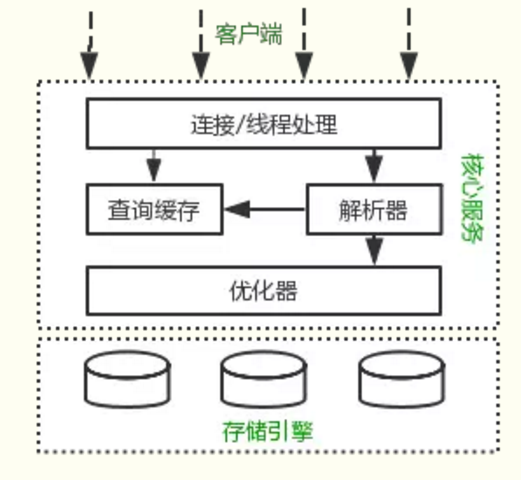
简单的剖析过程：

说起MySQL的**查询优化**，相信大家收藏了一堆奇技淫巧：不能使用SELECT \*、不使用NULL字段、合理创建索引、为字段选择合适的数据类型..... 你是否真的理解这些优化技巧？是否理解其背后的工作原理？在实际场景下性能真有提升吗？我想未必。因而理解这些优化建议背后的原理就尤为重要，希望本文能让你重新审视这些优化建议，并在实际业务场景下合理的运用。

MySQL逻辑架构

如果能在头**脑中构建一幅MySQL各组件之间如何协同工作的架构图**，有助于深入理解MySQL服务器。下图展示了MySQL的逻辑架构图。



MySQL逻辑架构，来自：高性能MySQL

MySQL逻辑架构整体分为三层，最上层为客户端层，并非MySQL所独有，诸如：连接处理、授权认证、安全等功能均在这一层处理。

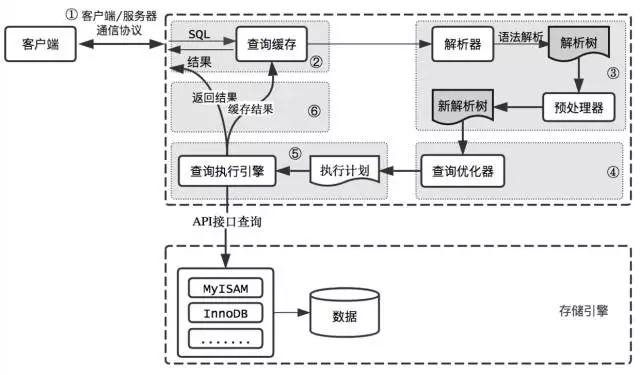
MySQL大多数核心服务均在中间这一层，包括**查询解析、分析、优化、缓存、内置函数(比如：时间、数学、加密等函数)**。所有的跨存储引擎的功能也在这一层实现：**存储过程、触发器、视图等。**

最下层为存储引擎，其负责**MySQL中的数据存储和提取**。和Linux下的文件系统类似，每种存储引擎都有其优势和劣势。中间的服务层通过API与存储引擎通信，这些API接口屏蔽了不同存储引擎间的差异。

MySQL查询过程

我们总是希望MySQL能够获得更高的查询性能，最好的办法是弄清楚MySQL是如何优化和执行查询的。一旦理解了这一点，就会发现：很多的查询优化工作实际上就是**遵循一些原则让MySQL的优化器能够按照预想的合理方式运行而已。**

当向MySQL发送一个请求的时候，MySQL到底做了些什么呢？



MySQL查询过程

**1.客户端/服务端通信协议**

MySQL客户端/服务端通信协议是**“半双工”**的：在任一时刻，要么是服务器向客户端发送数据，要么是客户端向服务器发送数据，这两个动作不能同时发生。一旦一端开始发送消息，另一端要接收完整个消息才能响应它，所以我们无法也无须将一个消息切成小块独立发送，也没有办法进行流量控制。

客户端用**一个单独的数据包将查询请求发送给服务器**，所以当查询语句很长的时候，需要设置max\_allowed\_packet参数。但是需要注意的是，**如果查询实在是太大，服务端会拒绝接收更多数据并抛出异常。**

与之相反的是，服务器响应给用户的数据通常会很多，**由多个数据包组成。**但是当服务器响应客户端请求时，客户端必须完整的接收整个返回结果，而不能简单的只取前面几条结果，然后让服务器停止发送。因而在实际开发中，尽量保持查询简单且只返回必需的数据，减小通信间数据包的大小和数量是一个非常好的习惯，**这也是查询中尽量避免使用SELECT \*以及加上LIMIT限制的原因之一。**

**2.查询缓存**

在解析一个查询语句前，如果查询缓存是打开的，那么MySQL会检查这个查询语句是否命中查询缓存中的数据。如果当前查询恰好**命中查询缓存**，在检查一次用户权限后直接返回缓存中的结果。这种情况下，查询不会被解析，也不会生成执行计划，更不会执行。

MySQL将**缓存存放在一个引用表**（不要理解成table，**可以认为是类似于HashMap的数据结构），通过一个哈希值索引**，这个哈希值通过查询本身、当前要查询的数据库、客户端协议版本号等一些可能影响结果的信息计算得来。**所以两个查询在任何字符上的不同（例如：空格、注释），都会导致缓存不会命中**。

如果查询中包含任何用户自定义函数、存储函数、用户变量、临时表、MySQL库中的系统表，其查询结果都不会被缓存。比如函数NOW()或者CURRENT\_DATE()会因为不同的查询时间，返回不同的查询结果，再比如包含CURRENT\_USER或者CONNECION\_ID()的查询语句会因为不同的用户而返回不同的结果，将这样的查询结果缓存起来没有任何的意义。

既然是缓存，就会失效，那查询缓存何时失效呢？**MySQL的查询缓存系统会跟踪查询中涉及的每个表，如果这些表（数据或结构）发生变化，那么和这张表相关的所有缓存数据都将失效**。正因为如此，在任何的写操作时，MySQL必须将对应表的所有缓存都设置为失效。如果查询缓存非常大或者碎片很多，这个操作就可能带来很大的系统消耗，甚至导致系统僵死一会儿。而且**查询缓存对系统的额外消耗也不仅仅在写操作，读操作也不例外：**

1. 任何的查询语句在开始之前都必须经过检查，即使这条SQL语句永远不会命中缓存
2. 如果查询结果可以被缓存，那么执行完成后，会将结果存入缓存，也会带来额外的系统消耗

基于此，我们要**知道并不是什么情况下查询缓存都会提高系统性能**，缓存和失效都会带来额外消耗，只有当缓存带来的资源节约大于其本身消耗的资源时，才会给系统带来性能提升。但要如何评估打开缓存是否能够带来性能提升是一件非常困难的事情，也不在本文讨论的范畴内。如果系统确实存在一些性能问题，可以尝试**打开查询缓存，并在数据库设计上做一些优化**，比如：

1. 用多个小表代替一个大表，注意不要过度设计
2. 批量插入代替循环单条插入
3. 合理控制缓存空间大小，一般来说其大小设置为几十兆比较合适
4. 可以通过SQL\_CACHE和SQL\_NO\_CACHE来控制某个查询语句是否需要进行缓存

最后的忠告是不要轻易打开查询缓存，特别是写密集型应用。如果你实在是忍不住，可以将query\_cache\_type设置为DEMAND，这时只有加入SQL\_CACHE的查询才会走缓存，其他查询则不会，这样可以非常自由地控制哪些查询需要被缓存。

当然查询缓存系统本身是非常复杂的，这里讨论的也只是很小的一部分，其他更深入的话题，比如：缓存是如何使用内存的？如何控制内存的碎片化？事务对查询缓存有何影响等等，读者可以自行阅读相关资料，这里权当抛砖引玉吧。

**3.语法解析和预处理**

MySQL通过关键字将SQL语句进行解析，并生成一颗对应的解析树。**这个过程解析器主要通过语法规则来验证和解析**。比如SQL中是否使用了错误的关键字或者关键字的顺序是否正确等等。预处理则会根据MySQL规则进一步检查解析树是否合法。比如检查要查询的数据表和数据列是否存在等。

查询优化

经过前面的步骤生成的语法树被认为是合法的了，并且由优化器将其转化成查询计划。多数情况下，一条查询可以有很多种执行方式，最后都返回相应的结果。**优化器的作用就是找到这其中最好的执行计划。**

MySQL使用基于成本的优化器，它尝试预测一个查询使用某种执行计划时的成本，并选择其中成本最小的一个。在MySQL可以通过查询当前会话的last\_query\_cost的值来得到其计算当前查询的成本。

mysql> select \* from t\_message limit 10;

...省略结果集

mysql> show status like 'last\_query\_cost';

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

| Variable\_name   | Value       |

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

| Last\_query\_cost | 6391.799000 |

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

示例中的结果表示优化器认为大概需要做6391个数据页的随机查找才能完成上面的查询。这个结果是根据一些列的统计信息计算得来的，这些统计信息包括：每张表或者索引的页面个数、索引的基数、索引和数据行的长度、索引的分布情况等等。

**有非常多的原因会导致MySQL选择错误的执行计划**，比如统计信息不准确、不会考虑不受其控制的操作成本（用户自定义函数、存储过程）、MySQL认为的最优跟我们想的不一样（我们希望执行时间尽可能短，但MySQL值选择它认为成本小的，但成本小并不意味着执行时间短）等等。

MySQL的查询优化器是一个非常复杂的部件，它使用了**非常多的优化策略来生成一个最优的执行计划：**

* 重新定义表的关联顺序（多张表关联查询时，并不一定按照SQL中指定的顺序进行，但有一些**技巧可以指定关联顺序**）
* 优化MIN()和MAX()函数（找某列的最小值，如果该列**有索引**，只需要查找B+Tree索引最左端，反之则可以找到最大值，具体原理见下文）
* 提前终止查询（比如：**使用Limit时**，查找到满足数量的结果集后会立即终止查询）
* 优化排序（在老版本MySQL会使用两次传输排序，即先读取行指针和需要排序的字段在内存中对其排序，然后再根据排序结果去读取数据行，而新版本采用的是单次传输排序，也就是一次读取所有的数据行，然后根据给定的列排序。对于I/O密集型应用，效率会高很多）

随着MySQL的不断发展，优化器使用的优化策略也在不断的进化，这里仅仅介绍几个非常常用且容易理解的优化策略，其他的优化策略，大家自行查阅吧。

1｝查询执行引擎

在完成解析和优化阶段以后，MySQL会生成对应的执行计划，查询执行引擎根据执行计划给出的指令逐步执行得出结果。整个执行过程的大部分操作均是通过调用存储引擎实现的接口来完成，这些接口被称为**handler API**。查询过程中的每一张表由一个handler实例表示。实际上，MySQL在查询优化阶段就为每一张表创建了一个handler实例，**优化器可以根据这些实例的接口来获取表的相关信息，包括表的所有列名、索引统计信息等**。存储引擎接口提供了非常丰富的功能，但其底层仅有几十个接口，这些接口像搭积木一样完成了一次查询的大部分操作。

2｝返回结果给客户端

查询执行的最后一个阶段就是将结果返回给客户端。即使查询不到数据，MySQL仍然会返回这个查询的相关信息，比如该查询影响到的行数以及执行时间等。

如果查询缓存被打开且这个查询可以被缓存，MySQL也会将结果存放到缓存中。

结果集返回客户端是一个增量且逐步返回的过程。有可能MySQL在生成第一条结果时，就开始向客户端逐步返回结果集了。这样服务端就无须存储太多结果而消耗过多内存，也可以让客户端第一时间获得返回结果。需要注意的是，结果集中的每一行都会以一个满足①中所描述的通信协议的数据包发送，再通过TCP协议进行传输，在传输过程中，可能对MySQL的数据包进行缓存然后批量发送。

回头总结一下MySQL整个查询执行过程，总的来说分为6个步骤：

* **客户端向MySQL服务器发送一条查询请求**
* **服务器首先检查查询缓存，如果命中缓存，则立刻返回存储在缓存中的结果。否则进入下一阶段**
* **服务器进行SQL解析、预处理、再由优化器生成对应的执行计划**
* **MySQL根据执行计划，调用存储引擎的API来执行查询**
* **将结果返回给客户端，同时缓存查询结果**

性能优化建议

看了这么多，你可能会期待给出一些优化手段，是的，下面会从3个不同方面给出一些优化建议。但请等等，还有一句忠告要先送给你：**不要听信你看到的关于优化的“绝对真理”，**包括本文所讨论的内容，而应该是在实际的业务场景下通过测试来验证你关于执行计划以及响应时间的假设。

***1***schema设计与数据类型优化

选择数据类型只要遵循**小而简单**的原则就好，越小的数据类型通常会更快，占用更少的磁盘、内存，处理时需要的CPU周期也更少。越简单的数据类型在计算时需要更少的CPU周期，比如，整型就比字符操作代价低，因而会使用整型来存储ip地址，使用DATETIME来存储时间，而不是使用字符串。

**这里总结几个可能容易理解错误的技巧：**

**1.一些优化的规范**

1. **通常来说把可为NULL的列改为NOT NULL不会对性能提升有多少帮助，只是如果计划在列上创建索引，就应该将该列设置为NOT NULL。**
2. 对整数类型指定宽度，比如INT(11)，没有任何卵用。INT使用32位（4个字节）存储空间，那么它的表示范围已经确定，所以INT(1)和INT(20)对于存储和计算是相同的。
3. **UNSIGNED表示不允许负值，大致可以使正数的上限提高一倍。比如TINYINT存储范围是-128 ~ 127，而UNSIGNED TINYINT存储的范围却是0 - 255。**
4. 通常来讲，**没有太大的必要使用DECIMAL数据类型**。即使是在需要存储财务数据时，仍然可以使用**BIGINT**。比如需要精确到万分之一，那么可以将数据乘以一百万然后使用BIGINT存储。**这样可以避免浮点数计算不准确和DECIMAL精确计算代价高的问题**。
5. **TIMESTAMP使用4个字节存储空间，DATETIME使用8个字节存储空间。因而，TIMESTAMP只能表示1970 - 2038年，比DATETIME表示的范围小得多，而且TIMESTAMP的值因时区不同而不同。**
6. 大多数情况下**没有使用枚举类型的必要**，其中一个缺点是枚举的字符串列表是固定的，添加和删除字符串（枚举选项）必须使用ALTER TABLE（如果只只是在列表末尾追加元素，不需要重建表）。
7. schema的列不要太多。原因是存储引擎的API工作时需要在服务器层和存储引擎层之间通过行缓冲格式拷贝数据，然后在服务器层将缓冲内容解码成各个列，这个转换过程的代价是非常高的。如果列太多而实际使用的列又很少的话，有可能会导致CPU占用过高。
8. **大表ALTER TABLE非常耗时**，MySQL执行大部分修改表结果操作的方法是用新的结构创建一个张空表，从旧表中查出所有的数据插入新表，然后再删除旧表。尤其当内存不足而表又很大，而且还有很大索引的情况下，耗时更久。当然有一些奇技淫巧可以解决这个问题，有兴趣可自行查阅。

***2*创建高性能索引**

索引是提高MySQL查询性能的一个重要途径，但过多的索引可能会导致过高的磁盘使用率以及过高的内存占用，从而影响应用程序的整体性能。应当尽量避免事后才想起添加索引，因为事后可能需要监控大量的SQL才能定位到问题所在，而且添加索引的时间肯定是远大于初始添加索引所需要的时间，可见索引的添加也是非常有技术含量的。

**1、MySQL不会使用索引的情况：非独立的列**

“独立的列”是指索引列不能是表达式的一部分，也不能是函数的参数。比如：

select \* from where id + 1 = 5

我们很容易看出其等价于 id = 4，但是MySQL无法自动解析这个表达式，使用函数是同样的道理。

**2、前缀索引**

**如果列很长，通常可以索引开始的部分字符，这样可以有效节约索引空间，从而提高索引效率。**

**3、多列索引和索引顺序（**把性价比高的索引放到左边）

在多数情况下，在多个列上建立独立的索引并不能提高查询性能。理由非常简单，MySQL不知道选择哪个索引的查询效率更好，所以在老版本，比如MySQL5.0之前就会随便选择一个列的索引，而新的版本会采用合并索引的策略。举个简单的例子，在一张电影演员表中，在actor\_id和film\_id两个列上都建立了独立的索引，然后有如下查询：

select film\_id,actor\_id from film\_actor where actor\_id = 1 or film\_id = 1

老版本的MySQL会随机选择一个索引，但新版本做如下的优化：

select film\_id,actor\_id from film\_actor where actor\_id = 1

union all

select film\_id,actor\_id from film\_actor where film\_id = 1 and actor\_id <> 1

* **当出现多个索引做相交操作时**（多个AND条件），通常来说**一个包含所有相关列的索引要优于多个独立索引**。
* **当出现多个索引做联合操作时（多个OR条件）**，对结果集的合并、排序等操作需要耗费大量的CPU和内存资源，特别是当其中的某些索引的选择性不高，需要返回合并大量数据时，查询成本更高。所以**这种情况下还不如走全表扫描。**

因此**explain时如果发现有索引合并**（Extra字段出现Using union），应该好好检查一下查询和表结构是不是已经是最优的，如果查询和表都没有问题，那只能说明索引建的非常糟糕，应当慎重考虑索引是否合适，有可能一个包含所有相关列比多列索引更适合。（最好使用独立索引）

前面我们提到过索引如何组织数据存储的，从图中可以看到多列索引时，**索引的顺序对于查询是至关重要的，很明显应该把选择性更高的字段放到索引的前面，这样通过第一个字段就可以过滤掉大多数不符合条件的数据。**

**索引选择性是指不重复的索引值和数据表的总记录数的比值**，选择性越高查询效率越高，因为选择性越高的索引可以让MySQL在查询时过滤掉更多的行。唯一索引的选择性是1，这时最好的索引选择性，性能也是最好的。

理解索引选择性的概念后，就不难确定哪个字段的选择性较高了，查一下就知道了，比如：

SELECT \* FROM payment where staff\_id = 2 and customer\_id = 584

是应该创建(staff\_id,customer\_id)的索引还是应该颠倒一下顺序？执行下面的查询，哪个字段的**选择性更接近1就把哪个字段索引前面就好。**

select count(distinct staff\_id)/count(\*) as staff\_id\_selectivity,

       count(distinct customer\_id)/count(\*) as customer\_id\_selectivity,

       count(\*) from payment

多数情况下使用这个原则没有任何问题，但仍然注意你的数据中是否存在一些特殊情况。举个简单的例子，比如要查询某个用户组下有过交易的用户信息：

select user\_id from trade where user\_group\_id = 1 and trade\_amount > 0

MySQL为这个查询选择了索引(user\_group\_id,trade\_amount)，如果不考虑特殊情况，这看起来没有任何问题，但实际情况是这张表的大多数数据都是从老系统中迁移过来的，由于新老系统的数据不兼容，所以就给老系统迁移过来的数据赋予了一个默认的用户组。这种情况下，通过索引扫描的行数跟全表扫描基本没什么区别，索引也就起不到任何作用。

推广开来说，经验法则和推论在多数情况下是有用的，可以指导我们开发和设计，但实际情况往往会更复杂，实际业务场景下的某些特殊情况可能会摧毁你的整个设计。

**4、避免多个范围条件**

实际开发中，我们会经常使用多个范围条件，比如想查询某个时间段内登录过的用户：

select user.\* from user where login\_time > '2017-04-01' and age between 18 and 30;

这个查询有一个问题：它有两个范围条件，login\_time列和age列，MySQL可以使用login\_time列的索引或者age列的索引，但无法同时使用它们。

**5、覆盖索引**

如果一个索引包含或者说覆盖所有需要查询的字段的值，那么就没有必要再回表查询，这就称为覆盖索引。覆盖索引是非常有用的工具，可以极大的提高性能，因为查询只需要扫描索引会带来许多好处：

* 索引条目远小于数据行大小，如果只读取索引，极大减少数据访问量
* 索引是有按照列值顺序存储的，对于I/O密集型的范围查询要比随机从磁盘读取每一行数据的IO要少的多

**6、使用索引扫描来排序**

MySQL有两种方式可以生产有序的结果集，**其一是对结果集进行排序的操作，其二是按照索引顺序扫描得出的结果自然是有序的。**如果explain的结果中type列的值为index表示使用了索引扫描来做排序。

扫描索引本身很快，因为只需要从一条索引记录移动到相邻的下一条记录。但如果索引本身不能覆盖所有需要查询的列，那么就不得不每扫描一条索引记录就回表查询一次对应的行。这个读取操作基本上是随机I/O，因此按照索引顺序读取数据的速度通常要比顺序地全表扫描要慢。

**在设计索引时，如果一个索引既能够满足排序，又满足查询，是最好的。**

只有当索引的列顺序和ORDER BY子句的顺序完全一致，并且所有列的排序方向也一样时，才能够使用索引来对结果做排序。如果查询需要关联多张表，则只有ORDER BY子句引用的字段全部为第一张表时，才能使用索引做排序。ORDER BY子句和查询的限制是一样的，都要满足最左前缀的要求（有一种情况例外，就是最左的列被指定为常数，下面是一个简单的示例），其它情况下都需要执行排序操作，而无法利用索引排序。

// 最左列为常数，索引：(date,staff\_id,customer\_id)

select  staff\_id,customer\_id from demo where date = '2015-06-01' order by staff\_id,customer\_id

**7、冗余和重复索引**

冗余索引是指在相同的列上按照相同的顺序创建的相同类型的索引，应当尽量避免这种索引，发现后立即删除。比如有一个索引(A,B)，再创建索引(A)就是冗余索引。冗余索引经常发生在为表添加新索引时，比如有人新建了索引(A,B)，但这个索引不是扩展已有的索引(A)。

大多数情况下都应该尽量扩展已有的索引而不是创建新索引。但有极少情况下出现性能方面的考虑需要冗余索引，比如扩展已有索引而导致其变得过大，从而影响到其他使用该索引的查询。

**8、删除长期未使用的索引**

定期删除一些长时间未使用过的索引是一个非常好的习惯。

关于索引这个话题打算就此打住，最后要说一句，索引并不总是最好的工具，只有当索引帮助提高查询速度带来的好处大于其带来的额外工作时，索引才是有效的。对于非常小的表，简单的全表扫描更高效。对于中到大型的表，索引就非常有效。对于超大型的表，建立和维护索引的代价随之增长，这时候其他技术也许更有效，比如分区表。最后的最后，explain后再提测是一种美德。

特定类型查询优化

**优化COUNT()查询**

COUNT()可能是被大家误解最多的函数了，它有两种不同的作用，其一是统计某个列值的数量，其二是统计行数。统计列值时，要求列值是非空的，它不会统计NULL。如果确认括号中的表达式不可能为空时，实际上就是在统计行数。最简单的就是当使用COUNT(\*)时，并不是我们所想象的那样扩展成所有的列，实际上，它会忽略所有的列而直接统计所有的行数。

我们最常见的误解也就在这儿，在括号内指定了一列却希望统计结果是行数，而且还常常误以为前者的性能会更好。但实际并非这样，如果要统计行数，直接使用COUNT(\*)，意义清晰，且性能更好。

有时候某些业务场景并不需要完全精确的COUNT值，可以用近似值来代替，EXPLAIN出来的行数就是一个不错的近似值，而且执行EXPLAIN并不需要真正地去执行查询，所以成本非常低。通常来说，执行COUNT()都需要扫描大量的行才能获取到精确的数据，因此很难优化，MySQL层面还能做得也就只有覆盖索引了。如果不还能解决问题，只有从架构层面解决了，比如添加汇总表，或者使用redis这样的外部缓存系统。

**优化关联查询**

在大数据场景下，表与表之间通过一个**冗余字段来关联，要比直接使用JOIN有更好的性能**。如果确实需要使用关联查询的情况下，需要特别注意的是：

1. 确保ON和USING字句中的列上有索引。在创建索引的时候就要考虑到关联的顺序。当表A和表B用列c关联的时候，如果优化器关联的顺序是A、B，那么就不需要在A表的对应列上创建索引。没有用到的索引会带来额外的负担，一般来说，除非有其他理由，只需要在关联顺序中的第二张表的相应列上创建索引（具体原因下文分析）。
2. 确保任何的GROUP BY和ORDER BY中的表达式只涉及到一个表中的列，这样MySQL才有可能使用索引来优化。

要理解优化关联查询的第一个技巧，就需要理解MySQL是如何执行关联查询的。当前MySQL关联执行的策略非常简单，它对任何的关联都执行**嵌套循环关联操作**，即先在一个表中循环取出单条数据，然后在嵌套循环到下一个表中寻找匹配的行，依次下去，直到找到所有表中匹配的行为为止。然后根据各个表匹配的行，返回查询中需要的各个列。

太抽象了？以上面的示例来说明，比如有这样的一个查询：

SELECT A.xx,B.yy

FROM A INNER JOIN B USING(c)

WHERE A.xx IN (5,6)

假设MySQL按照查询中的关联顺序A、B来进行关联操作，那么可以用下面的伪代码表示MySQL如何完成这个查询：

outer\_iterator = SELECT A.xx,A.c FROM A WHERE A.xx IN (5,6);

outer\_row = outer\_iterator.next;

while(outer\_row) {

    inner\_iterator = SELECT B.yy FROM B WHERE B.c = outer\_row.c;

    inner\_row = inner\_iterator.next;

    while(inner\_row) {

        output[inner\_row.yy,outer\_row.xx];

        inner\_row = inner\_iterator.next;

    }

    outer\_row = outer\_iterator.next;

}

可以看到，最外层的查询是根据A.xx列来查询的，A.c上如果有索引的话，整个关联查询也不会使用。再看内层的查询，很明显B.c上如果有索引的话，能够加速查询，因此只需要在关联顺序中的第二张表的相应列上创建索引即可。

**优化LIMIT分页**

当需要分页操作时，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字句。如果有对应的索引，通常效率会不错，否则，MySQL需要做大量的文件排序操作。

一个常见的问题是当偏移量非常大的时候，比如：LIMIT 10000 20这样的查询，MySQL需要查询10020条记录然后只返回20条记录，前面的10000条都将被抛弃，这样的代价非常高。

优化这种查询一个最简单的办法就**是尽可能的使用覆盖索引扫描**，而不是查询所有的列。然后根据需要做一次关联查询再返回所有的列。对于偏移量很大时，这样做的效率会提升非常大。

考虑下面的查询：

SELECT film\_id,description FROM film ORDER BY title LIMIT 50,5;

如果这张表非常大，那么这个查询最好改成下面的样子：

SELECT film.film\_id,film.description

FROM film INNER JOIN (

    SELECT film\_id FROM film ORDER BY title LIMIT 50,5

) AS tmp USING(film\_id);

这里的延迟关联将大大提升查询效率，**让MySQL扫描尽可能少的页面，获取需要访问的记录后在根据关联列回原表查询所需要的列。**

有时候如果可以使用书签记录上次取数据的位置，那么下次就可以直接从该书签记录的位置开始扫描，这样就可以避免使用OFFSET，比如下面的查询：

SELECT id FROM t LIMIT 10000, 10;

改为：

SELECT id FROM t WHERE id > 10000 LIMIT 10;

其它优化的办法还包括使用预先计算的汇总表，或者关联到一个冗余表，冗余表中只包含主键列和需要做排序的列。

MySQL处理UNION的策略是先创建临时表，然后再把各个查询结果插入到临时表中，最后再来做查询。因此很多优化策略在UNION查询中都没有办法很好的时候。经常需要手动将WHERE、LIMIT、ORDER BY等字句“下推”到各个子查询中，以便优化器可以充分利用这些条件先优化。

除非确实需要服务器去重，否则就一定要使用UNION ALL，如果没有ALL关键字，MySQL会给临时表加上DISTINCT选项，这会导致整个临时表的数据做唯一性检查，这样做的代价非常高。当然即使使用ALL关键字，MySQL总是将结果放入临时表，然后再读出，再返回给客户端。虽然很多时候没有这个必要，比如有时候可以直接把每个子查询的结果返回给客户端。

结语

理解查询是如何执行以及时间都消耗在哪些地方，再加上一些优化过程的知识，可以帮助大家更好的理解MySQL，理解常见优化技巧背后的原理。希望本文中的原理、示例能够帮助大家更好的将理论和实践联系起来，更多的将理论知识运用到实践中。

**总结一下：**

**字段**

* 尽量使用TINYINT、SMALLINT、MEDIUM\_INT作为整数类型而非INT，如果非负则加上UNSIGNED
* VARCHAR的长度只分配真正需要的空间
* 使用枚举或整数代替字符串类型
* 尽量使用TIMESTAMP而非DATETIME，
* 单表不要有太多字段，建议在20以内
* 避免使用NULL字段，很难查询优化且占用额外索引空间
* 用整型来存IP

**索引**

* 索引并不是越多越好，要根据查询有针对性的创建，考虑在WHERE和ORDER BY命令上涉及的列建立索引，可根据EXPLAIN来查看是否用了索引还是全表扫描
* 应尽量避免在WHERE子句中对字段进行NULL值判断，否则将导致引擎放弃使用索引而进行全表扫描
* 值分布很稀少的字段不适合建索引，例如”性别”这种只有两三个值的字段
* 字符字段只建前缀索引
* 字符字段最好不要做主键
* 不用外键，由程序保证约束
* 尽量不用UNIQUE，由程序保证约束
* 使用多列索引时主意顺序和查询条件保持一致，同时删除不必要的单列索引

**查询SQL**

* 可通过开启慢查询日志来找出较慢的SQL
* 不做列运算：SELECT id WHERE age + 1 = 10，任何对列的操作都将导致表扫描，它包括数据库教程函数、计算表达式等等，查询时要尽可能将操作移至等号右边
* sql语句尽可能简单：一条sql只能在一个cpu运算；大语句拆小语句，减少锁时间；一条大sql可以堵死整个库
* 不用SELECT \*
* OR改写成IN：OR的效率是n级别，IN的效率是log(n)级别，in的个数建议控制在200以内
* 不用函数和触发器，在应用程序实现
* 避免%xxx式查询
* 少用JOIN
* 使用同类型进行比较，比如用'123'和'123'比，123和123比
* 尽量避免在WHERE子句中使用 != 或 <> 操作符，否则将引擎放弃使用索引而进行全表扫描
* 对于连续数值，使用BETWEEN不用IN：SELECT id FROM t WHERE num BETWEEN 1 AND 5
* 列表数据不要拿全表，要使用LIMIT来分页，每页数量也不要太大

**读写分离**

也是目前常用的优化，从库读主库写，一般不要采用双主或多主引入很多复杂性，尽量采用文中的其他方案来提高性能。同时目前很多拆分的解决方案同时也兼顾考虑了读写分离。

**缓存**

尽量不要使用缓存，采用第三方redis来缓存最好。

**其它**

表设计（范式化和反范式化）

使用慢日志来记录与分析

Explain查看索引使用情况