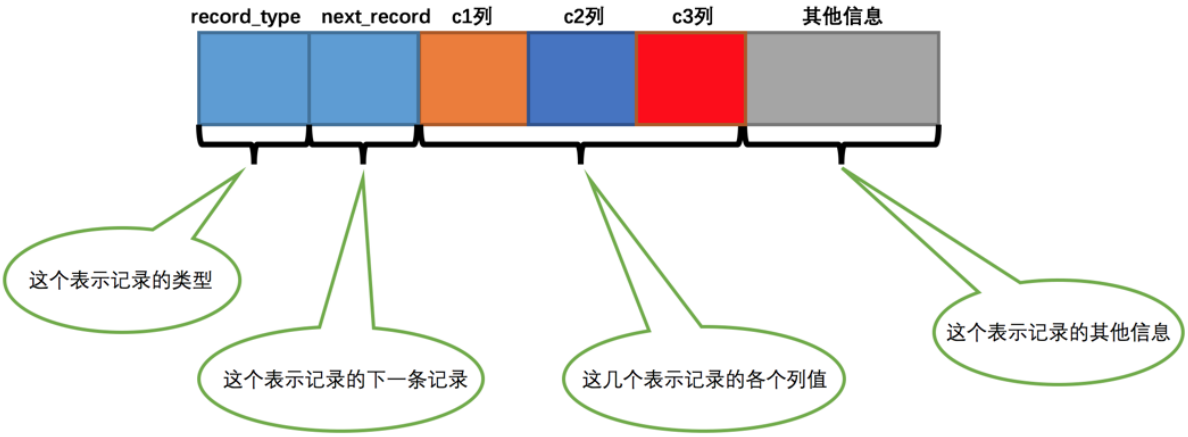
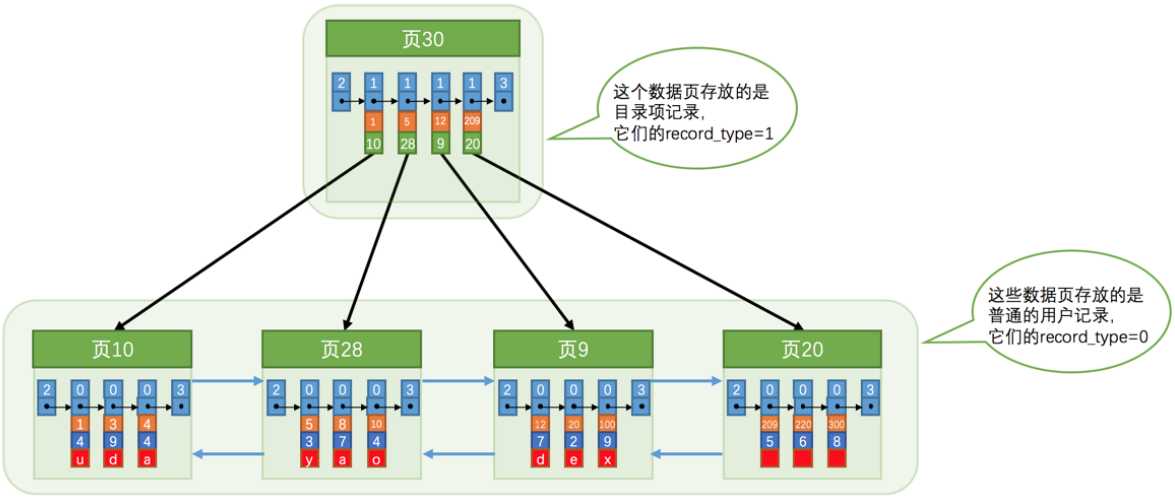
索引

概念

- 首先建表CREATE TABLE index_demo(-> c1 INT, -> c2 INT, -> c3 CHAR(1), -> PRIMARY KEY(c1) ->) ROW_FORMAT = Compact;
- 简化后的行格式示意图



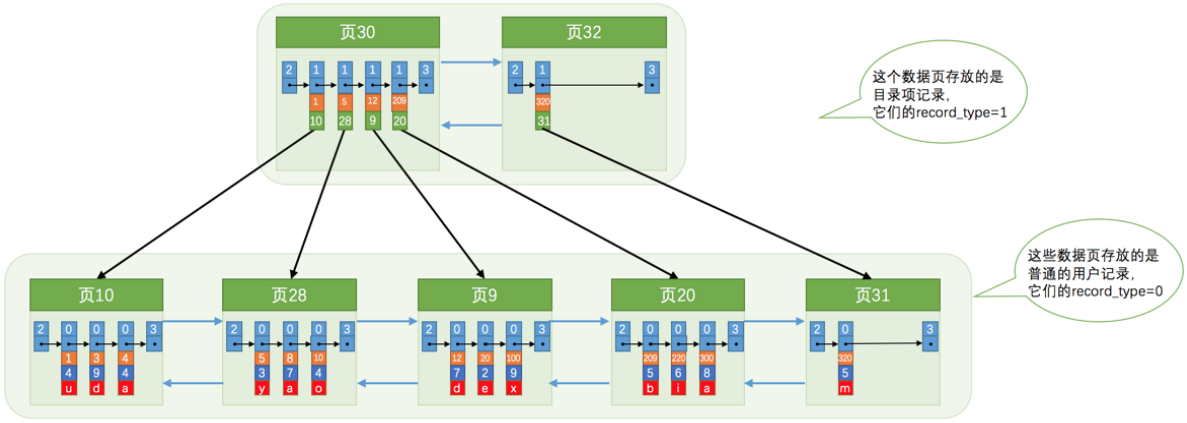
- 参数解析
  - `record_type`：记录头信息的一项属性，表示记录的类型，`0`表示普通记录、`2`表示最小记录、`3`表示最大记录、`1`表示目录项记录
  - `next_record`：记录头信息的一项属性，表示下一条地址相对于本条记录的地址偏移量，为了方便大家理解，我们都会用箭头来表明下一条记录是谁。
- 利用record_type去实现区分普通记录还是目录项记录



- 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。
- 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？

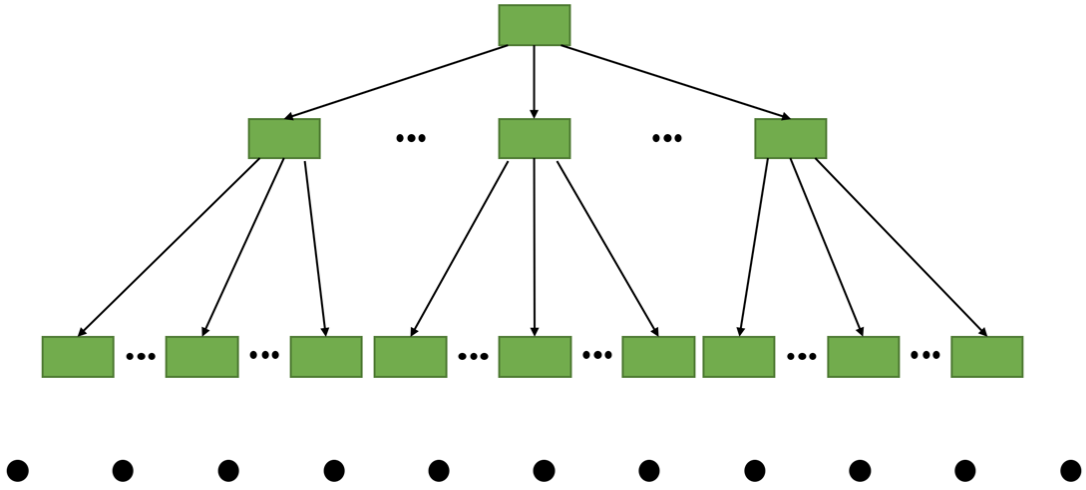
  当然是再多整一个存储目录项记录的页喽～ 为了大家更好的理解新分配一个目录项记录页的过程，我们假设一个存储目录项记录的页最多只能存放4条目录项记录（请注意是假设哦，真实情况下可以存放好多条的），所以如果此时我们再向上图中插入一条主键值为320的用户记录的话，那就需要分配一个新的存储目录项记录的页喽：



    - 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。
- B+树
    - 结构图



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`。

- 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。
- InnoDB 存储引擎中页的大小为 16KB，一般表的主键类型为 INT（占用4个字节） 或 BIGINT（占用8个字节），指针类型也一般为 4 或 8 个字节，也就是说一个页（B+Tree 中的一个节点）中大概存储 16KB/(8B+8B)=1K 个键值（因为是估值，为方便计算，这里的 K 取值为〖10〗^3）。也就是说一个深度为 3 的 B+Tree 索引可以维护10^3 *10^3* 10^3 = 10亿 条记录。实际情况中每个节点可能不能填充满，因此在数据库中，B+Tree 的高度一般都在 2~4 层。MySQL 的 InnoDB 存储引擎在设计时是将根节点常驻内存的，也就是说查找某一键值的行记录时最多只需要 1~3 次磁盘 I/O 操作。

聚簇索引

- B+树本身就是一个目录或者说是索引，它有两个特点
  - 使用主键的大小进行记录和页排序，包括三方面的含义：
    - 页内记录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。
  - **B+树的叶子节点存储的是完整的用户记录**
- 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。

- **聚簇索引只能在搜索条件是主键值时才能发挥作用**

二级索引

- 以别的列作为搜索条件
- 可以新建一颗B+树，例如`c2`列的大小作为数据页，页中记录的排序规则而不是主键。
- `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

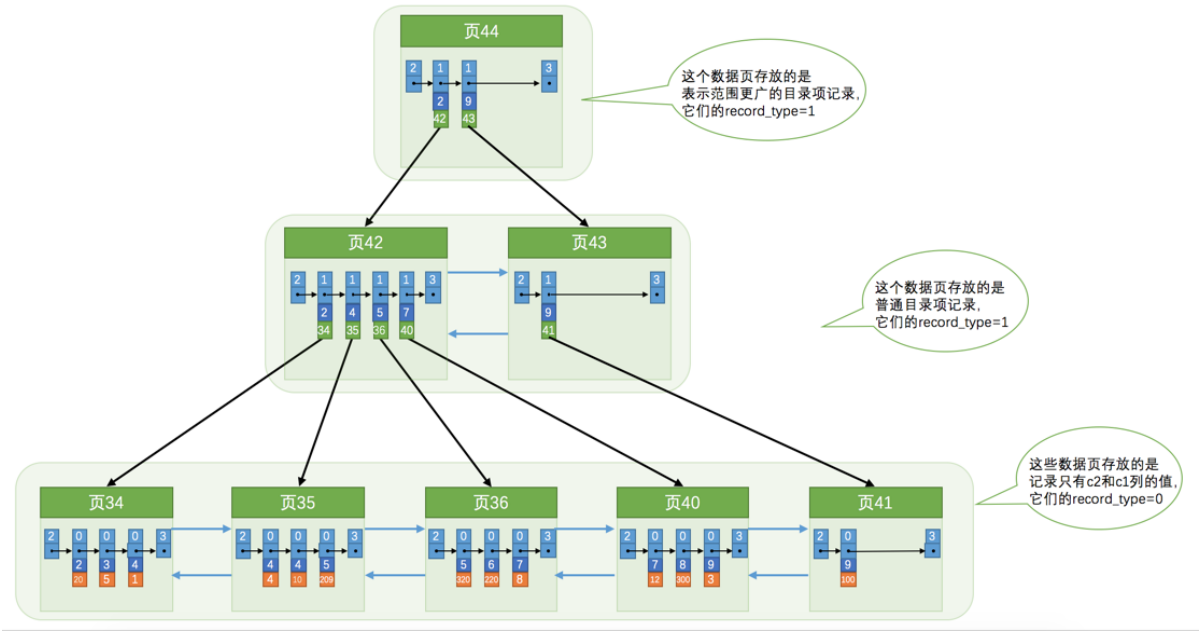
     根据`根页面`，也就是`页44`，可以快速定位到`目录项记录`所在的页为`页42`（因为`2 < 4 < 9`）。
  2. 通过`目录项记录`页确定用户记录真实所在的页。

     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。

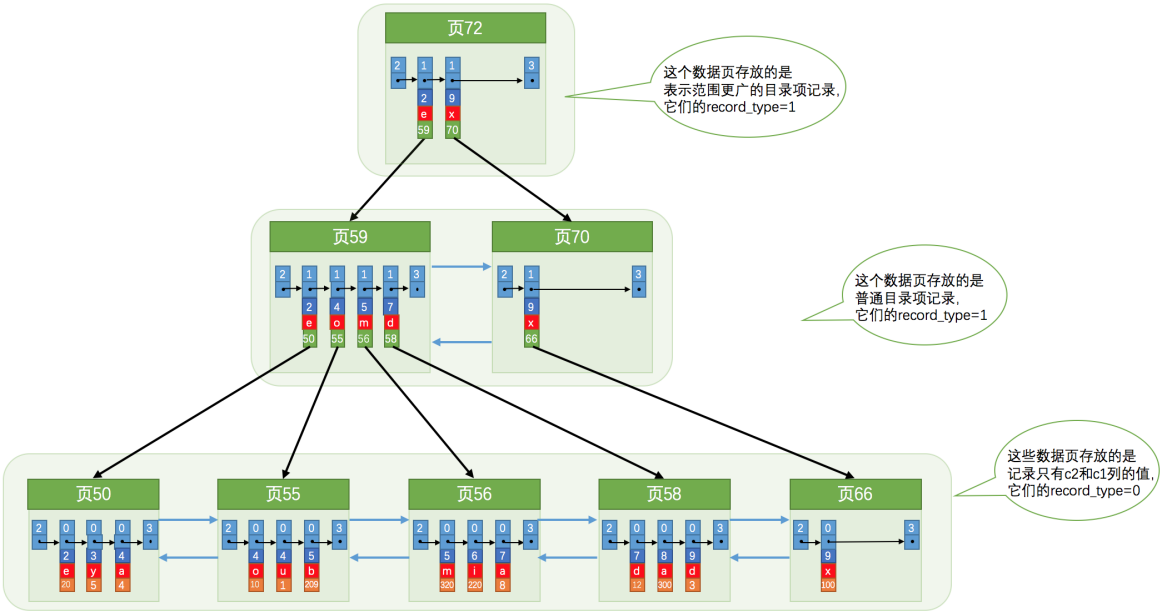
  看到步骤4的操作了么？我们根据这个以`c2`列大小排序的`B+`树只能确定我们要查找记录的主键值，所以如果我们想根据`c2`列的值查找到完整的用户记录的话，仍然需要到**聚簇索引**中再查一遍，这个过程也被称为**回表**。也就是根据`c2`列的值查询一条完整的用户记录需要使用到`2`棵`B+`树！！！

  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
- 此时B+树结构



- 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。

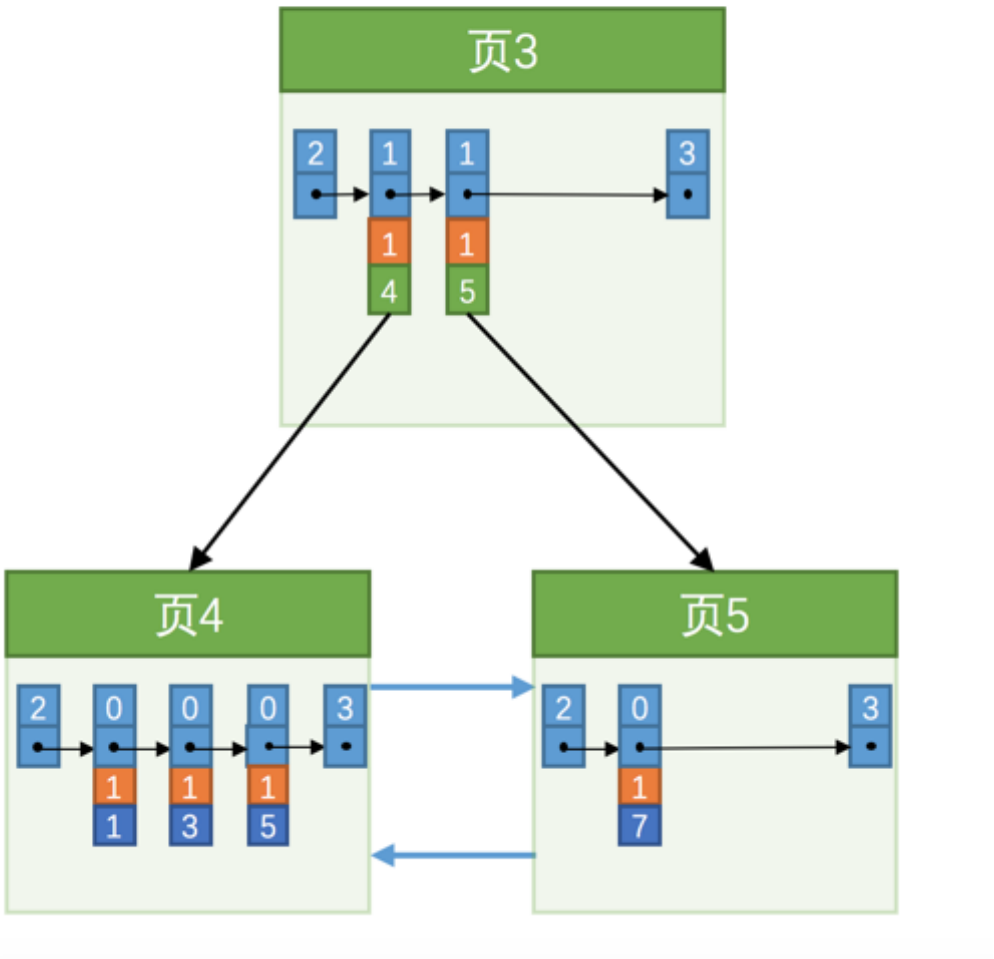
- 千万要注意一点，以c2和c3列的大小为排序规则建立的`B+`树称为**联合索引**，它的意思与分别为c2和c3列分别建立索引的表述是不同的，不同点如下：
  - 建立`联合索引`只会建立如上图一样的1棵`B+`树。
  - 为c2和c3列分别建立索引会分别以`c2`和`c3`列的大小为排序规则建立2棵`B+`树。

注意事项

- B+树的形成过程
  - 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，也没有目录项目录。
  - 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。
- 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	a
3	1	b
5	1	c
7	1	d

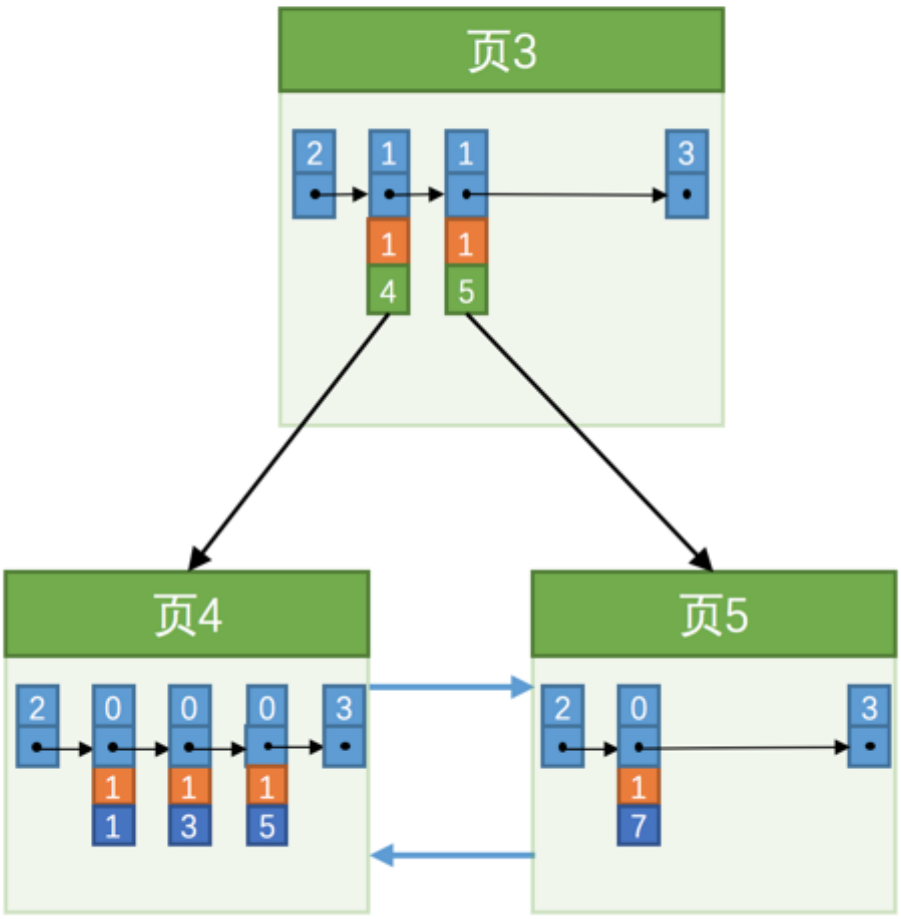
  - 如果二级索引中目录项记录的内容只是`索引列 + 页号`的搭配的话，那么为`c2`列建立索引后的`B+`树应该长这样：



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?

- 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。

- 一个页面最少存储两条记录
  - 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。

总结

- 对于`InnoDB`存储引擎来说，在单个页中查找某条记录分为两种情况：
  - 以主键为搜索条件，可以使用`Page Directory`通过二分法快速定位相应的用户记录。

    - 以其他列为搜索条件，需要按照记录组成的单链表依次遍历各条记录。
- `InnoDB`存储引擎的索引是一棵`B+`树，完整的用户记录都存储在`B+`树第`0`层的叶子节点，其他层次的节点都属于`内节点`，`内节点`里存储的是`目录项记录`。`InnoDB`的索引分为两大种：
    - 聚簇索引

      以主键值的大小为页和记录的排序规则，在叶子节点处存储的记录包含了表中所有的列。
    - 二级索引

      以自定义的列的大小为页和记录的排序规则，在叶子节点处存储的记录内容是`列 + 主键`。
- `MyISAM`存储引擎的数据和索引分开存储，这种存储引擎的索引全部都是`二级索引`，在叶子节点处存储的是`列 + 行号`。

面试题

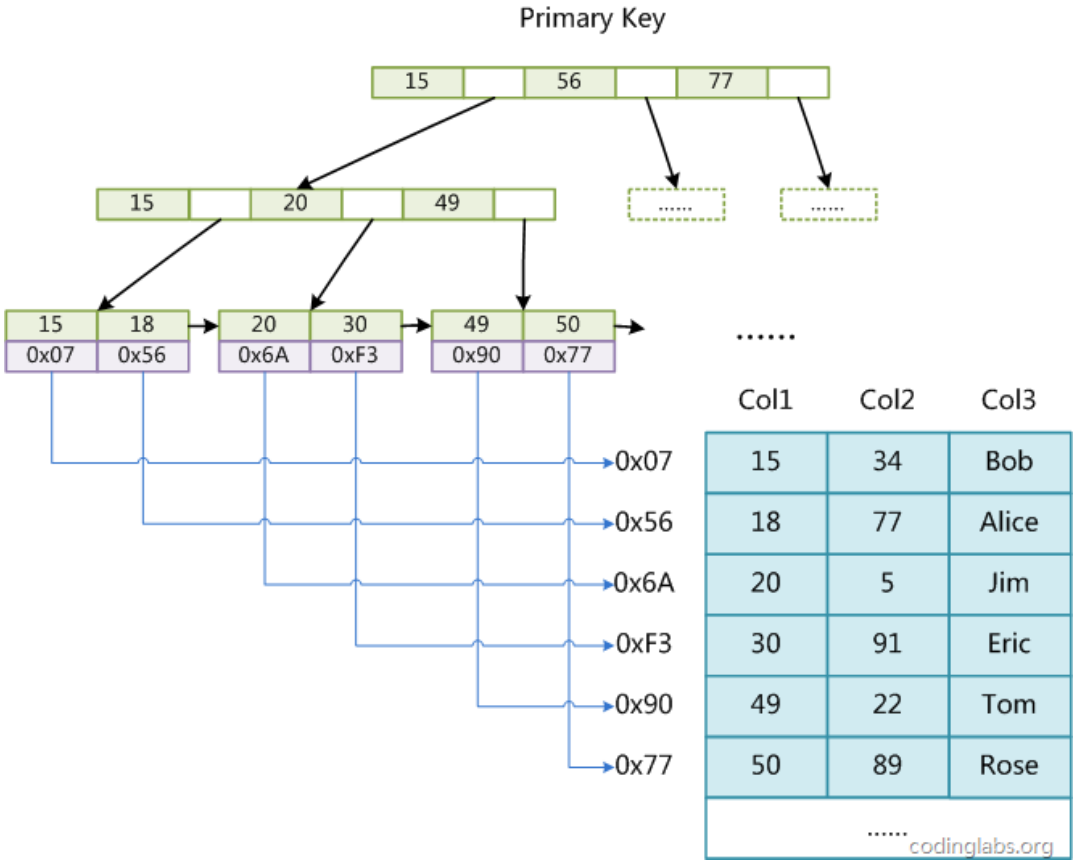
- B+树的查询时间复杂度log(n)
- 为什么索引不用B树，红黑树
    - 不用B树：文件系统和数据库的索引一般都是存在硬盘上，如果数据量较大的话不能一次性加载到内存中，而使用B系列树可以做到每次只加载一个节点到内存中，所以在内存中红黑树效率比B树高，但涉及到磁盘操作，B树就更优了，而B+树是在B树的基础上进行改造，它的数据都在叶子节点，同时叶子节点之间还增加了指针形成双向链表。而使用B+树的优点则是当数据库进行select操作时，可能会存在多条返回结果，这时如果使用B树的话可能会存在跨层访问，而B+树由于所有数据都在叶子节点，不用跨层。
    - 不用红黑树：因为红黑树父节点只有两个子节点，如果用做MySQL索引会导致树变得非常高。
    - 不用hash是因为查询单条数据时可能hash更快，但多条时因为B+树叶子节点有序，又有链表相连，它的查询效率就会比hash快。
- 为什么innodb不用hash索引
    - Hash 索引仅仅能满足"=","IN"和"<=>"查询，不能使用范围查询。
        - 由于 Hash 索引比较的是进行 Hash 运算之后的 Hash 值，所以它只能用于等值的过滤，不能用于基于范围的过滤，因为经过相应的 Hash 算法处理之后的 Hash 值的大小关系，并不能保证和Hash运算前完全一样。
    - Hash 索引无法被用来进行数据的排序操作。
        - 由于 Hash 索引中存放的是经过 Hash 计算之后的 Hash 值，而且Hash值的大小关系并不一定和 Hash 运算前的键值完全一样，所以数据库无法利用索引的数据来进行任何排序运算。
    - Hash 索引不能利用部分索引键查询。
        - 对于组合索引，Hash 索引在计算 Hash 值的时候是组合索引键合并后再一起计算 Hash 值，而不是单独计算 Hash 值，所以通过组合索引的前面一个或几个索引键进行查询的时候，Hash 索引也无法被利用。
    - Hash 索引在任何时候都不能避免表扫描。
        - 前面已经知道，**Hash 索引**是将索引键通过 Hash 运算之后，将 Hash运算结果的 Hash 值和所对应的行指针信息存放于一个 Hash 表中，由于不同索引键存在相同 Hash 值，所以即使取满足某个 Hash 键值的数据的记录条数，也无法从 Hash 索引中直接完成查

询，还是要通过访问表中的实际数据进行相应的比较，并得到相应的结果。

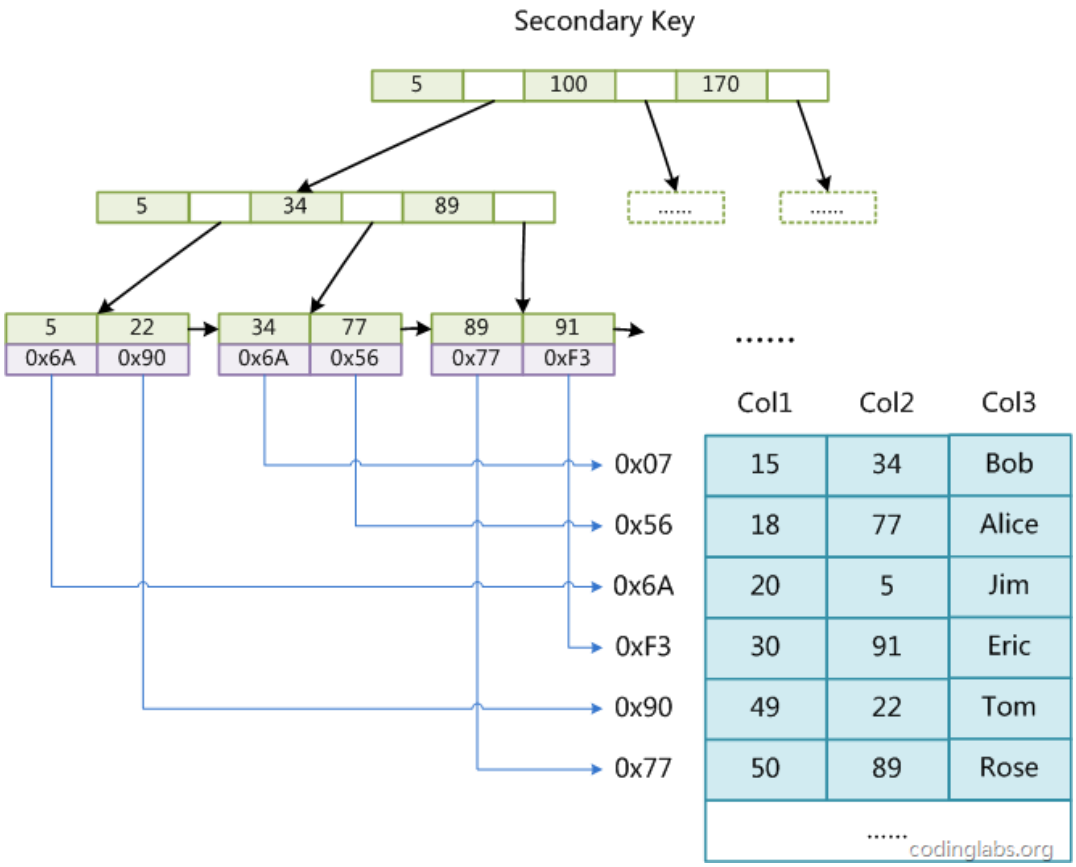
- Hash 索引遇到大量Hash值相等的情况后性能并不一定就会比B-Tree索引高。
  - 对于**选择性**比较低的索引键，如果创建 Hash 索引，那么将会存在大量记录指针信息存于同一个 Hash 值相关联。这样要定位某一条记录时就会非常麻烦，会浪费多次表数据的访问，而造成整体性能低下。
- Hash索引的优势在于等值查询。前提是键值都是唯一的。如果键值不是唯一的，就需要先找到该键所在位置，然后再根据链表往后扫描，直到找到相应的数据。

MyISAM 索引实现

- 主键索引
  - MyISAM引 擎使用B+Tree作为索引结构，**叶节点的data域存放的是数据记录的地址**。下图是MyISAM主键索引的原理图：



  - 这里假设表一共有三列，假设我们以 Col1 为主键，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。
- 辅助索引
  - **在 MyISAM 中，主索引和辅助索引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。**如果我们在 Col2 上建立一个辅助索引，则此索引的结构如下图所示：



- 同样也是一颗 B+Tree ， data 域保存数据记录的地址。因此， **MyISAM 中索引检索的算法为首先按照 B+Tree 搜索算法搜索索引，如果指定的 Key 存在，则取出其 data 域的值，然后以 data 域的值为地址，读取相应数据记录。**