## 笔者mysql的版本：5.6.30

## 1.0 范式

数据库表的设计规范

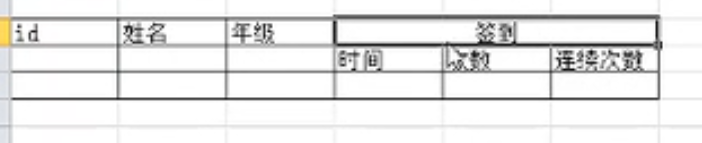
**范式：1NF 2NF 3NF**

第一范式，第二范式，第三范式

如果需要满足第N范式，需要先满足第N-1范式。

## 第一范式，字段具有原子性

符合第一范式时，每一个字段都是不可再拆分的，都是最小的，不包含其它字段，有原子性、不重复的特点。下面的就不符合第一范式



所有的关系型数据库，默认满足第一范式。

**逻辑上的含义：**

|  |  |
| --- | --- |
| Id | 授课时间（开始-结束） |
| 3 | 2016-03，2019-03 |

应该这样设计

|  |  |  |
| --- | --- | --- |
| Id | 授课时间（开始） | 授课时间（结束） |
| 3 | 2016-03 | 2019-03 |

## 第二范式，消除对主键部分依赖

主键：可以唯一标识 记录的字段或者字段集合

下面的表的主键应该是：老师+班级 形成的复合主键

|  |  |  |  |
| --- | --- | --- | --- |
| 老师 | 性别 | 班级 | 教室 |
| 孔子 | 男 | 102 | 102 |
| 李清照 | 女 | 102 | 102 |

依赖： A字段可以确定B字段，则B字段依赖A字段。

如 由老师可以确定性别，则好性别依赖于老师

对主键的部分依赖：如果某个字段依赖复合主键的一部分字段，则称之为对主键的部分依赖！ 例如，性别对主键就是部分依赖。

第二范式要求消除对主键的部分依赖。

怎样解决这个问题？。

增加一个独立字段即可，如增加一个id字段

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Id | 老师 | 性别 | 班级 | 教室 |
| 102 | 孔子 | 男 | 102 | 102 |
| 103 | 李清照 | 女 | 102 | 102 |

当id作为主键，对主键的部分依赖就不存在。

## 第二范式，消除对主键的传递依赖

传递依赖：

C依赖于B, B依赖于主键，则称C对主键存在传递依赖。

上图当中的：

性别C，讲师B，ID主键，性别对id主键，就存在传递依赖。

解决方案：

将独立数据单独建表，使用关联字段进行存储。如下

讲师表：

|  |  |  |
| --- | --- | --- |
| Id | 老师 | 性别 |

代课表：

|  |  |  |
| --- | --- | --- |
| Id | 讲师id | 其它字段 |

## 对三大范式的总结

#### 满足第三范式应该这样做：

1. 独立数据建表

2 在表中设计与业务逻辑无关的主键

1. 表之间的关系由关联字段进行表示

#### 好处：

1 减少数据的冗余

2 更好的维护

## 存储引擎

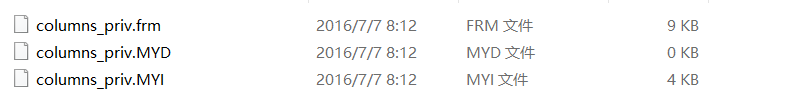
#### 存储文件的差异：

Innodb



但是老师演示的时候是这样的：

Mysiam



#### 具体的差别：

InnoDB引擎优点

1、支持事务处理、ACID事务特性；

2、实现了SQL标准的四种隔离级别；

3、支持行级锁和外键约束；

4、可以利用事务日志进行数据恢复。

5、锁级别为行锁，行锁优点是适用于高并发的频繁表修改，高并发是性能优于 MyISAM。缺点是系统消耗较大。

6、索引不仅缓存自身，也缓存数据，相比 MyISAM 需要更大的内存。

InnoDB引擎缺点

1因为它没有保存表的行数，当使用COUNT统计时会扫描全表。

2不支持表级的拷贝

MyISAM引擎

MyISAM 是 MySQL 5.5.5 之前的默认引擎，它的设计目标是快速读取。

MyISAM引擎优点

1.高性能读取；

2.因为它保存了表的行数，当使用COUNT统计时不会扫描全表；

3 支持表级的拷贝

MyISAM引擎缺点

1、锁级别为表锁，表锁优点是开销小，加锁快；缺点是锁粒度大，发生锁冲动概率较高，容纳并发能力低，这个引擎适合查询为主的业务。

2、此引擎不支持事务，也不支持外键。

3、INSERT和UPDATE操作需要锁定整个表；

4、它存储表的行数，于是SELECT COUNT(\*) FROM TABLE时只需要直接读取已经保存好的值而不需要进行全表扫描。



记录存储顺序：

inndb 按照主顺序存储 ，插入数据的时候要进行排序

myisam按照插入顺序进行存储

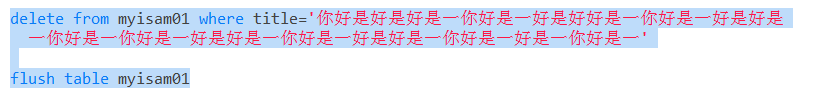
空间碎片：

inndb 按照主顺序存储 ，插入数据的时候要进行排序

myisam按照插入顺序进行存储

原始数据：



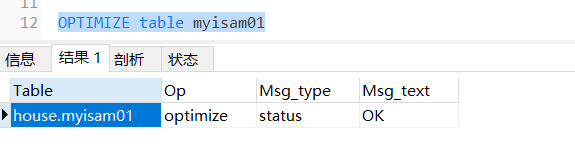


数据还是这样的：



这是由于虽然我们删除了一部分数据，但是空间还在占用着，这就是空洞和碎片的问题。

要想解决这样的问题：就是对myisam表进行定时的优化



下面就会看到数据文件减少了



#### 结论

Myisam 以读写插入为主的应用程序，例如，博客新闻发布，门户

Innodb: 更新担任为主要操作，或者要求数据的完整性较强，并必性好，支持事务和外键 保证数据完整性，例如OA系统。

## mysql锁的概念

#### 锁的类型

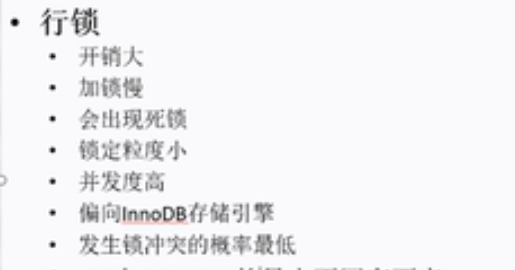
共享锁，读锁，S-lock

需要要在执行读操作时，操作的锁定类型，会共享读操作，其它任务也可以插读操作，阻塞写操作，但是不能进行写操作。

排他锁，独占锁，写锁，X-lock

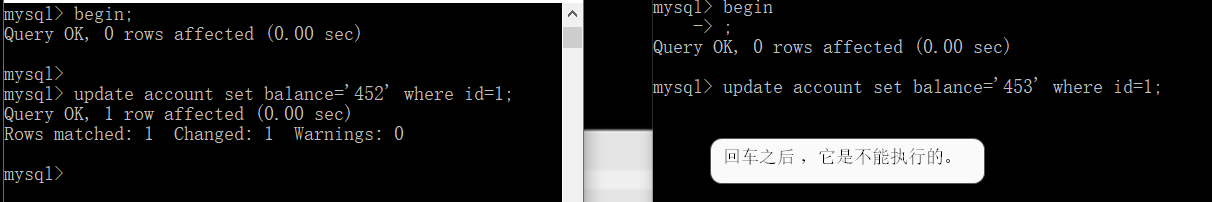
需要在执行写操作时增加的锁定类型，会独占该资源，其它任何既不能读也不能写

Mysql 在执行任何的sql操作的时候，都会自动增加锁定，通常不需要认为操作锁。

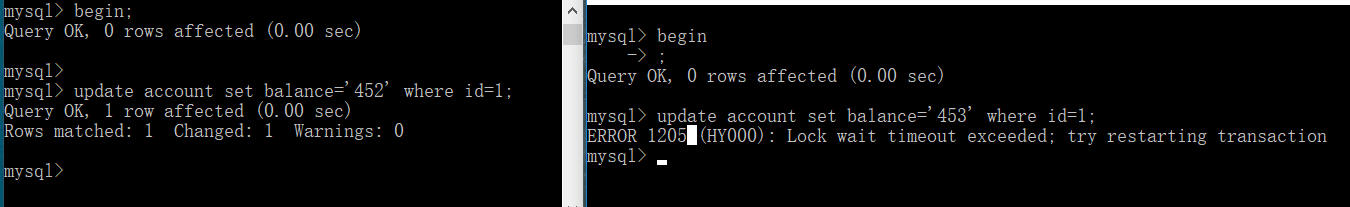


读锁与写锁都属于悲观锁

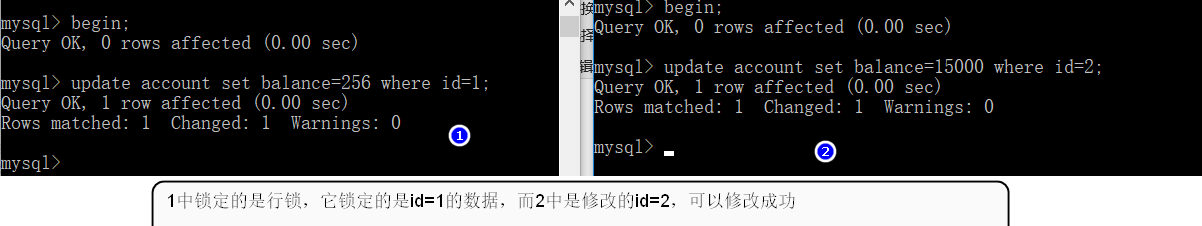
#### 行锁锁定的演示：



过了一段时间之后，就会报错了。如下图



再进行一个操作。



#### ****事务****



##### 一 数据库默认使用的隔离级别：

REPEATABLE-READ

##### 二 脏读 不可重复读 幻读

**1、脏读：事务A读取了事务B更新的数据，然后B回滚操作，那么A读取到的数据是脏数据**

**2、不可重复读：事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果 不一致。**

**3、幻读：系统管理员A将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。**



**小结：不可重复读的和幻读很容易混淆，不可重复读侧重于修改，幻读侧重于新增或删除。解决不可重复读的问题只需锁住满足条件的行，解决幻读需要锁表**

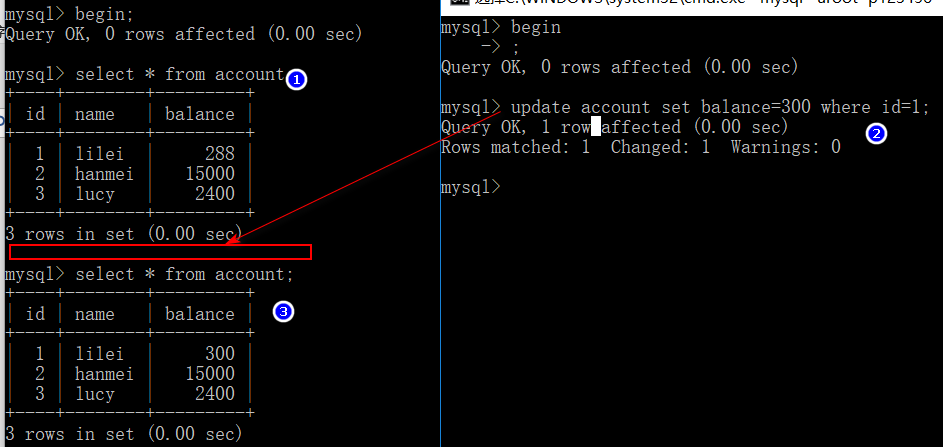


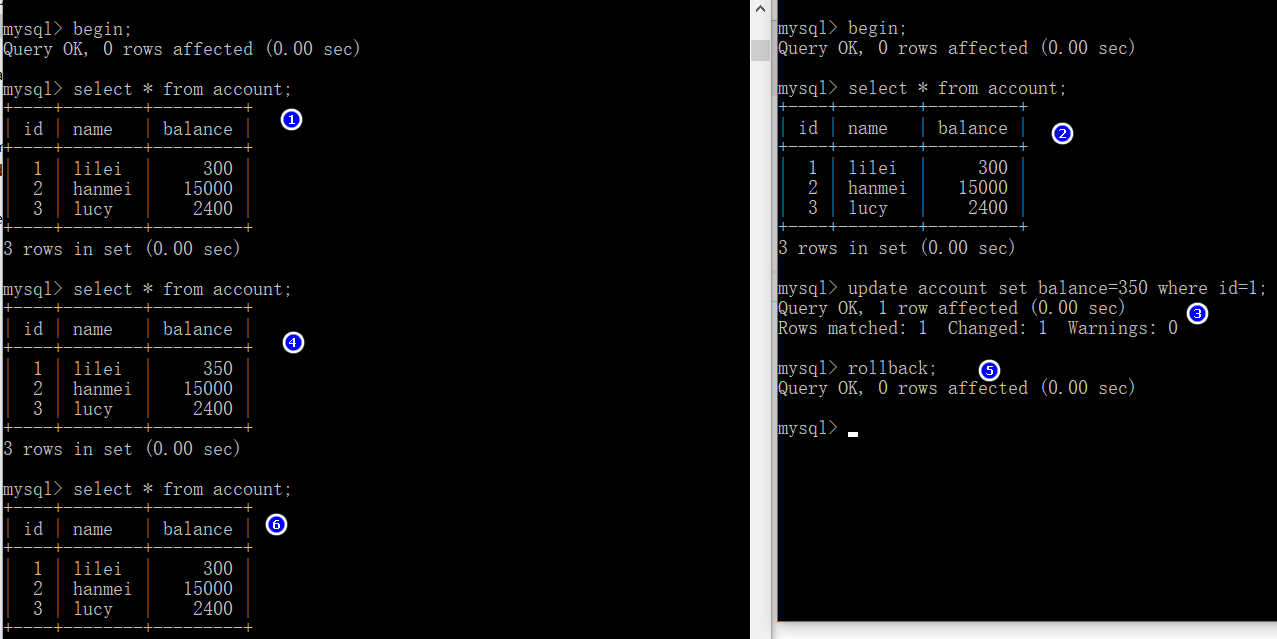
##### 三 隔离级别设置为读未提交—并出现脏读的情况

1 首先设定隔离级别：

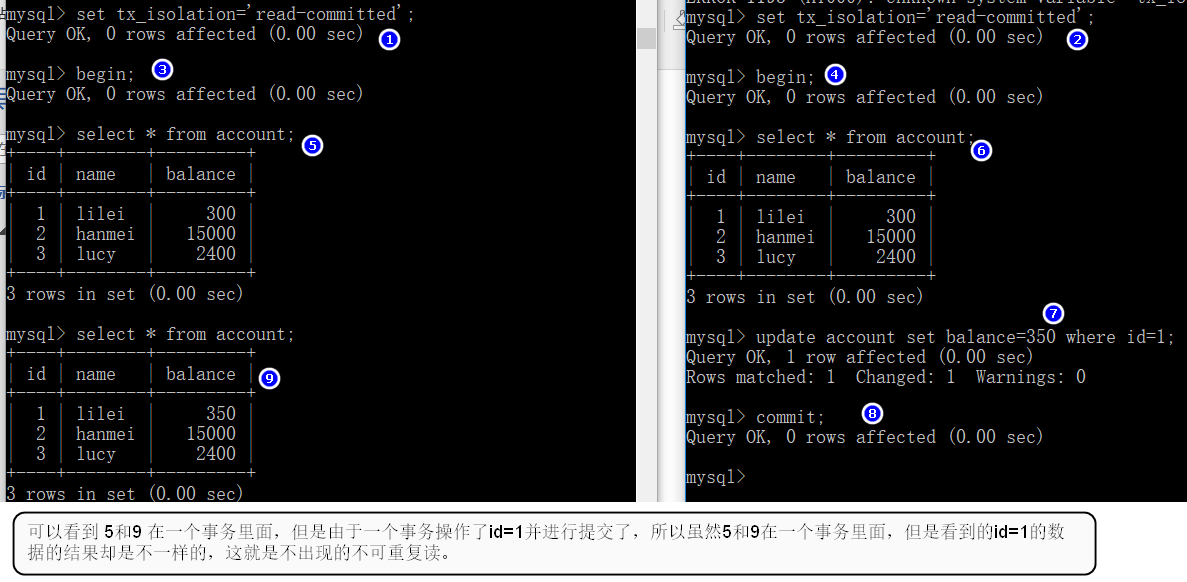
set tx\_isolation='read-uncommitted';

2 进行操作



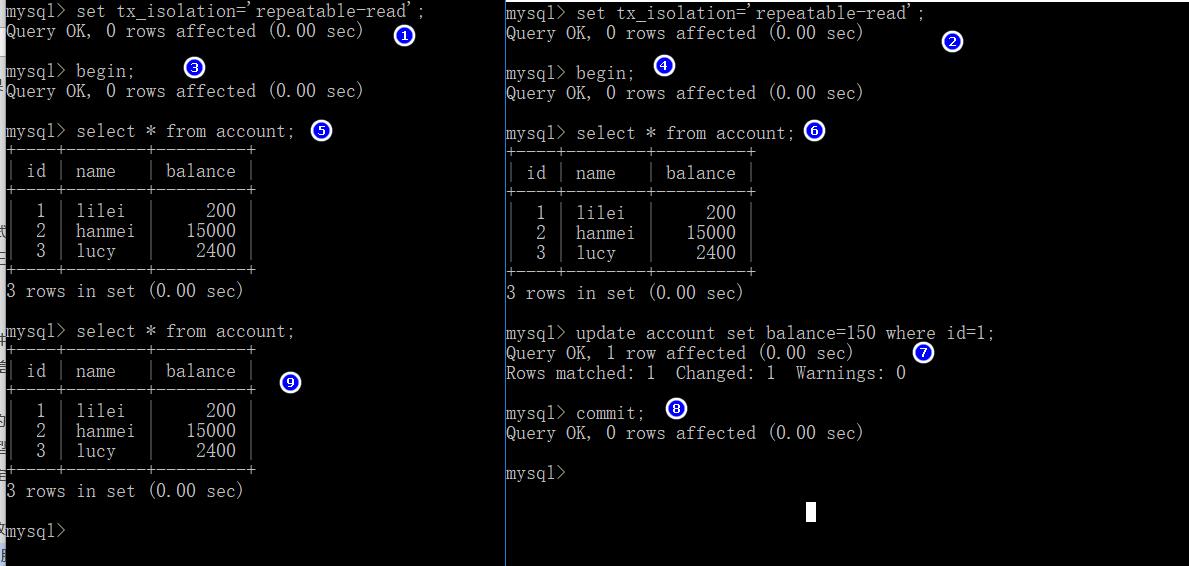


##### 四 隔离级别设置为已提交—并出现不可重复读的情况



##### 五 隔离级别设置为可重复读

###### 1 可以避免不可重复读的问题，如下图



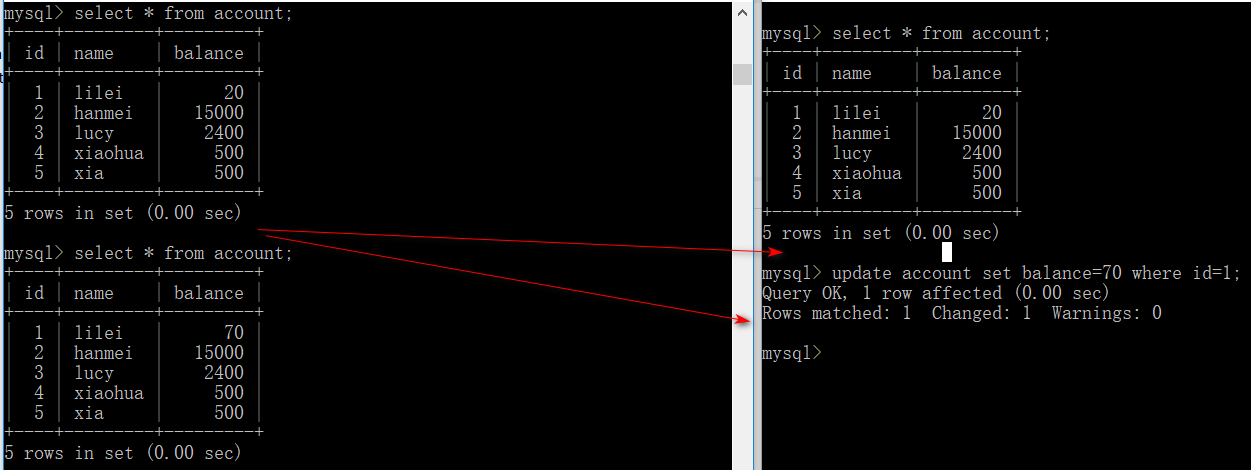
###### 但是不可以解决幻读的问题



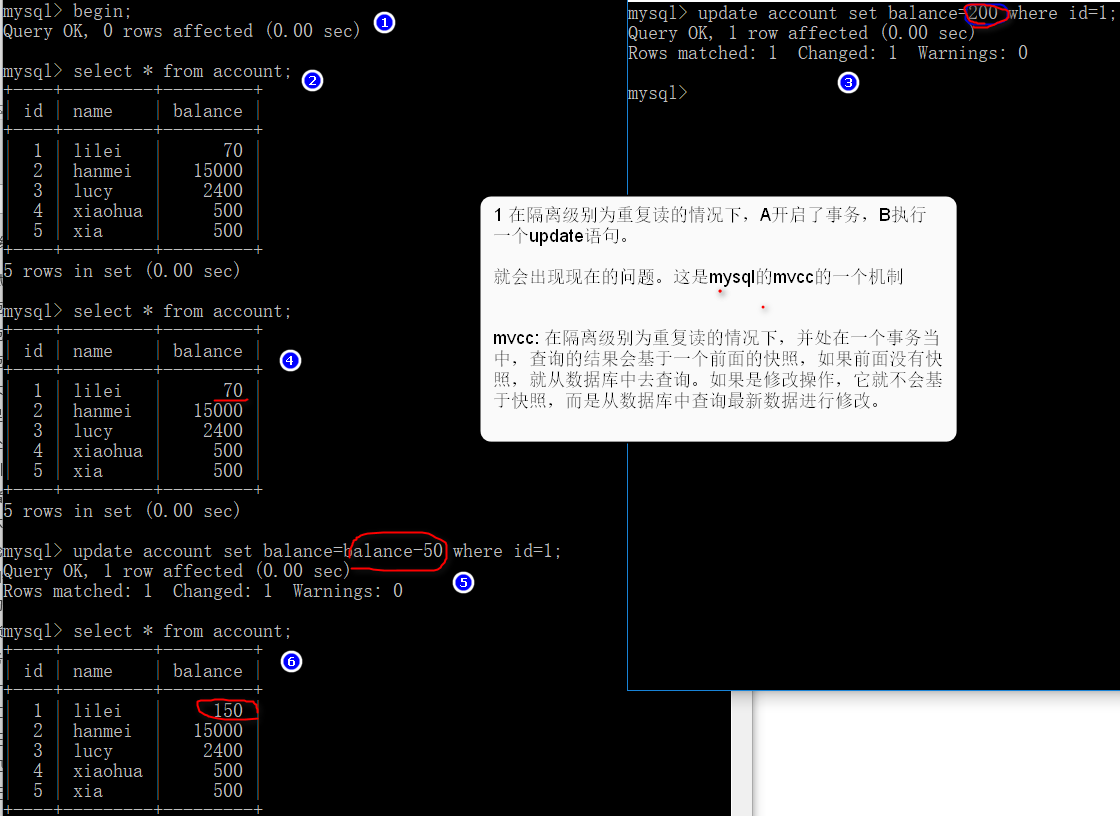
##### 六 数据库设置可重复读带来的一些问题

下面操作的前提是隔离级别是：可重复读。

A和B都没有开户事务



A开启了事务，但是B没有开启了事务



##### 七 锁的粒度

###### 表锁(innodb,myisam都可以用)

在mysql 中锁被实现了不同的粒度

一旦加锁，锁定的记录数量不同

表级锁，table-level 操作会锁定整张表，无论是共享锁，还是独占锁

Innodb同时实现了表级锁和行级锁

表锁的实现语法

|  |
| --- |
| LOCK TABLES  *tbl\_name* [AS *alias*] {READ [LOCAL] | [LOW\_PRIORITY] WRITE}      [, *tbl\_name* [AS *alias*] {READ [LOCAL] | [LOW\_PRIORITY] WRITE}] ...  UNLOCK TABLES  LOCK TABLES可以锁定用于当前线程的表。如果表被其它线程锁定，则造成堵塞，直到可以获取所有锁定为止。UNLOCK TABLES可以释放被当前线程保持的任何锁定。当线程发布另一个LOCK TABLES时，或当与服务器的连接被关闭时，所有由当前线程锁定的表被隐含地解锁。    表锁定只用于防止其它客户端进行不正当地读取和写入。保持锁定（即使是读取锁定）的客户端可以进行表层级的操作，比如DROP TABLE。 |

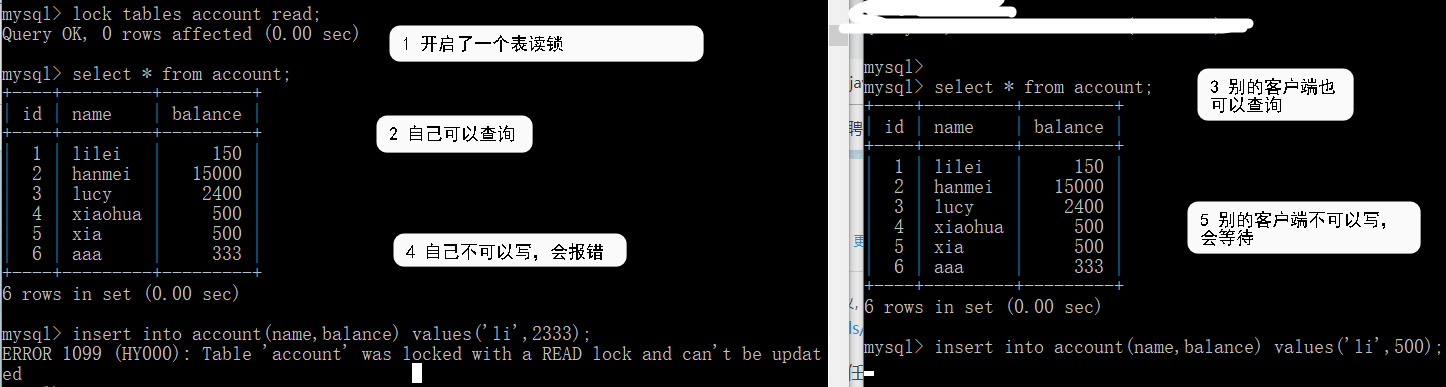
加锁：

Lock tables table\_name1,table\_name2 read|wirte

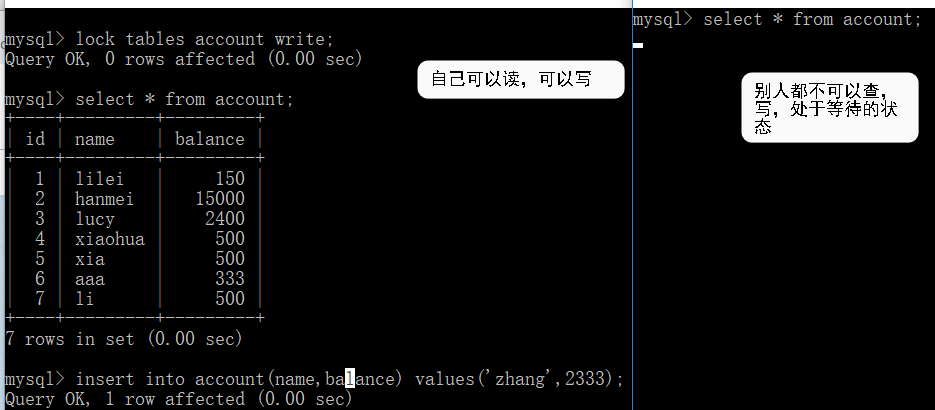
解锁：

Unlock tables;

表读锁的演示：（都可以读，都不可以写）



表写锁的演示（自己可以读写，别人不可以读写）



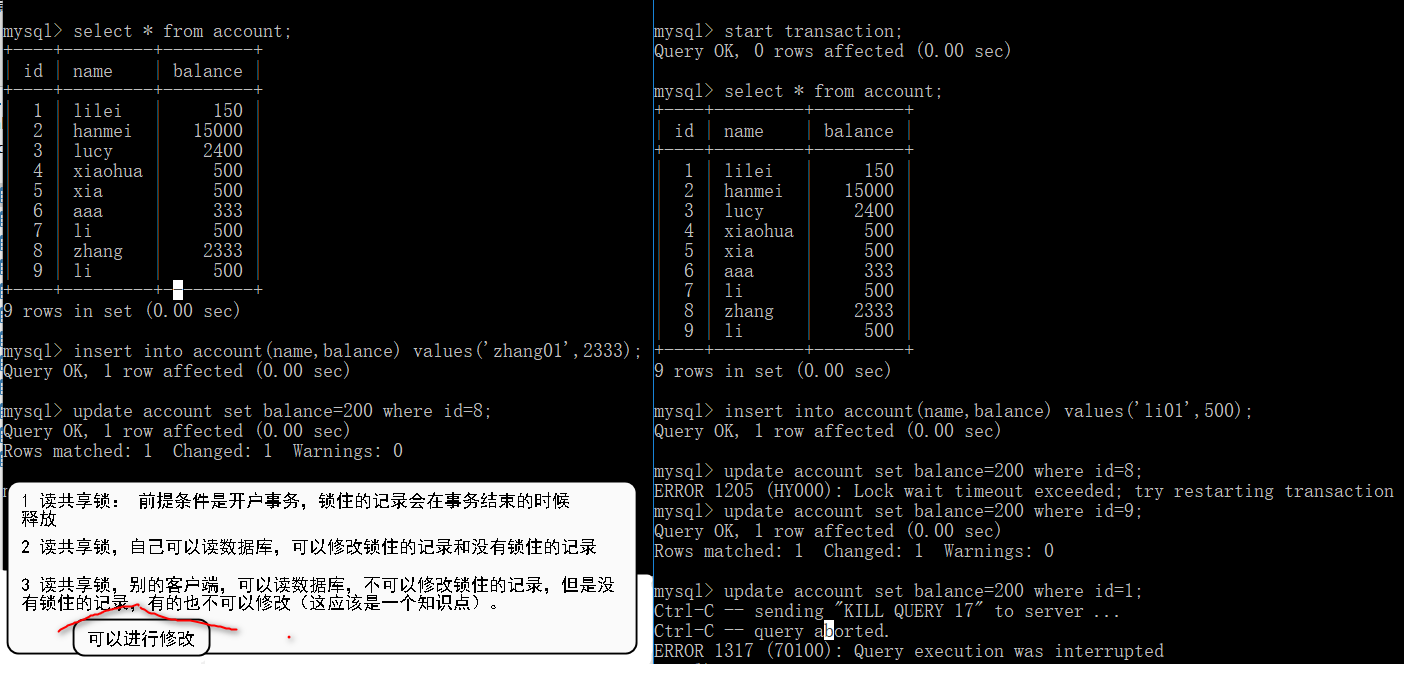
###### 行锁(innodb才支持)

row-level,操作会锁定操作的记录

共享行读锁和行排他锁要进行在事务当中

共享行读锁（自己可读写，别人可读所有，可写没有锁定）

对查询的记录增加共享锁 select \* from table where lock in share mode，案例为：



有一种特别的情况：

如果锁住的行是：id<8;

那么 增加一条数据 id=2,的数据，也是添加不进去的。

共享写行锁（自己可读写，可以读写没有锁定的数据）

对查询的记录增加排他锁 select \* from table where for update

1.7



## 1.7 索引

#### 1.7.0 关于索引，一个非常重要的文章

##### ****一、介绍****

###### 1.什么是索引？

一般的应用系统，读写比例在10:1左右，而且插入操作和一般的更新操作很少出现性能问题，在生产环境中，我们遇到最多的，也是最容易出问题的，还是一些复杂的查询操作，因此对查询语句的优化显然是重中之重。说起加速查询，就不得不提到索引了。

2.为什么要有索引呢？

索引在MySQL中也叫做“键”，是存储引擎用于快速找到记录的一种数据结构。索引对于良好的性能  
非常关键，尤其是当表中的数据量越来越大时，索引对于性能的影响愈发重要。  
索引优化应该是对查询性能优化最有效的手段了。索引能够轻易将查询性能提高好几个数量级。  
索引相当于字典的音序表，如果要查某个字，如果不使用音序表，则需要从几百页中逐页去查。

##### ****二、索引的原理****

###### **一 索引原理**

索引的目的在于提高查询效率，与我们查阅图书所用的目录是一个道理：先定位到章，然后定位到该章下的一个小节，然后找到页数。相似的例子还有：查字典，查火车车次，飞机航班等

**本质都是：通过不断地缩小想要获取数据的范围来筛选出最终想要的结果，同时把随机的事件变成顺序的事件，也就是说，有了这种索引机制，我们可以总是用同一种查找方式来锁定数据。**

数据库也是一样，但显然要复杂的多，因为不仅面临着等值查询，还有范围查询(>、<、between、in)、模糊查询(like)、并集查询(or)等等。数据库应该选择怎么样的方式来应对所有的问题呢？我们回想字典的例子，能不能把数据分成段，然后分段查询呢？最简单的如果1000条数据，1到100分成第一段，101到200分成第二段，201到300分成第三段......这样查第250条数据，只要找第三段就可以了，一下子去除了90%的无效数据。但如果是1千万的记录呢，分成几段比较好？稍有算法基础的同学会想到搜索树，其平均复杂度是lgN，具有不错的查询性能。但这里我们忽略了一个关键的问题，复杂度模型是基于每次相同的操作成本来考虑的。而数据库实现比较复杂，一方面数据是保存在磁盘上的，另外一方面为了提高性能，每次又可以把部分数据读入内存来计算，因为我们知道访问磁盘的成本大概是访问内存的十万倍左右，所以简单的搜索树难以满足复杂的应用场景。

###### **二 磁盘IO与预读**

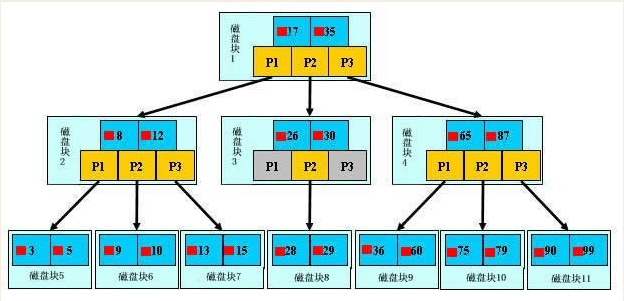
考虑到磁盘IO是非常高昂的操作，计算机操作系统做了一些优化，**当一次IO时，不光把当前磁盘地址的数据，而是把相邻的数据也都读取到内存缓冲区内**，因为局部预读性原理告诉我们，当计算机访问一个地址的数据的时候，与其相邻的数据也会很快被访问到。每一次IO读取的数据我们称之为一页(page)。具体一页有多大数据跟操作系统有关，一般为4k或8k，也就是我们读取一页内的数据时候，实际上才发生了一次IO，这个理论对于索引的数据结构设计非常有帮助。

##### ****三、索引的数据结构****

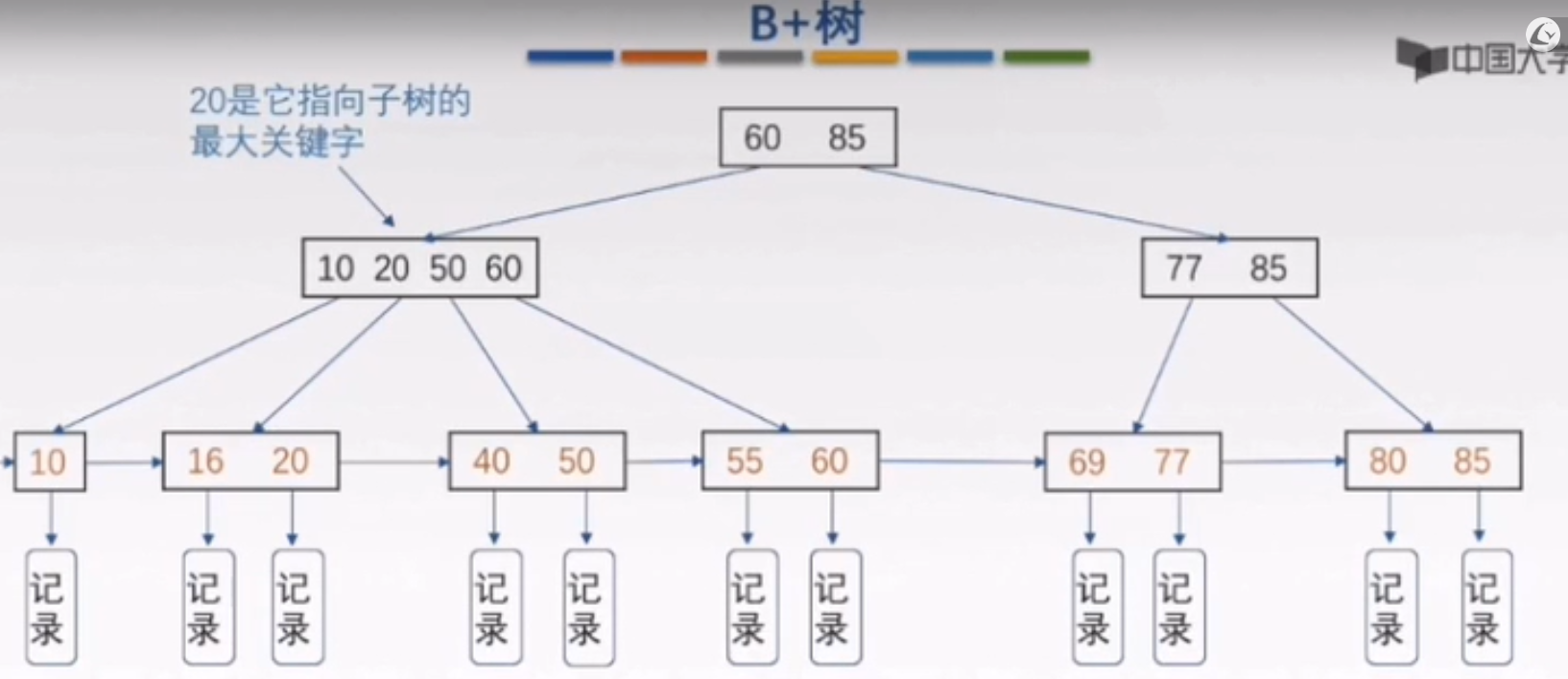
###### 数据结构

任何一种数据结构都不是凭空产生的，一定会有它的背景和使用场景，我们现在总结一下，我们需要这种数据结构能够做些什么，其实很简单，那就是：每次查找数据时把磁盘IO次数控制在一个很小的数量级，最好是常数数量级。那么我们就想到如果一个高度可控的多路搜索树是否能满足需求呢？就这样，b+树应运而生。

下面是一个b树



下面是一个b+树



10，16…. 会记录一个个地址，这个地址会映射一条条记录

如下图：



如上图，是一颗b+树，关于b+树的定义可以参见[B+树](http://zh.wikipedia.org/wiki/B%2B%E6%A0%91)，这里只说一些重点，浅蓝色的块我们称之为一个磁盘块，可以看到每个磁盘块包含几个数据项（深蓝色所示）和指针（黄色所示），如磁盘块1包含数据项17和35，包含指针P1、P2、P3，P1表示小于17的磁盘块，P2表示在17和35之间的磁盘块，P3表示大于35的磁盘块。真实的数据存在于叶子节点即3、5、9、10、13、15、28、29、36、60、75、79、90、99。非叶子节点只不存储真实的数据，只存储指引搜索方向的数据项，如17、35并不真实存在于数据表中。

###### b+树的查找过程 如图所示，如果要查找数据项29，那么首先会把磁盘块1由磁盘加载到内存，此时发生一次IO，在内存中用二分查找确定29在17和35之间，锁定磁盘块1的P2指针，内存时间因为非常短（相比磁盘的IO）可以忽略不计，通过磁盘块1的P2指针的磁盘地址把磁盘块3由磁盘加载到内存，发生第二次IO，29在26和30之间，锁定磁盘块3的P2指针，通过指针加载磁盘块8到内存，发生第三次IO，同时内存中做二分查找找到29，结束查询，总计三次IO。真实的情况是，3层的b+树可以表示上百万的数据，如果上百万的数据查找只需要三次IO，性能提高将是巨大的，如果没有索引，每个数据项都要发生一次IO，那么总共需要百万次的IO，显然成本非常非常高。

###### b+树性质 1****.索引字段要尽量的小****：通过上面的分析，我们知道IO次数取决于b+数的高度h，假设当前数据表的数据为N，每个磁盘块的数据项的数量是m，则有h=㏒(m+1)N，当数据量N一定的情况下，m越大，h越小；而m = 磁盘块的大小 / 数据项的大小，磁盘块的大小也就是一个数据页的大小，是固定的，如果数据项占的空间越小，数据项的数量越多，树的高度越低。这就是为什么每个数据项，即索引字段要尽量的小，比如int占4字节，要比bigint8字节少一半。这也是为什么b+树要求把真实的数据放到叶子节点而不是内层节点，一旦放到内层节点，磁盘块的数据项会大幅度下降，导致树增高。当数据项等于1时将会退化成线性表。 2.****索引的最左匹配特性（即从左往右匹配）****：当b+树的数据项是复合的数据结构，比如(name,age,sex)的时候，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。

###### 聚簇结构（关键字与记录存储在一起）



在mysql中，仅仅是innodb的主键索引为聚簇索引，其它的索引包括innodb的非主键索引，都是典型的btree结构。

###### Hash结构

当索引载入到内存时采用的存储结构，采用哈希结构存储，就是key-value的形式。

##### 四、Mysql索引管理

###### 功能

#1. 索引的功能就是加速查找

#2. mysql中的primary key，unique，联合唯一也都是索引，这些索引除了加速查找以外，还有约束的功能

二、MySQL的索引分类

###### 索引分类

1.普通索引index :加速查找

2.唯一索引

主键索引：primary key ：加速查找+约束（不为空且唯一）

唯一索引：unique：加速查找+约束 （唯一）

3.联合索引

-primary key(id,name):联合主键索引

-unique(id,name):联合唯一索引

-index(id,name):联合普通索引

4.全文索引fulltext :用于搜索很长一篇文章的时候，效果最好。

5.空间索引spatial :了解就好，几乎不用

###### 索引的两大类型hash与btree

#我们可以在创建上述索引的时候，为其指定索引类型，分两类

hash类型的索引：查询单条快，范围查询慢

btree类型的索引：b+树，层数越多，数据量指数级增长（我们就用它，因为innodb默认支持它）

#不同的存储引擎支持的索引类型也不一样

InnoDB 支持事务，支持行级别锁定，支持 B-tree、Full-text 等索引，不支持 Hash 索引；

MyISAM 不支持事务，支持表级别锁定，支持 B-tree、Full-text 等索引，不支持 Hash 索引；

Memory 不支持事务，支持表级别锁定，支持 B-tree、Hash 等索引，不支持 Full-text 索引；

NDB 支持事务，支持行级别锁定，支持 Hash 索引，不支持 B-tree、Full-text 等索引；

Archive 不支持事务，支持表级别锁定，不支持 B-tree、Hash、Full-text 等索引；

###### 创建/删除索引的语法

复制代码

1 #方法一：创建表时

2 　　CREATE TABLE 表名 (

3 字段名1 数据类型 [完整性约束条件…],

4 字段名2 数据类型 [完整性约束条件…],

5 [UNIQUE | FULLTEXT | SPATIAL ] INDEX | KEY

6 [索引名] (字段名[(长度)] [ASC |DESC])

7 );

8

9

10 #方法二：CREATE在已存在的表上创建索引

11 CREATE [UNIQUE | FULLTEXT | SPATIAL ] INDEX 索引名

12 ON 表名 (字段名[(长度)] [ASC |DESC]) ;

13

14

15 #方法三：ALTER TABLE在已存在的表上创建索引

16 ALTER TABLE 表名 ADD [UNIQUE | FULLTEXT | SPATIAL ] INDEX

17 索引名 (字段名[(长度)] [ASC |DESC]) ;

18

19 #删除索引：DROP INDEX 索引名 ON 表名字;

善用帮助文档

help create

help create index

==================

1.创建索引

-在创建表时就创建（需要注意的几点）

create table s1(

id int ,#可以在这加primary key

#id int index #不可以这样加索引，因为index只是索引，没有约束一说，

#不能像主键，还有唯一约束一样，在定义字段的时候加索引

name char(20),

age int,

email varchar(30)

#primary key(id) #也可以在这加

index(id) #可以这样加

);

-在创建表后在创建

create index name on s1(name); #添加普通索引

create unique age on s1(age);添加唯一索引

alter table s1 add primary key(id); #添加住建索引，也就是给id字段增加一个主键约束

create index name on s1(id,name); #添加普通联合索引

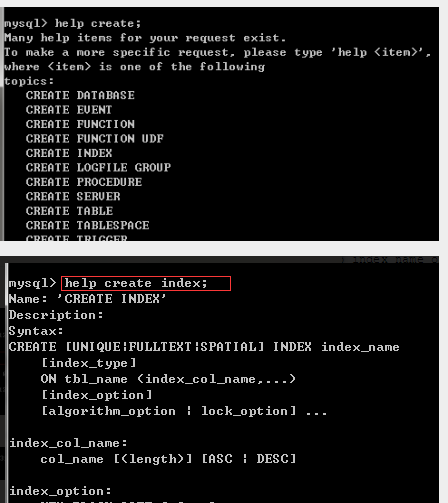
2.删除索引

drop index id on s1;

drop index name on s1; #删除普通索引

drop index age on s1; #删除唯一索引，就和普通索引一样，不用在index前加unique来删，直接就可以删了

alter table s1 drop primary key; #删除主键(因为它添加的时候是按照alter来增加的，那么我们也用alter来删)



##### 五、测试索引

#1. 准备表

create table s1(

id int,

name varchar(20),

gender char(6),

email varchar(50)

);

#2. 创建存储过程，实现批量插入记录

delimiter $$ #声明存储过程的结束符号为$$

create procedure auto\_insert1()

BEGIN

declare i int default 1;

while(i<3000000)do

insert into s1 values(i,concat('egon',i),'male',concat('egon',i,'@oldboy'));

set i=i+1;

end while;

END$$ #$$结束

delimiter ; #重新声明分号为结束符号

#3. 查看存储过程

show create procedure auto\_insert1\G

#4. 调用存储过程

call auto\_insert1();

2 、在没有索引的前提下测试查询速度

#无索引：从头到尾扫描一遍，所以查询速度很慢

mysql> select \* from s1 where id=333;

+------+---------+--------+----------------+

| id | name | gender | email |

+------+---------+--------+----------------+

| 333 | egon333 | male | 333@oldboy.com |

| 333 | egon333 | f | alex333@oldboy |

| 333 | egon333 | f | alex333@oldboy |

+------+---------+--------+----------------+

rows in set (0.32 sec)

mysql> select \* from s1 where email='egon333@oldboy';

....

... rows in set (0.36 sec)

3、 加上索引

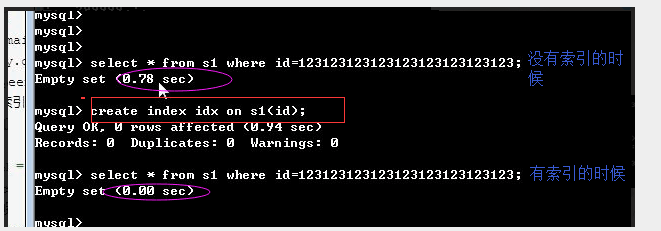
#1. 一定是为搜索条件的字段创建索引，比如select \* from t1 where age > 5;就需要为age加上索引

#2. 在表中已经有大量数据的情况下，建索引会很慢，且占用硬盘空间，插入删除更新都很慢，只有查询快

比如create index idx on s1(id);会扫描表中所有的数据，然后以id为数据项，创建索引结构，存放于硬盘的表中。

建完以后，再查询就会很快了

#3. 需要注意的是：innodb表的索引会存放于s1.ibd文件中，而myisam表的索引则会有单独的索引文件table1.MYI



##### 六、正确使用索引

###### 一、覆盖索引

#分析

select \* from s1 where id=123;

该sql命中了索引，但未覆盖索引。

利用id=123到索引的数据结构中定位到该id在硬盘中的位置，或者说再数据表中的位置。

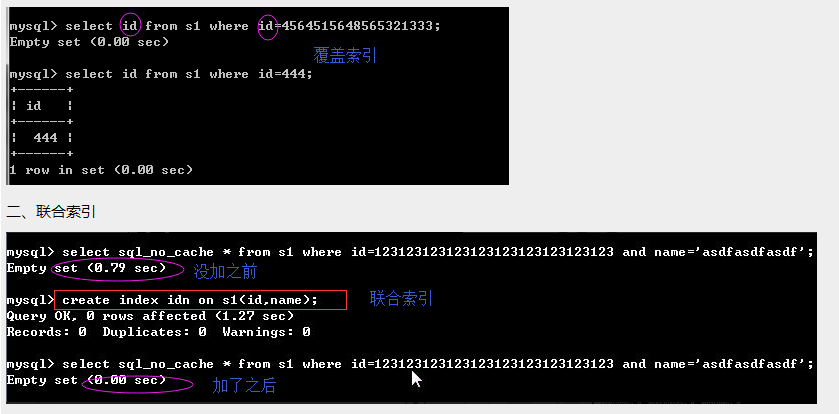
但是我们select的字段为\*，除了id以外还需要其他字段，这就意味着，我们通过索引结构取到id还不够，

还需要利用该id再去找到该id所在行的其他字段值，这是需要时间的，很明显，如果我们只select id，

就减去了这份苦恼，如下

select id from s1 where id=123;

这条就是覆盖索引了，命中索引，且从索引的数据结构直接就取到了id在硬盘的地址，速度很快



###### 三、索引合并

#索引合并：把多个单列索引合并使用

#分析：

组合索引能做到的事情，我们都可以用索引合并去解决，比如

create index ne on s1(name,email);#组合索引

我们完全可以单独为name和email创建索引

组合索引可以命中：

select \* from s1 where name='egon' ;

select \* from s1 where name='egon' and email='adf';

索引合并可以命中：

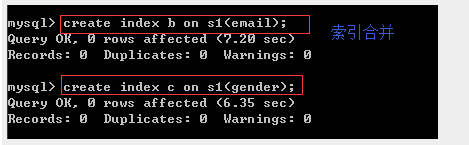
select \* from s1 where name='egon' ;

select \* from s1 where email='adf';

select \* from s1 where name='egon' and email='adf';

乍一看好像索引合并更好了：可以命中更多的情况，但其实要分情况去看，如果是name='egon' and email='adf',

那么组合索引的效率要高于索引合并，如果是单条件查，那么还是用索引合并比较合理



**三 若想利用索引达到预想的提高查询速度的效果，我们在添加索引时，必须遵循以下原则**

#1.最左前缀匹配原则，非常重要的原则，

create index ix\_name\_email on s1(name,email,)

- 最左前缀匹配：必须按照从左到右的顺序匹配

select \* from s1 where name='egon'; #可以

select \* from s1 where name='egon' and email='asdf'; #可以

select \* from s1 where email='alex@oldboy.com'; #不可以

mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，

比如a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，

d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。

#2.=和in可以乱序，比如a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任意顺序，mysql的查询优化器

会帮你优化成索引可以识别的形式

#3.尽量选择区分度高的列作为索引,区分度的公式是count(distinct col)/count(\*)，

表示字段不重复的比例，比例越大我们扫描的记录数越少，唯一键的区分度是1，而一些状态、

性别字段可能在大数据面前区分度就是0，那可能有人会问，这个比例有什么经验值吗？使用场景不同，

这个值也很难确定，一般需要join的字段我们都要求是0.1以上，即平均1条扫描10条记录

#4.索引列不能参与计算，保持列“干净”，比如from\_unixtime(create\_time) = ’2014-05-29’

就不能使用到索引，原因很简单，b+树中存的都是数据表中的字段值，

但进行检索时，需要把所有元素都应用函数才能比较，显然成本太大。

所以语句应该写成create\_time = unix\_timestamp(’2014-05-29’);

**最左前缀示范**

mysql> select \* from s1 where id>3 and name='egon' and email='alex333@oldboy.com' and gender='male';

Empty set (0.39 sec)

mysql> create index idx on s1(id,name,email,gender); #未遵循最左前缀

Query OK, 0 rows affected (15.27 sec)

Records: 0 Duplicates: 0 Warnings: 0

mysql> select \* from s1 where id>3 and name='egon' and email='alex333@oldboy.com' and gender='male';

Empty set (0.43 sec)

mysql> drop index idx on s1;

Query OK, 0 rows affected (0.16 sec)

Records: 0 Duplicates: 0 Warnings: 0

mysql> create index idx on s1(name,email,gender,id); #遵循最左前缀

Query OK, 0 rows affected (15.97 sec)

Records: 0 Duplicates: 0 Warnings: 0

mysql> select \* from s1 where id>3 and name='egon' and email='alex333@oldboy.com' and gender='male';

Empty set (0.03 sec)

1 6. 最左前缀匹配

2 index(id,age,email,name)

3 #条件中一定要出现id(只要出现id就会提升速度)

4 id

5 id age

6 id email

7 id name

8

9 email #不行 如果单独这个开头就不能提升速度了

10 mysql> select count(\*) from s1 where id=3000;

11 +----------+

12 | count(\*) |

13 +----------+

14 | 1 |

15 +----------+

16 1 row in set (0.11 sec)

17

18 mysql> create index xxx on s1(id,name,age,email);

19 Query OK, 0 rows affected (6.44 sec)

20 Records: 0 Duplicates: 0 Warnings: 0

21

22 mysql> select count(\*) from s1 where id=3000;

23 +----------+

24 | count(\*) |

25 +----------+

26 | 1 |

27 +----------+

28 1 row in set (0.00 sec)

29

30 mysql> select count(\*) from s1 where name='egon';

31 +----------+

32 | count(\*) |

33 +----------+

34 | 299999 |

35 +----------+

36 1 row in set (0.16 sec)

37

38 mysql> select count(\*) from s1 where email='egon3333@oldboy.com';

39 +----------+

40 | count(\*) |

41 +----------+

42 | 1 |

43 +----------+

44 1 row in set (0.15 sec)

45

46 mysql> select count(\*) from s1 where id=1000 and email='egon3333@oldboy.com';

47 +----------+

48 | count(\*) |

49 +----------+

50 | 0 |

51 +----------+

52 1 row in set (0.00 sec)

53

54 mysql> select count(\*) from s1 where email='egon3333@oldboy.com' and id=3000;

55 +----------+

56 | count(\*) |

57 +----------+

58 | 0 |

59 +----------+

60 1 row in set (0.00 sec)

**索引无法命中的情况需要注意：**

- like '%xx'

select \* from tb1 where email like '%cn';

- 使用函数

select \* from tb1 where reverse(email) = 'wupeiqi';

- or

select \* from tb1 where nid = 1 or name = 'seven@live.com';

特别的：当or条件中有未建立索引的列才失效，以下会走索引

select \* from tb1 where nid = 1 or name = 'seven';

select \* from tb1 where nid = 1 or name = 'seven@live.com' and email = 'alex'

- 类型不一致

如果列是字符串类型，传入条件是必须用引号引起来，不然...

select \* from tb1 where email = 999;

普通索引的不等于不会走索引

- !=

select \* from tb1 where email != 'alex'

特别的：如果是主键，则还是会走索引

select \* from tb1 where nid != 123

- >

select \* from tb1 where email > 'alex'

特别的：如果是主键或索引是整数类型，则还是会走索引

select \* from tb1 where nid > 123

select \* from tb1 where num > 123

#排序条件为索引，则select字段必须也是索引字段，否则无法命中

- order by

select name from s1 order by email desc;

当根据索引排序时候，select查询的字段如果不是索引，则不走索引

select email from s1 order by email desc;

特别的：如果对主键排序，则还是走索引：

select \* from tb1 order by nid desc;

- 组合索引最左前缀

如果组合索引为：(name,email)

name and email -- 使用索引

name -- 使用索引

email -- 不使用索引

- count(1)或count(列)代替count(\*)在mysql中没有差别了

- create index xxxx on tb(title(19)) #text类型，必须制定长度

- 避免使用select \*

- count(1)或count(列) 代替 count(\*)

- 创建表时尽量时 char 代替 varchar

- 表的字段顺序固定长度的字段优先

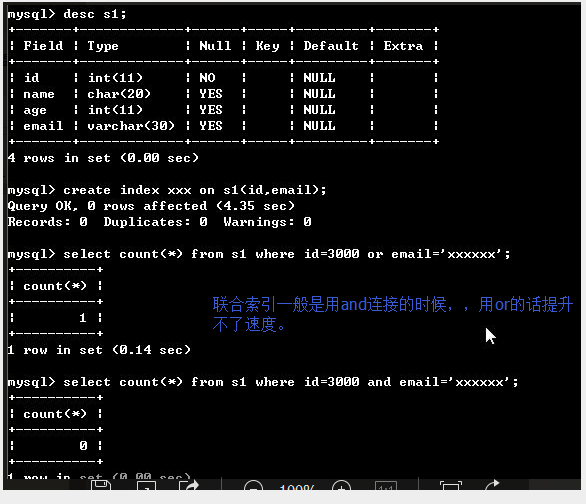
- 组合索引代替多个单列索引（经常使用多个条件查询时）

- 尽量使用短索引

- 使用连接（JOIN）来代替子查询(Sub-Queries)

- 连表时注意条件类型需一致

- 索引散列值（重复多的数据）不适合建索引，例：性别不适合



##### 七、慢查询优化的基本步骤

0.先运行看看是否真的很慢，注意设置SQL\_NO\_CACHE

1.where条件单表查，锁定最小返回记录表。这句话的意思是把查询语句的where都应用到表中返回的记录数最小的表开始查起，单表每个字段分别查询，看哪个字段的区分度最高

2.explain查看执行计划，是否与1预期一致（从锁定记录较少的表开始查询）

3.order by limit 形式的sql语句让排序的表优先查

4.了解业务方使用场景

5.加索引时参照建索引的几大原则

6.观察结果，不符合预期继续从0分析

#### 1.7.1 是什么

索引：关键字与数据位置映射关系，称之为索引

关键字：从数据中提取，用于标识、检索数据的特定内容

* 关键字相对于数据本身，数据量小、且排好序的

Mysql优化的最常用的手段

#### 1.7.2 mysql中索引的类型

支持的索引 普通索引，唯一索引，primary key 索引，全文索引(不支持中文)。

* 普通索引：对索引关键字没有限制
* 唯一索引：对索引关键字不能重复
* Primary key: 对索引关键字不能重复且不能为null

#### 1.7.3 索引语法

##### 创建索引

ALTER TABLE用来创建普通索引、UNIQUE索引或PRIMARY KEY索引。

(1.)ALTER TABLE table\_name ADD INDEX index\_name (column\_list)

(2.)ALTER TABLE table\_name ADD UNIQUE (column\_list)

(3.)ALTER TABLE table\_name ADD PRIMARY KEY (column\_list)、

##### 删除索引 可利用ALTER TABLE或DROP INDEX语句来删除索引。类似于CREATE INDEX语句，DROP INDEX可以在ALTER TABLE内部作为一条语句处理，语法如下。

(1)DROP INDEX index\_name ON talbe\_name  
(2)ALTER TABLE table\_name DROP INDEX index\_name  
(3)ALTER TABLE table\_name DROP PRIMARY KEY

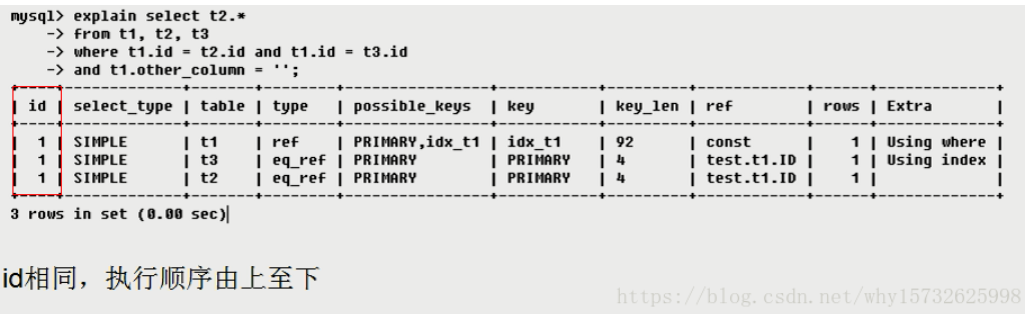
其中，前两条语句是等价的，删除掉table\_name中的索引index\_name。  
第3条语句只在删除PRIMARY KEY索引时使用，因为一个表只可能有一个PRIMARY KEY索引，因此不需要指定索引名。如果没有创建PRIMARY KEY索引，但表具有一个或多个UNIQUE索引，则MySQL将删除第一个UNIQUE索引。  
如果从表中删除了某列，则索引会受到影响。对于多列组合的索引，如果删除其中的某列，则该列也会从索引中删除。如果删除组成索引的所有列，则整个索引将被删除。

##### 查看索引 (1)show index from tblname; (2)show keys from tblname;

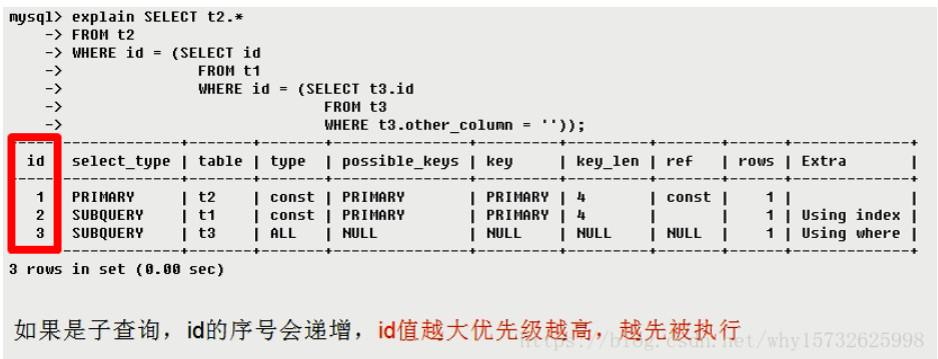
#### 1.7.4 explain的语法

##### Id

id相同，执行顺序由上至下

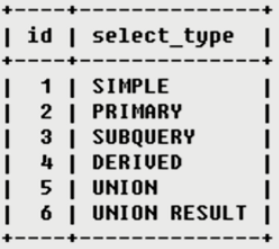


id不同，如果是子查询，id的序号会递增，id值越大优先级越高，越先被执行





##### select\_type



SIMPLE 简单的select查询，查询中不包含子查询或者UNION

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PRIMARY

SUBQUERY(子查询中的第一个SELECT，结果不依赖于外部查询)

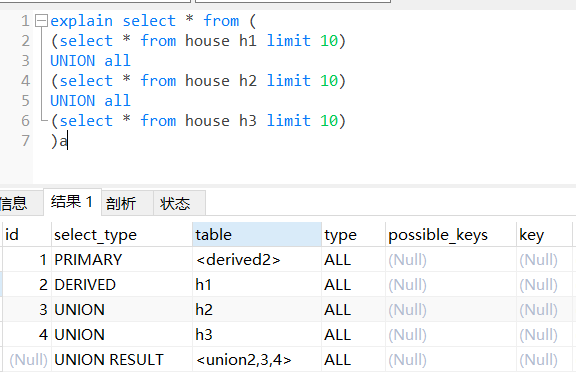
DEPENDENT SUBQUERY(子查询中的第一个SELECT，依赖于外部查询)

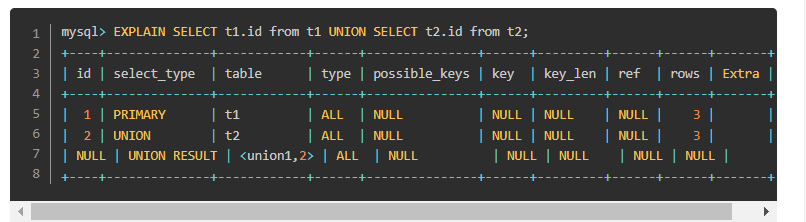
DERIVED 在FROM列表中包含的第一个子查询被标记为DERIVED（衍生），MySQL会递归执行这些子查询，把结果放在临时表中

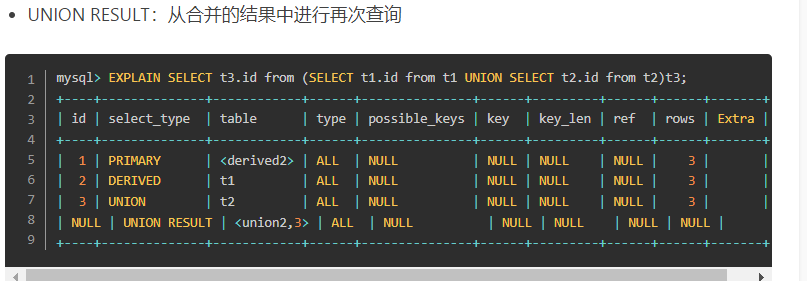
UNION 若第二个SELECT出现在UNION之后，则被标记为UNION：若UNION包含在FROM子句的子查询中，外层SELECT将被标记为：DERIVED

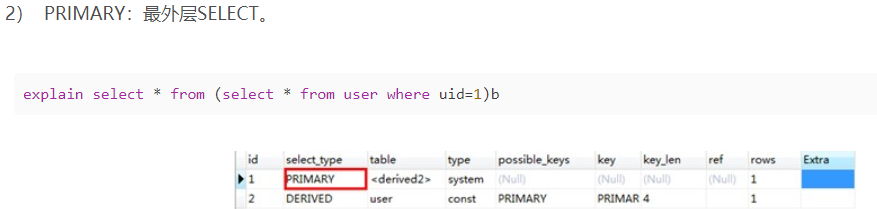
UNION RESULT 从UNION表获取结果的SELECT

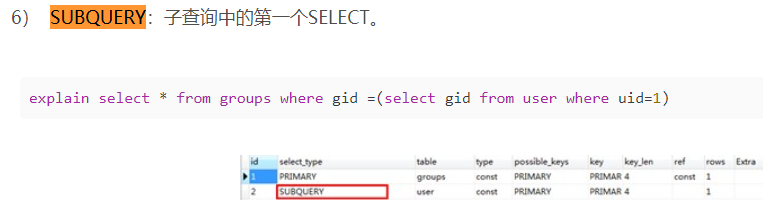
下面是几个例子：

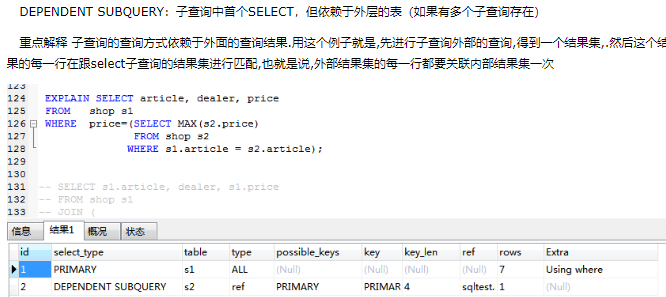


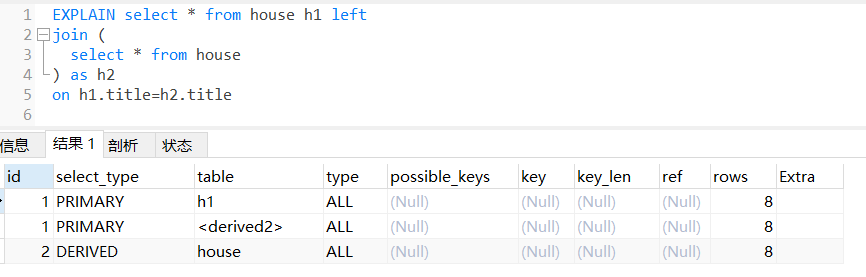


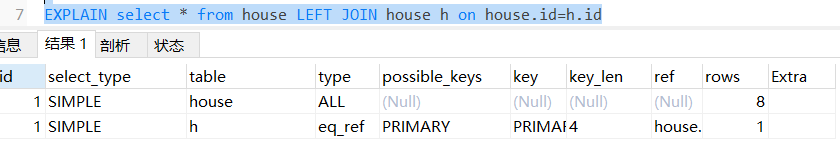












##### Type

**ALL、index、range、 ref、eq\_ref、const、system、NULL（从左到右，性能从差到好）**

ALL：Full Table Scan， MySQL将遍历全表以找到匹配的行

index: Full Index Scan，index与ALL区别为index类型只遍历索引树

range:只检索给定范围的行，使用一个索引来选择行

ref: 表示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值

eq\_ref: 类似ref，区别就在使用的索引是唯一索引，对于每个索引键值，表中只有一条记录匹配，简单来说，就是多表连接中使用primary key或者 unique key作为关联条件

const、system: 当MySQL对查询某部分进行优化，并转换为一个常量时，使用这些类型访问。如将主键置于where列表中，MySQL就能将该查询转换为一个常量，system是const类型的特例，当查询的表只有一行的情况下，使用system

NULL: MySQL在优化过程中分解语句，执行时甚至不用访问表或索引，例如从一个索引列里选取最小值可以通过单独索引查找完成。

##### ****possible\_keys****

**指出MySQL能使用哪个索引在表中找到记录，查询涉及到的字段上若存在索引，则该索引将被列出，但不一定被查询使用（该查询可以利用的索引，如果没有任何索引显示 null）**

该列完全独立于EXPLAIN输出所示的表的次序。这意味着在possible\_keys中的某些键实际上不能按生成的表次序使用。  
如果该列是NULL，则没有相关的索引。在这种情况下，可以通过检查WHERE子句看是否它引用某些列或适合索引的列来提高你的查询性能。如果是这样，创造一个适当的索引并且再次用EXPLAIN检查查询

##### **Key**

**key列显示MySQL实际决定使用的键（索引），必然包含在possible\_keys中**

如果没有选择索引，键是NULL。要想强制MySQL使用或忽视possible\_keys列中的索引，在查询中使用FORCE INDEX、USE INDEX或者IGNORE INDEX。

##### Ken\_len

key\_len 的值，根据这个值可以判断索引的长度，在组合索引里面可以更清楚的了解到了哪部分字段使用到了索引。

着重介绍一下这个长度计算方式：

Varchar:

Utf8 \*3

Gbk \*2

如果字段允许NULL的会多出一个字节，固定长度char字段不需要额外的字节

varchar是一个变长类型，它需要固定两个字节记录长度

int :

需要4个字节

如果字段允许NULL的会多出一个字节

##### ref

表示上述表的连接匹配条件，表示哪些列或者常量被用于查找索引列上的值

##### Rows

估算找到所需记录所需要读取的行数据

**Example**

[复制代码](javascript:void(0);)

(root@yayun-mysql-server) [test]>explain select \* from t1 , t2 where t1.id=t2.id and t2.name='atlas';

+----+-------------+-------+--------+---------------+---------+---------+------------+------+-------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

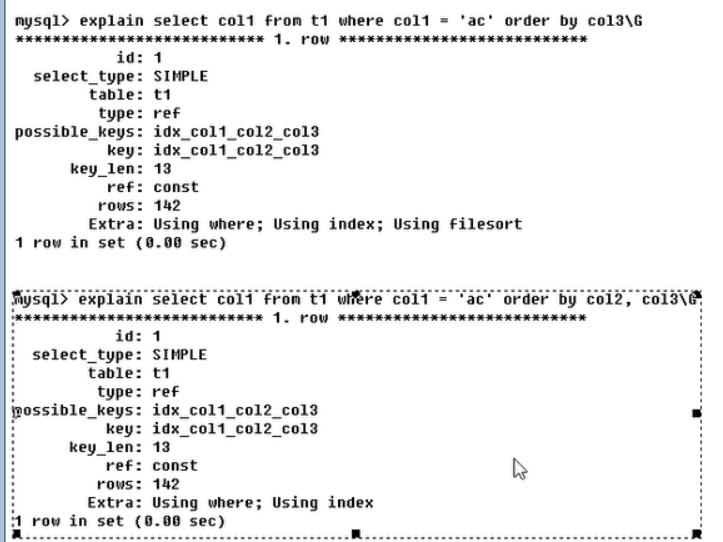
+----+-------------+-------+--------+---------------+---------+---------+------------+------+-------------+

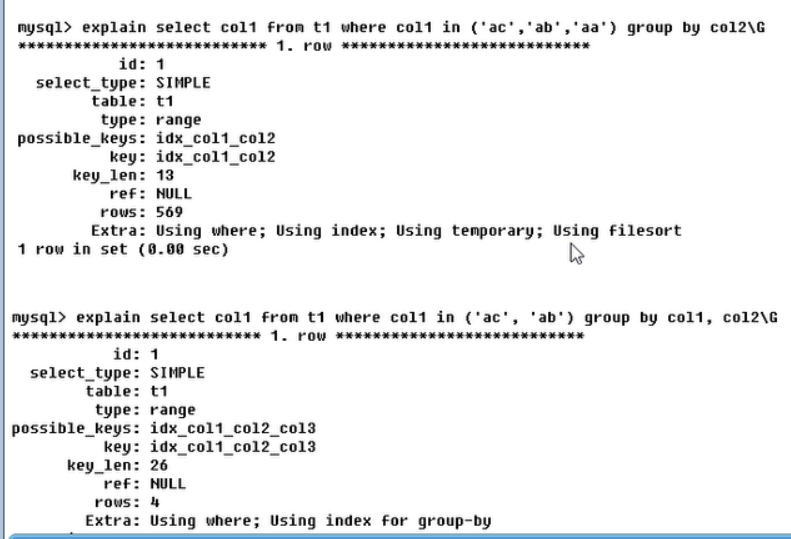
| **1** | SIMPLE | t2 | ref | PRIMARY,name | name | **63** | const | **1** | Using where |

| **1** | SIMPLE | t1 | eq\_ref | PRIMARY | PRIMARY | **4** | test.t2.id | **1** | |

+----+-------------+-------+--------+---------------+---------+---------+------------+------+-------

##### Extra





对于group by ,使用的列的个数与顺序与建立索引的个数与顺序应该一样

a. Using index  
该值表示相应的select操作中使用了覆盖索引（Covering Index）

**Example**

(root@yayun-mysql-server) [test]>explain select id from t1;

+----+-------------+-------+-------+---------------+------+---------+------+------+-------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+-------+-------+---------------+------+---------+------+------+-------------+

| **1** | SIMPLE | t1 | index | NULL | age | **5** | NULL | **4** | Using index |

+----+-------------+-------+-------+---------------+------+---------+------+------+-------------+

**1** row in set (**0.00** sec)

[复制代码](javascript:void(0);)

覆盖索引（Covering Index）  
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  
包含所有满足查询需要的数据的索引称为覆盖索引（Covering Index）  
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

如果同时出现using where 表明索引被用来执行索引键值的查找

如果没有出现using where 表明索引用来读取数据而非执行查找动作

b. Using where  
表示mysql服务器将在存储引擎检索行后再进行过滤。许多where条件里涉及索引中的列，当（并且如果）它读取索引时，就能被存储引擎检验，因此不是所有带where字句的查询都会显示"Using where"。有时"Using where"的出现就是一个暗示：查询可受益与不同的索引。

(root@yayun-mysql-server) [test]>explain select id,name from t1 where id<**4**;

+----+-------------+-------+-------+---------------+------+---------+------+------+--------------------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+-------+-------+---------------+------+---------+------+------+--------------------------+

| **1** | SIMPLE | t1 | index | PRIMARY | name | **63** | NULL | **4** | Using where; Using index |

+----+-------------+-------+-------+---------------+------+---------+------+------+--------------------------+

**1** row in set (**0.00** sec)

（1）返回所有记录的SQL，不使用where条件过滤数据，大概率不符合预期，对于这类SQL往往需要进行优化；

（2）使用了where条件的SQL，并不代表不需要优化，往往需要配合explain结果中的type（连接类型）来综合判断

　　本例虽然Extra字段说明使用了where条件过滤，但type属性是ALL，表示需要扫描全部数据，仍有优化空间。

　　常见的优化方法为，在where过滤属性上添加索引。

[复制代码](javascript:void(0);)

c. Using temporary（比use filesort还差的一种）  
使用了临时表保存中间结果，mysql在对查询结果排序时使用了临时表，常见于order by group by

[复制代码](javascript:void(0);)

(root@yayun-mysql-server) [test]>explain select id from t1 where id in (**1**,**2**) group by age,name;

+----+-------------+-------+-------+---------------+---------+---------+------+------+----------------------------------------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+-------+-------+---------------+---------+---------+------+------+----------------------------------------------+

| **1** | SIMPLE | t1 | range | PRIMARY | PRIMARY | **4** | NULL | **2** | Using where; Using temporary; Using filesort |

+----+-------------+-------+-------+---------------+---------+---------+------+------+----------------------------------------------+

[复制代码](javascript:void(0);)

d. Using filesort  
说明mysql 会对数据使用一个外部的索引排序，而不是按照表内的索引进行读取，mysql无法利用索引完成的排序操作称为‘文件排序。

一般人的回答是： “当行数据太大，导致内存无法容下这些数据产生的临时表时，他们就会被放入磁盘中排序。”  很不幸，这个答案是错的 ，临时表在太大的时候确实会到磁盘离去，但是EXPLAIN不会显示这些。那么事实是， filesort 这个名字取得太搓逼了。 filesort的意思是只要一个排序无法使用索引来排序，就叫filesort。他和file没半毛钱关系。filesort应该叫做sort。（笔者补充一下：意思是说如果无法用已有index来排序，那么就需要数据库服务器额外的进行数据排序，这样其实是会增加性能开销的。）

另外不光order by会出现filesort  group by没有使用索引一样会出现

当无法避免排序操作时，又该如何来优化呢？很显然，应该尽可能让 MySQL 选择使用第二种单路算法来进行排序。这样可以减少大量的随机IO操作，很大幅度地提高排序工作的效率。

**1. 合理设置索引 让排序字段使用上索引排序**

**2. 加大 max\_length\_for\_sort\_data 参数的设置**

**3. 去掉不必要的返回字段**

当内存不是很充裕时，不能简单地通过强行加大上面的参数来强迫 MySQL 去使用改进版的排序算法，否则可能会造成 MySQL 不得不将数据分成很多段，然后进行排序，这样可能会得不偿失。此时就须要去掉不必要的返回字段，让返回结果长度适应 max\_length\_for\_sort\_data 参数的限制。

**4. 增大 sort\_buffer\_size 参数设置**

增大 sort\_buffer\_size 并不是为了让 MySQL选择改进版的排序算法，而是为了让MySQL尽量减少在排序过程中对须要排序的数据进行分段，因为分段会造成 MySQL 不得不使用临时表来进行交换排序。

(root@yayun-mysql-server) [test]>explain select id,age from t1 order by name;

+----+-------------+-------+------+---------------+------+---------+------+------+----------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+-------+------+---------------+------+---------+------+------+----------------+

| **1** | SIMPLE | t1 | ALL | NULL | NULL | NULL | NULL | **4** | Using filesort |

+----+-------------+-------+------+---------------+------+---------+------+------+----------------+

**1** row in set (**0.00** sec)

(root@yayun-mysql-server) [test]>explain select id,age from t1 order by age;

+----+-------------+-------+-------+---------------+------+---------+------+------+-------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+-------+-------+---------------+------+---------+------+------+-------------+

| **1** | SIMPLE | t1 | index | NULL | age | **5** | NULL | **4** | Using index |

+----+-------------+-------+-------+---------------+------+---------+------+------+-------------+

**1** row in set (**0.00** sec)

(root@yayun-mysql-server) [test]>

[复制代码](javascript:void(0);)

**e. Using join buffer**  
**改值强调了在获取连接条件时没有使用索引，并且需要连接缓冲区来存储中间结果。如果出现了这个值，那应该注意，根据查询的具体情况可能需要添加索引来改进能。**

**Example**

[复制代码](javascript:void(0);)

(root@yayun-mysql-server) [test]>explain select t1.name from t1 inner join t2 on t1.name=t2.name;

+----+-------------+-------+-------+---------------+------+---------+--------------+------+--------------------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+-------+-------+---------------+------+---------+--------------+------+--------------------------+

| **1** | SIMPLE | t1 | index | name | name | **63** | NULL | **4** | Using index |

| **1** | SIMPLE | t2 | ref | name | name | **63** | test.t1.name | **2** | Using where; Using index |

+----+-------------+-------+-------+---------------+------+---------+--------------+------+--------------------------+

(root@yayun-mysql-server) [test]>alter table t1 drop key name;

Query OK, **0** rows affected (**0.02** sec)

Records: **0** Duplicates: **0** Warnings: **0**

(root@yayun-mysql-server) [test]>alter table t2 drop key name;

Query OK, **0** rows affected (**0.02** sec)

Records: **0** Duplicates: **0** Warnings: **0**

(root@yayun-mysql-server) [test]>explain select t1.name from t1 inner join t2 on t1.name=t2.name;

+----+-------------+-------+------+---------------+------+---------+------+------+--------------------------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+-------+------+---------------+------+---------+------+------+--------------------------------+

| **1** | SIMPLE | t1 | ALL | NULL | NULL | NULL | NULL | **4** | |

| **1** | SIMPLE | t2 | ALL | NULL | NULL | NULL | NULL | **4** | Using where; Using join buffer |

+----+-------------+-------+------+---------------+------+---------+------+------+--------------------------------+

**2** rows in set (**0.00** sec)

(root@yayun-mysql-server) [test]>

[复制代码](javascript:void(0);)

**f. Impossible where**  
**这个值强调了where语句会导致没有符合条件的行。**

**Example**

[复制代码](javascript:void(0);)

(root@yayun-mysql-server) [test]>EXPLAIN SELECT \* FROM t1 WHERE **1**=**2**;

+----+-------------+-------+------+---------------+------+---------+------+------+------------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+-------+------+---------------+------+---------+------+------+------------------+

| **1** | SIMPLE | NULL | NULL | NULL | NULL | NULL | NULL | NULL | Impossible WHERE |

+----+-------------+-------+------+---------------+------+---------+------+------+------------------+

**1** row in set (**0.00** sec)

(root@yayun-mysql-server) [test]>

[复制代码](javascript:void(0);)

**h. Select tables optimized away**  
**这个值意味着仅通过使用索引，优化器可能仅从聚合函数结果中返回一行.**

**Example**

[复制代码](javascript:void(0);)

(root@yayun-mysql-server) [test]>explain select max(id) from t1;

+----+-------------+-------+------+---------------+------+---------+------+------+------------------------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+-------+------+---------------+------+---------+------+------+------------------------------+

| **1** | SIMPLE | NULL | NULL | NULL | NULL | NULL | NULL | NULL | Select tables optimized away |

+----+-------------+-------+------+---------------+------+---------+------+------+------------------------------+

**1** row in set (**0.00** sec)

(root@yayun-mysql-server) [test]>

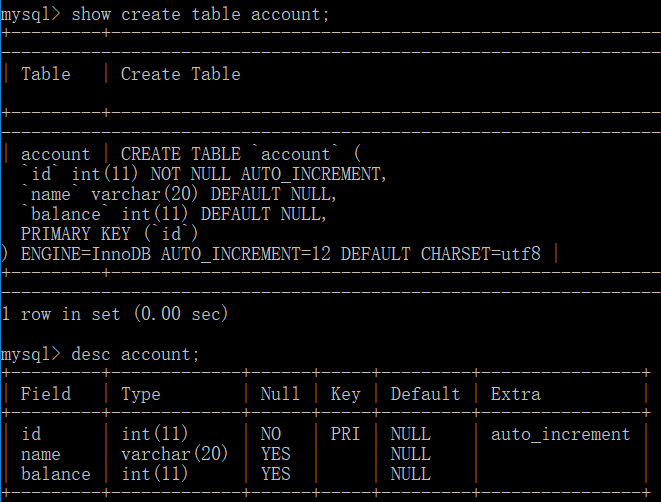
[复制代码](javascript:void(0);)

**I. Index merges**  
**当MySQL 决定要在一个给定的表上使用超过一个索引的时候，就会出现以下格式中的一个，详细说明使用的索引以及合并的类型。**  
**Using sort\_union(...)**  
**Using union(...)**  
**Using intersect(...)**

**总结：  
• EXPLAIN不会告诉你关于触发器、存储过程的信息或用户自定义函数对查询的影响情况  
• EXPLAIN不考虑各种Cache  
• EXPLAIN不能显示MySQL在执行查询时所作的优化工作  
• 部分统计信息是估算的，并非精确值  
• EXPALIN只能解释SELECT操作，其他操作要重写为SELECT后查看执行计划。**

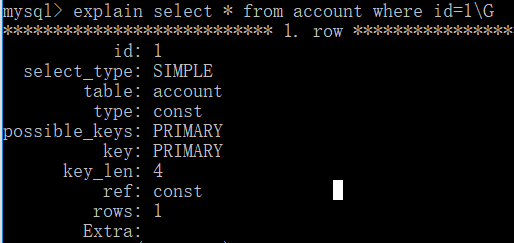
#### 1.7.5 索引的使用场景

##### 1.7.5.0 实验表数据

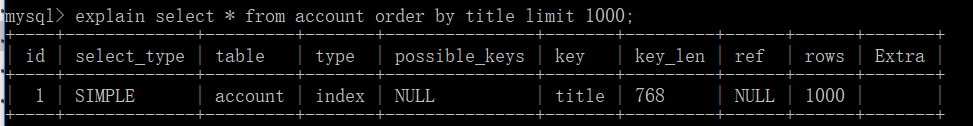


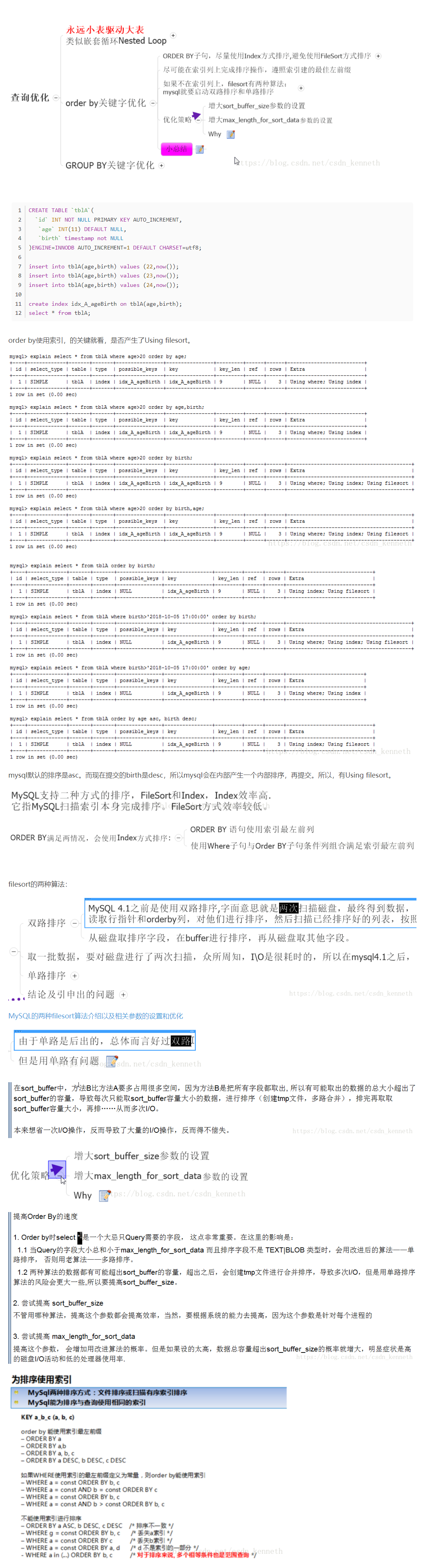
建立的索引，会在哪些情况被使用

##### 1.7.5.1 where查询

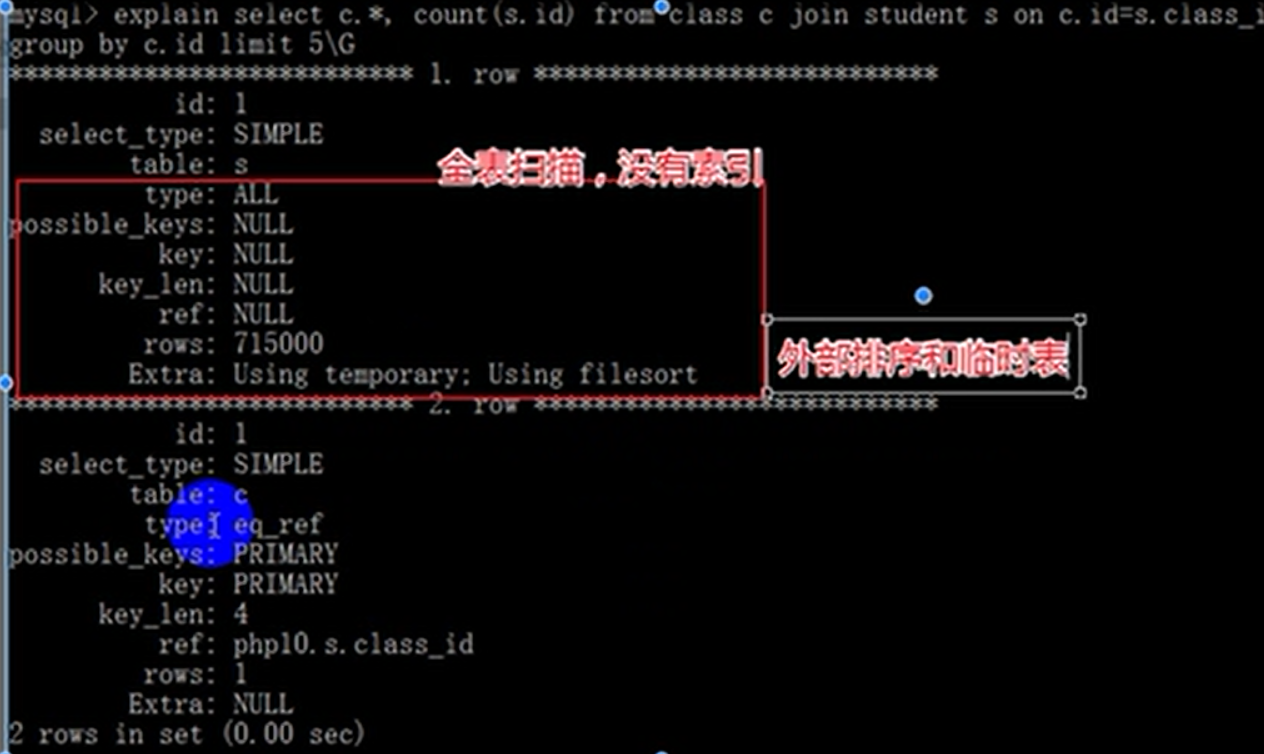


##### 1.7.5.2 order by

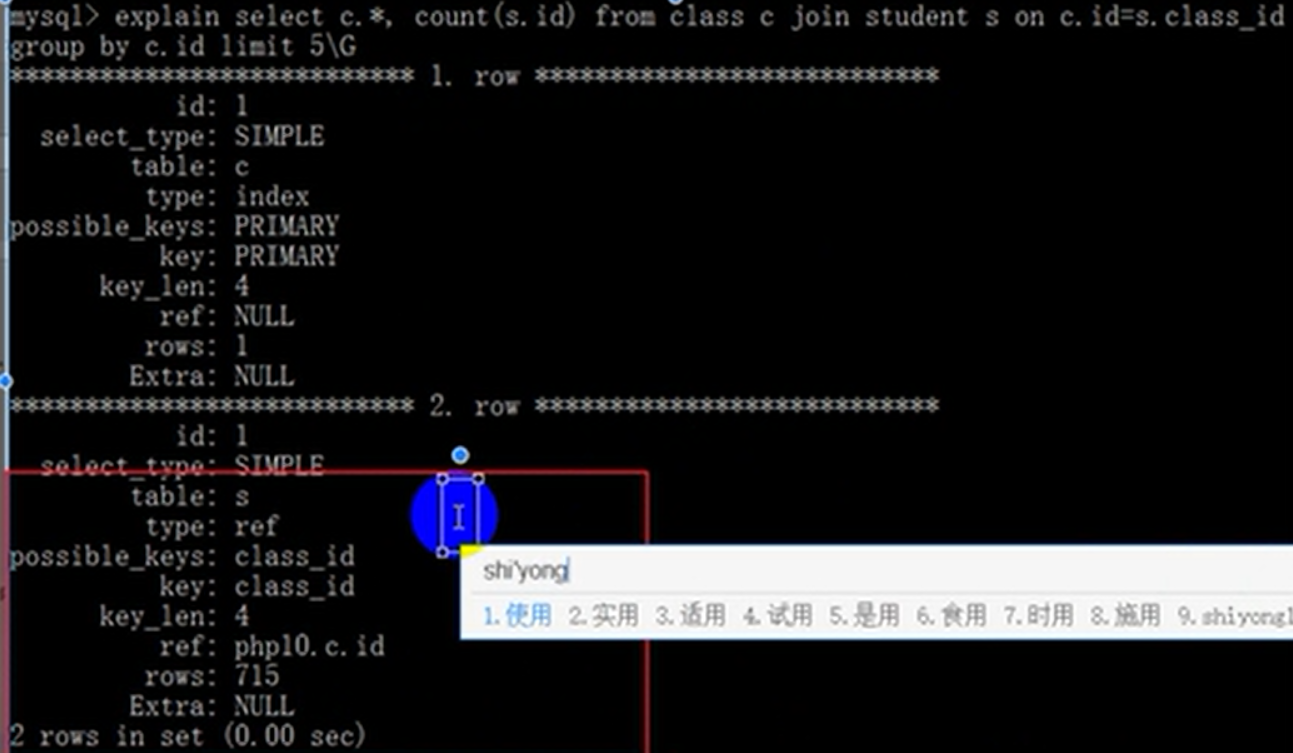




##### 1.7.5.3 join 连接

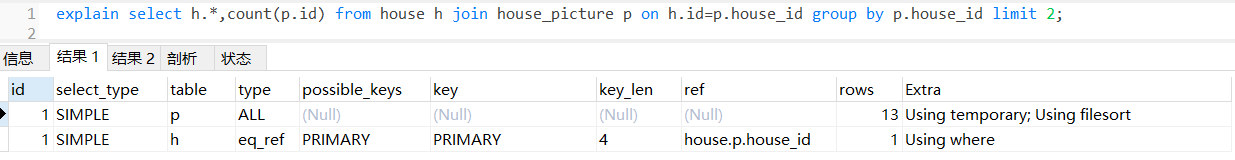


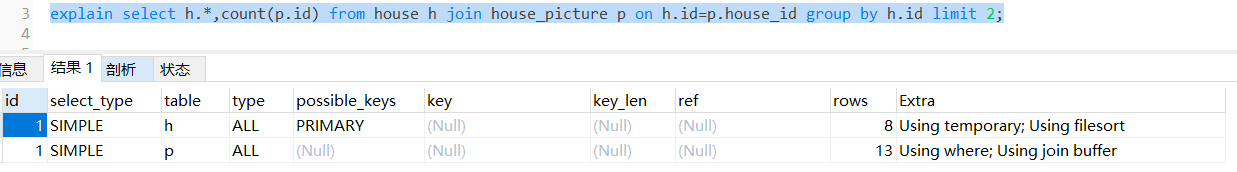
加上class\_id加上索引



下面是我做的一个例子：为什么出现下面的情况，我也不清楚

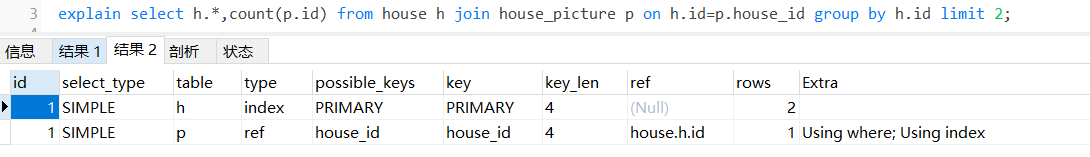
没有建立索引前：





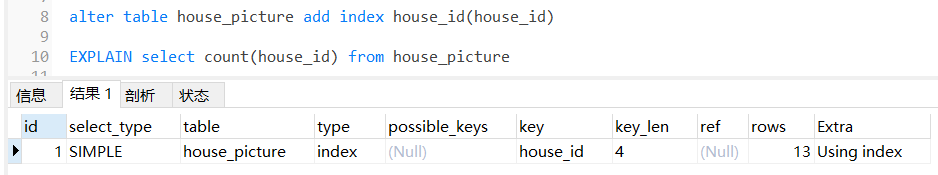
建立了索引之后 ：alter table house\_picture add index (house\_id)





1.7.5.4 索引覆盖

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。



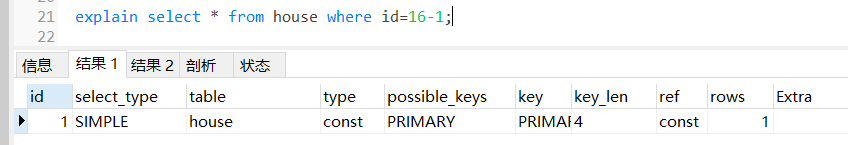
在extra 中使用了using index是覆盖索引的标志

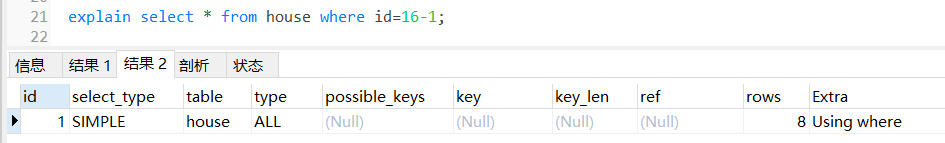
#### 1.7.6 使用索引的注意点

在满足使用索引的情况下，即使建立了索引，有的时候由于使用不对，也用不到索引。

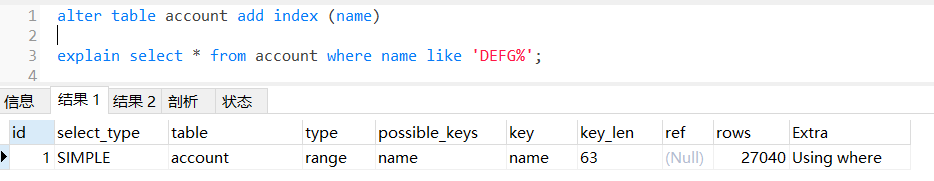
##### 1.7.6.1 索引字段需要独立出现，不出现运算

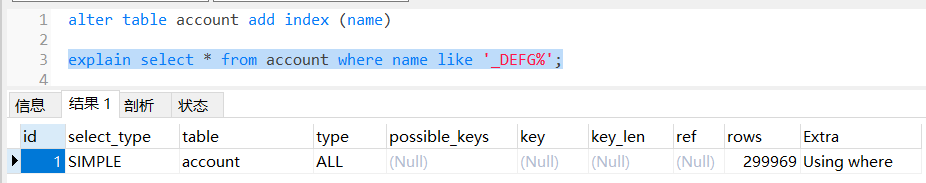
索引字段需要独立出现在表达式的一侧，不能对索引字段进行运算

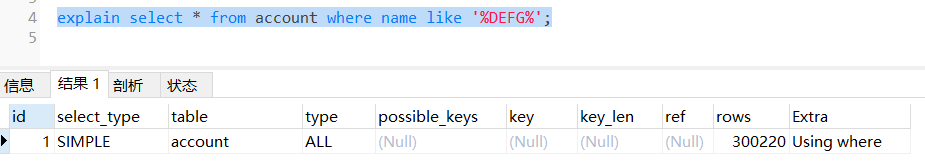




##### 1.7.6.2 索引不能使用通配符（ % \_）开头







包含的逻辑，like ‘%xx% 不能使用，正确的使用办法是进行全文索引。但是mysql的全文索引又不能使用中文。

##### 1.7.6.3 复合索引中，要符合最左前缀原则

复合索引的一点知识点

一：假设你在表的state、city和zip数据列上建立了复合索引。索引中的数据行按照state/city/zip次序排列，

因此它们也会自动地按照state/city和state次序排列。这意味着，即使你在查询中只指定了state值，

或者指定state和city值，MySQL也可以使用这个索引。因此，这个索引可以被用于搜索如下所示的数据列组合：

state, city, zip

state, city

state

二：MySQL不能利用这个索引来搜索没有包含在最左前缀的内容。例如，如果你按照city或zip来搜索，

就不会使用到这个索引。如果你搜索给定的state和具体的ZIP代码（索引的1和3列），

该索引也是不能用于这种组合值的，尽管MySQL可以利用索引来查找匹配的state从而缩小搜索的范围。

如果你考虑给已经索引过的表添加索引，那么就要考虑你将增加的索引是否是已有的多列索引的最左前缀。

如果是这样的，不用增加索引，因为已经有了(例如，如果你在state、city和zip上建立了索引，那么没有必要再增加state的索引)。

三：EXPLAIN SELECT \* FROM student WHERE cid=1 AND name='小红';

没有错，而且复合索引中的两个索引字段都能很好的利用到了！因为语句中最左面的name字段进行了等值匹配，所以cid是有序的，也可以利用到索引了。

你可能会问：

我建的索引是(name,cid)。

　而我查询的语句是cid=1 AND name='小红'; 我是先查询cid，再查询name的，不是先从最左面查的呀？

好吧，我再解释一下这个问题：首先可以肯定的是把条件判断反过来变成这样 name='小红' and cid=1; 最后所查询的结果是一样的。

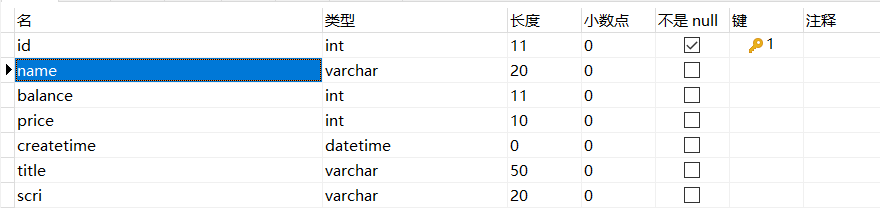
那么问题产生了？既然结果是一样的，到底以何种顺序的查询方式最好呢？

所以

而此时那就是我们的mysql查询优化器该登场了，mysql查询优化器会判断纠正这条sql语句该以什么样的顺序执行效率最高，最后才生成真正的执行计划。

所以，当然是我们能尽量的利用到索引时的查询顺序效率最高咯，所以mysql查询优化器会最终以这种顺序进行查询执行。

表结构：

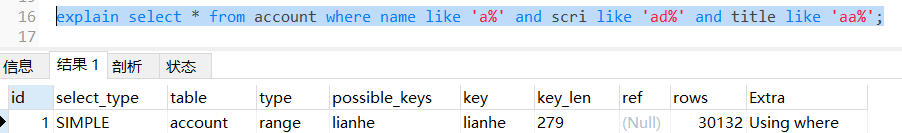


建立索引

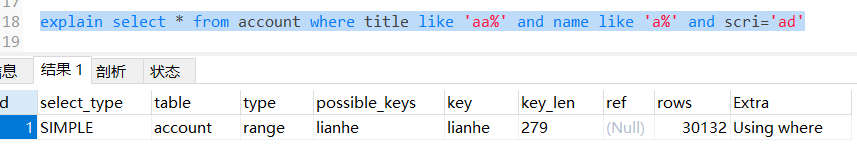
alter table account add index lianhe(name,title,scri)

演示效果：

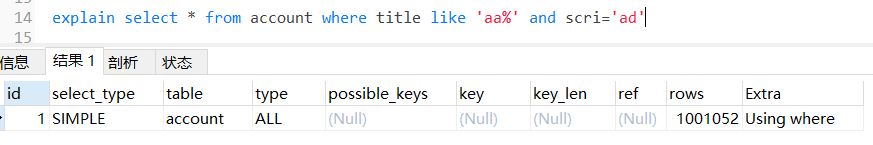
一：使用到了复合索引的全部，虽然我们查询的时候是按照 name scri title ，索引的顺序不是这样的，但是mysql优化器可以帮助我们优化



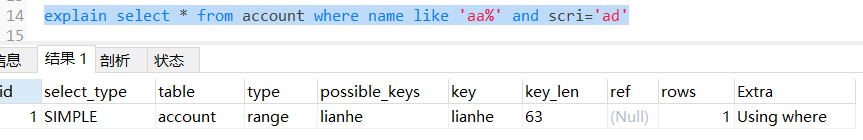
二：使用到了复合索引的全部



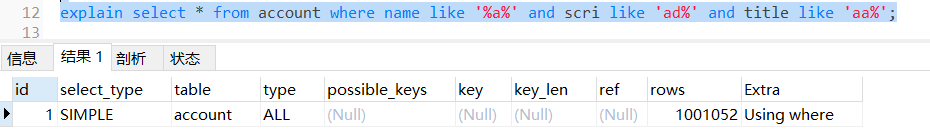
三 没有任何索引，没有符合最左前缀



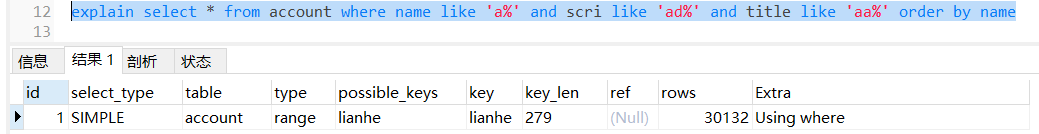
四：使用到了复合索引的一部分（name）



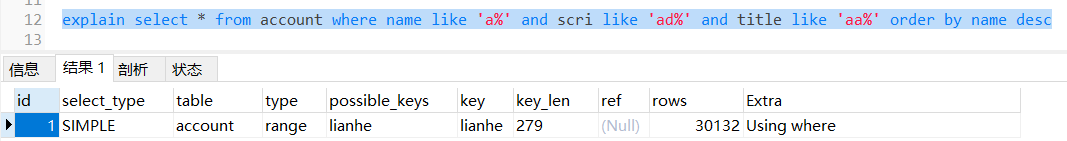
五：没有符合通配符的原则



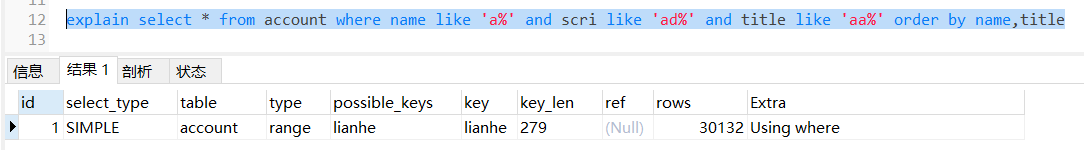
六：



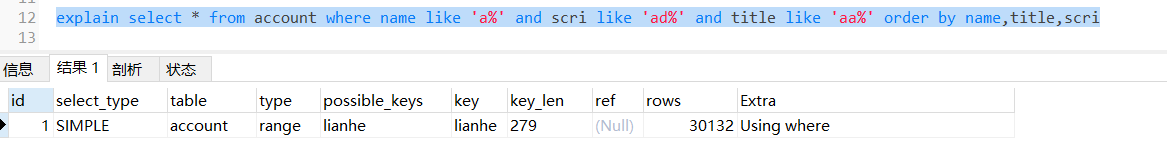
七：



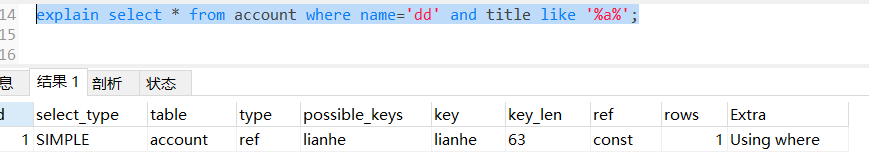
八：



九：

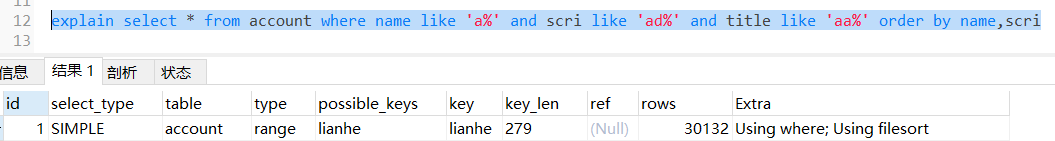


十 只使用了name

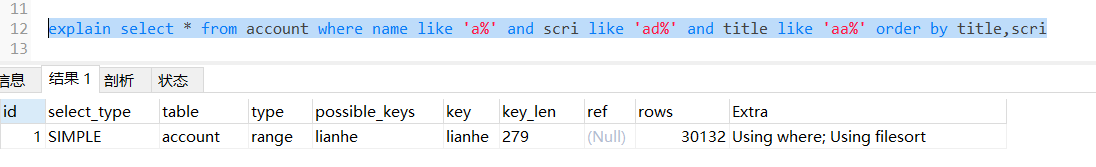


从十到十一可以看到，order by关键字后面，要按照索引的顺序进行，否则在排序的时候要使用到filesort

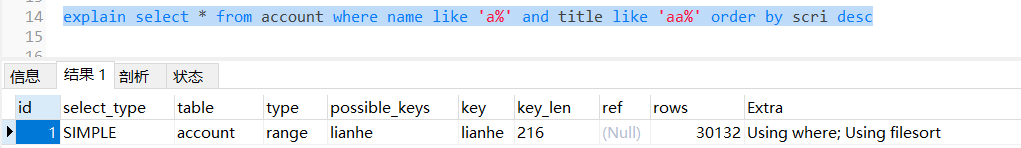
十 使用到了filesort排序



十一：使用到了filesort排序



十二：使用到了filesort排序

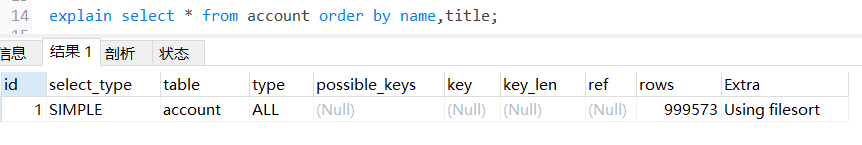


##### 1.7.6.4 升降序不一致，无法使用索引排序

###### 组合索引 即使是升降序一致也不能使用索引

alter table account add index name(name)

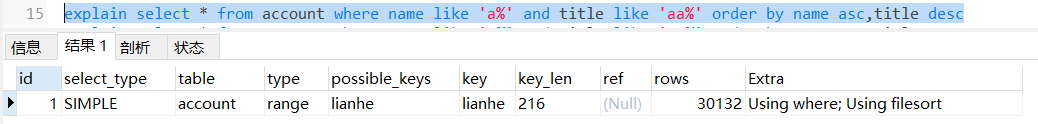
alter table account add index title(title)



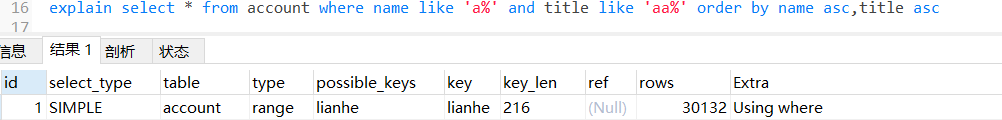
###### 复合索引 升降序一致能使用索引

alter table account add index lianhe(name,title,scri)

十三：索引升降序不一致，导致filesort

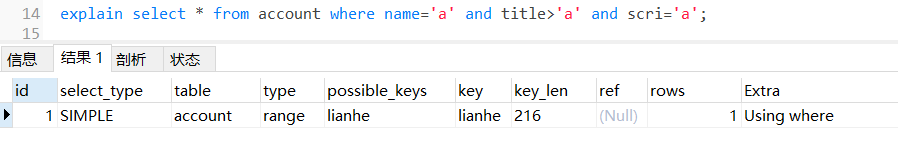


十四：索引升降序一致

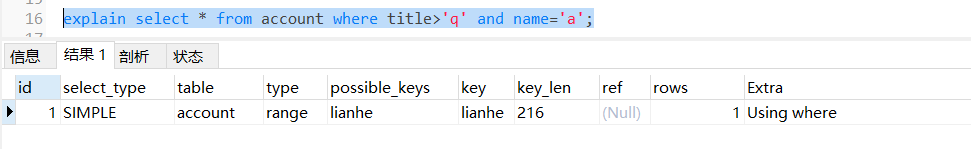


##### 1.7.6.5 > < like的使用不当

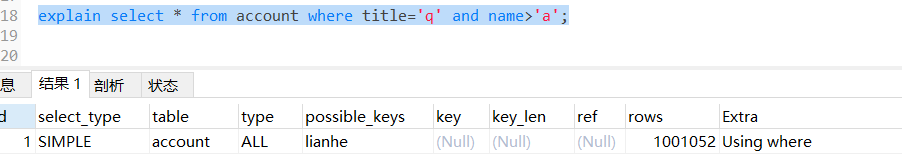
使用到了 name 与title,只要碰到了》 《 like 就停止使用索引

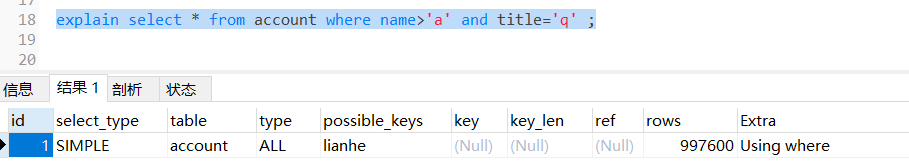


使用到了 name 与title



如果是》《 出现在复合索引的第一个就不会使用索引



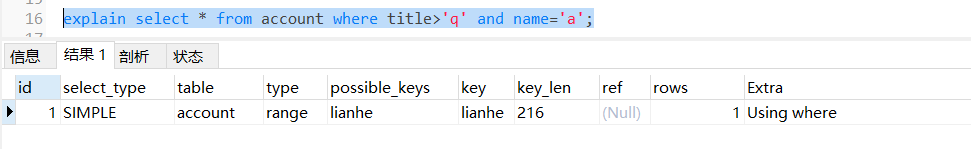


下面是对in的使用：

explain select \* from account where title in ('q','a') and name='a';

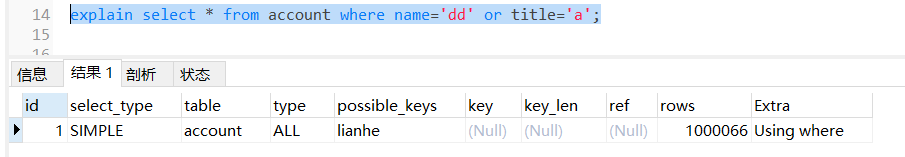
explain select \* from account where name in('a','dd') and title='q' ;

上面两名出现的结果是：

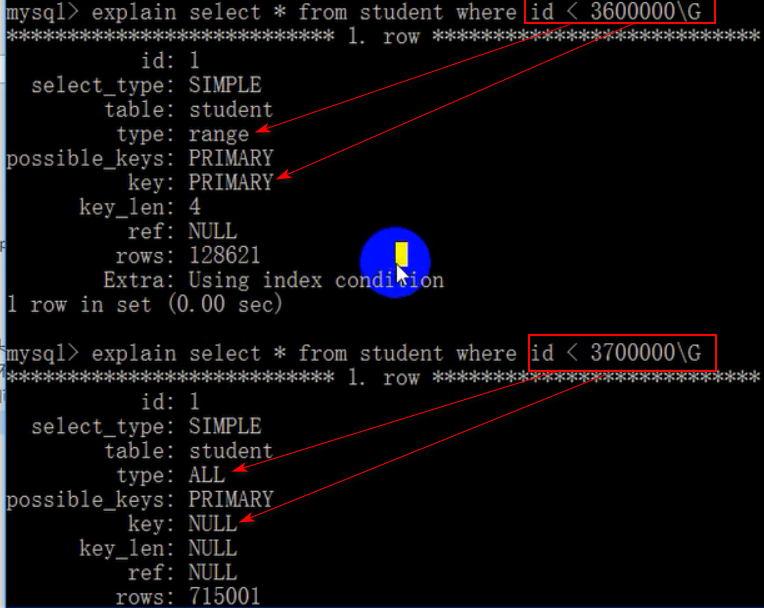


##### 1.7.6.6 or两边都要是有索引

Cond or cond2 两边必须都要有索引



##### 1.7.6.7 当一个索引匹配大量数据时，不一定使用索引



#### 1.7.7 前缀索引

Index(field(10)) 使用字段field的前10个字符建立索引

前缀的标识度足够，需要使用前缀索引

## 1.8 查询缓存

缓存的key 是select语句，value 是查询到的数据集

###### 配置位置

My.ini/my.cnf(linux)

配置在mysqld选项中

###### 查看设置的值：

show variables like '%query\_cache%';

###### query\_cache\_type的几种情况

query\_cache\_type: 是否开启缓存功能，取值为ON（默认值）, OFF, DEMAND，默认值为ON  
　　- 值为OFF或0时，查询缓存功能关闭；  
　　- 值为ON或1时，查询缓存功能打开，SELECT的结果符合缓存条件即会缓存，否则，不予缓存，显式指定SQL\_NO\_CACHE，不予缓存；  
　　- 值为DEMAND或2时，查询缓存功能按需进行，显式指定SQL\_CACHE的SELECT语句才会缓存；其它均不予缓存

query\_cache\_size 缓存量的大小，也是配置在mysqld的选项中

###### 清空缓存

Reset query cache

###### 缓存出现的一些问题

缓存失效问题

当数据表改动时，基于该数据表的任何缓存，都会失效。

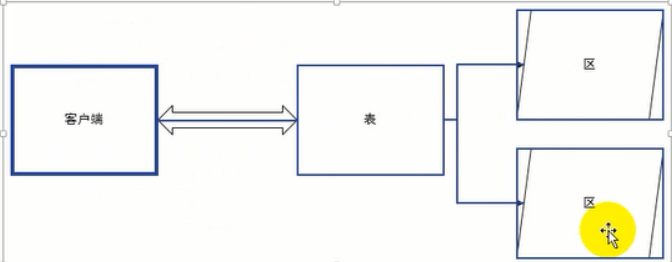
动态数据不能缓存 ，如now()

缓存检索依赖于sql语句的字符串语法规则

缓存检索依赖于sql语句的字符串语法规则 ，如大小写不一样，空格等。都会造成新的缓存。

## 1.9分区

innodb\_file\_per\_table=1 可以控制，innodb表的数据是放在一个文件中存储，还是放在不同的文件中存储。1 代表放在一个个文件当中进行存储，0代表放在一个文件中进行存储。



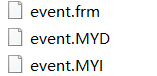
### 1.9.1 把一个表分布到5个partition当中的案例

将一张表中的数据和索引，分散到不同的文件中进行存储，杀身之祸为分区操作，划分出来的文件就是不同的分区

Innodb一个表分为下面的两个文件



Myisam 一个表分为下面的三个文件

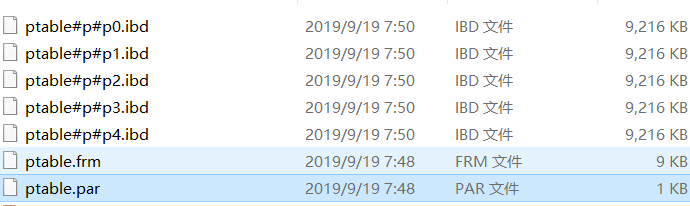


分区之后 ，每一个文件就小了

下面根据id将数据分区到5个partition当中



表结构如下



Hash(id) 算法，是最常用的，最普通 的分区算法，逻辑上表示 将记录均匀分布到不同的区域中，当数据表中的数据量很多时，同时没有明显的逻辑区分，使用该算法，如果是电话费详单，它就有明显的数据逻辑区分如：201906 201905。

### 1.9.2 几个分区算法

#### 1.9.2.1 hash 算法（求余分配）

HASH分区的特点

1根据MOD(分区键，分区数)的值把数据行存储到表的不同分区中

2数据可以平均的分布在各个分区中

3 HASH分区的键值必须是一个INT类型的值，或是通过函数可以转为INT类型

Hash:一类算法的总称，如md5,sha1算法。相同的输入有相同的输出。

Hash-table:哈希表，特殊的数据结构，关联数组即是哈希表，key就是输入，value就是输出

Hash 分区算法，在业务逻辑层面，表示均匀分配。

|  |
| --- |
| CREATE TABLE `ptable` (  `id` int(10) unsigned NOT NULL AUTO\_INCREMENT,  `subject` varchar(20) DEFAULT NULL,  `content` varchar(20) DEFAULT NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8  PARTITION BY HASH(id) PARTITIONS 5  或者是这样  CREATE TABLE `customer\_login\_log` (   `customer\_id` int(10) unsigned NOT NULL COMMENT '登录用户ID',   `login\_time` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE  ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='用户登录日志表'  PARTITION BY HASH(UNIX\_TIMESTAMP(login\_time)) PARTITIONS 4; |

可以直接利用程序进行插入数据：

|  |
| --- |
| **try**:  **with** connection.cursor() **as** cursor:  *# Create a new record* sql = **"INSERT INTO ptable(subject,content) VALUES (%s, %s)"** mylist = []  **for** x **in** range(0,10000):  *#name* myname = **'特别是在nopqrDEFG生成nopqrDEFG任务的时候abcdefghijklIJKLMNOPQRSTUVWXYZ都需vwsyzABCDE要一个唯一标识字符'** mynamestart = random.randint(0,65)  mynameend = mynamestart+4  subject = myname[mynamestart:mynameend]   mytitle = **'特别是在nopqrDEFG生成nopqrDEFG任务的时候abcdefghijklIJKLMNOPQRSTUVWXYZ都需vwsyzABCDE要一个唯一标识字符'** mynamestart = random.randint(0, 66)  mynameend = mynamestart + 3  content = mytitle[mynamestart:mynameend]   mylist.append((subject,content))  cursor.executemany(sql, mylist)  connection.commit(); **finally**:  connection.close() |

#### 1.9.2.2 key算法（求余分配）

也是一个hash算法，更加通用的hash 算法，在hash中，仅仅可以 针对整数进行求余运算。而key算法中，key可以是一个非整数

需要注意的一点：该key必须是主键，或者是主键的一部分：

creat table ptable02(

id int unsigned auto\_increment,

subject varchar(20),

content varchar(20),

primary key (id,subject)

)charset utf8 engine innodb

partition by key(id,subject) partitions 5;

// primary key(sid,cid)是联合主键，就是说通过字段sid和cid来确认一条记录的唯一性。

|  |
| --- |
| **try**:  **with** connection.cursor() **as** cursor:  *# Create a new record* sql = **"INSERT INTO ptable02(subject,content) VALUES (%s, %s)"** mylist = []  **for** x **in** range(0,10000):  *#name* myname = **'特别是在nopqrDEFG生成nopqrDEFG任务的时候abcdefghijklIJKLMNOPQRSTUVWXYZ都需vwsyzABCDE要一个唯一标识字符'** mynamestart = random.randint(0,65)  mynameend = mynamestart+4  subject = myname[mynamestart:mynameend]   mytitle = **'特别是在nopqrDEFG生成nopqrDEFG任务的时候abcdefghijklIJKLMNOPQRSTUVWXYZ都需vwsyzABCDE要一个唯一标识字符'** mynamestart = random.randint(0, 66)  mynameend = mynamestart + 3  content = mytitle[mynamestart:mynameend]   mylist.append((subject,content))  cursor.executemany(sql, mylist)  connection.commit(); **finally**:  connection.close() |

#### 1.9.2.3 range算法

是一种范围条件分区算法，不仅要指定分区数量，还要指定分区范围。

select UNIX\_TIMESTAMP('2017-05-03 23:59:59'); 可以得到时间撮

create table ptable03(

id int unsigned auto\_increment,

subject varchar(20),

content varchar(20),

pubtime int,

primary key (id,pubtime)

)charset utf8 engine innodb

partition by range(pubtime)(

partition p1 values less than (1493827199),

partition p2 values less than (1493927199),

partition p3 values less than (1494027199)

);

需要注意的两点：

该key必须是主键，或者是主键的一部分：

只有less than 语法



#### 1.9.2.4 list算法

create table ptable04(

id int unsigned auto\_increment,

subject varchar(20),

content varchar(20),

pubtime int,

status TINYINT,

primary key (id,status)

)charset utf8 engine innodb

partition by list(status)(

partition p1 values in (1,2),

partition p2 values in (3)

);

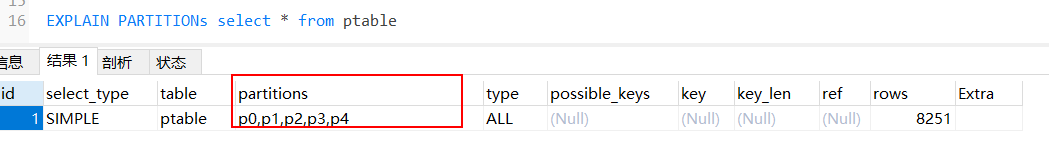
### 1.9.3 分区管理方法

1 Key hash 可以修改分区数量

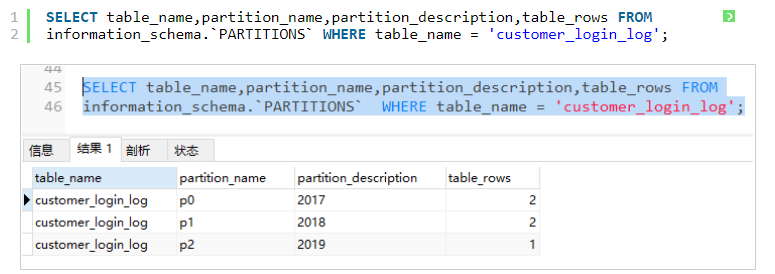
2 List range可以删除和新增分区

3 Show table status 这个语句可以查看到很多信息（有时间研究一下字段的意义）

4 EXPLAIN PARTITIONS SELECT 查看select语句怎样使用分区



5 查看分区表的情况



#### 1.9.3.1 list range 添加分区

alter table members add partition (

partition p3 values less than (1960),

partition p4 values less than (1970)

);

#### 1.9.3.2 list range 删除分区

alter table tr drop partition p3;

#### 1.9.3.3 key hash 增加分区数量

Alter table tablename add partition partitions 4; 增加4个分区

#### 1.9.3.4 key hash 减少分区数量

Alter table tablename **COALESCE partition 6 减少6个分区**

### 1.9.4 分区的使用

在客户端程序不变的情况下，将服务器端的数据分布到不同的物理文件中，进而做到提供数据表的处理能力。

#### 1.9.4.1分区使用的注意细节

1结合业务场景选择分区键，避免跨分区查询

2对分区表进行查询最好在WHERE从句中包含分区键

#### 1.9.4.2分区的优点

1分区，可以将分区文件，部署到不同的磁盘上，充分利用磁盘的性能和更大的查询吞吐量。

2根据查找条件，也就是where后面的条件，查找只查找相应的分区不用全部查找了

3可以通过删除分区删除相应的数据。

## 2.0分表

#### 2.0.1 分表与分区的区别

1 分区是把数据在数据服务器层(mysql)把数据进行划分，对应用层是透明的。而分表是建立一个一个相同的表，在应用层进行的划分，由程序决定把数据放在哪个表当中。

|  |
| --- |
| CREATE TABLE `table\_01` (  `id` int(10) unsigned NOT NULL AUTO\_INCREMENT primary key,  `use\_id` int unsigned,  `amount` decimal(10,2),  `billtime` int  ) ENGINE=InnoDB DEFAULT CHARSET=utf8  CREATE TABLE `table\_02` (  `id` int(10) unsigned NOT NULL AUTO\_INCREMENT primary key,  `use\_id` int unsigned,  `amount` decimal(10,2),  `billtime` int  ) ENGINE=InnoDB DEFAULT CHARSET=utf8 |

2数据库5.1 之后 才支持的分区

1. 数据的减压
2. 限制于分区的语法限制

对于分表，查询还好说一点，对于插入数据，怎样在各个表当中使用不一样的主键呢？

1 使用mc、redis 的自增方法先创建一个主键，再使用。

2 uuid

## 2.1 读写分离配置

Mysql的横向拓展是根本的解决手段。

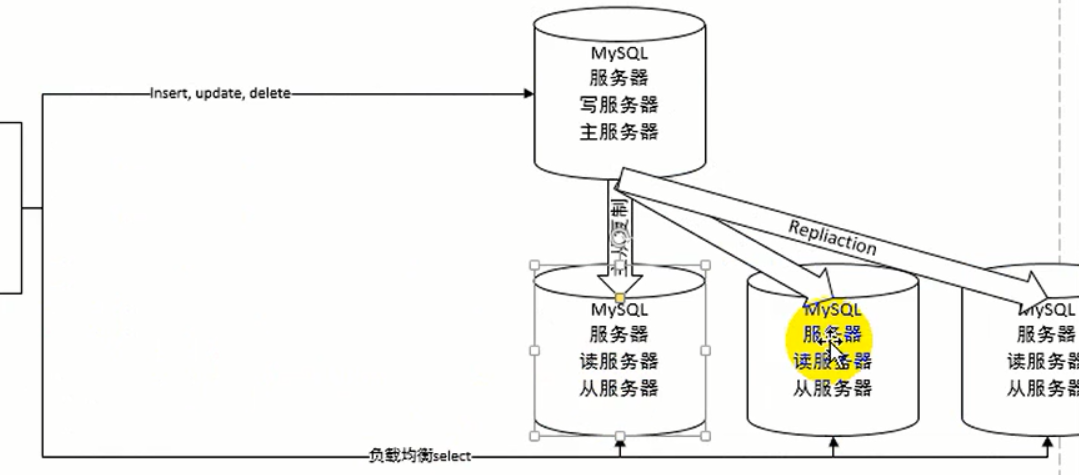
比较几个典型的概念是：

读写分离、负载均衡

## 2.2 负载均衡(load balance)

当存在多台服务器提供一个任务时，需要从其中选择某台完成当前任务，这个选择的过程就是负载均衡。

在mysql中架构中，往往采用一写多读的模式。



负载均衡的几个算法：

1. 轮询： 一个一个来

2 加载（重）： 依据能力来

1. 负载分配： 依据当前的空闲状态，不常用（因为，判断负载比较的复杂）

## 2.3高可用

## 2.4 典型的SQL写法

### 2.4.1 DDL操作

在线上服务器上，执行DDL，更新表的结构，这个操作，需要谨慎处理。

在5.6 之前 DDL 结构更改 是这样操作的

任务：增加一个列，或者增加一个索引，

并不是直接执行 alter table

而是采用下面的方式：

1. 创建一个新表满足新的结构
2. 将旧表数据导入到新表当中（读操作），它是逐条导入，保证一次性锁定的内容少，同 时表上可以执行其它任务。
3. 导入的过程中，记录表上的新更新操作，以日志的形式完成记录
4. 导入完毕，将更新新日志，在新表重做一遍
5. 新表去替换旧表即可。

而在5.6 之后 ，有了online ddl，大大缩短了锁定时间。在执行alter的时候，可以读数据，写数据没有测试过。

### 2.4.2 往新表中导入大量旧数据

#### 1 技巧一

导入时先禁用索引和约束，导入完成后，再开户索引和约束，一次性创建索引。

Alter table table\_name disable keys;

Alter table table\_name enable keys;

#### 2 技巧二

对于innodb存储引擎，可以将多个sql,放在一个事务中完成，

Begin:

Commit;

#### 3技巧三

使用preparestatement，减少编译次数

#### 4 技巧四

尽量选择自己需要的字段，不要select \*

#### 5技巧五

不要使用 rand() 进行排序



问题在于生成了大量的随机数，每条数据都生成一个随机数，但是我们只要其中的五条。

解决的方案，我们在外部随机生成5个主键，在数据库查询就可以了。

#### 6技巧六

Myisam 当中有一个计数器 count(\*) 它会很快返回记录数，但是innodb没有一个数值来记录这个count(\*)，比较消耗时间。

解决的方案，但是这个不常用。

|  |  |  |
| --- | --- | --- |
| Id | Table\_name | count\_num |

Count(\*) count(1) 是一样的。

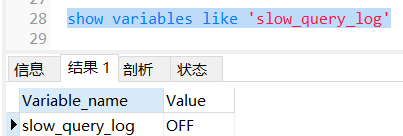
Count(字段) 是统计字段不为null 条数

#### 7技巧七

如果可以确定公公检索 一条，建议加上limit 1

## 2.5 慢查询日志

#### 1 首先确定是否开启

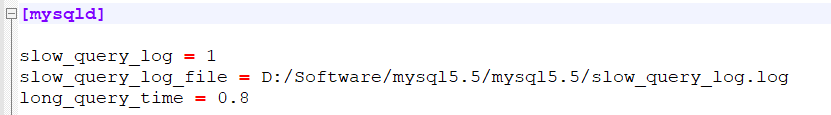


#### 2 进行配置

Slow\_query\_log ：1（ON）代表开启， 0(OFF) 代表未开启

Slow\_query\_log\_file 指定慢查询日志文件

Long\_query\_time 指定时间，单位秒



## 2.6 profile 信息

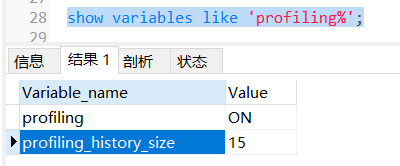
#### 作用

详细记录SQL的执行时间的工具。

开启profile,执行SQL 自动记录，读取SQL形成profile信息

#### 开启

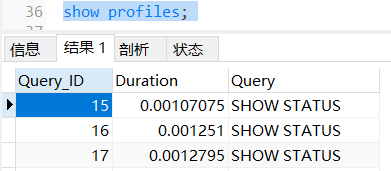
查看是否开启



set profiling=ON;

set profiling\_history\_size=30

#### 查看所有的执行情况



#### 查看单个详细的执行情况：



## 2.7 典型的配置

[mysqld]下：

#### 1 慢查询日志

slow\_query\_log = 1

slow\_query\_log\_file = D:/Software/mysql5.5/mysql5.5/slow\_query\_log.log

long\_query\_time = 0.8

#### 2 innodb每个表的数据，可以单独进行存储，1(单独存储) 0（不进行单独存储）

innodb\_file\_per\_table=1

#### 3 默认的存储引擎 default-storage-engine

default-storage-engine=INNODB

#### 4 最大连接数 max\_connections

max\_connections=100

#### 5 查询缓存 query\_cache\_size

query\_cache\_size=127108864

query\_cache\_type=2

（0：不开启，1 显式指定SQL\_NO\_CACHE，不予缓存，2 显示指定SQL\_CACHE 进行缓存）

#### key\_buffer\_size

索引缓存的大小

key\_buffer\_size 指定用于索引的缓冲区大小，增加它可得到更好处理的索引(对所有读和多重写)，到你能负担得起那样多。如果你使它太大，系统将开始换页并且真的变慢了。

对于内存在4GB左右的服务器该参数可设置为384M或512M。

通过检查状态值Key\_read\_requests和Key\_reads，可以知道 key\_buffer\_size 设置是否合理。

比例key\_reads / key\_read\_requests应该尽可能的低，至少是1:100，1:1000更好(上述状态值可以使用SHOW STATUS LIKE ‘key\_read%’获得)。

注意：该参数值设置的过大反而会是服务器整体效率降低!

测试服务器环境：内存4G　数据库MySQL5.6系统配置文件/etc/my.cnf中 key\_buffer\_size ＝512M，监测 key\_buffer\_size 设置是否合理，是否需要优化。

一、多大算合适 :

mysql> show status like 'key\_read%';

+------------------------+------------+

| Variable\_name         | Value      |

+------------------------+------------+

| Key\_read\_requests      | 3633676486 |

| Key\_reads            | 739392    |

+------------------------+------------+

key\_reads / key\_read\_requests = 1:4914　，表明 key\_buffer\_size ＝512M 设置很合理，无需修改。

Key\_read\_requests：从缓存读取索引的请求次数。  
Key\_reads：从磁盘读取索引的请求次数。

二、如何修改

vi /etc/my.cnf 配置文件，[mysqld] 下

key\_buffer\_size ＝512M

别忘了需mysql重启 service mysql restart 或 /etc/rc.d/init.d/mysql restart 后才生效！