华章科技

网易CEO丁磊隆重推荐：本书系统深入地讲解了MySQL数据库中SQL编程的各种方法、技巧和最佳实践，推荐DBA和开发人员参阅！

畅销书《MySQL技术内幕：InnoDB存储引擎》姊妹篇，揭示SQL编程的奥秘，演绎SQL编程之美

数据库技术丛书

Inside MySQL: SQL Programming

MySQL技术内幕

SQL编程

姜承尧◎著

机械工业出版社
China Machine Press

联合策划

MySQL技术内幕：SQL编程

Inside MySQL: SQL Programming

David是ChinaUnix社区资深的MySQL版主，几年来，在论坛里热心地为广大MySQL DBA和开发者解答了很多问题，非常受欢迎。David是MySQL领域的专家，不仅多年来一直在国内的大型企业从事与MySQL相关的工作，而且还为MySQL编写了许多开源工具和性能扩展补丁，对MySQL数据库的使用（管理与开发）和原理都非常精通。本书是David多年工作经验的结晶，系统深入地揭示了MySQL中SQL编程的方方面面，能为我们解决复杂的MySQL数据库问题提供很好的指导。强烈推荐！

——ChinaUnix社区 & ITpub社区

本书的内容是告诉你如何使用SQL语言指挥数据库这个“庞然大物”。人与人的沟通需要技巧，与数据库沟通更需要技巧，而一切沟通都需要先从熟悉语言开始。作者“沉迷”于数据库的世界多年，有天赋，也肯下苦功，是我见过的最优秀的DBA之一。他熟知SQL的各种语法技巧，结合日常工作中的经验和领悟，写出了这本“内幕”。相信我，不管你是需要一本SQL查询手册，还是想要获得更多关于SQL编程的经验技巧，本书都不可不读。

——**顾懿** 久游网首席运营官

我是几年前在某个技术社区中认识David的，记得他总能用深入原理性的阐述来解答网友的疑惑。从中可以看出他对MySQL尤其是InnoDB引擎的理解之深，他的上一本著作《MySQL技术内幕：InnoDB存储引擎》推出后广获好评，填补了业界系统化讲解InnoDB引擎原理的空白。而本书详细讲解了子查询、表联接、聚合、事务、索引等SQL编程各环节所涉及的内容，全面揭示了MySQL中SQL编程的奥妙，用事实证明了MySQL也是可以处理复杂SQL查询的，而非只是个玩具。在这里，我向开发人员、DBA以及所有对MySQL感兴趣的读者推荐本书，它将使你受益匪浅。相信它将再次掀起学习MySQL的热潮。

——**叶金荣** ChinaUnix社区 MySQL版主

无论是与MySQL数据库相关的开发工作，还是MySQL数据库的管理与维护，SQL语言都发挥着极为重要的作用。所以，对于数据管理人员（DBA）和开发人员来说，掌握SQL编程技巧是全面了解数据库系统的必备条件之一。本书结合MySQL的历史、辅助工具、内部工作机制等多角度详细讲解了SQL编程的方法，展示了SQL在MySQL数据库编程方面（尤其是性能方面）的高级技巧。对于想全面深入地学习SQL编程和MySQL数据库的读者而言，本书不可多得。

——**郭鹏** 《Cassandra实战》作者/专注于Hadoop&NoSQL技术研究

上架指导：计算机/数据库
ISBN 978-7-111-37764-1
9 787111 377641
定价：69.00元

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

数据库技术丛书

Inside MySQL: SQL Programming

MySQL技术内幕
SQL编程

姜承尧◎著

机械工业出版社
China Machine Press

本书是畅销书《MySQL 技术内幕：InnoDB 存储引擎》的姊妹篇，深刻揭示了 MySQL 中 SQL 编程的精髓与奥秘，能为开发者和 DBA 们利用 SQL 语言解决各种与开发和管理相关的 MySQL 难题提供很好的指导和帮助。

全书一共 10 章，全面探讨了 MySQL 中 SQL 编程的各种方法、技巧与最佳实践。第 1 章首先介绍了 SQL 编程的概念、数据库的应用类型以及 SQL 查询分析器，然后介绍了 SQL 编程的三个阶段，希望读者通过本书的学习能达到最后的融合阶段。第 2 章全面讲解了 MySQL 中的各种数据类型和与之相对应的各种编程问题。第 3 章深入探讨了逻辑查询与物理查询的原理与方法。第 4 章的主题是子查询，不仅讲解了各种常用的子查询方法及其优化，而且还讲解了 MariaDB 对子查询的优化。第 5 章首先详细地分析了 MySQL 中的各种联接及其内部的实现算法，以及 MariaDB 数据库中引入的 Hash Join，然后针对关于集合的各种操作给出了解决方案。第 6 章分享了聚合和旋转操作的方法与技巧，并对一些经典的常见问题给出了解决方案。第 7 章深入阐述了游标的使用，重点在于如何通过正确地使用游标来提高 SQL 编程的效率。第 8 章讲解了关于事务的各种编程技巧，同时对事务的分类进行了详细阐述。第 9 章详细分析了各种索引的内部实现，探讨了如何使用索引来提升查询效率。第 10 章介绍了分区的方法与技巧，阐明了如何通过分区来进行 SQL 编程。

无论你是开发人员还是 DBA，无论你是需要一本 SQL 查询手册还是希望系统深入地学习 SQL 编程，本书都会是不错的选择。

图书在版编目（CIP）数据

MySQL 技术内幕：SQL 编程 / 姜承尧著 . —北京：机械工业出版社，2012.4

ISBN 978-7-111-37764-1

I. M…　II. 姜…　III. 关系数据库－数据库管理系统，MySQL　IV. TP311.138

中国版本图书馆 CIP 数据核字（2012）第 043855 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）
责任编辑：佘　洁
北京京师印务有限公司印刷
2012 年 4 月第 1 版第 1 次印刷
186mm×240mm • 20.25 印张
标准书号：ISBN 978-7-111-37764-1
定价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991；88361066
购书热线：（010）68326294；88379649；68995259
投稿热线：（010）88379604
读者信箱：hzjsj@hzbook.com

序

最初的 MySQL，只是一个底层的面向报表的存储引擎。历经若干年发展之后，今天的 MySQL 已成为世界上最流行的开源数据库。由于具备高性能、高可靠性和易用性等优点，它备受全球互联网企业的青睐。如今 MySQL 已在超过 20 个平台上运行，提供全方位的数据库支持、培训和咨询服务。

自我认识 David（本书作者）以来，他就致力于 MySQL 底层原理的研究，当大多数人还在苦恼于如何编写 SQL 语句时，他已经在剖析 MySQL 内部的运作机理。也许，数据库呈现在众人眼前的还是一个黑匣子，而在 David 眼里就犹如一块由数百齿轮结合在一起精妙运转的怀表。当得知他的下本书将介绍 SQL 编程时，我不禁欣喜若狂。正如 David 所言，SQL 是一门语言，也有其特有的语言艺术，只有深层次地挖掘语言内部的含义才能将想执行的操作描述得简洁高效。当我阅读此书并跟着作者的思路层层解析 SQL 编程时，经常会感叹，在数据库的许多细小操作上竟有如此多令人意想不到的亮点。如果说作者的上一本书——《MySQL 技术内幕：InnoDB 存储引擎》是面向 OLTP 的精华教程，那么本书将是 OLAP 的高阶宝典，它从编程的角度诠释了数据库语言的魅力，这恰恰是我们在实际生产环境中需要的。

在平时的工作中，David 时常会有新颖的想法，记得在固态硬盘发布之初，其虽然有着优越的性能，但是高额的价格使开发者在大规模存储前对其望而止步。而 David 提出了使用固态硬盘做第二缓冲池（Secondary Buffer Pool）的设想，如今已被新浪投入生产使用，可见其独到的眼光。

《MySQL 技术内幕：SQL 编程》到底是怎样的一本书？在这本书中，David 将会为你揭示关于 MySQL 不为人知的奥秘。相信读完此书，你对 MySQL 的 SQL 编程认识将达到一个新的水平。

——鹿遥　火游网络科技有限公司数据库经理

前　言

为什么要写这本书

多年来，我一直在和各种不同的数据库打交道，见证了 MySQL 从一个小型的关系型数据库发展成为各大互联网企业的核心数据库系统的过程，期间参与了一些大大小小的项目开发工作，并成功地帮助开发人员构建了一些可靠的、健壮的应用程序。在这个过程中我积累了一些经验，正是这些不断累积的经验赋予了我灵感，于是有了本书。这本书实际上反映了这些年来我做了哪些事情，汇集了很多同行每天可能会遇到的一些问题，并给出了解决方案。

本书是“MySQL 技术内幕”系列的第二本书，我将其命名为“SQL 编程”而非“SQL 查询”，因为我想让更多的开发人员和 DBA 意识到 SQL 也是一门语言，与我们平时接触的 C 语言等编程语言并没有什么不同。正因如此，我们也要追求 SQL 的编程之美。

然而与其他语言不同的是，SQL 语言不仅是面向过程的语言，更多的时候，通过 SQL 语言提供的面向集合的思想可以解决数据库中遇到的很多问题。当然，SQL 语言本身也提供了面向过程的方法，但是如果使用不当，会在数据库性能方面遭遇梦魇。SQL 编程需要掌握的知识远比想象中多，只有掌握各种知识，综合运用面向过程和面向集合的思想，才能真正解决所遇到的问题。不要迷信网上的任何“神话”，不要被自己或他人的经验所左右。我一直坚信，只有理解了数据库内部运行的原理，才能承自然之道，“乘天地之正，而御六气之辩”，做到真正的“无招胜有招”。

另一方面，MySQL 数据库目前大多被用于互联网的联机事务处理应用中，给大部分用户造成 MySQL 数据库不能执行复杂 SQL 查询的错觉。本书将列举各种复杂的查询，使用户了解 MySQL 数据库处理复杂查询的执行过程。此外，由于 MySQL 数据库的不断发展，其

分支版本已经开始支持 Hash Join。相信随着时间的推移以及 MySQL 数据库本身的不断发展，MySQL 数据库同样会在联机分析处理应用中占有一席之地。大家需要做好这方面的准备，这也是本书将提供给你的。

最后，希望这本书可以引领开发人员及 DBA 从不同的角度来看待 SQL 语言和数据库的开发工作。倘若这本书能解决你在实际生产环境中遇到的问题，我会非常荣幸。

读者对象

- 数据库管理员
- 数据库开发人员
- 数据库架构设计师
- 各类应用程序开发人员

如何阅读本书

书中的示例一共用到三个数据库文件：一个是 employees 数据库，该数据库是 MySQL 数据库官方提供的示例数据库，主要用来模拟公司员工的数据，用户可以通过官网下载（http://dev.mysql.com/doc/index-other.html）；另一个是 dbt3 数据库，是通过 Database Test Suite 程序生成的，该数据库较大，主要用来展示一些复杂的查询；还有一个 tpcc 数据库，是一个模拟 TPC-C 测试的数据库，用户可以从 http://code.google.com/p/david-mysql-tools/ 下载。

本书一共有 10 章，每章都像一本迷你书，可以单独成册。用户可以有选择地阅读，但是推荐根据本书的组织方式进行阅读，这样会更具有条理性。

第 1 章　SQL 编程

主要介绍了 MySQL 数据库的发展历史和什么是 SQL 编程。希望读者能通过该章了解 MySQL 的深厚历史背景，并且知道它已经不再是一个小型关系型数据库系统。此外，还重点强调了 SQL 编程的三个阶段，希望读者可以通过本书的学习达到最后的融合阶段。

第 2 章　数据类型

详细介绍了 MySQL 数据库中的各种数据类型和与之相应的各种 SQL 编程问题。数据类型是 SQL 编程的基石。每一位 MySQL 数据库应用的开发人员都应该好好阅读本章。

第 3 章　查询处理

深入探讨了逻辑查询与物理查询。逻辑查询帮助读者理解数据库应该得到怎样的结果；物理查询是 MySQL 数据库通过分析表的结构，选择最小成本的执行计划来处理 SQL 语句，但是无论怎样，最终的结果应该和逻辑查询一样。

第 4 章　子查询

子查询是被很多开发人员和 DBA 诟病的一个方面。如何正确地理解 MySQL 子查询的执行方式并实现对其优化，是本章最重要的任务。本章的最后还讲解了 MySQL 分支版本 MariaDB 对子查询的优化。在 MariaDB 数据库中，子查询再也不是什么难题了。

第 5 章　联接与集合操作

联接和集合是关系数据库中常见的操作，该章详细而深入地介绍了 MySQL 数据库中的各种联接及其内部的实现算法，同时也介绍了 MariaDB 数据库中引入的 Hash Join，弥补了 MySQL 数据库在 OLAP 应用中的短板。本章最后针对集合的各种操作给出了解决方案。

第 6 章　聚合和旋转操作

聚合与旋转在报表系统中非常常见，本章主要讲解了 MySQL 数据库对上述两种操作的处理方法，以及一些常见问题的解决方案。

第 7 章　游标

游标是面向过程的编程方式，这与前几章介绍的面向集合的编程方式不同。虽然在大多数情况下游标处理的性能较低，但是只要在正确的场合使用，游标也会使 SQL 编程的效率得到极大提升。

第 8 章　事务编程

全面讲解了 MySQL 数据库中关于事务的各种编程技巧，同时也对事物的分类进行了详细的介绍。本章主要面向以 InnoDB 存储引擎为核心的应用编程。

第 9 章　索引

一般来说，索引可以提高 SQL 语句的执行速度，但是并非所有情况都是如此。本章详细分析了各种索引的内部实现，以及哪种情况下使用索引可以带来效率的提升。这对 SQL 编程来说非常重要。

第 10 章　分区

分区是设计表时需要考虑的重要问题之一。正确和有效地分区会对 SQL 编程带来巨大的影响。本章告诉读者应该如何分区，以及如何通过分区来进行 SQL 编程。

勘误和支持

由于作者的水平有限，编写时间仓促，书中难免会出现一些错误或不准确的地方，恳请读者批评指正。为此，特意创建了一个在线支持与应急方案的微博：http://weibo.com/insidemysql。你可以将书中的错误以及遇到的任何问题在此微博与我交流，我将尽力在线上为你提供最满意的解答。如果你有更多的宝贵意见，也欢迎发送邮件至邮箱 jiangchengyao@gmail.com，期待能够得到你的真挚反馈。

致谢

感谢我的老板丁磊（网易 CEO）在工作上给予我的指导和帮助，同时还要感谢他支持和鼓励我利用业余时间完成这本书的写作，这也充分体现了网易开放、分享的企业文化。

感谢网易研究院的所有同事们，能与一群才华出众的人一起工作让我感到非常荣幸与自豪，同时通过不断地与他人交流，使我个人在数据库方面得到了极大的提升并有所领悟。

感谢机械工业出版社华章公司的编辑杨福川和姜影，他们在这一年多的时间中始终支持我的写作，正是他们的鼓励和帮助引导我顺利完成全部书稿。

谨以此书献给我最亲爱的家人，以及众多热爱 MySQL 数据库的朋友们！

姜承尧（David Jiang）

2012 年 3 月于中国杭州

目　　录

第 1 章

SQL 编程

SQL 是一种编程语言，用来解决关系数据库中的相关问题。SQL 编程就是通过 SQL 语句来解决特定问题的一种编程方式。本章将介绍 MySQL 数据库中与 SQL 编程相关的基础知识，帮助 SQL 用户更好地理解 MySQL 数据库、数据库应用类型，以及与 SQL 编程相关的特定问题。

1.1 MySQL 数据库

1.1.1 MySQL 数据库历史

毫无疑问，目前 MySQL 已经成为最为流行的开源关系数据库系统，并且一步一步地占领了原有商业数据库的市场。可以看到 Google、Facebook、Yahoo、网易、久游等大公司都在使用 MySQL 数据库，甚至将其作为核心应用的数据库系统。而 MySQL 数据库也不再仅仅应用于 Web 项目，其扮演的角色更为丰富。在网络游戏领域中，大部分的后台数据库都采用 MySQL 数据库，如大家比较熟悉的劲舞团、魔兽世界、Second Life 等。很少能看到有哪个网络游戏数据库不是采用 MySQL 数据库的。此外，MySQL 数据库已成功应用于中国外汇交易中心、中国移动、国家电网等许多项目中。越来越多的企业级项目应用“见证”了 MySQL 数据库的飞速发展，并预示着 MySQL 数据库本身正在逐渐完善并走向成熟。以前会有人诟病 MySQL 为什么没有视图，没有存储过程，没有触发器，没有事件调度器。而现在，MySQL 还没有什么呢？经历了 MySQL 5.0 和 5.1 的发展，如今 MySQL 数据库迎来了重要的 5.5 版本。在了解 MySQL 5.5 带给我们的新特性之前，我们先来看看 MySQL 的发展历程。简单来说，MySQL 数据库的发展可以概括为三个阶段：

- 初期开源数据库阶段。
- Sun MySQL 阶段。
- Oracle MySQL 阶段。

很多人以为 MySQL 是最近 15 年内才出现的数据库，其实 MySQL 数据库的历史可以追溯到 1979 年，那时 Bill Gates 退学没多久，微软公司也才刚刚起步，而 Larry 的 Oracle 公司也才成立不久。那时有一个天才程序员 Monty Widenius 为一个名为 TcX 的小公司打工，并且用 BASIC 设计了一个报表工具，使其可以在 4MHz 主频和 16KB 内存的计算机上运行。没过多久，Monty 又将此工具用 C 语言进行了重写并移植到了 UNIX 平台。当时，这只是一个很底层的且仅面向报表的存储引擎，名叫 Unireg。

虽然 TcX 这个小公司资源有限，但 Monty 天赋极高，面对资源有限的不利条件，反而更能发挥他的潜能。Monty 总是力图写出最高效的代码，并因此养成了习惯。与 Monty 在一起的还有一些别的同事，很少有人能坚持把那些代码持续写到 20 年后，而 Monty 却做到了。

1990 年，TcX 公司的客户中开始有人要求为他的 API 提供 SQL 支持。当时有人提议直

接使用商用数据库，但是 Monty 觉得商用数据库的速度难以令人满意。于是，他直接借助于 mSQL 的代码，将它集成到自己的存储引擎中。令人失望的是，效果并不太令人满意，于是，Monty 雄心大起，决心自己重写一个 SQL 支持。

1996 年，MySQL 1.0 发布，它只面向一小拨人，相当于内部发布。到了 1996 年 10 月，MySQL 3.11.1 发布（MySQL 没有 2.x 版本），最开始只提供 Solaris 下的二进制版本。一个月后，Linux 版本出现了。

在接下来的两年里，MySQL 被依次移植到各个平台。在发布时，MySQL 数据库采用的许可策略有些与众不同：允许免费使用，但是不能将 MySQL 与自己的产品绑定在一起发布。如果想一起发布，就必须使用特殊许可，意味着要花“银子”。当然，商业支持也是需要花“银子”的。其他方面，随用户怎么用都可以。这种特殊许可为 MySQL 带来了一些收入，从而为它的持续发展打下了良好的基础。

MySQL 关系型数据库于 1998 年 1 月发行第一个版本。它使用系统核心的多线程机制提供完全的多线程运行模式，并提供了面向 C、C++、Eiffel、Java、Perl、PHP、Python 及 Tcl 等编程语言的编程接口（API），支持多种字段类型，并且提供了完整的操作符支持。

1999 ～ 2000 年，MySQL AB 公司在瑞典成立。Monty 雇了几个人与 Sleepycat 合作，开发出了 Berkeley DB 引擎，因为 BDB 支持事务处理，所以 MySQL 从此开始支持事务处理了。

2000 年 4 月，MySQL 对旧的存储引擎 ISAM 进行了整理，将其命名为 MyISAM。2001 年，Heikki Tuuri 向 MySQL 提出建议，希望能集成他的存储引擎 InnoDB，这个引擎同样支持事务处理，还支持行级锁。该引擎之后被证明是最为成功的 MySQL 事务存储引擎。

2003 年 12 月，MySQL 5.0 版本发布，提供了视图、存储过程等功能。

2008 年 1 月，MySQL AB 公司被 Sun 公司以 10 亿美金收购，MySQL 数据库进入 Sun 时代。在 Sun 时代，Sun 公司对其进行了大量的推广、优化、Bug 修复等工作。

2008 年 11 月，MySQL 5.1 发布，它提供了分区、事件管理，以及基于行的复制和基于磁盘的 NDB 集群系统，同时修复了大量的 Bug。

2009 年 4 月，Oracle 公司以 74 亿美元收购 Sun 公司，自此 MySQL 数据库进入 Oracle 时代，而其第三方的存储引擎 InnoDB 早在 2005 年就被 Oracle 公司收购。

2010 年 12 月，MySQL 5.5 发布，其主要新特性包括半同步的复制及对 SIGNAL/RESIGNAL 的异常处理功能的支持，最重要的是 InnoDB 存储引擎终于变为当前 MySQL 的默认存储引擎。MySQL 5.5 不是时隔两年后的一次简单的版本更新，而是加强了 MySQL 各个方面在企业级的特性。Oracle 公司同时也承诺 MySQL 5.5 和未来版本仍是采用 GPL 授权的开源产品。

随着 MySQL 的不断成熟及开放式的插件存储引擎架构的形成，越来越多的开发人员加入到 MySQL 存储引擎的开发中。而随着 InnoDB 存储引擎的不断完善，同时伴随着 LAMP 架构的崛起，在未来的数年中，MySQL 数据库仍将继续飞速发展。

1.1.2 MySQL 数据库的分支版本

MySQL 是开源的数据库，这意味着任何人都可以在其源码的基础上分支出自己的 MySQL 版本，并且可以在原 MySQL 数据库的基础上进行一定的修改，这是开源赋予用户的权力。

MariaDB 是由 MySQL 创始人之一 Monty 分支的一个版本。在 MySQL 数据库被 Oracle 公司收购后，Monty 担心 MySQL 数据库发展的未来，从而分支出一个版本。这个版本和其他分支有很大的不同，其默认使用崭新的 Maria 存储引擎，是原 MyISAM 存储引擎的升级版本。此外，其增加了对 Hash Join 的支持和对 Semi Join 的优化，使 MariaDB 在复杂的分析型 SQL 语句中较原版本的 MySQL 性能提高很多。另外，除了包含原有的一些存储引擎，如 InnoDB、Memory，还整合了 PBXT、FederatedX 存储引擎。不得不承认，MariaDB 数据库是目前 MySQL 分支版本中非常值得使用的一个版本，尤其是在 OLAP 的应用中，对 Hash Join 的支持和对 Semi Join 的优化可以大大提高 MySQL 数据库在这方面的查询性能。MariaDB 的官方网站为 http://mariadb.org/。

关于 MariaDB、MySQL、MaxDB 名字的由来，这里有个不得不说的小插曲。Monty 有一个女儿，名叫 My，因此他将自己开发的数据库命名为 MySQL。Monty 还有一个儿子，名为 Max，因此在 2003 年，SAP 公司与 MySQL 公司建立合作伙伴关系后，Monty 又将与 SAP 合作开发的数据库命名为 MaxDB。而现在的 MariaDB 中的 Maria 是 Monty 小孙女的名字。

Drizzle 是基于原 MySQL 6.0 代码分支出的一个版本，官方网站为 http://www.drizzle.org/。Drizzle 有个很明显的区别于 MySQL 的地方就是，它的核心代码很有限，目前也致力于继续保持微小内核的方式。Drizzle 支持一系列的接口，其他模块能很好地以插件方式加载进来，这样用户可以按照自己的需要进行扩展。同时，对于用户来说使用了什么模块更加一目了然，更加个性化。Drizzle 的特点为：

- 一个更适合云计算组件和 Web 应用的数据库。
- 专为多 CPU/ 多核 CPU 服务器在高并发情况下而设计。
- 高效的内存使用。
- 开放源代码、开源社区，开放型设计。

Percona Server 是 Percona 公司分支的一个 MySQL 数据库版本。该版本对高负载情况下的 InnoDB 存储引擎进行了一定的优化，为 DBA 提供一些非常有用的性能诊断工具，另外有更多的参数和命令可以用来控制服务器行为。Percona 公司最大的贡献是发布了免费开源的 XtraBackup 工具，可实现对 InnoDB 存储引擎表的在线热备份操作。

InnoSQL 是笔者分支的一个 MySQL 版本，其目标是提供更好的数据库性能，以及将一些富有创意的想法用于数据库的生产环境。InnoSQL 完全兼容于 Oracle MySQL 版本，所有添加的补丁、插件、存储引擎都是动态的。如果不开启这些功能，那么它和原版本是完全一

致的。目前其独有的功能有：

- InnoDB Flash Cache
- InnoDB Share Memory
- IO Statistics

InnoDB Flash Cache 将 SSD 作为 Flash Cache（之前版本的实现为 Secondary Buffer Pool）。目前一些解决方案如 Facebook Flash Cache 是通用的解决方案，Oracle 的 Flash Cache 性能较为一般。InnoSQL 的解决方案针对 MySQL 数据库的特性对 SSD 进行了大幅的优化，性能较直接将 SSD 作为持久存储性能可有 1 倍多的提升。

InnoDB Share Memory 将 Share Memory 作为 InnoDB 的共享内存，以此提高数据库的预热速度。预热对数据库的 benchmark 并没有多大的帮助，而对于生产环境中的使用却有着非常大的帮助。InnoDB Share Memory 可将 InnoDB 缓冲池迅速恢复到数据关闭时的状态，以此来保证应用的连续性。

IO Statistics 扩展了 MySQL 原有 Slow Log 的内容，现在可记录某 SQL 语句的逻辑读取和物理读取的 IO。这有助于 DBA 和开发人员更好地了解 SQL 语句的工作，同时帮助他们更好地进行 SQL 语句的调优。当打开 IO Statistics 时，会在 MySQL 的 Slow Log 中看到类似如下的内容：

```
# Time: 111227 16:29:54
# User@Host: root[root] @ localhost [::1]
# Query_time: 0.310018  Lock_time: 0.203012 Rows_sent:
1  Rows_examined: 30000 Logical_Reads: 30145 Physical_Reads: 50
use tpcc;
SET timestamp=1324974594;
SELECT COUNT(1) FROM history;
```

本书的一些地方会使用 InnoSQL 的 IO Statistics 功能，并结合原 MySQL 的 EXPLAIN 命令，以此来帮助用户更好地进行 SQL 编程。

1.2　SQL 编程

SQL 语言，同常见的编程语言 C、C++、Java、Python 一样，是一种编程语言。在每月由 Tiobe 公布的编程语言排行榜上可看到与 SQL 相关的语言上榜。同时，SQL 又是一种标准，每个数据库厂商都提供了对标准 SQL 的支持，此外 SQL 语言还扩展了每个数据库特有的 SQL 语法。

SQL 编程是指通过 SQL 语言来完成对于数据库的逻辑操作。这个逻辑操作可能比较简单，只需一个很简单的 SQL 语句来完成；这个逻辑也可能非常复杂，需要联接多张表或子查询等来完成；还有可能是这样的情况，即一条 SQL 语句并不能马上完成这个逻辑操作，需要

建立一个存储过程，通过封装在存储过程中的各种操作来最终得到结果。

由于工作的缘故，笔者的很大一部分时间是对开发人员进行数据库方面的沟通和培训。在这个过程中，笔者发现 SQL 编程已成为广大程序员日常工作中不可或缺的关键技术之一。学会 SQL 并不难，要成为优秀的 SQL 程序员却绝非易事。

大多数程序员都会接触各种不同的项目，每个项目所使用的数据库可能并不相同，因此，很多程序员往往惯性地认为数据库的 SQL 编程就是 SELECT、INSERT、UPDATE、DELETE 加上一些控制语句。同时，有些程序员在学习并从事了过程化或面向对象编程之后才转到 SQL 编程上，因此他们的 SQL 编程往往带有强烈的程序员特性（后面介绍的 SQL 编程第一阶段的特性）。在国内，通常很少有公司会聘请面向 SQL 开发的程序员，在国外却有 SQL 开发程序员的职位，工作内容是从事与 SQL 编程相关的开发工作。对于一些数据库方面的考试，如在 Oracle 的 OCP 考试分类中，特别区分了面向数据库的管理和面向数据库的开发。

而在国内，大部分 SQL 程序员就是 C、C++ 或者 Java 程序员兼的，他们既负责应用程序的开发，也包揽了 SQL 编程的工作，这样往往会导致在数据库中大量使用或者说滥用非关系模型的思想。这里以笔者在一家网游公司遇到的情况为例进行介绍。通常会因为各种原因需要对运营中的游戏大区进行合并，如将华东一区和华东二区合并，底层所做的操作就是将两个数据库中的数据进行合并，而两个数据库的表结构都是一样的。并区的脚本通常使用 SQL 来完成，当然复杂一点的可能需要借助一下脚本语言。但是不管怎么说，SQL 编程的好坏往往会影响并区时间长短。一个好的并区程序可以在四五个小时内完成几千万数据量的表，而有问题的 SQL 并区程序一个星期可能都不能完成任务。时间对一个网络游戏意味着很多，可能是玩家的回归，也可能是玩家最终的流失，更不用说那些在并区过程中收入的损失。

请原谅笔者在这里研究程序员的 SQL 习惯问题，对笔者来说 SQL 是一个具有重要哲学特征的领域。在这里，我把 SQL 编程分为三个阶段，当然不是每个人都必须同意笔者的观点。

第一阶段是面向过程化的 SQL 编程阶段。这是 SQL 程序员刚开始使用数据库的阶段，此时他们没有多少处理关系模型的经验和基于集合的思想。在这一阶段，经常会有滥用各种工具（如游标、临时表、动态 SQL 语句等）的情况，而程序员自己通常意识不到他们正在引起破坏。

第二阶段是面向集合的 SQL 编程阶段。这个阶段 SQL 程序员开始意识到 SQL 编程与面向过程和对象编程的不同之处，知道运用 SQL 编程需要更多的东西，慢慢发现 SQL 不再是妨碍编程的令人讨厌的东西，而是建立在基于关系模型集合理论的强大基础上的产物。从这一阶段开始，程序员开始相信那些说游标、临时表、动态 SQL 有害而永远不应该使用的“专家”。

第三阶段是融合的 SQL 编程阶段。这个阶段 SQL 程序员已经具有了丰富的知识并对 SQL 有了深入理解，他们对自己的代码非常自信，但是这并不意味着他们会停止钻研更深入

的知识以及提高关键性的技术。在这一阶段，SQL 程序员不再迷恋所谓的专家，他们可能意识到即使是游标，也并不是在所有情况下都是无用和有害的。

第三阶段的 SQL 程序员已经具备了判断什么时候使用纯静态的 SQL 编程方法不能完成某些任务的能力。尽管纯静态 SQL 编程是一种非常典型的方法，但是它只在大部分情况下适用。有时候，使用临时表可以显著地改善性能，使用动态 SQL 可以解决复杂的问题，适当地使用游标可以提高程序运行的效率，使用 C、C++ 这样的过程语言可以带来更大的灵活性，而且不会与关系模型发生冲突。

1.3　数据库的应用类型

对于 SQL 开发人员来说，必须先要了解进行 SQL 编程的对象类型，即要开发的数据库应用是哪种类型。一般来说，可将数据库的应用类型分为 OLTP（OnLine Transaction Processing，联机事务处理）和 OLAP（OnLine Analysis Processing，联机分析处理）两种。OLTP 是传统关系型数据库的主要应用，其主要面向基本的、日常的事务处理，例如银行交易。OLAP 是数据仓库系统的主要应用，支持复杂的分析操作，侧重决策支持，并且提供直观易懂的查询结果。

1.3.1　OLTP

OLTP 也被称为面向交易的处理系统，其基本特征是可以立即将顾客的原始数据传送到计算中心进行处理，并在很短的时间内给出处理结果，这个过程的最大优点是可以即时地处理输入的数据、及时地回答，因此 OLTP 又被称为实时系统 (Real Time System)。衡量 OLTP 系统的一个重要性能指标是系统性能，具体体现为实时响应时间 (Response Time)，即从用户在终端输入数据到计算机对这个请求做出回复所需的时间。OLTP 数据库旨在使事务应用程序仅完成对所需数据的写入，以便尽快处理单个事务。

OLTP 数据库通常具有以下特征：

- 支持大量并发用户定期添加和修改数据。
- 反映随时变化的单位状态，但不保存其历史记录。
- 包含大量数据，其中包括用于验证事务的大量数据。
- 具有复杂的结构。
- 可以进行优化以对事务活动做出响应。
- 提供用于支持单位日常运营的技术基础结构。
- 个别事务能够很快地完成，并且只需要访问相对较少的数据。OLTP 系统旨在处理同时输入的成百上千的事务。

1.3.2 OLAP

OLAP 的概念最早是由关系数据库之父 E.F.Codd 博士于 1993 年提出的，是一种用于组织大型商务数据库和支持商务智能的技术。OLAP 数据库分为一个或多个多维数据集，每个多维数据集都由多维数据集管理员组织和设计，以适应用户检索和分析数据的方式，从而更易于创建和使用所需的数据透视表和数据透视图。

OLAP 是共享多维信息的、针对特定问题的联机数据访问和分析的快速软件技术。它通过对信息的多种可能的观察形式进行快速、稳定一致和交互性的存取，允许管理决策人员对数据进行深入观察。决策数据是多维数据，是决策的主要内容。OLAP 专门用于支持复杂的分析操作，侧重对决策人员和高层管理人员的决策支持，可以根据分析人员的要求快速、灵活地进行大数据量的复杂查询处理，并且以一种直观易懂的形式将查询结果提供给决策人员，以便他们准确掌握企业（公司）的经营状况、了解对象的需求、制定正确的方案。

OLAP 具有灵活的分析功能、直观的数据操作和分析结果可视化表示等突出优点，从而使用户对基于大量复杂数据的分析变得轻松而高效，利于用户迅速做出正确判断。OLAP 可用于证实人们提出的复杂假设，是以图形或表格的形式来表示的对信息的总结。OLAP 并不将异常信息标记出来，采取的是一种知识证实的方法。

OLAP 的主要特点是直接仿照用户的多角度思考模式，预先为用户组建多维的数据模型。在这里，维指的是用户的分析角度，例如对销售数据的分析，时间周期是一个维度，产品类别、分销渠道、地理分布、客户群类也分别是不同的维度。一旦多维数据模型建立完成，用户可以快速地从各个分析角度获取数据，也能动态地在各个角度之间切换数据或者进行多角度综合分析，具有极大的分析灵活性。这也是 OLAP 在近年来被广泛关注的根本原因。OLAP 从设计理念和真正实现上都与旧有的管理信息系统有着本质的区别。

下面介绍一下 OLAP 的基本概念：

- 维（Dimension）：是用户观察数据的特定角度，是问题的一类属性，属性集合构成一个维（时间维、地理维等）。
- 维的层次（Level）：用户观察数据的某个特定角度（即某个维）还可能存在细节程度不同的各个描述方面（时间维包括日期、月份、季度、年）。
- 维的成员（Member）：即维的一个取值，是数据项在某个维中位置的描述，如“某年某月某日”是在时间维上的位置描述。
- 度量（Measure）：多维数组的取值。

OLAP 的基本多维分析操作有钻取（Drill-up 和 Drill-down）、切片（Slice）和切块（Dice）以及旋转（Pivot）等。

- 钻取：改变维的层次，变换分析的粒度。它包括向下钻取（Drill-down）和向上钻取（Drill-up）/ 上滚 (Roll-up)。向上钻取是在某一维上将低层次的细节数据概括到高层次的汇总数据，或者减少维数；而向下钻取则相反，从汇总数据深入到细节数据进行观察或增加新维。
- 切片和切块：在一部分维上选定值后，关心度量数据在剩余维上的分布。如果剩余的维只有两个，则是切片；如果有三个或以上，则是切块。
- 旋转：变换维的方向，即在表格中重新安排维的放置（如行列互换）。

1.3.3　OLTP 与 OLAP 的比较

OLTP 主要执行基本的、日常的事务处理，比如在银行存取一笔款，就是一个事务交易。OLTP 的特点一般有：

- 实时性要求高。
- 查询的数据量不是很大。
- 交易一般是确定的，所以 OLTP 是对确定性的数据进行存取。
- 并发性要求高，并且严格要求事务的完整性、安全性。

OLAP 是数据仓库系统的主要应用，其典型的应用就是复杂的动态报表系统。OLAP 的特点一般有：

- 实时性要求不是很高，很多应用最多每天更新一次数据。
- 数据量大。因为 OLAP 支持的是动态查询，用户要通过对很多数据的统计才能得到想要知道的信息，如时间序列分析等，所以处理的数据量很大。
- 因为重点在于决策支持，所以查询一般是动态的，也就是说允许用户随时提出查询的要求。因此，在 OLAP 中通过一个重要概念“维”来搭建一个动态查询的平台（或技术），供用户自己决定需要知道的信息。

OLAP 和 OLTP 的主要区别如表 1-1 所示。

表 1-1　OLTP 和 OLAP 之间的区别

	OLTP	OLAP
用户	操作人员	决策人员
功能	日常操作处理	分析决策
DB 设计	面向应用	面向主题
数据	当前的，最新的	历史的，聚集的
存取	读 / 写数十条记录	读上百万的数据
工作单位	简单的事务	复杂的查询
用户数	上千个	上百个
DB 大小	一般小于 500GB	大于 1TB

1.3.4 MySQL 存储引擎及其面向的数据库应用

由于工作的需要笔者有很长一段时间需要与开发人员进行沟通，并在必要时展开相关的培训工作。在这个过程中，笔者发现大多数开发人员不知道 MySQL 的存储引擎概念，这可能和他们以往开发的数据库应用如 Microsoft SQL Server、Oracle、DB2 等有关。而在 MySQL 数据库中，存储引擎的概念显得尤为重要，每个存储引擎可能面向一种特定或者最优的数据库应用环境。

图 1-1 显示了 MySQL 数据库的体系结构，可见 MySQL 数据库由以下几部分组成：

- 连接池组件（Connection Pool）。
- 管理服务和工具组件（Management Services &Utilities）。
- SQL 接口组件（SQL Interface）。
- 查询分析器组件（Parser）。
- 优化器组件（Optimizer）。
- 缓冲组件（Caches & Buffers）。
- 插件式存储引擎（Pluggable Storage Engines）。
- 物理文件（File system）。

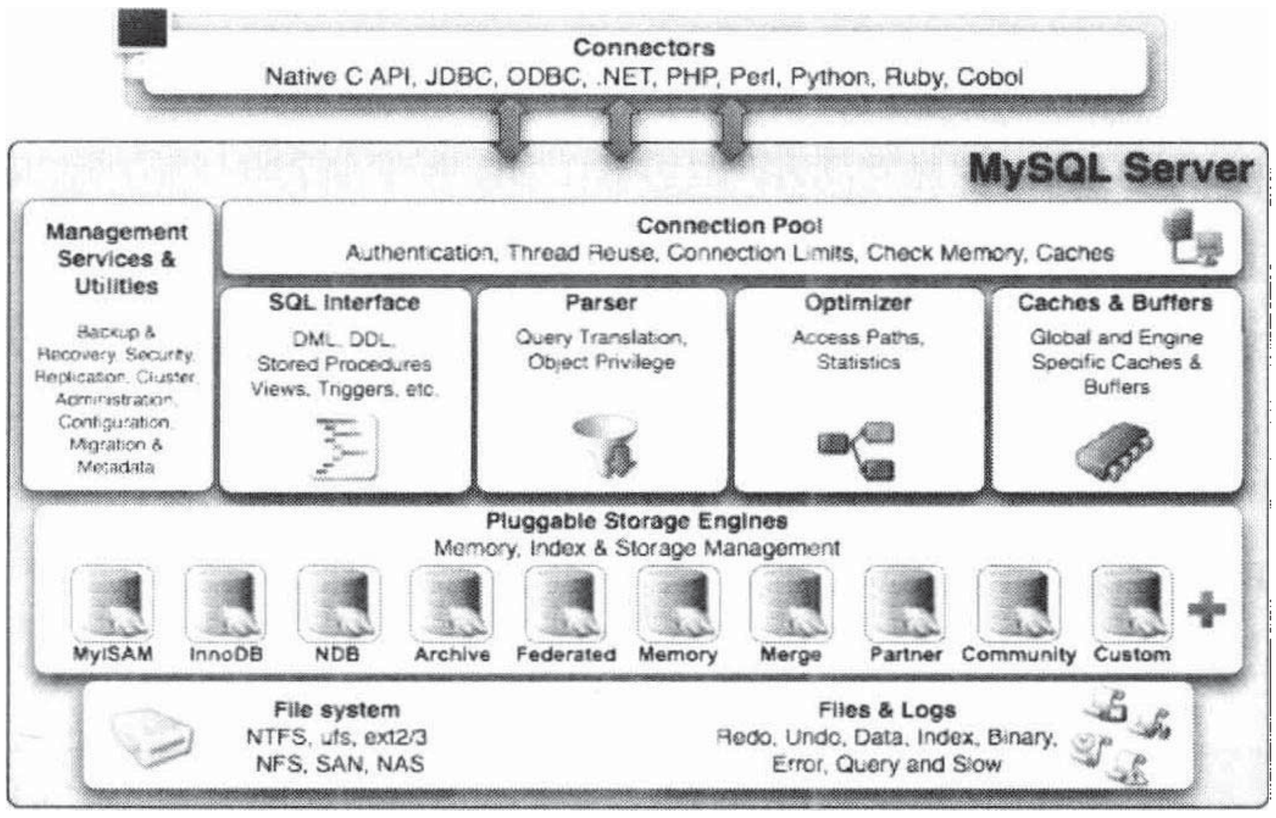


图 1-1 MySQL 数据库的体系结构

SQL 解析器、SQL 优化器、缓冲池、存储引擎等组件在每个数据库中都存在，但不是每个数据库都有这么多存储引擎。MySQL 的插件式存储引擎可以让存储引擎层的开发人员设计他们希望的存储层，例如，有的应用需要满足事务的要求，有的应用则不需要对事务有这么强的要求；有的希望数据能持久存储，有的只希望放在内存中，临时并快速地提供对数据的查询。下面将介绍 MySQL 数据库中一些常用的存储引擎及它们面向的数据库应用。

InnoDB 存储引擎：支持事务，其设计目标主要面向联机事务处理（OLTP）的应用。其

特点是行锁设计、支持外键，并支持类似 Oracle 的非锁定读，即默认读取操作不会产生锁。从 MySQL 5.5.8 版本开始是默认的存储引擎。

InnoDB 存储引擎将数据放在一个逻辑的表空间中，这个表空间就像黑盒一样由 InnoDB 存储引擎自身来管理。从 MySQL 4.1（包括 4.1）版本开始，可以将每个 InnoDB 存储引擎的表单独存放到一个独立的 ibd 文件中。此外，InnoDB 存储引擎支持将裸设备（row disk）用于建立其表空间。

InnoDB 通过使用多版本并发控制（MVCC）来获得高并发性，并且实现了 SQL 标准的 4 种隔离级别，默认为 REPEATABLE 级别，同时使用一种称为 netx-key locking 的策略来避免幻读（phantom）现象的产生。除此之外，InnoDB 存储引擎还提供了插入缓冲（insert buffer）、二次写（double write）、自适应哈希索引（adaptive hash index）、预读（read ahead）等高性能和高可用的功能。

对于表中数据的存储，InnoDB 存储引擎采用了聚集（clustered）的方式，每张表都是按主键的顺序进行存储的，如果没有显式地在表定义时指定主键，InnoDB 存储引擎会为每一行生成一个 6 字节的 ROWID，并以此作为主键。

InnoDB 存储引擎是 MySQL 数据库最为常用的一种引擎，Facebook、Google、Yahoo 等公司的成功应用已经证明了 InnoDB 存储引擎具备高可用性、高性能以及高可扩展性。对其底层实现的掌握和理解也需要时间和技术的积累。如果想深入了解 InnoDB 存储引擎的工作原理、实现和应用，可以参考《MySQL 技术内幕：InnoDB 存储引擎》一书。

MyISAM 存储引擎：不支持事务、表锁设计、支持全文索引，主要面向一些 OLAP 数据库应用，在 MySQL 5.5.8 版本之前是默认的存储引擎（除 Windows 版本外）。数据库系统与文件系统一个很大的不同在于对事务的支持，MyISAM 存储引擎是不支持事务的。究其根本，这也并不难理解。用户在所有的应用中是否都需要事务呢？在数据仓库中，如果没有 ETL 这些操作，只是简单地通过报表查询还需要事务的支持吗？此外，MyISAM 存储引擎的另一个与众不同的地方是，它的缓冲池只缓存（cache）索引文件，而不缓存数据文件，这与大多数的数据库都不相同。

NDB 存储引擎：2003 年，MySQL AB 公司从 Sony Ericsson 公司收购了 NDB 存储引擎。NDB 存储引擎是一个集群存储引擎，类似于 Oracle 的 RAC 集群，不过与 Oracle RAC 的 share everything 结构不同的是，其结构是 share nothing 的集群架构，因此能提供更高级别的高可用性。NDB 存储引擎的特点是数据全部放在内存中（从 5.1 版本开始，可以将非索引数据放在磁盘上），因此主键查找（primary key lookups）的速度极快，并且能够在线添加 NDB 数据存储节点（data node）以便线性地提高数据库性能。由此可见，NDB 存储引擎是高可用、高性能、高可扩展性的数据库集群系统，其面向的也是 OLTP 的数据库应用类型。

Memory 存储引擎：正如其名，Memory 存储引擎中的数据都存放在内存中，数据库重启或发生崩溃，表中的数据都将消失。它非常适合于存储 OLTP 数据库应用中临时数据的临

时表，也可以作为 OLAP 数据库应用中数据仓库的维度表。Memory 存储引擎默认使用哈希索引，而不是通常熟悉的 B+ 树索引。

Infobright 存储引擎：第三方的存储引擎。其特点是存储是按照列而非行的，因此非常适合 OLAP 的数据库应用。其官方网站是 http://www.infobright.org/，上面有不少成功的数据仓库案例可供分析。

NTSE 存储引擎：网易公司开发的面向其内部使用的存储引擎。目前的版本不支持事务，但提供压缩、行级缓存等特性，不久的将来会实现面向内存的事务支持。

MySQL 数据库还有很多其他存储引擎，上述只是列举了最为常用的一些引擎。如果你喜欢，完全可以编写专属于自己的引擎，这就是开源赋予我们的能力，也是开源的魅力所在。

1.4 图形化的 SQL 查询分析器

“工欲善其事，必先利其器”，即工匠想要做好工作，一定要先使工具锋利。这说明了工具的重要性。对于 SQL 编程，开发者不能仅依靠 MySQL 的命令行工具来完成 SQL 程序的开发，在开发存储过程时，可能要编写几百行甚至上千行的代码，然后还要对代码进行不断的调试，这时如果有一个便捷好用的工具，带来的效率提升将会是非常巨大和明显的。

1.4.1 MySQL Workbench

MySQL Workbench 是 MySQL 官方推出的一款开源的、免费的图形化工具，支持 Windows、Linux、Mac OS 操作系统。其主页为 http://wb.mysql.com/。MySQL Workbench 不单单是一个图形化的客户端，其还有如下主要功能：

- SQL 查询分析。
- 数据库建模。
- 数据库管理监控。

图 1-2 为 MySQL Workbench 在 Mac OS 操作系统下工作的情况。

MySQL Workbench 的功能实用，能很好地满足 SQL 编程工作。MySQL Workbench 是开源的、免费的及跨平台的，因此推荐大多数 SQL 开发人员使用。

1.4.2 Toad for MySQL

Toad for MySQL 是一款免费的开发工具，可让用户快速创建并执行查询、自动进行数据库对象管理、更高效地开发 SQL 代码。它提供的功能包括比较、提取和搜索对象，项目管理，数据导入或导出，以及数据库管理。用户还可通过它访问活跃的用户社区。Toad for

MySQL 可以大大提高工作效率，其主要功能有：

图 1-2　Mac OS 下的 MySQL Workbench

- ❑ 版本控制集成。
- ❑ 宏录制和播放。
- ❑ 数据库浏览。
- ❑ 代码段编辑。
- ❑ 安全管理。
- ❑ SQL 编辑。
- ❑ 快速的多标签模式浏览。
- ❑ 数据库提取、比较和搜索。
- ❑ 导入和导出。

图 1-3 显示了 Toad for MySQL 的工作界面。

Toad for MySQL 功能非常丰富，能满足各种不同类型业务的需求，缺点是只支持 Windows 平台并且收取一定的费用。不过鉴于其强大的功能，强烈建议广大 SQL 编程用户使用。

1.4.3　iMySQL-Front

iMySQL-Front 是一个开源的、跨平台的 MySQL 图形化的查询工具，支持 Windows、Linux、Mac OS 系统。其官方网站为 http://code.google.com/p/imysql-front/。图 1-4 为 iMySQL-Front 在 Mac OS 下工作的截图。

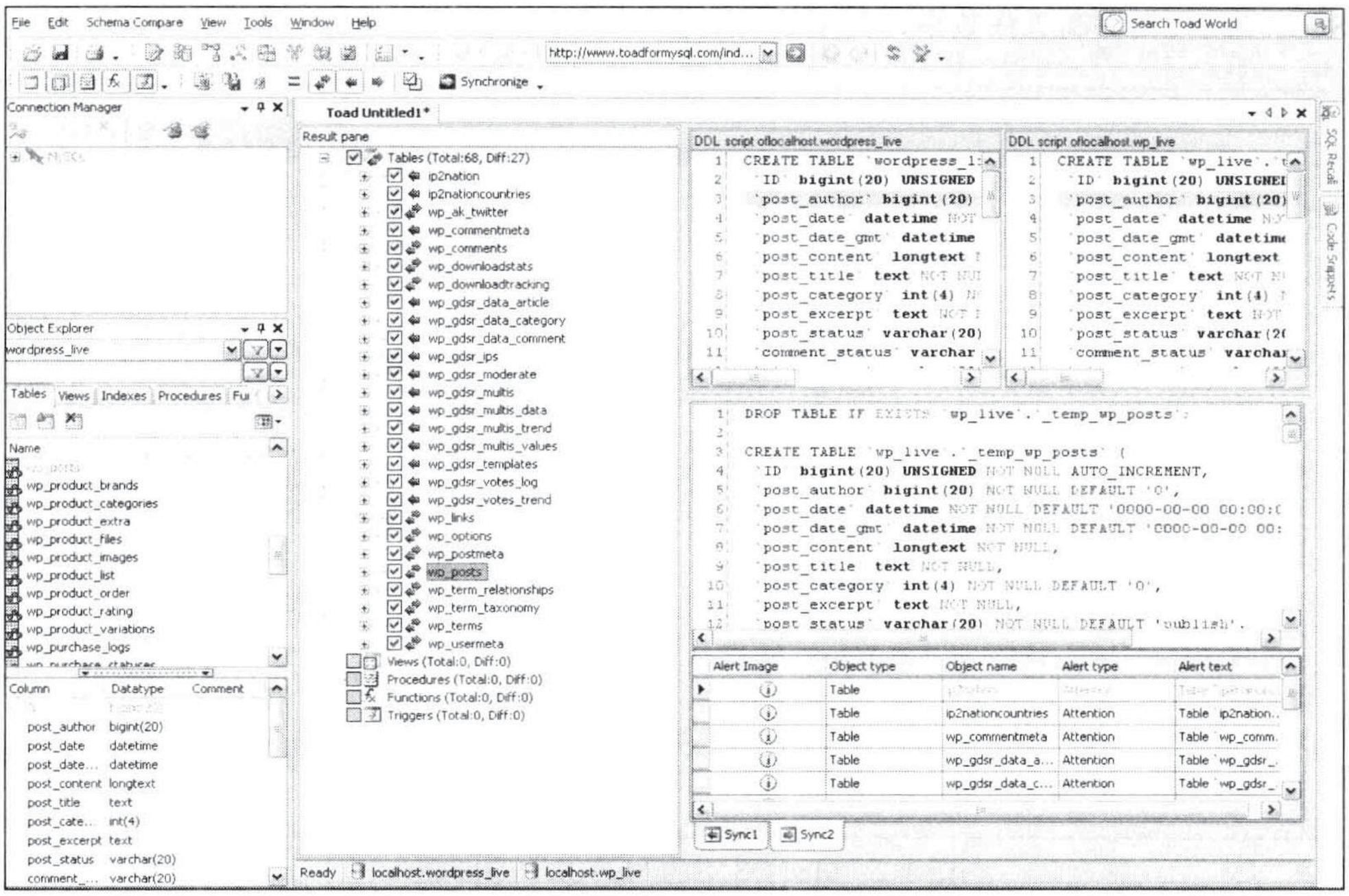


图 1-3　Toad for MySQL 界面

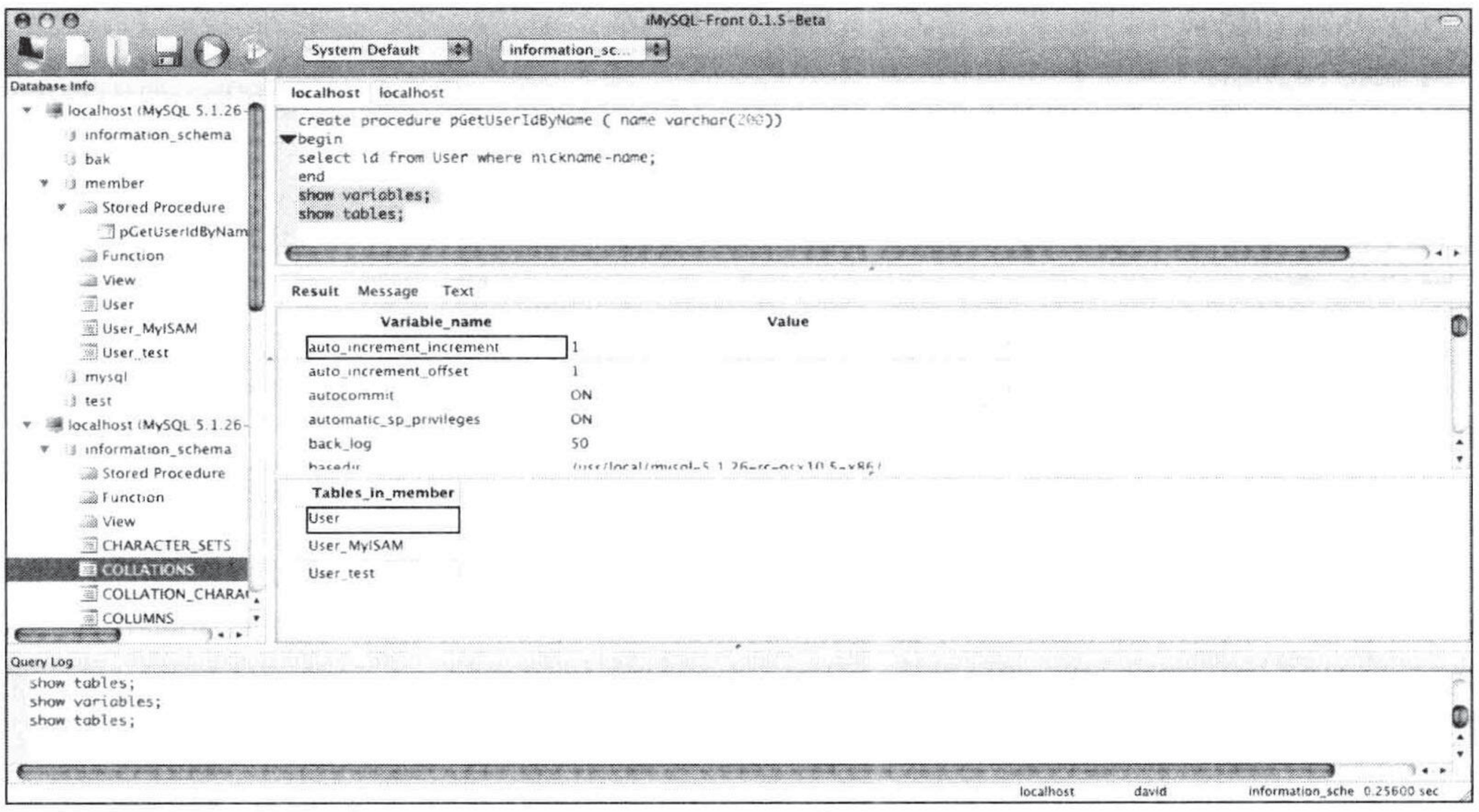


图 1-4　iMySQL-Front 工作在 Mac OS 下

iMySQL-Front 是用 wxWidgets（www.wxwidgets.org）开发的，因此可以很好地进行跨平台的开发。在开发和设计 iMySQL-Front 之前，其定位非常明确，即希望达到 Microsoft

SQL Server 图形化工具——查询分析器的工作效率，并且最大程度地符合数据库开发和管理的要求。iMySQL-Front 在 2009 年被 Softpedia 授为“100% FREE”。

1.5　小结

这一章先介绍了 MySQL 的历史以及它的分支版本，概括出 MySQL 数据库所经历的三个阶段：初期开源数据库阶段、Sun MySQL 阶段、Oracle MySQL 阶段。不论 MySQL 数据库处于哪个阶段，它都是开源的。此外还介绍了 MySQL 数据库独有的插件式存储引擎，使读者明白了存储引擎对不同数据库类型应用的重要性。最后，介绍了一些常用的 MySQL 图形化的查询分析器，通过这些图形化的工具，用户可以高效率地完成 SQL 的编程工作。

第 2 章
数据类型

数据类型在数据库中扮演着基础但又非常重要的角色。对数据类型的选择将影响与数据库交互的应用程序的性能。通常来说，如果一个页内可以存放尽可能多的行，那么数据库的性能就越好，因此选择一个正确的数据类型至关重要。另一方面，如果在数据库中创建表时选择了错误的数据类型，那么后期的维护成本可能非常大，用户需要花大量时间来进行 ALTER TABLE 操作。对于一张大表，可能需要等待更长的时间。因此，对于选择数据类型的应用程序设计人员，或是实现数据类型的 DBA，又或者是使用这些数据类型的程序员，花一些时间深入学习数据类型、理解它们的基本原理是非常必要的。在选择数据类型时要格外谨慎，因为在生产环境下更改数据类型可能是一种非常危险的操作。

建议读者花一点时间学习一下数据类型及它们的基本原理，比如学习“计算机组成原理”这些大学课程，虽然当时学起来可能觉得很枯燥，但是结合当前的数据库类型来看，可能会有另一番感受。此外，读者可以阅读《MySQL 技术内幕：InnoDB 存储引擎》这本书，其中详细并深入地介绍了某些数据类型的底层实现，如 VARCHAR、CHAR 和 BLOB 等类型。本章主要介绍一些与数据库类型相关的 SQL 编程问题，主要关注日期类型、数字类型及字符类型。

2.1 类型属性

在介绍数据类型前，先来介绍两个属性：UNSIGNED 和 ZEROFILL，是否使用这两个属性对选择数据类型有着莫大的关系。

2.1.1 UNSIGNED

UNSIGNED 属性就是将数字类型无符号化，与 C、C++ 这些程序语言中的 unsigned 含义相同。例如，INT 的类型范围是 −2 147 483 648 ～ 2 147 483 647，INT UNSIGNED 的范围类型就是 0 ～ 4 294 967 295。

看起来这是一个不错的属性选项，特别是对于主键是自增长的类型，因为一般来说，用户都希望主键是非负数。然而在实际使用中，UNSIGNED 可能会带来一些负面的影响，示例如下：

```
mysql> CREATE TABLE t ( a INT UNSIGNED, b INT UNSIGNED )
 ENGINE=INNODB;
Query OK, 0 rows affected (0.06 sec)

mysql> INSERT INTO t SELECT 1,2;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM t\G;
```

```
*************************** 1. row ***************************
a: 1
b: 2
1 row in set (0.00 sec)
```

我们创建了一个表 t，存储引擎为 InnoDB。表 t 上有两个 UNSIGNED 的 INT 类型。输入（1，2）这一行数据，目前看来都没有问题，接着运行如下语句：

```
SELECT a - b FROM t
```

这时结果会是什么呢？会是 −1 吗？答案是不确定的，可以是 −1，也可以是一个很大的正值，还可能会报错。在 Mac 操作系统中，MySQL 数据库提示如下错误：

```
mysql> SELECT a-b FROM t;
ERROR 1690 (22003): BIGINT UNSIGNED value is out of range in '(`test`.`t`.`a` -
    `test`.`t`.`b`)'
```

这个错误乍看起来非常奇怪，提示 BIGINT UNSIGNED 超出了范围，但是我们采用的类型都是 INT UNSIGNED 啊！而在另一台 Linux 操作系统中，运行的结果却是：

```
mysql> SELECT a -b FROM t\G;
*************************** 1. row ***************************
a - b: 4294967295
1 row in set (0.00 sec)
```

在发生上述这个问题的时候，有开发人员跑来和笔者说，他发现了一个 MySQL 的 Bug，MySQL 怎么会这么“傻”呢？在听完他的叙述之后，我写了如下的代码并告诉他，这不是 MySQL 的 Bug，C 语言同样也会这么“傻”。

```
#include <stdio.h>

int main(){
    unsigned int a;
    unsigned int b;
    a = 1;
    b = 2;
    printf(a - b: %d\n,a-b);
    printf(a - b: %u\n,a-b);
    return 1;
}
```

上述代码的运行结果是：

```
a - b: -1
a - b: 4294967295
```

可以看到，在 C 语言中 a−b 也可以返回一个非常巨大的整型数，这个值是 INT UNSIGNED 的最大值。难道 C 语言也发生了 Bug？这怎么可能呢？

在实际的使用过程中，MySQL 给开发人员的印象就是存在很多 Bug，只要结果出乎预料或者有开发人员不能理解的情况发生时，他们往往把这归咎于 MySQL 的 Bug。和其他数据库一样，MySQL 的确存在一些 Bug，其实并不是 MySQL 数据库的 Bug 比较多，去看一下 Oracle RAC 的 Bug，那可能就更多了，它可是 Oracle 的一款旗舰产品。因此，不能简单地认为这个问题是 MySQL 的 Bug。

对于上述这个问题，正如上述所分析的，如果理解整型数在数据库中的表示方法，那么这些就非常好理解了，这也是为什么之前强调需要看一些计算机组成原理方面相关书籍的原因。将上面的 C 程序做一些修改：

```
#include <stdio.h>

int main(){
    unsigned int a;
    unsigned int b;
    a = 1;
    b = 2;
    printf(a - b: %d,%x\n,a-b,a-b);
    printf(a - b: %u,%x\n,a-b,a-b);
    return 1;
}
```

这次不仅打印出 a-b 的结果，也打印出 a-b 的十六进制结果，运行程序后的结果如下所示：

```
a - b: -1,ffffffff
a - b: 4294967295,ffffffff
```

可以看到结果都是 0xFFFFFFFF，只是 0xFFFFFFFF 可以代表两种值：对于无符号的整型值，其是整型数的最大值，即 4 294 967 295；对于有符号的整型数来说，第一位代表符号位，如果是 1，表示是负数，这时应该是取反加 1 得到负数值，即 −1。

这个问题的核心是，在 MySQL 数据库中，对于 UNSIGNED 数的操作，其返回值都是 UNSIGNED 的。而正负数这个问题在《MySQL 技术内幕：InnoDB 存储引擎》中有更深入的分析，有兴趣的可以进一步研究。

那么，怎么获得 −1 这个值呢？这并不是一件难事，只要对 SQL_MODE 这个参数进行设置即可，例如：

```
mysql>SET sql_mode='NO_UNSIGNED_SUBTRACTION';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT a-b FROM t\G;
*************************** 1. row ***************************
a-b: -1
1 row in set (0.00 sec)
```

后面会对 SQL_MODE 进一步讨论，这里不进行深入的讨论。笔者个人的看法是尽量不要使用 UNSIGNED，因为可能会带来一些意想不到的效果。另外，对于 INT 类型可能存放不了的数据，INT UNSIGNED 同样可能存放不了，与其如此，还不如在数据库设计阶段将 INT 类型提升为 BIGINT 类型。

2.1.2 ZEROFILL

ZEROFILL 属性非常有意思，更像是一个显示的属性。很多初学者往往对 MySQL 数据库中数字类型后面的长度值很迷茫。下面通过 SHOW CREATE TABLE 命令来看一下 t 表的建表语句。

```
mysql> SHOW CREATE TABLE t\G;
*************************** 1. row ***************************
       Table: t
Create Table: CREATE TABLE `t` (
  `a` int(10) unsigned DEFAULT NULL,
  `b` int(10) unsigned DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1
1 row in set (0.01 sec)
```

可以看到 int（10），这代表什么意思呢？整型不就是 4 字节的吗？这 10 又代表什么呢？其实如果没有 ZEROFILL 这个属性，括号内的数字是毫无意义的。a 和 b 列就是前面插入的数据，例如：

```
mysql> SELECT * FROM t\G;
*************************** 1. row ***************************
a: 1
b: 2
1 row in set (0.00 sec)
```

但是在对列添加 ZEROFILL 属性后，显示的结果就有所不同了，例如对表 t 进行 ALTER TABLE 修改：

```
mysql> ALTER TABLE t CHANGE COLUMN a a int(4) UNSIGNED ZEROFILL;
Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

这里对 a 列进行了修改，为其添加了 ZEROFILL 属性，并且将默认的 int（10）修改为 int（4），这时再进行查找操作，返回的结果如下：

```
mysql> SELECT a,b FROM t\G;
*************************** 1. row ***************************
a: 0001
b: 2
1 row in set (0.00 sec)
```

这次可以看到 a 的值由原来的 1 变为 0001，这就是 ZEROFILL 属性的作用，如果宽度

小于设定的宽度（这里的宽度为4），则自动填充0。要注意的是，这只是最后显示的结果，在MySQL中实际存储的还是1。为什么是这样呢？我们可以用函数HEX来证明。

```
mysql> SELECT a,HEX(a) FROM t\G;
*************************** 1. row ***************************
     a: 0001
hex(a): 1
1 row in set (0.00 sec)
```

可以看到在数据库内部存储的还是1，0001只是设置了ZEROFILL属性后的一种格式化输出而已。进一步思考，如果数据库内部存储的是0001这样的字符串，又怎么进行整型的加、减、乘、除操作呢？

2.2 SQL_MODE设置

SQL_MODE可能是比较容易让开发人员和DBA忽略的一个变量，默认为空。SQL_MODE的设置其实是比较冒险的一种设置，因为在这种设置下可以允许一些非法操作，比如可以将NULL插入NOT NULL的字段中，也可以插入一些非法日期，如“2012-12-32”。因此在生产环境中强烈建议开发人员将这个值设为严格模式，这样有些问题可以在数据库的设计和开发阶段就能发现，而如果在生产环境下运行数据库后发现这类问题，那么修改的代价将变得十分巨大。此外，正确地设置SQL_MODE还可以做一些约束（Constraint）检查的工作。

对于SQL_MODE的设置，可以在MySQL的配置文件如my.cnf和my.ini中进行，也可以在客户端工具中进行，并且可以分别进行全局的设置或当前会话的设置。下面的命令可以用来查看当前SQL_MODE的设置情况。

```
mysql> SELECT @@global.sql_mode\G;
*************************** 1. row ***************************
@@global.sql_mode:
1 row in set (0.00 sec)

mysql> SELECT @@session.sql_mode\G;
*************************** 1. row ***************************
@@session.sql_mode: NO_UNSIGNED_SUBTRACTION
1 row in set (0.00 sec)
```

可以看到当前全局的SQL_MODE设置为空，而当前会话的设置为NO_UNSIGNED_SUBTRACTION。通过以下语句可以将当前的SQL_MODE设置为严格模式。

```
mysql> SET GLOBAL sql_mode='strict_trans_tables';
Query OK, 0 rows affected (0.00 sec)
```

严格模式是指将 SQL_MODE 变量设置为 STRICT_TRANS_TABLES 或 STRICT_ALL_TABLES 中的至少一种。现在来看一下 SQL_MODE 可以设置的选项。

STRICT_TRANS_TABLES：在该模式下，如果一个值不能插入到一个事务表（例如表的存储引擎为 InnoDB）中，则中断当前的操作不影响非事务表（例如表的存储引擎为 MyISAM）。

ALLOW_INVALID_DATES：该选项并不完全对日期的合法性进行检查，只检查月份是否在 1 ～ 12 之间，日期是否在 1 ～ 31 之间。该模式仅对 DATE 和 DATETIME 类型有效，而对 TIMESTAMP 无效，因为 TIMESTAMP 总是要求一个合法的输入。

ANSI_QUOTES：启用 ANSI_QUOTES 后，不能用双引号来引用字符串，因为它将被解释为识别符，示例如下：

```
mysql> CREATE TABLE z ( a VARCHAR(10))ENGINE=INNODB;
Query OK, 0 rows affected (0.00 sec)

mysql>INSERT INTO z SELECT "aaa";
Query OK, 1 rows affected (0.00 sec)

mysql> SET sql_mode='ANSI_QUOTES';
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO z SELECT "aaa";
ERROR 1054 (42S22): Unknown column 'aaa' in 'field list'
```

ERROR_FOR_DIVISION_BY_ZERO：在 INSERT 或 UPDATE 过程中，如果数据被零除（或 MOD（X，0)），则产生错误（否则为警告）。如果未给出该模式，那么数据被零除时 MySQL 返回 NULL。如果用到 INSERT IGNORE 或 UPDATE IGNORE 中，MySQL 生成被零除警告，但操作结果为 NULL。

HIGH_NOT_PRECEDENCE NOT：操作符的优先顺序是表达式。例如，NOT a BETWEEN b AND c 被解释为 NOT（a BETWEEN b AND c)，在一些旧版本 MySQL 中，前面的表达式被解释为（NOT a）BETWEEN b AND c。启用 HIGH_NOT_PRECEDENCE SQL 模式，可以获得以前旧版本的更高优先级的结果。下面看一个例子：

```
mysql> SELECT 0 BETWEEN -1 AND 1\G;
*************************** 1. row ***************************
0 BETWEEN -1 AND 1: 1
1 row in set (0.00 sec)
```

0 在 −1 到 1 之间，所以返回 1，如果加上 NOT，则返回 0，过程如下：

```
mysql> SELECT @@sql_mode\G;
*************************** 1. row ***************************
@@sql_mode:
1 row in set (0.00 sec)
```

```
mysql> SELECT not 0 BETWEEN -1 AND 1\G;
*************************** 1. row ***************************
NOT 0 BETWEEN -1 AND 1: 0
1 row in set (0.00 sec)
```

但是如果启用 HIGH_NOT_PRECEDENCE 模式，则 SELECT NOT 0 BETWEEN -1 AND 1 被解释为 SELECT（NOT 0）BETWEEN -1 AND 1，结果就完全相反，如下所示：

```
mysql> SELECT NOT 0 BETWEEN -1 AND 1\G;
*************************** 1. row ***************************
NOT 0 BETWEEN -1 AND 1: 1
1 row in set (0.00 sec)
```

从上述例子中还能看出，在 MySQL 数据库中 BETWEEN a AND b 被解释为 [a，b]。下面做两个简单的测试。

```
mysql> SELECT 1 BETWEEN -1 AND 1\G;
*************************** 1. row ***************************
1 BETWEEN -1 AND 1: 1
1 row in set (0.00 sec)

mysql> SELECT -1 BETWEEN -1 AND 1\G;
*************************** 1. row ***************************
-1 BETWEEN -1 AND 1: 1
1 row in set (0.00 sec)
```

IGNORE_SPACE：函数名和括号“(”之间有空格。除了增加一些烦恼，这个选项好像没有任何好处，要访问保存为关键字的数据库、表或列名，用户必须引用该选项。例如某个表中有 user 这一列，而 MySQL 数据库中又有 user 这个函数，user 会被解释为函数，如果想要选择 user 这一列，则需要引用。

NO_AUTO_CREATE_USER：禁止 GRANT 创建密码为空的用户。

NO_AUTO_VALUE_ON_ZERO：该选项影响列为自增长的插入。在默认设置下，插入 0 或 NULL 代表生成下一个自增长值。如果用户希望插入的值为 0，而该列又是自增长的，那么这个选项就有用了。

NO_BACKSLASH_ESCAPES：反斜杠“\”作为普通字符而非转义符，示例如下：

```
mysql> SET sql_mode='';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT '\\'\G;
*************************** 1. row ***************************
\: \
1 row in set (0.00 sec)

mysql> SET sql_mode='NO_BACKSLASH_ESCAPES';
```

```
Query OK, 0 rows affected (0.00 sec)

mysql> SET '\\'\G;
*************************** 1. row ***************************
\\: \\
1 row in set (0.00 sec)
```

NO_DIR_IN_CREATE：在创建表时忽视所有 INDEX DIRECTORY 和 DATA DIRECTORY 的选项。

NO_ENGINE_SUBSTITUTION：如果需要的存储引擎被禁用或未编译，那么抛出错误。默认用默认的存储引擎替代，并抛出一个异常。

NO_UNSIGNED_SUBTRACTION：之前已经介绍过，启用这个选项后两个 UNSIGNED 类型相减返回 SIGNED 类型。

NO_ZERO_DATE：在非严格模式下，可以插入形如“0000-00-00 00:00:00”的非法日期，MySQL 数据库仅抛出一个警告。而启用该选项后，MySQL 数据库不允许插入零日期，插入零日期会抛出错误而非警告。

NO_ZERO_IN_DATE：在严格模式下，不允许日期和月份为零。如“2011-00-01”和“2011-01-00”这样的格式是不允许的。采用日期或月份为零的格式时 MySQL 都会直接抛出错误而非警告，示例如下：

```
mysql> SET sql_mode='NO_ZERO_IN_DATE';
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE TABLE a ( a DATETIME );
Query OK, 0 rows affected (0.04 sec)

mysql> INSERT INTO a SELECT '2011-01-00';
ERROR 1292 (22007): Incorrect datetime value: '2011-01-00' for column 'a' at row 1
```

ONLY_FULL_GROUP_BY：对于 GROUP BY 聚合操作，如果在 SELECT 中的列没有在 GROUP BY 中出现，那么这句 SQL 是不合法的，因为 a 列不在 GROUP BY 从句中，示例如下：

```
mysql> SET sql_mode='ONLY_FULL_GROUP_BY';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT a,SUM(b) FROM t GROUP BY b;
ERROR 1055 (42000): 'test.t.a' isn't in GROUP BY
```

PAD_CHAR_TO_FULL_LENGTH：对于 CHAR 类型字段，不要截断空洞数据。空洞数据就是自动填充值为 0x20 的数据。先来看 MySQL 数据库在默认情况下的表现。

```
mysql> CREATE TABLE t ( a CHAR(10) );
Query OK, 0 rows affected (0.04 sec)

mysql> INSERT INTO t SELECT 'a';
```

```
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql>SELECT a,CHAR_LENGTH(a),HEX(a) FROM t\G;
*************************** 1. row ***************************
           a: a
CHAR_LENGTH(a): 1
       HEX(a): 61
1 row in set (0.04 sec)
```

可以看到，在默认情况下，虽然a列是CHAR类型，但是返回的长度是1，这是因为MySQL数据库已经对后面的空洞数据进行了截断。若启用PAD_CHAR_TO_FULL_LENGTH选项，则反映的是实际存储的内容，例如：

```
mysql> SELECT a,CHAR_LENGTH(a),HEX(a) FROM t\G;
*************************** 1. row ***************************
          a: a
CHAR_LENGTH(a): 10
       HEX(a): 61202020202020202020
1 row in set (0.00 sec)
```

可以看到在CHAR列a中实际存储的值为0x61202020202020202020。

PIPES_AS_CONCAT：将“||”视为字符串的连接操作符而非或运算符，这和Oracle数据库是一样的，也和字符串的拼接函数Concat相类似，例如：

```
mysql> SET sql_mode='pipes_as_concat';
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT 'a'||'b'||'c'\G;
*************************** 1. row ***************************
'a'||'b'||'c': abc
1 row in set (0.00 sec)
```

REAL_AS_FLOAT：将REAL视为FLOAT的同义词，而不是DOUBLE的同义词。

STRICT_ALL_TABLES：对所有引擎的表都启用严格模式。（STRICT_TRANS_TABLES只对支持事务的表启用严格模式）。

在严格模式下，一旦任何操作的数据产生问题，都会终止当前的操作。对于启用STRICT_ALL_TABLES选项的非事务引擎来说，这时数据可能停留在一个未知的状态。这可能不是所有非事务引擎愿意看到的一种情况，因此需要非常小心这个选项可能带来的潜在影响。

下面的几种SQL_MODE设置是之前讨论的几种选项的组合。

ANSI：等同于REAL_AS_FLOAT、PIPES_AS_CONCAT和ANSI_QUOTES、IGNORE_SPACE的组合。

ORACLE：等同于PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE、NO_KEY_

OPTIONS、NO_TABLE_OPTIONS、NO_FIELD_OPTIONS 和 NO_AUTO_CREATE_USER 的组合。

TRADITIONAL：等同于 STRICT_TRANS_TABLES、STRICT_ALL_TABLES、NO_ZERO_IN_DATE、NO_ZERO_DATE、ERROR_FOR_DIVISION_BY_ZERO、NO_AUTO_CREATE_USER 和 NO_ENGINE_SUBSTITUTION 的组合。

MSSQL：等同于 PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE、NO_KEY_OPTIONS、NO_TABLE_OPTIONS 和 NO_FIELD_OPTIONS 的组合。

DB2：等同于 PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE、NO_KEY_OPTIONS、NO_TABLE_OPTIONS 和 NO_FIELD_OPTIONS 的组合。

MYSQL323：等同于 NO_FIELD_OPTIONS 和 HIGH_NOT_PRECEDENCE 的组合。

MYSQL40：等同于 NO_FIELD_OPTIONS 和 HIGH_NOT_PRECEDENCE 的组合。

MAXDB：等同于 PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE、NO_KEY_OPTIONS、NO_TABLE_OPTIONS、NO_FIELD_OPTIONS 和 NO_AUTO_CREATE_USER 的组合。

2.3 日期和时间类型

MySQL 数据库中有五种与日期和时间有关的数据类型，各种日期数据类型所占空间如表 2-1 所示。

表 2-1 各种日期数据类型及其所占用的空间

类型	所占空间
DATETIME	8 字节
DATE	3 字节
TIMESTAMP	4 字节
YEAR	1 字节
TIME	3 字节

2.3.1 DATETIME 和 DATE

DATETIME 占用 8 字节，是占用空间最多的一种日期类型。它既显示了日期，同时也显示了时间。其可以表达的日期范围为“1000-01-01 00:00:00”到“9999-12-31 23:59:59”。

DATE 占用 3 字节，可显示的日期范围为“1000-01-01”到“9999-12-31”。

在 MySQL 数据库中，对日期和时间输入格式的要求是非常宽松的，以下的输入都可以视为日期类型。

- 2011-01-01 00:01:10
- 2011/01/01 00+01+10
- 20110101000110
- 11/01/01 00@01@10

其中，最后一种类型中的“11”有些模棱两可，MySQL 数据库将其视为 2011 还是 1911 呢？下面来做个测试：

```
mysql> SELECT CAST('11/01/01 00@01@10' AS DATETIME) AS datetime\G;
*************************** 1. row ***************************
datetime: 2011-01-01 00:01:10
1 row in set (0.00 sec)
```

可以看到数据库将其视为离现在最近的一个年份，这可能不是一个非常好的习惯。如果没有特别的条件和要求，还是在输入时按照标准的“YYYY-MM-DD HH:MM:SS”格式来进行。在 MySQL 5.5 版本之前（包括 5.5 版本），数据库的日期类型不能精确到微秒级别，任何的微秒数值都会被数据库截断，例如：

```
mysql> CREATE TABLE t ( a DATETIME );
Query OK, 0 rows affected (0.25 sec)

mysql> INSERT INTO t SELECT '2011-01-01 00:01:10.123456';
Query OK, 1 row affected (0.05 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM t\G;
*************************** 1. row ***************************
a: 2011-01-01 00:01:10
1 row in set (0.00 sec)
```

不过 MySQL 数据库提供了函数 MICROSECOND 来提取日期中的微秒值，示例如下：

```
mysql> SELECT MICROSECOND('2011-01-01 00:01:10.123456')\G;
*************************** 1. row ***************************
MICROSECOND('2011-01-01 00:01:10.123456'): 123456
1 row in set (0.00 sec)
```

有意思的是，MySQL 的 CAST 函数在强制转换到 DATETIME 时会保留到微秒数，不过在插入后同样会截断，示例如下：

```
mysql> SELECT CAST('2011-02-01 00:01:10.123456' AS DATETIME) D\G;
*************************** 1. row ***************************
D: 2011-02-01 00:01:10.123456
1 row in set (0.00 sec)

mysql> INSERT INTO t
    ->SELECT CAST('2011-02-01 00:01:10.123456' AS DATETIME);
```

```
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM t\G;
*************************** 1. row ***************************
a: 2011-01-01 00:01:10
*************************** 2. row ***************************
a: 2011-02-01 00:01:10
2 rows in set (0.00 sec)
```

然而从 MySQL 5.6.4 版本开始，MySQL 增加了对秒的小数部分（fractional second）的支持，具体语法为：

```
type_name (fsp)
```

其中，type_name 的类型可以是 TIME、DATETIME 和 TIMESTAMP。fsp 表示支持秒的小数部分的精度，最大为 6，表示微秒（microseconds）；默认为 0，表示没有小数部分，同时也是为了兼容之前版本中的 TIME、DATETIME 和 TIMESTAMP 类型。对于时间函数，如 CURTIME()、SYSDATE() 和 UTC_TIMESTAMP() 也增加了对 fsp 的支持，例如：

```
mysql> SELECT CURTIME(4) AS TIME\G;
*************************** 1. row ***************************
TIME: 10:22:37.4456
1 rows in set (0.00 sec)
```

注意 MariaDB 5.3 版本就对 TIME、DATETIME、TIMESTAMP 类型的微秒部分进行了支持。详细的说明可以从 http://kb.askmonty.org/en/microseconds-in-mariadb 中得到。

2.3.2 TIMESTAMP

TIMESTAMP 和 DATETIME 显示的结果是一样的，都是固定的“YYYY-MM-DD HH:MM:SS”的形式。不同的是，TIMESTAMP 占用 4 字节，显示的范围为“1970-01-01 00:00:00” UTC 到“2038-01-19 03:14:07” UTC。其实际存储的内容为“1970-01-01 00:00:00”到当前时间的毫秒数。

注意 UTC 协调世界时，又称世界统一时间、世界标准时间和国际协调时间。它从英文 Coordinated Universal Time 和法文 Temps Universel Cordonné 而来。

TIMESTAMP 类型和 DATETIME 类型除了在显示时间范围上有所不同外，还有以下不同：

- 在建表时，列为 TIMESTAMP 的日期类型可以设置一个默认值，而 DATETIME 不行。
- 在更新表时，可以设置 TIMESTAMP 类型的列自动更新时间为当前时间。

首先来看一个默认设置时间的例子。

```
mysql> CREATE TABLE t
    -> ( a INT,
    ->   b TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    -> )ENGINE=InnoDB;
Query OK, 0 rows affected (0.04 sec)

mysql> INSERT INTO t (a) VALUES (1);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM t\G;
*************************** 1. row ***************************
a: 1
b: 2011-02-02 16:42:57
1 row in set (0.01 sec)
```

接着来看一个执行 UPDATE 时更新为当前时间的例子。

```
mysql> CREATE TABLE t
    -> ( a INT ,
    ->  b TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
    -> );

mysql> INSERT INTO t SELECT 1,CURRENT_TIMESTAMP;
Query OK, 1 row affected (0.02 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM t\G;
*************************** 1. row ***************************
a: 1
b: 2011-02-02 16:45:40
1 row in set (0.01 sec)

# 等待一段时间

mysql> UPDATE t SET a=2;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM t\G;
*************************** 1. row ***************************
a: 2
b: 2011-02-02 16:47:39
1 row in set (0.00 sec)
```

可以发现在执行 UPDATE 操作后，b 列的时间由原来的 16:45:40 更新为了 16:47:39。如果执行了 UPDATE 操作，而实际上行并没有得到更新，那么是不会更新 b 列的，例如：

```
mysql> SELECT * FROM t\G;
*************************** 1. row ***************************
a: 2
```

```
b: 2011-02-02 16:47:39
1 row in set (0.00 sec)

mysql> UPDATE t SET a=2;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 1  Changed: 0  Warnings: 0

mysql> SELECT * FROM t\G;
*************************** 1. row ***************************
a: 2
b: 2011-02-02 16:47:39
1 row in set (0.00 sec)
```

可以看到执行 UPDATE t SET a=2 并没有改变行中的任何数据，显示 Changed:0 表示该行实际没有得到更新，故 b 列并不会进行相应的更新操作。

当然，可以在建表时将 TIMESTAMP 列设为一个默认值，也可以设为在更新时的时间，例如：

```
mysql> CREATE TABLE t (
    -> a INT,
    -> b TIMESTAMP DEFAULT ON UPDATE CURRENT_TIMESTAMP
    ->)ENGINE=InnoDB;
```

2.3.3 YEAR 和 TIME

YEAR 类型占用 1 字节，并且在定义时可以指定显示的宽度为 YEAR(4) 或 YEAR(2)，例如：

```
mysql> CREATE TABLE t ( a YEAR(2));
Query OK, 0 rows affected (0.05 sec)

mysql> INSERT INTO t SELECT '1990';

mysql> SELECT * FROM t;
+------+
| a    |
+------+
|   90 |
+------+
1 row in set (0.00 sec)
```

对于 YEAR（4），其显示年份的范围为 1901 ～ 2155；对于 YEAR（2），其显示年份的范围为 1970 ～ 2070。在 YEAR（2）的设置下，00 ～ 69 代表 2000 ～ 2069 年。

TIME 类型占用 3 字节，显示的范围为“−838：59：59”～“838：59：59”。有人会奇怪为什么 TIME 类型的时间可以大于 23。因为 TIME 类型不仅可以用来保存一天中的时间，也可以用来保存时间间隔，同时这也解释了为什么 TIME 类型也可以存在负值。和

DATETIME 类型一样，TIME 类型同样可以显示微秒时间，但是在插入时，数据库同样会进行截断操作，例如：

```
mysql> CREATE TABLE t ( a TIME );
Query OK, 0 rows affected (0.04 sec)

mysql> INSERT INTO t SELECT '14:40:20.123456';
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM t;
+----------+
| a        |
+----------+
| 14:40:20 |
+----------+
1 row in set (0.00 sec)
```

2.3.4 与日期和时间相关的函数

日期函数可能是比较常使用的一种函数。下面介绍一些最为常用的日期函数及一些容易忽略的问题。

1. NOW、CURRENT_TIMESTAMP 和 SYSDATE

这些函数都能返回当前的系统时间，它们之间有区别吗？先来看个例子。

```
mysql> SELECT NOW(),CURRENT_TIMESTAMP(),SYSDATE()\G;
*************************** 1. row ***************************
            NOW(): 2011-02-04 20:35:04
CURRENT_TIMESTAMP(): 2011-02-04 20:35:04
        SYSDATE(): 2011-02-04 20:35:04
```

从上面的例子看来，3 个函数都是返回当前的系统时间，再来看下面这个例子：

```
mysql>SELECT
    ->      NOW(),
    ->      CURRENT_TIMESTAMP(),
    ->      SYSDATE(),
    ->      SLEEP(2),
    ->      NOW(),
    ->      CURRENT_TIMESTAMP(),
    ->      SYSDATE()\G
*************************** 1. row ***************************
            NOW(): 2011-11-29 21:04:49
CURRENT_TIMESTAMP(): 2011-11-29 21:04:49
        SYSDATE(): 2011-11-29 21:04:49
         SLEEP(2): 0
           NOW(): 2011-11-29 21:04:49
```

```
CURRENT_TIMESTAMP(): 2011-11-29 21:04:49
        SYSDATE(): 2011-11-29 21:04:51
1 row in set (2.00 sec)
```

在上面这个例子中人为地加入了 SLEEP 函数，让其等待 2 秒，这时可以发现 SYSDATE 返回的时间和 NOW 及 CURRENT_TIMESTAMP 是不同的，SYSDATE 函数慢了 2 秒。究其原因是这 3 个函数有略微区别：

- CURRENT_TIMESTAMP 是 NOW 的同义词，也就是说两者是相同的。
- SYSDATE 函数返回的是执行到当前函数时的时间，而 NOW 返回的是执行 SQL 语句时的时间。

因此在上面的例子中，两次执行 SYSDATE 函数返回不同的时间是因为第二次调用执行该函数时等待了前面 SLEEP 函数 2 秒。而对于 NOW 函数，不管是在 SLEEP 函数之前还是之后执行，返回的都是执行这条 SQL 语句时的时间。

2. 时间加减函数

先来看一个例子。

```
mysql> SELECT NOW(),NOW()+0\G;
*************************** 1. row ***************************
  NOW(): 2011-02-04 20:46:33
NOW()+0: 20110204204633.000000
1 row in set (0.00 sec)
```

可以看到，NOW（）函数可以返回时间，也可以返回一个数字，就看用户如何使用。如果相对当前时间进行增加或减少，并不能直接加上或减去一个数字，而需要使用特定的函数，如 DATE_ADD 或 DATE_SUB，前者表示增加，后者表示减少。其具体的使用方法有 DATE_ADD（date,INTERVAL expr unit）和 DATE_SUB（date,INTERVAL expr unit），示例如下：

```
mysql> SELECT NOW() AS now,
       DATE_ADD(now(),INTERVAL 1 DAY) AS tomorrow,
       DATE_SUB(now(),INTERVAL 1 DAY) AS yesterday\G;
*************************** 1. row ***************************
     now: 2011-02-04 20:53:25
tomorrow: 2011-02-05 20:53:25
yesterday: 2011-02-03 20:53:25
1 row in set (0.00 sec)
```

其中 expr 值可以是正值也可以是负值，因此可以使用 DATE_ADD 函数来完成 DATE_SUB 函数的工作，例如：

```
mysql> SELECT NOW() AS now,
       DATE_ADD(NOW(),INTERVAL 1 DAY) AS tomorrow,
```

```
       DATE_ADD(NOW(),INTERVAL -1 DAY) AS yesterday\G;
*************************** 1. row ***************************
      now: 2011-02-04 20:55:40
 tomorrow: 2011-02-05 20:55:40
yesterday: 2011-02-03 20:55:40
1 row in set (0.00 sec)
```

还有一个问题，如果是闰月，那么DATE_ADD函数怎么处理呢？MySQL的默认行为是这样的：如果目标年份是闰月，那么返回的日期为2月29日；如果不是闰月，那么返回日期是2月28日。示例如下：

```
mysql> SELECT DATE_ADD('2000-2-29',INTERVAL 4 YEAR) AS year;
+------------+
| year       |
+------------+
| 2004-02-29 |
+------------+
1 row in set (0.00 sec)

mysql> SELECT DATE_ADD('2000-2-29',INTERVAL 1 YEAR) AS year;
+------------+
| year       |
+------------+
| 2001-02-28 |
+------------+
1 row in set (0.00 sec)
```

在上面的例子中使用了DAY和YEAR数据类型，其实也可以使用MICROSECOND、SECOND、MINUTE、HOUR、WEEK、MONTH等类型，例如：

```
mysql> SELECT NOW() AS now,
       DATE_ADD(NOW(), INTERVAL 1 HOUR) AS next time\G;
*************************** 1. row ***************************
      now: 2011-02-04 21:00:15
next time: 2011-02-04 22:00:15
1 row in set (0.00 sec)
```

3. DATE_FORMAT函数

这个函数本身没有什么需要探讨的地方，其作用只是按照用户的需求格式化打印出日期，例如：

```
mysql> SELECT DATE_FORMAT(NOW(),'%Y%m%d') AS datetime;
+----------+
| datetime |
+----------+
| 20110204 |
+----------+
1 row in set (0.00 sec)
```

但是开发人员往往会错误地使用这个函数，导致非常严重的后果。例如在需要查询某一天的数据时，有些开发人员会写如下的语句：

```
SELECT * FROM table WHERE DATE_FORMAT(date,'%Y%m%d')='xxxx-xx-xx';
```

一般来说表中都会有一个对日期类型的索引，如果使用上述的语句，优化器绝对不会使用索引，也不可能通过索引来查询数据，因此上述查询的执行效率可能非常低。

2.4 关于日期的经典 SQL 编程问题

前面已经介绍了时间和日期类型及相关的函数，这一节将探讨与日期相关的 SQL 编程问题。

2.4.1 生日问题

与日期相关的第一个问题是根据某个用户的出生日期和当前日期，计算他最近的生日。通过对这个问题的处理，演示如何通过使用日期函数来正确处理闰月。

在生日问题中，一般对闰月的处理如下：如果是闰月，那么返回 2 月 28 日；如果不是闰月，则返回 3 月 1 日（大部分是出于法律的要求）。例如，当前的日期是 2005 年 9 月 26 日，有人出生在 1972 年 2 月 29 日，查询后返回的该用户最近的生日应该是 2006 年 3 月 1 日。如果当前是 2007 年 9 月 26 日，那么查询后应该返回 2008 年 2 月 29 日。

在解决该问题之前，运行下列清单中的代码，初始化一些数据。在演示前，需要确认已经安装了 MySQL 官方的示例数据库 employees。

```
USE test;

CREATE TABLE employees LIKE employees.employees;

INSERT INTO employees
SELECT * FROM employees.employees LIMIT 10;

INSERT INTO employees
SELECT 10011,'1972-02-29','Jiang','David','M','1990-2-20';
```

这里人为地插入一个员工 David Jiang，其出生日期为“1972-02-29”，闰月。运行如下语句得到所有员工的出生信息。

```
SELECT
   CONCAT(last_name,' ',first_name) AS Name,
   birth_date AS BirthDay
FROM employees;
```

运行结果如表 2-2 所示。

表 2-2 每个员工的生日信息

Name	BirthDay
Facello Georgi	1953-09-02
Simmel Bezalel	1964-06-02
Bamford Parto	1959-12-03
Koblick Chirstian	1954-05-01
Maliniak Kyoichi	1955-01-21
Preusig Anneke	1953-04-20
Zielinski Tzvetan	1957-05-23
Kalloufi Saniya	1958-02-19
Peac Sumant	1952-04-19
Piveteau Duangkaew	1963-06-01
David Jiang	1972-02-29

下面是该解决方案的 SQL 查询：

```
SELECT name,birthday,
       IF(cur>today, cur,next) AS birth_day
FROM (
  SELECT name,birthday,today,
     DATE_ADD(cur, INTERVAL IF(DAY(birthday)=29
     && DAY(cur)=28,1,0) DAY) AS cur,
     DATE_ADD(next,INTERVAL IF(DAY(birthday)=29
     && DAY(next)=28,1,0) DAY) AS next
  FROM (
    SELECT name,birthday,today,
           DATE_ADD(birthday,INTERVAL diff YEAR) AS cur,
           DATE_ADD(birthday,INTERVAL diff+1 YEAR) AS next
    FROM (
      SELECT CONCAT(last_name,' ',first_name) AS Name,
             birth_date AS BirthDay,
             (YEAR(NOW())-YEAR(birth_date)) AS diff,
             NOW() AS today
      FROM employees ) AS a
    ) AS b
) AS c
```

这个查询需要 a、b、c 三个子查询来完成。第一个子查询 a 用来计算每位员工的出生日期与当前日期相差的年份，以及当前的日期。如果只运行子查询 a，将得到如表 2-3 所示的输出，假设当前的日期为“2011-02-04”。

要计算某员工最近的生日，需要在 BirthDay 列加上 Diff 列的年数。如果结果大于当前日期，则年龄需要再加一年。子查询 b 增加两列即 Cur 和 Next，这两列分别用于表示今年和

明年的生日。注意，如果出生日期是 2 月 29 日，且目标日期不是闰月，那么这两列所包含的将是 2 月 28 日，而不是 3 月 1 日。子查询 b 的结果如表 2-4 所示。

表 2-3　子查询 a 的结果

Name	BirthDay	Diff	Today
Facello Georgi	1953-09-02	58	2011-02-04
Simmel Bezalel	1964-06-02	47	2011-02-04
Bamford Parto	1959-12-03	52	2011-02-04
Koblick Chirstian	1954-05-01	57	2011-02-04
Maliniak Kyoichi	1955-01-21	56	2011-02-04
Preusig Anneke	1953-04-20	58	2011-02-04
Zielinski Tzvetan	1957-05-23	54	2011-02-04
Kalloufi Saniya	1958-02-19	53	2011-02-04
Peac Sumant	1952-04-19	59	2011-02-04
Piveteau Duangkaew	1963-06-01	48	2011-02-04
David Jiang	1972-02-29	39	2011-02-04

表 2-4　子查询 b 的结果

Name	BirthDay	Today	Cur	Next
Facello Georgi	1953-09-02	2011-02-04	2011-09-02	2012-09-02
Simmel Bezalel	1964-06-02	2011-02-04	2011-06-02	2012-06-02
Bamford Parto	1959-12-03	2011-02-04	2011-12-03	2012-12-03
Koblick Chirstian	1954-05-01	2011-02-04	2011-05-01	2012-05-01
Maliniak Kyoichi	1955-01-21	2011-02-04	2011-01-21	2012-01-21
Preusig Anneke	1953-04-20	2011-02-04	2011-04-20	2012-04-20
Zielinski Tzvetan	1957-05-23	2011-02-04	2011-05-23	2012-05-23
Kalloufi Saniya	1958-02-19	2011-02-04	2011-02-19	2012-02-19
Peac Sumant	1952-04-19	2011-02-04	2011-04-19	2012-04-19
Piveu Duangkaew	1963-06-01	2011-02-04	2011-06-01	2012-06-01
David Jiang	1972-02-29	2011-02-04	2011-02-28	2012-02-29

子查询 c 用来处理闰月的问题，如果出生的日期为闰月，并且当前的年份不是闰年，则日期加 1，表示 3 月 1 日为生日。对下一个年份使用同样的操作，子查询 c 的结果如表 2-5 所示。

表 2-5 子查询 c 的结果

Name	BirthDay	Today	Cur	Next
Facello Georgi	1953-09-02	2011-02-04	2011-09-02	2012-09-02
Simmel Bezalel	1964-06-02	2011-02-04	2011-06-02	2012-06-02
Bamford Parto	1959-12-03	2011-02-04	2011-12-03	2012-12-03
Koblick Chirstian	1954-05-01	2011-02-04	2011-05-01	2012-05-01
Maliniak Kyoichi	1955-01-21	2011-02-04	2011-01-21	2012-01-21
Preusig Anneke	1953-04-20	2011-02-04	2011-04-20	2012-04-20
Zielinski Tzvetan	1957-05-23	2011-02-04	2011-05-23	2012-05-23
Kalloufi Saniya	1958-02-19	2011-02-04	2011-02-19	2012-02-19
Peac Sumant	1952-04-19	2011-02-04	2011-04-19	2012-04-19
Pivetu Duangkaew	1963-06-01	2011-02-04	2011-06-01	2012-06-01
David Jiang	1972-02-29	2011-02-04	2011-03-01	2012-02-29

最后判断今年的生日是否已过，如果是，则返回下一年的生日，最后得到的查询结果如表 2-6 所示。

表 2-6 最后得到的查询结果

Name	BirthDay	Birth_Day
Facello Georgi	1953-09-02	2011-09-02
Simmel Bezalel	1964-06-02	2011-06-02
Bamford Parto	1959-12-03	2011-12-03
Koblick Chirstian	1954-05-01	2011-05-01
Maliniak Kyoichi	1955-01-21	2012-01-21
Preusig Anneke	1953-04-20	2011-04-20
Zielinski Tzvetan	1957-05-23	2011-05-23
Kalloufi Saniya	1958-02-19	2011-02-19
Peac Sumant	1952-04-19	2011-04-19
Piveteau Duangkaew	1963-06-01	2011-06-01
David Jiang	1972-02-29	2011-03-01

可以看到 Maliniak Kyoichi 今年的生日已过，下一个生日是 2012 年，而 David Jiang 的生日是 3 月 1 日。

2.4.2 重叠问题

很多与时间相关的查询问题都要求统计和标示出重叠的部分。下面将列举几个关于这些问题的示例。在这些示例中，需要使用 sessions 表，可以通过下列语句创建并填充该表。

```
# 建立 sessions 表
CREATE TABLE sessions
(
  id         INT          NOT NULL AUTO_INCREMENT,
  app        VARCHAR(10)  NOT NULL,
  usr        VARCHAR(10)  NOT NULL,
  starttime  TIME     NOT NULL,
  endtime    TIME     NOT NULL,
  PRIMARY KEY(id)
);

# 插入数据
INSERT INTO sessions(app, usr, starttime, endtime)
  VALUES('app1', 'user1', '08:30', '10:30');
INSERT INTO sessions(app, usr, starttime, endtime)
  VALUES('app1', 'user2', '08:30', '08:45');
INSERT INTO sessions(app, usr, starttime, endtime)
  VALUES('app1', 'user1', '09:00', '09:30');
INSERT INTO sessions(app, usr, starttime, endtime)
  VALUES('app1', 'user2', '09:15', '10:30');
INSERT INTO sessions(app, usr, starttime, endtime)
  VALUES('app1', 'user1', '09:15', '09:30');
INSERT INTO sessions(app, usr, starttime, endtime)
  VALUES('app1', 'user2', '10:30', '14:30');
INSERT INTO sessions(app, usr, starttime, endtime)
  VALUES('app1', 'user1', '10:45', '11:30');
INSERT INTO sessions(app, usr, starttime, endtime)
  VALUES('app1', 'user2', '11:00', '12:30');
INSERT INTO sessions(app, usr, starttime, endtime)
  VALUES('app2', 'user1', '08:30', '08:45');
INSERT INTO sessions(app, usr, starttime, endtime)
  VALUES('app2', 'user2', '09:00', '09:30');
INSERT INTO sessions(app, usr, starttime, endtime)
  VALUES('app2', 'user1', '11:45', '12:00');
INSERT INTO sessions(app, usr, starttime, endtime)
  VALUES('app2', 'user2', '12:30', '14:00');
INSERT INTO sessions(app, usr, starttime, endtime)
  VALUES('app2', 'user1', '12:45', '13:30');
INSERT INTO sessions(app, usr, starttime, endtime)
  VALUES('app2', 'user2', '13:00', '14:00');
INSERT INTO sessions(app, usr, starttime, endtime)
  VALUES('app2', 'user1', '14:00', '16:30');
INSERT INTO sessions(app, usr, starttime, endtime)
  VALUES('app2', 'user2', '15:30', '17:00');

# 创建索引
CREATE UNIQUE INDEX idx_app_usr_s_e_key
  ON sessions(app, usr, starttime, endtime, id);
CREATE INDEX idx_app_s_e ON sessions(app, starttime, endtime);
```

可以将这张 sessions 表理解为接入到互联网的会话表，如一些互联网提供商（ISP）按连接时间进行收费。该表的每行包含一个自增长的主键值、应用程序名（app）、用户名（usr）、开始时间（starttime），以及结束时间（endtime）。最后还为该表创建了两个索引以提高查询的性能。先来查看 sessions 表的内容，如表 2-7 所示。

表 2-7 sessions 表的内容

id	app	usr	starttime	endtime
1	app1	user1	08:30:00	10:30:00
2	app1	user2	08:30:00	08:45:00
3	app1	user1	09:00:00	09:30:00
4	app1	user2	09:15:00	10:30:00
5	app1	user1	09:15:00	09:30:00
6	app1	user2	10:30:00	14:30:00
7	app1	user1	10:45:00	11:30:00
8	app1	user2	11:00:00	12:30:00
9	app2	user1	08:30:00	08:45:00
10	app2	user2	09:00:00	09:30:00
11	app2	user1	11:45:00	12:00:00
12	app2	user2	12:30:00	14:00:00
13	app2	user1	12:45:00	13:30:00
14	app2	user2	13:00:00	14:00:00
15	app2	user1	14:00:00	16:30:00
16	app2	user2	15:30:00	17:00:00

对于 sessions 表，可以讨论 3 个涉及重叠的问题，分别是标示重叠、分组重叠及最大重叠会话数。

1. 标示重叠

标示重叠是指为每个会话标示出相同应用程序和用户重叠的会话，即对于每个会话，标示出其内部的所有会话情况。例如，app1、user1 这个会话在 08:30 ～ 10:30 时间段内有 3 次重叠的会话：08:30 ～ 10:30、09:00 ～ 09:30 和 09:15 ～ 09:30。对于这个问题，下面是完整的 SQL 解决方案：

```
SELECT
     a.app,a.usr,a.starttime,a.endtime,b.starttime,b.endtime
FROM sessions a,sessions b
WHERE a.app = b.app AND a.usr=b.usr
AND ( b.endtime >= a.starttime AND b.starttime <= a.endtime );
```

最终得到的结果如表 2-8 所示。

表 2-8 重叠的部分

app	usr	starttime	endtime	starttime	endtime
app1	user1	08:30:00	10:30:00	08:30:00	10:30:00
app1	user1	08:30:00	10:30:00	09:00:00	09:30:00
app1	user1	08:30:00	10:30:00	09:15:00	09:30:00
app1	user1	09:00:00	09:30:00	08:30:00	10:30:00
app1	user1	09:00:00	09:30:00	09:00:00	09:30:00
app1	user1	09:00:00	09:30:00	09:15:00	09:30:00
app1	user1	09:15:00	09:30:00	08:30:00	10:30:00
app1	user1	09:15:00	09:30:00	09:00:00	09:30:00
app1	user1	09:15:00	09:30:00	09:15:00	09:30:00
app1	user1	10:45:00	11:30:00	10:45:00	11:30:00
app1	user2	08:30:00	08:45:00	08:30:00	08:45:00
app1	user2	09:15:00	10:30:00	09:15:00	10:30:00
app1	user2	09:15:00	10:30:00	10:30:00	14:30:00
app1	user2	10:30:00	14:30:00	09:15:00	10:30:00
app1	user2	10:30:00	14:30:00	10:30:00	14:30:00
app1	user2	10:30:00	14:30:00	11:00:00	12:30:00
app1	user2	11:00:00	12:30:00	10:30:00	14:30:00
app1	user2	11:00:00	12:30:00	11:00:00	12:30:00
app2	user1	08:30:00	08:45:00	08:30:00	08:45:00
app2	user1	11:45:00	12:00:00	11:45:00	12:00:00
app2	user1	12:45:00	13:30:00	12:45:00	13:30:00
app2	user1	14:00:00	16:30:00	14:00:00	16:30:00
app2	user2	09:00:00	09:30:00	09:00:00	09:30:00
app2	user2	12:30:00	14:00:00	12:30:00	14:00:00
app2	user2	12:30:00	14:00:00	13:00:00	14:00:00
app2	user2	13:00:00	14:00:00	12:30:00	14:00:00
app2	user2	13:00:00	14:00:00	13:00:00	14:00:00
app2	user2	15:30:00	17:00:00	15:30:00	17:00:00

2. 分组重叠

还有一个问题是服务提供商可能允许多个 session 的连接，并把其计费统计为 1 次，这就是所谓的分组重叠。对于上面的例子，应该把 app1、user1 在 08:30 ～ 10:30 间的 3 次会话算为一次会话。

我们分步骤来讨论这个问题，先求出每个会话组开始时间，并用 DISTINCT 返回不重复的开始时间，具体过程如下：

```
SELECT DISTINCT app,usr,starttime AS s
  FROM
  sessions AS a
  WHERE NOT EXISTS (
    SELECT *
    FROM
    sessions AS b
    WHERE a.app = b.app
    AND a.usr = b.usr
    AND a.starttime > b.starttime
    AND a.starttime <= b.endtime
 );
```

得到的结果如表 2-9 所示。

表 2-9 每个会话开始的时间

app	usr	s
app1	user1	2010-02-12 08:30:00
app1	user1	2010-02-12 10:45:00
app1	user2	2010-02-12 08:30:00
app1	user2	2010-02-12 09:15:00
app2	user1	2010-02-12 08:30:00
app2	user1	2010-02-12 11:45:00
app2	user1	2010-02-12 12:45:00
app2	user1	2010-02-12 14:00:00
app2	user2	2010-02-12 09:00:00
app2	user2	2010-02-12 12:30:00
app2	user2	2010-02-12 15:30:00

用同样的方法得到会话组结束的时间，具体过程如下：

```
SELECT DISTINCT app,usr,endtime AS e
  FROM
  sessions AS a
 WHERE NOT EXISTS (
    SELECT *
    FROM
    sessions AS b
    WHERE a.app = b.app
    AND a.usr = b.usr
    AND a.endtime >= b.starttime
    AND a.endtime < b.endtime
 )
```

结果如表 2-10 所示。

表 2-10 每个会话结束的时间

app	usr	e
app1	user1	2010-02-12 10:30:00
app1	user1	2010-02-12 11:30:00
app1	user2	2010-02-12 08:45:00
app1	user2	2010-02-12 14:30:00
app2	user1	2010-02-12 08:45:00
app2	user1	2010-02-12 12:00:00
app2	user1	2010-02-12 13:30:00
app2	user1	2010-02-12 16:30:00
app2	user2	2010-02-12 09:30:00
app2	user2	2010-02-12 14:00:00
app2	user2	2010-02-12 17:00:00

最后只需把两张表合并，并通过 MIN 函数取得结束的时间。完整的 SQL 解决方案如下面的代码所示：

```
SELECT DISTINCT s.app,s.usr,s.s,
(
SELECT MIN(e)
FROM (
    SELECT DISTINCT app,usr,endtime AS e
        FROM
        sessions AS a
        WHERE NOT EXISTS (
          SELECT *
          FROM
          sessions AS b
          WHERE a.app = b.app
          AND a.usr = b.usr
          AND a.endtime >= b.starttime
          AND a.endtime < b.endtime
        )
) AS s2
WHERE s2.e>s.s AND s.app = s2.app AND s.usr=s2.usr ) AS e
FROM
(
  SELECT DISTINCT app,usr,starttime AS s
      FROM
      sessions AS a
      WHERE NOT EXISTS (
        SELECT *
        FROM
        sessions AS b
        WHERE a.app = b.app
        AND a.usr = b.usr
```

```
        AND a.starttime > b.starttime
        AND a.starttime <= b.endtime
  )
)
AS s,
(
    SELECT DISTINCT app,usr,endtime AS e
        FROM
        sessions AS a
        WHERE NOT EXISTS (
          SELECT *
          FROM
          sessions AS b
          WHERE a.app = b.app
          AND a.usr = b.usr
          AND a.endtime >= b.starttime
          AND a.endtime < b.endtime
        )
)
AS e
WHERE s.app = e.app AND s.usr=e.usr;
```

以上代码看起来非常复杂，因为需要使用多个子查询，实际上可以通过创建视图来行简化上述的 SQL 查询。可以进行这样的操作，将上述两个会话组开始和结束的临时表定义为视图，然后再进一步操作，如下所示：

```
CREATE VIEW v_s AS
SELECT DISTINCT app,usr,starttime AS s
  FROM
  sessions AS a
 WHERE NOT EXISTS (
    SELECT *
    FROM
    sessions AS b
    WHERE a.app = b.app
    AND a.usr = b.usr
    AND a.starttime > b.starttime
    AND a.starttime <= b.endtime
 );

CREATE VIEW v_e AS
SELECT DISTINCT app,usr,endtime AS e
  FROM
  sessions AS c
 WHERE NOT EXISTS (
    SELECT *
    FROM
    sessions AS d
    WHERE c.app = d.app
```

```
    AND c.usr = d.usr
    AND c.endtime >= d.starttime
    AND c.endtime < d.endtime
 );

SELECT DISTINCT s.app,s.usr,s.s,
(SELECT MIN(e) FROM v_e AS s2
WHERE s2.e>s.s AND s.app = s2.app AND s.usr=s2.usr )
FROM v_s AS s,v_e AS e
WHERE s.app = e.app AND s.usr=e.usr;
```

最后得到如表 2-11 所示的结果。

表 2-11　分组重叠的结果

app	usr	s	e
app1	user1	08:30:00	10:30:00
app1	user1	10:45:00	11:30:00
app1	user2	08:30:00	08:45:00
app1	user2	09:15:00	14:30:00
app2	user1	08:30:00	08:45:00
app2	user1	11:45:00	12:00:00
app2	user1	12:45:00	13:30:00
app2	user1	14:00:00	16:30:00
app2	user2	09:00:00	09:30:00
app2	user2	12:30:00	14:00:00
app2	user2	15:30:00	17:00:00

用视图作为派生表有一个缺点，那就是如果在一个查询中需要反复查询一个视图，那么这张视图需要被计算多次，上述 SQL 语句的执行计划如图 2-1 所示。

id	select_type	table	type	possible_keys	key	key_len	ref	ro	Extra
1	PRIMARY	<derived3>	ALL	NULL	NULL	NULL	NULL	11	Using temporary
1	PRIMARY	<derived7>	ALL	NULL	NULL	NULL	NULL	11	Using where; Distinct; Using join buffer
7	DERIVED	c	index	NULL	idx_app_usr_s_e_key	44	NULL	17	Using where; Using index; Using temporary
8	DEPENDENT SUBQUERY	d	ref	idx_app_usr_s_e_key,idx_app_s_e	idx_app_usr_s_e_key	24	test.c.app,test.c.usr	2	Using where; Using index
3	DERIVED	a	index	NULL	idx_app_usr_s_e_key	44	NULL	17	Using where; Using index
4	DEPENDENT SUBQUERY	b	ref	idx_app_usr_s_e_key,idx_app_s_e	idx_app_usr_s_e_key	24	test.a.app,test.a.usr	2	Using where; Using index
2	DEPENDENT SUBQUERY	<derived5>	ALL	NULL	NULL	NULL	NULL	11	Using where
5	DERIVED	c	index	NULL	idx_app_usr_s_e_key	44	NULL	17	Using where; Using index; Using temporary
6	DEPENDENT SUBQUERY	d	ref	idx_app_usr_s_e_key,idx_app_s_e	idx_app_usr_s_e_key	24	test.c.app,test.c.usr	2	Using where; Using index

图 2-1　SQL 执行计划

可以看到 v_e 这张视图被计算了两次，如果表中的记录非常大，那么需要提高视图的执行效率。怎么才能避免一个视图被查询多次呢？有一种方法是使用临时表，将 v_s、v_e 两张视图作为临时表。如果使用了临时表，那么将不再需要 MIN 函数的这个子查询，因为我们可以对临时表增加一个自增长的列，然后进行匹配即可。存储过程如下所示：

```
CREATE PROCEDURE pGroupOverlap ()
BEGIN
DROP TABLE IF EXISTS $s;
DROP TABLE IF EXISTS $e;

CREATE TEMPORARY TABLE $s (
 id INT AUTO_INCREMENT,
 app VARCHAR(10),
 usr VARCHAR(10),
 starttime DATETIME,
 PRIMARY KEY ( id ),
 KEY(usr,app)
)ENGINE=MEMORY;
CREATE TEMPORARY TABLE $e (
 id INT AUTO_INCREMENT,
 app VARCHAR(10),
 usr VARCHAR(10),
 endtime DATETIME,
 PRIMARY KEY ( id ),
 KEY(usr,app)
)ENGINE=MEMORY;

INSERT INTO $s(app,usr,starttime) (
  SELECT DISTINCT app,usr,starttime AS s
    FROM
    sessions AS a
    WHERE NOT EXISTS (
      SELECT *
      FROM
      sessions AS b
      WHERE a.app = b.app
      AND a.usr = b.usr
      AND a.starttime > b.starttime
      AND a.starttime <= b.endtime
    )
);
INSERT INTO $e(app,usr,endtime)(
  SELECT DISTINCT app,usr,endtime AS e
    FROM
    sessions AS c
   WHERE NOT EXISTS (
      SELECT *
      FROM
      sessions AS d
      WHERE c.app = d.app
      AND c.usr = d.usr
      AND c.endtime >= d.starttime
      AND c.endtime < d.endtime
   )
);
```

```
SELECT
   DISTINCT s.app,s.usr,starttime,endtime
FROM $s AS s,$e AS e
WHERE s.app = e.app AND s.usr=e.usr AND s.id=e.id;
DROP TABLE $s;
DROP TABLE $e;
END;
```

这次去掉了 MIN 函数，派生表从 3 个减少到两个。看似已经非常不错了，还能继续优化吗？应该是可以的。可不可以取消临时表的创建而达到同样的目的呢？是的，相信已经有人想到了，对两张临时表增加一个自增长的列，过程如下：

```
SET @A=0;
SET @B=0;
SELECT x.app,x.usr,s,e
FROM (
(SELECT @B:=@B+1 AS id,app,usr,s
FROM (
  SELECT DISTINCT app,usr,starttime AS s
    FROM
    sessions AS a
   WHERE NOT EXISTS (
      SELECT *
      FROM
      sessions AS b
      WHERE a.app = b.app
      AND a.usr = b.usr
      AND a.starttime > b.starttime
      AND a.starttime <= b.endtime
   )
) AS p) AS x,
(SELECT @A:=@A+1 AS id,app,usr,e
FROM (
SELECT DISTINCT app,usr,endtime AS e
  FROM
  sessions AS c
 WHERE NOT EXISTS (
    SELECT *
    FROM
    sessions AS d
    WHERE c.app = d.app
    AND c.usr = d.usr
    AND c.endtime >= d.starttime
    AND c.endtime < d.endtime
 )
) AS o) AS y
) WHERE x.id = y.id AND x.app = y.app AND x.usr = y.usr;
```

可以看到通过对每行添加一个伪列来记录行号，然后将两张派生表直接通过伪列进行匹

配，可以避免使用 MIN 函数，同时也减少了因为 MIN 函数带来的额外扫描开销。对于分组重叠这个问题，这里使用了 4 种方法来讨论，应该还存在更优的解决方案。这就是 SQL 编程的魅力，SQL 程序员可以不断地追求 SQL 编程之美。

3. 最大重叠会话数

这是要讨论的关于重叠的最后一个问题，其目的是为每个应用程序（这里不用考虑每个用户）返回并发会话的最大数。会话并发的最大数是指同一时间活动的会话的最多数量。一些服务提供商可能会根据并发连接的最大数进行授权许可并收费。这里假设，一个会话在另一个会话结束时开始这种情况不属于并发。

要解决这个问题，首先应该取出每个应用程序开始的时间，然后在外部查询中为每个开始时间统计这段时间内的并发数。使用的查询语句如下，查询结果如表 2-12 所示。

```
SELECT app, s,
(
SELECT COUNT(1) FROM sessions AS b
  WHERE s>=starttime AND s<endtime
) AS count
FROM (
SELECT DISTINCT app, starttime AS s
FROM
sessions
) AS a;
```

表 2-12 为每个开始时间统计这段时间内的并发数

app	s	count
app1	2010-02-12 08:30:00	3
app1	2010-02-12 09:00:00	3
app1	2010-02-12 09:15:00	5
app1	2010-02-12 10:30:00	1
app1	2010-02-12 10:45:00	2
app1	2010-02-12 11:00:00	3
app2	2010-02-12 08:30:00	3
app2	2010-02-12 09:00:00	3
app2	2010-02-12 11:45:00	3
app2	2010-02-12 12:30:00	2
app2	2010-02-12 12:45:00	3
app2	2010-02-12 13:00:00	4
app2	2010-02-12 14:00:00	2
app2	2010-02-12 15:30:00	2

前面提到过，如果一个会话正好在另一个会话结束时开始，则不属于并发，因此，这

里使用的谓词是 s>= starttime AND s < endtime，而不是 s BETWEEN starttime AND endtime。最后，在上一个查询的外面创建派生表，把数据按应用程序分组并返回每个应用程序的最大合计数。最终的 SQL 查询语句如下，执行结果如表 2-13 所示。

```
SELECT app,MAX(count) AS max
FROM (
  SELECT app,s,
  (
  SELECT COUNT(1) FROM sessions AS b
    WHERE a.app = b.app AND s>=starttime AND s<endtime
  ) AS count
  FROM (
  SELECT DISTINCT app,starttime AS s
  FROM
  sessions
  ) AS a
) AS bc
GROUP BY app ;
```

表 2-13 最大重叠会话数

app	max
app1	4
app2	3

对于最大重叠会话数的问题，这里使用了基于集合的方法，该解决方案的性能主要取决于 app、starttime 和 endtime 是否建立了索引、表的行数、平均重叠的会话数。实际上，就这个问题而言，使用基于游标的解决方案可以获得更好的性能（虽然这种情况并不常见）。我们将会在第 7 章讨论这种解决方案以及如何获得更好的性能。

2.4.3 星期数的问题

1. 计算日期是星期几

这个问题看上去非常简单，比如可以使用 MySQL 数据库内置的 WEEKDAY 函数来取得星期几。WEEKDAY 函数返回值为 0 ～ 6，0 代表 Monday，1 代表 Tuesday，……，6 代表 Sunday；也可以使用 DAYOFWEEK 函数，其返回值为 1 ～ 7，1 代表 Sunday，2 代表 Monday，……，7 代表 Saturday；还可以直接使用 DAYNAME 函数来返回日期具体的名称。这 3 个函数的使用如下：

```
mysql> SET @a='2011-01-01';
Query OK, 0 rows affected (0.00 sec)

mysql> SET WEEKDAY(@a),DAYOFWEEK(@a),DAYNAME(@a)\G;
```

```
*************************** 1. row ***************************
  WEEKDAY(@a): 5
DAYOFWEEK(@a): 7
  DAYNAME(@a): Saturday
1 row in set (0.00 sec)
```

WEEKDAY函数从周一开始，较符合中国人的习惯，但是周一的返回值是0。DAYOFWEEK函数从周日开始，更符合外国人的习惯。DAYNAME函数返回星期的名称，它和参数lc_time_names有所关联，该参数控制返回的日期显示方式，示例如下：

```
mysql> SELECT DAYNAME(NOW())\G;
*************************** 1. row ***************************
DAYNAME(NOW()): Sunday
1 row in set (0.00 sec)

mysql> SET lc_time_names='zh_CN';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT DAYNAME(NOW())\G;
*************************** 1. row ***************************
DAYNAME(NOW()): 星期日
1 row in set (0.00 sec)
```

这里要介绍另外一个有比较有趣的方法。如果知道“1900-01-01”是周一，那么“1900-01-02”就是周二，以此类推，如果用户想知道某个日期是否是周二，可以通过下面的方法：

```
mysql> SET @a='2011-01-01';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT DATEDIFF(@a,'1900-01-02')%7 = 0\G;
*************************** 1. row ***************************
DATEDIFF(@a,'1900-01-02')%7 = 0: 0
1 row in set (0.00 sec)

mysql> SELECT DATEDIFF(@a,'1900-01-06')%7 = 0\G;
*************************** 1. row ***************************
DATEDIFF(@a,'1900-01-06')%7 = 0: 1
1 row in set (0.00 sec)
```

如果日期是周二，意味着该日期与另一个星期二相差的天数可以被7整除。这里利用了参照日期法来进行判断，可以很容易地得知是否是我们想要的日期。

2. 按周分组

这个问题看似十分简单，却还有些小细节需要注意，这些小细节往往令人抓耳挠腮、烦恼不已。按周分组最容易遇到的问题是按周统计销量，如按周统计某个网络游戏的在线人数、充值记录等。一般的SQL程序员往往会使用WEEK这个函数，但是这个函数存在很多

问题。我们先来看下面这个问题：

```
mysql> SELECT
    WEEK('2011-01-01'),
    WEEK('2011-01-02'),
    WEEK('2011-01-03')\G;
*************************** 1. row ***************************
WEEK('2011-01-01'): 0
WEEK('2011-01-02'): 1
WEEK('2011-01-03'): 1
1 row in set (0.00 sec)
```

可以看到 WEEK 函数将 2011-01-01 视为第一周，而将 2011-01-02 和 2011-01-03 视为第二周。从中国人的理解角度来说，2011-01-01 是星期六，2011-01-01 和 2011-01-02 都应该算是第一周，2011-01-03 才是第二周的开始。造成这个问题的原因是 WEEK 这个函数是按照国外的习惯设计的，将周日视为每星期的开始。

因此 WEEK 函数并不是一个很好的做每周统计的方法。另外，如果某个周是跨年份的，如 2011-01-01，它是 2010 年到 2011 年跨年份的一周中的日期，WEEK 函数对此显得毫无办法，例如：

```
mysql> SELECT WEEK('2010-12-31'),WEEK('2011-01-01')\G;
*************************** 1. row ***************************
WEEK('2010-12-31'): 52
WEEK('2011-01-01'): 0
1 row in set (0.00 sec)
```

可以看到，MySQL 数据库将 2010-12-31 视为 2010 年的最后一周，而将 2011-01-01 视为 2011 年的第一周。此外，WEEK 函数还不能解决另一个问题，如果每周的报表要求按周二或周三开始，又应该怎么解决呢？

对于这个问题，我们也可以使用前面的参照法，参照 1900-01-01 是周一，DATEDIFF(date,'1900-01-01')/7=0 表示应该是在同一周。注意这里通过取整来判断是否在同一周，前面是取余操作，用来判断是星期几。如果我们的报表根据按周分组的要求从周二开始，那么我们只需将前面的判断修改为 DATEDIFF(date,'1900-01-02')/7=0 即可。

下面来看一个例子，先生成一些简单的测试数据，主要用来讨论某周跨年的分组问题。

```
USE test;
CREATE TABLE sales (
id   INT AUTO_INCREMENT NOT NULL,
date DATETIME NOT NULL,
cost INT UNSIGNED NOT NULL,
PRIMARY KEY(id) );

INSERT INTO sales(date,cost) VALUES ('2010-12-31',100);
INSERT INTO sales(date,cost) VALUES ('2011-01-01',200);
```

```
INSERT INTO sales(date,cost) VALUES ('2011-01-02',100);
INSERT INTO sales(date,cost) VALUES ('2011-01-06',100);
INSERT INTO sales(date,cost) VALUES ('2011-01-10',100);
```

再来看按周分组后的情况。

```
mysql> SELECT WEEK(date),SUM(cost)
    -> FROM sales
    -> GROUP BY WEEK(date);
+-------------+-----------+
| WEEK(date)  | SUM(cost) |
+-------------+-----------+
|           0 |       200 |
|           1 |       200 |
|           2 |       100 |
|          52 |       100 |
+-------------+-----------+
4 rows in set (0.00 sec)
```

正如我们所料，产生了错误，WEEK 函数把 2010-12-31 作为第 52 周，而 2010-12-31、2011-01-01 和 2011-01-02 应该属于同一周。这个问题的解决方案使用了前面介绍的参考法，具体方案如下：

```
mysql> SELECT FLOOR(DATEDIFF(date,'1900-01-01')/7) AS a,
    -> SUM(cost)
    -> FROM sales
    -> GROUP BY FLOOR(datediff(date,'1900-01-01')/7);
+------+-----------+
| a    | SUM(cost) |
+------+-----------+
| 5791 |       400 |
| 5792 |       100 |
| 5793 |       100 |
+------+-----------+
3 rows in set (0.00 sec)
```

这里我们需要使用 FLOOR 函数进行取整操作，得到的结果是正确的，但是 a 列返回的是距离 1900-01-01 的周数，显示不够直观。如果我们需要按照每周的开始和结束来显示，则对上述 SQL 语句再进行一些小修改：

```
mysql> SELECT
    -> DATE_ADD('1900-01-01',
    -> INTERVAL FLOOR(DATEDIFF(date,'1900-01-01')/7)*7 DAY)
       AS week_start,
    -> DATE_ADD('1900-01-01',
    -> INTERVAL FLOOR(DATEDIFF(date,'1900-01-01')/7)*7+6 DAY)
       AS week_end,
    -> SUM(cost)
    -> FROM sales
```

```
    -> GROUP BY FLOOR(datediff(date,'1900-01-01')/7);
+------------+------------+-----------+
| week_start | week_end   | sum(cost) |
+------------+------------+-----------+
| 2010-12-27 | 2011-01-02 |       400 |
| 2011-01-03 | 2011-01-09 |       100 |
| 2011-01-10 | 2011-01-16 |       100 |
+------------+------------+-----------+
3 rows in set (0.00 sec)
```

这里我们从周一开始统计每周的开始，如果要从周二或者周三开始统计每周的开始，那么只需要将 1900-01-01 修改为 1900-01-02 或者 1900-01-03 即可。

3. 计算工作日的问题

计算两个星期之间的工作日数量也是一个很常见的任务。我们可能需要包括假期和其他非工作日的特殊时间的辅助时间表。如果只把周末认为非工作日，那么根本就不需要辅助表，下面的存储过程用来计算 s 和 e 之间的工作日数。

```
CREATE PROCEDURE pGetWorkDays (s DATETIME, e DATETIME)
BEGIN
SELECT FLOOR(days/7)*5+days%7
- CASE WHEN 6 BETWEEN wd AND wd+days%7-1 THEN 1 ELSE 0 END
- CASE WHEN 7 BETWEEN wd AND wd+days%7-1 THEN 1 ELSE 0 END
FROM
(SELECT  DATEDIFF(e,s)+1 AS days,WEEKDAY(s)+1 AS wd ) AS a;
END;
```

这个解决方案的执行速度非常快，因为它不涉及任何的 I/O 操作。派生表查询计算 s 到 e 之间的天数（days），包括 s 和 e 在内，以及 s 是星期几（wd）。外部查询用于计算指定范围内的所有周的工作日数量（days/7*5）加上非完整一周内的工作日数量（days%7），如果非完整一周内的工作日包含周末，则减去相应的部分。下面我们来测试几个时间段内的工作日数量。

```
mysql> CALL pGetWorkDays('2005-01-01','2005-12-31') ;
+-----------+
| workdays  |
+-----------+
|       260 |
+-----------+
1 row in set (0.01 sec)

mysql> CALL pGetWorkDays('2005-01-01','2006-01-01') ;
+-----------+
| workdays  |
+-----------+
|       260 |
```

```
+----------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> CALL pGetWorkDays('2005-01-01','2006-01-02') ;
+----------+
| workdays |
+----------+
|      261 |
+----------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

可以看到2005-01-01到2005-12-31之间一共有260个工作日。因为2006-01-01是周日，所以2005-01-01到2006-01-01之间的工作日应该是260个，而2006-01-01是工作日，因此工作日数量变为261天。

2.5 数字类型

2.5.1 整型

MySQL数据库支持SQL标准支持的整型类型：INT、SMALLINT。此外MySQL数据库也支持诸如TINYINT、MEDIUMINT和BIGINT等类型。表2-14显示了各种整型占用的存储空间以及取值范围。

表2-14 整型类型占用空间和取值范围

类型	占用空间（字节）	最小值（Signed/Unsigned）	最大值(Signed/Unsigned)
TINYINT	1	−128	127
		0	255
SMALLINT	2	−32 768	32 767
		0	65 535
MEDIUMINT	3	−83 88 608	8 388 607
		0	16 777 215
INT	4	−2 147 483 648	2 147 483 647
		0	4 294 967 295
BIGINT	8	−9 223 372 036 854 775 808	9 223 372 036 854 775 807
		0	18 446 744 073 709 551 615

前面已经介绍过ZEROFILL可以格式化显示整型，这里还需要提及的是，一旦启用

ZEROFILL 属性，MySQL 数据库为列自动添加 UNSIGNED 属性，示例如下：

```
mysql> CREATE TABLE t ( a INT(4) ZEROFILL );
Query OK, 0 rows affected (0.04 sec)

mysql> SHOW CREATE TABLE t\G;
*************************** 1. row ***************************
       Table: t
Create Table: CREATE TABLE `t` (
  `a` int(4) unsigned zerofill DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
```

2.5.2 浮点型（非精确类型）

MySQL 数据库支持两种浮点类型：单精度的 FLOAT 类型及双精度的 DOUBLE PRECISION 类型。这两种类型都是非精确的类型，经过一些操作后并不能保证运算的正确性，例如 M*G/G 不一定等于 M，虽然数据库内部算法已经使其尽可能的正确，但是结果还会有偏差。国内某些财务软件在其数据库内使用 FLOAT 类型作为工资类型，个人觉得并不是一件值得推崇的事情。

FLOAT 类型用于表示近似数值数据类型。SQL 标准允许在关键字 FLOAT 后面的括号内用位来指定精度（但不能为指数范围）。MySQL 还支持可选的只用于确定存储大小的精度规定。0 到 23 的精度对应 FLOAT 列的 4 字节单精度，24 到 53 的精度对应 DOUBLE 列的 8 字节双精度。

MySQL 允许使用非标准语法：FLOAT（M,D）或 REAL（M,D）或 DOUBLE PRECISION（M,D）。这里，（M,D）表示该值一共显示 M 位整数，其中 D 位是小数点后面的位数。例如，定义为 FLOAT（7,4）的一个列可以显示为 −999.9999。MySQL 在保存值时会进行四舍五入，因此在 FLOAT（7,4）列内插入 999.00009 的近似结果是 999.0001。

MySQL 将 DOUBLE 视为 DOUBLE PRECISION（非标准扩展）的同义词，将 REAL 视为 DOUBLE PRECISION（非标准扩展）的同义词。若将 MySQL 服务器的模式设置为 REAL_AS_FLOAT，那这时 REAL 将被视为 FLOAT 类型。

为了保证最大的可移植性，需要使用近似数值数据值存储的代码，使用 FLOAT 或 DOUBLE PRECISION，并不规定精度或位数。

2.5.3 高精度类型

DECIMAL 和 NUMERIC 类型在 MySQL 中被视为相同的类型，用于保存必须为确切精度的值。对于前面提到的工资数据类型，当声明该类型的列时，可以（并且通常必须）指定精度和标度，例如：

```
salary DECIMAL(5,2)
```

在上述例子中，5是精度，2是标度。精度表示保存值的主要位数，标度表示小数点后面可以保存的位数。

在标准SQL中，语法DECIMAL（M）等价于DECIMAL（M,0）。在MySQL 5.5中M的默认值是10，例如：

```
mysql> CREATE TABLE t ( a DECIMAL);
Query OK, 0 rows affected (0.04 sec)

mysql> SHOW CREATE TABLE t\G;
*************************** 1. row ***************************
       Table: t
Create Table: CREATE TABLE `t` (
  `a` decimal(10,0) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
```

DECIMAL或NUMERIC的最大位数是65，但具体的DECIMAL或NUMERIC列的实际范围受具体列的精度或标度约束。如果分配给此类列的值的小数点后位数超过指定的标度允许的范围，值将按该标度进行转换。（具体操作与操作系统有关，一般结果均被截取到允许的位数）。

2.5.4 位类型

位类型，即BIT数据类型可用来保存位字段的值。BIT（M）类型表示允许存储M位数值，M范围为1到64，占用的空间为（M+7）/8字节。如果为BIT（M）列分配的值的长度小于M位，在值的左边用0填充。例如，为BIT（6）列分配一个值b'101'，其效果与分配b'000101'相同。要指定位值，可以使用b'*value*'符，例如：

```
mysql> CREATE TABLE t ( a BIT(4));
Query OK, 0 rows affected (0.07 sec)

mysql> INSERT INTO t SELECT b'1000';
Query OK, 1 row affected (0.02 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

但是直接用SELECT进行查看会出现如下情况：

```
mysql> SELECT * FROM t;
+------+
| a    |
+------+
|      |
+------+
1 row in set (0.00 sec)
```

这个值似乎是空的，其实不然，因为采用位的存储方式，所以不能直接查看，可能需要做类似如下的转化：

```
mysql> SELECT HEX(a) FROM t;
+---------+
| hex(a)  |
+---------+
| 8       |
+---------+
1 row in set (0.00 sec)
```

2.6 关于数字的经典 SQL 编程问题

2.6.1 数字辅助表

数字辅助表是一个只包含从 1 到 N 的 N 个整数的简单表，N 通常很大。因为数字辅助表是一个非常强大的工具，可能经常需要在解决方案中用到它，笔者建议创建一个持久的数字辅助表，并根据需要填充一定数据量的值。

实际上如何填充数字辅助表无关紧要，因为只需要运行这个过程一次。不过还可以对填充语句进行优化。一般的 SQL 编程人员会想到用如下方法来生成 1 ～ N 的数。

```
CREATE TABLE Nums(
   a INT UNSIGNED NOT NULL PRIMARY KEY
)ENGINE=InnoDB;

CREATE PROCEDURE pCreateNums (cnt INT UNSIGNED)
BEGIN
DECLARE s INT UNSIGNED DEFAULT 1;
TRUNCATE TABLE Nums;
WHILE s <= cnt DO
BEGIN
INSERT INTO Nums SELECT s;
SET s = s+1;
END;
END WHILE;
END;
```

这个方法没有任何的问题，只是效率不高。例如要插入 100 000 行的数据，在笔者的四核苹果电脑上至少需要 1 分钟。

```
mysql> CALL pCreateNums (100000);
Query OK, 1 row affected (1 min 11.56 sec)
```

这个方法的开销主要在于 INSERT 语句被执行了 100 000 次。我们可以通过下面这个方法来创建数字辅助表。

```
CREATE PROCEDURE pFastCreateNums (cnt INT UNSIGNED)
BEGIN
DECLARE s INT UNSIGNED DEFAULT 1;
TRUNCATE TABLE Nums;
INSERT INTO Nums SELECT s;
WHILE s*2 <= cnt DO
BEGIN
INSERT INTO Nums SELECT a+s FROM Nums;
SET s = s*2;
END;
END WHILE;
END;
```

在这个存储过程中，变量 s 保存插入该表的行数。该过程先把 1 插入数字辅助表，然后当 s*2<=cnt 成立时执行循环。在每次迭代中，该过程把数字辅助表当前所有行的值加上 s 后再插入数字辅助表中，即先插入 {1}，然后是 {2}，{3,4}，{5，6，7，8}，{9,10,11,12,13,14,15,16}，以此类推。因此这个存储过程的执行时间非常之快。要插入 200 000 行数据，情况如下：

```
mysql> CALL pFastCreateNums (200000);
Query OK, 65536 rows affected (1.00 sec)
```

可以看到执行时间缩短到了 1 秒钟，性能提高了 70 多倍。究其原因，是因为实际执行 INSERT 的次数少了。这里我们是按照 2 的指数次进行插入的，实际只执行了 17 次插入操作。这个解决方案的唯一缺点是，数字辅助表是按照 2 的指数次进行插入的，因此上述实际的插入行数是 131 072，而不是 200 000 行。查询一下刚才插入的数据，结果如下：

```
mysql> SELECT COUNT(1) FROM Nums;
+-----------+
| count(1) |
+-----------+
|    131072 |
+-----------+
1 row in set (0.03 sec)
```

不过这不是一个很大的问题，因为我们可以在取出数据时使用 <= 来截取指定的行数，如：

```
SELECT * FROM Nums WHERE a <= 100000
```

有了这张辅助表，用户可以通过它来辅助很多其他应用。例如，在数据仓库中，通常需要生成某个时间范围内的时间维度表，这时使用数字辅助表会非常简单和快捷，示例如下：

```
CREATE PROCEDURE pCreateDimTime(start DATE, end DATE)
BEGIN
SELECT DATE_ADD(start,INTERVAL a-1 DAY)
FROM Nums WHERE a<=DATEDIFF(end,start)+1;
END;
```

2.6.2 连续范围问题

连续范围问题也是一个非常经典的 SQL 编程问题。为了使讲解易于理解，我们先来创建一些测试数据。

```
CREATE TABLE t ( a INT UNSIGNED NOT NULL PRIMARY KEY );

INSERT INTO t VALUES(1);
INSERT INTO t VALUES(2);
INSERT INTO t VALUES(3);
INSERT INTO t VALUES(100);
INSERT INTO t VALUES(101);
INSERT INTO t VALUES(103);
INSERT INTO t VALUES(104);
INSERT INTO t VALUES(105);
```

可以看到 1 ～ 3 是连续的，100 ～ 101 是连续的，103 ～ 105 是连续的，那么怎么能得到如表 2-15 所示的结果呢？

表 2-15 连续范围

start_range	end_range
1	3
100	101
103	105

我们来看下面这句 SQL 语句及其返回的结果集。

```
mysql> SELECT a,@a:=@a+1 rn FROM t,(SELECT @a:=0) AS a;
+-----+------+
| a   | rn   |
+-----+------+
|   1 |    1 |
|   2 |    2 |
|   3 |    3 |
| 100 |    4 |
| 101 |    5 |
| 103 |    6 |
| 104 |    7 |
| 105 |    8 |
+-----+------+
8 rows in set (0.00 sec)
```

rn 列是人为计算出来的行号。是不是可以通过连续给出的行号来反映出连续范围的规律呢？如果还没有看出，那么再看下面这个 SQL 及它的返回结果集。

```
mysql> SELECT a,rn,a-rn
    -> FROM
```

```
    -> (SELECT a,@a:=@a+1 rn FROM t,(SELECT @a:=0) AS a)
    -> AS b;
+------+------+------+
| a    | rn   | a-rn |
+------+------+------+
|    1 |    1 |    0 |
|    2 |    2 |    0 |
|    3 |    3 |    0 |
|  100 |    4 |   96 |
|  101 |    5 |   96 |
|  103 |    6 |   97 |
|  104 |    7 |   97 |
|  105 |    8 |   97 |
+------+------+------+
8 rows in set (0.00 sec)
```

是的，在同一组连续值内，连续数值的差是一个常量，因为组内没有间断。当出现一个新组时，其列和行号之间的差值开始增大。所以对于连续范围的统计，我们可以根据差值来进行分组统计，具体过程如下：

```
mysql> SELECT MIN(a) start_range,MAX(a) end_range
    -> FROM
    -> (
    -> SELECT a,rn,a-rn AS diff
    -> FROM
    -> (SELECT a,@a:=@a+1 rn FROM t,(SELECT @a:=0) AS a)
    -> AS b
    ->) AS c
    -> GROUP BY diff
    -> ;
+-------------+-----------+
| start_range | end_range |
+-------------+-----------+
|           1 |         3 |
|         100 |       101 |
|         103 |       105 |
+-------------+-----------+
3 rows in set (0.00 sec)
```

在这里留给读者一个思考题，给出不连续的范围，也就是间断的范围，如何得到如表 2-16 所示的间断范围结果呢？

表 2-16 间断范围

start_range	end_range
4	99
102	102

2.7 字符类型

2.7.1 字符集

在讨论字符类型前，先讨论字符集的问题。笔者发现很多问题的产生都是因为开发人员忽视了字符集的问题，导致后期维护成本的提高。在 MySQL 数据库中，默认的字符集是 latin1，也许对于以英文为母语的大多数国家的 MySQL 应用来说，这个字符集并不会带来很大的问题。但是对于中文字符集，这可能会是一个灾难。

通过命令 SHOW CHARSET 可以查看 MySQL 数据库支持的字符集。MySQL 5.5 版本支持 39 个字符集，如表 2-17 所示。

表 2-17 MySQL 5.5 支持的 39 个字符集

Charset	Description	Default collation	Maxlen
big5	Big5 Traditional Chinese	big5_chinese_ci	2
dec8	DEC West European	dec8_swedish_ci	1
cp850	DOS West European	cp850_general_ci	1
hp8	HP West European	hp8_english_ci	1
koi8r	KOI8-R Relcom Russian	koi8r_general_ci	1
latin1	cp1252 West European	latin1_swedish_ci	1
latin2	ISO 8859-2 Central European	latin2_general_ci	1
swe7	7bit Swedish	swe7_swedish_ci	1
ascii	US ASCII	ascii_general_ci	1
ujis	EUC-JP Japanese	ujis_japanese_ci	3
sjis	Shift-JIS Japanese	sjis_japanese_ci	2
hebrew	ISO 8859-8 Hebrew	hebrew_general_ci	1
tis620	TIS620 Thai	tis620_thai_ci	1
euckr	EUC-KR Korean	euckr_korean_ci	2
koi8u	KOI8-U Ukrainian	koi8u_general_ci	1
gb2312	GB2312 Simplified Chinese	gb2312_chinese_ci	2
greek	ISO 8859-7 Greek	greek_general_ci	1
cp1250	Windows Central European	cp1250_general_ci	1
gbk	GBK Simplified Chinese	gbk_chinese_ci	2
latin5	ISO 8859-9 Turkish	latin5_turkish_ci	1
armscii8	ARMSCII-8 Armenian	armscii8_general_ci	1
utf8	UTF-8 Unicode	utf8_general_ci	3
ucs2	UCS-2 Unicode	ucs2_general_ci	2
cp866	DOS Russian	cp866_general_ci	1

（续）

Charset	Description	Default collation	Maxlen
keybcs2	DOS Kamenicky Czech-Slovak	keybcs2_general_ci	1
macce	Mac Central European	macce_general_ci	1
macroman	Mac West European	macroman_general_ci	1
cp852	DOS Central European	cp852_general_ci	1
latin7	ISO 8859-13 Baltic	latin7_general_ci	1
utf8mb4	UTF-8 Unicode	utf8mb4_general_ci	4
cp1251	Windows Cyrillic	cp1251_general_ci	1
utf16	UTF-16 Unicode	utf16_general_ci	4
cp1256	Windows Arabic	cp1256_general_ci	1
cp1257	Windows Baltic	cp1257_general_ci	1
utf32	UTF-32 Unicode	utf32_general_ci	4
binary	Binary pseudo charset	binary	1
geostd8	GEOSTD8 Georgian	geostd8_general_ci	1
cp932	SJIS for Windows Japanese	cp932_japanese_ci	2
eucjpms	UJIS for Windows Japanese	eucjpms_japanese_ci	3

第一列是字符集的名称，第三列是排序规则，这两列是2.7.2节会重点介绍的内容。最后一列代表这个字符集的1个字符可能占用的最大字节空间，单位是字节（Byte）。对于简体中文，我们习惯使用gbk或gb2312，这两者的区别是：gbk是gb2312的超集，因此可以支持更多的汉字。不过当前的MySQL不支持中文字符集gb18030，因此在有些应用中已经出现gbk不能显示特定中文字体的情况了。而对于繁体中文，big5可能是首选的字符集。

上面叙述的字符集gbk、gb2312和big5都是假设用户的应用程序只涉及这些字符范围所规定的字符，不需要额外的字符。而对于Facebook、Google、Yahoo和网易这些国际化公司，需要存储的字符可能是多种多样的，因此要使用Unicode编码的字符集。另外，为了平台的扩展性，有必要将字符集设置为Unicode编码。例如，笔者之前所在的网游公司成功地将游戏输出到韩国，并在当地取得了不小的成功，不过移植到韩国版本时，程序员花费了很大的精力来修改原字符集所带来的问题。因为原来使用的字符集是gbk，显然不能满足韩国玩家的要求。如果最初设计时采用utf8字符集，就不会有这个烦恼了。

此外，很多程序员和DBA不能区分Unicode和utf8的区别，他们总是认为两者是等价的，即Unicode就是utf8，utf8就是Unicode。其实，两者还是存在很大区别的。这里介绍一下这两者的区别。

Unicode是一种在计算机上使用的字符编码。它为每种语言中的每个字符设定了统一且唯一的二进制编码，以满足跨语言和跨平台进行文本转换和处理的要求。Unicode是1990年开始研发，1994年正式公布的。随着计算机工作能力的增强，Unicode在面世后的十多年里

得到普及。需要注意的是，Unicode 是字符编码，不是字符集。

Unicode 是基于通用字符集（Universal Character Set）的标准进行发展的，同时以书本的形式（The Unicode Standard，目前第五版由 Addison-Wesley Professional 出版）对外发表。2006 年 7 月的最新版 Unicode 是 5.0 版本。

Unicode 是国际组织制定的可以容纳世界上所有文字和符号的字符编码方案。Unicode 用数字 0 ～ 0x10FFFF 来映射这些字符，最多可以容纳 1 114 112 个字符，或者说有 1 114 112 个码位。码位就是可以分配给字符的数字。utf8、utf16 和 utf32 都是将数字转换到程序数据的编码方案。

我们可以通过下面的 SQL 语句来查询 MySQL 支持的 Unicode 编码的字符集。

```
mysql> SELECT * FROM
           CHARACTER_SETS
           WHERE DESCRIPTION like '%Unicode'\G;
*************************** 1. row ***************************
  CHARACTER_SET_NAME: utf8
DEFAULT_COLLATE_NAME: utf8_general_ci
         DESCRIPTION: UTF-8 Unicode
              MAXLEN: 3
*************************** 2. row ***************************
  CHARACTER_SET_NAME: ucs2
DEFAULT_COLLATE_NAME: ucs2_general_ci
         DESCRIPTION: UCS-2 Unicode
              MAXLEN: 2
*************************** 3. row ***************************
  CHARACTER_SET_NAME: utf8mb4
DEFAULT_COLLATE_NAME: utf8mb4_general_ci
         DESCRIPTION: UTF-8 Unicode
              MAXLEN: 4
*************************** 4. row ***************************
  CHARACTER_SET_NAME: utf16
DEFAULT_COLLATE_NAME: utf16_general_ci
         DESCRIPTION: UTF-16 Unicode
              MAXLEN: 4
*************************** 5. row ***************************
  CHARACTER_SET_NAME: utf32
DEFAULT_COLLATE_NAME: utf32_general_ci
         DESCRIPTION: UTF-32 Unicode
              MAXLEN: 4
5 rows in set (0.00 sec)
```

MySQL 5.5 数据库共支持 ucs2、utf8（utf8mb3）、utf8mb4、utf16，以及 utf32 五种 Unicode 编码，而 5.5 之前的版本只支持 ucs2 和 utf8 两种 Unicode 字符集，即只支持 Unicode 3.0 的标准。显然 MySQL 5.5 版本对于 Unicode 标准已经支持到 5.0。utf8 目前被视为 utf8mb3，即最大占用 3.个字节空间，而 utf8mb4 可以视做 utf8mb3 的扩展。对 BMP

（Basic Multilingual Plane）字符的存储，utf8mb3 和 utf8mb4 两者是完全一样的，区别只是 utf8mb4 对扩展字符的支持。

对于 Unicode 编码的字符集，强烈建议将所有的 CHAR 字段设置为 VARCHAR 字段，因为对于 CHAR 字段，数据库会保存最大可能的字节数。例如，对于 CHAR（30），数据库可能存储 90 字节的数据。

对字符集的设置可以在 MySQL 的配置文件中完成，例如：

```
[mysqld]
default-character-set=utf8
```

MySQL 5.5 版本开始移除了参数 default_character_set，取而代之的是参数 character_set_server，因此在配置文件中需进行如下设置：

```
[mysqld]
character_set_server=utf8
```

要查看当前使用的字符集，可以使用 STATUS 命令：

```
mysql> STATUS;
--------------
mysql  Ver 14.14 Distrib 5.5.8, for osx10.6 (i386) using readline 5.1

Connection id:          1
Current database:
Current user:           root@localhost
SSL:                    Not in use
Current pager:          stdout
Using outfile:          ''
Using delimiter:  ;
Server version:         5.5.8 MySQL Community Server (GPL)
Protocol version: 10
Connection:             Localhost via UNIX socket
Server characterset:    utf8
Db     characterset:    utf8
Client characterset:    utf8
Conn.  characterset:    utf8
UNIX socket:            /tmp/mysql.sock
Uptime:                 2 min 49 sec

Threads: 3  Questions: 30  Slow queries: 0  Opens: 33  Flush tables: 1  Open
    tables: 26  Queries per second avg: 0.177
```

命令 SET NAMES 可以用来更改当前会话连接的字符集、当前会话的客户端的字符集，以及当前会话返回结果集的字符集，例如：

```
mysql> SET NAMES 'gbk';
Query OK, 0 rows affected (0.00 sec)
```

MySQL 数据库一个比较“强悍”的地方是，可以细化每个对象字符集的设置，例如：

```
mysql> CREATE TABLE t (
    -> a VARCHAR(10) CHARSET gbk,
    -> b VARCHAR (10) CHARSET latin1,
    -> c VARCHAR (10) ) CHARSET=utf8;
Query OK, 0 rows affected (0.07 sec)
```

可以看到，我们创建了表 t，a 列的字符集是 gbk，b 列的字符集是 latin1，c 列的字符集没有在定义时指定，因此采用定义表时的字符集，这里是 utf8。如果定义表时没有给出具体的字符集，则采用创建架构（Schema，也称库）时指定的字符集。如果没有在创建架构时指定字符集，则使用数据库配置文件中指定的字符集。创建架构的时候可以指定架构的字符集及排序规则，过程如下：

```
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name
[create_specification] ...
create_specification:
[DEFAULT] CHARACTER SET [=] charset_name
| [DEFAULT] COLLATE [=] collation_name
```

SQL 标准支持 NCHAR 类型（即 National Character Set），在 MySQL 5.5 数据库中使用 utf8 来表示这种类型，例如：

```
mysql> CREATE TABLE t( a NCHAR(10))CHARSET=gbk;
Query OK, 0 rows affected (0.04 sec)

mysql> SHOW CREATE TABLE t\G;
*************************** 1. row ***************************
       Table: t
Create Table: CREATE TABLE `t` (
  `a` char(10) CHARACTER SET utf8 DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=gbk
1 row in set (0.00 sec)
```

可以看到，我们在创建表时将 a 列定义为 NCHAR（10），但是在创建数据库过程中实际使用的是 utf8 字符集。在别的数据库中，如 Microsoft SQL Server，NCHAR 可能被视为 ucs2 字符集。我们在客户端使用 N 前缀将字符串指定为 NCHAR 类型，也就是 UTF-8 类型，例如：

```
SELECT N'我们'
```

2.7.2 排序规则

排序规则（Collation）是指对指定字符集下不同字符的比较规则。其特征有以下几点：

- 两个不同的字符集不能有相同的排序规则。
- 每个字符集有一个默认的排序规则。
- 有一些常用的命名规则。如 _ci 结尾表示大小写不敏感（case insensitive），_cs 表示

大小写敏感（case sensitive），_bin 表示二进制的比较（binary）。

在 MySQL 数据库中，可以通过命令 SHOW COLLATION 来查看支持的各种排序规则，也可以通过 information_schema 架构下的表 COLLATIONS 来查看，例如：

```
mysql> SELECT COLLATION_NAME,CHARACTER_SET_NAME
    ->FROM COLLATIONS\G;
*************************** 1. row ***************************
    COLLATION_NAME: big5_chinese_ci
CHARACTER_SET_NAME: big5
*************************** 2. row ***************************
    COLLATION_NAME: big5_bin
CHARACTER_SET_NAME: big5
*************************** 3. row ***************************
    COLLATION_NAME: dec8_swedish_ci
CHARACTER_SET_NAME: dec8
*************************** 4. row ***************************
    COLLATION_NAME: dec8_bin
CHARACTER_SET_NAME: dec8
*************************** 5. row ***************************
    COLLATION_NAME: cp850_general_ci
CHARACTER_SET_NAME: cp850
......
195 rows in set (0.00 sec)
```

MySQL 5.5 数据库支持 195 种排序规则，这里只列出了一部分排序规则。当然，也可以通过 SHOW COLLATION LIKE 命令来过滤想要查看的排序规则，过程如下：

```
mysql> SHOW COLLATION LIKE 'gbk%'\G;
*************************** 1. row ***************************
Collation: gbk_chinese_ci
  Charset: gbk
       Id: 28
  Default: Yes
 Compiled: Yes
  Sortlen: 1
*************************** 2. row ***************************
Collation: gbk_bin
  Charset: gbk
       Id: 87
  Default:
 Compiled: Yes
  Sortlen: 1
2 rows in set (0.00 sec)
```

可以看到，对于 gbk 字符集，有两种排序规则，分别是 gbk_chinese_ci 和 gbk_bin。前面介绍的命令 SHOW CHARSET 已经显示了每种字符集的默认排序规则，如果用户需要查看 gbk 字符集的默认排序规则，那么可以使用如下命令：

```
mysql> SHOW CHARSET LIKE 'gbk%'\G;
```

```
*************************** 1. row ***************************
          Charset: gbk
      Description: GBK Simplified Chinese
Default collation: gbk_chinese_ci
           Maxlen: 2
1 row in set (0.00 sec)
```

前面介绍了一些命令用来查看当前 MySQL 支持的排序规则，那排序规则到底有什么用，它对我们的 SQL 编程又会产生怎样的影响呢？下面通过一个例子来说明排序规则的重要性。首先创建一张表，再插入两条数据，过程如下：

```
mysql> CREATE TABLE t (
    -> a VARCHAR(10)
    -> )CHARSET=utf8;
Query OK, 0 rows affected (0.03 sec)

mysql> INSERT INTO t SELECT 'a';
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> INSERT INTO t SELECT 'A';
Query OK, 1 row affected (0.02 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

表 t 非常简单，只有一个列 a 是 VARCHAR 类型的。设置表的字符集为 utf8，因此列 a 的字符集也是 utf8。我们插入了两条数据，“a” 和 “A”，然后执行这样的 SQL 查询：

```
mysql>select * from t where a='a';
+------+
| a    |
+------+
| a    |
| A    |
+------+
2 rows in set (0.00 sec)
```

执行 SQL 查询语句的目的可能只是要寻找小写字符 “a”，但是现在返回的却是两条记录，大写字符 “A” 也在返回结果集中。导致这个问题的原因是 utf8 字符集默认的排序规则是 utf8_general_ci。之前介绍过 _ci 结尾表示大小写不敏感，因此这个示例中的 “a” 和 “A” 被视为一致的字符而返回。在命令行中可以直接对这两个字符进行比较，例如：

```
mysql> SET NAMES utf8;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT 'a' = 'A';
+-----------+
| 'a' = 'A' |
+-----------+
|         1 |
```

```
+-----------+
1 row in set (0.00 sec)
```

可以看到返回值为1，即认为两个字符的比较结果是相等的。如果需要更改当前会话的排序规则，可以通过命令 SET NAMES... COLLATE...来实现，例如：

```
mysql> SET NAMES utf8 COLLATE utf8_bin;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT 'a' = 'A';
+-----------+
| 'a' = 'A' |
+-----------+
|         0 |
+-----------+
1 row in set (0.00 sec)
```

可以看到，这时数据库会认为“a”和“A”是不同的字符。对于创建的表t如果需要对a列区分大小字符，则可以将a列的排序规则修改为utf8_bin，例如：

```
mysql> ALTER TABLE t
    -> MODIFY COLUMN a VARCHAR(10) COLLATE UTF8_BIN;;
Query OK, 2 rows affected (0.05 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM t where a = 'a';
+------+
| a    |
+------+
| a    |
+------+
1 row in set (0.00 sec)
```

大小写敏感的需求要视应用程序的需求而定，并不总是要求区分大小写字符。例如，对于用户信息表，可能希望在用户注册时不要区分大小写，因为如果区分大小写，那么一个用户可以注册为David，另一个用户可以注册为david或者DaVid。这可能并不是我们通常想看到的一种情况。另外，排序规则不仅影响大小写的比较问题，也影响着索引。例如我们将t表的a列修改为之前的定义，然后再创建一个a列上的唯一索引。

```
mysql> ALTER TABLE t MODIFY COLUMN
    -> a VARCHAR(10) COLLATE UTF8_GENERAL_CI;;
Query OK, 0 rows affected (0.04 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> ALTER TABLE t ADD UNIQUE KEY (a);
ERROR 1062 (23000): Duplicate entry 'a' for key 'a'
```

可以看到，不能在a列上创建一个唯一索引，报错中提示有重复数据。索引是B+树，同

样需要对字符进行比较，因此在建立唯一索引时由于排序规则对大小写不敏感而导致了错误。

在 MySQL 数据库中还有一种被称为 binary 的排序规则，其和 _bin 的排序规则大致相同，有些小的区别读者可以在 MySQL 官方手册中进行查找。

2.7.3 CHAR 和 VARCHAR

CHAR 和 VARCHAR 是最常使用的两种字符串类型。一般来说，CHAR（N）用来保存固定长度的字符串，VARCHAR（N）用来保存变长字符类型。对于 CHAR 类型，N 的范围为 0 ～ 255，对于 VARCHAR 类型，N 的范围为 0 ～ 65 535。CHAR（N）和 VARCHAR（N）中的 N 都代表字符长度，而非字节长度。

注意 对于 MySQL 4.1 之前的版本，如 MySQL 3.23 和 MySQL 4.0，CHAR（N）和 VARCHAR（N）中的 N 代表字节长度。

对于 CHAR 类型的字符串，MySQL 数据库会自动对存储列的右边进行填充（Right Padded）操作，直到字符串达到指定的长度 N。而在读取该列时，MySQL 数据库会自动将填充的字符删除。有一种情况例外，那就是显式地将 SQL_MODE 设置为 PAD_CHAR_TO_FULL_LENGTH，例如：

```
mysql> CREATE TABLE t ( a CHAR(10));
Query OK, 0 rows affected (0.03 sec)

mysql> INSERT INTO t SELECT 'abc';
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT a,HEX(a),LENGTH(a) FROM t\G;
*************************** 1. row ***************************
        a: abc
   HEX(a): 616263
LENGTH (a): 3
1 row in set (0.00 sec)

mysql> SET SQL_MODE='PAD_CHAR_TO_FULL_LENGTH';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT a,HEX(a),LENGTH(a) FROM t\G;
*************************** 1. row ***************************
        a: abc
   HEX(a): 61626320202020202020
LENGTH (a): 10
1 row in set (0.00 sec)
```

在上述这个例子中，先创建了一张表 t，a 列的类型为 CHAR（10）。然后通过 INSERT

语句插入值“abc”，因为a列的类型为CHAR型，所以会自动在后面填充空字符串，使其长度为10。接下来在通过SELECT语句取出数据时会将a列右填充的空字符移除，从而得到值“abc”。通过LENGTH函数看到a列的字符长度为3而非10。

接着我们将SQL_MODE显式地设置为PAD_CHAR_TO_FULL_LENGTH。这时再通过SELECT语句进行查询时，得到的结果是“abc　　”，abc右边有7个填充字符0x20，并通过HEX函数得到了验证。这次LENGTH函数返回的长度为10。需要注意的是，LENGTH函数返回的是字节长度，而不是字符长度。对于多字节字符集，CHAR（N）长度的列最多可占用的字节数为该字符集单字符最大占用字节数*N。例如，对于utf8下，CHAR（10）最多可能占用30个字节。通过对多字节字符串使用CHAR_LENGTH函数和LENGTH函数，可以发现两者的不同，示例如下：

```
mysql> SET NAMES gbk;
Query OK, 0 rows affected (0.03 sec)

mysql> SELECT @a:='MySQL技术内幕';
Query OK, 0 rows affected (0.03 sec)

mysql> SELECT @a,HEX(@a),LENGTH(@a),CHAR_LENGTH(@a)\G;
*************************** 1. row ***************************
a: MySQL技术内幕
HEX(a): 4D7953514CBCBCCAF5C4DAC4BB
LENGTH (a): 13
CHAR_LENGTH(a): 9
1 row in set (0.00 sec)
```

变量@a是gbk字符集的字符串类型，值为“MySQL技术内幕”，十六进制为0x4D7953514CBCBCCAF5C4DAC4BB，LENGTH函数返回13，即该字符串占用13字节，因为gbk字符集中的中文字符占用两个字节，因此一共占用13字节。CHAR_LENGTH函数返回9，很显然该字符长度为9。

VARCHAR类型存储变长字段的字符类型，与CHAR类型不同的是，其存储时需要在前缀长度列表加上实际存储的字符，该字符占用1～2字节的空间。当存储的字符串长度小于255字节时，其需要1字节的空间，当大于255字节时，需要2字节的空间。所以，对于单字节的latin1来说，CHAR（10）和VARCHAR（10）最大占用的存储空间是不同的，CHAR（10）占用10个字节这是毫无疑问的，而VARCHAR（10）的最大占用空间数是11字节，因为其需要1字节来存放字符长度。

注意 对于有些多字节的字符集类型，其CHAR和VARCHAR在存储方法上是一样的，同样需要为长度列表加上字符串的值。对于GBK和UTF-8这些字符类型，其有些字符是以1字节存放的，有些字符是按2或3字节存放的，因此同样需要1～2字节的空间来存储字符的长度。这在《MySQL技术内幕：InnoDB存储引擎》一书已经进行了非常详细的阐述和证明。

虽然 CHAR 和 VARCHAR 的存储方式不太相同，但是对于两个字符串的比较，都只比较其值，忽略 CHAR 值存在的右填充，即使将 SQL _MODE 设置为 PAD_CHAR_TO_FULL_LENGTH 也一样，例如：

```
mysql> CREATE TABLE t ( a CHAR(10), b VARCHAR(10));
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO t SELECT 'a','a';
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT a=b FROM t\G;
*************************** 1. row ***************************
a=b: 1
1 row in set (0.00 sec)

mysql> SET SQL_MODE='PAD_CHAR_TO_FULL_LENGTH';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT a=b FROM t\G;
*************************** 1. row ***************************
a=b: 1
1 row in set (0.00 sec)
```

2.7.4 BINARY 和 VARBINARY

BINARY 和 VARBINARY 与前面介绍的 CHAR 和 VARCHAR 类型有点类似，不同的是 BINARY 和 VARBINARY 存储的是二进制的字符串，而非字符型字符串。也就是说，BINARY 和 VARBINARY 没有字符集的概念，对其排序和比较都是按照二进制值进行对比。

BINARY（N）和 VARBINARY（N）中的 N 指的是字节长度，而非 CHAR（N）和 VARCHAR（N）中的字符长度。对于 BINARY（10），其可存储的字节固定为 10，而对于 CHAR（10），其可存储的字节视字符集的情况而定。我们来看下面的例子。

```
mysql> CREATE TABLE t (
    -> a BINARY(1)
    -> )ENGINE=InnoDB CHARSET=GBK;
Query OK, 0 rows affected (0.02 sec)

mysql> SET NAMES GBK;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t SELECT '我';
Query OK, 1 row affected, 1 warning (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 1
```

```
mysql> SHOW WARNINGS\G;
*************************** 1. row ***************************
  Level: Warning
   Code: 1265
Message: Data truncated for column 'a' at row 1
1 row in set (0.00 sec)

mysql> SELECT a,HEX(a) FROM t\G;
*************************** 1. row ***************************
     a:
HEX(a): CE
```

表t包含一个类型为BINARY（1）的列，因为BINARY（N）中N代表字节，而gbk字符集中的中文字符“我”需要占用2字节，所以在插入时给出警告，提示字符被截断。如果SQL_MODE为严格模式，则会直接报错。查看表t的内容，则可发现a中只存储了字符“我”的前一个字节，后一个字节被截断了。如果表t的a列中字符的类型为CHAR类型，则完全不会有上述问题，例如：

```
mysql> CREATE TABLE t (
    -> a CHAR(1)
    -> )ENGINE=InnoDB CHARSET=GBK;
Query OK, 0 rows affected (0.02 sec)

mysql> INSERT INTO t SELECT '我';
Query OK, 1 row affected, 1 warning (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT a,HEX(a) FROM t\G;
*************************** 1. row ***************************
     a: 我
HEX(a): CED2
1 row in set (0.00 sec)
```

BINARY和VARBINARY对比CHAR和VARCHAR，第一个不同之处就是BINARY（N）和VARBINARY（N）中的N值代表的是字节数，而非字符长度；第二个不同点是，CHAR和VARCHAR在进行字符比较时，比较的只是字符本身存储的字符，忽略字符后的填充字符，而对于BINARY和VARBINARY来说，由于是按照二进制值来进行比较的，因此结果会非常不同，例如：

```
mysql> SELECT
    -> HEX('a'),
    -> HEX('a  '),
    -> 'a'='a   '\G;
*************************** 1. row ***************************
HEX('a'): 61
HEX('a  '): 612020
'a'='a   ': 1
```

```
1 row in set (0.00 sec)

mysql> SELECT
    -> HEX(BINARY('a')),
    -> HEX(BINARY('a  ')),
    -> BINARY('a')= BINARY('a   ')\G;
*************************** 1. row ***************************
          HEX(BINARY('a')): 61
        HEX(BINARY('a  ')): 612020
BINARY('a')= BINARY('a   '): 0
1 row in set (0.00 sec)
```

对于 CHAR 和 VARCHAR 来说，比较的是字符值，因此第一个比较的返回值是 1。对于 BINARY 和 VARBINARY 来说，比较的是二进制的值，“a”的十六进制为 61，“a ”的十六进制为 612020，显然不同，因此第二个比较的返回值为 0。

第三个不同的是，对于 BINARY 字符串，其填充字符是 0x00，而 CHAR 的填充字符为 0x20。可能是因为 BINARY 的比较需要，0x00 显然是比较的最小字符，示例如下：

```
mysql> CREATE TABLE t ( a BINARY(3));
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t SELECT 'a';
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT a,HEX(a) FROM t\G;
*************************** 1. row ***************************
     a: a
HEX(a): 610000
1 row in set (0.00 sec)
```

2.7.5 BLOB 和 TEXT

BLOB（Binary Large Object）是用来存储二进制大数据类型的。根据存储长度的不同 BLOB 可细分为以下 4 种类型，括号中的数代表存储的字节数：

- TINYBLOB（2^8）
- BLOB（2^{16}）
- MEDIUMBLOB（2^{24}）
- LONGBLOB（2^{32}）

TEXT 类型同 BLOB 一样，细分为以下 4 种类型：

- TINYTEXT（2^8）
- TEXT（2^{16}）
- MEDIUMTEXT（2^{24}）
- LONGTEXT（2^{32}）

在大多数情况下，可以将 BLOB 类型的列视为足够大的 VARBINARY 类型的列。同样，也可以将 TEXT 类型的列视为足够大的 VARCHAR 类型的列。然而，BLOB 和 TEXT 在以下几个方面又不同于 VARBINARY 和 VARCHAR：

- 在 BLOB 和 TEXT 类型的列上创建索引时，必须制定索引前缀的长度。而 VARCHAR 和 VARBINARY 的前缀长度是可选的。
- BLOB 和 TEXT 类型的列不能有默认值。
- 在排序时只使用列的前 max_sort_length 个字节。

max_sort_length 默认值为 1024，该参数是动态参数，任何客户端都可以在 MySQL 数据库运行时更改该参数的值，例如：

```
mysql> SET GLOBAL max_sort_length=2048;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @@global.max_sort_length\G;
*************************** 1. row ***************************
@@global.max_sort_length: 2048
1 row in set (0.00 sec)

mysql> SET max_sort_length=1536;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @@max_sort_length\G;
*************************** 1. row ***************************
@@max_sort_length: 1536
1 row in set (0.00 sec)
```

在数据库中，最小的存储单元是页（也可以称为块）。为了有效存储列类型为 BLOB 或 TEXT 的大数据类型，一般将列的值存放在行溢出页，而数据页存储的行数据只包含 BLOB 或 TEXT 类型数据列前一部分数据。

图 2-2 显示了 InnoDB 存储引擎对于 BLOB 类型列的存储方式，数据页由许多的行数据组成，每行数据由列组成，对于列类型为 BLOB 的数据，InnoDB 存储引擎只存储前 20 字节，而该列的完整数据则存放在 BLOB 的行溢出页中。在这种方式下，数据页中能存放大量的行数据，从而提高了数据的查询效率。

此外，在有些存储引擎内部，比如 InnoDB 存储引擎，会将大 VARCHAR 类型字符串（如 VARCHAR（65530））自动转化为 TEXT 或 BLOB 类型，这在笔者的另一本书《MySQL 技术内幕：InnoDB 存储引擎》中已经有了非常详细的介绍，这里不再赘述。

2.7.6 ENUM 和 SET 类型

ENUM 和 SET 类型都是集合类型，不同的是 ENUM 类型最多可枚举 65 536 个元素，而 SET 类型最多枚举 64 个元素。

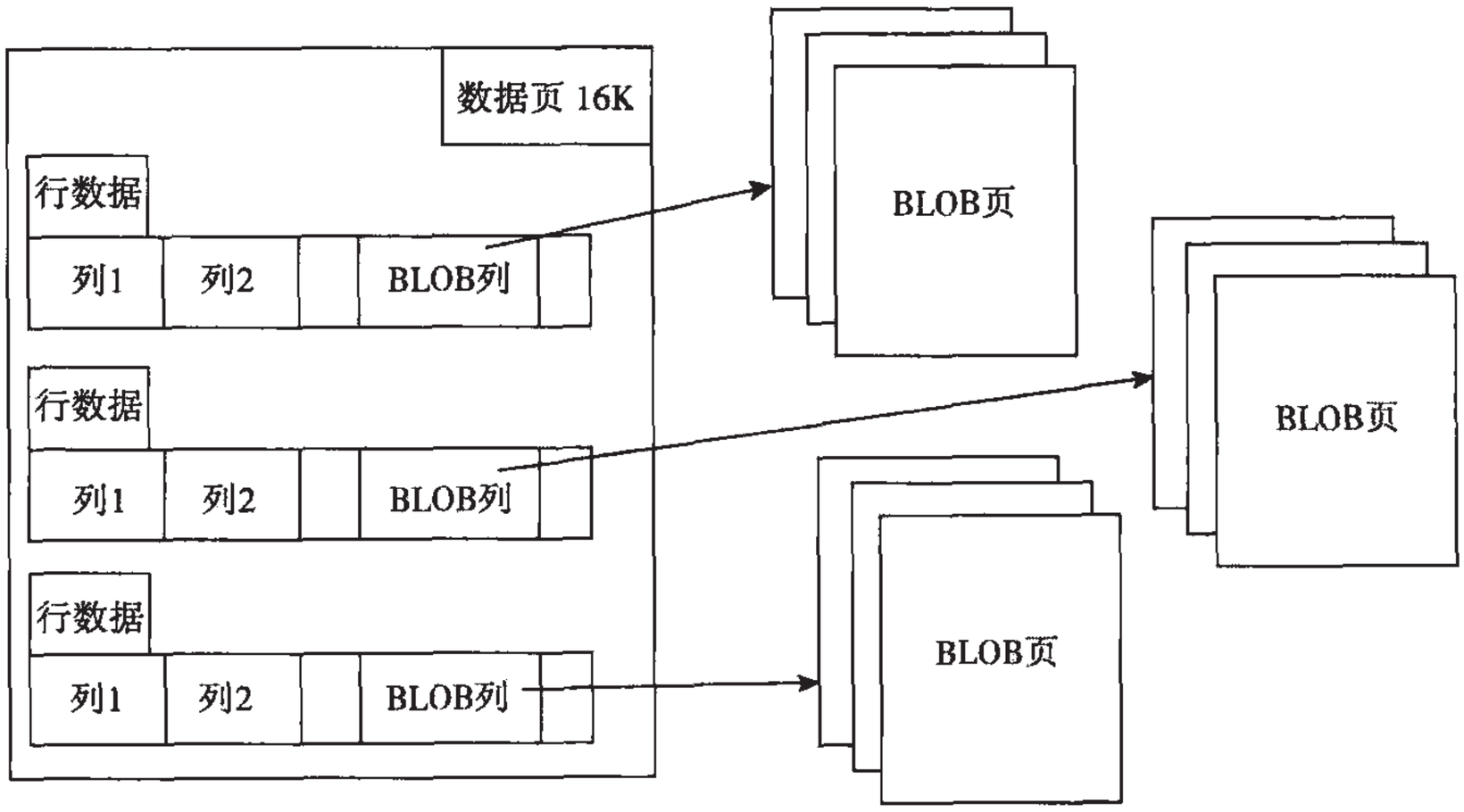


图 2-2 InnoDB 存储引擎中 BLOB 数据存储方式

由于 MySQL 不支持传统的 CHECK 约束，因此通过 ENUM 和 SET 类型并结合 SQL_MODE 可以解决一部分问题。例如，表中有一个“性别”列，规定域的范围只能是 male 和 female，在这种情况下可以通过 ENUM 类型结合严格的 SQL_MODE 模式进行约束，过程如下：

```
mysql> CREATE TABLE t (
    -> user VARCHAR(30),
    -> sex ENUM('male','female')
    -> )ENGINE=InnoDB;
Query OK, 0 rows affected (0.01 sec)

mysql> SET SQL_MODE='strict_trans_tables';
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t SELECT 'David','male';
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> INSERT INTO t SELECT 'Mariah','female';
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> INSERT INTO t SELECT 'John','bimale';
ERROR 1265 (01000): Data truncated for column 'sex' at row 1
```

可以看到，前两次的操作正确地插入了“male”和“female”值，而对于未定义的“bimale”，MySQL 数据库在严格的 SQL_MODE 下抛出了报错警告，起到了一定的约束作用。

2.8 小结

不能轻视数据类型的选择，这是构建关系数据库的基础，并且将影响到之后所有数据库的应用。本章用了一些篇幅来介绍日期和时间类型及其通常存在的一些问题，这是因为该类型比较复杂，而且使用频繁。此外，在使用字符类型时需要特别考虑字符集及排序规则的选择，这同样非常重要，由于需要越来越多地参与多国语言项目的开发，建议将数据库的默认字符集设置为 utf8，这对今后数据库的移植和开发大有益处。

第 3 章

查询处理

查询操作是关系数据库中使用最为频繁的操作，也是构成其他SQL语句（如DELETE、UPDATE）的基础。当要删除或更新某些记录时，首先要查询出这些记录，然后再对其进行相应的SQL操作。因此基于SELECT的查询操作就显得非常重要。对于查询处理，可将其分为逻辑查询处理及物理查询处理。逻辑查询处理表示执行查询应该产生什么样的结果，而物理查询代表MySQL数据库是如何得到该结果的。两种查询的方法可能完全不同，但是得到的结果必定是相同的。

3.1 逻辑查询处理

SQL语言不同于其他编程语言（如C、C++、Java、Python），最明显的不同体现在处理代码的顺序上。在大多数编程语言中，代码按编码顺序被处理。但在SQL语言中，第一个被处理的子句总是FROM子句。图3-1显示了逻辑查询处理的顺序以及步骤的序号。

```
(8) SELECT (9) DISTINCT<select_list>
(1) FROM <left_table>
(3) <join_type>JOIN<right_table>
(2)          ON<join_condition>
(4) WHERE<where_condition>
(5) GROUP BY<group_by_list>
(6) WITH {CUBE|ROLLUP}
(7) HAVING<having_condition>
(10) ORDER BY<order_by_list>
(11) LIMIT <limit_number>
```

图3-1 逻辑查询处理的步骤序号

可以看到一共有11个步骤，最先执行的是FROM操作，最后执行的是LIMIT操作。每个操作都会产生一张虚拟表，该虚拟表作为一个处理的输入。这些虚拟表对用户是透明的，只有最后一步生成的虚拟表才会返回给用户。如果没有在查询中指定某一子句，则将跳过相应的步骤。

接着我们来具体分析查询处理的各个阶段：

1）FROM：对FROM子句中的左表<left_table>和右表<right_table>执行笛卡儿积(Cartesian product)，产生虚拟表VT1。

2）ON：对虚拟表VT1应用ON筛选，只有那些符合<join_condition>的行才被插入虚拟表VT2中。

3）JOIN：如果指定了OUTER JOIN（如LEFT OUTER JOIN、RIGHT OUTER JOIN），那么保留表中未匹配的行作为外部行添加到虚拟表VT2中，产生虚拟表VT3。如果FROM子句包含两个以上表，则对上一个连接生成的结果表VT3和下一个表重复执行步骤1）～步骤3），直到处理完所有的表为止。

4）WHERE：对虚拟表VT3应用WHERE过滤条件，只有符合<where_condition>的记录才被插入虚拟表VT4中。

5）GROUP BY：根据GROUP BY子句中的列，对VT4中的记录进行分组操作，产生VT5。

6）CUBE | ROLLUP：对表VT5进行CUBE或ROLLUP操作，产生表VT6。

7）HAVING：对虚拟表 VT6 应用 HAVING 过滤器，只有符合 <having_condition> 的记录才被插入虚拟表 VT7 中。

8）SELECT：第二次执行 SELECT 操作，选择指定的列，插入到虚拟表 VT8 中。

9）DISTINCT：去除重复数据，产生虚拟表 VT9。

10）ORDER BY：将虚拟表 VT9 中的记录按照 <order_by_list> 进行排序操作，产生虚拟表 VT10。

11）LIMIT：取出指定行的记录，产生虚拟表 VT11，并返回给查询用户。

下面通过一个查询示例来详细描述逻辑处理的 11 个阶段。首先根据下面的代码，创建一个典型的 TPC-C 应用用户数据表 customers 和 orders，并填充一定量的数据。

```
CREATE TABLE customers
(
   customer_id VARCHAR(10) NOT NULL,
   city VARCHAR(10) NOT NULL ,
   PRIMARY KEY(customer_id)
)ENGINE=INNODB;

INSERT INTO customers SELECT '163','HangZhou';
INSERT INTO customers SELECT '9you','ShangHai';
INSERT INTO customers SELECT 'TX', 'HangZhou';
INSERT INTO customers SELECT 'baidu', 'HangZhou';

CREATE TABLE orders
(
   order_id INT NOT NULL AUTO_INCREMENT,
   customer_id VARCHAR(10),
   PRIMARY KEY(order_id)
)ENGINE=INNODB;

INSERT INTO orders SELECT NULL,'163';
INSERT INTO orders SELECT NULL,'9you';
INSERT INTO orders SELECT NULL,'9you';
INSERT INTO orders SELECT NULL,'9you';
INSERT INTO orders SELECT NULL,'TX';
INSERT INTO orders SELECT NULL,NULL;
```

表 customers 和表 orders 的内容如表 3-1 和表 3-2 所示。

表 3-1　customers 表中记录

customer_id	city
163	HangZhou
9you	ShangHai
TX	HangZhou
baidu	HangZhou

表 3-2 orders 表中记录

order_id	customer_id	order_id	customer_id
1	163	5	9you
2	163	6	TX
3	9you	7	NULL
4	9you		

通过如下语句来查询来自杭州且订单数少于 2 的客户，并且查询出他们的订单数量，查询结果按订单数从小到大排序，最后得到的结果如表 3-3 所示。

```
SELECT c.customer_id, count(o.order_id) AS total_orders
FROM customers as c
LEFT JOIN orders AS o
ON c.customer_id = o.customer_id
WHERE c.city = 'HangZhou'
GROUP BY c.customer_id
HAVING count(o.order_id) < 2
ORDER BY total_orders DESC;
```

表 3-3 来自杭州且订单数少于 2 的顾客

customer_id	total_orders
TX	1
baidu	0

3.1.1 执行笛卡儿积

第一步需要做的是对 FROM 子句前后的两张表进行笛卡儿积操作，也称做交叉连接(Cross Join)，生成虚拟表 VT1。如果 FROM 子句前的表中包含 a 行数据，FROM 子句后的表中包含 b 行数据，那么虚拟表 VT1 中将包含 a*b 行数据。虚拟表 VT1 的列由源表定义。对于前面的 SQL 查询语句，会先执行表 orders 和 customers 的笛卡儿积操作。

```
FROM customers as c ..... JOIN orders as o
```

表 3-4 显示了笛卡儿积产生的虚拟表 VT1。

表 3-4 笛卡儿积返回的虚拟表 VT1

customer_id	city	order_id	customer_id
163	HangZhou	1	163
9you	ShangHai	1	163
baidu	HangZhou	1	163
TX	HangZhou	1	163
163	HangZhou	2	163

（续）

customer_id	city	order_id	customer_id
9you	ShangHai	2	163
baidu	HangZhou	2	163
TX	HangZhou	2	163
163	HangZhou	3	9you
9you	ShangHai	3	9you
baidu	HangZhou	3	9you
TX	HangZhou	3	9you
163	HangZhou	4	9you
9you	ShangHai	4	9you
baidu	HangZhou	4	9you
TX	HangZhou	4	9you
163	HangZhou	5	9you
9you	ShangHai	5	9you
baidu	HangZhou	5	9you
TX	HangZhou	5	9you
163	HangZhou	6	TX
9you	ShangHai	6	TX
baidu	HangZhou	6	TX
TX	HangZhou	6	TX
163	HangZhou	7	NULL
9you	ShangHai	7	NULL
baidu	HangZhou	7	NULL
TX	HangZhou	7	NULL

3.1.2 应用 ON 过滤器

SELECT 查询一共有 3 个过滤过程，分别是 ON、WHERE、HAVING。ON 是最先执行的过滤过程。根据上一小节产生的虚拟表 VT1，过滤条件为：

```
ON c.customer_id = o.customer_id
```

对于大多数的编程语言而言，逻辑表达式的值只有两种：TRUE 和 FALSE。但是在关系数据库中起逻辑表达式作用的并非只有两种，还有一种称为三值逻辑的表达式。这是因为在数据库中对 NULL 值的比较与大多数编程语言不同。在 C 语言中，NULL == NULL 的比较返回的是 1，即相等，而在关系数据库中，NULL 的比较则完全不是这么回事，例如：

```
mysql> SELECT 1 = NULL\G;
*************************** 1. row ***************************
1 = NULL: NULL

mysql> SELECT NULL = NULL\G;
*************************** 1. row ***************************
NULL = NULL: NULL
1 row in set (0.00 sec)
```

第一个 NULL 值的比较返回的是 NULL 而不是 0，第二个 NULL 值的比较返回的仍然是 NULL，而不是 1。对于比较返回值为 NULL 的情况，用户应该将其视为 UNKNOWN，即表示未知的。因为在某些情况下，NULL 返回值可能代表 1，即 NULL 等于 NULL，而有时 NULL 返回值可能代表 0。

对于在 ON 过滤条件下的 NULL 值比较，此时的比较结果为 UNKNOWN，却被视为 FALSE 来进行处理，即两个 NULL 并不相同。但是在下面两种情况下认为两个 NULL 值的比较是相等的：

- GROUP BY 子句把所有 NULL 值分到同一组。
- ORDER BY 子句中把所有 NULL 值排列在一起。

下面来看一个例子，创建表 t 的语句如下所示：

```
CREATE TABLE t ( a CHAR(5) )ENGINE=INNODB;
INSERT INTO t SELECT 'a';
INSERT INTO t SELECT NULL;
INSERT INTO t SELECT 'b';
INSERT INTO t SELECT 'c';
INSERT INTO t SELECT NULL;
```

接着对表 t 中的列 a 进行 ORDER BY 操作，得到的结果如下所示：

```
mysql> SELECT * FROM t ORDER BY a\G;
*************************** 1. row ***************************
a: NULL
*************************** 2. row ***************************
a: NULL
*************************** 3. row ***************************
a: a
*************************** 4. row ***************************
a: b
*************************** 5. row ***************************
a: c
5 rows in set (0.00 sec)
```

再对列 a 进行分组统计：

```
mysql> SELECT a,COUNT(1) FROM t GROUP BY a\G;
*************************** 1. row ***************************
       a: NULL
```

```
COUNT(1): 2
*************************** 2. row ***************************
       a: a
COUNT(1): 1
*************************** 3. row ***************************
       a: b
COUNT(1): 1
*************************** 4. row ***************************
       a: c
COUNT(1): 1
4 rows in set (0.00 sec)
```

可见，对于 ORDER BY 的 SQL 查询语句，返回结果将两个 NULL 值先返回并排列在一起。对于 GROUP BY 的查询语句，返回 NULL 值的有两条记录。

因此在产生虚拟表 VT2 时，会增加一个额外的列来表示 ON 过滤条件的返回值，返回值有 TRUE、FALSE、UNKNOWN，如表 3-5 所示。

表 3-5　带有逻辑判断值的 VT1

Match	customer_id	city	order_id	customer_id
TRUE	163	HangZhou	1	163
FALSE	9you	ShangHai	1	163
FALSE	baidu	HangZhou	1	163
FALSE	TX	HangZhou	1	163
TRUE	163	HangZhou	2	163
FALSE	9you	ShangHai	2	163
FALSE	baidu	HangZhou	2	163
FALSE	TX	HangZhou	2	163
FALSE	163	HangZhou	3	9you
TRUE	9you	ShangHai	3	9you
FALSE	baidu	HangZhou	3	9you
FALSE	TX	HangZhou	3	9you
FALSE	163	HangZhou	4	9you
TRUE	9you	ShangHai	4	9you
FALSE	baidu	HangZhou	4	9you
FALSE	TX	HangZhou	4	9you
FALSE	163	HangZhou	5	9you
TRUE	9you	ShangHai	5	9you
FALSE	baidu	HangZhou	5	9you
FALSE	TX	HangZhou	5	9you
FALSE	163	HangZhou	6	TX
FALSE	9you	ShangHai	6	TX

（续）

Match	customer_id	city	order_id	customer_id
FALSE	baidu	HangZhou	6	TX
TRUE	TX	HangZhou	6	TX
UNKNOWN	163	HangZhou	7	NULL
UNKNOWN	9you	ShangHai	7	NULL
UNKNOWN	baidu	HangZhou	7	NULL
UNKNOWN	TX	HangZhou	7	NULL

取出比较值为 TRUE 的记录，产生虚拟表 VT2，结果如表 3-6 所示。

表 3-6 虚拟表 VT2

customer_id	city	order_id	customer_id
163	HangZhou	1	163
163	HangZhou	2	163
9you	ShangHai	3	9you
9you	ShangHai	4	9you
9you	ShangHai	5	9you
TX	HangZhou	6	TX

3.1.3 添加外部行

这一步只有在连接类型为 OUTER JOIN 时才发生，如 LEFT OUTER JOIN、RIGHT OUTER JOIN、FULL OUTER JOIN。虽然在大多数时候我们可以省略 OUTER 关键字，但 OUTER 代表的就是外部行。LEFT OUTER JOIN 把左表记为保留表，RIGHT OUTER JOIN 把右表记为保留表，FULL OUTER JOIN 把左右表都记为保留表。添加外部行的工作就是在 VT2 表的基础上添加保留表中被过滤条件过滤掉的数据，非保留表中的数据被赋予 NULL 值，最后生成虚拟表 VT3，如表 3-7 所示。

在这个例子中，保留表是 customers，设置保留表的过程如下：

```
customers as c LEFT JOIN orders as o
```

顾客 baidu 在 VT2 表中由于没有订单而被过滤，因此 baidu 作为外部行被添加到虚拟表 VT2 中，将非保留表中的数据赋值为 NULL。

如果需要连接表的数量大于 2，则对虚拟表 VT3 重做本节首的步骤 1）～步骤 3），最后产生的虚拟表作为下一个步骤的输出。

表 3-7 虚拟表 VT3

customer_id	city	order_id	customer_id
163	HangZhou	1	163
163	HangZhou	2	163
9you	ShangHai	3	9you
9you	ShangHai	4	9you
9you	ShangHai	5	9you
baidu	**HangZhou**	**NULL**	**NULL**
TX	HangZhou	6	TX

3.1.4 应用 WHERE 过滤器

对上一步骤产生的虚拟表 VT3 进行 WHERE 条件过滤，只有符合 <where_condition> 的记录才会输出到虚拟表 VT4 中。

在当前应用 WHERE 过滤器时，有两种过滤是不被允许的：

- 由于数据还没有分组，因此现在还不能在 WHERE 过滤器中使用 where_condition=MIN(col) 这类对统计的过滤。
- 由于没有进行列的选取操作，因此在 SELECT 中使用列的别名也是不被允许的，如 SELECT city as c FROM t WHERE c='ShangHai' 是不允许出现的。

首先来看一个在 WHERE 过滤条件中使用分组过滤查询导致出错的例子。

```
mysql> SELECT customer_id,COUNT(customer_id)
    -> FROM orders
    -> WHERE COUNT(customer_id)<2 ;
ERROR 1111 (HY000): Invalid use of group function
```

可以看到 MySQL 数据库提示错误地使用了分组函数。接着来看一个列别名使用出错的例子。

```
mysql> SELECT order_id AS o,customer_id AS c
    -> FROM orders
    -> WHERE c='163';
ERROR 1054 (42S22): Unknown column 'c' in 'where clause'
```

因为在当前的步骤中还未进行 SELECT 选取列名的操作，所以此时的列别名是不被支持的，MySQL 数据库抛出了错误，提示未知的列 c。

应用 WHERE 过滤器：WHERE c.city='HangZhou'，最后得到的虚拟表 VT4 如表 3-8 所示。

此外，在 WHERE 过滤器中进行的过滤和在 ON 过滤器中进行的过滤是有所不同的。对于 OUTER JOIN 中的过滤，在 ON 过滤器过滤完之后还会添加保留表中被 ON 条件过滤掉的

记录，而 WHERE 条件中被过滤掉的记录则是永久的过滤。在 INNER JOIN 中两者是没有差别的，因为没有添加外部行的操作。来看下面这个查询：

```
SELECT * FROM customers c
LEFT JOIN orders o
ON c.customer_id=o.customer_id AND c.city='HangZhou';
```

得到的结果如表 3-9 所示。

表 3-8 虚拟表 VT4

c.customer_id	c.city	o.order_id	o.customer_id
163	HangZhou	1	163
163	HangZhou	2	163
baidu	HangZhou	NULL	NULL
TX	HangZhou	6	TX

表 3-9 查询结果

c.customer_id	c.city	o.order_id	o.customer_id
163	HangZhou	1	163
163	HangZhou	2	163
9you	ShangHai	NULL	NULL
baidu	HangZhou	NULL	NULL
TX	HangZhou	6	TX

对比表 3-8 和表 3-9 可以发现，customer_id 为 9you 的记录被添加入后者的查询中。这是因为 ON 过滤条件虽然过滤掉了 city 不等于“HangZhou”的记录，但是由于查询是 OUTER JOIN，因此会对保留表中被排除的记录进行再次的添加操作。

3.1.5 分组

在本步骤中根据指定的列对上个步骤中产生的虚拟表进行分组，最后得到虚拟表 VT5，如表 3-10 所示。在示例中进行如下分组：

```
GROUP BY c.customer_id
```

表 3-10 虚拟表 VT5

c.customer_id	c.city	o.order_id	o.customer_id
163	HangZhou	1	163
163	HangZhou	2	163
baidu	HangZhou	NULL	NULL
TX	HangZhou	6	TX

前面已经介绍了在执行 GROUP BY 阶段，数据库认为两个 NULL 值是相等的，因此会将 NULL 值分到同一个分组中。

3.1.6 应用 ROLLUP 或 CUBE

如果指定了 ROLLUP 选项，那么将创建一个额外的记录添加到虚拟表 VT5 的最后，并生成虚拟表 VT6。因为我们的查询并未用到 ROLLUP，所以将跳过本步骤。

对于 CUBE 选项，MySQL 数据库虽然支持该关键字的解析，但是并未实现该功能。若执行带有 CUBE 选项的 SQL 语句，用户可能会得到如下的错误提示：

```
mysql>SELECT c.customer_id, count(o.order_id) as total_orders
    -> FROM customers as c
    -> LEFT JOIN orders as o
    -> ON c.customer_id = o.customer_id
    -> WHERE c.city = 'HangZhou'
    -> GROUP BY c.customer_id
    -> WITH CUBE
    -> ;
ERROR 1235 (42000): This version of MySQL doesn't yet support 'CUBE';

mysql>select @@version\G;
*************************** 1. row ***************************
@@version: 5.5.14
1 row in set (0.00 sec)
```

可以看到提示当前的 MySQL 数据库版本不支持 CUBE 操作。

3.1.7 应用 HAVING 过滤器

这是最后一个条件过滤器了，之前已经分别应用了 ON 和 WHERE 过滤器。在该步骤中对于上一步产生的虚拟表应用 HAVING 过滤器，HAVING 是对分组条件进行过滤的筛选器。对于示例的查询语句，其分组条件为：

```
HAVING count(o.order_id) < 2
```

因此将 customer_id 为 163 的订单从虚拟表中删除，生成的虚拟表 VT6 如表 3-11 所示。

表 3-11 虚拟表 VT6

c.customer_id	c.city	o.order_id	o.customer_id
baidu	HangZhou	NULL	NULL
TX	HangZhou	6	TX

需要特别注意的是，在这个分组中不能使用 COUNT（1）或 COUNT（*），因为这会把通过 OUTER JOIN 添加的行统计入内而导致最终查询结果与预期结果不同。在这个例子中只

能使用 COUNT o.order_id 才能得到预期的结果。例如，将分组查询语句改成 COUNT(*)，则会得到如表 3-12 所示的结果。

表 3-12 COUNT(*) 的结果

customer_id	total_orders
baidu	1
TX	1

显然顾客 baidu 并没有任何的订单操作，返回预期应该为 0，但是在 COUNT（*）的条件下，该返回值为 1。

注意 子查询不能用做分组的聚合函数，如 HAVING COUNT(SELECT ...)<2 是不合法的。

3.1.8 处理 SELECT 列表

虽然 SELECT 是查询中最先被指定的部分，但是直到步骤 8）时才真正进行处理。在这一步中，将 SELECT 中指定的列从上一步产生的虚拟表中选出。

有一点容易被忽视的是，列的别名不能在 SELECT 中的其他别名表达式中使用。因此对于下列查询 MySQL 数据库会抛出错误提示：

```
mysql>SELECT order_id AS o, o+1 AS n FROM orders;
ERROR 1054 (42S22): Unknown column 'o' in 'field list'
```

对于示例的 SQL 查询，其 SELECT 部分为：

```
SELECT c.customer_id, count(o.customer_id) AS total_orders
```

最后计算得到的虚拟表 VT7 如表 3-13 所示。

表 3-13 虚拟表 VT7

customer_id	total_orders
baidu	0
TX	1

3.1.9 应用 DISTINCT 子句

如果在查询中指定了 DISTINCT 子句，则会创建一张内存临时表（如果内存中存放不下就放到磁盘上）。这张内存临时表的表结构和上一步产生的虚拟表一样，不同的是对进行 DISTINCT 操作的列增加了一个唯一索引，以此来去除重复数据。

由于在这个 SQL 查询中未指定 DISTINCT，因此跳过本步骤。另外，对于使用了

GROUP BY 的查询，再使用 DISTINCT 是多余的，因为已经进行分组，不会移除任何行。

3.1.10 应用 ORDER BY 子句

根据 ORDER BY 子句中指定的列对上一步输出的虚拟表进行排列，返回新的虚拟表。还可以在 ORDER BY 子句中指定 SELECT 列表中列的序列号，如下面的语句：

```
SELECT order_id,customer_id
FROM orders
ORDER BY 2,1;
```

等同于：

```
SELECT order_id,customer_id
FROM orders
ORDER BY customer_id,order_id;
```

通常情况下，并不建议采用这种方式来进行排序，因为程序员可能修改了 SELECT 列表中的列，而忘记修改 ORDER BY 中的列表。当然，如果用户对网络传输要求很高，这也不失为一种节省网络传输字节的方法。

对于这里的示例，ORDER BY 子句为：

```
ORDER BY total_orders DESC;
```

最后得到的虚拟表如表 3-14 所示。

表 3-14 虚拟表 VT8

customer_id	total_orders
TX	1
baidu	0

大多数 DBA 和开发人员都错误地认为在选取表中的数据时，记录会按照表中主键的大小顺序地取出，即结果像进行了 ORDER BY 一样。导致这个经典错误的原因主要是没有理解什么才是真正的关系数据库。下面介绍一下关系数据库的起源。

1970 年，IBM 公司的研究员、有“关系数据库之父”之称的 E. F. Codd 博士在刊物《Communication of the ACM》上发表了题为“A Relational Model of Data for Large Shared Data banks（大型共享数据库的关系模型）”的论文，文中首次提出了“数据库的关系模型”的概念，奠定了关系模型的理论基础。后来 Codd 又陆续发表多篇文章，论述了范式理论和衡量关系系统的 12 条准则，用数学理论奠定了关系数据库的基础。IBM 的 Ray Boyce 和 Don Chamberlin 将 Codd 关系数据库的 12 条准则的数学定义以简单的关键字语法表现出来，里程碑式地提出了 SQL 语言。由于关系模型的相关书籍简单明了、具有坚实的数学理论基础，因此一经推出就受到了学术界和产业界的高度重视和广泛响应，并很快成为数据库市场

的主流。20 世纪 80 年代以来，计算机厂商推出的数据库管理系统几乎都支持关系模型，数据库领域当前的研究工作大都以关系模型为基础。

关系数据库是在数学的基础上发展起来的，关系对应于数学中集合的概念。数据库中常见的查询操作其实对应的是集合的某些运算：选择、投影、连接、并、交、差、除。最终的结果虽然是以一张二维表的方式呈现在用户面前，但是从数据库内部来看是一系列的集合操作。因此，对于表中的记录，用户需要以集合的思想来理解。对于 customers 和 orders 表，更准确的描述应如图 3-2 所示。

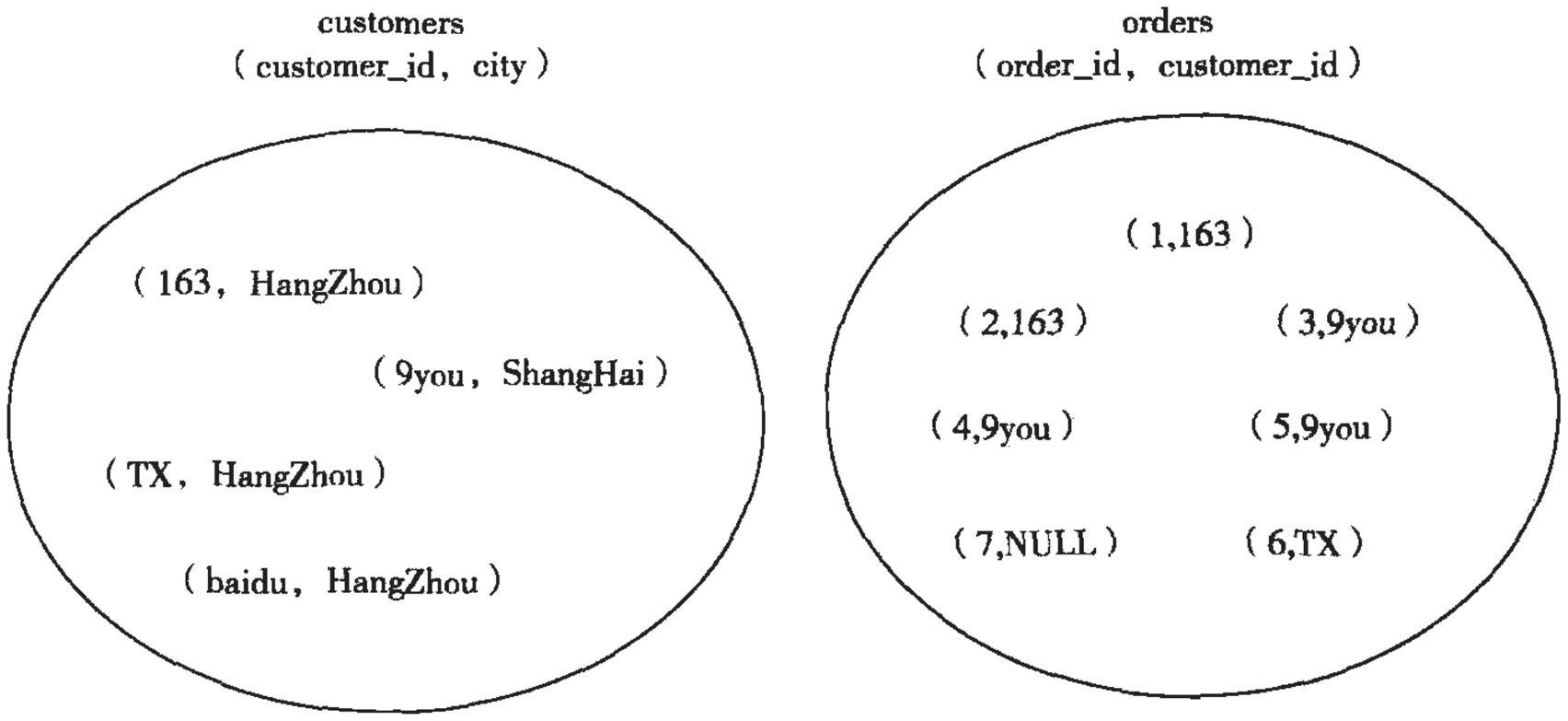


图 3-2 customers 和 orders 集合

将图 3-2 和表 3-1、表 3-2 进行对比。表 3-1、表 3-2 以一张二维表来展示，而实际上更应该将表中的数据理解为图 3-2。因为表中的数据是集合中的元素，而集合是无序的。因此对于没有 ORDER BY 子句的 SQL 语句，其解析结果应为：从集合中选择期望的子集合。这表明结果并不一定要有序。可能有的读者还不信，下面是一张生产环境下的表 userinfo，其存储引擎为 MyISAM，表结构定义如下：

```
mysql> SHOW CREATE TABLE userinfo\G;
*************************** 1. row ***************************
       Table: userinfo
Create Table: CREATE TABLE `userinfo` (
 `UserSN` int(10) unsigned NOT NULL default '0',
`UserNick` varchar(20) binary NOT NULL default '0',
`UserID` varchar(22) binary NOT NULL default '0',
`UserGender` enum('M','F') NOT NULL default 'M',
`UserPower` int(10) NOT NULL default '0',
`Exp` int(10) unsigned NOT NULL default '0',
`Level` mediumint(7) unsigned NOT NULL default '0',
`IsAllowMsg` enum('Y','N') NOT NULL default 'Y',
`IsAllowInvite` enum('Y','N') NOT NULL default 'Y',
`LastLoginTime` timestamp(14) NOT NULL,
```

```
`Password` varchar(23) NOT NULL default '0',
PRIMARY KEY (`UserSN`),
KEY `UserID` (`UserID`),
KEY `UserNick` (`UserNick`),
KEY `LastLoginTime` (`LastLoginTime`),
KEY `Exp` (`Exp`)
) TYPE=MyISAM
```

可以看到表的主键为 UserSN，执行 SELECT * FROM userinfo, 得到如下结果：

```
mysql> SELECT UserSN,'...'
    -> FROM userinfo LIMIT 5\G;
*************************** 1. row ***************************
UserSN: 140911
    ...: ...
*************************** 2. row ***************************
UserSN: 127317
    ...: ...
*************************** 3. row ***************************
UserSN: 142336
    ...: ...
*************************** 4. row ***************************
UserSN: 142390
    ...: ...
*************************** 5. row ***************************
UserSN: 131060
    ...: ...
5 rows in set (0.00 sec)
```

为了排版简单，这里没有列出无用的列信息。可以看到，虽然 UserSN 是主键，但是查询出来的结果却是无序的，并没有按 UserSN 的顺序进行选择。添加 ORDER BY 子句后结果如下：

```
mysql> SELECT UserSN,'...'
    -> FROM userinfo ORDER BY UserSN LIMIT 5\G;
*************************** 1. row ***************************
UserSN: 0
    ...: ...
*************************** 2. row ***************************
UserSN: 1
    ...: ...
*************************** 3. row ***************************
UserSN: 2
    ...: ...
*************************** 4. row ***************************
UserSN: 3
    ...: ...
*************************** 5. row ***************************
UserSN: 4
    ...: ...
5 rows in set (0.00 sec)
```

可以看到，如果想要按顺序选择数据，那么要应用 ORDER BY 子句，因为数据并非总是按照主键顺序进行排序的。有的 DBA 可能会说，上述例子中选择的存储引擎是 MyISAM，而 InnoDB 存储引擎是索引组织表（Index Organized Table），完全不同于 MyISAM 存储引擎，其结果应该是有序的，是按主键顺序排序的。

其实不然，要打破这个“误解”只需要一个很小的例子，如下所示：

```
CREATE TABLE animals
(
   id INT PRIMARY KEY,
   name VARCHAR(30),
   UNIQUE KEY(name)
)ENGINE=INNODB;

INSERT INTO animals SELECT 1,'Tiger';
INSERT INTO animals SELECT 2,'Dog';
INSERT INTO animals SELECT 3,'Cat';
```

当执行 SELECT * FROM animals 语句后，得到的结果如表 3-15 所示。

表 3-15 执行结果

id	name	id	name
3	Cat	1	Tiger
2	Dog		

表 animals 上有主键 id，对于没有 ORDER BY 子句进行查询的操作，“出人意料”地按照主键降序排列结果。由此可见，即使采用的是 InnoDB 存储引擎表，对于没有使用 ORDER BY 子句的选择查询，其结果永远不会是按照主键顺序进行排列的。因为没有 ORDER BY 子句的查询只代表从集合中查询数据，而集合是没有顺序概念的。

因此要牢记，不要为表中的行假定任何特定的顺序。就是说，在实际使用环境中，如果确实需要有序输出行记录，则必须使用 ORDER BY 子句。当然，排序需要成本，可以通过变量来查看数据库的排序操作，示例如下：

```
mysql> SELECT * FROM orders ORDER BY customer_id\G;
*************************** 1. row ***************************
   order_id: 7
customer_id: NULL
*************************** 2. row ***************************
   order_id: 1
customer_id: 163
......
*************************** 7. row ***************************
   order_id: 6
customer_id: TX
7 rows in set (0.00 sec)
```

```
mysql> SHOW STATUS LIKE '%sort%'\G;
*************************** 1. row ***************************
Variable_name: Sort_merge_passes
        Value: 0
*************************** 2. row ***************************
Variable_name: Sort_range
        Value: 0
*************************** 3. row ***************************
Variable_name: Sort_rows
        Value: 7
*************************** 4. row ***************************
Variable_name: Sort_scan
        Value: 1
4 rows in set (0.00 sec)
```

在上面这个例子中可以看到，SQL 语句对 customer_id 进行了 ORDER BY 排序操作，而在 customer_id 列上没有索引，因此需要进行排序操作。在当前会话的状态中，Sort_scan 为 1，Sort_rows 为 7，表示进行了一次排序扫描操作，共排序了 7 条记录。在实际的生产环境中，需要观察这些变量，判断是否可以通过添加索引来避免额外的排序开销。

前面已经提及，在 ORDER BY 子句中，NULL 值被认为是相同的值，会将其排序在一起。在 MySQL 数据库中，NULL 值在升序过程中总是首先被选出，即 NULL 值在 ORDER BY 子句中被视为最小值。

3.1.11 LIMIT 子句

在该步骤中应用 LIMIT 子句，从上一步骤的虚拟表中选出从指定位置开始的指定行数据。对于没有应用 ORDER BY 的 LIMIT 子句，结果同样可能是无序的，因此 LIMIT 子句通常和 ORDER BY 子句一起使用。

MySQL 数据库的 LIMIT 支持如下形式的选择：

```
LIMIT n,m
```

表示从第 n 条记录开始选择 m 条记录。而大多数开发人员喜欢使用这类语句来解决 Web 中经典的分页问题。对于小规模的数据，这并不会有太大的问题。对于论坛这类可能具有非常大规模数据的应用来说，LIMIT n,m 的效率是十分低的。因为每次都需要对数据进行选取。如果只是选取前 5 条记录，则非常轻松和容易；但是对 100 万条记录，选取从第 80 万行记录开始的 5 条记录，则还需要扫描记录到这个位置。因此，对于数据量非常庞大的分页问题，在应用层建立一定的缓存机制是十分必要的。

由于示例中 SQL 语句没有 LIMIT 子句，因此最后得到的结果应如表 3-16 所示。

表 3-16 最后返回的结果集

customer_id	total_orders	customer_id	total_orders
TX	1	baidu	0

3.2 物理查询处理

上一节介绍了逻辑查询处理，并且描述了执行查询应该得到什么样的结果。但是数据库也许并不会完全按照逻辑查询处理的方式来进行查询。图 1-1 显示了在 MySQL 数据库层有 Parser 和 Optimizer 两个组件。Parser 的工作就是分析 SQL 语句，而 Optimizer 的工作就是对这个 SQL 语句进行优化，选择一条最优的路径来选取数据，但是必须保证物理查询处理的最终结果和逻辑查询处理是相等的。

如果表上建有索引，那么优化器就会判断 SQL 语句是否可以利用该索引来进行优化。如果没有可以利用的索引，可能整个 SQL 语句的执行代价非常大。来看如下的一个例子，先生成表和数据，这里使用了第 2 章介绍的数字辅助表。

```
CREATE TABLE x ( a int ) ENGINE = InnoDB;
CREATE TABLE y ( a int ) ENGINE = InnoDB;
CALL pFastCreateNums(500000);
INSERT INTO x SELECT * FROM Nums LIMIT 100000;
INSERT INTO y SELECT * FROM Nums LIMIT 180000;
```

通过数字辅助表生成了 10 万行数据表 x 和 18 万行数据表 y。表 x 和表 y 上都没有索引，因此最终 SQL 解析器解析的执行结果为逻辑处理的步骤，也就是按照上一节中分析的，总共经过 11 个步骤来进行数据的查询。最先根据笛卡儿积生成一张虚拟表 VT1，表 x 有 10 万行数据，表 y 有 18 万行数据，这意味着进行笛卡儿积后产生的虚拟表 VT1 总共有 180 亿行的数据！因此运行这条 SQL 语句，在笔者的双核笔记本上，InnoDB 缓冲池配置为 128MB，总共执行 50 多分钟，具体情况如下：

```
mysql> SELECT COUNT(1)
    -> FROM x
    -> INNER JOIN y
    -> ON x.a=y.a;
+-----------+
| count(1)  |
+-----------+
|    100000 |
+-----------+
1 row in set (50 min 43.97 sec)
```

有人可能会认为，128MB 的 InnoDB 缓冲池太小，从而导致内存中无法存放这么多数据而使执行需要花费这么长的时间。其实不然，表 x 和表 y 的大小都没有超过 20MB，足够存放在 128MB 的内存缓冲池中，语句执行速度慢的主要原因是需要产生 180 亿次的数据。即

便是在内存中产生这么多次的数据，也需要花费很长的时间。然而，如果这时对表 y 添加一个主键值，再执行这条 SQL 语句，你会惊讶地发现只需要 0.85 秒，具体情况如下：

```
mysql> SELECT COUNT(1)
    -> FROM x
    -> INNER JOIN y
    -> ON x.a=y.a;
+----------+
| count(1) |
+----------+
|   100000 |
+----------+
1 row in set (0.85 sec)
```

性能提高了 3000 多倍！促使这个查询时间大幅减少的原因很简单，就是在添加索引后避免了笛卡儿表的产生，因此大幅缩短了语句运行的时间。我们可以通过 EXPLAIN 命令来查看经 SQL 优化器优化后 MySQL 数据库实际选择的执行方式，如下所示：

```
mysql> EXPLAIN SELECT COUNT(1) FROM
    -> x INNSER JOIN y
    ->ON x.a=y.a\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: x
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 100784
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: y
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: test.x.a
         rows: 1
        Extra: Using index
2 rows in set (0.01 sec)
```

添加索引是非常有技巧的一项工作，正确地利用索引的特性能显著提高 SQL 语句运行的效率。但是一味地添加很多索引反而会导致数据库运行得更慢。在后面的章节会详细介绍索引的数据结构，通过内部的实现来更好地理解如何使用索引。目前，读者要明白的是物理

查询会根据索引来进行优化，这也是 MySQL 数据库优化器的一项重要工作。

3.3 小结

理解逻辑查询各个处理阶段对于理解 SQL 编程所需要的特殊观念是非常重要且有必要的。本章详细介绍了逻辑处理的 11 个步骤，3 个过滤处理器 ON、WHERE 和 HAVING，以及这 3 个过滤器的使用过程和不同之处。其中，稍微涉及了一些关于 NULL 值的讨论，这在后面的章节也会遇到。

在本章最后，简单介绍了物理查询处理。逻辑查询只是描述了应该产生什么样的结果。至于 MySQL 数据库通过 SQL 解析器完成对于 SQL 语句的解析，并通过 SQL 优化器选择最优的执行路径，这是物理查询处理过程。物理查询可以利用表上的索引来缩短 SQL 语句运行的时间，以此来提高数据库的整体性能。

第 4 章
子 查 询

MySQL 数据库被诟病的一个地方就是子查询。很多开发人员和 DBA 认为 MySQL 数据库只是拥有该项功能，但性能很差，是一项很不实用的功能。在实际开发中开发人员很少使用子查询，因为在应用程序中使用子查询后，SQL 语句的查询性能变得非常糟糕。本章介绍子查询的使用方法，同时介绍为什么有些子查询的效率令人难以接受，并对子查询的优化给出一些方法和建议。最后，介绍生产环境中子查询的一些应用，如分区、行号计算、缺失范围、连续范围等。

4.1 子查询概述

4.1.1 子查询的优点和限制

子查询是指在一个 SELECT 语句中嵌套另一个 SELECT 语句。

MySQL 数据库从 4.1 版本开始支持子查询，并且支持所有 SQL 标准的子查询，也扩展了一些其独有的子查询标准。下面是一个子查询：

```
SELECT * FROM t1 WHERE column1 = (SELECT column1 FROM t2);
```

在这个示例中，SELECT * FROM t1 是外部查询（outer query），SELECT column1 FROM t2 是子查询。一般来说，称子查询嵌套 (nested) 于外部查询中。实际上也可以将两个或两个以上的子查询进行嵌套。需要注意的是，子查询必须包含括号。

通常来讲，使用子查询的好处如下：

- 子查询允许结构化的查询，这样就可以把一个查询语句的每个部分隔开。
- 子查询提供了另一种方法来执行有些需要复杂的 JOIN 和 UNION 来实现的操作。
- 在许多人看来，子查询可读性较高。而实际上，这也是子查询的由来。

一个子查询会返回一个标量（单一值）、一个行、一个列或一个表（一行或多行及一列或多列），这些子查询被称为标量、列、行和表子查询。可返回一个特定种类结果的子查询经常只用于特定的语境中，在后面各节中有说明。子查询可以包括普通 SELECT 可以包括的任何关键词或子句，如 DISTINCT、GROUP BY、ORDER BY、LIMIT、JOIN、UNION 等。

子查询的限制是其外部语句必须是以下语句之一：SELECT、INSERT、UPDATE、DELETE、SET 或 DO。还有一个限制是，目前用户不能既在一个子查询中修改一个表，又在同一个表中进行选择，虽然这样的操作可用于普通的 DELETE、INSERT、REPLACE 和 UPDATE 语句中，但是对子查询不可以同时进行这样的操作。

4.1.2 使用子查询进行比较

最常见的一种子查询使用方式如下：

```
non_subquery_operand comparison_operator (subquery)
```

comparison_operator 可以是以下操作符之一：=、>、<、>=、<=、<>。

例如：

```
... 'a' = (SELECT column1 FROM t1)
```

以下是一个常见的子查询比较的例子，其中不能使用 JOIN 来完成此类比较。表 t1 中有些值与表 t2 中的最大值相同，该子查询可以查找出所有这些行数。

```
SELECT column1 FROM t1
WHERE column1 = (SELECT MAX(column2) FROM t2);
```

下面是另一个例子，该例子也不可能使用 JOIN 来得到结果，因为该例子涉及对其中一个表进行总计。表 t1 中的有些行含有的值会在给定的列中出现两次，该例子可以查找出所有这些行。

```
SELECT * FROM t1 AS t
WHERE 2 = (SELECT COUNT(*) FROM t1 WHERE t1.id = t.id);
```

4.1.3 使用 ANY、IN 和 SOME 进行子查询

使用 ANY、IN 和 SOME 进行子查询的语法如下：

```
operand comparison_operator ANY (subquery)
operand IN (subquery)
operand comparison_operator SOME (subquery)
```

ANY 关键词必须与一个比较操作符一起使用。ANY 关键词的意思是“对于子查询返回的列中的任一数值，如果比较结果为 TRUE，则返回 TRUE”。例如：

```
SELECT s1 FROM t1 WHERE s1 > ANY (SELECT s1 FROM t2);
```

假设表 t1 中有一行包含（10）。如果表 t2 包含（21，14，7），则表达式为 TRUE，因为 t2 中有一个值为 7，该值小于 10。如果表 t2 包含（20，10），或者表 t2 为空表，则表达式为 FALSE。如果表 t2 包含（NULL, NULL, NULL），则表达式为 UNKNOWN。

词语 IN 是“= ANY”的别名。因此，这两个语句是一样的：

```
SELECT s1 FROM t1 WHERE s1 = ANY (SELECT s1 FROM t2);
SELECT s1 FROM t1 WHERE s1 IN (SELECT s1 FROM t2);
```

词语 SOME 是 ANY 的别名。因此，这两个语句是一样的：

```
SELECT s1 FROM t1 WHERE s1 <> ANY  (SELECT s1 FROM t2);
SELECT s1 FROM t1 WHERE s1 <> SOME (SELECT s1 FROM t2);
```

使用词语 SOME 的机会很少，但是这个例子显示了为什么 SOME 是有意义的。对于大

多数人来说，英语短语“a is not equal to any b”的意思是“没有一个b与a相等”，但是在SQL语法中不是这个意思，其意思是“有部分b与a不相等”。使用“<>SOME”有助于确认用户是否理解该查询的真正含义。

4.1.4 使用ALL进行子查询

使用ALL进行子查询的语法：

```
operand comparison_operator ALL (subquery)
```

词语ALL必须与比较操作符一起使用。ALL的意思是“对于子查询返回的列中的所有值，如果比较结果为TRUE，则返回TRUE”。例如：

```
SELECT s1 FROM t1 WHERE s1 > ALL (SELECT s1 FROM t2);
```

假设表t1中有一行包含（10）。如果表t2包含（−5，0，＋5），则表达式为TRUE，因为10比t2中的所有三个值都大。如果表t2包含（12，6，NULL，−100），则表达式为FALSE，因为表t2中有一个值12大于10。如果表t2包含（0，NULL，1），则表达式为UNKNOWN。

最后，如果表t2为空表，则结果为TRUE。因此，当表t2为空表时，以下语句为TRUE：

```
SELECT * FROM t1 WHERE 1 > ALL (SELECT s1 FROM t2);
```

但是，当表t2为空表时，以下语句为NULL：

```
SELECT * FROM t1 WHERE 1 > (SELECT s1 FROM t2);
```

另外，当表t2为空表时，以下语句为NULL：

```
SELECT * FROM t1 WHERE 1 > ALL (SELECT MAX(s1) FROM t2);
```

通常，包含NULL值的表和空表为“边缘情况”。当编写子查询代码时，都要考虑是否需要把这两种可能性计算在内。

NOT IN是“<>ALL”的别名。因此，以下两个语句是相同的：

```
SELECT s1 FROM t1 WHERE s1 <> ALL (SELECT s1 FROM t2);
SELECT s1 FROM t1 WHERE s1 NOT IN (SELECT s1 FROM t2);
```

4.2 独立子查询

子查询可以按两种方式进行分类。若按照期望值的数量，可以将子查询分为标量子查询和多值子查询；若按查询对外部查询的依赖可分为独立子查询（self-contained subquery）和

相关子查询（correlated subquery）。标量子查询和多值子查询可以是独立子查询，也可以是相关子查询。这一节先来介绍独立子查询。

独立子查询是不依赖外部查询而运行的子查询。与相关子查询相比，独立子查询更便于SQL 语句的调试。

标量子查询可以出现在查询中希望产生标量值的任何地方，而多值子查询可以出现在查询中希望产生多值集合的任何地方。只要标量子查询返回的是单个值或 NULL 值，就说明该子查询是有效的。如果标量子查询返回多个值，则 MySQL 数据库将抛出错误。如以下两句标量子查询都是正确的。

```
SELECT 'a' = (SELECT 'a') AS t;
SELECT 'a' = (SELECT NULL) AS t;
```

而下面的子查询会抛出异常：

```
mysql> SELECT 'a' = (SELECT 'a' UNION ALL SELECT 'b') AS t;
ERROR 1242 (21000): Subquery returns more than 1 row
```

因为子查询产生的派生表 t 返回两个值（a 和 b），而外部查询期望的是一个标量值，因此在运行时，MySQL 数据库抛出异常，提示该子查询返回多行数据。

目前已经介绍了独立子查询的基本使用方法，接着将继续讨论更加复杂的独立子查询问题。这里将从一个属于关系分区的问题开始。关系分区问题有很多细微的差别和实际的应用，从逻辑上看，关系分区就是一个集合划分出另外一个集合，并产生一个结果集。

例如，从 tpcc 数据库中返回每个美国员工至少处理过一个订单的所有客户。这里需要从所有美国员工的集合中来划分订单的集合，并得到期望匹配的客户集合。这个操作并不简单，因为需要对每一个客户检查多行以判断是否有匹配。

如果事先知道所有美国员工的 employeeid 列表，那么可以通过类似下列的语句来解决问题：

```
SELECT customerid
FROM orders
WHERE employeeid IN ( 1, 2, 3, 4, 8)
GROUP BY customerid
HAVING COUNT(DISTINCT employeeid) = 5
```

然而，一般来说，事先无法得知所有美国员工的 employeeid 列表，但是可能有很多很多的美国员工，这时 SQL 语句变得十分冗长。因此解决这个问题可以通过子查询来代替固定值：

```
SELECT customerid
FROM orders
WHERE employeeid IN
(SELECT employeeid FROM employees WHERE country = 'USA' )
GROUP BY customerid
HAVING COUNT(DISTINCT employeeid) =
    ( SELECT COUNT(*) FROM employees WHERE country = 'USA' )
```

另一个与独立子查询相关的问题是返回在每月最后实际订单日期发生的订单。在这里，需要注意的是，每月最后实际订单的日期可能并不是每月的最后一天。例如，周末可能没有订单产生。因此，每月最后订单日期需要通过子查询来判断。下面是该问题的SQL查询语句，生成的结果如表4-1所示。

```
SELECT orderid,customerid,employeeid,orderdate
FROM orders
WHERE orderdate IN
    ( SELECT MAX(orderdate)
            FROM orders
            GROUP BY (DATE_FORMAT(orderdate,'%Y%m'))
    )
```

表4-1 每月最后实际订单日期的订单

orderid	customerid	employeeid	orderdate
10269	WHITC	5	1996-07-31
10294	RATTC	4	1996-08-30
10317	LONEP	6	1996-09-30
10343	LEHMS	4	1996-10-31
10368	ERNSH	2	1996-11-29
10399	VAFFE	8	1996-12-31
10432	SPLIR	3	1997-01-31
10460	FOLKO	8	1997-02-28
10461	LILAS	1	1997-02-28
10490	HILAA	7	1997-03-31
10491	FURIB	8	1997-03-31
…			
11074	SIMOB	7	1998-05-06

独立子查询返回了每月的最后实际订单日期的列表，如表4-2所示。

子查询通过把订单按月分组并返回每个分组中最大的日期结果，这就是每个月最后订单生成的日期。而后外部查询通过子查询获得的日期来取得所有的订单信息。

这个问题看似得到了解决，实际上并没有这么简单。因为在这个数据量并不大的数据库中（实际orders表的大小只有336KB）执行上述SQL语句竟然需要6.08秒，这个时间是不能接受的。如果我们用EXPLAIN来对语句进行分析，就可以找到问题的所在。EXPLAIN查询的结果如图4-1所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	PRIMARY	orders	ALL	NULL	NULL	NULL	NULL	867	Using where
	2	DEPENDENT SUBQUERY	orders	index	NULL	OrderDate	9	NULL	867	Using index; Using temporary; Using filesort

图4-1 EXPLAIN查询的结果

表 4-2 每月的最后订单日期

orderdate	MAX（orderdate）	orderdate	MAX（orderdate）
1996-07	1996-07-31	1997-07	1997-07-31
1996-08	1996-08-30	1997-08	1997-08-29
1996-09	1996-09-30	1997-09	1997-09-30
1996-10	1996-10-31	1997-10	1997-10-31
1996-11	1996-11-29	1997-11	1997-11-28
1996-12	1996-12-31	1997-12	1997-12-31
1997-01	1997-01-31	1998-01	1998-01-30
1997-02	1997-02-28	1998-02	1998-02-27
1997-03	1997-03-31	1998-03	1998-03-31
1997-04	1997-04-30	1998-04	1998-04-30
1997-05	1997-05-30	1998-05	1998-05-06
1997-06	1997-06-30		

怎么是 DEPENDENT SUBQUERY？难道上述 SQL 的子查询变成了相关子查询？但是从逻辑上来说，优化器只需要先从子查询中得到每个月订单的最大日期就行了。这完全是一个独立子查询，不需要和外部查询有任何交互，为什么变成相关子查询呢？

这个是 MySQL 优化器对 IN 子查询优化时存在的一个问题，MySQL 优化器对于 IN 语句的优化是“LAZY”的。对于 IN 子句，如果不是显式的列表定义，如 IN ('a','b','c')，那么 IN 子句都会被转换为 EXISTS 的相关子查询。如下面这条独立子查询：

```
SELECT ... FROM t1 WHERE t1.a IN (SELECT b FROM t2);
```

优化器会将该语句重写为如下的相关子查询：

```
SELECT ... FROM t1 WHERE EXISTS (SELECT 1 FROM t2 WHERE t2.b = t1.a);
```

如果子查询和外部查询分别返回 M 和 N 行，那么该子查询被扫描为 $O(N+M*N)$ 而不是 $O(M+N)$。

因此，对于上述问题的 SQL 语句，MySQL 数据库的优化器将其对应地转换为如下的相关子查询：

```
SELECT orderid,customerid,employeeid,orderdate
FROM orders AS A
WHERE EXISTS
   ( SELECT *
          FROM orders
          GROUP BY(DATE_FORMAT(orderdate,'%Y%M'))
          HAVING MAX(orderdate)= A.OrderDate
   );
```

用户通过 EXPLAIN EXTENDED 命令可以更为明确地得到优化器的执行方式，如：

```
EXPLAIN EXTENDED
SELECT orderid,customerid,employeeid,orderdate
FROM orders
WHERE orderdate IN
    ( SELECT MAX(orderdate)
            FROM orders
            GROUP BY (DATE_FORMAT(orderdate,'%Y%M'))
    );
```

同样会得到图 4-1 的结果，但是命令行会显示有一个警告（WARNING），如果接着运行 SHOW WARNINGS 语句，将得到如下的结果：

```
/* select#1 */ select `tpcc`.`orders`.`OrderID` AS
`orderid`,`tpcc`.`orders`.`CustomerID` AS
`customerid`,`tpcc`.`orders`.`EmployeeID` AS
`employeeid`,`tpcc`.`orders`.`OrderDate` AS `orderdate` from
`tpcc`.`orders` where
<in_optimizer>(`tpcc`.`orders`.`OrderDate`,<exists>(/*select#2 */
select max(`tpcc`.`orders`.`OrderDate`) from `tpcc`.`orders` group by date_format
(`tpcc`.`orders`.`OrderDate`,'%Y%M') having
(<cache>(`tpcc`.`orders`.`OrderDate`) =
<ref_null_helper>(max(`tpcc`.`orders`.`OrderDate`)))))
```

通过斜体部分的语句可以看到优化器将 IN 语句转化为 EXISTS 语句。同时，这解释了为什么前面的 SQL 执行了那么长的时间。另外，有意思的是，翻阅官方的 MySQL 手册会发现，在子查询章节中有相关子查询的介绍，却没有独立子查询的介绍。这是因为在大多数情况下，MySQL 数据库都将独立子查询转换为相关子查询。

InnoSQL 数据库支持在慢查询日志中记录 InnoDB 的逻辑 IO 和物理 IO 的次数（物理 IO 就是实际读取磁盘的次数），故开启 IO 记录，可在慢查询日志中观察到类似如下的内容：

```
# Time: 111227 23:49:16
# User@Host: root[root] @ localhost [127.0.0.1]
# Query_time: 6.081214  Lock_time: 0.046800 Rows_sent: 42  Rows_examined: 727558
    Logical_reads: 91584 Physical_reads: 19
use tpcc;
SET timestamp=1325000956;
SELECT orderid,customerid,employeeid,orderdate
FROM orders
WHERE orderdate IN
    ( SELECT MAX(orderdate)
            FROM orders
            GROUP BY (DATE_FORMAT(orderdate,'%Y%M'))
    );
```

通过 InnoSQL 的慢查询日志可以看到逻辑 IO 有 91 584 次，物理 IO 只有 19 次。

不过对于上述语句，还是有一定的方法可以优化的。注意到慢的原因是独立子查询被转换成相关子查询，而这个相关子查询需要进行多次的分组操作。可以采取另一个方法，再嵌

套一层子查询，避免多次的分组操作，语句如下：

```
SELECT orderid,customerid,employeeid,orderdate
FROM orders A
WHERE EXISTS
   ( SELECT * FROM (SELECT MAX(orderdate) AS orderdate
          FROM orders
          GROUP BY (DATE_FORMAT(orderdate,'%Y%M')) ) B
          WHERE A.orderdate = B.orderdate
   );
```

这次该 SQL 语句的执行时间缩短到了 0.0976 秒。虽然从查询分析器（如图 4-2 所示）看，优化器还是将其转换为相关子查询，但是减少了外部查询与子查询的匹配次数。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	A	ALL	NULL	NULL	NULL	NULL	867	Using where
2	DEPENDENT SUBQUERY	<derived3>	ALL	NULL	NULL	NULL	NULL	23	Using where
3	DERIVED	orders	index	NULL	OrderDate	9	NULL	867	Using index; Using temporary; Using filesort

图 4-2　执行计划

通过 InnoSQL 数据库的慢查询日志，可以观察到：

```
# Time: 111227 23:45:49
# User@Host: root[root] @ localhost [127.0.0.1]
# Query_time: 0.097601   Lock_time: 0.052001 Rows_sent: 42   Rows_examined: 20444
    Logical_reads: 394 Physical_reads: 19
# 省略 SQL 语句的显示
```

虽然物理 IO 同样还是 19 次，但是逻辑 IO 从原来的 91 584 次减少为 394 次。逻辑 IO 次数差不多为原来的 1/230！

MariaDB 5.3 提供了对独立子查询的优化，尤其是对 IN 语句的优化。可以通过下列语句在 MariaDB 中显式地打开对 SEMIJOIN 的优化（MariaDB 5.3.3 默认已经打开此优化）：

```
SET optimizer_switch='semijoin=on, materialization=on';
```

通过 EXPLAIN 命令看到新的执行计划如图 4-3 所示。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	<subquery2>	ALL	distinct_key	NULL	NULL	NULL	795	100.00	Using where
1	PRIMARY	orders	ref	OrderDate	OrderDate	9	<subquery2>.MAX(orderdate)	1	100.00	
2	SUBQUERY	orders	index	NULL	OrderDate	9	NULL	795	100.00	Using index; Using temporary

图 4-3　在 MariaDB 中开启 SEMIJOIN 优化后的执行计划

从图 4-3 中可以看到，这次显示的是 SUBQUERY 而非之前的 DEPENDENT SUBQUERY，即这次子查询完全是独立的子查询。若再执行 EXPLAIN EXTENDED 命令，可以看到完全不同的执行方法：

```
select `tpcc`.`orders`.`OrderID` AS
`orderid`,`tpcc`.`orders`.`CustomerID` AS
`customerid`,`tpcc`.`orders`.`EmployeeID` AS
`employeeid`,`tpcc`.`orders`.`OrderDate` AS `orderdate` from
<materialize> (select max(`tpcc`.`orders`.`OrderDate`) from
`tpcc`.`orders` group by
date_format(`tpcc`.`orders`.`OrderDate`,'%Y%M')) join
`tpcc`.`orders` where (`tpcc`.`orders`.`OrderDate` =
`<subquery2>`.`MAX(orderdate)`)
```

可以看到，开启对于 SEMIJOIN 的优化后，优化器不再将 IN 子句转换为 EXISTS 语句，而是先将独立子查询产生的结果生成一张物化视图，之后再对外部查询的表进行 JOIN 操作。

4.3 相关子查询

相关子查询（Dependent Subquery 或 Correlated Subquery）是指引用了外部查询列的子查询，即子查询会对外部查询的每行进行一次计算。但是在优化器内部，这是一个动态的过程，随情况的变化会有所不同，通过不止一种优化方式来处理相关子查询。

下面通过一个示例来慢慢展现相关子查询的用法。例如，要查询每个员工最大订单日期的订单，因为一个员工可能有多个订单具有相同的订单日期，所以可能会为每个员工返回多行数据。

通过上一小节的介绍，用户可能会尝试这样的解决方案：使用一个子查询，该子查询与前面那个返回某月份内最后订单日期的订单的子查询类似。

```
SELECT orderid,customerid,employeeid,orderdate,requireddate
FROM orders
WHERE orderdate IN
   ( SELECT MAX(orderdate) FROM orders
         GROUP BY employeeid );
```

实际上，这个解决方案是错误的。虽然结果集会包含正确的订单，但是也可能包含不是该用户的最大订单日期的订单。因此在这个例子中，子查询必须关联外部查询，将内部查询的 employeeid 与外部的 employeeid 进行匹配：

```
SELECT orderid,customerid,employeeid,orderdate,requireddate
FROM orders AS A
WHERE orderdate =
   ( SELECT MAX(orderdate) FROM orders AS B
         WHERE A.employeeid=B.employeeid);
```

该查询得到的正确结果如表 4-3 所示。

表 4-3 每个员工具有最大订单日期的订单

orderid	customerid	employeeid	orderdate	requireddate
11043	SPECD	5	1998-04-22	1998-05-20
11045	BOTTM	6	1998-04-23	1998-05-21
11058	BLAUS	9	1998-04-29	1998-05-27
11063	HUNGO	3	1998-04-30	1998-05-28
11070	LEHMS	2	1998-05-05	1998-06-02
11073	PERIC	2	1998-05-05	1998-06-02
11074	SIMOB	7	1998-05-06	1998-06-03
11075	RICSU	8	1998-05-06	1998-06-03
11076	BONAP	4	1998-05-06	1998-06-03
11077	RATTC	1	1998-05-06	1998-06-03

该查询在笔者的笔记本电脑上运行，需要 1.18 秒，对于这样数据量的表来说，显然太慢了。通过 InnoSQL 数据库的慢查询日志来观察实际的 IO 次数，情况如下：

```
# Time: 111228 20:53:44
# User@Host: root[root] @ localhost [127.0.0.1]
# Query_time: 1.182068   Lock_time: 0.000000 Rows_sent: 10   Rows_examined: 89742
    Logical_reads: 180604 Physical_reads: 23
SET timestamp=1325076824;
SELECT orderid,customerid,employeeid,orderdate,requireddate
FROM orders AS A
WHERE orderdate =
   ( SELECT MAX(orderdate) FROM orders AS B
          WHERE A.employeeid=B.employeeid);
```

可以看到，虽然物理 IO 只有 23 次，但是逻辑 IO 大于 18 万次。对于有经验的 DBA，应该知道可以通过添加一个唯一索引来加快处理速度，具体语句如下：

```
CREATE UNIQUE INDEX idx_empid_od_rd_oid
ON orders(employeeid,orderdate,requireddate,orderid)
```

不错，显示只需要 0.468 秒，差不多节省了一半的时间。能不能更快一点呢？先来分析该 SQL 语句的执行计划，如图 4-4 所示。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	A	ALL	NULL	NULL	NULL	NULL	867	Using where
2	DEPENDENT SUBQUERY	B	ref	idx_empid_od_rd_oid,EmployeeID,EmployeesOrders	idx_empid_od_rd_oid	5	tpcc.A.EmployeeID	4	Using where; Using index

图 4-4 相关子查询的执行计划

从图 4-4 中的执行计划可以看到，优化器的确使用了新建的唯一索引，因此速度得到了一定的提升。然而问题仍然是相关子查询需要与外部子查询的列进行多次比较。通过 InnoSQL 数据库的慢查询日志来观察实际的 IO 次数，具体情况如下：

```
# User@Host: root[root] @ localhost [127.0.0.1]
# Query_time: 0.468033   Lock_time: 0.066003 Rows_sent: 10   Rows_examined: 89742
    Logical_reads: 12826 Physical_reads: 23
# 省略SQL语句的显示
```

可以看到，添加唯一索引后逻辑读取从18万次减少为12 000次，执行效率得到了大幅度的提升。然而，对于相关子查询，有时可以通过派生表来进行重写，以避免子查询与外部子查询的多次比较操作。故对于上述SQL语句，我们可以重写为：

```
SELECT
   A.orderid,A.customerid,A.employeeid,
   B.orderdate,requireddate
FROM orders AS A,
(SELECT employeeid,MAX(orderdate) AS orderdate FROM orders
GROUP BY employeeid) AS B
WHERE A.employeeid=B.EmployeeID AND A.OrderDate=B.orderdate;
```

这句SQL的执行计划如图4-5所示。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	ALL	NULL	NULL	NULL	NULL	17	Using where
1	PRIMARY	A	ref	idx_empid_od_rd_oid,EmployeeID,EmployeesOrders,OrderDate	idx_empid_od_rd_oid	14	B.employeeid,B.orderdate	1	
2	DERIVED	orders	range	NULL	idx_empid_od_rd_oid	5	NULL	17	Using index for group-by

图4-5 采用派生表后的执行计划

在重写的SQL语句中，我们使用子查询作为派生表B，然后再将表A和表B进行联接。这样做的好处是显而易见的，从执行计划中可以看到没有了相关子查询的过程。另外A表也能使用OrderDate的索引，因此这句SQL语句的执行时间缩短为0.064秒。表4-4显示了运行该语句后的结果。

表4-4 采用派生表后产生的结果

orderid	customerid	employeeid	orderdate	requireddate
11077	RATTC	1	1998-05-06	1998-06-03
11070	LEHMS	2	1998-05-05	1998-06-02
11073	PERIC	2	1998-05-05	1998-06-02
11063	HUNGO	3	1998-04-30	1998-05-28
11076	BONAP	4	1998-05-06	1998-06-03
11043	SPECD	5	1998-04-22	1998-05-20
11045	BOTTM	6	1998-04-23	1998-05-21
11074	SIMOB	7	1998-05-06	1998-06-03
11075	RICSU	8	1998-05-06	1998-06-03
11058	BLAUS	9	1998-04-29	1998-05-27

与表4-3的结果是一样的，唯一的不同是结果按照employeeid进行了排序。因为派生表

是通过索引 idx_empid_od_rd_oid 得到的，然后再进行关联操作。

最后，通过 InnoSQL 数据库的慢查询日志来观察数据库实际的 IO，具体情况如下：

```
# User@Host: root[root] @ localhost [127.0.0.1]
# Query_time: 0.064004   Lock_time: 0.074004 Rows_sent: 10   Rows_examined: 28
    Logical_reads: 284 Physical_reads: 20
# 省略SQL语句的显示
```

可以看到这次逻辑 IO 只有 284 次，远小于之前介绍的两种 SQL 语句。

这里再次提醒开发人员，对子查询的编写需要非常小心，尽可能地使用 EXPLAIN 来确认子查询的执行计划，并确认是否可以对其进行进一步优化。在测试机上执行一句 SQL 需要 1 秒的时间看似很短，但这通常是数据量较小的缘故；如果在大数据量的生产环境中，这可能会带来灾难性的后果。

继续相关子查询问题的介绍，在表 4-4 中可以看到 employeeid 为 2 的员工有多个订单。如果只想为每个员工返回一行数据，可能还需要引入一个条件，即最大订单 ID 的那行数据。因此这个问题的解决方案变为：

```
SELECT orderid,customerid,employeeid,orderdate,requireddate
FROM orders AS A
WHERE orderdate =
   ( SELECT MAX(orderdate) FROM orders AS B
          WHERE A.employeeid=B.employeeid)
AND orderid =
   ( SELECT MAX(orderID) FROM orders AS B
          WHERE A.employeeid=B.employeeid)
```

再看表 4-5 中的结果，可以发现这次 employeeid 为 2 的员工只有一行数据。

表 4-5 没有重复 employeeid 结果

orderid	customerid	employeeid	orderdate	requireddate
11077	RATTC	1	1998-05-06	1998-06-03
11073	PERIC	2	1998-05-05	1998-06-02
11063	HUNGO	3	1998-04-30	1998-05-28
11076	BONAP	4	1998-05-06	1998-06-03
11043	SPECD	5	1998-04-22	1998-05-20
11045	BOTTM	6	1998-04-23	1998-05-21
11074	SIMOB	7	1998-05-06	1998-06-03
11075	RICSU	8	1998-05-06	1998-06-03
11058	BLAUS	9	1998-04-29	1998-05-27

上述的 SQL 语句同样可以重写为嵌套子查询：

```
SELECT orderid,customerid,employeeid,orderdate,requireddate
```

```
FROM orders AS A
WHERE orderdate =
( SELECT MAX(orderdate) FROM orders AS B
   WHERE A.employeeid=B.employeeid
   AND A.orderid =
           ( SELECT MAX(orderID) FROM orders AS C
           WHERE A.employeeid=C.employeeid)
)
```

嵌套查询在逻辑上与非嵌套查询完全相同，不同的是非嵌套查询用了 0.64 秒，而嵌套查询用了 18 秒，时间差不多是非嵌套查询的 28 倍！导致这个问题的原因还是当前相关子查询的性能问题，因为在嵌套查询中需要进行两次相关子查询。SQL 的执行计划如表 4-6 所示。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	A	ALL	[redacted]	[redacted]	[redacted]	[redacted]	867	Using where
2	DEPENDENT SUBQUERY	B	ref	idx_empid_od_rd_oid,EmployeeID,EmployeesOrders	idx_empid_od_rd_oid	5	tpcc.A.EmployeeID	54	Using where; Using index
3	DEPENDENT SUBQUERY	C	ref	idx_empid_od_rd_oid,EmployeeID,EmployeesOrders	EmployeeID	5	tpcc.A.EmployeeID	48	Using index

图 4-6 嵌套子查询的执行计划

当然最快的方法还是使用派生表，避免相关子查询和外部查询的多次比较操作，查询过程如下：

```
SELECT C.orderid,A.customerid,A.employeeid,
   B.orderdate,requireddate
FROM orders AS A,
(SELECT employeeid,MAX(orderdate) AS orderdate FROM orders
GROUP BY employeeid) AS B,
(SELECT employeeid,MAX(orderid) AS orderid FROM orders
GROUP BY employeeid) AS C
WHERE A.employeeid=B.EmployeeID
   AND A.OrderDate=B.orderdate
   AND C.employeeid=A.employeeid
   AND C.orderid=A.orderid;
```

4.4 EXISTS 谓词

4.4.1 EXISTS

EXISTS 是一个非常强大的谓词，它允许数据库高效地检查指定查询是否产生某些行。通常 EXISTS 的输入是一个子查询，并关联到外部查询，但这不是必须的。根据子查询是否返回行，该谓词返回 TRUE 或 FALSE。与其他谓词和逻辑表达式不同的是，无论输入子查询是否返回行，EXISTS 都不会返回 UNKNOWN。如果子查询的过滤器为某行返回 UNKNOWN，则表示该行不返回，因此，这个 UNKNOWN 被认为是 FALSE。

在之前的小节中已经演示了 EXISTS 的用法。这里再通过一个例子来演示了 EXISTS 的

用法及其内部的一些特性。下面的查询返回来自西班牙（Spain）且发生过订单的消费者，生成的结果如表 4-6 所示。

```
SELECT customerid,companyname
FROM customers AS A
WHERE country = 'Spain'
   AND EXISTS
          ( SELECT * FROM orders AS B
                WHERE A.customerid = B.customerid )
```

表 4-6　来自西班牙且发生过订单的消费者

customerid	companyname
BOLID	Bólido Comidas preparadas
GALED	Galería del gastrónomo
GODOS	Godos Cocina Típica
ROMEY	Romero y tomillo

外部查询返回来自西班牙的消费者，EXISTS 谓词在表 orders 中匹配在外部查询中遇有相同 customerid 的行数据。

注意　尽管通常不建议在 SQL 语句中使用 *，因为可能会引起一些问题的产生，但是在 EXIST 子查询中 * 可以放心地使用。EXISTS 只关心行是否存在，而不会去取各列的值。

图 4-7 显示了上述 SQL 语句的执行计划。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	PRIMARY	A	ALL	NULL	NULL	NULL	NULL	89	Using where
	2	DEPENDENT SUBQUERY	B	ref	CustomerID,CustomersOrders	CustomerID	16	tpcc.A.CustomerID	4	Using where; Using index

图 4-7　EXISTS 查询的执行计划

通过查询计划可以看到，SQL 优化器首先根据 WHERE 条件先将 country 列为 Spain 的行数据取出，对于每个匹配的 customerid，该执行计划对 orders 表上 customerid 索引进行一次查询，以检查 orders 表中是否有 customerid 的订单。子查询中的索引是必需的，因为它允许直接访问 orders 表的 customerid 行。

有些 DBA 有过一些其他数据库的使用经验，在其他数据库中可能存在这样“幽默”的优化定理，就是将 IN 语句改写为 EXISTS，这样 SQL 查询的效率更高。据我所知，的确曾有过这种说法，这可能是因为当时优化器还不是很稳定和足够优秀。目前在绝大多数的情况下，IN 和 EXISTS 都具有相同的执行计划。但是要注意的是，NOT IN 和 NOT EXISTS 具有非常不同的执行计划。这个问题会在后面的小节中进行解释。

将上述的这句 SQL 重写为 IN 子查询：

```
SELECT customerid,companyname
FROM customers AS A
WHERE country = 'Spain'
   AND customerid IN ( SELECT customerid FROM orders );
```

这个 IN 查询的计划和图 4-7 所示的 EXISTS 计划一样。

4.4.2 NOT EXISTS

EXISTS 与 IN 的一个小区别体现在对三值逻辑的判断上。EXISTS 总是返回 TRUE 或 FALSE，而对于 IN，除了 TRUE、FALSE 值外，还有可能对 NULL 值返回 UNKNOWN。但是在过滤器中，UNKNOWN 的处理方式与 FALSE 相同，因此使用 IN 与使用 EXISTS 一样，SQL 优化器会选择相同的执行计划。

但是输入列表中包含 NULL 值时，NOT EXISTS 和 NOT IN 之间的差异就表现得非常明显了。输入列表中包含 NULL 值时，IN 总是返回 TRUE 和 UNKNOWN，因此 NOT IN 总是返回 NOT TRUE 和 NOT UNKNOWN，即 FALSE 和 UNKNOWN。我们来看下面的例子：

```
mysql>SELECT NULL IN ('a','b', NULL)\G;
*************************** 1. row ***************************
NULL IN ('a','b', NULL): NULL
1 row in set (0.00 sec)

mysql>SELECT NULL NOT IN ('a','b', NULL)\G;
*************************** 1. row ***************************
NULL NOT IN ('a','b', NULL): NULL
1 row in set (0.00 sec)

mysql>SELECT 'a' NOT IN ('a','b', NULL)\G;
*************************** 1. row ***************************
'a' NOT IN ('a','b', NULL): 0
1 row in set (0.00 sec)

mysql>SELECT 'c' NOT IN ('a','b', NULL)\G;
*************************** 1. row ***************************
'c' NOT IN ('a','b', NULL): NULL
1 row in set (0.00 sec)
```

IN 和 NOT IN 的返回值都是显而易见的。NULL IN('a','b',NULL) 返回的是 NULL，因为对 NULL 值进行比较返回的是 UNKNOWN 状态。最后，'c'NOT IN('a','b',NULL) 的结果可能出乎一些人的意料，其返回的是 NULL。因为之前已经说对于包含 NULL 值的 NOT IN 来说，其总是返回 FALSE 和 UNKNOWN，而对于 NOT EXISTS，其总是返回 TRUE 和 FALSE。这就是 NOT EXISTS 和 NOT IN 的最大区别。

例如，我们要返回来自西班牙且没有订单的客户信息，下面是使用 NOT EXISTS 谓词的

解决方案，生成的结果如表 4-7 所示。

```
SELECT customerid,companyname
FROM customers AS A
WHERE country = 'Spain'
   AND NOT EXISTS
          ( SELECT * FROM orders AS B
                 WHERE A.customerid = B.customerid )
```

表 4-7　来自西班牙且没有订单的客户信息

customerid	companyname
FISSA	FISSA Fabrica Inter. Salchichas S.A.

该查询的执行计划如图 4-8 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	PRIMARY	A	ALL	NULL	NULL	NULL	NULL	89	Using where
	2	DEPENDENT SUBQUERY	B	ref	CustomerID,CustomersOrders	CustomerID	16	tpcc.A.CustomerID	4	Using where; Using index

图 4-8　NOT EXISTS 的执行计划

该查询和 EXISTS 的执行计划并没有什么不同，首先过滤来自西班牙的消费者，然后再匹配相关子查询。接着我们再用 NOT IN 来解决这个问题，其返回和 NOT EXISTS 相同的结果。该查询的过程如下：

```
SELECT customerid,companyname
FROM customers AS A
WHERE country = 'Spain'
   AND customerid NOT IN ( SELECT customerid FROM orders )
```

再来看 SQL 语句的执行计划，如图 4-9 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	PRIMARY	A	ALL	NULL	NULL	NULL	NULL	89	Using where
	2	DEPENDENT SUBQUERY	orders	index_subquery	CustomerID,CustomersOrders	CustomerID	16	func	8	Using index

图 4-9　NOT IN 的执行计划

虽然 NOT IN 和 NOT EXISTS 产生相同的结果，但是执行计划却发生了一些细微的改变。在 NOT IN 中，相关子查询中的 type 列变为 index_subquery，而在 NOT EXISTS 中，type 列和 EXISTS 查询一样，都是 ref。

对于 NOT EXIST 和 NOT IN，虽然执行计划不同，但是返回的结果是相同的。这是因为 orders 表中不存在 customerid 为 NULL 的行。若人为地插入以下数据，再来比较 NOT EXISTS 和 NOT IN 之间的区别：

```
INSERT INTO orders(orderid) VALUES (NULL);
```

再次运行 NOT EXISTS 和 NOT IN 查询，就会发现 NOT EXISTS 依旧返回之前的

结果，但是 NOT IN 查询返回空集合，这是因为 orders 表中存在 customerid 为 NULL 的行。所以 NOT IN 的查询返回的是 FALSE 和 UNKNOWN，而不是 TRUE，从而导致我们找不到需要的数据。因此对于使用 NOT IN 的子查询，可以在子查询中先过滤掉 NULL 值，如：

```
SELECT customerid,companyname
FROM customers AS A
WHERE country = 'Spain' AND
   customerid NOT IN
   ( SELECT customerid FROM orders WHERE customerid IS NOT NULL )
```

测试完这些查询，执行下面的语句来移除 customerid 为 NULL 的行。

```
DELETE FROM orders WHERE customerid IS NULL;
```

4.5 派生表

目前为止已经介绍了标量子查询和多值子查询，这一节将介绍派生表。派生表又被称为表子查询，与其他表一样出现在 FROM 的子句中，但是是从子查询派生出的虚拟表中产生的。派生表的使用形式一般如下：

```
FROM ( subquery expression ) AS derived_table_alias
```

目前派生表在使用上有以下使用规则：

- 列的名称必须是唯一的。
- 在某些情况下不支持 LIMIT。

在派生表中，列的名称必须是唯一的，而在一般 SQL 语句中并没有这样的强制规定。例如：

```
mysql>SELECT 'c' AS a ,'b' AS a\G;
*************************** 1. row ***************************
a: c
a: b
1 row in set (0.00 sec)

mysql>SELECT * FROM (SELECT 'c' AS a ,'b' AS a) AS T\G;
ERROR 1060 (42S21): Duplicate column name 'a'
```

另外，对于某些 SQL 语句，派生表也不支持 LIMIT 子句，例如：

```
mysql>SELECT customerid,companyname
    -> FROM customers AS A
    -> WHERE customerid IN ( SELECT customerid FROM orders LIMIT 5 );
ERROR 1235 (42000): This version of MySQL doesn't yet support 'LIMIT & IN/ALL/
    ANY/SOME subquery'
```

注意，派生表是完全的虚拟表，并没有也不可能被物理地具体化，因此优化器不清楚派生表的信息，这对于涉及查看派生表的 EXPLAIN 执行计划来说，速度可能非常慢，例如对于下面的执行计划：

```
EXPLAIN SELECT * FROM ( SELECT * FROM salaries ) AS A
```

这个执行计划在笔者的笔记本电脑上运行需要 9.96 秒，而直接对表 salaries 进行 EXPLAIN 操作瞬间就能得到结果。

前面小节已经演示了将子查询重写为派生表来提高效率，这对 MySQL 数据库来说可能是非常有效的一种调优手段，不过这可能要求开发人员或 DBA 对子查询本身非常熟悉。另外，对于大数据量的表，查看派生表的执行计划可能并不是十分友好。

4.6 子查询可以解决的经典问题

4.6.1 行号

行号是指按顺序为查询结果集的行分配的连续整数。MySQL 数据库在行号方面的支持并不是十分友好，没有像其他数据库一样提供类似的 ROW_NUMBER 解决方案。因此得到行号是一个十分有技巧的问题。先根据如下代码创建 sales 表并填充该表：

```
CREATE TABLE sales (
empid VARCHAR(10) NOT NULL,
mgrid VARCHAR(10) NOT NULL,
qty INT NOT NULL,
PRIMARY KEY (empid)
);
INSERT INTO sales VALUES ( 'A','Z',300 );
INSERT INTO sales VALUES ( 'B','X',100 );
INSERT INTO sales VALUES ( 'C','X',200 );
INSERT INTO sales VALUES ( 'D','Y',200 );
INSERT INTO sales VALUES ( 'E','Z',250 );
INSERT INTO sales VALUES ( 'F','Z',300 );
INSERT INTO sales VALUES ( 'G','X',100 );
INSERT INTO sales VALUES ( 'H','Y',150 );
INSERT INTO sales VALUES ( 'I','X',250 );
INSERT INTO sales VALUES ( 'J','Z',100 );
INSERT INTO sales VALUES ( 'K','Y',200 );
```

运行如下查询，返回如表 4-8 所示的结果。

```
SELECT * FROM sales
```

表 4-8 表 sales 的内容

empid	mgrid	qty	empid	mgrid	qty
A	Z	300	G	X	100
B	X	100	H	Y	150
C	X	200	I	X	250
D	Y	200	J	Z	100
E	Z	250	K	Y	200
F	Z	300			

对于只需要对一个列（唯一列）进行行号统计的问题，如这里根据 empid 进行的行号统计，可以用下面的子查询来完成，最后得到的结果如表 4-9 所示。

```
SELECT empid,
( SELECT COUNT(*) FROM sales AS T2
WHERE T2.empid <= T1.empid) AS rownum
FROM sales AS T1
```

表 4-9 执行行号统计后的结果

empid	rownum	empid	rownum
A	1	G	7
B	2	H	8
C	3	I	9
D	4	J	10
E	5	K	11
F	6		

虽然采用子查询这个解决方案非常直观，但是它运行得非常慢。通过图 4-10 所示的查询计划可以了解其中的缘由。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	PRIMARY	T1	index	NULL	PRIMARY	32	NULL	11	Using index
	2	DEPENDENT SUBQUERY	T2	index	PRIMARY	PRIMARY	32	NULL	11	Using where; Using index

图 4-10 子查询行号方案的执行计划

从执行计划中可以发现，每条记录都需要在相关子查询中进行一次查找。因为 empid 列上已经有索引了，查找起来速度很快，所以对于表中有 N 条记录的行号统计来说，对于第 1 行数据需要扫描一行数据，对于第 2 行需要扫描两行数据，对于第 3 行需要扫描三行数据，……，对于第 N 行需要扫描 N 行数据，一共需要 $N*(N-1)/2$ 行数据。如果 empid 上没有索引，那么每次都需要扫描 N 行数据，一共需要扫描 N^2 行数据。但是不管是否有索引，行号统计子查询的解决方案的扫描成本为 $O(N^2)$。

在 MySQL 数据库中，最快得到行号的解决方案是采用 CROSS JOIN，这会在第 5 章介绍，其时间复杂度为 $O(N)$。

要对多个列排序后进行行号统计，这里需要按照 qty 和 empid 的顺序生成行号。针对这种情况我们可以采用下面的 SQL 语句，最后得到如表 4-10 所示的结果。

```
SELECT empid,qty,
( SELECT COUNT(*) FROM sales AS T2
WHERE T2.qty < T1.qty
OR ( T2.qty = T1.qty AND T2.empid <= T1.empid )) AS rownum
FROM sales AS T1
ORDER BY qty,empid
```

表 4-10　按 qty 和 empid 进行行号统计的结果

empid	qty	rownum	empid	qty	rownum
B	100	1	K	200	7
G	100	2	E	250	8
J	100	3	I	250	9
H	150	4	A	300	10
C	200	5	F	300	11
D	200	6			

以上讨论的行号问题都是行数据不存在重复的情况，若存在多个相同行记录，上述方法都会失效，因为统计出来的结果可能相同，不能作为行号，所以用子查询来解决这个问题就显得更加困难了。例如，先根据下列代码创建表 T 并填充数据。

```
CREATE TABLE T ( a CHAR(1) );

INSERT INTO T SELECT 'X';
INSERT INTO T SELECT 'X';
INSERT INTO T SELECT 'X';
INSERT INTO T SELECT 'Y';
INSERT INTO T SELECT 'Y';
INSERT INTO T SELECT 'Z';
```

这个问题的解决需要依赖前面讨论过的数字辅助表，其最终的 SQL 语句为：

```
SELECT nums.a+smaller AS rownum,C.a FROM (
   SELECT a,COUNT(*) AS count,
          ( SELECT COUNT(*) FROM t AS B
                WHERE B.a < A.a ) AS smaller
   FROM t AS A
   GROUP By a
) AS C,nums
WHERE nums.a <= count
```

该 SQL 语句并不是非常好理解的，特别是对于数字辅助表的使用。但是我们可以从里

到外，一层一层地来分析该语句的构成。先看最内部的子查询：

```
SELECT a,COUNT(*) AS count,
   ( SELECT COUNT(*) FROM t AS B
          WHERE B.a < A.a ) AS smaller
FROM t AS A
GROUP By a
```

上述语句先根据 a 列进行分组，并统计数量及小于该分区数据的数量，最后得到的结果如表 4-11 所示。

表 4-11 查询的第一步结果

a	count	smaller
X	3	0
Y	2	3
Z	1	5

接着使用数字辅助表来对上述产生的结果集进行复制，每行复制的记录数由 count 字段决定。记录 X 需要复制 3 次，记录 Y 需要复制 2 次，记录 Z 需要复制 1 次。与数字辅助表联接的语句如下，执行这段语句后得到的结果如表 4-12 所示。

```
SELECT * FROM (
   SELECT a,COUNT(*) AS count,
          ( SELECT COUNT(*) FROM t AS B
                 WHERE B.a < A.a ) AS smaller
   FROM t AS A
   GROUP By a
) AS C,nums
WHERE nums.a <= count
```

表 4-12 与数字辅助表联接后得到的结果

a	count	smaller	a	a	count	smaller	a
X	3	0	1	Y	2	3	1
X	3	0	2	Y	2	3	2
X	3	0	3	Z	1	5	1

仔细观察表 4-12 可以发现，smaller 列和 a 列的数字相加就是最后需要统计的行号。最终的结果如表 4-13 所示。

表 4-13 最后得到的结果

rownum	a	rownum	a
1	X	4	Y
2	X	5	Y
3	X	6	Z

4.6.2 分区

分区是通过在集合中进行分组操作，再对集合添加分区列来实现的。在子查询的解决方案中，通过在子查询内部添加相关性，并匹配内部表和外部表的分区列来实现分区。例如，对于表 dept_manager，按 dept_no 分区，并按照 emp_no 升序进行分区统计，相应的 SQL 语句如下，得到的最终结果如表 4-14 所示。

```
SELECT dept_no,emp_no,
(SELECT COUNT(*) FROM dept_manager AS S2
WHERE S1.dept_no = S2.dept_no
AND S2.emp_no <= S1.emp_no
) AS rownum
FROM dept_manager AS S1
ORDER BY dept_no,emp_no
```

表 4-14 按照 dept_no 和 emp_no 分区后的统计

dept_no	emp_no	rownum	dept_no	emp_no	rownum
d001	110022	1	d004	110303	1
d001	110039	2	d004	110344	2
d002	110085	1	d004	110386	3
d002	110114	2	d004	110420	4
d003	110183	1	…	…	…
d003	110228	2	d009	111939	4

对于分区问题，假设表中有 m 个分区，每个分区有 n 行数据，则扫描的总行数将是 $m*n*(n-1)/2$，而 $m*n=N$ 为表中的所有数据。因此，当 n 的值较小，即每个分区中含有的数据较少时，扫描的成本为 $O(N)$，而当 n 的值较大时，该子查询解决方案的扫描成本为 $O(N^2)$。

因此，对于表中数据较少的情况，这种方法非常快。如表 dept_manager 中只有 24 行数据，运行上述子查询的方案瞬间就能完成。同样的问题，对于含有 30 万行记录的表 dept_emp 而言，上述子查询的解决方案的运行效率是灾难性的，在笔者的笔记本电脑上运行了 8 个小时也没有得到结果。

但是就这个问题的本身而言，只需要一次扫描就能得到结果，因此对于该问题还可以采用除子查询外的其他解决方案，如基于游标的方案和基于临时表的解决方案。

基于游标的解决方案十分简洁明了。创建一个前向只读的游标，按分区列进行排序，然后再按面向过程的编程方式进行解题，每次将一行数据输入到一张临时表中，即将查询的最终结果存放于该临时表，具体的语句如下：

```
CREATE PROCEDURE pGetPartitionNumber()
BEGIN
```

```
DECLARE done INT DEFAULT 0;
DECLARE deptno CHAR(4) DEFAULT NULL;
DECLARE deptno_prev CHAR(4) DEFAULT NULL;
DECLARE empno INT;
DECLARE rn INT DEFAULT 0;
DECLARE cur1 CURSOR FOR
   SELECT dept_no,emp_no FROM dept_manager
   ORDER BY dept_no,emp_no;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

CREATE TEMPORARY TABLE $ret (
   dept_no CHAR(4),
   emp_no INT,
   rownum INT
)ENGINE=InnoDB;

OPEN cur1;
FETCH cur1 INTO deptno,empno;

START TRANSACTION;
WHILE done = 0 DO
   IF deptno_prev IS NULL THEN
          SET rn = 1;
   ELSEIF deptno = deptno_prev THEN
          SET rn = rn + 1;
   ELSE
          SET rn = 1;
   END IF;
   SET deptno_prev = deptno;
   INSERT INTO $ret SELECT deptno,empno,rn;
   FETCH cur1 INTO deptno,empno;
END WHILE;
COMMIT;

CLOSE cur1;
SELECT * FROM $ret;
DROP TABLE $ret;
END;
```

这里有几个值得注意的地方，首先使用CREATE TEMPORARY TABLE来创建临时表，这样可以在多个线程中并发运行该存储过程而不会有错误产生。其次，这里将产生的临时表指定为InnoDB引擎，对于插入的数据，必须要使用START TRANSACTION，使所有的插入操作放在同一个事务中完成，这样可以极大地提高插入的性能。最后，这里使用了InnoDB引擎，也可以使用MyISAM和Memory引擎，有兴趣的读者可以体验一下不同引擎的运行效率。在笔者的笔记本电脑上，使用InnoDB引擎作为临时表，运行该存储过程需要44.959秒；使用Memory引擎作为临时表需要的运行时间为24.586秒；而MyISAM需要81.261秒。可见基于游标的解决方案要远快于基于子查询集合的解决方案。

最后，介绍基于临时表的解决方案。该方案技巧性最高，执行效率也是最高的。基于临时表的解决方案可以在临时表中创建一个自增长列，按照分区的要求将数据插入临时表。这也是行号问题的另一个解决方案，即通过含有自增长列的临时表来得到行号，这个方法的性能一般也好于基于子查询的集合的解决方案。

而当前问题的特殊之处在于，并不只是要求取得行号，还要根据 dept_no 和 emp_no 进行分区后再进行行号的统计，因此需要再一次借助嵌套查询来解决这个问题，基于临时表的解决方案如下所示：

```
# 创建带有自增长列的临时表
CREATE TABLE $temp
(id INT UNSIGNED AUTO_INCREMENT,
dept_no CHAR(4),
emp_no INT, PRIMARY KEY(id),
UNIQUE INDEX(dept_no)
);

# 根据分区的要求插入数据
INSERT INTO $temp
SELECT NULL,dept_no,emp_no
FROM dept_manager
ORDER BY dept_no,emp_no;

# JOIN运算，通过嵌套子查询得到结果
SELECT m.dept_no,m.emp_no,id-minid+1 AS rownum
FROM $temp AS m
INNER JOIN (
SELECT dept_no, MIN(id) AS minid
FROM $temp
GROUP BY dept_no) AS n
ON m.dept_no = n.dept_no

# 删除临时表
DROP TABLE $temp;
```

就 dept_manager 这张表来说，基于临时表的解决方案并没有带来查询效率的显著提升。但是如果将表 dept_manager 换成拥有 30 万行记录的表 dept_emp，则两种解决方案的结果全然不同，采用基于子查询的集合的解决方案，在笔者笔记本电脑上运行 8 个小时也未得到结果。而采用基于临时表的解决方案，需要 36.141 秒，大部分的时间消耗在向临时表插入数据上，最后的嵌套子查询只需 8 秒就能完成。

在基于临时表的解决方案中获取数据时，采用嵌套查询来得到结果，将行号减去每个分区下最小的行号，在此基础上得到最后的结果。但是，如果采用如下相关子查询的方式来得到结果，那就和之前基于子查询的集合的解决方案没有什么不同了，同样在笔者的笔记本电脑上运行 8 个小时也得不到最终的结果。

```
SELECT dept_no,emp_no,
id-(
   SELECT MIN(id)
          FROM $temp AS s2
          WHERE s2.dept_no = s1.dept_no)+1 AS rownum
FROM $temp AS s1
```

4.6.3 最小缺失值问题

最小缺失值是另一个可用子查询解决的问题，一般应用 EXISTS 谓词。为了说明该问题，首先，创建并填充表 x，过程如下：

```
CREATE TABLE x (
   a INT UNSIGNED PRIMARY KEY,
   b CHAR(1) NOT NULL )ENGINE=InnoDB;

INSERT INTO x SELECT 3,'a';
INSERT INTO x SELECT 4,'b';
INSERT INTO x SELECT 6,'c';
INSERT INTO x SELECT 7,'d';
```

注意，a 列必须是一个正整数，所以这里的类型为 INT UNSIGNED。最小缺失值的问题是，假设列 a 从 1 开始，对于当前表中的数据 3、4、6、7，查询应返回 1。如果当前表的数据为 1、2、3、4、6、7、9，则查询应该返回 5。

最小缺失值的问题可以通过如下的表达式来解决：

```
SELECT
   CASE WHEN NOT EXISTS ( SELECT a FROM x WHERE a=1 ) THEN 1
   ELSE ( ..... 返回最小缺失值的子查询 ..... )
   END
```

如果表中不存在列 a 的值为 1 的情况，则结果返回 1，否则返回子查询的结果，该子查询返回最小缺失值。

下面是通过子查询得到的最小缺失值的过程。

```
SELECT MIN(a)+1 as missing
FROM x AS A
   WHERE NOT EXISTS
          ( SELECT * FROM x AS B
          WHERE A.a+1=B.a )
```

NOT EXISTS 谓词用来判断列 a 的值是否存在一个连续的值，使得 A.a+1=B.a。如果存在，表示是连续值；如果不存在，则表示是缺失值。MIN 函数用来返回最小的缺失值，如果要求返回最大缺失值，则用 MAX 函数。

整个解决方案的 SQL 语句如下：

```
SELECT
CASE
   WHEN NOT EXISTS ( SELECT a FROM x WHERE a=1 ) THEN 1
ELSE
   (SELECT MIN(a)+1 as missing
           FROM x AS A
           WHERE NOT EXISTS
                   ( SELECT * FROM x AS B
                           WHERE A.a+1=B.a ))
END AS missing
```

如果对表 x 执行上述语句，会得到值 1。若按如下方式向列 a 插入值 1 和 2 之后，重新运行上述查询，则会得到结果 5。

```
INSERT INTO x SELECT 1,'z';
INSERT INTO x SELECT 2,'x';
```

若要对最小缺失值进行补缺操作，则可以通过 INSERT...SELECT 语句来实现，如：

```
INSERT INTO x
SELECT
   CASE
   WHEN NOT EXISTS ( SELECT a FROM x WHERE a=1 ) THEN 1
   ELSE
   (SELECT MIN(a)+1 as missing
           FROM x AS A
           WHERE NOT EXISTS
                   ( SELECT * FROM x AS B
                   WHERE A.a+1=B.a ))
   END AS missing,
'p';
```

运行完上述语句再执行下面的语句查询表 x，最终会得到如表 4-15 所示的结果。注意操作生成了 a 为 5 的列，它就是我们补缺的值。

```
SELECT * FROM x;
```

表 4-15　插入最小缺失值后表 x 的记录

a	b	a	b
1	z	5	p
2	x	6	c
3	a	7	d
4	b		

4.6.4　缺失范围和连续范围

在生产环境中连续范围和缺失范围也是常见的问题。首先来创建并填充表 g，过程如下：

```
CREATE TABLE g ( a int );

INSERT INTO g select 1;
INSERT INTO g select 2;
INSERT INTO g select 3;
INSERT INTO g select 100;
INSERT INTO g select 101;
INSERT INTO g select 103;
INSERT INTO g select 104;
INSERT INTO g select 105;
INSERT INTO g select 106;
```

对于表 g，其缺失范围如表 4-16 所示。

表 4-16 缺失范围

start_range	end_range	start_range	end_range
4	99	102	102

对于表 g，其连续范围如表 4-17 所示。

表 4-17 连续范围

start_range	end_range	start_range	end_range
1	3	103	106
100	101		

这里演示的连续范围和缺失范围是整型类型。在生产环境中遇到的类型可能是整型，也有可能是时间类型，但是两者并没有大的区别，稍做修改就可以将该问题转换为时间类型。

对于缺失范围的问题，可以通过下列步骤来解决：

1）找出间断点之前的值，然后对该值加 1，即为 start_range；

2）通过间断点找出下一个值，对该值减 1，即为 end_range。

对于间断点之前的值，可以进行下面的子查询：

```
mysql>SELECT a
    -> FROM g AS A
    -> WHERE NOT EXISTS
    ->  (SELECT * FROM g AS B
    ->          WHERE A.a + 1 = B.a)
    -> ;
+------+
| a    |
+------+
|    3 |
|  101 |
|  106 |
+------+
3 rows in set (0.02 sec)
```

注意，最后的值 106 对查询来说是无用的，因为它是表中最大的值，所以需要过滤掉。而 start_range 的值为间断点加 1，因此最后 start_range 的值可以通过如下 SQL 语句得到：

```
mysql>SELECT a+1 AS start_range
    -> FROM g AS A
    -> WHERE NOT EXISTS
    ->  (SELECT * FROM g AS B
    ->          WHERE A.a + 1 = B.a )
    ->  AND a < ( SELECT MAX(a) FROM g )
    -> ;
+-------------+
| start_range |
+-------------+
|           4 |
|         102 |
+-------------+
2 rows in set (0.00 sec)
```

最后通过子查询为每个间断点返回表 g 中下一个已有的值并减 1，即得到间断点的 end_range，最终的 SQL 语句如下所示：

```
SELECT a+1 AS start_range,
   (SELECT MIN(a)-1 FROM g AS C
           WHERE C.a > A.a ) AS end_range
FROM g AS A
WHERE NOT EXISTS
   (SELECT * FROM g AS B
           WHERE A.a + 1 = B.a )
           AND a < ( SELECT MAX(a) FROM g )
```

这只是该问题的解决方法之一，更为简洁直观的方法是，将表 g 中的数据进行移位匹配，如果是连续的值，那么其差值应该为 1，如果不是连续的值，那么差值就应该大于 1。对于表 g，进行移位匹配后应该得到如表 4-18 所示的内容。

表 4-18　当前值和下一个值的配对

cur	next	cur	next
1	2	103	104
2	3	104	105
3	100	105	106
100	101	106	NULL
101	103		

可以看到 next-cur 的值为 1 表示连续的值，不连续的值为（3，100）、（101，103），而我们要求的不连续范围为（4，99）、（102，102），也就是（cur+1，next−1）就是我们要求的缺失范围。要得到表 4-20 所示的内容，可以执行下述 SQL 语句：

```
SELECT a AS cur,
   (SELECT MIN(a) FROM g AS B
           WHERE B.a > A.a ) AS next
FROM g AS A
```

而要得到最终的结果，只需在对当前值加1，下一个值减1即可。该解决方案的另一个好处是无需处理最大值，因此它的next值为NULL。该解决方案的SQL语句如下所示：

```
SELECT cur+1 AS start_range, next-1 AS end_range
FROM (
   SELECT a AS cur,
          (SELECT MIN(a) FROM g AS B
          WHERE B.a > A.a ) AS next
   FROM g AS A
) AS C
WHERE next - cur > 1
```

对于连续范围的问题，4.6.1节已经给出了解决方案的SQL语句。采用行号来进行分组是最快的一种做法。当然，这个问题也可以通过子查询来完成，不过其效率远没有采用行号分组的方法来得快。

如果采用子查询的方案，我们要手动创建（或得到）一个列，并对这个列进行分组。这个列应该是每个连续分组的最大值，对于{1，2，3}来说，这个最大值就应该为3。计算一组连续值中最大值所依据的原理是：返回大于或等于当前值且后面一个值为间断的最小值。下面是该子查询的SQL语句，生成的输出如表4-19所示。

```
SELECT a,
   (SELECT MIN(a)
           FROM g AS A
           WHERE NOT EXISTS
                  (SELECT * FROM g AS B
                  WHERE A.a + 1 = B.a)
                  AND A.a >= C.a
   ) as max
FROM g AS C
```

表4-19 求出每个分组的最大值

a	max	a	max
1	3	103	106
2	3	104	106
3	3	105	106
100	101	106	106
101	101		

剩下的工作就简单了，在上一步查询中执行如下语句对max列进行分组，得到分组中的最小值和最大值，这就是我们要求的连续范围。

```
SELECT MIN(a) AS start_range, MAX(a) AS end_range
FROM (
   SELECT a,
          (SELECT MIN(a)
                 FROM g AS A
                 WHERE NOT EXISTS
                 (SELECT * FROM g AS B
                        WHERE A.a + 1 = B.a)
                        AND A.a >= C.a
   ) as max
FROM g AS C ) AS D
GROUP BY max
```

上述查询给出了连续范围问题的解决方案，但是其性能是值得商榷的。这里的扫描成本变为了 $O(N^2)$。对于表中数据量非常大的情况，其性能又会变得十分糟糕。因此解决连续范围问题的最优方案是采用行号方法。因为已介绍过实现思路，这里仅给出 SQL 语句：

```
SELECT MIN(a) start_range, MAX(a) end_range
FROM
(
   SELECT a,rn,a-rn AS diff
       FROM
           (SELECT a,@a:=@a+1 rn FROM g,
           (SELECT @a:=0) AS a) AS b
) AS c
GROUP BY diff
```

4.7 MariaDB 对 SEMI JOIN 的优化

4.7.1 概述

前面已经提到 MariaDB 5.3 优化器提供了对于 SEMI JOIN 的优化。这一节主要介绍如何通过 MariaDB 新增的优化器功能来提高子查询的性能。对于用户来说，这一切都是透明的，MariaDB 5.3.3 已经默认开始对于 SEMI JOIN 进行优化。

那什么是 SEMI JOIN 呢？其 SQL 语句的一般形式如下：

```
SELECT ... FROM outer_tables
WHERE expr IN (SELECT ... FROM inner_tables ...) AND ...
```

从严格的数学定义来说，SEMI JOIN 的定义为：

```
t1 SEMI JOIN t2 ON sj_cond =
{ t1.row |∃ t2.row, sj_cond(t1.row, t2.row) = true}
```

由于目前 Oracle 和 MySQL 都将 SEMI JOIN 转换为了 EXISTS 语句，因此在执行效率

上显得非常低。从理论上来说，SEMI JOIN 应该只需要关心外部表中与子查询匹配的部分即可。这就是 MariaDB 要对 SEMI JOIN 进行的优化，在 MariaDB 中子查询变得实际可用得多，效率也得到了极大的提升。如果用户在实际环境中需要使用大量的 SEMI JOIN 子查询，那么 MariaDB 5.3 是最好的选择。从另一方面讲，如果用户能理解 MariaDB 对于子查询所做的优化，就能够将这些优化用在所编写的 SQL 语句中。

4.7.2 Table Pullout 优化

有些时候，一个子查询可以被重写为 JOIN，例如：

```
SELECT o_custkey FROM orders
WHERE o_custkey IN
    ( SELECT c_custkey FROM customer
        WHERE c_acctbal < -500 );
```

如果知道 c_custkey 是唯一的，即主键或唯一索引，那么上述的 SQL 语句可以被重写为如下形式：

```
SELECT o_custkey FROM orders,customer
WHERE o_custkey = c_custkey
AND c_acctbal < -500;
```

Table Pullout 的作用就是根据唯一索引将子查询重写为 JOIN 语句。在 MySQL 5.5 中，上述 SQL 语句的执行计划如图 4-11 所示。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	orders	index	NULL	i_o_custkey	5	NULL	1502510	100.00	Using where; Using index
2	DEPENDENT SUBQUERY	customer	unique_subquery	PRIMARY,idx_c_acctbal	PRIMARY	4	func	1	100.00	Using where

图 4-11 MySQL 5.5 的执行计划

如果通过 EXPLAIN EXTENDED 和 SHOW WARNINGS 命令，可以看到如下的结果：

```
select `dbt3`.`orders`.`o_custkey` AS `o_custkey` from
`dbt3`.`orders` where
<in_optimizer>(`dbt3`.`orders`.`o_custkey`,<exists>(<primary_index_lo
    okup>(<cache>(`dbt3`.`orders`.`o_custkey`) in customer on PRIMARY
    where ((`dbt3`.`customer`.`c_acctbal` < <cache>(-(500))) and
    (<cache>(`dbt3`.`orders`.`o_custkey`) = `dbt3`.`customer`.`c_custkey`)))))
```

而在 MariaDB 5.3 中，优化器会对 SQL 语句进行重写，得到的执行计划如图 4-12 所示。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	customer	range	PRIMARY,idx_c_acctbal	idx_c_acctbal	9	NULL	6802	100.00	Using where; Using index
1	PRIMARY	orders	ref	i_o_custkey	i_o_custkey	5	dbt3.customer.c_custkey	7	100.00	Using index

图 4-12 MariaDB 5.3 的执行计划

从图 4-12 可以发现，在 MariaDB 中，优化器没有将独立子查询重写为相关子查询。通

过 EXPLAIN EXTENDED 和 SHOW WARNINGS 命令，得到优化器的执行方式为：

```
select `dbt3`.`orders`.`o_custkey` AS `o_custkey` from
`dbt3`.`customer` join `dbt3`.`orders` where
((`dbt3`.`orders`.`o_custkey` = `dbt3`.`customer`.`c_custkey`) and
(`dbt3`.`customer`.`c_acctbal` < -(500)))
```

很显然，优化器将上述子查询重写为 JOIN 语句，这就是 Table Pullout 优化。表 4-20 显示了上述子查询分别在 MariaDB 5.3 和 MySQL 5.5 中的执行时间。

表 4-20　子查询在 MySQL 5.5 和 MariaDB 5.3 中的执行时间对比

	MySQL 5.5	MariaDB 5.3
无预热	16.459	9.391
预热	2.730	0.702

上述 SQL 语句中选择的存储引擎为 InnoDB。预热是指所要读取的表中的数据都已经在 InnoDB 存储引擎的缓冲池中，这时不涉及磁盘的读取。而无预热指的是数据库刚启动，缓冲池中没有数据，需要读取磁盘上的数据到缓冲池。

可以看到，在无预热的情况下，对于上述子查询，MariaDB 比 MySQL 要快 43%。对于数据已经预热的情况下，上述子查询在 MariaDB 中的执行时间接近 MySQL 中的四分之一。

4.7.3　Duplicate Weedout 优化

前一小节提到内部表查出的列是唯一的，因此 MariaDB 优化器会将子查询重写为 JOIN 语句，以提高 SQL 执行的效率。Duplicate Weedout 优化是指外部查询条件的列是唯一的，MariaDB 优化器会先将子查询查出的结果进行去重，这个步骤被称为 Duplicate Weedout 或者 Duplicate Elimination。我们先来看下面的 SQL 语句：

```
SELECT Name FROM City
WHERE
City.ID IN
(SELECT Capital FROM Country
WHERE SurfaceArea > 1000000 AND
Country.Population >10*City.Population);
```

因为 City.ID 是主键，所以应该对子查询得到的结果进行去重。在 MariaDB 数据库下，上述 SQL 语句的执行计划如图 4-13 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	PRIMARY	Country	ALL	NULL	NULL	NULL	NULL	201	Using where; Start temporary
	1	PRIMARY	City	eq_ref	PRIMARY	PRIMARY	4	world.Country.Capital	1	Using where; End temporary

图 4-13　MariaDB 中的 Duplicate Weedout 的执行计划

Extra 选项提示的 Start temporary 表示创建了一张去重的临时表，End temporary 表示删除该临时表。而通过 EXPLAIN EXTENDED 和 SHOW WARNINGS 命令还可以发现：

```
select `world`.`city`.`Name` AS `Name` from `world`.`city` semi
    join (`world`.`country`) where ((`world`.`city`.`ID` =
    `world`.`country`.`Capital`) and (`world`.`country`.`SurfaceArea` > 1000000)
    and (`world`.`country`.`Population` > (10 * `world`.`city`.`Population`)))
```

通过上述的清单可以发现，与 Table Pullout 不同的是，Duplicate Weedout 显示的是 SEMI JOIN 而不是 JOIN，其中原因在于多了一步去重的工作。对于上述的执行计划，其扫描成本约为 201+201*1=402 次。在无预热的情况下，执行所需时间为 0.109 秒。

而在 MySQL 5.5 下其执行计划如图 4-14 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	PRIMARY	City	ALL	NULL	NULL	NULL	NULL	3868	100.00	Using where
	2	DEPENDENT SUBQUERY	Country	ALL	NULL	NULL	NULL	NULL	264	100.00	Using where

图 4-14 MySQL 5.5 中的 Duplicate Weedout 的执行计划

可以看到，在 MySQL 5.5 中该语句是相关子查询，扫描成本约为 3868+3868*246=1 025 020 次。在无预热的情况下，执行该计划所需时间为 2.293 秒。

4.7.4 Materialization 优化

如果子查询是独立子查询，则优化器可以选择将独立子查询产生的结果填充到单独一张物化临时表（materialized temporary table）中，其实现原理如图 4-15 所示。

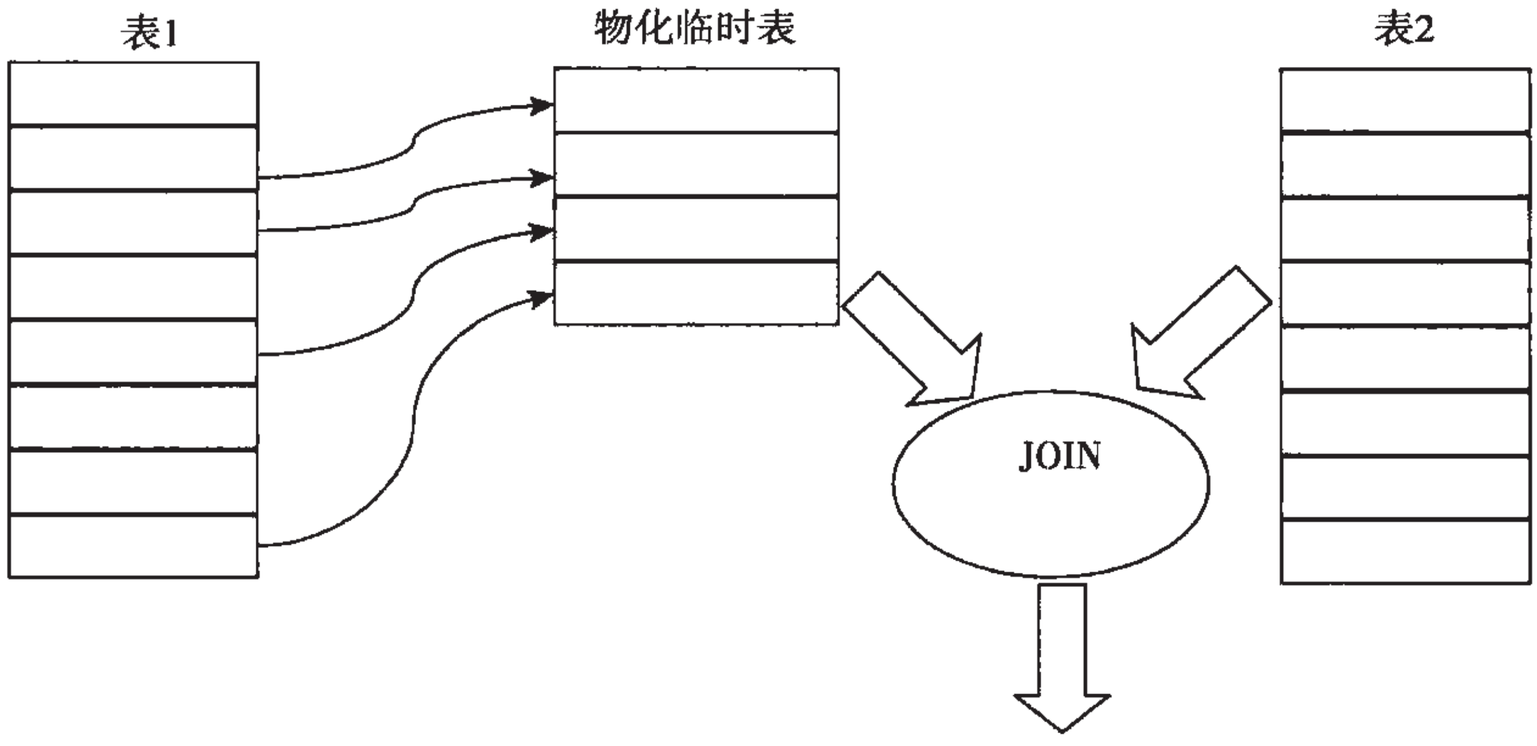


图 4-15 Materialization 优化原理

根据 JOIN 的顺序，Materialization 优化可分为：

- Materialization scan：JOIN 是将物化临时表和表进行联接。
- Materialization lookup：JOIN 是将表和物化临时表进行联接。

4.2 节已经涉及了 MariaDB SEMI JOIN 的优化，其语句为：

```
SELECT orderid,customerid,employeeid,orderdate
FROM orders
WHERE orderdate IN
   ( SELECT MAX(orderdate)
          FROM orders
          GROUP BY (DATE_FORMAT(orderdate,'%Y%M'))
   )
```

在 MariaDB 5.3 中，上述 SQL 语句的执行计划如图 4-16 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	PRIMARY	<subquery2>	ALL	distinct_key	NULL	NULL	NULL	795	100.00	Using where
	1	PRIMARY	orders	ref	OrderDate	OrderDate	9	<subquery2> MAX(orderdate)	1	100.00	
	2	SUBQUERY	orders	index	NULL	OrderDate	9	NULL	795	100.00	Using index; Using temporary

图 4-16　子查询在 MariaDB 中的执行计划

可以看到，在进行 JOIN 时（也就是 id 为 1 的步骤），先扫描的表是 <subquery2>，然后是 orders，因此这是 Materialize scan 优化。下面的子查询同样可以利用 Materialization 来进行优化。

```
SELECT * FROM part
WHERE p_partkey IN
    ( SELECT l_partkey FROM lineitem
       WHERE l_shipdate BETWEEN '1997-01-01' AND '1997-02-01' )
ORDER BY p_retailprice DESC LIMIT 10;
```

其执行计划如图 4-17 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	PRIMARY	part	ALL	PRIMARY	NULL	NULL	NULL	199755	Using temporary; Using filesort
	1	PRIMARY	<subquery2>	eq_ref	distinct_key	distinct_key	5	func	1	
	2	MATERIALIZED	lineitem	range	i_l_shipdate,i_l_suppkey_partkey,i_l_partkey	i_l_shipdate	4	NULL	157032	Using index condition

图 4-17　执行计划

可以看到上述的 SQL 语句和之前的 SQL 语句都可以通过 Materialization 来进行优化，但是在进行 JOIN 时顺序还是有所不同的。图 4-17 显示，这次 SQL 语句先扫描 part 表，然后再来联接 <subquery2> 这张物化的临时表，因此这时为 Materialization lookup 优化。

4.8　小结

本章主要介绍子查询，以及与子查询相关的一系列问题，讨论了标量子查询、表子查

询、相关子查询、嵌套子查询、行号问题、连续范围和缺失范围问题。另外，对于子查询的优化给出了各种解决方案，其中将相关查询的子查询转换为嵌套子查询是非常有效的一种方法，前提当然是可以进行转化。最后介绍 MariaDB 对于 SEMI JOIN 子查询的优化。本章包含了很多技巧性的问题，值得读者好好回味。掌握这些技巧并善于应用这些技巧，将会使你对子查询的应用更加得心应手。

第 5 章

联接与集合操作

本章介绍联接和集合操作。联接操作主要讲解 MySQL 数据库支持的三种联接方式，同时也给出了不同 ANSI 标准的语法。本章也对联接的内部实现算法进行了详细的介绍，这样能给用户更清晰的认识，使其选择正确的联接算法。最后对集合操作进行介绍，虽然 MySQL 数据库只支持 UNION 集合操作，但用户可以在 UNION 的基础之上实现 EXCEPT 和 INTERSECT 集合操作。

5.1　联接查询

联接查询是一种常见的数据库操作，即在两张表（或更多表）中进行行匹配的操作。一般称之为水平操作，这是因为对几张表进行联接操作所产生的结果集可以包含这几张表中所有的列。对应于联接的水平操作，一般将集合操作视为垂直操作。

MySQL 数据库支持如下的联接查询：

- CROSS JOIN（交叉联接）
- INNER JOIN（内联接）
- OUTER JOIN（外联接）
- 其他

在进行联接操作时，请牢记第 3 章描述的逻辑查询处理阶段，尤其是关于联接所涉及的阶段。每个联接都只发生在两个表之间，即使 FROM 子句中包含多个表也是如此。每次联接操作也只进行逻辑操作的前三个步骤，每次产生一个虚拟表，这个虚拟表再依次与 FROM 子句的下一个表进行联接，重复上述步骤，直到 FROM 子句中的表都被处理完为止。

需要注意的是，不同联接类型执行的步骤不同。对于 CROSS JOIN，只应用第一个阶段的笛卡儿积。INNER JOIN 应用第一和第二个步骤，OUTER JOIN 应用所有的前三个步骤。

5.1.1　新旧查询语法

MySQL 数据库支持两种不同的联接操作语法。在实际使用中，很少有 DBA 或开发人员可以观察到或区分出两者的不同。表 5-1 给出了两种不同的联接查询语法。

表 5-1　两种不同的联接查询语法

SQL 语句	
SELECT ... FROM a,b WHERE a.x = b.x	SELECT ... FROM a INNER JOIN b ON a.x = b.x

熟悉 MySQL 数据库的 DBA 和开发人员可能知道两条 SQL 语句产生的结果是一样的，在哪种场合使用完全取决于个人的习惯。那么两者有区别吗？这个问题就是本小节所要阐述

的重点，总地来说本小节将介绍如下内容：

- ❑ 产生两种新旧联接查询语法的历史背景。
- ❑ 什么时候使用哪种语法更好？
- ❑ 哪种语法执行得更好？
- ❑ 哪种语法是标准的？
- ❑ 旧语法会不会过时？

SQL（Structured Query Language，结构化查询语言）是用于数据库中的标准数据查询语言，IBM 公司最早使用在其开发的数据库系统中。1986 年 10 月，美国国家标准学会 (ANSI) 对 SQL 进行规范后，以此作为关系式数据库管理系统的标准语言 (ANSI X3. 135—1986)，1987 年在国际标准化组织的支持下成为国际标准。

1989 年，ANSI 采纳在 ANSI X3.135—1989 报告中定义的关系数据库管理系统的 SQL 标准语言，将其称为 ANSI SQL 89，该标准替代 ANSI X3.135—1986 版本。这是最早的 ANSI SQL 标准，之后又定义了 ANSI SQL92、ANSI SQL 99、ANSI SQL 2003。

对于表 5-1 左侧的 SQL 联接查询语句，其由 ANSI SQL 89 标准引入，与新语法的区别是 FROM 子句中的表名之间用逗号分隔，没有 JOIN 关键字，也没有 ON 子句，其语法格式如下：

```
FROM T1,T2
WHERE where_condition
```

ANSI SQL 89 只支持 CROSS JOIN 和 INNTER JOIN，不支持 OUTER JOIN。新语法是由 ANSI SQL 92 引入的，与旧语法的区别是引入了 JOIN 关键字和 ON 过滤子句，并去掉了表之间的逗号，其语法格式如下：

```
FROM T1
<join_type>JOIN T2
ON on_condition
WHERE where_condition
```

ANSI SQL 92 引入了对外部联接的支持，因此要严格区分 ON 过滤器和 WHERE 过滤器的作用，这在第 3 章已经介绍过，这章将在外部联接中展开更为详细的讨论。

虽然有新旧两种语法，但是 MySQL 数据库同时支持这两种语法，并保证对所有这两种 ANSI SQL 标准进行兼容，因此不必担心过时的问题。对于表 5-1 的两条 SQL 语句，两者的逻辑查询和物理查询也是相同的。

5.1.2 CROSS JOIN

CROSS JOIN 对两个表执行笛卡儿积，返回两个表中所有列的组合。若左表有 *m* 行数据，右表有 *n* 行数据，则 CROSS JOIN 将返回 *m***n* 行的表。

下面的查询生成了 employees 数据库的 dept_manager 表的所有可能的组合。

```
SELECT a.emp_no,b.emp_no
FROM dept_manager AS a
CROSS JOIN
dept_manager AS b;
```

因为 dept_manager 表共有 24 行记录，所以进行 CROSS JOIN 后共有 576 行记录。另外，因为是表 dept_manager 自己与自己进行联接操作，所以一定要指定别名，否则会出现如下的错误：

```
mysql> SELECT * FROM dept_manager
   ->CROSS JOIN dept_manager;
ERROR 1066 (42000): Not unique table/alias: 'dept_manager'
```

也可以使用下面的 ANSI SQL 89 语法来实现相同任务的查询。

```
SELECT *
FROM dept_manager AS a,dept_manager AS b;
```

对于交叉联接，笔者更喜欢使用 ANSI SQL 89 语法。这样代码会更短，语法更加易读。不必担心两者的性能，因为正如前面所说的，优化器将为两者生成相同的执行计划。

CROSS JOIN 的一个用处是快速生成重复测试数据，因为通过它可以很快地构造 *m*n*o* 行的数据。假设要生成网络游戏中用户每天购买道具的订单表，表 User 代表用户表，表 Item 代表道具表，可以快速通过如下的 CROSS JOIN 来生成每个用户购买道具的测试数据。

```
SELECT username, itemname,
   date_add('2011-07-01',interval a-1 day) as date
FROM User, Item, Nums;
WHERE Nums.a <= 31;
```

这里使用了第 2 章介绍的数字辅助表，用来快速生成 7 月的所有天数，因此上述 SQL 语句生成了用户在 7 月的每天购买所有道具的一些测试数据。假设 User 表有 1000 行数据，Item 表有 100 行数据，则一共可以生成 310 万行的数据（1000*100*31）。

CROSS JOIN 的另一个用处是可以作为返回结果集的行号。如对于表 dept_emp 表，得到额外的行号的结果集，如表 5-2 所示。

表 5-2　含有行号的结果集

emp_no	dept_no	row_num	emp_no	dept_no	row_num
10001	d005	1	10007	d008	7
10002	d007	2	10008	d005	8
10003	d004	3	10009	d006	9
10004	d004	4	10010	d004	10
10005	d003	5	…	…	…
10006	d005	6			

row_num 是用户想要得到的行号，解决这个问题可以使用下面子查询的 SQL 语句：

```
SELECT emp_no,dept_no,
(SELECT COUNT(1) FROM dept_emp t2
WHERE t1.emp_no<=t2.emp_no) AS row_num
FROM dept_emp t1;
```

尽管上述的 SQL 语句十分简单，也很好理解，执行之后能得到我们需要的行号，可是它运行得非常慢。要理解其中的原因我们可以看如图 5-1 所示的执行计划。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	PRIMARY	t1	index	NULL	emp_no	4	NULL	331883	Using index
	2	DEPENDENT SUBQUERY	t2	index	PRIMARY,emp_no	emp_no	4	NULL	331883	Using where; Using index

图 5-1　用子查询求解行号的执行计划

首先要扫描整个表 t1，得到所有的行，大约 30 多万行。然后将返回的每一行数据与表 t2 进行联接操作。每次行号计算都会涉及一次子查询的扫描操作。

在如下的子查询中：

```
FROM dept_emp t2
WHERE t1.emp_no<=t2.emp_no
```

第 1 次返回一行数据，第 2 次返回二行数据，第 3 次返回三行数据，……，第 N 次需返回 N 行数据，因此扫描的总行数是 1+2+3+…+N。你可能还没有意识到将有多少行数据被扫描。对于 dept_emp 这张一共有 30 万行的表来说，一共需要扫描 15 000 150 000 行，150 亿行！其实这几乎已经和笛卡儿积的扫描成本一样了（都是 $O(N^2)$），对 30 万行的表进行笛卡儿积也就需要扫描 900 亿行。

对于这个问题可以用 CROSS JOIN 来解决。虽然对两个 N 行表进行笛卡儿积会产生 N^2 行的数据。但是如果是对一行表与 N 行表进行 CROSS JOIN，笛卡尔儿积返回的还是 N 行数据。因此我们可以使用如下的 SQL 语句：

```
SELECT emp_no,dept_no,@a:=@a+1 AS row_num
FROM dept_emp,(SELECT @a:=0 ) t;
```

这里的 SELECT @a:=0 产生了只有一行的数据，因此虽然还需要扫描表 dept_emp 中所有的行，但是每行只和一行数据进行 CROSS JOIN，产生一行的记录，这要远快于前面介绍的 SQL 查询语句。使用 CROSS JOIN 求解行号问题的执行计划如图 5-2 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	PRIMARY	<derived2>	system	NULL	NULL	NULL	NULL	1	
	1	PRIMARY	dept_emp	index	NULL	emp_no	4	NULL	331883	Using index
	2	DERIVED	NULL	NULL	NULL	NULL	NULL	NULL	NULL	No tables used

图 5-2　使用 CROSS JOIN 求解行号问题的执行计划

可以看到，表 <derived2> 只有一行数据，因此用 CROSS JOIN 查询就大大提高了速度。很明显，这条 SQL 语句扫描成本为 $O(N)$。

5.1.3　INNER JOIN

通过 INNER JOIN 用户可以根据一些过滤条件来匹配表之间的数据。在逻辑查询的前三个处理阶段中，INNER JOIN 应用前两个阶段，即首先产生笛卡儿积的虚拟表，再按照 ON 过滤条件来进行数据的匹配操作。INNER JOIN 没有第三步操作，即不添加外部行，这是和 OUTER JOIN 最大的区别之一。也正因为不会添加外部行，指定过滤条件在 ON 子句和 WHERE 子句中是没有任何区别的。

如果使用的是 ANSI 92 语法，则选择在哪个子句中指定过滤条件，用户具有更多的灵活性。因为前面说了，从逻辑上讲，在哪里指定过滤条件都是一样的，通常不会有性能上的差异。唯一的准则就是可读性强。通过一种让 DBA、开发人员感觉更自然的方式进行代码编写。例如，在表之间匹配记录的过滤器放在 ON 子句中，而只从一个表中过滤数据的条件放在 WHERE 子句中。下面语句实现的是找出部门为 d001 的经理的用户编号、姓名。

```
SELECT a.emp_no,first_name,last_name
FROM employees a
[INNER] JOIN dept_manager b
ON a.emp_no = b.emp_no
WHERE dept_no = 'd001';
```

INNER 关键字可省略。前面说过，INNER JOIN 中 WHERE 的过滤条件可以写在 ON 子句中，因此下面的 SQL 查询语句得到的结果和上面是一样的。

```
SELECT a.emp_no,first_name,last_name
FROM employees a
INNER JOIN dept_manager b
ON a.emp_no = b.emp_no
AND dept_no = 'd001';
```

如果使用 ANSI 89 语法，用户只能在 WHERE 条件中指定所有的过滤条件，其查询语句如下：

```
SELECT a.emp_no,first_name,last_name
FROM employees a,dept_manager b
WHERE a.emp_no = b.emp_no
AND dept_no = 'd001';
```

对于 CROSS JOIN，笔者喜欢使用 ANSI 89 语法，而对于 INNER JOIN 正好相反，更倾向于使用 ANSI 92 语法。如果忘记指定联接条件，则使用 ANSI 89 语法可能有些危险，因为可能会得到很大的笛卡儿积返回集，如下面的代码所演示的那样。

```
SELECT a.emp_no,first_name,last_name
FROM employees a,dept_manager b
WHERE dept_no = 'd001';
```

特别需要注意的是，在 MySQL 数据库中，如果 INNER JOIN 后不跟 ON 子句，也是可以通过语法解析器的，这时 INNER JOIN 等于 CROSS JOIN，即产生笛卡儿积，示例如下：

```
SELECT * FROM
employees AS a
INNER JOIN dept_manager AS b
```

这一点和很多其他关系数据库非常不同，例如，在 Microsoft SQL Server 中必须指定 ON 子句，否则语法解析器会抛出异常。而在 MySQL 数据库中，CROSS JOIN 其实和 INNER JOIN 是同义词的关系，因此当没有 ON 子句时，SQL 解析器会将 INNER JOIN 理解为 CROSS JOIN。

此外，如果 ON 子句中的列具有相同的名称，可以使用 USING 子句来进行简化，得到的结果和上述两语法的语句结果是一样的：

```
SELECT a.emp_no,first_name,last_name
FROM employees a
INNER JOIN dept_manager b
USING(emp_no)
WHERE dept_no = 'd001';
```

5.1.4 OUTER JOIN

通过 OUTER JOIN 用户可以按照一些过滤条件来匹配表之间的数据。与 INNER JOIN 不同的是，在通过 OUTER JOIN 添加的保留表中存在未找到的匹配数据。MySQL 数据库支持 LEFT OUTER JOIN 和 RIGHT OUTER JOIN。与 INNER 关键字一样，可以省略 OUTER 关键字。

注意 目前 MySQL 数据库不支持 FULL OUTER JOIN。

OUTER JOIN 应用逻辑查询的前三个步骤，即产生笛卡儿积、应用 ON 过滤器和添加外部行。对于保留表中的行数据，如果是未找到匹配数据而被添加的记录，其值用 NULL 进行填充。在第 3 章的查询处理中我们已经接触过了 LEFT JOIN，即返回客户信息、订单信息，同时返回没有订单的客户，因为指定了 LEFT 关键字，因此表 customers 是保留表。查询过程如下：

```
SELECT c.customer_id, o.order_id
FROM customers as c
LEFT OUTER JOIN orders as o
ON c.customer_id = o.customer_id
```

OUTER JOIN 只在 ANSI SQL 92 中得到支持，在其他一些数据库中可以使用 (+)=、*= 来表示 LEFT JOIN，用 =(+)、=* 来扩展 ANSI SQL 89 语法使其支持 OUTER JOIN。但是对 MySQL 数据库来说，只有一种 OUTER JOIN 的联接语法。

通过 OUTER JOIN 和 IS NULL，可以返回没有用户订单的客户，例如：

```
SELECT c.customer_id
FROM customers as c
LEFT OUTER JOIN orders as o
ON c.customer_id = o.customer_id
WHERE o.order_id IS NULL
```

可以得到下面一条记录：

```
mysql>SELECT c.customer_id
    -> FROM customers as c
    -> LEFT OUTER JOIN orders as o
    -> ON c.customer_id = o.customer_id
    -> WHERE o.order_id IS NULL
    -> \G;
*************************** 1. row ***************************
customer_id: baidu
1 row in set (0.00 sec)
```

需要注意的是，INNER JOIN 中的过滤条件都可以写在 ON 子句中，而 OUTER JOIN 的过滤条件不可以这样处理，因为可能会得到不正确的结果，例如：

```
mysql>SELECT c.customer_id, o.order_id
    -> FROM customers as c
    -> LEFT OUTER JOIN orders as o
    -> ON c.customer_id = o.customer_id and o.order_id IS NULL\G;
*************************** 1. row ***************************
customer_id: 163
   order_id: NULL
*************************** 2. row ***************************
customer_id: 9you
   order_id: NULL
*************************** 3. row ***************************
customer_id: baidu
   order_id: NULL
*************************** 4. row ***************************
customer_id: TX
   order_id: NULL
4 rows in set (0.04 sec)
```

这次得到了 4 条记录，显然这不是我们想要的结果。

对于 OUTER JOIN，同样可以使用 USING 来简化 ON 子句，因此上述的语句可以改写为：

```
SELECT c.customer_id, o.order_id
FROM customers as c
LEFT OUTER JOIN orders as o
USING(customer_id)
```

与 INNER JOIN 不同的是，对于 OUTER JOIN，必须制定 ON 子句，否则 MySQL 数据库会抛出异常，例如：

```
mysql>SELECT c.customer_id, o.order_id
    -> FROM customers as c

    -> LEFT OUTER JOIN orders as o;
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that
    corresponds to your MySQL server version for
the right syntax to use near '' at line 3
```

第 4 章介绍了用子查询来解决最小缺失值的问题，利用 OUTER JOIN 同样可以解决该问题。先运行如下代码：

```
CREATE TABLE t (
   a INT,
   b VARCHAR(5),
   PRIMARY KEY(a)
)ENGINE=INNODB;

INSERT INTO t SELECT 1,'Z';
INSERT INTO t SELECT 2,'P';
INSERT INTO t SELECT 3,'G';
INSERT INTO t SELECT 5,'E';
INSERT INTO t SELECT 6,'F';
INSERT INTO t SELECT 7,'B';
INSERT INTO t SELECT 9,'V';
```

最小缺失值的问题是找出记录中不连续的最小值，在这个例子中显然是 4，我们可以通过 OUTER JOIN 来进行查询，具体的解决方案如下：

```
SELECT MIN(a.a+1)
FROM t x
LEFT OUTER JOIN t y
ON x.a + 1 = y.a
WHERE y.a IS NULL
```

该解决方案的第一步是对 t 表应用 LEFT OUTER JOIN，联接的条件是 x.a+1=y.a。因为是 LEFT OUTER JOIN，所以需要涉及逻辑查询的前三个步骤，笛卡儿积、ON 过滤器和 WHERE 过滤器。首先，忽略 WHERE 过滤器，看看执行到逻辑查询第二步时虚拟表的情况。ON 过滤器的条件是 x 表中每行 a 值比 y 表中每行 a 值大 1，因为使用的是 OUTER JOIN，所以未在保留表中存在的行将作为外部行被添加，产生的虚拟表如表 5-3 所示。

表 5-3　最小缺失值产生的中间虚拟表（1）

x.a	x.b	y.a	y.b
1	Z	2	P
2	P	3	G
3	G	NULL	NULL
5	E	6	F
6	F	7	B
7	B	NULL	NULL
9	V	NULL	NULL

接着，应用 WHERE 条件，过滤条件为 `y.a IS NULL`，产生虚拟表如表 5-4 所示。

表 5-4　最小缺失值产生的中间虚拟表（2）

x.a	x.b	y.a	y.b
3	G	NULL	NULL
7	B	NULL	NULL
9	V	NULL	NULL

最后应用 MIN 函数，取出最小的不连续值 4。

5.1.5　NATURAL JOIN

ANSI SQL 还支持 NATURAL JOIN，即自然联接。NATURAL JOIN 等同于 INNTER JOIN 与 USING 的组合，它隐含的作用是将两个表中具有相同名称的列进行匹配。同样的，NATURAL LEFT(RIGHT) JOIN 等同于 LEFT(RIGHT) OUTER JOIN 与 USING 的组合。对于下面这句 INNER JOIN：

```
SELECT a.emp_no,first_name,last_name
FROM employees a
INNER JOIN dept_manager b
ON a.emp_no = b.emp_no
```

上述语句和如下的 NATURAL JOIN 是等价的：

```
SELECT a.emp_no,first_name,last_name
FROM employees a
NATURAL JOIN dept_manager b
```

5.1.6　STRAIGHT_JOIN

STRAIGHT_JOIN 其实不是新的联接类型，而是用户对 SQL 优化器的控制，其等同于 JOIN。通过 STRAIGHT_JOIN，MySQL 数据库会强制先读取左边的表。先看一个未使用 STRAIGHT_JOIN 的 SQL 语句的执行计划，其 SQL 语句如下，得到的执行计划如图 5-3 所示。

```
SELECT a.emp_no,first_name,last_name
FROM employees a
INNER JOIN dept_manager b
ON a.emp_no = b.emp_no;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	b	index	PRIMARY,emp_no	emp_no	4	NULL	24	Using index
1	SIMPLE	a	eq_ref	PRIMARY	PRIMARY	4	employees.b.emp_no	1	

图 5-3　未使用 STRAIGHT_JOIN 时的执行计划

可以看到，MySQL 数据库先选择 b 表，也就是 dept_manager 表，然后进行匹配。这样做的好处是实际只进行了 24 次行匹配。如果使用 STRAIGHT_JOIN，则会强制使用左表，也就是 employees 表，例如：

```
EXPLAIN SELECT a.emp_no,first_name,last_name
FROM employees a
STRAIGHT_JOIN dept_manager b
ON a.emp_no = b.emp_no;
```

得到的执行计划如图 5-4 所示。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	a	ALL	PRIMARY	NULL	NULL	NULL	300806	
1	SIMPLE	b	ref	PRIMARY,emp_no	PRIMARY	4	employees.a.emp_no	1	Using index

图 5-4　使用 STRAIGHT_JOIN 时的执行计划

通过图 5-4 可以看到，MySQL 优化器会强制使用左边的表进行匹配，因为左表 employees 表有大约 30 多万行的数据，因此一共要匹配 30 多万次，显然选择的这个联接并不是最有效率的。在笔者的笔记本电脑上，INNER JOIN 语句的执行时间为 0.3 秒，而 STRAIGHT_JOIN 语句的执行时间为 3.3 秒。

对两张表进行 INNER JOIN，通常 MySQL 数据库的优化器都能工作得很好。但是对于有多张表参与联接的语句，MySQL 数据库的优化器选择可能并不总是正确的。这时，对于有经验的 DBA，要确定最优的路径，可以使用 STRAIGHT_JOIN，强制优化器按照自己的联接顺序来进行联接操作。不过，随着 MySQL 数据库的不断完善，这种情况正变得越来越少。

5.2　其他联接分类

到目前为止所介绍的都是基本的联接类型。除此之外还有其他几种联接分类方式。这一节将介绍 SELF JOIN、NONEQUI JOIN 和 SEMI JOIN。

5.2.1　SELF JOIN

SELF JOIN 是同一个表的两个实例之间的 JOIN 操作。前面的最小缺失值问题使用的就是 SELF JOIN，只是没有显式地进行归类。

```
SELECT MIN(a.a+1)
FROM t x
LEFT OUTER JOIN t y
ON x.a + 1 = y.a
WHERE y.a IS NULL
```

显然是表 t 自己对自己进行 JOIN 操作。再次提醒，对同一个表进行联接操作必须指定表的别名。下面介绍另外一个常见的使用 SELF JOIN 的问题——员工—经理问题。先根据下列语句来创建表 emp。

```
CREATE TABLE emp (
emp_no int PRIMARY KEY,
mgr_no int ,
emp_name varchar(30)
);

insert into emp select 1,NULL,'David';
insert into emp select 4,1,'Jim';
insert into emp select 3,1,'Tommy';
insert into emp select 2,3,'Mariah';
insert into emp select 5,3,'Selina';
insert into emp select 6,4,'John';
insert into emp select 8,3,'Monty';
```

在表 emp 中，emp_no 代表员工编号，mgr_no 代表员工的上级经理的员工编号。员工—经理问题就是得到每位员工的经理信息。这个问题的解决方案如下：

```
SELECT a.emp_name as employee, b.emp_name as manager
FROM emp a
LEFT JOIN emp b
ON a.mgr_no = b.emp_no
```

执行上述 SQL 语句得到的结果如表 5-5 所示。

表 5-5　员工和其对应的经理信息

employee	manager	employee	manager
David	NULL	Selina	Tommy
Mariah	Tommy	John	Jim
Tommy	David	Monty	Tommy
Jim	David		

由于 David 是最高级别的员工，所以他没有经理信息，其 manager 为 NULL。如果再深入地分析员工—经理问题，会发现它其实是一个层次结构问题，表 emp 表示的员工组织图如图 5-5 所示。

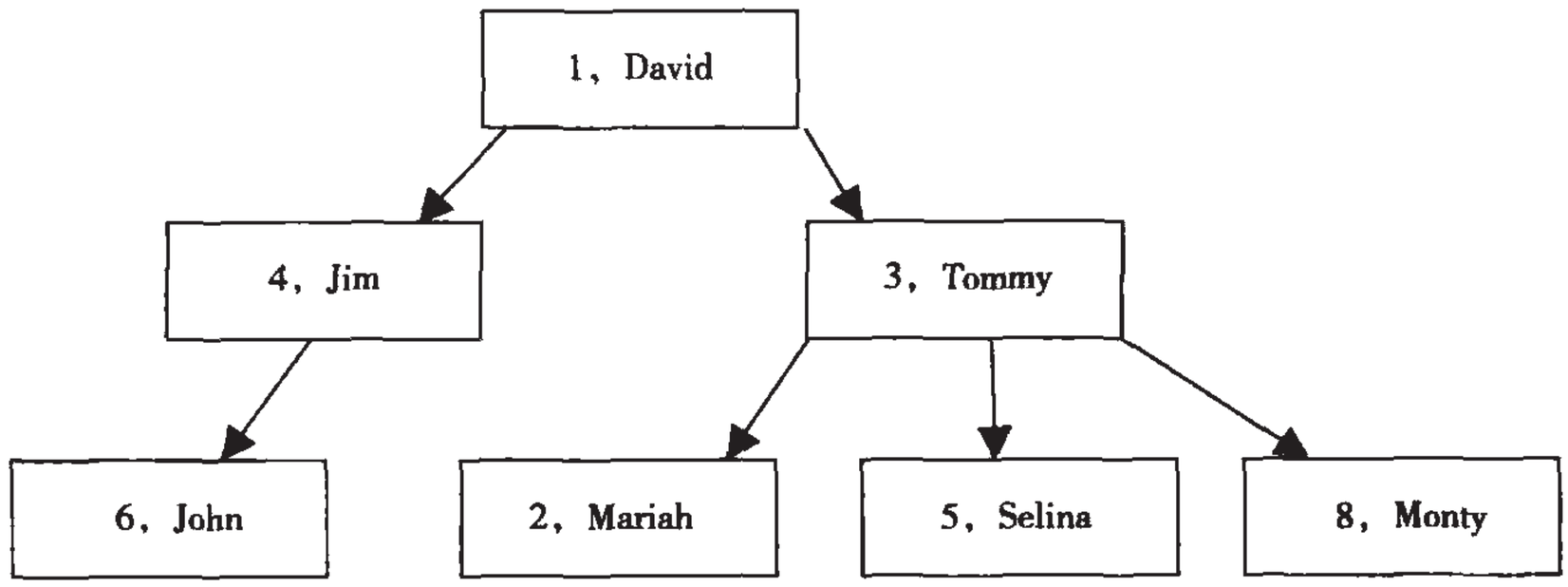


图 5-5　员工组织结构图

对于层次结构问题，通常可以通过 SELF JOIN 来解决。

5.2.2 NONEQUI JOIN

前面介绍的都是 EQUAL JOIN（等值联接），即联接条件是基于“等于”运算符的联接操作。NONEQUI JOIN 的联接条件包含“等于”运算符之外的运算符。

要生成 dept_manager 表中所有两个不同经理的组合，这里先假设当前表中仅包含员工号 A、B、C，执行 CROSS JOIN 后将生成下面九对：（A，A）、（A，B）、（A，C）、（B，A）、（B，B）、（B，C）、（C，A）、（C，B）、（C，C）。

显然，（A，A）、（B，B）和（C，C）包含相同的员工号，不是有效的员工组合。而（A，B）、（B，A）又表示同样的组合。要解决这个问题，可以指定一个左边值小于右边值的联接条件，这样可以移除上述两种情况。该问题的解决方案为：

```
SELECT a.emp_no,b.emp_no
FROM dept_manager a
INNER JOIN dept_manager b
ON a.emp_no < b.emp_no;
```

在 CROSS JOIN 小节中，已经介绍了通过 1*N 的 CROSS JOIN 产生行号。利用 NONEQUI JOIN 同样可以解决行号问题。对于表 dept_manager，可用利用下列 SQL 语句来产生行号：

```
SELECT a.dept_no,a.emp_no,count(1) as rownum
FROM dept_manager a
INNER JOIN dept_manager b
ON a.emp_no >= b.emp_no
GROUP BY a.emp_no;
```

读者可能对上面的 SQL 语句有些困惑，这里同样通过 A、B、C 来加深理解。首先通过与上个例子相反的方法来进行联接（大于等于联接），产生的结果是：(A，A)、(B，A)、(B，B)、(C，A)、(C，B)、(C，C)。

然后对左表的列进行分组，就可以得到：(A，1)、(B，2)、(C，3)。这就是我们需要的行号。可以从图 5-6 中查看该 SQL 语句的执行计划。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	a	index	PRIMARY,emp_no	emp_no	4	NULL	24	Using index; Using temporary; Using filesort
	1	SIMPLE	b	index	PRIMARY,emp_no	emp_no	4	NULL	24	Using where; Using index; Using join buffer

图 5-6　SQL 语句的执行计划

5.2.3　SEMI JOIN 和 ANTI SEMI JOIN

SEMI JOIN 是根据一个表中存在的相关记录找到另一个表中相关数据的联接。如果从左表返回记录，该联接被称为左半联接；如果从右表返回记录，该联接被称为右半联接。

实现 SEMI JOIN 的方法有多种，如内部联接、子查询、集合操作等。在使用内部联接方式时，只从一个表中选择记录，然后应用 DISTINCT。下面的 SQL 查询返回的是来自杭州且发生过订单的客户信息。

```
SELECT DISTINCT c.customer_id,city
FROM customers AS c
JOIN orders AS o
ON c.customer_id=o.customer_id
WHERE c.city='HangZhou'
```

上述 SEMI JOIN 的查询计划如图 5-7 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	o	ALL	NULL	NULL	NULL	NULL	1	Using temporary
	1	SIMPLE	c	eq_ref	PRIMARY	PRIMARY	32	test.o.customer_id	1	Using where

图 5-7　SEMI JOIN 的执行计划

最后得到的结果如表 5-6 所示。

表 5-6　SEMI JOIN 得到的结果

customer_id	city
163	HangZhou
TX	HangZhou

与 SEMI JOIN 相反的是 ANTI SEMI JOIN，它根据一个表中不存在的记录而从另一个表中返回记录。使用 OUTER JOIN 并过滤外部行，可以实现 ANTI SEMI JOIN。例如，下面的

SQL 查询返回的是来自杭州但没有订单的客户信息。

```
SELECT c.customer_id,c.city
FROM customers AS c
LEFT OUTER
JOIN orders AS o ON c.customer_id=o.customer_id
WHERE c.city='HangZhou' AND o.customer_id IS NULL;
```

上述 ANTI SEMI JOIN 查询的执行计划如图 5-8 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	c	ALL	NULL	NULL	NULL	NULL	4	Using where
	1	SIMPLE	o	ALL	NULL	NULL	NULL	NULL	1	Using where

图 5-8　ANTI SEMI JOIN 的执行计划

可以看到所有来自杭州的订单都被访问到。如果来自杭州的客户数量为 a，每个消费者的平均订单数为 b，则需要访问的订单数是 $a*b$。然后过滤外部行。最后得到的结果如表 5-7 所示。

表 5-7　执行 ANTI SEMI JOIN 得到的结果

customer_id	city
Baidu	HangZhou

5.3　多表联接

前面给出的例子都是关于两表之间的关联。多表联接是查询涉及三张或者更多张表之间的联接查询操作。

对于 INNER JOIN 的多表联接查询，可以随意安排表的顺序，而不会影响查询的结果。这是因为优化器会自动根据成本评估出访问表的顺序。在该查询的执行计划中，可能会发现优化器访问表的顺序不同于在查询中指定的顺序。

例如，下面的查询返回经理级别员工的编号、姓名、职称，以及所在部门的信息。

```
SELECT a.emp_no,c.first_name,b.title,d.dept_name
FROM dept_manager a
JOIN titles b ON a.emp_no = b.emp_no
JOIN employees c ON c.emp_no = a.emp_no
JOIN departments d ON D.dept_no = a.dept_no
```

这句 SQL 查询的执行计划如图 5-9 所示，从图中可以发现 SQL 优化器并未按照表在查询中的逻辑顺序进行联接。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	d	index	PRIMARY	dept_name	122	NULL	9	Using index
1	SIMPLE	a	ref	PRIMARY,emp_no,dept_no	dept_no	12	employees.d.dept_no	1	Using index
1	SIMPLE	c	eq_ref	PRIMARY	PRIMARY	4	employees.a.emp_no	1	
1	SIMPLE	b	ref	PRIMARY,emp_no	PRIMARY	4	employees.a.emp_no	1	Using index

图 5-9　多表联接的执行计划

可以看到优化器是按照表 d、a、c、b 的顺序来对表进行联接的。如果认为不按优化器所选择的顺序联接表会更加高效，可以通过前面介绍的 STRAIGHT_JOIN 来强制联接处理的顺序，例如：

```
SELECT a.emp_no,c.first_name,b.title,d.dept_name
FROM dept_manager a
STRAIGHT_JOIN titles b
ON a.emp_no = b.emp_no
STRAIGHT_JOIN employees c
ON c.emp_no = a.emp_no
STRAIGHT_JOIN departments d
ON D.dept_no = a.dept_no
```

强制使用 STRAIGHT_JOIN 进行多表联接的执行计划如图 5-10 所示，这时就会发现 SQL 优化器按照指定的顺序进行联接操作：

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	a	index	PRIMARY,emp_no,dept_no	emp_no	4	NULL	24	Using index
1	SIMPLE	b	ref	PRIMARY,emp_no	PRIMARY	4	employees.a.emp_no	1	Using index
1	SIMPLE	c	eq_ref	PRIMARY	PRIMARY	4	employees.b.emp_no	1	Using where
1	SIMPLE	d	eq_ref	PRIMARY	PRIMARY	12	employees.a.dept_no	1	

图 5-10　强制使用 STRAIGHT_JOIN 进行多表联接的执行计划

对于多表之间的 INNER JOIN 语句，也可以将进行 INNER JOIN 的表和 ON 过滤条件放在一起，例如：

```
SELECT a.emp_no,c.first_name,b.title,d.dept_name
FROM dept_manager a
JOIN (titles b, employees c,departments d)
ON
(
a.emp_no = b.emp_no AND
c.emp_no = a.emp_no AND
d.dept_no = a.dept_no
);
```

对括号中的表（b、c 和 d）进行的是 CROSS JOIN，然后根据后面的 ON 过滤条件将联接转换为 INNER JOIN，因此上述 SQL 又等同于：

```
SELECT a.emp_no,c.first_name,b.title,d.dept_name
FROM dept_manager a
JOIN (titles b CROSS JOIN employees CROSS JOIN departments d)
ON
(
a.emp_no = b.emp_no AND
c.emp_no = a.emp_no AND
d.dept_no = a.dept_no
);
```

从语法上来看，多表之间的 OUTER JOIN 和 INNER JOIN 操作并没有什么不同。然而，和 INNER JOIN 不同的是，利用 OUTER JOIN 进行多表联接的表之间的顺序关系可能会影响最后产生的结果。

5.4 滑动订单问题

在介绍了各种联接类型，以及相应的联接查询后，我们来看一个经典的问题——滑动订单问题，这个问题要涉及上述提及的各个联接问题。先根据以下代码来创建表 MonthlyOrders 并导入一定的数据。

```
CREATE TABLE MonthlyOrders(
ordermonth DATE,
ordernum INT UNSIGNED,
PRIMARY KEY (ordermonth)
);

INSERT INTO MonthlyOrders SELECT '2010-02-01',23;
INSERT INTO MonthlyOrders SELECT '2010-03-01',26;
INSERT INTO MonthlyOrders SELECT '2010-04-01',24;
INSERT INTO MonthlyOrders SELECT '2010-05-01',27;
INSERT INTO MonthlyOrders SELECT '2010-06-01',26;
INSERT INTO MonthlyOrders SELECT '2010-07-01',32;
INSERT INTO MonthlyOrders SELECT '2010-08-01',34;
INSERT INTO MonthlyOrders SELECT '2010-09-01',30;
INSERT INTO MonthlyOrders SELECT '2010-10-01',31;
INSERT INTO MonthlyOrders SELECT '2010-11-01',32;
INSERT INTO MonthlyOrders SELECT '2010-12-01',33;
INSERT INTO MonthlyOrders SELECT '2011-01-01',31;
INSERT INTO MonthlyOrders SELECT '2011-02-01',34;
INSERT INTO MonthlyOrders SELECT '2011-03-01',34;
INSERT INTO MonthlyOrders SELECT '2011-04-01',38;
INSERT INTO MonthlyOrders SELECT '2011-05-01',39;
INSERT INTO MonthlyOrders SELECT '2011-06-01',35;
INSERT INTO MonthlyOrders SELECT '2011-07-01',49;
INSERT INTO MonthlyOrders SELECT '2011-08-01',56;
INSERT INTO MonthlyOrders SELECT '2011-09-01',55;
```

```
INSERT INTO MonthlyOrders SELECT '2011-10-01',74;
INSERT INTO MonthlyOrders SELECT '2011-11-01',75;
INSERT INTO MonthlyOrders SELECT '2011-12-01',14;
```

滑动订单问题是指为每个月返回上一年度（季度或月度等）的滑动订单数，即为每个月份 *N*，返回从月份 *N*−11 到月份 *N* 的订单总数。这里，假设月份序列中不存在间断。

执行下面的 SQL 查询实现每个月返回上一年度的滑动订单总数，生成的结果如表 5-8 所示。

```
SELECT
   DATE_FORMAT(a.ordermonth,'%Y%m') AS frommonth,
   DATE_FORMAT(b.ordermonth,'%Y%m') AS tomonth,
   SUM(c.ordernum) AS orders
FROM monthlyorders a
INNER JOIN monthlyorders b
   ON DATE_ADD(a.ordermonth, INTERVAL 11 MONTH) = b.ordermonth
INNER JOIN monthlyorders c
   ON c.ordermonth BETWEEN a.ordermonth AND b.ordermonth
GROUP BY a.ordermonth,b.ordermonth;
```

表 5-8　上一年度的滑动订单总数

frommonth	tomonth	orders
201002	201101	349
201003	201102	360
201004	201103	368
201005	201104	382
201006	201105	394
201007	201106	403
201008	201107	420
201009	201108	442
201010	201109	467
201011	201110	510
201012	201111	553
201101	201112	534

该查询先对 MonthlyOrders 表进行自联接。a 表用做下边界（frommonth），b 表用做上边界（tomonth）。联接的条件为：`DATE_ADD(a.ordermonth, INTERVAL 11 MONTH) = b.ordermonth`。例如，a 表中的 2010 年 2 月将匹配 2011 年 1 月。完成自联接之后，需要对订单进行统计。这时需要再进行一次自联接，得到范围内每个月的订单数量。因此联接的条件为：`c.ordermonth BETWEEN a.ordermonth AND b.ordermonth`。基于上述方法，我们还可以统计每个季度订单的情况，以此作为环比和同比增长的比较依据。

```
SELECT
   DATE_FORMAT(a.ordermonth,'%Y%m') AS frommonth,
```

```
    DATE_FORMAT(b.ordermonth,'%Y%m') AS tomonth,
    SUM(c.ordernum) AS orders
FROM monthlyorders a
INNER JOIN monthlyorders b
    ON DATE_ADD(a.ordermonth, INTERVAL 2 MONTH) = b.ordermonth
    AND MONTH(a.ordermonth) % 3 = 1
INNER JOIN monthlyorders c
ON c.ordermonth BETWEEN a.ordermonth AND b.ordermonth
GROUP BY a.ordermonth,b.ordermonth;
```

上述 SQL 语句得到的结果如表 5-9 所示。

表 5-9　每个季度的订单数

frommonth	tomonth	orders
201004	201006	77
201007	201009	96
201010	201012	96
201101	201103	99
201104	201106	112
201107	201109	160
201110	201112	163

5.5　联接算法

联接算法是 MySQL 数据库用于处理联接的物理策略。目前 MySQL 数据库仅支持 Nested-Loops Join 算法。而 MySQL 的分支版本 MariaDB 除了支持 Nested-Loops Join 算法外，还支持 Classic Hash Join 算法。相信不久的将来，MySQL 数据库很快也会支持 Hash Join。因此，在这一节中，我们将介绍最为通用的 Nested-Loops Join 算法，当然也会介绍 Hash Join 算法，介绍 Hash Join 算法的目的是为将来做好准备。对于已经开始使用 MariaDB 生产环境的用户，更需要了解 Hash Join 算法及其使用的场合。

当联接的表上有索引时，Nested-Loops Join 是非常高效的算法。根据 B+ 树的特性，其联接的时间复杂度为 $O(N)$，若没有索引，则可视为最坏的情况，时间复杂度为 $O(N^2)$。MySQL 数据库根据不同的使用场合，支持两种 Nested-Loops Join 算法，一种是 Simple Nested-Loops Join（NLJ）算法，另一种是 Block Nested-Loops Join（BNL）算法。

5.5.1　Simple Nested-Loops Join 算法

Simple Nested-Loops Join 从第一张表中每次读取一条记录，然后将记录与嵌套表中的记录进行比较。其算法如下：

```
For each row r in R do
   For each row s in S do
         If r and s satisfy the join condition
               Then output the tuple <r,s>
```

其中 R 表为外部表（Outer Table），S 表为内部表（Inner Table）。这是一个最简单的算法，并且假设在两张表 R 和 S 上进行联接的列都不含有索引。这个算法的扫描次数为：$R_n+R_n*S_n$，扫描成本为 $O(R_n*S_n)$。R_n 和 S_n 分别代表 R 表和 S 表中分别含有的记录数。

假设 t1、t2、t3 三张表执行 INNER JOIN 查询，并且每张表使用的联接类型如表 5-10 所示。

表 5-10　每张表的联接类型

表	JOIN 类型	表	JOIN 类型
t1	rang	t3	ALL
t2	ref		

如果使用了 Simple Nested-Loops Join 的算法，则算法实现的伪代码如下：

```
for each row in t1 matching range {
   for each row in t2 matching reference key {
         for each row in t3 {
               if row satisfies join conditions,
               send to client
         }
   }
}
```

但是当内部表对所联接的列含有索引时，Simple Nested-Loops Join 算法可以利用索引的特性来进行快速匹配，此时算法调整如下所示：

```
For each row r in R do
   lookup r in S index
      if found s == r
            Then output the tuple <r,s>
```

对于联接的列含有索引的情况，外部表的每条记录不再需要扫描整张内部表，只需要扫描内部表上的索引即可得到联接的判断结果。如果内部表联接列的索引高度为 S_{BH}，那么上述算法的扫描次数为 $R_n+S_{BH}*R_n$，扫描成本为 $O(R_n)$。而一般 B+ 树的高度为 3 ～ 4 层，因此在有索引的情况下，Simple Nested-Loops Join 算法的执行速度是比较快的。

在 INNER JOIN 中，两张联接表的顺序是可以变换的，即 R INNER JOIN S ON Condition P 等效于 S INNER JOIN R ON Condition P。根据前面描述的 Simple Nested-Loops Join 算法，优化器在一般情况下总是选择将联接列含有索引的表作为内部表。如果两张表 R 和 S 在联接的列上都有索引，并且索引的高度相同，那么优化器会选择将记录数最少的表作为外部表，这是因为内部表的扫描次数总是索引的高度，与记录的数量无关。

接着来看一个例子，下面的 SQL 查询返回员工进入公司后的工资情况。

```
SELECT b.emp_no,a.title,a.from_date,a.to_date
FROM titles a
INNER JOIN employees b
on a.emp_no = b.emp_no;
```

emp_no 分别是表 titles 和 employees 上的主键，通过图 5-11 来查看该语句实际执行的物理计划。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	b	index	PRIMARY	PRIMARY	4	NULL	300363	Using index
	1	SIMPLE	a	ref	PRIMARY,emp_no	PRIMARY	4	employees.b.emp_no	1	

图 5-11　INNER JOIN 的执行计划

可以看到 SQL 执行计划是先查询表 employees，然后将表 titles 上的主键索引和表 employees 上的列 emp_no 进行匹配。

这里为什么首先使用 employees 表呢？因为表 employees 中的记录少于表 titles，这样联接需要匹配的次数就少了，所以 SQL 优化器选择表 employees 作为外部表，具体情况如下：

```
mysql> SELECT COUNT(1) FROM employees\G;
*************************** 1. row ***************************
count(1): 300024
1 row in set (0.36 sec)

mysql> SELECT COUNT(1) FROM titles\G;
*************************** 1. row ***************************
count(1): 443308
1 row in set (0.46 sec)
```

若我们执行的 SQL 语句包含多张表的联接，并且含有 WHERE 过滤条件，如：

```
SELECT * FROM T1
   INNER JOIN T2 ON P1(T1,T2)
   INNER JOIN T3 ON P2(T2,T3)
WHERE P(T1,T2,T3)
```

则上述 Simple Nested-Loops Join 算法的伪代码可能为：

```
FOR each row t1 in T1 {
   FOR each row t2 in T2 such that P1(t1,t2) {
          FOR each row t3 in T3 such that P2(t2,t3) {
                 IF P(t1,t2,t3) {
                        t:=t1||t2||t3; OUTPUT t;
                 }
          }
   }
}
```

P1（t1，t2）、P2（t2，t3）表示两张表进行联接操作的条件，t1||t2||t3 表示从表 t1、t2 和 t3 中选择需要的数据。前面已经说过，优化器会根据表中的记录来选择 INNER JOIN 的顺序，故针对上述三表的联接操作的算法也可能为：

```
FOR each row t3 in T3 {
   FOR each row t2 in T2 such that P2(t2,t3) {
          FOR each row t1 in T1 such that P1(t1,t2) {
                IF P(t1,t2,t3) {
                      t:=t1||t2||t3; OUTPUT t;
                }
          }
   }
}
```

但是这里我们忽略了一个非常重要的部分——pushed-down conditions 的优化。对于内部表的过滤条件 P（t1，t2，t3），SQL 优化器会将判断放到外部表中去，因为我们知道：

```
P(t1,t2,t3)= C1(t1)& C2(t2)&C3(t3)
```

因此针对上述三表的联接操作的算法可调整为：

```
FOR each row t1 in T1 such that C1(t1) {
   FOR each row t2 in T2 such that P1(t1,t2) AND C2(t2) {
          FOR each row t3 in T3 such that P2(t2,t3) AND C3(t3) {
                IF P(t1,t2,t3) {
                      t:=t1||t2||t3; OUTPUT t;
                }
          }
   }
}
```

如果 C1（t1）的条件能在外部表中过滤掉相当多的一部分数据，那么毫无疑问，内部表的查询次数就会相应地减少很多，联接的执行效率也会因此提高很多。

前面已经提到，进行 OUTER JOIN 操作的表的顺序是不能变的，因为可能引起最后数据的变化，因此对于如下进行 OUTER JOIN 操作的 SQL 语句：

```
SELECT * FROM T1 LEFT JOIN
   (T2 LEFT JOIN T3 ON P2(T2,T3))
   ON P1(T1,T2)
WHERE P(T1,T2,T3)
```

其联接算法的伪代码为：

```
FOR each row t1 in T1 {
   BOOL f1:=FALSE;
   FOR each row t2 in T2 such that P1(t1,t2) {
          BOOL f2:=FALSE;
          FOR each row t3 in T3 such that P2(t2,t3) {
```

```
                IF P(t1,t2,t3) {
                        t:=t1||t2||t3; OUTPUT t;
                }
                f2=TRUE;
                f1=TRUE;
            }
            IF (!f2) {
                IF P(t1,t2,NULL) {
                        t:=t1||t2||NULL; OUTPUT t;
                }
                f1=TRUE;
            }
        }
        IF (!f1) {
            IF P(t1,NULL,NULL) {
                    t:=t1||NULL||NULL; OUTPUT t;
            }
        }
    }
```

前面介绍了在 INNER JOIN 中可以使用 pushed-down conditions 的优化方式，但是不能直接在 OUTER JOIN 中使用该方式，因为有些不满足联接条件的记录会通过外部表行的方式再次添加到结果中，因此需要有条件地使用 pushed-down conditions 的优化。这里需要设置一个标志来表示是否启用 pushed-down conditions 的过滤，因此算法变为：

```
FOR each row t1 in T1 such that C1(t1) {
   BOOL f1:=FALSE;
   FOR each row t2 in T2
   such that P1(t1,t2) AND (f1?C2(t2):TRUE) {
           BOOL f2:=FALSE;
           FOR each row t3 in T3
           such that P2(t2,t3) AND (f1&&f2?C3(t3):TRUE) {
                   IF (f1&&f2?TRUE:(C2(t2) AND C3(t3))) {
                           t:=t1||t2||t3; OUTPUT t;
                   }
                   f2=TRUE;
                   f1=TRUE;
           }
           IF (!f2) {
                   IF (f1?TRUE:C2(t2) && P(t1,t2,NULL)) {
                           t:=t1||t2||NULL; OUTPUT t;
                   }
                   f1=TRUE;
           }
   }
   IF (!f1 && P(t1,NULL,NULL)) {
           t:=t1||NULL||NULL; OUTPUT t;
   }
}
```

5.5.2　Block Nested-Loops Join 算法

Simple Nested-Loops Join 算法在内层循环时，外部表的每行记录需要读取内部表一次。在内部表的联接上有索引的情况下，其扫描成本为 $O(R_n)$；若没有索引，则扫描成本为 $O(R_n*S_n)$。如果内部表 S 有相当多的记录，则 Simple Nested-Loops Join 会扫描内部表很多次，执行效率会非常差。而 Block Nested-Loops Join 算法就是针对没有索引的联接情况设计的，其使用 Join Buffer（联接缓冲）来减少内部循环读取表的次数。

例如，Block Nested-Loops Join 算法先把对 Outer Loop 表（外部表）每次读取的 10 行记录（准确地说是 10 行需要进行联接的列）放入 Join Buffer 中，然后在 Inner Loop 表（内部表）中直接匹配这 10 行数据。因此，对 Inner Loop 表的扫描减少了 1/10。对于没有索引的表来说，Block Nested-Loops Join 算法可以极大地提高联接的速度。

MySQL 数据库使用 Join Buffer 的原则如下：

- 系统变量 join_buffer_size 决定了 Join Buffer 的大小。
- Join Buffer 可被用于联接是 ALL、index 和 range 的类型。
- 每次联接使用一个 Join Buffer，因此多表的联接可以使用多个 Join Buffer。
- Join Buffer 在联接发生之前进行分配，在 SQL 语句执行完后进行释放。
- Join Buffer 只存储需要进行查询操作的相关列数据，而不是整行的记录。

对于上一小节提到的三个表进行联接操作，如果使用 Join Buffer，则算法的伪代码如下：

```
for each row in t1 matching range {
   for each row in t2 matching reference key {
   store used columns from t1, t2 in join buffer
   if buffer is full {
          for each row in t3 {
                 for each t1, t2 combination in join buffer {
                        if row satisfies join conditions,
                        send to client
                 }
          }
          empty buffer
   }
   }
}
if buffer is not empty {
   for each row in t3 {
          for each t1, t2 combination in join buffer {
                 if row satisfies join conditions,
                 send to client
          }
   }
}
```

接着我们来举一个例子，这次我们同样对之前的表 employees 和表 titles 进行联接，不同的是表 employees 和表 titles 的列 emp_no 上没有索引。先按下面的代码生成没有索引的表 employees_noindex 和表 titles_noindex，并导入同样数据量的数据。

```
CREATE TABLE `employees_noindex` (
`emp_no` int(11) NOT NULL,
`birth_date` date NOT NULL,
`first_name` varchar(14) NOT NULL,
`last_name` varchar(16) NOT NULL,
`gender` enum('M','F') NOT NULL,
`hire_date` date NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `titles_noindex` (
  `emp_no` int(11) NOT NULL,
  `title` varchar(50) NOT NULL,
  `from_date` date NOT NULL,
  `to_date` date DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO employees_noindex
SELECT *
FROM employees;

INSERT INTO titles_noindex
SELECT *
FROM titles;
```

接着执行下述 SQL 语句：

```
EXPLAIN SELECT b.emp_no,a.title,a.from_date,a.to_date
FROM employees_noindex b
INNER JOIN titles_noindex a ON a.emp_no = b.emp_no
WHERE b.birth_date >= '1965-01-01';
```

再查看这次的 SQL 执行计划，如图 5-12 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	b	ALL	NULL	NULL	NULL	NULL	300629	Using where
	1	SIMPLE	a	ALL	NULL	NULL	NULL	NULL	443463	Using where; Using join buffer

图 5-12　Block Nested-Loops Join 的执行计划

可以看到，SQL 执行计划的 Extra 列中提示 Using join buffer，这就代表使用了 Block Nested-Loops Join 算法。MySQL 5.6 会在 Extra 列显示更为详细的信息，如图 5-13 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	b	ALL	NULL	NULL	NULL	NULL	300228	Using where
	1	SIMPLE	a	ALL	NULL	NULL	NULL	NULL	443381	Using where; Using join buffer (Block Nested Loop)

图 5-13　Block Nested-Loops Join 在 MySQL 5.6 中的执行计划

这里要注意的是在 MySQL 5.5 版本中，Join Buffer 只能在 INNER JOIN 中使用，在 OUTER JOIN 中则不能使用，即 Block Nested-Loops Join 算法不支持 OUTER JOIN。对于如下 SQL 语句：

```
SELECT b.emp_no,a.title,a.from_date,a.to_date
FROM titles_noindex a
LEFT OUTER JOIN employees_noindex b ON a.emp_no = b.emp_no
WHERE b.birth_date >= '1965-01-01';
```

LEFT OUTER JOIN 在 MySQL5.5 中的执行计划如图 5-14 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	b	ALL	NULL	NULL	NULL	NULL	300629	Using where
	1	SIMPLE	a	ALL	NULL	NULL	NULL	NULL	443463	

图 5-14　LEFT OUTER JOIN 的执行计划

可以看到这次执行计划的 Extra 列中并没有 Using join buffer 的提示，这也就意味着此时优化器没有使用 Block Nested-Loops Join 算法。从 MySQL 5.6 及 MariaDB 5.3 开始，Join Buffer 的使用得到了进一步扩展，在 OUTER JOIN 中使用 Join Buffer 受到支持，因此上述 SQL 语句在 MySQL 5.6.3 中的执行计划如图 5-15 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	b	ALL	NULL	NULL	NULL	NULL	300504	Using where
	1	SIMPLE	a	ALL	NULL	NULL	NULL	NULL	444828	Using where; Using join buffer (Block Nested Loop)

图 5-15　MySQL 5.6 对于 OUTER JOIN 的执行计划

在 MySQL 5.6 中，上述 SQL 语句在笔者的台式机上运行时间为 57.483 秒，而在 MySQL 5.5 中总共需要 709.645 秒。可以看到 Join Buffer 为 OUTER JOIN 带来了性能提升。

在 MySQL 5.6（包括 MariaDB 5.3）中，优化了 Join Buffer 在多张表之间联接的内存使用效率。MySQL 5.6 将 Join Buffer 分为 Regular join buffer 和 Incremental join buffer。假设 B1 是表 t1 和 t2 联接使用的 Join Buffer，B2 是 t1 和 t2 联接产生的结果和表 t3 进行联接使用的 Join Buffer，那么：

- 如果 B2 是 Regular join buffer，那么 B2 就会包含 B1 的 Join Buffer 中 r1 相关的列，以及表 t2 中相关的列。
- 如果 B2 是 Incremental join buffer，那么 B2 包含表 t2 中的数据及一个指针，该指针指向 B1 中 r1 相对应的数据。

因此，对于第一次联接的表，使用的都是 Regular join buffer，之后再联接，则使用 Incremental join buffer。又因为 Incremental join buffer 只包含指向之前 Join Buffer 中数据的指针，所以 Join Buffer 的内存使用效率得到了大幅的提高。

5.5.3 Batched Key Access Join 算法

到目前为止，一共介绍了两种联接算法，这里来回顾一下。执行 JOIN 的过程如图 5-16 所示。要加快 JOIN 的执行速度，有如下两种方式：

- 加快每次 search-for-match（查询比较）操作的速度。
- search-for-match 根据 group（组）来进行，减少对内部表的访问次数。

对于第一种方式，用户可以通过添加索引，让优化器选择索引来进行 search-for-match 的操作。第二种方式采用的就是 Block Nested-Loops Join 算法的思想，使用额外的一小部分 Join Buffer 内存将外部表的数据放入 Join Buffer，然后在内部表中根据 group 来进行 search-for-match 的操作，以此来减少内部表的访问次数。

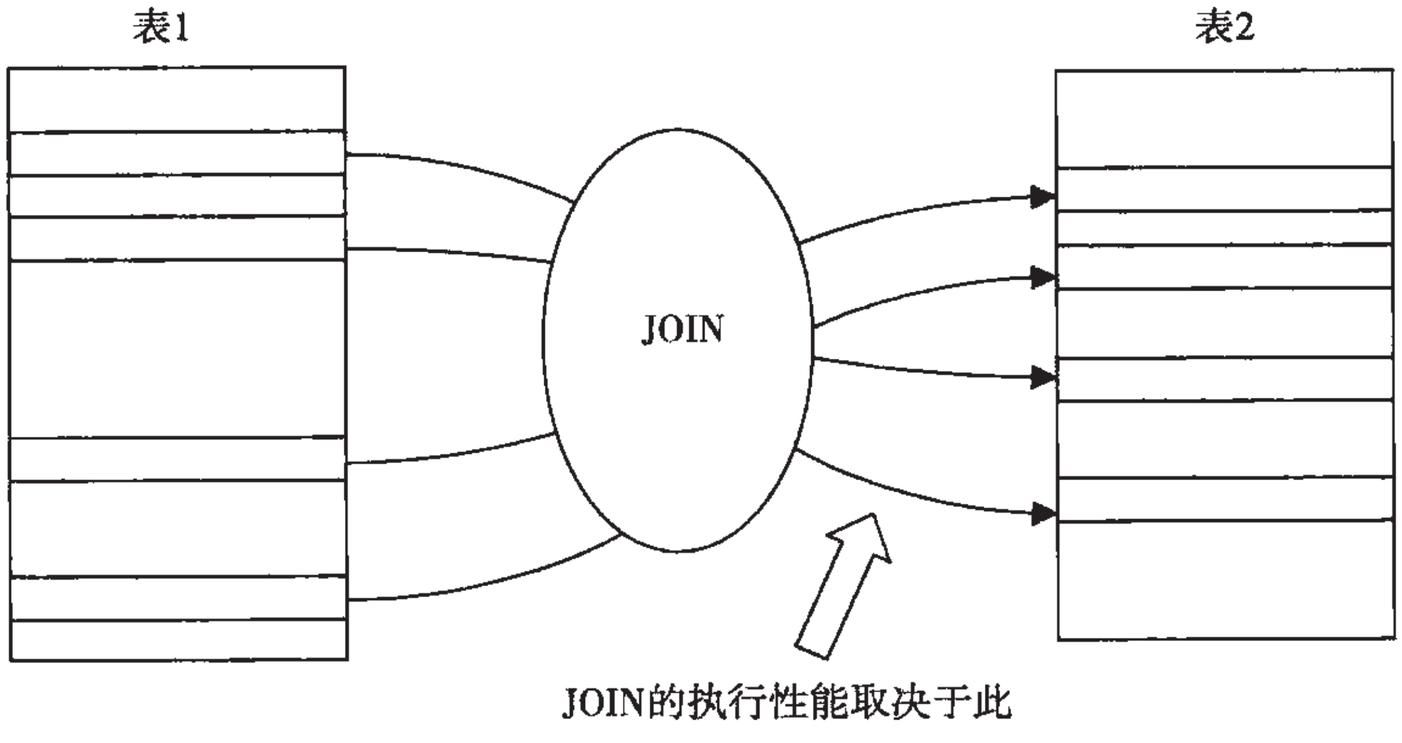


图 5-16 JOIN 的执行过程

MySQL 5.6（MariaDB 5.3）开始支持 Batched Key Access Join 算法（简称 BKA），该算法的思想为结合索引和 group 这两种方法（Simple Nested-Loops Join 和 Block Nested-Loops Join 只能使用一种）来提高 search-for-match 的操作，以此加快联接的执行效率。我们可以把这种算法理解为 group-index-lookup（组索引查询）。为什么使用 group-index-lookup 能提高 search-for-match 的操作呢？因为这样可以提高数据库整体资源的 Cache 命中率，从 CPU Cache、Page Cache 到 Disk 的访问，效率都变得更高。如果外部表扫描的是主键，那么表中的记录访问都是比较有序的，但是如果联接的列是非主键索引，那么对于表中记录的访问可能就是非常离散的。因此对于非主键索引的联接，Batched Key Access Join 算法将能极大提高 SQL 的执行效率。

Batched Key Access Join 算法的工作步骤如下：

1）将外部表中相关的列放入 Join Buffer 中。

2）批量地将 Key（索引键值）发送到 Multi-Range Read（MRR）接口。

3）Multi-Range Read（MRR）通过收到的 Key，根据其对应的 ROWID 进行排序，然后再进行数据的读取操作。

4）返回结果集给客户端。

Multi-Range Read 将会在介绍索引时详细介绍，这里只需要知道 Batched Key Access Join 算法使用该接口来提高获取记录的效率。现在，我们来看一个例子，其中使用的 SQL 语句如下：

```
SELECT MAX(l_extendedprice) FROM orders, lineitem
WHERE o_orderdate BETWEEN '1995-01-01' AND '1995-01-31' AND
l_orderkey=o_orderkey;
```

在上述 SQL 语句中，l_orderkey、o_orderdate 和 o_orderkey 上都有索引。l_extendedprice 是表 lineitem 中的列，不在索引 l_orderkey 中。因此当通过 l_orderkey 索引来进行 search-for-match 操作之后，还需要根据 ROWID 来查找 l_extendedprice 的值，而这个查找是离散的，故使用 Batched Key Access Join 算法能提高 SQL 语句执行的效率。

该语句在 MySQL 5.5 下的执行计划如图 5-17 所示。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	orders	range	PRIMARY,i_o_orderdate	i_o_orderdate	4	NULL	42008	Using where; Using index
1	SIMPLE	lineitem	ref	PRIMARY,i_l_orderkey,i_l_orderkey_quantity	PRIMARY	4	dbt3.orders.o_orderkey	1	

图 5-17　在 MySQL 5.5 下的执行计划

非常有意思的是，在 MySQL 5.5 下，数据库内部表的 search-for-match 操作选择了 PRIMARY 这个索引，也就是主键索引，而没有使用已经存在于列 l_orderkey 上的 i_l_orderkey 索引。产生这个结果的原因还是优化器认为利用辅助索引得到联接判断后，还要再次读取主键上的列 l_ extendedprice，这是离散读取，因此要远远慢于直接的主键访问，并且主键为（l_orderkey，l_linenumber）的联合索引。

当然，用户可以通过索引提示强制优化器使用 i_l_orderkey 索引，相应的 SQL 语句为：

```
SELECT MAX(l_extendedprice)
FROM orders, lineitem force index (i_l_orderkey)
WHERE o_orderdate BETWEEN '1995-01-01' AND '1995-01-31' AND
l_orderkey=o_orderkey;
```

这时 SQL 语句的执行计划如图 5-18 所示。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	orders	range	PRIMARY,i_o_orderdate	i_o_orderdate	4	NULL	42008	Using where; Using index
1	SIMPLE	lineitem	ref	i_l_orderkey	i_l_orderkey	4	dbt3.orders.o_orderkey	1	

图 5-18　强制使用索引 i_l_orderkey 的执行计划

在 MySQL 5.6 下，可以使用下列语句启用 Batched Key Access Join 算法。

```
SET optimizer_switch
        ='mrr=on,mrr_cost_based=off,batched_key_access=on';
```

这时 SQL 语句的执行计划如图 5-19 所示。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	orders	range	PRIMARY,i_o_orderdate	i_o_orderdate	4	NULL	42008	Using where; Using index
1	SIMPLE	lineitem	ref	PRIMARY,i_l_orderkey,i_l_orderkey_quantity	i_l_orderkey	4	dbt3.orders.o_orderkey	1	Using join buffer (Batched Key Access)

图 5-19　启用 Batched Key Access Join 算法后的执行计划

可以看到，在 Extra 列中不但显示了 Using join buffer，还有 Batched Key Access 的提示，这就代表优化器选择了 Batched Key Access Join 算法。表 5-11 列出了上述 SQL 语句在三种环境下的执行时间。

表 5-11　SQL 语句在三种环境下的执行时间

环境	执行时间（秒）
MySQL 5.5	125.3
MySQL 5.5（强制使用 i_l_orderkey 索引）	153.021
MySQL 5.6（Batched Key Access）	49.577

可以看到，在 MySQL 5.5 下执行时间为 125.3 秒，而在 MySQL 5.6 中启用 Batched Key Access 后，执行时间缩短为 49.577 秒，执行效率提高了 1.5 倍之多。而强制优化器使用 i_l_orderkey 索引并没有带来性能上的提升，反而执行效率最慢，需要 153.021 秒。其中的主要原因前面已经分析过，通过 i_l_orderkey 索引判断联接条件后还需要从主键中取得数据，这是离散读的操作，不如直接通过主键顺序地读取数据的效率高。

因为 Batched Key Access Join 算法的本质是通过 Multi-Range Read 接口将非主键索引对于记录的访问，转化为根据 ROWID 排序的较为有序的记录获取，所以要想通过 Batched Key Access Join 算法来提高性能，不但需要确保联接的列参与 match 的操作，还要有对非主键列的 search 操作。例如下列 SQL 语句：

```
SELECT
o_orderkey,o_custkey,l_extendedprice,l_discount
FROM orders, lineitem WHERE
o_orderdate BETWEEN '1995-01-01' AND '1995-01-31' AND
l_orderkey=o_orderkey;
```

列 l_extendedprice 和 l_discount 是表 lineitem 的数据，但不是通过搜索 l_orderkey 索引就能得到数据，还需要访问主键来获取数据。因此这时可以使用 Batched Key Access，其执行计划如图 5-20 所示。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	orders	range	PRIMARY,i_o_orderdate	i_o_orderdate	4	NULL	42008	Using index condition; Using MRR
1	SIMPLE	lineitem	ref	PRIMARY,i_l_orderkey,i_l_orderkey_quantity	i_l_orderkey	4	dbt3.orders.o_orderkey	1	Using join buffer (Batched Key Access)

图 5-20　SQL 语句的执行计划

可以看到这里的 Extra 列中有 Batched Key Access 的提示。但是如果联接不涉及针对主

键进一步获取数据，内部表只参与联接判断，那么就不会启用 Batched Key Access Join 算法，因为没有必要去调用 Multi-Range Read 接口。例如下列 SQL 语句：

```
SELECT o_orderkey,o_custkey FROM orders, lineitem WHERE
o_orderdate BETWEEN '1995-01-01' AND '1995-01-31' AND
l_orderkey=o_orderkey;
```

上述语句只返回外部表中的数据，只是对内部表做一个联接的判断，因此不会使用 Batch Key Access，其执行计划如图 5-21 所示。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	orders	range	PRIMARY,i_o_orderdate	i_o_orderdate	4		42008	Using index condition; Using MRR
1	SIMPLE	lineitem	ref	PRIMARY,i_l_orderkey,i_l_orderkey_quantity	i_l_orderkey	4	dbt3.orders.o_orderkey	1	Using index

图 5-21　SQL 语句的执行计划

总结：Batched Key Access Join 算法从本质上来说还是 Simple Nested-Loops Join 算法，其发生的条件为内部表上有索引，并且该索引为非主键的，并且联接需要访问内部表主键上的索引。这时 Batched Key Access Join 算法调用 Multi-Range Read 接口，批量地进行索引键的匹配和主键索引上获取数据的操作，以此来提高联接的执行效率。

5.5.4　Classic Hash Join 算法

Batched Key Access Join 算法虽然解决了一些问题，但是还存在一些问题。一方面，不是每个联接语句都是有索引的，对于没有索引的情况，MySQL 数据库目前只能使用 Block Nested-Loops Join 算法。这样虽然减少了内部表的扫描次数，但是没有减少执行 search-for-match 的次数。另一方面，即使有索引，但是当得到的数据占据表中大部分时，直接使用主键索引扫描更有效率，使用 Multi-Range Read 反而显得没有什么必要。

对于上述的两方面问题，可以通过另外一种联接算法来解决，即 Hash Join，是一种广泛应用于数据仓库和 OLAP 应用的经典联接算法。MySQL 数据库目前并不支持 Hash Join，但是其分支版本 MariaDB 5.3 开始支持 Classic Hash Join 算法。相信随着 MySQL 数据库的不断成熟，支持 Hash Join 算法只是一个时间问题。

Classic Hash Join 算法同样使用 Join Buffer，先将外部表中数据放入 Join Buffer 中，然后根据键值产生一张散列表，这是第一个阶段，称为 build 阶段。随后读取内部表中的一条记录，对其应用散列函数，将其和散列表中的数据进行比较，这是第二个阶段，称为 probe 阶段。

如果将 Hash 查找应用于 Simple Nested-Loops Join 中，则执行计划的 Extra 列会显示 BNLH。如果将 Hash 查找应用于 Batched Key Access Join 中，则执行计划的 Extra 列会显示 BKAH。

注意　Hash Join 只能应用于等值的联接操作中，因为已通过散列函数生成新的联接值，不能将 Hash Join 用于非等值的联接操作中。

倘若 Join Buffer 能够完全存放下外部表的数据，那么 Classic Hash Join 算法只需要扫描一次内部表。反之，Classic Hash Join 需要扫描多次内部表。为了使 Classic Hash Join 更有效率，应该更好地规划 Join Buffer 的大小。其算法的伪代码如下：

```
For each tuple r in R do
   store used columns as p from R in join buffer
   build hash table according join buffer
    for each tuple s in R do
          probe hash table
          if  find
   Then output the tuple <p,s>
```

要使用 Classic Hash Join 算法，需要将 join_cache_level 设置为大于等于 4 的值，并显式地打开优化器的选项，设置过程如下：

```
SET join_cache_level=4+;
SET optimizer_switch='join_cache_hashed=on';
```

对于上一小节中使用的 SQL 语句：

```
SELECT MAX(l_extendedprice) FROM orders, lineitem WHERE
o_orderdate BETWEEN '1995-01-01' AND '1995-01-31' AND
l_orderkey=o_orderkey;
```

这时的执行计划如图 5-22 所示。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	orders	range	PRIMARY,i_o_orderdate	i_o_orderdate	4	[illegible]	42008	Using where; Using index
1	SIMPLE	lineitem	hash_ALL	PRIMARY,i_l_orderkey,i_l_orderkey_quantity	#hash#PRIMARY	4	dbt3.orders.o_orderkey	5994679	Using join buffer (flat, BNLH join)

图 5-22　Classic Hash Join 算法的执行计划

在执行计划中可以看到，这里的 type 列显示为 hash_ALL，Extra 列提示了 Using join buffer（flat BNLH join），表示使用了 Classic Hash Join 的算法。通过 Hash Join 算法，上述 SQL 语句的执行时间进一步缩短，只需 23.104 秒，比 Batched Key Access Join 算法还要快一倍多。这里留一个思考问题，对上述语句使用 BKAH，并且比较上述语句在使用 BNLH 和 BKAH 时的执行效率。

注意　上述 SQL 语句通过 Classic Hash Join 算法只需 23.104 秒，这是因为我们将 join_buffer_size 设置为 32MB。若采用默认的 128KB 的大小，即使使用了 Classic Hash Join 算法，依然需要 127 秒。正如前面所说，Classic Hash Join 算法同样对 Join Buffer 的大小非常敏感。

表 5-12 显示了 SQL 语句在 MySQL 5.5、MySQL 5.6 和 MariaDB 5.3 下的通过各种 JOIN 算法执行的时间，并通过 InnoSQL 数据库提供的额外功能显示了通过各个 SQL 语句的逻辑

读取 IO 的次数。可以看到虽然 BNLH 读取的逻辑 IO 次数是最多的，但是其运行效率是最高的。因为其是通过顺序地扫描主键，而其他的 JOIN 算法可能都会涉及大量的离散读取。

表 5-12　SQL 语句在各种环境下的执行时间和逻辑 IO

环境	执行时间（秒）	逻辑 IO（次）
MySQL 5.5	125.3	157 061
MySQL 5.5（强制使用 i_l_orderkey 索引）	153.021	406 879
MySQL 5.6（Batched Key Access）	49.577	378 641
MariaDB 5.3（BNLH）	23.104	6 077 083
MariaDB 5.3（BAKH）	52.479	403 108

对于 OLTP 的应用来说，一般总是希望尽可能少地产生 IO 操作，这样就能加快数据库的响应时间。而对于 OLAP 的应用来说，例如前面的例子，响应时间通常不再和 IO 的次数成正比，而是需要选择正确的算法来得到更高的数据库响应时间。笔者认为，OLAP 优化所需掌握的知识及实现的难度要高于 OLTP 的应用。

正如前面所说，当 Join Buffer 不能存放下所有外部表中的数据时，Classic Hash Join 需要扫描内部表多次。对于这种情况，其他数据库中使用的是 Grace Hash Join 算法，对内部表和外部表都只需扫描一次，有兴趣的读者可以查找相关资料。MariaDB 有计划支持 Grace Hash Join 算法，相信不久的将来也能在 MySQL 数据库中看到 Grace Hash Join 算法。

5.6　集合操作

5.6.1　集合操作的概述

通常来说，将联接操作看成是表之间的水平操作，因为该操作生成的虚拟表包含两个表中的列。本小节介绍两个表之间的集合操作，一般将这些操作视为垂直操作。MySQL 数据库支持两种集合操作：UNION ALL 和 UNION DISTINCT。

与联接操作一样，集合操作也是对两个输入进行操作，并生成一个虚拟表。在联接操作中，一般把输入表称为左输入和右输入，或者第一个输入和第二个输入。集合操作的两个输入必须拥有相同的列数，若数据类型不同，MySQL 数据库会自动将进行隐式转化。同时，结果列的名称由第一个输入决定。

在进一步阐述集合操作之前，先根据下列语句创建测试表 x、y：

```
CREATE TABLE x (a CHAR(1)) ENGINE=InnoDB;
CREATE TABLE y (a CHAR(1)) ENGINE=InnoDB;
INSERT INTO x SELECT 'a';
INSERT INTO x SELECT 'b';
```

```
INSERT INTO x SELECT 'c';
INSERT INTO y SELECT 'a';
INSERT INTO y SELECT 'b';
```

接着来看对一个不同类型的数据进行集合操作：

```
mysql>SELECT a AS m FROM x
    -> UNION
    -> SELECT 1 AS n FROM DUAL
    -> UNION
    -> SELECT 'abc' AS o FROM DUAL
    -> UNION
    -> SELECT NOW() AS p FROM DUAL\G;
*************************** 1. row ***************************
m: a
*************************** 2. row ***************************
m: b
*************************** 3. row ***************************
m: c
*************************** 4. row ***************************
m: 1
*************************** 5. row ***************************
m: abc
*************************** 6. row ***************************
m: 2011-08-30 19:59:47
6 rows in set (0.00 sec)
```

这里对各种不同的类型进行了联接，先从 x 表中取出类型为 CHAR（1）的字符，第二个进行 UNION 操作的是整型数 1，第三个是字符串 abc，第四个是日期类型。虽然类型各不相同，但是 MySQL 数据库会自动对其进行判断，选出一种类型进行隐式转换。

另一方面，在这个例子中对每个选取操作都进行了别名定义，从最后的结果可以看出，MySQL 数据库选择了 m 这个别名，也就是集合操作中第一个 SELECT 输入的别名。

除了以下两点，集合操作中的 SELECT 语句和一般的 SELECT 查询并无不同：

- 只有最后一个 SELECT 可以应用 INTO OUTFILE，但是整个集合的操作将被输出到文件中。
- 不能在 SELECT 语句中使用 HIGH_PRIORITY 关键字。

注意，在集合操作中，INTO OUTFILE 只能存在于最后一个 SELECT 语句中，否则 MySQL 数据库会提示语法错误，例如：

```
mysql> SELECT a INTO OUTFILE 'ret.txt' FROM x
    -> UNION
    -> SELECT a FROM y;
ERROR 1221 (HY000): Incorrect usage of UNION and INTO
```

此外，虽然 INTO OUTFILE 只存在于最后一个 SELECT 语句中，但导出的结果是整个

集合操作的结果，例如：

```
mysql> SELECT a FROM x
    -> UNION
    -> SELECT a INTO OUTFILE 'ret.txt' FROM y;
Query OK, 3 rows affected (0.02 sec)

mysql>system cat /db/mysql_data/test/ret.txt;
a
b
c
```

还有一点需要注意的是，若 SELECT 语句中包含 LIMIT 和 ORDER BY 子句，最好的做法是为参与集合操作的各 SELECT 语句添加括号，否则执行集合查询会得到错误提示，例如：

```
# LIMIT 未添加括号
SELECT a FROM x ORDER BY a LIMIT 1
UNION
SELECT a FROM y

# 只对第一个数据添加括号
(SELECT a FROM x ORDER BY a LIMIT 1)
UNION
SELECT a FROM y
```

这个集合操作的正确 SQL 语句应该为：

```
(SELECT a FROM x LIMIT 1)
UNION
(SELECT a FROM y)
```

5.6.2　UNION DISTINCT 和 UNION ALL

UNION DISTINCT 组合两个输入，并应用 DISTINCT 过滤重复项。一般省略 DISTINCT 关键字，直接用 UNION，例如：

```
mysql>SELECT a FROM x
    -> UNION
    -> SELECT a FROM y;
+------+
| a    |
+------+
| a    |
| b    |
| c    |
+------+
3 rows in set (0.10 sec)
```

MySQL 数据库目前对 UNION DISTINCT 的实现方式如下：

- 创建一张临时表，即虚拟表。
- 对这张临时表的列添加唯一索引（Unique Index）。
- 将输入的数据插入临时表。
- 返回虚拟表。

因为添加了唯一索引，所以可以过滤掉集合中重复的项。可以通过观察服务器状态变量 Created_tmp_tables 来确认 UNION DISTINCT 的实现方式，例如：

```
mysql> SHOW STATUS LIKE 'Created_tmp_tables'\G;
*************************** 1. row ***************************
Variable_name: Created_tmp_tables
        Value: 12
1 row in set (0.00 sec)

mysql>SELECT a FROM x
    -> UNION
    -> SELECT a FROM y\G;
*************************** 1. row ***************************
a: a
*************************** 2. row ***************************
a: b
*************************** 3. row ***************************
a: c
3 rows in set (0.00 sec)

mysql> SHOW STATUS LIKE 'Created_tmp_tables'\G;
*************************** 1. row ***************************
Variable_name: Created_tmp_tables
        Value: 13
1 row in set (0.04 sec)
```

由于向临时表添加了唯一索引，插入的速度显然会因此而受到影响。如果确认进行 UNION 操作的两个集合中没有重复的选项，最有效的办法应该是使用 UNION ALL。

UNION ALL 组合两个输入中所有项的结果集，并包含重复的选项，例如：

```
mysql>SELECT * FROM x
    -> UNION ALL
    -> SELECT * FROM y;
+------+
| a    |
+------+
| a    |
| b    |
| c    |
| a    |
| b    |
+------+
5 rows in set (1.05 sec)
```

正如前面所说，如果确认两个输入中没有重复项，应该选择 UNION ALL。如果两个输入中有重复项，也可以在数据库端使用 UNION ALL，在应用程序端进行 DISTINCT 的去重操作。

5.6.3　EXCEPT

MySQL 数据库并不原生支持 EXCEPT 的语法，不过我们仍然可以通过一些手段来得到 EXCEPT 的结果。EXCEPT 集合操作允许用户找出位于第一个输入中但不位于第二个输入中的行数据。同 UNION 一样，EXCEPT 可分为 EXCEPT DISTINCT 和 EXCEPT ALL。

EXCEPT DISTINCT 返回位于第一个输入中但不位于第二个输入中的不重复行。常见的方法是使用 LEFT JOIN 或 NOT EXISTS。但是直接应用这些方法可能会有一些错误产生。下面通过一个例子来说明。先对前一个小节中表 x 和 y 应用 EXCEPT 集合操作：

```
mysql>SELECT x.a FROM x
    -> LEFT JOIN y
    -> ON x.a = y.a
    -> WHERE y.a IS NULL;
+------+
| a    |
+------+
| c    |
+------+
1 row in set (0.00 sec)

mysql>SELECT a FROM x
    -> WHERE NOT EXISTS (
    -> SELECT * FROM y where x.a = y.a
    -> );
+------+
| a    |
+------+
| c    |
+------+
1 row in set (0.00 sec)
```

上述两种方式都能返回集合的 EXCEPT 操作，看似没有问题，但是如果输入项中包含 NULL 值，情况就不这么简单了。我们对表 x 和 y 进行如下重构：

```
CREATE TABLE x ( a CHAR(1), b CHAR(1))ENGINE=InnoDB;
CREATE TABLE y ( a CHAR(1), b CHAR(1))ENGINE=InnoDB;

INSERT INTO x SELECT 'a','b';
INSERT INTO x SELECT 'b',NULL;
INSERT INTO x SELECT 'c','d';
INSERT INTO x SELECT 'c','d';
INSERT INTO x SELECT 'c','d';
```

```
INSERT INTO x SELECT 'c','c';
INSERT INTO x SELECT 'e','f';

INSERT INTO y SELECT 'a','b';
INSERT INTO y SELECT 'b',NULL;
INSERT INTO y SELECT 'c','d';
INSERT INTO y SELECT 'c','c';
```

这次表 x 和 y 中有两个列，并且可能包含 NULL 值。从上述的输入数据来看，对表 x 和 y 进行 EXCEPT 操作后应该得到（'e'，'f '），显然其存在于 x 表中，但不存在于 y 表中。按照之前介绍的 LEFT JOIN 和 NOT EXISTS 来进行 EXCEPT 集合操作，结果如下：

```
mysql> SELECT DISTINCT x.a,x.b FROM x
    -> LEFT JOIN y
    -> ON x.a = y.a AND x.b = y.b
    -> WHERE y.a IS NULL AND y.b IS NULL;
+------+------+
| a    | b    |
+------+------+
| b    | NULL |
| e    | f    |
+------+------+
2 rows in set (0.00 sec)

mysql> SELECT DISTINCT * FROM x
    -> WHERE NOT EXISTS (
    -> SELECT * FROM y WHERE x.a = y.a AND x.b = y.b );
+------+------+
| a    | b    |
+------+------+
| b    | NULL |
| e    | f    |
+------+------+
2 rows in set (0.00 sec)
```

可以看到，无论采用哪种方式，最后返回的结果集都是（'b'，NULL）和（'e'，'f'）。产生这个问题的“元凶”就是 b 列中的 NULL 值。在对 NULL 进行“等于”操作时，返回的是 NULL，即未知，这在之前已经讨论过。因此要产生正确的结果，需要使用下面的 SQL 语句：

```
mysql>SELECT * FROM (
    -> SELECT DISTINCT 'X' AS source,a,b FROM x
    -> UNION ALL
    -> SELECT DISTINCT 'Y' ,a,b FROM y) AS A
    -> GROUP BY a,b
    -> HAVING count(*)=1 AND A.source='X';
+--------+------+------+
| source | a    | b    |
+--------+------+------+
| X      | e    | f    |
```

```
+--------+------+-------+
1 row in set (0.00 sec)
```

这里需要先通过 UNION ALL 得到所有的集合。接着添加额外的列 source 来区分输入数据来自于哪个表。因为在 UNION 操作中只选择第一个查询的列作为名称，因此只需在第一个输入中指定别名即可。接着对派生表做 GROUP BY 分组操作，并且加上条件来源为第一个输入的表。

相比 EXCEPT DISTINCT，EXCEPT ALL 要复杂得多。EXCEPT ALL 不仅需要考虑行数据是否存在，还要关心每一行出现的次数。假设要返回输入 A “EXCEPT ALL ” 输入 B 的结果，如果一行数据在输入 A 中出现了 x 次，在输入 B 中出现了 y 次，则该行在输出中将出现 MAX（0，$x-y$）次 ，即如果 x 大于 y，该行结果中将出现 $x-y$ 次，否则结果中将不包含这一行数据。

对于之前的表 x 和 y，表 x 中有 3 条记录都为 ('c'，'d')，表 y 中有 1 条记录为（'c'，'d'）。因此，除了记录（'e'，'f '）外，执行 x 表 “EXCEPT ALL” 表 y 后，还应该有两条（'c'，'d'）的记录。这个过程的 SQL 语句如下：

```
SELECT M.a,M.b
FROM (
SELECT a,b,
MAX(CASE WHEN source = 'X' then cnt ELSE 0 END ) AS XCNT,
MAX(CASE WHEN source = 'Y' then cnt ELSE 0 END ) AS YCNT FROM (
SELECT DISTINCT 'X' AS source,a,b,count(*) AS cnt FROM x
GROUP BY a,b
UNION ALL
SELECT DISTINCT 'Y' ,a,b,count(*) FROM y
GROUP BY a,b
) AS P
GROUP BY a,b
) AS M
JOIN nums
WHERE nums.a<=XCNT-YCNT
```

这里使用了数字辅助表 nums 来统计 min（0，$x-y$）行的重复数据。这里的 SQL 语句看起来非常复杂，如果不能直接看明白，可以分别执行各个语句。这里将这个 SQL 语句分三个步骤来理解。先是生成派生表 P，执行如下的 SQL 语句，结果如表 5-13 所示。

```
SELECT a,b,
MAX(CASE WHEN source = 'X' then cnt ELSE 0 END ) AS XCNT,
MAX(CASE WHEN source = 'Y' then cnt ELSE 0 END ) AS YCNT FROM (
SELECT DISTINCT 'X' AS source,a,b,count(*) AS cnt FROM x
GROUP BY a,b
UNION ALL
SELECT DISTINCT 'Y' ,a,b,count(*) FROM y
GROUP BY a,b
) AS P
```

表 5-13 派生表 P

source	a	b	cnt
X	a	b	1
X	b	(NULL)	1
X	c	c	1
X	c	d	3
X	e	f	1
Y	a	b	1
Y	b	(NULL)	1
Y	c	c	1
Y	c	d	1

接着对派生表 P 进行分组操作，通过 MAX(CASE …) 表达式统计出从 X 和 Y 集合中选出的分组 a 和 b 的个数，分别用列 XCNT 和 YCNT 表示，并再生成派生表 M，得到结果如表 5-14 所示。

表 5-14 派生表 M

a	b	XCNT	YCNT
a	b	1	1
b	(NULL)	1	1
c	c	1	1
c	d	3	1
e	f	1	0

最后根据 XCNT−YCNT 的值得到最终数据，并通过 JOIN 表 nums 来得到该行记录的多个副本。通过 EXPLAIN 也能很清楚地得到 SQL 语句的执行计划，如图 5-23 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	PRIMARY	nums	index	PRIMARY	PRIMARY	4	NULL	5	Using index
	1	PRIMARY	<derived2>	ALL	NULL	NULL	NULL	NULL	11	Using where; Using join buffer (flat, BNL join)
	2	DERIVED	<derived3>	ALL	NULL	NULL	NULL	NULL	11	Using temporary; Using filesort
	3	DERIVED	x	ALL	NULL	NULL	NULL	NULL	7	Using temporary; Using filesort
	4	UNION	y	ALL	NULL	NULL	NULL	NULL	4	Using temporary; Using filesort
	NULL	UNION RESULT	<union3,4>	ALL	NULL	NULL	NULL	NULL	NULL	

图 5-23 EXCEPT ALL 语句的执行计划

5.6.4 INTERSECT

INTERSECT 返回在两个输入中都出现的行。和 EXCEPT 一样，不能简单地使用 LEFT JOIN 或者 NOT EXISTS 来解决 INTERSECT 问题，因为同样可能存在 NULL 值的问题。

对于 INTERSECT DISTINCT，可以通过如下 SQL 语句来实现：

```
SELECT a,b FROM (
SELECT DISTINCT a,b FROM x
UNION ALL
SELECT DISTINCT a,b FROM y
) AS P
GROUP BY a,b
HAVING COUNT(*) = 2
```

派生表查询对两个输入的不重复行数据执行 UNION ALL 操作。之后外部查询再按 a 和 b 列进行分组，并且只返回出现过两次的组。也就是说，该查询只返回在两个集中都出现过的不重复行，这就是 INTERSECT DISTINCT 的定义。

与 EXCEPT ALL 一样，INTERSECT ALL 也与行的出现次数有关。如果行 R 在一个输入表中出现了 x 次，在另一个输入表中出现了 y 次，则它应该在结果中出现 MIN(x，y) 次。这个问题的解决方案同样可以参考 EXCEPT ALL，借助数字辅助表 nums 来实现，示例如下：

```
SELECT M.a,M.b
FROM (
 SELECT a,b,MIN(cnt) as mincnt
 FROM (
   SELECT a,b,count(*) AS cnt FROM x
   GROUP BY a,b
   UNION ALL
   SELECT a,b,count(*) FROM y
   GROUP BY a,b
 ) AS P
 GROUP BY a,b HAVING COUNT(*) > 1
) AS M
JOIN nums
WHERE nums.a <= mincnt
```

派生表 P 对每个输入集合的不重复行记录执行 UNION ALL，并计算它们在集合中出现的次数。派生表 M 的查询按照 a 和 b 进行分组，并通过 HAVING 子句筛选出在两个输入集合中都存在的行记录（COUNT(*)>1），之后返回它们的最小计数，即 MIN(x，y)。最后通过数字辅助表 nums，并根据条件 nums.a <= mincnt 来生成行记录的多个副本。

5.7　小结

本章介绍了联接和集合操作的各个方面，展示了一些非常方便的新技术。外部联接有 ANSI SQL 89 和 SQL 92 两种标准，无论是哪种标准，MySQL 数据库都对其进行很完美地支持。对于集合操作，MySQL 数据库只简单地支持 UNION DISTINCT 和 UNION ALL 操作。但是在 UNION 的基础之上，完全可以实现 EXCEPT 和 INTERSECT 的集合运算。在处理 EXCEPT ALL 和 INTERSECT ALL 时，又可以使用一种通过数字辅助表对行记录进行复制的新技术。

第 6 章

聚合和旋转操作

本章将介绍 MySQL 数据库中的聚合技术。首先介绍 MySQL 数据库的聚合算法，在此基础上介绍了一些通过聚合解决问题的方案，如附加属性聚合、连续聚合（累积、滑动、年初至今）等。最后通过上述聚合操作介绍旋转操作。本章在使用前面已经介绍过的技术的同时引入一些新的技术。

6.1　聚合

6.1.1　聚合函数

MySQL 数据库支持聚合（aggregate）操作，按照分组对同一组内的数据聚合进行统计操作。目前 MySQL 数据库支持的聚合函数有：

- AVG()
- BIT_AND()
- BIT_OR()
- BIT_XOR()
- COUNT(DISTINCT)
- COUNT()
- GROUP_CONCAT()
- MAX()
- MIN()
- STD()
- STDDEV_POP()
- STDDEV_SAMP()
- STDDEV()
- SUM()
- VAR_POP()
- VAR_SAMP()
- VARIANCE()

可以参照 MySQL 官方手册对各聚合函数的说明来了解这些函数。这里主要介绍 GROUP_CONCAT 聚合函数。GROUP_CONCAT 将分组后的非 NULL 数据通过连接符进行拼接，对 NULL 数据返回 NULL 值。举个例子来更好地说明 GROUP_CONCAT 聚合函数的作用。先根据如下语句生成测试表 z 并填充数据。

```
CREATE TABLE z ( a INT, b INT );

INSERT INTO z SELECT 1,200;
INSERT INTO z SELECT 1,100;
INSERT INTO z SELECT 1,100;
INSERT INTO z SELECT 2,400;
INSERT INTO z SELECT 2,500;
INSERT INTO z SELECT 3,NULL;
```

接着根据 a 列进行分区，并使用聚合函数 GROUP_CONCAT，SQL 语句如下，得到的结果如表 6-1 所示。

```
SELECT a,GROUP_CONCAT(b)
FROM z
GROUP BY a;
```

表 6-1 GROUP_CANCAT 的结果

a	GROUP_CONCAT(b)	a	GROUP_CONCAT(b)
1	200,100,100	3	NULL
2	400,500		

可以看到对分组后 b 列的值进行了拼接。如对“1”这个分组，经 GROUP_CONCAT 聚合后得到（200，100，100）；对“2”这个分组，经 GROUP_CONCAT 聚合后得到（400，500）；对于“3”这个分组，经 GROUP_CONCAT 聚合后得到 NULL 值。

此外，GROUP_CONCAT 聚合函数还有一些其他的特性，其语法如下：

```
GROUP_CONCAT([DISTINCT] expr [,expr ...]
[ORDER BY {unsigned_integer | col_name | expr}
[ASC | DESC] [,col_name ...]]
[SEPARATOR str_val])
```

DISTINCT 选项可以去除重复值，ORDER BY 选项可以对列进行排序，SEPARATOR 用于选择拼接的字符串值。例如在上述例子中，可以对 b 列进行去重复并按递减的顺序排序，同时用“：”拼接字符串。其 SQL 语句可写为：

```
SELECT a,GROUP_CONCAT(DISTINCT b ORDER BY b DESC SEPARATOR ':')
FROM z
GROUP BY a;
```

最后得到的结果如表 6-2 所示。

表 6-2 GROUP_CONCAT 的结果 2

a	GROUP_CONCAT(b)	a	GROUP_CONCAT(b)
1	200:100	3	NULL
2	500:400		

6.1.2 聚合的算法

MySQL 数据库仅支持流聚合，而其他的数据库可能会支持散列聚合。流聚合依赖于获得的存储在 GROUP BY 列中的数据。如果一个 SQL 查询中包含的 GROUP BY 语句多于一行，流聚合会先根据 GROUP BY 对行进行排序。

下面是流聚合算法的伪代码：

```
sort result according to group by columns
foreach row in result:
```

```
    begin
            if the row does not match the current group by columns
            begin
                    print current aggregate results
                    clear current aggregate results
                    set current group by columns to input row
            end
    update aggregate results with input row
end
```

例如，为了计算 MAX，流聚合会考虑每个输入行的情况，如果输入行属于目前的分组，则流聚合可通过判断当前分组的 MAX 值与该行的值的比较来更新当前分组的 MAX 值。如果输入行不属于当前的分组，则输出当前分组的 MAX 值，并把输入行作为新的分组，同时将该分组的 MAX 值设为该行的值。

下面的查询计算每个地址和每个城市的订单数量，其查询计划如图 6-1 所示。

```
SELECT shipaddress,shipcity,count(*)
FROM orders
GROUP BY shipaddress,shipcity
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	orders	ALL	NULL	NULL	NULL	NULL	867	Using temporary; Using filesort

图 6-1　GROUP BY 的执行计划

可以看到 Extra 列中显示 Using temporary 和 Using filesort，这是因为采用流聚合算法会先对数据进行排序。若要避免第一次的排序操作，可以在分组的列上添加索引，对于这个例子，可以添加（shipaddress，shipcity）的联合索引，语句如下：

```
ALTER TABLE orders
ADD INDEX idx_ship_address_city (shipaddress,shipcity)
```

这时 SQL 语句的执行计划变为如图 6-2 所示。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	orders	index	NULL	idx_ship_address_city	231	NULL	867	Using index

图 6-2　添加索引后 GROUP BY 的执行计划

这里使用了之前添加的 idx_ship_address_city 索引，此时后面的 Extra 列显示了 Using index，而非之前的 Using temporary 和 Using filesort。因为对于索引而言，其本身就是顺序的，不再需要额外的排序操作。

此外，对于 SELECT DISTINCT 操作，其本质上和 GROUP BY 是一样的，例如：

```
SELECT DISTINCT customerid FROM orders
```

也可以这样使用：

```
SELECT customerid FROM orders GROUP BY customerid
```

这两个查询都有着相同的执行计划，如图 6-3 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	orders	range	NULL	CustomerID	16	NULL	217	Using index for group-by

图 6-3　SELECT DISTINCT 的执行计划

6.2　附加属性聚合

在第 4 章讨论过为每个员工返回最新订单的问题，其解决方案是使用子查询，最后通过派生表来优化子查询。记住子查询的性能严重依赖于索引。而在实际的生产环境中，用户不可能总是允许添加任意数量的索引。因此对于为每个员工返回最新订单的问题，这里可以使用聚合来完成。解决这个问题的聚合方式是使用下面的逻辑（伪代码）：

```
SELECT employeeid, MAX(orderdate,orderid)
FROM orders
GROUP BY employeeid
```

可惜的是，SQL 并不支持上述语句，因为 MAX 函数不允许对两列的数据进行判断。但是可以用别的方法将两列的数据串联模拟为一列，并且保证数据是整型，从而可以执行 MAX。

这里的技巧是为每个属性使用固定长度的字符串，并且以不改变顺序行为的方式转化这些属性。如果处理的都是正整数，则可以使用算术计算来合并值。假设有两个数 m 和 n，各占用 4 字节，可以将 m 左移 4 字节再加上 n 合并成一个整数，例如：

```
p = m << 4 + n
```

这个整数的特点是排序规则可以保持不变。

下面语句为每个员工返回具有 MAX（OrderDate）值的订单，并使用二进制串。

```
SELECT
   CONV(SUBSTRING(binstr,9,8),16,10) AS orderid,
   SUBSTRING(binstr,25) AS customerid,
   employeeid,
   CAST(
   FROM_UNIXTIME(CONV(SUBSTRING(binstr,1,8),16,10)) AS DATE
   )AS orderdate,
   CAST(
   FROM_UNIXTIME(CONV(SUBSTRING(binstr,17,8),16,10)) AS DATE)
   AS requireddate
FROM (
   SELECT
```

```
            employeeid,
            MAX(
            CONCAT(
            HEX(UNIX_TIMESTAMP(orderdate)),
            LPAD(HEX(orderid),8,'0'),
            LPAD(HEX(UNIX_TIMESTAMP(requireddate)),8,'0'),
            customerid
            )
            ) AS binstr
FROM orders
GROUP BY employeeid
) AS d;
```

这里使用 HEX 函数将数字都转换为十六进制，然后通过 CONCAT 函数进行拼接。使用 CONCAT 函数是因为使用字符串拼接可以避免使用左移进行计算，但是得到的结果却是相同的。LPAD 函数用来保证每个字段占用的字节数，以保证 CONCAT 的正确性。在 SELECT 中，再根据 binstr 分别得到要求返回的列。

这个解决方案的真正好处是：无论是否有合适的索引，都只扫描一次数据。这句 SQL 语句的执行计划如图 6-4 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	PRIMARY	<derived2>	ALL	NULL	NULL	NULL	NULL	9	
	2	DERIVED	orders	index	NULL	EmployeeID	5	NULL	837	

图 6-4　带附加属性查询的执行计划

上述 SQL 语句运行的时间为 0.156 秒。关键是不需要对表进行唯一索引的添加。这个方法看似复杂，理解了其本质这个解决方案还是非常简单的。

当排序列和附加属性的排序相反时，情况就显得稍微有些复杂。例如要执行 MAX（OrderDate）和 MIN（OrderID）时，可以通过 MAXINT-OrderID 来保持列的顺序。下面的 SQL 语句实现了这个逻辑。

```
SELECT
   2147483647-CONV(SUBSTRING(binstr,9,8),16,10) AS orderid,
   SUBSTRING(binstr,25) AS customerid,
   employeeid,
   CAST(
   FROM_UNIXTIME(CONV(SUBSTRING(binstr,1,8),16,10)) AS DATE
   )AS orderdate,
   CAST(
   FROM_UNIXTIME(CONV(SUBSTRING(binstr,17,8),16,10)) AS DATE)
   AS requireddate
FROM (
   SELECT
          employeeid,
          MAX(
```

```
            CONCAT(
            HEX(UNIX_TIMESTAMP(orderdate)),
            LPAD(HEX(2147483647-orderid),8,'0'),
            LPAD(HEX(UNIX_TIMESTAMP(requireddate)),8,'0'),
            customerid
            )
            ) AS binstr
FROM orders
GROUP BY employeeid
) AS d;
```

这里的 2 147 483 647 就是 UNSIGNED INT 的最大值。

6.3 连续聚合

连续聚合是按时间顺序对有序数据进行聚合的操作。连续聚合问题可能会有许多变体，本节将介绍其中几个重要的问题。

在下面的示例中将使用 EmpOrders 表，该表用于存放每位员工每月发生的订购数量。运行如下代码创建 EmpOrders 表并填充示例数据。

```
CREATE TABLE EmpOrders
( empid INT NOT NULL,
  ordermonth DATE NOT NULL,
  qty INT NOT NULL,
  PRIMARY KEY ( empid, ordermonth )
);

INSERT INTO EmpOrders
SELECT a.employeeid,
orderdate AS OrderDate,
SUM(quantity) AS qty FROM orders a
INNER JOIN orderdetails b
ON a.orderid=b.orderid
GROUP BY employeeid,DATE_FORMAT(orderdate,'%Y-%m')
;
```

下面的查询返回 EmpOrders 表的内容，其结果如表 6-3 所示。

```
SELECT
   empid,
   DATE_FORMAT(ordermonth,'%Y-%m') as ordermonth,
   qty
FROM EmpOrders
ORDER BY empid,ordermonth;
```

表 6-3　EmpOrder 表中的内容

empid	ordermonth	qty	empid	ordermonth	qty
1	1996-07	121	2	1996-07	50
1	1996-08	247	2	1996-08	94
1	1996-09	255	2	1996-09	137
1	1996-10	143	2	1996-10	248
1	1996-11	318	2	1996-11	237
1	1996-12	536	…	…	…
…	…	…			

下面将根据 EmpOrders 表讨论 3 个连续聚合问题：累积、滑动、年初至今。

6.3.1　累积聚合

累计聚合为聚合从序列内第一个元素到当前元素的数据，如为每个员工返回每月开始到现在累积的订单数量和平均订单数量。

行号问题有两个解决方案，分别为使用子查询和使用联接。子查询的方法通常比较直观，可读性强。但是在要求进行聚合时，子查询可能需要为每个聚合扫描一次数据，而联接方法通常只需要扫描一次数据就可以得到结果。下面的查询使用联接来得到结果。

```
SELECT
   a.empid,
   a.ordermonth,a.qty AS thismonth,
   SUM(b.qty) AS total,
   CAST(AVG(b.qty) AS DECIMAL(5,2)) AS avg
FROM EmpOrders a
INNER JOIN EmpOrders b
   ON a.empid = b.empid
   AND b.ordermonth <= a.ordermonth
GROUP BY a.empid,a.ordermonth,a.qty
ORDER BY a.empid,a.ordermonth;
```

得到的结果如表 6-4 所示。

表 6-4　每个员工每月的累计订单数量

empid	ordermonth	thismonth	total	avg
1	1996-07	121	121	121.00
1	1996-08	247	368	184.00
1	1996-09	255	623	207.67
1	1996-10	143	766	191.50
1	1996-11	318	1084	216.80
1	1996-12	536	1620	270.00
…	…	…		

（续）

empid	ordermonth	thismonth	total	avg
2	1996-07	50	50	50.00
2	1996-08	94	144	72.00
2	1996-09	137	281	93.67
2	1996-10	248	529	132.25
2	1996-11	237	766	153.20
…	…	…		

上述 SQL 语句的执行计划如图 6-5 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	a	ALL	PRIMARY	NULL	NULL	NULL	192	Using temporary; Using filesort
	1	SIMPLE	b	ref	PRIMARY	PRIMARY	4	tpcc.a.empid	10	Using where

图 6-5　通过联接解决累积问题的执行计划

正如前面所说的，这个问题可以通过如下的子查询来完成。

```
SELECT empid,ordermonth,qty AS thismonth,
    (SELECT SUM(b.qty) FROM EmpOrders AS b
        WHERE a.empid = b.empid
           AND b.ordermonth <= a.ordermonth ) AS total,
    CAST((SELECT AVG(b.qty) FROM EmpOrders AS b
        WHERE a.empid = b.empid
           AND b.ordermonth <= a.ordermonth ) AS DECIMAL(5,2)) AS avg
FROM EmpOrders a
GROUP BY a.empid,a.ordermonth;
```

子查询的执行计划如图 6-6 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	PRIMARY	a	index	NULL	PRIMARY	7	NULL	192	
	3	DEPENDENT SUBQUERY	b	ref	PRIMARY	PRIMARY	4	func	10	Using where
	2	DEPENDENT SUBQUERY	b	ref	PRIMARY	PRIMARY	4	func	10	Using where

图 6-6　通过子查询解决累积聚合问题的执行计划

从图 6-6 可以看出，在 select_type 列中出现了 DEPENDENT SUBQUERY，显然需要进行相关子查询的操作，并且需要两次。在这个问题中，子查询的性能并不会特别差。因为每个员工的订单数量相对整张表来说还是比较小的，即只对一小部分的行进行连续聚合，而且还有适合的索引可以利用，因此并不需要担心性能问题。

此外可能还需要筛选数据，例如只需要返回每个员工达到某一目标之前每月的订单情况。这里假设统计每个员工的合计订单数量达到 1000 之前的累积情况。这里，可以使用 HAVING 过滤器来完成查询，SQL 语句如下，得到的结果如表 6-5 所示。

```
SELECT
   a.empid,
   a.ordermonth,a.qty AS thismonth,
   SUM(b.qty) AS total,
   CAST(AVG(b.qty) AS DECIMAL(5,2)) AS avg
FROM EmpOrders a
INNER JOIN EmpOrders b
   ON a.empid = b.empid
   AND b.ordermonth <= a.ordermonth
GROUP BY a.empid,a.ordermonth,a.qty
HAVING SUM(b.qty) < 1000
ORDER BY a.empid,a.ordermonth;
```

表 6-5　累积订单小于 1000 之前各员工每月的订单情况

empid	ordermonth	thismonth	total	avg
1	1996-07-17	121	121	121.00
1	1996-08-01	247	368	184.00
1	1996-09-12	255	623	207.67
1	1996-10-09	143	766	191.50
2	1996-07-25	50	50	50.00
2	1996-08-09	94	144	72.00
2	1996-09-02	137	281	93.67
2	1996-10-11	248	529	132.25
2	1996-11-04	237	766	153.20
3	1996-07-08	182	182	182.00
…	…	…	…	…

这里并没有统计达到 1000 时该月的订单情况，如果要进行统计，则情况又有点复杂。如果指定 SUM(b.qty)<=1000，则只有该月订单数量正好为 1000 才进行统计，否则不会对该月进行统计。因此这个问题的过滤，可以从另外一方面来考虑。当累积订单小于 1000 时，累积订单与上个月的订单之差是小于 1000 的，同时也能对第一个订单数量超过 1000 的月份进行统计。故该解决方案的 SQL 语句如下，得到的结果如表 6-6 所示。

```
SELECT
   a.empid,
   a.ordermonth,a.qty AS thismonth,
   SUM(b.qty) AS total,
   CAST(AVG(b.qty) AS DECIMAL(5,2)) AS avg
FROM EmpOrders a
INNER JOIN EmpOrders b
   ON a.empid = b.empid
   AND b.ordermonth <= a.ordermonth
GROUP BY a.empid,a.ordermonth,a.qty
HAVING (SUM(b.qty) - a.qty < 1000)
ORDER BY a.empid,a.ordermonth;
```

表 6-6　统计累积订单小于 1000 但包含大于 1000 时的第一个月

empid	ordermonth	thismonth	total	avg
1	1996-07	121	121	121.00
1	1996-08	247	368	184.00
1	1996-09	255	623	207.67
1	1996-10	143	766	191.50
1	**1996-11**	**318**	**1084**	**216.80**
2	1996-07	50	50	50.00
2	1996-08	94	144	72.00
2	1996-09	137	281	93.67
2	1996-10	248	529	132.25
2	1996-11	237	766	153.20
2	**1996-12**	**319**	**1085**	**180.83**
3	1996-07	182	182	182.00
…	…	…	…	…

如果只想返回达到累积订单数为 1000 的当月数据，不返回之前的月份，则可以对上述 SQL 语句进一步过滤，再添加累积订单数量大于等于 1000 的条件。该问题的 SQL 语句如下，生成的结果如表 6-7 所示。

```
SELECT
   a.empid,
   a.ordermonth,a.qty AS thismonth,
   SUM(b.qty) AS total,
   CAST(AVG(b.qty) AS DECIMAL(5,2)) AS avg
FROM EmpOrders a
INNER JOIN EmpOrders b
   ON a.empid = b.empid
   AND b.ordermonth <= a.ordermonth
GROUP BY a.empid,a.ordermonth,a.qty
HAVING (SUM(b.qty) - a.qty < 1000) AND SUM(b.qty) >=1000
ORDER BY a.empid,a.ordermonth;
```

表 6-7　仅统计累积数据大于 1000 时当月的订单情况

empid	ordermonth	thismonth	total	avg
1	1996-11	318	1084	216.80
2	1996-12	319	1085	180.83
3	1997-01	364	1304	186.29
4	1996-10	613	1439	359.75
5	1997-05	247	1213	173.29
6	1997-01	64	1027	171.17
7	1997-03	191	1069	152.71
8	1997-01	305	1228	175.43
9	1997-06	161	1007	125.88

6.3.2　滑动聚合

滑动聚合是按顺序对滑动窗口范围内的数据进行聚合的操作。与累积聚合不同，滑动聚合并不是统计开始计算的位置到当前位置的数据。这里以统计最近三个月中员工每月的订单情况为例来介绍滑动聚合。

滑动聚合和累积聚合解决方案的主要区别在于联接的条件不同。滑动聚合的联接条件不再是 b.ordermonth <= a.ordermonth，而应该是 b.ordermonth 大于前三个月的月份，并且小于等于当前月份。因此滑动聚合解决方案的 SQL 语句如下，得到的结果如表 6-8 所示。

```
SELECT
   a.empid,
   DATE_FORMAT(a.ordermonth,"%Y-%m") AS ordermonth,
   a.qty AS thismonth,
   SUM(b.qty) AS total,
   CAST(AVG(b.qty) AS DECIMAL(5,2)) AS avg
FROM EmpOrders a
INNER JOIN EmpOrders b
   ON a.empid = b.empid
   AND b.ordermonth > DATE_ADD(a.ordermonth,INTERVAL -3 MONTH)
   AND b.ordermonth <= a.ordermonth
GROUP BY a.empid,DATE_FORMAT(a.ordermonth,"%Y-%m"),a.qty
ORDER BY a.empid,a.ordermonth;
```

表 6-8　每个员工最近三个月的订单聚合

empid	ordermonth	thismonth	total	avg
…	1996-07	121	121	121.00
1	1996-08	247	368	184.00
1	1996-09	255	623	207.67
1	1996-10	143	645	215.00
1	1996-11	318	716	238.67
1	1996-12	536	997	332.33
1	1997-01	304	1158	386.00
1	1997-02	168	1008	336.00
1	1997-03	275	747	249.00
…	…	…	…	…
2	1996-07	50	50	50.00
2	1996-08	94	144	72.00
2	1996-09	137	281	93.67
2	1996-10	248	479	159.67
2	1996-11	237	622	207.33
2	1996-12	319	804	268.00
2	1997-01	230	786	262.00
2	1997-02	36	585	195.00
…	…	…	…	…

该解决方案返回的是三个月为一个周期的滑动聚合，但是每个用户包含前两个月且未满 3 个月的聚合。如果只希望返回满 3 个月的聚合，不返回未满 3 个月的聚合，可以使用 HAVING 过滤器进行过滤，过滤的条件为 MIN(b.ordermonth)=DATE_ADD(a.ordermonth,INTERVAL-2 MONTH)，例如：

```
SELECT
   a.empid,
   a.ordermonth AS ordermonth,
  a.qty AS thismonth,
   SUM(b.qty) AS total,
   CAST(AVG(b.qty) AS DECIMAL(5,2)) AS avg
FROM EmpOrders a
INNER JOIN EmpOrders b
   ON a.empid = b.empid
   AND b.ordermonth > DATE_ADD(a.ordermonth,INTERVAL -3 MONTH)
   AND b.ordermonth <= a.ordermonth
GROUP BY a.empid,a.ordermonth,a.qty
HAVING MIN(b.ordermonth)
          = DATE_ADD(a.ordermonth,INTERVAL -2 MONTH)
ORDER BY a.empid,a.ordermonth;
```

6.3.3 年初至今聚合

年初至今聚合和滑动聚合类似，不同的地方仅在于统计的仅为当前一年的聚合。唯一的区别体现在下限的开始位置上。在年初至今的问题中，下限为该年的第一天，而滑动聚合的下限为前 *N* 个月的第一天。因此，年初至今问题的解决方案如下所示，得到的结果如表 6-9 所示。

```
SELECT
   a.empid,
   a.ordermonth AS ordermonth,
   a.qty AS thismonth,
   SUM(b.qty) AS total,
   CAST(AVG(b.qty) AS DECIMAL(5,2)) AS avg
FROM EmpOrders a
INNER JOIN EmpOrders b
   ON a.empid = b.empid
   AND b.ordermonth >= DATE_FORMAT(a.ordermonth,"%Y-01-01")
   AND b.ordermonth <= a.ordermonth
GROUP BY a.empid,a.ordermonth,a.qty
ORDER BY a.empid,a.ordermonth;
```

表 6-9　每个员工的年初至今聚合

empid	ordermonth	thismonth	total	avg
1	1996-07-01	121	121	121.00
1	1996-08-01	247	368	184.00
1	1996-09-01	255	623	207.67
1	1996-10-01	143	766	191.50
1	1996-11-01	318	1084	216.80
1	1996-12-01	536	1620	270.00
1	1997-01-01	304	304	304.00
1	1997-02-01	168	472	236.00
1	1997-03-01	275	747	249.00
…	…	…	…	…
2	1996-07-01	50	50	50.00
2	1996-08-01	94	144	72.00
2	1996-09-01	137	281	93.67
2	1996-10-01	248	529	132.25
2	1996-11-01	237	766	153.20
2	1996-12-01	319	1085	180.83
2	1997-01-01	230	230	230.00
2	1997-02-01	36	266	133.00
…	…	…	…	…

6.4　Pivoting

Pivoting 是一项可以把行旋转为列的技术。在执行 Pivoting 的过程中可能会使用到聚合。Pivoting 技术应用得非常广泛。在这一节中，先通过开放架构（Open Schema）来展现 MySQL 数据库中的 Pivoting 技术，之后介绍关系除法（Rational Division）和格式化聚合数据等问题。

注意　这里讨论的都是静态 Pivoting 查询，即用户需要提前知道旋转的属性列的值。对于动态 Pivoting，需要动态地构造查询字符串。

6.4.1　开放架构

开放架构是一种用于频繁更改架构的一种设计模式。利用关系数据库和 SQL 语句可以非常有效地处理 DML（Data Manipulation Language，数据操纵语句），包括 INSERT、SELECT、UPDATE 和 DELETE。然而，DDL（Data Definition Language，数据定义语句）在频繁进行架构更改时显得十分不方便。例如要对表结构进行修改，用户必须添加、修改或

删除列，而这种操作正是关系数据库不擅长的方面。

因此，在频繁更改架构的情况下，可以在一个表中存储所有的数据，每行存储一个属性的值，多用 VARCHAR 来存储，因为其可容纳各种类型的数据。下面的语句生成一张开放架构的表 t。

```
CREATE TABLE t
(
id INT,
attribute VARCHAR(10),
value VARCHAR(20),
PRIMARY KEY(id,attribute)
);

INSERT INTO t SELECT 1,'attr1','BMW';
INSERT INTO t SELECT 1,'attr2','100';
INSERT INTO t SELECT 1,'attr3','2010-01-01';
INSERT INTO t SELECT 2,'attr2','200';
INSERT INTO t SELECT 2,'attr3','2010-03-04';
INSERT INTO t SELECT 2,'attr4','M';
INSERT INTO t SELECT 2,'attr5','55.60';
INSERT INTO t SELECT 3,'attr1','SUV';
INSERT INTO t SELECT 3,'attr2','10';
INSERT INTO t SELECT 3,'attr3','2011-11-11';
```

表 t 的内容如表 6-10 所示。

表 6-10　开放架构表 t 的内容

id	attribute	value	id	attribute	value
1	attr1	BMW	2	attr4	M
1	attr2	100	2	attr5	55.60
1	attr3	2010-01-01	3	attr1	SUV
2	attr2	200	3	attr2	10
2	attr3	2010-03-04	3	attr3	2011-11-11

从上面的例子中可以看到，在对通过开放架构设计的表进行添加、修改或删除表和列时，只需要通过 INSERT、UPDATE、DELETE 操作来完成逻辑架构的更改即可。当然使用这种方法可能导致关系数据库的其他特性无法使用，如完整性约束、SQL 优化等，同时查询数据变得不如使用之前的 SQL 语句来得直接和直观。所以，对于利用开放架构设计的表，一般使用 Pivoting 技术来查询数据。

Pivoting 技术需要和聚合一起使用，首先要确定结果的行数与表中行数的关系。对于开放架构表 t，应该有 3 行 5 列，这可以通过分组 id 来得到。因此可以通过下列 Pivoting 技术进行行列互转以得到数据。

```
SELECT id,
    MAX(CASE WHEN attribute='attr1' THEN value END) AS attr1,
    MAX(CASE WHEN attribute='attr2' THEN value END) AS attr2,
    MAX(CASE WHEN attribute='attr3' THEN value END) AS attr3,
    MAX(CASE WHEN attribute='attr4' THEN value END) AS attr4,
    MAX(CASE WHEN attribute='attr5' THEN value END) AS attr5
FROM t
GROUP BY id;
```

Pivoting 先根据 id 进行分组，确定行列互转后记录的行数。之后通过已知的 5 个属性来确定行列互转后有 5 列数据，并通过 CASE 得到每列的值。由于使用了分组技术，因此一定要使用分组函数来取得列的值，故这里使用 MAX 函数，当然也可以使用 MIN 函数。最后得到的结果如表 6-11 所示。

表 6-11　行列旋转后表中的数据

id	attr1	attr2	attr3	attr4	attr5
1	BMW	100	2010-01-01	NULL	NULL
2	NULL	200	2010-03-04	M	55.60
3	SUV	10	2011-11-11	NULL	NULL

这种旋转方式是非常高效的，因为它只对表进行一次扫描。另外，这是一种静态的 Pivoting，用户必须事先知道一共有多少个属性，然而对于一般的开放架构表，用户都会定义一个最大的属性个数，这样可以比较容易地进行 Pivoting。

6.4.2　关系除法

关系除法（Rational Division）和常见的关系运算 JOIN、SEMI JOIN 一样，都是一个关系代数，可用 ÷ 表示。Rational Division 的定义如下：

$$R \div S = \{ t[a_1,...,a_n] : t \in R \wedge \forall s \in S ((t[a1,...,an] \cup s) \in R) \}$$

其中 $[a_1,...,a_n]$ 是集合 R 中的唯一属性记录，$t[a_1,...,a_n]$ 表示结果，是 $[a_1,...,a_n]$ 中受限的一部分记录，并且这部分记录满足 $(t[a_1,...,a_n] \cup s) \in R$。

当除数集合中的元素数量较小时，Pivoting 可用于解决关系除法问题。先通过下列语句创建表 t 并填充数据。

```
CREATE TABLE t (
orderid VARCHAR(10) NOT NULL,
productid INT NOT NULL,
PRIMARY KEY(orderid,productid)
);

INSERT INTO t SELECT 'A',1;
INSERT INTO t SELECT 'A',2;
```

```
INSERT INTO t SELECT 'A',3;
INSERT INTO t SELECT 'A',4;
INSERT INTO t SELECT 'B',2;
INSERT INTO t SELECT 'B',3;
INSERT INTO t SELECT 'B',4;
INSERT INTO t SELECT 'C',3;
INSERT INTO t SELECT 'C',4;
INSERT INTO t SELECT 'D',4;
```

表 t 存储订单中包含的产品，比如 A 订单所包含的产品的 ID 为 1、2、3、4，B 订单所包含的产品的 ID 为 2、3、4，以此类推。这是一个比较典型的关系除法问题。用 Pivoting 技术可以把每个订单中的产品旋转到单独的列中。例如要查询包含 productid 为 2、3、4 的订单，可以采用如下方法：

```
SELECT orderid
FROM (
SELECT
orderid,
MAX(CASE WHEN productid=2 THEN 1 END) AS p2,
MAX(CASE WHEN productid=3 THEN 1 END) AS p3,
MAX(CASE WHEN productid=4 THEN 1 END) AS p4
FROM t
GROUP BY orderid
) AS P
WHERE p2=1 AND p3=1 AND p4 = 1;
```

上述 SQL 语句返回“A”和“B”。如果单独运行派生表的子查询，将会得到每个订单对应的产品 ID，得到的结果如表 6-12 所示。

表 6-12　派生表 P 中的内容

orderid	p2	p3	p4
A	1	1	1
B	1	1	1
C	NULL	1	1
D	NULL	NULL	1

对于这个问题，聚合函数可以使用 COUNT 来替换 MAX，这会让派生表的结果显得更加直观。此时若产品存在则返回为 1，不存在则返回 0 而不是 NULL，故 SQL 语句可调整为：

```
SELECT orderid
FROM (
SELECT orderid,
COUNT(CASE WHEN productid=2 THEN 1 END) AS p2,
COUNT(CASE WHEN productid=3 THEN 1 END) AS p3,
MAX(CASE WHEN productid=4 THEN 1 END) AS p4
FROM t
```

```
GROUP BY orderid
) AS P
WHERE p2=1 AND p3=1 AND p4 = 1;
```

6.4.3　格式化聚合数据

Pivoting 技术还可以用来格式化聚合数据，一般用于报表的展现。为了演示用 Pivoting 技术来格式化聚合数据，下面给出一个例子。先根据如下语句创建并填充表 t。

```
CREATE TABLE t (
orderid INT NOT NULL,
orderdate DATE NOT NULL,
empid INT NOT NULL,
custid VARCHAR(10) NOT NULL,
qty INT NOT NULL,
PRIMARY KEY (orderid,orderdate)
);

INSERT INTO t SELECT 1,'2010-01-02',3,'A',10;
INSERT INTO t SELECT 2,'2010-04-02',2,'B',20;
INSERT INTO t SELECT 3,'2010-05-02',1,'A',30;
INSERT INTO t SELECT 4,'2010-07-02',3,'D',40;
INSERT INTO t SELECT 5,'2011-01-02',4,'A',20;
INSERT INTO t SELECT 6,'2011-01-02',3,'B',30;
INSERT INTO t SELECT 7,'2011-01-02',1,'C',40;
INSERT INTO t SELECT 8,'2009-01-02',2,'A',10;
INSERT INTO t SELECT 9,'2009-01-02',3,'B',20;
```

表 t 的内容如表 6-13 所示。

表 6-13　表 t 的内容

orderid	orderdate	empid	custid	qty
1	2010-01-02	3	A	10
2	2010-04-02	2	B	20
3	2010-05-02	1	A	30
4	2010-07-02	3	D	40
5	2011-01-02	4	A	20
6	2011-01-02	3	B	30
7	2011-01-02	1	C	40
8	2009-01-02	2	A	10
9	2009-01-02	3	B	20

可以将表 t 看做一张汇总表，比如网上商城的购物明细。这份汇总表显示了订单号、订单日期、员工编号、消费者编号和订单数量。要在此汇总表的基础上进一步统计每个消费者每年的订单数量，可能会想到通过分组来得到结果，例如：

```
SELECT custid,YEAR(orderdate) AS year,SUM(qty) AS sum_qty
FROM t GROUP BY custid,YEAR(orderdate);
```

上述 SQL 语句得到的结果如表 6-14 所示。

表 6-14　聚合后得到的结果

custid	year	sum_qty	custid	year	sum_qty
A	2009	10	B	2010	20
A	2010	40	B	2011	30
A	2011	20	C	2011	40
B	2009	20	D	2010	40

上述结果没有任何问题，只是显示可能不够直观。如果可以通过旋转得到如表 6-15 所示的输出结果，那就直观和清晰多了。

表 6-15　每个消费者各年的订单情况

custid	2009	2010	2011
A	10	40	20
B	20	20	30
C	0	0	40
D	0	40	0

这里同样可以使用 Pivoting 技术。与之前唯一不同的是，此处不再使用聚合函数 MAX，而是使用 SUM 函数。这个解决方案的 SQL 语句如下：

```
SELECT
    custid,
    IFNULL(SUM(CASE WHEN orderyear=2009 THEN qty END ),0) AS "2009",
    IFNULL(SUM(CASE WHEN orderyear=2010 THEN qty END ),0) AS "2010",
    IFNULL(SUM(CASE WHEN orderyear=2011 THEN qty END ),0) AS "2011"
FROM
(SELECT custid,YEAR(orderdate) AS orderyear,qty
FROM t ) AS p
GROUP BY custid;
```

上述 SQL 语句中的 IFNULL 函数用来将 NULL 值返回为 0，代表该年消费者没有产生任何订单操作。

使用 Pivoting 技术来格式化聚合数据会遇到一个问题，即当旋转的元素非常多时，会产生较长的查询字符串。要缩短查询的字符长度，可以预先产生一张矩阵表，包含每个要旋转列的属性。运行如下语句创建并填充矩阵表 Matrix。

```
CREATE TABLE Matrix (
orderyear INT PRIMARY KEY,
```

```
y2009 INT NULL,
y2010 INT NULL,
y2011 INT NULL);

INSERT INTO Matrix SELECT 2009,1,0,0;
INSERT INTO Matrix SELECT 2010,0,1,0;
INSERT INTO Matrix SELECT 2011,0,0,1;
```

矩阵表 Matrix 的内容如表 6-16 所示。

表 6-16　Pivoting 后的格式化聚合数据

orderyear	y2009	y2010	y2011
2009	1	0	0
2010	0	1	0
2011	0	0	1

因此可以通过将表 t 和表 Matrix 进行联接把原来的：

```
SUM(CASE WHEN orderyear=N THEN qty END) AS N
```

替换为：

```
SUM(qty*yN) AS N
```

完整的 SQL 查询语句为：

```
SELECT custid,
    SUM(qty*y2009) AS "2009",
    SUM(qty*y2010) AS "2010",
    SUM(qty*y2011) AS "2011"
FROM
(SELECT custid,YEAR(orderdate) AS orderyear,qty
FROM t ) AS O
INNER JOIN Matrix AS P
    ON O.orderyear = P.orderyear
GROUP BY custid;
```

6.5　Unpivoting

可以将 Unpivoting 看做 Pivoting 操作的反向操作，即将列旋转为行，如将表 6-15 的内容旋转为表 6-14 的内容。要完成这个示例，需要根据下列语句创建并填充表 p，p 的数据即为表 6-15 中的内容。

```
CREATE TABLE p (
custid VARCHAR(10) NOT NULL,
y2009 INT NULL,
```

```
y2010 INT NULL,
y2011 INT NULL,
PRIMARY KEY ( custid )
);

INSERT INTO p
SELECT
    custid,
    IFNULL(SUM(CASE WHEN orderyear=2009 THEN qty END ),0) AS "2009",
    IFNULL(SUM(CASE WHEN orderyear=2010 THEN qty END ),0) AS "2010",
    IFNULL(SUM(CASE WHEN orderyear=2011 THEN qty END ),0) AS "2011"
FROM
(SELECT custid,YEAR(orderdate) AS orderyear,qty
FROM t ) AS p
GROUP BY custid;
```

这里把表 6-15 的内容导入表 p 中，如果想得到表 6-14 中的内容，这个问题就变成了 Unpivoting 问题。解决这个问题需要将列旋转为行。这里使用的技巧是对每行数据产生 3 个副本，每个副本产生一个需要旋转的列，这个过程可以通过如下的 CROSS JOIN 来完成。

```
SELECT *
    FROM p,
    (SELECT 2009 AS orderyear
    UNION ALL SELECT 2010
    UNION ALL SELECT 2011) AS o
```

这里使用 UNION ALL 得到所有旋转列的值，将产生一个 3 行 1 列的表。再对表 p 执行，能为每行生成 3 个副本，其结果如表 6-17 所示。

表 6-17　为每行数据产生 3 个副本

custid	y2009	y2010	y2011	orderyear
A	10	40	20	2009
A	10	40	20	2010
A	10	40	20	2011
B	20	20	30	2009
B	20	20	30	2010
B	20	20	30	2011
C	0	0	40	2009
C	0	0	40	2010
C	0	0	40	2011
D	0	40	0	2009
D	0	40	0	2010
D	0	40	0	2011

接着问题就简单了，只需根据 orderyear 列来取得对应旋转列的值，例如：

```
CASE orderyear
   WHEN 2009 THEN y2009
     WHEN 2010 THEN y2010
     WHEN 2011 THEN y2011
END AS qty
```

因此这个 Unpivoting 问题的解决方案如下：

```
SELECT custid,orderyear,
    CASE orderyear
        WHEN 2009 THEN y2009
        WHEN 2010 THEN y2010
        WHEN 2011 THEN y2011
    END AS qty
    FROM p,
    (SELECT 2009 AS orderyear
    UNION ALL SELECT 2010
    UNION ALL SELECT 2011) AS o
```

若要得到和表 6-14 一样的表，则还需要过滤 qty 等于 0 的情况，因此最终的解决方案为：

```
SELECT custid,orderyear,qty
FROM (
    SELECT custid,orderyear,
    CASE orderyear
        WHEN 2009 THEN y2009
        WHEN 2010 THEN y2010
        WHEN 2011 THEN y2011
    END AS qty
    FROM p,
    (SELECT 2009 AS orderyear
    UNION ALL SELECT 2010
    UNION ALL SELECT 2011) AS o
    ) AS M
    WHERE qty <> 0;
```

6.6　CUBE 和 ROLLUP

MySQL 数据库支持 CUBE 和 ROLLUP 关键字，作为 GROUP BY 子句的选项，应用在对多个维度进行聚合的 OLAP 查询中。可以将 ROLLUP 看做 CUBE 的一种特殊情况。目前 MySQL 数据库仅支持 CUBE 关键字，而没有真正在数据库层面实现对 CUBE 的支持。但是可以通过 ROLLUP 来模拟 CUBE 的情况，只不过在性能上可能没有数据库原生支持更高效。

6.6.1　ROLLUP

ROLLUP 是根据维度在数据结果集中进行的聚合操作。假设用户需要对 *N* 个维度进行聚合查询操作，普通的 GROUP BY 语句需要 *N* 个查询和 *N* 次 GROUP BY 操作。而

ROLLUP 的优点是一次可以取得 N 次 GROUP BY 的结果，这样可以提高查询的效率，同时大大减少了网络的传输流量。

对 6.4.3 节中产生的表 t 进行如下的简单 ROLLUP 操作，可以得到如表 6-18 所示的结果。

```
SELECT
   YEAR(orderdate) AS YEAR,
   SUM(qty) AS SUM FROM t
GROUP BY YEAR(orderdate)
WITH ROLLUP;
```

表 6-18　对一个维度进行 ROLLUP 操作的结果

YEAR	SUM	YEAR	SUM
2009	30	2011	90
2010	100	NULL	220

上述的 ROLLUP 操作只对单个列，也就是一个维度进行聚合。这时产生的结果和 GROUP BY 操作产生的结果没有什么太大不同，唯一不同的是最后一行产生了（NULL，220）的结果，表示对所有 YEAR 的 SUM 再进行一次聚合，表示产生所有订单数量的总和。

对单个维度进行 ROLLUP 操作只是可以在最后得到聚合的数据，对比 GROUP BY 语句并没有非常大的优势。对多个维度进行 ROLLUP 才能体现出 ROLLUP 的优势，例如：

```
SELECT
   empid,custid,
   YEAR(orderdate) year,
   SUM(qty) sum
FROM t
GROUP BY empid,custid,YEAR(orderdate)
WITH ROLLUP;
```

上述 SQL 语句根据 empid、custid 和 YEAR（orderdate）3 列进行 GROUP BY 操作，即需要对 3 列进行层次的维度操作，得到的结果如表 6-19 所示。

表 6-19　对 3 个维度进行 ROLLUP 操作的结果

empid	custid	year	sum
1	A	2010	30
1	A	NULL	30
1	C	2011	40
1	C	NULL	40
1	NULL	NULL	70
2	A	2009	10
2	A	NULL	10
2	B	2010	20

（续）

empid	custid	year	sum
2	B	NULL	20
2	NULL	NULL	30
3	A	2010	10
3	A	NULL	10
3	B	2009	20
3	B	2011	30
3	B	NULL	50
3	D	2010	40
3	D	NULL	40
3	NULL	NULL	100
4	A	2011	20
4	A	NULL	20
4	NULL	NULL	20
NULL	NULL	NULL	220

输出的结果还是比较直观和容易理解的，（NULL，NULL，NULL）表示最后的聚合，（empid，custid，year）表示对这 3 列进行分组的聚合结果，（empid，custid，NULL）表示对（empid、custid）两列进行分组的聚合结果，（empid，NULL，NULL）表示仅对 empid 列进行分组的聚合结果。因此上述 ROLLUP 语句等同于下面的 SQL 语句：

```
SELECT empid,custid,YEAR(orderdate) year,SUM(qty) sum FROM t
GROUP BY empid,custid,YEAR(orderdate)
UNION
SELECT empid,custid,NULL,SUM(qty) AS SUM FROM t
GROUP BY empid,custid
UNION
SELECT empid,NULL,NULL,SUM(qty) FROM t
GROUP BY empid
UNION
SELECT NULL,NULL,NULL,SUM(qty) FROM t;
```

两者虽然可以得到相同的聚合结果，但是执行计划却完全不同。图 6-7 是通过 UNION 来实现 ROLLUP 操作的执行计划。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	PRIMARY	t	ALL	NULL	NULL	NULL	NULL	9	Using temporary; Using filesort
	2	UNION	t	ALL	NULL	NULL	NULL	NULL	9	Using temporary; Using filesort
	3	UNION	t	ALL	NULL	NULL	NULL	NULL	9	Using temporary; Using filesort
	4	UNION	t	ALL	NULL	NULL	NULL	NULL	9	
	NULL	UNION RESULT	<union1,2,3,4>	ALL	NULL	NULL	NULL	NULL	NULL	

图 6-7　通过 UNION 实现 ROLLUP 操作的执行计划

从图 6-7 中可以看出，SQL 执行器需要通过 4 次的表扫描操作来得到结果，然后再通过 UNION 来进行集合的合并操作。反观 ROLLUP，其执行计划如图 6-8 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	t	ALL	NULL	NULL	NULL	NULL	9	Using filesort

图 6-8　ROLLUP 的执行计划

ROLLUP 只需 1 次表扫描操作就能得到全部结果，因此 SQL 查询的效率在此得到了极大的提升。

在使用 ROLLUP 时需要注意以下几方面：

- ORDER BY
- LIMIT
- NULL

ORDER BY 语句一般用来完成排序操作，但是在含有 ROLLUP 的 SQL 语句中不能使用 ORDER BY 关键字，可见这两者为互斥的关键字。如果使用 ORDER BY，则会出现如下的错误提示：

```
Error Code: 1221. Incorrect usage of CUBE/ROLLUP and ORDER BY
```

在含有 ROLLUP 的 SQL 语句中可以使用 LIMIT，但是由于 ROLLUP 不能使用 ORDER BY 进行排序，因此 LIMIT 的结果阅读性差，在多数情况下无实际意义。

如果分组的列包含 NULL 值，那么 ROLLUP 的结果可能是不正确的，因为在 ROLLUP 中进行分组统计时值 NULL 具有特殊意义。因此在进行 ROLLUP 操作时，可以先将 NULL 值转换为一个不可能存在的值，或者没有特别含义的值，例如：

```
IFNULL(XXX, 0)
```

6.6.2　CUBE

ROLLUP 是 CUBE 的一种特殊情况。和 ROLLUP 操作一样，CUBE 也是一种对数据的聚合操作，但是 ROLLUP 只在层次上对数据进行聚合，而 CUBE 对所有的维度进行聚合。具有 N 个维度的列，CUBE 需要 2^N 次分组操作，而 ROLLUP 只需要 N 次分组操作。

目前 MySQL 数据库定义了 CUBE 关键字，但不支持 CUBE 操作。下面是在 MySQL 数据库中使用 CUBE 后出现的错误提示：

```
mysql>SELECT VERSION()\G;
*************************** 1. row ***************************
VERSION(): 5.6.3-m6
1 row in set (0.00 sec)

mysql> SELECT empid,custid,YEAR(orderdate),SUM(qty) FROM t
```

```
    -> GROUP BY empid,custid,YEAR(orderdate)
    -> WITH CUBE;
ERROR 1235 (42000): This version of MySQL doesn't yet support 'CUBE'
```

可以通过 ROLLUP 来模拟 CUBE，例如，上述 SQL 语句可重写为：

```
SELECT
   empid,custid,YEAR(orderdate),SUM(qty) FROM t
GROUP BY empid,custid,YEAR(orderdate)
WITH ROLLUP
UNION
SELECT
   empid,custid,YEAR(orderdate),SUM(qty) FROM t
GROUP BY empid,YEAR(orderdate),custid
WITH ROLLUP
UNION
SELECT
   empid,custid,YEAR(orderdate),SUM(qty) FROM t
GROUP BY custid,YEAR(orderdate),empid
WITH ROLLUP
UNION
SELECT
   empid,custid,YEAR(orderdate),SUM(qty) FROM t
GROUP BY custid,empid,YEAR(orderdate)
WITH ROLLUP
UNION
SELECT
   empid,custid,YEAR(orderdate),SUM(qty) FROM t
GROUP BY YEAR(orderdate),empid,custid
WITH ROLLUP
UNION
SELECT
   empid,custid,YEAR(orderdate),SUM(qty) FROM t
GROUP BY YEAR(orderdate),custid,empid
WITH ROLLUP;
```

6.7　小结

本章介绍了 MySQL 数据库中的聚合和旋转操作，其中使用了前几章已经涉及的一些 SQL 编程知识，同时又引入了新的解决方案，如附加属性聚合、Pivoting 和 Unpivoting。本章的最后还讲述了 MySQL 数据库中的 CUBE 和 ROLLUP 聚合。通过本章的介绍，希望读者可以深入浅出地掌握如何处理 MySQL 数据库的各类分组聚合问题。

第 7 章

游　　标

游标（cursor）是一种面向过程的SQL编程方法，与前面章节讨论的通过面向集合的方法处理关系数据库问题全然不同。本章将介绍MySQL数据库中的游标，分析游标的开销，并介绍在一些问题中正确合理地使用游标会较之使用面向集合的SQL编程在效率方面带来的极大提升。同时告诉不管有多少年SQL编程经验的开发人员或DBA，游标并不是恶魔，关键在于如何正确分析问题和合理使用游标。

7.1 面向集合与面向过程的开发

很多游标问题源于开发人员对SQL编程没有一个体系上的了解。最初，他们认为SQL和其掌握的其他编程语言如C、C++、Java一样，是一种面向过程的语言。因此，他们使用自己所熟悉的方法来编写SQL程序，游标成为了他们的首选。

慢慢地，他们发现游标的性能很差，一些有经验的SQL编程人员也会告诉他们，游标是恶魔，在程序中千万不要使用游标。因此，他们开始转向面向集合的开发，并发现面向集合的确能极大地提高SQL运行的速度。他们开始相信，游标是恶魔，并用面向集合的编程方法来解决所有的数据库问题，不再考虑任何使用游标的情况。

然而笔者相信：任何存在都是合理的。ANSI标准不会去定义一个毫无用处、性能低下、编程人员完全不会去使用的标准。同时，主流的数据库厂商也都对游标提供支持。另外，在笔者的经验中，有些问题使用游标反而能极大地提高程序运行的效率。

在第1章已经介绍了可将SQL编程分为三个阶段。在第二个阶段中，SQL程序员已经从最初的面向过程的开发转变到面向集合开发，并通过面向集合的方法解决了所遇到的大部分问题。但是在通向第三个阶段时，对于游标的认识可能需要经历否定之否定的阶段，最终达到一个新的认识高度，从而在面向过程与面向集合的思想上达到一个融会贯通的状态。当然，在第三个阶段SQL程序员也可能会用纯静态的SQL语言来完成所有的任务，他们还可能使用C、C++、Java等语言来获得更大的灵活性。

7.2 游标的使用

在MySQL数据库中，游标可以在存储过程和函数（Stored Routine）以及触发器（Trigger）和事件（Event）中使用。游标需要与handler一起使用，并且游标要在handler之前定义。游标有以下3个属性。

- Asensitive：数据库也可以选择不复制结果集。
- Read only：不可更新。
- Nonscrollable：游标只能向一个方向行进，并且不可以跳过任何一行数据。

要使用游标，先要定义一个游标变量：

```
DECLARE cursor_name CURSOR FOR select_statement
```

然后打开定义的游标变量：

```
OPEN cursor_name
```

接着从游标中取得数据：

```
FETCH cursor_name INTO var_name [, var_name] ...
```

最后关闭游标：

```
CLOSE cursor_name
```

一个使用游标的完整示例如下：

```
CREATE PROCEDURE cur_demo()
BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE _emp_no INT;
    DECLARE _dept_no VARCHAR(10);
    DECLARE cur1 CURSOR FOR SELECT emp_no,dept_no FROM dept_emp;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
    OPEN cur1;
    read_loop: LOOP
    FETCH cur1 INTO _emp_no,_dept_no;
    IF done THEN
        LEAVE read_loop;
    END IF;
    END LOOP;
    CLOSE cur1;
END;
```

上述示例只从游标中选取数据，并没有进行其他额外的操作。另外需要非常注意的是，在从游标中取得数据放入变量时，变量的名称不能和游标定义中的列名一样，否则选取出的值都为 NULL。因此在上述这个示例中，我们定义的变量名为 _dept_no、_emp_no，而在进行游标定义时 SELECT 中的列名为 dept_no、emp_no。

7.3 游标的开销

游标真的很慢吗？并非如此。游标最大的开销在于其需要对每一行数据进行处理，如果这个处理需要一定的时间，那么游标的操作可能非常慢。但是，如果游标中没有大量的操作，那么游标并不会这么慢。对于上一节的示例，其实游标执行得非常快，具体情况如下：

```
mysql> call cur_demo();
Query OK, 0 rows affected, 1 warning (0.51 sec)
```

可以看到上述存储过程执行了0.51秒，不是太坏的结果。可以用游标来模拟面向集合的方法，以此来观察游标的开销。先来模拟COUNT统计操作，使用的语句如下：

```
CREATE PROCEDURE cur_demo2()
BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE _emp_no INT;
    DECLARE _dept_no VARCHAR(10);
    DECLARE ret INT DEFAULT 0;
    DECLARE cur1 CURSOR FOR SELECT emp_no,dept_no FROM dept_emp;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
    OPEN cur1;
    read_loop: LOOP
    FETCH cur1 INTO _emp_no,_dept_no;
    IF done THEN
        LEAVE read_loop;
   SET ret = ret + 1;
    END IF;
    END LOOP;
    CLOSE cur1;
    SELECT ret;
END;
```

使用游标和直接使用SQL语句的时间对比如下：

```
mysql>call cur_demo2();
+--------+
| ret    |
+--------+
| 331603 |
+--------+
1 row in set (0.84 sec)

Query OK, 0 rows affected, 1 warning (0.84 sec)

mysql> select count(1) from dept_emp;
+----------+
| count(1) |
+----------+
|   331603 |
+----------+
1 row in set (0.13 sec)
```

游标的解决方案使用了0.84秒，虽然不是太慢，但是面向集合的SQL语句只使用了0.13秒。游标的解决方案需要的时间差不多是直接使用SQL语句的7倍。接着，通过使用游标来模拟SELECT操作：

```
CREATE TABLE ret ( emp_no INT, dept_no VARCHAR(10))ENGINE=MEMORY;
```

```
CREATE PROCEDURE cur_demo3()
BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE _emp_no INT;
    DECLARE _dept_no VARCHAR(10);
    DECLARE cur1 CURSOR FOR SELECT emp_no,dept_no FROM dept_emp;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
    OPEN cur1;
    read_loop: LOOP
    FETCH cur1 INTO _emp_no,_dept_no;
    INSERT INTO ret SELECT _emp_no,_dept_no;
    IF done THEN
        LEAVE read_loop;
    END IF;
    END LOOP;
    CLOSE cur1;
END;
```

这里先定义了一个 ret 表，用来存放游标得到的最后结果。其存储引擎的类型是 Memory，这与面向集合的方法是一样的。之后在存储过程 cur_demo2 中将游标每次取得的数据放入这张 ret 表中。运行该存储过程的情况如下：

```
mysql>call cur_demo3();
Query OK, 1 row affected (3.58 sec)
```

可以看到用了 3.58 秒。如果直接运行面向集合的 SQL 语句需要多少时间呢？如下所示：

```
mysql> SELECT emp_no,dept_no FROM dept_emp;
...
331603 rows in set (0.26 sec)
```

这里面向游标的解决方案所需的时间是面向集合的 SQL 方案的 13 倍。

通过上述两个例子可以总结出：游标的开销主要是因为需要对每行进行处理，处理过程越复杂，游标的效率越低。

7.4 使用游标解决问题

7.4.1 游标的性能分析

是否要使用游标，如何正确地使用游标，怎样使用游标来提高 SQL 编程的效率？这都不是一言两语就能讨论完的。就个人经验而言，对于面向集合的 SQL 语句，如果该 SQL 语句的扫描成本为 $O(N)$，那么使用游标不太可能带来性能上的巨大提升，因为游标需要额外的开销。但是对于某些扫描成本为 $O(N^2)$ 的问题，游标或许可以带来性能上的巨大提升。

对于某些问题，通过面向集合的方法不能一次扫描得到结果，但是通过面向过程的方

法，或许只需要一次扫描就能得到结果，这时游标可能成为一个很好的工具。我们假设一次扫描的成本为 $O(N)$，游标开销的放大倍数（scale factor）为 x，则使用游标解决方案的成本 P_C 为：

$$P_C = x*O(N) \tag{7-1}$$

若该问题使用面向集合的解决方案来解决，其扫描成本为 $O(N^2)$，则最终的扫描成本 P_S 为：

$$P_S = O(N^2) \tag{7-2}$$

因此，使用游标能带来 SQL 编程性能提高的前提为：

$$P_C << P_S$$

即从扫描成本上来看，使用游标的解决方案要远远小于使用集合的解决方案时，使用游标是非常有意义的。就之前讨论的行号问题来看，其面向集合的解决方案之一为：

```
SELECT col1,..., coln
( SELECT COUNT(*) FROM table AS T2
WHERE T2.empid <= T1.empid) AS rownnum
FROM table AS T1
```

我们先来分析这个解决方案的扫描成本。显然该方案的扫描成本为 $O(N^2)$，但是实际的扫描成本为 $1 + 2 + 3 + \cdots + n = 1/2*n*(n-1)$。然而行号问题显然可以通过一次扫描得到结果，因此若游标满足以下条件，则可以比面向集合的解决方案更有优势：

$$x*n << 1/2*n*(n-1)$$

我们假设游标开销的放大倍数为 10，上述条件具体化为：

$$10*n << 1/2*n*(n-1)$$

这是一个很简单的方程式，如果 n=5，那么两边就可以取等。也就是说，若扫描的表有 5 行数据，则两个解决方案的速度应该差不多。n 越大，采用游标解决方案的优势越明显。

对于不同的问题，其扫描成本可能都为 $O(N^2)$，但是在实际的生产环境中，根据每个问题的不同及索引的存在，其实际的扫描次数可能是不同的。例如对两个表根据各自的主键进行 INNER JOIN 操作，在没有索引的情况下，扫描次数可能为 N^2。但若存在索引，则扫描的次数可能为 $2N$ 或者 $3N$（取决于表的大小），这时扫描次数为 $10N$ 的游标解决方案可能并不是一个最优的解决方案。

7.4.2 连续聚合

连续聚合是第 6 章讨论的一个问题。连续聚合问题还可以细分为累积、滑动、年初至今问题。在上一章讨论的都是用面向集合的解决方案，在这里将讨论使用游标的解决方案。对于累积问题，可以通过下面的游标解决方案来解决。

```
CREATE PROCEDURE cur_aggredate()
BEGIN
```

```
 DECLARE done INT DEFAULT 0;
 DECLARE _empid INT DEFAULT -1;
 DECLARE _prev_empid INT DEFAULT -1;
 DECLARE _count INT DEFAULT 0;
 DECLARE _prev_count INT DEFAULT 0;
 DECLARE _ordermonth DATETIME;
 DECLARE _qty INT;
 DECLARE _total INT DEFAULT 0;
 DECLARE cur1 CURSOR FOR
SELECT empid,ordermonth,qty FROM EmpOrders
ORDER BY empid,ordermonth;
 DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

DROP TEMPORARY TABLE IF EXISTS ret;
 CREATE TEMPORARY TABLE ret(
 empid INT,
ordermonth DATETIME,
qty INT, total INT,
avg DOUBLE(5,2)
);

 OPEN cur1;
 READ_LOOP: LOOP
 FETCH cur1 INTO _empid,_ordermonth,_qty;
 IF done THEN
     LEAVE READ_LOOP;
 END IF;
 IF _empid <> _prev_empid && _prev_empid <> -1 THEN
     SET _prev_empid = _empid;
     SET _count = 1;
     SET _total = _qty;
     INSERT INTO ret SELECT
         _empid,_ordermonth,_qty,_total,_total/_count;
 ELSE
         IF _prev_empid = -1 THEN
             SET _prev_empid = _empid;
         END IF;
         SET _count = _count + 1;
         SET _total = _total + _qty;
         INSERT INTO ret SELECT
             _empid,_ordermonth,_qty,_total,_total/_count;
     END IF;
     END LOOP;
     CLOSE cur1;
     SELECT
    empid,
    DATE_FORMAT(ordermonth,"%Y-%m") AS ordermonth,
    qty,total,avg FROM ret;
     DROP TABLE ret;
END;
```

使用集合的解决方案如下：

```
SELECT
   a.empid,
   a.ordermonth,a.qty AS thismonth,
   SUM(b.qty) AS total,
   CAST(AVG(b.qty) AS DECIMAL(5,2)) AS avg
FROM EmpOrders a
INNER JOIN EmpOrders b
   ON a.empid = b.empid
   AND b.ordermonth <= a.ordermonth
GROUP BY a.empid,a.ordermonth,a.qty
ORDER BY a.empid,a.ordermonth;
```

基于游标的解决方案仅扫描表一次，这意味着该解决方案所用的时间根据表的行数呈线性变化。而基于集合的解决方案的扫描成本为 $O(N^2)$。我们假设一共有 x 个分组，每个分组有 n 行数据，因此一共需要 $x*1/2*n*(n-1)$ 次扫描，而这时基于集合的解决方案的扫描次数为 $x*n$。对于基于游标的解决方案的开销，如果放大倍数为 100，那么只要满足：

$$100*x*n << x*1/2*n*(n-1)$$

基于游标的解决方案就会非常有优势。从上面的公式可以发现，如果分组中每行的数据比较少，即 n 比较小，基于集合的解决方案就会比较有优势；如果分组中有较多的数据，则基于游标的解决方案更有优势。可以通过下列语句来扩大表 EmpOrders 中的数据，同时增大每个分组中的数据。

```
INSERT INTO EmpOrders
SELECT
   empid+(SELECT MAX(empid) FROM emporders ),
   ordermonth,
   qty
FROM EmpOrders;
```

表 7-1 显示了两种解决方案在数据的行数不同时的性能表现。

表 7-1 基于集合的解决方案和基于游标的解决方案的性能对比

行数	基于集合（秒）	基于游标（秒）	行数	基于集合（秒）	基于游标（秒）
200	0.015	0.031	50 000	1.264	0.842
1500	0.062	0.063	100 000	2.496	1.622
6000	0.156	0.124	700 000	47.346	13.245
12 000	0.327	0.266	1 500 000	273.111	26.13

从表 7-1 可以看到，当表的行数在 1500 时，基于游标的解决方案和基于集合的解决方案所需要的时间基本一样。随着表内数据的不断增加，基于游标的解决方案基本符合线性增长，而基于集合的解决方案则差了很多。当表内数据达到 150 万行时，基于集合的解决方案

需要的时间为基于游标的解决方案的 10 倍多。

7.4.3 最大会话数

在第 2 章介绍了重叠问题。重叠问题之一就是如何求得某一时间段内的最大会话数。下面先给出一个表 sessions，其内容如表 7-2 所示。

表 7-2 表 sessions 的内容

id	app	usr	starttime	endtime
1	app1	user1	08:30:00	10:30:00
2	app1	user2	08:30:00	08:45:00
3	app1	user1	09:00:00	09:30:00
4	app1	user2	09:15:00	10:30:00
5	app1	user1	09:15:00	09:30:00
6	app1	user2	10:30:00	14:30:00
7	app1	user1	10:45:00	11:30:00
8	app1	user2	11:00:00	12:30:00
9	app2	user1	08:30:00	08:45:00
10	app2	user2	09:00:00	09:30:00
11	app2	user1	11:45:00	12:00:00
12	app2	user2	12:30:00	14:00:00
13	app2	user1	12:45:00	13:30:00
14	app2	user2	13:00:00	14:00:00
15	app2	user1	14:00:00	16:30:00
16	app2	user2	15:30:00	17:00:00

之前介绍过通过面向集合的解决方案——子查询来求解最大会话数问题。该解决方案的思想为通过子查询找出某个 app 开始时之前有另外的 app，其开始运行时间在当前 app 之前，结束时间在当前 app 之后。故该解决方案的 SQL 语句如下，最后得到的结果如表 7-3 所示。

```
SELECT app,MAX(count) AS max
FROM (
  SELECT app,s,
  (
  SELECT COUNT(1) FROM sessions AS b
    WHERE a.app = b.app AND s>=starttime AND s<endtime
  ) AS count
  FROM (
  SELECT DISTINCT app,starttime AS s
  FROM
  sessions
  ) AS a
```

```
) AS bc
GROUP BY app ;
```

表 7-3 最大会话数结果

app	max	app	max
app1	4	app2	3

该解决方案的执行计划如图 7-1 所示。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	ALL	NULL	NULL	NULL	NULL	14	Using temporary; Using filesort
2	DERIVED	<derived4>	ALL	NULL	NULL	NULL	NULL	14	
4	DERIVED	sessions	index	NULL	idx_app_s_e	38	NULL	16	Using index
3	DEPENDENT SUBQUERY	b	ref	idx_app_usr_s_e_key,idx_app_s_e	idx_app_usr_s_e_key	32	a.app	1	Using where; Using index

图 7-1 基于集合的解决方案的执行计划

从执行计划中可以发现，该 SQL 语句先通过 SELECT DISTINCT app,starttime AS s FROM sessions 产生一张派生表 <derived 4>。然后该派生表与 sessions 表再进行相关子查询，求得派生表 <derived 2>，最后在派生表 <derived 2> 上进行排序分组，求得最终的结果。这里相关子查询部分的扫描成本同样为 $O(N^2)$。

那么这个问题可以通过游标来解决吗？初步分析并不能使用与之前相同的方法来执行基于游标的解决方案。不过有个小技巧可以使用，即在基于游标的解决方案中输入已排序的集合。这样根据每个 session 有两个状态 start 和 end，可以使用下面的查询来声明游标：

```
SELECT app,starttime AS time, 1 AS type
FROM sessions
UNION ALL
SELECT app,endtime, -1
FROM sessions
ORDER BY app,time,type DESC
```

该查询为每个会话按照开始时间和结束时间分别生成不同类型的事件，然后再进行 UNION ALL 的集合操作，得到的结果如表 7-4 所示。

表 7-4 根据应用程序、时间及类型排序后的游标结果

app	time	type
app1	08:30:00	1
app1	08:30:00	1
app1	08:45:00	−1
app1	09:00:00	1
app1	09:15:00	1
app1	09:15:00	1
app1	09:30:00	−1

（续）

app	time	type
app1	09:30:00	−1
app1	10:30:00	−1
app1	10:30:00	−1
app1	10:30:00	1
app1	10:45:00	1
app1	11:00:00	1
app1	11:30:00	−1
app1	12:30:00	−1
app1	14:30:00	−1
app2	08:30:00	1
app2	08:45:00	−1
app2	09:00:00	1
app2	09:30:00	−1
app2	11:45:00	1
app2	12:00:00	−1
app2	12:30:00	1
app2	12:45:00	1
app2	13:00:00	1
app2	13:30:00	−1
app2	14:00:00	−1
app2	14:00:00	−1
app2	14:00:00	1
app2	15:30:00	1
app2	16:30:00	−1
app2	17:00:00	−1

这样最大会话数问题可以转换为游标对获取每行时 SUM(type) 的最大值。故该问题的游标解决方案为：

```
CREATE PROCEDURE cur_max_sessions()
BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE _app VARCHAR(10);
    DECLARE _prev_app VARCHAR(10);
    DECLARE _time TIME;
    DECLARE _type INT;
    DECLARE _current INT;
    DECLARE _MAX INT;
    DECLARE cur1 CURSOR FOR
    SELECT app,starttime AS time, 1 AS type
```

```
FROM sessions
UNION ALL
SELECT app,endtime ,-1
FROM sessions
ORDER BY app,time,type DESC;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

DROP TEMPORARY TABLE IF EXISTS ret;
CREATE TEMPORARY TABLE ret(
    app VARCHAR(10),
    max INT
)ENGINE=MEMORY;

OPEN cur1;
FETCH cur1 INTO _app,_time,_type;

SET _prev_app = _app;
SET _current = 0;
SET _max = 0;

WHILE done = 0 DO
BEGIN
    IF _prev_app <> _app THEN
        INSERT INTO ret SELECT _prev_app,_max;
        SET _prev_app = _app;
        SET _max = 0;
        SET _current = 0;
    END IF;

    SET _current = _current + _type;
    IF _current > _max THEN
        SET _max = _current;
    END IF;

        FETCH cur1 INTO _app,_time,_type;
    END;
    END WHILE;

    IF _prev_app IS NOT NULL THEN
        INSERT INTO ret SELECT _prev_app,_max;
    END IF;

    SELECT * FROM ret;
    DROP TABLE ret;
END;
```

可以看出，基于游标的解决方案的扫描成本体现在集合的 UNION ALL 操作处，故扫描行的次数为 $2N$，扫描成本为 $O(N)$。当 sessions 表中含有大量数据时，相对于扫描成本为 $O(N^2)$ 的面向集合的解决方案来说，游标解决方案的执行效率会得到极大提升。

7.5 小结

本章从面向集合和面向过程两种不同的编程方法入手，对游标进行了全面而又系统的介绍。首先根据 MySQL 数据库对于游标的定义，简单介绍了游标的使用方法。之后根据不同的示例来显示游标的开销，从示例中可以发现，对表进行一次扫描，游标的开销要远大于面向集合的方法。但是游标的主要优势体现在，对于一些面向集合的解决方案所需扫描成本为 $O(N^2)$ 的情况，基于游标的解决方案可能只需 $O(N)$。在本章的最后对游标解决方案用于面向集合的解决方案的情况进行了讨论。在大数据量的情况下，基于游标的解决方案可以极大地提升程序执行的效率。对于游标的使用，读者一定要明白，任何存在都是合理的，关键看能否合理使用。

第 8 章

事务编程

事务（transaction）是数据库区别于文件系统的重要特性之一。在文件系统中，如果用户正在写文件，但是操作系统突然崩溃了，这个文件就很有可能被破坏。当然，有一些机制可以把文件恢复到某个时间点。不过，如果需要保证多个文件同步，这些文件系统可能就显得无能为力了。例如，当你更新两个文件时，更新完一个文件后，在更新完第二个文件之前系统重启了，你就会得到两个不同步的文件。

在 SQL 编程中，事务编程已然成为必不可少的一个组成部分。事务能保证数据库从一种一致状态转换为另一种一致状态。在数据库提交工作时，可以确保其要么对所有修改都已经保存，要么对所有修改操作都不保存。

不同的存储引擎对于事务的支持粒度完全不同，有的存储引擎甚至不支持事务。本章主要以 InnoDB 存储引擎为对象，因为 MySQL 的联机事务应用多使用此存储引擎，同时该引擎完全满足事务的 ACID 特性。

8.1 事务概述

事务可由一条非常简单的 SQL 语句组成，也可以由一组复杂的 SQL 语句组成。事务是访问并更新数据库中各种数据项的一个程序执行单元。在事务中的操作，要么都执行修改，要么都不执行，这就是事务的目的，也是事务模型区别于文件系统的重要特征之一。

从理论上说，事务有着极其严格的定义，它必须同时满足 4 个特性，即通常所说事务的 ACID 特性。值得注意的是，虽然理论上定义了严格的事务要求，但是数据库厂商出于各种目的并没有严格满足事务的 ACID 标准。例如，对于 MySQL 的 NDB Cluster 引擎来说，虽然其支持事务，但是不满足 D 的要求，即持久性的要求。对于 Oracle 数据库来说，其默认的事务隔离级别为 READ COMMITTED，不满足 I 的要求，即隔离性的要求。虽然在大多数情况下，这并不会导致严重的结果，甚至可能会带来性能的提升，但是用户首先需要了解严谨的事务标准，并在实际的生产应用中避免可能存在的潜在问题。对于 InnoDB 存储引擎而言，其默认的事务隔离级别为 READ REPEATABLE，完全遵循和满足事务的 ACID 特性。下面具体介绍事务的 ACID 特性，并给出相关概念。

- A（atomicity），原子性。在计算机系统中，每个人都将原子性视为理所当然。例如在 C 语言中调用 SQRT 函数，要么返回正确的平方根值，要么返回错误的代码，而不会在不可预知的情况下改变任何的数据结构和参数。如果 SQRT 函数被许多程序调用，那么一个程序的返回值也不会是其他程序要计算的平方根。

然而在数据的事务中实现调用操作的原子性，就不是那么理所当然了。例如一个用户在 ATM 机前取款，其取款的流程为：

1）登录 ATM 机平台，验证密码。

2）从远程银行的数据库中取得账户的信息。

3）用户在 ATM 机上输入欲提取的金额。

4）从远程银行的数据库中更新账户信息。

5）ATM 机出款。

6）用户取钱。

整个取款的操作过程应该视为原子操作。要么都做，要么都不做。不能出现用户钱未从 ATM 机上取得而银行卡上的钱已经被扣除的情况，相信任何人都不能接受。通过事务模型，可以保证该操作的原子性。

原子性指整个数据库事务是不可分割的工作单位。只有使事务中所有的数据库操作都执行成功，整个事务的执行才算成功。事务中任何一个 SQL 语句执行失败，那么已经执行成功的 SQL 语句也必须撤销，数据库状态应该退回到执行事务前的状态。

如果事务中的操作都是只读的，那么保持原子性很简单；如果发生任何错误，要么重试，要么返回错误代码，因为只读操作不会改变系统中的任何相关部分。但是，当事务中的操作需要改变系统中的状态，如插入或更新记录时，情况可能不像只读操作那么简单了；如果操作失败，很可能引起状态的变化，因此必须要防止系统中并发用户访问受影响的部分数据。

- C（consistency），一致性。一致性指事务将数据库从一种状态转变为另一种一致的状态。在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。例如，在表中有一个字段为姓名，它是一个唯一约束，即在表中姓名不能重复。如果一个事务对表进行了修改，但是在事务提交或当事务操作发生回滚后，表中的数据姓名变得非唯一了，那么就破坏了事务的一致性要求，即事务将数据库从一种状态变为了一种不一致的状态。因此，事务是一致性的单位，如果事务中某个动作失败了，系统可以自动地撤销事务使其返回初始化的状态。
- I（isolation），隔离性。隔离性还有其他的称呼，如并发控制（concurrency control）、可串行化（serializability）、锁（locking）。事务的隔离性要求每个读写事务的对象与其他事务的操作对象能相互分离，即该事务提交前对其他事务都不可见，这通常使用锁来实现。当前数据库系统中都提供了一种粒度锁（granular lock）的策略，允许事务仅锁住一个实体对象的子集，以此来提高事务之间的并发度。
- D（durability），持久性。事务一旦提交，其结果就是永久性的，即使发生宕机等故障，数据库也能将数据恢复。需要注意的是，持久性只能从事务本身的角度来保证结果的永久性，如事务提交后，所有的变化都是永久的，即使当数据库由于崩溃而需要恢复时，也能保证恢复后提交的数据都不会丢失。但如果不是数据库本身发生故障，而是一些外部的原因，如 RAID 卡损坏、自然灾害等导致数据库发生问题，那么所有提交的数据可能会丢失。因此持久性保证的是事务系统的高可靠性（high reliability），而不是高可用性（high availability）。对于高可用性的实现，事务本身并不能保证，需要一些系统共同配合来完成。

8.2 事务的分类

从理论的角度来说，可以把事务分为以下几种类型：

- 扁平事务（flat transactions）。
- 带有保存点的扁平事务（flat transactions with savepoints）。
- 链事务（chained transactions）。
- 嵌套事务（nested transactions）。
- 分布式事务（distributed transactions）。

扁平事务是事务类型中最简单的一种，而在实际生产环境中，这可能是使用最为频繁的事务。在扁平事务中，所有操作都处于同一层次，其由 BEGIN WORK 开始，由 COMMIT WORK 或 ROLLBACK WORK 结束。处于之间的操作是原子的，要么都执行，要么都回滚。因此，扁平事务是应用程序成为原子操作的基本组成模块。图 8-1 显示了扁平事务的三种不同情况。

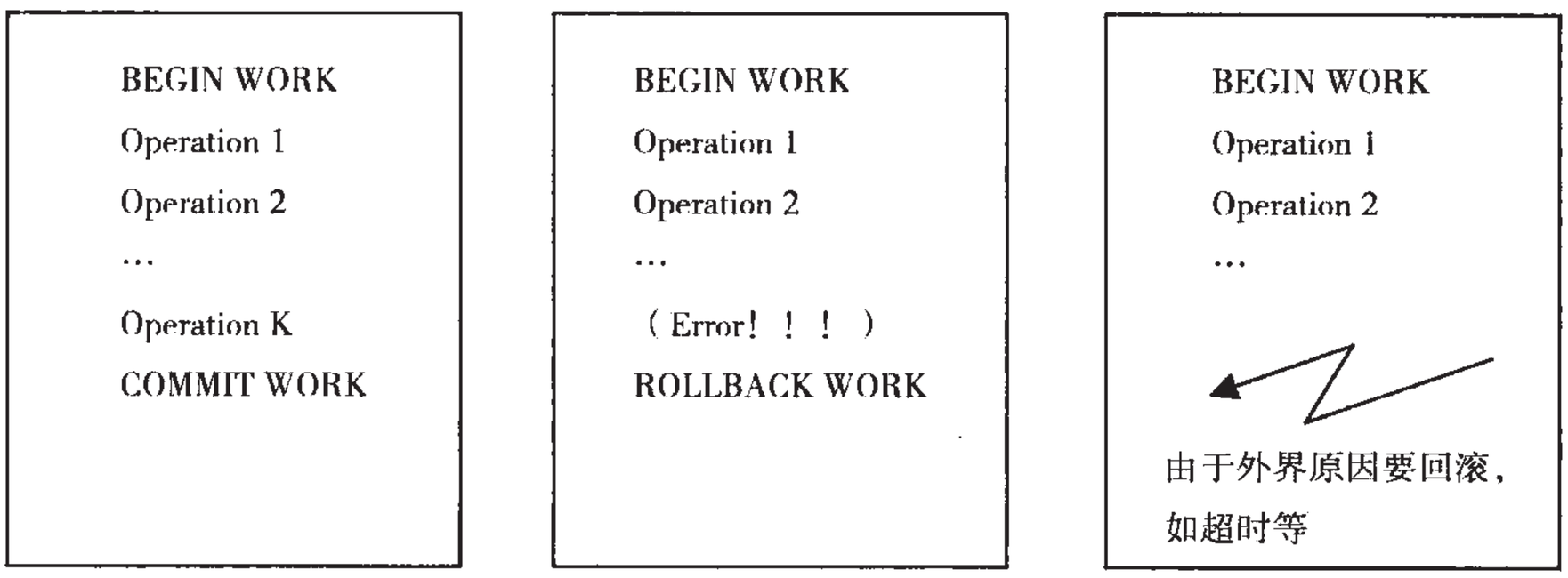


图 8-1　扁平事务的三种情况

图 8-1 给出了扁平事务的三种情况，同时也给出了在一个典型的事务处理应用中，每个结果大概占用的百分比。这里再次提醒，扁平事务虽然简单，但是在实际生产环境中使用最为频繁。正因为其简单、使用频繁，所以每个数据库系统都实现了对扁平事务的支持。

扁平事务的主要限制是不能提交或回滚事务的某一部分，或分几个步骤提交。下面来看一个扁平事务不足以支持的例子。用户在旅行网站上制订自己的旅行度假计划，他设想从杭州到意大利的佛罗伦萨，这两个城市之间没有直达的班机，需要用户预订并转乘航班或者火车。用户预订旅行度假的事务为：

```
BEGIN WORK
S1：预订杭州到上海的高铁
```

S2：上海浦东国际机场坐飞机，预订去米兰的航班
S3：在米兰转火车前往佛罗伦萨，预订去佛罗伦萨的火车

但是当用户执行到S3时，发现由于飞机到达米兰的时间太晚，当天已经没有开往佛罗伦萨的火车。这时用户希望在米兰当地住一晚，第二天出发去佛罗伦萨。这时如果事务为扁平事务，则需要回滚之前的S1、S2、S3三个操作，这个代价显然有点大。因为当再次进行该事务时，S1、S2的执行计划是不变的。也就是说，如果支持有计划的回滚操作，那么就不需要终止整个事务，因此就出现了带有保存点的扁平事务。

带有保存点的扁平事务，除了支持扁平事务支持的操作外，允许在事务执行过程中回滚到同一事务中较早的一个状态，这是因为可能某些事务在执行过程中出现的错误并不会对所有的操作都无效，放弃整个事务不合乎要求，开销也太大。保存点（savepoint）用来通知系统应该记住事务当前的状态，以便以后发生错误时，事务能回到该状态。

对于扁平事务来说，其隐式地设置了一个保存点，但是在整个事务中，只有这一个保存点，回滚只能回滚到事务开始时的状态。保存点用SAVE WORK函数来建立，通知系统记录当前的处理状态。当出现问题时，保存点能用做内部的重启动点，根据应用逻辑，决定是回到最近一个保存点还是其他更早的保存点。图8-2显示了在事务中使用保存点。

图8-2显示了如何在事务中使用保存点。灰色背景部分的操作表示由ROLLBACK WORK而导致部分回滚，是实际并没有执行的操作。当用BEGIN WORK开启一个事务时，隐式地包含了一个保存点，当事务通过ROLLBACK WORK：2发出部分回滚命令时，事务回滚到保存点2，接着依次执行，并再次执行到ROLLBACK WORK：7，直到执行最后的COMMIT WORK操作，这时表示事务结束，除灰色阴影部分的操作外，其余操作都已经执行并提交。

另一点需要注意的是，保存点在事务内部是递增的，这从图8-2中也能看出来。有人可能会想，返回保存点2以后，下一个保存点可能为3，因为之前的工作都终止了。然而新的保存点编号为5，这意味着回滚不影响保存点的计数，并且单调地址的编号能保持事务执行的整个历史过程，包括在执行过程中想法的改变。

此外，当事务通过ROLLBACK WORK：2命令发出部分回滚命令时，要记住事务并没有完全被回滚，只是回滚到了保存点2而已。这代表当前事务还是活跃的，如果想要完全回滚事务，还需要再执行命令ROLLBACK WORK。

链事务可视为保存点模式的一个变种。带有保存点的扁平事务，当发生系统崩溃时，所有的保存点都将消失，因为其保存点是易失的（volatile），而非持久的（persistent）。这意味着当恢复保存点时，事务需要从开始处重新执行，而不能从最近的一个保存点继续执行。

链事务的思想是：在提交一个事务时，释放不需要的数据对象，将必要的处理上下文隐式地传给下一个要开始的事务。注意，提交事务操作和开始下一个事务操作将合并为一个原子操作。这意味着下一个事务将看到上一个事务的结果，就好像在一个事务中进行的。

图 8-3 显示了链事务的工作方式。

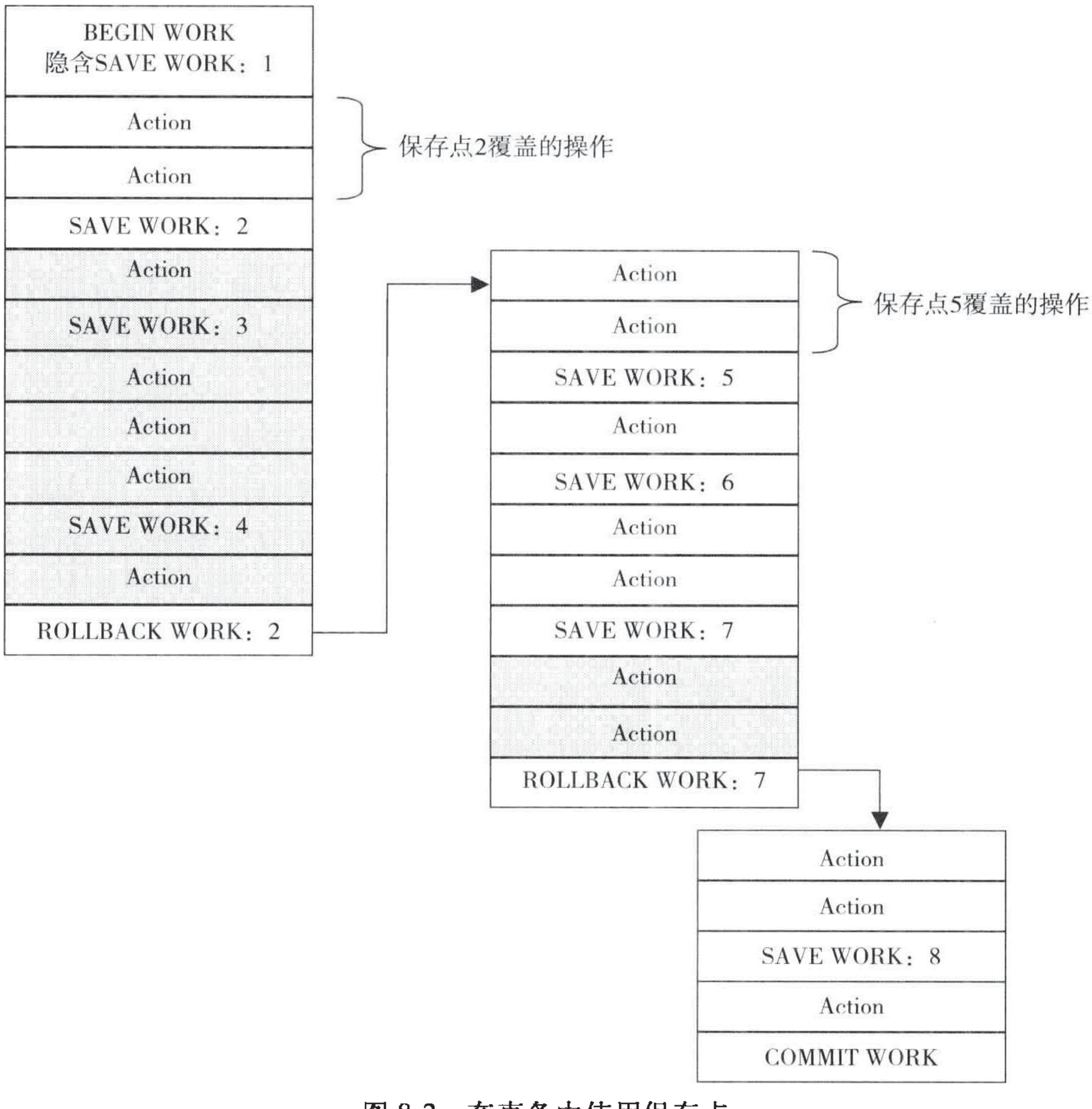


图 8-2 在事务中使用保存点

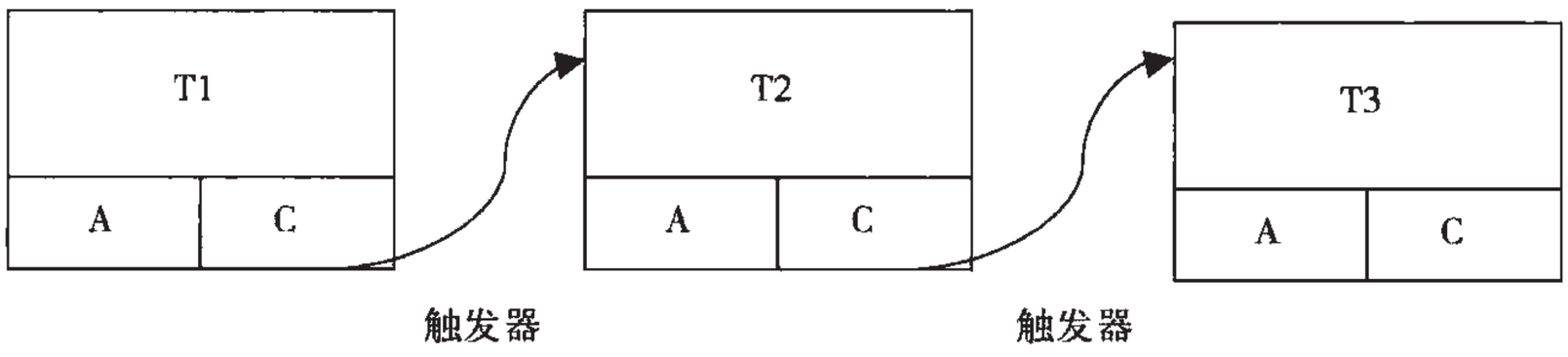


图 8-3 链事务的工作方式

链事务与带有保存点的扁平事务不同的是，带有保存点的扁平事务能回滚到任意正确的保存点。而链事务中的回滚仅限于当前事务，即只能恢复到最近一个保存点。对于锁的处理，两者也不相同。链事务在 COMMIT 后即释放了当前事务所持有的锁，而带有保存点的

扁平事务不影响迄今为止所持有的锁。

嵌套事务是一个层次结构框架。有一个顶层事务（top-level transaction）控制着各个层次的事务。顶层事务之下嵌套的事务被称为子事务（subtransaction），其控制每一个局部的变换。嵌套事务的层次结构如图 8-4 所示。

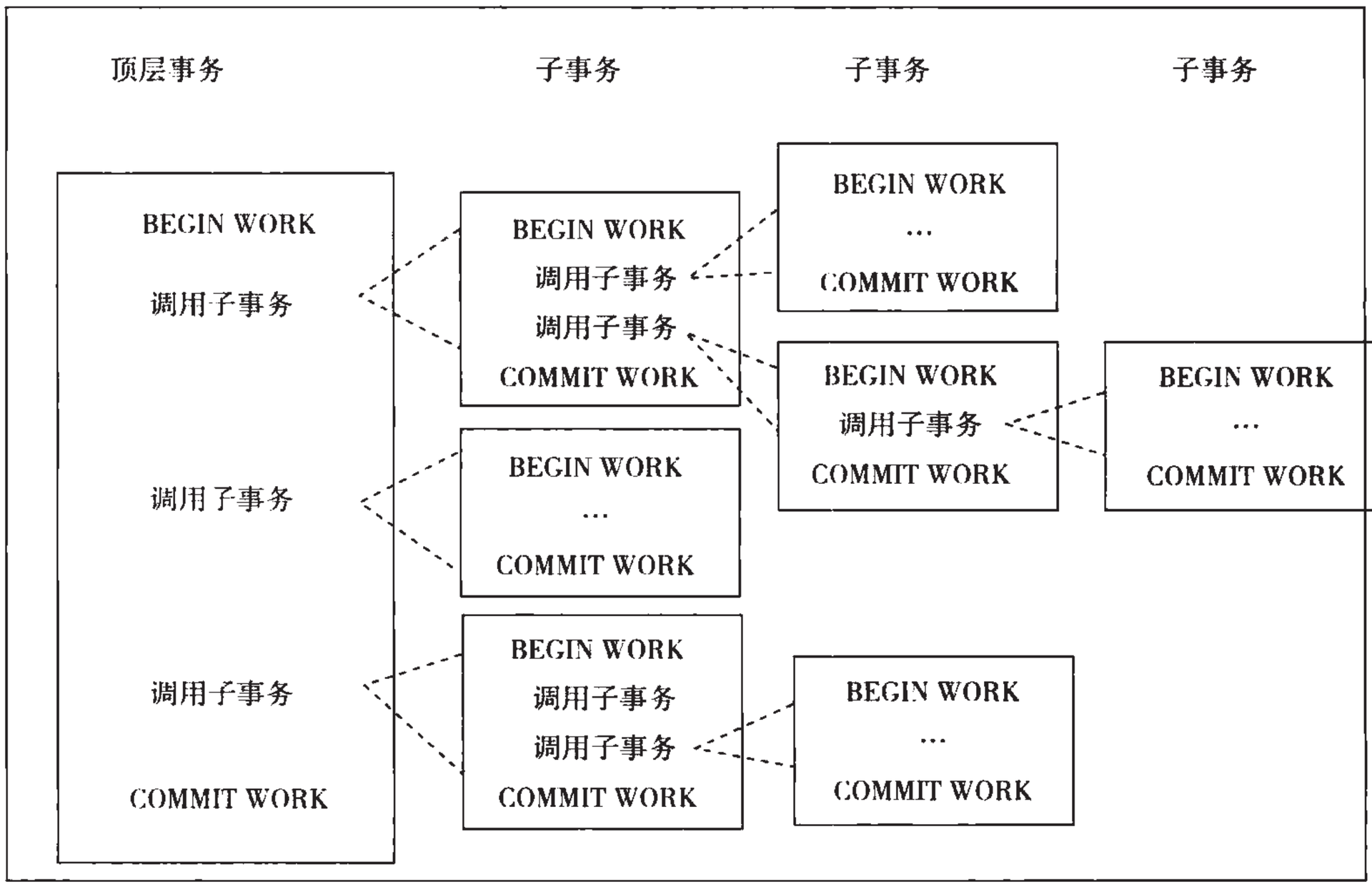


图 8-4 嵌套事务的层次结构

下面给出 Moss 对嵌套事务的定义：

- 嵌套事务是由若干事务组成的一棵树，子树既可以是嵌套事务，也可以是扁平事务。
- 处在叶节点的事务是扁平事务，但是每个子事务从根到叶节点的距离可以是不同的。
- 位于根节点的事务称为顶层事务，其他事务称为子事务。事务的前驱（predecessor）称为父事务（parent），事务的下一层称为儿子事务（child）。
- 子事务既可以提交也可以回滚，但是它的提交操作并不马上生效，除非由其父事务提交。因此可以推论出，任何子事务都在顶层事务提交后才真正提交。
- 树中的任意一个事务的回滚会引起它的所有子事务一同回滚。故子事务仅保留 A、C、I 特性，不具有 D 特性。

在 Moss 的理论中，实际的工作交由叶节点来完成，即只有叶节点的事务才能访问数据库、发送消息、获取其他类型的资源。而高层的事务仅负责逻辑控制，决定何时调用相关的子事务。即使一个系统不支持嵌套事务，用户也可以通过保存点技术来模拟嵌套事务，如图

8-5 所示。

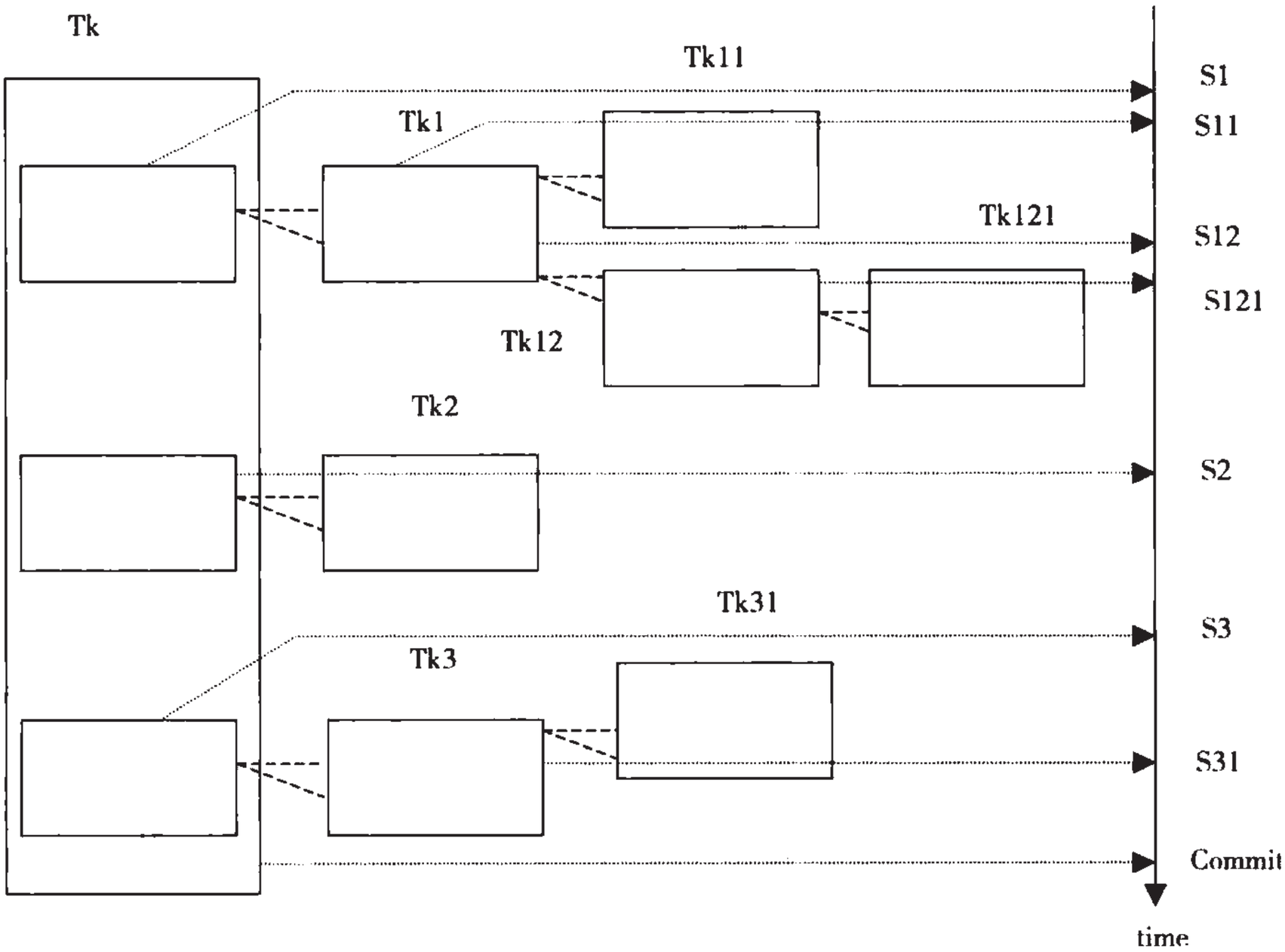


图 8-5　用保存点技术来模拟嵌套事务

从图 8-5 中也可以发现，保存点技术在恢复时比嵌套查询有更大的灵活性。例如在完成 Tk3 事务时，可以回滚到保存点 S2 的状态。而在嵌套查询的层次结构中，这是不允许的。

然而用保存点技术来模拟嵌套事务在锁的持有方面还是有些不同的。当通过保存点技术来模拟嵌套事务时，用户无法选择哪些锁需要被子事务继承、哪些需要被父事务保留。这就是说，无论有多少个保存点，所有被锁住的对象都可以被得到和访问。而在嵌套查询中，不同的子事务在数据库对象上持有的锁是不同的。例如有一个父事务 P_1，其持有对象 X 和 Y 的排他锁，现在要开始一个调用子事务 P_{11}，该父事务 P_1 可以不传递锁，也可以传递所有的锁，还可以只传递一个排他锁。如果子事务 P_{11} 还持有对象 Z 的排他锁，那么通过反向继承（counter-inherited）父事务 P_1 将持有 3 个对象 X、Y、Z 的排他锁。如果这时又再次调用了一个子事务 P_{12}，那么它可以选择传递已经持有的锁。

如果系统支持在嵌套事务中并行地执行各个子事务，那么在这种情况下用保存点的扁平事务来模拟嵌套事务就不切实际了。这从另一个方面反映出，要实现事务间的并行性需要真正支持嵌套事务。

分布式事务通常是一个在分布式环境下运行的扁平事务，因此需要根据数据所在位置访问网络中的不同节点。

假设一个用户在 ATM 机前进行银行的转账操作，要从招商银行的储蓄卡转账 10 000 元到工商银行的储蓄卡。在这种情况下，可以将 ATM 机视为节点 A，招商银行的后台数据库视为节点 B，工商银行的后台数据库视为节点 C，这个转账的操作可分解为以下步骤：

1）节点 A 发出转账命令。

2）节点 B 执行从储蓄卡中将余额值减去 10 000。

3）节点 C 执行从储蓄卡中将余额值加上 10 000。

4）节点 A 通知用户操作完成或失败。

这里需要使用分布式事务，因为节点 A 不能通过调用一台数据库就完成任务。这个过程需要访问网络中两个节点的数据库，而在每个节点的数据库中执行的事务操作又都是扁平的。对于分布式事务，同样需要满足 ACID 特性，要么都发生，要么都失效。对于上述例子，如果 2)、3）步骤中任何一个操作失败，都会导致整个分布式事务回滚。若不是这样，那结果会非常可怕。

对于 MySQL 数据库（InnoDB 存储引擎）来说，其支持扁平事务、带有保存点的扁平事务、链事务、分布式事务。对于嵌套事务，MySQL 数据库并不是原生的，因此对于有并行事务需求的用户来说 MySQL 就无能为力了，但是用户可以通过带有保存点的事务来模拟串行的嵌套事务。

8.3 事务控制语句

在 MySQL 命令行的默认设置下，事务都是自动提交（auto commit）的，即执行 SQL 语句后就会马上执行 COMMIT 操作。因此要显式地开启一个事务须使用命令 BEGIN 和 START TRANSACTION，或者执行命令 SET AUTOCOMMIT=0，以禁用当前会话的自动提交。每个数据库厂商对自动提交的设置都不相同，每个 DBA 或开发人员需要非常明白这对之后的 SQL 编程会有很大的不同，因此用户不能以之前的经验来判断 MySQL 数据库的运行方式。在具体介绍事务控制含义之前，先来看看用户可以使用哪些事务控制语句。

- START TRANSACTION | BEGIN：显式地开启一个事务。
- COMMIT：要想使用这个语句的最简形式，只需发出 COMMIT。也可以更详细一些，写为 COMMIT WORK，不过这二者几乎是等价的。COMMIT 会提交事务，并使已对数据库进行的所有修改成为永久性的。
- ROLLBACK：要使用这个语句的最简形式，只需发出 ROLLBACK。同样地，也可以写为 ROLLBACK WORK，但是二者几乎是等价的。回滚会结束用户的事务，并撤销正在进行的所有未提交的修改。
- SAVEPOINT identifier：SAVEPOINT 允许在事务中创建一个保存点，一个事务中可以有多个 SAVEPOINT。

- RELEASE SAVEPOINT identifier：删除一个事务的保存点，当没有一个保存点执行这句语句时，会抛出一个异常。
- ROLLBACK TO [SAVEPOINT] identifier ：这个语句与 SAVEPOINT 命令一起使用。可以把事务回滚到标记点，而不回滚在此标记点之前的任何工作。例如可以发出两条 UPDATE 语句，后面跟一个 SAVEPOINT，然后又是两条 DELETE 语句。如果执行 DELETE 语句期间出现了某种异常情况，而且捕获到这个异常，并发出 ROLLBACK TO SAVEPOINT 命令，事务就会回滚到指定的 SAVEPOINT，撤销 DELETE 完成的所有工作，而 UPDATE 语句完成的工作不受影响。
- SET TRANSACTION ：这个语句用来设置事务的隔离级别。InnoDB 存储引擎提供的事务隔离级别有 READ UNCOMMITTED、READ COMMITTED、REPEATABLE READ 和 SERIALIZABLE。

START TRANSACTION 和 BEGIN 语句都可以在 MySQL 命令行下显式地开启一个事务。但是在存储过程中，MySQL 数据库的分析器会自动将 BEGIN 识别为 BEGIN ... END，因此在存储过程中只能使用 START TRANSACTION 语句来开启一个事务。

COMMIT 和 COMMIT WORK 语句基本上是一致的，都用来提交事务。不同之处在于 COMMIT WORK 用来控制事务结束后的行为是 CHAIN 还是 RELEASE 的。如果是 CHAIN 方式，那么事务就变成了链事务。用户可以通过参数 completion_type 来进行控制，默认该参数为 0，表示没有任何操作。在这种设置下，COMMIT 和 COMMIT WORK 是完全等价的。当参数 completion_type 的值为 1 时，COMMIT WORK 等同于 COMMIT AND CHAIN，表示马上自动开启一个相同隔离级别的事务，如：

```
mysql> CREATE TABLE t ( a INT,PRIMARY KEY (a))ENGINE=INNODB;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @@autocommit\G;
*************************** 1. row ***************************
@@autocommit: 1
1 row in set (0.00 sec)

mysql> SET @@completion_type=1;
Query OK, 0 rows affected (0.00 sec)

mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t SELECT 1;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> COMMIT WORK;
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> INSERT INTO t SELECT 2;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> INSERT INTO t SELECT 2;
ERROR 1062 (23000): Duplicate entry '2' for key 'PRIMARY'

mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)

# 注意回滚之后只有1个记录，而没有2这个记录
mysql> SELECT * FROM t\G;
*************************** 1. row ***************************
a: 1
1 row in set (0.00 sec)
```

在这个实验中，我们设置completion_type为1，第一次通过COMMIT WORK来插入1这个记录。之后插入记录2时我们并没有使用BEGIN（或者START TRANSACTION）来显式地开启一个事务，之后再插入一条重复的记录2，这时会抛出异常。接着我们执行ROLLBACK操作，最后发现只有1这一条记录，2这条记录并没有被插入。因为completion_type为1时，COMMIT WORK会自动开启一个链事务，第二条 `INSERT INTO t SELECT 2` 语句是在同一个事务内的，因此回滚后2这条记录并没有被插入到表t中。

参数completion_type为2时，COMMIT WORK等同于COMMIT AND RELEASE。当事务提交后会自动断开与服务器的连接，如：

```
mysql> SET @@completion_type=2;
Query OK, 0 rows affected (0.00 sec)

mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t SELECT 3;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> COMMIT WORK;
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT @@version\G;
ERROR 2006 (HY000): MySQL server has gone away
No connection. Trying to reconnect...
Connection id:    54
Current database: test

*************************** 1. row ***************************
@@version: 5.1.45-log
```

```
1 row in set (0.00 sec)
```

通过上面的实验可以发现，当参数 completion_type 设置为 2 时，执行 COMMIT WORK 后再执行语句 SELECT @@version 会出现 ERROR 2006 (HY000): MySQL server has gone away 的错误。导致抛出该异常的原因是当前会话已经在上次执行 COMMIT WORK 语句后与服务器断开了连接。

ROLLBACK 和 ROLLBACK WORK 与 COMMIT 和 COMMIT WORK 的工作一样，这里不再赘述。

SAVEPOINT 记录了一个保存点，可以通过 ROLLBACK TO SAVEPOINT 回滚到某个保存点，但是如果回滚到一个不存在的保存点，会抛出异常，例如：

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> ROLLBACK TO SAVEPOINT t1;
ERROR 1305 (42000): SAVEPOINT t1 does not exist
```

InnoDB 存储引擎中的事务都是原子的，这说明下述两种情况：构成事务的每条语句都会提交（成为永久），或者所有语句都回滚。这种保护还延伸到单个语句中。一条语句要么完全成功提交，要么完全回滚（注意，这里说的是语句回滚）。因此一条语句失败并抛出异常并不会导致先前已经执行的语句自动回滚，它们的工作会得到保留，必须由用户自己来决定是否对其进行提交或回滚操作，如：

```
mysql> CREATE TABLE t ( a INT,PRIMARY KEY(a))ENGINE=INNODB;
Query OK, 0 rows affected (0.00 sec)

mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t SELECT 1;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> INSERT INTO t SELECT 1;
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'

mysql> SELECT * FROM t\G;
*************************** 1. row ***************************
a: 1
1 row in set (0.00 sec)
```

可以看到，插入第二条记录 1 时，因为重复的关系抛出了 1062 的错误，但是数据库并没有进行自动回滚，这时事务仍需要用户显式地来运行 COMMIT 或 ROLLBACK 命令。

另一个容易犯的错误是 ROLLBACK TO SAVEPOINT，虽然有 ROLLBACK，但是它并

没有真正地结束一个事务，因此即使执行了 ROLLBACK TO SAVEPOINT，之后也需要显式地运行 COMMIT 或 ROLLBACK 命令，例如：

```
mysql> CREATE TABLE t ( a INT,PRIMARY KEY(a))ENGINE=INNODB;
Query OK, 0 rows affected (0.00 sec)

mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t SELECT 1;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SAVEPOINT t1;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t SELECT 2;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SAVEPOINT t2;
Query OK, 0 rows affected (0.00 sec)

mysql> RELEASE SAVEPOINT t1;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t SELECT 2;
ERROR 1062 (23000): Duplicate entry '2' for key 'PRIMARY'

mysql> ROLLBACK TO SAVEPOINT t2;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM t;
+---+
| a |
+---+
| 1 |
| 2 |
+---+
2 rows in set (0.00 sec)

mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM t;
Empty set (0.00 sec)
```

从上面的例子中可以看到，虽然在发生重复错误后，用户通过 ROLLBACK TO SAVEPOINT t2 命令回滚到了保存点 t2，但是事务此时没有结束。再运行 ROLLBACK 命

令后，事务才会完整地回滚。这里再次提醒，ROLLBACK TO SAVEPOINT 命令并不真正地结束事务。

8.4 隐式提交的 SQL 语句

以下这些 SQL 语句会产生一个隐式的提交操作，即执行完这些语句后，会有一个隐式的 COMMIT 操作。

- DDL 语句：ALTER DATABASE ... UPGRADE DATA DIRECTORY NAME、ALTER EVENT、ALTER PROCEDURE、ALTER TABLE、ALTER VIEW、CREATE DATABASE、CREATE EVENT、CREATE INDEX、CREATE PROCEDURE、CREATE TABLE、CREATE TRIGGER、CREATE VIEW、DROP DATABASE、DROP EVENT、DROP INDEX、DROP PROCEDURE、DROP TABLE、DROP TRIGGER、DROP VIEW、RENAME TABLE 和 TRUNCATE TABLE。
- 用来隐式地修改 MySQL 架构的操作：CREATE USER、DROP USER、GRANT、RENAME USER、REVOKE 和 SET PASSWORD。
- 管理语句：ANALYZE TABLE、CACHE INDEX、CHECK TABLE、LOAD INDEX INTO CACHE、OPTIMIZE TABLE 和 REPAIR TABLE。

注意 Microsoft SQL Server 的数据库管理员或开发人员往往忽视对于 DDL 语句的隐式提交操作，因为在 Microsoft SQL Server 数据库中，即使是 DDL 也是可以回滚的，这和 InnoDB 存储引擎、Oracle 这些数据库完全不同。

另外需要注意的是，TRUNCATE TABLE 语句是 DDL，因此虽然和对整张表执行 DELETE 的结果是一样的，但它不能被回滚（这又是和 Microsoft SQL Server 数据库不同的地方）。例如：

```
mysql> SELECT * FRM t\G;
*************************** 1. row ***************************
a: 1
*************************** 2. row ***************************
a: 2
2 rows in set (0.00 sec)

mysql> BEGIN;
Query OK, 0 rows affected (0.01 sec)

mysql> TRUNCATE TABLE t;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM t;
Empty set (0.00 sec)
```

8.5 事务的隔离级别

令人惊讶的是，大部分数据库系统都没有提供真正的隔离性，最初或许是因为系统实现者并没有真正理解这些问题，如今这些问题已经弄清楚了，但是数据库实现者在正确性和性能之间做了妥协。虽然ISO和ANSI SQL标准制定了四种事务隔离级别的标准，但是很少有数据库厂商遵循这些标准。比如Oracle数据库就不支持READ UNCOMMITTED和REPEATABLE READ的事务隔离级别。

ANSI SQL标准定义的四个隔离级别为：

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

READ UNCOMMITTED被称为浏览访问（browse access），仅仅只是对事务而言的。READ COMMITTED被称为游标稳定（cursor stability）。REPEATABLE READ是2.9999°的隔离，没有幻读的保护。SERIALIZABLE被称为隔离，或3°的隔离。SQL和SQL2标准的默认事务隔离级别是SERIALIZABLE。

InnoDB存储引擎默认的支持隔离级别是REPEATABLE READ，但是与标准SQL不同的是，InnoDB存储引擎在REPEATABLE READ事务隔离级别下，使用Next-Key Lock的锁算法，因此避免了幻读的产生。这与其他数据库系统（如Microsoft SQL Server数据库）是不同的。所以说，InnoDB存储引擎在默认的REPEATABLE READ事务隔离级别下已经能完全保证事务的隔离性要求，即达到SQL标准的SERIALIZABLE隔离级别。

隔离级别越低，事务请求的锁越少或保持锁的时间就越短。这也是为什么大多数数据库系统默认的事务隔离级别是READ COMMITTED的原因。

大部分的用户会质疑SERIALIZABLE隔离级别带来的性能问题，但是根据Jim Gray的研究，两者的开销几乎是一样的，因此在InnoDB存储引擎中选择REPEATABLE READ的事务隔离级别并不会有任何性能上的损失。同样地，即使使用READ COMMITTED隔离级别，用户也不会得到性能的大幅提升。

在InnoDB存储引擎中，可以使用以下命令来设置当前会话或全局的事务隔离级别。

```
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL
```

```
{
READ UNCOMMITTED
| READ COMMITTED
| REPEATABLE READ
| SERIALIZABLE
}
```

如果想在 MySQL 库启动时就设置事务的默认隔离级别，那就需要修改 MySQL 的配置文件，在 [mysqld] 中添加如下行：

```
[mysqld]
transaction-isolation = READ-COMMITTED
```

查看当前会话的事务隔离级别，可以使用：

```
mysql> SELECT @@tx_isolation\G;
*************************** 1. row ***************************
@@tx_isolation: REPEATABLE-READ
1 row in set (0.01 sec)
```

查看全局的事务隔离级别，可以使用：

```
mysql> SELECT @@global.tx_isolation\G;
*************************** 1. row ***************************
@@global.tx_isolation: REPEATABLE-READ
1 row in set (0.00 sec)
```

在 SERIALIZABLE 的事务隔离级别，InnoDB 存储引擎会对每个 SELECT 语句后自动加上 LOCK IN SHARE MODE，即给每个读取操作加一个共享锁。在这个事务隔离级别下，读占用锁了，因此对于一致性的非锁定读不再予以支持。由于 InnoDB 存储引擎在 REPEATABLE READ 隔离级别下就可以达到 3° 的隔离，因此一般不在本地事务中使用 SERIALIZABLE 隔离级别，SERIALIZABLE 事务隔离级别主要用于 InnoDB 存储引擎的分布式事务。

在 READ COMMITTED 事务隔离级别下，除了唯一性的约束检查及外键约束的检查需要 Gap Lock，InnoDB 存储引擎不会使用 Gap Lock 的锁算法。但是使用这个事务隔离级别需要注意一些问题。首先，在 MySQL 5.1 中，READ COMMITTED 事务隔离级别默认只能工作在 Replication（复制）的二进制日志为 ROW 格式下。如果二进制日志工作在默认的 STATEMENT 下，则会出现如下的错误：

```
mysql> CREATE TABLE a (
    -> b INT,PRIMARY KEY(b)
    -> )ENGINE=INNODB;
Query OK, 0 rows affected (0.01 sec)

mysql> SET @@tx_isolation='read-committed';
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SELECT @@tx_isolation\G;
*************************** 1. row ***************************
@@tx_isolation: REPEATABLE-READ
1 row in set (0.00 sec)

mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO a SELECT 1;
ERROR 1598 (HY000): Binary logging not possible. Message: Transaction level 'READ-
    COMMITTED' in InnoDB is not safe for binlog mode 'STATEMENT'
```

MySQL 5.0之前的版本不支持ROW格式的二进制日志，有人通过将参数innodb_locks_unsafe_for_binlog设置为1来实现可以在二进制日志为STATEMENT下使用READ COMMITTED的事务隔离级别，例如：

```
mysql> SELECT @@version\G
*************************** 1. row ***************************
@@version: 5.0.77-log
1 row in set (0.00 sec)

mysql> system cat /etc/my.cnf | grep innodb_locks_unsafe_for_binlog
innodb_locks_unsafe_for_binlog=1

mysql> SHOW VARIABLES LIKE 'innodb_locks_unsafe_for_binlog'\G;
*************************** 1. row ***************************
Variable_name: innodb_locks_unsafe_for_binlog
        Value: ON
1 row in set (0.00 sec)

mysql> SET @@tx_isolation='read-committed';
Query OK, 0 rows affected (0.00 sec)

mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO a SELECT 1;
Query OK, 0 rows affected (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

是的，的确可以通过上述办法在MySQL 5.0之前的版本中使用READ COMMITTED事务隔离级别。但是就像参数innodb_locks_unsafe_for_binlog的名字一样，它是不安全的。在某些情况下，可能会导致master和slave之间数据的不一致。接着演示一种可能导致不同步的情况，先来看一下表a中的数据：

```
mysql> SELECT * FROM a\G;
*************************** 1. row ***************************
b: 1
*************************** 2. row ***************************
b: 2
*************************** 3. row ***************************
b: 4
*************************** 4. row ***************************
b: 5
4 rows in set (0.00 sec)
```

接着在 master 上开启一个会话 A 执行如下事务，并且不要提交。

```
# Session A on master
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> DELETE FROM a WHERE b<=5;
Query OK, 4 rows affected (0.01 sec)
```

同样地，在 master 上开启另一个会话 B 执行如下事务，并且提交。

```
# Session B on master
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO a SELECT 3;
Query OK, 0 rows affected (0.01 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

接着提交会话 A，并查看表 a 中的数据。

```
# Session A on master
mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM a\G;
*************************** 1. row ***************************
b: 3
```

但是在 slave 上看到的结果却是：

```
# Slave
mysql> select * from a;
Empty set (0.00 sec)
```

可以看到，数据产生了不一致。导致这个问题的原因有两点：

- 在 READ COMMITTED 事务隔离级别下，事务是没有 Gap Lock 锁的，因此可以在小

于等于 5 的范围内再插入一条记录。

- STATEMENT 记录的是 master 上产生的 SQL 语句，因此在 MASTER 服务器上执行的顺序为先删后插，但是在 STATEMENT 格式中记录的却是先插后删，逻辑上就产生了不一致。

要避免主从不一致的问题，只需解决上述问题中的一个就能保证数据的同步了。如使用 READ REPEATABLE 事务隔离级别可以避免上述第一种情况的发生，也就避免了 MASTER 和 SLAVE 数据不一致问题的产生。

在 MySQL 5.1 版本之后，由于支持了 ROW 格式的二进制日志记录格式，因此避免了第二种情况的发生，因此可以放心使用 READ COMMITTED 的事务隔离级别。但是即使不使用 READ COMMITTED 的事务隔离级别，也应该考虑将二进制日志的格式更换成 ROW，因为这个格式记录的是行的变更，而不是简单的 SQL 语句，所以可以避免一些不同步现象的产生，进一步保证数据的同步。InnoDB 存储引擎的创始人 Heikki Tuuri 也在 http://bugs.mysql.com/bug.php?id=33210 中建议使用 ROW 格式的二进制日志。

8.6 分布式事务编程

InnoDB 存储引擎提供了对于 XA 事务的支持，并通过 XA 事务来支持分布式事务的实现。分布式事务指的是允许多个独立的事务资源 (transactional resources) 参与到一个全局的事务中。事务资源通常是关系型数据库系统，也可以是其他类型的资源。全局事务要求在其中的所有参与的事务要么都提交，要么都回滚，这对于事务原有的 ACID 要求又有了提高。另外，在使用分布式事务时，InnoDB 存储引擎的事务隔离级别必须设置为 SERIALIZABLE。

XA 事务允许不同数据库之间的分布式事务，如一台服务器是 MySQL 数据库的，另一台是 Oracle 数据库的，可能还有一台服务器是 SQL Server 数据库的，只要参与到全局事务中的每个节点都支持 XA 事务即可。分布式事务可能在银行系统的转账中比较常见，如用户 David 需要从上海转 10 000 元到北京的用户 Mariah 的存储卡中，例如：

```
# Bank@Shanghai:
UPDATE account SET money = money - 10000 WHERE user='David';

# Bank@Beijing
UPDATE account SET money = money + 10000 WHERE user='Mariah';
```

在这种情况下，一定需要使用分布式事务来保证数据的安全。如果发生的操作不能都提交或都回滚，那么任何一个节点出现问题都会导致严重的结果，要么是 David 的账户被扣款而 Mariah 没收到钱，又或者是 David 的账户没有扣款而 Mariah 收到钱了。

XA 事务由一个或多个资源管理器（resource manager）、一个事务管理器（transaction manager）以及一个应用程序（application program）组成。

- 资源管理器：提供访问事务资源的方法。通常一个数据库就是一个资源管理器。
- 事务管理器：协调参与全局事务中的各个事务。需要和参与到全局事务中的所有资源管理器进行通信。
- 应用程序：定义事务的边界，指定全局事务中的操作。

在 MySQL 数据库的分布式事务中，资源管理器就是 MySQL 数据库，事务管理器为连接到 MySQL 服务器的客户端。图 8-6 显示了一个分布式事务的模型。

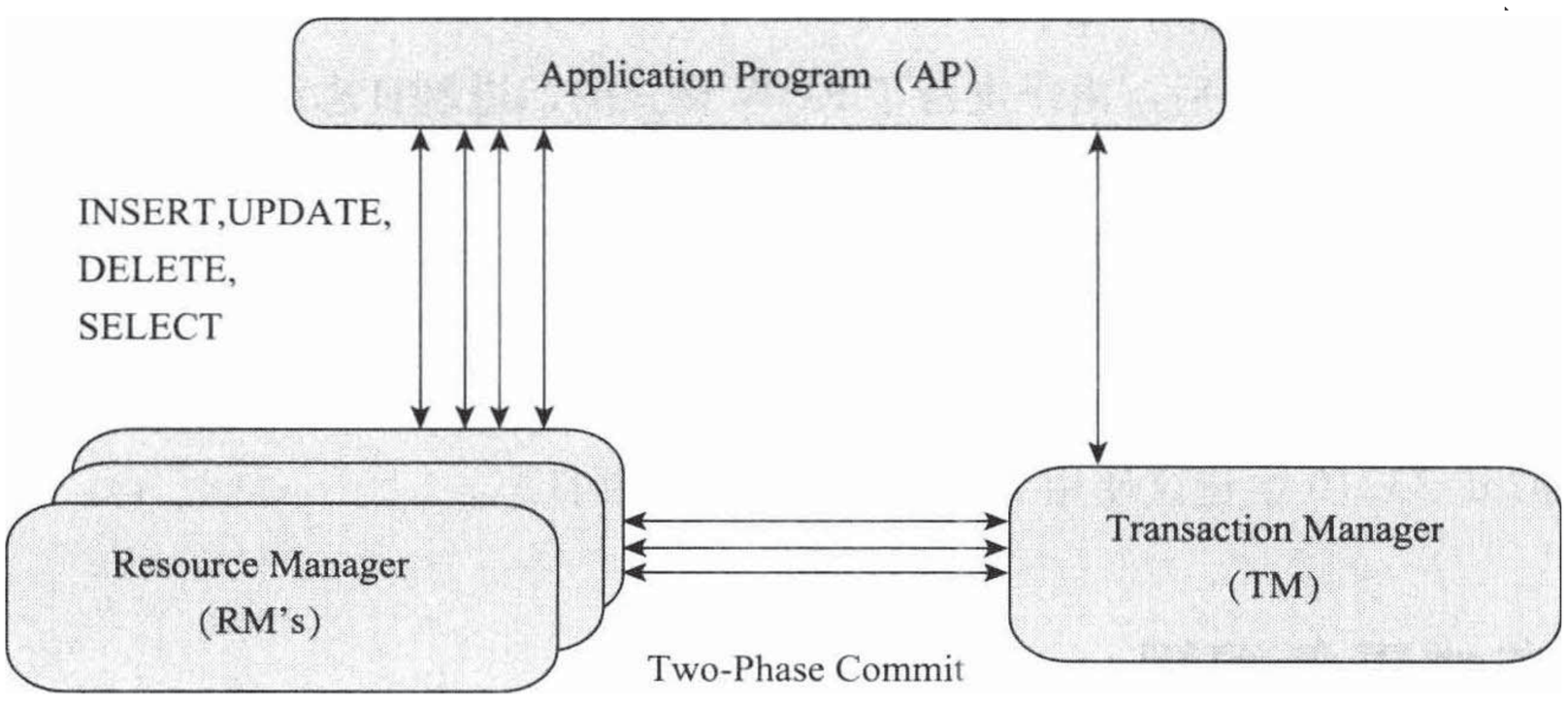


图 8-6　分布式事务模型

分布式事务使用两段式提交（two-phase commit）的方式。在第一个阶段，所有参与全局事务的节点都开始准备（PREPARE），告诉事务管理器它们准备好提交了。第二个阶段，事务管理器告诉资源管理器执行 ROLLBACK 还是 COMMIT。如果任何一个节点显示不能提交，则所有的节点都被告知需要回滚。可见与本地事务不同的是，需要多一次的 PREPARE 操作，待收到所有节点的同意信息后，再进行 COMMIT 或 ROLLBACK 操作。

MySQL 数据库 XA 事务的 SQL 语法如下：

```
XA {START|BEGIN} xid [JOIN|RESUME]

XA END xid [SUSPEND [FOR MIGRATE]]

XA PREPARE xid

XA COMMIT xid [ONE PHASE]

XA ROLLBACK xid

XA RECOVER
```

在单个节点上运行 XA 事务的例子：

```
mysql> XA START 'a';
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> INSERT INTO z SELECT 11;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> XA END 'a';
Query OK, 0 rows affected (0.00 sec)

mysql> XA PREPARE 'a';
Query OK, 0 rows affected (0.05 sec)

mysql> XA RECOVER\G;
*************************** 1. row ***************************
    formatID: 1
gtrid_length: 1
bqual_length: 0
        data: a
1 row in set (0.00 sec)

mysql> XA COMMIT 'a';
Query OK, 0 rows affected (0.05 sec)
```

在单个节点上运行分布式事务没有太大的实际意义，但是要在 MySQL 数据库的命令下演示多个节点参与的分布式事务也是行不通的。通常来说，都是通过编程语言来完成分布式事务的操作的。当前 Java 的 JTA（Java Transaction API）可以很好地支持 MySQL 的分布式事务，需要使用分布式事务应该认真参考其 API。 下面的一个示例显示了如何使用 JTA 来调用 MySQL 的分布式事务，实现的就是前面的银行转账，代码如下：

```
import java.sql.Connection;
import javax.sql.XAConnection;
import javax.transaction.xa.*;
import com.mysql.jdbc.jdbc2.optional.MysqlXADataSource;
import java.sql.*;

class MyXid implements Xid
{
   public int formatId;
   public byte gtrid[];
   public byte bqual[];

   public MyXid(){

   }

   public MyXid(int formatId, byte gtrid[], byte bqual[])
   {
          this.formatId = formatId;
          this.gtrid = gtrid;
```

```
            this.bqual = bqual;
        }

        public int getFormatId()
        {
            return formatId;
        }

        public byte[] getBranchQualifier()
        {
            return bqual;
        }

        public byte[] getGlobalTransactionId()
        {
            return gtrid;
        }
    }

    public class xa_demo {

        public static MysqlXADataSource GetDataSource(
                    String connString,
                    String user,
                    String passwd){
            try{
                    MysqlXADataSource ds = new MysqlXADataSource();
                    ds.setUrl(connString);
                    ds.setUser(user);
                    ds.setPassword(passwd);
                    return ds;
            }
            catch(Exception e){
                    System.out.println(e.toString());
                    return null;
            }
        }

        public static void main(String[] args) {
            String connString1 = "jdbc:mysql://192.168.24.43:3306/bank_shanghai";
            String connString2 =  "jdbc:mysql://192.168.24.166:3306/bank_beijing";
            try {
                    MysqlXADataSource ds1 =
                            GetDataSource(connString1,"peter"," 12345");
                    MysqlXADataSource ds2 =
                            GetDataSource(connString2,"david","12345");

                    XAConnection xaConn1 = ds1.getXAConnection();
                    XAResource xaRes1 = xaConn1.getXAResource();
                    Connection conn1 = xaConn1.getConnection();
```

```
Statement stmt1 = conn1.createStatement();

XAConnection xaConn2 = ds2.getXAConnection();
XAResource xaRes2 = xaConn2.getXAResource();
Connection conn2 = xaConn2.getConnection();
Statement stmt2 = conn2.createStatement();

Xid xid1 = new MyXid(
        100,
        new byte[]{0x01},
        new byte[]{0x02});
Xid xid2 = new MyXid(
        100,
        new byte[]{0x11},
        new byte[]{0x12});
try{
        xaRes1.start(xid1,XAResource.TMNOFLAGS);
        stmt1.execute("
             UPDATE account SET money = money-10000
             WHERE user='david'"
        );
        xaRes1.end(xid1,XAResource.TMSUCCESS);

        xaRes2.start(xid2,XAResource.TMNOFLAGS);
        stmt2.execute("
             UPDATE account SET money = money+10000
             WHERE user='mariah'"
        );
        xaRes2.end(xid2,XAResource.TMSUCCESS);

        int ret2 = xaRes2.prepare(xid2);
        int ret1 = xaRes1.prepare(xid1);

                if ( ret1 == XAResource.XA_OK
                        && ret2 == XAResource.XA_OK ){
                   xaRes1.commit(xid1,false);
                   xaRes2.commit(xid2,false);
                }
             }catch(Exception e){
                e.printStackTrace();
             }

        } catch (Exception e) {
             System.out.println(e.toString());
        }
    }
}
```

通过参数 innodb_support_xa 可以查看是否启用了对 XA 事务的支持（默认为 ON），例如：

```
mysql> SHOW VARIABLES LIKE 'innodb_support_xa'\G;
*************************** 1. row ***************************
Variable_name: innodb_support_xa
        Value: ON
1 row in set (0.01 sec)
```

另外需要注意的是，对 XA 事务的支持是在 MySQL 体系结构的存储引擎层，因此即使不参与外部的 XA 事务，MySQL 内部不同存储引擎层也会使用 XA 事务。假设用 START TRANSACTION 开启了一个本地事务，向 NDB Cluster 存储引擎的表 t1 插入一条记录，向 InnoDB 存储引擎的表 t2 插入一条记录，然后提交 COMMIT 命令。在 MySQL 内部也是通过 XA 事务来协调的，这样才可以实现两张表的原子性。

8.7 不好的事务编程习惯

8.7.1 在循环中提交

开发人员非常喜欢在循环中进行事务的提交，下面是他们常写的一个存储过程：

```
CREATE PROCEDURE load1(count INT UNSIGNED)
BEGIN
DECLARE s INT UNSIGNED DEFAULT 1;
DECLARE c CHAR(80) DEFAULT REPEAT('a',80);
WHILE s <= count DO
INSERT INTO t1 SELECT NULL,c;
COMMIT;
SET s = s+1;
END WHILE;
END;
```

其实，在上述的例子中，是否加上提交命令 COMMIT 并不是关键。由于 InnoDB 存储引擎默认为自动提交，因此去掉上述存储过程中的 COMMIT，结果其实是完全一样的。下面也是另一个容易被开发人员忽视的问题。

```
CREATE PROCEDURE load2(count INT UNSIGNED)
BEGIN
DECLARE s INT UNSIGNED DEFAULT 1;
DECLARE c CHAR(80) DEFAULT REPEAT('a',80);
WHILE s <= count DO
INSERT INTO t1 SELECT NULL,c;
SET s = s+1;
END WHILE;
END;
```

不论上面哪个存储过程都存在一个问题，当发生错误时，数据库会停留在一个未知的位

置。例如，我们要插入 10 000 条记录，但是在插入 5000 条时，发生了错误，而这时前 5000 条记录已经存放在数据库中，我们应该怎么处理呢？另一个问题是性能问题，上面两个存储过程都不会比下面的存储过程 load3 快，因为下面的存储过程将所有的 INSERT 都放在一个事务中。

```
CREATE PROCEDURE load3(count INT UNSIGNED)
BEGIN
DECLARE s INT UNSIGNED DEFAULT 1;
DECLARE c CHAR(80) DEFAULT REPEAT('a',80);
START TRANSACTION;
WHILE s <= count DO
INSERT INTO t1 SELECT NULL,c;
SET s = s+1;
END WHILE;
COMMIT;
END;
```

比较这 3 个存储过程的执行时间，情况如下：

```
mysql> CALL load1(10000);
Query OK, 0 rows affected (1 min 3.15 sec)

mysql> TRUNCATE TABLE t1;
Query OK, 0 rows affected (0.05 sec)

mysql> CALL load2(10000);
Query OK, 1 row affected (1 min 1.69 sec)

mysql> TRUNCATE TABLE t1;
Query OK, 0 rows affected (0.05 sec)

mysql> CALL load3(10000);
Query OK, 0 rows affected (0.63 sec)
```

显然，第三种方法要快得多！这是因为每一次提交都要写一次重做日志，所以存储过程 load1 和 load2 实际写了 10 000 次重做日志文件，而对于存储过程 load3 来说，实际只写了 1 次重做日志。可以对第二个存储过程 load2 的调用进行调整，同样可以达到存储过程 load3 的性能，例如：

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> CALL load2(10000);
Query OK, 1 row affected (0.56 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.03 sec)
```

大多数程序员会使用第一种或者第二种方法，有人可能不知道 InnoDB 存储引擎自动提交的情况，另外有些人可能持有以下两种观点：首先，在曾经使用过的数据库中，对于事务的要求总是尽快地进行释放，不能有长时间的事务；其次，可能担心存在 Oracle 数据库中由于没有足够 UNDO 空间产生的 Snapshot Too Old 的经典问题。MySQL InnoDB 存储引擎都没有上述两个问题，因此程序员不论从何种角度出发，都不应该在一个循环中反复地进行提交操作，不论是显式的提交还是隐式的提交。

8.7.2 使用自动提交

自动提交并不是一个好习惯，因为这容易让初级 DBA 犯错，另外一些开发人员可能产生错误的理解，如我们在前一小节中提到的循环提交问题。MySQL 数据库默认使用自动提交（autocommit），可以使用如下语句来改变当前自动提交的方式：

```
mysql> SET autocommit=0;
Query OK, 0 rows affected (0.00 sec)
```

也可以使用 START TRANSACTION 或 BEGIN 来显式地开启一个事务。显式开启事务后，在默认设置下（即参数 completion_type 等于 0），MySQL 会自动执行 SET AUTOCOMMIT=0 的命令，并在 COMMIT 或者 ROLLBACK 结束一个事务后执行 SET AUTOCOMMIT=1。

另外，在使用不同语言的 API 时，自动提交是不同的。MySQL C API 默认的提交方式是自动提交，而 MySQL Python API 会自动执行 SET AUTOCOMMIT=0，以禁用自动提交。因此选用不同的语言来编写数据库应用程序前，应该对连接 MySQL 的 API 做好研究。

笔者认为，在编写应用程序开发时，最好把事务的控制权限交给开发人员，即在程序端进行事务的开始和结束。同时，开发人员必须了解自动提交可能带来的问题。如果没有意识到自动提交的这个特性，等到出现错误时应用就会遇到大麻烦。

8.7.3 使用自动回滚

InnoDB 存储引擎支持通过定义一个 HANDLER 来进行事务的自动回滚操作，如在一个存储过程中发生了错误会自动对其进行回滚操作。因此很多开发人员喜欢在应用程序的存储过程中使用自动回滚操作，如下面的一个存储过程。

```
CREATE PROCEDURE sp_auto_rollback_demo ()
BEGIN
DECLARE EXIT HANDLER FOR SQLEXCEPTION ROLLBACK;
START TRANSACTION;
INSERT INTO b SELECT 1;
INSERT INTO b SELECT 2;
INSERT INTO b SELECT 1;
INSERT INTO b SELECT 3;
COMMIT;
```

```
END;
```

存储过程 sp_auto_rollback_demo 先定义了一个 EXIT 类型的 HANDLER，当捕获到错误时进行回滚。结构如下：

```
mysql> SHOW CREATE TABLE b\G;
*************************** 1. row ***************************
       Table: b
Create Table: CREATE TABLE `b` (
  `a` int(11) NOT NULL DEFAULT '0',
  PRIMARY KEY (`a`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
```

因此插入第二个记录 1 时会发生错误，但是因为启用了自动回滚的操作，所以这个存储过程的执行结果如下：

```
mysql> CALL sp_auto_rollback_demo;
Query OK, 0 rows affected (0.06 sec)

mysql> SELECT * FROM b;
Empty set (0.00 sec)
```

看起来没有问题，运行非常正常。但是，执行 sp_auto_rollback_demo 这个存储过程的结果到底是正确的还是错误的？对于同样的存储过程 sp_auto_rollback_demo，为了得到执行正确与否的信息，开发人员可以进行这样的处理：

```
CREATE PROCEDURE sp_auto_rollback_demo ()
BEGIN
DECLARE EXIT HANDLER FOR SQLEXCEPTION BEGIN ROLLBACK; SELECT -1; END;
START TRANSACTION;
INSERT INTO b SELECT 1;
INSERT INTO b SELECT 2;
INSERT INTO b SELECT 1;
INSERT INTO b SELECT 3;
COMMIT;
SELECT 1;
END;
```

当发生错误时，先回滚然后返回 −1，表示运行有错误；返回 1 表示运行正常。这次运行的结果如下：

```
mysql> CALL sp_auto_rollback_demo ()\G;
*************************** 1. row ***************************
-1: -1
1 row in set (0.04 sec)

mysql> SELECT * FROM b;
Empty set (0.00 sec)
```

用户似乎可以得到运行是否准确的信息。但问题还没有最终解决，对于开发者来说，重要的不仅是知道发生了错误，还要知道发生了什么样的错误。自动回滚存在不知道发生什么错误这样的一个问题。

喜欢使用自动回滚的人大多是以前使用 Microsoft SQL Server 数据库的开发人员。在 Microsoft SQL Server 数据库中，可以使用 SET XABORT ON 来自动回滚一个事务，因为 Microsoft SQL Server 数据库不仅会自动回滚当前的事务，还会抛出异常，开发人员可以捕获到这个异常。Microsoft SQL Server 数据库和 MySQL 数据库在这方面是不同的。

就像之前小节中所讲到的，对事务执行 BEGIN、COMMIT 和 ROLLBACK 操作的工作应该交给程序端来完成，存储过程所需完成的只是一个逻辑操作，即对逻辑进行封装。下面演示用 Python 语言编写的程序调用一个存储过程 sp_rollback_demo，存储过程 sp_rollback_demo 和之前的存储过程 sp_auto_rollback_demo 在逻辑上完成的内容大致相同。

```
CREATE PROCEDURE sp_rollback_demo ()
BEGIN
INSERT INTO b SELECT 1;
INSERT INTO b SELECT 2;
INSERT INTO b SELECT 1;
INSERT INTO b SELECT 3;
END;
```

和 sp_auto_rollback_demo 存储过程不同的是，在 sp_rollback_demo 存储过程中去掉了对于事务的控制语句，将这些操作交由程序来完成。接着来看 test_demo.py 的程序源代码。

```
#! /usr/bin/env python
#encoding=utf-8

import MySQLdb

try:
        conn = MySQLdb.connect(host="192.168.8.7",user="root",passwd="xx",db="test")
        cur = conn.cursor()
        cur.execute("SET autocommit=0")
        cur.execute("CALL sp_rollback_demo")
        cur.execute("COMMIT")
except Exception,e:
        cur.execute("ROLLBACK")
        print e
```

观察运行 test_demo.py 程序的结果：

```
[root@nineyou0-43 ~]# python test_demo.py
starting rollback
(1062, "Duplicate entry '1' for key 'PRIMARY'")
```

在程序中控制事务的好处是，用户可以得知发生错误的原因。如在上述这个例子中，我

们知道是因为发生了1062这个错误，错误的提示内容是Duplicate entry '1' for key 'PRIMARY'，即发生了主键重复的错误。之后可以根据错误发生的原因来进一步调试程序。

8.8 长事务

长事务（Long-Lived Transactions），顾名思义，就是需要执行时间较长的事务。例如银行系统的数据库，每过一个阶段可能需要更新对应账户的利息。如果对应账号的数量非常大，如对有1亿用户的表account，需要执行下列语句：

```
UPDATE account SET account total = account total + (1 + interest rate)
```

这时，这个事务可能需要非常长的时间来完成。可能是一个小时也可能是四五个小时，这取决于数据库的硬件配置。DBA和开发人员本身能做的事情非常少。然而，由于事务的ACID特性，这个操作被封装在一个事务中完成。这就产生了一个问题，在执行过程中，在数据库、操作系统或硬件等发生问题的情况下，重新开始事务的代价变得不可接受。数据库需要回滚所有已经发生的变化，而这个过程可能比产生这些变化的时间还要长。因此，对于长事务的问题，有时可以通过将其转化为小批量（mini batch）的事务来进行处理。当事务发生错误时，只需要回滚一部分数据，然后接着上次的已完成的事务继续进行。

例如，对于前面讨论的银行利息计算问题，我们可以将其分解为小批量事务来完成。下面给出的是伪代码，我们既可以通过程序来完成，也可以通过存储过程来完成。

```
void ComputeInterest(double interest_rate){

   long last_account_done, max_account_no, log_size;
   int batch_size = 100000;

   EXEC SQL SELECT COUNT(*) INTO log_size FROM batchcontext;

   if ( SQLCODE != 0 || log_size == 0 ){
         EXEC SQL DROP TABLE IF EXISTS batchcontext;
         EXEC SQL CREATE TABLE batchcontext ( last_account_done BIGINT );

         last_account_done = 0;
         INSERT INTO batchcontext SELECT 0;
   }
   else {
         EXEC SQL SELECT last_account_no
                     INTO last_account_done
                     FROM batchcontext;
   }

   EXEC SQL SELECT COUNT(*) INTO max_account_no
            FROM account LOCK IN SHARE MODE;
```

```
WHILE ( last_account_no < max_account_no ){
        EXEC SQL START TRANSACTION;
        EXEC SQL UPDATE account
                    SET account_total = account_total * ( 1+interest_rate );
                    WHERE account_no
                        BETWEEN last_account_no
                        AND last_account_no + batch_size;
        EXEC SQL UPDATE batchcontext
                    SET last_account_done = last_account_done + batch_size;
        EXEC SQL COMMIT WORK;
        last_account_done = last_account_done + batch_size;
}
```

上述代码将一个需要处理 1 亿用户的大事务分解为每次处理 10 万用户的小事务，通过批量处理小事务来完成大事务的逻辑。每完成一个小事务，将完成的结果存放在 batchcontext 表中，表示已完成批量事务的最大账号 ID。当事务运行过程中因产生问题而需要重做事务时，可以从这个已完成的最大账号 ID 继续进行批量的小事务，这样重新开启事务的代价就显得比较低，也更容易让用户接受。batchcontext 表的另外一个好处是，在长事务的执行过程中，用户可以知道现在大概已经执行到了哪个阶段。如一共有 1 亿条记录，现在表 batchcontext 中最大账号 ID 为 4000 万，也就是说明这个大事务大概完成了 40% 的工作。

这里还有一个小地方需要注意，在从表 account 中取得 max_account_no 时，人为地加上了一个共享锁，用来保证在事务处理过程中，没有其他的事务可以更新表中的数据，这是有意义并且也是非常有必要的操作。

8.9 小结

在本章我们了解到 MySQL InnoDB 存储引擎管理事务的许多方面。了解事务如何工作以及如何使用事务，这对于在任何数据库中正确实现应用都是必要的。此外，事务是数据库区别于文件系统的一个关键特性。

事务必须遵循 ACID 特性，即 Atomicity（原子性）、Consistency（一致性）、Isolation（隔离性）和 Durability（持久性）。接着介绍了 InnoDB 存储引擎支持的四个事务隔离级别，知道了 InnoDB 存储引擎的默认事务隔离级别是 REPEATABLE READ，不同于 SQL 标准对于事务隔离级别的要求，InnoDB 存储引擎在 REPEATABLE READ 隔离级别下就可以达到 SQL 标准的 SERIALIZABLE 的隔离要求。

本章最后讲解了操作事务的 SQL 语句以及怎样在应用程序中正确地使用事务。在默认

配置下，MySQL 数据库总是自动提交的——如果不知道这点，可能会带来非常不好的结果。此外，在应用程序中，最好的做法是把事务的 START TRANSACTION、COMMIT、ROLLBACK 操作交给程序端来完成，而不是在存储过程内完成。在完整了解了 InnoDB 存储引擎事务机制后，相信你可以开发出一个很好的企业级 MySQL InnoDB 数据库应用了。

第 9 章

索　　引

如何进行高质量的SQL编程，对索引的掌握和理解是关键。正确地使用索引可使SQL语句运行得更快，而错误地添加和使用索引可能带来有灾难性的结果。然而，如果DBA或开发人员能够了解索引的本质，例如索引是怎么实现的，优化器是怎么使用索引的，显然这会为SQL编程带来极大的提高。此外，不要盲从于一些所谓的规律，比如联合索引（a，b），索引不会用于对b列的查询。笔者始终相信，只要理解事物的本质，那么所有的问题都会迎刃而解。

9.1 缓冲池、顺序读取与随机读取

根据存储介质的不同，可以将数据库分为基于磁盘的数据库系统、基于内存的数据库系统，以及混合型数据库系统。基于磁盘的数据库系统（disk-base database）是最为常见的一种关系型数据库，比如MySQL、Oracle、SQL Server、DB2数据库。随着内存容量的不断增加，基于内存的数据库系统（in-memory database）也变得十分流行，MySQL NDB Cluster，Oracle Ten Times等数据库厂商都提供了内存关系型数据库系统。而混合型数据库系统（hybrid database）将上述两种数据库的优点进行了结合。

毫无疑问，基于内存的数据库系统是最快的，因为数据库不需要对磁盘进行操作。磁盘的速度要远慢于内存的速度，因此基于磁盘的数据库系统一般都有缓冲池，即一块内存区域，其作用是将从磁盘上读取的指定大小数据——称为页（或块），放入缓冲池。当再次读取时，数据库首先判断该页是否在缓冲池中，如果在则直接读取缓冲池中的页，如果不在则读取磁盘上的页。对于写操作，数据库将页读入缓冲池，然后在缓冲池中对页进行修改，修改完成后的页一般被异步地写入磁盘上。对于缓冲池的维护一般采用最近最少使用（Least Recently Used，LRU）算法。由此可见，缓冲池的大小决定了数据库的性能。若数据库中的数据可以完全存放于缓冲池中，则可以认为这时数据库的性能是最优的。除了同步/异步的写磁盘操作外，所有其他操作都可以在内存中完成。

对于MySQL数据库系统，由于其有着各种不同的存储引擎，因此其缓冲池是基于存储引擎的，也就是说每个存储引擎都有自己的缓冲池。对于MyISAM存储引擎来说，变量key_buffer_size决定了缓冲池的大小。对于InnoDB存储引擎来说，变量innodb_buffer_pool_size决定了缓冲池的大小。图9-1是Percona公司CTO Vadim对缓冲池所做的性能测试。

图9-1中的A点表示此时缓冲池为20GB，数据库的数据全部被缓存在缓冲池中，这时数据库的性能是最优的，测试的结果为2500TPS（transaction per second）。因为数据已经全部被缓存，继续增大缓冲池为22GB并不会对性能有多大的提升。B点表示缓冲池的大小为18 GB，此时只有一小部分的数据不能被缓存，测试的结果为900TPS。可见缓冲池大小减少了10%，性能却下降了164%。C点表示缓冲池的大小为6GB，只能缓存30%的数据，这时数据库的性能更差，测试结果为130 TPS。

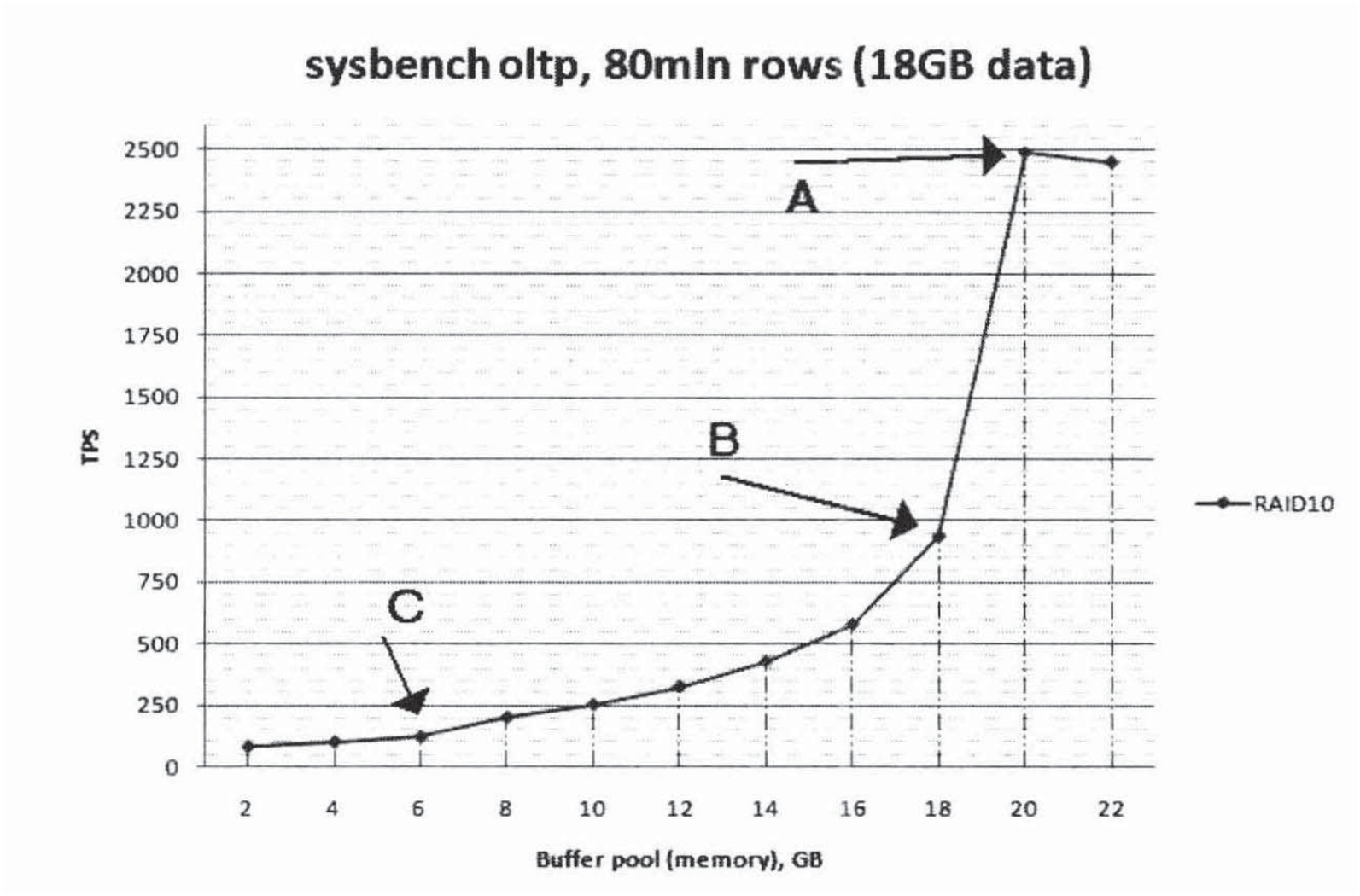


图 9-1　缓冲池大小对于数据库性能的影响

在现实的生产环境中，在大多数情况下经过线上的一段运行时间后，数据库的大小通常都会大于内存的大小。例如数据库的大小是 500GB，而服务器的内存可能只有 96GB 或者 128GB。这种情况非常正常，一般来说，对于 500GB 的数据库，热点的数据可能是小部分的，具体占据多大的比重，这取决于具体的应用。

在生产环境中缓冲池的大小小于数据文件的大小，因此不可避免地存在磁盘的读取操作。但是传统的机械硬盘的特性决定了顺序读取要远快于离散读取。图 9-2 显示了传统机械硬盘的构造。

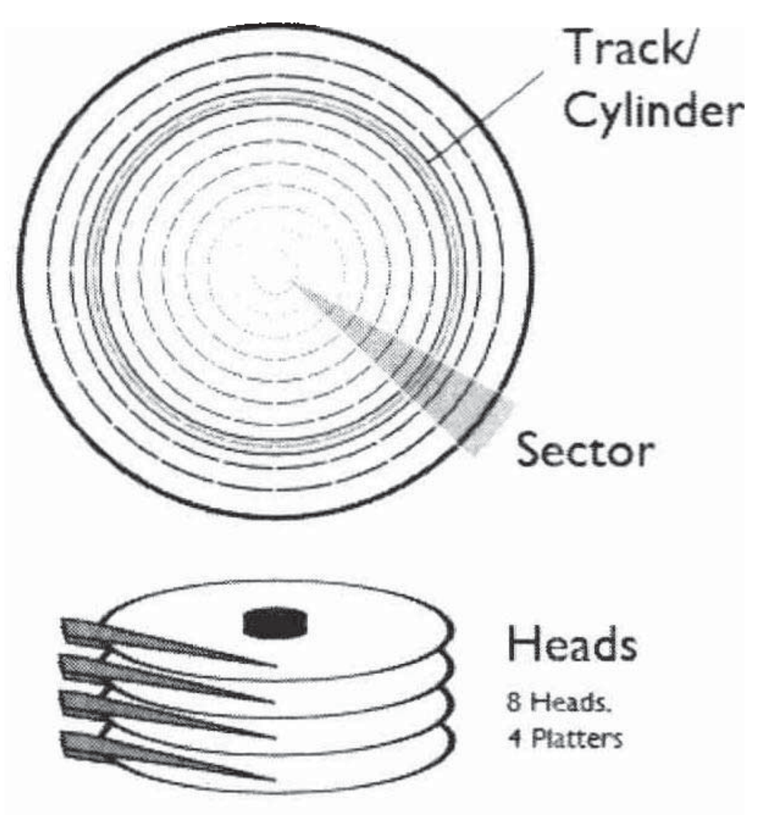


图 9-2　传统机械硬盘的组成

传统机械硬盘由磁头（head）、磁道（track）、扇区（sector）、柱面（cylinder）组成。读取时需要通过磁头的移动来定位数据，这个时间称为寻道时间（seek time）。目前，一块15000转SAS传统的机械硬盘的平均寻道时间为2.58ms。需要注意的是，这指的是平均时间。根据平均寻道时间，可以大致估算IOPS（IO per second），即1000/2.58 = 380 IOPS，而在实际情况下单块磁盘的IOPS一般为150左右。如果数据都是顺序存放的，显然寻道时间会快很多，这时的寻道时间一般只需0.14ms。可见，比平均寻道时间提高了18倍。

在介绍完上述内容后，可以给出顺序读取、随机读取的定义。顺序读取（sequntial read）是指顺序地读取磁盘上的页。随机读取（random read）是指访问的页不是连续的，需要磁盘的磁头不断移动。这里需要注意的是，这里的“顺序”指的是逻辑上的顺序，在物理上不可能保证所有的数据都是顺序的。而为了保证顺序，数据库存储引擎一般都根据区（extent）来管理页，例如在InnoDB存储引擎中1个区是连续的64个页。因此在顺序读取数据库时，可以保证这64个页是连续的，而区与区之间的页，可能是连续的也可能是不连续的。

固态硬盘是近几年出现的一种新的存储设备，其内部由闪存（flash memory）组成。因为具有低延迟性、低功耗、轻量，以及防震性，闪存设备已在移动设备上得到了广泛的应用。而企业级的生产应用使用固态硬盘，同时通过并联多块闪存来进一步提高数据传送的吞吐量。

图9-3显示了一个双通道的固态硬盘架构，通过支持4路的闪存交叉存储来降低固态硬盘的访问延时，同时增大并发的读写操作。而通过进一步增加通道的数量，固态硬盘的性能可以有线性的提高，例如常见的Intel SSD X-25M就是10通道的固态硬盘。

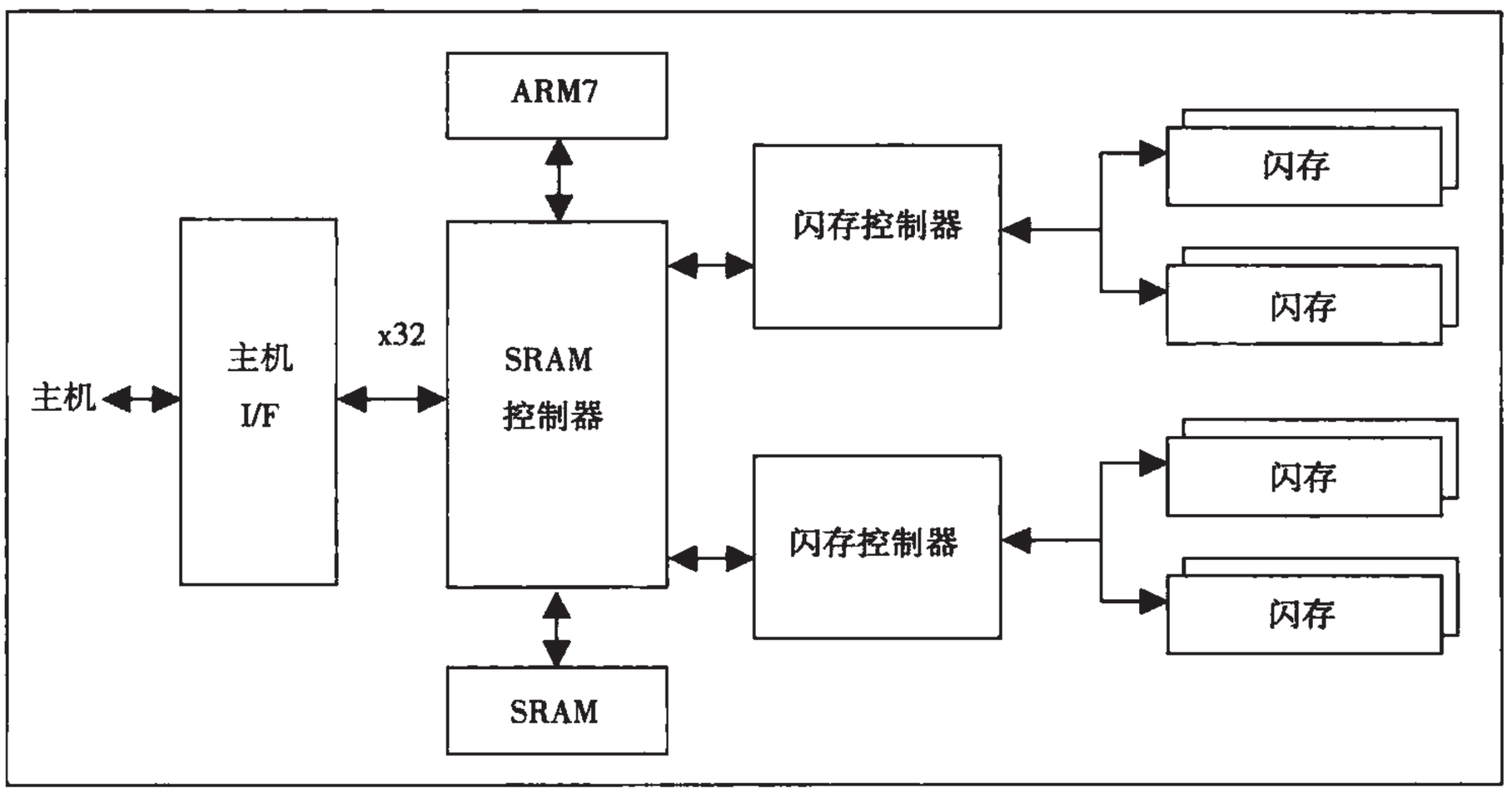


图9-3 双通道的固态硬盘架构

因为闪存是一个完全的电子设备，没有任何的读写磁头等移动部件，因此固态硬盘有着

较低的访问延时。当主机发布一个读写请求时，固态硬盘的控制器会把 I/O 命令从逻辑地址映射到实际的物理地址，写操作还需要修改相应的映射表信息。算上这些额外的开销，固态硬盘的访问延时一般小于 0.1 ms 左右，随机读取可达 4000 IOPS 甚至更高。即使是固态硬盘，顺序读取还是要快于随机读取。因此对于固态硬盘的优化需要不同于传统机械硬盘的思考，但是这更多的是数据库存储引擎内部需要考虑的问题。

9.2 数据结构与算法

B+ 树索引是最为常见，也是在数据库中使用最为频繁的一种索引。在介绍该索引之前先介绍与之密切相关的一些算法与数据结构，这有助于读者更好地理解 B+ 树索引的工作方式。

9.2.1 二分查找法

二分查找法（binary search）也称为折半查找法，用来查找一组有序记录数组中某一项记录。其基本思想是：将记录按有序化（递增或递减）排列，查找过程中采用跳跃式方式查找，即先以有序数列的中点位置为比较对象，如果要找的元素值小于该中点元素，则将待查序列缩小为左半部分，否则为右半部分。通过一次比较，将查找区间缩小一半。

例如对于 5、10、19、21、31、37、42、48、50、52 这 10 个数，要从中查找 48 这条记录，其查找过程如图 9-4 所示。

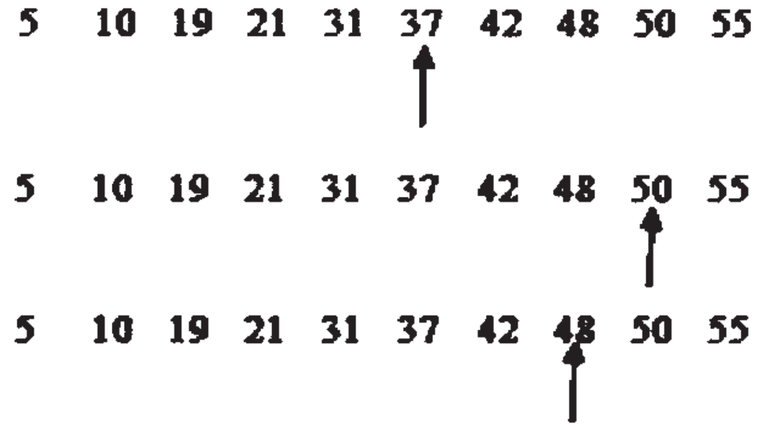


图 9-4 二分查找法

从图 9-4 可以看出，3 次就找到了 48 这个数。如果是顺序查找，则需要 8 次。因此二分查找法的效率比顺序查找法要好（平均来说），如果说查 5 这条记录，顺序查找只需 1 次，而二分查找法需要 4 次。对于上面 10 个数来说，平均查找次数为 (1+2+3+4+5+6+7+8+9+10)/10=5.5 次，而二分查找法为 (4+3+2+4+3+1+4+3+2+3)/10=2.9 次。在最坏的情况下，顺序查找的次数为 10，而二分查找的次数为 4。

二分查找法的应用极其广泛，并且它的思想易于理解。第一个二分查找算法早在 1946 年就出现了，但是第一个完全正确的二分查找算法直到 1962 年才出现。

在 B+ 树索引中，B+ 树索引只能找到某条记录所在的页，需再根据二分查找法来进一步

找到记录所在页的具体位置。

9.2.2 二叉查找树和平衡二叉树

在介绍 B+ 树前，先要了解一下二叉查找树。B+ 树是通过二叉查找树，再由平衡二叉树、B 树演化而来。相信在任何一本有关数据结构的书中都可以找到二叉查找树的章节，二叉查找树是一种经典的数据结构。图 9-5 显示了一棵二叉查找树。

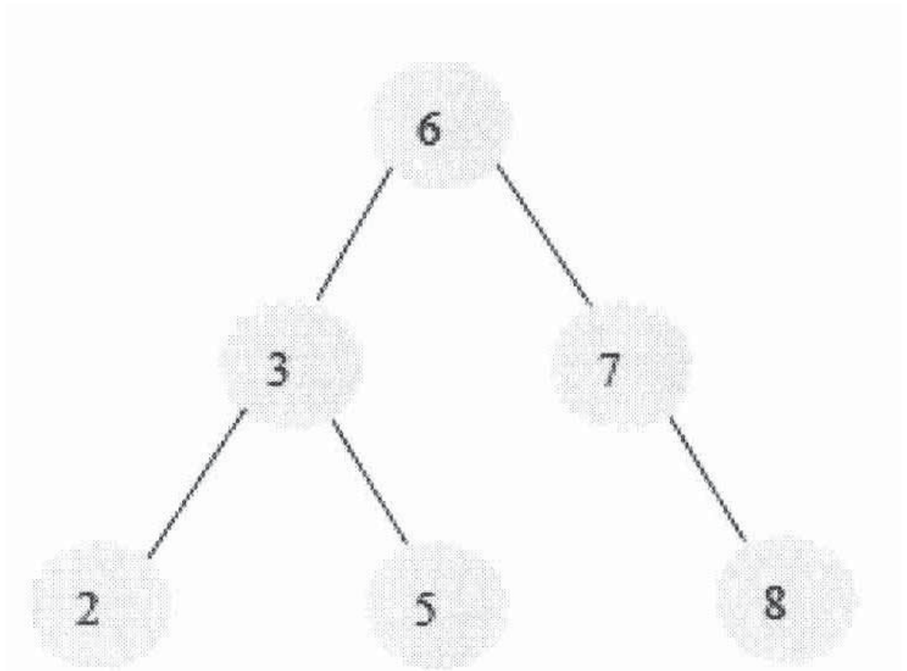


图 9-5 二叉查找树

图 9-5 中的数字代表每个节点的键值。在二叉查找树中，左子树的键值总是小于根的键值，右子树的键值总是大于根的键值，因此可以通过中序遍历得到键值的排序输出。对图 9-5 进行中序遍历后输出：2、3、5、6、7、8。

对图 9-5 的这棵二叉树进行查找，如查找键值为 5 的记录，先找到根，其键值是 6，6 大于 5，因此查找 6 的左子树，找到 3；而 5 大于 3，再找右子树……一共找了 3 次。如果按 2、3、5、6、7、8 的顺序来找同样需要 3 次。用同样的方法再查找键值为 8 的这条记录，这次用了 3 次查找，而顺序查找时需要 6 次。如果计算平均查找次数可得：顺序查找的平均查找次数为 (1+2+3+4+5+6)/6=3.3 次，二叉查找树的平均查找次数为 (3+3+3+2+2+1)/6=2.3 次。二叉查找树比顺序查找的平均效率要高。

二叉查找树可以任意构造，同样是 2、3、5、6、7、8 这五个数字，也可以按照图 9-6 的方式建立二叉查找树。

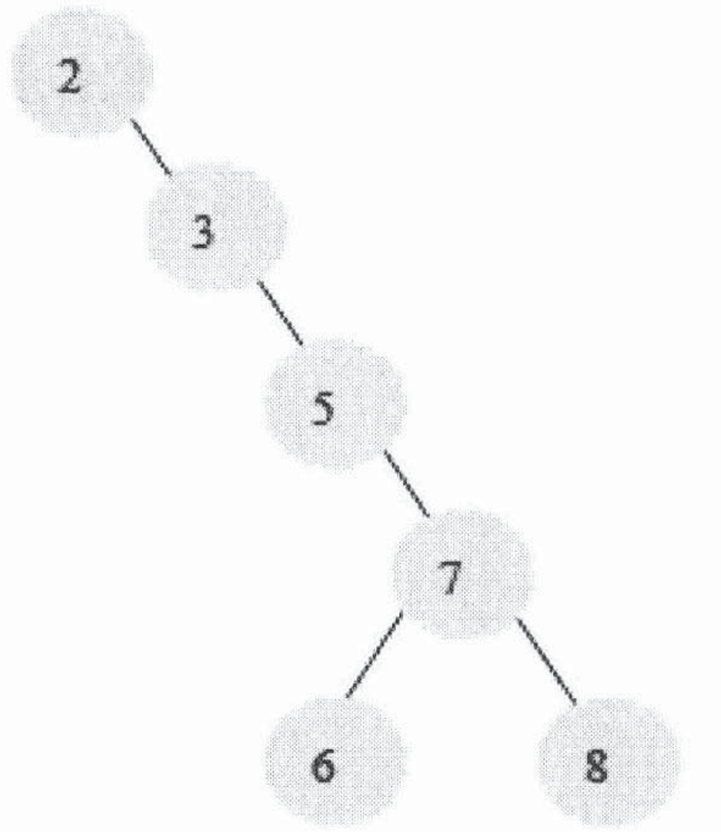


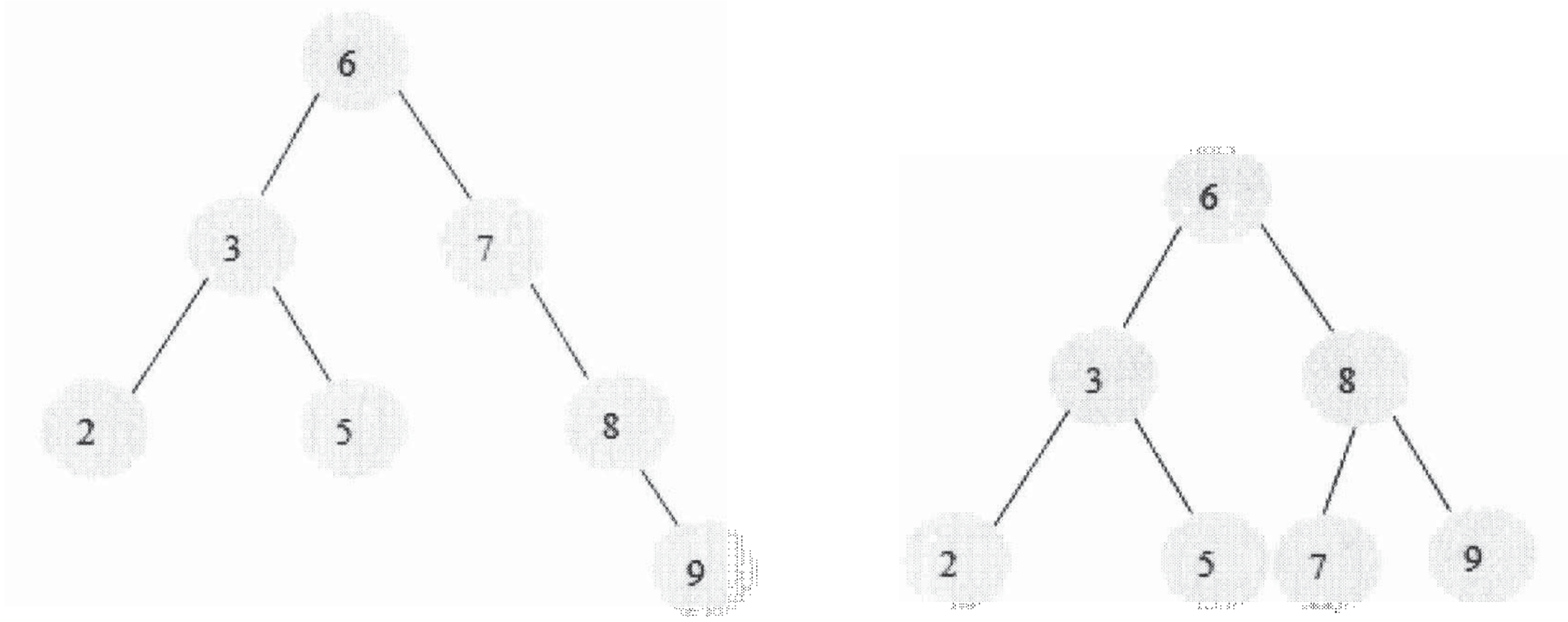
图 9-6 低效的一棵二叉查找树

图 9-6 的平均查找次数为 (1+2+3+4+5+5)/6=3.16 次，和顺序查找差不多。显然这棵二叉查找树的效率就差很多。因此若想最大性能地构造一个二叉查找树，需要这棵二叉查找树是平衡的，因此引入了新的定义——平衡二叉树，又称为 AVL 树。

平衡二叉树的定义如下：首先符合二叉查找树的定义，其次必须满足任何节点的两棵子树的高度最大差为 1。显然，图 9-6 不满足平衡二叉树的定义，而图 9-5 是一棵平衡二叉树。平衡二叉树在查找方面的性能是比较高的，但不是最高的，只是接近最高性能。要达到最好的性能需要建立一棵最优二叉树，但是最优二叉树的建立和维护需要大量的操作，因此一般只需建立一棵平衡二叉树即可。

平衡二叉树的查询速度的确很快，但是维护一棵平衡二叉树的代价非常大，通常需要 1

次或多次左旋或右旋来得到经过插入或更新操作后二叉树的平衡性。对于图 9-5 所示的平衡树，当我们需要插入一个新的键值为 9 的节点时，需要做如图 9-7 所示的变动。

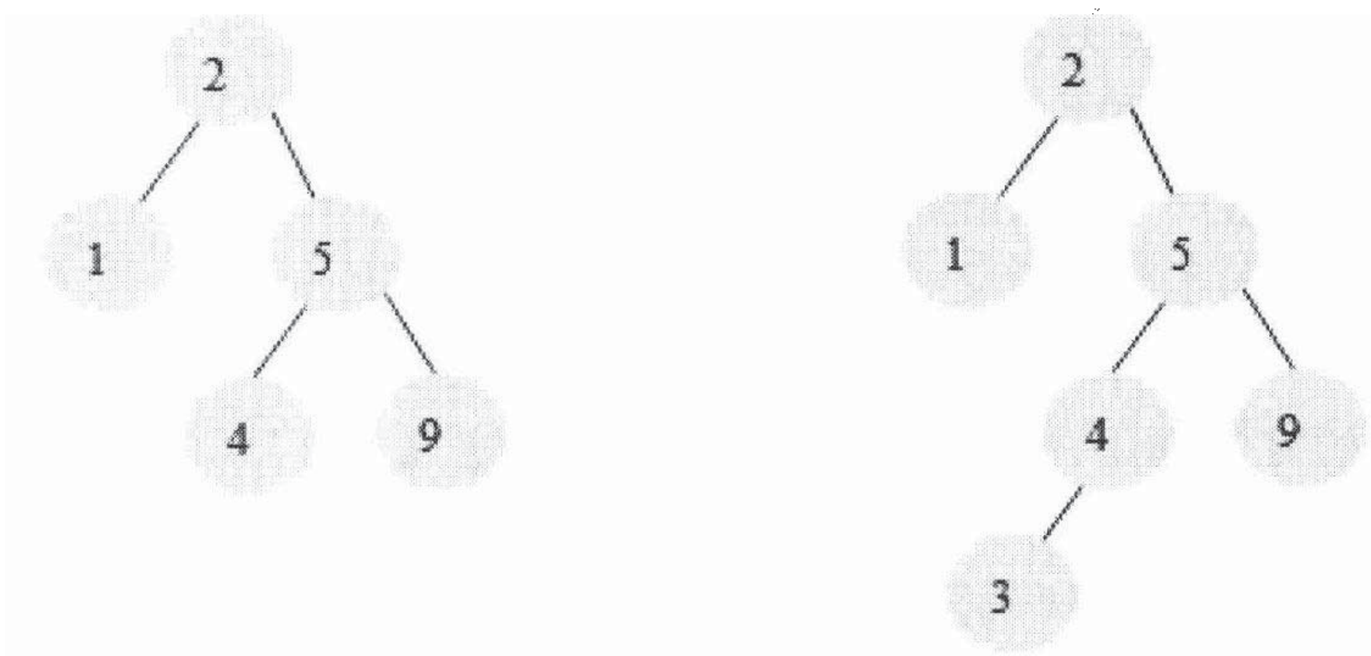


a）插入新值 9　　b）左旋以保证平衡

图 9-7　插入键值 9，平衡二叉树的变化

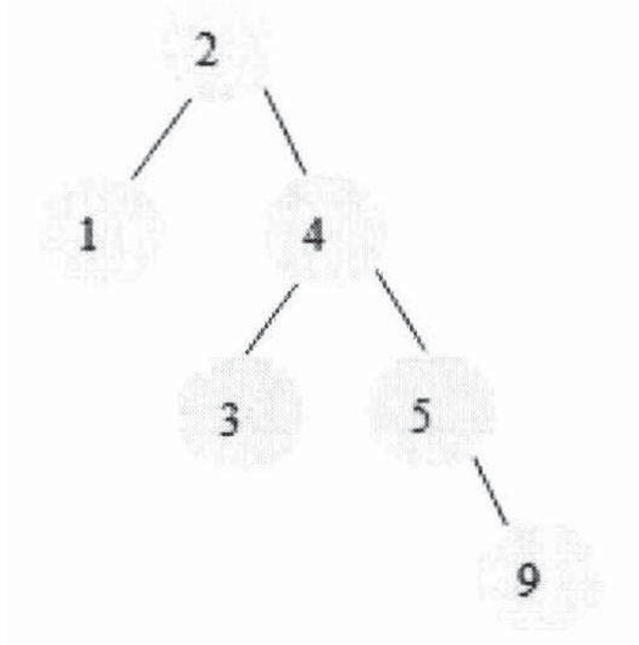
这里通过一次左旋操作就将插入后的树重新变为平衡二叉树。但是有的情况可能需要进行多次的旋转操作，如图 9-8 所示。

图 9-7 和图 9-8 演示了向一棵平衡二叉树插入一个新的节点后，平衡二叉树需要做的旋转操作。除了插入操作，还有更新和删除操作，不过进行这些操作后在恢复二叉树的平衡性上没有本质的区别，它们都是通过左旋或者右旋来完成。因此对一棵平衡树的维护是有一定开销的，不过平衡二叉树多用于内存结构对象中，因此维护的开销相对较小。

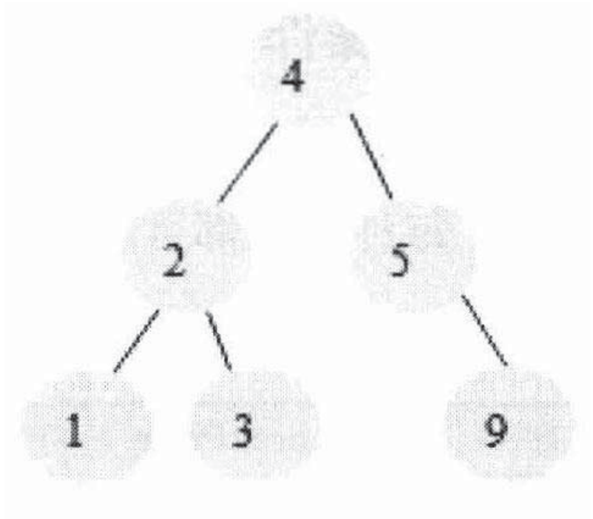


a）平衡二叉树　　b）插入新键值 3

图 9-8　需多次旋转的平衡二叉树



c）右旋一次



d）再左旋一次

图 9-8 （续）

9.3 B+ 树

B+ 树和二叉树、平衡二叉树都是经典的数据结构。B+ 树由 B 树和索引顺序访问方法（ISAM，是不是很熟悉？对，这也是 MyISAM 引擎最初参考的数据结构）演化而来，但是在实现过程中几乎没有使用 B 树的情况了。

相信在任何一本数据结构书中都能找到 B+ 树的定义，其定义十分复杂，这里列出 B+ 树的定义只会让读者感到更加困惑。因此这里只精简地给出 B+ 树的介绍：B+ 树是为磁盘或其他直接存取辅助设备设计的一种平衡查找树，在 B+ 树中，所有记录节点都是按键值的大小顺序存放在同一层的叶子节点，各叶子节点通过指针进行链接。先来看一个 B+ 树，如图 9-9 所示，其高度为 2，每页可存放 4 条记录，扇出（fan out）为 5：

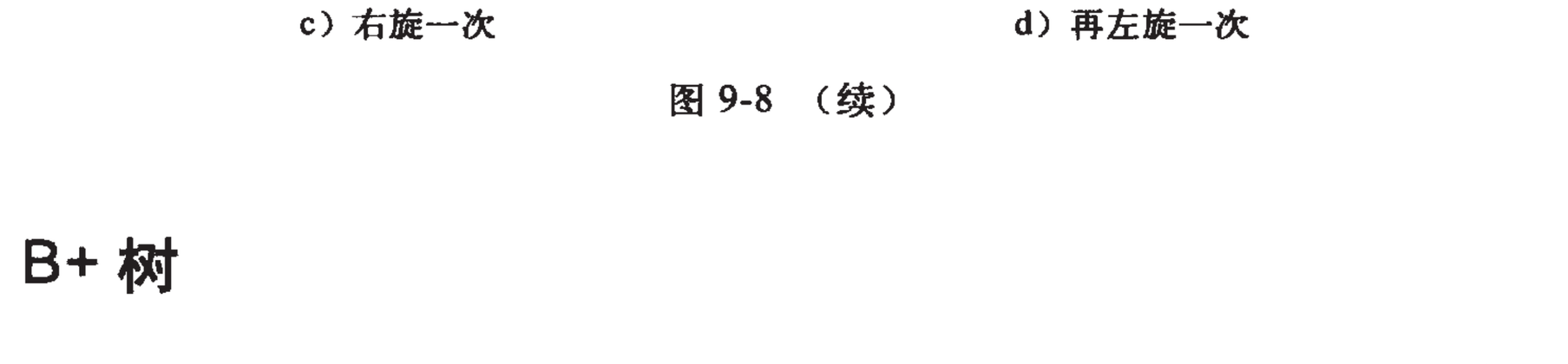


图 9-9 一棵高度为 2 的 B+ 树

从图 9-9 可以看出，所有记录都在叶子节点上，并且是顺序存放的，如果我们从最左边的叶子节点开始顺序遍历，可以得到所有键值的顺序排序：5、10、15、20、25、30、50、55、60、65、75、80、85、90。

9.3.1 B+ 树的插入操作

B+ 树的插入要求必须保证插入后叶子节点中的记录依然顺序排列，同时需要考虑插入到 B+ 树的 3 种情况，每种情况都可能导致不同的插入算法，如表 9-1 所示。

表 9-1 B+ 树插入的 3 种情况

Leaf Page 是否为满	Index Page 是否为满	操作
No	No	直接将记录插入到叶子节点
Yes	No	1）拆分 Leaf Page 2）将中间的节点放入到 Index Page 中 3）小于中间节点的记录放左边 4）大于等于中间节点的记录放右边
Yes	Yes	1）拆分 Leaf Page 2）小于中间节点的记录放左边 3）大于等于中间节点的记录放右边 4）拆分 Index Page 5）小于中间节点的记录放左边 6）大于中间节点的记录放右边 7）中间节点放入上一层 Index Page

通过实例来分析 B+ 树的插入。对于图 9-9 中的这棵 B+ 树，我们插入 28 这个键值，发现当前 Leaf Page 和 Index Page 都没有满，直接插入就可以了，如图 9-10 所示。

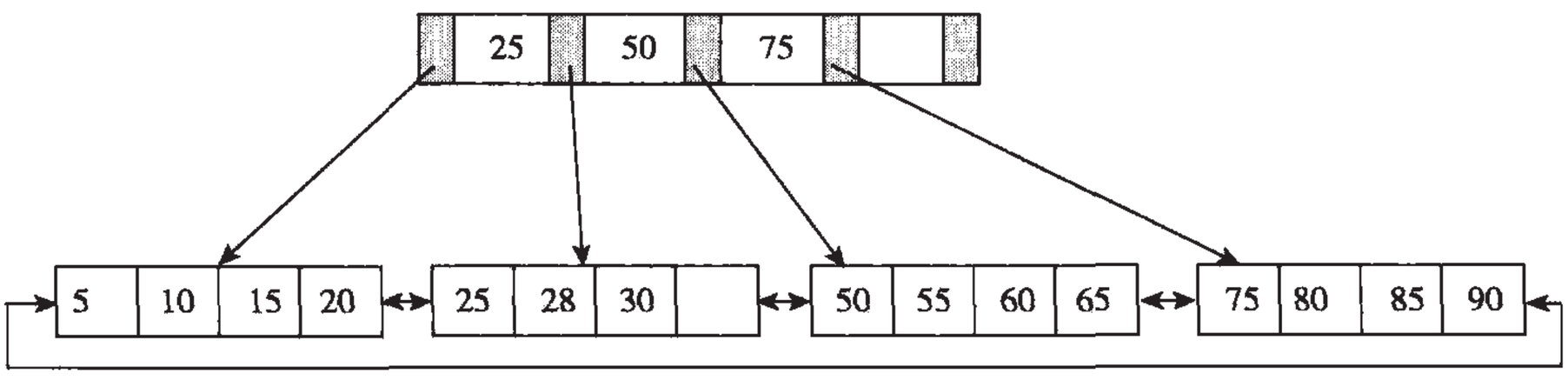


图 9-10 插入键值 28

再插入 70 这个键值，这时原先的 Leaf Page 已经满了，但是 Index Page 还没有满，符合表 9-1 的第二种情况，将 70 插入 Leaf Page 后的情况为 50、55、60、65、70。我们根据中间值 60 拆分叶子节点，可得图 9-11。

由于版面的关系，这次没有能在各叶子节点间标上双向链表指针。不过和图 9-9、图 9-10 一样，这个指针还是存在的。最后插入记录 95，这符合表 9-1 讨论的第三种情况，即 Leaf Page 和 Index Page 都满了，这时需要做两次拆分，如图 9-12 所示。

可以看到，不管怎么变化，B+ 树总会保持平衡，但是为了保持平衡需要在插入新的键值后做大量的拆分页（split）操作，而 B+ 树主要用于磁盘，因此页的拆分意味着磁盘的操作，应该在可能的情况下减少页的拆分。为此，B+ 树提供了旋转（rotation）的功能。

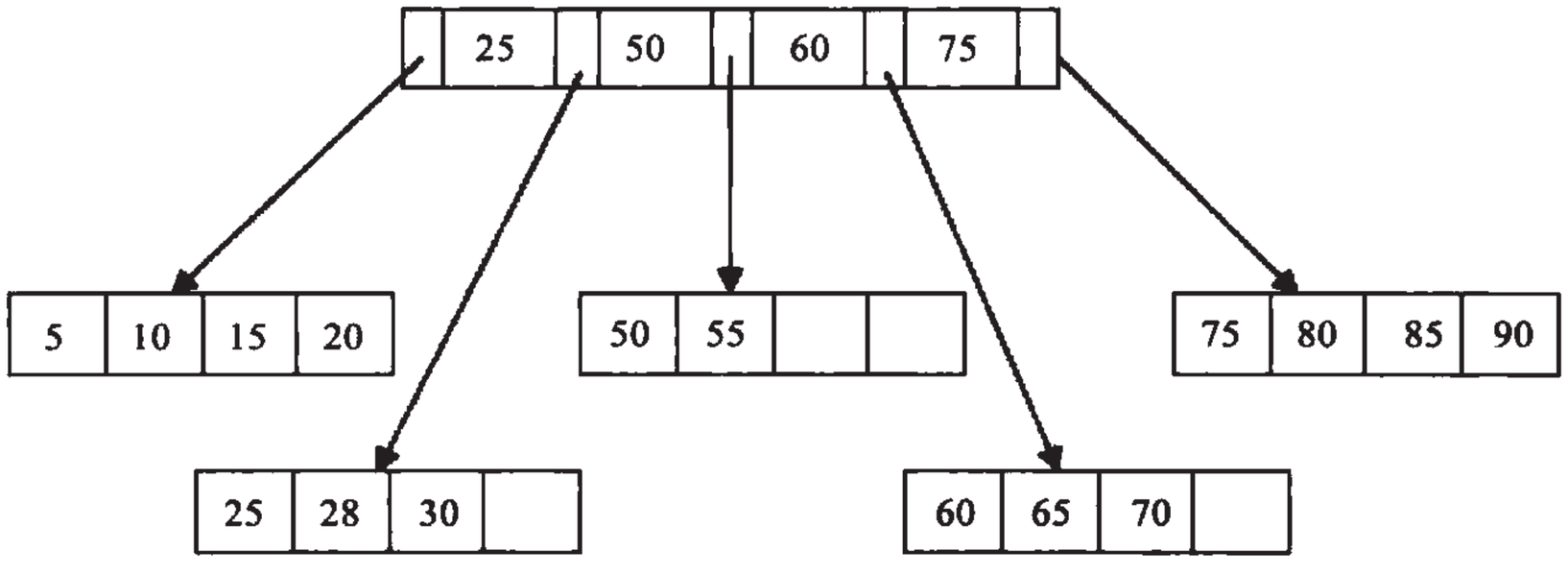


图 9-11 插入键值 70

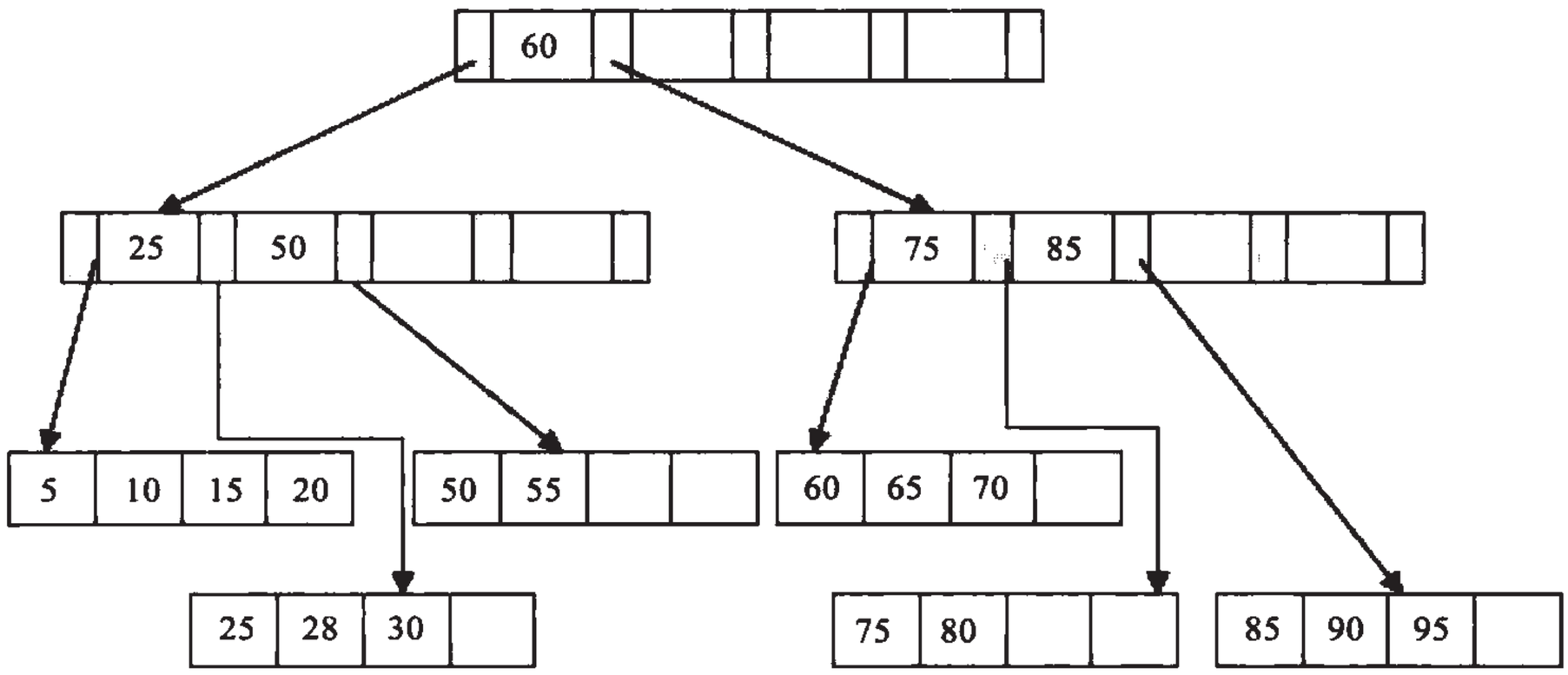


图 9-12 插入键值 95

旋转发生在 Leaf Page 已经满而其左右兄弟节点没有满的情况下。这时 B+ 树并不会急于进行拆分页的操作，而是将记录移到所在页的兄弟节点上。通常先判断左兄弟是否被用来做旋转操作，因此对于图 9-10 的情况，在插入键值 70 时，B+ 树并不会急于拆分叶子节点，而是去做旋转操作，如图 9-13 所示。

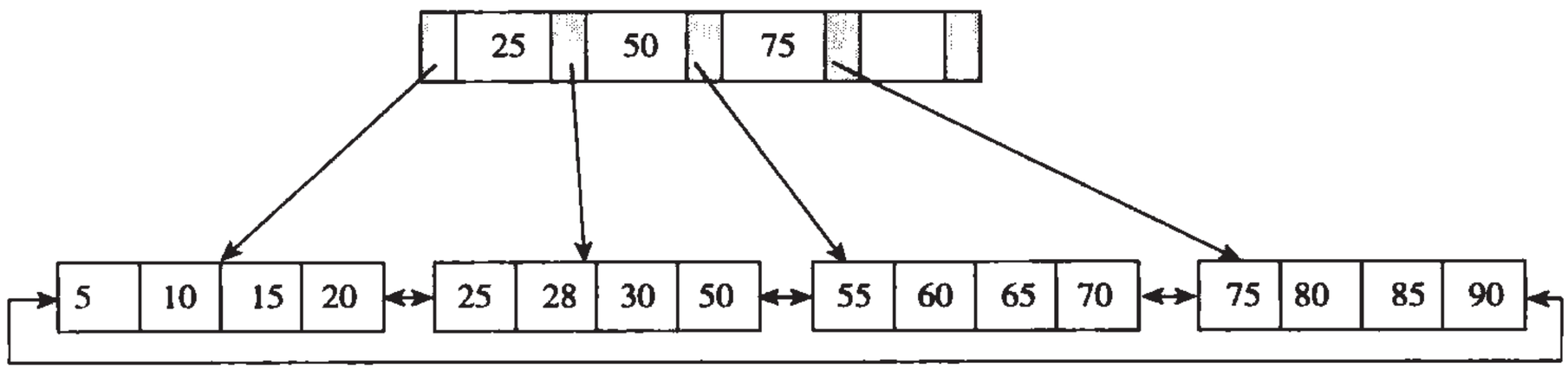


图 9-13 B+ 树的旋转操作

从图 9-13 可以看到，采用旋转操作使 B+ 树减少了一次页的拆分操作，这时 B+ 树的高度依然还是 2。

9.3.2 B+ 树的删除操作

B+ 树使用填充因子（fill factor）来控制树的删除变化，填充因子可设的最小值是50%。B+ 树的删除操作同样必须保证删除后叶子节点中的记录依然按序排列。同插入一样，B+ 树的删除操作同样需要考虑以下 3 种情况。与插入不同的是，删除根据填充因子的变化来衡量。

表 9-2 B+ 树删除操作

叶子节点小于填充因子	中间节点小于填充因子	操作
No	No	直接将记录从叶子节点删除，如果该节点还是 Index Page 的节点，用该节点的右节点代替
Yes	No	合并叶子节点和他的兄弟节点，同时更新 Index Page
Yes	Yes	1）合并叶子节点和他的兄弟节点 2）更新 Index Page 3）合并 Index Page 和他的兄弟节点

我们对图 9-12 的 B+ 树来进行删除操作。首先删除键值为 70 的这条记录，这符合表 9-2 讨论的第一种情况，删除后可得图 9-14 所示的 B+ 树。

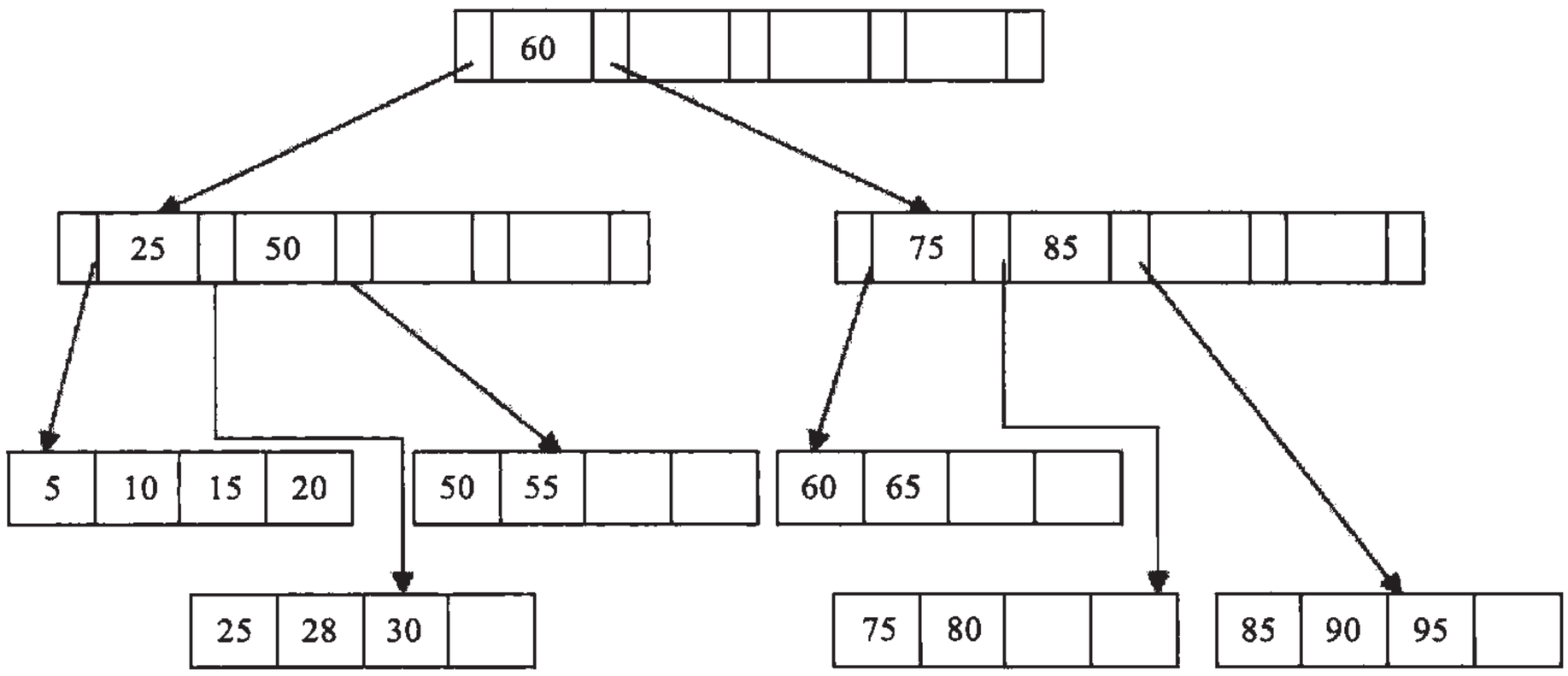


图 9-14 删除键值 70

接着删除键值为 25 的记录，这也符合表 9-2 讨论的第一种情况，但是该值还是 Index Page 中的值，因此在删除 Leaf Page 中的 25 后，还应将 25 的右兄弟节点 28 更新到 Page Index 中，最后可得图 9-15。

最后来看删除键值 60 的情况。删除 Leaf Page 中键值为 60 的记录后，填充因子小于 50%，这时需要做合并操作，同样，在删除 Index Page 中相关记录后需要做 Index Page 的合并操作，最后得图 9-16。

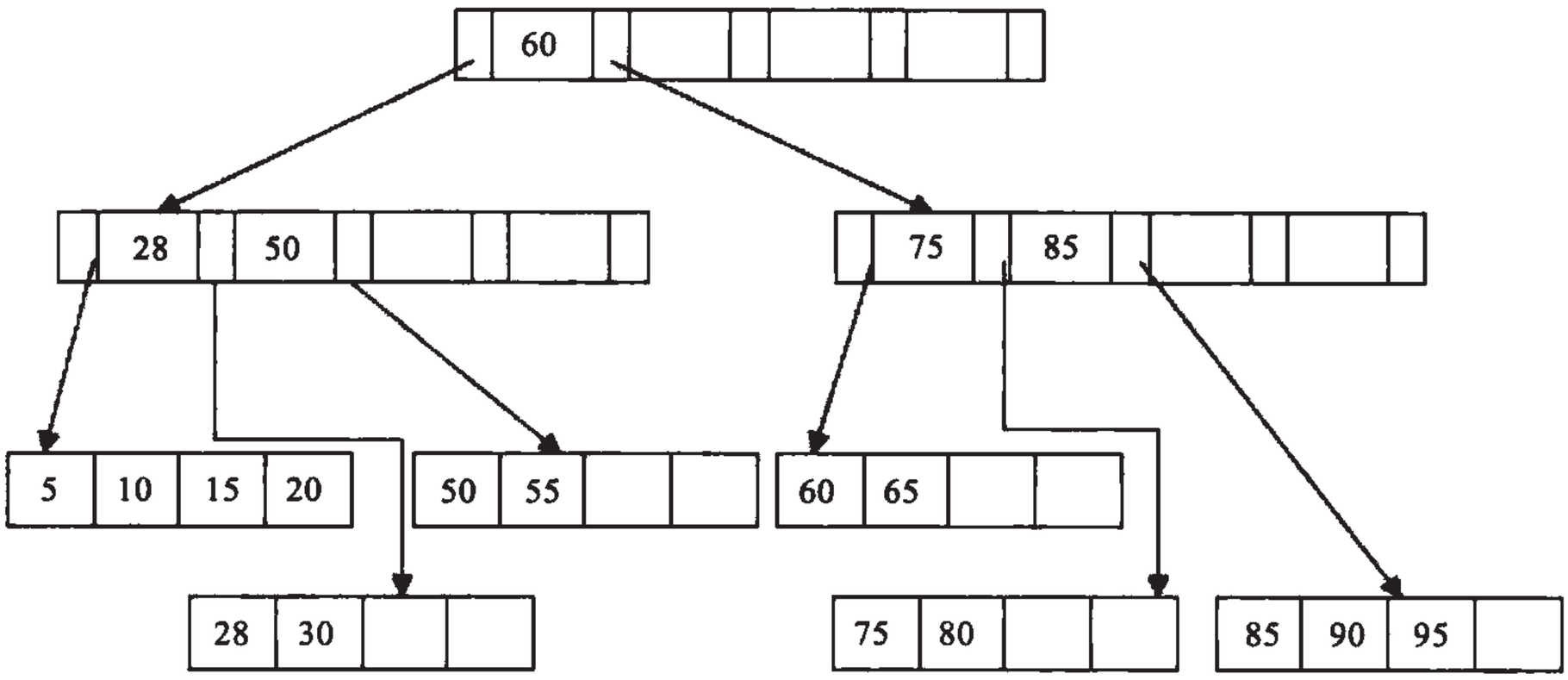


图 9-15 删除键值 25

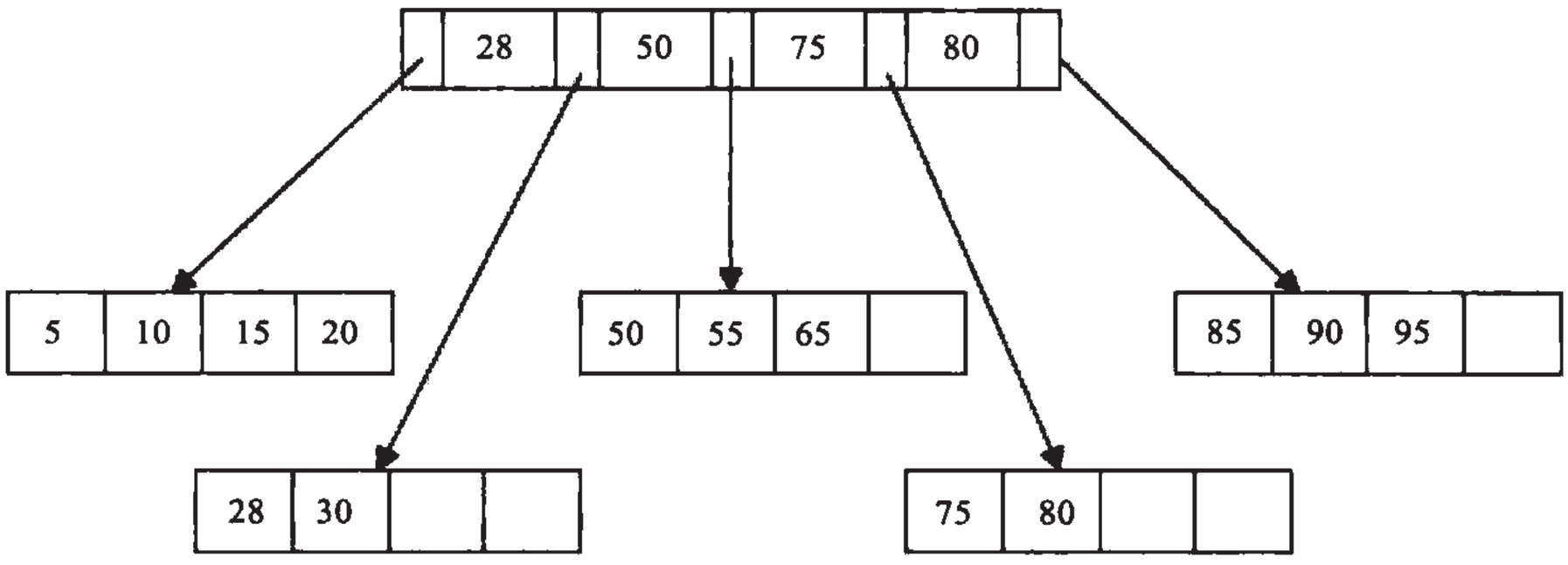


图 9-16 删除键值 60

9.4 B+ 树索引

前面讨论的都是 B+ 树的数据结构及其一般操作，B+ 树索引的本质就是 B+ 树在数据库中的实现，而 B+ 树索引在数据库中的一个特点就是高扇出性。例如在 InnoDB 存储引擎中，每个页的大小为 16KB。

因此在数据库中，B+ 树的高度一般都在 2 ～ 4 层，这意味着查找某一键值最多只需要 2 到 4 次 IO 操作，这还不错。因为现在一般的磁盘每秒至少可以做 100 次 IO 操作，2 ～ 4 次的 IO 操作意味着查询时间只需 0.02 ～ 0.04 秒。

在 MySQL 数据库中，索引是在存储引擎层实现的，这意味着每个引擎的 B+ 树索引的实现方式可能是不同的，取决于存储引擎实现的本身。

B+ 树索引可以分为聚集索引与辅助索引（非聚集索引），但是这两者本身都与之前讨论的 B+ 树的数据结构一样，区别仅在于所存放数据的内容。

9.4.1 InnoDB B+ 树索引

InnoDB 存储引擎是索引组织表（Index Organized Table，IOT），也就是说数据文件本身就是按照 B+ 树方式存放数据的。其中，B+ 树的键值为主键，若在建立时没有显式地指定主键，则 InnoDB 存储引擎会自动创建一个 6 字节的列作为主键。因此在 InnoDB 存储引擎中，可以将 B+ 树索引分为聚集索引（clustered index）和辅助索引（secondary index）。无论是何种索引，每个页的大小都为 16KB，且不能更改。

聚集索引是根据主键创建的一棵 B+ 树，聚集索引的叶子节点存放了表中的所有记录。辅助索引是根据索引键创建的一棵 B+ 树，与聚集索引不同的是，其叶子节点仅存放索引键值，以及该索引键值指向的主键。也就是说，如果通过辅助索引来查找数据，那么当找到辅助索引的叶子节点后，很有可能还需要根据主键值查找聚集索引来得到数据，这种查找方式又被称为书签查找（bookmark lookup）。因为辅助索引不包含行记录的所有数据，这就意味着每页可以存放更多的键值，因此其高度一般都要小于聚集索引。

注意 若辅助索引是一个包含主键的联合索引，那么并不需要一个额外的列来存放主键值。辅助索引会选择通过联合索引中的主键进行查找。

图 9-17 显示了在 InnoDB 存储引擎中聚集索引和辅助索引的关系。

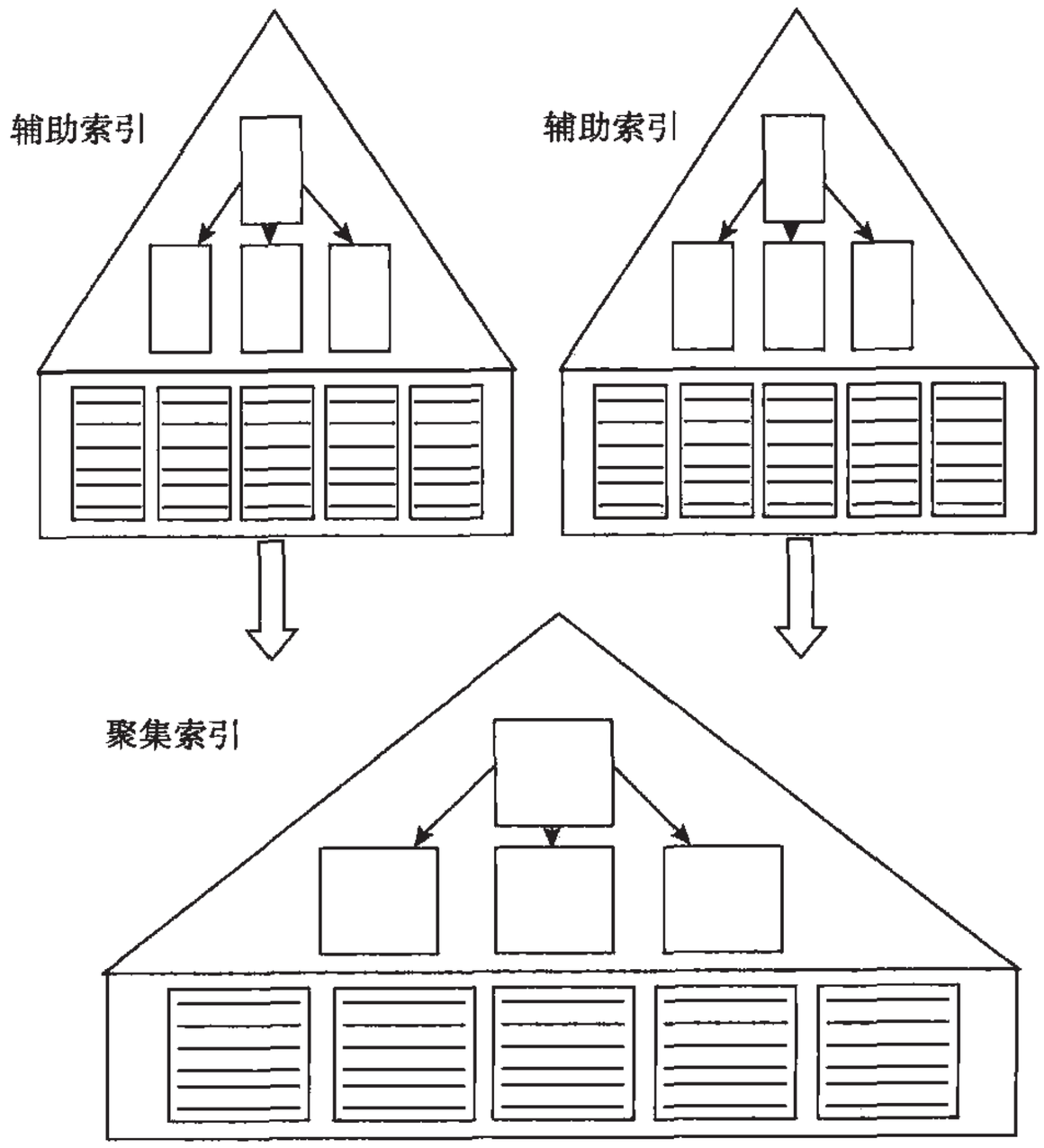


图 9-17 InnoDB 存储引擎中聚集索引与辅助索引之间的关系

接着来看一张表，这里以人为方式使其每个页只能存放两个行记录，例如：

```
CREATE TABLE t(
    a INT NOT NULL ,
    b VARCHAR(8000),
    c INT NOT NULL,
    PRIMARY KEY (a),
    KEY idx_c (c)
)ENGINE=INNODB;

INSERT INTO t SELECT 1,REPEAT('a',7000),-1;
INSERT INTO t SELECT 2,REPEAT('a',7000),-2;
INSERT INTO t SELECT 3,REPEAT('a',7000),-3;
INSERT INTO t SELECT 4,REPEAT('a',7000),-4;
```

图 9-18 是该表的聚集索引和辅助索引 idx_c 的关系。

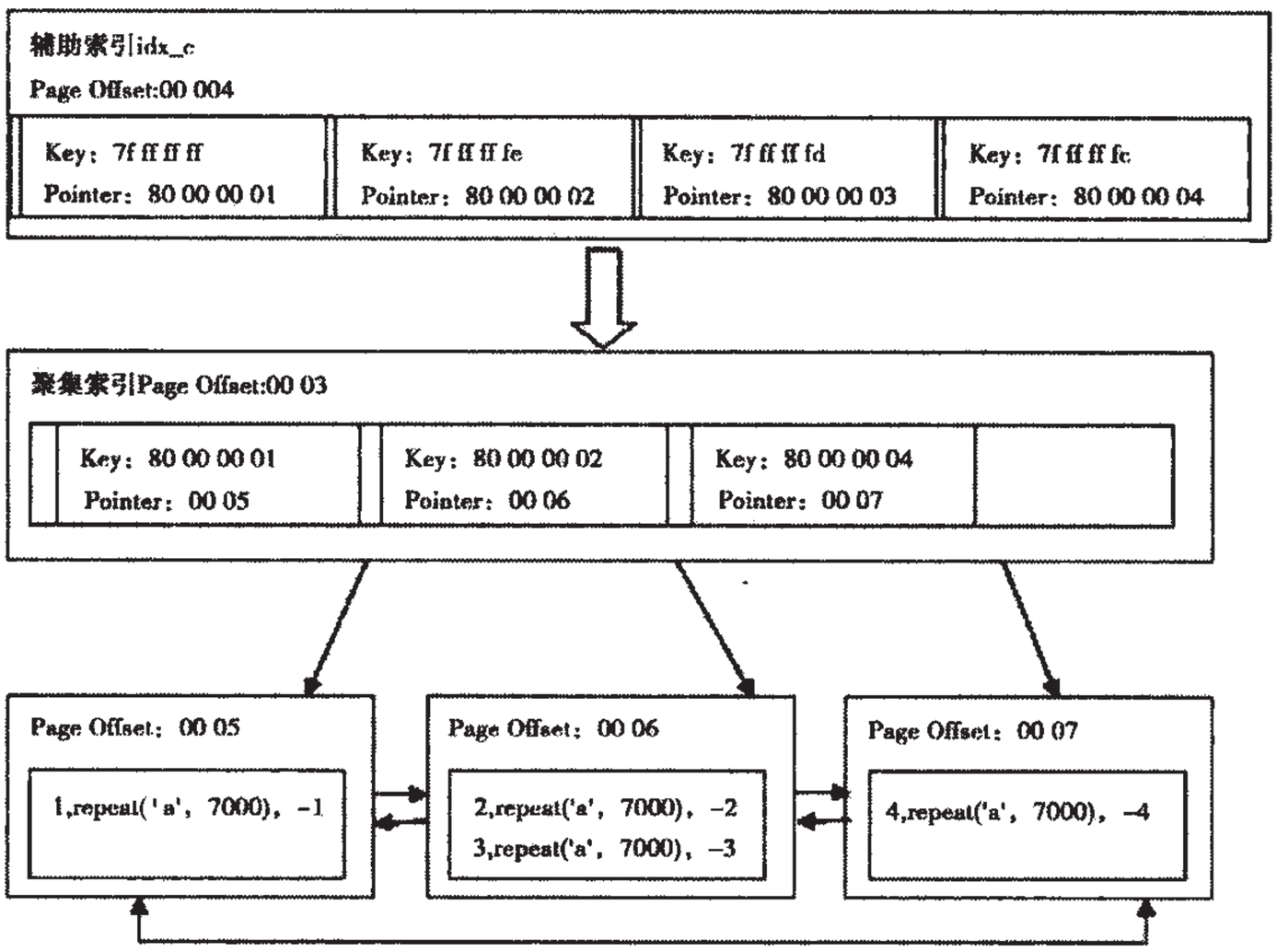


图 9-18 聚集索引与辅助索引之间的联系

可见辅助索引 idx_c 是一棵高度为 1 的 B+ 树，叶子节点存放了所有的索引键值及其对应的主键（−1，1），（−2，2），（−3，3），（−4，4）。由于 c 列是有符号的整型，因此在数据库内部用 0x7FFFFFFF 来表示 −1。聚集索引是一棵高度为 2 的 B+ 树，叶子节点存放了所有的记录（1，repeate('a',7000)，−1）、（2，repeat（'a'，7000），−2）、（3，repeat（'a'，7000），−3）、（4，repeat（'a'，7000），−4）。

注意 《MySQL 技术内幕：InnoDB 存储引擎》一书中对 InnoDB 存储引擎的 B+ 树索引有更为详细和深入的分析，强烈建议高级 DBA 和开发人员阅读此书，这对更好地理解 InnoDB 存储引擎的实现有着极大的帮助。

9.4.2 MyISAM B+ 树索引

MyISAM 存储引擎其实更像一张堆表，所有的行数据都存放于 MYD 文件中，其 B+ 树索引都是辅助索引，存放于 MYI 文件中。PRIMARY KEY 索引和其他索引不同之处在于其必须是唯一的，并且不可为 NULL 值。其索引页的大小默认为 1KB，同样不可以进行调整（MariaDB 支持通过变量 myisam_block_size 来对页的大小进行修改）。此外，与 InnoDB 存储引擎不同的是，因为没有聚集索引，其索引叶节点存放的键值不是主键值，而是在 MYD 文件中的物理位置。图 9-19 显示了在 MyISAM 存储引擎中索引与文件之间的关系。

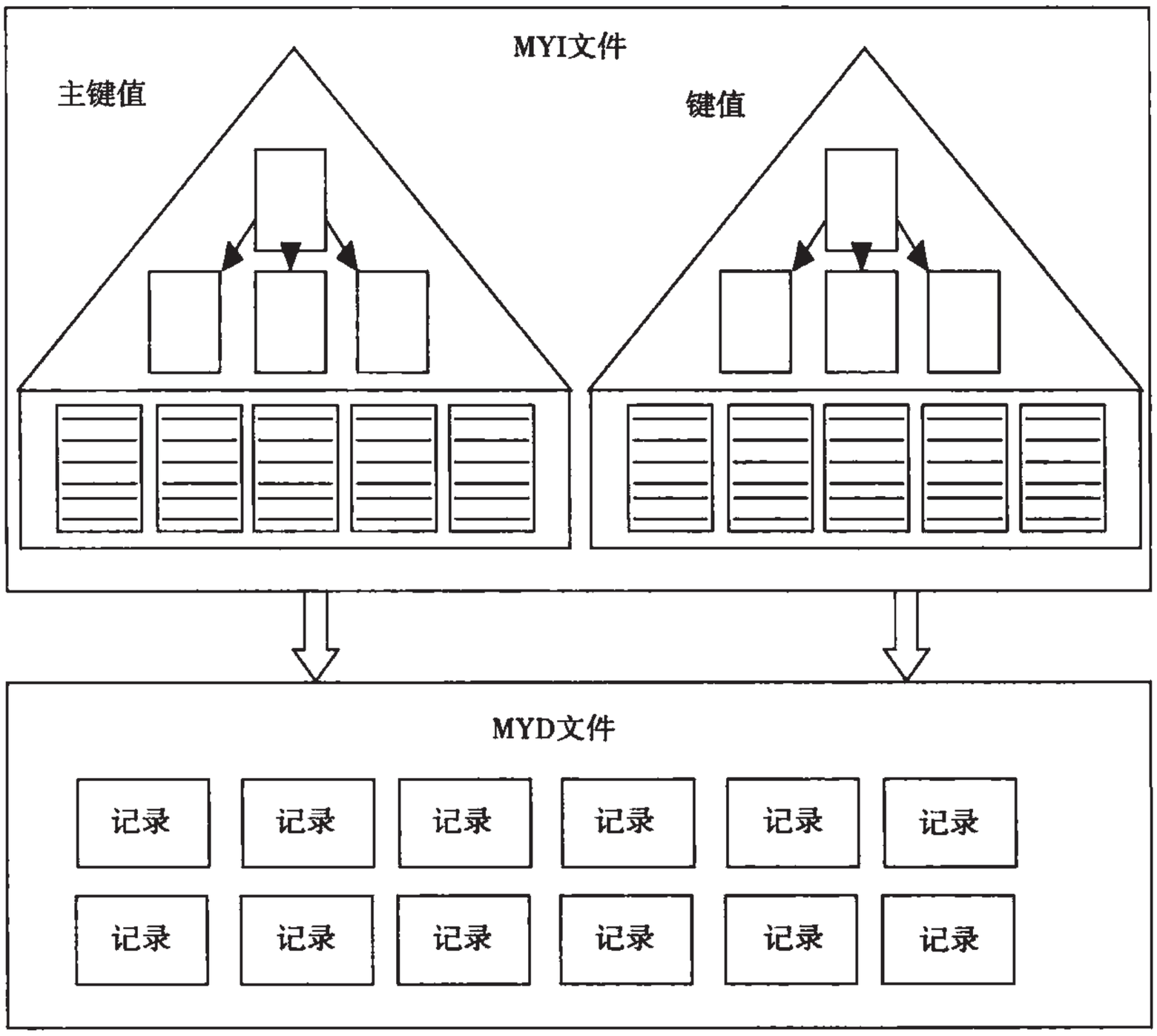


图 9-19 MyISAM 存储引擎 MYD 与 MYI 之间的关系

9.5 Cardinality

9.5.1 什么是 Cardinality

并不是所有在查询条件中出现的列都需要添加索引。对于什么时候添加 B+ 树索引，一般的经验是，在访问表中很少一部分行时使用 B+ 树索引才有意义。对于性别字段、地区字

段、类型字段，它们可取值的范围很小，称为低选择性，例如：

```
SELECT * FROM student WHERE sex='M'
```

按性别进行查询时，可取值的范围一般只有“M”和“F”，因此上述 SQL 语句得到的结果可能是该表 50% 的数据（我们假设男女比例 1:1），这时添加 B+ 树索引是完全没有必要的。相反，如果某个字段的取值范围很广，几乎没有重复，即是高选择性的，那么此时使用 B+ 树索引是最适合的。例如姓名字段，基本上在每一个应用中都不允许出现重名。

怎样查看索引是否是高选择性的呢？可以通过 SHOW INDEX 语句中的 Cardinality 列来观察。Cardinality 值非常关键，表示索引中唯一只记录数量的预估值。这里需要注意的是，Cardinality 是一个预估值，而不是一个准确值，用户也不可能得到一个准确的值。在实际应用中，Cardinality/n_rows_in_table 应尽可能接近 1，如果非常小，那么需要考虑是否还要建这个索引。因此在访问高选择性属性的字段，并从表中取出很少一部分数据时，对这个字段添加 B+ 树索引是非常有必要的，例如：

```
SELECT * FROM member WHERE usernick = 'David'
```

表 member 大约有 500 万行数据，usernick 字段上有一个唯一的索引。这时如果查找用户名为 David 的用户时，得到执行计划如下：

```
mysql>explain select * from member where usernick='David'\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: member
         type: const
possible_keys: usernick
          key: usernick
      key_len: 62
          ref: const
         rows: 1
        Extra:
1 row in set (0.00 sec)
```

可以看到使用了 usernick 这个索引，这符合我们前面提到的高选择性，选取表中很少的行的原则。

9.5.2 InnoDB 存储引擎怎样统计 Cardinality

上一小节介绍了 Cardinality 的重要性，并且告诉读者 Cardinality 表示索引的选择性。建立的索引是高选择性的这对数据库来说才是有意义的，但是数据库是怎样来统计 Cardinality 信息的呢？因为 MySQL 数据库中有各种不同的存储引擎，而每种存储引擎对 B+ 树索引的

实现又各不相同，所以对 Cardinality 的统计放在存储引擎中。

需要考虑到的是，在生产环境中，索引的更新操作可能非常频繁。如果在每次索引发生更新操作时就对其进行 Cardinality 的统计，那么将会给数据库带来很大的负担。另外需要考虑的是，如果一张表的数据量非常大，比如一张表有 50GB 的数据，那么统计一次 Cardinality 信息所需要的时间可能非常长。这在生产环境的应用中也是不能接受的。因此，数据库对于 Cardinality 的统计都是通过采样（sample）的方法来完成的。

在 InnoDB 存储引擎中，Cardinality 统计信息的更新发生在两个操作中：INSERT 和 UPDATE。根据前面的叙述，不可能在每次发生 INSERT 和 UPDATE 时都去更新 Cardinality 的信息，这会增加数据库系统的负荷，同时对大表进行统计时，时间上也不允许。因此 InnoDB 存储引擎对于更新 Cardinality 信息的策略为：

- 表中 1/16 的数据已发生变化。
- stat_modified_counter > 2 000 000 000。

第一种策略为自上次统计 Cardinality 信息后，表中 1/16 的数据已经发生变化。这时需要更新 Cardinality 信息。第二种策略考虑的是，如果对表中某一行数据频繁地进行更新操作，这时表中的数据实际并没有增加，发生变化的还是这一行数据，那么第一种更新策略就无法适用，故在 InnoDB 存储引擎内部有一个计数器 stat_modified_counter，用来表示发生变化的次数，当计数器的值大于 2 000 000 000 时，同样需要更新 Cardinality 信息。

接着考虑 InnoDB 存储引擎内部是怎样进行 Cardinality 信息的统计和更新操作的呢？同样是通过采样的方法。InnoDB 存储引擎只对 8 个叶节点进行采样。采样的过程为：

- 取得 B+ 树索引中叶节点的数量，即为 A。
- 随机取 B+ 树索引中的 8 个叶节点。统计每个页不同记录的个数，即为 P1，P2，...，P8。
- 根据采样信息给出 Cardinality 的预估值：Cardinality=（P1+P2+...+P8）*A/8。

通过上述的说明可以发现，在 InnoDB 存储引擎中，Cardinality 的值是通过对 8 个叶节点预估而得到的，不是一个精确的值。再者，每次对于 Cardinality 值的统计都是通过随机读取 8 个叶子节点得到的，这又暗示了另一个 Cardinality 的现象，即每次得到的 Cardinality 值可能是不同的。例如：

```
SHOW INDEX FROM OrderDetails
```

上述这句 SQL 语句会触发 MySQL 数据库对于 Cardinality 值的统计，第一次运行得到的结果如图 9-20 所示。

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment
orderdetails	0	PRIMARY	1	OrderID	A	2032				BTREE		
orderdetails	0	PRIMARY	2	ProductID	A	2032				BTREE		
orderdetails	1	OrderID	1	OrderID	A	2032				BTREE		
orderdetails	1	OrdersOrder_Details	1	OrderID	A	2032				BTREE		
orderdetails	1	ProductID	1	ProductID	A	156				BTREE		
orderdetails	1	ProductsOrder_Details	1	ProductID	A	156				BTREE		

图 9-20　第一次运行得到的结果

在上述测试过程中，并没有 INSERT、UPDATE 这类操作来改变表 OrderDetails 中的内容，但是当第二次运行 SHOW INDEX FROM 语句时，Cardinality 的值还是发生了如图 9-21 所示的变化。

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment
orderdetails	0	PRIMARY	1	OrderID	A	2192				BTREE		
orderdetails	0	PRIMARY	2	ProductID	A	2192				BTREE		
orderdetails	1	OrderID	1	OrderID	A	2192				BTREE		
orderdetails	1	OrdersOrder_Details	1	OrderID	A	2192				BTREE		
orderdetails	1	ProductID	1	ProductID	A	168				BTREE		
orderdetails	1	ProductsOrder_Details	1	ProductID	A	168				BTREE		

图 9-21 第二次运行得到的结果

可以看到，在第二次运行 SHOW INDEX FROM 语句时，虽然表 OrderDetails 本身并没有发生任何的变化，但是表 OrderDetails 中索引的 Cardinality 值都发生了变化。由于 Cardinality 是对随机的 8 个叶子节点进行分析，因此即使表没有发生变化，用户观察到的索引 Cardinality 值还是会发生变化的，这本身并不是 InnoDB 存储引擎的 Bug，只是随机采用数据导致的结果。

当然，可能会有用户每次观察到的索引 Cardinality 值都是一样的情况，那是因为表足够小，表的叶子节点小于或等于 8 个。这时即使随机采样，也总是会采取到这些页，因此每次得到的 Cardinality 值都相同。

9.6 B+ 树索引的使用

9.6.1 不同应用中 B+ 树索引的使用

在了解了 B+ 树索引的本质和实现后，下一个问题需要考虑怎样正确地使用 B+ 树索引？这不是一个简单的问题。笔者所总结的可能并不适用于所有的应用场，这里只是概括出一个大概的方向，在实际的生产环境中，每个 DBA 和开发人员还要根据具体应用来使用索引，观察索引使用的情况，判断是否需要添加索引。“Think Different”，不要盲从任何人给你的经验意见。

根据第 1 章的介绍，我们知道数据库中存在两种类型的应用：OLTP 和 OLAP 应用。在 OLTP 应用中，查询操作只从数据库中取得一小部分数据，一般都在 10 条记录以下，甚至大多数情况只获取 1 条记录，如根据主键值来取得用户信息、根据订单号取得订单的详细信息，这都是典型 OLTP 应用的查询语句。在这种情况下，建立 B+ 树索引后，对该索引的使用应该只是通过该索引取得表中小部分的数据。这时建立 B+ 树索引才是有意义的，否则即使建立了，优化器也可能选择不使用索引。

对于 OLAP 应用，情况可能稍显复杂一些。不过概括来说，在 OLAP 应用中都需要访问

表中大量的数据，并根据这些数据来产生查询的结果，而这些查询多是面向分析的查询，目的是为决策者提供支持。比如这个月每个用户的消费情况、销售额同比或环比增长的情况。因此在 OLAP 中添加索引依据的是宏观的信息，而不是微观信息，这是因为最终要得到的结果是提供给决策者的。例如不需要在 OLAP 中对姓名字段进行索引，因为很少会对单个用户进行查询。但是对于 OLAP 中的复杂查询，需要涉及多张表之间的联接操作，这时索引的添加是有意义的。如果联接操作使用 Hash Join，那么索引可能又变得不是非常重要了，所以 DBA 或开发人员要认真并仔细地研究自己的应用。不过在 OLAP 应用中，通常会需要对时间字段进行索引，这是因为大多数统计需要根据时间维度来进行判断。

9.6.2 联合索引

联合索引是指对表上的多个列进行索引。前面讨论的情况都是只对表中的一个列进行索引。联合索引的创建方法与单个索引创建的方法一样，不同之处仅在于有多个索引列。下面的语句创建了一张 t 表，其中索引 idx_a_b 是联合索引，联合的列为（a，b）。

```
CREATE TABLE t (
   a INT,
   b INT,
   PRIMARY KEY (a),
   KEY idx_a_b ( a,b)
)ENGINE=INNODB
```

何时需要使用联合索引呢？在讨论这个之前，先来看一下联合索引内部的结果。从本质上来说，联合索引还是一棵 B+ 树，不同的是联合索引的键值数量不是 1，而是大于等于 2。我们来讨论由两个整型列组成的联合索引，假定两个键值的名称分别为 a、b，如图 9-22 所示。

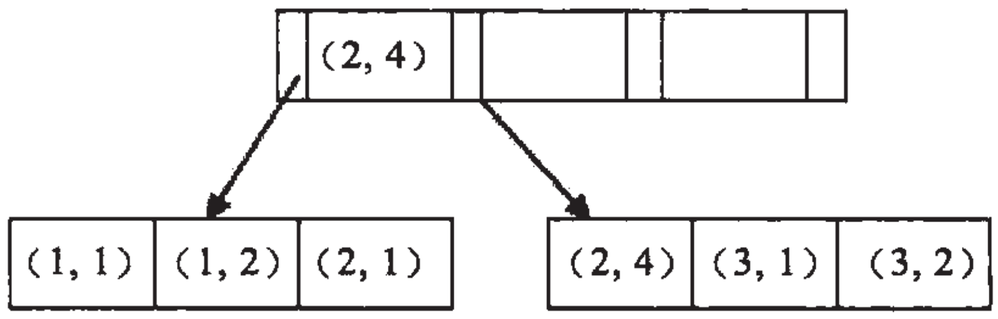


图 9-22 多个键值的 B+ 树

从图 9-22 中可以观察多个键值的 B+ 树情况。和之前讨论的单个键值的 B+ 树并没有什么不同，键值都是排序的，通过叶子节点可以逻辑上顺序地读出所有数据，这里是：(1,1)，(1,2)，(2,1)，(2,4)，(3,1)，(3,2)。数据按（a,b）的顺序进行了存放。

因此，对于查询 SELECT * FROM TABLE WHERE a=xxx and b=xxx，显然是可以使用（a,b）这个联合索引的。对于单个的 a 列查询 SELECT * FROM TABLE WHERE a=xxx 也可以使用这个（a,b）索引。但是对于 b 列的查询 SELECT * FROM TABLE WHERE b=xxx，不可以使用这棵 B+ 树索引。可以看到叶子节点上的 b 值为 1、2、1、4、1、2，显然不是顺序

的，因此对于 b 列的查询不能使用（a,b）的索引。

联合索引的另一个好处是可以对第二个键值进行排序。例如，在很多情况下我们都需要查询某个用户的购物情况，并按照时间排序，取出最近三次的购买记录，这时使用联合索引可以减少一次排序操作，因为索引本身在叶子节点已经排序了。来看一个例子，先根据如下语句来创建测试表 buy_log。

```
CREATE TABLE buy_log (
   userid INT UNSIGNED NOT NULL,
   buy_date DATE
)ENGINE=InnoDB;

INSERT INTO buy_log VALUES ( 1,'2009-01-01');
INSERT INTO buy_log VALUES ( 2,'2009-01-01');
INSERT INTO buy_log VALUES ( 3,'2009-01-01');
INSERT INTO buy_log VALUES ( 1,'2009-02-01');
INSERT INTO buy_log VALUES ( 3,'2009-02-01');
INSERT INTO buy_log VALUES ( 1,'2009-03-01');
INSERT INTO buy_log VALUES ( 1,'2009-04-01');

ALTER TABLE buy_log ADD KEY ( userid );
ALTER TABLE buy_log ADD KEY ( userid,buy_date );
```

本例中建立了两个索引来进行比较。两个索引都包含了 userid 字段。如果只对于 userid 进行查询，例如：

```
SELECT * FROM buy_log WHERE userid=2;
```

则优化器的选择如图 9-23 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	buy_log	ref	userid,userid_2	userid	4	const	1	

图 9-23 查询条件仅为 userid 的执行计划

从图 9-23 中可以发现，possible_keys 列有两个索引可供使用，分别是单个的 userid 索引和（userid，buy_date）联合索引。优化器最终的选择是 userid，因为该叶子节点包含单个键值，所以一个页能存放的记录应该更多。

接着假定要取出 userid 为 1 的最近 3 次的购买记录，其 SQL 语句如下，执行计划如图 9-24 所示。

```
SELECT * FROM buy_log
   WHERE userid=1 ORDER BY buy_date DESC LIMIT 3
```

同样地，对于上述的 SQL 语句既可以使用 userid 索引，也可以使用（userid，buy_date）索引。这次优化器使用了（userid，buy_date）的联合索引 userid_2，因为在这个联合索引中 buy_date 已经排序好了。根据该联合索引取出数据，无需再对 buy_date 做一次额外的排序操

作。若强制使用 userid 索引，则执行计划如图 9-25 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	buy_log	ref	userid,userid_2	userid_2	4	const	3	Using where; Using index

图 9-24　SQL 语句的执行计划

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	buy_log	ref	userid	userid	4	const	3	Using where; Using filesort

图 9-25　强制使用 userid 索引的执行计划

在 Extra 列可以看到 Using filesort，即需要额外的一次排序才能完成查询。而这次需要排序的显然是 buy_date，因为索引 userid 中的 buy-date 是未排序的。

9.6.3　覆盖索引

InnoDB 存储引擎支持覆盖索引（covering index），或称索引覆盖（index coverage），即从辅助索引中就可以得到查询的记录，而不需要查询聚集索引中的记录。使用覆盖索引的好处是辅助索引不包含整行记录的所有信息，故其大小要远小于聚集索引，因此可以减少大量的 IO 操作。

注意　*覆盖索引技术最早在 InnoDB Plugin 中完成并实现。这意味着对于 InnoDB 版本小于 1.0 的，或者 MySQL 数据库版本为 5.0 或以下的，InnoDB 存储引擎不支持覆盖索引特性。*

对于 InnoDB 存储引擎的辅助索引而言，由于其中包含了主键信息，因此其叶子节点存放的数据为（primary key1，primary key2，...，key1，key2，...）。下面语句都可以仅使用一次辅助联合索引来完成查询。

```
SELECT key2 FROM table WHERE key1=xxx;
SELECT primary key2,key2 FROM table WHERE key1=xxx;
SELECT primary key1,key2 FROM table WHERE key1=xxx;
SELECT primary key1,primary key2, key2 FROM table WHERE key1=xxx;
```

覆盖索引的另一个好处是对于某些统计问题，辅助索引小于聚集索引，可以减少 IO 操作。还是通过上一小节创建的表 buy_log 进行说明，如果要进行如下的查询：

```
SELECT COUNT(*) FROM buy_log ;
```

这时 InnoDB 存储引擎并不会选择通过查询聚集索引来进行统计，因为该表上还有辅助索引，同样的理由，因为辅助索引小于聚集索引，可以减少 IO 操作，故优化器的选择如图 9-26 所示。

通过图 9-26 可以看到，possible_keys 列为 NULL，但是实际执行时优化器却选择了

userid 索引，而 Extra 列的 Using index 就是代表了优化器进行了覆盖索引操作。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	buy_log	index	NULL	userid	4	NULL	7	Using index

图 9-26 COUNT(*) 操作的执行计划

此外，一般来说对于诸如（a，b）这样的联合索引，一般不可以选择 b 列作为查询条件。但是对于统计操作，如果是覆盖索引的，则优化器会进行选择，例如：

```
SELECT COUNT(*) FROM buy_log
   WHERE buy_date>='2011-01-01' AND buy_date<'2011-02-01'
```

表 buy_log 中有（userid，buy_date）的联合索引，这里只根据 b 列进行条件查询，一般情况下是不能使用该联合索引的，但是这句 SQL 查询是统计操作，并且可以利用覆盖索引的信息，因此优化器会选择该联合索引，其执行计划如图 9-27 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	buy_log	index	NULL	userid_2	8	NULL	7	Using where; Using index

图 9-27 利用覆盖索引执行统计操作

从图 9-27 可以发现，possible_keys 列依然为 NULL，但是 key 列为 userid_2，即表示（userid，buy_date）的联合索引。从 Extra 列同样可以发现 Using index 提示，表示为覆盖索引。

9.6.4 优化器选择不使用索引的情况

在某些情况下，当执行 EXPLAIN 时，会发现优化器并没有选择索引去查找数据，而是通过扫描聚集索引，也就是直接进行全表的扫描来得到数据。这种情况多发生于范围查找、JOIN 操作等，例如：

```
SELECT * FROM orderdetails
   WHERE orderid>10000 and orderid<102000;
```

上述 SQL 语句查找订单号大于 10 000 的订单详情，通过命令 SHOW INDEX FROM orderdetails 可观察到的索引如图 9-28 所示。

	Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment
▶	orderdetails	0	PRIMARY	1	OrderID	A	2311				BTREE	
	orderdetails	0	PRIMARY	2	ProductID	A	2311				BTREE	
	orderdetails	1	OrderID	1	OrderID	A	2311				BTREE	
	orderdetails	1	OrdersOrder_Details	1	OrderID	A	1155				BTREE	
	orderdetails	1	ProductID	1	ProductID	A	177				BTREE	
	orderdetails	1	ProductsOrder_Details	1	ProductID	A	177				BTREE	

图 9-28 表 orderdetails 的索引详情

可以看到表 orderdetails 有（OrderID，ProductID）的联合主键，此外还有对列 OrderID

的单个索引。这句 SQL 显然是可以通过 OrderID 来查找数据的。然而通过 EXPLAIN 命令，会发现优化器并没有按照 OrderID 上的索引来查找数据，如图 9-29 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	orderdetails	range	PRIMARY,OrderID,OrdersOrder_Details	PRIMARY	4	NULL	1155	Using where

图 9-29　上述范围查询的 SQL 执行计划

在 possible_keys 列可以看到查询可以使用 PRIMARY、OrderID、OrdersOrder_Details 三个索引，但是在最后的索引中，优化器选择了 PRIMARY 聚集索引，而非 OrderID 辅助索引。

这是为什么呢？原因在于我们要选取的数据是整行信息，而 OrderID 索引不能覆盖到我们要查询的信息，因此在对 OrderID 索引进行查询到指定数据的操作后，还需要进行一次书签访问来查找整行数据的信息。虽然在 OrderID 的索引中数据是顺序存放的，但是再一次的书签查找数据是无序的，因此变为了磁盘上的离散读取操作。如果要求访问的数据量很小，那么优化器还是会选择辅助索引，但是当访问的数据占整个表中数据的很大一部分时（一般是 20% 左右），优化器会选择通过聚集索引来查找数据。在 9.1 节中已经提到顺序读取要远远快于离散读取。图 9-30 演示的是用 sysbench 测试的顺序读取和随机读取的速度对比，同时对比了 RAID 开启 Write Back 和 Write Through 的性能差异。测试的磁盘是由 4 块 15 000 转的硬盘组成的 RAID10。测试文件大小为 2GB，页大小 64KB。

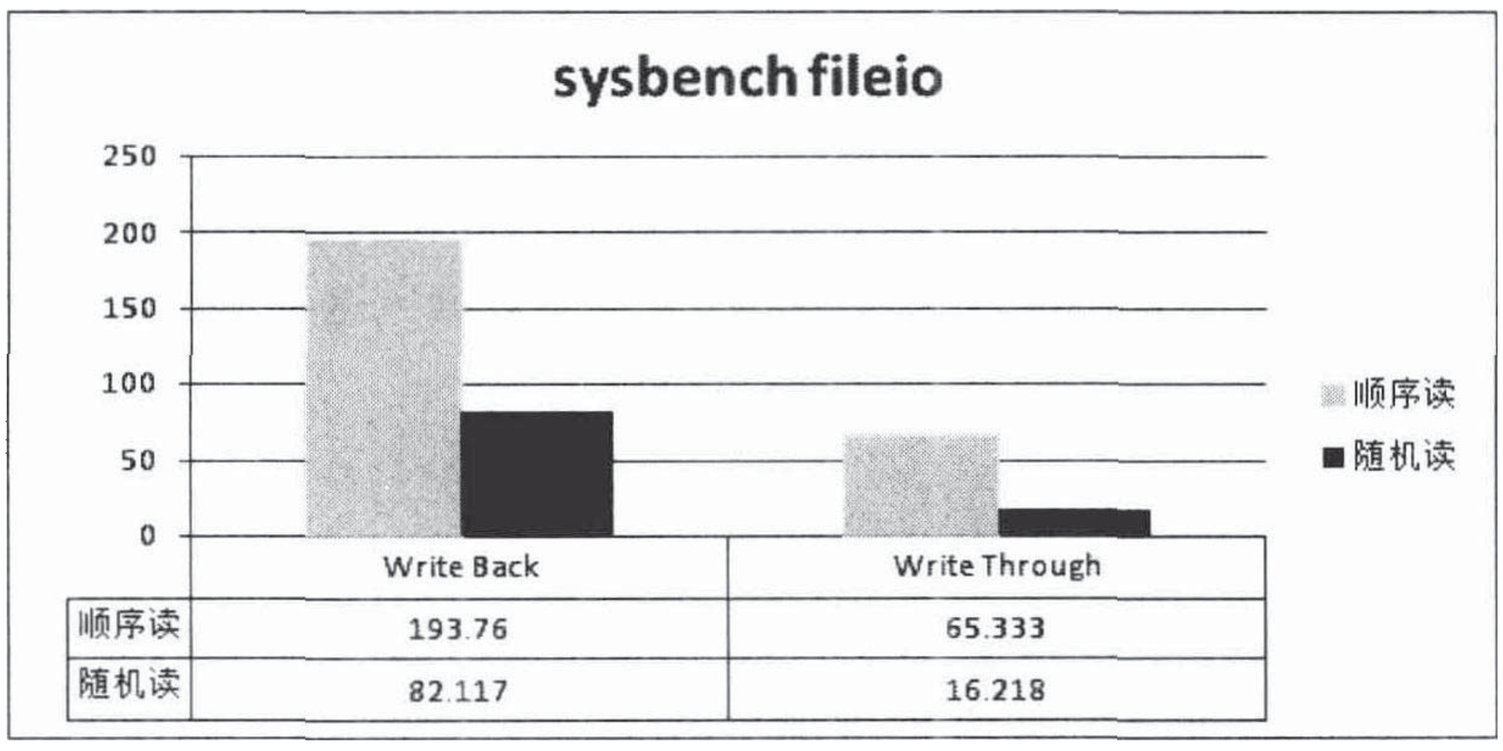


图 9-30　顺序读取与随机读取的性能对比

从图 9-30 可以发现，不管是否开启 RAID 卡的 Write Back 功能，磁盘的随机读取性能都远远小于顺序读取的性能。通过图 9-30 的比较也大致可以知道 Write Back 相对于 Write Through 的性能提升。

因此对于不能进行索引覆盖的情况，优化器选择辅助索引的情况是，通过辅助索引查找的数据是少量的，这是由当前传统机械硬盘的特性所决定，即利用顺序读取来替换随机读取的查找。若用户使用的磁盘是固态硬盘，随机读取操作非常快，同时有足够的自信来确认使

用辅助索引可以带来更好的性能，那么可以使用关键字 FORCE INDEX 来强制使用某个索引，例如：

```
SELECT * FROM orderdetails FORCE INDEX(OrderID)
WHERE orderid>10000 and orderid<102000;
```

这时的执行计划如图 9-31 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	orderdetails	range	OrderID	OrderID	4	NULL	943	Using where

图 9-31 强制使用辅助索引

9.6.5 INDEX HINT

MySQL 数据库支持 INDEX HINT（索引提示），显式地告诉优化器使用哪个索引。以下两种情况可能需要用到 INDEX HINT。

- MySQL 数据库的优化器错误地选择了某个索引，导致 SQL 语句运行得很慢。这种情况非常少见。优化器在绝大部分情况下工作都非常有效和正确。这时有经验的 DBA 或开发人员可以强制优化器使用某个索引，以达到提高 SQL 运行速度的目的。
- 某 SQL 语句可以选择的索引非常多，这时优化器选择执行计划时间的开销可能会大于 SQL 语句本身。例如，优化器分析 Range 查询本身就是比较耗时的操作，这时 DBA 或开发人员分析最优的索引选择，通过 INDEX HINT 来强制使优化器不进行各个执行路径的成本分析，直接执行选择指定的索引来完成查询。

在 MySQL 数据库中 INDEX HINT 的语法如下：

```
tbl_name [[AS] alias] [index_hint_list]
index_hint_list:
index_hint [, index_hint] ...
index_hint:
USE {INDEX|KEY}
[{FOR {JOIN|ORDER BY|GROUP BY}] ([index_list])
| IGNORE {INDEX|KEY}
[{FOR {JOIN|ORDER BY|GROUP BY}] (index_list)
| FORCE {INDEX|KEY}
[{FOR {JOIN|ORDER BY|GROUP BY}] (index_list)
index_list:
index_name [, index_name] ...
```

接着来看一个例子，先根据如下清单创建测试表 t，并填充相应数据。

```
CREATE TABLE t (
   a INT,
   b INT,
   KEY (a) ,
```

```
   KEY (b)
)ENGINE=INNODB;

INSERT INTO t SELECT 1,1;
INSERT INTO t SELECT 1,2;
INSERT INTO t SELECT 2,3;
INSERT INTO t SELECT 2,4;
INSERT INTO t SELECT 1,2;
```

然后执行如下的 SQL 语句：

```
SELECT * FROM t WHERE a=1 AND b = 2;
```

通过 EXPLAIN 命令得到如图 9-32 所示的执行计划。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	t	index_merge	a,b	b,a	5,5	NULL	1	Using intersect(b,a); Using where; Using index

图 9-32　SQL 语句的执行计划

图 9-32 中的 possible_keys 列显示了上述 SQL 语句可使用的索引为 a、b，而实际使用的索引为 key 列所示，同样为 a、b。也就是说，MySQL 数据库使用 a、b 两个索引来完成这一个查询。Extra 列提示的 Using intersect（b，a），表示根据两个索引得到的结果进行求交的数学运算，最后得到结果。

如果我们使用 USE INDEX 的 INDEX HINT 来使用 a 这个索引，如：

```
SELECT * FROM t USE INDEX(a) WHERE a=1 AND b = 2;
```

那么得到的执行计划如图 9-33 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	t	ALL	a	NULL	NULL	NULL	5	Using where

图 9-33　使用 USE INDEX 后的执行计划

从图 9-33 可以看到，虽然我们指定了使用 a 索引，但是优化器实际采用的是表扫的方式。因此，USE INDEX 只是告诉优化器可以选择该索引，而实际上优化器还是会根据自己的判断进行选择。如果使用 FORCE INDEX 的 INDEX HINT，例如：

```
SELECT * FROM t FORCE INDEX(a) WHERE a=1 AND b = 2;
```

那么这时的执行计划如图 9-34 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	t	ref	a	a	5	const	3	Using where

图 9-34　使用 FORCE INDEX 后的执行计划

从图 9-34 可以看到，这时优化器的最终选择和用户指定的索引是一致的。因此，如果用户确定指定某个索引来完成查询，那么最可靠的是使用 FORCE INDEX，而不是 USE INDEX。

9.7 Multi-Range Read

MySQL 数据库 5.6 版本开始支持 Multi-Range Read（MRR）优化。MRR 优化的目的就是减少磁盘的随机访问，并且将随机访问转化为较为顺序的数据访问，可为 IO-bound 类型的 SQL 查询语句带来性能的极大提升。MRR 优化适用于 range、ref 和 eq_ref 类型的查询。

MRR 优化有以下几个好处：

- 使得数据访问变得较为顺序。在查询辅助索引时，先对得到的查询结果按照主键进行排序，并按照主键排列的顺序进行书签查找。
- 减少缓冲池中页被替换的次数。
- 批量处理对键值的查询操作。

对于 InnoDB 和 MyISAM 存储引擎的范围查询和联接查询，MRR 的工作方式如下：

- 将查询得到的辅助索引键值存放于一个缓存中，这时缓存中的数据是根据辅助索引键值排序的。
- 将缓存中的键值根据 RowID 进行排序。
- 根据 RowID 的排序顺序来访问实际的数据文件。

此外，若 InnoDB 存储引擎或 MyISAM 存储引擎的缓冲池不是足够大，即不能存放下一张表中的所有数据，此时频繁的离散读取操作还会导致将缓存中的页替换出缓冲池，然后又不断地读入缓冲池。若按照主键顺序进行访问，则可以将重复行为的次数降为最低。例如下面的 SQL 语句：

```
SELECT * FROM salaries WHERE salary>10000 AND salary<40000;
```

salary 上有一个辅助索引 idx_s，因此除了通过辅助索引查找键值外，还需要通过书签查找来进行对整行数据的查询。当不启用 MRR 特性，看到的执行计划如图 9-35 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	salaries	range	idx_s	idx_s	4	NULL	23378	Using index condition

图 9-35 不启用 MRR 的执行计划

若启用 MRR 特性，则除了会在 Extra 列看到 Using index condition 外，还会看见 Using MRR 选项，如图 9-36 所示。

在实际的执行中会体会到以上两个执行时间差别非常大，如表 9-3 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	salaries	range	idx_s	idx_s	4	NULL	23378	Using index condition; Using MRR

图 9-36　启用 Multi-Range Read 的执行计划

表 9-3　是否启用 MRR 的执行时间对比

是否使用 MRR	执行时间（秒）	是否使用 MRR	执行时间（秒）
不使用 MRR	43.213	使用 MRR	4.212

在笔者的笔记本电脑上，上述两个语句的执行时间相差近 10 倍，可见 MRR 将访问数据转化为顺序的之后查询性能提高了。

注意　*上述测试都是在 MySQL 数据库启动后直接执行 SQL 查询语句，在这种情况下需确保缓冲池中没有被预热，以及需要查询的数据并不包含在缓冲池中。*

此外，MRR 还可以将某些范围查询拆分为键值对，以此来完成批量的数据查询。这样做的好处是可以在拆分过程中，直接过滤一些不符合查询条件的数据，例如：

```
SELECT * FROM t
  WHERE key_part1 >= 1000 AND key_part1 < 2000
  AND key_part2 = 10000;
```

表 t 中有（key_part1，key_part2）的联合索引，因此索引根据 key_part1、key_part2 的位置关系进行排序。若没有 MRR，此时查询类型为 Range，SQL 优化器会先将 key_part1 大于 1000 小于 2000 的数据都取出，就是使 key_part2 不等于 10 000，待取出行数据后再根据 key_part2 的条件进行过滤，这会导致无用数据被取出。如果存在大量的数据并且其 key_part2 不等于 10 000，则启用 MRR 优化性能会有巨大的提升。

倘若启用了 MRR 优化，那么优化器会先将查询条件进行拆分，然后再进行数据的查询。就上述查询语句而言，优化器会将查询条件拆分为（1000,10 000），（1001，10 000），（1002，10 000），...，（1999，10 000），最后再根据这些拆分出的条件进行数据查询。

可以来看一个实际的例子，查询如下：

```
SELECT * FROM salaries
WHERE (from_date between '1986-01-01' AND '1995-01-01')
AND (salary between 38000 and 40000);
```

若启用 MRR 优化，则执行计划如图 9-37 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	salaries	range	idx_s	idx_s	4	NULL	210740	Using index condition; Using MRR

图 9-37　启用 MRR 的执行计划

从图 9-37 中可以看出表 salaries 上有对 salary 的索引 idx_s。在执行上述 SQL 语句时，因为启用了 MRR 优化，所以会对查询条件进行拆分，在 Extra 列中可以看到 Using MRR 选项。

是否启用 MRR 优化可以通过参数 optimizer_switch 中的标记（flag）来控制。当启用 mrr 为 on 时，表示启用 MRR 优化。mrr_cost_based 标记表示是否通过 cost based 的方式来选择是否启用 mrr。若将 mrr 设为 on，mrr_cost_based 设为 off，则总是启用 MRR 优化。下面的语句可以将 MRR 优化总是设为开启状态。

```
mysql> SET @@optimizer_switch='mrr=on,mrr_cost_based=off';
Query OK, 0 rows affected (0.00 sec)
```

参数 read_rnd_buffer_size 用来控制键值的缓冲区大小，当大于该参数值时，执行器根据 RowID 对已经缓存的数据进行排序，并通过 RowID 来取得行数据。该值默认为 256KB。

```
mysql>SELECT @@read_rnd_buffer_size\G;
*************************** 1. row ***************************
@@read_rnd_buffer_size: 262144
1 row in set (0.00 sec)
```

9.8 Index Condition Pushdown

和 MRR 一样，Index Condition Pushdown（ICP）同样是 MySQL 5.6 开始支持的一种根据索引进行查询的优化方式。之前的 MySQL 版本不支持 ICP，当进行索引查询时，首先根据索引来查找记录，然后再根据 WHERE 条件来过滤记录。在支持 ICP 后，MySQL 数据库会在取出索引的同时，判断是否可以进行 WHERE 条件的过滤，即将 WHERE 的部分过滤操作放在了存储引擎层。在某些查询中，ICP 会大大减少上层 SQL 层对于记录的索取（fetch），从而提高数据库的整体性能。

ICP 优化支持 range、ref、eq_ref 和 ref_or_null 类型的查询，当前支持 MyISAM 和 InnoDB 存储引擎。当优化器选择 ICP 优化时，可在执行计划的 Extra 列看到 Using index condition 提示。

注意 NDB Cluster 存储引擎支持 Engine Condition Pushdown 优化。不仅支持索引的 Condition Pushdown，也可以支持非索引的 Condition Pushdown，不过这是由其引擎本身的特性所决定的。MySQL 5.1 版本 NDB Cluster 存储引擎就开始支持 Engine Condition Pushdown 优化。

假设某张表中有联合索引（zip_code，last_name，firset_name），并且查询语句如下：

```
SELECT * FROM people
  WHERE zipcode='95054'
  AND lastname LIKE '%etrunia%'
  AND address LIKE '%Main Street%';
```

对于上述语句，MySQL 数据库可以通过索引来定位 zipcode 等于 95 054 的记录，但是索引对 WHERE 条件的 `lastname LIKE '%etrunia%' AND address LIKE '%Main Street%'` 没有任何的帮助。若不支持 ICP 优化，则数据库需要先通过索引取出所有 zipcode 等于 95 054 的记录，然后再过滤 WHERE 之后的两个条件。

若支持 ICP 优化，则在取出索引时，就会进行 WHERE 条件的过滤，然后再去获取记录，这将极大提高查询的效率。当然，WHERE 可以过滤的条件是该索引可以覆盖到的范围。

再来看上一节中的 SQL 语句：

```
SELECT * FROM salaries
WHERE (from_date between '1986-01-01' AND '1995-01-01')
AND (salary between 38000 and 40000);
```

若不启用 MRR 优化，则其执行计划如图 9-38 所示。

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	salaries	range	idx_s	idx_s	4	NULL	210740	Using index condition

图 9-38　不进行 MRR 优化的执行计划

从图 9-38 中可以看到，Extra 列有 Using index condition 的提示。为什么这里 idx_s 索引会使用 ICP 优化呢？因为这张表的主键是（emp_no，from_date）的联合索引，idx_s 索引中包含了 from_date 的数据，所以可使用此优化方式。

表 9-4 对比了在 MySQL 5.5 和 MySQL 5.6 中上述 SQL 语句的执行时间，同时比较了开启 MRR 后查询消耗的时间。

表 9-4　SQL 语句的执行时间对比

MySQL 版本	执行时间（秒）
MySQL 5.5	46.738
MySQL 5.6 with ICP	37.924
MySQL 5.6 with ICP & MRR	7.816

在进行上述对执行时间的比较时同样不对缓冲池做任何的预热操作。可见 ICP 优化可以将查询效率在原有 MySQL 5.5 版本技术上提高了 23%。而再同时启用 MRR 优化后，性能还能有 400% 的提升！

9.9 T 树索引

9.9.1 T 树概述

对于 MySQL 数据库的 NDB Cluster 内存存储引擎，在使用它时可将其视为内存数据库（从 MySQL 5.1 开始 NDB Cluster 引擎可以将非索引数据存放在磁盘上，这种情况又可以视为混合存储引擎）。在内存数据库中，一般使用 T 树（T-Tree）作为其索引的数据结构。T 树是由平衡二叉树和 B 树发展而来。NDB Cluster 存储引擎中的 Order Index，也就是普通索引，使用的就是 T 树结构。T 树的好处是节点不存放数据，只存放指针，这样能减少对内存的使用，这对内存数据库来说显得尤为重要。同时 T 树也是一棵平衡二叉树，以此保证查找的性能。T 树中的 T 指的是 T 树中节点的形状。图 9-39 显示了 T 树节点（T-tree node）。

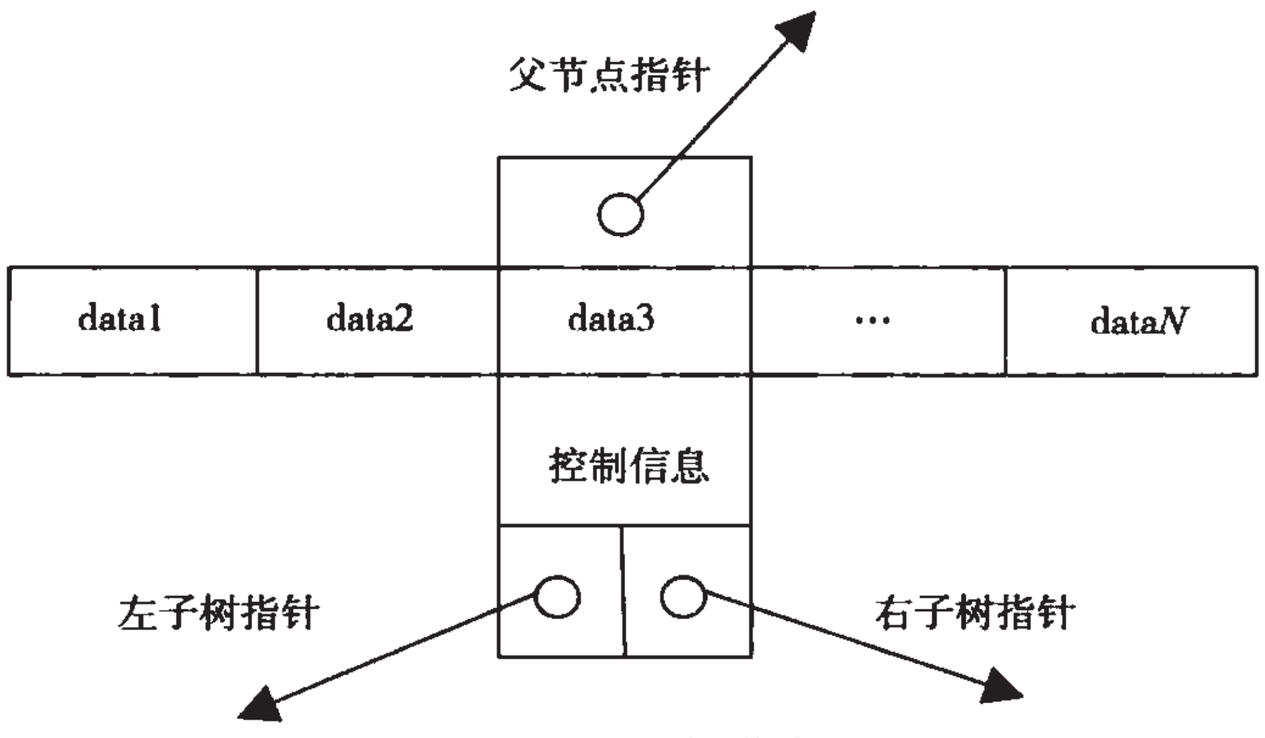


图 9-39 T 树节点

T 树节点由 3 个指针、一个有序数组（array），以及控制信息组成。3 个指针分别为指向父节点和左右子树的指针。有序数组保存的是数据指针，而非实际的数据。数组中的第一个数据称为最小元素（minimum element），最后一个数据称为最大元素（maximum element）。控制信息中存放了关于该 T 树节点的一些额外信息。T 树的结构如图 9-40 所示。

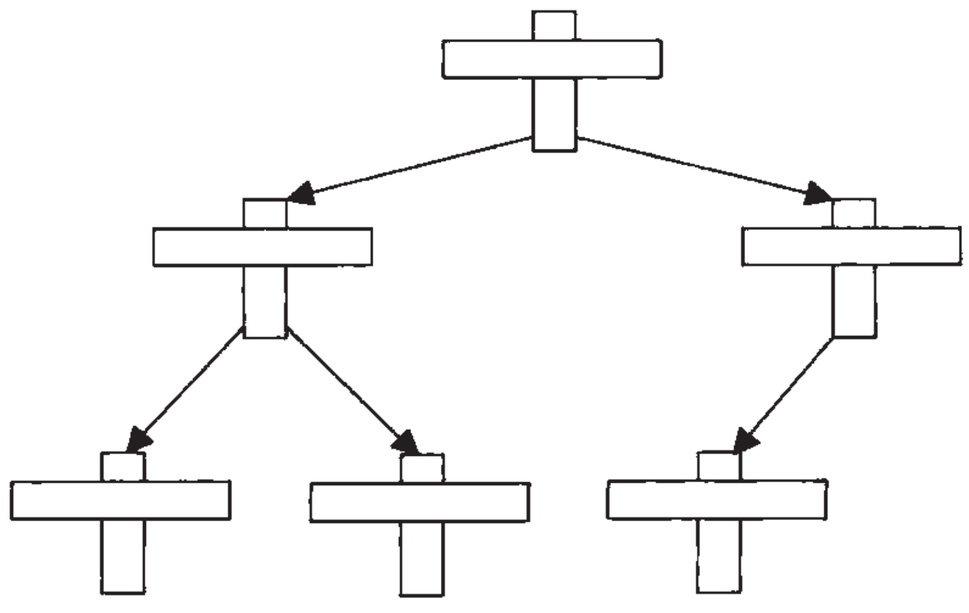


图 9-40 T 树结构

在 T 树中，含有左右子树的节点称为内部节点（internal node），没有子树的节点称为叶节点（leaf node），仅有一个子树的节点称为半叶节点（half-leaf node）。对于每个内部节点，存在一个相对应的叶节点或半叶节点存放内部节点的最小值，该值称为最大下界（greatest lower bound）。同时也存在相对应的叶节点或半叶节点用于存放内部节点的最大值，该值称为最小上界（least upper bound），如图 9-41 所示。如果一个值 *x* 在 T 树的节点 N 上，那么称 *x* 在 N 的边界内，N 为 *x* 的边界页。通过上述的内容，可以看出 T 树的节点借鉴了 B 树和平衡二叉树的思想。

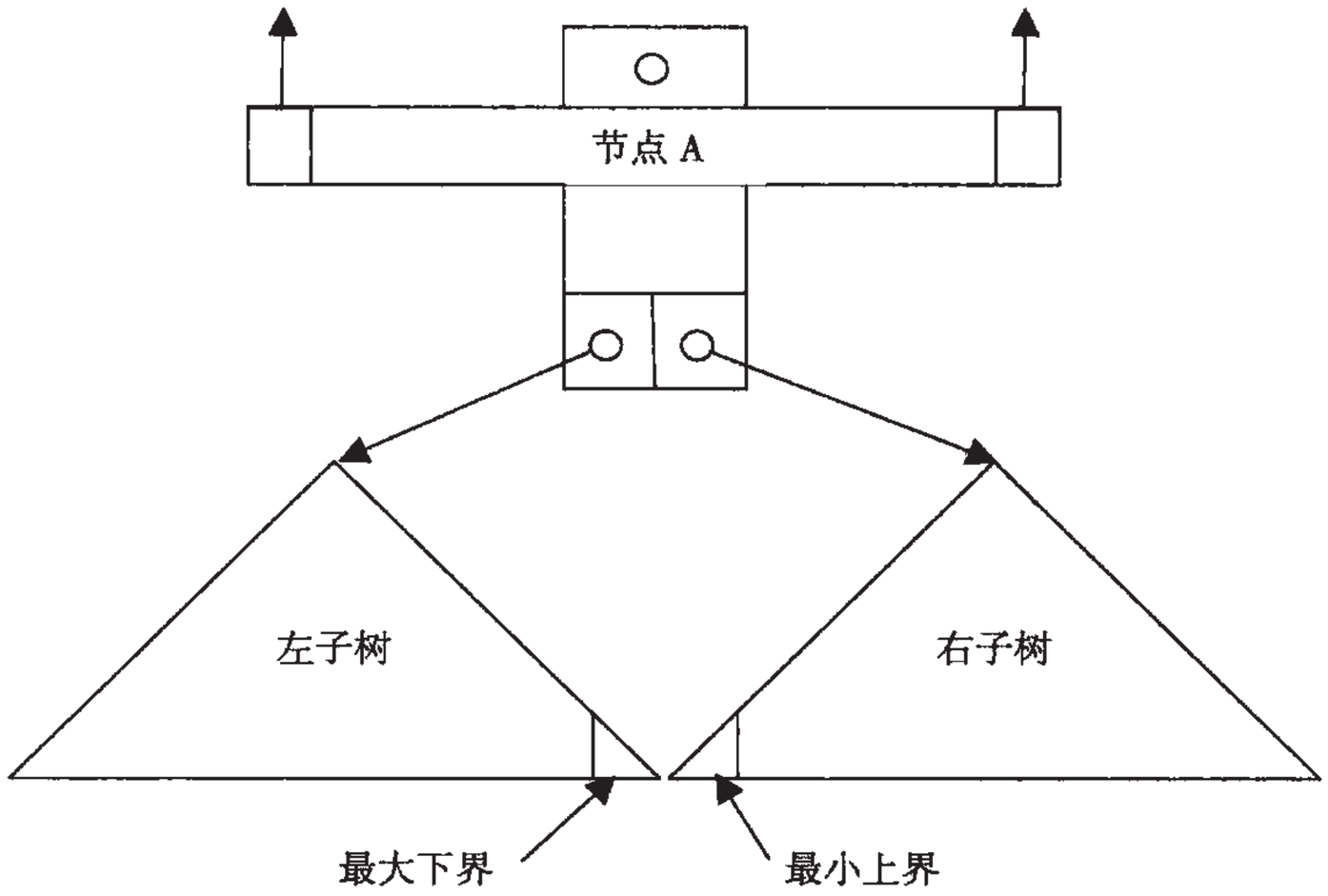


图 9-41　T 树的边界

9.9.2　T 树的查找、插入和删除操作

T 树的查找和二叉查找树类似，其查找算法为：

- 从根节点（root）开始查找，比较节点中最大值和最小值。如果查找的值在边界内，则用二叉法查找 T 树中的数组。
- 若查找的值比根节点的最小值小，则递归查找左子树。
- 若查找的值比根节点的最大值大，则递归查找右子树。
- 若不存在该值，返回为 NULL。

T 树的插入操作：先通过查找来定位边界页，新插入的值被插入边界页后需要对 T 树进行检查，看其是否平衡。如果不平衡，则需要通过旋转来保证 T 树的平衡。由于 T 树的节点可以存放多个值，因此其旋转的次数，较之平衡二叉树又减少了很多。

T 树的插入算法为：

- 查找边界页。
- 如果查到边界页，则判断是否有空间插入新的值。如果有，则直接插入，插入操作完成。如果没有空间，则删除该节点的最小值，插入新值。接着查询原插入值最大下界所在的节点，将删除的值插入该节点，并成为新的最大下界值。
- 如果没有查到边界页，那么在最后查询的节点中尝试插入记录。若该节点有空间可以插入，则直接插入，并且该记录成为该节点的最小或最大记录值。若没有空间插入，则分配一个叶节点进行插入。
- 如果分配了一个叶节点，则需要检查 T 树是否平衡。如果不平衡，即子树高度的差值大于 1，则需要进行与平衡二叉树同样的旋转操作，以保证 T 树重新回到平衡状态。当 T 树恢复到平衡状态，此次操作完成。

T 树的删除操作同插入操作类似，首先需要找到边界页，然后完成删除操作，最后判断是否需要旋转来完成平衡操作。其具体步骤如下：

- 查找删除记录所在的边界页。如果查找失败，则报告错误并停止操作。
- 如果删除不会引起下溢出（underflow）问题，即删除该记录后，节点中的记录数大于所要求该节点的最小记录数，那么直接删除记录。否则，如果节点是内部节点，那么删除该记录，同时将该节点的最大下界值放回该内部节点中。如果节点是叶节点或半叶节点，则直接删除记录（叶节点允许下溢出，半叶节点还需进行下一步操作）。
- 如果是半叶节点，则观察半叶节点是否可以和叶节点进行合并（merge）。如果可以，则强制将两个节点合并为一个节点，并删除半叶节点。
- 观察删除后的节点是否是空节点，若是则删除该节点。
- 观察 T 树是否平衡，若不平衡则通过旋转操作使 T 树重新回到平衡状态。

9.9.3 T 树的旋转

T 树的旋转操作和平衡二叉树十分相似。当插入或进行触发叶的添加或删除操作时，需要重新检查树是否平衡，需要判断是否通过旋转操作来使 T 树重新变为平衡。检查的路径为插入的节点到根节点所有的路径，观察左右子树的高度差值是否大于 1。若是，则需要执行旋转操作。对于插入操作，只需要进行一次旋转操作；对于删除操作，可能需要多次的旋转操作。图 9-42 显示了插入操作所触发的 LL 旋转和较为复杂的 LR 旋转操作。和平衡二叉树一样，共有四种类型的旋转，分为 LL、LR、RR、RL。其中 RR、RL 和 LL、LR 是对称的，这里不进行讨论。删除操作与插入操作一样，但是，删除操作使子树的高度变矮，而不是像插入操作一样使子树变高。

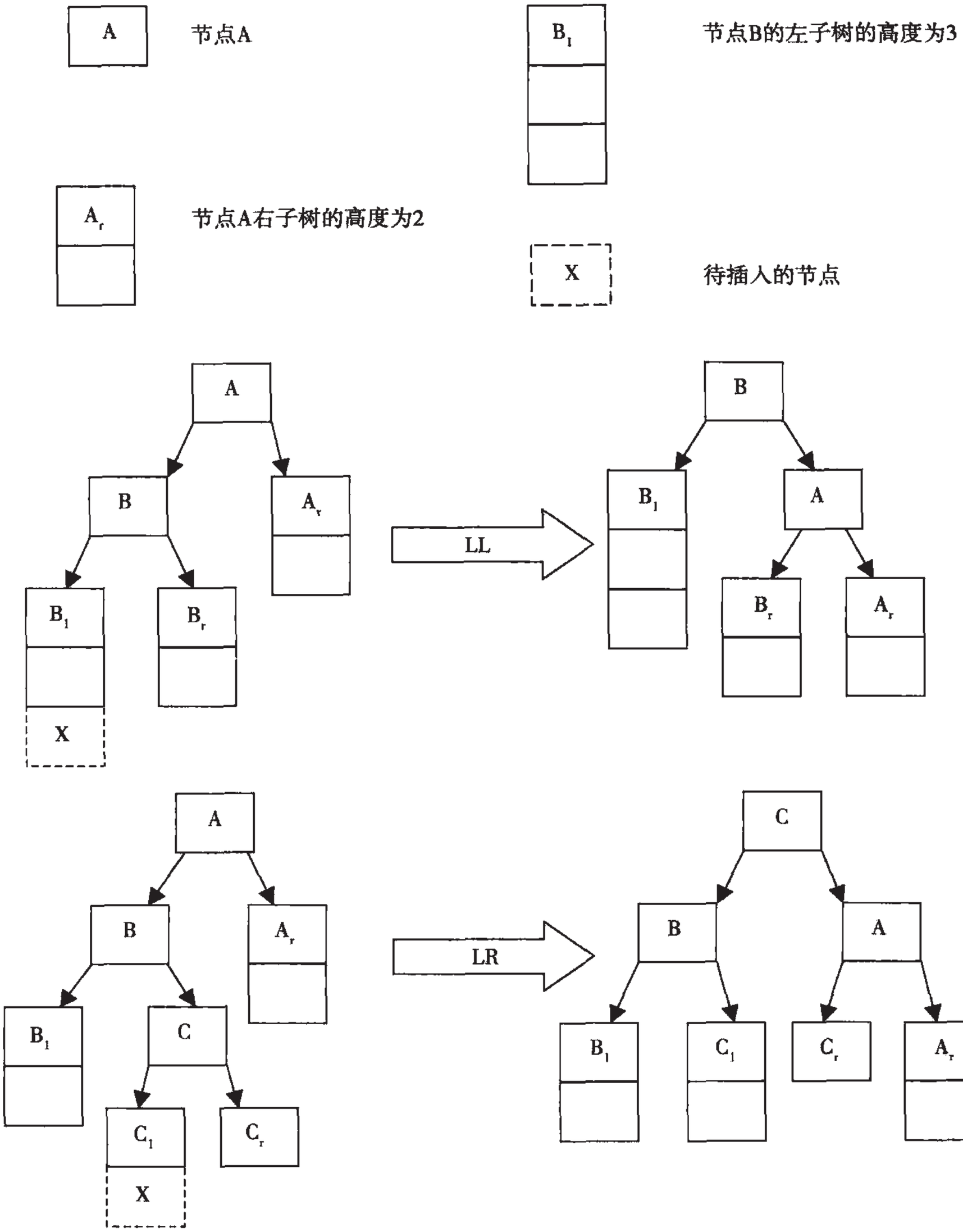


图 9-42　T 树的旋转平衡操作

T 树存在一种特殊的旋转，这是平衡二叉树所不存在的一种平衡操作。当 LR 或 RL 旋转操作完成后，如果节点 C 是叶节点，并且 A、B 节点都是半叶节点，那么通常的旋转平衡操作会将 C 节点变成一个内部节点，如图 9-43 所示。

因为数据的插入总是在内部节点的最大和最小边界范围中，所以当节点 C 只包含一个数据时，任何数据都不会插入 C 节点，除非 C 节点通过旋转再次变为叶节点或半叶节点。这

种情况会极大浪费存储空间，尤其对内存存储引擎来说。因此，在 T 树中有一种特殊的旋转操作，它会将节点 B 中的数据移动到节点 C 中，节点 C 变成满的内部节点，这个操作如图 9-44 所示。

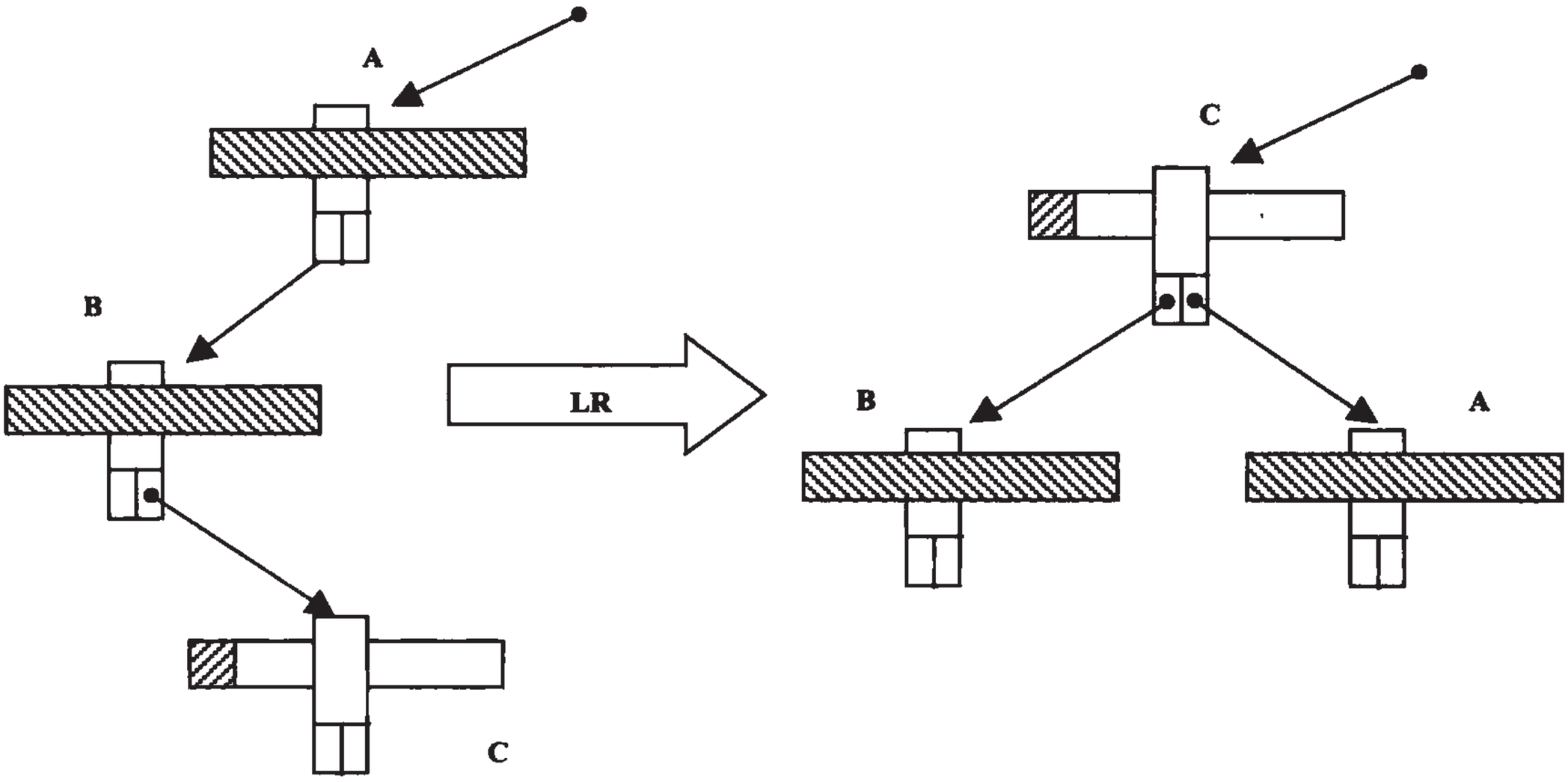


图 9-43 普通 T 树的 LR 旋转

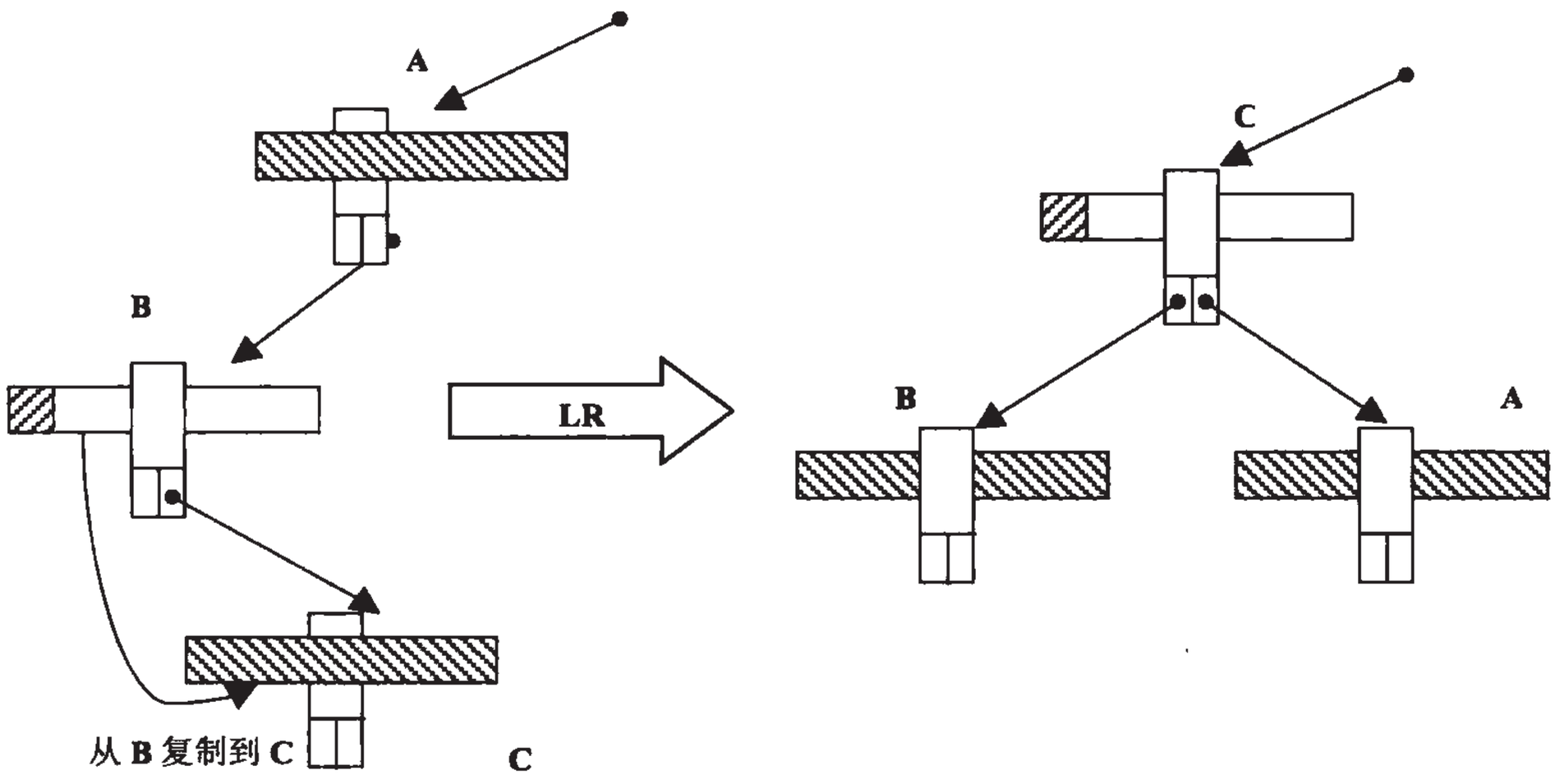


图 9-44 T 树的特殊 LR 旋转

9.10 哈希索引

当前 MySQL 数据库中，Memory 存储引擎支持哈希索引。InnoDB 存储引擎支持自适应哈希索引，用户仅能开启该特性，不能对其进行人工干预。散列算法是一种常见算法，时间复杂度为 $O(1)$，并且不只存在于索引中，每个数据库应用中都存在该数据库结构。设想这样一个问题，当前服务器的内存为 128GB，用户怎样从内存中得到某一个被缓存的页呢？内存中的查询速度很快，但是也不可能每次遍历所有内存进行查找。这时对于字典操作只需 $O(1)$ 复杂度的散列算法，就能有很好的用武之处。

9.10.1 散列表

散列表（hash table）由直接寻址表改进而来。先来看直接寻址表。当关键字的全域 U 比较小时，直接寻址是一种简单而有效的技术。假设某应用要用到一个动态集合，其中每个元素都有一个取自全域 U={0,1,...,m-1} ⊖的关键字，同时假设没有两个元素具有相同的关键字。

用一个数组（即直接寻址表）T[0..m-1] 表示动态集合，其中每个位置（称槽或桶）对应全域 U 中的一个关键字。如图 9-45 所示，槽 K 指向集合中一个关键字为 K 的元素。如果该集合中没有关键字为 K 的元素，则 T[K]=NULL。

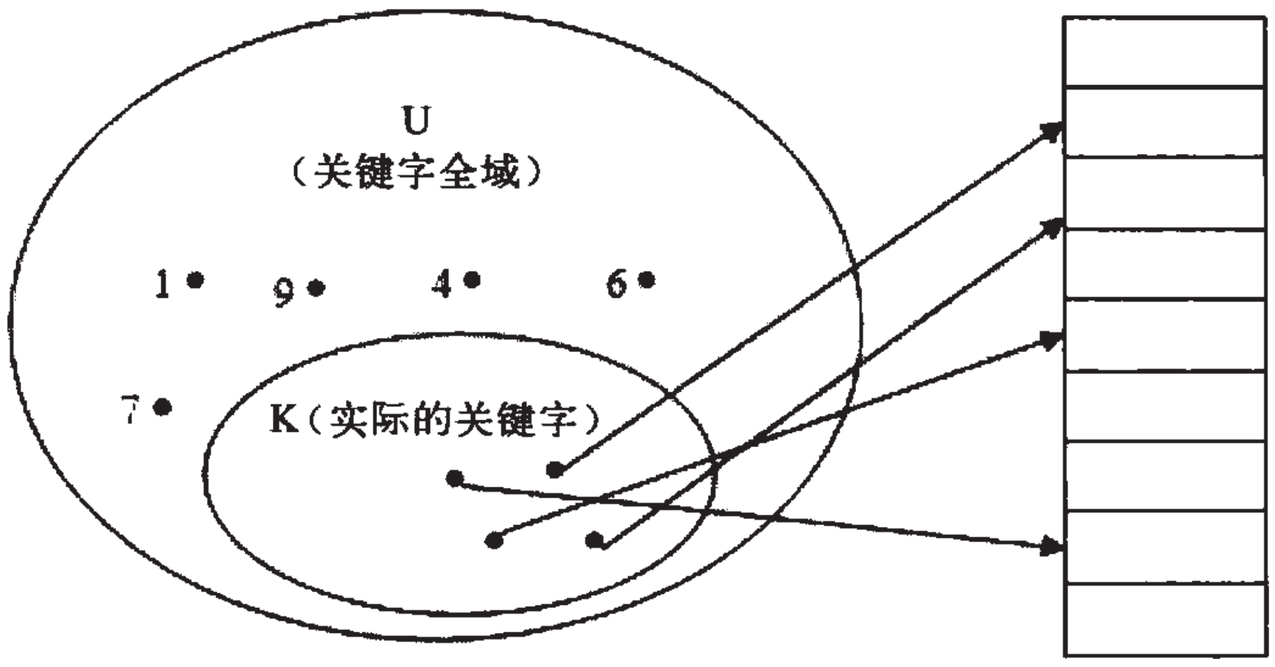


图 9-45 直接寻址表

直接寻址技术存在一个很明显的问题，如果全域 U 很大，在一台典型计算机的可用内存容量限制下，要存储大小为 U 的一张表 T 有点不实际，甚至是不可能的。如果实际要存储的关键字集合 K 相对于 U 来说可能很小，那么分配给 T 的大部分空间都要浪费掉。

因此，散列表的数据结构出现了。在散列方式下，该元素处于 h（K）中，即利用散列函数 h，根据关键字 K 计算出槽的位置。函数 h 将关键字全域 U 映射到散列表 T[0..m-1] 的槽位上，如图 9-46 所示。

⊖ 此处的 m 不是一个很大的数。

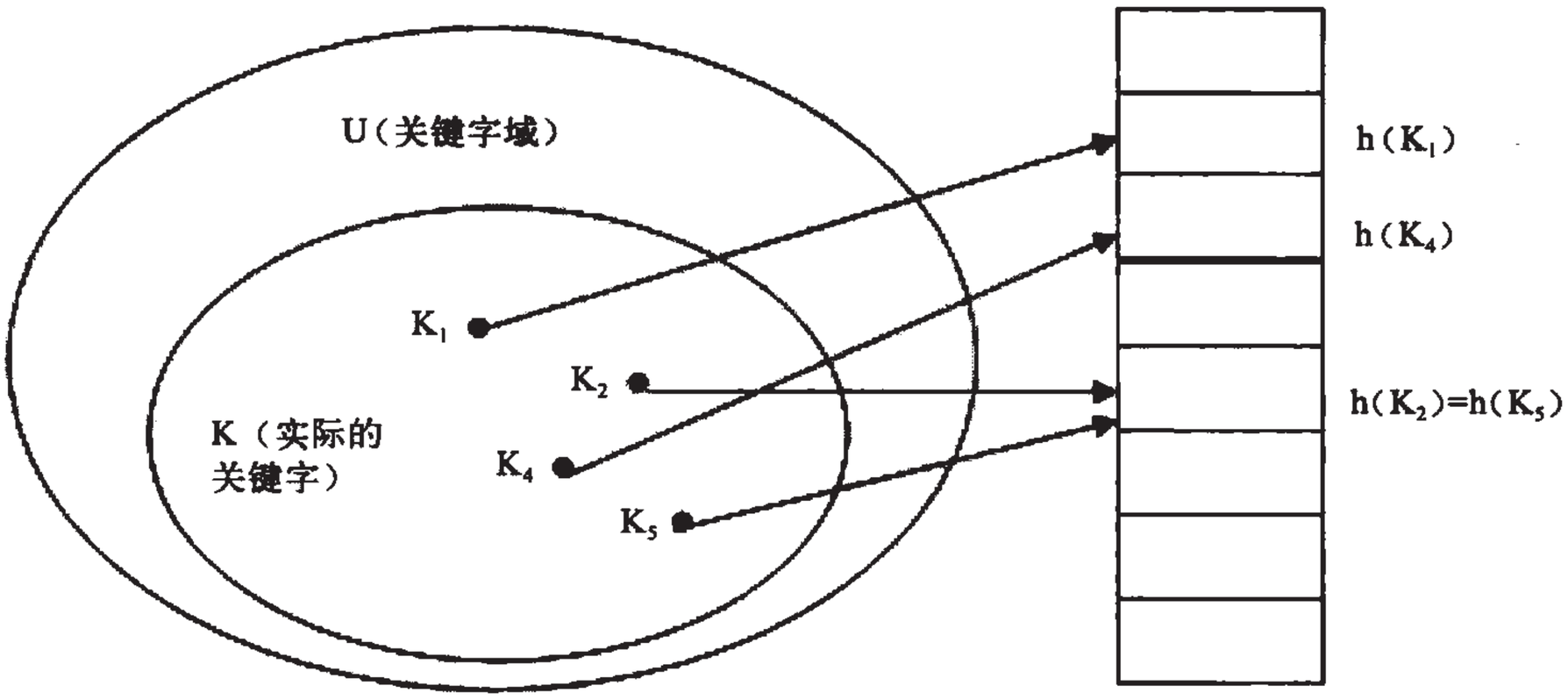


图 9-46 散列表

散列表很好地解决了直接寻址遇到的问题，但是这样做有一个小问题，如图 9-46 所示，两个关键字可能映射到同一个槽上。一般将这种情况称为发生了碰 撞（collision）。数据库中一般采用最简单的碰撞解决技术，即链接法（chaining）。

在链接法中，把散列到同一槽中的所有元素都放在一个链表中，如图 9-47 所示。每个槽中有一个指针，它指向由所有散列元素构成的链表的头，如果不存在这样的元素，则为 NULL。

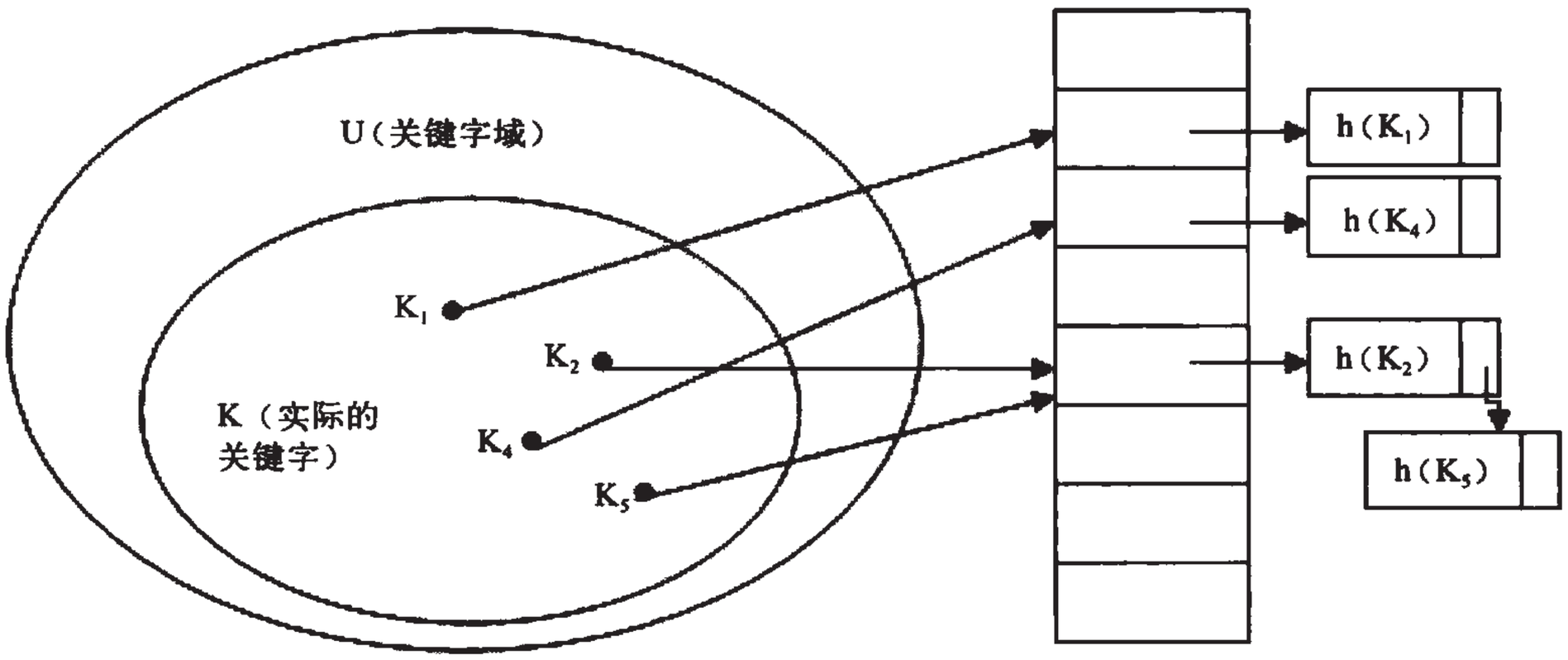


图 9-47 通过链表法解决碰撞的散列表

最后要考虑的是散列函数。散列函数 h 必须可以很好地进行散列。最好的情况是能避免碰撞的发生，即使不能避免，也应该使碰撞在最小程度下产生。一般都将关键字转换成自然数，然后通过除法散列、乘法散列或全域散列来实现。数据库中一般采用除法散列的方法。

9.10.2 InnoDB 存储引擎中的散列算法

InnoDB 存储引擎使用散列算法来对字典进行查找，其冲突机制采用链表方式，散列函数采用除法散列方式。对于缓冲池页的散列表来说，缓冲池中的 Page 页都有一个 chain 指针，它指向相同散列函数值的页。而对于除法散列，m 的取值为略大于 2 倍的缓冲池页数量的质数。例如，当前参数 innodb_buffer_pool_size 的大小为 10MB，则共有 640 个 16KB 的页，对于缓冲池页内存的散列表来说，需要分配 640*2=1280 个槽，而 1280 不是质数，因此需要取比 1280 略大的一个质数，应该是 1399，这样在启动时会分配 1399 个槽的散列表，用来散列查询所在缓冲池中的页。

那么 InnoDB 存储引擎对于页是怎么进行查找的呢？上面只是给出了一般的算法，怎么将要查找的页转换成自然数呢？其实也很简单，InnoDB 存储引擎的表空间都有一个 space 号，我们要查找的应该是某个表空间的某个连续 16KB 的页，即偏移量 offset。InnoDB 存储引擎将 space 左移 20 位，然后加上这个 space 和 offset，即关键字 K=space << 20 + space + offset，然后通过除法散列到各个槽中去。

9.10.3 自适应哈希索引

自适应哈希索引采用之前讨论的散列表的方式实现。不同的是，自适应哈希索引仅是数据库自身创建并使用的，DBA 本身并不能对其进行干预。自适应哈希索引经散列函数映射到一个散列表中，因此对于字典类型的查找非常快速，例如 SELECT * FROM TABLE WHERE index_col='xxx'，但是对于范围查找就无能为力了。通过命令 SHOW ENGINE INNODB STATUS 可以看到当前自适应哈希索引的使用状况，例如：

```
mysql>show engine innodb status\G;
*************************** 1. row ***************************
Status:
=====================================
090922 11:52:51 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 15 seconds
......
---------------------------------------
INSERT BUFFER AND ADAPTIVE HASH INDEX
---------------------------------------
Ibuf: size 2249, free list len 3346, seg size 5596,
374650 inserts, 51897 merged recs, 14300 merges
Hash table size 4980499, node heap has 1246 buffer(s)
1640.60 hash searches/s, 3709.46 non-hash searches/s
......
```

现在可以看到自适应哈希索引的使用信息，包括自适应哈希索引的大小、使用情况、每秒使用自适应哈希索引搜索的情况。需要注意的是，哈希索引只能用来搜索等值的查询，例

如 select * from table where index_col = 'xxx'，而对于其他查找类型，比如范围查找，是不能使用哈希索引的，因此，这里出现了 non-hash searches/s 的情况。由 hash searches : non-hash searches 可以大概了解使用哈希索引后的效率。

由于自适应哈希索引是由 InnoDB 存储引擎自己控制的，所以这些信息只供我们参考。不过可以通过参数 innodb_adaptive_hash_index 来禁用或启动此特性，默认为开启。

9.11 小结

本章更倾向于从内部的实现来分析索引，较之前的章节在风格上有很大的不同。这样处理是为了使读者更好地理解数据库中索引的实现，特别是 B+ 树索引。通过这种方式，用户才能进行高质量的 SQL 编程，写出高质量的应用程序。

第 10 章
分　　区

分区是一种表的设计模式。正确的分区可以极大地提升数据库的查询效率，完成更高质量的 SQL 编程。但是如果错误地使用分区，或者过于迷信分区，那么分区可能带来毁灭性的结果。本章介绍 MySQL 数据库中的分区功能，使用户了解何时使用分区及如何正确地使用分区功能。

10.1　分区概述

分区功能并不是在存储引擎层完成的，因此不只有 InnoDB 存储引擎支持分区，常见的存储引擎 MyISAM、NDB 等都支持分区。但是并不是所有的存储引擎都支持，如 CSV、FEDORATED、MERGE 等就不支持分区。在使用分区功能前，应该对选择的存储引擎对分区的支持有所了解。

MySQL 数据库在 5.1 版本时添加了对分区的支持。分区的过程是将一个表或索引分解为多个更小、更可管理的部分。就访问数据库的应用而言，从逻辑上讲，只有一个表或一个索引，但是在物理上这个表或索引可能由数十个物理分区组成。每个分区都是独立的对象，可以独自处理，也可以作为一个更大对象的一部分进行处理。

MySQL 数据库支持的分区类型为水平分区[⊖]，并不支持垂直分区[⊜]。此外，MySQL 数据库的分区是局部分区索引，一个分区中既存放数据又存放索引。全局分区是指，数据存放各个分区中，但是所有数据的索引放在一个对象中。目前，MySQL 数据库暂时不支持全局分区。

可以通过以下命令来查看当前数据库是否启用了分区功能。

```
mysql> SHOW VARIABLES LIKE '%partition%'\G;
*************************** 1. row ***************************
Variable_name: have_partitioning
        Value: YES
1 row in set (0.00 sec)
```

也可以通过命令 SHOW PLUGINS 来查看是否启用了分区。

```
mysql> SHOW PLUGINS\G;
......
*************************** 2. row ***************************
   Name: partition
 Status: ACTIVE
   Type: STORAGE ENGINE
Library: NULL
License: GPL
......
```

⊖ 水平分区，指将同一表中不同行的记录分配到不同的物理文件中。

⊜ 垂直分区，指将同一表中不同的列分配到不同的物理文件中。

```
9 rows in set (0.01 sec)
```

大多数 DBA 会有这样一个误区：只要启用了分区，数据库就会运行得更快。这种想法是存在很多问题的。以笔者的经验看来，分区可能会提高某些 SQL 语句性能，但是其主要用于高可用性，利于数据库的管理。在 OLTP 应用中，对分区的使用应该非常小心。总之，如果只是一味地使用分区，而不理解分区是如何工作的，也不清楚如何使用分区，那么分区极有可能只会对性能产生负面的影响。

当前 MySQL 数据库支持以下几种类型的分区。

- RANGE 分区：行数据基于属于一个给定连续区间的列值放入分区。MySQL 5.5 开始支持 RANGE COLUMNS 的分区。
- LIST 分区：和 RANGE 分区类型一样，只是 LIST 分区面向的是离散的值。MySQL 5.5 开始支持 LIST COLUMNS 的分区。
- HASH 分区：根据用户自定义表达式的返回值来进行分区，返回值不能为负数。
- KEY 分区：根据 MySQL 数据库提供的散列函数来进行分区。

不论创建何种类型的分区，如果表中存在主键或唯一索引时，分区列必须是唯一索引的一个组成部分，因此下面执行创建分区的 SQL 语句是会产生错误的。

```
mysql>CREATE TABLE t1 (
    -> col1 INT NOT NULL,
    -> col2 DATE NOT NULL,
    -> col3 INT NOT NULL,
    -> col4 INT NOT NULL,
    -> UNIQUE KEY (col1, col2)
    -> )
    -> PARTITION BY HASH(col3)
    -> PARTITIONS 4;
ERROR 1503 (HY000): A PRIMARY KEY must include all columns in the table's partitioning
    function
```

唯一索引可以是 NULL 值，并且只要求分区列是唯一索引的一个组成部分，不需要整个唯一索引列都是分区列，例如：

```
mysql>CREATE TABLE t1 (
    -> col1 INT  NULL,
    -> col2 DATE NULL,
    -> col3 INT NULL,
    -> col4 INT NULL,
    -> UNIQUE KEY (col1, col2, col3,col4)
    -> )
    -> PARTITION BY HASH(col3)
    -> PARTITIONS 4;
Query OK, 0 rows affected (0.53 sec)
```

当建表时没有指定主键和唯一索引时，可以指定任何一个列为分区列，因此下面两句创

建分区的SQL语句都可以正确运行。

```
CREATE TABLE t1 (
   col1 INT  NULL,
   col2 DATE NULL,
   col3 INT NULL,
   col4 INT NULL
)engine=innodb
PARTITION BY HASH(col3)
PARTITIONS 4;

CREATE TABLE t1 (
   col1 INT  NULL,
   col2 DATE NULL,
   col3 INT NULL,
   col4 INT NULL,
key (col4)
)engine=innodb
PARTITION BY HASH(col3)
PARTITIONS 4;
```

10.2 分区类型

10.2.1 RANGE分区

这里介绍的第一种分区类型是RANGE分区，也是最常用的一种分区类型。下面的CREATE TABLE语句创建了一个id列的区间分区表。当id小于10时，将数据插入到p0分区；当id大于等于10小于20时，将插入到p1分区。

```
CREATE TABLE t(
id INT
)ENGINE=INNDB
PARTITION BY RANGE (id)(
PARTITION p0 VALUES LESS THAN (10),
PARTITION p1 VALUES LESS THAN (20));
```

查看表在磁盘上的物理文件，启用分区之后，表不再由一个ibd文件组成，而是由建立分区时的各个分区ibd文件组成，比如下面的t#P#p0.ibd、t#P#p1.ibd。

```
mysql>  system ls -lh /usr/local/mysql/data/test2/t*
-rw-rw----  1 mysql mysql 8.4K  7月 31 14:11 /usr/local/mysql/data/test2/t.frm
-rw-rw----  1 mysql mysql   28  7月 31 14:11 /usr/local/mysql/data/test2/t.par
-rw-rw----  1 mysql mysql  96K  7月 31 14:12 /usr/local/mysql/data/test2/t#P#p0.ibd
-rw-rw----  1 mysql mysql  96K  7月 31 14:12 /usr/local/mysql/data/test2/t#P#p1.ibd
```

接着插入如下数据：

```
mysql> INSERT INTO t SELECT 9;
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> INSERT INTO t SELECT 10;
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> INSERT INTO t SELECT 15;
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

由于表 t 根据列 id 进行分区，因此数据根据 id 列值的范围存放在不同的物理文件中，可以通过查询 information_schema 架构下的 PARTITIONS 表来查看每个分区的具体信息，执行的语句如下：

```
mysql> SELECT * FROM information_schema.PARTITIONS
   ->WHERE table_schema=database() AND table_name='t'\G;
*************************** 1. row ***************************
                  TABLE_CATALOG: NULL
                   TABLE_SCHEMA: test2
                     TABLE_NAME: t
                 PARTITION_NAME: p0
              SUBPARTITION_NAME: NULL
   PARTITION_ORDINAL_POSITION: 1
SUBPARTITION_ORDINAL_POSITION: NULL
              PARTITION_METHOD: RANGE
           SUBPARTITION_METHOD: NULL
         PARTITION_EXPRESSION: id
      SUBPARTITION_EXPRESSION: NULL
        PARTITION_DESCRIPTION: 10
                   TABLE_ROWS: 1
               AVG_ROW_LENGTH: 16384
                  DATA_LENGTH: 16384
              MAX_DATA_LENGTH: NULL
                 INDEX_LENGTH: 0
                    DATA_FREE: 0
                  CREATE_TIME: NULL
                  UPDATE_TIME: NULL
                   CHECK_TIME: NULL
                     CHECKSUM: NULL
              PARTITION_COMMENT:
                     NODEGROUP: default
               TABLESPACE_NAME: NULL
*************************** 2. row ***************************
                  TABLE_CATALOG: NULL
                   TABLE_SCHEMA: test2
                     TABLE_NAME: t
                 PARTITION_NAME: p1
```

```
             SUBPARTITION_NAME: NULL
   PARTITION_ORDINAL_POSITION: 2
SUBPARTITION_ORDINAL_POSITION: NULL
               PARTITION_METHOD: RANGE
            SUBPARTITION_METHOD: NULL
          PARTITION_EXPRESSION: id
       SUBPARTITION_EXPRESSION: NULL
         PARTITION_DESCRIPTION: 20
                  TABLE_ROWS: 2
              AVG_ROW_LENGTH: 8192
                 DATA_LENGTH: 16384
             MAX_DATA_LENGTH: NULL
                INDEX_LENGTH: 0
                   DATA_FREE: 0
                 CREATE_TIME: NULL
                 UPDATE_TIME: NULL
                  CHECK_TIME: NULL
                    CHECKSUM: NULL
            PARTITION_COMMENT:
                   NODEGROUP: default
              TABLESPACE_NAME: NULL
2 rows in set (0.00 sec)
```

TABLE_ROWS 列反映了每个分区中记录的数量。由于之前向表中插入了 9、10、15 三条记录，因此可以看到，当前分区 p0 中有一条记录，分区 p1 中有两条记录。PARTITION_METHOD 表示分区的类型，这里显示的是 RANGE。

对于表 t，我们定义了分区，因此插入的值应该严格遵守分区的定义，当插入一个不在分区中定义的值时，MySQL 数据库会抛出一个异常。下面我们向表 t 中插入 30 这个值。

```
mysql> INSERT INTO t SELECT 30;
ERROR 1526 (HY000): Table has no partition for value 30
```

对于上述问题，我们可以对分区添加一个 MAXVALUE 值的分区。MAXVALUE 可以理解为正无穷，因此所有大于等于 20 并且小于 MAXVALUE 的值都可以放入 p2 分区，如下所示：

```
mysql> ALTER TABLE t
    -> ADD PARTITION(
    -> partition p2 values less than maxvalue );
Query OK, 0 rows affected (0.45 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> INSERT INTO t SELECT 30;
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

RANGE 分区主要用于日期列的分区，例如，对于销售类的表，可以根据年份来分区存放销售记录，如下是对 sales 表进行分区。

```
mysql> CREATE TABLE sales(
    -> money INT UNSIGNED NOT NULL,
    -> date DATETIME
    ->)ENGINE=INNODB
    -> PARTITION by RANGE  (YEAR(date)) (
    -> PARTITION p2008 VALUE LESS THEN (2009),
    -> PARTITION p2009 VALUE LESS THEN (2010),
    -> PARTITION p2010 VALUE LESS THEN (2011)
    -> );
Query OK, 0 rows affected (0.34 sec)

mysql> INSERT INTO sales SELECT 100, '2008-01-01';
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> INSERT INTO sales SELECT 100, '2008-02-01';
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> INSERT INTO sales SELECT 200, '2008-01-02';
Query OK, 1 row affected (0.04 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> INSERT INTO sales SELECT 100, '2009-03-01';
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> INSERT INTO sales SELECT 200, '2010-03-01';
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

这样创建的好处是便于对 sales 表的管理，如果要删除 2008 年的数据，我们不需要执行 DELETE FROM sales WHERE date>='2008-01-01' and date <'2009-01-01', 只需删除 2008 年数据所在的分区即可，执行的语句如下：

```
mysql>alter table sales drop partition p2008;
Query OK, 0 rows affected (0.18 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

这样创建的另一个好处是可以加快某些查询操作，例如，我们只需要查询 2008 年整年的销售额，操作如下：

```
mysql> EXPLAIN PARTITIONS
   -> SELECT * FROM sales
   ->WHERE date>='2008-01-01' AND date<='2008-12-31'\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales
```

```
   partitions: p2008
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 5
        Extra: Using where
1 row in set (0.00 sec)
```

通过 EXPLAIN PARTITION 命令我们可以发现，在上述语句中，SQL 优化器只需要搜索 p2008 这个分区，而不会搜索所有的分区——称为 Partition Pruning（分区修剪），故查询的速度得到了大幅度提升。需要注意的是，如果执行下列语句，结果是一样的，但是优化器的选择可能又会不同。

```
mysql> EXPLAIN PARTITIOENS
    -> SELECT * FROM sales
    ->WHERE date>='2008-01-01' AND date<'2009-01-01'\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales
   partitions: p2008,p2009
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 5
        Extra: Using where
1 row in set (0.00 sec)
```

这次的条件为 date<'2009-01-01' 而不是 date<='2008-12-31'，优化器会选择搜索 p2008 和 p2009 两个分区，这是我们不希望看到的，因此对于启用分区，应该根据分区的特性来编写最优的 SQL 语句。

对于 sales 这张分区表，笔者曾看到过另一种分区函数，设计者的原意是想按照每年每月来进行分区，例如：

```
mysql> CREATE TABLE sales(
    ->  money INT UNSIGNED NOT NULL,
    ->date DATETIME
    ->  )ENGINE=INNODB
    ->  PARTITION by RANGE  (YEAR(date)*100+MONTH(date)) (
    ->  PARTITION p201001 VALUES LESS THEN (201002),
    ->  PARTITION p201002 VALUES LESS THEN (201003),
    ->  PARTITION p201003 VALUES LESS THEN (201004)
    ->  );
Query OK, 0 rows affected (0.37 sec)
```

但是在执行 SQL 语句时开发人员发现，优化器不会根据分区进行选择，即使他们编写的 SQL 语句已经符合了分区的要求，例如：

```
mysql> EXPLAIN PARTITIONS
    -> SELECT * FROM sales
    ->WHERE date>='2010-01-01' AND date<='2010-01-31'\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales
   partitions: p201001,p201002,p201003
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 4
        Extra: Using where
1 row in set (0.00 sec)
```

可以看到优化对分区 p201001、p201002 和 p201003 都进行了搜索。产生这个问题的主要原因是，对 RANGE 分区的查询，优化器只能对 YEAR()、TO_DAYS()、TO_SECONDS() 和 UNIX_TIMESTAMP() 这类函数进行优化选择，因此对于上述要求，需要将分区函数改为 TO_DAYS，例如：

```
mysql> CREATE TABLE sales(
    ->money INT UNSIGNED NOT NULL,
    ->date DATETO,E
    ->)ENGINE=INNODB
    ->  PARTITION by range  (TO_DAYS(date)) (
    ->  PARTITION p201001
    ->VALUES LESS THEN(TO DAYS('2010-02-01')),
    ->  PARTITION p201002
    ->VALUES LESS THEN (TO_DAYS('2010-03-01')),
    ->  PARTITION p201003
    ->VALUES LESS THEN (TO_DAYS('2010-04-01'))
    ->   );
Query OK, 0 rows affected (0.36 sec)
```

这时再进行相同类型的查询，优化器就可以对特定的分区进行查询，执行的语句如下：

```
mysql> EXPLAIN PATITIONS
    -> SELECT * FROM sales
    ->WHERE date>='2010-01-01' AND date<='2010-01-31'\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales
```

```
   partitions: p201001
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 4
        Extra: Using where
1 row in set (0.00 sec)
```

10.2.2 LIST 分区

LIST 分区和 RANGE 分区非常相似，只是分区列的值是离散的，而非连续的，例如：

```
mysql> CREATE TABLE t (
    -> a INT,
    -> b INT)ENGINE=INNODB
    -> PARTITION BY LIST(b)(
    -> PARTITION p0 VALUES IN (1,3,5,7,9),
    -> PARTITION p1 VALUES IN (0,2,4,6,8)
    -> );
Query OK, 0 rows affected (0.26 sec)
```

不同于 RANGE 分区中定义的 VALUES LESS THAN 语句，LIST 分区使用 VALUES IN。因为每个分区的值是离散的，因此只能定义值。例如向表 t 中插入一些数据需要执行下列语句：

```
mysql> INSERT INTO t SELECT 1,1;
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> INSERT INTO t SELECT 1,2;
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> INSERT INTO t SELECT 1,3;
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> INSERT INTO t SELECT 1,4;
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT table_name,partition_name,table_rows
   -> FROM information_schema.PARTITIONS
   -> WHERE table_name='t' AND table_schema=DATABASE()\G;
*************************** 1. row ***************************
    table_name: t
partition_name: p0
```

```
    table_rows: 2
*************************** 2. row ***************************
    table_name: t
partition_name: p1
    table_rows: 2
2 rows in set (0.00 sec)
```

如果插入的值不在分区的定义中，那么 MySQL 数据库同样会抛出异常：

```
mysql> INSERT INTO t SELECT 1,10;
ERROR 1526 (HY000): Table has no partition for value 10
```

另外，在执行 INSERT 操作插入多个行数据的过程中如果遇到分区未定义的值，MyISAM 和 InnoDB 存储引擎的处理会完全不同。MyISAM 引擎会将之前的行数据都插入，但之后的数据不会被插入。而 InnoDB 存储引擎将其视为一个事务，没有任何数据被插入。先对 MyISAM 存储引擎进行演示，例如：

```
mysql> CRATE TABLE t (
    -> a INT,
    -> b INT)ENGINE=MyISAM
    -> PARTITION BY LIST(b)(
    -> PARTITION p0 VALUES IN (1,3,5,7,9),
    -> PARTITION p1 VALUES IN (0,2,4,6,8)
    -> );
Query OK, 0 rows affected (0.05 sec)

mysql> INSERT INTO t VALUES (1,2),(2,4),(6,10),(5,3);
ERROR 1526 (HY000): Table has no partition for value 10

mysql> SELECT * FROM t;
+------+------+
| a    | b    |
+------+------+
|    1 |    2 |
|    2 |    4 |
+------+------+
2 rows in set (0.00 sec)
```

可以看到，对于插入（6，10）、（5，3），没有成功执行，但是之前的（1，2）、（2，4）记录都已经成功插入了。而对于同一张表，存储引擎换成 InnoDB，结果完全不同，例如：

```
mysql> TRUNCATE TABLE t;
Query OK, 2 rows affected (0.00 sec)

mysql> ALTER TABLE t ENGINE=InnoDB;
Query OK, 0 rows affected (0.25 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> INSERT INTO t VALUES (1,2),(2,4),(6,10),(5,3);
```

```
ERROR 1526 (HY000): Table has no partition for value 10

mysql> SELECT * FROM t;
Empty set (0.00 sec)
```

可以看到，同样在插入（6，10）记录时报错，但是没有任何一条记录被插入到表t中。因此在使用分区时，也需要考虑不同存储引擎支持的事务特性。

10.2.3 HASH分区

HASH分区的目的是将数据均匀地分布到预先定义的各个分区中，保证各分区的数据数量大致是一样的。在RANGE和LIST分区中，必须明确指定一个给定的列值或列值集合应该保存在哪个分区中；而在HASH分区中，MySQL自动完成这些工作，用户所要做的只是基于将要被散列的列值指定一个列值或表达式，以及指定被分区的表将要被分割成的分区数量。

要使用HASH分区来分割一个表，要在CREATE TABLE语句上添加一个PARTITION BY HASH (expr)子句，其中“expr”是返回一个整数的表达式。expr可以仅仅是字段类型为MySQL整型的列名。此外，用户很可能需要在后面再添加一个PARTITIONS num子句，其中num是一个非负的整数，它表示表将要被分割成分区的数量。如果没有包含一个PARTITIONS子句，那么分区的数量将默认为1。

下面的例子创建了一个HASH分区的表t，按日期列b进行分区。

```
CREATE TABLE t_hash (
   a INT,
   b DATETIME
)ENGINE=InnoDB
PARTITION BY HASH (YEAR(b))
PARTITIONS 4;
```

如果插入2010-04-01这个记录到表t_hash的列b中，那么保存该条记录的分区确定如下：

```
MOD(YEAR( '2010-04-01' ), 4)
=MOD(2010,4)
=2
```

因此该记录会被放入分区p2中。我们可以按如下方法来验证：

```
mysql> INSERT INTO t_hash SELECT 1,'2010-04-01';
Query OK, 1 row affected (0.04 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT table_name,partition_name,table_rows
   -> FROM information_schema.PARTITIONS
   -> WHERE table_schema=DATABASE() AND table_name='t_hash'\G;
*************************** 1. row ***************************
    table_name: t_hash
```

```
partition_name: p0
    table_rows: 0
*************************** 2. row ***************************
    table_name: t_hash
partition_name: p1
    table_rows: 0
*************************** 3. row ***************************
    table_name: t_hash
partition_name: p2
    table_rows: 1
*************************** 4. row ***************************
    table_name: t_hash
partition_name: p3
    table_rows: 0
4 rows in set (0.00 sec)
```

可以看到 p2 分区中有 1 条记录。当然这个例子并不能把数据均匀地分布到各个分区中去，因为分区是按照 YEAR 函数进行的，而这个值本身是离散的。如果对连续的值进行 HASH 分区，如自增长的主键，则可以较好地将数据进行平均分布。

MySQL 数据库还支持一种称为 LINEAR HASH 的分区，它使用一个更加复杂的算法来确定新行插入到已经分区的表中的位置。它的语法和 HASH 分区的语法相似，只是将关键字 HASH 改为 LINEAR HASH。下面创建一个 LINEAR HASH 的分区表 t_linear_hash，这个表和之前的表 t_hash 相似，只是分区类型不同。

```
CREATE TABLE t_linear_hash(
   a INT,
   b DATETIME
)ENGINE=InnoDB
PARTITION BY LINEAR HASH (YEAR(b))
PARTITIONS 4;
```

同样插入 2010-04-01 这条记录，这次 MySQL 数据库根据以下方法来对分区进行判断。

- 取大于分区数量 4 的下一个 2 的幂值 V，V=POWER(2, CEILING(LOG(2, *num*))) = 4。
- 所在分区 N=YEAR('2010-04-01') & (V−1) = 2。

虽然还是在分区 2，但是计算的方法和之前进行 HASH 分区完全不同，接着进行插入实际数据的验证。

```
mysql> INSERT INTO t_linear_hash SELECT 1,'2010-04-01';
Query OK, 1 row affected (0.02 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT table_name,partition_name,table_rows
    -> FROM information_schema.PARTITIONS
    -> WHERE table_schema=DATABASE()
    -> AND table_name='t_linear_hash'\G;
*************************** 1. row ***************************
```

```
    table_name: t_linear_hash
partition_name: p0
    table_rows: 0
*************************** 2. row ***************************
    table_name: t_linear_hash
partition_name: p1
    table_rows: 0
*************************** 3. row ***************************
    table_name: t_linear_hash
partition_name: p2
    table_rows: 1
*************************** 4. row ***************************
    table_name: t_linear_hash
partition_name: p3
    table_rows: 0
4 rows in set (0.01 sec)
```

LINEAR HASH 分区的优点在于增加、删除、合并和拆分分区将变得更加快捷，这有利于处理含有大量数据的表。LINEAR HASH 分区的缺点在于，与使用 HASH 分区得到的数据分布相比，各个分区间数据的分布可能不大均衡。

10.2.4 KEY 分区

KEY 分区和 HASH 分区相似，不同之处在于 HASH 分区通过用户定义的函数进行分区，KEY 分区使用 MySQL 数据库提供的函数进行分区。NDB Cluster 引擎使用 MD5 函数来分区，对于其他存储引擎，MySQL 数据库使用其内部的散列函数来分区，这些函数基于与 PASSWORD() 一样的运算法则。例如：

```
mysql> CREATE TABLE t_key (
    -> a INT,
    -> b DATETIME)ENGINE=InnoDB
    -> PARTITION BY KEY (b)
    -> PARTITIONS 4;
Query OK, 0 rows affected (0.43 sec)
```

关键字 LINEAR 在 KEY 分区中使用和在 HASH 分区中使用具有同样的作用，分区的编号是通过 2 的幂值（powers-of-two）得到的，而不是通过模数算法。

10.2.5 COLUMNS 分区

在前面介绍的 RANGE、LIST、HASH 和 KEY 这四种分区中，分区的条件必须是整型（integer），如果不是整型，那么需要通过函数将其转化为整型，如 YEAR()、TO_DAYS()、MONTH() 等函数。MySQL 5.5 版本开始支持 COLUMNS 分区，可视为对 RANGE 分区和 LIST 分区的一种进化。COLUMNS 分区可以直接使用非整型的数据进行分区，分区根据类型直接比较而得到，不需要转化为整型。此外，RANGE COLUMNS 分区可以对多个列的值

进行分区。

COLUMNS 分区支持以下数据类型：

- 所有的整型类型，如 INT、SMALLINT、TINYINT 和 BIGINT。对 FLOAT 和 DECIMAL 则不予支持。
- 日期类型，如 DATE 和 DATETIME。对其余的日期类型不予支持。
- 字符串类型，如 CHAR、VARCHAR、BINARY 和 VARBINARY。对 BLOB 和 TEXT 类型不予支持。

对于日期类型的分区，我们不再需要 YEAR（）和 TO_DAYS（）函数，可以直接使用 COLUMNS，例如：

```
CREATE TABLE t_columns_range(
   a INT,
   b DATETIME
)ENGINE=INNODB
PARTITION BY RANGE COLUMNS (B)(
PARTITION p0 VALUES LESS THAN ('2009-01-01'),
PARTITION p1 VALUES LESS THAN ('2010-01-01')
);
```

同样地，可以直接使用字符串的分区，例如：

```
CREATE TABLE customers_1 (
first_name VARCHAR(25),
last_name VARCHAR(25),
street_1 VARCHAR(30),
street_2 VARCHAR(30),
city VARCHAR(15),
renewal DATE
)
PARTITION BY LIST COLUMNS(city) (
PARTITION pRegion_1
   VALUES IN('Oskarshamn', 'Högsby', 'Mönsterås'),
PARTITION pRegion_2
   VALUES IN('Vimmerby', 'Hultsfred', 'Västervik'),
PARTITION pRegion_3
   VALUES IN('Nässjö', 'Eksjö', 'Vetlanda'),
PARTITION pRegion_4
   VALUES IN('Uppvidinge', 'Alvesta', 'Växjo')
);
```

对于 RANGE COLUMNS 分区，可以使用多个列进行分区，例如：

```
CREATE TABLE rcx (
   a INT,
   b INT,
   c CHAR(3),
   d INT
```

```
)Engine=InnoDB
PARTITION BY RANGE COLUMNS(a,d,c) (
PARTITION p0 VALUES LESS THAN (5,10,'ggg'),
PARTITION p1 VALUES LESS THAN (10,20,'mmmm'),
PARTITION p2 VALUES LESS THAN (15,30,'sss'),
PARTITION p3 VALUES LESS THAN (MAXVALUE,MAXVALUE,MAXVALUE)
);
```

MySQL 5.5 开始支持 COLUMNS 分区，对于之前的 RANGE 和 LIST 分区，用户可以用 RANGE COLUMNS 和 LIST COLUMNS 分区来很好地代替。

10.3 子分区

子分区（subpartitioning）是在分区的基础上再进行分区，有时也称这种分区为复合分区(composite partitioning)。MySQL 数据库允许在 RANGE 和 LIST 的分区上再进行 HASH 或 KEY 的子分区，例如：

```
mysql> CREATE TABLE ts (a INT, b DATE)engine=innodb
    -> PARTITION BY RANGE( YEAR(b) )
    -> SUBPARTITION BY HASH( TO_DAYS(b) )
    -> SUBPARTITIONS 2 (
    -> PARTITION p0 VALUES LESS THAN (1990),
    -> PARTITION p1 VALUES LESS THAN (2000),
    -> PARTITION p2 VALUES LESS THAN MAXVALUE
    -> );
Query OK, 0 rows affected (0.01 sec)

mysql> system ls -lh /usr/local/mysql/data/test2/ts*
-rw-rw---- 1 mysql mysql 8.4K Aug  1 15:50 /usr/local/mysql/data/test2/ts.frm
-rw-rw---- 1 mysql mysql   96 Aug  1 15:50 /usr/local/mysql/data/test2/ts.par
-rw-rw---- 1 mysql mysql  96K Aug  1 15:50 /usr/local/mysql/data/test2/ts#P#p0#SP#p0sp0.ibd
-rw-rw---- 1 mysql mysql  96K Aug  1 15:50 /usr/local/mysql/data/test2/ts#P#p0#SP#p0sp1.ibd
-rw-rw---- 1 mysql mysql  96K Aug  1 15:50 /usr/local/mysql/data/test2/ts#P#p1#SP#p1sp0.ibd
-rw-rw---- 1 mysql mysql  96K Aug  1 15:50 /usr/local/mysql/data/test2/ts#P#p1#SP#p1sp1.ibd
-rw-rw---- 1 mysql mysql  96K Aug  1 15:50 /usr/local/mysql/data/test2/ts#P#p2#SP#p2sp0.ibd
-rw-rw---- 1 mysql mysql  96K Aug  1 15:50 /usr/local/mysql/data/test2/ts#P#p2#SP#p2sp1.ibd
```

表 ts 先根据 b 列进行了 RANGE 分区，然后又进行了一次 HASH 分区，所以分区的数量应该为（3*2=6）个，这通过查看物理磁盘上的文件也可以得到证实。我们也可以通过使用 SUBPARTITION 语法来显式地指出各个子分区的名字，对上述的 ts 表同样可以执行以下语句：

```
mysql> CREATE TABLE ts (a INT, b DATE)
    -> PARTITION BY RANGE( YEAR(b))
    -> SUBPARTITION BY HASH( TO_DAYS(b) ) (
```

```
-> PARTITION p0 VALUES LESS THAN (1990) (
-> SUBPARTITION s0,
-> SUBPARTITION s1
-> ),
-> PARTITION p1 VALUES LESS THAN (2000) (
-> SUBPARTITION s2,
-> SUBPARTITION s3
-> ),
-> PARTITION p2 VALUES LESS THAN MAXVALUE (
-> SUBPARTITION s4,
-> SUBPARTITION s5
-> )
-> );
Query OK, 0 rows affected (0.00 sec)
```

子分区的建立需要注意以下几个问题：

- 每个子分区的数量必须相同。
- 如果在一个分区表的任何分区上使用 SUBPARTITION 来明确定义任何子分区，那么就必须定义所有的子分区，因此下面的创建语句是错误的。

```
mysql> CREATE TABLE ts (a INT, b DATE)
    -> PARTITION BY RANGE( YEAR(b) )
    -> SUBPARTITION BY HASH( TO_DAYS(b) ) (
    -> PARTITION p0 VALUES LESS THAN (1990) (
    -> SUBPARTITION s0,
    -> SUBPARTITION s1
    -> ),
    -> PARTITION p1 VALUES LESS THAN (2000),
    -> PARTITION p2 VALUES LESS THAN MAXVALUE (
    -> SUBPARTITION s2,
    -> SUBPARTITION s3
    -> )
    -> );
ERROR 1064 (42000): Wrong number of subpartitions defined, mismatch with previous
    setting near '
PARTITION p2 VALUES LESS THAN MAXVALUE (
SUBPARTITION s2,
SUBPARTITION s3
)
)' at line 8
```

- 每个 SUBPARTITION 子句必须包括子分区的一个名字。
- 子分区的名字必须是唯一的，因此下面的创建语句是错误的。

```
mysql> CREATE TABLE ts (a INT, b DATE)
    -> PARTITION BY RANGE( YEAR(b) )
    -> SUBPARTITION BY HASH( TO_DAYS(b) ) (
    -> PARTITION p0 VALUES LESS THAN (1990) (
    -> SUBPARTITION s0,
```

```
    -> SUBPARTITION s1
    -> ),
    -> PARTITION p1 VALUES LESS THAN (2000) (
    -> SUBPARTITION s0,
    -> SUBPARTITION s1
    -> ),
    -> PARTITION p2 VALUES LESS THAN MAXVALUE (
    -> SUBPARTITION s0,
    -> SUBPARTITION s1
    -> )
    -> );
ERROR 1517 (HY000): Duplicate partition name s0
```

子分区可以用于特别大的表，在多个磁盘间分别分配数据和索引。假设有6个磁盘，分别为/disk0、/disk1、/disk2等。现在思考下面的例子。

```
mysql> CREATE TABLE ts (a INT, b DATE)ENGINE=MYISAM
    -> PARTITION BY RANGE( YEAR(b) )
    -> SUBPARTITION BY HASH( TO_DAYS(b) ) (
    -> PARTITION p0 VALUES LESS THAN (2000) (
    -> SUBPARTITION s0
    -> DATA DIRECTORY = '/disk0/data'
    -> INDEX DIRECTORY = '/disk0/idx',
    -> SUBPARTITION s1
    -> DATA DIRECTORY = '/disk1/data'
    -> INDEX DIRECTORY = '/disk1/idx'
    -> ),
    -> PARTITION p1 VALUES LESS THAN (2010) (
    -> SUBPARTITION s2
    -> DATA DIRECTORY = '/disk2/data'
    -> INDEX DIRECTORY = '/disk2/idx',
    -> SUBPARTITION s3
    -> DATA DIRECTORY = '/disk3/data'
    -> INDEX DIRECTORY = '/disk3/idx'
    -> ),
    -> PARTITION p2 VALUES LESS THAN MAXVALUE (
    -> SUBPARTITION s4
    -> DATA DIRECTORY = '/disk4/data'
    -> INDEX DIRECTORY = '/disk4/idx',
    -> SUBPARTITION s5
    -> DATA DIRECTORY = '/disk5/data'
    -> INDEX DIRECTORY = '/disk5/idx'
    -> )
    -> );
Query OK, 0 rows affected (0.02 sec)
```

由于InnoDB存储引擎使用表空间自动地进行数据和索引的管理，会忽略DATA DIRECTORY和INDEX DIRECTORY语法，因此上述分区表的数据和索引文件分开放置对InnoDB存储引擎表是无效的，例如：

```
mysql> CREATE TABLE ts (a INT, b DATE)engine=innodb
    -> PARTITION BY RANGE( YEAR(b) )
    -> SUBPARTITION BY HASH( TO_DAYS(b) ) (
    -> PARTITION p0 VALUES LESS THAN (2000) (
    -> SUBPARTITION s0
    -> DATA DIRECTORY = '/disk0/data'
    -> INDEX DIRECTORY = '/disk0/idx',
    -> SUBPARTITION s1
    -> DATA DIRECTORY = '/disk1/data'
    -> INDEX DIRECTORY = '/disk1/idx'
    -> ),
    -> PARTITION p1 VALUES LESS THAN (2010) (
    -> SUBPARTITION s2
    -> DATA DIRECTORY = '/disk2/data'
    -> INDEX DIRECTORY = '/disk2/idx',
    -> SUBPARTITION s3
    -> DATA DIRECTORY = '/disk3/data'
    -> INDEX DIRECTORY = '/disk3/idx'
    -> ),
    -> PARTITION p2 VALUES LESS THAN MAXVALUE (
    -> SUBPARTITION s4
    -> DATA DIRECTORY = '/disk4/data'
    -> INDEX DIRECTORY = '/disk4/idx',
    -> SUBPARTITION s5
    -> DATA DIRECTORY = '/disk5/data'
    -> INDEX DIRECTORY = '/disk5/idx'
    -> )
    -> );
Query OK, 0 rows affected (0.02 sec)

mysql> system ls -lh /usr/local/mysql/data/test2/ts*
-rw-rw---- 1 mysql mysql 8.4K Aug  1 16:24 /usr/local/mysql/data/test2/ts.frm
-rw-rw---- 1 mysql mysql   80 Aug  1 16:24 /usr/local/mysql/data/test2/ts.par
-rw-rw---- 1 mysql mysql  96K Aug  1 16:25 /usr/local/mysql/data/test2/ts#P#p0#SP#s0.ibd
-rw-rw---- 1 mysql mysql  96K Aug  1 16:25 /usr/local/mysql/data/test2/ts#P#p0#SP#s1.ibd
-rw-rw---- 1 mysql mysql  96K Aug  1 16:25 /usr/local/mysql/data/test2/ts#P#p1#SP#s2.ibd
-rw-rw---- 1 mysql mysql  96K Aug  1 16:25 /usr/local/mysql/data/test2/ts#P#p1#SP#s3.ibd
-rw-rw---- 1 mysql mysql  96K Aug  1 16:25 /usr/local/mysql/data/test2/ts#P#p2#SP#s4.ibd
-rw-rw---- 1 mysql mysql  96K Aug  1 16:25 /usr/local/mysql/data/test2/ts#P#p2#SP#s5.ibd
```

10.4 分区中的 NULL 值

MySQL 数据库允许对 NULL 值进行分区，但是处理方法可能与其他数据库完全不同。MySQL 数据库的分区总是把 NULL 值视为小于任何的一个非 NULL 值，这和 MySQL 数据库中处理 NULL 值的 ORDER BY 操作是一样的。因此对于不同的分区类型，MySQL 数据库对 NULL 值的处理也各不相同。

对于 RANGE 分区，如果向分区列插入了 NULL 值，那么 MySQL 数据库会将该值放入最左边的分区。例如：

```
mysql> CREATE TABLE t_range(
    -> a INT,
    -> b INT)ENGINE=InnoDB
    -> PARTITION BY RANGE(b)(
    -> PARTITION p0 VALUES LESS THAN (10),
    -> PARTITION p1 VALUES LESS THAN (20),
    -> PARTITION p2 VALUES LESS THAN MAXVALUE
    -> );
Query OK, 0 rows affected (0.01 sec)
```

接着向表中插入（1，1）、（1，NULL）两条数据，并观察每个分区中记录的数量：

```
mysql> INSERT INTO t_range SELECT 1,1;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> INSERT INTO t_range SELECT 1,NULL;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM t_range\G;
*************************** 1. row ***************************
a: 1
b: 1
*************************** 2. row ***************************
a: 1
b: NULL
2 rows in set (0.00 sec)

mysql> SELECT table_name,partition_name,table_rows
   -> FROM information_schema.PARTITIONS
   -> WHERE table_schema=DATABASE() AND table_name='t_range'\G;
*************************** 1. row ***************************
    table_name: t_range
partition_name: p0
    table_rows: 2
*************************** 2. row ***************************
    table_name: t_range
partition_name: p1
    table_rows: 0
*************************** 3. row ***************************
    table_name: t_range
partition_name: p2
    table_rows: 0
3 rows in set (0.00 sec)
```

可以看到这两条数据都放入了 p0 分区，也就是在 RANGE 分区下，NULL 值会放入最

左边的分区。另外需要注意的是，如果删除 p0 这个分区，那么删除的是小于 10 的记录，包括 NULL 值的记录，这点非常重要，例如：

```
mysql> ALTER TABLE t_range DROP PARTITION p0;
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM t_range;
Empty set (0.00 sec)
```

要在 LIST 分区下使用 NULL 值，必须显式地指出向哪个分区中放入 NULL 值，否则会报错，例如：

```
mysql> CREATE TABLE t_list(
    -> a INT,
    -> b INT)ENGINE=INNODB
    -> PARTITION BY LIST(b)(
    -> PARTITION p0 VALUES IN (1,3,5,7,9),
    -> PARTITION p1 VALUES IN (0,2,4,6,8)
    );
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t_list SELECT 1,NULL;
ERROR 1526 (HY000): Table has no partition for value NULL
```

若 p0 分区允许 NULL 值，则插入不会报错，例如：

```
mysql> CREATE TABLE t_list(
    -> a INT,
    -> b INT)ENGINE=INNODB
    -> PARTITION BY LIST(b)(
    -> PARTITION p0 VALUES IN (1,3,5,7,9,NULL),
    -> PARTITION p1 VALUES IN (0,2,4,6,8)
    );
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t_list SELECT 1,NULL;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT table_name,partition_name,table_rows
   -> FROM information_schema.PARTITIONS
   -> WHERE table_schema=DATABASE() AND table_name='t_list'\G;
*************************** 1. row ***************************
    table_name: t_list
partition_name: p0
    table_rows: 1
*************************** 2. row ***************************
    table_name: t_list
```

```
partition_name: p1
    table_rows: 0
2 rows in set (0.00 sec)
```

HASH和KEY分区对NULL的处理方式与RANGE和LIST分区不一样。任何分区函数都会将含有NULL值的记录返回为0，例如：

```
mysql> CREATE TABLE t_hash(
    -> a INT,
    -> b INT)ENGINE=InnoDB
    -> PARTITION BY HASH(b)
    -> PARTITIONS 4;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t_hash SELECT 1,0;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> INSERT INTO t_hash SELECT 1,NULL;
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT table_name,partition_name,table_rows
   -> FROM information_schema.PARTITIONS
   -> WHERE table_schema=DATABASE() AND table_name='t_hash'\G;
*************************** 1. row ***************************
    table_name: t_hash
partition_name: p0
    table_rows: 2
*************************** 2. row ***************************
    table_name: t_hash
partition_name: p1
    table_rows: 0
*************************** 3. row ***************************
    table_name: t_hash
partition_name: p2
    table_rows: 0
*************************** 4. row ***************************
    table_name: t_hash
partition_name: p3
    table_rows: 0
4 rows in set (0.00 sec)
```

10.5 分区和性能

有开发人员常说“对表做个分区，这样数据库的查询就会快了”。这是真的吗？实际上根本感觉不到查询速度的提升，甚至会觉得查询速度急剧下降。因此，在合理使用分区之

前，必须了解分区的使用环境。

数据库的应用分为两类：一类是 OLTP(联机事务处理)，如 Blog、电子商务、网络游戏等；另一类是 OLAP(联机分析处理)，如数据仓库、数据集市等。在一个实际的应用环境中，可能既有 OLTP 的应用，也有 OLAP 的应用。例如，在网络游戏中，玩家操作的游戏数据库应用就是 OLTP 的，但是游戏厂商可能需要对游戏产生的日志进行分析，通过分析得到的结果更好地服务于游戏、预测玩家的行为等，而这是 OLAP 的应用。

对于 OLAP 的应用，分区的确可以很好地提高查询的性能，因为 OLAP 应用的大多数查询需要频繁地扫描一张很大的表。假设有一张 1 亿行的表，其中有一个时间戳属性列，需要从这张表中获取一年的数据。如果按时间戳进行分区，则只需要扫描相应的分区即可。这就是前面介绍的 Partition Pruning 技术。

然而对于 OLTP 的应用，在分区时应该非常小心。在这种应用下，通常不可能会获取一张大表中 10% 的数据，大部分都是通过索引返回几条记录即可。而根据 B+ 树索引的原理可知，对于一张大表，一般的 B+ 树需要 2 ～ 3 次的磁盘 IO 操作，因此 B+ 树可以很好地完成对大表的查询操作，不需要分区的帮助，更何况设计不好的分区会带来严重的性能问题。

很多开发团队会认为含有一千万行的表是一张非常大的表，所以他们往往会选择分区，例如对主键做 10 个 HASH 分区，这样每个分区就只有一百万的数据了，认为这时查询应该变得更快了，比如执行 SELECT * FROM TABLE WHERE PK=@pk。但是考虑这样一个问题：如果一百万行和一千万行的数据本身构成的 B+ 树的层次是一样的，可能都是两层，那么上述主键分区的索引并不会带来性能的提高。假设一千万行数据的 B+ 树的高度是 3，一百万行数据的 B+ 树的高度是 2，这样上述主键分区的索引可以避免一次 IO，从而提高查询的效率。这没问题，但是这张表只有主键索引，没有任何其他列需要查询吗？如果还有这样的语句：SELECT * FROM TABLE WHERE KEY=@key，这时对于 KEY 的查询需要扫描所有的 10 个分区，即使每个分区的查询开销为 2 次 IO 操作，那么一共需要 20 次 IO。而对于原来单表的设计，对 KEY 的查询只需要 2 ～ 3 次 IO 操作。

接着来根据主键 ID 对 Profile 表进行 HASH 分区，HASH 分区的数量为 10，Profile 表有接近一千万行的数据。

```
mysql>CREATE TABLE `Profile` (
    ->   `id` int(11) NOT NULL AUTO_INCREMENT,
    ->   `nickname` varchar(20) NOT NULL DEFAULT '',
    ->   `password` varchar(32) NOT NULL DEFAULT '',
    ->   `sex` char(1) NOT NULL DEFAULT '',
    ->   `rdate` date NOT NULL DEFAULT '0000-00-00',
    ->   PRIMARY KEY (`id`),
    ->   KEY `nickname` (`nickname`)
    ->) ENGINE=InnoDB
    -> PARTITION BY HASH (id)
    -> PARTITIONS 10;
```

```
Query OK, 0 rows affected (1.29 sec)

mysql> SELECT COUNT(nickname) FROM Profile;
*************************** 1. row ***************************
count(1): 9999248
1 row in set (1 min 24.62 sec)
```

因为是 HASH 分区，所以每个分区的记录数大致相同，即数据分布比较均匀：

```
mysql> SELECT table_name,partition_name,table_rows
    -> FROM information_schema.PARTITIONS
    -> WHERE table_schema=DATABASE() AND table_name='Profile'\G;
*************************** 1. row ***************************
    table_name: Profile
partition_name: p0
    table_rows: 990703
*************************** 2. row ***************************
    table_name: Profile
partition_name: p1
    table_rows: 1086519
*************************** 3. row ***************************
    table_name: Profile
partition_name: p2
    table_rows: 976474
*************************** 4. row ***************************
    table_name: Profile
partition_name: p3
    table_rows: 986937
*************************** 5. row ***************************
    table_name: Profile
partition_name: p4
    table_rows: 993667
*************************** 6. row ***************************
    table_name: Profile
partition_name: p5
    table_rows: 978046
*************************** 7. row ***************************
    table_name: Profile
partition_name: p6
    table_rows: 990703
*************************** 8. row ***************************
    table_name: Profile
partition_name: p7
    table_rows: 978639
*************************** 9. row ***************************
    table_name: Profile
partition_name: p8
    table_rows: 1085334
*************************** 10. row ***************************
    table_name: Profile
```

```
    partition_name: p9
        table_rows: 982788
10 rows in set (0.80 sec)
```

注意 即使是根据自增长主键进行的 HASH 分区也不能保证分区数据是均匀的，因为插入的自增长 ID 并非总是连续的，如果该主键值因为某种原因被回滚了，则该值不会再次被自动使用。

如果进行主键的查询，那么可以发现分区的确是有意义的，例如：

```
mysql> EXPLAIN PARTITIONS SELECT * FROM Profile WHERE id=1\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: Profile
   partitions: p1
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 1
        Extra:
1 row in set (0.00 sec)
```

可以发现只寻找了 p1 分区，但是对于 Profile 表中 nickname 列索引的查询，执行分区后会得到如下的结果：

```
mysql> EXPLAIN PARTITIONS
    ->SELECT * FROM Profile WHERE nickname='david'\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: Profile
   partitions: p0,p1,p2,p3,p4,p5,p6,p7,p8,p9
         type: ref
possible_keys: nickname
          key: nickname
      key_len: 62
          ref: const
         rows: 10
        Extra: Using where
1 row in set (0.00 sec)
```

可以看到，MySQL 数据库会搜索所有分区，因此查询速度上会慢很多，将如下语句与上述语句进行比较：

```
mysql>  SELECT * FROM Profile WHERE nickname='david'\G;
*************************** 1. row ***************************
```

```
       id: 5566
nickname: david
password: 3e35d1025659d07ae28e0069ec51ab92
     sex: M
   rdate: 2003-09-20
1 row in set (1.05 sec)
```

上述简单的索引查找语句竟然需要 1.05 秒，这显然是因为查询需要遍历所有分区，且实际的 IO 操作执行了约 20 ～ 30 次。而在未分区的同样结构和大小的表上，执行上述 SQL 语句只需要 0.26 秒。

因此对于使用 InnoDB 存储引擎作为 OLTP 应用的表在使用分区时应该十分小心，在设计时要确认数据的访问模式，否则在 OLTP 应用下分区不仅不会提高查询速度，反而可能会使应用执行得更慢。

10.6 在表和分区间交换数据

MySQL 5.6 开始支持 ALTER TABLE ... EXCHANGE PARTITION 语法。该语句允许分区或子分区中的数据与另一个非分区的表中数据进行交换。如果非分区表的数据为空，那么相当于将分区中的数据移动到非分区表中。若分区表的数据为空，则相当于将外部表中的数据导入分区中。

要使用 ALTER TABLE ... EXCHANGE PARTITION 语句，必须满足下面的条件：

- 要交换的表须与分区表有相同的表结构，但是表不能含有分区。
- 非分区表中的数据必须在交换的分区内定义。
- 被交换的表中不能含有外键，或者其他表中不能含有对该表的外键引用。
- 用户除了需要 ALTER、INSERT 和 CREATE 权限外，还需要 DROP 的权限。

此外，有两个小的细节需要注意：

- 使用该语句时，不会触发交换表和被交换表上的触发器。
- AUTO_INCREMENT 列将被重置。

接着来看一个例子，先创建含有 RANGE 分区的表 e，并填充相应的数据。

```
CREATE TABLE e (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30)
)
PARTITION BY RANGE (id) (
        PARTITION p0 VALUES LESS THAN (50),
        PARTITION p1 VALUES LESS THAN (100),
        PARTITION p2 VALUES LESS THAN (150),
        PARTITION p3 VALUES LESS THAN (MAXVALUE)
```

```
);

INSERT INTO e VALUES
(1669, "Jim", "Smith"),
(337, "Mary", "Jones"),
(16, "Frank", "White"),
(2005, "Linda", "Black");
```

然后创建交换表 e2，表 e2 的结构和表 e 一样，但需要注意的是表 e2 不能含有分区。

```
mysql>CREATE TABLE e2 LIKE e;
Query OK, 0 rows affected (1.34 sec)

mysql>ALTER TABLE e2 REMOVE PARTITIONING;
Query OK, 0 rows affected (0.90 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

通过下列语句观察分区表中的数据：

```
mysql> SELECT PARTITION_NAME, TABLE_ROWS
    ->  FROM INFORMATION_SCHEMA.PARTITIONS
    ->  WHERE TABLE_NAME = 'e';
+----------------+------------+
| PARTITION_NAME | TABLE_ROWS |
+----------------+------------+
| p0             |          1 |
| p1             |          0 |
| p2             |          0 |
| p3             |          3 |
+----------------+------------+
4 rows in set (0.00 sec)
```

因为表 e2 中没有数据，使用如下语句将 e 表的分区 p0 中的数据移动到表 e2 中。

```
mysql> ALTER TABLE e EXCHANGE PARTITION p0 WITH TABLE e2;
Query OK, 0 rows affected (0.28 sec)
```

这时再观察表 e 中分区的数据，可以发现 p0 中的数据已经没有，如下所示：

```
mysql> SELECT PARTITION_NAME, TABLE_ROWS
    ->  FROM INFORMATION_SCHEMA.PARTITIONS
    ->  WHERE TABLE_NAME = 'e';
+----------------+------------+
| PARTITION_NAME | TABLE_ROWS |
+----------------+------------+
| p0             |          0 |
| p1             |          0 |
| p2             |          0 |
| p3             |          3 |
+----------------+------------+
4 rows in set (0.00 sec)
```

这时可以在表 e2 中观察到被移动的数据，如下所示：

```
mysql> SELECT * FROM e2;
+----+-------+-------+
| id | fname | lname |
+----+-------+-------+
| 16 | Frank | White |
+----+-------+-------+
1 row in set (0.00 sec)
```

10.7　小结

本章介绍了 MySQL 数据库中的分区。简单来说，分区是在索引的基础上对表进行水平分区。在 MySQL 数据库中可将分区划分为 RANGE、LIST、HASH、KEY、COLUMNS 和子分区。用户不能过于迷信分区带来的性能提高，只有正确地理解分区、适当应用、合理规划，才能发挥出分区的真正威力。

[General Information]
书名=MYSQL技术内幕 SQL编程
作者=姜承尧著
页数=307
SS号=12950877
出版日期=2012.04