MySQL高级

尚硅谷JavaEE

版本 V1.0

# 一、MySQL简介

## 1.什么是Mysql

* MySQL是一个关系型数据库管理系统，由瑞典MySQL AB公司开发，目前属于Oracle公司。
* Mysql是开源的，可以定制的，采用了GPL【都是开源的】协议，你可以修改源码来开发自己的Mysql系统。
* MySQL使用标准的SQL数据语言形式。
* Mysql可以允许于多个系统上，并且支持多种语言。这些编程语言包括C、C++、Python、Java、Perl、PHP、Eiffel、Ruby和Tcl等。
* MySQL支持大型数据库，支持5000万条记录的数据仓库，32位系统表文件最大可支持4GB，64位系统支持最大的表文件为8TB。

## 2. Mysql高手是怎样练成的

* 数据库内部结构和原理
* 数据库建模优化
* 数据库索引建立
* SQL语句优化
* SQL编程(自定义函数、存储过程、触发器、定时任务)
* mysql服务器的安装配置
* 数据库的性能监控分析与系统优化
* 各种参数常量设定
* 主从复制
* 分布式架构搭建、垂直切割和水平切割
* 数据迁移
* 容灾备份和恢复
* shell或python等脚本语言开发
* 对开源数据库进行二次开发

## 2.在Linux上安装Mysql

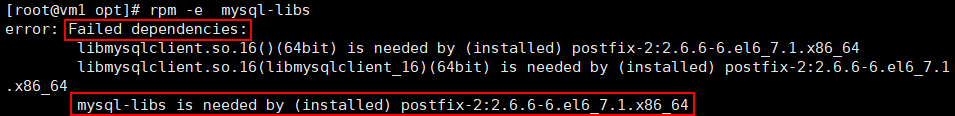
### 2.1 检查当前系统是否安装过Mysql

命令：rpm -qa|grep mysql



默认Linux在安装的时候，自带了mysql相关的组件。

先卸载系统自带的mysql，执行卸载命令rpm -e --nodeps mysql-libs

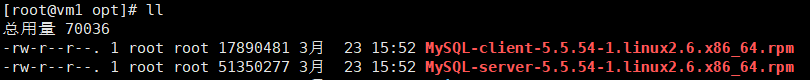




### 2.2 Mysql的安装

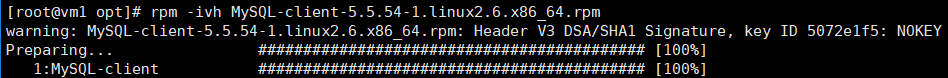
安装的版本是mysql 5.5，官网下载地址：<http://dev.mysql.com/downloads/mysql/>

①将rpm安装包拷贝到opt目录下

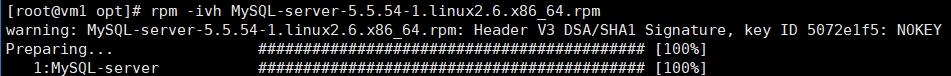


②在安装目录下执行rpm安装

rpm -ivh MySQL-client-5.5.54-1.linux2.6.x86\_64.rpm



rpm -ivh MySQL-server-5.5.54-1.linux2.6.x86\_64.rpm



安装完成后，出现如下警告，需要为软件设置root用户的密码。





③查看是否安装成功：mysqladmin --version



或者也可以通过rpm命令来查看：



④设置用户和密码： mysqladmin –u root password xxxxxx



rpm -qa | grep MySQL // 查看所有MySQL安装包的存储位置

通过find去查找某个文件 find / -name mysql

### 2.3 Mysql服务

#### 2.3.1 Mysql服务的启动和停止

查看状态：service mysql status



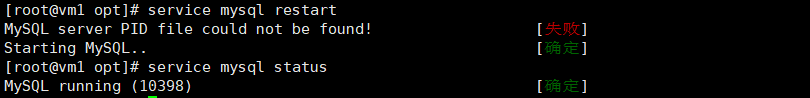
启动服务：service mysql start



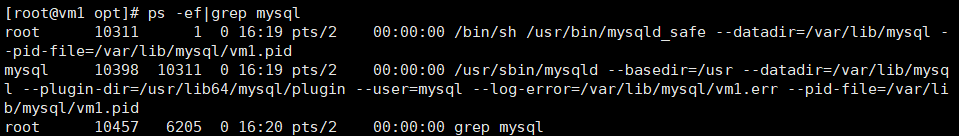
停止服务：service mysql stop



重启服务：service mysql restart



启动之后，查看进程：ps -ef | grep mysql



#### 2.3.2 Mysql的安装位置

|  |  |  |  |
| --- | --- | --- | --- |
| 参数 | 路径 | 解释 | 备注 |
| --datadir | /var/lib/mysql/ | mysql数据库文件的存放路径 |  |
| --basedir | /usr/bin | 相关命令目录 | mysqladmin mysqldump等命令 |
| --plugin-dir | /usr/lib64/mysql/plugin | mysql插件存放路径 |  |
| --log-error | /var/lib/mysql/jack.atguigu.err | mysql错误日志路径 |  |
| --pid-file | /var/lib/mysql/jack.atguigu.pid | 进程pid文件 |  |
| --socket | /var/lib/mysql/mysql.sock | 本地连接时用的unix套接字文件 |  |
|  | /usr/share/mysql | 配置文件目录 | mysql脚本及配置文件 |
|  | /etc/init.d/mysql | 服务启停相关脚本 |  |

#### 2.3.3 Mysql服务的自启动

Mysql服务是开机自动启动的！



如果要取消开机自启动，则输入命令ntsysv

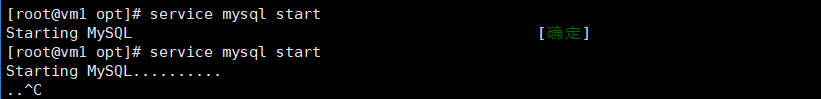


出现以下界面：

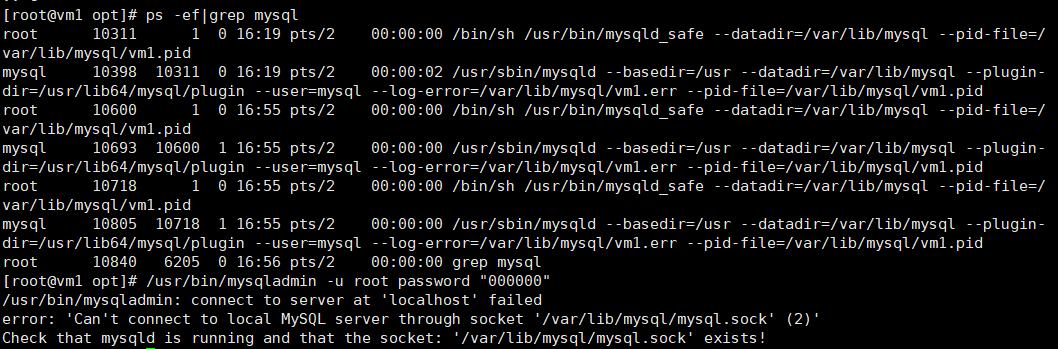


使用空格取消选中，然后按TAB确定！

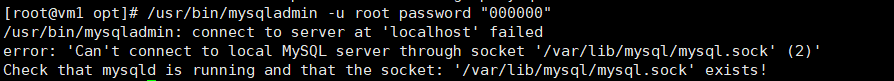
#### 2.3.4 Mysql的重复启动问题



此时查看，多了很多进程：



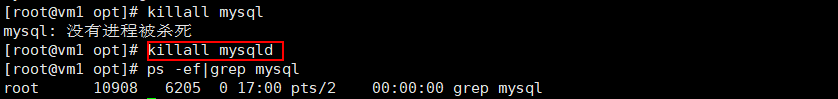
尝试去登录或者操作：报错！



查看服务状态：



解决：杀死所有和mysql进程相关的操作，然后重启服务！



注意是mysqld，d代表demon，守护进程。

然后再重启：



### 2.4 修改字符集

#### 2.4.1 常用命令

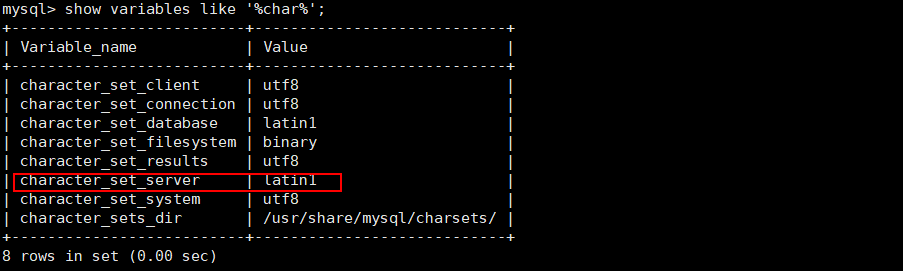
|  |  |  |
| --- | --- | --- |
| SQL语句 | 描述 | 备注 |
| show databases | 列出所有数据库 |  |
| create database 库名 | 创建一个数据库 |  |
| create database 库名 character set utf8 | 创建数据库，顺便执行字符集为utf-8 |  |
| show create database 库名 | 查看数据库的字符集 |  |
| show variables like ‘%char%’ | 查询所有跟字符集相关的信息 |  |
| set [字符集属性]=utf8 | 设置相应的属性为utf8 | 只是临时修改，当前有效。服务重启后，失效。 |
| alter database 库名character set 'utf8' | 修改数据库的字符集 |  |
| alter table 表名convert to character set 'utf8' | 修改表的字符集 |  |

实验SQL:

|  |
| --- |
| CREATE database mydb; |
| CREATE table mytable(id int,name varchar(30)); |
| insert into mytable(id,name) values (1,'jack'); |
| insert into mytable(id,name) values (2,'张三') |

#### 2.4.2 字符集乱码原因

如果在建库建表的时候，没有明确指定字符集，则采用默认的字符集latin1,其中是不包含中文字符的。查看默认的编码字符集：



#### 2.4.3 永久修改

在/usr/share/mysql/ 中找到my.cnf的配置文件，拷贝其中的my-huge.cnf 到 /etc/ 并命名为my.cnf 。添加以下内容后再重启服务。

注意：必须将文件拷贝到指定路径，且名称为my.cnf

[client]

default-character-set=utf8

[mysqld]

character\_set\_server=utf8

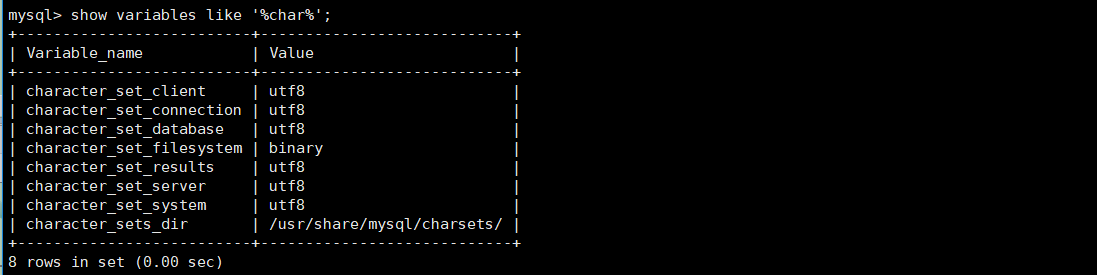
character\_set\_client=utf8

collation-server=utf8\_general\_ci

[mysql]

default-character-set=utf8

再次查看：

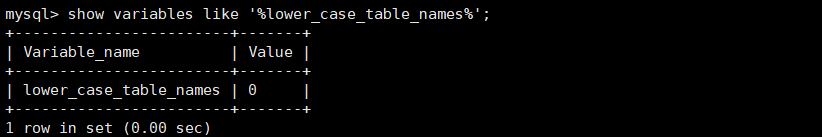


注意：已经创建的数据库的设定不会发生变化，参数修改只对新建的数据库有效！

### 2.5 设置大小写不敏感

①查看大小写是否敏感：show variables like '%lower\_case\_table\_names%'

windows系统默认大小写不敏感【1】，但是linux系统是大小写敏感的【0】



②设置大小写不敏感：在my.cnf这个配置文件 [mysqld] 中加入 lower\_case\_table\_names = 1 ，然后重启服务器

|  |  |
| --- | --- |
| 属性设置 | 描述 |
| 0 | 大小写敏感 |
| 1 | 大小写不敏感。创建的表，数据库都是以小写形式存放在磁盘上，对于sql语句都是转换为小写对表和DB进行查找 |
| 2 | 创建的表和DB依据语句上格式存放，凡是查找都是转换为小写进行 |

注意：如果要设置属性为大小写不敏感，要在重启数据库实例之前就需要将原来的数据库和表转换为小写，否则将找不到数据库名。在进行数据库参数设置之前，需要掌握这个参数带来的影响，切不可盲目设置。

### 2.6 sql\_mode

sql\_mode定义了对Mysql中sql语句语法的校验规则！

sql\_mode是个很容易被忽视的变量，默认值是空值，在这种设置下是可以允许一些非法操作的，比如允许一些非法数据的插入。在生产环境必须将这个值设置为严格模式，所以开发、测试环境的数据库也必须要设置，这样在开发测试阶段就可以发现问题。

#### 2.6.1 sql\_mode常用的值

|  |  |
| --- | --- |
| ONLY\_**FULL\_**GROUP\_BY | 对于GROUP BY聚合操作，如果在SELECT中的列，没有在GROUP BY中出现，那么这个SQL是不合法的，因为列不在GROUP BY从句中 |
| NO\_AUTO\_VALUE\_ON\_ZERO | 该值影响自增长列的插入。默认设置下，插入0或NULL代表生成下一个自增长值。如果用户 希望插入的值为0，而该列又是自增长的，那么这个选项就有用了 |
| STRICT\_TRANS\_TABLES | 在该模式下，如果一个值不能插入到一个事务表中，则中断当前的操作，对非事务表不做限制 |
| NO\_ZERO\_IN\_DATE | 在严格模式下，不允许日期和月份为零 |
| NO\_ZERO\_DATE | 设置该值，mysql数据库不允许插入零日期，插入零日期会抛出错误而不是警告 |
| ERROR\_FOR\_DIVISION\_BY\_ZERO | 在INSERT或UPDATE过程中，如果数据被零除，则产生错误而非警告。如 果未给出该模式，那么数据被零除时MySQL返回NULL |
| NO\_AUTO\_CREATE\_USER | 禁止GRANT创建密码为空的用户 |
| NO\_ENGINE\_SUBSTITUTION | 如果需要的存储引擎被禁用或未编译，那么抛出错误。不设置此值时，用默认的存储引擎替代，并抛出一个异常 |
| PIPES\_AS\_CONCAT | 将"||"视为字符串的连接操作符而非或运算符，这和Oracle数据库是一样的，也和字符串的拼接函数Concat相类似 |
| ANSI\_QUOTES | 启用ANSI\_QUOTES后，不能用双引号来引用字符串，因为它被解释为识别符 |
| ORACLE | 设置等同于PIPES\_AS\_CONCAT, ANSI\_QUOTES, IGNORE\_SPACE, NO\_KEY\_OPTIONS, NO\_TABLE\_OPTIONS, NO\_FIELD\_OPTIONS, NO\_AUTO\_CREATE\_USER |

#### 2.6.2 查看和修改

①查看当前的sql\_mode: select @@sql\_mode;

②sql\_mode的影响案例：group by 查询语法错误！

|  |
| --- |
| CREATE TABLE mytbl2 (id INT,NAME VARCHAR(200),age INT,dept INT);  INSERT INTO mytbl2 VALUES(1,'zhang3',33,101);  INSERT INTO mytbl2 VALUES(2,'li4',34,101);  INSERT INTO mytbl2 VALUES(3,'wang5',34,102);  INSERT INTO mytbl2 VALUES(4,'zhao6',34,102);  INSERT INTO mytbl2 VALUES(5,'tian7',36,102); |
| 查询每个dept中年龄最大的人：SELECT NAME,dept,MAX(age) FROM mytbl2 GROUP BY dept; |
| 正确写法：SELECT id,name,ab.dept,ab.maxage FROM mytbl2 m INNER JOIN(SELECT dept,MAX(age)maxage FROM mytbl2 GROUP BY dept)ab ON ab.dept=m.dept AND m.age=ab.maxage; |

③临时修改 sql\_mode: set @@sql\_mode=’’;

④永久修改，需要在配置文件my.cnf中修改：

[mysqld] 下添加 sql\_mode='' 然后重启mysql即可

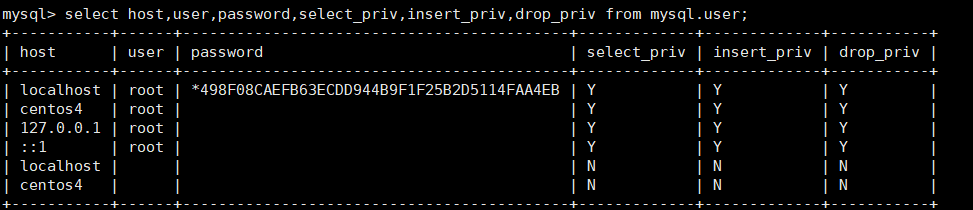
# 二、MySql的用户和权限管理

## 1. Mysql的用户管理

### 1.1 相关命令

|  |  |  |
| --- | --- | --- |
| 命令 | 描述 | 备注 |
| create user zhang3 identified by '123123'; | 创建名称为zhang3的用户，密码设为123123； |  |
| select host,user,password,select\_priv,insert\_priv,drop\_priv from mysql.user; | 查看用户和权限的相关信息 |  |
| set password =password('123456') | 修改当前用户的密码 |  |
| update mysql.user set password=password('123456') where user='li4'; | 修改其他用户的密码 | 所有通过user表的修改，必须用flush privileges;命令才能生效 |
| update mysql.user set user='li4' where user='wang5'; | 修改用户名 | 所有通过user表的修改，必须用flush privileges;命令才能生效 |
| drop user li4 | 删除用户 | 不要通过delete from user u where user='li4' 进行删除，系统会有残留信息保留。 |

### 1.2 示例说明



**host** :表示连接类型

% 表示所有远程通过 TCP方式的连接

IP 地址 如 (192.168.1.2,127.0.0.1) 通过制定ip地址进行的TCP方式的连接

机器名 通过制定i网络中的机器名进行的TCP方式的连接

::1 IPv6的本地ip地址 等同于IPv4的 127.0.0.1

localhost 本地方式通过命令行方式的连接 ，比如mysql -u xxx -p 123xxx 方式的连接。

**user**:表示用户名

同一用户通过不同方式链接的权限是不一样的。

**password**:密码

所有密码串通过 password(明文字符串) 生成的密文字符串。加密算法为MYSQLSHA1 ，不可逆 。

mysql 5.7 的密码保存到 authentication\_string 字段中不再使用password 字段。

**select\_priv , insert\_priv**等

为该用户所拥有的权限。

## 2. Mysql的权限管理

### 2.1 授予权限

|  |  |
| --- | --- |
| 命令 | 描述 |
| grant 权限1,权限2,…权限n on 数据库名称.表名称 to 用户名@用户地址 identified by ‘连接口令’ | 该权限如果发现没有该用户，则会直接新建一个用户。  示例：  grant select,insert,delete,drop on atguigudb.\* to li4@localhost ;  给li4用户用本地命令行方式下，授予atguigudb这个库下的所有表的插删改查的权限。 |
| grant all privileges on \*.\* to joe@'%' identified by '123'; | 授予通过网络方式登录的的joe用户 ，对所有库所有表的全部权限，密码设为123. |

### 2.2 收回权限

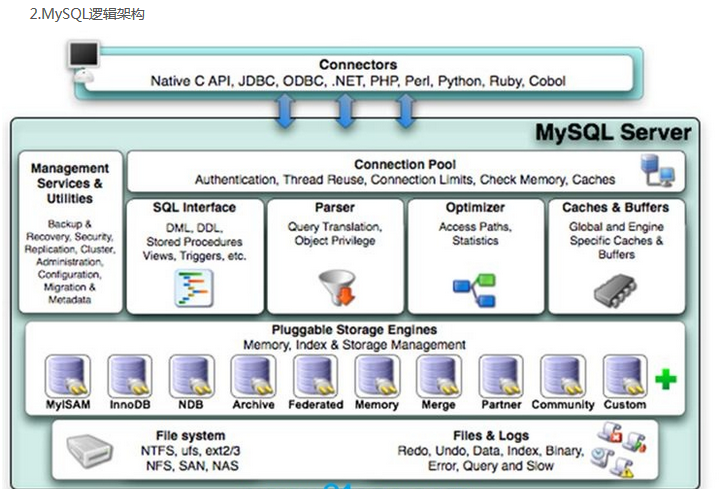
|  |  |  |
| --- | --- | --- |
| 命令 | 描述 | 备注 |
| show grants | 查看当前用户权限 |  |
| revoke [权限1,权限2,…权限n] on  库名.表名 from 用户名@用户地址 ; | 收回权限命令 |  |
| REVOKE ALL PRIVILEGES ON mysql.\* FROM joe@localhost; | 收回全库全表的所有权限 |  |
| REVOKE select,insert,update,delete ON mysql.\* FROM joe@localhost; | 收回mysql库下的所有表的插删改查权限 |  |

权限收回后，必须用户重新登录后，才能生效。

select host,user,password from mysql.user;

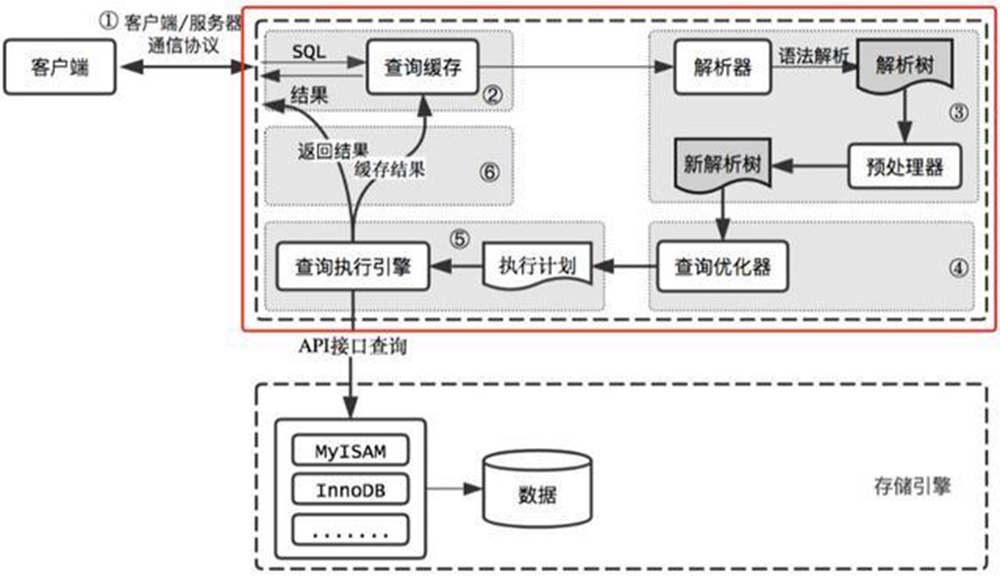
# 三、Mysql逻辑架构简介

## 1. 整体架构图



8以后没有了缓存使用redis实现





一条 SQL 语句过来的流程是什么样的？

①当客户端连接到 MySQL 服务器时，服务器对其进行认证。可以通过用户名与密码认证，也可以通过 SSL 证书进行认证。登录认证后，服务器还会验证客户端是否有执行某个查询的操作权限。

②在正式查询之前，服务器会检查查询缓存，如果能找到对应的查询，则不必进行查询解析，优化，执行等过程，直接返回缓存中的结果集。

③MySQL 的解析器会根据查询语句，构造出一个解析树，主要用于根据语法规则来验证语句是否正确，比如 SQL 的关键字是否正确，关键字的顺序是否正确。而预处理器主要是进一步校验，比如表名，字段名是否正确等。

④查询优化器将解析树转化为查询计划，一般情况下，一条查询可以有很多种执行方式，最终返回相同的结果，优化器就是根据成本找到这其中最优的执行计划。

⑤执行计划调用查询执行引擎，而查询引擎通过一系列 API 接口查询到数据。

⑥得到数据之后，在返回给客户端的同时，会将数据存在查询缓存中。

和其它数据库相比，MySQL有点与众不同，它的架构可以在多种不同场景中应用并发挥良好作用。主要体现在存储引擎的架构上，插件式的存储引擎架构将查询处理和其它的系统任务以及数据的存储提取相分离。这种架构可以根据业务的需求和实际需要选择合适的存储引擎。

各层介绍：

### 1.1 连接层

最上层是一些客户端和连接服务，包含本地sock通信和大多数基于客户端/服务端工具实现的类似于tcp/ip的通信。主要完成一些类似于连接处理、授权认证、及相关的安全方案。在该层上引入了线程池的概念，为通过认证安全接入的客户端提供线程。同样在该层上可以实现基于SSL的安全链接。服务器也会为安全接入的每个客户端验证它所具有的操作权限。

### 1.2 服务层

|  |  |
| --- | --- |
| Management Serveices & Utilities | 系统管理和控制工具 |
| SQL Interface: | SQL接口。接受用户的SQL命令，并且返回用户需要查询的结果。比如select from就是调用SQL Interface |
| Parser | 解析器。 SQL命令传递到解析器的时候会被解析器验证和解析 |
| Optimizer | 查询优化器。 SQL语句在查询之前会使用查询优化器对查询进行优化，比如有where条件时，优化器来决定先投影还是先过滤。 |
| Cache和Buffer | 查询缓存。如果查询缓存有命中的查询结果，查询语句就可以直接去查询缓存中取数据。这个缓存机制是由一系列小缓存组成的。比如表缓存，记录缓存，key缓存，权限缓存等 |

### 1.3.引擎层

存储引擎层，存储引擎真正的负责了MySQL中数据的存储和提取，服务器通过API与存储引擎进行通信。不同的存储引擎具有的功能不同，这样我们可以根据自己的实际需要进行选取。

### 1.4.存储层

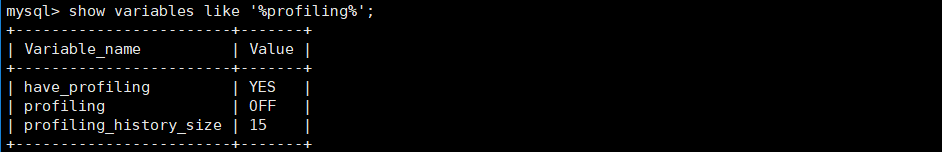
数据存储层，主要是将数据存储在运行于裸设备的文件系统之上，并完成与存储引擎的交互。

## 2. show profile

利用show profile可以查看sql的执行周期！

### 2.1 开启profile

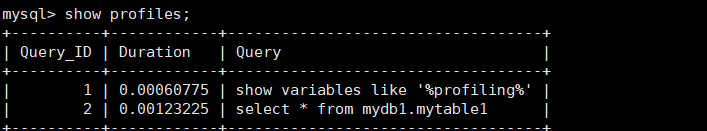
查看profile是否开启：show variables like '%profiling%'



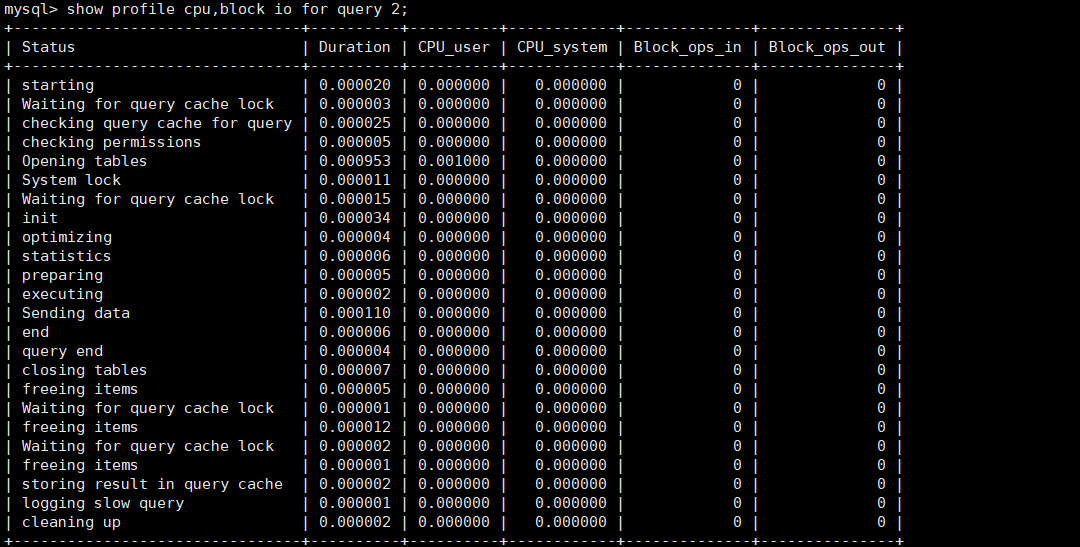
如果没有开启，可以执行set profiling=1开启！

### 2.2 使用profile

执行show prifiles命令，可以查看最近的几次查询。



根据Query\_ID,可以进一步执行show profile cpu,block io for query Query\_id来查看sql的具体执行步骤。



### 2.3 大致的查询流程

mysql的查询流程大致是：

mysql客户端通过协议与mysql服务器建连接，发送查询语句，先检查查询缓存，如果命中，直接返回结果，否则进行语句解析,也就是说，在解析查询之前，服务器会先访问查询缓存(query cache)——它存储SELECT语句以及相应的查询结果集。如果某个查询结果已经位于缓存中，服务器就不会再对查询进行解析、优化、以及执行。它仅仅将缓存中的结果返回给用户即可，这将大大提高系统的性能。

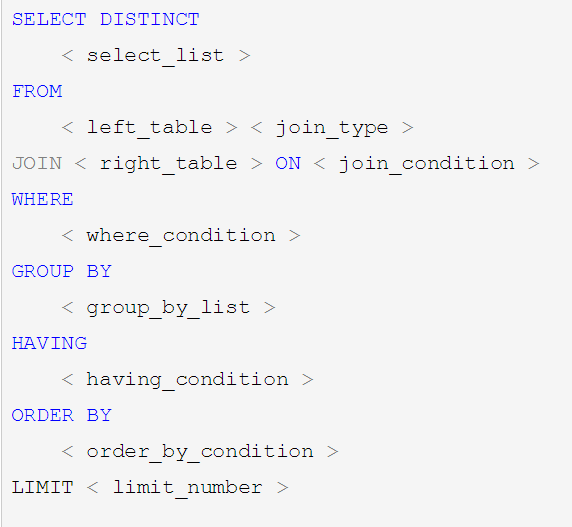
语法解析器和预处理：首先mysql通过关键字将SQL语句进行解析，并生成一颗对应的“解析树”。mysql解析器将使用mysql语法规则验证和解析查询；预处理器则根据一些mysql规则进一步检查解析数是否合法。

查询优化器当解析树被认为是合法的了，并且由优化器将其转化成执行计划。一条查询可以有很多种执行方式，最后都返回相同的结果。优化器的作用就是找到这其中最好的执行计划。。

然后，mysql默认使用的BTREE索引，并且一个大致方向是:无论怎么折腾sql，至少在目前来说，mysql最多只用到表中的一个索引。

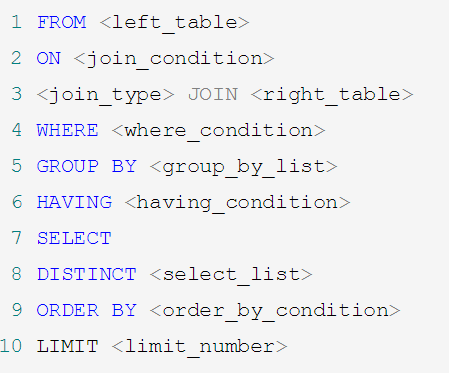
### 2.4 SQL的执行顺序

手写的顺序：

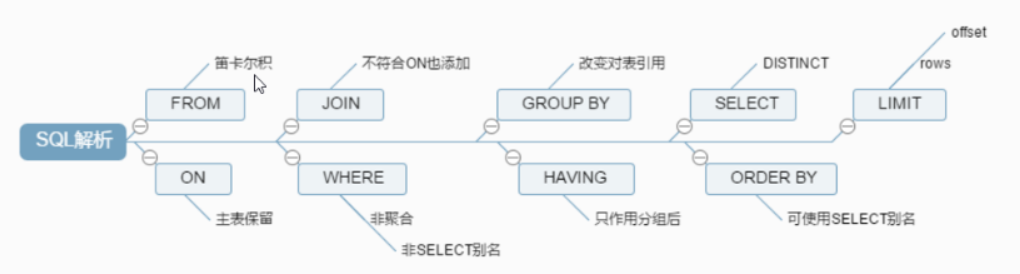


真正执行的顺序：

随着Mysql版本的更新换代，其优化器也在不断的升级，优化器会分析不同执行顺序产生的性能消耗不同而动态调整执行顺序。下面是经常出现的查询顺序：



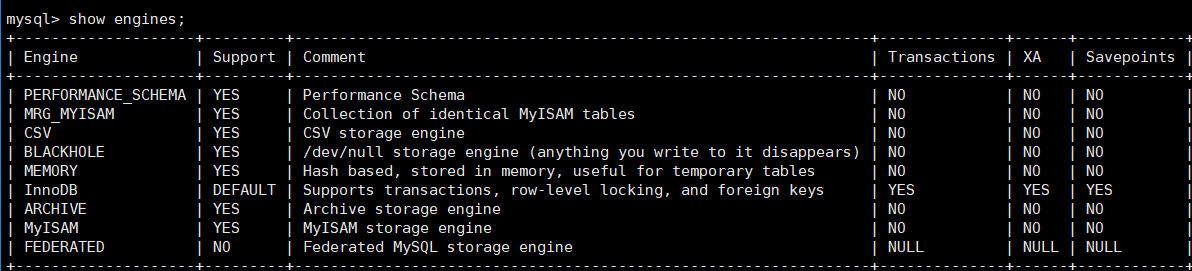
表的连接+where过滤+分组+having过滤+select+排序+限制



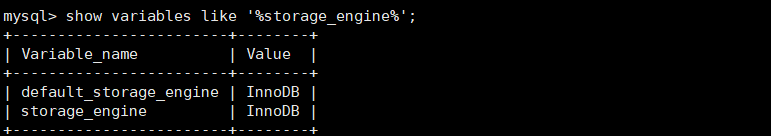
### 2.5 MyISAM和InnoDB

|  |  |  |
| --- | --- | --- |
| 对比项 | MyISAM | InnoDB |
| 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 |
| 关注点 | 读性能 | 并发写、事务、资源 |
| 默认安装 | Y | Y |
| 默认使用 | N | Y |
| 自带系统表使用 | Y | N |

show engines:查看所有的数据库引擎



show variables like '%storage\_engine%' 查看默认的数据库引擎



# 四、主从复制

**通过binlog实现主从复制**

不管使用何种存储引擎，在server层都可以开启binlog日志功能。binlog会记录所有的逻辑操作，并且是采取追加写的形式，将写操作命令，记录在一个二进制文件中。因此binlog日志通常用于恢复数据，或者是主从复制。

默认binlog文件会保存在/var/lib/mysql目录中，以mysql-bin.xxxx开头。可以使用工具mysqlbinlog进行查看。

mysql的主从复制为设置一台mysql服务实例为主机，其他一台为从机。主机的数据可以实时同步到从机。

步骤如下：

①确保主机和从机都开启了二进制日志

编辑 /etc/my.cnf文件，确保有如下配置：

|  |
| --- |
| log-bin=mysql-bin  server-id = x(x为一个int型的值，每个server实例需要有自己的id)  binlog\_format=mixed |

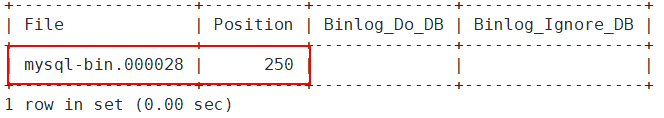
编辑完配置文件后，mysql服务需要重启。

②在主机上授权从机

使用root用户登录主机，在命令行执行如下命令。

|  |
| --- |
| GRANT replication slave ON \*.\* TO 'slave'@'%' IDENTIFIED BY '123456'; |

③在主机执行show master status查看主机的binlog日志信息



记录binlog的file和position属性。

④配置从机

使用mysql客户端连接从机服务，如果之前配置过主从，需要先执行stop slave停止主从关系。

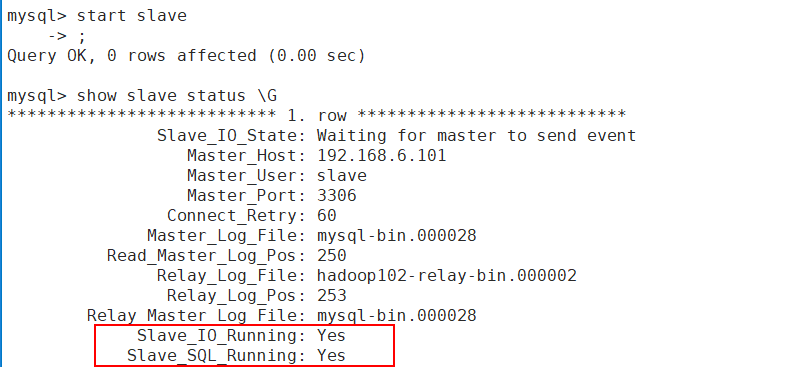
之后输入以下命令：

|  |
| --- |
| mysql> change master to  -> master\_user='slave',  -> master\_password='123456',  -> master\_host='192.168.6.101',  -> master\_log\_file='mysql-bin.000028',  -> master\_log\_pos=250; |

配置完成后，在从机的/var/lib/mysql目录下会产生一个名为master.info的主机配置信息文件。

⑤启动从机

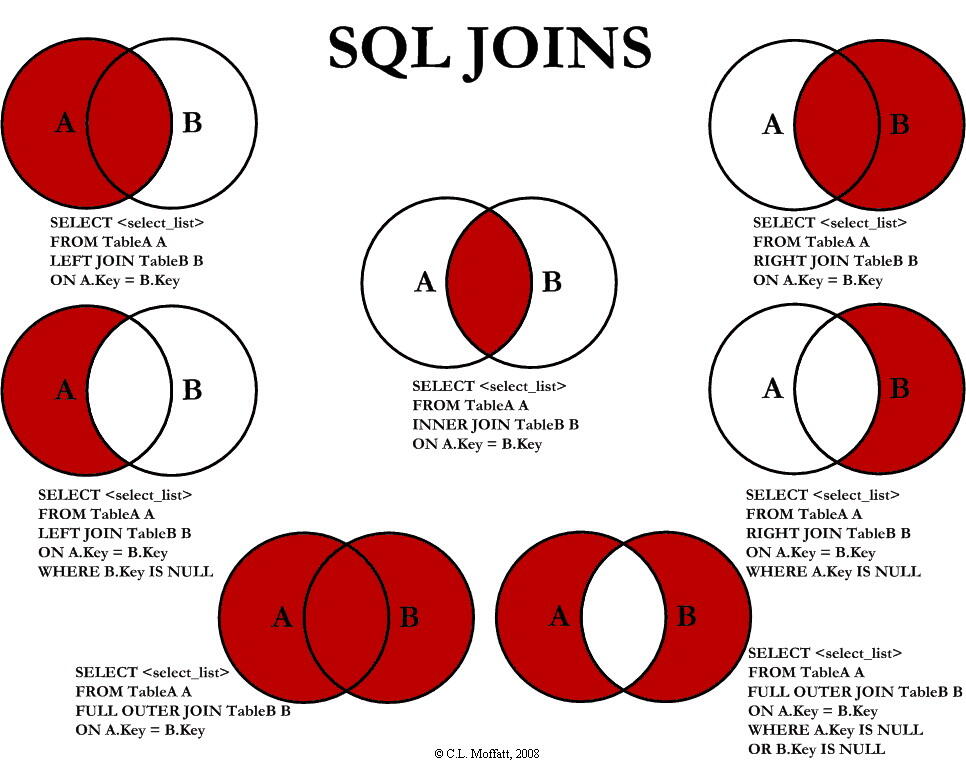
执行start slave命令，开启从机功能。



之后可以使用show slave status \G来查看从机是否正常启动。

# 五、SQL预热

## 1. 常见的Join查询图



Oracle数据库支持full join，mysql是不支持full join的，但仍然可以同过左外连接+ union+右外连接实现

UNION ALL只是简单的将两个结果合并后就返回。这样，如果返回的两个结果集中有重复的数据，那么返回的结果集就会包含重复的数据了。

从效率上说，UNION ALL 要比UNION快很多，所以，如果可以确认合并的两个结果集中不包含重复的数据的话，那么就使用UNION ALL，格式如下：select \* from A union all select \* from B //A B数据结构一定要一样

## 2. Join示例

### 2.1 建表语句

CREATE TABLE `t\_dept` (

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

`deptName` VARCHAR(30) DEFAULT NULL,

`address` VARCHAR(40) DEFAULT NULL,

PRIMARY KEY (`id`)

) ENGINE=INNODB AUTO\_INCREMENT=1 DEFAULT CHARSET=utf8;

CREATE TABLE `t\_emp` (

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

`name` VARCHAR(20) DEFAULT NULL,

`age` INT(3) DEFAULT NULL,

`deptId` INT(11) DEFAULT NULL,

empno int not null,

PRIMARY KEY (`id`),

KEY `idx\_dept\_id` (`deptId`)

#CONSTRAINT `fk\_dept\_id` FOREIGN KEY (`deptId`) REFERENCES `t\_dept` (`id`)

) ENGINE=INNODB AUTO\_INCREMENT=1 DEFAULT CHARSET=utf8;

INSERT INTO t\_dept(deptName,address) VALUES('华山','华山');

INSERT INTO t\_dept(deptName,address) VALUES('丐帮','洛阳');

INSERT INTO t\_dept(deptName,address) VALUES('峨眉','峨眉山');

INSERT INTO t\_dept(deptName,address) VALUES('武当','武当山');

INSERT INTO t\_dept(deptName,address) VALUES('明教','光明顶');

INSERT INTO t\_dept(deptName,address) VALUES('少林','少林寺');

INSERT INTO t\_emp(NAME,age,deptId,empno) VALUES('风清扬',90,1,100001);

INSERT INTO t\_emp(NAME,age,deptId,empno) VALUES('岳不群',50,1,100002);

INSERT INTO t\_emp(NAME,age,deptId,empno) VALUES('令狐冲',24,1,100003);

INSERT INTO t\_emp(NAME,age,deptId,empno) VALUES('洪七公',70,2,100004);

INSERT INTO t\_emp(NAME,age,deptId,empno) VALUES('乔峰',35,2,100005);

INSERT INTO t\_emp(NAME,age,deptId,empno) VALUES('灭绝师太',70,3,100006);

INSERT INTO t\_emp(NAME,age,deptId,empno) VALUES('周芷若',20,3,100007);

INSERT INTO t\_emp(NAME,age,deptId,empno) VALUES('张三丰',100,4,100008);

INSERT INTO t\_emp(NAME,age,deptId,empno) VALUES('张无忌',25,5,100009);

INSERT INTO t\_emp(NAME,age,deptId,empno) VALUES('韦小宝',18,null,100010);

1.所有有门派人员的信息（要求显示门派名称）

2. 列出所有人员及其门派信息

3. 列出所有门派

4. 所有无门派人士

5. 所有无人门派

6. 所有人员和门派的对应关系

7. 所有没有入门派的人员和没人入的门派

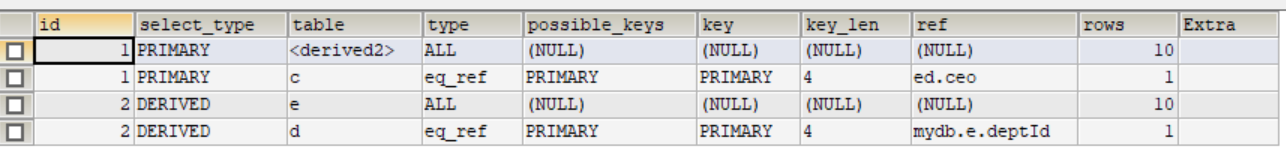
8.1求各个门派对应的掌门人名称

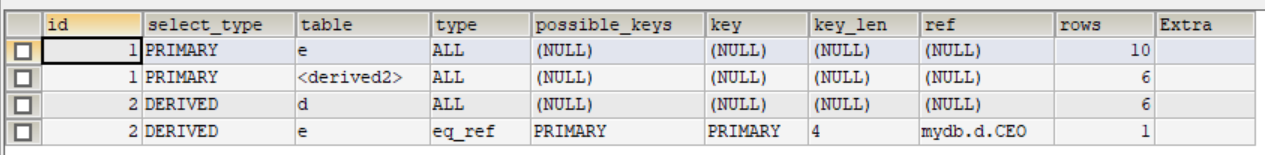
8.2求所有当上掌门人的平均年龄

8.3求所有人物对应的掌门名称

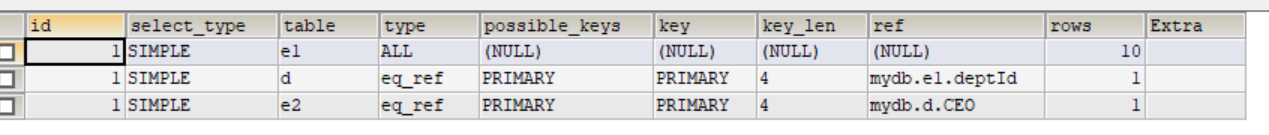
### 2.2 案例

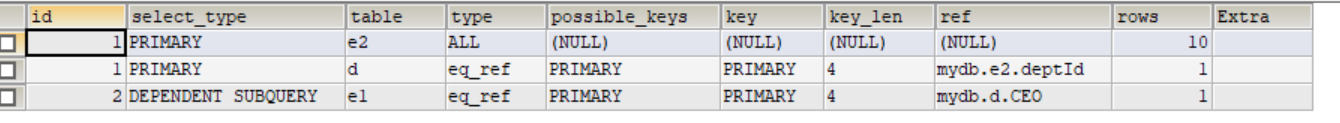
|  |
| --- |
| 1.所有有门派人员的信息（要求显示门派名称） |
| SELECT e.`name`,d.`deptName` FROM t\_emp e INNER JOIN t\_dept d ON e.`deptId`=d.`id`; |
| 2. 列出所有人员及其门派信息 |
| SELECT e.`name`,d.`deptName` FROM t\_emp e LEFT JOIN t\_dept d ON e.`deptId`=d.`id`; |
| 3. 列出所有门派 |
| SELECT \* FROM t\_dept; |
| 4. 所有无门派人士 |
| SELECT \* FROM t\_emp WHERE deptId IS NULL; |
| 5. 所有无人门派 |
| SELECT d.\* FROM t\_dept d LEFT JOIN t\_emp e ON d.`id`=e.`deptId` WHERE e.`deptId` IS NULL; |
| 6. 所有人员和门派的对应关系 |
| SELECT \* FROM t\_emp e LEFT JOIN t\_dept d ON e.`deptId`=d.`id`  UNION  SELECT \* FROM t\_emp e RIGHT JOIN t\_dept d ON e.`deptId`=d.`id`; |
| 7. 所有没有入门派的人员和没人入的门派 |
| SELECT \* FROM t\_emp e LEFT JOIN t\_dept d ON e.`deptId`=d.`id` WHERE e.deptId IS NULL  UNION  SELECT \* FROM t\_dept d LEFT JOIN t\_emp e ON d.`id`=e.`deptId` WHERE e.`deptId` IS NULL; |
| 8. 添加CEO字段  ALTER TABLE `t\_dept`  add CEO INT(11) ;  update t\_dept set CEO=2 where id=1;  update t\_dept set CEO=4 where id=2;  update t\_dept set CEO=6 where id=3;  update t\_dept set CEO=8 where id=4;  update t\_dept set CEO=9 where id=5; |
| 8.1 求各个门派对应的掌门人名称 |
| SELECT d.deptName,e.name FROM t\_dept d LEFT JOIN t\_emp e ON d.ceo=e.id |
| 8.2求所有当上掌门人的平均年龄 |
| SELECT AVG(e.age) FROM t\_dept d LEFT JOIN t\_emp e ON d.ceo=e.id |
| 8.3求所有人物对应的掌门名称 |
| SELECT ed.name '人物',c.name '掌门' FROM  (SELECT e.name,d.ceo from t\_emp e LEFT JOIN t\_dept d on e.deptid=d.id) ed  LEFT JOIN t\_emp c on ed.ceo= c.id; |
| SELECT e.name '人物',tmp.name '掌门'  FROM t\_emp e LEFT JOIN (SELECT d.id did,e.name FROM t\_dept d LEFT JOIN t\_emp e ON d.ceo=e.id)tmp  ON e.deptId=tmp.did; |
| SELECT e1.name '人物',e2.name '掌门'  FROM t\_emp e1  LEFT JOIN t\_dept d on e1.deptid = d.id  LEFT JOIN t\_emp e2 on d.ceo = e2.id ; |
| SELECT e2.name '人物',  (SELECT e1.name FROM t\_emp e1 where e1.id= d.ceo) '掌门'  from t\_emp e2 LEFT JOIN t\_dept d on e2.deptid=d.id; |





10 \*6\*6\*1 = 360





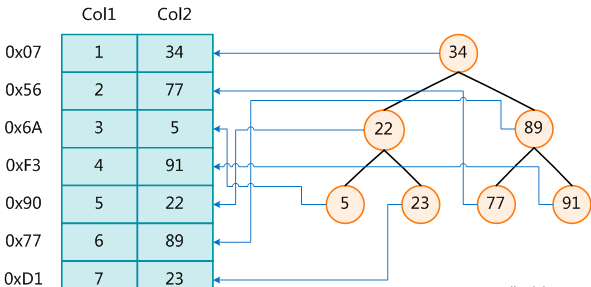
# 六、索引优化分析

## 1. 索引的概念

### 1.1 是什么

MySQL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。可以简单理解为排好序的快速查找数据结构。

在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。下图就是一种可能的索引方式示例：



左边是数据表，一共有两列七条记录，最左边的是数据记录的物理地址。为了加快Col2的查找，可以维护一个右边所示的二叉查找树，每个节点分别包含索引键值和一个指向对应数据记录物理地址的指 针，这样就可以运用二叉查找在一定的复杂度内获取到相应数据，从而快速的检索出符合条件的记录。

一般来说索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。

### 1.2 优缺点

**优势**：

* 提高数据检索的效率，降低数据库的IO成本。
* 通过索引列对数据进行排序，降低数据排序的成本，降低了CPU的消耗。

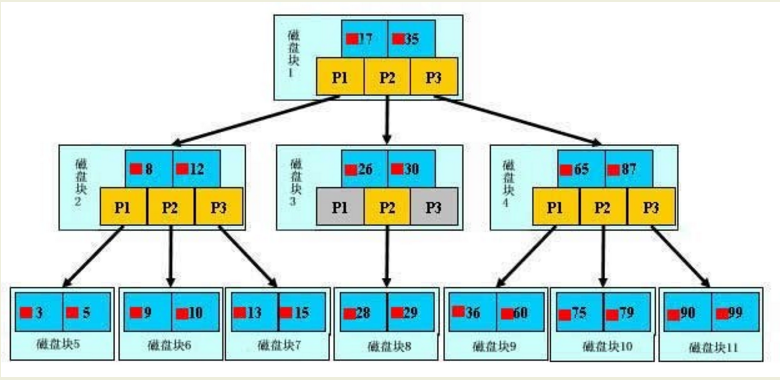
**劣势**：

* 虽然索引大大提高了查询速度，同时却会降低更新表的速度，如对表进行INSERT、UPDATE和DELETE。因为更新表时，MySQL不仅要保存数据，还要保存一下索引文件每次更新添加了索引列的字段，都会调整因为更新所带来的键值变化后的索引信息。
* 实际上索引也是一张表，该表保存了主键与索引字段，并指向实体表的记录，所以索引列也是要占用空间的。

## 2. Mysql的索引

### 2.1 Btree索引

MySQL使用的是Btree索引。



【初始化介绍】

一颗b树，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。

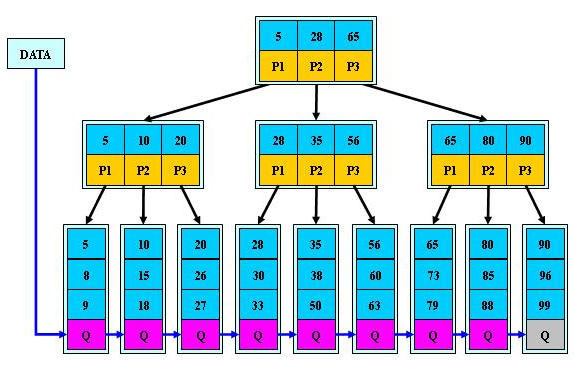
非叶子节点不存储真实的数据，只存储指引搜索方向的数据项，如17、35并不真实存在于数据表中。

【查找过程】

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。

### 2.2 B+tree索引



B+Tree与B-Tree 的区别

　1）B-树的关键字和记录是放在一起的，叶子节点可以看作外部节点，不包含任何信息；B+树的非叶子节点中只有关键字和指向下一个节点的索引，记录只放在叶子节点中。

　 2）在B-树中，越靠近根节点的记录查找时间越快，只要找到关键字即可确定记录的存在；而B+树中每个记录的查找时间基本是一样的，都需要从根节点走到叶子节点，而且在叶子节点中还要再比较关键字。从这个角度看B-树的性能好像要比B+树好，而在实际应用中却是B+树的性能要好些。因为B+树的非叶子节点不存放实际的数据，这样每个节点可容纳的元素个数比B-树多，树高比B-树小，这样带来的好处是减少磁盘访问次数。尽管B+树找到一个记录所需的比较次数要比B-树多，但是一次磁盘访问的时间相当于成百上千次内存比较的时间，因此实际中B+树的性能可能还会好些，而且B+树的叶子节点使用指针连接在一起，方便顺序遍历（例如查看一个目录下的所有文件，一个表中的所有记录等），这也是很多数据库和文件系统使用B+树的缘故。

思考：为什么说B+树比B-树更适合实际应用中操作系统的文件索引和数据库索引？

1) B+树的磁盘读写代价更低

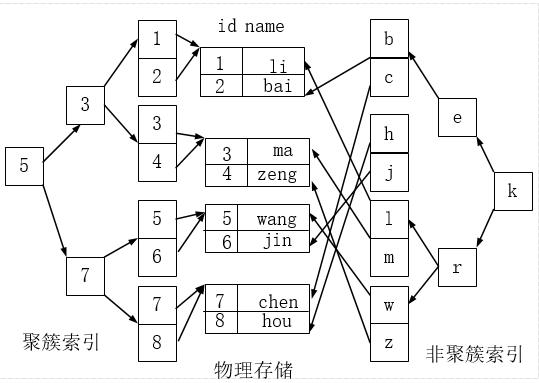
　　B+树的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说IO读写次数也就降低了。

2) B+树的查询效率更加稳定

由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

### 2.3 聚簇索引和非聚簇索引

聚簇索引并不是一种单独的索引类型，而是一种数据存储方式。术语‘聚簇’表示数据行和相邻的键值聚簇的存储在一起。如下图，左侧的索引就是聚簇索引，因为数据行在磁盘的排列和索引排序保持一致。



聚簇索引的好处：【聚集索引与非聚集索引】

按照聚簇索引排列顺序，查询显示一定范围数据的时候，由于数据都是紧密相连，数据库不不用从多个数据块中提取数据，所以节省了大量的io操作。

聚簇索引的限制：

对于mysql数据库目前只有innodb数据引擎支持聚簇索引，而Myisam并不支持聚簇索引。

由于数据物理存储排序方式只能有一种，所以每个Mysql的表只能有一个聚簇索引。一般情况下就是该表的主键。

为了充分利用聚簇索引的聚簇的特性，所以innodb表的主键列尽量选用有序的顺序id，而不建议用无序的id，比如uuid这种。

## 3. Mysql索引分类

### 3.1 单值索引

概念：即一个索引只包含单个列，一个表可以有多个单列索引

语法：

|  |
| --- |
| 随表一起创建： |
| CREATE TABLE customer (id INT(10) UNSIGNED AUTO\_INCREMENT ,customer\_no VARCHAR(200),customer\_name VARCHAR(200),  PRIMARY KEY(id),  KEY (customer\_name)  ); |
| 单独建单值索引： |
| CREATE INDEX idx\_customer\_name ON customer(customer\_name); |

### 3.2 唯一索引

概念：索引列的值必须唯一，但允许有空值

|  |
| --- |
| 随表一起创建： |
| CREATE TABLE customer (id INT(10) UNSIGNED AUTO\_INCREMENT ,customer\_no VARCHAR(200),customer\_name VARCHAR(200),  PRIMARY KEY(id),  KEY (customer\_name),  UNIQUE (customer\_no)  ); |
| 单独建唯一索引： |
| CREATE UNIQUE INDEX idx\_customer\_no ON customer(customer\_no); |

### 3.3 主键索引

概念：设定为主键后数据库会自动建立索引，innodb为聚簇索引

|  |
| --- |
| 随表一起建索引 |
| CREATE TABLE customer (id INT(10) UNSIGNED AUTO\_INCREMENT ,customer\_no VARCHAR(200),customer\_name VARCHAR(200),  PRIMARY KEY(id)  ); |
| 单独建主键索引： |
| ALTER TABLE customer add PRIMARY KEY customer(customer\_no); |
| 删除建主键索引： |
| ALTER TABLE customer drop PRIMARY KEY ; |
| 修改建主键索引： |
| 必须先删除掉(drop)原索引，再新建(add)索引 |

### 3.4 复合索引

概念：即一个索引包含多个列

|  |
| --- |
| 随表一起建索引： |
| CREATE TABLE customer (id INT(10) UNSIGNED AUTO\_INCREMENT ,customer\_no VARCHAR(200),customer\_name VARCHAR(200),  PRIMARY KEY(id),  KEY (customer\_name),  UNIQUE (customer\_name),  KEY (customer\_no,customer\_name)  ); |
| 单独建索引： |
| CREATE INDEX idx\_no\_name ON customer(customer\_no,customer\_name); |

### 3.5 基本语法

|  |  |
| --- | --- |
| 操作 | 命令 |
| 创建 | CREATE [UNIQUE ] INDEX [indexName] ON table\_name(column)) |
| 删除 | DROP INDEX [indexName] ON mytable; |
| 查看 | SHOW INDEX FROM table\_name\G |
| 使用Alter命令 | ALTER TABLE tbl\_name ADD PRIMARY KEY (column\_list) : 该语句添加一个主键，这意味着索引值必须是唯一的，且不能为NULL。 |
| ALTER TABLE tbl\_name ADD PRIMARY KEY (column\_list) |
| ALTER TABLE tbl\_name ADD INDEX index\_name (column\_list): 添加普通索引，索引值可出现多次。 |
| ALTER TABLE tbl\_name ADD FULLTEXT index\_name (column\_list):该语句指定了索引为 FULLTEXT ，用于全文索引。 |

## 4. 索引的创建时机

### 4.1 适合创建索引的情况

* 主键自动建立唯一索引；
* 频繁作为查询条件的字段应该创建索引
* 查询中与其它表关联的字段，外键关系建立索引
* 单键/组合索引的选择问题， 组合索引性价比更高
* 查询中排序的字段，排序字段若通过索引去访问将大大提高排序速度
* 查询中统计或者分组字段

### 4.2 不适合创建索引的情况

* 表记录太少
* 经常增删改的表或者字段
* Where条件里用不到的字段不创建索引
* 过滤性不好的不适合建索引

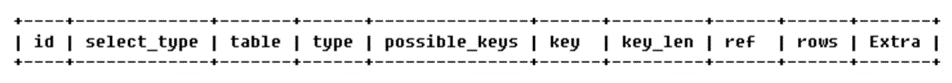
# 七、Explain性能分析

## 1. 概念

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。

用法： Explain+SQL语句。

Explain执行后返回的信息：



主要参数：【type】【rows】【Extra】

## 2. Explain准备工作

CREATE TABLE t1(id INT(10) AUTO\_INCREMENT,content VARCHAR(100) NULL , PRIMARY KEY (id));

CREATE TABLE t2(id INT(10) AUTO\_INCREMENT,content VARCHAR(100) NULL , PRIMARY KEY (id));

CREATE TABLE t3(id INT(10) AUTO\_INCREMENT,content VARCHAR(100) NULL , PRIMARY KEY (id));

CREATE TABLE t4(id INT(10) AUTO\_INCREMENT,content VARCHAR(100) NULL , PRIMARY KEY (id));

INSERT INTO t1(content) VALUES(CONCAT('t1\_',FLOOR(1+RAND()\*1000)));

INSERT INTO t2(content) VALUES(CONCAT('t2\_',FLOOR(1+RAND()\*1000)));

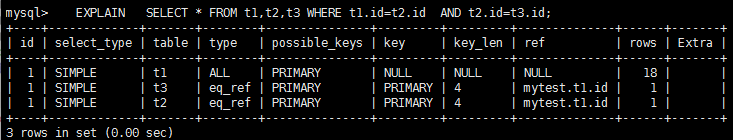
INSERT INTO t3(content) VALUES(CONCAT('t3\_',FLOOR(1+RAND()\*1000)));

INSERT INTO t4(content) VALUES(CONCAT('t4\_',FLOOR(1+RAND()\*1000)));

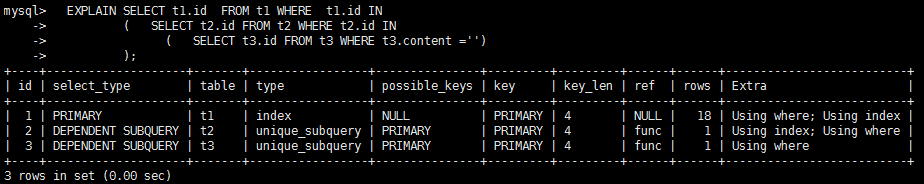
## 3. id

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。

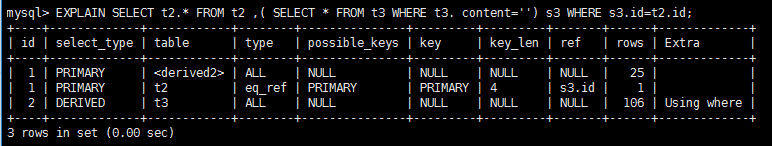
①id相同，执行顺序由上至下



②id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



③有相同也有不同

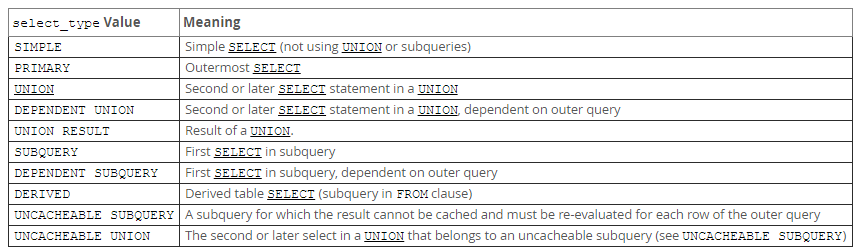


id如果相同，可以认为是一组，从上往下顺序执行；在所有组中，id值越大，优先级越高，越先执行衍生 = DERIVED

关注点：id号每个号码，表示一趟独立的查询。一个sql 的查询趟数越少越好。

## 4. select\_type

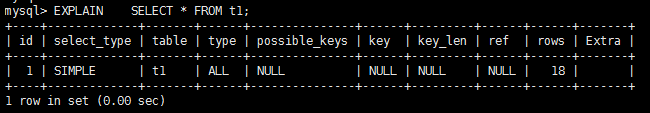
select\_type代表查询的类型，主要是用于区别普通查询、联合查询、子查询等的复杂查询。



|  |  |
| --- | --- |
| select\_type属性 | 含义 |
| 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 |
| DERIVED | 在FROM列表中包含的子查询被标记为DERIVED(衍生)  MySQL会递归执行这些子查询, 把结果放在临时表里。 |
| SUBQUERY | 在SELECT或WHERE列表中包含了子查询 |
| DEPEDENT SUBQUERY | 在SELECT或WHERE列表中包含了子查询,子查询基于外层 |
| UNCACHEABLE SUBQUERY | 无法使用缓存的子查询 |
| 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 |

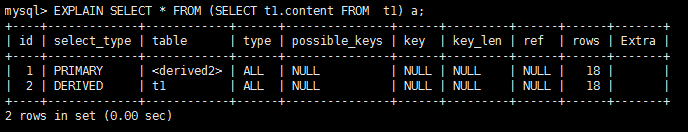
### 4.1 SIMPLE

SIMPLE代表单表查询；



### 4.2 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。

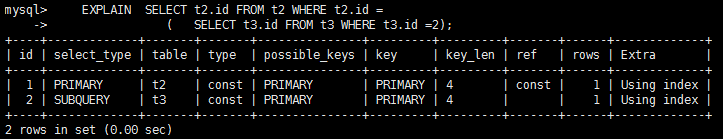


### 4.3 DERIVED

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

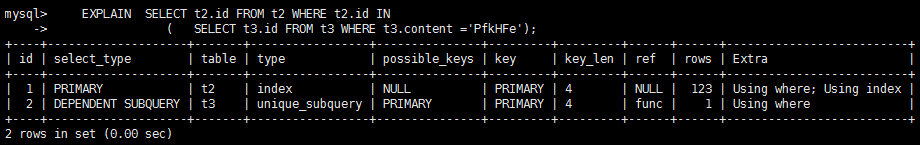
### 4.4 SUBQUERY

在SELECT或WHERE列表中包含了子查询。



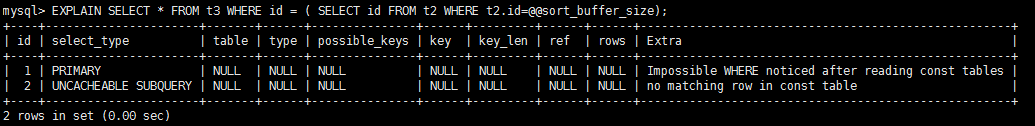
### 4.5 DEPENDENT SUBQUERY

在SELECT或WHERE列表中包含了子查询,子查询基于外层。



都是where后面的条件，subquery是单个值，dependent subquery是一组值。

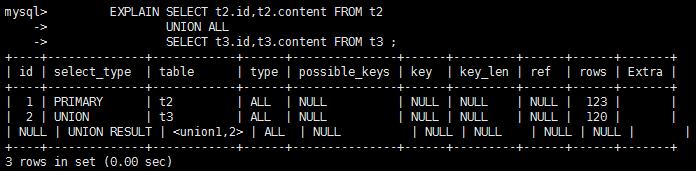
### 4.6 UNCACHEABLE SUBQUREY



当使用了@@来引用系统变量【系统变量表获取值】的时候，不会使用缓存。

### 4.7 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。



### 4.8 UNION RESULT

从UNION表获取结果的SELECT。

## 5. table

这个数据是基于哪张表的。

## 6. type

type是查询的访问类型。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 ，一般来说，得保证查询至少达到range级别，最好能达到ref。【all:没有使用到索引，可以考虑添加】

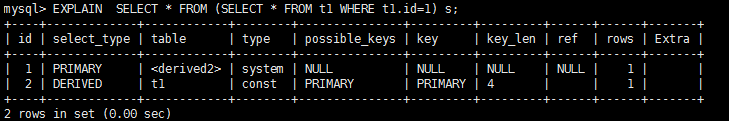
### 6.1 system

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

### 6.2 const

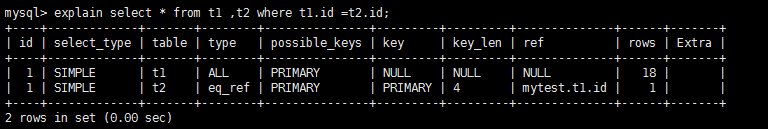
表示通过索引一次就找到了,const用于比较primary key或者unique索引。因为只匹配一行数据，所以很快

如将主键置于where列表中，MySQL就能将该查询转换为一个常量。【需要添加索引】



### 6.3 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。

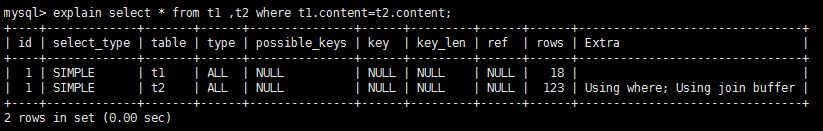


左表全表扫描，右表找匹配的字段，可能是一对多，所以是eq\_ref。

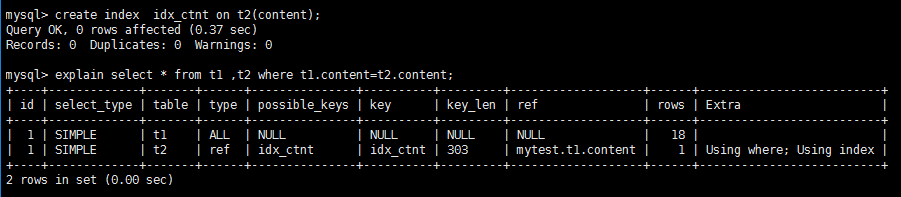
### 6.4 ref

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。

没用索引前：

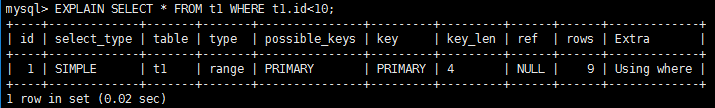


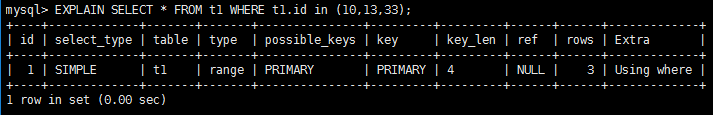
建立索引后：



### 6.5 range

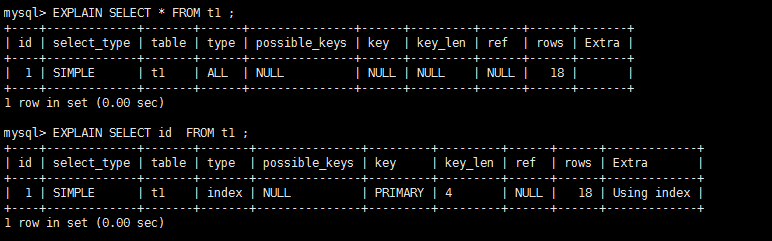
只检索给定范围的行,使用一个索引来选择行。key 列显示使用了哪个索引一般就是在你的where语句中出现了between、<、>、in等的查询这种范围扫描索引扫描比全表扫描要好，因为它只需要开始于索引的某一点，而结束语另一点，不用扫描全部索引。





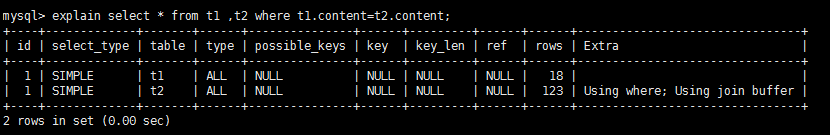
### 6.6 index

出现index是sql使用了索引但是没用通过索引进行过滤，一般是使用了覆盖索引【没有开表直接缓存中找】或者是利用索引进行了排序分组。



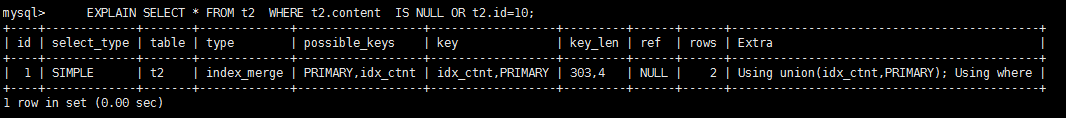
### 6.7 all

Full Table Scan，将遍历全表以找匹配的行。



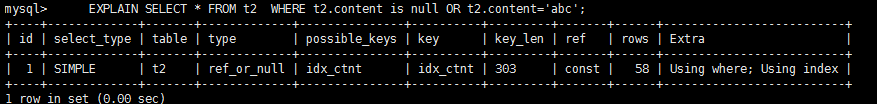
### 6.8 index\_merge

在查询过程中需要多个索引组合使用，通常出现在有 or 的关键字的sql中。



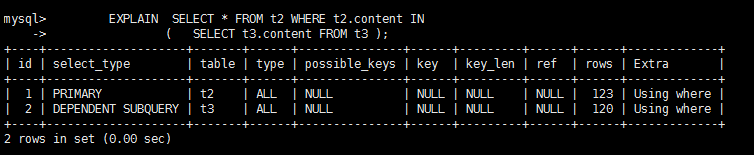
### 6.9 ref\_or\_null

对于某个字段既需要关联条件，也需要null值得情况下。查询优化器会选择用ref\_or\_null连接查询。

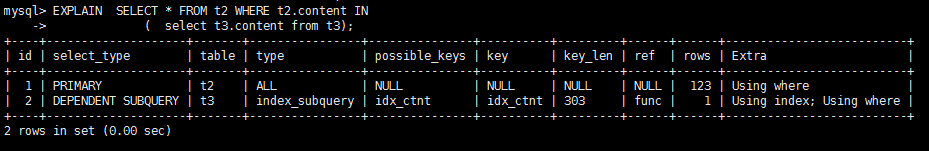


### 6.10 index\_subquery

利用索引来关联子查询，不再全表扫描。



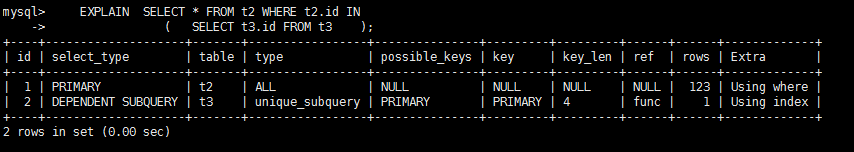
graphic



子查询索引使用到了

### 6.11 unique\_subquery

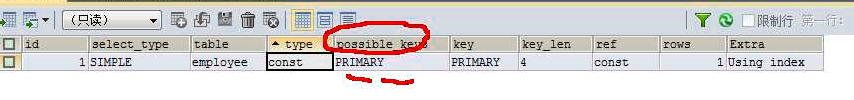
该联接类型类似于index\_subquery。 子查询中的唯一索引。【子查询索引都用上了，推荐】



备注：一般来说，得保证查询至少达到range级别，最好能达到ref。

## 7. possible\_keys

显示可能应用在这张表中的索引，一个或多个。查询涉及到的字段上若存在索引，则该索引将被列出，**但不一定被查询实际使用。**

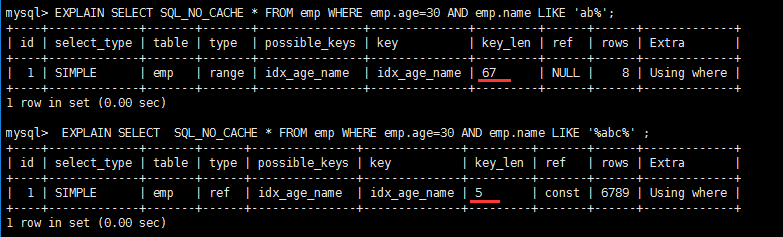


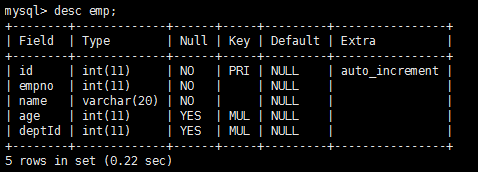
## 8. key

实际使用的索引。如果为NULL，则没有使用索引。

## 9. 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。 key\_len字段能够帮你检查是否充分的利用上了索引。ken\_len越长，说明索引使用的越充分。【where后面】





**如何计算：**

①先看索引上字段的类型+长度比如 int=4 ; varchar(20) =20 ; char(20) =20

②如果是varchar或者char这种字符串字段，视字符集要乘不同的值，比如utf-8 要乘 3,GBK要乘2，

③varchar这种动态字符串要加2个字节

④允许为空的字段要加1个字节

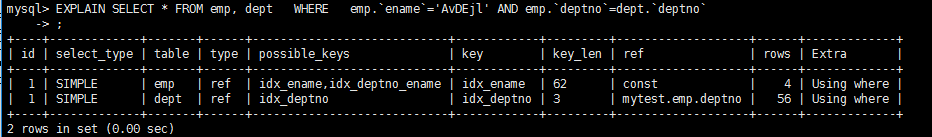
第一组：key\_len=age的字节长度+name的字节长度=4+1 + ( 20\*3+2)=5+62=67

第二组：key\_len=age的字节长度=4+1=5



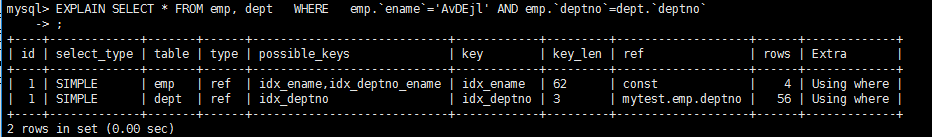
## 10. ref

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



## 11. rows

rows列显示MySQL认为它执行查询时必须检查的行数。越少越好！



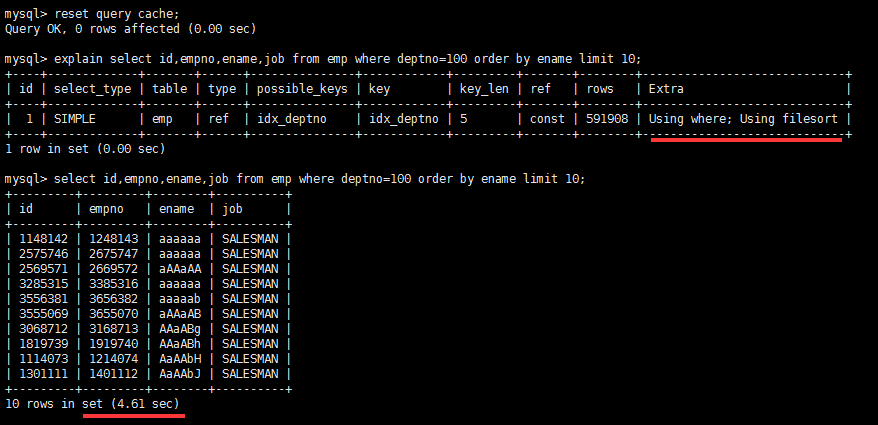
## 12. Extra

其他的额外重要的信息。

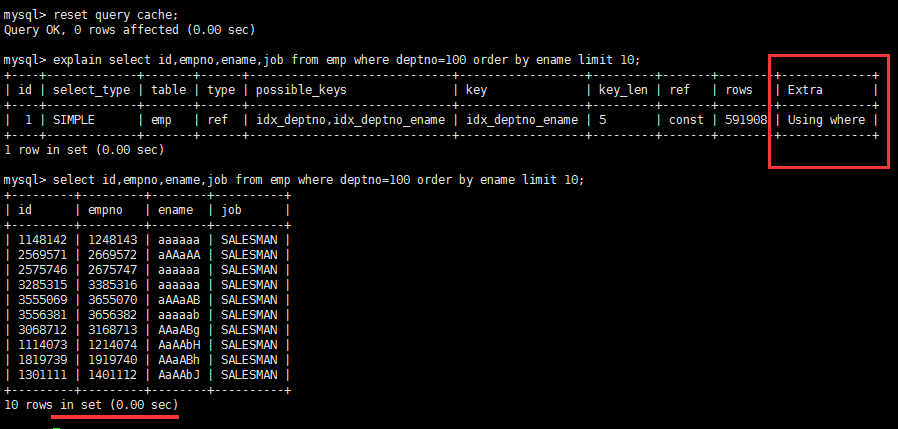
### 12.1 Using filesort

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”。【没有按照指定的索引操作，需要优化】

出现filesort的情况：



优化后，不再出现filesort的情况：

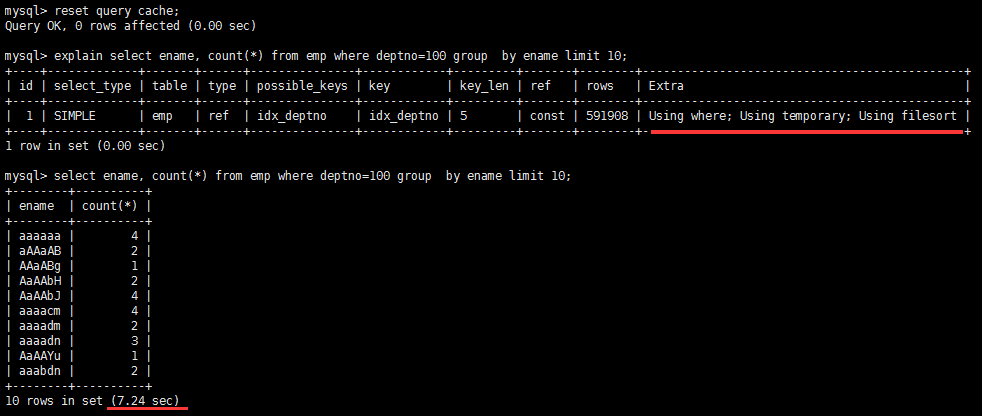


**查询中排序的字段，排序字段若通过索引去访问将大大提高排序速度。**

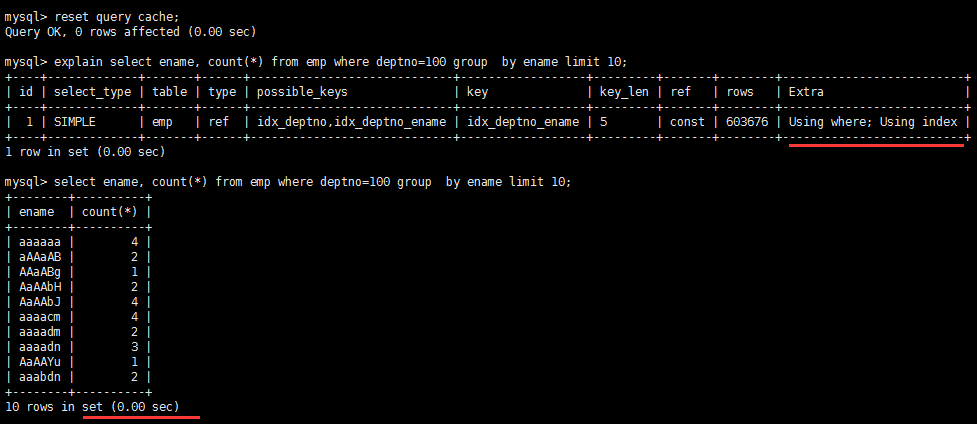
### 12.2 Using temporary

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。【出现了表明没有使用到索引】

优化前：



优化后：



### 12.3 Using index

Using index代表表示相应的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！

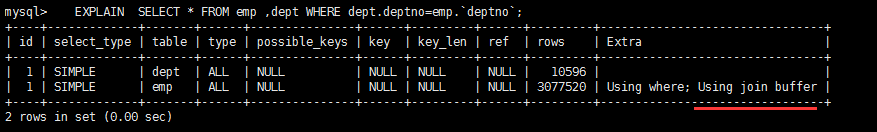
如果同时出现using where，表明索引被用来执行索引键值的查找;如果没有同时出现using where，表明索引只是用来读取数据而非利用索引执行查找。

利用索引进行了排序或分组。

### 12.4 Using where

表明使用了where过滤。

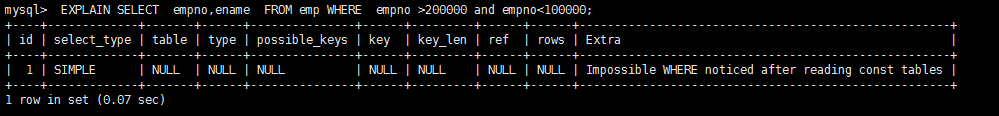
### 12.5 Using join buffer



使用了连接缓存。

### 12.6 impossible where

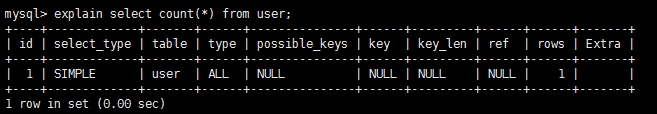
where子句的值总是false，不能用来获取任何元组。



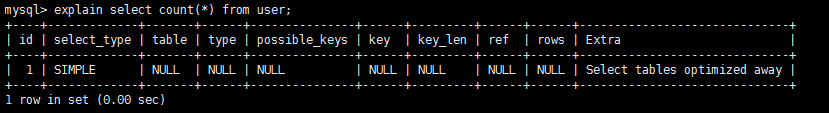
### 12.7 select tables optimized away

在没有GROUPBY子句的情况下，基于索引优化MIN/MAX操作或者对于MyISAM存储引擎优化COUNT(\*)操作，不必等到执行阶段再进行计算，查询执行计划生成的阶段即完成优化。

在innodb中：



在Myisam中：



# 八、批量数据脚本

## 1. 插入数据

### 1.1 建表语句

CREATE TABLE `dept` (

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

`deptName` VARCHAR(30) DEFAULT NULL,

`address` VARCHAR(40) DEFAULT NULL,

ceo INT NULL ,

PRIMARY KEY (`id`)

) ENGINE=INNODB AUTO\_INCREMENT=1 DEFAULT CHARSET=utf8;

CREATE TABLE `emp` (

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

`empno` INT NOT NULL ,

`name` VARCHAR(20) DEFAULT NULL,

`age` INT(3) DEFAULT NULL,

`deptId` INT(11) DEFAULT NULL,

PRIMARY KEY (`id`)

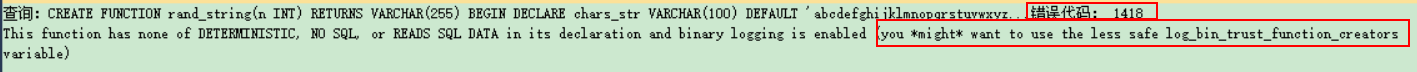
#CONSTRAINT `fk\_dept\_id` FOREIGN KEY (`deptId`) REFERENCES `t\_dept` (`id`)

) ENGINE=INNODB AUTO\_INCREMENT=1 DEFAULT CHARSET=utf8;

### 1.2 设置参数

在执行创建函数之前，首先请保证log\_bin\_trust\_function\_creators参数为1，即on开启状态。

否则会报错：



查询：show variables like 'log\_bin\_trust\_function\_creators';

设置：set global log\_bin\_trust\_function\_creators=1;

当然，如上设置只存在于当前操作，想要永久生效，需要写入到配置文件中：

在[mysqld]中加上log\_bin\_trust\_function\_creators=1

### 1.3 编写随机函数

创建函数，保证每条数据都不同。

#### 1.3.1 随机产生字符串

DELIMITER $$

CREATE FUNCTION rand\_string(n INT) RETURNS VARCHAR(255)

BEGIN

DECLARE chars\_str VARCHAR(100) DEFAULT 'abcdefghijklmnopqrstuvwxyzABCDEFJHIJKLMNOPQRSTUVWXYZ';

DECLARE return\_str VARCHAR(255) DEFAULT '';

DECLARE i INT DEFAULT 0;

WHILE i < n DO

SET return\_str =CONCAT(return\_str,SUBSTRING(chars\_str,FLOOR(1+RAND()\*52),1));

SET i = i + 1;

END WHILE;

RETURN return\_str;

END $$

如果要删除函数，则执行：drop function rand\_string;

#### 1.3.2 随机产生部门编号

#用于随机产生多少到多少的编号

DELIMITER $$

CREATE FUNCTION rand\_num (from\_num INT ,to\_num INT) RETURNS INT(11)

BEGIN

DECLARE i INT DEFAULT 0;

SET i = FLOOR(from\_num +RAND()\*(to\_num -from\_num+1)) ;

RETURN i;

END$$

如果要删除函数：drop function rand\_num;

### 1.4 创建存储过程

#### 1.4.1 创建往emp表中插入数据的存储过程

DELIMITER $$

CREATE PROCEDURE insert\_emp( START INT , max\_num INT )

BEGIN

DECLARE i INT DEFAULT 0;

#set autocommit =0 把autocommit设置成0

SET autocommit = 0;

REPEAT

SET i = i + 1;

INSERT INTO emp (empno, NAME ,age ,deptid ) VALUES ((START+i) ,rand\_string(6) , rand\_num(30,50),rand\_num(1,10000));

UNTIL i = max\_num

END REPEAT;

COMMIT;

END$$

#删除

# DELIMITER ;

# drop PROCEDURE insert\_emp;

#### 1.4.2 创建往dept表中插入数据的存储过程

#执行存储过程，往dept表添加随机数据

DELIMITER $$

CREATE PROCEDURE `insert\_dept`( max\_num INT )

BEGIN

DECLARE i INT DEFAULT 0;

SET autocommit = 0;

REPEAT

SET i = i + 1;

INSERT INTO dept ( deptname,address,ceo ) VALUES (rand\_string(8),rand\_string(10),rand\_num(1,500000));

UNTIL i = max\_num

END REPEAT;

COMMIT;

END$$

#删除

# DELIMITER ;

# drop PROCEDURE insert\_dept;

### 1.5 调用存储过程

#### 1.5.1 添加数据到部门表

#执行存储过程，往dept表添加1万条数据

DELIMITER ;

CALL insert\_dept(10000);

#### 1.5.2 添加数据到员工表

#执行存储过程，往emp表添加50万条数据

DELIMITER ;

CALL insert\_emp(100000,500000);

### 1.6 批量删除某个表上的所有索引

#### 1.6.1 删除索引的存储过程

DELIMITER $$

CREATE PROCEDURE `proc\_drop\_index`(dbname VARCHAR(200),tablename VARCHAR(200))

BEGIN

DECLARE done INT DEFAULT 0;

DECLARE ct INT DEFAULT 0;

DECLARE \_index VARCHAR(200) DEFAULT '';

DECLARE \_cur CURSOR FOR SELECT index\_name FROM information\_schema.STATISTICS WHERE table\_schema=dbname AND table\_name=tablename AND seq\_in\_index=1 AND index\_name <>'PRIMARY' ;

DECLARE CONTINUE HANDLER FOR NOT FOUND set done=2 ;

OPEN \_cur;

FETCH \_cur INTO \_index;

WHILE \_index<>'' DO

SET @str = CONCAT("drop index ",\_index," on ",tablename );

PREPARE sql\_str FROM @str ;

EXECUTE sql\_str;

DEALLOCATE PREPARE sql\_str;

SET \_index='';

FETCH \_cur INTO \_index;

END WHILE;

CLOSE \_cur;

END$$

#### 1.6.2 执行存储过程

调用：CALL proc\_drop\_index("dbname","tablename");

# 九、单表使用索引常见的索引失效

## 1. 全值匹配好

### 1.1 有以下SQL语句

EXPLAIN SELECT SQL\_NO\_CACHE \* FROM emp WHERE emp.age=30

EXPLAIN SELECT SQL\_NO\_CACHE \* FROM emp WHERE emp.age=30 and deptid=4

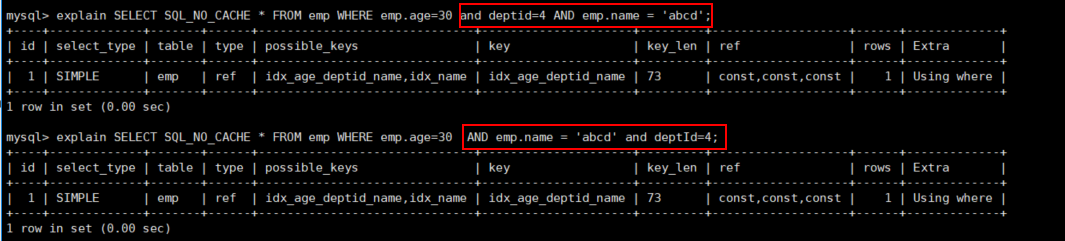
EXPLAIN SELECT SQL\_NO\_CACHE \* FROM emp WHERE emp.age=30 and deptid=4 AND emp.name = 'abcd'

### 1.2 建立索引

CREATE INDEX idx\_age\_deptid\_name ON emp(age,deptid,NAME);

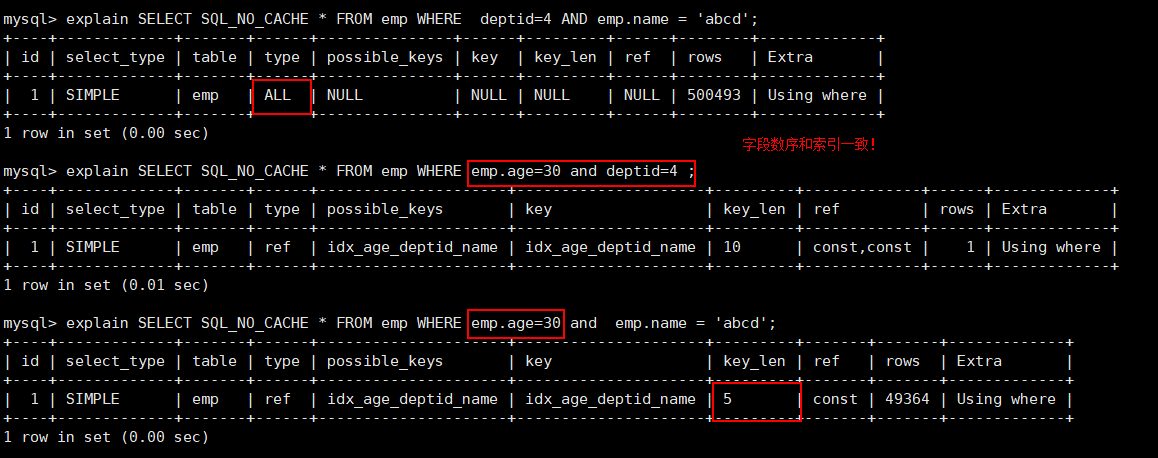


结论：全职匹配我最爱指的是，查询的字段按照顺序在索引中都可以匹配到！



SQL中查询字段的顺序，跟使用索引中字段的顺序，没有关系。优化器会在不影响SQL执行结果的前提下，给你自动地优化。

## 2. 最佳左前缀法则



查询字段与索引字段顺序的不同会导致，索引无法充分使用，甚至索引失效！

原因：使用复合索引，需要遵循最佳左前缀法则，即如果索引了多列，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。

**结论：过滤条件要使用索引必须按照索引建立时的顺序，依次满足，一旦跳过某个字段，索引后面的字段都无法被使用。【最佳左前缀法则】**

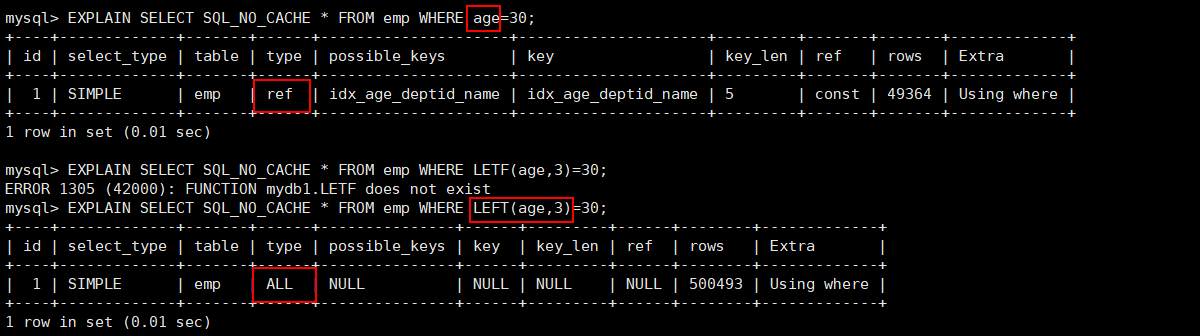
## 3. 不要在索引列上做任何计算

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。

### 3.1 在查询列上使用了函数

EXPLAIN SELECT SQL\_NO\_CACHE \* FROM emp WHERE age=30;

EXPLAIN SELECT SQL\_NO\_CACHE \* FROM emp WHERE LEFT(age,3)=30; // left左边取3位结果是30

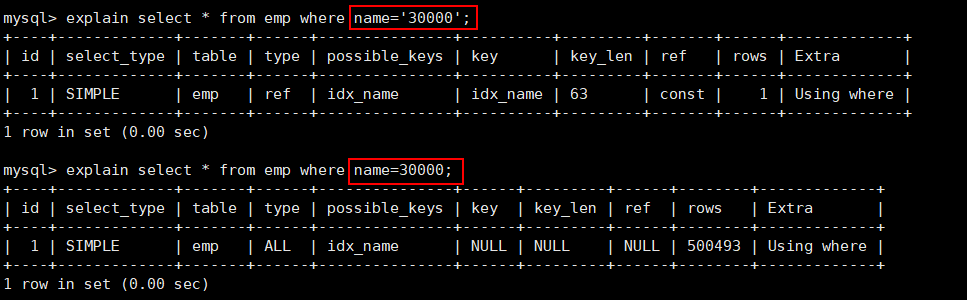


结论：等号左边无计算！不要在索引列做任何计算

### 3.2 在查询列上做了转换

|  |
| --- |
| create index idx\_name on emp(name); |
| explain select sql\_no\_cache \* from emp where name='30000'; |
| explain select sql\_no\_cache \* from emp where name=30000; |

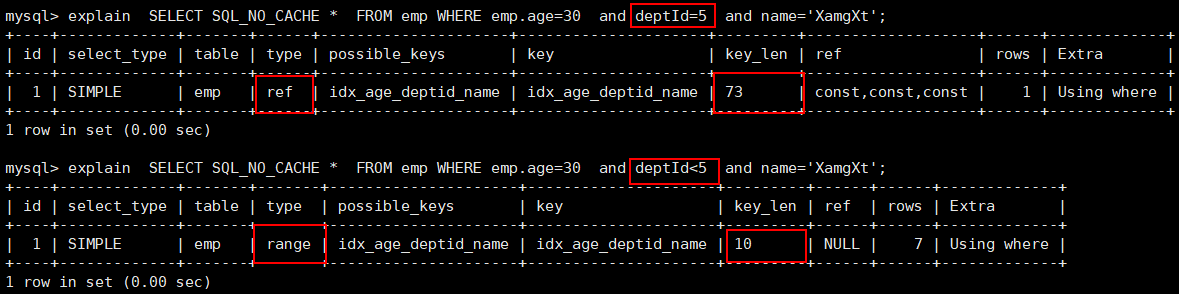
字符串不加单引号，则会在name列上做一次转换！



结论：等号右边无转换！

## 4. 索引列上不能有范围查询

|  |
| --- |
| explain SELECT SQL\_NO\_CACHE \* FROM emp WHERE emp.age=30 and deptid=5 AND emp.name = 'abcd'; |
| explain SELECT SQL\_NO\_CACHE \* FROM emp WHERE emp.age=30 and deptid<5 AND emp.name = 'abcd'; |

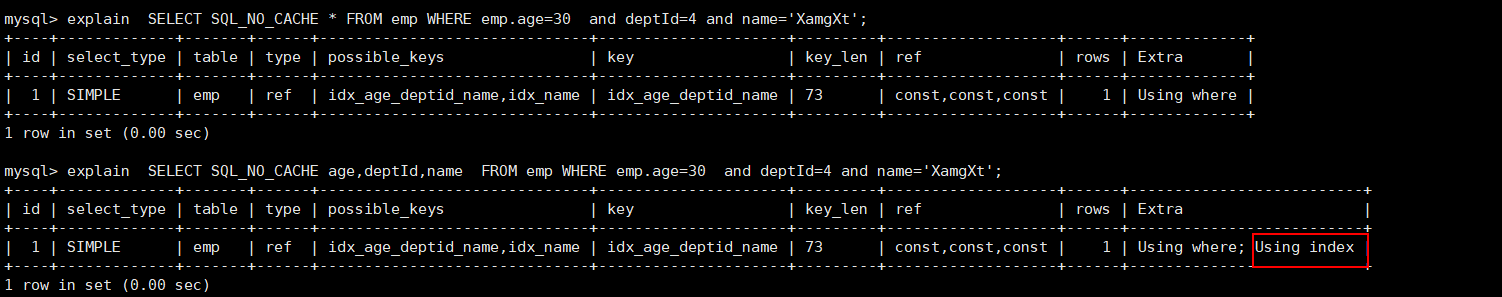


建议：将可能做范围查询的字段的索引顺序放在最后【where中范围后面的字段的索引会失效】

## 5. 尽量使用覆盖索引

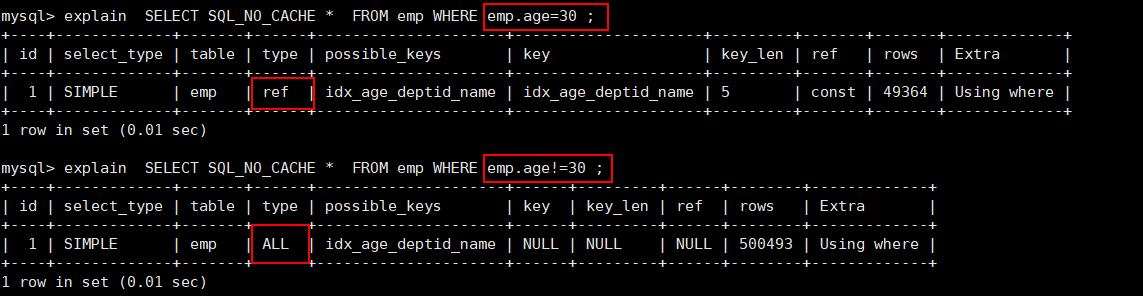
即查询列和索引列一直，不要写select \*!【尽量不要使用\* 分组过滤的时候索引会失效】

|  |
| --- |
| explain SELECT SQL\_NO\_CACHE \* FROM emp WHERE emp.age=30 and deptId=4 and name='XamgXt'; |
| explain SELECT SQL\_NO\_CACHE age,deptId,name FROM emp WHERE emp.age=30 and deptId=4 and name='XamgXt'; |

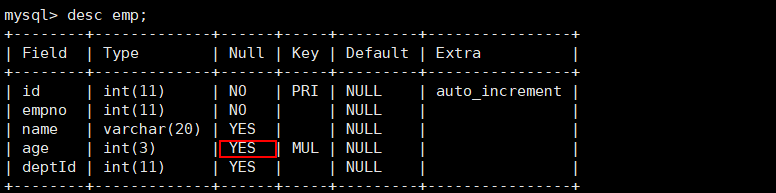


## 6. 使用不等于(!= 或者<>)的时候

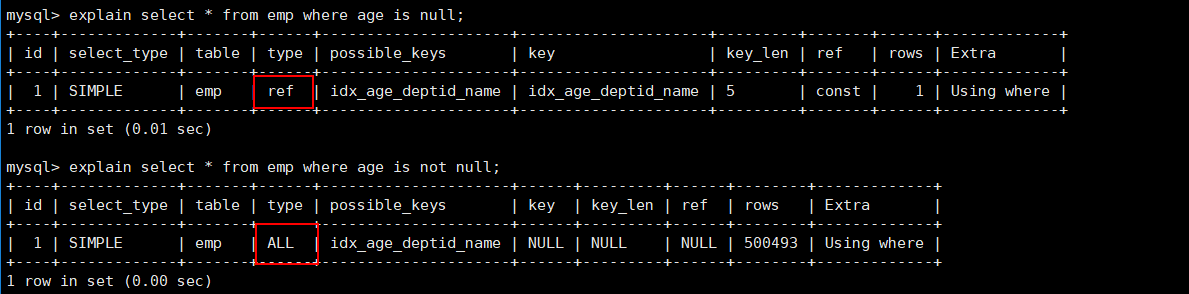
mysql 在使用不等于(!= 或者<>)时，有时会无法使用索引会导致全表扫描。【不等于索引会失效】



## 7. 字段的is not null 和 is null

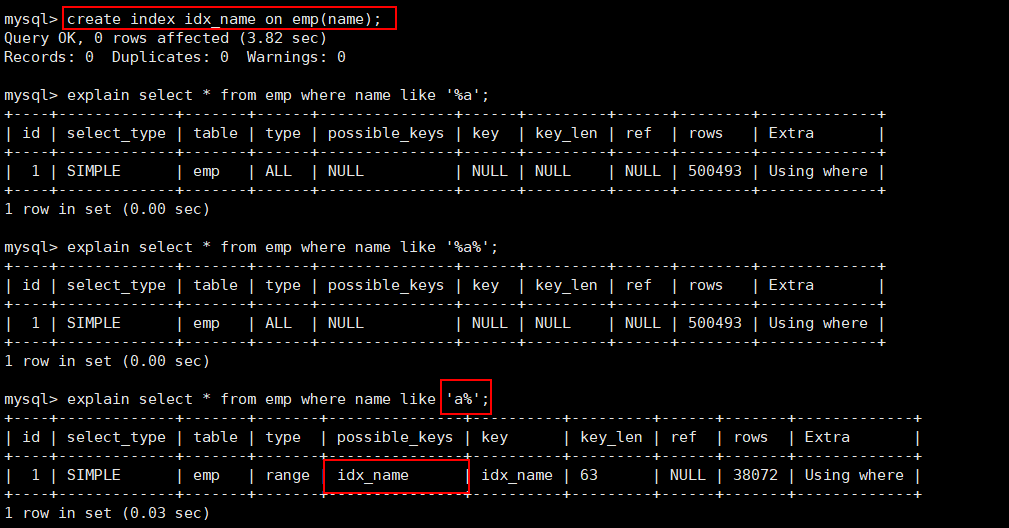


当字段允许为Null的条件下：



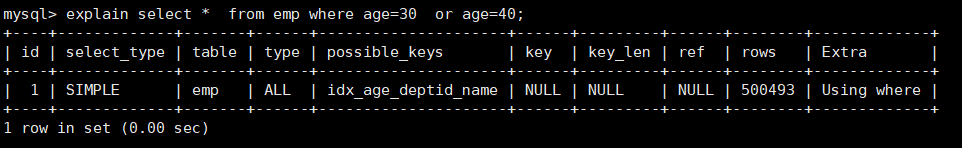
is not null用不到索引，is null可以用到索引。

## 8. like的前后模糊匹配

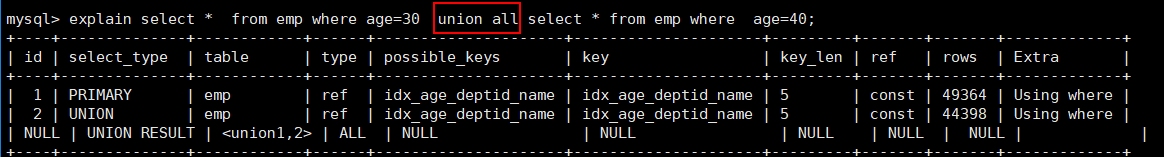


前缀不能出现模糊匹配！

## 9. 减少使用or



使用union all或者union来替代：



## 10. 练习

假设index(a,b,c)；

|  |  |
| --- | --- |
| **Where语句** | **索引是否被使用** |
| where a = 3 | Y,使用到a |
| where a = 3 and b = 5 | Y,使用到a，b |
| where a = 3 and b = 5 and c = 4【可以交换次序哈】 | Y,使用到a,b,c |
| where b = 3 或者 where b = 3 and c = 4 或者 where c = 4 | N |
| where a = 3 and c = 5 | 使用到a， 但是c不可以，b中间断了 |
| where a = 3 and b > 4 and c = 5 | 使用到a和b， c不能用在范围之后，b断了 |
| where a is null and b is not null | is null 支持索引 但是is not null 不支持,所以 a 可以使用索引,但是 b不可以使用 |
| where a <> 3 | 不能使用索引 |
| where abs(a) =3 | 不能使用 索引 |
| where a = 3 and b like 'kk%' and c = 4 | Y,使用到a,b,c |
| where a = 3 and b like '%kk' and c = 4 | Y,只用到a |
| where a = 3 and b like '%kk%' and c = 4 | Y,只用到a |
| where a = 3 and b like 'k%kk%' and c = 4 | Y,使用到a,b,c |

## 11. 口诀

全值匹配我最爱，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。

# 十、关联查询优化

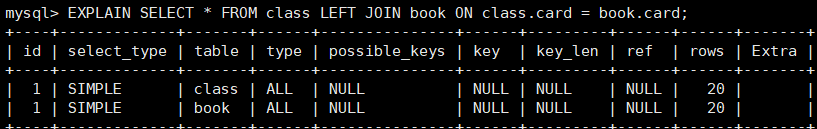
## 1. 建表语句

|  |
| --- |
| CREATE TABLE IF NOT EXISTS `class` (  `id` INT(10) UNSIGNED NOT NULL AUTO\_INCREMENT,  `card` INT(10) UNSIGNED NOT NULL,  PRIMARY KEY (`id`)  );  CREATE TABLE IF NOT EXISTS `book` (  `bookid` INT(10) UNSIGNED NOT NULL AUTO\_INCREMENT,  `card` INT(10) UNSIGNED NOT NULL,  PRIMARY KEY (`bookid`)  );    INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));    INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));  INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20))); |

## 2. 案例

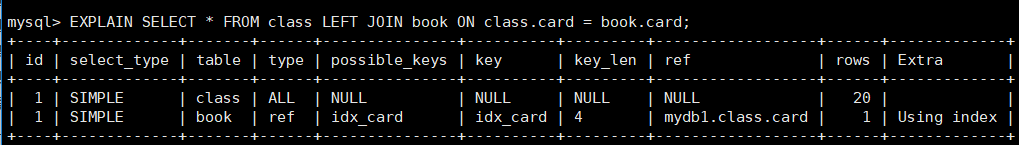
### 2.1 left join

①EXPLAIN SELECT \* FROM class LEFT JOIN book ON class.card = book.card;



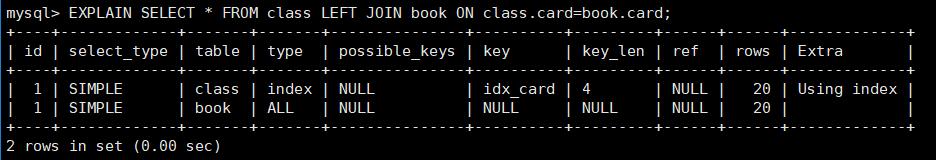
②如何优化？在哪个表上建立索引？

ALTER TABLE `book` ADD INDEX idx\_card( `card`);



③删除book表的索引：drop index idx\_card on book;

在class表上建立索引：alter table class add index idx\_card(card);



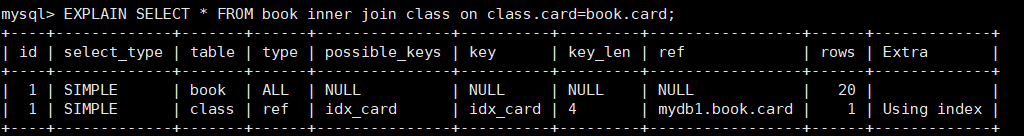
结论：

①在优化关联查询时，只有在被驱动表上建立索引才有效！

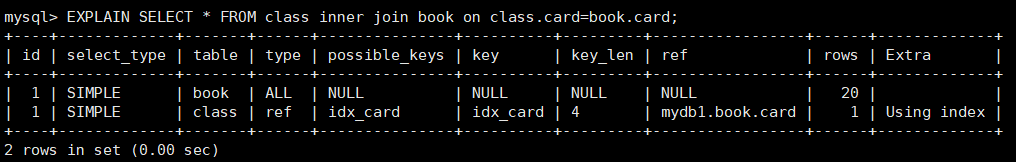
②left join时，左侧的为驱动表，右侧为被驱动表！

### 2.2 inner join

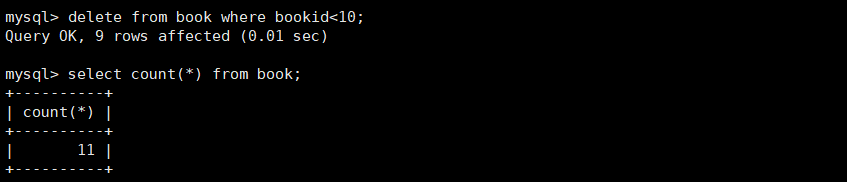
①EXPLAIN SELECT \* FROM book inner join class on class.card=book.card;

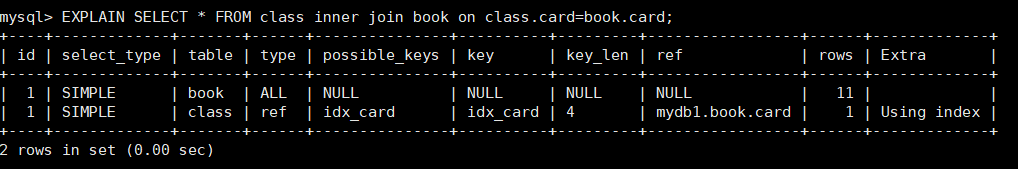


②两个查询字段调换顺序，发现结果也是一样的！



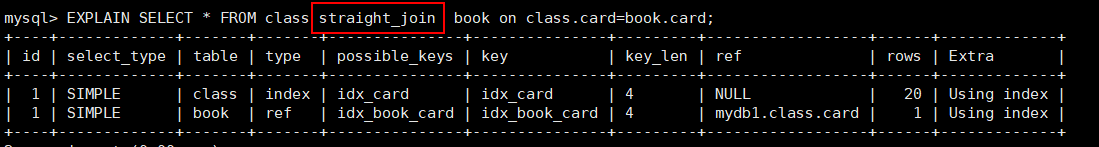
③在book表中，删除9条记录





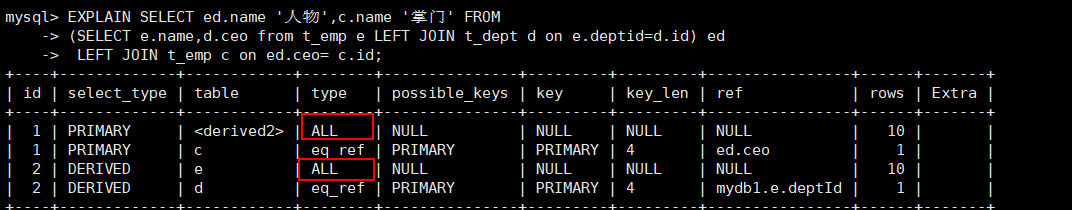
④结论：inner join 时，mysql会自己帮你把小结果集的表选为驱动表。

⑤straight\_join: 效果和inner join一样，但是会强制将左侧作为驱动表！



### 2.3 四个关联查询案例分析

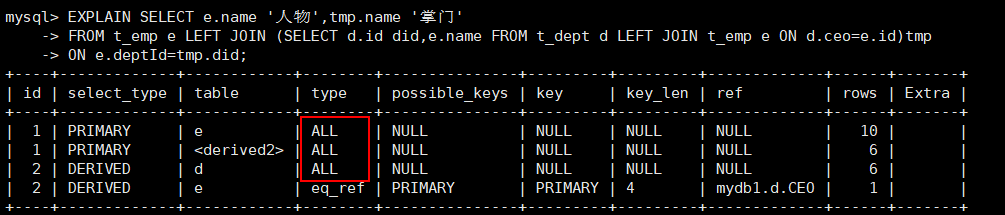
|  |
| --- |
| EXPLAIN SELECT ed.name '人物',c.name '掌门' FROM  (SELECT e.name,d.ceo from t\_emp e LEFT JOIN t\_dept d on e.deptid=d.id) ed  LEFT JOIN t\_emp c on ed.ceo= c.id; |



EXPLAIN SELECT e.name '人物',tmp.name '掌门'

FROM t\_emp e LEFT JOIN (SELECT d.id did,e.name FROM t\_dept d LEFT JOIN t\_emp e ON d.ceo=e.id)tmp

ON e.deptId=tmp.did;



上述两个案例，第一个查询效率较高，且有优化的余地。第二个案例中，子查询作为被驱动表，由于子查询是虚表，无法建立索引，因此不能优化。

**结论**：

子查询尽量不要放在被驱动表，有可能使用不到索引；

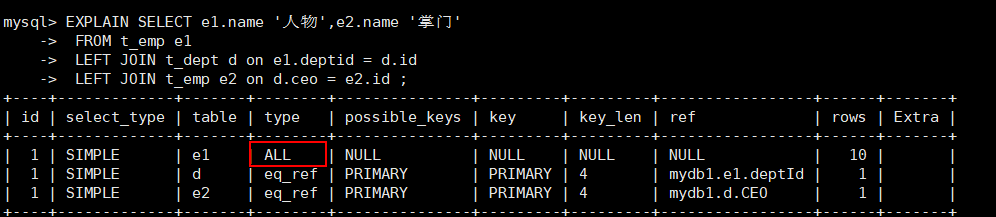
left join时，尽量让实体表作为被驱动表。

EXPLAIN SELECT e1.name '人物',e2.name '掌门'

FROM t\_emp e1

LEFT JOIN t\_dept d on e1.deptid = d.id

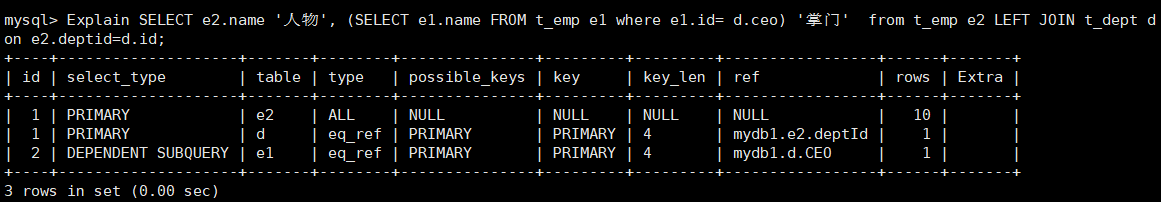
LEFT JOIN t\_emp e2 on d.ceo = e2.id ;



Explain SELECT e2.name '人物',

(SELECT e1.name FROM t\_emp e1 where e1.id= d.ceo) '掌门'

from t\_emp e2 LEFT JOIN t\_dept d on e2.deptid=d.id;



**结论**：能够直接多表关联的尽量直接关联，不用子查询！

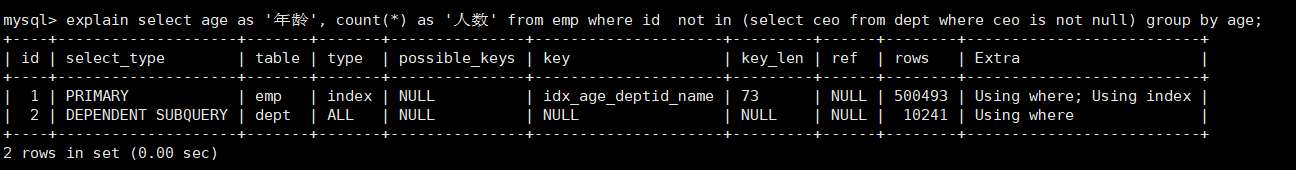
# 十一、子查询优化

## 1. 案例

取所有不为掌门人的员工，按年龄分组！

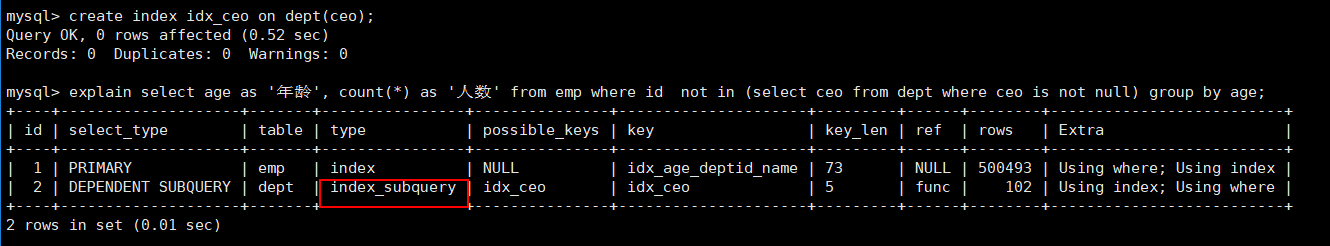
select age as '年龄', count(\*) as '人数' from t\_emp where id not in

(select ceo from t\_dept where ceo is not null) group by age;

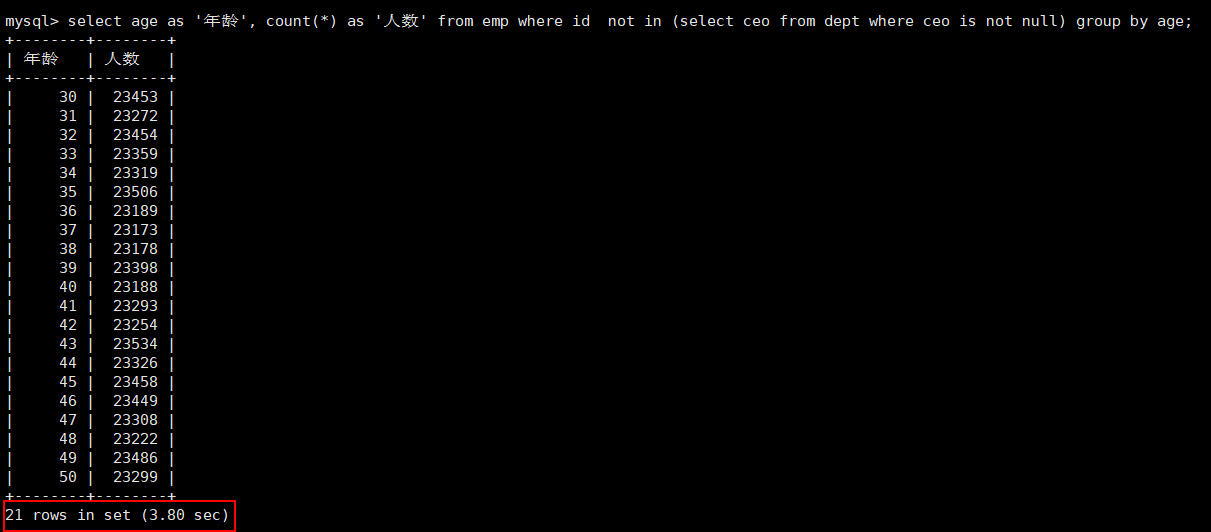


如何优化？

①解决dept表的全表扫描，建立ceo字段的索引：



此时，再次查询：



②进一步优化，替换not in。

上述SQL可以替换为：

select age as '年龄',count(\*) as '人数' from emp e left join dept d on e.id=d.ceo where d.id is null group by age;



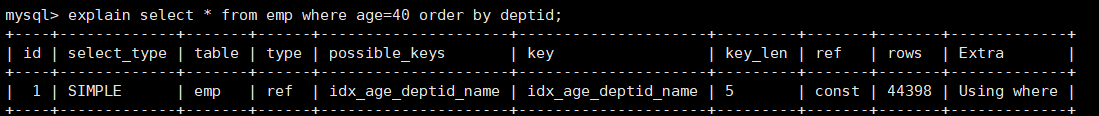
结论： 在范围判断时，尽量不要使用not in和not exists，使用 left join on xxx is null代替。

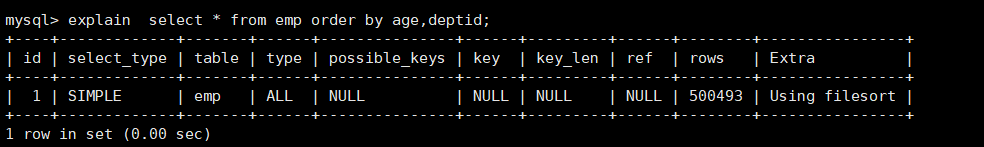
# 十二、排序分组优化

where 条件和 on的判断这些过滤条件，作为优先优化的部分，是要被先考虑的！其次，如果有分组和排序，那么也要考虑grouo by 和order by。

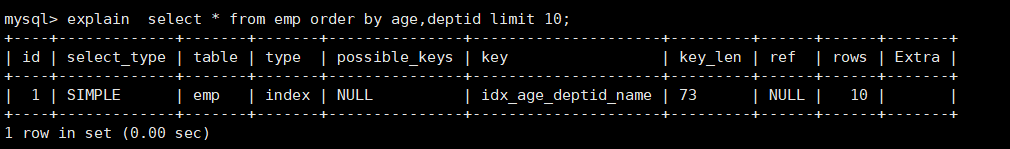
## 1. 无过滤不索引

|  |
| --- |
| create index idx\_age\_deptid\_name on emp (age,deptid,name); |
| explain select \* from emp where age=40 order by deptid; |
| explain select \* from emp order by age,deptid; |
| explain select \* from emp order by age,deptid limit 10; |





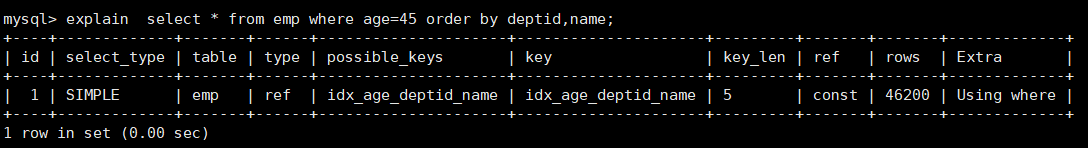
using filesort说明进行了手工排序！原因在于没有where作为过滤条件！



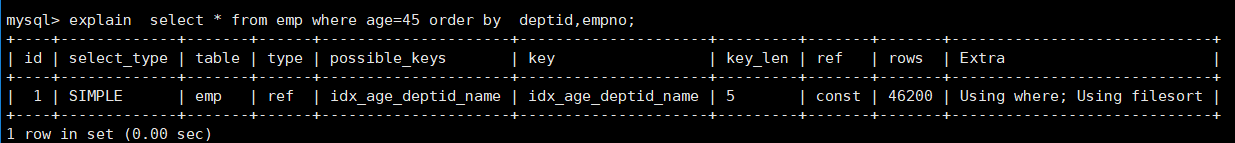
结论： 无过滤，不索引。where，limt都相当于一种过滤条件，所以才能使用上索引！

## 2. 顺序错，必排序

①explain select \* from emp where age=45 order by deptid,name;

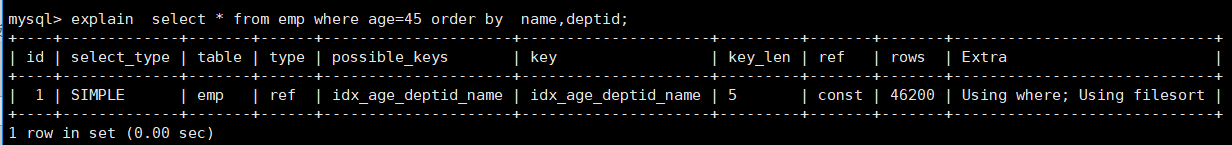


②explain select \* from emp where age=45 order by deptid,empno;



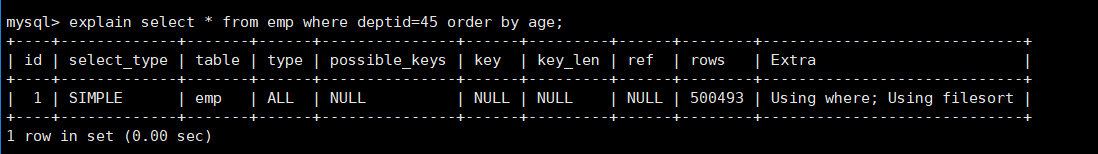
empno字段并没有建立索引，因此也无法用到索引，此字段需要排序！

③explain select \* from emp where age=45 order by name,deptid;



where 两侧列的顺序可以变换，效果相同，但是order by列的顺序不能随便变换！

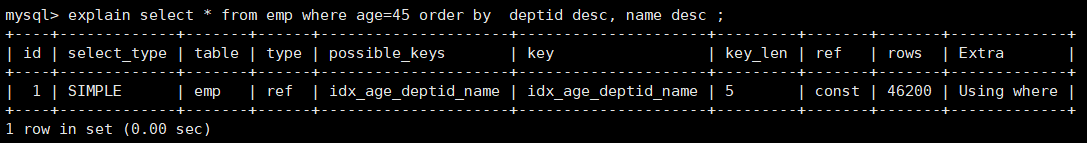
④explain select \* from emp where deptid=45 order by age;



deptid作为过滤条件的字段，无法使用索引，因此排序没法用上索引

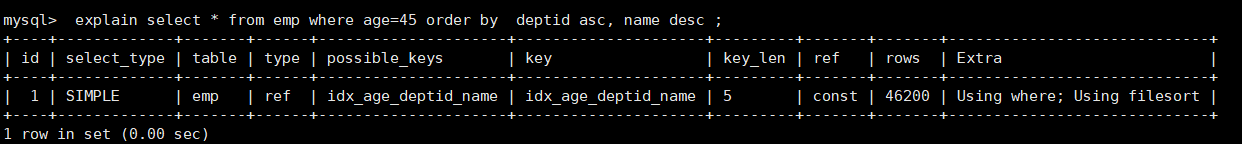
## 3. 方向反，必排序

①explain select \* from emp where age=45 order by deptid desc, name desc ;



如果可以用上索引的字段都使用正序或者逆序，实际上是没有任何影响的，无非将结果集调换顺序。

②explain select \* from emp where age=45 order by deptid asc, name desc ;



如果排序的字段，顺序有差异，就需要将差异的部分，进行一次倒置顺序，因此还是需要手动排序的！

【orderby后面的字段如果有升序有降序需要重新排序，都升序或者都降序不会重排】

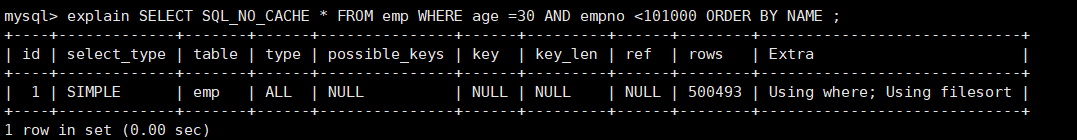
## 4. 索引的选择

①首先，清除emp上面的所有索引，只保留主键索引！

drop index idx\_age\_deptid\_name on emp;

②查询：年龄为30岁的，且员工编号小于101000的用户，按用户名称排序

explain SELECT SQL\_NO\_CACHE \* FROM emp WHERE age =30 AND empno <101000 ORDER BY NAME ;

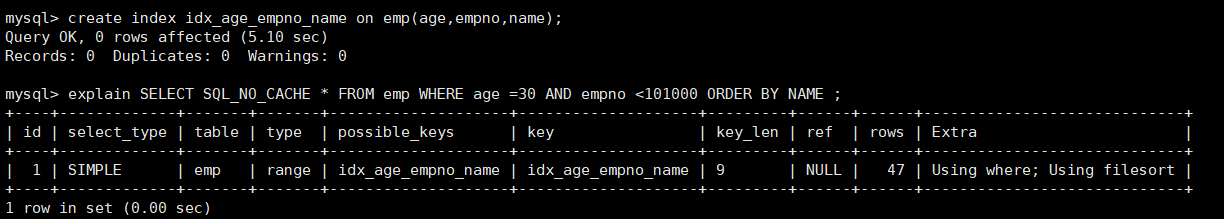


③全表扫描肯定是不被允许的，因此我们要考虑优化。

思路：首先需要让where的过滤条件，用上索引；

查询中，age.empno是查询的过滤条件，而name则是排序的字段，因此我们来创建一个此三个字段的复合索引：

create index idx\_age\_empno\_name on emp(age,empno,name);



再次查询，发现using filesort依然存在。

原因： empno是范围查询，因此导致了索引失效，所以name字段无法使用索引排序。

所以，三个字段的符合索引，没有意义，因为empno和name字段只能选择其一！

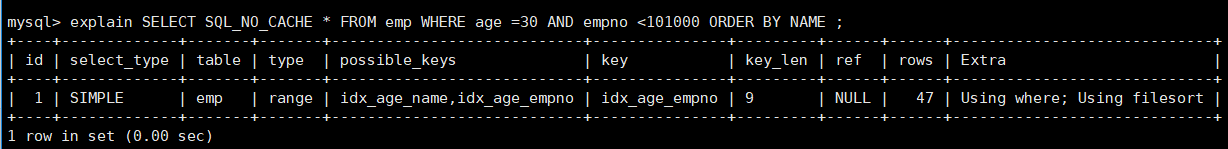
④解决： 鱼与熊掌不可兼得，因此，要么选择empno,要么选择name

drop index idx\_age\_empno\_name on emp;

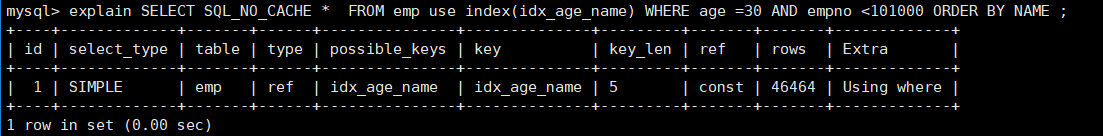
create index idx\_age\_name on emp(age,name);

create index idx\_age\_empno on emp(age,empno);

两个索引同时存在，mysql会选择哪个？



explain SELECT SQL\_NO\_CACHE \* FROM emp use index(idx\_age\_name) WHERE age =30 AND empno <101000 ORDER BY NAME ;

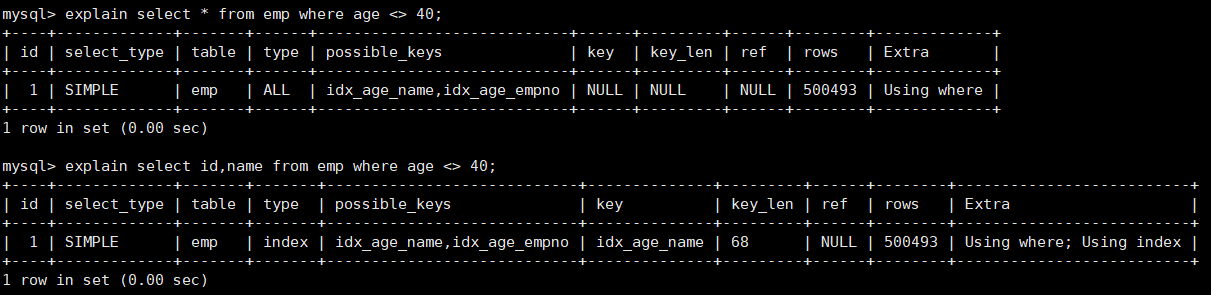


原因：所有的排序都是在条件过滤之后才执行的，所以如果条件过滤了大部分数据的话，几百几千条数据进行排序其实并不是很消耗性能，即使索引优化了排序但实际提升性能很有限。 相对的 empno<101000 这个条件如果没有用到索引的话，要对几万条的数据进行扫描，这是非常消耗性能的，使用empno字段的范围查询，过滤性更好（empno从100000开始）！

结论： 当范围条件和group by 或者 order by 的字段出现二选一时 ，优先观察条件字段的过滤数量，如果过滤的数据足够多，而需要排序的数据并不多时，优先把索引放在范围字段上。反之，亦然。

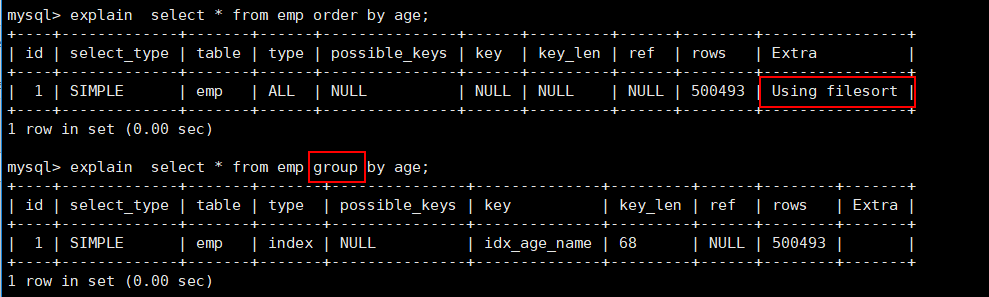
## 5. 使用覆盖索引

覆盖索引：SQL只需要通过索引就可以返回查询所需要的数据，而不必通过二级索引查到主键之后再去查询数据。



## 6. group by

group by 使用索引的原则几乎跟order by一致 ，唯一区别是group by 即使没有过滤条件用到索引，也可以直接使用索引。【group by可以不使用过滤】



# 十三、练习

## 1. 案例一

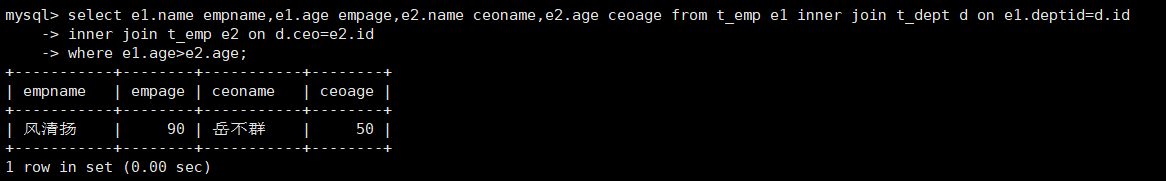
列出自己的掌门比自己年龄小的人员

select e1.name empname,e1.age empage,e2.name ceoname,e2.age ceoage

from t\_emp e1 inner join t\_dept d on e1.deptid=d.id

inner join t\_emp e2 on d.ceo=e2.id

where e1.age>e2.age;

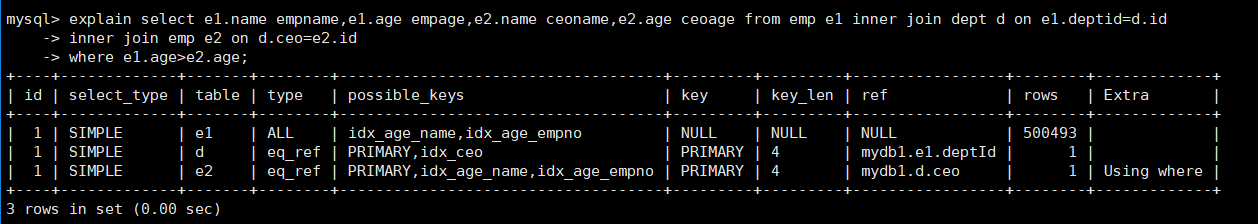


更换为大表，进行分析：

explain select e1.name empname,e1.age empage,e2.name ceoname,e2.age ceo age from emp e1 inner join dept d on e1.deptid=d.id

inner join emp e2 on d.ceo=e2.id

where e1.age>e2.age;



两次inner join的被驱动表都已经用上了索引。

## 2. 案例二

列出所有年龄低于自己门派平均年龄的人员

思路： 先取门派的平均年龄，再跟自己的年龄做对比！

select e1.name from t\_emp e1

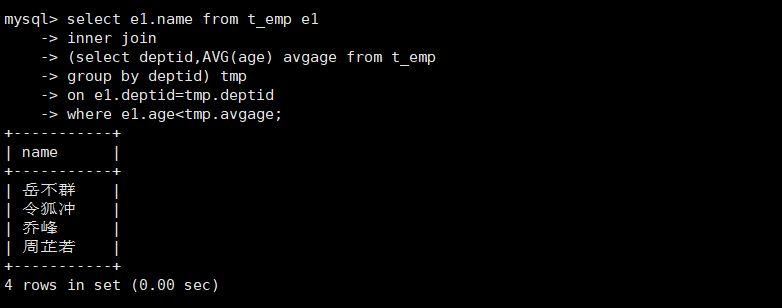
inner join

(select deptid,AVG(age) avgage from t\_emp

group by deptid) tmp

on e1.deptid=tmp.deptid

where e1.age<tmp.avgage;



更换为大表：

explain select e1.name from emp e1

inner join

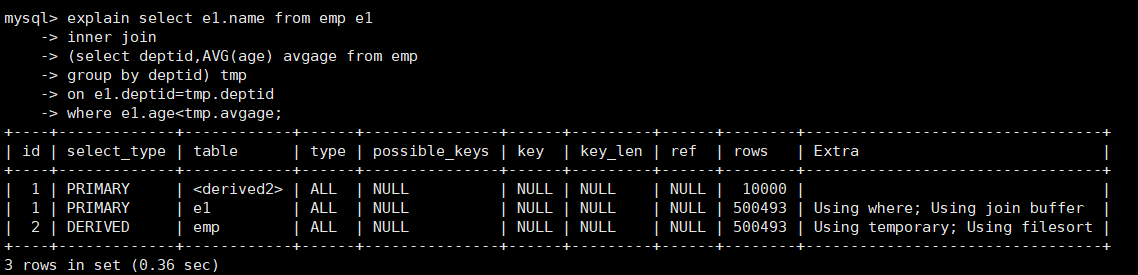
(select deptid,AVG(age) avgage from emp

group by deptid) tmp

on e1.deptid=tmp.deptid

where e1.age<tmp.avgage;

在没有索引的前提下：



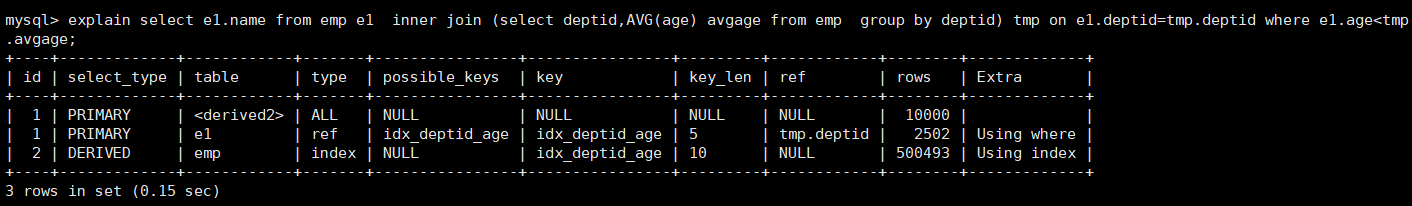
如何优化：

①首先在子查询中，需要根据deptid做groupby操作，因此，需要在deptid上面建立索引；

②因为是inner join,因此会自动将小表作为驱动表，也就是说，分组后的tmp是驱动表，而e1是被驱动表；

③而在e1中，需要查询deptid和age两个字段，因此这两个字段也需要建立索引

结果：创建deptid和age的符合索引: create index idx\_deptid\_age on emp(deptid,age);



## 3. 案例三

列出至少有2个年龄大于40岁的成员的门派名称

思路： 先查询大于40岁的成员，然后按照门派分组，然后再判断至少有2个的门派！

select d.deptName,count(\*)

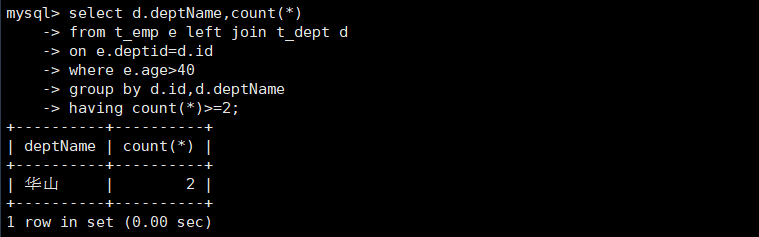
from t\_emp e inner join t\_dept d

on e.deptid=d.id

where e.age>40

group by d.id,d.deptName

having count(\*)>=2



大表优化：

explain select d.deptName,count(\*)

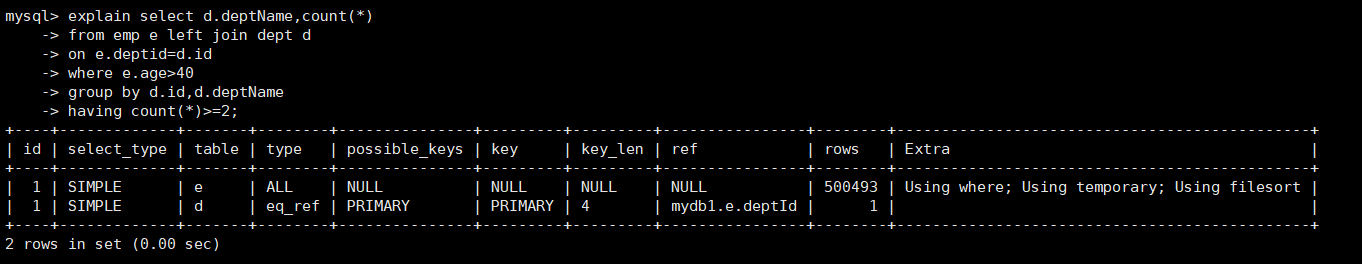
from emp e inner join dept d

on e.deptid=d.id

where e.age>40

group by d.id,d.deptName

having count(\*)>=2;



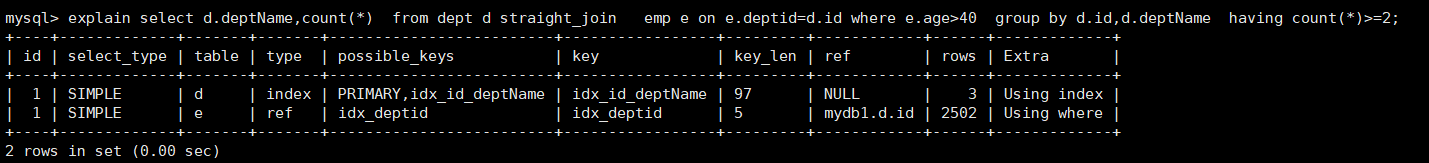
优化：

①两表关联，我们可以考虑将小表作为驱动表。

②group by的字段 id,deptName还可以建立索引: create index idx\_id\_deptName on dept(id,deptName);

③被驱动表的deptid作为关联字段，可以建立索引：create index idx\_deptid on emp(deptid);

create index idx\_id\_deptname on dept(id,deptName);



## 4. 案例四

至少有2位非掌门人成员的门派

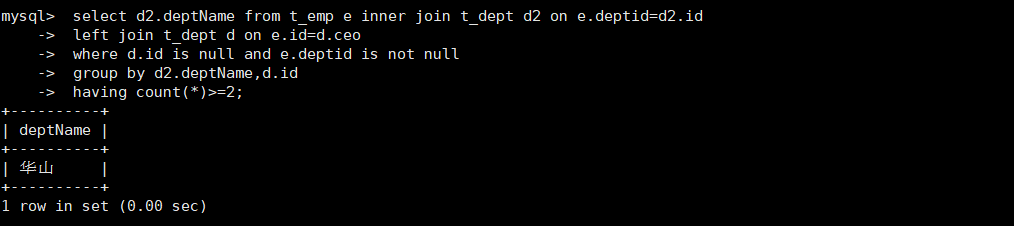
select d2.deptName from t\_emp e inner join t\_dept d2 on e.deptid=d2.id

left join t\_dept d on e.id=d.ceo

where d.id is null and e.deptid is not null

group by d2.deptName,d2.id

having count(\*)>=2;



切换大表：

explain select d2.deptName from emp e inner join dept d2 on e.deptid=d2.id

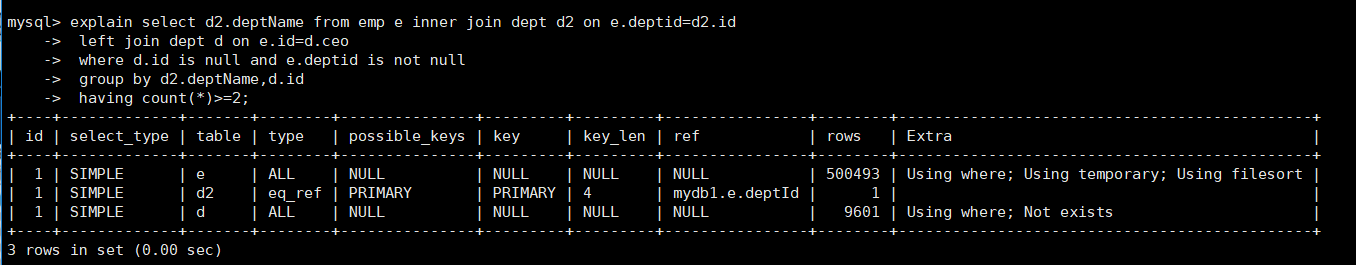
left join dept d on e.id=d.ceo

where d.id is null and e.deptid is not null

group by d2.deptName,d2.id

having count(\*)>=2;

没有索引的情况下：



优化分析： 三个表关联，然后做group by分组！

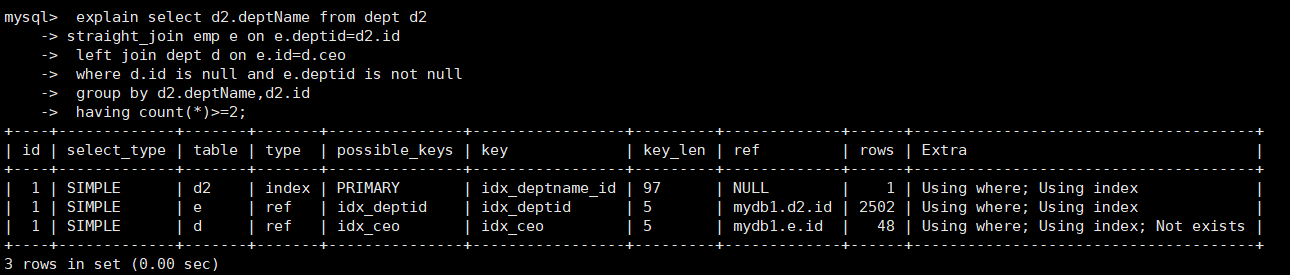
①group by 的字段，可以加上索引：create index idx\_deptname\_id on dept(deptName,id);

②可以将部门表作为驱动表

③第一次join时，e表作为被驱动表，可以将deptid设置索引：create index idx\_deptid on emp(deptid);

④最有一次join中，使用了dept表作为被驱动表，查询ceo字段，因此可以在ceo上面建立索引

create index idx\_ceo on dept(ceo);



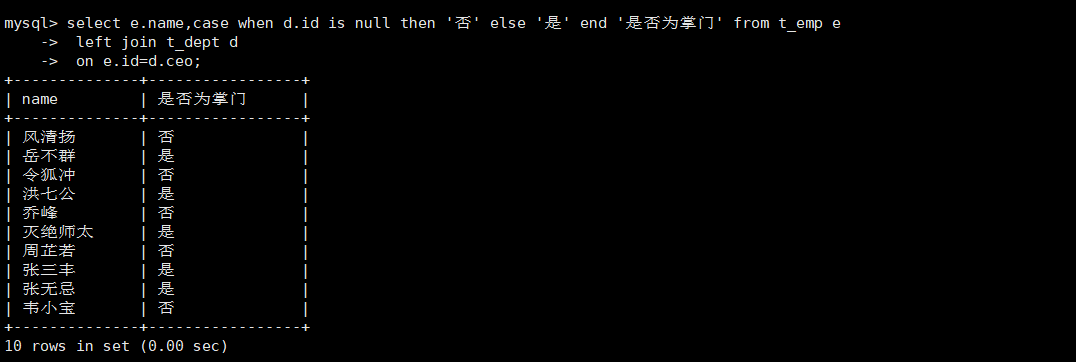
## 5. 案例五

列出全部人员，并增加一列备注“是否为掌门”，如果是掌门人显示是，不是掌门人显示否

select e.name,case when d.id is null then '否' else '是' end '是否为掌门' from t\_emp e

left join t\_dept d

on e.id=d.ceo;



大表关联：

explain select e.name,case when d.id is null then '否' else '是' end '是否为掌门' from emp e

left join dept d

on e.id=d.ceo;

优化：在d表的ceo字段建立索引即可！

## 6. 案例六

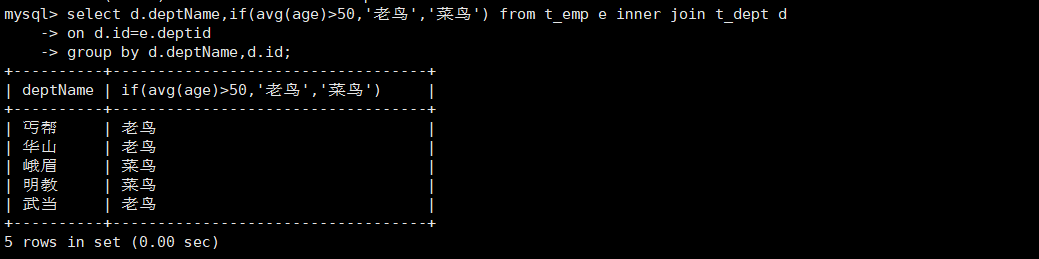
列出全部门派，并增加一列备注“老鸟or菜鸟”，若门派的平均值年龄>40显示“老鸟”，否则显示“菜鸟”

思路： 先从emp表求出，各门派的平均年龄，分组，然后在关联dept表，使用if函数进行判断！

select d.deptName,if(avg(age)>40,'老鸟','菜鸟') from t\_emp e inner join t\_dept d

on d.id=e.deptid

group by d.deptName,d.id

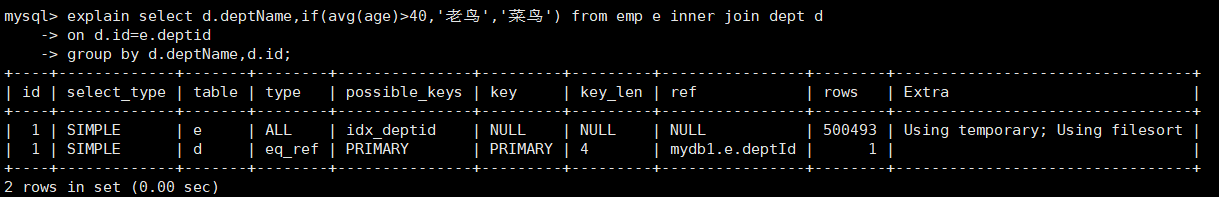


切换大表：

explain select d.deptName,if(avg(age)>40,'老鸟','菜鸟') from dept d inner join emp e

on d.id=e.deptid

group by d.deptName,d.id

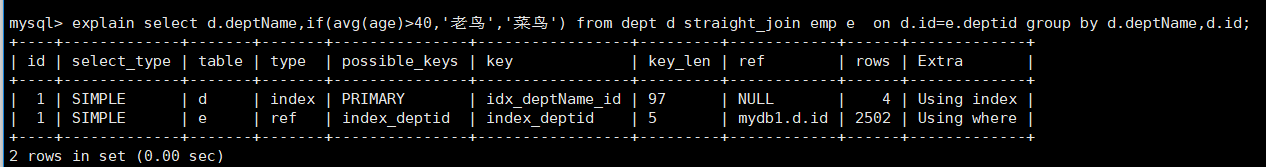


优化：

①使用dept作为驱动表

②在dept上建立deptName和id的索引：create index idx\_deptName\_id on dept(deptName,id);

③在emp上建立deptid字段的索引： create index index\_deptid on emp(deptid);



## 7. 案例七

显示每个门派年龄最大的人

思路：先查询emp表，求出每个门派年龄最大的人，并按照deptid分组；然后再次关联emp表，关联其他的信息！

select \* from t\_emp e

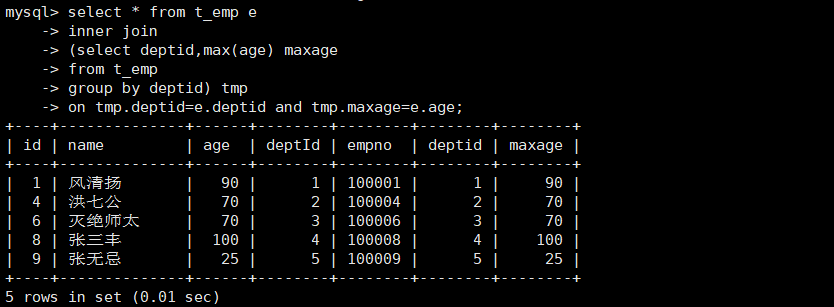
inner join

(select deptid,max(age) maxage

from t\_emp

group by deptid) tmp

on tmp.deptid=e.deptid and tmp.maxage=e.age;



大表优化：

explain select \* from emp e

inner join

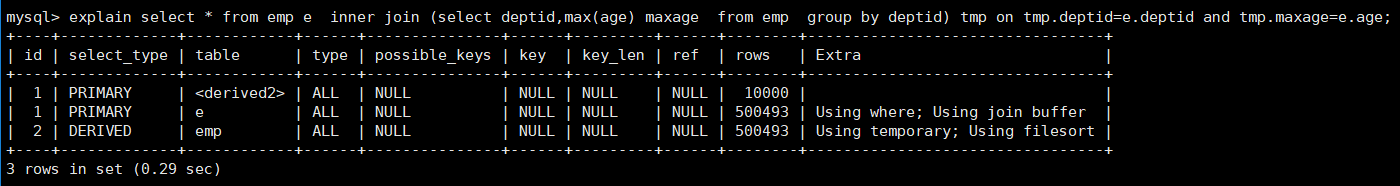
(select deptid,max(age) maxage

from emp

group by deptid) tmp

on tmp.deptid=e.deptid and tmp.maxage=e.age;

优化前：



优化思路：

①子查询中，emp表根据deptid进行分组，因此可以建立deptid字段的索引；

②inner join查询中，关联了age和deptid，因此可以在deptid,age字段建立索引

create index idx\_deptid\_age on emp(deptid,age);

