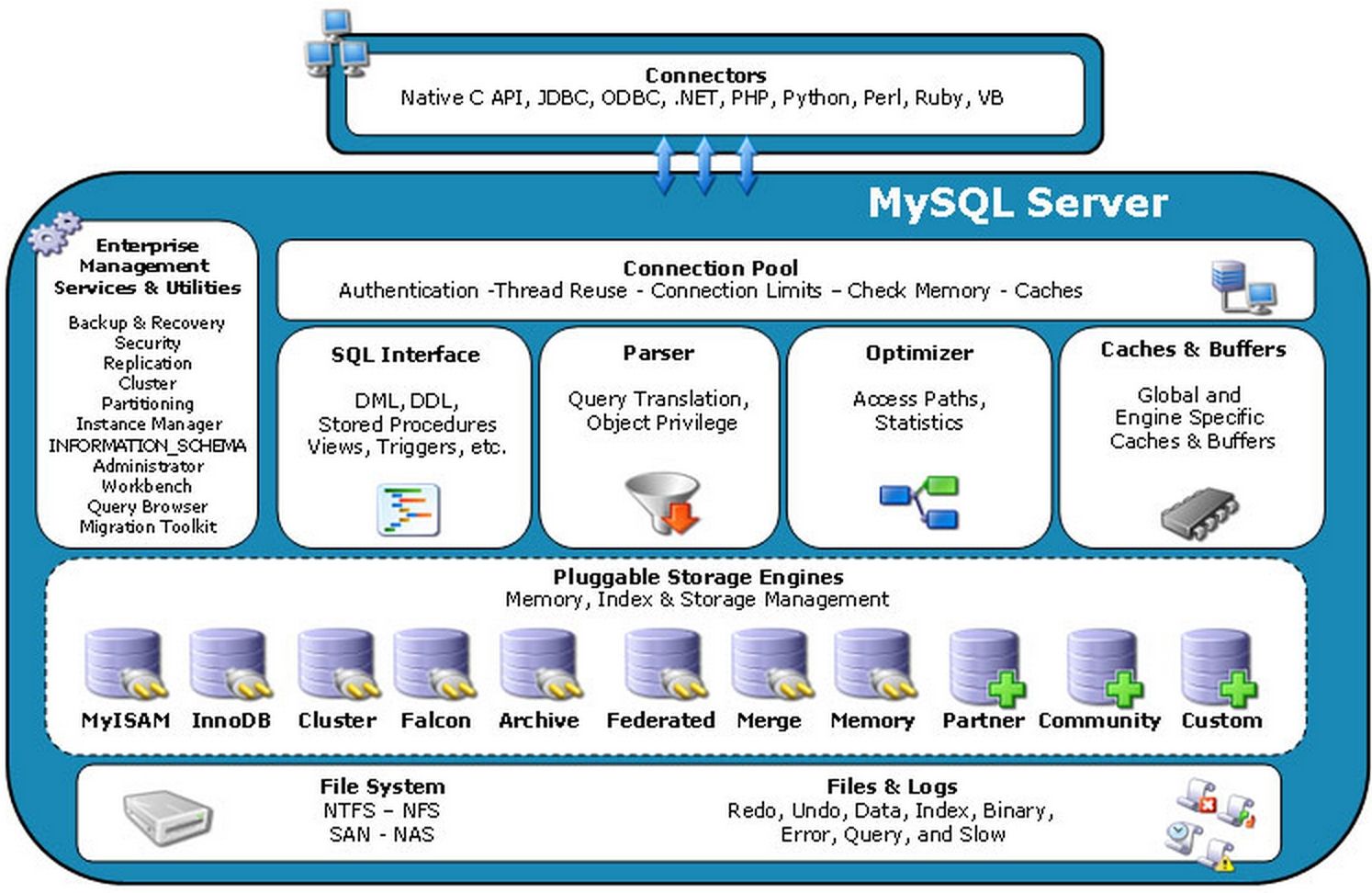
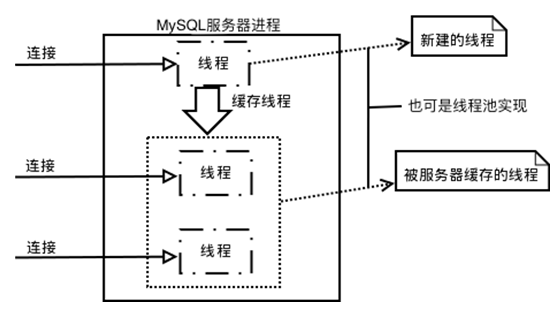
# Mysql的逻辑架构

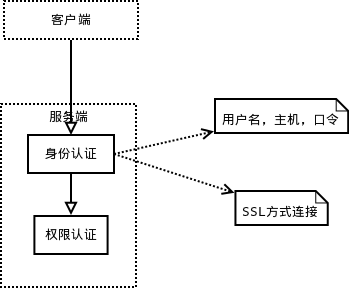


## 连接层

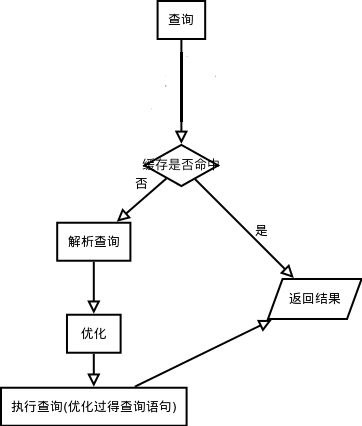


当MySQL启动（MySQL服务器就是一个进程），等待客户端连接，每一个客户端连接请求，服务器都会新建一个线程处理（如果是线程池的话，则是分配一个空的线程），每个线程独立，拥有各自的内存处理空间，但是，如果这个请求只是查询，没关系，但是若是修改数据，很显然，当两个线程修改同一块内存是会引发数据同步问题的。

## SQL处理层



连接到服务器，服务器需要对其进行验证，也就是用户名、IP、密码验证，一旦连接成功，还要验证是否具有执行某个特定查询的权限（例如，是否允许客户端对某个数据库某个表的某个操作）

这一层主要功能有：SQL语句的解析、优化，缓存的查询，MySQL内置函数的实现，跨存储引擎功能（所谓跨存储引擎就是说每个引擎都需提供的功能（引擎需对外提供接口）），例如：存储过程、触发器、视图等。

1.如果是查询语句（select语句），首先会查询缓存是否已有相应结果，有则返回结果，无则进行下一步（如果不是查询语句，同样调到下一步）

2.解析查询，创建一个内部数据结构（解析树），这个解析树主要用来SQL语句的语义与语法解析；

3.优化：优化SQL语句，例如重写查询，决定表的读取顺序，以及选择需要的索引等。这一阶段用户是可以查询的，查询服务器优化器是如何进行优化的，便于用户重构查询和修改相关配置，达到最优化。这一阶段还涉及到存储引擎，优化器会询问存储引擎，比如某个操作的开销信息、是否对特定索引有查询优化等。

## 缓存

show variables like '%query\_cache\_type%'

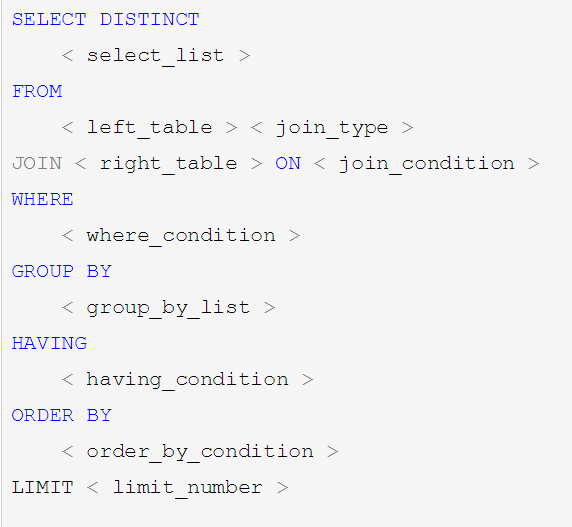
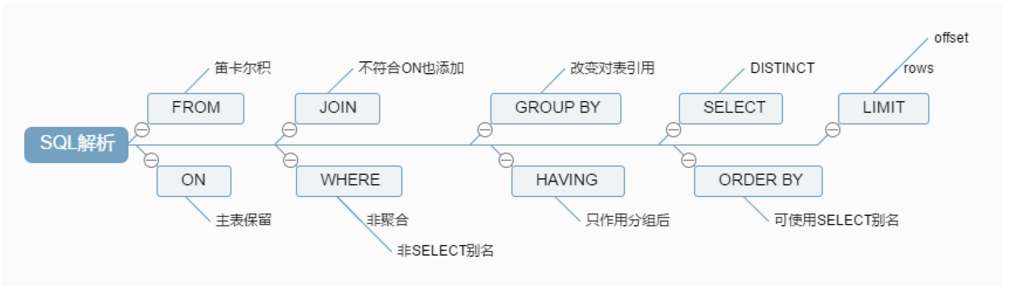
C:\ProgramData\MySQL\MySQL Server 5.6

SET GLOBAL query\_cache\_size = 4000;

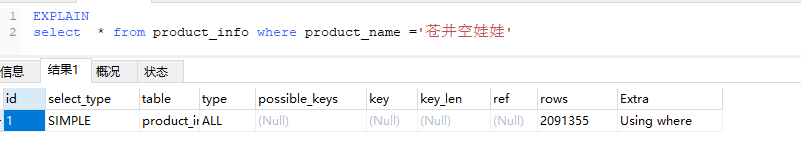
SET GLOBAL query\_cache\_size = 134217728;

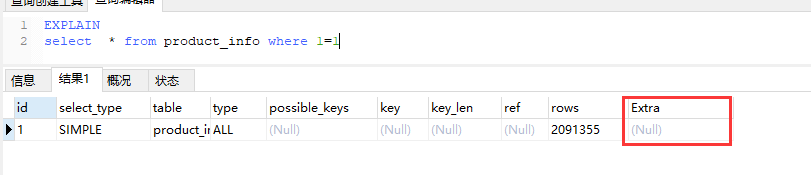
select \* from product\_info where product\_name ='娃娃'

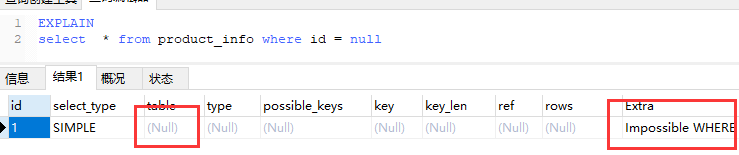
## 解析查询

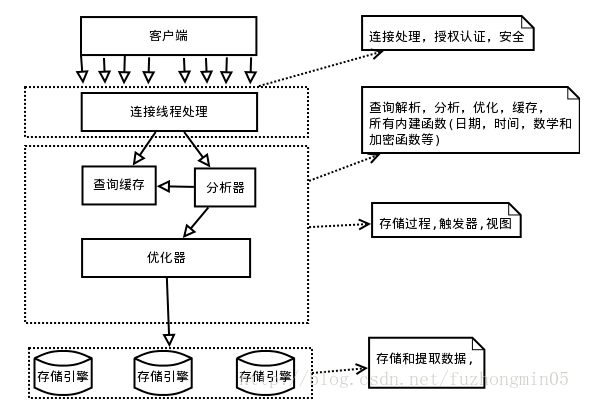
优化

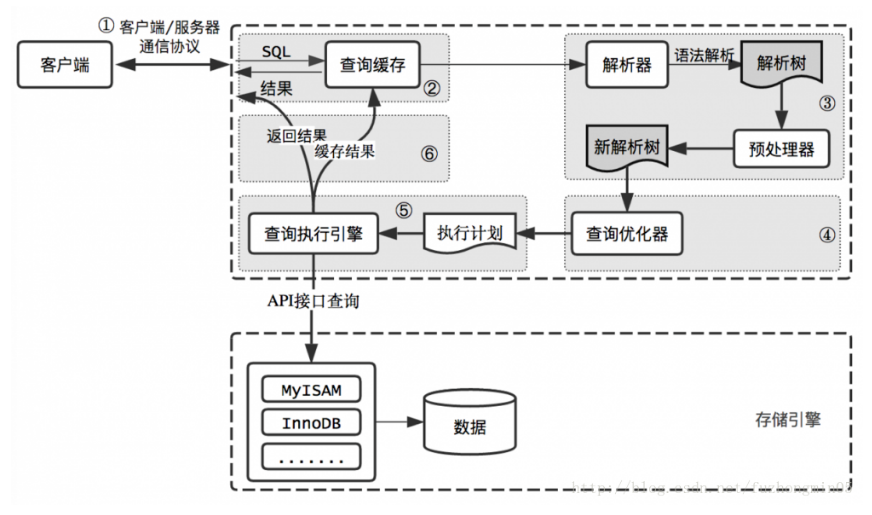






逻辑视图





mysql语句查询过程

# 存储引擎

#看你的mysql现在已提供什么存储引擎:

mysql> show engines;

#看你的mysql当前默认的存储引擎:

mysql> show variables like '%storage\_engine%';

show table status like '表名';

或

show table status like '表名' \G

1.查看数据库下面是否支持事务（InnoDB支持）？

show engines;

2.查看mysql当前默认的存储引擎？

show variables like '%storage\_engine%';

3.查看某张表的存储引擎？

show create table 表名 ;

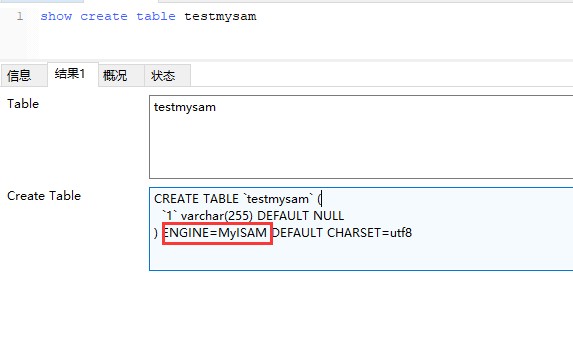
4.对于表的存储结构的修改？

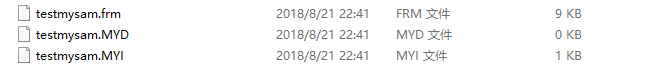
建立InnoDB 表：Create table .... type=InnoDB； Alter table table\_name type=InnoDB

## MyISAM

MySql 5.5之前默认的存储引擎

MyISAM 存储引擎由MYD和MYI组成





特性：

并发性与锁级别-表级锁

支持全文检索

支持数据压缩

myisampack -b -f testmysam.MYI

.\myisampack.exe -b -f "C:\ProgramData\MySQL\MySQL Server 5.6\dat

a\mysqldemo\product\_info.MYI"

.\myisamchk.exe -b -f "C:\ProgramData\MySQL\MySQL Server 5.6\data

\mysqldemo\product\_info.MYI"

CHECK table product\_info //校验表

REPAIR table product\_info //修复表结构

适用场景：

非事务型应用（数据仓库，报表，日志数据）

只读类应用

空间类应用（空间函数，坐标）

## Innodb

MySql 5.5以及以后版本默认存储引擎

show variables like 'innodb\_file\_per\_table';

ON:独立的表空间：data目录下,文件夹名是数据库名,每张表包括.frm(表架构)与.ibd(表数据)文件

OFF:系统表空间：ibdataX,在data目录下的ibdata1

独立表空间是一个表一个空间,系统表空间是所有的表数据都在系统空间.

表空间整理： optimize table cai\_info;，此操作过程会锁表

mysql5.6以前默认为系统表空间

系统表空间和独立表空间

* 系统表空间无法简单的收缩文件大小
* 独立表空间可以通过optimize table cai\_info;收缩系统文件（释放表空间）
* 系统表空间会产生IO瓶颈
* 独立表空间可以同时向多个文件刷新数据
* 建议：Innodb使用独立表空间

特性

Innodb是一种事务性存储引擎

完全支持事务得ACID特性

Redo Log 和 Undo Log

Innodb支持行级锁（并发程度更高）

|  |  |  |
| --- | --- | --- |
| **对比项** | **MyISAM** | **InnoDB** |
| 主外键 | 不支持 | 支持 |
| 事务 | 不支持 | 支持 |
| 行表锁 | 表锁，即使操作一条记录也会锁住整个表  不适合高并发的操作 | 行锁,操作时只锁某一行，不对其它行有影响  适合高并发的操作 |
| 缓存 | 只缓存索引，不缓存真实数据 | 不仅缓存索引还要缓存真实数据，对内存要求较高，而且内存大小对性能有决定性的影响 |
| 表空间 | 小 | 大 |
| 关注点 | 性能 | 事务 |
| 默认安装 | Y | Y |

## CSV

数据以文本方式存储在文件

.csv文件存储内容

.csm文件存储表得元数据如表状态和数据量

.frm 表结构



特点

以csv格式进行数据存储

所有列都不能为null的

不支持索引（不适合大表，不适合在线处理）

可以对数据文件直接编辑（保存文本文件内容）

create table mycsv(id int not null,c1 VARCHAR(10) not null,c2 char(10) not null) engine=csv DEFAULT CHARSET=gbk;

insert into mycsv values(1,'aaa','bbb'),(2,'cccc','dddd');

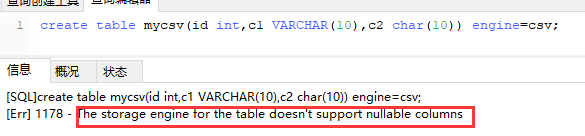
CHECK TABLE mycsv;修复表

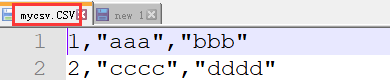
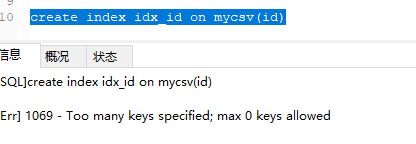
修改文本数据之后要刷新表

flush TABLES;

select \* from mycsv

create index idx\_id on mycsv(id)



使用场景: 适合做为数据交换的中间表（能够在服务器运行的时候，拷贝和拷出文件，可以将电子表格存储为CSV文件再拷贝到MySQL数据目录下，就能够在数据库中打开和使用。同样，如果将数据写入到CSV文件数据表中，其它web程序也可以迅速读取到数据。

## Archive

组成

以zlib对表数据进行压缩，磁盘I/O更少

数据存储在ARZ为后缀的文件中

特点：

只支持insert和select操作

只允许在自增ID列上加索引



create table myarchive(id int auto\_increment not null,c1 VARCHAR(10),c2 char(10), key(id)) engine = archive;

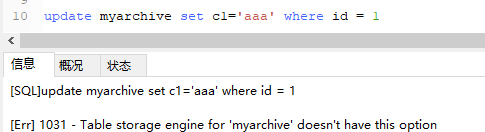
create index idx\_c1 on myarchive(c1)

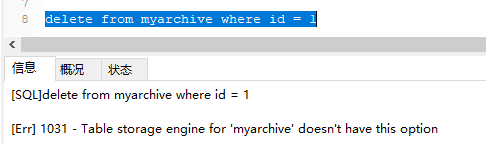
INSERT into myarchive(c1,c2) value('aa','bb'),('cc','dd');

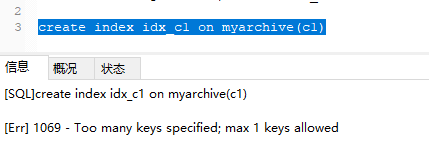
delete from myarchive where id = 1

update myarchive set c1='aaa' where id = 1

create index idx\_c1 on myarchive(c1)







使用场景

日志和数据采集应用

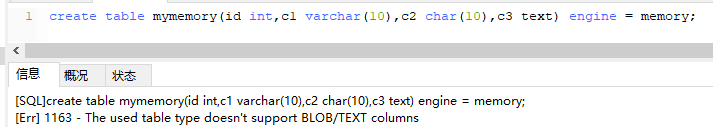
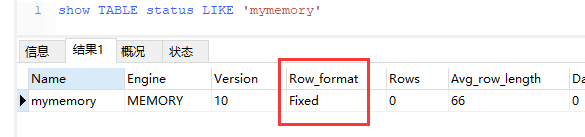
## Memory

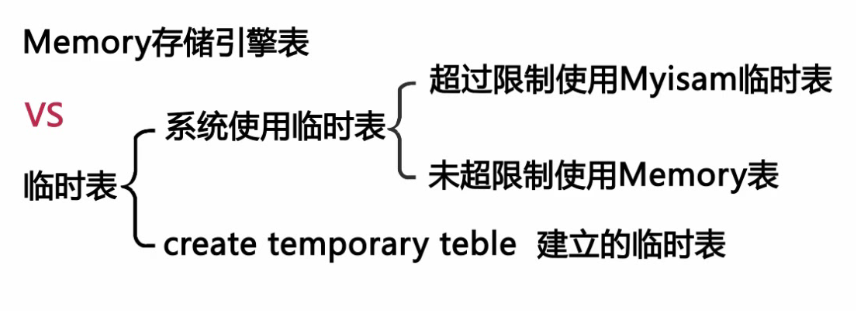
* 文件系统存储特点

也称HEAP存储引擎，所以数据保存在内存中

* 支持HASH索引和BTree索引
* 所有字段都是固定长度 varchar(10) = char(10)
* 不支持Blog和Text等大字段
* Memory存储引擎使用表级锁
* 最大大小由max\_heap\_table\_size参数决定

show VARIABLES like 'max\_heap\_table\_size'



使用场景

* hash索引用于查找或者是映射表（邮编和地区的对应表）
* 用于保存数据分析中产生的中间表
* 用于缓存周期性聚合数据的结果表

memory数据易丢失，所以要求数据可再生

## Ferderated

特点

* 提供了访问远程MySQL服务器上表的方法
* 本地不存储数据，数据全部放到远程服务器上
* 本地需要保存表结构和远程服务器的连接信息

使用场景

偶尔的统计分析及手工查询

如何使用

默认禁止，启用需要再启动时增加federated参数

mysql://user\_name[:password]@hostname[:port\_num]/db\_name/table\_name

show ENGINES

create database local;

create database remote;

create table remote\_fed(id int auto\_increment not null,c1 varchar(10) not null default '',c2 char(10) not null default '',primary key(id)) engine = INNODB

INSERT into remote\_fed(c1,c2) values('aaa','bbb'),('ccc','ddd'),('eee','fff');

CREATE TABLE `local\_fed` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`c1` varchar(10) NOT NULL DEFAULT '',

`c2` char(10) NOT NULL DEFAULT '',

PRIMARY KEY (`id`)

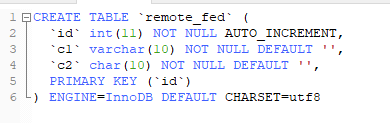
) ENGINE=federated CONNECTION ='mysql://root:root1234%@127.0.0.1:3306/remote/remote\_fed'

select \* from local\_fed

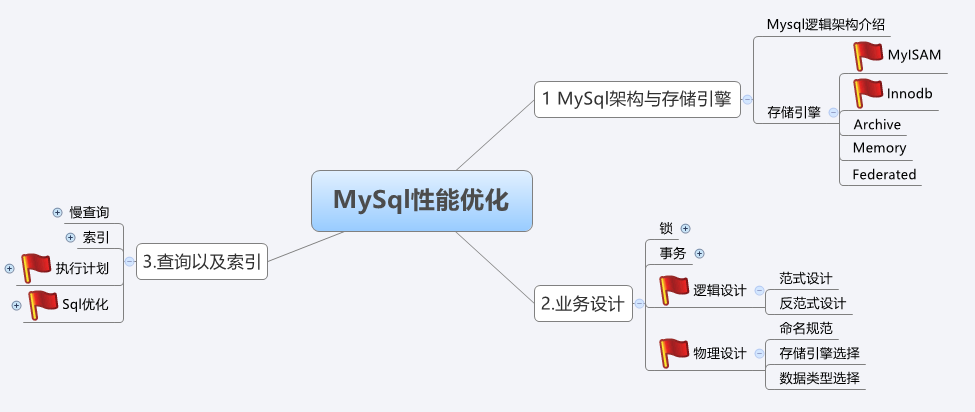
delete from local\_fed where id = 2

select \* from remote.remote\_fed





# MYSQL性能优化



## 衡量指标

TPS：

Transactions Per Second（每秒传输的事物处理个数），这是指服务器每秒处理的事务数，支持事务的存储引擎如InnoDB等特有的一个性能指标。

QPS：

Queries Per Second（每秒查询处理量）同时适用与InnoDB和MyISAM 引擎

等待时间：执行Sql等待返回结果之间的等待时间

TPS = (COM\_COMMIT + COM\_ROLLBACK)/UPTIME

QPS=QUESTIONS/UPTIME

**use information\_schema;**

**select VARIABLE\_VALUE into @num\_com from GLOBAL\_STATUS where VARIABLE\_NAME ='COM\_COMMIT';**

**select VARIABLE\_VALUE into @num\_roll from GLOBAL\_STATUS where VARIABLE\_NAME ='COM\_ROLLBACK';**

**select VARIABLE\_VALUE into @uptime from GLOBAL\_STATUS where VARIABLE\_NAME ='UPTIME';**

**select (@num\_com+@num\_roll)/@uptime;**

use information\_schema;

select VARIABLE\_VALUE into @num\_queries from GLOBAL\_STATUS where VARIABLE\_NAME ='QUESTIONS';

select VARIABLE\_VALUE into @uptime from GLOBAL\_STATUS where VARIABLE\_NAME ='UPTIME';

select @num\_queries/@uptime;

## MySqlSlap

MySQLSlap是从MySQL的5.1.4版开始就开始官方提供的压力测试工具

* 创建schema、table、test data ；
* 运行负载测试，可以使用多个并发客户端连接；
* 测试环境清理（删除创建的数据、表等，断开连接）

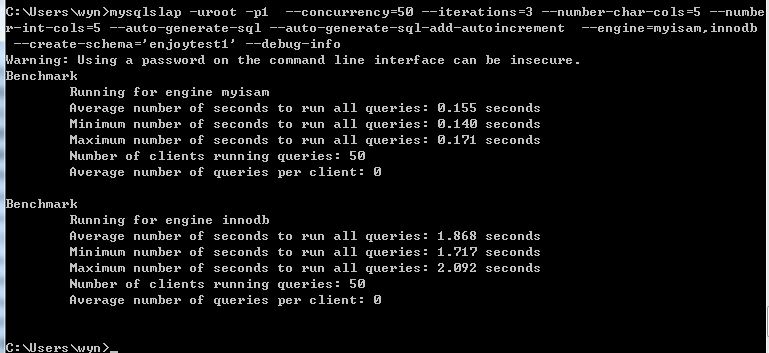
|  |  |
| --- | --- |
| **参数** | **作用** |
| --create-schema=name | 指定测试的数据库名，默认是mysqlslap |
| --engine=name | 创建测试表所使用的存储引擎，可指定多个 |
| --concurrency=N | 模拟N个客户端并发执行。可指定多个值，以逗号或者 |
| --number-of-queries=N | 总的测试查询次数(并发客户数×每客户查询次数)，比如并发是10，总次数是100，那么10个客户端各执行10个 |
| --iterations=N | 迭代执行的次数，即重复的次数（相同的测试进行N次，求一个平均值），指的是整个步骤的重复次数，包括准备数据、测试load、清理 |
| --commit=N | 执行N条DML后提交一次 |
| --auto-generate-sql, -a | # 自动生成测试表和数据，表示用mysqlslap工具自己生成的SQL脚本来测试并发压力。 |
| --auto-generate-sql-load-type=name | # 测试语句的类型。代表要测试的环境是读操作还是写操作还是两者混合的。  # 取值包括：read (scan tables), write (insert into tables), key (read primary keys), update (update primary keys), or mixed (half inserts, half scanning selects). 默认值是：mixed. |
| --auto-generate-sql-add-auto-increment | 对生成的表自动添加auto\_increment列 |
| --number-char-cols=name | 自动生成的测试表中包含N个字符类型的列，默认1 |
| --number-int-cols=name | 自动生成的测试表中包含N个数字类型的列，默认1 |
| --debug-info | 打印内存和CPU的信息 |

show variables like 'max\_connections';查最大连接数

set global max\_connections=**1000**;

**./mysqlslap -uroot -p1 --concurrency=50 --iterations=3 --number-char-cols=5 --number-int-cols=5 --auto-generate-sql --auto-generate-sql-add-autoincrement --engine=myisam,innodb --create-schema='enjoytest1' --debug-info**

(windows不要./)



**./mysqlslap -uroot -p1 --concurrency=50 --iterations 3 -a --auto-generate-sql-add-autoincrement --engine=innodb --number-of-queries=50**

**./mysqlslap -uroot -p1 --concurrency=100 --iterations 3 -a --auto-generate-sql-add-autoincrement --engine=innodb --number-of-queries=100**

**./mysqlslap -uroot -p1 --concurrency=1,50,100,200 --iterations=3 --number-char-cols=5 --number-int-cols=5 --auto-generate-sql --auto-generate-sql-add-autoincrement --engine=myisam,innodb --create-schema='enjoytest1' --debug-info**

# 锁

* 锁是计算机协调多个进程或线程并发访问某一资源的机制。
* 在数据库中，数据也是一种供许多用户共享的资源。如何保证数据并发访问的一致性、有效性是所有数据库必须解决的一个问题，锁冲突也是影响数据库并发访问性能的一个重要因素。
* 锁对数据库而言显得尤其重要，也更加复杂。

MySQL的锁机制比较简单

其最显著的特点是不同的存储引擎支持不同的锁机制

比如：

MyISAM和MEMORY存储引擎采用的是表级锁（table-level locking）；

InnoDB存储引擎既支持行级锁（row-level locking），也支持表级锁，但默认情况下是采用行级锁。

* 表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高,并发度最低。
* 行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低,并发度也最高。
* 页面锁：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般。

仅从锁的角度来说：

表级锁更适合于以查询为主，只有少量按索引条件更新数据的应用，如OLAP系统

行级锁则更适合于有大量按索引条件并发更新少量不同数据，同时又有并发查询的应用，如一些在线事务处理（OLTP）系统。

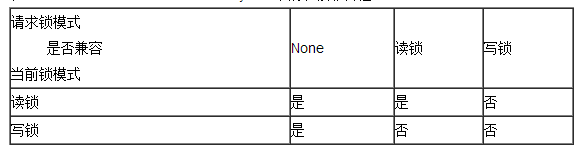
很难笼统地说哪种锁更好，只能就具体应用的特点来说哪种锁更合适

## MyISAM的表锁-共享读锁

MySQL的表级锁有两种模式：

表共享读锁（Table Read Lock）

表独占写锁（Table Write Lock）



共享读锁语法

给表加锁

加共享读锁

lock table 表名 read

1. lock table testmysam READ 启动另外一个session select \* from testmysam 可以查询

2.同一个session中， insert into testmysam value(2);

update testmysam set id=2 where id=1;

报错

3.在另外一个session中

insert into testmysam value(2); 等待

4.在同一个session中

insert into testdemo value(2,'2','3'); 报错

select \* from testdemo ; 报错

5.在另外一个session中

insert into testdemo value(2,'2','3'); 成功

6.加锁在同一个session 中 select s.\* from testmysam s 报错

lock table 表名 as 别名 read;

查看 show status LIKE 'table\_locks\_waited' 表被锁过几次

加了读锁，同一个session中只能读不能更新，也不能对另外表进行读写。

另外session中，当前表可以读，写等待。另外表可以读写

表独占写锁语法

给表加锁

加独占写锁

lock table 表名 write

1.lock table testmysam WRITE

在同一个session中

insert testmysam value(3);

delete from testmysam where id = 3

select \* from testmysam

2.对不同的表操作（报错）

select s.\* from testmysam s

insert into testdemo value(2,'2','3');

3.在其他session中 （等待）

select \* from testmysam

总结

* 对MyISAM表的读操作，不会阻塞其他用户对同一表的读请求，但会阻塞对同一表的写请求
* 对MyISAM表的读操作，不会阻塞当前session对表读，当对表进行修改会保存
* 一个session使用LOCK TABLE命令给表f加了读锁，这个session可以查询锁定表中的记录，但更新或访问其他表都会提示错误；
* 另外一个session可以查询表中的记录，但更新就会出现锁等待
* 对 MyISAM表的写操作，则会阻塞其他用户对同一表的读和写操作；
* 对 MyISAM表的写操作，当前session可以对本表做CRUD,但对其他表进行操作会报错

## InnoDb行锁

在mysql 的 InnoDB引擎支持行锁

行锁:

* 共享锁又称：读锁。当一个事务对某几行上读锁时，允许其他事务对这几行进行读操作，但不允许其进行写操作，也不允许其他事务给这几行上排它锁，但允许上读锁。
* 排它锁又称：写锁。当一个事务对某几个上写锁时，不允许其他事务写，但允许读。更不允许其他事务给这几行上任何锁。包括写锁。

语法

上共享锁的写法：lock in share mode

例如： select \* from 表 where 条件 lock in share mode；

上排它锁的写法：for update

例如：select \* from 表 where 条件 for update；

注意：

1.两个事务不能锁同一个索引。

2.insert ，delete ， update在事务中都会自动默认加上排它锁。

3.行锁必须有索引才能实现，否则会自动锁全表，那么就不是行锁了。

面试题：系统运行一段时间，数据量已经很大，这时候系统升级，有张表A需要增加个字段，并发量白天晚上都很大，请问怎么修改表结构

面试考点

（修改表结构会导致表锁,数据量大修改数据很长，导致大量用户阻塞，无法访问！）

1. 首先创建一个和你要执行的alter操作的表一样的空的表结构。

2. 执行我们赋予的表结构的修改，然后copy原表中的数据到新表里面。

3. 在原表上创建一个触发器在数据copy的过程中，将原表的更新数据的操作全部更新到新的表中来。

4. copy完成之后，用rename table 新表代替原表，默认删除原表。

可以用工具

* 下载安装perl环境 http://www.perl.org/get.html
* 下载percona-toolkit工具集合 https://www.percona.com/doc/percona-toolkit
* ppm install DBI 依赖
* ppm install DBD::mysql 安装 mysql驱动依赖

在perl目录下执行

pt-online-schema-change h=127.0.0.1,u=root,D=mysqldemo,t=product\_info --alter "modify product\_name varchar(150) not null default '' " --execute

事实上,5.6版本后的alter表并不锁表.

两千万数据的表alter加字段耗时1个半小时,期间应用正常运行.

查询版本号: SELECT VERSION();

# 事务

1.查看数据库下面是否支持事务（InnoDB支持）？

show engines;

2.查看mysql当前默认的存储引擎？

show variables like '%storage\_engine%';

3.查看某张表的存储引擎？

show create table 表名 ;

4.对于表的存储结构的修改？

建立InnoDB 表：Create table .... type=InnoDB； Alter table table\_name type=InnoDB

## 事务特性

事务应该具有4个属性：原子性、一致性、隔离性、持久性。这四个属性通常称为ACID特性。

* 原子性（atomicity）。一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。
* 一致性（consistency）。事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。
* 隔离性（isolation）。一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。
* 持久性（durability）。持久性也称永久性（permanence），指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

## 隔离级别

<https://www.cnblogs.com/huanongying/p/7021555.html>

未提交读（READ UNCOMMITED）脏读

已提交读 （READ COMMITED）不可重复读(读取已提交)

可重复读（REPEATABLE READ）

可串行化（SERIALIZABLE）

mysql默认的事务隔离级别为repeatable-read

查询隔离级别语句

show variables like '%tx\_isolation%';

set TRANSACTION ISOLATION LEVEL repeatable read;

事务并发问题

* 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
* 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。
* 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。

不可重复读是update,幻读是insert.

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

### 未提交读（READ UNCOMMITED）脏读

set SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

show variables like '%tx\_isolation%';

set SESSION TRANSACTION ISOLATION LEVEL read UNCOMMITTED;

一个session中

start TRANSACTION

update account set balance = balance -50 where id = 1

另外一个session中查询

select \* from account

回到第一个session中 回滚事务

ROLLBACK

在第二个session种

update account set balance = balance -50 where id = 1

查询结果还是 400

### 已提交读 （READ COMMITED）不可重复读

set SESSION TRANSACTION ISOLATION LEVEL read committed

show variables like '%tx\_isolation%';

set SESSION TRANSACTION ISOLATION LEVEL read committed;

一个session中

start TRANSACTION

update account set balance = balance -50 where id = 1

另外一个session中查询 (数据并没改变)

select \* from account

回到第一个session中

commit

在第二个session种

select \* from account (数据已经改变)

### 可重复读（REPEATABLE READ）

set SESSION TRANSACTION ISOLATION LEVEL repeatable read

show variables like '%tx\_isolation%';

set SESSION TRANSACTION ISOLATION LEVEL repeatable read;

一个session中

start TRANSACTION

update account set balance = balance -50 where id = 1

另外一个session中查询 (数据并没改变)

select \* from account

回到第一个session中 回滚事务

commit

在第二个session种

select \* from account (数据并未改变)

此时执行

update account set balance = balance -50 where id = 1

结果是按照session1执行的结果继续执行，数据正常，未被破坏。

可重复读的实现是MVCC机制，select不更新版本号，读的快照读，I u d会更新版本号，使用最新的版本号。

事务之间读的是加锁时的快照,但是如果用for update加锁时此句是事务提交后的数据.

比如语句2等待别的事务提交时, balance=1000,别的事务提交后balance=900,此时2处运行balance=900, 但是语句3处依然是加锁时的快照, balance=1000.但最后执行的结果正确,为800.如果语句3处要读出balance=900,需要更改事务级别为读已提交.

1. set autocommit=0;
2. SELECT \* from zuan\_account where user\_id=446 FOR UPDATE;
3. SELECT \* from zuan\_account where user\_id = 446;
4. UPDATE zuan\_account SET balance = balance - 100 WHERE user\_id = 446;
5. SELECT \* from zuan\_account where user\_id = 446;
6. COMMIT;

### 可串行化（SERIALIZABLE）

set SESSION TRANSACTION ISOLATION LEVEL serializable

show variables like '%tx\_isolation%';

set SESSION TRANSACTION ISOLATION LEVEL serializable;

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

1.开启一个事务

begin

select \* from account 发现3条记录

2.开启另外一个事务

begin

select \* from account 发现3条记录 也是3条记录

insert into account VALUES(4,'deer',500)

查询 4条记录

select \* from account

3.回到第一个session

insert into account VALUES(5,'james',500)

select \* from account 4条记录

4.session1 与 session2 都提交事务

|  |  |  |  |
| --- | --- | --- | --- |
| **事务隔离级别** | **脏读** | **不可重复读** | **幻读** |
| 读未提交（read-uncommitted） | 是 | 是 | 是 |
| 不可重复读（read-committed） | 否 | 是 | 是 |
| 可重复读（repeatable-read） | 否 | 否 | 是 |
| 串行化（serializable） | 否 | 否 | 否 |

总结:

事务隔离级别为可重复读时，如果有索引（包括主键索引）的时候，以索引列为条件更新数据，会存在间隙锁间、行锁、页锁的问题，从而锁住一些行；如果没有索引，更新数据时会锁住整张表

事务隔离级别为串行化时，读写数据都会锁住整张表

隔离级别越高，越能保证数据的完整性和一致性，但是对并发性能的影响也越大，对于多数应用程序，可以优先考虑把数据库系统的隔离级别设为Read Committed，它能够避免脏读取，而且具有较好的并发性能

## 事务语法

* 开启事务

1、begin

2、START TRANSACTION（推荐）

3、begin work

* 事务回滚 rollback
* 事务提交 commit
* 还原点savepoint

show variables like '%autocommit%'; 自动提交事务是开启的

set autocommit=0;

insert into testdemo values(5,5,5);

savepoint s1;

insert into testdemo values(6,6,6);

savepoint s2;

insert into testdemo values(7,7,7);

savepoint s3;

select \* from testdemo

rollback to savepoint s2

rollback

# 业务设计

## 逻辑设计

### 范式

* 数据库设计的第一大范式
* 数据库表中的所有字段都只具有单一属性,列不可再分
* 单一属性的列是由基本数据类型所构成的
* 设计出来的表都是简单的二维表

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **name-age** | | |
| 1 | 张三-23 | | |
| **id** | | **name** | **age** | |
| 1 | | 张三 | 23 | |

* 数据库设计的第二大范式
* 要求表中只具有一个业务主键，每个记录能被唯一地区分.
* 符合第二范式的表在第一范式的基础上属性完全依赖于主键. 所谓完全依赖是指不能存在仅依赖主关键字一部分的属性，如果存在，那么这个属性和主关键字的这一部分应该分离出来形成一个新的实体，新实体与原实体之间是一对多的关系。

订单编号 和产品ID没有直接关联

|  |  |  |
| --- | --- | --- |
| **订单表ID（主键）** | **订单时间** | **产品ID** |
| 1 | 2018-12-12 | 3 |
| 1 | 2018-12-12 | 4 |

* 数据库设计的第三大范式
* 指每一个非非主属性既不部分依赖于也不传递依赖于业务主键，也就是在第二范式的基础上任何非主属性不得传递依赖于主属性。

|  |  |  |  |
| --- | --- | --- | --- |
| **订单表ID（主键）** | **订单时间** | **客户编号** | **客户姓名** |
| 1 | 2018-12-12 | 1 | 张三 |
| 2 | 2018-12-12 | 2 | 李四 |

客户编号 和订单编号管理 关联

客户姓名 和订单编号管理 关联

客户编号 和 客户姓名 关联

把客户姓名这列删除，只放到客户表中

### 设计示例

* 按要求设计一个电子商务网站的数据库结构
* 本网站只销售图书类产品
* 需要具备以下功能

用户登陆 商品展示 供应商管理

用户管理 商品管理 订单销售

* 用户登陆及用户管理
* 用户必须注册并登陆系统才能进行网上交易，用户名用来作为用户信息的业务主键
* 同一时间一个用户只能在一个地方登陆

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **用户名** | **密码** | **手机号** | **姓名** | **注册时间** | **在线状态** | **出生日期** |

* 商品信息
* 一本书可以属于多个分类

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **商品名称** | **出版社名称** | **图书价格** | **图书表述** | **作者** |

* 分类信息

|  |  |
| --- | --- |
| **分类名称** | **分类描述** |

* 商品分类对应关系表

|  |  |
| --- | --- |
| **商品名称** | **分类名称** |

* 供应商信息

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **出版社名称** | **地址** | **电话** | **联系人** | **银行账号** |

* 在线销售—错误

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **下单日期** | **订单金额** | **订单商品分类** | **订单商品名pk** | **订单商品单价** | **订单商品数量** | **支付金额** | **物流单号** |

* 订单表

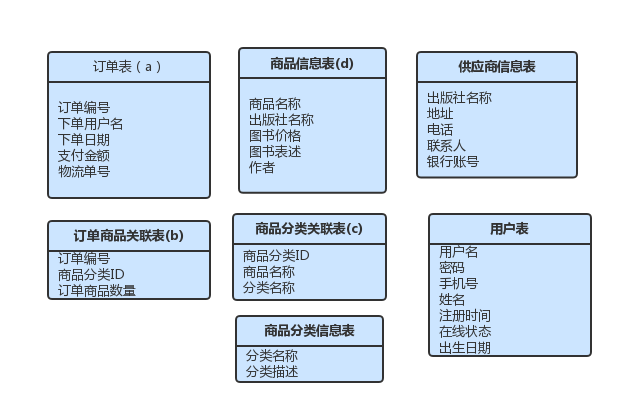
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **订单编号** | **下单用户名** | **下单日期** | **支付金额** | **物流单号** |

* 订单商品关联表

|  |  |  |  |
| --- | --- | --- | --- |
| **订单编号** | **订单商品分类** | **订单商品名** | **订单商品数量** |

|  |  |  |
| --- | --- | --- |
| **订单编号** | **商品分类ID** | **订单商品数量** |

表汇总



* 编写SQL查询出每一个用户的订单总金额

SELECT

a.单用户名,

sum(d.商品价格 \* b.商品数量)

FROM

订单表 a

JOIN 订单分类关联表 b ON a.订单编号 = b.订单编号

JOIN 商品分类关联表 c ON c.商品分类ID = b.商品分类ID

JOIN 商品信息表 d ON d.商品名称 = c.商品名称

GROUP BY a.下单用户名

* 编写SQL查询出下单用户和订单详情

SELECT

a.订单编号,

e.用户名,

e.手机号,

d.商品名称,

c.商品数量,

d.商品价格

FROM

订单表 a

JOIN 订单分类关联表 b ON a.订单编号 = b.订单编号

JOIN 商品分类关联表 c ON c.商品分类ID = b.商品分类ID

JOIN 商品信息表 d ON d.商品名称 = c.商品名称

JOIN 用户信息表 e ON e.用户名 = a.下单用户

* 大量的表关联非常影响查询的性能
* 完全符合范式化的设计有时并不能得到良好得SQL查询性能

### 反范式

* **什么叫反范式化设计**
* 反范式化是针对范式化而言得，在前面介绍了数据库设计得范式
* 所谓得反范式化就是为了性能和读取效率得考虑而适当得对数据库设计范式得要求进行违反
* 允许存在少量得冗余，换句话来说反范式化就是使用空间来换取时间
* 订单表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **订单编号** | **下单用户名** | **手机号** | **下单日期** | **支付金额** | **物流单号** | **订单金额** |

* 订单商品关联表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **订单编号** | **订单商品分类** | **订单商品名** | **订单单价** | **订单商品数量** |

* 编写SQL查询出每一个用户的订单总金额

SELECT

下单用户名,

sum(订单金额)

FROM

订单表

GROUP BY

下单用户名;

* 编写SQL查询出下单用户和订单详情

SELECT

a.订单编号,

a.用户名,

a.手机号,

b.商品名称,

b.商品单价,

b.商品数量

FROM

订单表 a

JOIN 订单商品关联表 b ON a.订单编号 = b.订单编号

总结

不能完全按照范式得要求进行设计

考虑以后如何使用表

**范式化设计优缺点**

**优点：**

* 可以尽量得减少数据冗余
* 范式化的更新操作比反范式化更快
* 范式化的表通常比反范式化的表更小

**缺点：**

* 对于查询需要对多个表进行关联
* 更难进行索引优化

**反范式化设计优缺点**

**优点：**

* 可以减少表的关联
* 可以更好的进行索引优化

**缺点：**

* 存在数据冗余及数据维护异常
* 对数据的修改需要更多的成本

## 物理设计

### 进行存储结构的设计

**物理设计**

* **定义数据库、表及字段的命名规范**
* **选择合适的存储引擎**
* **为表中的字段选择合适的数据类型**
* **建立数据库结构**

### 命名规范

* **数据库、表、字段的命名要遵守可读性原则**

**使用大小写来格式化的库对象名字以获得良好的可读性**

**例如：使用custAddress而不是custaddress来提高可读性。**

* **数据库、表、字段的命名要遵守表意性原则**

**对象的名字应该能够描述它所表示的对象**

**例如：**

**对于表，表的名称应该能够体现表中存储的数据内容；对于存储过程**

**存储过程应该能够体现存储过程的功能。**

**数据库、表、字段的命名要遵守长名原则**

**尽可能少使用或者不使用缩写**

### 选择合适的存储引擎

|  |  |  |
| --- | --- | --- |
| **对比项** | **MyISAM** | **InnoDB** |
| 主外键 | 不支持 | 支持 |
| 事务 | 不支持 | 支持 |
| 行表锁 | 表锁，即使操作一条记录也会锁住整个表  不适合高并发的操作 | 行锁,操作时只锁某一行，不对其它行有影响  适合高并发的操作 |
| 缓存 | 只缓存索引，不缓存真实数据 | 不仅缓存索引还要缓存真实数据，对内存要求较高，而且内存大小对性能有决定性的影响 |
| 表空间 | 小 | 大 |
| 关注点 | 性能 | 事务 |
| 默认安装 | Y | Y |

### 为表中的字段选择合适的数据类型

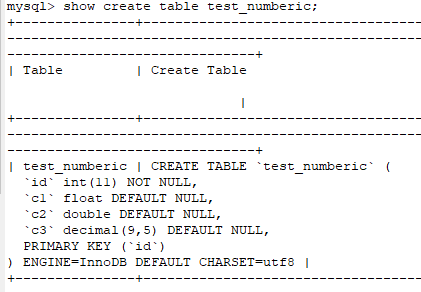
* **当一个列可以选择多种数据类型时**

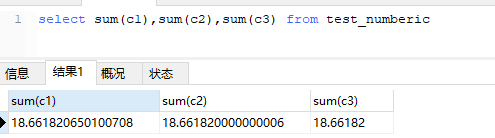
1. **优先考虑数字类型**
2. **其次是日期、时间类型**
3. **最后是字符类型**
4. **对于相同级别的数据类型，应该优先选择占用空间小的数据类型**

**浮点型:**

**常用的与钱有关的decimal(12,2)**

|  |  |  |
| --- | --- | --- |
| **列类型** | **存储空间** | **是否精确类型** |
| FlOAT | 4个字节 | 否 |
| DOUBLE | 8个字节 | 否 |
| DECIMAL | 每4个字节存9个数字，小数点占1个字节 | 是 |



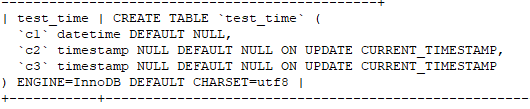


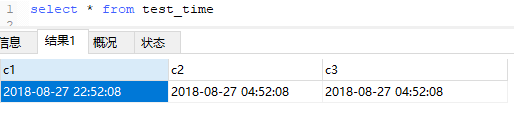
* **日期类型**

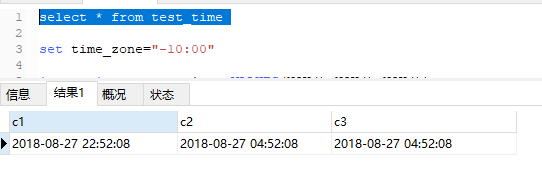
**timestamp 类型 与 datetime区别**

**timestamp 和时区有关，而datetime无关**









# 慢查询

慢查询日志，顾名思义，就是查询慢的日志，是指mysql记录所有执行超过long\_query\_time参数设定的时间阈值的SQL语句的日志。该日志能为SQL语句的优化带来很好的帮助。默认情况下，慢查询日志是关闭的，要使用慢查询日志功能，首先要开启慢查询日志功能。

参数查询 show VARIABLES like '%slow\_query\_log%'

参数设置 set global long\_query\_time=10

## 配置

* slow\_query\_log 启动停止技术慢查询日志
* slow\_query\_log\_file 指定慢查询日志得存储路径及文件（默认和数据文件放一起）
* long\_query\_time 指定记录慢查询日志SQL执行时间得伐值（单位：秒，默认10秒）
* log\_queries\_not\_using\_indexes 是否记录未使用索引的SQL
* log\_output 日志存放的地方【TABLE】【FILE】【FILE,TABLE】

配置了慢查询后，它会记录符合条件的SQL

包括：

* 查询语句
* 数据修改语句
* 已经回滚得SQL

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

show VARIABLES like '%slow\_query\_log\_file%'

show VARIABLES like '%long\_query\_time%'

show VARIABLES like '%log\_queries\_not\_using\_indexes%'

show VARIABLES like 'log\_output'

set global long\_query\_time=0; ---默认10秒，这里为了演示方便设置为0

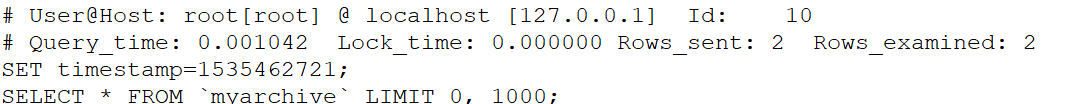
set GLOBAL slow\_query\_log = 1; --开启慢查询日志

set global log\_output='FILE,TABLE' --项目开发中日志只能记录在日志文件中，不能记表中

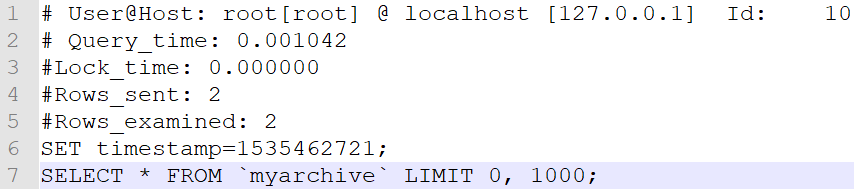
设置完成后，查询一些列表可以发现慢查询的日志文件里面有数据了。

## 解读

从慢查询日志里面摘选一条慢查询日志，数据组成如下



把为解读放吧，慢查询格式显示



第一行：用户名 、用户的IP信息、线程ID号

第二行：执行花费的时间【单位：毫秒】

第三行：执行获得锁的时间

第四行：获得的结果行数

第五行：扫描的数据行数

第六行：这SQL执行的具体时间

第七行：具体的SQL语句

## 分析

慢查询的日志记录非常多，要从里面找寻一条查询慢的日志并不是很容易的事情，一般来说都需要一些工具辅助才能快速定位到需要优化的SQL语句，下面介绍两个慢查询辅助工具

### Mysqldumpslow

常用的慢查询日志分析工具，汇总除查询条件外其他完全相同的SQL，并将分析结果按照参数中所指定的顺序输出。

**语法：**

mysqldumpslow -s r -t 10 slow-mysql.log

-s order (c,t,l,r,at,al,ar)

c:总次数

t:总时间

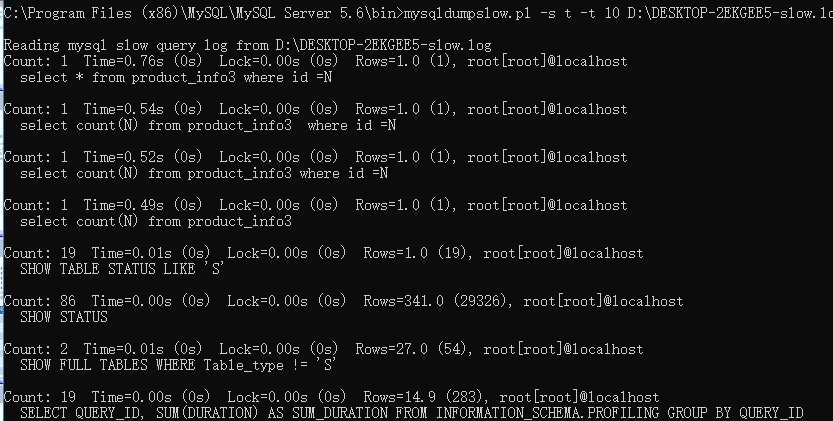
l:锁的时间

r:总数据行

at,al,ar :t,l,r平均数 【例如：at = 总时间/总次数】

-t top 指定取前面几天作为结果输出

mysqldumpslow.pl -s t -t 10 D:\DESKTOP-2EKGEE5-slow.log



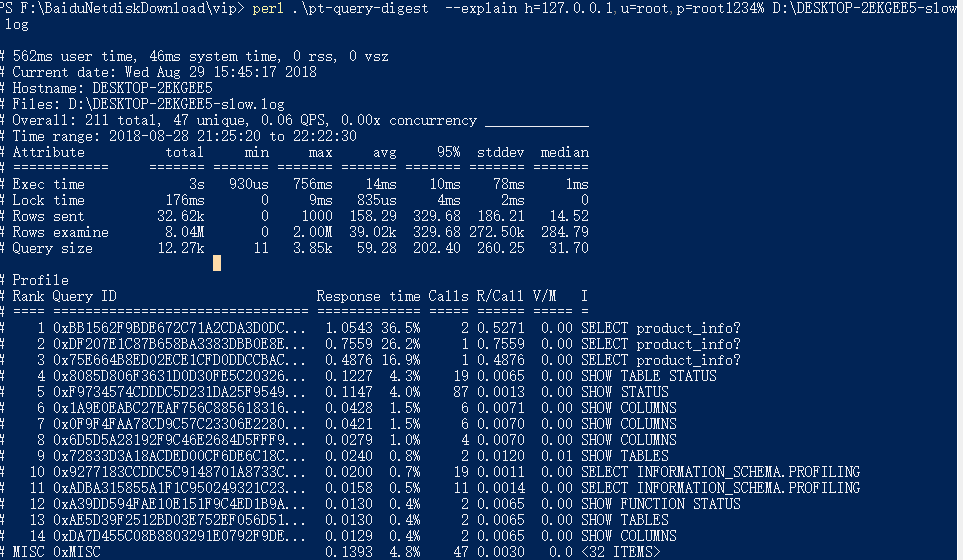
### pt\_query\_digest

是用于分析mysql慢查询的一个工具，与mysqldumpshow工具相比，py-query\_digest 工具的分析结果更具体，更完善。

有时因为某些原因如权限不足等，无法在服务器上记录查询。这样的限制我们也常常碰到。

首先来看下一个命令

perl .\pt-query-digest --explain h=127.0.0.1,u=root,p=root1234% D:\DESKTOP-2EKGEE5-slow.log



汇总的信息【总的查询时间】、【总的锁定时间】、【总的获取数据量】、【扫描的数据量】、【查询大小】

Response: 总的响应时间。

time: 该查询在本次分析中总的时间占比。

calls: 执行次数，即本次分析总共有多少条这种类型的查询语句。

R/Call: 平均每次执行的响应时间。

Item : 查询对象

**pt-query-digest语法及重要选项**

pt-query-digest [OPTIONS] [FILES] [DSN]

* --create-review-table 当使用--review参数把分析结果输出到表中时，如果没有表就自动创建。
* --create-history-table 当使用--history参数把分析结果输出到表中时，如果没有表就自动创建。
* --filter 对输入的慢查询按指定的[字符串](http://www.php.cn/wiki/57.html)进行匹配过滤后再进行分析
* --limit 限制输出结果百分比或数量，默认值是20,即将最慢的20条语句输出，如果是50%则按总响应时间占比从大到小排序，输出到总和达到50%位置截止。
* --host mysql服务器地址
* --user mysql用户名
* --password mysql用户密码
* --history 将分析结果保存到表中，分析结果比较详细，下次再使用--history时，如果存在相同的语句，且查询所在的时间区间和历史表中的不同，则会记录到数据表中，可以通过查询同一CHECKSUM来比较某类型查询的历史变化。
* --review 将分析结果保存到表中，这个分析只是对查询条件进行参数化，一个类型的查询一条记录，比较简单。当下次使用--review时，如果存在相同的语句分析，就不会记录到数据表中。
* --output 分析结果输出类型，值可以是report(标准分析报告)、slowlog(Mysql slow log)、[json](http://www.php.cn/wiki/1488.html)、json-anon，一般使用report，以便于阅读。
* --since 从什么时间开始分析，值为字符串，可以是指定的某个”yyyy-mm-dd [hh:mm:ss]”格式的时间点，也可以是简单的一个时间值：s(秒)、h(小时)、m(分钟)、d(天)，如12h就表示从12小时前开始统计。

--until 截止时间，配合—since可以分析一段时间内的慢查询。

**分析pt-query-digest输出结果**

**第一部分总体统计结果**

* Overall：总共有多少条查询
* Time range：查询执行的时间范围
* unique：唯一查询数量，即对查询条件进行参数化以后，总共有多少个不同的查询
* total：总计 min：最小 max：最大 avg：平均
* 95%：把所有值从小到大排列，位置位于95%的那个数，这个数一般最具有参考价值
* median：中位数，把所有值从小到大排列，位置位于中间那个数

# 该工具执行日志分析的用户时间，系统时间，物理内存占用大小，虚拟内存占用大小

# 340ms user time, 140ms system time, 23.99M rss, 203.11M vsz

# 工具执行时间

# Current date: Fri Nov 25 02:37:18 2016

# 运行分析工具的主机名

# Hostname: localhost.localdomain

# 被分析的文件名

# Files: slow.log

# 语句总数量，唯一的语句数量，QPS，并发数

# Overall: 2 total, 2 unique, 0.01 QPS, 0.01x concurrency

# 日志记录的时间范围

# Time range: 2016-11-22 06:06:18 to 06:11:40

# 属性 总计 最小 最大 平均 95% 标准 中等

# Attribute total min max avg 95% stddev median

# ============ ======= ======= ======= ======= ======= ======= =======

# 语句执行时间

# Exec time 3s 640ms 2s 1s 2s 999ms 1s

# 锁占用时间

# Lock time 1ms 0 1ms 723us 1ms 1ms 723us

# 发送到客户端的行数

# Rows sent 5 1 4 2.50 4 2.12 2.50

# select语句扫描行数

# Rows examine 186.17k 0 186.17k 93.09k 186.17k 131.64k 93.09k

# 查询的字符数

# Query size 455 15 440 227.50 440 300.52 227.50

**第二部分查询分组统计结果**

* Rank：所有语句的排名，默认按查询时间降序排列，通过--order-by指定
* Query ID：语句的ID，（去掉多余空格和文本字符，计算[hash](http://www.php.cn/wiki/762.html)值）
* Response：总的响应时间
* time：该查询在本次分析中总的时间占比
* calls：执行次数，即本次分析总共有多少条这种类型的查询语句
* R/Call：平均每次执行的响应时间
* V/M：响应时间Variance-to-mean的比率
* Item：查询[对象](http://www.php.cn/wiki/60.html)

# Profile

# Rank Query ID Response time Calls R/Call V/M Item

# ==== ================== ============= ===== ====== ===== ===============

# 1 0xF9A57DD5A41825CA 2.0529 76.2% 1 2.0529 0.00 SELECT

# 2 0x4194D8F83F4F9365 0.6401 23.8% 1 0.6401 0.00 SELECT wx\_member\_base

**第三部分：每一种查询的详细统计结果**

由下面查询的详细统计结果，最上面的[表格](http://www.php.cn/code/5947.html)列出了执行次数、最大、最小、平均、95%等各项目的统计。

* ID：查询的ID号，和上图的Query ID对应
* Databases：数据库名
* Users：各个用户执行的次数（占比）
* Query\_time distribution ：查询时间分布, 长短体现区间占比，本例中1s-10s之间查询数量是10s以上的两倍。
* Tables：查询中涉及到的表
* Explain：SQL语句

# Query 1: 0 QPS, 0x concurrency, ID 0xF9A57DD5A41825CA at byte 802

# This item is included in the report because it matches --limit.

# Scores: V/M = 0.00

# Time range: all events occurred at 2016-11-22 06:11:40

# Attribute pct total min max avg 95% stddev median

# ============ === ======= ======= ======= ======= ======= ======= =======

# Count 50 1

# Exec time 76 2s 2s 2s 2s 2s 0 2s

# Lock time 0 0 0 0 0 0 0 0

# Rows sent 20 1 1 1 1 1 0 1

# Rows examine 0 0 0 0 0 0 0 0

# Query size 3 15 15 15 15 15 0 15

# String:

# Databases test

# Hosts 192.168.8.1

# Users mysql

# Query\_time distribution

# 1us

# 10us

# 100us

# 1ms

# 10ms

# 100ms

# 1s ################################################################

# 10s+

# EXPLAIN /\*!50100 PARTITIONS\*/

select sleep(2)\G

# 索引与执行计划

## 索引

1MySQL官方对索引的定义为：索引（Index）是帮助MySQL高效获取数据的数据结构。

可以得到索引的本质：索引是数据结构

* 普通索引：即一个索引只包含单个列，一个表可以有多个单列索引。一个简单的查询语句，多个条件，只会用到一个索引。
* 唯一索引：索引列的值必须唯一，但允许有空值
* 复合索引：即一个索引包含多个列
* 聚簇索引(聚集索引)：并不是一种单独的索引类型，而是一种数据存储方式。具体细节取决于不同的实现，InnoDB的聚簇索引其实就是在同一个结构中保存了B-Tree索引(技术上来说是B+Tree)和数据行。
* 非聚簇索引：不是聚簇索引，就是非聚簇索引

show global variables like "%datadir%";

### 聚簇索引和非聚簇索引

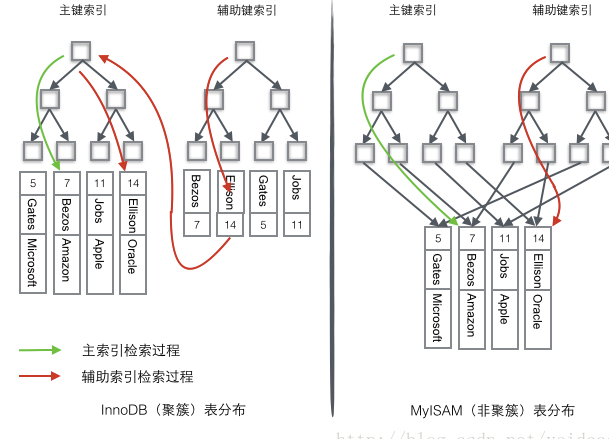
对于聚簇索引存储来说，行数据和主键B+树存储在一起，辅助键B+树只存储辅助键和主键，主键和非主键B+树几乎是两种类型的树。对于非聚簇索引存储来说，主键B+树在叶子节点存储指向真正数据行的指针，而非主键。每一个表都有一个聚簇索引，并且只有一个

InnoDB使用的是聚簇索引，将主键组织到一棵B+树中，而行数据就储存在叶子节点上，若使用"where id = 14"这样的条件查找主键，则按照B+树的检索算法即可查找到对应的叶节点，之后获得行数据。若对Name列进行条件搜索，则需要两个步骤：第一步在辅助索引B+树中检索Name，到达其叶子节点获取对应的主键。第二步使用主键在主索引B+树种再执行一次B+树检索操作，最终到达叶子节点即可获取整行数据。

MyISM使用的是非聚簇索引，非聚簇索引的两棵B+树看上去没什么不同，节点的结构完全一致只是存储的内容不同而已，主键索引B+树的节点存储了主键，辅助键索引B+树存储了辅助键。表数据存储在独立的地方，这两颗B+树的叶子节点都使用一个地址指向真正的表数据，对于表数据来说，这两个键没有任何差别。由于索引树是独立的，通过辅助键检索无需访问主键的索引树。

 以myisam为例，一个数据表table中，它是有table.frm、table.myd以及table.myi组成。table.myd记录了数据，table.myi记录了索引的数据。在用到索引时，先到table.myi（索引树）中进行查找，取到数据所在table.myd的行位置，拿到数据。所以**myisam引擎的索引文件和数据文件是独立分开的，则称之为非聚簇索引**。

如何设置聚簇索引呢? primary key(user\_Name) 这个就是聚簇索引索引



### 索引语法

* 查看索引

SHOW INDEX FROM table\_name\G

* 创建索引

CREATE [UNIQUE ] INDEX indexName ON mytable(columnname(length));

ALTER TABLE 表名 ADD [UNIQUE ] INDEX [indexName] ON (columnname(length))

* 删除索引

DROP INDEX [indexName] ON mytable;

## 执行计划

* 执行计划是什么

使用EXPLAIN关键字可以模拟优化器执行SQL查询语句，从而知道MySQL是如何处理你的SQL语句的。分析你的查询语句或是表结构的性能瓶颈

作用

* 表的读取顺序
* 数据读取操作的操作类型
* 哪些索引可以使用
* 哪些索引被实际使用
* 表之间的引用
* 每张表有多少行被优化器查询

执行计划的语法其实非常简单： 在SQL查询的前面加上EXPLAIN关键字就行。

比如：EXPLAIN select \* from table1

重点的就是EXPLAIN后面你要分析的SQL语句

通过EXPLAIN关键分析的结果由以下列组成，接下来挨个分析每一个列



### ID列

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

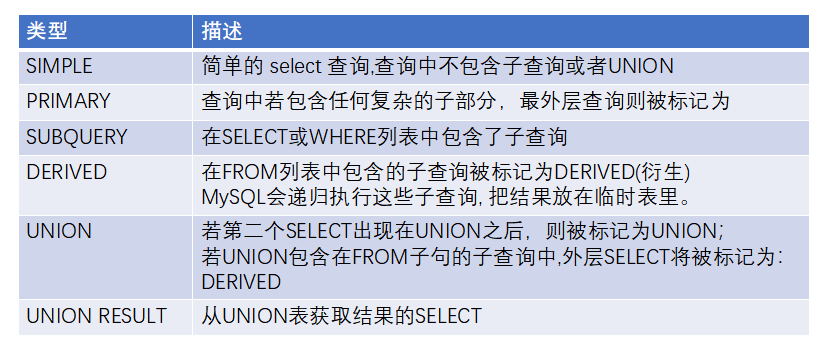
根据ID的数值结果可以分成一下三种情况

* 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。如果id相同，从上往下，在所有组中，id越大优先级越高。

### select\_type列

Select\_type:查询的类型，

要是用于区别:普通查询、联合查询、子查询等的复杂查询



### table列

显示这一行的数据是关于哪张表的

### Type列

|  |  |
| --- | --- |
| ALL | 全数据表扫描 |
| index | 全索引表扫描 |
| RANGE | 对索引列进行范围查找 |
| INDEX\_MERGE | 合并索引，使用多个单列索引搜索 |
| REF | 根据索引查找一个或多个值 |
| EQ\_REF | 搜索时使用primary key 或 unique类型 |
| CONST | 常量，表最多有一个匹配行,因为仅有一行,在这行的列值可被优化器剩余部分认为是常数,const表很快,因为它们只读取一次。 |
| SYSTEM | 系统，表仅有一行(=系统表)。这是const联接类型的一个特例。 |

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。

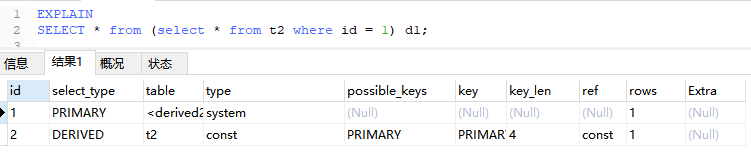
#### System与const

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

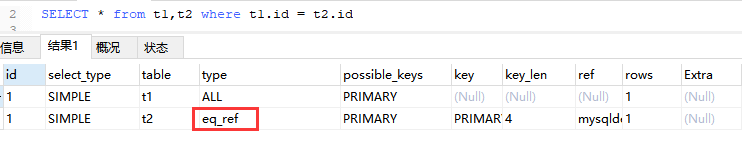
const用于比较primary key或者unique索引。因为只匹配一行数据，所以很快

如将主键置于where列表中，MySQL就能将该查询转换为一个常量



#### eq\_ref

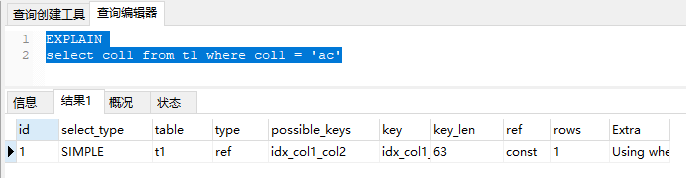
唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配。常见于主键或唯一索引扫描



#### 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

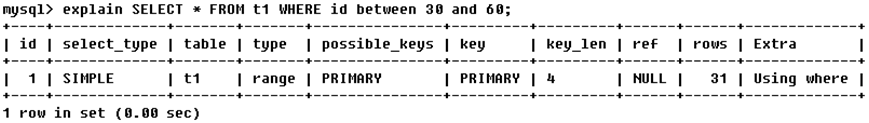


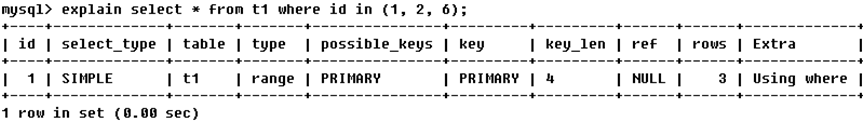
#### Range

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。

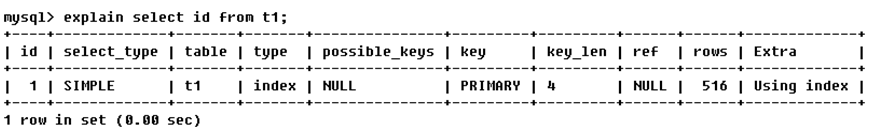




#### Index

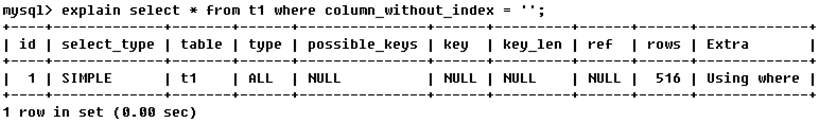
当查询的结果全为索引列的时候，虽然也是全部扫描，但是只查询的索引库，而没有去查询

数据。(覆盖索引)



#### All

Full Table Scan，将遍历全表以找到匹配的行



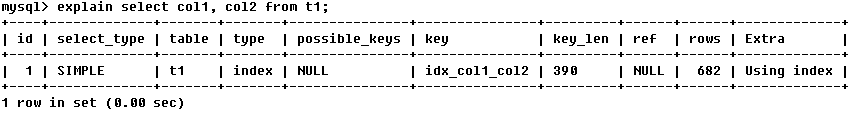
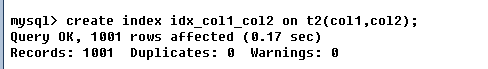
### possible\_keys 与Key 列

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

**覆盖索引**：如果一个索引包含所有需要查询的字段的值，则这个索引就为覆盖索引 .SQL不必通过二级索引(表上的每个非聚簇以外都是二级索引)查到主键之后再去查询数据。



其中key和possible\_keys都可以出现null的情况

### key\_len列

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

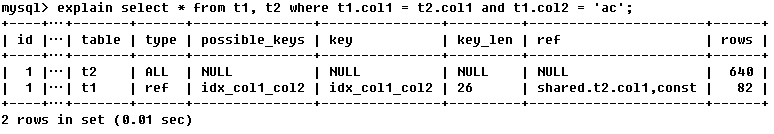
key\_len显示的值为索引字段的最大可能长度，并非实际使用长度，即key\_len是根据表定义计算而得，不是通过表内检索出的

* 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）**



### Ref列

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

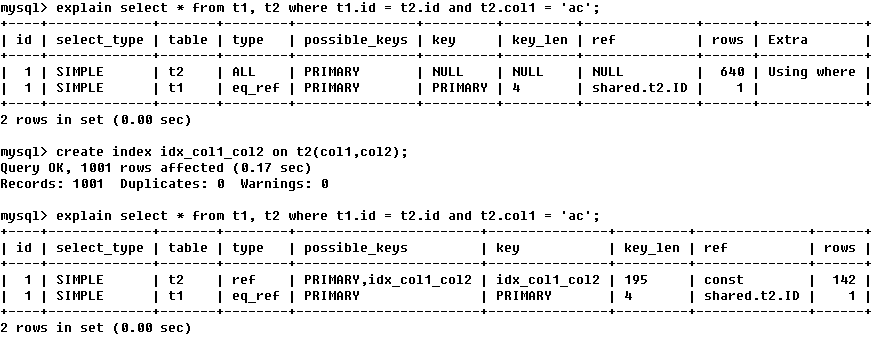


由key\_len可知t1表的idx\_col1\_col2被充分使用，col1匹配t2表的col1，col2匹配了一个常量，即 'ac'

其中 【shared.t2.col1】 为 【数据库.表.列】

### Rows

根据表统计信息及索引选用情况，大致估算出找到所需的记录所需要读取的行数



### Extra

包含不适合在其他列中显示但十分重要的额外信息。

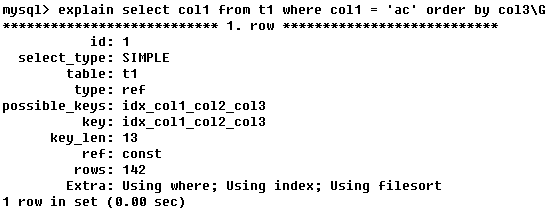


#### Using filesort

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顺序进行读取。

MySQL中无法利用索引完成的排序操作称为“文件排序”

当发现有Using filesort 后，实际上就是发现了可以优化的地方



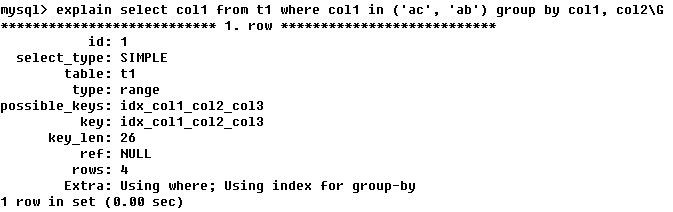
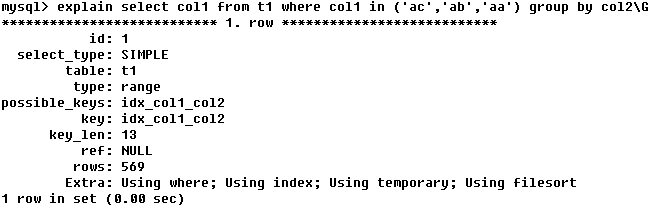
上图其实是一种索引失效的情况，后面会讲，可以看出查询中用到了个联合索引，索引分别为col1,col2,col3



当我排序新增了个col2，发现using filesort 就没有了。

#### Using temporary

使了用临时表保存中间结果,MySQL在对查询结果排序时使用临时表。常见于排序 order by 和分组查询 group by

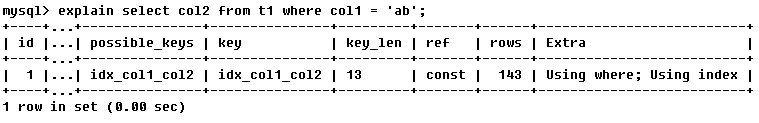


尤其发现在执行计划里面有using filesort而且还有Using temporary的时候，特别需要注意

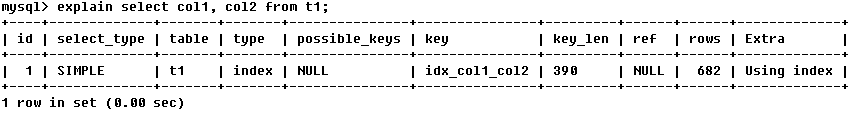
#### Using index

表示相应的select操作中使用了覆盖索引(Covering Index)，避免访问了表的数据行，效率不错！

**如果同时出现using where，表明索引被用来执行索引键值的查找;**



**如果没有同时出现using where，表明索引用来读取数据而非执行查找动作**



##### 覆盖索引：

如果一个索引包含(或覆盖)所有需要查询的字段的值，称为‘覆盖索引’。即只需扫描索引而无须回表。

优点：

    1.索引条目通常远小于数据行大小，只需要读取索引，则mysql会极大地减少数据访问量。

    2.因为索引是按照列值顺序存储的，所以对于IO密集的范围查找会比随机从磁盘读取每一行数据的IO少很多。

    3.一些存储引擎如myisam在内存中只缓存索引，数据则依赖于操作系统来缓存，因此要访问数据需要一次系统调用

    4.innodb的聚簇索引，覆盖索引对innodb表特别有用。(innodb的二级索引在叶子节点中保存了行的主键值，所以如果二级主键能够覆盖查询，则可以避免对主键索引的二次查询)

覆盖索引必须要存储索引列的值，而哈希索引、空间索引和全文索引不存储索引列的值，所以mysql**只能用B-tree索引**做覆盖索引。

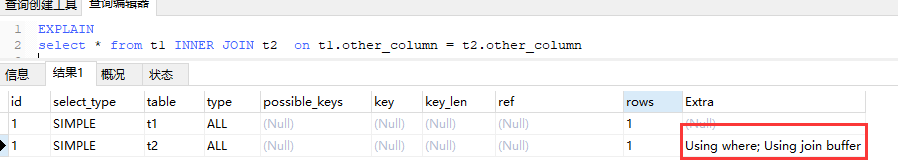
#### Using where 与 using join buffer

Using where

表明使用了where过滤

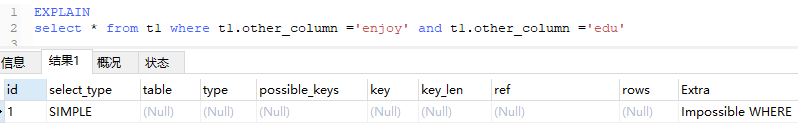
using join buffer

使用了连接缓存：



#### impossible where

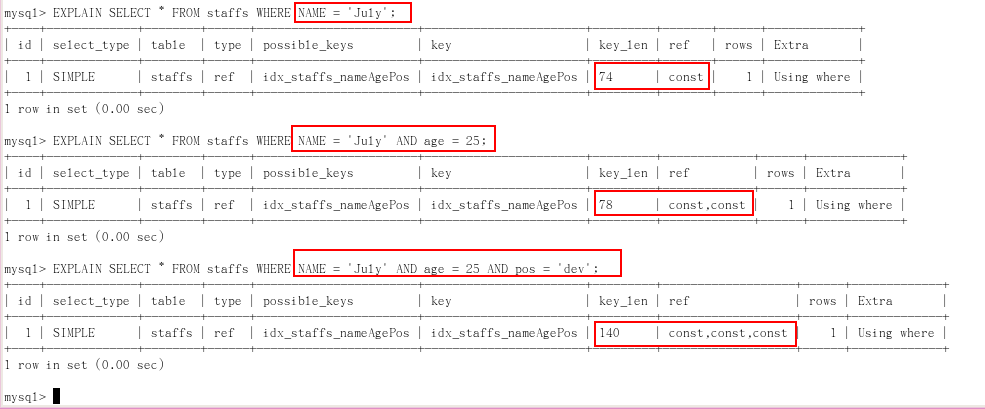
where子句的值总是false，不能用来获取任何元组



# SQL优化

## 优化实战

### 策略1.尽量全值匹配



CREATE TABLE `staffs`(

id int primary key auto\_increment,

name varchar(24) not null default "" comment'姓名',

age int not null default 0 comment '年龄',

pos varchar(20) not null default "" comment'职位',

add\_time timestamp not null default current\_timestamp comment '入职时间'

)charset utf8 comment '员工记录表';

insert into staffs(name,age,pos,add\_time) values('z3',22,'manage',now());

insert into staffs(name,age,pos,add\_time) values('july',23,'dev',now());

insert into staffs(name,age,pos,add\_time) values('2000',23,'dev',now());

alter table staffs add index idx\_staffs\_nameAgePos(name,age,pos);

EXPLAIN SELECT \* FROM staffs WHERE NAME = 'July';

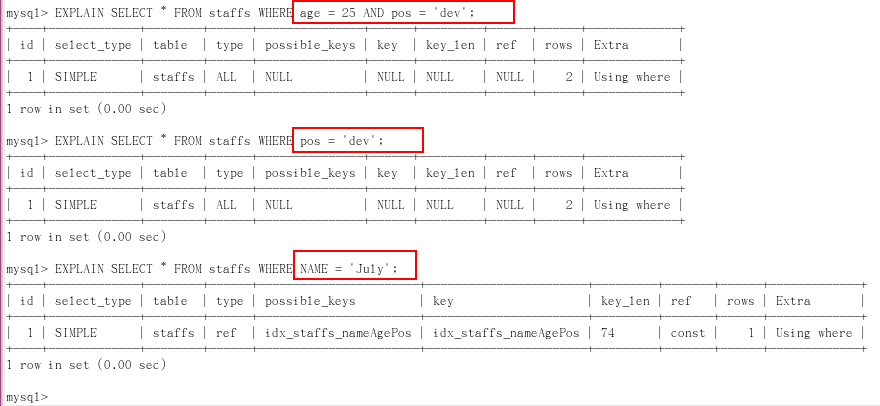
EXPLAIN SELECT \* FROM staffs WHERE NAME = 'July' AND age = 25;

EXPLAIN SELECT \* FROM staffs WHERE NAME = 'July' AND age = 25 AND pos = 'dev'

当建立了索引列后，能在wherel条件中使用索引的尽量所用。

### 策略2.最佳左前缀法则

如果索引了多列，要遵守最左前缀法则。指的是查询从索引的最左前列开始并且不跳过索引中的列。



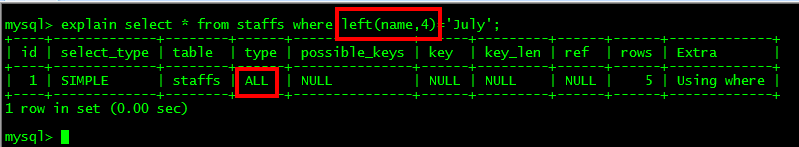
EXPLAIN SELECT \* FROM staffs WHERE age = 25 AND pos = 'dev'

EXPLAIN SELECT \* FROM staffs WHERE pos = 'dev'

EXPLAIN SELECT \* FROM staffs WHERE NAME = 'July'

### 策略3.不在索引列上做任何操作

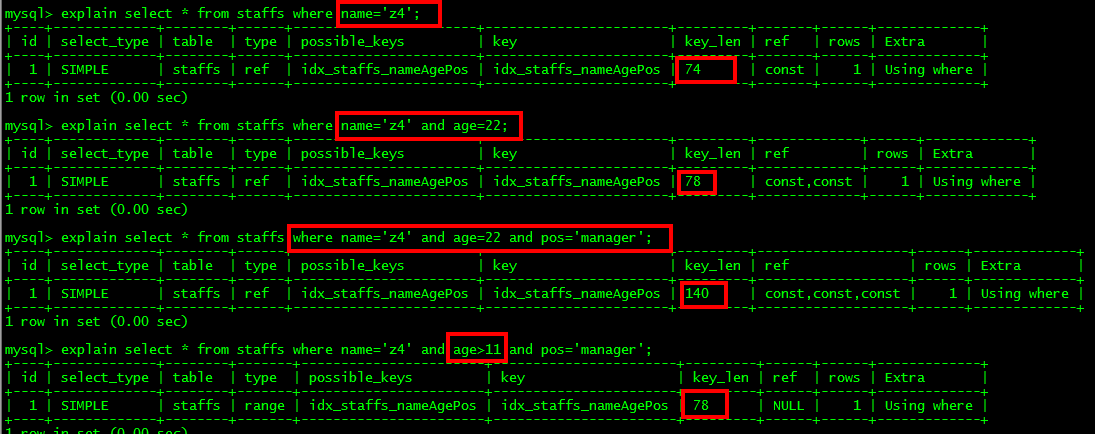
不在索引列上做任何操作（计算、函数、(自动or手动)类型转换），会导致索引失效而转向全表扫描



EXPLAIN SELECT \* FROM staffs WHERE NAME = 'July';

EXPLAIN SELECT \* FROM staffs WHERE left(NAME,4) = 'July';

### 策略4.范围条件放最后



EXPLAIN SELECT \* FROM staffs WHERE NAME = 'July' ;

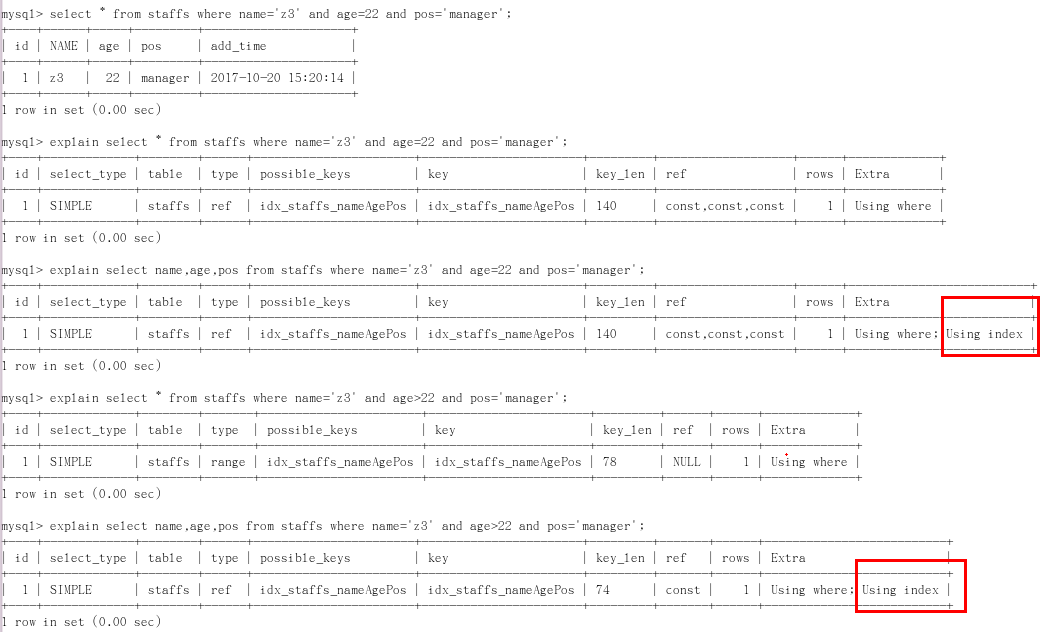
EXPLAIN SELECT \* FROM staffs WHERE NAME = 'July' and age =22;

EXPLAIN SELECT \* FROM staffs WHERE NAME = 'July' and age =22 and pos='manager'

中间有范围查询会导致后面的索引列全部失效（联合索引的情况，一条simple查询只能使用一个索引）

EXPLAIN SELECT \* FROM staffs WHERE NAME = 'July' and age >22 and pos='manager'

### 策略5.覆盖索引尽量用



尽量使用覆盖索引(只访问索引的查询(索引列和查询列一致))，减少select \*

EXPLAIN SELECT \* FROM staffs WHERE NAME = 'July' and age =22 and pos='manager'

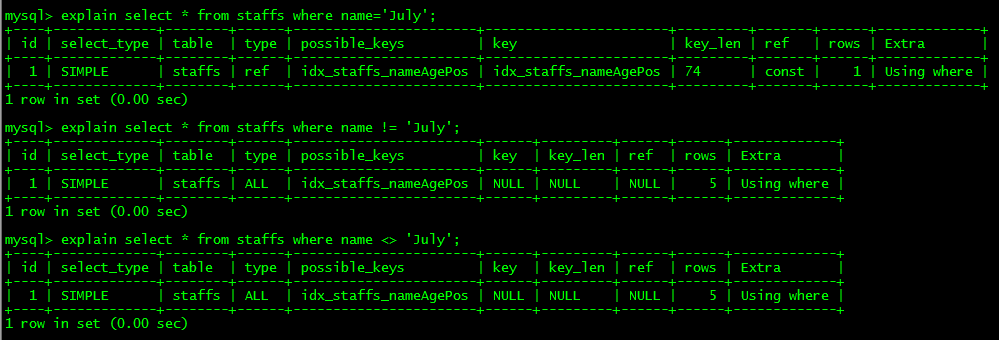
EXPLAIN SELECT name,age,pos FROM staffs WHERE NAME = 'July' and age =22 and pos='manager'

EXPLAIN SELECT \* FROM staffs WHERE NAME = 'July' and age >22 and pos='manager'

EXPLAIN SELECT name,age,pos FROM staffs WHERE NAME = 'July' and age >22 and pos='manager'

### 策略6.不等于要甚用

mysql 在使用不等于(!= 或者<>)的时候无法使用索引会导致全表扫描



EXPLAIN SELECT \* FROM staffs WHERE NAME = 'July';

EXPLAIN SELECT \* FROM staffs WHERE NAME != 'July';

EXPLAIN SELECT \* FROM staffs WHERE NAME <> 'July';

**如果定要需要使用不等于,请用覆盖索引**

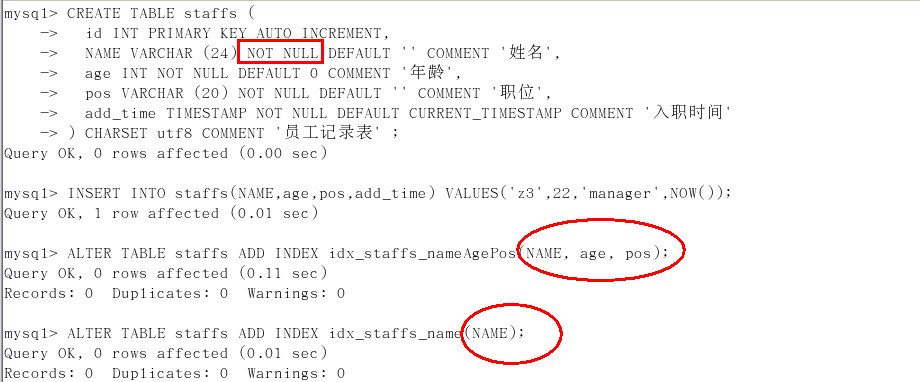
**EXPLAIN SELECT name,age,pos FROM staffs WHERE NAME != 'July';**

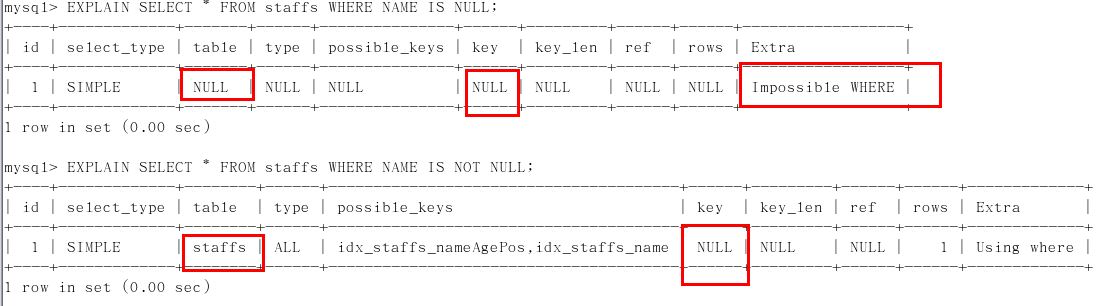
**EXPLAIN SELECT name,age,pos FROM staffs WHERE NAME <> 'July';**

### 策略7.Null/Not 有影响

**注意null/not null对索引的可能影响**

#### 自定定义为NOT NULL





EXPLAIN select \* from staffs where name is null

EXPLAIN select \* from staffs where name is not null

在字段为not null的情况下，使用is null 或 is not null 会导致索引失效

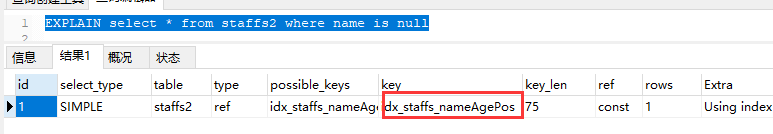
解决方式：覆盖索引

EXPLAIN select name,age,pos from staffs where name is not null

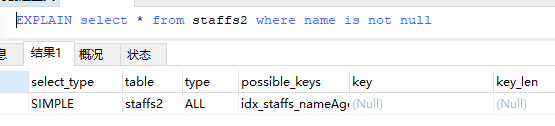
#### 自定义为NULL或者不定义



EXPLAIN select \* from staffs2 where name is null



EXPLAIN select \* from staffs2 where name is not null



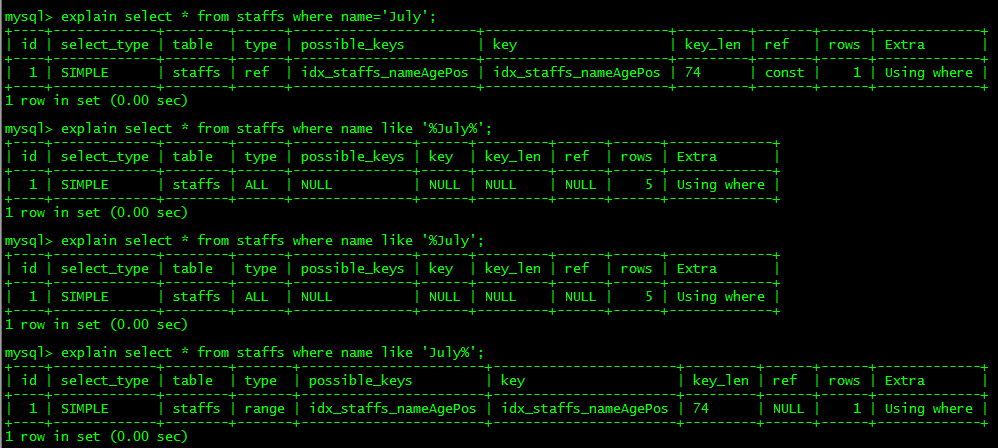
Is not null 的情况会导致索引失效

解决方式：覆盖索引

EXPLAIN select name,age,pos from staffs where name is not null

### 策略8.Like查询要当心

like以通配符开头('%abc...')mysql索引失效会变成全表扫描的操作



EXPLAIN select \* from staffs where name ='july'

EXPLAIN select \* from staffs where name like '%july%'

EXPLAIN select \* from staffs where name like '%july'

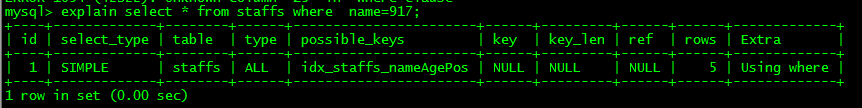
EXPLAIN select \* from staffs where name like 'july%'

解决方式：覆盖索引

EXPLAIN select name,age,pos from staffs where name like '%july%'

### 策略9.字符类型加引号

字符串不加单引号索引失效



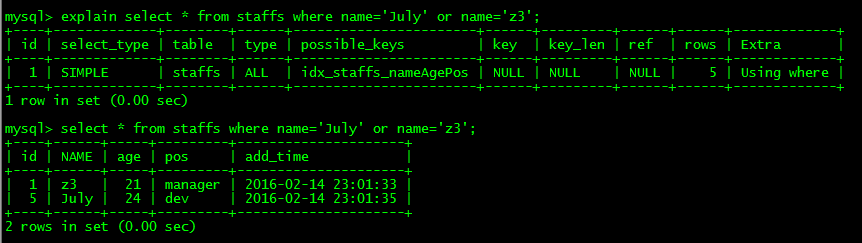
EXPLAIN select \* from staffs where name = 917

~~解决方式：覆盖索引~~

~~EXPLAIN select name,age,pos from staffs where name = 917~~

解决方式：请加引号

### 策略10.OR改UNION效率高



EXPLAIN

select \* from staffs where name='July' or name = 'z3'

EXPLAIN

select \* from staffs where name='July'

UNION

select \* from staffs where name = 'z3'

解决方式：覆盖索引

EXPLAIN

select name,age from staffs where name='July' or name = 'z3'

### 测试题



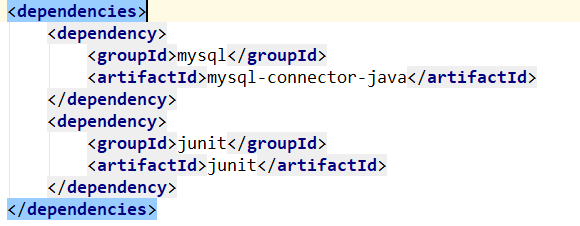
答案：



记忆总结：

* 全职匹配我最爱，最左前缀要遵守；
* 带头大哥不能死，中间兄弟不能断；
* 索引列上少计算，范围之后全失效；
* LIKE百分写最右，覆盖索引不写\*；
* 不等空值还有OR，索引影响要注意；
* VAR引号不可丢， SQL优化有诀窍。

## 批量导入



### insert语句优化；

* **提交前关闭自动提交**
* **尽量使用批量insert语句**
* **可以使用MyISAM存储引擎**

### LOAD DATA INFLIE

LOAD DATA INFLIE；

使用LOAD DATA INFLIE ,比一般的insert语句快20倍

select \* into OUTFILE 'D:\\product.txt' from product\_info

load data INFILE 'D:\\product.txt' into table product\_info

show VARIABLES like 'secure\_file\_priv'

secure\_file\_priv 为 NULL 时，表示限制mysqld不允许导入或导出。

secure\_file\_priv 为 /tmp 时，表示限制mysqld只能在/tmp目录中执行导入导出，其他目录不能执行。

secure\_file\_priv 没有值时，表示不限制mysqld在任意目录的导入导出。

# 参数查询设置

参数查询 show VARIABLES like '%slow\_query\_log%'

参数设置 set global long\_query\_time=10