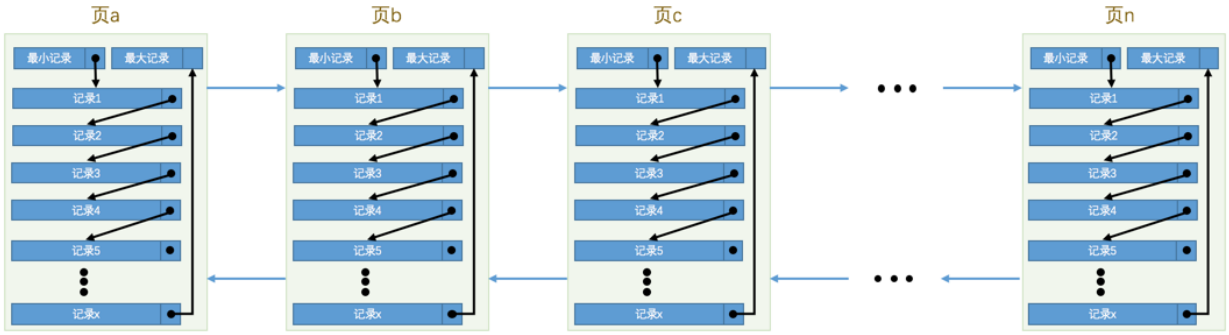
"

B+树索引

标签： MySQL是怎样运行的

前边我们详细唠叨了InnoDB数据页的7个组成部分，知道了各个数据页可以组成一个双向链表，而每个数据页中的记录会按照主键值从小到大的顺序组成一个单向链表，每个数据页都会为存储在它里边儿的记录生成一个页目录，在通过主键查找某条记录的时候可以在页目录中使用二分法快速定位到对应的槽，然后再遍历该槽对应分组中的记录即可快速找到指定的记录（如果你对这段话有一丁点儿疑惑，那么接下来的部分不适合你，返回去看一下数据页结构吧）。页和记录的关系示意图如下：



其中页a、页b、页c ... 页n这些页可以不在物理结构上相连，只要通过双向链表相关联即可。

没有索引的查找

本集的主题是索引，在正式介绍索引之前，我们需要了解一下没有索引的时候是怎么查找记录的。为了方便大家理解，我们下边先只唠叨搜索条件为对某个列精确匹配的情况，所谓精确匹配，就是搜索条件中用等于=连接起的表达式，比如这样：

```
SELECT [列名列表] FROM 表名 WHERE 列名 = xxx;
```

在一个页中的查找

假设目前表中的记录比较少，所有的记录都可以被存放到一个页中，在查找记录的时候可以根据搜索条件的不同分为两种情况：

- 以主键为搜索条件

  这个查找过程我们已经很熟悉了，可以在页目录中使用二分法快速定位到对应的槽，然后再遍历该槽对应分组中的记录即可快速找到指定的记录。

- 以其他列作为搜索条件

  对非主键列的查找的过程可就不这么幸运了，因为在数据页中并没有对非主键列建立所谓的页目录，所以我们无法通过二分法快速定位相应的槽。这种情况下只能从最小记录开始依次遍历单链表中的每条记录，然后对比每条记录是不是符合搜索条件。很显然，这种查找的效率是非常低的。

在很多页中查找

大部分情况下我们表中存放的记录都是非常多的，需要好多的数据页来存储这些记录。在很多页中查找记录的话可以分为两个步骤：

1. 定位到记录所在的页。
2. 从所在的页内中查找相应的记录。

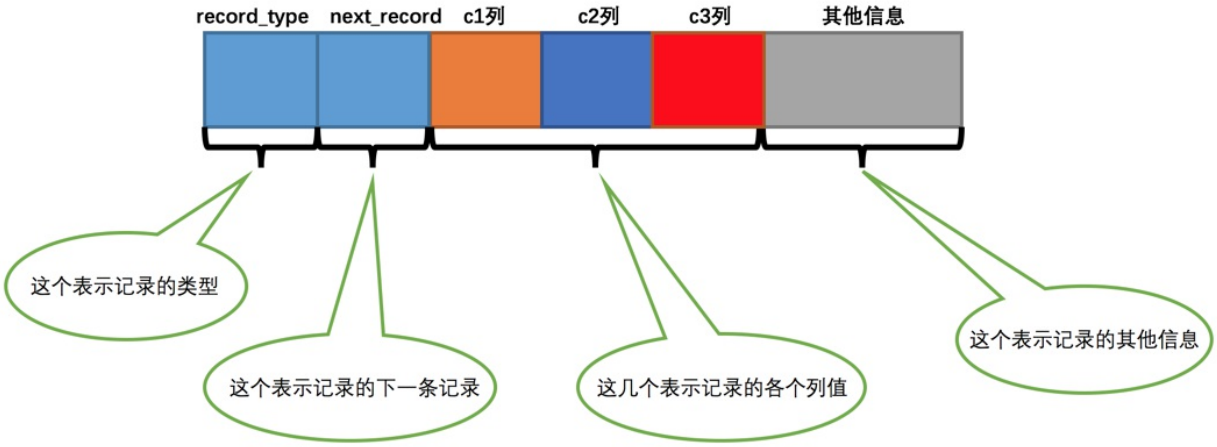
在没有索引的情况下，不论是根据主键列或者其他列的值进行查找，由于我们并不能快速的定位到记录所在的页，所以只能从第一个页沿着双向链表一直往下找，在每一个页中根据我们刚刚唠叨过的查找方式去查找指定的记录。因为要遍历所有的数据页，所以这种方式显然是超级耗时的，如果一个表有一亿条记录，使用这种方式去查找记录那要等到猴年马月才能等到查找结果。所以祖国和人民都在期盼一种能高效完成搜索的方法，索引同志就要亮相登台了。

索引

为了故事的顺利发展，我们先建一个表：

```
mysql> CREATE TABLE index_demo(
    ->     c1 INT,
    ->     c2 INT,
    ->     c3 CHAR(1),
    ->     PRIMARY KEY(c1)
    -> ) ROW_FORMAT = Compact;
Query OK, 0 rows affected (0.03 sec)
```

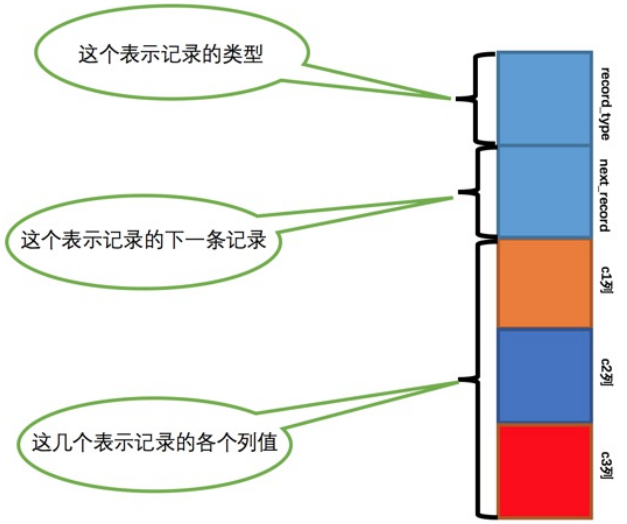
这个新建的index_demo表中有2个INT类型的列，1个CHAR(1)类型的列，而且我们规定了c1列为主键，这个表使用Compact行格式来实际存储记录的。为了我们理解上的方便，我们简化了一下index_demo表的行格式示意图：



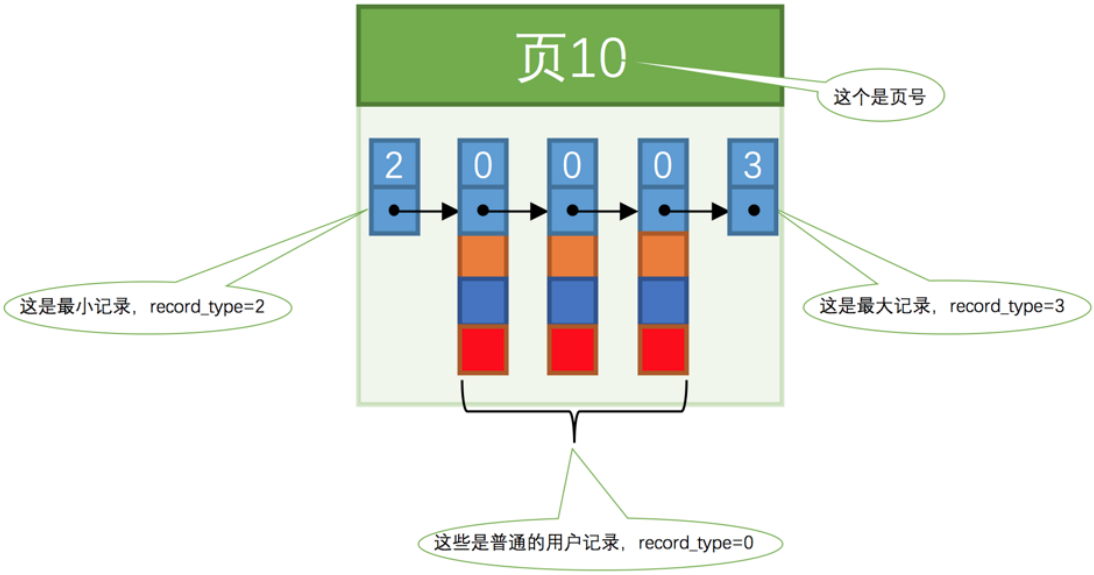
我们只在示意图里展示记录的这几个部分：

- `record_type`：记录头信息的一项属性，表示记录的类型，`0`表示普通记录、`2`表示最小记录、`3`表示最大记录、`1`我们还没用过，等会再说～
- `next_record`：记录头信息的一项属性，表示下一条地址相对于本条记录的地址偏移量，为了方便大家理解，我们都会用箭头来表明下一条记录是谁。
- 各个列的值：这里只记录在`index_demo`表中的三个列，分别是`c1`、`c2`和`c3`。
- 其他信息：除了上述3种信息以外的所有信息，包括其他隐藏列的值以及记录的额外信息。

为了节省篇幅，我们之后的示意图中会把记录的其他信息这个部分省略掉，因为它占地方并且不会有什么观赏效果。另外，为了方便理解，我们觉得把记录竖着放看起来感觉更好，所以将记录格式示意图的其他信息去掉并把它竖起来的效果就是这样：



把一些记录放到页里边的示意图就是：



一个简单的索引方案

回到正题，我们在根据某个搜索条件查找一些记录时为什么要遍历所有的数据页呢？因为各个页中的记录并没有规律，我们并不知道我们的搜索条件匹配哪些页中的记录，所以 ***不得不*** 依次遍历所有的数据页。所以如果我们想快速的定位到需要查找的记录在哪些数据页中该咋办？还记得我们为根据主键值快速定位一条记录在页中的位置而设立的

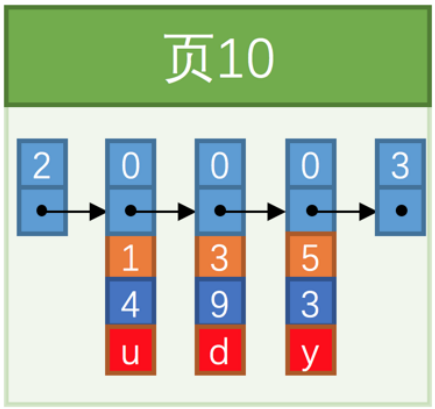
页目录么？我们也可以想办法为快速定位记录所在的数据页而建立一个别的目录，建这个目录必须完成下边这些事儿：

- 下一个数据页中用户记录的主键值必须大于上一个页中用户记录的主键值。

  为了故事的顺利发展，我们这里需要做一个假设：假设我们的每个数据页最多能存放3条记录（实际上一个数据页非常大，可以存放下好多记录）。有了这个假设之后我们向index_demo表插入3条记录：

```
mysql> INSERT INTO index_demo VALUES(1, 4, 'u'), (3, 9, 'd'), (5, 3, 'y');
Query OK, 3 rows affected (0.01 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

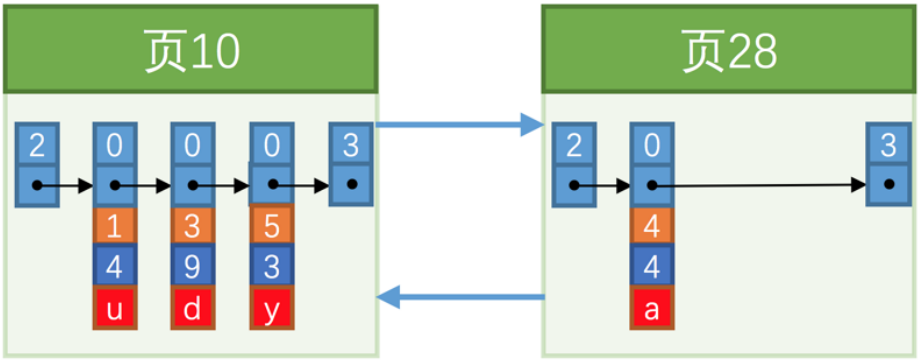
  那么这些记录已经按照主键值的大小串联成一个单向链表了，如图所示：



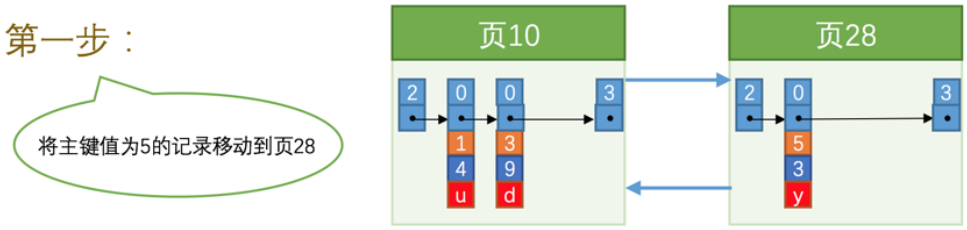
  从图中可以看出来，index_demo表中的3条记录都被插入到了编号为10的数据页中了。此时我们再来插入一条记录：

```
mysql> INSERT INTO index_demo VALUES(4, 4, 'a');
Query OK, 1 row affected (0.00 sec)
```

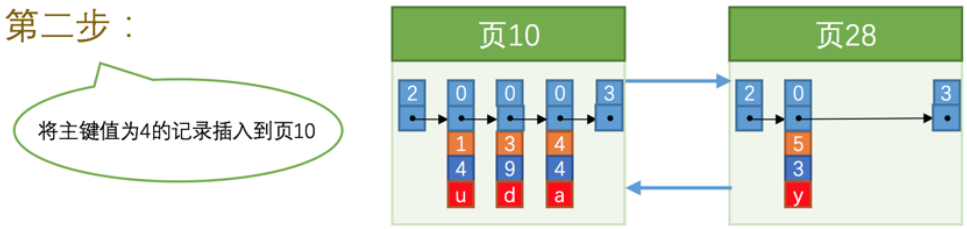
  因为页10最多只能放3条记录，所以我们不得不再分配一个新页：



  咦？怎么分配的页号是28呀，不应该是11么？再次强调一遍，新分配的数据页编号可能并不是连续的，也就是说我们使用的这些页在存储空间里可能并不挨着。它们只是通过维护着上一个页和下一个页的编号而建立了链表关系。另外，页10中用户记录最大的主键值是5，而页28中有一条记录的主键值是4，因为5 > 4，所以这就不符合下一个数据页中用户记录的主键值必须大于上一个页中用户记录的主键值的要求，所以在插入主键值为4的记录的时候需要伴随着一次记录移动，也就是把主键值为5的记录移动到页28中，然后再把主键值为4的记录插入到页10中，这个过程的示意图如下：



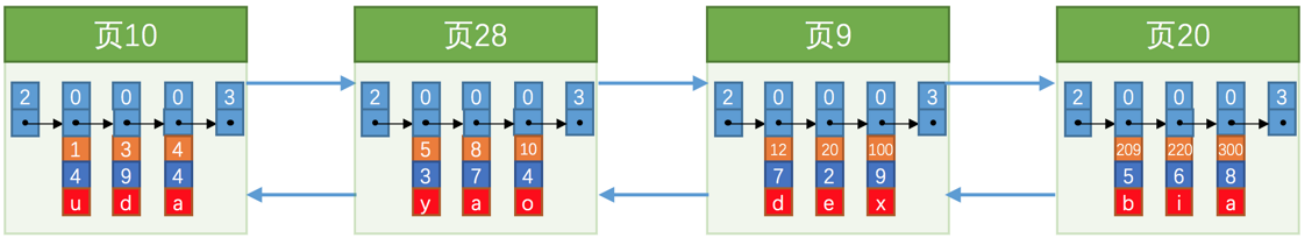
华丽丽的分割线



这个过程表明了在对页中的记录进行增删改操作的过程中，我们必须通过一些诸如记录移动的操作来始终保证这个状态一直成立：下一个数据页中用户记录的主键值必须大于上一个页中用户记录的主键值。这个过程我们也可以称为页分裂。

- 给所有的页建立一个目录项。

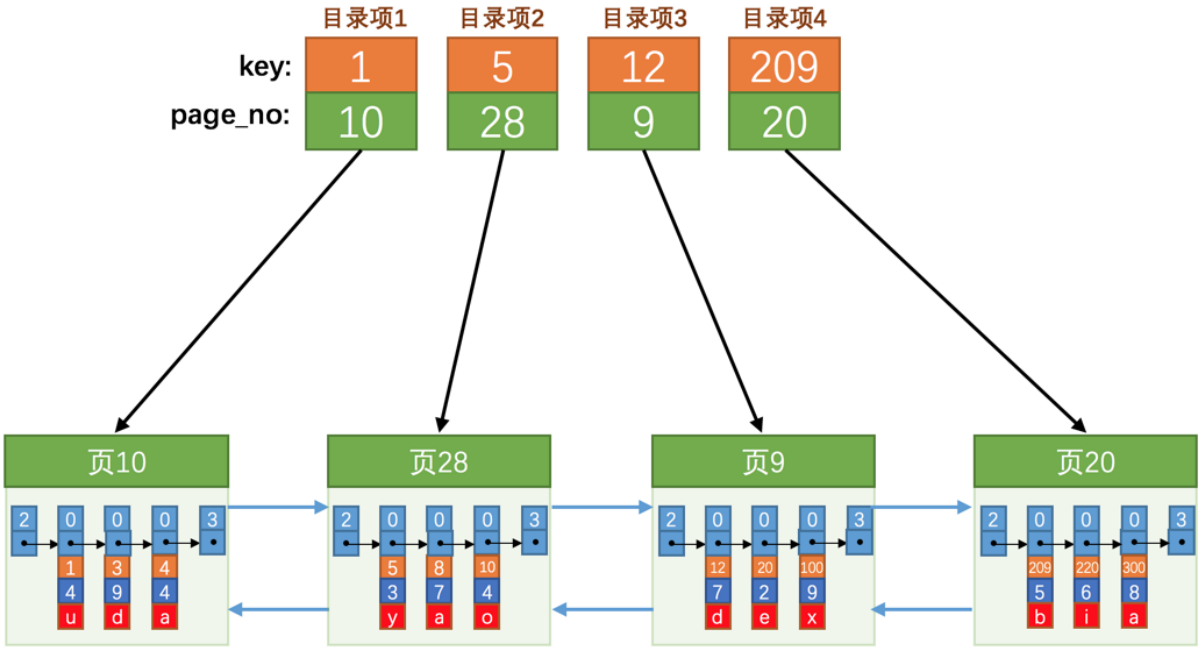
  由于数据页的编号可能并不是连续的，所以在向index_demo表中插入许多条记录后，可能是这样的效果：



因为这些16KB的页在物理存储上可能并不挨着，所以如果想从这么多页中根据主键值快速定位某些记录所在的页，我们需要给它们做个目录，每个页对应一个目录项，每个目录项包括下边两个部分：

- 页的用户记录中最小的主键值，我们用key来表示。
- 页号，我们用page_no表示。

所以我们为上边几个页做好的目录就像这样子：



以页28为例，它对应目录项2，这个目录项中包含着该页的页号28以及该页中用户记录的最小主键值5。我们只需要把几个目录项在物理存储器上连续存储，比如把他们放到一个数组里，就可以实现根据主键值快速查找某条记录的功能了。比方说我们想找主键值为20的记录，具体查找过程分两步：

1. 先从目录项中根据二分法快速确定出主键值为20的记录在目录项3中（因为 12 < 20 < 209），它对应的页是页9。
2. 再根据前边说的在页中查找记录的方式去页9中定位具体的记录。

至此，针对数据页做的简易目录就搞定了。不过忘了说了，这个目录有一个别名，称为索引。

InnoDB中的索引方案

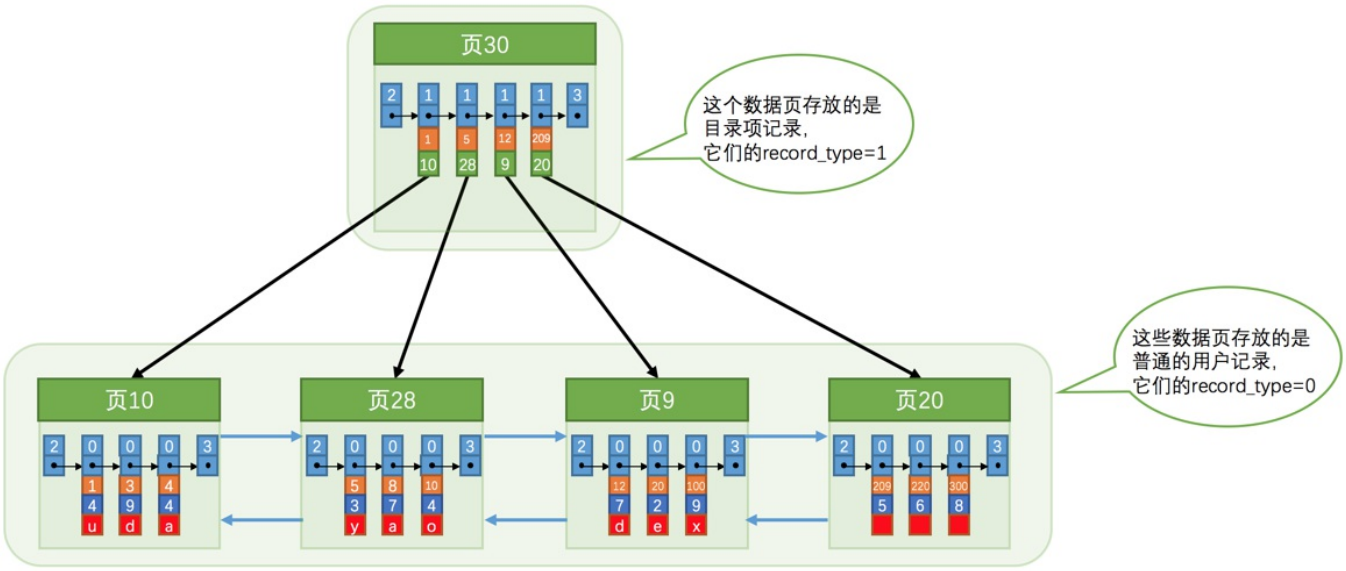
上边之所以称为一个简易的索引方案，是因为我们为了在根据主键值进行查找时使用二分法快速定位具体的目录项而假设所有目录项都可以在物理存储器上连续存储，但是这样做有几个问题：

- InnoDB是使用页来作为管理存储空间的基本单位，也就是最多能保证16KB的连续存储空间，而随着表中记录数量的增多，需要非常大的连续的存储空间才能把所有的目录项都放下，这对记录数量非常多的表是不现实的。
- 我们时常会对记录进行增删，假设我们把页28中的记录都删除了，页28也就没有存在的必要了，那意味着目录项2也就没有存在的必要了，这就需要把目录项2后的目录项都向前移动一下，这种牵一发而动全身的设计不是什么好主意～

所以，设计InnoDB的大叔们需要一种可以灵活管理所有目录项的方式。他们灵光乍现，忽然发现这些目录项其实长得跟我们的用户记录差不多，只不过目录项中的两个列是主键和页号而已，所以他们复用了之前存储用户记录的数据页来存储目录项，为了和用户记录做一下区分，我们把这些用来表示目录项的记录称为目录项记录。那InnoDB怎么区分一条记录是普通的用户记录还是目录项记录呢？别忘了记录头信息里的record_type属性，它的各个取值代表的意思如下：

- 0：普通的用户记录
- 1：目录项记录
- 2：最小记录
- 3：最大记录

哈哈，原来这个值为1的record_type是这个意思呀，我们把前边使用到的目录项放到数据页中的样子就是这样：



从图中可以看出来，我们新分配了一个编号为30的页来专门存储目录项记录。这里再次强调一遍目录项记录和普通的用户记录的不同点：

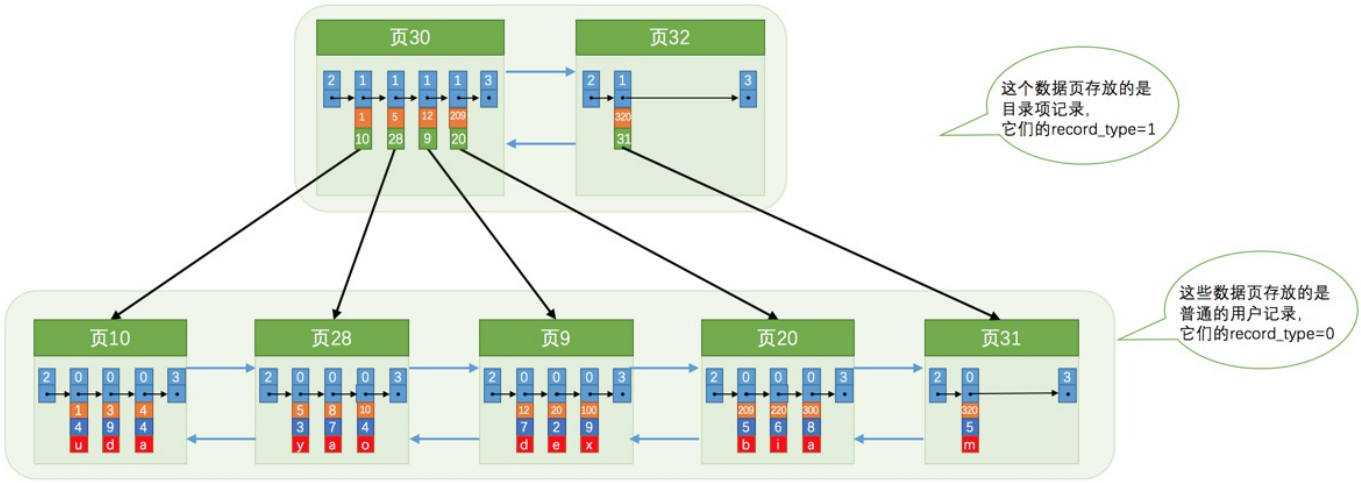
- 目录项记录的record_type值是1，而普通用户记录的record_type值是0。
- 目录项记录只有主键值和页的编号两个列，而普通的用户记录的列是用户自己定义的，可能包含很多列，另外还有InnoDB自己添加的隐藏列。
- 还记得我们之前在唠叨记录头信息的时候说过一个叫min_rec_mask的属性么，只有在存储目录项记录的页中的主键值最小的目录项记录的min_rec_mask值为1，其他别的记录的min_rec_mask值都是0。

除了上述几点外，这两者就没啥差别了，它们用的是一样的数据页（页面类型都是0x45BF，这个属性在File Header中，忘了的话可以翻到前边的文章看），页的组成结构也是一样一样的（就是我们前边介绍过的7个部分），都会为主键值生成Page Directory（页目录），从而在按照主键值进行查找时可以使用二分法来加快查询速度。现在以查找主键为20的记录为例，根据某个主键值去查找记录的步骤就可以大致拆分成下边两步：

1. 先到存储目录项记录的页，也就是页30中通过二分法快速定位到对应目录项，因为12 < 20 < 209，所以定位到对应的记录所在的页就是页9。
2. 再到存储用户记录的页9中根据二分法快速定位到主键值为20的用户记录。

虽然说目录项记录中只存储主键值和对应的页号，比用户记录需要的存储空间小多了，但是不论怎么说一个页只有16KB大小，能存放的目录项记录也是有限的，那如果表中的数据太多，以至于一个数据页不足以存放所有的目录项记录，该咋办呢？

当然是再多整一个存储目录项记录的页喽～ 为了大家更好的理解新分配一个目录项记录页的过程，我们假设一个存储目录项记录的页最多只能存放4条目录项记录（请注意是假设哦，真实情况下可以存放好多条的），所以如果此时我们再向上图中插入一条主键值为320的用户记录的话，那就需要分配一个新的存储目录项记录的页喽：



从图中可以看出，我们插入了一条主键值为320的用户记录之后需要两个新的数据页：

- 为存储该用户记录而新生成了页31。
- 因为原先存储目录项记录的页30的容量已满（我们前边假设只能存储4条目录项记录），所以不得不需要一个新的页32来存放页31对应的目录项。

现在因为存储目录项记录的页不止一个，所以如果我们想根据主键值查找一条用户记录大致需要3个步骤，以查找主键值为20的记录为例：

1. 确定目录项记录页

   我们现在的存储目录项记录的页有两个，即页30和页32，又因为页30表示的目录项的主键值的范围是[1, 320)，页32表示的目录项的主键值不小于320，所以主键值为20的记录对应的目录项记录在页30中。

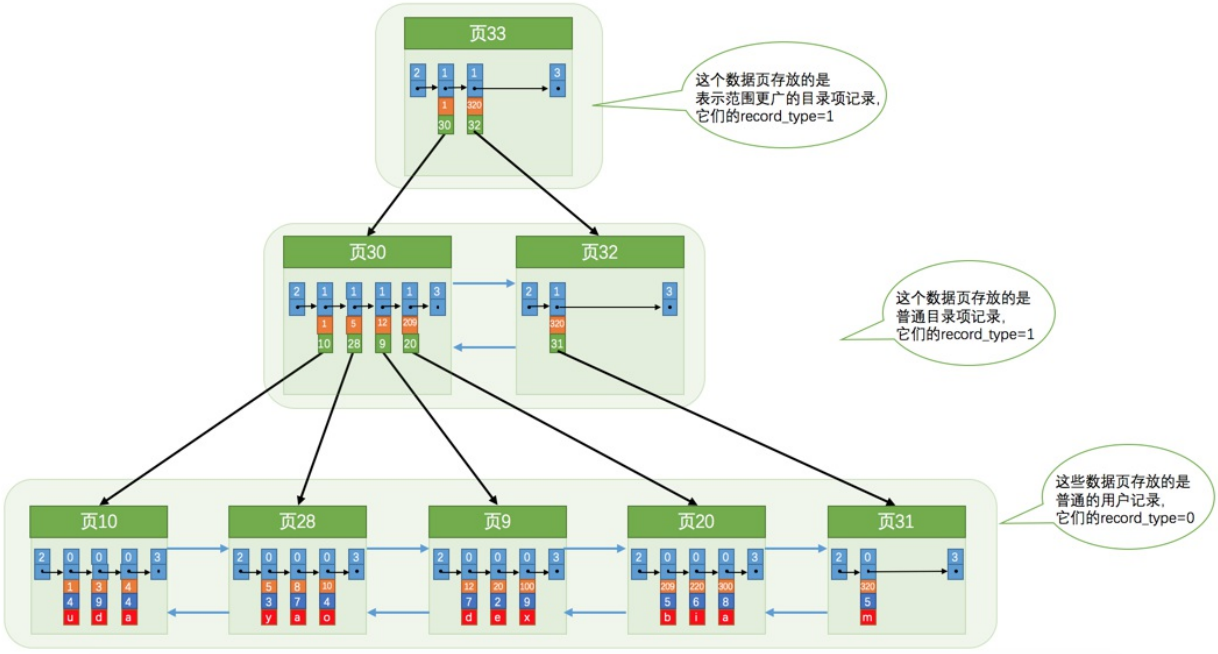
2. 通过目录项记录页确定用户记录真实所在的页。

   在一个存储目录项记录的页中通过主键值定位一条目录项记录的方式说过了，不赘述了～

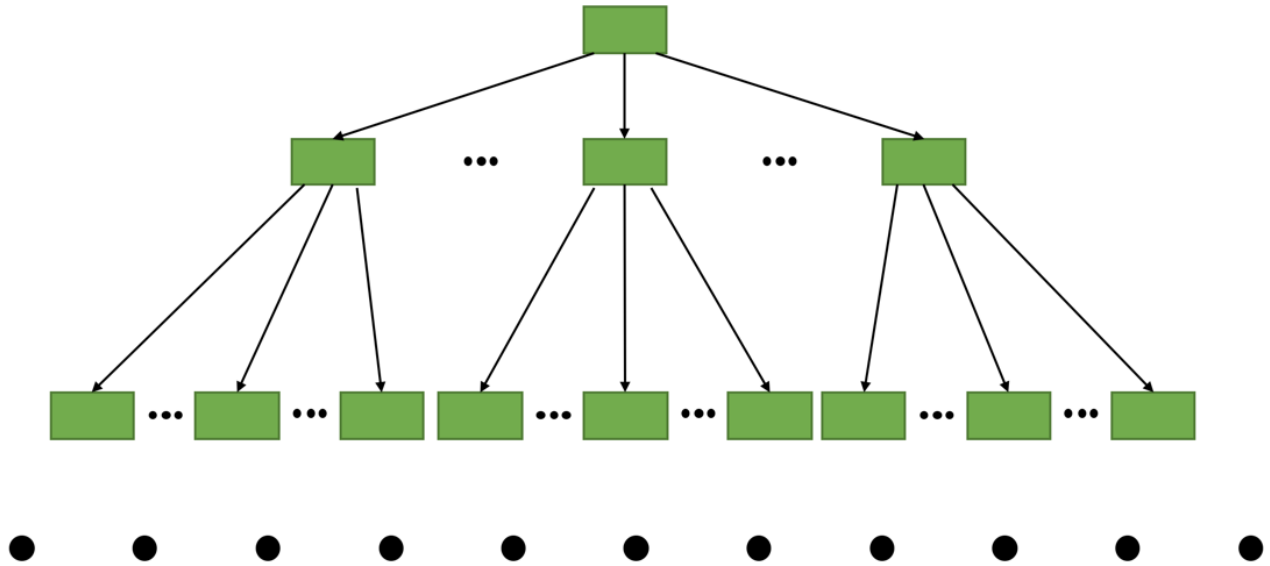
3. 在真实存储用户记录的页中定位到具体的记录。

   在一个存储用户记录的页中通过主键值定位一条用户记录的方式已经说过200遍了，你再不会我就，我就，我就求你到上一篇唠叨数据页结构的文章中多看几遍，求你了～

那么问题来了，在这个查询步骤的第1步中我们需要定位存储目录项记录的页，但是这些页在存储空间中也可能不挨着，如果我们表中的数据非常多则会产生很多存储目录项记录的页，那我们怎么根据主键值快速定位一个存储目录项记录的页呢？其实也简单，为这些存储目录项记录的页再生成一个更高级的目录，就像是一个多级目录一样，大目录里嵌套小目录，小目录里才是实际的数据，所以现在各个页的示意图就是这样子：



如图，我们生成了一个存储更高级目录项的页33，这个页中的两条记录分别代表页30和页32，如果用户记录的主键值在[1, 320)之间，则到页30中查找更详细的目录项记录，如果主键值不小于320的话，就到页32中查找更详细的目录项记录。不过这张图好漂亮喔，随着表中记录的增加，这个目录的层级会继续增加，如果简化一下，那么我们可以用下边这个图来描述它：



这玩意儿像不像一个倒过来的树呀，上头是树根，下头是树叶！其实这是一种组织数据的形式，或者说是一种数据结构，它的名称是B+树。

小贴士： 为啥叫`B+`呢，`B`树是个啥？喔对不起，这不是我们讨论的范围，你可以去找一本数据结构或算法的书来看。什么？数据结构的书看不懂？等我～

不论是存放用户记录的数据页，还是存放目录项记录的数据页，我们都把它们存放到B+树这个数据结构中了，所以我们也称这些数据页为节点。从图中可以看出来，我们的实际用户记录其实都存放在B+树的最底层的节点上，这些节点也被称为叶子节点或叶节点，其余用来存放目录项的节点称为非叶子节点或者内节点，其中B+树最上边的那个节点也称为根节点。

从图中可以看出来，一个B+树的节点其实可以分成好多层，设计InnoDB的大叔们为了讨论方便，规定最下边的那层，也就是存放我们用户记录的那层为第0层，之后依次往上加。之前的讨论我们做了一个非常极端的假设：存放用户记录的页最多存放3条记录，存放目录项记录的页最多存放4条记录。其实真实环境中一个页存放的记录数量是非常大的，假设，假设，假设所有存放用户记录的叶子节点代表的数据页可以存放100条用户记录，所有存放目录项记录的内节点代表的数据页可以存放1000条目录项记录，那么：

- 如果B+树只有1层，也就是只有1个用于存放用户记录的节点，最多能存放100条记录。
- 如果B+树有2层，最多能存放1000×100=100000条记录。
- 如果B+树有3层，最多能存放1000×1000×100=100000000条记录。
- 如果B+树有4层，最多能存放1000×1000×1000×100=100000000000条记录。哇咔咔～这么多的记录！！！

你的表里能存放100000000000条记录么？所以一般情况下，我们用到的B+树都不会超过4层，那我们通过主键值去查找某条记录最多只需要做4个页面内的查找（查找3个目录项页和一个用户记录页），又因为在每个页面内有所谓的Page Directory（页目录），所以在页面内也可以通过二分法实现快速定位记录，这不是很牛么，哈哈！

聚簇索引

我们上边介绍的B+树本身就是一个目录，或者说本身就是一个索引。它有两个特点：

1. 使用记录主键值的大小进行记录和页的排序，这包括三个方面的含义：
    - 页内的记录是按照主键的大小顺序排成一个单向链表。
    - 各个存放用户记录的页也是根据页中用户记录的主键大小顺序排成一个双向链表。
    - 存放目录项记录的页分为不同的层次，在同一层次中的页也是根据页中目录项记录的主键大小顺序排成一个双向链表。
2. B+树的叶子节点存储的是完整的用户记录。

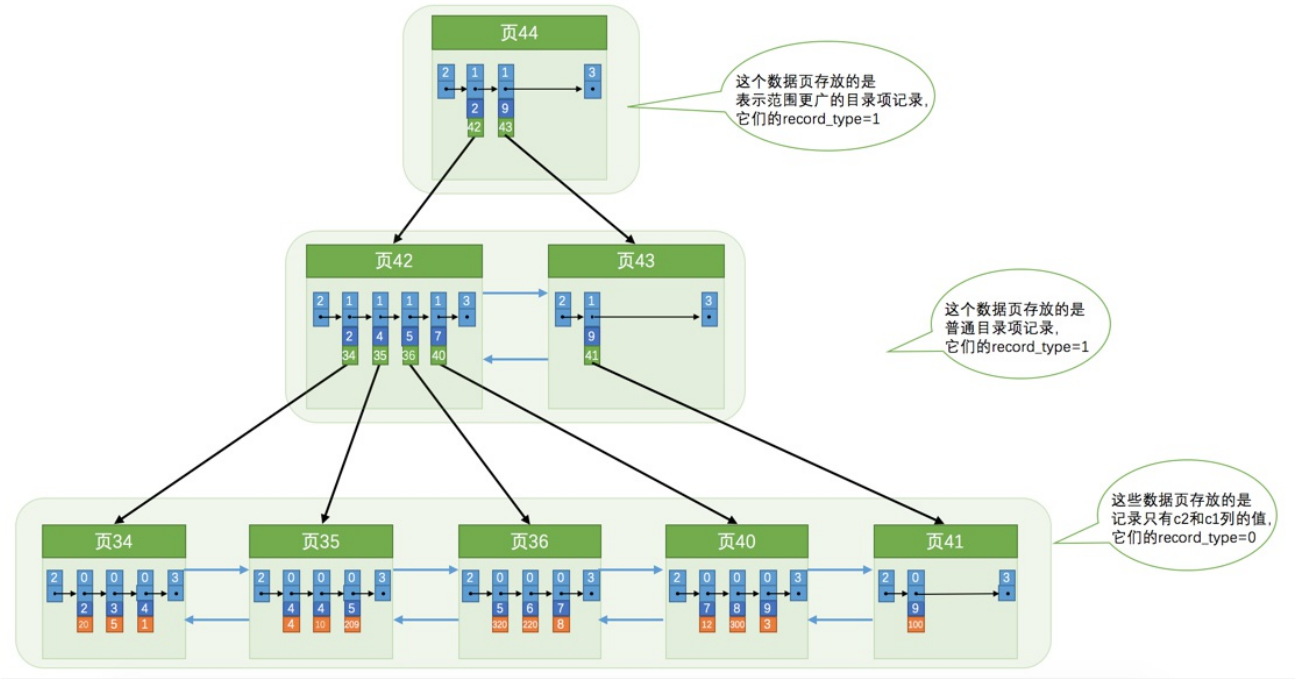
    所谓完整的用户记录，就是指这个记录中存储了所有列的值（包括隐藏列）。

我们把具有这两种特性的B+树称为聚簇索引，所有完整的用户记录都存放在这个聚簇索引的叶子节点处。这种聚簇索引并不需要我们在MySQL语句中显式的使用INDEX语句去创建（后边会介绍索引相关的语句），InnoDB存储引擎会自动的为我们创建聚簇索引。另外有趣的一点是，在InnoDB存储引擎中，聚簇索引就是数据的存储方式（所有的用户记录都存储在了叶子节点），也就是所谓的索引即数据，数据即索引。

二级索引

大家有木有发现，上边介绍的聚簇索引只能在搜索条件是主键值时才能发挥作用，因为B+树中的数据都是按照主键进行排序的。那如果我们想以别的列作为搜索条件该咋办呢？难道只能从头到尾沿着链表依次遍历记录么？

不，我们可以多建几棵B+树，不同的B+树中的数据采用不同的排序规则。比方说我们用c2列的大小作为数据页、页中记录的排序规则，再建一棵B+树，效果如下图所示：



这个B+树与上边介绍的聚簇索引有几处不同：

- 使用记录c2列的大小进行记录和页的排序，这包括三个方面的含义：
  - 页内的记录是按照c2列的大小顺序排成一个单向链表。
  - 各个存放用户记录的页也是根据页中记录的c2列大小顺序排成一个双向链表。
  - 存放目录项记录的页分为不同的层次，在同一层次中的页也是根据页中目录项记录的c2列大小顺序排成一个双向链表。
- B+树的叶子节点存储的并不是完整的用户记录，而只是c2列+主键这两个列的值。
- 目录项记录中不再是主键+页号的搭配，而变成了c2列+页号的搭配。

所以如果我们现在想通过c2列的值查找某些记录的话就可以使用我们刚刚建好的这个B+树了。以查找c2列的值为4的记录为例，查找过程如下：

1. 确定目录项记录页

   根据根页面，也就是页44，可以快速定位到目录项记录所在的页为页42（因为2 < 4 < 9）。

2. 通过目录项记录页确定用户记录真实所在的页。

   在页42中可以快速定位到实际存储用户记录的页，但是由于c2列并没有唯一性约束，所以c2列值为4的记录可能分布在多个数据页中，又因为2 < 4 ≤ 4，所以确定实际存储用户记录的页在页34和页35中。

3. 在真实存储用户记录的页中定位到具体的记录。

   到页34和页35中定位到具体的记录。

4. 但是这个B+树的叶子节点中的记录只存储了c2和c1（也就是主键）两个列，所以我们必须再根据主键值去聚簇索引中再查找一遍完整的用户记录。

各位各位，看到步骤4的操作了么？我们根据这个以c2列大小排序的B+树只能确定我们要查找记录的主键值，所以如果我们想根据c2列的值查找到完整的用户记录的话，仍然需要到聚簇索引中再查一遍，这个过程也被称为回表。也就是根据c2列的值查询一条完整的用户记录需要使用到2棵B+树！！！

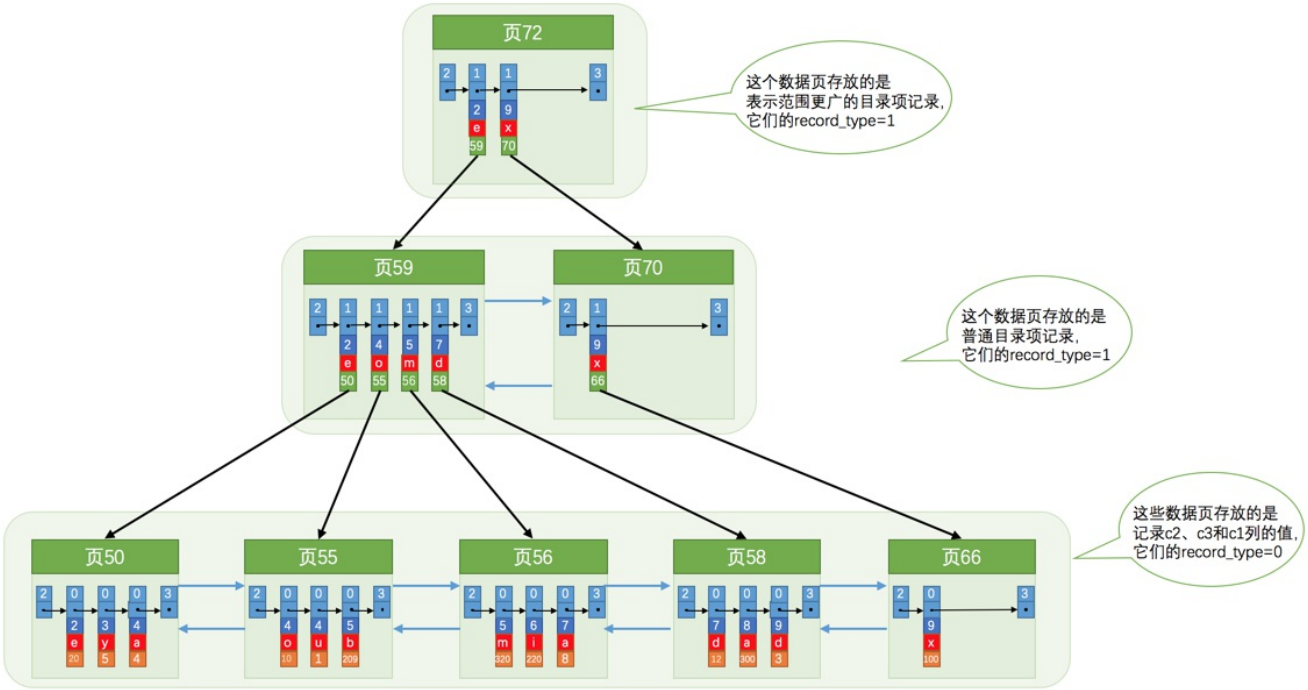
为什么我们还需要一次回表操作呢？直接把完整的用户记录放到叶子节点不就好了么？你说的对，如果把完整的用户记录放到叶子节点是可以不用回表，但是太占地方了呀～相当于每建立一棵B+树都需要把所有的用户记录再都拷贝一遍，这就有点太浪费存储空间了。因为这种按照非主键列建立的B+树需要一次回表操作才可以定位到完整的用户记录，所以这种B+树也被称为二级索引（英文名secondary index），或者辅助索引。由于我们使用的是c2列的大小作为B+树的排序规则，所以我们也称这个B+树为为c2列建立的索引。

联合索引

我们也可以同时以多个列的大小作为排序规则，也就是同时为多个列建立索引，比方说我们想让B+树按照c2和c3列的大小进行排序，这个包含两层含义：

- 先把各个记录和页按照c2列进行排序。
- 在记录的c2列相同的情况下，采用c3列进行排序

为c2和c3列建立的索引的示意图如下：



如图所示，我们需要注意一下几点：

- 每条目录项记录都由c2、c3、页号这三个部分组成，各条记录先按照c2列的值进行排序，如果记录的c2列相同，则按照c3列的值进行排序。
- B+树叶子节点处的用户记录由c2、c3和主键c1列组成。

千万要注意一点，以c2和c3列的大小为排序规则建立的B+树称为联合索引，本质上也是一个二级索引。它的意思与分别为c2和c3列分别建立索引的表述是不同的，不同点如下：

- 建立联合索引只会建立如上图一样的1棵B+树。
- 为c2和c3列分别建立索引会分别以c2和c3列的大小为排序规则建立2棵B+树。

InnoDB的B+树索引的注意事项

根页面万年不动窝

我们前边介绍B+树索引的时候，为了大家理解上的方便，先把存储用户记录的叶子节点都画出来，然后接着画存储目录项记录的内节点，实际上B+树的形成过程是这样的：

- 每当为某个表创建一个B+树索引（聚簇索引不是人为创建的，默认就有）的时候，都会为这个索引创建一个根节点页面。最开始表中没有数据的时候，每个B+树索引对应的根节点中既没有用户记录，也没有目录项记录。
- 随后向表中插入用户记录时，先把用户记录存储到这个根节点中。
- 当根节点中的可用空间用完时继续插入记录，此时会将根节点中的所有记录复制到一个新分配的页，比如页a中，然后对这个新页进行页分裂的操作，得到另一个新页，比如页b。这时新插入的记录根据键值（也就是聚簇索引中的主键值，二级索引中对应的索引列的值）的大小就会被分配到页a或者页b中，而根节点便升级为存储目录项记录的页。

这个过程需要大家特别注意的是：一个B+树索引的根节点自诞生之日起，便不会再移动。这样只要我们对某个表建立一个索引，那么它的根节点的页号便会被记录到某个地方，然后凡是InnoDB存储引擎需要用到这个索引的时候，都会从那个固定的地方取出根节点的页号，从而来访问这个索引。

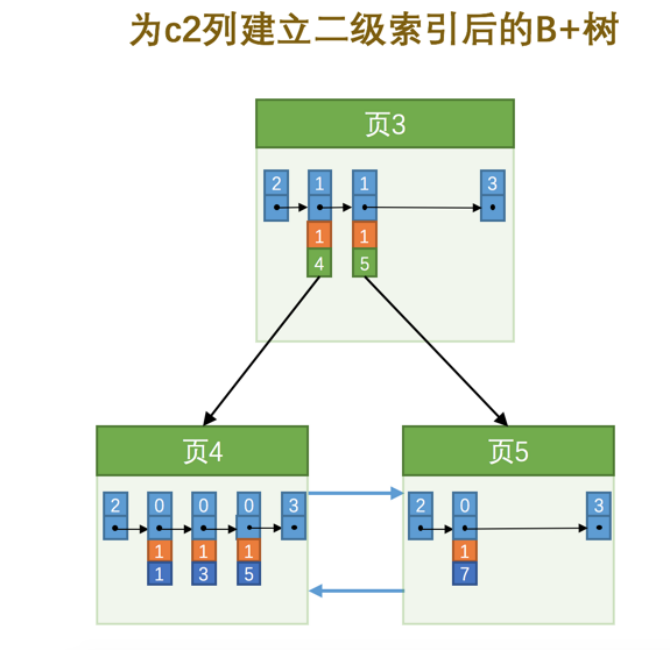
> 小贴士：跟大家剧透一下，这个存储某个索引的根节点在哪个页面中的信息就是传说中的数据字典中的一项信息，关于更多数据字典的内容，后边会详细唠叨，别着急哈。

内节点中目录项记录的唯一性

我们知道B+树索引的内节点中目录项记录的内容是索引列 + 页号的搭配，但是这个搭配对于二级索引来说有点儿不严谨。还拿index_demo表为例，假设这个表中的数据是这样的：

```
c1 c2 c3
1  1  'u'
3  1  'd'
5  1  'y'
7  1  'a'
```

如果二级索引中目录项记录的内容只是索引列 + 页号的搭配的话，那么为c2列建立索引后的B+树应该长这样：

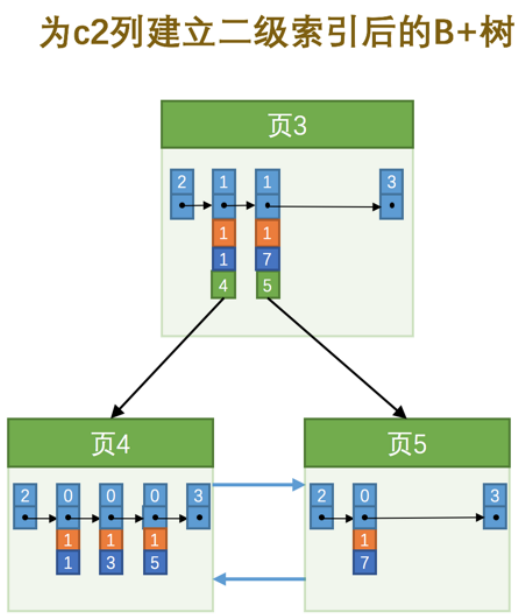


如果我们想新插入一行记录，其中c1、c2、c3的值分别是：9、1、'c'，那么在修改这个为c2列建立的二级索引对应的B+树时便碰到了个大问题：由于页3中存储的目录项记录是由c2列 + 页号的值构成的，页3中的两条目录项记录对应的c2列的值都是1，而我们新插入的这条记录的c2列的值也是1，那我们这条新插入的记录到底应该放到页4中，还是应该放到页5中啊？答案是：对不起，懵逼了。

为了让新插入记录能找到自己在那个页里，我们需要保证在B+树的同一层内节点的目录项记录除页号这个字段以外是唯一的。所以对于二级索引的内节点的目录项记录的内容实际上是由三个部分构成的：

- 索引列的值
- 主键值
- 页号

也就是我们把主键值也添加到二级索引内节点中的目录项记录了，这样就能保证B+树每一层节点中各条目录项记录除页号这个字段外是唯一的，所以我们为c2列建立二级索引后的示意图实际上应该是这样子的：



这样我们再插入记录(9, 1, 'c')时，由于页3中存储的目录项记录是由c2列 + 主键 + 页号的值构成的，可以先把新记录的c2列的值和页3中各目录项记录的c2列的值作比较，如果c2列的值相同的话，可以接着比较主键值，因为B+树同一层中不同目录项记录的c2列 + 主键的值肯定是不一样的，所以最后肯定能定位唯一的一条目录项记录，在本例中最后确定新记录应该被插入到页5中。

一个页面最少存储2条记录

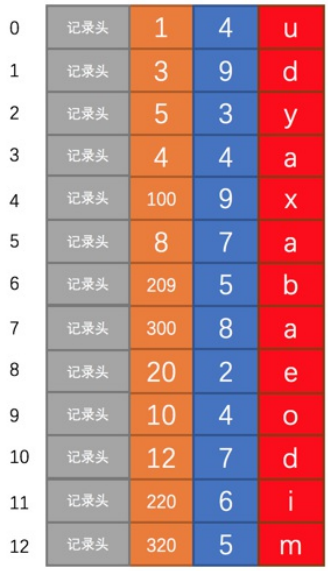
我们前边说过一个B+树只需要很少的层级就可以轻松存储数亿条记录，查询速度杠杠的！这是因为B+树本质上就是一个大的多层级目录，每经过一个目录时都会过滤掉许多无效的子目录，直到最后访问到存储真实数据的目录。那如果一个大的目录中只存放一个子目录是个啥效果呢？那就是目录层级非常非常非常多，而且最后的那个存放真实数据的目录中只能存放一条记录。费了半天劲只能存放一条真实的用户记录？逗我呢？所以InnoDB的一个数据页至少可以存放两条记录，这也是我们之前唠叨记录行格式的时候说过一个结论（我们当时依据这个结论推导了表中只有一个列时该列在不发生行溢出的情况下最多能存储多少字节，忘了的话回去看看吧）。

MyISAM中的索引方案简单介绍

至此，我们介绍的都是InnoDB存储引擎中的索引方案，为了内容的完整性，以及各位可能在面试的时候遇到这类的问题，我们有必要再简单介绍一下MyISAM存储引擎中的索引方案。我们知道InnoDB中索引即数据，也就是聚簇索引的那棵B+树的叶子节点中已经把所有完整的用户记录都包含了，而MyISAM的索引方案虽然也使用树形结构，但是却将索引和数据分开存储：

- 将表中的记录按照记录的插入顺序单独存储在一个文件中，称之为数据文件。这个文件并不划分为若干个数据页，有多少记录就往这个文件中塞多少记录就成了。我们可以通过行号而快速访问到一条记录。

MyISAM记录也需要记录头信息来存储一些额外数据，我们以上边唠叨过的index_demo表为例，看一下这个表中的记录使用MyISAM作为存储引擎在存储空间中的表示：



由于在插入数据的时候并没有刻意按照主键大小排序，所以我们并不能在这些数据上使用二分法进行查找。

- 使用MyISAM存储引擎的表会把索引信息另外存储到一个称为索引文件的另一个文件中。MyISAM会单独为表的主键创建一个索引，只不过在索引的叶子节点中存储的不是完整的用户记录，而是主键值 + 行号的组合。也就是先通过索引找到对应的行号，再通过行号去找对应的记录！

  这一点和InnoDB是完全不相同的，在InnoDB存储引擎中，我们只需要根据主键值对聚簇索引进行一次查找就能找到对应的记录，而在MyISAM中却需要进行一次回表操作，意味着MyISAM中建立的索引相当于全部都是二级索引！

- 如果有需要的话，我们也可以对其它的列分别建立索引或者建立联合索引，原理和InnoDB中的索引差不多，不过在叶子节点处存储的是相应的列 + 行号。这些索引也全部都是二级索引。

  小贴士： MyISAM的行格式有定长记录格式（Static）、变长记录格式（Dynamic）、压缩记录格式（Compressed）。上边用到的index_demo表采用定长记录格式，也就是一条记录占用存储空间的大小是固定的，这样就可以轻松算出某条记录在数据文件中的地址偏移量。但是变长记录格式就不行了，MyISAM会直接在索引叶子节点处存储该条记录在数据文件中的地址偏移量。通过这个可以看出，MyISAM的回表操作是十分快速的，因为是拿着地址偏移量直接到文件中取数据的，反观InnoDB是通过获取主键之后再去聚簇索引里边儿找记录，虽然说也不慢，但还是比不上直接用地址去访问。 此处我们只是非常简要的介绍了一下MyISAM的索引，具体细节全拿出来又可以写一篇文章了。这里只是希望大家理解InnoDB中的索引即数据，数据即索引，而MyISAM中却是索引是索引、数据是数据。

MySQL中创建和删除索引的语句

光顾着唠叨索引的原理了，那我们如何使用MySQL语句去建立这种索引呢？InnoDB和MyISAM会自动为主键或者声明为UNIQUE的列去自动建立B+树索引，但是如果我们想为其他的列建立索引就需要我们显式的去指明。为啥不自动为每个列都建立个索引呢？别忘了，每建立一个索引都会建立一棵B+树，每插入一条记录都要维护各个记录、数据页的排序关系，这是很费性能和存储空间的。

我们可以在创建表的时候指定需要建立索引的单个列或者建立联合索引的多个列：

```
CREATE TALBE 表名 (
    各种列的信息 ··· ,
    [KEY|INDEX] 索引名 (需要被索引的单个列或多个列)
)
```

其中的KEY和INDEX是同义词，任意选用一个就可以。我们也可以在修改表结构的时候添加索引：

```
ALTER TABLE 表名 ADD [INDEX|KEY] 索引名 (需要被索引的单个列或多个列);
```

也可以在修改表结构的时候删除索引：

```
ALTER TABLE 表名 DROP [INDEX|KEY] 索引名;
```

比方说我们想在创建index_demo表的时候就为c2和c3列添加一个联合索引，可以这么写建表语句：

```
CREATE TABLE index_demo(
    c1 INT,
    c2 INT,
    c3 CHAR(1),
    PRIMARY KEY(c1),
    INDEX idx_c2_c3 (c2, c3)
);
```

在这个建表语句中我们创建的索引名是idx_c2_c3，这个名称可以随便起，不过我们还是建议以idx_为前缀，后边跟着需要建立索引的列名，多个列名之间用下划线_分隔开。

如果我们想删除这个索引，可以这么写：

```
ALTER TABLE index_demo DROP INDEX idx_c2_c3;
```