单表访问方法

标签： MySQL 是怎样运行的

对于我们这些MySQL的使用者来说，MySQL其实就是一个软件，平时用的最多的就是查询功能。DBA时不时丢过来一些慢查询语句让优化，我们如果连查询是怎么执行的都不清楚还优化个毛线，所以是时候掌握真正的技术了。我们在第一章的时候就曾说过，MySQL Server有一个称为查询优化器的模块，一条查询语句进行语法解析之后就会被交给查询优化器来进行优化，优化的结果就是生成一个所谓的执行计划，这个执行计划表明了应该使用哪些索引进行查询，表之间的连接顺序是啥样的，最后会按照执行计划中的步骤调用存储引擎提供的方法来真正的执行查询，并将查询结果返回给用户。不过查询优化这个主题有点儿大，在学会跑之前还得先学会走，所以本章先来瞅瞅MySQL怎么执行单表查询（就是FROM子句后边只有一个表，最简单的那种查询～）。不过需要强调的一点是，在学习本章前务必看过前边关于记录结构、数据页结构以及索引的部分，如果你不能保证这些东西已经完全掌握，那么本章不适合你。

为了故事的顺利发展，我们先得有个表：

```
CREATE TABLE single_table (
    id INT NOT NULL AUTO_INCREMENT,
    key1 VARCHAR(100),
    key2 INT,
    key3 VARCHAR(100),
    key_part1 VARCHAR(100),
    key_part2 VARCHAR(100),
    key_part3 VARCHAR(100),
    common_field VARCHAR(100),
    PRIMARY KEY (id),
    KEY idx_key1 (key1),
    UNIQUE KEY idx_key2 (key2),
    KEY idx_key3 (key3),
    KEY idx_key_part(key_part1, key_part2,
key_part3)
) Engine=InnoDB CHARSET=utf8;
```

我们为这个single_table表建立了1个聚簇索引和4个二级索引，分别是：

- 为id列建立的聚簇索引。
- 为key1列建立的idx_key1二级索引。
- 为key2列建立的idx_key2二级索引，而且该索引是唯一二级索引。
- 为key3列建立的idx_key3二级索引。
- 为key_part1、key_part2、key_part3列建立的idx_key_part二级索引，这也是一个联合索引。

然后我们需要为这个表插入 10000 行记录，除id列外其余的列都插入随机值就好了，具体的插入语句我就不写了，自己写个程序插入吧（id列是自增主键列，不需要我们手动插入）。

访问方法（access method）的概念

想必各位都用过高德地图来查找到某个地方的路线吧（此处没有为高德地图打广告的意思，他们没给我钱，大家用百度地图也可以啊），如果我们搜西安钟楼到大雁塔之间的路线的话，地图软件会给出 n 种路线供我们选择，如果我们实在闲的没事儿干并且足够有钱的话，还可以用南辕北辙的方式绕地球一圈到达目的地。也就是说，不论采用哪一种方式，我们最终的目标就是到达大雁塔这个地方。回到MySQL中来，我们平时所写的那些查询语句本质上只是一种声明式的语法，只是告诉MySQL我们要获取的数据符合哪些规则，至于MySQL背地里是怎么把查询结果搞出来的那是MySQL自己的事儿。对于单个表的查询来说，设计 MySQL 的大叔把查询的执行方式大致分为下边两种：

- 使用全表扫描进行查询

  这种执行方式很好理解，就是把表的每一行记录都扫一遍嘛，把符合搜索条件的记录加入到结果集就完了。不管是啥查询都可以使用这种方式执行，当然，这种也是最笨的执行方式。

- 使用索引进行查询

  因为直接使用全表扫描的方式执行查询要遍历好多记录，所以代价可能太大了。如果查询语句中的搜索条件可以使用到某个索引，那直接使用索引来执行查询可能会加快查询执行的时间。使用索引来执行查询的方式五花八门，又可以细分为许多种类：

  - 针对主键或唯一二级索引的等值查询

- 针对普通二级索引的等值查询
- 针对索引列的范围查询
- 直接扫描整个索引

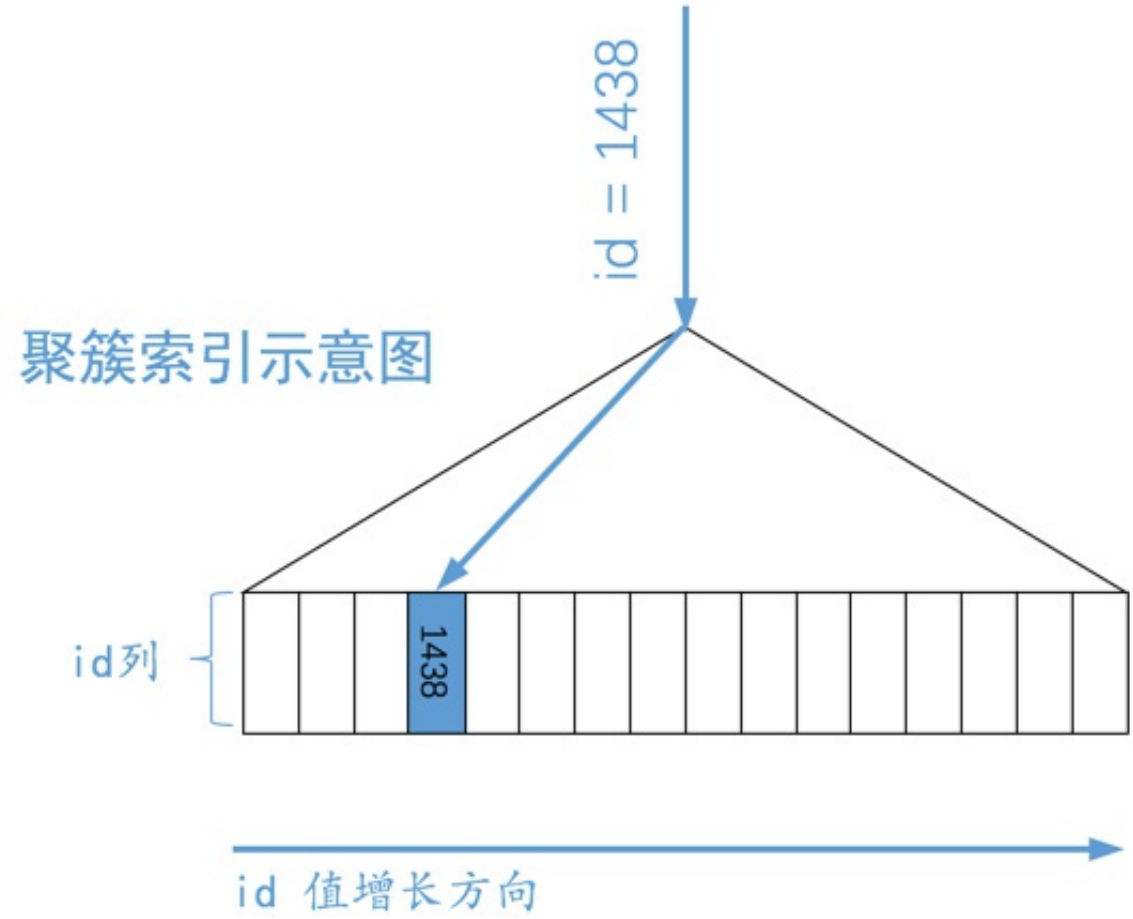
设计MySQL的大叔把MySQL执行查询语句的方式称之为访问方法或者访问类型。同一个查询语句可能可以使用多种不同的访问方法来执行，虽然最后的查询结果都是一样的，但是执行的时间可能差老鼻子远了，就像是从钟楼到大雁塔，你可以坐火箭去，也可以坐飞机去，当然也可以坐乌龟去。下边细细道来各种访问方法的具体内容。

const

有的时候我们可以通过主键列来定位一条记录，比方说这个查询：

```
SELECT * FROM single_table WHERE id = 1438;
```

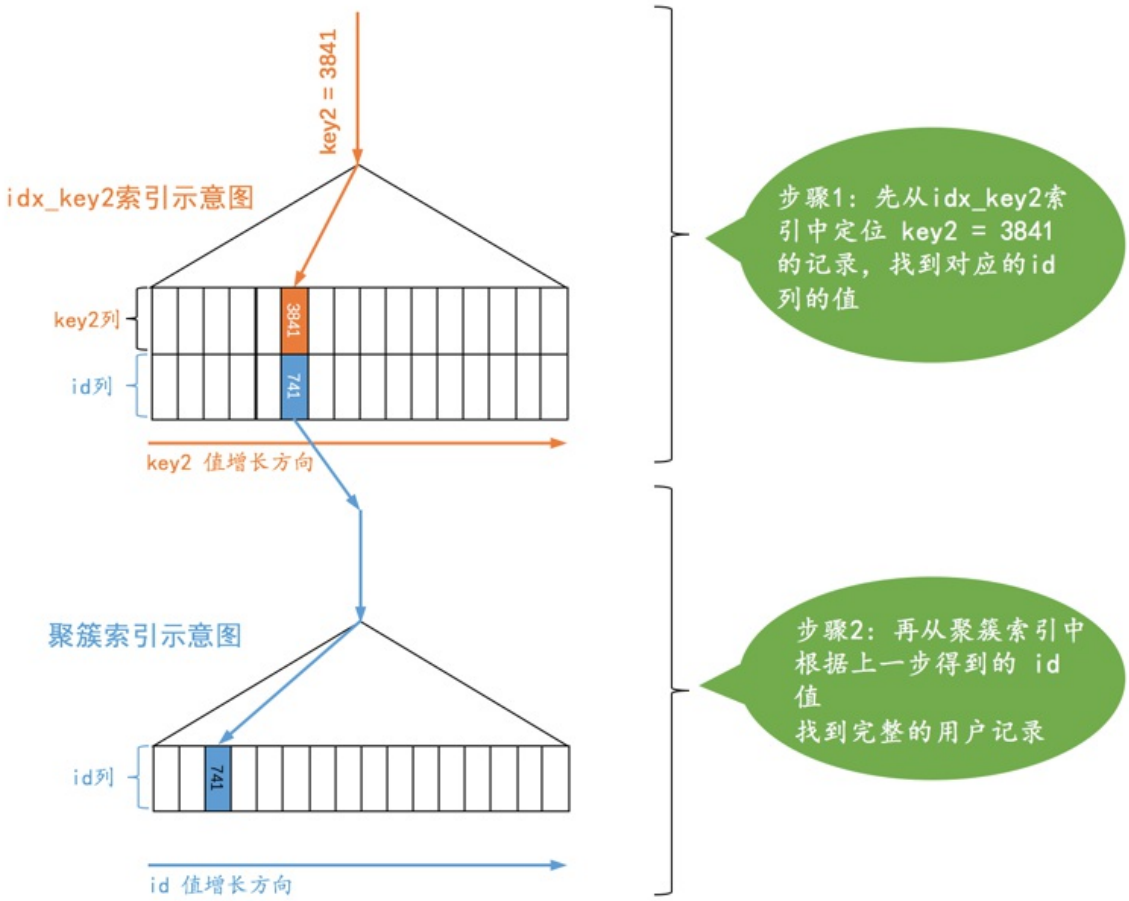
MySQL会直接利用主键值在聚簇索引中定位对应的用户记录，就像这样：



原谅我把聚簇索引对应的复杂的B+树结构搞了一个极度精简版，为了突出重点，我们忽略掉了页的结构，直接把所有的叶子节点的记录都放在一起展示，而且记录中只展示我们关心的索引列，对于single_table表的聚簇索引来说，展示的就是id列。我们想突出的重点就是：B+树叶子节点中的记录是按照索引列排序的，对于的聚簇索引来说，它对应的B+树叶子节点中的记录就是按照id列排序的。B+树本来就是一个矮矮的大胖子，所以这样根据主键值定位一条记录的速度贼快。类似的，我们根据唯一二级索引列来定位一条记录的速度也是贼快的，比如下边这个查询：

```
SELECT * FROM single_table WHERE key2 = 3841;
```

这个查询的执行过程的示意图就是这样：



可以看到这个查询的执行分两步，第一步先从idx_key2对应的B+树索引中根据key2列与常数的等值比较条件定位到一条二级索引记录，然后再根据该记录的id值到聚簇索引中获取到完整的用户记录。

设计MySQL的大叔认为通过主键或者唯一二级索引列与常数的等值比较来定位一条记录是像坐火箭一样快的，所以他们把这种通过主键或者唯一二级索引列来定位一条记录的访问方法定义为：const，意思是常数级别的，代价是可以忽略不计的。不过这种const访问方法只能在主键列或者唯一二级索引列和一个常数进行等值比较时才有效，如果主键或者唯一二级索引是由多个列构成的话，索引中的每一个列都需要与常数进行等值比较，这个const访问方法才有效（这是因为只有该索引中全部列都采用等值比较才可以定位唯一的一条记录）。

对于唯一二级索引来说，查询该列为NULL值的情况比较特殊，比如这样：

```
SELECT * FROM single_table WHERE key2 IS NULL;
```

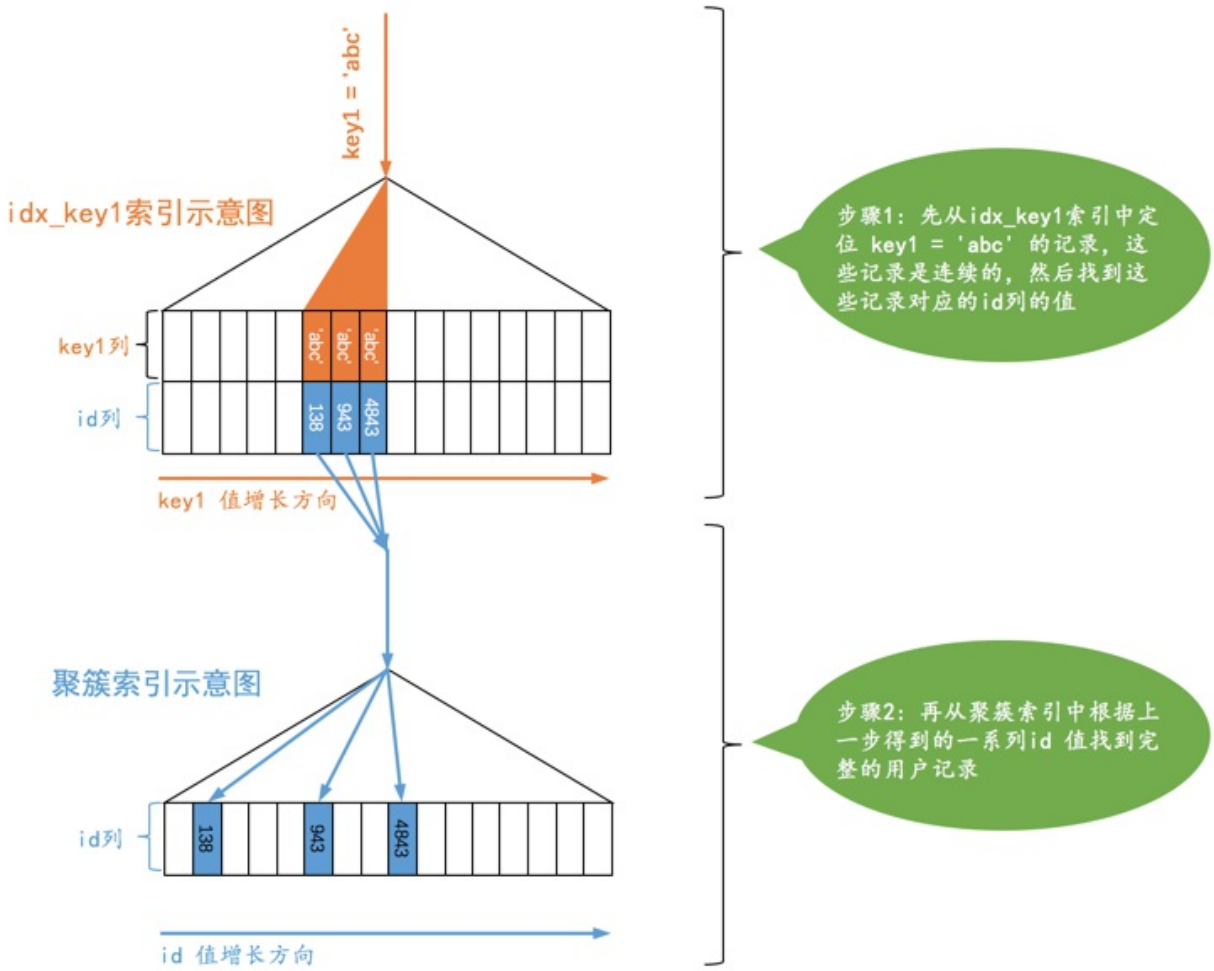
因为唯一二级索引列并不限制NULL值的数量，所以上述语句可能访问到多条记录，也就是说上边这个语句不可以使用const访问方法来执行。

ref

有时候我们对某个普通的二级索引列与常数进行等值比较，比如这样：

```
SELECT * FROM single_table WHERE key1 = 'abc';
```

对于这个查询，我们当然可以选择全表扫描来逐一对比搜索条件是否满足要求，我们也可以先使用二级索引找到对应记录的id值，然后再回表到聚簇索引中查找完整的用户记录。由于普通二级索引并不限制索引列值的唯一性，所以可能找到多条对应的记录，也就是说使用二级索引来执行查询的代价取决于等值匹配到的二级索引记录条数。如果匹配的记录较少，则回表的代价还是比较低的，所以MySQL可能选择使用索引而不是全表扫描的方式来执行查询。设计MySQL的大叔就把这种搜索条件为二级索引列与常数等值比较，采用二级索引来执行查询的访问方法称为：ref。我们看一下采用ref访问方法执行查询的图示：



从图示中可以看出，对于普通的二级索引来说，通过索引列进行等值比较后可能匹配到多条连续的记录，而不是像主键或者唯一二级索引那样最多只能匹配1条记录，所以这种ref访问方法比const差了那么一丢丢，但是在二级索引等值比较时匹配的记录数较少时的效率还是很高的（如果匹配的二级索引记录太多那么回表的成本就太大了），跟坐高铁差不多。不过需要注意下边两种情况：

- 二级索引列值为NULL的情况

  不论是普通的二级索引，还是唯一二级索引，它们的索引列对包含NULL值的数量并不限制，所以我们采用key IS NULL这种形式的搜索条件最多只能使用ref的访问方法，而不是const的访问方法。

- 对于某个包含多个索引列的二级索引来说，只要是最左边的连续索引列是与常数的等值比较就可能采用ref的访问方法，比方说下边这几个查询：

```
SELECT * FROM single_table WHERE key_part1 = 'god like';

SELECT * FROM single_table WHERE key_part1 = 'god like' AND key_part2 = 'legendary';

SELECT * FROM single_table WHERE key_part1 = 'god like' AND key_part2 = 'legendary' AND key_part3 = 'penta kill';
```

但是如果最左边的连续索引列并不全部是等值比较的话，它的访问方法就不能称为ref了，比方说这样：

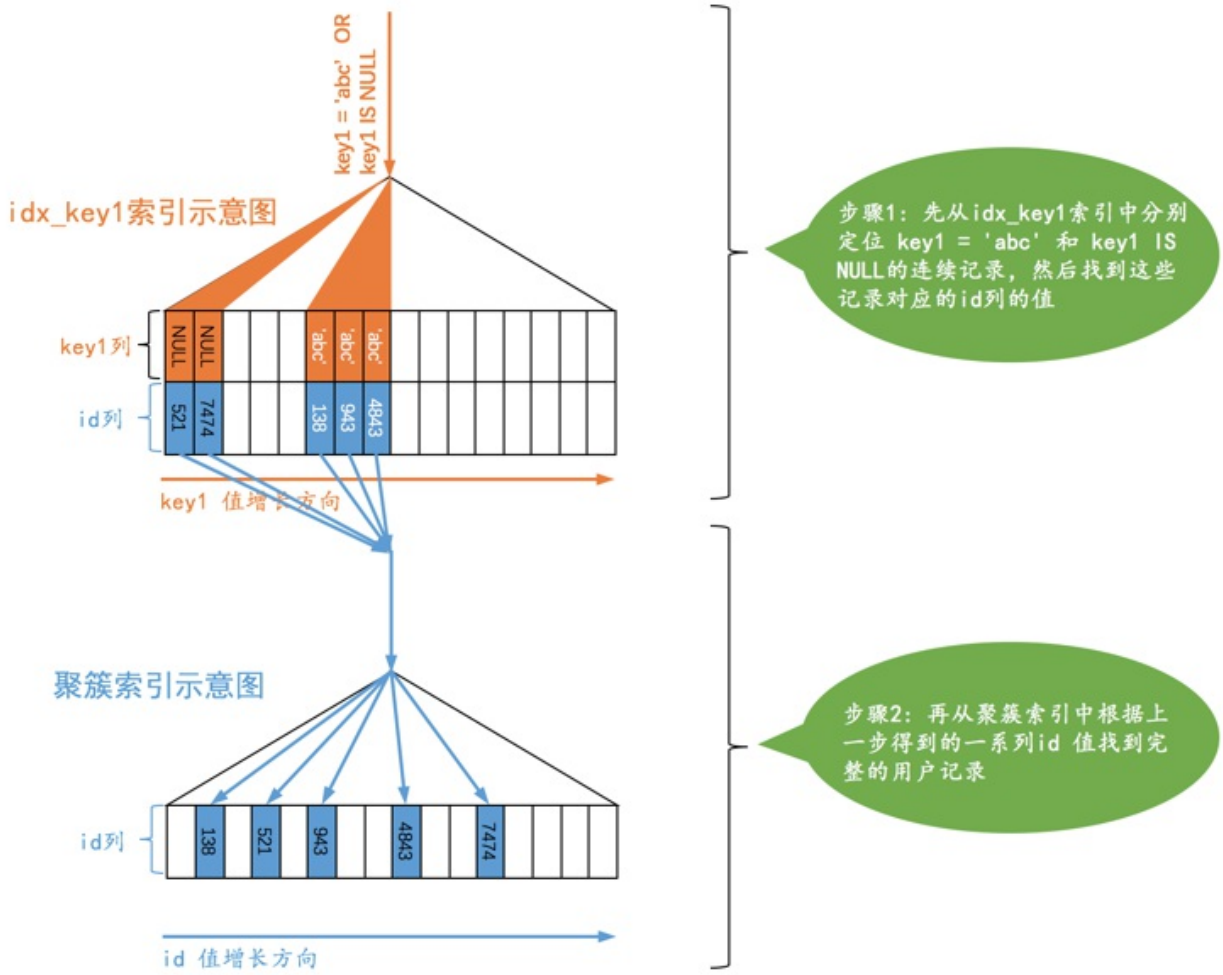
```
SELECT * FROM single_table WHERE key_part1 = 'god like' AND key_part2 > 'legendary';
```

ref_or_null

有时候我们不仅想找出某个二级索引列的值等于某个常数的记录，还想把该列的值为NULL的记录也找出来，就像下边这个查询：

```
SELECT * FROM single_demo WHERE key1 = 'abc' OR key1 IS NULL;
```

当使用二级索引而不是全表扫描的方式执行该查询时，这种类型的查询使用的访问方法就称为ref_or_null，这个ref_or_null访问方法的执行过程如下：



可以看到，上边的查询相当于先分别从idx_key1索引对应的B+树中找出key1 IS NULL和key1 = 'abc'的两个连续的记录范围，然后根据这些二级索引记录中的id值再回表查找完整的用户记录。

range

我们之前介绍的几种访问方法都是在对索引列与某一个常数进行等值比较的时候才可能使用到（ref_or_null比较奇特，还计算了值为NULL的情况），但是有时候我们面对的搜索条件更复杂，比如下边这个查询：

```
SELECT * FROM single_table WHERE key2 IN (1438, 6328) OR (key2 >= 38 AND key2 <= 79);
```

我们当然还可以使用全表扫描的方式来执行这个查询，不过也可以使用二级索引 + 回表的方式执行，如果采用二级索引 + 回表的方式来执行的话，那么此时的搜索条件就不只是要求索引列与常数的等值匹配了，而是索引列需要匹配某个或某些范围的值，在本查询中key2列的值只要匹配下列3个范围中的任何一个就算是匹配成功了：

- key2的值是1438
- key2的值是6328
- key2的值在38和79之间。

设计MySQL的大叔把这种利用索引进行范围匹配的访问方法称之为：range。

小贴士： 此处所说的使用索引进行范围匹配中的 `索引` 可以是聚簇索引，也可以是二级索引。

如果把这几个所谓的key2列的值需要满足的范围在数轴上体现出来的话，那应该是这个样子：

也就是从数学的角度看，每一个所谓的范围都是数轴上的一个区间，3个范围也就对应着3个区间：

- 范围1：key2 = 1438
- 范围2：key2 = 6328

- 范围3：key2 ∈ [38，79]，注意这里是闭区间。

我们可以把那种索引列等值匹配的情况称之为单点区间，上边所说的范围1和范围2都可以被称为单点区间，像范围3这种的我们可以称为连续范围区间。

index

看下边这个查询：

```
SELECT key_part1, key_part2, key_part3 FROM
single_table WHERE key_part2 = 'abc';
```

由于key_part2并不是联合索引idx_key_part最左索引列，所以我们无法使用ref或者range访问方法来执行这个语句。但是这个查询符合下边这两个条件：

- 它的查询列表只有3个列：key_part1, key_part2, key_part3，而索引idx_key_part又包含这三个列。
- 搜索条件中只有key_part2列。这个列也包含在索引idx_key_part中。

也就是说我们可以直接通过遍历idx_key_part索引的叶子节点的记录来比较key_part2 = 'abc'这个条件是否成立，把匹配成功的二级索引记录的key_part1, key_part2, key_part3列的值直接加到结果集中就行了。由于二级索引记录比聚簇索记录小的多（聚簇索引记录要存储所有用户定义的列以及所谓的隐藏列，而二级索引记录只需要存放索引列和主键），而且这个过程也不用进行回表操作，所以直接遍历二级索引比直接遍历聚簇索引的成本要小很多，设计MySQL的大叔就把这种采用遍历二级索引记录的执行方式称之为：index。

all

最直接的查询执行方式就是我们已经提了无数遍的全表扫描，对于InnoDB表来说也就是直接扫描聚簇索引，设计MySQL的大叔把这种使用全表扫描执行查询的方式称之为：`all`。

注意事项

重温 二级索引 + 回表

一般情况下只能利用单个二级索引执行查询，比方说下边的这个查询：

```
SELECT * FROM single_table WHERE key1 = 'abc' AND key2 > 1000;
```

查询优化器会识别到这个查询中的两个搜索条件：

- `key1 = 'abc'`
- `key2 > 1000`

优化器一般会根据`single_table`表的统计数据来判断到底使用哪个条件到对应的二级索引中查询扫描的行数会更少，选择那个扫描行数较少的条件到对应的二级索引中查询（关于如何比较的细节我们后边的章节中会唠叨）。然后将从该二级索引中查询到的结果经过回表得到完整的用户记录后再根据其余的`WHERE`条件过滤记录。一般来说，等值查找比范围查找需要扫描的行数更少（也就是`ref`的访问方法一般比`range`好，但这也不总是一定的，也可能采用`ref`访问方法的那个索引列的值为特定值的行数特别多），所以这里假设优化器决定使用`idx_key1`索引进行查询，那么整个查询过程可以分为两个步骤：

- 步骤1：使用二级索引定位记录的阶段，也就是根据条件key1 = 'abc'从idx_key1索引代表的B+树中找到对应的二级索引记录。

- 步骤2：回表阶段，也就是根据上一步骤中找到的记录的主键值进行回表操作，也就是到聚簇索引中找到对应的完整的用户记录，再根据条件key2 > 1000到完整的用户记录继续过滤。将最终符合过滤条件的记录返回给用户。

这里需要特别提醒大家的一点是，因为二级索引的节点中的记录只包含索引列和主键，所以在步骤1中使用idx_key1索引进行查询时只会用到与key1列有关的搜索条件，其余条件，比如key2 > 1000这个条件在步骤1中是用不到的，只有在步骤2完成回表操作后才能继续针对完整的用户记录中继续过滤。

小贴士： 需要注意的是，我们说一般情况下执行一个查询只会用到二级索引，不过还是有特殊情况的，我们后边会详细唠叨的。

明确range访问方法使用的范围区间

其实对于B+树索引来说，只要索引列和常数使用=、<=>、IN、NOT IN、IS NULL、IS NOT NULL、>、<、>=、<=、BETWEEN、!=（不等于也可以写成<>）或者LIKE操作符连接起来，就可以产生一个所谓的区间。

小贴士：

LIKE操作符比较特殊，只有在匹配完整字符串或者匹配字符串前缀时才可以利用索引，具体原因我们在前边的章节中唠叨过了，这里就不赘述了。

IN操作符的效果和若干个等值匹配操作符`=`之间用`OR`连接起来是一样的，也就是说会产生多个单点区间，比如下边这两个语句的效果是一样的：

```
SELECT * FROM single_table WHERE key2 IN (1438, 6328);

SELECT * FROM single_table WHERE key2 = 1438 OR key2 = 6328;
```

不过在日常的工作中，一个查询的WHERE子句可能有很多个小的搜索条件，这些搜索条件需要使用AND或者OR操作符连接起来，虽然大家都知道这两个操作符的作用，但我还是要再说一遍：

- cond1 AND cond2：只有当cond1和cond2都为TRUE时整个表达式才为TRUE。
- cond1 OR cond2：只要cond1或者cond2中有一个为TRUE整个表达式就为TRUE。

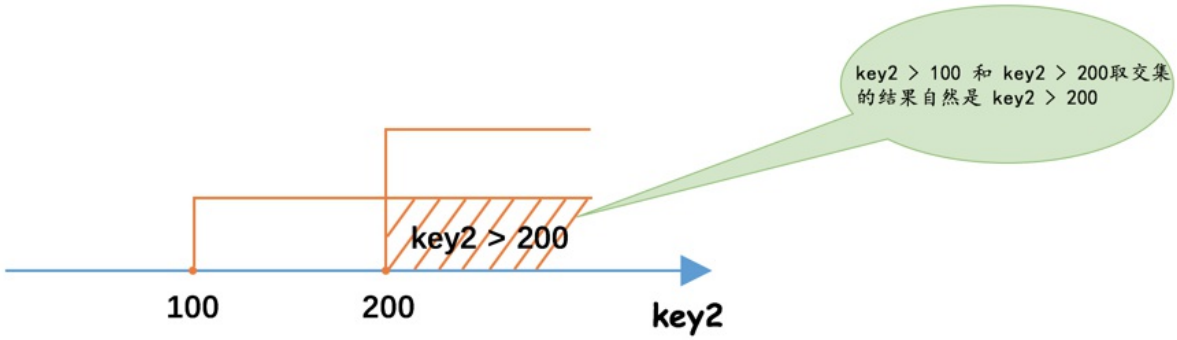
当我们想使用range访问方法来执行一个查询语句时，重点就是找出该查询可用的索引以及这些索引对应的范围区间。下边分两种情况看一下怎么从由AND或OR组成的复杂搜索条件中提取出正确的范围区间。

所有搜索条件都可以使用某个索引的情况

有时候每个搜索条件都可以使用到某个索引，比如下边这个查询语句：

```
SELECT * FROM single_table WHERE key2 > 100 AND key2 > 200;
```

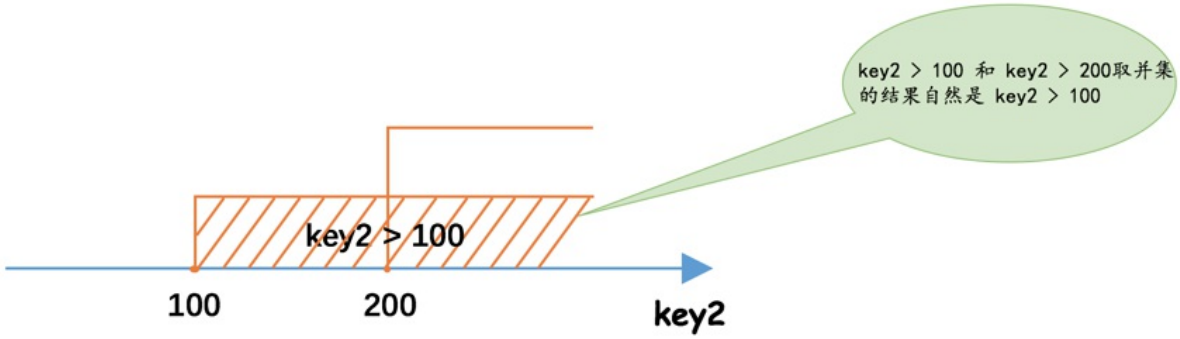
这个查询中的搜索条件都可以使用到key2，也就是说每个搜索条件都对应着一个idx_key2的范围区间。这两个小的搜索条件使用AND连接起来，也就是要取两个范围区间的交集，在我们使用range访问方法执行查询时，使用的idx_key2索引的范围区间的确定过程就如下图所示：



key2 > 100和key2 > 200交集当然就是key2 > 200了，也就是说上边这个查询使用idx_key2的范围区间就是(200, +∞)。这东西小学都学过吧，再不济初中肯定都学过。我们再看一下使用OR将多个搜索条件连接在一起的情况：

```
SELECT * FROM single_table WHERE key2 > 100 OR key2 > 200;
```

OR意味着需要取各个范围区间的并集，所以上边这个查询在我们使用range访问方法执行查询时，使用的idx_key2索引的范围区间的确定过程就如下图所示：



也就是说上边这个查询使用idx_key2的范围区间就是(100, +∞)。

有的搜索条件无法使用索引的情况

比如下边这个查询：

```
SELECT * FROM single_table WHERE key2 > 100 AND common_field = 'abc';
```

请注意，这个查询语句中能利用的索引只有idx_key2一个，而idx_key2这个二级索引的记录中又不包含common_field这个字段，所以在使用二级索引idx_key2定位定位记录的阶段用不到common_field = 'abc'这个条件，这个条件是在回表获取了完整的用户记录后才使用的，而范围区间是为了到索引中取记录中提出的概念，所以在确定范围区间的时候不需要考虑common_field = 'abc'这个条件，我们在为某个索引确定范围区间的时候只需要把用不到相关索引的搜索条件替换为TRUE就好了。

小贴士：

之所以把用不到索引的搜索条件替换为TRUE，是因为我们不打算使用这些条件进行在该索引上进行过滤，所以不管索引的记录满不满足这些条件，我们都把它们选取出来，待到之后回表的时候再使用它们过滤。

我们把上边的查询中用不到idx_key2的搜索条件替换后就是这样：

```
SELECT * FROM single_table WHERE key2 > 100 AND
TRUE;
```

化简之后就是这样：

```
SELECT * FROM single_table WHERE key2 > 100;
```

也就是说最上边那个查询使用idx_key2的范围区间就是：(100, +∞)。

再来看一下使用OR的情况：

```
SELECT * FROM single_table WHERE key2 > 100 OR
common_field = 'abc';
```

同理，我们把使用不到idx_key2索引的搜索条件替换为TRUE：

```
SELECT * FROM single_table WHERE key2 > 100 OR
TRUE;
```

接着化简：

```
SELECT * FROM single_table WHERE TRUE;
```

额，这也就说说明如果我们强制使用idx_key2执行查询的话，对应的范围区间就是(-∞，+∞)，也就是需要将全部二级索引的记录进行回表，这个代价肯定比直接全表扫描都大了。也就是说一个使用到索引的搜索条件和没有使用该索引的搜索条件使用OR连接起来后是无法使用该索引的。

复杂搜索条件下找出范围匹配的区间

有的查询的搜索条件可能特别复杂，光是找出范围匹配的各个区间就挺烦的，比方说下边这个：

```
SELECT * FROM single_table WHERE
        (key1 > 'xyz' AND key2 = 748 ) OR
        (key1 < 'abc' AND key1 > 'lmn') OR
        (key1 LIKE '%suf' AND key1 > 'zzz' AND
(key2 < 8000 OR common_field = 'abc')) ;
```

我滴个神，这个搜索条件真是绝了，不过大家不要被复杂的表象迷住了双眼，按着下边这个套路分析一下：

- 首先查看WHERE子句中的搜索条件都涉及到了哪些列，哪些列可能使用到索引。

  这个查询的搜索条件涉及到了key1、key2、common_field这3个列，然后key1列有普通的二级索引idx_key1，key2列有唯一二级索引idx_key2。

- 对于那些可能用到的索引，分析它们的范围区间。

  - 假设我们使用idx_key1执行查询

    - 我们需要把那些用不到该索引的搜索条件暂时移除掉，移除方法也简单，直接把它们替换为TRUE就好了。上边的查询中除了有关key2和common_field列不能使用到idx_key1索引外，key1 LIKE '%suf'也使用不到索引，所以把这些搜索条件替换为TRUE之后的样子就是这样：

      ```
      (key1 > 'xyz' AND TRUE ) OR
      (key1 < 'abc' AND key1 > 'lmn') OR
      (TRUE AND key1 > 'zzz' AND (TRUE OR
      TRUE))
      ```

化简一下上边的搜索条件就是下边这样：

```
(key1 > 'xyz') OR
(key1 < 'abc' AND key1 > 'lmn') OR
(key1 > 'zzz')
```

- 替换掉永远为TRUE或FALSE的条件

  因为符合key1 < 'abc' AND key1 > 'lmn'永远为FALSE，所以上边的搜索条件可以被写成这样：

  ```
  (key1 > 'xyz') OR (key1 > 'zzz')
  ```

- 继续化简区间

  key1 > 'xyz'和key1 > 'zzz'之间使用OR操作符连接起来的，意味着要取并集，所以最终的结果化简的到的区间就是：key1 > xyz。也就是说：上边那个有一坨搜索条件的查询语句如果使用idx_key1 索引执行查询的话，需要把满足key1 > xyz的二级索引记录都取出来，然后拿着这些记录的id再进行回表，得到完整的用户记录之后再使用其他的搜索条件进行过滤。

- 假设我们使用idx_key2执行查询

  - 我们需要把那些用不到该索引的搜索条件暂时使用TRUE条件替换掉，其中有关key1和common_field的搜索条件都需要被替换掉，替换结果就是：

```
(TRUE AND key2 = 748 ) OR
(TRUE AND TRUE) OR
(TRUE AND TRUE AND (key2 < 8000 OR
TRUE))
```

哎呀呀，key2 < 8000 OR TRUE的结果肯定是TRUE呀，也就是说化简之后的搜索条件成这样了：

```
key2 = 748 OR TRUE
```

这个化简之后的结果就更简单了：

```
TRUE
```

这个结果也就意味着如果我们要使用idx_key2索引执行查询语句的话，需要扫描idx_key2二级索引的所有记录，然后再回表，这不是得不偿失么，所以这种情况下不会使用idx_key2索引的。

索引合并

我们前边说过MySQL在一般情况下执行一个查询时最多只会用到单个二级索引，但不是还有特殊情况么，在这些特殊情况下也可能在一个查询中使用到多个二级索引，设计MySQL的大叔把这种使用到多个索引来完成一次查询的执行方法称之为：index merge，具体的索引合并算法有下边三种。

Intersection合并

Intersection翻译过来的意思是交集。这里是说某个查询可以使用多个二级索引，将从多个二级索引中查询到的结果取交集，比方说下边这个查询：

```
SELECT * FROM single_table WHERE key1 = 'a' AND key3 = 'b';
```

假设这个查询使用Intersection合并的方式执行的话，那这个过程就是这样的：

- 从idx_key1二级索引对应的B+树中取出key1 = 'a'的相关记录。
- 从idx_key3二级索引对应的B+树中取出key3 = 'b'的相关记录。
- 二级索引的记录都是由索引列 + 主键构成的，所以我们可以计算出这两个结果集中id值的交集。
- 按照上一步生成的id值列表进行回表操作，也就是从聚簇索引中把指定id值的完整用户记录取出来，返回给用户。

这里有同学会思考：为啥不直接使用idx_key1或者idx_key2只根据某个搜索条件去读取一个二级索引，然后回表后再过滤另外一个搜索条件呢？这里要分析一下两种查询执行方式之间需要的成本代价。

只读取一个二级索引的成本：

- 按照某个搜索条件读取一个二级索引
- 根据从该二级索引得到的主键值进行回表操作，然后再过滤其他的搜索条件

读取多个二级索引之后取交集成本：

- 按照不同的搜索条件分别读取不同的二级索引
- 将从多个二级索引得到的主键值取交集，然后进行回表操作

虽然读取多个二级索引比读取一个二级索引消耗性能，但是读取二级索引的操作是顺序I/O，而回表操作是随机I/O，所以如果只读取一个二级索引时需要回表的记录数特别多，而读取多个二级索引之后取交集的记录数非常少，当节省的因为回表而造成的性能损耗比访问多个二级索引带来的性能损耗更高时，读取多个二级索引后取交集比只读取一个二级索引的成本更低。

MySQL在某些特定的情况下才可能会使用到Intersection索引合并：

- 情况一：二级索引列是等值匹配的情况，对于联合索引来说，在联合索引中的每个列都必须等值匹配，不能出现只出现匹配部分列的情况。

  比方说下边这个查询可能用到idx_key1和idx_key_part这两个二级索引进行Intersection索引合并的操作：

  ```
  SELECT * FROM single_table WHERE key1 = 'a'
  AND key_part1 = 'a' AND key_part2 = 'b' AND
  key_part3 = 'c';
  ```

  而下边这两个查询就不能进行Intersection索引合并：

  ```
  SELECT * FROM single_table WHERE key1 > 'a'
  AND key_part1 = 'a' AND key_part2 = 'b' AND
  key_part3 = 'c';

  SELECT * FROM single_table WHERE key1 = 'a'
  AND key_part1 = 'a';
  ```

  第一个查询是因为对key1进行了范围匹配，第二个查询是因为联合索引idx_key_part中的key_part2列并没有出现在搜索条件中，所以这两个查询不能进行Intersection索引合并。

- 情况二：主键列可以是范围匹配

  比方说下边这个查询可能用到主键和idx_key_part进行Intersection索引合并的操作：

  ```
  SELECT * FROM single_table WHERE id > 100 AND key1 = 'a';
  ```

为啥呢？凭啥呀？突然冒出这么两个规定让大家一脸懵逼，下边我们慢慢品一品这里头的玄机。这话还得从InnoDB的索引结构说起，你要是记不清麻烦再回头看看。对于InnoDB的二级索引来说，记录先是按照索引列进行排序，如果该二级索引是一个联合索引，那么会按照联合索引中的各个列依次排序。而二级索引的用户记录是由索引列 + 主键构成的，二级索引列的值相同的记录可能会有好多条，这些索引列的值相同的记录又是按照主键的值进行排序的。所以重点来了，之所以在二级索引列都是等值匹配的情况下才可能使用Intersection索引合并，是因为只有在这种情况下根据二级索引查询出的结果集是按照主键值排序的。

so？还是没看懂根据二级索引查询出的结果集是按照主键值排序的对使用Intersection索引合并有啥好处？小伙子，别忘了Intersection索引合并会把从多个二级索引中查询出的主键值求交集，如果从各个二级索引中查询的到的结果集本身就是已经按照主键排好序的，那么求交集的过程就很easy啦。假设某个查询使用Intersection索引合并的方式从idx_key1和idx_key2这两个二级索引中获取到的主键值分别是：

- 从idx_key1中获取到已经排好序的主键值：1、3、5
- 从idx_key2中获取到已经排好序的主键值：2、3、4

那么求交集的过程就是这样：逐个取出这两个结果集中最小的主键值，如果两个值相等，则加入最后的交集结果中，否则丢弃当前较小的主键值，再取该丢弃的主键值所在结果集的后一个主键值来比较，

直到某个结果集中的主键值用完了，如果还是觉得不太明白那继续往下看：

- 先取出这两个结果集中较小的主键值做比较，因为1 < 2，所以把idx_key1的结果集的主键值1丢弃，取出后边的3来比较。
- 因为3 > 2，所以把idx_key2的结果集的主键值2丢弃，取出后边的3来比较。
- 因为3 = 3，所以把3加入到最后的交集结果中，继续两个结果集后边的主键值来比较。
- 后边的主键值也不相等，所以最后的交集结果中只包含主键值3。

别看我们写的啰嗦，这个过程其实可快了，时间复杂度是O(n)，但是如果从各个二级索引中查询出的结果集并不是按照主键排序的话，那就要先把结果集中的主键值排序完再来做上边的那个过程，就比较耗时了。

> 小贴士：
>
> 按照有序的主键值去回表取记录有个专有名词儿，叫：Rowid Ordered Retrieval，简称ROR，**以后大家在某些地方见到这个名词儿就眼熟了。**

另外，不仅是多个二级索引之间可以采用Intersection索引合并，索引合并也可以有聚簇索引参加，也就是我们上边写的情况二：在搜索条件中有主键的范围匹配的情况下也可以使用Intersection索引合并索引合并。为啥主键这就可以范围匹配了？还是得回到应用场景里，比如看下边这个查询：

```
SELECT * FROM single_table WHERE key1 = 'a' AND id > 100;
```

假设这个查询可以采用Intersection索引合并，我们理所当然的以为这个查询会分别按照id > 100这个条件从聚簇索引中获取一些记录，在通过key1 = 'a'这个条件从idx_key1二级索引中获取一些记录，然后再求交集，其实这样就把问题复杂化了，没必要从聚簇索引中获取一次记录。别忘了二级索引的记录中都带有主键值的，所以可以在从idx_key1中获取到的主键值上直接运用条件id > 100过滤就行了，这样多简单。所以涉及主键的搜索条件只不过是为了从别的二级索引得到的结果集中过滤记录罢了，是不是等值匹配不重要。

当然，上边说的情况一和情况二只是发生Intersection索引合并的必要条件，不是充分条件。也就是说即使情况一、情况二成立，也不一定发生Intersection索引合并，这得看优化器的心情。优化器在下边两个条件满足的情况下才趋向于使用Intersection索引合并：

- 单独根据搜索条件从某个二级索引中获取的记录数太多，导致回表开销太大
- 通过Intersection索引合并后需要回表的记录数大大减少

Union合并

我们在写查询语句时经常想把既符合某个搜索条件的记录取出来，也把符合另外的某个搜索条件的记录取出来，我们说这些不同的搜索条件之间是OR关系。有时候OR关系的不同搜索条件会使用到同一个索引，比方说这样：

```
SELECT * FROM single_table WHERE key1 = 'a' OR key3 = 'b'
```

Intersection是交集的意思，这适用于使用不同索引的搜索条件之间使用AND连接起来的情况；Union是并集的意思，适用于使用不同索引的搜索条件之间使用OR连接起来的情况。与Intersection

索引合并类似，MySQL在某些特定的情况下才可能会使用到Union索引合并：

- 情况一：二级索引列是等值匹配的情况，对于联合索引来说，在联合索引中的每个列都必须等值匹配，不能出现只出现匹配部分列的情况。

  比方说下边这个查询可能用到idx_key1和idx_key_part这两个二级索引进行Union索引合并的操作：

  ```
  SELECT * FROM single_table WHERE key1 = 'a'
  OR ( key_part1 = 'a' AND key_part2 = 'b' AND
  key_part3 = 'c');
  ```

  而下边这两个查询就不能进行Union索引合并：

  ```
  SELECT * FROM single_table WHERE key1 > 'a'
  OR (key_part1 = 'a' AND key_part2 = 'b' AND
  key_part3 = 'c');

  SELECT * FROM single_table WHERE key1 = 'a'
  OR key_part1 = 'a';
  ```

  第一个查询是因为对key1进行了范围匹配，第二个查询是因为联合索引idx_key_part中的key_part2列并没有出现在搜索条件中，所以这两个查询不能进行Union索引合并。

- 情况二：主键列可以是范围匹配

- 情况三：使用Intersection索引合并的搜索条件

  这种情况其实也挺好理解，就是搜索条件的某些部分使用Intersection索引合并的方式得到的主键集合和其他方式得到的主键集合取交集，比方说这个查询：

```
SELECT * FROM single_table WHERE key_part1 =
'a' AND key_part2 = 'b' AND key_part3 = 'c'
OR (key1 = 'a' AND key3 = 'b');
```

优化器可能采用这样的方式来执行这个查询：

- 先按照搜索条件key1 = 'a' AND key3 = 'b'从索引idx_key1和idx_key3中使用Intersection索引合并的方式得到一个主键集合。
- 再按照搜索条件key_part1 = 'a' AND key_part2 = 'b' AND key_part3 = 'c'从联合索引idx_key_part中得到另一个主键集合。
- 采用Union索引合并的方式把上述两个主键集合取并集，然后进行回表操作，将结果返回给用户。

当然，查询条件符合了这些情况也不一定就会采用Union索引合并，也得看优化器的心情。优化器在下边两个条件满足的情况下才趋向于使用Union索引合并：

- 单独根据搜索条件从某个二级索引中获取的记录数比较少
- 通过Intersection索引合并后需要回表的记录数大大减少

Sort-Union合并

Union索引合并的使用条件太苛刻，必须保证各个二级索引列在进行等值匹配的条件下才可能被用到，比方说下边这个查询就无法使用到Union索引合并：

```
SELECT * FROM single_table WHERE key1 < 'a' OR
key3 > 'z'
```

这是因为根据key1 < 'a'从idx_key1索引中获取的二级索引记录的主键值不是排好序的，根据key3 > 'z'从idx_key3索引中获取的二级索引记录的主键值也不是排好序的，但是key1 < 'a'和key3 > 'z'这两个条件又特别让我们动心，所以我们可以这样：

- 先根据key1 < 'a'条件从idx_key1二级索引总获取记录，并按照记录的主键值进行排序
- 再根据key3 > 'z'条件从idx_key3二级索引总获取记录，并按照记录的主键值进行排序
- 因为上述的两个二级索引主键值都是排好序的，剩下的操作和Union索引合并方式就一样了。

我们把上述这种先按照二级索引记录的主键值进行排序，之后按照Union索引合并方式执行的方式称之为Sort-Union索引合并，很显然，这种Sort-Union索引合并比单纯的Union索引合并多了一步对二级索引记录的主键值排序的过程。

小贴士：

为啥有Sort-Union索引合并，就没有Sort-Intersection索引合并么？是的，的确没有Sort-Intersection索引合并这么一说，

Sort-Union的适用场景是单独根据搜索条件从某个二级索引中获取的记录数比较少，这样即使对这些二级索引记录按照主键值进行排序的成本也不会太高

而Intersection索引合并的适用场景是单独根据搜索条件从某个二级索引中获取的记录数太多，导致回表开销太大，合并后可以明显降低回表开销，但是如果加入Sort-Intersection后，就需要为大量的二级索引记录按照主键值进行排序，这个成本可能比回表查询都高了，所以也就没有引入Sort-Intersection这个玩意儿。

索引合并注意事项

联合索引替代Intersection索引合并

```
SELECT * FROM single_table WHERE key1 = 'a' AND key3 = 'b';
```

这个查询之所以可能使用Intersection索引合并的方式执行，还不是因为idx_key1和idx_key2是两个单独的B+树索引，你要是把这两个列搞一个联合索引，那直接使用这个联合索引就把事情搞定了，何必用啥索引合并呢，就像这样：

```
ALTER TABLE single_table drop index idx_key1, idx_key3, add index idx_key1_key3(key1, key3);
```

这样我们把没用的idx_key1、idx_key3都干掉，再添加一个联合索引idx_key1_key3，使用这个联合索引进行查询简直是又快又好，既不用多读一棵B+树，也不用合并结果，何乐而不为？

小贴士：

不过小心有单独对key3列进行查询的业务场景，这样子不得不再把key3列的单独索引给加上。