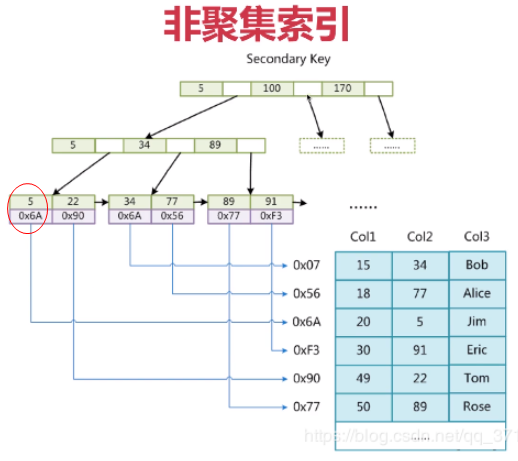
# mysql的索引

解决的问题：

1. 什么是聚集索引？

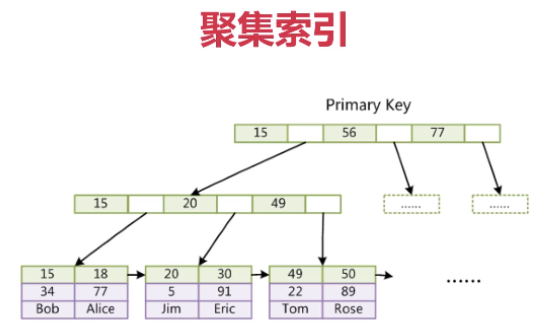
指的是 **指的是B+Tree 叶结点存的是指针还是数据记录**

**myisam使用的是非聚集索引，因为索引和数据记录分离**

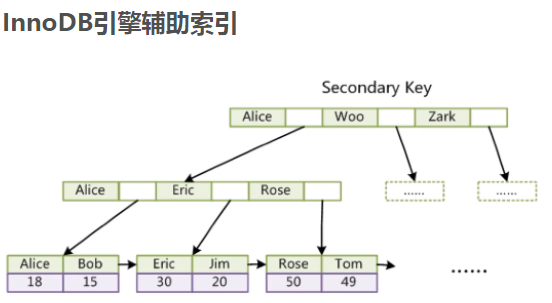


叶子节点 5 存放的是指针0x6A,指针指向的是数据的表文件（数据文件），索引文件和数据文件是分离的。# 非聚集索引，存储需要两个文件myisam\_table.MYD数据文件和myisam\_table.MYI索引文件

**innodb 数据文件就是索引文件，使用的是非聚集索引。**



叶子节点存储 主键 同时也存储 数据表里面的内容，叶子节点15 存放主键34，同时也存放数据Bob. *# 聚集索引，只有一个文件innodb\_table.idb，说明数据和索引存储在同一个文件*



辅助索引存储的是主键的值，通过主键找到对应的数据，先找到主键，然后再找数据，所以要比直接使用主键索引效率要慢。

**聚集和非聚集区别是在 B+Tree的叶结点存储数据还是指针 ,**

**MyISAM 默认使用索引是非聚集的，**

**InnoDB 主键索引是聚集索引**

1. 索引的本质是什么？

本质是 排好序的数据结构（b+tree）

1. 为什么会存在最左前缀原则

联合索引(name,age,sex)，b+数是按照从左到右的顺序来建立搜索树的，比如当(张三,20,F)这样的数据来检索的时候，b+树会优先比较name来确定下一步的所搜方向，如果name相同再依次比较age和sex，最后得到检索的数据；但当(20,F)这样的没有name的数据来的时候，b+树就不知道下一步该查哪个节点，因为建立搜索树的时候name就是第一个比较因子，必须要先根据name来搜索才能知道下一步去哪里查询。比如当(张三,F)这样的数据来检索时，b+树可以用name来指定搜索方向，但下一个字段age的缺失，所以只能把名字等于张三的数据都找到，然后再匹配性别是F的数据了， 这个是非常重要的性质，即索引的最左匹配特性。一条sql到底什么时候走索引？什么时候进行全表扫描？

## b+tree是如何生成的

## 索引优化实战

### 优化步骤

**1.创建索引**  
创建索引要注意并不是所有的where条件都要加上索引才会快，过多索引反而会降低查询速度。一般要在最常用的查询条件，order by,group by上加索引。使用一个索引来满足ORDER BY子句，WHERE 条件和 ORDER BY使用相同的索引，并且ORDER BY的顺序和索引顺序相同，并且ORDER BY的字段都是升序或者都是降序。

**2.查看级别type代表当前表的级别**



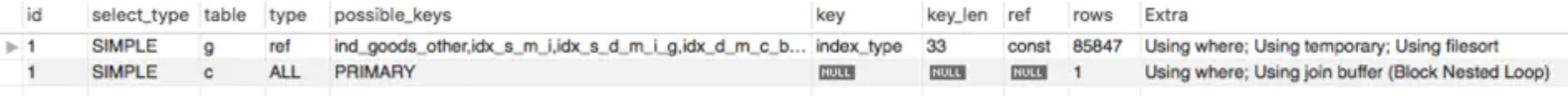
possible\_keys代表预测使用的索引  
key代表实际使用的索引  
注意：key在数据量不同的情况会出现变换索引的情况，这时候要充分考虑索引建立的健壮性。

**3.extra描述**  
Using temporary; Using filesort  
出现上边这两个条件说明该语句必须要优化了，Using temporary 如果有ORDER BY子句和一个不同的GROUP BY子句，或者如果ORDER BY或GROUP BY中的字段都来自其他的表而非连接顺序中的第一个表的话，就会创建一个临时表了。  
Using filesort 由于出现了临时表，此时使用order by就相当于一个全表排序。所以sql巨慢。

**4.思路**  
mysql在非直接关联的基础上进行排序会很慢，需要进行优化  
1.将主要表作为第一表，将其他表（如果是多表）非直接关联转变成直接关联（合多为一）作为新表  
**2.多表关联+排序(一大一小两表关联查询情况)  
(1)将条件(where,goup by,order by,limit)都转移到主表上，将主表数据查好后，  
(2)作为一张数据量很少的新表 再与其他表进行表关联查询  
步骤(1)中则使用索引保证索引不失效，步骤(2)则是一张limit后数据量关联一个小表，mysql查询性能完全足够支撑**

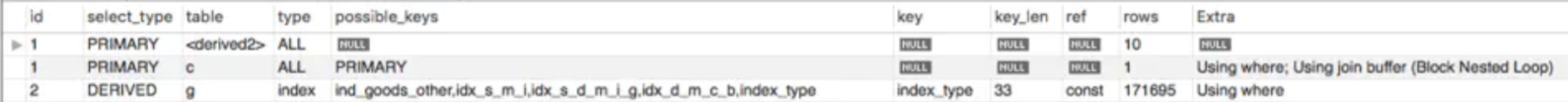
**5.实例**  
主表商品表50w数据，附表品牌表几百数据  
优化前查询时间最低为10s+，最高几分钟

explain select g.\*,c.store\_cat\_name from es\_goods g  
left join es\_store\_cat c on g.shop\_cat\_id=c.store\_cat\_id  
where g.seller\_id=1 and g.disabled=0 and g.market\_enable !=2  
and g.goods\_type= 'normal' order by g.create\_time desc limit 15000,10】



优化后查询时间最低10ms，最高1s

explain select g1.\* ,c.store\_cat\_name from (  
select g.\* from es\_goods g  
where g.seller\_id=1 and g.disabled=0 and g.market\_enable !=2  
and g.goods\_type='normal' order by g.create\_time desc limit 0,10) g1  
left join es\_store\_cat c on g1.shop\_cat\_id=c.store\_cat\_id;



如二次优化可使用覆盖索引等小技巧

聚集索引(主键索引)和辅助索引(二级索引)

聚集索引（主键索引）：

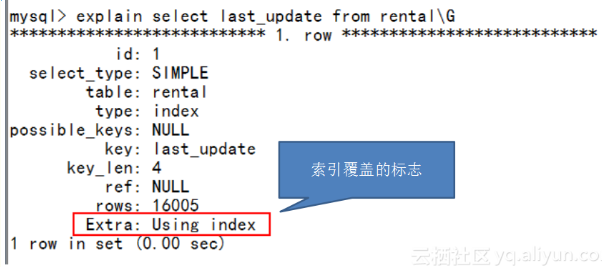
           聚集索引就是按照每张表的主键构造一颗B+树，同时叶子节点中存放的即为整张表的记录数据。

           聚集索引的叶子节点称为数据页，聚集索引的这个特性决定了索引组织表中的数据也是索引的一部分。

       辅助索引（二级索引）：

           非主键索引，叶子节点=键值+书签。Innodb存储引擎的书签就是相应行数据的主键索引值。

* 解释一： 就是select的数据列只用从索引中就能够取得，不必从数据表中读取，换句话说查询列要被所使用的索引覆盖。
* 解释二： 索引是高效找到行的一个方法，当能通过检索索引就可以读取想要的数据，那就不需要再到数据表中读取行了。如果一个索引包含了（或覆盖了）满足查询语句中字段与条件的数据就叫 做覆盖索引。
* 解释三：是非聚集组合索引的一种形式，它包括在查询里的Select、Join和Where子句用到的所有列（即建立索引的字段正好是覆盖查询语句[select子句]与查询条件[Where子句]中所涉及的字段，也即，索引包含了查询正在查找的所有数据）。
* 不是所有类型的索引都可以成为覆盖索引。覆盖索引必须要存储索引的列，而哈希索引、空间索引和全文索引等都不存储索引列的值，所以MySQL只能使用B-Tree索引做覆盖索引
* 当发起一个被索引覆盖的查询(也叫作索引覆盖查询)时，在EXPLAIN的Extra列可以看到“Using index”的信息



从执行结果上看，这个SQL语句只通过索引，就取到了所需要的数据，这个过程就叫做索引覆盖。

explain select sql\_no\_cache ，rental\_date from t1 where inventory\_id<80000\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: t1

type: range

possible\_keys: inventory\_id

key: inventory\_id

key\_len: 3

ref: NULL

rows: 153734

Extra: Using index condition

1 row in set (0.00 sec)

Extra：Using index condition 表示使用的索引方式为二级检索，即79999个书签值被用来进行回表查询。可想而知，还是会有一定的性能消耗的

　　尝试针对这个SQL建立联合索引，如下：

**alter table t1 add key(inventory\_id,rental\_date);**

explain select sql\_no\_cache rental\_date from t1 where inventory\_id<80000\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: t1

type: range

possible\_keys: inventory\_id,inventory\_id\_2

key: inventory\_id\_2

key\_len: 3

ref: NULL

rows: 162884

Extra: Using index

1 row in set (0.00 sec)

Extra：Using index 表示没有会标查询的过程，实现了索引覆盖

**查询语句：**

select order\_code,order\_amount from t\_order order by order\_code limit **1000**;

    发现虽然在order\_code上建了索引，但是看查询计划却不走索引，为什么呢？因为数据行读取order\_amount，所以是随机IO。那怎么办？重新建索引，使用覆盖索引。

ALTER TABLE `t\_order` ADD INDEX `idx\_ordercode\_orderamount` USING BTREE (`order\_code` ASC, `order\_amount` ASC);

这样再查看SQL的执行计划，就发现可以走到索引了。

### 总结：覆盖索引的优化及限制

 覆盖索引是一种非常强大的工具，能大大提高查询性能，只需要读取索引而不需要读取数据，有以下优点：

 1、索引项通常比记录要小，所以MySQL访问更少的数据。

 2、索引都按值得大小存储，相对于随机访问记录，需要更少的I/O。

 3、数据引擎能更好的缓存索引，比如MyISAM只缓存索引。

 4、覆盖索引对InnoDB尤其有用，因为InnoDB使用聚集索引组织数据，如果二级索引包含查询所需的数据，就不再需要在聚集索引中查找了。

**限制：**

 1、覆盖索引也并不适用于任意的索引类型，索引必须存储列的值。

 2、Hash和full-text索引不存储值，因此MySQL只能使用BTree。

 3、不同的存储引擎实现覆盖索引都是不同的，并不是所有的存储引擎都支持覆盖索引。

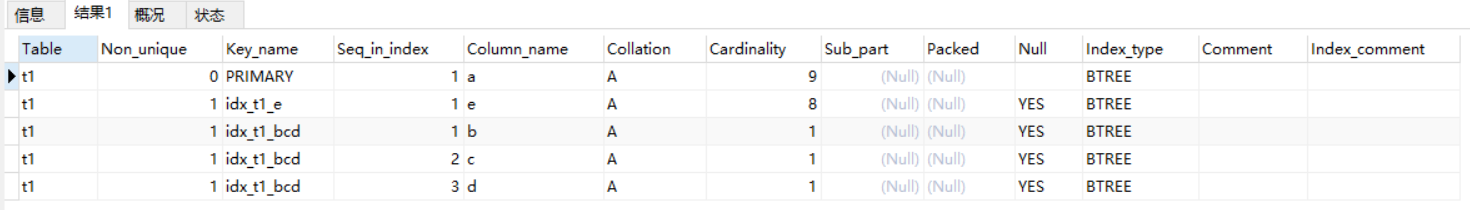
 4、如果要使用覆盖索引，一定要注意SELECT列表值取出需要的列，不可以SELECT \* ，因为如果将所有字段一起做索引会导致索引文件过大，查询性能下降。

## 3 面试分析

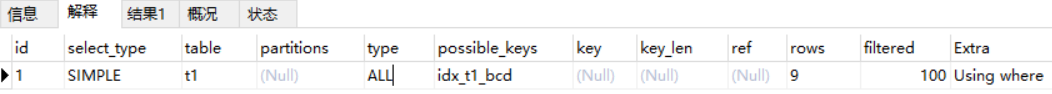
sql:

|  |
| --- |
| create table t1(  a int primary key,  b int,  c int,  d int,  e varchar(20)  )engine=InnoDB;  insert into t1 values(4,3,1,1,'d');  insert into t1 values(1,1,3,1,'a');  insert into t1 values(8,8,3,6,'c');  insert into t1 values(2,3,2,4,'c');  insert into t1 values(5,3,6,1,'f');  insert into t1 values(3,2,8,2,'g');  insert into t1 values(7,4,1,1,'r');  insert into t1 values(6,3,9,8,'y');  insert into t1 values(9,6,1,1,'w');  create index idx\_t1\_bcd on t1(b,c,d);  create index idx\_t1\_e on t1(e);  alter table t1 drop index idx\_t1\_bcd;  show index from t1; |

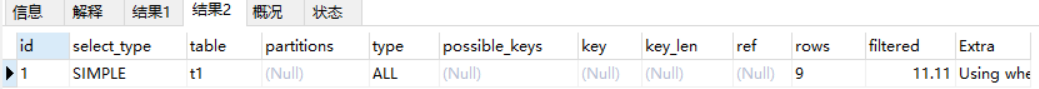




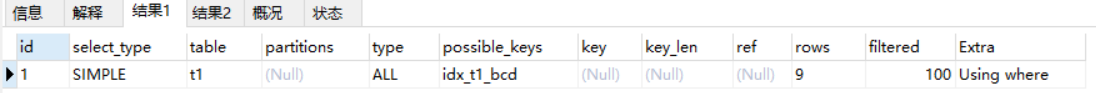
select \* from t1 where b =3;



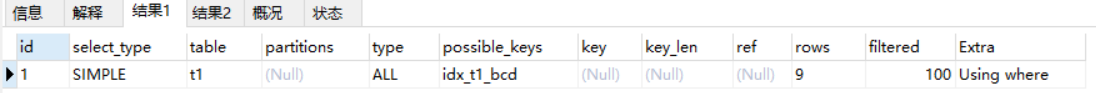
EXPLAIN select \* from t1 where c =1;

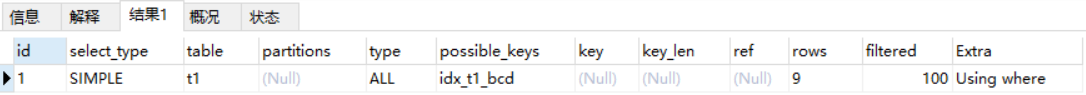


EXPLAIN select \* from t1 where b=3 and c =1;

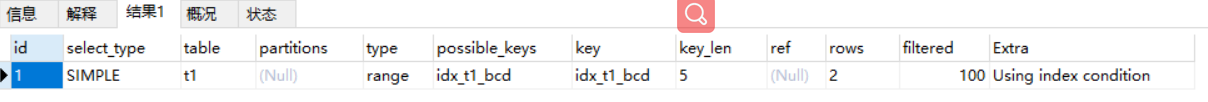


EXPLAIN select \* from t1 where c =1 and b=3;



EXPLAIN select \* from t1 where b >1; 

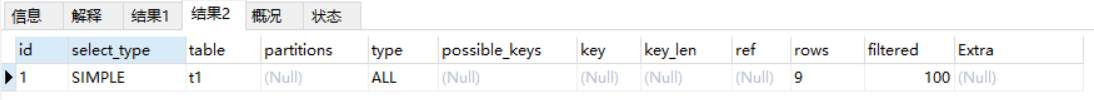
EXPLAIN select \* from t1 where b >5;



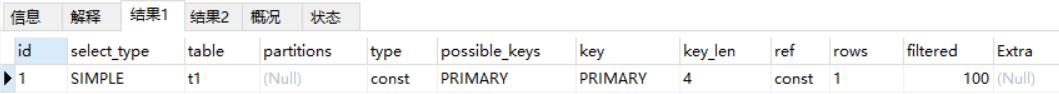
EXPLAIN select b from t1;



EXPLAIN select b,e from t1;



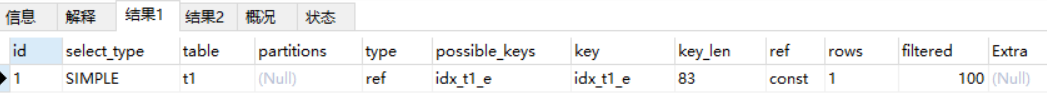
EXPLAIN select \* from t1 where a='1';



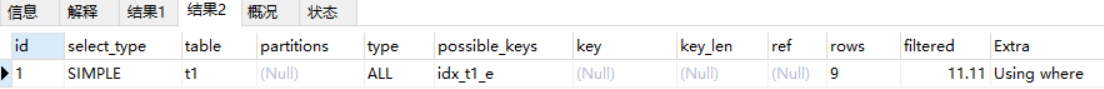
EXPLAIN select \* from t1 where a=1;



EXPLAIN select \* from t1 where e='1';



EXPLAIN select \* from t1 where e=1;



EXPLAIN select \* from t1 ORDER BY b asc, c asc, d asc;



EXPLAIN select \* from t1 ORDER BY b asc, c asc, d desc;



mysql基本概念：

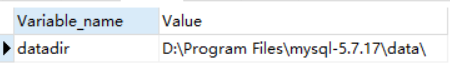
show GLOBAL STATUS like 'innodb\_page\_size'; --页 4kb



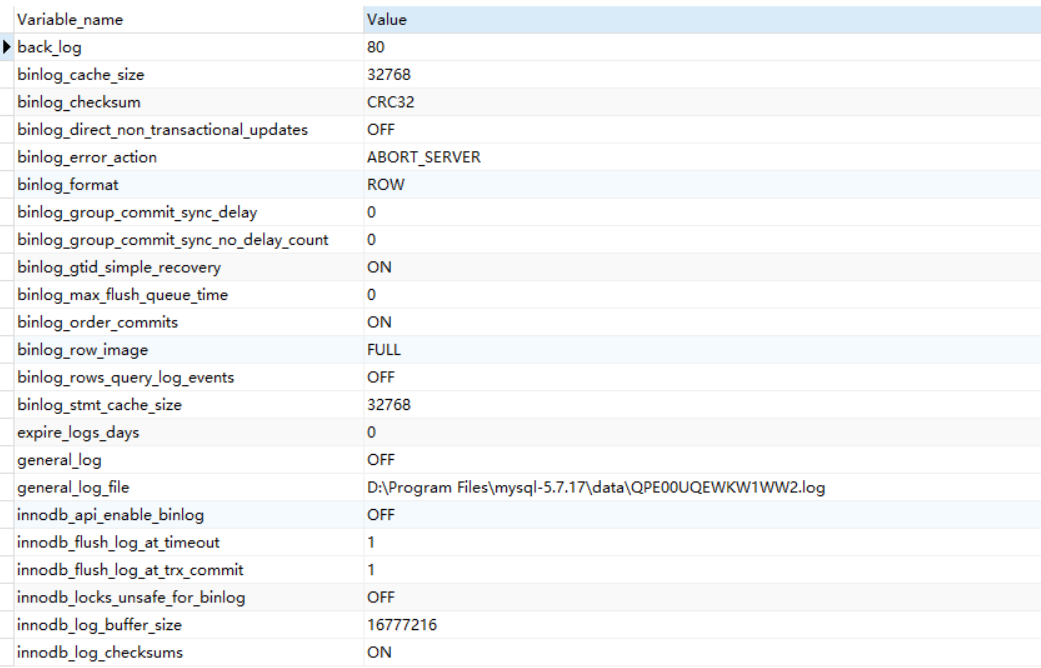
select 16384/1024 ;

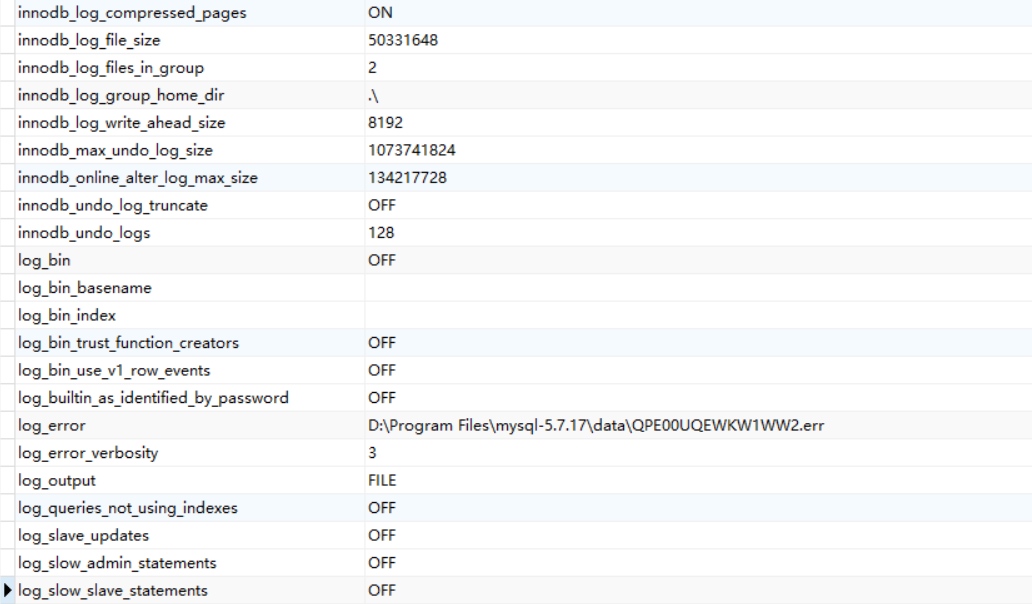


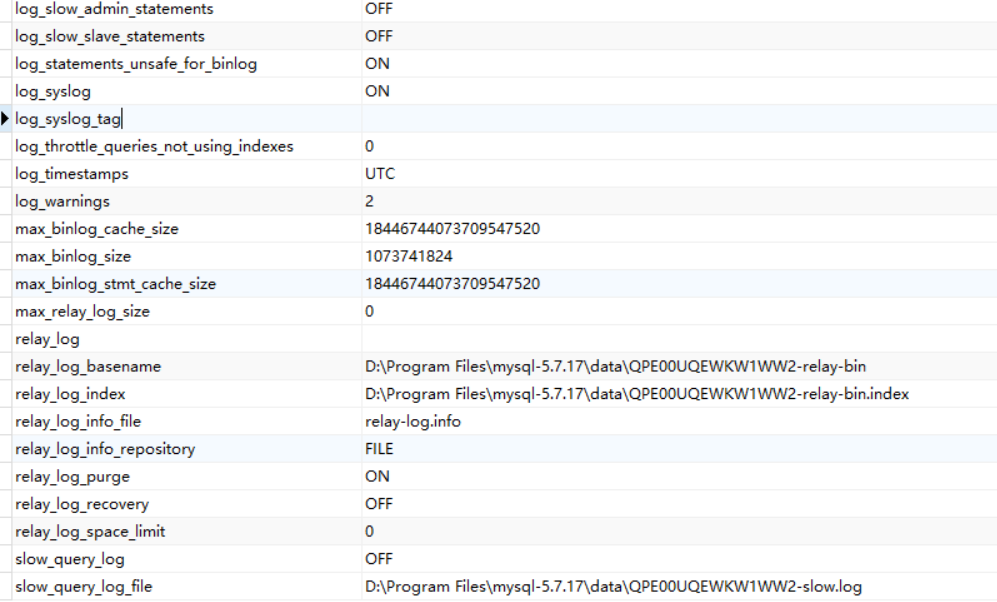
show GLOBAL VARIABLES like '%datadir%';

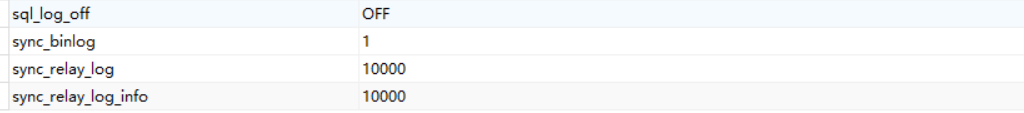


show GLOBAL VARIABLES like '%log%';









show GLOBAL VARIABLES like '%innodb\_file\_per\_table%';



show GLOBAL VARIABLES like '%tmp\_table\_size%';

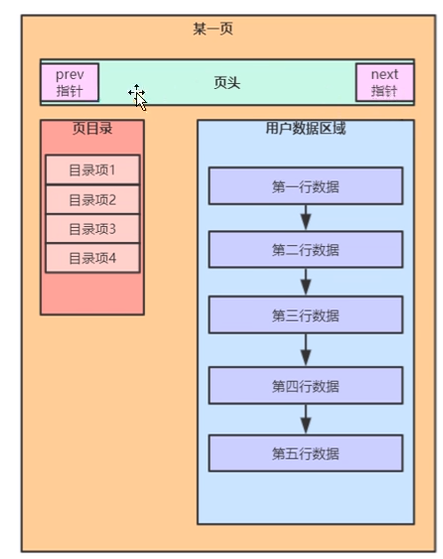


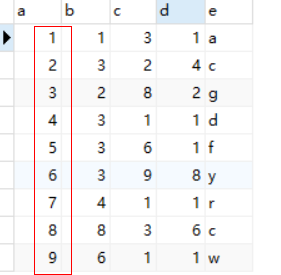
show GLOBAL VARIABLES like '%innodb\_stats\_method%';

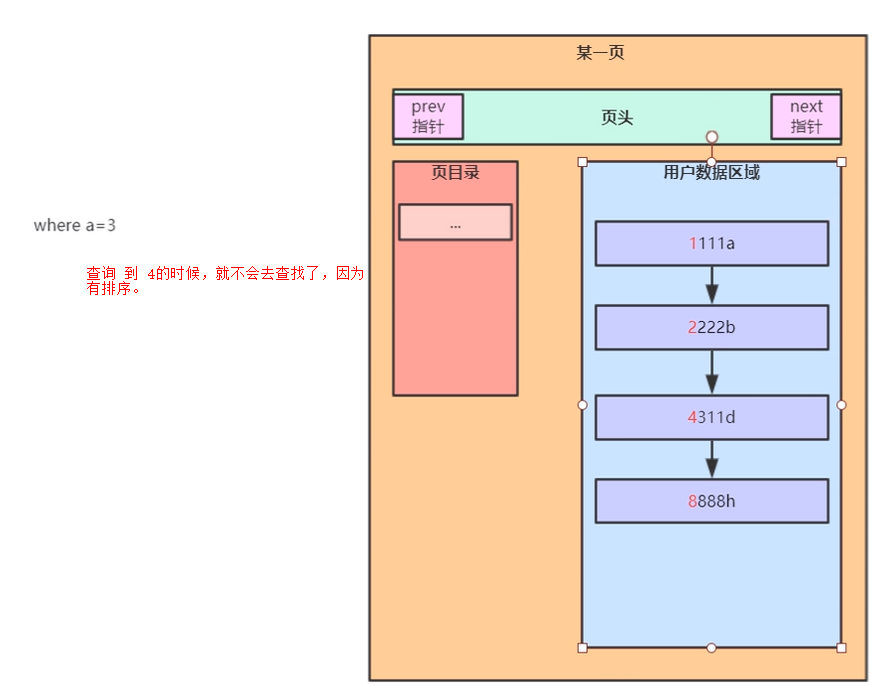


去磁盘取出第一条数据 放到内存 交给cpu去比较，如果不符合，继续去磁盘取第二条数据（两次磁盘io了），每次从磁盘取出一条数据就不好了，引出一个 页 的概念，页（16kb），这里的8条数据，比如这里一行数据占了10多个字节，去取第一条数据的时候顺便把以下的数据行业放到内存中，下次去从内存中找数据。

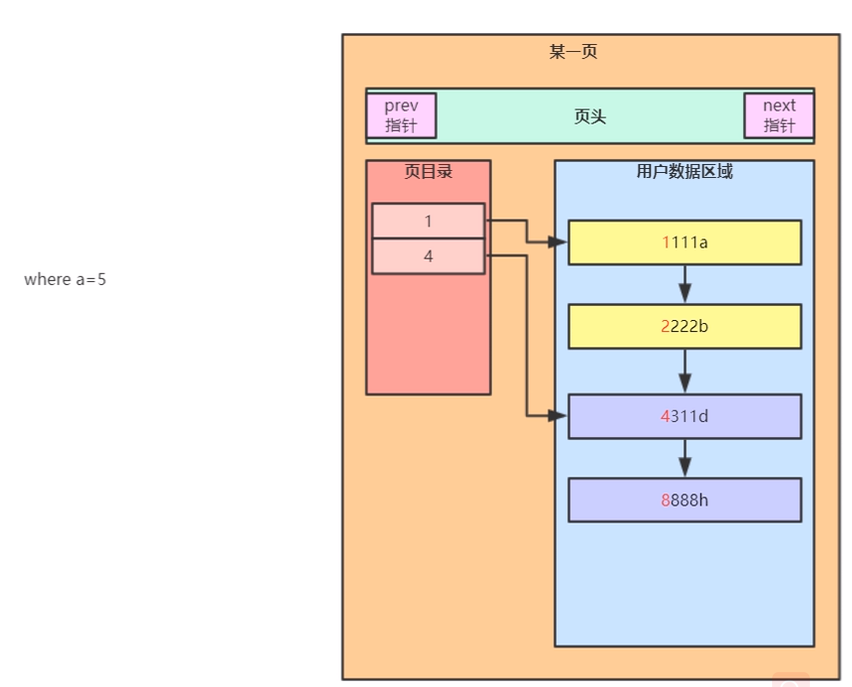
页结构



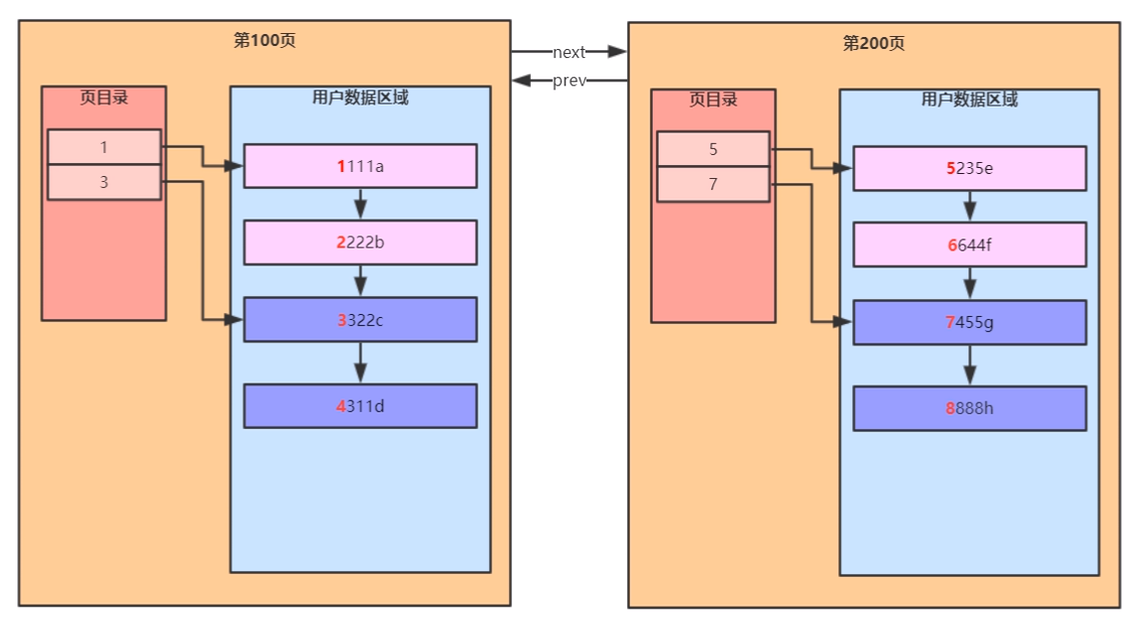
在插入的时候按照 主键 升序排序，即使主键有乱序，mysql内部会自动给排序了。



这里这时候会出现 链表的查询结构，如果查询的数字 比较大的时候，这个链表就太长了，影响查询效率，要改进。

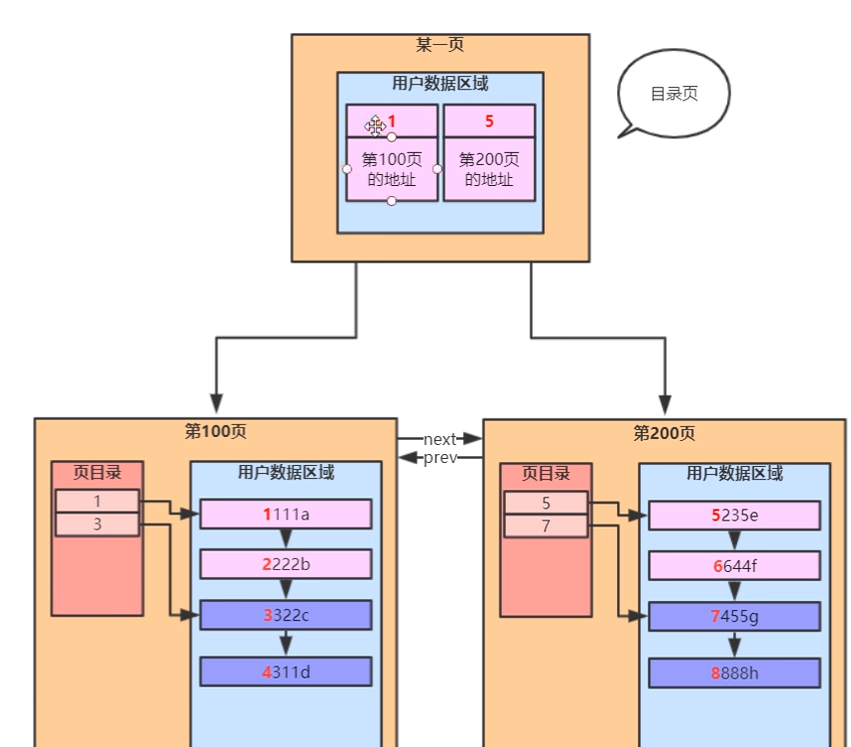


两页（如果一行记录占得字节比较大，一页只能放4条记录）



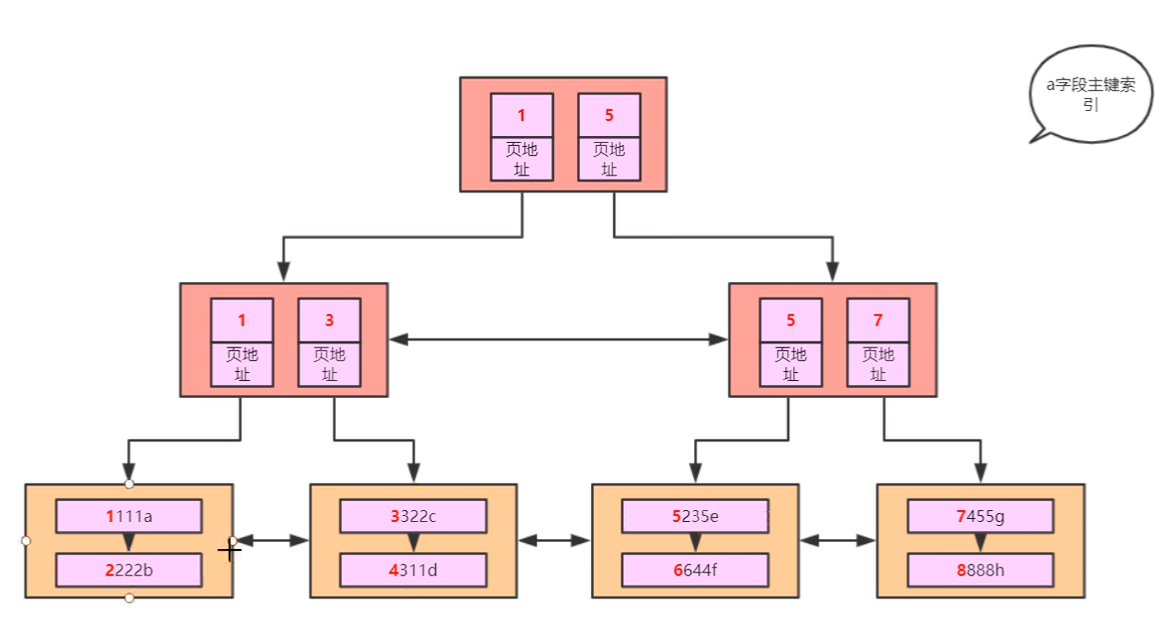
利用页目录解决了 数据行 形成链表过长的问题，但是页数比较多的情况呢？

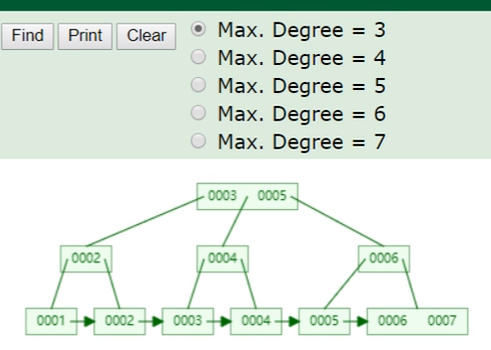
目录页 数据页 页目录（目录项）



目录 就是 索引

如果是两条记录占用一页的话：





叶子节点都有一个冗余，冗余索引的键值。

全表扫描：从叶子节点开始找

走索引：从目录页 开始查找

如果没有主键 如何排序呢？就会把唯一索引（去找）当一个主键来排序。唯一索引也没有的话，自己生成一个隐藏的id（rowid），是一定要有一个主键的。

create index idx\_t1\_bcd on t1(b,c,d);

