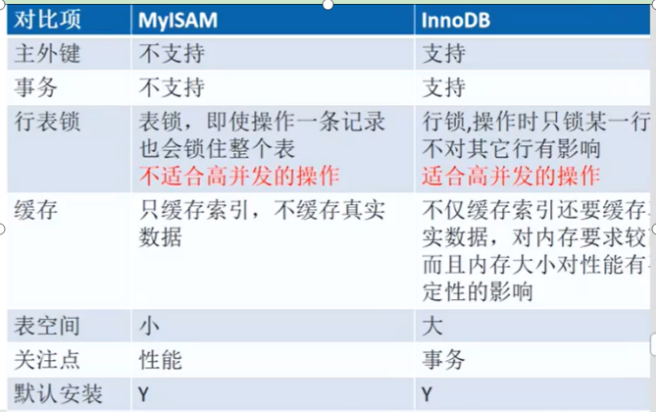
**MySQL**

# MyISAM和InnoDB的区别

1. MyISAM只支持表锁，InfoDB支持表锁和行锁。
2. MyISAM不支持事务，InfoDB支持事务。



# MyISAM

## 共享读锁

1. 创建一张表

|  |
| --- |
| CREATE TABLE `testisam` (  `id` int(11) DEFAULT NULL  ) ENGINE=MyISAM DEFAULT CHARSET=utf8mb4; |

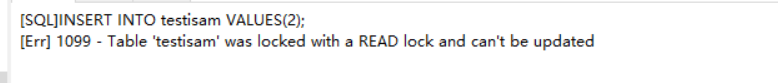
1. 插入数据

|  |
| --- |
| INSERT INTO testisam VALUES(2); |

1. 开启行锁

|  |
| --- |
| LOCK TABLE testisam READ; |

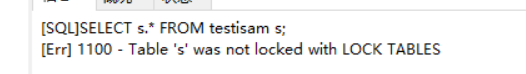
1. 再次执行第二步中的插入语句，结果如下：



1. 加上别名

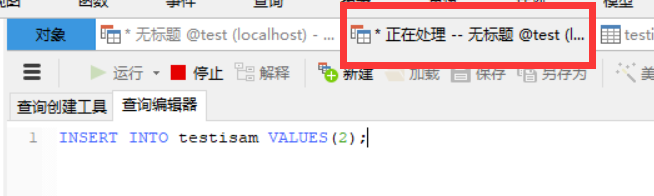
|  |
| --- |
| SELECT s.\* FROM testisam s; |

查询报错:

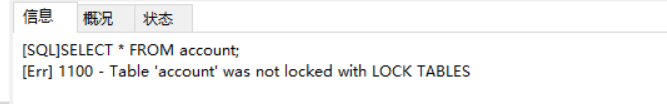


1. 打开另一个会话，继续进行插入语句，结果如下:

处于等待状态



1. 在开启行锁的相同的会话中，查询其他的表，结果如下:



1. 解锁

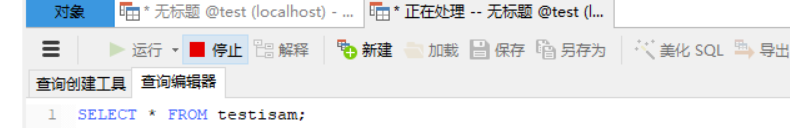


## 写锁

1. 开启读锁

|  |
| --- |
| LOCK TABLE testisam WRITE; |

1. 执行插入、更新、删除、查询语句都是可以的。
2. 查询其他表也是不能成功的（和开启读锁同一个会话）。
3. 在另一个会话中，查询testisam 中的数据，结果：**等待中**



## 总结锁



# InnoDB

## innodb\_page\_size

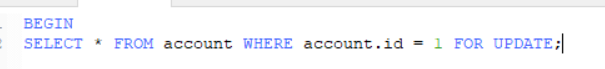
show variables like 'innodb\_page\_size';



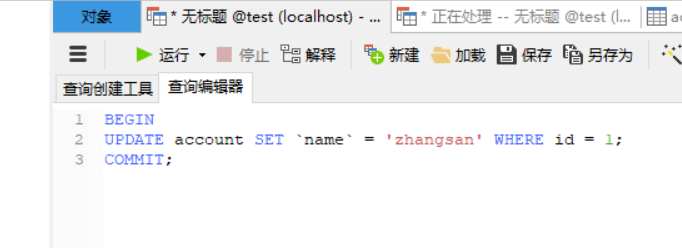
只支持B-tree索引

## 行锁

1. 开启行锁



**Update，删除语句会自动添加行锁**



1. 在另一个会话中修改数据，id一样，结果**:在等待，因为开启了行锁，在将上面的事务提交之后，该条数据自动修改成功。**



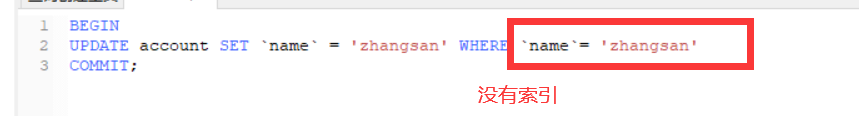
**超时之后，自动取消**



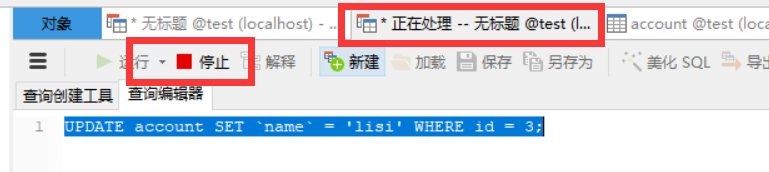
1. 在另一个会话中修改数据，但是id不一样，结果是可以修改成功，因为和锁住的数据的id不一样。

---------------------------------------------------------------------------------------------------------------------------------

1. 开启事务，执行一条修改语句，条件涉及的字段是没有索引的

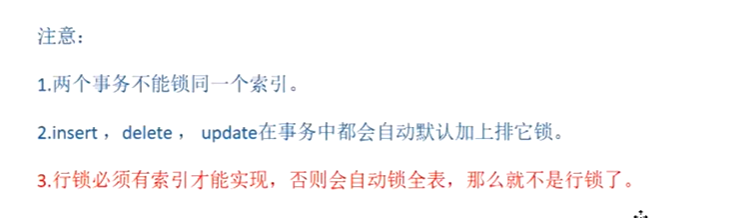


1. 在另一个会话中修改数据，与上个语句的条件不一样，结果:数据不一样，为啥还修改不成功呢？答案：在上一个会话中SQL语句涉及的字段是没有索引的，那么修改的时候会锁住所有的数据，所以这里修改的时候会等待上一个事务提交。



## 总结（行锁）

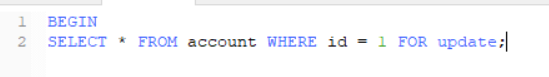
两条SQL语句，id一样，但是只能有一条才能修改成功，如果加上版本号，一条数据修改成功之后，版本号+1，另一条SQL语句再执行不会成功，因为版本号已经改变了，这就是乐观锁。

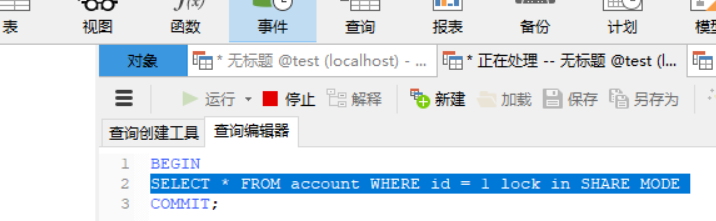


## 查询锁的信息



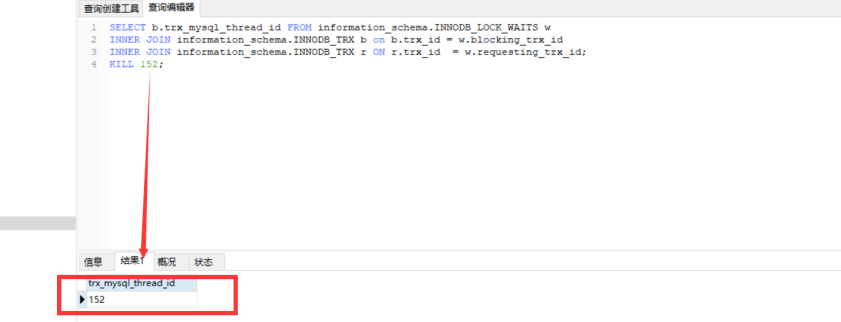
## 解决锁的问题，在开发过程中可能会遇到



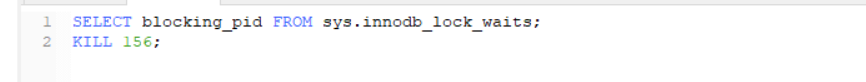


执行这个第二提交SQL语句，在等待中，解决方法如下(5.6版本):

|  |
| --- |
| SELECT b.trx\_mysql\_thread\_id FROM information\_schema.INNODB\_LOCK\_WAITS w  INNER JOIN information\_schema.INNODB\_TRX b on b.trx\_id = w.blocking\_trx\_id  INNER JOIN information\_schema.INNODB\_TRX r ON r.trx\_id = w.requesting\_trx\_id; |



### 5.7版本解决锁等待的问题



### 查看锁的超时时间

SHOW GLOBAL VARIABLES LIKE 'innodb\_lock\_wait\_timeout';

# 事务

## 什么是事务

事务是指是程序中一系列严密的逻辑操作，而且所有操作必须全部成功完成，否则在每个操作中所作的所有更改都会被撤消。可以通俗理解为：就是把多件事情当做一件事情来处理，好比大家同在一条船上，要活一起活，要完一起完 。

## 事务的特性

* **原子性**（Atomicity）**：**操作这些指令时，要么全部执行成功，要么全部不执行。只要其中一个指令执行失败，所有的指令都执行失败，数据进行回滚，回到执行指令前的数据状态。
* **一致性**（Consistency）**：**事务的执行使数据从一个状态转换为另一个状态，但是对于整个数据的完整性保持稳定。
* **隔离性**（Isolation）**：**隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。
* **持久性**（Durability）**：**当事务正确完成后，它对于数据的改变是永久性的。

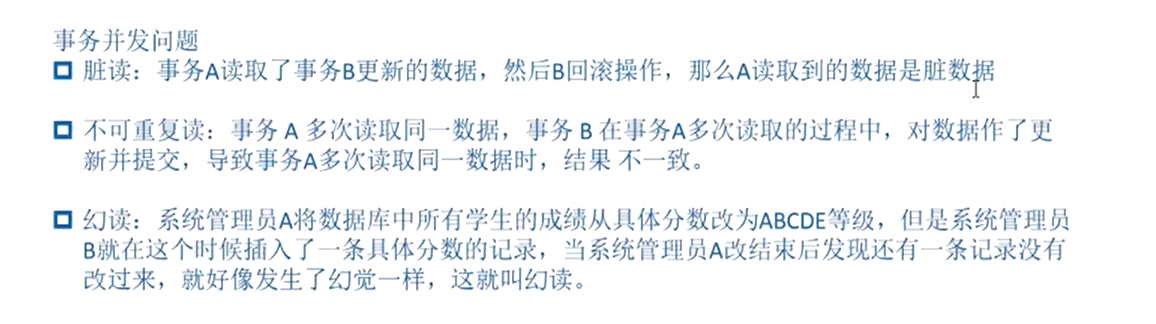
## 隔离性

未提交读(脏读)

已提交读(不可重复读)

可重复读

序列化



## 通过案例来演示事务的隔离性

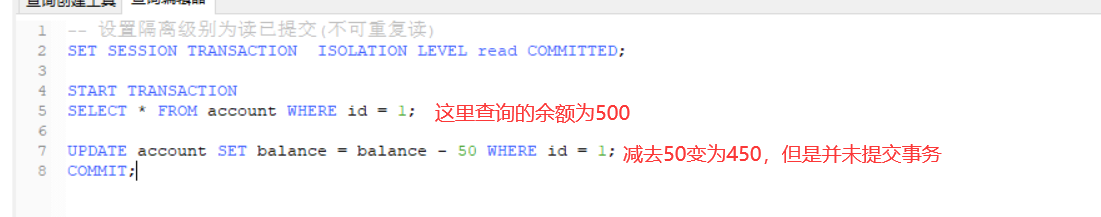
### 读未提交

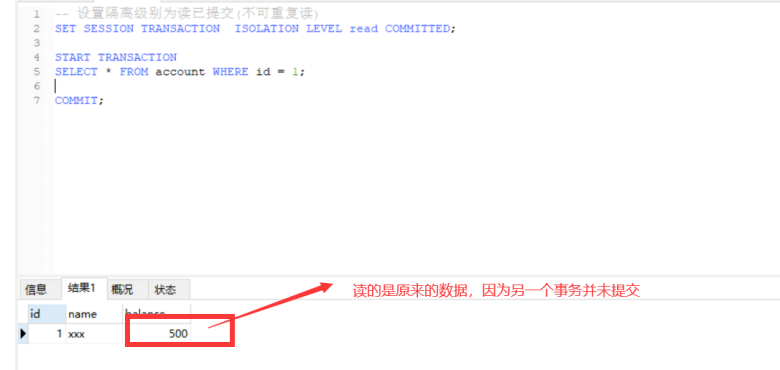




### 读已提交（不可重复读）

|  |
| --- |
| -- 设置隔离级别为读已提交(不可重复读)  SET SESSION TRANSACTION ISOLATION LEVEL read COMMITTED;  START TRANSACTION  SELECT \* FROM account WHERE id = 1;  UPDATE account SET balance = balance - 50 WHERE id = 1;  COMMIT; |



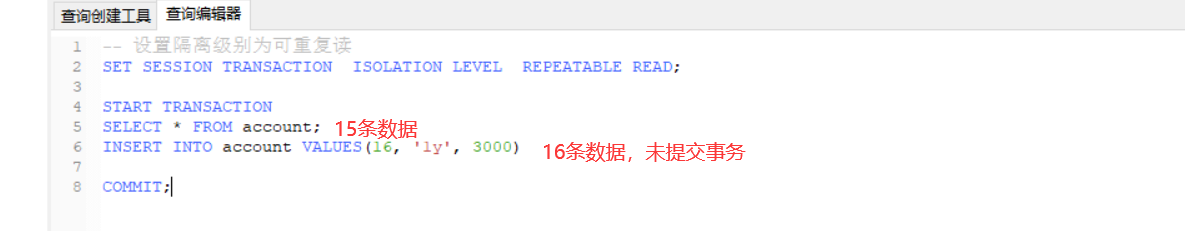


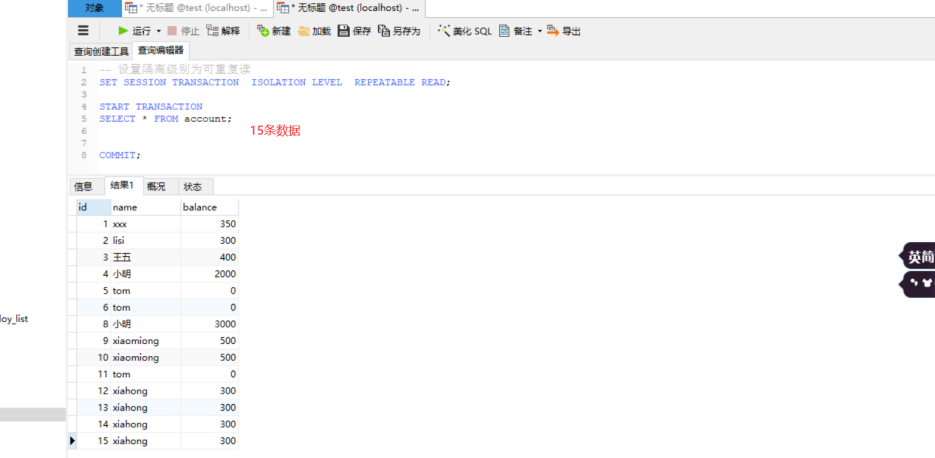
将第一个事务提交，余额变为450，第二个会话再去查询，结果也变为了450，第二个会话查询的两次的结果并不一样，这种情况叫做不可重复读。

### 可重复读

解决了不可重复读的问题

**存在的问题:幻读**



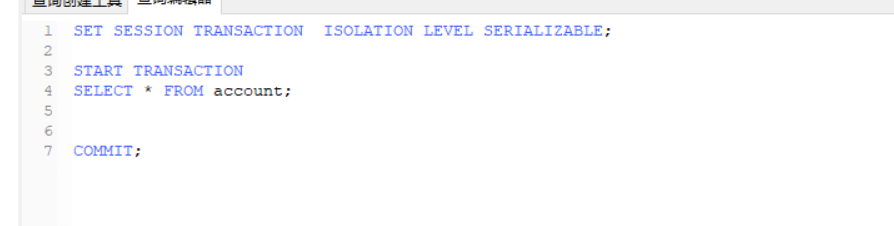


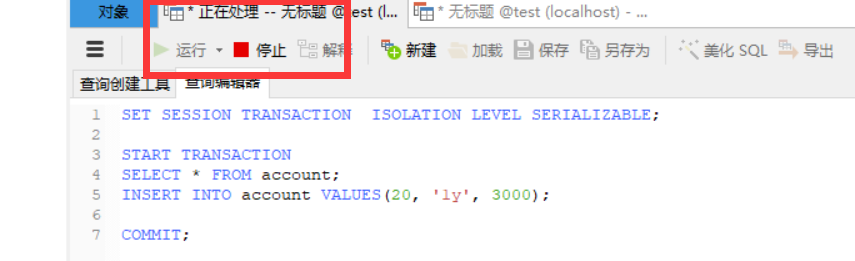
将第一个会话中的事务提交

再在第二会话中查询，结果如下:结果还是15条数据，然后再插入一条id为16的数据，却插入不进去，因为id=16的数据已经存在了，这种情况叫做幻读。

### 可序列化

事务串行执行，一个一个执行，a事务在执行，b执行必须等到a事务提交之后才能执行。

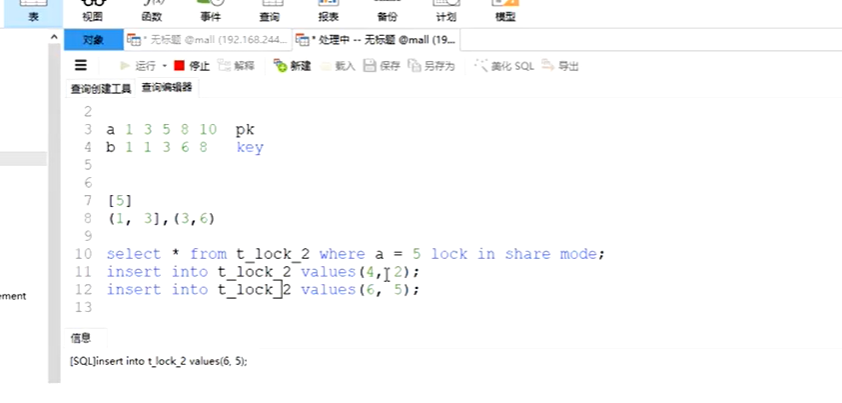




# 间隙锁

<https://www.cnblogs.com/crazylqy/p/7821481.html>

a为主键，b为普通的索引

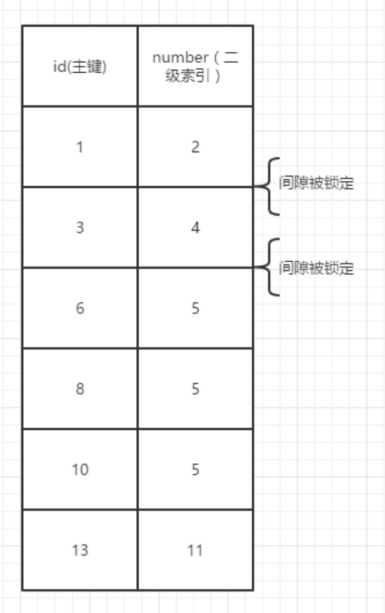


|  |
| --- |
| CREATE TABLE `grap` (  `id` int(11) NOT NULL,  `number` int(11) DEFAULT NULL,  PRIMARY KEY (`id`),  KEY `number` (`number`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4; |

## 案例一:

|  |
| --- |
| session **1**:  start transaction ;select \* from news where number=**4** for update ;  session **2**:  start transaction ;  insert into news value(**2**,**4**);#（阻塞）  insert into news value(**2**,**2**);#（阻塞）  insert into news value(**4**,**4**);#（阻塞）  insert into news value(**4**,**5**);#（阻塞）  insert into news value(**7**,**5**);#（执行成功）  insert into news value(**9**,**5**);#（执行成功）  insert into news value(**11**,**5**);#（执行成功） |

**检索条件number=4,向左取得最靠近的值2作为左区间，向右取得最靠近的5作为右区间，因此，session 1的间隙锁的范围（2，4），（4，5），如下图所示：**



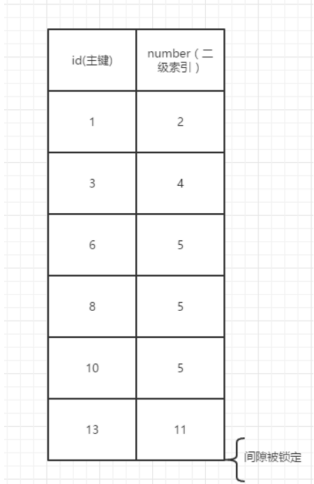
间隙锁锁定的区间为（2，4），（4，5），即记录（id=1,number=2）和记录（id=3,number=4）之间间隙会被锁定，记录（id=3,number=4）和记录（id=6,number=5）之间间隙被锁定。

因此记录（id=2,number=4），（id=2,number=2），（id=4,number=4），（id=4,number=5）正好处在（id=3,number=4）和（id=6,number=5）之间，所以插入不了，需要等待锁的释放，而记录(id=7,number=5)，（id=9,number=5），（id=11,number=5）不在上述锁定的范围内，因此都会插入成功。

## 案例二:

|  |
| --- |
| SESSION 1;  start transaction ;  select \* from grap where number=13 for update ;  SESSION 2;  start transaction ;  insert into grap value(11,5);#(执行成功)  insert into grap value(12,11);#(执行成功)  insert into grap value(14,11);#(阻塞)  insert into grap value(15,12);#(阻塞)  update grap set id=14 where number=11;#(阻塞)  update grap set id=11 where number=11;#(执行成功) |

检索条件number=13,向左取得最靠近的值11作为左区间，向右由于没有记录因此取得无穷大作为右区间，因此，session 1的间隙锁的范围（11，无穷大），如下图所示：



此表中没有number=13的记录的，innodb依然会为该记录左右两侧加间隙锁，间隙锁的范围（11，无穷大）。

有人会问，为啥update news set id=14 where number=11会阻塞，但是update news set id=11 where number=11却执行成功呢？

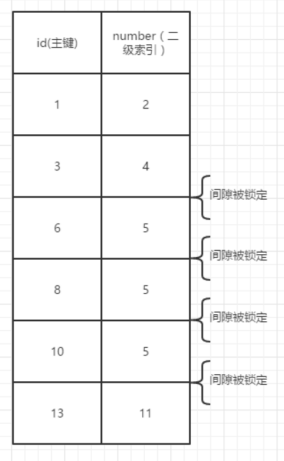
间隙锁采用在指定记录的前面和后面以及中间的间隙上加间隙锁的方式避免数据被插入，此图间隙锁锁定的区域是（11，无穷大），也就是记录（id=13,number=11）之后不能再插入记录，update news set id=14 where number=11这条语句如果执行的话，将会被插入到（id=13,number=11）的后面，也就是在区间（11，无穷大）之间，由于该区间被间隙锁锁定，所以只能阻塞等待，而update news set id=11 where number=11执行后是会被插入到（id=13,number=11）的记录前面，也就不在（11，无穷大）的范围内，所以无需等待，执行成功。

## 案例三

|  |
| --- |
| session 1:  start transaction ;  select \* from grap where number=5 for update;  session 2:  start transaction ;  insert into grap value(4,4);#(阻塞)  insert into grap value(4,5);#(阻塞)  insert into grap value(5,5);#(阻塞)  insert into grap value(7,11);#(阻塞)  insert into grap value(9,12);#(执行成功)  insert into grap value(12,11);#(阻塞)  update grap set number=5 where id=1;#(阻塞)  update grap set id=11 where number=11;#(阻塞)  update grap set id=2 where number=4 ;#（执行成功）  update grap set id=4 where number=4 ;#（阻塞）  ROLLBACK |

插入的时候先按照主键排序，再按照number排序

检索条件number=5,向左取得最靠近的值4作为左区间，向右取得11为右区间，因此，session 1的间隙锁的范围（4，5），（5，11），如下图所示：



有人会问，为啥insert into news value(9,12)会执行成功？间隙锁采用在指定记录的前面和后面以及中间的间隙上加间隙锁的方式避免数据被插入，（id=9,number=12）很明显在记录（13,11）的后面，因此不再锁定的间隙范围内。

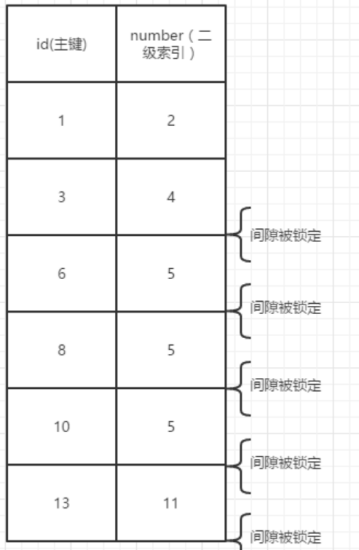
为啥update news set number=5 where id=1会阻塞？  
number=5的记录的前面，后面包括中间都被封锁了，你这个update news set number=5 where id=1根本没法执行，因为innodb已经把你可以存放的位置都锁定了，因为只能等待。

同理，update news set id=11 where number=11由于记录（id=10,number=5）与记录（id=13,number=11）中间的间隙被封锁了，你这句sql也没法执行，必须等待，因为存放的位置被封锁了。

## 案例四

|  |
| --- |
| session 1:  start transaction;  select \* from grap where number>4 for update;  session 2:  start transaction;  update grap set id=2 where number=4 ;#(执行成功)  update grap set id=4 where number=4 ;#(阻塞)  update grap set id=5 where number=5 ;#(阻塞)  insert into grap value(2,3);#(执行成功)  insert into grap value(null,13);#(阻塞)  ROLLBACK; |

检索条件number>4,向左取得最靠近的值4作为左区间，向右取无穷大，因此，session 1的间隙锁的范围（4，无穷大），如下图所示：

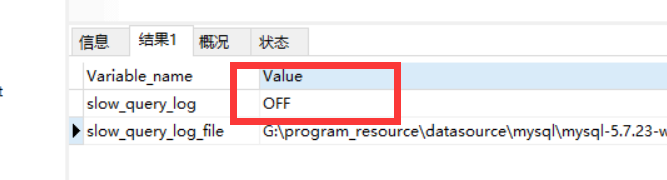


session2中之所以有些阻塞，有些执行成功，其实就是因为插入的区域被锁定，从而阻塞。

# 慢查询

show VARIABLES LIKE '%slow\_query\_log%'

默认是关闭的

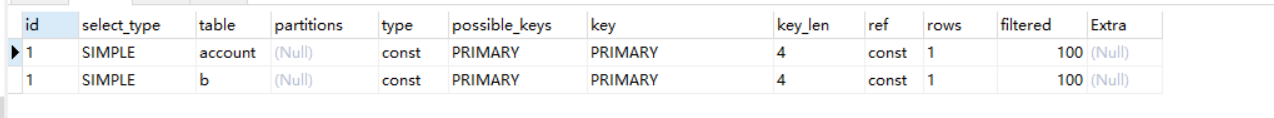


SET GLOBAL slow\_query\_log = 1;// 开启慢查询

SET GLOBAL long\_query\_time=0;// 查过0秒的查询都会记录慢查询日志文件中

可能修改查询时间并没有生效，只要将会话关闭再重新打开就生效了。

# 执行计划



## Id

1. Id相同，执行顺序由上到下
2. Id不同，id越大，优先级越高
3. Id相同不同

# Hash和B+tree

## 扇出

<https://blog.csdn.net/shenchaohao12321/article/details/83243314>

<https://blog.csdn.net/hao65103940/article/details/89032538>

索引的分裂和合并

https://blog.csdn.net/shenchaohao12321/article/details/83243314

## Hash

Hash 索引结构的特殊性，其检索效率非常高，索引的检索可以一次定位，不像B-Tree 索引需要从根节点到枝节点，最后才能访问到页节点这样多次的IO访问，所以 Hash 索引的查询效率要远高于 B-Tree 索引。

可能很多人又有疑问了，既然 Hash 索引的效率要比 B-Tree 高很多，为什么大家不都用 Hash 索引而还要使用 B-Tree 索引呢？任何事物都是有两面性的，Hash 索引也一样，虽然 Hash 索引效率高，但是 Hash 索引本身由于其特殊性也带来了很多限制和弊端，主要有以下这些。

（1）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。

（2）Hash 索引无法被用来避免数据的排序操作。

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避免任何排序运算；

（3）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。

（4）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。

1. 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。

## 设计索引的规则

1. 最适合索引的列是出现在where子句中的列，或连接子句中指定的列，而不是出现在select关键字后的选择列表中的列。
2. 索引的列的基数越大，索引的效果越好。
3. 尽量使用短索引。能够节省大量索引空间，也可能使查询更快。
4. 不要过度索引。索引都要占用额外的磁盘空间，并降低写操作的性能。在修改表的内容时，索引必须进行更新，有时可能需要重构。

## BTREE索引和HASH索引的比较

1、B-Tree索引

索引存储的值按索引列中的顺序排列。可以用B-Tree索引进行全关键字、关键字范围和关键字前缀查询。如果使用索引，必须保证按索引最左边前缀进行查询。由于B树中节点是顺序存储的，可以对查询结果进行order by。

限制：

1)查询必须从索引的最左边的列开始

2)不能跳过某一索引列。

3)存储引擎不能使用索引中范围条件右边的列。

例如，如果你的查询语句为WHERE last\_name=”Smith” AND first\_name LIKE ‘J%’ AND dob=’1976-12-23’，则该查询只会使用索引中的前两列，因为LIKE是范围查询，所以dob是不能使用索引的。

1. Hash索引

MySQL中只有Memory存储引擎显示支持hash索引，是默认索引类型，它也支持B-Tree索引。

如果多个值有相同的hash code，索引把它们的行指针用链表保存到同一个hash表项中。

因为索引自己仅仅存储很短的值，所以索引非常紧凑，hash值不取决于列的数据类型，

int列的索引和长字符串列的索引一样大。

1. 总结
2. HASH索引适合等式比较的操作，不能用来加速order by操作，也不能确定在两个值之间大约有多少行，会影响一些查询的执行效率。而且只能使用整个关键字来搜索一行。
3. BTREE索引，使用大于，小于，BETWEEN，不等于，LIKE等操作符的时候都可以用。对索引字段进行范围查询的时候，只有BTREE索引可以通过索引访问。HASH索引实际上是全表扫描的。

**如果对多列进行索引，列的顺序非常重要，MySQL仅能对索引最左边的前缀进行有效查找。**

假设存在组合索引it1c1c2(c1,c2)，查询语句select \* from t1 where c1=1 and c2=2能够

使用该索引。查询语句select \* from t1 where c1=1也能够使用该索引。

但是，查询语句select \* from t1 where c2=2不能够使用该索引，

因为没有组合索引的引导列，即，要想使用c2列进行查找，必需出现c1等于某值。

索引是在存储引擎中实现的，不是在服务器层中实现的。所以，每种存储引擎的索引都不

一定完全相同，并不是所有的存储引擎都支持所有的索引类型。

## 高性能索引策略

### 聚簇索引

https://baijiahao.baidu.com/s?id=1645514817836645220&wfr=spider&for=pc&qq-pf-to=pcqq.group

<https://www.cnblogs.com/jiawen010/p/11805241.html>

将数据存储与索引放到了一块，找到索引也就找到了数据

非聚簇索引:将数据存储于索引分开结构，索引结构的叶子节点指向了数据的对应行，myisam通过key\_buffer把索引先缓存到内存中，当需要访问数据时（通过索引访问数据），在内存中直接搜索索引，然后通过索引找到磁盘相应数据，这也就是为什么索引不在key buffer命中时，速度慢的原因.

聚簇索引保证关键字的值相近的元组存储的物理位置也相同（所以字符串类型不宜建立聚簇索引，特别是随机字符串，会使得系统进行大量的移动操作），且一个表只能有一个聚簇索引。

因为由存储引擎实现索引，所以，并不是所有的引擎都支持聚簇索引。

目前，只有solidDB和InnoDB支持。

### 覆盖索引

1）索引不会包含有null值得列

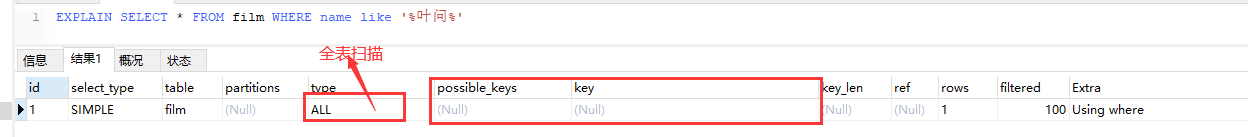
2）尽量使用短索引，对串列进行索引，如果可能应该指定一个前缀长度。短索引不仅可以提高查询速度而且可以节省磁盘空间和IO操作

3）like语句操作：like “%aaa%” 不会使用索引而like “aaa%”可以使用索引。

1. 不要在列上进行运算

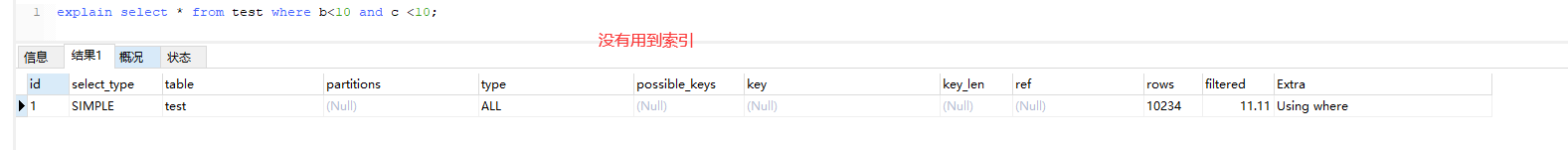
# 什么情况下索引会失效

1. like ‘%.....’，将%放到最前面，索引会失效.

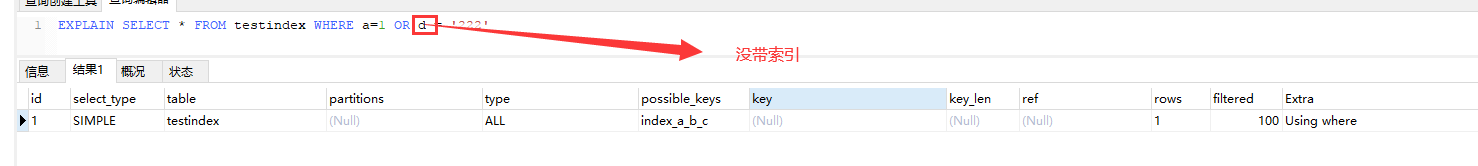


1. 不遵循最左原则

|  |
| --- |
| create table test( a int , b int, c int, d int, key index\_abc(a,b,c) )engine=InnoDB default charset=utf8; |
| DROP PROCEDURE IF EXISTS proc\_initData;  DELIMITER $  CREATE PROCEDURE proc\_initData()  BEGIN  DECLARE i INT DEFAULT 1;  WHILE i<=10000 DO  INSERT INTO test(a,b,c,d) VALUES(i,i,i,i);  SET i = i+1;  END WHILE;  END $  CALL proc\_initData(); |



1. 如果条件中有 or， 有条件没有使用索引，即使其中有条件带索引也不会使用，换言之， 就是要求使用的所有字段，都必须单独使用时能使用索引。



(4)如果列类型是字符串，那么一定要在条件中使用引号引用起来，否则不使用索引。



1. 如果mysql认为全表扫面要比使用索引快，则不使用索引。
2. 不要再索引列上做运算。

# 十大优化策略

1. 尽量全值匹配
2. 最左前缀法则
3. 不再索引列上做任何操作
4. 范围条件放最后
5. 覆盖索引尽量用
6. 不等于要慎用
7. Null/not有影响
8. Like查询要当心
9. 字符类型要加引号
10. OR该Union效率更高

# 一个6亿的表a，一个3亿的表b，通过外间tid关联，你如何最快的查询出满足条件的第50000到第50200中的这200条数据记录。

1、如果A表TID是自增长,并且是连续的,B表的ID为索引

select \* from a,b where a.tid = b.id and a.tid>500000 limit 200;

2、如果A表的TID不是连续的,那么就需要使用覆盖索引.TID要么是主键,要么是辅助索引,B表ID也需要有索引。

select \* from b , (select tid from a limit 50000,200) a where b.id = a .tid;

# 左连接，右连接，内连接，笛卡尔积

内连接查出相同的数据

## 左外连接

左外连接是A表的所有行匹配上B表得出的结果集。

## 右外连接

右外连接是B表的所有行匹配上A表得出的结果集。

## 一个表的主键为自增主键，现在共有10条数据，最大的id为10，删掉一条，在增加一条，此时的id为多少

如果索引是存到内存中，那么就是10，如果是存到磁盘中的，那么就是11。

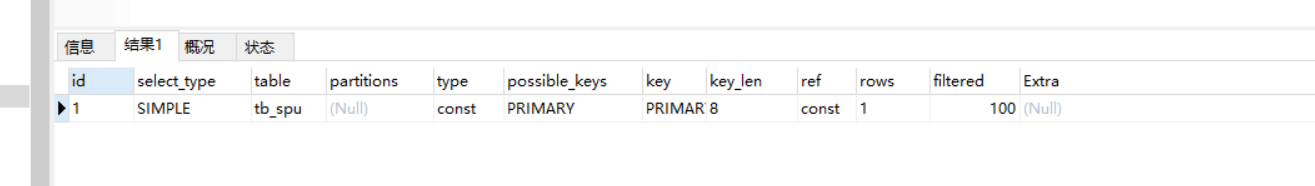
# 设计表

1. 增加适当的冗余字段
2. 一个表的字段数量不要太多
3. 尽量将比较长的字段放到另一个表中
4. 字符串尽量使用varchar而不是char

# 执行计划

使用的数据库为orange；

|  |
| --- |
| EXPLAIN  SELECT \* FROM tb\_spu WHERE tb\_spu.id = 2; |



## Id

ID 列：描述 select 查询的序列号,包含一组数字，表示查询中执行 select 子句或操作表的顺

序

根据 ID 的数值结果可以分成一下三种情况

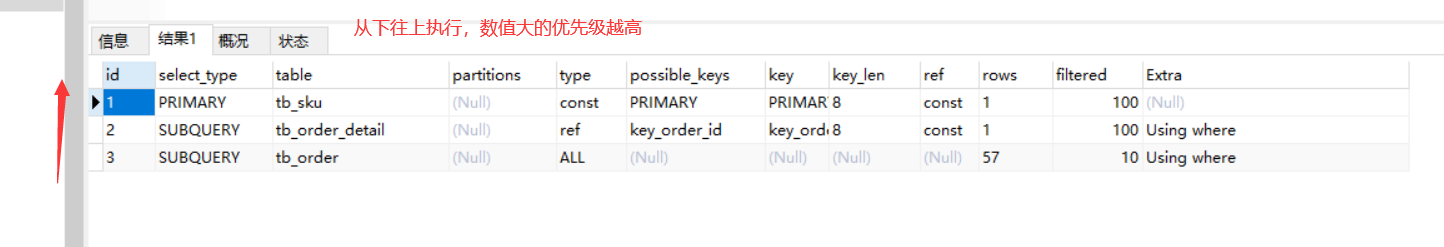
 id 相同：执行顺序由上至下

 id 不同：如果是子查询，id 的序号会递增，id 值越大优先级越高，越先被执行

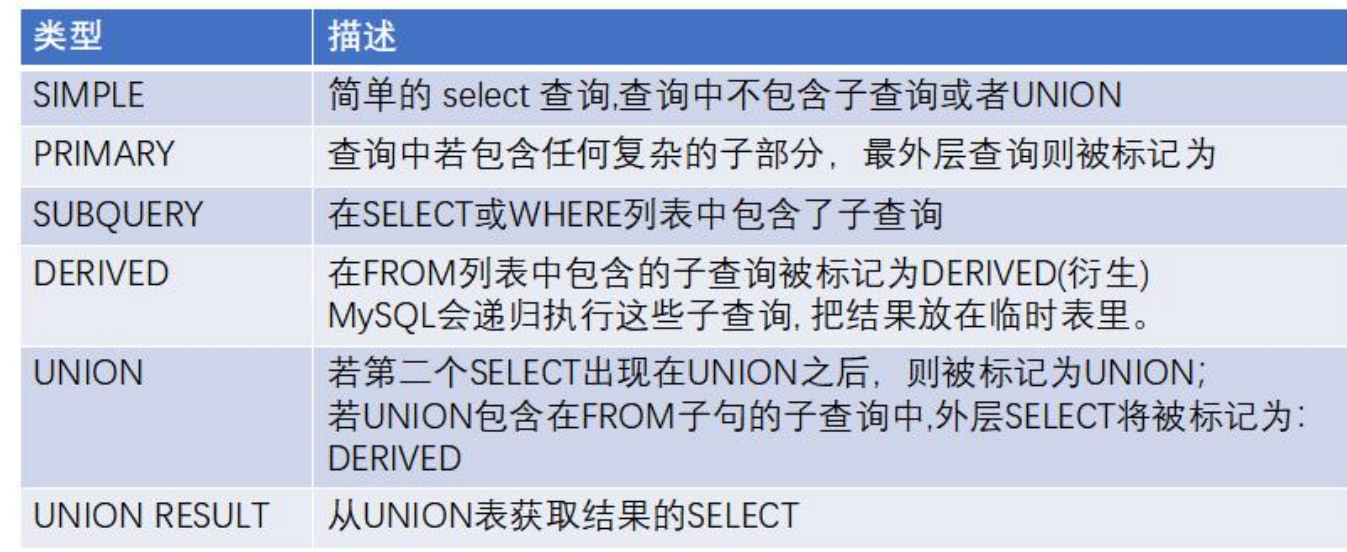
 id 相同不同：同时存在

### Id不相同

|  |
| --- |
| EXPLAIN  SELECT \* FROM tb\_sku WHERE tb\_sku.id = (  SELECT sku\_id FROM tb\_order\_detail WHERE tb\_order\_detail.order\_id  = (SELECT order\_id FROM tb\_order WHERE tb\_order.total\_pay = 132900)  ) |



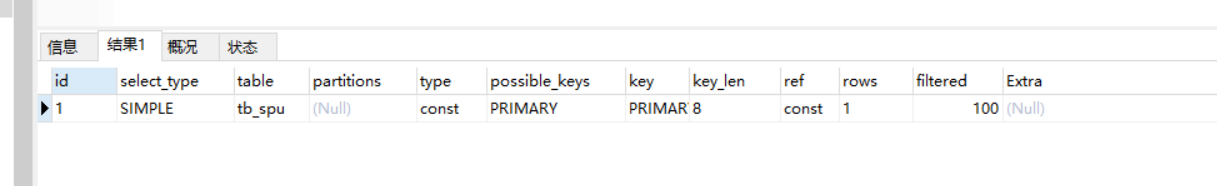
## select\_type



### SIMPLE

简单的 select 查询,查询中不包含子查询或者 UNION

|  |
| --- |
| EXPLAIN  SELECT \* FROM tb\_spu WHERE tb\_spu.id = 2; |

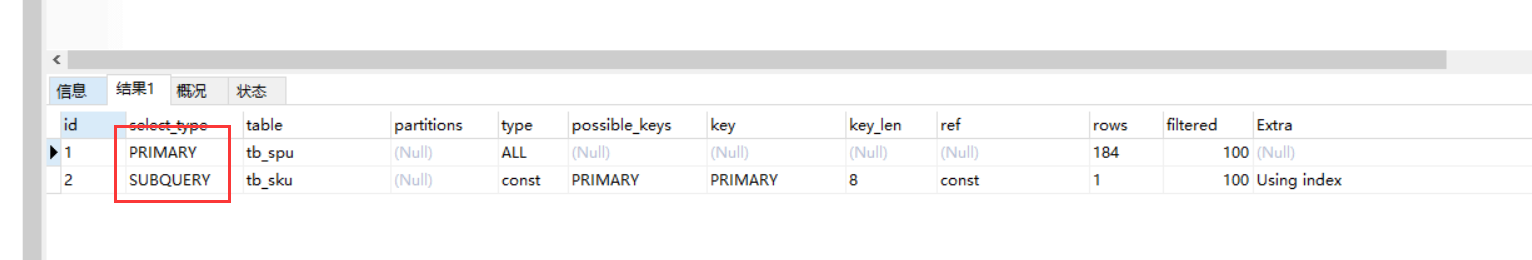


## **PRIMARY 与 SUBQUERY**

PRIMARY：查询中若包含任何复杂的子部分，最外层查询则被标记为 。

SUBQUERY：在 SELECT 或 WHERE 列表中包含了子查询。

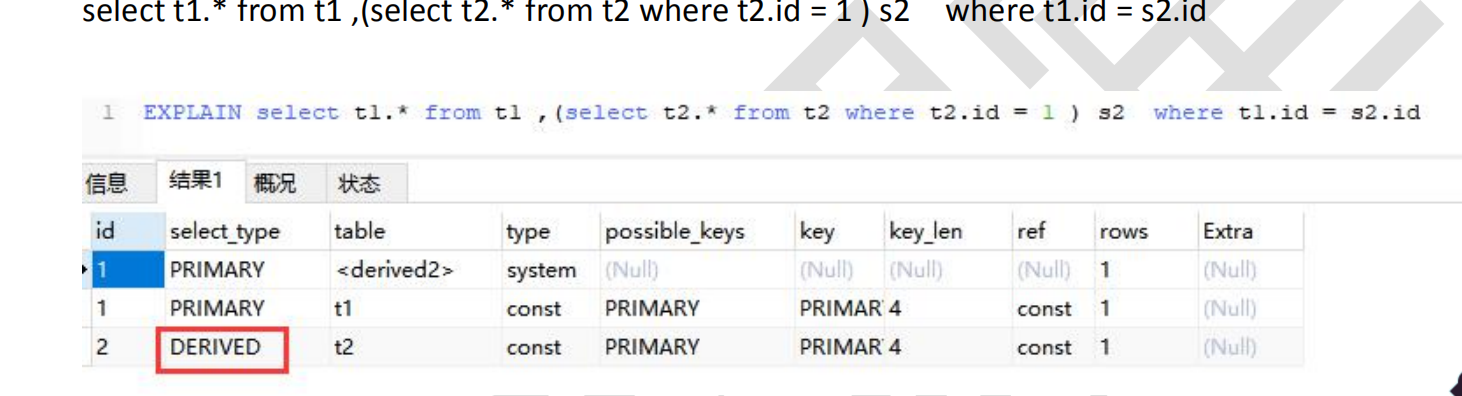
|  |
| --- |
| EXPLAIN  SELECT tb\_spu.\*,(SELECT tb\_sku.id FROM tb\_sku WHERE tb\_sku.id = 2600242)  FROM tb\_spu |



### DERIVED

在 FROM 列表中包含的子查询被标记为 DERIVED(衍生)

MySQL 会递归执行这些子查询, 把结果放在临时表里。

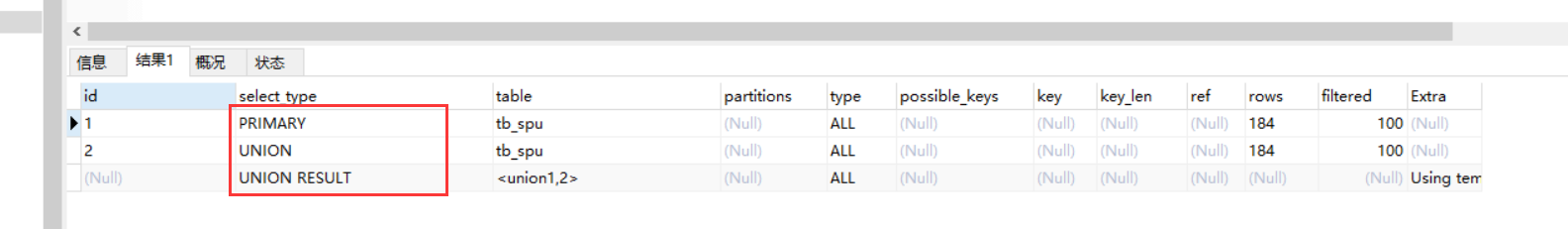


### UNION

UNION：若第二个 SELECT 出现在 UNION 之后，则被标记为 UNION；

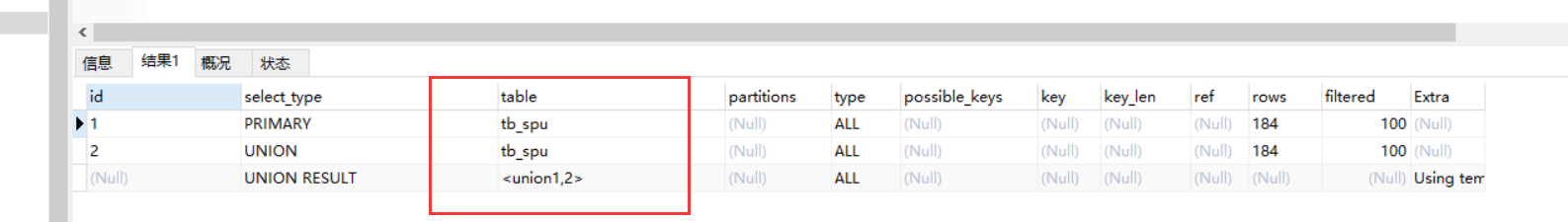
UNION RESULT：从 UNION 表获取结果的 SELECT。

|  |
| --- |
| EXPLAIN  SELECT tb\_spu.\* FROM tb\_spu  UNION  SELECT tb\_spu.\* FROM tb\_spu |



## Table

显示这一行的数据是关于哪张表的。



## **Type 列**

type 显示的是访问类型，是较为重要的一个指标，结果值从最好到最坏依次是：

system > const > eq\_ref > ref > fulltext > ref\_or\_null > index\_merge > unique\_subquery >

index\_subquery > range > index > ALL。

需要记忆的

system>const>eq\_ref>ref>range>index>ALL

一般来说，得保证查询至少达到 range 级别，最好能达到 ref。

### **System 与 const**

System：表只有一行记录（等于系统表），这是 const 类型的特列，平时不会出现，这个也

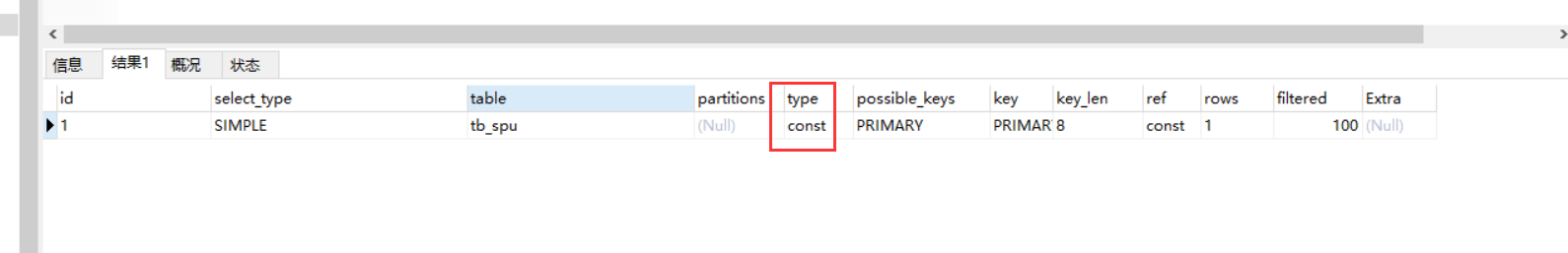
可以忽略不计

Const：表示通过索引一次就找到了

const 用于比较 primary key 或者 unique 索引。因为只匹配一行数据，所以很快

如将主键置于 where 列表中，MySQL 就能将该查询转换为一个常量。

|  |
| --- |
| EXPLAIN  SELECT \* FROM tb\_spu WHERE id = 2; |

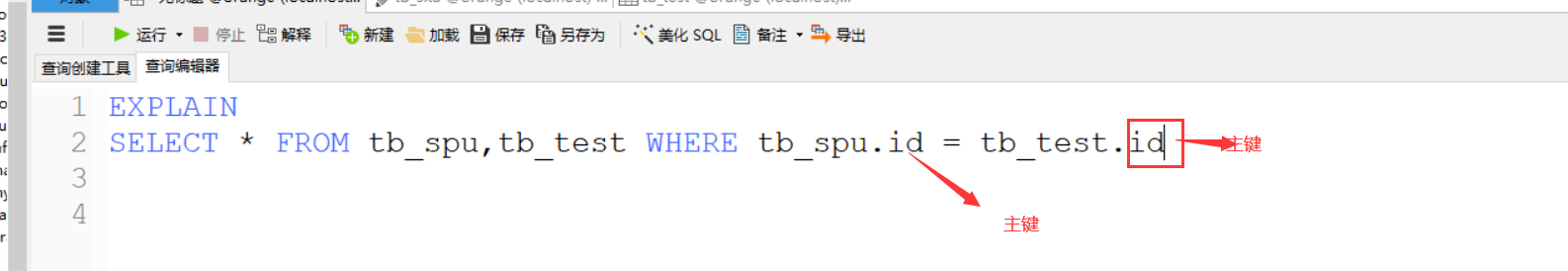


### eq\_ref

唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配。常见于主键或唯一索引

扫描。

多个表查询，一个表引用了另一个表的主键。





### Ref

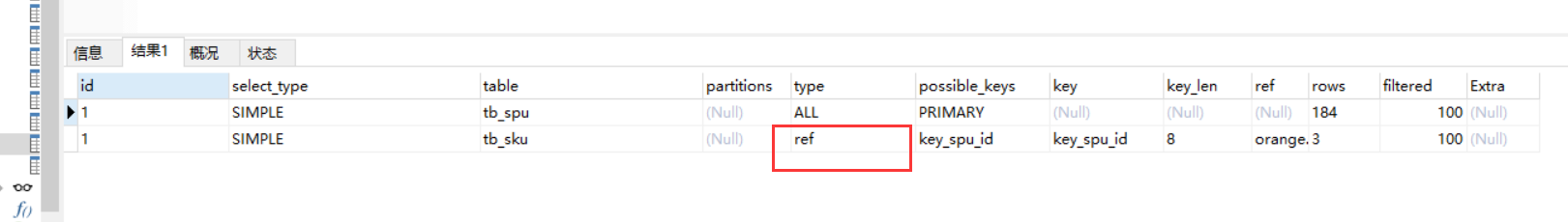
非唯一性索引扫描，返回匹配某个单独值的所有行.

本质上也是一种索引访问，它返回所有匹配某个单独值的行，然而，它可能会找到多个符合

条件的行，所以他应该属于查找和扫描的混合体。

假设有a，b两张表，b表引用了a表的id，但是并不是主键或者唯一性索引，用这个条件查询的type为ref。

|  |
| --- |
| EXPLAIN  SELECT \* FROM tb\_spu,tb\_sku WHERE tb\_spu.id = tb\_sku.spu\_id |



## Range

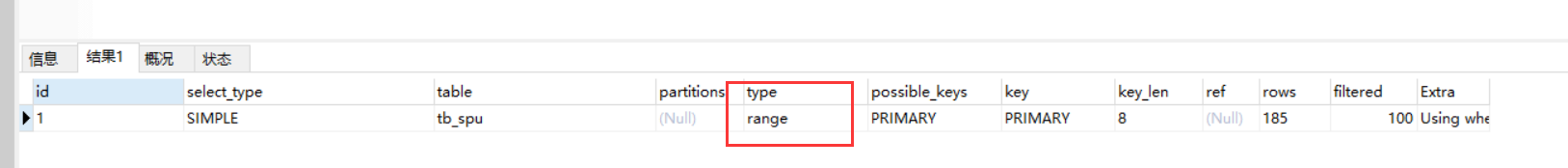
只检索给定范围的行,使用一个索引来选择行。key 列显示使用了哪个索引

一般就是在你的 where 语句中出现了 between、<、>、in 等的查询

这种范围扫描索引扫描比全表扫描要好，因为它只需要开始于索引的某一点，而结束语另一

点，不用扫描全部索引。

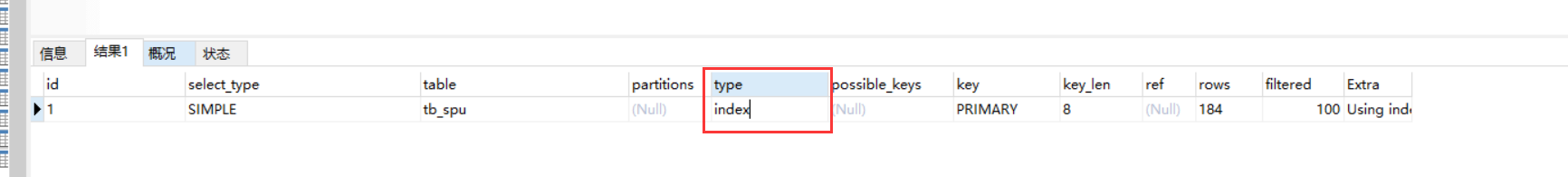
|  |
| --- |
| EXPLAIN  SELECT \* FROM tb\_spu WHERE tb\_spu.id > 1; |



### Index

当查询的结果全为索引列的时候，虽然也是全部扫描，但是只查询的索引库，而没有去查询

数据。



### All

Full Table Scan，将遍历全表以找到匹配的行。

|  |
| --- |
| EXPLAIN  SELECT title FROM tb\_spu |



## **possible\_keys 与 Key**

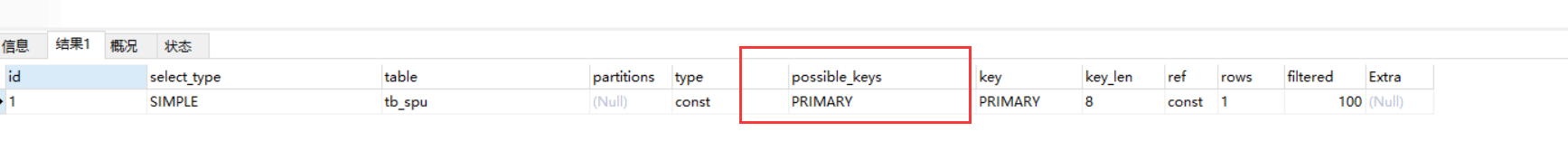
possible\_keys:可能使用的 key

Key:实际使用的索引。如果为 NULL，则没有使用索引

查询中若使用了**覆盖索引**，则该索引和查询的 select 字段重叠

### possible\_keys

|  |
| --- |
| EXPLAIN  SELECT title FROM tb\_spu WHERE id = 2; |



### key\_len

Key\_len 表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度。在不损失精

确性的情况下，长度越短越好

key\_len 显示的值为索引字段的最大可能长度，并非实际使用长度，即 key\_len 是根据表定义

计算而得，不是通过表内检索出的

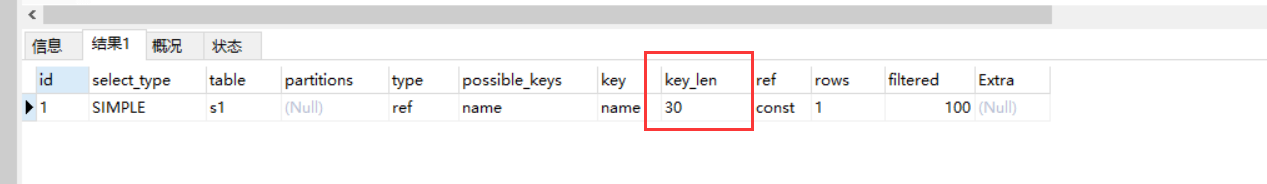
* key\_len 表示索引使用的字节数， 
* 根据这个值，就可以判断索引使用情况，特别是在组合索引的时候，判断所有的索引字 段是否都被查询用到。
* char 和 varchar 跟字符编码也有密切的联系,
* latin1 占用 1 个字节，gbk 占用 2 个字节，utf8 占用 3 个字节。（不同字符编码占用的**存储空间不同）。**

### 字符类型



### **字符类型-索引字段为 char 类型+不可为 Null 时**

|  |
| --- |
| CREATE TABLE `s1` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `name` char(10) NOT NULL,  `addr` varchar(20) DEFAULT NULL,  PRIMARY KEY (`id`),  KEY `name` (`name`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8;  explain select \* from s1 where name='enjoy'; |



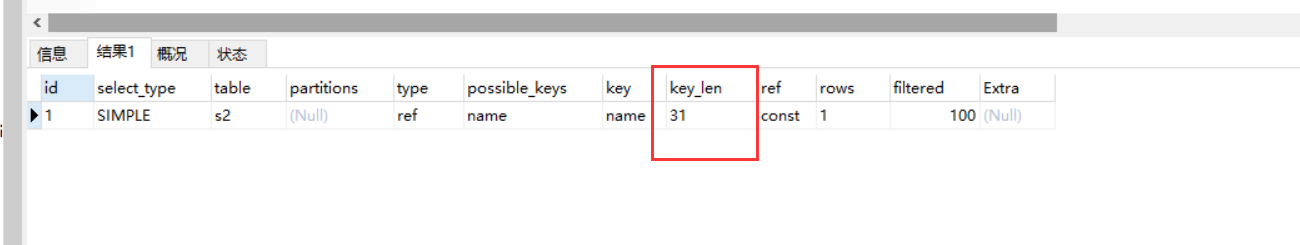
name 这一列为 char(10),字符集为 utf-8 占用 3 个字节

Keylen=10\*3

### **字符类型-索引字段为 char 类型+允许为 Null 时**

|  |
| --- |
| CREATE TABLE `s2` (  `id` INT (11) NOT NULL AUTO\_INCREMENT,  `name` CHAR (10) DEFAULT NULL,  `addr` VARCHAR (20) DEFAULT NULL,  PRIMARY KEY (`id`),  KEY `name` (`name`)  ) ENGINE = INNODB DEFAULT CHARSET = utf8; |

|  |
| --- |
| explain select \* from s2 where name='enjoyedu'; |

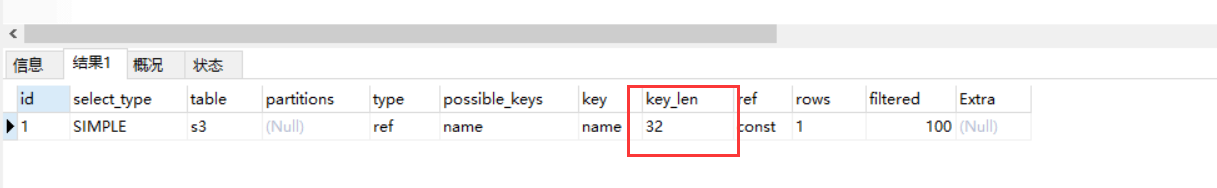


name 这一列为 char(10),字符集为 utf-8 占用 3 个字节,外加需要存入一个 null 值

Keylen=10\*3+1(null) 结果为 31。

### **索引字段为 varchar 类型+不可为 Null 时**

|  |
| --- |
| CREATE TABLE `s3` (  `id` INT (11) NOT NULL AUTO\_INCREMENT,  `name` VARCHAR (10) NOT NULL,  `addr` VARCHAR (20) DEFAULT NULL,  PRIMARY KEY (`id`),  KEY `name` (`name`)  ) ENGINE = INNODB DEFAULT CHARSET = utf8; |
| explain select \* from s3 where name='enjoyeud'; |



Keylen=varchar(n)变长字段+不允许 Null=n\*(utf8=3,gbk=2,latin1=1)+2

### **索引字段为 varchar 类型+允许为 Null 时**

|  |
| --- |
| CREATE TABLE `s4` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `name` varchar(10) default NULL,  `addr` varchar(20) DEFAULT NULL,  PRIMARY KEY (`id`),  KEY `name` (`name`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8; |

|  |
| --- |
| explain select \* from s4 where name='enjoyeud'; |

### 

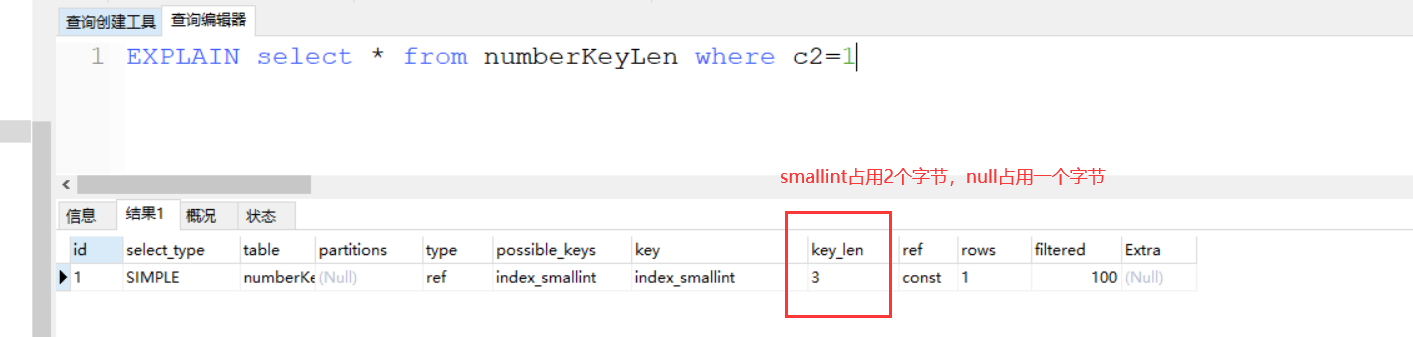
Keylen=varchar(n)变长字段+允许 Null=n\*(utf8=3,gbk=2,latin1=1)+1(NULL)+2

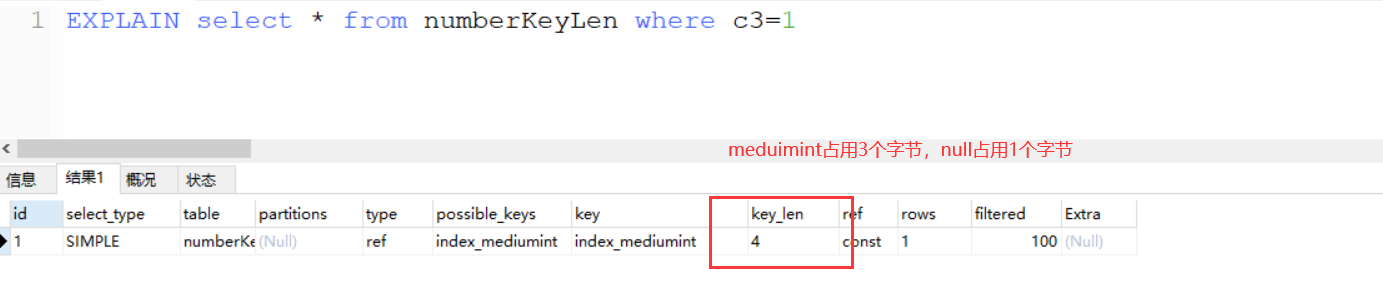
### 数值类型



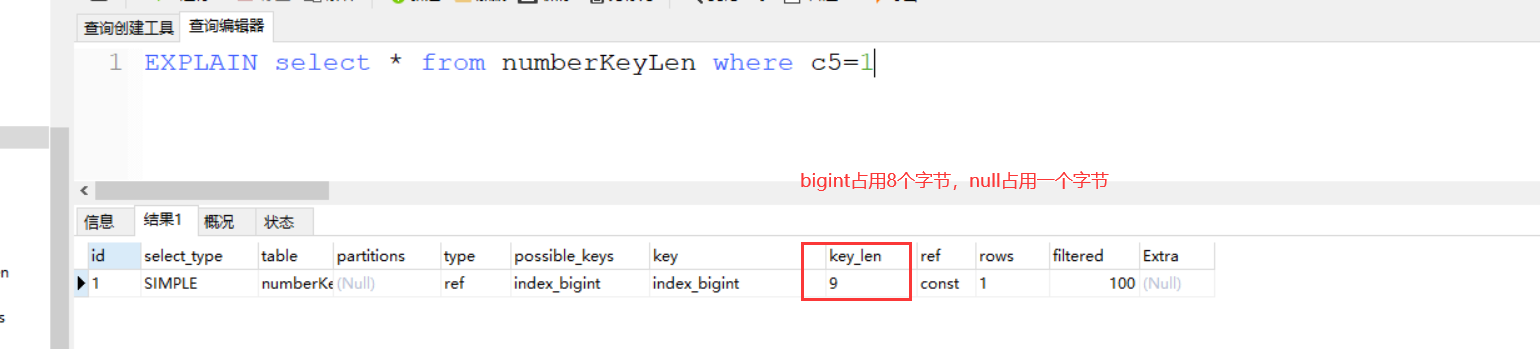
|  |
| --- |
| CREATE TABLE `numberKeyLen` (  `c0` INT (255) NOT NULL,  `c1` TINYINT (255) NULL DEFAULT NULL,  `c2` SMALLINT (255) NULL DEFAULT NULL,  `c3` MEDIUMINT (255) NULL DEFAULT NULL,  `c4` INT (255) NULL DEFAULT NULL,  `c5` BIGINT (255) NULL DEFAULT NULL,  `c6` FLOAT (255, 0) NULL DEFAULT NULL,  `c7` DOUBLE (255, 0) NULL DEFAULT NULL,  PRIMARY KEY (`c0`),  INDEX `index\_tinyint` (`c1`) USING BTREE,  INDEX `index\_smallint` (`c2`) USING BTREE,  INDEX `index\_mediumint` (`c3`) USING BTREE,  INDEX `index\_int` (`c4`) USING BTREE,  INDEX `index\_bigint` (`c5`) USING BTREE,  INDEX `index\_float` (`c6`) USING BTREE,  INDEX `index\_double` (`c7`) USING BTREE  ) ENGINE = INNODB DEFAULT CHARACTER  SET = utf8 COLLATE = utf8\_general\_ci ROW\_FORMAT = COMPACT; |

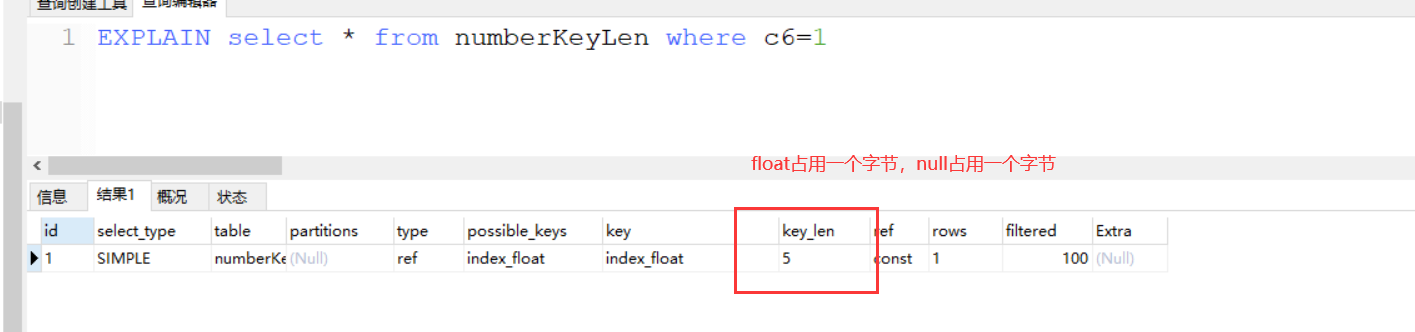


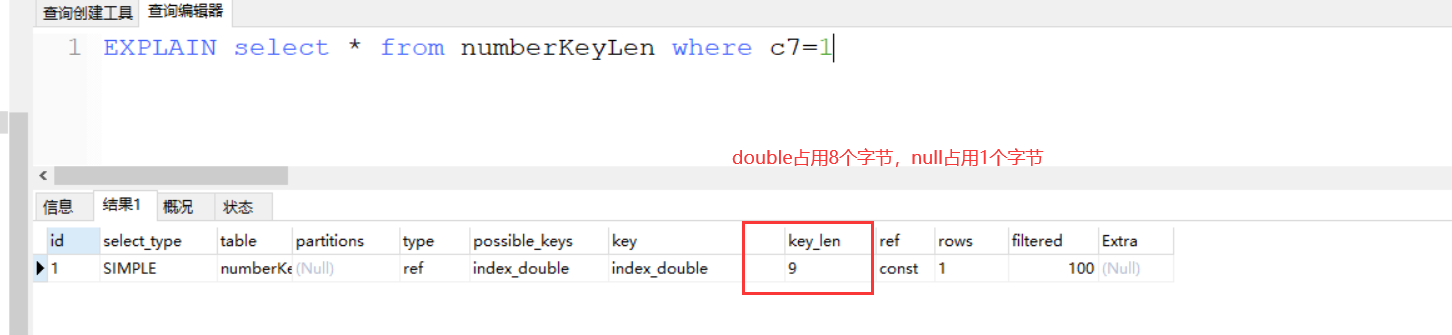












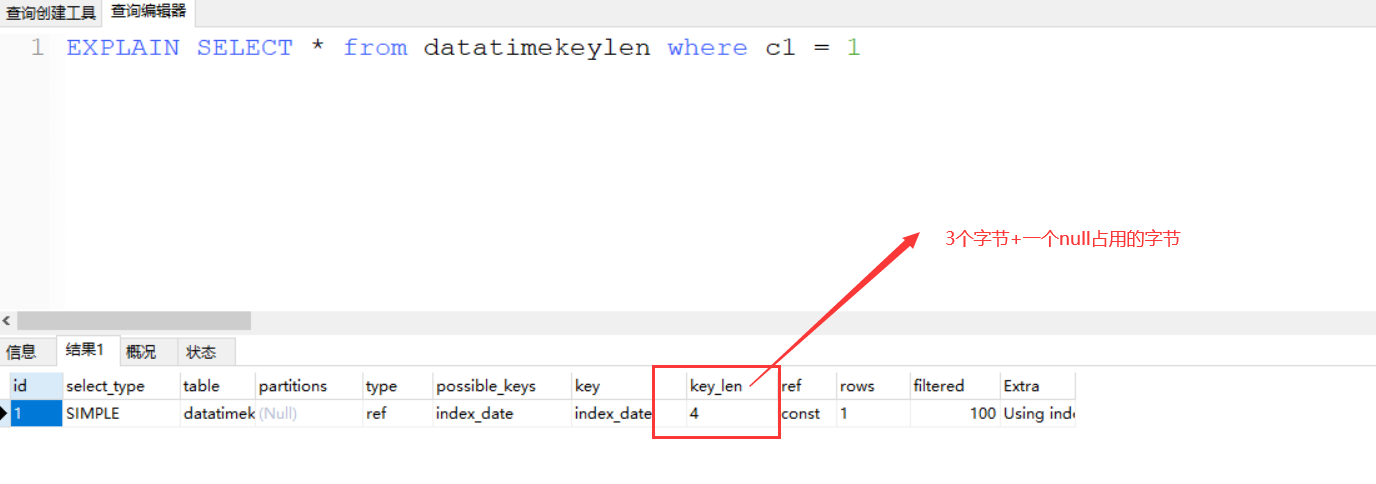
### 日期类型

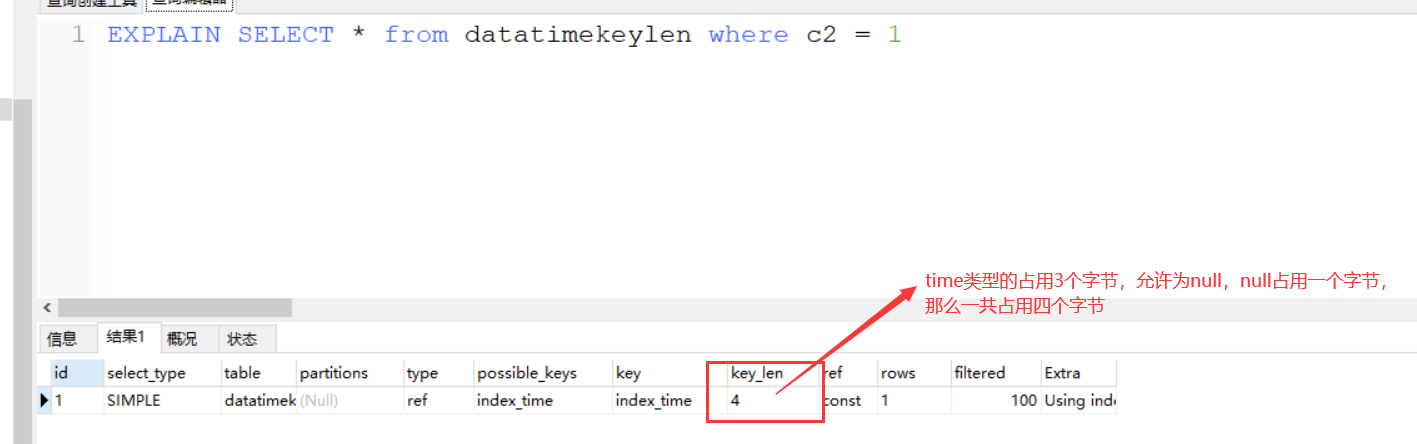


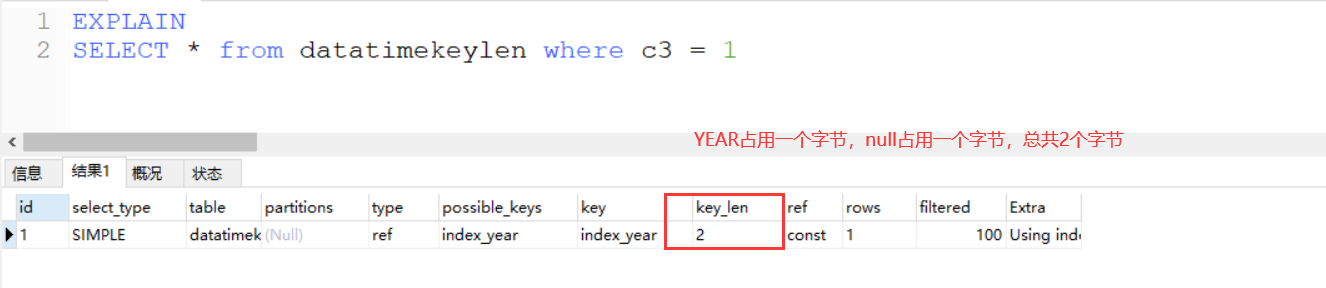
datetime 类型在 5.6 中字段长度是 5 个字节

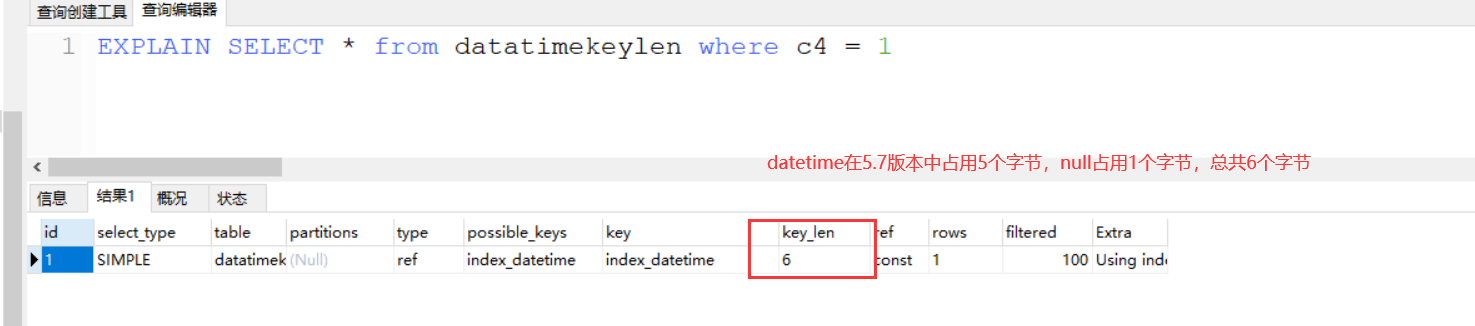
datetime 类型在 5.5 中字段长度是 8 个字节

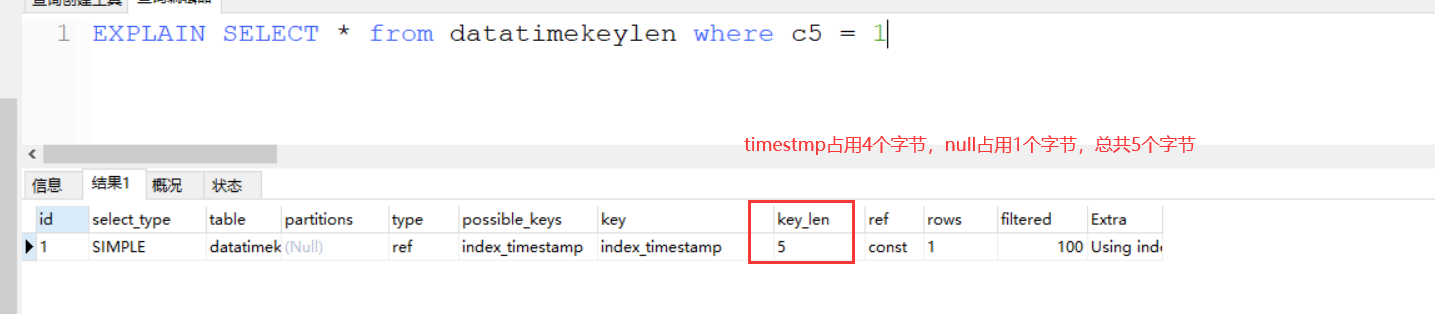
|  |
| --- |
| CREATE TABLE `datatimekeylen` (  `c1` date NULL DEFAULT NULL,  `c2` time NULL DEFAULT NULL,  `c3` YEAR NULL DEFAULT NULL,  `c4` datetime NULL DEFAULT NULL,  `c5` TIMESTAMP NULL DEFAULT NULL,  INDEX `index\_date` (`c1`) USING BTREE,  INDEX `index\_time` (`c2`) USING BTREE,  INDEX `index\_year` (`c3`) USING BTREE,  INDEX `index\_datetime` (`c4`) USING BTREE,  INDEX `index\_timestamp` (`c5`) USING BTREE  ) ENGINE = INNODB DEFAULT CHARACTER  SET = utf8 COLLATE = utf8\_general\_ci ROW\_FORMAT = COMPACT; |











### 总结

#### 字符类型

变长字段需要额外的 2 个字节（VARCHAR 值保存时只保存需要的字符数，另加一个字节来

记录长度(如果列声明的长度超过 255，则使用两个字节)，所以 VARCAHR 索引长度计算时候

要加 2），固定长度字段不需要额外的字节。

而 NULL 都需要 1 个字节的额外空间,所以索引字段最好不要为 NULL，因为 NULL 让统计更加

复杂并且需要额外的存储空间。

复合索引有最左前缀的特性，如果复合索引能全部使用上，则是复合索引字段的索引长度之

和，这也可以用来判定复合索引是否部分使用，还是全部使用。

#### **整数/浮点数/时间类型的索引长度**

NOT NULL=字段本身的字段长度

NULL=字段本身的字段长度+1(因为需要有是否为空的标记，这个标记需要占用 1

个字节)

## Ref

显示索引的哪一列被使用了，如果可能的话，是一个常数。哪些列或常量被用于查找索引列

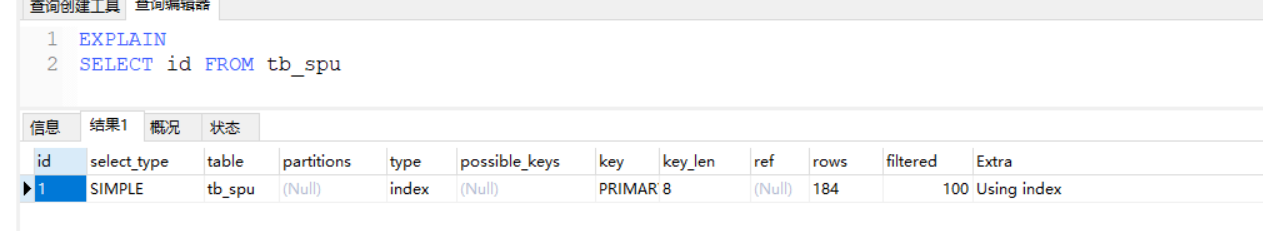
上的值

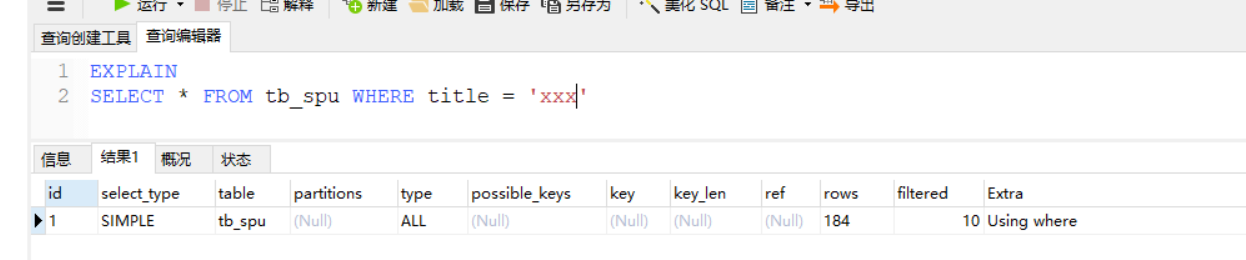
## Rows

## Extra

包含不适合在其他列中显示但十分重要的额外信息。







# 覆盖索引

覆盖索引（Covering Index）,一说为索引覆盖。

理解方式一:就是 select 的数据列只用从索引中就能够取得，不必读取数据行，MySQL 可以

利用索引返回 select 列表中的字段，而不必根据索引再次读取数据文件,换句话说查询列要

被所建的索引覆盖。

理解方式二:索引是高效找到行的一个方法，但是一般数据库也能使用索引找到一个列的数

据，因此它不必读取整个行。毕竟索引叶子节点存储了它们索引的数据;当能通过读取索引

就可以得到想要的数据，那就不需要读取行了。一个索引包含了(或覆盖了)满足查询结果的

数据就叫做覆盖索引

注意：

如果要使用覆盖索引，一定要注意 select 列表中只取出需要的列，不可 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。

所以，千万不能为了查询而在所有列上都建立索引，会严重影响修改维护的性能。