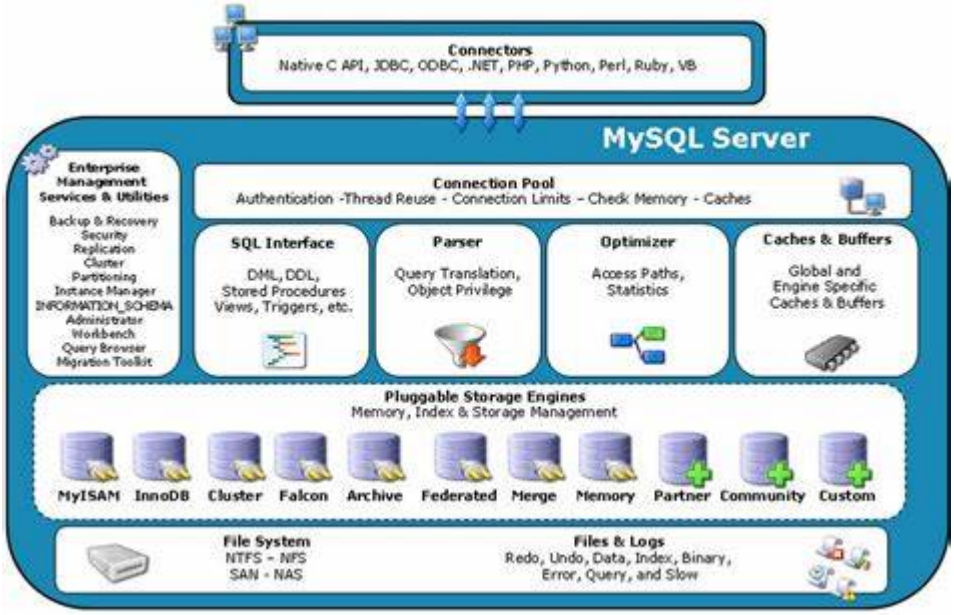
MySQL学习

1.sql的执行原理



- Connectors

  连接、支持多种协议，各种语言

- Management service

  系统管理和控制工具，例如：备份、集群副本管理等

- pool

  连接池

- sql interfaces

  sql接口-接收命令返回结果

- parser

  分析解析器：验证

- optimizer

  优化器：优化sql执行效率

- cache and buffer

  查询缓存

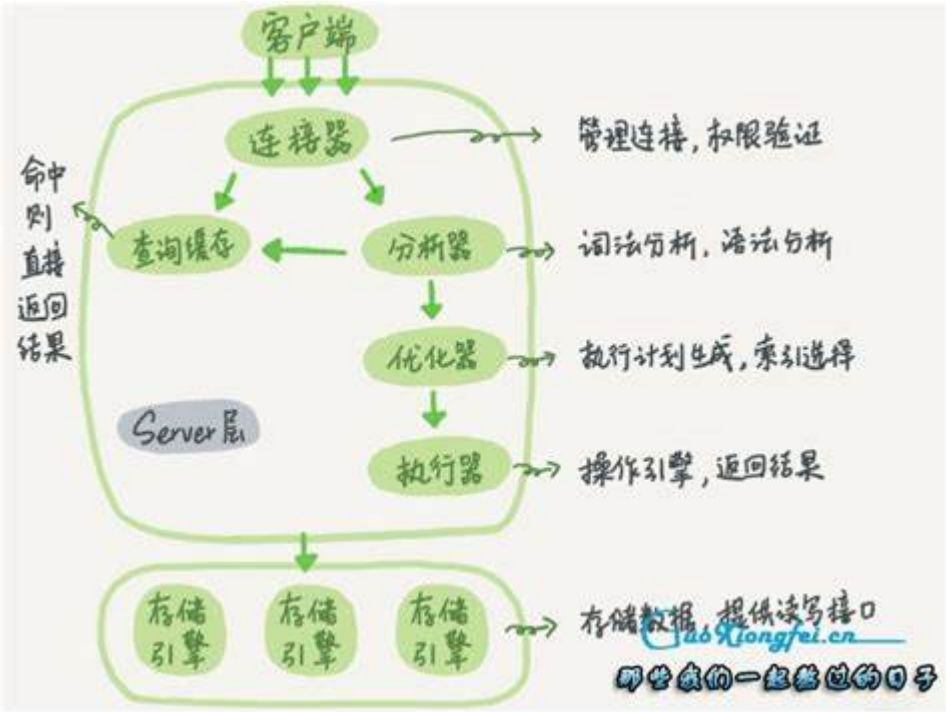
- storage engines

  存储引擎：可插拔

- file system

  文件系统

1.1mysql的基本架构和执行原理



> 两个部分：server层与存储引擎层

- 连接器

  > 建立客户端和服务器连接、权限获取、维持管理连接

  ```
  #mysql -u$user -p$password
  mysql -uroot -p
  ```

  > mysql本质为客户端连接工具,tcp握手成功，需要进行身份认证

  > 查询连接状态：

  > 短链接：执行少数几次查询就会断开，浪费大量资源
  >
  > 长连接：连接成功后，长时间保持连接,wait_timeout默认连接时间

  > 查看默认连接时长

  ```
  show variables like 'wait_timeout';
  ```

  > 查当前连接时长

  ```
  show processlist;
  ```

  长连接驻留内存解决方法

  1、固定时间自动重新连接

  2、mysql_reset_connection 重置连接

- 查询缓存

  > query_cache_type:查询缓存类型

  ```
  show variables like 'query_cache_type';
  ```

  > 如果需要使用，需要设置为DEMAND(按需)

  缓存失效

  > 1、增删改操作导致缓存失效

  mysql8.0已将缓存移除

- 分析器

  词法分析：检测每个单词的含义

  语法分析：对sql语法规则校验，是否满足mysql语法规范

- 优化器

  选择最优解

  索引选择、执行顺序可能影响执行效率，优化器进行最优选择

  ```
  select * from t1 inner join t2 using(ID) where t1.c = 10 and t2.d = 20;
  ```

  如下两个执行顺序：

  1、先找到t1.c=10的所有记录和整个t2表关联，最后筛选t2.d=20的记录

  2、先找到t2.d=20的所有记录和整个t1表关联，最后筛选t1.c=10的记录

- 执行器

  ```
  select * from emp where no = 10;
  ```

  1、表操作权限验证

  2、innodb存储引擎查询第一行，查看no=10则写道结果集，如果不等于10则跳过

  。。。整张表查询完毕

  3、将结果返回给客户端

2.存储引擎

2.1存储引擎简介

> 数据库存储引擎是数据库底层软件组织，数据库管理管理系统（DBMS）使用数据引擎进行创建、查询、更新和删除数据。不同的存储引擎提供不同的存储机制、索引技巧、锁定水平等功能，使用不同的存储引擎，还可以获得特定的功能。MySQL的核心就是存储引擎。
>
> mysql底层设计采用可插拔式的存储引擎，用户根据需求，选择或自定义存储引擎

> mysql5.5之后默认存储引擎为innodb

1、查看mysql可支持的存储引擎

```
show engines;
```

2、各种存储引擎对比

特点	Myisam	NDB	Memory	InnoDB	XtraDB
存储限制	没有	没有	有	64TB	64TB
事务安全		支持		支持	支持
锁机制	表锁	页锁	表锁	行锁	行锁
B树索引	支持	支持	支持	支持	支持
哈希索引			支持	支持	支持
全文索引	支持				
集群索引				支持	支持
数据缓存			支持	支持	支持
索引缓存	支持		支持	支持	支持
数据可压缩	支持				
空间使用	低	低	N/A	高	高
内存使用	低	低	中等	高	高
批量插入的速度	高	高	高	低	低
支持外键				支持	支持

存储引擎	事务	锁粒度	主要应用	忌用
MyISAM	不支持	支持并发插入的表级锁	SELECT,INSERT，	读写操作频繁
MRG_MYISAM	不支持	支持并发插入的表级锁	分段归档，数据仓库	全局查找过多的场景
Innodb	支持	支持MVCC的行级锁	事务处理	无
Archive	不支持	行级锁	日志记录,只支持insert,select	需要随机读取，更新,删除
Ndb cluster	支持	行级锁	高可用性	大部份应用

- innoDB

  mysql5.5默认的存储引擎，事务型数据库。

  1.数据底层的存储：数据表文件-->>.frm(表结构)文件和.ibd(数据和索引)文件

  2.事务：支持热备份，对数据完整性要求较高，mysql是较好选择

3.锁的粒度：采用MVVC（多版本并发）支持高并发操作，支持四种事务隔离级别，行锁

4.存储特点：采用聚簇索引

5.适用场景：更新和查询比较频繁，并发操作、要求事务，支持外键约束

```
#查看mysql数据存储位置
show variables like '%data%';
```

- MyISAM

  1.存储形式：数据表文件-->>.frm(表结构)和.MYD和.MYI(数据和索引分离)

  2.事务：不支持

  3.存储特点：非聚簇

  4.其他：全文检索，压缩，延迟更新索引等

  5.适用场景：count计算、查询

- Memory

  1.数据保存在内存中，增删改查效率高，但是不能持久化

  2.不支持事务、表级锁

2.2如何设置存储引擎

```
#默认mysql的配置文件路径
修改存储引擎：
default_storage_engine=INNODB

#创建表时修改存储引擎：
create table tname(col) engine = INNODB;

#查看某张表的基本信息
#命令行模式
show tables status from dbname where name = tname \G;
```

3.索引

3.1索引简介

索引：为了提升查询效率创建数据机构

3.2常见的索引模型

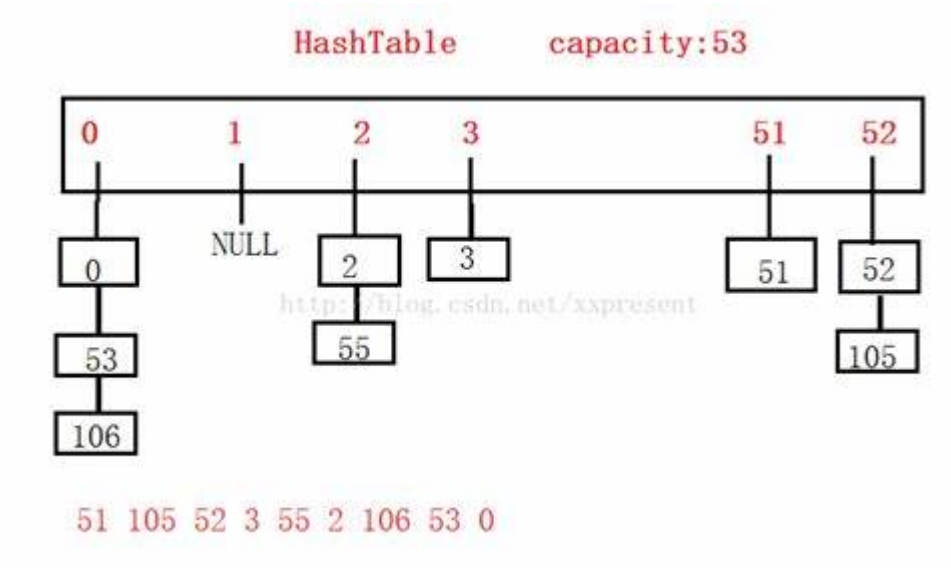
- 哈希表

键-值方式存储数据结构。使用key进行hash计算获取到一个值（位置），去该位置寻找数据（值）
数组，hash函数

适用场景：
等值查询
不适用场景：
范围查询，存储是无序的

哈希碰撞（哈希冲突）
解决方法：
1、链表（拉链法）

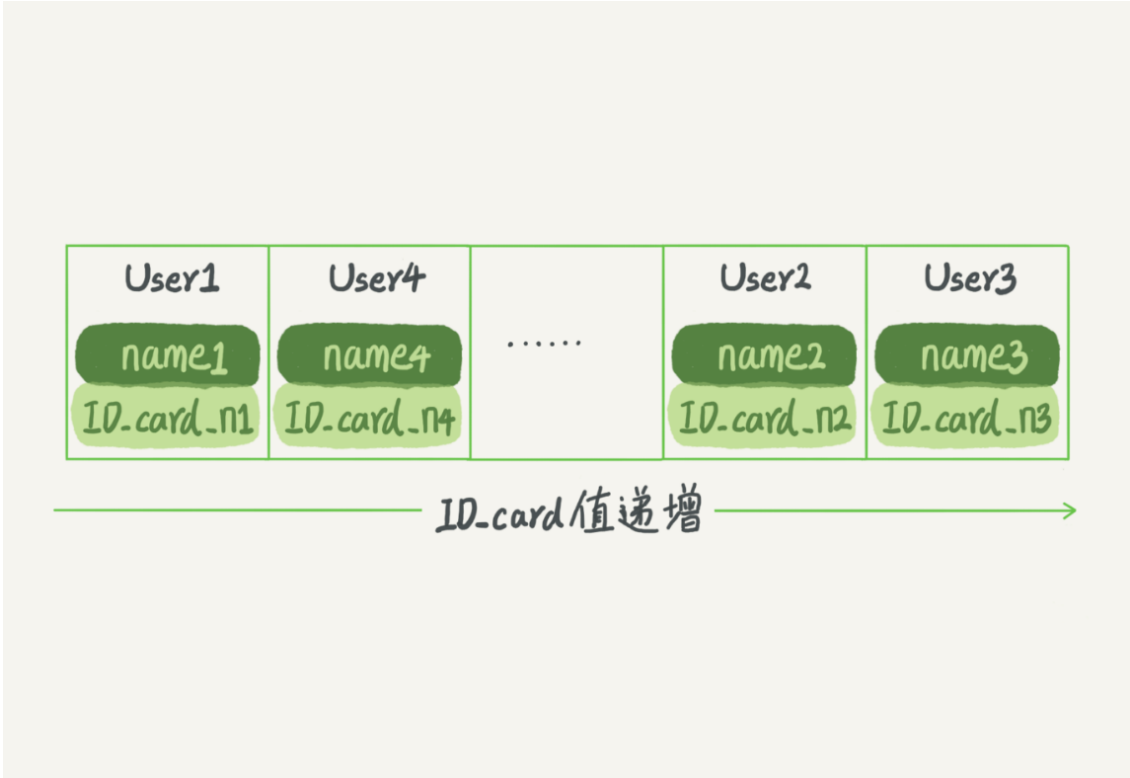
- 拉链法



  - 有序数组

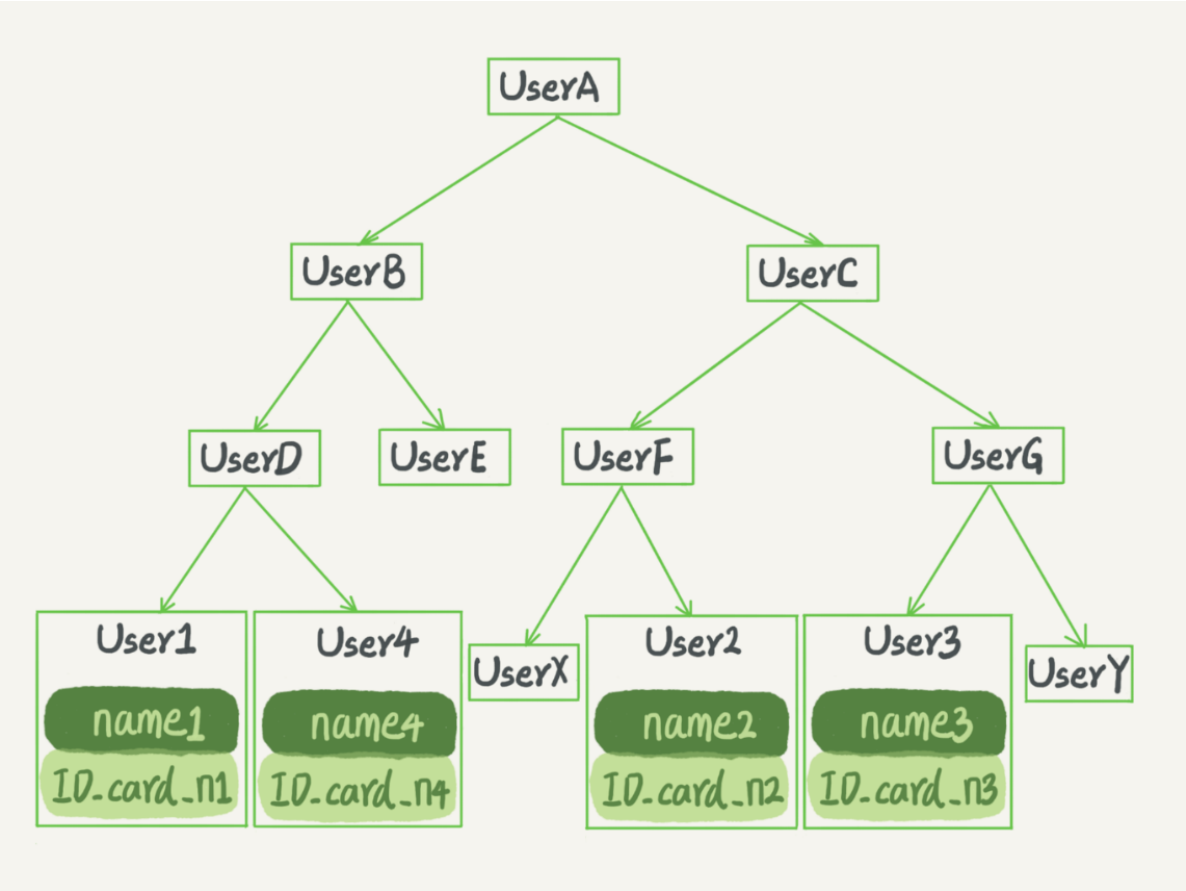
    整个数组中排列的是有序的，查找非常方便，可以使用二分法
    适用场景：等值查询、范围查询

    不适用场景：
    进行数据的插入与删除



- 二叉搜索树

二叉树结构进行数据存储
树最影响效率的是树的深度，尽可能平衡



3.3innoDB索引模式

> innodb根据主键的顺序已索引的方式进行存储。
>
> innodb中默认采用B+树索引模型，数据组织存储到B+树中。

1、索引类型分为主键索引和非主键索引

主键索引：叶子节点中存储的整行数据，称为聚簇索引。

非主键索引：叶子节点中存储的是主键的值称为二级索引。

2、主键索引和非主键索引的区别

主键索引只需要查找一次B+树就可以定位数据

非主键索引第一次只能查找到主键的值，再根据主键的值查找到数据。

3、索引需要维护

添加索引避免在增删操作较多的表

索引自生也需要占用空间存储

4、索引选择

优先选择主键索引

主键长度越小越好

尽量不要选择业务字段作为主键

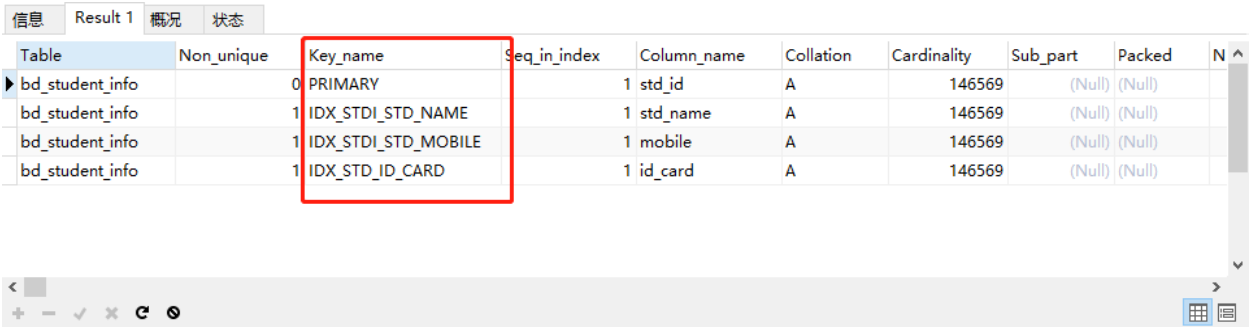
表的数据量较多的时候使用

经可能选择高基数列（尽量唯一）

3.4添加索引

- 查看索引

```
show index from emp;
```



信息 Result 1 概况 状态

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	N
bd_student_info	0	PRIMARY	1	std_id	A	146569	(Null)	(Null)	
bd_student_info	1	IDX_STDI_STD_NAME	1	std_name	A	146569	(Null)	(Null)	
bd_student_info	1	IDX_STDI_STD_MOBILE	1	mobile	A	146569	(Null)	(Null)	
bd_student_info	1	IDX_STD_ID_CARD	1	id_card	A	146569	(Null)	(Null)	

- 普通索引

```
#添加索引
添加索引语法
create index index_name on tname(colname);

创建表时添加
create table tname(
col ...
....
index index_name(colname)
)

#删除索引(全部)
drop index INDEX_NAME ON tname;
```

- 唯一索引

> 索引列必须唯一

```
#添加唯一索引
create unique index index_name UN_ID on tname(col);

#删除唯一索引
alter table tname drop index index_name;
如:
alter table stu drop index UN_ID;
```

- 主键索引

> 特殊的唯一索引

```
#添加主键索引就是将唯一索引设置为主键
如:
alter table tname add constraint PK_ID primary key (id);
```

- 组合索引

```
可以在多个字段添加索引，但是查询走索引必须使用第一个索引字段，否则不走索引
create index index_name on tname(col1,col2...);
```

4.sql优化

4.1使用explain分析查询

explain：查看执行计划

```
explain select * from stu where id = 1;
```

EXPLAIN 输出列对照表		
Column	JSON Name	含义
id	select_id	SELECT 标识符
select_type	没有	SELECT 类型（PRIMARY、UNION、SUBQUERY）
table	table_name	指数据库读取的表名，按照读取的先后顺序排列
partitions	partitions	匹配的分区
type	access_type	指本数据表与其他数据表之间的关联关系，其它取值有 system、const、eq_ref、ref、range、index 和 All
possible_keys	possible_keys	搜索时可选用的各个索引
key	key	使用的实际索引
key_len	key_length	索引长度，按字节计算，数值越小，表示越快
ref	ref	标识那些用来进行索引比较的列或者常量
rows	rows	预计要检索的行数
filtered	filtered	表条件过滤的行的百分比
Extra	没有	附加信息，提供了与操作有关联的信息

type：连接类型

```
ALL：表示全表扫描，若表中数据有百万至千万级别，必须要优化，否则性能很慢；
Index：full index scan,index与ALL的区别为index只遍历索引树，这通常比ALL快，因为索引文件通常比数据文件小（也就是说ALL和index都是读全表，但是index是从索引中读取的，而ALL是从硬盘中读取的）
range：只检索给定范围的行，使用一个索引来选择行，key列显示是用来那个索引，一般就是在你where语句中出现了between、<、>、in等查询，这种范围扫描比全表扫描要好，因为它只需要开始索引某一点，而结束语另一点，不用扫描全部索引。
ref：非唯一性索引扫描，返回匹配某一个单独值得所有行，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，所以它应该属于查找和扫描的混合体
eq_ref:唯一索引扫描，对于每一个索引键，表中只有一条记录与之匹配，常见于主键或唯一索引扫描；
const：表示通过索引一次就找到了，const用户比较primary key或者unique索引。因为只匹配一行数据，所以很快；如将逐渐置于where列表中，MySQL就能将查询转换为一个常量；
system：表示只有一行记录（等于系统表），这是const类型的特列，平时不会出现，可以忽略不计。
```

possiable keys:可能用到的索引

key：真正用到的索引

ley_len：索引长度，越短越好

ref：那一列被使用

row：查询的行数

4.2profiling分析语句执行时间

```
set profiling = on;  打开统计分析
执行查询语句后执行以下语句：
show profiles;  查看执行时间
```

4.3sql的基本优化原则

- 1.字段维护在Btree上，需要对数据类型有要求（设计要求）
  - 尽可能使用较小的类型
  - 尽可能使用简单类型
  - 尽可能设置合理长度，固定长度比变化长度要快
  - not null 约束
- 2.sql编写要求
  - 尽量不要使用select *
  - sql尽量减少嵌套
  - limit限制结果
- 3.索引失效的状况
  - 索引列使用函数sub(),包含不要使用（会导致索引失效）

    ```
    select *from emp where id = 1; 索引
    select *from emp where id like '%1%'; 全表
    ```

  - 索引列不要用于计算

```
select * from emp where sal *0.9 = 1000; 索引失效
select * from emp where sal  = 1000/0.9; 索引有效
```

- 索引列不要使用（!= <>）

```
select * from emp where no <> 10; 全表

select * from emp where no > 10
union
select * from emp where no < 10;  索引
```

- in和exists

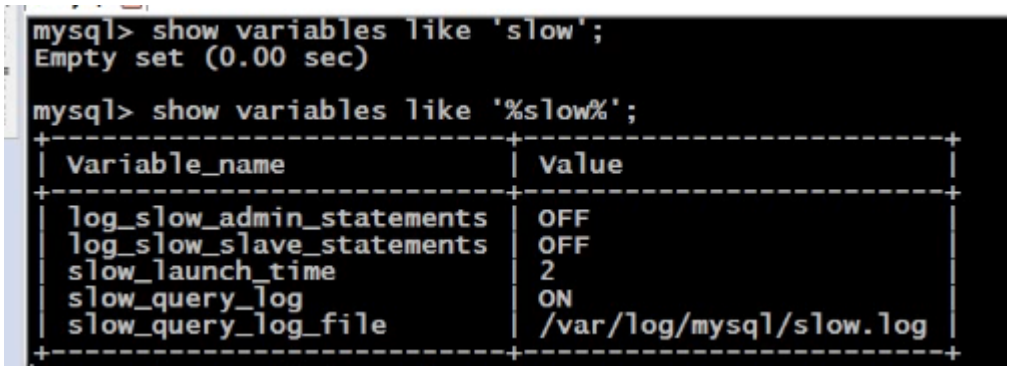
```
in 把内表和外表进行hash连接，exists把外表loop
内表和外表数据量差距不大，性能基本相同
内表数据量小使用in，外表数据量小使用exists
```

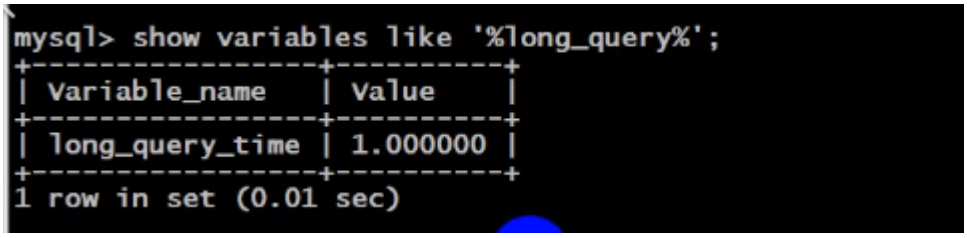
5.慢查询

> 开启慢查询日志，MySQL记录执行阶段影响性能的sql信息，定位分析系统的瓶颈

5.1慢查询的参数

```
#查看慢查询是否开启及慢查询日志位置
show variables like 'slow';
#查看慢查询的时间长度（超过多长时间就是慢查询）
show variables like 'long_query';
```





5.2设置慢查询

- 开启慢查询(命令开启，重启数据库后会失效)

```
#开启慢查询
set global show_query_log=on;
```

- 设置配置文件开启（重启MySQL服务后生效）

```
#/etc/my.cnf
[mysqld]
slow_query_log_file=/var/log/mysql/slow.log  :注意修改权限使之可以读写777
slow_query_log = ON
long_query_time=1
```

- 查看执行慢查询的次数

```
show global status like '%slow_queries%';
```

- 慢查询分析工具
  - mysqlduampslow（内置慢查询分析工具）

    ```
    mysqlduampslow /var/log/mysql/slow.log
    ```

  - pt-query-digest(第三方分析工具)

    ```
    pt-query-digest /var/log/mysql/slow.log
    ```