面试

问：数据库中最常见的慢查询优化方式是什么？ 同学A：加索引。

问：为什么加索引能优化慢查询？同学A：...不知道同学B：因为索引其实就是一种优化查询的数据结构，比如Mysql中的索引是用B+树实现的，而B+树就是一种数据结构，可以优化查询速度，可以利用索引快速查找数据，所以能优化查询。

问：你知道哪些数据结构可以提高查询速度？（听到这个问题就感觉此处有坑...）同学B：哈希表、完全平衡二叉树、B树、B+树等等。

问：那这些数据结构既然都能优化查询速度，那Mysql种为何选择使用B+树？同学B：...不知道

提问

```
SHOW INDEX FROM employees.titles;
```

Table	Non_uni...	Key_name	Seq_in_ind...	Column_name	Collation	Cardinality	Sub_p...	Pack...	Null	Index_ty...	Comm...	Index_com...
titles	0	PRIMARY	1	emp_no	A	296714	NULL	NULL		BTREE		
titles	0	PRIMARY	2	title	A	442308	NULL	NULL		BTREE		
titles	0	PRIMARY	3	from_date	A	442308	NULL	NULL		BTREE		

有一个titles表，主键由empno，title，fromdate三个字段组成。

那么以下几个语句会用到索引吗?

1. select*fromemployees.titleswhereemp_no=1
2. select*fromemployees.titleswheretitle='1'
3. select*fromemployees.titleswhereemp_no='1'andtitle=1
4. select*fromemployees.titleswheretitle='1'andemp_no=1

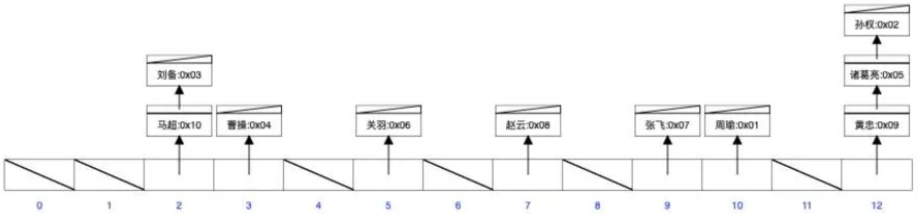
为什么哈希表、完全平衡二叉树、B树、B+树都可以优化查询，为何Mysql独独喜欢B+树？

哈希表有什么特点？

假如有这么一张表(表名：sanguo)：

id	name	role
▶ 1	周瑜	吴国大都督
2	孙权	吴国国王
3	刘备	蜀国国王
4	曹操	魏国国王
5	诸葛亮	蜀国军师
6	关羽	五虎上将一
7	张飞	五虎上将二
8	赵云	五虎上将三
9	黄忠	五虎上将四
10	马超	五虎上将五
NULL	NULL	NULL

现在对name字段建立哈希索引：



注意字段值所对应的数组下标是哈希算法随机算出来的，所以可能出现**哈希冲突**。那么对于这样一个索引结构，现在来执行下面的sql语句：

```
select*fromsanguowherename=’周瑜’
```

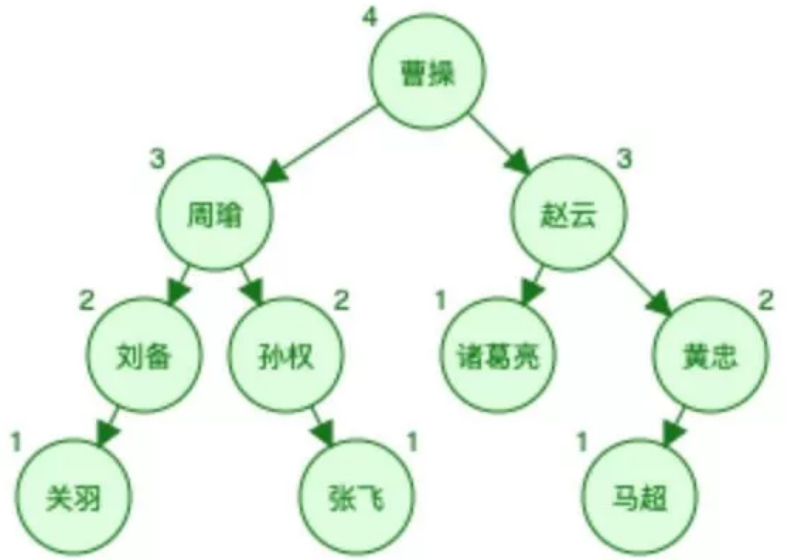
可以直接对‘周瑜’按哈希算法算出来一个数组下标，然后可以直接从数据中取出数据并拿到锁对应那一行数据的地址，进而查询那一行数据。 那么如果现在执行下面的sql语句：

```
select*fromsanguowherename>’周瑜’
```

则无能为力，因为哈希表的特点就是**可以快速的精确查询，但是不支持范围查询**。

如果用完全平衡二叉树呢？

还是上面的表数据用完全平衡二叉树表示如下图（为了简单，数据对应的地址就不画在图中了。）：



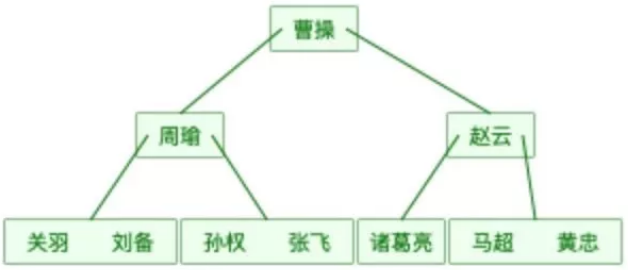
图中的每一个节点实际上应该有四部分：

1. 左指针，指向左子树
2. 键值
3. 键值所对应的数据的存储地址
4. 右指针，指向右子树

另外需要提醒的是，二叉树是有顺序的，简单的说就是“左边的小于右边的”假如我们现在来查找‘周瑜’，需要找2次（第一次曹操，第二次周瑜），比哈希表要多一次。而且由于完全平衡二叉树是有序的，所以也是支持范围查找的。

如果用B树呢？

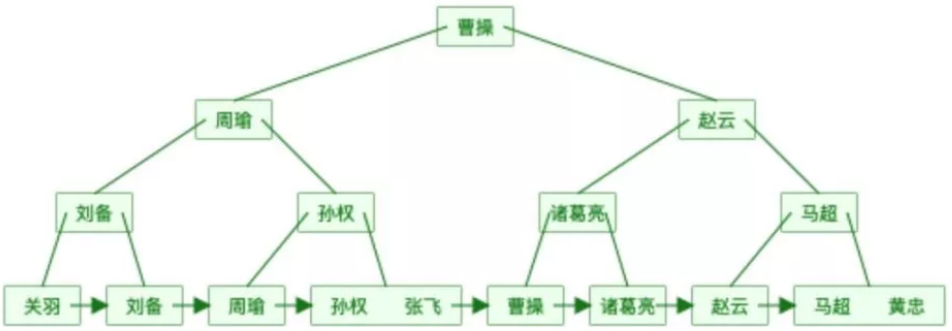
还是上面的表数据用B树表示如下图（为了简单，数据对应的地址就不画在图中了。）：



可以发现同样的元素，B树的表示要比完全平衡二叉树要“矮”，原因在于B树中的一个节点可以存储多个元素。

如果用B+树呢？

还是上面的表数据用B+树表示如下图（为了简单，数据对应的地址就不画在图中了。）：



我们可以发现同样的元素，B+树的表示要比B树要“胖”，原因在于B+树中的非叶子节点会冗余一份在叶子节点中，并且叶子节点之间用指针相连。

那么B+树到底有什么优势呢？

这里我们用“反证法”，假如我们现在就用完全平衡二叉树作为索引的数据结构，我们来看一下有什么不妥的地方。实际上，索引也是很“大”的，因为索引也是存储元素的，我们的一个表的数据行数越多，那么对应的索引文件其实也是会很大的，实际上**也是需要存储在磁盘中的，而不能全部都放在内存中**，所以我们在考虑选用哪种数据结构时，我们可以换一个角度思考，**哪个数据结构更适合从磁盘中读取数据**，或者**哪个数据结构能够提高磁盘的IO效率**。回头看一下完全平衡二叉树，当我们需要查询“张飞”时，需要以下步骤

1. 从磁盘中取出“曹操”到内存，CPU从内存取出数据进行笔记，“张飞”<“曹操”，取左子树（产生了一次磁盘IO）
2. 从磁盘中取出“周瑜”到内存，CPU从内存取出数据进行笔记，“张飞”>“周瑜”，取右子树（产生了一次磁盘IO）
3. 从磁盘中取出“孙权”到内存，CPU从内存取出数据进行笔记，“张飞”>“孙权”，取右子树（产生了一次磁盘IO）
4. 从磁盘中取出“黄忠”到内存，CPU从内存取出数据进行笔记，“张飞”=“张飞”，找到结果（产生了一次磁盘IO）

同理，回头看一下B树，我们发现只发送三次磁盘IO就可以找到“张飞”了，这就是B树的优点：**一个节点可以存储多个元素，相对于完全平衡二叉树所以整棵树的高度就降低了，磁盘IO效率提高了**。

而B+树是B树的升级版，只是把非叶子节点冗余一下，这么做的好处是**为了提高范围查找的效率**。

> 到这里可以总结出来，Mysql选用B+树这种数据结构作为索引，可以提高查询索引时的磁盘IO效率，并且可以提高范围查询的效率，并且B+树里的元素也是有序的。

那么，一个B+树的节点中到底存多少个元素合适呢？

其实也可以换个角度来思考**B+树中一个节点到底多大合适？**

答案是：**B+树中一个节点为一页或页的倍数最为合适**。因为如果一个节点的大小小于1页，那么读取这个节点的时候其实也会读出1页，造成资源的浪费；如果一个节点的大小大于1页，比如1.2页，那么读取这个节点的时候会读出2页，也会造成资源的浪费；所以为了不造成浪费，所以最后把一个节点的大小控制在1页、2页、3页、4页等倍数页大小最为合适。

那么，Mysql中B+树的一个节点大小为多大呢？

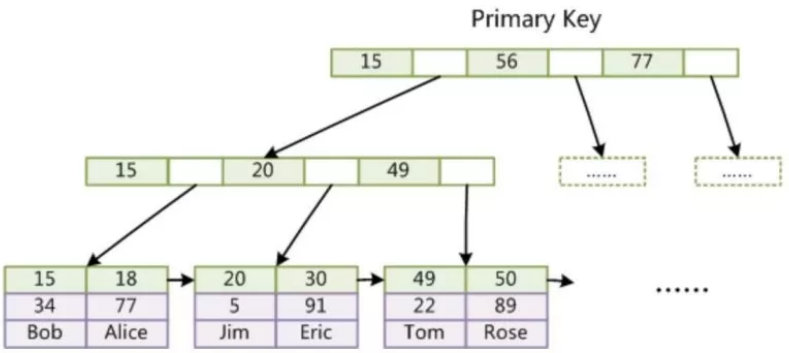
这个问题的答案是“1页”，这里说的“页”是Mysql自定义的单位（其实和操作系统类似），Mysql的Innodb引擎中一页的默认大小是16k（如果操作系统中一页大小是4k，那么Mysql中1页=操作系统中4页），可以使用命令SHOW GLOBAL STATUS like 'Innodbpagesize'; 查看。

并且还可以告诉你的是，一个节点为1页就够了。

为什么一个节点为1页（16k）就够了？

解决这个问题，我们先来看一下Mysql中利用B+树的具体实现。

Mysql中MyISAM和innodb使用B+树



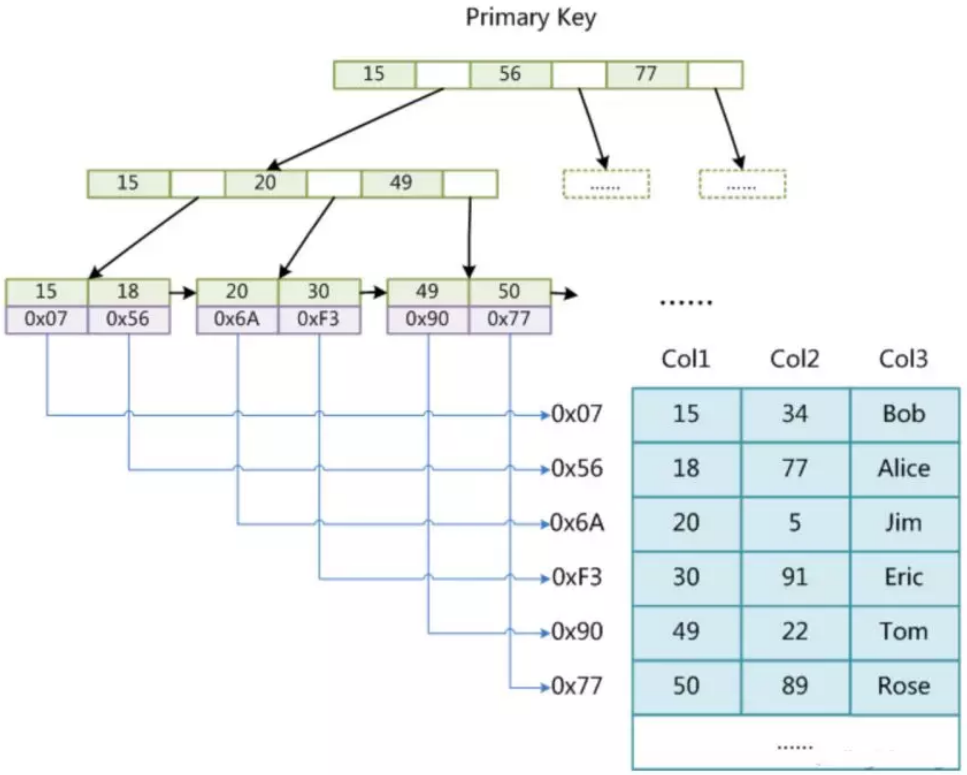
通常我们认为B+树的非叶子节点不存储数据，只有叶子节点才存储数据；而B树的非叶子和叶子节点都会存储数据，会导致非叶子节点存储的索引值会更少，树

的高度相对会比B+树高，平均的I/O效率会比较低，所以使用B+树作为索引的数据结构，再加上B+树的叶子节点之间会有指针相连，也方便进行范围查找。上图的data区域两个存储引擎会有不同。

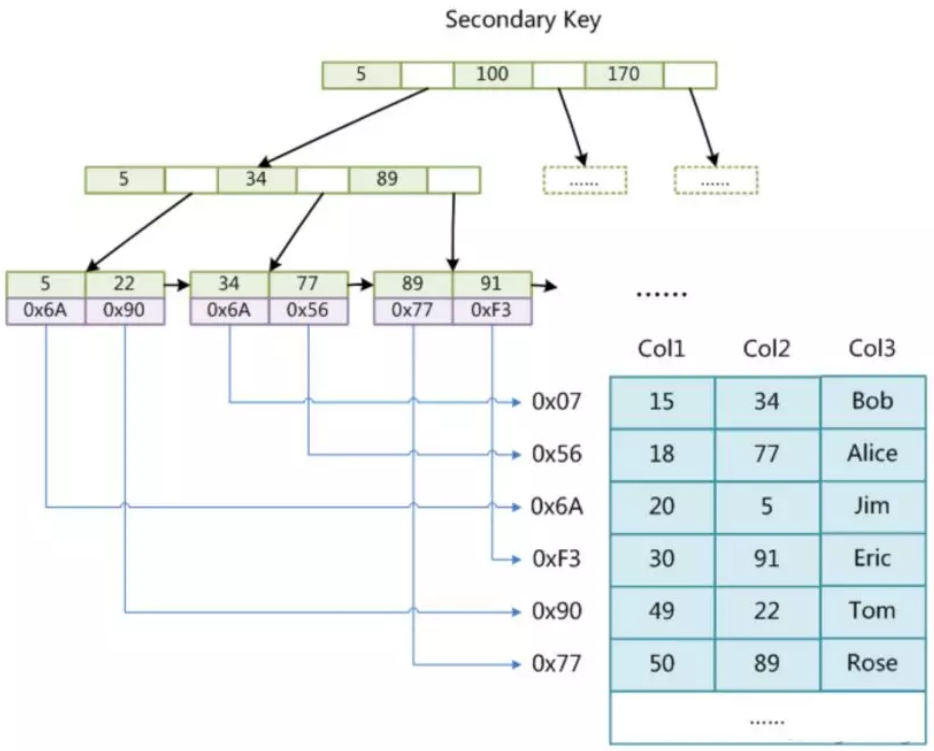
MyISAM中的B+树

MYISAM中叶子节点的数据区域存储的是数据记录的地址

主键索引



辅助索引

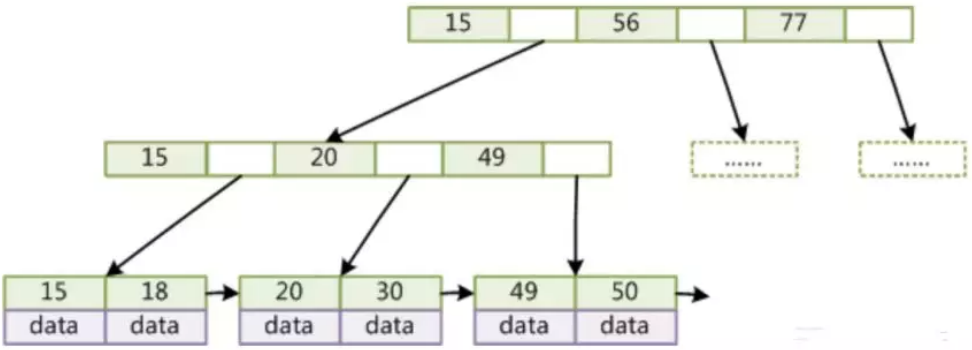


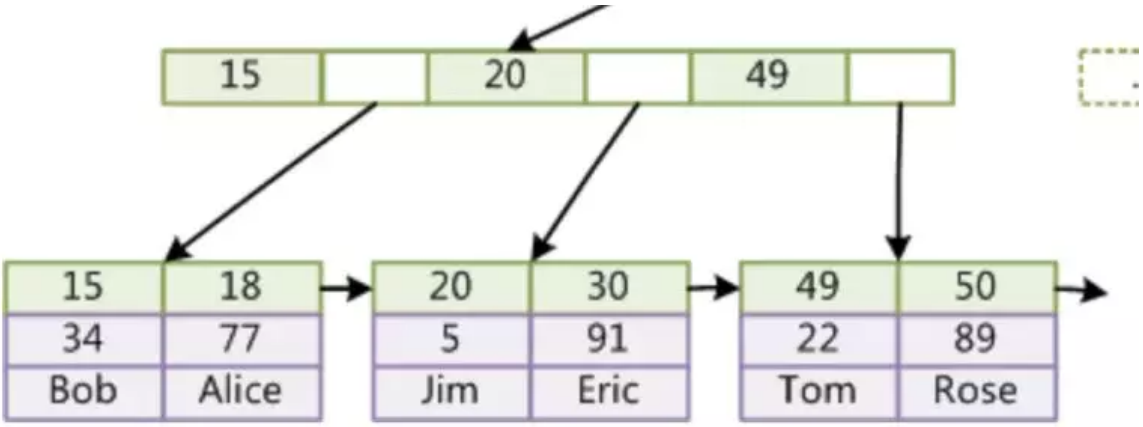
MyISAM存储引擎在使用索引查询数据时，会先根据索引查找到数据地址，再根据地址查询到具体的数据。并且主键索引和辅助索引没有太多区别。

InnoDB中的B+树

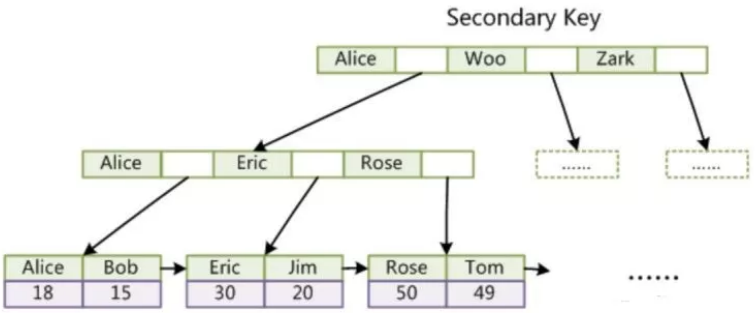
InnoDB中主键索引的叶子节点的数据区域存储的是数据记录，辅助索引存储的是主键值

主键索引





辅助索引



Innodb中的主键索引和实际数据时绑定在一起的，也就是说Innodb的一个表一定要有主键索引，如果一个表没有手动建立主键索引，Innodb会查看有没有唯一索引，如果有则选用唯一索引作为主键索引，如果连唯一索引也没有，则会默认建立一个隐藏的主键索引（用户不可见）。另外，Innodb的主键索引要比MyISAM的主键索引查询效率要高（少一次磁盘IO），并且比辅助索引也要高很多。所以，我们在使用Innodb作为存储引擎时，我们最好：

1. 手动建立主键索引
2. 尽量利用主键索引查询

回到我们的问题：为什么一个节点为1页（16k）就够了？

对着上面Mysql中Innodb中对B+树的实际应用（主要看主键索引），可以发现B+树中的一个节点存储的内容是：

- 非叶子节点：主键+指针
- 叶子节点：数据

那么，假设我们一行数据大小为1K，那么一页就能存16条数据，也就是一个叶子节点能存16条数据；再看非叶子节点，假设主键ID为bigint类型，那么长度为

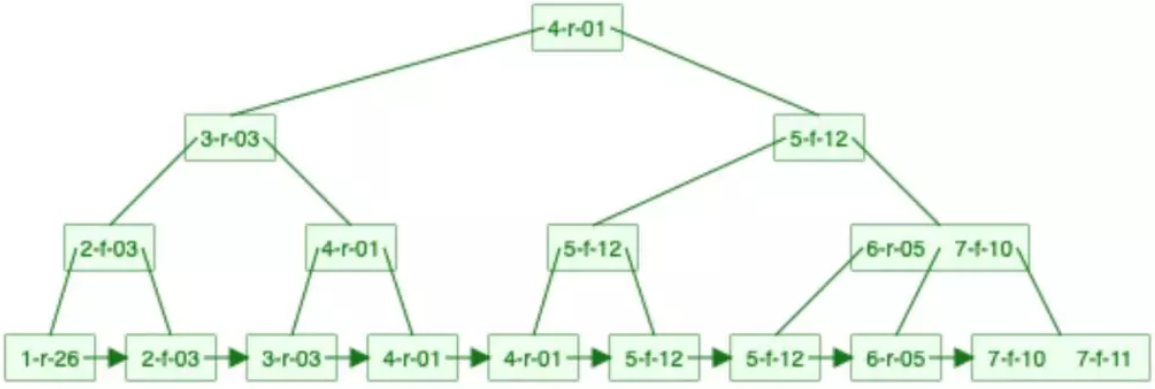
8B，指针大小在Innodb源码中为6B，一共就是14B，那么一页里就可以存储16K/14=1170个(主键+指针)，那么一颗高度为2的B+树能存储的数据为：117016=18720条，一颗高度为3的B+树能存储的数据为：11701170*16=21902400（千万级条）。所以在InnoDB中B+树高度一般为1-3层，它就能满足千万级的数据存储。在查找数据时一次页的查找代表一次IO，所以通过主键索引查询通常只需要1-3次IO操作即可查找到数据。所以也就回答了我们的问题，1页=16k这么设置是比较合适的，是适用大多数的企业的，当然这个值是可以修改的，所以也能根据业务的时间情况进行调整。

最左前缀原则

我们模拟数据建立一个联合索引 `select*,concat(right(emp_no,1),"-",right(title,1),"-",right(from_date,2))fromemployees.titles limit10;`

emp_no	title	from_date	to_date	concat(right(emp_no,1), "-", right(title,1), "-",...
10001	Senior Engineer	1986-06-26	9999-01-01	1-r-26
10002	Staff	1996-08-03	9999-01-01	2-f-03
10003	Senior Engineer	1995-12-03	9999-01-01	3-r-03
10004	Engineer	1986-12-01	1995-12-01	4-r-01
10004	Senior Engineer	1995-12-01	9999-01-01	4-r-01
10005	Senior Staff	1996-09-12	9999-01-01	5-f-12
10005	Staff	1989-09-12	1996-09-12	5-f-12
10006	Senior Engineer	1990-08-05	9999-01-01	6-r-05
10007	Senior Staff	1996-02-11	9999-01-01	7-f-11
10007	Staff	1989-02-10	1996-02-11	7-f-10

那么对应的B+树为



我们判断一个查询条件能不能用到索引，我们要分析这个查询条件能不能利用某个索引缩小查询范围

对于 `select*fromemployees.titleswhereemp_no=1`是能用到索引的，因为它能利用上面的索引所有查询范围，首先和第一个节点“4-r-01”比较，1<4，所以可以直接确定结果在左子树，同理，依次按顺序进行比较，逐步可以缩小查询范围。对于 `select*fromemployees.titleswheretitle='1'`是不能用到索引的，因为它不能用

到上面的所以，和第一节点进行比较时，没有empno这个字段的值，不能确定到底该去左子树还是右子树继续进行查询。对于`select*fromemployees.titleswheretitle='1'andemp_no=1`是能用到索引，按照我们的上面的分析，先用title='1'这个条件和第一个节点进行比较，是没有结果的，但是mysql会对这个sql进行优化，优化之后会将empno=1这个条件放到第一位，从而可以利用索引。