问北

目录

梦里都是你 『欢迎关注公众号：编程大道』

MySQL索引那些事

原文链接

大家有没有遇到过慢查询的情况，执行一条SQL需要几秒，甚至十几、几十秒的时间，这时候DBA就会建议你去把查询的 SQL 优化一下，怎么优化？你能想到的就是加索引吧？

为什么加索引就查的快了？这就要从索引的本质以及他的底层原理说起。

回到顶部

索引是什么

那索引到底是什么呢?你是不是还停留在大学学『数据库原理』时老师讲的“索引就像字典的目录”这样的概念？老师讲的没错，但没有深入去讲。

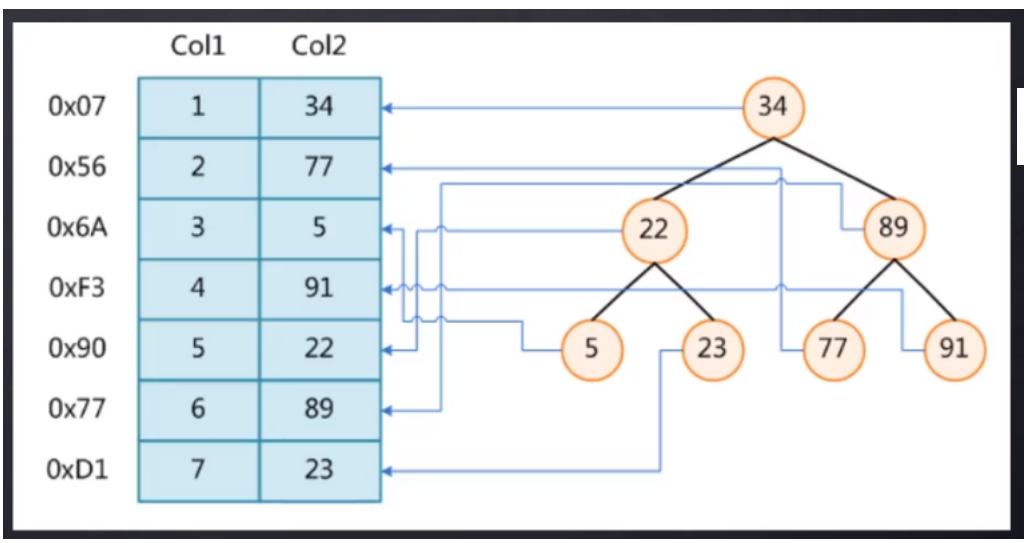
其实索引就是一种用于快速查找数据的数据结构，是帮助MySQL高效获取数据的排好序的数据结构。

回到顶部

索引的好处

举例说明索引的好处以及是怎么加快查询的。

假设我们有一个表t，它有俩字段，Col1和Col2，如下：



不加索引

不加索引的情况下，SQL： `select * from t where t.col2=89` ，需要从表的第一行一行行遍历比对col2的值是否等于89，这样需要比对6次才能查到。这只是只有几行记录的表，那如果是百万级千万级的表呢？是不是就比较的次数就更多了，那还不得慢死。

目录

加索引

如果col2这列加了索引，mysql内部会维护一个数据结构。假设mysql用的数据结构是红黑树（右子树的元素大于根节点，左子树的元素小于根节点）的数据结构建立索引，那就像上图右边那样。这样的话，刚才的那条SQL是不是只需要2次磁盘IO就查到了，是不是快很多了。

这就是索引的好处。索引使用比较巧妙的数据结构，利用数据结构的特性来大大减少查找遍历次数。

回到顶部

索引底层数据结构的探索

既然索引底层原理是利用一些巧妙的数据结构维护我们的数据，使得查询效率很高，那索引底层使用的什么数据结构呢？又是怎样来维护我们的数据呢？下面就带着大家一起探索一下索引的底层数据结构。

索引可选的数据结构：

- 二叉树
- 红黑树
- hash
- B-tree

但mysql索引的底层用的并不是二叉树和红黑树。因为二叉树和红黑树在某些场景下都会暴露出一些弊端或者说缺点。

二叉树

我们看一下二叉树如果作为索引的底层数据结构在什么样的场景下有怎么样的缺点和不足。

假设把刚才的SQL改一下，用col1作为条件来查找，SQL： `select * from t where t.col1 = 6` 。

假如把col1作为索引，col1这列的数据特点是从上到下依次递增，类似于自增主键，那在每插入一行在维护二叉树这样一个数据结构的时候，我们看一下二叉树维护成什么样子了。

打开这个网址(国外的)，可以演示数据结构维护的过程。依次插入1、2、3、4、5...

通过这个网站的演示插入这些数据，我们可以看到这样的一个二叉树是不是一直在单边增长，没有左子树。再仔细看一下和 4 过的链表是不是很像，也就是说**二叉树在某些场景下退化成了链表。**

关注我

排名 – 4832

随笔分类 (156)

dubbo(1)
Java(36)
Netty(1)
nginx(4)
Redis(21)
spring(2)
SpringBoot(2)
Zookeeper(6)
并发/多线程(3)
经验分享(6)
更多

随笔档案 (114)

2020年10月(1)
2020年9月(1)
2020年8月(5)
2020年7月(3)
2020年6月(2)
2020年5月(1)
2020年4月(2)
2020年3月(2)
2020年2月(4)
2020年1月(6)
更多

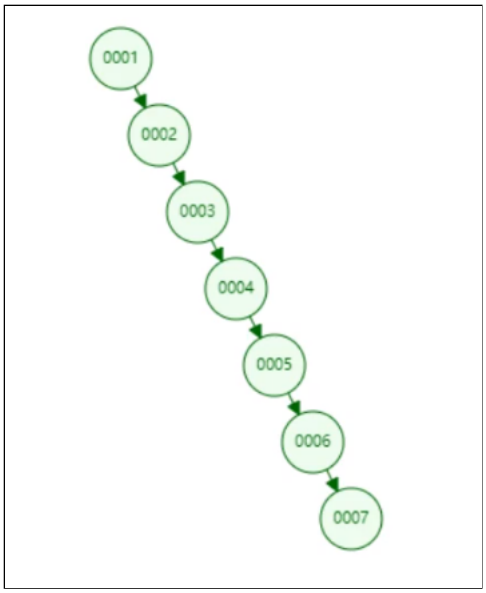
阅读排行榜

1. HTTP 400 Ba
2. com.alibaba.f 对象与JSON转换
3. 访问http接口时 什么原因怎么解决
4. Socket超时时
5. httpclient信任 ion:Unrecognize t connection(172
6. SpringBoot自 过滤器执行顺序(
7. MySQL数据库 44)
8. MySQL数据库
9. SecureCRT远 m refused the c
10. [需求设计]从一 独特魅力(6986)

评论排行榜

1. [需求设计]从一 特魅力(37)
2. Redis分布式锁
3. 最近面试遇到的 的吗？(12)
4. 新 试官对
5. H 00 Ba

目录



链表的查找是不是需要从头部遍历啊，这时候和没加索引从表的第一行遍历是不是没什么太大区别？这就是mysql索引底层没有使用二叉树这种数据结构的原因之一。

当二叉树像上图一样退化成链表后，我们去查col1=6的记录是不是从二叉树的根节点依次遍历，遍历6次才能查到，和不加索引从表里一行行的遍历没太大差别。这是二叉树所谓索引底层数据结构的弊端之一。

红黑树

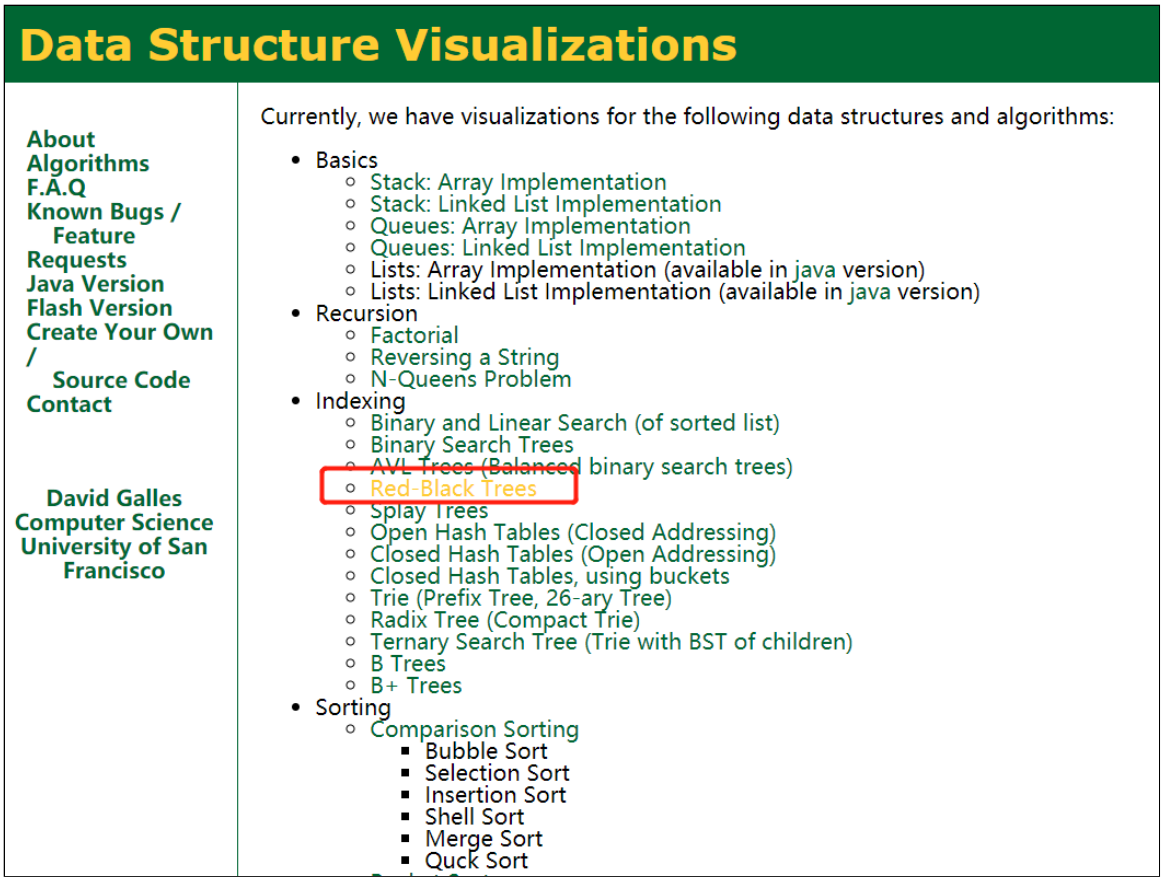
那有没有更好的数据结构用来存储索引，帮助我们更快的查找呢？比方说红黑树或hash表。

我们先看下红黑树。红黑树是什么？

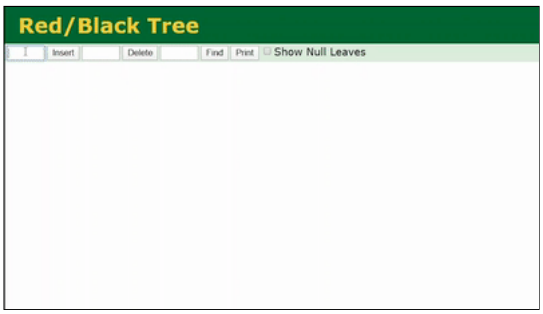
是一种平衡二叉树，JDK1.8的HashMap就用到了红黑树。

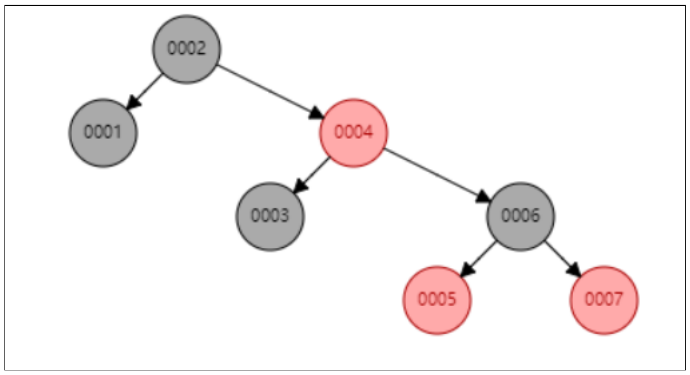
那我们把刚才的一样的数据用红黑树来看一下是什么样的效果，同样打开刚才的网址，我们选择红黑树。

目录



依次插入1、2、3、4、5、6、7看一下效果，可以看到，当有单边增长的趋势时红黑树会进行一个平衡（旋转）。这时，我们查询col1=6的数据时，查了3次，比二叉树又有了改进。





4

关注我

先告诉你mysql索引用的数据结构也不是红黑树，而是B+Tree（B-Tree的变种）。那为什么MySQL也没用红黑树做索引的数据结构呢？说白了红黑树还是有缺陷的。

红黑树做索引底层数据结构的缺陷

我们可以想一下，对于一些大公司特别是互联网公司，表数据动辄数百万数千万，那这样的表我们可以想象一下，现在我们只有7条记录，树的高度就达到了4层，那数百万数千万甚至上亿记录的表创建的索引它的树高得有多高?

假如说我查找的数据在底层的叶子节点上，一般来说都是从根节点开始查找，假如树的高度是50，那我要进行50次查找，50次磁盘IO那得多慢啊这开销已经很大了。这就是红黑树作为索引数据结构的弊端：**树的高度过高导致查询效率变慢。**

那能不能做一点改造呢？我们看，红黑树的树越高遍历次数会越多，会因为树的高度影响查询效率。所以我们要解决的问题就是减少树的高度，尽量控制它的高度在一个阈值范围内。假设说不大于5，即使数据达到1千万2千万最多也就5次磁盘IO就找到了，5次磁盘IO也是可以接受的毕竟表数据这么大嘛。

目录

怎么改造能达到这个效果呢？？？？想一下，既然树的高度不让增加，又想存很多数据。也就是说限制了纵向发展，那就横向发展呗。（身高已经增长不了了，长胖还是可以的）

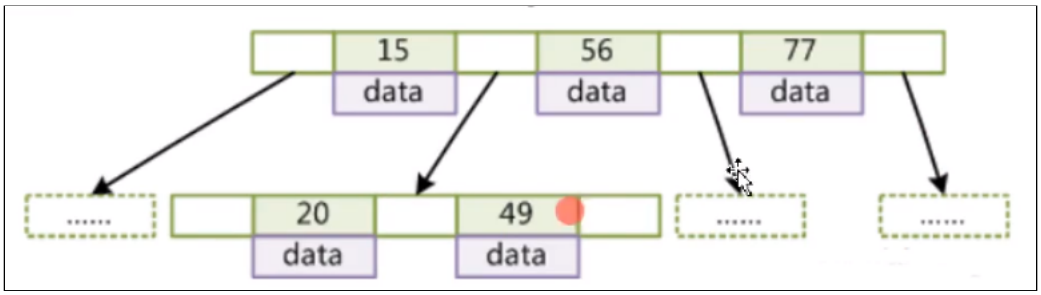
对于上图的红黑树来说每个节点的子节点最多就2个，那基于横向增长的思想就让他变成3叉、4叉、5叉.....让子节点增加，让每一个高度可以存储更多的索引元素，每个节点又分叉，分出来的叉又有很多个节点。那么存储同等数量级别的数据，横向存储的越多，树高就越小了。这样的一个改造结果就是B-Tree。

Hash

待会儿有别的问题会引入hash。

B-Tree

- 叶节点具有相同的深度，叶节点的指针为空
- 所有索引元素不重复
- 节点中的数据索引从左到右递增排序



就这样的一个结构。也就是说在一个节点上可以存储更多的元素，k-v，key就是索引字段，data就是索引字段所在的那一行的数据或是那一行数据坐在的的磁盘文件地址、指针，再去查找元素的时候一次性不是Load一个小元素，而是把一个大的节点的数据一次性全部load到内存，然后再在内存里再去比对，在内存里操作是比较快的。

如果我们要查找49这个元素，实际上是从根节点开始查找的，它一次性将根节点这个大节点一次性load到内存里，然后用要查找的元素在这里去比对，49大于15小于56，在15和56之间有一个节点存储的是下一个节点的磁盘地址指向下一个节点（这个节点的索引都是大于15小于56的），然后再将这个节点一次性load到内存去找这个元素，然后比对就找到了。

注意，一次load节点是一次磁盘IO，是非常慢的，但是我们把它load到内存中之后在你内存里随机的找某一个元素是非常快 盘IO这个时间消耗去比对的话几乎可以忽略不计。

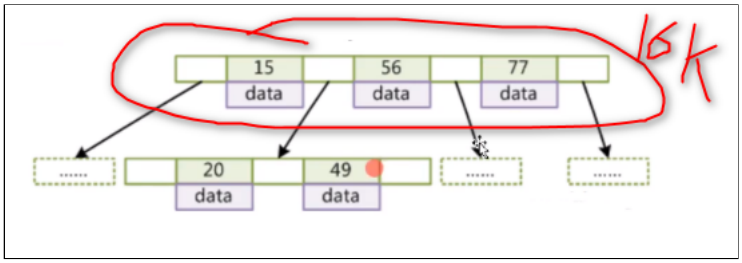
那按这种说法树的高度越小越好，那按这种思路可不可以把一个表的数据都放到一个大的节点上？然后把这个节点一次性load到内存里，我再在内存里一个个去比对不行吗？不是说内存里去比较查找元素是非常的快嘛，跟一次磁盘IO去比对快的多。不可以这样吗？

答案是否定的。

凡事都有个度。你想想，假如我们有几千万数据，在磁盘上面全部放到一个节点上去是不可能的，你的数据表是一行行插入的，存在磁盘上面几百兆甚至几个G，一次性load到内存中合适吗？内存本来就有限，一次性load这么大的数据，而且如果你学过计算机组成原理你也知道，磁盘IO跟内存打交道的单位是4K，一次可能读取4K的数据，可能有时候有一些局部读取的原理可能会取几十K（4的整数倍），取个16K，24K也是可以的 。但是一次交互取这么大是搞不定的，这是计算机组成原理定的，一次磁盘IO取那么多数据，对内存也是非常的浪费，而且

这一次磁盘IO也是非常慢的。所以这个节点的大小设置要合适，不能太大也不能太小，mysql对这个节点大小设置的是16K，用下面这个SQL就是可以查到 `show clobal status like 'Innodb_page_size'` 。

目录



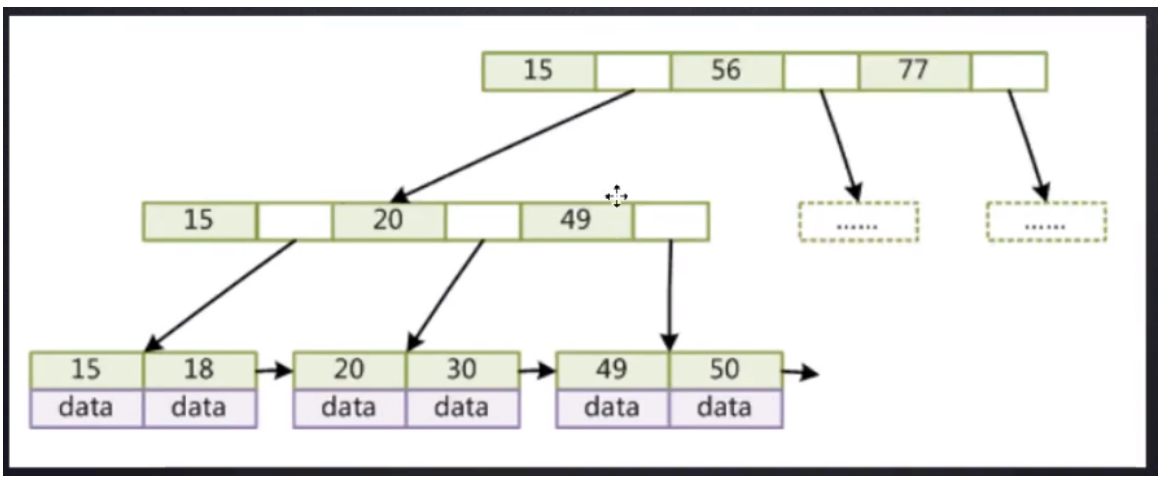
为啥设置16K？为什么不是更大的如16M呢，16K已经足够用了。等会儿会具体讲。

MySQL索引选择的不是原生的B-Tree，而是对他进行了改造，得到的是一种叫做B+Tree的数据结构

B+Tree（B-Tree变种）

- 非叶子节点不存储data，只存储索引（冗余），可以放更多的索引
- 叶子节点包含所有索引字段
- 叶子节点用指针连接，提高区间访问的性能

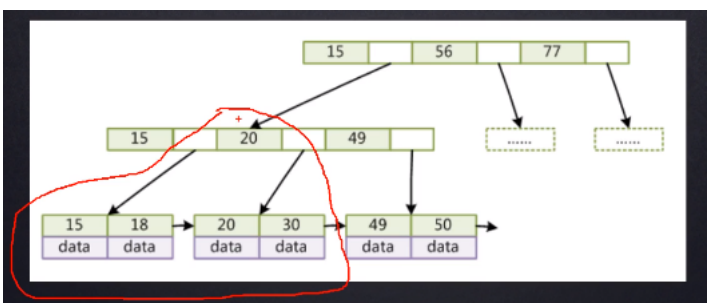
（实际上叶子节点间的指针是双向的，图有问题）



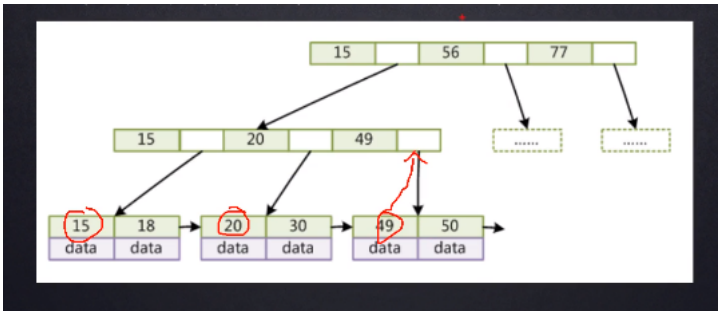
和B-Tree有啥区别？非叶子节点没有数据，数据都挪到叶子节点，叶子节点之间还有指针，非叶子节点之间跟原来一样没有指针。

为啥data元素挪到叶子节点？非叶子节点只存储索引元素，叶子节点存储了一份完整表的所有行的索引字段，data元素是每个索引元素对应要查找的行记录的位置或行数据，这样非叶子节点的每个节点就可以存储更多的索引元素（等会会有一个大致的估算）。实际上非叶子节点存储的是一些冗余索引，看一下上图，15/20/49，选择的是整张表的哪些数据作为索引？选择的是处于中间位置的，因为它 4 3+Tree一些比大小去查找，B+Tree本质可以叫做多叉平衡树，单看B+Tree的某一小块他还是一个二叉树。

关注我



还有一个特点，某一个节点的元素处于一个递增的顺序，会提取叶子节点的一些处于中间位置的数据作为冗余索引，查找的时候从根节点开始查找，先把根节点加载到内存里去，然后在内存里去比对。



目录

比如要查找索引为30的数据，先在根节点跟15去比较，大于15，然后小于56，然后从他俩中间的指针查找下一个节点把它load到内存，再在内存里去比对，大于15，大于20，然后小于49，就根据20和49之间的指针找到下一个节点，然后load到内存，去比对，不等于20下一个30，相等，OK了。

为什么把中间的元素提取出来做冗余元素，为的是查找效率更高。

回到刚刚的问题，为啥要搞这些冗余索引，而且把这些冗余索引的data元素搞到叶子节点？也就是说B+Tree相对于B-Tree来说我的非叶子节点是不存储data元素的，叶子节点才存储data元素？

你想一下，一个节点不能太大也不能太小，就是16K，把data元素挪走以后，是不是这个节点就能存更多的冗余索引了，意味着分叉就更多了，意味着叶子节点就能存储更多的数据了。

mysql为什么把节点大小设置为16K，而不是更大？

假设索引字段类型是Bigint，8bit，每两个元素之间存的是下一个节点的地址，mysql分配的是6bit，也就是说一个索引后面配对一个节点地址，成对出现，可以算一下16K的节点可以存多少对也就是多少个索引，8b+6b=14b，16K /14b=1170个索引，叶子节点有索引有data元素，假设占1K，那一个节点就放16K/1K=16个元素，假设树高是3，所有节点都放满，能放多少数据？可以算一下，1170*1170*16=21902400，2千多万，mysql设置16K的大小，数据就可以存2千多万就已经足够了吧，既能保证一次磁盘IO不要Load太多的数据 又能保证一次load的性能，即便表的数据在几千万的数量也能保证树的高度在一个可控的范围。

可以看一下几千万的数据表是不是加了索引几十毫秒几百毫秒就出结果了，所以就解释了几千万的表精确的使用索引后他的性能依旧比较高。

树的高度只有3的情况下就能存储2千多万的数据，即便某一个索引在叶子节点，那也就2、3次磁盘IO就能查找到，当然很快了。而且mysql底层的索引他的根节点，是常驻内存的，直接就放到内存的，查找叶子节点，一个2千万的数据放到B+Tree上面，要查找叶子节点，就只需要2次磁盘IO就搞定了，在内存里比对的时间基本可以忽略。

回到顶部

MySQL是如何存储索引和数据的

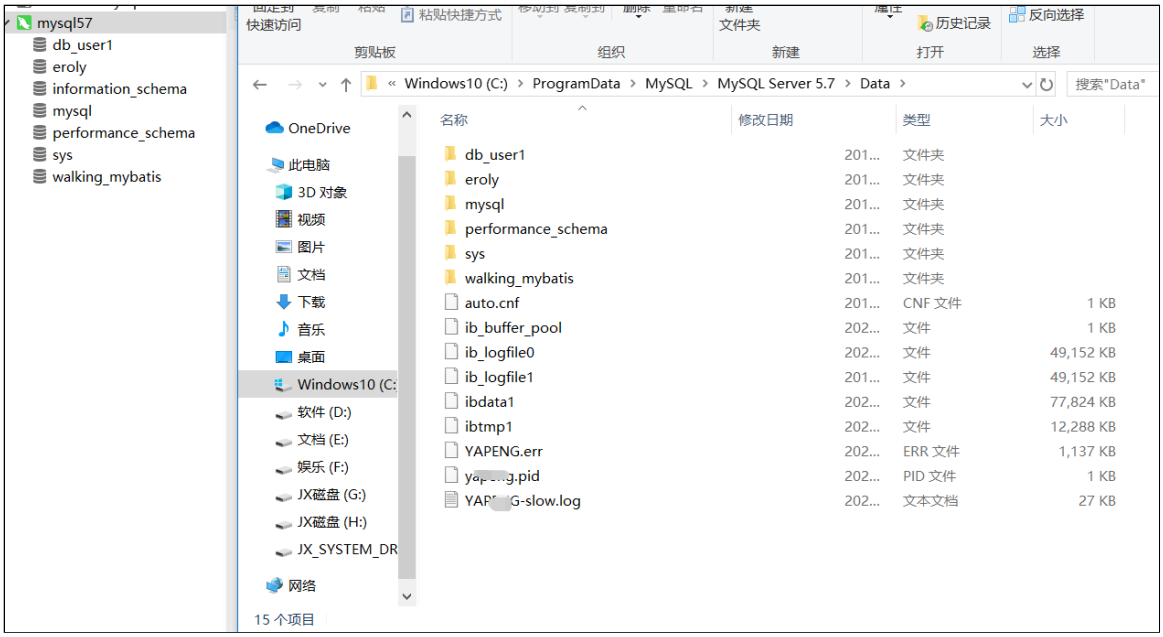
刚才讲的原理性的比较多，现在结合具体的mysql的表不同的索引来看一下它底层到底是如何运用B+Tree来维护索引的。

索引和数据存放位置是哪?

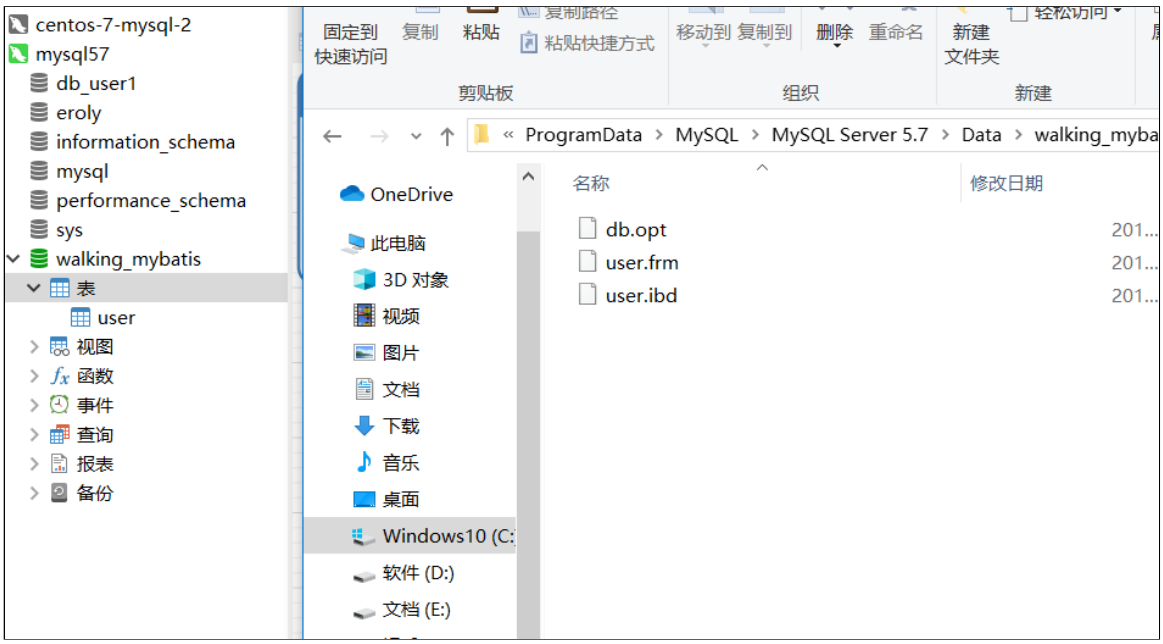
首先问下mysql的表、数据、索引是放到那里的?

磁盘=》默认是安装目录的data文件里（不同版本可能有所不同），每个数据库对应data文件夹里的一个文件夹

目录



我们打开一个walking_mybatis数据库看一下有一个user表，再打开对应的文件夹看一下，里面的文件名和表名有关系，然后有不同的后缀，这里面的不同的放法和mysql的存储引擎有关，和你选择的哪种存储引擎有关。



存储引擎是修饰什么的?

大家都知道，mysql常见的存储引擎有InnoDB存储引擎，MYISAM存储引擎，那存储引擎是形容mysql数据库的还是某一张表

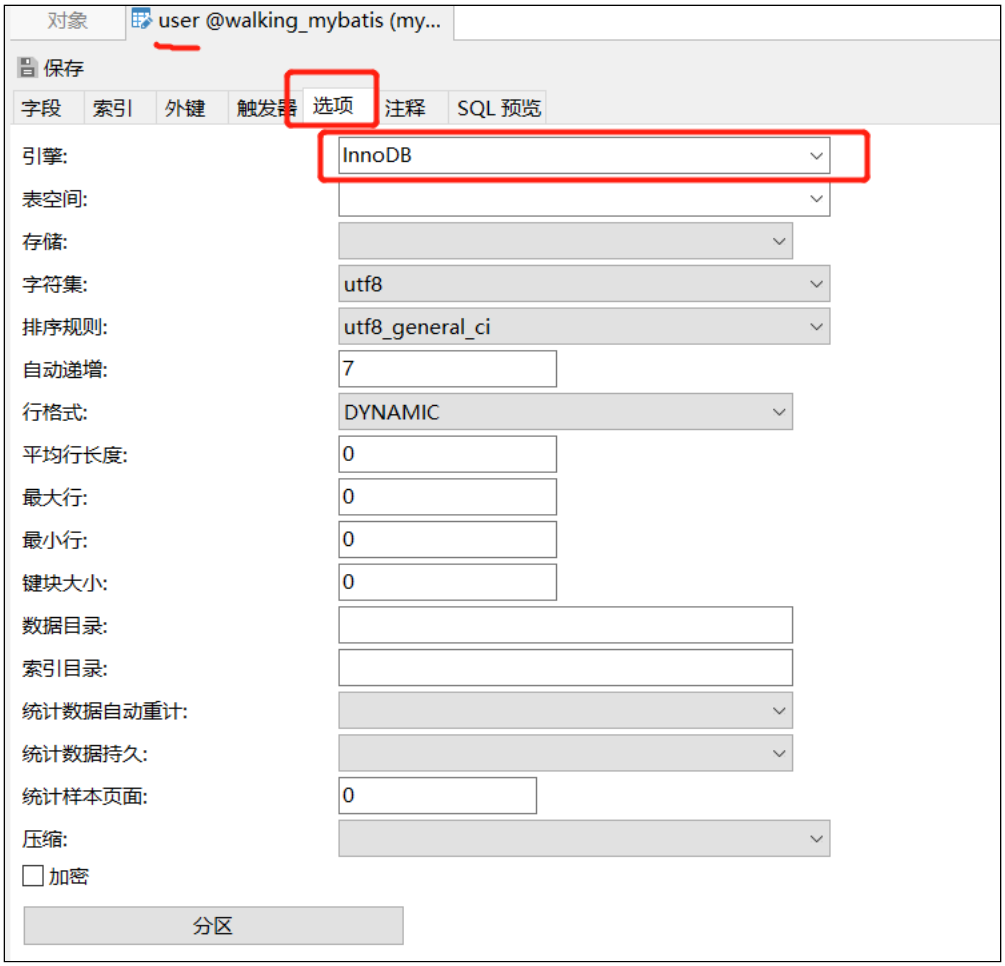
4

关注我

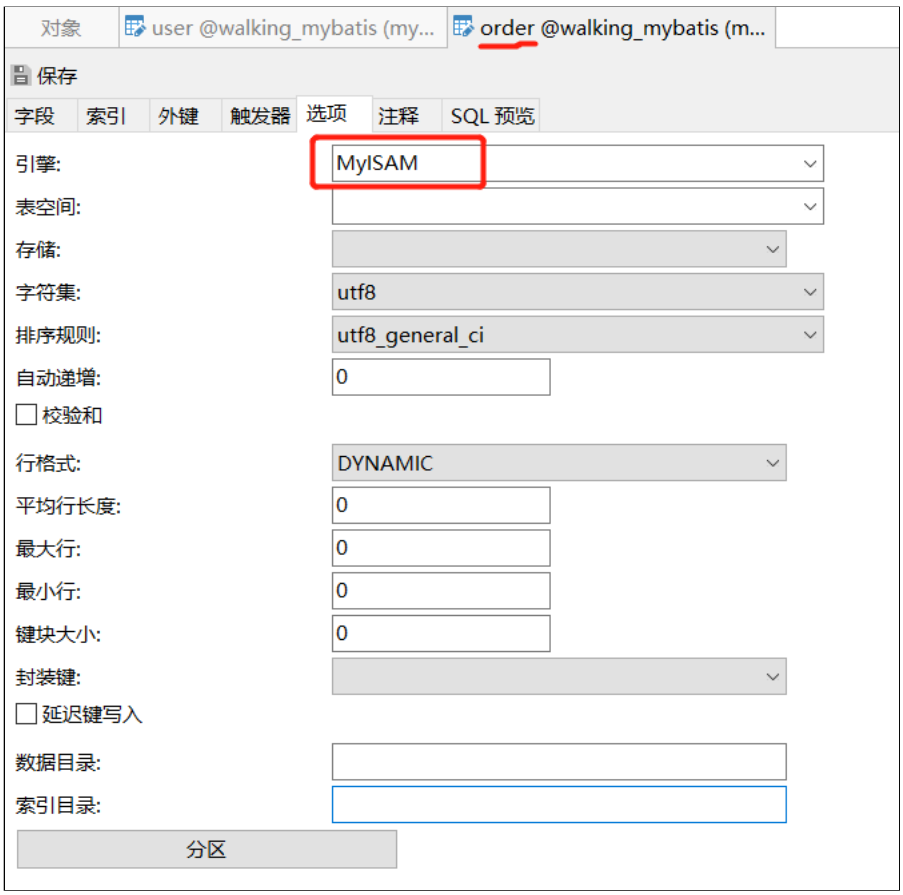
是表，尽管数据库级别也有存储引擎选项，但最终还是以表的存储引擎为主的。

如果你用Navicat工具去建表，也许你最多就用了“字段”这一栏去增加字段，你可以点一下“选项”看一下，可以选择存储引擎。

目录



我这边又新建一个order表，然后选择为MYISAM存储引擎



4

关注我

在表上右键选择『对象信息』->『DDL』查看

```
CREATE TABLE `order` (
  `order_id` int(11) NOT NULL,
  `order_price` decimal(10,2) DEFAULT NULL,
  `order_user` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`order_id`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

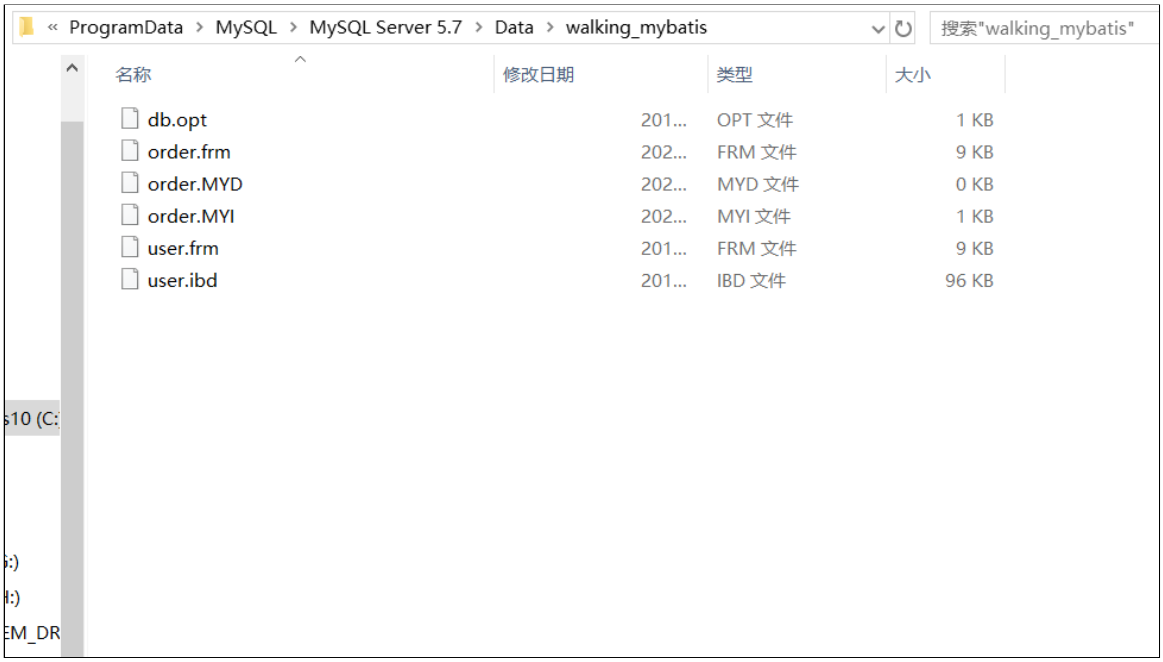
目录

看一下user表的

```
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  `sex` varchar(1) DEFAULT NULL,
  `phone` varchar(11) DEFAULT NULL,
  `email` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=7 DEFAULT CHARSET=utf8;
```

索引和数据文件

再来看一下这个数据库文件夹下这俩表的数据文件。



我们会发现，user表（InnoDB存储引擎）对应两个文件，order表（MYISAM存储引擎）对应3个文件。其中.frm文件是存储的是表结构，两个存储引擎都一样，而InnoDB的.ibd文件是索引+数据，MYISAM的.MYI（I：index）和.MYD（D：data）文件分别是索引字段的索引结构和数据文件，也就是说MYISAM存储引擎的索引和数据是分开的，而InnoDB存储引擎的数据和索引是在一个文件里的。

4

关注我

InnoDB和MYISAM的一些不同

MYISAM存储引擎

MYISAM索引实现（非聚集）

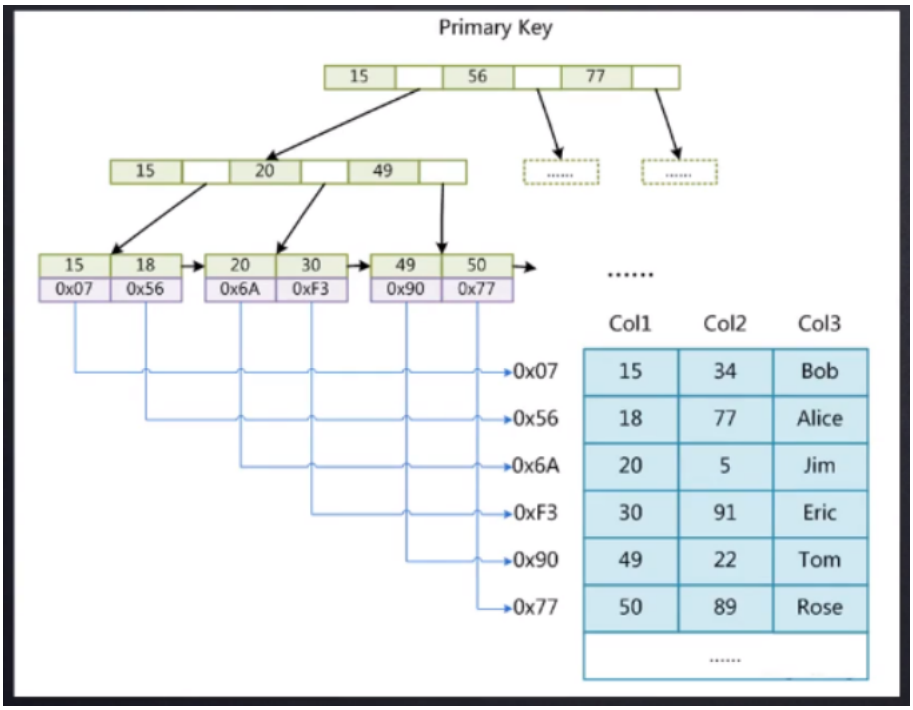
- 索引文件和数据文件是分离的（非聚集）

数据、行记录是存储在MYD文件，假如col1是索引字段那么这一列是存储在MYI文件里以B+Tree的结构来组织的，然后他的叶子节点的data部分存储的是索引所在行记录的磁盘文件地址，根据磁盘文件地址指针就可以从MYD文件里快速的找到我们的这一行记录。

查找过程

所以MYISAM这个存储引擎他的查找的一个大致过程就是，先看条件字段有没有用到索引，是索引字段就先去到索引文件去查找这个索引所在的那一行的磁盘文件地址，就借助B+Tree的特点从根节点顺藤摸瓜找到磁盘文件地址指针，然后从MYD文件一次性定位到所找的数据，也就是说MYISAM会垮两个文件。

目录



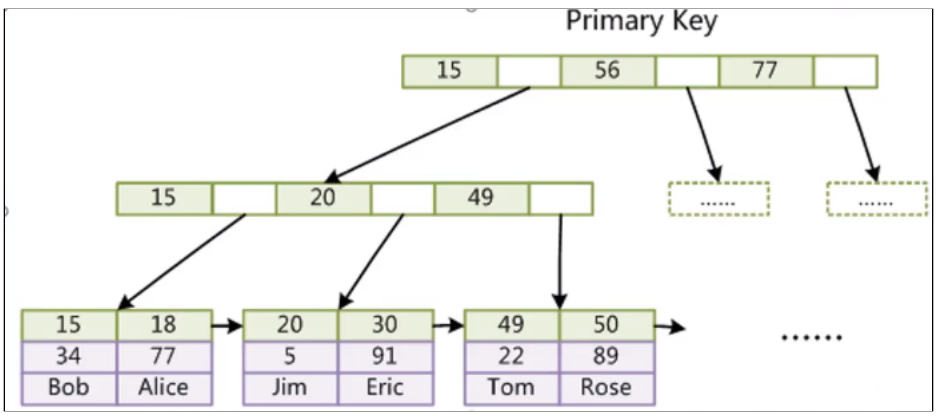
InnoDB存储引擎

InnoDB索引实现（聚集）

- 表数据文件本身就是按B+Tree组织的一个索引结构文件
- 聚集索引–叶子节点包含了完整的数据记录
- 为什么InnoDB表必须有主键，并且推荐使用整型的自增主键?
- 为什么非主键索引结构叶子节点存储的是主键值？（一致性和节省存储空间）

用的最多的InnoDB存储引擎是什么样子的呢？我们可以看到，它只有两个文件。.frm文件和MYISAM一样，都是表结构文件，.ibd文件就是MYISAM的MYI和MYD文件的合并，索引文件和数据文件都存储到一个文件。

InnoDB存储引擎索引存储结构大概是下图这样的，它也是一个B+Tree，但是它的叶子节点和MYISAM有点区别，它存储的是索引所在行的所有字段。



4

关注我

这个好处是是什么？不用回表（如果把垮文件查找理解为回表）了，性能应该比MYISAM高，你看MYISAM查找到索引所在行记录的磁盘地址后还要回MYD文件读取一次。

回到顶部

聚集索引/非聚集索引

目录

聚集索引/聚簇索引，叶子节点包含了完整的数据记录，InnoDB的主键索引就是一个聚集索引，他的索引和数据是绑定在一起的(叶子节点)。MYISAM的是非聚集索引，索引和数据是分开存储的。InnoDB的主键索引我们叫做聚集索引。

回到顶部

为什么InnoDB表必须有主键，并且推荐使用整型的自增主键?

我们看一下这个问题**为什么InnoDB表必须有主键，并且推荐使用整型的自增主键?**

为甚InnoDB表建议要有自增的主键，尽量建主键，建整形自增的？其实很简单，设计如此，mysql设计的就是innoDB把你的数据和主键索引用B+Tree来组织的，没有主键他的数据就没有一个结构来存储。

建innoDB表的时候没有建主键，表也能建成功，为什么?

不建主键不代表没有主键，没有建主键innodb会帮你选一个字段，一个可以标识唯一的字段，选为默认字段，如果这个字段唯一的话，不重复,可以建唯一索引的话，就会作为类似于唯一索引，用这个字段来作为唯一索引来维护整个表的数据。如果没有，mysql会生成一个唯一的列，类似于rowid，只不过你看不到，他会用生成的这个唯一列，维护B+Tree的结构，查数据的时候还是用B+Tree的结构去查找。

为什么推荐整形呢?

我们想象一下查找过程，是把节点load到内存然后在内存里去比较大小，也就是在查找的过程中要不断的去进行数据的比对。假设UUID，既不自增也不是整形。问一下，是整形的1<2比较的效率高还是字符串的“abc”和“abe”比较的效率高呢？显然是前者，因为字符串的比较是转换成ASCII码一位一位的比，如果最后一位不一样，比到最后才比较出大小，就比整形比较慢多了，存储空间来说，整形更小。索引越节约资源越好。

为什么是自增的呢?

我们可以看一下B-Tree的叶子节点之间是没有指针的，B+Tree优化后增加了叶子节点之间的指针，如果我们遍历数据，从当前节点遍历完之后，就可以根据节点间的指针快速找到下一个节点去遍历。讲到这，穿插一下B+Tree为什么要比B-Tree多一个节点间指针呢？那就讲一下索引的另一种数据结构就是hash。

回到顶部

HASH索引

99.99的情况都是用B+Tree，也有些情况用hash。假设我们的索引选的是hash的数据结构，每插入一个元素会把我们的索引字段做一次hash计算，把运算的到的结果值和这一行的所在磁盘地址做一个映射。

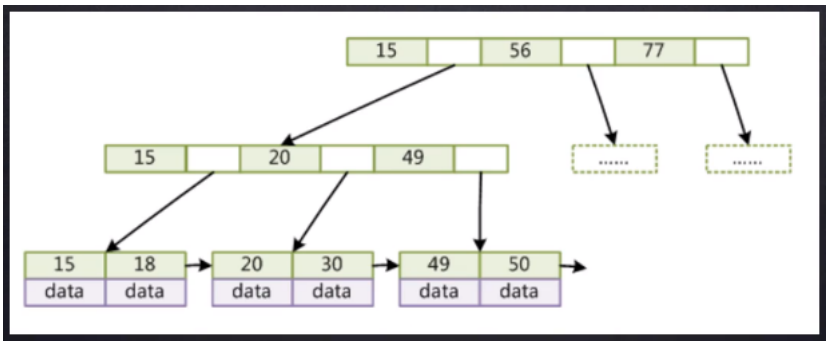
对索引元素的值做一次hash运算就可以在hash映射表里快速找到这一行的磁盘文件地址，经过一次hash就可以快速定位到索引所在行的磁盘文件地址，hash这么快，表有一亿个数据按这种算法，那也就可能经历一次hash运算就可以快速找到某页任意一行数据元素的所在的磁盘文件地址，那比B+Tree快的多啊！就是快的多，为啥99.99的都是B+Tree不是hash呢?

hash的等值查询比B+Tree快，上亿依然很快，为啥很快却不使用？最主要的原因是什么？因为如果使用范围查找，hash就没，范围查找也是很常用的吧，所以基本就不怎么用hash这种数据结构。那B+Tree就很好的支撑范围查找吗?

是，B+Tree可以很好的支撑。

看一下这个B+Tree的结构

目录



刚才我们说了B+Tree的任一叶子节点内部是从左到右都是递增的，且节点之间有一个指针（双向的，图不标准），

假设我们查大于20的记录，mysql内部是怎么查找的？先从根节点，定位到大于20的元素，然后依次从左到右找到30，然后这个节点遍历完了，就可以根据指针找到下一个节点的位置，因为B+Tree的特点，后面的元素全都大于20，就这样顺藤摸瓜把后面的元素全弄出来。

那B-Tree没有这个指针的话查找大于20 的元素那得多麻烦，先找出第一个节点中大于20的全部元素，因为还有别的节点，所以又要从根节点去遍历找下一个叶子节点，是不是非常慢。没有这个指针每次都要从根节点开始查找然后合并，那是非常慢的。

回到顶部

为什么非主键索引结构叶子节点存储的是主键值?

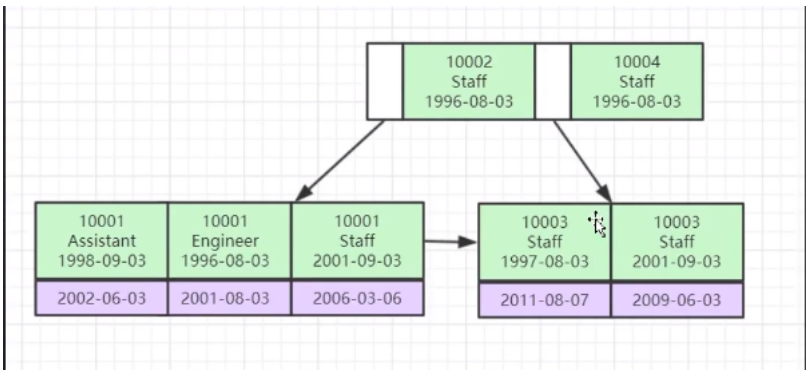
为了一致性和节省存储空间。已经维护了一套主键索引+数据的B+Tree结构，如果再有其他的非主键索引的话，索引的叶子节点存储的是主键，这是为了节省空间，因为继续存数据的话，那就会导致一份数据存了多份，空间占用就会翻倍。另一方面也是一致性的考虑，都通过主键索引来找到最终的数据，避免维护多份数据导致不一致的情况。

回到顶部

联合索引

尽量建联合索引，少建单值索引 。刚讲的都是单值索引

联合索引的底层数据结构是什么样的?



4

关注我

多个列逐个字段去比较， （a,b,c）

多个索引有多个B+树结构，非主键索引叶子节点存储的不是数据，而是主键（一致性和节省空间）

手机端可扫码关注公众号，方便阅读