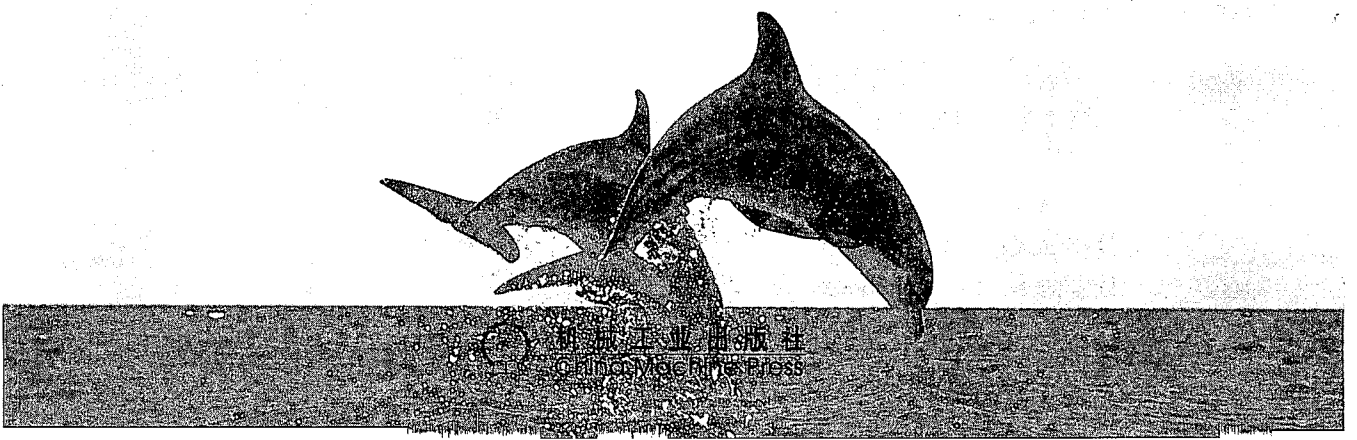
数据库技术丛书

Inside MySQL: InnoDB Storage Engine

MySQL技术内幕
InnoDB存储引擎

姜承尧◎著



机械工业出版社
China Machine Press

本书是国内目前唯一的一本关于InnoDB的著作，由资深MySQL专家亲自执笔，中外数据库专家联袂推荐，权威性毋庸置疑。

内容深入，从源代码的角度深度解析了InnoDB的体系结构、实现原理、工作机制，并给出了大量最佳实践，能帮助你系统而深入地掌握InnoDB，更重要的是，它能为你设计和管理高性能、高可用的数据库系统提供绝佳的指导。注重实战，全书辅有大量的案例，可操作性极强。

全书首先全景式地介绍了MySQL独有的插件式存储引擎，分析了MySQL的各种存储引擎的优势和应用环境；接着以InnoDB的内部实现为切入点，逐一详细讲解了InnoDB存储引擎内部的各个功能模块，包括InnoDB存储引擎的体系结构、内存中的数据结构、基于InnoDB存储引擎的表和页的物理存储、索引与算法、文件、锁、事务、备份，以及InnoDB的性能调优等重要的知识；最后深入解析了InnoDB存储引擎的源代码结构，对大家阅读和理解InnoDB的源代码有重要的指导意义。

本书适合所有希望构建和管理高性能、高可用性的MySQL数据库系统的开发者和DBA阅读。

图书在版编目（CIP）数据

MySQL技术内幕：InnoDB存储引擎 / 姜承尧著. —北京：机械工业出版社，2010.11

ISBN 978-7-111-32188-0

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（2010）第198251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：陈佳媛

北京市荣盛彩色印刷有限公司印刷

2011年1月第1版第1次印刷

186mm × 240 mm　·　25.25印张

标准书号：ISBN 978-7-111-32188-0

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

推 荐 序

It's fair to say that MySQL is the most popular open source database. It has a very large installed base and number of users. Let's see what are the reasons MySQL is so popular, where it stands currently, and maybe touch on some of its future (although predicting the future is rarely successful).

Looking at the customer area of MySQL, which includes Facebook, Flickr, Adobe (in Creative Suite 3), Drupal, Digg, LinkedIn, Wikipedia, eBay, YouTube, Google AdSense (source http://mysql.com/customers/ and public resources), it's obvious that MySQL is everywhere. When you log in to your popular forum (powered by Bulleting) or blog (powered by WordPress), most likely it has MySQL as its backend database. Traditionally, two MySQL's characteristics, simplicity of use and performance, were what allowed it to gain such popularity. In addition to that, availability on a very wide range of platforms (including Windows) and built-in replication, which provides an easy scale-out solution for read-only clients, gave more user attractions and production deployments. There is simple evidence of MySQL's simplicity: In 15 minutes or less, you really can get installed, have a working database, and start running queries and store data. From its early stages MySQL had a good interface to most popular languages for Web development - PHP and Perl, and also Java and ODBC connectors.

There are two best known storage engines in MySQL: MyISAM and InnoDB (I don't cover NDB cluster here; it's a totally different story). MyISAM comes as the default storage engine and historically it is the oldest, but InnoDB is ACID compliant and provides transactions, row-level locking, MVCC, automatic recovery and data corruption detection. This makes it the storage engine you want to choose for your application. Also, there is the third-party transaction storage engine PBXT, with characteristics similar to InnoDB, which is included in the MariaDB distribution.

MySQL's simplicity has its own drawback. Just as it is very easy to start working with it, it is very easy to start getting into trouble with it. As soon as your website or forum gets popular, you

may figure out that the database is a bottleneck, and that you need special skills and tools to fix it.

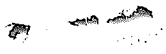
The author of this book is a MySQL expert, especially in InnoDB storage engineB. Hence, I highly recommend this book to new users of InnoDB as well as uers who already have well-tuned InnoDB-based applications but need to get internal out of them.

Vadim Tkachenko

全球知名MySQL数据库服务提供商Percona公司CTO

知名MySQL数据库博客MySQLPerformanceBlog.com作者

《高性能MySQL（第2版）》作者之一



计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

前　言

过去这些年，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企业的核心数据库系统的过程，并且参与了一些大大小小的项目的开发工作，成功地帮助开发人员构建了一些可靠、健壮的应用程序。在这个过程中积累了一些经验，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都会遇到的一些问题，并给出了解决方案。

MySQL数据库独有的插件式存储引擎架构使得它与其他任何数据库都不同，不同的存储引擎有着完全不同的功能，而InnoDB存储引擎的存在使得MySQL跃入了企业级数据库领域。**本书完整地讲解了InnoDB存储引擎中最重要的一些内容，即InnoDB的体系结构和工作原理，并结合InnoDB的源代码讲解了它的内部实现机制。**

本书不仅介绍了InnoDB存储引擎的诸多功能和特性，而且还阐述了如何正确地使用这些功能和特性。更重要的是，它还尝试教大家如何Think Different。Think Different是20世纪90年代苹果公司在其旷日持久的宣传活动中提出的一个口号，借此来重振公司的品牌，更重要的是改变人们对技术在日常生活中的作用的看法。需要注意的是，苹果的口号不是Think Differently，而是Think Different。这里的Different是名词，意味该思考些什么。

很多DBA和开发人员都相信某些“神话”，然而这些“神话”往往都是错误的。无论计算机技术发展的速度变得多快、数据库的使用变得多么简单，任何时候WHY都比WHAT重要。只有真正地理解了内部实现原理、体系结构，才能更好地去使用。这正是人类正确思考问题的原则。因此，对于当前出现的技术，尽管学习应用层面的技术很重要，但更重要的是，应当正确地理解和使用这些技术。

关于这本书，我想实现好几个目标，但最重要的是想告诉大家如下几个简单的观点：

- ❑ 不要相信任何“神话”，学会自己思考。
- ❑ 不要墨守成规，大部分人都知道的事情可能是错误的。
- ❑ 不要相信网上的传言，去测试，根据自己的实践做出决定。
- ❑ 花时间充分地思考，敢于提出质疑。

为什么写本书

当前有关MySQL的书籍大部分都集中在教读者如何使用MySQL，例如SQL语句的使用、复制的搭建、数据的切分等。没错，这对快速掌握和使用MySQL数据库非常有好处，但是真正的数据库工作者需要了解的不仅仅是应用，更多的是内部的具体实现。

MySQL数据库独有的插件式存储引擎结构使得想要在一本书内完整地讲解各个存储引擎变得十分困难。有的书可能偏重于对MyISAM的介绍，有的书则可能偏重于对InnoDB存储引擎的介绍。对于初级的DBA来说，这可能会使他们的理解变得更困难。对于大多数MySQL DBA和开发人员来说，他们往往更希望了解作为MySQL企业级数据库应用的第一存储引擎——InnoDB。我想在本书中，他们可以找到他们想要的内容。

再强调一遍，任何时候WHY都比WHAT重要。本书从源代码的角度对InnoDB的存储引擎的整个体系架构的各个组成部分进行了系统的分析和讲解，剖析了InnoDB存储引擎的核心实现和工作机制，相信这在其他书中是很难找到的。

本书面向的读者

本书不是一本面向应用的数据库类书籍，也不是一本参考手册，更不会教你如何在MySQL中使用SQL语句。本书面向那些使用MySQL InnoDB存储引擎作为数据库后端开发应用程序的开发者和有一定经验的MySQL DBA。书中的大部分例子都是用SQL语句来展示关键特性的，如果想通过本书来了解如何启动MySQL，如何配置Replication环境，可能并不能如愿。不过，通过本书，你将理解InnoDB存储引擎是如何工作的，它的关键特性的功能和作用是什么，以及如何正确地配置和使用这些特性。

如果想更好地使用InnoDB存储引擎，如果想让你的数据库应用获得更好的性能，就请阅读本书。从某种程度上讲，技术经理或总监也要非常了解数据库，要知道数据库对于企业的重要性。如果技术经理或总监想安排员工参加MySQL数据库技术方面的培训，完全可以利用本书来“充电”，相信你一定不会失望的。

要想更好地学习本书，要求具备以下条件：

- 掌握SQL。
- 掌握基本的MySQL操作。
- 接触过一些高级语言，如C、C++、Python或Java。
- 对一些基本算法有所了解，因为本书会分析InnoDB存储引擎的部分源代码，如果你能看懂这些代码，这会对你的理解非常有帮助。

如何阅读本书

本书一共有10章，每一章都像一本“迷你书”，可以单独成册，也就说，你完全可以从书中任何一章开始阅读。例如，要了解第10章中的InnoDB源代码编译和调试的知识，就不必先去阅读第3章有关文件的知识。当然，如果你不太确定自己是否已经对本书所涉及的内容已经完全掌握，建议你系统地阅读本书。

本书不是一本入门书，不会一步步引导你去如何操作。倘若你尚不了解InnoDB存储引擎，本书对你来说可能就显得沉重了些，建议你先查阅官方的API文档，大致掌握InnoDB的基础知识，然后再来阅读本书，相信你会领略到不同的风景。

需要特别说明的是，附录B～D详细给出了Master Thread、Doublewrite、哈希算法和哈希表的源代码，前面学习了这些理论知识，再适当地阅读这些源代码，相信有经验的DBA可以更好地掌握和理解InnoDB存储引擎的本质。为了便于大家阅读，本书在提供源代码下载的同时也将源代码附在了书中，因此占去了一些篇幅，还请大家理解。

致　谢

在编写本书的过程中，我得到了很多朋友的热心帮助。首先要感谢Pecona公司的CEO Peter Zaitsev和CTO Vadim Tkachenko，通过与他们的不断交流，使得我对InnoDB存储引擎有了更进一步的了解，同时知道了怎样才能正确地将InnoDB存储引擎的补丁应用到生产环境。

其次，我要感谢久游网公司的各位同事们。能在才华横溢、充满创意的团队中工作，我感到非常荣幸和兴奋。也因为在这个开放的工作环境中，我才得以不断地进行研究和创新。

此外，我还要感谢我的母亲，写书不是一件容易的事，特别是本书还想传达一些思想，在这个过程中我遇到了很多的困难，感谢她在这个过程中给予我的支持和鼓励。

最后，一份特别的感谢要送给本书的策划编辑杨福川先生和曾珊女士，他们使得本书变得生动和更具有灵魂。

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

目　录

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

第1章 MySQL体系结构和存储引擎

MySQL被设计为一个可移植的数据库，几乎能在当前所有的操作系统上运行，如Linux、Solaris、FreeBSD、Mac和Windows。尽管各种系统在底层（如线程）实现方面各有不同，但是MySQL几乎能保证在各平台上的物理体系结构的一致性。所以，你应该能很好地理解MySQL在所有这些平台上是如何工作的。

1.1 定义数据库和实例

在数据库领域中有两个词很容易混淆，它们就是“实例”（instance）和“数据库”（database）。作为常见的数据库术语，这两个词的定义如下。

- 数据库：物理操作系统文件或其他形式文件类型的集合。在MySQL中，数据库文件可以是frm、myd、myi、ibd结尾的文件。当使用NDB引擎时，数据库的文件可能不是操作系统上的文件，而是存放于内存之中的文件，但是定义仍然不变。
- 数据库实例：由数据库后台进程/线程以及一个共享内存区组成。共享内存可以被运行的后台进程/线程所共享。需要牢记的是，数据库实例才是真正用来操作数据库文件的。

这两个词有时可以互换使用，但两者的概念完全不同。在MySQL中，实例和数据库的通常关系是一一对应，即一个实例对应一个数据库，一个数据库对应一个实例。但是，在集群情况下可能存在一个数据库可被多个实例使用的情况。

MySQL被设计为一个单进程多线程架构的数据库，这点与SQL Server比较类似，但与Oracle多进程的架构有所不同（Oracle的Windows版本也是单进程多线程的架构）。这也就是说，MySQL数据库实例在系统上的表现就是一个进程。

在Linux操作系统中启动MySQL数据库实例，命令如下所示：

```
[root@xen-server bin]# ./mysqld_safe &
```

```
  [root@xen-server bin]# ps -ef | grep mysqld
  root      3441  3258  0 10:23 pts/3    00:00:00
/bin/sh ./mysqld_safe
mysql 3578 3441 0 10:23 pts/3 00:00:00
/usr/local/mysql/libexec/mysqld --basedir=/usr/local/mysql
--datadir=/usr/local/mysql/var --user=mysql
--log-error=/usr/local/mysql/var/xen-server.err
--pid-file=/usr/local/mysql/var/xen-server.pid
--socket=/tmp/mysql.sock --port=3306
  root      3616  3258  0 10:27 pts/3    00:00:00 grep mysqld
```

请注意进程号为3578的进程，该进程就是MySQL实例。这里我们使用了mysqld_safe命令来启动数据库，启动MySQL实例的方法还有很多，在各种平台下的方式可能会有所不同。在这里不一一举例。

当启动实例时，MySQL数据库会去读取配置文件，根据配置文件的参数来启动数据库实例。这与Oracle的参数文件（spfile）相似，不同的是，在Oracle中，如果没有参数文件，启动时会提示找不到该参数文件，数据库启动失败。而在MySQL数据库中，可以没有配置文件，在这种情况下，MySQL会按照编译时的默认参数设置启动实例。用以下命令可以查看，当MySQL数据库实例启动时，它会在哪些位置查找配置文件。

```
[root@xen-server bin]# ./mysql --help | grep my.cnf
order of preference, my.cnf, $MYSQL_TCP_PORT,
/etc/my.cnf /etc/mysql/my.cnf /usr/local/mysql/etc/my.cnf ~/.my.cnf
```

可以看到，MySQL是按/etc/my.cnf→/etc/mysql/my.cnf→/usr/local/mysql/etc/my.cnf→~/.my.cnf的顺序读取配置文件的。可能有人会问：“如果几个配置文件中都有同一个参数，MySQL以哪个配置文件为准？”答案很简单，MySQL会以读取到的最后一个配置文件中的参数为准。在Linux环境下，配置文件一般放在/etc/my.cnf下。在Windows平台下，配置文件的后缀名可以是.cnf，也可能是.ini。运行mysql -help命令，可以找到以下的内容：

```
Default options are read from the following files in the given order:
C:\Windows\my.ini C:\Windows\my.cnf C:\my.ini C:\my.cnf
C:\Program Files\MySQL\M\MySQL Server 5.1\my.cnf
```

配置文件中有一个datadir参数，该参数指定了数据库所在的路径。在Linux操作系统下，datadir默认为/usr/local/mysql/data。当然，你可以修改该参数，或者就使用该路径，但该路径只是一个链接，如下所示：

```
mysql> show variables like 'datadir'\G;
*************************** 1. row ***************************
Variable_name: datadir
        Value: /usr/local/mysql/data/
1 row in set (0.00 sec)1 row in set (0.00 sec)

mysql>system ls -lh /usr/local/mysql/data
total 32K
drwxr-xr-x  2 root mysql 4.0K Aug  6 16:23 bin
drwxr-xr-x  2 root mysql 4.0K Aug  6 16:23 docs
drwxr-xr-x  3 root mysql 4.0K Aug  6 16:04 include
drwxr-xr-x  3 root mysql 4.0K Aug  6 16:04 lib
drwxr-xr-x  2 root mysql 4.0K Aug  6 16:23 libexec
drwxr-xr-x 10 root mysql 4.0K Aug  6 16:23 mysql-test
drwxr-xr-x  5 root mysql 4.0K Aug  6 16:04 share
drwxr-xr-x  5 root mysql 4.0K Aug  6 16:23 sql-bench
lrwxrwxrwx  1 root mysql   16 Aug  6 16:05 data -> /opt/mysql_data/
```

从上面可以看到，其实data目录是一个链接，该链接指向了/opt/mysql_data目录。当然，必须保证/opt/mysql_data的用户和权限，使得只有mysql用户和组可以访问。

1.2 MySQL体系结构

由于工作的缘故，我很大一部分时间都花在对开发人员进行数据库方面的沟通和培训上。此外，不论他们是DBA，还是开发人员，似乎都对MySQL的体系结构了解得不够透彻。很多人喜欢把MySQL与他们以前使用过的SQL Server、Oracle、DB2作比较。因此，我常常会听到这样的疑问：

- 为什么MySQL不支持全文索引？
- MySQL速度快是因为它不支持事务？
- 数据量大于1 000W时，MySQL的性能会急剧下降吗？

……

对于MySQL的疑问还有很多很多，在我解释这些问题之前，我认为，不管使用哪种数据库，了解数据库的体系结构都是最为重要的。

在给出体系结构图之前，我想你应该理解了前一节提出的两个概念：数据库和数据库

实例。很多人会把这两个概念混淆，即MySQL是数据库，MySQL也是数据库实例。你这样来理解Oracle和SQL Server可能是正确的，但这对于以后理解MySQL体系结构中的存储引擎可能会带来问题。从概念上来说，数据库是文件的集合，是依照某种数据模型组织起来并存放于二级存储器中的数据集合；数据库实例是应用程序，是位于用户与操作系统之间的一层数据管理软件，用户对数据库数据的任何操作，包括数据库定义、数据查询、数据维护、数据库运行控制等，都是在数据库实例下进行的，应用程序只有通过数据库实例才能和数据库打交道。

如果这样讲解后你还是觉得不明白，那我再换一种更直白的方式来解释：数据库是由一个个文件组成（一般来说都是二进制的文件）的，如果要对这些文件执行诸如SELECT、INSERT、UPDATE和DETELE之类的操作，不能通过简单的操作文件来更改数据库的内容，需要通过数据库实例来完成对数据库的操作。所以，如果你把Oracle、SQL Server、MySQL简单地理解成数据库，可能是有失偏颇的，虽然在实际使用中我们并不会这么强调两者之间的区别。好了，在给出上述这些复杂枯燥的定义后，现在我们可以来看看MySQL数据库的体系结构了，如图1-1所示。

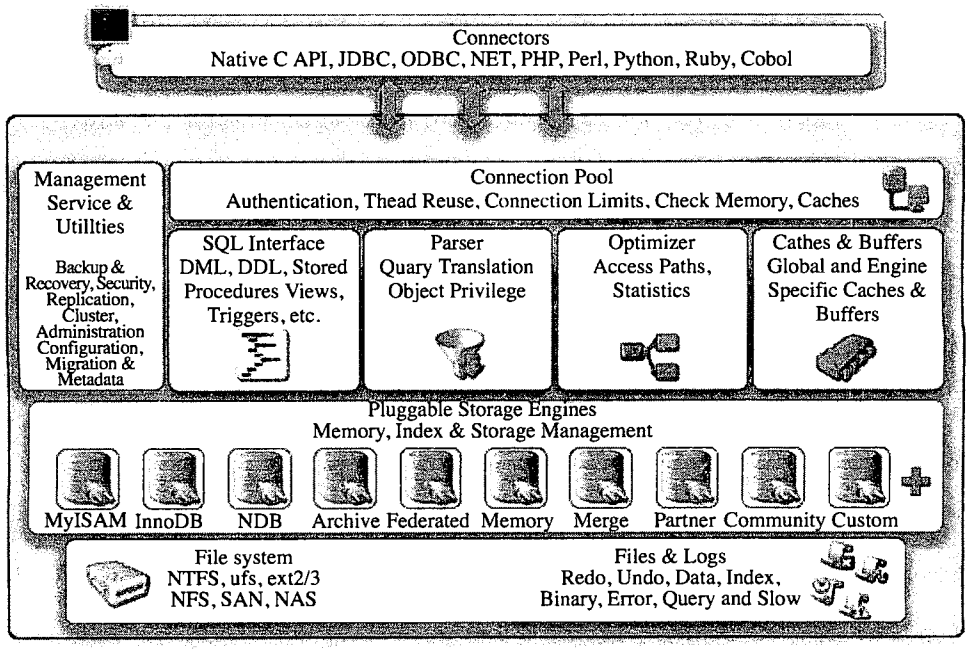


图1-1 MySQL体系结构

从图1-1我们可以发现，MySQL由以下几部分组成：

- 连接池组件。
- 管理服务和工具组件。
- SQL接口组件。
- 查询分析器组件。
- 优化器组件。
- 缓冲（Cache）组件。
- 插件式存储引擎。
- 物理文件。

从图1-1还可以看出，MySQL区别于其他数据库的最重要的特点就是其插件式的表存储引擎。MySQL插件式的存储引擎架构提供了一系列标准的管理和服务支持，这些标准与存储引擎本身无关，可能是每个数据库系统本身都必需的，如SQL分析器和优化器等，而存储引擎是底层物理结构的实现，每个存储引擎开发者都可以按照自己的意愿来进行开发。

注意：存储引擎是基于表的，而不是数据库。请牢牢记住图1-1所示的MySQL体系结构图，它对于你以后深入了解MySQL有极大的帮助。

1.3 MySQL表存储引擎

1.2节大致讲解了MySQL数据库独有的插件式体系结构，并介绍了存储引擎是MySQL区别于其他数据库的一个最重要特性。存储引擎的好处是，每个存储引擎都有各自的特点，能够根据具体的应用建立不同的存储引擎表。对于开发人员来说，存储引擎对其是透明的，但了解各种存储引擎的区别对于开发人员来说也是有好处的。对于DBA来说，他们应该深刻地认识到MySQL的核心是存储引擎。

由于MySQL是开源的，你可以根据MySQL预定义的存储引擎接口编写自己的存储引擎，或者是如果你对某种存储引擎不满意，可以通过修改源码来实现自己想要的特性，这就是开源的魅力所在。比如，eBay的Igor Chernyshev工程师对MySQL Memory存储引擎的

改进（http://code.google.com/p/mysql-heap-dynamic-rows/），并应用于eBay的Personalization Platform，Google和Facebook等公司也对MySQL进行了相类似的修改。我曾尝试过对InnoDB存储引擎的缓冲池进行扩展，为其添加了基于SSD的辅助缓冲池[⊖]，通过利用SSD的高随机读取性能来进一步提高数据库本身的性能。当然，MySQL自身提供的存储引擎已经足够满足绝大多数应用的需求。如果你有兴趣，完全可以开发自己的存储引擎来满足自己特定的需求。MySQL官方手册的第16章给出了编写自定义存储引擎的过程，不过这已超出了本书的范围。

由于MySQL的开源特性，存储引擎可以分为MySQL官方存储引擎和第三方存储引擎。有些第三方存储引擎很强大，如大名鼎鼎的InnoDB存储引擎（现已被Oracle收购），其应用就极其广泛，甚至是MySQL数据库OLTP（Online Transaction Processing，在线事务处理）应用中使用最广泛的存储引擎。还是那句话，我们应该根据具体的应用选择适合的存储引擎，以下是对一些存储引擎的简单介绍，以便于大家选择时参考。

1.3.1 InnoDB存储引擎

InnoDB存储引擎支持事务，主要面向在线事务处理（OLTP）方面的应用。其特点是行锁设计、支持外键，并支持类似于Oracle的非锁定读，即默认情况下读取操作不会产生锁。MySQL 在Windows版本下的InnoDB是默认的存储引擎，同时InnoDB默认地被包含在所有的MySQL二进制发布版本中。

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自身进行管理。从MySQL 4.1（包括4.1）版本开始，它可以将每个InnoDB存储引擎的表单独存放到一个独立的ibd文件中。与Oracle类似，InnoDB存储引擎同样可以使用裸设备（row disk）来建立其表空间。

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。同时使用一种被称为next-key locking的策略来避免幻读（phantom）现象的产生。除此之外，InnoDB储存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

⊖ 详见：http://code.google.com/p/david-mysql-tools/wiki/innodb_secondary_buffer_pool。

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，这种方式类似于Oracle的索引聚集表（index organized table，IOT）。每张表的存储都按主键的顺序存放，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。

1.3.2 MyISAM存储引擎

MyISAM存储引擎是MySQL官方提供的存储引擎。其特点是不支持事务、表锁和全文索引，对于一些OLAP（Online Analytical Processing，在线分析处理）操作速度快。除Windows版本外，是所有MySQL版本默认的存储引擎。

MyISAM存储引擎表由MYD和MYI组成，MYD用来存放数据文件，MYI用来存放索引文件。可以通过使用myisampack工具来进一步压缩数据文件，因为myisampack工具使用赫夫曼（Huffman）编码静态算法来压缩数据，因此使用myisampack工具压缩后的表是只读的，当然你也可以通过myisampack来解压数据文件。

在MySQL 5.0版本之前，MyISAM默认支持的表大小为4G，如果需要支持大于4G的MyISAM表时，则需要制定MAX_ROWS和AVG_ROW_LENGTH属性。从MySQL 5.0版本开始，MyISAM默认支持256T的单表数据，这足够满足一般应用的需求。

注意：对于MyISAM存储引擎表，MySQL数据库只缓存其索引文件，数据文件的缓存交由操作系统本身来完成，这与其他使用LRU算法缓存数据的大部分数据库大不相同。此外，在MySQL 5.1.23版本之前，无论是在32位还是64位操作系统环境下，缓存索引的缓冲区最大只能设置为4G。在之后的版本中，64位系统可以支持大于4G的索引缓冲区。

1.3.3 NDB存储引擎

2003年，MySQL AB公司从Sony Ericsson公司收购了NDB 集群引擎（图1-1中Cluster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 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 lookup）的速度极快，并且通过添加NDB数据存储节点（Data Node）可以线性地提高数据库性能，是高可用、高性能的集群系统。

关于NDB存储引擎，有一个问题值得注意，那就是NDB存储引擎的连接操作（JOIN）是在MySQL数据库层完成的，而不是在存储引擎层完成的。这意味着，复杂的连接操作需要巨大的网络开销，因此查询速度很慢。如果解决了这个问题，NDB存储引擎的市场应该是非常巨大的。

注意：MySQL NDB Cluster存储引擎有社区版本和企业版本，并且NDB Cluster已作为Carrier Grade Edition单独下载版本而存在，可以通过http://dev.mysql.com/downloads/cluster/index.html获得最新版本的NDB Cluster存储引擎。

1.3.4 Memory存储引擎

Memory存储引擎（之前称为HEAP存储引擎）将表中的数据存放在内存中，如果数据库重启或发生崩溃，表中的数据都将消失。它非常适合用于存储临时数据的临时表，以及数据仓库中的纬度表。它默认使用哈希索引，而不是我们熟悉的B+树索引。

虽然Memory存储引擎速度非常快，但在使用上还是有一定的限制。比如，其只支持表锁，并发性能较差，并且不支持TEXT和BLOB列类型。最重要的是，存储变长字段（varchar）时是按照定常字段（char）的方式进行的，因此会浪费内存（这个问题之前已经提到，eBay的Igor Chernyshev工程师已经给出了Patch方案）。

此外有一点常被忽视的是，MySQL数据库使用Memory存储引擎作为临时表来存放查询的中间结果集（intermediate result）。如果中间结果集大于Memory存储引擎表的容量设置，又或者中间结果含有TEXT或BLOB列类型字段，则MySQL数据库会把其转换到MyISAM存储引擎表而存放到磁盘。之前提到MyISAM不缓存数据文件，因此这时产生的临时表的性能对于查询会有损失。

1.3.5 Archive存储引擎

Archive存储引擎只支持INSERT和SELECT操作，MySQL 5.1开始支持索引。其使用zlib算法将数据行（row）进行压缩后存储，压缩比率一般可达1：10。正如其名称所示，Archive存储引擎非常适合存储归档数据，如日志信息。Archive存储引擎使用行锁来实现高并发的插入操作，但是本身并不是事物安全的存储引擎，其设计目标主要是提供高速的插入和压缩功能。

1.3.6 Federated存储引擎

Federated存储引擎表并不存放数据，它只是指向一台远程MySQL数据库服务器上的表。这非常类似于SQL Server的链接服务器和Oracle的透明网关，不同的是，当前Federated存储引擎只支持MySQL数据库表，不支持异构数据库表。

1.3.7 Maria存储引擎

Maria存储引擎是新开发的引擎，设计目标主要是用来取代原有的MyISAM存储引擎，从而成为MySQL的默认存储引擎，开发者是MySQL的创始人之一的Michael Widenius。因此，它可以看作是MyISAM的后续版本。其特点是：缓存数据和索引文件，行锁设计，提供MVCC功能，支持事务和非事务安全的选项支持，以及更好的BLOB字符类型的处理性能。

1.3.8 其他存储引擎

除了上面提到的7种存储引擎外，还有很多其他的存储引擎，包括Merge、CSV、Sphinx和Infobright，它们都有各自适用的场合，这里不再一一做介绍了。了解了MySQL拥有这么多存储引擎后，现在我可以回答1.2节中提到的问题了。

- 为什么MySQL不支持全文索引？不！MySQL支持，MyISAM、Sphinx存储引擎支持全文索引。
- MySQL快是因为不支持事务吗？错！MySQL MyISAM存储引擎不支持事务，但是InnoDB支持。快是相对于不同应用来说的，对于ETL这种操作，MyISAM当然有其

优势。

- 当表的数据量大于1000W时，MySQL的性能会急剧下降吗？不！MySQL是数据库，不是文件，随着数据行数的增加，性能当然会有所下降，但是这些下降不是线性的，如果你选择了正确的存储引擎以及正确的配置，再大的数据量MySQL也是能承受的。如官方手册上提及的，Mytrix和Inc.在InnoDB上存储了超过1TB的数据，还有一些其他网站使用InnoDB存储引擎处理平均每秒800次插入/更新的操作。

1.4 各种存储引擎之间的比较

通过1.3节的介绍，我们了解了存储引擎是MySQL体系结构的核心。本节我们将通过简单比较几个存储引擎来让读者更直观地理解存储引擎的概念。图1-2取自于MySQL的官方手册，展现了一些常用MySQL存储引擎之间的不同之处，包括存储容量的限制、事务支持、锁的粒度、MVCC支持、支持的索引、备份和复制等。

Feature	MyISAM	BDB	Memory	InnoDB	Archive	NDB
Storage Limits	No	No	Yes	64TB	No	Yes
Transactions (commit, rollback, etc.)		✔		✔		
Locking granularity	Table	Page	Table	Row	Row	Row
MVCC/Snapshot Read				✔	✔	✔
Geospatial support	✔					
B-Tree indexes	✔	✔	✔	✔		✔
Hash indexes			✔	✔		✔
Full text search index	✔					
Clustered index				✔		
Data Caches			✔	✔		✔
Index Caches	✔		✔	✔		✔
Compressed data	✔				✔	
Encrypted data (via function)	✔	✔	✔	✔	✔	✔
Storage cost (space used)	Low	Low	N/A	High	Very Low	Low
Memory cost	Low	Low	Medium	High	Low	High
Bulk Insert Speed	High	High	High	Low	Very High	High
Cluster database support						✔
Replication support	✔	✔	✔	✔	✔	✔
Foreign key support				✔		
Backup/Point-in-time recovery	✔	✔	✔	✔	✔	✔
Query cache support	✔	✔	✔	✔	✔	✔
Update Statistics for Data Dictionary	✔	✔	✔	✔	✔	✔

图1-2 不同MySQL存储引擎相关特性的比较

可以看到，每种存储引擎的实现都不相同。有些竟然不支持事务，我相信在任何一本关于数据库原理的书中，都可能会提到数据库与传统文件系统的最大区别在于数据库是支

持事务的。而MySQL的设计者在开发时却认为不是所有的应用都需要事务，所以存在不支持事务的存储引擎。更有不明其理的人把MySQL称作文件系统数据库，其实不然，只是MySQL的设计思想和存储引擎的关系可能让人产生了理解上的偏差。

可以通过SHOW ENGINES语句查看当前使用的MySQL数据库所支持的存储引擎，也可以通过查找information_schema架构下的ENGINES表来查看，如下所示：

```
mysql> show engines\G;
*************************** 1. row ***************************
      Engine: InnoDB
     Support: YES
     Comment: Supports transactions, row-level locking, and foreign keys
Transactions: YES
          XA: YES
  Savepoints: YES
*************************** 2. row ***************************
      Engine: MRG_MYISAM
     Support: YES
     Comment: Collection of identical MyISAM tables
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 3. row ***************************
      Engine: BLACKHOLE
     Support: YES
     Comment: /dev/null storage engine (anything you write to it disappears)
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 4. row ***************************
      Engine: CSV
     Support: YES
     Comment: CSV storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 5. row ***************************
      Engine: MEMORY
     Support: YES
     Comment: Hash based, stored in memory, useful for temporary tables
Transactions: NO
          XA: NO
  Savepoints: NO
```

```
*************************** 6. row ***************************
      Engine: FEDERATED
     Support: NO
     Comment: Federated MySQL storage engine
Transactions: NULL
          XA: NULL
  Savepoints: NULL
*************************** 7. row ***************************
      Engine: ARCHIVE
     Support: YES
     Comment: Archive storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 8. row ***************************
      Engine: MyISAM
     Support: DEFAULT
     Comment: Default engine as of MySQL 3.23 with great performance
Transactions: NO
          XA: NO
  Savepoints: NO
8 rows in set (0.00 sec)
```

下面我将通过MySQL提供的示例数据库来简单显示各种存储引擎之间的区别。我们将分别运行以下语句，然后统计每次使用各种存储引擎后表的大小。

```
mysql> create table mytest engine=myisam as select * from salaries;
Query OK, 2844047 rows affected (4.37 sec)
Records: 2844047  Duplicates: 0  Warnings: 0

mysql> alter table mytest engine=innodb;
Query OK, 2844047 rows affected (15.86 sec)
Records: 2844047  Duplicates: 0  Warnings: 0

mysql> alter table mytest engine=ARCHIVE;
Query OK, 2844047 rows affected (16.03 sec)
Records: 2844047  Duplicates: 0  Warnings: 0
```

通过每次的统计我们发现，当最初的表使用MyISAM存储引擎时，表的大小为40.7MB，使用InnoDB存储引擎时表增大到了113.6MB，而使用Archive存储引擎时表的大小却只有20.2MB。该示例只从表的大小方面简单地揭示了各存储引擎的不同。

注意： MySQL提供了一个非常好的用来演示MySQL各项功能的示例数据库，如SQL Server提供的AdventureWorks示例数据库和Oracle提供的示例数据库。就我所知，知道MySQL示例数据库的人很少，可能是因为这个示例数据库没有在安装的时候提示用户是否安装（如Oracle和SQL Server），以及这个示例数据库的下载竟然和文档放在一起。可以通过以下链接找到示例数据库的下载地址：http://dev.mysql.com/doc/。

1.5 连接MySQL

本节将介绍连接MySQL数据库的常用方式。需要理解的是，连接MySQL操作是连接进程和MySQL数据库实例进行通信。从开发的角度来说，本质上是进程通信。如果对进程通信比较了解，应该知道常用的进程通信方式有管道、命名管道、命名字、TCP/IP套接字、Unix域名套接字。MySQL提供的连接方式从本质上看都是上述提及的进程通信方式。

1.5.1 TCP/IP

TCP/IP套接字方式是MySQL在任何平台下都提供的连接方式，也是网络中使用得最多的一种方式。这种方式在TCP/IP连接上建立一个基于网络的连接请求，一般情况下客户端在一台服务器上，而MySQL实例在另一台服务器上，这两台机器通过一个TCP/IP网络连接。例如，我可以在Windows服务器下请求一台远程Linux服务器下的MySQL实例，如下所示。

```
C:\>mysql -h192.168.0.101 -u david -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 18358
Server version: 5.0.77-log MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

这里的客户端是Windows，它向一台Host IP为192.168.0.101的MySQL实例发起了TCP/IP连接请求，并且连接成功。之后，就可以对MySQL数据库进行一些数据库操作，如DDL和DML等。

这里需要注意的是，在通过TCP/IP连接到MySQL实例时，MySQL会先检查一张权限视图，用来判断发起请求的客户端IP是否允许连接到MySQL实例。该视图在mysql库下，表名为user，如下所示。

```
mysql> use mysql;
Database changed
mysql> select host,user,password from user;
*************************** 1. row ***************************
host: 192.168.24.%
user: root
password: *75DBD4FA548120B54FE693006C41AA9A16DE8FBE
*************************** 2. row ***************************
host: nineyou0-43
user: root
password: *75DBD4FA548120B54FE693006C41AA9A16DE8FBE
*************************** 3. row ***************************
host: 127.0.0.1
user: root
password: *75DBD4FA548120B54FE693006C41AA9A16DE8FBE
*************************** 4. row ***************************
host: 192.168.0.100
user: zlm
password: *DAE0939275CC7CD8E0293812A31735DA9CF0953C
*************************** 5. row ***************************
host: %
user: david
password:
5 rows in set (0.00 sec)
```

从这张权限表中可以看到，MySQL允许david这个用户在任何IP段下连接该实例，并且不需要密码。此外，还给出了root用户在各个网段下的访问控制权限。

1.5.2 命名管道和共享内存

在Windows 2000、Windows XP、Windows 2003和Windows Vista以及在此之后的Windows操作系统中，如果两个需要通信的进程在同一台服务器上，那么可以使用命名管

道，SQL Server数据库默认安装后的本地连接也使用命名管道。在MySQL数据库中，需在配置文件中启用--enable-named-pipe选项。在MySQL 4.1之后的版本中，MySQL还提供了共享内存的连接方式，在配置文件中添加--shared-memory。如果想使用共享内存的方式，在连接时，Mysql客户端还必须使用-protocol=memory选项。

1.5.3 Unix域套接字

在Linux和Unix环境下，还可以使用Unix域套接字。Unix域套接字其实不是一个网络协议，所以只能在MySQL客户端和数据库实例在同一台服务器上的情况下使用。你可以在配置文件中指定套接字文件的路径，如-socket=/tmp/mysql.sock。当数据库实例启动后，我们可以通过下列命令来进行Unix域套接字文件的查找：

```
mysql> show variables like 'socket';
*************************** 1. row ***************************
Variable_name: socket
        Value: /tmp/mysql.sock
1 row in set (0.00 sec)
```

知道了Unix套接字文件的路径后，就可以使用该方式进行连接了，如下所示：

```
[root@stargazer ~]# mysql -udavid -S /tmp/mysql.sock
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 20333
Server version: 5.0.77-log MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

1.6 小结

本章首先介绍了数据库和数据库实例的定义，紧接着分析了MySQL的体系结构，从而进一步突出强调了“实例”和“数据库”的区别。我相信，不管是MySQL DBA还是MySQL的开发人员，都应该已经从宏观上了解了MySQL的体系结构，特别是MySQL独有的插件式存储引擎的概念。因为很多MySQL用户很少意识到这一点，从而给他们的管理、使用和开发带来了困扰。

本章还详细讲解了各种常见的表存储引擎的特性、适用情况以及它们之间的区别，以便于大家在选择存储引擎时作为参考。最后强调一点，虽然MySQL有许多存储引擎，但是它们之间不存在优劣性的差异，我们应根据不同的应用选择适合自己的存储引擎。当然，如果你能力很强，完全可以修改存储引擎的源代码，甚至是创建自己需要的特定的存储引擎，这不就是开源的魅力吗？

第2章 InnoDB存储引擎

InnoDB是事务安全的MySQL存储引擎，设计上采用了类似于Oracle的架构。一般而言，在OLTP的应用中，InnoDB应该作为核心应用表的首选存储引擎。同时，也是因为InnoDB的存在，才使得MySQL变得更有魅力。本章将详细介绍InnoDB存储引擎的体系架构及其不同于其他数据库的特性。

2.1 InnoDB存储引擎概述

InnoDB由Innobase Oy公司[⊖]开发，被包括在MySQL所有的二进制发行版本中，是Windows下默认的表存储引擎。该存储引擎是第一个完整支持ACID事务的MySQL存储引擎（BDB是第一个支持事务的MySQL存储引擎，现在已经停止开发），行锁设计，支持MVCC，提供类似于Oracle风格的一致性非锁定读，支持外键，被设计用来最有效地利用内存和CPU。如果你熟悉Oracle的架构，你会发现InnoDB与Oracle很类似，也许这也是为什么Oracle要急于在MySQL AB之前收购该公司的原因。

Heikki Tuuri（1964年出生于芬兰赫尔辛基）是InnoDB存储引擎的创始人，与著名的Linux创始人Linus同是芬兰赫尔辛基大学校友。在1990年完成赫尔辛基大学的数学逻辑博士学位后，他于1995年成立Innobase Oy公司并担任CEO。同时，我欣喜地注意到，在InnoDB存储引擎的开发团队中，有来自中国科技大学的Calvin Sun。

InnoDB存储引擎已经被许多大型网站使用，例如我们熟知的Yahoo、Facebook、youtube、Flickr，在网络游戏领域有Wow、SecondLife、神兵玄奇等。淘宝网正在有计划地将一部分核心应用由Oracle转到MySQL，目前他们的存储引擎选择是InnoDB。我不是MySQL的布道者，也不是InnoDB的鼓吹者，但是我认为，如果实施一个新的OLTP项目不

⊖ 该公司2006年已经被Oracle公司收购。

使用MySQL InnoDB存储引擎将是多么愚蠢。

从MySQL的官方手册还能得知，著名的Internet新闻站点Slashdot.org运行在InnoDB上。Mytrix, Inc.在InnoDB上存储超过1TB的数据，还有一些其他站点在InnoDB上处理平均每秒800次插入/更新的操作。这些都证明了InnoDB是一个高性能、高可用、高可扩展的存储引擎。

InnoDB和MySQL一样，在GNU GPL 2下发行。有关更多MySQL证书的信息，可参考http://www.mysql.com/company/legal/licensing/，这里不再做过多介绍。

2.2 InnoDB体系架构

通过第1章我们了解了MySQL的体系结构，现在可能你想更深入地了解InnoDB的架构模型。图2-1简单显示了InnoDB的存储引擎的体系架构。InnoDB有多个内存块，你可以认为这些内存块组成了一个大的内存池，负责如下工作：

- 维护所有进程/线程需要访问的多个内部数据结构。
- 缓存磁盘上的数据，方便快速地读取，并且在对磁盘文件的数据进行修改之前在这里缓存。
- 重做日志（redo log）缓冲。

……

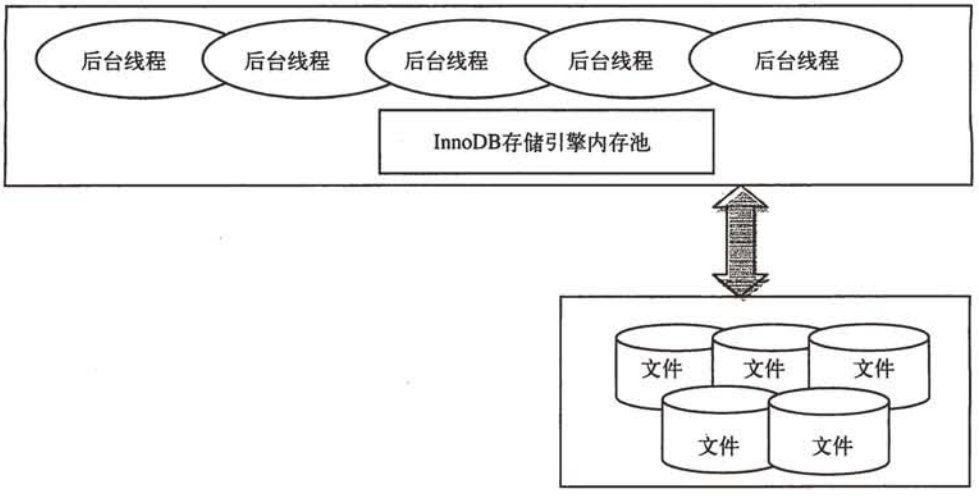


图2-1 InnoDB体系结构

后台线程的主要作用是负责刷新内存池中的数据，保证缓冲池中的内存缓存的是最近的数据。此外，将已修改的数据文件刷新到磁盘文件，同时保证在数据库发生异常情况下InnoDB能恢复到正常运行状态。

2.2.1 后台线程

由于Oracle是多进程的架构（Windows下除外），因此可以通过一些很简单的命令来得知Oracle当前运行的后台进程，如ipcs命令。一般来说，Oracle的核心后台进程有CKPT、DBWn、LGWR、ARCn、PMON、SMON等。

很多DBA问我，InnoDB存储引擎是否也是这样的架构，只不过是多线程版本的实现后，我决定去看InnoDB的源代码，发现InnoDB并不是这样对数据库进程进行操作的。InnoDB存储引擎是在一个被称做master thread的线程上几乎实现了所有的功能。

默认情况下，InnoDB存储引擎的后台线程有7个——4个IO thread，1个master thread，1个锁（lock）监控线程，1个错误监控线程。IO thread的数量由配置文件中的innodb_file_io_threads参数控制，默认为4，如下所示。

```
mysql> show engine innodb status\G;
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
100719 21:34:03 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 50 seconds
......
--------
FILE I/O
--------
I/O thread 0 state: waiting for i/o request (insert buffer thread)
I/O thread 1 state: waiting for i/o request (log thread)
I/O thread 2 state: waiting for i/o request (read thread)
I/O thread 3 state: waiting for i/o request (write thread)
Pending normal aio reads: 0, aio writes: 0,
 ibuf aio reads: 0, log i/o's: 0, sync i/o's: 0
Pending flushes (fsync) log: 0; buffer pool: 0
45 OS file reads, 562 OS file writes, 412 OS fsyncs
```

```
0.00 reads/s, 0 avg bytes/read, 0.00 writes/s, 0.00 fsyncs/s
......
-----------------------------
END OF INNODB MONITOR OUTPUT
=============================

1 row in set (0.00 sec)
```

可以看到，4个IO线程分别是insert buffer thread、log thread、read thread、write thread。在Linux平台下，IO thread的数量不能进行调整，但是在Windows平台下可以通过参数innodb_file_io_threads来增大IO thread。InnoDB Plugin版本开始增加了默认IO thread的数量，默认的read thread和write thread分别增大到了4个，并且不再使用innodb_file_io_threads参数，而是分别使用innodb_read_io_threads和innodb_write_io_threads参数，如下所示。

```
mysql> show variables like 'innodb_version'\G;
*************************** 1. row ***************************
Variable_name: innodb_version
        Value: 1.0.6
1 row in set (0.00 sec)

mysql> show variables like 'innodb_%io_threads'\G;
*************************** 1. row ***************************
Variable_name: innodb_read_io_threads
        Value: 4
*************************** 2. row ***************************
Variable_name: innodb_write_io_threads
        Value: 4
2 rows in set (0.00 sec)

mysql> show engine innodb status\G;
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
100719 21:55:26 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 36 seconds
......
--------
```

```
FILE I/O
--------
I/O thread 0 state: waiting for i/o request (insert buffer thread)
I/O thread 1 state: waiting for i/o request (log thread)
I/O thread 2 state: waiting for i/o request (read thread)
I/O thread 3 state: waiting for i/o request (read thread)
I/O thread 4 state: waiting for i/o request (read thread)
I/O thread 5 state: waiting for i/o request (read thread)
I/O thread 6 state: waiting for i/o request (write thread)
I/O thread 7 state: waiting for i/o request (write thread)
I/O thread 8 state: waiting for i/o request (write thread)
I/O thread 9 state: waiting for i/o request (write thread)
Pending normal aio reads: 0, aio writes: 0,
 ibuf aio reads: 0, log i/o's: 0, sync i/o's: 0
Pending flushes (fsync) log: 0; buffer pool: 0
3229856 OS file reads, 7830947 OS file writes, 1601902 OS fsyncs
reads/s, 0 avg bytes/read, 0.00 writes/s, 0.00 fsyncs/s
......
-----------------------------
END OF INNODB MONITOR OUTPUT
=============================

1 row in set (0.01 sec)
```

在Windows下，我们还可以通过Visual Studio来调试MySQL，并设置断点来观察所有的线程信息，如图2-2所示。

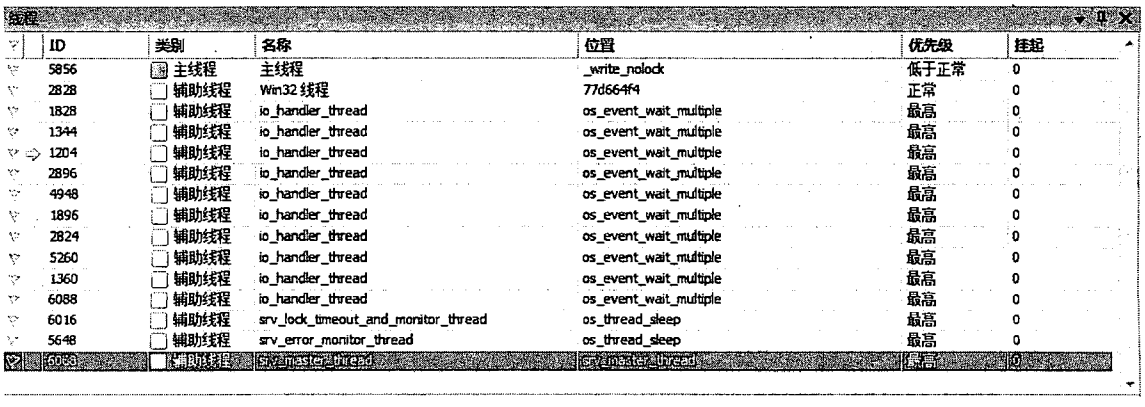


图2-2 Windows下InnoDB存储引擎的线程

2.2.2 内存

InnoDB存储引擎内存由以下几个部分组成：缓冲池（buffer pool）、重做日志缓冲池（redo log buffer）以及额外的内存池（additional memory pool），分别由配置文件中的参数innodb_buffer_pool_size和innodb_log_buffer_size的大小决定。以下显示了一台MySQL数据库服务器，它将InnoDB存储引擎的缓冲池、重做日志缓冲池以及额外的内存池分别设置为2.5G、8M和8M（分别以字节显示）。

```
mysql> show variables like 'innodb_buffer_pool_size'\G;
*************************** 1. row ***************************
Variable_name: innodb_buffer_pool_size
        Value: 2621440000
1 row in set (0.00 sec)

mysql> show variables like 'innodb_log_buffer_size'\G;
*************************** 1. row ***************************
Variable_name: innodb_log_buffer_size
        Value: 8388608
1 row in set (0.00 sec)

mysql> show variables like 'innodb_additional_mem_pool_size'\G;
*************************** 1. row ***************************
Variable_name: innodb_additional_mem_pool_size
        Value: 8388608
1 row in set (0.00 sec)
```

缓冲池是占最大块内存的部分，用来存放各种数据的缓存。因为InnoDB的存储引擎的工作方式总是将数据库文件按页（每页16K）读取到缓冲池，然后按最近最少使用（LRU）的算法来保留在缓冲池中的缓存数据。如果数据库文件需要修改，总是首先修改在缓存池中的页（发生修改后，该页即为脏页），然后再按照一定的频率将缓冲池的脏页刷新（flush）到文件。可以通过命令SHOW ENGINE INNODB STATUS来查看innodb_buffer_pool的具体使用情况，如下所示。

```
mysql> show engine innodb status\G;
*************************** 1. row ***************************
Status:
=====================================
090921 10:55:03 INNODB MONITOR OUTPUT
```

```
=====================================
Per second averages calculated from the last 24 seconds
......
----------------------
BUFFER POOL AND MEMORY
----------------------
......
Buffer pool size   65536
Free buffers       51225
Database pages     12986
Modified db pages  8
......
```

在BUFFER POOL AND MEMORY里可以看到InnoDB存储引擎缓冲池的使用情况，buffer pool size表明了一共有多少个缓冲帧（buffer frame），每个buffer frame为16K，所以这里一共分配了65536*16/1024=1G内存的缓冲池。Free buffers表示当前空闲的缓冲帧，Database pages表示已经使用的缓冲帧，Modified db pages表示脏页的数量。就当前状态看来，这台数据库的压力并不大，因为在缓冲池中有大量的空闲页可供数据库进一步使用。

注意：show engine innodb status的命令显示的不是当前的状态，而是过去某个时间范围内InnoDB存储引擎的状态，从上面的示例中我们可以看到，Per second averages calculated from the last 24 seconds表示的信息是过去24秒内的数据库状态。

具体来看，缓冲池中缓存的数据页类型有：索引页、数据页、undo页、插入缓冲（insert buffer）、自适应哈希索引（adaptive hash index）、InnoDB存储的锁信息（lock info）、数据字典信息（data dictionary）等。不能简单地认为，缓冲池只是缓存索引页和数据页，它们只是占缓冲池很大的一部分而已。图2-3很好地显示了InnoDB存储引擎中内存的结构情况。

参数innodb_buffer_pool_size指定了缓冲池的大小，在32位Windows系统下，参数innodb_buffer_pool_awe_mem_mb还可以启用地址窗口扩展（AWE）功能，突破32位下对于内存使用的限制。但是，在使用这个参数的时候需要注意，一旦启用AWE功能，InnoDB存储引擎将自动禁用自适应哈希索引（adaptive hash index）的功能。

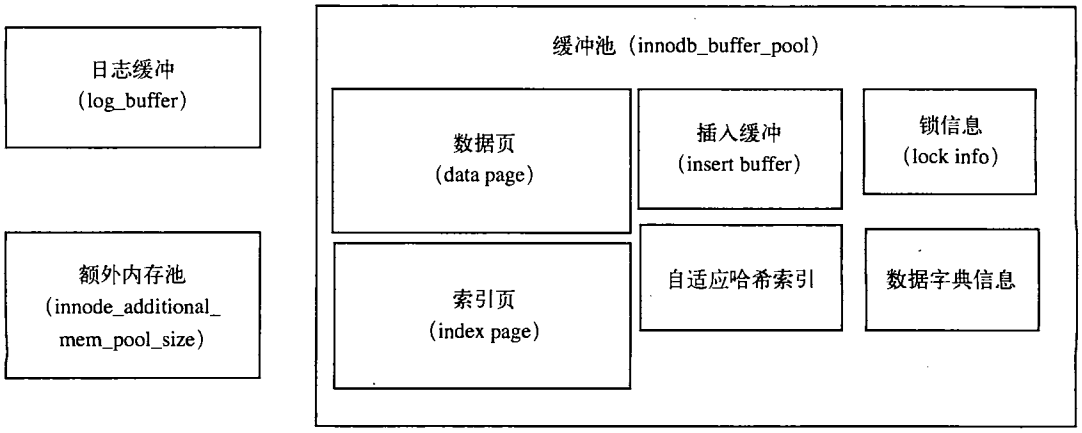


图2-3 InnoDB存储引擎内存结构

日志缓冲[⊖]将重做日志信息先放入这个缓冲区，然后按一定频率将其刷新到重做日志文件。该值一般不需要设置为很大，因为一般情况下每一秒钟就会将重做日志缓冲刷新到日志文件，因此我们只需要保证每秒产生的事务量在这个缓冲大小之内即可。

额外的内存池通常被DBA忽略，认为该值并不是十分重要，但恰恰相反的是，该值其实同样十分重要。在InnoDB存储引擎中，对内存的管理是通过一种称为内存堆（heap）的方式进行的。在对一些数据结构本身分配内存时，需要从额外的内存池中申请，当该区域的内存不够时，会从缓冲池中申请。InnoDB实例会申请缓冲池（innodb_buffer_pool）的空间，但是每个缓冲池中的帧缓冲（frame buffer）还有对应的缓冲控制对象（buffer control block），而且这些对象记录了诸如LRU、锁、等待等方面的信息，而这个对象的内存需要从额外内存池中申请。因此，当你申请了很大的InnoDB缓冲池时，这个值也应该相应增加。

2.3 master thread

通过对前一小节的学习我们已经知道，InnoDB存储引擎的主要工作都是在一个单独的后台线程master thread中完成的。这一节我们将具体解释该线程的具体实现以及该线程可能存在的问题。

⊖ 严格地说，应该是重做（redo）日志缓冲。

2.3.1 master thread源码分析

master thread的线程优先级别最高。其内部由几个循环（loop）组成：主循环（loop）、后台循环（background loop）、刷新循环（flush loop）、暂停循环（suspend loop）。master thread会根据数据库运行的状态在loop、background loop、 flush loop和suspend loop中进行切换。

loop称为主循环，因为大多数的操作都在这个循环中，其中有两大部分操作：每秒钟的操作和每10秒的操作。伪代码如下：

```
void master_thread(){
loop:
for(int i = 0; i < 10; i++){
    do thing once per second
    sleep 1 second if necessary
}
do things once per ten seconds
goto loop;
}
```

可以看到，loop循环通过thread sleep来实现，这意味着所谓的每秒一次或每10秒一次的操作是不精确的。在负载很大的情况下可能会有延迟（delay），只能说大概在这个频率下。当然，InnoDB源代码中还采用了其他的方法来尽量保证这个频率。

每秒一次的操作包括：

- 日志缓冲刷新到磁盘，即使这个事务还没有提交（总是）。
- 合并插入缓冲（可能）。
- 至多刷新100个InnoDB的缓冲池中的脏页到磁盘（可能）。
- 如果当前没有用户活动，切换到background loop（可能）。

即使某个事务还没有提交，InnoDB存储引擎仍然会每秒将重做日志缓冲中的内容刷新到重做日志文件。这一点是必须知道的，这可以很好地解释为什么再大的事务commit的时间也是很快的。

合并插入缓冲（insert buffer）并不是每秒都发生。InnoDB存储引擎会判断当前一秒内发生的IO次数是否小于5次，如果小于5次，InnoDB认为当前的IO压力很小，可以执行合并插入缓冲的操作。

同样，刷新100个脏页也不是每秒都在发生。InnoDB存储引擎通过判断当前缓冲池中脏页的比例（buf_get_modified_ratio_pct）是否超过了配置文件中innodb_max_dirty_pages_pct这个参数（默认为90，代表90%），如果超过了这个阈值，InnoDB存储引擎认为需要做磁盘同步操作，将100个脏页写入磁盘。

总结上述3个操作，伪代码可以进一步具体化，如下所示：

```
void master_thread(){
    goto loop;
loop:
for(int i = 0; i<10; i++){
    thread_sleep(1) // sleep 1 second
    do log buffer flush to disk
    if (last_one_second_ios < 5 )
        do merge at most 5 insert buffer
    if ( buf_get_modified_ratio_pct > innodb_max_dirty_pages_pct )
        do buffer pool flush 100 dirty page
    if ( no user activity )
        goto backgroud loop
}
do things once per ten seconds
background loop:
    do something
    goto loop:
}
```

接着来看每10秒的操作，包括如下内容：

- 刷新100个脏页到磁盘（可能）。
- 合并至多5个插入缓冲（总是）。
- 将日志缓冲刷新到磁盘（总是）。
- 删除无用的Undo页（总是）。
- 刷新100个或者10个脏页到磁盘（总是）。
- 产生一个检查点（总是）。

在以上的过程中，InnoDB存储引擎会先判断过去10秒之内磁盘的IO操作是否小于200次。如果是，InnoDB存储引擎认为当前有足够的磁盘IO操作能力，因此将100个脏页刷新到磁盘。接着，InnoDB存储引擎会合并插入缓冲。不同于每1秒操作时可能发生的合并插

入缓冲操作，这次的合并插入缓冲操作总会在这个阶段进行。之后，InnoDB存储引擎会再执行一次将日志缓冲刷新到磁盘的操作，这与每秒发生的操作是一样的。

接着InnoDB存储引擎会执行一步full purge操作，即删除无用的Undo页。对表执行update、delete这类操作时，原先的行被标记为删除，但是因为一致性读（consistent read）的关系，需要保留这些行版本的信息。但是在full purge过程中，InnoDB存储引擎会判断当前事务系统中已被删除的行是否可以删除，比如有时候可能还有查询操作需要读取之前版本的Undo信息，如果可以，InnoDB会立即将其删除。从源代码中可以发现，InnoDB存储引擎在操作full purge时，每次最多删除20个Undo页。

然后，InnoDB存储引擎会判断缓冲池中脏页的比例（buf_get_modified_ratio_pct），如果有超过70%的脏页，则刷新100个脏页到磁盘；如果脏页的比例小于70%，则只需刷新10%的脏页到磁盘。

最后，InnoDB存储引擎会产生一个检查点（checkpoint），InnoDB存储引擎的检查点也称为模糊检查点（fuzzy checkpoint）。InnoDB存储引擎在checkpoint时并不会把所有缓冲池中的脏页都写入磁盘，因为这样可能会对性能产生影响，而只是将最老日志序列号（oldest LSN）的页写入磁盘。

现在，我们可以完整地把主循环（main loop）的伪代码写出来了，内容如下：

```
void master_thread(){
    goto loop;
loop:
for(int i = 0; i<10; i++){
    thread_sleep(1) // sleep 1 second
    do log buffer flush to disk
    if (last_one_second_ios < 5 )
        do merge at most 5 insert buffer
    if ( buf_get_modified_ratio_pct > innodb_max_dirty_pages_pct )
        do buffer pool flush 100 dirty page
    if ( no user activity )
        goto backgroud loop
}
if ( last_ten_second_ios < 200 )
    do buffer pool flush 100 dirty page
do merge at most 5 insert buffer
do log buffer flush to disk
```

```
do full purge
if ( buf_get_modified_ratio_pct > 70% )
    do buffer pool flush 100 dirty page
else
    buffer pool flush 10 dirty page
do fuzzy checkpoint
goto loop
background loop:
    do something
goto loop:
}
```

接着来看background loop，若当前没有用户活动（数据库空闲时）或者数据库关闭时，就会切换到这个循环。这个循环会执行以下操作：

- 删除无用的Undo页（总是）。
- 合并20个插入缓冲（总是）。
- 跳回到主循环（总是）。
- 不断刷新100个页，直到符合条件（可能，跳转到flush loop中完成）。

如果flush loop中也没有什么事情可以做了，InnoDB存储引擎会切换到suspend_loop，将master thread挂起，等待事件的发生。若启用了InnoDB存储引擎，却没有使用任何InnoDB存储引擎的表，那么master thread总是处于挂起状态。

最后，master thread完整的伪代码如下：

```
void master_thread(){
    goto loop;
loop:
for(int i = 0; i<10; i++){
    thread_sleep(1) // sleep 1 second
    do log buffer flush to disk
    if ( last_one_second_ios < 5 )
        do merge at most 5 insert buffer
    if ( buf_get_modified_ratio_pct > innodb_max_dirty_pages_pct )
        do buffer pool flush 100 dirty page
    if ( no user activity )
        goto backgroud loop
}
if ( last_ten_second_ios < 200 )
    do buffer pool flush 100 dirty page
do merge at most 5 insert buffer
```

```
do log buffer flush to disk
do full purge
if ( buf_get_modified_ratio_pct > 70% )
    do buffer pool flush 100 dirty page
else
    buffer pool flush 10 dirty page
do fuzzy checkpoint
goto loop
background loop:
do full purge
do merge 20 insert buffer
if not idle:
goto loop:
else:
    goto flush loop
flush loop:
do buffer pool flush 100 dirty page
if ( buf_get_modified_ratio_pct> innodb_max_dirty_pages_pct )
    goto flush loop
goto suspend loop
suspend loop:
suspend_thread()
waiting event
goto loop;
}
```

从InnoDB Plugin开始，用命令SHOW ENGINE INNODB STATUS可以查看当前master thread的状态信息，如下所示：

```
mysql> show engine innodb status\G;
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
090921 14:24:56 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 6 seconds
----------
BACKGROUND THREAD
----------
srv_master_thread loops: 45 1_second, 45 sleeps, 4 10_second, 6 background, 6 flush
srv_master_thread log flush and writes: 45   log writes only: 69
......
```

这里可以看到主循环执行了45次，每秒sleep的操作执行了45次（说明负载不是很大），10秒一次的活动执行了4次，符合1：10。background loop执行了6次，flush loop执行了6次。因为当前这台服务器的压力很小，所以能在理论值上运行。但是，如果是在一台压力很大的MySQL服务器上，我们看到的可能会是下面的情景：

```
mysql> show engine innodb status\G;
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
091009 10:14:34 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 42 seconds
----------
BACKGROUND THREAD
----------
srv_master_thread loops: 2188 1_second, 1537 sleeps, 218 10_second, 2
background, 2 flush
srv_master_thread log flush and writes: 1777  log writes only: 5816
......
```

可以看到当前主循环运行了2188次，但是循环中的每一秒钟SLEEP的操作只运行了1537次。这是因为InnoDB对其内部进行了一些优化，当压力大时并不总是等待1秒。所以说，我们并不能认为1_second和sleeps的值总是相等的。在某些情况下，可以通过两者之间差值的比较来反映当前数据库的负载压力。

2.3.2 master thread的潜在问题

在了解了master thread的具体实现过程后，我们会发现InnoDB存储引擎对于IO其实是有限制的，在缓冲池向磁盘刷新时其实都做了一定的硬性规定（hard coding）。在磁盘技术飞速发展的今天，当固态磁盘出现时，这种规定在很大程度上限制了InnoDB存储引擎对磁盘IO的性能，尤其是写入性能。

从前面的伪代码来看，无论何时，InnoDB存储引擎最多都只会刷新100个脏页到磁盘，合并20个插入缓冲。如果是在密集写的应用程序中，每秒中可能会产生大于100个的脏页，或是产生大于20个插入缓冲，此时master thread似乎会“忙不过来”，或者说它总是做得很

慢。即使磁盘能在1秒内处理多于100个页的写入和20个插入缓冲的合并，由于hard coding，master thread也只会选择刷新100个脏页和合并20个插入缓冲。同时，当发生宕机需要恢复时，由于很多数据还没有刷新回磁盘，所以可能会导致恢复需要很快的时间，尤其是对于insert buffer。

这个问题最初是由Google的工程师Mark Callaghan提出的，之后InnoDB对其进行了修正并发布了补丁。InnoDB存储引擎的开发团队参考了Google的patch，提供了类似的方法来修正该问题。因此InnoDB Plugin开始提供了一个参数，用来表示磁盘IO的吞吐量，参数为innodb_io_capacity，默认值为200。对于刷新到磁盘的数量，会按照innodb_io_capacity的百分比来刷新相对数量的页。规则如下：

- 在合并插入缓冲时，合并插入缓冲的数量为innodb_io_capacity数值的5%。
- 在从缓冲区刷新脏页时，刷新脏页的数量为innodb_io_capacity。

如果你使用了SSD类的磁盘，或者将几块磁盘做了RAID，当你的存储拥有更高的IO速度时，完全可以将innodb_io_capacity的值调得再高点，直到符合你的磁盘IO的吞吐量为止。

另一个问题是参数innodb_max_dirty_pages_pct的默认值，在MySQL 5.1版本之前（包括5.1），该值的默认值为90，意味着脏页占缓冲池的90%。但是该值“太大”了，因为你会发现，InnoDB存储引擎在每1秒刷新缓冲池和flush loop时，会判断这个值，如果大于innodb_max_dirty_pages_pct，才刷新100个脏页。因此，如果你有很大的内存或你的数据库服务器的压力很大，这时刷新脏页的速度反而可能会降低。同样，在数据库的恢复阶段可能需要更多的时间。

在很多论坛上都有对这个问题的讨论，有人甚至将这个值调到了20或10，然后测试发现性能会有所提高，但是将innodb_max_dirty_pages_pct调到20或10会增加磁盘的压力，对系统的负担还是会有所增加。Google对这个问题进行了测试，可以证明20并不是一个最优值[⊖]。而从InnoDB Plugin开始，innodb_max_dirty_pages_pct默认值变为了75，和Google测试的80比较接近。这样既可以加快刷新脏页的频率，也能保证磁盘IO的负载。

⊖ 有兴趣的读者请参考：http://code.google.com/p/google-mysql-tools/wiki/InnodbIoOltpDisk。

InnoDB Plugin带来的另一个参数是innodb_adaptive_flushing（自适应地刷新），该值影响每1秒刷新脏页的数量。原来的刷新规则是：如果脏页在缓冲池所占的比例小于innodb_max_dirty_pages_pct时，不刷新脏页。大于innodb_max_dirty_pages_pct时，刷新100个脏页，而innodb_adaptive_flushing参数的引入，InnoDB存储引擎会通过一个名为buf_flush_get_desired_flush_rate的函数来判断需要刷新脏页最合适的数量。粗略地翻阅源代码后你会发现，buf_flush_get_desired_flush_rate是通过判断产生重做日志的速度来判断最合适的刷新脏页的数量。因此，当脏页的比例小于innodb_max_dirty_pages_pct时，也会刷新一定量的脏页。

通过上述的讨论和解释，从InnoDB Plugin开始，master thread的伪代码最终变成了：

```
void master_thread(){
    goto loop;
loop:
for(int i = 0; i<10; i++){
    thread_sleep(1) // sleep 1 second
    do log buffer flush to disk
    if ( last_one_second_ios < 5% innodb_io_capacity )
        do merge 5% innodb_io_capacity insert buffer
    if ( buf_get_modified_ratio_pct > innodb_max_dirty_pages_pct )
        do buffer pool flush 100% innodb_io_capacity dirty page
    else if enable adaptive flush
        do buffer pool flush desired amount dirty page
    if ( no user activity )
        goto backgroud loop
}
if ( last_ten_second_ios < innodb_io_capacity)
    do buffer pool flush 100% innodb_io_capacity dirty page
do merge at most 5% innodb_io_capacity insert buffer
do log buffer flush to disk
do full purge
if ( buf_get_modified_ratio_pct > 70% )
    do buffer pool flush 100% innodb_io_capacity dirty page
else
    do buffer pool flush 10% innodb_io_capacity dirty page
do fuzzy checkpoint
goto loop
background loop:
do full purge
do merge 100% innodb_io_capacity insert buffer
```

```
if not idle:
goto loop:
else:
    goto flush loop
flush loop:
do buffer pool flush 100% innodb_io_capacity dirty page
if ( buf_get_modified_ratio_pct> innodb_max_dirty_pages_pct )
    go to flush loop
    goto suspend loop
suspend loop:
suspend_thread()
waiting event
goto loop;
}
```

很多测试都显示，InnoDB Plugin较之以前的InnoDB存储引擎在性能方面有了极大的提高，其实这与以上master thread的改动是密不可分的，因为InnoDB存储引擎的核心操作大部分都是在master thread中。

2.4 关键特性

InnoDB存储引擎的关键特性包括插入缓冲、两次写（double write）、自适应哈希索引（adaptive hash index）。这些特性为InnoDB存储引擎带来了更好的性能和更高的可靠性。

2.4.1 插入缓冲

插入缓冲是InnoDB存储引擎关键特性中最令人激动的。不过，这个名字可能会让人认为插入缓冲是缓冲池中的一个部分。其实不然，InnoDB缓冲池中有Insert Buffer信息固然不错，但是Insert Buffer和数据页一样，也是物理页的一个组成部分。

我们知道，主键是行唯一的标识符，在应用程序中行记录的插入顺序是按照主键递增的顺序进行插入的。因此，插入聚集索引一般是顺序的，不需要磁盘的随机读取。比如说我们按下列SQL定义的表。

```
mysql> create table t ( id int auto_increment, name varchar(30),primary key (id));
Query OK, 0 rows affected (0.14 sec)
```

id列是自增长的，这意味着当执行插入操作时，id列会自动增长，页中的行记录按id执

行顺序存放。一般情况下，不需要随机读取另一页执行记录的存放。因此，在这样的情况下，插入操作一般很快就能完成。但是，不可能每张表上只有一个聚集索引，在更多的情况下，一张表上有多个非聚集的辅助索引（secondary index）。比如，我们还需要按照name这个字段进行查找，并且name这个字段不是唯一的。即，表是按如下的SQL语句定义的：

```
mysql> create table t ( id int auto_increment, name varchar(30),primary key
(id),key(name));
Query OK, 0 rows affected (0.21 sec)
```

这样的情况下产生了一个非聚集的并且不是唯一的索引。在进行插入操作时，数据页的存放还是按主键id的执行顺序存放，但是对于非聚集索引，叶子节点的插入不再是顺序的了。这时就需要离散地访问非聚集索引页，插入性能在这里变低了。然而这并不是这个name字段上索引的错误，因为B+树的特性决定了非聚集索引插入的离散性。

InnoDB存储引擎开创性地设计了插入缓冲，对于非聚集索引的插入或更新操作，不是每一次直接插入索引页中。而是先判断插入的非聚集索引页是否在缓冲池中。如果在，则直接插入；如果不在，则先放入一个插入缓冲区中，好似欺骗数据库这个非聚集的索引已经插到叶子节点了，然后再以一定的频率执行插入缓冲和非聚集索引页子节点的合并操作，这时通常能将多个插入合并到一个操作中（因为在一个索引页中），这就大大提高了对非聚集索引执行插入和修改操作的性能。

插入缓冲的使用需要满足以下两个条件：

- 索引是辅助索引。
- 索引不是唯一的。

当满足以上两个条件时，InnoDB存储引擎会使用插入缓冲，这样就能提高性能了。不过考虑一种情况，应用程序执行大量的插入和更新操作，这些操作都涉及了不唯一的非聚集索引，如果在这个过程中数据库发生了宕机，这时候会有大量的插入缓冲并没有合并到实际的非聚集索引中。如果是这样，恢复可能需要很长的时间，极端情况下甚至需要几个小时来执行合并恢复操作。

辅助索引不能是唯一的，因为在把它插入到插入缓冲时，我们并不去查找索引页的情况。如果去查找肯定又会出现离散读的情况，插入缓冲就失去了意义。

可以通过命令SHOW ENGINE INNODB STATUS来查看插入缓冲的信息：

```
mysql> show engine innodb status\G;
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
100727 22:21:48 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 44 seconds
......
-------------------------------------
INSERT BUFFER AND ADAPTIVE HASH INDEX
-------------------------------------
Ibuf: size 7545, free list len 3790, seg size 11336,
8075308 inserts, 7540969 merged recs, 2246304 merges
......
-----------------------------
END OF INNODB MONITOR OUTPUT
=============================

1 row in set (0.00 sec)
```

seg size显示了当前插入缓冲的大小为11 336*16KB，大约为177MB，free list len代表了空闲列表的长度，size代表了已经合并记录页的数量。下面一行可能是我们真正关心的，因为它显示了提高性能了。Inserts代表插入的记录数，merged recs代表合并的页的数量，merges代表合并的次数。merges：merged recs大约为3：1，代表插入缓冲将对于非聚集索引页的IO请求大约降低了3倍。

目前插入缓冲存在一个问题是，在写密集的情况下，插入缓冲会占用过多的缓冲池内存，默认情况下最大可以占用1/2的缓冲池内存。以下是InnoDB存储引擎源代码中对insert buffer的初始化操作：

```
/** Buffer pool size per the maximum insert buffer size */
#define IBUF_POOL_SIZE_PER_MAX_SIZE    2
ibuf->max_size = buf_pool_get_curr_size() / UNIV_PAGE_SIZE
            / IBUF_POOL_SIZE_PER_MAX_SIZE;
```

这对其他的操作可能会带来一定的影响。Percona已发布一些patch来修正插入缓冲占用太多缓冲池内存的问题，具体的可以到http://www.percona.com/percona-lab.html查找。

简单来说，修改IBUF_POOL_SIZE_PER_MAX_SIZE就可以对插入缓冲的大小进行控制，例如，将IBUF_POOL_SIZE_PER_MAX_SIZE改为3，则最大只能使用1/3的缓冲池内存。

2.4.2 两次写

如果说插入缓冲带给InnoDB存储引擎的是性能，那么两次写带给InnoDB存储引擎的是数据的可靠性。当数据库宕机时，可能发生数据库正在写一个页面，而这个页只写了一部分（比如16K的页，只写前4K的页）的情况，我们称之为部分写失效（partial page write）。在InnoDB存储引擎未使用double write技术前，曾出现过因为部分写失效而导致数据丢失的情况。

有人也许会想，如果发生写失效，可以通过重做日志进行恢复。这是一个办法。但是必须清楚的是，重做日志中记录的是对页的物理操作，如偏移量800，写'aaaa'记录。如果这个页本身已经损坏，再对其进行重做是没有意义的。这就是说，在应用（apply）重做日志前，我们需要一个页的副本，当写入失效发生时，先通过页的副本来还原该页，再进行重做，这就是doublewrite。InnoDB存储引擎doublewrite的体系架构如图2-4所示：

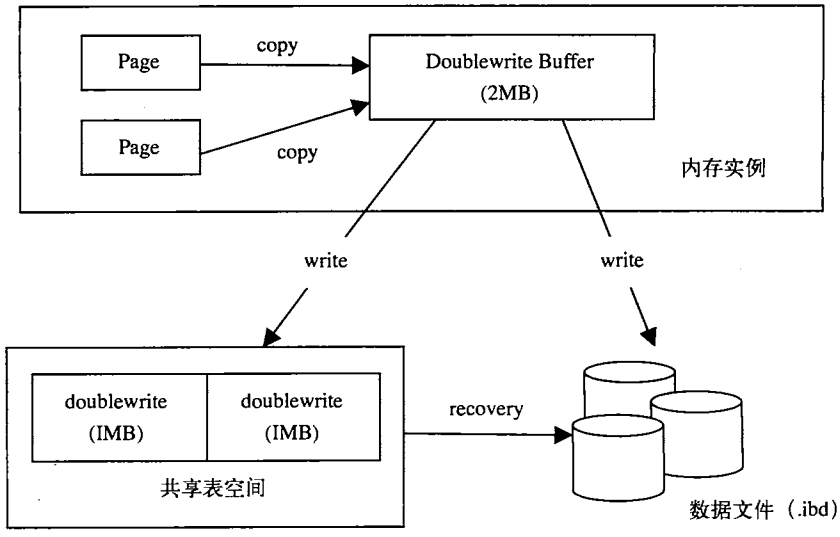


图2-4 InnoDB存储引擎doublewrite架构

doublewrite由两部分组成：一部分是内存中的doublewrite buffer，大小为2MB；另一部分是物理磁盘上共享表空间中连续的128个页，即两个区（extent），大小同样为2MB。

当缓冲池的脏页刷新时，并不直接写磁盘，而是会通过memcpy函数将脏页先拷贝到内存中的doublewrite buffer，之后通过doublewrite buffer再分两次，每次写入1MB到共享表空间的物理磁盘上，然后马上调用fsync函数，同步磁盘，避免缓冲写带来的问题。在这个过程中，因为doublewrite页是连续的，因此这个过程是顺序写的，开销并不是很大。在完成doublewrite页的写入后，再将doublewrite buffer中的页写入各个表空间文件中，此时的写入则是离散的。可以通过以下命令观察到doublewrite运行的情况：

```
mysql> show global status like 'innodb_dblwr%'\G;
*************************** 1. row ***************************
Variable_name: Innodb_dblwr_pages_written
        Value: 6325194
*************************** 2. row ***************************
Variable_name: Innodb_dblwr_writes
        Value: 100399
2 rows in set (0.00 sec)
```

可以看到，doublewrite一共写了6 325 194个页，但实际的写入次数为100 399，基本上符合64：1。如果发现你的系统在高峰时Innodb_dblwr_pages_written: Innodb_dblwr_writes远小于64：1，那么说明你的系统写入压力并不是很高。

如果操作系统在将页写入磁盘的过程中崩溃了，在恢复过程中，InnoDB存储引擎可以从共享表空间中的doublewrite中找到改页的一个副本，将其拷贝到表空间文件，再应用重做日志。下面显示了由doublewrite进行恢复的一种情况：

```
090924 11:36:32  mysqld restarted
090924 11:36:33  InnoDB: Database was not shut down normally!
InnoDB: Starting crash recovery.
InnoDB: Reading tablespace information from the .ibd files...
InnoDB: Error: space id in fsp header 0, but in the page header 4294967295
InnoDB: Error: tablespace id 4294967295 in file ./test/t.ibd is not sensible
InnoDB: Error: tablespace id 0 in file ./test/t2.ibd is not sensible
090924 11:36:33  InnoDB: Operating system error number 40 in a file operation.
InnoDB: Error number 40 means 'Too many levels of symbolic links'.
InnoDB: Some operating system error numbers are described at
InnoDB: http://dev.mysql.com/doc/refman/5.0/en/operating-system-error-codes.html
InnoDB: File name ./now/member
InnoDB: File operation call: 'stat'.
InnoDB: Error: os_file_readdir_next_file() returned -1 in
InnoDB: directory ./now
```

```
InnoDB: Crash recovery may have failed for some .ibd files!
InnoDB: Restoring possible half-written data pages from the doublewrite
InnoDB: buffer...
```

参数skip_innodb_doublewrite可以禁止使用两次写功能，这时可能会发生前面提及的写失效问题。不过，如果你有多台从服务器（slave server），需要提供较快的性能（如slave上做的是RAID0），也许启用这个参数是一个办法。不过，在需要提供数据高可靠性的主服务器（master server）上，任何时候我们都应确保开启两次写功能。

注意： 有些文件系统本身就提供了部分写失效的防范机制，如ZFS文件系统。在这种情况下，我们就不要启用doublewrite了。

2.4.3 自适应哈希索引

哈希（hash）是一种非常快的查找方法，一般情况下查找的时间复杂度为O（1）。常用于连接（join）操作，如SQL Server和Oracle中的哈希连接（hash join）。但是SQL Server和Oracle等常见的数据库并不支持哈希索引（hash index）。MySQL的Heap存储引擎默认的索引类型为哈希，而InnoDB存储引擎提出了另一种实现方法，自适应哈希索引（adaptive hash index）。

InnoDB存储引擎会监控对表上索引的查找，如果观察到建立哈希索引可以带来速度的提升，则建立哈希索引，所以称之为自适应（adaptive）的。自适应哈希索引通过缓冲池的B+树构造而来，因此建立的速度很快。而且不需要将整个表都建哈希索引，InnoDB存储引擎会自动根据访问的频率和模式来为某些页建立哈希索引。

根据InnoDB的官方文档显示，启用自适应哈希索引后，读取和写入速度可以提高2倍；对于辅助索引的连接操作，性能可以提高5倍。在我看来，自适应哈希索引是非常好的优化模式，其设计思想是数据库自优化（self-tuning），即无需DBA对数据库进行调整。

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，如下所示：

```
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了，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。值得注意的是，哈希索引只能用来搜索等值的查询，如select * from table where index_col = 'xxx'，而对于其他查找类型，如范围查找，是不能使用的。因此，这里出现了non-hash searches/s的情况。用hash searches : non-hash searches命令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存储引擎控制的，所以这里的信息只供我们参考。不过我们可以通过参数innodb_adaptive_hash_index来禁用或启动此特性，默认为开启。

2.5 启动、关闭与恢复

InnoDB存储引擎是MySQL的存储引擎之一，因此InnoDB存储引擎的启动和关闭更准确地是指在MySQL实例的启动过程中对InnoDB表存储引擎的处理过程。

在关闭时，参数innodb_fast_shutdown影响着表的存储引擎为InnoDB的行为。该参数可取值为0、1、2。0代表当MySQL关闭时，InnoDB需要完成所有的full purge和merge insert buffer操作，这会需要一些时间，有时甚至需要几个小时来完成。如果在做InnoDB plugin升级，通常需要将这个参数调为0，然后再关闭数据库。1是该参数的默认值，表示不需要完成上述的full purge和merge insert buffer操作，但是在缓冲池的一些数据脏页还是会刷新到磁盘。2表示不完成full purge和merge insert buffer操作，也不将缓冲池中的数据脏页写回磁盘，而是将日志都写入日志文件。这样不会有任何事务会丢失，但是MySQL数

据库下次启动时，会执行恢复操作（recovery）。

当正常关闭MySQL数据库时，下一次启动应该会很正常。但是，如果没有正常地关闭数据库，如用kill命令关闭数据库，在MySQL数据库运行过程中重启了服务器，或者在关闭数据库时将参数innodb_fast_shutdown设为了2，MySQL数据库下次启动时都会对InnoDB存储引擎的表执行恢复操作。

参数innodb_force_recovery影响了整个InnoDB存储引擎的恢复状况。该值默认为0，表示当需要恢复时执行所有的恢复操作。当不能进行有效恢复时，如数据页发生了corruption，MySQL数据库可能会宕机，并把错误写入错误日志中。

但是，在某些情况下，我们可能并不需要执行完整的恢复操作，我们自己知道如何进行恢复。比如正在对一个表执行alter table操作，这时意外发生了，数据库重启时会对InnoDB表执行回滚操作。对于一个大表，这需要很长时间，甚至可能是几个小时。这时我们可以自行进行恢复，例如可以把表删除，从备份中重新将数据导入表中，这些操作的速度可能要远远快于回滚操作。

innodb_force_recovery还可以设置为6个非零值：1～6。大的数字包含了前面所有小数字的影响，具体情况如下。

- 1（SRV_FORCE_IGNORE_CORRUPT）：忽略检查到的corrupt页。
- 2（SRV_FORCE_NO_BACKGROUND）：阻止主线程的运行，如主线程需要执行full purge操作，会导致crash。
- 3（SRV_FORCE_NO_TRX_UNDO）：不执行事务回滚操作。
- 4（SRV_FORCE_NO_IBUF_MERGE）：不执行插入缓冲的合并操作。
- 5（SRV_FORCE_NO_UNDO_LOG_SCAN）：不查看撤销日志（Undo Log），InnoDB存储引擎会将未提交的事务视为已提交。
- 6（SRV_FORCE_NO_LOG_REDO）：不执行前滚的操作。

需要注意的是，当设置参数innodb_force_recovery大于0后，可以对表进行select、create、drop操作，但insert、update或者delete这类操作是不允许的。

我们来做个实验，模拟故障的发生。在第一会话中，对一张接近1 000W行的InnoDB存储引擎表执行更新操作，但是完成后不要马上提交：

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> update Profile set password='';
Query OK, 9587770 rows affected (7 min 55.73 sec)
Rows matched: 9999248  Changed: 9587770  Warnings: 0
```

start transaction语句开启了事务，同时防止了自动提交的发生，update操作则会产生大量的回滚日志。这时，我们人为地kill掉MySQL数据库服务器。

```
[root@nineyou0-43 ~]# ps -ef | grep mysqld
root     28007     1  0 13:40 pts/1    00:00:00 /bin/sh ./bin/mysqld_safe
--datadir=/usr/local/mysql/data --pid-file=/usr/local/mysql/data/nineyou0-43.pid
mysql    28045 28007 42 13:40 pts/1    00:04:23
/usr/local/mysql/bin/mysqld --basedir=/usr/local/mysql
--datadir=/usr/local/mysql/data --user=mysql
--pid-file=/usr/local/mysql/data/nineyou0-43.pid
--skip-external-locking --port=3306 --socket=/tmp/mysql.sock
root     28110 26963  0 13:50 pts/11   00:00:00 grep mysqld
[root@nineyou0-43 ~]# kill -9 28007
[root@nineyou0-43 ~]# kill -9 28045
```

通过kill命令，我们人为地模拟了一次数据库宕机故障，当MySQL数据库下次启动时会对update的这个事务执行回滚操作，而这些信息都会记录在错误日志文件中，默认后缀名为err。如果查看错误日志文件，可得到如下结果：

```
090922 13:40:20  InnoDB: Started; log sequence number 6 2530474615
InnoDB: Starting in background the rollback of uncommitted transactions
090922 13:40:20  InnoDB: Rolling back trx with id 0 5281035, 8867280 rows to undo

InnoDB: Progress in percents: 1090922 13:40:20
090922 13:40:20 [Note] /usr/local/mysql/bin/mysqld: ready for connections.
Version: '5.0.45-log'  socket: '/tmp/mysql.sock'  port: 3306  MySQL Community
Server (GPL)
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57
58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84
85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
InnoDB: Rolling back of trx id 0 5281035 completed
090922 13:49:21  InnoDB: Rollback of non-prepared transactions completed
```

可以看到，如果采用默认的策略，即把innodb_force_recovery设为0，InnoDB会在每次

启动后对发生问题的表执行恢复操作，通过错误日志文件，可知这次回滚操作需要回滚8 867 280行记录，总共耗时约9分多钟。

我们做再做一次同样的测试，只不过在启动MySQL数据库前将参数innodb_force_recovery设为3，然后观察InnoDB存储引擎是否还会执行回滚操作，查看错误日志文件，可看到：

```
090922 14:26:23  InnoDB: Started; log sequence number 7 2253251193
InnoDB: !!! innodb_force_recovery is set to 3 !!!
090922 14:26:23 [Note] /usr/local/mysql/bin/mysqld: ready for connections.
Version: '5.0.45-log'  socket: '/tmp/mysql.sock'  port: 3306  MySQL Community
Server (GPL)
```

这里出现了“！！！”，InnoDB警告你已经将innodb_force_recovery设置为3，不会进行undo的回滚操作了。因此数据库很快启动完成，但是你应该很小心当前数据库的状态，并仔细确认是否不需要回滚操作。

2.6 InnoDB Plugin = 新版本的InnoDB存储引擎

MySQL 5.1这个版本的一个很大的变动是采用了插件式的架构。通过分析源代码会发现，每个存储引擎是通过继承一个handler的C++基类（之前的版本大多是通过函数指针来实现）。如果查看InnoDB存储引擎的源代码，会看到下面的内容：

```
/* The class defining a handle to an Innodb table */
class ha_innobase: public handler
{
row_prebuilt_t*prebuilt;/* prebuilt struct in InnoDB, used
to save CPU time with prebuilt data
structures*/
THD*user_thd;/* the thread handle of the user
currently using the handle; this is
set in external_lock function */
THR_LOCK_DATAlock;
INNOBASE_SHARE*share;

uchar*upd_buff;/* buffer used in updates */
uchar*key_val_buff;/* buffer used in converting
search key values from MySQL format
to Innodb format */
```

```
ulongupd_and_key_val_buff_len;
/* the length of each of the previous
two buffers */
......
```

这样设计的好处是，现在所有的存储引擎都是真正的插件式了。在以前，如果发现一个InnoDB存储引擎的Bug，你能做的就是等待MySQL新版本的发布，InnoDB公司本身对此只能通过补丁的形式来解决，你还需要重新编译一次MySQL才行。现在，你可以得到一个新版本的InnoDB存储引擎，用来替代有Bug的旧引擎。这样，当有重大问题时，不用等待MySQL的新版本，只要相应的存储引擎提供商发布新版本即可。

当然，InnoDB Plugin是上述的一个实现，但更重要的是InnoDB Plugin还给我们带来了其他非常棒的新特性，你应该把它看做是新版本的InnoDB存储引擎，我们通过参数innodb_version来查看当前InnoDB的版本：

```
mysql> show variables like 'innodb_version'\G;
*************************** 1. row ***************************
Variable_name: innodb_version
        Value: 1.0.4
1 row in set (0.00 sec)
```

在MySQL 5.1.38前的版本中，当你需要安装InnoDB Plugin时，必须下载Plugin的文件，解压后再进行一系列的安装。从MySQL 5.1.38开始，MySQL包含了2个不同版本的InnoDB存储引擎——一个是旧版本的引擎，称之为build-in innodb；另一个是1.0.4版本的InnoDB存储引擎。如果你想使用新的InnDB Plugin引擎，只需在配置文件做如下设置：

```
[mysqld]
ignore-builtin-innodb
plugin-load=innodb=ha_innodb_plugin.so
  ;innodb_trx=ha_innodb_plugin.so
  ;innodb_locks=ha_innodb_plugin.so
  ;innodb_cmp=ha_innodb_plugin.so
  ;innodb_cmp_reset=ha_innodb_plugin.so
  ;innodb_cmpmem=ha_innodb_plugin.so
  ;innodb_cmpmem_reset=ha_innodb_plugin.so
```

在写本书时，InnoDB Plugin的最新版本是1.0.4，并已经是MySQL 5.4版本默认的InnoDB存储引擎版本，其新功能包括以下几点。

- 快速索引重建。
- 更好的多核性能。
- 新的页结构。
- 页压缩功能。
- 更好的BLOB处理能力。

感兴趣的读者可以在官网http://www.innodb.com/products/innodb_plugin/上获得更多的信息。本书在之后的章节中会对新版本InnoDB存储引擎进行详细讲解。

2.7 小结

本章首先介绍了InnoDB存储引擎的历史、InnoDB存储引擎的体系结构（包括后台线程和内存结构），之后又详细讲解了InnoDB存储引擎的关键特性，这些特性使得InnoDB存储引擎变得更具“魅力”，最后叙述了启动和关闭MySQL时一些配置文件参数对InnoDB存储引擎的影响。

通过本章的铺垫，后文就能更深入和全面地讲解InnoDB引擎。第3章开始介绍MySQL的文件，包括MySQL本身的文件和与InnoDB存储引擎有关的文件，之后的章节将介绍基于InnoDB存储引擎的表，并揭示其内部的存储构造。

第3章 文　　件

本章将分析构成MySQL数据库和InnoDB存储引擎表的各种类型文件，如下所示。

- 参数文件：告诉MySQL实例启动时在哪里可以找到数据库文件，并且指定某些初始化参数，这些参数定义了某种内存结构的大小等设置，还会介绍各种参数的类型。
- 日志文件：用来记录MySQL实例对某种条件做出响应时写入的文件。如错误日志文件、二进制日志文件、满查询日志文件、查询日志文件等。
- socket文件：当用Unix域套接字方式进行连接时需要的文件。
- pid文件：MySQL实例的进程ID文件。
- MySQL表结构文件：用来存放MySQL表结构定义文件。
- 存储引擎文件：因为MySQL表存储引擎的关系，每个存储引擎都会有自己的文件来保存各种数据。这些存储引擎真正存储了数据和索引等数据。本章主要介绍与InnoDB有关的存储引擎文件。

3.1 参数文件

在第1章中已经介绍过了，当MySQL实例启动时，MySQL会先去读一个配置参数文件，用来寻找数据库的各种文件所在位置以及指定某些初始化参数，这些参数通常定义了某种内存结构有多大等设置。默认情况下，MySQL实例会按照一定的次序去取，你只需通过命令mysql --help | grep my.cnf来寻找即可。

MySQL参数文件的作用和Oracle的参数文件极其类似；不同的是，Oracle实例启动时若找不到参数文件，是不能进行装载（mount）操作的。MySQL稍微有所不同，MySQL实例可以不需要参数文件，这时所有的参数值取决于编译MySQL时指定的默认值和源代码中指定参数的默认值。但是，如果MySQL在默认的数据库目录下找不到mysql架构，则启动同样会失败，你可能在错误日志文件中找到如下内容：

```
090922 16:25:52  mysqld started
090922 16:25:53  InnoDB: Started; log sequence number 8 2801063211
InnoDB: !!! innodb_force_recovery is set to 1 !!!
090922 16:25:53 [ERROR] Fatal error: Can't open and lock privilege tables: Table
'mysql.host' doesn't exist
090922 16:25:53  mysqld ended
```

MySQL中的mysql架构中记录了访问该实例的权限，当找不到这个架构时，MySQL实例不会成功启动。

和Oracle参数文件不同的是，Oracle的参数文件分为二进制的参数文件（spfile）和文本类型的参数文件（init.ora），而MySQL的参数文件仅是文本的，方便的是，你可以通过一些常用的编辑软件（如vi和emacs）进行参数的编辑。

3.1.1 什么是参数

简单地说，可以把数据库参数看成一个键/值对。第2章已经介绍了一个对于InnoDB存储引擎很重要的参数innodb_buffer_pool_size。如我们将这个参数设置为1G，即innodb_buffer_pool_size=1G,，这里的“键”是innodb_buffer_pool_size，“值”是1G，这就是我们的键值对。可以通过show variables查看所有的参数，或通过like来过滤参数名。从MySQL 5.1版本开始，可以通过information_schema架构下的GLOBAL_VARIABLES视图来进行查找，如下所示。

```
mysql> select * from GLOBAL_VARIABLES where VARIABLE_NAME like 'innodb_buffer%'\G;
*************************** 1. row ***************************
VARIABLE_NAME: INNODB_BUFFER_POOL_SIZE
VARIABLE_VALUE: 1073741824
1 row in set (0.00 sec)

mysql> show variables like 'innodb_buffer%'\G;
*************************** 1. row ***************************
Variable_name: innodb_buffer_pool_size
        Value: 1073741824
1 row in set (0.00 sec)
```

无论使用哪种方法，输出的信息基本上都一样的，只不过通过视图GLOBAL_VARIABLES需要指定视图的列名。推荐使用show variables命令，因为这个命令使用更为简单，各版本的MySQL数据库都支持它。

Oracle的参数有所谓的隐藏参数（undocumented parameter），以供Oracle“内部人士”使用，SQL Server也有类似的参数。有些DBA曾问我，MySQL中是否也有这类参数。我的回答是：没有，也不需要。即使Oracle和SQL Server中都有些所谓的隐藏参数，在绝大多数情况下，这些数据库厂商也不建议你在生产环境中对其进行很大的调整。

3.1.2 参数类型

MySQL参数文件中的参数可以分为两类：动态（dynamic）参数和静态（static）参数。动态参数意味着你可以在MySQL实例运行中进行更改；静态参数说明在整个实例生命周期内都不得进行更改，就好像是只读（read only）的。可以通过SET命令对动态的参数值进行修改，SET的语法如下：

```
SET
| [global | session] system_var_name= expr
| [@@global. | @@session. | @@]system_var_name= expr
```

这里可以看到global和session关键字，它们表明该参数的修改是基于当前会话还是整个实例的生命周期。有些动态参数只能在会话中进行修改，如autocommit；有些参数修改完后，在整个实例生命周期中都会生效，如binlog_cache_size；而有些参数既可以在会话又可以在整个实例的生命周期内生效，如read_buffer_size。举例如下：

```
mysql> set read_buffer_size=524288;
Query OK, 0 rows affected (0.00 sec)

mysql> select @@session.read_buffer_size\G;
*************************** 1. row ***************************
@@session.read_buffer_size: 524288
1 row in set (0.00 sec)

mysql> select @@global.read_buffer_size\G;
*************************** 1. row ***************************
@@global.read_buffer_size: 2093056
1 row in set (0.00 sec)
```

上面我将read_buffer_size的会话值从2MB调整为了512KB，你可以看到全局的read_buffer_size的值仍然是2MB，也就是说，如果有另一个会话登录到MySQL实例，它的read_buffer_size的值是2MB，而不是512KB。这里使用了set globallsession来改变动态变量

的值。我们同样可以直接使用set @@globll@@session来更改，如下所示：

```
mysql> set @@global.read_buffer_size=1048576;
Query OK, 0 rows affected (0.00 sec)

mysql> select @@session.read_buffer_size\G;
*************************** 1. row ***************************
@@session.read_buffer_size: 524288
1 row in set (0.00 sec)

mysql> select @@global.read_buffer_size\G;
*************************** 1. row ***************************
@@global.read_buffer_size: 1048576
1 row in set (0.00 sec)
```

这次我们把read_buffer_size全局值更改为1MB，而当前会话的read_buffer_size的值还是512KB。这里需要注意的是，对变量的全局值进行了修改，在这次的实例生命周期内都有效，但MySQL实例本身并不会对参数文件中的该值进行修改。也就是说下次启动时，MySQL实例还是会读取参数文件。如果你想让数据库实例下一次启动时该参数还是保留为当前修改的值，则必须修改参数文件。要想知道MySQL所有动态变量的可修改范围，可以参考MySQL官方手册的第5.1.4.2节（Dynamic System Variables）的相关内容。

对于静态变量，如果对其进行修改，会得到类似如下的错误：

```
mysql> set global datadir='/db/mysql';
ERROR 1238 (HY000): Variable 'datadir' is a read only variable
```

3.2 日志文件

日志文件记录了影响MySQL数据库的各种类型活动。MySQL数据库中常见的日志文件有错误日志、二进制日志、慢查询日志、查询日志。这些日志文件为DBA对数据库优化、问题查找等带来了极大的便利。

3.2.1 错误日志

错误日志文件对MySQL的启动、运行、关闭过程进行了记录。MySQL DBA在遇到问题时应该首先查看该文件。该文件不但记录了出错信息，也记录一些警告信息或者正确的

信息。总的来说，这个文件更类似于Oracle的alert文件，只不过在默认情况下是err结尾。你可以通过show variables like 'log_error'来定位该文件，如：

```
mysql> show variables like 'log_error';
+---------------+---------------------------------------------+
| Variable_name | Value                                       |
+---------------+---------------------------------------------+
| log_error     | /usr/local/mysql/data/stargazer.err |
+---------------+---------------------------------------------+
1 row in set (0.00 sec)

mysql> system hostname
stargazer
```

可以看到错误文件的路径和文件名，默认情况下错误文件的文件名为服务器的主机名。如上面我们看到的，该主机名为stargazer，所以错误文件名为startgazer.err。当出现MySQL数据库不能正常启动时，第一个必须查找的文件应该就是错误日志文件，该文件记录了出错信息，能很好地指导我们找到问题，如果当数据库不能重启，通过查错误日志文件可以得到如下内容：

```
[root@nineyou0-43 data]# tail -n 50 nineyou0-43.err
090924 11:31:18  mysqld started
090924 11:31:18  InnoDB: Started; log sequence number 8 2801063331
090924 11:31:19 [ERROR] Fatal error: Can't open and lock privilege tables:
Table 'mysql.host' doesn't exist
090924 11:31:19  mysqld ended
```

这里，错误日志文件提示了你找不到权限库mysql，所以启动失败。有时我们可以直接在错误日志文件里得到优化的帮助，因为有些警告（warning）很好地说明了问题所在。而这时我们可以不需要通过查看数据库状态来得知，如：

```
090924 11:39:44  InnoDB: ERROR: the age of the last checkpoint is 9433712,
InnoDB: which exceeds the log group capacity 9433498.
InnoDB: If you are using big BLOB or TEXT rows, you must set the
InnoDB: combined size of log files at least 10 times bigger than the
InnoDB: largest such row.
090924 11:40:00  InnoDB: ERROR: the age of the last checkpoint is 9433823,
InnoDB: which exceeds the log group capacity 9433498.
InnoDB: If you are using big BLOB or TEXT rows, you must set the
InnoDB: combined size of log files at least 10 times bigger than the
InnoDB: largest such row.
```

```
090924 11:40:16  InnoDB: ERROR: the age of the last checkpoint is 9433645,
InnoDB: which exceeds the log group capacity 9433498.
InnoDB: If you are using big BLOB or TEXT rows, you must set the
InnoDB: combined size of log files at least 10 times bigger than the
InnoDB: largest such row.
```

3.2.2 慢查询日志

前一小节提到可以通过错误日志得到一些关于数据库优化的信息帮助，而慢查询能为SQL语句的优化带来很好的帮助。可以设一个阈值，将运行时间超过该值的所有SQL语句都记录到慢查询日志文件中。该阈值可以通过参数long_query_time来设置，默认值为10，代表10秒。

默认情况下，MySQL数据库并不启动慢查询日志，你需要手工将这个参数设为ON，然后启动，可以看到如下结果：

```
mysql> show variables like '%long%';
+-----------------+------------+
| Variable_name   | Value |
+-----------------+------------+
| long_query_time | 10    |
+-----------------+------------+
1 row in set (0.00 sec)

mysql> show variables like 'log_slow_queries';
+-----------------+------------+
| Variable_name | Value |
+-----------------+------------+
| log_slow_queries | ON|
+-----------------+------------+
1 row in set (0.01 sec)
```

这里需要注意两点。首先，设置long_query_time这个阈值后，MySQL数据库会记录运行时间超过该值的所有SQL语句，但对于运行时间正好等于long_query_time的情况，并不会被记录下。也就是说，在源代码里是判断大于long_query_time，而非大于等于。其次，从MySQL 5.1开始，long_query_time开始以微秒记录SQL语句运行时间，之前仅用秒为单位记录。这样可以更精确地记录SQL的运行时间，供DBA分析。对DBA来说，一条SQL语句运行0.5秒和0.05秒是非常不同的，前者可能已经进行了表扫，后面可能是走了索引。下

面的代码中，是在MySQL 5.1中将long_query_time设置为了0.05：

```
mysql> show variables like 'long_query_time';
+-----------------+--------------+
| Variable_name   | Value|
+-----------------+--------------+
| long_query_time | 0.050000 |
+-----------------+--------------+
1 row in set (0.00 sec)
```

另一个和慢查询日志有关的参数是log_queries_not_using_indexes，如果运行的SQL语句没有使用索引，则MySQL数据库同样会将这条SQL语句记录到慢查询日志文件。首先，确认打开了log_queries_not_using_indexes:

```
mysql> show variables like 'log_queries_not_using_indexes';
+-------------------------------+------------------+
| Variable_name          | Value|
+-------------------------------+------------------+
| log_queries_not_using_indexes | ON   |
+-------------------------------+------------------+
1 row in set (0.00 sec)
update 'low_game_schema'.'item' set SLOT='8' where GUID='2222249168632297608'
and is_destroy='0';
```

这里详细记录了SQL语句的信息，如上述SQL语句运行的账户和IP、运行时间、锁定的时间、返回行等。我们可以通过慢查询日志来找出有问题的SQL语句，对其进行优化。随着MySQL数据库服务器运行时间的增加，可能会有越来越多的SQL查询被记录到了慢查询日志文件中，这时要分析该文件就显得不是很容易了。MySQL这时提供的mysqldumpslow命令，可以很好地解决这个问题：

```
[root@nh122-190 data]# mysqldumpslow nh122-190-slow.log
Reading mysql slow query log from nh122-190-slow.log
Count:  11    Time=10.00s  (110s)    Lock=0.00s  (0s)    Rows=0.0  (0),
dbother[dbother]@localhost
   insert into test.DbStatus select now(),(N-com_select)/(N-uptime),(N-
com_insert)/(N-uptime),(N-com_update)/(N-uptime),(N-com_delete)/(N-uptime),N-
(N/N),N-(N/N),N.N/N,N-N/(N*N),GetCPULoadInfo(N) from test.CheckDbStatus order by
check_id desc limit N
Count: 653 Time=0.00s (0s)  Lock=0.00s (0s)  Rows=0.0 (0), 9YOUgs_SC[9YOUgs_SC]@
[192.168.43.7]
  select custom_name_one from 'low_game_schema'.'role_details' where role_id='S'
rse and summarize the MySQL slow query log. Options are
```

```
  --verbose    verbose
  --debug      debug
  --help       write this text to standard output

  -v           verbose
  -d           debug
  -s ORDER     what to sort by (al, at, ar, c, l, r, t), 'at' is default
                al: average lock time
                ar: average rows sent
                at: average query time
                 c: count
                 l: lock time
                 r: rows sent
                 t: query time
  -r           reverse the sort order (largest last instead of first)
  -t NUM       just show the top n queries
  -a           don't abstract all numbers to N and strings to 'S'
  -n NUM       abstract numbers with at least n digits within names
  -g PATTERN   grep: only consider stmts that include this string
  -h HOSTNAME  hostname of db server for *-slow.log filename (can be wildcard),
               default is '*', i.e. match all
  -i NAME      name of server instance (if using mysql.server startup script)
  -l           don't subtract lock time from total time
```

如果我们想得到锁定时间最长的10条SQL语句，可以运行：

```
[root@nh119-141 data]# /usr/local/mysql/bin/mysqldumpslow -s al -n 10 david.log
Reading mysql slow query log from david.log
Count: 5  Time=0.00s (0s)  Lock=0.20s (1s)  Rows=4.4 (22), Audition
[Audition]@[192.168.30.108]
  SELECT OtherSN, State FROM wait_friend_info WHERE UserSN = N

Count: 1  Time=0.00s (0s)  Lock=0.00s (0s)  Rows=1.0 (1), audition-kr[audition-
kr]@[192.168.30.105]
  SELECT COUNT(N) FROM famverifycode WHERE UserSN=N AND verifycode='S'
......
```

MySQL 5.1开始可以将慢查询的日志记录放入一张表中，这使我们的查询更加直观。慢查询表在mysql架构下，名为slow_log。其表结构定义如下：

```
mysql> show create table mysql.slow_log;
*************************** 1. row ***************************
       Table: slow_log
Create Table: CREATE TABLE 'slow_log' (
```

```
    'start_time' timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    'user_host' mediumtext NOT NULL,
    'query_time' time NOT NULL,
    'lock_time' time NOT NULL,
    'rows_sent' int(11) NOT NULL,
    'rows_examined' int(11) NOT NULL,
    'db' varchar(512) NOT NULL,
    'last_insert_id' int(11) NOT NULL,
    'insert_id' int(11) NOT NULL,
    'server_id' int(11) NOT NULL,
    'sql_text' mediumtext NOT NULL
) ENGINE=CSV DEFAULT CHARSET=utf8 COMMENT='Slow log'
1 row in set (0.00 sec)
```

参数log_output指定了慢查询输出的格式，默认为FILE，你可以将它设为TABLE，然后就可以查询mysql架构下的slow_log表了，如：

```
mysql> show variables like 'log_output';
+---------------+---------+
| Variable_name | Value |
+---------------+---------+
| log_output    | FILE |
+---------------+---------+
1 row in set (0.00 sec)

mysql> set global log_output='TABLE';
Query OK, 0 rows affected (0.00 sec)

mysql> show variables like 'log_output';
+---------------+---------+
| Variable_name | Value |
+---------------+---------+
| log_output  | TABLE |
+---------------+---------+
1 row in set (0.00 sec)

mysql> select sleep(10);
+-----------+
| sleep(10)|
+-----------+
|       0 |
+-----------+
1 row in set (10.01 sec)
```

```
mysql> select * from mysql.slow_log\G;
*************************** 1. row ***************************
    start_time: 2009-09-25 13:44:29
     user_host: david[david] @ localhost []
    query_time: 00:00:09
     lock_time: 00:00:00
     rows_sent: 1
 rows_examined: 0
            db: mysql
last_insert_id: 0
     insert_id: 0
     server_id: 0
      sql_text: select sleep(10)
1 row in set (0.00 sec)
```

参数log_output是动态的，并且是全局的。我们可以在线进行修改。在上表中我设置了睡眠（sleep）10秒，那么这句SQL语句就会被记录到slow_log表了。

查看slow_log表的定义会发现，该表使用的是CSV引擎，对大数据量下的查询效率可能不高。我们可以把slow_log表的引擎转换到MyISAM，用来进一步提高查询的效率。但是，如果已经启动了慢查询，将会提示错误：

```
mysql> alter table mysql.slow_log engine=myisam;
ERROR 1580 (HY000): You cannot 'ALTER' a log table if logging is enabled

mysql> set global slow_query_log=off;
Query OK, 0 rows affected (0.00 sec)

mysql> alter table mysql.slow_log engine=myisam;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

不能忽视的是，将slow_log表的存储引擎更改为MyISAM后，对数据库还是会造成额外的开销。不过好在很多关于慢查询的参数都是动态的，我们可以方便地在线进行设置或者修改。

3.2.3 查询日志

查询日志记录了所有对MySQL数据库请求的信息，不论这些请求是否得到了正确的执行。默认文件名为：主机名.log。我们查看一个查询日志：

```
[root@nineyou0-43 data]# tail nineyou0-43.log
090925 11:00:24   44 Connect     zlm@192.168.0.100 on
44 Query       SET AUTOCOMMIT=0
               44 Query       set autocommit=0
               44 Quit
090925 11:02:37 45 Connect   Access denied for user 'root'@'localhost' (using
password: NO)
090925 11:03:51 46 Connect   Access denied for user 'root'@'localhost' (using
password: NO)
090925 11:04:38  23 Query    rollback
```

通过上述查询日志你会发现，查询日志甚至记录了对access denied的请求。同样，从MySQL 5.1开始，可以将查询日志的记录放入mysql架构下的general_log表，该表的使用方法和前面小节提到的slow_log基本一样，这里不再赘述。

3.2.4 二进制日志

二进制日志记录了对数据库执行更改的所有操作，但是不包括SELECT和SHOW这类操作，因为这类操作对数据本身并没有修改，如果你还想记录SELECT和SHOW操作，那只能使用查询日志，而不是二进制日志了。此外，二进制还包括了执行数据库更改操作的时间和执行时间等信息。二进制日志主要有以下两种作用：

- 恢复（recovery）。某些数据的恢复需要二进制日志，如当一个数据库全备文件恢复后，我们可以通过二进制日志进行point-in-time的恢复。
- 复制（replication）。其原理与恢复类似，通过复制和执行二进制日志使得一台远程的MySQL数据库（一般称为slave或者standby）与一台MySQL数据库（一般称为master或者primary）进行实时同步。

通过配置参数log-bin[=name]可以启动二进制日志。如果不指定name，则默认二进制日志文件名为主机名，后缀名为二进制日志的序列号，所在路径为数据库所在目录(datadir)，如：

```
mysql> show variables like 'datadir';
+---------------+-----------------------------+
| Variable_name | Value      |
+---------------+-----------------------------+
| datadir       | /usr/local/mysql/data/ |
```

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

mysql> system ls -lh /usr/local/mysql/data/;
total 2.1G
-rw-rw----  1 mysql mysql 6.5M Sep 25 15:13 bin_log.000001
-rw-rw----  1 mysql mysql   17 Sep 25 00:32 bin_log.index
-rw-rw----  1 mysql mysql 300M Sep 25 15:13 ibdata1
-rw-rw----  1 mysql mysql 256M Sep 25 15:13 ib_logfile0
-rw-rw----  1 mysql mysql 256M Sep 25 15:13 ib_logfile1
drwxr-xr-x  2 mysql mysql 4.0K May  7 10:08 mysql
drwx------  2 mysql mysql 4.0K May  7 10:09 test
```

这里的bin_log.00001即为二进制日志文件，我们在配置文件中指定了名称，所以没有用默认的文件名。bin_log.index为二进制的索引文件，用来存储过往生产的二进制日志序号，通常情况下，不建议手工修改这个文件。

二进制日志文件在默认情况下并没有启动，需要你手动指定参数来启动。可能有人会质疑，开启这个选项是否会对数据库整体性能有所影响。不错，开启这个选项的确会影响性能，但是性能的损失十分有限。根据MySQL官方手册中的测试表明，开启二进制日志会使得性能下降1%。但考虑到可以使用复制（replication）和point-in-time的恢复，这些性能损失绝对是可以并且是应该被接受的。

以下配置文件的参数影响着二进制日志记录的信息和行为：

- max_binlog_size
- binlog_cache_size
- sync_binlog
- binlog-do-db
- binlog-ingore-db
- log-slave-update
- binlog_format

参数max-binlog-size指定了单个二进制日志文件的最大值，如果超过该值，则产生新的二进制日志文件，后缀名+1，并记录到.index文件。从MySQL 5.0开始的默认值为1 073 741 824，代表1GB（之前的版本max-binlog-size默认大小为1.1GB）。

当使用事务的表存储引擎（如InnoDB存储引擎）时，所有未提交（uncommitted）的二进制日志会被记录到一个缓存中，等该事务提交时（committed）时直接将缓冲中的二进制日志写入二进制日志文件，而该缓冲的大小由binlog_cache_size决定，默认大小为32KB。此外，binlog_cache_size是基于会话（session）的，也就是说，当一个线程开始一个事务时，MySQL会自动分配一个大小为binlog_cache_size的缓存，因此该值的设置需要相当小心，不能设置过大。当一个事务的记录大于设定的binlog_cache_size时，MySQL会把缓冲中的日志写入一个临时文件中，因此该值又不能设得太小。通过SHOW GLOBAL STATUS命令查看binlog_cache_use、binlog_cache_disk_use的状态，可以判断当前binlog_cache_size的设置是否合适。Binlog_cache_use记录了使用缓冲写二进制日志的次数，binlog_cache_disk_use记录了使用临时文件写二进制日志的次数。现在来看一个数据库的状态：

```
mysql> show variables like 'binlog_cache_size';
+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| binlog_cache_size | 32768 |
+-------------------+-------+
1 row in set (0.00 sec)

mysql> show global status like 'binlog_cache%';
+-----------------------+---------------+
| Variable_name         | Value         |
+-----------------------+---------------+
| binlog_cache_disk_use | 0             |
| binlog_cache_use      | 33553         |
+-----------------------+---------------+
2 rows in set (0.00 sec)
```

使用缓冲次数33 553次，临时文件使用次数为0。看来，32KB的缓冲大小对于当前这个MySQL数据库完全够用，所以暂时没有必要增加binlog_cache_size的值。

默认情况下，二进制日志并不是在每次写的时候同步到磁盘（我们可以理解为缓冲写）。因此，当数据库所在操作系统发生宕机时，可能会有最后一部分数据没有写入二进制日志文件中。这会给恢复和复制带来问题。参数sync_binlog=[N]表示每写缓冲多少次就同步到磁盘。如果将N设为1，即sync_binlog=1表示采用同步写磁盘的方式来写二进制日志，这

时写操作不使用操作系统的缓冲来写二进制日志。该默认值为0，如果使用InnoDB存储引擎进行复制，并且想得到最大的高可用性，建议将该值设为ON。不过该值为ON时，确实会对数据库的IO系统带来一定的影响。

但是，即使将sync_binlog设为1，还是会有一种情况会导致问题的发生。当使用InnoDB存储引擎时，在一个事务发出COMMIT动作之前，由于sync_binlog设为1，因此会将二进制日志立即写入磁盘。如果这时已经写入了二进制日志，但是提交还没有发生，并且此时发生了宕机，那么在MySQL数据库下次启动时，因为COMMIT操作并没有发生，所以这个事务会被回滚掉。但是二进制日志已经记录了该事务信息，不能被回滚。这个问题可以通过将参数innodb_support_xa设为1来解决，虽然innodb_support_xa与XA事务有关，但它同时也确保了二进制日志和InnoDB存储引擎数据文件的同步。

参数binlog-do-db和binlog-ignore-db表示需要写入或者忽略写入哪些库的日志。默认为空，表示需要将所有库的日志同步到二进制日志。

如果当前数据库是复制中的slave角色，则它不会将从master取得并执行的二进制日志写入自己的二进制日志文件中。如果需要写入，则需要设置log-slave-update。如果你需要搭建master=>slave=>slave架构的复制，则必须设置该参数。

binlog_format参数十分重要，这影响了记录二进制日志的格式。在MySQL 5.1版本之前，没有这个参数。所有二进制文件的格式都是基于SQL语句（statement）级别的，因此基于这个格式的二进制日志文件的复制（Replication）和Oracle 逻辑Standby有点相似。同时，对于复制是有一定要求的如rand、uuid等函数，或者有使用触发器等可能会导致主从服务器上表的数据不一致（not sync），这可能使得复制变得没有意义。另一个影响是，你会发现InnoDB存储引擎的默认事务隔离级别是REPEATABLE READ。这其实也是因为二进制日志文件格式的关系，如果使用READ COMMITTED的事务隔离级别（大多数数据库，如Oracle、Microsoft SQL Server数据库的默认隔离级别）会出现类似丢失更新的现象，从而出现主从数据库上的数据不一致。

MySQL 5.1开始引入了binlog_format参数，该参数可设的值有STATEMENT、ROW和MIXED。

（1）STATEMENT格式和之前的MySQL版本一样，二进制日志文件记录的是日志的逻

辑SQL语句。

（2）在ROW格式下，二进制日志记录的不再是简单的SQL语句了，而是记录表的行更改情况。基于ROW格式的复制类似于Oracle的物理Standby（当然，还是有些区别）。同时，对于上述提及的Statement格式下复制的问题给予了解决。MySQL 5.1版本开始，如果设置了binlog_format为ROW，你可以将InnoDB的事务隔离基本设为READ COMMITTED，以获得更好的并发性。

（3）MIXED格式下，MySQL默认采用STATEMENT格式进行二进制日志文件的记录，但是在一些情况下会使用ROW格式，可能的情况有：

1）表的存储引擎为NDB，这时对于表的DML操作都会以ROW格式记录。

2）使用了UUID()、USER()、CURRENT_USER()、FOUND_ROWS()、ROW_COUNT()等不确定函数。

3）使用了INSERT DELAY语句。

4）使用了用户定义函数（UDF）。

5）使用了临时表（temporary table）。

此外，binlog_format参数还有对于存储引擎的限制，如表3-1所示。

表3-1 存储引擎二进制日志格式支持情况

存储引擎	Row格式	Statement格式
InnoDB	支持	支持
MyISAM	支持	支持
HEAP	支持	支持
MERGE	支持	支持
NDB	支持	不支持
Archive	支持	支持
CSV	支持	支持
Federate	支持	支持
Blockhole	不支持	支持

binlog_format是动态参数，因此可以在数据库运行环境下进行更改，例如，我们可以将当前会话的binlog_format设为ROW，如：

```
mysql> set @@session.binlog_format='ROW';
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> select @@session.binlog_format;
+-------------------------------+
| @@session.binlog_format |
+-------------------------------+
| ROW                  |
+-------------------------------+
1 row in set (0.00 sec)
```

当然，也可以将全局的binlog_format设置为你想要的格式。不过通常情况下，这个操作可能会带来问题，运行时，请确保更改后不会对你的复制带来影响。如：

```
mysql> set global binlog_format='ROW';
Query OK, 0 rows affected (0.00 sec)

mysql> select @@global.binlog_format;
+-------------------------------+
| @@global.binlog_format |
+-------------------------------+
| ROW              |
+-------------------------------+
1 row in set (0.00 sec)
```

通常情况下，我们将参数binlog_format设置为ROW，这可以为数据库的恢复和复制带来更好的可靠性。但是不能忽略一点的是，这会带来二进制文件大小的增加，有些语句下的ROW格式可能需要更大的容量。比如我们有两张一样的表，大小都为100W，执行UPDATE操作，观察二进制日志大小的变化：

```
mysql> select @@session.binlog_format\G;
*************************** 1. row ***************************
@@session.binlog_format: STATEMENT
1 row in set (0.00 sec)

mysql> show master status\G;
*************************** 1. row ***************************
             File: test.000003
         Position: 106
     Binlog_Do_DB:
Binlog_Ignore_DB:
1 row in set (0.00 sec)

mysql> update t1 set username=upper(username);
Query OK, 89279 rows affected (1.83 sec)
```

```
Rows matched: 100000  Changed: 89279  Warnings: 0

mysql> show master status\G;
*************************** 1. row ***************************
            File: test.000003
        Position: 306
    Binlog_Do_DB:
Binlog_Ignore_DB:
1 row in set (0.00 sec)
```

可以看到，在binlog_format格式为STATEMENT下，执行UPDATE语句二进制日志大小只增加了200字节（306-106）。如果我们使用ROW格式，同样来操作t2表，可以看到：

```
mysql> set session binlog_format='ROW';
Query OK, 0 rows affected (0.00 sec)

mysql> show master status\G;
*************************** 1. row ***************************
            File: test.000003
        Position: 306
    Binlog_Do_DB:
Binlog_Ignore_DB:
1 row in set (0.00 sec)

mysql> update t2 set username=upper(username);
Query OK, 89279 rows affected (2.42 sec)
Rows matched: 100000  Changed: 89279  Warnings: 0

mysql> show master status\G;
*************************** 1. row ***************************
            File: test.000003
        Position: 13782400
    Binlog_Do_DB:
Binlog_Ignore_DB:
1 row in set (0.00 sec)
```

这时你会惊讶地发现，同样的操作在ROW格式下竟然需要13 782 094字节，二进制日志文件差不多增加了13MB，要知道t2表的大小也不超过17MB。而且执行时间也有所增加（这里我设置了sync_binlog=1）。这就是因为，这时MySQL数据库不再将逻辑的SQL操作记录到二进制日志，而是记录对于每行的更改记录信息。

上面的这个例子告诉我们，将参数binlog_format设置为ROW，对于磁盘空间要求有了

一定的增加。而由于复制是采用传输二进制日志方式实现的，因此复制的网络开销也有了增加。

二进制日志文件的文件格式为二进制（好像有点废话），不能像错误日志文件，慢查询日志文件用cat、head、tail等命令来查看。想要查看二进制日志文件的内容，须通过MySQL提供的工具mysqlbinlog。对于STATEMENT格式的二进制日志文件，使用mysqlbinlog后，看到就是执行的逻辑SQL语句，如：

```
[root@nineyou0-43 data]# mysqlbinlog --start-position=203 test.000004
/*!40019 SET @@session.max_insert_delayed_threads=0*/;
....
#090927 15:43:11 server id 1  end_log_pos 376    Query    thread_id=188
exec_time=1     error_code=0
SET TIMESTAMP=1254037391/*!*/;
update t2 set username=upper(username) where id=1
/*!*/;
# at 376
#090927 15:43:11 server id 1  end_log_pos 403   Xid = 1009
COMMIT/*!*/;
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

update t2 set username=upper（username） where id=1，这个可以看到日志的记录以SQL语句的方式（为了排版的方便，省去了一些开始的信息）。在这个情况下，mysqlbinlog和Oracle LogMiner类似。但是如果这时使用ROW格式的记录方式，则会发现mysqlbinlog的结果变得“不可读”（unreadable），如：

```
[root@nineyou0-43 data]# mysqlbinlog  --start-position=1065 test.000004
/*!40019 SET @@session.max_insert_delayed_threads=0*/;
......
# at 1135
# at 1198
#090927 15:53:52 server id 1  end_log_pos 1198  Table_map: 'member'.'t2' mapped
to number 58
#090927 15:53:52 server id 1  end_log_pos 1378  Update_rows: table id 58 flags:
STMT_END_F

BINLOG '
EBq/ShMBAAAAPwAAAK4EAAAAADoAAAAAAAAABm1lbWJlcgACdDIACgMPDw/+CgsPAQwKJAAoAEAA
```

```
/gJAAAAA
EBq/ShgBAAAAtAAAAGIFAAAQADoAAAAAAAEACv////8A/AEAAAALYWxleDk5ODh5b3UEOXlvdSA3
Y2JiMzI1MmJhNmI3ZTljNDIyZmFjNTMzNGQyMjA1NAFNLacPAAAAAABjEnpxPBIAAAD8AQAAAAtB
TEVYOTk4OFlPVQQ5eW91IDdjYmIzMjUyYmE2YjdlOWM0MjJmYWM1MzM0ZDIyMDU0AU0tpw8AAAAA
AGMSenE8EgAA
'/*!*/;
# at 1378
#090927 15:53:52 server id 1  end_log_pos 1405  Xid = 1110
COMMIT/*!*/;
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

我们看不到执行的SQL语句，反而是一大串我们看不到的字符。其实只要加上参数-v或者-vv，就能清楚地看到执行的具体信息了，-vv会比-v多显示出更新的类型，这次我们加上-vv选项，得到：

```
[root@nineyou0-43 data]# mysqlbinlog -vv  --start-position=1065 test.000004
......
BINLOG '
EBq/ShMBAAAAPwAAAK4EAAAAADoAAAAAAAAABm1lbWJlcgACdDIACgMPDw/+CgsPAQwKJAAoAEAA
/gJAAAAA
EBq/ShgBAAAAtAAAAGIFAAAQADoAAAAAAAEACv////8A/AEAAAALYWxleDk5ODh5b3UEOXlvdSA3
Y2JiMzI1MmJhNmI3ZTljNDIyZmFjNTMzNGQyMjA1NAFNLacPAAAAAABjEnpxPBIAAAD8AQAAAAtB
TEVYOTk4OFlPVQQ5eW91IDdjYmIzMjUyYmE2YjdlOWM0MjJmYWM1MzM0ZDIyMDU0AU0tpw8AAAAA
AGMSenE8EgAA
'/*!*/;
### UPDATE member.t2
### WHERE
###   @1=1 /* INT meta=0 nullable=0 is_null=0 */
###   @2='david' /* VARSTRING(36) meta=36 nullable=0 is_null=0 */
###   @3='family' /* VARSTRING(40) meta=40 nullable=0 is_null=0 */
###   @4='7cbb3252ba6b7e9c422fac5334d22054' /* VARSTRING(64) meta=64 nullable=0
is_null=0 */
###   @5='M' /* STRING(2) meta=65026 nullable=0 is_null=0 */
###   @6='2009:09:13' /* DATE meta=0 nullable=0 is_null=0 */
###   @7='00:00:00' /* TIME meta=0 nullable=0 is_null=0 */
###   @8='' /* VARSTRING(64) meta=64 nullable=0 is_null=0 */
###   @9=0 /* TINYINT meta=0 nullable=0 is_null=0 */
###   @10=2009-08-11 16:32:35 /* DATETIME meta=0 nullable=0 is_null=0 */
### SET
###   @1=1 /* INT meta=0 nullable=0 is_null=0 */
```

```
###   @2='DAVID' /* VARSTRING(36) meta=36 nullable=0 is_null=0 */
###   @3=family /* VARSTRING(40) meta=40 nullable=0 is_null=0 */
###   @4='7cbb3252ba6b7e9c422fac5334d22054' /* VARSTRING(64) meta=64 nullable=0
is_null=0 */
###   @5='M' /* STRING(2) meta=65026 nullable=0 is_null=0 */
###   @6='2009:09:13' /* DATE meta=0 nullable=0 is_null=0 */
###   @7='00:00:00' /* TIME meta=0 nullable=0 is_null=0 */
###   @8='' /* VARSTRING(64) meta=64 nullable=0 is_null=0 */
###   @9=0 /* TINYINT meta=0 nullable=0 is_null=0 */
###   @10=2009-08-11 16:32:35 /* DATETIME meta=0 nullable=0 is_null=0 */
# at 1378
#090927 15:53:52 server id 1  end_log_pos 1405  Xid = 1110
COMMIT/*!*/;
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

现在mysqlbinlog向我们解释了具体做的事情。可以看到，一句简单的update t2 set username=upper（username）where id=1语句记录为了对于整个行更改的信息，这也解释了为什么前面我们更新了10万行的数据，在ROW格式下，二进制日志文件会增大了13MB。

3.3 套接字文件

前面提到过，Unix系统下本地连接MySQL可以采用Unix域套接字方式，这种方式需要一个套接字（socke）文件。套接字文件可由参数socket控制。一般在/tmp目录下，名为mysql.sock：

```
mysql> show variables like 'socket'\G;
*************************** 1. row ***************************
Variable_name: socket
      Value: /tmp/mysql.sock
1 row in set (0.00 sec)
```

3.4 pid文件

当MySQL实例启动时，会将自己的进程ID写入一个文件中——该文件即为pid文件。该文件可由参数pid_file控制。默认路径位于数据库目录下，文件名为主机名.pid。

```
mysql> show variables like 'pid_file'\G;
*************************** 1. row ***************************
Variable_name: pid_file
        Value: /usr/local/mysql/data/xen-server.pid
1 row in set (0.00 sec)
```

3.5 表结构定义文件

因为MySQL插件式存储引擎的体系结构的关系，MySQL对于数据的存储是按照表的，所以每个表都会有与之对应的文件（对比SQL Server是按照每个数据库下的所有表或索引都存在mdf文件中）。不论采用何种存储引擎，MySQL都有一个以frm为后缀名的文件，这个文件记录了该表的表结构定义。

frm还用来存放视图的定义，如我们创建了一个v_a视图，那么对应地会产生一个v_a.frm文件，用来记录视图的定义，该文件是文本文件，可以直接使用cat命令进行查看：

```
[root@xen-server test]# cat v_a.frm
TYPE=VIEW
query=select 'test'.'a'.'b' AS 'b' from 'test'.'a'
md5=4eda70387716a4d6c96f3042dd68b742
updatable=1
algorithm=0
definer_user=root
definer_host=localhost
suid=2
with_check_option=0
timestamp=2010-08-04 07:23:36
create-version=1
source=select * from a
client_cs_name=utf8
connection_cl_name=utf8_general_ci
view_body_utf8=select 'test'.'a'.'b' AS 'b' from 'test'.'a'
```

3.6 InnoDB存储引擎文件

之前介绍的文件都是MySQL数据库本身的文件，和存储引擎无关。除了这些文件外，每个表存储引擎还有其自己独有的文件。这一节将具体介绍和InnoDB存储引擎密切相关的文件，这些文件包括重做日志文件、表空间文件。

3.6.1 表空间文件

InnoDB存储引擎在存储设计上模仿了Oracle，将存储的数据按表空间进行存放。默认配置下，会有一个初始化大小为10MB、名为ibdata1的文件。该文件就是默认的表空间文件（tablespace file）。你可以通过参数innodb_data_file_path对其进行设置。格式如下：

```
innodb_data_file_path=datafile_spec1[;datafile_spec2]...
```

你也可以用多个文件组成一个表空间，同时制定文件的属性，如：

```
[mysqld]
innodb_data_file_path = /db/ibdata1:2000M;/dr2/db/ibdata2:2000M:autoextend
```

这里将/db/ibdata1和/dr2/db/ibdata2两个文件用来组成表空间。若这两个文件位于不同的磁盘上，则可以对性能带来一定程度的提升。两个文件的文件名后都跟了属性，表示文件idbdata1的大小为2000MB，文件ibdata2的大小为2000MB，但是如果用满了这2000MB后，该文件可以自动增长（autoextend）。

设置innodb_data_file_path参数后，之后对于所有基于InnoDB存储引擎的表的数据都会记录到该文件内。而通过设置参数innodb_file_per_table，我们可以将每个基于InnoDB存储引擎的表单独产生一个表空间，文件名为表名.ibd，这样不用将所有数据都存放于默认的表空间中。下面这台服务器设置了innodb_file_per_table，可以看到：

```
mysql> show variables like 'innodb_file_per_table'\G;
*************************** 1. row ***************************
Variable_name: innodb_file_per_table
        Value: ON
1 row in set (0.00 sec)

mysql> system ls -lh /usr/local/mysql/data/member/*
-rw-r-----  1 mysql mysql 8.7K 2009-02-24  /usr/local/mysql/data/member/
Profile.frm
-rw-r-----  1 mysql mysql 1.7G  9月 25 11:13 /usr/local/mysql/data/member/
Profile.ibd
-rw-rw----  1 mysql mysql 8.7K  9月 27 13:38 /usr/local/mysql/data/member/t1.frm
-rw-rw----  1 mysql mysql  17M  9月 27 13:40 /usr/local/mysql/data/member/t1.ibd
-rw-rw----  1 mysql mysql 8.7K  9月 27 15:42 /usr/local/mysql/data/member/t2.frm
-rw-rw----  1 mysql mysql  17M  9月 27 15:54 /usr/local/mysql/data/member/t2.ibd
```

表Profile、t1、t2都是InnoDB的存储引擎，由于设置参数innodb_file_per_table=ON，

因此产生了单独的.ibd表空间文件。需要注意的是，这些单独的表空间文件仅存储该表的数据、索引和插入缓冲等信息，其余信息还是存放在默认的表空间中。图3-1显示了InnoDB存储引擎对于文件的存储方式：

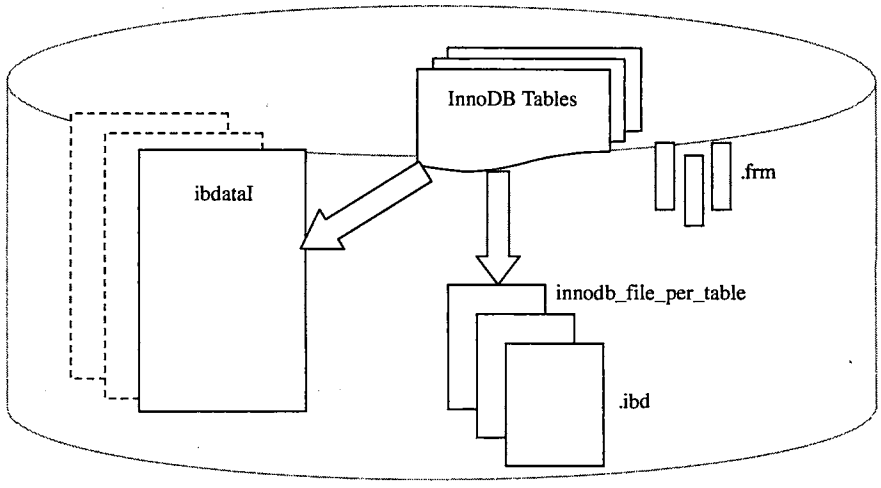


图3-1 InnoDB表存储引擎文件

3.6.2 重做日志文件

默认情况下会有两个文件，名称分别为ib_logfile0和ib_logfile1。MySQL官方手册中将其称为InnoDB存储引擎的日志文件，不过更准确的定义应该是重做日志文件（redo log file）。为什么强调是重做日志文件呢？因为重做日志文件对于InnoDB存储引擎至关重要，它们记录了对于InnoDB存储引擎的事务日志。

重做日志文件的主要目的是，万一实例或者介质失败（media failure），重做日志文件就能派上用场。如数据库由于所在主机掉电导致实例失败，InnoDB存储引擎会使用重做日志恢复到掉电前的时刻，以此来保证数据的完整性。

每个InnoDB存储引擎至少有1个重做日志文件组（group），每个文件组下至少有2个重做日志文件，如默认的ib_logfile0、ib_logfile1。为了得到更高的可靠性，你可以设置多个镜像日志组（mirrored log groups），将不同的文件组放在不同的磁盘上。日志组中每个重做日志文件的大小一致，并以循环方式使用。InnoDB存储引擎先写重做日志文件1，当达到文件的最后时，会切换至重做日志文件2，当重做日志文件2也被写满时，会再切换到重

做日志文件1中。图3-2显示了一个拥有3个重做日志文件的重做日志文件组。

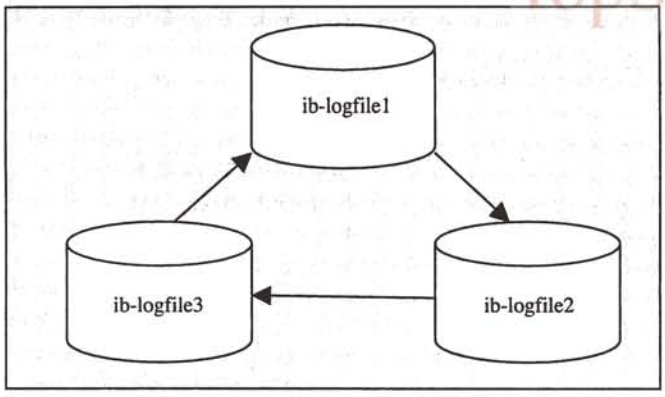


图3-2 日志文件组

参数innodb_log_file_size、innodb_log_files_in_group、innodb_mirrored_log_groups、innodb_log_group_home_dir影响着重做日志文件的属性。参数innodb_log_file_size指定了重做日志文件的大小；innodb_log_files_in_group指定了日志文件组中重做日志文件的数量，默认为2；innodb_mirrored_log_groups指定了日志镜像文件组的数量，默认为1，代表只有一个日志文件组，没有镜像；innodb_log_group_home_dir指定了日志文件组所在路径，默认在数据库路径下。以下显示了一个关于重做日志组的配置：

```
mysql> show variables like 'innodb%log%'\G;
*************************** 1. row ***************************
Variable_name: innodb_flush_log_at_trx_commit
        Value: 1
*************************** 2. row ***************************
Variable_name: innodb_locks_unsafe_for_binlog
        Value: OFF
*************************** 3. row ***************************
Variable_name: innodb_log_buffer_size
        Value: 8388608
*************************** 4. row ***************************
Variable_name: innodb_log_file_size
        Value: 5242880
*************************** 5. row ***************************
Variable_name: innodb_log_files_in_group
        Value: 2
*************************** 6. row ***************************
Variable_name: innodb_log_group_home_dir
```

```
        Value: ./
*************************** 7. row ***************************
Variable_name: innodb_mirrored_log_groups
        Value: 1
7 rows in set (0.00 sec)
```

重做日志文件的大小设置对于MySQL数据库各方面还是有影响的。一方面不能设置得太大，如果设置得很大，在恢复时可能需要很长的时间；另一方面又不能太小了，否则可能导致一个事务的日志需要多次切换重做日志文件。在错误日志中可能会看到如下警告：

```
090924 11:39:44  InnoDB: ERROR: the age of the last checkpoint is 9433712,
InnoDB: which exceeds the log group capacity 9433498.
InnoDB: If you are using big BLOB or TEXT rows, you must set the
InnoDB: combined size of log files at least 10 times bigger than the
InnoDB: largest such row.
090924 11:40:00  InnoDB: ERROR: the age of the last checkpoint is 9433823,
InnoDB: which exceeds the log group capacity 9433498.
InnoDB: If you are using big BLOB or TEXT rows, you must set the
InnoDB: combined size of log files at least 10 times bigger than the
InnoDB: largest such row.
090924 11:40:16  InnoDB: ERROR: the age of the last checkpoint is 9433645,
InnoDB: which exceeds the log group capacity 9433498.
InnoDB: If you are using big BLOB or TEXT rows, you must set the
InnoDB: combined size of log files at least 10 times bigger than the
InnoDB: largest such row.
```

上面错误集中在InnoDB: ERROR: the age of the last checkpoint is 9433645,InnoDB: which exceeds the log group capacity 9433498。这是因为重做日志有一个capacity变量，该值代表了最后的检查点不能超过这个阈值，如果超过则必须将缓冲池（innodb buffer pool）中刷新列表（flush list）中的部分脏数据页写回磁盘。

也许有人会问，既然同样是记录事务日志，那和我们之前的二进制日志有什么区别？首先，二进制日志会记录所有与MySQL有关的日志记录，包括InnoDB、MyISAM、Heap等其他存储引擎的日志。而InnoDB存储引擎的重做日志只记录有关其本身的事务日志。其次，记录的内容不同，不管你将二进制日志文件记录的格式设为STATEMENT还是ROW，又或者是MIXED，其记录的都是关于一个事务的具体操作内容。而InnoDB存储引擎的重做日志文件记录的关于每个页（Page）的更改的物理情况（如表3-2所示）。此外，写入的时间也不同，二进制日志文件是在事务提交前进行记录的，而在事务进行的过程中，不断

有重做日志条目（redo entry）被写入重做日志文件中。

表3-2 重做日志结构

Space id	PageNo	OpCode	Data

在第2章中已经提到，对于写入重做日志文件的操作不是直接写，而是先写入一个重做日志缓冲（redo log buffer）中，然后根据按照一定的条件写入日志文件。图3-3很好地表示了这个过程。

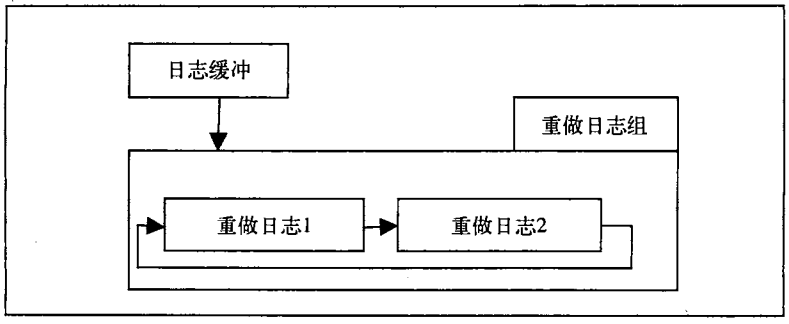


图3-3 重做日志写入过程

上面提到了从日志缓冲写入磁盘上的重做日志文件是按一定条件的，那这些条件有哪些呢？第2章分析了主线程（master thread），知道在主线程中每秒会将重做日志缓冲写入磁盘的重做日志文件中，不论事务是否已经提交。另一个触发这个过程是由参数innodb_flush_log_at_trx_commit控制，表示在提交（commit）操作时，处理重做日志的方式。

参数innodb_flush_log_at_trx_commit可设的值有0、1、2。0代表当提交事务时，并不将事务的重做日志写入磁盘上的日志文件，而是等待主线程每秒的刷新。而1和2不同的地方在于：1是在commit时将重做日志缓冲同步写到磁盘；2是重做日志异步写到磁盘，即不能完全保证commit时肯定会写入重做日志文件，只是有这个动作。

3.7 小结

本章介绍了与MySQL数据库相关的一些文件，并了解了文件可以分为MySQL数据库文件以及和各存储引擎有关的文件。与MySQL数据库有关的文件中，错误文件和二进制日志文件非常重要。当MySQL数据库发生任何错误时，DBA首先就应该去查看错误文件，

从文件提示的内容中找出问题的所在。当然，错误文件不仅记录了错误的内容，也记录了警告的信息，通过一些警告也有助于DBA对于数据库和存储引擎的优化。

二进制日志的作用非常关键，可以用来进行point in time的恢复以及复制（replication）环境的搭建。因此，建议在任何时候时都启用二进制日志的记录。从MySQL 5.1开始，二进制日志支持STATEMENT、ROW、MIX三种格式，用来更好地同步数据库。DBA应该十分清楚三种不同格式之间的差异。

本章的最后介绍了和InnoDB存储引擎相关的文件，包括表空间文件和重做日志文件。表空间文件是用来管理InnoDB存储引擎的存储，分为共享表空间和独立表空间。重做日志非常重要，用来记录InnoDB存储引擎的事务日志，也因为重做日志的存在，才使得InnoDB存储引擎可以提供可靠的事务。

第4章 表

本章将从对InnoDB存储引擎表的基本介绍开始，然后重点分析表的物理存储特征——数据是如何组织和存放的。简单来说，表就是关于特定实体的数据集合，这也是关系型数据库模型的核心。

4.1 InnoDB存储引擎表类型

对比Oracle支持的各种表类型，InnoDB存储引擎表更像是Oracle中的索引组织表（index organized table）。在InnoDB存储引擎表中，每张表都有个主键，如果在创建表时没有显式地定义主键（Primary Key），则InnoDB存储引擎会按如下方式选择或创建主键。

- 首先表中是否有非空的唯一索引（Unique NOT NULL），如果有，则该列即为主键。
- 不符合上述条件，InnoDB存储引擎自动创建一个6个字节大小的指针。

4.2 InnoDB逻辑存储结构

InnoDB存储引擎的逻辑存储结构和Oracle大致相同，所有数据都被逻辑地存放在一个空间中，我们称之为表空间（tablespace）。表空间又由段（segment）、区（extent）、页（page）组成。页在一些文档中有时也称为块（block），InnoDB存储引擎的逻辑存储结构大致如图4-1所示。

4.2.1 表空间

表空间可以看做是InnoDB存储引擎逻辑结构的最高层，所有的数据都是存放在表空间中。第3章中已经介绍了默认情况下InnoDB存储引擎有一个共享表空间ibdata1，即所有数据都放在这个表空间内。如果我们启用了参数innodb_file_per_table，则每张表内的数据可以单独放到一个表空间内。

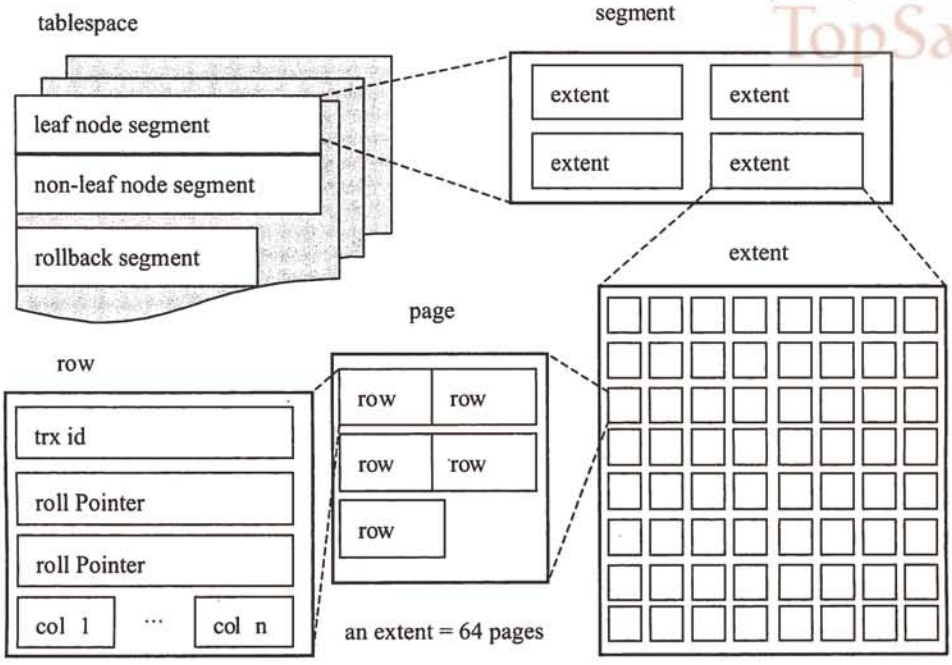


图4-1 InnoDB存储引擎的逻辑存储结构

对于启用了innodb_file_per_table的参数选项，需要注意的是，每张表的表空间内存放的只是数据、索引和插入缓冲，其他类的数据，如撤销（Undo）信息、系统事务信息、二次写缓冲（double write buffer）等还是存放在原来的共享表空间内。这也就说明了另一个问题：即使在启用了参数innodb_file_per_table之后，共享表空间还是会不断地增加其大小。现在我们来做个实验，实验之前我已经将innodb_file_per_table设为ON了，看看初始共享表空间文件有多大：

```
mysql>  show variables like 'innodb_file_per_table'\G;
*************************** 1. row ***************************
Variable_name: innodb_file_per_table
        Value: ON
1 row in set (0.00 sec)

mysql> system ls -lh /usr/local/mysql/data/ib*
-rw-rw---- 1 mysql mysql 58M Mar 11 13:58 /usr/local/mysql/data/ibdata1
-rw-rw---- 1 mysql mysql 64M Mar 11 13:58 /usr/local/mysql/data/ib_logfile0
-rw-rw---- 1 mysql mysql 64M Mar 11 13:58 /usr/local/mysql/data/ib_logfile1
```

可以看到，共享表空间ibdata1的大小为58M，接下去我们产生Undo操作，利用第1章

我们产生的mytest表，并把其存储引擎更改为InnoDB，执行如下操作：

```
mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> update mytest set salary=0;
Query OK, 2844047 rows affected (19.47 sec)
Rows matched: 2844047  Changed: 2844047  Warnings: 0

mysql> system ls -lh /usr/local/mysql/data/ib*
-rw-rw---- 1 mysql mysql 114M Mar 11 14:00 /usr/local/mysql/data/ibdata1
-rw-rw---- 1 mysql mysql  64M Mar 11 14:00 /usr/local/mysql/data/ib_logfile0
-rw-rw---- 1 mysql mysql  64M Mar 11 14:00 /usr/local/mysql/data/ib_logfile1
```

首先将自动提交设为0，即我们需要显式提交事务（注意，上面结束时我们并没有commit或者rollback该事务）。接着我们执行会产生大量Undo操作的语句update mytest set salary=0，完成后我们再观察共享表空间，会发现ibdata1已经增长到了114MB，这就说明了共享表空间中还包含有Undo信息。有人会问，如果我rollback这个事务，ibdata1这个表空间会不会缩减至原来的58MB大小？我们接下去就来验证：

```
mysql> rollback;
Query OK, 0 rows affected (0.00 sec)

mysql> system ls -lh /usr/local/mysql/data/ib*
-rw-rw---- 1 mysql mysql 114M Mar 11 14:00 /usr/local/mysql/data/ibdata1
-rw-rw---- 1 mysql mysql  64M Mar 11 14:00 /usr/local/mysql/data/ib_logfile0
-rw-rw---- 1 mysql mysql  64M Mar 11 14:00 /usr/local/mysql/data/ib_logfile1
```

很“可惜”，还是114MB，即InnoDB存储引擎不会在rollback时去收缩这个表空间。虽然InnoDB不会帮你回收这些空间，但是MySQL会自动判断这些Undo信息是否需要，如果不需要，则会将这些空间标记为可用空间，供下次Undo使用。回想一下我们在第2章中提到的master thread每10秒会执行一次full purge操作。因此很有可能的一种情况是，你再次执行上述的UPDATE语句后，会发现ibdata1不会再增大了，那就是这个原因了。

我用python编写了一个py_innodb_page_info小工具，用来查看表空间中各页的类型和信息，你可以在code.google.com上搜索david-mysql-tools进行查找。使用方法如下：

```
[root@nineyou0-43 py]# python py_innodb_page_info.py /usr/local/mysql/data/ibdata1
Total number of page: 83584:
Insert Buffer Free List: 204
```

```
Freshly Allocated Page: 5467
Undo Log Page: 38675
File Segment inode: 4
B-tree Node: 39233
File Space Header: 1
```

可以看到共有83 584个页，其中插入缓冲的空闲列表有204个页、5467个可用页、38 675个Undo页、39 233个数据页，等等。你可以通过添加-v参数来查看更详细的内容。由于该工具还在开发之中，因此并不保证在本书出版时此工具最终显示结果的变化。

4.2.2 段

图4-1中显示了表空间是由各个段组成的，常见的段有数据段、索引段、回滚段等。因为前面已经介绍过了InnoDB存储引擎表是索引组织的（index organized），因此数据即索引，索引即数据。那么数据段即为B+树的页节点（图4-1的leaf node segment），索引段即为B+树的非索引节点（图4-1的non-leaf node segment）。

与Oracle不同的是，InnoDB存储引擎对于段的管理是由引擎本身完成，这和Oracle的自动段空间管理（ASSM）类似，没有手动段空间管理（MSSM）的方式，这从一定程度上简化了DBA的管理。

需要注意的是，并不是每个对象都有段。因此更准确地说，表空间是由分散的页和段组成。

4.2.3 区

区是由64个连续的页组成的，每个页大小为16KB，即每个区的大小为1MB。对于大的数据段，InnoDB存储引擎最多每次可以申请4个区，以此来保证数据的顺序性能。

但是，这里还有这样一个问题：在我们启用了参数innodb_file_per_talbe后，创建的表默认大小是96KB。区是64个连续的页，那创建的表的大小至少是1MB才对啊？其实这是因为在每个段开始时，先有32个页大小的碎片页（fragment page）来存放数据，当这些页使用完之后才是64个连续页的申请。这里通过一个实验来显示InnoDB存储引擎对于区的申请：

```
mysql>create table t1 (
coll int not null auto_increment,
```

```
col2 varchar(7000) ,
primary key (col1))engine=InnoDB;

mysql> system ls -lh /usr/local/mysql/data/test/t1.ibd;
-rw-rw----  1 mysql mysql 96K 10月 12 14:59 /usr/local/mysql/data/test/tl.ibd
```

我们创建了t1表，col2字段设为varchar（7000），这样能保证一个页中可以存放2条记录。可以看到，初始创建完t1后表空间默认大小为96KB，接着运行如下SQL语句：

```
mysql> insert into t1 select NULL,repeat('a',7000);
Query OK, 1 row affected (0.04 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> insert into t1 select NULL,repeat('a',7000);
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> system ls -lh /usr/local/mysql/data/test/t1.ibd;
-rw-rw----  1 mysql mysql 96K 10月 12 16:24 /usr/local/mysql/data/test/t1.ibd
```

插入两条记录，这两条记录应该在一个页中。如果用py_innodb_page_info工具来查看表空间，可以看到：

```
[root@nineyou0-43 py]# ./py_innodb_page_info.py -v /usr/local/mysql/data/test/t1.ibd
page offset 00000000, page type <File Space Header>
page offset 00000001, page type <Insert Buffer Bitmap>
page offset 00000002, page type <File Segment inode>
page offset 00000003, page type <B-tree Node>, page level <0000>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
Total number of page: 6:
Freshly Allocated Page: 2
Insert Buffer Bitmap: 1
File Space Header: 1
B-tree Node: 1
File Segment inode: 1
```

这次我用-v详细模式来看表空间的内容，注意到了page offset为3的页，这个就是数据页，page level表示所在索引层，0表示叶节点。因为当前所有记录都在一个页中，因此没有非叶节点。但是如果这时我们再插入一条记录，就会产生一个非页节点了：

```
mysql> insert into t1 select NULL,repeat('a',7000);
Query OK, 1 row affected (0.01 sec)
```

```
Records: 1  Duplicates: 0  Warnings: 0

[root@nineyou0-43 py]# ./py_innodb_page_info.py -v /usr/local/mysql/data/test/t1.ibd
page offset 00000000, page type <File Space Header>
page offset 00000001, page type <Insert Buffer Bitmap>
page offset 00000002, page type <File Segment inode>
page offset 00000003, page type <B-tree Node>, page level <0001>
page offset 00000004, page type <B-tree Node>, page level <0000>
page offset 00000005, page type <B-tree Node>, page level <0000>
Total number of page: 6:
Insert Buffer Bitmap: 1
File Space Header: 1
B-tree Node: 3
File Segment inode: 1
```

现在我们可以看到page level是1的页了，但这个页的类型还是B-tree Node。然后再插入60条记录，也就是说一共有63条记录，共32个页，在这之前先建立一个导入的存储过程：

```
mysql>delimieter //

mysql> create procedure load_t1(count int unsigned)
    -> begin
    -> declare s int unsigned default 1;
    -> declare c varchar(7000) default repeat('a',7000);
    -> while s <= count do
    -> insert into t1 select NULL,c;
    -> set s = s+1;
    -> end while;
    -> end;
    -> //
Query OK, 0 rows affected (0.04 sec)

mysql> delimiter ;

mysql> call load_t1(60);
Query OK, 1 row affected (1.59 sec)

mysql> select count(*) from t1\G;
*************************** 1. row ***************************
count(*): 63
1 row in set (0.00 sec)1 row in set (0.00 sec)
```

```
mysql> system ls -lh /usr/local/mysql/data/test/t1.ibd;
-rw-rw----  1 mysql mysql 576K 10月 12 16:56 /usr/local/mysql/data/test/t1.ibd
```

可以看到，在导入了63条数据后，数据空间的申请还是通过碎片区，而不是通过64个连续页的区。这时如果用py_innodb_page_info工具再来看表空间文件，可得：

```
[root@nineyou0-43 py]# ./py_innodb_page_info.py -v /usr/local/mysql/data/test/t1.ibd
page offset 00000000, page type <File Space Header>
page offset 00000001, page type <Insert Buffer Bitmap>
page offset 00000002, page type <File Segment inode>
page offset 00000003, page type <B-tree Node>, page level <0001>
page offset 00000004, page type <B-tree Node>, page level <0000>
page offset 00000005, page type <B-tree Node>, page level <0000>
page offset 00000006, page type <B-tree Node>, page level <0000>
page offset 00000007, page type <B-tree Node>, page level <0000>
page offset 00000008, page type <B-tree Node>, page level <0000>
page offset 00000009, page type <B-tree Node>, page level <0000>
page offset 0000000a, page type <B-tree Node>, page level <0000>
page offset 0000000b, page type <B-tree Node>, page level <0000>
page offset 0000000c, page type <B-tree Node>, page level <0000>
page offset 0000000d, page type <B-tree Node>, page level <0000>
page offset 0000000e, page type <B-tree Node>, page level <0000>
page offset 0000000f, page type <B-tree Node>, page level <0000>
page offset 00000010, page type <B-tree Node>, page level <0000>
page offset 00000011, page type <B-tree Node>, page level <0000>
page offset 00000012, page type <B-tree Node>, page level <0000>
page offset 00000013, page type <B-tree Node>, page level <0000>
page offset 00000014, page type <B-tree Node>, page level <0000>
page offset 00000015, page type <B-tree Node>, page level <0000>
page offset 00000016, page type <B-tree Node>, page level <0000>
page offset 00000017, page type <B-tree Node>, page level <0000>
page offset 00000018, page type <B-tree Node>, page level <0000>
page offset 00000019, page type <B-tree Node>, page level <0000>
page offset 0000001a, page type <B-tree Node>, page level <0000>
page offset 0000001b, page type <B-tree Node>, page level <0000>
page offset 0000001c, page type <B-tree Node>, page level <0000>
page offset 0000001d, page type <B-tree Node>, page level <0000>
page offset 0000001e, page type <B-tree Node>, page level <0000>
page offset 0000001f, page type <B-tree Node>, page level <0000>
page offset 00000020, page type <B-tree Node>, page level <0000>
page offset 00000021, page type <B-tree Node>, page level <0000>
page offset 00000022, page type <B-tree Node>, page level <0000>
page offset 00000023, page type <B-tree Node>, page level <0000>
Total number of page: 36:
```

```
Insert Buffer Bitmap: 1
File Space Header: 1
B-tree Node: 33
File Segment inode: 1
```

你可以注意到page level等于0的页有32个，也就是说，对于数据段，已经有32个碎片页了，之后表空间的申请是连续的64个页的大小开始增长了。好的，接着我们就这样来做，插入一条数据，看之后表空间的大小：

```
mysql> call load_t1(1);
Query OK, 1 row affected (0.10 sec)

mysql> system ls -lh /usr/local/mysql/data/test/t1.ibd;
-rw-rw----  1 mysql mysql 2.0M 10月 12 17:02 /usr/local/mysql/data/test/t1.ibd
```

因为已经用完了32个碎片页，新的页会采用区的方式进行空间的申请，如果我们再用py_innodb_page_info工具来看表空间文件t1.ibd，应该可以看到很多页的类型为Freshly Allocated Page：

```
[root@nineyou0-43 test2]# ~/py/py_innodb_page_info.py t1.ibd -v
page offset 00000000, page type <File Space Header>
page offset 00000001, page type <Insert Buffer Bitmap>
page offset 00000002, page type <File Segment inode>
page offset 00000003, page type <B-tree Node>, page level <0001>
page offset 00000004, page type <B-tree Node>, page level <0000>
page offset 00000005, page type <B-tree Node>, page level <0000>
page offset 00000006, page type <B-tree Node>, page level <0000>
page offset 00000007, page type <B-tree Node>, page level <0000>
page offset 00000008, page type <B-tree Node>, page level <0000>
page offset 00000009, page type <B-tree Node>, page level <0000>
page offset 0000000a, page type <B-tree Node>, page level <0000>
page offset 0000000b, page type <B-tree Node>, page level <0000>
page offset 0000000c, page type <B-tree Node>, page level <0000>
page offset 0000000d, page type <B-tree Node>, page level <0000>
page offset 0000000e, page type <B-tree Node>, page level <0000>
page offset 0000000f, page type <B-tree Node>, page level <0000>
page offset 00000010, page type <B-tree Node>, page level <0000>
page offset 00000011, page type <B-tree Node>, page level <0000>
page offset 00000012, page type <B-tree Node>, page level <0000>
page offset 00000013, page type <B-tree Node>, page level <0000>
page offset 00000014, page type <B-tree Node>, page level <0000>
page offset 00000015, page type <B-tree Node>, page level <0000>
```

```
page offset 00000016, page type <B-tree Node>, page level <0000>
page offset 00000017, page type <B-tree Node>, page level <0000>
page offset 00000018, page type <B-tree Node>, page level <0000>
page offset 00000019, page type <B-tree Node>, page level <0000>
page offset 0000001a, page type <B-tree Node>, page level <0000>
page offset 0000001b, page type <B-tree Node>, page level <0000>
page offset 0000001c, page type <B-tree Node>, page level <0000>
page offset 0000001d, page type <B-tree Node>, page level <0000>
page offset 0000001e, page type <B-tree Node>, page level <0000>
page offset 0000001f, page type <B-tree Node>, page level <0000>
page offset 00000020, page type <B-tree Node>, page level <0000>
page offset 00000021, page type <B-tree Node>, page level <0000>
page offset 00000022, page type <B-tree Node>, page level <0000>
page offset 00000023, page type <B-tree Node>, page level <0000>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000040, page type <B-tree Node>, page level <0000>
page offset 00000000, page type <Freshly Allocated Page>
```

```
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
```

```
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
Total number of page: 128:
Freshly Allocated Page: 91
Insert Buffer Bitmap: 1
File Space Header: 1
B-tree Node: 34
File Segment inode: 1
```

4.2.4 页

同大多数数据库一样，InnoDB有页（page）的概念（也可以称为块），页是InnoDB磁盘管理的最小单位。与Oracle类似的是，Microsoft SQL Server数据库默认每页大小为8KB，不同于InnoDB页的大小（16KB），且不可以更改（也许通过更改源码可以）。

常见的页类型有：

□ 数据页（B-tree Node）。

□ Undo页（Undo Log Page）。

□ 系统页（System Page）。

□ 事务数据页（Transaction system Page）。

□ 插入缓冲位图页（Insert Buffer Bitmap）。

□ 插入缓冲空闲列表页（Insert Buffer Free List）。

- 未压缩的二进制大对象页（Uncompressed BLOB Page）。
- 压缩的二进制大对象页（Compressed BLOB Page）。

4.2.5 行

InnoDB存储引擎是面向行的（row-oriented），也就是说数据的存放按行进行存放。每个页存放的行记录也是有硬性定义的，最多允许存放16KB / 2 ~ 200行的记录，即7992行记录。这里提到面向行（row-oriented）的数据库，那么也就是说，还存在有面向列（column-orientied）的数据库。MySQL infobright储存引擎就是按列来存放数据的，这对于数据仓库下的分析类SQL语句的执行以及数据压缩很有好处。类似的数据库还有Sybase IQ、Google Big Table。面向列的数据库是当前数据库发展的一个方向，但是这超出了本书涵盖的内容。有兴趣的读者可以在网上寻找相关资料。

4.3 InnoDB物理存储结构

从物理意义上来看，InnoDB表由共享表空间、日志文件组（更准确地说，应该是Redo文件组）、表结构定义文件组成。若将innodb_file_per_table设置为on，则每个表将独立地产生一个表空间文件，以ibd结尾，数据、索引、表的内部数据字典信息都将保存在这个单独的表空间文件中。表结构定义文件以frm结尾，这个是与存储引擎无关的，任何存储引擎的表结构定义文件都一样，为.frm文件。

4.4 InnoDB行记录格式

InnoDB存储引擎和大多数数据库一样（如Oracle和Microsoft SQL Server数据库），记录是以行的形式存储的。这意味着页中保存着表中一行行的数据。到MySQL 5.1时，InnoDB存储引擎提供了Compact和Redundant两种格式来存放行记录数据，Redundant是为兼容之前版本而保留的，如果你阅读过InnoDB的源代码，会发现源代码中是用PHYSICAL RECORD （NEW STYLE）和PHYSICAL RECORD（OLD STYLE）来区分两种格式的。MySQL 5.1默认保存为Compact行格式。你可以通过命令SHOW TABLE STATUS LIKE

'table_name'来查看当前表使用的行格式，其中row_format就代表了当前使用的行记录结构类型。例如：

```
mysql> show table status like 'mytest%'\G;
*************************** 1. row ***************************
           Name: mytest
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 6
 Avg_row_length: 2730
    Data_length: 16384
Max_data_length: 0
   Index_length: 0
      Data_free: 0
 Auto_increment: NULL
    Create_time: 2009-03-17 13:33:50
    Update_time: NULL
     Check_time: NULL
      Collation: latin1_swedish_ci
       Checksum: NULL
 Create_options:
        Comment:
*************************** 2. row ***************************
           Name: mytest2
         Engine: InnoDB
        Version: 10
     Row_format: Redundant
           Rows: 0
 Avg_row_length: 0
    Data_length: 16384
Max_data_length: 0
   Index_length: 0
      Data_free: 0
 Auto_increment: NULL
    Create_time: 2009-03-17 13:57:23
    Update_time: NULL
     Check_time: NULL
      Collation: latin1_swedish_ci
       Checksum: NULL
 Create_options: row_format=REDUNDANT
        Comment:
2 rows in set (0.00 sec)
```

可以看到，这里的mytest1表是Compact的行格式，mytest2表是Redundant的行格式。数据库实例的一个作用就是读取页中存放的行记录。如果我们知道规则，那么也可以读取其中的记录，如之前的py_innodb_page_info工具。下面将具体分析各格式存放数据的规则。

4.4.1 Compact 行记录格式

Compact行记录是在MySQL 5.0时被引入的，其设计目标是能高效存放数据。简单来说，如果一个页中存放的行数据越多，其性能就越高。Compact行记录以如下方式进行存储：

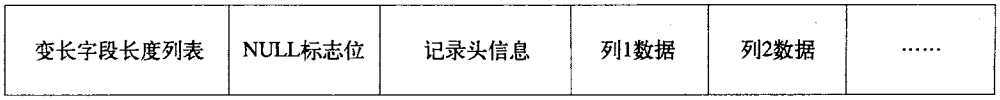


图4-2 Compact行记录格式

从图4-2可以看到，Compact行格式的首部是一个非NULL变长字段长度列表，而且是按照列的顺序逆序放置的。当列的长度小于255字节，用1字节表示，若大于255个字节，用2个字节表示，变长字段的长度最大不可以超过2个字节（这也很好地解释了为什么MySQL中varchar的最大长度为65 535，因为2个字节为16位，即2^{16}=1=65 535）。第二个部分是NULL标志位，该位指示了该行数据中是否有NULL值，用1表示。该部分所占的字节应该为 bytes。接下去的部分是为记录头信息（record header），固定占用5个字节（40位），每位的含义见表4-1。最后的部分就是实际存储的每个列的数据了，需要特别注意的是，NULL不占该部分任何数据，即NULL除了占有NULL标志位，实际存储不占有任何空间。另外有一点需要注意的是，每行数据除了用户定义的列外，还有两个隐藏列，事务ID列和回滚指针列，分别为6个字节和7个字节的大小。若InnoDB表没有定义Primary Key，每行还会增加一个6字节的RowID列。

表4-1 Compact页格式

名称	大小(bit)	描述
()	1	未知
()	1	未知
deleted_flag	1	该行是否已被删除
min_rec_flag	1	为1，如果该记录是预先被定义为最小的记录
n_owned	4	该记录拥有的记录数

（续）

名称	大小（bit）	描述
heap_no	13	索引堆中该条记录的排序记录
record_type	3	记录类型 000=普通 001=B+树节点指针 010= Infimum 011= Supremum 1xx=保留
next_recorder	16	页中下一条记录的相对位置

下面用一个具体事例来分析Compact行记录的内部结构：

```
mysql>create table mytest ( t1 varchar(10),t2 varchar(10),t3 char(10),t4
varchar(10)) engine=innodb charset=latin1 row_format=compact;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into mytest values ('a','bb','bb','ccc');
Query OK, 1 row affected (0.01 sec)

mysql> insert into mytest values ('d','ee','ee','fff');
Query OK, 1 row affected (0.00 sec)

mysql> insert into mytest values ('d',NULL,NULL,'fff');
Query OK, 1 row affected (0.00 sec)

mysql> select * from mytest\G;
*************************** 1. row ***************************
t1: a
t2: bb
t3: bb
t4: ccc
*************************** 2. row ***************************
t1: d
t2: ee
t3: ee
t4: fff
*************************** 3. row ***************************
t1: d
t2: NULL
t3: NULL
t4: fff
3 rows in set (0.00 sec)
```

我们创建了mytest表，有4个列，t1、t2、t4都为varchar变长字段类型，t3为固定长度类型char。接着我们插入了3条有代表性的数据，接着打开mytest.ibd（我启用了innodb_file_per_table，若你没有启用该选项，请打开默认的共享表空间文件ibdata1）。在

Windows下，可以选择用UltraEdit打开该二进制文件（在Linux环境下，使用hexdump -C -v mytest.ibd > mytest.txt即可），打开mytest.txt文件，找到如下内容：

```
0000c070  73 75 70 72 65 6d 75 6d  03 02 01 00 00 00 10 00  |supremum........|
0000c080  2c 00 00 00 2b 68 00 00  00 00 00 06 05 80 00 00  |,...+h..........|
0000c090  00 32 01 10 61 62 62 62  62 20 20 20 20 20 20 20  |.2..abbbb      |
0000c0a0  20 63 63 63 03 02 01 00  00 00 18 00 2b 00 00 00  | ccc........+...|
0000c0b0  2b 68 01 00 00 00 00 06  06 80 00 00 00 32 01 10  |+h...........2..|
0000c0c0  64 65 65 65 65 20 20 20  20 20 20 20 20 20 66 66 66  |deeeefff    |
0000c0d0  03 01 06 00 00 20 ff 98  00 00 00 2b 68 02 00 00  |..... .....+h...|
0000c0e0  00 00 06 07 80 00 00 00  32 01 10 64 66 66 66 00  |........2..dfff.|
```

该行记录从0000c078开始，若整理如下，相信你会有更好的理解：

```
03 02 01/*变长字段长度列表，逆序*/
00 /*NULL标志位，第一行没有NULL值*/
00 00 10 00 2c /*记录头信息，固定5字节长度*/
00 00 00 2b 68 00/*RowID 我们建的表没有主键，因此会有RowID*/
00 00 00 00 06 05/*TransactionID*/
80 00 00 00 32 01 10/*Roll Pointer*/
61/*列1数据'a'*/
62 62/*列2 'bb'*/
62 62 20 20 20 20 20 20 20 20/*列3数据'bb' */
63 63 63/*列4数据'ccc'*/
```

现在第一行数据就展现在我们眼前了。需要注意的是，变长字段长度列表是逆序存放的，03 02 01，而不是01 02 03。还需要注意的是InnoDB每行有隐藏列。同时可以看到，固定长度char字段在未填充满其长度时，会用0x20来进行填充。再来分析一下，记录头信息的最后4个字节代表next_recorder，0x6800代表下一个记录的偏移量，当前记录的位置+0x6800就是下一条记录的起始位置。所以InnoDB存储引擎在页内部是通过一种链表的结构来串联各个行记录的。

第二行我将不做整理，除了RowID不同外，它和第一行大同小异，有兴趣的读者可以用上面的方法自己试试。现在我们关注有NULL值的第三行：

```
03 01/*变长字段长度列表，逆序*/
06 /*NULL标志位，第三行有NULL值*/
00 00 20 ff 98/*记录头信息*/
00 00 00 2b 68 02/*RowID*/
00 00 00 00 06 07/*TransactionID*/
80 00 00 00 32 01 10/*Roll Pointer*/
```

```
64/*列1数据 'd'*/
66 66 66/*列4数据 'fff'*/
```

第三行有NULL值，因此NULL标志位不再是00而是06了，转换成二进制为00000110，为1的值即代表了第2列和第3列的数据为NULL，在其后存储列数据的部分，我们会发现没有存储NULL，只存储了第1列和第4列非NULL的值。这个例子很好地说明了：不管是char还是varchar类型，NULL值是不占用存储空间的。

4.4.2 Redundant行记录格式

Redundant是MySQL 5.0版本之前InnoDB的行记录存储方式，MySQL 5.0支持Redundant是为了向前兼容性。Redundant行记录以如下方式存储：

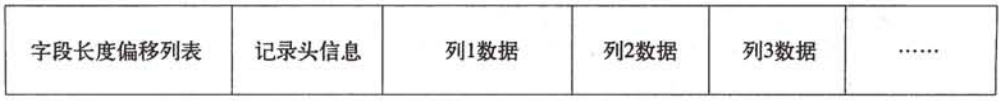


图4-3 Redundant行记录格式

从图4-3可以看到，不同于Compact行记录格式，Redundant行格式的首部是一个字段长度偏移列表，同样是按照列的顺序逆序放置的。当列的长度小于255字节，用1字节表示；若大于255个字节，用2个字节表示。第二个部分为记录头信息（record header），不同于Compact行格式，Redundant行格式固定占用6个字节（48位），每位的含义见表4-2。从表中可以看到，n_fields值代表一行中列的数量，占用10位，这也很好地解释了为什么MySQL一个行支持最多的列为1023。另一个需要注意的值为1byte_offs_flags，该值定义了偏移列表占用1个字节还是2个字节。最后的部分就是实际存储的每个列的数据了。

表4-2 Redundant页格式

名称	大小(bit)	描述
()	1	未知
()	1	未知
deleted_flag	1	该行是否已被删除
min_rec_flag	1	为1，如果该记录是预先被定义为最小的记录
n_owned	4	该记录拥有的记录数
heap_no	13	索引堆中该条记录的排序记录
n_fields	10	记录中列的数量
1byte_offs_flag	1	偏移列表为1个字节还是2个字节
next_record	16	页中下一条记录的相对位置

接着，我们创建一张和mytest内容完全一样、但行格式为Redundant的表mytest2。

```
mysql> create table mytest2 engine=innodb row_format=redundant
    -> as
    -> select * from mytest;
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> show table status like 'mytest2'\G;
*************************** 1. row ***************************
           Name: mytest2
         Engine: InnoDB
        Version: 10
     Row_format: Redundant
           Rows: 3
 Avg_row_length: 5461
    Data_length: 16384
Max_data_length: 0
   Index_length: 0
      Data_free: 0
 Auto_increment: NULL
    Create_time: 2009-03-18 15:49:42
    Update_time: NULL
     Check_time: NULL
      Collation: latin1_swedish_ci
       Checksum: NULL
 Create_options: row_format=REDUNDANT
        Comment:
1 row in set (0.00 sec)

mysql> select * from mytest2\G;
*************************** 1. row ***************************
t1: a
t2: bb
t3: bb
t4: ccc
*************************** 2. row ***************************
t1: d
t2: ee
t3: ee
t4: fff
*************************** 3. row ***************************
t1: d
t2: NULL
```

```
t3: NULL
t4: fff
3 rows in set (0.00 sec)
```

可以看到，现在row_format变为Redundant。同样，通过hexdump将表空间mytest2.ibd导出到文本文件mytest2.txt。打开文件，找到类似如下行：

```
0000c070  08 03 00 00 73 75 70 72  65 6d 75 6d 00 23 20 16  |....supremum.# .|
0000c080  14 13 0c 06 00 00 10 0f  00 ba 00 00 00 2b 68 0b  |.............+h.|
0000c090  00 00 00 00 06 53 80 00  00 00 32 01 10 61 62 62  |.....S....2..abb|
0000c0a0  62 62 20 20 20 20 20 20  20 20 63 63 63 23 20 16  |bb        ccc# .|
0000c0b0  14 13 0c 06 00 00 18 0f  00 ea 00 00 00 2b 68 0c  |.............+h.|
0000c0c0  00 00 00 00 06 53 80 00  00 00 32 01 1e 64 65 65  |.....S....2..dee|
0000c0d0  65 65 20 20 20 20 20 20  20 20 66 66 66 21 9e 94  |ee        fff!..|
0000c0e0  14 13 0c 06 00 00 20 0f  00 74 00 00 00 2b 68 0d  |...... ..t...+h.|
0000c0f0  00 00 00 00 06 53 80 00  00 00 32 01 2c 64 00 00  |.....S....2.,d..|
0000c100  00 00 00 00 00 00 00 00  66 66 66 00 00 00 00 00  |........fff.....|
```

整理可以得到如下内容：

```
23 20 16 14 13 0c 06/*长度偏移列表，逆序*/
00 00 10 0f  00 ba/* 记录头信息，固定6个字节*/
00 00 00 2b 68 0b/* RowID*/
00 00 00 00 06 53/*TransactionID*/
80 00 00 00 32 01 10/*Roll Point*/
61/*列1数据 'a'*/
62 62/*列2数据 'bb'*/
62 62 20 20 20 20 20 20 20 20/*列3数据 'bb' Char类型*/
63 63 63/*列4数据 'ccc'*/
```

23 20 16 14 13 0c 06，逆转为06,0c,13,14,16,20,23。分别代表第一列长度6，第二列长度6（6+6=0x0C），第三列长度为7（6+6+7=0x13），第四列长度1（6+6+7+1=0x14），第五列长度2（6+6+7+1+2=0x16），第六列长度10（6+6+7+1+2+10=0x20），第七列长度3（6+6+7+1+2+10+3=0x23）。

接着的记录头信息中应该注意48位中22～32位，为0000000111，表示表共有7个列（包含了隐藏的3列），接下去的33位为1，代表偏移列表为一个字节。

后面的信息就是实际每行存放的数据了，这与Compact行格式大致相同。请注意是大致相同，因为如果我们来看第三行，会发现对于NULL的处理两者是不同的。

```
21 9e 94 14 13 0c 06/*长度偏移列表，逆序*/
```

```
00 00 20 0f 00 74/*记录头信息，固定6个字节*/
00 00 00 2b 68 0d/*RowID*/
00 00 00 00 06 53/*TransactionID*/
80 00 00 00 32 01 10/*Roll Point*/
64/*列1数据'a'*/
00 00 00 00 00 00 00 00 00 00/*列3数据NULL*/
66 66 66/*列4数据'fff'*/
```

这里与之前Compact行格式有着很大的不同了，首先来看长度偏移列表，我们逆序排列后得到06 0c 13 14 94 9e 21，前4个值都很好理解，第5个NULL变为了94，接着第6个列char类型的NULL值为9e（94+10=0x9e），之后的21代表14+3=0x21。可以看到对于varchar的NULL值，Redundant行格式同样不占用任何存储空间，因而char类型的NULL值需要占用空间。

当前表mytest2的字符集为Latin1，每个字符最多只占用1个字节。若这里将表mytest2的字符集转换为utf8，第三列char固定长度类型就不再是只占用10个字节了，而是10×3=30个字节，Redundant行格式下char固定字符类型将会占据可能存放的最大值字节数。有兴趣的读者可以自行尝试。

4.4.3 行溢出数据

InnoDB存储引擎可以将一条记录中的某些数据存储在真正的数据页面之外，即作为行溢出数据。一般认为BLOB、LOB这类的大对象列类型的存储会把数据存放在数据页面之外。但是，这个理解有点偏差，BLOB可以不将数据放在溢出页面，而即使是varchar列数据类型，依然有可能存放为行溢出数据。我们先来对varchar类型进行研究。很多DBA喜欢MySQL的VARCHAR类型，因为相对于Oracle VARCHAR2最大存放4000个字节，SQL Server最大存放的8000个字节，MySQL的VARCHAR数据类型可以存放65 535个字节。但是，这是真的吗？真的可以存放65 535个字节吗？如果创建varchar长度为65 535的表，我们会得到下面所示的出错信息：

```
mysql> create table test ( a varchar(65535))charset=latin1 engine=innodb;
ERROR 1118 (42000): Row size too large. The maximum row size for the used table
type, not counting BLOBs, is 65535. You have to change some columns to TEXT or BLOBs
```

从出错消息可以看到，InnoDB存储引擎并不支持65 535长度的varchar。这是因为还有

别的开销，因此实际能存放的长度为65 532。下面的表创建就不会报错了：

```
mysql> create table test2 ( a varchar(65532)) charset=latin1 engine=innodb;
Query OK, 0 rows affected (0.15 sec)
```

需要注意的是，如果在做上述例子的时候并没有将sql_mode设为严格模式，则可能会出现可以建立表，但是会有一条警告信息：

```
mysql>  create table test ( a varchar(65535))engine=innodb;
Query OK, 0 rows affected, 1 warning (0.14 sec)

mysql> show warnings\G;
*************************** 1. row ***************************
  Level: Note
   Code: 1246
Message: Converting column 'a' from VARCHAR to TEXT
1 row in set (0.00 sec)
```

警告信息提示了，之所以这次可以创建，是因为MySQL自动将VARCHAR转换成了TEXT类型。如果我们看test的表结构，会发现MySQL自动将VARCHAR类型转换为了MEDIUMTEX类型：

```
mysql> show create table test\G;
*************************** 1. row ***************************
       Table: test
Create Table: CREATE TABLE 'test' (
  'a' mediumtext
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)
```

还需要注意的是，上述创建VARCHAR长度为65 532的表其字符类型是latin1的，如果换成GBK或者UTF-8，又会产生怎样的结果呢？

```
mysql>  create table test ( a varchar(65532))charset=gbk engine=innodb;
ERROR 1074 (42000): Column length too big for column 'a' (max = 32767); use BLOB
or TEXT instead
mysql>  create table test ( a varchar(65532))charset=utf8 engine=innodb;
ERROR 1074 (42000): Column length too big for column 'a' (max = 21845); use BLOB
or TEXT instead
```

这次即使创建列的VARCHAR长度为65 532也会报错，但是两次报错中对于max值的提示是不同的。因此我们应该理解VARCHAR（N）中，N指的是字符的长度，VARCHAR类

型最大支持65 535指的是65 535个字节。

此外，MySQL官方手册中定义的65 535长度是指所有VARCHAR列的长度总和，如果列的长度总和超出这个长度，依然无法创建，如下所示：

```
mysql> create table test2 ( a varchar(22000),b varchar(22000),c varchar(22000))
charset=latin1 engine=innodb;
ERROR 1118 (42000): Row size too large. The maximum row size for the used table
type, not counting BLOBs, is 65535. You have to change some columns to TEXT or BLOBs
```

3个列长度总和是66 000，因此InnoDB存储引擎再次报了同样的错误。即使我们能存放65 532个字节了，但是有没有想过，InnoDB存储引擎的页为16KB，即16 384个字节，怎么能存放65 532个字节呢？一般情况下，数据都是存放在B-tree Node的页类型中，但是当发生行溢处时，则这个存放行溢处的页类型为Uncompress BLOB Page。我们来看个例子：

```
mysql> create table t ( a varchar(65532));
Query OK, 0 rows affected (0.15 sec)

mysql> insert into t select repeat('a',65532);
Query OK, 1 row affected (0.08 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

这里创建了拥有一个长度为65 532的varchar类型表，接着用py_innodb_page_info工具看下面的表空间文件，看看页的类型有哪些。

```
[root@nineyou0-43 mytest]# py_innodb_page_info.py -v t.ibd
page offset 00000000, page type <File Space Header>
page offset 00000001, page type <Insert Buffer Bitmap>
page offset 00000002, page type <File Segment inode>
page offset 00000003, page type <B-tree Node>, page level <0000>
page offset 00000004, page type <Uncompressed BLOB Page>
page offset 00000005, page type <Uncompressed BLOB Page>
page offset 00000006, page type <Uncompressed BLOB Page>
page offset 00000007, page type <Uncompressed BLOB Page>
Total number of page: 8:
Insert Buffer Bitmap: 1
Uncompressed BLOB Page: 4
File Space Header: 1
B-tree Node: 1
File Segment inode: 1
```

可以看到一个B-tree Node页类型，另外有4个为Uncompressed BLOB Page，这些页中

才是真正存放了65 532个字节的数据。既然实际存放的数据都放到BLOB页中，那数据页中又存放了些什么东西呢？同样，通过之前的hexdump来读取表空间文件，从数据页c000开始查看：

```
0000c000  67 ce fc 0b 00 00 00 03  ff ff ff ff ff ff ff ff  |g...............|
0000c010  00 00 00 0a 6a d9 c0 89  45 bf 00 00 00 00 00 00  |....j...E.......|
0000c020  00 00 00 00 00 c3 00 02  03 a7 80 03 00 00 00 00  |................|
0000c030  00 80 00 05 00 00 00 01  00 00 00 00 00 00 00 00  |................|
0000c040  00 00 00 00 00 00 00 00  01 a1 00 00 00 c3 00 00  |................|
0000c050  00 02 00 f2 00 00 00 c3  00 00 00 02 00 32 01 00  |.............2..|
0000c060  02 00 1d 69 6e 66 69 6d  75 6d 00 02 00 0b 00 00  |...infimum......|
0000c070  73 75 70 72 65 6d 75 6d  14 c3 00 00 00 10 ff f0  |supremum........|
0000c080  00 00 00 b6 2b 00 00 00  00 51 4b 06 80 00 00 00  |....+....QK.....|
0000c090  2d 01 10 61 61 61 61 61  61 61 61 61 61 61 61 61  |-..aaaaaaaaaaaaa|
0000c0a0  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c0b0  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c0c0  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c0d0  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c0e0  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c0f0  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c100  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c110  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c120  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c130  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c140  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c150  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c160  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c170  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c180  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c190  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c1a0  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c1b0  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c1c0  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c1d0  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c1e0  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c1f0  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c200  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c210  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c220  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c230  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c240  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
```

```
0000c250  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c260  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c270  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c280  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c290  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c2a0  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c2b0  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c2c0  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c2d0  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c2e0  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c2f0  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c300  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c310  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c320  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c330  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c340  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c350  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c360  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c370  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c380  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  |aaaaaaaaaaaaaaaa|
0000c390  61 61 61 00 00 00 c3 00  00 00 04 00 00 00 26 00  |aaa..........&.|
0000c3a0  00 00 00 00 00 fc fc 00  00 00 00 00 00 00 00 00  |................|
```

可以看到，从0x0000c093到0x0000c392数据页面其实只保存了varchar（65 532）的前768个字节的前缀（prefix）数据（这里都是a），之后跟的是偏移量，指向行溢出页，也就是前面我们看到的Uncompressed BLOB Page。因此，对于行溢出数据，其存放方式如图4-4所示。

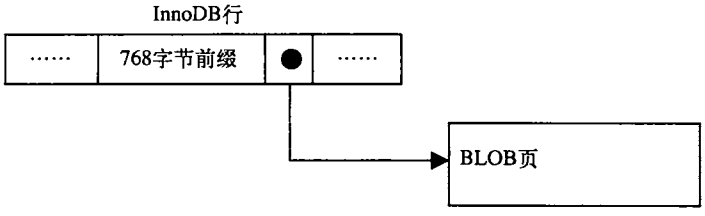


图4-4 行溢出

那多少长度VARCHAR是保存在数据页里的，多少长度开始又保存在BLOB页呢？我们来思考一下，InnoDB存储引擎表是索引组织的，即B+树的结构。因此每个页中至少应该有两个行记录（否则失去了B+树的意义，变成链表了）。因此如果当页中只能存放下一条记录，那么InnoDB存储引擎会自动将行数据存放到溢出页中。考虑下面表的一种情况：

```
mysql> create table t ( a varchar(9000));
Query OK, 0 rows affected (0.13 sec)

mysql> insert into t select repeat('a',9000);
Query OK, 1 row affected (0.04 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

表t的变长字段长度为9000，能放在一个页中，但是不能保证2条记录都能存放在一个页中，所以此时如果用py_innodb_page_info工具查看，可知是存放在BLOB页中：

```
[root@nineyou0-43 mytest]# py_innodb_page_info.py -v t.ibd
page offset 00000000, page type <File Space Header>
page offset 00000001, page type <Insert Buffer Bitmap>
page offset 00000002, page type <File Segment inode>
page offset 00000003, page type <B-tree Node>, page level <0000>
page offset 00000004, page type <Uncompressed BLOB Page>
page offset 00000005, page type <Uncompressed BLOB Page>
Total number of page: 6:
Insert Buffer Bitmap: 1
Uncompressed BLOB Page: 2
File Space Header: 1
B-tree Node: 1
File Segment inode: 1
```

但是如果可以在一个页中至少放入两行的数据，那varchar就不会存放到BLOB页中。经过试验我发现，这个阈值的长度为8098。如我们建立列为varchar（8098）的表，然后插入两条记录：

```
mysql> create table t ( a varchar(8098));
Query OK, 0 rows affected (0.12 sec)

mysql> insert into t select repeat('a',8098);
Query OK, 1 row affected (0.04 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> insert into t select repeat('a',8098);
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

接着用py_innodb_page_info工具对表空间t.ibd进行查看，可以发现此时的行记录都是存放在数据页中，而不是BLOB页了。如果熟悉Microsoft SQL Server数据库的DBA，可能会感觉InnoDB存储引擎对于varchar的管理和SQL Server中的VARCHAR（MAX）类似。

```
[root@nineyou0-43 mytest]# py_innodb_page_info.py -v t.ibd
page offset 00000000, page type <File Space Header>
page offset 00000001, page type <Insert Buffer Bitmap>
page offset 00000002, page type <File Segment inode>
page offset 00000003, page type <B-tree Node>, page level <0000>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
Total number of page: 6:
Freshly Allocated Page: 2
Insert Buffer Bitmap: 1
File Space Header: 1
B-tree Node: 1
File Segment inode: 1
```

对于溢出行的管理，同样是采用段的方式，即InnoDB存储引擎同Oracle一样有BLOB行溢出段。另一个问题是，对于TEXT或者BLOB的数据类型，我们总是以为它们是放在Uncompressed BLOB Page中的，其实这也是不准确的，放在数据页还是BLOB页同样和前面讨论的VARCHAR一样，至少保证一个页能存放两条记录，如：

```
mysql> create table t ( a blob);
Query OK, 0 rows affected (0.12 sec)

mysql> insert into t select repeat('a',8000);
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> insert into t select repeat('a',8000);
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> insert into t select repeat('a',8000);
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> insert into t select repeat('a',8000);
Query OK, 1 row affected (0.06 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

我们建立一个BLOB列的表，插入4行数据长度为8000的记录，如果用py_innodb_page_info工具对表空间t.ibd进行查看，会发现这些记录其实并没有保存在BLOB页中：

```
[root@nineyou0-43 mytest]# py_innodb_page_info.py -v t.ibd
```

```
page offset 00000000, page type <File Space Header>
page offset 00000001, page type <Insert Buffer Bitmap>
page offset 00000002, page type <File Segment inode>
page offset 00000003, page type <B-tree Node>, page level <0001>
page offset 00000004, page type <B-tree Node>, page level <0000>
page offset 00000005, page type <B-tree Node>, page level <0000>
page offset 00000006, page type <B-tree Node>, page level <0000>
page offset 00000000, page type <Freshly Allocated Page>
Total number of page: 8:
Freshly Allocated Page: 1
Insert Buffer Bitmap: 1
File Space Header: 1
B-tree Node: 4
File Segment inode: 1
```

当然，既然我们使用了BLOB列类型，一般情况下我们不可能存放长度这么小的数据，因此对于大多数的情况，BLOB的行数据还是会发生行溢出，实际数据保存在BLOB页中，数据页只保存数据的前768个字节。

4.4.4 Compressed与Dynamic行记录格式

InnoDB Plugin引入了新的文件格式（file format，可以理解为新的页格式），对于以前支持的Compact和Redundant格式将其称为Antelope文件格式，新的文件格式称为Barracuda如图4-5所示。Barracuda文件格式下拥有两种新的行记录格式Compressed和Dynamic两种。新的两种格式对于存放BLOB的数据采用了完全的行溢出的方式，在数据页中只存放20个字节的指针，实际的数据都存放在BLOB Page中，而之前的Compact和Redundant两种格式会存放768个前缀字节。

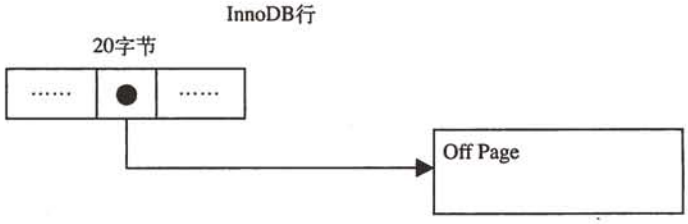


图4-5 Barracuda文件格式的溢出行

Compressed行记录格式的另一个功能就是，存储在其中的行数据会以zlib的算法进行压缩，因此对于BLOB、TEXT、VARCHAR这类大长度类型的数据能进行非常有效的存储。

4.4.5 Char的行结构存储

通常的理解VARCHAR是存储变长长度的字符类型，CHAR是存储定长长度的字符类型。前面的小结我们已经分析了行结构的内部存储，可以发现每行的变长字段长度的列表都没有存储对于CHAR类型的长度。但是有没有注意到，我给出的两个例子中字符集都是单字节的latin1格式。从MySQL 4.1开始，CHR（N）中的N指的是字符的长度，而不是之前版本的字节长度。那也就是说，在不同的字符集下，CHAR的内部存储的不是定长的数据。我们来看下面的这个情况：

```
mysql> create table j ( a char(2))charset=gbk;
Query OK, 0 rows affected (0.11 sec)

mysql> insert into j select 'ab';
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> set names gbk;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into j select '我们';
Query OK, 1 row affected (0.04 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> insert into j select 'a';
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

j表的字符集是GBK的，我们分别插入了两个字符的数据'ab'和'我们'，查看所占字节可得如下结果：

```
mysql> select a,char_length(a),length(a) from j\G;
*************************** 1. row ***************************
             a: ab
char_length(a): 2
     length(a): 2
*************************** 2. row ***************************
             a: 我们
char_length(a): 2
     length(a): 4
*************************** 3. row ***************************
```

```
            a: a
char_length(a): 1
     length(a): 1
3 rows in set (0.00 sec)
```

通过不同的字符串长度函数可以看到，前两个记录'ab'和'我们'字符串的长度都是2，但是内部存储上'ab'占用两个字节，而'我们'占用4个字节。如果看内部十六进制的存储，可以看到：

```
mysql> select a,hex(a) from j\G;
*************************** 1. row ***************************
     a: ab
hex(a): 6162
*************************** 2. row ***************************
     a: 我们
hex(a): CED2C3C7
*************************** 3. row ***************************
     a: a
hex(a): 61
3 rows in set (0.00 sec)
```

对于字符串'ab'的存储内部为0x6162，而'我们'是0xCED2C3C7，这就可以很明显地看出区别了。因此对于多字节的字符编码CHAR类型，不再代表是固定长度的字符串了，比如UTF-8下CHAR（10）最小可以存储10个字节的字符，而最大可以存储30个字节的字符。所以，对于多字节字符编码的CHAR数据类型的存储，InnoDB存储引擎在内部将其视为是变长的字符，这就表示了，在每行变长长度列表中会记录CHAR数据类型的长度。通过hexdump工具我们来看j.ibd文件的内部：

```
0000c070  73 75 70 72 65 6d 75 6d  02 00 00 00 10 00 1c 00  |supremum........|
0000c080  00 00 b6 2b 2b 00 00 00  51 52 da 80 00 00 00 2d  |...++...QR.....-|
0000c090  01 10 61 62 04 00 00 00  18 ff d5 00 00 00 b6 2b  |..ab...........+|
0000c0a0  2c 00 00 00 51 52 db 80  00 00 00 2d 01 10 ce d2  |,...QR.....-....|
0000c0b0  c3 c7 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

整理后可以得到如下结果：

```
# 第一行记录
02                      /* 变长字段长度2，char视作变长类型 */
00                      /* NULL标志位 */
00 00 10 00 1c          /* 记录头信息 */
00 00 00 b6 2b 2b       /* RowID */
```

```
00 00 00 51 52 da          /* TransactionID */
80 00 00 00 2d 01 10       /* Roll Point */
61 62                      /* 字符'ab' */

# 第二行记录
04                         /* 变长字段长度4，char视作变长类型 */
00                         /* NULL标志位 */
00 00 18 ff d5             /* 记录头信息 */
00 00 00 b6 2b 2c          /* RowID */
00 00 00 51 52 db          /* TransactionID */
80 00 00 00 2d 01 10       /* Roll Point */
c3 d2 c3 c7                /* 字符'我们' */

# 第三行记录
02                         /* 变长字段长度2，char视作变长类型 */
00                         /* NULL标志位 */
00 00 20 ff b7             /* 记录头信息 */
00 00 00 b6 2b 2d          /* RowID */
00 00 00 51 53 17          /* TransactionID */
80 00 00 00 2d 01 10       /* Roll Point */
61 20                      /* 字符'a' */
```

现在很清楚地表明了，在InnoDB存储引擎内部对于CHAR类型在多字节字符集类型的存储了，CHAR很明确地被视为了变长类型，对于未能占满长度的字符还是填充0x20。内部对于字符的存储和我们用hex函数看到的也是一致的。我们可以说，在多字节字符集的情况下，CHAR和VARCHAR的行存储基本是没有区别的。

4.5 InnoDB数据页结构

通过前面几个小节的介绍，我们已经知道页是InnoDB存储引擎管理数据库的最小磁盘单位。页类型为B-tree node的页，存放的即是表中行的实际数据了。在这一节中，我们将从底层具体地介绍InnoDB数据页的内部存储结构。

注意： InnoDB公司本身并没有详细介绍其页结构的实现，MySQL的官方手册中也基本没有提及InnoDB的页结构。该小节是我通过阅读源代码来了解InnoDB的页结构，同时参考了Peter对于InnoDB页结构的分析（Peter在写这部分的时候可能时间

久远，在其之后InnoDB引入了Compact格式，因此页结构有所改动）。因此可能出现对页结构分析错误的情况，如有错误，希望可以指出。

InnoDB数据页由以下七个部分组成，如图4-6所示：

- File Header（文件头）。
- Page Header （页头）。
- Infimun + Supremum Records。
- User Records （用户记录，即行记录）。
- Free Space （空闲空间）。
- Page Directory （页目录）。
- File Trailer （文件结尾信息）。

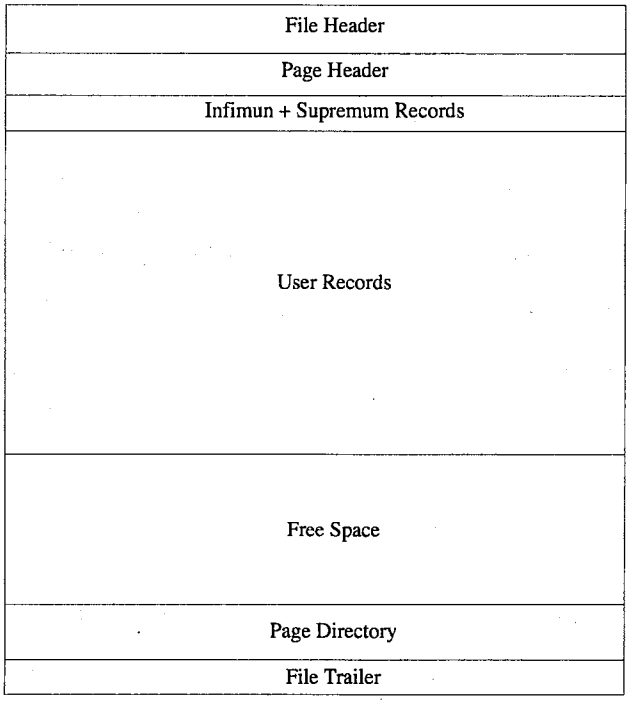


图4-6 InnoDB存储引擎数据页结构

File Header、Page Header、File Trailer的大小是固定的，用来标示该页的一些信息，如Checksum、数据所在索引层等。其余部分为实际的行记录存储空间，因此大小是动态的。在接下来的各小节中，我们将具体分析各组成部分的作用。

4.5.1 File Header

File Header用来记录页的一些头信息，由如下8个部分组成，共占用38个字节，如表4-3所示：

表4-3 File Header组成部分

名称	大小(字节)
FIL_PAGE_SPACE_OR_CHKSUM	4
FIL_PAGE_OFFSET	4
FIL_PAGE_PREV	4
FIL_PAGE_NEXT	4
FIL_PAGE_LSN	8
FIL_PAGE_TYPE	2
FIL_PAGE_FILE_FLUSH_LSN	8
FIL_PAGE_ARCH_LOG_NO_OR_SPACE_ID	4

- FIL_PAGE_SPACE_OR_CHKSUM：当MySQL版本小于MySQL-4.0.14，该值代表该页属于哪个表空间，因为如果我们没有开启innodb_file_per_table，共享表空间中可能存放了许多页，并且这些页属于不同的表空间。之后版本的MySQL，该值代表页的checksum值（一种新的checksum值）。
- FIL_PAGE_OFFSET：表空间中页的偏移值。
- FIL_PAGE_PREV，FIL_PAGE_NEXT：当前页的上一个页以及下一个页。B+ Tree特性决定了叶子节点必须是双向列表。
- FIL_PAGE_LSN：该值代表该页最后被修改的日志序列位置LSN（Log Sequence Number）。
- FIL_PAGE_TYPE：页的类型。通常有以下几种，见表4-4。请记住0x45BF，该值代表了存放的数据页。

表4-4 Page类型

名称	十六进制	解释
FIL_PAGE_INDEX	0x45BF	B+树叶节点
FIL_PAGE_UNDO_LOG	0x0002	Undo Log页
FIL_PAGE_INODE	0x0003	索引节点
FIL_PAGE_IBUF_FREE_LIST	0x0004	Insert Buffer空闲列表
FIL_PAGE_TYPE_ALLOCATED	0x0000	该页为最新分配
FIL_PAGE_IBUF_BITMAP	0x0005	Insert Buffer位图

（续）

名称	十六进制	解释
FIL_PAGE_TYPE_SYS	0x0006	系统页
FIL_PAGE_TYPE_TRX_SYS	0x0007	事务系统数据
FIL_PAGE_TYPE_FSP_HDR	0x0008	File Space Header
FIL_PAGE_TYPE_XDES	0x0009	扩展描述页
FIL_PAGE_TYPE_BLOB	0x000A	BLOB页

- FIL_PAGE_FILE_FLUSH_LSN：该值仅在数据文件中的一个页中定义，代表文件至少被更新到了该LSN值。
- FIL_PAGE_ARCH_LOG_NO_OR_SPACE_ID：从MySQL 4.1开始，该值代表页属于哪个表空间。

4.5.2 Page Header

接着File Header部分的是Page Header，用来记录数据页的状态信息，由以下14个部分组成，共占用56个字节。见表4-5。

表4-5 Page Header组成部分

名称	大小(字节)
PAGE_N_DIR_SLOTS	2
PAGE_HEAP_TOP	2
PAGE_N_HEAP	2
PAGE_FREE	2
PAGE_GARBAGE	2
PAGE_LAST_INSERT	2
PAGE_DIRECTION	2
PAGE_N_DIRECTION	2
PAGE_N_RECS	2
PAGE_MAX_TRX_ID	8
PAGE_LEVEL	2
PAGE_INDEX_ID	8
PAGE_BTR_SEG_LEAF	10
PAGE_BTR_SEG_TOP	10

- PAGE_N_DIR_SLOTS：在Page Directory（页目录）中的Slot（槽）数。Page Directory会在4.5.5节中介绍。
- PAGE_HEAP_TOP：堆中第一个记录的指针。
- PAGE_N_HEAP：堆中的记录数。

- PAGE_FREE：指向空闲列表的首指针。
- PAGE_GARBAGE：已删除记录的字节数，即行记录结构中，delete flag为1的记录大小的总数。
- PAGE_LAST_INSERT：最后插入记录的位置。
- PAGE_DIRECTION：最后插入的方向。可能的取值为PAGE_LEFT (0x01)，PAGE_RIGHT (0x02)，PAGE_SAME_REC (0x03)，PAGE_SAME_PAGE (0x04)，PAGE_NO_DIRECTION (0x05)。
- PAGE_N_DIRECTION：一个方向连续插入记录的数量。
- PAGE_N_RECS：该页中记录的数量。
- PAGE_MAX_TRX_ID：修改当前页的最大事务ID，注意该值仅在Secondary Index定义。
- PAGE_LEVEL：当前页在索引树中的位置，0x00代表叶节点。
- PAGE_INDEX_ID：当前页属于哪个索引ID。
- PAGE_BTR_SEG_LEAF：B+树的叶节点中，文件段的首指针位置。注意该值仅在B+树的Root页中定义。
- PAGE_BTR_SEG_TOP：B+树的非叶节点中，文件段的首指针位置。注意该值仅在B+树的Root页中定义。

4.5.3 Infimum和Supremum记录

在InnoDB存储引擎中，每个数据页中有两个虚拟的行记录，用来限定记录的边界。Infimum记录是比该页中任何主键值都要小的值，Supremum指比任何可能大的值还要大的值。这两个值在页创建时被建立，并且在任何情况下不会被删除。在Compact行格式和Redundant行格式下，两者占用的字节数各不相同。图4-7显示了Infimum和Supremum Records。

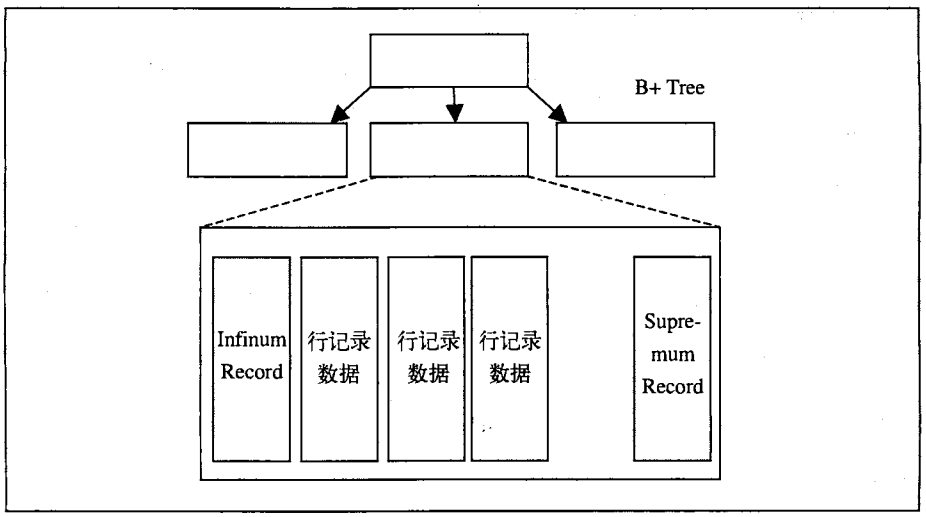


图4-7 Infinum和Supremum记录

4.5.4 User Records与FreeSpace

User Records就是之前我们讨论过的部分，即实际存储行记录的内容。再次强调，InnoDB存储引擎表总是B+树索引组织的。

很明显，Free Space指的就是空闲空间，同样也是个链表数据结构。当一条记录被删除后，该空间会被加入空闲链表中。

4.5.5 Page Directory

Page Directory（页目录）中存放了记录的相对位置（注意，这里存放的是页相对位置，而不是偏移量），有些时候这些记录指针称为Slots（槽）或者目录槽（Directory Slots）。与其他数据库系统不同的是，InnoDB并不是每个记录拥有一个槽，InnoDB存储引擎的槽是一个稀疏目录（sparse directory），即一个槽中可能属于（belong to）多个记录，最少属于4条记录，最多属于8条记录。

Slots中记录按照键顺序存放，这样可以利用二叉查找迅速找到记录的指针。假设我们有（'i'，'d'，'c'，'b'，'e'，'g'，'l'，'h'，'f'，'j'，'k'，'a'），同时假设一个槽中包含4条记录，则Slots中的记录可能是（'a'，'e'，'i'）。

由于InnoDB存储引擎中Slots是稀疏目录，二叉查找的结果只是一个粗略的结果，所以

InnoDB必须通过recorder header中的next_record来继续查找相关记录。同时，slots很好地解释了recorder header中的n_owned值的含义，即还有多少记录需要查找，因为这些记录并不包括在slots中。

需要牢记的是，B+树索引本身并不能找到具体的一条记录，B+树索引能找到只是该记录所在的页。数据库把页载入内存，然后通过Page Directory再进行二叉查找。只不过二叉查找的时间复杂度很低，同时内存中的查找很快，因此通常我们忽略了这部分查找所用的时间。

4.5.6 File Trailer

为了保证页能够完整地写入磁盘（如可能发生的写入过程中磁盘损坏、机器宕机等原因），InnoDB存储引擎的页中设置了File Trailer部分。File Trailer只有一个FIL_PAGE_END_LSN部分，占用8个字节。前4个字节代表该页的checksum值，最后4个字节和File Header中的FIL_PAGE_LSN相同。通过这两个值来和File Header中的FIL_PAGE_SPACE_OR_CHKSUM和FIL_PAGE_LSN值进行比较，看是否一致（checksum的比较需要通过InnoDB的checksum函数来进行比较，不是简单的等值比较），以此来保证页的完整性（not corrupted）。

4.5.7 InnoDB数据页结构示例分析

通过前面各小节的介绍，相信读者对于InnoDB存储引擎的数据页已经有了一个大致的了解。本节我将通过一个具体的表，结合前面的介绍来具体分析一个数据页的内部存储结构。首先我们建立一张表，并导入一定量的数据：

```
mysql> drop table if exists t;
Query OK, 0 rows affected (0.04 sec)

mysql> create table t ( a int unsigned not null auto_increment, b char(10),
primary key (a))ENGINE=InnoDB CHARSET=UTF-8;
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter $$
mysql> create procedure load_t (count int unsigned)
```

```
    -> begin
    -> set @c = 0;
    -> while @c < count do
    -> insert into t select null,repeat(char(97+rand()*26),10);
    -> set @c=@c+1;
    -> end while;
    -> end;
    -> $$
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
mysql> call load_t(100);
Query OK, 0 rows affected (0.60 sec)

mysql> select * from t limit 10;
+----+---------------+
| a  | b             |
+----+---------------+
|  1 | ddddddddd |
|  2 | hhhhhhhhh |
|  3 | bbbbbbbbb |
|  4 | iiiiiiiii |
|  5 | nnnnnnnnn |
|  6 | qqqqqqqqq |
|  7 | ooooooooo |
|  8 | yyyyyyyyy |
|  9 | yyyyyyyyy |
| 10 | vvvvvvvvv |
+----+---------------+
10 rows in set (0.00 sec)
```

接着我们用工具py_innodb_page_info来分析t.ibd，得到如下内容:

```
[root@nineyou0-43 mytest]# py_innodb_page_info.py -v t.ibd
page offset 00000000, page type <File Space Header>
page offset 00000001, page type <Insert Buffer Bitmap>
page offset 00000002, page type <File Segment inode>
page offset 00000003, page type <B-tree Node>, page level <0000>
page offset 00000000, page type <Freshly Allocated Page>
page offset 00000000, page type <Freshly Allocated Page>
Total number of page: 6:
Freshly Allocated Page: 2
Insert Buffer Bitmap: 1
File Space Header: 1
```

```
B-tree Node: 1
File Segment inode: 1
```

看到第四个页（page offset 3）是数据页，然后通过hexdump来分析t.ibd文件，打开整理得到的十六进制文件，数据页在0x0000c000（16K*3=0xc000）处开始，得到以下内容：

```
0000c000  52 1b 24 00 00 00 00 03  ff ff ff ff ff ff ff ff  |R.$.............|
0000c010  00 00 00 0a 6a e0 ac 93  45 bf 00 00 00 00 00 00  |....j...E.......|
0000c020  00 00 00 00 00 dc 00 1a  0d c0 80 66 00 00 00 00  |...........f....|
0000c030  0d a5 00 02 00 63 00 64  00 00 00 00 00 00 00 00  |.....c.d........|
0000c040  00 00 00 00 00 00 00 00  01 ba 00 00 00 dc 00 00  |................|
0000c050  00 02 00 f2 00 00 00 dc  00 00 00 02 00 32 01 00  |.............2..|
0000c060  02 00 1c 69 6e 66 69 6d  75 6d 00 05 00 0b 00 00  |...infimum......|
0000c070  73 75 70 72 65 6d 75 6d  0a 00 00 00 10 00 22 00  |supremum......".|
0000c080  00 00 01 00 00 00 51 6d  eb 80 00 00 00 2d 01 10  |......Qm.....-..|
0000c090  64 64 64 64 64 64 64 64  64 64 0a 00 00 00 18 00  |dddddddddd......|
0000c0a0  22 00 00 00 02 00 00 00  51 6d ec 80 00 00 00 2d  |".......Qm.....-|
0000c0b0  01 10 68 68 68 68 68 68  68 68 68 68 0a 00 00 00  |..hhhhhhhhhh....|
0000c0c0  20 00 22 00 00 00 03 00  00 00 51 6d ed 80 00 00  | ."........Qm....|
0000c0d0  00 2d 01 10 62 62 62 62  62 62 62 62 62 62 0a 00  |.-..bbbbbbbbbb..|
0000c0e0  04 00 28 00 22 00 00 00  04 00 00 00 51 6d ee 80  |..(."........Qm..|
0000c0f0  00 00 00 2d 01 10 69 69  69 69 69 69 69 69 69 69  |...-..iiiiiiiiii|
0000c100  0a 00 00 00 30 00 22 00  00 00 05 00 00 00 51 6d  |....0."........Qm|
0000c110  ef 80 00 00 00 2d 01 10  6e 6e 6e 6e 6e 6e 6e 6e  |.....-..nnnnnnnn|
0000c120  6e 6e 0a 00 00 00 38 00  22 00 00 00 06 00 00 00  |nn....8."........|
0000c130  51 6d f0 80 00 00 00 2d  01 10 71 71 71 71 71 71  |Qm.....-..qqqqqq|
0000c140  71 71 71 71 0a 00 00 00  40 00 22 00 00 00 07 00  |qqqq....@."......|
0000c150  00 00 51 6d f1 80 00 00  00 2d 01 10 6f 6f 6f 6f  |..Qm.....-..oooo|
0000c160  6f 6f 6f 6f 6f 6f 0a 00  04 00 48 00 22 00 00 00  |oooooo....H."...|
0000c170  08 00 00 00 51 6d f2 80  00 00 00 2d 01 10 79 79  |....Qm.....-..yy|
0000c180  79 79 79 79 79 79 79 79  0a 00 00 00 50 00 22 00  |yyyyyyyy....P.".|
0000c190  00 00 09 00 00 00 51 6d  f3 80 00 00 00 2d 01 10  |......Qm.....-..|
0000c1a0  79 79 79 79 79 79 79 79  79 79 0a 00 00 00 58 00  |yyyyyyyyyy....X.|
0000c1b0  22 00 00 00 0a 00 00 00  51 6d f4 80 00 00 00 2d  |".......Qm.....-|
0000c1c0  01 10 76 76 76 76 76 76  76 76 76 76 0a 00 00 00  |..vvvvvvvvvv....|
0000c1d0  60 00 22 00 00 00 0b 00  00 00 51 6d f5 80 00 00  |'."........Qm....|
0000c1e0  00 2d 01 10 6b 6b 6b 6b  6b 6b 6b 6b 6b 6b 0a 00  |.-..kkkkkkkkkk..|
0000c1f0  04 00 68 00 22 00 00 00  0c 00 00 00 51 6d f6 80  |..h."........Qm..|
......
0000ffc0  00 00 00 00 00 70 0d 1d  0c 95 0c 0d 0b 85 0a fd  |.....p..........|
0000ffd0  0a 75 09 ed 09 65 08 dd  08 55 07 cd 07 45 06 bd  |.u...e...U...E..|
0000ffe0  06 35 05 ad 05 25 04 9d  04 15 03 8d 03 05 02 7d  |.5...%.........}|
0000fff0  01 f5 01 6d 00 e5 00 63  95 ae 5d 39 6a e0 ac 93  |...m...c..]9j...|
```

先来分析前面File Header的38个字节：

- 52 1b 24 00 数据页的Checksum值。
- 00 00 00 03 页的偏移量，从0开始。
- ff ff ff ff前一个页，因为只有当前一个数据页，所以这里为0xffffffff。
- ff ff ff ff 下一个页，因为只有当前一个数据页，所以这里为0xffffffff。
- 00 00 00 0a 6a e0 ac 93 页的LSN。
- 45 bf 页类型，0x45bf代表数据页。
- 00 00 00 00 00 00 00 这里暂时不管该值。
- 00 00 00 dc 表空间的SPACE ID。

先不急着看下面的Page Header部分，我们来看File Trailer部分。因为File Trailer通过比较File Header部分来保证页写入的完整性。

```
95 ae 5d 39 Checksum值，该值通过checksum函数和File Header部分的checksum值进行比较。
6a e0 ac 93 注意到该值和File Header部分页的LSN后4个值相等。
```

接着我们来分析56个字节的Page Header部分，对于数据页而言，Page Header部分保存了该页中行记录的大量细节信息。分析后可得：

```
Page Header (56 bytes):
PAGE_N_DIR_SLOTS = 0x001a
PAGE_HEAP_TOP=0x0dc0
PAGE_N_HEAP=0x8066
PAGE_FREE=0x0000
PAGE_GARBAGE=0x0000
PAGE_LAST_INSERT=0x0da5
PAGE_DIRECTION=0x0002
PAGE_N_DIRECTION=0x0063
PAGE_N_RECS= 0x0064
PAGE_MAX_TRX_ID=0x0000000000000000
PAGE_LEVEL=00 00
PAGE_INDEX_ID=0x00000000000001ba
PAGE_BTR_SEG_LEAF=0x000000dc0000000200f2
PAGE_BTR_SEG_TOP =0x000000dc000000020032
```

PAGE_N_DIR_SLOTS=0x001a，代表Page Directory有26个槽，每个槽占用2个字节，我们可以从0x0000ffc4到0x0000fff7找到如下内容。

```
0000ffc0  00 00 00 00 00 70 0d 1d  0c 95 0c 0d 0b 85 0a fd  |.....p..........|
0000ffd0  0a 75 09 ed 09 65 08 dd  08 55 07 cd 07 45 06 bd  |.u...e...U...E..|
0000ffe0  06 35 05 ad 05 25 04 9d  04 15 03 8d 03 05 02 7d  |.5...%.........}|
0000fff0  01 f5 01 6d 00 e5 00 63  95 ae 5d 39 6a e0 ac 93  |...m...c..]9j...|
```

PAGE_HEAP_TOP=0x0dc0代表空闲空间开始位置的偏移量，即0xc000+0x0dc0=0xcdc0处开始，我们观察这个位置的情况，可以发现这的确是最后一行的结束，接下去的部分都是空闲空间了:

```
0000cdb0  00 00 00 2d 01 10 70 70  70 70 70 70 70 70 70 70  |...-..pppppppppp|
0000cdc0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0000cdd0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0000cde0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

PAGE_N_HEAP=0x8066，当行记录格式为Compact时，初始值为0x0802，当行格式为Redundant时，初始值是2。其实这些值表示页初始时就已经有Infinimun和Supremum的伪记录行，0x8066-0x8002=0x64，代表该页中实际的记录有100条记录。

PAGE_FREE=0x0000代表删除的记录数，因为这里我们没有进行过删除操作，所以这里的值为0。

PAGE_GARBAGE=0x0000，代表删除的记录字节为0，同样因为我们没有进行过删除操作，所以这里的值依然为0。

PAGE_LAST_INSERT=0x0da5，表示页最后插入的位置的偏移量，即最后的插入位置应该在0xc0000+0x0da5=0xcda5，查看该位置：

```
0000cda0  00 03 28 f2 cb 00 00 00  64 00 00 00 51 6e 4e 80  |..(.....d...QnN.|
0000cdb0  00 00 00 2d 01 10 70 70  70 70 70 70 70 70 70 70  |...-..pppppppppp|
0000cdc0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

可以看到，最后这的确是最后插入a列值为100的行记录，但是这次直接指向了行记录的内容，而不是指向行记录的变长字段长度的列表位置。

PAGE_DIRECTION=0x0002，因为我们是通过自增长的方式进行行记录的插入，所以PAGE_DIRECTION的方向是向右。

PAGE_N_DIRECTION=0x0063，表示一个方向连续插入记录的数量，因为我们是以自增长的方式插入了100条记录，因此该值为99。

PAGE_N_RECS= 0x0064，表示该页的行记录数为100，注意该值与PAGE_N_HEAP的

比较，PAGE_N_HEAP包含两个伪行记录，并且是通过有符号的方式记录的，因此值为0x8066。

PAGE_LEVEL=0x00，代表该页为叶子节点。因为数据量目前较少，因此当前B+树索引只有一层。B+数叶子层总是为0x00。

PAGE_INDEX_ID=0x00000000000001ba，索引ID。

上面就是数据页的Page Header部分了，接下去就是存放的行记录了，前面提到过InnoDB存储引擎有2个伪记录行，用来限定行记录的边界，我们接着往下看：

```
0000c050  00 02 00 f2 00 00 00 dc  00 00 00 02 00 32 01 00  |.............2..|
0000c060  02 00 1c 69 6e 66 69 6d  75 6d 00 05 00 0b 00 00  |...infimum......|
0000c070  73 75 70 72 65 6d 75 6d  0a 00 00 00 10 00 22 00  |supremum......".|
```

观察0xc05E到0xc077，这里存放的就是这两个伪行记录，InnoDB存储引擎设置伪行只有一个列，且类型是Char（8）。伪行记录的读取方式和一般的行记录并无不同，我们整理后可以得到如下的结果：

```
# Infimum伪行记录
01 00 02 00 1c          /* recorder header */
69 6e 66 69 6d 75 6d 00  /* 只有一个列的伪行记录，记录内容就是Infimum(多了一个0x00字节)
*/
# Supremum伪行记录
05 00 0b 00 00         /* recorder header */
73 75 70 72 65 6d 75 6d /*只有一个列的伪行记录，记录内容就是Supremum */
```

我们来分析infimum行记录的recorder header部分，最后2个字节位00 1c表示下一个记录的位置的偏移量，即当前行记录内容的位置0xc063+0x001c，得到0xc07f。0xc07f应该很熟悉了，我们前面的分析的行记录结构都是从这个位置开始。我们来看一下：

```
0000c070  73 75 70 72 65 6d 75 6d  0a 00 00 00 10 00 22 00  |supremum......".|
0000c080  00 00 01 00 00 00 51 6d  eb 80 00 00 00 2d 01 10  |......Qm.....-..|
0000c090  64 64 64 64 64 64 64 64  64 64 0a 00 00 00 18 00  |dddddddddd......|
0000c0a0  22 00 00 00 02 00 00 00  51 6d ec 80 00 00 00 2d  |".......Qm.....-|
```

可以看到这就是第一条实际行记录内容的位置了，如果整理后可以得到：

```
/* 第一条行记录 */
00 00 00 01 /* 因为我们建表时设定了主键，这里ROWID即位列a的值1 */
00 00 00 51 6d eb /* Transaction ID */
80 00 00 00 2d 01 10 /* Roll Pointer */
```

```
64 64 64 64 64 64 64 64 64 64 /* b列的值'aaaaaaaaaa' */
```

这和我们查表得到的数据是一致的：

```
mysql> select a,b,hex(b) from t order by a limit 1;
+---+------------+--------------------------------+
| a | b          | hex(b)                         |
+---+------------+--------------------------------+
| 1 | ddddddddd | 64646464646464646464 |
+---+------------+--------------------------------+
1 row in set (0.00 sec)
```

通过recorder header最后2个字节记录的下一行记录的偏移量，我们就可以得到该页中所有的行记录；通过page header的PAGE_PREV，PAGE_NEXT就可以知道上一个页和下个页的位置。这样，我们就能读到整张表所有的行记录数据。

最后我们来分析Page Directory，前面我们已经提到了从0x0000ffc4到0x0000fff7是当前页的Page Directory，如下：

```
0000ffc0  00 00 00 00 00 70 0d 1d  0c 95 0c 0d 0b 85 0a fd  |.....p..........|
0000ffd0  0a 75 09 ed 09 65 08 dd  08 55 07 cd 07 45 06 bd  |.u...e...U...E..|
0000ffe0  06 35 05 ad 05 25 04 9d  04 15 03 8d 03 05 02 7d  |.5...%.........}|
0000fff0  01 f5 01 6d 00 e5 00 63  95 ae 5d 39 6a e0 ac 93  |...m...c..]9j...|
```

需要注意的是，Page Directory是逆序存放的，每个槽2个字节。因此我们可以看到：00 63是最初行的相对位置，即0xc063；0070就是最后一行记录的相对位置，即0xc070。我们发现，这就是前面我们分析的infimum和supremum的伪行记录。Page Directory槽中的数据都是按照主键的顺序存放，因此找具体的行就需要通过部分进行。前面已经提到，InnoDB存储引擎的槽是稀疏的，还需通过recorder header的n_owned进行进一步的判断。如，我们要找主键a为5的记录，通过二叉查找Page Directory的槽，我们找到记录的相对位置在00 e5处，找到行记录的实际位置0xc0e5：

```
0000c0e0  04 00 28 00 22 00 00 00  04 00 00 00 51 6d ee 80  |..(."........Qm..|
0000c0f0  00 00 00 2d 01 10 69 69  69 69 69 69 69 69 69 69  |...-..iiiiiiiiii|
0000c100  0a 00 00 00 30 00 22 00  00 00 05 00 00 00 51 6d  |....0."........Qm|
0000c110  ef 80 00 00 00 2d 01 10  6e 6e 6e 6e 6e 6e 6e 6e  |.....-..nnnnnnnn|
0000c120  6e 6e 0a 00 00 00 38 00  22 00 00 00 06 00 00 00  |nn....8."........|
0000c130  51 6d f0 80 00 00 00 2d  01 10 71 71 71 71 71 71  |Qm.....-..qqqqqq|
0000c140  71 71 71 71 0a 00 00 00  40 00 22 00 00 00 07 00  |qqqq....@."......|
```

可以看到第一行的记录是4不是我们要找的6，但是我们看前面的5个字节的record

header，04 00 28 00 22，找到4~8位表示n_owned值的部分，该值为4，表示该记录有4个记录，因此还需要进一步查找。通过recorder 和ader最后2个字节的偏移量0x0022，找到下一条记录的位置0xc107，这才是我们要找的主键为5的记录。

这一节中，我们通过一个实例深入浅出地分析了页中各信息的存储，相信会对今后更好地理解InnoDB存储引擎和优化数据库带来好处。

4.6 Named File Formats

随着InnoDB存储引擎的发展，新的页数据结构有时用来支持新的功能特性。比如前面提到的InnoDB Plugin，提供了新的页数据结构来支持表压缩功能，完全溢出的（Off page）大变长字符类型字段的存储。这些新的页数据结构和之前版本的页并不兼容。因此从InnoDB Plugin版本开始，InnoDB存储引擎通过Named File Formats机制来解决不同版本下页结构兼容性的问题。

InnoDB Plugin将之前版本的文件格式（file format）定义为Antelope，将这个版本支持的文件格式定义为Barracuda。新的文件格式总是包含于之前的版本的页格式。图4-8显示了Barracuda文件格式和Antelope文件格式之间的关系，Antelope文件格式有Compact和Redudant的行格式，Barracuda文件格式即包括了Antelope所有的文件格式，另外新加入了前面我们已经提到过的Compressed何Dynamic行格式。

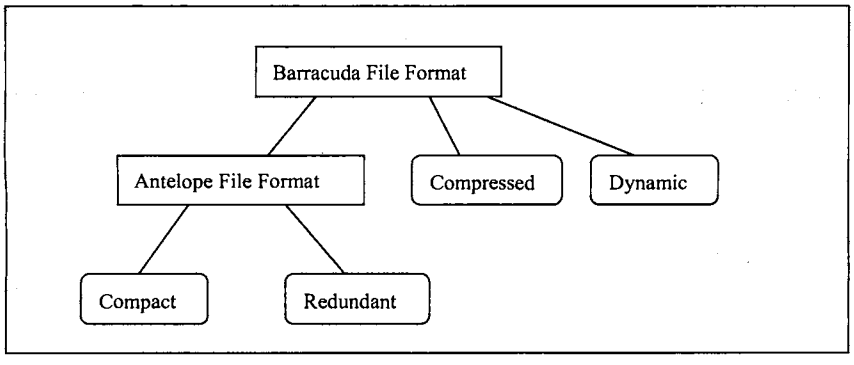


图4-8 文件格式

在InnoDB Plugin的官方手册中提到，未来版本的InnoDB存储引擎还将引入的新的文件格式，文件格式的名称取自动物的名字（这个学Apple？），并按照字母排序进行命名。

我翻阅了源代码，发现目前已经定义好的文件格式有：

```
/** List of animal names representing file format. */
static const char* file_format_name_map[] = {
"Antelope",
"Barracuda",
"Cheetah",
"Dragon",
"Elk",
"Fox",
"Gazelle",
"Hornet",
"Impala",
"Jaguar",
"Kangaroo",
"Leopard",
"Moose",
"Nautilus",
"Ocelot",
"Porpoise",
"Quail",
"Rabbit",
"Shark",
"Tiger",
"Urchin",
"Viper",
"Whale",
"Xenops",
"Yak",
"Zebra"
};
```

参数innodb_file_format用来指定文件格式，可以通过下面的方式查看当前所使用的InnoDB存储引擎的文件格式：

```
mysql> select @@version\G;
*************************** 1. row ***************************
@@version: 5.1.37
1 row in set (0.00 sec)

mysql> show variables like 'innodb_version'\G;
*************************** 1. row ***************************
Variable_name: innodb_version
        Value: 1.0.4
```

```
1 row in set (0.00 sec)

mysql> show variables like 'innodb_file_format'\G;
*************************** 1. row ***************************
Variable_name: innodb_file_format
        Value: Barracuda
1 row in set (0.00 sec)
```

参数innodb_file_format_check用来检测当前InnoDB存储引擎文件格式的支持度，该值默认为ON，如果出现不支持的文件格式，你可能在错误日志文件中看到类似如下的错误：

```
InnoDB: Warning: the system tablespace is in a
file format that this version doesn't support
```

4.7 约束

4.7.1 数据完整性

关系型数据库系统和文件系统的一个不同点是，关系数据库本身能保证存储数据的完整性，不需要应用程序的控制，而文件系统一般需要在程序端进行控制。几乎所有的关系型数据库都提供了约束（constraint）机制，约束提供了一条强大而简易的途径来保证数据库中的数据完整性，数据完整性有三种形式：

- 实体完整性　保证表中有一个主键。在InnoDB存储引擎表中，我们可以通过定义Primary Key或者Unique Key约束来保证实体的完整性。或者我们还可以通过编写一个触发器来保证数据完整性。
- 域完整性　保证数据的值满足特定的条件。在InnoDB存储引擎表中，域完整性可以通过以下几种途径来保证：选择合适的数据类型可以确保一个数据值满足特定条件，外键（Foreign Key）约束，编写触发器，还可以考虑用DEFAULT约束作为强制域完整性的一个方面。
- 参照完整性　保证两张表之间的关系。InnoDB存储引擎支持外键，因此允许用户定义外键以强制参照完整性，也可以通过编写触发器以强制执行。

对于InnoDB存储引擎而言，提供了4中约束：

- *Primary* Key

- Unique Key
- Foreign Key
- Default
- NOT NULL

4.7.2 约束的创建和查找

对于约束的建立，可以在表建立时就进行定义，也可以在之后使用ALTER TABLE命令来进行创建。对于Unique Key的约束，我们还可以通过Create Unique Index来进行建立。对于主键约束而言，其默认约束名为PRIMARY KEY。而对于Unique Key约束而言，默认约束名和列名一样，当然可以人为的指定一个名字。对于Foreign Key约束，似乎会有一个比较神秘的默认名称。下面是一个简单的创建表的语句，表上有一个主键和一个唯一键：

```
mysql> create table u ( id int , name varchar(20), id_card char(18), primary key
( id) ,unique key ( name ));
Query OK, 0 rows affected (0.16 sec)

mysql> select constraint_name,constraint_type
    -> from
    -> information_schema.TABLE_CONSTRAINTS
    -> where table_schema='mytest' and table_name='u'\G;
*************************** 1. row ***************************
constraint_name: PRIMARY
constraint_type: PRIMARY KEY
*************************** 2. row ***************************
constraint_name: name
constraint_type: UNIQUE
2 rows in set (0.00 sec)
```

可以看到，约束名就如我们前面说的。当然我们还可以通过ALTER TABLE来进行创建，并且可以定义约束的名字，如:

```
mysql> alter table u add unique key uk_id_card (id_card);
Query OK, 0 rows affected (0.19 sec)
Records: 0  Duplicates: 0  Warnings: 0

*************************** 1. row ***************************
constraint_name: PRIMARY
constraint_type: PRIMARY KEY
```

```
*************************** 2. row ***************************
constraint_name: name
constraint_type: UNIQUE
*************************** 3. row ***************************
constraint_name: uk_id_card
constraint_type: UNIQUE
3 rows in set (0.00 sec)
```

接着来看Foreign Key的约束，因此我们必须来创建另一张表：

```
mysql> create table p ( id int, u_id int, primary key ( id), foreign key (u_id)
references p (id));
Query OK, 0 rows affected (0.13 sec)

mysql> select constraint_name,constraint_type
    -> from
    -> information_schema.TABLE_CONSTRAINTS
    -> where table_schema='mytest' and table_name='p'\G;
*************************** 1. row ***************************
constraint_name: PRIMARY
constraint_type: PRIMARY KEY
*************************** 2. row ***************************
constraint_name: p_ibfk_1
constraint_type: FOREIGN KEY
2 rows in set (0.00 sec)
```

这里我们通过information_schema架构下的表TABLE_CONSTRAINTS来查看当前MySQL库下所有的约束。对于Foreign Key的约束的定义，我们还可以通过查看表REFERENTIAL_CONSTRAINTS，并且可以详细地了解外键的属性，如：

```
mysql> select * from information_schema.REFERENTIAL_CONSTRAINTS where
constraint_schema='mytest'\G;
*************************** 1. row ***************************
       CONSTRAINT_CATALOG: NULL
        CONSTRAINT_SCHEMA: test2
          CONSTRAINT_NAME: p_ibfk_1
UNIQUE_CONSTRAINT_CATALOG: NULL
 UNIQUE_CONSTRAINT_SCHEMA: test2
   UNIQUE_CONSTRAINT_NAME: PRIMARY
             MATCH_OPTION: NONE
              UPDATE_RULE: RESTRICT
              DELETE_RULE: RESTRICT
               TABLE_NAME: p
```

```
    REFERENCED_TABLE_NAME: p
1 row in set (0.00 sec)
```

4.7.3 约束和索引的区别

在前面的小节中我们已经看到Primary key和Unique Key的约束。有人不禁会问，这不就是我们创建索引的方法吗？那约束和索引有什么区别呢？的确，当你创建了一个唯一索引，就创建了一个唯一的约束。但是约束和索引的概念还是有所不同的，约束更是一个逻辑的概念，用来保证数据的完整性，而索引是一个数据结构，有逻辑上的概念，在数据库中更是一个物理存储的方式。

4.7.4 对于错误数据的约束

默认情况下，MySQL数据库允许非法或者不正确数据的插入或更新，或者内部将其转化为一个合法的值，如对于NOT NULL的字段插入一个NULL值，会将其更改为0再进行插入，因此本身没有对数据的正确性进行约束。我们来看一个例子：

```
mysql> create table a ( id int not null, date date not null);
Query OK, 0 rows affected (0.13 sec)

mysql> insert into a select NULL,'2009-02-30';
Query OK, 1 row affected, 2 warnings (0.04 sec)
Records: 1  Duplicates: 0  Warnings: 2

mysql> show warnings;
*************************** 1. row ***************************
  Level: Warning
   Code: 1048
Message: Column 'id' cannot be null
*************************** 2. row ***************************
  Level: Warning
   Code: 1265
Message: Data truncated for column 'date' at row 1
2 rows in set (0.00 sec)

mysql> select * from a;
+----+-------------+
| id  | date       |
```

```
+----+-------------+
|  0 | 0000-00-00 |
+----+-------------+
1 row in set (0.00 sec)
```

对于NOT NULL的列我插入了一个NULL值，同时插入了一个非法日期'2009-02-30'，MySQL都没有报错，而是显示了警告（warning）。如果我们想约束对于非法数据的插入或更新，MySQL是提示报错而不是警告，那么我们应该设置参数sql_mode，用来严格审核输入的参数，如：

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

mysql>  insert into a select NULL,'2009-02-30';
ERROR 1048 (23000): Column 'id' cannot be null

mysql>  insert into a select 1,'2009-02-30';
ERROR 1292 (22007): Incorrect date value: '2009-02-30' for column 'date' at row 1
```

我们的目的达到了，这次MySQL约束了输入值的合法性了，而且针对不同的错误，提示的错误内容也都不同。参数sql_mode可设的值有很多，具体的请参考MySQL官方文档。

4.7.5 ENUM和SET约束

MySQL不支持传统的CHECK约束，但是通过ENUM和SET类型可以解决部分这样的约束需求。如我们的表上有一个性别类型，规定域的范围只能是male或者female，这种情况下我们可以通过ENUM类型来进行约束：

```
mysql> create table a ( id int, sex enum('male','female'));
Query OK, 0 rows affected (0.12 sec)

mysql> insert into a select 1,'female';
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> insert into a select 2,'bi';
Query OK, 1 row affected, 1 warning (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 1
```

可以看到，对于第二条记录的插入依然是抱了警告。因此如果想实现CHECK约束，还

需要设置参数sql_mode：

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

mysql> insert into a select 2,'bi';
ERROR 1265 (01000): Data truncated for column 'sex' at row 1
```

这次对于非法的输入值进行了约束，但是只限于对离散数值的约束，对于传统CHECK约束支持的连续值的范围约束或者更复杂的约束，ENUM和SET类型还是无能为力，这时我们就需要通过触发器来实现约束了。

4.7.6 触发器与约束

从前面小节的介绍中我们知道，完整性约束通常也可以使用触发器来实现，因此在了解数据完整性前我们先对触发器来做一个了解。

触发器的作用是在INSERT、DELETE和UPDATE命令之前或之后自动调用SQL命令或者存储过程。MySQL 5.0对于触发器的实现还不是非常完善，限制比较多；而从MySQL 5.1开始，触发器已经相对稳定，功能也较之前有了大幅的提高。

创建触发器的命令是CREATE TRIGGER，只有具备Super权限的MySQL用户才可以执行这条命令：

```
CREATE
[DEFINER = { user | CURRENT_USER }]
TRIGGER trigger_name BEFORE|AFTER INSERT|UPDATE|DELETE
ON tbl_name FOR EACH ROW trigger_stmt
```

最多可以为一个表建立6个触发器，即分别为INSERT、UPDATE、DELETE的BEFORE和AFTER各定义一个。BEFORE和AFTER代表触发器发生的时间，表示是在每行操作的之前发生还是之后发生。当前MySQL只支持FOR EACH ROW的触发方式，即按每行记录进行触发，不支持如DB2的FOR EACH STATEMENT的触发方式。

通过触发器，我们可以实现MySQL数据库本身并不支持的一些特性，如对于传统CHECK约束的支持、物化视图、高级复制、审计等特性。这里我们先关注触发器对于约束的支持。我们考虑用户消费表，每次用户购买一样物品后其金额都是减的，若这时有不怀好意的人做了类似减去一个负值的操作，这样的话用户的钱没减少反而会不断地增加。

```
mysql> create table usercash ( userid int , cash int unsigned not null );
Query OK, 0 rows affected (0.11 sec)

mysql> insert into usercash select 1,1000;
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> update usercash set cash=cash-(-20) where userid=1;
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

对于数据库来说，上述的内容没有任何问题，都可以正常运行，不会报错。但是从业务的逻辑上来说，这是错误的，消费总是应该减去一个正值，而不是负值。因此这时如果通过触发器来约束这个逻辑行为的话，可以如下操作：

```
mysql> create table usercash_err_log (
    -> userid int not null,
    -> old_cash int unsigned not null,
    -> new_cash int unsigned not null,
    -> user varchar(30),
    -> time datetime);
Query OK, 0 rows affected (0.13 sec)

mysql> delimiter $$
mysql> create trigger tgr_usercash_update before update on usercash
    -> for each row
    -> begin
    -> if new.cash-old.cash > 0 then
    -> insert into usercash_err_log select old.userid,old.cash,new.cash,user(),now();
    -> set new.cash = old.cash;
    -> end if;
    -> end;
    -> $$
Query OK, 0 rows affected (0.00 sec)

mysql> delete from usercash;
Query OK, 1 row affected (0.02 sec)

mysql> insert into usercash select 1,1000;
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> update usercash set cash = cash - (-20) where userid=1;
```

```
Query OK, 0 rows affected (0.02 sec)
Rows matched: 1  Changed: 0  Warnings: 0

mysql> select * from usercash;
+--------+-------+
| userid  | cash |
+--------+-------+
|   1    | 1000 |
+--------+-------+
1 row in set (0.00 sec)

mysql> select * from usercash_err_log ;
+--------+-----------+-----------+-------------------
+-----------------------------+
| userid  | old_cash | new_cash | user          | time                  |
+--------+-----------+-----------+---------==-------
+-----------------------------+
|      1 |   1000 |     1020 | root@localhost | 2009-11-06 11:49:49  |
+--------+-----------+-----------+----------==------
+-----------------------------+
1 row in set (0.00 sec)
```

我们创建了一张表用来记录错误数值更新的日志，首先判断新旧值之间的差值，正常情况下消费总是减的，因此新值应该总是小于原来的值，因此对于大于原值的数据，我们判断为非法的输入，将cash值设定为原来的值。

4.7.7 外键

外键用来保证参照完整性，MySQL默认的MyISAM存储引擎本身并不支持外键，对于外键的定义只是起到一个注释的作用。InnoDB存储引擎则完整支持外键约束。外键的定义如下：

```
[CONSTRAINT [symbol]] FOREIGN KEY
[index_name] (index_col_name, ...)
REFERENCES tbl_name (index_col_name,...)
[ON DELETE reference_option]
[ON UPDATE reference_option]
reference_option:
RESTRICT | CASCADE | SET NULL | NO ACTION
```

我们可以在CREATE TABLE时就添加外键，也可以在表创建后通过ALTER TABLE命

令来添加。一个简单的外键的创建示例如下：

```
mysql> CREATE TABLE parent (id INT NOT NULL,
    -> PRIMARY KEY (id)
    -> ) ENGINE=INNODB;
Query OK, 0 rows affected (0.13 sec)

mysql> CREATE TABLE child (id INT, parent_id INT,
    -> INDEX par_ind (parent_id),
    -> FOREIGN KEY (parent_id) REFERENCES parent(id)
    -> ) ENGINE=INNODB;
Query OK, 0 rows affected (0.16 sec)
```

一般来说，我们称被引用的表为父表，另一个引用的表为子表。外键定义为，ON DELETE和ON UPDATE表示父表做DELETE和UPDATE操作时子表所做的操作。可定义的子表操作有：

- CASCADE：当父表发生DELETE或UPDATE操作时，相应的子表中的数据也被DELETE或UPDATE。
- SET NULL：当父表发生DELETE或UPDATE操作时，相应的子表中的数据被更新为NULL值。当然，子表中相对应的列必须允许NULL值。
- NO ACTION：当父表发生DELETE或UPDATE操作时，抛出错误，不允许这类操作发生。
- RESTRICT：当父表发生DELETE或UPDATE操作时，抛出错误，不允许这类操作发生。如果定义外键时没有指定ON DELETE或ON UPDATE，这就是默认的外键设置。

在Oracle中，有一种称为延时检查（deferred check）的外键约束，而目前MySQL的约束都是即时检查（immediate check）的，因此从上面的定义可以看出，在MySQL数据库中NO ACTION和RESTRICT的功能是相同的。

在Oracle数据库中，外键通常被人忽视的地方是，对于建立外键的列，一定不要忘记给这个列加上一个索引。而InnoDB存储引擎在外键建立时会自动地对该列加一个索引，这和Microsoft SQL Server数据库的做法一样。因此可以很好地避免外键列上无索引而导致的死锁问题的产生。

对于参照完整性约束，外键能起到一个非常好的作用。但是对于数据的导入操作，外

键往往导致大量时间花费在外键约束的检查上，因为MySQL的外键是即时检查的，因此导入的每一行都会进行外键检查。但是我们可以在导入过程中忽视外键的检查，如：

```
mysql> SET foreign_key_checks = 0;
mysql> LOAD DATA ......
mysql> SET foreign_key_checks = 1;
```

4.8 视图

视图（View）是一个命名的虚表，它由一个查询来定义，可以当做表使用。与持久表（permanent table）不同的是，视图中的数据没有物理表现形式。

4.8.1 视图的作用

视图在数据库中发挥着重要的作用。视图的主要用途之一是被用做一个抽象装置，特别是对于一些应用程序，程序本身不需要关心基表（base table）的结构，只需要按照视图定义来获取数据或者更新数据，因此，视图同时在一定程度上起到一个安全层的作用。

MySQL 从5.0版本开始支持视图，创建视图的语法如下：

```
CREATE
[OR REPLACE]
[ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
[DEFINER = { user | CURRENT_USER }]
[SQL SECURITY { DEFINER | INVOKER }]
VIEW view_name [(column_list)]
AS select_statement
[WITH [CASCADED | LOCAL] CHECK OPTION]
```

虽然视图是基于基表的一个虚拟表，但是我们可以对某些视图进行更新操作，其实就是通过视图的定义来更新基本表，我们称可以进行更新操作的视图为可更新视图（updatable view）。视图定义中的WITH CHECK OPTION就是指对于可更新的视图，更新的值是否需要检查。我们先看个例子:

```
mysql> create table t ( id int );
Query OK, 0 rows affected (0.13 sec)

mysql> create view v_t as select * from t where t<10;
ERROR 1054 (42S22): Unknown column 't' in 'where clause'
```

```
mysql> create view v_t as select * from t where id<10;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into v_t select 20;
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> select * from v_t;
Empty set (0.00 sec)
```

我们创建了一个id<10的视图，但是往里插入了id为20的值，插入操作并没有报错，但是我们查询视图还是没有能查到数据。接着我们更改一下视图的定义，加上WITH CHECK OPTION：

```
mysql> alter view v_t as select * from t where id<10 with check option;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into v_t select 20;
ERROR 1369 (HY000): CHECK OPTION failed 'mytest.v_t'
```

这次MySQL数据库会对更新视图插入的数据进行检查，对于不满足视图定义条件的，将会抛出一个异常，不允许数据的更新。

MysQL DBA一个常用的命令是show tables，会显示出当前数据库下的表，视图是虚表，同样被作为表而显示出来，我们来看前面的例子：

```
mysql> show tables;
+--------------------+
| Tables_in_mytest |
+--------------------+
| t                  |
| v_t                |
+--------------------+
2 rows in set (0.00 sec)
```

show tables命令把表t和视图v_t都显示出来了。如果我们只想查看当前数据库下的基表，可以通过information_schema架构下的TABLE表来查询，并搜索表类型为BASE TABLE的表，如：

```
mysql> select * from information_schema.TABLES where table_type='BASE TABLE' and
table_schema=database()\G;
*************************** 1. row ***************************
```

```
  TABLE_CATALOG: NULL
   TABLE_SCHEMA: mytest
     TABLE_NAME: t
     TABLE_TYPE: BASE TABLE
         ENGINE: InnoDB
        VERSION: 10
     ROW_FORMAT: Compact
     TABLE_ROWS: 1
 AVG_ROW_LENGTH: 16384
    DATA_LENGTH: 16384
MAX_DATA_LENGTH: 0
   INDEX_LENGTH: 0
      DATA_FREE: 0
 AUTO_INCREMENT: NULL
    CREATE_TIME: 2009-11-09 16:27:52
    UPDATE_TIME: NULL
     CHECK_TIME: NULL
TABLE_COLLATION: utf8_general_ci
       CHECKSUM: NULL
 CREATE_OPTIONS:
  TABLE_COMMENT:
1 row in set (0.00 sec)
```

要想查看视图的一些元数据（meta data），可以访问information_schema架构下的VIEWS表，该表给出了视图的详细信息，包括视图定义者（definer）、定义内容、是否是可更新视图、字符集等。如我们查询VIEWS表，可得：

```
mysql> select * from information_schema.VIEWS where table_schema=database()\G;
*************************** 1. row ***************************
       TABLE_CATALOG: NULL
        TABLE_SCHEMA: mytest
          TABLE_NAME: v_t
     VIEW_DEFINITION: select 'mytest'.'t'.'id' AS 'id' from 'mytest'.'t' where
('mytest'.'t'.'id' < 10)
        CHECK_OPTION: CASCADED
        IS_UPDATABLE: YES
             DEFINER: root@localhost
       SECURITY_TYPE: DEFINER
CHARACTER_SET_CLIENT: latin1
COLLATION_CONNECTION: latin1_swedish_ci
1 row in set (0.00 sec)
```

4.8.2 物化视图

Oracle数据库支持物化视图——该视图不是基于基表的虚表，而是根据基表实际存在的实表。物化视图可以用于预先计算并保存表连接或聚集等耗时较多的操作结果，这样，在执行复杂查询时，就可以避免进行这些耗时的操作，从而快速得到结果。物化视图的好处是，对于一些复杂的统计类查询能直接查出结果。在Microsoft SQL Server数据库中，称这种视图为索引视图。

在Oracle数据库中，物化视图的创建方式包括BUILD IMMEDIATE和BUILD DEFERRED这两种。BUILD IMMEDIATE是默认的创建方式，在创建物化视图的时候就生成数据，而BUILD DEFERRED则在创建时不生成数据，以后根据需要再生成数据。

查询重写是指当对物化视图的基表进行查询时，Oracle会自动判断能否通过查询物化视图来得到结果。如果可以，则避免了聚集或连接操作，而直接从已经计算好的物化视图中读取数据。

物化视图的刷新是指当基表发生了DML操作后，物化视图何时采用哪种方式和基表进行同步。刷新的模式有两种：ON DEMAND和ON COMMIT。ON DEMAND指物化视图在用户需要的时候进行刷新，ON COMMIT指物化视图在对基表的DML操作提交的同时进行刷新。刷新的方法有四种：FAST、COMPLETE、FORCE和NEVER。FAST刷新采用增量刷新，只刷新自上次刷新以后进行的修改。COMPLETE刷新对整个物化视图进行完全的刷新。如果选择FORCE方式，则Oracle在刷新时会去判断是否可以进行快速刷新，如果可以则采用 FAST方式，否则采用COMPLETE的方式。NEVER指物化视图不进行任何刷新。

MySQL数据库本身并不支持物化视图，换句话说，MySQL数据库中的视图总是虚拟的，但是我们可以通过一些机制来实现物化视图的功能。要创建一个ON DEMAND的物化视图还是比较简单的，我们可以定时把数据导入另一张表。例如，我们有如下的订单表，记录了用户采购电脑设备：

```
mysql> create table Orders
    -> (
    -> order_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    -> product_name VARCHAR(30) NOT NULL,
    -> price DECIMAL(8,2) NOT NULL,
```

```
    -> amount SMALLINT     NOT NULL,
    -> primary key (order_id)
    -> )ENGINE=InnoDB;
Query OK, 0 rows affected (0.13 sec)

mysql> INSERT INTO Orders VALUES
    -> (NULL, 'CPU', 135.5, 1),
    -> (NULL,'Memory',48.2,3),
    -> (NULL, 'CPU', 125.6, 3),
    -> (NULL, 'CPU', 105.3,4)
    -> ;
Query OK, 4 rows affected (0.03 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> select * from Orders\G;
*************************** 1. row ***************************
    order_id: 1
product_name: CPU
       price: 135.50
      amount: 1
*************************** 2. row ***************************
    order_id: 2
product_name: Memory
       price: 48.20
      amount: 3
*************************** 3. row ***************************
    order_id: 3
product_name: CPU
       price: 125.60
      amount: 3
*************************** 4. row ***************************
    order_id: 4
product_name: CPU
       price: 105.30
      amount: 4
4 rows in set (0.00 sec)
```

接着我们需要建立一张物化视图，用来统计每件物品的信息，如：

```
mysql> CREATE TABLE Orders_MV(
    ->     product_name VARCHAR(30)  NOT NULL
    ->   , price_sum    DECIMAL(8,2) NOT NULL
    ->   , amount_sum   INT           NOT NULL
    ->   , price_avg    FLOAT         NOT NULL
```

```
    ->   , orders_cnt    INT           NOT NULL
    ->   , UNIQUE INDEX (product_name)
    -> );
Query OK, 0 rows affected (0.13 sec)

mysql> INSERT INTO Orders_MV
    -> SELECT product_name
    ->      , SUM(price), SUM(amount), AVG(price)
    ->      , COUNT(*)
    ->   FROM Orders
    -> GROUP BY product_name;
Query OK, 2 rows affected (0.02 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql>
mysql>
mysql> select * from Orders_MV;
+----------------+-------------+----------------+-------------+-------------+
| product_name | price_sum | amount_sum | price_avg | orders_cnt |
+----------------+-------------+----------------+-------------+-------------+
| CPU          |   366.40 |           8 |  122.133 |         3 |
| Memory       |    48.20 |           3 |     48.2 |         1 |
+----------------+-------------+----------------+-------------+-------------+
2 rows in set (0.00 sec)
```

这里我们把物化视图定义为一张表，只不过表名以_MV结尾，让DBA能很好地理解这张表的作用。这样就有了一个统计信息，如果是要实现ON DEMAND的物化视图，只需把表清空，重新导入数据即可。当然，这是完全（Complete）刷新方式。要实现快（Fast）刷新方式，其实也是可以的，只不过稍微复杂点，需要记录上次统计时的order_id的位置。

但是如果要实现On Commit的物化视图，这就不是如上面这么简单了。Oracle数据库中通过物化视图日志来实现，很显然MySQL数据库没有这个日志，但是通过触发器，我们同样可以达到这个目的：

```
DELIMITER $$

CREATE TRIGGER tgr_Orders_insert
AFTER INSERT ON Orders
FOR EACH ROW
BEGIN
  SET @old_price_sum = 0;
```

```
  SET @old_amount_sum = 0;
  SET @old_price_avg = 0;
  SET @old_orders_cnt = 0;

   SELECT IFNULL(price_sum, 0), IFNULL(amount_sum, 0), IFNULL(price_avg, 0),
IFNULL(orders_cnt, 0)
    FROM Orders_MV
   WHERE product_name = NEW.product_name
INTO @old_price_sum, @old_amount_sum, @old_price_avg,  @old_orders_cnt;

  SET @new_price_sum = @old_price_sum + NEW.price;
  SET @new_amount_sum = @old_amount_sum + NEW.amount;
  SET @new_orders_cnt = @old_orders_cnt + 1;
  SET @new_price_avg = @new_price_sum / @new_orders_cnt ;

  REPLACE INTO Orders_MV
   VALUES(NEW.product_name, @new_price_sum, @new_amount_sum, @new_price_avg,
@new_orders_cnt );

END;
$$

DELIMITER ;

mysql> insert into Orders values (NULL,'SSD',299,3);
Query OK, 1 row affected, 1 warning (0.03 sec)

mysql> insert into Orders values (NULL,'Memory',47.9,5);
Query OK, 1 row affected (0.03 sec)

mysql> select * from Orders_MV;
+----------------+-------------+---------------+-----------+-------------+
| product_name | price_sum | amount_sum | price_avg | orders_cnt |
+----------------+-------------+---------------+-----------+-------------+
| CPU          |   366.40 |          8 |  122.133 |         3 |
| Memory       |    96.10 |          8 |    48.05 |         2 |
| SSD          |   299.00 |          3 |      299 |         1 |
+----------------+-------------+---------------+-----------+-------------+
3 rows in set (0.00 sec)
```

这里对表Orders添加了一个INSERT的触发器，每次Insert操作都会重新统计Orders_MV中的数据，这样就实现了ON_Commit的物化视图功能。但是Orders表可能还会有Update和Delete的操作，所以应该还需要实现Delete和Update的触发器——不过这就留着读

者自己实现了。

通过触发器我们实现了物化视图的功能，但是MySQL本身并不支持物化视图，因此对于物化视图支持的查询重写（Query Rewrite）功能就显得无能为力了。

4.9 分区表

4.9.1 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。但也并不是所有的存储引擎都支持，如CSV、FEDERATED、MERGE等就不支持。在使用分区功能前，应该了解所选择的存储引擎对于分区的支持。

MySQL数据库在5.1版本时添加了对于分区的支持，这个过程是将一个表或者索引物分解为多个更小、更可管理的部分。就访问数据库的应用而言，从逻辑上讲，只有一个表或者一个索引，但是在物理上这个表或者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，一个分区中既存放了数据又存放了索引。

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:

```
mysql> show plugins\G;
......
*************************** 2. row ***************************
   Name: partition
```

⊖ 水平分区，指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。

```
 Status: ACTIVE
   Type: STORAGE ENGINE
Library: NULL
License: GPL
......
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，数据库就会变得更快。这个结论是存在很多问题的。就我的经验来看，分区对于某些SQL语句性能可能会带来提高，但是分区主要用于高可用性，利于数据库的管理。在OLTP应用中，对于分区的使用应该非常小心。总之，如果你只是一味地使用分区，而不理解分区是如何工作的，也不清楚你的应用如何使用分区，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数据库5.5开始支持RANGE COLUMNS的分区。
- LIST分区：和RANGE分区类似，只是LIST分区面向的是离散的值。MySQL数据库5.5开始支持LIST COLUMNS的分区。
- HASH分区：根据用户自定义的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数据库提供的哈希函数来进行分区。

不论创建何种类型的分区，如果表中存在主键或者是唯一索引时，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
```

唯一索引可以是允许NULL值的，并且分区列只要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，如下代码所示。

```
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，唯一索引时，可以指定任何一个列为分区列，因此下面2句创建分区的SQL语句都是可以正确运行的：

```
mysql> CREATE TABLE t1 (
    -> col1 INT  NULL,
    -> col2 DATE NULL,
    -> col3 INT NULL,
    -> col4 INT NULL
    -> )engine=innodb
    -> PARTITION BY HASH(col3)
    -> PARTITIONS 4;
Query OK, 0 rows affected (0.40 sec)

mysql> drop table t1;
Query OK, 0 rows affected (0.11 sec)

mysql> 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;
Query OK, 0 rows affected (0.65 sec)
```

4.9.2 RANGE分区

我们介绍的第一种类型是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，数据插入p0分区。当id大于等

于10小于20时，插入p1分区：

```
mysql> create table t(
    -> id int)engine=innodb
    -> 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文件组成了，而是由建立分区时的各个分区ibd文件组成，如下所示的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
```

因为表t根据列id进行分区，因此数据是根据id列的值的范围存放在不同的物理文件中的，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：

```
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1条记录、分区p1中有2条记录。PARTITION_METHOD表示分区的类型，这里显示的是RANGE。

对于表t，因为我们定义了分区，因此对于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。如下所示，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的值放入p2分区：

```
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，如对于销售类的表，可以根据年来分区存放销售记录，如以下所示的分区表sales：

```
mysql> create table sales(
    -> money int unsigned not null,
    -> date datetime
    -> )engine=innodb
    -> partition by range  (YEAR(date)) (
    -> partition p2008 values less than(2009),
    -> partition p2009 values less than (2010),
    -> partition p2010 values less than (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这张表的管理。如果我们要删除2008年的数据，就不需要执行DELETE FROM sales WHERE date>='2008-01-01' and date <'2009-01-01'，而只需删除2008年数据所在的分区即可：

```
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，可以加快某些查询的操作。如果我们只需要查询2008年整年的销售额：

```
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优化器只需要去搜索p2008这个分区，而不会去搜索所有的分区，因此大大提高了执行的速度。需要注意的是，如果执行下列语句，结果是一样的，但是优化器的选择又会不同了：

```
mysql> explain partitions 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)
```

这次条件改为date<'2009-01-01'而不是date<='2008-12-31'时，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，你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，我曾看到过另一种分区函数，设计者的原意是想可以按照每年每月来进行分区，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 than(201002),
    ->      partition p201002 values less than (201003),
    ->      partition p201003 values less than (201004)
    ->      );
Query OK, 0 rows affected (0.37 sec)
```

但是在执行SQL语句时开发人员会发现，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，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、p201003都进行了搜索。产生这个问题的主要原因是，对于RANGE分区的查询，优化器只能对YEAR()、TO_DAYS()、TO_SECONDS()、UNIX_TIMESTAMP()这类函数进行优化选择，因此对于上述的要求，需要将分区函数改为TO_DAYS，如：

```
mysql> create table sales(
    -> money int unsigned not null,
    -> date datetime
    -> )engine=innodb
    -> partition by range(to_days(date)) (
    -> partition p201001 values less than(to_days('2010-02-01')),
    -> partition p201002 values less than(to_days('2010-03-01')),
    -> partition p201003 values less than(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了：

```
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
         type: ALL
possible_keys: NULL
          key: NULL
```

```
       key_len: NULL
           ref: NULL
          rows: 4
         Extra: Using where
1 row in set (0.00 sec)
```

4.9.3 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。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，所以每个分区的值是离散的，只能是定义的值。如我们往表中插入一些数据：

```
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，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，在用INSERT插入多个行数据的过程中遇到分区未定义的值时，MyISAM和InnoDB存储引擎的处理完全不同。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，因此没有任何数据插入。先对MyISAM存储引擎进行演示，如：

```
mysql> create 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)
```

可以看到对于插入的（6，10）记录没有成功，但是之前的（1，2），（2，4）记录都已经插入成功了。而同一张表，存储引擎换成InnoDB，则结果完全不同：

```
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）记录是报错，但是没有任何一条记录被插入表t中。因此在使用分区时，也需要对不同存储引擎支持的事务特性进行考虑。

4.9.4 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都是一样的。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，你所要做的只是基于将要被哈希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”是一个返回一个整数的表达式。它可以仅仅是字段类型为MySQL 整型的一列的名字。此外，你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。如果没有包括一个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：

```
mysql> create table t_hash (
    -> a int,
    -> b datetime)engine=innodb
    -> partition by hash (YEAR(b))
    -> partitions 4;
Query OK, 0 rows affected (0.42 sec)
```

如果将一个列b为2010-04-01这个记录插入表t_hash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
```

因此会放入分区2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分区有1条记录。当然这个例子中并不能把数据均匀地分布到各个分区中，因为分区是按照YEAR函数，因此这个值本身可以视为是离散的。如果对于连续的值进行HASH分区，如自增长的主键，则可以很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，它和之前的表t_hash相似，只是分区类型不同：

```
mysql> create table t_linear_hash(
    -> a int,
    -> b datetime)engine=innodb
```

```
    -> partition by linear hash (year(b))
    -> partition by 4;
Query OK, 0 rows affected (0.42 sec)
```

同样插入‘2010-04-01’的记录，这次MySQL数据库根据以下的方法来进行分区的判断：

（1）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；

（2）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的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；它的缺点在于，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。

4.9.5 KEY分区

KEY分区和HASH分区相似；不同在于，HASH分区使用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哈希函数，这些函数是基于与PASSWORD()一样的运算法则。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)
```

在KEY分区中使用关键字LINEAR，和在HASH分区中具有同样的作用，分区的编号是通过2的幂（powers-of-two）算法得到的，而不是通过模数算法。

4.9.6 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(interger)，如果不是整型，那应该需要通过函数将其转化为整型，如YEAR()、TO_DAYS()、MONTH()等函数。MySQL数据库5.5版本开始支持COLUMNS分区，可视为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，不需要转化为整型。其次，RANGE COLUMNS分区可以对多个列的值进行分区。

COLUMNS分区支持以下的数据类型：

- 所有的整型类型，如INT、SMALLINT、TINYINT、BIGINT。FLOAT和DECIMAL则不予支持。
- 日期类型，如DATE和DATETIME。其余的日期类型不予支持。
- 字符串类型，如CHAR、VARCHAR、BINARY和VARBINARY。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()函数了，而直接可以使用

COLUMNS，如：

```
mysql> 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')
    -> );
Query OK, 0 rows affected (0.00 sec)
```

同样，可以直接使用字符串的分区：

```
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('Nössjö', '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，如：

```
mysql> CREATE TABLE rcx (
-> a INT,
-> b INT,
-> c CHAR(3),
-> d INT
-> )
-> 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)
-> );
Partitioning
1494
Query OK, 0 rows affected (0.15 sec)
```

MySQL数据库版本5.5.0开始支持COLUMNS分区，对于之前的RANGE和LIST分区，我们应该可以用RANGE COLUMNS和LIST COLUMNS分区进行很好的代替。

4.9.7 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或者是KEY的子分区，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，然后又再进行了一次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语法来显式指出各个子分区的名称，同样对上述的ts表：

```
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。
- 如果在一个分区表上的任何分区上使用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称。
- 在每个分区内，子分区的名称必须是唯一的。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。现在考虑下面的例子：

```
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)
```

但是InnoDB存储引擎会忽略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其是无效的：

```
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
```

4.9.8 分区中的NULL值

MySQL数据库允许对NULL值做分区，但是处理的方法和Oracle数据库完全不同。MYSQL数据库的分区总是把NULL值视为小于任何一个非NULL值，这和MySQL数据库中对于NULL的ORDER BY的排序是一样的。因此对于不同的分区类型，MySQL数据库对于NULL值的处理是不一样的。

对于RANGE分区，如果对于分区列插入了NULL值，则MySQL数据库会将该值放入最左边的分区（这和Oracle数据库完全不同，Oracle数据库会将NULL值放入MAXVALUE分区中）。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)
```

接着往表中插入（1,1）和（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)
```

可以看到两条数据都放入了p0分区，也就是说明了RANGE分区下，NULL值会放入最左边的分区中。另外需要注意的是，如果删除p0这个分区，你删除的是小于10的记录，并且还有NULL值的记录，这点非常重要：

```
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)
```

LIST分区下要使用NULL值，则必须显式地指出哪个分区中放入NULL值，否则会报错，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：

```
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分区对于NULL的处理方式，和RANGE分区、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。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)
```

4.9.9 分区和性能

我常听到开发人员说“对表做个分区，然后数据库的查询就会快了”。但是这是真的吗？实际中可能你根本感觉不到查询速度的提升，甚至是查询速度急剧的下降。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（在线事务处理），如博客、电子商务、网络游戏等；另一类是OLAP（在线分析处理），如数据仓库、数据集市。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。如网络游戏中，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来更好地服务于游戏、预测玩家的行为等，而这却是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。你的查询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。

对于OLTP的应用，分区应该非常小心。在这种应用下，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（到现在我都没看到过4层的B+树索引）。因此B+树可以很好地完成操作，不需要分区的帮助，并且设计不好的分区会带来严重的性能问题。

我发现，很多开发团队会认为含有1000万行的表是一张非常巨大的表，所以他们往往会选择采用分区，如对主键做10个HASH的分区，这样每个分区就只有100万行的数据了，因此查询应该变得更快了，如SELECT * FROM TABLE WHERE PK=@pk。但是有没有考虑过这样一个问题：100万行和1000万行的数据本身构成的B+树的层次都是一样的，可能都是2层？那么上述走主键分区的索引并不会带来性能的提高。是的，即使1000万行的B+树的高度是3，100万行的B+树的高度是2，那么上述走主键分区的索引可以避免1次IO，从而提高查询的效率。嗯，这没问题，但是这张表只有主键索引，而没有任何其他的列需要查询？如果还有类似如下的语句SQL：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，则一共需要20次IO。而对于原来单表的设计，对于KEY的查询还是2～3次IO。如下表Profile，根据主键ID进行了HASH分区，HASH分区的数量为10，表Profile有接近1000万行的数据：

```
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)
```

因为是根据HASH分区的，因此每个区分的记录数大致是相同的，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的均匀。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，则该值将不会再次被自动使用。

如果进行主键的查询，可以发现分区的确是有意义的：

```
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，但是对于表Profile中nickname列索引的查询，EXPLAIN PARTITIONS则会得到如下的结果：

```
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，因此查询速度会慢很多，比较上述的语句：

```
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，这显然是因为搜索所有分区的关系，实际的IO执行了20～30次，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，设计时要确认数据的访问模式，否则在OLTP应用下分区可能不仅不会带来查询速度的提高，反而可能会使你的应用执行得更慢。

4.10 小结

读完这一章后，希望你对InnoDB存储引擎表有了一个更深刻的理解。

在这一章中，我们首先介绍了InnoDB存储引擎表总是按照主键索引顺序进行存放的。

然后深入介绍了表的物理实现（如行结构和页结构），这一部分有助于你更进一步了解表物理存储的底层。

接着介绍了和表有关的约束问题，MySQL通过约束来保证表中数据的各种完整性，其中也提到了有关InnoDB存储引擎支持的外键特性。

之后介绍了视图，MySQL中视图总是虚拟的表，本身不支持物化视图。但是通过一些其他技巧（如触发器），同样也可以实现一些简单的物化视图的功能。

最后介绍了分区，MySQL数据库支持RANGE、LIST、HASH、KEY、COLUMNS分区，并且可以使用HASH或者KEY来进行子分区。需要注意的是，分区并不总是适合于OLTP应用，你应该根据自己的应用好好规划自己的分区设计。

第5章　索引与算法

索引是应用程序设计和开发的一个重要方面。如果索引太多，应用的性能可能会受到影响；如果索引太少，对查询性能又会产生影响。要找到一个合适的平衡点，这对应用的性能至关重要。

一些开发人员总是在事后才想起添加索引——我一直认为，这源于一种错误的开发模式。如果知道数据的使用，从一开始就应该在需要处添加索引。开发人员对于数据库的工作往往停留在应用的层面，比如编写SQL语句、存储过程之类，他们甚至可能不知道索引的存在，或者认为事后让相关DBA加上即可。而DBA往往不了解业务的数据流，添加索引需要通过监控大量的SQL语句，从中找到问题。这个步骤需要的时间肯定是大于初始添加索引所需要的时间，并且可能会遗漏一部分索引。当然索引不是越多越好，我曾经遇到这样一个问题：某台MySQL服务器iostat显示磁盘使用率100%，经过分析后发现，是由于开发人员添加了太多的索引。在删除一些不必要的索引之后，磁盘使用率马上下降为20%，因此索引的添加也是有一定技巧的。

这一章的主旨是对InnoDB存储引擎支持的索引做一个概述，深入解析索引内部的机制，通过了解索引内部构造掌握在哪里可以使用索引。本章的风格和别的有关MySQL的书籍可能有所不同，更偏重于索引内部的实现和算法问题的讨论。

5.1　InnoDB存储引擎索引概述

InnoDB存储引擎支持两种常见的索引，一种是B+树索引，另一种是哈希索引。前面已经提到过，InnoDB存储引擎支持的哈希索引是自适应的，InnoDB存储引擎会根据表的使用情况自动为表生成哈希索引，不能人为干预是否在一张表中生成哈希索引。

B+树索引就是传统意义上的索引，这是目前关系型数据库系统中最常用、最有效的索引。B+树索引的构造类似于二叉树，根据键值（Key Value）快速找到数据。需要注意的

是，B+树中的B不是代表二叉（binary），而是代表平衡（balance），因为B+树是从最早的平衡二叉树演化而来，但是B+树不是一个二叉树。

一个常常被DBA忽视的问题是：B+树索引并不能找到一个给定键值的具体行。B+树索引能找到的只是被查找数据行所在的页。然后数据库通过把页读入内存，再在内存中进行查找，最后得到查找的数据。

5.2 二分查找法

二分查找法（binary search）也称为折半查找法，用来查找一组有序的记录数组中的某一记录。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。

例如我们有5、10、19、21、31、37、42、48、50、52这10个数，现要从这10个数中查找48这条记录，其查找过程如图5-1所示。

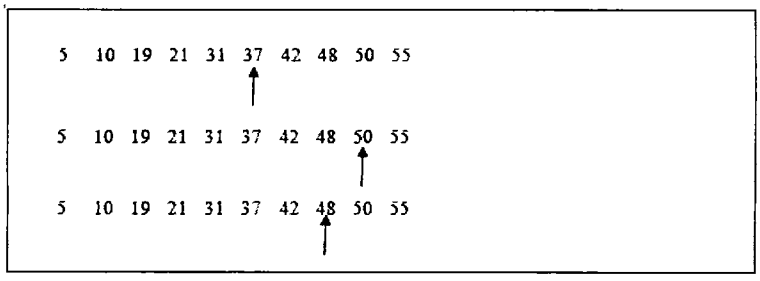


图5-1 二分查找法

从图5-1可以看出，用了3次就找到了48这个数。如果是顺序查找的话，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）。但如果查5这条记录，顺序查找只需1次，而二分查找法却需要4次。对于上面10个数来说，顺序查找的平均查找次数为（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，而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。在前面的章节中，

我们已经知道了，每页Page Directory中的槽是按照主键的顺序存放的，对于某一条具体记录的查询是通过对Page Directory进行二分查找得到的。

5.3 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5-2显示了一颗二叉查找树。

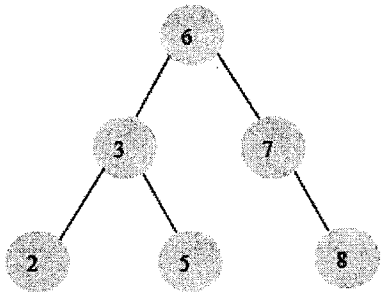


图5-2 二叉查找树

图5-2中的数字代表每个节点的键值，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，图5-2中序遍历后输出：2、3、5、6、7、8。

对图5-2的这颗二叉树进行查找，如查键值为5的记录，先找到根，其键值是6，6大于5，因此找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。用同样的方法再找键值为8的这个记录，这次用了3次查找，而顺序查找时需要6次。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快。

二叉查找树可以任意构造，同样的2、3、5、6、7、8这五个数字，也可以按照图5-3所示的方式建立二叉查找树。

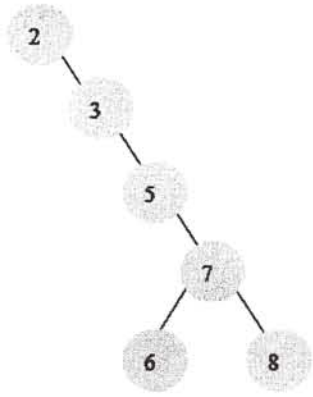


图5-3 效率较低的一颗二叉查找树

图5-3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。显然这次二叉查找树的效率就低了。因此若想最大性能地构造一个二叉查找树，需要这颗二叉查找树是平衡的，因此引出了新的定义——平衡二叉树，或称为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左右两个子树的高度最大差为1。显然，图5-3不满足平衡二叉树的定义，而图5-2是一颗平衡二叉树。平衡二叉树对于查找的性能是比较高的，但不是最高的，只是接近最高性能。要达到最好的性能，需要建立一颗最优二叉树，但是最优二叉树的建立和维护需要大量的操作，因此我们一般只需建立一颗平衡二叉树即可。

平衡二叉树对于查询速度的确很快，但是维护一颗平衡二叉树的代价是非常大的，通常需要1次或多次左旋和右旋来得到插入或更新后树的平衡性。如图5-2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5-4所示的变动。

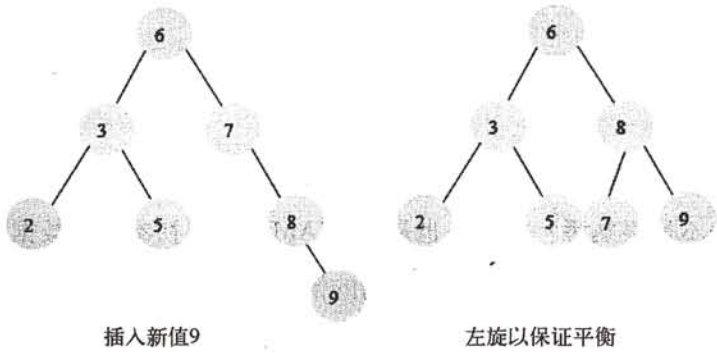


图5-4 插入键值9，平衡二叉树的变化

这里通过一次左旋操作就将插入后的树重新变为平衡的了。但是有的情况下可能需要多次旋转，如图5-5所示。

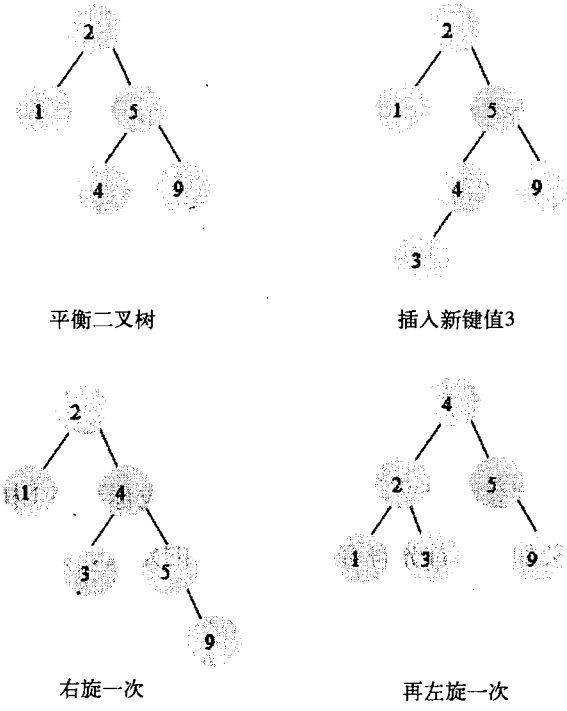


图5-5 需多次旋转的平衡二叉树

图5-4和图5-5中例举了向一颗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，不过这和插入没有本质的区别，它们都是通过左旋或者右旋来完成的。因此对于一颗平衡二叉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。

5.4 B+树

B+树和二叉树、平衡二叉树一样，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，但是在实际使用过程中几乎已经没有使用B树的情况了。

B+树的定义相信在任何一本数据结构书中都能找到，其定义十分复杂，相信这里我列出来只会让大家更困惑。因此只简要地介绍B+树：B+树是为磁盘或其他直接存取辅助设

备而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节点中，各叶节点指针进行连接。我们先来看一个B+树，其高度为2，每页可存放4条记录，扇出（fan out）为5。

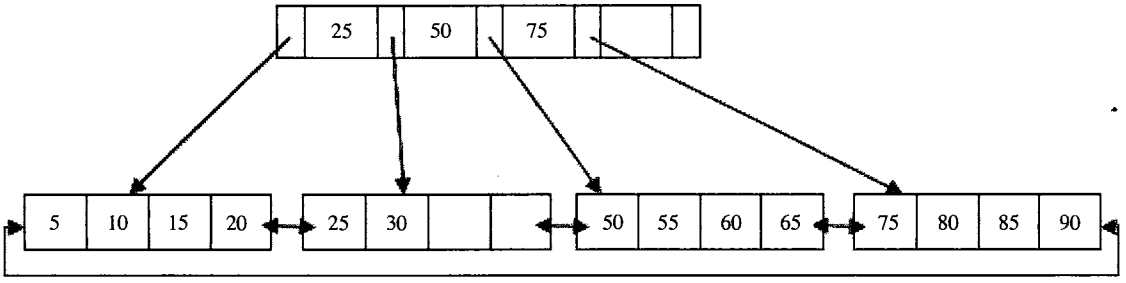


图5-6 一棵高度为2的B+树

从图5-6可以看出，所有记录都在叶节点中，并且是顺序存放的，如果我们从最左边的叶节点开始顺序遍历，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。

5.4.1 B+树的插入操作

B+树的插入必须保证插入后叶节点中的记录依然排序，同时需要考虑插入B+树的三种情况，每种情况都可能会导致不同的插入算法，如表5-1所示。

表5-1 B+树插入的3种情况

Leaf Page Full	Index Page Full	操作
No	No	直接将记录插入叶节点
Yes	No	1. 拆分Leaf Page 2. 将中间的节点放入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

我们用实例来分析B+树的插入，对于图5-6中的这颗B+树，我们插入28这个键值，发

现当前Leaf Page和Index Page都没有满，我们直接插入就可以了。之后得到图5-7。

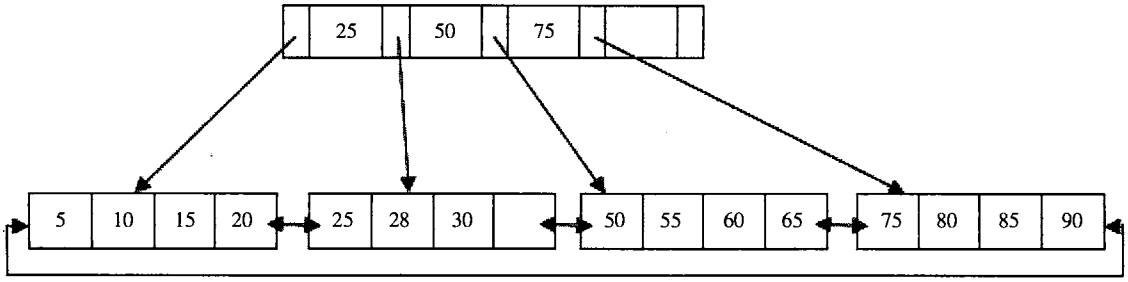


图5-7 插入键值28

这次我们再插入一条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5-1的第二种情况，这时插入Leaf Page后的情况为55、55、60、65、70。我们根据中间的值60拆分叶节点，可得到图5-8。

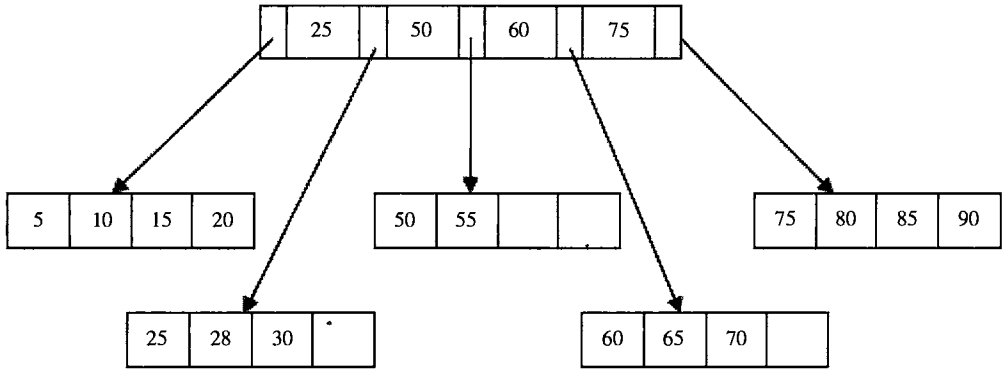


图5-8 插入键值70

因为图片显示的关系，这次我没有能在各叶节点加上双向链表指针。不过和图5-6、图5-7一样，它还是存在的。最后我们来插入记录95，这时符合表5-1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5-9所示：

可以看到，不管怎么变化，B+树总是会保持平衡。但是为了保持平衡，对于新插入的键值可能需要做大量的拆分页（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尽量减少页的拆分。因此，B+树提供了旋转（rotation）的功能。

旋转发生在Leaf Page已经满了、但是其左右兄弟节点没有满的情况下。这时B+树并不会急于去做拆分页的操作，而是将记录移到所在页的兄弟节点上。通常情况下，左兄弟被

首先检查用来做旋转操作，因此我们再来看图5-7的情况，这时我们插入键值70，其实B+树并不会急于去拆分叶节点，而是做旋转，得到如图5-10所示的操作。

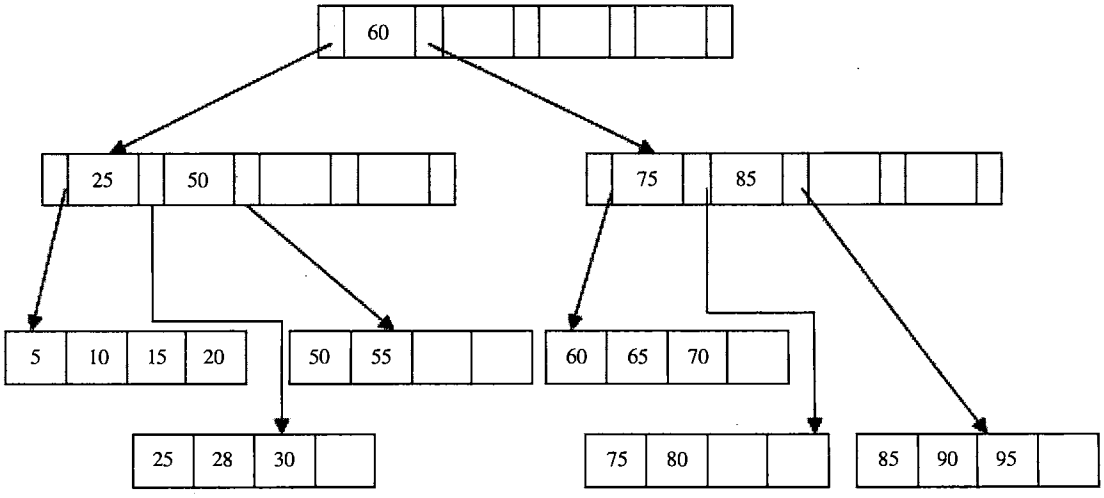


图5-9 插入键值95

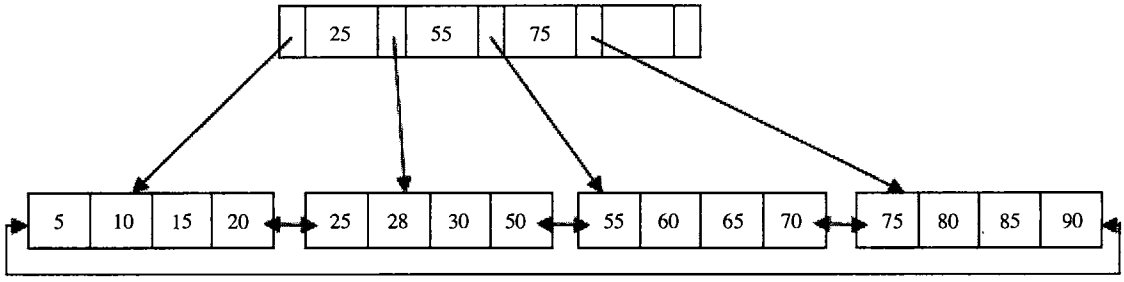


图5-10 B+树的旋转操作

从图5-10可以看到，采用旋转操作使B+树减少了一次页的拆分操作，而这时B+树的高度依然还是2。

5.4.2 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，50%是填充因子可设的最小值。B+树的删除操作同样必须保证删除后叶节点中的记录依然排序，同插入一样，B+树的删除操作同样需要考虑如表5-2所示的三种情况，与插入不同的是，删除根据填充因子的变化来衡量。

表5-2 B+树删除操作

Leaf Page Below Fill Factor	Index Page Below Fill Factor	操作
No	No	直接将记录从叶节点删除，如果该节点还是Index Page的节点，则用该节点的右节点代替
Yes	No	合并叶节点及其兄弟节点，同时更新Index Page
Yes	Yes	1. 合并叶节点及其兄弟节点 2. 更新Index Page 3. 合并Index Page及其兄弟节点

我们根据图5-9的B+树来进行删除操作。首先，删除键值为70的这条记录，该记录符合表5-2讨论的第一种情况，删除后可得到图5-11。

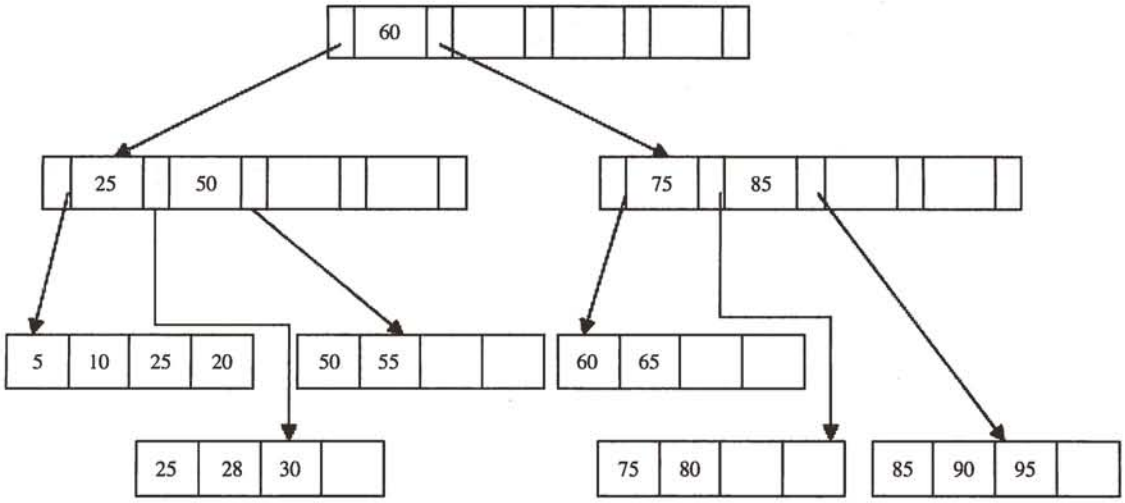


图5-11 删除键值70

接着我们删除键值为25的记录，这也是表5-2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25的值后，还应将25的右兄弟节点的28更新到Page Index中，最后可得到图5-12。

最后我们来看删除键值为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，最后得到图5-13。

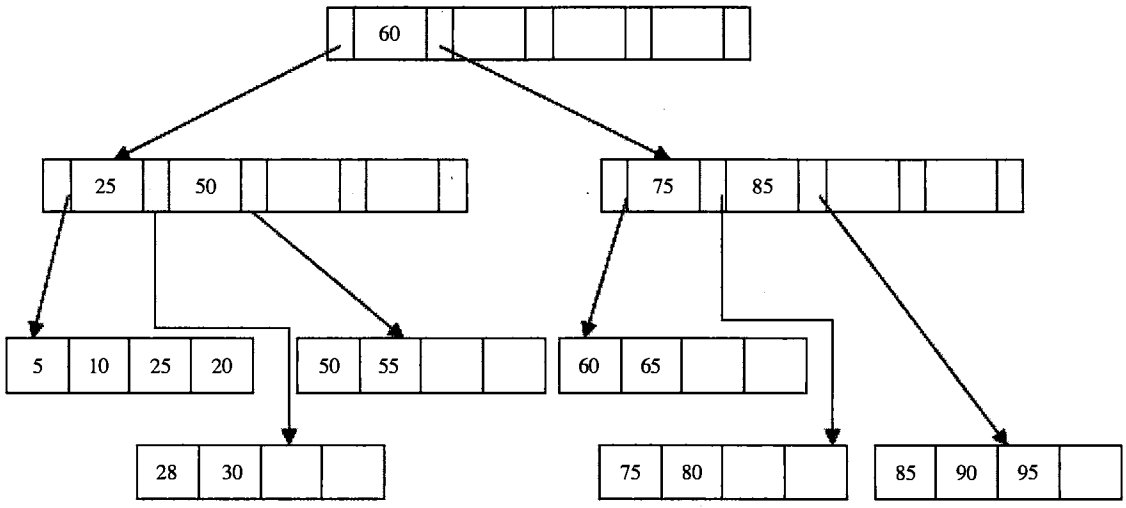


图5-12 删除键值25

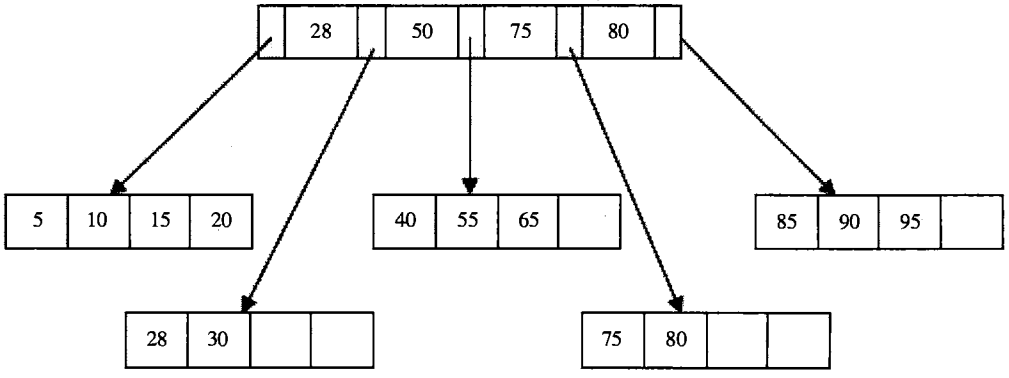


图5-13 删除键值60

5.5 B+树索引

前面我们讨论的都是B+树的数据结构及其一般操作，B+树索引其本质就是B+树在数据库中的实现，但是B+索引在数据库中有一个特点就是其高扇出性，因此在数据库中，B+树的高度一般都在2～3层，也就是对于查找某一键值的行记录，最多只需要2到3次IO，这倒不错。因为我们知道现在一般的磁盘每秒至少可以做100次IO，2～3次的IO意味着查询时间只需0.02～0.03秒。

数据库中的B+树索引可以分为聚集索引（clustered index）和辅助聚集索引（secondary

index）[⊖]，但是不管是聚集还是非聚集的索引，其内部都是B+树的，即高度平衡的，叶节点存放着所有的数据。聚集索引与非聚集索引不同的是，叶节点存放的是否是一整行的信息。

5.5.1 聚集索引

之前已经介绍过了，InnoDB存储引擎表是索引组织表，即表中数据按照主键顺序存放。而聚集索引就是按照每张表的主键构造一颗B+树，并且叶节点中存放着整张表的行记录数据，因此也让聚集索引的叶节点成为数据页。聚集索引的这个特性决定了索引组织表中数据也是索引的一部分。同B+树数据结构一样，每个数据页都通过一个双向链表来进行链接。

由于实际的数据页只能按照一颗B+树进行排序，因此每张表只能拥有一个聚集索引。在许多情况下，查询优化器非常倾向于采用聚集索引，因为聚集索引能够让我们在索引的叶节点上直接找到数据。此外，由于定义了数据的逻辑顺序，聚集索引能够特别快地访问针对范围值的查询。查询优化器能够快速发现某一段范围的数据页需要扫描。

现在我们来看一张表，我们以人为的方式让其每个页只能存放两个行记录，如：

```
mysql> create table t (a int not null primary key , b varchar(8000));
Query OK, 0 rows affected (0.30 sec)

mysql> insert into t select 1,repeat('a',7000);
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> insert into t select 2,repeat('a',7000);
Query OK, 1 row affected (0.02 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> insert into t select 3,repeat('a',7000);
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> insert into t select 4,repeat('a',7000);
Query OK, 1 row affected (0.05 sec)
```

⊖ 辅助聚集索引有时也称非聚集索引（non-clustered index）。

```
Records: 1  Duplicates: 0  Warnings: 0
```

可以看到，我们表的定义和插入方式使得目前每个页只能存放两个行记录，我们用py_innodb_page_info工具来分析表空间，可得：

```
[root@nineyou0-43 data]# py_innodb_page_info.py -v mytest/t.ibd
page offset 00000000, page type <File Space Header>
page offset 00000001, page type <Insert Buffer Bitmap>
page offset 00000002, page type <File Segment inode>
page offset 00000003, page type <B-tree Node>, page level <0001>
page offset 00000004, page type <B-tree Node>, page level <0000>
page offset 00000005, page type <B-tree Node>, page level <0000>
page offset 00000006, page type <B-tree Node>, page level <0000>
page offset 00000000, page type <Freshly Allocated Page>
Total number of page: 8:
Freshly Allocated Page: 1
Insert Buffer Bitmap: 1
File Space Header: 1
B-tree Node: 4
File Segment inode: 1
```

page level为0000的即是数据页，前面的章节也对数据页进行了分析，这不是我们当前所关注的部分。我们要分析的是page level为0001的页，该页是B+树的根，我们来看看索引的根页中存放的数据：

```
0000c000  c2 33 62 95 00 00 00 03  ff ff ff ff ff ff ff ff  |.3b.............|
0000c010  00 00 00 0a b6 8c ce 57  45 bf 00 00 00 00 00 00  |.......WE.......|
0000c020  00 00 00 00 00 f9 00 02  00 a2 80 05 00 00 00 00  |................|
0000c030  00 9a 00 02 00 02 00 03  00 00 00 00 00 00 00 00  |................|
0000c040  00 01 00 00 00 00 00 00  01 e2 00 00 00 f9 00 00  |................|
0000c050  00 02 00 f2 00 00 00 f9  00 00 00 02 00 32 01 00  |.............2..|
0000c060  02 00 1b 69 6e 66 69 6d  75 6d 00 04 00 0b 00 00  |...infimum......|
0000c070  73 75 70 72 65 6d 75 6d  00 10 00 11 00 0e 80 00  |supremum........|
0000c080  00 01 00 00 00 04 00 00  00 19 00 0e 80 00 00 02  |................|
0000c090  00 00 00 05 00 00 00 21  ff d6 80 00 00 04 00 00  |.......!........|
0000c0a0  00 06 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0000c0b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0000c0c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
......
0000fff0  00 00 00 00 00 70 00 63  73 d8 52 3a b6 8c ce 57  |.....p.cs.R:...W|
```

我们直接通过页尾的Page Directory来分析，从00 63可以知道该页中行开始的位置。接着通过Recorder Header来分析，0xc063开始的值为69 6e 66 69 6d 75 6d 00，就代表

infimum伪行记录。之前的5个字节01 00 02 00 1b就是Recorder Header，分析第4位到第8位的值1代表该行记录中只有一个记录（需要记住的是，InnoDB的Page Directory是稀疏的），即infimum记录本身。我们通过Recorder Header中最后的两个字节00 1b来判断下一条记录的位置，即c063+1b=c073，读取键值可得80 01，就是主键为1的键值（我们制定的INT是无符号的，因此二进制是0x8001，而不是0x0001），80 01后值00 00 00 04代表指向数据页的页号。以同样的方式，可以找到80 02和80 04这两个键值以及它们指向的数据页。

通过以上对于非数据页节点的分析，我们发现数据页上存放的是完整的行记录，而在非数据页的索引页中，存放的仅仅是键值以及指向数据页的偏移量，而不是一个完整的行记录。因此我们构造的这颗二叉树大致如图5-14所示。

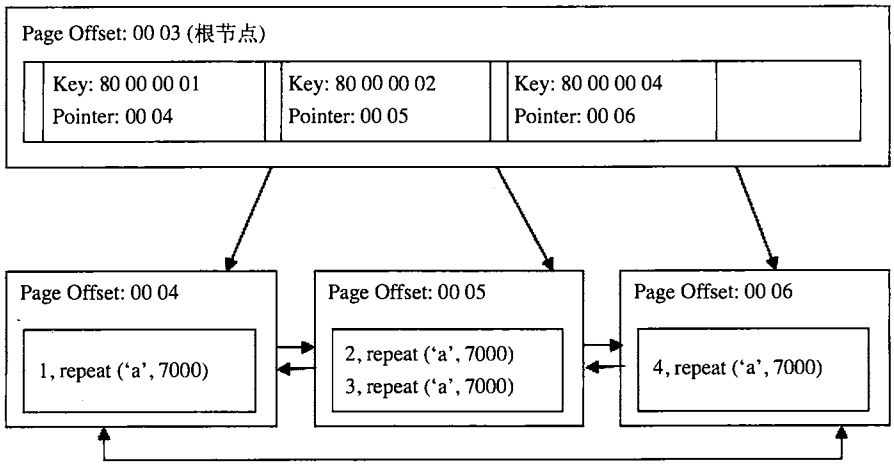


图5-14 B+树索引

许多数据库的文档会这样告诉读者：聚集索引按照顺序物理地存储数据。如果看图5-14，可能也会有这样的感觉。但是试想，如果聚集索引必须按照特定顺序存放物理记录的话，则维护成本即显得非常之高了。所以，聚集索引的存储并不是物理上的连续，相反是逻辑上连续的。这其中有两点：一是我们前面说过的页通过双向链表链接，页按照主键的顺序排列。另一点是每个页中的记录也是通过双向链表进行维护，物理存储上可以同样不按照主键存储。

聚集索引的另一个好处是，它对于主键的排序查找和范围查找速度非常快。叶节点的数据就是我们要查询的数据，如我们要查询一张注册用户的表，查询最后注册的10位用户，

由于B+树索引是双向链表的，我们可以快速找到最后一个数据页，并取出10条记录，我们用Explain进行分析，可得：

```
mysql> explain select * from Profile order by id limit 10\G;
*************************** 1. row ***************************
           id: 1
  select type: SIMPLE
        table: Profile
         type: index
possible_keys: NULL
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 10
        Extra:
1 row in set (0.00 sec)
```

另一个是范围查询（range query），如果要查找主键某一范围内的数据，通过叶节点的上层中间节点就可以得到页的范围，之后直接读取数据页即可，又如：

```
mysql> explain select * from Profile where id>10 and id<10000\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: Profile
         type: range
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 14868
        Extra: Using where
1 row in set (0.01 sec)
```

Explain得到了MySQL的执行计划（execute plan），并且rows列给出了一个查询结果的预估返回行数。要注意的是，rows代表的是一个预估值，不是确切的值，如果我们实际进行这句SQL的查询，可以看到实际上只有9 946行记录：

```
mysql> select count(*) from Profile where id>10 and id<10000;
+------------+
| count(*)   |
+------------+
|     9946   |
```

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

5.5.2 辅助索引

对于辅助索引（也称非聚集索引），叶级别不包含行的全部数据。叶节点除了包含键值以外，每个叶级别中的索引行中还包含了一个书签（bookmark），该书签用来告诉InnoDB存储引擎，哪里可以找到与索引相对应的行数据。因为InnoDB存储引擎表是索引组织表，因此InnoDB存储引擎的辅助索引的书签就是相应行数据的聚集索引键。图 5-15显示了InnoDB存储引擎中辅助索引与聚集索引的关系。

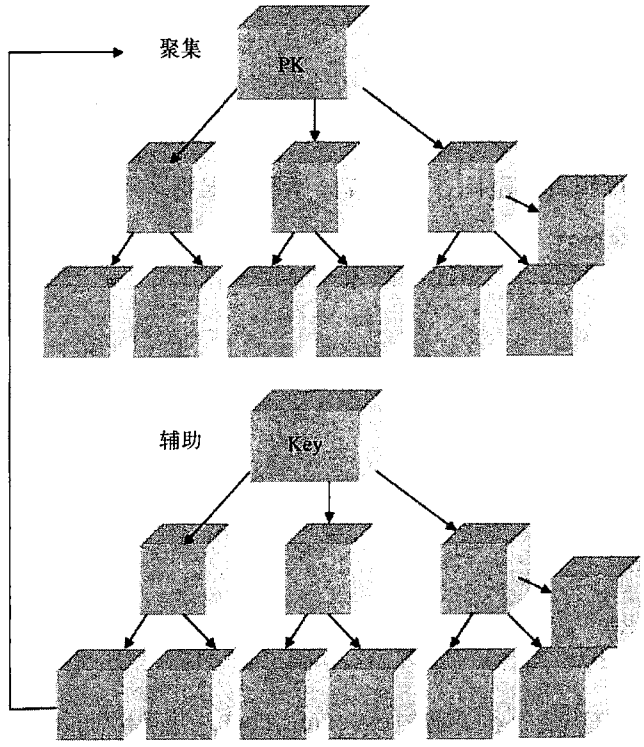


图5-15 辅助索引与聚集索引的关系

辅助索引的存在并不影响数据在聚集索引中的组织，因此每张表上可以有多个辅助索引。当通过辅助索引来寻找数据时，InnoDB存储引擎会遍历辅助索引并通过叶级别的指针获得指向主键索引的主键，然后再通过主键索引来找到一个完整的行记录。举例来说，如果在一颗高度为3的辅助索引树中查找数据，那么需要对这颗辅助索引遍历3次找到指定主

键；如果聚集索引树的高度同样为3，那么还需要对聚集索引进行3次查找，才能最终找到一个完整的行数据所在的页，因此一共需要6次逻辑IO来访问最终的一个数据页。

对于其他的一些数据库，如Microsoft SQL Server数据库，其表类型有一种不是索引组织表，称为堆表。在数据的存放按插入顺序方面，与MySQL的MyISAM存储引擎有些类似。堆表的特性决定了堆表上的索引都是非聚集的，但是堆表没有主键。因此这时书签是一个行标识符（row identifier, RID），可以用如“文件号：页号：槽号”的格式来定位实际的行。

也许有人会问，堆表的非聚集索引既然不需要再通过主键对聚集索引进行查找，那不是速度会更快吗？答案是也许，在某些只读的情况下，书签为行标识符方式的非聚集索引可能会比书签为主键方式的非聚集索引快。但是考虑在OLTP（OnLine Transaction Processing，在线事务处理）应用的情况下，表可能还需要发生插入、更新、删除等DML操作。当进行这类操作时，书签为行标识符方式的非聚集索引可能需要不断更新行标识符所指向数据页的位置，这时的开销可能就会大于书签为主键方式的非聚集索引了。

有的Microsoft SQL Server数据库DBA问过我这样的问题，为什么在SQL Server上还要使用索引组织表？堆表的书签性使得非聚集查找可以比主键书签方式更快，并且非聚集可能在一张表中存在多个，我们需要对多个非聚集索引的查找。而且对于非聚集索引的离散读取，索引组织表上的非聚集索引会比堆表上的聚集索引慢一些。

当然，在一些情况下，使用堆表的确会比索引组织表更快，但是我觉得大部分是由于存在于OLAP（On-Line Analytical Processing，在线分析处理）的应用。其次就是前面提到的，表中数据是否需要更新，并且更新会否影响到物理地址的变更。此外另一个不能忽视的是对于排序和范围查找，索引组织表可以通过B+树的中间节点就找到要查找的所有页，然后进行读取，而堆表的特性决定了这对其是不能实现的。最后，非聚集索引的离散读，的确是存在上述情况，但是一般的数据库都通过实现预读（read ahead）技术来避免多次的离散读操作。因此，具体是建堆表还是索引组织表，这取决于你的应用，不存在哪个更优的情况。这和InnoDB存储引擎好还是MyISAM存储引擎好的问题是一样的，具体情况具体分析。

接下来，我们通过阅读表空间文件来分析InnoDB存储引擎的非聚集索引，我们还是分

析上一小节所用的表t。不同的是，在表t上再建立一个列c，并对列c创建非聚集索引：

```
mysql> alter table t add c int not null;
Query OK, 4 rows affected (0.24 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> update t set c=0-a ;
Query OK, 4 rows affected (0.04 sec)
Rows matched: 4  Changed: 4  Warnings: 0

mysql> alter table t add key idx_c (c);
Query OK, 4 rows affected (0.28 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> show index from t\G;
*************************** 1. row ***************************
       Table: t
  Non_unique: 0
    Key_name: PRIMARY
Seq_in_index: 1
 Column_name: a
   Collation: A
 Cardinality: 2
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
*************************** 2. row ***************************
       Table: t
  Non_unique: 1
    Key_name: idx_c
Seq_in_index: 1
 Column_name: c
   Collation: A
 Cardinality: 2
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
2 rows in set (0.00 sec)

mysql> select a,c from t;
```

```
+---+----+
| a | c  |
+---+----+
| 4 | -4 |
| 3 | -3 |
| 2 | -2 |
| 1 | -1 |
+---+----+
4 rows in set (0.00 sec)
```

然后用py_innodb_page_info工具来分析表空间，可得：

```
[root@nineyou0-43 mytest]# py_innodb_page_info.py -v t.ibd
page offset 00000000, page type <File Space Header>
page offset 00000001, page type <Insert Buffer Bitmap>
page offset 00000002, page type <File Segment inode>
page offset 00000003, page type <B-tree Node>, page level <0001>
page offset 00000004, page type <B-tree Node>, page level <0000>
page offset 00000005, page type <B-tree Node>, page level <0000>
page offset 00000006, page type <B-tree Node>, page level <0000>
page offset 00000007, page type <B-tree Node>, page level <0000>
page offset 00000000, page type <Freshly Allocated Page>
Total number of page: 9:
Freshly Allocated Page: 1
Insert Buffer Bitmap: 1
File Space Header: 1
B-tree Node: 5
File Segment inode: 1
```

对比前一次我们的分析，可以看到这次多了一个页。分析page offset为4的页，该页为非聚集索引所在页，通过工具hexdump分析可得：

```
00010000  b9 aa 8e d0 00 00 00 04  ff ff ff ff ff ff ff ff  |................|
00010010  00 00 00 0a ec ea 4e 27  45 bf 00 00 00 00 00 00  |......N'E.......|
00010020  00 00 00 00 01 02 00 02  00 ac 80 06 00 00 00 00  |................|
00010030  00 a4 00 01 00 03 00 04  00 00 00 00 00 52 d4 8b  |.............R..|
00010040  00 00 00 00 00 00 00 00  01 f2 00 00 01 02 00 00  |................|
00010050  00 02 02 72 00 00 01 02  00 00 00 02 01 b2 01 00  |...r............|
00010060  02 00 41 69 6e 66 69 6d  75 6d 00 05 00 0b 00 00  |..Ainfimum......|
00010070  73 75 70 72 65 6d 75 6d  00 00 10 ff f3 7f ff ff  |supremum........|
00010080  ff 80 00 00 01 00 00 18  ff f3 7f ff ff fe 80 00  |................|
00010090  00 02 00 00 20 ff f3 7f  ff ff fd 80 00 00 03 00  |.... ...........|
000100a0  00 28 ff f3 7f ff ff fc  80 00 00 04 00 00 00 00  |.(..............|
......
```

```
00013ff0  00 00 00 00 00 70 00 63  f3 46 77 f2 ec ea 4e 27  |.....p.c.Fw...N'|
```

因为只有4行数据，并且列c只有4个字节，因此在一个非聚集索引页中即可完成，整理分析可得如图5-16所示的关系：

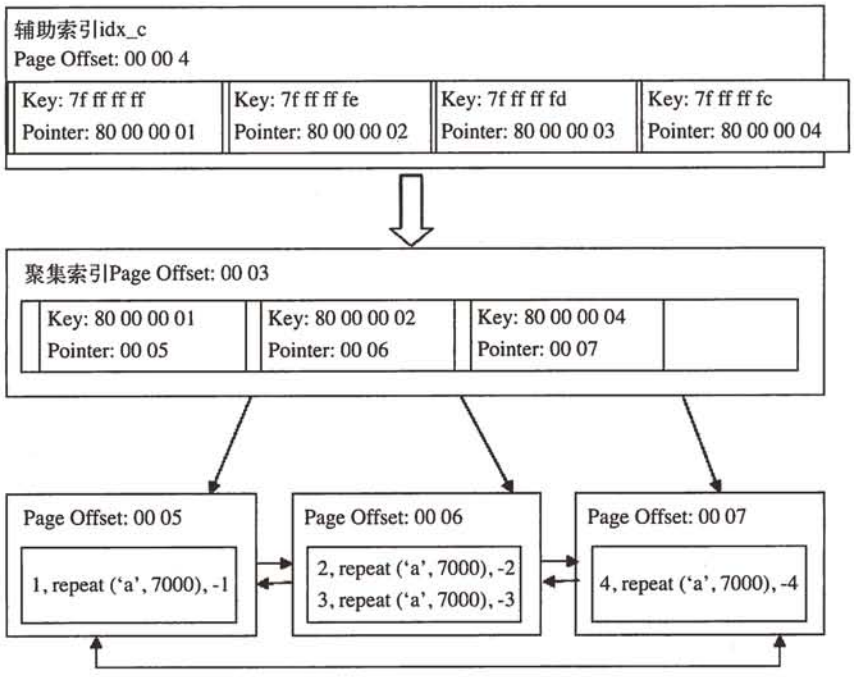


图5-16 辅助索引分析

图5-16显示了表t中辅助索引idx_c和聚集索引的关系。可以看到辅助索引的叶节点中包含了列c的值和主键的值。这里键值因为我特意设为负值，你会发现-1以7f ff ff ff的方式进行内部存储。7（0111）最高位为0，代表负值，实际的值应该取反后，加1，即得-1。

5.5.3 B+树索引的管理

索引的创建和删除可以通过两种方法，一种是ALTER TABLE，另一种是CREATE / DROP INDEX。ALTER TABLE创建索引的语法为：

```
ALTER TABLE tbl_name
| ADD {INDEX|KEY} [index_name]
[index_type] (index_col_name,...) [index_option] ...

ALTER TABLE tbl_name
DROP PRIMARY KEY
```

```
| DROP {INDEX|KEY} index_name
```

CREATE/DROP INDEX的语法同样很简单：

```
CREATE  [UNIQUE] INDEX index_name
[index_type]
ON tbl_name (index_col_name,...)

DROP INDEX index_name ON tbl_name
```

索引可以索引整个列的数据，也可以只索引一个列的开头部分数据，如前面我们创建的表t，b列为varchar（8000），但是我们可以只索引前100个字段，如：

```
mysql> alter table t add key idx_b (b(100));
Query OK, 4 rows affected (0.32 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

目前MySQL数据库存在的一个普遍问题是，所有对于索引的添加或者删除操作，MySQL数据库是先创建一张新的临时表，然后把数据导入临时表，删除原表，再把临时表重名为原来的表名。因此对于一张大表，添加和删除索引需要很长的时间。对于从Microsoft SQL Server或Oracle数据库的DBA来说，MySQL数据库的索引维护始终让他们非常苦恼。

InnoDB存储引擎从版本InnoDB Plugin开始，支持一种称为快速索引创建方法。当然这种方法只限定于辅助索引，对于主键的创建和删除还是需要重建一张表。对于辅助索引的创建，InnoDB存储引擎会对表加上一个S锁。在创建的过程中，不需要重建表，因此速度极快。但是在创建的过程中，由于上了S锁，因此创建的过程中该表只能进行读操作。删除辅助索引操作就更简单了，只需在InnoDB存储引擎的内部视图更新下，将辅助索引的空间标记为可用，并删除MySQL内部视图上对于该表的索引定义即可。

查看表中索引的信息可以使用SHOW INDEX语句。如我们来分析表t，之前先加一个联合索引，可得：

```
mysql> alter table t add key idx_a_b (a,c);
Query OK, 4 rows affected (0.31 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> show index from t\G;
*************************** 1. row ***************************
       Table: t
  Non_unique: 0
```

```
    Key_name: PRIMARY
Seq_in_index: 1
 Column_name: a
   Collation: A
 Cardinality: 2
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
*************************** 2. row ***************************
       Table: t
  Non_unique: 1
    Key_name: idx_b
Seq_in_index: 1
 Column_name: b
   Collation: A
 Cardinality: 2
    Sub_part: 100
      Packed: NULL
        Null: YES
  Index_type: BTREE
     Comment:
*************************** 3. row ***************************
       Table: t
  Non_unique: 1
    Key_name: idx_a_b
Seq_in_index: 1
 Column_name: a
   Collation: A
 Cardinality: 2
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
*************************** 4. row ***************************
       Table: t
  Non_unique: 1
    Key_name: idx_a_b
Seq_in_index: 2
 Column_name: c
   Collation: A
```

```
 Cardinality: 2
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
*************************** 5. row ***************************
       Table: t
  Non_unique: 1
    Key_name: idx_c
Seq_in_index: 1
 Column_name: c
   Collation: A
 Cardinality: 2
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
5 rows in set (0.00 sec)
```

因为在表t上有3个索引：一个主键索引，c列上的索引，和b列前100个字节构成的索引。接着我们来具体讲解每个列的含义：

- Table：索引所在的表名。
- Non_unique：非唯一的索引，可以看到primary key是0，因为必须是唯一的。
- Key_name：索引的名称，我们可以通过这个名称来DROP INDEX。
- Seq_in_index：索引中该列的位置，如果看联合索引idx_a_b就比较直观了。
- Column_name：索引的列
- Collation：列以什么方式存储在索引中。可以是'A'或者NULL。B+树索引总是A，即排序的。如果使用了Heap存储引擎，并且建立了Hash索引，这里就会显示NULL了。因为Hash根据Hash桶来存放索引数据，而不是对数据进行排序。
- Cardinality：非常关键的值，表示索引中唯一值的数目的估计值。Cardinality/表的行数应尽可能接近1，如果非常小，那么需要考虑是否还需要建这个索引。
- Sub_part：是否是列的部分被索引。如果看idx_b这个索引，这里显示100，表示我们只索引b列的前100个字符。如果索引整个列，则该字段为NULL。

- Packed：关键字如何被压缩。如果没有被压缩，则为NULL。
- Null：是否索引的列含有NULL值。可以看到idx_b这里为Yes。因为我们定义的b列允许NULL值。
- Index_type：索引的类型。InnoDB存储引擎只支持B+树索引，所以这里显示的都是BTREE。
- Comment：注释。

Cardinality值非常关键，优化器会根据这个值来判断是否使用这个索引。但是这个值并不是实时更新的，并非每次索引的更新都会更新该值，因为这样代价太大了。因此这个值是不太准确的，只是一个大概的值。上面显示的结果主键的Cardinality为2，但是很显然我们表中有4条记录，这个值应该是4。如果需要更新索引Cardinality的信息，可以使用ANALYZE TABLE命令。如：

```
mysql> analyze table t\G;
*************************** 1. row ***************************
   Table: mytest.t
      Op: analyze
Msg_type: status
Msg_text: OK
1 row in set (0.01 sec)

mysql> show index from t\G;
*************************** 1. row ***************************
       Table: t
  Non_unique: 0
    Key_name: PRIMARY
Seq_in_index: 1
 Column_name: a
   Collation: A
 Cardinality: 4
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
......
```

这时的Cardinality的值就对了。不过，在每个系统上可能得到的结果不一样，因为

ANALYZE TABLE现在还存在一些问题，可能会影响得到最后的结果。另一个问题是MySQL数据库对于Cardinality计数的问题，在运行一段时间后，可能会看到下面的结果：

```
mysql> show index from Profile\G;
*************************** 1. row ***************************
       Table: Profile
  Non_unique: 0
    Key_name: UserName
Seq_in_index: 1
 Column_name: username
   Collation: A
 Cardinality: NULL
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
```

Cardinality为NULL，在某些情况下可能会发生索引建立了、但是没有用到，或者explain两条基本一样的语句，但是最终出来的结果不一样。一个使用索引，另外一个使用全表扫描，这时最好的解决办法就是做一次ANALYZE TABLE的操作。因此我建议在一个非高峰时间，对应用程序下的几张核心表做ANALYZE TABLE操作，这能使优化器和索引更好地为你工作。

5.6 B+树索引的使用

5.6.1 什么时候使用B+树索引

并不是在所有的查询条件下出现的列都需要添加索引。对于什么时候添加B+树索引，我的经验是访问表中很少一部行时，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，即低选择性。如：

```
SELECT * FROM student WHERE sex='M'
```

对于性别，可取值的范围只有'M'、'F'。对上述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，即高选择性，则此时使用B+树索引是最适合的，例如姓名

字段，基本上在一个应用中都不允许重名的出现。

因此，当访问高选择性字段并从表中取出很少一部分行时，对这个字段添加B+树索引是非常有必要的。但是如果出现了访问字段是高选择性的，但是取出的行数据占表中大部分的数据时，这时MySQL数据库就不会使用B+树索引了，我们先来看一个例子：

```
mysql> show index from member\G;
*************************** 1. row ***************************
       Table: member
  Non_unique: 0
    Key_name: PRIMARY
Seq_in_index: 1
 Column_name: id
   Collation: A
 Cardinality: 4833601
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
*************************** 2. row ***************************
       Table: member
  Non_unique: 0
    Key_name: usernick
Seq_in_index: 1
 Column_name: usernick
   Collation: A
 Cardinality: 4833601
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
*************************** 3. row ***************************
       Table: member
  Non_unique: 1
    Key_name: idx_regdate
Seq_in_index: 1
 Column_name: registdate
   Collation: A
 Cardinality: 4833601
    Sub_part: NULL
      Packed: NULL
```

```
        Null:
  Index_type: BTREE
     Comment:
4 rows in set (0.00 sec)
```

表member大约有500万行数据。usernick字段上有一个唯一的索引。这时如果我们查找'David'这个用户时，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，这也符合我们前面提到的高选择性、选取表中很少行的原则。但是如果执行下面这条语句：

```
mysql> explain select * from member where usernick>'David'\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: member
         type: ALL
possible_keys: usernick
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 4833601
        Extra: Using where
1 row in set (0.00 sec)
```

可以看到possible_keys依然是usernick，但是实际优化器使用的索引key显示的是NULL。为什么？因为这不符合我们前面说的原则，虽然usernick这个字段的值是高选择性的，但是我们取出的行占了表中很大的一部分。

```
mysql> select @a:=count(id) from member where usernick>'David';
+--------------------+
| @a:=count(id)  |
+--------------------+
|       4544637 |
+--------------------+
1 row in set (3.12 sec)

mysql> select @b:=count(id) from member;
+------------------+
| @b:=count(id) |
+------------------+
|      4827542 |
+------------------+
1 row in set (2.20 sec)

mysql> select @a/@b;
+-----------+
| @a/@b  |
+-----------+
| 0.9414  |
+----------+
1 row in set (0.00 sec)
```

可以看到我们将取出表中94%的行，因此优化器没有使用索引。也许有人看到这里会问，谁会执行这句话啊？查找姓名大于David的字段，这种情况几乎不存在。的确如此，但是我们来考虑member表上的registdate字段（代表用户的注册时间），该字段是日期类型，字段上有一个regdate的非唯一索引。我们来看下面两条语句的执行计划：

```
mysql> explain select * from member where registdate<'2006-04-23'\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: member
         type: range
possible_keys: idx_regdate
          key: idx_regdate
      key_len: 8
          ref: NULL
         rows: 788696
        Extra: Using where
1 row in set (0.00 sec)
```

```
mysql> explain select * from member where registdate<'2006-04-24'\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: member
         type: ALL
possible_keys: idx_regdate
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 4833601
        Extra: Using where
1 row in set (0.00 sec)
```

查找用户注册时间小于某个时间的SQL语句。出人意料的是，只是相差了1天，2条SQL语句的执行计划竟然不同。在执行第二条SQL语句的时候，虽然同样可以使用idx_regdate索引，但是优化器却没有使用该索引，而是对其全表进行扫描。MySQL数据库的优化器会通过EXPLAIN的rows字段预估查询可能得到的行，如果大于某一个值，则B+树会选择全表的扫表。至于这个值，根据我的经验（并没有在源代码中得到验证）一般在20%。即当取出的数据量超过表中数据的20%，优化器就不会使用索引，而是进行全表的扫表。

但是，预估的返回行数的值是不准确的，可以看到优化器判断注册日期小于2006-04-23的行为788 696，但实际得到的却是：

```
mysql> select count(id) from member where registdate<'2006-04-23';
+------------+
| count(id)  |
+------------+
|   523046 |
+------------+
1 row in set (0.34 sec)
```

实际却只有523 046行，少了33%。这可能对于优化器的选择产生一定的后果，如果我们对比强制使用索引和使用优化器选择的全表扫描来查询注册日期小于2006-04-24的数据，最终会发现：

```
mysql> select id,userid,sex,registdate into outfile 'a' from member force
index(idx_regdate) where registdate<'2006-04-24';
```

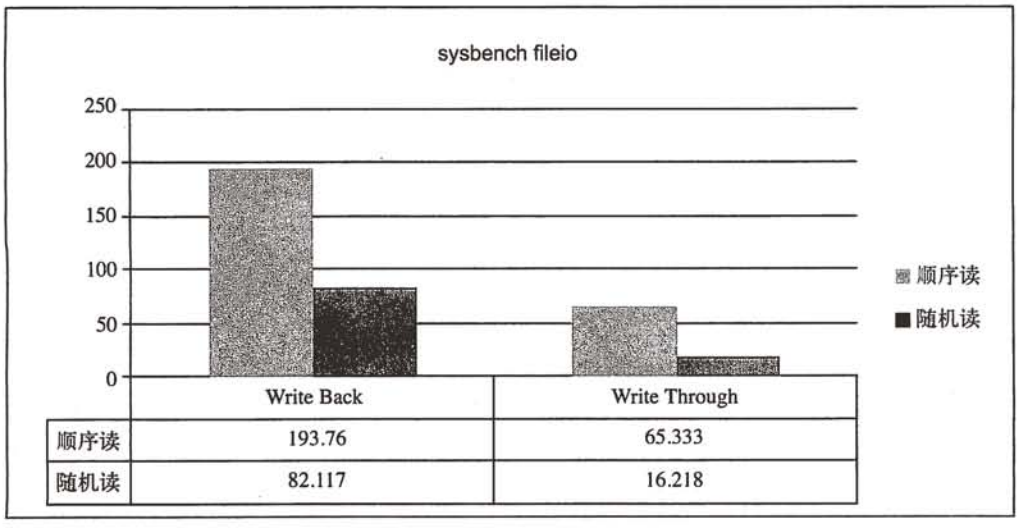
```
Query OK, 551664 rows affected (4.15 sec)

mysql> select id,userid,sex,registdate into outfile 'b' from member where
registdate<'2006-04-24';
Query OK, 551664 rows affected (18.70 sec)
```

第一句SQL语句我们强制使用idx_regdate索引，所用的时间为4.15秒，根据优化器选择的全表扫方式，执行第二句SQL语句却需要18.7秒。因此优化器的选择并不完全是正确的，有时你更应该相信自己的判断。

5.6.2 顺序读、随机读与预读取

任何时候Why都比What重要，我已经告诉了你索引使用的原则，即高选择、取出表中少部分的数据。但是为什么只能是少部分数据？在知道为什么之前，我想让你了解两个概念——顺序读和随机读。

顺序读（Sequntial Read）是指顺序地读取磁盘上的块（Block）；随机读（Random Read）是指访问的块不是连续的，需要磁盘的磁头不断移动。当前传统机械磁盘的瓶颈之一就是随机读取的速度较低。图5-17是用sysbench测试的顺序读和随机读的速度对比，同时对比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RAID 10。测试文件大小为2GB，块大小64KB。



	Write Back	Write Through
顺序读	193.76	65.333
随机读	82.117	16.218

图5-17 顺序读与随机读的对比

可以看到，不管是否开启RAID卡的Write Back功能，磁盘的随机读性能都远远小于顺序读的性能。通过图5-17的比较，我们也大致可以知道Write Back相对于Write Through的性能提升。

在数据库中，顺序读是指根据索引的叶节点数据就能顺序地读取所需的行数据。这个顺序只是逻辑地顺序读，在物理磁盘上可能还是随机读取。但是相对来说，物理磁盘上的数据还是比较顺序的，因为是根据区来管理的，区是64个连续页。如根据主键进行读取，或许通过辅助索引的叶节点就能读取到数据。

随机读，一般是指访问辅助索引叶节点不能完全得到结果的，需要根据辅助索引叶节点中的主键去找实际行数据。因为一般来说，辅助索引和主键所在的数据段不同，因此访问是随机的方式。前一小节中的SQL语句select id,userid,sex,registdate into outfile 'a' from member force index（idx_regdate） where registdate<'2006-04-24';就是一句典型的随机读取。而正是因为读取的方式是随机的，并且随机读的性能会远低于顺序读，因此优化器才会选择全表的扫描方式，而不是去走idx_regdate这个辅助索引。

为了提高读取的性能，InnoDB存储引擎引入了预读取技术（read ahead或者prefetch）。预读取是指通过一次IO请求将多个页预读取到缓冲池中，并且估计预读取的多个页马上会被访问。传统的IO请求每次只读取1个页，在传统机械硬盘较低的IOPS下，预读技术可以大大提高读取的性能。

InnoDB存储引擎有两个预读方法，称为随机预读取（random read ahead）和线性（linear read ahead）预读取。随机预读是指当一个区（64个连续页）中13个页也在缓冲区中，并在LRU列表的前端（即页是频繁地被访问），则InnoDB存储引擎会将这个区中剩余的所有页预读到缓冲区。线性预读基于缓冲池中页的访问模式，而不是数量。如果一个区中的24个页都被顺序地访问了，则InnoDB存储引擎会读取下一个区的所有页。

但是，InnoDB存储引擎的预读取技术在实际测试下却非常糟糕，我曾写过一个patch来禁用预读取功能，这个patch很简单：

```
---  mysql/mysql-5.1.35/storage/innobase/srv/srv0srv.c      2009-08-03
16:05:53.000000000 +0800
+++ mysql-5.1.35/storage/innobase/srv/srv0srv.c 2009-05-14 19:35:20.000000000 +0800
@@ -49,9 +49,6 @@
 #include "row0mysql.h"
```

```
 #include "ha_prototypes.h"

-/* Set innodb read ahead level */
-ulong  srv_innodb_read_ahead_level = 3; /* 0: disable 1: random  2: sequential  3: Both */
-
 /* This is set to TRUE if the MySQL user has set it in MySQL; currently
 affects only FOREIGN KEY definition parsing */
 ibool  srv_lower_case_table_names      = FALSE;

---  mysql/mysql-5.1.35/storage/innobase/buf/buf0rea.c       2009-08-03
16:05:53.000000000 +0800
+++ mysql-5.1.35/storage/innobase/buf/buf0rea.c 2009-05-14 19:35:12.000000000 +0800
@@ -23,7 +23,6 @@
 extern ulint srv_read_ahead_rnd;
 extern ulint srv_read_ahead_seq;
 extern ulint srv_buf_pool_reads;
-extern ulong srv_innodb_read_ahead_level;

 /* The size in blocks of the area where the random read-ahead algorithm counts
 the accessed pages when deciding whether to read-ahead */
@@ -177,9 +176,6 @@
        ulint           err;
        ulint           i;

-       if (srv_innodb_read_ahead_level != 3 && srv_innodb_read_ahead_level != 1 )
-               return(0);
-
        if (srv_startup_is_before_trx_rollback_phase) {
                /* No read-ahead to avoid thread deadlocks */
                return(0);
@@ -390,9 +386,6 @@
        ulint           err;
        ulint           i;

-       if (srv_innodb_read_ahead_level != 3 && srv_innodb_read_ahead_level != 2 )
-               return(0);
-
        if (srv_startup_is_before_trx_rollback_phase) {
                /* No read-ahead to avoid thread deadlocks */
                return(0);
```

对比数据库TPC-C测试性能，发现TPC-C的结果禁用预读取的性能比启用预读取的性能提高了10%。InnoDB存储引擎官方也发现了这个问题，从InnoDB Plugin 1.0.4开始，随

机访问的预读取被取消了，而线性的预读取还是保留了，并且加入了innodb_read_ahead_threshold参数。该参数表示一个区中的多少页被顺序访问时，InnoDB存储引擎才启用预读取，即预读下一个区的所有页。参数innodb_read_ahead_threshold的默认值为56，即当一个区中56个页都已被访问过并且访问模式是顺序的，则预读取下一个区的所有页。

```
mysql> show variables like 'innodb_read_ahead_threshold';
+-----------------------------------+-------+
| Variable_name                     | Value |
+-----------------------------------+-------+
| innodb_read_ahead_threshold | 56    |
+-----------------------------------+--------+
1 row in set (0.00 sec)
```

我关注的另一个问题是固态硬盘，固态硬盘的接口规范、定义、功能和使用等方面与传统机械硬盘相同，但是它们的内部构造完全不同，固态硬盘没有读写磁头，读取数据不需要围绕中心轴旋转，因此，它的随机读性能得到了质的飞跃。在使用固态硬盘的情况下，优化器的20%选择原理可能就不怎么准确了，我们应该更充分地利用固态硬盘的特性。当然，这不只是InnoDB存储引擎遇到的问题，对于其他数据库，目前都存在没有充分利用固态硬盘特性的情况。相信随着固态硬盘的普及，各数据库厂商会加快这一方面的优化。

5.6.3 辅助索引的优化使用

前面已经提到了，辅助索引的叶节点包含有主键，但是辅助索引的叶并不包含完整的行信息。因此，InnoDB存储引擎总是会先从辅助索引的叶节点判断是否能得到所需的数据。让我们来看一个例子：

```
mysql> create table t (a int not null, b varchar(20), primary key(a),key(b));
Query OK, 0 rows affected (0.19 sec)

mysql> insert into t select 1,'kangaroo';
Query OK, 1 row affected (0.02 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> insert into t select 2,'dolphin';
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

```
mysql> insert into t select 3,'dragon';
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> insert into t select 4,'antelope';
Query OK, 1 row affected (0.02 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

如果执行SELECT * FROM t，估计很多人以为会得到如下的结果：

```
mysql>  select * from t order by a\G;
*************************** 1. row ***************************
a: 1
b: kangaroo
*************************** 2. row ***************************
a: 2
b: dolphin
*************************** 3. row ***************************
a: 3
b: dragon
*************************** 4. row ***************************
a: 4
b: antelope
4 rows in set (0.01 sec)
```

但是实际执行的结果却是：

```
mysql>  select * from t\G;
*************************** 1. row ***************************
a: 4
b: antelope
*************************** 2. row ***************************
a: 2
b: dolphin
*************************** 3. row ***************************
a: 3
b: dragon
*************************** 4. row ***************************
a: 1
b: kangaroo
4 rows in set (0.00 sec)
```

这就是我们前面所提到的，因为辅助索引中包含了主键a的值，因此访问b列上的辅助索引就能得到a值，那这样就可以得到表中所有的数据。并且通常情况下，一个辅助索引

页中能存放的数据比主键页上存放的数据多，因此优化器选择了辅助索引，如果我们解释这句SQL语句，可得到如下结果：

```
mysql> explain select * from t\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t
         type: index
possible_keys: NULL
          key: b
      key_len: 63
          ref: NULL
         rows: 4
        Extra: Using index
1 row in set (0.00 sec)
```

可以看到，优化器最终选择的索引是b，如果想得到对列a排序的结果，你还需对其进行ORDER BY操作，这样优化器会直接走主键，避免对a列的排序操作。如：

```
mysql> explain select * from t order by a\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t
         type: index
possible_keys: NULL
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 4
        Extra:
1 row in set (0.00 sec)

mysql>  select * from t order by a\G;
*************************** 1. row ***************************
a: 1
b: kangaroo
*************************** 2. row ***************************
a: 2
b: dolphin
*************************** 3. row ***************************
a: 3
```

```
b: dragon
*************************** 4. row ***************************
a: 4
b: antelope
4 rows in set (0.01 sec)
```

或者强制使用主键来得到结果：

```
mysql> select * from t force index(PRIMARY)\G;
*************************** 1. row ***************************
a: 1
b: kangaroo
*************************** 2. row ***************************
a: 2
b: dolphin
*************************** 3. row ***************************
a: 3
b: dragon
*************************** 4. row ***************************
a: 4
b: antelope
4 rows in set (0.00 sec)
```

5.6.4 联合索引

联合索引是指对表上的多个列做索引。前面我们讨论的情况，都是只对表上的一个列进行了索引。联合索引的创建方法与之前介绍的一样，如：

```
mysql> alter table t add key idx_a_b(a,b);
Query OK, 4 rows affected (0.25 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

什么时候需要使用联合索引呢？在讨论这个之前，我们要来看一下联合索引内部的结果。从本质上来说，联合索引还是一颗B+树，不同的是联合索引的键值的数量不是1，而是大于等于2。我们来讨论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5-18所示。

从图5-18可以看到多个键值的B+树情况，其实和我们之前讨论的单个键值没有什么不同，键值都是排序的，通过叶节点可以逻辑上顺序地读出所有数据，就上面的例子来说即：(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。

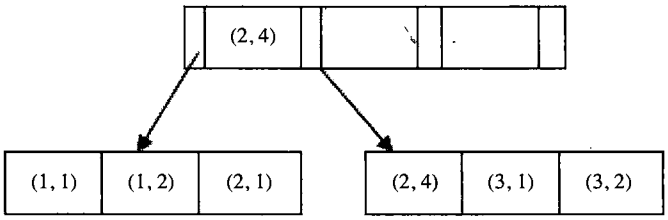


图5-18 多个键值的B+树

因此，对于查询SELECT * FROM TABLE WHERE a=xxx and b=xxx，显然是可以使用(a,b) 的这个联合索引。对于单个的a列查询SELECT * FROM TABLE WHERE a=xxx也是可以使用这个 (a,b) 索引。但是对于b列的查询SELECT * FROM TABLE WHERE b=xxx，不可以使用这颗B+树索引。可以看到叶节点上的b值为1、2、1、4、1、2，显然不是排序的，因此对于b列的查询使用不到 (a,b) 的索引。

联合索引的第二个好处是，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避免多一次的排序操作，因为索引本身在叶节点已经排序了。

```
create table buy_log ( userid int unsigned not null, buy_date date );

mysql> insert into buy_log values ( 1,'2009-01-01');
Query OK, 1 row affected (0.02 sec)

mysql> insert into buy_log values ( 2,'2009-01-01');
Query OK, 1 row affected (0.05 sec)

mysql> insert into buy_log values ( 3,'2009-01-01');
Query OK, 1 row affected (0.03 sec)

mysql> insert into buy_log values ( 1,'2009-02-01');
Query OK, 1 row affected (0.01 sec)

mysql> insert into buy_log values ( 3,'2009-02-01');
Query OK, 1 row affected (0.01 sec)

mysql> insert into buy_log values ( 1,'2009-03-01');
Query OK, 1 row affected (0.01 sec)

mysql> insert into buy_log values ( 1,'2009-04-01');
Query OK, 1 row affected (0.01 sec)
```

```
mysql> alter table buy_log add key ( userid );
Query OK, 7 rows affected (0.43 sec)
Records: 7  Duplicates: 0  Warnings: 0

mysql> alter table buy_log add key ( userid,buy_date );
Query OK, 7 rows affected (0.50 sec)
Records: 7  Duplicates: 0  Warnings: 0
```

我们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，优化器的选择是：

```
mysql> explain select * from buy_log where userid=2\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: buy_log
         type: ref
possible_keys: userid,userid_2
          key: userid
      key_len: 4
          ref: const
         rows: 1
        Extra:
1 row in set (0.00 sec)
```

可以看到possible_keys这里有两个索引可以使用，分别是单个的userid索引和userid、buy_date的联合索引。但是优化器最终的选择是userid，因为该叶节点包含单个键值，因此一个页能存放的记录应该更多。接着看以下的查询，我们假定要取出userid=1的最近3次购买记录，并分析使用单个索引和联合索引的区别：

```
mysql> explain select * from buy_log where userid=1 order by buy_date desc limit 3\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: buy_log
         type: ref
possible_keys: userid,userid_2
          key: userid_2
      key_len: 4
          ref: const
         rows: 3
        Extra: Using where; Using index
```

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

同样，对于上述的SQL语句都可以使用userid和userid,buy_date的索引。但是这次优化器使用了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。如果我们强制使用userid的单个索引，会得到如下结果：

```
mysql> explain select * from buy_log force index(userid) where userid=1 order by
buy_date desc limit 3\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: buy_log
         type: ref
possible_keys: userid
          key: userid
      key_len: 4
          ref: const
         rows: 3
        Extra: Using where; Using filesort
1 row in set (0.00 sec)
```

在Extra这里，我们可以看到Using filesort，filesort是指排序，但是并不是在文件中完成。我们可以对比执行：

```
mysql> show status like 'sort_rows';
+-----------------+-------+
| Variable_name | Value |
+-----------------+-------+
| Sort_rows     | 7     |
+-----------------+-------+
1 row in set (0.00 sec)

mysql> select * from buy_log force index(userid) where userid=1 order by
buy_date desc limit 3;
+--------+---------------+
| userid  | buy_date    |
+--------+---------------+
|      1  | 2009-04-01 |
|      1  | 2009-03-01 |
|      1  | 2009-02-01 |
+--------+---------------+
3 rows in set (0.00 sec)
```

```
mysql> show status like 'sort_rows';
+-----------------+-------+
| Variable_name | Value |
+-----------------+-------+
| Sort_rows     | 10    |
+-----------------+-------+
1 row in set (0.00 sec)
```

可以看到增加了排序的操作，但是如果使用userid、buy_date的联合索引userid_2，就不会有这一次的额外操作了，如：

```
mysql> show status like 'sort_rows';
+-----------------+-------+
| Variable_name | Value |
+-----------------+-------+
| Sort_rows     | 10    |
+-----------------+-------+
1 row in set (0.00 sec)

mysql> select * from buy_log where userid=1 order by buy_date desc limit 3;
+--------+---------------+
| userid  | buy_date    |
+--------+---------------+
|      1 | 2009-04-01 |
|      1 | 2009-03-01 |
|      1 | 2009-02-01 |
+--------+---------------+
3 rows in set (0.00 sec)

mysql> show status like 'sort_rows';
+-----------------+-------+
| Variable_name | Value |
+-----------------+-------+
| Sort_rows     | 10    |
+-----------------+-------+
1 row in set (0.00 sec)
```

5.7 哈希算法

InnoDB存储引擎中自适应哈希索引使用的是散列表（Hash Table）的数据结构。但是散列表不只存在于自适应哈希中，在每个数据库中都存在。设想一个问题，当前我的内存

为128G，我怎么得到内存中的某一个被缓存的页呢？内存中查询速度很快，但是也不可能遍历所有内存。这时，对于字典操作，O（1）的散列技术就能有很好的用武之地。

5.7.1 哈希表

哈希表（Hash Table）也称散列表，由直接寻址表改进而来，所以我们先来看直接寻址表。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（或称槽或桶）对应全域U中的一个关键字。图5-19说明这个方法，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。

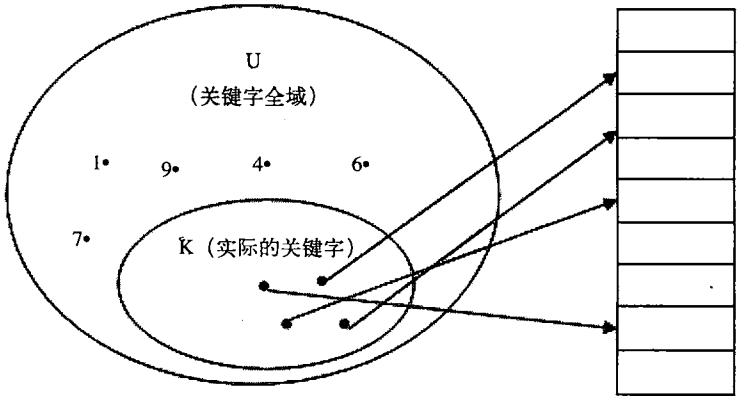


图5-19 直接寻址表

直接寻址技术存在一个很明显的问题：如果域U很大，在典型计算机的可用容量限制下，要在机器中存储大小为U的表T就有点不实际，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来说很小，因而分配给T的大部分空间都要浪费掉。

因此，哈希表出现了，在哈希方式下，该元素处于h（k）中，亦即利用哈希函数h、根据关键字k计算出槽的位置。函数h将关键字域U映射到哈希表T[0..m-1]的槽位上，见图5-20所示。

⊖ 此处的m不是一个很大的数。

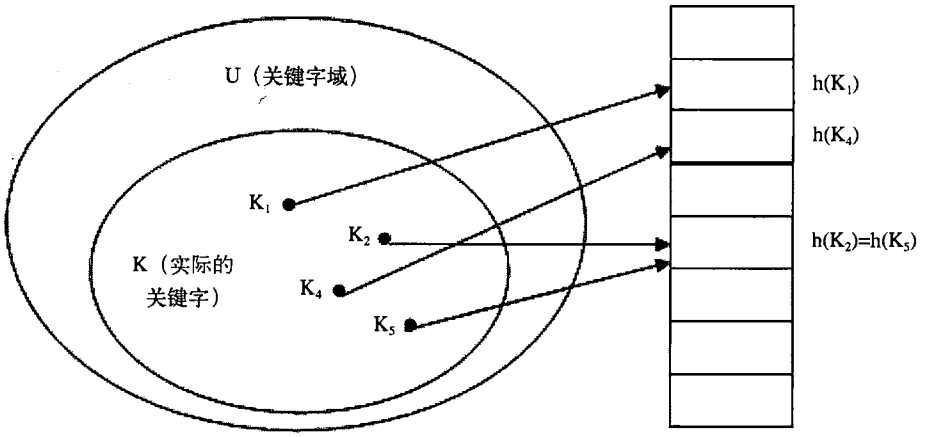


图5-20 哈希表

哈希表技术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5-20所示的两个关键字可能映射到同一个槽上。一般将这种情况称之为发生了碰撞（collision）。数据库中一般采用最简单的碰撞解决技术，称之为链接法（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5-21所示。槽j中有一个指针，他指向由所有散列到j的元素构成的链表的头；如果不存在这样的元素，则j中为NULL。

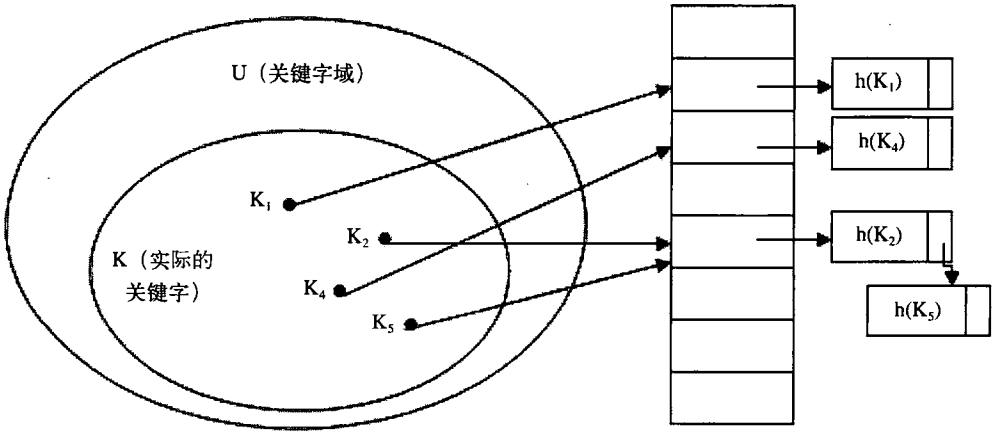


图5-21 通过链表法解决碰撞的哈希表

最后要考虑的是哈希函数了，哈希函数h必须很好地进行散列。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。一般来说，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。

在用来设计哈希函数的除法散列法中，通过取k除以m的余数，来将关键字k映射到m个槽的某一个去。即哈希函数为：

$$h(k) = k \bmod m$$

5.7.2 InnoDB存储引擎中的哈希算法

InnoDB存储引擎使用哈希算法对字典进行查找，其冲突机制采用链表方式，哈希函数采用除法散列方式。对于缓冲池页的哈希表来说，在缓冲池中的Page页都有一个chain指针，它指向相同哈希函数值的页。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的设置大小为10MB，则共有640个16KB的页。那对于缓冲池页内存的哈希表来说，需要分配640 × 2=1280个槽，但是1280不是质数，需要取比1 280略大的一个质数，应该是1399，所以在启动时会分配1399个槽的哈希表，用来哈希查询所在缓冲池中的页。哈希表本身需要20个字节，每个槽需要4个字节，因此一共需要20+4 × 1399=5616个字节。其中哈希表的20个字节从innodb_additional_mem_pool_size中进行分配，4 × 1399=5596个字节从系统申请分配。因此在对InnoDB存储引擎进行内存分配规划时，也应该规划好哈希表这部分内存，这部分内存一般从系统分配，没有参数可以控制。对于前面我们说的128GB的缓冲池内存，则分配的哈希表和槽一共需要差不多640MB的额外内存空间。

那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。

5.7.3 自适应哈希索引

自适应哈希索引采用之前，我们讨论哈希表的方式实现。不同的是，这又是数据库自

己创建并使用的，DBA本身并不能对其进行干预。当在配置文件中启用了参数innodb_adaptive_hash_index后，数据库启动时会自动创建槽数为innodb_buffer_pool_ size/256个的哈希表。例如，对当前参数innodb_buffer_pool_size设置为10MB，则启动时InnoDB存储引擎会创建一个有10M/256=40 960个槽的自适应哈希表。

自适应哈希索引经哈希函数映射到一个哈希表中，因此自适应哈希索引对于字典类型的查找非常快速，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，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了，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，如select * from table where index_col = 'xxx'，而对于其他查找类型，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。hash searches : non-hash searches可以大概知道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，所以这里的信息只供我们参考而已。不过我们可以通过参数innodb_adaptive_hash_index来禁用或启动此特性，默认为开启。

5.8 小结

这一章中，我们介绍了一些常用的数据结构，如二分查找树、平衡树、B+树、直接寻址表和哈希表。我们还从数据结构的角度，切入数据库中常见的B+树索引和哈希索引的使用，并从内部机制上讨论了使用上述索引的环境和优化方法。

第6章 锁

开发多用户、数据库驱动的应用时，最大的一个难点是：一方面要最大程度地利用数据库的并发访问，另外一方面还要确保每个用户能以一致的方式读取和修改数据。为此就有了锁（locking）机制，这也是数据库系统区别于文件系统的一个关键特性。InnoDB存储引擎较之MySQL数据库的其他存储引擎，在这方面技高一筹，其实现方式非常类似于Oracle数据库。只有正确了解内部这些锁的机制，才能完全发挥InnoDB存储引擎在锁方面的优势。

在这一章中，我们将详细介绍InnoDB存储引擎对表中数据的锁定，同时分析InnoDB存储引擎会以怎样的粒度锁定数据。本章还对MyISAM、Oracle、SQL Server之间的锁进行了比较，主要是为了消除关于行级锁的一个“神话”：人们认为行级锁总会增加开销。实际上，只有当实现本身会增加开销时，行级锁才会增加开销。InnoDB存储引擎不需要锁升级，因为一个锁和多个锁的开销是相同的。

6.1 什么是锁

锁是数据库系统区别于文件系统的一个关键特性。锁机制用于管理对共享资源的并发访问[⊖]。InnoDB存储引擎会在行级别上对表数据上锁，这固然不错。不过InnoDB存储引擎也会在数据库内部其他多个地方使用锁，从而允许对多种不同资源提供并发访问。例如，操作缓冲池中的LRU列表，删除、添加、移动LRU列表中的元素，为了保证一致性，必须有锁的介入。数据库系统使用锁是为了支持对共享资源进行并发访问，提供数据的完整性和一致性。

另一点需要理解的是，虽然现在数据库系统做得越来越类似，但是有多少种数据库，

⊖ 注意：这里说的是“共享资源”，而不仅仅是“行记录”。

就可能有多少种锁的实现方法。在SQL语法层面，因为SQL标准的存在，要熟悉多个关系数据库系统并不是一件难事。而对于锁，你可能对某个特定的关系数据库系统的锁定模型有一定的经验，但这并不意味着你知道其他数据库。在使用InnoDB存储引擎之前，我还使用过MySQL的MyISAM和NDB Cluster存储引擎；在使用MySQL之前，我还使用过Microsoft SQL Server、Oracle等数据库，但它们对于锁的实现完全不同。

对于MyISAM引擎来说，其锁是表锁。并发情况下的读没有问题，但是并发插入时的性能就要差一些了，若插入是在“底部”的情况，MyISAM引擎还是可以有一定的并发操作。对于Microsoft SQL Server来说，在Microsoft SQL Server 2005版本之前都是页锁的，相对表锁的MyISAM引擎来说，并发性能有所提高。到2005版本，Microsoft SQL Server开始支持乐观并发和悲观并发。在乐观并发下开始支持行级锁，但是其实现方式与InnoDB存储引擎的实现方式完全不同。你会发现在Microsoft SQL Server下，锁是一种稀有的资源，锁越多，开销就越大，因此它会有锁升级。在这种情况下，行锁会升级到表锁，这时并发的性能又回到了以前。

InnoDB存储引擎锁的实现和Oracle非常类似，提供一致性的非锁定读、行级锁支持，行级锁没有相关的开销，可以同时得到并发性和一致性。

6.2 InnoDB存储引擎中的锁

6.2.1 锁的类型

InnoDB存储引擎实现了如下两种标准的行级锁：

- 共享锁（S Lock），允许事务读一行数据。
- 排他锁（X Lock），允许事务删除或者更新一行数据。

当一个事务已经获得了行r的共享锁，那么另外的事务可以立即获得行r的共享锁，因为读取并没有改变行r的数据，我们称这种情况为锁兼容。但如果有事务想获得行r的排他锁，则它必须等待事务释放行r上的共享锁——这种情况我们称为锁不兼容。表6-1列出了共享锁和排他锁的兼容性。

表6-1 排他锁和共享锁的兼容性

	X	S
X	冲突	冲突
S	冲突	兼容

InnoDB存储引擎支持多粒度锁定，这种锁定允许在行级上的锁和表级上的锁同时存在。为了支持在不同粒度上进行加锁操作，InnoDB存储引擎支持一种额外的锁方式，我们称之为意向锁。意向锁是表级别的锁，其设计目的主要是为了在一个事务中揭示下一行将被请求的锁的类型。InnoDB存储引擎支持两种意向锁：

- 意向共享锁（IS Lock），事务想要获得一个表中某几行的共享锁。
- 意向排他锁（IX Lock），事务想要获得一个表中某几行的排他锁。

因为InnoDB存储引擎支持的是行级别的锁，所以意向锁其实不会阻塞除全表扫以外的任何请求。

可以通过SHOW ENGINE INNODB STATUS命令来查看当前请求锁的信息：

```
mysql> show engine innodb status\G;
.......
------------
TRANSACTIONS
------------
Trx id counter 48B89BF
Purge done for trx's n:o < 48B89BA undo n:o < 0
History list length 0
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 0, not started, process no 13757, OS thread id 1255176512
MySQL thread id 42, query id 80424887 localhost root
show engine innodb status
---TRANSACTION 48B89BE, ACTIVE 193 sec, process no 13757, OS thread id
1254910272 starting index read
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 368, 1 row lock(s)
MySQL thread id 41, query id 80424886 localhost root Sending data
select * from t where a < 4 lock in share mode
------- TRX HAS BEEN WAITING 2 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 30 page no 3 n bits 72 index 'PRIMARY' of table 'test'.'t'
trx id 48B89BE lock mode S waiting
------------------
TABLE LOCK table 'test'.'t' trx id 48B89BE lock mode IS
```

```
RECORD LOCKS space id 30 page no 3 n bits 72 index 'PRIMARY' of table 'test'.'t'
trx id 48B89BE lock mode S waiting
---TRANSACTION 48B89BD, ACTIVE 205 sec, process no 13757, OS thread id
1257838912
2 lock struct(s), heap size 368, 1 row lock(s)
MySQL thread id 40, query id 80424881 localhost root
TABLE LOCK table 'test'.'t' trx id 48B89BD lock mode IX
RECORD LOCKS space id 30 page no 3 n bits 72 index 'PRIMARY' of table 'test'.'t'
trx id 48B89BD lock_mode X locks rec but not gap
---------------------------
END OF INNODB MONITOR OUTPUT
============================

1 row in set (0.01 sec)
```

可以看到SQL语句select * from t where a < 4 lock in share mode在等待，RECORD LOCKS space id 30 page no 3 n bits 72 index 'PRIMARY' of table 'test'.'t' trx id 48B89BD lock_mode X locks rec but not gap表示锁住的资源。locks rec but not gap代表锁住是一个索引，不是一个范围。

在InnoDB Plugin之前，我们只能通过SHOW FULL PROCESSLIST、SHOW ENGINE INNODB STATUS等命令来查看当前的数据库请求，然后再判断当前事务中锁的情况。新版本的InnoDB Plugin中，在INFORMATION_SCHEMA架构下添加了INNODB_TRX、INNODB_LOCKS、INNODB_LOCK_WAITS。通过这三张表，可以更简单地监控当前的事务并分析可能存在的锁的问题。通过实例我们来分析这三张表，先看表INNODB_TRX，INNODB_TRX由8个字段组成：

- trx_id：InnoDB存储引擎内部唯一的事务ID。
- trx_state：当前事务的状态。
- trx_started：事务的开始时间。
- trx_requested_lock_id：等待事务的锁ID。如trx_state的状态为LOCK WAIT，那么该值代表当前的事务等待之前事务占用锁资源的ID。若trx_state不是LOCK WAIT，则该值为NULL。
- trx_wait_started：事务等待开始的时间。
- trx_weight：事务的权重，反映了一个事务修改和锁住的行数。在InnoDB存储引擎

中，当发生死锁需要回滚时，InnoDB存储引擎会选择该值最小的进行回滚。

- trx_mysql_thread_id：MySQL中的线程ID，SHOW PROCESSLIST显示的结果。
- trx_query：事务运行的SQL语句。在实际使用中发现，该值有时会显示为NULL（不知道是不是Bug）。

一个具体的例子如下：

```
mysql> select * from information_schema.INNODB_TRX\G;
*************************** 1. row ***************************
                    trx_id: 7311F4
                 trx_state: LOCK WAIT
               trx_started: 2010-01-04 10:49:33
     trx_requested_lock_id: 7311F4:96:3:2
          trx_wait_started: 2010-01-04 10:49:33
                trx_weight: 2
       trx_mysql_thread_id: 471719
                 trx_query: select * from parent lock in share mode
*************************** 2. row ***************************
                    trx_id: 730FEE
                 trx_state: RUNNING
               trx_started: 2010-01-04 10:18:37
     trx_requested_lock_id: NULL
          trx_wait_started: NULL
                trx_weight: 2
       trx_mysql_thread_id: 471718
                 trx_query: NULL
2 rows in set (0.00 sec)
```

可以看到，事务730FEE当前正在运行，而事务7311F4目前处于“LOCK WAIT”状态，运行的SQL语句是select * from parent lock in share mode。这个只是显示了当前运行的InnoDB的事务，并不能判断锁的一些情况，如果需要查看锁，则需要INNODB_LOCKS表，该表由如下字段组成：

- lock_id：锁的ID。
- lock_trx_id：事务ID。
- lock_mode：锁的模式。
- lock_type：锁的类型，表锁还是行锁。
- lock_table：要加锁的表。

- lock_index：锁的索引。
- lock_space：InnoDB存储引擎表空间的ID号。
- lock_page：被锁住的页的数量。若是表锁，则该值为NULL。
- lock_rec：被锁住的行的数量。若是表锁，则该值为NULL。
- lock_data：被锁住的行的主键值。当是表锁时，该值为NULL。

接着上面的例子，我们继续查看INNODB_LOCKS表：

```
mysql> select * from information_schema.INNODB_LOCKS\G;
*************************** 1. row ***************************
    lock_id: 7311F4:96:3:2
lock_trx_id: 7311F4
  lock_mode: S
  lock_type: RECORD
 lock_table: 'mytest'.'parent'
 lock_index: 'PRIMARY'
 lock_space: 96
  lock_page: 3
   lock_rec: 2
  lock_data: 1
*************************** 2. row ***************************
    lock_id: 730FEE:96:3:2
lock_trx_id: 730FEE
  lock_mode: X
  lock_type: RECORD
 lock_table: 'mytest'.'parent'
 lock_index: 'PRIMARY'
 lock_space: 96
  lock_page: 3
   lock_rec: 2
  lock_data: 1
2 rows in set (0.00 sec)
```

这次可能看到当前锁的信息了，ID为730FEE的事务向表parent加了一个X的行锁，ID为7311F4的事务向表parent申请了一个S的行锁。lock_data都是1，申请相同的资源，因此会有等待。这也可以解释INNODB_TRX中为什么一个事务的trx_state是“RUNNING”，另一个是“LOCK WAIT”了。

另外需要注意的是，我发现lock_data这个值并非是“可信”的值。例如当我们运行一个范围查找时，lock_data可能只返回第一行的主键值。另一个不能忽视的是，如果当前资

源被锁住了，与此同时，由于锁住的页因为InnoDB存储引擎缓冲池的容量，而导致替换缓冲池该页，当查看INNODB_LOCKS表时，该值会显示为NULL，即InnoDB存储引擎不会从磁盘进行再一次查找。

查处了每张表上锁的情况后，我们可以来判断由此而引发的等待情况了。当事务较小时，我们人为地、直观地就可以进行判断了。但是当事务量非常大，锁和等待也时常发生时，这个时候不容易判断，但是通过INNODB_LOCK_WAITS，可以很直观地反映出当前的等待。INNODB_LOCK_WAITS由4个字段组成：

- requesting_trx_id：申请锁资源的事务ID。
- requesting_lock_id：申请的锁的ID。
- blocking_trx_id：阻塞的事务ID。
- blocking_trx_id：阻塞的锁的ID。

接着上面的例子，运行如下查询：

```
mysql> select * from information_schema.INNODB_LOCK_WAITS\G;
*************************** 1. row ***************************
requesting_trx_id: 7311F4
requested_lock_id: 7311F4:96:3:2
  blocking_trx_id: 730FEE
 blocking_lock_id: 730FEE:96:3:2
1 row in set (0.00 sec)
```

这次我们可以清楚直观地看到哪个事务阻塞了另一个事务。当然这里只给出了事务和锁的ID，如果需要根据INNODB_TRX、INNODB_LOCKS、INNODB_LOCK_WAITS这三张表直观地看到详细信息，我们可以执行如下联合查询：

```
mysql> SELECT r.trx_id waiting_trx_id, r.trx_mysql_thread_id waiting_thread,
r.trx_query waiting_query, b.trx_id blocking_trx_id, b.trx_mysql_thread_id
blocking_ thread, b.trx_query blocking_query FROM information_schema.innodb_
lock_waits w INNER JOIN information_schema.innodb_trx b ON b.trx_id =
w.blocking_trx_id INNER JOIN information_schema.innodb_trx r ON r.trx_id =
w.requesting_trx_id\G;
*************************** 1. row ***************************
 waiting_trx_id: 73122F
 waiting_thread: 471719
  waiting_query: NULL
blocking_trx_id: 7311FC
blocking_thread: 471718
```

```
 blocking_query: NULL
1 row in set (0.00 sec)
```

6.2.2 一致性的非锁定读操作

一致性的非锁定行读（consistent nonlocking read）是指InnoDB存储引擎通过行多版本控制（multi versioning）的方式来读取当前执行时间数据库中行的数据。如果读取的行正在执行DELETE、UPDATE操作，这时读取操作不会因此而会等待行上锁的释放，相反，InnoDB存储引擎会去读取行的一个快照数据。如图 6-1所示：

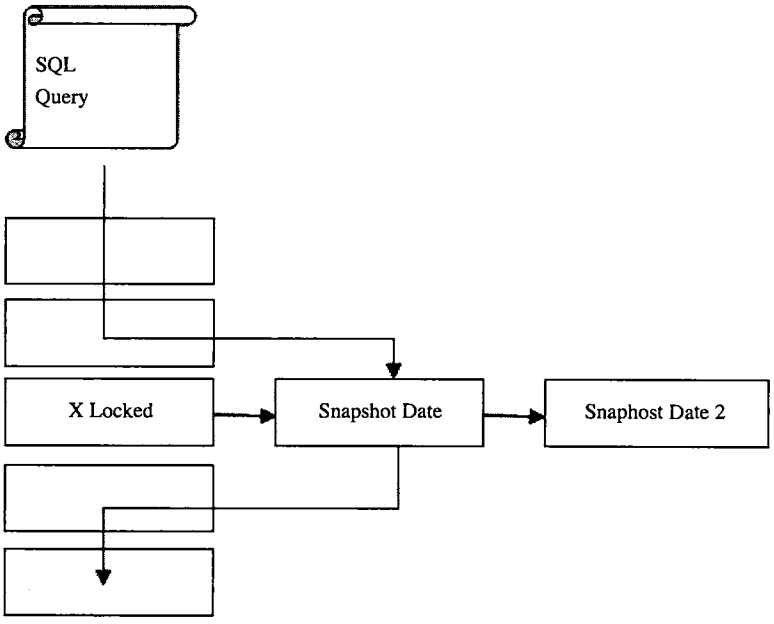


图6-1 InnoDB存储引擎非锁定的一致性读

图 6-1直观地展现了InnoDB存储引擎一致性的非锁定读。之所以称其为非锁定读，因为不需要等待访问的行上X锁的释放。快照数据是指该行之前版本的数据，该实现是通过Undo段来实现。而Undo用来在事务中回滚数据，因此快照数据本身是没有额外的开销。此外，读取快照数据是不需要上锁的，因为没有必要对历史的数据进行修改。

可以看到，非锁定读的机制大大提高了数据读取的并发性，在InnoDB存储引擎默认设置下，这是默认的读取方式，即读取不会占用和等待表上的锁。但是在不同事务隔离级别下，读取的方式不同，并不是每个事务隔离级别下读取的都是一致性读。同样，即使都是

使用一致性读，但是对于快照数据的定义也不相同。

通过图6-1我们知道，快照数据其实就是当前行数据之前的历史版本，可能有多个版本。就图6-1显示的，一个行可能有不止一个快照数据。我们称这种技术为行多版本技术。由此带来的并发控制，称之为多版本并发控制（Multi Version Concurrency Control，MVCC）。

在Read Committed和Repeatable Read（InnoDB存储引擎的默认事务隔离级别）下，InnoDB存储引擎使用非锁定的一致性读。然而，对于快照数据的定义却不相同。在Read Committed事务隔离级别下，对于快照数据，非一致性读总是读取被锁定行的最新一份快照数据。在Repeatable事务隔离级别下和Repeatable Read事务隔离级别下，对于快照数据，非一致性读总是读取事务开始时的行数据版本。我们来看个例子，在一个MySQL的连接会话A中执行如下事务：

```
# Session A
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from parent where id = 1;
+----+
| id |
+----+
|  1 |
+----+
1 row in set (0.00 sec)
```

会话A中事务已Begin（开始），读取了id=1的数据，但是没有结束事务。这时我们再开启另一个会话B，这样可以模拟并发的情况，然后对会话B做如下操作：

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> update parent set id=3 where id=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

会话B中将id=1的行修改为id=3，但是事务同样没有提交，这样id=1的行其实加了一个X锁。这时如果再在会话A中读取id=1的数据，根据InnoDB存储引擎的特性，在Read Committed和Repeatable Read的事务隔离级别下，会使用非锁定的一致性读。我们回到会话A，接着上次未提交的事务，执行select * from parent where id = 1的操作，这时不管使

用Read Committed还是Repeatable的事务隔离级别，显示的数据应该都是：

```
mysql> select * from parent where id = 1;
+----+
| id |
+----+
|  1 |
+----+
1 row in set (0.00 sec)
```

因为当前id=1的数据被修改了1次，因此只有一个版本的数据。接着，我们在会话B中提交上次的事务。如：

```
# Session B
mysql> commit;
Query OK, 0 rows affected (0.01 sec)
```

会话B提交事务后，这时在会话A中再运行select * from parent where id = 1的SQL语句，在Read Committed和Repeatable事务隔离级别下得到的结果就不一样了。对于Read Committed的事务隔离级别，它总是读取行的最新版本，如果行被锁定了，则读取该行版本的最新一个快照（fresh snapshot）。在上述例子中，因为会话B已经提交了事务，所以Read Committed事务隔离级别下会得到如下结果：

```
mysql> select @@tx_isolation;
+------------------------+
| @@tx_isolation         |
+------------------------+
| READ-COMMITTED |
+------------------------+
1 row in set (0.00 sec)

mysql> select * from parent where id = 1;
Empty set (0.00 sec)
```

对于Repeatable的事务隔离级别，总是读取事务开始时的行数据。因此对于Repeatable Read事务隔离级别，其结果如下：

```
mysql> select @@tx_isolation;
+------------------------+
| @@tx_isolation         |
+------------------------+
| REPEATABLE-READ |
```

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

mysql> select * from parent where id = 1;
+----+
| id |
+----+
|  1 |
+----+
1 row in set (0.00 sec)
```

下面将从时间的角度展现上述演示的示例。对于Read Committed的事务隔离级别而言，从数据库理论的角度来看，其实违反了事务ACID中的I的特性，即隔离性。

```
Time          Session A                                  Session B
 |            begin;
 |            select * from parent where id = 1;
 |                                                       begin;
 |                                                       update parent set id=3 where id = 1;
 |            select * from parent where id = 1;
 |                                                       commit;
 |            select * from parent where id = 1;
 V            commit;
```

6.2.3 SELECT ... FOR UPDATE & SELECT ... LOCK IN SHARE MODE

6.2.2小节讲到，在默认情况下，InnoDB存储引擎的SELECT操作使用一致性非锁定读。但是在某些情况下，我们需要对读取操作进行加锁。InnoDB存储引擎对于SELECT语句支持两种加锁操作：

- SELECT ... FOR UPDATE　对读取的行记录加一个X锁。其他事务想在这些行上加任何锁都会被阻塞。
- SELECT ... LOCK IN SHARE MODE　对读取的行记录加一个S锁。其他事务可以向被锁定的记录加S锁，但是对于加X锁，则会被阻塞。

对于一致性非锁定读，即使读取的行已被使用SELECT ... FOR UPDATE，也是可以进行读取的。另外，SELECT ... FOR UPDATE，SELECT ... LOCK IN SHARE MODE必须在一个事务中，当事务提交了，锁也就释放了。因为在使用上述两句SELECT锁定语句时，

务必加上BEGIN、START TRANSACTION或者SET AUTOCOMMIT=0。

6.2.4 自增长和锁

自增长在数据库中是非常常见的一种属性，也是很多DBA或开发人员首选的主键方式。在InnoDB存储引擎的内存结构中，对每个含有自增长值的表都有一个自增长计数器（auto-increment counter）。当对含有自增长计数器的表进行插入操作时，这个计数器会被初始化，执行如下的语句来得到计数器的值：

```
SELECT MAX(auto_inc_col) FROM t FOR UPDATE;
```

插入操作会依据这个自增长的计数器值加1赋予自增长列。这个实现方式称做AUTO-INC Locking。这种锁其实是采用一种特殊的表锁机制，为了提高插入的性能，锁不是在一个事务完成后才释放，而是在完成对自增长值插入的SQL语句后立即释放。

虽然AUTO-INC Locking从一定程度上提高了并发插入的效率，但这里还是存在一些问题。首先，对于有自增长值的列的并发插入性能较差，所以必须等待前一个插入的完成（虽然不用等待事务的完成）。其次，对于INSERT ...SELECT的大数据量的插入，会影响插入的性能，因为另一个事务中的插入会被阻塞。

从MySQL 5.1.22版本开始，InnoDB存储引擎中提供了一种轻量级互斥量的自增长实现机制，这种机制大大提高了自增长值插入的性能。并且从MySQL 5.1.22版本开始，InnoDB存储引擎提供了一个参数innodb_autoinc_lock_mode，默认值为1。在继续讨论新的自增长实现方式之前，我们需要对自增长的插入进行分类：

- INSERT-like：INSERT-like指所有的插入语句，如INSERT、REPLACE、INSERT…SELECT、REPLACE…SELECT、LOAD DATA等。
- Simple inserts：Simple inserts指能在插入前就确定插入行数的语句。这些语句包括INSERT、REPLACE等。需要注意的是：Simple inserts不包含INSERT ...ON DUPLICATE KEY UPDATE这类SQL语句。
- Bulk inserts：Bulk inserts指在插入前不能确定得到插入行数的语句，如INSERT…SELECT，REPLACE…SELECT，LOAD DATA。
- Mixed-mode inserts：Mixed-mode inserts指插入中有一部分的值是自增长的。有一

部分是确定的，如: INSERT INTO t1 (c1,c2) VALUES (1,'a'), (NULL,'b'), (5,'c'), (NULL,'d')，也可以是指INSERT ...ON DUPLICATE KEY UPDATE这类SQL语句。

参数innodb_autoinc_lock_mode有三个可选值：

- innodb_autoinc_lock_mode=0　这是5.1.22版本之前自增长的实现方式，即通过表锁的AUTO-INC Locking方式。因为有了新的自增长实现方式，所以0这个选项不应该是你的首选项。
- innodb_autoinc_lock_mode=1　这是该参数的默认值。对于“Simple inserts”，该值会用互斥量（mutex）去对内存中的计数器进行累加的操作。对于“Bulk inserts”，还是使用传统表锁的AUTO-INC Locking方式。这样做，如果不考虑回滚操作，对于自增值的增长还是连续的。而且在这种方式下，Statement-Based方式的Replication还是能很好地工作。需要注意的是，如果已经使用AUTO-INC Locing的方式产生自增长的值，而这时需要再进行“Simple inserts”的操作时，还是要等待AUTO-INC Locking的释放。
- innodb_autoinc_lock_mode=2　在这个模式下，对于所有“INSERT-like”自增长值的产生都是通过互斥量，而不是AUTO-INC Locking的方式。显然，这是最高性能的方式。然而，这会带来一定的问题。因为并发插入的存在，所以每次插入时，自增长的值可能不是连续的。此外，最重要的是，基于Statement-Base Replication会出现问题。因此，使用这个模式，任何时候都应该使用Row-Base Replication。这样才能保证最大的并发性能和Replication数据的同步。

对于自增长另外需要注意的是，InnoDB存储引擎中的实现和MyISAM不同，MyISAM是表锁的，自增长不用考虑并发插入的问题。因此在Master用InnoDB存储引擎，Slave用MyISAM存储引擎的Replication架构下你必须考虑这种情况。

另外，InnoDB存储引擎下，自增长值的列必须是索引，并且是索引的第一个列，如果是第二个列则会报错；而MyISAM存储引擎则没有这个问题，如：

```
mysql> create table t (a int auto_increment, b int , key (b,a))engine=InnoDB;
ERROR 1075 (42000): Incorrect table definition; there can be only one auto
column and it must be defined as a key
```

```
mysql> create table t (a int auto_increment, b int , key (b,a))engine=MyISAM;
Query OK, 0 rows affected (0.01 sec)
```

6.2.5 外键和锁

前面已经介绍了外键，外键主要用于引用完整性的约束检查。在InnoDB存储引擎中，对于一个外键列，如果没有显式地对这个列加索引，InnoDB存储引擎自动对其加一个索引，因为这样可以避免表锁——这比Oracle做得好，Oracle不会自动添加索引，用户必须自己手工添加，这也是导致很多死锁问题产生的原因。

对于外键值的插入或者更新，首先需要查询父表中的记录，即SELECT父表。但是对于父表的SELECT操作，不是使用一致性非锁定读的方式，因为这样会发生数据不一致的问题，因此这时使用的是SELECT ... LOCK IN SHARE MODE方式，主动对父表加一个S锁。如果这时父表上已经这样加X锁，那么子表上的操作会被阻塞，如下面的例子所示：

Time	Session A	Session B
|	begin;	
|	delete from parent where id=3;	begin;
|		insert into child select 2,3
|		#第二列是外键，执行该句时被阻塞
v		(waiting)

图6-2 外键测试用例

上面的例子中，两个事务都没有COMMIT或者ROLLBACK，这时Session B的操作会被阻塞。因为id=3的父表上在Session A中已经加了一个X锁，而这时我们需要对父表中id=3的行加一个S锁，这时insert的操作会被阻塞。设想如果访问父表时，使用的是一致性的非锁定读，这时Session B会读到父表有id=3的记录，可以进行插入操作。但是如果Session A对事务提交了，则父表中就没有id=3的记录。数据在父子表就会存在不一致的情况。如果我们查询INNODB_LOCKS表，会得到如下结果：

```
mysql> select * from information_schema.INNODB_LOCKS\G;
*************************** 1. row ***************************
    lock_id: 7573B8:96:3:4
lock_trx_id: 7573B8
  lock_mode: S
  lock_type: RECORD
 lock_table: 'mytest'.'parent'
```

```
 lock_index: 'PRIMARY'
 lock_space: 96
  lock_page: 3
   lock_rec: 4
  lock_data: 3
*************************** 2. row ***************************
    lock_id: 7573B3:96:3:4
lock_trx_id: 7573B3
  lock_mode: X
  lock_type: RECORD
 lock_table: 'mytest'.'parent'
 lock_index: 'PRIMARY'
 lock_space: 96
  lock_page: 3
   lock_rec: 4
  lock_data: 3
2 rows in set (0.00 sec)
```

6.3 锁的算法

InnoDB存储引擎有3中行锁的算法设计，分别是：

- Record Lock：单个行记录上的锁。
- Gap Lock：间隙锁，锁定一个范围，但不包含记录本身。
- Next-Key Lock：Gap Lock + Record Lock，锁定一个范围，并且锁定记录本身。

Record Lock总是会去锁住索引记录。如果InnoDB存储引擎表建立的时候没有设置任何一个索引，这时InnoDB存储引擎会使用隐式的主键来进行锁定。

Next-Key Lock是结合了Gap Lock和Record Lock的一种锁定算法，在Next-Key Lock算法下，InnoDB对于行的查询都是采用这种锁定算法。对于不同SQL查询语句，可能设置共享的（Share）Next-Key Lock和排他的（exlusive）Next-Key Lock。

可以通过一个例子来演示Next-Key Lock的锁定算法，建立一张表t，插入值为1、2、3、4、7、8的6条记录。

```
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 2;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> insert into t select 3;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> insert into t select 4;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> insert into t select 7;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> insert into t select 8;
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from t;
*************************** 1. row ***************************
a: 1
*************************** 2. row ***************************
a: 2
*************************** 3. row ***************************
a: 3
*************************** 4. row ***************************
a: 4
*************************** 5. row ***************************
a: 7
*************************** 6. row ***************************
a: 8
6 rows in set (0.00 sec)
```

接着开启两个会话，会话A在一个事务中执行select * from t where a < 6 lock in share mode，会话B中，插入小于6或者等于6的记录，如下所示：

```
Time          Session A                          Session B
 |            begin;
 |            select * from t where a < 6
 |        lock in share mode;                    begin;
 |                                               insert into t select 5(或者6);
 |                                               (blocking)
 V
```

在这种情况下，不论插入的记录是5还是6，都会被锁定。因为在Next-Key Lock算法下，锁定的是（−∞，6）这个数值区间的所有数值。但是插入9这个数值是可以的，因为该记录不在锁定的范围内，而对于单个值的索引查询，不需要用到Gap Lock，只要加一个Record Lock即可，因此InnoDB存储引擎会自己选一个最小的算法模型。同样，对于上面的表t，进行如下操作：

```
Time          Session A                          Session B
 |            begin;
 |            select * from t where a = 7
 |        lock in share mode;                    begin;
 |                                               insert into t select 5(或者6);
 |                                               (success)
 V
```

这时插入记录5或6都是可行的了。需要注意的是，上面演示的两个例子都是在InnoDB的默认配置下，即事务的隔离级别为REPEATABLE READ的模式下。因为在REPEATABLE READ模式下，Next-Key Lock算法是默认的行记录锁定算法。

6.4 锁问题

通过锁可以实现事务的隔离性的要求，使得事务可以并发地工作。锁提高了并发，但是却会带来问题。不过，好在因为事务隔离性的要求，锁只会带来3种问题。如果可以防止这3种情况的发生，那将不会产生并发异常。

6.4.1 丢失更新

丢失更新（lost update）是一个经典的数据库问题。实际上，所有多用户计算机系统环境下有可能产生这个问题。简单说来，出现下面的情况时，就会发生丢失更新：

（1）事务T1查询一行数据，放入本地内存，并显示给一个终端用户User1。

（2）事务T2也查询该行数据，并将取得的数据显示给终端用户User2。

（3）User1修改这行记录，更新数据库并提交。

（4）User2修改这行记录，更新数据库并提交。

显然，这个过程中用户User1的修改更新操作“丢失”了。这可能会发生一个恐怖的结果。设想银行丢失了更新操作：一个用户账户中有10 000元人民币，他用两个网上银行的客户端转账，第一次转9 000人民币，因为网络和数据的关系，这时需要等待。但是如果这时用户可以操作另一个网上银行客户端，转账1元。如果最终两笔操作都成功了，用户的账号余款是9 999人民币，第一转的9 000人民币并没有得到更新。也许有人会说，不对，我的网银是绑定USB Key的，不会发生这种情况——通过USB Key登录也许可以解决这个问题，但是更重要的是，要在数据库层解决这个问题，以避免任何可能发生丢失更新的情况。

要避免丢失更新发生，其实需要让这种情况下的事务变成串行操作，而不是并发的操作。即在上述四种的第（1）种情况下，对用户读取的记录加上一个排他锁，同样，发生第（2）种情况下的操作时，用户也需要加一个排他锁。这种情况下，第（2）步就必须等待第（1）、（3）步完成，最后完成第（4）步，如以下所示：

```
Time        Session A                              Session B
 |          begin;
 |          select cash into @cash from
 |      account where user = pUser for        begin;
 |      update;                                   select cash into @cash from
 |                                          account where user = pUser for
 |                                          update;
 |          update account set cash=@cash-
 |      9000 where user=pUser
 |          commit;                               update account set cash=@cash-1
 |                                          where user=pUser
 V                                              commit;
```

我发现，程序员可能在了解如何使用SELECT、INSERT、UPDATE、DELETE语句后就开始编写应用程序。因此，丢失更新是程序员最容易犯的错误，也是最不易发现的一个错误，特别是由于这种现象只是随机的、零星的出现，但其可能造成的后果却十分严重。

6.4.2 脏读

理解脏读之前，需要理解脏数据的概念。脏数据和脏页有所不同。脏页指的是在缓冲池中已经被修改的页，但是还没有刷新到磁盘，即数据库实例内存中的页和磁盘的页中的数据是不一致的，当然在刷新到磁盘之前，日志都已经被写入了重做日志文件中。而所谓脏数据，是指在缓冲池中被修改的数据，并且还没有被提交（commit）。

对于脏页的读取，是非常正常的。脏页是因为数据库实例内存和磁盘的异步同步造成的，这并不影响数据的一致性。并且因为是异步的，因此可以带来性能的提高。而脏数据却不同，脏数据是指未提交的数据。如果读到了脏数据，即一个事务可以读到另外一个事务中未提交的数据，则显然违反了数据库的隔离性。

脏读指的就是在不同的事务下，可以读到另外事务未提交的数据，简单来说，就是可以读到脏数据。比如下面的例子所示：

```
Time        Session A                          Session B
|           set @@tx_isolation='read-
|       uncommitted';                          set @@tx_isolation='read-uncommitted';
|           begin;                             begin;
|                                              mysql> select * from t;
|                                              +---+
|                                              | a |
|                                              +---+
|                                              | 1 |
|                                              +---+
|                                              1 row in set (0.00 sec)
|
|           insert into t select 2;            mysql> select * from t;
|                                              +---+
|                                              | a |
|                                              +---+
|                                              | 1 |
|                                              | 2 |
|                                              +---+
v                                              2 rows in set (0.00 sec)
```

表t为我们之前6.4.1中建的表，不同的是在上述例子中，事务的隔离级别进行了更换，由默认的REPEATABLE READ换成了READ UNCOMMITTED，因此在会话A中事务并没有提交的前提下，会话B中两次SELECT操作取得了不同的结果，并且这两个记录是在会话A中并未提交的数据，即产生了脏读，违反了事务的隔离性。

脏读现象在生产环境中并不常发生。从上面的例子中就可以发现，脏读发生的条件是需要事务的隔离级别为READ UNCOMMITTED，而目前绝大部分的数据库都至少设置成READ COMMITTED。InnoDB存储引擎默认的事务隔离级别为READ REPEATABLE，Microsoft SQL Server数据库为READ COMMITTED，Oracle数据库同样也是READ COMMITTED。

6.4.3 不可重复读

不可重复读是指在一个事务内多次读同一数据。在这个事务还没有结束时，另外一个事务也访问该同一数据。那么，在第一个事务的两次读数据之间，由于第二个事务的修改，第一个事务两次读到的数据可能是不一样的。这样就发生了在一个事务内两次读到的数据是不一样的，因此称为不可重复读。

不可重复读和脏读的区别是：脏读是读到未提交的数据；而不可重复读读到的确实是已经提交的数据，但是其违反了数据库事务一致性的要求。可以通过下面一个例子来观察不可重复读的情况：

```
Time          Session A                          Session B
 |            set @@tx_isolation='read-committed';
 |                                               set @@tx_isolation='read-committed';
 |            begin;                             begin;
 |            select * from t ;                   insert into t select 2;
 |            +---+
 |            | a |
 |            +---+
 |            | 1 |
 |            +---+
 |            1 row in set (0.00 sec)
 |
 |            +---+
 |
```

```
|            | a |
|            +---+
|            | 1 |
|            +---+
|            1 row in set (0.00 sec)              commit;
|
|            mysql> select * from t;
|            +---+
|            | a |
|            +---+
|            | 1 |
|            | 2 |
|            +---+
V            2 rows in set (0.00 sec)
```

会话A中开始一个事务，第一次读取到的记录是1；另一个会话B中开始了另一个事务，插入一条2的记录。在没有提交之前，会话A中的事务再次读取时，读到的记录还是1，没有发生脏读的现象。但会话2中的事务提交后，在对会话A中的事务进行读取时，这时读到的是1和2两条记录。这个例子的前提是，在事务开始前，会话A和会话B的事务隔离级别都调整为了READ COMMITTED。

一般来说，不可重复读的问题是可以接受的，因为其读到的是已经提交的数据，本身并不会带来很大的问题。因此，很多数据库厂商（如Oracle、Microsoft SQL Server）将其数据库事务的默认隔离级别设置为READ COMMITTED，在这种隔离级别下允许不可重复读的现象。

InnoDB存储引擎中，通过使用Next-Key Lock算法来避免不可重复读的问题。在MySQL官方文档中，将不可重复读定义为Phantom Problem，即幻象问题。在Next-Key Lock算法下，对于索引的扫描，不仅仅是锁住扫描到的索引，而且还锁住这些索引覆盖的范围（gap）。因此对于这个范围内的插入都是不允许的。这样就避免了另外的事务在这个范围内插入数据导致的不可重复读的问题。因此，InnoDB存储引擎的默认事务隔离级别是READ REPEATABLE，采用Next-Key Lock算法，就避免了不可重复读的现象。

6.5 阻塞

因为不同锁之间的兼容性关系，所以在有些时刻，一个事务中的锁需要等待另一个事

务中的锁释放它所占用的资源。在InnoDB存储引擎的源代码中，用Mutex数据结构来实现锁。在访问资源前需要用mutex_enter函数进行申请，在资源访问或修改完毕后立即执行mutex_exit函数。当一个资源已被一个事务占有时，另一个事务执行mutex_enter函数会发生等待，这就是阻塞。阻塞并不是一件坏事，阻塞是为了保证事务可以并发并且正常运行。

在InnoDB存储引擎中，参数innodb_lock_wait_timeout用来控制等待的时间（默认是50秒），innodb_rollback_on_timeout用来设定是否在等待超时时对进行中的事务进行回滚操作（默认是OFF，代表不回滚）。参数innodb_lock_wait_timeout是动态的，可以在MySQL数据库运行时进行调整，而innodb_rollback_on_timeout是静态的，不可在启动时进行修改，如：

```
mysql> set @@innodb_lock_wait_timeout=60;
Query OK, 0 rows affected (0.00 sec)

mysql> set @@innodb_rollback_on_timeout=on;
ERROR 1238 (HY000): Variable 'innodb_rollback_on_timeout' is a read only variable
当发生超时时，数据库会抛出一个1205的错误，如：
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from t where a = 1 for update;
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

需要牢记的是，默认情况下InnoDB存储引擎不会回滚超时引发的错误异常。其实InnoDB存储引擎在大部分情况下都不会对异常进行回滚。如在一个会话中执行了如下语句：

```
# 会话A
mysql> select * from t;
+---+
| a |
+---+
| 1 |
| 2 |
| 4 |
+---+
3 rows in set (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> select * from t where a < 4 for update;
+---+
| a |
+---+
| 1 |
| 2 |
+---+
2 rows in set (0.00 sec)
```

会话A中开启了一个事务，Next-Key Lock算法下锁定了小于4的所有记录（其实也锁定了4这个记录）。在另一个会话中执行如下语句：

```
# 会话B
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t select 5;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> insert into t select 3;
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
mysql> select * from t;
+---+
| a |
+---+
| 1 |
| 2 |
| 4 |
| 5 |
| 8 |
+---+
5 rows in set (0.00 sec)
```

可以看到，在会话B中插入记录5是可以的，但是插入记录3的话，因为Next-Key Lock算法的关系，需要等待会话A中事务释放这个资源，因此等待后产生了超时。但是在超时后，我们再进行SELECT会发现，5这个记录并没有并回滚。其实这时事务发生了错误，但是既没有commit，也没有rollback，这是十分危险的，用户必须判断是需要commit还是需要rollback，然后再进行下一步操作。

6.6 死锁

如果程序是串行的，那么不可能发生死锁。死锁只发生于并发的情况，数据库就是一个并发进行着的程序，因此可能会发生死锁。在2.2.1小节中我们已经知道，InnoDB存储引擎有一个后台的锁监控线程，该线程负责查看可能的死锁问题，并自动告知用户。下面的操作演示了死锁的一种经典的情况，即A等待B，B在等待A：

```
Time   Session A                              Session B
 |     begin;
 |     mysql> select * from t where a = 1     begin;
 |     for update;                            mysql> select * from t where a = 2
 |     +---+                                  for update;
 |     | a |                                  +---+
 |     +---+                                  | a |
 |     | 1 |                                  +---+
 |     +---+                                  | 2 |
 |     1 row in set (0.00 sec)                +---+
 |                                            1 row in set (0.00 sec)
 |
 |     mysql> select * from t where a = 2
 |     for update;
 |     (blocking)
 |
 |
 |
 |
 |
 |                                            mysql> select * from t where a = 1
 |                                            for update;
 |                                            ERROR 1213 (40001): Deadlock found
 |                                            when trying to get lock; try
 |     (get the result)                       restarting transaction
 |     +---+
 |     | a |
 |     +---+
 |     | 2 |
 |     +---+
 |     1 row in set (0.00 sec)
 V
```

在上述操作中，会话中的事务抛出了1213这个出错提示，即发生了死锁。死锁的原因

是会话A和B的资源互相在等待。大多数的死锁InnoDB存储自己可以侦测到，不需要人为进行干预。但是在上面的例子中，会话B中的事务抛出死锁异常后，会话A中马上得到了记录为2的这个资源，这其实是因为会话B中的事务发生了回滚，否则会话A中的事务是不可能得到该资源的。你还记得6.5节中所说的内容吗？InnoDB存储引擎并不会回滚大部分的错误异常，但是死锁除外。发现死锁后，InnoDB存储引擎会马上回滚一个事务，这点是需要注意的。如果在应用程序中捕获了1213这个错误，其实并不需要对其进行回滚。

Oracle数据库中产生死锁的常见原因是没有对外键添加索引，而InnoDB存储引擎会自动对其进行添加，因此很好地避免了这种情况的发生。人为删除外键上的索引数据库会抛出一个异常：

```
mysql> create table p ( a int,primary key(a));
Query OK, 0 rows affected (0.00 sec)

mysql> create table c ( b int,foreign key(b) references p(a))engine=innodb;
Query OK, 0 rows affected (0.00 sec)

mysql> show index from c\G;
*************************** 1. row ***************************
       Table: c
  Non_unique: 1
    Key_name: b
Seq_in_index: 1
 Column_name: b
   Collation: A
 Cardinality: 0
    Sub_part: NULL
      Packed: NULL
        Null: YES
  Index_type: BTREE
     Comment:
1 row in set (0.00 sec)

mysql> drop index b on c;
ERROR 1553 (HY000): Cannot drop index 'b': needed in a foreign key constraint
```

可以看到，虽然在建立子表时指定了外键，但是InnoDB存储引擎还是自动在外键列上建立了一个索引b，而人为删除这个列却是不允许的。

6.7 锁升级

锁升级（Lock Escalation）是指将当前锁的粒度降低。举例来说，数据库可以把一个表的1 000个行锁升级为一个页锁，或者将页锁升级为表锁。如果数据库的设计中认为锁是一种稀有资源，而且想避免锁的开销，那数据库中会频繁出现锁升级现象。

Microsoft SQL Server数据库的设计认为锁是一种稀有的资源，在适合的时候会自动地将行、键或者分页级锁升级为更粗粒度的表级锁。这种升级保护了系统资源，防止系统使用太多的内存来维护锁，从一定程度上提高了效率。

即使在Microsoft SQL Server 2005的版本之后，SQL Server数据库支持了行锁，但是其设计和InnoDB存储引擎完全不同，在以下情况下依然可能发生锁升级：

- 由一句单独的SQL语句在一个对象上持有的锁数量超过了阈值，默认的这个阈值为5000。值得注意的是，如果是不同对象的话，则不会发生锁升级。
- 锁资源占用的内存超过了激活内存的40%时，就会发生锁升级。

在Microsoft SQL Server数据库中，因为锁是一种稀有的资源，因此锁升级会带来一定的效率提高。但是锁升级带来的一个问题却是，因为锁粒度的降低而导致并发性能的降低。

InnoDB存储引擎不存在锁升级的问题。在InnoDB存储引擎中，1个锁的开销与1 000 000个锁是一样的，都没有开销。这一点和Oracle数据库比较类似。

6.8 小结

这一章介绍的内容非常多，可能会让你觉得很难，不时地抓耳挠腮。尽管锁本身相当直接，但是它的一些副作用却不是这样。关键是你要理解锁带来的问题，如丢失更新、脏读、不可重复读等。如果不知道这一点，那么开发的应用程序性能就会很差。如果不学会怎样通过一些命令和数据字典来查看谁锁住了哪些资源，那可能永远不知道到底发生了什么事情。你可能只是认为数据库有时会阻塞而已。

本章在介绍锁的同时，还比较了MySQL InnoDB存储引擎、MyISAM存储引擎、Microsoft SQL Server数据库、Oracle数据库锁的特性。通过上面的比较了解到，虽然每个数据库在SQL语句层面上的差别可能不是很大，但在内部底层的实现却各有不同。理解InnoDB存储引擎锁的特性，对于开发一个高性能、高并发的数据库应用显得十分重要和有帮助。

第7章 事 务

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，如果你正在写文件，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，如果需要保证两个文件同步，这些文件系统可能就显得无能为力了。如当你需要更新两个文件时，更新完一个文件后，在更新完第二个文件之前系统重启了，你就会有两个不同步的文件。

这正是数据库系统引入事务的主要目的：事务会把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，可以确保其要么所有修改都已经保存了，要么所有修改都不保存。

InnoDB存储引擎中的事务完全符合ACID的特性。ACID是以下4个词的缩写：

- 原子性（atomicity）
- 一致性（consistency）
- 隔离性（isolation）
- 持久性（durability）

第6章已介绍了锁，讨论InnoDB是如何实现事务的隔离性的。本章主要关注事务的原子性这一概念，并说明怎样正确使用事务以及编写正确的事务应用程序，避免在事务方面养成一些不好的习惯。

7.1 事务概述

事务是数据库区别于文件系统的重要特性之一。事务用来保证数据库的完整性——要么都做修改，要么都不做。同时，事务有严格的定义，它必须同时满足四个特性。

- 原子性（atomicity）

原子性是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

执行都成功，才算整个事务成功。如果事务中任何一个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。

❑ 一致性（consistency）

一致性指事务将数据库从一种状态转变为下一种一致的状态。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。

❑ 隔离性（isolation）

一个事务的影响在该事务提交前对其他事务都不可见——这通过锁来实现。

❑ 持久性（durability）

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。

7.2 事务的实现

隔离性由第6章讲述的锁得以实现。原子性、一致性、持久性通过数据库的redo和undo来完成。

7.2.1 redo

在InnoDB存储引擎中，事务日志通过重做（redo）日志文件和InnoDB存储引擎的日志缓冲（InnoDB Log Buffer）来实现。当开始一个事务时，会记录该事务的一个LSN（Log Sequence Number，日志序列号）；当事务执行时，会往InnoDB存储引擎的日志缓冲里插入事务日志；当事务提交时，必须将InnoDB存储引擎的日志缓冲写入磁盘（默认的实现，即innodb_flush_log_at_trx_commit=1）。也就是在写数据前，需要先写日志。这种方式称为预写日志方式（Write-Ahead Logging，WAL）。

InnoDB存储引擎通过预写日志的方式来保证事务的完整性。这意味着磁盘上存储的数据页和内存缓冲池中的页是不同步的，对于内存缓冲池中页的修改，先是写入重做日志文件，然后再写入磁盘，因此是一种异步的方式。可以通过命令SHOW ENGINE INNODB STATUS来观察当前磁盘和日志的“差距”：

```
create table z (a int,primary key(a))engine=innodb;
```

```
create procedure load_test (count int)
begin
declare i int unsigned default 0;
start transaction;
while i < count do
insert into z select i;
set i=i+1;
end while;
commit;
end;
```

首先建立一张表z，然后建立一个往表z中导入数据的存储过程load_test。通过命令SHOW ENGINE INNODB STATUS观察当前的重做日志情况：

```
mysql> show engine innodb status\G;
......
---
LOG
---
Log sequence number 11 3047174608
Log flushed up to   11 3047174608
Last checkpoint at  11 3047174608
0 pending log writes, 0 pending chkp writes
142 log i/o's done, 0.00 log i/o's/second
......
1 row in set (0.00 sec)
```

Log sequence number表示当前的LSN，Log flushed up to表示刷新到重做日志文件的LSN，Last checkpoint at表示刷新到磁盘的LSN。因为当前没有任何操作，所以这三者的值是一样的。接着开始导入10 000条记录：

```
mysql>call load_test(10000);

mysql> show engine innodb status\G;
......
---
LOG
---
Log sequence number 11 3047672789
Log flushed up to   11 3047672789
Last checkpoint at  11 3047174608
0 pending log writes, 0 pending chkp writes
143 log i/o's done, 0.08 log i/o's/second
```

```
.......
1 row in set (0.00 sec)
```

这次SHOW ENGINE INNODB STATUS的结果就不同了，Log sequence number的LSN为113047672789，Log flushed up to的LSN为113047672789，Last checkpoint at的LSN为113047174608，可以把Log flushed up to和Last checkpoint at的差值498 181（~486.5K）理解为重做日志产生的增量（以字节为单位）。

虽然在上面的例子中，Log sequence number和Log flushed up to的值是相等的，但是在实际的生产环境中，该值有可能是不同的。因为在一个事务中从日志缓冲刷新到重做日志文件，并不只是在事务提交时发生，每秒都会有从日志缓冲刷新到重做日志文件的动作（这部分内容我们在3.6.2小节已经讲解过了）。下面是一个生产环境下重做日志的信息：

```
mysql> show engine innodb status\G;

---
LOG
---
Log sequence number 203318213447
Log flushed up to   203318213326
Last checkpoint at  203252831194
1 pending log writes, 0 pending chkp writes
103447 log i/o's done, 7.00 log i/o's/second
.......
1 row in set (0.00 sec)
```

可以看到，在生产环境下Log sequence number、Log flushed up to、Last checkpoint at三个值可能是不同的。

7.2.2 undo

重做日志记录了事务的行为，可以很好地通过其进行“重做”。但是事务有时还需要撤销，这时就需要undo。undo与redo正好相反，对于数据库进行修改时，数据库不但会产生redo，而且还会产生一定量的undo，即使你执行的事务或语句由于某种原因失败了，或者如果你用一条ROLLBACK语句请求回滚，就可以利用这些undo信息将数据回滚到修改之前的样子。与redo不同的是，redo存放在重做日志文件中，undo存放在数据库内部的一个特殊段（segment）中，这称为undo段（undo segment），undo段位于共享表空间内。可

以通过py_innodb_page_info.py工具，来查看当前共享表空间中undo的数量：

```
[root@xen-server ~]# python py_innodb_page_info.py /usr/local/mysql/data/ibdata1
Total number of page: 46208:
Insert Buffer Free List: 13093
Insert Buffer Bitmap: 3
System Page: 5
Transaction system Page: 1
Freshly Allocated Page: 4579
undo Log Page: 2222
File Segment inode: 6
B-tree Node: 26296
File Space Header: 1
扩展描述页：2
```

可以看到，当前的共享表空间ibdata1内有2222个undo页。

我们通常对于undo有这样的误解：undo用于将数据库物理地恢复到执行语句或事务之前样子——但事实并非如此。数据库只是逻辑地恢复到原来的样子，所有修改都被逻辑地取消，但是数据结构本身在回滚之后可能大不相同，因为在多用户并发系统中，可能会有数十、数百甚至数千个并发事务。数据库的主要任务就是协调对于数据记录的并发访问。如一个事务在修改当前一个页中某几条记录，但同时还有别的事务在对同一个页中另几条记录进行修改。因此，不能将一个页回滚到事务开始的样子，因为这样会影响其他事务正在进行的工作。

例如：我们的事务执行了一个INSERT 10万条记录的SQL语句，这条语句可能会导致分配一个新的段，即表空间会增大。如果我们执行ROLLBACK时，会将插入的事务进行回滚，但是表空间的大小并不会因此而收缩。因此，当InnoDB存储引擎回滚时，它实际上做的是与先前相反的工作。对于每个INSERT，InnoDB存储引擎会完成一个DELETE；对于每个DELETE，InnoDB存储引擎会执行一个INSERT；对于每个UPDATE，InnoDB存储引擎则会执行一个相反的UPDATE，将修改前的行放回去。

Oracle和Microsoft SQL Server数据库都有内部的数据字典来观察当前undo的信息；InnoDB存储引擎在这方面做得还是不够的，所以DBA只能通过原理和经验来进行判断。我写过一个补丁（patch）来扩展SHOW ENGINE INNODB STATUS命令的显示结果，可以用来查看当前内存缓冲池中undo页的数量，如下代码所示。

```
mysql> show engine innodb status\G;
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
100721 11:46:58 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 14 seconds
......
----------------------
UNDO PAGE INFO
----------------------
Undo Page count: 1.
......
----------------------------
END OF INNODB MONITOR OUTPUT
============================

1 row in set (0.01 sec)
```

可以看到，当前内存缓冲中有1个undo页。接着我们开启一个事务，执行插入10万条记录的操作，需要注意的是，这并不进行提交操作：

```
mysql> create table t like order_line;
Query OK, 0 rows affected (0.23 sec)

mysql> insert into t select * from order_line limit 100000;
Query OK, 100000 rows affected (45.01 sec)
Records: 100000  Duplicates: 0  Warnings: 0
```

之后在另一个会话中执行命令SHOW ENGINE INNODB STATUS，可以看到之前的会话产生的undo量：

```
mysql> show engine innodb status\G;
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
100721 11:46:58 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 14 seconds
......
```

```
----------------------
UNDO PAGE INFO
----------------------
Undo Page count: 129.
......
----------------------------
END OF INNODB MONITOR OUTPUT
============================

1 row in set (12.38 sec)
```

可以看到，此时undo页的数量变成了129，也就是说，刚才的一个事务大致产生了129个undo页。另外，即使对INSERT的事务进行了提交，我们在一段时间内还是可以看到内存中有129个undo页。这是因为，对于undo页的回收是在master thread中进行的，master thread也不是每次回收所有的undo页。关于master thread的工作原理，我们在第2.3.1小节曾介绍过。

7.3 事务控制语句

在MySQL命令行的默认设置下，事务都是自动提交的，即执行SQL语句后就会马上执行COMMIT操作。因此开始一个事务，必须使用BEGIN、START TRANSACTION，或者执行SET AUTOCOMMIT=0，以禁用当前会话的自动提交。这和Microsoft SQL Server数据库的方式一致，需要显式地开始一个事务。而Oracle数据库不需要专门的语句来开始事务，事务会在修改数据的第一条语句处隐式地开始。在具体介绍其含义之前，先来看看我们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会提交你的事务，并使得已对数据库做的所有修改成为永久性的。
- ROLLBACK：要想使用这个语句的最简形式，只需发出ROLLBACK。同样，你也可以写为ROLLBACK WORK，但是二者几乎是等价的。回滚会结束你的事务，并撤销正在进行的所有未提交的修改。

- SAVEPOINT identifier：SAVEPOINT允许你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，而且你捕获到这个异常，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、SERIALIZABLE。

START TRANSACTION、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分析会自动将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语句基本是上一致的，都是用来提交事务。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。可以通过参数completion_type来进行控制，默认情况下该参数为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这个记录，而没有2这个记录
mysql> select * from t\G;
*************************** 1. row ***************************
a: 1
1 row in set (0.00 sec)
```

这个实验中我们将completion_type设置为1，第一次通过COMMIT WORK来插入1这个记录。之后插入记录2时我们并没有用BEGIN（或者START TRANSACTION）来开启一个事务，之后再插入一条重复的记录2，这时会抛出异常。我们执行ROLLBACK操作，最后发现只有1这一个记录，2并没有被插入。因为completion_type为1时，COMMIT WORK会自动开启一个事务，因此两个INSERT语句是在同一个事务内的，因此回滚后就没有进行插入。

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，COMMIT WORK后，我们再执行select @@version，会出现ERROR 2006 （HY000）：MySQL server has gone away的错误，这其实就是因为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，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：

```
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：或者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的语句。一条语句要么完全成功，要么完全回滚（注意，我说的是语句回滚）。如果一条语句失败，并不

会导致先前已经执行的语句自动回滚。它们的工作会保留，必须由你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，这时事务仍需要我们显式地运行COMMIT或者ROLLBACK。

另一个容易犯的错误是ROLLBACK TO SAVEPOINT，虽然有ROLLBACK，但是它并不是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或者ROLLBACK命令。

```
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)
```

在上面的例子中可以看到，虽然我们在发生重复错误后，通过ROLLBACK TO SAVEPOINT t2命令回滚到了保存点t2，但是事务此时并没有结束，我们再接着运行ROLLBACK后，事务才完整回滚。需要再次提醒的是，ROLLBACK TO SAVEPOINT命令并不真正地结束事务。

7.4 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、TRUNCATE TABLE。
- 用来隐式地修改mysql架构的操作：CREATE USER、DROP USER、GRANT、RENAME USER、REVOKE、SET PASSWORD。
- 管理语句：ANALYZE TABLE、CACHE INDEX、CHECK TABLE、LOAD INDEX INTO CACHE、OPTIMIZE TABLE、REPAIR TABLE。

注意：我发现Microsoft SQL Server的数据库管理员或者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数据库都不同。

另外需要注意的是，TRUNCATE TABLE语句是DDL，因此虽然和DELETE整张表的结果是一样的，但它是不能被回滚的（这又是和Microsoft SQL Server数据不同的地方）：

```
mysql> select * from 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)
```

7.5 对于事务操作的统计

因为InnoDB存储引擎是支持事务的，因此对于InnoDB存储引擎的应用，在考虑每秒请求数（Question Per Second，QPS）的同时，也许更应该关注每秒事务处理的能力（Transaction Per Second，TPS）。

计算TPS的方法是（com_commit + com_rollback）/ time。但是用这种方法计算的前提是：所有的事务必须都是显式提交的，如果存在隐式的提交和回滚（默认 autocommit=1），不会计算到com_commit和com_rollback变量中。如：

```
mysql> show global status like 'com_commit'\G;
*************************** 1. row ***************************
Variable_name: Com_commit
        Value: 5
1 row in set (0.00 sec)

mysql> insert into t select 3;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> select * from t\G;
*************************** 1. row ***************************
a: 1
*************************** 2. row ***************************
a: 2
*************************** 3. row ***************************
a: 3
3 rows in set (0.00 sec)


mysql> show global status like 'com_commit'\G;
*************************** 1. row ***************************
Variable_name: Com_commit
        Value: 5
1 row in set (0.00 sec)
```

MySQL另外还有2个参数handler_commit和handler_rollback。但是我注意到，这两个参数在MySQL 5.1中可以很好地用来统计InnoDB存储引擎显式和隐式的事务提交操作，而在InnoDB Plugin中该参数的表现有些“怪异”，并不能很好地统计事务的次数。所以，如果你的程序都是显式控制事务的提交和回滚，那么可以通过com_commit和com_rollback进

行统计。如果不是，那么情况就显得复杂了。

7.6 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。ISO和ANIS SQL标准制定了四种事务隔离级别的标准，但是很少有数据库开发厂商遵循这些标准，比如Oracle数据库就不支持read uncommitted和repeatable read的事务隔离级别。

SQL标准定义的四个隔离级别为：

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

READ UNCOMMITTED称为浏览访问（browse access），仅仅只对事务而言的。READ COMMITTED称为游标稳定（cursor stability）。REPEATABLE READ是2.9999°的隔离，没有幻读的保护。SERIALIZABLE称为隔离，或3°。SQL和SQL 2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锁的算法，因此避免幻读的产生。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REPEATABLE READ的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，或者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。

在InnoDB存储引擎中，可以使用以下命令来设置当前会话或者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SERIALIABLE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。因此在这个事务隔离级别下，读占用锁了，一致性的非锁定读不再予以支持。因为InnoDB存储引擎在REPEATABLE READ隔离级别下就可以达到3°的隔离，所以一般不在本地事务中使用SERIALIABLE的隔离级别，SERIALIABLE的事务隔离级别主要用于InnoDB存储引擎的分布式事务。

在READ COMMITTED的事务隔离级别下，除了唯一性的约束检查以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的格式下。如果二进制日志工作在默认的STATEMENT下，则会指出如下的错误：

```
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版本之前不支持ROW格式的二进制日志时，也许有人知道，可以通过将参数innodb_locks_unsafe_for_binlog设置为1，来使得可以在二进制日志为STATEMENT下使用READ COMMITTED的事务隔离级别：

```
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使得在MySQL 5.0的版本之前使用READ COMMITED事务隔离级别。但是就像参数innodb_locks_unsafe_for_binlog的名称一样，它是unsafe的。在某些情况下，可能会导致master和slave之间数据的不一致。接着我来演示一种可能导致不同步的情况，首先来看下表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，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)
```

接着会话A提交，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。导致这个问题发生的原因有两点：首先，在READ COMMITTED事务隔离级别下，事务是没有Gap Lock锁的，因此我们可以在小于等于5的范围内再插入一条记录；其次，statement记录的是master上产生的SQL语句，因此在master上是先删后插，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。因此，使用READ REPEATABLE事务隔离级别就可以避免第一种情况的发生，因而也就避免了master和slave不一致问题的产生。

在MySQL 5.1的版本之后，因为支持了ROW格式的二进制日志记录格式，所以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。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，因此可以避免一些不同步现象的产生。Heikki Tuuri也在http://bugs.mysql.com/bug.php?id=33210这个帖子中建议使用ROW格式的二进制日志。

7.7 分布式事务

InnoDB存储引擎支持XA事务，通过XA事务可以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）参与一个全局的事务中。事务资源通常是关系型数据库系统，但也可以是其他类型的资源。全局事务要求在其中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SERIALIABLE。

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，又可能还有一台服务器是SQL Server数据库的，只要参与全局事

务中的每个节点都支持XA事务。分布式事务可能在银行系统的转账中比较常见，如一个用户需要从上海转10 000元到北京的一个用户上：

```
# Bank@Shanghai:
update account set money = money - 10000 where user='David';

# Bank@Beijing
Update account set money = money + 10000 where user='Mariah';
```

这种情况一定需要分布式的事务，如果不能都提交或都回滚，在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，但是Mariah没收到；又或者是David的账户没有扣款，但是Mariah还是收到钱了。

分布式事务由一个或者多个资源管理器（Resource Managers）、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7-1显示了一个分布式事务的模型：

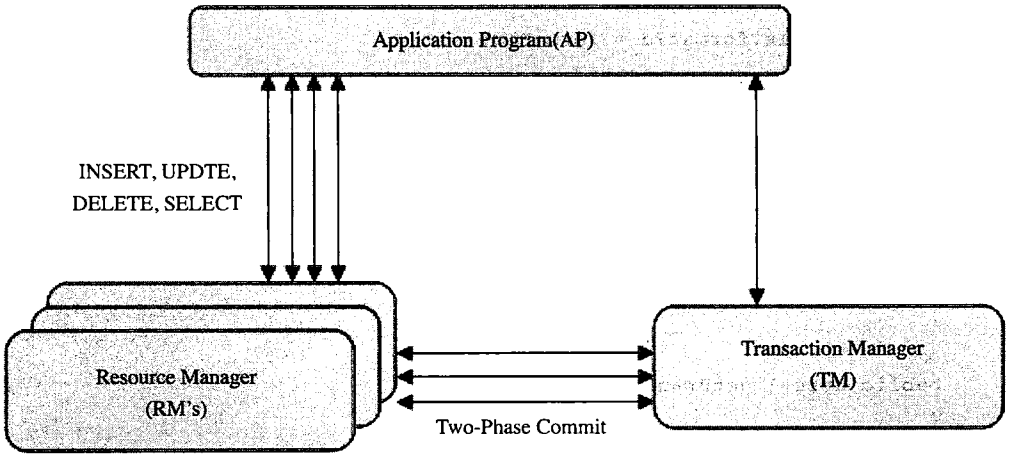


图7-1 分布式事务模型

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。

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，例子就是前面的银行转账，如下所示。

```
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());
              }
    }
}
```

参数innodb_support_xa可以查看是否启用了XA事务支持（默认为ON）：

```
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，对于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。假设我们用START TRANSACTION开启了一个本地的事务，往NDB Cluster存储引擎的表t1插入一条记录，往InnoDB存储引擎的表t2插入一条记录，然后COMMIT。在MySQL内部，也是通过XA事务来协调的，这样才可以保证两张表的原子性。

7.8 不好的事务习惯

7.8.1 在循环中提交

我发现，开发人员非常喜欢在循环中进行事务的提交，下面是他们可能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，在这个例子中，是否加上commit并不关键，因为InnoDB存储引擎默认为自动提交，因此上面的存储过程中去掉commit，结果是完全一样的。这也是另一种容易忽视的问题：

```
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。如我们要插入的是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，那我们应该怎么处理呢？还有一个问题是性能问题，上面两个存储过程都不会比在下面的一个存储过程快，因为它是放在一个事务里：

```
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：

```
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，因此存储过程load1和load2实际写了10 000次，而对于存储过程load3来说，实际只写了1次。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，如下代码所示。

```
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，在他们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，他们可能担心存在Oracle数据库中由于没有足够UNDO产生的Snapshot Too Old的经典问题。MySQL InnoDB存储引擎上述两个问题都没有，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进行提交操作，不论是显式的提交还是隐式的提交。

7.8.2 使用自动提交

自动提交并不是好习惯，因为这对于初级DBA容易犯错，另外对于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数据库默认设置使用自动提交（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、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，在不同的语言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的，而MySQL Python API则是自动执行SET AUTOCOMMIT=0，以禁用自动提交。因此在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。

我认为，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。我曾经见过

很多开发人员没有意识到自动提交这个特性，等到出现错误时应用遇到了大麻烦。

7.8.3 使用自动回滚

InnoDB存储引擎支持通过定义一个HANDLER来进行自动事务的回滚操作，如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首先定义了一个exit类型的handler，当捕获到错误时进行回滚。结构如下所示：

```
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，因此这个存储过程的执行结果如下所示：

```
mysql> call sp_auto_rollback_demo;
Query OK, 0 rows affected (0.06 sec)

mysql> select * from b;
Empty set (0.00 sec)
```

看起来运行没有问题，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呢？对于同样的存储过程sp_auto_rollback_demo，开发人员

可能会进行这样的处理：

```
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。运行正常，返回值1。因此这次运行的结果就会变成：

```
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)
```

看起来我们可以得到运行是否准确的信息。但问题还没有最终解决，对于开发来说，重要的不仅是知道发生了错误，而是发生了什么样的错误。因此自动回滚存在这样的一个问题。

我知道，使用自动回滚大多是以前使用Microsoft SQL Server数据库。在Microsoft SQL Server数据库中，可以使用SET XABORT ON来回滚一个事务。但是Microsoft SQL Server数据库不仅会自动回滚当前的事务，并且还会抛出异常，开发人员可以捕获到这个异常。因此，Microsoft SQL Server数据库和MySQL数据库在这方面是有所不同的。

就像之前小节中所讲到的，对于事务的BEGIN、COMMIT和ROLLBACK操作，应该交给程序端来完成，存储过程只要完成一个逻辑的操作。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，将这些操作都交由程序来完成。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这个程序的结果：

```
[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，我们可以得知发生错误的原因。如上述这个例子中，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，然后可以根据发生的原因来调试我们的程序。

7.9 小结

在这一章中，我们了解了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，这在任何数据库中对于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Atomic（原子性）、Consistency（一致性）、Isolation（隔离性）和Durability（持久性）。隔离性通过第6章介绍过的锁来完成；原子性、一致性、隔离性通过redo和undo来完成。通过对于redo和undo的了解，可以进一步明白事务的工作原理以及更好地使用事务。接着我们讲到了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的。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3°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8章　备份与恢复

对于DBA来说，最基本的工作就是数据库的备份与恢复，在意外情况下（如服务器宕机、磁盘损坏等）要保证数据不丢失，或者是最小程度地丢失。每个DBA应该每时每刻都关心自己所负责的数据库的备份情况。

本章主要介绍对InnoDB存储引擎的备份，MySQL数据库提供的大多数工具（如mysqldump、ibbackup、replication）都能很好地完成备份的工作，当然也可以通过第三方的一些工具来完成，如xtrabackup、LVM快照备份等。DBA应该根据自己的业务要求设计出损失最小、对数据库影响最小的备份策略。

8.1　备份与恢复概述

根据备份的方法可以分为：

- Hot Backup（热备）
- Cold Backup（冷备）
- Warm Backup（温备）

Hot Backup是指在数据库运行中直接备份，对正在运行的数据库没有任何影响。这种方式在MySQL官方手册中称为Online Backup（在线备份）。Cold Backup是指在数据库停止的情况下进行备份，这种备份最为简单，一般只需要拷贝相关的数据库物理文件即可。这种方式在MySQL官方手册中称为Offline Backup（离线备份）。Warm Backup备份同样是在数据库运行时进行，但是会对当前数据库的操作有所影响，例如加一个全局读锁以保证备份数据的一致性。

如果按照备份后文件的内容，又可以分为：

- 逻辑备份
- 裸文件备份

在MySQL数据库中，**逻辑备份**是指备份后的文件内容是可读的，通常是文本文件，内容一般是SQL语句，或者是表内的实际数据，如mysqldump和SELECT * INTO OUTFILE的方法。这类方法的好处是可以看到导出文件的内容，一般适用于数据库的升级、迁移等工作，但是恢复所需要的时间往往较长。

裸文件备份是指拷贝数据库的物理文件，数据库既可以处于运行状态（如ibbackup、xtrabackup这类工具），也可以处于停止状态。这类备份的恢复时间往往较逻辑备份短很多。

若按照备份数据库的内容来分，又可以分为：

- 完全备份
- 增量备份
- 日志备份

完全备份是指对数据库进行一个完整的备份。**增量备份**是指在上次的完全备份基础上，对更新的数据进行备份。**日志备份**主要是指对MySQL数据库二进制日志的备份，通过对一个完全备份进行二进制日志的重做来完成数据库的point-in-time的恢复工作。MySQL数据库复制（Replication）的原理就是异步实时进行二进制日志重做。

对于MySQL数据库来说，官方没有提供真正的增量备份的方法，大部分是通过二进制日志来实现的。这种方法与真正的增量备份相比，效率还是很低的。假设有一个100G的数据库，如果通过二进制日志来完成备份，可能同一个页需要多次执行SQL语句来完成重做的工作。但是对于真正的增量备份来说，只需要记录当前每个页最后的检查点的LSN。如果大于之前完全备份时的LSN，则备份该页，否则不用备份。这大大加快了备份的速度以及缩短了恢复的时间，同时这也是xtrabackup工具增量备份的原理。

此外，还需要理解数据库备份的一致性，这要求在备份的时候数据在这一时间点上是一致的。举例来说，在一个网络游戏中有一个玩家购买了道具，这个事务的过程是：先扣除相应的金钱，然后往其装备表中插入道具，确保扣费和得到的道具是互相一致的。否则，在恢复时，可能出现金钱被扣除了，但是装备丢失的情况。

对于InnoDB存储引擎来说，因为其支持MVCC功能，因此实现备份一致比较容易。可以先开启一个事务，然后导出一组相关的表，最后提交。当然，事务隔离级别必须是REPEATABLE READ的，这样的做法就可以给你一个完美的一致性备份。然而，这个方法

的前提是需要你正确地设计应用程序。上述购买道具的过程不可以分为两个事务来完成，如一个完成扣费，一个完成道具的购买。若备份发生在这两者之间，则会因为逻辑设计的问题导致备份出的数据依然是不一致的。

对于mysqldump备份工具来说，可以通过添加-single-transaction选项来获得InnoDB存储引擎的一致性备份，原理和我们之前所说的相同。需要了解的是，这时的备份是在一个执行时间很长的事务中完成的。另外，对于InnoDB存储引擎的备份，要务必加上-single-transaction的选项（虽然是mysqldump的一个可选选项，但是我找不出任何不加的理由）。

同时，我建议每个公司根据自己的备份策略编写一个备份的应用程序，这个程序可以方便地设置备份的方法以及监控备份的结果，并且可通过第三方接口实时地通知DBA，这样才能真正地做到24×7的备份监控。我们公司开发过一套DAO（Database Admin Online）系统，这套系统完全由DBA开发完成，整个平台用python语言编写，Web操作界面采用Django。利用这个系统，DBA可以方便地对几百台MySQL数据库服务器进行备份，同时查看备份完成后备份文件的状态。之后我们的DBA又对其进行了扩展，不仅可以完成备份工作，还可以实时监控数据库的状态、系统的状态和硬件的状态，当发生问题时，通过飞信接口在第一时间以短信的方式告知DBA。

最后，任何时候都需要做好远程异地备份，也就是容灾的防范。只是同一机房的两台机器的备份是远远不够的。我所在的公司曾在2008年的汶川地震中出现了一个机房可能会被淹的情况，这时远程异地备份就显得至关重要了。

8.2 冷备

对InnoDB存储引擎的**冷备**非常简单，只需要备份MySQL数据库的frm文件、共享表空间文件、独立表空间文件（*.ibd）、重做日志文件。另外，我建议，定期备份MySQL数据库的配置文件my.cnf，这样有利于恢复操作。

通常，DBA会写一个脚本来执行冷备的操作，DBA可能还会对备份完的数据库进行打包和压缩，这并不是一件难事。关键在于，不要遗漏原本需要备份的物理文件，如共享表空间和重做日志文件，少了这些文件数据库可能都无法启动。另外一种经常发生的情况是，由于磁盘空间已满而导致的备份失败，DBA可能习惯性地认为运行脚本的备份是没有问题

的，少了检验的机制。

在同一台机器上对数据库进行冷备是远远不够的，还需要将本地的备份放入一台远程服务器中，以确保不会因为本地数据库宕机而影响备份文件的使用。

冷备的优点是：

- 备份简单，只要拷贝相关文件即可。
- 备份文件易于在不同操作系统、不同MySQL版本上进行恢复。
- 恢复相当简单，只需要把文件恢复到指定位置即可。
- 恢复速度快，不需要执行任何SQL语句，也不需要重建索引。

冷备的缺点是：

- InnoDB存储引擎冷备的文件通常比逻辑文件大很多，因为表空间中存放着很多其他数据，如Undo段、插入缓冲等信息。
- 冷备并不总是可以轻易地跨平台。操作系统、MySQL的版本、文件大小写敏感和浮点数格式都会成为问题。

8.3 逻辑备份

8.3.1 mysqldump

mysqldump备份工具最初由Igor Romanenko编写完成，通常用来完成转存（dump）数据库的备份以及不同数据库之间的移植，例如从低版本的MySQL数据库升级到高版本的MySQL数据库，或者从MySQL数据库移植到Oracle和SQL Server等数据库等。

mysqldump的语法如下：

```
shell> mysqldump [arguments] > file_name
```

如果想要备份所有的数据库，可以使用--all-databaes选项：

```
shell> mysqldump --all-databases > dump.sql
```

如果想要备份指定的数据库，可以使用--databases选项:

```
shell> mysqldump --databases db1 db2 db3 > dump.sql
```

如果想要对test这个架构进行备份，可以使用如下语句：

```
[root@xen-server ~]# mysqldump --single-transaction test > test_backup.sql
```

这就产生了一个对test架构的备份，我们使用--single-transaction选项来保证备份的一致性，备份出的test_backup.sql是文本文件，通过命令cat就可以查看文件的内容：

```
[root@xen-server ~]# cat test_backup.sql
-- MySQL dump 10.13  Distrib 5.5.1-m2, for unknown-linux-gnu (x86_64)
--
-- Host: localhost    Database: test
-- ------------------------------------------------------
-- Server version       5.5.1-m2-log

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Table structure for table 'a'
--

DROP TABLE IF EXISTS 'a';
/*!40101 SET @saved_cs_client = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE 'a' (
  'b' int(11) NOT NULL DEFAULT '0',
  PRIMARY KEY ('b')
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Dumping data for table 'a'
--

LOCK TABLES 'a' WRITE;
/*!40000 ALTER TABLE 'a' DISABLE KEYS */;
INSERT INTO 'a' VALUES (1),(2),(4),(5);
/*!40000 ALTER TABLE 'a' ENABLE KEYS */;
```

```
UNLOCK TABLES;

--
-- Table structure for table 'z'
--

DROP TABLE IF EXISTS 'z';
/*!40101 SET @saved_cs_client = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE 'z' (
  'a' int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Dumping data for table 'z'
--

LOCK TABLES 'z' WRITE;
/*!40000 ALTER TABLE 'z' DISABLE KEYS */;
INSERT INTO 'z' VALUES (1),(1);
/*!40000 ALTER TABLE 'z' ENABLE KEYS */;
UNLOCK TABLES;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
/*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

-- Dump completed on 2010-08-03 13:36:17
```

可以看到，备份出的文件内容就是表结构和数据，所有这些都是用SQL语句表示的。文件开始和结束处的注释是用来设置MySQL数据库的各项参数的，一般用来使还原工作能更有效和准确的进行。之后的部分先是CREATE TABLE语句，之后就是INSERT语句了。

mysqldump的参数选项很多，可以通过mysqldump -help命令来查看所有的参数，有些参数有缩写，如--lock-tables的缩写为-l，这里重点介绍一些比较重要的参数。

- --single-transaction：在备份开始前，先执行START TRANSACTION命令，以此来

获得备份的一致性，当前该参数只对InnoDB存储引擎有效。当启用该参数并进行备份时，确保没有其他任何的DDL语句执行，因为一致性读并不能隔离DDL语句。

- --lock-tables（-l）：在备份中，依次锁住每个架构下的所有表。一般用于MyISAM存储引擎，备份时只能对数据库进行读取操作，不过备份依然可以保证一致性。对于InnoDB存储引擎，不需要使用该参数，用--single-transaction即可，并且-lock-tables和-single-transaction是互斥（exclusive）的，不能同时使用。如果你的MySQL数据库中既有MyISAM存储引擎的表，又有InnoDB存储引擎的表，那么这时你的选择只有--lock-tables了。另外，前面说了，--lock-tables选项是依次对每个架构中的表上锁的，因此只能保证每个架构下表备份的一致性，而不能保证所有架构下表的一致性。
- --lock-all-tables（-x）：在备份过程中，对所有架构中的所有表上锁。这可以避免之前提及的--lock-tables参数不能同时锁住所有表的问题。
- --add-drop-database：在CREATE DATABASE前先运行DROP DATABASE。这个参数需要和-all-databases或者-databases选项一起使用。默认情况下，导出的文本文件中并不会有CREATE DATABASE，除非你指定了这个参数，因此可能会看到如下内容：

```
[root@xen-server ~]# mysqldump --single-transaction --add-drop-database --
databases test > test_backup.sql
[root@xen-server ~]# cat test_backup.sql
-- MySQL dump 10.13  Distrib 5.5.1-m2, for unknown-linux-gnu (x86_64)
......
--
-- Current Database: 'test'
--

/*!40000 DROP DATABASE IF EXISTS 'test'*/;

CREATE DATABASE /*!32312 IF NOT EXISTS*/ 'test' /*!40100 DEFAULT CHARACTER SET latin1 */;

USE 'test';
......
```

- --master-data[=value]：通过该参数产生的备份转存文件主要用来建立一个slave replication。当value的值为1时，转存文件中记录CHANGE MASTER语句；当value的值为2时，CHANGE MASTER语句被写成SQL注释。默认情况下，value的值为空。

当value值为1时，在备份文件中会看到：

```
[root@xen-server ~]# mysqldump --single-transaction --add-drop-database --
master-data=1 --databases test > test_backup.sql
[root@xen-server ~]# cat test_backup.sql
-- MySQL dump 10.13  Distrib 5.5.1-m2, for unknown-linux-gnu (x86_64)
--
-- Host: localhost    Database: test
-- ------------------------------------------------------
-- Server version       5.5.1-m2-log
......

--
-- Position to start replication or point-in-time recovery from
--

CHANGE MASTER TO MASTER_LOG_FILE='xen-server-bin.000006', MASTER_LOG_POS=8095;
......
```

当value为2时，在备份文件中会看到CHANGE MASTER语句被注释了：

```
[root@xen-server ~]# mysqldump --single-transaction --add-drop-database --
master-data=2 --databases test > test_backup.sql
[root@xen-server ~]# cat test_backup.sql
-- MySQL dump 10.13  Distrib 5.5.1-m2, for unknown-linux-gnu (x86_64)
--
-- Host: localhost    Database: test
-- ------------------------------------------------------
-- Server version       5.5.1-m2-log
......
--
-- Position to start replication or point-in-time recovery from
--

--
......
```

- --master-data会自动忽略-lock-tables选项。如果没有使用-single-transaction选项，则会自动使用-lock-all-tables选项。
- --events（-E）：备份事件调度器。
- --routines（-R）：备份存储过程和函数。
- --triggers：备份触发器。

- --hex-blob：将BINARY、VARBINARY、BLOG、BIT列类型备份为十六进制的格式。mysqldump导出的文件一般是文本文件，但是，如果导出的数据中有上述这些类型，文本文件模式下可能有些字符不可见，若添加-he-blob选项，结果会以十六进制的方式显示，如：

```
[root@xen-server ~]# mysqldump --single-transaction --add-drop-database --
master-data=2 --no-autocommit --databases test3 > test3_backup.sql
[root@xen-server ~]# cat test3_backup.sql
-- MySQL dump 10.13  Distrib 5.5.1-m2, for unknown-linux-gnu (x86_64)
--
-- Host: localhost    Database: test3
-- ------------------------------------------------------
-- Server version       5.5.1-m2-log
......
LOCK TABLES 'a' WRITE;
/*!40000 ALTER TABLE 'a' DISABLE KEYS */;
set autocommit=0;
INSERT INTO 'a' VALUES (0x61000000000000000000);
/*!40000 ALTER TABLE 'a' ENABLE KEYS */;
UNLOCK TABLES;
```

可以看到，这里用0x61000000000000000000（十六进制的格式）来导出数据。

- --tab=path（-T path）：产生TAB分割的数据文件。对于每张表，mysqldump创建一个包含CREATE TABLE语句的table_name.sql文件和和包含数据的tbl_name.txt。可以使用--fields-terminated-by=...，--fields-enclosed-by=...，--fields-optionally-enclosed-by=...，--fields-escaped-by=...，--lines-terminated-by=...来改变默认的分割符、换行符等，如：

```
[root@xen-server test]# mysqldump --single-transaction --add-drop-database  --
tab="/usr/local/mysql/data/test" test
[root@xen-server test]# ls -lh
total 244K
-rw-rw---- 1 mysql mysql 8.4K Jul 21 16:02 a.frm
-rw-rw---- 1 mysql mysql  96K Jul 22 17:18 a.ibd
-rw-r--r-- 1 root  root  1.3K Aug  3 15:36 a.sql
-rw-rw-rw- 1 mysql mysql    8 Aug  3 15:36 a.txt
-rw-rw---- 1 mysql mysql   65 Jul 17 15:54 db.opt
-rw-rw---- 1 mysql mysql 8.4K Aug  2 17:22 z.frm
-rw-rw---- 1 mysql mysql  96K Aug  2 17:22 z.ibd
```

```
-rw-r--r-- 1 root  root  1.3K Aug  3 15:36 z.sql
-rw-rw-rw- 1 mysql mysql    4 Aug  3 15:36 z.txt
-----------
-- Server version       5.5.1-m2-log

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Table structure for table 'a'
--

DROP TABLE IF EXISTS 'a';
/*!40101 SET @saved_cs_client = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE 'a' (
  'b' int(11) NOT NULL DEFAULT '0',
  PRIMARY KEY ('b')
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
/*!40101 SET character_set_client = @saved_cs_client */;

/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

-- Dump completed on 2010-08-03 15:36:56
[root@xen-server test]# cat a.txt
1
2
4
5
```

大多数DBA喜欢用SELECT ... INTO OUTFILE的方式来导出一张表，但是通过

mysqldump一样可以完成工作，而且可以一次完成多张表的导出，并且保证导出数据的一致性。

- --where='where_condition' (-w 'where_condition')：导出给定条件的数据。例如，导出b架构下的表a，并且表a的数据大于2如下所示。

```
[root@xen-server bin]# mysqldump --single-transaction --where='b>2' test a > a.sql

[root@xen-server bin]# cat a.sql
-- MySQL dump 10.13  Distrib 5.5.1-m2, for unknown-linux-gnu (x86_64)
--
-- Host: localhost    Database: test
-- ------------------------------------------------------
-- Server version       5.5.1-m2-log
......

--
-- Dumping data for table 'a'
--
-- WHERE:  b>2

LOCK TABLES 'a' WRITE;
/*!40000 ALTER TABLE 'a' DISABLE KEYS */;
INSERT INTO 'a' VALUES (4),(5);
/*!40000 ALTER TABLE 'a' ENABLE KEYS */;
UNLOCK TABLES;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;
......
```

8.3.2 SELECT ... INTO OUTFILE

SELECT...INTO语句也是一种逻辑备份的方法，或者更准确地说是导出一张表中的数据。SELECT ... INTO的语法如下：

```
SELECT [column 1],[column 2] ...
INTO
OUTFILE 'file_name'
[{FIELDS | COLUMNS}
[TERMINATED BY 'string']
[[OPTIONALLY] ENCLOSED BY 'char']
[ESCAPED BY 'char']
]
```

```
[LINES
[STARTING BY 'string']
[TERMINATED BY 'string']
]
FROM TABLE WHERE ......
```

其中字段[TERMINATED BY 'string'] 表示每个列的分隔符，[[OPTIONALLY] ENCLOSED BY 'char']表示对于字符串的包含符，[ESCAPED BY 'char']表示转义符，[STARTING BY 'string']表示每行的开始符号，TERMINATED BY 'string'表示每行的结束符号。如果没有指定任何FIELDS和LINES的选项，默认使用以下的设置：

```
FIELDS TERMINATED BY '\t' ENCLOSED BY '' ESCAPED BY '\\'
LINES TERMINATED BY '\n' STARTING BY ''
```

file_name表示导出的文件，但文件所在的路径的权限必须是mysql：mysql，否则MySQL会报告没有权限导出：

```
mysql> select * into outfile '/root/a.txt' from a;
ERROR 1 (HY000): Can't create/write to file '/root/a.txt' (Errcode: 13)
```

若已经存在该文件，则同样会报错：

```
[root@xen-server ~]# mysql test -e "select * into outfile '/home/mysql/a.txt'
fields terminated by ',' from a";
ERROR 1086 (HY000) at line 1: File '/home/mysql/a.txt' already exists
```

查看通过SELECT INTO导出的表a文件：

```
mysql> select * into outfile '/home/mysql/a.txt' from a;
Query OK, 3 rows affected (0.02 sec)

mysql> quit
Bye
[root@xen-server ~]# cat /home/mysql/a.txt
1       a
2       b
3       c
```

可以发现，默认导出的文件是以TAB进行列分割的，如果想要使用其他分割符，如“,”，则可以使用FIELDS TERMINATED BY 'string'选项，如：

```
[root@xen-server ~]# mysql test -e "select * into outfile '/home/mysql/a.txt'
fields terminated by ',' from a";
```

```
[root@xen-server ~]# cat /home/mysql/a.txt
1,a
2,b
3,c
```

在Windows平台下，因为其换行符是“\r\n”，因此在导出时可能需要指定LINES TERMINATED BY选项，如：

```
[root@xen-server mysql]# mysql test -e "select * into outfile '/home/mysql/a.
txt' fields terminated by ',' lines terminated by '\r\n' from a";

[root@xen-server mysql]# od -c a.txt
0000000   1   ,   a  \r  \n   2   ,   b  \r  \n   3   ,   c  \r  \n
0000017
```

8.3.3 逻辑备份的恢复

mysqldump的恢复操作比较简单，因为备份的文件就是导出的SQL语句，一般只需要执行这个文件就可以了，可以通过以下的方法：

```
[root@xen-server ~]# mysql -uroot -p < test_backup.sql
Enter password:
```

如果在导出时包含了创建和删除数据库的SQL语句，则必须确保删除架构时架构目录下没有其他与数据库无关的文件，否则可能会出现以下的错误：

```
mysql> drop database test;
ERROR 1010 (HY000): Error dropping database (can't rmdir './test', errno: 39)
```

因为逻辑备份的文件是由SQL语句组成的，所以也可以通过SOURCE命令来执行导出的逻辑备份文件，如下所示：

```
mysql> source /home/mysql/test_backup.sql;
Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

......

Query OK, 0 rows affected (0.00 sec)

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

通过mysqldump可以恢复数据库，但是常发生的一个问题是mysqldump可以导出存储过程、触发器、事件、数据，但是却不能导出视图。因此，如果你的数据库中还使用了视图，那么在用mysqldump备份完数据库后还需要导出视图的定义，或者保存视图定义的frm文件，并在恢复时进行导入，这样才能保证mysqldump数据库的完全恢复。

8.3.4 LOAD DATA INFILE

若是通过mysqldump --tab或SELECT INTO OUTFILE导出的数据需要恢复时，这时需要通过LOAD DATA INFILE命令来进行导入，LOAD DATA INFILE的语法如下所示：

```
LOAD DATA [LOW_PRIORITY | CONCURRENT] [LOCAL] INFILE 'file_name'
[REPLACE | IGNORE]
INTO TABLE tbl_name
[CHARACTER SET charset_name]
[{FIELDS | COLUMNS}
[TERMINATED BY 'string']
[[OPTIONALLY] ENCLOSED BY 'char']
[ESCAPED BY 'char']
]
[LINES
[STARTING BY 'string']
[TERMINATED BY 'string']
]
[IGNORE number LINES]
[(col_name_or_user_var,...)]
[SET col_name = expr,...]
```

要对服务器文件使用LOAD DATA INFILE，必须拥有FILE权，其中导入格式的选项和之前介绍的SELECT INTO OUTFILE命令完全一样。IGNORE number LINES选项可以忽略导入的前几行。下面来看一个用LOAD DATA INFILE命令导入文件的示例，并忽略第一行的导入：

```
mysql> load data infile '/home/mysql/a.txt' into table a;
Query OK, 3 rows affected (0.00 sec)
Records: 3  Deleted: 0  Skipped: 0  Warnings: 0
```

为了加快InnoDB存储引擎的导入，你可能希望导入过程忽略对外键的检查，因此可以使用如下方式。

```
mysql> set @@foreign_key_checks=0;
Query OK, 0 rows affected (0.00 sec)

mysql> load data infile '/home/mysql/a.txt' into table a;
Query OK, 4 rows affected (0.00 sec)
Records: 4  Deleted: 0  Skipped: 0  Warnings: 0

mysql>  set @@foreign_key_checks=1;
Query OK, 0 rows affected (0.00 sec)
```

另外可以针对指定的列进行导入，如将数据导入列a、b，而c列等于a、b列之和：

```
mysql> create table b ( a int,b int,c int,primary key(a))engine=innodb;
Query OK, 0 rows affected (0.01 sec)

mysql> load data infile '/home/mysql/a.txt' into table b fields terminated by
',' (a,b) set c=a+b;
Query OK, 4 rows affected (0.01 sec)
Records: 4  Deleted: 0  Skipped: 0  Warnings: 0
```

LOAD DATA INFILE命令可以用来导入数据，但同时可以完成对Linux操作系统的监控。如果需要监控CPU的使用情况，可以通过加载/proc/stat来完成。首先我们需要建立一张监控CPU的表cpu_stat，其结构如下所示：

```
mysql> CREATE TABLE IF NOT EXISTS DBA.cpu_stat (
    ->   id bigint auto_increment primary key,
    ->   value char(25) NOT NULL,
    ->   user bigint,
    ->   nice bigint,
    ->   system bigint,
    ->   idle bigint,
    ->   iowait bigint,
    ->   irq bigint,
    ->   softirq bigint,
    ->   steal bigint,
    ->   guest bigint,
    ->   other bigint,
    ->   time datetime
    -> );
Query OK, 0 rows affected (0.00 sec)
```

接着可以通过用LOAD DATA INFILE命令来加载/proc/stat文件，但需要对其中一些数

值进行转化，命令如下所示：

```
mysql> LOAD DATA INFILE '/proc/stat'
    ->    IGNORE INTO TABLE DBA.cpu_stat
    ->    FIELDS TERMINATED BY ' '
    ->    (@value, @val1, @val2, @val3, @val4, @val5, @val6, @val7, @val8, @val9, @val10)
    -> SET
    ->     value = @value,
    ->     user = IF(@value NOT LIKE 'cpu%', NULL, IF(@value != 'cpu',
IFNULL(@val1, 0), IFNULL(@val2,0))),
    ->     nice = IF(@value NOT LIKE 'cpu%', NULL, IF(@value != 'cpu',
IFNULL(@val2, 0), IFNULL(@val3,0))),
    ->   system = IF(@value NOT LIKE 'cpu%', NULL, IF(@value != 'cpu',
IFNULL(@val3, 0), IFNULL(@val4,0))),
    ->     idle = IF(@value NOT LIKE 'cpu%', NULL, IF(@value != 'cpu',
IFNULL(@val4, 0), IFNULL(@val5,0))),
    ->   iowait = IF(@value NOT LIKE 'cpu%', NULL, IF(@value != 'cpu',
IFNULL(@val5, 0), IFNULL(@val6,0))),
    ->      irq = IF(@value NOT LIKE 'cpu%', NULL, IF(@value != 'cpu',
IFNULL(@val6, 0), IFNULL(@val7,0))),
    ->  softirq = IF(@value NOT LIKE 'cpu%', NULL, IF(@value != 'cpu',
IFNULL(@val7, 0), IFNULL(@val8,0))),
    ->    steal = IF(@value NOT LIKE 'cpu%', NULL, IF(@value != 'cpu',
IFNULL(@val8, 0), IFNULL(@val9,0))),
    ->    guest = IF(@value NOT LIKE 'cpu%', NULL, IF(@value != 'cpu',
IFNULL(@val9, 0), IFNULL(@val10,0))),
    ->    other = IF(@value like 'cpu%', user + nice + system + idle + iowait +
irq + softirq + steal + guest, @val1),
    ->    time = now();
Query OK, 15 rows affected, 1 warning (0.00 sec)
Records: 15  Deleted: 0  Skipped: 0  Warnings: 1
```

接着查看表cpu_stat，可以看到类似如下的内容：

```
mysql> select * from cpu_stat\G;
*************************** 1. row ***************************
     id: 1
  value: cpu
   user: 2632896
   nice: 67761
 system: 688488
   idle: 4136329105
 iowait: 1468238
```

```
     irq: 0
 softirq: 106303
   steal: 148300
   guest: 0
   other: 4141441091
    time: 2010-09-10 12:01:04
*************************** 2. row ***************************
      id: 2
   value: cpu0
    user: 1092690
    nice: 32285
  system: 170083
    idle: 515689748
  iowait: 652274
     irq: 0
 softirq: 13856
   steal: 29217
   guest: 0
   other: 517680153
    time: 2010-09-10 12:01:04
......
*************************** 14. row ***************************
      id: 14
   value: procs_running
    user: NULL
    nice: NULL
  system: NULL
    idle: NULL
  iowait: NULL
     irq: NULL
 softirq: NULL
   steal: NULL
   guest: NULL
   other: 1
    time: 2010-09-10 12:01:04
*************************** 15. row ***************************
      id: 15
   value: procs_blocked
    user: NULL
    nice: NULL
  system: NULL
    idle: NULL
```

```
 iowait: NULL
    irq: NULL
softirq: NULL
  steal: NULL
  guest: NULL
  other: 0
   time: 2010-09-10 12:01:04
15 rows in set (0.00 sec)
```

接着可以设置一个定时器来让MySQL数据库自动地运行上述LOAD DATA INFILE语句，这样就会有每个时间点的CPU信息被记录到表cpu_stat。执行下述语句就可以得到每个时间点上CPU的使用情况：

```
mysql> select
-> 100*((new.user-old.user )/(new.other-old.other)) user,
-> 100*(( new.nice - old.nice ) / ( new.other - old.other ) ) nice,
-> 100 * (( new.system - old.system) / (new.other - old.other) ) system,
-> 100*(( new.idle - old.idle) / ( new.other - old.other ) ) idle,
-> 100*(( new.iowait - old.iowait) / (new.other - old.other)) iowait,
-> 100*(( new.irq - old.irq ) / ( new.other - old.other ) ) irq,
-> 100*((new.softirq - old.softirq)/(new.other - old.other)) softer,
-> 100 * (( new.steal - old.steal ) / ( new.other - old.other ) ) steal,
-> 100 * (( new.guest - old.guest ) / ( new.other - old.other ) ) guest,
-> new.time
-> from DBA.cpu_stat old,
->          DBA.cpu_stat new
-> where new.id -15 = old.id
->       and old.value = 'cpu'
->       and new.value = old.value\G;
*************************** 1. row ***************************
  user: 0.0357
  nice: 0.0036
system: 0.2237
  idle: 99.5964
iowait: 0.1376
   irq: 0.0000
softer: 0.0000
 steal: 0.0029
 guest: 0.0000
  time: 2010-09-10 12:07:25
*************************** 2. row ***************************
```

```
  user: 1.5346
  nice: 0.3534
system: 8.9854
  idle: 85.2297
iowait: 3.8768
   irq: 0.0000
softer: 0.0000
 steal: 0.0202
 guest: 0.0000
  time: 2010-09-10 12:07:38
*************************** 3. row ***************************
  user: 1.4837
  nice: 0.0000
system: 7.9300
  idle: 79.2285
iowait: 11.3476
   irq: 0.0000
softer: 0.0000
 steal: 0.0102
 guest: 0.0000
  time: 2010-09-10 12:07:50
3 rows in set (0.00 sec)
```

同样，我们还可以对/proc/diskstat文件执行如上述所示的操作，这样就又可以对磁盘进行监控操作了。

8.3.5 mysqlimport

mysqlimport是MySQL数据库提供的一个命令行程序，从本质上来说，是LOAD DATA INFILE的命令接口，而且大多数的选项都和LOAD DATA INFILE语法相同。其语法格式如下：

```
shell> mysqlimport [options] db_name textfile1 [textfile2 ...]
```

与LOAD DATA INFILE不同的是，mysqlimport命令是可以导入多张表的，并且通过--user-thread参数来并发导入不同的文件。这里的并发是指并发导入多个文件，并不是指mysqlimport可以并发地导入一个文件，这是有区别的，并且并发地对同一张表进行导入，效果一般都不会比串行的方式好。通过mysqlimport并发地导入两张表。

```
[root@xen-server mysql]# mysqlimport --use-threads=2 test /home/mysql/t.txt
/home/mysql/s.txt
test.s: Records: 5000000  Deleted: 0  Skipped: 0  Warnings: 0
test.t: Records: 5000000  Deleted: 0  Skipped: 0  Warnings: 0
```

如果在上述命令运行的过程中查看MySQL的数据库线程列表，应该可以看到类似如下的内容：

```
mysql> show full processlist\G;
*************************** 1. row ***************************
     Id: 46
   User: rep
   Host: www.dao.com:1028
     db: NULL
Command: Binlog Dump
   Time: 37651
  State: Master has sent all binlog to slave; waiting for binlog to be updated
   Info: NULL
*************************** 2. row ***************************
     Id: 77
   User: root
   Host: localhost
     db: test
Command: Query
   Time: 0
  State: NULL
   Info: show full processlist
*************************** 3. row ***************************
     Id: 83
   User: root
   Host: localhost
     db: test
Command: Query
   Time: 73
  State: NULL
   Info: LOAD DATA   INFILE '/home/mysql/t.txt' INTO TABLE 't' IGNORE 0 LINES
*************************** 4. row ***************************
     Id: 84
   User: root
   Host: localhost
     db: test
Command: Query
```

```
    Time: 73
   State: NULL
    Info: LOAD DATA   INFILE '/home/mysql/s.txt' INTO TABLE 's' IGNORE 0 LINES
4 rows in set (0.00 sec)
```

可以看到mysqlimport实际上是同时执行了2条LOAD DTA INFILE语句来完成并发导入操作。

8.4 二进制日志备份与恢复

二进制日志非常关键，我们可以通过它来完成point-in-time的恢复工作。MySQL数据库的复制同样需要二进制日志。默认情况下并不启用二进制日志，要使用二进制日志，首先必须启用它，在配置文件中进行如下设置：

```
[mysqld]
log-bin
```

在3.2.4小节中已经阐述过，对于InnoDB存储引擎只是简单启用二进制日志是不够的，还需要启用一些其他参数来保证安全和正确地记录二进制日志，推荐的二进制日志的服务器配置应该是：

```
[mysqld]
log-bin
sync_binlog = 1
innodb_support_xa = 1
```

备份二进制日志文件前，可以通过FLUSH LOGS命令来生成一个新的二进制日志文件，然后备份之前的二进制日志。

要恢复二进制日志也非常简单，通过mysqlbinlog即可，mysqlbinlog的使用方法如下：

```
shell> mysqlbinlog [options] log_file ...
```

例如，要还原binlog.0000001，可以使用如下命令：

```
shell>mysqlbinlog binlog. 0000001 |  mysql -uroot -p test
```

如果需要恢复多个二进制日志文件，最正确的做法应该是同时恢复多个二进制日志文件，而不是一个一个地恢复，如：

```
shell> mysqlbinlog binlog.[0-10]* | mysql -u root -p test
```

也可以先通过mysqlbinlog命令导出到一个文件，然后再通过SOURCE命令来导入。这种做法的好处是，可以对导出的文件进行修改后再导入，如：

```
shell> mysqlbinlog binlog.000001 > /tmp/statements.sql
shell> mysqlbinlog binlog.000002 >> /tmp/statements.sql
shell> mysql -u root -p -e "source /tmp/statements.sql"
```

--start-position和--stop-position选项可以用来指定从二进制日志的某个偏移量来进行恢复，这样可以跳过某些不正确的语句，如：

```
shell>mysqlbinlog --start-position=107856 binlog. 0000001 |  mysql -uroot -p test
```

--start-datetime和--stop-datetime选项可以用来指定从二进制日志的某个时间点来进行恢复，用法和--start-position和--stop-position选项基本相同。

8.5 热备

8.5.1 ibbackup

ibbackup是InnoDB存储引擎官方提供的热备工具，可以同时备份MyISAM存储引擎表和InnoDB存储引擎表。InnoDB存储引擎表的备份工作原理如下：

（1）记录备份开始时，InnoDB存储引擎重做日志文件检查点的LSN。

（2）拷贝共享表空间文件以及独立表空间文件。

（3）记录拷贝完表空间文件后，InnoDB存储引擎重做日志文件检查点的LSN。

（4）拷贝在备份时产生的重做日志。

对于事务型的数据库，如SQL Server数据库、Oracle数据库，热备的原理与上述大致相同。可以发现，在备份期间不会对数据库本身有任何影响，所做的操作只是拷贝数据库文件，因此任何对数据库的操作都是允许的，不会出现阻塞情况。因此，ibbackup的优点如下：

- 在线备份。不阻塞任何SQL语句。
- 备份性能好。备份的实质是复制数据库文件和重做日志文件。
- 支持压缩备份。通过选项，可以支持不同级别的压缩。
- 跨平台支持。ibbackup可以运行在Linux、Windows以及主流的Unix系统平台上。

ibbackup对InnoDB存储引擎表的恢复过程如下：

（1）恢复表空间文件。

（2）应用重做日志文件恢复InnoDB存储引擎表。

ibbackup提供了一种高性能的热备方式，是InnoDB存储引擎备份的首选方式。不过它是收费软件，好在Pecona公司给我们带来了开源、免费的XtraBackup热备工具，它可以实现ibbackup的所有功能，并且还扩展支持真正的增量备份功能。因此，更好的选择应该是使用XtraBackup来完成热备的工作。

8.5.2 XtraBackup

XtraBackup支持MySQL 数据库5.0版和5.1版，同时也支持InnoDB Plugin版本新增的Barracuda页格式。XtraBackup在GPL v2开源协议下发布，官方网站地址是：https://launchpad.net/percona-xtrabackup。

xtrabackup命令的使用方法如下：

```
xtrabackup --backup | --prepare [OPTIONS]
```

xtrabackup命令的可选参数如下所示：

```
(The defaults options should be given as the first argument)
--print-defaults        Prints the program's argument list and exit.
--no-defaults           Don't read the default options from any file.
--defaults-file=        Read the default options from this file.
--defaults-extra-file=  Read this file after the global options files have been
read.
--target-dir=           The destination directory for backups.
--backup                Make a backup of a mysql instance.
--stats                 Calculate the statistic of the datadir (it is
recommended you take mysqld offline).
--prepare               Prepare a backup so you can start mysql server with your
restore.
--export                Create files to import to another database after it has
been prepared.
--print-param           Print the parameters of mysqld that you will need for a
for copyback.
--use-memory=           This value is used instead of buffer_pool_size.
--suspend-at-end        Creates a file called xtrabackup_suspended and waits
until the
```

```
                        user deletes that file at the end of the backup.
--throttle=             (use with --backup) Limits the IO operations (pairs of
reads and writes)
                        per second to the values set here.
--log-stream            outputs the contents of the xtrabackup_logfile to
stdout.
--incremental-lsn=      (use with --backup) Copy only .ibd pages newer than
the specified LSN high:low.
                        ##ATTENTION##: checkpoint lsn *must* be used. Be Careful!
--incremental-basedir=  (use with --backup) Copy only .ibd pages newer than
                        the existing backup at the specified directory.
--incremental-dir=      (use with --prepare) Apply .delta files and logfiles
located in the specified directory.
--tables=name            Regular Expression list of table names to be backed up.
--create-ib-logfile     (NOT CURRENTLY IMPLEMENTED) will create ib_logfile*
after a --prepare.
                        ### If you want to create ib_logfile* only re-execute
this command using the same options. ###
--datadir=name          Path to the database root.
--tmpdir=name           Path for temporary files. Several paths may be
specified as a colon (:) separated string.
                        If you specify multiple paths they are used round-robin.
```

如果我们要做一个完全备份，可以执行如下命令：

```
# ./xtrabackup --backup
./xtrabackup  Ver alpha-0.2 for 5.0.75 unknown-linux-gnu (x86_64)
>> log scanned up to (0 1009910580)
Copying ./ibdata1
     to /home/kinoyasu/xtrabackup_work/mysql-5.0.75/innobase/xtrabackup/tmp2/ibdata1
        ...done
Copying ./tpcc/stock.ibd
     to /home/kinoyasu/xtrabackup_work/mysql-5.0.75/innobase/xtrabackup/tmp2/tpcc/stock.ibd
        ...done
Copying ./tpcc/new_orders.ibd
     to /home/kinoyasu/xtrabackup_work/mysql-5.0.75/innobase/xtrabackup/tmp2/tpcc/new_orders.ibd
        ...done
Copying ./tpcc/history.ibd
     to /home/kinoyasu/xtrabackup_work/mysql-5.0.75/innobase/xtrabackup/tmp2/tpcc/history.ibd
        ...done
Copying ./tpcc/customer.ibd
     to /home/kinoyasu/xtrabackup_work/mysql-5.0.75/innobase/xtrabackup/tmp2/tpcc/customer.ibd
>> log scanned up to (0 1010561109)
        ...done
```

```
Copying ./tpcc/district.ibd
       to /home/kinoyasu/xtrabackup_work/mysql-5.0.75/innobase/xtrabackup/tmp2/
tpcc/district.ibd
        ...done
Copying ./tpcc/item.ibd
       to /home/kinoyasu/xtrabackup_work/mysql-5.0.75/innobase/xtrabackup/tmp2/
tpcc/item.ibd
        ...done
Copying ./tpcc/order_line.ibd
       to /home/kinoyasu/xtrabackup_work/mysql-5.0.75/innobase/xtrabackup/tmp2/
tpcc/order_line.ibd
>> log scanned up to (0 1012047066)
        ...done
Copying ./tpcc/orders.ibd
       to /home/kinoyasu/xtrabackup_work/mysql-5.0.75/innobase/xtrabackup/tmp2/
tpcc/orders.ibd
        ...done
Copying ./tpcc/warehouse.ibd
       to /home/kinoyasu/xtrabackup_work/mysql-5.0.75/innobase/xtrabackup/tmp2/
tpcc/warehouse.ibd
        ...done
>> log scanned up to (0 1014592707)
Stopping log copying thread..
Transaction log of lsn (0 1009910580) to (0 1014592707) was copied.
```

8.5.3 XtraBackup实现增量备份

MySQL数据库本身提供的工具并不支持真正的增量备份，更准确地说，二进制日志的恢复应该是point-in-time的恢复而不是增量备份。XtraBackup工具支持对InnoDB存储引擎的增量备份，其工作原理如下：

（1）首先完成一个完全备份，并记录下此时检查点的LSN。

（2）在进行增量备份时，比较表空间中每个页的LSN是否大于上次备份时的LSN，如果是，则备份该页，同时记录当前检查点的LSN。

因此XtraBackup的备份和恢复过程大致如下：

```
(full backup)
# ./xtrabackup --backup --target-dir=/backup/base
...
```

```
(incremental backup)
# ./xtrabackup --backup --target-dir=/backup/delta --incremental-
basedir=/backup/base
...

(prepare)
# ./xtrabackup --prepare --target-dir=/backup/base
...

(apply incremental backup)
# ./xtrabackup --prepare --target-dir=/backup/base --incremental-
dir=/backup/delta
...
```

上述过程中，首先将全部文件备份到/backup/base目录下，增量备份产生的文件备份到/backup/delta。恢复过程中，首先指定完全备份的路径，然后将增量备份应用于该完全备份。以下显示了一个完整的增量备份过程：

```
# ./xtrabackup --backup
./xtrabackup  Ver beta-0.4 for 5.0.75 unknown-linux-gnu (x86_64)
>> log scanned up to (0 378161500)
...
The latest check point (for incremental): '0:377883685'     <===== 使用这个LSN
>> log scanned up to (0 379294296)
Stopping log copying thread..
Transaction log of lsn (0 377883685) to (0 379294296) was copied.

(must do --prepare before the each incremental backup)
# ./xtrabackup --prepare
...

# ./xtrabackup --backup --incremental=0:377883685
incremental backup from 0:377883685 is enabled.
./xtrabackup  Ver beta-0.4 for 5.0.75 unknown-linux-gnu (x86_64)
>> log scanned up to (0 379708047)
Copying ./ibdata1
     to /home/kinoyasu/xtrabackup_work/mysql-5.0.75/innobase/xtrabackup/
tmp_diff/ibdata1.delta
        ...done
...
The latest check point (for incremental): '0:379438233'     <==== 下一个增量备份开始的LSN
>> log scanned up to (0 380663549)
Stopping log copying thread..
Transaction log of lsn (0 379438233) to (0 380663549) was copied.
```

8.6 快照备份

MySQL数据库本身并不支持快照功能，因此快照备份是指通过文件系统支持的快照功能对数据库进行备份。备份的前提是将所有数据库文件放在同一文件分区中，然后对该分区执行快照工作。支持快照功能的文件系统和设备包括FreeBSD的UFS文件系统，Solaris的ZFS文件系统，GNU/Linux的逻辑卷管理器（Logical Volume Manager、LVM）等。这里以*LVM*为例进行介绍，UFS和ZFS的快照实现大致和LVM相似。

LVM是Linux系统下对磁盘分区进行管理的一种机制。LVM在硬盘和分区之上建立一个逻辑层来提高磁盘分区管理的灵活性。管理员可以通过LVM系统轻松管理磁盘分区，例如，将若干个磁盘分区连接为一个整块的卷组（Volume Group），形成一个存储池。管理员可以在卷组上随意创建逻辑卷（Logical Volume），并进一步在逻辑卷上创建文件系统。管理员通过LVM可以方便地调整卷组的大小，并且可以对磁盘存储按照组的方式进行命名、管理和分配。简单地说，我们可以通过LVM由物理块设备（如硬盘等）创建物理卷，由一个或多个物理卷创建卷组，最后从卷组中创建任意数量的逻辑卷（不超过卷组大小），如图8-1所示。

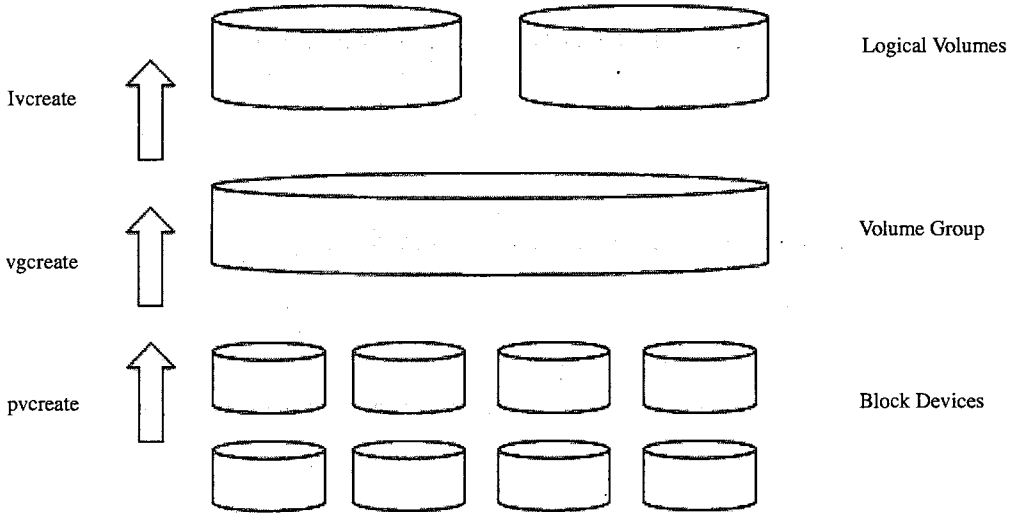


图8-1 LVM的工作原理

图8-2显示了由多块磁盘组成的逻辑卷LV0。

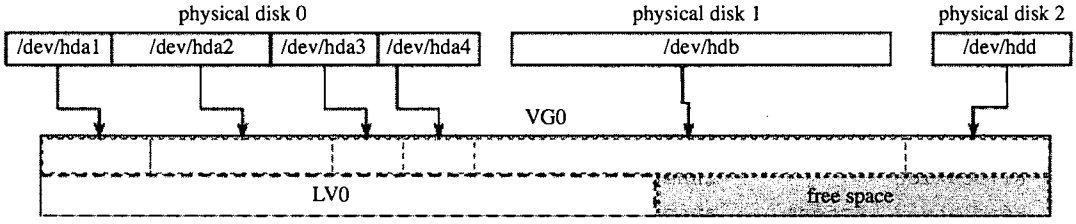


图8-2 物理到逻辑卷的映射

通过vgdisplay命令查看系统中有哪些卷组，如：

```
[root@nh124-98 ~]# vgdisplay
  --- Volume group ---
  VG Name               rep
  System ID
  Format                lvm2
  Metadata Areas        1
  Metadata Sequence No  1873
  VG Access             read/write
  VG Status             resizable
  MAX LV                0
  Cur LV                3
  Open LV               1
  Max PV                0
  Cur PV                1
  Act PV                1
  VG Size               260.77 GB
  PE Size               4.00 MB
  Total PE              66758
  Alloc PE / Size       66560 / 260.00 GB
  Free  PE / Size       198 / 792.00 MB
  VG UUID               MQJiye-j4NN-LbZG-F3CQ-UdTU-fo9D-RRfXD5
```

vgdisplay命令的输出结果显示当前系统有一个rep的卷组，大小为260.77GB，该卷组访问权限是read/write等。可以用命令lvdisplay来查看当前系统中有哪些逻辑卷：

```
[root@nh124-98 ~]# lvdisplay
  --- Logical volume ---
  LV Name                /dev/rep/repdata
  VG Name                rep
  LV UUID                7tOlDt-seKZ-ChpY-QMXC-WaFD-zXAl-MRbofK
  LV Write Access        read/write
  LV snapshot status     source of
                         /dev/rep/dho_datasnapshot100805143507 [active]
```

```
                        /dev/rep/dho_datasnapshot100805163504 [active]
  LV Status             available
  # open                1
  LV Size               100.00 GB
  Current LE            25600
  Segments              1
  Allocation            inherit
  Read ahead sectors    auto
  - currently set to    256
  Block device          253:0

  --- Logical volume ---
  LV Name                /dev/rep/dho_datasnapshot100805143507
  VG Name                rep
  LV UUID                fSSXzh-IBnZ-aZIn-eP03-b7pk-CPjN-5xUktE
  LV Write Access        read only
  LV snapshot status     active destination for /dev/rep/repdata
  LV Status              available
  # open                 0
  LV Size                100.00 GB
  Current LE             25600
  COW-table size         80.00 GB
  COW-table LE           20480
  Allocated to snapshot  0.13%
  Snapshot chunk size    4.00 KB
  Segments               1
  Allocation             inherit
  Read ahead sectors     auto
  - currently set to     256
  Block device           253:1

  --- Logical volume ---
  LV Name                /dev/rep/dho_datasnapshot100805163504
  VG Name                rep
  LV UUID                3B9NP1-qWVG-pfJY-Bdgm-DIdD-dUMu-s2L6qJ
  LV Write Access        read only
  LV snapshot status     active destination for /dev/rep/repdata
  LV Status              available
  # open                 0
  LV Size                100.00 GB
  Current LE             25600
  COW-table size         80.00 GB
  COW-table LE           20480
  Allocated to snapshot  0.02%
```

```
Snapshot chunk size     4.00 KB
Segments                1
Allocation              inherit
Read ahead sectors      auto
- currently set to      256
Block device            253:4
```

可以看到，一共有3个逻辑卷，都属于卷组rep，每个逻辑卷的大小都是100GB。/dev/rep/repdata这个逻辑卷有两个只读快照，并且当前都是激活状态的。

LVM使用了写时复制（Copy-on-write）技术来创建快照。当创建一个快照时，仅拷贝原始卷里数据的元数据，并不会有数据的物理操作，因此快照的创建过程是非常快的。当快照创建完成后，原始卷上有写操作时，快照会跟踪原始卷块的改变，将要改变的数据在改变之前拷贝到快照预留的空间里，因此这个原理的实现叫做写时复制。而对于快照的读取操作，如果读取的数据块是创建快照后没有修改过的，那么会将读操作直接重定向到原始卷上；如果要读取的是已经修改过的块，则将读取保存在快照中的原始数据。因此，采用写时复制机制保证了读取快照时得到的数据与快照创建时的数据是一致的。

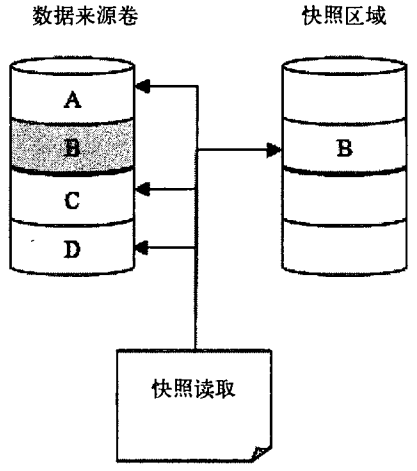


图8-3 LVM的快照读取

图8-3显示了LVM的快照读取，可见B区块被修改了，因此历史数据放入了快照区域。读取快照数据时，A、C、D块还是从原有卷中读取，而B块就需要从快照读取了。

可以用命令lvcreate来创建一个快照，--permission r表示创建的快照是只读的：

```
[root@nh119-215 data]# lvcreate --size 100G --snapshot --permission r -n
```

```
datasnapshot /dev/rep/repdata
  Logical volume "datasnapshot" created
```

在快照制作完成后，可以用lvdisplay命令来查看，输出中的COW-table size字段表示该快照最大空间的大小，Allocated to snapshot字段表示该快照当前空间的使用状况：

```
[root@nh124-98 ~]# lvdisplay
......
--- Logical volume ---
  LV Name                /dev/rep/dho_datasnapshot100805163504
  VG Name                rep
  LV UUID                3B9NP1-qWVG-pfJY-Bdgm-DIdD-dUMu-s2L6qJ
  LV Write Access        read only
  LV snapshot status     active destination for /dev/rep/repdata
  LV Status              available
  # open                 0
  LV Size                100.00 GB
  Current LE             25600
  COW-table size         80.00 GB
  COW-table LE           20480
  Allocated to snapshot  0.04%
  Snapshot chunk size    4.00 KB
  Segments               1
  Allocation             inherit
  Read ahead sectors     auto
  - currently set to     256
  Block device           253:4
```

可以看到，当前快照只使用了0.04%的空间。快照在最初创建时总是很小的，只有当源数据卷的数据不断被修改，这些数据库才会放入快照空间，这时快照的大小才会慢慢增大。

用LVM快照备份InnoDB存储引擎表相当简单，只要把与InnoDB存储引擎相关的文件如共享表空间、独立表空间、重做日志文件等放在同一个逻辑卷中，然后对这个逻辑卷进行快照备份即可。

在对InnoDB存储引擎文件做快照时，数据库无须关闭，可以进行在线备份。虽然此时数据库中可能还有任务需要往磁盘上写数据，但这不会妨碍备份的正常进行。因为InnoDB存储引擎是事务安全的引擎，在下次恢复时，数据库会自动检查表空间中页的状态，并决定是否应用重做日志，恢复就好像数据库被意外重启了。

启用LVM快照需要规划以下几个方面。

- 快照空间大小的划分。如果源逻辑卷的大小是100G，快照最大可能生产的空间是100GB，但是如果每4个小时生成一份快照，则无须划分100G的空间，只要判断在4小时内最多产生的快照空间即可。
- 快照启用后磁盘的性能。启用快照后，磁盘的性能必定会有所下降，因为第一次修改一个数据块时会将该块复制到快照空间。虽然之后的修改不会再复制到快照空间，但磁盘的开销显然还是有所增大。需要判断这些性能上的损失数据库是否可以承担。

不能把快照备份当作完全备份来使用，因为当数据库所在服务器发生硬件故障时，如RAID卡损坏，这时快照备份是不能进行恢复的。快照备份更偏向于对误操作的防范，可以将数据库迅速地恢复到快照产生的时间点，然后再根据二进制日志执行point-in-time的恢复。就我的习惯，我通常把LVM的备份放在replication的Slave服务器上。

8.7 复制

8.7.1 复制的工作原理

复制是MySQL数据库提供的一种高可用、高性能的解决方案，一般用来建立大型的应用。总体来说，复制的工作原理分为以下三个步骤：

（1）主服务器把数据更新记录到二进制日志中。

（2）从服务器把主服务器的二进制日志拷贝到自己的中继日志（Relay Log）中。

（3）从服务器重做中继日志中的时间，把更新应用到自己的数据库上。

工作原理并不复杂，其实就是完全备份和二进制日志备份的还原。不同的是，这个二进制日志的还原操作基本上是实时进行的。注意，不是完全的实时，而是异步的实时。其中存在主从服务器之间的执行延时，如果主服务器的压力很大，则这个延时可能更长。其工作原理如图8-4所示。

从服务器有两个线程：一个是I/O线程，负责读取主服务器的二进制日志，并将其保存为中继日志；另一个是SQL线程，复制执行中继日志。在MySQL 4.0版本之前，从服务器只有1个线程，既负责读取二进制日志，又负责执行二进制日志中的SQL语句。这种方式不符合高性能的要求，已被淘汰。因此，如果查看一个从服务器的状态，应该可以看到类

似如下的内容。

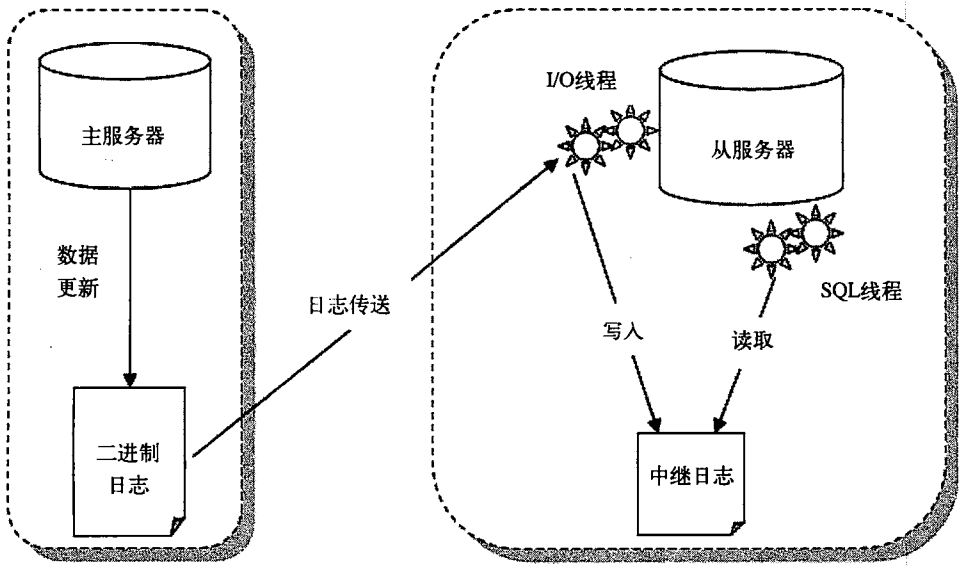


图8-4 MySQL数据库复制工作原理

```
mysql> show full processlist\G;
*************************** 1. row ***************************
     Id: 1
   User: system user
   Host:
     db: NULL
Command: Connect
   Time: 6501
  State: Waiting for master to send event
   Info: NULL
*************************** 2. row ***************************
     Id: 2
   User: system user
   Host:
     db: NULL
Command: Connect
   Time: 0
  State: Has read all relay log; waiting for the slave I/O thread to update it
   Info: NULL
*************************** 3. row ***************************
     Id: 206
   User: root
   Host: localhost
```

```
     db: NULL
Command: Query
   Time: 0
  State: NULL
   Info: show full processlist
3 rows in set (0.00 sec)
```

可以看到，ID为1的线程就是I/O线程，目前的状态是等待主服务器发送二进制日志。ID为2的线程是SQL线程，负责执行读取中继日志并执行。目前的状态是已读取所有的中继日志，等待中继日志被I/O线程更新。

在复制的主服务器上应该可以看到有一个线程负责发送二进制日志，类似如下的内容：

```
mysql> show full processlist\G;
......
*************************** 65. row ***************************
     Id: 26541
   User: rep
   Host: 192.168.190.98:39549
     db: NULL
Command: Binlog Dump
   Time: 6857
  State: Has sent all binlog to slave; waiting for binlog to be updated
   Info: NULL
......
```

之前已经说过，MySQL的复制是异步同步的，并非完全的主从同步。要想查看当前的延迟，可以使用命令SHOW SLAVE STATUS和SHOW MASTER STATUS，如下所示：

```
mysql> show slave status\G;
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 192.168.190.10
                  Master_User: rep
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-bin.000007
          Read_Master_Log_Pos: 555176471
               Relay_Log_File: gamedb-relay-bin.000048
                Relay_Log_Pos: 224355889
        Relay_Master_Log_File: mysql-bin.000007
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
```

```
              Replicate_Do_DB:
          Replicate_Ignore_DB:
           Replicate_Do_Table:
       Replicate_Ignore_Table:
      Replicate_Wild_Do_Table:
  Replicate_Wild_Ignore_Table: mysql.%,DBA.%
                   Last_Errno: 0
                   Last_Error:
                 Skip_Counter: 0
          Exec_Master_Log_Pos: 555176471
              Relay_Log_Space: 224356045
              Until_Condition: None
               Until_Log_File:
                Until_Log_Pos: 0
           Master_SSL_Allowed: No
           Master_SSL_CA_File:
           Master_SSL_CA_Path:
              Master_SSL_Cert:
            Master_SSL_Cipher:
               Master_SSL_Key:
        Seconds_Behind_Master: 0
Master_SSL_Verify_Server_Cert: No
                Last_IO_Errno: 0
                Last_IO_Error:
               Last_SQL_Errno: 0
               Last_SQL_Error:
1 row in set (0.00 sec)
```

通过SHOW SLAVE STATUS命令可以观察当前复制的运行状态，主要它的参数有：

- Slave_IO_State。显示当前IO线程的状态，上述状态显示的是等待主服务器发送二进制日志。
- Master_Log_File。显示当前同步的主服务器的二进制日志，上述状态显示当前同步的是主服务器的mysql-bin.000007。
- Read_Master_Log_Pos。显示当前同步到主服务器上二进制日志的偏移量位置，单位是字节。上述示例显示了当前同步到mysql-bin.000007的555176471偏移量位置，即已经同步了mysql-bin.000007这个二进制日志中529M（555176471/1024/1024）的内容。
- Relay_Master_Log_File。当前中继日志同步的二进制日志。
- Relay_Log_File。显示当前写入的中继日志。

- Relay_Log_Pos。显示当前执行到中继日志的偏移量位置。
- Slave_IO_Running。从服务器中IO线程的运行状态，YES表示运行正常。
- Slave_SQL_Running。从服务器中SQL线程的运行状态，YES表示运行正常。
- Exec_Master_Log_Pos。表示同步到主服务器的二进制日志偏移量的位置。(Read_Master_Log_Pos - Exec_Master_Log_Pos）可以表示当前SQL线程运行的延时，单位是字节。上述示例显示当前主从服务器是完全同步的。

命令SHOW MASTER STATUS可以用来查看主服务器中二进制日志的状态，如：

```
mysql> show master status\G;
*************************** 1. row ***************************
            File: mysql-bin.000007
        Position: 606181078
    Binlog_Do_DB:
Binlog_Ignore_DB:
1 row in set (0.01 sec)
```

可以看到，当前二进制日志记录了偏移量606181078的位置，该值减去这一时间点时从服务上的Read_Master_Log_Pos，就可以得知I/O线程的延时。

一个好的数据库复制监控不仅要监控从服务器上的I/O线程和SQL线程是否运行正常，而且还应该监控从服务器和主服务器之间的延迟，确保从服务器上的数据库状态总是非常接近主服务器上数据库的状态。

8.7.2 快照+复制的备份架构

复制可以用来作为备份，但其功能不仅限于备份，其主要功能如下：

- 数据分布。由于MySQL数据库提供的复制并不需要很大的带宽，因此可以在不同的数据中心之间实现数据的拷贝。
- 读取的负载平衡。通过建立多个从服务器，可将读取平均地分布到这些从服务器中，从而减少主服务器的压力。一般可以通过DNS的Round-Robin和Linux的LVS功能实现负载平衡。
- 数据库备份。复制对备份很有帮助，但是从服务器不是备份，不能完全代替备份。
- 高可用性和故障转移。通过复制建立的从服务器有助于故障转移，减少故障的停机

时间和恢复时间。

可见，复制的设计目的不是简简单单用来备份的，并且只用复制来进行备份是远远不够的。假设当前应用采用了主从式的复制架构，从服务器用来作为备份，一个不太有经验的DBA执行了误操作，如DROP DATABASE或者DROP TABLE，这时从服务器也跟着运行了，那这时如何从服务器进行恢复呢？

一种比较好的方法是通过对从服务器上的数据库所在的分区做快照，以此来避免复制对误操作的处理能力。当主服务器上发生误操作时，只需要恢复从服务器上的快照，然后再根据二进制日志执行point-in-time的恢复即可。因此，快照+复制的备份架构如图8-5所示：

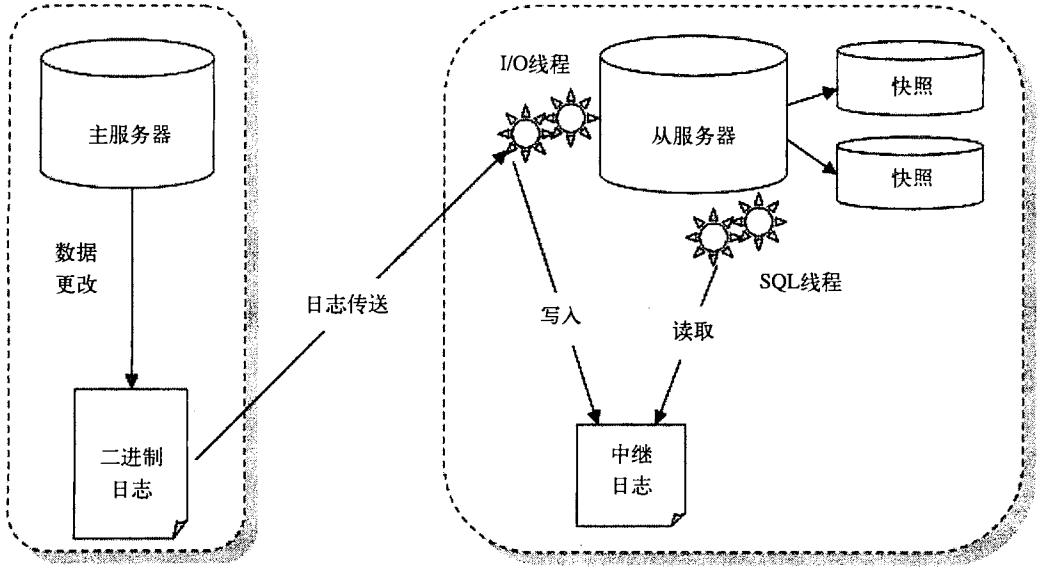


图8-5 快照+复制的备份架构

还有一些其他方法可用来调整复制，比如采用延时复制，即间歇性地开启从服务器上的同步功能，保证大约一小时的延迟。这的确也是一个方法，只是数据库在高峰和非高峰期间每小时产生的二进制日志量是不同的，很难精准地控制。另外，这种方法也不能完全防止误操作的发生。

从服务器上还可以启用read-only选项：

```
[mysqld]
```

```
read-only
```

启用read-only选项后，如果操作从服务器的用户没有SUPER权限，则对从服务器执行任何修改操作都会抛出一个错误，如：

```
mysql> insert into z select 2;
ERROR 1290 (HY000): The MySQL server is running with the --read-only option so
it cannot execute this statement
```

8.8 小结

本章首先介绍了不同的备份类型，并讲解了MySQL数据库常用的一些备份方式。其中主要讲解对InnoDB存储引擎的备份，mysqldump和xtrabackup等工具都可以很好地对InnoDB存储引擎表进行在线热备份工作。最后，讲解了复制，通过快照和复制技术的结合，可以保证我们得到一个实时的在线MySQL备份解决方案。

第9章 性能调优

性能优化不是一项简单的工作，但也不是复杂的难事，关键在于对InnoDB存储引擎特性的了解。如果之前各章的内容你已经完全理解并掌握了，那么你应该基本掌握了如何使得InnoDB存储引擎更好地为你工作。本章节将从以下几个方面集中讲解InnoDB的性能问题：

- 选择合适的CPU。
- 内存的重要性。
- 硬盘对数据库性能的影响。
- 合理地设置RAID。
- 操作系统的选择也很重要。
- 不同文件系统对数据库的影响。
- 选择合适的基准测试工具。

9.1 选择合适的CPU

首先要清楚数据库应用的分类，一般分为两类：OLTP（Online Transaction Processing，在线事务处理）和OLAP（Online Analytical Processing，在线分析处理），这是两种完全不同的数据库应用。OLAP多用在数据仓库或数据集市中，一般需要执行复杂的SQL语句来进行查询；OLTP多用在日常的事物处理应用中，如银行交易、在线商品交易、Blog、网络游戏等应用。相对于OLAP，数据库的容量较小。

InnoDB存储引擎一般都应用于OLTP的数据库应用，这种应用的特点如下所示：

- 用户操作的并发量大。
- 事务处理的时间一般比较短。
- 查询的语句较为简单，一般都走索引。
- 复杂的查询较少。

可以看出，OLTP的数据库应用本身对CPU的要求并不高，因为复杂的查询可能需要执行比较、排序、连接等非常耗CPU的操作，这些操作在OLTP的数据库应用中很少发生。因此，可以说OLAP是CPU密集型的操作，而OLTP是IO密集型的操作。建议在采购设备时，应将更多的注意力放在提高IO的配置上。

此外，为了获得更多内存的支持，CPU必须支持64位的应用，否则无法支持64位操作系统的安装。因此，为新的应用选择64位的CPU是必要的前提。现在4核的CPU已经非常普遍，而今年Intel和AMD又都推出了6核的CPU，将来随着操作系统的升级，我们还可能看到128核的CPU，这都需要数据库更好地对其进行支持。

从InnoDB存储引擎的设计架构上来看，其主要的后台操作都是在一个单独的MASTER THREAD中完成的，因此并不能很好地支持多核的应用。当然，开源社区已经通过多种方法来改变这种局面，新的InnoDB Plugin版本在各种测试下已经显示对多核CPU的处理性能有了极大的提高。因此，如果你的CPU支持多核，InnoDB Plugin是更好的选择。另外，如果你的CPU是多核的，你可以通过修改参数innodb_read_io_threads和innodb_write_io_threads来增大IO的线程，这样也能更充分利用CPU的多核性能。

在当前的MySQL版本中，一条SQL查询语句只能在一个CPU行工作，并不支持多CPU的处理。OLTP的数据库应用操作一般都很简单，因此对OLTP应用的影响并不是很大。但是，多个CPU或多核CPU对处理大并发量的请求还有非常有帮助的。

9.2 内存的重要性

内存的大小最能直接反应数据库的性能。通过前面的章节我们已经了解到，InnoDB存储引擎既缓存数据，又缓存索引，并将其缓存于一个很大的缓冲池中，我们将这个缓冲池称为InnoDB Buffer Pool，它的大小直接影响了数据库的性能。Percona公司的CTO Vadim对此做了一次测试，以此反应内存的重要性，结果如图9-1所示。

在上述的测试中，数据和索引总大小为18GB，然后将缓冲池的大小分别设为2GB、4GB、6GB、8GB、10GB、12GB、14GB、16GB、18GB、20GB、22GB，再进行sysbench的测试。可以发现，随着缓冲池的增大，测试结果TPS（Transaction Per Second）会线性增长。当缓冲池增大到20GB和22GB，数据库的性能有了极大的提高，因为这时缓

冲池的大小已经大于数据文件本身的大小，所有对数据文件的操作都可以在内存中进行，因此这时的性能应该是最优的，再调大缓冲池并不能再提高数据库的性能。

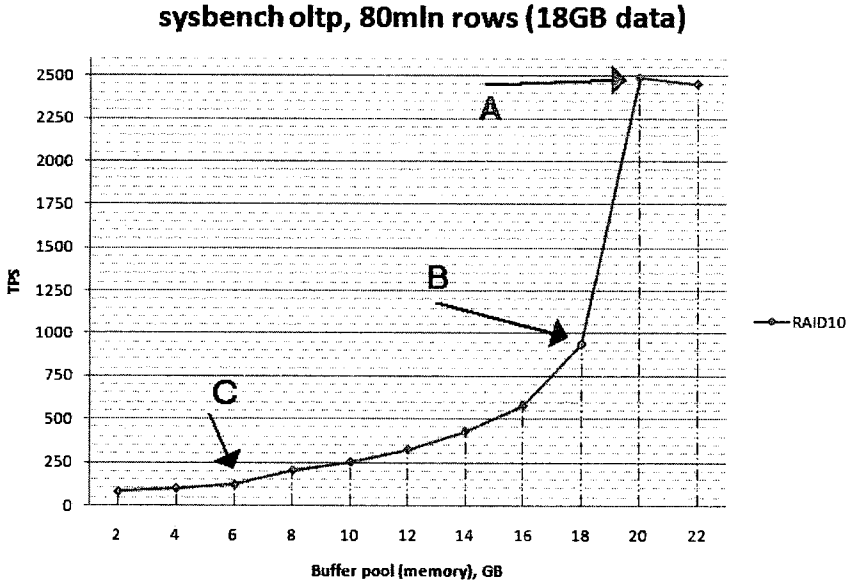


图9-1 不同内存容量下InnoDB存储引擎的性能表现

所以，应该在开发应用前预估“活跃“数据库的大小可能会是多少，并以此确定数据库服务器内存的大小。当然，要使用更多的内存，还必须使用64位的操作系统。

如何判断当前数据库的内存是否已经达到瓶颈了呢？可以通过查看当前服务器的状态，比较物理磁盘的读取和内存读取的比例来判断缓冲池的命中率，通常InnoDB存储引擎的缓冲池的命中率不应该小于99%，如：

```
mysql> show global status like 'innodb%read%'\G;
*************************** 1. row ***************************
Variable_name: Innodb_buffer_pool_read_ahead
        Value: 0
*************************** 2. row ***************************
Variable_name: Innodb_buffer_pool_read_ahead_evicted
        Value: 0
*************************** 3. row ***************************
Variable_name: Innodb_buffer_pool_read_requests
        Value: 167051313
*************************** 4. row ***************************
Variable_name: Innodb_buffer_pool_reads
```

```
        Value: 129236
*************************** 5. row ***************************
Variable_name: Innodb_data_pending_reads
        Value: 0
*************************** 6. row ***************************
Variable_name: Innodb_data_read
        Value: 2135642112
*************************** 7. row ***************************
Variable_name: Innodb_data_reads
        Value: 130309
*************************** 8. row ***************************
Variable_name: Innodb_pages_read
        Value: 130215
*************************** 9. row ***************************
Variable_name: Innodb_rows_read
        Value: 17651085
9 rows in set (0.00 sec)
```

上述参数的具体含义如下所示：

- Innodb_buffer_pool_reads：表示从物理磁盘读取页的次数。
- Innodb_buffer_pool_read_ahead：预读的次数。
- Innodb_buffer_pool_read_ahead_evicted：预读的页，但是没有被读取就从缓冲池中被替换的页的数量，一般用来判断预读的效率。
- Innodb_buffer_pool_read_requests：从缓冲池中读取页的次数。
- Innodb_data_read：总共读入的字节数。
- Innodb_data_reads：发起读取请求的次数，每次读取可能需要读取多个页。

以下公式可以计算各种对缓冲池的操作：

$$\text{缓冲池命中率}=\frac{\text{innode\_buffer\_pool\_read\_requests}}{(\text{innodb\_buffer\_pool\_read\_requests+Innodb\_buffer\_pool\_read\_ahead+Innodb\_buffer\_pool\_reads})}$$

$$\text{平均每次读取的字节数}=\frac{\text{innodb\_data\_read}}{\text{innodb\_data\_reads}}$$

从上面的例子看，缓冲池命中率= 167224026 / (167224026+ 129264+0) = 99.92%，可见当前内存的压力并不是很大。

即使缓冲池的大小已经大于数据库文件的大小，这也不意味着没有磁盘操作。数据库

的缓冲池只是一个用来存放热点的区域，后台的master线程还负责将脏页异步地写入磁盘，每次事务提交时还需要立即写入重做日志文件。

9.3 硬盘对数据库性能的影响

9.3.1 传统机械硬盘

当前大多数数据库使用的都是传统的机械硬盘。机械硬盘的技术目前已非常成熟，在服务器领域一般使用SAS或SATA接口的硬盘。服务器机械硬盘开始向小型化转型，目前已经有大量2.5寸的SAS机械硬盘。

机械硬盘有两个重要的指标：一个是寻道时间，另一个是转速。当前服务器机械硬盘的寻道时间已经能够达到3ms，转速为15 000rpm。传统机械硬盘最大的问题在于读写磁头，读写磁头的设计使得硬盘可以不再像磁带一样，只能进行顺序访问，而是可以随机访问。但是，硬盘的访问需要耗费长时间的磁头旋转和定位来查找，因此顺序访问的速度远远高于随机访问。数据库的很多设计也都是在尽量充分地利用顺序访问的特性。

可以将多块硬盘组成RAID来提高数据库的性能，也可以将数据文件分布在不同硬盘上来达到访问负载的均衡。

9.3.2 固态硬盘

固态硬盘，更准确地说是基于闪存的固态硬盘，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。因为闪存的低延迟性、低功耗以及防震性，所以闪存设备已在移动设备上得到了广泛的应用。企业级应用一般使用固态硬盘，通过并联多块闪存来进一步提高数据传输的吞吐量。传统的存储服务提供商EMC公司已经开始提供基于闪存的固态硬盘的TB级别存储解决方案。数据库厂商Oracle公司最近也开始提供绑定固态硬盘的Exadata服务器。

不同于传统的机械硬盘，闪存是一个完全的电子设备，没有传统机械硬盘的读写磁头。因此，固态硬盘不需要像传统机械硬盘一样，需要耗费大量时间的磁头旋转和定位来查找数据，所以固态硬盘可以提供一致的随机访问时间。固态硬盘这种对数据的快速读写和定

位特性是值得研究的。

另一方面，闪存中的数据是不可以更新的，只能通过扇区（sector）的覆盖重写，而在覆盖重写之前，需要执行非常耗时的擦除（erase）操作。擦除操作不能在所含数据的扇区上完成，而是需要擦除整个被称为擦除块的基础上，这个擦除块的尺寸大于扇区的大小，通常为128KB。此外，每个擦除块有擦写次数的限制。已经有一些算法来解决这个问题。但是对于数据库应用，需要认真考虑固态硬盘在写入方面存在的问题。

因为存在上述写入方面的问题，闪存提供的读写速度是非对称的。读取速度要远快于写入的速度，因此对于固态硬盘在数据库中的应用，应该好好利用其读取的性能，避免过多的写入操作。

图9-2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。通过进一步增加通道的数量，固态硬盘的性能可以线性地提高，如我们常见的Intel X-25M固态硬盘就是10通道的固态硬盘。

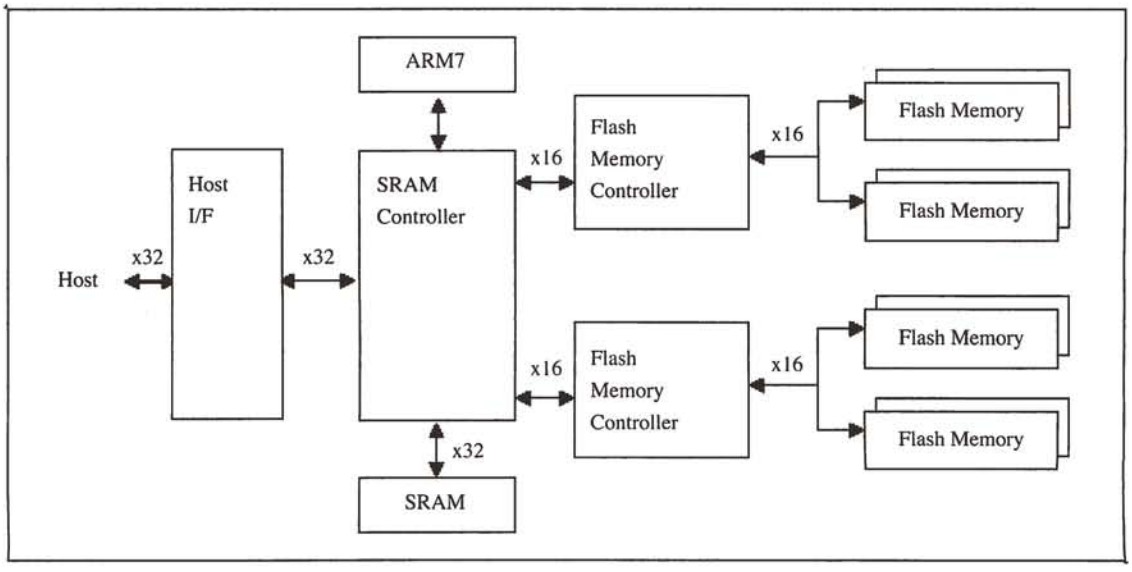


图9-2 双通道的固态硬盘架构

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成实际的物理地址，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0.1ms左右。图9-3显示了传统机械硬盘、内存、固态硬盘的随机

访问延时之间的比较：

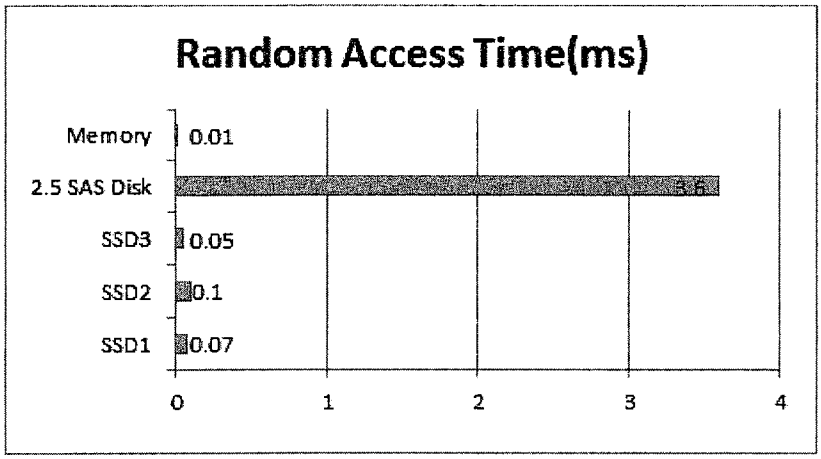


图9-3 固态硬盘和传统机械硬盘随机访问延时的比较

对于固态硬盘在InnoDB存储引擎中的优化，可以增加innodb_io_capacity变量的值，以达到充分利用固态硬盘带来的高IOPS的特性。同时也可以通过修改源代码来禁用InnoDB存储引擎的预读、邻接页的写入特性。此外，还可以使用我开发的InnoDB Secondary Buffer Pool开源patch[⊖]，该补丁可以充分利用固态硬盘的超高速随机读取性能，在内存缓冲池和传统存储层之间建立一层基于闪存固态硬盘的二级缓冲池，以此来扩充缓冲池的容量，提高数据库的性能。

9.4 合理地设置RAID

9.4.1 RAID类型

RAID（Redundant Array of Independent Disks，独立磁盘冗余数组）的基本思想，就是把多个相对便宜的硬盘组合起来，成为一个磁盘数组，使性能达到甚至超过一个价格昂贵、容量巨大的硬盘。由于将多个硬盘组合成为一个逻辑扇区，RAID看起来就像一个单独的硬盘或逻辑存储单元，因此操作系统只会把它当作一个硬盘。

⊖ 官网地址：http://code.google.com/p/david-mysql-tools/wiki/innodb_secondary_buffer_pool。

RAID的作用是：

- ❑ 增强数据集成度
- ❑ 增强容错功能
- ❑ 增加处理量或容量

根据不同磁盘的组合方式，常见的RAID组合方式可分为RAID 0、RAID 1、RAID 5、RAID 10和RAID 50等。

（1）RAID 0：将多个磁盘合并成一个大的磁盘，不会有冗余，并行I/O，速度最快。RAID 0亦称为带区集，它是将多个磁盘并列起来，使之成为一个大磁盘。在存放数据时，其将数据按磁盘的个数来进行分段，然后同时将这些数据写进这些盘中。所以，在所有的级别中，RAID 0的速度是最快的。但是RAID 0没有冗余功能，如果一个磁盘（物理）损坏，则所有的数据都会丢失。理论上，多磁盘的效能就等于[单一磁盘效能]x[磁盘数]，但实际上受限于总线I/O瓶颈及其他因素的影响，所以RAID效能会随边际递减。也就是说，假设一个磁盘的效能是50MB/秒，两个磁盘的RAID 0效能约96MB/秒，三个磁盘的RAID 0也许是130MB/秒，而不是150MB/秒。

（2）RAID 1：两组以上的N个磁盘相互作为镜像，在一些多线程操作系统中能有很好的读取速度，但写入速度略有降低。除非拥有相同数据的主磁盘与镜像同时损坏，否则只要一个磁盘正常，即可维持运作，因此可靠性最高。RAID 1就是镜像，其原理为，在主硬盘上存放数据的同时也在镜像硬盘上写一样的数据。当主硬盘（物理）损坏时，镜像硬盘则代替主硬盘的工作。因为有镜像硬盘做数据备份，所以RAID 1的数据安全性在所有的RAID级别中是最好的。但是，无论用多少磁盘，作为RAID 1，仅算一个磁盘的容量，所以RAID1是所有RAID中磁盘利用率最低的一个级别。

（3）RAID 5：是一种存储性能、数据安全和存储成本兼顾的存储解决方案。它使用的是Disk Striping（硬盘分区）技术。RAID 5至少需要三个硬盘，RAID 5不对存储的数据进行备份，而是把数据和相对应的奇偶校验信息存储到组成RAID 5的各个磁盘上，并且奇偶校验信息和相对应的数据分别存储于不同的磁盘上。当RAID 5的一个磁盘数据发生损坏后，利用剩下的数据和相应的奇偶校验信息去恢复被损坏的数据。RAID 5可以理解为是RAID 0和RAID 1的折中方案。RAID 5可以为系统提供数据安全保障，但保障程度要比镜像低，而磁盘空间利用率要比镜像高。RAID 5具有和RAID 0相近似的数据读取速度，只是多了

一个奇偶校验信息，所以写入数据的速度相当慢，若使用Write Back可以让性能改善不少。同时，由于多个数据对应一个奇偶校验信息，因此RAID 5的磁盘空间利用率要比RAID 1高，存储成本相对较低。

(4) RAID 10和RAID 01：RAID 10是先镜射，再分区数据。它将所有硬盘分为两组，视为RAID 0的最低组合，然后将这两组各自视为RAID 1运作。RAID 10有着不错的读取速度，而且拥有比RAID 0更高的数据保护性。RAID 01则与RAID 10的程序相反，是先分区，再将数据镜射到两组硬盘。它将所有的硬盘分为两组，变成RAID 1的最低组合，而将两组硬盘各自视为RAID 0运作。RAID 01比起RAID 10有着更快的读写速度，不过也多了一些会让整个硬盘组停止运转的概率：因为只要同一组的硬盘全部损毁，RAID 01就会停止运作，而RAID 10则可以在牺牲RAID 0的优势下正常运作。RAID 10巧妙地利用了RAID 0的速度以及RAID 1的安全（保护）两种特性，它的缺点是需要较多的硬盘，因为至少必须拥有4个以上的偶数硬盘才能使用。

(5) RAID 50：RAID 50也被称为镜像阵列条带，由至少6块硬盘组成，像RAID 0一样，数据被分区成条带，在同一时间内向多块磁盘写入；像RAID 5一样，RAID 50也是以数据的校验位来保证数据的安全，且校验条带均匀分布在各个磁盘上，其目的在于提高RAID 5的读写性能。

对于数据库应用来说，RAID 10是最好的选择，它同时兼顾了RAID 1和RAID 0的特性。但是，当一个磁盘失效时，性能可能会受到很大的影响，因为条带（strip）会成为瓶颈。我曾在生产环境下遇到过这样的情况：2台负载基本相同的数据库，一台正常的服务器磁盘IO负载为20%左右，而另一台服务器IO负载却高达90%。

9.4.2 RAID Write Back功能

RAID Write Back功能是指RAID控制器能够将写入的数据放入自身的缓存中，并把它们安排到后面再执行。这样做的好处是，不用等待物理磁盘实际写入的完成，因此写入变得更快了。对于数据库来说，这显得十分重要。例如，对重做日志的写入、在将sync_binlog设为1的情况下二进制日志的写入、脏页的刷新等，这些都可以使性能明显的提升。

但是，如果系统发生意外，Write Back功能可能会破坏数据库的数据，因为缓存可能

还在RAID卡中，这样磁盘并没有写入时故障就发生了。对此，大部分的硬件RAID卡都提供了电池备份单元（BBU, Battery Backup Unit），因此可以放心地开启Write Back的功能。不过我发现，每台服务器的出厂设置都是不同的，应该将你的RAID设置要求告知服务器提供商，开启一些你认为需要的参数。

如果没有启用Write Back功能，那么在RAID卡设置中显示的就是Write Through。Write Through没有缓冲写入，因此写入性能可能不是很好，但它的确是最安全的写入。

即使开启了Write Back功能，RAID卡也可能只是启用了Write Through，因为之前提到过，安全使用Write Back的前提是RAID卡有电池备份单元。为了确保电池的有效性，RAID卡会定期检查电池状态，并在电池电量不足时对其充电，在充电的这段时间内，会将Write Back功能切换为最为安全的Write Through。

你可以在没有电池备份单元的情况下强制启用Write Back功能，也可以在电池充电时强制使用Write Back功能。只是写入是不安全的，你应该非常确信这点，否则不应该在没有电池备份单元的情况下启用Write Back。

可以通过插入20W的记录来比较Write Back和Write Through的性能差异：

```
mysql> create table t ( a char(2))engine=innodb;
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter //
mysql>
mysql> create procedure p()
    -> begin
    -> declare v int;
    -> set v=0;
    -> while v<200000 do
    ->   insert into t values('aa');
    ->   set v=v+1;
    -> end while;
    -> end
    ->
    -> //
Query OK, 0 rows affected (0.12 sec)

mysql> delimiter ;
```

我们创建了一个往t表插入20万条记录的存储过程，并在Write Back和Write Through的

设置下分别进行测试，测试结果如表9-1所示：

表 9-1

RAID卡设置	时间
Write Back	43秒
Write Through	31分钟
Write Through with innodb_flush_log_at_trx_commit=0	68秒

我们的测试不是在一个事务中，而是直接用命令CALL P来运行的，因此数据库实际执行了20万次的事务。很明显可以看到，在Write Back模式下执行时间只需要43秒，而在Write Through模式下执行时间需要31分钟，大约是40多倍的差距。

当然，在Write Through模式下，通过将参数innodb_flush_log_at_trx_commit设置为0也可以提高执行存储过程P的性能，这时只需要68秒了。因为在此设置下，重做日志的写入不是发生在每次事务提交时，而是发生在后台master线程每秒钟自动刷新的时候，因此减少了物理磁盘的写入请求，所以执行速度也有明显的提高。

9.4.3 RAID配置工具

RAID卡的配置可以在服务器启动时进入一个类似于BIOS的配置界面，然后再对其进行各种设置。此外，很多厂商都开发了各种操作系统下的软件来进行RAID卡的配置，如果你使用的是LSI公司生产提供的RAID卡，则可以使用MegaCLI工具进行配置。

MegaCLI为多个操作系统提供了支持，在Windows下还提供了GUI界面的配置，相对来说比较简单。这里我们主要介绍命令行下MegaCLI的使用，Windows下可以使用MegaCLI.exe程序。

使用MegaCLI查看RAID卡的信息：

```
[root@xen-server ~]# /opt/MegaRAID/MegaCli/MegaCli64 -AdpAllInfo -a0

Adapter #0

==============================================================================
                    Versions
                ================
Product Name    : MegaRAID SAS 8708ELP
Serial No       : P012233608
```

```
FW Package Build: 9.0.1-0030

......
                HW Configuration
                ================
SAS Address     : 500605b000d1e180
BBU             : Present
Alarm           : Present
NVRAM           : Present
Serial Debugger : Present
Memory          : Present
Flash           : Present
Memory Size     : 256MB
TPM             : Absent

                Default Settings
                ================
Phy Polarity                        : 0
Phy PolaritySplit                   : 240
Background Rate                     : 30
Stripe Size                         : 64kB
Flush Time                          : 4 seconds
Write Policy                        : WB
Read Policy                         : None
Cache When BBU Bad                  : Disabled
Cached IO                           : No
SMART Mode                          : Mode 6
Alarm Disable                       : Yes
Coercion Mode                       : 1GB
ZCR Config                          : Unknown
Dirty LED Shows Drive Activity      : No
BIOS Continue on Error              : No
Spin Down Mode                      : None
Allowed Device Type                 : SAS/SATA Mix
Allow Mix in Enclosure              : Yes
Allow HDD SAS/SATA Mix in VD        : Yes
Allow SSD SAS/SATA Mix in VD        : No
Allow HDD/SSD Mix in VD             : No
Allow SATA in Cluster               : No
Max Chained Enclosures              : 3
Disable Ctrl-R                      : Yes
Enable Web BIOS                     : Yes
Direct PD Mapping                   : No
BIOS Enumerate VDs                  : Yes
```

```
Restore Hot Spare on Insertion   : No
Expose Enclosure Devices         : Yes
Maintain PD Fail History         : Yes
Disable Puncturing               : No
Zero Based Enclosure Enumeration : No
PreBoot CLI Enabled              : Yes
LED Show Drive Activity          : No
Cluster Disable                  : Yes
SAS Disable                      : No
Auto Detect BackPlane Enable     : SGPIO/i2c SEP
Use FDE Only                     : No
Enable Led Header                : No
Delay during POST                : 0
```

出于排版的原因，这里我只列出了输出的一小部分。用过上述命令，可以看到RAID卡的一些硬件设置，如这块RAID卡的型号是MegaRAID SAS 8708ELP，缓存大小是256MB，一些默认的配置，如默认启用的Write Policy为WB（Write Back）等。

MegaCLI还可以用来查看当前物理磁盘的信息，如：

```
[root@xen-server ~]# /opt/MegaRAID/MegaCli/MegaCli64 -PDList -aALL

Adapter #0

Enclosure Device ID: 252
Slot Number: 0
Device Id: 8
Sequence Number: 2
Media Error Count: 0
Other Error Count: 0
Predictive Failure Count: 0
Last Predictive Failure Event Seq Number: 0
PD Type: SAS
Raw Size: 279.396 GB [0x22ecb25c Sectors]
Non Coerced Size: 278.896 GB [0x22dcb25c Sectors]
Coerced Size: 278.464 GB [0x22cee000 Sectors]
Firmware state: Online
SAS Address(0): 0x5000c5000f363b55
SAS Address(1): 0x0
Connected Port Number: 0(path0)
Inquiry Data: SEAGATE ST3300655SS      00023LM5MGZZ
FDE Capable: Not Capable
FDE Enable: Disable
```

```
Secured: Unsecured
Locked: Unlocked
Foreign State: None
Device Speed: Unknown
Link Speed: Unknown
Media Type: Hard Disk Device
......
```

可以看到当前使用的磁盘型号是SEAGATE ST3300655SS。你可以从这个型号继续找到这个硬盘的具体信息，如在希捷官网http://discountechnology.com/Seagate-ST3300655SS-SAS-Hard-Drive上可以知道，这块硬盘大小是3.5寸的，转速为15 000，硬盘的Cache为16MB，随机读取的寻道时间是3.5毫秒，随机写入的寻道时间是4.0毫秒等。

可以通过下面的命令来查看是否开启了Write Back功能：

```
[root@xen-server ~]# /opt/MegaRAID/MegaCli/MegaCli64  -LDGetProp -Cache -LALL -aALL

Adapter 0-VD 0(target id: 0): Cache Policy:WriteBack, ReadAheadNone, Direct, No
Write Cache if bad BBU
Adapter 0-VD 1(target id: 1): Cache Policy:WriteBack, ReadAheadNone, Direct, No
Write Cache if bad BBU

Exit Code: 0x00
```

可以看到当前开启了Write Back功能，并且当BBU有问题时或者在充电时禁用Write Back功能。这里还显示了不需要启用RAID卡的预读功能，写入为直接写入方式。

通过下面的命令可以对当前的写入策略进行调整：

```
#/opt/MegaRAID/MegaCli/MegaCli64 -LDSetProp WB -LALL -aALL
#/opt/MegaRAID/MegaCli/MegaCli64 -LDSetProp WT -LALL -aALL
```

注意： 当写入策略从Write Back切换为Write Through时，该更改立即生效，但是从Write Through切换为Write Back时，必须重启服务器才能使其生效。

9.5 操作系统的选择也很重要

Linux是MySQL数据库服务器中最常见的操作系统。与其他操作系统不同的是，Linux有着众多的发行版本，可能每个人的偏好都不相同。但是，在将Linux操作系统作为数据

库服务器时，需要考虑更多的是操作系统的稳定性，而不是新特性。

除了Linux操作系统外，FreeBSD也是另一个常见的操作系统。之前版本的FreeBSD对MySQL数据库支持得不是很好，需要选择单独的线程库进行手动编译，但是新版本的FreeBSD对MySQL数据库的支持已经好了很多，直接下载二进制安装包即可。

Solaris之前是基于SPARC硬件的操作系统，现在已经移植到了X86平台上。Solaris是高性能、高可靠性的操作系统，同时其提供的ZFS文件系统非常适合MySQL的数据库应用。如果需要，你可以尝试它的开源版本Open Solaris。

Windows在MySQL的数据库应用中也非常常见。也有的公司喜欢在开发环境下使用Windows版本的MySQL，到正式生产环境下使用Linux。这本身没有什么问题，但问题通常发生于大小写敏感方面。Windows下表名不区分大小写，而Linux操作系统却是大小写敏感的，这点在开发阶段需要特别注意。

4GB内存在当前已经非常普遍了，即使是桌面用户也开始使用8GB的内存。为了可以更好地使用大于4GB的内存容量，必须使用64位的操作系统，上述介绍的这些操作系统都提供了64位的版本。此外，使用64位的操作系统还必须使用64位的软件。这听上去是句废话，但是我多次看到32位的MySQL数据库安装在64位的系统上，而这样会导致不能充分发挥64位操作系统的寻址能力。

9.6 不同的文件系统对数据库性能的影响

每个操作系统都默认支持一种文件系统并推荐用户使用，如Windows默认支持NTFS，Solaris默认支持ZFS。而对于Linux这样的操作系统，不同发行版本默认支持的文件系统又各不相同，有的默认支持EXT3，有的是ReiserFS，有的是EXT4，有的是XFS。

虽然有着很多不同特性的文件系统，但是我在实际使用过程中从未感觉到文件系统的性能差异有多大。网上有多个关于XFS文件系统的“神话”，认为其是多么适合数据库应用，性能较之EXT3有极大的提升。但是我在实际测试和使用后发现，它的性能和EXT3在整体上没有大的差距。因此，DBA应该把更多的注意力放到数据库上面，而不是纠结于文件系统。

文件系统可提供的功能也许是DBA需要关注的，例如ZFS文件系统本身就可以支持快

照，因此就不需要LVM这样的逻辑卷管理工具。此外，可能还需要知道mount的参数，这些参数在每个文件系统中又可能有所不同。

9.7 选择合适的基准测试工具

基准测试工具可以用来对数据库或者操作系统调优后的性能进行对比。MySQL数据库本身提供了一些比较优秀的工具，这里将介绍另外两款更优秀、更常用的工具：sysbench和mysql-tpcc。

9.7.1 sysbench

sysbench是一个模块化的、跨平台的、多线程基准测试工具，主要用于测试各种不同系统参数下的数据库负载情况。它主要包括以下几种测试方式：

- CPU性能。
- 磁盘IO性能。
- 调度程序性能。
- 内存分配及传输速度。
- POSIX线程性能。
- 数据库OLTP基准测试。

sysbench的数据库OLTP测试支持MySQL、PostgreSQL和Oracle。目前sysbench主要用于Linux操作系统，开源社区已经将sysbench移植到Windows，并支持对Microsoft SQL Server数据库的测试。

sysbench的官网地址是：http://sysbench.sourceforge.net，可以从上述地址下载最新版本的sysbench工具，然后编译和安装。此外，有些Linux操作系统发行版本（如RED HAT），可能本身已经提供了sysbench的安装包，直接安装即可。

sysbench可以通过不同的参数设置来进行不同项目的测试，使用方法如下所示：

```
[root@xen-server ~]# sysbench
Missing required command argument.
Usage:
  sysbench [general-options]... --test=<test-name> [test-options]... command
```

```
General options:
  --num-threads=N             number of threads to use [1]
  --max-requests=N            limit for total number of requests [10000]
  --max-time=N                limit for total execution time in seconds [0]
  --thread-stack-size=SIZE    size of stack per thread [32K]
  --init-rng=[on|off]         initialize random number generator [off]
  --test=STRING               test to run
  --debug=[on|off]            print more debugging info [off]
  --validate=[on|off]         perform validation checks where possible [off]
  --help=[on|off]             print help and exit
  --version=[on|off]          print version and exit

Compiled-in tests:
  fileio - File I/O test
  cpu - CPU performance test
  memory - Memory functions speed test
  threads - Threads subsystem performance test
  mutex - Mutex performance test
  oltp - OLTP test

Commands: prepare run cleanup help version

See 'sysbench --test=<name> help' for a list of options for each test.
```

对于InnoDB存储引擎的数据库应用来说，我们可能更关心的是磁盘和OLTP的性能，因此主要测试fileio和oltp这两个项目。对于磁盘的测试，sysbench提供了以下的测试选项：

```
[root@xen-server ~]# sysbench --test=fileio help
sysbench 0.4.10:  multi-threaded system evaluation benchmark

fileio options:
  --file-num=N                  number of files to create [128]
  --file-block-size=N           block size to use in all IO operations [16384]
  --file-total-size=SIZE        total size of files to create [2G]
  --file-test-mode=STRING       test mode {seqwr, seqrewr, seqrd, rndrd, rndwr, rndrw}
  --file-io-mode=STRING         file operations mode {sync,async,fastmmap,slowmmap} [sync]
  --file-extra-flags=STRING     additional flags to use on opening files
{sync,dsync,direct} []
  --file-fsync-freq=N           do fsync() after this number of requests (0 -
don't use fsync()) [100]
  --file-fsync-all=[on|off]     do fsync() after each write operation [off]
  --file-fsync-end=[on|off]     do fsync() at the end of test [on]
  --file-fsync-mode=STRING      which method to use for synchronization {fsync,
fdatasync} [fsync]
```

```
  --file-merged-requests=N      merge at most this number of IO requests if
possible (0 - don't merge) [0]
  --file-rw-ratio=N             reads/writes ratio for combined test [1.5]
```

各个参数的含义如下所示：

- --file-num：生成测试文件的数量，默认为128。
- --file-block-size：测试期间文件块的大小，如果你想磁盘针对InnoDB存储引擎进行测试，可以将其设置为16 384，即InnoDB存储引擎页的大小。默认为16 384。
- --file-total-size：每个文件的大小，默认为2GB。
- --file-test-mode：文件测试模式，包含seqwr（顺序写）、seqrewr（顺序读写）、seqrd（顺序读）、rndrd（随机读）、rndwr（随机写）和rndrw（随机读写）。
- --file-io-mode：文件操作的模式，同步还是异步，或者选择MMAP（map映射）模式。默认为同步。
- -- file-extra-flags：打开文件时的选项，这是与API相关的参数。
- --file-fsync-freq：执行fsync函数的频率。fsync主要是同步磁盘文件，因为可能有系统和磁盘缓冲的关系。
- --file-fsync-all：每执行完一次写操作，就执行一次fsync。默认为off。
- --file-fsync-end：在测试结束时执行fsync。默认为on。
- --file-fsync-mode：文件同步函数的选择，同样是和API相关的参数，由于多个操作系统对于fdatasync支持的不同，因此不建议使用fdatasync。默认为fsync。
- --file-rw-ratio：测试时的读写比例，默认时读写2:1。

sysbench的fileio测试需要经过prepare、run和clean三个阶段。prepare是准备阶段，生产我们需要的测试文件，run是实际测试阶段，cleanup是清理测试产生的文件。如我们进行16个文件、总大小2GB的fileio测试：

```
[root@xen-server ssd]# sysbench --test=fileio --file-num=16 --file-total-size=2G prepare
sysbench 0.4.10:  multi-threaded system evaluation benchmark

16 files, 131072Kb each, 2048Mb total
Creating files for the test...
```

接着在相应的目录下就会产生16个文件了，因为总大小是2GB，所以每个文件的大小

应该是128MB：

```
[root@xen-server ssd]# ls -lh
total 2G
-rw------- 1 root  root  128M Aug 12 10:42 test_file.0
-rw------- 1 root  root  128M Aug 12 10:42 test_file.1
-rw------- 1 root  root  128M Aug 12 10:42 test_file.10
-rw------- 1 root  root  128M Aug 12 10:42 test_file.11
-rw------- 1 root  root  128M Aug 12 10:42 test_file.12
-rw------- 1 root  root  128M Aug 12 10:42 test_file.13
-rw------- 1 root  root  128M Aug 12 10:42 test_file.14
-rw------- 1 root  root  128M Aug 12 10:42 test_file.15
-rw------- 1 root  root  128M Aug 12 10:42 test_file.2
-rw------- 1 root  root  128M Aug 12 10:42 test_file.3
-rw------- 1 root  root  128M Aug 12 10:42 test_file.4
-rw------- 1 root  root  128M Aug 12 10:42 test_file.5
-rw------- 1 root  root  128M Aug 12 10:42 test_file.6
-rw------- 1 root  root  128M Aug 12 10:42 test_file.7
-rw------- 1 root  root  128M Aug 12 10:42 test_file.8
-rw------- 1 root  root  128M Aug 12 10:42 test_file.9
```

接着就可以基于这些文件进行测试了，下面是在16个线程下的随机读取性能：

```
[root@xen-server ssd]# sysbench --test=fileio --file-total-size=2G --file-test-
mode=rndrd--max-time=180--max-requests=100000000 --num-threads=16 --init-rng=on --
file-num=16 --file-extra-flags=direct --file-fsync-freq=0 --file-block-size=16384 run
```

上述测试的最大随机读取请求是100 000 000次，如果在180秒内不能完成，测试即结束。测试结束后可以看到如下的测试结果：

```
[root@xen-server ssd]# sysbench --test=fileio --file-total-size=2G --file-test-
mode=rndrd--max-time=180 --max-requests=100000000 --num-threads=16 --init-rng=on --
file-num=16 --file-extra-flags=direct --file-fsync-freq=0 --file-block-size=16384 run
sysbench 0.4.10:  multi-threaded system evaluation benchmark

Running the test with following options:
Number of threads: 16
Initializing random number generator from timer.


Extra file open flags: 16384
16 files, 128Mb each
2Gb total file size
Block size 16Kb
Number of random requests for random IO: 100000000
```

```
Read/Write ratio for combined random IO test: 1.50
Calling fsync() at the end of test, Enabled.
Using synchronous I/O mode
Doing random read test
Threads started!
Time limit exceeded, exiting...
(last message repeated 15 times)
Done.

Operations performed:  619908 Read, 0 Write, 0 Other = 619908 Total
Read 9.459Gb  Written 0b  Total transferred 9.459Gb  (53.81Mb/sec)
 3443.85 Requests/sec executed

Test execution summary:
    total time:                          180.0044s
    total number of events:              619908
    total time taken by event execution: 2878.0750
    per-request statistics:
         min:                                  0.42ms
         avg:                                  4.64ms
         max:                                 27.30ms
         approx.  95 percentile:               8.13ms

Threads fairness:
    events (avg/stddev):           38744.2500/102.69
    execution time (avg/stddev):   179.8797/0.00
```

可以看到随机读取的性能为53.81MB/sec，随机读的IOPS为3443.85。测试的硬盘是固态硬盘，因此随机读取的性能较为强劲。此外还可以看到每次请求的一些具体数据，如最大值、最小值、平均值等。

测试结束后，记得要执行clean，以确保测试所产生的文件都已删除：

```
[root@xen-server ssd]# sysbench --test=fileio --file-num=16 --file-total-size=2G cleanup
sysbench 0.4.10:  multi-threaded system evaluation benchmark

Removing test files...
```

可能你需要测试随机读、随机写、随机读写、顺序写、顺序读等所有这些模式，并且还可能需要测试不同的线程和不同文件块下磁盘的性能表现，这时你可能需要类似如下的脚本来帮你自动完成这些测试：

```
#!/bin/sh
```

```
set -u
set -x
set -e
for size in 8G 64G; do
for mode in seqrd seqrw rndrd rndwr rndrw; do
for blksize in 4096 16384 ; do
sysbench --test=fileio --file-num=64 --file-total-size=$size prepare
for threads in 1 4 8 16 32; do
echo "====== testing $blksize in $threads threads"
echo PARAMS $size $mode $threads $blksize> sysbench-size-$size-mode-$mode-
threads-$threads-blksz-$blksize
for i in 1 2 3 ; do
sysbench --test=fileio --file-total-size=$size --file-test-mode=$mode\
--max-time=180 --max-requests=100000000 --num-threads=$threads --init-rng=on \
--file-num=64 --file-extra-flags=direct --file-fsync-freq=0 --file-block-
size=$blksize run \
| tee -a sysbench-size-$size-mode-$mode-threads-$threads-blksz-$blksize 2>&1
done
done
sysbench --test=fileio --file-total-size=$size cleanup
done
done
done
```

对于mysql的OLTP测试，和fileio一样，同样需要经历prepare、run和cleanup的阶段。prepare阶段会根据选项产生一张指定行数的表，默认表在sbtest架构下，表名为sbtest（sysbench默认生成表的存储引擎为InnoDB），如创建一张有8000万条记录的表：

```
[root@xen-server ~]#  sysbench --test=oltp --oltp-table-size= 80000000 --db-
driver=mysql  --mysql-socket=/tmp/mysql.sock --mysql-user=root prepare
sysbench 0.4.10:  multi-threaded system evaluation benchmark

Creating table 'sbtest'...
Creating 80000000 records in table 'sbtest'...
```

接着就可以根据产生的表进行oltp的测试：

```
sysbench --test=oltp --oltp-table-size=80000000 --oltp-read-only=off --init-
rng=on --num-threads=16 --max-requests=0 --oltp-dist-type=uniform --max-time=3600
--mysql-user=root   --mysql-socket=/tmp/mysql.sock --db-driver=mysql   run > res
```

我们将测试结果放入了文件res中，查看res可得到类似如下的结果：

```
sysbench 0.4.10:  multi-threaded system evaluation benchmark
```

```
WARNING: Preparing of "BEGIN" is unsupported, using emulation
(last message repeated 15 times)
Running the test with following options:
Number of threads: 16
Initializing random number generator from timer.


Doing OLTP test.
Running mixed OLTP test
Using Uniform distribution
Using "BEGIN" for starting transactions
Using auto_inc on the id column
Threads started!
Time limit exceeded, exiting...
(last message repeated 15 times)
Done.

OLTP test statistics:
    queries performed:
        read:                            6043324
        write:                           2158330
        other:                           863332
        total:                           9064986
    transactions:                        431666 (119.90 per sec.)
    deadlocks:                           0 (0.00 per sec.)
    read/write requests:                 8201654 (2278.07 per sec.)
    other operations:                    863332 (239.80 per sec.)

Test execution summary:
    total time:                          3600.2672s
    total number of events:              431666
    total time taken by event execution: 57598.5965
    per-request statistics:
         min:                                  6.84ms
         avg:                                133.43ms
         max:                               7155.61ms
         approx.  95 percentile:             325.84ms

Threads fairness:
    events (avg/stddev):           26979.1250/64.14
    execution time (avg/stddev):   3599.9123/0.06.
```

结果中罗列出了测试时很多操作的详细信息，transactions代表了测试结果的评判标准，

即TPS，上述测试的结果是119.9tps。你可以对数据库进行调优后再运行sysbench的oltp测试，看看tps是否有所提高。注意，sysbench的测试只是基准测试，并不代表实际企业环境下的性能指标。

9.7.2 mysql-tpcc

TPC（Transaction Processing Performance Council，事务处理性能协会）是一个评价大型数据库系统软硬件性能的非盈利组织。TPC-C是TPC协会制定的，用来测试典型的复杂OLTP（在线事务处理）系统的性能。目前，在学术界和业界，普遍采用TPC-C来评价OLTP应用的性能。

TPC-C用3NF（第三范式）虚拟实现了一家仓库销售供应商公司，拥有一批分布在不同地方的仓库和地区分公司。当公司业务扩大时，将建立新的仓库和地区分公司。通常每个仓库供货覆盖10家地区分公司，每个地区分公司服务3000名客户。该公司共有100 000种商品，分别储存在各个仓库中。该系统包含了库存管理、销售、分发产品、付款、订单查询等一系列操作，一共包含了9个基本关系，基本关系图如图9-4所示。

TPC-C的性能度量单位是tpmC，tpm是transaction per minute的缩写，C代表TPC的C基准测试。该值越大，代表事务处理的性能越高。

tpcc-mysql是开源的TPC-C测试工具，该测试工具完全遵守TPC-C的标准。其官方网站为：https://code.launchpad.net/~percona-dev/perconatools/tpcc-mysql。之前tpcc-mysql主要工作在Linux操作系统上，我已经将其移植到了Windows平台，可以在http://code.google.com/p/david-mysql-tools/downloads/list下载到windows版本的tpcc-mysql。

tpcc-mysql由以下两个工具组成：

- tpcc_load：根据仓库数量，生成9张表中的数据。
- tpcc_start：根据不同选项进行tpcc测试。

tpcc_load命令的使用方法如下所示：

```
[root@xen-server ~]# tpcc_load
************************************
*** ###easy### TPC-C Data Loader  ***
************************************
```

```
usage: tpcc_load [server] [DB] [user] [pass] [warehouse]
     OR
       tpcc_load [server] [DB] [user] [pass] [warehouse] [part] [min_wh] [max_wh]

          * [part]: 1=ITEMS 2=WAREHOUSE 3=CUSTOMER 4=ORDERS
```

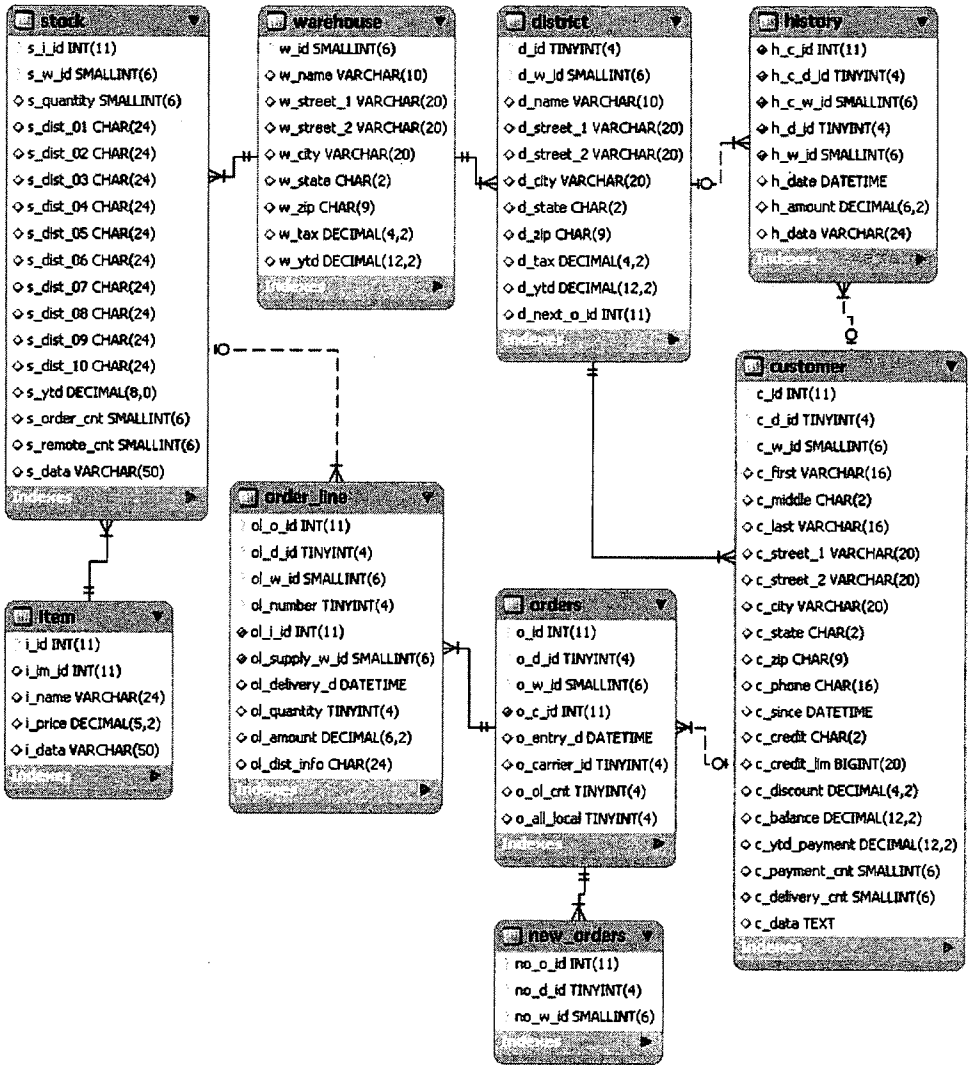


图9-4 TPC-C基本关系图

上述各参数解析如下：

- server：导入的MySQL服务器IP。
- DB：导入的数据库。

- user：mysql的用户名。
- pass：mysql的密码。
- warehouse：要生产的仓库数量。

如果用tpcc_load工具创建100个仓库的数据库tpcc，可以这样：

```
[root@xen-server tpcc-mysql]# mysql tpcc< create_table.sql
[root@xen-server tpcc-mysql]# mysql tpcc < add_fkey_idx.sql
[root@xen-server tpcc-mysql]# tpcc_load 127.0.0.1 tpcc2 root xxxxxx 100
************************************
*** ###easy### TPC-C Data Loader  ***
************************************
<Parameters>
     [server]: 127.0.0.1
     [DBname]: tpcc2
       [user]: root
       [pass]:
  [warehouse]: 100
TPCC Data Load Started...
Loading Item
.................................................. 5000
.................................................. 10000
.................................................. 15000
......(略)
...DATA LOADING COMPLETED SUCCESSFULLY.
```

tpcc_start命令的使用方法如下所示：

```
[root@xen-server ~]# tpcc_start
************************************
*** ###easy### TPC-C Load Generator ***
************************************

 usage: tpcc_start [server] [DB] [user] [pass] [warehouse] [connection] [rampup] [measure]
```

相关参数的作用如下：

- connection：测试时的线程数量。
- rampup：热身时间，单位为秒，这段时间的操作不计入统计信息。
- measure：测试时间，单位为秒。

如我们使用tpcc_start进行16个线程的测试，热身时间为10分钟、测试时间为20分钟，如下代码所示。

```
[root@xen-server ~]# tpcc_start 127.0.0.1 tpcc root xxxxxx 100 16 600 1200
****************************************
*** ###easy### TPC-C Load Generator ***
****************************************
<Parameters>
     [server]: 127.0.0.1
     [DBname]: tpcc
       [user]: root
       [pass]: xxxxxx
  [warehouse]: 100
 [connection]: 16
     [rampup]: 600 (sec.)
    [measure]: 1200 (sec.)
......
```

在测试的时候，你也许会在终端上看到如下输出：

```
RAMP-UP TIME.(1 sec.)

MEASURING START.

  10, 624(0):0.4, 624(0):0.2, 62(0):0.2, 63(0):0.6, 62(0):0.8
  20, 990(0):0.2, 988(0):0.2, 98(0):0.2, 99(0):0.4, 98(0):0.6
  30, 1435(0):0.2, 1436(0):0.2, 144(0):0.2, 143(0):0.2, 144(0):0.4
  40, 1736(0):0.2, 1739(0):0.2, 174(0):0.2, 174(0):0.2, 174(0):0.4
  50, 2041(0):0.2, 2044(0):0.2, 204(0):0.2, 204(0):0.2, 207(0):0.2
  60, 2195(0):0.2, 2193(0):0.2, 220(0):0.2, 221(0):0.2, 218(0):0.2
  70, 2332(0):0.2, 2335(0):0.2, 233(0):0.2, 232(0):0.2, 234(0):0.2
  80, 2408(0):0.2, 2401(0):0.2, 241(0):0.2, 239(0):0.2, 241(0):0.2
  90, 2473(0):0.2, 2476(0):0.2, 247(0):0.2, 250(0):0.2, 248(0):0.2
 100, 2350(0):0.2, 2347(0):0.2, 235(0):0.2, 233(0):0.2, 235(0):0.2
......
```

这些信息是每10秒tpcc测试的结果数据，tpcc测试一共测试5个模块，分别是New Order、Payment、Order-Status、Delivery、Stock-Level。第一个值即为New Order，这也是TPCC测试结果的一个重要考量标准New Order Per 10 Second（每十秒订单处理能力），可以将测试时所有的数据组成一张折线图或散点图，观察InnoDB存储引擎每10秒的性能表现，如图9-5所示。

而tpcc_load最后结束时产生的tpmC，也是通过New Order Per 10 Second来进行的：首先求出New Order Per 10 Second的平均值，然后乘以6，得到的就是最终的tpmC。

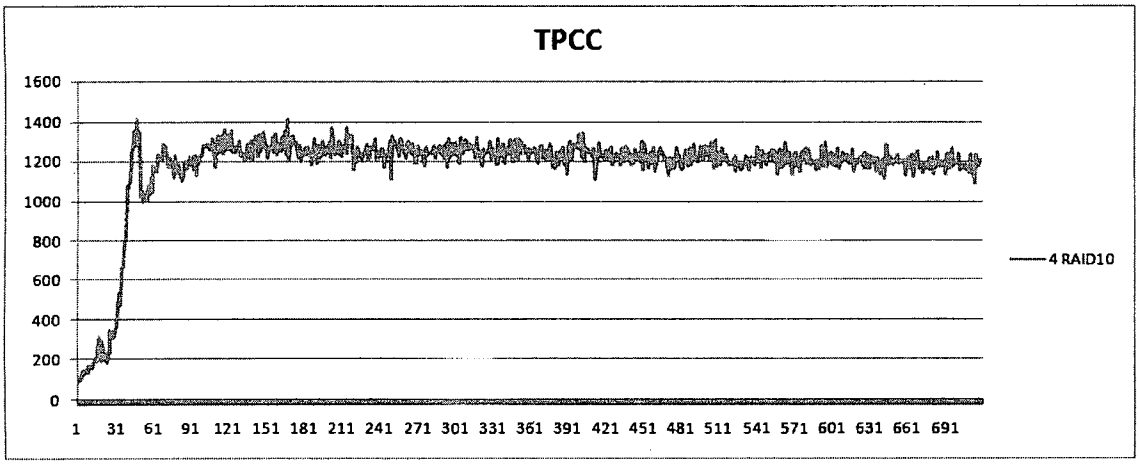


图9-5 New Order Per 10 Second

```
......
<Constraint Check> (all must be [OK])
 [transaction percentage]
        Payment: 43.48% (>=43.0%)[OK]
   Order-Status: 4.35% (>= 4.0%) [OK]
       Delivery: 4.35% (>= 4.0%) [OK]
    Stock-Level: 4.35% (>= 4.0%) [OK]
 [response time (at least 90% passed)]
      New-Order: 99.72%  [OK]
        Payment: 99.95%  [OK]
   Order-Status: 99.93%  [OK]
       Delivery: 100.00% [OK]
    Stock-Level: 100.00% [OK]

<TpmC>
                7949.942 TpmC
```

9.8 小结

在这一章中，我们根据InnoDB存储引擎的应用特点对CPU、内存、硬盘、固态硬盘、RAID卡做了详细的介绍。只有通过理解InnoDB存储引擎的应用场合和范围，才能更好地对其进行调优。最后，介绍了两个在Linux操作系统平台下常用的基准测试工具sysbench和tpcc-mysql。借助这两个工具，可以更有效地得知当前系统的负载承受能力，以及对数据库的调优结果进行分析。

第10章 InnoDB存储引擎源代码的编译和调试

InnoDB存储引擎是开源的，这意味着你可以获得其源代码，并查看内部的具体实现。任何时候，WHY都比WHAT重要。通过研究源代码，可以更好地理解数据库是如何工作的，从而知道如何使数据库更好地为你工作。如果你有一定的编程能力，则完全可以对InnoDB存储引擎进行扩展，开发出新的功能模块来更好地支持你的数据库应用。

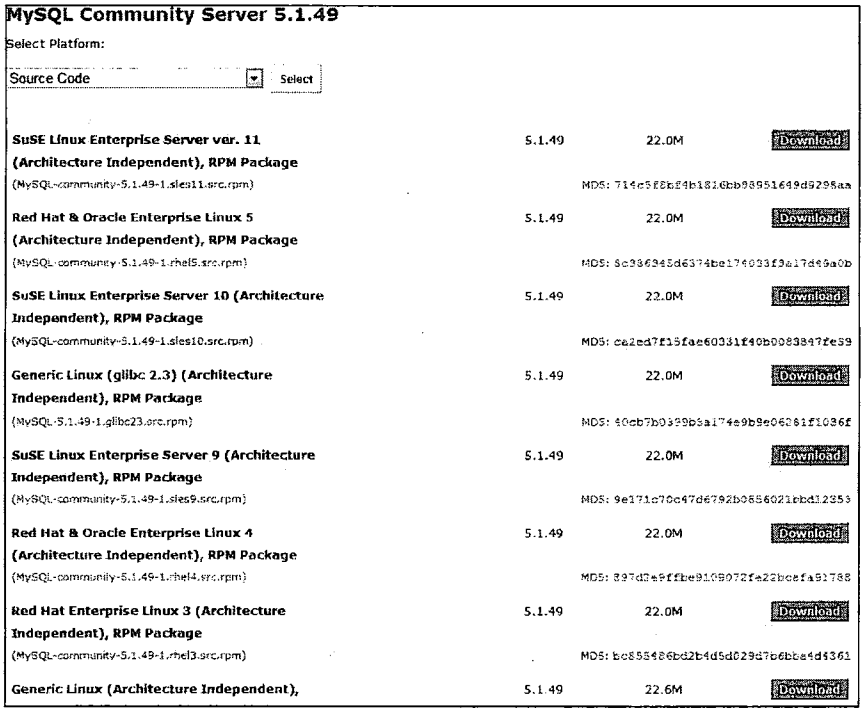
10.1 获取InnoDB存储引擎源代码

InnoDB存储引擎的源代码被包含在MySQL数据库的源代码中，在MySQL的官方网站[⊖]上下载MySQL数据库的源代码即可，如图10-1所示。

可以看到，这里有不同操作系统下的源代码可供下载，一般只需下载Generic Linux的版本即可。通过MySQL官网首页的Download链接，可以迅速地找到GA版本的下载。但是，如果想要下载目前正在开发的MySQL版本，如MySQL 5.5.5（现在是milestone的版本，离GA版本还有很长的开发时间），可能在官网找了很久都找不到链接。这时，只要把下载的链接从www换到dev即可：如http://dev.mysql.com/downloads/mysql，在这里可以找到开发中的MySQL版本的源代码了，如图10-2所示。

单击“Download”下载标签后可以进入下载页面。当然，如果你有mysql.com账户，可以进行登录。MySQL官方提供了大量的镜像用来分流下载，你可以根据所在的位置选择下载速度最快的地址，中国用户一般可以在“Asia”这里的镜像下载，如图10-3所示。

⊖ 链接为：http://www.mysql.com/downloads/mysql/。



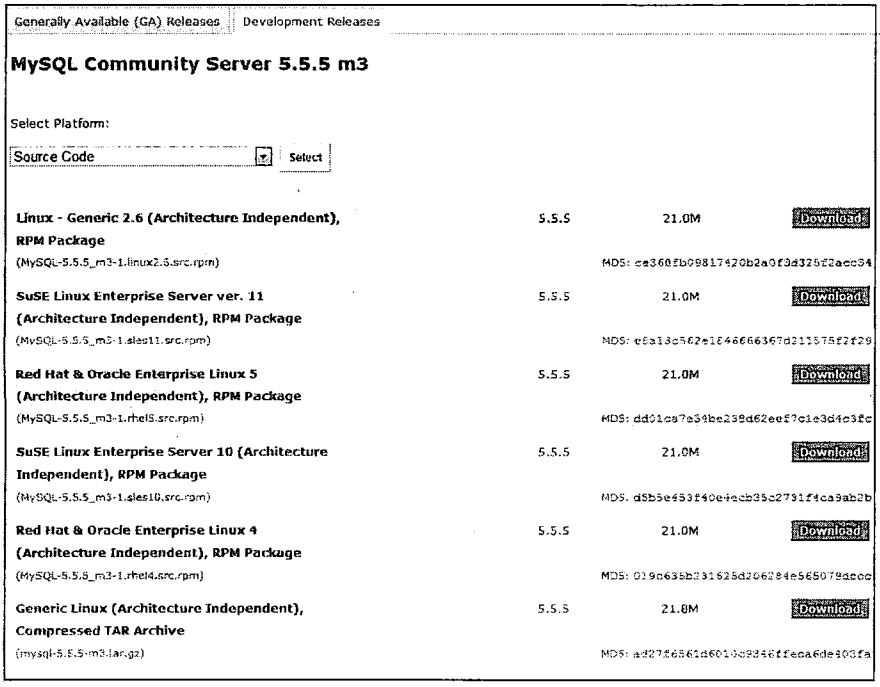
MySQL Community Server 5.1.49

Select Platform:

Source Code | Select

SuSE Linux Enterprise Server ver. 11 (Architecture Independent), RPM Package	5.1.49	22.0M	Download
(MySQL-community-5.1.49-1.sles11.src.rpm)		MD5: 714c5f8bf4b1816bb98951649d9298aa	
Red Hat & Oracle Enterprise Linux 5 (Architecture Independent), RPM Package	5.1.49	22.0M	Download
(MySQL-community-5.1.49-1.rhel5.src.rpm)		MD5: 8c386345d6374be174033f3a17d69a0b	
SuSE Linux Enterprise Server 10 (Architecture Independent), RPM Package	5.1.49	22.0M	Download
(MySQL-community-5.1.49-1.sles10.src.rpm)		MD5: ca2ed7f15fae60331f40b0083847fe59	
Generic Linux (glibc 2.3) (Architecture Independent), RPM Package	5.1.49	22.0M	Download
(MySQL-5.1.49-1.glibc23.src.rpm)		MD5: 40cb7b0399b5a174e9b9e06281f1036f	
SuSE Linux Enterprise Server 9 (Architecture Independent), RPM Package	5.1.49	22.0M	Download
(MySQL-community-5.1.49-1.sles9.src.rpm)		MD5: 9e171c70c47d6792b0856021bbd12253	
Red Hat & Oracle Enterprise Linux 4 (Architecture Independent), RPM Package	5.1.49	22.0M	Download
(MySQL-community-5.1.49-1.rhel4.src.rpm)		MD5: 897d9e9ffbe9109072fa22bcefa81788	
Red Hat Enterprise Linux 3 (Architecture Independent), RPM Package	5.1.49	22.0M	Download
(MySQL-community-5.1.49-1.rhel3.src.rpm)		MD5: bc855486bd2b4d5d029d7b6bba4d4361	
Generic Linux (Architecture Independent),	5.1.49	22.6M	Download

图10-1 MySQL源代码下载



Generally Available (GA) Releases | Development Releases

MySQL Community Server 5.5.5 m3

Select Platform:

Source Code | Select

Linux - Generic 2.6 (Architecture Independent), RPM Package	5.5.5	21.0M	Download
(MySQL-5.5.5_m3-1.linux2.6.src.rpm)		MD5: ce360fb09817420b2a0f3d325f2acc54	
SuSE Linux Enterprise Server ver. 11 (Architecture Independent), RPM Package	5.5.5	21.0M	Download
(MySQL-5.5.5_m3-1.sles11.src.rpm)		MD5: e5a13c562e1846666367d211575f2f29	
Red Hat & Oracle Enterprise Linux 5 (Architecture Independent), RPM Package	5.5.5	21.0M	Download
(MySQL-5.5.5_m3-1.rhel5.src.rpm)		MD5: dd01ca7a34be239d62eef7c1e3d4c3fc	
SuSE Linux Enterprise Server 10 (Architecture Independent), RPM Package	5.5.5	21.0M	Download
(MySQL-5.5.5_m3-1.sles10.src.rpm)		MD5: d5b5e453f40e4ecb35c2731f4ca9ab2b	
Red Hat & Oracle Enterprise Linux 4 (Architecture Independent), RPM Package	5.5.5	21.0M	Download
(MySQL-5.5.5_m3-1.rhel4.src.rpm)		MD5: 019c635bd31625d206284e565079dcdc	
Generic Linux (Architecture Independent), Compressed TAR Archive	5.5.5	21.0M	Download
(mysql-5.5.5-m3.tar.gz)		MD5: ad27f6561d6010c9346ffeca6de403fa	

图10-2 MySQL开发中版本的源代码下载

图10-3 MySQL亚洲下载镜像

下载的文件是tar.gz结尾的文件，可以通过Linux的tar命令、Windows的WinRAR工具来进行解压，解压后得到一个文件夹，这里面就包含了MySQL数据库的所有源代码，源代码的结构如图10-4所示。

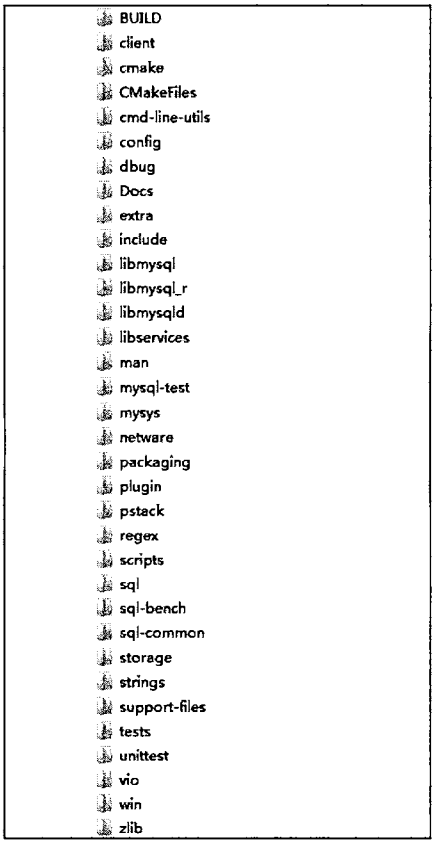


图10-4 MySQL源代码目录结构

所有存储引擎的源代码都被放在storage的文件夹下，其源代码结构如图10-5所示。

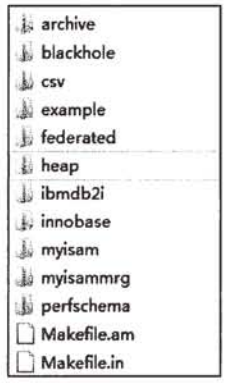


图10-5 存储引擎源代码文件夹

可以看到，所有存储引擎的源代码都在这里。文件夹名一般就是存储引擎的名称，如archive、blackhole、csv、fedorated、heap、ibmdb2i、myisam、innobase。从MySQL 5.5版本开始，InnoDB Plugin已经作为默认的InnoDB存储引擎版本；而在MySQL 5.1的源代码中，应该可以看到两个版本的InnoDB存储引擎源代码，如图10-6所示。

图10-6 MySQL 5.1存储引擎目录结构

可以看到有innobase和innodb_plugin两个文件夹：innobase文件夹是旧的InnoDB存储引擎的源代码；innodb_plugin文件夹是InnoDB Plugin存储引擎的源代码。如果你想将InnoDB Plugin直接静态编译到MySQL数据库中，那么需要删除innobase文件夹，再将innodb_plugin文件夹重命名为innobase。

10.2 InnoDB源代码结构

进入InnoDB存储引擎的源代码文件夹，应该可以看到如图10-7所示的源代码结构。

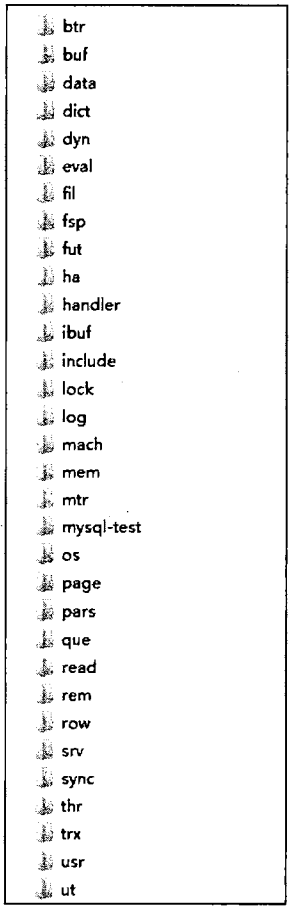


图10-7 InnoDB存储引擎源代码的文件夹结构

下面介绍一些主要文件夹内源代码的具体作用：

- btr：B+树的实现。
- buf：缓冲池的实现，包括LRU算法、Flush刷新算法等。
- dict：InnoDB存储引擎内存数据字典的实现。
- dyn：InnoDB存储引擎动态数组的实现。
- fil：InnoDB存储引擎中文件数据结构以及对于文件的一些操作。
- fsp：你可以理解为file space，即对InnoDB存储引擎物理文件的管理，如页、区、段等。

- ha：哈希算法的实现。
- handler：继承于MySQL的handler，插件式存储引擎的实现。
- ibuf：插入缓冲的实现。
- include：InnoDB将头文件（.h，.ic）都统一放在这个文件夹下。
- lock：InnoDB存储引擎锁的实现，如S锁、X锁以及定义锁的一系列算法。
- log：日志缓冲和重组日志文件的实现。对重组日志感兴趣的，应该好好阅读该源代码。
- mem：辅助缓冲池的实现，用来申请一些数据结构的内存。
- mtr：事务的底层实现。
- os：封装一些对于操作系统的操作。
- page：页的实现。
- row：对于各种类型行数据的操作。
- srv：对于InnoDB存储引擎参数的设计。
- sync：InnoDB存储引擎互斥量（Mutex）的实现。
- thr：InnoDB储存引擎封装的可移植的线程库。
- trx：事务的实现。
- ut：工具类。

10.3 编译和调试InnoDB源代码

10.3.1 Windows下的调试

在Windows平台下，可以通过Visual Studion 2003、2005和2008开发工具对MySQL的源代码进行编译和调试。在此之前，需要预先安装如下的工具：

- CMake：可以从http://www.cmake.org下载。
- bison：可以从http://gnuwin32.sourceforge.net/packages/bison.htm下载。

安装之后，还需要通过configure.js这个命令进行配置：

```
C:\workdir>win\configure.js options
```

option比较重要的选项如下所示。

- WITH_INNOBASE_STORAGE_ENGINE：支持InnoDB存储引擎。
- WITH_PARTITION_STORAGE_ENGINE：分区支持。
- WITH_ARCHIVE_STORAGE_ENGINE：支持Archive存储引擎。
- WITH_BLACKHOLE_STORAGE_ENGINE：支持Blackhole存储引擎。
- WITH_EXAMPLE_STORAGE_ENGINE：支持Example存储引擎，这个存储引擎是展示给开发人员的，你可以从这个存储引擎开始构建自己的存储引擎。
- WITH_FEDERATED_STORAGE_ENGINE：支持Federated存储引擎。
- WITH_NDBCLUSTER_STORAGE_ENGINE：支持NDB Cluster存储引擎。

如果只是比较关心InnoDB存储引擎，可以这样进行设置，如图10-8所示。

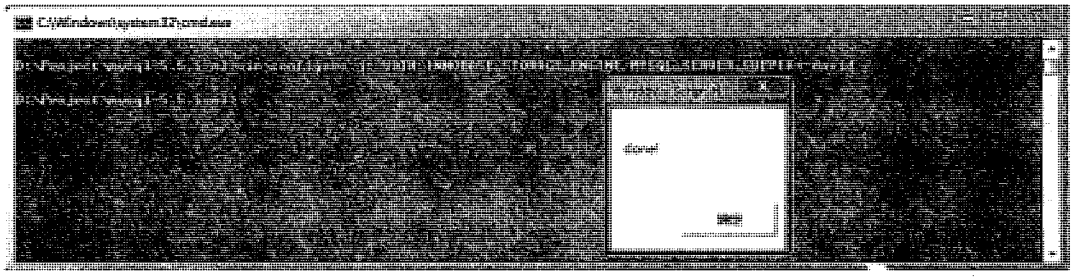


图10-8 configure.js配置

之后，可以根据你使用的是Visual Studio 2005还是Visual Studio 2008，在win文件下运行build-vsx.bat文件来生成Visual Studio的工程文件。build-vs8.bat表示Visual Studio 2005，build-vs8_x64.bat表示需要编译64位的MySQL数据库。如我们需要在32位的操作系统下使用Visual Studio 2008进行调试工作，则可以使用如下命令：

```
D:\Project\mysql-5.5.5-m3>win\build-vs9.bat
-- Check for working C compiler: C:/Program Files/Microsoft Visual Studio
9.0/VC/bin/cl.exe
-- Check for working C compiler: C:/Program Files/Microsoft Visual Studio
9.0/VC/bin/cl.exe -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: C:/Program Files/Microsoft Visual Studio
9.0/VC/bin/cl.exe
-- Check for working CXX compiler: C:/Program Files/Microsoft Visual Studio
9.0/VC/bin/cl.exe -- works
-- Detecting CXX compiler ABI info
```

```
-- Detecting CXX compiler ABI info - done
-- Check size of void *
-- Check size of void * - done
SIZEOF_VOIDP=4
-- Looking for include files HAVE_CXXABI_H
-- Looking for include files HAVE_CXXABI_H - not found.
-- Looking for include files HAVE_NDIR_H
-- Looking for include files HAVE_NDIR_H - not found.
-- Looking for include files HAVE_SYS_NDIR_H
-- Looking for include files HAVE_SYS_NDIR_H - not found.
-- Looking for include files HAVE_ASM_TERMBITS_H
-- Looking for include files HAVE_ASM_TERMBITS_H - not found.
-- Looking for include files HAVE_TERMBITS_H
-- Looking for include files HAVE_TERMBITS_H - not found.
-- Looking for include files HAVE_VIS_H
-- Looking for include files HAVE_VIS_H - not found.
-- Looking for include files HAVE_WCHAR_H
-- Looking for include files HAVE_WCHAR_H - found
-- Looking for include files HAVE_WCTYPE_H
-- Looking for include files HAVE_WCTYPE_H - found
-- Looking for include files HAVE_XFS_XFS_H
-- Looking for include files HAVE_XFS_XFS_H - not found.
-- Looking for include files CMAKE_HAVE_PTHREAD_H
-- Looking for include files CMAKE_HAVE_PTHREAD_H - not found.
-- Found Threads: TRUE
-- Looking for pthread_rwlockattr_setkind_np
-- Looking for pthread_rwlockattr_setkind_np - not found
-- Performing Test HAVE_SOCKADDR_IN_SIN_LEN
-- Performing Test HAVE_SOCKADDR_IN_SIN_LEN - Failed
-- Performing Test HAVE_SOCKADDR_IN6_SIN6_LEN
-- Performing Test HAVE_SOCKADDR_IN6_SIN6_LEN - Failed
-- Cannot find wix 3, installer project will not be generated
-- Configuring done
-- Generating done
-- Build files have been written to: D:/Project/mysql-5.5.5-m3
```

这样就生成了MySQL.sln的工程文件，打开这个工程文件并将mysqld这个项目设置为默认的启动项，就可以进行MySQL的编译和调试了。

之后的编译、断点的设置和调试，与在Visual Studio下操作一般的程序没有什么区别，图10-9演示了对InnoDB存储引擎master thread进行调试。

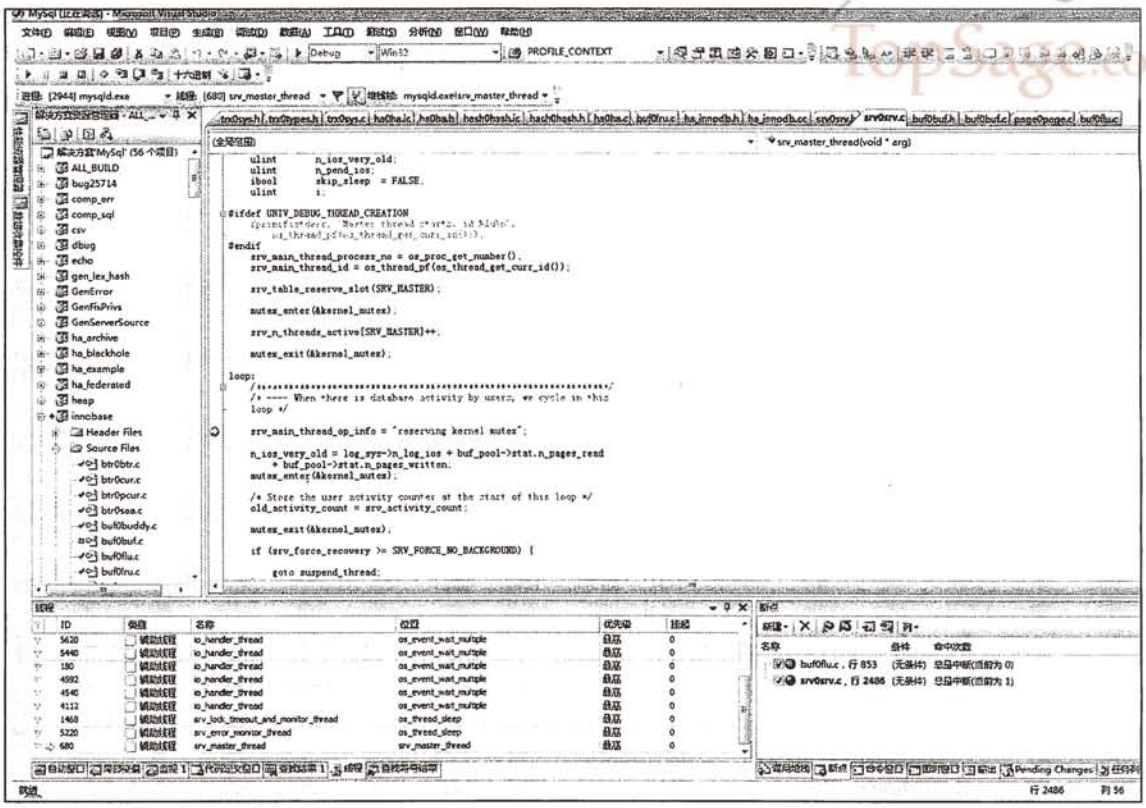


图10-9 调试master thread

10.3.2 Linux下的调试

Linux下的调试，通常使用Eclipse。其他一些类Unix操作系统，如Solaris、FreeBSD、MAC，同样可以使用Eclipse进行调试。首先到http://www.eclipse.org/downloads/ 下载并安装Eclipse IDE for C/C++ Developers。然后，解压MySQL源代码到指定目录，如解压到/root/workspace/mysql-5.5.5-m3，接着运行如下命令产生Make文件（Eclipse会使用产生的这些Make文件）：

```
[root@xen-server mysql-5.5.5-m3]# BUILD/compile-amd64-debug-max-no-ndb -c
```

BUILD下有很多compile文件，你可以选择你所需要的文件。本书编译的平台是64位的Linux系统，并且我希望可以进行Debug调试，因此选择了compile-amd64-debug-max-no-ndb文件。注意-c选项，这个选项只生产Make文件，不进行编译。

接着打开Eclipse，新建一个C++的项目，如图10-10所示。

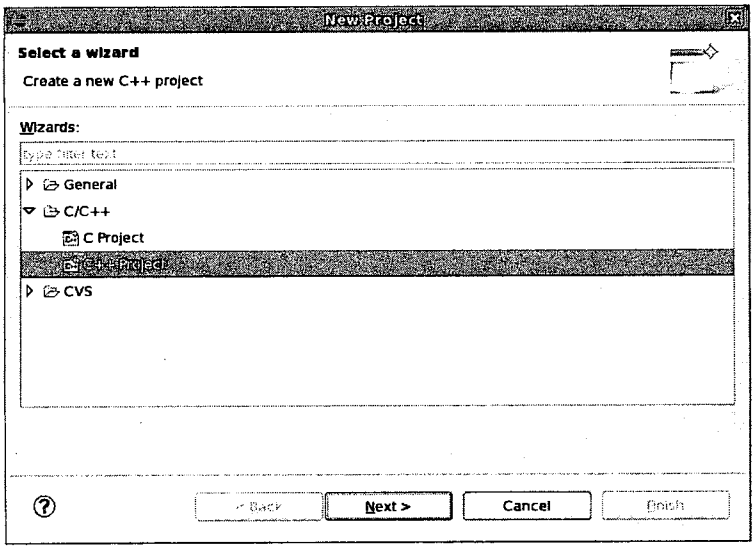


图10-10 新建一个C++项目

给项目取个名称，如这里的项目名为mysql_5_5_5，并选择一个空的项目，如图10-11所示。

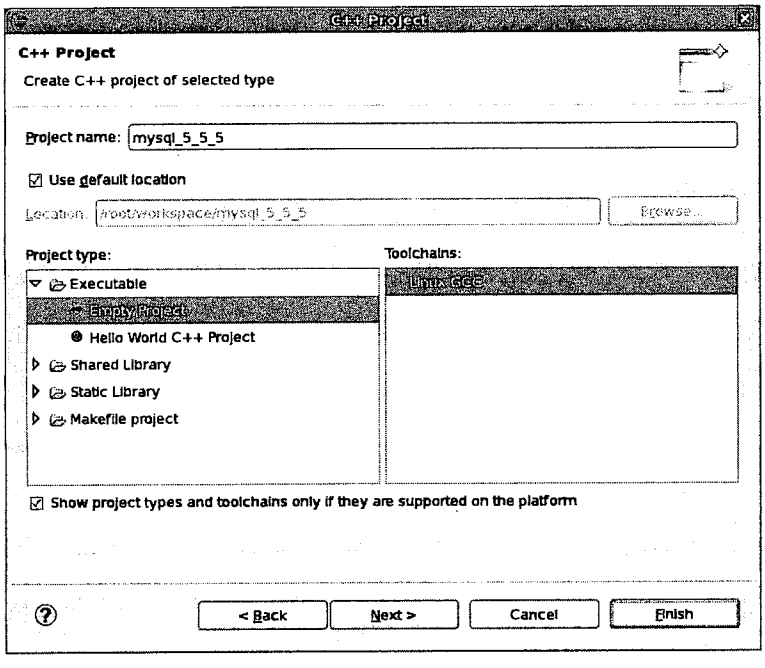


图10-11 选择空项目

选择Finish按钮后，可以看到新产生的一个空项目，如图10-12所示。

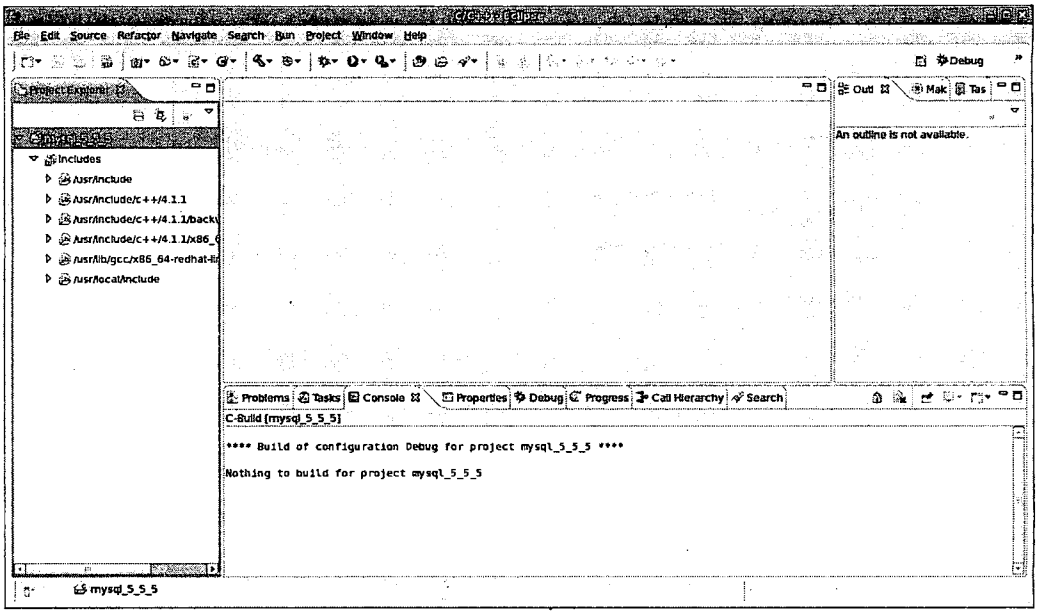


图10-12 新建的C++项目

之后选择左边的Project Explorer，右击项目mysql_5_5_5，选择新建文件夹，将文件夹/root/workspace/mysql-5.5.5-m3导入工程中，如图10-13所示。

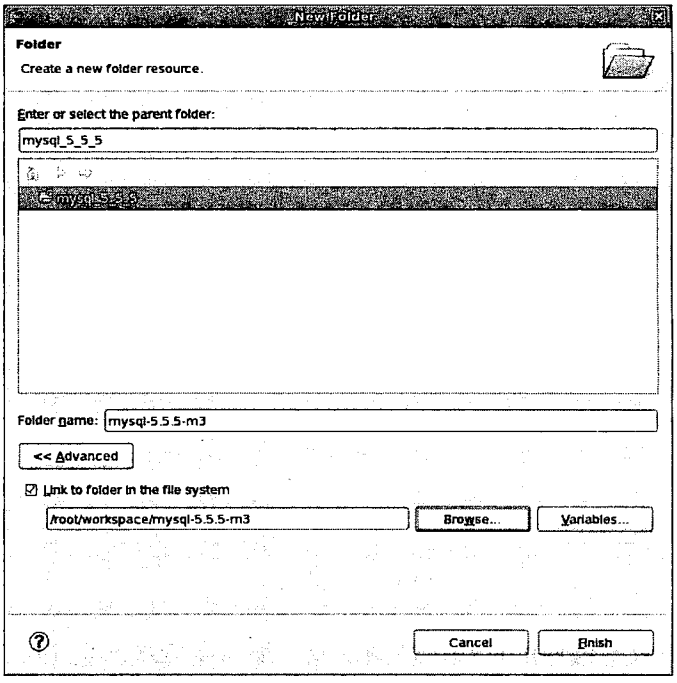


图10-13 选择文件夹

导入文件夹后，再右击项目名mysql_5_5_5，选择项目属性，在C/C++ Build选项这里进行设置，需要将Build directory选择为源代码所在路径，如图10-14所示。

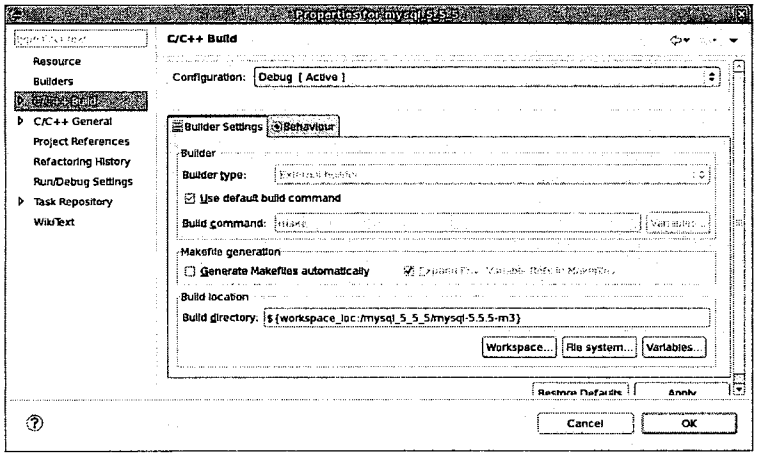


图10-14 编译配置

编译配置完后，程序就会自动开始执行编译工作了，如图10-15所示。

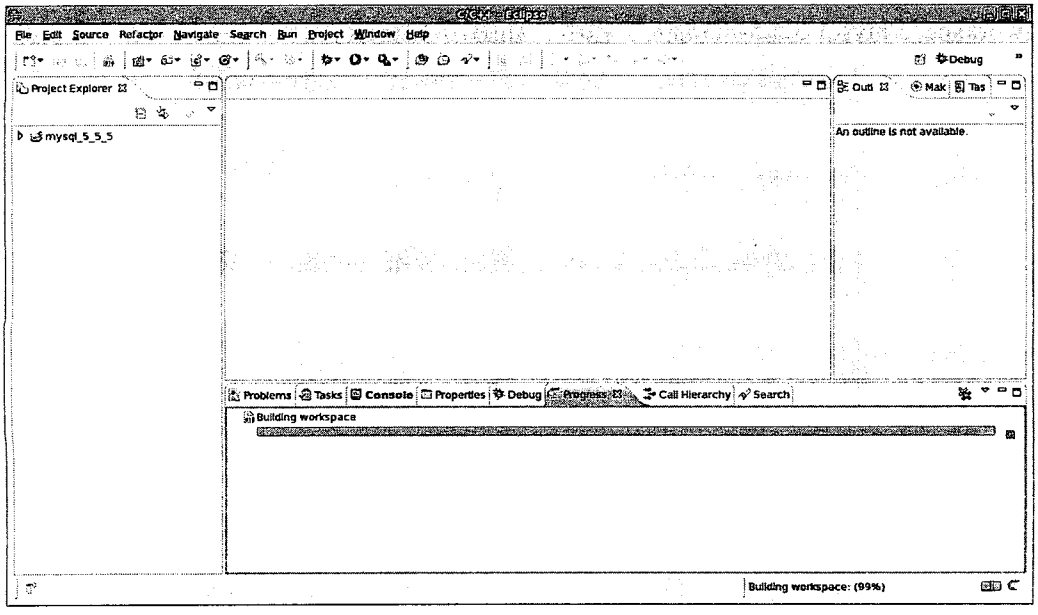


图10-15 执行编译

上述的这个过程只是编译的过程，换句话说，编译完后就产生了mysqld这样的执行文件。如果想要进行调试，还需要在Debug这里进行如下的配置，如图10-16所示。

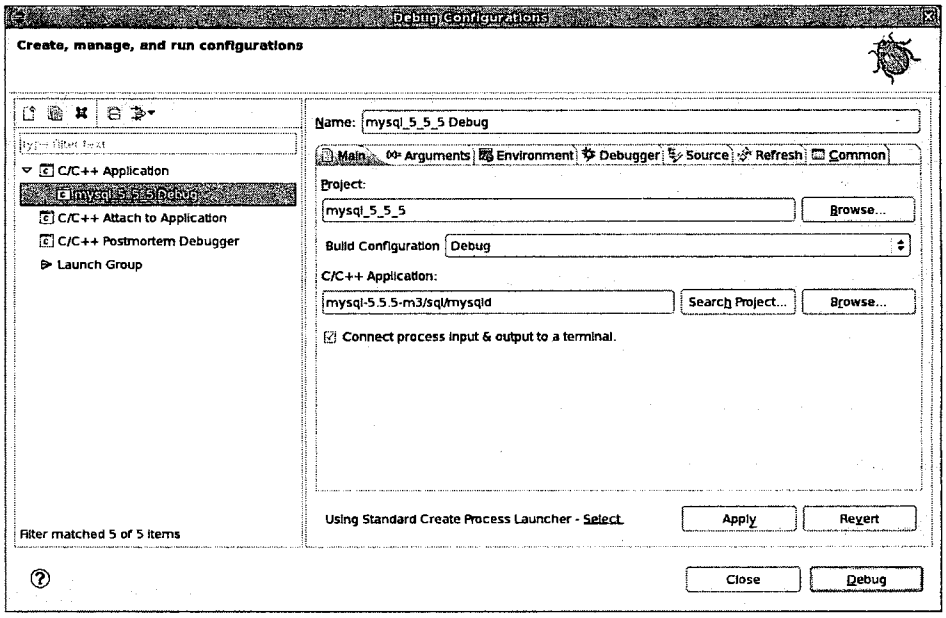


图10-16 Debug配置

另外如果需要配置一些额外的参数，需要切换到Arguments选项，如图10-17所示。

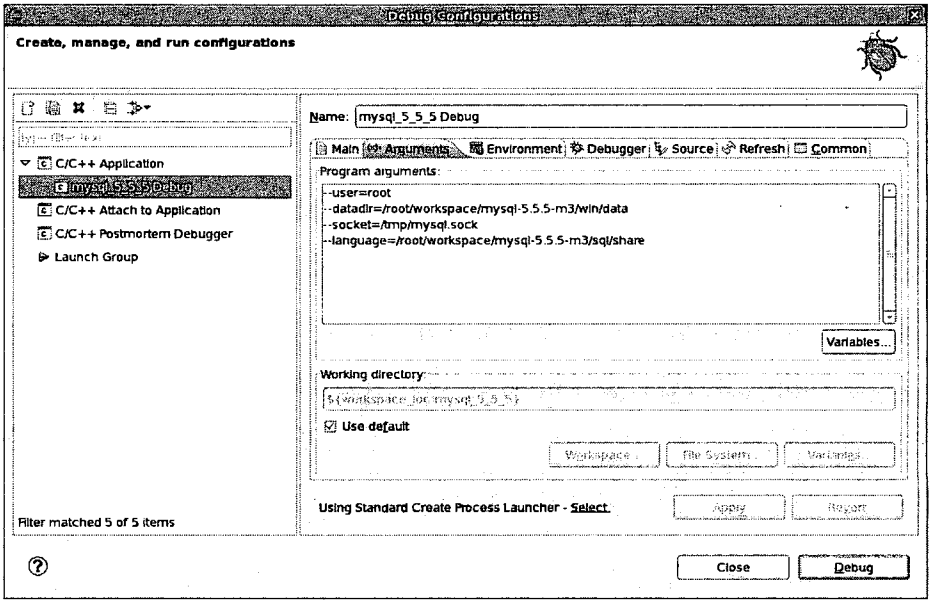


图10-17 调试参数

之后就可以设置断点，进行调试工作了，这和一般的程序并没有什么不同，如图10-18所示。

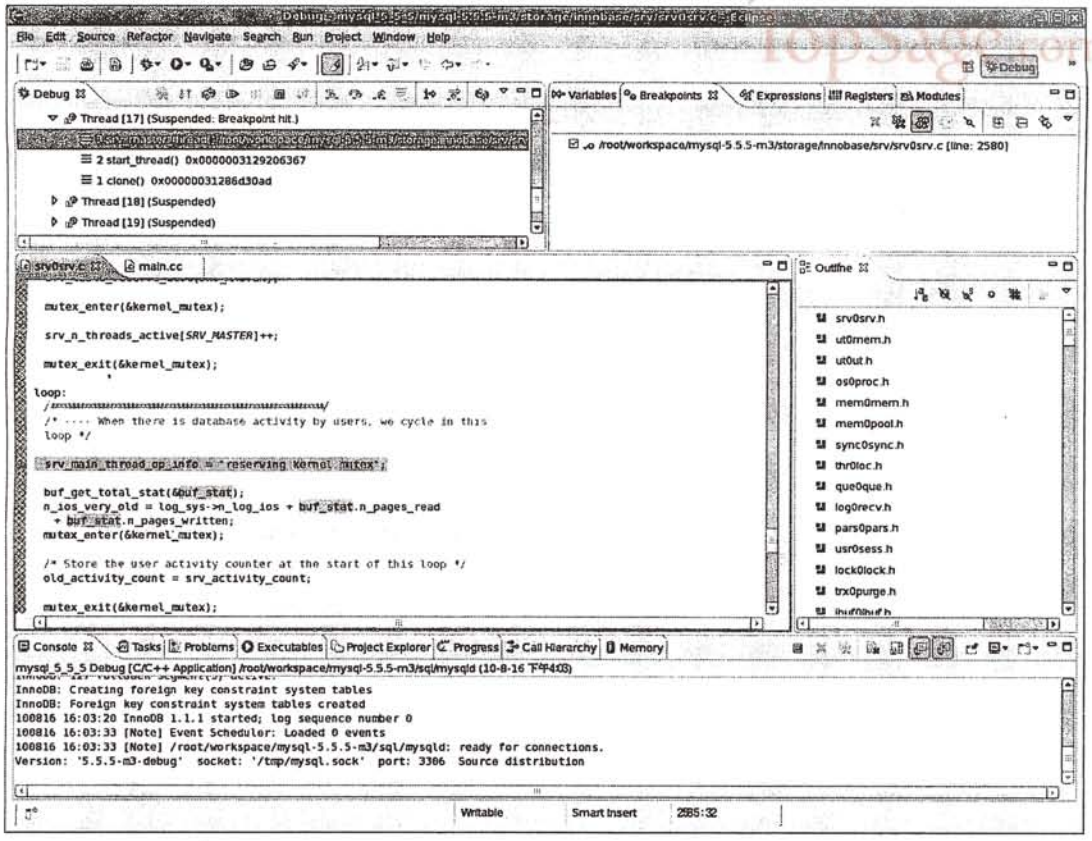


图10-18 用Eclipse进行调试

10.4 小结

MySQL数据库和InnoDB存储引擎都是开源的，我们可以通过常用的开发工具，如Visual Studio、Eclipse对其进行编译和调试，以此来更好地了解数据库内部运行机制。有能力的开发人员可以进一步扩展数据库的功能，这就是开源的魅力；而这些，在Oracle、Microsoft SQL Server、DB2这些商业数据库中是永远不可能发生的。

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

附录A Secondary Buffer Pool For InnoDB

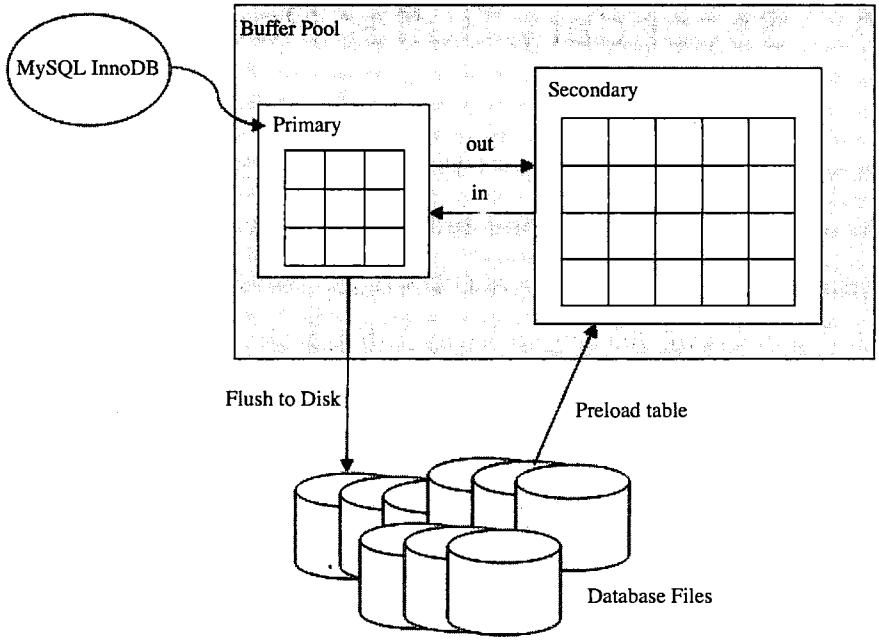
Secondary Buffer Pool是我开发的一个针对于InnoDB存储引擎的补丁，通过该补丁可以实现将固态硬盘作为InnoDB存储引擎的辅助缓冲池（或者称为L2 Cache），通过利用固态硬盘的高随机读取的性能来提高数据库的整体性能。其官方网站为：http://code.google.com/p/david-mysql-tools/wiki/innodb_secondary_buffer_pool。

当前，基于磁盘数据库是通过页或块的方式来进行数据库的存储管理。当要读取数据时，首先判断内存缓冲池中是否有该页的缓存，如果存在，读取内存中的页即可。若不存在，则将磁盘上的页读入内存池进行缓冲后再读取。同样，对于写入操作，首先在缓冲池的页中完成，一般称之为脏页，然后由后台的调度线程或进程将脏页刷新同步回磁盘。从上述过程可以发现，数据库的性能很大程度上依赖于缓冲池的大小。而当前数据库的物理大小通常大于内存的容量，因此通过利用固态硬盘的高随机读取而设计的辅助缓冲池，可以避免物理的磁盘随机读取操作，从而提高数据库的性能。

图A-1显示了辅助缓冲池的工作原理。当主缓冲池中的页通过LRU算法移出时（图中的out箭头），若该页不在辅助缓冲池内，则将该页放入辅助缓冲池中。当下一次再需要请求这个页时，首先判断页是否在辅助缓冲池内，若在，则读入主缓冲池中（图中的in箭头），这时就不需要访问磁盘上的页了。若缓冲池内的页修改了，并且也存在于辅助缓冲池内，这时并不需要修改辅助缓冲池内的页，因为一个页可能在缓冲池内被多次修改。每次同步辅助缓冲池中的页会使固态硬盘的写入性能问题暴露，所以此时把辅助缓冲池中的页放入空闲列表的尾端即可。只有当主缓冲池的页被移出时，才再次放入辅助缓冲池中。可以看出，辅助缓冲池的设计是作为一个大量读操作的场所，这也符合固态硬盘的特性。

虽然辅助缓冲池可以提高数据库的性能，但是在第一次从主内存缓冲池放入辅助缓冲池时，需要进行页的拷贝操作。如果并发量很大时，会产生性能瓶颈，即数据库需等待页被放入辅助缓冲池。为了避免出现这种情形，我们设计了一种预载入技术，在数据库启动

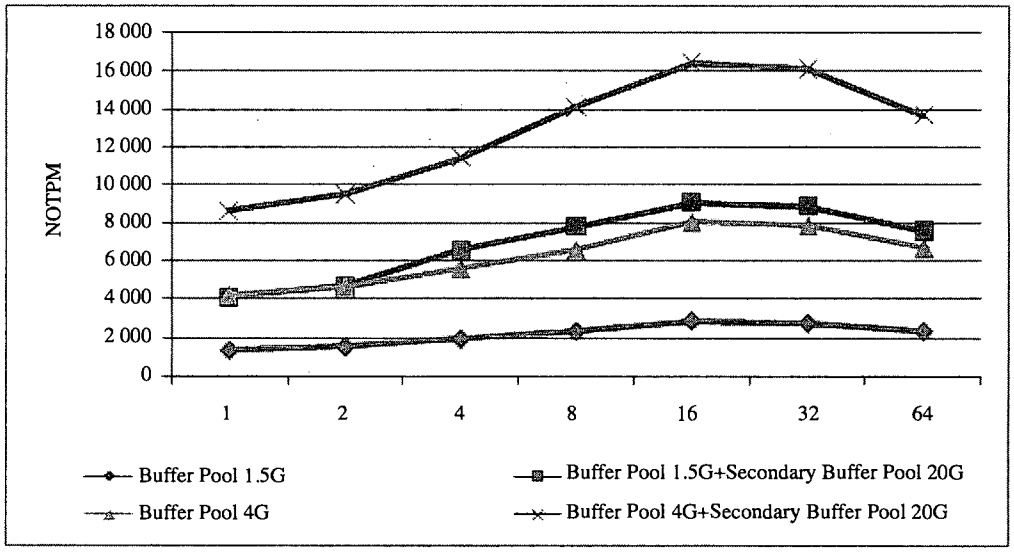
时，可以通过数据库的配置文件将指定库中表的数据页和索引页放入辅助缓冲池（如图A-1中的Preload table箭头所示）。



图A-1 Secondary Buffer Pool体系架构

下面通过开启Secondary Buffer Pool功能对比数据库的在线事务处理（OLTP）能力。我们选取了两种不同的测试环境，每种测试环境又考虑了是否开启辅助缓冲。第一种环境，主缓冲池的大小为1.5GB，即只能缓冲数据库15%的数据，这时内存缓冲池明显较小。第二种情况是4GB，即可以缓冲数据库40%的数据，这和大部分实际应用中数据库和内存缓冲池的大小关系相当。辅助缓冲池都设置为20GB，数据库的所有数据文件都可以被缓冲入辅助缓冲池中。

测试得到的结果如图A-2所示。可以看到辅助缓冲池的引入极大地提高了数据库的性能。在主缓冲池为1.5GB的情况下，启用辅助缓冲池后数据库的性能可有3.0～3.2倍的提高。在主缓冲池为4GB的情况下，启用辅助缓冲池后数据库的性能可有1.0～1.2倍的提高。可见当主缓冲池越小，辅助缓冲池对于数据库性能提高的帮助就越大。



图A-2 TPC-C测试结果

附录B　Master Thread源代码

```
/*************************************************************************//**
The master thread controlling the server.
@return    a dummy parameter */
UNIV_INTERN
os_thread_ret_t
srv_master_thread (
/*===============*/
     void*     arg __attribute__ ((unused)))
                         /*!< in: a dummy parameter required by
                         os_thread_create */
{
     buf_pool_stat_t buf_stat;
     os_event_tevent;
     ulint               old_activity_count;
     ulint               n_pages_purged       = 0;
     ulint               n_bytes_merged;
     ulint               n_pages_flushed;
     ulint               n_bytes_archived;
     ulint               n_tables_to_drop;
     ulint               n_ios;
     ulint               n_ios_old;
     ulint               n_ios_very_old;
     ulint               n_pend_ios;
     ulint               next_itr_time;
     ulint               i;

  #ifdef UNIV_DEBUG_THREAD_CREATION
     fprintf (stderr, "Master thread starts, id %lu\n",
              os_thread_pf (os_thread_get_curr_id ())) ;
  #endif

  #ifdef UNIV_PFS_THREAD
     pfs_register_thread (srv_master_thread_key) ;
  #endif

     srv_main_thread_process_no = os_proc_get_number () ;
```

```
srv_main_thread_id = os_thread_pf (os_thread_get_curr_id ()) ;

srv_table_reserve_slot (SRV_MASTER) ;

mutex_enter (&kernel_mutex) ;

srv_n_threads_active[SRV_MASTER]++;

mutex_exit (&kernel_mutex) ;

loop:
/*****************************************************************/
/* ---- When there is database activity by users, we cycle in this
loop */

srv_main_thread_op_info = "reserving kernel mutex";

buf_get_total_stat (&buf_stat) ;
n_ios_very_old = log_sys->n_log_ios + buf_stat.n_pages_read
         + buf_stat.n_pages_written;
mutex_enter (&kernel_mutex) ;

/* Store the user activity counter at the start of this loop */
old_activity_count = srv_activity_count;

mutex_exit (&kernel_mutex) ;

if  (srv_force_recovery >= SRV_FORCE_NO_BACKGROUND)  {

         goto suspend_thread;
}

/* ---- We run the following loop approximately once per second
when there is database activity */

srv_last_log_flush_time = time (NULL) ;
next_itr_time = ut_time_ms () ;

for  (i = 0; i < 10; i++)  {
         ulint      cur_time = ut_time_ms () ;

         buf_get_total_stat (&buf_stat) ;

         n_ios_old = log_sys->n_log_ios + buf_stat.n_pages_read
```

```
            + buf_stat.n_pages_written;

srv_main_thread_op_info = "sleeping";
srv_main_1_second_loops++;

if (next_itr_time > cur_time) {

            /* Get sleep interval in micro seconds. We use
            ut_min() to avoid long sleep in case of
            wrap around. */
            os_thread_sleep (ut_min (1000000,
                                     (next_itr_time - cur_time)
                                      * 1000)) ;
            srv_main_sleeps++;
}

/* Each iteration should happen at 1 second interval. */
next_itr_time = ut_time_ms () + 1000;

/* ALTER TABLE in MySQL requires on Unix that the table handler
can drop tables lazily after there no longer are SELECT
queries to them. */

srv_main_thread_op_info = "doing background drop tables";

row_drop_tables_for_mysql_in_background () ;

srv_main_thread_op_info = "";

if (srv_fast_shutdown && srv_shutdown_state > 0) {

            goto background_loop;
}

/* Flush logs if needed */
srv_sync_log_buffer_in_background () ;

srv_main_thread_op_info = "making checkpoint";
log_free_check () ;

/* If i/os during one second sleep were less than 5% of
capacity, we assume that there is free disk i/o capacity
available, and it makes sense to do an insert buffer merge. */
```

```
buf_get_total_stat (&buf_stat) ;
n_pend_ios = buf_get_n_pending_ios ()
          + log_sys->n_pending_writes;
n_ios = log_sys->n_log_ios + buf_stat.n_pages_read
          + buf_stat.n_pages_written;
if  (n_pend_ios < SRV_PEND_IO_THRESHOLD
    &&  (n_ios - n_ios_old < SRV_RECENT_IO_ACTIVITY))  {
          srv_main_thread_op_info = "doing insert buffer merge";
          ibuf_contract_for_n_pages (FALSE, PCT_IO (5)) ;

          /* Flush logs if needed */
          srv_sync_log_buffer_in_background () ;
}

if  (UNIV_UNLIKELY (buf_get_modified_ratio_pct ()
                  > srv_max_buf_pool_modified_pct))  {

          /* Try to keep the number of modified pages in the
          buffer pool under the limit wished by the user */

          srv_main_thread_op_info =
                  "flushing buffer pool pages";
          n_pages_flushed = buf_flush_list (
                  PCT_IO (100) , IB_ULONGLONG_MAX) ;

} else if  (srv_adaptive_flushing)  {

          /* Try to keep the rate of flushing of dirty
          pages such that redo log generation does not
          produce bursts of IO at checkpoint time. */
          ulint n_flush = buf_flush_get_desired_flush_rate () ;

          if  (n_flush)  {
                  srv_main_thread_op_info =
                                  "flushing buffer pool pages";
                  n_flush = ut_min (PCT_IO (100) , n_flush) ;
                  n_pages_flushed =
                                  buf_flush_list (
                                  n_flush,
                                  IB_ULONGLONG_MAX) ;
          }
}

if  (srv_activity_count == old_activity_count)  {
```

```
                        /* There is no user activity at the moment, go to
                        the background loop */

                        goto background_loop;
                }
        }

        /* ---- We perform the following code approximately once per
        10 seconds when there is database activity */

#ifdef MEM_PERIODIC_CHECK
        /* Check magic numbers of every allocated mem block once in 10
        seconds */
        mem_validate_all_blocks () ;
#endif
        /* If i/os during the 10 second period were less than 200% of
        capacity, we assume that there is free disk i/o capacity
        available, and it makes sense to flush srv_io_capacity pages.

        Note that this is done regardless of the fraction of dirty
        pages relative to the max requested by the user. The one second
        loop above requests writes for that case. The writes done here
        are not required, and may be disabled. */

        buf_get_total_stat (&buf_stat) ;
        n_pend_ios = buf_get_n_pending_ios ()  + log_sys->n_pending_writes;
        n_ios = log_sys->n_log_ios + buf_stat.n_pages_read
                + buf_stat.n_pages_written;

        srv_main_10_second_loops++;
        if  (n_pend_ios < SRV_PEND_IO_THRESHOLD
            &&  (n_ios - n_ios_very_old < SRV_PAST_IO_ACTIVITY))  {

                srv_main_thread_op_info = "flushing buffer pool pages";
                buf_flush_list (PCT_IO (100) , IB_ULONGLONG_MAX) ;

                /* Flush logs if needed */
                srv_sync_log_buffer_in_background () ;
        }

        /* We run a batch of insert buffer merge every 10 seconds,
        even if the server were active */

        srv_main_thread_op_info = "doing insert buffer merge";
```

```
ibuf_contract_for_n_pages (FALSE, PCT_IO (5)) ;

/* Flush logs if needed */
srv_sync_log_buffer_in_background () ;

if  (srv_n_purge_threads == 0)  {
        srv_main_thread_op_info = "master purging";

        srv_master_do_purge () ;

        if  (srv_fast_shutdown && srv_shutdown_state > 0)  {

                goto background_loop;
        }
}

srv_main_thread_op_info = "flushing buffer pool pages";

/* Flush a few oldest pages to make a new checkpoint younger */

if  (buf_get_modified_ratio_pct ()  > 70)  {

        /* If there are lots of modified pages in the buffer pool
        (> 70 %) , we assume we can afford reserving the disk (s)  for
        the time it requires to flush 100 pages */

        n_pages_flushed = buf_flush_list (
                PCT_IO (100) , IB_ULONGLONG_MAX) ;
} else {
        /* Otherwise, we only flush a small number of pages so that
        we do not unnecessarily use much disk i/o capacity from
        other work */

        n_pages_flushed = buf_flush_list (
                PCT_IO (10) , IB_ULONGLONG_MAX) ;
}

srv_main_thread_op_info = "making checkpoint";

/* Make a new checkpoint about once in 10 seconds */

log_checkpoint (TRUE, FALSE) ;

srv_main_thread_op_info = "reserving kernel mutex";
```

```
mutex_enter (&kernel_mutex) ;

/* ---- When there is database activity, we jump from here back to
the start of loop */

if  (srv_activity_count != old_activity_count)  {
          mutex_exit (&kernel_mutex) ;
          goto loop;
}

mutex_exit (&kernel_mutex) ;

/* If the database is quiet, we enter the background loop */

/*****************************************************************/
background_loop:
/* ---- In this loop we run background operations when the server
is quiet from user activity. Also in the case of a shutdown, we
loop here, flushing the buffer pool to the data files. */

/* The server has been quiet for a while: start running background
operations */
srv_main_background_loops++;
srv_main_thread_op_info = "doing background drop tables";

n_tables_to_drop = row_drop_tables_for_mysql_in_background () ;

if  (n_tables_to_drop > 0)  {
          /* Do not monopolize the CPU even if there are tables waiting
          in the background drop queue.  (It is essentially a bug if
          MySQL tries to drop a table while there are still open handles
          to it and we had to put it to the background drop queue.)  */

          os_thread_sleep (100000) ;
}

if  (srv_n_purge_threads == 0)  {
          srv_main_thread_op_info = "master purging";

          srv_master_do_purge () ;
}

srv_main_thread_op_info = "reserving kernel mutex";
```

```
mutex_enter (&kernel_mutex) ;
if  (srv_activity_count != old_activity_count)  {
          mutex_exit (&kernel_mutex) ;
          goto loop;
}
mutex_exit (&kernel_mutex) ;

srv_main_thread_op_info = "doing insert buffer merge";

if  (srv_fast_shutdown && srv_shutdown_state > 0)  {
          n_bytes_merged = 0;
} else {
          /* This should do an amount of IO similar to the number of
          dirty pages that will be flushed in the call to
          buf_flush_list below. Otherwise, the system favors
          clean pages over cleanup throughput. */
          n_bytes_merged = ibuf_contract_for_n_pages (FALSE,
                                                      PCT_IO (100)) ;
}

srv_main_thread_op_info = "reserving kernel mutex";

mutex_enter (&kernel_mutex) ;
if  (srv_activity_count != old_activity_count)  {
          mutex_exit (&kernel_mutex) ;
          goto loop;
}
mutex_exit (&kernel_mutex) ;

flush_loop:
srv_main_thread_op_info = "flushing buffer pool pages";
srv_main_flush_loops++;
if  (srv_fast_shutdown < 2)  {
          n_pages_flushed = buf_flush_list (
                    PCT_IO (100) , IB_ULONGLONG_MAX) ;
} else {
          /* In the fastest shutdown we do not flush the buffer pool
          to data files: we set n_pages_flushed to 0 artificially. */

          n_pages_flushed = 0;
}

srv_main_thread_op_info = "reserving kernel mutex";
```

```
mutex_enter (&kernel_mutex) ;
if  (srv_activity_count != old_activity_count)  {
          mutex_exit (&kernel_mutex) ;
          goto loop;
}
mutex_exit (&kernel_mutex) ;

srv_main_thread_op_info = "waiting for buffer pool flush to end";
buf_flush_wait_batch_end (NULL, BUF_FLUSH_LIST) ;

/* Flush logs if needed */
srv_sync_log_buffer_in_background () ;

srv_main_thread_op_info = "making checkpoint";

log_checkpoint (TRUE, FALSE) ;

if  (buf_get_modified_ratio_pct ()  > srv_max_buf_pool_modified_pct)  {

          /* Try to keep the number of modified pages in the
          buffer pool under the limit wished by the user */

          goto flush_loop;
}

srv_main_thread_op_info = "reserving kernel mutex";

mutex_enter (&kernel_mutex) ;
if  (srv_activity_count != old_activity_count)  {
          mutex_exit (&kernel_mutex) ;
          goto loop;
}
mutex_exit (&kernel_mutex) ;
/*
srv_main_thread_op_info = "archiving log  (if log archive is on) ";

log_archive_do (FALSE, &n_bytes_archived) ;
*/
n_bytes_archived = 0;

/* Keep looping in the background loop if still work to do */

if  (srv_fast_shutdown && srv_shutdown_state > 0)  {
          if  (n_tables_to_drop + n_pages_flushed
```

```
            + n_bytes_archived != 0)  {

                    /* If we are doing a fast shutdown  (= the default)
                    we do not do purge or insert buffer merge. But we
                    flush the buffer pool completely to disk.
                    In a 'very fast' shutdown we do not flush the buffer
                    pool to data files: we have set n_pages_flushed to
                    0 artificially. */

                    goto background_loop;
            }
    } else if  (n_tables_to_drop
                + n_pages_purged + n_bytes_merged + n_pages_flushed
                + n_bytes_archived != 0)  {
            /* In a 'slow' shutdown we run purge and the insert buffer
            merge to completion */

            goto background_loop;
    }

    /* There is no work for background operations either: suspend
    master thread to wait for more server activity */

    suspend_thread:
    srv_main_thread_op_info = "suspending";

    mutex_enter (&kernel_mutex) ;

    if  (row_get_background_drop_list_len_low ()  > 0)  {
            mutex_exit (&kernel_mutex) ;

            goto loop;
    }

    event = srv_suspend_thread () ;

    mutex_exit (&kernel_mutex) ;

    /* DO NOT CHANGE THIS STRING. innobase_start_or_create_for_mysql ()
    waits for database activity to die down when converting < 4.1.x
    databases, and relies on this string being exactly as it is. InnoDB
    manual also mentions this string in several places. */
    srv_main_thread_op_info = "waiting for server activity";
```

```
        os_event_wait (event) ;

        if  (srv_shutdown_state == SRV_SHUTDOWN_EXIT_THREADS)  {
                /* This is only extra safety, the thread should exit
                already when the event wait ends */

                os_thread_exit (NULL) ;

        }

        /* When there is user activity, InnoDB will set the event and the
        main thread goes back to loop. */

        goto loop;

        OS_THREAD_DUMMY_RETURN;      /* Not reached, avoid compiler warning */
}
```

附录C　Doublewrite源代码

trx0sys.h

```
/** Doublewrite control struct */
struct trx_doublewrite_struct{
     mutex_t  mutex;             /*!< mutex protecting the first_free field and
                                 write_buf */
     ulint    block1;            /*!< the page number of the first
                                 doublewrite block (64 pages) */
     ulint    block2;            /*!< page number of the second block */
     ulint    first_free;/*!< first free position in write_buf measured
                                 in units of UNIV_PAGE_SIZE */
     byte*    write_buf; /*!< write buffer used in writing to the
                                 doublewrite buffer, aligned to an
                                 address divisible by UNIV_PAGE_SIZE
                                 (which is required by Windows aio) */
     byte*    write_buf_unaligned;
                                 /*!< pointer to write_buf, but unaligned */
     buf_page_t**
              buf_block_arr;     /*!< array to store pointers to the buffer
                                 blocks which have been cached to write_buf */
};
```

trx0sys.c

```
/*****************************************************************//**
Determines if a page number is located inside the doublewrite buffer.
@return TRUE if the location is inside the two blocks of the
doublewrite buffer */
UNIV_INTERN
ibool
trx_doublewrite_page_inside (
/*=========================*/
     ulint    page_no)   /*!< in: page number */
{
     if (trx_doublewrite == NULL) {

              return (FALSE) ;
     }
```

```
        if  (page_no >= trx_doublewrite->block1
            && page_no < trx_doublewrite->block1
            + TRX_SYS_DOUBLEWRITE_BLOCK_SIZE)  {
                return (TRUE) ;
        }

        if  (page_no >= trx_doublewrite->block2
            && page_no < trx_doublewrite->block2
            + TRX_SYS_DOUBLEWRITE_BLOCK_SIZE)  {
                return (TRUE) ;
        }

        return (FALSE) ;
}

/****************************************************************//**
Creates or initialializes the doublewrite buffer at a database start. */
static
void
trx_doublewrite_init (
/*==================*/
        byte*    doublewrite)       /*!< in: pointer to the doublewrite buf
                                    header on trx sys page */
{
        trx_doublewrite = mem_alloc (sizeof (trx_doublewrite_t)) ;

        /* Since we now start to use the doublewrite buffer, no need to call
        fsync ()  after every write to a data file */
#ifdef UNIV_DO_FLUSH
        os_do_not_call_flush_at_each_write = TRUE;
#endif /* UNIV_DO_FLUSH */

        mutex_create (&trx_doublewrite->mutex, SYNC_DOUBLEWRITE) ;

        trx_doublewrite->first_free = 0;

        trx_doublewrite->block1 = mach_read_from_4 (
                doublewrite + TRX_SYS_DOUBLEWRITE_BLOCK1) ;
        trx_doublewrite->block2 = mach_read_from_4 (
                doublewrite + TRX_SYS_DOUBLEWRITE_BLOCK2) ;
        trx_doublewrite->write_buf_unaligned = ut_malloc (
                (1 + 2 * TRX_SYS_DOUBLEWRITE_BLOCK_SIZE)  * UNIV_PAGE_SIZE) ;

        trx_doublewrite->write_buf = ut_align (
```

```
                trx_doublewrite->write_buf_unaligned, UNIV_PAGE_SIZE) ;
        trx_doublewrite->buf_block_arr = mem_alloc (
                2 * TRX_SYS_DOUBLEWRITE_BLOCK_SIZE * sizeof (void*)) ;
}
```

buf0flu.c

```
/************************************************************************//**
Flushes possible buffered writes from the doublewrite memory buffer to disk,
and also wakes up the aio thread if simulated aio is used. It is very
important to call this function after a batch of writes has been posted,
and also when we may have to wait for a page latch! Otherwise a deadlock
of threads can occur. */
static
void
buf_flush_buffered_writes (void)
/*============================*/
{
        byte*                   write_buf;
        ulint                   len;
        ulint                   len2;
        ulint                   i;

        if (!srv_use_doublewrite_buf || trx_doublewrite == NULL) {
                /* Sync the writes to the disk. */
                buf_flush_sync_datafiles () ;
                return;
        }

        mutex_enter (& (trx_doublewrite->mutex)) ;

        /* Write first to doublewrite buffer blocks. We use synchronous
        aio and thus know that file write has been completed when the
        control returns. */

        if (trx_doublewrite->first_free == 0) {

                mutex_exit (& (trx_doublewrite->mutex)) ;

                return;
        }

        for (i = 0; i < trx_doublewrite->first_free; i++) {

                const buf_block_t* block;
```

```
            block = (buf_block_t*) trx_doublewrite->buf_block_arr[i];

            if (buf_block_get_state (block) != BUF_BLOCK_FILE_PAGE
                || block->page.zip.data) {
                    /* No simple validate for compressed pages exists. */
                    continue;
            }

            if (UNIV_UNLIKELY
                 (memcmp (block->frame + (FIL_PAGE_LSN + 4) ,
                          block->frame + (UNIV_PAGE_SIZE
                                          - FIL_PAGE_END_LSN_OLD_CHKSUM + 4) ,
                          4))) {
                    ut_print_timestamp (stderr) ;
                    fprintf (stderr,
                            "InnoDB: ERROR: The page to be written"
                            "seems corrupt!\n"
                            "InnoDB: The lsn fields do not match!"
                            "Noticed in the buffer pool\n"
                            "InnoDB: before posting to the"
                            "doublewrite buffer.\n") ;
            }

            if (!block->check_index_page_at_flush) {
            } else if (page_is_comp (block->frame)) {
                    if (UNIV_UNLIKELY
                         (!page_simple_validate_new (block->frame))) {
corrupted_page:
                            buf_page_print (block->frame, 0) ;

                            ut_print_timestamp (stderr) ;
                            fprintf (stderr,
                                    "InnoDB: Apparent corruption of an"
                                    "index page n:o %lu in space %lu\n"
                                    "InnoDB: to be written to data file."
                                    "We intentionally crash server\n"
                                    "InnoDB: to prevent corrupt data"
                                    "from ending up in data\n"
                                    "InnoDB: files.\n",
                                    (ulong) buf_block_get_page_no (block) ,
                                    (ulong) buf_block_get_space (block)) ;

                            ut_error;
```

```
                    }
            } else if  (UNIV_UNLIKELY
                        (!page_simple_validate_old (block->frame)))  {

                    goto corrupted_page;
            }
    }

    /* increment the doublewrite flushed pages counter */
    srv_dblwr_pages_written+= trx_doublewrite->first_free;
    srv_dblwr_writes++;

    len = ut_min (TRX_SYS_DOUBLEWRITE_BLOCK_SIZE,
                  trx_doublewrite->first_free)  * UNIV_PAGE_SIZE;

    write_buf = trx_doublewrite->write_buf;
    i = 0;

    fil_io (OS_FILE_WRITE, TRUE, TRX_SYS_SPACE, 0,
           trx_doublewrite->block1, 0, len,
            (void*)  write_buf, NULL) ;

    for  (len2 = 0; len2 + UNIV_PAGE_SIZE <= len;
         len2 += UNIV_PAGE_SIZE, i++)  {
            const buf_block_t* block =  (buf_block_t*)
                        trx_doublewrite->buf_block_arr[i];

            if  (UNIV_LIKELY (!block->page.zip.data)
                && UNIV_LIKELY (buf_block_get_state (block)
                                == BUF_BLOCK_FILE_PAGE)
                && UNIV_UNLIKELY
                 (memcmp (write_buf + len2 +  (FIL_PAGE_LSN + 4) ,
                          write_buf + len2
                          +  (UNIV_PAGE_SIZE
                              - FIL_PAGE_END_LSN_OLD_CHKSUM + 4) , 4)))  {
                    ut_print_timestamp (stderr) ;
                    fprintf (stderr,
                            "InnoDB: ERROR: The page to be written"
                            "seems corrupt!\n"
                            "InnoDB: The lsn fields do not match!"
                            "Noticed in the doublewrite block1.\n") ;
            }
    }
```

```
        if  (trx_doublewrite->first_free <= TRX_SYS_DOUBLEWRITE_BLOCK_SIZE)  {
                goto flush;
        }

        len =  (trx_doublewrite->first_free - TRX_SYS_DOUBLEWRITE_BLOCK_SIZE)
                * UNIV_PAGE_SIZE;

        write_buf = trx_doublewrite->write_buf
                + TRX_SYS_DOUBLEWRITE_BLOCK_SIZE * UNIV_PAGE_SIZE;
        ut_ad (i == TRX_SYS_DOUBLEWRITE_BLOCK_SIZE) ;

        fil_io (OS_FILE_WRITE, TRUE, TRX_SYS_SPACE, 0,
               trx_doublewrite->block2, 0, len,
                (void*)  write_buf, NULL) ;

        for  (len2 = 0; len2 + UNIV_PAGE_SIZE <= len;
             len2 += UNIV_PAGE_SIZE, i++)  {
                const buf_block_t* block =  (buf_block_t*)
                        trx_doublewrite->buf_block_arr[i];

                if  (UNIV_LIKELY (!block->page.zip.data)
                    && UNIV_LIKELY (buf_block_get_state (block)
                                    == BUF_BLOCK_FILE_PAGE)
                    && UNIV_UNLIKELY
                     (memcmp (write_buf + len2 +  (FIL_PAGE_LSN + 4) ,
                              write_buf + len2
                              +  (UNIV_PAGE_SIZE
                                  - FIL_PAGE_END_LSN_OLD_CHKSUM + 4) , 4)))  {
                        ut_print_timestamp (stderr) ;
                        fprintf (stderr,
                                "InnoDB: ERROR: The page to be"
                                "written seems corrupt!\n"
                                "InnoDB: The lsn fields do not match!"
                                "Noticed in"
                                "the doublewrite block2.\n") ;
                }
        }

flush:
        /* Now flush the doublewrite buffer data to disk */

        fil_flush (TRX_SYS_SPACE) ;

        /* We know that the writes have been flushed to disk now
```

```
and in recovery we will find them in the doublewrite buffer
blocks. Next do the writes to the intended positions. */

for (i = 0; i < trx_doublewrite->first_free; i++) {
        const buf_block_t* block = (buf_block_t*)
                trx_doublewrite->buf_block_arr[i];

        ut_a (buf_page_in_file (&block->page)) ;
        if (UNIV_LIKELY_NULL (block->page.zip.data)) {
                fil_io (OS_FILE_WRITE | OS_AIO_SIMULATED_WAKE_LATER,
                        FALSE, buf_page_get_space (&block->page) ,
                        buf_page_get_zip_size (&block->page) ,
                        buf_page_get_page_no (&block->page) , 0,
                        buf_page_get_zip_size (&block->page) ,
                         (void*) block->page.zip.data,
                         (void*) block) ;

                /* Increment the counter of I/O operations used
                for selecting LRU policy. */
                buf_LRU_stat_inc_io () ;

                continue;
        }

        ut_a (buf_block_get_state (block) == BUF_BLOCK_FILE_PAGE) ;

        if (UNIV_UNLIKELY (memcmp (block->frame + (FIL_PAGE_LSN + 4) ,
                                   block->frame
                                   + (UNIV_PAGE_SIZE
                                      - FIL_PAGE_END_LSN_OLD_CHKSUM + 4) ,
                                   4))) {
                ut_print_timestamp (stderr) ;
                fprintf (stderr,
                        "InnoDB: ERROR: The page to be written"
                        "seems corrupt!\n"
                        "InnoDB: The lsn fields do not match!"
                        "Noticed in the buffer pool\n"
                        "InnoDB: after posting and flushing"
                        "the doublewrite buffer.\n"
                        "InnoDB: Page buf fix count %lu,"
                        "io fix %lu, state %lu\n",
                        (ulong) block->page.buf_fix_count,
                        (ulong) buf_block_get_io_fix (block) ,
                        (ulong) buf_block_get_state (block)) ;
```

```
                }

                fil_io (OS_FILE_WRITE | OS_AIO_SIMULATED_WAKE_LATER,
                        FALSE, buf_block_get_space (block) , 0,
                        buf_block_get_page_no (block) , 0, UNIV_PAGE_SIZE,
                         (void*) block->frame,  (void*) block) ;

                /* Increment the counter of I/O operations used
                for selecting LRU policy. */
                buf_LRU_stat_inc_io () ;
        }

        /* Sync the writes to the disk. */
        buf_flush_sync_datafiles () ;

        /* We can now reuse the doublewrite memory buffer: */
        trx_doublewrite->first_free = 0;

        mutex_exit (& (trx_doublewrite->mutex)) ;
}
```

附录D　哈希算法和哈希表源代码

hash0hash.h

```
/*****************************************************************************

Copyright (c) 1997, 2009, Innobase Oy. All Rights Reserved.

This program is free software; you can redistribute it and/or modify it under
the terms of the GNU General Public License as published by the Free Software
Foundation; version 2 of the License.

This program is distributed in the hope that it will be useful, but WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with
this program; if not, write to the Free Software Foundation, Inc., 59 Temple
Place, Suite 330, Boston, MA 02111-1307 USA

*****************************************************************************/

/********************************************************//**
@file include/hash0hash.h
The simple hash table utility

Created 5/20/1997 Heikki Tuuri
************************************************************/

#ifndef hash0hash_h
#define hash0hash_h

#include "univ.i"
#include "mem0mem.h"
#ifndef UNIV_HOTBACKUP
# include "sync0sync.h"
#endif /* !UNIV_HOTBACKUP */

typedef struct hash_table_struct hash_table_t;
```

```
typedef struct hash_cell_struct hash_cell_t;

typedef void* hash_node_t;

/* Fix Bug #13859: symbol collision between imap/mysql */
#define hash_create hash0_create

/*************************************************************//**
Creates a hash table with >= n array cells. The actual number
of cells is chosen to be a prime number slightly bigger than n.
@return       own: created table */
UNIV_INTERN
hash_table_t*
hash_create (
/*========*/
     ulint     n) ;        /*!< in: number of array cells */
#ifndef UNIV_HOTBACKUP
/*************************************************************//**
Creates a mutex array to protect a hash table. */
UNIV_INTERN
void
hash_create_mutexes_func (
/*======================*/
     hash_table_t*        table,              /*!< in: hash table */
#ifdef UNIV_SYNC_DEBUG
     ulint                sync_level,         /*!< in: latching order level of the
                                              mutexes: used in the debug version */
#endif /* UNIV_SYNC_DEBUG */
     ulint                n_mutexes) ;        /*!< in: number of mutexes */
#ifdef UNIV_SYNC_DEBUG
# define hash_create_mutexes (t,n,level)  hash_create_mutexes_func (t,level,n)
#else /* UNIV_SYNC_DEBUG */
# define hash_create_mutexes (t,n,level)  hash_create_mutexes_func (t,n)
#endif /* UNIV_SYNC_DEBUG */
#endif /* !UNIV_HOTBACKUP */

/*************************************************************//**
Frees a hash table. */
UNIV_INTERN
void
hash_table_free (
/*=============*/
     hash_table_t*        table) ;            /*!< in, own: hash table */
/*************************************************************//**
```

```
Calculates the hash value from a folded value.
@return        hashed value */
UNIV_INLINE
ulint
hash_calc_hash (
/*============*/
      ulint                fold,   /*!< in: folded value */
      hash_table_t*        table) ;             /*!< in: hash table */
#ifndef UNIV_HOTBACKUP
/************************************************************************//**
Assert that the mutex for the table in a hash operation is owned. */
# define HASH_ASSERT_OWNED (TABLE, FOLD)                                  \
ut_ad (! (TABLE) ->mutexes || mutex_own (hash_get_mutex (TABLE, FOLD))) ;
#else /* !UNIV_HOTBACKUP */
# define HASH_ASSERT_OWNED (TABLE, FOLD)
#endif /* !UNIV_HOTBACKUP */

/************************************************************************//**
Inserts a struct to a hash table. */

#define HASH_INSERT (TYPE, NAME, TABLE, FOLD, DATA) \
do {\
      hash_cell_t*         cell3333;\
      TYPE*                struct3333;\
\
      HASH_ASSERT_OWNED (TABLE, FOLD) \
\
      (DATA) ->NAME = NULL;\
\
      cell3333 = hash_get_nth_cell (TABLE, hash_calc_hash (FOLD, TABLE)) ;\
\
      if  (cell3333->node == NULL)  {\
                cell3333->node = DATA;\
      } else {\
                struct3333 =  (TYPE*)  cell3333->node;\
\
                while  (struct3333->NAME != NULL)  {\
\
                          struct3333 =  (TYPE*)  struct3333->NAME;\
                }\
\
                struct3333->NAME = DATA;\
      }\
} while  (0)
```

```
#ifdef UNIV_HASH_DEBUG
# define HASH_ASSERT_VALID (DATA) ut_a ((void*) (DATA) != (void*) -1)
# define HASH_INVALIDATE (DATA, NAME) DATA->NAME = (void*) -1
#else
# define HASH_ASSERT_VALID (DATA) do {} while (0)
# define HASH_INVALIDATE (DATA, NAME) do {} while (0)
#endif

/*************************************************************//**
Deletes a struct from a hash table. */

#define HASH_DELETE (TYPE, NAME, TABLE, FOLD, DATA) \
do {\
     hash_cell_t*         cell3333;\
     TYPE*                struct3333;\
\
     HASH_ASSERT_OWNED (TABLE, FOLD) \
\
     cell3333 = hash_get_nth_cell (TABLE, hash_calc_hash (FOLD, TABLE)) ;\
\
     if (cell3333->node == DATA) {\
               HASH_ASSERT_VALID (DATA->NAME) ;\
               cell3333->node = DATA->NAME;\
     } else {\
               struct3333 = (TYPE*) cell3333->node;\
\
               while (struct3333->NAME != DATA) {\
\
                         struct3333 = (TYPE*) struct3333->NAME;\
                         ut_a (struct3333) ;\
               }\
\
               struct3333->NAME = DATA->NAME;\
     }\
     HASH_INVALIDATE (DATA, NAME) ;\
} while (0)

/*************************************************************//**
Gets the first struct in a hash chain, NULL if none. */

#define HASH_GET_FIRST (TABLE, HASH_VAL) \
     (hash_get_nth_cell (TABLE, HASH_VAL) ->node)

/*************************************************************//**
```

```
Gets the next struct in a hash chain, NULL if none. */

#define HASH_GET_NEXT (NAME, DATA)              ((DATA) ->NAME)

/*************************************************************************//**
Looks for a struct in a hash table. */
#define HASH_SEARCH (NAME, TABLE, FOLD, TYPE, DATA, ASSERTION, TEST) \
{\
\
     HASH_ASSERT_OWNED (TABLE, FOLD) \
\
     (DATA) = (TYPE) HASH_GET_FIRST (TABLE, hash_calc_hash (FOLD, TABLE)) ;\
     HASH_ASSERT_VALID (DATA) ;\
\
     while ((DATA) != NULL) {\
               ASSERTION;\
               if (TEST) {\
                         break;\
               } else {\
                         HASH_ASSERT_VALID (HASH_GET_NEXT (NAME, DATA)) ;\
                         (DATA) = (TYPE) HASH_GET_NEXT (NAME, DATA) ;\
               }\
     }\
}

/*************************************************************************//**
Looks for an item in all hash buckets. */
#define HASH_SEARCH_ALL (NAME, TABLE, TYPE, DATA, ASSERTION, TEST)          \
do {                                                                        \
     ulint    i3333;                                                       \
                                                                           \
     for (i3333 = (TABLE) ->n_cells; i3333--; ) {                           \
               (DATA) = (TYPE) HASH_GET_FIRST (TABLE, i3333) ;              \
                                                                           \
               while ((DATA) != NULL) {                                    \
                         HASH_ASSERT_VALID (DATA) ;                        \
                         ASSERTION;                                        \
                                                                           \
                         if (TEST) {                                       \
                                 break;                                    \
                         }                                                 \
                                                                           \
                         (DATA) = (TYPE) HASH_GET_NEXT (NAME, DATA) ;      \
               }                                                           \
```

```
                                                                    \
            if  ((DATA)  != NULL)  {                                \
                        break;                                      \
            }                                                       \
    }                                                               \
} while  (0)

/************************************************************//**
Gets the nth cell in a hash table.
@return       pointer to cell */
UNIV_INLINE
hash_cell_t*
hash_get_nth_cell (
/*==============*/
    hash_table_t*       table,  /*!< in: hash table */
    ulint               n) ;    /*!< in: cell index */

/************************************************************//**
Clears a hash table so that all the cells become empty. */
UNIV_INLINE
void
hash_table_clear (
/*=============*/
    hash_table_t*       table) ;            /*!< in/out: hash table */

/************************************************************//**
Returns the number of cells in a hash table.
@return       number of cells */
UNIV_INLINE
ulint
hash_get_n_cells (
/*=============*/
    hash_table_t*       table) ;            /*!< in: table */
/*******************************************************************//**
Deletes a struct which is stored in the heap of the hash table, and compacts
the heap. The fold value must be stored in the struct NODE in a field named
'fold'. */

#define HASH_DELETE_AND_COMPACT (TYPE, NAME, TABLE, NODE) \
do {\
    TYPE*               node111;\
    TYPE*               top_node111;\
    hash_cell_t*        cell1111;\
    ulint               fold111;\
```

```
\
    fold111 =  (NODE) ->fold;\
\
    HASH_DELETE (TYPE, NAME, TABLE, fold111, NODE) ;\
\
    top_node111 =  (TYPE*) mem_heap_get_top (\
                              hash_get_heap (TABLE, fold111) ,\
                                                  sizeof (TYPE)) ;\
\
    /* If the node to remove is not the top node in the heap, compact the\
    heap of nodes by moving the top node in the place of NODE. */\
\
    if  (NODE != top_node111)  {\
\
            /* Copy the top node in place of NODE */\
\
            * (NODE)  = *top_node111;\
\
            cell1111 = hash_get_nth_cell (TABLE,\
                              hash_calc_hash (top_node111->fold, TABLE)) ;\
\
            /* Look for the pointer to the top node, to update it */\
\
            if  (cell1111->node == top_node111)  {\
                        /* The top node is the first in the chain */\
\
                        cell1111->node = NODE;\
            } else {\
                        /* We have to look for the predecessor of the top\
                        node */\
                        node111 = cell1111->node;\
\
                        while  (top_node111 != HASH_GET_NEXT (NAME, node111))
{\
\
                                node111 = HASH_GET_NEXT (NAME, node111) ;\
                        }\
\
                        /* Now we have the predecessor node */\
\
                        node111->NAME = NODE;\
            }\
    }\
\
```

```
     /* Free the space occupied by the top node */\
\
     mem_heap_free_top (hash_get_heap (TABLE, fold111) , sizeof (TYPE)) ;\
} while (0)

#ifndef UNIV_HOTBACKUP
/************************************************************//**
Move all hash table entries from OLD_TABLE to NEW_TABLE. */

#define HASH_MIGRATE (OLD_TABLE, NEW_TABLE, NODE_TYPE, PTR_NAME, FOLD_FUNC)  \
do {\
     ulint               i2222;\
     ulint               cell_count2222;\
\
     cell_count2222 = hash_get_n_cells (OLD_TABLE) ;\
\
     for (i2222 = 0; i2222 < cell_count2222; i2222++) {\
          NODE_TYPE* node2222 = HASH_GET_FIRST ((OLD_TABLE) , i2222) ;\
\
          while (node2222) {\
               NODE_TYPE*          next2222 = node2222->PTR_NAME;\
               ulint               fold2222 = FOLD_FUNC (node2222) ;\
\
               HASH_INSERT (NODE_TYPE, PTR_NAME, (NEW_TABLE) ,\
                    fold2222, node2222) ;\
\
               node2222 = next2222;\
          }\
     }\
} while (0)

/************************************************************//**
Gets the mutex index for a fold value in a hash table.
@return        mutex number */
UNIV_INLINE
ulint
hash_get_mutex_no (
/*==============*/
     hash_table_t*       table,  /*!< in: hash table */
     ulint               fold) ; /*!< in: fold */
/************************************************************//**
Gets the nth heap in a hash table.
@return        mem heap */
UNIV_INLINE
```

```
mem_heap_t*
hash_get_nth_heap (
/*===============*/
      hash_table_t*        table,  /*!< in: hash table */
      ulint                i) ;    /*!< in: index of the heap */
/*****************************************************************//**
Gets the heap for a fold value in a hash table.
@return        mem heap */
UNIV_INLINE
mem_heap_t*
hash_get_heap (
/*===========*/
      hash_table_t*        table,  /*!< in: hash table */
      ulint                fold) ; /*!< in: fold */
/*****************************************************************//**
Gets the nth mutex in a hash table.
@return        mutex */
UNIV_INLINE
mutex_t*
hash_get_nth_mutex (
/*================*/
      hash_table_t*        table,  /*!< in: hash table */
      ulint                i) ;    /*!< in: index of the mutex */
/*****************************************************************//**
Gets the mutex for a fold value in a hash table.
@return        mutex */
UNIV_INLINE
mutex_t*
hash_get_mutex (
/*============*/
      hash_table_t*        table,  /*!< in: hash table */
      ulint                fold) ; /*!< in: fold */
/*****************************************************************//**
Reserves the mutex for a fold value in a hash table. */
UNIV_INTERN
void
hash_mutex_enter (
/*==============*/
      hash_table_t*        table,  /*!< in: hash table */
      ulint                fold) ; /*!< in: fold */
/*****************************************************************//**
Releases the mutex for a fold value in a hash table. */
UNIV_INTERN
void
```

```
hash_mutex_exit (
/*=============*/
     hash_table_t*        table,  /*!< in: hash table */
     ulint                fold) ; /*!< in: fold */
/************************************************************//**
Reserves all the mutexes of a hash table, in an ascending order. */
UNIV_INTERN
void
hash_mutex_enter_all (
/*==================*/
     hash_table_t*        table) ;             /*!< in: hash table */
/************************************************************//**
Releases all the mutexes of a hash table. */
UNIV_INTERN
void
hash_mutex_exit_all (
/*=================*/
     hash_table_t*        table) ;             /*!< in: hash table */
#else /* !UNIV_HOTBACKUP */
# define hash_get_heap (table, fold)          ((table) ->heap)
# define hash_mutex_enter (table, fold)       ((void)  0)
# define hash_mutex_exit (table, fold)        ((void)  0)
#endif /* !UNIV_HOTBACKUP */

struct hash_cell_struct{
     void*     node;       /*!< hash chain node, NULL if none */
};

/* The hash table structure */
struct hash_table_struct {
#if defined UNIV_AHI_DEBUG || defined UNIV_DEBUG
# ifndef UNIV_HOTBACKUP
     ibool                adaptive;/* TRUE if this is the hash table of the
                                  adaptive hash index */
# endif /* !UNIV_HOTBACKUP */
#endif /* UNIV_AHI_DEBUG || UNIV_DEBUG */
     ulint                n_cells;/* number of cells in the hash table */
     hash_cell_t*         array;  /*!< pointer to cell array */
#ifndef UNIV_HOTBACKUP
     ulint                n_mutexes;/* if mutexes != NULL, then the number of
                                  mutexes, must be a power of 2 */
     mutex_t* mutexes;/* NULL, or an array of mutexes used to
                                  protect segments of the hash table */
     mem_heap_t**         heaps;  /*!< if this is non-NULL, hash chain nodes for
```

```
                                external chaining can be allocated from these
                                memory heaps; there are then n_mutexes many of
                                these heaps */
#endif /* !UNIV_HOTBACKUP */
    mem_heap_t*         heap;
    ulint               magic_n;
};

#define HASH_TABLE_MAGIC_N      76561114

#ifndef UNIV_NONINL
#include "hash0hash.ic"
#endif

#endif
```

ha0ha.h

```
/*****************************************************************************

Copyright (c) 1994, 2009, Innobase Oy. All Rights Reserved.

This program is free software; you can redistribute it and/or modify it under
the terms of the GNU General Public License as published by the Free Software
Foundation; version 2 of the License.

This program is distributed in the hope that it will be useful, but WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with
this program; if not, write to the Free Software Foundation, Inc., 59 Temple
Place, Suite 330, Boston, MA 02111-1307 USA

*****************************************************************************/

/**************************************************//**
@file include/ha0ha.h
The hash table with external chains

Created 8/18/1994 Heikki Tuuri
*******************************************************/

#ifndef ha0ha_h
#define ha0ha_h
```

```
#include "univ.i"

#include "hash0hash.h"
#include "page0types.h"
#include "buf0types.h"

/*****************************************************************//**
Looks for an element in a hash table.
@return pointer to the data of the first hash table node in chain
having the fold number, NULL if not found */
UNIV_INLINE
void*
ha_search_and_get_data (
/*====================*/
      hash_table_t*        table,  /*!< in: hash table */
      ulint                fold) ; /*!< in: folded value of the searched data */
/*****************************************************************//**
Looks for an element when we know the pointer to the data and updates
the pointer to data if found. */
UNIV_INTERN
void
ha_search_and_update_if_found_func (
/*===============================*/
      hash_table_t*        table,  /*!< in/out: hash table */
      ulint                fold,   /*!< in: folded value of the searched data */
      void*                data,   /*!< in: pointer to the data */
#if defined UNIV_AHI_DEBUG || defined UNIV_DEBUG
      buf_block_t*         new_block,/*!< in: block containing new_data */
#endif /* UNIV_AHI_DEBUG || UNIV_DEBUG */
      void*                new_data) ;/*!< in: new pointer to the data */

#if defined UNIV_AHI_DEBUG || defined UNIV_DEBUG
/** Looks for an element when we know the pointer to the data and
updates the pointer to data if found.
@param table              in/out: hash table
@param fold               in: folded value of the searched data
@param data               in: pointer to the data
@param new_block          in: block containing new_data
@param new_data                   in: new pointer to the data */
# define ha_search_and_update_if_found (table,fold,data,new_block,new_data)  \
      ha_search_and_update_if_found_func (table,fold,data,new_block,new_data)
#else /* UNIV_AHI_DEBUG || UNIV_DEBUG */
/** Looks for an element when we know the pointer to the data and
updates the pointer to data if found.
```

```
@param table            in/out: hash table
@param fold             in: folded value of the searched data
@param data             in: pointer to the data
@param new_block        ignored: block containing new_data
@param new_data                 in: new pointer to the data */
# define ha_search_and_update_if_found (table,fold,data,new_block,new_data)  \
     ha_search_and_update_if_found_func (table,fold,data,new_data)
#endif /* UNIV_AHI_DEBUG || UNIV_DEBUG */
/*****************************************************************//**
Creates a hash table with at least n array cells.  The actual number
of cells is chosen to be a prime number slightly bigger than n.
@return       own: created table */
UNIV_INTERN
hash_table_t*
ha_create_func (
/*============*/
     ulint    n,                /*!< in: number of array cells */
#ifdef UNIV_SYNC_DEBUG
     ulint    mutex_level,      /*!< in: level of the mutexes in the latching
                                order: this is used in the debug version */
#endif /* UNIV_SYNC_DEBUG */
     ulint    n_mutexes) ;      /*!< in: number of mutexes to protect the
                                hash table: must be a power of 2, or 0 */
#ifdef UNIV_SYNC_DEBUG
/** Creates a hash table.
@return                  own: created table
@param n_c    in: number of array cells.  The actual number of cells is
chosen to be a slightly bigger prime number.
@param level  in: level of the mutexes in the latching order
@param n_m    in: number of mutexes to protect the hash table;
              must be a power of 2, or 0 */
# define ha_create (n_c,n_m,level)  ha_create_func (n_c,level,n_m)
#else /* UNIV_SYNC_DEBUG */
/** Creates a hash table.
@return                  own: created table
@param n_c    in: number of array cells.  The actual number of cells is
chosen to be a slightly bigger prime number.
@param level  in: level of the mutexes in the latching order
@param n_m    in: number of mutexes to protect the hash table;
              must be a power of 2, or 0 */
# define ha_create (n_c,n_m,level)  ha_create_func (n_c,n_m)
#endif /* UNIV_SYNC_DEBUG */

/*****************************************************************//**
```

```
Empties a hash table and frees the memory heaps. */
UNIV_INTERN
void
ha_clear (
/*======*/
     hash_table_t*        table) ;              /*!< in, own: hash table */

/*************************************************************//**
Inserts an entry into a hash table. If an entry with the same fold number
is found, its node is updated to point to the new data, and no new node
is inserted.
@return        TRUE if succeed, FALSE if no more memory could be allocated */
UNIV_INTERN
ibool
ha_insert_for_fold_func (
/*====================*/
     hash_table_t*        table,  /*!< in: hash table */
     ulint                fold,   /*!< in: folded value of data; if a node with
                                  the same fold value already exists, it is
                                  updated to point to the same data, and no new
                                  node is created! */
#if defined UNIV_AHI_DEBUG || defined UNIV_DEBUG
     buf_block_t*         block,  /*!< in: buffer block containing the data */
#endif /* UNIV_AHI_DEBUG || UNIV_DEBUG */
     void*                data) ; /*!< in: data, must not be NULL */

#if defined UNIV_AHI_DEBUG || defined UNIV_DEBUG
/**
Inserts an entry into a hash table. If an entry with the same fold number
is found, its node is updated to point to the new data, and no new node
is inserted.
@return        TRUE if succeed, FALSE if no more memory could be allocated
@param t       in: hash table
@param f       in: folded value of data
@param b       in: buffer block containing the data
@param d       in: data, must not be NULL */
# define ha_insert_for_fold (t,f,b,d)  ha_insert_for_fold_func (t,f,b,d)
#else /* UNIV_AHI_DEBUG || UNIV_DEBUG */
/**
Inserts an entry into a hash table. If an entry with the same fold number
is found, its node is updated to point to the new data, and no new node
is inserted.
@return        TRUE if succeed, FALSE if no more memory could be allocated
@param t       in: hash table
```

```
@param f        in: folded value of data
@param b        ignored: buffer block containing the data
@param d        in: data, must not be NULL */
# define ha_insert_for_fold (t,f,b,d)  ha_insert_for_fold_func (t,f,d)
#endif /* UNIV_AHI_DEBUG || UNIV_DEBUG */

/*************************************************************//**
Looks for an element when we know the pointer to the data and deletes
it from the hash table if found.
@return         TRUE if found */
UNIV_INLINE
ibool
ha_search_and_delete_if_found (
/*===========================*/
      hash_table_t*         table,  /*!< in: hash table */
      ulint                 fold,   /*!< in: folded value of the searched data */
      void*                 data) ; /*!< in: pointer to the data */
#ifndef UNIV_HOTBACKUP
/*****************************************************************//**
Removes from the chain determined by fold all nodes whose data pointer
points to the page given. */
UNIV_INTERN
void
ha_remove_all_nodes_to_page (
/*=========================*/
      hash_table_t*         table,  /*!< in: hash table */
      ulint                 fold,   /*!< in: fold value */
      const page_t*         page) ; /*!< in: buffer page */
/*************************************************************//**
Validates a given range of the cells in hash table.
@return         TRUE if ok */
UNIV_INTERN
ibool
ha_validate (
/*=========*/
      hash_table_t*         table,                /*!< in: hash table */
      ulint                 start_index,          /*!< in: start index */
      ulint                 end_index) ;          /*!< in: end index */
/*************************************************************//**
Prints info of a hash table. */
UNIV_INTERN
void
ha_print_info (
/*===========*/
```

```
        FILE*                   file,   /*!< in: file where to print */
        hash_table_t*           table) ;            /*!< in: hash table */
#endif /* !UNIV_HOTBACKUP */

/** The hash table external chain node */
typedef struct ha_node_struct ha_node_t;

/** The hash table external chain node */
struct ha_node_struct {
        ha_node_t*              next;   /*!< next chain node or NULL if none */
#if defined UNIV_AHI_DEBUG || defined UNIV_DEBUG
        buf_block_t*            block;  /*!< buffer block containing the data, or NULL */
#endif /* UNIV_AHI_DEBUG || UNIV_DEBUG */
        void*                   data;   /*!< pointer to the data */
        ulint                   fold;   /*!< fold value for the data */
};

#ifndef UNIV_HOTBACKUP
/** Assert that the current thread is holding the mutex protecting a
hash bucket corresponding to a fold value.
@param table   in: hash table
@param fold    in: fold value */
# define ASSERT_HASH_MUTEX_OWN (table, fold)
        ut_ad (! (table) ->mutexes || mutex_own (hash_get_mutex (table, fold)))
#else /* !UNIV_HOTBACKUP */
/** Assert that the current thread is holding the mutex protecting a
hash bucket corresponding to a fold value.
@param table   in: hash table
@param fold    in: fold value */
# define ASSERT_HASH_MUTEX_OWN (table, fold)   ((void)  0)
#endif /* !UNIV_HOTBACKUP */

#ifndef UNIV_NONINL
#include "ha0ha.ic"
#endif

#endif
```

ha0ha.ic

```
/***********************************************************************

Copyright (c) 1994, 2009, Innobase Oy. All Rights Reserved.

This program is free software; you can redistribute it and/or modify it under
```

```
the terms of the GNU General Public License as published by the Free Software
Foundation; version 2 of the License.

This program is distributed in the hope that it will be useful, but WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with
this program; if not, write to the Free Software Foundation, Inc., 59 Temple
Place, Suite 330, Boston, MA 02111-1307 USA

*****************************************************************************/

/*********************************************************************//**
@file include/ha0ha.ic
The hash table with external chains

Created 8/18/1994 Heikki Tuuri
*************************************************************************/

#include "ut0rnd.h"
#include "mem0mem.h"

/*****************************************************************//**
Deletes a hash node. */
UNIV_INTERN
void
ha_delete_hash_node (
/*=================*/
     hash_table_t*       table,              /*!< in: hash table */
     ha_node_t*          del_node) ;         /*!< in: node to be deleted */

/*****************************************************************//**
Gets a hash node data.
@return       pointer to the data */
UNIV_INLINE
void*
ha_node_get_data (
/*==============*/
     ha_node_t*          node)   /*!< in: hash chain node */
{
     return (node->data) ;
}
```

```
/************************************************************************//**
Sets hash node data. */
UNIV_INLINE
void
ha_node_set_data_func (
/*===================*/
      ha_node_t*            node,   /*!< in: hash chain node */
#if defined UNIV_AHI_DEBUG || defined UNIV_DEBUG
      buf_block_t*          block,  /*!< in: buffer block containing the data */
#endif /* UNIV_AHI_DEBUG || UNIV_DEBUG */
      void*                 data)   /*!< in: pointer to the data */
{
#if defined UNIV_AHI_DEBUG || defined UNIV_DEBUG
      node->block = block;
#endif /* UNIV_AHI_DEBUG || UNIV_DEBUG */
      node->data = data;
}

#if defined UNIV_AHI_DEBUG || defined UNIV_DEBUG
/** Sets hash node data.
@param n        in: hash chain node
@param b        in: buffer block containing the data
@param d        in: pointer to the data */
# define ha_node_set_data (n,b,d)  ha_node_set_data_func (n,b,d)
#else /* UNIV_AHI_DEBUG || UNIV_DEBUG */
/** Sets hash node data.
@param n        in: hash chain node
@param b        in: buffer block containing the data
@param d        in: pointer to the data */
# define ha_node_set_data (n,b,d)  ha_node_set_data_func (n,d)
#endif /* UNIV_AHI_DEBUG || UNIV_DEBUG */

/************************************************************************//**
Gets the next node in a hash chain.
@return         next node, NULL if none */
UNIV_INLINE
ha_node_t*
ha_chain_get_next (
/*===============*/
      ha_node_t*            node)   /*!< in: hash chain node */
{
      return (node->next) ;
}
```

```
/*****************************************************************//**
Gets the first node in a hash chain.
@return       first node, NULL if none */
UNIV_INLINE
ha_node_t*
ha_chain_get_first (
/*================*/
     hash_table_t*        table,  /*!< in: hash table */
     ulint                fold)   /*!< in: fold value determining the chain */
{
     return ((ha_node_t*)
            hash_get_nth_cell (table, hash_calc_hash (fold, table)) ->node) ;
}

/*************************************************************//**
Looks for an element in a hash table.
@return pointer to the first hash table node in chain having the fold
number, NULL if not found */
UNIV_INLINE
ha_node_t*
ha_search (
/*======*/
     hash_table_t*        table,  /*!< in: hash table */
     ulint                fold)   /*!< in: folded value of the searched data */
{
     ha_node_t*           node;

     ASSERT_HASH_MUTEX_OWN (table, fold) ;

     node = ha_chain_get_first (table, fold) ;

     while  (node)  {
               if  (node->fold == fold)  {

                         return (node) ;
               }

               node = ha_chain_get_next (node) ;
     }

     return (NULL) ;
}

/*************************************************************//**
```

```
Looks for an element in a hash table.
@return pointer to the data of the first hash table node in chain
having the fold number, NULL if not found */
UNIV_INLINE
void*
ha_search_and_get_data (
/*====================*/
        hash_table_t*           table,  /*!< in: hash table */
        ulint                   fold)   /*!< in: folded value of the searched data */
{
        ha_node_t*              node;

        ASSERT_HASH_MUTEX_OWN (table, fold) ;

        node = ha_chain_get_first (table, fold) ;

        while  (node)  {
                if  (node->fold == fold)  {

                        return (node->data) ;
                }

                node = ha_chain_get_next (node) ;
        }

        return (NULL) ;
}

/*************************************************************//**
Looks for an element when we know the pointer to the data.
@return         pointer to the hash table node, NULL if not found in the table */
UNIV_INLINE
ha_node_t*
ha_search_with_data (
/*================*/
        hash_table_t*           table,  /*!< in: hash table */
        ulint                   fold,   /*!< in: folded value of the searched data */
        void*                   data)   /*!< in: pointer to the data */
{
        ha_node_t*              node;

        ASSERT_HASH_MUTEX_OWN (table, fold) ;

        node = ha_chain_get_first (table, fold) ;
```

```
        while (node) {
                if (node->data == data) {

                        return (node) ;
                }

                node = ha_chain_get_next (node) ;
        }

        return (NULL) ;
}

/*****************************************************//**
Looks for an element when we know the pointer to the data, and deletes
it from the hash table, if found.
@return         TRUE if found */
UNIV_INLINE
ibool
ha_search_and_delete_if_found (
/*===========================*/
        hash_table_t*           table,  /*!< in: hash table */
        ulint                   fold,   /*!< in: folded value of the searched data */
        void*                   data)   /*!< in: pointer to the data */
{
        ha_node_t*              node;

        ASSERT_HASH_MUTEX_OWN (table, fold) ;

        node = ha_search_with_data (table, fold, data) ;

        if (node) {
                ha_delete_hash_node (table, node) ;

                return (TRUE) ;
        }

        return (FALSE) ;
}
```

ha0ha.c

```
/*****************************************************************************

Copyright (c) 1994, 2009, Innobase Oy. All Rights Reserved.
```

```
This program is free software; you can redistribute it and/or modify it under
the terms of the GNU General Public License as published by the Free Software
Foundation; version 2 of the License.

This program is distributed in the hope that it will be useful, but WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with
this program; if not, write to the Free Software Foundation, Inc., 59 Temple
Place, Suite 330, Boston, MA 02111-1307 USA

*****************************************************************************/

/***********************************************************************//**
@file ha/ha0ha.c
The hash table with external chains

Created 8/22/1994 Heikki Tuuri
*************************************************************************/

#include "ha0ha.h"
#ifdef UNIV_NONINL
#include "ha0ha.ic"
#endif

#ifdef UNIV_DEBUG
# include "buf0buf.h"
#endif /* UNIV_DEBUG */
#ifdef UNIV_SYNC_DEBUG
# include "btr0sea.h"
#endif /* UNIV_SYNC_DEBUG */
#include "page0page.h"

/*************************************************************//**
Creates a hash table with at least n array cells.  The actual number
of cells is chosen to be a prime number slightly bigger than n.
@return       own: created table */
UNIV_INTERN
hash_table_t*
ha_create_func (
/*============*/
     ulint    n,                 /*!< in: number of array cells */
#ifdef UNIV_SYNC_DEBUG
```

```
        ulint     mutex_level,      /*!< in: level of the mutexes in the latching
                                    order: this is used in the debug version */
#endif /* UNIV_SYNC_DEBUG */
        ulint     n_mutexes) /*!< in: number of mutexes to protect the
                                    hash table: must be a power of 2, or 0 */
{
        hash_table_t*       table;
#ifndef UNIV_HOTBACKUP
        ulint               i;
#endif /* !UNIV_HOTBACKUP */

        ut_ad (ut_is_2pow (n_mutexes)) ;
        table = hash_create (n) ;

#if defined UNIV_AHI_DEBUG || defined UNIV_DEBUG
# ifndef UNIV_HOTBACKUP
        table->adaptive = TRUE;
# endif /* !UNIV_HOTBACKUP */
#endif /* UNIV_AHI_DEBUG || UNIV_DEBUG */
        /* Creating MEM_HEAP_BTR_SEARCH type heaps can potentially fail,
        but in practise it never should in this case, hence the asserts. */

        if  (n_mutexes == 0)  {
                table->heap = mem_heap_create_in_btr_search (
                                ut_min (4096, MEM_MAX_ALLOC_IN_BUF)) ;
                ut_a (table->heap) ;

                return (table) ;
        }

#ifndef UNIV_HOTBACKUP
        hash_create_mutexes (table, n_mutexes, mutex_level) ;

        table->heaps = mem_alloc (n_mutexes * sizeof (void*)) ;

        for  (i = 0; i < n_mutexes; i++)  {
                table->heaps[i] = mem_heap_create_in_btr_search (4096) ;
                ut_a (table->heaps[i]) ;
        }
#endif /* !UNIV_HOTBACKUP */

        return (table) ;
}
```

```
/************************************************************//**
Empties a hash table and frees the memory heaps. */
UNIV_INTERN
void
ha_clear (
/*======*/
     hash_table_t*        table)  /*!< in, own: hash table */
{
     ulint     i;
     ulint     n;

#ifdef UNIV_SYNC_DEBUG
     ut_ad (rw_lock_own (&btr_search_latch, RW_LOCK_EXCLUSIVE)) ;
#endif /* UNIV_SYNC_DEBUG */

#ifndef UNIV_HOTBACKUP
     /* Free the memory heaps. */
     n = table->n_mutexes;

     for  (i = 0; i < n; i++)  {
               mem_heap_free (table->heaps[i]) ;
     }
#endif /* !UNIV_HOTBACKUP */

     /* Clear the hash table. */
     n = hash_get_n_cells (table) ;

     for  (i = 0; i < n; i++)  {
               hash_get_nth_cell (table, i) ->node = NULL;
     }
}

/************************************************************//**
Inserts an entry into a hash table. If an entry with the same fold number
is found, its node is updated to point to the new data, and no new node
is inserted.
@return        TRUE if succeed, FALSE if no more memory could be allocated */
UNIV_INTERN
ibool
ha_insert_for_fold_func (
/*====================*/
     hash_table_t*        table,  /*!< in: hash table */
     ulint                fold,   /*!< in: folded value of data; if a node with
                                  the same fold value already exists, it is
```

```
                                updated to point to the same data, and no new
                                node is created! */
#if defined UNIV_AHI_DEBUG || defined UNIV_DEBUG
        buf_block_t*            block,  /*!< in: buffer block containing the data */
#endif /* UNIV_AHI_DEBUG || UNIV_DEBUG */
        void*                   data)   /*!< in: data, must not be NULL */
{
        hash_cell_t*            cell;
        ha_node_t*              node;
        ha_node_t*              prev_node;
        ulint                   hash;

        ut_ad (table && data) ;
#if defined UNIV_AHI_DEBUG || defined UNIV_DEBUG
        ut_a (block->frame == page_align (data)) ;
#endif /* UNIV_AHI_DEBUG || UNIV_DEBUG */
        ASSERT_HASH_MUTEX_OWN (table, fold) ;

        hash = hash_calc_hash (fold, table) ;

        cell = hash_get_nth_cell (table, hash) ;

        prev_node = cell->node;

        while  (prev_node != NULL)  {
                if  (prev_node->fold == fold)  {
#if defined UNIV_AHI_DEBUG || defined UNIV_DEBUG
# ifndef UNIV_HOTBACKUP
                        if  (table->adaptive)  {
                                buf_block_t* prev_block = prev_node->block;
                                ut_a (prev_block->frame
                                      == page_align (prev_node->data)) ;
                                ut_a (prev_block->n_pointers > 0) ;
                                prev_block->n_pointers--;
                                block->n_pointers++;
                        }
# endif /* !UNIV_HOTBACKUP */

                        prev_node->block = block;
#endif /* UNIV_AHI_DEBUG || UNIV_DEBUG */
                        prev_node->data = data;

                        return (TRUE) ;
                }
```

```
                prev_node = prev_node->next;
        }

        /* We have to allocate a new chain node */

        node = mem_heap_alloc (hash_get_heap (table, fold) , sizeof (ha_node_t)) ;

        if  (node == NULL)  {
                /* It was a btr search type memory heap and at the moment
                no more memory could be allocated: return */

                ut_ad (hash_get_heap (table, fold) ->type & MEM_HEAP_BTR_SEARCH) ;

                return (FALSE) ;
        }

        ha_node_set_data (node, block, data) ;

#if defined UNIV_AHI_DEBUG || defined UNIV_DEBUG
# ifndef UNIV_HOTBACKUP
        if  (table->adaptive)  {
                block->n_pointers++;
        }
# endif /* !UNIV_HOTBACKUP */
#endif /* UNIV_AHI_DEBUG || UNIV_DEBUG */

        node->fold = fold;

        node->next = NULL;

        prev_node = cell->node;

        if  (prev_node == NULL)  {

                cell->node = node;

                return (TRUE) ;
        }

        while  (prev_node->next != NULL)  {

                prev_node = prev_node->next;
        }
```

```
	prev_node->next = node;

	return (TRUE) ;
}

/*****************************************************************//**
Deletes a hash node. */
UNIV_INTERN
void
ha_delete_hash_node (
/*=================*/
	hash_table_t*		table,		/*!< in: hash table */
	ha_node_t*		del_node)	/*!< in: node to be deleted */
{
#if defined UNIV_AHI_DEBUG || defined UNIV_DEBUG
# ifndef UNIV_HOTBACKUP
	if (table->adaptive) {
		ut_a (del_node->block->frame = page_align (del_node->data)) ;
		ut_a (del_node->block->n_pointers > 0) ;
		del_node->block->n_pointers--;
	}
# endif /* !UNIV_HOTBACKUP */
#endif /* UNIV_AHI_DEBUG || UNIV_DEBUG */

	HASH_DELETE_AND_COMPACT (ha_node_t, next, table, del_node) ;
}

/*****************************************************************//**
Looks for an element when we know the pointer to the data, and updates
the pointer to data, if found. */
UNIV_INTERN
void
ha_search_and_update_if_found_func (
/*===============================*/
	hash_table_t*		table,	/*!< in/out: hash table */
	ulint			fold,	/*!< in: folded value of the searched data */
	void*			data,	/*!< in: pointer to the data */
#if defined UNIV_AHI_DEBUG || defined UNIV_DEBUG
	buf_block_t*		new_block,/*!< in: block containing new_data */
#endif /* UNIV_AHI_DEBUG || UNIV_DEBUG */
	void*			new_data) /*!< in: new pointer to the data */
{
	ha_node_t*		node;
```

```
        ASSERT_HASH_MUTEX_OWN (table, fold) ;
#if defined UNIV_AHI_DEBUG || defined UNIV_DEBUG
        ut_a (new_block->frame == page_align (new_data)) ;
#endif /* UNIV_AHI_DEBUG || UNIV_DEBUG */

        node = ha_search_with_data (table, fold, data) ;

        if (node) {
#if defined UNIV_AHI_DEBUG || defined UNIV_DEBUG
# ifndef UNIV_HOTBACKUP
                if (table->adaptive) {
                        ut_a (node->block->n_pointers > 0) ;
                        node->block->n_pointers--;
                        new_block->n_pointers++;
                }
# endif /* !UNIV_HOTBACKUP */

                node->block = new_block;
#endif /* UNIV_AHI_DEBUG || UNIV_DEBUG */
                node->data = new_data;
        }
}

#ifndef UNIV_HOTBACKUP
/*****************************************************************//**
Removes from the chain determined by fold all nodes whose data pointer
points to the page given. */
UNIV_INTERN
void
ha_remove_all_nodes_to_page (
/*=========================*/
        hash_table_t*           table,  /*!< in: hash table */
        ulint                   fold,   /*!< in: fold value */
        const page_t*           page)   /*!< in: buffer page */
{
        ha_node_t*              node;

        ASSERT_HASH_MUTEX_OWN (table, fold) ;

        node = ha_chain_get_first (table, fold) ;

        while (node) {
                if (page_align (ha_node_get_data (node)) == page) {
```

```
                        /* Remove the hash node */

                        ha_delete_hash_node (table, node) ;

                        /* Start again from the first node in the chain
                        because the deletion may compact the heap of
                        nodes and move other nodes! */

                        node = ha_chain_get_first (table, fold) ;
                } else {
                        node = ha_chain_get_next (node) ;
                }
        }
#ifdef UNIV_DEBUG
        /* Check that all nodes really got deleted */

        node = ha_chain_get_first (table, fold) ;

        while (node) {
                ut_a (page_align (ha_node_get_data (node)) != page) ;

                node = ha_chain_get_next (node) ;
        }
#endif
}

/*************************************************************//**
Validates a given range of the cells in hash table.
@return        TRUE if ok */
UNIV_INTERN
ibool
ha_validate (
/*=========*/
        hash_table_t*        table,              /*!< in: hash table */
        ulint                start_index,        /*!< in: start index */
        ulint                end_index)          /*!< in: end index */
{
        hash_cell_t*         cell;
        ha_node_t*           node;
        ibool                ok      = TRUE;
        ulint                i;

        ut_a (start_index <= end_index) ;
        ut_a (start_index < hash_get_n_cells (table)) ;
```

```
        ut_a (end_index < hash_get_n_cells (table)) ;

        for  (i = start_index; i <= end_index; i++)  {

                cell = hash_get_nth_cell (table, i) ;

                node = cell->node;

                while  (node)  {
                        if  (hash_calc_hash (node->fold, table)  != i)  {
                                ut_print_timestamp (stderr) ;
                                fprintf (stderr,
                                        "InnoDB: Error: hash table node"
                                        "fold value %lu does not\n"
                                        "InnoDB: match the cell number %lu.\n",
                                        (ulong)  node->fold,  (ulong)  i) ;

                                ok = FALSE;
                        }

                        node = node->next;
                }
        }

        return (ok) ;
}

/*****************************************************************//**
Prints info of a hash table. */
UNIV_INTERN
void
ha_print_info (
/*===========*/
        FILE*                file,   /*!< in: file where to print */
        hash_table_t*        table)  /*!< in: hash table */
{
#ifdef UNIV_DEBUG
/* Some of the code here is disabled for performance reasons in production
builds, see http://bugs.mysql.com/36941 */
#define PRINT_USED_CELLS
#endif /* UNIV_DEBUG */

#ifdef PRINT_USED_CELLS
        hash_cell_t*         cell;
```

```
        ulint               cells   = 0;
        ulint               i;
#endif /* PRINT_USED_CELLS */
        ulint               n_bufs;

#ifdef PRINT_USED_CELLS
        for  (i = 0; i < hash_get_n_cells (table) ; i++)  {

                cell = hash_get_nth_cell (table, i) ;

                if  (cell->node)  {

                        cells++;
                }
        }
#endif /* PRINT_USED_CELLS */

        fprintf (file, "Hash table size %lu",
                (ulong)  hash_get_n_cells (table)) ;

#ifdef PRINT_USED_CELLS
        fprintf (file, ", used cells %lu",  (ulong)  cells) ;
#endif /* PRINT_USED_CELLS */

        if  (table->heaps == NULL && table->heap != NULL)  {

                /* This calculation is intended for the adaptive hash
                index: how many buffer frames we have reserved? */

                n_bufs = UT_LIST_GET_LEN (table->heap->base)  - 1;

                if  (table->heap->free_block)  {
                        n_bufs++;
                }

                fprintf (file, ", node heap has %lu buffer (s) \n",
                        (ulong)  n_bufs) ;
        }
}
#endif /* !UNIV_HOTBACKUP */
```

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引