第 6 章 影响 MySQL Server 性能的相关因素

前言：

大部分人都一致认为一个数据库应用系统（这里的数据库应用系统概指所有使用数据库的系统）的性能瓶颈最容易出现在数据的操作方面，而数据库应用系统的大部分数据操作都是通过数据库管理软件所提供的相关接口来完成的。所以数据库管理软件也就很自然的成为了数据库应用系统的性能瓶颈所在，这是当前业界比较普遍的一个看法。但我们的应用系统的性能瓶颈真的完全是因为数据库管理软件和数据库主机自身造成的吗？我们将通过本章的内容来进行一个较为深入的分析，让大家了解到一个数据库应用系统的性能到底与哪些地方有关，让大家寻找出各自应用系统的出现性能问题的根本原因，而尽可能清楚的知道该如何去优化自己的应用系统。

考虑到本书的数据库对象是 MySQL，而 MySQL 最多的使用场景是 WEB 应用，那么我们就以一个 WEB 应用系统为例，逐个分析其系统构成，结合笔者在大型互联网公司从事 DBA 工作多年的经验总结，分析出数据库应用系统中各个环境对性能的影响。

6.1 商业需求对性能的影响

应用系统中的每一个功能在设计初衷肯定都是出于为用户提供某种服务，或者满足用户的某种需求，但是，并不是每一个功能在最后都能很成功，甚至有些功能的推出可能在整个系统中是画蛇添足。不仅没有为用户提高任何体验度，也没有为用户改进多少功能易用性，反而在整个系统中成为一个累赘，带来资源的浪费。

不合理需求造成资源投入产出比过低

需求是否合理很多时候可能并不是很容易界定，尤其是作为技术人员来说，可能更难以确定一个需求的合理性。即使指出，也不一定会被产品经历们认可。那作为技术人员的我们怎么来证明一个需求是否合理呢？

第一、每次产品经理们提出新的项目（或者功能需求）的时候，应该要求他们同时给出该项目的预期收益的量化指标，以备项目上先后统计评估投入产出比率；

第二、在每次项目进行过程中，应该详细记录所有的资源投入，包括人力投入，硬件设施的投入，以及其他任何项目相关的资源投入；

第三、项目（或者功能需求）上线之后应该及时通过手机相关数据统计出项目的实际收益值，以便计算投入产出比率的时候使用；

第四、技术部门应该尽可能推动设计出一个项目（或者功能需求）的投入产出比率的计算规则。在项目上线一段时间之后，通过项目实际收益的统计数据和项目的投入资源量，计算出整个项目的实际投入产出值，并公布给所有参与项目的部门知晓，同时存放以备后查。

有了实际的投入产出比率，我们就可以和项目立项之初产品经理们的预期投入产出比率做出比较，判定出这个项目做的是否值得。而且当积累了较多的项目投入产出比率之后，我们可以根据历史数据分

析出一个项目合理的投入产出比率应该是多少。这样，在项目立项之初，我们就可以判定出产品经理们的预期投入产出比率是否合理，项目是否真的有进行的必要。

有了实际的投入产出比率之后，我们还可以拿出数据给老板们看，让他知道功能并不是越多越好，让他知道有些功能是应该撤下来的，即使撤下该功能可能需要投入不少资源。

实际上，一般来说，在产品开发及运营部门内部都会做上面所说的这些事情的。但很多时候可能更多只是一种形式化的过程。在有些比较规范的公司可能也完成了上面的大部分流程，但是要么数据不公开，要么公开给其他部门的数据存在一定的偏差，不具备真实性。

为什么会这样？其实就一个原因，就是部门之间的利益冲突及业绩冲突问题。产品经理们总是希望尽可能的让用户觉得自己设计的产品功能齐全，让老板觉得自己做了很多事情。但是从来都不会去关心因为做一个功能所带来的成本投入，或者说是不会特别的关心这一点。而且很多时候他们也并不能太理解技术方面带来的复杂度给产品本身带来的负面影响。

这里我们就拿一个看上去很简单的功能来分析一下。
需求：一个论坛帖子总量的统计
附加要求：实时更新

在很多人看来，这个功能非常容易实现，不就是执行一条 SELECT COUNT(*)的 Query 就可以得到结果了么？是的，确实只需要如此简单的一个 Query 就可以得到结果。但是，如果我们采用不是 MyISAM 存储引擎，而是使用的 Innodb 的存储引擎，那么大家可以试想一下，如果存放帖子的表中已经有上千万的帖子的时候，执行这条 Query 语句需要多少成本？恐怕再好的硬件设备，恐怕都不可能在 10 秒之内完成一次查询吧。如果我们的访问量再大一点，还有人觉得这是一件简单的事情么？

既然这样查询不行，那我们是不是该专门为这个功能建一个表，就只有一个字段，一条记录，就存放这个统计量，每次有新的帖子产生的时候，都将这个值增加 1，这样我们每次都只需要查询这个表就可以得到结果了，这个效率肯定能够满足要求了。确实，查询效率肯定能够满足要求，可是如果我们的系统帖子产生很快，在高峰时期可能每秒就有几十甚至上百个帖子新增操作的时候，恐怕这个统计表又要成为大家的噩梦了。要么因为并发的问题造成统计结果的不准确，要么因为锁资源争用严重造成整体性能的大幅度下降。

其实这里问题的焦点不应该是实现这个功能的技术细节，而是在于这个功能的附加要求“实时更新”上面。当一个论坛的帖子数量很大了之后，到底有多少人会关注这个统计数据是否是实时变化的？有多少人在乎这个数据在短时间内的不精确性？我想恐怕不会有人会傻傻的盯着这个统计数字并追究当自己发了一个帖子然后回头刷新页面发现这个统计数字没有加 1 吧？即使明明白白的告诉用户这个统计数据是每过多长时间段更新一次，那有怎样？难道会有很多用户就此很不爽么？

只要去掉了这个“实时更新”的附加条件，我们就可以非常容易的实现这个功能了。就像之前所提到的那样，通过创建一个统计表，然后通过一个定时任务每隔一定时间段去更新一次里面的统计值，这样既可以解决统计值查询的效率问题，又可以保证不影响新发贴的效率，一举两得。

实际上，在我们应用的系统中还有很多很多类似的功能点可以优化。如某些场合的列表页面参与列表的数据量达到一个数量级之后，完全可以不用准确的显示这个列表总共有多少条信息，总共分了多少

页，而只需要一个大概的估计值或者一个时间段之前的统计值。这样就省略了我们的分页程序需要在分以前实时 COUNT 出满足条件的记录数。

其实，在很多应用系统中，实时和准实时，精确与基本准确，在很多地方所带来的性能消耗可能是几个性能的差别。在系统性能优化中，应该尽量分析出那些可以不实时和不完全精确的地方，作出一些相应的调整，可能会给大家带来意想不到的巨大性能提升。

无用功能堆积使系统过度复杂影响整体性能

很多时候，为系统增加某个功能可能并不需要花费太多的成本，而要想将一个已经运行了一段时间的功能从原有系统中撤下来却是非常困难的。

首先，对于开发部门，可能要重新整理很多的代码，找出可能存在与增加该功能所编写的代码有交集的其他功能点，删除没有关联的代码，修改有关联的代码；

其次，对于测试部门，由于功能的变动，必须要回归测试所有相关的功能点是否正常。可能由于界定困难，不得不将回归范围扩展到很大，测试工作量也很大。

最后，所有与撤除下线某个功能相关的工作参与者来说，又无法带来任何实质性的收益，而恰恰相反是，带来的只可能是风险。

由于上面的这几个因素，可能很少有公司能够有很完善的项目（或者功能）下线机制，也很少有公司能做到及时将系统中某些不合适的功能下线。所以，我们所面对的应用系统可能总是越来越复杂，越来越庞大，短期内的复杂可能并无太大问题，但是随着时间的积累，我们所面对的系统就会变得极其臃肿。不仅维护困难，性能也会越来越差。尤其是有些并不合理的功能，在设计之初或者是刚上线的时候由于数据量较小，带来不了多少性能损耗。可随着时间的推移，数据库中的数据量越来越大，数据检索越来越困难，对真个系统带来的资源消耗也就越来越大。

而且，由于系统复杂度的不断增加，给后续其他功能的开发带来实现的复杂度，可能很多本来很简单的功能，因为系统的复杂而不得不增加很多的逻辑判断，造成系统应用程序的计算量不断增加，本身性能就会受到影响。而如果这些逻辑判断还需要与数据库交互通过持久化的数据来完成的话，所带来的性能损失就更大，对整个系统的性能影响也就更大了。

6.2 系统架构及实现对性能的影响

一个 WEB 应用系统，自然离不开 Web 应用程序（Web App）和应用程序服务器（App Server）。App Server 我们能控制的内容不多，大多都是使用已经久经考验的成熟产品，大家能做的也就只是通过一些简单的参数设置调整来进行调优，不做细究。而 Web App 大部分都是各自公司根据业务需求自行开发，可控性要好很多。所以我们从 Web 应用程序着手分析一个应用程序架构的不同设计对整个系统性能的影响将会更合适。

上一节中商业需求告诉了我们一个系统应该有什么不应该有什么，系统架构则则决定了我们系统的构建环境。就像修建一栋房子一样，在清楚了这栋房子的用途之后，会先有建筑设计师来画出一章基本

的造型图，然后还需要结构设计师为我们设计出结构图。系统架构设计的过程就和结构工程好似设计结构图一样，需要为整个系统搭建出一个尽可能最优的框架，让整个系统能够有一个稳定高效的结构体系让我们实现各种商业需求。

谈到应用系统架构的设计，可能有人的心里会开始嘀咕，一个 DBA 有什么资格谈论人家架构师（或者程序员）所设计的架构？其实大家完全没有必要这样去考虑，我们谈论架构只是分析各种情形下的性能消耗区别，仅仅是根据自己的专业特长来针对相应架构给出我们的建议及意见，并不是要批判架构整体的好坏，更不是为了推翻某个架构。而且我们所考虑的架构大多数时候也只是数据层面相关的架构。

我们数据库中存放的数据都是适合在数据库中存放的吗？

对于有些开发人员来说，数据库就是一个操作最方便的万能存储中心，希望什么数据都存放在数据库中，不论是需要持久化的数据，还是临时存放的过程数据，不论是普通的纯文本格式的字符数据，还是多媒体的二进制数据，都喜欢全部塞如数据库中。因为对于应用服务器来说，数据库很多时候都是一个集中式的存储环境，不像应用服务器那样可能有很多台；而且数据库有专门的 DBA 去帮忙维护，而不像应用服务器很多时候还需要开发人员去做一些维护；还有一点很关键的就是数据库的操作非常简单统一，不像文件操作或者其他类型的存储方式那么复杂。

其实我个人认为，现在的很多数据库为我们提供了太多的功能，反而让很多并不是太了解数据库的人错误的使用了数据库的很多并不是太擅长或者对性能影响很大的功能，最后却全部怪罪到数据库身上。

实际上，以下几类数据都是不适合在数据库中存放的：

1. 二进制多媒体数据

   将二进制多媒体数据存放在数据库中，一个问题是数据库空间资源耗用非常严重，另一个问题是这些数据的存储很消耗数据库主机的 CPU 资源。这种数据主要包括图片，音频、视频和其他一些相关的二进制文件。这些数据的处理本不是数据的优势，如果我们硬要将他们塞入数据库，肯定会造成数据库的处理资源消耗严重。

2. 流水队列数据

   我们都知道，数据库为了保证事务的安全性（支持事务的存储引擎）以及可恢复性，都是需要记录所有变更的日志信息的。而流水队列数据的用途就决定了存放这种数据的表中的数据会不断的被 INSERT，UPDATE 和 DELETE，而每一个操作都会生成与之对应的日志信息。在 MySQL 中，如果是支持事务的存储引擎，这个日志的产生量更是要翻倍。而如果我们通过一些成熟的第三方队列软件来实现这个 Queue 数据的处理功能，性能将会成倍的提升。

3. 超大文本数据

   对于 5.0.3 之前的 MySQL 版本，VARCHAR 类型的数据最长只能存放 255 个字节，如果需要存储更长的文本数据到一个字段，我们就必须使用 TEXT 类型（最大可存放 64KB）的字段，甚至是更大的 LONGTEXT 类型（最大 4GB）。而 TEXT 类型数据的处理性能要远比 VARCHAR 类型数据的处理性能低下很多。从 5.0.3 版本开始，VARCHAR 类型的最大长度被调整到 64KB 了，但是当实际数据小于 255 Bytes 的时候，实际存储空间和实际的数据长度一样，可一旦长度超过 255 Bytes 之后，所占用的存储空间就是实际数据长度的两倍。

   所以，超大文本数据存放在数据库中不仅会带来性能低下的问题，还会带来空间占用的浪费问题。

是否合理的利用了应用层 Cache 机制？

对于 Web 应用，活跃数据的数据量总是不会特别的大，有些活跃数据更是很少变化。对于这类数据，我们是否有必要每次需要的时候都到数据库中去查询呢？如果我们能够将变化相对较少的部分活跃数据通过应用层的 Cache 机制 Cache 到内存中，对性能的提升肯定是成数量级的，而且由于是活跃数据，对系统整体的性能影响也会很大。

当然，通过 Cache 机制成功的案例数不胜数，但是失败的案例也同样并不少见。如何合理的通过 Cache 技术让系统性能得到较大的提升也不是通过寥寥几笔就能说明的清楚，这里我仅根据以往的经验列举一下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。所以用户基本信息的 Cache，很容易让整个应用系统的性能出现一个质的提升。
3. 活跃用户的个性化定制信息数据；
   虽然用户个性化定制的数据从访问频率来看，可能并没有用户的基本信息那么的频繁，但相对于系统整体来说，也占了很大的比例，而且变更皮律一样不会太多。从 Ebay 的 PayPal 通过 MySQL 的 Memory 存储引擎实现用户个性化定制数据的成功案例我们就能看出对这部分信息进行 Cache 的价值了。虽然通过 MySQL 的 Memory 存储引擎并不像我们传统意义层面的 Cache 机制，但正是对 Cache 技术的合理利用和扩充造就了项目整体的成功。
4. 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，只有当达到重新 Build 该统计数据的时候需要做一次全量更新操作。虽然这种数据即使通过数据库来读取效率可能也会比较高，但是执行频率很高之后，同样会消耗不少资源。既然数据库服务器的资源非常珍贵，我们为什么不能放在应用相关的内存 Cache 中呢？
5. 其他一些访问频繁但变更较少的数据；
   出了上面这四种数据之外，在我们面对的各种系统环境中肯定还会有各种各样的变更较少但是访问很频繁的数据。只要合适，我们都可以将对他们的访问从数据库移到 Cache 中。

我们的数据层实现都是最精简的吗？

从以往的经验来看，一个合理的数据存取实现和一个拙劣的实现相比，在性能方面的差异经常会超出一个甚至几个数量级。我们先来分析一个非常简单且经常会遇到类似情况的示例：

在我们的示例网站系统中，现在要实现每个用户查看各自相册列表（假设每个列表显示 10 张相片）的时候，能够在相片名称后面显示该相片的留言数量。这个需求大家认为应该如何实现呢？我想 90%的开发开发工程师会通过如下两步来实现该需求：

1、通过“SELECT id,subject,url FROM photo WHERE user_id = ? limit 10” 得到第一页的相片

相关信息；

2、通过第 1 步结果集中的 10 个相片 id 循环运行十次“SELECT COUNT(*) FROM photo_comment WHERE photh_id = ?” 来得到每张相册的回复数量然后再瓶装展现对象。

此外可能还有部分人想到了如下的方案：

1、和上面完全一样的操作步骤；

2、通过程序拼装上面得到的 10 个 photo 的 id，再通过 in 查询“SELECT photo_id,count(*) FROM photo_comment WHERE photo_id in (?) GROUP BY photo_id” 一次得到 10 个 photo 的所有回复数量，再组装两个结果集得到展现对象。

我们来对以上两个方案做一下简单的比较：

1、从 MySQL 执行的 SQL 数量来看 ，第一种解决方案为 11（1+10=11）条 SQL 语句，第二种解决方案为 2 条 SQL 语句（1+1）；

2、从应用程序与数据库交互来看，第一种为 11 次，第二种为 2 次；

3、从数据库的 IO 操作来看，简单假设每次 SQL 为 1 个 IO，第一种最少 11 次 IO，第二种小于等于 11 次 IO，而且只有当数据非常之离散的情况下才会需要 11 次；

4、从数据库处理的查询复杂度来看，第一种为两类很简单的查询，第二种有一条 SQL 语句有 GROUP BY 操作，比第一种解决方案增加了了排序分组操作；

5、从应用程序结果集处理来看，第一种 11 次结果集的处理，第二中 2 次结果集的处理，但是第二种解决方案中第二词结果处理数量是第一次的 10 倍；

6、从应用程序数据处理来看，第二种比第一种多了一个拼装 photo_id 的过程。

我们先从以上 6 点来做一个性能消耗的分析：

1、由于 MySQL 对客户端每次提交的 SQL 不管是相同还是不同，都需要进行完全解析，这个动作主要消耗的资源是数据库主机的 CPU，那么这里第一种方案和第二种方案消耗 CPU 的比例是 11:2。SQL 语句的解析动作在整个 SQL 语句执行过程中的整体消耗的 CPU 比例是较多的;

2、应用程序与数据库交互所消耗的资源基本上都在网络方面，同样也是 11：2；

3、数据库 IO 操作资源消耗为小于或者等于 1：1；

4、第二种解决方案需要比第一种多消耗内存资源进行排序分组操作，由于数据量不大，多出的消耗在语句整体消耗中占用比例会比较小，大概不会超过 20%，大家可以针对性测试；

5、结果集处理次数也为 11：2，但是第二中解决方案第二次处理数量较大，整体来说两次的性能消耗区别不大；

6、应用程序数据处理方面所多出的这个 photo_id 的拼装所消耗的资源是非常小的，甚至比应用程序与 MySQL 做一次简单的交互所消耗的资源还要少。

综合上面的这 6 点比较，我们可以很容易得出结论，从整体资源消耗来看，第二中方案会远远优于第一种解决方案。而在实际开发过程中，我们的程序员却很少选用。主要原因其实有两个，一个是第二种方案在程序代码实现方面可能会比第一种方案略为复杂，尤其是在当前编程环境中面向对象思想的普及，开发工程师可能会更习惯于以对象为中心的思考方式来解决问题。还有一个原因就是我们的程序员可能对 SQL 语句的使用并不是特别的熟悉，并不一定能够想到第二条 SQL 语句所实现的功能。对于第一个原因，我们可能只能通过加强开发工程师的性能优化意识来让大家能够自觉纠正，而第二个原因的解决就正是需要我们出马的时候了。 SQL 语句正是我们的专长，定期对开发工程师进行一些相应的数据库知识包括 SQL 语句方面的优化培训，可能会给大家带来意想不到的收获的。

这里我们还仅仅只是通过一个很长见的简单示例来说明数据层架构实现的区别对整体性能的影响，实际上可以简单的归结为过渡依赖嵌套循环的使用或者说是过渡弱化 SQL 语句的功能造成性能消耗过多的实例。后面我将进一步分析一下更多的因为架构实现差异所带来的性能消耗差异。

过度依赖数据库 SQL 语句的功能造成数据库操作效率低下

前面的案例是开发工程师过渡弱化 SQL 语句的功能造成的资源浪费案例，而这里我们再来分析一个完全相反的案例：在群组简介页面需要显示群名称和简介，每个群成员的nick_name，以及群主的个人签名信息。

需求中所需信息存放在以下四个表中：user，user_profile，groups，user_group

我们先看看最简单的实现方法，一条 SQL 语句搞定所有事情：

```
SELECT name,description,user_type,nick_name,sign
FROM groups,user_group,user ,user_profile
WHERE groups.id = ?
    AND groups.id = user_group.group_id
    AND user_group.user_id = user.id
    AND user_profile.user_id = user.id
```

当然我们也可以通过如下稍微复杂一点的方法分两步搞定：

首先取得所有需要展示的 group 的相关信息和所有群组员的 nick_name 信息和组员类别：

```
SELECT name,description,user_type,nick_name
FROM groups,user_group,user
WHERE groups.id = ?
    AND groups.id = user_group.group_id
    AND user_group.user_id = user.id
```

然后在程序中通过上面结果集中的 user_type 找到群主的 user_id 再到 user_profile 表中取得群主的签名信息：

```
SELECT sign FROM user_profile WHERE user_id = ?
```

大家应该能够看出两者的区别吧，两种解决方案最大的区别在于交互次数和 SQL 复杂度。而带来的实际影响是第一种解决方案对 user_profile 表有不必要的访问（非群主的 profile 信息），造成 IO 访问的直接增加在 20%左右。而大家都知道，IO 操作在数据库应用系统中是非常昂贵的资源。尤其是当这个功能的 PV 较大的时候，第一种方案造成的 IO 损失是相当大的。

重复执行相同的 SQL 造成资源浪费

这个问题其实是每个人都非常清楚也完全认同的一个问题，但是在应用系统开发过程中，仍然会常有这样的现象存在。究其原因，主要还是开发工程师思维中面向对象的概念太过深入，以及为了减少自己代码开发的逻辑和对程序接口过度依赖所造成的。

我曾经在一个性能优化项目中遇到过一个案例，某个功能页面一侧是“分组”列表，是一列“分

组”的名字。页面主要内容则是该“分组”的所有“项目”列表。每个“项目”以名称（或者图标）显示，同时还有一个 SEO 相关的需求就是每个“项目”名称的链接地址中是需要有“分组”的名称的。所以在“项目”列表的每个 “项目”的展示内容中就需要得到该项目所属的组的名称。按照开发工程师开发思路，非常容易产生取得所有“项目”结果集并映射成相应对象之后，再从对象集中获取“项目”所属组的标识字段，然后循环到“分组”表中取得需要的”组名“。然后再将拼装成展示对象。

看到这里，我想大家应该已经知道这里存在的一个最大的问题就是多次重复执行了完全相同的 SQL 得到完全相同的内容。同时还犯了前面第一个案例中所犯的错误。或许大家看到之后会不相信有这样的案例存在，我可以非常肯定的告诉大家，事实就是这样。同时也请大家如果有条件的话，好好 Review 自己所在的系统的代码，非常有可能同样存在上面类似的情形。

还有部分解决方案要远优于上面的做法，那就是不循环去取了，而是通过 Join 一次完成，也就是解决了第一个案例所描述的性能问题。但是又误入了类似于第二个案例所描述的陷阱中了，因为实际上他只需要一次查询就可以得到所有“项目”所属的“分组”的名称（所有项目都是同一个组的）。

当然，也有部分解决方案也避免了第二个案例的问题，分为两条 SQL，两步完成了这个需求。这样在性能上面基本上也将近是数量级的提升了。

但是这就是性能最优的解决方案了么？不是的，我们甚至可以连一次都不需要访问就获得所需要的“分组”名称。首先，侧栏中的“分组”列表是需要有名称的，我们为什么不能直接利用到呢？

当然，可能有些系统的架构决定了侧栏和主要内容显示区来源于不同的模板（或者其他结构），那么我们也完全可以通过在进入这个功能页面的链接请求中通过参数传入我们需要的“分组”名称。这样我们就可以完全不需要根据“项目”相关信息去数据库获取所属“分组”的信息，就可以完成相应需求了。当然，是否需要通过请求参数来节省最后的这一次访问，可能会根据这个功能页面的 PV 来决定，如果访问并不是非常频繁，那么这个节省可能并不是很明显，而应用系统的复杂度却有所增加，而且程序看上去可能也会不够优雅，但是如果访问非常频繁的场景中，所节省的资源还是比较可观的。

上面还仅仅只是列举了我们平时比较常见的一些实现差异对性能所带来的影响，除了这些实现方面所带来的问题之外，应用系统的整体架构实现设计对系统性能的影响可能会更严重。下面大概列举了一些较为常见的架构设计实现不当带来的性能问题和资源浪费情况。

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，对系统

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。

以上仅仅是一些比较常见的症结，在各种不同的应用环境中肯定还会有很多不同的性能问题，可能

需要大家通过仔细的数据分析和对系统的充分了解才能找到，但是一旦找到症结所在，通过相应的优化措施，所带来的收益也是相当可观的。

6.3 Query 语句对系统性能的影响

前面一节我们介绍了应用系统的实现差异对数据库应用系统整体性能的影响，这一节我们将分析 SQL 语句的差异对系统性能的影响。

我想对于各位读者来说，肯定都清楚 SQL 语句的优劣是对性能有影响的，但是到底有多大影响可能每个人都会有不同的体会，每个 SQL 语句在优化之前和优化之后的性能差异也是各不相同，所以对于性能差异到底有多大这个问题我们我们这里就不做详细分析了。我们重点分析实现同样功能的不同 SQL 语句在性能方面会产生较大的差异的根本原因，并通过一个较为典型的示例来对我们的分析做出相应的验证。

为什么返回完全相同结果集的不同 SQL 语句，在执行性能方面存在差异呢？这里我们先从 SQL 语句在数据库中执行并获取所需数据这个过程来做一个大概的分析了。

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 所设涉及到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。

注：这里所说的统计数据，是我们通过 ANALYZE TABLE 命令通知 MySQL 对表的相关数据做分析之后所获得到的一些数据统计量。这些统计数据对 MySQL 优化器而言是非常重要的，优化器所生成的执行计划的好坏，主要就是由这些统计数据所决定的。实际上，在其他一些数据库管理软件中也有类似相应的统计数据。

我们都知道，在数据库管理软件中，最大的性能瓶颈就是在于磁盘 IO，也就是数据的存取操作上面。而对于同一份数据，当我们以不同方式去寻找其中的某一点内容的时候，所需要读取的数据量可能会有天壤之别，所消耗的资源也自然是区别甚大。所以，当我们需要从数据库中查询某个数据的时候，所消耗资源的多少主要就取决于数据库以一个什么样的数据读取方式来完成我们的查询请求，也就是取决于 SQL 语句的执行计划。

对于唯一一个 SQL 语句来说，经过 MySQL Parse 之后分解的结构都是固定的，只要统计信息稳定，其执行计划基本上都是比较固定的。而不同写法的 SQL 语句，经过 MySQL Parse 之后分解的结构结构就可能完全不同，即使优化器使用完全一样的统计信息来进行优化，最后所得出的执行计划也可能完全不一样。而执行计划又是决定一个 SQL 语句最终的资源消耗量的主要因素。所以，实现功能完全一样的 SQL 语句，在性能上面可能会有差别巨大的性能消耗。当然，如果功能一样，而且经过 MySQL 的优化器优化之后的执行计划也完全一致的不同 SQL 语句在资源消耗方面可能就相差很小了。当然这里所指的消耗主要是 IO 资源的消耗，并不包括 CPU 的消耗。

下面我们将通过一两个具体的示例来分析写法不一样而功能完全相同的两条 SQL 的在性能方面的差异。

示例一

需求：取出某个 group（假设 id 为 100）下的用户编号（id），用户昵称（nick_name）、用户性别（sexuality）、用户签名（sign）和用户生日（birthday），并按照加入组的时间（user_group.gmt_create）来进行倒序排列，取出前 20 个。

解决方案一、

```
SELECT id,nick_name
FROM user,user_group
WHERE user_group.group_id = 1
    and user_group.user_id = user.id
limit 100,20;
```

解决方案二、

```
SELECT user.id,user.nick_name
FROM (
    SELECT user_id
    FROM user_group
    WHERE user_group.group_id = 1
    ORDER BY gmt_create desc
    limit 100,20) t,user
WHERE t.user_id = user.id;
```

我们先来看看执行计划：

```
sky@localhost : example 10:32:13> explain
    -> SELECT id,nick_name
    ->     FROM user,user_group
    ->     WHERE user_group.group_id = 1
    ->         and user_group.user_id = user.id
    ->     ORDER BY user_group.gmt_create desc
    ->     limit 100,20\G
    *************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: user_group
         type: ref
possible_keys: user_group_uid_gid_ind,user_group_gid_ind
          key: user_group_gid_ind
      key_len: 4
          ref: const
         rows: 31156
```

```
        Extra: Using where; Using filesort
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: user
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: example.user_group.user_id
         rows: 1
        Extra:

sky@localhost : example 10:32:20> explain
    ->  SELECT user.id,user.nick_name
    ->     FROM (
    ->         SELECT user_id
    ->         FROM user_group
    ->         WHERE user_group.group_id = 1
    ->         ORDER BY gmt_create desc
    ->         limit 100,20) t,user
    ->     WHERE t.user_id = user.id\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: <derived2>
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 20
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: PRIMARY
        table: user
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: t.user_id
         rows: 1
        Extra:
```

```
*************************** 3. row ***************************
           id: 2
  select_type: DERIVED
        table: user_group
         type: ref
possible_keys: user_group_gid_ind
          key: user_group_gid_ind
      key_len: 4
          ref: const
         rows: 31156
        Extra: Using filesort
```

执行计划对比分析：

解决方案一中的执行计划显示MySQL在对两个参与Join的表都利用到了索引，user_group表利用了user_group_gid_ind索引（key: user_group_gid_ind），user表利用到了主键索引（key: PRIMARY），在参与Join前MySQL通过Where过滤后的结果集与user表进行Join，最后通过排序取出Join后结果的“limit 100,20”条结果返回。

解决方案二的SQL语句利用到了子查询，所以执行计划会稍微复杂一些，首先可以看到两个表都和解决方案1一样都利用到了索引（所使用的索引也完全一样），执行计划显示该子查询以user_group为驱动，也就是先通过user_group进行过滤并马上进行这一论的结果集排序，也就取得了SQL中的“limit 100,20”条结果，然后与user表进行Join，得到相应的数据。这里可能有人会怀疑在自查询中从user_group表所取得与user表参与 Join的记录条数并不是20条，而是整个group_id=1的所有结果。那么清大家看看该执行计划中的第一行，该行内容就充分说明了在外层查询中的所有的20条记录全部被返回。

通过比较两个解决方案的执行计划，我们可以看到第一中解决方案中需要和user表参与Join的记录数MySQL通过统计数据估算出来是31156，也就是通过user_group表返回的所有满足group_id=1的记录数（系统中的实际数据是20000）。而第二种解决方案的执行计划中，user 表参与Join的数据就只有20条，两者相差很大，通过本节最初的分析，我们认为第二中解决方案应该明显优于第一种解决方案。

下面我们通过对比两个解决觉方案的SQL实际执行的profile详细信息，来验证我们上面的判断。由于SQL语句执行所消耗的最大两部分资源就是IO和CPU，所以这里为了节约篇幅，仅列出BLOCK IO和CPU两项profile信息（Query Profiler 的详细介绍将在后面章节中独立介绍）：

先打开profiling功能，然后分别执行两个解决方案的SQL语句：

```
sky@localhost : example 10:46:43> set profiling = 1;
Query OK, 0 rows affected (0.00 sec)

sky@localhost : example 10:46:50> SELECT id,nick_name
    ->      FROM user,user_group
    ->      WHERE user_group.group_id = 1
    ->          and user_group.user_id = user.id
    ->      ORDER BY user_group.gmt_create desc
```

```
    ->     limit 100,20;
+--------+-----------+
| id     | nick_name |
+--------+-----------+
| 990101 | 990101    |
| 990102 | 990102    |
| 990103 | 990103    |
| 990104 | 990104    |
| 990105 | 990105    |
| 990106 | 990106    |
| 990107 | 990107    |
| 990108 | 990108    |
| 990109 | 990109    |
| 990110 | 990110    |
| 990111 | 990111    |
| 990112 | 990112    |
| 990113 | 990113    |
| 990114 | 990114    |
| 990115 | 990115    |
| 990116 | 990116    |
| 990117 | 990117    |
| 990118 | 990118    |
| 990119 | 990119    |
| 990120 | 990120    |
+--------+-----------+
20 rows in set (1.02 sec)

sky@localhost : example 10:46:58> SELECT user.id,user.nick_name
    ->     FROM (
    ->         SELECT user_id
    ->         FROM user_group
    ->         WHERE user_group.group_id = 1
    ->         ORDER BY gmt_create desc
    ->         limit 100,20) t,user
    ->     WHERE t.user_id = user.id;
+--------+-----------+
| id     | nick_name |
+--------+-----------+
| 990101 | 990101    |
| 990102 | 990102    |
| 990103 | 990103    |
| 990104 | 990104    |
| 990105 | 990105    |
| 990106 | 990106    |
```

```
| 990107 | 990107    |
| 990108 | 990108    |
| 990109 | 990109    |
| 990110 | 990110    |
| 990111 | 990111    |
| 990112 | 990112    |
| 990113 | 990113    |
| 990114 | 990114    |
| 990115 | 990115    |
| 990116 | 990116    |
| 990117 | 990117    |
| 990118 | 990118    |
| 990119 | 990119    |
| 990120 | 990120    |
+--------+-----------+
20 rows in set (0.96 sec)
```

查看系统中的 profile 信息，刚刚执行的两个 SQL 语句的执行 profile 信息已经记录下来了：

```
sky@localhost : example 10:47:07> show profiles\G
*************************** 1. row ***************************
Query_ID: 1
Duration: 1.02367600
   Query: SELECT id,nick_name
    FROM user,user_group
    WHERE user_group.group_id = 1
        and user_group.user_id = user.id
    ORDER BY user_group.gmt_create desc
    limit 100,20
*************************** 2. row ***************************
Query_ID: 2
Duration: 0.96327800
   Query: SELECT user.id,user.nick_name
    FROM (
        SELECT user_id
        FROM user_group
        WHERE user_group.group_id = 1
        ORDER BY gmt_create desc
        limit 100,20) t,user
    WHERE t.user_id = user.id
2 rows in set (0.00 sec)

sky@localhost : example 10:47:34> SHOW profile CPU,BLOCK IO io FOR query  1;
```

Status	Duration	CPU_user	CPU_system	Block_ops_in	Block_ops_out
(initialization)	0.000068	0	0	0	0
Opening tables	0.000015	0	0	0	0
System lock	0.000006	0	0	0	0
Table lock	0.000009	0	0	0	0
init	0.000026	0	0	0	0
optimizing	0.000014	0	0	0	0
statistics	0.000068	0	0	0	0
preparing	0.000019	0	0	0	0
executing	0.000004	0	0	0	0
Sorting result	1.03614	0.5600349	0.428027	0	15632
Sending data	0.071047	0	0.004	88	0
end	0.000012	0	0	0	0
query end	0.000006	0	0	0	0
freeing items	0.000012	0	0	0	0
closing tables	0.000007	0	0	0	0
logging slow query	0.000003	0	0	0	0

16 rows in set (0.00 sec)

sky@localhost : example 10:47:40> SHOW profile CPU,BLOCK IO io FOR query 2;

Status	Duration	CPU_user	CPU_system	Block_ops_in	Block_ops_out
(initialization)	0.000087	0	0	0	0
Opening tables	0.000018	0	0	0	0
System lock	0.000007	0	0	0	0
Table lock	0.000059	0	0	0	0
optimizing	0.00001	0	0	0	0
statistics	0.000068	0	0	0	0
preparing	0.000017	0	0	0	0
executing	0.000004	0	0	0	0
Sorting result	0.928184	0.572035	0.352022	0	32
Sending data	0.000112	0	0	0	0
init	0.000025	0	0	0	0
optimizing	0.000012	0	0	0	0
statistics	0.000025	0	0	0	0
preparing	0.000013	0	0	0	0
executing	0.000004	0	0	0	0
Sending data	0.000241	0	0	0	0
end	0.000005	0	0	0	0
query end	0.000006	0	0	0	0

freeing items	0.000015	0	0	0	0
closing tables	0.000004	0	0	0	0
removing tmp table	0.000019	0	0	0	0
closing tables	0.000005	0	0	0	0
logging slow query	0.000004	0	0	0	0

我们先看看两条 SQL 执行中的 IO 消耗，两者区别就在于“Sorting result”，我们回顾一下前面执行计划的对比，两个解决方案的排序过滤数据的时机不一样，排序后需要取得的数据量一个是 20000，一个是 20，正好和这里的 profile 信息吻合，第一种解决方案的“Sorting result”的 IO 值是第二种解决方案的将近 500 倍。

然后再来看看 CPU 消耗，所有消耗中，消耗最大的也是“Sorting result”这一项，第一个消耗多出的缘由和上面 IO 消耗差异是一样的。

结论：

通过上面两条功能完全相同的 SQL 语句的执行计划分析，以及通过实际执行后的 profile 数据的验证，都证明了第二种解决方案优于第一种解决方案。同时通过后者的实际验证，也再次证明了我们前面所做的执行计划基本决定了 SQL 语句性能。

6.4 Schema 设计对系统的性能影响

前面两节中，我们已经分析了在一个数据库应用系统的软环境中应用系统的架构实现和系统中与数据库交互的 SQL 语句对系统性能的影响。在这一节我们再分析一下系统的数据模型设计实现对系统的性能影响，更通俗一点就是数据库的 Schema 设计对系统性能的影响。

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，基本就算完事了。

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。

由于在后面的“MySQL 数据库应用系统设计”中的“系统架构最优化“这一节中会介较为详细的从性能优化的角度来分析如何如何设计数据库 Schema，所以这里暂时先不介绍如何来设计性能优异的数据库 Schema 结构，仅仅通过一个实际的示例来展示 Schema 结构的不一样在性能方面所带来的差异。

需求概述：一个简单的讨论区系统，需要有用户，用户组，组讨论区这三部分基本功能

简要分析：1、需要存放用户数据的表；

2、需要存放分组信息和存放用户与组关系的表

3、需要存放讨论信息的表；

解决方案：

原始方案一：分别用用四个表来存放用户，分组，用户与组关系以及各组的讨论帖子的信息如下：

user 用户表：

Field	Type	Null	Key	Default	Extra
id	int(11)	NO		0	
nick_name	varchar(32)	NO		NULL	
password	char(64)	YES		NULL	
email	varchar(32)	NO		NULL	
status	varchar(16)	NO		NULL	
sexuality	char(1)	NO		NULL	
msn	varchar(32)	YES		NULL	
sign	varchar(64)	YES		NULL	
birthday	date	YES		NULL	
hobby	varchar(64)	YES		NULL	
location	varchar(64)	YES		NULL	
description	varchar(1024)	YES		NULL	

groups 分组表：

Field	Type	Null	Key	Default	Extra
id	int(11)	NO		NULL	
gmt_create	datetime	NO		NULL	
gmt_modified	datetime	NO		NULL	
name	varchar(32)	NO		NULL	
status	varchar(16)	NO		NULL	
description	varchar(1024)	YES		NULL	

user_group 关系表：

Field	Type	Null	Key	Default	Extra
user_id	int(11)	NO	MUL	NULL	
group_id	int(11)	NO	MUL	NULL	
user_type	int(11)	NO		NULL	
gmt_create	datetime	NO		NULL	
gmt_modified	datetime	NO		NULL	
status	varchar(16)	NO		NULL	

+-------------+------------+------+-----+---------+-------+

group_message 讨论组帖子表：

Field	Type	Null	Key	Default	Extra
id	int(11)	NO		NULL	
gmt_create	datetime	NO		NULL	
gmt_modified	datetime	NO		NULL	
group_id	int(11)	NO		NULL	
user_id	int(11)	NO		NULL	
subject	varchar(128)	NO		NULL	
content	text	YES		NULL	

优化后方案二：

user 用户表：

Field	Type	Null	Key	Default	Extra
id	int(11)	NO		0	
nick_name	varchar(32)	NO		NULL	
password	char(64)	YES		NULL	
email	varchar(32)	NO		NULL	
status	varchar(16)	NO		NULL	

user_profile 用户属性表（记录与 user 一一对应）：

Field	Type	Null	Key	Default	Extra
sexuality	char(1)	NO		NULL	
msn	varchar(32)	YES		NULL	
sign	varchar(64)	YES		NULL	
birthday	date	YES		NULL	
hobby	varchar(64)	YES		NULL	
location	varchar(64)	YES		NULL	
description	varchar(1024)	YES		NULL	

groups 和 user_group 这两个表和方案一完全一样

group_message 讨论组帖子表：

+-------------+------------+------+-----+---------+-------+

Field	Type	Null	Key	Default	Extra
id	int(11)	NO		NULL	
gmt_create	datetime	NO		NULL	
gmt_modified	datetime	NO		NULL	
group_id	int(11)	NO		NULL	
user_id	int(11)	NO		NULL	
author	varchar(32)	NO		NULL	
subject	varchar(128)	NO		NULL	

group_message_content 帖子内容表（记录与 group_message 一一对应）：

Field	Type	Null	Key	Default	Extra
group_msg_id	int(11)	NO		NULL	
content	text	NO		NULL	

我们先来比较一下两个解决方案所设计的 Schema 的区别。区别主要体现在两点，一个区别是在 group_message 表中增加了 author 字段来存放发帖作者的昵称，与 user 表的 nick_name 相对应，另外一个就是第二个解决方案将 user 表和 group_message 表都分拆成了两个表，关系分别都是一一对应。

方案二看上去比方案一要更复杂一些，首先是表的数量多了 2 个，然后是在 group_message 中冗余存放了作者昵称。我们试想一下，一个讨论区系统，访问最多的页面会是什么？我想大家都会很清楚是帖子标题列表页面。而帖子标题列表页面最主要的信息就是都是来自 group_message 表中，同时帖子标题后面的作者一般都是通过用户名成（昵称）来展示。按照第一种解决方案来设计的 Schema，我们就需要执行类似如下这样的 SQL 语句来得到数据：

```
SELECT t.id, t.subject,user.id, u.nick_name
FROM (
    SELECT id, user_id, subject
    FROM group_message
    WHERE group_id = ?
    ORDER BY gmt_modified DESC LIMIT 20
) t, user u
WHERE t.user_id = u.id
```

但是第二中解决方案所需要执行的 SQL 就会简单很多，如下：

```
SELECT t.id, t.subject, t.user_id, t.author
FROM group_message
WHERE group_id = ?
ORDER BY gmt_modified DESC LIMIT 20
```

两个 SQL 相比较，大家都能很明显的看出谁优谁劣了，第一个是需要读取两个表的数据进行 Join，

与第二个 SQL 相比性能差距很大，尤其是如果第一个再写的差一点，性能更是非常糟糕，两者所带来的资源消耗就更相差玄虚了。

不仅仅如此，由于第一个方案中的 group_message 表中还包含一个大字段“content”，该字段所存放的信息要占整个表的绝大部分存储空间，但在这条系统中执行最频繁的 SQL 之一中是完全不需要该字段所存放信息的，但是由于这个 SQL 又没办法做到不访问 group_message 表的数据，所以第一条 SQL 在数据读取过程中会需要读取大量没有任何意义的数据。

在系统中用户数据的读取也是比较频繁的，但是大多数地方所需要的用户数据都只是用户的几个基本属性，如用户的 id，昵称，密码，状态，邮箱等，所以将用户表的这几个属性单独分离出来后，也会让大量的 SQL 语句在运行的时候减少数据的检索量，从而提高性能。

可能有人会觉得，在我们将一个表分成两个表的时候，我们如果要访问被分拆出去的信息的时候，性能不是就会变差了吗？是的，对于那些需要访问如 user 的 sign，msn 等原来只需要一个表就可以完成的 SQL 来说，现在都需要两条 SQL 来完成，性能确实会 有所降低，但是由于两个表都是一对一的关联关系，关联字段的过滤性也非常高，而且这样的查询需求在整个系统中所占有的比例也并不高，所以这里所带来的性能损失实际上要远远小于在其他 SQL 上所节省出来的资源，所以完全不必为此担心

6.5 硬件环境对系统性能的影响

在本章之前的所有部分都是介绍的整个系统中的软件环境对系统性能的影响，这一节我们将从系统硬件环境来分析对数据库系统的影响，并从数据库服务器主机的角度来做一些针对性的优化建议。

任何一个系统的硬件环境都会对起性能起到非常关键的作用，这一点我想每一位读者朋友都是非常清楚的。而数据库应用系统环境中，由于数据库自身的特点和在系统中的角色决定了他在整个系统中是最难以扩展的部分。所以在大多数环境下，数据库服务器主机（或者主机集群）的性能在很大程度上决定了整个应用系统的性能。

既然我们的数据库主机资源如此重要，肯定很多读者朋友会希望知道，数据库服务器主机的各部分硬件到底谁最重要，各部分对整体性能的影响各自所占的比例是多少，以便能够根据这些比例选取合适的主机机型作为数据库主机。但是我只能很遗憾的告诉大家，没有任何一个定律或者法则可以很准确的给出这个答案。

当然，大家也不必太沮丧。虽然没有哪个法则可以准确的知道我们到底该如何选配一个主机的各部分硬件，但是根据应用类型的不同，总体上还是有一个可以大致遵循的原则可以参考的。

首先，数据库主机是存取数据的地方，那么其 IO 操作自然不会少，所以数据库主机的 IO 性能肯定是需要最优先考虑的一个因素，这一点不管是什么类型的数据库应用都是适用的。不过，这里的 IO 性能并不仅仅只是指物理的磁盘 IO，而是主机的整体 IO 性能，是主机整个 IO 系统的总体 IO 性能。而 IO 性能本身又可以分为两类，一类是每秒可提供的 IO 访问次数，也就是我们常说的 IOPS 数量，还有一种就是每秒的 IO 总流量，也就是我们常说的 IO 吞吐量。在主机中决定 IO 性能部件主要由磁盘和内存所决定，当然也包括各种与 IO 相关的板卡。

其次，由于数据库主机和普通的应用程序服务器相比，资源要相对集中很多，单台主机上所需要进行的计算量自然也就比较多，所以数据库主机的CPU处理能力也不能忽视。

最后，由于数据库负责数据的存储，与各应用程序的交互中传递的数据量比其他各类服务器都要多，所以数据库主机的网络设备的性能也可能会成为系统的瓶颈。

由于上面这三类部件是影响数据库主机性能的最主要因素，其他部件成为性能瓶颈的几率要小很多，所以后面我们通过对各种类型的应用做一个简单的分析，再针对性的给出这三类部件的基本选型建议。

1、典型OLTP应用系统

对于各种数据库系统环境中大家最常见的OLTP系统，其特点是并发量大，整体数据量比较多，但每次访问的数据比较少，且访问的数据比较离散，活跃数据占总体数据的比例不是太大。对于这类系统的数据库实际上是最难维护，最难以优化的，对主机整体性能要求也是最高的。因为他不仅访问量很高，数据量也不小。

针对上面的这些特点和分析，我们可以对OLTP的得出一个大致的方向。

虽然系统总体数据量较大，但是系统活跃数据在数据总量中所占的比例不大，那么我们可以通过扩大内存容量来尽可能多的将活跃数据cache到内存中；

虽然IO访问非常频繁，但是每次访问的数据量较少且很离散，那么我们对磁盘存储的要求是IOPS表现要很好，吞吐量是次要因素；

并发量很高，CPU每秒所要处理的请求自然也就很多，所以CPU处理能力需要比较强劲；

虽然与客户端的每次交互的数据量并不是特别大，但是网络交互非常频繁，所以主机与客户端交互的网络设备对流量能力也要求不能太弱。

2、典型OLAP应用系统

用于数据分析的OLAP系统的主要特点就是数据量非常大，并发访问不多，但每次访问所需要检索的数据量都比较多，而且数据访问相对较为集中，没有太明显的活跃数据概念。

基于OLAP系统的各种特点和相应的分析，针对OLAP系统硬件优化的大致策略如下：

数据量非常大，所以磁盘存储系统的单位容量需要尽量大一些；

单次访问数据量较大，而且访问数据比较集中，那么对IO系统的性能要求是需要有尽可能大的每秒IO吞吐量，所以应该选用每秒吞吐量尽可能大的磁盘；

虽然IO性能要求也比较高，但是并发请求较少，所以CPU处理能力较难成为性能瓶颈，所以CPU处理能力没有太苛刻的要求；

虽然每次请求的访问量很大，但是执行过程中的数据大都不会返回给客户端，最终返回给客户端的数据量都较小，所以和客户端交互的网络设备要求并不是太高；

此外，由于OLAP系统由于其每次运算过程较长，可以很好的并行化，所以一般的OLAP系统都是由多台主机构成的一个集群，而集群中主机与主机之间的数据交互量一般来说都是非常大的，所以在集群中主机之间的网络设备要求很高。

3、除了以上两个典型应用之外，还有一类比较特殊的应用系统，他们的数据量不是特别大，但是访问请求及其频繁，而且大部分是读请求。可能每秒需要提供上万甚至几万次请求，每次请求都非常简

单，可能大部分都只有一条或者几条比较小的记录返回，就比如基于数据库的 DNS 服务就是这样类型的服务。

虽然数据量小，但是访问极其频繁，所以可以通过较大的内存来 cache 住大部分的数据，这能够保证非常高的命中率，磁盘 IO 量比较小，所以磁盘也不需要特别高性能的；

并发请求非常频繁，比需要较强的 CPU 处理能力才能处理；

虽然应用与数据库交互量非常大，但是每次交互数据较少，总体流量虽然也会较大，但是一般来说普通的千兆网卡已经足够了。

在很多人看来，性能的根本决定因素是硬件性能的好坏。但实际上，硬件性能只能在某些阶段对系统性能产生根本性影响。当我们的 CPU 处理能力足够的多，IO 系统的处理能力足够强的时候，如果我们的应用架构和业务实现不够优化，一个本来很简单的实现非得绕很多个弯子来回交互多次，那再强的硬件也没有用，因为来回的交互总是需要消耗时间。尤其是有些业务逻辑设计不是特别合理的应用，数据库 Schema 设计的不够合理，一个任务在系统中又被分拆成很多个步骤，每个步骤都使用了非常复杂的 Query 语句。笔者曾经就遇到过这样一个系统，该系统是购买的某知名厂商的一个项目管理软件。该系统最初运行在一台 Dell2950 的 PC Server 上面，使用者一直抱怨系统响应很慢，但我从服务器上面的状态来看系统并繁忙（系统并发不是太大）。后来使用者强烈要求通过更换硬件设施来提升系统性能，虽然我一直反对，但最后在管理层的要求下，更换成了一台 Sun 的 S880 小型机，主机 CPU 的处理能力至少是原来机器的 3 倍以上，存储系统也从原来使用本地磁盘换成使用 EMC 的中断存储 CX300。可在试用阶段，发现系统整体性能没有任何的提升，最终还是取消了更换硬件的计划。

所以，在应用系统的硬件配置方面，我们应该要以一个理性的眼光来看待，只有合适的才是最好的。并不是说硬件资源越好，系统性能就一定会越好。而且，硬件系统本身总是有一个扩展极限的，如果我们一味的希望通过升级硬件性能来解决系统的性能问题，那么总有一天将会遇到无法逾越的瓶颈。到那时候，就算有再多的钱去砸也无济于事了。

6.6 小结

虽然本章是以影响 MySQL Server 性能的相关因素来展开分析，但实际上很多内容都对于大多数数据库应用系统适用。数据库管理软件仅仅是实现了数据库应用系统中的数据存取操作，和数据的持久化。数据库应用系统的优化真正能带来最大收益的就是商业需求和系统架构及业务实现的优化，然后是数据库 Schema 设计的优化，然后才是 Query 语句的优化，最后才是数据库管理软件自身的一些优化。通过笔者的经验，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。

第 7 章 MySQL 数据库锁定机制

前言：

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。本章将对 MySQL 中两种使用最为频繁的存储引擎 MyISAM 和 Innodb 各自的锁定机制进行较为详细的分析。

7.1 MySQL 锁定机制简介

数据库锁定机制简单来说就是数据库为了保证数据的一致性而使各种共享资源在被并发访问访问变得有序所设计的一种规则。对于任何一种数据库来说都需要有相应的锁定机制，所以 MySQL 自然也不能例外。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。

总的来说，MySQL 各存储引擎使用了三种类型（级别）的锁定机制：行级锁定，页级锁定和表级锁定。下面我们先分析一下 MySQL 这三种锁定的特点和各自的优劣所在。

- 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。

- 表级锁定（table-level）

和行级锁定相反，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。

在数据库实现资源锁定的过程中，随着锁定资源颗粒度的减小，锁定相同数据量的数据所需要消耗的内存数量是越来越多的，实现算法也会越来越复杂。不过，随着锁定资源颗粒度的减小，应用程序的访问请求遇到锁等待的可能性也会随之降低，系统整体并发度也随之提升。

在 MySQL 数据库中，使用表级锁定的主要是 MyISAM，Memory，CSV 等一些非事务性存储引擎，而使用行级锁定的主要是 Innodb 存储引擎和 NDB Cluster 存储引擎，页级锁定主要是 BerkeleyDB 存储引擎的锁定方式。

MySQL 的如此的锁定机制主要是由于其最初的历史所决定的。在最初，MySQL 希望设计一种完全独立于各种存储引擎的锁定机制，而且在早期的 MySQL 数据库中，MySQL 的存储引擎（MyISAM 和 Momery）的设计是建立在“任何表在同一时刻都只允许单个线程对其访问（包括读）”这样的假设之上。但是，随着 MySQL 的不断完善，系统的不断改进，在 MySQL3.23 版本开发的时候，MySQL 开发人员不得不修正之前的假设。因为他们发现一个线程正在读某个表的时候，另一个线程是可以对该表进行 insert 操作的，只不过只能 INSERT 到数据文件的最尾部。这也就是从 MySQL 从 3.23 版本开始提供的我们所说的 Concurrent Insert。

当出现 Concurrent Insert 之后，MySQL 的开发人员不得不修改之前系统中的锁定实现功能，但是仅仅只是增加了对 Concurrent Insert 的支持，并没有改动整体架构。可是在不久之后，随着 BerkeleyDB 存储引擎的引入，之前的锁定机制遇到了更大的挑战。因为 BerkeleyDB 存储引擎并没有 MyISAM 和 Memory 存储引擎同一时刻只允许单一线程访问某一个表的限制，而是将这个单线程访问限制的颗粒度缩小到了单个 page，这又一次迫使 MySQL 开发人员不得不再一次修改锁定机制的实现。

由于新的存储引擎的引入，导致锁定机制不能满足要求，让 MySQL 的人意识到已经不可能实现一种完全独立的满足各种存储引擎要求的锁定实现机制。如果因为锁定机制的拙劣实现而导致存储引擎的整体性能的下降，肯定会严重打击存储引擎提供者的积极性，这是 MySQL 公司非常不愿意看到的，因为这完全不符合 MySQL 的战略发展思路。所以工程师们不得不放弃了最初的设计初衷，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。

7．2 各种锁定机制分析

在整体了解了MySQL锁定机制之后，这一节我们将详细分析MySQL自身提供的表锁定机制和其他储引擎实自身实现的行锁定机制，并通过MyISAM存储引擎和Innodb存储引擎实例演示。

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)

当前持有读锁的所有线程的相关信息都能够在Current read-lock queue中找到，队列中的信息按照获取到锁的时间依序存放。而正在等待锁定资源的信息则存放在Pending read-lock queue里面，另外两个存放写锁信息的队列也按照上面相同规则来存放信息。

虽然对于我们这些使用者来说MySQL展现出来的锁定（表锁定）只有读锁定和写锁定这两种类型，但是在MySQL内部实现中却有多达11种锁定类型，由系统中一个枚举量（thr_lock_type）定义，各值描述如下：

锁定类型	说明
IGNORE	当发生锁请求的时候内部交互使用，在锁定结构和队列中并不会有任何信息存储
UNLOCK	释放锁定请求的交互用所类型
READ	普通读锁定
WRITE	普通写锁定
READ_WITH_SHARED_LOCKS	在Innodb中使用到，由如下方式产生 如：SELECT ... LOCK IN SHARE MODE
READ_HIGH_PRIORITY	高优先级读锁定
READ_NO_INSERT	不允许Concurent Insert的锁定
WRITE_ALLOW_WRITE	这个类型实际上就是当由存储引擎自行处理锁定的时候，mysqld允许其他的线程再获取读或者写锁定，因为即使资源冲突，存储引擎自己也会知道怎么来处理
WRITE_ALLOW_READ	这种锁定发生在对表做DDL（ALTER TABLE ...）的时候，MySQL可以允许其他线程获取读锁定，因为MySQL是通过重建整个表然后再RENAME而实现的该功能，所在整个过程原表仍然可以提供读服务
WRITE_CONCURRENT_INSERT	正在进行Concurent Insert时候所使用的锁定方式，该锁定进行的时候，除了READ_NO_INSERT之外的其他任何读锁定请求都不会被阻塞
WRITE_DELAYED	在使用INSERT DELAYED时候的锁定类型
WRITE_LOW_PRIORITY	显示声明的低级别锁定方式，通过设置LOW_PRIORITY_UPDAT = 1而产生
WRITE_ONLY	当在操作过程中某个锁定异常中断之后系统内部需要进行CLOSE TABLE操作，在这个过程中出现的锁定类型就是WRITE_ONLY

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；

如果满足了上面两个条件之后，该请求会被立即通过，并将相关的信息存入 Current read-lock queue 中，而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。当然，也可能遇到以下这两种特殊情况：

1. 请求锁定的类型为 WRITE_DELAYED;
2. 请求锁定的类型为 WRITE_CONCURRENT_INSERT 或者是 TL_WRITE_ALLOW_WRITE，同时 Current read lock 是 READ_NO_INSERT 的锁定类型。

当遇到这两种特殊情况的时候，写锁定会立即获得而进入 Current write-lock queue 中

如果刚开始第一次检测就 Current write-lock queue 中已经存在了锁定相同资源的写锁定存在，那么就只能进入等待队列等待相应资源锁定的释放了。

读请求和写等待队列中的写锁请求的优先级规则主要为以下规则决定：

1. 除了 READ_HIGH_PRIORITY 的读锁定之外，Pending write-lock queue 中的 WRITE 写锁定能够阻塞所有其他的读锁定；
2. READ_HIGH_PRIORITY 读锁定的请求能够阻塞所有 Pending write-lock queue 中的写锁定；
3. 除了 WRITE 写锁定之外，Pending write-lock queue 中的其他任何写锁定都比读锁定的优先级低。

写锁定出现在 Current write-lock queue 之后，会阻塞除了以下情况下的所有其他锁定的请求：

1. 在某些存储引擎的允许下，可以允许一个 WRITE_CONCURRENT_INSERT 写锁定请求
2. 写锁定为 WRITE_ALLOW_WRITE 的时候，允许除了 WRITE_ONLY 之外的所有读和写锁定请求
3. 写锁定为 WRITE_ALLOW_READ 的时候，允许除了 READ_NO_INSERT 之外的所有读锁定请求
4. 写锁定为 WRITE_DELAYED 的时候，允许除了 READ_NO_INSERT 之外的所有读锁定请求
5. 写锁定为 WRITE_CONCURRENT_INSERT 的时候，允许除了 READ_NO_INSERT 之外的所有读锁定请求

随着 MySQL 存储引擎的不断发展，目前 MySQL 自身提供的锁定机制已经没有办法满足需求了，很多存储引擎都在 MySQL 所提供的锁定机制之上做了存储引擎自己的扩展和改造。

MyISAM 存储引擎基本上可以说是对 MySQL 所提供的锁定机制所实现的表级锁定依赖最大的一种存储引擎了，虽然 MyISAM 存储引擎自己并没有在自身增加其他的锁定机制，但是为了更好的支持相关特性，

MySQL 在原有锁定机制的基础上为了支持其 Concurrent Insert 的特性而进行了相应的实现改造。

而其他几种支持事务的存储存储引擎，如 Innodb，NDB Cluster 以及 Berkeley DB 存储引擎则是让 MySQL 将锁定的处理直接交给存储引擎自己来处理，在 MySQL 中仅持有 WRITE_ALLOW_WRITE 类型的锁定。

由于 MyISAM 存储引擎使用的锁定机制完全是由 MySQL 提供的表级锁定实现，所以下面我们将以 MyISAM 存储引擎作为示例存储引擎，来实例演示表级锁定的一些基本特性。由于，为了让示例更加直观，我将使用显示给表加锁来演示：

时刻	Session a	Session b
	READ	
1	sky@localhost : example 11:21:08> lock table test_table_lock read; Query OK, 0 rows affected (0.00 sec) 显示给 test_table_lock 加读锁定	
2	sky@localhost : example 11:21:10> select * from test_table_lock limit 1; +------+------+ | a | b | +------+------+ | 1 | 1 | +------+------+ 1 row in set (0.01 sec) 自己的读操作未被阻塞	sky@localhost : example 11:21:13> select * from test_table_lock limit 1; +------+------+ | a | b | +------+------+ | 1 | 1 | +------+------+ 1 row in set (0.01 sec) 其他线程的读也未被阻塞
3	sky@localhost : example 11:21:15> update test_table_lock set b = a limit 1; ERROR 1099 (HY000): Table 'test_table_lock' was locked with a READ lock and can't be updated	sky@localhost : example 11:21:20> update test_table_lock set b = a limit 1; 写一下试试看？被阻塞了
4	sky@localhost : example 11:21:09> unlock tables; Query OK, 0 rows affected (0.00 sec) 解除读锁	
5		sky@localhost : example 11:21:20> update test_table_lock set b = a limit 1; Query OK, 0 rows affected (1 min 15.52 sec) Rows matched: 1 Changed: 0 Warnings: 0 在 session a 释放锁定资源之后，session b 获得了资源，更新成功
	sky@localhost : example 11:48:19> l	sky@localhost : example 11:48:20> ins

	ock table test_table_lock read local; Query OK, 0 rows affected (0.00 sec) 获取读锁定的时候增加 local 选项	ert into test_table_lock values(1,'s','c'); Query OK, 1 row affected (0.00 sec) 其他 session 的 insert 未被阻塞
		sky@localhost : example 11:48:23> update test_table_lock set a = 1 limit 1; 其他 session 的更新操作被阻塞
	WRITE	
6	这次加写锁试试看： sky@localhost : example 11:27:01> lock table test_table_lock write; Query OK, 0 rows affected (0.00 sec)	
7	sky@localhost : example 11:27:10> select * from test_table_lock limit 1; +------+------+ | a | b | +------+------+ | 1 | 1 | +------+------+ 1 row in set (0.01 sec) 自己 session 可以继续读	sky@localhost : example 11:27:16> select * from test_table_lock limit 1; 其他 session 被阻塞
8	sky@localhost : example 11:27:02> unlock tables; Query OK, 0 rows affected (0.00 sec) 释放锁定资源	
9		sky@localhost : example 11:27:16> select * from test_table_lock limit 1; +------+------+ | a | b | +------+------+ | 1 | 1 | +------+------+ 1 row in set (36.36 sec) 其他 session 获取的资源
	WRITE_ALLOW_READ	
10	sky@localhost : example 11:42:24> alter table test_table_lock add(c varchar(16)); Query OK, 5242880 rows affected (7.06 sec) Records: 5242880 Duplicates: 0 Warnings: 0 通过执行 DDL（ALTER TABLE），获取 W	sky@localhost : example 11:42:25> select * from test_table_lock limit 1; +------+------+ | a | b | +------+------+ | 1 | 1 | +------+------+ 1 row in set (0.01 sec) 其他 session 的读未被阻塞

	RITE_ALLOW_READ 类型的写锁定	

行级锁定

行级锁定不是 MySQL 自己实现的锁定方式，而是由其他存储引擎自己所实现的，如广为大家所知的 Innodb 存储引擎，以及 MySQL 的分布式存储引擎 NDB Cluster 等都是实现了行级锁定。

Innodb 锁定模式及实现机制

考虑到行级锁定君由各个存储引擎自行实现，而且具体实现也各有差别，而 Innodb 是目前事务型存储引擎中使用最为广泛的存储引擎，所以这里我们就主要分析一下 Innodb 的锁定特性。

总的来说，Innodb 的锁定机制和 Oracle 数据库有不少相似之处。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。

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：

	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

虽然 Innodb 的锁定机制和 Oracle 有不少相近的地方，但是两者的实现确是截然不同的。总的来说就是 Oracle 锁定数据是通过需要锁定的某行记录所在的物理 block 上的事务槽上表级锁定信息，而 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。

除了间隙锁给 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：

- 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；
- 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；
- 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

Innodb 各事务隔离级别下锁定及死锁

Innodb 实现的在 ISO / ANSI SQL92 规范中所定义的 Read UnCommited，Read Commited，Repeatable Read 和 Serializable 这四种事务隔离级别。同时，为了保证数据在事务中的一致性，实现了多版本数据访问。

之前在第一节中我们已经介绍过，行级锁定肯定会带来死锁问题，Innodb 也不可能例外。至于死锁的产生过程我们就不在这里详细描述了，在后面的锁定示例中会通过一个实际的例子为大家爱展示死锁的产生过程。这里我们主要介绍一下，在 Innodb 中当系检测到死锁产生之后是如何来处理的。

在 Innodb 的事务管理和锁定机制中，有专门检测死锁的机制，会在系统中产生死锁之后的很短时间内就检测到该死锁的存在。当 Innodb 检测到系统中产生了死锁之后，Innodb 会通过相应的判断来选这产生死锁的两个事务中较小的事务来回滚，而让另外一个较大的事务成功完成。那 Innodb 是以什么来为标准判定事务的大小的呢？MySQL 官方手册中也提到了这个问题，实际上在 Innodb 发现死锁之后，会计算出两个事务各自插入、更新或者删除的数据量来判定两个事务的大小。也就是说哪个事务所改变的记录条数越多，在死锁中就越不会被回滚掉。但是有一点需要注意的就是，当产生死锁的场景中涉及到不止 Innodb 存储引擎的时候，Innodb 是没办法检测到该死锁的，这时候就只能通过锁定超时限制来解决该死锁了。另外，死锁的产生过程的示例将在本节最后的 Innodb 锁定示例中演示。

Innodb 锁定机制示例

```
mysql> create table test_innodb_lock (a int(11),b varchar(16)) engine=innodb;
Query OK, 0 rows affected (0.02 sec)

mysql> create index test_innodb_a_ind on test_innodb_lock(a);
Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> create index test_innodb_lock_b_ind on test_innodb_lock(b);
Query OK, 11 rows affected (0.01 sec)
Records: 11  Duplicates: 0  Warnings: 0
```

时刻	Session a	Session b
	行锁定基本演示	
1	mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec)	mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec)

	mysql> update test_innodb_lock set b = 'b1' where a = 1; Query OK, 1 row affected (0.00 sec) Rows matched: 1 Changed: 1 Warnings: 0 更新，但是不提交	
2		mysql> update test_innodb_lock set b = 'b1' where a = 1; 被阻塞，等待
3	mysql> commit; Query OK, 0 rows affected (0.05 sec) 提交	
4		mysql> update test_innodb_lock set b = 'b1' where a = 1; Query OK, 0 rows affected (36.14 sec) Rows matched: 1 Changed: 0 Warnings: 0 解除阻塞，更新正常进行
	无索引升级为表锁演示	
5	mysql> update test_innodb_lock set b = '2' where b = 2000; Query OK, 1 row affected (0.02 sec) Rows matched: 1 Changed: 1 Warnings: 0	
6		mysql> update test_innodb_lock set b = '3' where b = 3000; 被阻塞，等待
7	mysql> commit; Query OK, 0 rows affected (0.10 sec)	
8		mysql> update test_innodb_lock set b = '3' where b = 3000; Query OK, 1 row affected (1 min 3.41 sec) Rows matched: 1 Changed: 1 Warnings: 0 阻塞解除，完成更新
	间隙锁带来的插入问题演示	
9	mysql> select * from test_innodb_lock; +------+------+ | a | b |	

	+------+------+ | 1 | b2 | | 3 | 3 | | 4 | 4000 | | 5 | 5000 | | 6 | 6000 | | 7 | 7000 | | 8 | 8000 | | 9 | 9000 | | 1 | b1 | +------+------+ 9 rows in set (0.00 sec) mysql> update test_innodb_lock set b = a * 100 where a < 4 and a > 1; Query OK, 1 row affected (0.02 sec) Rows matched: 1 Changed: 1 Warnings: 0	
10		mysql> insert into test_innodb_lock values(2,'200'); 被阻塞，等待
11	mysql> commit; Query OK, 0 rows affected (0.02 sec)	
12		mysql> insert into test_innodb_lock values(2,'200'); Query OK, 1 row affected (38.68 sec) 阻塞解除，完成插入
	使用共同索引不同数据的阻塞示例	
13	mysql> update test_innodb_lock set b = 'bbbbb' where a = 1 and b = 'b2'; Query OK, 1 row affected (0.00 sec) Rows matched: 1 Changed: 1 Warnings: 0	
14		mysql> update test_innodb_lock set b = 'bbbbb' where a = 1 and b = 'b1'; 被阻塞
15	mysql> commit; Query OK, 0 rows affected (0.02 sec)	

16		mysql> update test_innodb_lock set b = 'bbbbb' where a = 1 and b = 'b1'; Query OK, 1 row affected (42.89 sec) Rows matched: 1 Changed: 1 Warnings: 0 session 提交事务，阻塞去除，更新完成
	死锁示例	
17	mysql> update t1 set id = 110 where id = 11; Query OK, 0 rows affected (0.00 sec) Rows matched: 0 Changed: 0 Warnings: 0	
18		mysql> update t2 set id = 210 where id = 21; Query OK, 1 row affected (0.00 sec) Rows matched: 1 Changed: 1 Warnings: 0
19	mysql> update t2 set id = 2100 where id = 21; 等待 session b 释放资源，被阻塞	
20		mysql> update t1 set id = 1100 where id = 11; Query OK, 0 rows affected (0.39 sec) Rows matched: 0 Changed: 0 Warnings: 0 等待 session a 释放资源，被阻塞
	两个 session 互相等等待对方的资源释放之后才能释放自己的资源，造成了死锁	

7.3 合理利用锁机制优化 MySQL

MyISAM 表锁优化建议

对于 MyISAM 存储引擎，虽然使用表级锁定在锁定实现的过程中比实现行级锁定或者页级锁所带来的附加成本都要小，锁定本身所消耗的资源也是最少。但是由于锁定的颗粒度比较到，所以造成锁定资源的争用情况也会比其他的锁定级别都要多，从而在较大程度上会降低并发处理能力。

所以，在优化 MyISAM 存储引擎锁定问题的时候，最关键的就是如何让其提高并发度。由于锁定级别是不可能改变的了，所以我们首先需要尽可能让锁定的时间变短，然后就是让可能并发进行的操作尽可能的并发。

1、 缩短锁定时间

缩短锁定时间，短短几个字，说起来确实听容易的，但实际做起来恐怕就并不那么简单了。如何让锁定时间尽可能的短呢？唯一的办法就是让我们的 Query 执行时间尽可能的短。

a) 尽两减少大的复杂 Query，将复杂 Query 分拆成几个小的 Query 分布进行；
b) 尽可能的建立足够高效的索引，让数据检索更迅速；
c) 尽量让 MyISAM 存储引擎的表只存放必要的信息，控制字段类型；
d) 利用合适的机会优化 MyISAM 表数据文件；

2、 分离能并行的操作

说到MyISAM的表锁，而且是读写互相阻塞的表锁，可能有些人会认为在MyISAM存储引擎的表上就只能是完全的串行化，没办法再并行了。大家不要忘记了，MyISAM 的存储引擎还有一个非常有用的特性，那就是 Concurrent Insert（并发插入）的特性。

MyISAM 存储引擎有一个控制是否打开 Concurrent Insert 功能的参数选项：concurrent_insert，可以设置为 0，1 或者 2。三个值的具体说明如下：

a) concurrent_insert=2，无论 MyISAM 存储引擎的表数据文件的中间部分是否存在因为删除数据而留下的空闲空间，都允许在数据文件尾部进行 Concurrent Insert;
b) concurrent_insert=1，当 MyISAM 存储引擎表数据文件中间不存在空闲空间的时候，可以从文件尾部进行 Concurrent Insert;
c) concurrent_insert=0，无论 MyISAM 存储引擎的表数据文件的中间部分是否存在因为删除数据而留下的空闲空间，都不允许 Concurrent Insert。

3、合理利用读写优先级

在本章各种锁定分析一节中我们了解到了 MySQL 的表级锁定对于读和写是有不同优先级设定的，默认情况下是写优先级要大于读优先级。所以，如果我们可以根据各自系统环境的差异决定读与写的优先级。如果我们的系统是一个以读为主，而且要优先保证查询性能的话，我们可以通过设置系统参数选项 low_priority_updates=1，将写的优先级设置为比读的优先级低，即可让告诉 MySQL 尽量先处理读请求。当然，如果我们的系统需要有限保证数据写入的性能的话，则可以不用设置 low_priority_updates 参数了。

这里我们完全可以利用这个特性，将 concurrent_insert 参数设置为 1，甚至如果数据被删除的可能性很小的时候，如果对暂时性的浪费少量空间并不是特别的在乎的话，将 concurrent_insert 参数设置为 2 都可以尝试。当然，数据文件中间留有空域空间，在浪费空间的时候，还会造成在查询的时候需要读取更多的数据，所以如果删除量不是很小的话，还是建议将 concurrent_insert 设置为 1 更为合适。

Innodb 行锁优化建议

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。

要想合理利用 Innodb 的行级锁定，做到扬长避短，我们必须做好以下工作：

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；

由于 Innodb 的行级锁定和事务性，所以肯定会产生死锁，下面是一些比较常用的减少死锁产生概率的的小建议，读者朋友可以根据各自的业务特点针对性的尝试：

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

对于两种锁定级别，MySQL 内部有两组专门的状态变量记录系统内部锁资源争用情况，我们先看看 MySQL 实现的表级锁定的争用状态变量：

```
mysql> show status like 'table%';
+----------------------+-------+
| Variable_name        | Value |
+----------------------+-------+
| Table_locks_immediate | 100   |
| Table_locks_waited    | 0     |
+----------------------+-------+
```

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：

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

两个状态值都是从系统启动后开始记录，没出现一次对应的事件则数量加 1。如果这里的 Table_locks_waited 状态值比较高，那么说明系统中表级锁定争用现象比较严重，就需要进一步分析为什么会有较多的锁定资源争用了。

对于 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，如下：

```
mysql> show status like 'innodb_row_lock%';
+-------------------------------+--------+
| Variable_name                 | Value  |
+-------------------------------+--------+
| Innodb_row_lock_current_waits | 0      |
| Innodb_row_lock_time          | 490578 |
| Innodb_row_lock_time_avg      | 37736  |
| Innodb_row_lock_time_max      | 121411 |
| Innodb_row_lock_waits         | 13     |
+-------------------------------+--------+
```

Innodb 的行级锁定状态变量不仅记录了锁定等待次数，还记录了锁定总时长，每次平均时长，以及最大时长，此外还有一个非累积状态量显示了当前正在等待锁定的等待数量。对各个状态量的说明如下：

- 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；

对于这 5 个状态变量，比较重要的主要是 Innodb_row_lock_time_avg（等待平均时长），Innodb_row_lock_waits（等待总次数）以及 Innodb_row_lock_time（等待总时长）这三项。尤其是当等待次数很高，而且每次等待时长也不小的时候，我们就需要分析系统中为什么会有如此多的等待，然后根据分析结果着手指定优化计划。

此外，Innodb 出了提供这五个系统状态变量之外，还提供的其他更为丰富的即时状态信息供我们分析使用。可以通过如下方法查看：

1. 通过创建 Innodb Monitor 表来打开 Innodb 的 monitor 功能：

```
mysql> create table innodb_monitor(a int) engine=innodb;
Query OK, 0 rows affected (0.07 sec)
```

2. 然后通过使用“SHOW INNODB STATUS”查看细节信息（由于输出内容太多就不在此记录了）；

可能会有读者朋友问为什么要先创建一个叫 innodb_monitor 的表呢？因为创建该表实际上就是告诉 Innodb 我们开始要监控他的细节状态了，然后 Innodb 就会将比较详细的事务以及锁定信息记录进入 MySQL 的 error log 中，以便我们后面做进一步分析使用。

7．4 小结

本章以 MySQL Server 中的锁定简介开始，分析了当前 MySQL 中使用最为广泛的锁定方式表级锁定和行级锁定的基本实现机制，并通过 MyISAM 和 Innodb 这两大典型的存储引擎作为示例存储引擎所使用的表级锁定和行级锁定做了较为详细的分析和演示。然后，再通过分析两种锁定方式的特性，给出相应的优化建议和策略。最后了解了一下在 MySQL Server 中如何获得系统当前各种锁定的资源争用状况。希望本章内容能够对各位读者朋友在理解 MySQL 锁定机制方面有一定的帮助。

第 8 章 MySQL 数据库 Query 的优化

前言：

在之前“影响 MySQL 应用系统性能的相关因素”一章中我们就已经分析过了 Query 语句对数据库性能的影响非常大，所以本章将专门针对 MySQL 的 Query 语句的优化进行相应的分析。

8.1 理解 MySQL 的 Query Optimizer

8.1.1 MySQL Query Optimizer 是什么？

在“MySQL 架构组成”一章中的 “MySQL 逻辑组成”一节中我们已经了解到，在 MySQL 中有一个专门负责优化 SELECT 语句的优化器模块，这就是我们本节将要重点分析的 MySQL Optimizer，其主要的功能就是通过计算分析系统中收集的各种统计信息，为客户端请求的 Query 给出他认为最优的执行计划，也就是他认为最优的数据检索方式。

当 MySQL Optimizer 接收到从 Query Parser （解析器）送过来的 Query 之后，会根据 MySQL Query 语句的相应语法对该 Query 进行分解分析的同时，还会做很多其他的计算转化工作。如常量转化，无效内容删除，常量计算等等。所有这些工作都只为了 Optimizer 工作的唯一目的，分析出最优的数据检索方式，也就是我们常说的执行计划。

8.1.2 MySQL Query Optimizer 基本工作原理

在分析 MySQL Optimizer 的工作原理之前，先了解一下 MySQL 的 Query Tree。MySQL 的 Query Tree 是通过优化实现 DBXP 的经典数据结构和 Tree 构造器而生成的一个指导完成一个 Query 语句的请求所需要处理的工作步骤，我们可以简单的认为就是一个的数据处理流程规划，只不过是以一个 Tree 的数据结构存放而已。通过 Query Tree 我们可以很清楚的知道一个 Query 的完成需要经过哪些步骤的处理，每一步的数据来源在哪里，处理方式是怎样的。在整个 DBXP 的 Query Tree 生成过程中，MySQL 使用了 LEX 和 YACC 这两个功能非常强大的语法（词法）分析工具。MySQL Query Optimizer 的所有工作都是基于这个 Query Tree 所进行的。各位读者朋友如果对 MySQL Query Tree 实现生成的详细信息比较感兴趣，可以参考 Chales A. Bell 的《Expert MySQL》这本书，里面有比较详细的介绍。

MySQL Query Optimizer 并不是一个纯粹的 CBO（Cost Base Optimizer），而是在 CBO 的基础上增加了一个被称为 Heuristic Optimize（启发式优化）的功能。也就是说，MySQL Query Optimizer 在优化一个 Query 选择出他认为的最优执行计划的时候，并不一定完全按照系数据库的元信息和系统统计信息，而是在此基础上增加了某些特定的规则。其实我个人的理解就是在 CBO 的实现中增加了部分 RBO（Rule Base Optimizer）的功能，以确保在某些特别的场景下控制 Query 按照预定的方式生成执行计划。

当客户端向 MySQL 请求一条 Query ，到命令解析器模块完成请求分类区别出是 SELECT 并转发给 Query Optimizer 之后，Query Optimizer 首先会对整条 Query 进行，优化处理掉一些常量表达式的预算，直接换算成常量值。并对 Query 中的查询条件进行简化和转换，如去掉一些无用或者显而易见的条

件，结构调整等等。然后则是分析 Query 中的 Hint 信息（如果有），看显示 Hint 信息是否可以完全确定该 Query 的执行计划。如果没有 Hint 或者 Hint 信息还不足以完全确定执行计划，则会读取所涉及对象的统计信息，根据 Query 进行写相应的计算分析，然后再得出最后的执行计划。

Query Optimizer 是一个数据库软件非常核心的功能，虽然在这里说起来只是简单的几句话，但是在 MySQL 内部，Query Optimizer 实际上是经过了很多复杂的运算分析，才得出最后的执行计划。对于 MySQL Query Optimizer 更多的信息，各位读者可以通过 MySQL Internal 文档进行更为全面的了解。

8.2 Query 语句优化基本思路和原则

在分析如何优化 MySQL Query 之前，我们需要先了解一下 Query 语句优化的基本思路和原则。一般来说，Query 语句的优化思路和原则主要提现在以下几个方面：

1. 优化更需要优化的 Query；
2. 定位优化对象的性能瓶颈；
3. 明确的优化目标；
4. 从 Explain 入手；
5. 多使用 profile
6. 永远用小结果集驱动大的结果集；
7. 尽可能在索引中完成排序；
8. 只取出自己需要的 Columns；
9. 仅仅使用最有效的过滤条件；
10. 尽可能避免复杂的 Join 和子查询；

上面所列的几点信息，前面 4 点可以理解为 Query 优化的一个基本思路，后面部分则是我们优化中的基本原则。

下面我们先针对 Query 优化的基本思路做一些简单的分析，理解为什么我们的 Query 优化到底该如何进行。

优化更需要优化的 Query

为什么我们需要优化更需要优化的 Query？这个地球人都知道的“并不能成为问题的问题”我想就并不需要我过多解释吧，哈哈。

那什么样的 Query 是更需要优化呢？对于这个问题我们需要从对整个系统的影响来考虑。什么 Query 的优化能给系统整体带来更大的收益，就更需要优化。一般来说，高并发低消耗（相对）的 Query 对整个系统的影响远比低并发高消耗的 Query 大。我们可以通过以下一个非常简单的案例分析来充分说明问题。

假设有一个 Query 每小时执行 10000 次，每次需要 20 个 IO。另外一个 Query 每小时执行 10 次，每次需要 20000 个 IO。

我们先通过 IO 消耗方面来分析。可以看出，两个 Query 每小时所消耗的 IO 总数目是一样的，都是 200000 IO/小时。假设我们优化第一个 Query，从 20 个 IO 降低到 18 个 IO，也就是仅仅降低了 2 个 IO，则我们节省了 2 * 10000 = 20000 （IO/小时）。而如果希望通过优化第二个 Query 达到相同的效果，我们必须要让每个 Query 减少 20000 / 10 = 2000 IO。我想大家都会相信让第一个 Query 节省 2 个 IO 远比第二个 Query 节省 2000 个 IO 来的容易。

其次，如果通过 CPU 方面消耗的比较，原理和上面的完全一样。只要让第一个 Query 稍微节省一小块资源，就可以让整个系统节省出一大块资源，尤其是在排序，分组这些对 CPU 消耗比较多的操作中尤其突出。

最后，我们从对整个系统的影响来分析。一个频繁执行的高并发 Query 的危险性比一个低并发的 Query 要大很多。当一个低并发的 Query 走错执行计划，所带来的影响主要只是该 Query 的请求者的体验会变差，对整体系统的影响并不会特别的突出，之少还属于可控范围。但是，如果我们一个高并发的 Query 走错了执行计划，那所带来的后可很可能就是灾难性的，很多时候可能连自救的机会都不给你就会让整个系统 Crash 掉。曾经我就遇到这样一个案例，系统中一个并发度较高的 Query 语句走错执行计划，系统顷刻间 Crash，甚至我都还没有反应过来是怎么回事。当重新启动数据库提供服务后，系统负载立刻直线飙升，甚至都来不及登录数据库查看当时有哪些 Active 的线程在执行哪些 Query。如果是遇到一个并发并不太高的 Query 走错执行计划，至少我们还可以控制整个系统不至于系统被直接压跨，甚至连问题根源都难以抓到。

定位优化对象的性能瓶颈

当我们拿到一条需要优化的 Query 之后，第一件事情是什么？是反问自己，这条 Query 有什么问题？我为什么要优化他？只有明白了这些问题，我们才知道我们需要做什么，才能够找到问题的关键。而不能就只是觉得某个 Query 好像有点慢，需要优化一下，然后就开始一个一个优化方法去轮番尝试。这样很可能整个优化过程会消耗大量的人力和时间成本，甚至可能到最后还是得不到一个好的优化结果。这就像看病一样，医生必须要清楚的知道我们病的根源才能对症下药。如果只是知道我们什么地方不舒服，然后就开始通过各种药物尝试治疗，那这样所带来的后果可能就非常严重了。

所以，在拿到一条需要优化的 Query 之后，我们首先要判断出这个 Query 的瓶颈到底是 IO 还是 CPU。到底是因为在数据访问消耗了太多的时间，还是在数据的运算（如分组排序等）方面花费了太多资源？

一般来说，在 MySQL 5.0 系列版本中，我们可以通过系统自带的 PROFILING 功能很清楚的找出一个 Query 的瓶颈所在。当然，如果读者朋友为了使用 MySQL 的某些在 5.1 版本中才有的新特性（如 Partition，EVENT 等）亦或者是比较喜欢尝试新事务而早早使用的 MySQL 5.1 的预发布版本，可能就没办法使用这个功能了，因为该功能在 MySQL5.1 系列刚开始的版本中并不支持，不过让人非常兴奋的是该功能在最新出来的 MySQL 5.1 正式版（5.1.30）又已经提供了。而如果读者朋友正在使用的 MySQL 是 4.x 版本，那可能就只能通过自行分析 Query 的各个执行步骤，找到性能损失最大的地方。

明确的优化目标

当我们定为到了一条 Query 的性能瓶颈之后，就需要通过分析该 Query 所完成的功能和 Query 对系统的整体影响制订出一个明确的优化目标。没有一个明确的目标，优化过程将是一个漫无目的而且低

效的过程，也很难达收到一个理想的效果。尤其是对于一些实现应用中较为重要功能点的 Query 更是如此。

如何设定优化目标？这可能是很多人都非常头疼的问题，对于我自己也一样。要设定一个合理的优化目标，不能过于理想也不能放任自由，确实是一件非常头疼的事情。一般来说，我们首先需要清楚的了解数据库目前的整体状态，同时也要清楚的知道数据库中与该 Query 相关的数据库对象的各种信息，而且还要了解该 Query 在整个应用系统中所实现的功能。了解了数据库整体状态，我们就能知道数据库所能承受的最大压力，也就清楚了我们能够接受的最悲观情况。把握了该 Query 相关数据库对象的信息，我们就应该知道实现该 Query 的消耗最理想情况下需要消耗多少资源，最糟糕又需要消耗多少资源。最后，通过该 Query 所实现的功能点在整个应用系统中的重要地位，我们可以大概的分析出该 Query 可以占用的系统资源比例，而且我们也能够知道该 Query 的效率给客户带来的体验影响到底有多大。

当我们清楚了这些信息之后，我们基本可以得出该 Query 应该满足的一个性能范围是怎样的，这也就是我们的优化目标范围，然后就是通过寻找相应的优化手段来解决问题了。如果该 Query 实现的应用系统功能比较重要，我们就必须让目标更偏向于理想值一些，即使在其他某些方面作出一些让步与牺牲，比如调整 schema 设计，调整索引组成等，可能都是需要的。而如果该 Query 所实现的是一些并不是太关键的功能，那我们可以让目标更偏向悲观值一些，而尽量保证其他更重要的 Query 的性能。这种时候，即使需要调整商业需求，减少功能实现，也不得不应该作出让步。

从 Explain 入手

现在，优化目标也已经明确了，自然是奥开始动手的时候了。我们的优化到底该从何处入手呢？答案只有一个，从 Explain 开始入手。为什么？因为只有 Explain 才能告诉你，这个 Query 在数据库中是以一个什么样的执行计划来实现的。

但是，有一点我们必须清楚，Explain 只是用来获取一个 Query 在当前状态的数据库中的执行计划，在优化动手之前，我们比需要根据优化目标在自己头脑中有一个清晰的目标执行计划。只有这样，优化的目标才有意义。一个优秀的 SQL 调优人员（或者成为 SQL Performance Tuner），在优化任何一个 SQL 语句之前，都应该在自己头脑中已经先有一个预定的执行计划，然后通过不断的调整尝试，再借助 Explain 来验证调整的结果是否满足自己预定的执行计划。对于不符合预期的执行计划需要不断分析 Query 的写法和数据库对象的信息，继续调整尝试，直至得到预期的结果。

当然，人无完人，并不一定每次自己预设的执行计划都肯定是最优的，在不断调整测试的过程中，如果发现 MySQL Optimizer 所选择的执行计划的实际执行效果确实比自己预设的要好，我们当然还是应该选择使用 MySQL optimizer 所生成的执行计划。

上面的这个优化思路，只是给大家指了一个优化的基本方向，实际操作还需要读者朋友不断的结合具体应用场景不断的测试实践来体会。当然也并不一定所有的情况都非要严格遵循这样一个思路，规则是死的，人是活的，只有更合理的方法，没有最合理的规则。

在了解了上面这些优化的基本思路之后，我们再来看看优化的几个基本原则。

永远用小结果集驱动大的结果集

很多人喜欢在优化 SQL 的时候说用小表驱动大表，个人认为这样的说法不太严谨。为什么？因为大表经过 WHERE 条件过滤之后所返回的结果集并不一定就比小表所返回的结果集大，可能反而更小。在这种情况下如果仍然采用小表驱动大表，就会得到相反的性能效果。

其实这样的结果也非常容易理解，在 MySQL 中的 Join，只有 Nested Loop 一种 Join 方式，也就是 MySQL 的 Join 都是通过嵌套循环来实现的。驱动结果集越大，所需要循环的此时就越多，那么被驱动表的访问次数自然也就越多，而每次访问被驱动表，即使需要的逻辑 IO 很少，循环次数多了，总量自然也不可能很小，而且每次循环都不能避免的需要消耗 CPU ，所以 CPU 运算量也会跟着增加。所以，如果我们仅仅以表的大小来作为驱动表的判断依据，假若小表过滤后所剩下的结果集比大表多很多，结果就是需要的嵌套循环中带来更多的循环次数，反之，所需要的循环次数就会更少，总体 IO 量和 CPU 运算量也会少。而且，就算是非 Nested Loop 的 Join 算法，如 Oracle 中的 Hash Join，同样是小结果集驱动大的结果集是最优的选择。

所以，在优化 Join Query 的时候，最基本的原则就是“小结果集驱动大结果集”，通过这个原则来减少嵌套循环中的循环次数，达到减少 IO 总量以及 CPU 运算的次数。
尽可能在索引中完成排序

只取出自己需要的 Columns

任何时候在 Query 中都只取出自己需要的 Columns，尤其是在需要排序的 Query 中。为什么？

对于任何 Query，返回的数据都是需要通过网络数据包传回给客户端，如果取出的 Column 越多，需要传输的数据量自然会越大，不论是从网络带宽方面考虑还是从网络传输的缓冲区来看，都是一个浪费。

如果是需要排序的 Query 来说，影响就更大了。在 MySQL 中存在两种排序算法，一种是在 MySQL4.1 之前的老算法，实现方式是先将需要排序的字段和可以直接定位到相关行数据的指针信息取出，然后在我们所设定的排序区（通过参数 sort_buffer_size 设定）中进行排序，完成排序之后再次通过行指针信息取出所需要的 Columns，也就是说这种算法需要访问两次数据。第二种排序算法是从 MySQL4.1 版本开始使用的改进算法，一次性将所需要的 Columns 全部取出，在排序区中进行排序后直接将数据返回给请求客户端。改行算法只需要访问一次数据，减少了大量的随机 IO，极大的提高了带有排序的 Query 语句的效率。但是，这种改进后的排序算法需要一次性取出并缓存的数据比第一种算法要多很多，如果我们将并不需要的 Columns 也取出来，就会极大的浪费排序过程所需要的内存。在 MySQL4.1 之后的版本中，我们可以通过设置 max_length_for_sort_data 参数大小来控制 MySQL 选择第一种排序算法还是第二种排序算法。当所取出的 Columns 的单条记录总大小 max_length_for_sort_data 设置的大小的时候，MySQL 就会选择使用第一种排序算法，反之，则会选择第二种优化后的算法。为了尽可能提高排序性能，我们自然是更希望使用第二种排序算法，所以在 Query 中仅仅取出我们所需要的 Columns 是非常有必要的。

仅仅使用最有效的过滤条件

很多人在优化 Query 语句的时候很容易进入一个误区，那就是觉得 WHERE 子句中的过滤条件越多越好，实际上这并不是一个非常正确的选择。其实我们分析 Query 语句的性能优劣最关键的就是要让他

选择一条最佳的数据访问路径，如何做到通过访问最少的数据量完成自己的任务。

为什么说过滤条件多不一定是好事呢？请看下面示例：

需求： 查找某个用户在所有 group 中所发的讨论 message 基本信息。

场景： 1、知道用户 ID 和用户 nick_name

2、信息所在表为 group_message

3、group_message 中存在用户 ID(user_id)和 nick_name(author)两个索引

方案一：将用户 ID 和用户 nick_name 两者都作为过滤条件放在 WHERE 子句中来查询，Query 的执行计划如下：

```
sky@localhost : example 11:29:37> EXPLAIN SELECT * FROM group_message
    ->  WHERE user_id = 1 AND author='1111111111'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: ref
possible_keys: group_message_author_ind,group_message_uid_ind
          key: group_message_author_ind
      key_len: 98
          ref: const
         rows: 1
        Extra: Using where
1 row in set (0.00 sec)
```

方案二：仅仅将用户 ID 作为过滤条件放在 WHERE 子句中来查询，Query 的执行计划如下：

```
sky@localhost : example 11:30:45> EXPLAIN SELECT * FROM group_message
    -> WHERE user_id = 1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: ref
possible_keys: group_message_uid_ind
          key: group_message_uid_ind
      key_len: 4
          ref: const
         rows: 1
        Extra:
1 row in set (0.00 sec)
```

方案二：仅将用户 nick_name 作为过滤条件放在 WHERE 子句中来查询，Query 的执行计划如下：

```
sky@localhost : example 11:38:45> EXPLAIN SELECT * FROM group_message
    -> WHERE author = '1111111111'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: ref
possible_keys: group_message_author_ind
          key: group_message_author_ind
      key_len: 98
          ref: const
         rows: 1
        Extra: Using where
1 row in set (0.00 sec)
```

初略一看三个执行计划好像都挺好的啊，每一个 Query 的执行类型都利用到了索引，而且都是“ref”类型。可是仔细一分析，就会发现，group_message_uid_ind 索引的索引键长度为 4（key_len: 4），由于 user_id 字段类型为 int，所以我们可以判定出 Query Optimizer 给出的这个索引键长度是完全准确的。而 group_message_author_ind 索引的索引键长度为 98（key_len: 98），因为 author 字段定义为 varchar(32) ，而所使用的字符集是 utf8，32 * 3 + 2 = 98。而且，由于 user_id 与 author（来源于 nick_name）全部都是一一对应的，所以同一个 user_id 有哪些记录，那么所对应的 author 也会有完全相同的记录。所以，同样的数据在 group_message_author_ind 索引中所占用的存储空间要远远大于 group_message_uid_ind 索引所占用的空间。占用空间更大，代表我们访问该索引所需要读取的数据量就会更多。所以，选择 group_message_uid_ind 的执行计划才是最有的执行计划。也就是说，上面的方案二才是最有方案，而使用了更多的 WHERE 条件的方案一反而没有仅仅使用 user_id 一个过滤条件的方案一优。

可能有些人会说，那如果将 user_id 和 author 两者建立联合索引呢？告诉你，效果可能比没有这个索引的时候更差，因为这个联合索引的索引键更长，索引占用的空间将会更大。

这个示例并不一定能代表所有场景，仅仅是希望让大家明白，并不是任何时候都是使用的过滤条件越多性能会越好。在实际应用场景中，肯定会存在更多更复杂的情形，怎样使我们的 Query 有一个更优化的执行计划，更高效的性能，还需要靠大家仔细分析各种执行计划的具体差别，才能选择出更优化的 Query。

尽可能避免复杂的 Join 和子查询

我们都知道，MySQL 在并发这一块做的并不是太好，当并发量太高的时候，系统整体性能可能会急剧下降，尤其是遇到一些较为复杂的 Query 的时候更是如此。这主要与 MySQL 内部资源的争用锁定控制有关，如读写相斥等等。对于 Innodb 存储引擎由于实现了行级锁定可能还要稍微好一些，如果使用的 MyISAM 存储引擎，并发一旦较高的时候，性能下降非常明显。所以，我们的 Query 语句所涉及到的表越多，所需要锁定的资源就越多。也就是说，越复杂的 Join 语句，所需要锁定的资源也就越多，所阻塞的其他线程也就越多。相反，如果我们将比较复杂的 Query 语句分拆成多个较为简单的 Query 语

句分步执行，每次锁定的资源也就会少很多，所阻塞的其他线程也要少一些。

可能很多读者会有疑问，将复杂 Join 语句分拆成多个简单的 Query 语句之后，那不是我们的网络交互就会更多了吗？网络延时方面的总体消耗也就更大了啊，完成整个查询的时间不是反而更长了吗？是的，这种情况是可能存在，但也并不是肯定就会如此。我们可以再分析一下，一个复杂的 Join Query 语句在执行的时候，所需要锁定的资源比较多，可能被别人阻塞的概率也就更大，如果是一个简单的 Query，由于需要锁定的资源较少，被阻塞的概率也会小很多。所以 较为复杂的 Join Query 也有可能在执行之前被阻塞而浪费更多的时间。而且，我们的数据库所服务的并不是单单这一个 Query 请求，还有很多很多其他的请求，在高并发的系统中，牺牲单个 Query 的短暂响应时间而提高整体处理能力也是非常值得的。优化本身就是一门平衡与取舍的艺术，只有懂得取舍，平衡整体，才能让系统更优。

对于子查询，可能不需要我多说很多人就明白为什么会不被推荐使用。在 MySQL 中，子查询的实现目前还比较差，很难得到一个很好的执行计划，很多时候明明有索引可以利用，可 Query Optimizer 就是不用。从 MySQL 官方给出的信息说，这一问题将在 MySQL6.0 中得到较好的解决，将会引入 SemiJoin 的执行计划，可 MySQL6.0 离我们投入生产环境使用恐怕还有很遥远的一段时间。所以，在 Query 优化的过程中，能不用子查询的时候就尽量不要使用子查询。

上面这些仅仅只是一些常用的优化原则，并不是说在 Query 优化中就只需要做到这些原则就可以，更不是说 Query 优化只能通过这些原则来优化。在实际优化过程中，我们还可能会遇到很多带有较为复杂商业逻辑的场景，具体的优化方法就只能根据不同的应用场景来具体分析，逐步调整。其实，最有效的优化，就是不要用，也就是不要实现这个商业需求。

8.3 充分利用 Explain 和 Profiling

8.3.1 Explain 的使用

说到 Explain，肯定很多读者之前都都已经用过了，MySQL Query Optimizer 通过我让们执行 EXPLAIN 命令来告诉我们他将使用一个什么样的执行计划来优化我们的 Query。所以，可以说 Explain 是在优化 Query 时最直接有效的验证我们想法的工具。在本章前面部分我就说过，一个好的 SQL Performance Tuner 在动手优化一个 Query 之前，头脑中就应该已经有一个好的执行计划，后面的优化工作只是为实现该执行计划而作出各种调整。

在我们对某个 Query 优化过程中，需要不断的使用 Explain 来验证我们的各种调整是否有效。就像本书之前的很多示例都会通过 Explain 来验证和展示结果一样，所有的 Query 优化都应该充分利用他。

我们先看一下在 MySQL Explain 功能中给我们展示的各种信息的解释：

- ID：Query Optimizer 所选定的执行计划中查询的序列号；
- Select_type：所使用的查询类型，主要有以下这几种查询类型
  - ◇ DEPENDENT SUBQUERY：子查询中内层的第一个 SELECT，依赖于外部查询的结果集；
  - ◇ DEPENDENT UNION：子查询中的 UNION，且为 UNION 中从第二个 SELECT 开始的后面所有

SELECT，同样依赖于外部查询的结果集；
- ◇ PRIMARY：子查询中的最外层查询，注意并不是主键查询；
- ◇ SIMPLE：除子查询或者 UNION 之外的其他查询；
- ◇ SUBQUERY：子查询内层查询的第一个 SELECT，结果不依赖于外部查询结果集；
- ◇ UNCACHEABLE SUBQUERY：结果集无法缓存的子查询；
- ◇ UNION：UNION 语句中第二个 SELECT 开始的后面所有 SELECT，第一个 SELECT 为 PRIMARY
- ◇ UNION RESULT：UNION 中的合并结果；

- ◆ Table：显示这一步所访问的数据库中的表的名称；
- ◆ Type：告诉我们对表所使用的访问方式，主要包含如下集中类型；
  - ◇ all：全表扫描
  - ◇ const：读常量，且最多只会有一条记录匹配，由于是常量，所以实际上只需要读一次；
  - ◇ eq_ref：最多只会有一条匹配结果，一般是通过主键或者唯一键索引来访问；
  - ◇ fulltext：
  - ◇ index：全索引扫描；
  - ◇ index_merge：查询中同时使用两个（或更多）索引，然后对索引结果进行 merge 之后再读取表数据；
  - ◇ index_subquery：子查询中的返回结果字段组合是一个索引（或索引组合），但不是一个主键或者唯一索引；
  - ◇ rang：索引范围扫描；
  - ◇ ref：Join 语句中被驱动表索引引用查询；
  - ◇ ref_or_null：与 ref 的唯一区别就是在使用索引引用查询之外再增加一个空值的查询；
  - ◇ system：系统表，表中只有一行数据；
  - ◇ unique_subquery：子查询中的返回结果字段组合是主键或者唯一约束；
  - ◇
- ◆ Possible_keys：该查询可以利用的索引. 如果没有任何索引可以使用，就会显示成 null，这一项内容对于优化时候索引的调整非常重要；
- ◆ Key：MySQL Query Optimizer 从 possible_keys 中所选择使用的索引；
- ◆ Key_len：被选中使用索引的索引键长度；
- ◆ Ref：列出是通过常量（const），还是某个表的某个字段（如果是 join）来过滤（通过 key）的；
- ◆ Rows：MySQL Query Optimizer 通过系统收集到的统计信息估算出来的结果集记录条数；
- ◆ Extra：查询中每一步实现的额外细节信息，主要可能会是以下内容：
  - ◇ Distinct：查找 distinct 值，所以当 mysql 找到了第一条匹配的结果后，将停止该值的查询而转为后面其他值的查询；
  - ◇ Full scan on NULL key：子查询中的一种优化方式，主要在遇到无法通过索引访问 null 值的使用使用；
  - ◇ Impossible WHERE noticed after reading const tables：MySQL Query Optimizer 通过收集到的统计信息判断出不可能存在结果；
  - ◇ No tables：Query 语句中使用 FROM DUAL 或者不包含任何 FROM 子句；
  - ◇ Not exists：在某些左连接中 MySQL Query Optimizer 所通过改变原有 Query 的组成而使用的优化方法，可以部分减少数据访问次数；
  - ◇ Range checked for each record (index map: N)：通过 MySQL 官方手册的描述，当 MySQL Query Optimizer 没有发现好的可以使用的索引的时候，如果发现如果来自前面的表的列值已知，可能部分索引可以使用。对前面的表的每个行组合，MySQL 检查是否可以使

用 range 或 index_merge 访问方法来索取行。

- ◇ Select tables optimized away：当我们使用某些聚合函数来访问存在索引的某个字段的时候，MySQL Query Optimizer 会通过索引而直接一次定位到所需的数据行完成整个查询。当然，前提是在 Query 中不能有 GROUP BY 操作。如使用 MIN()或者 MAX（）的时候；
- ◇ Using filesort：当我们的 Query 中包含 ORDER BY 操作，而且无法利用索引完成排序操作的时候，MySQL Query Optimizer 不得不选择相应的排序算法来实现。
- ◇ Using index：所需要的数据只需要在 Index 即可全部获得而不需要再到表中取数据；
- ◇ Using index for group-by：数据访问和 Using index 一样，所需数据只需要读取索引即可，而当 Query 中使用了 GROUP BY 或者 DISTINCT 子句的时候，如果分组字段也在索引中，Extra 中的信息就会是 Using index for group-by；
- ◇ Using temporary：当 MySQL 在某些操作中必须使用临时表的时候，在 Extra 信息中就会出现 Using temporary 。主要常见于 GROUP BY 和 ORDER BY 等操作中。
- ◇ Using where：如果我们不是读取表的所有数据，或者不是仅仅通过索引就可以获取所有需要的数据，则会出现 Using where 信息；
- ◇ Using where with pushed condition：这是一个仅仅在 NDBCluster 存储引擎中才会出现的信息，而且还需要通过打开 Condition Pushdown 优化功能才可能会被使用。控制参数为 engine_condition_pushdown 。

这里我们通过分析示例来看一下不同的 Query 语句通过 Explain 所显示的不同信息：

我们先看一个简单的单表 Query：

```
sky@localhost : example 11:33:18> explain select count(*),max(id),min(id)
    -> from user\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: NULL
         type: NULL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
        Extra: Select tables optimized away
```

对 user 表的单表查询，查询类型为 SIMPLE，因为既没有 UNION 也不是子查询。聚合函数 MAX MIN 以及 COUNT 三者所需要的数据都可以通过索引就能够直接定位得到数据，所以整个实现的 Extra 信息为 Select tables optimized away。

再来看一个稍微复杂一点的 Query，一个子查询：

```
sky@localhost : example 11:38:48> explain select name from groups
    -> where id in ( select group_id from user_group where user_id = 1)\G
*************************** 1. row ***************************
```

```
           id: 1
  select_type: PRIMARY
        table: groups
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 50000
        Extra: Using where
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: user_group
         type: ref
possible_keys: user_group_gid_ind,user_group_uid_ind
          key: user_group_uid_ind
      key_len: 4
          ref: const
         rows: 1
        Extra: Using where
```

通过 id 信息我们可以得知 MySQL Query Optimizer 给出的执行计划是首先对 groups 进行全表扫描，然后第二步才访问 user_group 表，所使用的查询方式是 DEPENDENT SUBQUERY，对所需数据的访问方式是索引扫描，由于过滤条件是一个整数，所以索引扫描的类型为 ref，过滤条件是 const。可以使用的索引有两个，一个是基于 user_id，另一个则是基于 group_id 的。为什么基于 group_id 的索引 user_group_gid_ind 也被列为可选索引了呢？是因为与子查询的外层查询所关联的条件是基于 group_id 的。当然，最后 MySQL Query Optimizer 还是选择了使用基于 user_id 的索引 user_group_uid_ind。

由于篇幅关系，这里就不再继续举例了，大家可以通过自行通过 Explain 功能分析各自应用环境中的各种 Query，了解他们在我们的 MySQL 中到底是怎么运行的。

8.3.2 Profiling 的使用

在本章第一节中我们还提到过通过 Query Profiler 来定位一条 Query 的性能瓶颈，这里我们再详细介绍一下 Profiling 的用途及使用方法。

要想优化一条 Query，我们就需要清楚的知道这条 Query 的性能瓶颈到底在哪里，是消耗的 CPU 计算太多，还是需要的的 IO 操作太多？要想能够清楚的了解这些信息，在 MySQL 5.0 和 MySQL 5.1 正式版中已经可以非常容易做到了，那就是通过 Query Profiler 功能。

MySQL 的 Query Profiler 是一个使用非常方便的 Query 诊断分析工具，通过该工具可以获取一条 Query 在整个执行过程中多种资源的消耗情况，如 CPU，IO，IPC，SWAP 等，以及发生的 PAGE FAULTS，

CONTEXT SWITCHE 等等，同时还能得到该 Query 执行过程中 MySQL 所调用的各个函数在源文件中的位置。下面我们看看 Query Profiler 的具体用法。

1、开启 profiling 参数

```
root@localhost : (none) 10:53:11> set profiling=1;
Query OK, 0 rows affected (0.00 sec)
```

通过执行 “set profiling”命令，可以开启关闭 Query Profiler 功能。

2、执行 Query

```
    ... ...
root@localhost : test 07:43:18> select status,count(*)
    -> from test_profiling group by status;
+---------------+----------+
| status        | count(*) |
+---------------+----------+
| st_xxx1       |       27 |
| st_xxx2       |     6666 |
| st_xxx3       |   292887 |
| st_xxx4       |       15 |
+---------------+----------+
5 rows in set (1.11 sec)
    ... ...
```

在开启 Query Profiler 功能之后，MySQL 就会自动记录所有执行的 Query 的 profile 信息了。

3、获取系统中保存的所有 Query 的 profile 概要信息

```
root@localhost : test 07:47:35> show profiles;
+----------+-----------+-----------------------------------------------------------+
| Query_ID | Duration  | Query                                                     |
+----------+-----------+-----------------------------------------------------------+
|        1 | 0.00183100 | show databases                                           |
|        2 | 0.00007000 | SELECT DATABASE()                                        |
|        3 | 0.00099300 | desc test                                                |
|        4 | 0.00048800 | show tables                                              |
|        5 | 0.00430400 | desc test_profiling                                      |
|        6 | 1.90115800 | select status,count(*) from test_profiling group by status |
+----------+-----------+-----------------------------------------------------------+
3 rows in set (0.00 sec)
```

通过执行 “SHOW PROFILE” 命令获取当前系统中保存的多个 Query 的 profile 的概要信息。

4、针对单个 Query 获取详细的 profile 信息。

在获取到概要信息之后，我们就可以根据概要信息中的 Query_ID 来获取某个 Query 在执行过程中

详细的 profile 信息了，具体操作如下：

```
root@localhost : test 07:49:24> show profile cpu, block io for query 6;
+----------------------+----------+----------+------------+--------------+---------------+
| Status               | Duration | CPU_user | CPU_system | Block_ops_in | Block_ops_out |
+----------------------+----------+----------+------------+--------------+---------------+
| starting             | 0.000349 | 0.000000 |   0.000000 |            0 |             0 |
| Opening tables       | 0.000012 | 0.000000 |   0.000000 |            0 |             0 |
| System lock          | 0.000004 | 0.000000 |   0.000000 |            0 |             0 |
| Table lock           | 0.000006 | 0.000000 |   0.000000 |            0 |             0 |
| init                 | 0.000023 | 0.000000 |   0.000000 |            0 |             0 |
| optimizing           | 0.000002 | 0.000000 |   0.000000 |            0 |             0 |
| statistics           | 0.000007 | 0.000000 |   0.000000 |            0 |             0 |
| preparing            | 0.000007 | 0.000000 |   0.000000 |            0 |             0 |
| Creating tmp table   | 0.000035 | 0.000999 |   0.000000 |            0 |             0 |
| executing            | 0.000002 | 0.000000 |   0.000000 |            0 |             0 |
| Copying to tmp table | 1.900619 | 1.030844 |   0.197970 |          347 |           347 |
| Sorting result       | 0.000027 | 0.000000 |   0.000000 |            0 |             0 |
| Sending data         | 0.000017 | 0.000000 |   0.000000 |            0 |             0 |
| end                  | 0.000002 | 0.000000 |   0.000000 |            0 |             0 |
| removing tmp table   | 0.000007 | 0.000000 |   0.000000 |            0 |             0 |
| end                  | 0.000002 | 0.000000 |   0.000000 |            0 |             0 |
| query end            | 0.000003 | 0.000000 |   0.000000 |            0 |             0 |
| freeing items        | 0.000029 | 0.000000 |   0.000000 |            0 |             0 |
| logging slow query   | 0.000001 | 0.000000 |   0.000000 |            0 |             0 |
| logging slow query   | 0.000002 | 0.000000 |   0.000000 |            0 |             0 |
| cleaning up          | 0.000002 | 0.000000 |   0.000000 |            0 |             0 |
+----------------------+----------+----------+------------+--------------+---------------+
```

上面的例子中是获取 CPU 和 Block IO 的消耗，非常清晰，对于定位性能瓶颈非常适用。希望得到取其他的信息，都可以通过执行 “SHOW PROFILE *** FOR QUERY n” 来获取，各位读者朋友可以自行测试熟悉。

8.4 合理设计并利用索引

索引，可以说是数据库相关优化尤其是在 Query 优化中最常用的优化手段之一了。但是很多人在大部分时候都只是大概了解索引的用途，知道索引能够让 Query 执行的更快，而并不知道为什么会更快。尤其是索引的实现原理，存储方式，以及不同索引之间的区别等就更不是太清楚了。正因为索引对我们的 Query 性能影响很大，所以我们更应该深入理解 MySQL 中索引的基本实现，以及不同索引之间的区别，才能分析出如何设计出最优的索引来最大幅度的提升 Query 的执行效率。

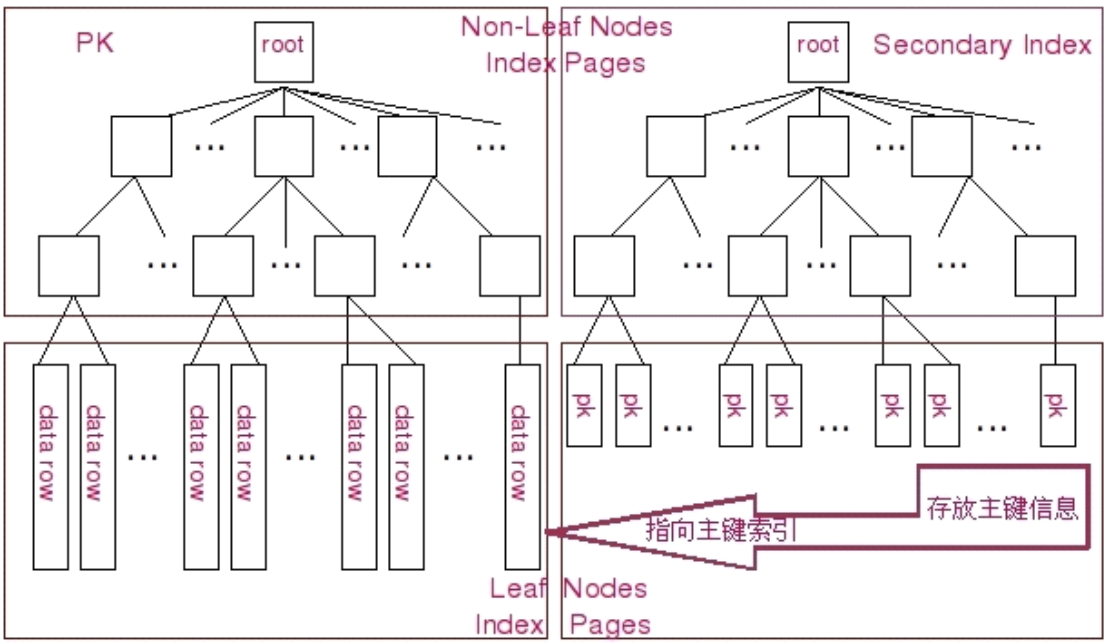
在 MySQL 中，主要有四种类型的索引，分别为：B-Tree 索引，Hash 索引，Fulltext 索引和 R-Tree 索引，下面针对这四种索引的基本实现方式及存储结构做一个大概的分析。

B-Tree 索引

B-Tree 索引是 MySQL 数据库中使用最为频繁的索引类型，除了 Archive 存储引擎之外的其他所有的存储引擎都支持 B-Tree 索引。不仅仅在 MySQL 中是如此，实际上在其他的很多数据库管理系统中 B-Tree 索引也同样是作为最主要的索引类型，这主要是因为 B-Tree 索引的存储结构在数据库的数据检索中有非常优异的表现。

一般来说，MySQL 中的 B-Tree 索引的物理文件大多都是以 Balance Tree 的结构来存储的，也就是所有实际需要的数据都存放于 Tree 的 Leaf Node，而且到任何一个 Leaf Node 的最短路径的长度都是完全相同的，所以我们大家都称之为 B-Tree 索引当然，可能各种数据库（或 MySQL 的各种存储引擎）在存放自己的 B-Tree 索引的时候会对存储结构稍作改造。如 Innodb 存储引擎的 B-Tree 索引实际使用的存储结构实际上是 B+Tree，也就是在 B-Tree 数据结构的基础上做了很小的改造，在每一个 Leaf Node 上面出了存放索引键的相关信息之外，还存储了指向与该 Leaf Node 相邻的后一个 Leaf Node 的指针信息，这主要是为了加快检索多个相邻 Leaf Node 的效率考虑。

在 Innodb 存储引擎中，存在两种不同形式的索引，一种是 Cluster 形式的主键索引（Primary Key），另外一种则是和其他存储引擎（如 MyISAM 存储引擎）存放形式基本相同的普通 B-Tree 索引，这种索引在 Innodb 存储引擎中被称为 Secondary Index。下面我们通过图示来针对这两种索引的存放形式做一个比较。



图示中左边为 Clustered 形式存放的 Primary Key，右侧则为普通的 B-Tree 索引。两种索引在 Root Node 和 Branch Nodes 方面都还是完全一样的。而 Leaf Nodes 就出现差异了。在 Primary Key 中，Leaf Nodes 存放的是表的实际数据，不仅仅包括主键字段的数据，还包括其他字段的数据，整个数据以主键值有序的排列。而 Secondary Index 则和其他普通的 B-Tree 索引没有太大的差异，只是在 Leaf Nodes 出了存放索引键的相关信息外，还存放了 Innodb 的主键值。

所以，在 Innodb 中如果通过主键来访问数据效率是非常高的，而如果是通过 Secondary Index 来访问数据的话，Innodb 首先通过 Secondary Index 的相关信息，通过相应的索引键检索到 Leaf Node 之后，需要再通过 Leaf Node 中存放的主键值再通过主键索引来获取相应的数据行。

MyISAM 存储引擎的主键索引和非主键索引差别很小，只不过是主键索引的索引键是一个唯一且非空的键而已。而且 MyISAM 存储引擎的索引和 Innodb 的 Secondary Index 的存储结构也基本相同，主要的区别只是 MyISAM 存储引擎在 Leaf Nodes 上面出了存放索引键信息之外，再存放能直接定位到 MyISAM 数据文件中相应的数据行的信息（如 Row Number），但并不会存放主键的键值信息。

Hash 索引

Hash 索引在 MySQL 中使用的并不是很多，目前主要是 Memory 存储引擎使用，而且在 Memory 存储引擎中将 Hash 索引作为默认的索引类型。所谓 Hash 索引，实际上就是通过一定的 Hash 算法，将需要索引的键值进行 Hash 运算，然后将得到的 Hash 值存入一个 Hash 表中。然后每次需要检索的时候，都会将检索条件进行相同算法的 Hash 运算，然后再和 Hash 表中的 Hash 值进行比较并得出相应的信息。

在 Memory 存储引擎中，MySQL 还支持非唯一的 Hash 索引。可能很多人会比较惊讶，如果是非唯一的 Hash 索引，那相同的值该如何处理呢？在 Memory 存储引擎的 Hash 索引中，如果遇到非唯一值，存储引擎会将他们链接到同一个 hash 键值下以一个 链表的形式存在，然后在取得实际键值的时候时候再过滤不符合的键。

由于 Hash 索引结构的特殊性，其检索效率非常的高，索引的检索可以一次定位，而不需要像 B-Tree 索引需要从根节点再到枝节点最后才能访问到页节点这样多次 IO 访问，所以 Hash 索引的效率要远高于 B-Tree 索引。

可能很多人又会有疑问了，既然 Hash 索引的效率要比 B-Tree 高很多，为什么大家不都用 Hash 索引而还要使用 B-Tree 索引呢？任何事物都是有两面性的，，Hash 索引也一样，虽然 Hash 索引检索效率非常之高，但是 Hash 索引本身由于其实的特殊性也带来了很多限制和弊端，主要有以下这些：

1. Hash 索引仅仅只能满足“=”，“IN”和“<=>”查询，不能使用范围查询；
   由于 Hash 索引所比较的是进行 Hash 运算之后的 Hash 值，所以 Hash 索引只能用于等值的过滤，而不能用于基于范围的过滤，因为经过相应的 Hash 算法处理之后的 Hash 值的大小关系，并不能保证还和 Hash 运算之前完全一样。
2. Hash 索引无法被利用来避免数据的排序操作；
   由于 Hash 索引中存放的是经过 Hash 计算之后的 Hash 值，而且 Hash 值的大小关系并不一定和 Hash 运算前的键值的完全一样，所以数据库无法利用索引的数据来避免任何和排序运算；
3. Hash 索引不能利用部分索引键查询；
   对于组合索引，Hash 索引在计算 Hash 值的时候是组合索引键合并之后再一起计算 Hash 值，而不是单独计算 Hash 值，所以当我们通过组合索引的前面一个或几个索引键进行查询的时候，Hash 索引也无法被利用到；
4. Hash 索引在任何时候都不能避免表扫面；
   前面我们已经知道，Hash 索引是将索引键通过 Hash 运算之后，将 Hash 运算结果的 Hash 值和所对应的行指针信息存放于一个 Hash 表中，而且由于存在不同索引键存在相同 Hash 值的

可能，所以即使我们仅仅取满足某个 Hash 键值的数据的记录条数，都无法直接从 Hash 索引中直接完成查询，还是要通过访问表中的实际数据进行相应的比较而得到相应的结果。

5. Hash 索引遇到大量 Hash 值相等的情况后性能并不一定就会比 B-Tree 索引高；
对于选择性比较低的索引键，如果我们创建 Hash 索引，那么我们将会存在大量记录指针信息存与同一个 Hash 值相关连。这样要定位某一条记录的时候就会非常的麻烦，可能会浪费非常多次表数据的访问，而造成整体性能的地下。

Full-text 索引

Full-text 索引也就是我们常说的全文索引，目前在 MySQL 中仅有 MyISAM 存储引擎支持，而且也并不是所有的数据类型都支持全文索引。目前来说，仅有 CHAR，VARCHAR 和 TEXT 这三种数据类型的列可以建 Full-text 索引。

一般来说，Fulltext 索引主要用来替代效率低下的 LIKE '%***%' 操作。实际上，Full-text 索引并不只是能简单的替代传统的全模糊 LIKE 操作，而且能通过多字段组合的 Full-text 索引一次全模糊匹配多个字段。

Full-text 索引和普通的 B-Tree 索引的实现区别较大，虽然他同样是以 B-Tree 形式来存放索引数据，但是他并不是通过字段内容的完整匹配，而是通过特定的算法，将字段数据进行分隔后再进行的索引。一般来说 MySQL 系统会按照四个字节来分隔。在整个 Full-text 索引中，存储内容被分为两部分，一部分是分隔前的索引字符串数据集合，另一部分是分隔后的词（或者词组）的索引信息。所以，Full-text 索引中，真正在 B-Tree 索引细细中的并不是我们表中的原始数据，而是分词之后的索引数据。在 B-Tree 索引的节点信息中，存放了各个分隔后的词信息，以及指向包含该词的分隔前字符串信息在索引数据集合中的位置信息。

Full-text 索引不仅仅能实现模糊匹配查找，在实现了基于自然语言的的匹配度查找。当然，这个匹配读到底有多准确就需要读者朋友去自行验证了。Full-text 通过一些特定的语法信息，针对自然语言做了各种相应规则的匹配，最后给出非负的匹配值。

此外，有一点是需要大家注意的，MySQL 目前的 Full-text 索引在中文支持方面还不太好，需要借助第三方的补丁或者插件来完成。而且 Full-text 的创建所消耗的资源也是比较大的，所以在应用于实际生产环境之前还是尽量做好评估。

关于 Full-text 的实际使用方法由于不是本书的重点，感兴趣的读者朋友可以自行参阅 MySQL 关于 Full-text 相关的使用手册来了解更为详尽的信息。

R-Tree 索引

R-Tree 索引可能是我们在其他数据库中很少见到的一种索引类型，主要用来解决空间数据检索的问题。

在 MySQL 中，支持一种用来存放空间信息的数据类型 GEOMETRY，且基于 OpenGIS 规范。在 MySQL5.0.16 之前的版本中，仅仅 MyISAM 存储引擎支持该数据类型，但是从 MySQL5.0.16 版本开始，

BDB，Innodb，NDBCluster 和 Archive 存储引擎也开始支持该数据类型。当然，虽然多种存储引擎都开始支持 GEOMETRY 数据类型，但是仅仅之后 MyISAM 存储引擎支持 R-Tree 索引。

在 MySQL 中采用了具有二次分裂特性的 R-Tree 来索引空间数据信息，然后通过几何对象（MRB）信息来创建索引。

虽然仅仅只有 MyISAM 存储引擎支持空间索引（R-Tree Index），但是如果我们是精确的等值匹配，创建在空间数据上面的 B-Tree 索引同样可以起到优化检索的效果，空间索引的主要优势在于当我们使用范围查找的时候，可以利用到 R-Tree 索引，而这时候，B-Tree 索引就无能为力了。

对于 R-Tree 索引的详细介绍和使用信息清参阅 MySQL 使用手册。

索引的利弊与如何判定是否需要索引

相信没一位读者朋友都知道索引能够极大的提高我们数据检索的效率，让我们的 Query 执行的更快，但是可能并不是每一位朋友都清楚索引在极大提高检索效率的同时，也给我们的数据库带来了一些负面的影响。下面我们就分别对 MySQL 中索引的利与弊做一个简单的分析。

索引的利处

索引能够给我们带来的最大益处可能读者朋友基本上都有一定的了解，但是我相信并不是每一位读者朋友都能够了解的比较全面。很多朋友对数据库中的索引的认识可能主要还是只限于“能够提高数据检索的效率，降低数据库的 IO 成本”。

确实，在数据库中个表的某个字段创建索引，所带来的最大益处就是将该字段作为检索条件的时候可以极大的提高检索效率，加快检索时间，降低检索过程中所需要读取的数据量。但是索引所给我们带来的收益只是提高表数据的检索效率吗？当然不是，索引还有一个非常重要的用途，那就是降低数据的排序成本。

我们知道，每个索引中索引数据都是按照索引键键值进行排序后存放的，所以，当我们的 Query 语句中包含排序分组操作的时候，如果我们的排序字段和索引键字段刚好一致，MySQL Query Optimizer 就会告诉 mysqld 在取得数据之后不用排序了，因为根据索引取得的数据已经是满足客户的排序要求。

那如果是分组操作呢？分组操作没办法直接利用索引完成。但是分组操作是需要先进行排序然后才分组的，所以当我们的 Query 语句中包含分组操作，而且分组字段也刚好和索引键字段一致，那么 mysqld 同样可以利用到索引已经排好序的这个特性而省略掉分组中的排序操作。

排序分组操作主要消耗的是我们的内存和 CPU 资源，如果我们能够在进行排序分组操作中利用好索引，将会极大的降低 CPU 资源的消耗。

索引的弊端

索引的益处我们都已经清楚了，但是我们不能光看到索引给我们带来的这么多益处之后就认为索引是解决 Query 优化的圣经，只要发现 Query 运行不够快就将 WHERE 子句中的条件全部放在索引中。

确实，索引能够极大的提高数据检索效率，也能够改善排序分组操作的性能，但是我们不能忽略的一个问题就是索引是完全独立于基础数据之外的一部分数据。假设我们在 Table ta 中的 Column ca 创建了索引 idx_ta_ca，那么任何更新 Column ca 的操作，MySQL 都需要在更新表中 Column ca 的同时，也更新 Column ca 的索引数据，调整因为更新所带来键值变化后的索引信息。而如果我们没有对 Column ca 进行索引的话,MySQL 所需要做的仅仅只是更新表中 Column ca 的信息。这样，所带来的最明显的资源消耗就是增加了更新所带来的 IO 量和调整索引所致的计算量。此外，Column ca 的索引 idx_ta_ca 是需要占用存储空间的，而且随着 Table ta 数据量的增长，idx_ta_ca 所占用的空间也会不断增长。所以索引还会带来存储空间资源消耗的增长。

如何判定是否需要创建索引

在了解了索引的利与弊之后，我们知道了索引并不是越多越好，知道了索引也是会带来副作用的。那我们到底该如何来判断某个索引是否应该创建呢？

实际上，并没有一个非常明确的定律可以清晰的定义出什么字段应该创建索引什么字段不该创建索引。因为我们的应用场景实在是太复杂，存在太多的差异。当然，我们还是仍然能够找到几点基本的判定策略来帮助我们分析是否需要创建索引。

- 较频繁的作为查询条件的字段应该创建索引；
  提高数据查询检索的效率最有效的办法就是减少需要访问的数据量，从上面所了解到的索引的益处中我们知道了，索引正是我们减少通过索引键字段作为查询条件的 Query 的 IO 量的最有效手段。所以一般来说我们应该为较为频繁的查询条件字段创建索引。

- 唯一性太差的字段不适合单独创建索引，即使频繁作为查询条件；
  唯一性太差的字段主要是指哪些呢？如状态字段，类型字段等等这些字段中存方的数据可能总共就是那么几个几十个值重复使用，每个值都会存在于成千上万或是更多的记录中。对于这类字段，我们完全没有必要创建单独的索引的。因为即使我们创建了索引，MySQL Query Optimizer 大多数时候也不会去选择使用，如果什么时候 MySQL Query Optimizer 抽了一下风选择了这种索引，那么非常遗憾的告诉你，这可能会带来极大的性能问题。由于索引字段中每个值都含有大量的记录，那么存储引擎在根据索引访问数据的时候会带来大量的随机 IO，甚至有些时候可能还会出现大量的重复 IO。

  这主要是由于数据基于索引扫描的特点所引起的。当我们通过索引访问表中的数据的时候，MySQL 会按照索引键的键值的顺序来依序进行访问。一般来说每个数据页中大都会存放多条记录，但是这些记录可能大多数都不会是和你所使用的索引键的键值顺序一致。

  假如有以下场景，我们通过索引查找键值为 A 和 B 的某些数据。当我们先通过 A 键值找到第一条满足要求的记录后，我们会读取这条记录所在的 X 数据页，然后我们继续往下查找索引，发现 A 键值所对应的另外一条记录也满足我们的要求，但是这条记录不在 X 数据页上面，而在 Y 数据页上面，这时候存储引擎就会丢弃 X 数据页，而读取 Y 数据页。如此继续一直到查找完 A 键值所对应的所有记录。然后轮到 B 键值了，这时候发现正在查找的记录又在 X 数据页上面，可之前读取的 X 数据页已经被丢弃了，只能再次读取 X 数据页。这时候，实际上已经出现重复读取 X 数据页两次了。在继续往后的查找中，可能还会出现一次又一次的重复读取。

这无疑极大的给存储引擎增大了 IO 访问量。

不仅如此，如果一个键值对应了太多的数据记录，也就是说通过该键值会返回占整个表比例很大的记录的时候，由于根据索引扫描产生的都是随机 IO，其效率比进行全表扫描的顺序 IO 的效率要差很多，即使不会出现重复 IO 的读取，同样会造成整体 IO 性能的下降。

很多比较有经验的 Query 调优专家经常说，当一条 Query 所返回的数据超过了全表的 15% 的时候，就不应该再使用索引扫描来完成这个 Query 了。对于“15%”这个数字我们并不能判定是否很准确，但是之少侧面证明了唯一性太差的字段并不适合创建索引。

- 更新非常频繁的字段不适合创建索引；

上面在索引的弊端中我们已经分析过了，索引中的字段被更新的时候，不仅仅需要更新表中的数据，同时还要更新索引数据，以确保索引信息是准确的。这个问题所带来的是 IO 访问量的较大增加，不仅仅影响更新 Query 的响应时间，还会影响整个存储系统的资源消耗，加大整个存储系统的负载。

当然，并不是存在更新的字段就比适合创建索引，从上面判定策略的用语上面也可以看出，是“非常频繁”的字段。到底什么样的更新频率应该算是“非常频繁”呢？每秒，每分钟，还是每小时呢？说实话，这个还真挺难定义的。很多时候还是通过比较同一时间段内被更新的次数和利用该字段作为条件的查询次数来判断，如果通过该字段的查询并不是很多，可能几个小时或者是更长才会执行一次，而更新反而比查询更频繁，那这样的字段肯定不适合创建索引。反之，如果我们通过该字段的查询比较频繁，而且更新并不是特别多，比如查询十几二十次或是更多才可能会产生一次更新，那我个人觉得更新所带来的附加成本也是可以接受的。

- 不会出现在 WHERE 子句中的字段不该创建索引；

不会还有人会问为什么吧？自己也觉得这是废话了，哈哈！

单键索引还是组合索引

在大概了解了一下 MySQL 各种类型的索引以及索引本身的利弊与判断一个字段是否需要创建索引之后，我们就需要着手创建索引来优化我们的 Query 了。在很多时候，我们的 WHERE 子句中的过滤条件并不只是针对于单一的某个字段，而是经常会有多个字段一起作为查询过滤条件存在于 WHERE 子句中。在这种时候，我们就必须要作出判断，是该仅仅为过滤性最好的字段建立索引还是该在所有字段（过滤条件中的）上面建立一个组合索引呢？

对于这种问题，很难有一个绝对的定论，我们需要从多方面来分析考虑，平衡两种方案各自的优劣，然后选择一种最佳的方案来解决。因为从上一节中我们了解到了索引在提高某些查询的性能的同时，也会让某些更新的效率下降。而组合索引中因为有多个字段的存在，理论上被更新的可能性肯定比单键索引要大很多，这样可能带来的附加成本也就比单键索引要高。但是，当我们的 WHERE 子句中的查询条件含有多个字段的时候，通过这多个字段共同组成的组合索引的查询效率肯定比仅仅只用过滤条件中的某一个字段创建的索引要高。因为通过单键索引所能过滤的数据并不完整，和通过组合索引相比，存储引擎需要访问更多的记录数，自然就会访问更多的数据量，也就是说需要更高的 IO 成本。

可能有些朋友会说，那我们可以通过创建多个单键索引啊。确实，我们可以将 WHERE 子句中的每一个字段都创建一个单键索引。但是这样真的有效吗？在这样的情况下，MySQL Query Optimizer 大多数时候都只会选择其中的一个索引，然后放弃其他的索引。即使他选择了同时利用两个或者更多的索引通过 INDEX_MERGE 来优化查询，可能所收到的效果并不会比选择其中某一个单键索引更高效。因为如果选择通过 INDEX_MERGE 来优化查询，就需要访问多个索引，同时还要将通过访问到的几个索引进行 merge 操作，所带来的成本可能反而会比选择其中一个最有效的索引来完成查询更高。

在一般的应用场景中，只要不是其中某个过滤字段在大多数场景下都能过滤出90%以上的数据，而且其他的过滤字段会存在频繁的更新，我一般更倾向于创建组合索引，尤其是在并发量较高的场景下更是应该如此。因为当我们的并发量较高的时候，即使我们为每个 Query 节省很少的 IO 消耗，但因为执行量非常大，所节省的资源总量仍然是非常可观的。

当然，我们创建组合索引并不是说就需要将查询条件中的所有字段都放在一个索引中，我们还应该尽量让一个索引被多个 Query 语句所利用，尽量减少同一个表上面索引的数量，减少因为数据更新所带来的索引更新成本，同时还可以减少因为索引所消耗的存储空间。

此外，MySQL 还为我们提供了一个减少优化索引自身的功能，那就是前缀索引。在 MySQL 中，我们可以仅仅使用某个字段的前面部分内容做为索引键来索引该字段，来达到减小索引占用的存储空间和提高索引访问的效率。当然，前缀索引的功能仅仅适用于字段前缀比较随机重复性很小的字段。如果我们需要索引的字段的前缀内容有较多的重复，索引的过滤性自然也会随之降低，通过索引所访问的数据量就会增加，这时候前缀索引虽然能够减少存储空间消耗，但是可能会造成 Query 访问效率的极大降低，反而得不偿失。

Query 的索引选择

在有些场景下，我们的 Query 由于存在多个过滤条件，而这多个过滤条件可能会存在于两个或者更多的索引中。在这种场景下，MySQL Query Optimizer 一般情况下都能够根据系统的统计信息选择出一个针对该 Query 最优的索引完成查询，但是在有些情况下，可能是由于我们的系统统计信息的不够准确完整，也可能是 MySQL Query Optimizer 自身功能的缺陷，会造成他并没有选择一个真正最优的索引而选择了其他查询效率较低的索引。在这种时候，我们就不得不通过认为干预，在 Query 中增加 Hint 提示 MySQL Query Optimizer 告诉他该使用哪个索引而不该使用哪个索引，或者通过调整查询条件来达到相同的目的。

我们这里再次通过在本章第 2 节“Query 语句优化基本思路和原则”的“仅仅使用最有效的过滤条件”中示例的基础上将 group_message 表的索引做部分调整，然后再进行分析。

在 group_message 上增加如下索引：

```
create index group_message_author_subject on group_message(author,subject(16));
```

调整后的索引信息如下（出于篇幅考虑省略了主键索引）：

```
sky@localhost : example 07:13:38> show indexes from group_message\G
   ......
*************************** 2. row ***************************
       Table: group_message
```

```
  Non_unique: 1
    Key_name: group_message_author_subject
Seq_in_index: 1
 Column_name: author
   Collation: A
 Cardinality: NULL
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
*************************** 3. row ***************************
       Table: group_message
  Non_unique: 1
    Key_name: group_message_author_subject
Seq_in_index: 2
 Column_name: subject
   Collation: A
 Cardinality: NULL
    Sub_part: 16
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
*************************** 4. row ***************************
       Table: group_message
  Non_unique: 1
    Key_name: idx_group_message_uid
Seq_in_index: 1
 Column_name: user_id
   Collation: A
 Cardinality: NULL
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
*************************** 5. row ***************************
       Table: group_message
  Non_unique: 1
    Key_name: idx_group_message_author
Seq_in_index: 1
 Column_name: author
   Collation: A
```

```
 Cardinality: NULL
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
```

从索引的 Sub_part 中，我们可以看到 subject 字段是取前 16 个字符的前缀作为索引键。下面假设我们知道某个用户的 user_id ，nick_name 和 subject 字段的部分前缀信息（weiurazs），希望通过这些条件查询出所有满足上面存在于 group_message 中的信息。我们知道存在三个索引可以被利用：idx_group_message_author ， idx_group_message_uid 和 group_message_author_subject，而且也知道每个 user_id 实际上都是和 一个 author 分别唯一对应的。所以实际上，无论是使用 user_id 和 author（nick_name）中的某一个来作为条件或者两个条件都使用，所得到的数据都是完全一样的。当然，我们还需要 subject LIKE 'weiurazs%' 这个条件来过滤 subject 相关的信息。

根据三个索引的组成，和我们的查询条件，我们知道 group_message_author_subject 索引可以让我们得到最高的检索效率，因为只有他索引了 subject 相关的信息，subject 是我们的查询必须包含的过滤条件。下面我们分别看看使用 user_id ，author 和 两者共同使用时候的执行计划。

```
sky@localhost : example 07:48:45> EXPLAIN SELECT * FROM group_message
    -> WHERE user_id = 3 AND subject LIKE 'weiurazs%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: ref
possible_keys: idx_group_message_uid
          key: idx_group_message_uid
      key_len: 4
          ref: const
         rows: 8
        Extra: Using where
1 row in set (0.00 sec)
```

很明显，这不是我们所期望的执行计划，当然我们并不能责怪 MySQL，因为我们都没有使用 author 来进行过滤，Optimizer 当然不会选择 group_message_author_subject 这个索引，这是我们自己的错。

```
sky@localhost : example 07:48:49> EXPLAIN SELECT * FROM group_message
    -> WHERE author = '3' AND subject LIKE 'weiurazs%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
```

```
         type: range
possible_keys: group_message_author_subject,idx_group_message_author
          key: idx_group_message_author
      key_len: 98
          ref: NULL
         rows: 8
        Extra: Using where
1 row in set (0.00 sec)
```

这次我们改为使用 author 作为查询条件了，可 MySQL Query Optimizer 仍然没有选择 group_message_author_subject 这个索引，即使我们通过 analyze 分析也是同样的结果。

```
sky@localhost : example 07:48:57> EXPLAIN SELECT * FROM group_message
    -> WHERE user_id = 3 AND author = '3' AND subject LIKE 'weiurazs%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: range
possible_keys: group_message_author_subject,idx_group_message_uid,
               idx_group_message_author
          key: idx_group_message_uid
      key_len: 98
          ref: NULL
         rows: 8
        Extra: Using where
1 row in set (0.00 sec)
```

同时使用 user_id 和 author 两者的时候，MySQL Query Optimizer 又再次选择了 idx_group_message_uid 这个索引，仍然不是我们期望的结果。

```
sky@localhost : example 07:51:11>  EXPLAIN SELECT * FROM group_message
    -> FORCE INDEX(idx_group_message_author_subject)
    -> WHERE user_id = 3 AND author = '3' AND subject LIKE 'weiurazs%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: range
possible_keys: group_message_author_subject
          key: group_message_author_subject
      key_len: 148
          ref: NULL
         rows: 8
```

Extra: Using where

在最后，我们不得不利用 MySQL 为我们提供的在优化 Query 时候所使用的高级功能，通过显式告诉 MySQL Query Optimizer 我们要使用哪个索引的 Hint 功能。强制 MySQL 使用 group_message_author_subject 这个索引来完成查询，才达到我们所需要的效果。

或许有些读者会想，会不会是因为选择 group_message_author_subject 这个索引本身就不是一个最有的选择呢？大家请看下面通过 mysqlslap 进行的实际执行各条 Query 的测试结果：

```
sky@sky:~$ mysqlslap  --create-schema=example --query="SELECT * FROM group_message WHERE user_id = 3 AND subject LIKE 'weiurazs%'" --iterations=10000
Benchmark
    Average number of seconds to run all queries: 0.021 seconds
    Minimum number of seconds to run all queries: 0.010 seconds
    Maximum number of seconds to run all queries: 0.030 seconds
    Number of clients running queries: 1
    Average number of queries per client: 1

sky@sky:~$ mysqlslap  --create-schema=example --query="SELECT * FROM group_message WHERE author = '3' AND subject LIKE 'weiurazs%'" --iterations=10000
Benchmark
    Average number of seconds to run all queries: 0.025 seconds
    Minimum number of seconds to run all queries: 0.012 seconds
    Maximum number of seconds to run all queries: 0.031 seconds
    Number of clients running queries: 1
    Average number of queries per client: 1

sky@sky:~$ mysqlslap  --create-schema=example --query="SELECT * FROM group_message WHERE user_id = 3 AND author = '3' AND subject LIKE 'weiurazs%'" --iterations=10000
Benchmark
    Average number of seconds to run all queries: 0.026 seconds
    Minimum number of seconds to run all queries: 0.013 seconds
    Maximum number of seconds to run all queries: 0.030 seconds
    Number of clients running queries: 1
    Average number of queries per client: 1

sky@sky:~$ mysqlslap  --create-schema=example --query="SELECT * FROM group_message force index(group_message_author_subject) WHERE author = '3' subject LIKE 'weiurazs%'" --iterations=10000
Benchmark
    Average number of seconds to run all queries: 0.017 seconds
    Minimum number of seconds to run all queries: 0.010 seconds
    Maximum number of seconds to run all queries: 0.027 seconds
    Number of clients running queries: 1
```

Average number of queries per client: 1

我们可以清晰的看出，通过我们添加 Hint 之后选择 group_message_author_subject 这个索引的 Query 确实比其他的三条要快很多。

通过这个示例，我们可以看出在优化 Query 的时候，选择合适的索引是非常重要的，而且我们也同时实例证明了 MySQL Query Optimizer 并不是任何时候都能够选择出最佳的执行计划，在有些时候，我们不得不通过人为的手工干预来让 MySQL Query Optimizer 改变他的“想法”，而按照我们的思路走。

当然，这个示例仅仅只是告诉了我们选择合适索引的重要性，并且不能任何时候都完全相信 MySQL Query Optimizer，但并没有告诉我们到底该如何来选择一个更合适的索引。下面是我对于选择合适索引的几点建议，并不一定在任何场景下都合适，但在大多数场景下还是比较适用的。

1. 对于单键索引，尽量选择针对当前 Query 过滤性更好的索引；
2. 在选择组合索引的时候，当前 Query 中过滤性最好的字段在索引字段顺序中排列越靠前越好；
3. 在选择组合索引的时候，尽量选择可以能够包含当前 Query 的 WHERE 子句中更多字段的索引；
4. 尽可能通过分析统计信息和调整 Query 的写法来达到选择合适索引的目的而减少通过使用 Hint 人为控制索引的选择，因为这会使后期的维护成本增加，同时增加维护所带来的潜在风险。

MySQL 中索引的限制

在使用索引的同时，我们还应该了解在 MySQL 中索引存在的限制，以便在索引应用中尽可能的避开限制所带来的问题。下面列出了目前 MySQL 中索引使用相关的限制。

1. MyISAM 存储引擎索引键长度总和不能超过 1000 字节；
2. BLOB 和 TEXT 类型的列只能创建前缀索引；
3. MySQL 目前不支持函数索引；
4. 使用不等于（!= 或者 <>）的时候 MySQL 无法使用索引；
5. 过滤字段使用了函数运算后（如 abs(column)），MySQL 无法使用索引；
6. Join 语句中 Join 条件字段类型不一致的时候 MySQL 无法使用索引；
7. 使用 LIKE 操作的时候如果条件以通配符开始（ '%abc...'）MySQL 无法使用索引；
8. 使用非等值查询的时候 MySQL 无法使用 Hash 索引；
9.

在我们使用索引的时候，需要注意上面的这些限制，尤其是要注意无法使用索引的情况，因为这很容易让我们因为疏忽而造成极大的性能隐患。

8.5 Join 的实现原理及优化思路

前面我们已经了解了 MySQL Query Optimizer 的工作原理，学习了 Query 优化的基本原则和思路，理解了索引选择的技巧，这一节我们将围绕 Query 语句中使用非常频繁，且随时可能存在性能隐患的 Join 语句，继续我们的 Query 优化之旅。

Join 的实现原理

在寻找 Join 语句的优化思路之前，我们首先要理解在 MySQL 中是如何来实现 Join 的，只要理解了实现原理之后，优化就比较简单了。下面我们先分析一下 MySQL 中 Join 的实现原理。

在 MySQL 中，只有一种 Join 算法，就是大名鼎鼎的 Nested Loop Join，他没有其他很多数据库所提供的 Hash Join，也没有 Sort Merge Join。顾名思义，Nested Loop Join 实际上就是通过驱动表的结果集作为循环基础数据，然后一条一条的通过该结果集中的数据作为过滤条件到下一个表中查询数据，然后合并结果。如果还有第三个参与 Join，则再通过前两个表的 Join 结果集作为循环基础数据，再一次通过循环查询条件到第三个表中查询数据，如此往复。

下面我们将通过一个三表 Join 语句示例来说明 MySQL 的Nested Loop Join 实现方式。

注意：由于要展示 Explain 中的一个在 MySQL 5.1.18 才开始出现的输出信息（在之前版本中只是没有输出信息，实际执行过程并没有变化），所以下面的示例环境是 MySQL5.1.26。

Query 如下：

```
select m.subject msg_subject, c.content msg_content
from user_group g,group_message m,group_message_content c
where g.user_id = 1
    and m.group_id = g.group_id
    and c.group_msg_id = m.id
```

为了便于示例，我们通过如下操作为 group_message 表增加了一个 group_id 的索引：

```
create index idx_group_message_gid_uid on group_message(group_id);
```

然后看看我们的 Query 的执行计划：

```
sky@localhost : example 11:17:04> explain select m.subject msg_subject, c.content
msg_content
    -> from user_group g,group_message m,group_message_content c
    -> where g.user_id = 1
    -> and m.group_id = g.group_id
    -> and c.group_msg_id = m.id\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: g
         type: ref
possible_keys: user_group_gid_ind,user_group_uid_ind,user_group_gid_uid_ind
          key: user_group_uid_ind
      key_len: 4
          ref: const
```

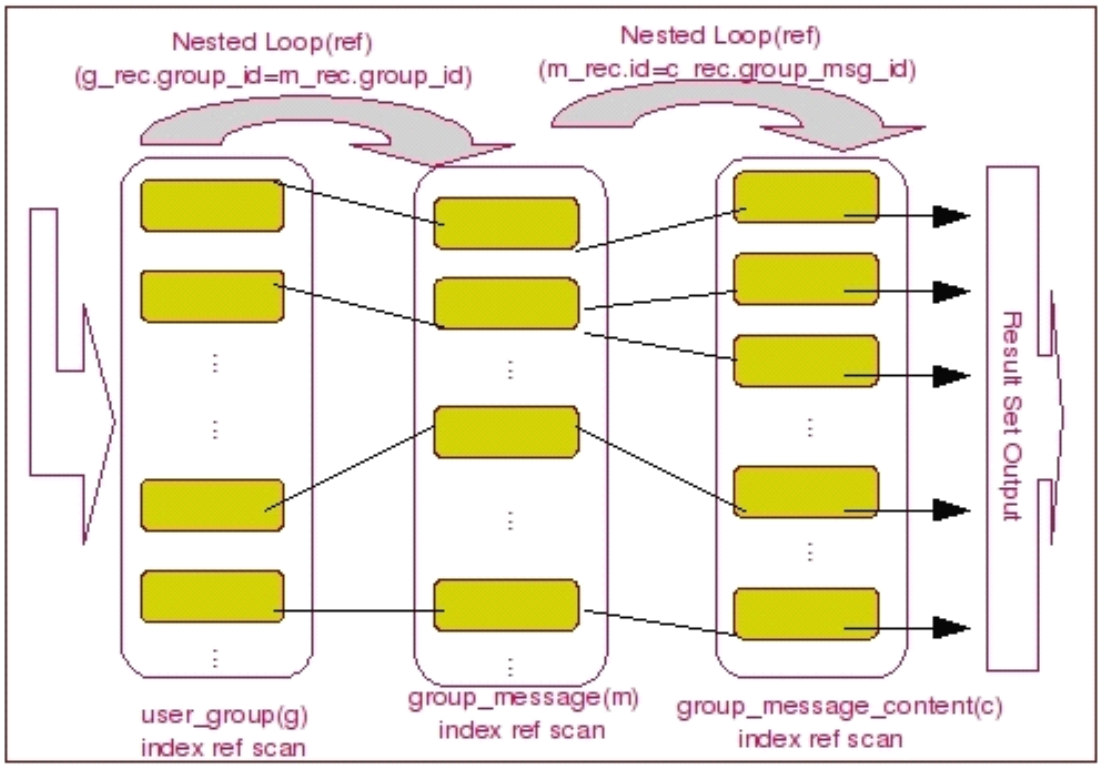
```
           rows: 2
          Extra:
*************************** 2. row ***************************
             id: 1
    select_type: SIMPLE
          table: m
           type: ref
  possible_keys: PRIMARY,idx_group_message_gid_uid
            key: idx_group_message_gid_uid
        key_len: 4
            ref: example.g.group_id
           rows: 3
          Extra:
*************************** 3. row ***************************
             id: 1
    select_type: SIMPLE
          table: c
           type: ref
  possible_keys: idx_group_message_content_msg_id
            key: idx_group_message_content_msg_id
        key_len: 4
            ref: example.m.id
           rows: 2
          Extra:
```

我们可以看出，MySQL Query Optimizer 选择了 user_group 作为驱动表，首先利用我们传入的条件 user_id 通过 该表上面的索引 user_group_uid_ind 来进行 const 条件的索引 ref 查找，然后以 user_group 表中过滤出来的结果集的 group_id 字段作为查询条件，对 group_message 循环查询，然后再通过 user_group 和 group_message 两个表的结果集中的 group_message 的 id 作为条件 与 group_message_content 的 group_msg_id 比较进行循环查询，才得到最终的结果。

这个过程可以通过如下表达式来表示：

```
for each record g_rec in table user_group that g_rec.user_id=1{
 for each record m_rec in group_message that m_rec.group_id=g_rec.group_id{
  for each record c_rec in group_message_content that c_rec.group_msg_id=m_rec.id
   pass the (g_rec.user_id, m_rec.subject, c_rec.content) row
   combination to output;
 }
}
```

下图可以更清晰的标识出实际的执行情况：



假设我们去掉 group_message_content 表上面的 group_msg_id 字段的索引，然后再看看执行计划会变成怎样：

```
sky@localhost : example 11:25:36> drop index idx_group_message_content_msg_id on
group_message_content;
Query OK, 96 rows affected (0.11 sec)

sky@localhost : example 10:21:06> explain
    -> select m.subject msg_subject, c.content msg_content
    -> from user_group g,group_message m,group_message_content c
    -> where g.user_id = 1
    -> and m.group_id = g.group_id
    -> and c.group_msg_id = m.id\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: g
         type: ref
possible_keys: idx_user_group_uid
          key: idx_user_group_uid
      key_len: 4
          ref: const
         rows: 2
        Extra:
```

```
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: m
         type: ref
possible_keys: PRIMARY,idx_group_message_gid_uid
          key: idx_group_message_gid_uid
      key_len: 4
          ref: example.g.group_id
         rows: 3
        Extra:
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: c
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 96
        Extra: Using where; Using join buffer
```

我们看到不仅仅 user_group 表的访问从 ref 变成了 ALL，此外，在最后一行的 Extra 信息从没有任何内容变成为 Using where; Using join buffer，也就是说，对于从 ref 变成 ALL 很容易理解，没有可以使用的索引的索引了嘛，当然得进行全表扫描了，Using where 也是因为变成全表扫描之后，我们需要取得的 content 字段只能通过对表中的数据进行 where 过滤才能取得，但是后面出现的 Using join buffer 是一个啥呢？

实际上，这里的 Join 正是利用到了我们在之前 “MySQL Server 性能优化”一章中所提到的一个 Cache 参数相关的内容，也就是我们通过 join_buffer_size 参数所设置的 Join Buffer。

实际上，Join Buffer 只有当我们的 Join 类型为 ALL（如示例中），index，rang 或者是 index_merge 的时候 才能够使用，所以，在我们去掉 group_message_content 表的 group_msg_id 字段的索引之前，由于 Join 是 ref 类型的，所以我们的执行计划中并没有看到有使用 Join Buffer。

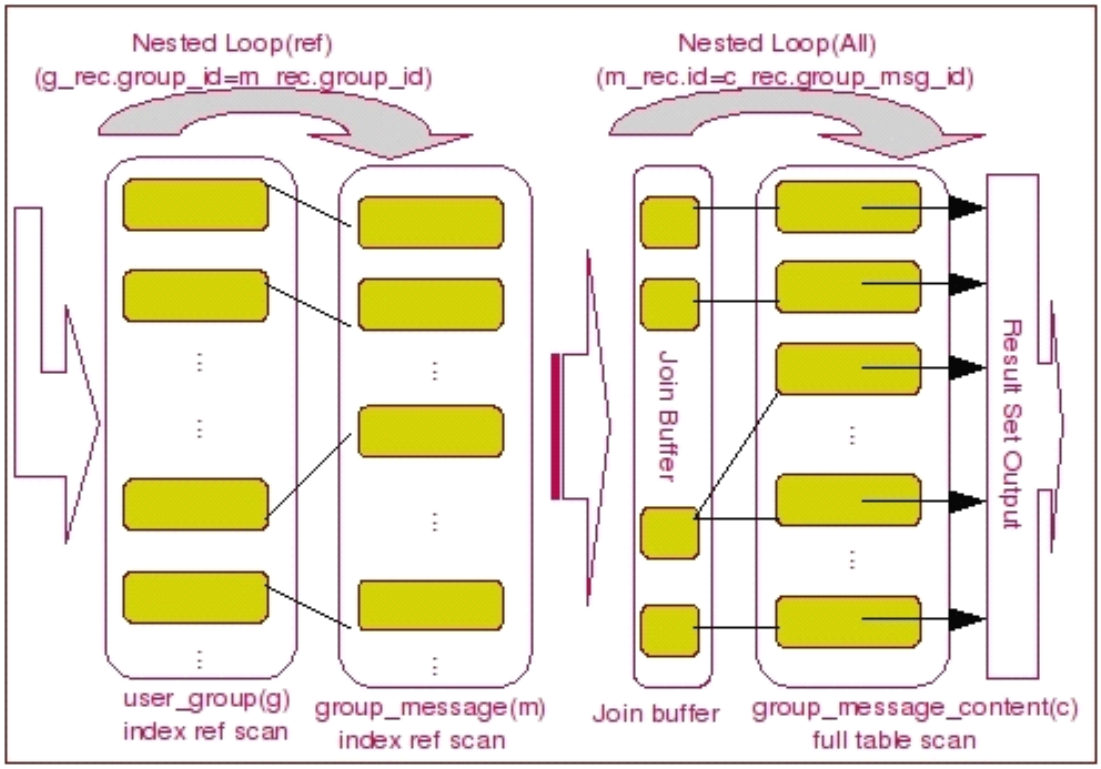
当我们使用了 Join Buffer 之后，我们可以通过下面的这个表达式描述出示例中我们的 Join 完成过程：

```
for each record g_rec in table user_group{
 for each record m_rec in group_message that m_rec.group_id=g_rec.group_id{
  put (g_rec, m_rec) into the buffer
   if (buffer is full)
     flush_buffer();
```

```
 }
}

flush_buffer(){
 for each record c_rec in group_message_content that
 c_rec.group_msg_id = c_rec.id{
  for each record in the buffer
   pass (g_rec.user_id, m_rec.subject, c_rec.content) row combination to output;
 }
 empty the buffer;
}
```

当然，如果通过类似于上面的图片来展现或许大家会觉得更容易理解一些，如下：



通过上面的示例，我想大家应该对 MySQL 中 Nested Join 的实现原理有了一个了解了，也应该清楚 MySQL 使用 Join Buffer 的方法了。当然，这里并没有涉及到 外连接的内容，实际对于外连接来说，可能存在的区别主要是连接顺序以及组合空值记录方面。

Join 语句的优化

在明白了 MySQL 中 Join 的实现原理之后，我们就比较清楚的知道该如何去优化一个一个 Join 语句了。

1. 尽可能减少 Join 语句中的 Nested Loop 的循环总次数；
   如何减少 Nested Loop 的循环总次数？最有效的办法只有一个，那就是让驱动表的结果集尽可

能的小，这也正是在本章第二节中的优化基本原则之一“永远用小结果集驱动大的结果集”。

为什么？因为驱动结果集越大，意味着需要循环的次数越多，也就是说在被驱动结果集上面所需要执行的查询检索次数会越多。比如，当两个表（表 A 和 表 B） Join 的时候，如果表 A 通过 WHERE 条件过滤后有 10 条记录，而表 B 有 20 条记录。如果我们选择表 A 作为驱动表，也就是被驱动表的结果集为 20，那么我们通过 Join 条件对被驱动表（表 B）的比较过滤就会有 10 次。反之，如果我们选择表 B 作为驱动表，则需要有 20 次对表 A 的比较过滤。

当然，此优化的前提条件是通过 Join 条件对各个表的每次访问的资源消耗差别不是太大。如果访问存在较大的差别的时候（一般都是因为索引的区别），我们就不能简单的通过结果集的大小来判断需要 Join 语句的驱动顺序，而是要通过比较循环次数和每次循环所需要的消耗的乘积的大小来得到如何驱动更优化。

2. 优先优化 Nested Loop 的内层循环；

不仅仅是在数据库的 Join 中应该做的，实际上在我们优化程序语言的时候也有类似的优化原则。内层循环是循环中执行次数最多的，每次循环节约很小的资源，在整个循环中就能节约很大的资源。

3. 保证 Join 语句中被驱动表上 Join 条件字段已经被索引；

保证被驱动表上 Join 条件字段已经被索引的目的，正是针对上面两点的考虑，只有让被驱动表的 Join 条件字段被索引了，才能保证循环中每次查询都能够消耗较少的资源，这也正是优化内层循环的实际优化方法。

4. 当无法保证被驱动表的 Join 条件字段被索引且内存资源充足的前提下，不要太吝惜 Join Buffer 的设置；

当在某些特殊的环境中，我们的 Join 必须是 All，Index，range 或者是 index_merge 类型的时候，Join Buffer 就会派上用场了。在这种情况下，Join Buffer 的大小将对整个 Join 语句的消耗起到非常关键的作用。

8.6 ORDER BY，GROUP BY 和 DISTINCT 优化

除了常规的 Join 语句之外，还有一类 Query 语句也是使用比较频繁的，那就是 ORDER BY，GROUP BY 以及 DISTINCT 这三类查询。考虑到这三类查询都涉及到数据的排序等操作，所以我将他们放在了一起，下面就针对这三类 Query 语句做基本的分析。

ORDER BY 的实现与优化

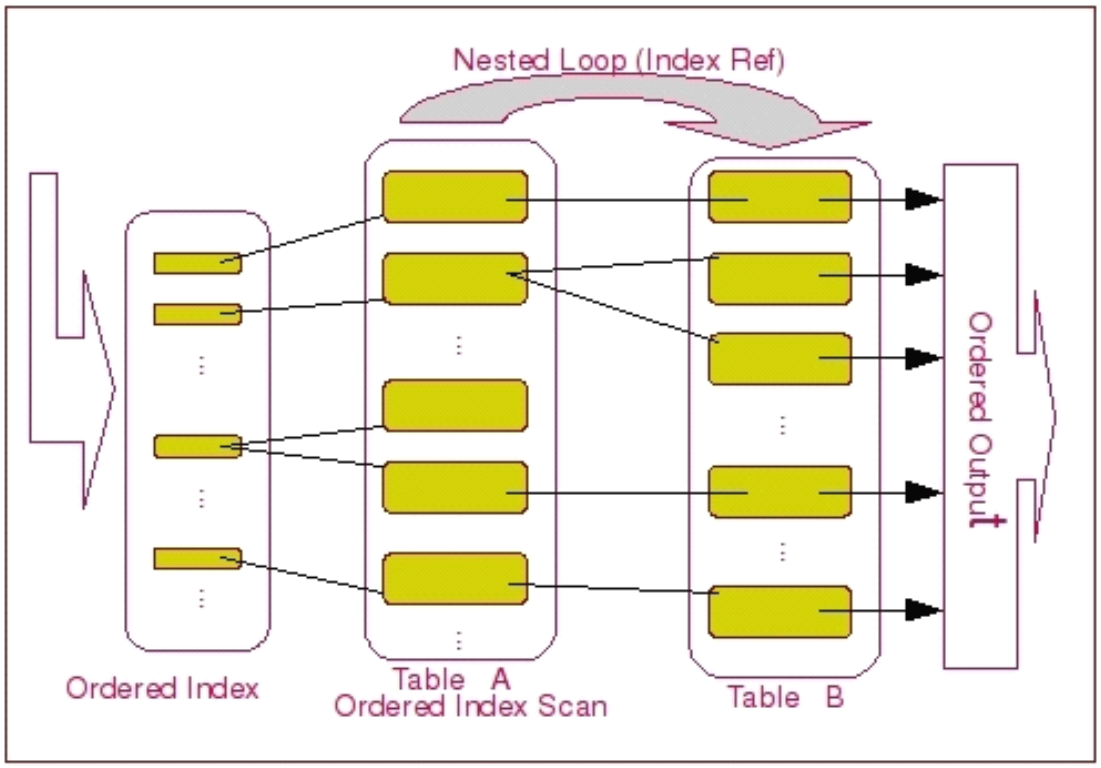
在 MySQL 中，ORDER BY 的实现有如下两种类型：

- 一种是通过有序索引而直接取得有序的数据，这样不用进行任何排序操作即可得到满足客户端要求的有序数据返回给客户端；
- 另外一种则需要通过 MySQL 的排序算法将存储引擎中返回的数据进行排序然后再将排序后的数据返回给客户端。

下面我们就针对这两种实现方式做一个简单的分析。首先分析一下第一种不用排序的实现方式。同样还是通过示例来说话吧：

```
sky@localhost : example 09:48:41> EXPLAIN
    -> SELECT m.id,m.subject,c.content
    -> FROM group_message m,group_message_content c
    -> WHERE m.group_id = 1 AND m.id = c.group_msg_id
    -> ORDER BY m.user_id\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: m
         type: ref
possible_keys: PRIMARY,idx_group_message_gid_uid
          key: idx_group_message_gid_uid
      key_len: 4
          ref: const
         rows: 4
        Extra: Using where
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: c
         type: ref
possible_keys: group_message_content_msg_id
          key: group_message_content_msg_id
      key_len: 4
          ref: example.m.id
         rows: 11
        Extra:
```

看看上面的这个 Query 语句，明明有 ORDER BY user_id，为什么在执行计划中却没有排序操作呢？其实这里正是因为 MySQL Query Optimizer 选择了一个有序的索引来进行访问表中的数据（idx_group_message_gid_uid），这样，我们通过 group_id 的条件得到的数据已经是按照 group_id 和 user_id 进行排序的了。而虽然我们的排序条件仅仅只有一个user_id，但是我们的 WHERE 条件决定了返回数据的 group_id 全部一样，也就是说不管有没有根据 group_id 来进行排序，返回的结果集都是完全一样的。我们可以通过如下的图示来描述整个执行过程：



图中的 Table A 和 Table B 分别为上面 Query 中的 group_message 和 gruop_message_content 这两个表。

这种利用索引实现数据排序的方法是 MySQL 中实现结果集排序的最佳做法，可以完全避免因为排序计算所带来的资源消耗。所以，在我们优化 Query 语句中的 ORDER BY 的时候，尽可能利用已有的索引来避免实际的排序计算，可以很大幅度的提升 ORDER BY 操作的性能。在有些 Query 的优化过程中，即使为了避免实际的排序操作而调整索引字段的顺序，甚至是增加索引字段也是值得的。当然，在调整索引之前，同时还需要评估调整该索引对其他 Query 所带来的影响，平衡整体得失。

如果没有索引利用的时候，MySQL 又如何来实现排序呢？这时候 MySQL 无法避免需要通过相关的排序算法来将存储引擎返回的数据进行排序运算了。下面我们再针对这种实现方式进行相应的分析。

在 MySQL 第二种排序实现方式中，必须进行相应的排序算法来实现数据的排序。MySQL 目前可以通过两种算法来实现数据的排序操作。

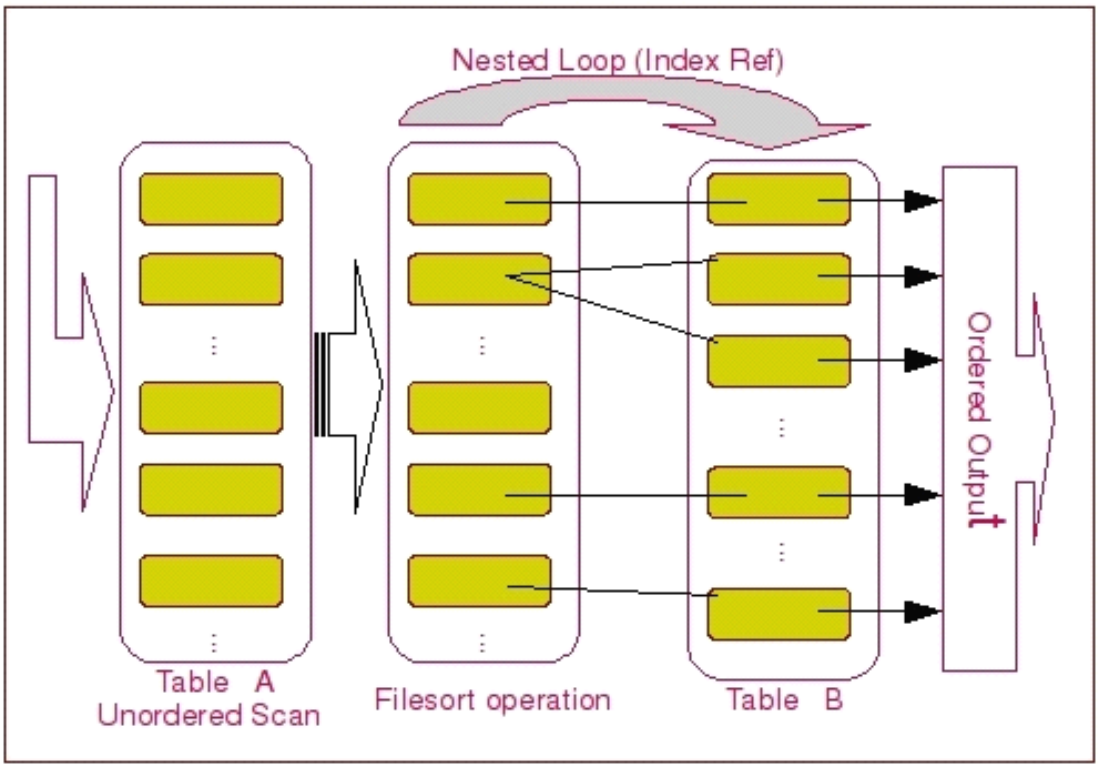
1. 取出满足过滤条件的用于排序条件的字段以及可以直接定位到行数据的行指针信息，在 Sort Buffer 中进行实际的排序操作，然后利用排好序之后的数据根据行指针信息返回表中取得客户端请求的其他字段的数据，再返回给客户端；
2. 根据过滤条件一次取出排序字段以及客户端请求的所有其他字段的数据，并将不需要排序的字段存放在一块内存区域中，然后在 Sort Buffer 中将排序字段和行指针信息进行排序，最后再利用排序后的行指针与存放在内存区域中和其他字段一起的行指针信息进行匹配合并结果集，再按照顺序返回给客户端。

上面第一种排序算法是 MySQL 一直以来就有的排序算法，而第二种则是从 MySQL4.1 版本才开始增加的改进版排序算法。第二种算法与第一种相比较，主要优势就是减少了数据的二次访问。在排序之后

不需要再一次回到表中取数据，节省了 IO 操作。当然，第二种算法会消耗更多的内存，正是一种典型的通过内存空间换取时间的优化方式。下面我们同样通过一个实例来看看当 MySQL 不得不使用排序算法的时候的执行计划，仅仅只是更改一下排序字段：

```
sky@localhost : example 10:09:06> explain
    -> select m.id,m.subject,c.content
    -> FROM group_message m,group_message_content c
    -> WHERE m.group_id = 1 AND m.id = c.group_msg_id
    -> ORDER BY m.subject\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: m
         type: ref
possible_keys: PRIMARY,idx_group_message_gid_uid
          key: idx_group_message_gid_uid
      key_len: 4
          ref: const
         rows: 4
        Extra: Using where; Using filesort
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: c
         type: ref
possible_keys: group_message_content_msg_id
          key: group_message_content_msg_id
      key_len: 4
          ref: example.m.id
         rows: 11
        Extra:
```

大概一看，好像整个执行计划并没有什么区别啊？但是细心的读者朋友可能已经发现，在 group_message 表的 Extra 信息中，多了一个“Using filesort”的信息，实际上这就是 MySQL Query Optimizer 在告诉我们，他需要进行排序操作才能按照客户端的要求返回有序的数据。执行图示如下：



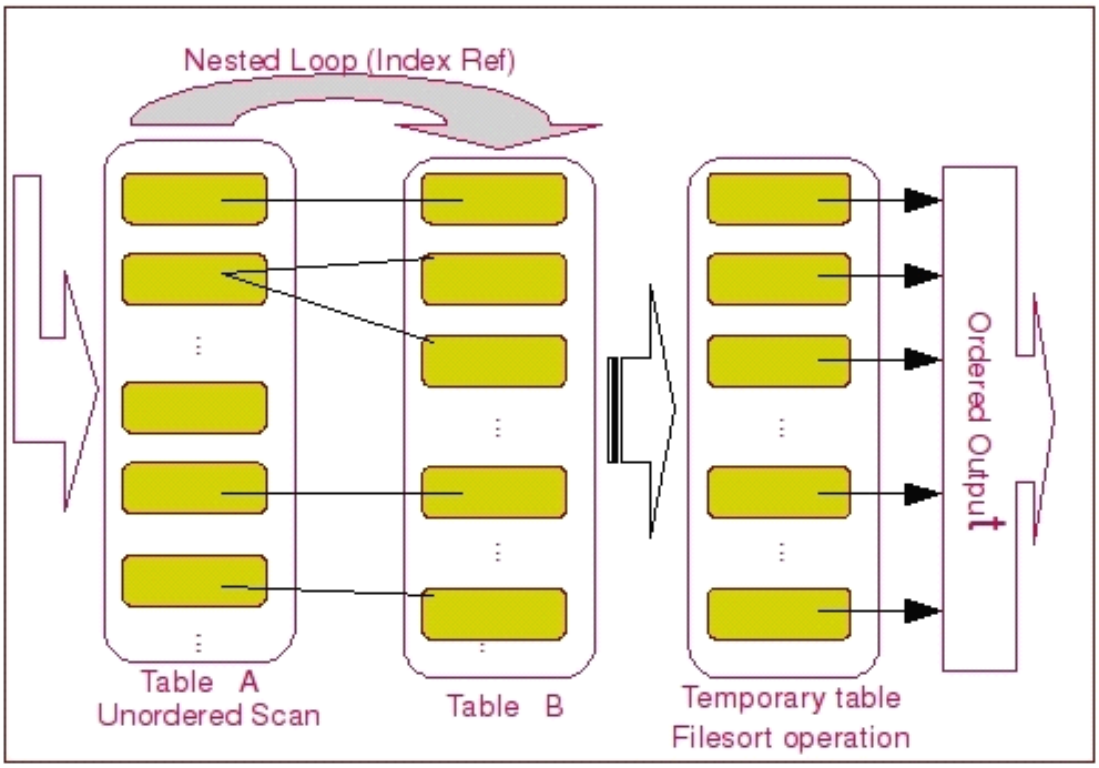
这里我们看到了，MySQL 在取得第一个表的数据之后，先根据排序条件将数据进行了一次 filesort，也就是排序操作。然后再利用排序后的结果集作为驱动结果集来通过 Nested Loop Join 访问第二个表。当然，大家不要误解，这个 filesort 并不是说通过磁盘文件进行排序，仅仅只是告诉我们进行了一个排序操作。

上面，我们看到了排序结果集来源仅仅只是单个表的比较简单的 filesort 操作。而在我们实际应用中，很多时候我们的业务要求可能并不是这样，可能需要排序的字段同时存在于两个表中，或者 MySQL 在经过一次 Join 之后才进行排序操作。这样的排序在 MySQL 中并不能简单的里利用 Sort Buffer 进行排序，而是必须先通过一个临时表将之前 Join 的结果集存放入临时表之后在将临时表的数据取到 Sort Buffer 中进行操作。下面我们通过再次更改排序要求来示例这样的执行计划，当我们选择通过 group_message_content 表上面的 content 字段来进行排序之后：

```
sky@localhost : example 10:22:42> explain
    -> select m.id,m.subject,c.content
    -> FROM group_message m,group_message_content c
    -> WHERE m.group_id = 1 AND m.id = c.group_msg_id
    -> ORDER BY c.content\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: m
         type: ref
possible_keys: PRIMARY,idx_group_message_gid_uid
          key: idx_group_message_gid_uid
```

```
       key_len: 4
           ref: const
          rows: 4
         Extra: Using temporary; Using filesort
*************************** 2. row ***************************
            id: 1
  select_type: SIMPLE
         table: c
          type: ref
possible_keys: group_message_content_msg_id
           key: group_message_content_msg_id
       key_len: 4
           ref: example.m.id
          rows: 11
         Extra:
```

这时候的执行计划中出现了“Using temporary”，正是因为我们的排序操作需要在两个表 Join 之后才能进行，下图展示了这个 Query 的执行过程：



首先是 Table A 和 Table B 进行 Join，然后结果集进入临时表，再进行 filesort，最后得到有序的结果集数据返回给客户端。

上面我们通过两个不同的示例展示了当 MySQL 无法避免要使用相应的排序算法进行排序操作的时候的实现原理。虽然在排序过程中所使用的排序算法有两种，但是两种排序的内部实现机制大体上差不

多。

当我们无法避免排序操作的时候，我们又该如何来优化呢？很显然，我们应该尽可能让 MySQL 选择使用第二种算法来进行排序。这样可以减少大量的随机 IO 操作，很大幅度的提高排序工作的效率。

1. 加大 max_length_for_sort_data 参数的设置；

   在 MySQL 中，决定使用第一种老式的排序算法还是新的改进算法的依据是通过参数 max_length_for_sort_data 来决定的。当我们所有返回字段的最大长度小于这个参数值的时候，MySQL 就会选择改进后的排序算法，反之，则选择老式的算法。所以，如果我们有充足的内存让 MySQL 存放需要返回的非排序字段的时候，可以加大这个参数的值来让 MySQL 选择使用改进版的排序算法。

2. 去掉不必要的返回字段；

   当我们的内存并不是很充裕的时候，我们不能简单的通过强行加大上面的参数来强迫 MySQL 去使用改进版的排序算法，因为如果那样可能会造成 MySQL 不得不将数据分成很多段然后进行排使用序，这样的结果可能会得不偿失。在这种情况下，我们就需要去掉不必要的返回字段，让我们的返回结果长度适应 max_length_for_sort_data 参数的限制。

3. 增大 sort_buffer_size 参数设置；

   增大 sort_buffer_size 并不是为了让 MySQL 可以选择改进版的排序算法，而是为了让 MySQL 可以尽量减少在排序过程中对需要排序的数据进行分段，因为这样会造成 MySQL 不得不使用临时表来进行交换排序。

GROUP BY 的实现与优化

由于 GROUP BY 实际上也同样需要进行排序操作，而且与 ORDER BY 相比，GROUP BY 主要只是多了排序之后的分组操作。当然，如果在分组的时候还使用了其他的一些聚合函数，那么还需要一些聚合函数的计算。所以，在 GROUP BY 的实现过程中，与 ORDER BY 一样也可以利用到索引。

在 MySQL 中，GROUP BY 的实现同样有多种（三种）方式，其中有两种方式会利用现有的索引信息来完成 GROUP BY，另外一种为完全无法使用索引的场景下使用。下面我们分别针对这三种实现方式做一个分析。

1. 使用松散（Loose）索引扫描实现 GROUP BY

何谓松散索引扫描实现 GROUP BY 呢？实际上就是当 MySQL 完全利用索引扫描来实现 GROUP BY 的时候，并不需要扫描所有满足条件的索引键即可完成操作得出结果。

下面我们通过一个示例来描述松散索引扫描实现 GROUP BY，在示例之前我们需要首先调整一下 group_message 表的索引，将 gmt_create 字段添加到 group_id 和 user_id 字段的索引中：

```
sky@localhost : example 08:49:45> create index idx_gid_uid_gc
    -> on group_message(group_id,user_id,gmt_create);
Query OK,  rows affected (0.03 sec)
Records: 96  Duplicates: 0  Warnings: 0
```

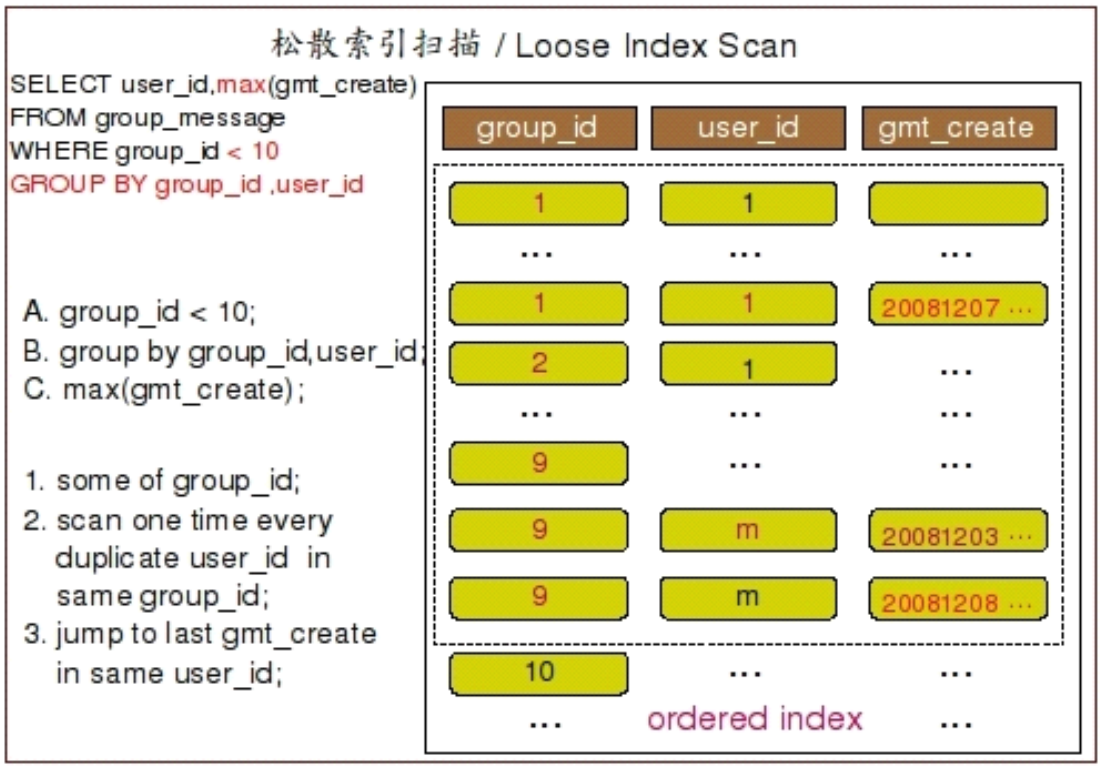
```
sky@localhost : example 09:07:30> drop index idx_group_message_gid_uid
    -> on group_message;
Query OK, 96 rows affected (0.02 sec)
Records: 96  Duplicates: 0  Warnings: 0
```

然后再看如下 Query 的执行计划：

```
sky@localhost : example 09:26:15> EXPLAIN
    -> SELECT user_id,max(gmt_create)
    -> FROM group_message
    -> WHERE group_id < 10
    -> GROUP BY group_id,user_id\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: range
possible_keys: idx_gid_uid_gc
          key: idx_gid_uid_gc
      key_len: 8
          ref: NULL
         rows: 4
        Extra: Using where; Using index for group-by
1 row in set (0.00 sec)
```

我们看到在执行计划的 Extra 信息中有信息显示“Using index for group-by”，实际上这就是告诉我们，MySQL Query Optimizer 通过使用松散索引扫描来实现了我们所需要的 GROUP BY 操作。

下面这张图片描绘了扫描过程的大概实现：



要利用到松散索引扫描实现 GROUP BY，需要至少满足以下几个条件：

- GROUP BY 条件字段必须在同一个索引中最前面的连续位置；
- 在使用 GROUP BY 的同时，只能使用 MAX 和 MIN 这两个聚合函数；
- 如果引用到了该索引中 GROUP BY 条件之外的字段条件的时候，必须以常量形式存在；

为什么松散索引扫描的效率会很高？

因为在没有 WHERE 子句，也就是必须经过全索引扫描的时候， 松散索引扫描需要读取的键值数量与分组的组数量一样多，也就是说比实际存在的键值数目要少很多。而在 WHERE 子句包含范围判断式或者等值表达式的时候， 松散索引扫描查找满足范围条件的每个组的第 1 个关键字，并且再次读取尽可能最少数量的关键字。

2. 使用紧凑（Tight）索引扫描实现 GROUP BY

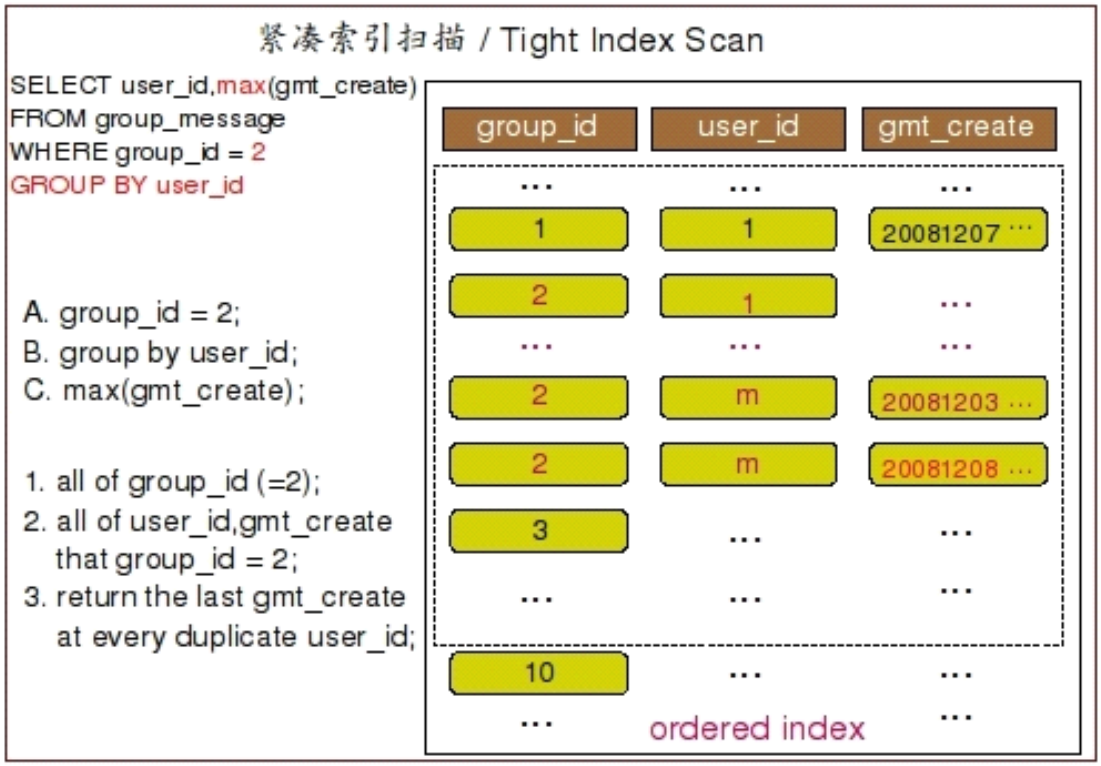
紧凑索引扫描实现 GROUP BY 和松散索引扫描的区别主要在于他需要在扫描索引的时候，读取所有满足条件的索引键，然后再根据读取恶的数据来完成 GROUP BY 操作得到相应结果。

```
sky@localhost : example 08:55:14> EXPLAIN
    -> SELECT max(gmt_create)
    -> FROM group_message
    -> WHERE group_id = 2
    -> GROUP BY user_id\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
```

```
          type: ref
possible_keys: idx_group_message_gid_uid,idx_gid_uid_gc
           key: idx_gid_uid_gc
       key_len: 4
           ref: const
          rows: 4
         Extra: Using where; Using index
1 row in set (0.01 sec)
```

这时候的执行计划的 Extra 信息中已经没有“Using index for group-by”了，但并不是说 MySQL 的 GROUP BY 操作并不是通过索引完成的，只不过是需要访问 WHERE 条件所限定的所有索引键信息之后才能得出结果。这就是通过紧凑索引扫描来实现 GROUP BY 的执行计划输出信息。

下面这张图片展示了大概的整个执行过程：



在 MySQL 中，MySQL Query Optimizer 首先会选择尝试通过松散索引扫描来实现 GROUP BY 操作，当发现某些情况无法满足松散索引扫描实现 GROUP BY 的要求之后，才会尝试通过紧凑索引扫描来实现。

当 GROUP BY 条件字段并不连续或者不是索引前缀部分的时候，MySQL Query Optimizer 无法使用松散索引扫描，设置无法直接通过索引完成 GROUP BY 操作，因为缺失的索引键信息无法得到。但是，如果 Query 语句中存在一个常量值来引用缺失的索引键，则可以使用紧凑索引扫描完成 GROUP BY 操作，因为常量填充了搜索关键字中的“差距”，可以形成完整的索引前缀。这些索引前缀可以用于索引查找。而如果需要排序 GROUP BY 结果，并且能够形成索引前缀的搜索关键字，MySQL 还可以避免额外的排序操作，因为使用有顺序的索引的前缀进行搜索已经按顺序检索到了所有关键字。

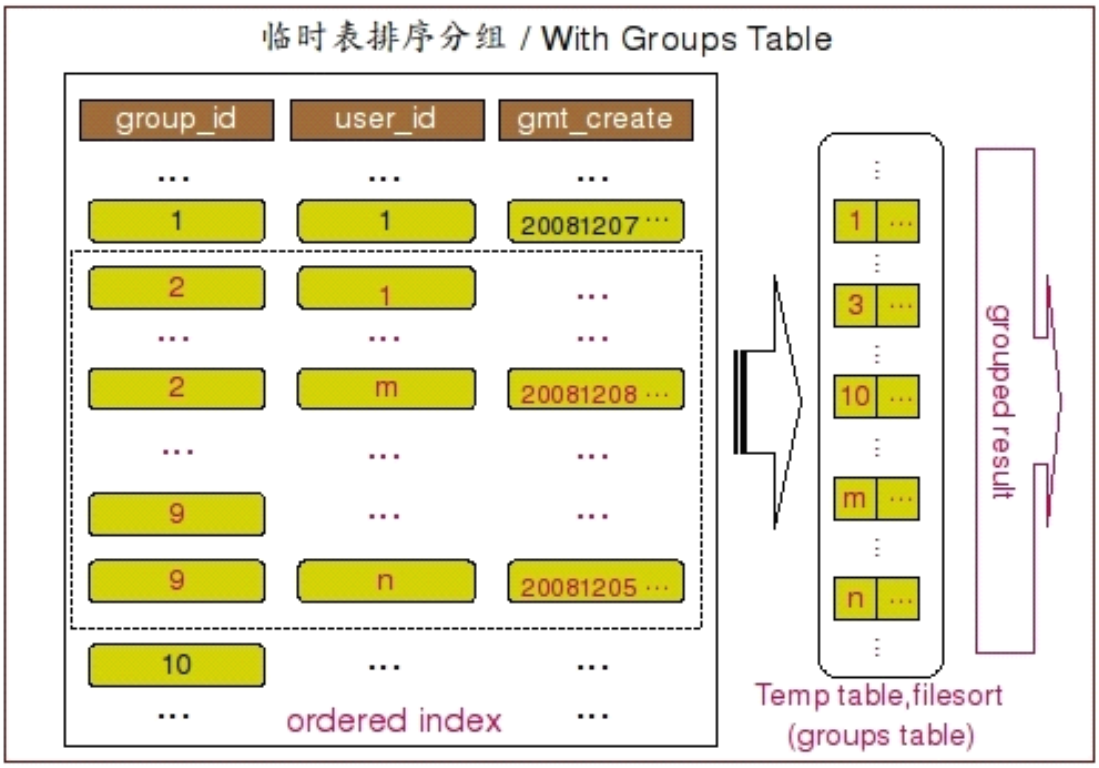
3. 使用临时表实现 GROUP BY

MySQL 在进行 GROUP BY 操作的时候要想利用所有，必须满足 GROUP BY 的字段必须同时存放于同一个索引中，且该索引是一个有序索引（如 Hash 索引就不能满足要求）。而且，并不只是如此，是否能够利用索引来实现 GROUP BY 还与使用的聚合函数也有关系。

前面两种 GROUP BY 的实现方式都是在有可以利用的索引的时候使用的，当 MySQL Query Optimizer 无法找到合适的索引可以利用的时候，就不得不先读取需要的数据，然后通过临时表来完成 GROUP BY 操作。

```
sky@localhost : example 09:02:40> EXPLAIN
    -> SELECT max(gmt_create)
    -> FROM group_message
    -> WHERE group_id > 1 and group_id < 10
    ->  GROUP BY user_id\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: range
possible_keys: idx_group_message_gid_uid,idx_gid_uid_gc
          key: idx_gid_uid_gc
      key_len: 4
          ref: NULL
         rows: 32
        Extra: Using where; Using index; Using temporary; Using filesort
```

这次的执行计划非常明显的告诉我们 MySQL 通过索引找到了我们需要的数据，然后创建了临时表，又进行了排序操作，才得到我们需要的 GROUP BY 结果。整个执行过程大概如下图所展示：



当 MySQL Query Optimizer 发现仅仅通过索引扫描并不能直接得到 GROUP BY 的结果之后，他就不得不选择通过使用临时表然后再排序的方式来实现 GROUP BY 了。

在这样示例中即是这样的情况。 group_id 并不是一个常量条件，而是一个范围，而且 GROUP BY 字段为 user_id。所以 MySQL 无法根据索引的顺序来帮助 GROUP BY 的实现，只能先通过索引范围扫描得到需要的数据，然后将数据存入临时表，然后再进行排序和分组操作来完成 GROUP BY。

对于上面三种 MySQL 处理 GROUP BY 的方式，我们可以针对性的得出如下两种优化思路：

1. 尽可能让 MySQL 可以利用索引来完成 GROUP BY 操作，当然最好是松散索引扫描的方式最佳。在系统允许的情况下，我们可以通过调整索引或者调整 Query 这两种方式来达到目的；
2. 当无法使用索引完成 GROUP BY 的时候，由于要使用到临时表且需要 filesort，所以我们必须要有足够的 sort_buffer_size 来供 MySQL 排序的时候使用，而且尽量不要进行大结果集的 GROUP BY 操作，因为如果超出系统设置的临时表大小的时候会出现将临时表数据 copy 到磁盘上面再进行操作，这时候的排序分组操作性能将是成数量级的下降；

至于如何利用好这两种思路，还需要大家在自己的实际应用场景中不断的尝试并测试效果，最终才能得到较佳的方案。此外，在优化 GROUP BY 的时候还有一个小技巧可以让我们在有些无法利用到索引的情况下避免 filesort 操作，也就是在整个语句最后添加一个以 null 排序（ORDER BY null）的子句，大家可以尝试一下试试看会有什么效果。

DISTINCT 的实现与优化

DISTINCT 实际上和 GROUP BY 的操作非常相似，只不过是在 GROUP BY 之后的每组中只取出一条记

录而已。所以，DISTINCT 的实现和 GROUP BY 的实现也基本差不多，没有太大的区别。同样可以通过松散索引扫描或者是紧凑索引扫描来实现，当然，在无法仅仅使用索引即能完成 DISTINCT 的时候，MySQL 只能通过临时表来完成。但是，和 GROUP BY 有一点差别的是，DISTINCT 并不需要进行排序。也就是说，在仅仅只是 DISTINCT 操作的 Query 如果无法仅仅利用索引完成操作的时候，MySQL 会利用临时表来做一次数据的“缓存”，但是不会对临时表中的数据进行 filesort 操作。当然，如果我们在进行 DISTINCT 的时候还使用了 GROUP BY 并进行了分组，并使用了类似于 MAX 之类的聚合函数操作，就无法避免 filesort 了。

下面我们就通过几个简单的 Query 示例来展示一下 DISTINCT 的实现。

1. 首先看看通过松散索引扫描完成 DISTINCT 的操作：

```
sky@localhost : example 11:03:41> EXPLAIN SELECT DISTINCT group_id
    -> FROM group_message\G
*************************** 1. row ***************************
           id: 1
  SELECT_type: SIMPLE
        table: group_message
         type: range
possible_keys: NULL
          key: idx_gid_uid_gc
      key_len: 4
          ref: NULL
         rows: 10
        Extra: Using index for group-by
1 row in set (0.00 sec)
```

我们可以很清晰的看到，执行计划中的 Extra 信息为“Using index for group-by”，这代表什么意思？为什么我没有进行 GROUP BY 操作的时候，执行计划中会告诉我这里通过索引进行了 GROUP BY 呢？其实这就是于 DISTINCT 的实现原理相关的，在实现 DISTINCT 的过程中，同样也是需要分组的，然后再从每组数据中取出一条返回给客户端。而这里的 Extra 信息就告诉我们，MySQL 利用松散索引扫描就完成了整个操作。当然，如果 MySQL Query Optimizer 要是能够做的再人性化一点将这里的信息换成“Using index for distinct”那就更好更容易让人理解了，呵呵。

2. 我们再来看看通过紧凑索引扫描的示例：

```
sky@localhost : example 11:03:53>  EXPLAIN SELECT DISTINCT user_id
    -> FROM group_message
    -> WHERE group_id = 2\G
*************************** 1. row ***************************
           id: 1
  SELECT_type: SIMPLE
        table: group_message
         type: ref
possible_keys: idx_gid_uid_gc
          key: idx_gid_uid_gc
```

```
      key_len: 4
          ref: const
         rows: 4
        Extra: Using WHERE; Using index
1 row in set (0.00 sec)
```

这里的显示和通过紧凑索引扫描实现 GROUP BY 也完全一样。实际上，这个 Query 的实现过程中，MySQL 会让存储引擎扫描 group_id = 2 的所有索引键，得出所有的 user_id，然后利用索引的已排序特性，每更换一个 user_id 的索引键值的时候保留一条信息，即可在扫描完所有 gruop_id = 2 的索引键的时候完成整个 DISTINCT 操作。

3. 下面我们在看看无法单独使用索引即可完成 DISTINCT 的时候会是怎样：

```
    sky@localhost : example 11:04:40> EXPLAIN SELECT DISTINCT user_id
    -> FROM group_message
    -> WHERE group_id > 1 AND group_id < 10\G
*************************** 1. row ***************************
           id: 1
  SELECT_type: SIMPLE
        table: group_message
         type: range
possible_keys: idx_gid_uid_gc
          key: idx_gid_uid_gc
      key_len: 4
          ref: NULL
         rows: 32
        Extra: Using WHERE; Using index; Using temporary
1 row in set (0.00 sec)
```

当 MySQL 无法仅仅依赖索引即可完成 DISTINCT 操作的时候，就不得不使用临时表来进行相应的操作了。但是我们可以看到，在 MySQL 利用临时表来完成 DISTINCT 的时候，和处理 GROUP BY 有一点区别，就是少了 filesort。实际上，在 MySQL 的分组算法中，并不一定非要排序才能完成分组操作的，这一点在上面的 GROUP BY 优化小技巧中我已经提到过了。实际上这里 MySQL 正是在没有排序的情况下实现分组最后完成 DISTINCT 操作的，所以少了 filesort 这个排序操作。

4. 最后再和 GROUP BY 结合试试看：

```
sky@localhost : example 11:05:06> EXPLAIN SELECT DISTINCT max(user_id)
    -> FROM group_message
    -> WHERE group_id > 1 AND group_id < 10
    -> GROUP BY group_id\G
*************************** 1. row ***************************
           id: 1
  SELECT_type: SIMPLE
```

```
        table: group_message
         type: range
possible_keys: idx_gid_uid_gc
          key: idx_gid_uid_gc
      key_len: 4
          ref: NULL
         rows: 32
        Extra: Using WHERE; Using index; Using temporary; Using filesort
1 row in set (0.00 sec)
```

最后我们再看一下这个和 GROUP BY 一起使用带有聚合函数的示例，和上面第三个示例相比，可以看到已经多了 filesort 排序操作了，因为我们使用了 MAX 函数的缘故。

对于 DISTINCT 的优化，和 GROUP BY 基本上一致的思路，关键在于利用好索引，在无法利用索引的时候，确保尽量不要在大结果集上面进行 DISTINCT 操作，磁盘上面的IO操作和内存中的IO操作性能完全不是一个数量级的差距。

8.7 小结

本章重点介绍了 MySQL Query 语句相关的性能调优的部分思路和方法，也列举了部分的示例，希望能够帮助读者朋友在实际工作中开阔一点点思路。虽然本章涉及到的内容包含了最初的索引设计，到编写高效 Query 语句的一些原则，以及最后对语句的调试，但 Query 语句的调优远不只这些内容。很多的调优技巧，只有到在实际的调优经验中才会真正体会，真正把握其精髓。所以，希望各位读者朋友能多做实验，以理论为基础，以事实为依据，只有这样，才能不断提升自己对 Query 调优的深入认识。

第 9 章 MySQL 数据库 Schema 设计的性能优化

前言：

很多人都认为性能是在通过编写代码（程序代码或者是数据库代码）的过程中优化出来的，其实这是一个非常大的误区。真正影响性能最大的部分是在设计中就已经产生了的，后期的优化很多时候所能够带来的改善都只是在解决前妻设计所遗留下来的一些问题而已，而且能够解决的问题通常也比较有限。本章将就如何在 MySQL 数据库 Schema 设计的时候保证尽可能的高效，尽可能减少后期的烦恼。

9.1 高效的模型设计

最规范的就一定是最合理的吗？

在数据库 Schema 设计理论方面，一直有一个被大家奉为“葵花宝典”的规范化范式理论。通过范式理论所设计的数据库 Schema 逻辑清晰，关系明确，扩展方便，就连存储的数据量也做到了尽可能的少，尤其是当范式级别较高的时候，几乎找不到任何的冗余数据。在很多人眼里，数据库 Schema 满足的范式级别越高则该 Schema 设计的越优秀。

但是，很多人忽略了一点，那就是产生该理论的时期和出发点。关系性数据库的规范化范式理论诞生于上世纪七十年代初，最根本的目的是让数据库中尽量的去除数据的冗余，保持数据的一致，使数据的修改简单。

实际上，尽量去除数据的冗余不仅仅是为了让我们查询相同的数据量的时候能够多返回几条记录，还有一个很重要的原因就是在当时的那个年代，数据的存储空间是及其昂贵的，而且存储设备的容量也都非常的小，这一点在硬件存储设备发展如此迅速的如今，空间大小已经不再是太大的问题了。

而范式理论中的数据一致性和使数据修改简单保证主要是依靠添在数据库中添加各种约束来保证，而各种约束对于数据库来说本身其实就是一个非常消耗资源的事情。

所以，对于基于性能的数据库 Schema 设计，我们并不能完全以规范化范式理论来作为唯一的指导。在设计过程中，应该从实际需求出发，以性能提升为根本目标来展开设计工作，很多时候为了尽可能提高性能，我们必须做反范式设计。

适度冗余 - 让 Query 尽两减少 Join

熟悉 MySQL 的优化器的读者可能清楚，MySQL 的优化器虽然号称使用了新一代的优化器技术实现的非常优秀，但是由于目前 MySQL 所收集的数据统计信息还不是特别的多，所以起表现并不是特别的让人满意，也并非如 MySQL 官方所宣传的那样智能。虽然处理普通 Join 的时候一般都能比较智能的得到比较高效的执行计划，但是当遇到一些自查询或者较为复杂的 Join 的时候，很容易出现不太合理的执行计划，不少时候对各表的访问顺序选择的并不合适，造成复杂 Query 的整体执行效率低下。

所以，为了让我们的 Query 执行计划尽可能的最优化，最直接有效的方式就是尽量减少 Join，而要

减少 Join，我们就不可避免的需要通过表字段的冗余来实现。

这里我们继续通过“影响 MySQL Server 性能的相关因素” 一章中 “Schema 设计对性能的影响”这一节的一个例子来进一步分析资源消耗的差异。方案一中的 group_message 表中仅保存了发布信息者的 ID 信息，而通过冗余优化之后的 group_message 表中增加了发布信息者的 nick_name 信息存为 author。

优化前实现列表功能的 Query 和执行计划（group_message_bad 是优化前的表，优化后为 group_message 表）：

```
sky@localhost : example 09:13:41> explain
    -> SELECT t.id, t.subject,user.id, user.nick_name
    ->     FROM (
    ->         SELECT id, user_id, subject
    ->         FROM group_message
    ->         WHERE group_id = 1
    ->         ORDER BY gmt_modified DESC LIMIT 1,10
    ->     ) t, user
    ->     WHERE t.user_id = user.id\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: <derived2>
         type: system
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: PRIMARY
        table: user
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 1
        Extra:
*************************** 3. row ***************************
           id: 2
  select_type: DERIVED
```

```
        table: group_message
         type: ALL
possible_keys: group_message_gid_ind
          key: group_message_gid_ind
      key_len: 4
          ref:
         rows: 1
        Extra: Using filesort
```

优化后实现列表功能的Query和执行计划：

```
sky@localhost : example 09:14:06> explain
    -> SELECT t.id, t.subject, t.user_id, t.author
    ->     FROM group_message t
    ->     WHERE group_id = 1
    ->     ORDER BY gmt_modified DESC LIMIT 1,10\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t
         type: ref
possible_keys: group_message_gid_ind
          key: group_message_gid_ind
      key_len: 4
          ref: const
         rows: 1
        Extra: Using where; Using filesort
```

从优化前和优化后的执行计划可以看出两者的差别非常大的，优化前必须检索2个表（group_message和user）才能得到结果，而优化后只需要检索group_message一个表就可以完成，因为我们将“作者”信息冗余到了group_message。

从数据库范式理论来看，这样的设计是不合理的。因为可能造成user表和group_message表中的用户昵称数据不一致。每次更新用户昵称的时候，都需要更新两个表的数据，为了尽可能让两者数据保证一致，应用程序中需要处理更多的逻辑。但是，从性能角度来看的话，这种冗余是非常有价值的，虽然我们的数据更新逻辑复杂了，但是我们在考虑更新带来的附加成本的时候，还应该考虑我们到底会有多少更新发生在用户昵称上面呢？我们需要考虑的是一个系统的整体性能，而不是系统中单个行为的性能。就像示例中的昵称数据，虽然更新的成本增加了，但是查询的效率提高了，而且发生示例中查询的频率要远大于更新的频率，通过少部分操作的成本投入换取更大的性能收获，实际上是我们系统性能优化中经常使用的策略。

在大部分应用系统中，类似于上面示例中的这种查询频繁但是更新较少的数据非常非常多，很多时候如果我们一味的追求范式化理论的Schema设计在高性能要求的系统中是非常不合适的。我个人认为，数据库的规范化理论其实质是在概念上的单一化，虽然规范后的数据库中的表一般都较小，使表中相关列最少。这虽然可能在某些情况下增强了数据库的可维护性，但在系统要完成一些数据的查询检索时，

可能要用复杂的 Join 才能实现，这势必会造成查询检索的性能低下。如果我们通过拆分 Join，通过多次简单的查询来在应用中实现 Join 逻辑，那所带来的网络开销将会是非常巨大的。

大字段垂直分拆 - summary 表优化

实际上，在上面的示例中我们同时还用到了另外一种优化策略，也就是“大字段垂直拆分”策略。大字段垂直拆分策略相对于前面介绍的适度冗余策略在做法上可以说产不多是完全相反的做法。适度冗余策略是将别的表中的字段拿过来在自己身上也存一份数据，而大字段垂直拆分简单来说就是将自己身上的字段拆分出去放在另外（单独）的表里面。

可能很多读者朋友都会有疑惑了，我们刚刚才分析出了将别的字段拿过来放自己表里面为什么现在又要将自己的字段分出去呢？这样不是有些自相矛盾了吗？

其实并没有任何矛盾，前面我们将别人的字段那过来，是因为我们很多时候的查询需要使用该字段，为了减少 Join 带来的性能消耗才拿过来的。而我们将大字段拿出去，也是将一些我们在大部分查询中并不需要使用该字段的时候才会拿出去。而且，在我们拿出去之前，我们肯定会通过全面的评估比较之后才能做出拆分出去的决定。

那到底什么样的字段适合于从表中拆分出去呢？

首要肯定是大字段。为什么？原因很简单，就是因为他的大。大字段一般都是存放着一些较长的 Detail 信息，如文章的内容，帖子的内容，产品的介绍等等。

其次是和表中其他字段相比访问频率明显要少很多。由于大字段存放的内容较多，大部分情况都是占整条记录的 80%以上，而数据库中数据在数据文件中的格式一般都是以一条一条记录为单位来存放。也就是说，如果我们要查询某些记录的某几个字段，数据库并不是只需要访问我们需要查询的哪几个字段，而是需要读取其他所有字段（可以在索引中完成整个查询的情况除外），也无法做到只读取我们需要的几个字段的数据。这样，我们就不得不读取包括大字段在内的很多并不相干的数据。而由于大字段所占的空间比例非常大，自然所浪费的 IO 资源也就非常之大了。

在这样的场景下，我们就需要将该大字段从原表中拆分出来，通过单独的表进行存放，让我们在访问其他数据的时候大大降低 IO 访问，从而使性能得到较大的改善。

可能有人会疑惑，虽然移出之后访问其他字段的效率提高了，但是当我们需要大字段的信息的时候，我们就无法避免的需要通过 Join 来实现，而使用 Join 之后的处理效率可能会大打折扣的。其实这个担心是很合理的，这也就是我们在分拆出大字段之前需要还需要考虑的第二个因素，访问频率的因素了。前面我们就介绍了，决定是否要分拆出，出了“大”之外，还要“频率低”才行，当然，这里的“频率低”只是“相对频率”而已。而且，这种分拆之后的两个表的关系都是完全确定的一一对应关系，使用 Join 在性能方面的影响也并不是特别的大。

那我们在移出大字段的同时，是否还需要将其他字段也一并移出呢？其实如果我们已经确定有大字段需要分拆出主表的时候，对于其他的字段，只要满足访问频率和大字段一样相对于表中其他字段要低很多的都可以和大字段同时分拆出来。

实际上，在有些时候，我们甚至都不一定非要大字段才能进行垂直分拆。在有些场景下，有的表中大部分字段平时都很少访问，而其中的某几个字段却是访问频率非常高。对于这种表，也非常适合通过垂直分拆来达到优化性能的目的。

在“Schema 设计对性能的影响”一节中的示例中，实际上是有两处用到了“垂直分拆”这个优化策略。一处是 group_message_bad 表中的 content 大字段从原表中分拆出来为 group_message_content 表。另一处就是将原 user_bad 表中虽然不大但是平时使用很少的字段拆分出来新增了 user_profile 表。

大表水平分拆 - 基于类型的分拆优化

“大表水平拆分”策略在性能优化方面可能被人使用的频率并不是太多，但是如果使用得当，很可能会给我们带来不小的惊喜。

我们还是直接通过实例来说明问题吧。假设我们将前面示例中的需求稍微做一下扩展，我们希望 group 系统总管理员能够发布系统消息，而且在每一个 group 的讨论帖的没一页都能置顶显示。

在得到该需求之后，我们的第一反应肯定是通过在 group_message 表中增加一个标识列，用来存放帖子的类型，标识出是普通会员的讨论贴还是系统管理员的置顶帖。然后在每个列表展示页面都通过对 group_message 表的两次查询（一次置顶信息，一次普通讨论帖）然后在应用程序中合并再展示。这样的结果是由于整个 group_message 表的数据较大，查询置顶信息的 Query 成本会相对有些高。

下面我们换一个思路来考虑一下这个问题：
首先，置顶信息和其他讨论帖完全不会产生任何关联交互；
其次，置顶信息的变化相对于其他讨论帖来说变化很少；
再次，置顶信息的访问频率非常高；
最后，置顶信息的量和普通讨论帖来比非常之少；

通过上面的这几个分析，如果我们将置顶信息单独存放在普通讨论帖之外的其他表里面，首先不会带来什么附加的性能消耗，而且可以使每次检索置顶信息的成本都有所下降。由于访问频率非常的高，则因为每次检索置顶信息的成本下降而得到较大的节省。数量少而且变化不怎么频繁的特点则非常适合使用 MySQL 的 Query Cache，而如果和普通讨论帖在一起由于普通讨论帖的频繁变化带来 group_message 表相关的 Query Cache 失效问题会让他无法使用 Query Cache 功能。

通过上面的分析，我们很容易得出一个更为优化的方案来存放这些置顶信息，那就是新增一张类似于 group_message 的表来专门存放置顶信息，我们暂且命名为 top_message 如下：

```
sky@localhost : example 10:49:20> desc top_message;
+-------------+-------------+------+-----+---------+-------+
| Field       | Type        | Null | Key | Default | Extra |
+-------------+-------------+------+-----+---------+-------+
| id          | int(11)     | NO   |     | 0       |       |
| gmt_create  | datetime    | NO   |     | NULL    |       |
| gmt_modified| datetime    | NO   |     | NULL    |       |
| user_id     | int(11)     | NO   |     | NULL    |       |
| author      | varchar(32) | NO   |     | NULL    |       |
```

```
| subject       | varchar(128) | NO   |     | NULL    |       |
+-------------+-------------+------+-----+---------+-------+
```

由于是全局的，所以省略了 group_id 信息，而 content 信息，还是同样可以存放在 group_message_content 表中。

上面仅仅只是一个示例，可能在实际应用中并不是如此的简单，但这里只是给大家一个思路，让大家知道如何通过大表的水平拆分来对通过优化 Schema 设计提供系统的整体性能。在很多大型的应用中，由于数据量非常庞大，并发访问又非常高，到达单台主机都无法支撑单个表的访问的时候，常常会通过这种大表的水平拆分，存放在多台主机的多个数据库中实现整体扩展性的提升，这方面的内容我们将在“架构设计”部分的“可扩展设计之数据切分”章节再做更为详细的介绍。

统计表 - 准实时优化

统计表的准实时优化策略实际上我们在“影响 MySQL Server 性能的相关因素”一章的“商业需求对性能的影响”部分有提出过。简单来说就是通过定时统计数据来替代实时统计查询。

为什么要准实时？
很多人看到这个优化策略之后可能都会提出这样的质疑，为什么要改变需求将“可以实时”的统计信息做成准实时的呢？原因很简单，因为实时统计的性能消耗成本太高。因为每一次展示（也就是每一次刷新页面）都需要进行统计计算，带来大量的重复资源浪费。而做成准实时的统计信息之后，我们每次只需要访问很小的数据量即可，不需要频繁的统计计算的工作。

当然，并不是所有的统计数据都适合于通过准实时的统计表优化策略来实现的，即使我们希望，产品经理们也不会允许，即使产品经理们也希望那样，我们的使用者肯定也会不同意。

什么类型的统计信息适合通过准实时统计表来优化实现？
首先，统计信息的准确性要求并不是特别的严格；
其次，统计信息对时间并不是太敏感；
再次，统计信息的访问非常频繁，重复执行较多；
最后，参与统计数据量较大；

看看上面的要求，还真不少。不过，大家所维护的系统中确实很可能存在这样的统计数据展示功能。如系统当前在线人数，论坛系统当前总帖数、回帖数等，多条件大结果集查询页面的总结果数以及总页数，某些虚拟积分的 top n 排名等等。

这些统计的计算都会设计到大量的数据，同时也需要大量的计算资源，访问频率也都非常的高。如果都通过实时统计，恐怕只要数据量稍微大一些，都会带来非常大的硬件资源开销。但在短时间内的不够精确，又并不会带来太大用户体验的降低。所以完全可以通过定时任务程序，没隔一定时间段进行一次统计后存放在专门设计的统计表中。这样，在统计数据需要展示的时候，我们只需要从统计好的结果数据中取出即可。这样每次统计数据的展示性能将会成数量级的提升，反而会使整体的用户体验上升。

9.2 合适的数据类型

实际上在很多数据库的设计优化文档中都有关于通过优化数据类型的优化说明内容，在 MySQL 中，我们同样也可以通过数据类型的优化达到优化整个 Schema 设计的目的。

优化数据类型提高性能的主要原理在于以下几个方面：

1. 通过选用更“小”的数据类型减少存储空间，使查询相同数据需要的 IO 资源降低；
2. 通过合适的数据类型加速数据的比较；

下面我们还是通过分析一些常用数据类型的数据存储格式和长度来看看哪些数据类型可以在优化中利用上吧。

数字日期类型

我们先来看看存放长度基本固定的一些数据类型的存储长度和取值范围。

类型（同义词）	存储长度	最小值（无符号）	最大值（无符号）
整型数字			
TINYINT	1	－128（0）	127（255）
SMALLINT	2	-32768（0）	32767（65535）
MEDIUMINT	3	-8388608（0）	8388607（16777215）
INT（INTEGER）	4	-2147483648（0）	2147483647（4294967295）
BIGINT	8	-9223372036854775808（0）	9223372036854775807（18446744073709551615）
小数支持			
FLOAT[(M[,D])]	4 or 8	-3.402823466E+38-1.175494351E-38 0 1.175494351E-38～3.402823466E+38	
DOUBLE[(M[,D])]（REAL, DOUBLE PRECISION）	8	-1.7976931348623157E+308～-2.2250738585072014E-308; 0 2.2250738585072014E-308～ 1.7976931348623157E+308	
时间类型			
DATETIME	8	1001-01-01 00:00:00	9999-12-31 23:59:59
DATE	3	1001-01-01	9999-12-31
TIME	3	00:00:00	23:59:59
YEAR	1	1001	9999
TIMESTAMP	4	1970-01-01 00:00:00	

对于数字类型，这里分别列出了整数类型和小数类型，也就是浮点数类型。实际上，还有一类通过二进制格式以字符串来存放的数字类型如 DECIMAL(DEC)[(M[,D])]，NUMERIC[(M[,D])]，由于其存放长度主要通过其定义时候的的 M 所决定，M 定义为多大，则实际存放就有多长。M 代表整个位数长度，而 D 则表示小数点后的位数，默认 M 为 10，D 为 0。一般来说，主要用在固定精度的场合，由于其存放长度较大，而且考虑到这种数据完全可以变化形式以整数存放，所以笔者个人并不是特别推荐。

对于数字的存储，一般使用到浮点型数据的场合也不应该太多。主要出于两个原因，一个是浮点型数据本身实际上是一个并不精确的数字，只是一个近似值，另一个原因就是完全可以通过乘以一个固定的系数转换为整型数据来存放。这样不仅可以解决数据不精确的问题，同时也让数据的处理更为高效。

时间存储格式总类并不是太多，我们常用的主要就是 DATETIME，DATE 和 TIMESTAMP 这三种了。从存储空间来看 TIMESTAMP 最少，四个字节，而其他两种数据类型都是八个字节，多了一倍。而 TIMESTAMP 的缺点在于他只能存储从 1970 年之后的时间，而另外两种时间类型可以存放最早从 1001 年开始的时间。如果有需要存放早于 1970 年之前的时间的需求，我们必须放弃 TIMESTAMP 类型，但是只要我们不需要使用 1970 年之前的时间，最好尽量使用 TIMESTAMP 来减少存储空间的占用。

上面所列出的主要是一些存放固定长度，且我们平时可能常用到的一些类型。通过这个对照表格，我们可以很直观的看出哪种类型占用的存储空间大，哪种占用的空间小。这样，在数据类型选择的时候，我们就可以结合各种类型的存储范围以及业务中可能存在的数据作出对应，然后选择存储空间最先的类型来使用。

字符存储类型

我们再来看看存放字符的数据类型。

类型	存储占用最大空间
CHAR[(M)]	255 characters(independent of charset)
VARCHAR[(M)]	65535 bytes or 255 characters
TINYTEXT[(M)]	255 characters（sigle-byte）
TEXT[(M)]	65535 characters（sigle-byte）
MEDIUMTEXT[(M)]	16777215 characters（sigle-byte）
LONGTEXT[(M)]	4294967295 characters（sigle-byte）

CHAR[(M)]类型属于静态长度类型，存放长度完全以字符数来计算，所以最终的存储长度是基于字符集的，如 latin1 则最大存储长度为 255 字节，但是如果使用 gbk 则最大存储长度为 510 字节。CHAR 类型的存储特点是不管我们实际存放多长数据，在数据库中都会存放 M 个字符，不够的通过空格补上，M 默认为 1。虽然 CHAR 会通过空格补齐存放的空间，但是在访问数据的时候，MySQL 会忽略最后的所有空格，所以如果我们的实际数据中如果在最后确实需要空格，则不能使用 CHAR 类型来存放。在 MySQL5.0.3 之前的版本中，如果我们定义 CHAR 的时候 M 值超过 255，MySQL 会自动将 CHAR 类型进行转换为可以存入对应数据量的 TEXT 类型，如 CHAR(1000)会自动转换为 TEXT，CHAR(10000)则会转为 MEDIUMTEXT。而从 MySQL5.0.3 开始，所有超过 255 的定义 MySQL 都会直接拒绝并给出错误信息，不再自动转换。

VARCHAR[(M)]属于动态存储长度类型，仅存占用实际存储数据的长度。其存放的最大长度与 MySQL 版本有关，在 5.0.3 之前的版本 VARCHAR 以字符数控制最存储的最大长度，最大只能存放 255 个字符，占用存储空间的实际大小与字符集有关。但是从 5.0.3 开始，VARCHAR 的最大存储限制已经更改为字节数限制了，扩展到可以存放 65535 bytes 的数据，不同的字符集可能存放的字符数并不一样。也就是说，在 MySQL5.0.3 之前的版本，M 所代表的是字符数，而从 5.0.3 版本开始，M 的代表意思已经是字节数了。VARCHAR 的存储特点是不管我们设定 M 为多大的值，真正占用的存储空间都只有我们所存入的实际数据的大小，和 CHAR 不同的是 VARCHAR 会保留我们存入数据最后的空格，也就是说我们存入是什么样，MySQL 返回给我们的也会是什么样。在 VARCHAR 类型字段的数据中，MySQL 会在每个 VARCHAR 数据中使用 1 个或者 2 个字节用来存放 VARCHAR 数据的实际长度，当我们的实际数据在 255 字节之内的时候，会使用 1 字节来存放实际长度，而大于 255 字节的时候，则需要使用 2 字节来存放。

TINYTEXT，TEXT，MEDIUMTEXT 和 LONGTEXT 这四种类型同属于一种存储方式，都是动态存储长度类

型，不同的仅仅是最大长度的限制。四种类型的定义都是通过最大字符数来限制，但是他们的字符数限制实际上是可以理解为字节数限制的，因为当我们使用多字节字符集的时候，实际能存放的字符书并没最大字符数那么多，而是以单字节字符来计算的字符数。此外，由于是动态存储长度类型，所以和VARCHAR一样，每个字段数据之前都需要一个存放实际长度的空间。TINYTEXT需要1个字节来存放，TEXT需要2个字节，MEDIUMTEXT和LONGTEXT则分别需要3个和4个字节来存放实际数据长度。实际上，出了MySQL内嵌的最大长度限制之外，他们还受到客户端与服务器端的网络通信缓冲区最大值（max_allowed_packet）的限制。

这四种TEXT类型和CHAR及VARCHAR在实际使用中存在几个不一样的地方：

- 不能设置默认值；
- 只有TEXT可以使用TEXT[(M)]这样的方式通过M设置大小；
- 基于这四种类型的索引必须指定前缀长度；

其他常用类型

除了上面这些字段类型之外会被我们经常使用到之外，我们还会使用到的数据类型主要有以下这些。

类型	存储占用最大空间
BIT[(M)]	(M+7)/8 bytes ,最大(64+7)/8
SET('v1','v2'...)	1,2,4 or 8 bytes（取决于存储值的数目，最大64个值）
ENUM('v1','v2'...)	1 or 2 bytes （取决于存储值的数目，最大65535个值）

对于BIT类型，M表示每个值的bits数目，默认为1，最大为64 bits。对于MySQL来说这是一个新的类型，因为从MySQL5.0.3才开始真正实现（在之前实际上是TINYINT（1）），而且仅仅支持MyISAM存储引擎，但是从MySQL5.0.5开始Memory，Innodb和NDB Cluster存储引擎也开始“支持”了。在MyISAM中，BIT的存储空间很小，是真正的实现了通过bit来存储，但是在其他的一些存储引擎中就不一样了，因为他们是转换为最小的INT类型存储的，所以占用的空间也没有节省，还不如直接使用INT类的数据类型存放来得直观。

对于SET和ENUM类型，主要内容基本处于较少变化状态且值比较少的字段。虽然这两个字段所占用的存储空间都较少，但是由于在使用方面较其他的数据类型要略为复杂一些，所以在实际环境中一般使用还是较少。

谁都知道，数据量（这里主要指数据记录条数）的增加肯定会让数据库的检索查询效率降低。所以很多时候人们大都希望通过减少数据库中关键表的记录条数来获得数据库性能的提升。实际上，除了这种通过控制数据记录条数来控制数据总量的办法之外，我们还可以通过选择更小的数据类型来让数据库通过更小的空间存放相同的数据量，这对于检索同样的数据所带来的IO消耗自然会降低，性能也就很自然得到了提升。

此外，由于CPU对不同数据的处理方式不一样，就会造成不同类型的数据在各种运算处理如比较，排序等方面的处理效率存在差异。所以，对于我们需要经常进行比较计算以及排序等消耗CPU资源的字段，应该尽量选择处理更为迅速的字段类型。如通过整数类型代替浮点数或者字符类型。

9.3 规范的对象命名

规范的命名本身并不会对性能有任何影响，在这里单独列出一节来讲，主要是因为这是一个不太被人重视，但是对后期的数据库维护影响非常大的内容。就像编程语言各自的一些不成文编码基本规范一样，虽然在最初使用的时候并看不错太多的利益，反而会被人认为是一种束缚，但是当每一个人在接手维护一段编写很不规范的代码的时候，我估计大部分人都会非常郁闷，甚至在心里暗骂当初的编写者。其实任何系统都一样，没有任何规范可循，完全一个天马行空的作风，只会给后人（甚至可能是自己）留下一个让人摸不着头脑的烂摊子，难以维护。

对于数据库对象的命名规范其实可以很简单，而且业界也并不存在一个严格的统一规定，只需要在一个公司内足够统一基本就可以了。

一般来说，我个人建议需要注意以下一些方面：

1、 数据库和表名应尽可能和所服务的业务模块名一致；

这样，在 DBA 维护相关数据库对象的时候，新开发人员程序开发过程中，相关技术（或非技术）人员整理业务逻辑和数据关系的时候，都能够非常容易理解其中的关系。

2、 服务于同一子模块的一类表尽量以子模块名（或部分单词）为前缀或后缀；

对同类功能的表增加前缀或者后缀，也是让查看使用该表的各类人员能够很快的根据相关对象的名称就联想到相应的功能，以及相关业务。不论是从维护角度，还是从使用角度来看都会带来非常大的便利性。

3、 表名应尽量包含与所存放数据相对应的单词；

这对于新员工来说尤其重要，要想尽快的熟悉数据，尽快了解相关业务，快速的定位数据库中各表对应的数据意义是非常有帮助的。

4、 字段名称也尽量保持和实际数据相对应

这一点的意义我想各位读者朋友应该都非常的清楚，每个表都会有很多的字段对应数据的各种不同属性，要搞清楚各自代表的含义，除了完整规范的说明文档之外，命名清晰合理的字段名也是一个有用的补充，而且更为直接。

5、 索引名称尽量包含所有的索引键字段名或者缩写，且各字段名在索引名中的顺序应与索引键在索引中的索引顺序一致，且尽量包含一个类似于 idx 或者 ind 之类的前缀或者后缀，以表名其对象类型是索引，同时还可以包含该索引所属表的名称；

这样做最大的好处在于 DBA 在维护过程中能够非常直接清晰的通过索引名称就了解到该索引大部分的信息。

6、 约束等其他对象也应该尽可能包含所属表或其他对象的名称，以表名各自关系。

上面列出的只是一个比较初略的规范建议，各位读者朋友完全可以根据各自公司的习惯，制定自己的命名规范，只要适用，就可以了。规范不在多，而在实用。而且一旦制定的规范，就必须严格的按照规范执行，否则就变成了个花架子没有任何实际的意义了。

9.4 小结

通过这一章的内容，希望能够让大家明白一个道理，“数据库系统的性能不是优化出来的，更多的是设计出来的”。数据库 Schema 的设计并不如很多人想象的那样只是一个简单的对象对应实现，而是一个系统工程。要想设计出一个既性能高效又足够满足业务需求，既逻辑清晰又关系简单的数据库 Schema 结构，不仅仅需要足够的数据库系统知识，还需要足够了解应用系统的业务逻辑。

第 10 章 MySQL Server 性能优化

前言：

本章主要通过针对 MySQL Server（mysqld）相关实现机制的分析，得到一些相应的优化建议。主要涉及 MySQL 的安装以及相关参数设置的优化，但不包括 mysqld 之外的比如存储引擎相关的参数优化，存储引擎的相关参数设置建议将主要在下一章“常用存储引擎的优化”中进行说明。

10.1 MySQL 安装优化

选择合适的发行版本

1. 二进制发行版（包括 RPM 等包装好的特定二进制版本）

由于 MySQL 开源的特性，不仅仅 MySQL AB 提供了多个平台上面的多种二进制发行版本可以供大家选择，还有不少第三方公司（或者个人）也给我们提供了不少选择。

使用 MySQL AB 提供的二进制发行版本我们可以得到哪些好处？

a) 通过非常简单的安装方式快速完成 MySQL 的部署；
b) 安装版本是经过比较完善的功能和性能测试的编译版本；
c) 所使用的编译参数更具通用性的，且比较稳定；
d) 如果购买了 MySQL 的服务，将能最大程度的得到 MySQL 的技术支持；

第三方提供的 MySQL 发行版本大多是在 MySQL AB 官方提供的源代码方面做了或多或少的针对性改动，然后再编译而成。这些改动有些是在某些功能上面的改进，也有些是在某写操作的性能方面的改进。还有些由各 OS 厂商所提供的发行版本，则可能是在有些代码方面针对自己的 OS 做了一些相应的底层调用的调整，以使 MySQL 与自己的 OS 能够更完美的结合。当然，也有一些第三方发行版本并没有动过 MySQL 一行代码，仅仅只是在编译参数方面做了一些相关的调整，而让 MySQL 在某些特定场景下表现更优秀。

这样一说，听起来好像第三方发行的 MySQL 二进制版本要比 MySQL AB 官方提供的二进制发行版有更大的吸引力，那么我们是否就应该选用第三方提供的二进制发行版呢？先别着急，我们还需要进一步分析一下第三方发行版本可能存在哪些问题。

首先，由于第三方发行版本对 MySQL 所做的改动，很多都是为了应对发行者自己所处的特定场景而做出来的。所以，第三方发行版本并不一定适合其他所有使用者所处的环境。

其次，由于第三方发行版本的发行者并一定都是一个足够让人信任的公司（或者个人），在其生成自己的发行版本之前，是否有做过足够全面的功能和性能测试我们不得而知，在我们使用的时候是否会出现 MySQL AB 官方的发行版本中并不存在的 bug？

最后，如果我们购买了 MySQL 的相关服务，而又使用了第三方的发行版本，当我们的系统出现问题的时候，恐怕 MySQL 的支持工程师的支持工作会大打折扣，甚至可能会拒绝提供支持。

如果大家可以完全抛开以上这些可能存在隐患的顾虑，完全可以尝试使用非 MySQL AB 官方提供的二进制版本，而选用可能具有更多特性或者更高性能的发行版本了。

之前我也对网络上各种第三方二进制分发版本做过一些测试和比较，也发现了一些比较不错的版本，如 Percona 在整合了一些比较优秀的 Patch 之后的发行版本整体质量都还不错，使用者也比较多。当然，Percona 不仅仅分发二进制版本，同时也分发整合了一些优秀 Patch 的源码包。对于希望使 Percona 提供的一些 Patch 的朋友，同时又希望能够自行编译以进一步优化和定制 MySQL 的朋友，也可以下载 Percona 提供的源码包。

对于二进制分发版本的安装，对于安装本身来说，我们基本上没有太多可以优化的地方，唯一可以做的就是当我们决定了选择第三方分发版本之后，可以根据自身环境和应用特点来选择适合我们环境的优化发行版本来安装。

2. 源码安装

与二进制发行版本相比，如果我们选择了通过源代码进行安装，那么在安装过程中我们能够对MySQL所做的调整将会更多更灵活一些。因为通过源代码编译我们可以：

a) 针对自己的硬件平台选用合适的编译器来优化编译后的二进制代码；
b) 根据不同的软件平台环境调整相关的编译参数；
c) 针对我们特定应用场景选择需要什么组件不需要什么组件；
d) 根据我们的所需要存储的数据内容选择只安装我们需要的字符集；
e) 同一台主机上面可以安装多个MySQL；
f) 等等其他一些可以根据特定应用场景所作的各种调整。

在源码安装给我们带来更大灵活性的同时，同样也给我们带来了可能引入的隐患：

a) 对编译参数的不够了解造成编译参数使用不当可能使编译出来的二进制代码不够稳定；
b) 对自己的应用环境把握失误而使用的优化参数可能反而使系统性能更差；
c) 还有一个并不能称之为隐患的小问题就是源码编译安装将使安装部署过程更为复杂，所花费的时间更长；

通过源码安装的最大特点就是可以让我们自行调整编译参数，最大程度的定制安装结果。下面我将自己在通过源码编译安装中的一些优化心得做一个简单的介绍，希望能够对大家有所帮助。

在通过源码安装的时候，最关键的一步就是配置编译参数，也就是执行通过 configure 命令所设定的各种编译选项。我们可以在 MySQL 源码所在的文件夹下面通过执行执行“./configure —help”得到可以设置的所有编译参数选项，如下：

```
`configure' configures this package to adapt to many kinds of systems.

Usage: ./configure [OPTION]... [VAR=VALUE]...
    ... ...
Installation directories:
  --prefix=PREFIX         install architecture-independent files in PREFIX
    ... ...
For better control, use the options below.

Fine tuning of the installation directories:
  --bindir=DIR           user executables [EPREFIX/bin]
    ... ...
Program names:
  --program-prefix=PREFIX       prepend PREFIX to installed program names
      ... ...
System types:
  --build=BUILD     configure for building on BUILD [guessed]
      ... ...
Optional Features:
  --disable-FEATURE  do not include FEATURE (same as --enable-FEATURE=no)
      ... ...
Optional Packages:
```

```
  --with-charset=CHARSET
      ... ...
  --without-innodb        Do not include the InnoDB table handler
      ... ...
Some influential environment variables:
  CC          C compiler command
      ... ...
  CCASFLAGS   assembler compiler flags (defaults to CFLAGS)
      ... ...
```

上面的输出内容中很多都已经省略了，大家完全可以通过自行测试得到更为丰富的内容输出。下面针对几个比较重要的编译参数做一个简单的介绍：

- “—prefix”：设定安装路径，默认为“/usr/local”；
- “—datadir”：设定 MySQL 数据文件存放路径；
- “—with-charset”：设定系统的默认字符集；
- “—with-collation”：系统默认的校验规则；
- “—with-extra-charsets”：出了默认字符集之外需要编译安装的字符集；
- “—with-unix-socket-path”：设定 socket 文件地址；
- “—with-tcp-port”：指定特定监听端口，默认为 3306；
- “—with-mysqld-user”：指定运行 mysqld 的 os 用户，默认为 mysql；
- “—without-query-cache”：禁用 Query Cache 功能；
- “—without-innodb”：禁用 Innodb 存储引擎；
- “--with-partition”：在 5.1 版本中开启 partition 支持特性；
- “--enable-thread-safe-client”：以线程方式编译客户端；
- “—with-pthread”：强制使用 pthread 线程库编译；
- “—with-named-thread-libs”：指定使用某个特定的线程库编译；
- “—without-debug”：使用非 debug 模式；
- “—with-mysqld-ldflags”：mysqld 的额外 link 参数；
- “—with-client-ldflags”：client 的额外 link 参数；
-

以上这些参数是在源码安装中比较常用的一些编译参数，其中前面几个编译参数主要是为了方便我们在安装的时候可以定制自己的系统，让系统更适合我们自己应用环境的相关规范，做到环境统一，并按照实际需求生成相应的二进制代码。而后面的一些参数主要是用来优化编译结果的。

我想大家应该都能理解一般来说，一个系统功能越复杂，其性能一般都会越差。所以，在我们安装编译 MySQL 的时候应该尽量只选用我们需要的组件，仅安装我们需要的存储引擎，仅编译我们需要的字符集，让我们的系统能够尽可能的简单，因为这样的 MySQL 也会给我们带来尽可能高的性能。

此外，对于一些特定的软件环境上，可能会有多种线程库的选择的，如果你对各个线程库较为了解，完全可以通过编译参数设定让 MySQL 使用最合适的线程库，让 MySQL 在我们特定的环境中发挥他最优化的一面。

源码包的编译参数中默认会以Debug模式生成二进制代码，而Debug模式给MySQL带来的性能损失是比较大的，所以当我们编译准备安装的产品代码的时候，一定不要忘记使用“—without-debug”参数禁用Debug模式。

而“—with-mysqld-ldflags”和“—with-client-ldflags”两个编译参数如果设置为“-all-static”的话，可以告诉编译器以静态方式编译来使编译结果代码得到最高的性能。使用静态编译和动态方式编译的代码相比，性能差距可能会达到5%到10%之多。

就我个人来说最常使用的编译配置参数如下，各位可以参照自行增删相关内容：

```
./configure --prefix=/usr/local/mysql \
--without-debug \
--without-bench \
--enable-thread-safe-client \
--enable-assembler \
--enable-profiling \
--with-mysqld-ldflags=-all-static \
--with-client-ldflags=-all-static \
--with-charset=latin1 \
--with-extra-charset=utf8,gbk \
--with-innodb \
--with-csv-storage-engine \
--with-federated-storage-engine \
--with-mysqld-user=mysql \
--without-embedded-server \
--with-server-suffix=-community \
--with-unix-socket-path=/usr/local/mysql/sock/mysql.sock
```

10.2 MySQL 日志设置优化

在安装完MySQL之后，肯定是需要对MySQL的各种参数选项进行一些优化调整的。虽然MySQL系统的伸缩性很强，既可以在有很充足的硬件资源环境下高效的运行，也可以在极少资源环境下很好的运行，但不管怎样，尽可能充足的硬件资源对MySQL的性能提升总是有帮助的。在这一节我们主要分析一下MySQL的日志（主要是Binlog）对系统性能的影响，并根据日志的相关特性得出相应的优化思路。

日志产生的性能影响

由于日志的记录带来的直接性能损耗就是数据库系统中最为昂贵的IO资源，所以对于日志的

在之前介绍MySQL物理架构的章节中，我们已经了解到了MySQL的日志包括错误日志（Error Log），更新日志（Update Log），二进制日志（Binlog），查询日志（Query Log），慢查询日志（Slow Query Log）等。当然，更新日志是老版本的MySQL才有的，目前已经被二进制日志替代。

在默认情况下，系统仅仅打开错误日志，关闭了其他所有日志，以达到尽可能减少IO损耗提高系统

性能的目的。但是在一般稍微重要一点的实际应用场景中，都至少需要打开二进制日志，因为这是MySQL很多存储引擎进行增量备份的基础，也是 MySQL 实现复制的基本条件。有时候为了进一步的性能优化，定位执行较慢的 SQL 语句，很多系统也会打开慢查询日志来记录执行时间超过特定数值（由我们自行设置）的SQL 语句。

一般情况下，在生产系统中很少有系统会打开查询日志。因为查询日志打开之后会将 MySQL 中执行的每一条 Query 都记录到日志中，会该系统带来比较大的 IO 负担，而带来的实际效益却并不是非常大。一般只有在开发测试环境中，为了定位某些功能具体使用了哪些 SQL 语句的时候，才会在短时间段内打开该日志来做相应的分析。所以，在MySQL 系统中，会对性能产生影响的MySQL 日志（不包括各存储引擎自己的日志）主要就是 Binlog 了。

Binlog 相关参数及优化策略

我们首先看看 Binlog 的相关参数，通过执行如下命令可以获得关于 Binlog 的相关参数。当然，其中也显示出了“ innodb_locks_unsafe_for_binlog”这个 Innodb 存储引擎特有的与 Binlog 相关的参数：

```
mysql> show variables like '%binlog%';
+--------------------------------+------------+
| Variable_name                  | Value      |
+--------------------------------+------------+
| binlog_cache_size              | 1048576    |
| innodb_locks_unsafe_for_binlog | OFF        |
| max_binlog_cache_size          | 4294967295 |
| max_binlog_size                | 1073741824 |
| sync_binlog                    | 0          |
+--------------------------------+------------+
```

“binlog_cache_size″：在事务过程中容纳二进制日志 SQL 语句的缓存大小。二进制日志缓存是服务器支持事务存储引擎并且服务器启用了二进制日志(—log-bin 选项)的前提下为每个客户端分配的内存，注意，是每个 Client 都可以分配设置大小的 binlog cache 空间。如果读者朋友的系统中经常会出现多语句事务的华，可以尝试增加该值的大小，以获得更有的性能。当然，我们可以通过 MySQL 的以下两个状态变量来判断当前的 binlog_cache_size 的状况：Binlog_cache_use 和 Binlog_cache_disk_use。

“max_binlog_cache_size”：和″binlog_cache_size″相对应，但是所代表的是 binlog 能够使用的最大 cache 内存大小。当我们执行多语句事务的时候，max_binlog_cache_size 如果不够大的话，系统可能会报出“Multi-statement transaction required more than 'max_binlog_cache_size' bytes of storage”的错误。

“max_binlog_size”：Binlog 日志最大值，一般来说设置为 512M 或者 1G，但不能超过 1G。该大小并不能非常严格控制Binlog 大小，尤其是当到达Binlog 比较靠近尾部而又遇到一个较大事务的时候，系统为了保证事务的完整性，不可能做切换日志的动作，只能将该事务的所有 SQL 都记录进入当前日志，直到该事务结束。这一点和 Oracle 的 Redo 日志有点不一样，因为 Oracle 的 Redo 日志所记录的是数据文件的物理位置的变化，而且里面同时记录了 Redo 和 Undo 相关的信息，所以同一个事务是否在一个日志中对 Oracle 来说并不关键。而 MySQL 在 Binlog 中所记录的是数据库逻辑变化信息，MySQL 称之为 Event，实际上就是带来数据库变化的 DML 之类的 Query 语句。

“sync_binlog”：这个参数是对于MySQL系统来说是至关重要的，他不仅影响到Binlog对MySQL所带来的性能损耗，而且还影响到MySQL中数据的完整性。对于“sync_binlog”参数的各种设置的说明如下：

- sync_binlog=0，当事务提交之后，MySQL不做fsync之类的磁盘同步指令刷新binlog_cache中的信息到磁盘，而让Filesystem自行决定什么时候来做同步，或者cache满了之后才同步到磁盘。
- sync_binlog=n，当每进行n次事务提交之后，MySQL将进行一次fsync之类的磁盘同步指令来将binlog_cache中的数据强制写入磁盘。

在MySQL中系统默认的设置是sync_binlog=0，也就是不做任何强制性的磁盘刷新指令，这时候的性能是最好的，但是风险也是最大的。因为一旦系统Crash，在binlog_cache中的所有binlog信息都会被丢失。而当设置为“1”的时候，是最安全但是性能损耗最大的设置。因为当设置为1的时候，即使系统Crash，也最多丢失binlog_cache中未完成的一个事务，对实际数据没有任何实质性影响。从以往经验和相关测试来看，对于高并发事务的系统来说，“sync_binlog”设置为0和设置为1的系统写入性能差距可能高达5倍甚至更多。

大家都知道，MySQL的复制（Replication），实际上就是通过将Master端的Binlog通过利用IO线程通过网络复制到Slave端，然后再通过SQL线程解析Binlog中的日志再应用到数据库中来实现的。所以，Binlog量的大小对IO线程以及Msater和Slave端之间的网络都会产生直接的影响。

MySQL中Binlog的产生量是没办法改变的，只要我们的Query改变了数据库中的数据，那么就必须将该Query所对应的Event记录到Binlog中。那我们是不是就没有办法优化复制了呢？当然不是，在MySQL复制环境中，实际上是是有8个参数可以让我们控制需要复制或者需要忽略而不进行复制的DB或者Table的，分别为：

- Binlog_Do_DB：设定哪些数据库（Schema）需要记录Binlog；
- Binlog_Ignore_DB：设定哪些数据库（Schema）不要记录Binlog；
- Replicate_Do_DB：设定需要复制的数据库（Schema），多个DB用逗号（“,”）分隔；
- Replicate_Ignore_DB：设定可以忽略的数据库（Schema）；
- Replicate_Do_Table：设定需要复制的Table；
- Replicate_Ignore_Table：设定可以忽略的Table；
- Replicate_Wild_Do_Table：功能同Replicate_Do_Table，但可以带通配符来进行设置；
- Replicate_Wild_Ignore_Table：功能同Replicate_Ignore_Table，可带通配符设置；

通过上面这八个参数，我们就可以非常方便按照实际需求，控制从Master端到Slave端的 Binlog量尽可能的少，从而减小Master端到Slave端的网络流量，减少IO线程的IO量，还能减少SQL线程的解析与应用SQL的数量，最终达到改善Slave上的数据延时问题。

实际上，上面这八个参数中的前面两个是设置在Master端的，而后面六个参数则是设置在Slave端的。虽然前面两个参数和后面六个参数在功能上并没有非常直接的关系，但是对于优化MySQL的Replication来说都可以启到相似的功能。当然也有一定的区别，其主要区别如下：

- 如果在Master端设置前面两个参数，不仅仅会让Master端的Binlog记录所带来的IO量减少，还会让Master端的IO线程就可以减少Binlog的读取量，传递给Slave端的IO线程的Binlog量自然就会较少。这样做的好处是可以减少网络IO，减少Slave端IO线程的IO量，减少Slave端的SQL线程的工作量，从而最大幅度的优化复制性能。当然，在Master端设置也存在一定的弊端，因为MySQL的判断是否需要复制某个Event不是根据产生该Event的Query所更改的数据

所在的DB，而是根据执行Query时刻所在的默认Schema，也就是我们登录时候指定的DB或者运行“USE DATABASE”中所指定的DB。只有当前默认DB和配置中所设定的DB完全吻合的时候 IO线程才会将该Event读取给Slave的IO线程。所以如果在系统中出现在默认DB和设定需要复制的DB不一样的情况下改变了需要复制的DB中某个Table的数据的时候，该Event是不会被复制到Slave中去的，这样就会造成Slave端的数据和Master的数据不一致的情况出现。同样，如果在默认Schema下更改了不需要复制的Schema中的数据，则会被复制到Slave端，当Slave端并没有该Schema的时候，则会造成复制出错而停止；

- 而如果是在Slave端设置后面的六个参数，在性能优化方面可能比在Master端要稍微逊色一点，因为不管是需要还是不需要复制的Event都被会被IO线程读取到Slave端，这样不仅仅增加了网络IO量，也给Slave端的IO线程增加了Relay Log的写入量。但是仍然可以减少Slave的SQL线程在Slave端的日志应用量。虽然性能方面稍有逊色，但是在Slave端设置复制过滤机制，可以保证不会出现因为默认Schema的问题而造成Slave和Master数据不一致或者复制出错的问题。

Slow Query Log相关参数及使用建议

再来看看Slow Query Log的相关参数配置。有些时候，我们为了定位系统中效率比较地下的Query语句，则需要打开慢查询日志，也就是Slow Query Log。我们可以如下查看系统慢查询日志的相关设置：

```
mysql> show variables like 'log_slow%';
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| log_slow_queries | ON    |
+------------------+-------+
1 row in set (0.00 sec)

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

“log_slow_queries”参数显示了系统是否已经打开Slow Query Log功能，而“long_query_time”参数则告诉我们当前系统设置的Slow Query记录执行时间超过多长的Query。在MySQL AB发行的MySQL版本中Slow Query Log可以设置的最短慢查询时间为1秒，这在有些时候可能没办法完全满足我们的要求，如果希望能够进一步缩短慢查询的时间限制，可以使用Percona提供的microslow-patch（件成为msl Patch）来突破该限制。msl patch不仅仅能将慢查询时间减小到毫秒级别，同时还能通过一些特定的规则来过滤记录的SQL，如仅记录涉及到某个表的Slow Query等等附加功能。考虑到篇幅问题，这里就不介绍msl patch给我们带来的更为详细的功能和使用，大家请参考官方介绍（http://www.mysqlperformanceblog.com/2008/04/20/updated-msl-microslow-patch-installation-walk-through/）

打开 Slow Query Log 功能对系统性能的整体影响没有 Binlog 那么大，毕竟 Slow Query Log 的数据量比较小，带来的 IO 损耗也就较小，但是，系统需要计算每一条 Query 的执行时间，所以消耗总是会有一些的，主要是 CPU 方面的消耗。如果大家的系统在 CPU 资源足够丰富的时候，可以不必在乎这一点点损耗，毕竟他可能会给我们带来更大性能优化的收获。但如果我们的 CPU 资源也比较紧张的时候，也完全可以在大部分时候关闭该功能，而只需要间断性的打开 Slow Query Log 功能来定位可能存在的慢查询。

MySQL 的其他日志由于使用很少（Query Log）或者性能影响很少，我们就不在此过多分析了，至于各个存储引擎相关的日志，我们留在后面“常用存储引擎优化”部分再做相应的分析。

10.3 Query Cache 优化

谈到 Query Cache，恐怕使用过 MySQL 的大部分人都会或多或少有一些了解，因为在很多人看来他可以帮助我们将数据库的性能产生一个“质”的提升。但真的是这样吗？这一节我们就将如何合理的使用 MySQL 的 Query Cache 进行一些相应的分析并得出部分优化建议。

Query Cache 真的是“尚方宝剑”吗？

MySQL 的 Query Cache 实现原理实际上并不是特别的复杂，简单的来说就是将客户端请求的 Query 语句（当然仅限于 SELECT 类型的 Query）通过一定的 hash 算法进行一个计算而得到一个 hash 值，存放在一个 hash 桶中。同时将该 Query 的结果集（Result Set）也存放在一个内存 Cache 中的。存放 Query hash 值的链表中的每一个 hash 值所在的节点中同时还存放了该 Query 所对应的 Result Set 的 Cache 所在的内存地址，以及该 Query 所涉及到的所有 Table 的标识等其他一些相关信息。系统接受到任何一个 SELECT 类型的 Query 的时候，首先计算出其 hash 值，然后通过该 hash 值到 Query Cache 中去匹配，如果找到了完全相同的 Query，则直接将之前所 Cache 的 Result Set 返回给客户端而完全不需要进行后面的任何步骤即可完成这次请求。而后端的任何一个表的任何一条数据发生变化之后，也会通知 Query Cache，需要将所有与该 Table 有关的 Query 的 Cache 全部失效，并释放出之前占用的内存地址，以便后面其他的 Query 能够使用。

从上面的实现原理来看，Query Cache 确实是以比较简单的实现带来巨大性能收益的功能。但是很多人可能都忽略了使用 QueryCache 之后所带来的负面影响：

a) Query 语句的 hash 运算以及 hash 查找资源消耗。当我们使用 Query Cache 之后，每条 SELECT 类型的 Query 在到达 MySQL 之后，都需要进行一个 hash 运算然后查找是否存在该 Query 的 Cache，虽然这个 hash 运算的算法可能已经非常高效了，hash 查找的过程也已经足够的优化了，对于一条 Query 来说消耗的资源确实是非常非常的少，但是当我们每秒都有上千甚至几千条 Query 的时候，我们就不能对产生的 CPU 的消耗完全忽视了。
b) Query Cache 的失效问题。如果我们的表变更比较频繁，则会造成 Query Cache 的失效率非常高。这里的表变更不仅仅指表中数据的变更，还包括结构或者索引等的任何变更。也就是说我们每次缓存到 Query Cache 中的 Cache 数据可能在刚存入后很快就会因为表中的数据被改变而被清除，然后新的相同 Query 进来之后无法使用到之前的 Cache。
c) Query Cache 中缓存的是 Result Set ，而不是数据页，也就是说，存在同一条记录被 Cache 多次的可能性存在。从而造成内存资源的过渡消耗。当然，可能有人会说我们可以限定 Query Cache 的大小啊。是的，我们确实可以限定 Query Cache 的大小，但是这样，Query Cache 就很容易造成因为内存不足而被换出，造成命中率的下降。

对于 Query Cache 的上面三个负面影响，如果单独拿出每一个影响来说都不会造成对整个系统多大的问题，并不会让大家对使用 Query Cache 产生太多顾虑。但是，当综合这三个负面影响一起考虑的话，恐怕 Query Cache 在很多人心目中就不再是以前的那把“尚方宝剑”了。

适度使用 Query Cache

虽然 Query Cache 的使用会存在一些负面影响，但是我们也应该相信其存在是必定有一定价值。我们完全不用因为 Query Cache 的上面三个负面影响就完全失去对 Query Cache 的信心。只要我们理解了 Query Cache 的实现原理，那么我们就完全可以通过一定的手段在使用 Query Cache 的时候扬长避短，重发发挥其优势，并有效的避开其劣势。

首先，我们需要根据 Query Cache 失效机制来判断哪些表适合使用 Query 哪些表不适合。由于 Query Cache 的失效主要是因为 Query 所依赖的 Table 的数据发生了变化，造成 Query 的 Result Set 可能已经有所改变而造成相关的 Query Cache 全部失效，那么我们就应该避免在查询变化频繁的 Table 的 Query 上使用，而应该在那些查询变化频率较小的 Table 的 Query 上面使用。MySQL 中针对 Query Cache 有两个专用的 SQL Hint（提示）：SQL_NO_CACHE 和 SQL_CACHE，分别代表强制不使用 Query Cache 和强制使用 Query Cache。我们完全可以利用这两个 SQL Hint，让 MySQL 知道我们希望哪些 SQL 使用 Query Cache 而哪些 SQL 就不要使用了。这样不仅可以让变化频繁 Table 的 Query 浪费 Query Cache 的内存，同时还可以减少 Query Cache 的检测量。

其次，对于那些变化非常小，大部分时候都是静态的数据，我们可以添加 SQL_CACHE 的 SQL Hint，强制 MySQL 使用 Query Cache，从而提高该表的查询性能。

最后，有些 SQL 的 Result Set 很大，如果使用 Query Cache 很容易造成 Cache 内存的不足，或者将之前一些老的 Cache 冲刷出去。对于这一类 Query 我们有两种方法可以解决，一是使用 SQL_NO_CACHE 参数来强制他不使用 Query Cache 而每次都直接从实际数据中去查找，另一种方法是通过设定“query_cache_limit”参数值来控制 Query Cache 中所 Cache 的最大 Result Set ，系统默认为 1M（1048576）。当某个 Query 的 Result Set 大于“query_cache_limit”所设定的值的时候，Query Cache 是不会 Cache 这个 Query 的。

Query Cache 的相关系统参数变量和状态变量

我们首先看看 Query Cache 的系统变量，可以通过执行如下命令获得 MySQL 中 Query Cache 相关的系统参数变量：

```
mysql> show variables like '%query_cache%';
+------------------------------+-----------+
| Variable_name                | Value     |
+------------------------------+-----------+
| have_query_cache             | YES       |
| query_cache_limit            | 1048576   |
| query_cache_min_res_unit     | 4096      |
| query_cache_size             | 268435456 |
| query_cache_type             | ON        |
```

```
| query_cache_wlock_invalidate | OFF       |
+------------------------------+-----------+
```

- “have_query_cache”：该MySQL是否支持Query Cache；
- “query_cache_limit”：Query Cache存放的单条Query最大 Result Set ，默认1M；
- “query_cache_min_res_unit”：Query Cache每个 Result Set 存放的最小内存大小，默认4k；
- “query_cache_size”：系统中用于Query Cache内存的大小；
- “query_cache_type”：系统是否打开了Query Cache功能；
- “query_cache_wlock_invalidate”：针对于MyISAM存储引擎，设置当有WRITE LOCK在某个Table上面的时候，读请求是要等待 WRITE LOCK 释放资源之后再查询还是允许直接从Query Cache中读取结果，默认为FALSE（可以直接从Query Cache中取得结果）。

以上参数的设置主要是“query_cache_limit”和“query_cache_min_res_unit”两个参数的设置需要做一些针对于应用的相关调整。如果我们需要Cache的 Result Set 一般都很小（小于4k）的话，可以适当将“query_cache_min_res_unit”参数再调小一些，避免造成内存的浪费，“query_cache_limit”参数则不用调整。而如果我们需要Cache的 Result Set 大部分都大于4k的话，则最好将“query_cache_min_res_unit”调整到和 Result Set 大小差不多，“query_cache_limit”的参数也应大于 Result Set 的大小。当然，可能有些时候我们比较难准确的估算 Result Set 的大小，那么当 Result Set 较大的时候，我们也并不是非得将“query_cache_min_res_unit”设置的和每个Result Set 差不多大，是每个结果集的一半或者四分之一大小都可以，要想非常完美的完全不浪费任何内存确实也是不可能做到的。

如果我们要了解Query Cache的使用情况，则可以通过Query Cache相关的状态变量来获取，如通过如下命令：

```
mysql> show status like 'Qcache%';
+-------------------------+------------+
| Variable_name           | Value      |
+-------------------------+------------+
| Qcache_free_blocks      | 7499       |
| Qcache_free_memory      | 190662000  |
| Qcache_hits             | 1888430018 |
| Qcache_inserts          | 1014096388 |
| Qcache_lowmem_prunes    | 106071885  |
| Qcache_not_cached       | 7951123988 |
| Qcache_queries_in_cache | 19315      |
| Qcache_total_blocks     | 47870      |
+-------------------------+------------+
```

- “Qcache_free_blocks”：Query Cache中目前还有多少剩余的blocks。如果该值显示较大，则说明Query Cache中的内存碎片较多了，可能需要寻找合适的机会进行整理（）。
- “Qcache_free_memory”：Query Cache中目前剩余的内存大小。通过这个参数我们可以较为准确的观察出当前系统中的Query Cache内存大小是否足够，是需要增加还是过多了；
- “Qcache_hits”：多少次命中。通过这个参数我们可以查看到Query Cache的基本效果；
- “Qcache_inserts”：多少次未命中然后插入。通过“Qcache_hits”和“Qcache_inserts”两个参数我们就可以算出Query Cache的命中率了：

Query Cache 命中率 = Qcache_hits / (Qcache_hits + Qcache_inserts);

- “Qcache_lowmem_prunes”：多少条 Query 因为内存不足而被清除出 Query Cache。通过“Qcache_lowmem_prunes”和“Qcache_free_memory”相互结合，能够更清楚的了解到我们系统中 Query Cache 的内存大小是否真的足够，是否非常频繁的出现因为内存不足而有 Query 被换出
- “Qcache_not_cached”：因为 query_cache_type 的设置或者不能被 cache 的 Query 的数量；
- “Qcache_queries_in_cache”：当前 Query Cache 中 cache 的 Query 数量；
- “Qcache_total_blocks”：当前 Query Cache 中的 block 数量；

Query Cache 的限制

Query Cache 由于存放的都是逻辑结构的 Result Set，而不是物理的数据页，所以在性能提升的同时，也会受到一些特定的限制。

a) 5.1.17 之前的版本不能 Cache 帮定变量的 Query，但是从 5.1.17 版本开始，Query Cache 已经开始支持帮定变量的 Query 了；
b) 所有子查询中的外部查询 SQL 不能被 Cache；
c) 在 Procedure，Function 以及 Trigger 中的 Query 不能被 Cache；
d) 包含其他很多每次执行可能得到不一样结果的函数的 Query 不能被 Cache。

鉴于上面的这些限制，在使用 Query Cache 的过程中，建议通过精确设置的方式来使用，仅仅让合适的表的数据可以进入 Query Cache，仅仅让某些 Query 的查询结果被 Cache。

10.4 MySQL Server 其他常用优化

除了安装，日志，Query Cache 之外，可能影响 MySQL Server 整体性能的设置其他很多方面，如网络连接，线程管理，Table 管理等。这一节我们将分析除了前面几节内容之外的可能影响 MySQL Server 性能的其他可优化的部分。

网络连接与连接线程

虽然 MySQL 的连接方式不仅仅只有通过网络方式，还可以通过命名管道的方式，但是不论是何种方式连接 MySQL，在 MySQL 中都是通过线程的方式管理所有客户端请求的连接。每一个客户端连接都会有一个与之对应的生成一个连接线程。我们先看一下与网络连接的性能配置项及对性能的影响。

- max_conecctions：整个 MySQL 允许的最大连接数；
  这个参数主要影响的是整个 MySQL 应用的并发处理能力，当系统中实际需要的连接量大于 max_conecctions 的情况下，由于 MySQL 的设置限制，那么应用中必然会产生连接请求的等待，从而限制了相应的并发量。所以一般来说，只要 MySQL 主机性能允许，都是将该参数设置的尽可能大一点。一般来说 500 到 800 左右是一个比较合适的参考值

- max_user_connections：每个用户允许的最大连接数；
  上面的参数是限制了整个 MySQL 的连接数，而 max_user_connections 则是针对于单个用户的连接限制。在一般情况下我们可能都较少使用这个限制，只有在一些专门提供 MySQL 数据存储服务，或者是提供虚拟主机服务的应用中可能需要用到。除了限制的对象区别之外，其他方面和 max_connections 一样。这个参数的设置完全依赖于应用程序的连接用户数，对于普通的应用来

说，完全没有做太多的限制，可以尽量放开一些。

- net_buffer_length：网络包传输中，传输消息之前的 net buffer 初始化大小；
  这个参数主要可能影响的是网络传输的效率，由于该参数所设置的只是消息缓冲区的初始化大小，所以造成的影响主要是当我们的每次消息都很大的时候 MySQL 总是需要多次申请扩展该缓冲区大小。系统默认大小为 16KB，一般来说可以满足大多数场景，当然如果我们的查询都是非常小，每次网络传输量都很少，而且系统内存又比较紧缺的情况下，也可以适当将该值降低到 8KB。

- max_allowed_packet：在网络传输中，一次传消息输量的最大值；
  这个参数与 net_buffer_length 相对应，只不过是 net buffer 的最大值。当我们的消息传输量大于 net_buffer_length 的设置时，MySQL 会自动增大 net buffer 的大小，直到缓冲区大小达到 max_allowed_packet 所设置的值。系统默认值为 1MB，最大值是 1GB，必须设定为 1024 的倍数，单位为字节。

- back_log：在 MySQL 的连接请求等待队列中允许存放的最大连接请求数。
  连接请求等待队列，实际上是指当某一时刻客户端的连接请求数量过大的时候，MySQL 主线程没办法及时给每一个新的连接请求分配（或者创建）连接线程的时候，还没有分配到连接线程的所有请求将存放在一个等待队列中，这个队列就是 MySQL 的连接请求队列。当我们的系统存在瞬时的大量连接请求的时候，则应该注意 back_log 参数的设置。系统默认值为 50，最大可以设置为 65535。当我们增大 back_log 的设置的时候，同时还需要主义 OS 级别对网络监听队列的限制，因为如果 OS 的网络监听设置小于 MySQL 的 back_log 设置的时候，我们加大“back_log”设置是没有意义的。

上面介绍了网络连接交互相关的主要优化设置，下面我们再来看看与每一个客户端连接想对应的连接线程。

在 MySQL 中，为了尽可提高客户端请求创建连接这个过程的性能，实现了一个 Thread Cache 池，将空闲的连接线程存放在其中，而不是完成请求后就销毁。这样，当有新的连接请求的时候，MySQL 首先会检查 Thread Cache 池中是否存在空闲连接线程，如果存在则取出来直接使用，如果没有空闲连接线程，才创建新的连接线程。在 MySQL 中与连接线程相关的系统参数及状态变量说明如下：

- thread_cache_size：Thread Cache 池中应该存放的连接线程数。
  当系统最初启动的时候，并不会马上就创建 thread_cache_size 所设置数目的连接线程存放在 Thread Cache 池中，而是随着连接线程的创建及使用，慢慢的将用完的连接线程存入其中。当存放的连接线程达到 thread_cache_size 值之后，MySQL 就不会再续保存用完的连接线程了。

  如果我们的应用程序使用的短连接，Thread Cache 池的功效是最明显的。因为在短连接的数据库应用中，数据库连接的创建和销毁是非常频繁的，如果每次都需要让 MySQL 新建和销毁相应的连接线程，那么这个资源消耗实际上是非常大的，而当我们使用了 Thread Cache 之后，由于连接线程大部分都是在创建好了等待取用的状态，既不需要每次都重新创建，又不需要在使用完之后销毁，所以可以节省下大量的系统资源。所以在短连接的应用系统中，thread_cache_size 的值应该设置的相对大一些，不应该小于应用系统对数据库的实际并发请求数。

而如果我们使用的是长连接的时候，Thread Cache 的功效可能并没有使用短连接那样的大，但也并不是完全没有价值。因为应用程序即使是使用了长连接，也很难保证他们所管理的所有连接都能处于很稳定的状态，仍然会有不少连接关闭和新建的操作出现。在有些并发量较高，应用服务器数量较大的系统中，每分钟十來次的连接创建与关闭的操作是很常见的。而且如果应用服务器的连接池管理不是太好，容易产生连接池抖动的话，所产生的连接创建和销毁操作将会更多。所以即使是在使用长连接的应用环境中，Thread Cache 机制的利用仍然是对性能大有帮助的。只不过在长连接的环境中我们不需要将 thread_cache_size 参数设置太大，一般来说可能 50 到 100 之间应该就可以了。

- thread_stack：每个连接线程被创建的时候，MySQL 给他分配的内存大小。
  当 MySQL 创建一个新的连接线程的时候，是需要给他分配一定大小的内存堆栈空间，以便存放客户端的请求 Query 以及自身的各种状态和处理信息。不过一般来说如果不是对 MySQL 的连接线程处理机制十分熟悉的话，不应该轻易调整该参数的大小，使用系统的默认值（192KB）基本上可以所有的普通应用环境。如果该值设置太小，会影响 MySQL 连接线程能够处理客户端请求的 Query 内容的大小，以及用户创建的 Procedures 和 Functions 等。

上面介绍的这些都是我们可以怎样配置网络连接交互以及连接线程的性能相关参数，下面我们再看看该怎样检验上面所做的设置是否合理，是否有需要调整的地方。我们可以通过在系统中执行如下的几个命令来获得相关的状态信息来帮助大家检验设置的合理性：

我们现看看连接线程相关的系统变量的设置值：

```
mysql> show variables like 'thread%';
+-------------------+--------+
| Variable_name     | Value  |
+-------------------+--------+
| thread_cache_size | 64     |
| thread_stack      | 196608 |
+-------------------+--------+
```

再来看一下系统被连接的次数以及当前系统中连接线程的状态值：

```
mysql> show status like 'connections';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Connections   | 127   |
+---------------+-------+

mysql> show status like '%thread%';
+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| Delayed_insert_threads | 0     |
| Slow_launch_threads    | 0     |
| Threads_cached         | 4     |
```

```
| Threads_connected       | 7     |
| Threads_created         | 11    |
| Threads_running         | 1     |
+----------------------+-------+
```

通过上面的命令，我们可以看出，系统设置了 Thread Cache 池最多将缓存 32 个连接线程，每个连接线程创建之初，系统分配 192KB 的内存堆栈空给他。系统启动到现在共接收到客户端的连接 127 次，共创建了 11 个连接线程，但前有 7 个连接线程处于和客户端连接的状态，而 7 个连接状态的线程中只有一个是 active 状态，也就是说只有一个正在处理客户端提交的俄请求。而在 Thread Cache 池中当共 Cache 了 4 个连接线程。

通过系统设置和当前状态的分析，我们可以发现，thread_cache_size 的设置已经足够了，甚至还远大于系统的需要。所以我们可以适当减少 thread_cache_size 的设置，比如设置为 8 或者 16。根据 Connections 和 Threads_created 这两个系统状态值，我们还可以计算出系统新建连接连接的 Thread Cache 命中率，也就是通过 Thread Cache 池中取得连接线程的次数与系统接收的总连接次数的比率，如下：

Threads_Cache_Hit = (Connections - Threads_created) / Connections * 100%

我们可以通过上面的这个运算公式计算一下上面环境中的 Thread Cache 命中率：Thread_Cache_Hit = (127 - 12) / 127 * 100% = 90.55%

一般来说，当系统稳定运行一段时间之后，我们的 Thread Cache 命中率应该保持在 90%左右甚至更高的比率才算正常。可以看出上面环境中的 Thread Cache 命中比率基本还算是正常的。

Table Cache 相关的优化

我们先来看一下 MySQL 打开表的相关机制。由于多线程的实现机制，为了尽可能的提高性能，在 MySQL 中每个线程都是独立的打开自己需要的表的文件描述符，而不是通过共享已经打开的表的文件描述符的机制来实现。当然，针对于不同的存储引擎可能有不同的处理方式。如 MyISAM 表，每一个客户端线程打开任何一个 MyISAM 表的数据文件都需要打开一个文件描述符，但如果是索引文件，则可以多个线程共享同一个索引文件的描述符。对于 Innodb 的存储引擎，如果我们使用的是共享表空间来存储数据，那么我们需要打开的文件描述符就比较少，而如果我们使用的是独享表空间方式来存储数据，则同样，由于存储表数据的数据文件较多，则同样会打开很多的表文件描述符。除了数据库的实际表或者索引打开以外，临时文件同样也需要使用文件描述符，同样会占用系统中 open_files_limit 的设置限额。

为了解决打开表文件描述符太过频繁的问题，MySQL 在系统中实现了一个 Table Cache 的机制，和前面介绍的 Thread Cache 机制有点类似，主要就是 Cache 打开的所有表文件的描述符，当有新的请求的时候不需要再重新打开，使用结束的时候也不用立即关闭。通过这样的方式来减少因为频繁打开关闭文件描述符所带来的资源消耗。我们先看一看 Table Cache 相关的系统参数及状态变量。

在 MySQL 中我们通过 table_cache（从 MySQL5.1.3 开始改为 table_open_cache），来设置系统中为我们 Cache 的打开表文件描述符的数量。通过 MySQL 官方手册中的介绍，我们设置 table_cache 大小的时候应该通过 max_connections 参数计算得来，公式如下：

table_cache = max_connections * N;

其中N代表单个Query语句中所包含的最多Table的数量。但是我个人理解这样的计算其实并不是太准确，分析如下：

首先，max_connections是系统同时可以接受的最大连接数，但是这些连接并不一定都是active状态的，也就是说可能里面有不少连接都是处于Sleep状态。而处于Sleep状态的连接是不可能打开任何Table的。

其次，这个N为执行Query中包含最多的Table的Query所包含的Table的个数也并不是太合适，因为我们不能忽略索引文件的打开。虽然索引文件在各个连接线程之间是可以共享打开的连接描述符的，但总还是需要的。而且，如果我Query中的每个表的访问都是通过现通过索引定位检索的，甚至可能还是通过多个索引，那么该Query的执行所需要打开的文件描述符就更多了，可能是N的两倍甚至三倍。

最后，这个计算的公式只能计算出我们同一时刻需要打开的描述符的最大数量，而table_cache的设置也不一定非得根据这个极限值来设定，因为table_cache所设定的只是Cache打开的描述符的数量的大小，而不是最多能够打开的量的大小。

当然，上面的这些只是我个人的理解，也可能并不是太严谨，各位读者朋友如果觉得有其他的理解完全可以提出来大家再探讨。

我们可以通过如下方式查看table_cache的设置和当前系统中的使用状况：

```
mysql> show variables like 'table_cache';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| table_cache   | 512   |
+---------------+-------+

mysql> show status like 'open_tables';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Open_tables   | 6     |
+---------------+-------+
```

上面的结果显示系统设置的table_cache为512个，也就是说在该MySQL中，Table Cache中可以Cache 512个打开文件的描述符；当前系统中打开的描述符仅仅则只有6个。

那么Table Cache池中Cache的描述符在什么情况下会被关闭呢？一般来说主要有以下集中情况会出现被Cache的描述符被关闭：

a) Table Cache的Cache池满了，而某个连接线程需要打开某个不在Table Cache中的表时，MySQL会通过一定的算法关闭某些没有在使用中的描述符；
b) 当我们执行Flush Table等命令的时候，MySQL会关闭当前Table Cache中Cache的所有文件描述符；
c) 当Table Cache中Cache的量超过table_cache参数设置的值的时候；

Sort Buffer，Join Buffer 和 Read Buffer

在 MySQL 中，之前介绍的多种 Cache 之外，还有在 Query 执行过程中的两种 Buffer 会对数据库的整体性能产生影响。

```
mysql> show variables like '%buffer%';
+-------------------------------+----------+
| Variable_name                 | Value    |
+-------------------------------+----------+
    ... ...
| join_buffer_size              | 4190208  |
    ... ...
| sort_buffer_size              | 2097144  |
+-------------------------------+----------+
```

- join_buffer_size：当我们的 Join 是 ALL，index，rang 或者 index_merge 的时候使用的 Buffer；
  实际上这种 Join 被称为 Full Join。实际上参与 Join 的每一个表都需要一个 Join Buffer，所以在 Join 出现的时候，至少是两个。Join Buffer 的设置在 MySQL 5.1.23 版本之前最大为 4GB，但是从 5.1.23 版本开始，在除了 Windows 之外的 64 位的平台上可以超出 4BG 的限制。系统默认是 128KB。

- sort_buffer_size：系统中对数据进行排序的时候使用的 Buffer；
  Sort Buffer 同样是针对单个 Thread 的，所以当多个 Thread 同时进行排序的时候，系统中就会出现多个 Sort Buffer。一般我们可以通过增大 Sort Buffer 的大小来提高 ORDER BY 或者是 GROUP BY 的处理性能。系统默认大小为 2MB，最大限制和 Join Buffer 一样，在 MySQL 5.1.23 版本之前最大为 4GB，从 5.1.23 版本开始，在除了 Windows 之外的 64 位的平台上可以超出 4GB 的限制。

如果应用系统中很少有 Join 语句出现，则可以不用太在乎 join_buffer_size 参数的大小设置，但是如果 Join 语句不是很少的话，个人建议可以适当增大 join_buffer_size 的设置到 1MB 左右，如果内存充足甚至可以设置为 2MB。对于 sort_buffer_size 参数来说，一般设置为 2MB 到 4MB 之间可以满足大多数应用的需求。当然，如果应用系统中的排序都比较大，内存充足且并发量不是特别的大的时候，也可以继续增大 sort_buffer_size 的设置。在这两个 Buffer 设置的时候，最需要注意的就是不要忘记是每个 Thread 都会创建自己独立的 Buffer，而不是整个系统共享的 Buffer，不要因为设置过大而造成系统内存不足。

10.5 小结

通过参数设置来进行性能优化所能够带来的性能提升可能并不会如很多人想象的那样产生质的飞跃，除非是之前的设置存在严重的不合理情况。我们不能将性能调优完全依托在通过 DBA 在数据库上线后的参数调整之上，而应该在系统设计和开发阶段就尽可能减少性能问题。当然，也不能否认参数调整在某些场景下对系统性能的影响比较大，但毕竟只是少数的特殊情况。

第 11 章 常用存储引擎优化

前言：

MySQL 提供的非常丰富的存储引擎种类供大家选择，有多种选择固然是好事，但是需要我们理解掌握的知识也会增加很多。每一种存储引擎都有各自的特长，也都存在一定的短处。如何将各种存储引擎在自己的应用环境中结合使用，扬长避短，也是一门不太简单的学问。本章选择最为常用的两种存储引擎进行针对性的优化建议，希望能够对读者朋友有一定的帮助。

11.1 MyISAM存储引擎优化

我们知道，MyISAM 存储引擎是 MySQL 最为古老的存储引擎之一，也是最为流行的存储引擎之一。对于以读请求为主的非事务系统来说，MyISAM 存储引擎由于其优异的性能表现及便利的维护管理方式无疑是大家最优先考虑的对象。这一节我们将通过分析 MyISAM 存储引擎的相关特性，来寻找提高 MyISAM 存储引擎性能的优化策略。

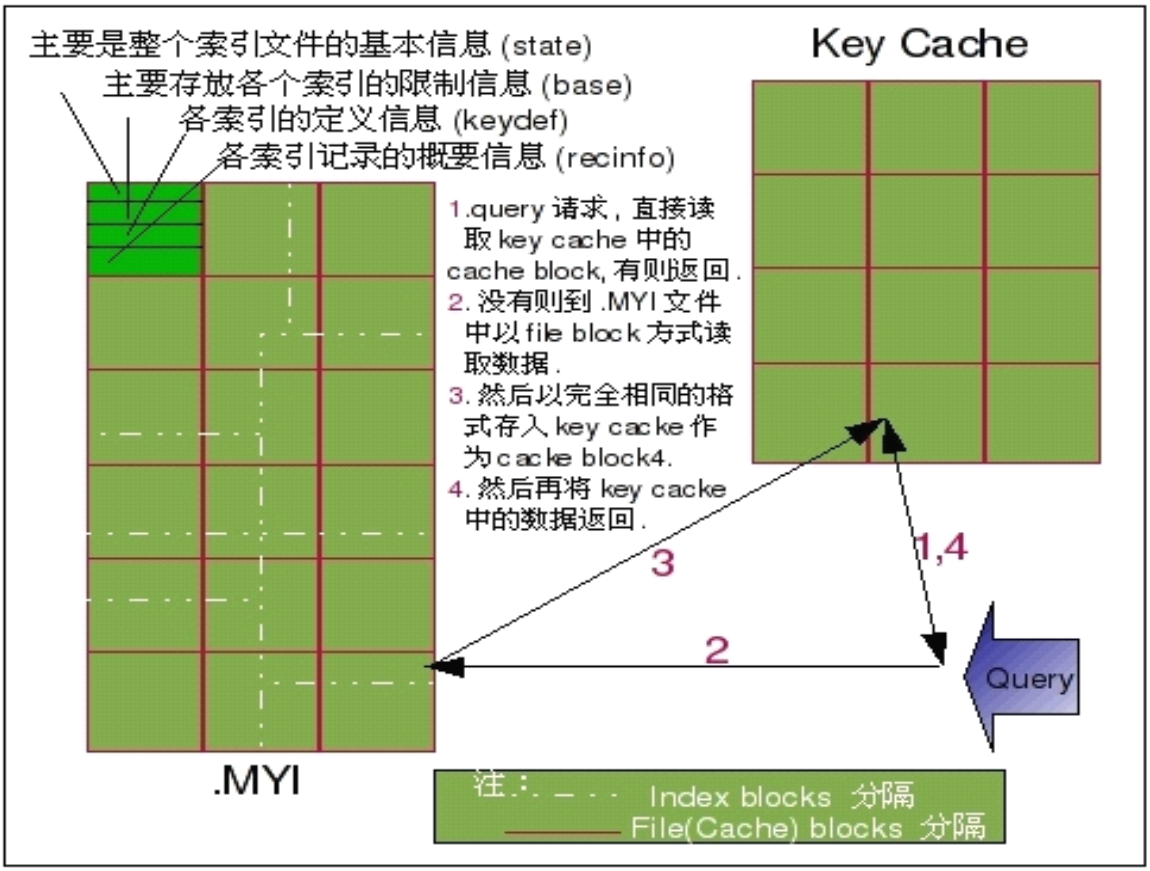
索引缓存优化

MyISAM 存储引擎的缓存策略是其和很多其他数据库乃至 MySQL 数据库的很多其他存储引擎不太一样的最大特性。因为他仅仅缓存索引数据，并不会缓存实际的表数据信息到内存中，而是将这一工作交给了 OS 级别的文件系统缓存。　所以，在数据库优化中非常重要的优化环节之一“缓存优化”的工作在使用 MyISAM 存储引擎的数据库的情况下，就完全集中在对索引缓存的优化上面了。

在分析优化索引缓存策略之前，我们先大概了解一下 MyISAM 存储引擎的索引实现机制以及索引文件的存放格式。

MyISAM 存储引擎的索引和数据是分开存放于“.MYI”文件中，每个“.MYI”文件由文件头和实际的索引数据。“.MYI”的文件头中主要存放四部分信息，分别称为：state（主要是整个索引文件的基本信息），base（各个索引的相关信息，主要是索引的限制信息），keydef（每个索引的定义信息）和 recinfo（每个索引记录的相关信息）。在文件头后面紧接着的就是实际的索引数据信息了。索引数据以 Block（Page）为最小单位，每个 block 中只会存在同一个索引的数据，这主要是基于提高索引的连续读性能的目的。在 MySQL 中，索引文件中索引数据的 block 被称为 Index Block，每个 Index Block 的大小并不一定相等。

在“.MYI”中，Index Block 的组织形式实际上只是一种逻辑上的，并不是物理意义上的。在物理上，实际上是以 File Block 的形式来存放在磁盘上面的。在 Key Cache 中缓存的索引信息是以“Cache Block”的形式组织存放的，“Cache Block”是相同大小的，和“.MYI”文件物理存储的 Block（File Block）一样。在一条 Query 通过索引检索表数据的时候，首先会检查索引缓存（key_buffer_cache）中是否已经有需要的索引信息，如果没有，则会读取“.MYI”文件，将相应的索引数据读入 Key Cache 中的内存空间中，同样也是以 Block 形式存放，被称为 Cache Block。不过，数据的读入并不是以 Index Block 的形式来读入，而是以 File Block 的形式来读入的。以 File Block 形式读入到 Key Cache 之后的 Cache Block 实际上是于 File Block 完全一样的。如下图所示：



当我们从“.MYI”文件中读入 File Block 到 Key Cache 中 Cache Block 时候，如果整个 Key Cache 中已经没有空闲的 Cache Block 可以使用的话，将会通过 MySQL 实现的 LRU 相关算法将某些 Cache Block 清除出去，让新进来的 File Block 有地方呆。

我们先来分析一下与MyISAM索引缓存相关的几个系统参数和状态参数：

- key_buffer_size，索引缓存大小；

  这个参数用来设置整个MySQL中的常规 Key Cache大小。一般来说，如果我们的MySQL是运行在32位平台纸上，此值建议不要超过2GB大小。如果是运行在64位平台纸上则不用考虑此限制，但也最好不要超过4GB。

- key_buffer_block_size，索引缓存中的Cache Block Size；

  在前面我们已经介绍了，在Key Cache中的所有数据都是以Cache Block的形式存在，而key_buffer_block_size就是设置每个Cache Block的大小，实际上也同时限定了我们将“.MYI”文件中的Index Block被读入时候的File Block的大小。

- key_cache_division_limit，LRU链表中的Hot Area和Warm Area分界值；

  实际上，在MySQL的Key Cache中所使用的LRU算法并不像传统的算法一样仅仅只是通过访问频率以及最后访问时间来通过一个唯一的链表实现，而是将其分成了两部分。一部分用来存放使用比较频繁的Hot Cacke Lock（Hot Chain），被成为Hot Area，另外一部分则用来存放使用不是太频繁的Warm Cache Block（Warm Chain），被成为Warm Area。这样做的目的主要是为了保护使用比较频繁的Cache Block更不容易被换出。而key_cache_division_limit参数则是告诉MySQL该如何划分整个Cache Chain划分为Hot Chain和Warm Chain两部分，参数值为Warm Chain占整个Chain的百分比值。设置范围1～100，系统默认为100，也就是只有Warm Chain。

- key_cache_age_threshold，控制Cache Block从Hot Area降到Warm Area的限制；

  key_cache_age_threshold参数控制Hot Area中的Cache Block何时该被降级到Warm Area中。系统默认值为300，最小可以设置为100。值越小，被降级的可能性越大。

通过以上参数的合理设置，我们基本上可以完成MyISAM整体优化的70％的工作。但是如何的合理设置这些参数却不是一个很容易的事情。尤其是key_cache_division_limit和key_cache_age_threshold这两个参数的合理使用。

对于key_buffer_size的设置我们一般需要通过三个指标来计算，第一个是系统索引的总大小，第二个是系统可用物理内存，第三个是根据系统当前的Key Cache命中率。对于一个完全从零开始的全新系统的话，可能出了第二点可以拿到很清楚的数据之外，其他的两个数据都比较难获取，第三点是完全没有。当然，我们可以通过MySQL官方手册中给出的一个计算公式粗略的估算一下我们系统将来的索引大小，不过前提是要知道我们会创建哪些索引，然后通过各索引估算出索引键的长度，以及表中存放数据的条数，公式如下：

Key_Size = key_number * (key_length+4)/0.67

Max_key_buffer_size < Max_RAM - QCache_Usage - Threads_Usage - System_Usage

Threads_Usage = max_connections * (sort_buffer_size + join_buffer_size + read_buffer_size + read_rnd_buffer_size + thread_stack)

当然，考虑到活跃数据的问题，我们并不需要将key_buffer_size设置到可以将所有的索引都放下的大小，这时候我们就需要Key Cache的命中率数据来帮忙了。下面我们再来看一下系统中记录的与Key

Cache 相关的性能状态参数变量。

- Key_blocks_not_flushed，已经更改但还未刷新到磁盘的 Dirty Cache Block；
- Key_blocks_unused，目前未被使用的 Cache Block 数目；
- Key_blocks_used，已经使用了的 Cache Block 数目；
- Key_read_requests，Cache Block 被请求读取的总次数；
- Key_reads，在 Cache Block 中找不到需要读取的 Key 信息后到“.MYI”文件中读取的次数；
- Key_write_requests，Cache Block 被请求修改的总次数；
- Key_writes，在 Cache Block 中找不到需要修改的 Key 信息后到“.MYI”文件中读入再修改的次数；

由于上面各个状态参数在 MySQL 官方文档中都有较为详细的描述，所以上面仅做基本的说明。当我们的系统上线之后，我们就可以通过上面这些状态参数的状态值得到系统当前的 Key Cache 使用的详细情况和性能状态。

Key_buffer_UsageRatio = (1 - Key_blocks_used/(Key_blocks_used + Key_blocks_unused)) * 100%

Key_Buffer_Read_HitRatio = (1 - Key_reads/Key_read_requests) * 100%

Key_Buffer_Write_HitRatio = (1 - Key_writes/Key_Write_requests) * 100%

通过上面的这三个比率数据，就可以很清楚的知道我们的 Key Cache 设置是否合理，尤其是 Key_Buffer_Read_HitRatio 参数和 Key_buffer_UsageRatio 这两个比率。一般来说 Key_buffer_UsageRatio 应该在 99%以上甚至 100%，如果该值过低，则说明我们的 key_buffer_size 设置过大，MySQL 根本使用不完。Key_Buffer_Read_HitRatio 也应该尽可能的高。如果该值较低，则很有可能是我们的 key_buffer_size 设置过小，需要适当增加 key_buffer_size 值，也有可能是 key_cache_age_threshold和key_cache_division_limit的设置不当，造成Key Cache cache失效太快。一般来说，在实际应用场景中，很少有人调整 key_cache_age_threshold 和 key_cache_division_limit 这两个参数的值，大都是使用系统的默认值。

多 Key Cache 的使用

从 MySQL4.1.1 版本开始，MyISAM 开始支持多个 Key Cache 并存的的功能。也就是说我们可以根据不同的需要设置多个 Key Cache 了，如将使用非常频繁而且基本不会被更新的表放入一个 Key Cache 中以防止在公共Key Cache 中被清除出去，而那些使用并不是很频繁而且可能会经常被更新的Key 放入另外一个 Key Cache 中。这样就可以避免出现某些场景下大批量的 Key 被读入 Key Cache 的时候，因为 Key Cache 空间问题使本来命中率很高的 Key 也不得不被清除出去。

MySQL 官方建议在比较繁忙的系统上一般可以设置三个 Key Cache：
一个 Hot Cache 使用 20%的大小用来存放使用非常频繁且更新很少的表的索引；
一个 Cold Cache 使用 20%的大小用来存放更新很频繁的表的索引；
一个 Warm Cache 使用剩下的 60%空间，作为整个系统默认的 Key Cache；

多个Key Cache 的具体使用方法在MySQL 官方手册中有比较详细的介绍，这里就不再累述了，有兴趣

的读者朋友可以自行查阅研究。

Key Cache 的 Mutex 问题

MySQL 索引缓存是所有线程共享的全局缓存，当多线程同时并发读取某一个 Cache Block 的时候并不会有任何问题，每个线程都可以同时读取该 Cache Block。但是当某个 Cache Block 正在被一个线程更新或者读入的时候，则该线程就会通过 mutex 锁定该 Cache Block 以达到不允许其他线程再同时更新或者读取。所以在高并发的环境下，如果 Key Cache 大小不够充足是非常容易因为 Cache Block 的 Mutex 问题造成严重的性能影响。而且在目前正式发行的所有 MySQL 版本中，Mutex 的处理机制存在一定的问题，使得当我们的 Active 线程数量稍微高一些的时候，就非常容易出现 Cache Block 的 Mutex 问题，甚至有人将此性能问题作为 Bug（#31551）报告给了 MySQL AB。

Key Cache 预加载

在 MySQL 中，为了让系统刚启动之后不至于因为 Cache 中没有任何数据而出现短时间的负载过高或者是响应不够及时的问题。MySQL 提供了 Key Cache 预加载功能，可以通过相关命令（LOAD INDEX INTO CACHE tb_name_list ...），将指定表的所有索引都加载到内存中，而且还可以通过相关参数控制是否只 Load 根结点和枝节点还是将页节点也全部 Load 进来，主要是为 Key Cache 的容量考虑。

对于这种启动后立即加载的操作，可以利用 MySQL 的 init_file 参数来设置相关的命令，如下：

```
mysql@sky:~$ cat /usr/local/mysql/etc/init.sql
SET GLOBAL hot_cache.key_buffer_size=16777216
SET GLOBAL cold_cache.key_buffer_size=16777216
CACHE INDEX example.top_message in hot_cache
CACHE INDEX example.event in cold_cache
LOAD INDEX INTO CACHE example.top_message,example.event IGNORE LEAVES
LOAD INDEX INTO CACHE example.user IGNORE LEAVES,exmple.groups
```

这里我的 init file 中首先设置了两个 Key Cache（hot cache 和 cold cache）各为 16M，然后分别将 top_message 这个变动很少的表的索引 Cache 到 Hot Cache，再将 event 这个变动非常频繁的表的索引 Cache 到了 Cold Cache 中，最后再通过 LOAD INDEX INTO CACHE 命令预加载了 top_message，groups 这两个表所有索引的所有节点以及 event 和 user 这两个表索引的非叶子节点数据到 Key Cache 中，以提高系统启动之初的响应能力。

NULL 值对统计信息的影响

虽然都是使用 B-Tree 索引，但是 MyISAM 索引和 Oracle 索引的处理方式不太一样，MyISAM 的索引中是会记录值为 NULL 的列信息的，只不过 NULL 值的索引键占用的空间非常少。所以，NULL 值的处理方式可能会影响到 MySQL 的查询优化器对执行计划的选择。所以 MySQL 就给我们提供了 myisam_stats_method 这个参数让我们可以自行决定对索引中的 NULL 值的处理方式。

myisam_stats_method 参数的作用就是让我们告诉 MyISAM 在收集统计信息的时候，是认为所有 NULL 值都是等同还是认为每个 NULL 值都认为是完全不相等的值，所以其可设置的值也为 nulls_unequal 和 nulls_equal。

当我们设置 myisam_stats_method = nulls_unequal，MyISAM 在搜集统计信息的时候会认为每个 NULL 值都不同，则基于该字段的索引的 Cardinality 就会更大，也就是说 MyISAM 会认为 DISTINCT 值数

量更多，这样就会让查询优化器处理 Query 的时候使用该索引的倾向性更高。

而当我们设置 myisam_stats_method = nulls_equal 之后，MyISAM 搜集统计信息的时候则会认为每个 NULL 值的都是一样的，这样 Cardinality 数值会降低，优化器选择执行计划的时候放弃该索引的倾向性会更高。

当然，上面所说的都是相对于使用等值查询的时候，而且 NULL 值占比较大的情况下，如果我们的 NULL 值本身就很少，那不管我们是使用 nulls_unequal 还是 nulls_equal，对优化器选择执行计划的影响是很小很小的。

表读取缓存优化

在 MySQL 中有两种读取数据文件的缓冲区，一种是 Sequential Scan 方式（如全表扫描）扫描表数据的时候使用，另一种则是在 Random Scan（如通过索引扫描）的时候使用。虽然这两种文件读取缓冲区并不是 MyISAM 存储引擎所特有的，但是由于 MyISAM 存储引擎并不会 Cache 数据（.MYD）文件，每次对数据文件的访问都需要通过调用文件系统的相关指令从磁盘上面读取物理文件。所以，每次读取数据文件需要使用的内存缓冲区的设置就对数据文件访问的性能非常重要了。在 MySQL 中对应这两种缓冲区的相关参数如下：

- read_buffer_size，以 Sequential Scan 方式扫描表数据时候使用的 Buffer；
  每个 Thread 进行 Sequential Scan 的时候都会产生该 Buffer，所以在设置的时候尽量不要太高，避免因为并发太大造成内存不够。系统默认为 128KB，最大为 2GB，设置的值必须是 4KB 的倍数，否则系统会自动更改成小于设置值的最大的 4KB 的倍数。
  一般来说，可以尝试适当调大此参数看是否能够改善全表扫描的性能。在不同的平台上可能会有不同的表现，这主要与 OS 级别的文件系统 IO 大小有关。所以该参数的设置最好是在真实环境上面通过多次更改测试调整，才能选找到一个最佳值。

- read_rnd_buffer_size，进行 Random Scan 的时候使用的 Buffer；
  read_rnd_buffer_size 所设置的 Buffer 实际上刚好和 read_buffer_size 所设置的 Buffer 相反，一个是顺序读的时候使用，一个是随机读的时候使用。但是两者都是针对于线程的设置，每个线程都可能产生两种 Buffer 中的任何一个。read_rnd_buffer_size 的默认值 256KB，最大值为 4G。
  一般来说，read_rnd_buffer_size 值的适当调大，对提高 ORDER BY 操作的性能有一定的效果。

这两个读取缓冲区都是线程独享的，每个线程在需要的时候都会创建一个（或者两个）系统中设置大小的缓冲区，所以在设置上面两个参数的时候一定不要过于激进，而应该根须系统可能的最大连接数和系统可用内存大小，计算出最大可设置值。

并发优化

在查询方面，MyISAM 存储引擎的并发并没有太大的问题，而且性能也非常的高。而且如果觉得光靠 Key Cache 来缓存索引还是不够快的话，我们还可以通过 Query Cache 功能来直接缓存 Query 的结果集。

但是，由于 MyISAM 存储引擎的表级锁定机制，以及读写互斥的问题，其并发写的性能一直是一个让人比较头疼的问题。一般来说，我们能做的主要也就只有以下几点了：

1. 打开 concurrent_insert 的功能，提高 INSERT 操作和 SELECT 之间的并发处理，使二者尽可能并行。大部分情况下 concurrent_insert 的值都被设置为 1，当表中没有删除记录留下的空余空间的时候都可以在尾部并行插入。这其实也是 MyISAM 的默认设置。如果我们的系统主要以写为主，尤其是

有大量的 INSERT 的时候。为了尽可能提高 INSERT 的效率，我们可以将 concurrent_insert 设置为 2，也就是告诉 MyISAM，不管在表中是否有删除行留下的空余空间，都在尾部进行并发插入，使 INSERT 和 SELECT 能够互不干扰。

2. 控制写入操作的大小，尽量让每次写入操作都能够很快的完成，以防止时间过程的阻塞动作。

3. 通过牺牲读取效率来提高写入效率。为了尽可能让写入更快，可以适当调整读和写的优先级别，让写入操作的优先级高于读操作的优先级。

对于一个表级锁定的存储引擎来说，除了 concurrent_insert 这个比较特殊的特性之外，可以说基本上都只能是串行的写。所以虽然上面给出了三点建议，但是后面两点也只能算是优化建议，并不是真正意义上的并发优化建议。

其他可以优化的地方

除了上面我们分析的这几个方面之外，MyISAM 实际上还存在其他一些可以优化的地方和一些常用的优化技巧。

1. 通过 OPTIMIZE 命令来整理 MyISAM 表的文件。这就像我们使用 Windows 操作系统会每过一段时间后都会做一次磁盘碎片整理，让系统中的文件尽量使用连续空间，提高文件的访问速度。MyISAM 在通过 OPTIMIZE 优化整理的时候，主要也是将因为数据删除和更新造成的碎片空间清理，使整个文件连续在一起。一般来说，在每次做了较大的数据删除操作之后都需要做一次 OPTIMIZE 操作。而且每个季度都应该有一次 OPTIMIZE 的维护操作。

2. 设置 myisam_max_[extra]_sort_file_size 足够大，对 REPAIR TABLE 的效率可能会有较大改善。

3. 在执行 CREATE INDEX 或者 REPAIR TABLE 等需要大的排序操作的之前可以通过调整 session 级别的 myisam_sort_buffer_size 参数值来提高排序操作的效率。

4. 通过打开 delay_key_write 功能，减少 IO 同步的操作，提高写入性能。

5. 通过调整 bulk_insert_buffer_size 来提高 INSERT...SELECT...这样的 bulk insert 操作的整体性能，LOAD DATA INFILE...的性能也可以得到改善。当然，在设置此参数的时候，也不应该一味的追求很大，很多时候过渡追求极端反而会影响系统整体性能，毕竟系统性能是从整体来看的，而不能仅仅针对某一个或者某一类操作。

11.2 Innodb 存储引擎优化

Innodb 存储引擎和 MyISAM 存储引擎最大区别主要有四点，第一点是缓存机制，第二点是事务支持，第三点是锁定实现，最后一点就是数据存储方式的差异。在整体性能表现方面，Innodb 和 MyISAM 两个存储引擎在不同的场景下差异比较大，主要原因也正是因为上面这四个主要区别所造成的。锁定相关的优化我们已经在“MySQL 数据库锁定机制”一章中做过相关的分析了，所以，本节关于 Innodb 存储引擎优化的分析，也将主要从其他三个方面展开。

11.2.1 Innodb 缓存相关优化

无论是对于哪一种数据库来说，缓存技术都是提高数据库性能的关键技术，物理磁盘的访问速度永远都会与内存的访问速度永远都不是一个数量级的。通过缓存技术无论是在读还是写方面都可以大大提高数据库整体性能。

Innodb_buffer_pool_size 的合理设置

Innodb 存储引擎的缓存机制和 MyISAM 的最大区别就在于 Innodb 不仅仅缓存索引，同时还会缓存实际的数据。所以，完全相同的数据库，使用 Innodb 存储引擎可以使用更多的内存来缓存数据库相关的信息，当然前提是要有足够的物理内存。这对于在现在这个内存价格不断降低的时代，无疑是个很吸引人的特性。

innodb_buffer_pool_size 参数用来设置 Innodb 最主要的 Buffer(Innodb_Buffer_Pool)的大小，也就是缓存用户表及索引数据的最主要缓存空间，对 Innodb 整体性能影响也最大。无论是 MySQL 官方手册还是网络上很多人所分享的 Innodb 优化建议，都简单的建议将 Innodb 的 Buffer Pool 设置为整个系统物理内存的 50% ～ 80% 之间。如此轻率的给出此类建议，我个人觉得实在是有些不妥。

不管是多么简单的参数，都可能与实际运行场景有很大的关系。完全相同的设置，不同的场景下的表现可能相差很大。就从 Innodb 的 Buffer Pool 到底该设置多大这个问题来看，我们首先需要确定的是这台主机是不是就只提供 MySQL 服务？MySQL 需要提供的的最大连接数是多少？MySQL 中是否还有 MyISAM 等其他存储引擎提供服务？如果有，其他存储引擎所需要使用的 Cache 需要多大？

假设是一台单独给 MySQL 使用的主机，物理内存总大小为 8G，MySQL 最大连接数为 500，同时还使用了 MyISAM 存储引擎，这时候我们的整体内存该如何分配呢？

内存分配为如下几大部分：

a) 系统使用，假设预留 800M；

b) 线程独享，约 2GB = 500 * (1MB + 1MB + 1MB + 512KB + 512KB)，组成大概如下：

sort_buffer_size：1MB

join_buffer_size：1MB

read_buffer_size：1MB

read_rnd_buffer_size：512KB

thread_statck：512KB

c) MyISAM Key Cache，假设大概为 1.5GB；

d) Innodb Buffer Pool 最大可用量：8GB - 800MB - 2GB - 1.5GB = 3.7GB；

假设这个时候我们还按照 50%～80%的建议来设置，最小也是 4GB，而通过上面的估算，最大可用值在 3.7GB 左右，那么很可能在系统负载很高当线程独享内存差不多出现极限情况的时候，系统很可能就会出现内存不足的问题了。而且上面还仅仅只是列出了一些使用内存较大的地方，如果进一步细化，很可能可用内存会更少。

上面只是一个简单的示例分析，实际情况并不一定是这样的，这里只是希望大家了解，在设置一些参数的时候，千万不要想当然，一定要详细的分析可能出现的情况，然后再通过不断测试调整来达到自己所处环境的最优配置。就我个人而言，正式环境上线之初，我一般都会采取相对保守的参数配置策略。上线之后，再根据实际情况和收集到的各种性能数据进行针对性的调整。

当系统上线之后，我们可以通过 Innodb 存储引擎提供给我们的关于 Buffer Pool 的实时状态信息作出进一步分析，来确定系统中 Innodb 的 Buffer Pool 使用情况是否正常高效：

```
sky@localhost : example 08:47:54> show status like 'Innodb_buffer_pool_%';
+---------------------------------+-------+
| Variable_name                   | Value |
+---------------------------------+-------+
| Innodb_buffer_pool_pages_data     | 70    |
| Innodb_buffer_pool_pages_dirty    | 0     |
| Innodb_buffer_pool_pages_flushed  | 0     |
| Innodb_buffer_pool_pages_free     | 1978  |
| Innodb_buffer_pool_pages_latched  | 0     |
| Innodb_buffer_pool_pages_misc     | 0     |
| Innodb_buffer_pool_pages_total    | 2048  |
| Innodb_buffer_pool_read_ahead_rnd | 1     |
| Innodb_buffer_pool_read_ahead_seq | 0     |
| Innodb_buffer_pool_read_requests  | 329   |
| Innodb_buffer_pool_reads          | 19    |
| Innodb_buffer_pool_wait_free      | 0     |
| Innodb_buffer_pool_write_requests | 0     |
+---------------------------------+-------+
```

从上面的值我们可以看出总共 2048 pages，还有 1978 是 Free 状态的仅仅只有 70 个 page 有数据，read 请求 329 次，其中有 19 次所请求的数据在 buffer pool 中没有，也就是说有 19 次是通过读取物理磁盘来读取数据的，所以很容易也就得出了 Innodb Buffer Pool 的 Read 命中率大概在为：(329 - 19) / 329 * 100% = 94.22%。

当然，通过上面的数据，我们还可以分析出 write 命中率，可以得到发生了多少次 read_ahead_rnd，多少次 read_ahead_seq，发生过多少次 latch，多少次因为 Buffer 空间大小不足而产生 wait_free 等等。

单从这里的数据来看，我们设置的 Buffer Pool 过大，仅仅使用 70 / 2048 * 100% = 3.4%。

在 Innodb Buffer Pool 中，还有一个非常重要的概念，叫做“预读”。一般来说，预读概念主要是在一些高端存储上面才会有，简单来说就是通过分析数据请求的特点来自动判断出客户在请求当前数据块之后可能会继续请求的数据快。通过该自动判断之后，存储引擎可能就会一次将当前请求的数据库和后面可能请求的下一个（或者几个）数据库一次全部读出，以期望通过这种方式减少磁盘 IO 次数提高 IO 性能。在上面列出的状态参数中就有两个专门针对预读：

Innodb_buffer_pool_read_ahead_rnd，记录进行随机读的时候产生的预读次数；

Innodb_buffer_pool_read_ahead_seq，记录连续读的时候产生的预读次数；

innodb_log_buffer_size 参数的使用

顾名思义，这个参数就是用来设置 Innodb 的 Log Buffer 大小的，系统默认值为 1MB。Log Buffer 的主要作用就是缓冲 Log 数据，提高写 Log 的 IO 性能。一般来说，如果你的系统不是写负载非常高且以大事务居多的话，8MB 以内的大小就完全足够了。

我们也可以通过系统状态参数提供的性能统计数据来分析 Log 的使用情况：

```
sky@localhost : example 10:11:05> show status like 'innodb_log%';
+---------------------------+-------+
| Variable_name             | Value |
+---------------------------+-------+
| Innodb_log_waits          | 0     |
| Innodb_log_write_requests | 6     |
| Innodb_log_writes         | 2     |
+---------------------------+-------+
```

通过这三个状态参数我们可以很清楚的看到 Log Buffer 的等待次数等性能状态。

当然，如果完全从 Log Buffer 本身来说，自然是大一些会减少更多的磁盘 IO。但是由于 Log 本身是为了保护数据安全而产生的，而 Log 从 Buffer 到磁盘的刷新频率和控制数据安全一致的事务直接相关，并且也有相关参数来控制（innodb_flush_log_at_trx_commit），所以关于 Log 相关的更详细的实现机制和优化在后面的“事务优化”中再做更详细的分析，这里就不展开了。

innodb_additional_mem_pool_size 参数理解

innodb_additional_mem_pool_size 所设置的是用于存放 Innodb 的字典信息和其他一些内部结构所需要的内存空间。所以我们的 Innodb 表越多，所需要的空间自然也就越大，系统默认值仅有 1MB。当然，如果 Innodb 实际运行过程中出现了实际需要的内存比设置值更大的时候，Innodb 也会继续通过 OS 来申请内存空间，并且会在 MySQL 的错误日志中记录一条相应的警告信息让我们知晓。

从我个人的经验来看，一个常规的几百个 Innodb 表的 MySQL，如果不是每个表都是上百个字段的话，20MB 内存已经足够了。当然，如果你有足够多的内存，完全可以继续增大这个值的设置。实际上，innodb_additional_mem_pool_size 参数对系统整体性能并无太大的影响，所以只要能存放需要的数据即可，设置超过实际所需的内存并没有太大意义，只是浪费内存而已。

Double Write Buffer

Double Write Buffer 是 Innodb 所使用的一种较为独特的文件 Flush 实现技术，主要做用是为了通过减少文件同步次数提高 IO 性能的情况下，提高系统 Crash 或者断电情况下数据的安全性，避免写入的数据不完整。

一般来说，Innodb 在将数据同步到数据文件进行持久化之前，首先会将需要同步的内容写入存在于

表空间中的系统保留的存储空间，也就是被我们称之为 Double Write Buffer 的地方，然后再将数据进行文件同步。所以实质上，Double Write Buffer 中就是存放了一份需要同步到文件中数据的一个备份，以便在遇到系统 Crash 或者主机断电的时候，能够校验最后一次文件同步是否准确的完成了，如果未完成，则可以通过这个备份来继续完成工作，保证数据的正确性。

那这样 Innodb 不是又一次增加了整体 IO 量了吗？这样不是可能会影响系统的性能么？这个完全不用太担心，因为 Double Write Buffer 是一块连续的磁盘空间，所有写入 Double Write Buffer 的操作都是连续的顺序写入操作，与整个同步过程相比，这点 IO 消耗所占的比例是非常小的。为了保证数据的准确性，这样一点点性能损失是完全可以接受的。

实际上，并不是所有的场景都需要使用 Double Write 这样的机制来保证数据的安全准确性，比如当我们使用某些特别文件系统的时候，如在 Solaris 平台上非常著名的 ZFS 文件系统，他就可以自己保证文件写入的完整性。而且在我们的 Slave 端，也可以禁用 Double Write 机制。

Adaptive Hash Index

在 Innodb 中，实现了一个自动监测各表索引的变化情况的机制，然后通过一系列的算法来判定如果存在一个 Hash Index 是否会对索引搜索带来性能改善。如果 Innodb 认为可以通过 Hash Index 来提高检索效率，他就会在内部自己建立一个基于某个 B-Tree 索引的 Hash Index，而且会根据该 B-Tree 索引的变化自行调整，这就是我们常说的 Adaptive Hash Index。当然，Innodb 并不一定会将整个 B-Tree 索引完全的转换为 Hash Index，可能仅仅只是取用该 B-Tree 索引键一定长度的前缀来构造一个 Hash Index。

Adaptive Hash Index 并不会进行持久化存放在磁盘上面，仅仅存在于 Buffer Pool 中。所以，在每次 MySQL 刚启动之后是并不存在 Adaptive Hash Index 的，只有在停工服务之后，Innodb 才会根据相应的请求来构建。

Adaptive Hash Index 的目的并不是为了改善磁盘 IO 的性能，而是为了提高 Buffer Pool 中的数据的访问效率，说的更浅显一点就是给 Buffer Pool 中的数据做的索引。所以，Innodb 在具有大容量内存（可以设置大的 Buffer Pool）的主机上，对于其他存储引擎来说，会存在一定的性能优势。

11.2.2 事务优化

选择合适的事务隔离级别

Innodb 存储引擎是 MySQL 中少有的支持事务的存储引擎之一，这也是其成为目前 MySQL 环境中使用最广泛存储引擎之一的一个重要原因。由于事务隔离的实现本身是需要消耗大量的内存和计算资源，而且不同的隔离级别所消耗的资源也不一样，性能表现也各不相同。所以我们

首先我们大概了解一下 Innodb 所支持的各种事务隔离级别。通过 Innodb 的参考手册，我们得到 Innodb 在事务隔离级别方面支持的信息如下：

1. READ UNCOMMITTED

常被成为 Dirty Reads（脏读），可以说是事务上的最低隔离级别：在普通的非锁定模式下 SELECT 的执行使我们看到的数据可能并不是查询发起时间点的数据，因而在这个隔离度下是非 Consistent Reads（一致性读）；

2. READ COMMITTED

这个事务隔离级别有些类似 Oracle 数据库默认的隔离级。属于语句级别的隔离，如通过 SELECT ... FOR UPDATE 和 SELECT ... LOCK IN SHARE MODE 来执行的请求仅仅锁定索引记录，而不锁定之前的间隙，因而允许在锁定的记录后自由地插入新记录。当然，这与 Innodb 的锁定实现机制有关。如果我们的 Query 可以很准确的通过索引定位到需要锁定的记录，则仅仅只需要锁定相关的索引记录，而不需要锁定该索引之前的间隙。但如果我们的 Query 通过索引检索的时候无法通过索引准确定位到需要锁定的记录，或者是一个基于范围的查询，InnoDB 就必须设置 next-key 或 gap locks 来阻塞其它用户对范围内的空隙插入。Consistent Reads 的实现机制与 Oracle 基本类似： 每一个 Consistent Read，甚至是同一个事务中的，均设置并作为它自己的最新快照。

这一隔离级别下，不会出现 Dirty Read，但是可能出现 Non-Repeatable Reads(不可重复读) 和 Phantom Reads（幻读）。

3. REPEATABLE READ

REPEATABLE READ 隔离级别是 InnoDB 默认的事务隔离级。SELECT ... FOR UPDATE, SELECT ... LOCK IN SHARE MODE, UPDATE, 和 DELETE ，这些以唯一条件搜索唯一索引的，只锁定所找到的索引记录，而不锁定该索引之前的间隙。 否则这些操作将使用 next-key 锁定，以 next-key 和 gap locks 锁定找到的索引范围，并阻塞其它用户的新建插入。在 Consistent Reads 中，与前一个隔离级相比这是一个重要的差别： 在这一级中，同一事务中所有的 Consistent Reads 均读取第一次读取时已确定的快照。这个约定就意味着如果在同一事务中发出几个无格式(plain)的 SELECTs ，这些 SELECT 的相互关系是一致的。

在 REPEATABLE READ 隔离级别下，不会出现 Dirty Reads，也不会出现 Non-Repeatable Reads，但是仍然存在 Phantom Reads 的可能性。

4. SERIALIZABLE

SERIALIZABLE 隔离级别是标准事务隔离级别中的最高级别。设置为 SERIALIZABLE 隔离级别之后，在事务中的任何时候所看到的数据都是事务启动时刻的状态，不论在这期间有没有其他事务已经修改了某些数据并提交。所以，SERIALIZABLE 事务隔离级别下，Phantom Reads 也不会出现。

以上四种事务隔离级别实际上就是 ANSI/ISO SQL92 标准所定义的四种隔离级别，Innodb 全部都为我们实现了。对于高并发应用来说，为了尽可能保证数据的一致性，避免并发可能带来的数据不一致问题，自然是事务隔离级别越高越好。但是，对于 Innodb 来说，所使用的事务隔离级别越高，实现复杂度自然就会更高，所需要做的事情也会更多，整体性能也就会更差。

所以，我们需要分析自己应用系统的逻辑，选择可以接受的最低事务隔离级别。以在保证数据安全一致性的同时达到最高的性能。

虽然 Innodb 存储引擎默认的事务隔离级别是 REPEATABLE READ，但实际上在我们大部分的应用场景下，都只需要 READ COMMITED 的事务隔离级别就可以满足需求了。

事务与 IO 的关系及优化

我想大部分人都清楚，Innodb 存储引擎通过缓存技术，将常用数据和索引缓存到内存中，这样我们在读取数据或者索引的时候就可以尽量减少物理 IO 来提高性能。那我们修改数据的时候 Innodb 是如何处理的呢，是否修改数据的时候 Innodb 是不是象我们常用的应用系统中的缓存一样，更改缓存中的数据的同时，将更改同时应用到相应的数据持久化系统中？

可能很多人都会有上面的这个疑问。实际上，Innodb 在修改数据的时候同样也只是修改 Buffer Pool 中的数据，并不是在一个事务提交的时候就将 BufferPool 中被修改的数据同步到磁盘，而是通过另外一种支持事务的数据库系统常用的手段，将修改信息记录到相应的事务日志中。

为什么不是直接将 Buffer Pool 中被修改的数据直接同步到磁盘，还有记录一个事务日志呢，这样不是反而增加了整体 IO 量了么？是的，对于系统的整体 IO 量而言，确实是有所增加。但是，对于系统的整体性能却有很大的帮助。

这里我们需要理解关于磁盘读写的两个概念：连续读写和随机读写。简单来说，磁盘的顺序读写就是将数据顺序的写入连续的物理位置，而随即读写则相反，数据需要根据各自的特定位置被写入各个位置，也就是被写入了并不连续的物理位置。对于磁盘来说，写入连续的位置最大的好处就是磁头所做的寻址动作很少，而磁盘操作中最耗费时间的就是磁头的寻址。所以，在磁盘操作中，连续读写操作比随即读写操作的性能要好很多。

我们的应用所修改的 Buffer Pool 中的数据都很随机，每次所做的修改都是一个或者少数几个数据页，多次修改的数据页也很少会连续。如果我们每次修改之后都将 Buffer Pool 中的数据同步到磁盘，那么磁盘就只能一直忙于频繁的随即读写操作。而事务日志在创建之初就是申请的连续的物理空间，而且每次写入都是紧接着之前的日志数据顺序的往后写入，基本上都是一个顺序的写入过程。所以，日志的写入操作远比同步 Buffer Pool 中被修改的数据要更快。

当然，由于事务日志都是通过几个日志文件轮循反复写入，而且每个日志文件大小固定，即使再多的日志也会有旧日志被新产生的日志覆盖的时候。所以，Buffer Pool 中的数据还是不可避免的需要被刷新到磁盘上进行持久化，而且这个持久化的动作必须在旧日志被新日志覆盖之前完成。只不过，随着被更新的数据（Dirty Buffer）的增加，需要刷新的数据的连续性就越高，所需要做的随机读写也就越少，自然，IO 性能也就得到了提升。

而且事务日志本身也有 Buffer（log buffer），每次事务日志的写入并不是直接写入到文件，也都是暂时先写入到 log buffer 中，然后再在一定的事件触发下才会同步到文件。当然，为了尽可能的减少事务日志的丢失，我们可以通过 innodb_log_buffer_size 参数来控制 log buffer 的大小。关于事务日志何时同步的说明稍后会做详细分析。

事务日志文件的大小与 Innodb 的整体 IO 性能有非常大的关系。理论上来讲，日志文件越大，则 Buffer Pool 所需要做的刷新动作也就越少，性能也越高。但是，我们也不能忽略另外一个事情，那就是当系统 Crash 之后的恢复。

事务日志的作用主要有两个，一个就是上面所提到的提高系统整体 IO 性能，另外一个就是当系统 Crash 之后的恢复。下面我们就来简单的分析一下当系统 Crash 之后，Innodb 是如何利用事务日志来进行数据恢复的。

Innodb 中记录了我们每一次对数据库中的数据及索引所做的修改，以及与修改相关的事务信息。同时还记录了系统每次 checkpoint 与 log sequence number（日志序列号）。

假设在某一时刻，我们的MySQL Crash了，那么很显然，所有Buffer Pool中的数据都会丢失，也包括已经修改且没有来得及刷新到数据文件中的数据。难道我们就让这些数据丢失么？当然不会，当MySQL 从 Crash 之后再次启动，Innodb会通过比较事务日志中所记录的checkpoint信息和各个数据文件中的checkpoint信息，找到最后一次checkpoint所对应的log sequence number，然后通过事务日志中所记录的变更记录，将从Crash之前最后一次checkpoint往后的所有变更重新应用一次，同步所有的数据文件到一致状态，这样就找回了因为系统Crash而造成的所有数据丢失。当然，对于log buffer中未来得及同步到日志文件的变更数据就无法再找回了。系统Crash的时间离最后一次checkpoint的时间越长，所需要的恢复时间也就越长。而日志文件越大，Innodb所做的checkpoint频率也越低，自然遇到长时间恢复的可能性也就越大了。

总的来说，Innodb的事务日志文件设置的越大，系统的IO性能也就越高，但是当遇到MySQL ，OS或者主机 Crash 的时候系统所需要的恢复时间也就越长；反之，日志越小，IO性能自然也就相对会差一些，但是当MySQL，OS 或者主机 Crash 之后所需要的恢复时间也越小。所以，到底该将事务日志设置多大其实是一个整体权衡的问题，既要考虑到系统整体的性能，又要兼顾到 Crash 之后的恢复时间。一般来说，在我个人维护的环境中，比较偏向于将事务日志设置为3组，每个日志设置为256MB大小，整体效果还算不错。

前面所描述的场景还只是MySQL Crash的场景，我们所丢失的仅仅只是Buffer Pool中的数据。实际上Innodb事务日志也不一定每次事务提交或者回滚都保证会同步log buffer中的数据到文件系统并通知文件系统做文件同步操作。所以当我们的OS Crash，或者是主机断点之后，事务日志写入文件系统Buffer中的数据还是可能会丢失，这种情况下，如果我们的事务日志没有及时同步文件系统刷新缓存中的数据到磁盘文件的话，就可能会产生日志数据丢失而造成数据永久性丢失的情况。

其实Innodb也早就考虑到了这种情况的存在，所以在系统中为我们设计了下面这个控制Innodb事务日志刷新方式的参数：innodb_flush_log_at_trx_commit。这个参数的主要功能就是让我们告诉系统，在什么情况下该通知文件系统刷新缓存中的数据到磁盘文件，可设置为如下三种值

- innodb_flush_log_at_trx_commit = 0，Innodb中的Log Thread没隔1秒钟会将log buffer中的数据写入到文件，同时还会通知文件系统进行文件同步的flush操作，保证数据确实已经写入到磁盘上面的物理文件。但是，每次事务的结束（commit或者是rollback）并不会触发Log Thread将log buffer中的数据写入文件。所以，当设置为0的时候，当 MySQL Crash 和OS Crash 或者主机断电之后，最极端的情况是丢失1秒时间的数据变更。
- innodb_flush_log_at_trx_commit = 1，这也是Innodb的默认设置。我们每次事务的结束都会触发Log Thread将log buffer中的数据写入文件并通知文件系统同步文件。这个设置是最安全的设置，能够保证不论是MySQL Crash 还是 OS Crash 或者是主机断电都不会丢失任何已经提交的数据。
- innodb_flush_log_at_trx_commit = 2，当我们设置为2的时候，Log Thread会在我们每次事务结束的时候将数据写入事务日志，但是这里的写入仅仅是调用了文件系统的文件写入操作。而我们的文件系统都是有缓存机制的，所以Log Thread的这个写入并不能保证内容真的已经写入到物理磁盘上面完成持久化的动作。文件系统什么时候会将缓存中的这个数据同步到物理磁

盘文件 Log Thread 就完全不知道了。所以，当设置为 2 的时候，MySQL Crash 并不会造成数据的丢失，但是 OS Crash 或者是主机断电后可能丢失的数据量就完全控制在文件系统上了。各种文件系统对于自己缓存的刷新机制各不相同，各位读者朋友如果有兴趣可以自行参阅相关的手册。

从上面的分析我们可以看出，当 innodb_flush_log_at_trx_commit 设置为 1 的时候是最安全的，但是由于所做的 IO 同步操作也最多，所以性能也是三种设置中最差的一种。如果设置为 0，则每秒有一次同步，性能相对高一些。如果设置为 2，可能性能是三这种最好的。但是也可能是出现鼓掌后丢失数据最多的。到底该如何设置设置，就要根据具体的场景来分析了。一般来说，如果完全不能接受数据的丢失，那么我们肯定会通过牺牲一定的性能来换取数据的安全性，选择设置为 1。而如果我们可以丢失很少量的数据(比如说 1 秒之内)，那么我们可以设置为 0。当然，如果大家觉得我们的 OS 足够稳定，主机硬件设备，而且主机的供电系统也足够安全，我们也可以将 innodb_flush_log_at_trx_commit 设置为 2 让系统的整体性能尽可能的高。

前面我们还提到了设置 Log Buffer 大小的参数 innodb_log_buffer_size。这里我们也简单的介绍一下 Log Buffer 的设置要领。Log Buffer 所存放的数据就是事务日志在写入文件之前在内存中的一个缓冲区域。所以理论上来讲，Log Buffer 越大，系统的性能也会越高。但是，由于触发 Log Thread 将 Log Buffer 中的数据写入文件的事件并不仅仅是 Log Buffer 空间用完的情况，还与 innodb_flush_log_at_trx_commit 参数的设置有关。如果该参数设置为 1 或者 2，那么我们的 Log Buffer 中仅仅只需要保存单个事务的变更量与系统最高并发事务的乘积。也就是说，如果我们的系统同时进行修改的并发事务最高为 20 的话，那么我们的 Log Buffer 就只需要存放 20 个事务所作的变更。当然，如果我们设置为 0 的话，Log Buffer 中所需要存放的数据则是 1 秒内所有的变更量。所以，大家需要根据自己系统的具体环境来针对性分析 innodb_log_buffer_size 的设置大小。一般来说，如果不是特别高的事务并发度或者系统中都是大事务的话，8MB 的内存空间已经完全够用了。

11.2.3 数据存储优化

从“MySQL 存储引擎简介”一章中我们已经对 Innodb 存储引擎的物理结构有了一定的了解，这一节我们将通过分析 Innodb 的物理文件结构寻找可以优化的线索。

理解 Innodb 数据及索引文件存储格式

Innodb 存储引擎的数据（包括索引）存放在相同的文件中，这一点和 MySQL 默认存储引擎 MyISAM 的区别较大，后者分别存放于独立的文件。除此之外，Innodb 的数据存放格式也比较独特，每个 Innodb 表都会将主键以聚簇索引的形式创建。所有的数据都是以主键来作为升序排列在物理磁盘上面，所以主键查询并且以主键排序的查询效率也会非常高。

由于主键是聚族索引的缘故，Innodb 的基于主键的查询效率非常高。如果我们在创建一个 Innodb 存储引擎的表的时候并没有创建主键，那么 Innodb 会尝试在创建于我们表上面的其他索引，如果存在由单个 not null 属性列的唯一索引，Innodb 则会选择该索引作为聚族索引。如果也没有任何单个 not null 属性列的唯一索引，Innodb 会自动生成一个隐藏的内部列，该列会在每行数据上占用 6 个字节的存储长度。所以，实质上每个 Innodb 表都之少会有一个索引存在。

在 Innodb 上面出了聚族索引之外的索引被成为 secondary index，每个 secondary index 上都会包含有聚族索引的索引键信息，方便通过其他索引查找数据的时候能够更快的定位数据位置所在。

当然，聚族索引也并不是只有好处没有任何问题，要不然其他所有数据库早就大力推广了。聚族索引的最大问题就是当索引键被更新的时候，所带来的成本并不仅仅只是索引数据可能会需要移动，而是相关的所有记录的数据都需要移动。所以，为了性能考虑，我们应该尽可能不要更新 Innodb 的主键值。

Page

Innodb 存储引擎中的所有数据，不论是表还是索引，亦或是存储引擎自己的各种结构，都是以 page 作为最小物理单位来存放，每个 page 默认大小为 16KB。

extent

extent 是一个由多个连续的 page 组成一个物理存储单位。一般来说，每个 extent 为 64 个 page。

segment

segment 在 Innodb 存储引擎中实际上也代表“files”的意思，每个 segment 由一个或多个 extent 组成，而且每个 segment 都存放同一种数据。一般来说，每个表数据会存放于一个单独的 segment 中，实际上也就是每个聚族索引会存放于一个单独的 segment 中。

tablespace

tablespace 是 Innodb 中最大物理结构单位了，由多个 segment 组成。

当 tablespace 中的某个 segment 需要增长的时候，Innodb 最初仅仅分配某一个 extent 的前 32 个 pages，然后如果继续增长才会分配整个 extent 来使用。我们还可以通过执行如下命令来查看 Innodb 表空间的使用情况：

```
sky@localhost : example 01:26:43> SHOW TABLE STATUS  like 'test'\G
*************************** 1. row ***************************
           Name: test
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 8389019
 Avg_row_length: 29
    Data_length: 249298944
Max_data_length: 0
   Index_length: 123387904
      Data_free: 0
 Auto_increment: NULL
    Create_time: 2008-11-15 01:26:43
    Update_time: NULL
     Check_time: NULL
      Collation: latin1_swedish_ci
       Checksum: NULL
```

```
 Create_options:
        Comment: InnoDB free: 5120 kB
```

通过上面的显示，我们可以看出

虽然每个索引页（index page）大小为 16KB，但是实际上 Innodb 在第一次使用该 page 的时候，如果是一个顺序的索引插入，都会预留 1KB 的空间。而如果是随机插入的话，那么大约会使用 (8- 15/16) KB 的空间，而如果一个 Index page 在进行多次删除之后如果所占用的空间已经低于 8KB（1/2）的话，Innodb 会通过一定的收缩机制收缩索引，并释放该 index page。此外，每个索引记录中都存放了一个 6 字节的头信息，主要用于行锁定时候的记录以及各个索引记录的关联信息。

Innodb 在存放数据页的时候不仅仅只是存放我们实际定义的列，同时还会增加两个内部隐藏列，其中一个隐含列的信息主要为事务相关信息，会占用 6 个字节的长度。另外一个则占用 7 字节长度，主要用来存放一个指向 Undo Log 中的 Undo Segment 的指针相关信息，主要用于事务回滚，以及通过 Undo Segment 中的信息构造多版本数据页。

通过上面的信息，我们至少可以得出以下几点对性能有较大影响的地方：

1. 为了尽量减小 secondary index 的大小，提高访问效率，作为主键的字段所占用的存储空间越小越好，最好是 INTEGER 类型。当然这并不是绝对的，字符串类型的数据同样也可以作为 Innodb 表的主键；
2. 创建表的时候尽量自己指定相应的主键，让数据按照自己预设的顺序排序存放，一提高特定条件下的访问效率；
3. 尽可能不要在主键上面进行更新操作，减少因为主键值的变化带来数据的移动。
4. 尽可能提供主键条件进行查询；

分散 IO 提升磁盘响应

由于 Innodb 和其他非事务存储引擎相比在记录数据文件的同时还记录有相应的事务日志（Transaction Log），相当于增加的整体的 IO 量，虽然事务日志是以完全顺序的方式写入磁盘，但总是会有一定的 IO 消耗，所以对于没有做 Raid 的磁盘系统来说，建议将数据文件和事务日志文件分别存放于不同的物理磁盘上面以降低磁盘的相互争用，提高整体 IO 性能。我们可以通过 innodb_log_group_home_dir 参数来指定 Innodb 日志存放位置，同时再通过设置数据文件位置 innodb_data_home_dir 参数来告诉 Innodb 我们希望将数据文件存放在哪里。

当然，如果我们使用独享表空间的话，Innodb 会为每个 Innodb 表创建一个表空间，并且会将该表空间存放在和“.frm”文件相同的路径下。不过幸运的是，Innodb 允许通过软链接的方式来访问数据或者日志文件。所以，如果我们有必要，甚至可以将每个表存放于单独的物理磁盘，然后再通过软链接的方式来告诉 Innodb 我们的实际文件在哪里。

当我们使用共享表空间的时候，最后一个数据文件必须是可以自动扩展的，这样就会带来一个疑问，在每次扩展的时候，到底该扩展多大空间性能会比较好呢？Innodb 给我们设计了 innodb_autoextend_increment 这个参数，让我们可以自行控制表空间文件每次增加的大小。

11.2.4 Innodb 其他优化

除了上面这些可以优化的地方之外，实际上 Innodb 还有其他一些可能影响到性能的参数设置：

- Innodb_flush_method

  用来设置 Innodb 打开和同步数据文件以及日志文件的方式，不过只有在 Linux & Unix 系统上面有效。系统默认值为 fdatasync，即 Innodb 默认通过 fsync()来 flush 数据和日志文件数据。

  此外，还可以设置为 O_DSYNC 和 O_DIRECT，当我们设置为 O_DSYNC，则系统以 O_SYNC 方式打开和刷新日志文件，通过 fsync()来打开和刷新数据文件。而设置为 O_DIRECT 的时候，则通过 O_DIRECT（Solaris 上为 directio()）打开数据文件，同时以 fsync()来刷新数据和日志文件。

  总的来说，innodb_flush_method 的不同设置主要影响的是 Innodb 在不同运行平台下进行 IO 操作的时候所调用的操作系统 IO 借口的区别。而不同的 IO 操作接口对数据的处理方式会有一定的区别，所以处理性能也会有一定的差异。一般来说，如果我们的磁盘是通过 RAID 卡做了硬件级别的 RAID，建议可以使用 O_DIRECT，可以一定程度上提高 IO 性能，但如果 RAID Cache 不够的话，还是需要谨慎对待。此外，根据 MySQL 官方手册上面的介绍，如果我们的存储环境是 SAN 环境，使用 O_DIRECT 有可能会反而使性能降低。对于支持 O_DSYNC 的平台，也可以尝试设置为 O_DSYNC 方式看是否能对写 IO 性能有所帮助。

- innodb_thread_concurrency

  这个参数主要控制 Innodb 内部的并发处理线程数量的最大值，系统内部会有相应的检测机制进行检测控制并发线程数量，Innodb 建议设置为 CPU 个数与磁盘个数之和。但是这个参数一直是一个非常有争议的参数，而且还有一个非常著名的 BUG（#15815）一直被认为就于 innodb_thread_concurrency 参数所控制的内容相关。从该参数在系统中的默认值的变化我们也可以看出即使是 Innodb 开发人员也并不是很清楚到底该将 innodb_thread_concurrency 设置为多少合适。在 MySQL5.0.8 之前，默认值为 8，从 MySQL5.0.8 开始到 MySQL5.0.18，默认值又被更改为 20，然后在 MySQL5.0.19 和 MySQL5.0.20 两个版本中又默认设置为 0。之后，从 MySQL5.0.21 开始默认值再次被更改回 8。

  innodb_thread_concurrency 参数的设置范围是 0～1000，但是在 MySQL5.0.19 之前的版本，只要该值超过 20，Innodb 就会认为不需要对并发线程数做任何限制，也就是说 Innodb 不会再进行并行线程的数目检查。同样，我们也可以通过设置为 0 来禁用并行线程检查，完全让 Innodb 自己根据实际需要创建并行线程，而且在不少场景下设置为 0 还是一个非常不错的选择，尤其是当系统写 IO 压力较大的时候。

  总的来说，innodb_thread_concurrency 参数的设置并没有一个很好的规则来判断什么场景该设置多大，完全需要通过不断的调整尝试，寻找出适合自己应用的设置。

- autocommit

  autocommit 的用途我想大家应该都很清楚，就是当我们将该参数设置为 true(1)之后，在我们每次执行完一条会修改数据的 Query 之后，系统内部都会自动提交该操作，基本上可以理解为屏蔽了事务的概念。

  设置 aotucommit 为 true（1）之后，我们的提交相对于自己手工控制 commit 时机来说可能会变得要频繁很多。这样带来的直接影响就是 Innodb 的事务日志可能会需要非常频繁的执行磁盘同

步操作，当然还与 innodb_flush_log_at_trx_commit 参数的设置相关。
一般来说，在我们通过 LOAD ... INFILE ... 或者其他的某种方式向 Innodb 存储引擎的表加载数据的时候，将 autocommit 设置为 false 可以极大的提高加载性能。而在正常的应用中，也最好尽量通过自行控制事务的提交避免过于频繁的日志刷新来保证性能。

11.2.5 Innodb 性能监控

我们可以通过执行“SHOW INNODB STATUS”命令来获取比较详细的系统当前 Innodb 性能状态，如下：

```
    sky@localhost : example 03:11:19> show innodb status\G
*************************** 1. row ***************************
Status:
=====================================
081115 15:56:30 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 10 seconds
----------
SEMAPHORES
----------
OS WAIT ARRAY INFO: reservation count 720, signal count 719
Mutex spin waits 0, rounds 16962, OS waits 460
RW-shared spins 489, OS waits 244; RW-excl spins 3, OS waits 3
------------
TRANSACTIONS
------------
Trx id counter 0 11605
Purge done for trx's n:o < 0 11604 undo n:o < 0 0
History list length 10
Total number of lock structs in row lock hash table 0
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 0 0, not started, process no 13383, OS thread id 2892274576
MySQL thread id 9, query id 54 localhost sky
show innodb status
--------
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
1123 OS file reads, 2791 OS file writes, 1941 OS fsyncs
0.00 reads/s, 0 avg bytes/read, 0.00 writes/s, 0.00 fsyncs/s
-------------------------------------
INSERT BUFFER AND ADAPTIVE HASH INDEX
-------------------------------------
Ibuf: size 1, free list len 0, seg size 2,
0 inserts, 0 merged recs, 0 merges
Hash table size 138401, used cells 2, node heap has 1 buffer(s)
0.00 hash searches/s, 0.00 non-hash searches/s
---
LOG
---
Log sequence number 0 1072999334
Log flushed up to   0 1072999334
Last checkpoint at  0 1072999334
0 pending log writes, 0 pending chkp writes
1301 log i/o's done, 0.00 log i/o's/second
----------------------
BUFFER POOL AND MEMORY
----------------------
Total memory allocated 58787017; in additional pool allocated 1423616
Buffer pool size   2048
Free buffers       803
Database pages     1244
Modified db pages  0
Pending reads 0
Pending writes: LRU 0, flush list 0, single page 0
Pages read 15923, created 22692, written 23332
0.00 reads/s, 0.00 creates/s, 0.00 writes/s
No buffer pool page gets since the last printout
--------------
ROW OPERATIONS
--------------
0 queries inside InnoDB, 0 queries in queue
1 read views open inside InnoDB
Main thread process no. 13383, id 2966408080, state: waiting for server activity
Number of rows inserted 8388614, updated 0, deleted 0, read 8388608
0.00 inserts/s, 0.00 updates/s, 0.00 deletes/s, 0.00 reads/s
----------------------------
END OF INNODB MONITOR OUTPUT
============================
```

通过上面的输出，我们可以看到整个信息分为 7 个部分，分别说明如下：

- SEMAPHORES，这部分主要显示系统中当前的信号等待信息以及各种等待信号的统计信息，这部分输出的信息对于我们调整 innodb_thread_concurrency 参数有非常大的帮助，当等待信号量非常大的时候，可能就需要禁用并发线程检测设置 innodb_thread_concurrency=0；
- TRANSACTIONS，这里主要展示系统的锁等待信息和当前活动事务信息。通过这部分输出，我们可以查追踪到死锁的详细信息；
- FILE I/O，文件 IO 相关的信息，主要是 IO 等待信息；
- INSERT BUFFER AND ADAPTIVE HASH INDEX；显示插入缓存当前状态信息以及自适应 Hash Index 的状态；
- LOG，Innodb 事务日志相关信息，包括当前的日志序列号（Log Sequence Number），已经刷新同步到哪个序列号，最近的 Check Point 到哪个序列号了。除此之外，还显示了系统从启动到现在已经做了多少次 Ckeck Point，多少次日志刷新；
- BUFFER POOL AND MEMORY，这部分主要显示 Innodb Buffer Pool 相关的各种统计信息，以及其他一些内存使用的信息；
- ROW OPERATIONS，顾名思义，主要显示的是与客户端的请求 Query 和这些 Query 所影响的记录统计信息。

这里只是对输出做了一个简单的介绍，如果各位读者朋友希望更深入的了解相应的细节，建议查阅 Innodb 相关手册，此外，《High Performance MySQL》作者之一 Peter Zaitsev 有一篇叫做“SHOW INNODB STATUS walk through”的文件专门做了较为详细的分析，大家可以通过访问 http://www.mysqlperformanceblog.com/网址去了解。

当然，如果我们总是要通过不断执行“SHOW INNODB STATUS”命令来获取这样的性能信息是在是有些麻烦，所以 Innodb 存储引擎为我们设计了一个比较奇怪的方式来持续获取该信息并输出到 MySQL Error Log 中。

实现方式就是通过创建一个名为 innodb_monitor，存储引擎为 Innodb 的表，够奇特吧，如下：

```
CREATE TABLE innodb_monitor(a int) ENGINE=INNODB;
```

当我们创建这样一个表之后，Innodb 就会每过 15 秒输出一次 Innodb 整体状态信息，也就是上面所展示的信息到 Error Log 中。我们可以通过删除该表停止该 Monitor 功能，如下：

```
DROP TABLE innodb_monitor;
```

除此之外，我们还可以通过相同的方式打开和关闭 innodb_tablespace_monitor,innodb_lock_monitor,innodb_table_monitor这三种监控功能，各位读者朋友可以自行尝试。

通过上面的各种监控信息的输出信息，我们可以比较详细的了解到 Innodb 当前的运行状态，帮助我们及时发现性能问题。

11.3 小结

MyISAM 和 Innodb 两种存储引擎各有特点，很多使用者对这两种存储引擎各有偏好，认为某一种要优于另外一种，实际上这是比较片面的认识。两种存储引擎各自都存在对方没有的优点，也存在自身的缺点，我们只有充分了解了各自的优缺点之后，在实际应用环境中根据不同的需要选择不同的存储引擎，才能将 MySQL 用到最好。

此外，随着 MySQL Cluster 的不断成熟，除了上面详细分析的两种存储引擎之外，实际上还有 NDB Cluster 存储引擎正在被越来越多的使用，关于 NDB Cluster 相关的内容，将在架构设计篇中再进行比较详细的介绍。