# 深入理解MYSQL索引之B+TREE

2019-01-13 21:17:25 [he\_321](https://me.csdn.net/b_x_p" \t "https://blog.csdn.net/b_x_p/article/details/_blank) 阅读数 6455更多

分类专栏： [数据库](https://blog.csdn.net/b_x_p/article/category/7260546" \t "https://blog.csdn.net/b_x_p/article/details/_blank)

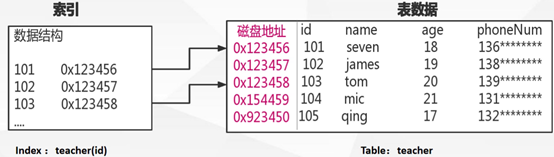
版权声明：本文为博主原创文章，遵循[CC 4.0 BY-SA](http://creativecommons.org/licenses/by-sa/4.0/" \t "https://blog.csdn.net/b_x_p/article/details/_blank)版权协议，转载请附上原文出处链接和本声明。

本文链接：<https://blog.csdn.net/b_x_p/article/details/86434387>

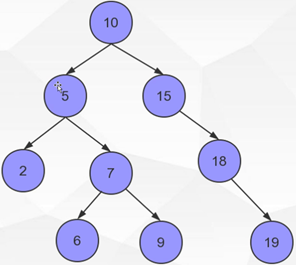
首先，正确的创建合适的索引，是提升数据库查询性能的基础。

****索引是什么？****

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。

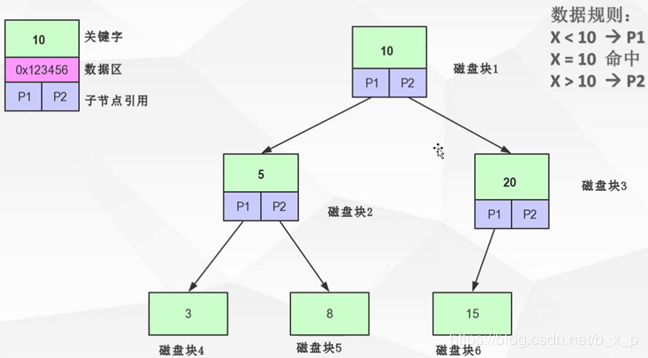
****索引的工作机制是怎样的？****  


如上图中，如果现在有一条sql语句 select \* from teacher where id = 101，如果没有索引的条件下，我们要找到这条记录，我们就需要就行全表扫描，匹配id = 101的数据。如果有了索引，我们就可以快速的通过索引找到101所对应的行记录在磁盘中的地址，再根据给定的地址取出对应的行数据。

****MYSQL数据库为什么要使用B+TREE作为索引的数据结构？****  
对数据的加速检索，首先想到的就是二叉树，二叉树的查找时间复杂度可以达到O(log2(n))。下面看一下二叉树的存储结构：  


二叉树搜索相当于一个二分查找。二叉查找能大大提升查询的效率，但是它有一个问题：二叉树以第一个插入的数据作为根节点，如上图中，如果只看右侧，就会发现，就是一个线性链表结构。如果我们现在的数据只包含1, 2, 3, 4，5, 6，就会出现一下情况：  


如果我们要查询的数据为6则需要遍历所有的节点才能找到6，即，相当于全表扫描，就是由于存在这种问题，所以二叉查找树不适合用于作为索引的数据结构。

基于这样的推演，为了解决存在线性链表的问题，很容易就能够想到平衡二叉查找树。下面看看平衡二叉树是怎样的：  
  
平衡二叉查找树定义为：节点的子节点高度差不能超过1,如上图中的节点20，左节点高度为1，右节点高度0，差为1，所以上图没有违反定义，他就是一个平衡二叉树。保证二叉树平衡的方式为左旋，右旋等操作，至于如果左旋右旋，可以自行去搜索相关的知识。

如果上图中平衡二叉树保存的是id索引，现在要从id = 8的数据，首先要把根节点加载进内存，用8和10进行比较，发现8比10小，继续加载10的左子树。把5加载进内存，用8和5比较，同理，加载5节点的右子树。此时发现命中，现在要加载id为8的索引对应的数据。

****怎么找到索引对应的数据呢？****

索引保存数据的方式一般有两种，第一种为在节点的数据区保存id = 8的行数据的所有数据具体内容。另外一种方式，数据区保存的是真正保存数据的磁盘地址。

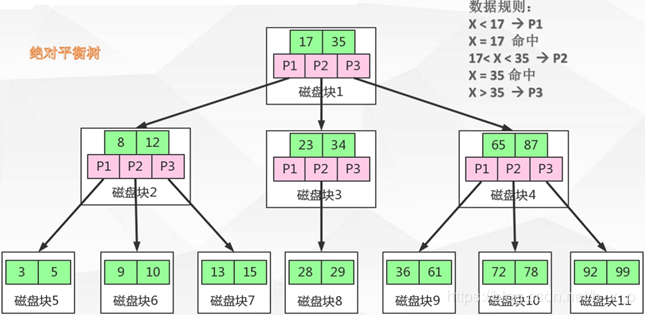
到这里，平衡二叉树解决了存在线性链表的问题，数据查询的效率好像也还可以，基本能达到O(log2(n))， ****那为什么mysql不选择这样的数据结构呢，他又存在什么样的问题呢？****

****问题1：**** 搜索效率不足，一般来说，在树结构中，数据所处的深度，决定了搜索时的IO次数。如上图中搜索id = 8的数据，需要进行3次IO。当数据量到达几百万的时候，树的高度就会很恐怖。

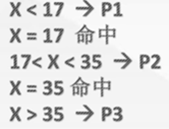
****问题2：**** 查询不不稳定，如果查询的数据落在根节点，只需要一次IO，如果是叶子节点或者是支节点，会需要多次IO才可以。

****问题3：**** 节点存储的数据内容太少。没有很好利用操作系统和磁盘数据交换特性，也没有利用好磁盘IO的预读能力。因为操作系统和磁盘之间一次数据交换是已页为单位的，一页 = 4K，即每次IO操作系统会将4K数据加载进内存。但是，在二叉树每个节点的结构只保存一个关键字，一个数据区，两个子节点的引用，并不能够填满4K的内容。幸幸苦苦做了一次的IO操作，却只加载了一个关键字，在树的高度很高，恰好又搜索的关键字位于叶子节点或者支节点的时候，取一个关键字要做很多次的IO。

****那有没有一种结构能够解决二叉树的这种问题呢？****

有，多路平衡查找树：（Balance Tree）：  
B Tree 是一个绝对平衡树，所有的叶子节点在同一高度，如下图所示：  
  
****B TREE有什么优势，又是怎么去解决一些问题的呢？****

先看定义，上图为一个2-3树（每个节点存储2个关键字，有3路），多路平衡查找树也就是多叉的意思，从上图中可以看出，每个节点保存的关键字的个数和路数关系为：  
关键字个数 = 路数 – 1。

假设要从上图中去寻找id = 28的数据，B TREE 搜索过程如下：  
首先把根节点加载进内存，加载了17,35两个关键字，判断规则为：  


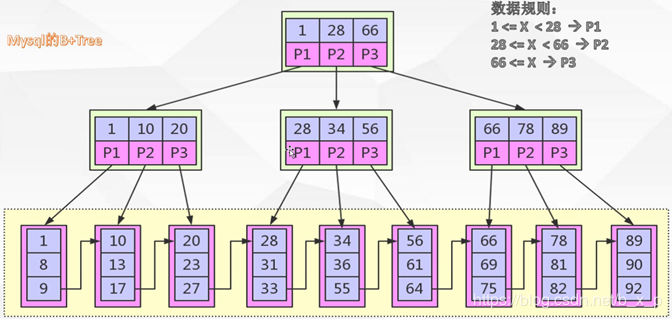
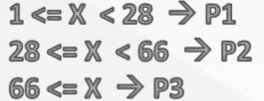
根据以上规则命中28后，接下来加载28对应的数据， 就去找28对应的数据区，数据区中存储的是具体的数据或者是指向数据的指针。

****为什么说这种结构能够解决平衡二叉树存在的问题呢？****

能够很好的利用操作系统和磁盘的交互特性， MYSQL为了很好的利用磁盘的预读能力，将页大小为16K，即将一个节点（磁盘块）的大小设置为16K，一次IO将一个节点（16K）内容加载进内存。这里，假设关键字类型为 int，即4字节，若每个关键字对应的数据区也为4字节，不考虑子节点引用的情况下，则上图中的每个节点大约能够存储（16 \* 1000）/ 8 = 2000个关键字，则共2001个路数。对于二叉树，三层高度，最多可以保存7个关键字，而对于这种有2001路的B树，三层高度能够搜索的关键字个数远远的大于二叉树。

在B TREE保证树的平衡的过程中，每次关键字的变化，都会导致结构发生很大的变化，这个过程是特别浪费时间的，所以创建索引一定要创建合适的索引，而不是把所有的字段都创建索引，创建冗余索引只会在对数据进行新增，删除，修改时增加性能消耗。

****既然B树已经很好的解决了问题，为什么MYSQL还要用B+TREE？****

先看看B+TREE是怎样的，B+TREE是B TREE的一个变种，在B+树种，B树种的路数和关键字的个数的关系不再成立了，B+TREE中，数据检索规则采用的是左闭合区间，路数和关键个数关系为1比1，具体如下图所示：  
  
如果上图中是用ID做的索引，如果是搜索id = 1的数据，搜索规则如下：  


根据如上规则，最终在叶子节点中命中数据，根据叶子节点中节点1的数据区取得真正的数据。

****B TREE和B+TREE区别是什么？****

1、B+TREE 关键字的搜索采用的是左闭合区间，之所以采用左闭合区间是因为他要最好的去支持自增id，这也是mysql的设计初衷。即，如果id = 1命中，会继续往下查找，直到找到叶子节点中的1。

2、B+TREE 根节点和支节点没有数据区，关键字对应的数据只保存在叶子节点中。即只有叶子节点中的关键字数据区才会保存真正的数据内容或者是内容的地址。而在B树种，如果根节点命中，则会直接返回数据。并且在B+TREE中，叶子节点不会去保存子节点的引用。

3、B+TREE叶子节点是顺序排列的，并且相邻的节点具有顺序引用的关系，如上图中叶子节点之间有指针相连接。

****MYSQL为什么最终要去选择B+TREE？****

1、B+TREE是B TREE的变种，B TREE能解决的问题，B+TREE也能够解决（降低树的高度，增大节点存储数据量）

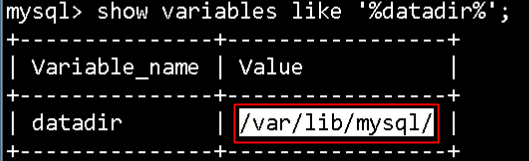
2、 B+TREE扫库和扫表能力更强，如果我们要根据索引去进行数据表的扫描，对B TREE进行扫描，需要把整棵树遍历一遍，而B+TREE只需要遍历他的所有叶子节点即可（叶子节点之间有引用）。

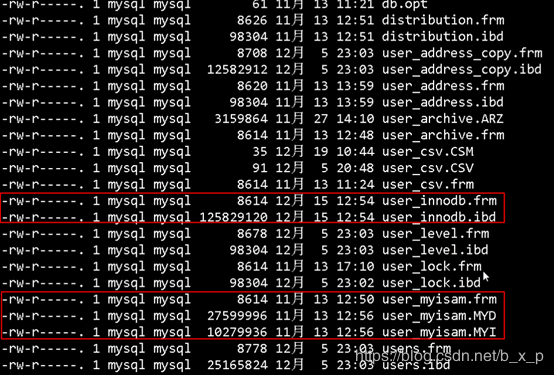
3、B+TREE磁盘读写能力更强，他的根节点和支节点不保存数据区，所有根节点和支节点同样大小的情况下，保存的关键字要比B TREE要多。而叶子节点不保存子节点引用。所以，B+TREE读写一次磁盘加载的关键字比B TREE更多。

4、B+TREE排序能力更强，如上面的图中可以看出，B+TREE天然具有排序功能。

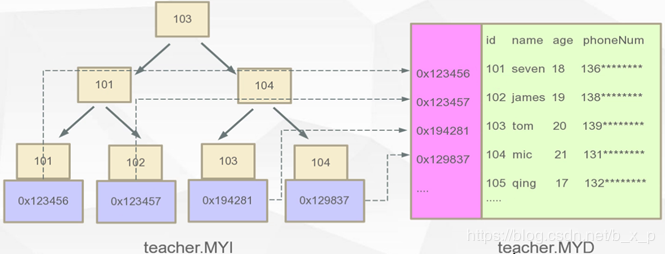
5、B+TREE查询效率更加稳定，每次查询数据，查询IO次数一定是稳定的。当然这个每个人的理解都不同，因为在B TREE如果根节点命中直接返回，确实效率更高。

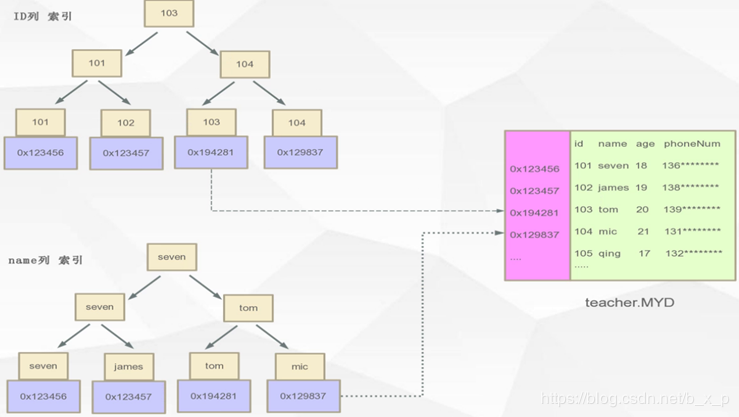
****MYSQL B+TREE具体落地形式****

这里主要讲解的是MYSQL根据B+TREE索引结构不同的两种存储引擎（MYISAM 和 INNODB）的实现，首先找到MYSQL保存数据的文件夹，看看mysql是如何保存数据的：  


进入到这个目录下，这个目录下保存的是所有数据库，再进入到具体的一个数据库目录下。在这里，有多种数据的存储引擎，这里讲解MYISAM和innodb，如图中所示：  
  
MYISAM存储引擎索引：  
从图中可以看出，使用MYISAM存储引擎存储数据库数据，一共有三个文件：  
Frm，表的定义文件。MYD：数据文件，所有的数据保存在这个文件中。MYI：索引文件。

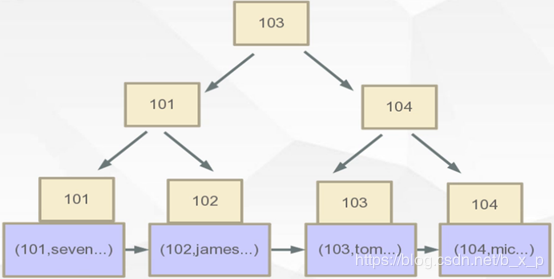
在MYISAM存储引擎中，数据和索引的关系如下：

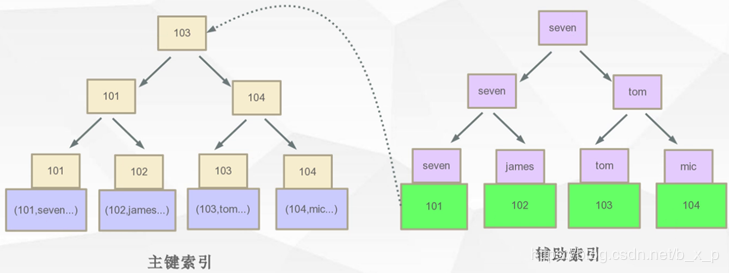
  
如何查找数据的呢？如果要查询id = 101的数据，先根据MYI索引文件（如上图左）去找id = 101的节点，通过这个节点的数据区拿到真正保存数据的磁盘地址，再通过这个地址从MYD数据文件（如上图右）中加载对应的记录。

如果有多个索引，表现形式如下：  
  
所以在MYISAM存储引擎中，主键索引和辅助索引是同级别的，没有主次之分。

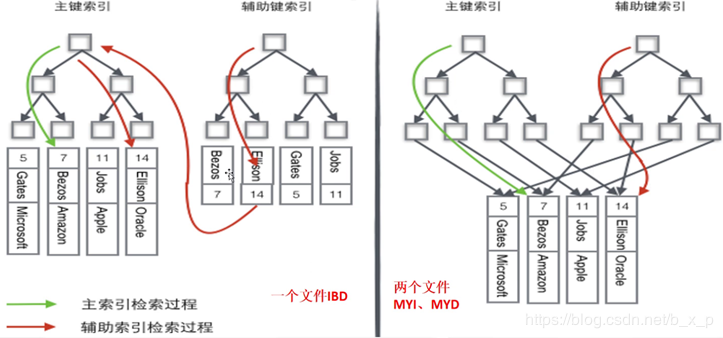
Innodb存储引擎：  
首先看一下聚集索引的概念，聚集索引定义为：数据库表行中数据的物理顺序和键值的逻辑顺序相同。

Innodb以主键为索引来聚集组织数据的存储，下面看看Innodb是如何组织数据的。

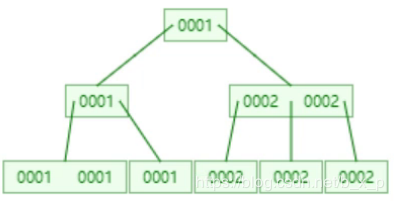
Innodb只有两个文件，Frm文件： 表的定义文件，和Ibd文件，没有专门保存数据的文件。数据以主键进行聚集存储，把真正的数据保存在叶子节点中。innodb设计初衷认为主键才是最主要的索引。具体如下图所示：  
  
如上图中，叶子节点的数据区保存的就是真实的数据，在通过索引进行检索的时候，命中叶子节点，就可以直接从叶子节点中取出行数据。mysql5.5版本之前采用的是MYISAM引擎，5.5之后采用的是innodb引擎。

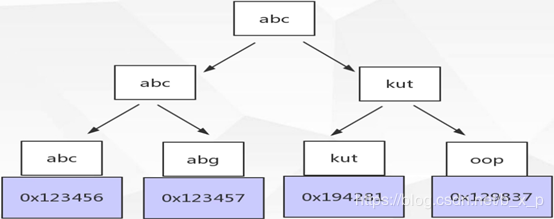
在innodb中，辅助索引的格式如下图所示？  
  
如上图，****主键索引的叶子节点保存的是真正的数据。而辅助索引叶子节点的数据区保存的是主键索引关键字的值****。搜索过程为：假如要查询name = seven的数据，先在辅助索引中查询最后找到主键id = 101，再在主键索引中搜索id为101的数据，最终在主键索引的叶子节点中获取到真正的数据。所以通过辅助索引进行检索，需要检索两次索引。

把Innodb 和 MYISAM区别放在一张图中看，就如下所示：

  
****创建索引的几大原则：****

1、列的离散型：  
离散型的计算公式：count(distinct col):count(col)，离散型越高，选择型越好。  
如下表中各个字段，哪一列的离散型最好：  
  
上图中，显然可以看出，name的离散型最好，如果用sex创建索引：

为什么说离散型越高，选择型越好？  
如下图，如果对Sex创建索引，则索引结构将会如下：  
  
如果此时检索 sex = 1的数据，根节点判断的时候，结果是查询左子树，但是当在左子树第二层再进行判断的时候，因为左右分支都满足条件，所以很难抉择选择哪一个分支继续搜索，或者是把两个分支同时进行搜索。

2、最左匹配原则  
对于索引中的关键字进行对比的时候，一定是从左往右以此对比，且不可跳过。之前讲解的id都为int型数据，如果id为字符串的时候，如下图：  
  
当进行匹配的时候，会把字符串转换成ascll码，如abc变成97 98 99，然后从左往右一个字符一个字符进行对比。所以在sql查询中使用like %a 时候索引会失效，因为%表示全匹配，如果已经全匹配就不需要索引，还不如直接全表扫描。

3、最少空间原则  
前面已经说过，当关键字占用的空间越小，则每个节点保存的关键字个数就越多，每次加载进内存的关键字个数就越多，检索效率就越高。

****联合索引：****

单列索引：节点中的关键字[name]  
联合索引：节点中的关键字[name, phoneNum]  
可以把单列索引看成特殊的联合索引，联合索引的比较也是根据最左匹配原则。  
联合索引列的选择原则：  
（1） 经常用的列优先（最左匹配原则）  
（2） 离散度高的列优先（离散度高原则）  
（3） 宽度小的列优先，（最少空间原则）

下面简单举例平时经常会遇到的问题：  
如，平时经常使用的查询sql如下：  
Select \* from users where name = ?  
Select \* from users where name = ? and pahoneNum = ?

为了加快检索速度，为上面的查询sql创建索引如下：  
Create index idx\_name on users(name)  
Create index idx\_name\_phoneNum on users(name, phoneNum)

在上面解决方案中，根据最左匹配原则，idx\_name为冗余索引， where name = ?同样可以利用索引idx\_name\_phoneNum进行检索。冗余索引会增减维护B+TREE平衡时的性能消耗，并且占用磁盘空间。

****覆盖索引：****

如果查询的列，通过索引项的信息可直接返回，则该索引称之为查询SQL的覆盖索引。覆盖索引可以提高查询的效率。  
下面通过例子说明覆盖索引。  
表：teacher  
索引：PK（id）， key（name， phoneNum）， unique（teacherNo）

下面哪些sql使用到了覆盖索引？

Select teacherNo from teacher where teacherNo = ?：使用到了，检索到teacherNo 时候，可以直接将索引中的teacherNo 值返回，不需要进入数据区。

Select id，teacherNo from teacher where teacherNo = ?：使用到了，辅助索引的叶子节点保存了主索引的值，所以检索到辅助索引的叶子节点的时候就可以之间返回id。

Select name，phoneNum from teacher where teacherNo = ?：没有用到

Select phoneNum from teacher where name = ?, 使用到了。

知道了覆盖索引，就知道了为什么sql中要求尽量不要使用select \*，要写明具体要查询的字段，一个原因就是，这样在使用到覆盖索引的情况下，不需要进入到数据区，数据就能直接返回，提升了查询效率。

****通过前面的学习，我们可以很容易的明白如下一下结论：****

1、索引列的数据长度满足业务的情况下能少则少。

2、表中的索引并不是越多越好。

3、Where 条件中，like 9%， like %9%， like%9，三种方式都用不到索引。后两种方式对于索引是无效的。第一种9%是不确定的，决定于列的离散型，结论上讲可以用到，如果发现离散情况特别差的情况下，查询优化器觉得走索引查询性能更差，还不如全表扫描。

4、Where条件中 NOT IN 无法使用索引

5、多用指定查询，只返回自己想要的列，少用select \*。

6、查询条件中使用函数，索引将会失效，这和列的离散型有关，一旦使用到函数，函数具有不确定性。

7、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。

8、对联合索引精确匹配最左前列并范围匹配另一列，可以使用到索引。

9、联合索引中，如果查询有某个列的范围查询，其右边所有的列都无法使用索引。