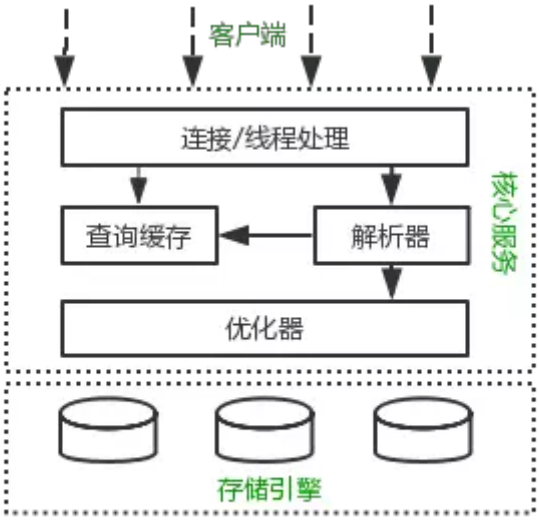
说起 MySQL 的查询优化，相信大家收藏了一堆奇技淫巧：不能使用 SELECT *、不使用 NULL 字段、合理创建索引、为字段选择合适的数据类型..... 你是否真的理解这些优化技巧？是否理解其背后的工作原理？在实际场景下性能真有提升吗？我想未必。因而理解这些优化建议背后的原理就尤为重要，希望本文能让你重新审视这些优化建议，并在实际业务场景下合理的运用。

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。

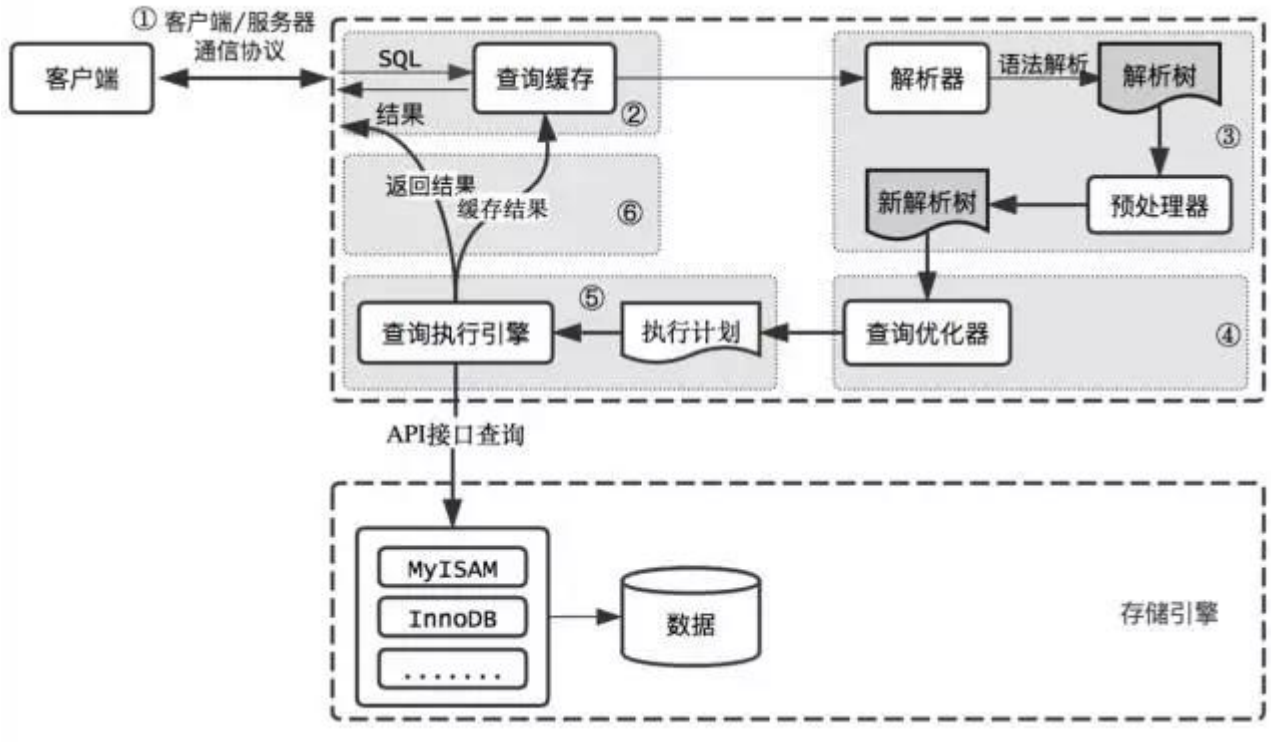
MySQL 大多数核心服务均在中间这一层，包括查询解析、分析、优化、缓存、内置函数(比如：时间、数学、加密等函数)。所有的跨存储引擎的功能也在这一层实现：存储过程、触发器、视图等。

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

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。

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。

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。

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。

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*Scheme 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. 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

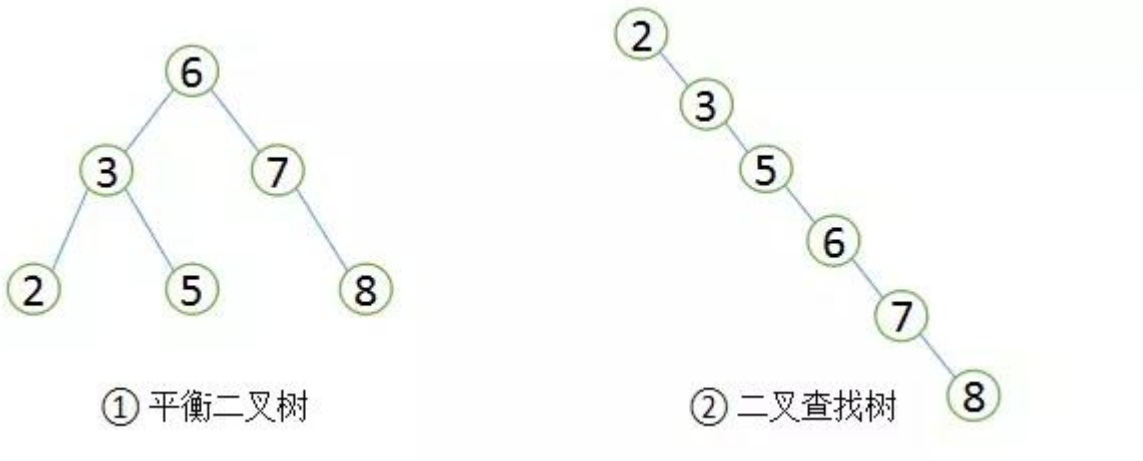
索引是提高 MySQL 查询性能的一个重要途径，但过多的索引可能会导致过高的磁盘使用率以及过高的内存占用，从而影响应用程序的整体性能。应当尽量避免事后才想起添加索引，因为事后可能需要监控大量的 SQL 才能定位到问题所在，而且添加索引的时间肯定是远大于初始添加索引所需要的时间，可见索引的添加也是非常有技术含量的。

接下来将向你展示一系列创建高性能索引的策略，以及每条策略其背后的工作原理。但在此之前，先了解与索引相关的一些算法和数据结构，将有助于更好的理解后文的内容。

3 索引相关的数据结构和算法

通常我们所说的索引是指 B-Tree 索引，它是目前关系型数据库中查找数据最为常用和有效的索引，大多数存储引擎都支持这种索引。使用 B-Tree 这个术语，是因为 MySQL 在 CREATE TABLE 或其它语句中使用了这个关键字，但实际上不同的存储引擎可能使用不同的数据结构，比如 InnoDB 就是使用的 B+Tree。
B+Tree 中的 B 是指 balance，意为平衡。需要注意的是，B+树索引并不能找到一个给定键值的具体行，它找到的只是被查找数据行所在的页，接着数据库会把页读入到内存，再在内存中进行查找，最后得到要查找的数据。

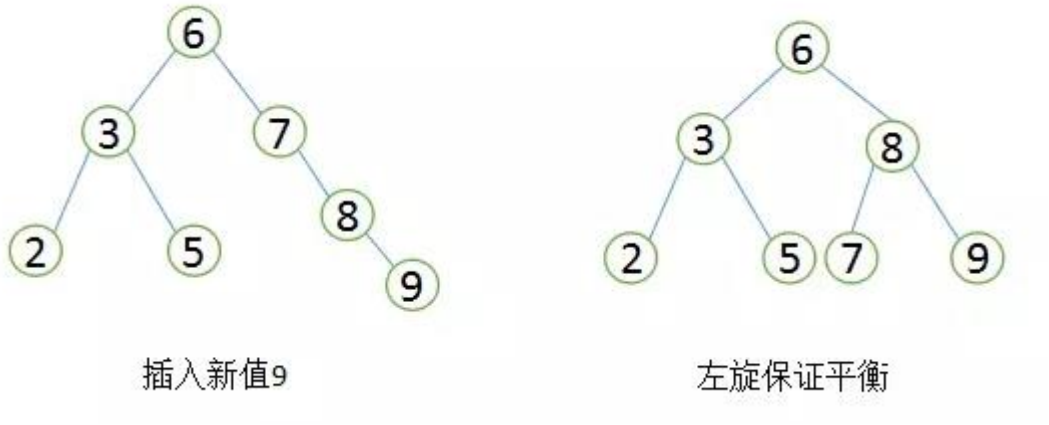
在介绍 B+Tree 前，先了解一下二叉查找树，它是一种经典的数据结构，其左子树的值总是小于根的值，右子树的值总是大于根的值，如下图①。如果要在这课树中查找值为 5 的记录，其大致流程：先找到根，其值为 6，大于 5，所以查找左子树，找到 3，而 5 大于 3，接着找 3 的右子树，总共找了 3 次。同样的方法，如果查找值为 8 的记录，也需要查找 3 次。所以二叉查找树的平均查找次数为(3 + 3 + 3 + 2 + 2 + 1) / 6 = 2.3 次，而顺序查找的话，查找值为 2 的记录，仅需要 1 次，但查找值为 8 的记录则需要 6 次，所以顺序查找的平均查找次数为：(1 + 2 + 3 + 4 + 5 + 6) / 6 = 3.3 次，因此大多数情况下二叉查找树的平均查找速度比顺序查找要快。



二叉查找树和平衡二叉树

由于二叉查找树可以任意构造，同样的值，可以构造出如图②的二叉查找树，显然这棵二叉树的查询效率和顺序查找差不多。若想二叉查找数的查询性能最高，需要这棵二叉查找树是平衡的，也即平衡二叉树（AVL 树）。

平衡二叉树首先需要符合二叉查找树的定义，其次必须满足任何节点的两个子树的高度差不能大于 1。显然图②不满足平衡二叉树的定义，而图①是一课平衡二叉树。平衡二叉树的查找性能是比较高的（性能最好的是最优二叉树），查询性能越好，维护的成本就越大。比如图①的平衡二叉树，当用户需要插入一个新的值 9 的节点时，就需要做出如下变动。

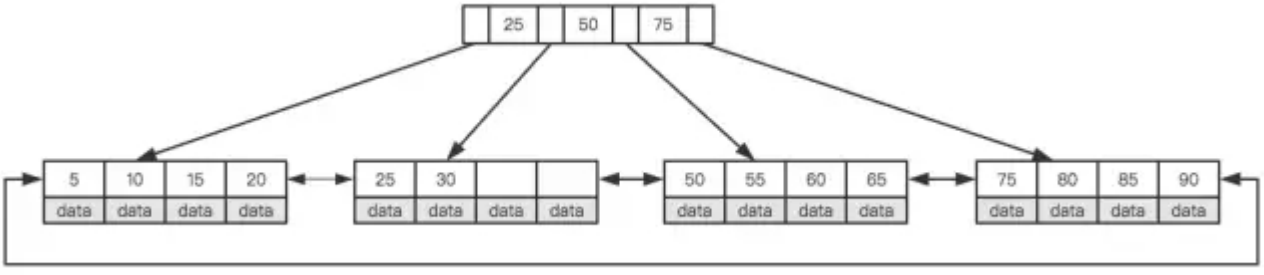


平衡二叉树旋转

通过一次左旋操作就将插入后的树重新变为平衡二叉树是最简单的情况了，实际应用场景中可能需要旋转多次。至此我们可以考虑一个问题，平衡二叉树的查找效率还不错，实现也非常简单，相应的维护成本还能接受，为什么 MySQL 索引不直接使用平衡二叉树？

随着数据库中数据的增加，索引本身大小随之增加，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘 I/O 消耗，相对于内存存取，I/O 存取的消耗要高几个数量级。可以想象一下一棵几百万节点的二叉树的深度是多少？如果将这么大深度的一颗二叉树放磁盘上，每读取一个节点，需要一次磁盘的 I/O 读取，整个查找的耗时显然是不能够接受的。那么如何减少查找过程中的 I/O 存取次数？

一种行之有效的解决方法是减少树的深度，将二叉树变为 m 叉树（多路搜索树），而 B+Tree 就是一种多路搜索树。理解 B+Tree 时，只需要理解其最重要的两个特征即可：第一，所有的关键字（可以理解为数据）都存储在叶子节点（Leaf Page），非叶子节点（Index Page）并不存储真正的数据，所有记录节点都是按键值大小顺序存放在同一层叶子节点上。其次，所有的叶子节点由指针连接。如下图为高度为 2 的简化了的 B+Tree。



简化 B+Tree

怎么理解这两个特征？MySQL 将每个节点的大小设置为一个页的整数倍（原因下文会介绍），也就是在节点空间大小一定的情况下，每个节点可以存储更多的内结点，这样每个结点能索引的范围更大更精确。所有的叶子节点使用指针链接的好处是可以进行区间访问，比如上图中，如果查找大于 20 而小于 30 的记录，只需要找到节点 20，就可以遍历指针依次找到 25、

30。如果没有链接指针的话，就无法进行区间查找。这也是 MySQL 使用 B+Tree 作为索引存储结构的重要原因。

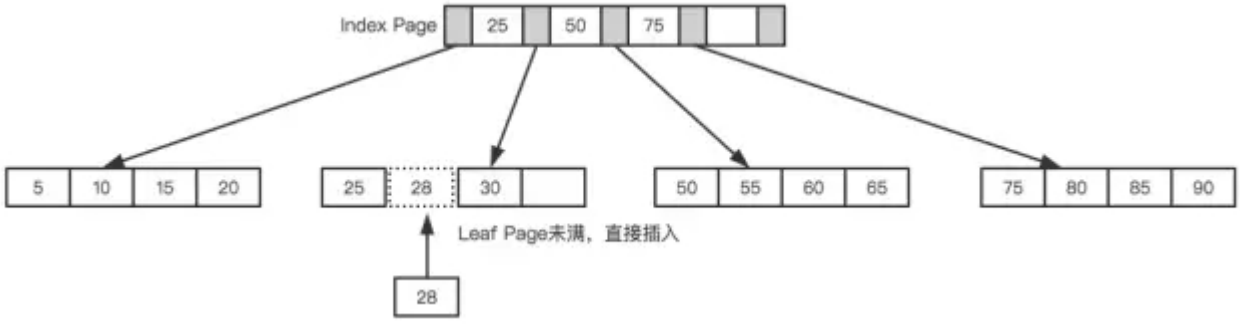
MySQL 为何将节点大小设置为页的整数倍，这就需要理解磁盘的存储原理。磁盘本身存取就比主存慢很多，在加上机械运动损耗（特别是普通的机械硬盘），磁盘的存取速度往往是主存的几百万分之一，为了尽量减少磁盘 I/O，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存，预读的长度一般为页的整数倍。

“页是计算机管理存储器的逻辑块，硬件及 OS 往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（许多 OS 中，页的大小通常为 4K）。主存和磁盘以页为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后一起返回，程序继续运行。”

MySQL 巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次 I/O 就可以完全载入。为了达到这个目的，每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了读取一个节点只需一次 I/O。假设 B+Tree 的高度为 h，一次检索最多需要 h-1I/O（根节点常驻内存），复杂度$O(h) = O(\log_{M}N)$。实际应用场景中，M 通常较大，常常超过 100，因此树的高度一般都比较小，通常不超过 3。

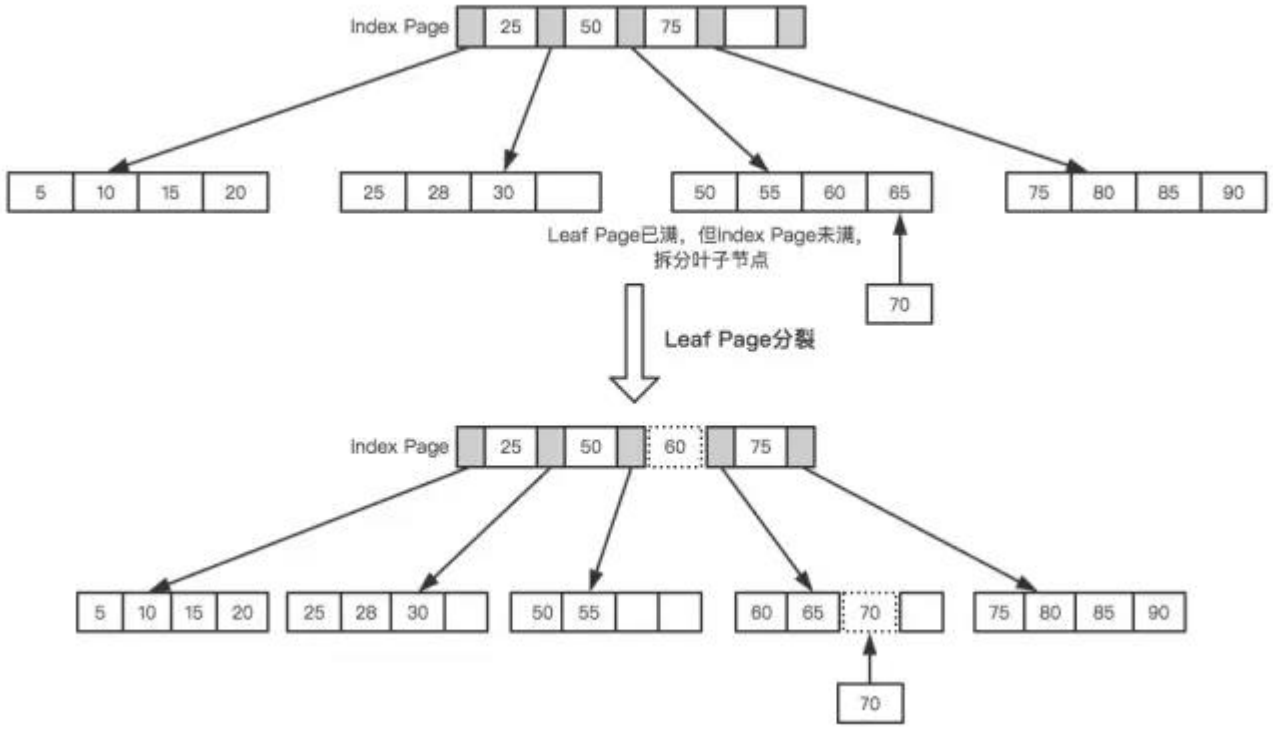
最后简单了解下 B+Tree 节点的操作，在整体上对索引的维护有一个大概的了解，虽然索引可以大大提高查询效率，但维护索引仍要花费很大的代价，因此合理的创建索引也就尤为重要。

仍以上面的树为例，我们假设每个节点只能存储 4 个内节点。首先要插入第一个节点 28，如下图所示。



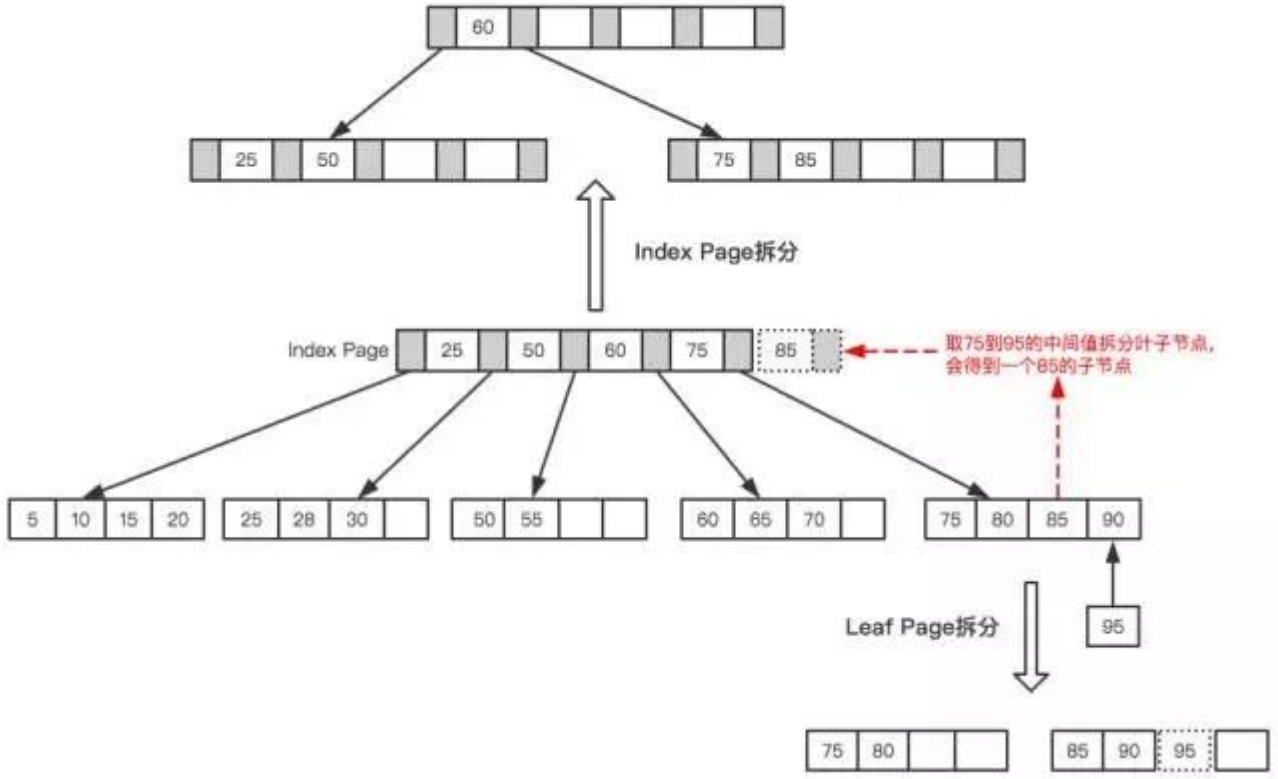
leaf page 和 index page 都没有满

接着插入下一个节点 70，在 Index Page 中查询后得知应该插入到 50 - 70 之间的叶子节点，但叶子节点已满，这时候就需要进行也分裂的操作，当前的叶子节点起点为 50，所以根据中间值来拆分叶子节点，如下图所示。

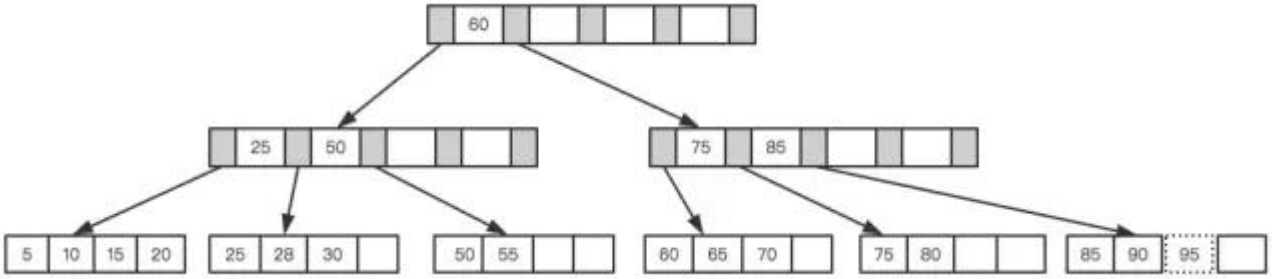


Leaf Page 拆分

最后插入一个节点 95，这时候 Index Page 和 Leaf Page 都满了，就需要做两次拆分，如下图所示。

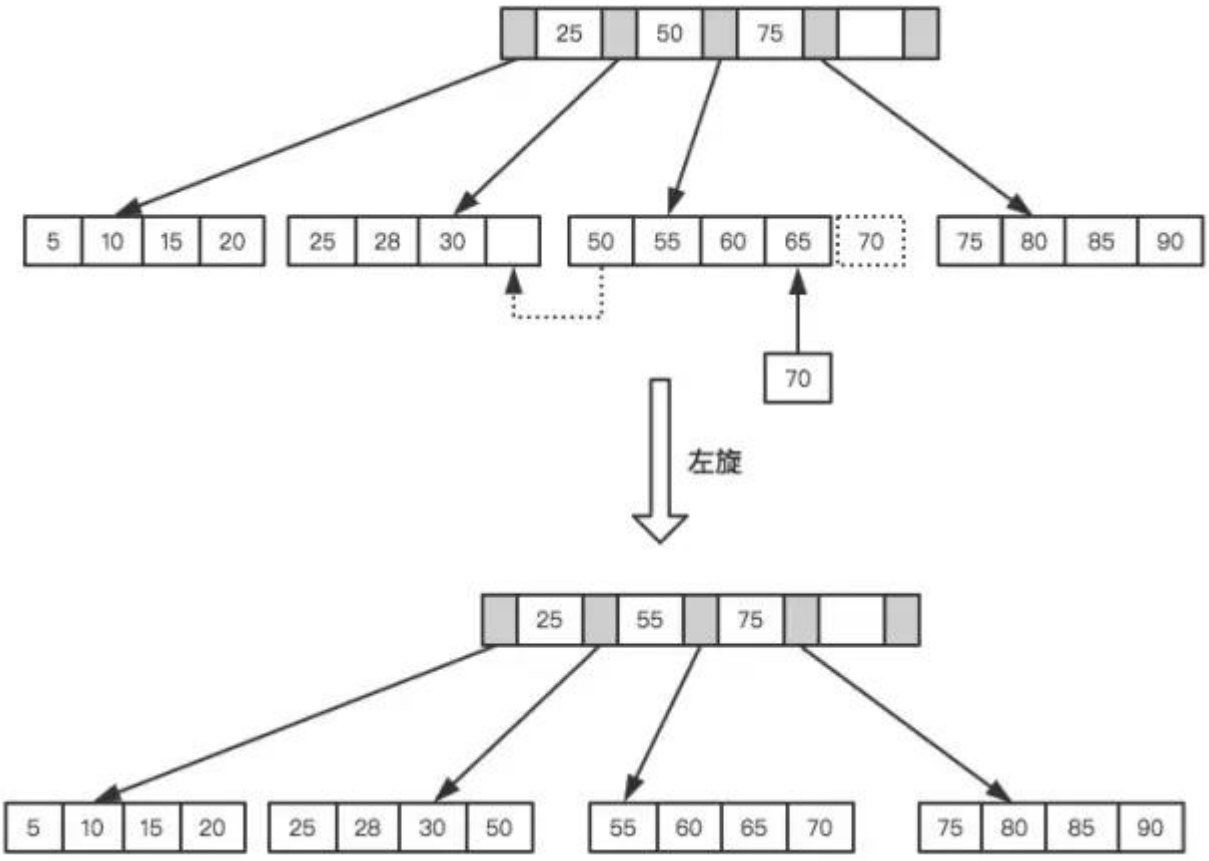


拆分后最终形成了这样一颗树。



最终树

B+Tree 为了保持平衡，对于新插入的值需要做大量的拆分页操作，而页的拆分需要 I/O 操作，为了尽可能的减少页的拆分操作，B+Tree 也提供了类似于平衡二叉树的旋转功能。当 Leaf Page 已满但其左右兄弟节点没有满的情况下，B+Tree 并不急于去做拆分操作，而是将记录移到当前所在页的兄弟节点上。通常情况下，左兄弟会被先检查用来做旋转操作。就比如上面第二个示例，当插入 70 的时候，并不会去做页拆分，而是左旋操作。



左旋操作

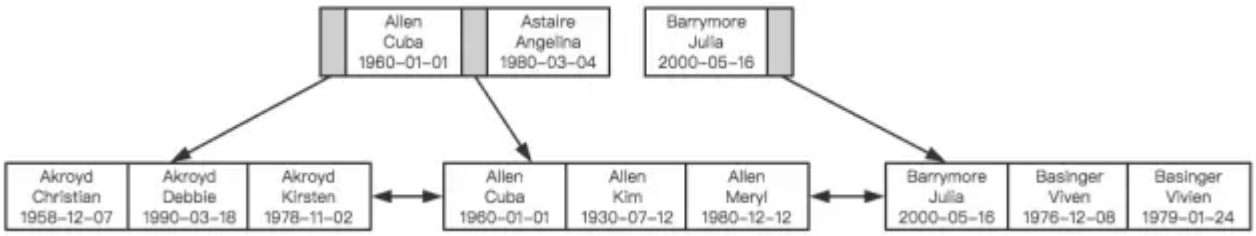
通过旋转操作可以最大限度的减少页分裂，从而减少索引维护过程中的磁盘的 I/O 操作，也提高索引维护效率。需要注意的是，删除节点跟插入节点类似，仍然需要旋转和拆分操作，这里就不再说明。

高性能策略

通过上文，相信你对 B+Tree 的数据结构已经有了大致的了解，但 MySQL 中索引是如何组织数据的存储呢？以一个简单的示例来说明，假如有如下数据表：

```
CREATE TABLE People(
        last_name varchar(50) not null,
        first_name varchar(50) not null,
        dob date not null,
        gender enum(`m`,`f`) not null,
        key(last_name,first_name,dob)
);
```

对于表中每一行数据，索引中包含了 last_name、first_name、dob 列的值，下图展示了索引是如何组织数据存储的。



索引如何组织数据存储，来自：高性能 MySQL

可以看到，索引首先根据第一个字段来排列顺序，当名字相同时，则根据第三个字段，即出生日期来排序，正是因为这个原因，才有了索引的“最左原则”。

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

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的多列索引更适合。

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。

<section style="margin: 10px 0px; padding: 15px 20px 15px 45px; max-width: 100%; box-sizing: border-box; font-size: 14px; line-height: 22.39px; outline: 0px; border-width: 0px; border-style: initial; border-color: currentcolor; vertical-align: baseline; background-image: url(" http:="" mmbiz.qpic.cn="" mmbiz_jpg="" tibrg3aoijttt5wd7pstdp8xn9fcaqn0hzm4ung7awpvy0vhxe5stzfr97tfcd3orepfelzkiawqpkjmvgnbnenq="" 0?wx_fmt="jpeg");" background-color:="" rgb(241,="" 241,="" 241);="" background-position:="" 1%="" 5px;="" background-repeat:="" no-repeat;="" word-wrap:="" break-word="" !important;"="">
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

1.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。

2. 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。

3.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。

4.优化 UNION

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。

其他也没啥说的了，给大家留两个思考题吧，可以在脑袋里想想答案，这也是大家经常挂在嘴边的，但很少有人会思考为什么？

1. 有非常多的程序员在分享时都会抛出这样一个观点：尽可能不要使用存储过程，存储过程非常不容易维护，也会增加使用成本，应该把业务逻辑放到客户端。既然客户端都能干这些事，那为什么还要存储过程？
2. JOIN 本身也挺方便的，直接查询就好了，为什么还需要视图呢？

参考资料

1. 姜承尧 著；MySQL 技术内幕-InnoDB 存储引擎；机械工业出版社，2013
2. Baron Scbwartz 等著；宁海元 周振兴等译；高性能 MySQL(第三版)；电子工业出版社，2013
3. 由 B-/B+树看 MySQL 索引结构

   https://segmentfault.com/a/1190000004690721