# MySql

www.atool.org/sort.php

=====================树====================

二叉树：任何一个节点的子节点的数量不超过2

二叉树的子节点分左节点和右节点

二叉树分：满二叉树和完全二叉树

满二叉树：所有叶子节点都在最后一层（没有子节点的节点叫叶子节点），而且节点的总数为2的n（树的层数）次方-1;其实也是一颗完全二叉树

完全二叉树：所有叶子节点都在最后一层或倒数第二层，且最后一层的叶子节点在左边连续，倒数第二层的叶子节点在右边连续；

空树

左斜树

右斜树

树的遍历：前序遍历，中序遍历，后续遍历

顺序存储的二叉树

线索二叉树

二叉排序树（二叉查找数，二叉搜索树）： 删除插入不难 查找也不难（兼容了下边俩个）

概念：对于二叉树中的任何一个非叶子节点，要求左子节点比当前节点值小，右子节点比当前节点值大；空树也认为是一个二叉排序树

顺序存储 删除插入困难 查找快 可以二分查找（前提是要有序）

链式存储 ：无论是否排序 查找困难

二叉排序树要是成斜树不平衡了 其实性能反而不如单链表 性能更差（所以引出了平衡二叉树）

平衡二叉树首先必须是个二叉排序树：左子树和右子树的高度差的绝对值不超过1

计算机中数据的存储原理（多路查找树）

为什么文件系统或者数据库系统会使用多路查找树（B树和B+树） 以上的树适合在内存中应用（数据量大节点会很多，树高度也高，IO会频繁，这也是为什么我们要在文件系统或数据库系统中使用B树） 数据量大要用这种树

即使固态硬盘也远远低于内存速度

2-3树和2-3-4树（也是B树） 前提也是二叉排序的创建规则貌似

B树中所有的叶子节点都在同一层

2-3树：有俩个子节点的节点叫二节点；有三个子节点的节点叫三节点

二节点要么有俩个子节点，要么没有子节点

三节点要么有三个子节点，要么没有子节点

以此类推2-3-4树

B树和B+树

B树的阶

2-3树是3阶的B树

2-3-4树是4阶的B树

B+树是B树的变形：非节点只存储索引信息，不存储数据；叶子节点最右边的指针指向下一个相邻的叶节点；所有的叶节点组成了一个有序链表

B树数据索引都在一起

mysql俩种引擎 B树和B+树

## 为什么MySQL数据库索引选择使用B+树？

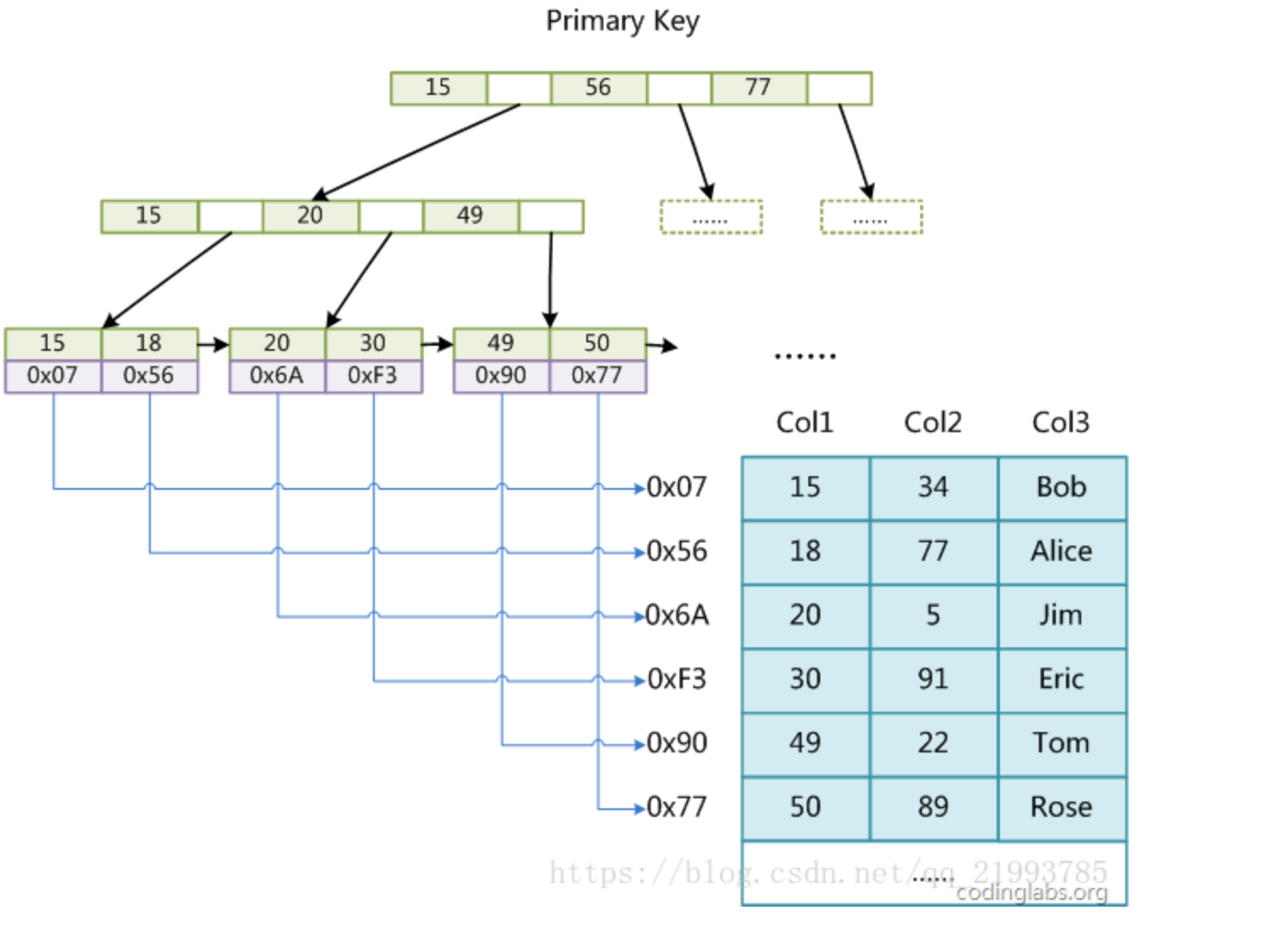
<https://blog.csdn.net/bntX2jSQfEHy7/article/details/79453193>

## mysql索引底层实现原理

<https://www.cnblogs.com/boothsun/p/8970952.html#autoid-5-0-0>

## MyISAM索引实现

MyISAM引擎使用B+Tree作为索引结构，叶子节点的权存放的数据记录的地址，下图是MyISAM索引的原理图：



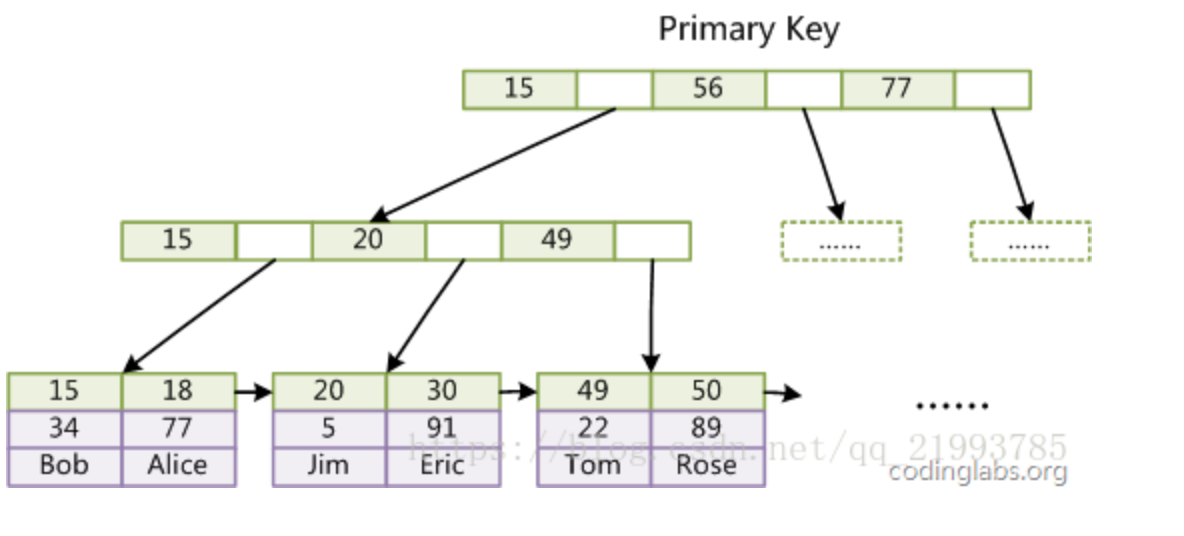
这里设表一共有三列，假设我们以Col1为主键，则上图是一个MyISAM表的主索引（Primary key）示意。可以看出MyISAM的索引文件仅仅保存数据记录的地址。在MyISAM中，主索引和辅助索引（Secondary key）在结构上没有任何区别，只是主索引要求key是唯一的，而辅助索引的key可以重复；

MyISAM的索引方式也叫做“非聚集”的，之所以这么称呼是为了与InnoDB的聚集索引区分。

## InnoDB索引实现

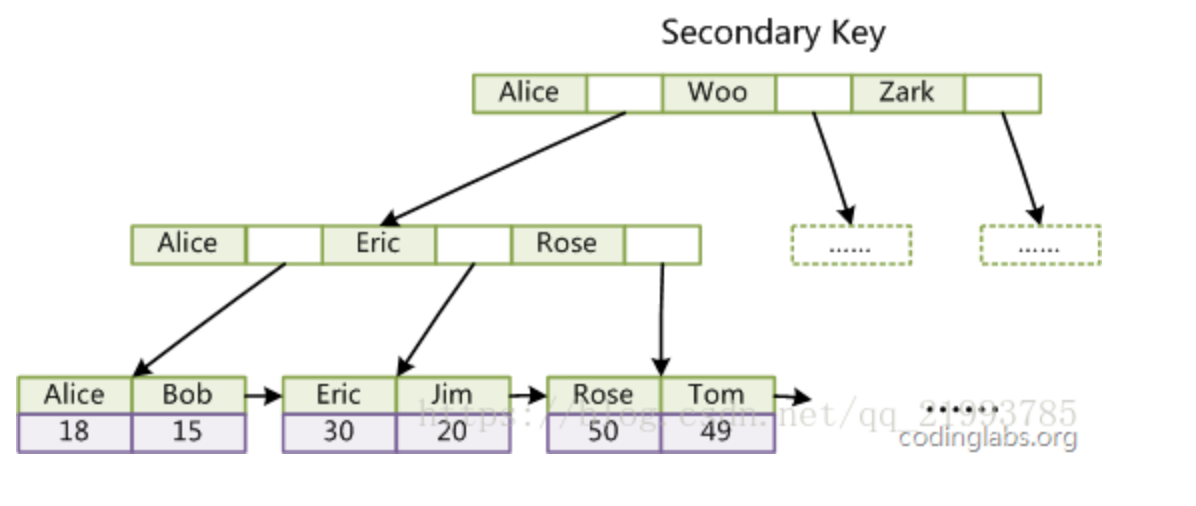
 虽然InnoDB也使用B+Tree作为索引结构，但具体实现方式却与MyISAM截然不同。

       第一个重大区别是InnoDB的数据文件本身就是索引文件。从上文知道，MyISAM索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。而在InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。

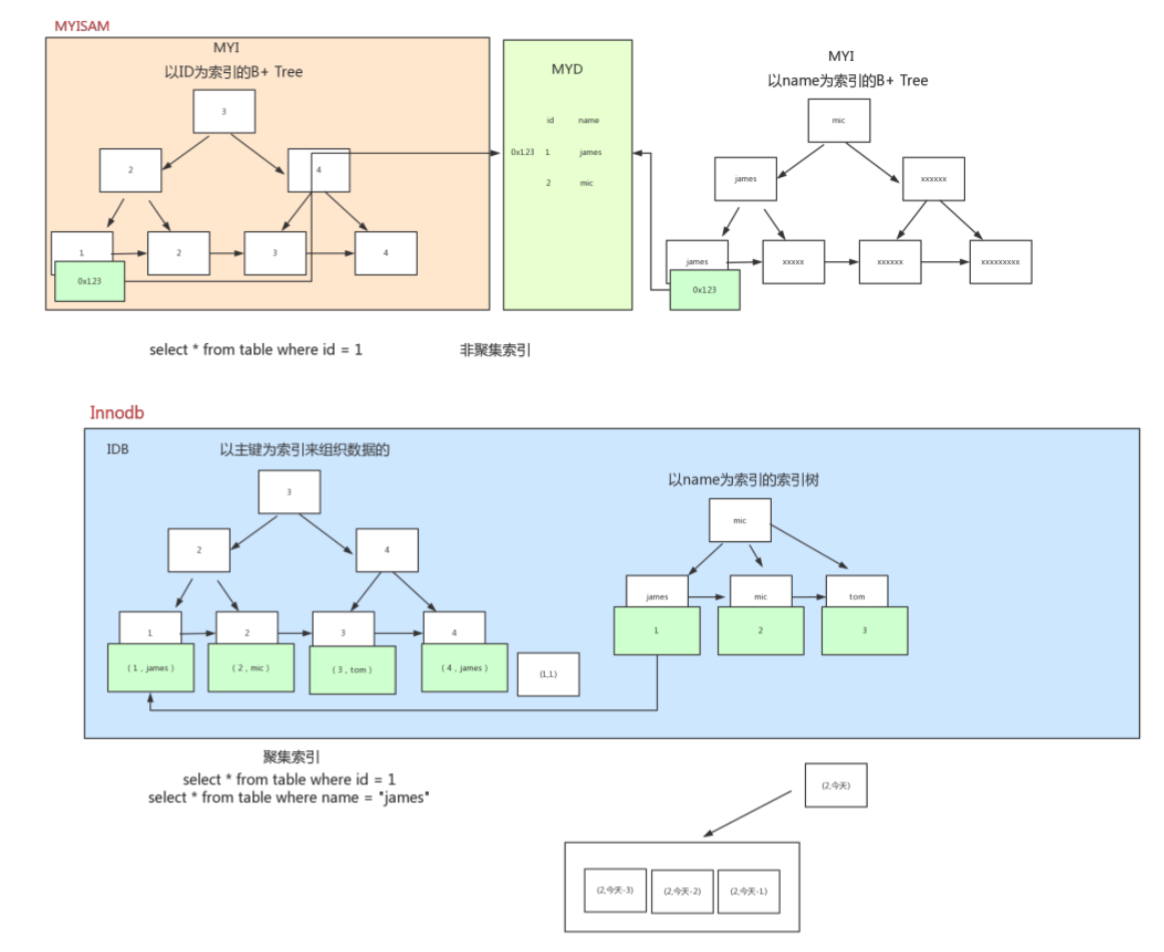


上图是InnoDB主索引（同时也是数据文件）的示意图，可以看到叶节点包含了完整的数据记录。这种索引叫做聚集索引。因为InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键（MyISAM可以没有），如果没有显式指定，则MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。

       第二个与MyISAM索引的不同是InnoDB的辅助索引data域存储相应记录主键的值而不是地址。换句话说，InnoDB的所有辅助索引都引用主键作为data域。



聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录



了解不同存储引擎的索引实现方式对于正确使用和优化索引都非常有帮助，例如知道了InnoDB的索引实现后，就很容易明白为什么不建议使用过长的字段作为主键，因为所有辅助索引都引用主索引，过长的主索引会令辅助索引变得过大。再例如，用非单调的字段作为主键在InnoDB中不是个好主意，因为InnoDB数据文件本身是一颗B+Tree，非单调的主键会造成在插入新记录时数据文件为了维持B+Tree的特性而频繁的分裂调整，十分低效，而使用自增字段作为主键则是一个很好的选择。

## Linux epoll模型理解

<https://www.cnblogs.com/hepingqingfeng/p/6656073.html>

<https://blog.csdn.net/men_wen/article/details/53456491>

<https://www.jianshu.com/p/8c368129c658>

## Kafka高可用很好的一篇文档

<https://www.cnblogs.com/qingyunzong/p/9004703.html>

## zookeeper实现分布式锁的俩种方式

<https://www.cnblogs.com/linjiqin/p/6052031.html>

* 节点名称唯一性：多个客户端创建一个节点，只有成功创建节点的客户端才能获得锁
* 临时顺序节点：所有客户端在某个目录下创建自己的临时顺序节点，只有序号最小的才获得锁

## redis高可用

vip方案：<https://www.jianshu.com/p/c2ab606b00b7>

此文章虽然实现了高可用，但是木桶原理 数据大小限制肯定是由于最小的内存决定的 ，所以最终数据分片是最好的

## Netty原理分析（太帅了）

<https://blog.csdn.net/suifeng3051/article/details/28861883>

<https://blog.csdn.net/jiiiang400/article/details/77936122>

## jvm笔记

<https://blog.csdn.net/weixin_39924752/article/details/80228719>

## reactor模式底层分析

<https://www.cnblogs.com/winner-0715/p/8733787.html>