

Mysqli 简介与性能优化

MySQL 的引擎：

- 1、MyISAM
- 2 Innodb
- 3 NDB Cluster
- 4

Innodb 存储引擎简介

在 MySQL 中使用最为广泛的除了 MyISAM 之外，就非 Innodb 莫属了。Innodb 做为第三方公司所开发的存储引擎，和 MySQL 遵守相同的开源 License 协议。

Innodb 之所以能如此受宠，主要是在于其功能方面的较多特点：

1、支持事务安装

Innodb 在功能方面最重要的一点就是对事务安全的支持，这无疑是让 Innodb 成为 MySQL 最为流行的存储引擎之一的一个非常重要原因。而且实现了 SQL92 标准所定义的所有四个级别（READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ 和 SERIALIZABLE）。对事务安全的支持，无疑让很多之前因为特殊业务要求而不得不放弃使用 MySQL 的用户转向支持 MySQL，以及之前对数据库选型持观望态度的用户，也大大增加了对 MySQL 好感。

2、锁定机制的改进

Innodb 改变了 MyISAM 的锁机制，实现了行锁。虽然 Innodb 的行锁机制的实现是通过索引来完成的，但毕竟在数据库中 99% 的 SQL 语句都是要使用索引来做检索数据的。所以，行锁定机制也无疑为 Innodb 在承受高并发压力的环境下增强了不小的竞争力。

影响 MySQL Server 性能的相关因素

- 1 商业需求对性能的影响
- 2 系统架构及实现对性能的影响
- 3 Query 语句对系统性能的影响
- 4 Schema 设计对系统的性能影响
- 5 硬件环境对系统性能的影响

商业需求对性能的影响

只有精确准确的去理解商业需求，才能设计出高性能的系统。

例：

需求：一个论坛帖子总量的统计

附加条件：实时更新

请大家进行设计：

商业需求对性能的影响

第一种： `select count(1) from table`

第二种： 建一张只有一个字段的表来记录帖子的总量，当帖子增加时就将此数据加1

第三种： ? ? ? ? ? 【请大家考虑】

考虑真实要面对的问题：

我们分页时要获取总条数时，能否优化！

系统架构及实现对性能的影响

问题：大家做系统架构的时候，对数据的存储和获取经常是怎么考虑的？

系统架构及实现对性能的影响

我们数据库中存放的数据都是适合在数据库中存放的吗？

几类数据不适合在数据库中存放的：

1. 二进制多媒体数据
2. 流水队列数据
3. 超大文本数据

系统架构及实现对性能的影响

是否合理的利用了应用层 Cache 机制？

什么样的数据适合通过 Cache 技术来提高系统性能：

1. 系统各种配置及规则数据：由于这些配置信息变动的频率非常低，访问概率又很高，所以非常适合存使用 Cache；
2. 活跃用户的基本信息数据：虽然我们经常会听到某某网站的用户量达到成百上千万，但是很少有系统的活跃用户量能够都达到这个数量级。也很少有用户每天没事干去将自己的基本信息改来改去，更为重要的一点是用户的基本信息在应用系统中的访问频率极其频繁。

系统架构及实现对性能的影响

3. 准实时的统计信息数据：所谓准实时的统计数据，实际上就是基于时间段的统计数据。这种数据不会实时更新，也很少需要增量更新
4. 其他一些访问频繁但变更较少的数据：

常见的问题：

- 1、Cache 系统的不合理利用导致 Cache 命中率低下造成数据库访问量的增加，同时也浪费了 Cache 系统的硬件资源投入；
- 2、过度依赖面向对象思想
- 3、对可扩展性的过渡追求，促使系统设计的时候将对象拆得过于离散，造成系统中大量的复杂 Join 语句，而 MySQL Server 在各数据库系统中的主要优势在于处理简单逻辑的查询，这与其锁定的机制也有较大关系；
- 4、对数据库的过渡依赖，将大量更适合存放于文件系统中的数据存入了数据库中，造成数据库资源的浪费，影响到系统的整体性能，如各种日志信息；
- 5、过度理想化系统的用户体验，使大量非核心业务消耗过多的资源，如大量不需要实时更新的数据做了实时统计计算。

Query语句对系统性能的影响

当 MySQL Server 的连接线程接收到 Client 端发送过来的 SQL 请求之后，会经过一系列的分解Parse，进行相应的分析。然后，MySQL 会通过查询优化器模块（Optimizer）根据该 SQL 所涉及的数据表的相关统计信息进行计算分析，然后再得出一个 MySQL 认为最合理最优化的数据访问方式，也就是我们常说的“执行计划”，然后再根据所得到的执行计划通过调用存储引擎借口来获取相应数据。然后再将存储引擎返回的数据进行相关处理，并以 Client 端所要求的格式作为结果集返回给 Client 端的应用程序。

Schema 设计对系统的性能影响

在很多人看来，数据库 Schema 设计是一件非常简单的事情，就大体按照系统设计时候的相关实体对象对应成一个一个的表格基本上就可以了。然后为了在功能上做到尽可能容易扩展，再根据数据库范式规则进行调整，做到第三范式或者第四范式，基本就算完了。

数据库 Schema 设计真的有如上面所说的这么简单么？可以非常肯定的告诉大家，数据库 Schema 设计所需要做的事情远远不止如此。如果您之前的数据库 Schema 设计一直都是这么做的，那么在该设计应用于正式环境之后，很可能带来非常大的性能代价。

Schema 设计对系统的性能影响

--- 根据具体业务去设计数据库，必要时采用冗余字段。

总结：

在整个系统的性能优化中，如果按照百分比来划分上面几个层面的优化带来的性能收益，可以得出大概如下的数据：

需求和架构及业务实现优化：55%

Query 语句的优化：30%

数据库自身的优化：15%

很多时候，大家看到数据库应用系统中性能瓶颈出现在数据库方面，就希望通过数据库的优化来解决问题，但不管 DBA 对数据库多们了解，对 Query 语句的优化多么精通，最终还是很难解决整个系统的性能问题。原因就在于并没有真正找到根本的症结所在。所以，数据库应用系统的优化，实际上是一个需要多方面配合，多方面优化的才能产生根本性改善的事情。简单来说，可以通过下面三句话来简单的概括数据库应用系统的性能优化：商业需求合理化，系统架构最优化，逻辑实现精简化，硬件设施理性化。

MySQL 数据库锁定机制

为了保证数据的一致完整性，任何一个数据库都存在锁定机制。锁定机制的优劣直接应想到一个数据库系统的并发处理能力和性能，所以锁定机制的实现也就成为了各种数据库的核心技术之一。

MySQL 数据库由于其自身架构的特点，存在多种数据存储引擎，每种存储引擎所针对的应用场景特点都不太一样，为了满足各自特定应用场景的需求，每种存储引擎的锁定机制都是为各自所面对的特定场景而优化设计，所以各存储引擎的锁定机制也有较大区别。

MySQL锁定机制简介

- 1 行级锁定 (row-level)
- 2 表级锁定 (table-level)
- 3 页级锁定 (page-level)

行级锁定 (row-level)

行级锁定最大的特点就是锁定对象的颗粒度很小，也是目前各大数据库管理软件所实现的锁定颗粒度最小的。由于锁定颗粒度很小，所以发生锁定资源争用的概率也最小，能够给予应用程序尽可能大的并发处理能力而提高一些需要高并发应用系统的整体性能。

虽然能够在并发处理能力上面有较大的优势，但是行级锁定也因此带来了不少弊端。由于锁定资源的颗粒度很小，所以每次获取锁和释放锁需要做的事情也更多，带来的消耗自然也就更大了。此外，行级锁定也最容易发生死锁。

死锁

规范定义：集合中的每一个进程都在等待只能由本集合中的其他进程才能引发的事件，那么该组进程是死锁的。

例：线程A锁住了记录1并等待记录2，而线程B锁住了记录2并等待记录1，这样两个线程就发生了死锁现象。

注意：死锁与正常阻塞的区别

表级锁定 (table-level)

表级别的锁定是 MySQL 各存储引擎中最大颗粒度的锁定机制。该锁定机制最大的特点是实现逻辑非常简单，带来的系统负面影响最小。所以获取锁和释放锁的速度很快。由于表级锁一次会将整个表锁定，所以可以很好的避免困扰我们的死锁问题。当然，锁定颗粒度大所带来最大的负面影响就是出现锁定资源争用的概率也会最高，致使并大度大打折扣。

页级锁定 (page-level)

页级锁定是 MySQL 中比较独特的一种锁定级别，在其他数据库管理软件中也并不是太常见。页级锁定的特点是锁定颗粒度介于行级锁定与表级锁之间，所以获取锁定所需要的资源开销，以及所能提供的并发处理能力也同样是介于上面二者之间。另外，页级锁定和行级锁定一样，会发生死锁。

探索： MySQL在锁定实现机制中作出修改，允许存储引擎自己改变MySQL通过接口传入的锁定类型而自行决定该怎样锁定数据。

各种锁定机制分析

通过 MyISAM 存储引擎和 Innodb 存储引擎实例演示

各种锁定机制分析

表级锁定：

MySQL 的表级锁定主要分为两种类型，一种是读锁定，另一种是写锁定。在 MySQL 中，主要通过四个队列来维护这两种锁定：两个存放当前正在锁定中的读和写锁定信息，另外两个存放等待中的读写锁定信息，如下：

- Current read-lock queue (lock->read)
- Pending read-lock queue (lock->read_wait)
- Current write-lock queue (lock->write)
- Pending write-lock queue (lock->write_wait)

读锁定

一个新的客户端请求在申请获取读锁定资源的时候，需要满足两个条件：

- 1、 请求锁定的资源当前没有被写锁定；
- 2、 写锁定等待队列（Pending write-lock queue）中没有更高优先级的写锁定等待；

如果上面两个条件中任何一个没有满足，都会被迫进入等待队列 Pending read-lock queue 中等待资源的释放。

写锁定

当客户端请求写锁定的时候，MySQL 首先检查在 `Current write-lock queue` 是否已经有锁定相同资源的信息存在。如果 `Current write-lock queue` 没有，则再检查 `Pending write-lock queue`，如果在 `Pending write-lock queue` 中找到了，自己也需要进入等待队列并暂停自身线程等待锁定资源。反之，如果 `Pending write-lock queue` 为空，则再检测 `Current read-lock queue`，如果有锁定存在，则同样需要进入 `Pending write-lock queue` 等待。**如果遇到特殊的锁定类型时，会直接进入`Current write-lock queue` 中。**

对于表级锁定的演示SQL 详见附件

table-lock_1.sql

table-lock-2.sql

mySql修改表的存储引擎:

<http://blog.csdn.net/shellching/article/details/8106156>

各种锁定机制分析

行级锁定：

Innodb 的行级锁定同样分为两种类型，共享锁和排他锁，而在锁定机制的实现过程中为了让行级锁定和表级锁定共存，**Innodb** 也同样使用了意向锁（表级锁定）的概念，也就有了意向共享锁和意向排他锁这两种。

行级锁定

当一个事务需要给自己需要的某个资源加锁的时候，如果遇到一个共享锁正锁定着自己需要的资源的时候，自己可以再加一个共享锁，不过不能加排他锁。但是，如果遇到自己需要锁定的资源已经被一个排他锁占有之后，则只能等待该锁定释放资源之后自己才能获取锁定资源并添加自己的锁定。而意向锁的作用就是当一个事务在需要获取资源锁定的时候，如果遇到自己需要的资源已经被排他锁占用的时候，该事务可以需要锁定行的表上面添加一个合适的意向锁。如果自己需要一个共享锁，那么就在表上面添加一个意向共享锁。而如果自己需要的是某行（或者某些行）上面 添加一个排他锁的话，则先在表上面添加一个意向排他锁。意向共享锁可以同时并存多个，但是意向排他锁同时只能有一个存在。所以，可以说 **Innodb** 的锁定模式实际上可以分为四种：共享锁（**S**），排他锁（**X**），意向共享锁（**IS**）和意向排他锁（**IX**），我们可以通过以下表格来总结上面这四种所的共存逻辑关系：

行级锁定

	共享锁 (S)	排他锁 (X)	意向共享锁 (IS)	意向排他锁 (IX)
共享锁 (S)	兼容	冲突	兼容	冲突
排他锁 (X)	冲突	冲突	冲突	冲突
意向 共 享 锁 (IS)	兼容	冲突	兼容	兼容
意 向 排 他 锁 (IX)	冲突	冲突	兼容	兼容

行级锁定

Innodb的锁定则是通过在指向数据记录的第一个索引键之前和最后一个索引键之后的空域空间上标记锁定信息而实现的。Innodb 的这种锁定实现方式被称为“**NEXT-KEY locking**”（间隙锁），因为 Query 执行过程中通过过范围查找的华，他会锁定整个范围内所有的索引键值，即使这个键值并不存在。

间隙锁有一个比较致命的弱点，就是当锁定一个范围键值之后，即使某些不存在的键值也会被无辜的锁定，而造成在锁定的时候无法插入锁定键值范围内的任何数据。在某些场景下这可能会对性能造成很大的危害。而 Innodb 给出的解释是为了组织幻读的出现，所以他们选择的间隙锁来实现锁定。

行级锁定

间隙锁给 Innodb 带来性能的负面影响之外，通过索引实现锁定的方式还存在其他几个较大的性能隐患：

- 1 当 Query 无法利用索引的时候，Innodb 会放弃使用行级别锁定而改用表级别的锁定，造成并发性能的降低。
- 2 当 Quuery 使用的索引并不包含所有过滤条件的时候，数据检索使用到的索引键所只想的数据可能有部分并不属于该 Query 的结果集的行列，但是也会被锁定，因为间隙锁锁定的是一个范围，而不是具体的索引键；
- 3 当 Query 在使用索引定位数据的时候，如果使用的索引键一样但访问的数据行不同的时候（索引只是过滤条件的一部分），一样会被锁定

合理利用锁机制优化 MySQL

Innodb 存储引擎由于实现了行级锁定，虽然在锁定机制的实现方面所带来的性能损耗可能比表级锁定会要更高一些，但是在整体并发处理能力方面要远远优于 **MyISAM** 的表级锁定的。当系统并发量较高的时候，**Innodb** 的整体性能和 **MyISAM** 相比就会有比较明显的优势了。但是，**Innodb** 的行级锁定同样也有其脆弱的一面，当我们使用不当的时候，可能会让 **Innodb** 的整体性能表现不仅不能比 **MyISAM** 高，甚至可能会更差。

合理利用 Innodb 的行级锁定，做到扬长避短：

- a) 尽可能让所有的数据检索都通过索引来完成，从而避免 Innodb 因为无法通过索引键加锁而升级为表级锁定；
- b) 合理设计索引，让 Innodb 在索引键上面加锁的时候尽可能准确，尽可能的缩小锁定范围，避免造成不必要的锁定而影响其他 Query 的执行；
- c) 尽可能减少基于范围的数据检索过滤条件，避免因为间隙锁带来的负面影响而锁定了不该锁定的记录；
- d) 尽量控制事务的大小，减少锁定的资源量和锁定时间长度；
- e) 在业务环境允许的情况下，尽量使用较低级别的事务隔离，以减少 MySQL 因为实现事务隔离级别所带来的附加成本；

合理利用 Innodb 的行级锁定，做到扬长避短：

- a) 尽可能让所有的数据检索都通过索引来完成，从而避免 Innodb 因为无法通过索引键加锁而升级为表级锁定；
- b) 合理设计索引，让 Innodb 在索引键上面加锁的时候尽可能准确，尽可能的缩小锁定范围，避免造成不必要的锁定而影响其他 Query 的执行；
- c) 尽可能减少基于范围的数据检索过滤条件，避免因为间隙锁带来的负面影响而锁定了不该锁定的记录；
- d) 尽量控制事务的大小，减少锁定的资源量和锁定时间长度；
- e) 在业务环境允许的情况下，尽量使用较低级别的事务隔离，以减少 MySQL 因为实现事务隔离级别所带来的附加成本；

减少死锁产生概率的小建议：

- a) 类似业务模块中，尽可能按照相同的访问顺序来访问，防止产生死锁；
- b) 在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁产生概率；
- c) 对于非常容易产生死锁的业务部分，可以尝试使用升级锁定颗粒度，通过表级锁定来减少死锁产生的概率；

系统锁定争用情况查询

```
SQL :show status like 'table%'
```

这里有两个状态变量记录 MySQL 内部表级锁定的情况，两个变量说明如下：

- 1 **Table_locks_immediate**: 产生表级锁定的次数；
- 2 **Table_locks_waited**: 出现表级锁定争用而发生等待的次数；

系统锁定争用情况查询

Innodb 所使用的行级锁定，系统中是通过另外一组更为详细的状态变量来记录的。这里有两个状态变量记录 MySQL 内部表级锁定的情况，两个变量说明如下：

SQL: `show status like 'innodb_row_lock%'`

`Innodb_row_lock_current_waits`: 当前正在等待锁定的数量；

`Innodb_row_lock_time`: 从系统启动到现在锁定总时间长度；

`Innodb_row_lock_time_avg`: 每次等待所花平均时间；

`Innodb_row_lock_time_max`: 从系统启动到现在等待最常的一次所花的时间；

`Innodb_row_lock_waits`: 系统启动后到现在总共等待的次数；