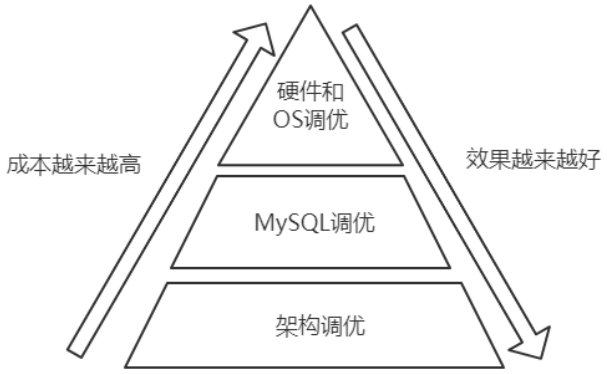
# 从架构师角度全局理解Mysql性能优化

## 总论

有道云链接：http://note.youdao.com/noteshare?id=f1d34fba6f218de1acf9160f45c284af&sub=9B48B919EE3C4046BC4CD564AA325A72

MySQL性能优化其实是个很大的课题，在优化上存在着一个调优金字塔的说法：




很明显从图上可以看出，越往上走，难度越来越高，收益却是越来越小的。比如硬件和OS调优，需要对硬件和OS有着非常深刻的了解，仅仅就磁盘一项来说，一般非DBA能想到的调整就是SSD盘比用机械硬盘更好，但其实它至少包括了，使用什么样的磁盘阵列（RAID）级别、是否可以分散磁盘IO、是否使用裸设备存放数据，使用哪种文件系统（Linux下常见的有ext2、3、4和xfs,zfs等，目前比较推荐的是XFS），操作系统的磁盘调度算法（目前比较推荐deadline，对机械硬盘和SSD都比较合适。从内核2.5开始，默认的I/O调度算法是Deadline，之后默认I/O调度算法为Anticipatory，直到内核2.6.17为止，从内核2.6.18开始，CFQ成为默认的IO调度算法，但CFQ并不推荐作为数据库服务器的磁盘调度算法）的选择，是否需要调整操作系统文件管理方面等等。

*TIPS：裸设备(raw device)，也叫裸分区（原始分区），是一种没有经过格式化，不被Unix通过文件系统来读取的特殊块设备文件。由应用程序负责对它进行读写操作。不经过文件系统的缓冲。它是不被操作系统直接管理的设备。这种设备少了操作系统这一层，I/O效率更高。*

*TIPS：小演示，查看磁盘调度算法：*

```
# dmesg |grep -i scheduler
# df -m
# more /sys/block/vda/queue/scheduler
```

*永久地修改IO调度算法,需要修改内核引导参数*

```
[root@iZwz9j203ithc4gu1uvb2wZ ~]# dmesg |grep -i scheduler
[    0.777484] io scheduler noop registered
[    0.777487] io scheduler deadline registered (default)
[    0.777507] io scheduler cfq registered
[    0.777511] io scheduler mq-deadline registered
[    0.777514] io scheduler kyber registered
[root@iZwz9j203ithc4gu1uvb2wZ ~]# df -m
Filesystem     1M-blocks  Used Available Use% Mounted on
/dev/vda1          40184  8350     29975  22% /
devtmpfs             909     0       909   0% /dev
tmpfs                920     0       920   0% /dev/shm
tmpfs                920     1       919   1% /run
tmpfs                920     0       920   0% /sys/fs/cgroup
tmpfs                184     0       184   0% /run/user/1000
tmpfs                184     0       184   0% /run/user/0
[root@iZwz9j203ithc4gu1uvb2wZ ~]# more /sys/block/vda/queue/scheduler
[mq-deadline] kyber none
```

所以在进行优化时，首先需要关注和优化的应该是架构，如果架构不合理，即使是DBA能做的事情其实是也是比较有限的。

对于架构调优，在系统设计时首先需要充分考虑业务的实际情况，是否可以把不适合数据库做的事情放到数据仓库、搜索引擎或者缓存中去做；然后考虑写的并发量有多大，是否需要采用分布式；最后考虑读的压力是否很大，是否需要读写分离。对于核心应用或者金融类的应用，需要额外考虑数据安全因素，数据是否不允许丢失。

作为金字塔的底部的架构调优，采用更适合业务场景的架构能最大程度地提升系统的扩展性和可用性。在设计中进行垂直拆分能尽量解耦应用的依赖，对读压力比较大的业务进行读写分离能保证读性能线性扩展，而对于读写并发压力比较大的业务在MySQL上也有采用读写分离的大量案例。

在底层硬件系统、SQL语句和参数都基本定型的情况下，单个MySQL数据库能提供的性能、扩展性等就基本定型了。但是通过架构设计和优化，却能承载几倍、几十倍甚至百倍于单个MySQL数据库能力的业务请求能力。

对于MySQL调优，需要确认业务表结构设计是否合理，SQL语句优化是否足够，该添加的索引是否都添加了，是否可以剔除多余的索引等等。

最后确定系统、硬件有哪些地方需要优化，系统瓶颈在哪里，哪些系统参数需要调整优化，进程资源限制是否提到足够高；在硬件方面是否需要更换为具有更高I/O性能的存储硬件，是否需要升级内存、CPU、网络等。

如果在设计之初架构就不合理，比如没有进行读写分离，那么后期的MySQL和硬件、系统优化的成本就会很高，并且还不一定能最终解决问题。如果业务性能的瓶颈是由于索引等MySQL层的优化不够导致的，那么即使配置再高性能的I/O存储硬件或者CPU也无法支撑业务的全表扫描。

当然我们本课程重点关注的是MySQL方面的调优，特别是索引。SQL/索引调优要求对业务和数据流非常清楚。在阿里巴巴内部，有三分之二的DBA是业务DBA，从业务需求讨论到表结构审核、SQL语句审核、上线、索引更新、版本迭代升级，甚至哪些数据应该放

到非关系型数据库中，哪些数据放到数据仓库、搜索引擎或者缓存中，都需要这些DBA跟踪和复审。他们甚至可以称为数据架构师（Data Architecher）。

# 查询性能优化

前面的章节我们知道如何设计最优的库表结构、如何建立最好的索引，这些对于高性能来说是必不可少的。但这些还不够—还需要合理的设计查询。如果查询写得很糟糕，即使库表结构再合理、索引再合适，也无法实现高性能。

## 什么是慢查询

慢查询日志，顾名思义，就是查询花费大量时间的日志，是指mysql记录所有执行超过long_query_time参数设定的时间阈值的SQL语句的日志。该日志能为SQL语句的优化带来很好的帮助。默认情况下，慢查询日志是关闭的，要使用慢查询日志功能，首先要开启慢查询日志功能。

## 慢查询配置

我们已经知道慢查询日志可以帮助定位可能存在问题的SQL语句，从而进行SQL语句层面的优化。但是默认值为关闭的，需要我们手动开启。

show VARIABLES like '%slow_query_log%';

```
mysql> show VARIABLES like 'slow_query_log';
+----------------+-------+
| Variable_name  | Value |
+----------------+-------+
| slow_query_log | OFF   |
+----------------+-------+
1 row in set (0.00 sec)
```

开启：

set GLOBAL slow_query_log=1;

```
mysql> set GLOBAL  slow_query_log = 1;
Query OK, 0 rows affected (0.00 sec)

mysql> show VARIABLES like 'slow_query_log';
+----------------+-------+
| Variable_name  | Value |
+----------------+-------+
| slow_query_log | ON    |
+----------------+-------+
1 row in set (0.00 sec)
```

但是多慢算慢？MySQL中可以设定一个阈值，将运行时间超过该值的所有SQL语句都记录到慢查询日志中。long_query_time参数就是这个阈值。默认值为10，代表10秒。

show VARIABLES like '%long_query_time%';

当然也可以设置

set global long_query_time=0;　　---默认10秒，这里为了演示方便设置为0

```
mysql> show VARIABLES like '%long_query_time%';
+-----------------+-----------+
| Variable_name   | Value     |
+-----------------+-----------+
| long_query_time | 10.000000 |
+-----------------+-----------+
1 row in set (0.00 sec)
```

同时对于没有运行的SQL语句没有使用索引，则MySQL数据库也可以将这条SQL语句记录到慢查询日志文件，控制参数是：

show VARIABLES like '%log_queries_not_using_indexes%';

```
mysql> show VARIABLES like '%log_queries_not_using_indexes%';
+-------------------------------+-------+
| Variable_name                 | Value |
+-------------------------------+-------+
| log_queries_not_using_indexes | OFF   |
+-------------------------------+-------+
1 row in set (0.00 sec)
```

对于产生的慢查询日志，可以指定输出的位置，通过参数log_output来控制，可以输出到[TABLE][FILE][FILE,TABLE]。比如

set global log_output='FILE,TABLE'，缺省是输出到文件，我们的配置把慢查询输出到表，不过一般不推荐输出到表。

show VARIABLES like 'log_output';ls

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

## 慢查询解读分析

### *日志格式*

开启慢查询功能以后，会根据我们的配置产生慢查询日志

```
[mysql@iZwz9j203ithc4gu1uvb2wZ data]$ cat iZwz9j203ithc4gu1uvb2wZ-slow.log
/home/mysql/mysql57/bin/mysqld, Version: 5.7.32 (MySQL Community Server (GPL
Tcp port: 3306  Unix socket: /home/mysql/mysql57/log/mysql.sock
Time                 Id Command    Argument
# Time: 2021-04-05T07:50:23.580909Z
# User@Host: root[root] @ localhost []  Id:     3
# Query_time: 0.000103  Lock_time: 0.000000 Rows_sent: 1  Rows_examined: 0
SET timestamp=1617609023;
select @@version_comment limit 1;
# Time: 2021-04-05T07:50:49.898353Z
# User@Host: root[root] @ localhost []  Id:     3
# Query_time: 0.000103  Lock_time: 0.000000 Rows_sent: 1  Rows_examined: 0
SET timestamp=1617609049;
SELECT DATABASE();
# Time: 2021-04-05T07:50:49.898603Z
# User@Host: root[root] @ localhost []  Id:     3
# Query_time: 0.000085  Lock_time: 0.000000 Rows_sent: 1  Rows_examined: 0
use mysqladv;
SET timestamp=1617609049;
```

从慢查询日志里面摘选一条慢查询日志，数据组成如下

“Time: 2021-04-05T07:50:53.243703Z”：查询执行时间

“User@Host: root[root] @ localhost []  Id:  3”：用户名 、用户的IP信息、线程ID号

“Query_time: 0.000495”：执行花费的时长【单位：秒】

“Lock_time: 0.000170”：执行获得锁的时长

“Rows_sent”：获得的结果行数

“Rows_examined”：扫描的数据行数

“SET timestamp”：这SQL执行的具体时间

最后一行：执行的SQL语句

### *慢查询分析*

慢查询的日志记录非常多，要从里面找寻一条查询慢的日志并不是很容易的事情，一般来说都需要一些工具辅助才能快速定位到需要优化的SQL语句，下面介绍两个慢查询辅助工具

#### mysqldumpslow

常用的慢查询日志分析工具，汇总除查询条件外其他完全相同的SQL，并将分析结果按照参数中所指定的顺序输出。当然它的参数不少，我们常用的也就是那么几个。

**语法：**

mysqldumpslow -s r -t 10 slow-mysql.log

-s order (c,t,l,r,at,al,ar)

c:总次数

t:总时间

l:锁的时间

r:获得的结果行数

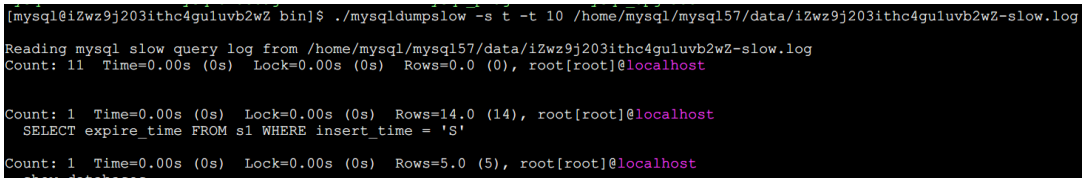
at,al,ar :指t,l,r平均数  【例如：at = 总时间/总次数】

-s 对结果进行排序，怎么排，根据后面所带的 (c,t,l,r,at,al,ar)，缺省为at

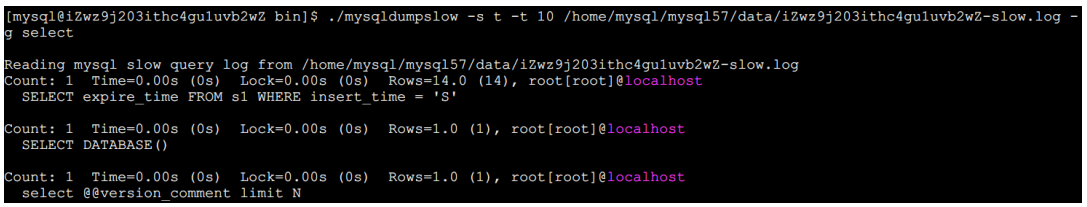
-t  NUM  just show the top n queries：仅显示前n条查询

-g PATTERN  grep: only consider stmts that include this string：通过grep来筛选语句。

./mysqldumpslow -s t -t 10 /home/mysql/mysql57/data/iZwz9j203ithc4gu1uvb2wZ-slow.log



./mysqldumpslow -s t -t 10 /home/mysql/mysql57/data/iZwz9j203ithc4gu1uvb2wZ-slow.log -g select




## 优化SQL查询方法论

查询性能低下最基本的原因是访问的数据太多。大部分性能低下的查询都可以通过减少访问的数据量的方式进行优化。对于低效的查询，一般通过下面两个步骤来分析总是很有效：

1．确认应用程序是否在检索大量超过需要的数据。这通常意味着访问了太多的行，但有时候也可能是访问了太多的列。

2．确认MySQL服务器层是否在分析大量超过需要的数据行。

### *业务层-请求了不需要的数据?*

有些查询会请求超过实际需要的数据，然后这些多余的数据会被应用程序丢弃。这会给MySQL服务器带来额外的负担，并增加网络开销，另外也会消耗应用服务器的CPU和内存资源。比如：

#### 查询不需要的记录

例如先使用SELECT语句查询大量的结果，然后获取前面的N行后关闭结果集（例如取出100条记录，但是只是在页面上显示前面10条）。

#### 总是取出全部列

每次看到SELECT*的时候都需要用怀疑的眼光审视，是不是真的需要返回全部的列？很可能不是必需的。取出全部列，会让优化器无法完成索引覆盖扫描这类优化,还会为服务器带来额外的I/O、内存和CPU的消耗。因此，一些DBA是严格禁止SELECT *的写法的，这样做有时候还能避免某些列被修改带来的问题。

什么时候应该允许查询返回超过需要的数据？如果这种有点浪费数据库资源的方式可以简化开发，因为能提高相同代码片段的复用性，如果清楚这样做的性能影响，那么这种做法也是值得考虑的。

或者如果应用程序使用了某种缓存机制，或者有其他考虑，获取超过需要的数据也可能有其好处，但不要忘记这样做的代价是什么。获取并缓存所有的列的查询，相比多个独立的只获取部分列的查询可能就更有好处。

#### 重复查询相同的数据

不断地重复执行相同的查询，然后每次都返回完全相同的数据。比较好的方案是，当初次查询的时候将这个数据缓存起来，需要的时候从缓存中取出，这样性能显然会更好。

### *执行层-是否在扫描额外的记录*

在确定查询只返回需要的数据以后，接下来应该看看查询为了返回结果是否扫描了过多的数据。对于MySQL，最简单的衡量查询开销的三个指标如下:

**响应时间、扫描的行数、返回的行数**

这三个指标都会记录到MySQL的慢日志中，所以检查慢日志记录是找出扫描行数过多的查询的好办法。

### 响应时间

响应时间是两个部分之和:服务时间和排队时间。

服务时间是指数据库处理这个查询真正花了多长时间。

排队时间是指服务器因为等待某些资源而没有真正执行查询的时间——可能是等I/O操作完成，也可能是等待行锁，等等。

当你看到一个查询的响应时间的时候，首先需要问问自己，这个响应时间是否是一个合理的值。从我们的前面课程中，我们知道如何分析一个SQL查询需要哪些索引以及它的执行计划是什么,然后计算大概需要读取多少个页面和记录数，是能够大致分析出当前响应时间是不是一个合理的值。

### 扫描的行数和返回的行数

分析查询时，查看该查询扫描的行数是非常有帮助的。这在一定程度上能够说明该查询找到需要的数据的效率高不高。

理想情况下扫描的行数和返回的行数应该是相同的。但实际情况中这种“美事”并不多。

例如不正确的使用Limit，在系统中需要进行分页操作的时候，我们通常会使用LIMIT加上偏移量的办法实现，同时加上合适的ORDER BY子句。

在偏移量非常大的时候，SQL语句就变成了类似`select * from order_exp limit 10000,10;`

这样的查询，这时MySQL需要查询10010条记录然后只返回最后10条，前面10 000条记录都将被抛弃，这样的代价非常高。在第四期MySQL《05-VIP-Mysql索引优化实战二》中专门讲过“分页查询优化”，请自行参考。

又或者在做一个关联查询时，服务器必须要扫描多行才能生成结果集中的一行。扫描的行数对返回的行数的比率通常很小，一般在1:1和10:1之间，不过有时候这个值也可能非常非常大。

### 扫描的行数和访问类型

在评估查询开销的时候，需要考虑一下从表中找到某一行数据的成本。MySQL有好几种访问方式可以查找并返回一行结果。有些访问方式可能需要扫描很多行才能返回一行结果，也有些访问方式可能无须扫描就能返回结果。

在EXPLAIN语句中的type列反应了访问类型。访问类型有很多种，从全表扫描到索引扫描、范围扫描、唯一索引查询、常数引用等，速度是从慢到快，扫描的行数也是从小到大。对其中相关的扫描表、扫描索引、范围访问和单值访问的概念要非常熟悉。

如果查询没有办法找到合适的访问类型，那么解决的最好办法通常就是增加一个合适的索引，为什么索引对于查询优化如此重要了。索引让 MySQL以最高效、扫描行数最少的方式找到需要的记录。

对于我们在SQL语句中常见的WHERE条件，一般 MySQL能够使用如下三种方式应用WHERE条件，从效率和扫描行数多少来评价的话，从好到坏依次为:

1、在索引中使用WHERE条件来过滤不匹配的记录。这是在存储引擎层完成的。

2、使用索引覆盖扫描（在Extra列中出现了Using index）来返回记录，直接从索引中过滤不需要的记录并返回命中的结果。这是在 MySQL服务器层完成的，但无须再回表查询记录。

3、从数据表中返回数据，然后过滤不满足条件的记录（在Extra列中出现Using Where)。这在 MySQL服务器层完成，MySQL需要先从数据表读出记录然后过滤。

好的索引可以让查询使用合适的访问类型，尽可能地只扫描需要的数据行。

如果发现查询需要扫描大量的数据但只返回少数的行，那么通常可以尝试下面的技巧去优化它:

1、使用索引覆盖扫描，把所有需要用的列都放到索引中，这样存储引擎无须回表获取对应行就可以返回结果了（在前面的章节中我们已经讨论过了)。

2、改变库表结构。例如使用单独的汇总表。

3、重写这个复杂的查询，让 MySQL优化器能够以更优化的方式执行这个查询。

## 重构SQL查询的方法论

在优化有问题的查询时，目标应该是找到一个更优的方法获得实际需要的结果——而不一定总是需要从MySQL获取一模一样的结果集。有时候，可以将查询转换一种写法让其返回一样的结果,但是性能更好。但也可以通过修改应用代码,用另一种方式完成查询，最终达到一样的目的。

### *一个复杂查询还是多个简单查询*

设计查询的时候一个需要考虑的重要问题是，是否需要将一个复杂的查询分成多个简单的查询。在传统实现中，总是强调需要数据库层完成尽可能多的工作，这样做的逻辑在于以前总是认为网络通信、查询解析和优化是一件代价很高的事情。但是这样的想法对于MySQL并不适用，MySQL 从设计上让连接和断开连接都很轻量级，在返回一个小的查询结果方面很高效。现代的网络速度比以前要快很多，无论是带宽还是延迟，即使是一个千兆网卡也能轻松满足每秒超过2000次的查询。

所以有时候，可以将一个复杂的大查询分解为多个小查询会效率更高。

不过，在应用设计的时候，如果一个查询能够胜任时还写成多个独立查询是不明智的。例如，应用对一个数据表做10次独立的查询来返回10行数据，每个查询返回一条结果，查询10次。

### *切分查询*

有时候对于一个大查询我们需要“分而治之”，将大查询切分成小查询，每个查询功能完全一样，只完成一小部分，每次只返回一小部分查询结果。

删除旧的数据就是一个很好的例子。定期地清除大量数据时，如果用一个大的语句一次性完成的话，则可能需要一次锁住很多数据、占满整个事务日志、耗尽系统资源、阻塞很多小的但重要的查询。将一个大的DELETE语句切分成多个较小的查询可以尽可能小地影响MySQL性能，同时还可以减少MySQL复制的延迟。

一般来说，将每个SQL所处理的记录控制在5000到10000是个比较好的权衡值。

### *分解关联查询*

很多高性能的应用都会对关联查询进行分解。简单地，可以对每一个表进行一次单表查询，然后将结果在应用程序中进行关联。到底为什么要这样做?乍一看，这样做并没有什么好处，原本一条查询，这里却变成多条查询，返回的结果又是一模一样的。事实上，用分解关联查询的方式重构查询有如下的优势:

**让缓存的效率更高。**许多应用程序可以方便地缓存单表查询对应的结果对象。将查询分解后，执行单个查询**可以减少锁的竞争**。

在应用层做关联，可以更容易对数据库进行拆分，**更容易做到高性能和可扩展**。查询本身效率也可能会有所提升。

可以**减少冗余记录的查询**。在应用层做关联查询，意味着对于某条记录应用只需要查询一次,而在数据库中做关联查询,则可能需要重复地访问一部分数据。从这点看，这样的重构还可能会减少网络和内存的消耗。

更进一步，这样做相当于**在应用中实现了哈希关联**，而不是使用MySQL的嵌套循环关联。某些场景哈希关联的效率要高很多。

在很多场景下，通过重构查询将关联放到应用程序中将会更加高效，这样的场景有很多，比如:当应用能够方便地缓存单个查询的结果的时候、当可以将数据分布到不同的MySQL服务器上的时候、当能够使用IN()的方式代替关联查询的时候、当查询中使用同一个数据表的时候。

## 从MySQL执行全流程考虑性能优化

### *为什么查询速度会慢*

如果把查询看作是一个任务，那么它由一系列子任务组成，每个子任务都会消耗一定的时间。如果要优化查询，实际上要优化其子任务，要么消除其中一些子任务，要么减少子任务的执行次数，要么让子任务运行得更快。

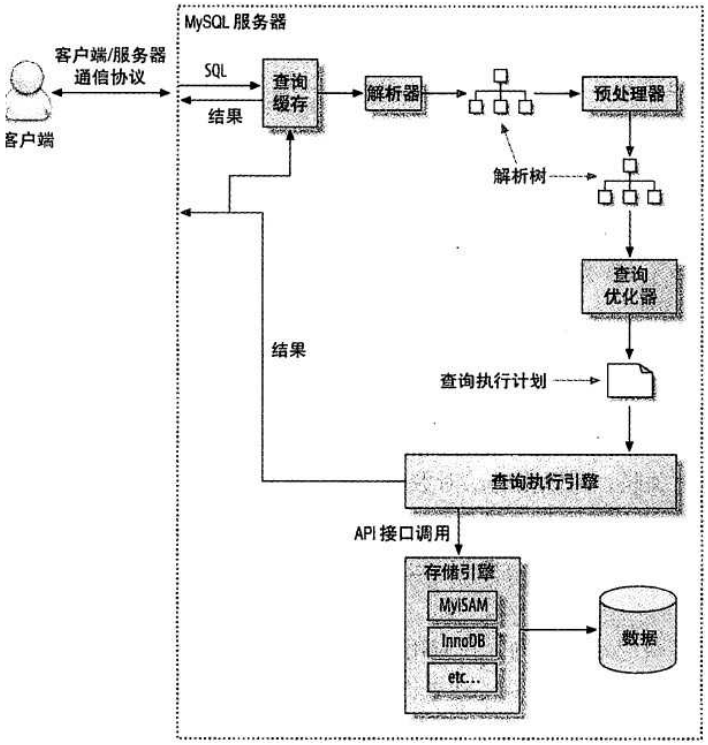
在完成这些子任务的时候，查询需要在不同的地方花费时间，包括网络，CPU计算，生成统计信息和执行计划、锁等待（互斥等待）等操作，尤其是向底层存储引擎检索数据的调用操作,这些调用需要在内存操作,CPU操作和内存不足时导致的IO操作上消耗时间。根据存储引擎不同，可能还会产生大量的上下文切换以及系统调用。

优化查询的目的就是减少和消除这些操作所花费的时间。

### *查询执行的流程再回顾*

当希望MySQL 能够以更高的性能运行查询时，最好的办法就是弄清楚MySQL是如何优化和执行查询的。一旦理解这一点，很多查询优化工作实际上就是遵循一些原则让优化器能够按照预想的合理的方式运行。

换句话说，是时候回头看看我们前面讨论的内容了:MySQL执行一个查询的过程。根据下图，我们可以看到当向MySQL发送一个请求的时候，MySQL 到底做了些什么:




1.客户端发送一条查询给服务器。

2．服务器先检查查询缓存，如果命中了缓存，则立刻返回存储在缓存中的结果。否则进入下一阶段（当然从MySQL8.0开始，这个部分就没有了）。

3．服务器端进行SQL解析、预处理，再由优化器生成对应的执行计划。

4．MySQL根据优化器生成的执行计划，调用存储引擎的API来执行查询。

5．将结果返回给客户端。

所以MySQL查询的生命周期大致可以按照顺序来看:从客户端，到服务器，然后在服务器上进行解析，生成执行计划，执行，并返回结果给客户端。其中“执行”可以认为是整个生命周期中最重要的阶段，这其中包括了大量为了检索数据到存储引擎的调用以及调用后的数据处理，包括排序、分组等。

上面的每一步都比想象的复杂，查询优化器固然是其中特别复杂也特别难理解的部分。但是其他部分就对查询的性能毫无影响？

### *MySQL客户端/服务器通信*

一般来说，不需要去理解MySQL通信协议的内部实现细节，只需要大致理解通信协议是如何工作的。MySQL客户端和服务器之间的通信协议是“半双工”的，这意味着，在任何一个时刻，要么是由服务器向客户端发送数据，要么是由客户端向服务器发送数据，这两个动作不能同时发生。所以，我们无法也无须将一个消息切成小块独立来发送。

这种协议让 MySQL通信简单快速，但是也从很多地方限制了MySQL。一个明显的限制是，这意味着没法进行流量控制。一旦一端开始发生消息，另一端要接收完整个消息才能响应它。这就像来回抛球的游戏；在任何时刻，只有一个人能控制球，而且只有控制球的人才能将球抛回去（发送消息）。

客户端用一个单独的数据包将查询传给服务器。这也是为什么当查询的语句很长的时候，参数max_allowed_packet就特别重要了。一旦客户端发送了请求，它能做的事情就只是等待结果了。

相反的，一般服务器响应给用户的数据通常很多，由多个数据包组成。当服务器开始响应客户端请求时，客户端必须完整地接收整个返回结果，而不能简单地只取前面几条结果，然后让服务器停止发送数据。这种情况下，客户端若接收完整的结果，然后取前面几条需要的结果，或者接收完几条结果后就“粗暴”地断开连接，都不是好主意。这也是在必要的时候一定要在查询中加上LIMIT限制的原因。

换一种方式解释这种行为:当客户端从服务器取数据时，看起来是一个拉数据的过程，但实际上是MySQL在向客户端推送数据的过程。客户端不断地接收从服务器推送的数据，客户端也没法让服务器停下来。

多数连接MySQL 的库函数都可以获得全部结果集并缓存到内存里，还可以逐行获取需要的数据。默认一般是获得全部结果集并缓存到内存中。MySQL通常需要等所有的数据都已经发送给客户端才能释放这条查询所占用的资源，所以接收全部结果并缓存通常可以减少服务器的压力，让查询能够早点结束、早点释放相应的资源。

当使用库函数从MySQL获取数据时，其结果看起来都像是从MySQL服务器获取数据，而实际上都是从这个库函数的缓存获取数据。多数情况下这没什么问题，但是如果需要返回一个很大的结果集的时候，这样做并不好，因为库函数会花很多时间和内存来存储所有的结果集。

对于Java程序来说，很有可能发生OOM，所以MySQL的JDBC里提供了setFetchSize() 之类的功能，来解决这个问题：

1、当statement设置以下属性时，采用的是流数据接收方式，每次只从服务器接收部份数据，直到所有数据处理完毕，不会发生JVM OOM。

setResultSetType(ResultSet.TYPE_FORWARD_ONLY);

setFetchSize(Integer.MIN_VALUE);

2、调用statement的enableStreamingResults方法，实际上enableStreamingResults方法内部封装的就是第1种方式。

3、设置连接属性useCursorFetch=true (5.0版驱动开始支持)，statement以TYPE_FORWARD_ONLY打开，再设置fetch size参数，表示采用服务器端游标，每次从服务器取fetch_size条数据。

比如：

```
con = DriverManager.getConnection(url);
ps = (PreparedStatement)
con.prepareStatement(sql,ResultSet.TYPE_FORWARD_ONLY,
ResultSet.CONCUR_READ_ONLY);
ps.setFetchSize(Integer.MIN_VALUE);
ps.setFetchDirection(ResultSet.FETCH_REVERSE);
rs = ps.executeQuery();
while (rs.next()) {……实际的业务处理}
```

### *生命周期中的查询优化处理*

查询的生命周期的下一步是将一个SQL转换成一个执行计划，MySQL再依照这个执行计划和存储引擎进行交互。这包括多个子阶段:解析SQL、预处理、优化SQL执行计划。这个过程中任何错误（例如语法错误）都可能终止查询。在实际执行中，这几部分可能一起执行也可能单独执行。

我们通过前面的学习，可以看到MySQL的查询优化器是一个非常复杂的部件，它使用了很多优化策略来生成一个最优的执行计划。优化策略可以简单地分为两种，一种是静态优化，一种是动态优化。静态优化可以直接对解析树进行分析，并完成优化。例如，优化器可以通过一些简单的代数变换将WHERE条件转换成另一种等价形式。静态优化不依赖于特别的数值，如WHERE条件中带入的一些常数等。静态优化在第一次完成后就一直有效，即使使用不同的参数重复执行查询也不会发生变化。可以认为这是一种“编译时优化”。

相反，动态优化则和查询的上下文有关，也可能和很多其他因素有关，例如WHERE条件中的取值、索引中条目对应的数据行数等。这需要在每次查询的时候都重新评估，可以认为这是“运行时优化”。

优化器是相当复杂性和智能的。建议大家“不要自以为比优化器更聪明”。如果没有必要，不要去干扰优化器的工作，让优化器按照它的方式工作。尽量按照优化器的提示去优化我们的表、索引和SQL语句，比如写查询，或者重新设计更优的库表结构，或者添

加更合适的索引。但是请尽可能的保持SQL语句的简洁，SQL语句变得很复杂的情况下，请相信我，维护会成为一个地狱。而带来的最终的收益微乎其微。

当然，虽然优化器已经很智能了，但是有时候也无法给出最优的结果。有时候你可能比优化器更了解数据，例如，由于应用逻辑使得某些条件总是成立；还有时，优化器缺少某种功能特性，如哈希索引；再如前面提到的，从优化器的执行成本角度评估出来的最优执行计划，实际运行中可能比其他的执行计划更慢。

如果能够确认优化器给出的不是最佳选择，并且清楚优化背后的原理，那么也可以帮助优化器做进一步的优化。

当出现不理想的SQL查询时，我们就需要知道查询优化器是如何工作的，以便有针对性的进行改进，不管是SQL语句本身还是表结构相关，比如索引。这个时候请仔细耐心的对慢查询进行分析。

### *查询执行引擎*

在解析和优化阶段，MySQL将生成查询对应的执行计划，MySQL的查询执行引擎则根据这个执行计划来完成整个查询。相对于查询优化阶段，查询执行阶段不是那么复杂：MySQL 只是简单地根据执行计划给出的指令逐步执行。

### *返回结果给客户端*

查询执行的最后一个阶段是将结果返回给客户端。即使查询不需要返回结果集给客户端，MySQL仍然会返回这个查询的一些信息，如该查询影响到的行数。

如果查询可以被缓存，那么MySQL在这个阶段也会将结果存放到查询缓存中。

MySQL将结果集返回客户端是一个增量、逐步返回的过程。一旦服务器开始生成第一条结果时，MySQL就可以开始向客户端逐步返回结果集了。

这一点从MySQL的源码sql_union.cc中其实可以看得很清楚：

sql_union.cc (sql) ×

```
1512:       }
1513:       if (fake_select_lex != nullptr) {
1514:         thd->inc_examined_row_count(fake_select_lex->join->examined_rows);
1515:       }
1516:     });
1517:
1518:     //开始循环处理数据集
1519:     for (;;) {
1520:       //读取数据记录
1521:       int error = m_root_iterator->Read();
1522:       DBUG_EXECUTE_IF("bug13822652_1", thd->killed = THD::KILL_QUERY;);
1523:
1524:       if (error > 0 || thd->is_error())  // Fatal error
1525:         return true;
1526:       else if (error < 0)
1527:         break;
1528:       else if (thd->killed)  // Aborted by user
1529:       {
1530:         thd->send_kill_message();
1531:         return true;
1532:       }
1533:
1534:       //发送记录给查询客户端
1535:       ++*send_records_ptr;
1536:       if (query_result->send_data(thd, *fields)) {
1537:         return true;
1538:       }
1539:       thd->get_stmt_da()->inc_current_row_for_condition();
1540:     }
1541:
1542:     // NOTE: join_cleanup must be done before we send EOF, so that we get the
1543:     // row counts right.
1544:   }
1545:
1546:   thd->current_found_rows = *send_records_ptr;
1547:
1548:   //发送记录集结束符给查询客户端
1549:   return query_result->send_eof(thd);
1550: }
```

这样处理有两个好处：服务器端无须存储太多的结果，也就不会因为要返回太多结果而消耗太多内存。另外，这样的处理也让 MySQL客户端第一时间获得返回的结果。结果集中的每一行都会以一个满足MySQL客户端/服务器通信协议的封包发送，再通过TCP协议进行传输，在TCP传输的过程中，可能对MySQL的封包进行缓存然后批量传输。

### 查询状态

对于一个MySQL连接，或者说一个线程，任何时刻都有一个状态，该状态表示了MySQL当前正在做什么。在一个查询的生命周期中，状态会变化很多次。

### 通过show profile分析SQL

通过上面的描述可知，当我们通过应用程序访问MySQL服务时，有时候性能不一定全部卡在语句的执行上。当然通过慢查询日志定位那些执行效率较低的SQL语句时候我们常用的手段，但是：

一、慢查询日志在查询结束以后才记录，在应用反映执行效率出现问题的时候查询未必执行完成；

二、有时候问题的产生不一定是语句的执行，有可能是其他原因导致的。慢查询日志并不能定位问题。

#### show processlist

这个时候通过show processlist;查看线程状态非常有用,这可以让我们很快地了解当前MySQL在进行的线程,包括线程的状态、是否锁表等，可以实时地查看SQL 的执行情况，同时对一些锁表操作进行优化。在一个繁忙的服务器上，可能会看到大量的不正常的状态，例如statistics正占用大量的时间。这通常表示，某个地方有异常了。线程常见的状态有很多，比如

statistics

The server is calculating statistics to develop a query execution plan. If a thread is in this state for a long time, the server is probably disk-bound performing other work.

服务器正在计算统计信息以研究一个查询执行计划。如果线程长时间处于此状态，则服务器可能是磁盘绑定执行其他工作。

Creating tmp table

The thread is creating a temporary table in memory or on disk. If the table is created in memory but later is converted to an on-disk table, the state during that operation is Copying to tmp table on disk.

该线程正在内存或磁盘上创建临时表。如果表在内存中创建但稍后转换为磁盘表，则该操作期间的状态将为 Copying to tmp table on disk

Sending data

The thread is reading and processing rows for a SELECT statement, and sending data to the client. Because operations occurring during this state tend to perform large amounts of disk access (reads), it is often the longest-running state over the lifetime of a given query.

线程正在读取和处理 SELECT 语句的行，并将数据发送到客户端。由于在此状态期间发生的操作往往会执行大量磁盘访问（读取），因此它通常是给定查询生命周期中运行时间最长的状态。

其余的可以参考：

https://dev.mysql.com/doc/refman/8.0/en/general-thread-states.html

### 通过show profile分析

对于每个线程到底时间花在哪里，可以通过show profile来分析。

1、首先检查当前MySQL是否支持profile

```
select @@have_profiling;
```

```
mysql> select @@have_profiling;
+------------------+
| @@have_profiling |
+------------------+
| YES              |
+------------------+
1 row in set, 1 warning (0.00 sec)
```

2、默认profiling是关闭的，可以通过set语句在 Session级别开启 profiling:

```
select @@profiling;
set profiling=1;
```

```
mysql> select @@profiling;
+-------------+
| @@profiling |
+-------------+
|           0 |
+-------------+
1 row in set, 1 warning (0.00 sec)

mysql> set profiling=1;
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

3、执行一个SQL查询

```
select count(*) from order_exp;
```

4、通过show profiles语句，看到当前SQL的Query ID

```
show profiles;
```

```
mysql> show profiles;
+----------+------------+--------------------------------+
| Query_ID | Duration   | Query                          |
+----------+------------+--------------------------------+
|        1 | 0.00212775 | select count(*) from order_exp |
+----------+------------+--------------------------------+
1 row in set, 1 warning (0.00 sec)
```

5、通过show profile for query语句能够看到执行过程中线程的每个状态和消耗的时间

show profile for query 1;

```
mysql> show profile for query 1;
+----------------------+----------+
| Status               | Duration |
+----------------------+----------+
| starting             | 0.000084 |
| checking permissions | 0.000012 |
| checking permissions | 0.000004 |
| Opening tables       | 0.000025 |
| init                 | 0.000041 |
| System lock          | 0.000010 |
| optimizing           | 0.000007 |
| statistics           | 0.000031 |
| preparing            | 0.000019 |
| executing            | 0.000003 |
| Sending data         | 0.001846 |
| end                  | 0.000005 |
| query end            | 0.000009 |
| closing tables       | 0.000007 |
| freeing items        | 0.000013 |
| cleaning up          | 0.000015 |
+----------------------+----------+
16 rows in set, 1 warning (0.00 sec)
```

通过仔细检查show profile for query 的输出，能够发现在执行COUNT(*)的过程中，时间主要消耗在 Sending data这个状态上。

6、在获取到最消耗时间的线程状态后，MySQL 支持进一步选择all、cpu、block io、contextswitch、page faults等明细类型来查看MySQL在使用什么资源上耗费了过高的时间:

show profile all for query 1\G

```
*************************** 11. row ***************************
             Status: Sending data
           Duration: 0.001846
           CPU_user: 0.001846
         CPU_system: 0.000000
  Context_voluntary: 0
Context_involuntary: 0
       Block_ops_in: 0
      Block_ops_out: 0
      Messages_sent: 0
  Messages_received: 0
  Page_faults_major: 0
  Page_faults_minor: 0
              Swaps: 0
    Source_function: exec
        Source_file: sql_executor.cc
        Source_line: 202
```

能够发现Sending data状态下，时间主要消耗在 CPU上了。

所以show profile能够在做SQL优化时帮助我们了解时间都耗费到哪里去了，同时如果MySQL源码感兴趣，还可以通过show profile source for query查看SQL解析执行过程中每个步骤对应的源码的文件、函数名以及具体的源文件行数。