【译】第一章：MySQL的结构与历史（HP-MySQL 3rd）

2016-08-07 | 30

> High Performance MySQL, 3rd Edition

MySQL与其他数据库不同，其体系结构特质使它能够被广泛的使用，但还是有不宜使用的特殊情况。MySQL并不完美，但也在特定环境下也能很好的工作，比如web应用。现在，MySQL能用于嵌入式设备、数据仓库、内容索引以及分发系统（delivery software）、高可用的冗余系统、联机（在线）交易处理（OLTP）等等。

为了更好的使用MySQL，理应了解其设计。MySQL在很多方面比较灵活。比如，在多种硬件上可以使用它，它也支持多种数据类型。MySQL最与众不同的是它的存储引擎结果，不同的存储引擎使得查询语句的处理和数据存储、获得方式都不同。我们都需要选择使用何种数据存储方法，以及我们最关心哪种性能、特点。

这一章简述了MySQL的服务器的结构、不同存储引擎的差别以及为什么存储十分重要。也会谈到MySQL的历史和性能检测。这一章会简化细节和举例，这样使得新人和熟悉别的数据库的人能够尽快学习。

MySQL的逻辑架构

一张好图能让你理解MySQL各个部分是如何结合在一起工作的。下图展示了MySQL的逻辑架构。

Isolation level	Dirty reads possible	Nonrepeatable reads possible	Phantom reads possible	Locking reads
READ UNCOMMITTED	Yes	Yes	Yes	No
READ COMMITTED	No	Yes	Yes	No
REPEATABLE READ	No	No	Yes	No
SERIALIZABLE	No	No	No	Yes

MySQL的逻辑架构

最上层并非是MySQl独有的，它们存在于大多数基于网络C/S系统中：连接处理、认证、安全等等。
第二层比较重要，这是MySQL的核心所在：查询语句的解析代码、分析、优化、缓存、所有内建函数（比如:dates、times、math和加密）。所有在不同引擎通用的功能都在这一层：存储过程、钩子方法、视图。
第三层是存储引擎。它们负责存储和取回数据。如GNU/Linux一样，不同的存储引擎有各自的优缺点，MySQL通过API与存储引擎通信。接口API对查询层（第二层）暴露的程度依据存储引擎的不同而不同。API包含大量大类低级别的操作如开始一个事务、获得含有这个主键的行。存储引擎不会解析SQL或彼此通信，他们只能对MySQL服务器发出的请求进行回应。(有一个例外，就是InnoDB会自己解析外健、因为MySQL还没有实现这个功能)

连接管理和安全

每一个客户端的连接都能获得由服务器一个线程提供的服务。连接的查询也会在这个线程中进行，线程轮流获得核或者CPU。MySQL服务器缓存了线程，所以不会每次连接时都发生线程的创建和销毁。（MySQL 5.5以及更新的版本支持一个API，它能够使用线程池插件、所以有一个小的线程池服务大多数连接）
当客户端应用连接到MySQL服务器，服务器会验证它们。验证主要依据：用户名、发起的主机、密码。X.509数字证书在SSL连接中也能发挥作用。一旦客户端连接上服务器，服务器会验证客户端发起的每一个查询的权限（如这次客户端发起的SELECT请能否读取word数据库中Country表的数据）。

优化以及执行

MySQL解析查询语句，然后创造一个内部结构（解析树），并且应用一系列优化。包括重写查询语句、决定表读取的顺序、选择需要使用的索引等。我们也可以通过查询语句一些特别的关键字传送提示给优化器，影响优化过程。我们也能让服务器解释优化的过程，这使得我们能了解服务器做了什么决定，给我们一些提示如何重写SQL语句、制定表结构，使得查询尽可能高效。第六章我们将详谈这一点。
优化器不关心一张表到底选择了哪种存储引擎，然而存储引擎确实影响了服务器优化器如何进行优化。优化器会询问存储引擎一些问题：性能、某些操作的代价和表中数据的情况。比如，某些存储引擎的索引将极大的加速查询。这些关于索引和表结构优化的内容将在第四章和第五章谈及。

其实在解析查询语句之前，服务器会检索查询缓存，它只会存储SELECT语句和结果集。如果查询语句恰好在查询缓存中，服务器不需要解析、优化、执行语句，因为可以直接返回已经存储的结果集。第七章将详谈这一点。

并发控制

当同时有两条语句要改变数据时，并发控制问题便发生了。MySQL会在两个层级做出控制：服务器层、存储引擎层。并发控制是一个复杂话题，大量文件在讨论它。因此我们只会简单的讲述一下MySQL如何处理并发的读写者问题，这样你就可以储备一定的知识完成本书的阅读。
使用UNIX的邮件作为例子。经典的*mbox*文件格式非常简单，将所有消息都一个接一个的在信箱中。这使得非常容易读取和解析信箱消息，这也是发送十分简单，直接将消息添加到文件的最后一行即可。
但是当两个进程同时想要取发送信箱中的消息怎么办？显然在信箱中造成了冲突，留下了两个相互交叉的消息在文件的末尾。高级的信箱系统使用锁来阻止冲突，当第二个客户端试着取发送消息，然而信箱被锁住了，它必须等到获得了锁之后才能发送消息。

实践中也是如上述过程运作的，但是对并发没有好处。因为只有一个进程能改变邮箱，当邮箱容量过高时，这将成为一个瓶颈。

读/写锁

从邮箱里读没啥问题，多个客户端会读到同样的内容。因为他们没有要求改变，没什么会出错。但是当有个客户端试着删除第25号消息呢？这时，从邮箱读取的情况就要没准了。但是读时应该避免混乱的情况或与多个读者之间不一致的情况，所以了尽管是读也需要特别的考量。
把邮箱视作表，把消息视作行，那么数据库也就面临的同样的问题。
并发控制的解决方案是利用两把锁，读锁和写锁分别控制数据库的读和写。读锁也可以称为分享锁，写也可被称为排斥锁。
不关心实现细节的流程如下：一个资源的读锁是可以被共享，多个客户端可以同时读，不会相互影响。然而写锁是相互排斥的，它会阻碍其他客户端的读和写。因为唯一安全的策略就是在一个时间段内只让一个客户端对资源写，同时阻止其他客户端的读与写。
数据库世界中，锁随时都在发生：MySQL会阻止客户端去读一个正在被写的数据。并且锁操作会耗费大量的时间。

锁的粒度

改进并发性能的一种方法就是被锁资源的精心控制。不是一下锁住所有资源，而只锁住需要改变的部分，最好能精准地、不多不少的锁住。最小化锁住被改变的部分使得可以对一个资源同时进行尽可能多改变，只要被改变的地方不相互冲突。
还有一个问题就是锁是非常耗费资源的。每一个锁的操作：得到锁、检查锁是否空闲、释放锁等等非常麻烦。如果系统耗费大量资源在锁上，而不是存储和取回数据，那么性能堪忧。

锁的策略是在锁的复杂和数据安全之间的一种权衡（妥协），这会影响性能。商业数据库不会给我们太多选择：我们可以会使用行粒度的锁，并且使用一堆复杂的方式保证了性能。
MySQL提供选择。存储引擎会实现各自锁的策略和锁的粒度。锁的管理对于存储引擎的设计十分重要。虽然某种固定锁的粒度对特定场景非常有用，但是一般不能广泛应用。因为MySQL提供多种数据引擎，所以也不是特定某种场景的解决方案。让我们来看看两种最重要的锁策略：

表锁

最简单也是最基本的方式就是表锁，也就是锁住整张表。当一个客户端想要写的时候（INSET、DELETE、UPDATE），即可获得写锁。它会阻碍其他客户端所有的读和写。当没人写时，读者会获得读锁，读者之间并不冲突。

表锁有一些适用不同场景的变体。比如READ LOCAL表锁允许一些写操作的并发。写锁权限比读锁高，当写锁请求到来时会跳过所有已经等待的读锁，但读锁不能。
尽管锁策略由存储引擎决定，但是MySQL针对特殊操作也设计了表锁，比如执行ALTER TABLE时MySQL将会进行表锁，无论是哪个存储引擎。

行锁

行锁能提供最大限度的并发（当然也承担了最高的复杂性），InnoDB、XtraDB等其他引擎提供这种方式。行锁主要由存储引擎实现。MySQL服务器几乎不会察觉到行锁的实现。之后将会读到，各种存储引擎实现了自己的锁策略。

事务

在没有接触事务之前，你不用担心太多数据库的高级特征。事务一组SQL必须原子性处理（不可分割），作为整体一起处理。如果数据库引擎恰好能一次性处理掉这组SQL，那当然最好，但是如果这组查询中有任何一个崩溃了或者其他原因，没有执行成功，那么这一组SQL都不该被执行。要么全执行了，要么都没执行。

下面要说的不是针对MySQL的，这是所有数据库关于事务的基础知识。
银行应用是一个典型的标志着事务的重要性的例子。银行数据库有两张表：checking 或者 savings。从Jane的支票帐户转移$200到她的储蓄账户，至少需要三步：

1. 确认她的支票账户余额大于$200。
2. 从支票账户减去200。
3. 在存储账户增加200。

整个过程（三步）应视作一个事务。所以，其中哪一步失败了，那么整个过程都需要回滚（没有一条语句被操作）。

可以使用START TRANSACTION语句表示开始事务，使用提交的语句生效可以使用COMMIT，或者使用ROLLBACK撤销改变。我们例子的SQL如下：

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

但是事务远没有那么简单。如果当数据库服务器执行第四句的时候崩溃了，那么客户就相当于刚好失去了200元。但当执行到3、4句之间时，有一个进程来移除了整个支票账户。那么银行相当于白送了200给这个人。
系统必须经过了ACID测试才能正确的应对上述情况。所谓ACID是指原子性、一致性、分离性和持久性。这是能处理事务的数据库必须满足的。

原子性

事务的几句SQL必须视作一个整体，要么都执行，要么都回滚。只要是原子性的，就不存在一些SQL执行了，另一些没有。

一致性

数据库的状态应该是从一个一致的状态到另一个的一致状态。一致性是指即使在3、4行崩溃了，也不能导致支票帐号的200元消失了。因为事务没有被COMMIT，不应有任何语句作用于数据库。

分离性

独立性是指事务之间相互独立，不可交叉，事务未被提交前也不能被其他语句察觉。比如如果有账户总额查询的语句在3、4句间执行了，那么总额查询还是会得知有200美元在支票账户。这就是当我们谈及独立性时，通常意味着相互之间不可见。

持久性

一旦被提交，事务造成的改变是永久的。这也意味着改变应该被记录，即使当系统崩溃了也不受影响。持久性这个概念有点混淆，因为分了很多个层级。有些策略提供了更强的安全保证，但是没有一个能达到100%持久（如果数据库本身是真的持久化的，那么备份如何持久化），我们以后将讨论持久化在MySQL中的真正含义。

ACID事务确保了银行不会搞丢你的钱。如果仅用逻辑去处理这是一个极难的问题。一个满足ACID的数据库在你没有知觉的情况下完成各种各样复杂的操作。

正如锁粒度一样，安全的保证也导致了数据库的复杂性。满足ACID事务的数据库需要更多的CPU、内存、磁盘空间等。这也就是MySQL的好处，你可以决定你的数据库是否需要事务，如果你真的不需要，你可以使用一个高性能的没有事务的存储引擎。没有事务的情况，你可以使用LOCK TABLES给予一定的保护。

分离级别

分离实际上非常复杂。SQL标准建立了四种分离级别，决定了哪些操作对内、对外可见。低级别的分离层级允许更高的并发和复杂性较低。

> 分离级别没有统一标准，所以各个存储引擎情况不同。如果需求，请查阅具体的需要实用的数据库引擎关于分离的介绍。

简单看一下四种分离级别：

能读到未提交的数据

在这个分离级别，事务能够读到未提交的事务操作结果。在这个级别，能引发很多很多乱七八糟的事。实践中几乎不会这么做，因为这个级别的性能也没有比别的级别提高多少，但是别的级别有很多优势。读到了未提交的数据，我们称为脏读。

能读到提交的数据

这个级别是大多数数据库默认的分离级别（但MySQL不是！）。它满足前面提到的分离性原则：一个事务只能看到别的事务已经提交的改变。如果没有提交，那么别的事务看不见。这个级别也被称之为*不可能重复读*，因为同一条语句可能有两个不同的结果。

可重复读

*可重复读*解决了*能读到提交的数据*的问题，它保证了事务读到的每一行在之后同一事务重复读时(没有提交，读了两次)的一样。但是在理论上又引发了另一个问题：幻读。简单来说，就是幻读发生在当我选择一个范围的行时，另一个事务在这个范围内插入了新行，但是当你同样再查询这个范围时，就会出现新插入的行。 之后我们将知道，InnoDB和XtraDB用多版本并发控制解决了幻读。

*可重复读*是MySQL默认的事务分离级别。

串行化

最高级别的分离层级，串行化通过强制要求事务按序执行解决了幻读的问题，因为这不可能冲突。简而言之，串行化就是对每一个需要读的行都放置了一把锁。这个级别引发了大量的时间消耗和锁操作。几乎没人用这个级别，但也许你的应用需要，那么只要痛过降低并发，增加数据和查询的稳定性。

下图是各个分离级别和其缺点：

Isolation level	Dirty reads possible	Nonrepeatable reads possible	Phantom reads possible	Locking reads
READ UNCOMMITTED	Yes	Yes	Yes	No
READ COMMITTED	No	Yes	Yes	No
REPEATABLE READ	No	No	Yes	No
SERIALIZABLE	No	No	No	Yes

死锁

死锁是指两个及以上的事务相互拥有又需要彼此的锁，也就是创建了依赖了的环。事务需求资源顺序的不同，会造成死锁。当多个事务锁住相同资源时，也能发生死锁。比如，下面两个事务都需要对StockPrice这张表进行操作：

事务 #1

```
1  START TRANSACTION;
2  UPDATE StockPrice SET close = 45.50 WHERE stock_id = 4 and date = '2002-05-01';
```

```
3  UPDATE StockPrice SET close = 19.80 WHERE stock_id = 3 and date = '2002-05-02';
4  COMMIT;
```

事务 #2

```
1  START TRANSACTION;
2  UPDATE StockPrice SET high = 20.12 WHERE stock_id = 3 and date = '2002-05-02';
3  UPDATE StockPrice SET high = 47.20 WHERE stock_id = 4 and date = '2002-05-01';
4  COMMIT;
```

如果不走运时，每一个事务刚刚执行完第一句，并且更新了一个行数据，也锁住了对应行，当每个事务刚好要执行第二句时，只能发现需要的行已经被锁住了。两个事务会永远相互等待对方完成，除非有什么意外打断了死锁。

为了预防此类死锁，数据库系统实现了各种各样的死锁检查和超时。高级点的系统，如InnoDB存储引擎会主要到相互依赖的环，立刻返回错误。这是好事，否则死锁会让查询很慢。其他系统只是让查询执行一会，最终超时返回，这并不太好。InnoDB目前的处理死锁的方式就是回滚拥有对行的执行锁最少的事务（对行拥有执行锁最少，可以认为是一个近似指标，表示最容易回滚）。

锁行为和顺序根据存储引擎不同而不同，所以有些某语句在这个存储引擎会发生死锁，但在别的不会。死锁发生有两个原因：有些是真的发生了资源的冲突；有些则是因为存储引擎工作的方式。

不回滚一个事务，是不可能解决死锁的，无论一个还是全部。在处理系统的事务时，一定要处理好各个事务之间的关系。许多应用的解决办法往往是直接重试。

事务日志

事务日志使得事务更高效。不是每个有一丁点更新就磁盘上的表，存储引擎会更新数据在内存中的备份，这样速度很快。存储引擎会在事务日志里打一条记录表示做了什么改变，这是在磁盘上的，可以认为是双保险。这也是相对较快的操作，因为这只是添加了一行，属于在磁盘上一个小地方做序列I/O，而不是费时的随机I/O。之后，合适的时候，一个进程会更新磁盘上的表。也就是说，很多存储引擎会使用这个技术（在写之前的日志：在对表进行改变时先记录下下来发生什么改变。）避免对磁盘产生了多次操作。
即使在事务日志记录改变之后，在磁盘数据真正发生改变之前发生了崩溃。存储引擎在重启后仍然可以从事务日志里恢复。不同的存储引擎恢复方式不同。

MySQL中的事务

MySQL提供两种事务存储引擎：InnoDB和NDB。几个第三方数据引擎也可以用，比较有名的是XtraDB和PBXT。下一节我们会讨论几种引擎的特殊性质。

自动提交

MySQL默认开启自动提交，也就是说除非我显式开启一个事务，否则会将任何一个语句当作事务来处理。我们在每一个连接中打开或者关闭自动提交（AUTOCOMMIT）

```
mysql> SHOW VARIABLES LIKE 'AUTOCOMMIT';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | ON    |
+---------------+-------+
1 row in set (0.00 sec)
mysql> SET AUTOCOMMIT = 1;
```

值1是开启的意思，0表示关闭。当设置AUTOCOMMIT为0时，始终处于事务内，除非发送COMMIT或者ROLLBACK。MySQL然后就会开启一个新事务。改变对非事务表设置AUTOCOMMIT没有用，比如MyISAM和内存表，这些表没有COMMIT和ROLL BACK的概念。

有些语句，当存在一个事务时被发起的话，会导致MySQL提交事务再执行这些语句。这种被称为数据定义语句（DDL）会导致显著的变化，如ALTER TABLE，但如LOCK TABLE和别的语句也有此类效果。更多请查阅与版本对应MySQL官方文档以了解会导致事务自动提交的语句。

MySQL可以让你设置分离级别，语句为：TRANSACTION ISOLATION LEVEL，但在下次事务时才会有效果。也可以在配置文件中设置整个服务器的分离级别，或者仅仅是一段会话（有个解释是会话为逻辑概念、连接是物理概念，连接可以有多个会话，也可以没有会话）

```
1   mysql> SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

MySQL能够识别4种分离级别，InnoDB都支持。

一个事务的混合存储引擎

在服务器级别是不会控制事务的，更下层的存储引擎才负责实现事务。所以在一个单一事务中不能有多个存储引擎。
如果混合事务和非事务的表（如InnoDB和MyISAM的表）在一个事务中，可以正常进行。
但是如果执行ROLLBACK，非事务表是不会做任何改变的。这会导致数据库处于一个不一致的状态，所以在选择存储引擎的时候需要非常小心。

MySQL不总是会提示你或者抛错，让你知道你在非事务表上执行事务。偶尔只是在ROLLBACk的时候提醒一下：Some nontransactional changed tables couldn't be rolled back。除此之外，没有任何信息表明你在一个非事务表上执行事务。

隐式锁、显示锁

InnoDB使用两种锁协议，在一个事务中可以在任何时候获得锁，但是除非COMMIT或者ROLLBACK，是不会释放的。所有锁都是在同一个时间释放。上面描述锁都是隐式。InnoDB会根据你的分离级别自动处理锁。
同时，InnoDB支持显示锁，但SQL的标准教程却完全没有提过：

- SELECT … LOCK IN SHARE MODE
- SELECT … FOR UPDATE

MySQL也支持LOCK TABLES 和UNLOCK TABLES 语句，它在服务器界别上处理，而不是存储引擎上。他们有自己特别的用途，而不是事务的替代品。如果你真的需要事务，应该使用支持事务的存储引擎。
有时候需要将存储引擎从MyISAM切换至InnoDB，那么之后没必要使用LOCK TABLES，因为InnoDB有行级别的锁，而且LOCK TABLES性能极低。

> LOCK TABLES和事务的交叉是异常复杂的，有很多无法预期的行为。所以无论在使用哪种存储引擎，我们推荐永远别在事务中使用LOCK TABLES，除非AUTOCOMMIT已经关了。

多版本并发控制

大多数MySQL的事务存储引擎并不是简单的使用锁行的机制。除了锁行的机制外，还会配合使用一种被称为多版本并发控制的机制来增加并发。MVCC并不是MySQL的专利，Oracle、PostgreSQL，以及其他的数据库系统也使用它，但是它们之间还是充满着差异，这是因为没有一个MVCC的标准来约束它如何工作。
你也可以认为MVCC是行锁的补充，它使得行锁并不是锁住全部，减少了复杂性。根据其实现的方式，它允许*不锁的读*，而只是锁住在写时必要的行。
MVCC工作的方式是保持一份某个时间点时数据的快照，这意味着一个事务可以看到一致的数据，无论这个事务运行多久。这也意味着不同事务看到的是不同的数据（同一张表）在同一个时间，因为事务总在对各自的表进行操作（体现了多版本）。听起来很困惑，以后酒慢慢熟悉了。
不同的存储引擎实现MVCC的方式不同。有一些变种：悲观和乐观的并发控制。我们通过InnoDB的行为来解释一个简单版本的MVCC。
InnoDB实现MVCC通过为每一行额外存储两个记录：当这一行被创建和销魂时。并不是记录实际发生的时间，而是记录了发生这些事件时的系统版本。系统版本是一个随事务增加时的数。每个事务开始时都会保存当前的系统版本。每个查询都会将每一行的系统版本数和事务存储的版本数作比较。让我们看看具体的操作是怎样的（当当前分离级别设置为可重复读时）：

1. SELECT
   InnoDB 会检查每一行并且满足两个条件：

a. InnoDB会对每一行找到一个版本，这个版本必须比事务的版本小。(也就是这个版本号必须低于或者等于事务的版本号)。这确保了存在的每一行都在是事务开始之前的状态、或者是事务创建、改变了这一行。

b. 行的删除时的系统版本必须是为未定义的，或者大于事务的版本。这确保了这一行没有在事务开始之后被删除。

2. INSERT

   InnoDB设置了新插入行的系统版本号为当前系统版本号。

3. DELETE

   InnoDB设置了被删除行的删除系统版本号为当前系统版本号。

4. UPDATE

   InnoDB复制一行新的，使用当前系统版本号为新行的版本号。也设置旧行的删除系统版本号为当前版本号。

这些额外的努力都确保了大多数读不需要锁。这使得读数据会尽可能的快，只需要确保读的行是满足那两个条件的。
当然代价就是需要存储引擎存储额外的数据，还需要做额外的检查，做一些类似于打扫房间似的操作。

MVCC只能在*可重复读*和*能读到提交了的数据*这两个分离级别发生效力。*能读到未提交的数据*并不适用MVCC，因为查询是不会检查行的系统版本号与事务版本号之间的关系。串行化也不适用于MVCC，因为串行化每次都要拿到读锁。

MySQL的存储引擎

这一节简介MySQL的存储引擎，但不会谈论的太细，因为后面会慢慢谈。起始这本是也不足够细，最详细的资料是MySQL手册。
MySQL在数据目录下建立子目录来存储每一个数据库（也称为schema）。当你建立一张表，MySQL会以表名建立一个*.frm*的文件用于存储表的定义。比如表名为MyTable，就会有文件MyTable.frm。因为是用文件系统来存储数据库和表名，所以大小写是否敏感就看操作系统了。Windows大小写不敏感，*nix大小写敏感。各个存储引擎会负责表数据和索引的存储，但是服务器会负责处理表定义。

SHOW TABLE STATUS（当MySQL版本大于5.0时，INFORMATION_SCHEMA ）可用于显示表的信息。比如，查询表user的情况。

```
mysql> SHOW TABLE STATUS LIKE 'user' \G
*************************** 1. row ***************************
           Name: user
         Engine: MyISAM
     Row_format: Dynamic
           Rows: 6
 Avg_row_length: 59
    Data_length: 356
Max_data_length: 4294967295
   Index_length: 2048
      Data_free: 0
 Auto_increment: NULL
    Create_time: 2002-01-24 18:07:17
    Update_time: 2002-01-24 21:56:29
     Check_time: NULL
      Collation: utf8_bin
       Checksum: NULL
 Create_options:
        Comment: Users and global privileges
1 row in set (0.00 sec)
```

显然是一张MyISAMd的表，下面介绍一下比较难的：

Row_format：行的格式。对于MyISAM有三种情况：Dynamic、Fixed、Compressed。Dynamic存储可变的数据如：VARCHAR、BLOB。Fixed始终存储相同大小的数据，长度不变的：CHAR、INTEGER。Compressed只存在于Compressed表，后文会介绍。
Row:拥有的行数，除了InnoDB是估算值，大多数引擎都是准确的。
Avg_row_length:行的平均长度。
Data_length：整个表的数据大小（字节为单位）。
Max_data_length:表的最大大小，依据引擎而定。
Index_length: 索引耗费了多少磁盘空间。
Data_free: 对于MyISAM，就是分配但是未被使用的空间。包含之前删除了行，但仍然能被使用。
Auto_increment：下一个自增长的数。
Check_time: 使用CHECK TABLE或myisamchk被CHECK的时间。

Collation：默认的字符集格式。

Checksum：内容的检验码，如果开启了的话。

Create_options：当表创建时额外的指定的选项。

Comment：包含各种各样额外的信息。对于MyISAM，它就包含了评论，如果有就是创建时设置的。如果是InnoDB存储引擎，这里是剩余空间。如果这个表是视图，那么这个评论就显示'VIEW'。

InnoDB存储引擎

InnoDB是MySQL默认的事务型存储引擎、最重要、广泛使用的引擎。它针对短时长的事务，并且多半是会被完成而不是回滚的。但是其性能和崩溃自动恢复也使得其为比较常用的非事务存储引擎。其实，基本上你就默认使用InnoDB吧，除非你已经找到充分的理由使用别的存储引擎。如果你就是为了研究数据引擎，那么InnoDB也是最佳选择。

InnoDB的历史

InnoDB的历史很复杂，但理解了也很有用。2008年InnoDB插件随着MySQL5.1发布，这是被Oracle创建的第二代InnoDB存储引擎，这时候InnoDB属于Oracle，不属于MySQL。经过广泛的讨论，MySQL持续性的融合老版本的InnoDB。反正后来Oracle收购了Sun，MySQL5.5默认安装了最新的InnoDB版本。

现在最新版本的InnoDB，在MySQL5.1的时候就已经引进了，并且有很多新功能比如利用排序建立索引，不重建表的情况建立、删除索引、支持压缩、存储大格式如BLOB的新的方式、还有文件格式管理，在你使用MySQL的版本大于5.1时，请注意你使用的是最新版本的InnoDB。

> 说了两大段，就是InnoDB好，很多公司花了精力去改进它，所以很好用。现在瓶颈主要在CPU上，在24核上已经非常稳定了，而在其他场景还支持32甚至过多核。MySQL5.6之后都有这些功能。

InnoDB概览

InnoDB管理所有数据以一系列文件的形式，这些文件被称为表空间，这是一个黑盒，被InnoDB自己控制。新版本中InnoDB分开存储索引和数据。InnoDB可以使用磁盘分许，但是现代化的文件系统不需要它这样了。

InnoDB使用MVCC来实现了高并发，并且实现了四种分离级别，并默认使用可重复读的分离级别。同时用*锁住下一个键*的策略阻止了幻读：除了锁住需要查询的行，InnoDB会锁住下一个索引的位置，防止插入，防止了幻读。

InnoDB使用*聚集索引*来建立表，本书后面章节和详细讨论。这种索引结构和大多数存储引擎不同，它会使主键的查询非常快。此外，二级索引也会包含主键那一列，所以如果主键数据量大，那么别的索引也非常大。请用心设计主键，如果表中有多个索引的话。存储格式是平台中立的，这意味着我们可以在多个Intel平台上通过复制使用数据。

InnoDB有很多优势：预测预读（从而提前从磁盘读入数据）、自适应的哈希索引（在内存中自动建立哈希索引加快查询）、维持插入池加速插入。之后将详细讨论。

当需要使用InnoDB时，强烈推荐阅读MySQL手册的*InnoDB Transaction Model and Locking*部分，由于MVCC的存在，很多细微之处需要注意。如果需要保证数据对所有用户一致，并且有用户改变数据时，这非常复杂。

作为一个事务型存储引擎，InnoDB支持在线备份（使用了Oracle的专利）。MySQL其他引擎都不支持在线备份，必须在阻止了所有读之后才能备份。

MyISAM引擎

5.1之前 MyISAM 是 MySQL 默认的引擎，有大量的特征：全文索引、压缩、空间函数（GIS）。MyISAM不支持事务和行级别的锁。最大的不足其实是没有远程崩溃修复，不能保证数据的安全。MySQL作为著名的非事务数据服务器全靠有MyISAM。当然，如果你需要只读的数据，或者表不大，修复起来也没啥，那么毫无疑问可以使用MyISAM，但是别默认使用它哦。

存储

MyISAM 用两份文件存储一张表：数据文件和索引文件，分别以.MYD和.MYI结尾。MyISAM可以存储动态或者静态（长度固定）的行，这是表定义时确定的。最大行数是由磁盘空间和操作系统的最大文件大小所决定的。

MySQL5.0之后的 MyISAM（可变长度行）表默认能够处理256TB的数据，使用6字节的指针对应一条记录。指针大小也可以达到8字节。想要改变指针大小，可以通过设置MAX_ROWS和AVG_ROW_LENGTH来调整，这两个参数大致表明你需要多大的空间。这个设置会导致表和索引的重写，非常耗时。

MyISAM 优点

- 锁与并发
  MyISAM会锁住整张表。读者需要拿到读锁，写者需要写锁。但是读着可以插入数据（也就是并发插入）。

- 修复
  MySQL支持手动和自动检查和修复表，注意这个不是只事务或者是崩溃自恢复。修复表后，你极有可能发现有些数据就是简简单单的丢了。修复极慢。可以使用CHECK TABLE mytable和REPAIR TABLE mytable来检查错误和修复。当服务器关闭时，myisamchk可以用于离线修复数据。

- 特别的索引
  可以对BLOB和TEXT格式的前500个字符做索引。MyISAM支持全文索引，也就是索引单个的字符以应对复杂的搜索操作。更多信息请见第五张。
- 索引数据延迟写入
  DELAY_KEY_WRITE表示当SQL语句结束时不将索引的数据立即写入磁盘，只改变当前内存中的数据。关闭表或者冲洗缓存时才会将改变写入磁盘。这会增强性能，但是当崩溃（系统或者服务器）发生后，这些索引的数据需要被修复。这个DELAY_KEY_WRITE可以全局配置，也可以单独对一张表配置。

压缩MyISAM的表

当表一旦建立、装入数据后用不改变时，适用被压缩。
myisampack工具用于压缩表，压缩后不能改变（除非解压、插入、重新压缩）。压缩后占用空间少，性能更佳，这是因为较小的空间只需要更少的磁盘搜索就能找到数据。压缩后的表也有索引。

读数据时不用担心解压带来的性能问题(磁盘IO才是大问题)，因为解压时并不会解压整张表，只会解压需要的行，因为行是独立压缩的。

MyISAM的性能

因为简单的设计，MyISAM对特定的使用能够较好的性能。当然也有严重的问题，比如在健缓存的互斥（暂时不知道啥意思，可能是取名的健只能存一个）。MariaDB（MySQL的分支）通过将片段的健来避免这个问题。当然最大的问题是锁表，如果你所有查询都被暂停了，那你可能遭遇了锁表。

其他内置的MySQL引擎

都是一些有着特殊用处的表。

Archive引擎

Archive仅支持INSERT和SELECT语句，它MySQL5.1之前不支持索引。因为它会缓存所有读的数据和压缩插入的每一行为zlib。所以比MyISAM大量减少了磁盘I/O。但是每一个SELECT都会引发全表扫描。Archive特别适合日志，因为对日志分析时一般都会全表扫描，并且你也需要更快的插入。Archive是非事务的，支持高并发的插入和压缩存储。

Blackhole引擎

读了半天没觉得有特色，应该不会用。

CSV引擎

CSV还有点意思，把CSV的文件丢到数据目录下，就可以像表一样操作这些文件里，读、写都会发生这些文件上，方便拷贝。但不支持索引。

Federated引擎

其实是其他数据服务器的代理，可以通过它与别的数据服务器（Oracle，SQL server）通信、收发数据。现在MySQL已经默认不禁止使用了。但MariaDB有。

Memory引擎

当你需要快速读数据，也需要改变或持久化的时候，内存表（正式点：堆表）是有用的。读的速度会比 MyISAM 快一个量级，因为所有的数据都在内存中有序了，不需要磁盘IO，但是服务器重启后数据会丢失，但表结构可以保存。优势如下:

- 对于字典类的需要可以使用，比如邮政编码和各州的名字
- 对于缓存繁杂计算的结果
- 分析数据的中间结果

> 这儿玩意目前是由redis替代了，所以没必要深入研究和使用。
>
> 注意临时表和内存引擎表的区别，必须使用显示的CREATE TEMPORARY TABLE来创建临时表。临时表可以使用任何数据引擎，临时表在一次连接内存在，连接关闭就消失。

NDB 集群引擎

MySQL集群就是由MySQL服务器、NDB集群引擎、（分布式、无共享的、高容错的高可用的）NDB数据库组成。当谈到MySQL集群时我们会谈到这一点。

> 集群来看，貌似是HBase比较有名。

第三方的数据引擎

因为MySQL提供了数据引擎的API，所以有一些第三方的数据引擎，保持了多样性。

OLTP 存储引擎

> 大概浏览了一下，介绍类似于InnoDB的数据库，我应该不太会使用除InnoDB以外的数据库，所以没仔细看。

面向列的存储引擎

> 调查了一些，如果要用面向列的数据库，那么用Hbase应该不错，又开源又分布式。此外，mongodb也不受限于面向列，所以也是不错的替代品。
> 所以就原文就不翻译了，毕竟科技还是发展的很快的。

社区的存储引擎

都是些生产环境中没啥用的，所以略过。

选择正确的引擎

默认选InnoDB总是不太会错的。比如说尽管需要全文检索，我们更推荐InnoDB和Sphinx组合，而非使用 MyISAM。另外一般来说，除了有非常特别的需要，才不会用InnoDB。比如我们不要稳定性、不需要高并发、不要扩张、不要安全，只要磁盘空间的节省，那么我们可以选择MyISAM。

不推荐使用混合的存储引擎。存储引擎之间的交互非常复杂，而且连备份都没法一致、有时还导致识别服务器错误。

选择存储引擎需要考虑到以下几点：

- 事务：如果需要事务，最好选用InnoDB。如果不要事务，又有大量的SELECT和INSERT，可以选择MyISAM，实际上只有一些特殊的场景才会需要这个，比如日志。

- 备份：除非可以关机备份，那么只有InnoDB支持在线备份。

- 崩溃后恢复：MyISAM容易崩溃，又需要较长时间来恢复。所以大多人基于这个原因选择InnoDB，尽管不需要事务。
- 特殊需求：此时有两种选择，要么直接选满足特殊需求的引擎，要么精心设计还是用InnoDB。最好详细调查后再下决策，因为有些引擎看起来满足，实际上没有满足。

读到这里不要急于下选择，因为引擎之间的更多优缺点没有谈到，所以读完本书再说。另外，如果搞不清需要什么就选InnoDB，其实现在不太懂的情况最好也选InnoDB。

上面谈的比较抽象，下面将举例子，比如具体的应用、具体的表应该用什么引擎。

日志

比如电话中心记录电话的日志，网站记录网站的日志（*mod_log_sql*可以使Apache的日志直接插入数据库），记录这些日志需要极快的插入速度，所以：MyISAM和Archive是不错的选择。
当需要SELECT这些日志产出总结报告时，为了不影响插入，有两种解决方案。

1. MySQL内置的复制功能，将数据复制到另一台服务器上，在另一台服务器上执行查询操作。但是当数据量大增的时不要使用这种方案。
2. 按年月建表，如：web_logs_2012_01。这样可以在不需要插入的表中执行查询操作。

以读为主的表

这类场景主要是列表展示类的：工作数据、拍卖数据、房地产数据，别想太多，还是InnDB。

1. 不要低估了崩溃后无法修复的功能，你可以自己扯一次服务器的电源试试看，到底有多惨。
2. 影响查询速度是各方面因素决定：数据大小、I/O次数、主键、次级索引等等。
3. MyISAM开始时还不错，当插入变得繁忙，数据量变大时就不好说了。

订单处理

InnoDB，因为肯定是事务。其次还能建立外键约束。

论坛

刚开始的时候写入相对较少，但是流量大了写入就会变多。另外，单独保存计数器（别的信息）也很重要，而不是每次都执行SELECT统计有多少帖子。也别存在一张大表里，以后会吃苦头的。
吃了苦头像切换存储引擎，实际上会改写很多SQL：

```
1   SELECT COUNT(*) FROM table;
```

MyISAM很快，但是别的存储引擎比较慢，之后我们再详细讨论还有哪些这样的语句。

CD-ROM 应用

因为CD-ROM就是只读的，所以可以使用 MyISAM，主要是因为MyISAM能压缩空间。

大数据量

3TB~5TB的话，单机的InnoDB可以胜任的，但需要专心巧妙的设计硬件，挑选机器之类的。但是这个大小，如果使用MyISAM，崩溃了你就哭死。
再大，10TB的话，就是数据仓库了，那么推荐：Infobright、TokuDB。

> 开源的OLAP：Apache Kylin，原来OLAP非常复杂。

表搜索引擎的转换

有三种方式可以完成，各有优缺点，下面详述。

ALTER TABLE

命令ALTER TABLE可以完成存储引擎的转换，如下转换为InnoDB。

```
1   ALTER TABLE mytable ENGINE = InnoDB;
```

所有引擎都可以，但是比较费时。因为MySQL会一行一行的复制，且占用所有的IO，如果这张表比较繁忙，最好不要这样做，下面有更好的办法。
切换引擎时，之前引擎特有的功能全都会消失，比如从InnoDB到MyISAM，外键会消失。

备份再导入

*mysqldump*命令能将表数据下载为一份文档，这份文档可以编辑，我们加上CREATE TABLE即可，注意改变表名和引擎类型，MySQL不允许表相同。此外*mysqldump*默认会DROP TABLE，然而再CREATE TABLE，请注意这一点。

CREATE和SELECT

第三种方式是第一种的快速和第二种的安全之间的妥协，不u是下载全部数据，而是创建一个新表，再INSERT … SELECT 填满新表。

```
1   mysql> CREATE TABLE innodb_table LIKE myisam_table;
2   mysql> ALTER TABLE innodb_table ENGINE=InnoDB;
3   mysql> INSERT INTO innodb_table SELECT * FROM myisam_table;
```

数据量不大时，这样做挺好。但是推荐迭代时的完成，比如id是主键，每次插入x与y之间的数据，SQL语句如下：

```
1   mysql> START TRANSACTION;
2   mysql> INSERT INTO innodb_table SELECT * FROM myisam_table
3   -> WHERE id BETWEEN x AND y; mysql> COMMIT;
```

注意也可能需要锁住旧表，以免造成了数据不一致。

Percona的工具箱中有 *pt-online-schema-change* ，是一个能修正表结构的错误工具。

MySQL发展史

稍微记住一下下面两张图

Table 1-2. Readonly benchmarks of several MySQL versions

Threads	MySQL 4.1	MySQL 5.0	MySQL 5.1	MySQL 5.1 with InnoDB plugin	MySQL 5.5	MySQL 5.6[a]
1	686	640	596	594	531	526
2	1307	1221	1140	1139	1077	1019
4	2275	2168	2032	2043	1938	1831
8	3879	3746	3606	3681	3523	3320
16	4374	4527	4393	6131	5881	5573
32	4591	4864	4698	7762	7549	7139
64	4688	5078	4910	7536	7269	6994

[a] At the time of our benchmark, MySQL 5.6 was not yet released as GA.

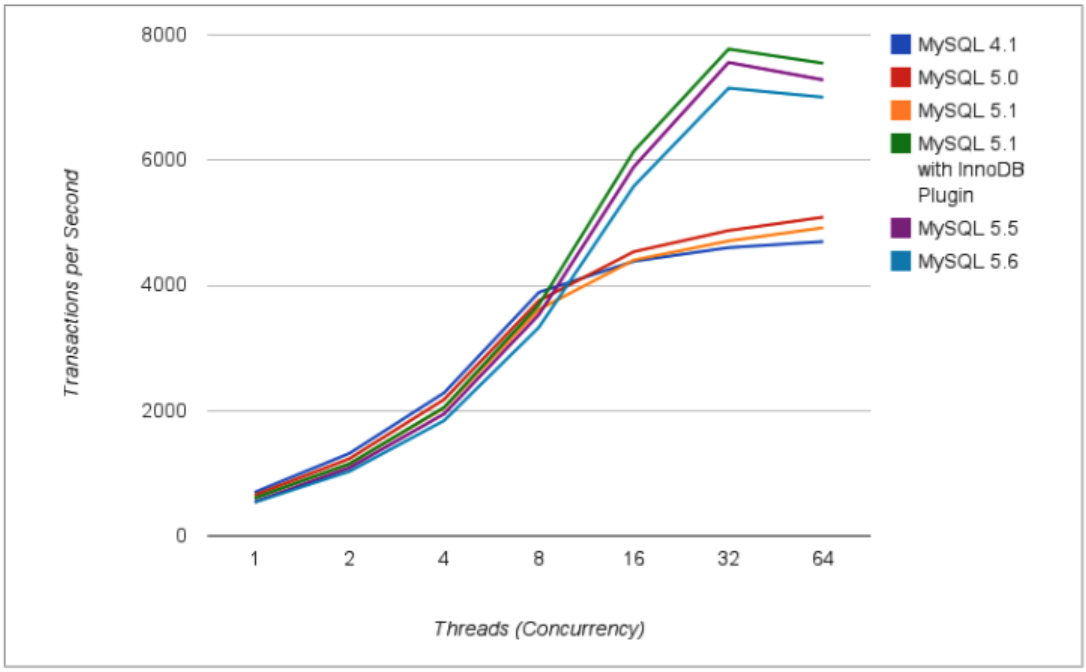

Figure 1-2. Readonly benchmarks of several MySQL versions

测试是在2.5GB的数据。

因为新版本有复杂的SQL语法解析的过程，所以线程数少时，效果没旧版本好。

新版本固然要好些，生产中也不一定总选最新的版本，也可以选择最稳定的版本，因为新版本也许有一些bug。

MySQL的发展模式

没有实质性的，就不翻译了。

总结

MySQL有分层的架构，上层是普通服务和查询解析执行，下层是存储引擎。MySQL执行查询也就是调用了存储引擎的API。

因为历史原因，MySQL处理很多事情都有些奇怪的小技巧。比如执行ALTER TABLE时，InnoDB还是把它当成一个事务来处理，InnoDB存储引擎会把所有语句当成事务。

存储引擎虽多，但是还是InnoDB好。除了超级特殊的用途，别想太多了。

> 不好意思后来慢慢写成总结了，写成总结效率高很多，不用固执的翻译每一句了。

(完)

#MySQL

‹【译】cs231n第四课：理解后向传播

用SQL求得7日均和周均值›