1. 商业需求对性能的影响

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。

2. 系统架构及实现对性能的影响

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255Bytes 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。

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？

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GROUPBY 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。

过度依赖数据库SQL 语句的功能造成数据库操作效率低下

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

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？

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。

3. Query 语句对系统性能的影响

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。

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

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。

4. Schema 设计对系统的性能影响

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

5. 硬件环境对系统性能的影响

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。

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%