的时候，就已经通过索引信息知道将返回多少行数据了。因为查询优化器已经明确知道有多少个值（WHERE条件中的值）需要做索引查询，所以这里的表访问类型是const。

在执行计划的第二步时，MySQL将第一步中返回的film_id列当作一个已知取值的列来处

理。因为优化器清楚在第一步执行完成后，该值就会是明确的了。注意，正如在第一步中一样，使用film_actor字段对表的访问类型也是const。

另一种会看到常数条件的情况是通过等式将常数值从一个表传到另一个表，这可以通过WHERE、USING或者ON语句来限制某列取值为常数。在上面的例子中，因为使用了USING子句，所以优化器知道这会限制film_id在整个查询过程中始终都是一个常量——因为它必须等于WHERE子句中的那个取值。

覆盖索引扫描

当索引中的列包含所有查询中需要使用的列的时候，MySQL就可以使用索引返回需要的数据，而无须查询对应的数据行，这在前面的章节中已经讨论过了。

子查询优化

MySQL在某些情况下可以将子查询转换为一种效率更高的形式，从而减少多个查询多次对数据进行访问。

提前终止查询

在发现已经满足查询需求的时候，MySQL总是能够立刻终止查询。一个典型的例子就是当使用了LIMIT子句的时候。除此之外，MySQL在其他几类情况下也会提前终止查询，例如发现了一个不成立的条件，这时MySQL可以立刻返回一个空结果。从下面的例子中可以看到这一点：

```
mysql> EXPLAIN SELECT film.film_id FROM sakila.film WHERE film_id = -1;
*************************** 1. row ***************************
 id: 1
 select_type: SIMPLE
 table: NULL
 partitions: NULL
 type: NULL
 possible_keys: NULL
 key: NULL
 key_len: NULL
 ref: NULL
 rows: NULL
 filtered: NULL
 Extra: Impossible WHERE
```

从这个例子中可看到，查询在优化阶段就已经终止。除此之外，MySQL在执行过程中，如果发现某些特殊的条件，则会提前终止查询。当查询执行引擎需要检索“不同取值”或者判断存在性的时候，MySQL都可以使用这类优化。例如，我们现在需要找到没有演员的所有电影：[7]

```
SELECT film.film_id
FROM sakila.film
LEFT OUTER JOIN sakila.film_actor USING(film_id)
WHERE film_actor.film_id IS NULL;
```

这个查询将会过滤掉所有有演员的电影。每一部电影可能都会有很多的演员，但是上面的查询一旦找到任何一个演员，就会停止并立刻判断下一部电影，因为只要有一名演员，那么WHERE条件就会过滤掉这部电影。类似这种“不同值/不存在”的优化一般可用于DISTINCT、NOT EXIST()或者LEFT JOIN类型的查询。

等值传播

如果两列的值可通过等式联接，那么MySQL能够把其中一列的WHERE条件传递到另一列上。例如，我们看下面的查询：

```
SELECT film.film_id
FROM sakila.film
INNER JOIN sakila.film_actor USING(film_id)
WHERE film.film_id > 500;
```

因为这里使用了film_id字段进行等值联接，MySQL知道这里的WHERE子句不仅适用于film表，而且对于film_actor表同样适用。如果使用的是其他的数据库管理系统，可能还需要手动通过一些条件来告知优化器这个WHERE条件适用于两个表，那么写法就会如下：

```
... WHERE film.film_id > 500 AND film_actor.film_id > 500
```

在MySQL中这是不必要的，这样写反而会让查询更难维护。

列表IN()的比较

在很多数据库服务器中，IN()完全等同于多个OR条件的子句，因为这两者是完全等价的。在MySQL中这点是不成立的，MySQL将IN()列表中的数据先进行排序，然后通过二分查找的方式来确定列表中的值是否满足条件，这是一个O（log$n$）复杂度的操作，等价地转换成OR查询的复杂度为O（n），对于IN()列表中有大量取值的时候，MySQL的处理速度将会更快。

上面列举的远不是MySQL优化器的全部，MySQL还会做大量其他的优化，即使本章全部用来描述这一点也会篇幅不足，但上面的这些例子已经足以让大家明白优化器的复杂性和智能性了。如果说从上面这段讨论中我们应该学到什么，那就是不要自以为比优化器更聪明。最终你可能会占一些便宜，但是有可能会使查询变得更加复杂而难以维护，而最终的收益为零。让优化器按照它的方式工作就可以了。

当然，虽然优化器已经很智能了，但是有时候也无法给出最优的结果。有时候你可能比优化器更了解数据，例如，由于应用逻辑使得某些条件总是成立；还有时候，优化器缺少某种功能特性，如哈希索引；再如前面提到的，从优化器的执行成本角度评估出来的最优执行计划，在实际运行中可能比其他的执行计划更慢。

如果能够确认优化器给出的不是最佳选择，并且清楚背后的原理，那么也可以帮助优化器做进一步的优化。例如，可以在查询中添加hint提示[8]，也可以重写查询，或者重新设计更优的库表结构，或者添加更合适的索引。

表和索引的统计信息

重新回忆一下图1-1，MySQL架构由多个层次组成。在服务器层有查询优化器，却没有保

存数据和索引的统计信息。统计信息由存储引擎实现，不同的存储引擎可能会存储不同的统计信息（也可以按照不同的方式存储统计信息）。

因为服务器没有存储任何统计信息，所以MySQL查询优化器在生成查询的执行计划时，需要向存储引擎获取相应的统计信息。存储引擎则给优化器提供对应的统计信息，包括：每个表或者索引有多少个页面、每个表的每个索引的基数是多少、数据行和索引的长度是多少、索引的分布信息等。优化器根据这些信息来选择一个最优的执行计划。在后面的小节中我们将看到统计信息是如何影响优化器的。

MySQL如何执行联接查询

MySQL中使用的术语“联接”（对应英文为Join）的范围可能比你熟悉的更广泛。总的来说，MySQL认为每一个查询都是联接——不仅是匹配两张表中对应行的查询，而是每一个查询、每一个片段（包括子查询，甚至基于单表的SELECT）都是联接。因此，理解MySQL如何执行联接查询是非常重要的。

所以，理解MySQL如何执行UNION查询至关重要。我们先来看一个UNION查询的例子。对于UNION查询，MySQL先将一系列的单个查询结果放到一个临时表中，然后再重新读出临时表中的数据来完成UNION查询。在MySQL的概念中，每个查询都是一次联接，所以读取临时表的结果也是一次联接。

当前MySQL的联接执行策略很简单：MySQL对任何联接都执行嵌套循环联接操作，即MySQL先在一个表中循环取出单条数据，然后再嵌套循环到下一个表中寻找匹配的行，依次下去，直到找到所有表中匹配的行为止。最后根据各个表匹配的行，返回查询中需要的各列。MySQL会尝试在最后一个联接表中找到所有匹配的行，如果最后一个联接表无法找到更多的行，MySQL返回到上一层次的联接表，看是否能够找到更多的匹配记录，依此类推，迭代执行。[9]

在MySQL 8.0.20版本之后，已经不再使用基于块的嵌套循环联接操作，取而代之的是哈希联接（参见链接33）。这让联接操作性能变得更好，特别是当数据集可以全部存储在内存时。

执行计划

和很多其他关系数据库不同，MySQL并不会生成查询字节码来执行查询。MySQL生成查询的一棵指令树[10]，然后通过查询执行引擎执行完成这棵指令树并返回结果。最终的执行计划包含了重构查询的全部信息。如果你对某个查询执行EXPLAIN EXTENDED后，再执行SHOW WARNINGS，就可以看到重构出的查询。[11]

任何多表查询都可以使用一棵树来表示，例如，可以按照图8-2执行一个四表的联接操作。

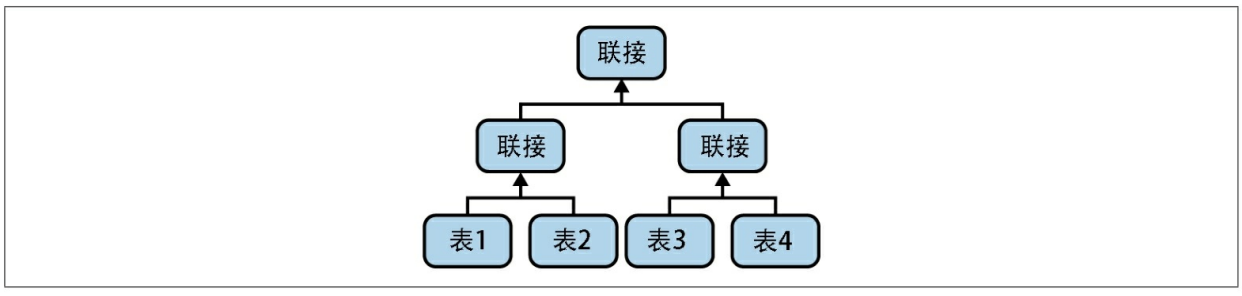

图8-2：多表联接的一种方式

在计算机科学中，这被称为一棵平衡树。但是，这并不是MySQL执行查询的方式。正如我们在前面章节中介绍的，MySQL总是从一个表开始，一直嵌套循环、回溯完成所有表联接。所以，MySQL的执行计划总是如图8-3所示，是一棵左侧深度优先的树。

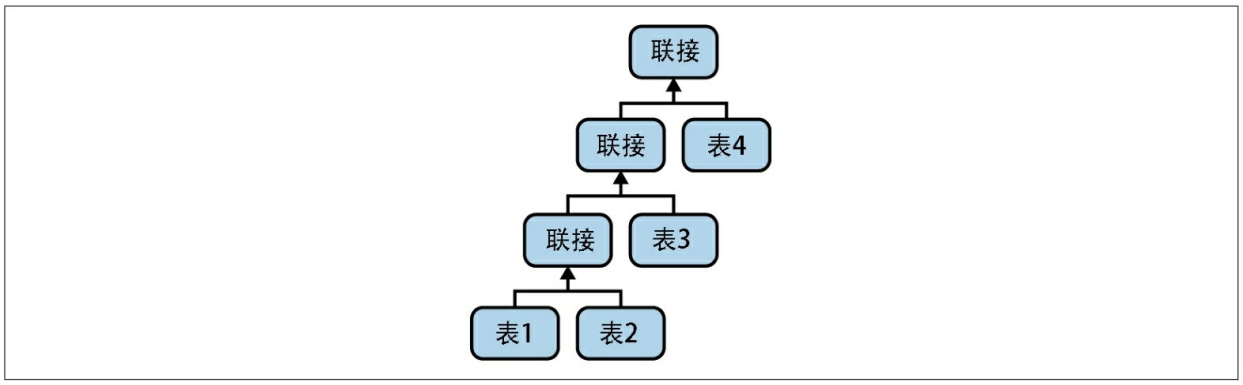

图8-3：MySQL如何实现多表联接

联接查询优化器

MySQL查询优化器最重要的一部分就是联接查询优化器，它决定了多个表联接时的顺序。通常多表联接的时候，可以有多种不同的联接顺序来获得相同的执行结果。联接查询优化器通过评估不同顺序时的成本来选择一个成本最低的联接顺序。

下面的查询可以通过不同顺序的联接最后获得相同的结果：

```
SELECT film.film_id, film.title, film.release_year, actor.actor_id,
actor.first_name, actor.last_name
FROM sakila.film
INNER JOIN sakila.film_actor USING(film_id)
INNER JOIN sakila.actor USING(actor_id);
```

很容易看出，可以通过一些不同的执行计划来完成上面的查询。例如，MySQL可以从film表开始，使用film_actor表的索引film_id来查找对应的actor_id值，然后再根据actor表的主键找到对应的记录。Oracle用户会用下面的术语描述：“film表作为驱动表先查找file_actor表，然后以此结果为驱动表再查找actor表”。这样做效率应该很高，我们再使用EXPLAIN看看MySQL将如何执行这个查询：

```
*************************** 1. row ***************************
 id: 1
 select_type: SIMPLE
 table: actor
 partitions: NULL
 type: ALL
 possible_keys: PRIMARY
 key: NULL
 key_len: NULL
 ref: NULL
 rows: 200
 filtered: 100.00
 Extra: NULL
*************************** 2. row ***************************
 id: 1
 select_type: SIMPLE
 table: film_actor
 partitions: NULL
 type: ref
 possible_keys: PRIMARY,idx_fk_film_id
 key: PRIMARY
 key_len: 2
 ref: sakila.actor.actor_id
 rows: 27
 filtered: 100.00
 Extra: Using index
*************************** 3. row ***************************
 id: 1
 select_type: SIMPLE
 table: film
 partitions: NULL
 type: eq_ref
 possible_keys: PRIMARY
 key: PRIMARY
 key_len: 2
 ref: sakila.film_actor.film_id
 rows: 1
 filtered: 100.00
 Extra: NULL
```

这和我们前面给出的执行计划完全不同。MySQL从actor表开始（从上面的EXPLAIN结果的第一行输出可以看出这一点），然后与我们前面的计划按照相反的顺序进行联接。这样是否效率更高呢？我们来看看。我们先使用STRAIGHT_JOIN关键字，按照之前的顺序执行，下面是对应的EXPLAIN输出结果：

```
mysql> EXPLAIN SELECT STRAIGHT_JOIN film.film_id...\G
*************************** 1. row ***************************
 id: 1
 select_type: SIMPLE
 table: film
 partitions: NULL
 type: ALL
 possible_keys: PRIMARY
 key: NULL
 key_len: NULL
 ref: NULL
 rows: 1000
 filtered: 100.00
 Extra: NULL
*************************** 2. row ***************************
 id: 1
 select_type: SIMPLE
 table: film_actor
 partitions: NULL
 type: ref
 possible_keys: PRIMARY,idx_fk_film_id
 key: idx_fk_film_id
 key_len: 2
 ref: sakila.film.film_id
 rows: 5
 filtered: 100.00
 Extra: Using index
*************************** 3. row ***************************
 id: 1
 select_type: SIMPLE
 table: actor
 partitions: NULL
 type: eq_ref
 possible_keys: PRIMARY
 key: PRIMARY

 key_len: 2
 ref: sakila.film_actor.actor_id
 rows: 1
 filtered: 100.00
 Extra: NULL
```

这说明了MySQL为什么要反转联接顺序：反转后可以使查询在第一个表中检查更少的行。在这两种情况下，都能够在第二个和第三个表中执行快速索引查找，不同的是，需要执行的索引查找次数不一样。将film表作为第一个表需要检查大约1000行记录（参见rows字段），每一行都是一个探针，用于针对film_actor和actor表进行索引查找。如果MySQL先扫描actor表，则只需要对后面的表进行200次索引查找。换句话说，反转联接顺序会让查询进行更少的回溯和重读操作。

这个简单的例子主要想说明MySQL是如何选择合适的联接顺序来让查询执行的成本尽可能低的。重新定义联接的顺序是优化器非常重要的一项功能。不过有的时候，优化器给出的并不是最优的联接顺序。这时可以使用STRAIGHT_JOIN关键字重写查询，让优化器按照你认为的最优的联接顺序执行——不过老实说，人的判断很难那么精准。绝大多数时候，优化器做出的选择都比普通人的判断要更准确。

联接优化器会尝试在所有的联接顺序中选择一个成本最低的来生成执行计划树。如果可能，优化器会遍历每一个表，然后逐个做嵌套循环，计算执行每一棵可能的计划树的成本，最后返回一个最优的执行计划。

不过，糟糕的是，*n*个表的联接可能有*n*的阶乘种联接顺序，我们称之为所有可能的查询计划的“搜索空间”。搜索空间的增长速度非常块，例如，若是10个表的联接，那么共有3628800种不同的联接顺序！当搜索空间非常大的时候，优化器不可能逐一评估每一种联接顺序的成本。这时，优化器选择使用“贪婪”搜索的方式查找“最优”的联接顺序。实际上，当需要联接的表超过optimizer_search_depth的限制的时候，就会选择“贪婪”搜索模式了（optimizer_search_depth参数可以根据需要指定大小）。

在MySQL这些年的发展过程中，优化器积累了很多“启发式”的优化策略来加速执行计划的生成。在绝大多数情况下这都是有效的，但因为不会去计算每一种联接顺序的成本，所以偶尔也会选择不是最优的执行计划。

有时查询不能重新排序，联接优化器可以利用这一点通过消除选择来减小搜索空间。左联接（LEFT JOIN）和相关子查询都是很好的例子（稍后将详细介绍子查询）。这是因为，一个表的结果依赖于另外一个表中检索的数据，这种依赖关系通常可以帮助联接优化器通过消除选择来减少搜索空间。

排序优化

无论如何排序都是一个成本很高的操作，所以从性能角度考虑，应尽可能避免排序或者尽可能避免对大量数据进行排序。

当不能使用索引生成排序结果的时候，MySQL需要自己进行排序，如果数据量小则在内存中进行，如果数据量大则需要使用磁盘，不过MySQL将这个过程统一称为文件排序（filesort），即使完全是在内存中排序不需要任何磁盘文件时也是如此。

如果需要排序的数据量小于“排序缓冲区”，MySQL使用内存进行快速排序操作。如果内存不够排序，那么MySQL会先将数据分块，对每个独立的块使用“快速排序”进行排序，并将各个块的排序结果存放在磁盘上，然后将各个排好序的块进行合并（merge），最后返回排序结果。

MySQL有如下两种排序算法。

两次传输排序（旧版本使用）

> 读取行指针和需要排序的字段，对其进行排序，然后再根据排序结果读取所需要的数据行。
>
> 这需要进行两次数据传输，即需要从数据表中读取两次数据，第二次读取数据的时候，因为是读取排序列进行排序后的所有记录，这会产生大量的随机I/O，所以两次传输排序的成本非常高。

单次传输排序（新版本使用）

先读取查询所需要的所有列，然后再根据给定列进行排序，最后直接返回排序结果。因为不再需要从数据表中读取两次数据，对于I/O密集型的应用来说，这样做的效率高了很多。另外，相比两次传输排序，这个算法只需要一次顺序I/O就可读取所有的数据，而无须任何的随机I/O。然而，这种方式可能占用更多空间，因为会保存查询中每一行所需要的列，而不仅仅是进行排序操作所需要的列。这意味着更少的元组可以放入排序缓冲区，使得文件排序（filesort）操作必须执行更多的排序合并过程。

MySQL在进行文件排序时需要使用的临时存储空间可能会比想象的要大得多。原因在于MySQL在排序时，对每一个排序记录都会分配一个足够长的定长空间来存放。这个定长空间必须足够长才能容纳其中最长的字符串，例如，如果是VARCHAR列，则需要分配其完整长度；如果使用utf8mb4字符集，那么MySQL将会为每个字符预留4字节。我们曾经在一个库表结构设计不合理的案例中看到，排序消耗的临时空间比磁盘上的原表要大很多倍。

在联接查询的时候如果需要排序，MySQL会分两种情况来处理这样的文件排序。如果ORDER BY子句中的所有列都来自联接的第一个表，那么MySQL在联接处理第一个表的时候就进行文件排序。如果是这样，那么在MySQL的EXPLAIN结果中可以看到Extra字段会有“Using filesort”字样。除此之外的所有情况，MySQL都会先将联接的结果存放到一个临时表中，然后在所有的联接都结束后，再进行文件排序。在这种情况下，在MySQL的EXPLAIN结果的Extra字段可以看到“Using temporary；Using filesort”字样。如果查询中有LIMIT的话，LIMIT也会在文件排序之后应用，所以即使需要返回较少的数据，临时表和需要排序的数据量仍然会非常大。

查询执行引擎

在解析和优化阶段，MySQL将生成查询对应的执行计划，MySQL的查询执行引擎会根据这个执行计划来完成整个查询。这里的执行计划是一个数据结构，而不是和很多其他的关系数据库那样生成对应的可执行的字节码。

相对于查询优化阶段，查询执行阶段不是那么复杂：MySQL只是简单地根据执行计划给出的指令逐步执行。在根据执行计划逐步执行的过程中，有大量的操作需要通过调用存储引擎实现的接口来完成，这些接口也就是我们称为“handler API”的接口。查询中的每一个表都由一个handler的实例表示。如果一个表在查询中出现了三次，服务器会创建三个handler对象。前面我们有意忽略了这一点，实际上，MySQL在优化阶段就为每个表创建了一个handler实例，优化器根据这些实例的接口可以获取表的相关信息，包括表的所有列名、索引统计信息，等等。

存储引擎接口有着非常丰富的功能，但是底层接口却只有几十个，这些接口像“搭积木”一样能够完成查询的大部分操作。例如，有一个查询某个索引的第一行的接口，其还有查询某个索引条目的下一个条目的功能，有了这两个功能就可以完成全索引扫描的操作了。这种简单的接口模式，让MySQL的存储引擎的插件式架构成为可能，但是正如前面的讨论，这也给优化器带来了一定的限制。

并不是所有的操作都由handler完成。例如，当MySQL需要进行表锁的时候。handler可能会实现自己的级别的、更细粒度的锁，如InnoDB就实现了自己的行级基本锁，但这并不能代替服务器层的表锁。正如我们在第1章所介绍的，如果是所有存储引擎共有的特性则由服务器层实现，比如时间和日期函数、视图、触发器等。

为了执行查询，MySQL只需要重复执行计划中的各个操作，直到完成所有的数据查询。

将结果返回给客户端

执行查询的最后一个阶段是将结果返回给客户端。即使查询不需要给客户端返回结果集，MySQL仍然会返回这个查询的一些信息，如该查询影响到的行数。

MySQL将结果集返回客户端是一个增量且逐步返回的过程。例如，我们回头看看前面的联接操作，一旦服务器处理完最后一个联接表，开始生成第一条结果时，MySQL就可以开始向客户端逐步返回结果集了。这样处理有两个好处：服务器端无须存储太多的结果，也就不会因为要返回太多结果而消耗太多内存。另外，这样的处理也可让MySQL客户端第一时间获得返回的结果。[12]结果集中的每一行都会以一个满足MySQL客户端/服务器通信协议的封包发送，再通过TCP协议进行传输，在TCP传输的过程中，可能对MySQL的封包进行缓存，然后批量传输。

MySQL查询优化器的局限性

MySQL所实现的查询执行方式并不是对每种查询都是最优的。不过还好，MySQL查询优化器只对少部分查询不适用，而且我们往往可以通过改写查询让MySQL高效地完成工作。

UNION的限制

有时，MySQL无法将限制条件从UNION的外层“下推”到内层，这使得原本能够限制部分返回结果的条件无法应用到内层查询的优化上。

如果希望UNION的各个子句能够根据LIMIT只取部分结果集，或者希望能够先排好序再合并结果集的话，就需要在UNION的各个子句中分别使用这些子句。例如，想将两个子查询结果联合起来，然后再取前20条记录，那么MySQL会将两个表存放到同一个临时表中，然后再取出前20行记录：

```
(SELECT first_name, last_name
 FROM sakila.actor
 ORDER BY last_name)
UNION ALL
(SELECT first_name, last_name
 FROM sakila.customer
 ORDER BY last_name)
LIMIT 20;
```

这条查询将会把actor表中的200条记录和customer表中的599条记录存放在一个临时表中，然后再从临时表中取出前20条。可以通过在UNION的两个子查询中分别加上一个LIMIT 20来减少临时表中的数据：

```
(SELECT first_name, last_name
 FROM sakila.actor
 ORDER BY last_name
 LIMIT 20)
UNION ALL
(SELECT first_name, last_name
 FROM sakila.customer
 ORDER BY last_name
 LIMIT 20)
LIMIT 20;
```

现在临时表只包含40条记录了，除了考虑性能之外，在这里还需要注意一点：从临时表中取出数据的顺序并不是一定的，所以如果想获得正确的顺序，还需要在最后的LIMIT操作前加上一个全局的ORDER BY操作。

等值传递

某些时候，等值传递会带来一些意想不到的额外消耗。例如，考虑一列上的巨大IN()列表，优化器知道它将等于其他表中的一些列，这是由于WHERE、ON或USING子句使列彼此相等。

优化器通过将列表复制到所有相关表中的相应列来“共享”列表。通常，因为各个表新增了过滤条件，所以优化器可以更高效地从存储引擎过滤记录。但是如果这个列表非常大，则会导致优化和执行都会变慢。在写作本书的时候，除了修改MySQL源代码，目前还没有什么办法能够绕过该问题（不过这个问题很少会碰到）。

并行执行

MySQL无法利用多核特性来并行执行查询。很多其他的关系数据库能够提供这个特性，但是MySQL做不到。这里特别指出是想告诉读者不要花时间去尝试寻找并行执行查询的方法。

在同一个表中查询和更新

MySQL不允许对一张表同时进行查询和更新。这其实并不是优化器的限制，如果你清楚MySQL是如何执行查询的，就可以避免这种情况。下面是一段无法运行的SQL语句，尽管这是一段符合标准的SQL语句。这个查询会将表中每一行的c字段值更新为和该行的type字段值相同的行数量：

```
mysql> UPDATE tbl AS outer_tbl
    -> SET c = (
    -> SELECT count(*) FROM tbl AS inner_tbl
    -> WHERE inner_tbl.type = outer_tbl.type
    -> );
ERROR 1093 (HY000): You can't specify target table 'outer_tbl'
for update in FROM clause
```

可以使用生成表的形式来绕过上面的限制，因为MySQL只会把这个表当作一个临时表来处理。实际上，这执行了两个查询：一个是子查询中的SELECT语句，另一个是多表UPDATE查询，其中包含原表和子查询的联接结果。子查询会在UPDATE语句打开表之前就完成，所以下面的查询将会正常执行：

```
mysql> UPDATE tbl
    -> INNER JOIN(
    -> SELECT type, count(*) AS c
    -> FROM tbl
    -> GROUP BY type
    -> ) AS der USING(type)
    -> SET tbl.c = der.c;
```

优化特定类型的查询

这一节，我们将介绍如何优化特定类型的查询。在本书的其他部分会分散介绍这些优化技巧，不过这里将会汇总一下，以便参考和查阅。

本节介绍的多数优化技巧都和特定的版本有关，所以对于未来MySQL的版本未必适用。毫无疑问，某一天优化器自己也会实现这里列出的部分或者全部优化技巧。

优化**COUNT()**查询

COUNT()聚合函数，以及如何优化使用了该函数的查询，很可能是MySQL中最容易被误解的前10个话题之一。你在网上随便搜索一下就能看到很多错误的理解，可能比我们想象中的要多得多。

在做优化之前，先来看看COUNT()函数真正的作用是什么。

COUNT()的作用

COUNT()是一个特殊的函数，有两种非常不同的作用：它可以统计某列的值的数量，也可以统计行数。在统计列值时要求列值是非空的（不统计NULL）。如果在COUNT()的括号中指定了列或者列的表达式，则统计的就是这个表达式有值的结果数。因为很多人对NULL理解有问题，所以这里很容易产生误解。如果你想了解更多关于SQL语句中NULL的含义，建议阅读一些关于SQL语句基础的书籍。（关于这个话题，互联网上的一些信息是不够准确的。）

COUNT()的另一个作用是统计结果集的行数。当MySQL确认括号内的表达式值不可能为空时，实际上就是在统计行数。最简单的就是当我们使用COUNT（*）的时候，这种情况下通配符*并不会像我们猜想的那样扩展成所有的列，实际上，它会忽略所有的列而直接统计所有的行数。

我们发现最常见的错误之一是，当需要统计行数时，在COUNT()函数的括号内指定了列名。如果想要知道结果中的行数，应该始终使用COUNT（*），这样可以更清晰地传达意图，避免糟糕的性能表现。

简单优化

通常会看到这样的问题：如何在一个查询中统计同一列的不同值的数量，以减少查询的语句量。例如，假设可能需要通过一个查询返回各种不同颜色的商品数量，此时不能使用OR语句（比如，SELECT COUNT（color='blue'OR color='red'）FROM items；），因为这样做无法区分不同颜色的商品数量；也不能在WHERE条件中指定颜色（比如，SELECT COUNT（*）FROM items WHERE color='blue'AND color='RED'；），因为颜色的条件是互斥的。下面的查询可以在一定程度上解决这个问题：[13]

```
SELECT SUM(IF(color = 'blue', 1, 0)) AS blue,SUM(IF(color = 'red', 1, 0))
AS red FROM items;
```

也可以使用COUNT()而不是SUM()实现同样的目的，只需要将满足条件设置为真，不满足条件设置为NULL即可：

```
SELECT COUNT(color = 'blue' OR NULL) AS blue, COUNT(color = 'red' OR NULL)
AS red FROM items;
```

使用近似值

有时候，某些业务场景并不要求完全精确的统计值，此时可以用近似值来代替。EXPLAIN出来的优化器估算的行数就是一个不错的近似值，执行EXPLAIN并不需要真正地去执行查询，所以成本很低。

很多时候，计算精确值非常复杂，而计算近似值则非常简单。曾经有一个客户希望我们统计他的网站的当前活跃用户数是多少，这个活跃用户数保存在缓存中，过期时间为30分钟，所以每隔30分钟需要重新计算并放入缓存。这个活跃用户数本身就不是精确值，所以使用近似值代替是可以接受的。另外，如果要精确统计在线人数，使用WHERE条件会很复杂，一方面需要剔除当前非活跃用户，另一方面还要剔除系统中某些特定ID的“默认”用户，去掉这些约束条件对总数的影响很小，但却可能提升该查询的性能。更进一步的优化则可以尝试删除DISTINCT这样的约束来避免文件排序。这样重写过的查询比原来精确统计的查询快很多，而返回的结果则几乎相同。

更复杂的优化

通常来说，COUNT()查询需要扫描大量的行（意味着要访问大量数据）才能获得精确的结果，因此是很难优化的。除了前面提到的方法，在MySQL层面还能做的就只有索引覆盖扫描了。如果这还不够，那就需要考虑修改应用的架构，可以增加类似*Memcached*这样的外部缓存系统。不过，可能很快你就会陷入一个熟悉的困境：“快速、精确和实现简单”。三者永远只能满足其二，必须舍掉一个。

优化联接查询

这个话题基本上整本书都在讨论，这里需要特别提到以下几点。

- 确保ON或者USING子句中的列上有索引。在创建索引的时候就要考虑到联接的顺序。当表A和表B用列c联接的时候，如果优化器的联接顺序是B、A，那么就不需要在B表的对应列上建索引。没有用到的索引只会带来额外的负担。一般来说，除非有其他理由，否则只需在联接顺序中的第二个表的相应列上创建索引。
- 确保任何GROUP BY和ORDER BY中的表达式只涉及一个表中的列，这样MySQL才有可能使用索引来优化这个过程。
- 当升级MySQL的时候需要注意：联接语法、运算符优先级等其他可能会发生变化的地方。因为以前是普通联接的地方可能会变成笛卡儿积，不同类型的联接可能会生成不同的结果，甚至会产生语法错误。

使用**WITH ROLLUP**优化**GROUP BY**

分组查询的一个变种就是要求MySQL对返回的分组结果再做一次超级聚合。可以使用WITH ROLLUP子句来实现这种逻辑，但可能优化得不够。可以通过EXPLAIN来观察其执行计划，特别要注意分组是否是通过文件排序或者临时表实现的。然后再去掉WITH ROLLUP子句来看执行计划是否相同。也可以通过本节前面介绍的优化器提示来强制执行

计划。

很多时候，如果可以，在应用程序中做超级聚合是更好的，虽然这需要给客户端返回更多的结果。也可以在FROM子句中嵌套使用子查询，或者是通过一个临时表存放中间数据，然后和临时表执行UNION来得到最终结果。

最好的办法是尽可能地将WITH ROLLUP功能转移到应用程序中处理。

优化**LIMIT**和**OFFSET**子句

在系统中需要进行分页操作的时候，我们通常会使用LIMIT加上偏移量的办法实现，同时加上合适的ORDER BY子句。如果有对应的索引，通常效率会不错，否则，MySQL需要做大量的文件排序操作。

一个非常常见又令人头疼的问题是，在偏移量非常大的时候，例如，可能是LIMIT 1000，20这样的查询，这时MySQL需要查询10020条记录然后只返回最后20条，前面10000条记录都将被抛弃，这样的代价非常高。如果所有的页面被访问的频率都相同，那么这样的查询平均需要访问半个表的数据。要优化这种查询，要么是在页面中限制分页的数量，要么是优化大偏移量的性能。

优化此类分页查询的一个最简单的办法就是尽可能地使用索引覆盖扫描，而不是查询所有的行。然后根据需要做一次联接操作再返回所需的列。在偏移量很大的时候，这样做的效率会有非常大的提升。考虑下面的查询：

```
SELECT film_id, description FROM sakila.film ORDER BY title LIMIT 50, 5;
```

如果这个表非常大，那么这个查询最好改写成下面的样子：

```
SELECT film.film_id, film.description
FROM sakila.film
INNER JOIN (
SELECT film_id FROM sakila.film
ORDER BY title LIMIT 50, 5
) AS lim USING(film_id);
```

这种“延迟联接”之所以有效，是因为它允许服务器在不访问行的情况下检查索引中尽可能少的数据，然后，一旦找到所需的行，就将它们与整个表联接，以从该行中检索其他列。类似的技术也适用于带有LIMIT子句的联接。

有时候也可以将LIMIT查询转换为已知位置的查询，让MySQL通过范围扫描获得对应的结果。例如，如果在一个位置列上有索引，并且预先计算出了边界值，上面的查询就可以改写为：

```
SELECT film_id, description FROM sakila.film
WHERE position BETWEEN 50 AND 54 ORDER BY position;
```

对数据进行排名的问题也与此类似，但往往还会同时和GROUP BY混合使用。在这种情况下通常需要预先计算并存储排名信息。

LIMIT和OFFSET的问题，其实是OFFSET的问题，它会导致MySQL扫描大量不需要的行

然后再抛弃掉。如果可以使用书签记录上次取数据的位置，那么下次就可以直接从该书签记录的位置开始扫描，这样就可以避免使用OFFSET。例如，若需要按照租借记录做翻页，那么可以根据最新一条租借记录向回追溯，这种做法可行是因为租借记录的主键是单调增长的。首先使用下面的查询获得第一组结果：

```
SELECT * FROM sakila.rental
ORDER BY rental_id DESC LIMIT 20;
```

假设上面的查询返回的是主键为16，049到16，030的租借记录，那么下一页查询就可以从16，030这个点开始：

```
SELECT * FROM sakila.rental
WHERE rental_id < 16030
ORDER BY rental_id DESC LIMIT 20;
```

该技术的好处是无论翻页到多么靠后，其性能都会很好。

其他优化办法还包括使用预先计算的汇总表，或者联接到一个冗余表，冗余表只包含主键列和需要做排序的数据列。

优化**SQL CALC FOUND ROWS**

分页的时候，另一个常用的技巧是在LIMIT语句中加上SQL_CALC_FOUND_ROWS提示（hint），这样就可以获得去掉LIMIT以后满足条件的行数，因此可以作为分页的总数。看起来，MySQL做了一些非常“高深”的优化，像是通过某种方法预测了总行数。但实际上，MySQL只有在扫描了所有满足条件的行以后，才会知道行数，所以加上这个提示以后，不管是否需要，MySQL都会扫描所有满足条件的行，然后再抛弃掉不需要的行，而不是在满足LIMIT的行数后就终止扫描。所以该提示的代价可能非常高。

一个更好的设计是将具体的页数换成“下一页”按钮，假设每页显示20条记录，那么我们每次查询时都是用LIMIT返回21条记录并只显示20条，如果第21条存在，那么就显示“下一页”按钮，否则就说明没有更多的数据，也就无须显示“下一页”按钮了。

另一种做法是先获取并缓存较多的数据——例如，缓存1000条——然后每次分页都从这个缓存中获取。这样做可以让应用程序根据结果集的大小采取不同的策略，如果结果集小于1000，就可以在页面上显示所有的分页链接，因为数据都在缓存中，所以这样做不会对性能造成影响。如果结果集大于1000，则可以在页面上设计一个额外的“找到的结果多于1000条”之类的按钮。这两种策略都比每次生成全部结果集再抛弃不需要的数据的效率高很多。

有时候也可以考虑使用EXPLAIN的结果中的rows列的值来作为结果集总数的近似值（实际上，Google的搜索结果总数也是一个近似值）。当需要精确结果的时候，再单独使用COUNT（*）来满足需求，这时如果能够使用索引覆盖扫描则通常也会比SQL_CALC_FOUND_ROWS快得多。

优化**UNION**查询

MySQL总是通过创建并填充临时表的方式来执行UNION查询，因此很多优化策略在

UNION查询中都没法很好地被使用。经常需要手工地将WHERE、LIMIT、ORDER BY等子句“下推”到UNION的各个子查询中，以便优化器可以充分利用这些条件进行优化（例如，直接将这些子句冗余地写一份到各个子查询）。

除非你确实需要服务器消除重复的行，否则一定要使用UNION ALL，这一点很重要。如果没有ALL关键字，MySQL会给临时表加上DISTINCT选项，这会导致对整个临时表的数据做唯一性检查。这样做的代价非常高。即使有ALL关键字，MySQL仍然会使用临时表存储结果。事实上，MySQL总是将结果放入临时表，然后再读出，再返回给客户端，虽然很多时候这样做是没有必要的（例如，MySQL可以直接把这些结果返回给客户端）。

小结

如果把创建高性能应用程序比作一个环环相扣的“难题”，除了前面介绍的schema、索引和查询语句设计之外，查询优化应该是解开“难题”的最后一步。要想写一个好的查询，你必须理解schema设计、索引设计等，反之亦然。

理解查询是如何被执行的以及时间都消耗在哪些地方，依然是前面我们介绍的响应时间的一部分。如果再加上一些诸如解析和优化过程的知识，就可以更进一步地理解我们在第7章中讨论的MySQL如何访问表和索引的内容了。这也可从另一个维度帮助你理解MySQL在访问表和索引时查询和索引的关系。

优化通常需要三管齐下：不做、少做、快速地做。

---

[1] 如果应用服务器和数据库不在同一台主机上，网络开销就十分明显。即使是在同一台服务器上，也仍然会有数据传输的开销。

[2] 更多内容请参考本章后面的**“优化COUNT()查询”**一节。

[3] **Percona Toolkit**中的***pt-archiver***工具就可以安全而简单地完成这类工作。

[4] 如果查询太大，服务端会拒绝接收更多的数据并抛出相应错误。

[5] 可以使用**SQL_BUFFER_RESULT**，后面会介绍这方面的知识。

[6] 例如，在联接操作中，范围检查的执行计划会针对每一行重新评估进行索引。可以通过**EXPLAIN**执行计划中的**Extra**列是否有**“range checked for each record”**来确认这一点。该执行计划还会增加**select_full_range_join**这个服务器变量的值。

[7] 一部电影没有演员，是有点儿奇怪。不过在示例数据库**Sakila**中，影片***Slacker Liaisons***中就没有任何演员，它的描述是**“**鲨鱼和见过中国古代鳄鱼的学生的简短传说**”**。

[8] 可以通过查看**MySQL**手册中的**“**索引提示**”“**优化器提示**”**等章节来确认在某个具体的版本中支持哪些提示，以及如何使用这些提示。

[9] 如后文所述，**MySQL**的执行计划并不是这里描述的这么简单，过程中有非常多的细节优化，使得整个过程是一个非常复杂的过程。

[10] 可以通过在**EXPLAIN**语句中新增**FORMAT=TREE**关键字来查看树形结构。

[11] **MySQL**根据执行计划生成输出。这和原查询有完全相同的语义，但是查询语句可能并不完全相同。

[12] 可以通过一些办法来影响这个行为，例如，可以使用**SQL_BUFFER_RESULT**。参考**MySQL**官方手册中的**“Optimizer Hints”**一节可获得更多信息。

[13] 也可以写成这样的**SUM()**表达式：**SUM**（**color='blue'**），**SUM**（**color='red'**）。

第9章 复制

MySQL内置的复制功能是构建基于MySQL的大规模、高性能应用的基础，这类应用使用所谓的“水平扩展”的架构。我们可以通过为服务器配置一个或多个备库的方式来进行数据同步。复制功能不仅有利于构建高性能的应用，同时也是高可用性、可扩展性、灾难恢复、备份、数据分析以及数据仓库等工作的基础。

在本章中，我们的重点不是介绍每个功能，而是应该在什么场景下使用这些功能。在MySQL的官方文档（参见链接34）中，已经非常详尽地介绍了各个功能，包括半同步复制、多源复制等，在配置这些功能的时候，参考官方文档就可以了。

关于术语的说明

长久以来，MySQL的用户对复制技术相关的术语“主库”（master）和“从库”（slave）非常熟悉。这些术语已经被“源”（source）和“副本”（replica）所替代。本书中也尽量使用新的术语。一些老版本的MySQL依旧会使用老的术语，必要的时候建议参考MySQL手册。

复制概述

复制解决的基本问题是让一台服务器的数据与其他服务器保持同步。它的实现机制可以这样概括，首先在源服务器（source server）上，任何数据修改和数据结构变更的事件（event）都会被写入日志文件中，然后，副本服务器从源服务器上的日志文件中读取这些事件并在本地重放执行。这是一个异步处理的过程，也就是说，并不能保证副本服务器上的数据是最新的。复制延迟（副本数据和最新数据之间的时间差）也并没有上限。一个大的SQL查询语句可能会导致副本服务器落后于源服务器几秒钟、几分钟，甚至是几小时。MySQL的复制基本上是向后兼容的，新版本的服务器可以作为老版本的服务器的副本，但反过来，将老版本的服务器作为新版本的服务器的副本通常是不可行的，因为它可能无法解析新版本的新特性或SQL语法，而且复制使用的文件格式也可能存在差异。例如，不能从MySQL 5.6复制到MySQL 5.5。在进行大的版本升级前，例如，从5.6升级到5.7，或从5.7升级到8.0，最好先对复制的设置进行测试。但对于小版本号的升级，如从5.7.34升级到5.7.35，则通常是兼容的。阅读每次版本的更新日志可以找到不同版本间的具体变化。

通过复制可以将读操作指向副本来获得更好的读扩展性，但除非设计得当，否则并不适合通过复制来扩展写操作。在一主库多副本库的架构中，写操作会被执行多次，这时候整个系统的性能取决于写入最慢的那部分。

下面是复制比较常见的用途。

数据分发

> MySQL的复制通常不会对带宽造成很大的压力，在后面的内容中，你将会看到基于行的复制会比传统的基于语句的复制模式的带宽压力更大。你可以随意地停止或开始复制，并在不同的地理位置来分布数据备份，例如不同的数据中心。即使在不稳定的网络环境下，远程复制也可以工作。但如果为了保持很低的复制延迟，最好有一个稳定的、低延迟连接。

读流量扩展

> 通过MySQL的复制可以将读操作分布到多台服务器上，实现对读密集型应用的优化，并且实现很方便，通过简单的代码修改就能实现基本的负载均衡。对于小规模的应用，可以简单地对机器名做硬编码或使用DNS轮询（将一个机器名指向多个IP地址）。当然也可以使用更复杂的方法，例如网络负载均衡这一类的标准负载均衡解决方案，它们能够很好地将负载分配到不同的MySQL服务器上。

备份

> 复制是一项有助于备份的有价值的技术，但副本不是备份，也不能够取代备份。

分析与报告

> 为报告/分析（在线分析处理，OLAP）查询使用专用的副本是一项很好的策略，可以很好地隔离此类查询产生的压力，以避免对满足外部客户需求的在线业务产生影响。复制可以很好地实现此类隔离。

高可用性和故障切换

复制有助于避免MySQL成为应用程序中的单点故障，一个包含复制的设计良好的故障切换系统能够显著地缩短宕机时间。

MySQL升级测试

这种做法比较普遍，先使用一个更高版本的MySQL作为副本，确保查询能够在此副本上按照预期执行，再升级所有的实例。

复制如何工作

在详细介绍如何设置复制之前，让我们先看看MySQL实际上是如何复制数据的。在这里，我们使用最简单的复制拓扑结构，单一源服务器和单一副本服务器。

总的来说，复制有三个步骤：

1. 源端把数据更改记录到二进制日志中，称之为“二进制日志事件”（binary log events）。
2. 副本将源上的日志复制到自己的中继日志中。
3. 副本读取中继日志中的事件，将其重放到副本数据之上。

图9-1描述了复制的最基础形态的更多细节。

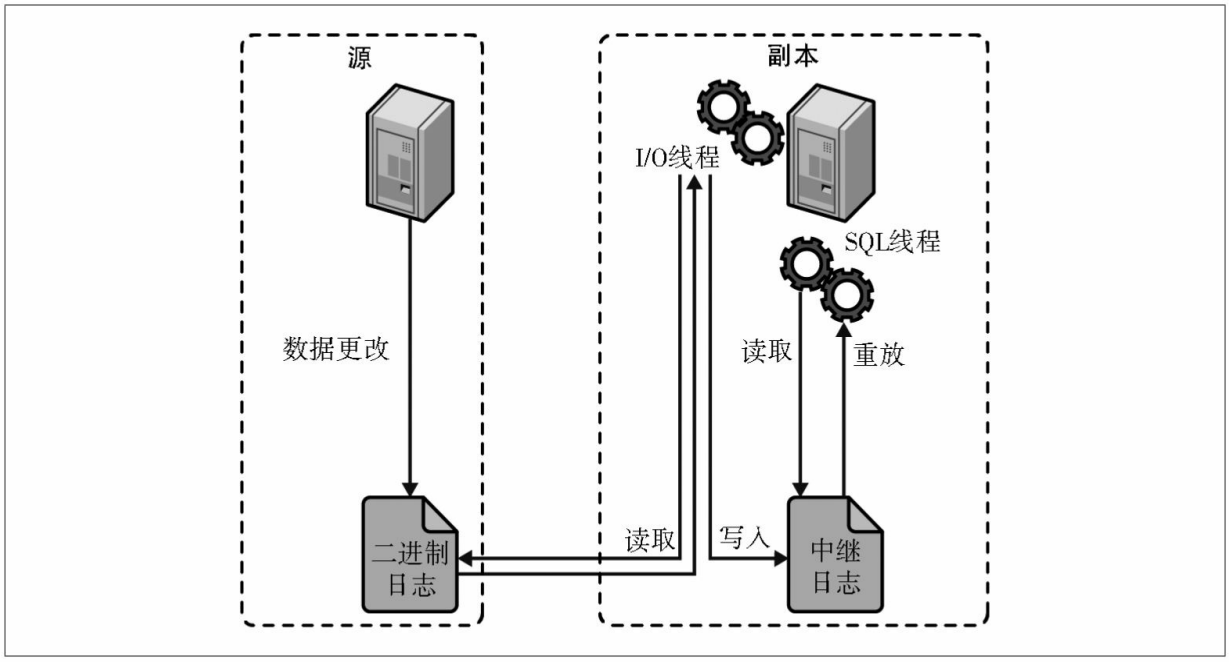

图9-1：MySQL的复制是如何工作的

在复制架构中，读取和重放日志事件是解耦的，这就允许读取日志和重放日志异步进行，也就是说，这里的I/O线程和SQL线程都是可以独立运行的。

复制原理

我们已经介绍了复制的一些基本概念，接下来要更深入地了解复制。让我们看看复制究竟是如何工作的，有哪些优点和弱点，最后介绍一些更高级的复制配置选项。

选择复制格式

MySQL提供了三种不同的二进制日志格式用于复制：基于语句的、基于行的和混合模式。可以通过系统参数binlog_format控制日志写入时使用哪种日志格式。

基于语句的复制是通过记录所有在源端执行的数据变更语句来实现的。当副本从中继日志读取到事件并执行时，实际上是重新执行在源端执行过的SQL语句。这种格式的主要优点是简单且紧凑。一条更新了大量数据的SQL语句，在二进制日志中可能仅仅需要几十字节存储。其最大的弊端则在于会遇到某些具有“不确定性”的SQL语句问题。假设有一条语句删除了一张有1000行记录的表中的100行，但没有用ORDER BY子句。如果在源和副本上，记录的排序不同，这条SQL语句在源和副本上删除的100条记录就会不同，这将导致数据不一致。

基于行的复制将事件写入二进制日志，该事件包含了该行记录发生了什么改变。这听起来很简单，但是，因为这种模式的确定性，相比基于语句的方式来说，其实发生了很大的变化。使用基于行的复制，通过查看二进制日志中的事件，可以看到究竟是哪一行记录发生了什么样的改变。在基于语句的复制模式下，SQL语句在执行时被解析，服务器在执行时找到所有需要变更的记录进行操作。而在基于行的复制模式下，每条被改变的记录都会作为事件被写入二进制日志，这可能会让二进制日志的大小发生巨大的增长。

“混合模式”（the mixed method）试图结合以上两种格式的优点。在这种模式下，事件的写入，默认使用基于语句的格式，仅在需要时才切换到基于行的格式。我们说“试图”是因为这种模式真的非常“努力”，在写入每个事件时会有很多的判断条件[1]，以确定使用哪种格式，而这也会导致二进制日志中出现不可预测的事件。我们认为二进制日志数据应该是两种模式选其一，而不应该是两者的混合。

我们建议坚持使用基于行的复制，除非某些场景下明确需要临时使用基于语句的复制。基于行的复制提供了最安全的数据复制方法。

全局事务标识符

在MySQL 5.6之前，副本必须跟踪连接到源时读取的二进制日志文件和日志位置。例如，一个副本连接到上游源并从文件binlog.000002的2749位置读取数据。当副本从该二进制日志中读取事件时，它每次都会向后推进日志位点。如果就在这时，故障发生了！比如源数据库崩溃了，必须从备份中重建数据。那么问题来了：在源端，如果二进制日志位点重新开始，怎么能重新将副本连接到源库？确定从哪个位点开始连接是一个非常复杂的过程。如果指向的位点太早，那么副本上就会获得重复的事件，如果指向的位点太晚，则会漏掉事件。无论使用哪种方式，都难以正确地将副本连接到源库。

为了解决这个问题，MySQL新增了另一种跟踪复制位点的方法：全局事务标识符

（GTID）。使用GTID，源服务器提交的每个事务都被分配一个唯一标识符。此标识符是由server_uuid[2]和一个递增的事务编号组成的。当事务被写入二进制日志时，GTID也随之被写入。回顾本章前面的内容可以了解到，副本将从源库读取的二进制日志事件先写入本地中继日志，再使用SQL线程执行该事务，将变更应用到本地副本上。当SQL线程提交事务时，它也会将GTID标记为执行完成。

为了更好地说明这一点，我们来看一个例子。假设我们的源服务器刚刚完成配置，里面还没有数据——甚至没有创建数据库。在这个源服务器上，也会生成server_uuid，假设为b9acac5a-7bbe-11eb-a043-42010af8001a。我们对副本也做同样的配置，并使用适当的命令来搭建到源服务器的复制。

在源服务器上，我们需要创建一个新数据库：

```
CREATE DATABASE misc;
```

这个事件将被写入二进制日志，这样副本也可以创建同样的数据库。在二进制日志中，我们将看到这个由GTID标识的单个事件：

```
b9acac5a-7bbe-11eb-a043-42010af8001a:1
```

当副本服务器应用此事件时，它会记录已经完成了该事务b9acac5a-7bbe-11eb-a043-42010af8001a：1。

在我们设计的示例中，假设此时在副本上停止MySQL的执行。它已经提交一个事务，如果源库继续写入新的事务，那么事务列表将继续增长：2、3、4、5等。当再次启动副本服务器[3]的时候，它知道已经完成了事务1，可以开始处理事务2了。

GTID解决了运行MySQL复制的一个令人痛苦的问题：处理日志文件和位置。强烈建议始终按照MySQL官方文档中的说明，在数据库中启用GTID。

崩溃后的复制安全

虽然GTID解决了日志文件和位置问题，但还有一些其他问题困扰着MySQL的管理员。在本章后面，我们将讨论一些常见的故障场景；但是，在此之前，有一些设置可以极大地改善使用复制的体验。

为了尽量降低复制中断的可能性，建议MySQL的部分参数按照如下讲解内容进行配置。

innodb_flush_log_at_trx_commit=1

> 严格来说，这并不是一个复制相关的配置。不过，这个参数可以保障每个事务日志都被同步地写到磁盘。这是一个符合ACID要求的配置，将最大限度地保护你的数据——即使是在复制中也是这样。这是因为二进制日志事件首先被提交，然后事务将被提交并写入磁盘。将此参数设置为1将增加磁盘写入操作的频次，同时确保数据的持久性。

sync_binlog=1

> 该变量控制MySQL将二进制日志数据同步到磁盘的频率。将此值设置为1意味着在每次事务执行的时候都会把二进制日志同步写入磁盘。这可以防止在服务器崩溃时丢失事务。就像之前的配置参数一样，它也会增加磁盘写入量。

relay_log_info_repository=TABLE

以前，MySQL的复制通常依赖磁盘上的文件来跟踪复制位置。这意味着，复制完成事务操作之后，还需要完成同步写入磁盘操作。如果在事务提交和同步之间发生了服务器崩溃，此时，磁盘上的文件将可能包含错误的文件和位置信息。在该配置下，该信息将被转移到MySQL本身的InnoDB表中，允许复制更新同一事务中的事务和中继日志信息。这会在一个原子操作中完成，并有助于崩溃恢复。

relay_log_recovery=ON

简单地说，该参数使得副本服务器在检测到崩溃时会丢弃所有本地中继日志，并从源服务器中获取丢失的数据。这确保了在崩溃中发生的磁盘上的任何损坏或不完整的中继日志都是可恢复的。配置该参数后，不再需要配置sync_relay_log，因为在发生崩溃时，中继日志将被删除，也就无须花费额外的操作将它们同步到磁盘。

延迟复制

在某些场景下，在一个拓扑结构中，某些副本有一些延迟反而是有好处的。在这个策略下，可以让副本中的数据保持在线并且持续运行，但同时落后于源数据库数小时或者数天。延迟复制的配置语句是CHANGE REPLICATION SOURCE TO，配置选项为SOURCE_DELAY。

想象一下这样的场景，你正在处理大量数据，突然意外地做了一些变更：删除了一个表。从备份中恢复可能需要几个小时。如果使用了延迟复制的副本，则可以找到DROP TABLE语句对应的GTID，使副本服务器的复制运行到表被删除之前的时间点，这会大大减少修复时间。

不过，这种复制结构也会带来一些复杂性。虽然延迟复制在某些数据丢失场景下非常有用，但它也给许多其他的操作带来了复杂性。如果你决定使用延迟复制，则需要考虑，在选择新的源服务器时，通常要排除延迟副本（如果故障恢复与切换操作是完全自动化的，这点则更加重要），如何监控延迟复制，以及如何处理这个特殊的副本等。这些都是引入延迟复制时应该考虑到的额外复杂性。

多线程复制

在复制技术中，历史非常悠久的挑战之一就是，虽然在源上数据可以并行写入，但在副本上只能是单线程的。最新的MySQL版本则提供了多线程复制能力（参见图9-2），可以在副本端运行多个SQL线程，从而加快本地中继日志的应用。

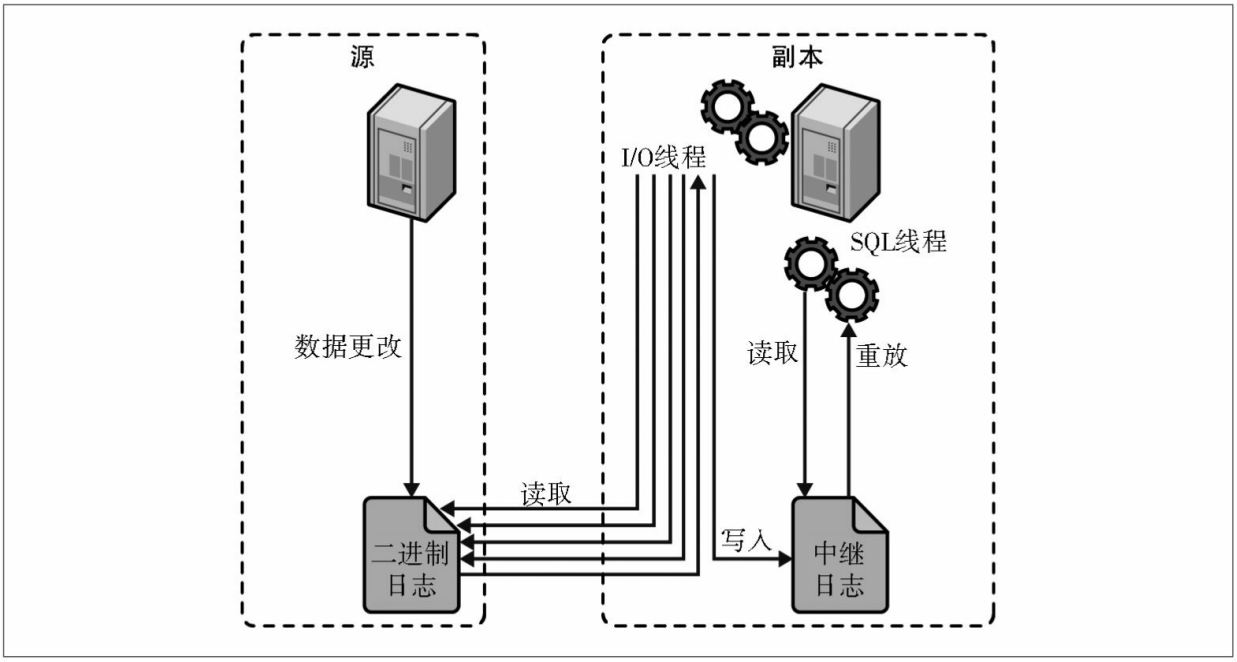

图9-2：多线程复制结构

多线程复制有两种模式：DATABASE和LOGICAL_CLOCK。在DATABASE模式下，可以使用多线程更新不同的数据库；但不会有两个线程同时更新同一个数据库。如果将数据分布在MySQL的多个数据库中，则可以同时并且一致地更新它们，这种模式非常有效。另一个模式LOGICAL_CLOCK允许对同一个数据库进行并行更新，只要它们都是同一个二进制日志组提交的一部分。

什么是二进制日志的组提交

为了更好地解释这一点，这里将使用Morgan Tocker的一个类比（参见链接35）：一艘渡轮试图将乘客从A点运送到B点。

在MySQL 5.0中，渡轮将从A点接载一位乘客，然后将他转移到B点。A和B之间的行程大约需要10分钟，在此期间可能会有几名新乘客到达A点并排队等待，这并不重要。当渡轮回到A点时，它只会搭载排队中的下一位乘客。

在MySQL 5.6中，渡轮将在A点接载所有排队等待的乘客，然后将他们转移到B点。每次返回A点接载新乘客时，它会继续接载所有排队等待的乘客并将他们转移到B点。

在实际的情况中，许多乘客都会在等待渡轮返回A的过程中到达A，并且A和B之间的行程往往需要一些时间，这种方式性能明显会更好。当然，这在单线程运行的简单基准测试中，是无法测量的。

MySQL 5.7及更高版本的行为与5.6类似，因为它将从A点接载所有等候的乘客并将他们转移到B点，但新版本中有另一个显著的增强！

当渡轮返回A点以接载等候的乘客时，可以将其配置为等待更长的时间，因为知道可能会有新乘客到达。例如：如果你知道A点和B点之间的行程持续10分钟，为什么不在A点多等30秒再出发呢？这可以节省往返行程次数，并提高单次运输的乘客总数。

人工延迟的配置参数是binlog_group_commit_sync_delay（以微秒为单位的延迟）和binlog_group_commit_sync_no_delay_count（决定中止等待之前要等待的事务数）。

在这个例子中，乘客显然是事务，而轮渡是一个开销很大的fsync()操作。需要注意的是，只有一艘渡轮在运行（一组有序的二进制日志），因此控制这些行为的一些参数，是一些高级的选项。

在大多数情况下，可以简单地通过将replica_parallel_workers设置为非零值来开启该配置，并立即看到效果。如果在单个数据库上操作，还需要将replica_parallel_type更改为LOGICAL_CLOCK。由于多线程复制还需要使用协调线程，因此管理这些线程的状态，也会带来一些额外的开销。此外，确保你的副本配置了参数replica_preserve_commit_order，这样就不会出现无序提交的问题。请参阅官方文档中Gaps（“差距”）小节，那里详细解释了这个配置的重要性。

有两种方法可以确定最合适的replica_parallel_workers值。一种不那么精确的方式是先停止复制，然后测试使用不同数量的线程赶上最新位置所需的时间，直到找到最佳配置。这种方式有一些缺点，因为它假设复制总是发送同样数量的数据操作语句（DML），并且它们的性能都基本相同。在实践中，这个假设很难成立。

更精确的方法是，根据实际的工作负载情况，查看每个应用程序线程的繁忙程度，然后再确定并行度。为此，我们需要启用Performance Schema的插桩和消费者表，允许它收集一些信息，然后查看实际情况。

首先，我们需要启用这些视图：[4]

```
UPDATE performance_schema.setup_consumers SET ENABLED = 'YES'
WHERE NAME LIKE 'events_transactions%';

UPDATE performance_schema.setup_instruments SET ENABLED = 'YES', TIMED = 'YES'
WHERE NAME = 'transaction';
```

允许复制在一段时间内处理二进制事件。在理想情况下，应该在写入负载最大的时候或其他观察到出现复制延迟的时候：

```
mysql> USE performance_schema;
events_transactions_summary_by_thread_by_event_name.thread_id AS THREAD_ID,
events_transactions_summary_by_thread_by_event_name.count_star AS COUNT_STAR
FROM events_transactions_summary_by_thread_by_event_name
WHERE
events_transactions_summary_by_thread_by_event_name.thread_id IN (SELECT
replication_applier_status_by_worker.thread_id
FROM replication_applier_status_by_worker);
+-----------+------------+
| THREAD_ID | COUNT_STAR |
+-----------+------------+
|   1692957 |      23413 |
|   1692958 |       7150 |
|   1692959 |       1568 |
|   1692960 |        291 |
|   1692961 |         46 |
|   1692962 |          9 |
+-----------+------------+
6 rows in set (0.00 sec)
```

这条查询将帮助你确定每个线程处理了多少个事务。正如我们从这个示例工作负载的结果中看到的，最佳配置是在3到4个线程之间，超出此范围的线程则很少被用到。

半同步复制

在启用半同步复制后，源在完成每个事务提交时，都需要确保事务至少被一个副本所接收。[5]需要确认副本已收到并成功将其写入自己的中继日志（但不一定应用到本地数据）。

由于每个事务都必须等待其他节点的响应，因此该功能会给服务器执行的每个事务都增加额外的延迟。需要根据实际情况考虑是否开启该选项。

一个值得注意的重要事情是，如果在一定时间范围内没有副本确认事务，MySQL将恢复到标准的异步复制模式。这时事务并不会失败。这也说明，半同步复制不是一种防止数据丢失的方法，而是可以让你拥有更具弹性的故障切换的更大工具集的一部分。

考虑到异步复制的回退机制，我们很难找到一个好的用例来解释为什么要启用它。在一个比较合理的场景中，发生网络分区异常后，在某个独立分区内的源服务器中就不会再写入新的数据。不幸的是，在超时后，该源将恢复为异步模式，并继续接受写入。因此，建议不要依赖该功能来保证数据完整性。

复制过滤器

复制过滤选项可以让副本仅复制一部分数据，不过这个功能并没有想象中那么实用。有两种复制过滤器：一种是从源上的二进制日志中过滤事件，另一种是从副本上的中继日志中过滤事件。图9-3展示了这两种类型。

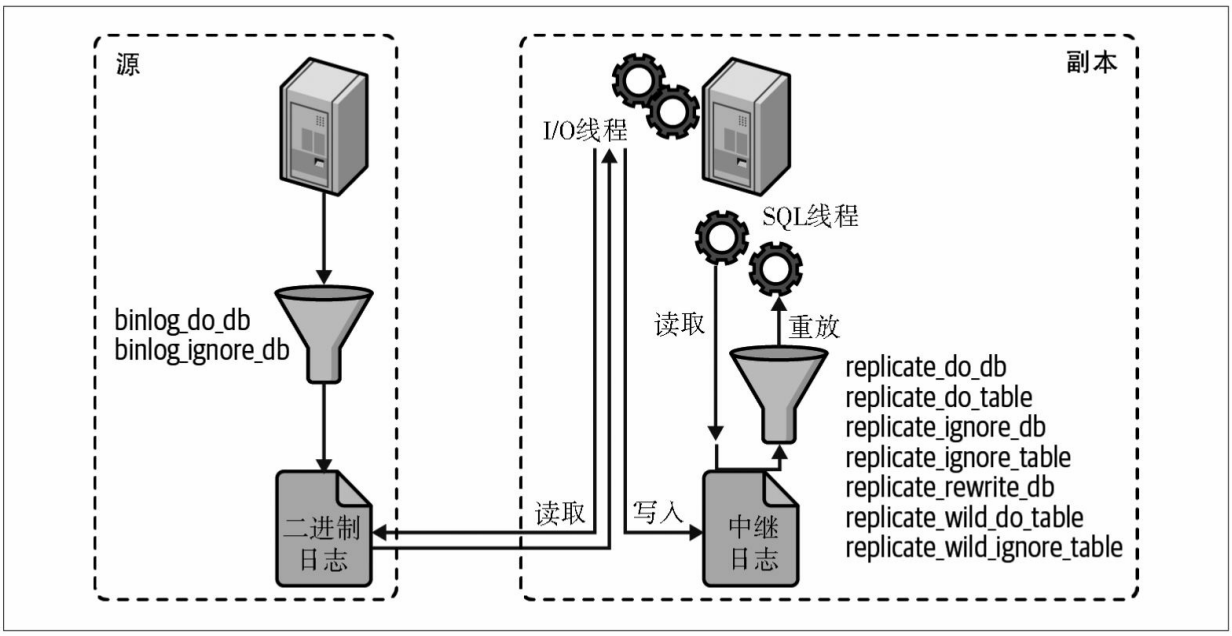

图9-3：复制过滤选项

控制二进制日志过滤的选项是binlog_do_db和binlog_ignore_db。事实上，并不建议启用这些参数，否则，你可能会经常要向老板解释为什么数据会永久消失并且无法恢复。

在副本上，replication_*选项在SQL线程从中继日志中读取事件时过滤事件。你可以复制或忽略一个或多个数据库，将一个数据库重写为另一个数据库，也可以基于LIKE语法的模式匹配来忽略表。

了解这些选项最重要的一点是，源和副本上的*_do_db和*_ignore_db选项并不像你预期的那样工作。你可能会认为它们会根据对象的数据库名称来进行过滤，但实际上它们是根据当前的默认数据库进行过滤的，也就是说，如果在源上执行以下语句：

```
USE test;
DELETE FROM sakila.film;
```

*_do_db和*_ignore_db参数将在test上过滤DELETE语句，而不是在sakila上。这并不是你想要的，这很可能会导致错误的语句被复制或者被忽略。*_do_db和*_ignore_db参数有用，但是应用场景很有限，使用时应该非常小心。如果使用这些参数，复制很容易出现不同步或失败。

binlog_do_db和binlog_ignore_db选项不仅有可能破坏复制，还会使从备份中进行时间点恢复变得不可能。在大多数情况下都不应该使用它们。

通常，复制过滤器是一颗定时炸弹。例如，假设你想阻止权限更改传递到副本，这是一个相当常见的场景。（如果你打算这么做，可能要想想是不是有哪里做错了什么；应该有更好的方法可以实现你的真正目标）。系统表中的复制过滤器肯定会阻止GRANT语句复制，但它们也会阻止调度事件、函数、存储过程等对象的复制。因为这种种不可预见的后果，所以建议谨慎使用复制过滤功能。如果需要防止特定语句被复制，一个推荐的做法是使用SET SQL_LOG_BIN=0，尽管这种做法也有一定的危险性。一般来说，应该非常小心地使用复制过滤器，并且只在真正需要它们时才使用，因为它们很容易破坏复制，并会在

最不方便的时候出现问题，例如在灾难恢复期间。

话虽如此，在某些特定情况下复制过滤器也是有益的。例如，你创建了多个数据库users_1、users_2、users_3和users_4，现在，服务器上的性能压力非常大。通过恢复备份并配置好复制，可以准备将users_3和users_4的查询移动到另一台服务器上。这通常都没什么问题，但你的新数据库上还有users_1和users_2。在某些时候，为了降低性能影响，不希望复制不需要的数据，那么可以考虑这个方案。先恢复备份，然后删除users_1和users_2。再配置复制规则以忽略users_1和users_2并完成复制设置。现在，在新服务器上仅需处理users_3和users_4的事件。一旦赶上了最新的复制位点，副本服务器就可以接收生产流量了。

过滤选项在MySQL手册中有非常详细的说明，这里不再重复那些细节。

复制切换

在本章的开头，我们提到复制是高可用性的基础。相比备份，总是保留一份持续更新的副本数据，会让灾难恢复更简单。除此之外，有些维护操作也只需要简单的重启MySQL即可。

在本节，我们想讨论将副本切换为新的源节点的正确方法。这个操作很容易出错，而且出错后会导致数据问题和延长停机时间。我们想澄清，“切换副本”（promoting a replica）和“故障切换”（failing over）是同义词，它们都意味着源服务器不再接收写入，并将副本提升为新的源服务器。

MySQL官方文档中的“Switching Sources During Failover”一节对如何处理这一问题进行了更详细的解释，但考虑到这一系列操作的重要性，本书将依旧从某些角度介绍这个问题。

计划内切换

切换的最常见原因是某些维护事件，包括安全补丁、内核更新，有时候甚至只是重新启动一下MySQL，因为有一些配置选项更改后需要重新启动才能生效。这种类型的切换被称为计划内切换。

要成功地执行此类切换，需要完成以下步骤：

1. 确定将哪个副本切换为新的源。这通常是一个包含所有数据的副本。这就是要操作的目标。
2. 检查延时，确保延时在秒级别。
3. 通过设置super_read_only停止数据写入源服务器。[6]
4. 等待副本与目标完全同步。可以通过比较GTID来确定这一点。
5. 在目标（需要切换为源的副本）上取消read_only设置。
6. 将应用流量切换到目标上。
7. 将所有副本重新指向新的源，包括刚刚被降级为副本的服务器。如果配置了GTID和AUTO_POSITION=1，这很简单。

计划外切换

只要时间够长，每个系统都会因软件或硬件而出现故障。当故障发生在承担写入的源服务器上时，会对用户体验产生重大影响。大多数应用程序这时候会返回一个错误，并让用户自己重试。这种情况下就需要计划外切换。

因为这时候不再存在一个实时运行的源服务器了，因此这将是一个很简短的计划外切换，需要根据副本上的已有数据进行选择：

1. 确定需要切换的副本。通常会选择数据最完整的副本。这就是要切换的目标。
2. 在目标上关闭read_only设置。
3. 将应用流量切换到目标上。

4. 将所有副本重新指向新源（目标服务器），包括恢复后的原来提供服务的源服务器。在使用了GTID之后，这些操作都很简单。

还需要注意，切换前的源服务器再次启动时，需要默认启用super_read_only。这将有助于防止任何意外的写入。

切换时的权衡

需要说明的是，大多时候，大家对故障的第一反应就是切换。但因为很难知道切换后的目标（新的源服务器）可能丢失了多少数据，所以有时不进行故障切换可能是更好的策略。通常，计划外切换并不是一个非常熟练的操作——也就是说，你并不会经常执行该操作。当需要执行时通常要查阅文档以确保不会错过任何步骤。在执行过程中还必须检查其他副本以验证哪个是最优的候选者。所有这些都需要时间。在某些情况下，等待服务器或MySQL进程重新恢复可能会更快。另外，等待恢复的好处是，如果你按照第5章中的ACID合规步骤进行操作，就不会丢失任何数据，并且所有副本将继续从中断的地方工作。

复制拓扑

几乎任何一个源和副本都可以配置MySQL复制。复制可以有许多复杂的拓扑结构，但即使是简单的拓扑也可以非常灵活。一种拓扑可以有许多不同的用途。如果描述详尽，关于使用复制拓扑就能轻易写一整本书。

巨大的灵活性也意味着可能会设计出难以维护的拓扑。在能够满足需求的情况下，我们强烈建议尽可能保持复制拓扑结构简单。话虽如此，我们推荐了两种可能的策略，它们几乎涵盖了所有用例。在实际中，你可能有充分的理由偏离这些，但是当你的设计变得更复杂的时候，一定要问自己是否还在解决正确的问题，有没有过度设计。

主动/被动模式

在主动/被动拓扑中，应用将所有读取和写入都指向单个源服务器。此外，还维护了少量不主动服务于任何应用程序流量的被动副本。选择此模型的主要原因是，不用担心复制延迟的问题。由于所有读取都指向源服务器，因此可以防止应用程序不接受读取延迟数据的问题。

图9-4展示了多个副本的配置。

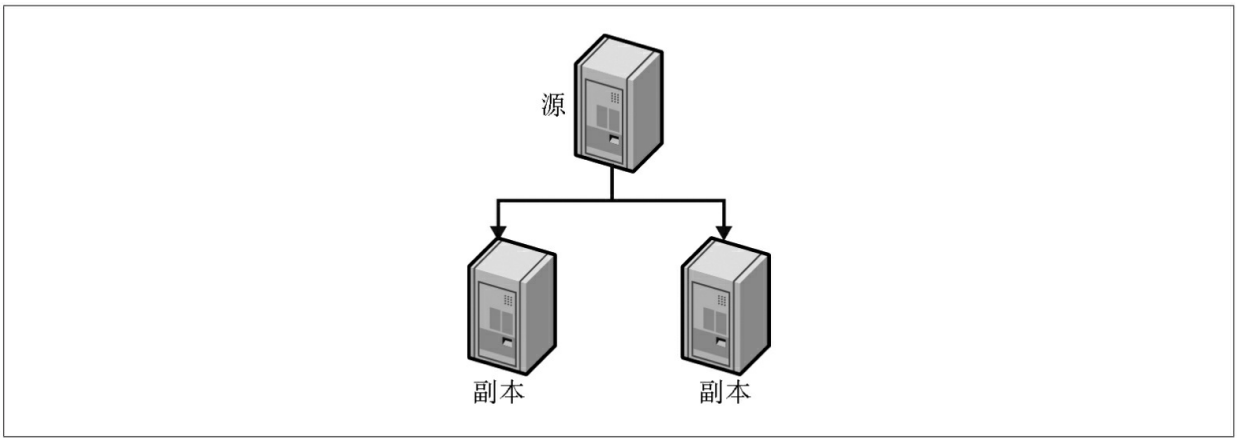

图9-4：一个源和多个副本

配置

在这个拓扑架构下，应该尽量让源和副本在CPU、内存等方面具有相同的配置。当需要进行切换的时候，例如，例行维护、软件升级、打补丁或者硬件故障等，可以从当前正在运行的源切换到其中某个副本上。因为副本上使用相同的硬件和软件配置，因此就可以确保能够支持切换之前的应用流量容量和吞吐量。

冗余

在物理硬件环境中，推荐使用至少三台服务器的*n*+2冗余。如果其中一台服务器发生硬件故障，还有一台额外的副本服务器用于故障切换。如果无法在源服务器上进行备份操作，可以使用其中一个副本作为备份服务器。

在云环境中，如果数据足够少，或者可以轻松复制数据，也可以使用至少两台服务器的

n+1冗余，否则还是建议*n*+2。如果使用了*n*+1架构，云服务商的动态资源调配特性会让管理操作更容易。对于打补丁等维护事件，可以按需配置第三个副本，对其执行任何必要的操作（如升级内核或应用安全更新），然后替换另一个副本。最后再进行故障切换并在前一个源上重复该过程。目的是让副本随时可以成为故障切换的目标。

在任何一种情况下，都可以将其中一个副本放置在地理位置较远的位置，不过需要关注复制延迟情况并确保其可用。副本应该是可恢复的，并且任何数据丢失都应符合预期的标准。我们还会在第10章的“定义恢复需求”一节中讨论这个问题。

注意事项

在此模式下，实际上是将读扩展绑定到单台服务器的容量上。如果达到读扩展上限，则必须演进到更复杂的拓扑（比如主动/只读池配置），否则就不得不利用分片来减少源上的读取压力。

主动/只读池模式

在主动/只读池模式中，你将所有写入指向源服务器。根据应用程序的需要，读取则可以被发送到源服务器或只读池。只读池可以实现读取密集型应用程序的读水平扩展。在某些时候，由于源上的复制请求，水平扩展能力会受到限制。

图9-5展示了单个源和副本池的架构。

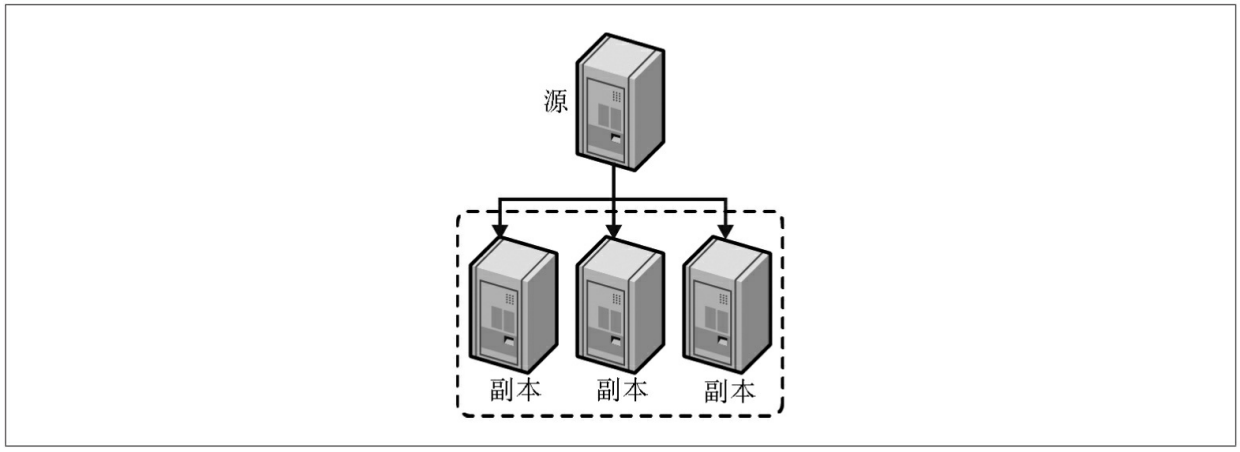

图9-5：一个源服务器和对应的一组只读池

配置

在理想情况下，至少要有一个副本（最好是两个）与源服务器具有相同的配置。同样地，当需要故障切换到其中一个副本时，该副本应该有足够的容量支撑业务的流量。

如果随着时间的推移，只读池在持续增长，则可以让副本用不同的硬件配置来优化成本。在这种情况下，请尝试将流量进行加权，运行在更好的硬件配置的副本上可以承担更多的流量。如果故障切换的目标副本上有32个CPU核，其他副本有8个核，则可以向32核的节点发送4倍以上的流量，以确保资源被充分利用。

冗余

在只读池中的服务器数量应满足先前提出的要求，还需要至少一台服务器可以充当故障切

换的目标。此外，还需要有足够的节点来支撑读流量，以及用于节点故障的小缓冲区。对于读取，最有效的使用率参考指标是CPU使用率，因此，目标池中每个节点的使用率应该在50%～60%。随着CPU使用率的增加，节点会花费更多的时间在工作和延迟之间进行上下文切换，尝试在满足应用程序期望的延迟和使用率之间找到适当的平衡。

注意事项

在使用读取池的时候，应用程序必须对延迟读取有一定的容忍度，因为你永远无法保证在源服务器上完成的写入已被复制到副本上。可能还需要一种方法来识别那些复制延迟太大的节点，并根据需要将其踢出只读池。

只读池的大小会决定管理的复杂度，以及何时应该考虑自动化。如果只读池有16个节点，那么每次更新内核或安装安全补丁就需要执行16遍，这时如果可以自动化地完成节点剔除、安装补丁、重新启动、重新加入节点池，那么会大大降低运维操作的工作量。

不推荐的一些拓扑架构

通过使用前面介绍的两种模式，可以让复制的拓扑结构简单易懂。在本书的早期版本中，介绍了很多其他的拓扑架构，或者你也可能听说过另一家公司的拓扑结构是如何设置的。我们在这里称其中一些为不推荐的架构，因为它们带来的风险和复杂性比预期的要大。

双源主动-主动架构

双源复制（也称为双向复制）涉及两台服务器，每台服务器都配置为源和另一台的副本——换句话说，一对协同源。图9-6展示了这种架构。

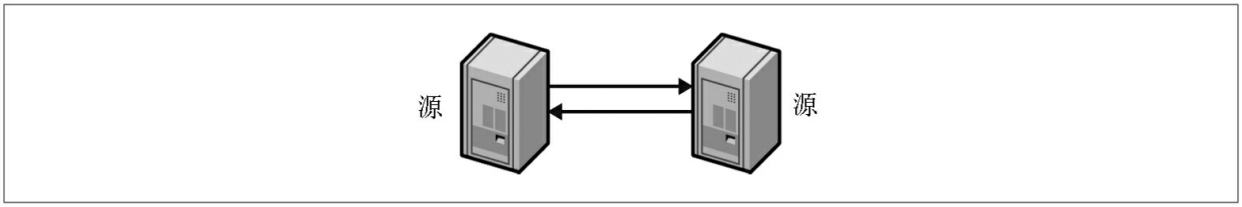

图9-6：主动-主动模式的双源服务器

乍一看，这与使用两台服务器的主动-被动模式没有什么不同，只是复制额外配置了相反的方向。这种主动-主动的架构，真正的危险会发生在明确向双方发送写入流量的时候。这时，在主动-主动模式下，很难正确地运行。一种策略是根据偶数/奇数等散列方法选择发送到哪一方。这确保了对同一行的写入和读取是一致的，但如果查询包含了另一侧记录的查询则可能出现不一致。例如，从一侧读取ID为1、3和5的行将始终保持一致。试想，如果要读取ID 1～6的查询呢？应该将该查询发送到哪里？如果由于复制延迟在另一侧存在而未反映在这一侧的更新该如何处理？

还有一个需要考虑的问题是容量。在这个方案中，每台服务器都是另一台服务器的副本，并且最有可能成为故障切换的目标。在进行容量规划的时候，需要确保当流量从一侧转移到另一侧时，CPU资源不会被耗尽。另外，故障切换后会引入完全不同的工作数据集。InnoDB的缓冲池也会受影响，这时候通常需要将缓存池中的部分数据移除，为新的热数据集腾出空间。

我们建议不要使用这种架构。通过让被动服务器处理流量而不是闲置，可能会让你觉得正在“使用”被动服务器。你最终会将数据不一致引入应用程序，并且始终处于没有足够容量

进行故障切换的边缘。一旦失去了故障切换的可能性，也就失去了灵活性。

双源主动-被动模式

主动-主动模式下的双源服务器有一个变种，可以避免我们刚刚讨论的陷阱。主要区别在于其中一台服务器是只读的“被动”服务器，如图9-7所示。

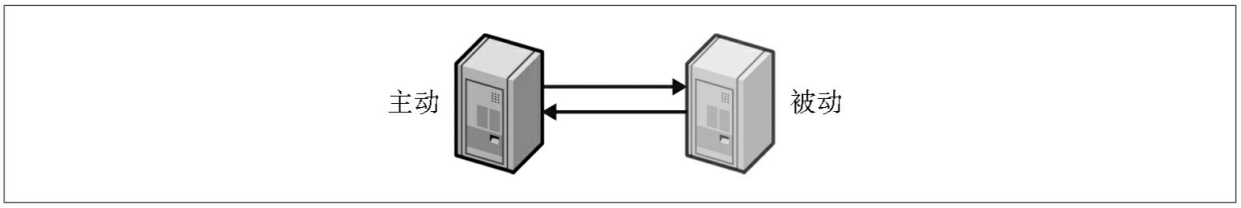

图9-7：主动-被动模式的双源服务器

从表面上看，这种配置似乎没有任何问题。它与我们建议的主动-被动模式的唯一不同之处在于，复制已预先配置回另一台服务器。这仅适用于两台服务器的配置。如果环境中运行了两台以上的服务器，则需要确定哪个节点是故障切换的最佳目标。预配置复制会将切换目标直接绑定到一个副本，在出现故障中断时无法提供灵活性。

我们坚持认为，之前也讨论过，配置复制是故障切换过程的一部分，是一个简单、可自动化的步骤。所以，这种模式并没有必要，只会引起混淆。

带有副本的双源模式

这是一种让事情变得更加复杂的模式，它可以为每个协同源添加一个或多个副本，如图9-8所示。

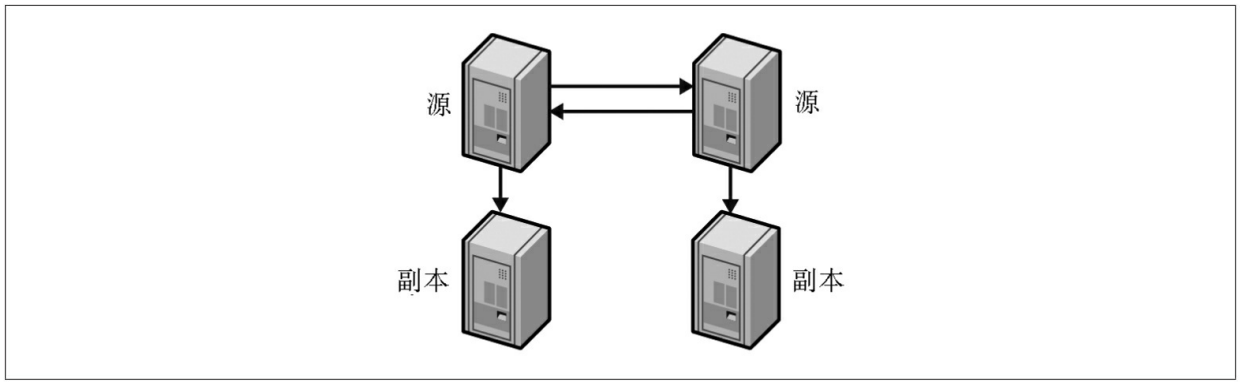

图9-8：带有额外副本的双源拓扑

这似乎解决了双源主动-主动模式的大部分问题，其中最重要的是如何进行流量路由。它解决了故障切换时有关容量规划和缓冲池减少的问题。在发生故障切换时，有其他步骤可以将其中一个源指向其上新切换的副本。

我们也坚决反对使用这种模式，主要是出于数据访问问题的考虑。两个可写的源只会带来麻烦。

环形复制

环形复制具有三个或更多个源，其中每台服务器都在环中，它之前的服务器是它的副本，它之后的服务器作为它的源，如图9-9所示。这种拓扑也称为循环复制。

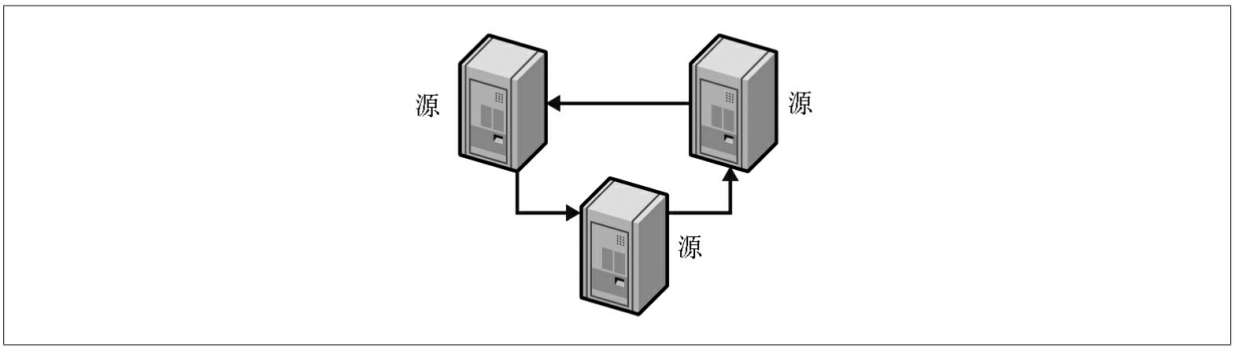

图9-9：环形复制架构

如果此拓扑中的任何服务器都不再提供服务，则拓扑结构就会被损坏，并且所有的数据更新都不会在环中流动。这种架构也有附加副本的变体，例如，将图9-9中的每个源都新增一个专用的副本。这仍然意味着环被破坏时，依旧要等到新的副本替换其位置后，环形复制才能重新恢复。

这种拓扑结构一点儿也不简单，也没有任何优势。

多源复制

虽然保持复制拓扑简单很重要，但在某些情况下，还是需要使用更高级的功能来处理一次性需求的。例如，你建立了一个全新的视频上传和观看网站，现在变得很流行。早期设计决策之一是将有关视频的数据和有关用户的数据分离到两个单独的数据库集群中。随着业务的成长，有一些数据查询需要将它们重新合并在一起。这时，可以通过多源复制来完成此操作，以将两个数据集聚集到一个副本中，如图9-10所示。

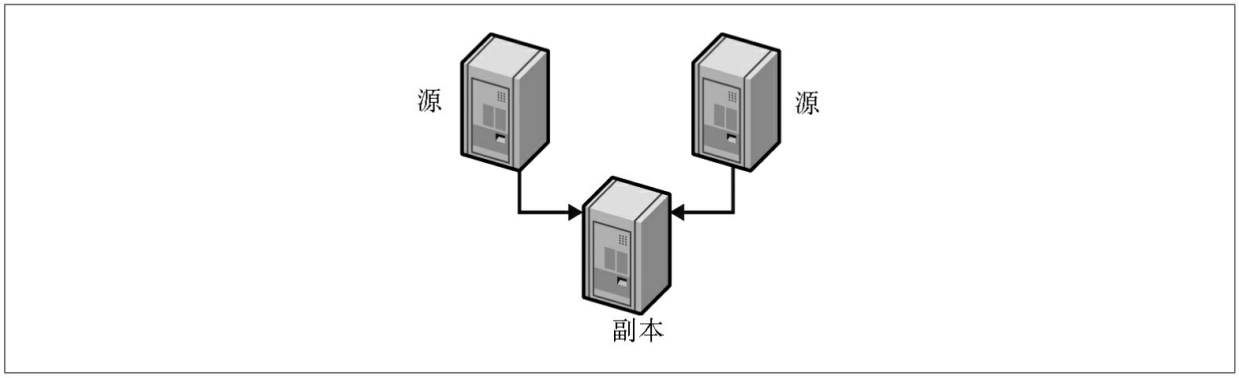

图9-10：多源复制

此功能建立在被称为复制通道的概念之上。在前面的示例中，需要使用第三个MySQL集群节点。这个新的第三个集群节点将创建两个复制通道：一个用于视频数据，一个用于用户数据。一旦加载并复制了数据，你可能需要一个非常短暂的停机时间，在此期间先停止对两个源服务器的写入，并推送应用代码以切换到新的组合数据库进行读写操作。至此，你已将两个数据库合并为一个了。

在继续下面的讲解之前，有一个重要的限制需要了解：不能将一个副本配置为对同一个源多次使用多源复制。

这种拓扑结构非常适合特殊使用的情况，我们非常不建议长期运行这种架构。暂时使用它

来合并数据是一个可接受的场景，最终目标是回到之前建议的两种架构之一。

复制管理和维护

在数据量很小而且写入负载一致的时候，通常不太需要经常查看复制延迟或者复制中断相关的问题。但随着数据量的增加，相关的管理和维护工作也会随之而来。

复制监控

复制增加了MySQL监控的复杂性。虽然复制实际上同时在源和副本上工作，但大部分工作都是在副本上完成的，这也是经常发生问题的地方。比如，所有的副本都在正常运行吗？有没有某个副本有错误？最慢的副本的复制延迟是多少？虽说是开箱即用，MySQL也提供了回答这些问题所需的大部分信息，但是自动化监控过程和如何让复制更加健壮，则需要你花费更多精力。

在配置复制监控时，需要注意以下几点。

复制同时需要源和副本上的磁盘空间

> 如图9-1所示，复制在源上使用二进制日志，在副本上使用中继日志。如果源上没有可用的磁盘空间，事务将无法完成并开始超时。如果在副本上发生相同的情况，MySQL会通过暂停复制并等待可用的磁盘空间，相比来说这更优雅一些。所以，两边的磁盘空间都需要进行监控，以确保复制稳定运行。

监控复制的状态和错误

> 尽管复制是一项长期存在的功能并且非常强大，但网络问题、数据不一致和数据损坏等外部因素可能会导致其中断。因此，监控复制线程是否正常运行是非常重要的，如果复制线程没有正常运行，那么可以查看报错信息，以确定下一步应该做什么来修复错误。在本章后面的“复制问题和解决方案”一节中，详细介绍了如何解决其中的一些问题。

延迟复制的实际延迟

> 之前提到了延迟复制，因此建议设置监控以确保延迟副本实际上延迟了正确的时间量。太长的延迟可能会使其使用起来更加耗时。如果延迟太短——或者更糟糕的是，根本没有延迟——可能在真的需要用到延迟副本时却无法使用。

观测复制延迟

通常，最需要监控的就是副本与源之间的复制延迟是多少。虽然SHOW REPLICA STATUS中的Seconds_behind_source列理论上显示了副本的延迟，但实际上并不总是准确的，原因有多种：

- 副本通过将服务器的当前时间戳与二进制日志事件中记录的时间戳进行比较来计算Seconds_behind_source，因此除非有正在处理的语句，否则副本甚至无法报告其延迟情况。
- 如果复制线程不是正在运行，副本通常会报告NULL。
- 一些错误（例如，源和副本之间的max_allowed_packet设置不匹配或网络不稳定）

可能会中断复制或停止复制线程，但Seconds_behind_source将报告0而不是错误。

- 即使复制过程正在运行，副本有时也无法计算延迟。如果发生这种情况，副本可能会报告0或NULL。
- 长事务可能会导致延迟时间的波动。例如，如果有一个更新数据的事务持续运行了1小时，然后才提交，则更新将在实际发生1小时后才写入二进制日志。副本处理语句时，会显示一小时的复制延迟，然后再突然跳回延迟零秒。

要解决这些问题，只能忽略Seconds_behind_source，并使用一些可以直接观察和测量的指标来监控复制延迟。最好的解决方案是心跳记录，它需要在源上每秒更新一次时间戳。需要计算延迟时，就可以简单地在副本上用当前时间戳减去记录的心跳时间。这种方式不受刚才提到的所有问题的影响，它还有一个额外的好处——创建了一个方便的时间戳，展示副本中的数据在什么时间点是最新的。Percona Toolkit中包含的pt-heartbeat脚本是当前最流行的复制心跳实现方案。

心跳记录还有其他好处。二进制日志中的复制心跳记录可用于许多目的，例如在灾难恢复时遇到的其他难以解决的问题。

刚刚提到的所有延迟指标都不能说明副本实际赶上源需要多长时间。这取决于许多因素，例如，副本配置是否强大以及源继续处理了多少写入。有关该主题的更多信息，请参阅本章后面“复制问题和解决方案”部分中的“复制延迟过大”小节。

确定副本数据的一致性

在完美世界中，副本始终是其源的精确复制减去其复制延迟的部分。但在现实世界中，各种情况都会导致副本与源不一致。一些可能的原因是：

- 意外写入副本。
- 使用双源复制，双方都写入了数据。
- 非确定性语句和基于语句的复制。
- 当运行在弱持久化模式时MySQL崩溃（参考第5章）。
- MySQL中的Bug。

建议遵循如下规则或配置。

在副本上，始终启用**super_read_only**

使用read_only可以防止没有SUPER权限的用户写入，但这不会阻止DBA在没有意识到他们在副本上的情况下运行DELETE或ALTER。super_read_only设置只允许复制写入，是运行副本的最安全方式。

使用基于行的复制或确定性语句

尽管存在使用大量磁盘空间的情况，但基于行的复制是复制数据的最一致的方式。这是因为它包括为每个条目更改的确切行数据。

考虑以下基于语句的复制：

```
DELETE FROM users WHERE last_login_date <= NOW() LIMIT 10;
```

当这个表中有1000个用户匹配WHERE子句时会发生什么？MySQL将使用表中的自然顺序，然后删除前10行。表的自然顺序在副本上可能不同，因此可能会影响不同的10行数据。后续运行的语句根据last_login_date修改或删除行可能存在也可能不存在。这可能会导致数据不一致。

编写此代码的最佳方法是使用ORDER BY，使行顺序具有确定性：

```
DELETE FROM users WHERE last_login_date <= NOW() ORDER BY user_id
LIMIT 10;
```

使用这个语句，只要源和副本中的数据一致，就会删除相同的10行。

不要尝试同时写入复制拓扑中的多台服务器

这包括使用在两侧都支持写入的协同源或环形复制。最实用的复制拓扑是使用一个源，执行所有写入操作，以及一个或多个副本，可选择地执行读取操作。

最后，我们强烈建议如果遇到任何复制错误，请使用MySQL官方文档中的策略（参见链接36）来重建副本。

复制问题和解决方案

MySQL复制的简单实现使其易于配置，但也意味着有很多方法可以停止、混淆或破坏它。在本章前面，我们讨论了崩溃安全复制和有助于保持源和副本同步的规则。本节讨论常见问题，如何发现以及如何解决甚至预防它们。

源端二进制日志损坏

如果源上的二进制日志已被损坏，那么别无选择，只能重建副本。跳过损坏的条目将跳过一些事务，这些事务在副本上将不再有机会被执行，也就会导致数据不一致。

非唯一的服务器**ID**

这是复制过程中可能遇到的难以捉摸的问题之一。如果你不小心配置了具有相同服务器ID的两个副本，不仔细观察的话，它们似乎可以正常工作。但如果使用诸如*innotop*之类的工具查看错误日志或源服务器，就会发现一些非常奇怪的情况。

在源服务器上，任何时候都只能看到两个副本中的一个。（通常，所有副本都已连接并一直在复制）。在副本上，可以在错误日志中看到频繁的断开连接和重新连接的错误消息，但没有提及与错误配置的服务器ID有关。

根据MySQL的版本不同，副本可能会正确但缓慢地复制，或者实际上无法正确复制——任何给定的副本都可能错过二进制日志事件甚至重复它们，从而导致重复键错误（或静默数据损坏）。由于副本之间的冲突增加了负载，还可能在源上引起问题。而且，如果副本之间的竞争足够严重，错误日志文件可能会在很短的时间内变得巨大。

解决此问题的唯一方法是在设置副本时要小心。你可能会发现创建副本到服务器ID映射的规范列表很有帮助，这样就不会忘记哪个ID属于哪个副本。如果副本完全位于一个子网中，则可以使用每台机器IP地址的最后一个八位字节来作为唯一ID。

未配置服务器**ID**

如果没有配置服务器ID，MySQL将显示为使用CHANGE REPLICATION SOURCE TO配置了复制，但不会允许启动副本：

```
mysql> START REPLICA;
ERROR 1200 (HY000): The server is not configured as replica; fix in config file
or with CHANGE REPLICATION SOURCE TO
```

如果使用CHANGE REPLICATION SOURCE TO配置复制并通过SHOW REPLICA STATUS验证了配置，则此错误信息尤其令人困惑。你可能会从SELECT@@server_id中获得一个值，但这只是一个默认值。必须显式设置该值。

临时表丢失

临时表在某些场景中使用起来很方便，但不幸的是，它们与基于语句的复制不兼容。如果

副本崩溃或将其关闭，则副本线程正在使用的任何临时表都会消失。当重新启动副本时，引用丢失的临时表的任何相关语句都将失败。

这种情况下最好的方法是使用基于行的复制。其次是统一命名临时表（例如，以temporary_为前缀），然后使用复制规则完全跳过复制临时表。

没有复制所有变更

如果误用SET SQL_LOG_BIN=0或不了解复制过滤规则，可能会导致副本不会执行源上发生的某些变更。有时希望将其用于存档，但这通常是偶然的并且会产生不良后果。

例如，假设有一条replicate_do_db规则仅将sakila数据库复制到某个副本中。如果在源上执行以下命令，副本的数据将与源上的数据不同：

```
mysql> USE test;
mysql> UPDATE sakila.actor ...
```

某些类型的语句甚至可能导致复制失败并出现错误，但这些错误却与复制没有关系。

复制延迟过大

复制延迟是一个常见问题。无论如何，在设计应用程序的时候，都需要考虑复制可能会出现一些延迟。以下是一些减少复制延迟的常用方法。

多线程复制

确保使用了多线程复制，并且已经按照手册查看了如何调整各种选项以从中获得最佳效率。

使用分片技术

这看起来是一种逃避问题的方式，但使用分片技术将写入分散到多个源也是一种非常有效的策略。MySQL长期存在的经验法则是：使用副本扩展读取操作，使用分片技术扩展写入操作。

临时降低持久化要求

完美主义者不同意这种做法，但有时可能已经用尽了所有的调优技术，并且分片也不可行。如果复制延迟主要是由于写入操作导致的，则可以临时设置sync_binlog=0和innodb_flush_log_at_trx_commit=0以提高复制速度。

如果采用最后一种方法，那么要非常非常小心。最好只在副本上执行此操作，如果此副本也用于执行备份操作，则更改这些设置可能会使你无法从备份中恢复完整的数据。[7]此外，如果副本在降低持久化要求期间崩溃，那么必须从源服务器获取数据并重新配置副本。最后，如果是手动执行此操作，很容易忘记将持久化配置改回去。请确保有良好的监控或编写了某种脚本来再次设置持久化参数。

一种可能的策略是监控SHOW REPLICA STATUS命令中的Seconds_behind_source值，当它超过某个阈值时，触发执行以下操作：

1. 验证是否启用了super_read_only，以确保服务器是不可写副本。
2. 更改sync_binlog和innodb_flush_log_at_trx_commit的配置以减少写操作。

3. 定期检查SHOW REPLICA STATUS以获取Seconds_behind_source的值。
4. 当延迟低于可接受的阈值时，将持久性相关参数恢复到正常值。

来自源服务器的超大数据包

当源服务器的max_allowed_packet大小与副本不匹配时，可能会出现另一个难以跟踪的复制问题。在这种情况下，源服务器可以记录副本认为过大的数据包，当副本检索该二进制日志事件时，它可能会遇到各种问题，其中包括无休止的错误循环、重试或中继日志中的损坏。

磁盘空间耗尽

复制确实可以用二进制日志、中继日志或临时文件占满磁盘，尤其是当源服务器执行大量LOAD DATA INFILE语句并在副本上启用log_replica_updates时。复制延迟越大，用于已从源端获取但尚未执行的中继日志占用的磁盘空间就越多。可以通过监控磁盘使用情况并设置relay_log_space参数来防止这些错误出现。

复制的限制

因为MySQL自身固有的一些限制，无论有没有出现明确的报错，MySQL复制都可能失败或不同步。有非常多的SQL函数和编程实践都无法可靠地被复制（我们在本章中也提到了很多案例）。通常，很难确保在生产代码中没有使用这些限制的内容，特别是当应用程序或团队规模很大的情况下。

另一个问题是服务器中的bug。并不是我们很消极，但MySQL服务器的许多大版本历史上都有复制方面的bug，尤其是在大版本的第一个版本中。诸如存储过程等新特性通常会导致更多问题。

对于大多数用户来说，这不是回避新功能的理由，而应该是仔细测试的理由，尤其是在升级应用程序或MySQL时。监控也很重要，你需要知道什么时候会导致问题。

MySQL复制已经很复杂了，所以应用程序越复杂，就越需要小心。但是，如果你学会了如何使用它，它就能很好地工作。

小结

MySQL复制是其内置的一把“瑞士军刀”，极大地增加了MySQL的功能范围和实用性。事实上，这可能是MySQL如此迅速流行的主要原因之一。

尽管复制有许多限制和需要注意的事项，但事实证明，其中大多数相对来说都不重要或容易避免。有些缺点只是高级功能的特殊行为，大多数人不会用到，但对少数需要它们的用户来说非常有用。

在复制方面，你的座右铭应该是“保持简单”。除非确实需要，否则不要做任何花哨的事情，例如，使用环形复制或复制过滤器。应该简单地使用复制来镜像数据的整个副本，包括所有权限。在各个方面保持副本与源完全相同将有助于避免很多问题。

---

[1] 正如预期的那样，推荐查看官方手册，以确保了解最新版本中混合模式是怎样处理不同类型**SQL**语句的。

[2] 注意，**server_uuid**与名称类似的**server_id**是完全不同的。**server_id**参数是由用户指定的一个服务器配置值，如果**MySQL**在启动时，没有检测到***auto.cnf***文件，则会自动生成它的**server_uuid**。

[3] 这里假设在执行**CHANGE REPLICATION SOURCE TO**命令时，配置了**SOURCE_AUTO_POSITION=1**选项，这是推荐的做法。

[4] 开启**Performance Schema**的插桩和消费者表可以收集内部的额外数据，但也会带来一些额外的**CPU**消耗。需要注意，在生产环境开启之前，需要充分测试这些消耗对于生产负载的影响程度。

[5] 所需的完成确认的副本数量是一个可配置的选项（**rpl_semi_sync_source_wait_for_replica_count**）。在一个更大的集群中，可以考虑在提交事务之前有两个甚至三个副本完成确认。

[6] 设置**super_read_only**时会隐式开启**read_only**。相反，关闭**read_only**会隐式关闭**super_read_only**。在此过程中，没有必要同时启用或关闭这两个变量。

[7] 这通常适用于使用**LVM**快照或基于云的磁盘快照方法进行备份的情况。

第10章 备份与恢复

如果你没有提前做好备份规划，也许以后会发现已经错失了一些最佳的选择。例如，在服务器已经配置好以后，才想起应该使用LVM，以便获取文件系统的快照——但这时已经太迟了。在为备份配置系统参数时，可能没有注意到某些系统配置对性能有着重要影响。如果你没有计划做定期的恢复演练，当真的需要恢复时，就会发现并没有那么顺利。

在本章中，我们不会涵盖备份和恢复解决方案的所有部分，而是仅涉及与MySQL相关的部分。以下是我们决定不在此处包含的一些要点，但在你的整体备份和恢复策略中仍旧应该考虑涵盖：

- 安全（访问备份的入口，恢复数据的权限，文件是否需要加密）。
- 备份存储在哪里，包括它们应该离源数据多远（在一块不同的磁盘上、一台不同的服务器上，或离线存储），以及如何将数据从源头移动到目的地。
- 保留策略、审计、法律要求，以及相关的条款。
- 存储解决方案和介质，压缩，以及增量备份。
- 存储的格式。
- 对备份的监控和报告。
- 存储层内置备份功能，或者其他专用设备，例如预制式文件服务器。

在开始本章的介绍之前，让我们先澄清几个核心术语。首先，经常可以听到所谓的热备份、暖备份和冷备份。人们经常使用这些词来表示一个备份的影响：例如，“热备份”不需要任何的服务停机时间。问题是对这些术语的理解因人而异。有些工具虽然在名字中使用了“热备份”，但实际上并不是我们所认为的那样。我们会尽量避开这些术语，而直接说明某个特别的技术或工具对服务器的影响。

另外两个让人困惑的词是还原和恢复。在本章中它们有其特定的含义。还原意味着从备份文件中获取数据，可以将这些文件加载到MySQL里，也可以将这些文件放置到MySQL期望的路径中。恢复一般意味着当某些异常发生后对一个系统或其部分的拯救。它包括从备份中还原数据，以及使服务器完全恢复功能的所有必要步骤，例如，重启MySQL、改变配置和预热服务器的缓存等。

在很多人的观念中，恢复仅意味着修复崩溃后损坏的表。这与恢复一个完整的服务器是不同的。存储引擎的崩溃恢复要求数据和日志文件一致。要确保数据文件中只包含已经提交的事务所做的修改，恢复操作会将日志中还没有应用到数据文件的事务重新执行。这也许是恢复过程的一部分，甚至是备份的一部分。然而，这和一个意外的DROP TABLE事故后要做的事是不一样的。根据你的恢复所需要解决的问题的不同，所要采取的操作可能也会有很大不同。

最后，有两种主要类型的备份：裸文件备份和逻辑备份。裸文件备份（有时也被称为物理[1]备份）是指文件系统中的文件副本。逻辑备份是指重建数据所需的SQL语句。

为什么要备份

下面是进行备份的几个非常重要的理由。

灾难恢复

> 灾难恢复是在下列场景下需要做的事情：硬件故障、一个不经意的Bug导致数据损坏，或者服务器及其数据由于某些原因不可获取或无法使用等。你需要准备好应付很多问题：某人偶然连错服务器执行了一个ALTER TABLE操作，机房大楼被烧毁，恶意的黑客攻击或MySQL的Bug等。尽管遭受任何一个特殊的灾难的概率都非常低，但所有的风险叠加在一起就很有可能会碰到了。

人们改变想法

> 不必惊讶，很多人经常会在删除某些数据后又想恢复这些数据。

审计

> 有时候需要知道数据或Schema在过去的某个时间点是什么样的。例如，你也许被卷入一场法律诉讼，或发现了应用的一个Bug，想知道这段代码之前干了什么（有时候，仅仅依靠代码的版本控制还不够）。

测试

> 一个最简单的基于实际数据来测试的方法是，定期用最新的生产环境数据更新测试服务器。如果使用备份的方案，那就非常简单了：只要把备份文件还原到测试服务器上即可。

重新审视你的一些假设。例如，你认为共享主机供应商会提供MySQL服务器的备份？许多主机供应商根本不备份MySQL服务器，另外一些也仅仅在服务器运行时复制文件，这可能会创建一个损坏的没有用的备份。

定义恢复需求

如果一切正常，那么永远也不需要考虑恢复。但是，一旦需要恢复，只有世界上最好的备份系统是没用的，还需要一个强大的恢复系统。

不幸的是，让备份系统平滑工作比构造良好的恢复过程和工具更容易。原因如下：

- 备份在先。只有已经做了备份才可能恢复，因此在构建系统时，注意力自然会集中在备份上。
- 备份由脚本和任务自动完成。经常不经意地，我们会花些时间调优备份过程。花5分钟来对备份过程做小的调整看起来并不重要，但是你是否天天同样地重视恢复呢？
- 备份是日常任务，但恢复常常发生在危急情形下。
- 因为安全的需要。如果你正在做异地备份，可能需要对备份数据进行加密，或采取其他措施来进行保护。数据被盗用的危害人人皆知，但是有没有人想过，当没有人能对用来恢复数据的加密卷解锁，或需要从一个整块的加密文件中抽取单个文件时，数据被盗用的危害又会有多大？
- 只有一个人来规划、设计和实施备份。当灾难袭来时，那个人可能不在。因此需要培养几个人并有计划地让他们互为备份，这样就无须由一个不合格的人来恢复数据。

在规划备份和恢复的策略时，有两个重要的需求可以帮助思考：恢复点目标（PRO）和恢复时间目标（RTO）。你可能会注意到，这与第2章讨论的SLO非常类似。它们定义了可以容忍丢失多少数据，以及需要等待多久能将数据恢复。在定义RPO和RTO时，先尝试回答下面几类问题：

- 在不导致严重后果的情况下，可以容忍丢失多少数据？需要随时从故障中恢复，还是可以接受自从上次日常备份后所有的工作全部丢失？是否有法律法规的要求？
- 恢复需要在多长时间内完成？哪种类型的宕机是可接受的？哪种影响（例如，部分服务不可用）是应用和用户可以接受的？当那些场景出现时，又该如何持续服务？
- 需要恢复什么？常见的需求是恢复整个服务器、单个数据库、单个表，或仅仅是特定的事务或语句。

建议将上面这些问题的答案明确地用文档记录下来，同时还应该明确备份策略，以及备份过程。

> 备份误区**1**：复制就是备份
>
> 这是我们经常碰到的一个误区。复制不是备份，当然使用RAID阵列也不是备份。为什么这么说？可以考虑一下，如果意外地在生产库上执行了DROP DATABASE，它们是否可以帮你恢复所有的数据？RAID和复制连这个简单的测试都没法通过。它们不是备份，也不是备份的替代品。只有备份才能满足备份的要求。

设计**MySQL**备份方案

备份MySQL比看起来要难。究其根本，备份仅是数据的一个副本，但是受限于应用程序的要求、MySQL的存储引擎架构，以及系统配置等因素，复制一份数据变得很困难。

在深入所有选项细节之前，先来看一下我们的建议：

- 在生产实践中，对于大数据库来说，裸文件备份是必需的：逻辑备份太慢并受到资源限制，从逻辑备份中恢复需要很长时间。基于快照的备份，例如Percona XtraBackup和MySQL Enterprise Backup，是最好的选择。对于较小的数据库，逻辑备份可以很好地胜任。
- 保留多个备份集。
- 定期从逻辑备份（或者裸文件备份）中抽取数据进行恢复测试。
- 保存二进制日志用于基于故障时间点的恢复。应该将expire_logs_days参数的值设置得足够大，至少确保可以从最近两次裸文件备份中做基于时间点的恢复，这样就可以在保持源运行且不应用任何二进制日志的情况下创建一个副本。要使备份二进制日志独立于过期设置，二进制日志需要保存在备份中足够长的时间，以便能从最近的逻辑备份中进行恢复。
- 完全不借助备份工具本身来监控备份和备份的过程。需要额外验证备份是否正常。
- 通过演练整个恢复过程来测试备份和恢复。测算恢复所需要的资源（CPU、磁盘空间、实际时间，以及网络带宽等）。
- 要仔细考虑安全性。如果有人能接触生产服务器，他是否也能访问备份服务器？反过来呢？

弄清楚RPO和RTO可以指导备份策略。是需要基于故障时间点的恢复能力，还是能从昨晚的备份中恢复数据但会丢失此后的所有数据也可以接受？如果需要基于故障时间点的恢复，可能要建立日常备份并保证所需要的二进制日志是有效的，这样才能从备份中还原，并通过重放二进制日志来恢复到想要的时间点。

一般说来，对数据丢失的承受力越强，备份越简单。如果你有非常苛刻的需求，比如要确保能恢复所有数据，备份就很困难。基于故障时间点的恢复也有几类。一个“宽松”的基于故障时间点的恢复需求意味着需要重建数据，直到“足够接近”问题发生的时刻。一个“硬性”的需求意味着不能容忍丢失任何一个已提交的事务，即使某些可怕的事情发生（例如，服务器着火了）。这需要特别的技术，例如将二进制日志保存在一个独立的SAN卷，或使用DRBD磁盘复制。

在线备份还是离线备份

如果可能，关闭MySQL做备份是最简单、最安全的，也是所有获取一致性副本的方法中最好的，而且损坏或不一致的风险最小。如果关闭了MySQL，就根本不用关心InnoDB缓冲池中的脏页或其他缓存。也不需要担心数据在尝试备份的过程中被修改，并且因为服务器不对应用提供访问，所以可以更快地完成备份。

尽管如此，让服务器停机的代价可能比看起来要更昂贵。因此，必须设计不需要生产服务器停机的备份。即便如此，由于一致性的需要，在对服务器进行在线备份时仍然会有明显的服务中断。

在规划备份时，有一些与性能相关的因素需要考虑。

备份时间

将备份复制到目的地需要多久？

备份负载

在将备份复制到目的地时对服务器性能的影响有多大？

恢复时间

把备份镜像从存储位置复制到MySQL服务器、重放二进制日志等，需要多久？

最主要的权衡是在备份时间与备份负载中做出选择。可以牺牲其一以增强另外一个。例如，可以提高备份的优先级，代价是降低服务器的性能。

同样地，也可以利用负载的特性来设计备份。例如，如果服务器在晚上的8小时内仅仅有50%的负载，那么可以尝试这样规划备份：使服务器的负载低于50%且仍能在8小时内完成备份。可以采用许多方法来实现这个目标，例如，可以用*ionice*和*nice*来提高复制或压缩操作的优先级，使用不同的压缩等级，或在备份服务器上压缩而不是在MySQL服务器上。甚至可以利用*lzo*或*pigz*以获取更快的压缩。也可以使用O_DIRECT或fadvise() 在复制操作时绕开操作系统的缓存，以避免污染服务器的缓存。像Percona XtraBackup和MySQL Enterprise Backup这样的工具都有限流选项，可在使用*p v*时加--rate-limit选项来限制备份脚本的吞吐量。

逻辑备份还是裸文件备份

有两种主要的方法来备份MySQL数据：逻辑备份（也叫导出）和直接复制原始文件的裸文件备份。逻辑备份以一种MySQL能够解析的格式来包含数据，要么是SQL语句，要么是以某个符号分隔的文本。[2]原始文件是指存放于硬盘上的文件。

任何一种备份都有其优点和缺点。

逻辑备份

逻辑备份有如下优点：

- 逻辑备份备份的文件是可以用编辑器或像*grep*和*sed*之类的命令查看和操作的普通文件。当需要恢复数据，或只想查看数据但不恢复时，这都非常有帮助。
- 恢复非常简单。可以通过管道把它们输入*mysql*，或者使用*mysqlimport*。
- 可以通过网络来备份和恢复，也就是说，可以在与MySQL主机不同的另外一台机器上操作。
- 可以在类似云数据库这样不能访问底层文件系统的系统中使用。
- 逻辑备份非常灵活，因为*mysqldump*——大部分人喜欢的工具——可以接受许多选项，例如，可以用WHERE子句来限制需要备份哪些行。
- 与存储引擎无关。因为你是从MySQL服务器中提取数据而生成备份的，所以消除

了底层数据存储引擎的差异。[3]

- 有助于避免数据损坏。如果因磁盘驱动器有故障而要复制原始文件时，你将会得到一个错误，并且/或生成一个部分甚至全部损坏的备份。如果MySQL在内存中的数据还没有损坏，当不能得到一个正常的裸文件备份时，或许可以得到一个可以信赖的逻辑备份。

尽管如此，逻辑备份也有一些缺点：

- 必须由数据库服务器完成生成逻辑备份的工作，因此要占用更多的CPU周期。
- 逻辑备份在某些场景下比数据库文件本身更大。[4]ASCII码形式的数据不总是和存储引擎存储数据一样高效。例如，一个整型数需要4字节来存储，但是用ASCII码写入时，可能需要12个字符。当然也可以压缩文件以得到一个更小的备份文件，但这样会使用更多的CPU资源，导致恢复的时间增加。（如果索引比较多，逻辑备份一般要比裸文件备份小。）
- 无法保证导出后再还原出来的一定是同样的数据。浮点表示的问题、软件Bug等都会导致问题，尽管非常少见。
- 从逻辑备份中还原需要MySQL加载和解释语句，将它们转化为存储格式，并重建索引，所有这一切会很慢。

逻辑备份最大的缺点是从MySQL中导出数据和通过SQL语句将其加载回去的庞大开销。如果使用逻辑备份，测试恢复需要的时间将非常重要。

裸文件备份

裸文件备份有如下优点：

- 裸文件备份基于文件的物理备份，它只需将需要的文件复制到其他地方即可完成备份，不需要其他额外的工作来生成原始文件。
- 裸文件备份非常容易跨平台、操作系统和MySQL版本工作。（逻辑导出亦如此。这里特别指出这一点是为了消除大家的担心。）
- 从裸文件备份中恢复会更快，因为MySQL服务器不需要执行任何SQL语句或构建索引。如果有很大的InnoDB表，无法完全缓存到内存中，则裸文件备份的恢复要快得多——至少要快一个数量级。事实上，逻辑备份最可怕的地方就是不确定的还原时间。

裸文件备份也有缺点，比如：

- InnoDB的原始文件通常比相应的逻辑备份要大得多。InnoDB的表空间往往包含很多未使用的空间。还有很多空间被用于存储数据以外的用途（插入缓冲、回滚段等）。
- 裸文件备份不总是可以跨平台、操作系统及MySQL版本的。文件名大小写敏感和浮点格式是可能会遇到麻烦的。你很可能因浮点格式不同而不能将文件移动到另一个系统（虽然主流处理器都使用IEEE浮点格式。）

裸文件备份通常更加简单高效。[5]尽管如此，对于需要长期保留或者是用于满足法律合规要求的备份，尽量不要完全依赖裸文件备份。每隔一段时间需要做一次逻辑备份。

除非经过测试，不要假定备份（特别是裸文件备份）是正常的。对InnoDB来说，这意味

着需要启动一个MySQL实例，执行InnoDB恢复操作，然后运行CHECK TABLES。也可以跳过这一操作，仅对文件运行*innochecksum*，但我们不建议这样做。

建议混合使用裸文件备份和逻辑备份两种方式：先使用裸文件备份，用得到的数据启动MySQL服务器实例并运行*mysqlcheck*。然后，周期性地使用*mysqldump*执行逻辑备份。这样做可以获得两种方法的优点，不会使生产服务器在导出时有过度负担。如果能够方便地利用文件系统的快照，也可以生成一个快照，将该快照复制到另外一台服务器上并释放，然后测试原始文件，再执行逻辑备份。

备份什么

恢复的需求决定需要备份什么。最简单的策略是只备份数据和表定义，但这是一个最低的要求。在生产环境中恢复数据库一般需要做更多的工作。下面是MySQL备份需要考虑的几点。

非显著数据

: 不要忘记那些容易被忽略的数据：例如，二进制日志和InnoDB事务日志。在理想情况下，应该把整个数据目录和MySQL一起备份起来。

代码

: 现代的MySQL服务器可以存储许多代码，例如，触发器和存储过程的代码。如果你备份了mysql数据库，那么也就备份了大部分这类代码，但如果需要还原单个业务数据库，那会比较麻烦，因为这个数据库中的部分“数据”，例如存储过程，实际是存放在mysql数据库中的。

服务器配置

: 假设要从一个实际的灾难中恢复，比如说，地震过后在一个新数据中心中构建服务器，如果备份中包含服务器配置，你一定会喜出望外。

选定的操作系统文件

: 对于服务器配置来说，备份中对生产服务器至关重要的任何外部配置，都十分重要。在UNIX服务器上，这可能包括*cron*任务、用户和组的配置、管理脚本，以及*sudo*规则。

这些建议在许多场景下会被当作“备份一切”。然而，如果有大量的数据，这样做的开销将非常高，如何做备份，需要更加明智地去考虑。特别是，可能需要在不同备份中备份不同的数据。例如，可以单独地备份数据、二进制日志和操作系统及系统配置。

增量备份和差异备份

当数据量很庞大时，一个常见的策略是做定期的增量备份或差异备份。二者有时容易让人混淆，所以先来澄清这两个术语：差异备份是对自上次全备份后所有改变的部分而做的备份，而增量备份则是对自任意类型的上次备份后的所有修改做的备份。

例如，假如在每周日做一个全备份。在周一，对自周日以来所有的改变做一个差异备份。在周二，就有两个选择：备份自周日以来所有的改变（差异），或只备份自周一备份后所有的改变（增量）。

增量备份和差异备份都是部分备份：它们一般不包含完整的数据集，因为某些数据没有改变。部分备份对减少服务器开销、备份时间及备份空间而言都很适合。尽管某些部分备份并不会真正减少服务器的开销。例如，Percona XtraBackup和MySQL Enterprise Backup，仍然会扫描服务器上的所有数据块，因而并不会节约太多的开销，但它们确实会减少一定量的备份时间和大量用于压缩的CPU时间，当然也会减少磁盘空间的使用。[6]

不要因为会用高级备份技术而自负，解决方案越复杂，面临的风险也越大。要注意分析隐藏的危险，例如，如果多次迭代备份紧密地耦合在一起，则只要其中的一次迭代备份有损坏，就可能导致所有的备份都无效。

下面有一些建议：

- 使用Percona XtraBackup和MySQL Enterprise Backup中的增量备份特性。
- 备份二进制日志。可以在每次备份后使用FLUSH LOGS来开始记录一个新的二进制日志，这样就只需要备份新的二进制日志。
- 如果有一些“引用”表，例如，包含不同语种、各个月的名称列表，或者州或区域的简写等，可以考虑将它们单独放在一个数据库中，这样就不需要每次都备份这些表。
- 一个更好的选择可能是把这些数据放到程序代码中，而不是保存在数据库中。
- 某些数据根本不需要备份。有时候这样做影响会很大——例如，如果有一个通过其他数据构建的数据仓库，从技术上讲完全是冗余的，可以仅备份构建仓库的数据，而不是数据仓库本身。即使从源数据文件重建仓库的“恢复”时间较长，这也是一个好想法。相对于从全备份中可能获得的快速恢复时间，避免备份可以节约更多时间开销。临时数据也不用备份，例如保存网站会话数据的表。
- 备份所有的数据，然后发送到一个有去重特性的地方，例如ZFS文件管理程序。

增量备份的缺点包括会增加恢复的复杂性、额外的风险，以及更长的恢复时间。如果可以做全备份，考虑到简便性，我们建议尽量做全备份。

不管如何，还是需要经常做全备份——建议至少一周一次。你肯定不想使用一个月的所有增量备份来进行恢复。即使是一周的，也还是有很多的工作和风险的。

复制

从副本中备份最大的好处是可以不干扰源库，避免在源库上增加额外的负载。这是一个建立副本服务器的好理由，即使不需要用它做负载均衡或提供高可用性。如果钱是一个问题，你也可以把备份用的副本用于其他用途，例如，报表服务——只要不对其做写操作，以确保备份时不会修改数据。副本不必只用于备份的目的，只需在下次备份时能及时跟上源，即使有时因作为其他用途导致复制延时也没有关系。

当在副本上进行备份时，如第9章所述，使用GTID是非常明智的。这避免了必须保存有关复制过程的所有信息，例如，副本相对于源的位置。这对于很多情况都非常有用：克隆新的副本，重新将二进制日志应用到源上以获得指定时间点的恢复，将副本切换为源等。如果要停止在副本上备分，需要确保没有打开的临时表，否则可能导致重启副本失败。

如第9章的“延迟复制”小节中所述，故意将一个副本延迟复制一段时间对于某些灾难场景

非常有用。例如延迟复制一小时，当一个预期外的语句在源上运行后，将有一个小时的时间用于察觉，并在通过中继日志重复事件之前停掉复制。然后可以将副本提升为源，并重放少量相关的日志事件，以跳过错误的语句。这比我们后面将要讨论的指定时间点的恢复技术要快很多。

副本可能与源库中的数据不完全一样。许多人认为副本是与源库完全一样的拷贝，但以我们的经验，源库与副本数据不匹配是很常见的，并且MySQL没有方法检测这个问题。检测这个问题的唯一方法是使用Percona Toolkit中的*pt-table-checksum*之类的工具。防止这种情况的最好方法是使用super_read_only来确保只有复制可以写入副本。

拥有一个复制的副本可在诸如源库的硬盘烧坏时提供帮助，但不能提供彻底的保障。毕竟复制不是备份。

管理和备份二进制日志

服务器的二进制日志是需要备份的最重要元素之一。它们对于基于时间点的恢复是必需的，并且通常比数据要小，所以更容易被进行频繁的备份。如果有某个时间点的数据备份和所有从那时以后的二进制日志，就可以重放从上次全备份以来的二进制日志并“向前回滚”所有的变更。

MySQL也使用二进制日志进行复制，因此备份和恢复的策略经常和复制配置相互影响。经常备份二进制日志是一个好主意。如果你不能承受丢失超过30分钟数据的代价，至少要每30分钟就备份一次。

需要制定日志的过期策略以防止磁盘被二进制日志写满。日志增长的大小取决于工作负载和日志格式（基于行的日志会产生更大的日志记录）。如果可能的话，我们建议尽可能保留日志以备所需。保留日志对于设置复制、分析服务器负载、审计和从上次全备份中按时间点进行恢复，都很有帮助。当决定要保留日志多久时，应该考虑这些需求。

一个常见的设置是使用binlog_expire_logs_seconds变量来通知MySQL定期清理日志，而不应该手动地去删除这些文件。

参数binlog_expire_logs_seconds在服务器启动或MySQL轮询二进制日志时生效，因此如果二进制日志从未填满并轮询，服务器将不会清除旧条目。MySQL服务器通过查看修改时间而不是内容来决定要清除哪些文件。

备份和恢复工具

备份工具有很多，有的很好用，有的较一般。对于裸文件备份，推荐使用Percona XtraBackup。它是开源的，被广泛使用，而且文档也非常完善。对于逻辑备份，我们首选*mydumper*。尽管*mysqldump*随MySQL一起提供，可以开箱即用，但是它的单线程特性会让备份和恢复的时间变得非常长。*mydumper*具有内置的并行性，可以更快地完成逻辑备份。

MySQL Enterprise Backup

这个工具是由Oracle提供的MySQL Enterprise订阅服务的一部分。使用此工具进行备份不需要停止MySQL，也不需要设置锁或中断正常的数据库活动（但是会给服务器带来一些额外的负载）。它支持类似压缩备份、增量备份和备份到其他服务器上的流备份等特性，是MySQL官方的备份工具。

Percona XtraBackup

Percona XtraBackup与MySQL Enterprise Backup在很多方面都非常类似，但它是开源并且免费的。它支持类似流、增量、压缩和多线程（并行）备份操作，它还有许多特别的功能，可以降低在高负载系统上进行备份的影响。

Percona XtraBackup的工作方式是在后台线程不断追踪InnoDB日志文件尾部，然后复制InnoDB数据文件。这是一个稍微复杂的过程，依靠特别的检测机制以确保复制的数据是一致的。当所有的数据文件被复制完，日志复制线程就结束了，结果是在不同的时间点有所有数据的副本。然后可以使用InnoDB崩溃恢复程序将日志应用到数据文件上，以达到所有数据文件有一致的状态。这一步叫作准备过程。一旦准备好，备份就会完全一致，并且包含文件复制过程最后时间点已经提交的事务。一切都在MySQL外部完成，因此不需要以任何方式连接或访问MySQL。

mydumper

几名在职和前MySQL工程师利用他们多年的经验创建了*mydumper*（参见链接37），用来替代*mysqldump*。这是一个多线程（并发）的MySQL备份和恢复工具集，有许多很好的特性。很多人发现多线程备份和恢复的高速度是这个工具最吸引人的特色。

mysqldump

大部分人在使用这个与MySQL一起发行的程序，因此，尽管它有缺点，但在创建数据和schema的逻辑备份时最常见的选择还是*mysqldump*。可以参考官方手册获取更多关于如何使用该工具的细节。

备份数据

大多数时候，生成备份的方法有好的也有差的——有时候显而易见的方法并不是好方法。一个有用的技巧是应该最大化地利用网络、磁盘和CPU的能力以尽可能快地完成备份。这是一个需要不断去平衡的事情，必须通过实验找到“最佳平衡点”。

逻辑**SQL**备份

逻辑SQL导出是很多人所熟悉的，因为它们是*mysqldump*默认的方式。例如，用默认选项导出一个小表将产生如下（有删减）输出：

```
$ mysqldump test t1
-- [Version and host comments]
/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
-- [More version-specific comments to save options for restore]
--
-- Table structure for table `t1`
--

DROP TABLE IF EXISTS `t1`;
/*!40101 SET @saved_cs_client = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `t1` (
 `a` int NOT NULL,
 PRIMARY KEY (`a`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
/*!40101 SET character_set_client = @saved_cs_client */;
--
-- Dumping data for table `t1`
--
LOCK TABLES `t1` WRITE;
/*!40000 ALTER TABLE `t1` DISABLE KEYS */;
INSERT INTO `t1` VALUES (1);
/*!40000 ALTER TABLE `t1` ENABLE KEYS */;
UNLOCK TABLES;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;
/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
-- [More option restoration]
```

导出文件包含表结构和数据，均以有效的SQL命令形式写出。文件以设置MySQL各种选项的注释开始。这些选项要么是为了使恢复工作更高效，要么是为了保证兼容性和正确性。接下来可以看到表结构，然后是数据。最后，脚本重置在导出开始时变更的选项。

导出的输出对于恢复操作来说是可执行的。这很方便，但*mysqldump*的默认选项对于生成一个巨大的备份却不是太适合的。

*mysqldump*不是生成SQL逻辑备份的唯一工具。例如，还可以用*mydumper*或*phpMyAdmin*

工具来创建。我们想指出的是，不是某一个特定的工具有多大的问题，而是单独的SQL逻辑备份本身就有一些缺点。下面是几个主要的问题。

将库表结构和数据存储在一起

如果想从单个文件恢复，这样做会非常方便，但如果只想恢复一个表或只想恢复数据就很困难了。可以通过导出两次的方法来缓解这个问题——一次只导出数据，另外一次只导出schema——但还是会有下一个麻烦。

巨大的SQL语句

服务器解析和执行SQL语句的工作量非常大，所以加载数据时会非常慢。

单个巨大的文件

大部分文本编辑器都不能编辑巨大的或者包含非常长的行的文件。尽管有时候可以用命令行的流编辑器——例如，*sed*或*grep*——来抽出需要的数据，但保持文件小型化仍然是更可取的。

逻辑备份的成本很高

比起逻辑备份这种从存储引擎中读取数据后通过客户端/服务器协议发送结果集的方式，还有其他更高效的方法。

可以看到，在生产环境使用逻辑备份可能很难满足要求。如果使用逻辑备份，强烈建议考虑使用*mydumper*，以避免单线程备份的一些问题，并实际测试使用该工具备份对数据库的影响。

文件系统快照

文件系统快照是一种非常好的在线备份方法。支持快照的文件系统能够瞬间创建与用来备份的内容一致的镜像。支持快照的文件系统和设备包括FreeBSD的文件系统、ZFS文件系统、GNU/Linux的逻辑卷管理（LVM），以及许多的SAN系统和文件存储解决方案，例如，NetApp存储。也有部分云厂商提供远程可挂载磁盘，其可以提供快照的功能。

不要把快照和备份相混淆。创建快照是减少必须持有锁的时间的一个简单方法；释放锁后，必须将文件复制到备份中。事实上，使用文件系统快照，有些时候甚至无须任何锁定，就可以创建InnoDB的备份快照。我们将要展示两种使用LVM来为全InnoDB文件系统做备份的方法，你可以选择最小化锁或零锁。

备份误区**2**：快照就是备份

快照，无论是LVM、ZFS还是SAN快照，都不是真正的备份，因为它们不包含数据的完整副本。因为快照是写时复制，所以它们只包含数据的实时副本与快照发生时的数据之间的差异。如果未修改的块在数据的实时副本中损坏，则没有可用于恢复的该块的完整副本，这时，每个快照都会看到与实时卷相同的损坏块。在进行备份时，可以使用快照“冻结”数据，但不要将快照本身当作备份来依赖。

如果备份是用于某些特殊用户的，那么快照可能是一个非常好的方法。一个例子是在升级过程中遇到问题需要回退的情况。可以在升级前创建一个快照，这样如果升级有问题，只需要回滚到该快照。可以对任何不确定和有风险的操作都这么做，例如，对一个巨大的表

做变更（需要多少时间是未知的）。

LVM快照的工作原理

LVM使用写时复制（copy-on-write）的技术来创建快照——例如，对整个卷的某个瞬间的逻辑副本。这与数据库中的MVCC有点像，不同的是，LVM只保留一个老的数据版本。

注意，我们说的不是物理副本。逻辑副本看起来好像包含了创建快照时卷中所有的数据，但实际上一开始快照是不包含数据的。相比将数据复制到快照中，LVM只是简单地标记创建快照的时间点，对该快照请求读数据时，实际上是从原始卷中读取的。因此，初始的复制基本上是一个瞬间就能完成的操作，不管创建快照的卷有多大。

当原始卷中的某些数据有变化时，LVM在任何变更写入之前，都会将受影响的块复制到快照预留的区域中。LV M不保留数据的多个“老版本”，因此对原始卷中变更块的额外写入并不需要对快照做更多的其他工作。换句话说，对于每个块，只有第一次对每个块的写入才会导致将写时复制写入预留的区域。

现在，在快照中请求这些块时，LVM会从复制块而不是从原始卷中读取。所以，可以继续看到快照中相同时间点的数据而不需要阻塞任何原始卷。图10-1描述了这个方案。

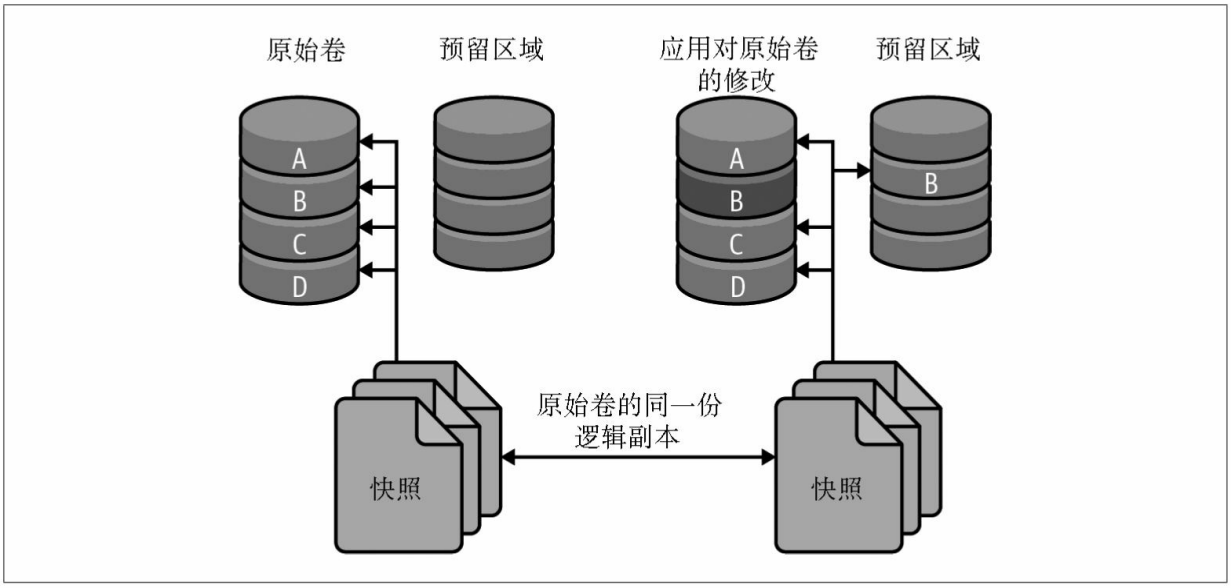

图10-1：写时复制技术是如何减少快照所需空间的

快照会在*/dev*目录下创建一个新的逻辑卷，可以像挂载其他设备一样挂载它。

从理论上讲，这种技术可以对一个非常大的卷做快照，而只需非常少的物理存储空间。但是，必须设置足够的空间，保证在快照打开时，能够保存所有期望在原始卷上更新的块。如果不预留足够的写时复制空间，当快照用完所有的空间后，设备就会变得不可用。这个影响就像拔出一个外部设备：任何从该设备上读取数据的备份任务都会因I/O错误而失败。

先决条件和配置

创建一个快照的消耗非常小，但还是需要确保系统配置允许你获取所有备份瞬间所需文件的一致性副本。首先，确保系统满足下面这些条件。

- 所有的InnoDB文件（InnoDB的表空间文件和InnoDB的事务日志）必须都位于单个

逻辑卷（分区）上。你需要绝对的时间点一致性，LVM不能为一个以上的卷做某个时间点一致的快照。（这是LVM的一个限制，其他系统没有这个问题。）

- 如果需要备份表的定义，MySQL数据目录必须在相同的逻辑卷中。如果使用另外一种方法来备份表的定义，例如，只将schema备份到版本控制系统中，就不需要担心这个问题。
- 必须在卷组中有足够的空闲空间来创建快照。需要多少取决于负载。当配置系统时，应该留一些未分配的空间以便后面做快照。

LVM有卷组的概念，它包含一个或多个逻辑卷。可以按照如下方式查看系统中的卷组：

```
$ vgs
 VG #PV #LV #SN Attr VSize VFree
 vg 1 4 0 wz--n- 534.18G 249.18G
```

输出显示了一个分布在一个物理卷上的卷组，它有4个逻辑卷，大概有250GB空闲空间。如果需要，可用vgdisplay命令产生更详细的输出。现在让我们看一下系统上的逻辑卷：

```
$ lvs
 LV VG Attr LSize Origin Snap% Move Log Copy%
 home vg -wi-ao 40.00G
 mysql vg -wi-ao 225.00G
 tmp vg -wi-ao 10.00G
 var vg -wi-ao 10.00G
```

输出显示mysql卷有225GB的空间。设备名是*/dev/vg/mysql*。这仅是一个名字，尽管看起来像一个文件系统路径。更加让人困惑的是，还有一个从同名文件到*/dev/mapper/vg-mysql*的设备节点的符号链接，用ls和mount命令可以观察到：

```
$ ls -l /dev/vg/mysql
lrwxrwxrwx 1 root root 20 Sep 19 13:08 /dev/vg/mysql -> /dev/mapper/vg-mysql
# mount | grep mysql
/dev/mapper/vg-mysql on /var/lib/mysql
```

有了这个信息，就可以创建文件系统快照了。

创建、挂载和删除**LVM**快照

一条命令就能创建快照。只需决定快照存放的位置和分配给写时复制的空间大小即可。不要纠结于是否使用比想象中的需求更多的空间。LV M不会马上用完所有指定的空间，只是要为后续使用预留空间而已。因此多预留一点空间并没有坏处，除非你必须同时为其他快照预留空间。

让我们来练习创建一个名为backup_mysql的快照，我们给它16GB的写时复制空间：

```
$ lvcreate --size 16G --snapshot --name backup_mysql /dev/vg/mysql
 Logical volume "backup_mysql" created
```

这里特意命名为backup_mysql卷而不是mysql_backup，是为了避免Ta　b键自动补全造成误会。这有助于避免因为Tab键自动补全导致突然误删除mysql卷组的可能。

现在让我们看看新创建的卷的状态：

```
$ lvs
 LV VG Attr LSize Origin Snap% Move Log Copy%
 backup_mysql vg swi-a- 16.00G mysql 0.01
 home vg -wi-ao 40.00G
 mysql vg owi-ao 225.00G
 tmp vg -wi-ao 10.00G
 var vg -wi-ao 10.00G
```

可以注意到，快照的属性与原始设备不同，而且该输出还显示了一些额外的信息：原始卷组和分配了16G B的写时复制空间目前已经使用了多少空间。备份时对此进行监控是一个非常好的做法，可以知道是否会因为设备写满而使备份失败。可以交互地监控设备的状态，或使用诸如Nagios这样的监控系统：

```
$ watch 'lvs | grep backup'
```

从前面mount的输出可以看到，mysql卷包含一个文件系统。这意味着快照也同样如此，可以像其他文件系统一样挂载：

```
$ mkdir /tmp/backup
$ mount /dev/mapper/vg-backup_mysql /tmp/backup
$ ls -l /tmp/backup
total 188880
-rw-r-----. 1 mysql mysql 56 Jul 30 22:16 auto.cnf
-rw-r-----. 1 mysql mysql 475 Jul 30 22:31 binlog.000001
-rw-r-----. 1 mysql mysql 156 Jul 30 22:31 binlog.000002
-rw-r-----. 1 mysql mysql 32 Jul 30 22:31 binlog.index
-rw-------. 1 mysql mysql 1676 Jul 30 22:16 ca-key.pem
-rw-r--r--. 1 mysql mysql 1120 Jul 30 22:16 ca.pem
-rw-r--r--. 1 mysql mysql 1120 Jul 30 22:16 client-cert.pem
-rw-------. 1 mysql mysql 1676 Jul 30 22:16 client-key.pem
... omitted ...
```

这里只是为了练习，因此我们卸载这个快照并用lvremove命令将其删除：

```
$ umount /tmp/backup
$ rmdir /tmp/backup
$ lvremove --force /dev/vg/backup_mysql
 Logical volume "backup_mysql" successfully removed
```

使用**LVM**快照进行无锁**InnoDB**备份

对于MySQL 8.0或者以上的版本，如果使用的都是InnoDB表，而且使用GTID和完全符合ACID的模式，那么备份就非常简单。在MySQL运行时，只需进行快照、挂载快照，然后将文件复制到备份存储中，无须锁定任何文件、捕获任何输出或执行任何特殊操作。从这些备份中进行恢复时，将执行InnoDB崩溃恢复，而GTID配置则会记录哪些事务已被处理。

规划**LVM**备份

规划备份最重要的事情是，为快照分配足够多的空间。我们一般采取下面的方法：

- 记住，LVM只需要将每个修改块复制到快照一次。当MySQL将一个块写到原始卷中时，它会将这个块复制到快照中，然后在异常表中为复制的块生成一个标记。后续对这个块的写操作不会产生任何复制到快照的后果。
- 如果只使用InnoDB，要考虑InnoDB是如何写数据的。InnoDB实际需要对数据写两遍，至少一半的InnoDB的写I/O会落到双写缓冲（doublewrite buffer）、日志文件，以及其他磁盘上相对小的区域中。这部分会多次重用相同的磁盘块，因此第一次写时对快照有影响，但写过一次后就不会对快照带来写压力。
- 接下来，不是反复修改同样的数据，而是要评估有多少I/O需要写入那些还没有被复制到快照写时复制空间的块，对评估的结果要保留足够的余量。
- 使用*vmstat*或*iostat*来收集服务器每秒写多少块的统计信息。
- 衡量（或评估）将备份复制到其他地方需要多久。换言之，需要在复制期间保持LVM快照打开多长时间。

假设评估出有一半的写会导致落到快照的写时复制空间的写操作，并且服务器支持10MB/s的写入。如果需要一个小时（3600s）将快照复制到另外一台服务器上，那么将需要1/2×10MB×3600即18GB的快照空间。考虑到容错，还要增加一些额外的空间。

通常，当快照保持打开时，很容易计算会有多少数据发生改变。

快照的其他用途和替代方案

快照不仅仅用于备份，还有很多其他用途。例如，之前提到过，在一个有潜在危险的动作之前生成一个“检查点”会有帮助。有些系统允许将快照提升为原文件系统，这使得回滚到生成快照的时间点的数据非常简单。

文件系统快照不是取得数据瞬间副本的唯一方法。另外一个选择是RAID分裂：举个例子，如果你有一个三磁盘的软RAID镜像，就可以从该RAID组中移出一块磁盘单独加载。这样做没有写时复制的代价，并且需要时将此类“快照”提升为主副本的操作也很简单。不过，天下没有免费的午餐，如果要将磁盘加回到RAID集合，就必须重新进行同步。

Percona XtraBackup

XtraBackup是备份MySQL最流行的解决方案之一，这是有充分理由的。它支持非常灵活的配置，包括备份压缩、文件加密等。

XtraBackup的工作原理

InnoDB是一个崩溃安全的存储引擎。如果MySQL发生崩溃，它会使用基于重做日志（redo log）的崩溃恢复模式来恢复正确的数据。Percona XtraBackup也是基于这种设计

的。当使用Percona XtraBackup进行备份时，它会记录日志序列号（LSN），并使用该序列号对备份文件执行崩溃恢复。它还会在一些特定的时刻锁定实例，以确保与复制相关的信息准确无误。更详细的说明，请参阅XtraBackup的文档（可参见链接38）。

这是一个XtraBackup备份的示例：

```
$ xtrabackup --backup --target-dir=/backups/

xtrabackup version 8.0.25-17 based on MySQL server 8.0.25 Linux (x86_64)
(revision id: d27028b)
Using server version 8.0.25-15
210821 17:01:40 Executing LOCK TABLES FOR BACKUP…
```

至此，我们可以看到，XtraBackup已经确定了MySQL的运行版本。这可以帮助它确定针对这个版本应该支持哪些功能以及如何备份文件。比如，在这个例子中，可以使用命令LOCK TABLES FOR BACKUP来获取表锁：

```
210821 17:01:41 [01] Copying ./ibdata1 to /backups/ibdata1
210821 17:01:41 [01] ...done
210821 17:01:41 [01] Copying ./sys/sys_config.ibd to /backups/sys/sys_config.ibd
210821 17:01:41 [01] ...done
210821 17:01:41 [01] Copying ./test/t1.ibd to /backups/test/t1.ibd
210821 17:01:41 [01] ...done
210821 17:01:41 [01] Copying ./foo/t1.ibd to /backups/foo/t1.ibd
210821 17:01:41 [01] ...done
210821 17:01:41 [01] Copying ./sakila/actor.ibd to /backups/sakila/actor.ibd
210821 17:01:41 [01] ...done
```

XtraBackup正在将文件从源复制到目的地：

```
210821 17:01:42 Finished backing up non-InnoDB tables and files
210821 17:01:42 Executing FLUSH NO_WRITE_TO_BINLOG BINARY LOGS
210821 17:01:42 Selecting LSN and binary log position from p_s.log_status
210821 17:01:42 [00] Copying /var/lib/mysql/binlog.40 to /backups/binlog.04
up to position 156
210821 17:01:42 [00] ...done
210821 17:01:42 [00] Writing /backups/binlog.index
210821 17:01:42 [00] ...done
210821 17:01:42 [00] Writing /backups/xtrabackup_binlog_info
210821 17:01:42 [00] ...done
```

完成文件复制后，它会收集与复制相关的信息：

```
210821 17:01:42 Executing FLUSH NO_WRITE_TO_BINLOG ENGINE LOGS...
xtrabackup: The latest check point (for incremental): '35005805'
xtrabackup: Stopping log copying thread at LSN 35005815.
210821 17:01:42 >> log scanned up to (35005825)
Starting to parse redo log at lsn = 35005460
210821 17:01:43 Executing UNLOCK TABLES
210821 17:01:43 All tables unlocked
```

现在，XtraBackup已经确定了InnoDB的最新检查点。这将有助于它应用备份期间发生的写入日志。它使用UNLOCK TABLES释放先前LOCK TABLES FOR BACKUP获取的锁：

```
210821 17:01:43 [00] Copying ib_buffer_pool to /backups/ib_buffer_pool
210821 17:01:43 [00] ...done
210821 17:01:43 Backup created in directory '/backups/'
MySQL binlog position: filename 'binlog.000004', position '156'
210821 17:01:43 [00] Writing /backups/backup-my.cnf
210821 17:01:43 [00] ...done
210821 17:01:43 [00] Writing /backups/xtrabackup_info
210821 17:01:43 [00] ...done
xtrabackup: Transaction log of lsn (35005795) to (35005835) was copied.
210821 17:01:44 completed OK!
```

最后一步是记录LSN，导出缓存池中的页面信息，并将其写入最终文件。另外，还有*my.cnf*文件的副本，*xtrabackup_info*文件则包含有关备份的元数据，如MySQL UUID、服务器和XtraBackup的版本信息。

使用示例

我们将重点介绍一些XtraBackup的常见用法，但在此之前有一些事项需要注意：

- 在安装MySQL的时候应正确配置密码保护。所以，这里也需要确保使用--user和--password选项指定具有足够权限的账户来进行备份。
- XtraBackup的输出也很冗长。在示例中，删除了部分输出，以突出每个用例中最重要的部分。
- 此时依旧建议，在运行任何命令之前，都先查看一下Percona XtraBackup的官方手册，因为语法和选项可能会发生变化。虽然我们还没有发现任何与该工具相关的数据丢失问题，但也还是建议你在正式上线使用之前，先对非生产环境进行相关的测试与验证。

将数据备份到某个目录。第一个示例展示了如何使用XtraBackup将数据完整备份到另一个目录中。之后，你再考虑如何处理数据，这里的目录可以是另一块磁盘上的目录，也可以是同一块磁盘上的目录，或更大的备份服务器上挂载的共享文件目录。需要注意，因为是进行全量备份，所以需要有足够的空间来存储备份文件。

这是XtraBackup最基本的用法，指定模式（--backup）和备份文件的位置（--target-dir）：

```
$ xtrabackup --backup --target-dir=/backups/
```

执行后，输出将类似于本章前面“XtraBackup的工作原理”小节中的输出。如果成功，*/backups*目录将包含数据的完整副本。

流式备份。将所有文件复制到新目录可能不是最理想的用例。通常，会在一个目录中保存多个备份。这时可以使用流备份选项（--stream）。这种方式可以将备份写为单个文件：

```
$ xtrabackup --backup --stream=xbstream > /backups/backup.xbstream
```

在这种用法中，我们仍然指定以backup模式运行，并删除了target-dir选项，因为输出将是

标准输出（STDOUT）。然后，再将标准输出重定向到一个文件就可以了。

请注意，还可以使用带有日期的Bash shell命令在目标文件名中包含时间戳，如下所示：

```
$ xtrabackup --backup --stream=xbstream > /backups/backup-$(date +%F).xbstream
```

在整个备份过程中运行方式不变，<STDOUT>始终都是备份的目标端。备份文件将被写入*/backups*中的*xbstream*文件。

压缩备份。正如我们之前提到的，目标端需要有足够的空间来存储数据文件的完整副本，或者需要有足够的空间来存放单个*xbstream*文件。减少空间需求的一种常见方法是使用XtraBackup的压缩功能：

```
$ xtrabackup --backup --compress --stream=xbstream > /backups/backup-
compressed.xbstream
```

你会注意到，以上命令在执行过程中输出的信息不再是“Streaming”而是“Compressing and streaming”。在这个测试中，加载了Sakila样本数据库，并观察到一个原本是94MB的未压缩*xbstream*文件变成了一个6.5MB的压缩文件。

加密备份。要介绍的最后一个示例是在备份中使用加密。加密会使用更多的CPU资源，备份过程会花费更长的时间；但是，考虑到备份很容易通过一个文件获取大量数据，所以作为折中方案还是可以使用加密的。在这里，运行参数依旧覆盖了备份模式、流式传输，但额外使用密码加密encrypt和encrypt-key-file来指向密钥所在的位置：

```
$ xtrabackup --backup --encrypt=AES256 --encrypt-key-
file=/safe/key/location/encrypt.key --stream=xbstream > /backups/backup-
encrypted.xbstream
```

在执行过程中，命令输出的每个备份文件的信息都变成了“Encrypting and streaming”。

请注意，也可以使用参数--encrypt-key在命令行中指定密钥。但最好不要这样做，否则，密钥将在进程列表中，或作为Linux上*/proc*文件系统的一部分，被公开展示。

其他重要参数。通常，我们会关注备份需要多长时间。为了加快备份速度，可以查看并尝试--parallel和-compress-threads选项。使用这些选项会增加CPU使用率，但会减少备份所需的总时间。加密也有类似的并行化选项。

如果你有大量的数据库和表，可以使用--rsync来优化文件复制过程。

从备份中恢复数据

如何恢复数据取决于数据是怎么备份的。可能需要以下部分或全部步骤。

1. 停止MySQL服务器。
2. 记录服务器的配置和文件权限。
3. 将数据从备份中移到MySQL数据目录。
4. 改变配置。
5. 改变文件权限。
6. 以限制访问模式重启服务器，等待其完成启动。
7. 载入逻辑备份文件。
8. 检查和重放二进制日志。
9. 检测已经还原的数据。
10. 以完全权限重启服务器。

我们将在接下来的章节中演示这些步骤的具体操作，还会对本节及本章后面几节提及的一些特殊的备份方法和工具做一些解释。

如果有机会使用文件的当前版本，就不要用备份中的文件来代替。例如，如果备份包含二进制日志，并且需要重放这些日志来做基于时间点的恢复，那么不要把当前二进制日志用备份中的老的副本替代。如果有需要，可以将其重命名或移动到其他地方。

在恢复过程中，需要确保MySQL除了恢复进程外不接受其他访问，这一点往往比较重要。我们首选以--skip-networking和--socket=/tmp/mysql_recover.sock选项来启动MySQL，以确保它对于已经存在的应用不可访问，直到检测完并重新提供服务。这对于按块加载的逻辑备份的恢复来说尤其重要。

恢复逻辑备份

如果恢复的是逻辑备份而不是裸文件备份，则与使用操作系统将文件简单地复制到适当位置的方式不同，需要使用MySQL服务器本身来将数据加载到表中。

在加载导出文件之前，应该先花一点时间考虑文件有多大，需要多久加载完，以及在启动之前还需要做什么事情，例如，通知用户或禁掉部分应用。禁掉二进制日志也是一个好主意，除非需要将还原操作复制到副本：服务器加载一个巨大的导出文件的代价很高，并且写二进制日志会增加更多的（可能没有必要的）开销。加载巨大的文件对于一些存储引擎也有影响。例如，在单个事务中将100GB数据加载到InnoDB就不是一个好想法，因为巨大的回滚段将会导致问题。应该以可控大小的块来加载，并且逐个提交事务。

有两种类型的逻辑备份，所以相应地有两种类型的恢复操作。

如果有一个SQL导出文件，它包含可执行的SQL语句。需要做的就是运行这个文件。假设将Sakila示例数据库和schema备份到单个文件，下面是用来恢复的常用命令：

```
$ mysql < sakila-backup.sql
```

也可以从*mysql*命令行客户端用SOURCE命令加载文件。这只是做相同事情的不同方法，不过该方法可使得某些事情变得更简单。例如，如果你是MySQL管理用户，那么可以关闭在客户端连接中执行语句的二进制记录，然后加载文件，而不需要重启MySQL服务器：

```
SET SQL_LOG_BIN = 0;
SOURCE sakila-backup.sql;
SET SQL_LOG_BIN = 1;
```

需要注意的是，如果使用SOURCE，当将文件重定向到*mysql*时，在默认情况下，会发生一个错误，但不会导致一批语句退出。

如果对备份做过压缩，那么不要分别解压缩和加载，应该在单个操作中完成解压缩和加载。这样做会快很多：

```
$ gunzip -c sakila-backup.sql.gz | mysql
```

如果只想恢复单个表（例如，actor表），要怎么做呢？如果数据没有分行但有schema信息，那么恢复数据并不难：

```
$ grep 'INSERT INTO `actor`' sakila-backup.sql | mysql sakila
```

或者，如果文件是被压缩过的，那么命令如下：

```
$ gunzip -c sakila-backup.sql.gz | grep 'INSERT INTO `actor`'| mysql sakila
```

如果需要创建表并恢复数据，而在单个文件中有整个数据库，则必须先编辑这个文件。这也是有一些人喜欢将每个表导出到各自文件中的原因。大部分编辑器无法应付巨大的文件，尤其如果文件是被压缩过的。另外，你也不会想真正地去编辑文件本身——而只想抽取相关的行——因此可能必须做一些命令行工作。使用*grep*仅抽出给定表的INSERT语句较简单，就像我们在前面命令中做的那样，但得到CREATE TABLE语句比较难。下面是抽取所需段落的*sed*脚本：

```
$ sed -e '/./{H;$!d;}' -e 'x;/CREATE TABLE `actor`/!d;q' sakila-backup.sql
```

不得不承认，这条命令非常晦涩。如果你必须以这种方式恢复数据，那只能说明备份设计得非常糟糕。如果有所规划，可能就不需要痛苦地去尝试弄清楚*sed*如何工作了。只需要将每个表备份到各自的文件，或者可以更进一步，分别备份数据和schema。

从快照中恢复

恢复裸文件备份往往非常直接，换言之，没有太多的选项。这可能是好事，也可能是坏事，具体取决于恢复的需求。一般过程是简单地将文件复制到正确位置。

如果InnoDB的配置方式是比较古老的，比如使用了单表空间来存储所有的表，那么就必须关闭MySQL，将文件复制或移动到正确位置，然后重启。同时还需要确保InnoDB的事务日志文件与表空间文件相匹配。如果文件不匹配——例如，替换了表空间文件但没有替换事务日志文件，InnoDB将无法启动。这也是将日志和数据文件一起备份非常关键的一个原因。

如果使用InnoDB的file-per-table特性（innodb_file_per_table），InnoDB会将每个表的数据和索引存储于一个*.ibd*文件中。可以在服务器运行时通过复制这些文件来备份和还原单个表，但也并不是那么简单。这些文件并不完全独立于InnoDB。每个*.ibd*文件都有一些内部的信息，保存着它与主（共享）表空间之间的关系。在还原这样的文件时，需要让InnoDB先“导入”这个文件。

这个过程有许多限制，如果有需要可以阅读MySQL用户手册中关于每个表使用独立表空间的部分。最大的限制是，只能在当初备份的服务器上还原单个表。用这种配置来备份和还原多个表不是不可能，但可能比想象中要更棘手。

所有这些复杂度意味着恢复裸文件备份会非常乏味，并且容易出错。一个值得倡导的规则是，恢复过程越难越复杂，也就越需要逻辑备份的保护。为了防止一些无法预料的情况，或者应付某些无法使用裸文件备份的场景，准备好逻辑备份总是值得推荐的。

使用**Percona XtraBackup**进行恢复

在本章前面“XtraBackup的工作原理”一节中，我们提到XtraBackup使用InnoDB的崩溃恢复过程来进行安全备份。这意味着在使用XtraBackup备份的文件之前，还需要执行额外的步骤。

如果使用流式备份，则需要先解压缩*xbstream*文件。对于*xbstream*，可以使用xbstream命令提取：

```
$ xbstream -x < backup.xbstream
```

这个命令会将所有文件提取到当前的位置，也可用-C选项预先指定特定的目录。如果使用了压缩或加密选项，还需要使用类似的选项来反转该过程。对于压缩文件，使用--decompress，对于加密过的文件，在指定--encrypt-key-file位置时使用-decrypt选项：

```
$ xbstream -x --decompress < backup-compressed.xbstream

$ xbstream -x --decrypt --encrypt-key-file=/safe/key/location/encrypt.key
 < backup-encrypted.xbstream
```

接下来，是准备文件。准备的过程是实际执行崩溃恢复操作并确保你正在恢复所有数据的过程：

```
$ xtrabackup --prepare --target-dir=/restore
```

如果你没有使用流模式，则可以在备份后立刻执行准备阶段。存储已经完成准备操作的备份文件，在后续需要恢复时，这会大大缩短恢复时间。

一旦完成并成功，就可以立即使用这些文件来启动MySQL：

```
$ xtrabackup --move-back --target-dir=/restore
```

可以将参数--copy-back或--move-back与xtrabackup一起使用，以将文件复制或移动到正确的位置。

XtraBackup将自动从MySQL配置中检测你的data-dir变量，并将文件移动到正确的位置。

原始文件恢复后启动**MySQL**

在启动正在恢复的MySQL服务器之前，还有一些步骤要做。

首先，最重要且最容易忘记的事情是，在启动MySQL服务器之前检查服务器的配置，确保恢复的文件有正确的归属和权限。这些属性必须完全正确，否则MySQL可能无法启动。这些属性因系统的不同而不同，因此要仔细检查是否和之前做的记录吻合。一般需要*mysql*用户和组拥有这些文件和目录，并且只有这个用户和组拥有读/写权限。

建议观察MySQL启动时的错误日志。在类UNIX系统上，可以这样观察文件：

```
$ tail -f /var/log/mysql/mysql.err
```

注意，错误日志的准确位置会有所不同。一旦开始监测文件，就可以启动MySQL服务器并监测错误。如果一切进展顺利，MySQL启动后就有一个恢复好的数据库服务器了。

观察错误日志对于新的MySQL版本更为重要。老版本在InnoDB有错时不会启动，但新版本不管怎样都会启动，只是会让InnoDB失效。即使服务器的启动看起来没有任何问题，那也应该对每个数据库运行SHOW TABLE STATUS来再次检测错误日志。

小结

每个人都知道需要备份，但并不是每个人都能意识到需要的是可恢复的备份。有许多方法可以规划能满足恢复需求的备份。为了避免出现问题，我们建议你要明确并记录恢复点目标和恢复时间目标，并且在选择备份系统时将其作为参考。

在日常基础上做恢复测试以确保备份可以正常工作也很重要。设置*mysqldump*并让它在每天晚上运行是很简单的，但很多时候你没有意识到数据随着时间已经增长到可能只有几天或几周才能再次导入的地步。最糟糕的是，当你真正需要恢复的时候，才发现原来需要这么长时间。毫不夸张地说，一个在几个小时内完成的备份可能需要几周时间来恢复，具体取决于硬件、schema、索引和数据。

不要掉进副本就是备份的陷阱。副本对生成备份而言是一个干涉较少的源，但它不是备份本身。对于RAID卷、SAN和文件系统快照，道理也一样。确保备份可以通过DROP TABLE测试（或“遭受黑客攻击”的测试），还要能通过数据中心失败的测试。如果是基于备库生成备份，需要通过从源重建备份，并从那时起强制执行super_read_only来确保你的所有副本是一致的。

毫无疑问，我们推荐的备份方式是使用Percona XtraBackup进行裸文件备份，或使用*mydumper*进行逻辑备份。这两种方法都可以无侵入地实现二进制的原始数据备份，这样的备份可以通过启动*mysqld*实例检查所有的表来进行验证。有时候甚至可以一石二鸟：在开发或者预开发环境中每天对裸文件备份进行恢复测试，然后再将数据导出为逻辑备份。我们也建议备份二进制日志，并且尽可能久地保留多份备份的数据和二进制文件。这样即使最近的备份无法使用，还可以使用较老的备份来执行恢复或者创建新的副本。

---

[1] 如果只是因为移动到备份目标的是物理文件，将裸文件备份等同于物理备份也可能并不直观。之所以说“不直观”，是因为文件本身根本不是物理的！

[2] 由***mysqldump***生成的逻辑备份并不一定是文本文件。**SQL**导出的数据会包含许多不同的字符集，同样也会包含二进制数据，这些数据并不是有效的字符。对于许多编辑器来说，文件行也可能会太长。但是，大多数这样的文件还是可以被编辑器打开和读取的，特别是当***mysqldump***使用了**--hex-blob**选项时。

[3] 请记住，虽然备份导出数据与引擎无关，但不同存储引擎的功能可能不兼容。例如，不能在导出定义了外键关系的**InnoDB**数据库后，却期望该数据库在一些不支持外键的引擎中还能正常恢复。

[4] 以我们的经验，逻辑备份往往比裸文件备份要小许多，但也并不总是如此。

[5] 值得一提的是，裸文件备份更易出错，它很难像***mysqldump***一样简单。

[6] **Percona XtraBackup**正在开发“真正的”增量备份特性。它将能够备份变更的数据块，而不需要扫描每个数据块。

第11章 扩展MySQL

在个人项目或者说在一家初创公司中运行MySQL，与在一个拥有成熟市场和呈“曲棍球棒式增长”（形容高速增长）的企业中运行它非常不同。在高速的业务环境中，流量可能逐年增长几个数量级，环境会变得更加复杂，随之而来的数据需求也会快速增加。扩展MySQL与其他类型的服务器非常不同，这在很大程度上是因为数据是有状态的。如果要扩展We b服务器，常见的方式就是在负载均衡的后端添加更多的服务器节点，而这通常就是扩展We b服务器的全部工作。

在本章中，我们将解释可扩展性意味着什么，并带你了解可能需要扩展的不同维度。我们将探讨为什么读可扩展是必不可少的，并展示如何安全地完成扩展，以及通过排队等策略使扩展写入更可预测。最后，我们介绍通过使用像ProxySQL和Vitess这样的工具将数据集进行分片以达到扩展写入的目的。在阅读完本章时，你应该能够确定系统有哪些周期性模型，以及如何去扩展读和写。

什么是可扩展性

可扩展性是系统支撑不断增长的流量的能力。一个系统扩展能力的好坏可以用成本和简单性来衡量。如果增加系统的扩展能力十分昂贵或复杂，那么在达到天花板时可能需要花费更多的努力来解决相关问题。

容量是一个和可扩展性相关的概念。系统容量表示在一定时间内能够完成的工作量[1]，但容量必须是可以被有效利用的。系统的最大吞吐量并不等同于容量。大多数基准测试能够衡量一个系统的最大吞吐量，但真实的系统一般不会被使用到极限。如果达到最大吞吐量，则性能会下降，响应时间会变得不可接受且非常不稳定。我们将系统的真实容量定义为在保证可接受性能的情况下能够达到的吞吐量。

容量和可扩展性并不依赖于性能。以高速公路上的汽车来类比的话：

- 系统是高速公路，包含许多车道和汽车。
- 性能是汽车的速度。
- 容量是车道数乘以最大安全速度。
- 可扩展性是在不减慢速度的前提下，能够增加更多车辆和车道的程度。

在这个类比中，可扩展性依赖于多个条件，如换道设计是否合理、路上有多少车辆发生事故或抛锚，以及汽车行驶速度是否不同或是否频繁变道——但一般来说，和汽车的引擎是否强大无关。这并不是说性能不重要，性能确实重要，只是需要指出，即使系统性能不是很高也可以具备可扩展性。

从更高的视角来看，可扩展性就是能够通过增加资源来提升容量的能力。

即使MySQL架构是可扩展的，但应用本身也可能无法扩展，如果很难增加容量，不管原因是什么，应用都是不可扩展的。之前我们以吞吐量来定义容量，但同样也需要从更高的层面来看待容量问题。从这个角度来看，容量可以简单地被认为是处理负载的能力，从几个不同的角度来考虑负载很有帮助。

数据量

应用所能累积的大量数据是可扩展性最普遍的一个挑战，特别是对于现在的很多Web应用而言，因为这些应用从不删除任何数据。例如社交网站通常不会删除以前的消息或评论。

用户数

即使每个用户只有少量数据，如果用户很多，数据量也会累积起来，而且数据量的增长速度会超过用户数量的增长速度。更多的用户意味着要处理更多的事务，并且事务数可能和用户数不成比例。最后，大量用户（以及更多的数据）也意味着更多复杂的查询，特别是当查询依赖于用户间的关系时（关系的数量可以用$N*（N-1）/2$来计算，这里N表示用户数）。

用户活跃度

不是所有的用户的活跃度都相同，并且用户活跃度也不总是不变的。如果用户突然变得活跃，例如，由于增加了一个吸引人的新特性，那么负载可能会明显提升。用户活

跃度不仅仅指页面浏览量，而且即使页面浏览量相同，如果网站中需要大量工作才能生成的部分变得更受欢迎，那么也可能导致更多的工作。另外，某些用户会比其他用户更活跃：他们可能比一般人有更多的朋友、消息和照片。

相关数据集的大小

如果用户间存在关系，应用可能需要在整个相关用户群体间执行查询和计算，这比处理单个用户及其数据要复杂得多。社交网站经常会面临由那些人气很旺的用户组或朋友很多的用户所带来的挑战。

扩展性的挑战可能来自多个方面。在下一节，我们将讨论如何确定瓶颈在哪里，以及如何解决瓶颈。

读限制与写限制工作负载

在考虑扩展数据库基础架构时，首先要审视的问题是，需要扩展读限制工作负载还是写限制工作负载。读限制工作负载是指读取（SELECT）总流量超过服务器容量的工作负载，写限制工作负载则超过了服务器提供DML（INSERT、UPDATE、DELETE）操作的容量。理解你正在对抗的目标包括理解系统的工作负载。

理解工作负载

数据库工作负载包含很多内容。首先是容量，如我们在前面提到的，它是对一段时间内工作的衡量。对于数据库来说，这通常可以归结为每秒的请求量，工作负载的定义之一是系统能达到的QPS数。然而，不要对此感到失望。在20%使用率的CPU下，1000 QPS并不意味着还可以增加4000 QPS，因为并非每个查询都是相等的。

查询有各种形式：读取、写入、主键查找、子查询、联接、批量插入等。每一种都有相应的成本，成本以CPU时间或响应延迟来衡量。当查询等待磁盘返回信息的时间较长时，该时间被计入成本。[2]了解资源的容量很重要，比如有多少个CPU，磁盘读写IOPS、吞吐量的限制是多少，以及网络吞吐量是多少。这其中每一个都会对延迟产生影响，而延迟将直接关系到工作负载。

工作负载是所有类型的查询及其延迟的混合。如果我们用20%的CPU处理1000 QPS，只要它们的延迟相同，我们就可以再增加4000 QPS[3]，这听起来很合理。如果我们真的引入4000个查询，并触及磁盘IOPS瓶颈，那么所有读操作的延迟都会增加。

如果仅仅可以访问基本的系统指标，如CPU、内存和磁盘相关指标，那么你几乎无法理解正在影响哪个指标，你需要确定读和写的性能如何。我们在第3章的“检查读写性能”一节中提供了一个例子，通过那个示例，可以确定读取和写入的延迟。如果结合时间来对这些数字进行趋势分析，可以看到读或写的延迟是否正在增加，以及哪里可能因此受到限制。

读限制工作负载

假设你正在设计一款产品，并选择采用单源主机来处理所有数据库流量的简便方式。增加更多应用节点可以扩展服务用户请求的客户端数，但最终会被单源数据库主机的能力所限制，该数据库主机将要负责响应所有的读取请求。与此相关的主要指标是CPU使用率，高CPU使用率意味着服务器正花费所有的时间处理查询。CPU的使用率越高，查询的延迟也会越长。然而，这并不是唯一的指标。你还可以看到很高的磁盘读IOPS或吞吐量，这表明磁盘被频繁访问或从磁盘读取了大量的数据行。

一开始可以通过添加索引、优化查询和缓存数据来加以改善。一旦这些改进方法用完，就只剩下读限制工作负载这个难题，这是引入副本来扩展读流量的时机。我们将在本章的后面讨论如何使用读副本池来扩展读流量，如何为副本池进行健康检查，以及在开始使用该架构时需要规避哪些陷阱。

写限制工作负载

你也可能会遇到写限制负载，下面是一些关于写限制数据库负载的示例：

- 注册人数或许正以指数级增长。
- 现在是电子商务旺季，销售额和订单数量都在增长。
- 现在是选举季，存在大量的竞选宣传。

这都是业务用例，它们都会导致数据库的写入量呈指数级增加，必须对数据库进行扩展。同样，一个单源数据库，即使可以垂直扩展一段时间，也只能到此为止。当写入量成为瓶颈时，必须开始考虑使用拆分数据的方法，以便在单独的子数据集上接受并行的写入。我们也将在本章后面讨论如何进行分片来达到写扩展的目的。

这是一个合乎逻辑的设问，“如果我同时看到这两种类型的增长呢？”此时需要仔细检查schema，确定是否存在读需求增长比其他写需求增长更快的表数据子集。试图同时为这两者扩展数据库集群会带来很多痛苦和事故。我们建议将表分离到不同的功能集群中，以独立地扩展读取和写入，这是更有效地使用读池来扩展读取流量的先决条件。

既然已经确定了是读限制负载还是写限制负载，接下来我们将讨论如何以有效的方式帮助指导数据的功能拆分。

功能拆分

基于业务中的“功能”来拆分数据是一项和业务背景强相关的任务，需要深入了解数据的用途。这需要与流行的软件架构模式相结合，如面向服务架构（SOA）和微服务。并非所有按功能拆分的方式都相同，在极端情况下，如果要把每张表都放在独立的“功能”数据库中，可能会由于过多的碎片而令事情变得更糟。

如何恰当地将大型的整体/混合数据库拆分为一组合理的较小集群，以帮助业务扩展？以下是一些需要记住的指导原则：

- 不要根据工程团队的组织架构进行拆分，它会经常变动。
- 根据业务功能来拆分表。支撑账户注册的表可以与负责现有客户设置的表分开，而支持新功能的表应该在单独的数据库中开发。
- 不要回避处理数据中混杂了不同业务关系的问题，你不仅需要倡导数据分离，还需要倡导应用程序重构，并需要引入API来实现相互跨界的访问。我们常见的一个例子是将客户身份与其账单混合在一起。

通常一开始会有一些明确业务功能及访问模式的表，因此分离为单独的集群是一个容易实现的目标，但随着业务发展，这种分离会变得越来越不一样。

现在我们已经基于业务功能以一种深思熟虑的方式分割数据，接下来让我们讨论如何使用副本读池来进行扩展以应对读限制负载的问题。

使用读池扩展读

集群中的副本可用于多个目的。首先，在当前源节点出于任何原因需要停止服务时，副本是故障切换的候选对象，无论切换的方式是有计划的还是无计划的。由于这些副本也在不断地运行更新以匹配源节点中的数据，因此也可以使用它们来处理读取请求。

在图11-1中，可以看到这个具有读副本池的新型架构的全貌。

为了简单起见，我们假设应用程序节点仍然通过直连源数据库的方式来完成写入请求。稍后我们将讨论为何连接到源节点能更好地进行扩展。但是，请注意，相同的应用程序节点连接到一个虚拟I P，该虚拟I P充当节点和读副本之间的中间层。这就是副本读池，用来将不断增长的读负载扩展到多台主机。你可能还会注意到，并非所有的复制副本都在池中，这是一种防止不同的读取工作负载相互影响的常见方法。如果你有报告进程，或你的备份进程倾向于消耗所有磁盘I/O资源并导致复制延迟，则可以预留出一个或多个副本节点来完成这些任务，并将其排除在服务于面向客户流量的读池之外。或者，也可以通过复制检查来增强负载均衡器的健康检查机制，自动从读池中移除落后的节点，并在其赶上复制进度时重新引入。当应用程序需要从单点读取时，将读副本转换为可替换资源的灵活性会显著提升，并且可以在不影响到客户的前提下无缝地管理这些资源。

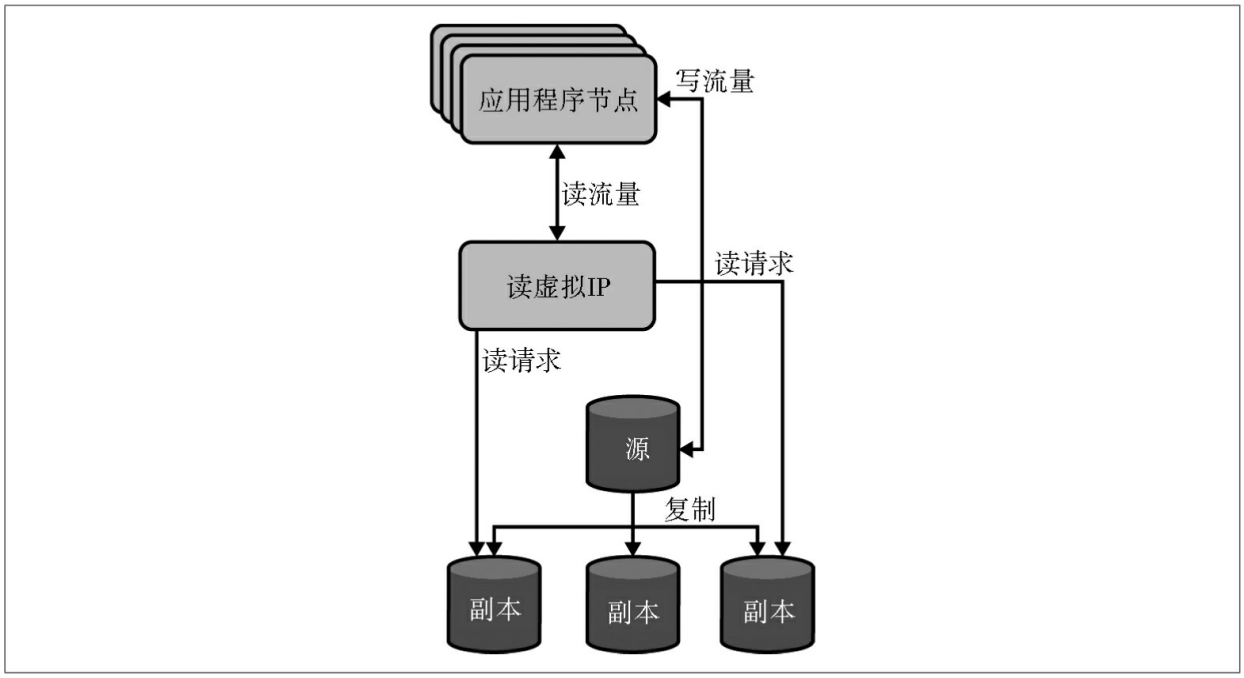

图11-1：应用程序节点通过虚拟IP访问读副本

使用读池时会有不止一台服务读请求的数据库主机，为使生产环境顺利运行，有几点需要考虑：

- 如何将流量路由到读副本？
- 如何均匀地分配负载？
- 如何进行健康检查并移除不健康或延迟的副本，以避免提供陈旧的数据？

- 如何避免因意外地删除所有节点而对应用程序流量造成更多损害？
- 如何手动移除服务器以进行维护？
- 如何将新配置的服务器添加到负载均衡器中？
- 有哪些自动检查可避免在负载均衡器准备就绪之前添加新配置的节点？
- 对“准备好迎接新节点”的定义是否足够明确？

管理这些读池的一种非常常见的方法是使用负载均衡器来提供虚拟I P，该I P充当所有要访问读副本的流量的中介。实现这一目的的技术包括HAProxy、自用主机时的硬件负载均衡器，或在公共云环境中运行时的网络负载均衡器。在使用HAProxy的情况下，所有应用程序主机将连接到一个“前端”，然后HAProxy负责将这些请求引导到后端定义的读副本。下面是一个HAProxy的配置文件示例，它定义了一个虚拟的IP前端，并将其映射到作为后端池的多个读副本：

```
global
 log 127.0.0.1 local0 notice
 user haproxy
 group haproxy

defaults
 log global
 retries 2
 timeout connect 3000
 timeout server 5000
 timeout client 5000

listen mysql-readpool
 bind 127.0.0.1:3306
 mode tcp
 option mysql-check user haproxy_check
 balance leastconn
 server mysql-1 10.0.0.1:3306 check
 server mysql-2 10.0.0.2:3306 check
```

通常，你可以使用配置管理来自动填充这样的配置文件。在进行配置时需要注意几点：在MySQL中，建议使用*leastconn*实现池节点之间的平衡；在负载升高时，像“轮询”这样的随机平衡策略将无助于使用未过载的主机；确保在MySQL实例上创建了正确的数据库用户以运行健康检查，否则所有节点都将被标记为不正常。

一些用于数据分片的工具，如Vitess和ProxySQL，也可以充当负载均衡器的角色。我们将在本章的最后介绍这些工具。

管理读池的配置

现在，在应用程序节点和副本之间有一扇“门”，你需要一种方法来使用选择的负载均衡器，以便轻松管理读池中包含或未包含的节点。你不会希望这是一个手动管理的配置。你已经走上扩展到大量数据库实例的道路，手动管理配置文件可能会导致错误、响应时间变长和主机故障，而且根本无法扩展。

服务发现是一个很好的选择，它可以自动发现新的主机并将其加入池列表。这可能意味着，如果服务发现可用的话，可将服务发现解决方案部署为技术堆栈的一部分，或者依赖于云服务商的托管服务发现选项。这里需要注意的重点是，应确保这些情况读副本符合对应读池这个前提条件成立。理想情况下，可以将源节点和一到多个专门用于报表的副本排除在外。但也许还需要有更复杂的考虑，比如是否进一步分割副本以服务于不同应用程序的读取负载。我们建议除了备份/报表服务器或源节点，每个副本池至少还要有三个节点服务于特定应用。

无论是运行自己的服务发现[4]，还是使用云服务商提供的方案，都应该意识到该服务的保障。不管是运行服务发现还是与一个团队协同工作，都需要考虑以下几点：

- 需要多久能检测到主机的故障？
- 数据传输速率如何？
- 当有数据库实例宕机时，如何刷新负载均衡器上的配置？
- 数据库的成员变更是作为后台进程处理的，还是需要切断现有的连接？
- 如果服务发现自身宕机，会发生什么呢？这是否会损害任何新建的数据库连接，或仅会导致负载均衡器成员身份的错误更改？在这种情况下，可以手动更改吗？

灵活性带来了复杂性，因此必须在两者之间进行权衡，以便在生产中出现故障时能有最好的结果。要做的是始终将决定与所追求的SLI和SLO联系起来，而不是实现一个不切实际的100%正常运行时间的目标。

现在，当需要增加或者减少读节点的时候，就知道该如何配置和更新实例了。

读池健康检查

此时，对于判断读副本是否健康并已准备好接受来自应用程序的流量读取请求，需要考虑一个合理标准。这些标准可以是简单的“数据库进程启动并运行，端口正常响应”，但也可以更加复杂，如“数据库启动，复制延迟不能超过30秒，并且读请求的执行延时不能超过100毫秒”。

检查变量read_only和super_read_only的状态，以确保负载均衡器配置的读池成员都是真正的副本。

决定这些健康检查能做到什么程度时，需要与应用程序开发团队进行讨论，以便每个人对从数据库中读取时所期望的行为都能有相同的理解和认识。以下是一些需要与团队交流的问题，它们可以用来帮助指导这个决策过程：

- 可以接受数据“过期”多长时间？如果返回几分钟前的数据，会有什么影响？
- 应用程序可接受的最大查询延迟是多久？
- 是否有读请求的重试逻辑，如果有，它是怎样的逻辑，以及它是幂等补偿的吗？
- 是否已为应用程序定义SLO？SLO倾向于查询延迟，还是仅关心正常运行时间？

- 当出现数据缺失时，系统的行为是怎样的？这种退化是可以接受的吗？如果可接受，能接受其持续多久？

在很多情况下，只需通过端口检查来确认，MySQL进程是活动的且可以接受连接，就可以了。这意味着只要数据库在运行，就会成为池的一部分并为请求提供服务。

但是，有时也需要更复杂的检测，因为所涉及的数据集非常重要，当复制延迟超过几秒或复制根本没有运行时，便不希望让其提供服务。对于这些场景，仍然可以使用读池，但需要使用HTTP检查来增强健康检查。此工作方式是，让选择的负载均衡器运行一个命令（通常是一个脚本），并根据响应代码确定节点是否正常。例如，在HAProxy中，后端将有这样的代码行：

```
option httpchk GET /check-lag
```

这行代码表示，对于读池中的每台主机，负载均衡器将使用GET调用来调用路径/check-lag，并检查响应代码。该路径运行的脚本包含关于多久延迟可接受的逻辑。脚本将现有滞后状态与阈值进行比较，负载均衡器根据比较结果判断副本是否健康。

尽管健康检查是一个强大的工具，但要小心使用那些具有复杂逻辑的工具（如之前描述的滞后检查），并确保有一个计划，可以确定在池中所有副本都没有通过健康检查时执行什么操作。可以有一个静态的“回退（fallback）”池，在出现某些全局故障（例如，整个集群滞后）时将所有节点重新引入，以避免意外地中断所有读取请求。有关一家公司如何加强这一点的更多细节，请参阅GitHub博客上的这篇文章（参见链接40）。

选择负载均衡器算法

有许多不同的算法可以确定哪台服务器应该接收下一个连接。每个供应商使用不同的术语，下面提供了一些可用的概念。

随机

负载均衡器将每个请求定向到从可用服务器池中随机选择的服务器。

轮询

负载均衡器以重复的顺序向服务器发送请求，如，A、B、C、A、B、C，等等。

最少连接

下一个连接指向拥有最少活跃连接的服务器。

最快响应

处理请求最快的服务器接收下一个连接。当池中包含快慢不一的机器时，这能很好地工作。然而，当查询复杂度变化很大时，处理SQL语句会很棘手。即使是相同的查询，在不同情况下的执行表现也会大不相同，比如当查询缓存为其提供服务时，或者当服务器的缓存已经包含了所需的数据时。

哈希

负载均衡器对连接的源IP地址进行哈希处理，这会将地址映射到池中的一台服务器。当每次连接请求来自同一IP地址时，负载均衡器都会将其发送到同一台服务器。只有当池中的机器数量发生变化时，绑定才会更改。

权重

负载均衡器可以组合几种算法并添加权重。例如，你可能有单CPU和双CPU机器，双CPU机器的能力大约是单CPU机器的两倍，因此可以通知负载均衡器向双CPU机器发送平均请求数两倍的请求。

MySQL的最佳算法取决于具体工作负载。例如，当将新服务器添加到可用服务器池中时，使用最少连接算法可能会导致连接在新服务器缓存未预热之前大量涌入。

你需要进行实验以找到适合工作负载的最佳性能。一定要考虑在特殊情况下和日常情况下会发生什么，以便在某些特殊的情况下——例如，在高查询负载期间，当进行schema更改时，或者当异常数量的服务器脱机时——你至少能承担一些糟糕的后果。

我们在这里只描述了即时资源调度算法，它们不对连接请求进行排队处理。有时，使用排队算法可能更有效。例如，一个算法可能会在数据库服务器上维护一个给定的并发度，比如不允许同时超过*N*个活动事务。如果有太多的活动事务，该算法可以将新的请求放入队列，并根据条件由第一个变为“可用”的服务器为其提供服务。有一些连接池支持排队算法。

到此，我们已经讨论了如何扩展读负载以及如何进行健康检查，现在是时候讨论如何扩展写入了。在了解如何直接扩展写入之前，可以先看看排队机制能使哪些地方的写流量增长更易管理。接下来我们讨论排队是如何帮助扩展写性能的。

排队机制

当使用设计上倾向于一致性而不是可用性的数据存储来扩展写事务时，扩展应用程序层会变得复杂得多。写入一个源数据节点的多个应用程序节点将导致数据库系统更容易出现锁定超时、死锁和必须重试的写失败，所有这些最终将导致影响客户的错误或不可接受的延迟。

在研究我们接下来要讨论的数据分片之前，应该先检查数据中的写入热点，并考虑是否所有的写入都真的需要主动持久化到数据库中。其中的一些是否可以被放入队列中，并在一个可接受的时间范围内写入数据库？

假设有一个存储大量客户历史数据的数据库。客户偶尔会发送API请求来检索此数据，但你还需要安排一个API来删除此数据。你似乎可以用越来越多的副本来提供读API调用的服务，但是删除怎么办呢？HTTP RFC提供了一个响应代码，“202已接受”，可以先返回该响应代码，然后将请求放在队列中（例如，Apache Kafka或Amazon简单队列服务），并控制处理这些请求的速度，以确保删除调用不会直接导致数据库超载。

这显然与意味着“请求已立即完成”的200响应代码不同。人人都知道，与产品团队的协商对于保证API的可信赖和可实现是至关重要的。200和202响应代码之间的区别在于，为了支持更多的并行写操作，需要对数据进行哪些分片处理工作。

如果将队列应用于写负载，那么一个重要的设计选择是，预先确定这些调用在被放入队列后所期望完成的时间范围。监控请求在队列中花费时间的增长，对于此策略何时运行，以及你何时确实需要开始分割此数据集以支持更多的并行写负载，能提供重要的衡量指标。这可以通过分片来实现，我们接下来将讨论分片。

使用分片扩展写

如果不能通过最佳的查询和排队写入来管理写入流量的增长，那么分片将是剩下的选项。

分片意味着将数据切分成不同的、更小的数据库集群，这样就可以同时在更多的源主机上执行更多的写入操作。可以进行两种不同类型的分片或分割：功能分割或数据分片。

功能分割（Functional partitioning），或称为职责划分，意味着将不同的节点用于不同的任务。其中的一个例子可能是将用户记录放在一个集群上，并将其计费放在另一个集群上，这种方法允许每个集群单独扩展。用户注册量的激增可能会给用户集群带来压力。基于使用单独的系统，计费集群负载较少，从而允许进行客户计费操作。相反，如果计费周期日是月初的第一天，你可以清楚地知道运行计费操作不会在其他时间影响用户注册。

数据分片（Data sharding）是当今扩展超大型MySQL应用程序最常见和最成功的方法。通过将数据切分成更小的部分或分片，并将它们存储在不同的节点上，可以达到拆分数据的目的。

大多数应用程序只对需要分片的数据进行切分——通常是数据集中增长非常大的部分。假设你正在建立一个博客服务。如果你期望有1000万名用户，那么可能不需要对用户注册信息进行分片，因为可以将所有用户（或其活动子集）完全放在内存中。另外，如果期望有5亿用户，那可能需要进行数据分片。用户产生的内容，比如博文和评论，在各自的场景中几乎可以肯定需要分片，因为这类记录要大得多，而且数量更多。

大型应用程序可能有多个逻辑数据集，可以以不同的方式被切分。可以将它们存储在不同的服务器集上，但没必要这样做。还可以通过多种方式切分相同的数据，这取决于访问数据的方式。

当计划“只切分需要分片的数据”时，要小心。这个概念不仅需要覆盖快速增长的数据，还需要覆盖逻辑上与之相关的数据，这些数据将定期被查询。如果是基于user_id字段进行切分，但另有一组更小的表在大部分查询中都通过相同的user_id来联接它，那么将这些表一起分片是有意义的，这样可以保证大部分应用查询每次仅在单个分片中进行，能避免跨数据库联接。

选择切分方案

分片技术最重要的挑战是查找和检索数据。如何查找数据取决于如何切分数据。有很多方法可以做好这件事，但其中一些方法更为可取。

目标是使最重要和最频繁的查询接触到尽可能少的分片（请记住，可扩展性的原则之一是避免节点之间的交叉访问）。该过程中最关键的部分是为数据选择一个或多个分片键。分片键决定了每个分片上应该分布哪些行。如果知道对象的分片键，就可以回答下面这两个问题：

- 我应该把这些数据存储在哪里？
- 在哪里可以找到我需要获取的数据？

稍后，我们将展示选择和使用分片键的各种方法。现在，让我们来看一个例子。假设我们

参考MySQL的NDB集群中的做法，使用每个表的主键哈希来将数据切分到所有分片。这是一个非常简单的方法，但它不能很好地扩展，因为经常需要为想要的数据而检查所有分片。例如，如果想要用户3的博客文章，在哪里可以找到？它们可能均匀地分散在所有的分片上，因为它们是按主键而不是按用户划分的。使用主键哈希让我们更容易知道数据被存在哪里，但可能会使获取数据变得更加困难，这取决于需要哪些数据以及是否清楚主键是什么。

我们总是希望将查询本地化到一个分片。当水平切分数据时，总是希望避免必须通过跨分片查询来完成任务。跨分片关联数据将增加应用层的复杂性，并从一开始就削弱了数据分片的好处。使用数据分片的最坏情况是，当不知道所需的数据存储在哪里时，需要扫描每个分片来找到它。

一个良好的分片键通常是数据库中一个非常重要的实体的主键。这些键决定了分片的单元。例如，如果按用户ID或客户端ID切分数据，则分片的单元是用户或客户端。

一个很好的开展这项工作的方法是，用一张实体-关系（E-R）图或一个显示所有实体及其关系的等效工具来绘制数据模型。尝试调整图表布局，从而使相关的实体紧密相连。通常可以直观地查看这样的关系图，并找到可能会错过的分片键的候选项。不过，不要只看图表，还要考虑应用程序的查询。即使两个实体在某种程度上是有关系的，如果很少或从不以该关系进行关联，那么也可以打破此关系来实现分片。

根据实体关系图中的连接程度，某些数据模型比其他数据模型更容易进行切分。图11-2的左图描述了一个容易切分的数据模型，而右图所示的模型则难以切分。

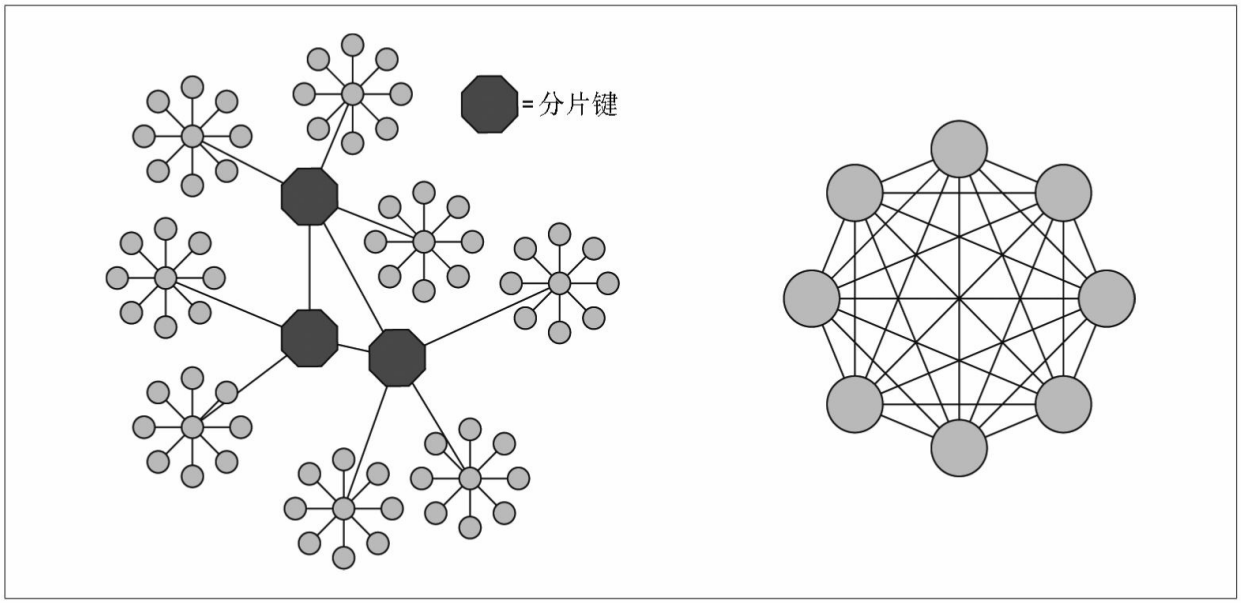

图11-2：两个数据模型，一个容易切分，另一个难以切分[5]

左边的数据模型很容易被切分，因为它有许多连接的子图，这些子图主要由只有一个连接的节点组成，可以相对容易地“切断”子图之间的连接。右边的模型很难被切分，因为它没有这样的子图。幸运的是，大多数的数据模型看起来更像左图，而不是右图。

在选择分片键时，尝试挑选某些能尽可能避免跨分片查询的键，但也要使得分片足够小，这样将不会出现不成比例的大数据块带来的问题。如果有可能，让分片最终均匀地小，如

果做不到，至少要小到很容易通过将不同数量的分片组合在一起来实现平衡。例如，如果应用程序仅在美国运行，而你想把数据集分成20个分片，那么也许不应该按州划分，因为加州拥有庞大的人口数量。但可以按县或电话区号来切分，因为即使人口分布不均匀，这种划分方式也已经足够让你可以选出20个集合，使每个集合的总人口数大致相同，你还可以按亲和关系来选择，这有助于避免跨分片查询。

多个分片键

复杂的数据模型使数据分片更加困难。许多应用程序有多个分片键，特别是当数据中有两个或更多重要的维度时。换句话说，应用程序可能需要从不同的角度看到一个有效的、连贯的数据视图。这意味着可能需要在系统中至少将一些数据存储两次。

例如，你可能需要通过用户ID和博文ID这两者来切分博客应用的数据，因为这是应用程序查看数据的两种常见方式。可以这样想：经常需要看到一个用户的所有文章以及一篇文章的所有评论。按用户切分将无法找到文章的评论，按文章切分不能找到某个用户的文章。如果需要以两种类型的查询来访问单个分片，则必须以两种方式来切分。

仅仅因为需要多个分片键，并不意味着需要设计两份完全冗余的数据存储。让我们看另一个例子：一个社交网络的读书俱乐部网站，此网站的用户可以对书籍发表评论，而该网站可以显示对一本书的所有评论，以及用户已经阅读和评论的所有书籍。

可以为用户数据构建一份切分的数据存储，并为书籍数据构建另一份。评论数据同时具有用户ID和书籍ID，因此它们跨越了不同的分片。可以按用户数据来存储评论，然后按书籍数据仅存储评论的标题和ID，而不是完全复制评论。这可能足以在不访问两份数据存储的情况下呈现书籍评论的大多数视图，如果需要显示完整的评论文本，则可以从用户数据存储中检索它。

跨分片查询

大多数分片应用程序都会有一些查询需要聚合或联接多个分片的数据。例如，如果读书俱乐部网站显示了最受欢迎或最活跃的用户，那么它必须根据定义访问每个分片。实现数据切分最困难的部分是使这类查询正常工作，因为应用程序所认为的单个查询需要被分裂成多个查询且并行执行，每个分片一个。一个好的数据库抽象层可以帮助减轻查询的负担，但即便如此，这样的查询也比分片内查询慢得多，且代价更高，因此主动缓存通常也是有必要的。

如果跨分片查询是例外情况而不是常态，你所选择的分片方案就是一个好方案。你应该努力使查询尽可能简单，并被包含在一个分片内。对于那些需要跨分片聚合的情况，我们建议从应用程序逻辑的整体加以考虑。

跨分片查询也可以从汇总表中获益。可以通过遍历所有分片并在完成后将结果冗余地存储在每个分片上来构建汇总表。如果觉得在每个分片上复制数据太浪费，可以将汇总表合并到另一个数据存储中，这样其就只被存储一次。

非分片的数据通常存在于全局节点中，并使用大量的缓存来使其免受负载的影响。

当一致的数据分布很重要，或没有好的分片键时，一些应用程序会使用基本上随机的分片，分布式搜索应用就是一个很好的例子。在这种情况下，跨分片查询和聚合是常态的，

而不是例外。

跨分片查询并不是使用分片技术后唯一的难题，维护数据的一致性也很困难。外键不能跨分片工作，所以通常的解决方案是根据应用程序的需要检查引用完整性，或者当分片的内部一致性很重要的时候可以在分片中使用外键。可以使用X A事务（参见链接41），但由于开销问题，这在实践中并不常见。

还可以设计间歇性运行的清理程序。例如，如果用户的读书俱乐部账户过期，不必立即删除，可以编写一个定期作业来从每本书的分片中删除用户的评论，而且还可以构建一个定期运行的检测脚本，用来确保不同分片间的数据是一致的。

现在我们已经解释了将数据分到多个集群的不同方法，以及如何选择分片键，接下来我们介绍两个最流行的开源工具，它们可以帮助数据分片与功能分割的顺利进行。

Vitess

Vitess是面向MySQL的一个数据库集群系统。它起源于YouTube的内部项目，后来成长为PlanetScale这个独立的产品，以及一家由Jiten Vaidya和Sugu Sougoumarane共同创立的公司。

Vitess支持许多特性：

- 支持水平分片，包括数据分片。
- 拓扑结构管理。
- 源节点故障切换管理。
- schema变更管理。
- 连接池。
- 查询重写。

让我们来探索Vitess的架构及其组件。

Vitess架构概述

图11-3是Vitess网站上的一张图，展示了其架构的各个部分。

以下是一些需要了解的术语。

Vitess pod

对一组数据库和Vitess相关部件的通用封装，该部件支持分片、拓扑结构管理、schema变更管理和对这些数据库的应用程序访问。

VTGate

为应用程序与操作员控制数据库实例访问提供的服务，以供其尝试管理拓扑、添加节点或切分部分数据。它类似于前面描述的架构体系中的负载均衡器。

VTTablet

在Vitess管理的每个数据库实例上运行的代理。它可以接收来自操作员的数据库管理命令，并代表操作员执行它们。

Topology（元数据存储）

在给定的pod中保存由Vitess管理的数据库实例清单以及相应的信息。

vtctl

对Vitess pod进行操作更改的命令行工具。

vtctld

用来进行相同管理操作的图形化界面。

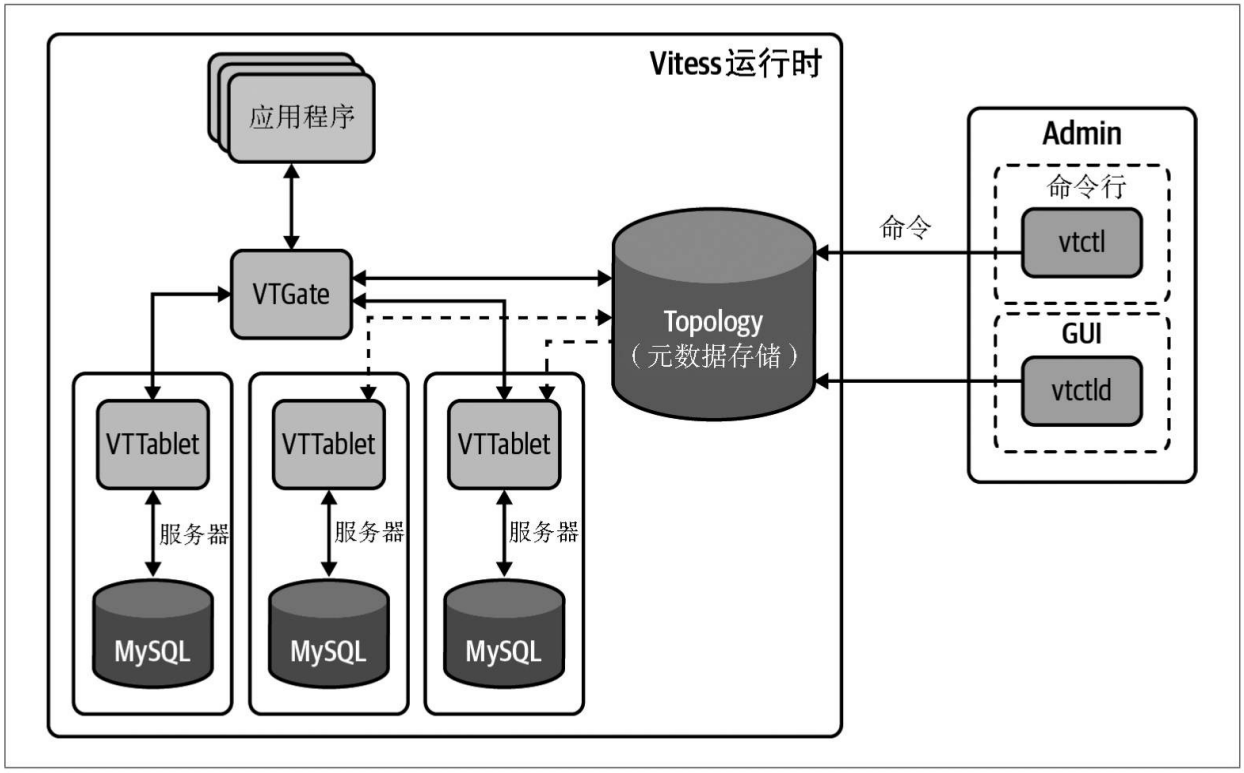

图11-3：Vitess架构图（改编自vitess.io）

Vitess的体系结构从一个一致的拓扑存储开始，该存储包含对所有集群、MySQL实例和*vtgate*实例的定义。这种一致的元数据存储在管理拓扑结构变更方面起着至关重要的作用。当操作员想要更改由Vitess管理的集群的拓扑结构时，实际上是通过一个名为*vtctl*的服务将命令发送到该数据存储，然后该数据存储将此命令的组件操作发送到*vtgate*。

Vitess提供了可以在Kubernetes中部署*vtgate*层和元数据存储的数据库操作器（operator）。在像Kubernetes这样的平台上设置其控制平面，可以提高对单点故障的恢复能力。

Vitess最大的优势之一是它关于如何扩展MySQL的理论思想（参见链接42），其中包括以下内容。

优先使用更小的实例

按功能、水平方式或两者兼而有之来分割数据。当故障发生时，实例越小，所造成的影响面也越小。

通过复制和自动写故障切换来增强弹性

Vitess并没有承诺通过多写节点技术来实现“100%写入可用”。而是在发生故障时，自动进行写入故障切换，并管理拓扑更改和对数据库节点的应用程序访问，以使写停机

时间尽可能短。

通过半同步复制保证持久性

Vitess强烈建议使用半同步复制（相对于默认的异步复制），以确保在向应用程序确认写入操作之前，此写入已被数据库层中的多个节点持久化。这是为了保证持久性而在延迟方面采取的一个重要折中方案，当Vitess需要以一种计划外的方式对写入主机进行故障切换时，这种折中方案会带来好处。

这些体系结构原则有助于维持业务流量的指数级增长，并让基础设施的数据库层具有更大的弹性。无论具体是将Vitess还是其他解决方案用作架构的一部分，都应该注意这些最佳实践。

将技术栈迁移至**Vitess**

Vitess是一个用于运行数据库层的稳定平台，而不是一个临时的解决方案。因此，在将它作为数据库的访问层之前，需要深思熟虑地计划如何实现这种转换。

具体来说，在评估将Vitess作为一个可能的解决方案时，一定要考虑以下迁移步骤。

1. 测试并记录向整个系统引入的延迟。

   在应用程序技术栈中引入像Vitess这样的复杂堆栈肯定会增加一些延迟，特别是在考虑强制使用半同步复制时。确保这种权衡已被很好地记录，并与下游依赖团队进行过明确沟通，以便他们在构建依赖此数据库体系架构的SLO时做出明智的决策。

2. 使用金丝雀部署模型（参见链接43）。

   在生产过渡期间，可以将*vttablet*配置为“外部管理”。当通过应用程序节点组缓慢地增加连接更改时，这允许应用既可以通过*vttablet*也可以直接连接到数据库服务器。

3. 开始分片

   一旦所有的应用程序层访问都是通过*vtgate/vttablet*而不是直接访问MySQL的，就可以开始使用Vitess的完整功能集来将表分离到新的集群，水平切分数据以获得更高的写吞吐量，或者简单地添加副本以获得更多的读负载容量。[6]

Vitess是一个强大的数据库访问和管理产品，从最初在谷歌被研发出来后已经走过了很长的路。它已经证明了其实现显著增长和作为弹性数据库基础设施的能力。然而，这种能力和灵活性是以增加复杂性为代价的。Vitess不像传输流量的负载均衡器那么简单，你需要在业务需求和引入并维护像Vitess这样复杂数据库管理工具的成本之间做出取舍。

ProxySQL

ProxySQL是专门为MySQL协议编写的，通过通用公共许可证（GPL）发布。René Cannaò是该工具的主要创作者，他是一名曾为很多公司提供咨询服务的DBA，也是MySQL的长期贡献者。ProxySQL现在是一家成熟的公司，提供ProxySQL产品的付费支持和开发。

让我们深入了解它的体系结构、配置模式、用例和特性。

ProxySQL的体系结构概述

你可以使用ProxySQL作为任何应用程序代码和MySQL实例的中间层。ProxySQL为应用程序与数据库交互提供了一个会话感知的、基于MySQL协议的接口。与应用程序直接打开

到数据库实例的连接不同，ProxySQL会代表应用程序来打开数据库连接。

这种设计使代理对应用程序节点而言好像是不可见的。它的会话感知允许在MySQL实例之间移动这些连接，而不需要停机。当处理不再维护的应用程序时，这特别有用，因为现在可以利用ProxySQL的特性，而无须对代码进行任何不确定的更改。

ProxySQL还提供了强大的连接池。由应用程序打开的到ProxySQL的连接，与由ProxySQL打开的到已配置为需要连接的数据库实例的连接是分开的。这种隔离使数据库实例不受应用程序层中突然出现的流量峰值的影响。

当你能够单独管理客户端连接与实际的数据库连接时，便引入了以前所没有的灵活性。现在你可以扩展应用程序节点池，而不必担心它会增加到数据库的连接负载，以致超出可以支持的负载上限。这适用于不同应用程序和业务需求的场景，我们将在使用ProxySQL的常见模式中解释这一点。

配置**ProxySQL**

ProxySQL使用配置文件来启动，但会在内存和集成的SQLite文件中维护其运行时配置，可以通过管理界面直接访问和查询这些配置。

ProxySQL的管理界面允许你发出命令来更改正在运行的配置，然后使用MySQL命令将新配置转储到磁盘以进行持久化。这允许你对正在运行的ProxySQL实例进行不停机更改。还可以使用此管理界面来执行由配置管理或自动故障切换脚本发出的自动更改。从图11-4中可以看到体系架构通常是如何将ProxySQL和服务发现结合使用，从而为服务提供一个健壮的访问层的。

需要注意的是，尽管在图11-4中将ProxySQL作为单一对象展示，我们也仍强烈建议在生产环境中利用其集群机制，并在给定的堆栈中部署多个实例。切勿形成单点故障（SPoF）。

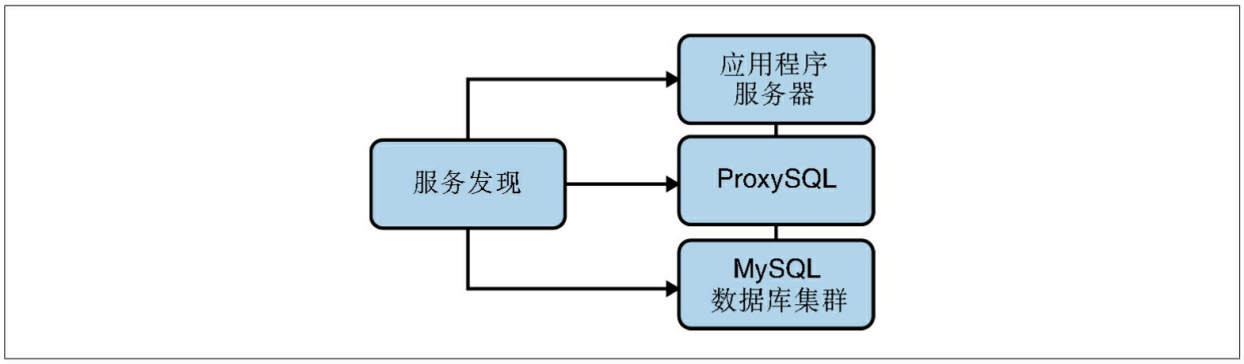

图11-4：应用程序节点、ProxySQL和服务发现之间的交互（改编自Bill Sickles的图表）

ProxySQL对其连接的数据库具有独立的分层健康检查。基于这些健康检查的结果，ProxySQL会添加或删除主机，或调整流量权重。你可以指定复制延迟阈值、成功连接的时间和连接失败重试次数以及许多其他配置选项，以控制在服务和应用程序所需要的环境中可接受多少容错。这些配置选项允许ProxySQL通过临时删除后端数据库，并在稍后重

复进行健康检查，或者完全删除遇到问题的后端成员，直到操作员介入处理，对无响应的主机做出准确反应。

使用**ProxySQL**进行分片

ProxySQL对许多分片拓扑架构都非常有用。虽然它不像Vitess那样为数据的实际切分带来自动化，但它是一个很好的轻量级中间层，而且是分片感知的，还可以相应地路由应用程序连接。让我们来介绍一下使用它作为分片路由层的不同方法。

按用户分片。如果你的数据按业务功能被分割到不同的数据库集群，而且由不同的应用程序组来访问这些数据库集群，那么你还应该为每个应用程序使用完全不同的数据库凭据。ProxySQL可以利用此用户参数，将流量路由到完全分离的用于写或读的后端数据库池。

通过在ProxySQL的管理界面上运行以下命令，然后将更改保存到其磁盘配置文件，可以在ProxySQL中配置此类路由：

```
INSERT INTO mysql_users
(username, password, active, default_hostgroup, comment)
VALUES
('accounts', 'shard0_pass', 1, 0, 'Routed to the accounts shard'),
('transactions', 'shard1_pass', 1, 1, 'Routed to the transactions shard'),
('logging', 'shard2_pass', 1, 2, 'Routed to the logging shard');

LOAD MYSQL USERS RULES TO RUNTIME;
SAVE MYSQL USERS RULES TO DISK;
```

应该始终确保ProxySQL的运行时配置和磁盘上的配置保持同步，以避免在ProxySQL进程重新启动时出现严重的意外。

这也增加了为了合规而记录这些用户所做的所有操作的便利性，而不会给数据库造成任何负载。在第13章中你将看到，出于合规性的原因，我们还建议使用单独的数据库用户，因此，这种设计也符合一些合规性目标。

按schema分片。使用ProxySQL来支持分片数据集的另一种方法是使用schema名称作为规则来管理流量路由。下面是一个在ProxySQL配置中定义的示例：

```
INSERT INTO mysql_query_rules (rule_id, active, schemaname,
destination_hostgroup, apply)
VALUES
(1, 1, 'shard_0', 0, 1),
(2, 1, 'shard_1', 1, 1),
(3, 1, 'shard_2', 2, 1);

LOAD MYSQL QUERY RULES TO RUNTIME;
SAVE MYSQL QUERY RULES TO DISK;
```

注意，只要正确命名schema，此配置就可以用于水平分片或功能分割。

最后一个重要的建议是，当以这种方式使用ProxySQL时，一定要使用其原生的集群特性，以确保像mysql_rules这样的关键配置表被同步到集群中的所有ProxySQL节点，从而在中间件层提供冗余。

使用**ProxySQL**的其他好处

让我们来讨论一些常见的模式。在这些模式中，使用ProxySQL有助于缓解快速增长环境中的常见问题。

在很多应用程序中，当查询延迟开始攀升时，通常会看到“打开更多数据库连接”的提示。然而，在实践中，这可能导致服务中断[7]，并往往导致大量空闲的连接，只消耗资源但不做任何事情。当应用程序层打开直接到数据库的更多连接时，数据库服务器在连接管理上花费的资源量也会增加。这将滚雪球般地形成数千个连接，并压垮已经过载的数据库实例。所有这些活动都会导致长时间的停机、多个微服务中的级联故障，以及扩大面向客户的影响。

ProxySQL的连接管理体系结构通过只向数据库打开可以工作的连接，帮助数据库层避免意外的应用程序峰值出现。ProxySQL可以为不同的客户端请求重用这些连接。这种行为可以将到数据库服务器的单一连接能完成的工作最大化，从而减少了管理连接的资源消耗，并能够更有效地使用数据库服务器的内存资源。

ProxySQL的其他显著特性

ProxySQL还有很多其他功能，使其在通用应用程序代理中脱颖而出：

- 基于端口、用户或简单正则匹配的查询路由。
- 在前端应用程序连接和后端到数据库的连接上都支持TLS。
- 支持各种MySQL风格，如AWS Aurora、Galera Cluster和Clickhouse。
- 连接镜像。
- 缓存结果集。
- 查询重写。
- 审计日志。

通过查阅ProxySQL的文档（参见链接46），可以了解其广泛的功能集（远远超出了分片支持）。

ProxySQL是一个强大的工具，你可以使用它来扩展应用程序，为数据库层提供适当的性能保护，它还添加了支持各种业务需求的特性（如合规性、安全规则等）。如果贵公司发现自己正处于高速增长的轨道上，有大量的新老服务共享数据库资源，那么ProxySQL将是一个保证安全地持续增长的强大工具。ProxySQL提供了一个易于部署的抽象，比HAProxy更复杂，但在基础设施和复杂性方面的前期投入较少。然而，它也没有提供在Vitess中提供的一些更高级的特性，如数据集的自动分片、schema变更管理和VReplication（参见链接47），VReplication是一个强大的工具，用于支持提取、转换、加载（ETL）管道和实时数据流。

小结

扩展MySQL是一段旅程。在阅读完本章后，你应该更有准备地评估扩展需求，并了解如何扩展读取和写入流量，以及如何通过添加队列来使流量增长更可预测。现在，你还应该理解了通过分片来扩展写入以及随之而来的所有复杂决策。

在深入调查可扩展性的瓶颈之前，请确保已经优化了查询、检查了索引，并对MySQL进行了可靠的配置。这可能会为你赢得必要的时间来制定更好的长期策略。一旦优化完成，请专注于确定业务是读限制的还是写限制的，然后考虑哪些策略能最有效地解决当前问题。在规划解决方案时，一定要考虑如何为系统设置长期的可扩展性。

对于读限制的工作负载，我们建议使用读池，除非复制延迟是一个不可克服的问题。如果延迟是问题，或者工作负载是写限制的，那么需要考虑将分片作为下一步的方案。

---

[1] 从物理学来看，单位时间内做的功称为功率（**power**），而在计算机领域，**“power”**是一个被用在多处的术语，有多种含义，因此应避免使用它。但是关于容量的精确定义是系统的最大输出功率。

[2] 简单起见，为解释这个问题，我们选择忽略多个**CPU**核数和上下文切换的复杂性。

[3] 这仍然不是完全准确的，因为随着**CPU**的使用率接近**100%**，延迟就会增加，并将无法再添加**4000**个查询。

[4] 最常用的也是我们推荐的方案是来自**Hashicorp**的**Consul**（参见链接**39**）。

[5] 感谢**HiveDB**项目和**Britt Crawford**提供了这些优雅的图表。

[6] **Morgan Tocker**在**2019**年**Kubecon**大会上的一次演讲中详细解释了这一部署策略（参见链接**44**）。

[7] 更多信息请参见维基百科上关于惊群问题的条目（参见链接**45**）。

第12章 云端的MySQL

对于是否迁移到云端，甚至组织最终采用哪一家云服务商，你很可能都没有太多的控制权。你可以控制的是如何构建数据库环境。有两个方向可以选择：托管MySQL或在VM上构建。托管MySQL往往不需要太多的干涉，但它通常更昂贵，给你的控制权也更少。在VM上构建意味着你在如何构建和观察平台方面获得了更多的灵活性，但这需要更多的时间和操作开销。

在本章中，我们将概述托管MySQL的主要选项，以及它们的作用。还将解释如何开始构建VM选项，包括选择正确的规格和磁盘类型，并且涵盖在云端的VM上运行MySQL时必须了解的操作复杂性（如主机重启）。

我们不会太关注云服务商提供的方案中的一些问题。这些产品都是不断发展的，所以我们建议通过动态信息来源保持获取最新动向，如时事简报或bug页，而不是将像这本书这样某个时间点的资料作为参考。

托管MySQL

一些团队希望随着产品的增长和特性集的扩展能够减少运维MySQL的认知负担，云服务商提供的托管MySQL产品为他们带来了很多便利。每家公有云服务商对托管的SQL数据库应该是什么样的以及它应该如何工作，都有自己的理解。亚马逊网络服务（AWS）提供了几种风格的Aurora MySQL（我们将详细讨论），谷歌云平台（GCP）提供了Cloud SQL，等等，几乎所有的公有云服务商都提供了类似的产品。

托管解决方案的主要吸引力在于服务商提供了一个可访问的数据库设置程序，而不需要用户深入了解MySQL的具体细节。只要点击几下鼠标或运行terraform apply，就可以拥有一个包含副本和定时备份的在线数据库，这样就算准备好了。对于那些想要快速起步的公司或团队来说，这是一个非常有吸引力的选择。

另一方面，使用托管MySQL将缺乏很多的可见性和控制能力。你不能访问操作系统或文件系统，并且在进程自身内能做的事情也会受到限制。除了云服务商提供的信息，你无法检查系统的其他内容。在大多数情况下，如果遇到了问题，只能提交支持工单并等待响应。你不能设置任何高级拓扑结构，而且能用的备份和恢复的方法仅限于云服务商所提供的。

值得注意的是，这些云产品中很多都提供了与MySQL兼容的数据存储。这是一个具有SQL接口的数据存储区，但其内部工作方式可能与本书主要关注的Oracle MySQL完全不同。我们将讨论通用的权衡思路，以及每种托管解决方案有何不同，以帮助你选择最适合团队和业务需求的选项。

Amazon Aurora for MySQL

Aurora MySQL是一个兼容MySQL的托管数据库。Aurora最吸引人的卖点是将计算和存储分开，这使二者可以更灵活地单独扩展。Aurora管理许多经常需要处理的操作任务，例如，执行快照备份、管理快速schema更改、审计日志记录和管理单个区域内的复制。

> 关于兼容性的说明
>
> 当亚马逊说“MySQL兼容”时，你必须确认指的是MySQL的哪个大版本。例如，Aurora中的所有托管解决方案都不兼容MySQL 8.0，而一些较旧的解决方案只兼容MySQL 5.6。如果正在考虑从自建MySQL转移到Amazon Aurora MySQL，在迁移产品数据之前，请务必在应用程序测试中注意这一点。

Aurora MySQL也有不同的产品。我们将简要介绍这些产品之间的区别。

标准的Aurora产品是长期运行的计算实例，在其中选择一个实例类型（就像运行自己的MySQL时一样），然后你就能得到内部的复制到6个副本的附加存储。

截至撰写本书时，Aurora快速DDL被AWS认为是一个“实验室模式”的功

能。如果你在阅读这段文字时情况仍然如此，我们建议你参考第6章来了解更多关于使用数据库外部工具来实现在线schema更改选项的信息。

需要注意的是，Aurora集群内的复制完全是Amazon专有的，而不是我们在Oracle MySQL中所知道和使用的复制。由于集群中的所有Aurora实例共享相同的存储层来访问数据，因此集群内的复制将使用块存储来完成。[1]然而，Aurora支持以我们在社区版MySQL中熟悉的格式写二进制日志，这适用于一些团队，这些团队希望从Aurora集群将数据复制到其他集群，也适用于二进制日志的任何其他用途，如变更数据捕获。[2]

如果你想在Aurora上运行任何执行“关键任务”的数据库，我们强烈建议你考虑使用亚马逊的RDS Proxy来管理应用程序与Aurora通信的方式。在你知道应用程序端可能出现新的连接风暴的情况下，RDS Proxy可以非常方便地让数据库免受新的连接数影响。

自从Aurora MySQL在2015年亮相以来，AWS已经扩展了Aurora MySQL的选项，可以利用这些选项来满足更大范围的用例和业务需求。

Aurora无服务器（Serverless）

Aurora MySQL的无服务器服务移除了长期运行的计算程序，并利用亚马逊的无服务器平台为数据库的计算层提供服务。如果工作负载不需要持续运行，那么这将提供很大的成本灵活性。

Aurora全局数据库（Global Database）

这是Aurora的解决方案，适用于那些需要在多个地理区域提供可用数据，但不想使用二进制日志复制来手动管理从主集群到其他地区集群的数据更改的用户。请注意，这是有代价的，你应该经常查阅亚马逊的文档，以确保当前的理解是正确的。

Aurora多主节点（Multi-Master）

多主节点是Aurora集群的一种特殊风格，可以同时在多个计算节点上接受写操作。它旨在充当一个高度可用的解决方案，其中单个区域的写可用性具有最高优先级。请注意，Aurora多主节点架构有自己的一系列限制。首先，在撰写本书时，它运行的是MySQL 5.6的服务器内核，这使得无法使用很多新特性。在同一个集群中有最大节点数的限制，并且不能在同一部署中混合使用多主和全局数据库。我们认为，在基于每个数据存储和应用程序交互的可用性和一致性进行选择时，Aurora是一个非常稳定的解决方案，并且建议在选择之前仔细考虑你所规定的约束条件和权衡。

AWS持续更新和改进其托管的关系数据库产品，因此我们将不用深入了解各Aurora版本之间功能差异的细节。图12-1提供了一张流程图，可帮助你大致了解哪种类型的Aurora最适合你的需求，以及应在哪些方面做取舍。

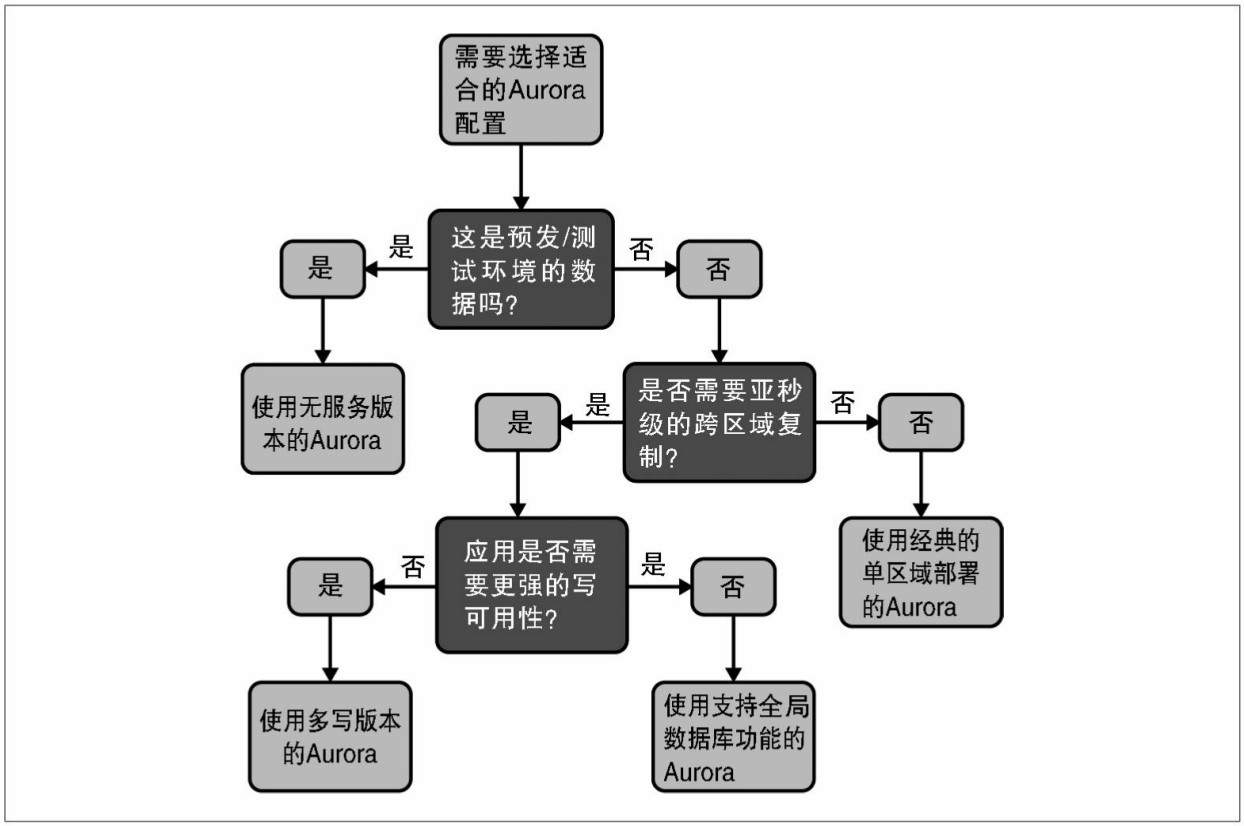

图12-1：一张帮助你选择哪种风格的Aurora是适合你的需求的流程图

图12-1提供的基本的决策树，可以帮助你在各种Aurora选项中做出抉择。重要的是，尽管Aurora提供了很多选择，但总要有所取舍。例如，不能同时实现多写节点的高可用性和亚秒级的跨区域复制。但是，可以使用这些产品来展示这些取舍，并推动关于下面这两者中哪个最重要的产品讨论：写可用性还是区域复制。

Aurora不是唯一由云服务商提供的托管MySQL产品，GCP也有自己的产品。

GCP Cloud SQL

Cloud SQL是GCP的托管MySQL的产品。这款产品与AWS所提供的产品的一个核心区别是，它运行的是社区版MySQL，但特别禁用了某些功能，以实现产品的多租户和可管理。以下是一些不能在Cloud SQL上使用的功能，即使它运行的是社区版MySQL：

- SUPER权限被禁用。
- 插件加载功能被禁用。[3]
- 一些客户端也被禁用了，如*mysqldump*和*mysqlimport*。

与AWS提供的产品类似，你无法获取这些实例的SSH访问权限。

另外，Cloud SQL可以管理以下许多操作任务：

- 原生的高可用性支持。使用配置选项实现自动化故障转移。
- 静止数据的原生加密。
- 使用多种方法实现灵活管理的升级。请注意，这些维护窗口最终涉及一些停机时间

（类似于AWS Aurora），你有责任在其与应用程序的SLO之间做出权衡。[4]

正如我们在本章开头所提到的，你可能无法选择在哪个云服务商那里构建这些数据库，所以你更可能需要知道云服务商的托管选项，以及如何使用它——或者是直接使用VM而不是托管MySQL。

现在我们已经了解了托管关系数据库选项以及这些选择的复杂性，接下来让我们谈谈稍微复杂一点的路径：在云托管的虚拟机上运行MySQL。

虚拟机上的**MySQL**

托管MySQL的特性对于那些想要快速起步的人可能极具吸引力，那么为什么会有人选择运行自建的数据库呢？在虚拟机上运行MySQL就像在裸金属服务器上运行一样，你可以完整和彻底地控制所有的操作面。你可以在单个区域运行MySQL主节点，但在其他区域设置副本以达到灾难恢复的目的——或者运行一个延时的副本。你也可以定制备份方法，以使其对于工作负载来说是最佳的。如果性能下降或遇到问题，则可以完全控制操作系统和文件系统，从而进行任何需要的反省。

云上的机器类型

如在第4章中所讨论的，用于MySQL的CPU核数和可用的RAM会直接影响MySQL的性能。为数据中心选择特定硬件规格的缺点是不能很容易地更改。如果有一台56核、512 GB RAM的机器，当然可以减少安装的RAM——但你已经为它付费了，所以除非可以在其他地方重用RAM，否则你可能在硬件上超支。

当使用云服务时，为工作负载优化机器规格就容易得多。主要的云服务商都允许选择机器规格，这会设置虚拟CPU（vCPU）的范围、可用RAM的总量，以及网络和磁盘限制。这样一来就可以在工作负载发生变化时调整VM的规格。这可能意味着，如果在一年中的特定时间遇到流量高峰——比如假期——则可以通过暂时提升机器规格来应对。一旦流量回落，可以再次将机器规格调低。这种灵活性是许多人转向云计算的原因。

选择正确的机器类型

如果你已经使用了云服务，那么选择一台机器是相当简单的。如果遇到了vCPU、内存或网络瓶颈，可以找到合适的机器类型来解决并调整规格。但是，如果要从一个数据中心开始迁移，那么预先确定正确的配置可能会很棘手。

CPU

在第4章中，我们讨论了如何为工作负载选择合适的CPU。当过渡到云服务时，大部分的指导原则仍然成立。请记住，对于云服务，你得到的是虚拟CPU，而不是物理CPU。这意味着CPU并不是完全属于你的，而是你与同一物理主机上的其他租户共享的。你很可能会看到比在自己的独占服务器上更多的延迟和使用率变化。

如果你正在从物理机迁移到云服务，那么估算CPU使用量也会很麻烦。我们已经成功地总结了以下的vCPU数量的计数公式：

（CPU核的数量×95%CPU总使用量）×2

例如，假设你在一个数据中心中有一台40核的服务器。在过去的30天里，CPU使用量的峰值是30%，那么在云服务环境中达到50%使用率需要运行多少个核？使用前面的公式，我们估算为24个核。如果你选择的云服务商没有提供24核的机器类型，请考虑舍入到最接近的类型，或者确定云服务商是否提供自定义机器类型。[5]

随着CPU使用率或核心数量的增加，上下文切换（在CPU上切换任务的行为）也随之增加。正因为如此，你并不想在100%的CPU容量下运行，因为这将在线程之间进行切换时浪费大量的时间，进而表现为查询的延迟。我们建议将50%作为常规的使用率目标，最高可达到65%～70%。如果维持在70%或更高的CPU使用率，将可能会看到延迟增加，此时应该考虑添加更多的CPU。

如果可以选择的话，还要注意CPU芯片系列。如果运行的是一个高流量的Web应用程序，则可能需要确保使用更新一代的芯片。同样地，如果正在研究后端数据处理，较旧的、稍微慢一点的CPU芯片系列也可以，这可能会节省成本。

内存

如第1章和第4章所讨论的，RAM可以极大地影响MySQL的性能。

应为工作数据集选择最适合需求的机器规格，但错误的做法是RAM太多而不是不够。

网络性能

虽然CPU和内存大小是选择机器类型时最重要的部分，但也一定要检查合理的网络性能限制，以确保不会“饿死”应用程序。例如，如果有一个将读取大量数据的批处理进程，则你可能会发现在较小的机型上带宽已耗尽。

值得注意的是，各云区域或地域之间的网络出口通常有与其相关的成本。当设置副本时，这可能会令人惊讶，但我们仍然认为，为了冗余的目的，将副本放在单独的区域中是很重要的。

选择正确的磁盘类型

虽然机器类型通常是动态的，但对数据存储所做的选择很可能是你遇到的最复杂的决定。一旦选择了一种磁盘类型并开始将其用于数据，换成另一种磁盘类型就会变得十分困难。通常需要挂载第二块磁盘并复制数据。更改不是不可能，但这肯定要比仅快速重启以添加更多的CPU更复杂。

选择正确的磁盘类型也与预期运行的工作负载场景高度相关。高读密集型的工作负载将受益于更多的内存而不是磁盘性能，因为内存访问要快几个数量级。如果工作集大于InnoDB缓冲池，那么将总是需要到磁盘上读取一些数据。写密集型的工作负载总是会转移到磁盘上，这也是大多数人第一次看到磁盘瓶颈的地方。

磁盘的连接类型

首先要做的决定是使用本地连接的磁盘还是网络连接的磁盘。本地连接磁盘的好处是，其提供了令人难以置信的高性能和一致的吞吐量，但也很容易导致数据丢失。这是因为它们被视为仅用于短暂数据的磁盘。如果运行带有本地数据的虚拟机的硬件崩溃，可能会丢失

本地磁盘上的所有数据。同样地，在某些情况下，即使只是关闭实例，也可能意味着，当再次启动时，你是在不同的主机和空磁盘上。本地连接的磁盘通常没有任何复制或RAID。主机级别的磁盘故障可能会导致数据丢失。如果非得如此，我们强烈建议你考虑使用软件RAID，这至少能将数据丢失的概率降到最低。更多信息请参见我们在第4章中对RAID的讨论。

相比之下，网络连接的磁盘可提供冗余和可靠性。网络连接的磁盘可能会遇到本地连接的磁盘不会出现的停顿。并不是说网络连接的磁盘的性能不好——它只是不如本地连接的磁盘的性能好。在本地通常还可以实现更高的吞吐量和IOPS。

当使用网络连接的磁盘时，云服务商提供了便捷的备份或快照工具。假设进行了ACID兼容的配置[6]，并且设计了合理的备份解决方案，那么这些配置对于MySQL的使用来说将工作得很好。可以在任何时候获取磁盘快照，并通过正常的崩溃恢复流程进行恢复而不会出现任何问题。

还可以使用磁盘快照技术让副本复制变得非常快，即使在许多TB级大小的磁盘上。这样做可以让需要在副本可用之前赶上来的复制的延迟降到最低。

请注意，如果使用本地连接的磁盘而不是网络连接的磁盘，那么将需要解决如何使用LVM或像XtraBackup这样的第三方工具自己备份数据的问题。有关备份的更详细的讨论，请参阅第10章。

关于连接类型的最后一个注意点是，云服务商并不像硬件上的RAID卡那样提供诸如写缓存那样的机制（电池或闪存备份）。

SSD与**HDD**

总的来说，你会希望将SSD用于所有内容——特别是MySQL数据卷。如果你发现预算特别紧张，那么可以使用HDD作为引导盘的一种更便宜的选择。在实验中我们发现，SSD的启动速度比HDD快两到三倍。如果启动时间很重要，特别是在停机或重新启动的情况下，那么请始终使用SSD。

IOPS和吞吐量

另一个复杂的因素是确定IOPS和吞吐量需求。在选择需要的磁盘类型之前，你应该充分了解这些需求的历史和未来情况。

如果你正从现有的工作负载中迁移，而且理想情况下已经有了磁盘使用的历史指标，这将保证你做出最好的选择。如果没有，则可以使用Percona Toolkit包中的pt-diskstats（参见链接51）来收集一天的指标以衡量峰值。

对于新的数据库，请花一些时间了解应用程序的密集程度。即使是了解读写比率这样基本的东西也能有所帮助。如果各种尝试都不奏效，请在性能和成本之间找到一个良好的折中点，并设定一个以后可能需要调整的预期。

额外的建议

如果选择在虚拟机上运行自建的MySQL，那么比起使用托管服务，你将要负责更多的事情。你需要自己执行磁盘大小调整、备份等操作。如果采用这种方式，这里有一些建议你可以考虑一下。

处理主机重启

你的虚拟机实际上是运行在别人的硬件上的。尽管我们不喜欢，但硬件可能而且确实会出现故障，当这种情况发生时，虚拟机将立即终止运行。如果提前配置过，虚拟机将开始在另一台主机上启动备份。如果在为生产流量提供服务时发生这种情况，特别是在进行写入的源节点上，这对用户来说会造成中断。

没有什么神奇的解决方案可以让你避免这种情况——你只能处理它。如果发生这种情况，往往有两个选择：启动一个复制副本的故障切换程序（在第9章的“复制切换”一节中有介绍），或者等待源节点恢复在线。处理一个计划外的副本提升可能是一件非常棘手的事情。我们的建议是，只允许服务器恢复在线和复制，以让它自己自然地重新连接。

可以通过以下建议使这个过程更容易处理：

- 使用SSD引导磁盘来允许尽可能快地重新启动。通常系统会在5分钟内恢复在线。
- 禁止长达5分钟时间内的任何服务器宕机的告警通知，以使系统完全重启并恢复正常。
- 如果源服务器重新启动，你可以编写一个选项来动态关闭read_only标志，从而让写入在没有人工干预的情况下继续进行。当与*crond*@reboot选项（该选项将在系统启动时运行一个脚本）结合使用时这会有很好的效果。唯一需要注意的是，你需要能够通过查询系统来确定其是否应该接受写操作。
- 通过自动地向可能需要了解中断的团队或渠道发送邮件或消息，来让沟通最大化。“主机FQDN意外故障，应该在5分钟内恢复在线”警告可能足以阻止人们给你发信息，甚至呼叫你。

分离操作系统与**MySQL**数据

无论选择本地连接的磁盘还是网络连接的磁盘，出于下面这些原因，我们都建议将操作系统和MySQL数据分开：

- 磁盘快照将仅限于MySQL数据，并且不包含任何操作系统信息。
- 在使用网络连接的磁盘的情况下，可以轻松地断开磁盘的连接并重新连接到另一台机器。
- 同样地，对于网络连接的磁盘，还可以升级或替换操作系统，而不必将数据重新复制到文件系统中。

还要考虑将特定文件放在哪里，比如MySQL的进程ID文件、各种日志文件和套接字文件。我们建议将这些文件保留在操作系统中，尽管日志可能会存放在数据磁盘上。

备份二进制日志

将二进制日志发送到一个存储桶中。在桶上设置生命周期控制，以便在一段确定的时间后自动清除旧文件。同样地，阻止在特定时间之前删除文件，或完全不允许删除。

别忘了考虑这里的安全问题。将这个桶开放给所有人读取，可能是一场即将出现的噩梦。控制谁可以读取或删除这些数据对于维护安全的备份策略至关重要。建议允许所有数据库机器写入，但没有机器能够读取或删除。分别或同时控制受限账户、机器的读取和删除。

自动扩展磁盘

对于网络连接的磁盘，需要为预分配的而不是已使用的空间付费。这意味着会在MySQL

数据盘上留下大量已分配但未使用的空间，这是很浪费的。一种优化方法是将磁盘空间使用率目标提到更高的百分比，比如90%，但是如何降低耗尽磁盘空间的风险呢？

云服务商通常有一个可用来扩展磁盘大小的API调用。只需要一点点代码，就可以确定服务器是否超过了90%的“磁盘打满”标记位，并调用该API来扩展磁盘。这可以减少对磁盘空间即将耗尽的服务器进行分页的可能性。总的来说，这个过程可能会大大改善让你在预分配的磁盘空间上的花费。

不过，关于这一点我们将分享一些注意事项：

- 要知道用来查询磁盘空间使用率的代码的合理运行频率。可以根据磁盘的吞吐量计算出一个进程完全填满剩余磁盘需要多长时间，基于这个计算结果，你的代码应该要运行得足够频繁。
- 如果进程失控并无限制地扩展磁盘，那么卷容量可能会一夜之间扩展到64TB。当给服务商支付费用时，这可能是一个昂贵的“惊吓”。
- 此磁盘扩展API调用会导致磁盘短暂停止。一定要在负载下进行测试，以确保它不会对用户产生不利影响。

小结

如果你公司的数据运行在公有云上，那么当涉及如何运行数据库时，可以有很多选择。作为一名数据库工程师，你将被问及使用哪种托管解决方案，是否使用托管关系数据库解决方案，以及每种选择的利弊。当在讨论中提出自己的观点时，要记住的最重要的一点是，天下没有免费的午餐，每一个选择都伴随着一系列的权衡。你所能做的最有用的事情就是，根据业务如何运行以及处于哪个成熟度阶段的综合背景，为这些权衡制定框架，以帮助指导你的组织做出最合适的选择。我们希望本章能够帮助你进行这些话题讨论，从而能够将可掌控的权衡与公司的需求进行比较。

---

[1] 如果你真的想知道该架构的细节，我们强烈推荐你阅读**Aurora**团队在**2017**年发表的**SIGMOD**论文（参见链接**48**）。

[2] **CDC**（变更数据捕获）是数据体系架构中的一种设计模式，用于确定数据何时发生更改，并跨域及跨系统传输该更改。欲了解更多内容，我们强烈推荐你阅读**Martin Kleppman**（**O'Reilly**）编写的《数据密集型应用系统程序》一书的第**11**章。

[3] **Cloud SQL**提供了自己的审计日志解决方案（参见链接**49**），以支持合规性需求。

[4] 更多信息请参见**Cloud SQL**文档中的**“Minimizing the Impact of Maintenance”**（参见链接**50**）。

[5] 请注意，自定义机器类型可能会比预定的机器类型花费更多。当在有大量实例的环境中工作时，在选择规格时考虑成本总是很重要的。

[6] 提醒一下，设置包括**innodb_flush_log_at_trx_commit=1**和**sync_binlog=1**。

第13章 MySQL的合规性

数据库工程团队的角色是许多内部业务相关部门感兴趣的。正如我们已经介绍过的，你不仅要为性能和业务的在线时间进行规划，还要为基础设施成本、灾难恢复和各种合规性需求做规划。

DBA的工作并不局限于在业务运行时管理这些数据，还需要帮助企业保护数据，并使数据符合法律要求，或获得对企业至关重要的监管认证。你必须了解满足这些需求的业务目标，并将这些需求包括在所有数据体系结构设计中，包括如何自动化操作任务、管理访问，以及如何将管理任务转换为自动化此类任务的代码。

本章涵盖了企业可能需要的不同类型的合规性认证，以及它们所关注的各种特定于数据库的问题。我们会帮助解释如何针对不同的合规性需求进行设计，并讨论访问日志记录如何成为满足合规性要求的关键部分。最后，我们将讨论数据主权，其是所有类型的业务中数据架构实践的一个新兴关注点。

本章并不是要给你提供法律建议。我们希望在你运行大量数据库时帮助你管理合规性需求，以及如何在早期为合规性进行设计。当寻求关于如何正确执行具体控制的建议时，你应该始终咨询公司的法律团队。

什么是合规性

治理（Governance）、风险管理（Risk management）和合规性（Compliance），合称为GRC，它们是一些原则、流程和法律，用于指导企业如何评估和优先考虑其资产的风险，以及如何遵守个人或健康数据处理和传输的法律法规。早期的创业公司通常没有太多合规性需求，重要的是发现适合市场的产品。然而，随着业务的增长，你将开始碰到一些合规性要求。有些合规性要求适用于企业的所有数据，有些则适用于特定的部分数据。

合规性中经常使用的术语是控制。控制是公司内部定义的流程和规则，用于减少意外风险结果发生的概率。

让我们介绍一些你应该了解的合规性要求。稍后，我们将介绍一些更改体系结构的方式，可以使满足不同合规性要求的管理更容易。

服务组织控制类型**2**

服务组织控制类型2（SOC 2）（参见链接52）是一组合规性控制要求，服务组织可以使用这些控制来报告其与安全性、可用性、处理完整性、机密性和隐私性相关的实践。希望获得SOC 2认证的组织中的数据库工程师需要在数据库变更管理、备份和恢复过程，以及管理对数据库实例的访问方面具有良好的实践经验。

萨班斯-奥克斯利法案

2002年发布的萨班斯-奥克斯利法案（SOX）（参见链接53）是所有上市公司都必须遵守的法律。其目的是通过提高根据证券法和其他目的进行的公司信息披露的准确性和可靠性来保护投资者。对于工程组织来说，SOX职责要求证明，包含影响收入数据的数据库仅由有需要的人访问，对该数据的任何更改都会被记录，并且这些更改的原因也是有文档记录的。

如果你是一家上市公司，SOX control 404是法律要求的控制，你必须熟悉并履行。其目的是用证据保证公司报告的财务状况是由数据访问和变更管理实践支持的，这些实践准确地将所提供的服务呈现为对应的收入，并为此类数据的任何变更提供审计跟踪。

支付卡行业数据安全标准

支付卡行业数据安全标准（PCI-DSS）（参见链接54）是所有处理信用卡数据的金融机构所需遵守的标准。它的目的是保护信用卡持有人的数据不被泄露和用于欺诈交易。

作为数据库工程师，PCI-DSS控制的一个重要方面是管理对持卡人数据的访问。这意味着需要考虑架构中的控制，以确保卡数据被单独管理。我们将在本章稍后介绍角色分离时如何实现这一点。

健康保险可携带性和责任法案

1996年发布的《健康保险可携带性和责任法案》（HIPAA）是美国的一项法规（参见链

接55），旨在保护健康提供者、健康计划或其商业伙伴收集和处理的与个人健康相关的隐私数据。本法适用于被定义为电子个人健康信息（ePHI）的数据。提供符合HIPAA要求的产品的组织需要他们的数据库工程师实现控制，例如，对ePHI的访问控制、所有ePHI的加密，以及访问ePHI时的活动日志。

联邦风险和授权管理计划

联邦风险和授权管理计划（FedRAMP）（参见链接56）是联邦政府提供的一项认证，旨在证明在美国运营并希望与美国政府实体开展业务的公司有资格成为联邦实体的云服务商。它是一个标准的集合，有资格以联邦实体作为客户的企业都需要满足该项认证。这些标准包括配置管理、访问控制、安全评估，以及对数据访问和更改的审核。

通用数据保护条例

通用数据保护条例（GDPR）是欧盟于2016年推出的一项法规，旨在管理作为数据处理机构的实体如何存储和管理欧盟人员的个人身份信息，无论这些实体的总部位于哪里。它引入了管理数据隐私的第一步，比如在收集私人数据前需要征得同意，对处理机构组织内访问私人数据设置限制，并为个人提供法律途径，允许个人提交申请，从数据处理机构的系统中清除任何可能通过他们的在线活动收集的数据。这被称为个人的“被遗忘的权利”。

Schrems II

2020年，欧盟司法法院裁决了欧盟和Facebook爱尔兰分部之间的一桩案件。这项通常被称为Schrems II（参见链接57）的裁决，对所有运营和收集欧盟人员数据的美国公司产生了广泛影响。

Privacy Shield（参见链接58）是美国公司多年来在欧盟运营的合法平台。施雷姆斯的裁决宣布，当美国公司在欧盟的实体收集欧盟人员的数据时，不足以保护他们的隐私。取消的核心是由欧盟司法法院裁决的，隐私保护不足以提供保证，欧盟的人将不会由美国政府通过美国法律手段被监控[即使用一种机制所提供的1978年外国情报监视法案（参见链接59），或外国情报监视法]，因此，美国实体收集的欧盟人员的个人身份数据必须留在欧盟，不能进入美国资产或被美国人获取。

与GDPR的初始版本相比，这项裁决使得对数据架构进行推理变得更加复杂。由于这项裁决是在最近才做出的，所以执行情况仍是个未知数。这种情况让每个业务决定它收集和处理的数据有多少在范围内，以及以何种方式处理。如果你在欧洲有任何当前或未来的客户，可以放心地假设Schrems II将为你运行的应用程序和数据基础设施提供服务。

建立合规控制体系

正如我们所见，全球都在加强企业的合规性要求，由于企业运营所使用的数据量是巨大的，根据每项法律的目标或企业需要的认证，控制措施可能会有很多。好消息是，同样的工作有足够的空间覆盖多个合规性控制，从而可以在管理基础设施时实现高效和更一致的实践。然而，你确实需要了解业务需要哪些合规性控制措施，以及出于什么目的。一旦公司发展到一定规模，需要开始实施这些监管合规控制的任何子集，你将成为向不同类型的审计师提供合规证据的人。了解每项控制的目的将大大有助于提供正确的证据，使审计更容易。

合规性的构建是一个持续的过程，在需要的时候不容易“添加”。一旦你的公司度过了“仍在寻找市场匹配”的阶段，本章中提出的许多架构建议（角色分离、跟踪变更等）都是你应该考虑和倡导的东西。当合规性成为一种真正的需要，而不仅仅是一种“不错的拥有”时，这些做法将为你的企业的成功奠定基础。

机密信息管理

在讨论如何管理机密信息之前，让我们先看看基础设施中的哪些内容可能属于该定义：

- 应用程序与数据库交互的密码字符串。
- 支持人员/操作人员管理数据库实例的密码字符串。
- 可以访问/修改数据的API令牌。
- SSH私有密钥。
- 证书密钥。

在组织中，为了便于进行大量的安全控制，你需要一种核心能力，即安全地管理机密信息并将其与管理配置分开。你需要一种方法来交付和转换敏感数据（如数据库访问凭据），无论它是供应用程序还是团队使用，或是用于报告目的。

如果你在云环境中运行应用程序和数据库，我们强烈建议在构建自己的机密信息管理解决方案前，优先从云服务商那里寻求合适的方案。方案中的加密级别必须满足国家标准与技术研究所（NIST）标准[1]可接受的最低要求，包括HIPAA和FedRAMP在内的许多法规都有加密级别的要求。

如果云服务商没有满足要求的机密信息管理解决方案，那你可能不得不自己构建。这对组织来说可能是一项新的尝试，需要比本书提到的方案付出更多的努力。

无论是使用托管解决方案，还是最终需要运行自己的解决方案，都要注意机密信息管理解决方案会给整个体系结构带来复杂性。此解决方案的目标是管理机密信息，而不是成为产品的单一故障点。当机密信息管理解决方案可用时，应该清楚地写好取舍的理由，解释会发生什么，这将派上用场。提前与法律团队和安全组织进行明确的对话，了解哪些机密信

息应被缓存，以及以何种方式缓存，这将有助于避免以后的预期偏差。

通常，一家公司在其早期开发阶段基于便利性做出的决策，在制定合规性控制计划之前需要重新考虑。你应该准备好向领导解释为什么这项工作在改善你的安全态势和降低风险方面很重要。

不要共享用户

不要跨服务共享数据库凭据。假设数据库中的信息意外泄露，现在必须评估该问题的影响面，以确定哪些应用程序和流程必须使用新的数据库凭据。如果各个服务使用了独立的数据库凭据，那么在处理这个问题时能节省大量成本。作为一名数据库工程师，加入一家初创公司是很常见的，在那里，人们选择了看似方便的快捷方式，即“所有代码都使用同一对凭据来访问数据库”。相信我们：这是一条非常昂贵的捷径，如果将每个数据库用户的访问权限限制为其所需的服务，你将来会感谢你自己的。

我们已经介绍了这个基本但至关重要的基础实践，现在让我们讨论在选择存储数据库凭证或密码的解决方案时需要考虑的事项。

不要在代码仓库中提交生产数据库凭据

这似乎是显而易见的，但正如我们在很多大大小小公司的安全事件事后报告中看到的那样，这种情况一直在发生。对此，保持谦虚的心态是很重要的，不要认为这种错误在你的组织中不太可能发生。一种“信任但验证”的方法大大有助于防止此类问题。一种常见的做法是在合并一个pull请求之前，扫描代码库寻找潜在的机密字符串（GitHub等代码仓库托管服务可以实现）。如果你的组织还没有考虑到这一点，那么你需要成为这一需求的倡导者。请记住，合规性和安全性是整个组织所需要的，尽管不是所有的事情都可以或者应该由数据库团队完成，但在整个工程组织谈论这些优先事项的方式中，你是一个利益相关者。

这些实践对于正确开始合规性和安全姿态至关重要。在使用机密信息管理解决方案时，它们将要涉及的一些操作变得更加简单，尤其是需要进行紧急更改时，这将进一步降低业务风险。

接下来让我们看看选择机密信息管理解决方案时需要做的取舍。

选择机密信息管理解决方案

选择机密信息管理解决方案将取决于运行环境，以及如何最简单地实现与数据库和应用程序堆栈的集成。在方便和满足所有需求之间，总会有取舍。因此，你需要让所有利益相关者（其中一些人不在工程领域）清楚有哪些限制，以及可用性或可恢复性的权衡是什么。当检查你的云（或私有）基础设施可以提供什么的时候，你应该考虑以下取舍。

空间限制

> 很多云服务商提供的机密信息管理解决方案对机密信息的长度做了限制，如果想存储比数据库的用户和密码对更长的内容，可能会导致意外。如果合规性控制还要求将较长的文本字符串视为机密信息（例如，SSH密钥或SSL专用证书），则应检查给定密钥上可存储容量的最大大小。一些组织后来偶然发现的地雷之一是，随着机密信息长度的增长，需要一种新的、不同的解决方案来容纳更长的机密信息。现在，要么必须迁移（这可能会影响正常运行时间），要么使用工具来集成两个独立的机密信息解决方案，这两种方式都有其自身的复杂性。

机密信息的轮换

如果是在公有云中运行，且可以使用其托管的机密信息管理解决方案，那么有一个好消息：截至本文撰写之时，所有三种主要的云服务都提供了一些方法，可以自动化地处理机密信息的轮换和版本控制，从而使你的服务无缝地过渡到新的机密信息。如果选择的机密信息管理解决方案不支持轮换，那么你需要规划如何进行计划内的更改（例如，可能有一个控制机制要求数据库凭据按一定的节奏轮换）和计划外的紧急更改（例如，有人意外地在公共代码存储库中上传了数据库密码）。如何做到这一点而不影响正在运行的应用程序呢？这在很大程度上取决于如何将配置更改发布到正在运行的应用程序中，以及部署管道的操作方式。这就是如何编排它的大致思路。这个更改是一个部署工作。即使你的应用程序将数据库凭据作为配置项访问，你仍然需要将此配置更改应用到所有系统，并且通常还需要在不影响整个服务可用性的情况下安排重新启动。

区域的可用性

考虑到机密信息管理解决方案不仅仅是为了存储机密信息，也是为了传递机密信息。如果要避免在代码中存储机密信息等已知错误做法，则需要应用程序能够在运行时检索处理请求所需的机密信息。这意味着你现在必须考虑如何检索这些机密信息，如果应用程序无法访问机密信息管理服务会发生什么，以及这种新的依赖关系会引入什么故障模式。如果你负责在许多不同的地理区域运行应用程序，则机密信息管理解决方案的区域可用性将成为另一件需要考虑的事情。云服务商的解决方案是否会自动将机密信息复制到其他地区？还是你必须自己建立这种能力？

角色与数据分离

这些监管法律的一个重要目标是，基于数据泄露将给企业或客户带来的风险等级对数据进行分离。这是最小特权的概念，既适用于人工访问，又适用于应用程序代码访问。这种策略根据数据类型以及与之相关的风险进行分类，以便进行更恰当的控制和跟踪。

出于合规性原因进行分片

强制将特定的数据集分离到专用集群的一个原因是，不同的控制机制有不同的合规性要求。假设你是一家营销传播供应商，正在开发一种新的、独立的产品，重点放在医疗保健技术上。目前使用的数据与健康无关，不受许多法律要求的约束。一旦企业进入健康技术领域，你就拥有了一个客户子集及其数据，在这个子集中，你的公司承担了作为个人健康信息（PHI）处理机构的法律责任。在这种情况下，从一开始就在专用的数据存储中开发新产品是有意义的，这样就可以更恰当地应用HIPAA合规性控制，而不会给现有的数据集及其相关的应用程序增加不必要的负担。

独立的数据库用户

随着产品变得越来越复杂，技术栈也随之增加，你将开始拥有多个具有数据访问的应用程序。尽早开始使用良好的数据访问控制机制非常重要，多个应用之间不要共享相同的数据库访问凭据。无论公司规模大小，都会发生安全事件和凭证意外泄露事件，当这些事件发生时，你需要紧急轮换密钥。如果泄露的密钥对业务运营的影响是已知的，并且影响范围是孤立的，那么你可以更好地处理这类事件。

跟踪变更

很多合规性法规都附带了跟踪变更的控制措施：对影响财务报告的数据子集的更改、对生成发票的系统的更改，以及如何审查、测试和跟踪这些更改。与这种合规性控制相关的重要的地方之一就是数据库。当你为一家合规性责任不断扩大的企业工作时，像“如何在生产环境中审查、应用和跟踪schema更改”这样的过程需要更严格和更有计划性，而不仅仅是简单地记录一句“工程师登录生产环境源节点并进行更改”。如果你与内部审计团队或合规团队一起规划如何处理审计，这将大大减轻你的负担。

这里的目标是避免在年度审计时，各个团队争相为审计团队收集证据时手忙脚乱。相反，如果你对这些正常业务操作的工作方式进行了一些改进，那么就可以更容易获得和收集用于审计的“自带的”证据。在本节中，你将看到一个反复出现的主题，即从所有内容生成结构化日志。这是有意为之的。这有助于以相同的方式跟踪各种变更，而且这种一致性有助于业务以更少的成本实现其审计需求。可以在图13-1中看到这一切是如何组合在一起的。

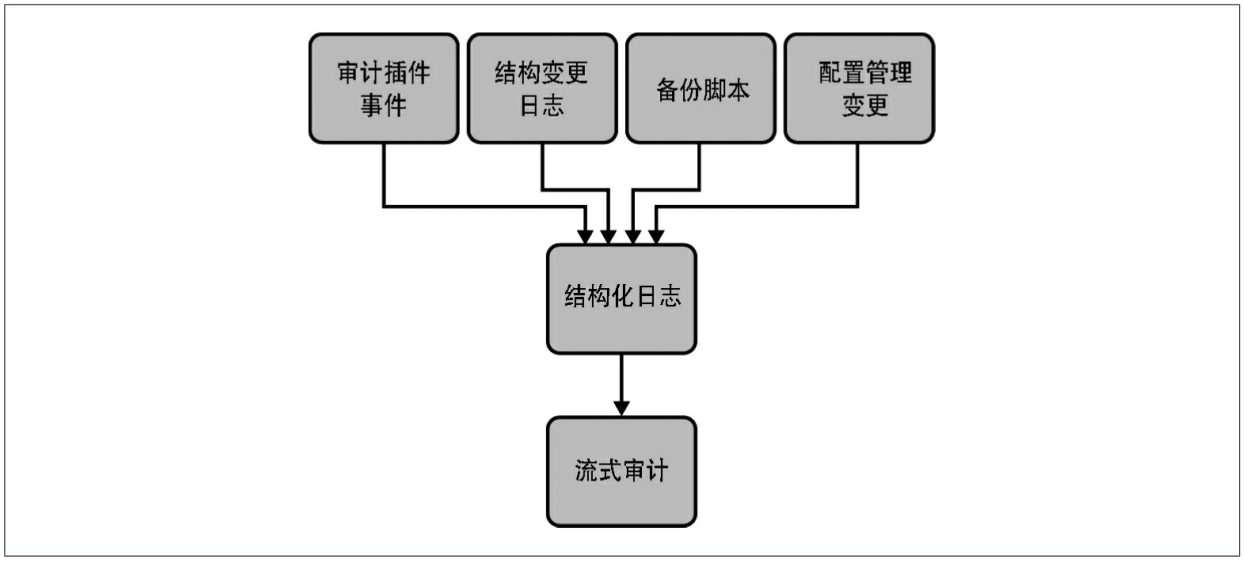

图13-1：不同操作任务将结构化日志发送到一个地方，从而使审核更容易的示例

让我们来看一下数据库系统的不同类型变更，以及如何为它们自动执行合规性跟踪。

数据访问日志

很多合规性控制都要求维护特定数据集的变更日志或访问日志。这可以用于跟踪财务数据的变化，也可以用于PCI或HIPAA等法规，这些法规中的数据足够敏感，需要跟踪所有访问。

你可以直接在数据库级别解决这一需求，如果运行的是Percona发行的MySQL分支，则可以使用Percona的审计日志插件（参见链接61）；如果运行的是MySQL的社区版本，可以使用等效的MySQL企业审计插件（参见链接62）。这样做的好处是，可以在数据更改之前的最后一跳来跟踪变更，尤其是在数据库变更可以通过多个路径进行的环境中。

跟踪更改不可取的选项。你可能会问，“为什么不使用触发器来跟踪我关心的表的更改？”这确实是我们过去见过的方法，但不推荐使用。以下是不建议使用触发器的一些原因：

- 众所周知，触发器会导致写性能下降，这会在最糟糕的时候对性能造成影响。
- 触发器相当于在数据库中存储业务逻辑，这是不推荐的。
- 在数据库中存储代码可能会绕过测试、转移和部署代码的任何过程。触发器可能意外地导致故障。
- 触发器只能支持跟踪写入操作。无法扩展成可以跟踪读取访问的解决方案。

让我们来看看如何使用Percona审计日志插件，以及如何对其进行调优。

安装和优化Percona审计日志。Percona审计日志插件是Percona发行的MySQL分支的一部分，但是默认情况下没有安装或启用。你可以在新实例的引导过程中，在实例上运行以下命令来安装：

```
INSTALL PLUGIN audit_log SONAME 'audit_log.so';
SHOW PLUGINS;
```

第二个命令列出了正在运行的插件，这里应该确认审计日志插件已经作为服务器进程的一部分运行。不仅要开启审计日志插件，还需要确定如何接收其输出。这才是真正重要规划的地方。

Percona的审计日志插件允许定义需要跟踪的语句谓词。这样可以灵活地满足各种控制要求，而不必在审计日志中记录大量无关的干扰信息。请确保查看其文档，以便根据需要正确配置该变量。

这个插件有一个灵活的优点，就是可以安装但不实际监控查询。[2]如果你还在研究如何接收插件的输出，并且需要在不影响正常运行时间的情况下关闭和打开，那么这将非常有用。但这种灵活性也带来了复杂性。除了管理审计日志插件附带的配置变量外，还需要监控其是否一直在运行。如果这是业务的关键功能，就意味着值得监控。由于可以在不重启服务器的情况下动态禁用插件，因此你需要做的不仅仅是检查磁盘上的*my.cnf*文件以确认它正在执行需要的操作，最好的办法是使用shell查询来解析插件的当前状态，并确认它实际上是在监控查询。可以分别执行下面两条单行shell命令：

```
# 检查审计日志插件是否处于活动状态的单行程序
$ mysql -e "show plugins" | grep -w audit_log | grep -iw active

# 检查插件策略是否实际监控查询的单行程序
$ mysqladmin variables | grep -w audit_log_policy | grep -iw queries
```

这里假设你只监控查询。如果还希望使用插件跟踪登录，则需要编辑这些用于检查的shell语句。

接收和使用审计日志。正如你所见，审计日志插件有很大的灵活性，但它也只能生成审计事件。你需要确定接收这些日志的最佳方式，将它们放在易于搜索和分析的地方，并合理地发现其中的异常，而不会让其成为主要负担。该插件可以简单地将输出转储到本地文件，但这可能会增加宕机的风险，因为日志文件可能会填满数据库主机中的磁盘。

一个更复杂的选项是使用插件的能力将输出发送到rsyslog（一种常见的UNIX日志管理工具），然后使用rsyslog将所有这些事件转发到组织所选择的结构化日志平台。这个选项很

有吸引力，因为它将这些数据带到了组织已经进行结构化日志存储的地方，这降低了数据库团队之外的利益相关者查看、搜索和分析这些事件的障碍。但是请记住，以这种方式使用rsyslog进行日志转发需要熟悉它的工作原理，确保以一种有意的方式决定和记录此数据流的rsyslog配置方式。[3]rsyslog中可能有许多默认配置对你想要的结果没有帮助，应该尽最大努力找到这些配置并进行相应的更改。

由插件输出的审计日志，要确保有很好的文档描述（即使是临时存储）。如果这些文件的传输方法变慢，那么在插件中缓冲事件可能会影响数据库服务器本身的性能。这种故障状态很难调试，因为它的唯一症状是查询执行速度变慢。考虑到可恢复性，应该计划对这些日志的整个管道进行混沌测试。

Percona审计日志插件是一个功能强大的工具，可以帮助实现很多合规性控制。根据我们的经验，这是一个比使用触发器性能更好的解决方案，并且它与配置管理和结构化日志软件集成得很好，使得解决方案在许多利益相关者团队中都是有效的。

schema变更的版本控制

第6章中介绍了不同的策略和工具，有助于大规模地进行schema变更。现在让我们看看这些策略带来的合规性问题。

使用版本控制来跟踪和运行schema变更可以附带内置的跟踪功能，包括谁请求了更改、谁审查和批准了更改，以及它在生产环境中是如何运行的。这也是为每个数据库集群的变更使用单独代码存储库的一个很好的原因。随着公司中数据库量级的增加，你会发现并不是所有的数据库都是相同的。有些数据库将需要遵循更严格的法规（例如，保存财务数据的数据库），而有些实验性数据库并不那么重要。如果使用了单独代码存储库，在进行审计时为每个数据集和集群提供变更记录将会非常方便。

这种基于合规性需求的数据和集群schema管理的分离，使得在版本控制管理中更容易控制谁可以提交或批准schema变更。在进行业务审计时，通常需要确定谁可以更改数据库。减少可以修改数据的人工操作员的人数，符合最小权限安全原则。

数据库用户管理

对数据库的更改不限于schema变更。还需要以一种可跟踪和可重复的方式管理数据库用户及其细粒度权限。让我们看看如何满足一些用于处理数据库访问控制的常见合规性控制。

使用配置管理。一种简单的使数据库用户跟踪合规的方式是，使用和数据库配置变更合规性相同的过程。你可以在配置管理的代码存储库中管理这一切，并使用源代码控制、Pull Request流程和同行评审来提供证据，证明对数据库用户的所有更改都是以一种可以审计和跟踪的方式完成的。

按计划轮换凭据。无论是针对计划外的安全事件，还是因为有一个按计划轮换凭据的控制规则，你都需要制订一个计划，以便在不影响应用程序在线时间的情况下轮换数据库用户。这可能意味着要更改应用程序使用的用户名和密码字符串。如果尚未运行具有双密码支持的最新的数据库版本，那么你应该采取以下步骤，在不影响服务在线时间的情况下，在生产应用程序中轮换数据库凭据：

1. 首先在数据库中引入新的用户名/密码对。
2. 测试新凭据是否具有与旧凭据相同的访问权限。在理想情况下，可以在部署新凭据时自动通过比较SHOW GRANTS来确认是否有相同的权限。
3. 部署应用程序，替换原应用程序配置中的凭据。
4. 重新启动此服务的所有实例，以确保新的用户名/密码对可以使用。
5. 删除旧的用户名/密码对。

无论是例行变更还是由于凭证泄露或安全风险而造成的紧急变更，此过程都应该相同。因为后者不是一个可以控制或可以完全避免发生的事件，因此最好让这个过程自动化，或者至少在运行手册中有很好的记录并可以定制演练，这样当意外发生时，对你的团队来说，才不会是一次可怕的消防演习。

在不影响可用性的情况下，在MySQL中轮换数据库的用户密码曾经是一项复杂的协调工作。在MySQL 8.0.14引入了双密码支持，再结合密码过期策略，使得轮换密码更加容易操作。

删除未使用的数据库用户。任何未使用的、在实例上保持激活状态的数据库用户都会带来额外的安全负担。定期审计实例上激活的数据库用户（与应用程序上配置的用户相比），并删除任何应用程序不再使用的用户，这一点很重要。

在满足公司的合规性需求时，你会发现许多合规性控制会要求公司跟踪某些资产的一切更改。这些控制在SOC 2等报告中是典型的，我们在本章前面介绍了SOC 2，其中主要关注的是提供数据完整性和安全性的证据。

有几种方法可以查明是否在使用一个指定的数据库用户。我们在第3章中详细介绍了Performance Schema，是将其作为一种检查服务器性能的方法进行介绍的。Performance Schema中有一张用户表，其中存储了有关已连接到服务器的用户的历史信息。这种历史跟踪可以追溯到服务器进程的生命周期或该表允许的最大大小（以最先发生的为准）。由于该表跟踪已经连接的用户，而不是没有连接的用户，因此你需要遍历已知用户，看看是否有用户没有出现在该表中，可将这个结果作为他们可能不再被使用的信号。

以下是在Performance Schema中启用该工具的查询：

```
mysql> UPDATE performance_schema.setup_instruments
    -> SET ENABLED='YES' WHERE NAME='memory/sql/user_conn';
Query OK, 1 rows affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

启用该功能后，可以通过Performance_schema.users表查找此信息。

如果你使用审计日志解决方案进行合规控制，例如，我们前面提到的Percona插件，那么可以使用这些日志来确定用户是否在给定的时间内连接过实例。

无论以何种方式确定这一点，都建议设置这样一个策略：“删除六个月内未连接过数据库的用户”。这一做法将有助于防止使用不需要的、现在已成为负担的访问。

数据库将处于需要这种级别的努力的控制范围内。随着公司逐渐成熟，并开始考虑变得更加合规，你将需要证明对生产数据库的变更在执行之前已经过审核和跟踪。合规控制关注的另一件事是，证明当灾难性事件发生时，你具有恢复数据和服务的能力。为此，接下来讨论备份和恢复，看看它们是如何符合合规要求的。

备份和恢复过程

第10章介绍了不同类型的备份。备份显然很重要。它们在发生事故时会非常有用，也是许多合规控制的关键部分。在大多数SOC 2实现中，都有创建和测试备份的控制要求（即使没有合规性要求，你也应该测试备份）。随着管理的数据库集群数量的增长，你很快就会发现无法继续手动执行备份和备份测试等过程，甚至无法通过手动读取日志文件来报告成功和失败。

从合规性角度，在评估如何管理备份时，需要满足以下一些要求：

- 需要将备份过程自动化。
- 如果备份失败，需要在备份过程中发出警报。
- 需要自动测试备份。
- 失败的备份测试也应该是可以在某处跟踪的事件。

接下来，我们将讨论如何安排备份和备份测试。

运行自动备份和备份测试

你需要一种机制来满足这些要求，该机制不仅可以运行调度作业（如Linux系统中的*crond*），而且可以确保作业按计划运行，且可以向监控系统和工单系统发送事件，以便在出现故障时发出警报，并跟踪故障以便以后进行审计。实现这一点的一种方法是，运行备份和备份测试作为监控检查[4]，但备份可能需要一段时间才能运行，尤其是当一些数据库实例的大小为TB级时。只要用于运行备份任务的监控系统能够处理比通常几秒长得多的检查，作为监控项运行备份就可以工作。因此，请确保运行监控系统的团队了解这个用例。

如果监控系统不能处理这种用例，那么请确保有一些方法可以留下“面包屑”，以跟踪备份以及备份测试已经发生并成功完成。这样的“面包屑”可以是一个包含时间戳的文件，备份程序在每次备份或备份测试运行结束时编辑该文件，作为任务发生和完成的证明。一旦“面包屑”就位，就可以使用监控系统更快地检查其存在。

在所有这些策略中，你想要的，以及SOC 2控制所需要的，是成功完成备份以及失败备份的跟踪记录，显示它们已被正确跟踪到工作项。

用于备份和备份测试的集中式日志

你还可能被要求展示成功完成备份和备份测试过程的日志。通过使用集中式日志解决方案，你可以将日志发送到该解决方案以保持日志连续性，从而为此类审计项目做好准备。请记住，我们为这些业务需求构建的解决方案应该假设服务器是易于重复替换而不是定制的。因此，如果你在下次审计之前让机器“退役”并更换，那么随机实例上的本地日志文件就不是理想的方案。你应该将任何与业务相关的资产（如备份过程的日志）都放在一个集中的位置，任何具有正确访问策略的人都可以访问。

通过备份进行灾难恢复规划

SOC 2还要求有适当的灾难恢复规划。这意味着可以证明你测试了系统生成的任何备份，跟踪了这些测试失败以及故障被恢复的时间，并且，在理想情况下，你可以知道数据灾难恢复需要多长时间。后面这一条需要跟踪备份测试所需的时间。在第2章提到过，数据库实例大小是一个衡量标准，用于确定在预定的目标时间内，数据库实例是否变得太大而无法进行备份恢复。让这成为一个自我改进的循环的方法是，让备份和测试备份的脚本也发送每个备份所需时间的度量。这样，你就有了每个数据库群集的备份时间以及恢复和测试这些备份所需的时间。现在，你有了一种方法来跟踪任何给定的数据集是否变得太大而无法满足业务对MTTR的期望。如果是，那么你就有了数据，可以对工作进行优先级划分，将数据集拆分为可接受的大小，或者重新检查关于数据恢复的SLA。

关于备份的最后一个重要注意事项是，需要确保你的安全利益相关者对当前数据库和备份的访问受控制规范管控。确保你的云服务商不会将存储备份文件的存储桶访问策略设置为默认允许访问。许多安全事件不是通过破坏当前的基础设施而发生的，而是通过从某个存储桶泄露的备份而发生的。

小结

合规是一个涉及政策和控制的广泛领域，对每种政策和控制的解释也很多。它不仅影响企业运行数据库的方式，还影响法律、财务和IT部门，甚至影响软件变更的部署方式。本章重点介绍了每种常见的合规性法规如何影响你作为数据库工程师的职责。然后，我们讨论了可能受到这些规则影响的不同实践和架构决策，你也需要考虑这些规则。

总的来说，解决与控制措施相关的难题的最佳方法是及早计划。分离应用程序用户，计划凭据轮换策略，并确保密码始终以加密方式而不是以明文形式存储。确保在需要开启数据库访问日志之前，有一个可以信任的日志接收管道。最后，你需要对schema变更进行控制和记录。

本章的目标并不是让你一下子就理解整个基础设施的所有控制，而是让你为每个法规范围内的零碎要求提供证据的任务变得更容易，并且尽可能自动化或简单地将它们组装起来。最终，这些控制措施旨在保护企业和客户的隐私。随着公司的发展和进入更广阔的市场，透彻理解每项控制的目标，将使这项重要任务对你和你的团队来说更易于管理。

本书由“满屋书香”整理，如果你不知道读什么书或者想获得更多免费电子书请加小编微信：sisijuan2012或QQ：151680600 小编也可以结交一些喜欢读书的朋友。或者关注小编个人微信公众号名称：满屋书香。 公众号中可以获得书香君所有分享的电子书！

---

[1] 有关这些标准的更多信息，请与你友好的信息安全团队开始对话，或从**O'Reilly**获取**NIST**网络安全框架（参见链接**60**）的袖珍指南。

[2] 更多信息请参阅参考文档（参见链接**63**）。

[3] 作为起点，这里有一整页关于**“**可靠的日志转发**”**的信息（参见链接**64**）。

[4] 在博客文章**“**为**DBA**任务使用**Sensu”**中（参见链接**65**），你可以看到一些将备份等任务作为数据库监控解决方案的一部分的示例。

附录A 升级MySQL

升级需要在稳定性[1]和功能之间进行权衡，在选择升级时应仔细考虑。使用MySQL最大的好处之一是它广泛的安装基础。这意味着可以从其他许多人测试和使用MySQL的经验中获益。如果升级到一个太新的版本，可能会在不知不觉中在你的环境中引入bug或引起性能退化。如果版本太旧，可能会遇到不明显的bug，或者无法利用优化了性能的新特性。

为什么要升级

进行版本升级是一个有风险的过程。它通常包括备份所有数据、测试更改，然后运行升级过程。在讨论细节之前，很重要的一点是了解升级的原因。

升级的原因有很多。

安全漏洞

: 随着时间的推移，这种可能性会越来越小，但人们仍然有可能在MySQL中发现安全漏洞。你或安全团队需要对此进行评估，并确定是否应该执行升级。

已知的bug

: 在生产环境中遇到未知或无法解释的行为时，我们建议你确定当前运行的MySQL版本，然后阅读后续版本到最新版本的发布说明。通过阅读相关文档，你完全有可能发现遇到的问题实际上是MySQL中的一个软件bug引起的。如果该bug在新版本中已经修复，那么可能需要升级MySQL。

新功能

: MySQL并不总是遵循严格的主要（major）/次要（minor）/点（point）的版本发布策略来添加功能。许多人可能会认为一个单点版本（例如，从8.0.21到8.0.22）只包含bug修复，而一个小的版本更改（从8.0到8.1）会包含一些小的功能。Oracle通常会在次要版本中发布新功能，这可能会对工作负载产生影响。这种策略是一把双刃剑，也是你应该在升级之前阅读所有发布说明的原因。

MySQL支持周期的终止

: Oracle为MySQL设置了生命周期终止（EOL，end-of-life）的时间框架。一般来说，建议保持在受支持的版本内，以便至少仍支持安全修复。

我们已经介绍了决定升级的各种原因，以及具体应该升级到哪个版本，接下来讨论一下规划和安全地完成升级的过程。

升级的生命周期

一旦决定升级，通常需要采取以下步骤：

1. 阅读该版本的发行说明，包括任何微小的更改。
2. 阅读官方文档中的升级说明。
3. 对新版本进行测试。
4. 最后执行升级。

版本发布说明通常包含重要的信息，如新特性、更改或已弃用的特性，以及已经修复的bug列表。升级说明提供了如何执行升级的详细叙述，并提醒你注意在继续之前需要了解的重要信息。

此外，你还应该制订一个计划，以便在出现问题时知道如何应对，比如查询的性能开始变差，或者更糟的是，遇到引发MySQL崩溃的bug。对于所有主要和次要的版本变更（例如，从8.0降级到5.7或从5.7降级到5.6），降级的唯一方法是恢复升级前的备份。这使得升级尤其危险，所以一定要有计划。

值得注意的是，自从MySQL 8.0以来，不能再降级点发布版本。例如，如果你运行的是8.0.25版本，除非导出所有数据并重新导入，否则不能再降级到8.0.24。

升级测试

阅读完发行说明和升级说明之后，你就应该清楚需要测试的关注点或重点领域了。下一步是测试这个新版本将如何处理工作负载。你还需要验证是否已查看了配置文件。较新版本的MySQL通常会重命名或完全弃用部分变量。

测试是一件需要细心和技巧的工作，每种方法都有其局限性。考虑到之前提到的降级风险，你应该在升级之前尽可能多地使用下面介绍的这些方法。

开发环境测试

希望你的数据库有一个开发环境。这是开始测试的好地方，无论是在共享的开发数据库中，还是在独立的数据库中。使用它的主要目的是暴露明显的语法问题。大多数开发环境不会包含和生产环境相同大小的数据，因此很难运行准确的测试。例如，你可以运行常用的查询，这可能没有任何问题，因为只访问表中的10行数据。而在生产环境中，同一个表中有1000万行，那时你就可能会发现性能退化了。

生产环境镜像

另一种选择是创建生产数据的副本，并向其发送SQL流量的副本。Etsy的*Code As Craft*博客上的一篇博文展示了这种方法（参见链接66）。简而言之，你可以生成生产数据库的一个副本，然后停止复制，并在副本上升级MySQL。完成后，使用*tcpdump*和*pt-query-digest*的组合将生产环境的流量发送到副本环境中。应用程序仍然只使用生产系统，而升级后的副本环境可以用于观察性能指标、发现语法错误。

副本

如果你的拓扑中有只读副本，并且可以控制只读节点的查询流量，那么可以考虑升级其中一个副本。这将允许你查看在实际生产工作负载下读取流量的执行情况。如果观察到错误或性能下降，可以将查询流量从副本切走并进行调整。这种方法的缺点是不能测试写入流量及其性能。

工具

Percona Toolkit提供的工具*pt-upgrade*，可以接受将查询作为输入，在两个不同的目标库中运行查询，最后生成一个报告，展示行数统计、行数据或错误的任何差异。因为可以接受

许多不同类型的输入（慢查询日志、通用查询日志、二进制日志），因此是获得额外测试覆盖率的一个很好的选择。

最好的使用方法是首先使用慢速查询日志或二进制日志收集最关注的查询。然后配置两个相同的系统，将其中一个升级到新版本，然后对这两个系统运行*pt-upgrade*以查看差异。

大规模升级

升级MySQL非常简单，在MySQL官方文档中有详细介绍。简而言之，如果要进行就地升级，你只需停止MySQL，替换二进制文件，启动MySQL，然后运行mysql_upgrade脚本。

在数百台MySQL服务器上执行此操作是一件重复性的工作。我们的建议是尽可能实现自动化。一种方法是使用Ansible。

以下推荐一个执行安全升级的框架流程，可以将其作为构建Ansible剧本的指南。

1. 验证目标。

    要做的第一件事是防止意外地升级了生产系统。如果有一个系统可以查看数据库是否正在活跃地接收流量，那么可以在该系统中做检查。如果遵循了我们在第5章中的建议，你应该使用read_only标志来防止对副本的意外写入。如果没有一个可以检查的系统，read_only标志可以作为一个很好的选择。如果服务器是可写的，很可能不应该升级它，因为它可能正在进行生产写操作。你还可以在这个步骤验证尚未升级的服务器。这样可以多次运行升级脚本而不会重复升级同一台服务器。

2. 设置停机时间。

    如果你的系统配置了监控告警，下一步涉及设置某种形式的停机时间或警报抑制，避免在版本升级过程中重新启动MySQL时触发大量警报。

3. 其他的先决条件。

    如果你有任何其他依赖的服务，比如配置管理工具或其他监控工具，它们会在MySQL离线时产生错误，现在是时候关闭它们了。

4. 删除旧的包。

    首选的方法是完全删除MySQL安装的所有包。这有助于避免主要版本（5.7到8.0）的包冲突。

5. 安装新的包。

    接下来需要将新包安装到系统上。

6 启动mysqld。

    启动mysqld服务。

7. 运行mysql_upgrade。

    如果是早于MySQL 8.0的版本[2]，请运行mysql_upgrade程序。需要特别注意的是，如果像我们推荐的那样使用了super_read_only方式运行MySQL，那么在执行mysql_upgrade步骤前，你需要将其设置为OFF。

8. 重启mysqld。

    到了这一点，我们更愿意重新启动mysqld。这将确保它使用升级后的文件正确启动，并且配置文件也能正常工作。

9. 验证是否可以连接。

只需连接并运行SELECT 1即可确保MySQL正常运行。

10. 恢复所有已禁用的服务。

如果升级前关闭了任何配置管理或监控工具，请再次启用它们。

11. 清除停机时间。

让服务器脱离停机状态，以便观察升级过程是否有任何失败的地方。

通过这个过程，你可以到任何服务器上运行升级脚本，只升级未升级且没有实际流量的节点。

小结

升级MySQL的原因有很多，最主要的原因是修复遇到的bug，或者能够利用某个新特性。例如，MySQL 8.0为InnoDB引入了一个特性，可以立即添加列而不需要重建整个表。这种类型的特性增强可以为需要执行大量ALTER TABLE..ADD COLUMN的公司节省大量的时间。虽然在安全升级过程中需要付出很多努力，但在执行这些列添加语句时能节省大量时间，同时还能提升开发人员的体验。

然而，大版本的升级可能会让人望而却步。应该投入大量精力来测试升级是否有任何负面影响。通常，你需要检查升级是否会导致任何查询响应时间出现偏差或引发新的报错。一旦你有了信心，可以慢慢地展开工作，并且需要准备好回滚过程。

最后，如果你有大量的服务器需要管理，请考虑在自动化过程上加大投入。与直接登录到每台服务器相比，自动化可以使升级过程易于重复，并且效率更高，而且因出现拼写错误或者到错误的服务器上执行升级而导致意外停机的概率也更低。

---

[1] **MySQL**社区的长期成员**Stewart Smith**创造了著名的**dot-20**规则：**“[**这个规则**]**是，一款软件在**dot-20**版本发布之前永远不会真正成熟。**”**虽然这不是一条硬性规定，但它确实强调了新版本和稳定性之间的权衡。

[2] 在**MySQL 8.0**版本之后，已将**mysql_upgrade**过程转移到服务器本身的启动中。无须将此作为附加步骤运行。

附录B Kubernetes上的MySQL

如果过去五年里你一直在技术领域工作，应该听说过Kubernetes、与运行Kubernetes的团队合作过、看过很多关于Kubernetes的会议演讲，或者读过很多解释Kubernetes的博客文章。如果你的组织在运行Kubernetes集群，你会在某个时候被问及在Kubernetes上运行MySQL是否也是一个好主意。从表面上看，这似乎是一条合理的路经。管理大量Kubernetes集群是一项复杂的任务，通常需要专门的团队投入。对组织来说，投入这些专业团队，考虑不仅仅在Kubernetes集群中运行无状态的工作负载是合理的。我们有充分的理由在Kubernetes上运行MySQL，也有充分的理由不这样做。让我们在这里揭开在Kubernetes上运行MySQL的一些FUD（恐惧、不确定性、怀疑）的神秘面纱。

使用**Kubernetes**调配资源

在Kubernetes技术流行之前，许多公司要么完全定制技术栈来供应和管理虚拟机和物理服务器，要么只完成了将资源生命周期管理一小部分的开源项目黏合在一起。后来，Kubernetes作为一个管理计算和存储资源的更完整的生态系统出现了，将其用作资源调配系统来管理所有资源也变得越来越有吸引力。然而，像MySQL这样的有状态负载仍然落后，没有带来附加价值，因为人们普遍认为“不能在容器上运行数据库”。

仔细规划你的目标

需要重点记住的是，我们使用Kubernetes想要获得哪些具体的价值。Kubernetes运行无状态负载的能力非常强大，因为它带来了计算资源的弹性和效率。然而，在考虑统一资源调配系统时，我们有理由将目标范围缩小到“我们只想使用Kubernetes调配和配置数据库资源系统”。这意味着需要预先明确，应该将Kubernetes提供的数据库工作负载与无状态工作负载分开管理，需要不同的操作技能，并将以不同的方式处理容器故障。

选择控制平面

Kubernetes有各种各样的MySQL operator，但是选择哪一个是最好的将主要取决于Kubernetes对MySQL的管理范围。你是否需要一个operator来完成所有的工作：供应、故障转移和管理到数据库的连接？或者只是简单地使用Kubernetes分配资源，并使用其他方法来管理服务中的数据库？尽早决定你对控制平面的期望，因为这决定许多操作细节。

更多细节

一旦决定开始使用Kubernetes配置MySQL资源，你需要在整个组织内就此解决方案支持的数据大小达成一致。请记住，这是一种运行关系数据库的新的操作模型，这不是一条平坦的道路，随着规模的扩大，一切都会变得更加复杂。当你与Kubernetes工程团队（希望你有一个专门的团队）就如何支持有状态的工作负载合作时，下面是一些需要重点考虑的事情：

- 需要支持的单个数据库实例的最大数据集大小是多少？
- 是否将把卷挂载到容器中，并将容器恢复与数据挂载分开管理？或者数据是容器的一部分？
- 支持的最大查询吞吐量是多少？将如何管理资源？

- 如何确保运行数据库工作负载的Kubernetes节点专用于数据库工作负载，而不是与无状态、更有弹性的工作负载共享？
- 将使用哪个控制平面来运行数据库实例？它是Kubernetes原生的吗？
- 备份如何工作？恢复过程是怎样的？
- 如何控制和安全地执行配置更改和MySQL升级？
- 如何在不造成中断的情况下升级Kubernetes集群？

与你的合作伙伴Kubernetes工程团队就该解决方案的工作方式达成一致，这将大大有助于为希望使用该解决方案的功能开发团队建立完善的SLO，并正确地沟通它解决了什么问题，以及团队需要自行解决的问题是什么。

我们对于在Kubernetes上运行MySQL的建议是，投资学习一个已经在Kubernetes生态系统中被验证过的控制平面，比如Vitess。但在试图跑之前要先学会爬。MySQL不应该是在组织中Kubernetes上运行工作负载的第一个实验对象。在尝试运行更复杂的用例（如MySQL）之前，团队首先要通过运行无状态工作负载来学习其优点，证明其可行性。选取有状态应用的最佳初始用例时，从小型数据集群（磁盘上只有几GB的数据库）开始，用较少的关键任务数据集让你的团队、Kubernetes团队、功能开发团队熟悉在Kubernetes上运行有状态工作负载的新操作模型，这样可降低业务的风险。[1]

在Kubernetes上运行有状态工作负载的技术在过去的几年中已经很成熟了，并且在一些公司的关键贡献下，这些公司已经投入了大量的工程时间，使Kubernetes更有可能成为现实，但与直接在VM上运行相比，Kubernetes仍然处于起步阶段。你会发现，采用缓慢而谨慎的方法从长远来看是值得的。特别是考虑MySQL在Kubernetes上的失败模式，并问一下自己：如果一切都出了问题，如何将其恢复正常运行？会丢失数据吗？确保你是有答案的。

小结

Kubernetes是目前技术领域发展最快的基础设施平台之一，这是有原因的。它带来的工程师速度及云原生基金会所支持的丰富生态系统，使其成为对企业有吸引力的投资。但是你应该从团队和公司的风险和回报的角度来考虑是否在Kubernetes上运行MySQL。应确保对有状态服务（如数据存储）在组织的Kubernetes旅程中的位置有共同的理解。要想利用Kubernetes的现有投资来满足所有工作负载是可以理解的，但这需要与数据存储层的稳定性需求进行很好的平衡。

---

[1] 关于在**Kubernetes**上运行数据库工作负载的精彩**“**一线**”**会议，我们推荐**Alice Goldfuss**的**“The Container Operator's Manual”**（参见链接**67**）主题演讲。

关于作者

Silvia Botros是Twilio的软件架构师。在SendGrid任职期间，她帮助构建了支持发送数十亿封电子邮件的数据库平台，支持其他产品，并从零开始进行了数据存储设计。

Jeremy Tinley是Etsy的高级职员系统工程师，拥有超过20年的MySQL使用经验。在整个职业生涯中，他着眼于可用性、可靠性和操作效率，管理了成千上万个MySQL实例。

封面动物

本书的封面动物是一只雀鹰（Accipiter nisus），是欧亚大陆和北非地区隼科的一种小型林地成员。雀鹰有长尾巴和短翅膀；雄性身体为蓝灰色，胸部为浅棕色，雌性身体为棕灰色，胸部几乎完全为白色。雄性（11英寸）通常比雌性（15英寸）小一些。

雀鹰生活在针叶林中，以小型哺乳动物、昆虫和鸟类为食。它们通常在树上筑巢，有时也在悬崖上筑巢。初夏时，雌性在最高的树枝上筑巢，产下四到六枚白色的蛋，蛋上有红色和棕色斑点。雄性会喂养雌性及其幼崽。

像所有的鹰一样，雀鹰也能在空中高速飞行。无论是翱翔还是滑翔，雀鹰都有一种特有的拍打滑翔动作；它的大尾巴使其可以毫不费力地扭动和转身。

O'Reilly出版社出版的图书的封面上的许多动物都濒临灭绝，它们对世界很重要。

封面插图由Karen Montgomery创作，取材于Lydekker的*The Royal Natural History*中的一幅古董线条版画。

O'REILLY®

高性能MySQL（第4版）

如何释放MySQL的全部能量？通过《高性能MySQL》（第4版），你将学习到各种高级技术，包括设置服务器级别目标，设计schema、索引和查询，调整服务器、操作系统和硬件，以充分发挥平台的潜力。本书还向数据库管理员介绍了通过复制、负载均衡、高可用性和故障切换来扩展应用程序的安全且实用的方法。

《高性能MySQL》（第4版）旨在反映云和自托管MySQL的最新进展、InnoDB性能，以及新特性和新工具，可以帮助你设计一个可随业务扩展的关系数据平台。你将学习到数据库安全方面的最佳实践，以及在性能和稳定性方面来之不易的经验。

- **深入了解MySQL的体系结构，包括其存储引擎的关键事实。**
- **了解服务器配置如何与硬件和部署选择配合使用。**
- **让查询性能成为软件交付过程的一部分。**
- **检查MySQL复制和高可用性的增强功能。**
- **比较托管云环境中的不同MySQL产品。**
- **探索MySQL从应用端配置到服务器调优的全栈优化。**
- **将传统的数据库管理任务转变为自动化流程。**

“新版本将这本书的重点转向了现代、务实的通过团队成员传递商业价值的思维模式。它不再只关注内部结构和理论，转向了更全面的视角。本书仍然覆盖关于‘数据库如何工作’的知识，但现在是以一个全新的、人性化的观点呈现的，这是非常有必要的。”

—Baron Schwartz，
《高性能MySQL》第2版和第3版的主要作者

Silvia Botros是Twilio的软件架构师。在加入Twilio之前，她带领SendGrid的数据库基础设施团队对每年处理数千亿封电子邮件的系统进行大规模运维。

Jeremy Tinley是Etsy的高级系统工程师，拥有超过20年的MySQL操作经验。

图书分类：数据库

责任编辑：张春雨
封面设计：Karen Montgomery 张健

Broadview®
WWW.BROADVIEW.COM.CN

O'Reilly Media, Inc.授权电子工业出版社出版

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

ISBN 978-7-121-44257-5

定价：100.00元