#### 性能优化之Mysql

## 关键字：索引、存储引擎、二叉查找树、平衡二叉查找树、多路平衡查找树（B-Tree）、加强版多路平衡查找树（B+Tree）、磁盘预读、Myisam、Innodb、列的离散型、

**最左匹配原则、联合索引、覆盖索引**

## 索引机制

正确的创建合适的索引

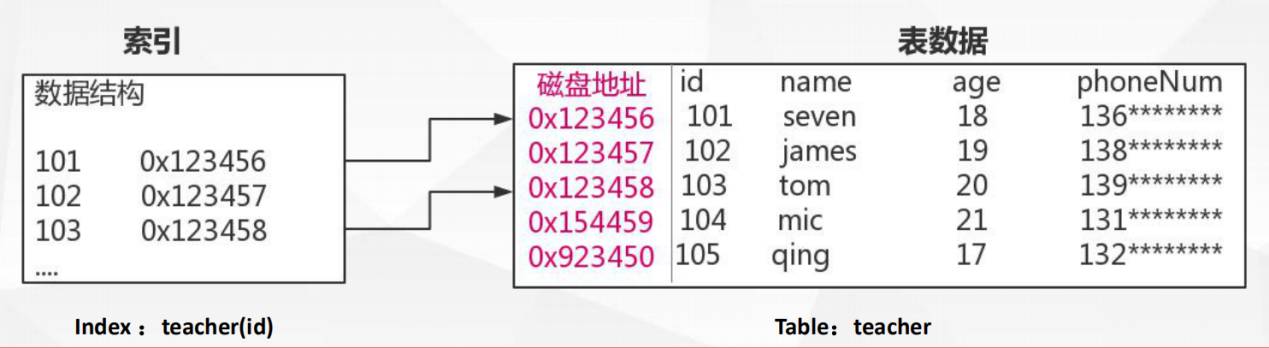
是提升数据库查询性能的基础

* **什么是索引**

索引是为了加速对表中数据行的检索而创建的一种分散存储的数据结构

索引是存储引擎用于快速找到记录的一种数据结构，索引对于良好的性能非常关键，尤其是当表中的数据量越来越大时，索引对性能的影响愈发重要。

索引优化是查询性能优化最有效的手段。



* **为什么要有索引**

目的：为了提高查询效率。

能极大的减少存储引擎需要扫描的数据量

可以把随机IO变成顺序IO

可以帮助我们在进行分组、排序等操作时，避免使用临时表

* **索引的原理**

通过不断的缩小想要获取数据的范围来筛选出最终想要的结果，同时随机事件变成顺序事件。

一般来说，索引本身也很大，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在 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，所以评价一个数据结构作为索引的优劣最重要的指标就是在查找过程中磁盘I/O操作次数的渐进复杂度。换句话说，索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数。

* **空间局部性原理（磁盘IO与预读）**

磁盘读取依靠的是[机械运动](https://www.baidu.com/s?wd=%E6%9C%BA%E6%A2%B0%E8%BF%90%E5%8A%A8&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "https://blog.csdn.net/sinat_27602945/article/details/_blank)，分为寻道时间、旋转延迟、传输时间三个部分，这三个部分耗时相加就是一次磁盘IO的时间，大概9ms左右。这个成本是访问内存的十万倍左右；正是由于磁盘IO是非常昂贵的操作，所以计算机操作系统对此做了优化：预读；每一次IO时，不仅仅把当前磁盘地址的数据加载到内存，同时也把相邻数据也加载到内存缓冲区中。因为局部预读原理说明：当访问一个地址数据的时候，与其相邻的数据很快也会被访问到。每次磁盘IO读取的数据我们称之为一页（page）。一页的大小与操作系统有关，一般为4k或者8k。这也就意味着读取一页内数据的时候，实际上发生了一次磁盘IO。

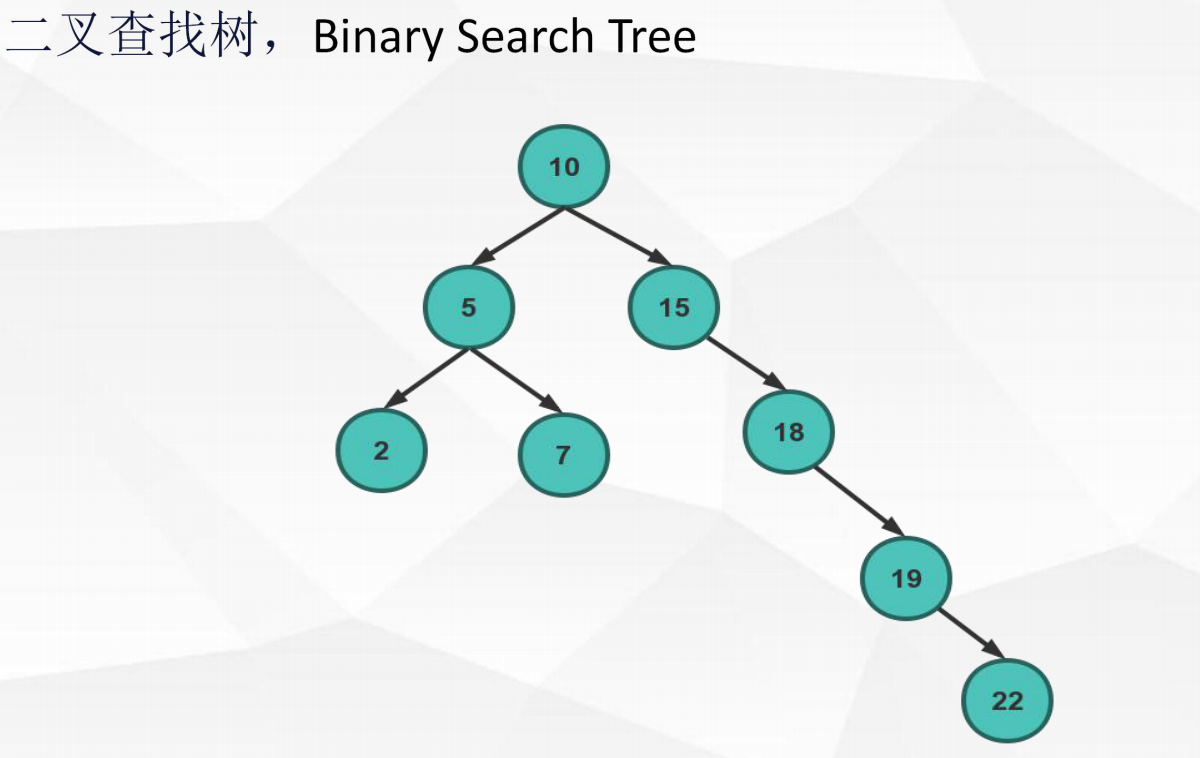
数据库索引是存储在磁盘上，当表中的数据量比较大时，索引的大小也跟着增长，达到几个G甚至更多。当我们利用索引进行查询的时候，不可能把索引全部加载到内存中，只能逐一加载每个磁盘页，这里的磁盘页就对应索引树的节点。

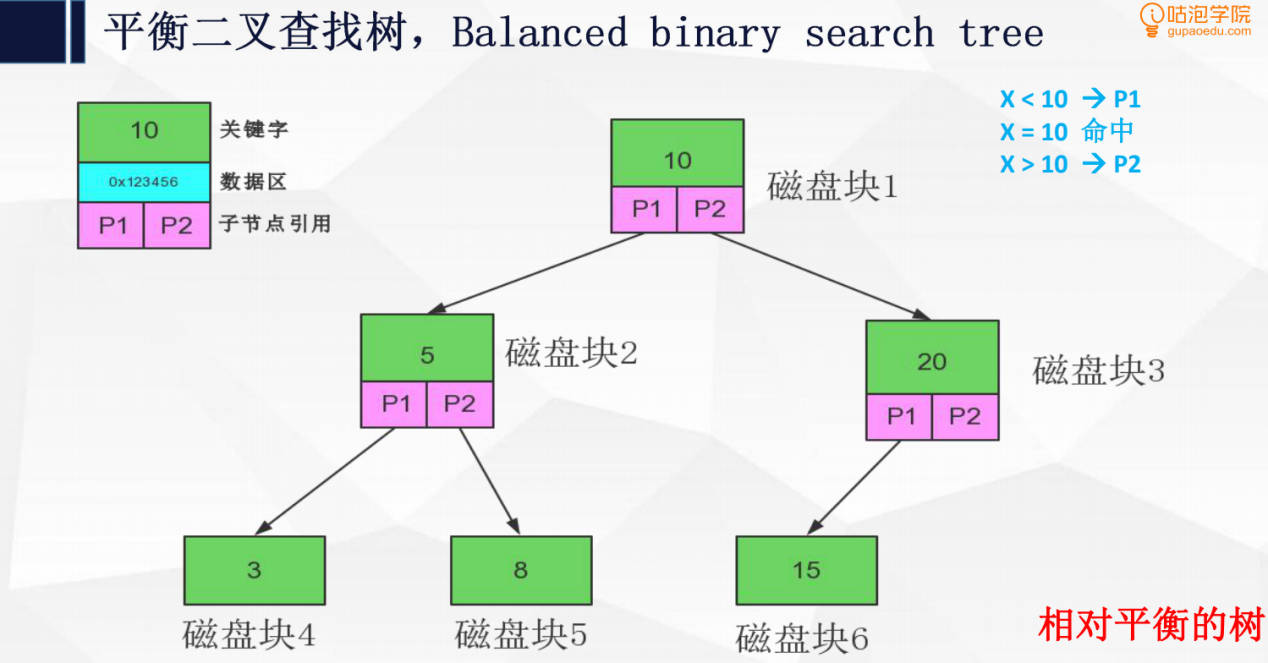
* **为什么选择B+Tree**

客观稍安勿躁，且听老夫婉婉道来

首先来一发二叉查找树，顾名思义，这就是个2X 2b,极端情况下 数据结构会变成一条线性连表，一条腿的延伸下去

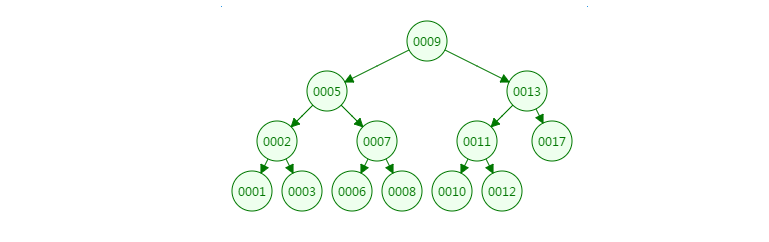
**二叉树**



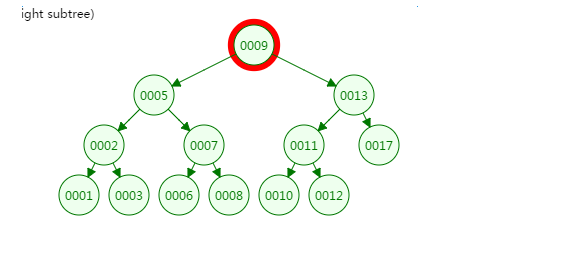


相对二叉查找树来说，这是一棵相对平衡的二叉查找树，当叶节点和跟节点高度差>1时，树结构会发生滑动改变。想体验这个过程可以去www.baidu.com操作一下。

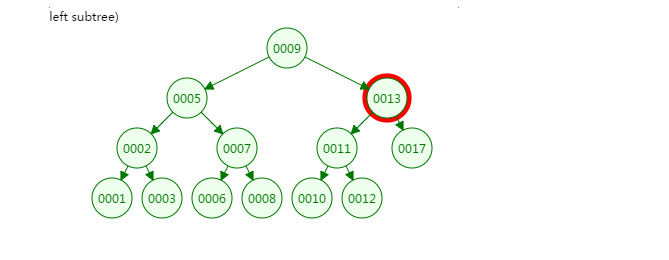
我们先来看二叉树查找时磁盘IO的次：定义一个树高为4的二叉树，查找值为10：



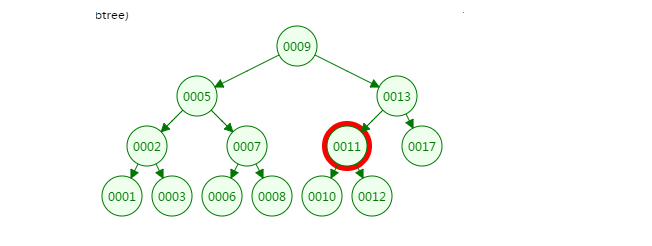
第一次磁盘IO：



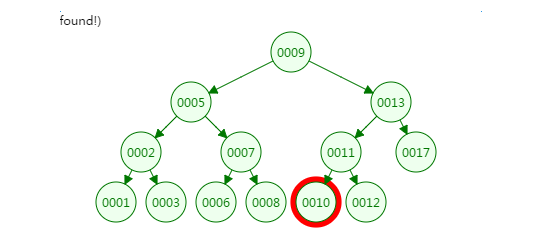
 第二次磁盘IO：



第三次磁盘IO:



第四次磁盘IO：



从二叉树的查找过程来看，树的高度和磁盘IO的次数都是4，所以最坏的情况下磁盘IO的次数由树的高度来决定，IO操作耗时大。而且每一次IO操作拿到的数据（磁盘块、节点、页）太少了，没有利用好磁盘IO的预读能力（空间局部性原理），从而带来频繁的IO操作

从前面分析情况来看，减少磁盘IO的次数就必须要压缩树的高度，让瘦高的树尽量变成矮胖的树，所以B-Tree就在这样伟大的时代背景下诞生了。

**多路平衡查找树 B-Tree**

m路B-Tree满足以下条件：

1、每个节点最多拥有m个子树

2、根节点至少有2个子树

3、分支节点至少拥有m/2颗子树（除根节点和叶子节点外都是分支节点）

4、所有叶子节点都在同一层、每个节点最多可以有m-1个key，并且以升序排列

 如下图有一个3路的B树，观察查找元素61的过程：

1. **首先吧 17 35 所在的磁盘块加载到内存，发现61 > 35，拿到p3 磁盘地址**
2. **通过p3找到65 87 所在的磁盘块加载到内存，发现61 < 65,拿到p1磁盘地址**
3. **找到13 61 所在的磁盘块加载到内存，发现 61 = 61 命中**
4. **Tree中的对比操作都是在内存中完成，耗时可以忽略不计，B-Tree中的节点中可以放很多的key,个数由路数决定**

**Ps: 数据库中有张user表，以id为int类型的主键作为索引，数据库一次IO操作加载16k数据**

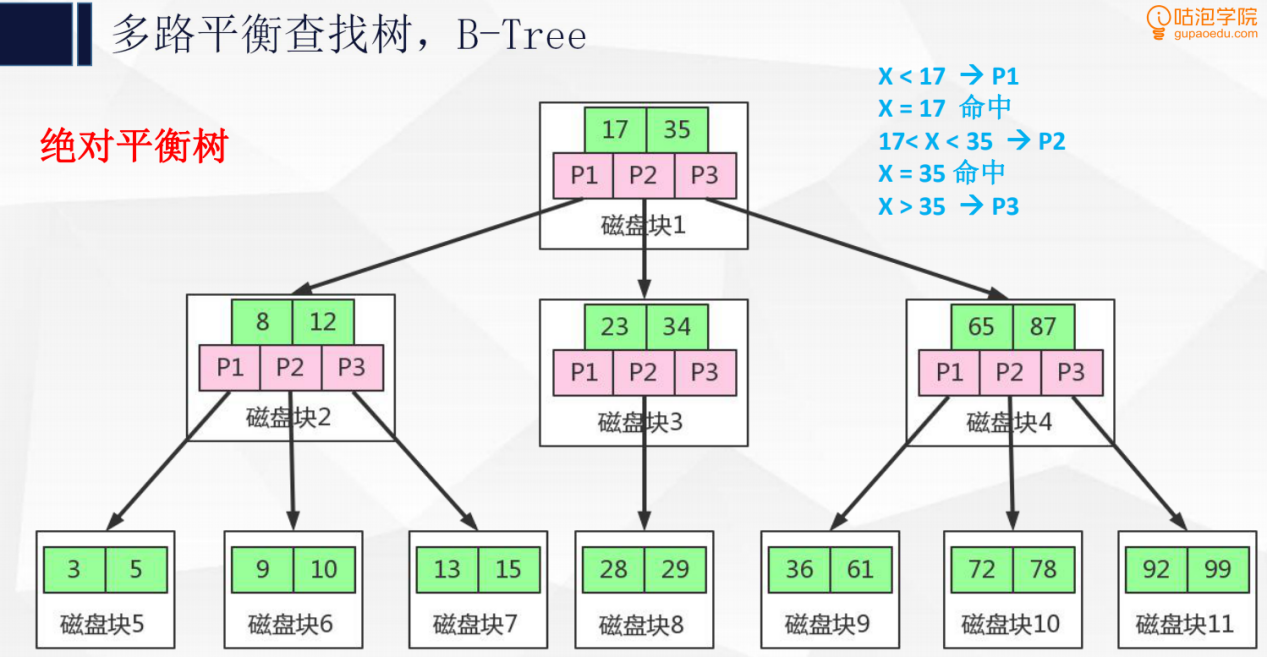
**int类型所占内存空间为4byte 加上 其他冗余 数据 姑且也认为 是4byte 共8byte**

**16k \* 1024/8 = 2048**

**也就是说数据库的一次IO操作可以加载2048条索引的数据，那么这个B-Tree就是2049路平衡查找树 比二叉树的一次加载一条索引数据**

**相同数量的key在B树中生成的节点要远远少于二叉树中的节点，相差的节点数量就等同于磁盘IO的次数。这样到达一定数量后，性能的差异就显现出来了。**

**结论： 细长 不如 矮胖**



1. Tree

B-Tree 的新增和删除操作都会使树结构发生裂变，已保证树的绝对平衡

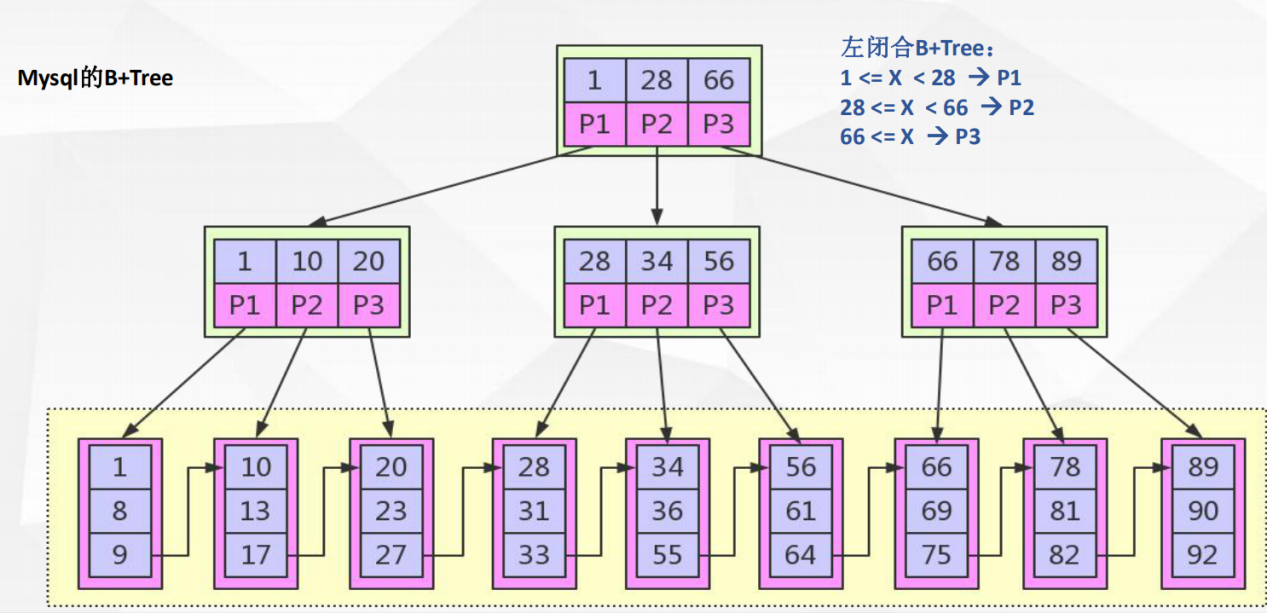
想观察过程的依然可以通过www.baidu.com去现场体验一把

因此这也从侧面论证了 为什么表中的索引不是建的越多越好，建立了索引的列发生新增修改操作时会使得原本稳定的树结构发生裂变反应。

但是：Mysql中的并没有使用B-Tree作为索引，Mysql采用的是B-Tree的一个变种（加强版）B+Tree

B树主要应用于文件系统以及部分数据库索引，如MongoDB，大部分关系型数据库索引则是使用B+树实现。

**B+Tree:加强版多路平衡查找树**



特定：

1，B+节点关键字搜索采用闭合区间

2，B+非叶节点不保存数据相关信息，只保存关键字和子节点的引用

3，B+关键字对应的数据保存在叶子节点中

4，B+叶子节点是顺序排列的，并且相邻节点具有顺序引用的关系

**为什么说****B+树比B树更适合实际应用中[操作系统](http://lib.csdn.net/base/operatingsystem" \o "操作系统知识库" \t "https://blog.csdn.net/u013400245/article/details/_blank)的文件索引和数据库索引？**

1. B+树的磁盘读写代价更低 B+Tree的内部结点并没有指向关键字具体信息（表中数据所在的磁盘位置）的指针。因此其内部结点相对B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说IO读写次数也就降低了。
2. B+树扫库、表能力强
3. B+树的排序能力更强
4. B+树的查询效率更加稳定，由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

e.数据库索引采用B+树的主要原因是 B树在提高了磁盘IO性能的同时并没有解决元素遍历的效率低下的问题。正是为了解决这个问题，B+树[应运而生](https://www.baidu.com/s?wd=%E5%BA%94%E8%BF%90%E8%80%8C%E7%94%9F&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "https://blog.csdn.net/u013400245/article/details/_blank)。B+树只要遍历叶子节点就可以实现整棵树的遍历。而且在数据库中基于范围的查询是非常频繁的，而B树不支持这样的操作（或者说效率太低）。

走进搜索引擎的作者梁斌老师针对B树、B+树给出了他的意见（为了真实性，特引用其原话，未作任何改动）： “B+树还有一个最大的好处，方便扫库，B树必须用中序遍历的方法按序扫库，而B+树直接从叶子结点挨个扫一遍就完了，B+树支持range-query非常方便，而B树不支持。这是数据库选用B+树的最主要原因。

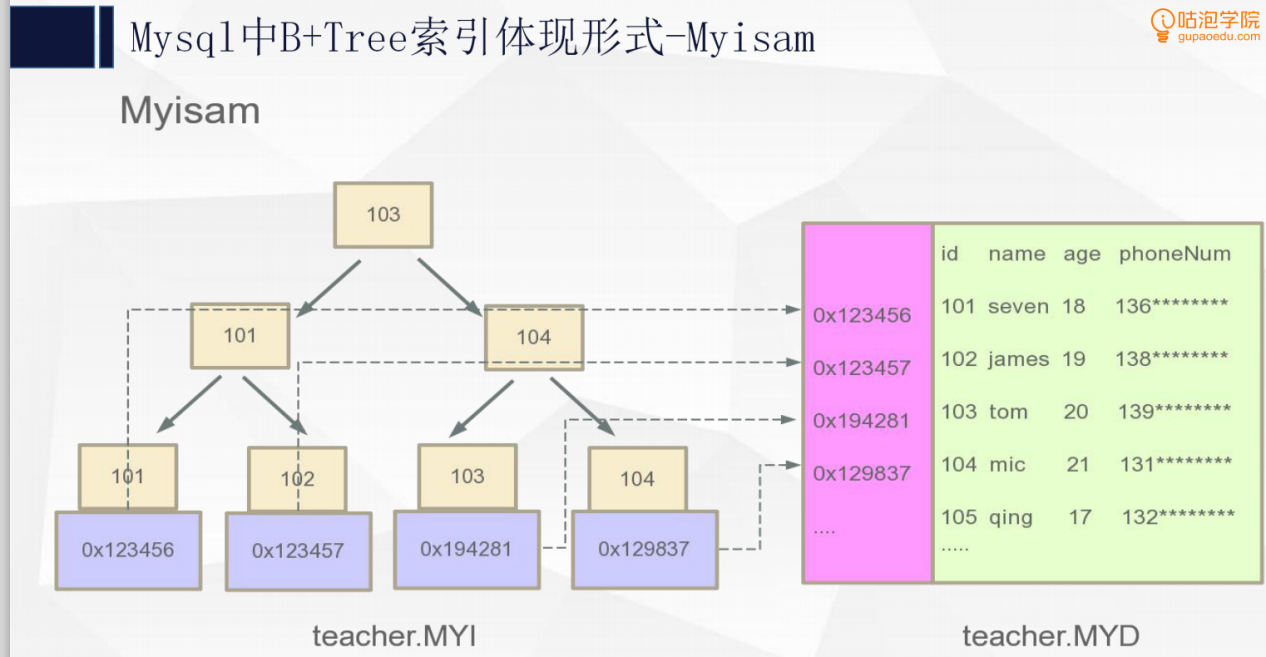
    比如要查 5-10之间的，B+树一把到5这个标记，再一把到10，然后串起来就行了，B树就非常麻烦。B树的好处，就是成功查询特别有利，因为树的高度总体要比B+树矮。不成功的情况下，B树也比B+树稍稍占一点点便宜。

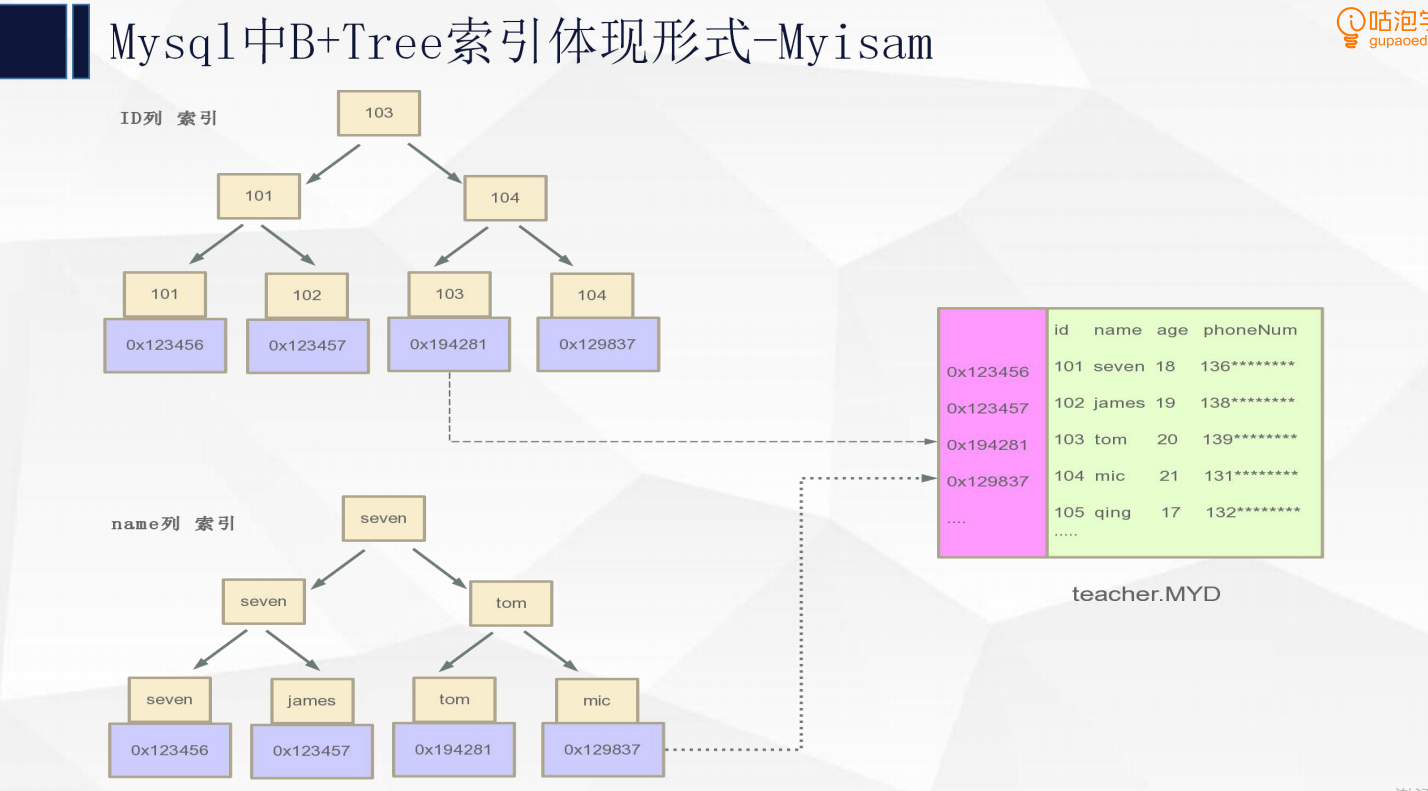
    B树比如你的例子中查，17的话，一把就得到结果了，  
有很多基于频率的搜索是选用B树，越频繁query的结点越往根上走，前提是需要对query做统计，而且要对key做一些变化。

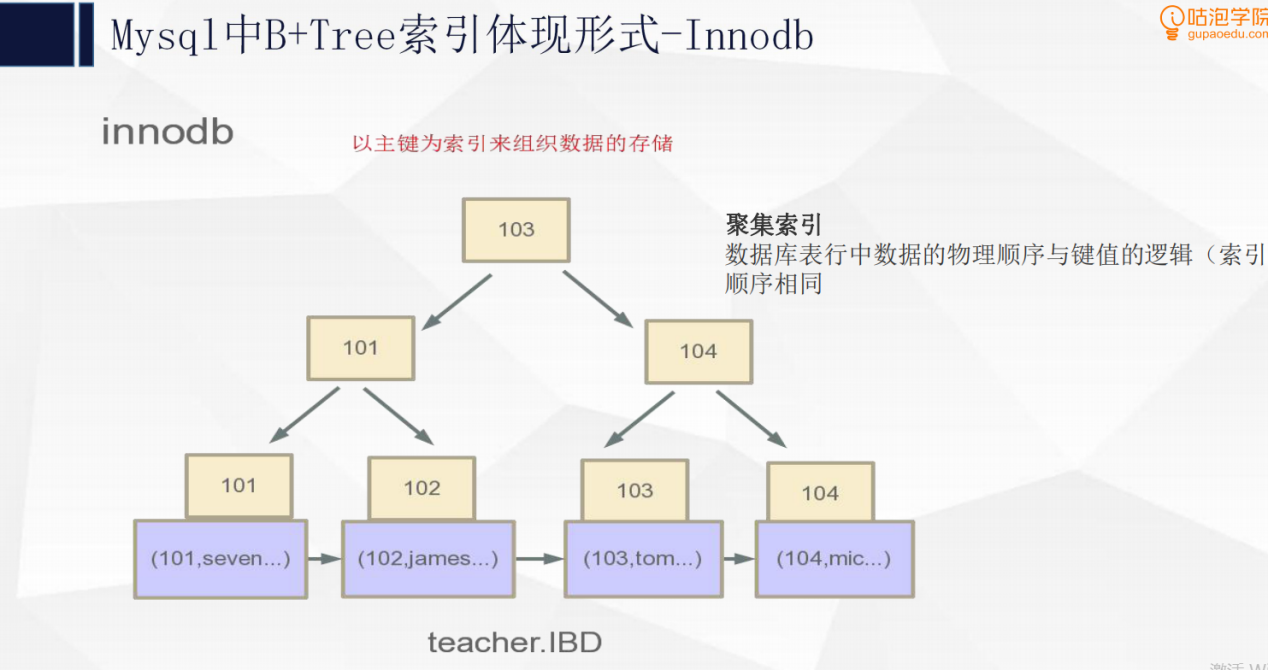
另外B树也好B+树也好，根或者上面几层因为被反复query，所以这几块基本都在内存中，不会出现读磁盘IO，一般已启动的时候，就会主动换入内存。”非常感谢。

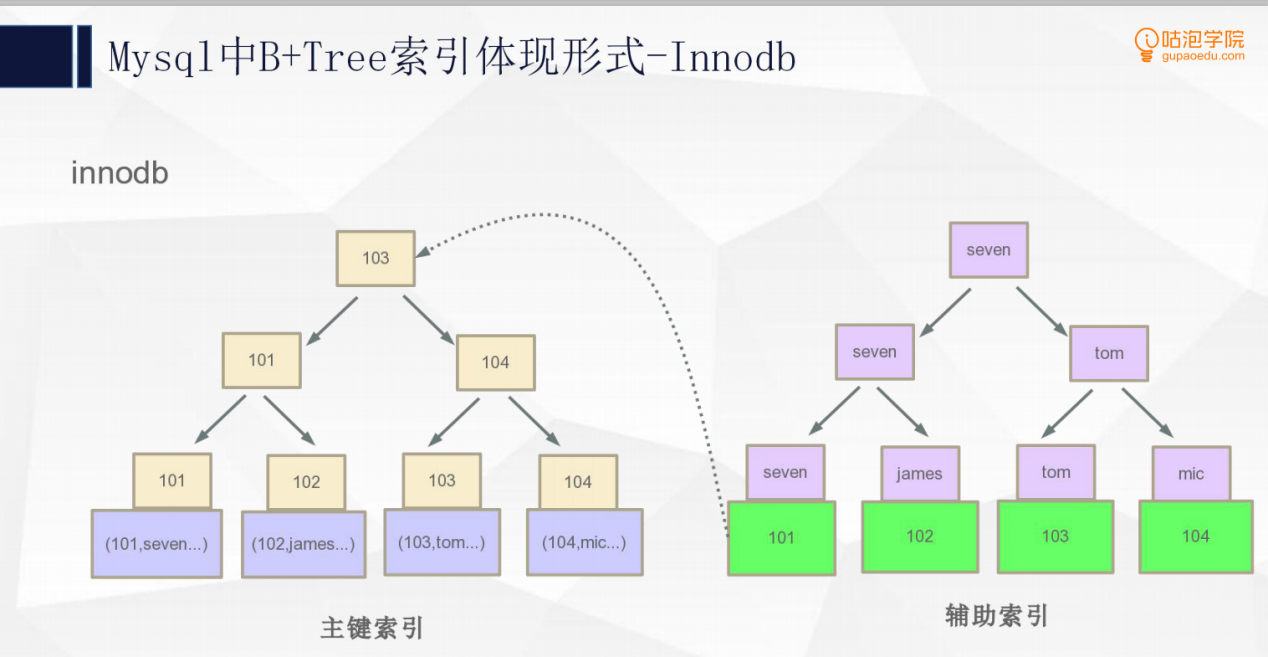
Bucket Li："mysql 底层存储是用B+树实现的，知道为什么么？内存中B+树是没有优势的，但是一到磁盘，B+树的威力就出来了"。

* **B+Tree在两大存储引擎如何体现**

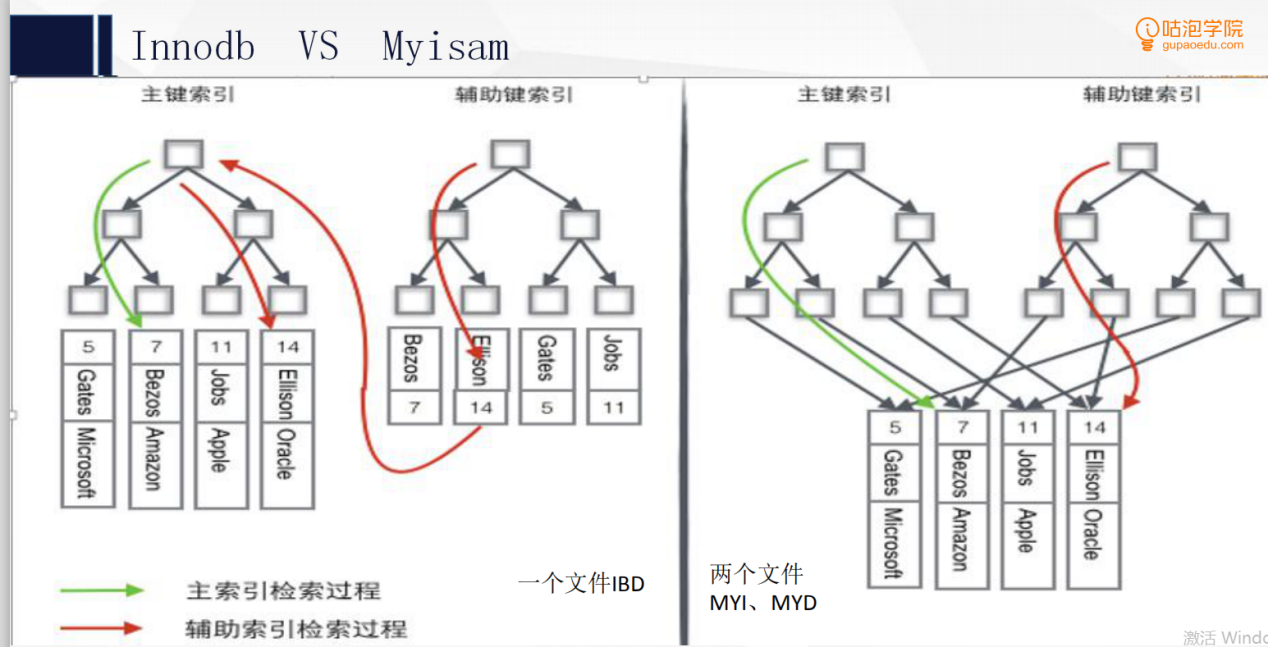






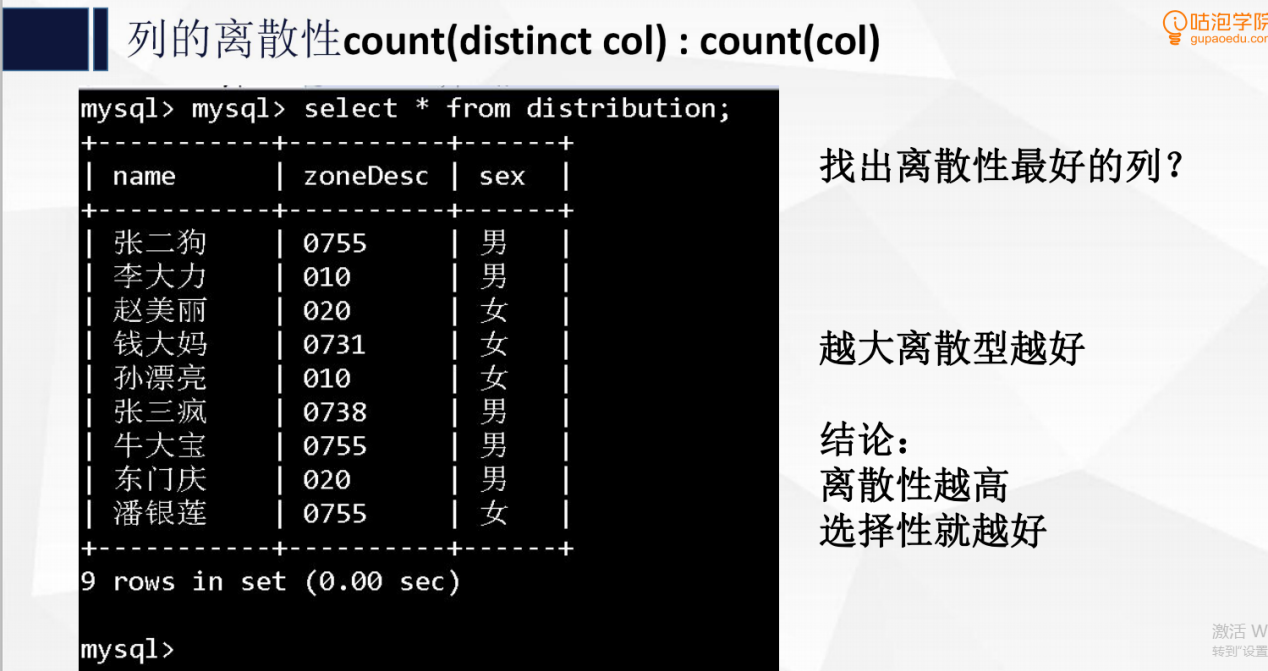


Innodb 和 Myisam的索引检索过程

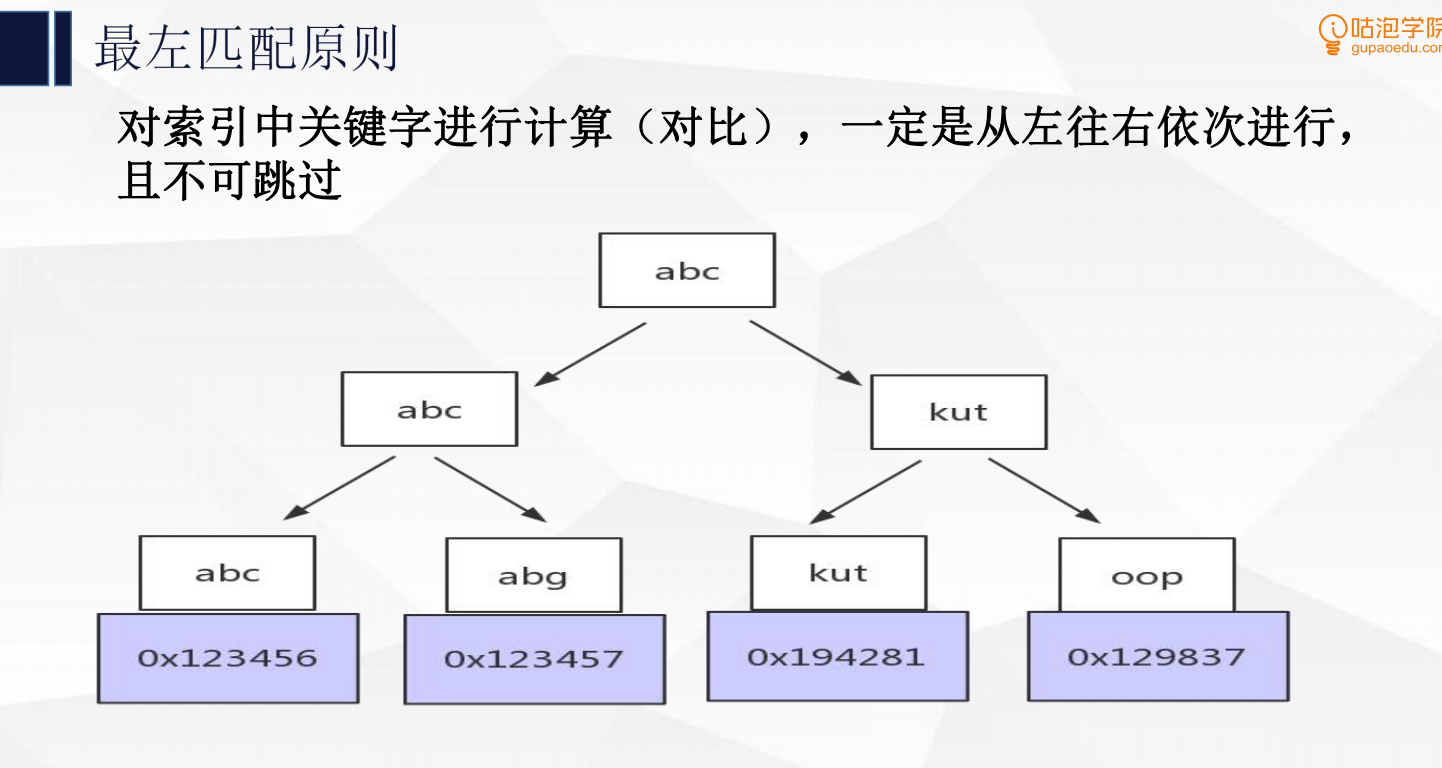


* **索引知识补充**

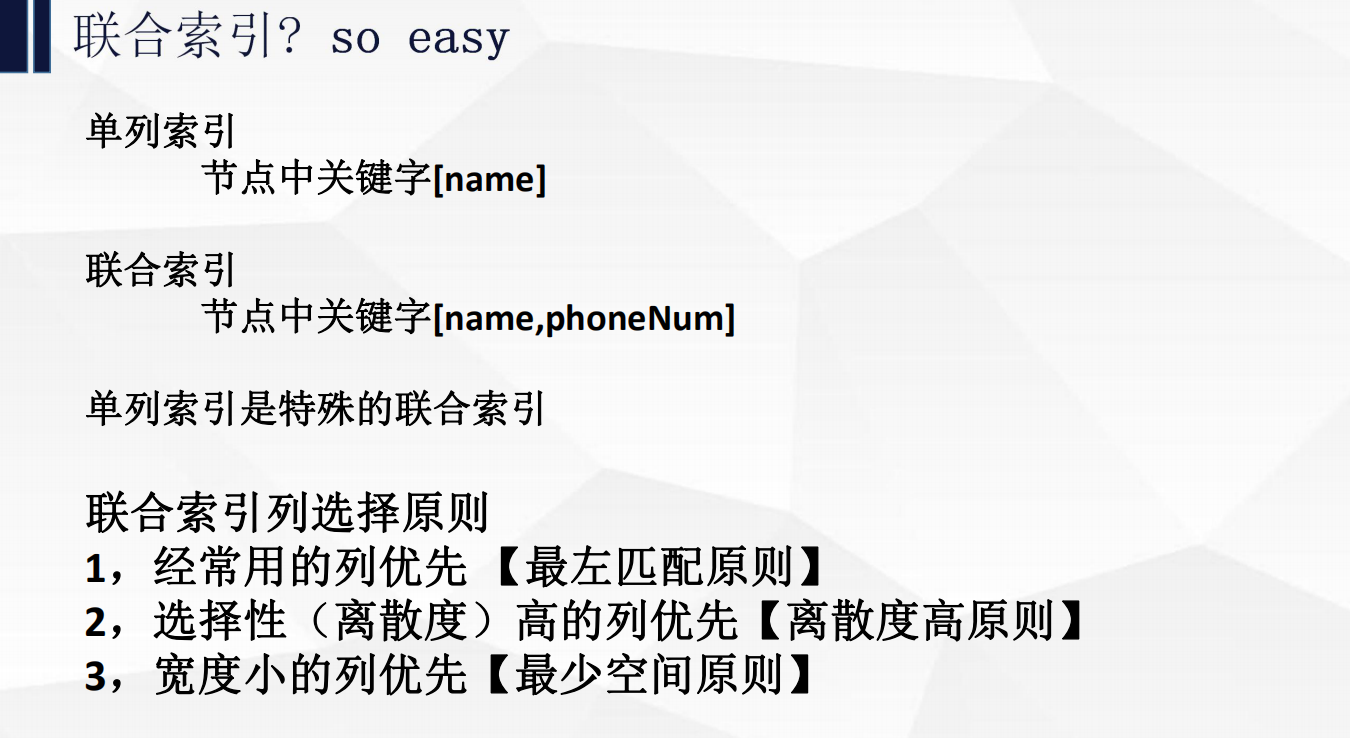
**列的离散型**



最左匹配原则



联合索引





覆盖索引

如果查询列可通过索引节点中的关键字直接返回，则该索引称之为

覆盖索引。

覆盖索引可减少数据库IO，将随机IO变为顺序IO，可提高查询性能

* **总结以及验证**

**索引列的数据长度能少则少。**

**索引一定不是越多越好，越全越好，一定是建合适的。**

**匹配列前缀可用到索引 like 9999%，like %9999%、like %9999用不到索引；**

**Where 条件中 not in 和 <>操作无法使用索引；**

**匹配范围值，order by 也可用到索引；**

**多用指定列查询，只返回自己想到的数据列，少用select \*；**

**联合索引中如果不是按照索引最左列开始查找，无法使用索引；**

**联合索引中精确匹配最左前列并范围匹配另外一列可以用到索引；**

**联合索引中如果查询中有某个列的范围查询，则其右边的所有列都无法使用索引；**

**请关注58同城30条军规**