# 索引

## 前缀索引

建立索引关键字一种方案。

通常会使用字段的整体作为索引关键字。

有时，即使使用字段前部分数据，也可以去识别某些记录。

语法：

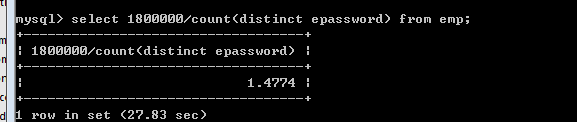
Index `index\_name` (`index\_field`(N))

使用index\_name前N个字符建立的索引。

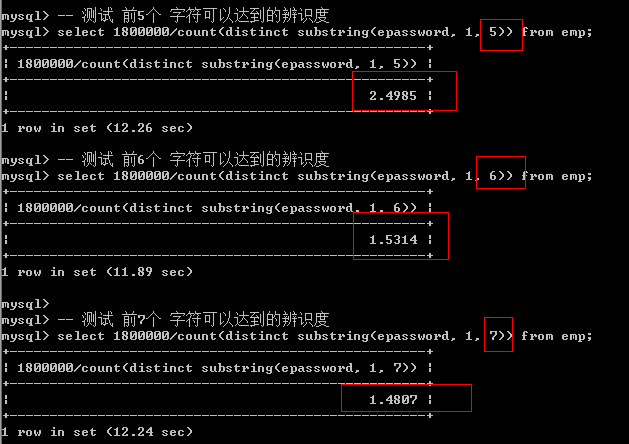
N，究竟是多少？

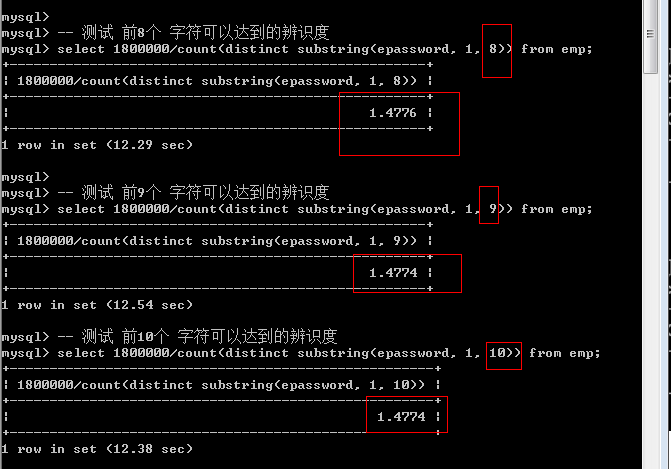
使用N长度所达到的辨识度，极限接近于使用全部长度的辨识度即可！

计算最大的辨识度：

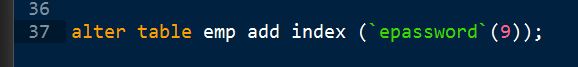


依次取得前N个字符，进行对比，找到极限接近的：





可见，9 时，已经极限接近，提高长度，不能明显 提升辨识度，因此可以使用前9个字符：



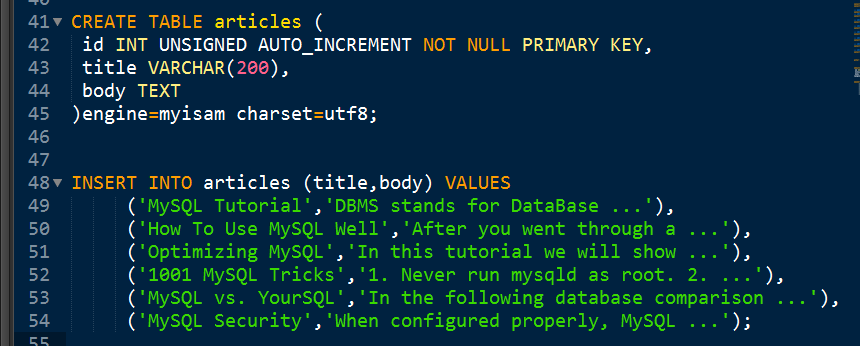
Tip：前缀索引不能用于索引覆盖！

## 全文索引

该类型的索引特殊在：关键字的创建上。

为了解决 like ‘%keyword%’这类查询的匹配问题。

测试：



业务：

查询 标题或者内容包含 database 关键字的记录。

形成的SQL如下：

Select \* from articles where title like ‘%database%’ or body like ‘%database%’;

此时不能建立普通索引，查询不符合 左原则，建立了也使用不了。

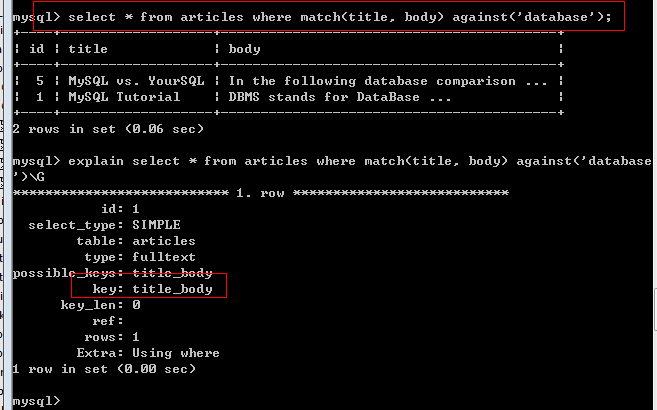
此时 全文索引就可以其作用了：



直接使用上面的SQL：

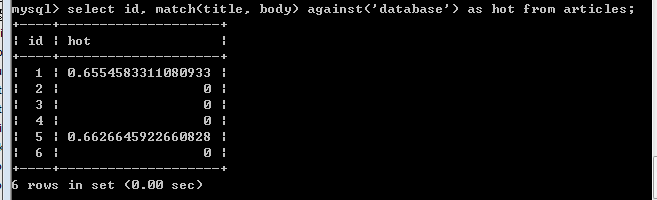
需要使用特殊的全文索引匹配语法才可以生效：

Match() against();

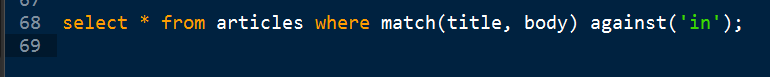


Tip: 该MYSQL提供的全文索引，不能对中文起作用！

Match() against() ,返回的关键字的匹配度（关键字与记录的关联程度）。



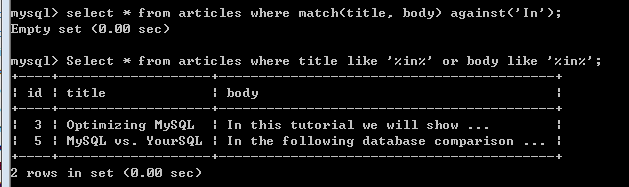
停止词：



发现in这个词，是不能被全文索引所检索到的。

思考：与 like %in% 是否相同？b

不同。



原因何在呢？

全文索引索引的的关键字，不是整个字段数据，而是从数据中提取的关键词。

## 索引的数据结构-了解

Hash

### B-Tree（B树）

两种数据结构。指的是mysql存储索引所采用的数据结构。其中，用户所维护的所有的索引结构 B-Tree结构

B-Tree的结构如下：

每个节点，存储多个关键字。

关键字也会对应记录地址

以上设计为了解决，一次性磁盘IO开销，可以读取到更多的关键字数量。

每个关键字之间，存在子节点指针



如果是复合索引：

关键字的排序先排左侧字段，在左侧字段相同的情况下，再排序右侧字段：

### 聚集索引，聚簇索引

B+Tree(B-Tree的变种)

在innodb的存储引擎上，主索引是与数据记录 存储在一起的（聚簇在一起的）。



带来的问题：

Innodb的其他索引，非主键索引（二级索引）：

关键字对应的不再是记录的地址，而是记录的主键。



可见，检索需要 二次检索。先检索到主键ID，在检索记录。

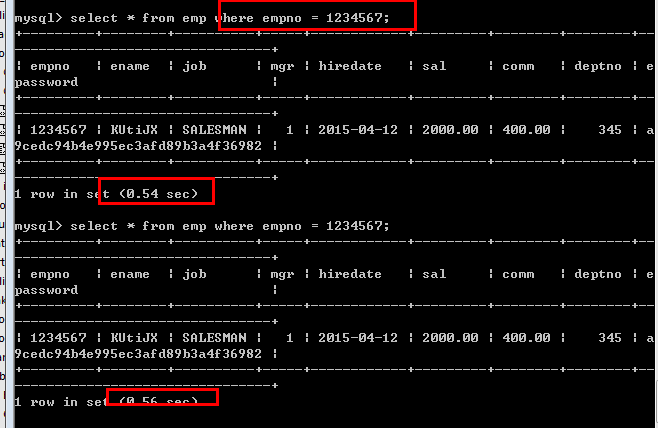
# 查询缓存 query\_cache

将select的结果，存取起来共二次使用的缓存区域：



MySQL提供的缓存区：

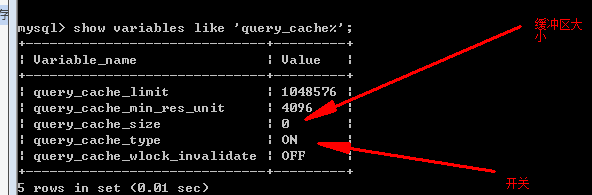
未开启前：



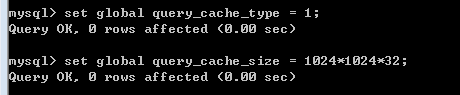
两次查询时间消耗一致。

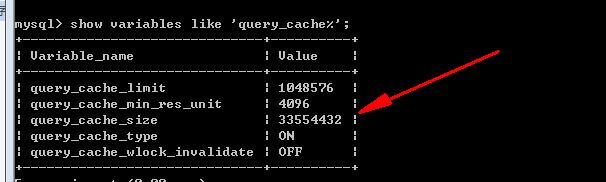
开启查询缓存：

通过变量控制：

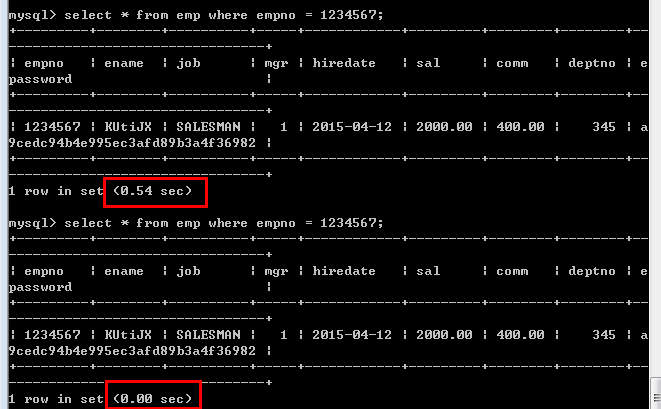


开启并设置大小：





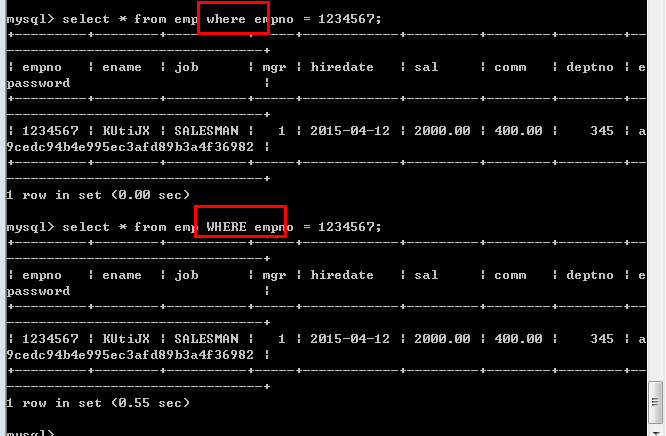
再次执行查询：



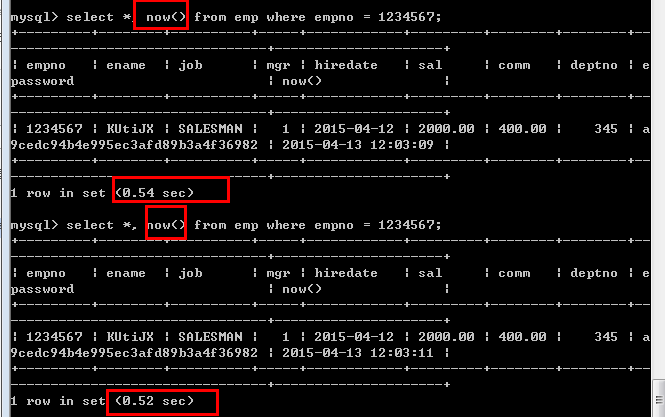
可见，第二次查询，使用了开启的缓存！

注意事项：

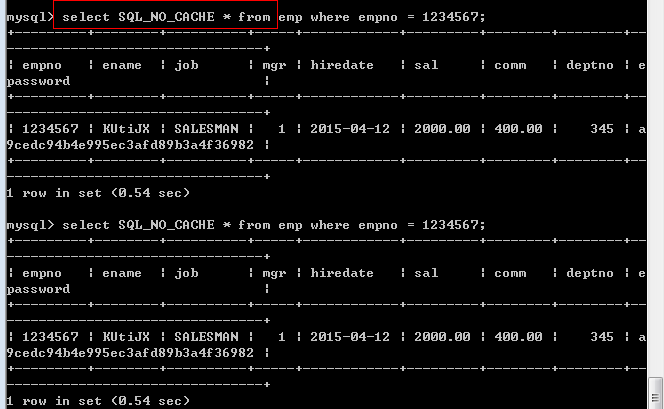
1. 查询缓存存在判断是严格依赖于select语句本身的：严格保证SQL一致



1. 如果查询时包含动态数据，则不能被缓存。



1. 一旦开启查询缓存，MySQL会将所有可以被缓存的select语句都缓存。如果存在不想使用缓存的SQL执行，则可以使用 SQL\_NO\_CACHE语法提示达到目的：



# 分表

## 分区，partition

分表，水平分表，分裂

将某张表数据，分别存储到不同的区域中。



其实：每个分区，就是独立的表。都要存储该分区数据的数据，索引等信息。

创建分区：

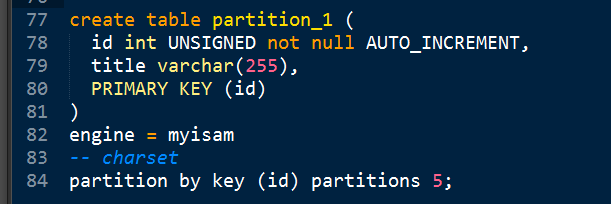
在创建表时，指定分区的选项：

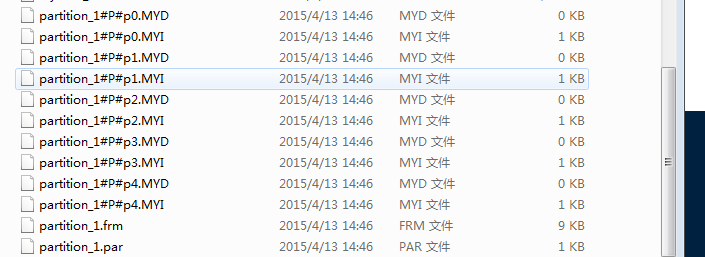
Create table table\_name (

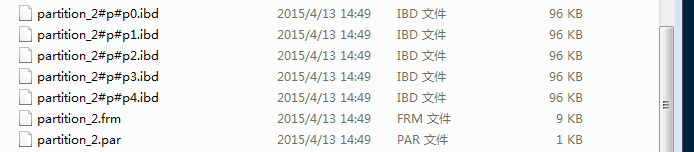
定义

)

Partition by 分区算法 (参数) 分区选项。

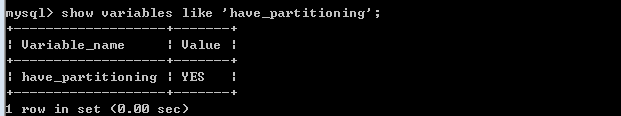






Tip：分区与存储引擎无关，是MySQL逻辑层完成的。

通过变量查看当前mysql是否支持分区：



分区算法：

MySQL提供4种

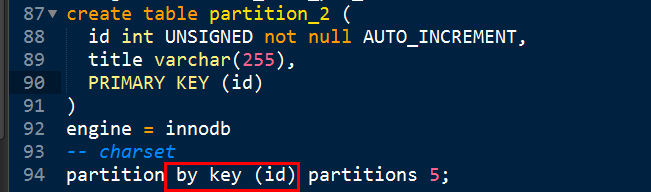
取余：Key，hash

条件：List，range

提示，参与分区的参数字段需要为主键的一部分。

### KEY – 取余

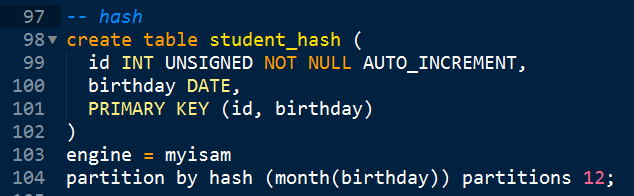
按照某个字段进行取余



### Hash – 取余

按照某个表达式的值进行取余

学生表分区，按照生日的月份，划分到12个表中。



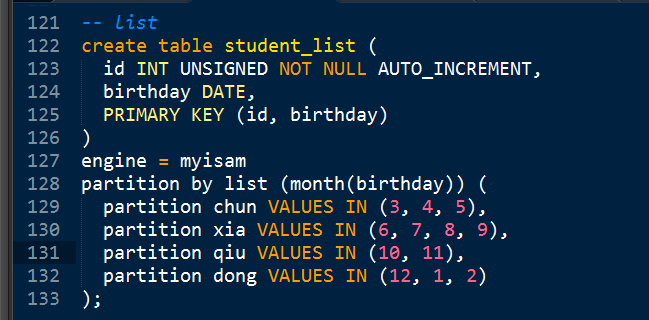
Tip：

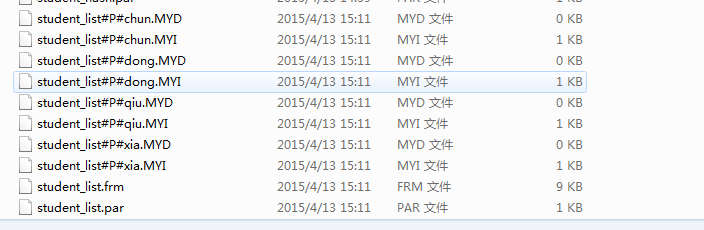
Key，hash都是取余算法，要求分区参数，返回的数据必须为整数。

### List – 条件 – 列表

需要指定的每个分区数据的存储条件。

按照生日中的月份，分成春夏秋冬四个分区。



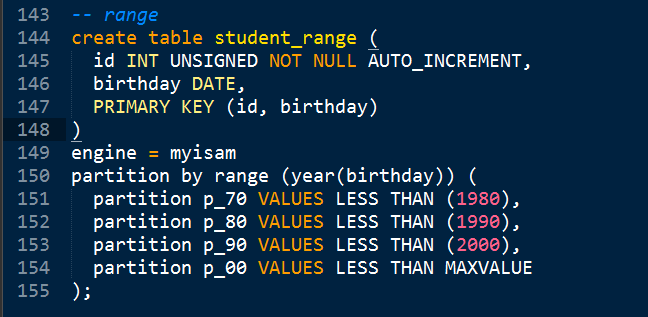


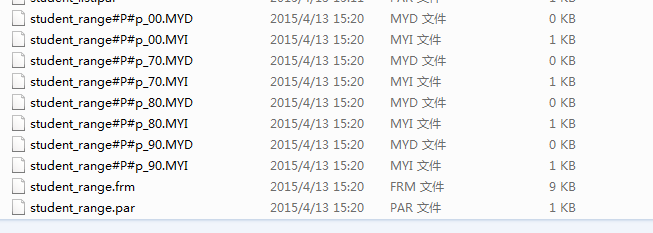
List，条件依赖的数据时列表形式。

### Range - 条件 – 范围

Range：条件依赖的数据是一个条件表达式。

逻辑：按照生日的年份分成



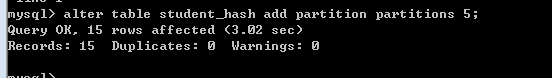


### 管理分区语法

#### 取余：key，hash

增加分区数量：

Add partition N



减少分区数量：

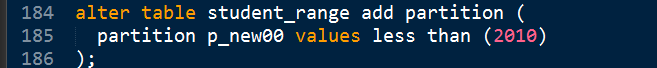
COALESCE partition N



采用取余算法的分区数量的修改，不会导致已有分区数据的丢失，需要重新分配数据到新大地分区。

#### 条件：list，range

添加分区



删除分区：

Drop partition partition\_name;



注意：删除条件算法的分区，导致分区数据丢失。

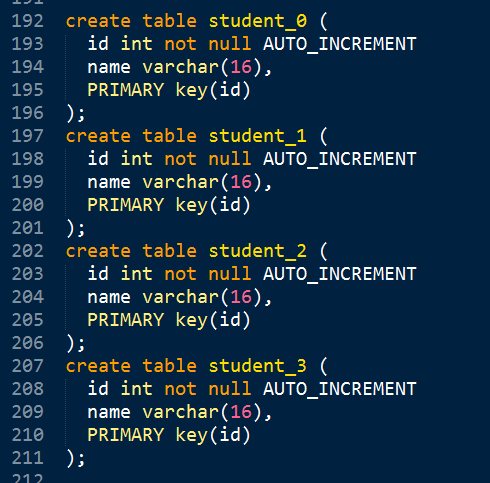
### 选择分区算法

平均分配: 就按照主键进行key(primary key)即可(非常常见)

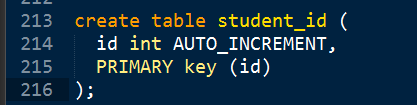
按照某种业务逻辑分区： 选择那种最容易被筛选的字段，整数型

## 分表，水平分表

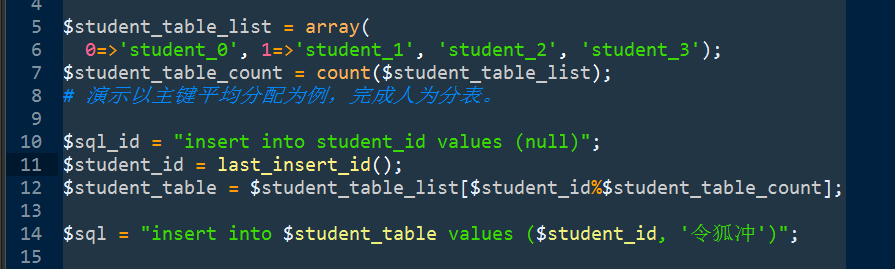
创建结构相同的N个表；



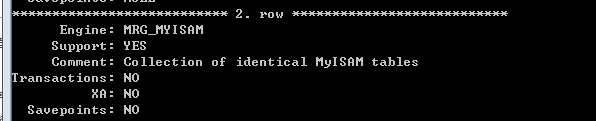
再创建用于管理学生ID的表：



PHP客户端逻辑：



## Merge，mrg\_myisam



MySQL提供一个可以将多个结构相同的myisam表，合并到一起的存储引擎：



## 垂直分表

表中存在多个字段。

常用字段 - 非常有字段

主要目的，减少每条记录的长度。

例如学生表可以分成：

基础表和额外表，两张表中记录为1:1的关系。

基础信息表

Student\_base

Id name age

额外信息表

Student\_extra

Id 籍贯 政治面貌

# 架构层面

不仅仅是用一台MySQL

主从复制：

Mysql服务器内部支持复制功能，仅仅需要通过配置完成下面的拓扑结构。

一主多从典型结果：



读写分离，负载均衡



上面的架构可以提升整体服务器的效率，高性能。

同时，服务器架构需要保证 高可用，7x24不宕机。



# SQL

完成功能。

并发性的SQL

少用（不用）多表操作（subquery，join）

将复杂的SQL拆分多次执行。

商品，分类：

Select \* from category;分类列表

Select cat\_id, count(\*) from goods group by goods;分类对应的商品数量。

如果查询很原子（很小），会增加查询缓存的利用率

大量数据的插入

多条 insert

Load data into table

建议，先关闭约束及索引，完成数据插入，再重新生成索引及约束。

针对于myisam：

Alter table table\_name disable keys; 禁用索引约束

大量数据插入

Alter table table\_name enable keys; 启用

针对innodb：

Drop index, drop constraint 要保留主键

Begin transaction|set autocommit=0;

[数据本身已经按照主键值排序]

大量的插入

Commit;

Add index, add constraint

Insert into table\_name values ();

Insert into table\_name values ();

Insert into table\_name values ();

Insert into table\_name values ();

PK

Insert into table\_name values (), (), (), (), ();

区分与每条记录的长度，以10量级为单位即可，不要过多。

多次执行相同结构别忘了prepare预编译的执行方式。

分页

Limit offset, size;

Size = 10;

|  |  |
| --- | --- |
| Page | offset |
| 5 | 40, 10 |
| 50 | 490, 10 |
| 5000 | 4990, 10 |
| 500000 | 499990, 10 |

Limit 的使用，会大大提升无效数据的检索（被跳过）。

应该使用条件等过滤方式，将检索到的数据尽可能精确定位到需要的数据上。

例如分页：

Limit size;

Order by Rand()



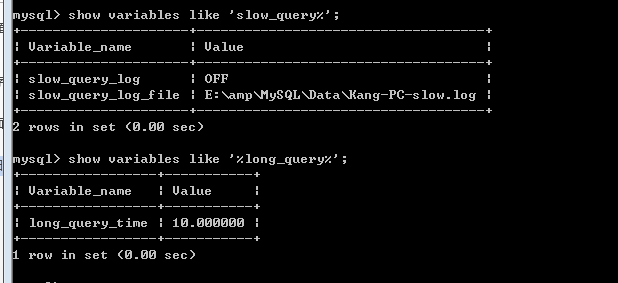
上面的查询，会导致每条记录都执行rand()，成本很高！

建议，通过某种运算，先确定的随机主键，从数据表中获取数据。

例如：先获取最小与最大的学生ID，随机10次找到需要的记录。

# 慢查询日志

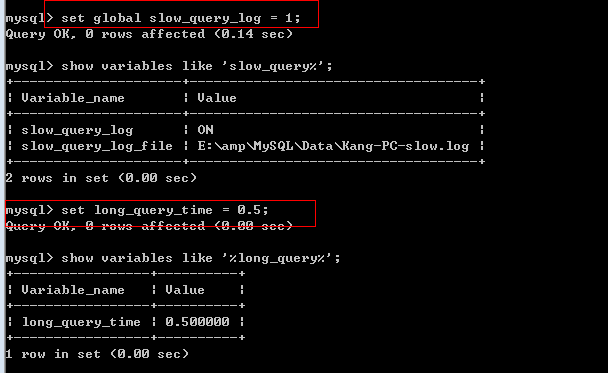
定位执行较慢的查询语句方案。



Slow\_query\_log = 0|1

Long\_query\_time = N 超过该时间临界点，就为慢查询。

开启日志



执行SQL：

查看：

