

# 1. MySQL 重要性质

什么都不说，先看张图

<https://db-engines.com/en/ranking>

Rank	DBMS			Database Model	Score		
	Jun 2019	May 2019	Jun 2018		Jun 2019	May 2019	Jun 2018
1.	1.	1.	Oracle +	Relational, Multi-model	1299.21	+13.67	-12.04
2.	2.	2.	MySQL +	Relational, Multi-model	1223.63	+4.67	-10.06
3.	3.	3.	Microsoft SQL Server +	Relational, Multi-model	1087.76	+15.57	+0.03
4.	4.	4.	PostgreSQL +	Relational, Multi-model	476.62	-2.27	+65.95
5.	5.	5.	MongoDB +	Document	403.90	-4.17	+60.12
6.	6.	6.	IBM Db2 +	Relational, Multi-model	172.20	-2.24	-13.44
7.	7.	8.	Elasticsearch +	Search engine, Multi-model	148.82	+0.20	+17.78
8.	8.	7.	Redis +	Key-value, Multi-model	146.13	-2.28	+9.83
9.	9.	9.	Microsoft Access	Relational	141.01	-2.77	+10.02
10.	10.	10.	Cassandra +	Wide column	125.18	-0.54	+5.97
11.	11.	11.	SQLite +	Relational	124.89	+1.99	+10.63
12.	12.	13.	MariaDB +	Relational, Multi-model	85.20	-1.32	+19.35
13.	13.	14.	Splunk	Search engine	84.62	-0.62	+18.84
14.	14.	18.	Hive +	Relational	79.06	+1.16	+21.73
15.	15.	12.	Teradata +	Relational	76.64	+0.60	+0.87

在所有数据库中，MySQL 排在第二，而 nosql 中 mongodb 排在第一，你可能在想是不是有必要把 Oracle 也学习下，别着急，再看张图



全球访问量最大的 20 家网站，他们分别使用了什么数据库呢，绝大多数使用 mysql, 有两个完整 live.com 和 bing 使用的是 mssql，并不是他们使用不了 mysql，而是他要支持自己的数据库。

在国外可能挺多使用 mssql 或者 oracle 的，但是在过能，在去 IOE 的大背景下，包括银行在内的很多传统公司慢慢都在像 mysql 转型，不过其中有个老大不掉的公司，中国电力，依然使用 oracle，在十年的时间仅仅在 oracle 的使用上，中国电力就支出 390 几个亿，平均一年 30, 40 个亿，它有钱，如果你所在公司随随便便也能拿个几百个亿，那你也用 oracle 吧

# 2. MySQL 安装

在第一期，课程是在 windows 安装的 mysql，但第二期，同学们都有 linux 基础了，所以，第二期会在 linux 上使用，没有使用过 linux 的同学先去看下第一期 peter 老师的视频。

## 2.1. 准备工作

Linux 使用的版本是 centos 7, 为方便起见, 先把防火墙关闭, 配置好网络, 在安装部分, 会分成两部分讲, 首先讲单实例安装, 也就是一台服务器上就装一个 mysql, 接下来就多实例安装, 在一个服务器上安装 2 个甚至多个 mysql.

## 2.2. 单实例安装

```
set global show_compatibility_56=on;
```

```
cp /soft/mysql-5.7.9-linux-glibc2.5-x86_64.tar.gz /usr/local/
```

解压 mysql 到 /usr/local 目录

解压:

```
tar -zxvf mysql-5.7.9-linux-glibc2.5-x86_64.tar.gz
```

安装需要的依赖

```
yum install -y libaio
```

具体安装

```
shell> groupadd mysql  
shell> useradd -r -g mysql mysql  
shell> cd /usr/local  
shell> tar zxvf /path/to/mysql-VERSION-OS.tar.gz  
shell> ln -s full-path-to-mysql-VERSION-OS mysql  
shell> cd mysql  
shell> mkdir mysql-files  
shell> chmod 770 mysql-files  
shell> chown -R mysql .  
shell> chgrp -R mysql .  
shell> bin/mysqld --initialize --user=mysql      # MySQL 5.7.6 and up  
shell> bin/mysql_ssl_rsa_setup                  # MySQL 5.7.6 and up  
shell> chown -R root .  
shell> chown -R mysql data mysql-files  
shell> bin/mysqld_safe --user=mysql &  
# Next command is optional  
shell> cp support-files/mysql.server /etc/init.d/mysql.server
```

配置环境变量：

```
export PATH=/usr/local/mysql/bin:$PATH
```

配置开启启动

```
chkconfig mysql.server on
```

```
chkconfig --list
```

登陆，修改密码

```
set password = 'root1234%';
```

允许远程登陆

```
GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED BY 'root1234%'
```

```
GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED BY 'root1234%' WITH GRANT OPTION;
```

```
flush privileges;
```

启动的时候可能会报错

```
[[root@mysql mysql]# bin/mysqld_safe --user=mysql &
[1] 8511
[root@mysql mysql]# 190611 10:52:15 mysqld_safe Logging to '/var/log/mariadb/mariadb.log'.
touch: cannot touch '/var/log/mariadb/mariadb.log': No such file or directory
chmod: cannot access '/var/log/mariadb/mariadb.log': No such file or directory
touch: cannot touch '/var/log/mariadb/mariadb.log': No such file or directory
chown: cannot access '/var/log/mariadb/mariadb.log': No such file or directory
190611 10:52:15 mysqld_safe starting mysqld daemon with databases from /var/lib/mysql
bin/mysqld_safe: line 130: /var/log/mariadb/mariadb.log: No such file or directory
bin/mysqld_safe: line 164: /var/log/mariadb/mariadb.log: No such file or directory
touch: cannot touch '/var/log/mariadb/mariadb.log': No such file or directory
chown: cannot access '/var/log/mariadb/mariadb.log': No such file or directory
chmod: cannot access '/var/log/mariadb/mariadb.log': No such file or directory
190611 10:52:15 mysqld_safe mysqld from pid file /var/run/mariadb/mariadb.pid ended
bin/mysqld_safe: line 130: /var/log/mariadb/mariadb.log: No such file or directory
^C
```

这是因为 mysql 启动的时候需要配置文件，而在安装 centos 的时候，哪怕是 mini 版本都会有个默认的配置在/etc 目录中

```
/usr/local/mysql/bin/mysqld --verbose --help |grep -A 1 'Default options'
```

```
Default options are read from the following files in the given order:
```

```
/etc/my.cnf /etc/mysql/my.cnf /usr/local/mysql/etc/my.cnf ~/.my.cnf
```

Mysql 启动的时候会以上面所述的顺序加载配置文件

如果报错，先重命名 my.cnf 文件

## 2.3. 多实例安装

以前一些很 low 的方法是，解压两个 mysql，分别放到不同文件夹，其实在 mysql 中已经考虑到了多实例安装的情况。也有相应的脚本命令的支持。

现在要求装两个 mysql 一个 3307，3308

新建 /etc/my.cnf 配置如下

```
[mysqld]
sql_mode
"STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION,NO_ZERO_DATE,NO_ZERO_IN_DATE,ERROR
FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER"

[mysqld_multi]
mysqld = /usr/local/mysql/bin/mysqld_safe
mysqladmin = /usr/local/mysql/bin/mysqladmin
log = /var/log/mysqld_multi.log

[mysqld1]
server-id = 11
socket = /tmp/mysql.sock1
port = 3307
datadir = /data1
user = mysql
performance_schema = off
innodb_buffer_pool_size = 32M
skip_name_resolve = 1
log_error = error.log
pid-file = /data1/mysql.pid1
[mysqld2]
server-id = 12
socket = /tmp/mysql.sock2
port = 3308
datadir = /data2
user = mysql
performance_schema = off
innodb_buffer_pool_size = 32M
skip_name_resolve = 1
log_error = error.log
pid-file = /data2/mysql.pid2
```

创建 2 个数据目录

```
mkdir /data1
mkdir /data2
```

```
chown mysql.mysql /data{1..2}
```

```
mysqld --initialize --user=mysql --datadir=/data1
```

```
mysqld --initialize --user=mysql --datadir=/data2
```

```
cp /usr/local/mysql/support-files/mysqld_multi.server /etc/init.d/mysqld_multid
```

配置开机启动

```
chkconfig mysqld_multid on
```

查看状态

```
mysqld_multi report
```

```
[root@mysql ~]# mysqld_multi report  
-bash: /usr/local/mysql/bin/mysqld_multi: /usr/bin/perl: bad interpreter: No such file or directory
```

这个时候发现还需要 perl 的环境，安装

```
yum -y install perl perl-devel
```

在运行，发现已经有实例了

```
mysqld_multi report
```

```
[root@mysql ~]# mysqld_multi report  
Reporting MySQL servers  
MySQL server from group: mysqld1 is not running  
MySQL server from group: mysqld2 is not running
```

```
mysqld_multi start
```

启动，分别修改密码，允许远程连接

```
mysql -u root -S /tmp/mysql.sock1 -p -P3307
```

```
mysql -u root -S /tmp/mysql.sock2 -p -P3308
```

```
set password = 'root1234%';  
GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED BY 'root1234%';  
flush privileges;
```

## 3. Mysql 权限

### 3.1. 最简单的 MySql 权限

最简单也是最高效的，如果解决新手们删库跑路的问题其实也是很简单的，对于正式库只给一个增删改查的权限，或者只给一个查询权限（是不是就解决了删库的可能性？）

以下内容如果看官是大牛，请稍安勿躁，我讲内容的方式是从简单到入门，从入门到进阶，从进阶到实战，从实战到。。。 （包你满意）

使用 Root 用户，执行

```
grant SELECT on mall.* TO 'dev'@'192.168.244.%' IDENTIFIED BY '123' WITH GRANT OPTION;
```

很简单的一句 sql，创建了一个 dev 的用户，密码为 123，仅仅运行在网段为 192.168.0.\* 的网段进行查询操作。

再执行一条命令

```
show grants for 'dev'@'192.168.244.%'
```

```
Grants for dev@192.168.0.%  
▶ GRANT USAGE ON *.* TO 'dev'@'192.168.0.%' IDENTIFIED BY PASSWORD ''23AE809DDACAF96AF0FD78ED04B6A265E05AA257'  
GRANT SELECT ON `mall`.* TO 'dev'@'192.168.0.%' WITH GRANT OPTION
```

不错，可以链接，也可以执行 select，这个时候还想删库？做梦吧~

## 3.2. 深入研究下 MySQL 权限

### 3.2.1. 用户标识是什么

上面一句简单的 SQL 堪称完美的解决了程序员新手的删库跑路的问题，高兴吧，你学到了新姿势，但是如果想面试给面试官留下好映像，上面的知识好像还不够，有必要好好深入研究下 MySQL 的权限了。

这里有个小的知识点需要先具备，在 mysql 中的权限不是单纯的赋予给用户的，而是赋予给“用户+IP”的

比如 dev 用户是否能登陆，用什么密码登陆，并且能访问什么数据库等都需要加上 IP，这样才算一个完整的用户标识，换句话说 'dev'@'192.168.0.168' 、 'dev'@'127.0.0.1' 与 'dev'@'localhost' 这 3 个是完全不同的用户标识（哪怕你本机的 ip 就是 192.168.0.168）。

### 3.2.2. 用户权限所涉及的表

有了用户标识的概念接下来就可以看权限涉及的表了，这也是面试的时候加分项哦。

有几张表你可以好好的记记的，mysql.user，mysql.db，mysql.table\_priv，mysql\_column\_priv  
你可以熟悉其中的 user 表，甚至手动的改过里面的数据（不合规范哦！）

那这些表有什么用，和权限又有什么关系呢？

- User 的一行记录代表一个用户标识
- db 的一行记录代表对数据库的权限
- table\_priv 的一行记录代表对表的权限

- column\_priv 的一行记录代表对某一列的权限

新建一个表 account

```
DROP TABLE IF EXISTS `account`;
CREATE TABLE `account` (
  `id` int(11) NOT NULL,
  `name` varchar(50) DEFAULT NULL,
  `balance` int(255) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `idx_balance` (`balance`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
INSERT INTO `account` VALUES ('1', 'lilei', '900');
INSERT INTO `account` VALUES ('2', 'hanmei', '100');
INSERT INTO `account` VALUES ('3', 'lucy', '250');
INSERT INTO `account` VALUES ('5', 'tom', '0');
```

很诧异吧，mysql 其实权限并不特别 low，权限的粒度甚至到了某一列上，举例来说，有个表 account 表

id	name	balance
1	lilei	900
2	hanmei	100
3	lucy	250
5	tom	0

对于前面创建的 dev 用户我不想让他访问 balance 列，但是 id 和 name 列是可以访问的，这样的需求在工作中不是没有。

```
grant select(id,name) on mall.account to 'dev'@'192.168.244.%';
```

这时候可以在分别看下 table\_priv, column\_priv 的数据

对象 tables_priv @mysql (localhost)						
Host	Db	User	Table_name	Grantor	Timestamp	Table_priv
192.168.0.6	mysqldemo	dev	account	root@localhost	0000-00-00 00:00:00	Select

对象 columns_priv @mysql (localhost)						
Host	Db	User	Table_name	Column_name	Timestamp	Column_priv
192.168.0.%	mysqldemo	dev	account	name	0000-00-00 00:00:00	Select
192.168.0.%	mysqldemo	dev	account	id	0000-00-00 00:00:00	Select

这个就应该豁然开朗了吧

```
REVOKE SELECT on mall.* from 'dev'@'192.168.244.%'
```

你使用 dev 登陆查询试试



你再要查询所有记录？不好意思不让查

而你查询 id, name 查询又是可以了。

A screenshot of MySQL Workbench showing the results of a query:

```
1 select id, name from account
```

The results table has four columns: 信息, 结果1, 概况, and 状态. The data is as follows:

信息	结果1	概况	状态
id	name		
1	lilei		
2	hanmei		
3	lucy		
5	tom		

### 3.2.3. MySql 的角色

#### 3.2.3.1. 准备工作

MySQL 基于“用户+IP”的这种授权模式其实还是挺好用的，但如果你使用 Oracle、PostgreSQL、SqlServer 你可能会发牢骚

这样对于每个用户都要赋权的方式是不是太麻烦了，如果我用户多呢？有没有角色或者用户组这样的功能呢？

好吧，你搓中了 mysql 的软肋，很痛，在 mysql5.7 开始才正式支持这个功能，而且连 mysql 官方把它叫做“Role Like”（不是角色，长得比较像而已，额~~~）？

好咧，那在 5.7 中怎么玩这个不像角色的角色呢？

```
show variables like "%proxy%"
```

1 show variables like "%proxy%"	
信息	结果1
Variable_name	Value
► check_proxy_users	OFF
mysql_native_password_proxy_users	OFF
proxy_user	
sha256_password_proxy_user	OFF

你得先把 `check_proxy_users`, `mysql_native_password_proxy_users` 这两个变量设置成 `true` 才行

```
set GLOBAL check_proxy_users =1;
set GLOBAL mysql_native_password_proxy_users = 1;
```

当然，你也可以把这两个配置设置到 `my.cnf` 中

### 3.2.3.2. 创建一个角色

```
create USER 'dev_role'
```

可能被你发现了，我这里创建得是个 `user`,为了稍微像角色一点点，我给这 `user` 取名叫 `dev_role`，而且为了方便也没用使用密码了。

### 3.2.3.3. 创建 2 个开发人员账号：

```
create USER 'deer'
create USER 'enjoy'
```

这两个用户我也没设置密码

### 3.2.3.4. 把两个用户加到组里面

```
grant proxy on 'dev_role' to  'deer'
grant proxy on 'dev_role' to  'enjoy'
```

可以看下其中一个用户的权限

```
21  
22 show GRANTS for 'deer'  
23
```

信息	结果1	概况	状态
Grants for deer@%			
▶	GRANT USAGE ON `*.*` TO 'deer'@'%' GRANT PROXY ON 'dev_role'@'%' TO 'deer'@'%'		

这里有个小的地方需要注意：如果你是远程链接，你可能会收获一个大大的错误，你没有权限做这一步，这个时候你需要再服务器上执行一条

```
GRANT PROXY ON "@" TO 'root'@'%' WITH GRANT OPTION;
```

### 3.2.3.5. 给角色 dev\_role 应该有的权限

有了用户了，这用户也归属到了 dev\_role 这角色下面，那接下来要做的就很简单了，根据业务需求给这角色设置权限就好了

```
grant select(id,name) on mall.account to 'dev_role'
```

### 3.2.3.6. 测试

好咧，大功告成，现在使用'deer'用户登陆系统试试

```
select id ,name from mall.account
```

学到了吧，涨姿势了吧，你可能又会问，这种角色的权限是存哪的呢？

给你看表

对象	proxies_priv @mysql (24.1...)	开始事务	备注	筛选	排序	导入	导出
Host	User	Proxied_host	Proxied_user	With_grant	Grantor	Timestamp	
localhost	root				1 boot@connecting host	0000-00-00 00:00:00	
%	deer	%	dev_role		0 root@localhost	0000-00-00 00:00:00	
%	enjoy	%	dev_role		0 root@localhost	0000-00-00 00:00:00	
%	root				1 root@localhost	0000-00-00 00:00:00	

## 4. MySql 数据类型

### 4.1.1. Int 类型

类型	字节	最小值	最大值
(带符号的/无符号的)			(带符号的/无符号的)
TINYINT	1	-128	127
		0	255
SMALLINT	2	-32768	32767
		0	65535
MEDIUMINT	3	-8388608	8388607
		0	16777215
INT	4	-2147483648	2147483647
		0	4294967295
BIGINT	8	-9223372036854775808	9223372036854775807
		0	18446744073709551615

#### 4.1.1.1. 有无符号

在项目中使用 BIGINT，而且是有符号的。

演示

```
create table test_unsigned(a int unsigned, b int unsigned);
insert into test_unsigned values(1, 2);
select b - a from test_unsigned;
select a - b from test_unsigned; --运行出错
```

#### 4.1.1.2. INT(N)是什么？

演示

```
create table test_int_n(a int(4) zerofill);
insert into test_int_n values(1);
insert into test_int_n values(123456);
```

- int(N)中的 N 是显示宽度，不表示 存储的数字的 长度 的上限。
- zerofill 表示当存储的数字 长度 < N 时，用 数字 0 填充左边，直至补满长度 N
- 当存储数字的长度 超过 N 时，按照 实际存储 的数字显示

#### 4.1.1.3. 自动增长的面试题

这列语法有错误吗？

```
create table test_auto_increment(a int auto_increment);
```

```
create table test_auto_increment(a int auto_increment primary key);
```

以下结果是什么？

```
insert into test_auto_increment values(NULL);
insert into test_auto_increment values(0);
insert into test_auto_increment values(-1);
insert into test_auto_increment values(null),(100),(null),(10),(null)
```

#### 4.1.2. 字符类型

类型	说明	N 的含义	是否有字符集	最大长度
CHAR(N)	定长字符	字符	是	255

VARCHAR(N)	变长字符	字符	是	16384
BINARY(N)	定长二进制字节	字节	否	255
VARBINARY(N)	变长二进制字节	字节	否	16384
TINYBLOB(N)	二进制大对象	字节	否	256
BLOB(N)	二进制大对象	字节	否	16K
MEDIUMBLOB(N)	二进制大对象	字节	否	16M
LONGBLOB(N)	二进制大对象	字节	否	4G
TINYTEXT(N)	大对象	字节	是	256
TEXT(N)	大对象	字节	是	16K
MEDIUMTEXT(N)	大对象	字节	是	16M
LONGTEXT(N)	大对象	字节	是	4G

#### 4.1.2.1. 排序规则

```
select 'a' = 'A';
create table test_ci (a varchar(10), key(a));
insert into test_ci values('a');
insert into test_ci values('A');

select * from test_ci where a = 'a'; --结果是什么?
set names utf8mb4 collate utf8mb4_bin
```

#### 4.1.3. 时间类型

日期类型	占用空间	表示范围
DATETIME	8	1000-01-01 00:00:00 ~ 9999-12-31 23:59:59
DATE	3	1000-01-01 ~ 9999-12-31
TIMESTAMP	4	1970-01-01 00:00:00UTC ~ 2038-01-19 03:14:07UTC
YEAR	1	YEAR(2):1970-2070, YEAR(4):1901-2155
TIME	3	-838:59:59 ~ 838:59:59

datetime 与 timestamp 区别

```
create table test_time(a timestamp, b datetime);
insert into test_time values (now(), now());
select * from test_time;
select @@time_zone;
set time_zone='+00:00';
select * from test_time;
```

## 4.1.4. JSON 类型

### 4.1.4.1. JSON 入门

新建表

```
create table json_user (
    uid int auto_increment,
    data json,
    primary key(uid)
);
```

插入数据

```
insert into json_user values (
    null, '{
        "name":"lison",
        "age":18,
        "address":"enjoy"
    }');
```

```
insert into json_user values (
    null,
    '{
        "name":"james",
        "age":28,
        "mail":"james@163.com"
    }');
```

### 4.1.4.2. JSON 函数

#### 4.1.4.2.1. json\_extract 抽取

```
select json_extract('[10, 20, [30, 40]]', '[1]');
mysql> select json_extract('[10, 20, [30, 40]]', '[1]');
+-----+
| json_extract('[10, 20, [30, 40]]', '[1]') |
+-----+
```

```
select
    json_extract(data, '$.name'),
    json_extract(data, '$.address')
```

```

from json_user;

mysql> select
->   json_extract(data, '$.name'),
->   json_extract(data, '$.address')
->   from json_user;
+-----+-----+
| json_extract(data, '$.name') | json_extract(data, '$.address') |
+-----+-----+
| "lison"                   | "enjoy"                    |
| "james"                   | NULL                       |
+-----+-----+

```

#### 4.1.4.2.2. JSON\_OBJECT 将对象转为 json

```
select json_object("name", "enjoy", "email", "enjoy.com", "age", 35);
```

```
insert into json_user values (
null,
json_object("name", "enjoy", "email", "enjoy.com", "age", 35));
```

```

mysql> select * from json_user;
+----+-----+
| uid | data           |
+----+-----+
| 1  | {"age": 18, "name": "lison", "address": "enjoy"} |
| 2  | {"age": 28, "mail": "james@163.com", "name": "james"} |
| 3  | {"age": 35, "name": "enjoy", "email": "enjoy.com"}  |
+----+-----+

```

#### 4.1.4.2.3. json\_insert 插入数据

语法: JSON\_INSERT(json\_doc, path, val[, path, val] ...)

```
set @json = '{"a": 1, "b": [2, 3]}';
select json_insert(@json, '$.a', 10, '$.c', [true, false]);
```

```
update json_user set data = json_insert(data, "$.address_2", "xiangxue") where uid = 1;
```

```

mysql> select * from json_user;
+----+-----+
| uid | data           |
+----+-----+
| 1  | {"age": 18, "name": "lison", "address": "enjoy", "address_2": "xiangxue"} |
| 2  | {"age": 28, "mail": "james@163.com", "name": "james"}                      |
| 3  | {"age": 35, "name": "enjoy", "email": "enjoy.com"}                         |
+----+-----+
3 rows in set (0.00 sec)

```

#### 4.1.4.2.4. json\_merge 合并数据并返回

```
select json_merge('{"name": "enjoy"}', '{"id": 47}');
```

```
select
json_merge(
json_extract(data, '$.address'),
json_extract(data, '$.address_2'))
from json_user where uid = 1;
```

```
mysql> select
-> json_merge(
->   json_extract(data, '$.address'),
->   json_extract(data, '$.address_2'))
->   from json_user where uid = 1;
+-----+
| json_merge(
|   json_extract(data, '$.address'),
|   json_extract(data, '$.address_2')) |
+-----+
| ["enjoy", "xiangxue"] |
+-----+
1 row in set (0.00 sec)
```

#### 4.1.4.2.5. 其他函数:

<https://dev.mysql.com/doc/refman/5.7/en/json-function-reference.html>

### 4.1.5. JSON 索引

JSON 类型数据本身 无法直接 创建索引，需要将需要索引的 JSON 数据 重新 生成虚拟列 (Virtual Columns) 之后，对 该列 进行 索引

```
create table test_inex_1(
  data json,
  gen_col varchar(10) generated always as (json_extract(data, '$.name')),
  index idx (gen_col)
);
```

```
insert into test_inex_1(data) values ('{"name":"king", "age":18, "address":"cs"}');
insert into test_inex_1(data) values ('{"name":"peter", "age":28, "address":"zz"}');
```

```
select * from test_inex_1;
```

```

mysql> select * from test_inex_1;
+-----+-----+
| data | gen_col |
+-----+-----+
| {"age": 18, "name": "king", "address": "cs"} | "king" |
| {"age": 28, "name": "peter", "address": "zz"} | "peter" |
+-----+-----+

```

疑问：这条 sql 查询的结果是？

```
select json_extract(data,"$.name") as username from test_inex_1 where gen_col="king";
```

```
select json_extract(data,"$.name") as username from test_inex_1 where gen_col=="king";
```

```

mysql> select json_extract(data,"$.name") as username from test_inex_1 where gen_col=='king';
+-----+
| username |
+-----+
| "king"   |
+-----+
1 row in set (0.00 sec)

```

```
explain select json_extract(data,"$.name") as username from test_index_1 where gen_col="king"
```

```

mysql> explain select json_extract(data,"$.name") as username from test_index_1 where gen_col="king"
->;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE    | test_index_1 | NULL      | ref  | idx            | idx | 33     | const | 1    | 100.00 | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

```

## 4.1.6. 作业：

作业：自己查阅官方文档，建立虚拟列，这个列查询的时候不需要加上“” 符号

```

create table test_index_2 (
  data json,
  gen_col varchar(10) generated always as (
    json_unquote(
      json_extract(data, "$.name")
    )),
  key idx(gen_col)
);

```

```

insert into test_index_2(data) values ('{"name":"king", "age":18, "address":"cs"}');
insert into test_index_2(data) values ('{"name":"peter", "age":28, "address":"zz"}');

```

```
select json_extract(data,"$.name") as username from test_index_2 where gen_col="king";
```

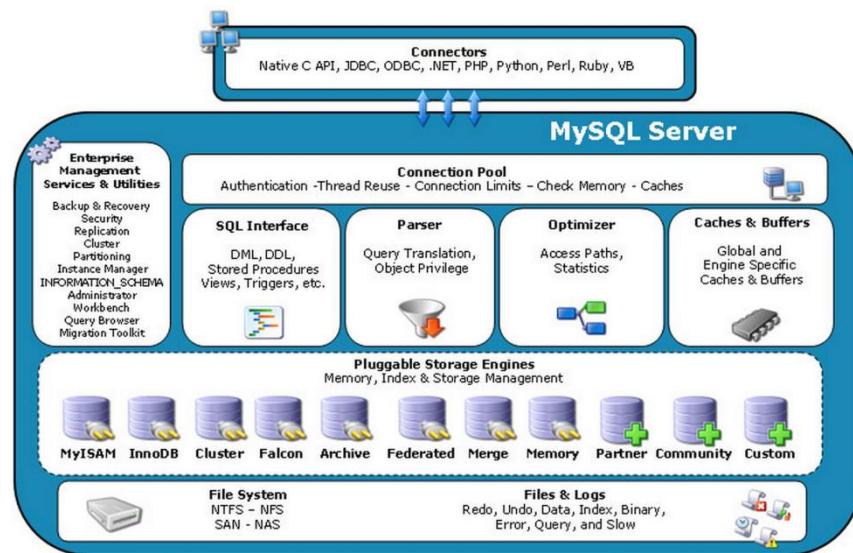
```

mysql> select json_extract(data,"$.name") as username from test_index_2 where gen_col="king";
+-----+
| username |
+-----+
| "king"   |
+-----+
1 row in set (0.01 sec)

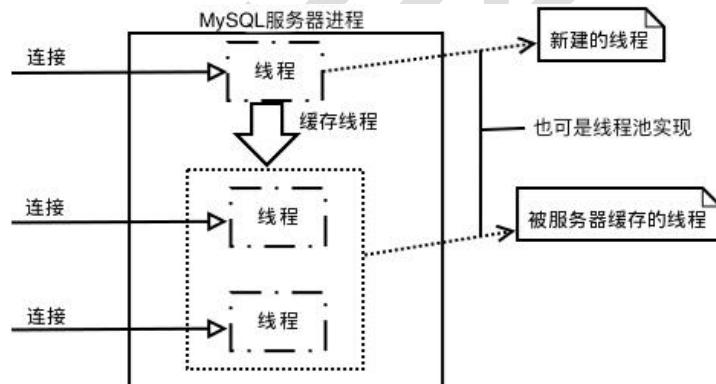
```

# 5. MySQL 架构

## 5.1. 体系

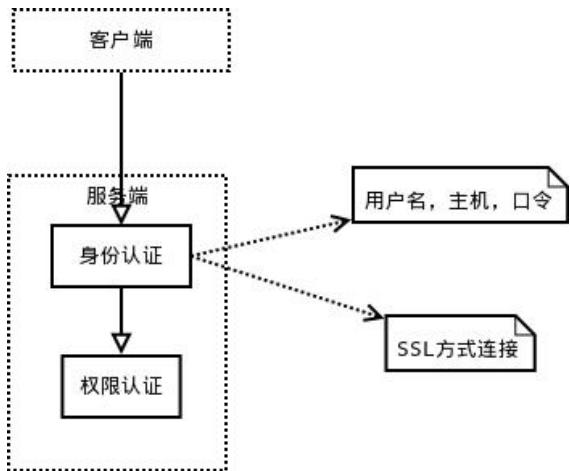


### 5.1.1. 连接层



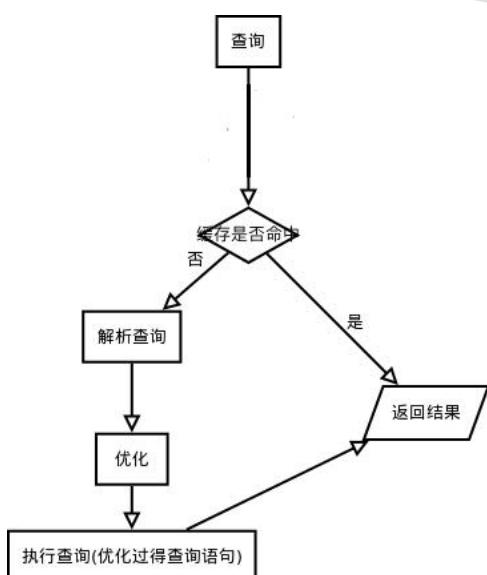
当 MySQL 启动（MySQL 服务器就是一个进程），等待客户端连接，每一个客户端连接请求，服务器都会新建一个线程处理（如果是线程池的话，则是分配一个空的线程），每个线程独立，拥有各自的内存处理空间。

```
show VARIABLES like '%max_connections%'
```



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）

### 5.1.2. SQL 处理层



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。

### 5.1.2.1. 缓存

```
show variables like '%query_cache_type%' -- 默认不开启  
show variables like '%query_cache_size%' --默认值 1M
```

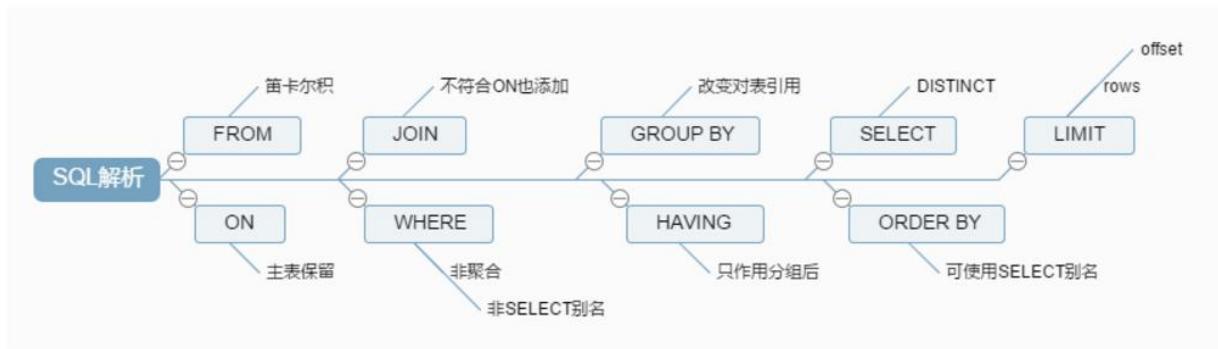
SET GLOBAL query\_cache\_type = 1; --会报错  
query\_cache\_type 只能配置在 my.cnf 文件中，这大大限制了 qc 的作用

在生产环境建议不开启，除非经常有 sql 完全一模一样的查询

QC 严格要求 2 次 SQL 请求要完全一样，包括 SQL 语句，连接的数据库、协议版本、字符集等因素都会影响

### 5.1.2.2. 解析查询

```
SELECT DISTINCT  
    < select_list >  
FROM  
    < left_table > < join_type >  
JOIN < right_table > ON < join_condition >  
WHERE  
    < where_condition >  
GROUP BY  
    < group_by_list >  
HAVING  
    < having_condition >  
ORDER BY  
    < order_by_condition >  
LIMIT < limit_number >
```



### 5.1.2.3. 优化



1 EXPLAIN

2 select \* from account where name = '1'

信息 结果1 概况 状态											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows		
1	SIMPLE	account	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	(Null)		4

1 EXPLAIN

2 select \* from account where 1=1

信息 结果1 概况 状态											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows		
1	SIMPLE	account	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	4		filter

1 EXPLAIN

2 select \* from account where id is null

信息 结果1 概况 状态											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows		
1	SIMPLE	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)		filter

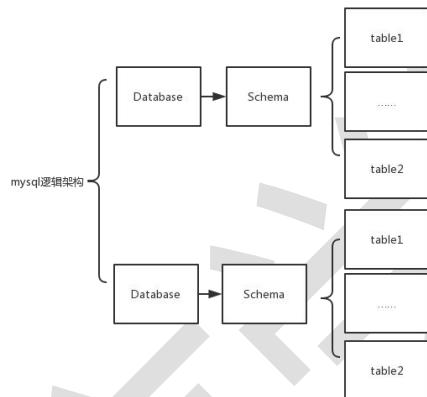
通过上面的 sql 大概就能看出，虽然现在还没学执行计划，但通过这个已经看出一个 sql 并不一定会去查询物理数据，sql 解析器会通过优化器来优化程序员写的 sql

```
explain
```

```
select * from account t where t.id in (select t2.id from account t2)  
show warnings;
```

```
mysql> explain select * from account t where t.id in (select t2.id from account t2)  
+-----+  
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra  
+-----+  
| 1 | SIMPLE | t2 | NULL | index | PRIMARY | idx_balance | 5 | NULL | NULL | 4 | 100.00 | Using index  
| 1 | SIMPLE | t | NULL | ALL | PRIMARY | NULL | NULL | NULL | 4 | 25.00 | Using where; using join buffer (Block Nested Loop)  
+-----+  
2 rows in set, 1 warning (0.00 sec)  
mysql> show warnings;  
+-----+  
| Level | Code | Message  
+-----+  
| Note | 1003 | /* select#1 */ select `mall`.`t`.`id` AS `id`, `mall`.`t`.`name` AS `name`, `mall`.`t`.`balance` AS `balance` from `mall`.`account` `t2` [join] `mall`.`account` `t` where (`mall`.`t`.`id` = `mall`.`t2`.`id`)  
+-----+  
1 row in set (0.00 sec)
```

## 5.2. 逻辑架构



在 mysql 中其实还有个 schema 的概念，这概念没什么太多作用，只是为了兼容其他数据库，所以也提出了这个。

在 mysql 中 database 和 schema 是等价的

```
create database demo;  
show databases;  
drop schema demo;  
show databases;
```

```

mysql> create database demo;
Query OK, 1 row affected (0.02 sec)

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| demo |
| mall |
| mysql |
| performance_schema |
| sys |
+-----+
6 rows in set (0.00 sec)

mysql> drop schema demo;
Query OK, 0 rows affected (0.09 sec)

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mall |
| mysql |
| performance_schema |
| sys |
+-----+
5 rows in set (0.00 sec)

```

## 5.3. 物理存储结构

### 5.3.1. 数据库的数据库 (DataDir)

mysql 安装的时候都要指定 datadir, 其查看方式为:  
`show VARIABLES like 'datadir'`, 其规定所有建立的数据库存放位置

```

mysql> show VARIABLES like 'datadir';
+-----+
| variable_name | value |
+-----+
| datadir | /usr/local/mysql/data/ |
+-----+
1 row in set (0.02 sec)

```

### 5.3.2. 数据库

创建了一个数据库后, 会在上面的 datadir 目录新建一个子文件夹

```

[root@mysql ~]# ll /usr/local/mysql/data/
total 188520
-rw-r--r--. 1 mysql mysql      56 Jun 11 11:09 auto.cnf
-rw-r--r--. 1 mysql root    1675 Jun 11 11:09 ca-key.pem
-rw-r--r--. 1 mysql root    1070 Jun 11 11:09 ca.pem
-rw-r--r--. 1 mysql root    1078 Jun 11 11:09 client-cert.pem
-rw-r--r--. 1 mysql root    1675 Jun 11 11:09 client-key.pem
-rw-r--r--. 1 mysql mysql     575 Jun 11 13:22 ib_buffer_pool
-rw-r--r--. 1 mysql mysql 79691776 Jun 11 16:51 ibdata1
-rw-r--r--. 1 mysql mysql 50331648 Jun 11 16:51 ib_logfile0
-rw-r--r--. 1 mysql mysql 50331648 Jun 11 11:09 ib_logfile1
-rw-r--r--. 1 mysql mysql 12582912 Jun 11 18:25 ibtmp1
drwxr-x---. 2 mysql mysql   4096 Jun 11 16:50 mall
drwxr-x---. 2 mysql mysql   4096 Jun 11 11:09 mysql
-rw-rw----. 1 root  root      6 Jun 11 13:22 mysqld_safe.pid
-rw-r-----. 1 mysql  root    20272 Jun 11 17:41 mysql.err
-rw-r-----. 1 mysql mysql      6 Jun 11 13:22 mysql.pid
drwxr-x---. 2 mysql mysql   8192 Jun 11 11:09 performance_schema
-rw-r--r--. 1 mysql  root    1679 Jun 11 11:09 private_key.pem
-rw-r--r--. 1 mysql  root     451 Jun 11 11:09 public_key.pem
-rw-r--r--. 1 mysql  root    1078 Jun 11 11:09 server-cert.pem
-rw-r-----. 1 mysql  root    1675 Jun 11 11:09 server-key.pem
drwxr-x---. 2 mysql mysql   8192 Jun 11 11:09 sys

```

### 5.3.3. 表文件

```
[root@mysql ~]# ll /usr/local/mysql/data/mall/
total 1024
-rw-r-----. 1 mysql mysql 8622 Jun 11 13:34 account.frm
-rw-r-----. 1 mysql mysql 114688 Jun 11 13:34 account.ibd
-rw-r-----. 1 mysql mysql 61 Jun 11 13:29 db.opt
-rw-r-----. 1 mysql mysql 8588 Jun 11 15:53 json_user.frm
-rw-r-----. 1 mysql mysql 98304 Jun 11 16:17 json_user.ibd
-rw-r-----. 1 mysql mysql 8554 Jun 11 14:30 test_auto_increment.frm
-rw-r-----. 1 mysql mysql 98304 Jun 11 14:37 test_auto_increment.ibd
-rw-r-----. 1 mysql mysql 8554 Jun 11 15:00 test_char.frm
-rw-r-----. 1 mysql mysql 98304 Jun 11 15:02 test_char.ibd
-rw-r-----. 1 mysql mysql 8649 Jun 11 16:50 test_index_2.frm
-rw-r-----. 1 mysql mysql 114688 Jun 11 16:51 test_index_2.ibd
-rw-r-----. 1 mysql mysql 8628 Jun 11 16:31 test_inex_1.frm
-rw-r-----. 1 mysql mysql 114688 Jun 11 16:34 test_inex_1.ibd
-rw-r-----. 1 mysql mysql 8554 Jun 11 14:18 test_int_n.frm
-rw-r-----. 1 mysql mysql 98304 Jun 11 14:18 test_int_n.ibd
-rw-r-----. 1 mysql mysql 8578 Jun 11 15:28 test_time.frm
-rw-r-----. 1 mysql mysql 98304 Jun 11 15:28 test_time.ibd
-rw-r-----. 1 mysql mysql 8578 Jun 11 14:09 test_unsigned.frm
-rw-r-----. 1 mysql mysql 98304 Jun 11 14:09 test_unsigned.ibd
[root@mysql ~]# ■
```

用户建立的表都会在上面的目录中，它和具体的存储引擎相关，但有个共同的就是都有个 `frm` 文件，它存放的是表的数据格式。

```
mysqlfrm --diagnostic /usr/local/mysql/data/mall/account.frm
```

```
[root@mysql mysql-utilities-1.6.5]# mysqlfrm --diagnostic /usr/local/mysql/data/mall/account.frm
# WARNING: Cannot generate character set or collation names without the -server option.
# CAUTION: The diagnostic mode is a best-effort parse of the .frm file. As such, it may not identify all of the
# default values for the columns and the resulting statement may not be syntactically correct.
# Reading .frm file for /usr/local/mysql/data/mall/account.frm:
# The .frm file is a TABLE.
# CREATE TABLE Statement:

CREATE TABLE `mall`.`account` (
  `id` int(11) NOT NULL,
  `name` varchar(150) DEFAULT NULL,
  `balance` int(255) DEFAULT NULL,
  PRIMARY KEY `PRIMARY` (`id`),
  KEY `idx_balance` (`balance`)
) ENGINE=InnoDB;
```

### 5.3.4. mysql utilities 安装

```
tar -zvxf mysql-utilities-1.6.5.tar.gz
cd mysql-utilities-1.6.5
python ./setup.py build
python ./setup.py install
```

## 6. 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%';
```

## 6.1. MyISAM

MySql 5.5 之前默认的存储引擎

MyISAM 存储引擎由 MYD 和 MYI 组成

```
create table testmysam (
    id int PRIMARY key
) ENGINE=myisam
insert into testmysam  VALUES(1),(2),(3)
```

### 6.1.1. 表压缩

```
myisampack -b -f /usr/local/mysql/data/mall/testmysam.MYI
```

压缩后再往表里面新增数据就新增不了

```
insert into testmysam  VALUES(1),(2),(3)
```

```
[SQL]insert into testmysam  VALUES(1),(2),(3)
```

```
[Err] 1036 - Table 'testmysam' is read only
```

压缩后，需要

```
myisamchk -r -f /usr/local/mysql/data/mall/testmysam.MYI
```

### 6.1.2. 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）

由于现在 innodb 越来越强大， myisam 已经停止维护  
**(绝大多数场景都不适合)**

## 6.2. Innodb

- 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 <b>不适合高并发的操作</b>	行锁,操作时只锁某一行，不对其它行有影响 <b>适合高并发的操作</b>
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

show VARIABLES like 'innodb\_log\_buffer\_size'

## 6.3. CSV

- 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;
insert into mycsv values(1,'aaa','bbb'),(2,'cccc','ddddd');
vi /usr/local/mysql/data/mall/mycsv.CSV 修改文本数据
flush TABLES;
select * from mycsv
create index idx_id on mycsv(id)
```

## 6.4. 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
```

## 6.5. Memory

### 6.5.1. 特点

- 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'

```
create table mymemory(id int,c1 varchar(10),c2 char(10),c3 text) engine = memory;
create table mymemory(id int,c1 varchar(10),c2 char(10)) engine = memory;
create index idx_c1 on mymemory(c1);
create index idx_c2 using btree on mymemory(c2);
show index from mymemory
show TABLE status LIKE 'mymemory'
```

### 6.5.2. 与临时表的区别

## Memory存储引擎表

VS

临时表

- 系统使用临时表 -

## 超过限制使用Myisam临时表

## 未超限制使用Memory表

- create temporary table 建立的临时表

### 6.5.3. 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

## 6.6. Federated

- 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（某些游戏行业）

默认禁止，启用需要再启动时增加 federated 参数

```
[root@mysql ~]# vi /etc/my.cnf
[mysqld]
sql_mode = "STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION,NO_ZERO_DATE,NO_ZERO_IN_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER"
federated
```

```
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
```

## 7. 锁

### 7.1. 锁的简介

#### 7.1.1. 为什么需要锁？

到淘宝上买一件商品，商品只有一件库存，这个时候如果还有另一个人买，那么如何解决是你买到还是另一个人买到的问题？

#### 7.1.2. 锁的概念

- 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。

#### 7.1.3. MySQL 中的锁

- 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(gap 锁,间隙锁)：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。

在这个部分只讲表级锁、行级锁，gap 锁放到事务中讲

## 7.1.4. 表锁与行锁的使用场景

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

## 7.2. MyISAM 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）

请求锁模式	None	读锁	写锁
是否兼容	None	是	否
当前锁模式	读锁	是	否
写锁	是	否	否

### 7.2.1. 共享读锁

语法：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. 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 account value(4,'aa',123); 报错

select \* from account ; 报错

5. 在另外一个 session 中

insert into account value(4,'aa',123); 成功

6. 加索在同一个 session 中 select s.\* from testmysam s 报错

lock table 表名 as 别名 read;

## 7.2.2. 独占写锁

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 account value(4,'aa',123);
```

3. 在其他 session 中（等待）

```
select * from testmysam
```

## 7.2.3. 总结：

- 读锁，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
- 读锁，对 MyISAM 表的读操作，不会阻塞当前 session 对表读，当对表进行修改会报错
- 读锁，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；
- 写锁，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；
- 写锁，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

## 7.3. InnoDB 锁

在 mysql 的 InnoDB 引擎支持行锁

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。

### 7.3.1. 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;

### 7.3.2. 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。

```
CREATE TABLE testdemo (
`id` int(255) NOT NULL ,
`c1` varchar(300) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL ,
`c2` int(50) NULL DEFAULT NULL ,
PRIMARY KEY (`id`),
INDEX `idx_c2`(`c2`) USING BTREE
)
ENGINE=InnoDB;

insert into testdemo VALUES(1,'1',1),(2,'2',2);
```

1.

BEGIN

select \* from testdemo where id =1 for update

在另外一个 session 中

update testdemo set c1 = '1' where id = 2 成功

update testdemo set c1 = '1' where id = 1 等待

2.BEGIN

update testdemo set c1 = '1' where id = 1

在另外一个 session 中

update testdemo set c1 = '1' where id = 1 等待

3.

```
BEGIN  
update testdemo set c1 = '1' where c1 = '1'
```

在另外一个 session 中

```
update testdemo set c1 = '2' where c1 = '2' 等待
```

4. 第一个 session 中

```
select * from testdemo where id = 1 for update
```

第二个 session

```
select * from testdemo where id = 1 lock in share mode
```

回到第一个 session UNLOCK TABLES 并不会解锁  
使用 commit 或者 begin 或者 ROLLBACK 才会解锁

5. 再来看下表锁

```
lock table testdemo WRITE
```

使用 commit, ROLLBACK 并不会解锁

使用 UNLOCK TABLES 或者 begin 会解锁

## 7.4. 锁的等待问题

好了，掌握了上面这些，你对锁的了解已经超过了很多人，那么现在来说一个实际的问题，在工作中经常一个数据被锁住，导致另外的操作完全进行不下去。

你肯定碰到过这问题，有些程序员在 debug 程序的时候，经常会锁住一部分数据库的数据，而这个时候你也要调试这部分功能，却发现代码总是运行超时，你是否碰到过这问题了，其实这问题的根源我相信你也知道了。

举例来说，有两个会话。

程序员甲，正直调试代码

```
BEGIN  
SELECT * FROM testdemo WHERE id = 1 FOR UPDATE
```

你正直完成的功能也要经过那部分的代码，你得上个读锁

```
BEGIN  
SELECT * FROM testdemo WHERE id = 1 lock in share mode
```

这个时候很不幸，你并不知道发生了什么问题，在你调试得过程中永远就是一个超时得异常，而这种问题不管在开发中还是在实际项目运行中都可能会碰到，那么怎么排查这个问题呢？这其实也是有小技巧的。

```
select * from information_schema.INNODB_LOCKS;
```

信息	结果1	概况	状态						
lock_id	lock trx_id	lock mode	lock type	lock_table	lock_index	lock_space	lock_page	lock_rec	lock_data
42161429624216142962301S		X	RECORD	'mall`.`testdemo'	PRIMARY	58	3	2	1
4374:58:3:2 4374			RECORD	'mall`.`testdemo'	PRIMARY	58	3	2	1

真好，我通过这个 sql 语句起码发现在同一张表里面得同一个数据有了 2 个锁其中一个是 X (写锁)，另外一个是 S (读锁)，我可以跳过这一条数据，使用其他数据做调试

可能如果我就是绕不过，一定就是要用这条数据呢？吼一嗓子吧（哪个缺德的在 debug 这个表，请不要锁这不动），好吧，这是个玩笑，其实还有更好的方式来看

```
select * from sys.innodb_lock_waits
```

1	select * from sys.innodb lock waits
信息 结果1 概况 状态	
locked_type	
locked_type	RECORD
waiting_trx_id	421614296230176
waiting_trx_started	2019-06-14 14:23:50
waiting_trx_age	00:00:31
waiting_trx_rows_lo...	1
waiting_trx_rows_m...	0
waiting_pid	23
waiting_query	SELECT * FROM testdemo WHERE id = 1 lock in share mode
waiting_lock_id	
waiting_lock_id	421614296230176:58:3:2
waiting_lock_mode	S
blocking_trx_id	4372
blocking_pid	16
blocking_query	
blocking_lock_id	
blocking_lock_id	4372:58:3:2
blocking_lock_mode	X
blocking_trx_started	2019-06-14 14:23:46
blocking_trx_age	00:00:35
blocking_trx_rows_lo...	1
blocking_trx_rows_m...	0
sql_kill_blocking_qu...	KILL QUERY 16
sql_kill_blocking_co...	KILL 16

我现在执行的这个 sql 语句有了，另外看下最下面，kill 命令，你在工作中完全可以通过 kill 吧阻塞了的 sql 语句给干掉，你就可以继续运行了，不过这命令也要注意使用过，如果某同事正在做比较重要的调试，你 kill，被他发现可能会被暴打一顿。

上面的解决方案不错，但如果你的 MySQL 不是 5.7 的版本呢？是 5.6 呢，你根本就没有 sys 库，这个时候就难办了，不过也是有办法的。

```
SELECT
```

```
r trx_id waiting trx_id,
```

```

r trx_mysql_thread_id waiting_thread,
r trx_query waiting_query,
b trx_id blocking_trx_id,
b trx_mysql_thread_id blocking_thread
FROM
information_schema.innodb_lock_waits w
INNER JOIN
information_schema.innodb trx b ON b.trx_id = w.blocking_trx_id
INNER JOIN
information_schema.innodb trx r ON r.trx_id = w.requesting_trx_id;

```

waiting_trx_id	waiting_thread	waiting_query	blocking_trx_id	blocking_thread
421614296230176	23	SELECT *FROM testdemo WHERE id = 14379		29

看到没有，接下来你是否也可以执行 kill 29 这样的大招了。

## 8. 事务

### 8.1. 为什么需要事务

现在的很多软件都是多用户，多程序，多线程的，对同一个表可能同时有很多人在用，为保持数据的一致性，所以提出了事务的概念。

A 给 B 要划钱，A 的账户 -1000 元，B 的账户就要 +1000 元，这两个 update 语句必须作为一个整体来执行，不然 A 扣钱了，B 没有加钱这种情况很难处理。

### 8.2. 什么存储引擎支持事务

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;

## 8.3. 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)
- 一致性 (consistency)
- 隔离性 (isolation)
- 持久性 (durability)

### 8.3.1. 原子性 (atomicity)

一个事务必须被视为一个不可分割的最小单元，整个事务中的所有操作要么全部提交成功，要么全部失败，对于一个事务来说，不可能只执行其中的一部分操作

老婆大人给 Deer 老师发生活费

- 1.老婆大人工资卡扣除 500 元
- 2.Deer 老师工资卡增加 500

整个事务要么全部成功，要么全部失败

### 8.3.2. 一致性 (consistency)

一致性是指事务将数据库从一种一致性转换到另外一种一致性状态，在事务开始之前和事务结束之后数据库中数据的完整性没有被破坏

老婆大人给 Deer 老师发生活费

- 1.老婆大人工资卡扣除 500 元
- 2.Deer 老师工资卡增加 500
- 2.Deer 老师工资卡增加 1000

扣除的钱 (-500) 与增加的钱 (500) 相加应该为 0

### 8.3.3. 持久性 (durability)

一旦事务提交，则其所做的修改就会永久保存到数据库中。此时即使系统崩溃，已经提交的修改数据也不会丢失

并不是数据库的角度完全能解决

### 8.3.4. 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。

(对数据库的并行执行，应该像串行执行一样)

- 未提交读 (READ UNCOMMITTED) 脏读
- 已提交读 (READ COMMITTED) 不可重复读
- 可重复读 (REPEATABLE READ)
- 可串行化 (SERIALIZABLE)

mysql 默认的事务隔离级别为 repeatable-read

show variables like '%tx\_isolation%';

#### 8.3.4.1. 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

### 8.3.4.2. 未提交读 (READ UNCOMMITTED) 脏读

```
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 中

```
select * from account
```

在另外一个 session 中读取到了未提交的数据，这部分的数据为脏数据

### 8.3.4.3. 已提交读 (READ COMMITTED) 不可重复读

```
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 回滚事务

```
commit
```

在第二个 session 中

```
select * from account (数据已经改变)
```

#### 8.3.4.4. 可重复读 (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 中

```
select * from account (数据并未改变)
```

#### 8.3.4.5. 可串行化 (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) 发现根本就不让插入
```

3. 回到第一个事务 commit

### 8.3.4.6. 间隙锁 (gap 锁)

其实在 mysql 中，可重复读已经解决了幻读问题，借助的就是间隙锁

	Session A	Session B
time	SET autocommit=0;	SET autocommit=0;
	SELECT * FROM t; empty set	INSERT INTO t VALUES (1, 2);
v	SELECT * FROM t; empty set	COMMIT;
	SELECT * FROM t; empty set	RR级别
	COMMIT;	假如事务A，在这个时候进行了对事务B插入的数据进行修改，依然会是empty set吗？
	SELECT * FROM t;	
	-----   1   2   -----	

#### 实验 1:

```
select @@tx_isolation;
create table t_lock_1 (a int primary key);
insert into t_lock_1 values(10),(11),(13),(20),(40);

begin
select * from t_lock_1 where a <= 13 for update;
```

在另外一个会话中

```
insert into t_lock_1 values(21) 成功
insert into t_lock_1 values(19) 阻塞
```

在 rr 隔离级别中者个会扫描到当前值（13）的下一个值（20），并把这些数据全部加锁

#### 实验：2

```
create table t_lock_2 (a int primary key,b int, key (b));
insert into t_lock_2 values(1,1),(3,1),(5,3),(8,6),(10,8);
```

会话 1

```
BEGIN
```

```
select * from t_lock_2 where b=3 for update;
```

1 3 5 8 10

1 1 3 6 8

会话 2

```
select * from t_lock_2 where a = 5 lock in share mode; -- 不可执行, 因为 a=5 上有一把记录锁
insert into t_lock_2 values(4, 2); -- 不可以执行, 因为 b=2 在(1, 3]内
insert into t_lock_2 values(6, 5); -- 不可以执行, 因为 b=5 在(3, 6)内
insert into t_lock_2 values(2, 0); -- 可以执行, (2, 0)均不在锁住的范围内
insert into t_lock_2 values(6, 7); -- 可以执行, (6, 7)均不在锁住的范围内
insert into t_lock_2 values(9, 6); -- 可以执行
insert into t_lock_2 values(7, 6); -- 不可以执行
```

二级索引锁住的范围是 (1, 3], (3, 6)

主键索引只锁住了 a=5 的这条记录 [5]

## 8.4. 事务语法

### 8.4.1. 开启事务

- 1、begin
- 2、START TRANSACTION (推荐)
- 3、begin work

### 8.4.2. 事务回滚

rollback

### 8.4.3. 事务提交

```
commit
```

### 8.4.4. 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
```

## 9. 业务设计

### 9.1. 逻辑设计

#### 9.1.1. 范式设计

##### 9.1.1.1. 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

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

<b>id</b>	<b>name-age</b>
1	张三-23

name-age 列具有两个属性，一个 name,一个 age 不符合第一范式，把它拆分成两列

<b>id</b>	<b>name</b>	<b>age</b>
1	张三	23

### 9.1.1.2. 数据库设计的第二大范式

要求表中只具有一个业务主键，也就是说符合第二范式的表不能存在非主键列只对部分主键的依赖关系

有两张表：订单表，产品表

<b>订单表ID (主键)</b>	<b>订单时间</b>	<b>产品ID</b>
1	2018-12-12	3
1	2018-12-12	4

<b>产品表ID</b>	<b>产品名称</b>
2	娃娃
3	飞机
4	java入门

一个订单有多个产品，所以订单的主键为【订单 ID】和【产品 ID】组成的联合主键，这样 2 个组件不符合第二范式，而且产品 ID 和订单 ID 没有强关联，故，把订单表进行拆分为订单表与订单与商品的中间表

订单表ID	订单时间
1	2018-12-12
订单-商品中间表ID	订单ID
1	3
2	4
产品表ID	产品名称
2	娃娃
3	飞机
4	java入门

### 9.1.1.3. 数据库设计的第三范式

指每一个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本上消除了非主键对主键的传递依赖

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

其中

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

如果客户编号发生改变，用户姓名也会改变，这样不符合第三范式，应该把客户姓名这一列删除

### 9.1.1.4. 范式设计实战

按要求设计一个电子商务网站的数据库结构

1. 本网站只销售图书类产品

2. 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

#### 9.1.1.4.1. 用户登陆及用户管理

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

只有一个业务主键，一定是符合第二范式

没有属性和业务主键存在传递依赖的关系，符合第三范式

#### 9.1.1.4.2. 商品信息

ID	商品名称	分类名称	出版社名称	图书价格	图书表述	作者
----	------	------	-------	------	------	----

一个商品可以属于多个分类，故，商品名称和分类应该是组合主键，会有大量冗余，不符合第二范式。应该把分类信息单独存放

分类信息

分类名称	分类描述
------	------

另外再建立一个中间表把分类信息和商品信息进行关联

商品分类对应关系表

商品名称	分类名称
------	------

最后的三张表如下

商品信息

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

#### 9.1.1.4.3. 供应商管理功能

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

符合三大范式，不需要修改，但假如增加新的一列【银行支行】，这样随着银行账户的变化，银行支行也会编号，不符合第三大范式

出版社名称	地址	电话	联系人	银行账号	银行支行
-------	----	----	-----	------	------

#### 9.1.1.4.4. 在线销售功能

订单 编号 <b>pk</b>	下单 用户名	下单 日期	订单 金额	订单 商品 分类	订单 商品 名 <b>pk</b>	订单 商品 单价	订单 商品 数量	支付 金额	物流 单号
-----------------------	-----------	----------	----------	----------------	-------------------------	----------------	----------------	----------	----------

有多个业务主键，不符合第二范式

订单商品单价。订单数量，订单金额 存在传递依赖关系，不符合第三范式

拆分的结果如下

##### □ 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
------	--------	-------	--------

这时候，【订单商品分类】与【订单商品名】有依赖关联，故合并如下

订单编号	商品分类中间表ID	订单商品数量
------	-----------	--------

#### 9.1.1.4.5. 表汇总



#### 9.1.1.4.6. 查询练习

编写 SQL 查询出每一个用户的订单总金额（用户名，订单总金额）

SELECT

a.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（订单编号，用户名，手机号，商品名称，商品数量，

商品价格)

```
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
```

问题：

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的 SQL 查询性能

## 9.1.2. 反范式设计

### 9.1.2.1. 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

#### 9.1.2.1.1. 商品信息反范式设计

下面是范式设计的商品信息表

商品信息

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

商品信息和分类信息经常一起查询，所以把分类信息也放到商品表里面，冗余存放

### 9.1.2.1.2. 在线销售功能反范式

下面是在线手写功能的范式设计

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	商品分类ID	订单商品数量
------	--------	--------

首先来看订单表

1. 查询订单信息要关联查询到用户表，但用户表的电话是可能改变的，而且查询订单的时候经常查询到用户的电话
2. 查询订单经常会查询到订单金额，所以把订单金额也冗余进来

新设计的订单表如下

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

再来看订单关联表

1. 和商品信息反范式设计一样，查询订单的时候经常查询商品分类，所以把商品分类和订

单名冗余进来

2.商品的单价可能会编号，如果关联查询查询只能查询到最新的商品价格，而查询不到下订单时候的价格，并且商品单价经常会查询。所以把订单单价也冗余进来

新设计的商品关联表如下

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

### 9.1.2.1.3. 查询练习

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
```

### 9.1.3. 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

### 9.1.3.1. 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

### 9.1.3.2. 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

## 9.2. 物理设计

### 9.2.1. 命名规范

#### 9.2.1.1. 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。

### 9.2.1.1.2. 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。

### 9.2.1.1.3. 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

## 9.2.2. 存储引擎选择

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 <b>不适合高并发的操作</b>	行锁,操作时只锁某一行， 不对其它行有影响 <b>适合高并发的操作</b>
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

## 9.2.3. 数据类型选择

当一个列可以选择多种数据类型时

- 优先考虑数字类型
- 其次是日期、时间类型
- 最后是字符类型
- 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

### 9.2.3.1.1. 浮点类型

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

注意 float 和 double 是非精度类型，如果是和金额相关尽量用 decimal

```
mysql> show create table test_numeric;
+-----+
| Table      | Create Table
+-----+
| test_numeric | CREATE TABLE `test_numeric` (
  `id` int(11) NOT NULL,
  `c1` float DEFAULT NULL,
  `c2` double DEFAULT NULL,
  `c3` decimal(5,5) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 |
+-----+
1 select sum(c1),sum(c2),sum(c3) from test_numeric
1  select sum(c1),sum(c2),sum(c3) from test_numeric
```

信息	结果1	概况	状态
	sum(c1) sum(c2) sum(c3)	18.661820650100708 18.661820000000006 18.66182	

### 9.2.3.1.2. 日期类型

面试经常问道 timestamp 类型 与 datetime 区别

类型	大小 (字节)	范围	格式	用途
DATETIME	8	1000-01-01 00:00:00/9999-12-31 23:59:59	YYYY-MM-DD HH:MM:SS	混合日期和时间值
TIMESTAMP	4	1970-01-01 00:00:00/2037 年某时	YYYYMMDD HHMMSS	混合日期和时间值， 时间戳

datetime 类型在 5.6 中字段长度是 5 个字节  
 datetime 类型在 5.5 中字段长度是 8 个字节

timestamp 和时区有关，而 datetime 无关

```
| test_time | CREATE TABLE `test_time` (
  `c1` datetime DEFAULT NULL,
  `c2` timestamp NULL DEFAULT NULL ON UPDATE CURRENT_TIMESTAMP,
  `c3` timestamp NULL DEFAULT NULL ON UPDATE CURRENT_TIMESTAMP
) ENGINE=InnoDB DEFAULT CHARSET=utf8 |
+-----+
```

```
insert into test_time VALUES(NOW(),NOW(),NOW());
```

```
set time_zone="-10:00"
```

```
1 select * from test_time
?
信息 结果1 概况 状态
c1 c2 c3
2018-08-27 22:52:08 2018-08-27 04:52:08 2018-08-27 04:52:08
```

## 10. 慢查询

### 10.1. 什么是慢查询

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。

## 10.2. 慢查询配置

### 10.2.1. 慢查询基本配置

- 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 执行时间得阀值（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

实操：

通过下面命令查看下上面的配置：

```
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。

```
cat /usr/local/mysql/data/mysql-slow.log
```

```

[root@mysql data]# cat /usr/local/mysql/data/mysql-slow.log
/usr/local/mysql/bin/mysqld, version: 5.7.9 (MySQL Community Server (GPL)). started with:
Tcp port: 3306  unix socket: /tmp/mysql.sock
Time           Id Command   Argument
# Time: 2019-06-11T17:52:49.138679Z
# User@Host: root[root] @ [192.168.244.1]  Id:      60
# Query_time: 0.000109  Lock_time: 0.000000 Rows_sent: 0  Rows_examined: 0
SET timestamp=1560275569;
SET NAMES utf8;
# Time: 2019-06-11T17:52:49.151830Z
# User@Host: root[root] @ [192.168.244.1]  Id:      60
# Query_time: 0.000088  Lock_time: 0.000000 Rows_sent: 0  Rows_examined: 0
use mail;
SET timestamp=1560275569;
# administrator command: Init DB;
# Time: 2019-06-11T17:52:49.152317Z
# User@Host: root[root] @ [192.168.244.1]  Id:      60
# Query_time: 0.000319  Lock_time: 0.000101 Rows_sent: 1  Rows_examined: 1
SET timestamp=1560275569;
SHOW TABLE STATUS LIKE 'account';
# Time: 2019-06-11T17:52:49.152853Z
# User@Host: root[root] @ [192.168.244.1]  Id:      60
# Query_time: 0.000119  Lock_time: 0.000030 Rows_sent: 9  Rows_examined: 9
SET timestamp=1560275569;
SHOW ENGINES;
# Time: 2019-06-11T17:52:49.235872Z
# User@Host: root[root] @ [192.168.244.1]  Id:      60
# Ouerv time: 0.000458  Lock time: 0.000180 Rows sent: 5  Rows examined: 5

```

## 10.2.2. 慢查询解读

从慢查询日志里面摘选一条慢查询日志，数据组成如下

```

# User@Host: root[root] @ localhost [127.0.0.1]  Id:      10
# Query_time: 0.001042  Lock_time: 0.000000 Rows_sent: 2  Rows_examined: 2
SET timestamp=1535462721;
SELECT * FROM `mvarchive` LIMIT 0, 1000;

```

把为解读放吧，慢查询格式显示

```

1 # User@Host: root[root] @ localhost [127.0.0.1]  Id:      10
2 # Query_time: 0.001042
3 #Lock_time: 0.000000
4 #Rows_sent: 2
5 #Rows_examined: 2
6 SET timestamp=1535462721;
7 SELECT * FROM `myarchive` LIMIT 0, 1000;

```

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

## 10.3. 慢查询分析

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

### 10.3.1. 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 -s t -t 10
```

```
/usr/local/mysql/data/mysql-slow.log
```

```
Reading mysql slow query log from /usr/local/mysql/data/mysql-slow.log  
Count: 1 Time=0.00s (0s) Lock=0.00s (0s) Rows=0.0 (0), 0users@0hosts  
Time: N-N-11T17:N:N.096168Z  
# User@Host: root[root] @ [N.N.N.N] Id: N  
# Query_time: N.N Lock_time: N.N Rows_sent: N Rows_examined: N  
SET timestamp=N;  
SHOW COLUMNS FROM `mall`.`json_user`  
  
Count: 1 Time=0.00s (0s) Lock=0.00s (0s) Rows=0.0 (0), 0users@0hosts  
Time: N-N-11T17:N:N.008385Z  
# User@Host: root[root] @ [N.N.N.N] Id: N  
# Query_time: N.N Lock_time: N.N Rows_sent: N Rows_examined: N  
SET timestamp=N  
  
Count: 1 Time=0.00s (0s) Lock=0.00s (0s) Rows=0.0 (0), 0users@0hosts  
Time: N-N-11T17:N:N.151830Z  
# User@Host: root[root] @ [N.N.N.N] Id: N  
# Query_time: N.N Lock_time: N.N Rows_sent: N Rows_examined: N  
use mall;  
SET timestamp=N  
  
Count: 1 Time=0.00s (0s) Lock=0.00s (0s) Rows=0.0 (0), 0users@0hosts  
Time: N-N-11T17:N:N.696084Z  
# User@Host: root[root] @ [N.N.N.N] Id: N  
# Query_time: N.N Lock_time: N.N Rows_sent: N Rows_examined: N  
SET timestamp=N;  
SHOW ENGINES
```

### 10.3.2. 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

```
Yum -y install 'perl(Data::Dumper)';
yum -y install perl-Digest-MD5
yum -y install perl-DBI
yum -y install perl-DBD-MySQL

perl ./pt-query-digest --explain h=127.0.0.1,u=root,p=root1234%
/usr/local/mysql/data/mysql-slow.log
```

```
PS F:\BaiduNetdiskDownload\vip> perl ./pt-query-digest --explain h=127.0.0.1,u=root,p=root1234% D:\DESKTOP-2EKGEE5-slow.log

# 562ms user time, 46ms system time, 0 rss, 0 vsz
# Current date: Wed Aug 29 15:45:17 2018
# Hostname: DESKTOP-2EKGEE5
# File: D:\DESKTOP-2EKGEE5-slow.log
# Overall: 211 total, 47 unique, 0.06 QPS, 0.00x concurrency -----
# Time range: 2018-08-28 21:25:20 to 22:22:30
# Attribute          total      min      max      avg      95%    stddev   median
# ======          ======      ======      ======      ======      ======    ======   ======
# Exec time         3s   930us   756ms   14ms   10ms    78ms    1ms
# Lock time        176ms      0     9ms   835us   4ms    2ms    0
# Rows sent       32.62k      0   1000   158.29  329.68  186.21   14.52
# Rows examine    8.04M      0   2.00M   39.02k  329.68  272.50k  284.79
# Query size      12.27k     11   3.85k   59.28  202.40  260.25   31.70

# Profile
# Rank Query ID          Response time Calls R/Call V/M   I
# ====== ======          ====== ====== ====== ====== ====== =====
# 1 0xBB1562F9BDE672C71A2CDA3D0DC... 1.0543 36.5%   2 0.5271 0.00 SELECT product_info?
# 2 0xDF207E1C87B653BA3383DBB0E3E... 0.7559 26.2%   1 0.7559 0.00 SELECT product_info?
# 3 0x75E664B8ED02ECE1CFD0DDCCBAC... 0.4876 16.9%   1 0.4876 0.00 SELECT product_info?
# 4 0x8085D806F3631D0D30FE5C20326... 0.1227  4.3%   19 0.0065 0.00 SHOW TABLE STATUS
# 5 0xF9734574CDDDC5D231DA25F9549... 0.1147  4.0%   87 0.0013 0.00 SHOW STATUS
# 6 0x1A9E0EABC27EAF756C885618316... 0.0428  1.5%   6 0.0071 0.00 SHOW COLUMNS
# 7 0x0F9F4FAA78CD9C57C23306E2280... 0.0421  1.5%   6 0.0070 0.00 SHOW COLUMNS
# 8 0x6D5D5A28192F9C46E2684D5FFF9... 0.0279  1.0%   4 0.0070 0.00 SHOW COLUMNS
# 9 0x72833D3A18ACDED00CF6DB6C18C... 0.0240  0.8%   2 0.0120 0.01 SHOW TABLES
# 10 0x9277183CCDC5C9143701A8733C... 0.0200  0.7%   19 0.0011 0.00 SELECT INFORMATION_SCHEMA.PROFILING
# 11 0xADBA315855A1F1C950249321C23... 0.0158  0.5%   11 0.0014 0.00 SELECT INFORMATION_SCHEMA.PROFILING
# 12 0xA39DD594FAE10E151F9C4ED1B9A... 0.0130  0.4%   2 0.0065 0.00 SHOW FUNCTION STATUS
# 13 0xA85D39F2512BD03E752EF056D51... 0.0130  0.4%   2 0.0065 0.00 SHOW TABLES
# 14 0xDA7D455C08B8803291E0792F9DE... 0.0129  0.4%   2 0.0065 0.00 SHOW COLUMNS
# MISC 0xMISC           0.1393  4.8%   47 0.0030 0.0 <32 ITEMS>
```

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

### 10.3.2.1. 扩展阅读：

#### 10.3.2.1.1. 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**、**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。

#### 10.3.2.1.2. 分析 pt-query-digest 输出结果

##### 10.3.2.1.2.1. 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
```

### 10.3.2.1.2.2. 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 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

```
# 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

```

### 10.3.2.1.2.3. 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    2s    0    2s
# Lock time   0    0    0    0    0    0    0    0    0
# Rows sent   20   1    1    1    1    1    1    0    1
# Rows examine 0    0    0    0    0    0    0    0    0
# Query size  3    15   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

```

# 11. 索引与执行计划

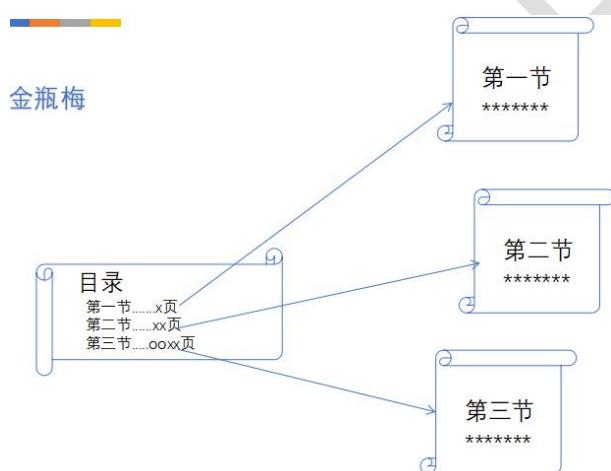
## 11.1. 索引入门

### 11.1.1. 索引是什么

#### 11.1.1.1. 生活中的索引

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**。

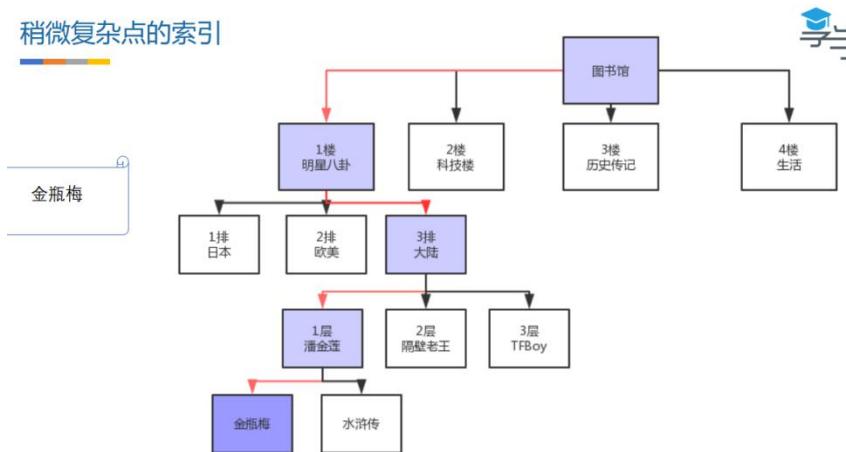
上面的理解比较抽象，举一个例子，平时看任何一本书，首先看到的都是目录，通过目录去查询书籍里面的内容会非常的迅速。



上图就是一本金瓶梅的书，书籍的目录是按顺序放置的，有第一节，第二节它本身就是一种顺序存放的数据结构，是一种顺序结构。

另外通过目录（索引），可以快速查询到目录里面的内容，它能高效获取数据，通过这个简单的案例可以理解所以就是**高效获取数据的数据结构**

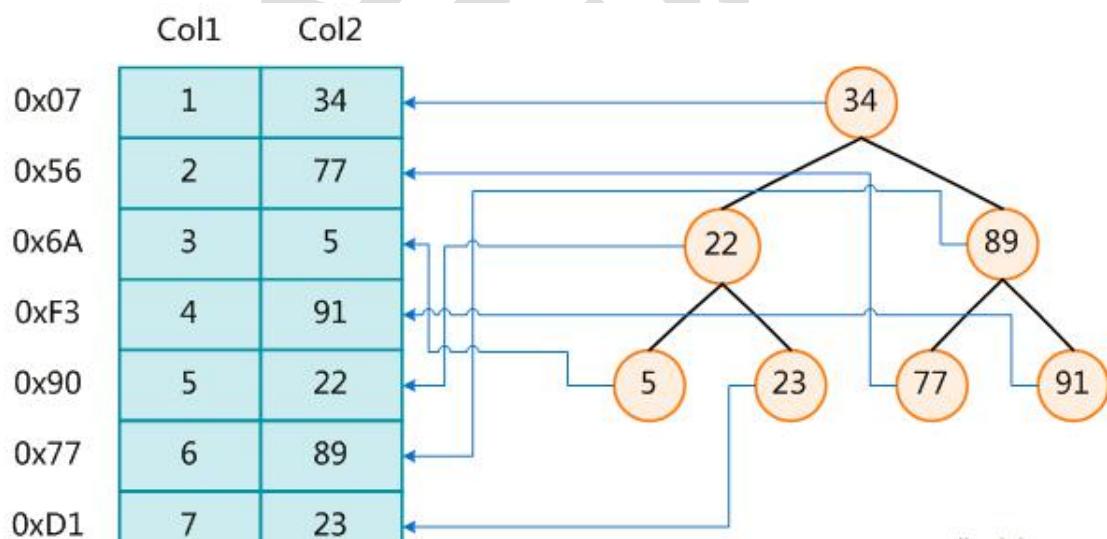
再来看一个发杂的情况，



我们要去图书馆找一本书，这图书馆的书肯定不是线性存放的，它对不同的书籍内容进行了分类存放，整索引由于一个个节点组成，根节点有中间节点，中间节点下面又由子节点，最后一层是叶子节点，

可见，整个索引结构是一棵倒挂着的树，其实它就是一种数据结构，这种数据结构比前面讲到的线性目录更好的增加了查询的速度。

### 11.1.1.2. MySql 中的索引



MySql 中的索引其实也是这么一回事，我们可以在数据库中建立一系列的索引，比如创建主键的时候默认会创建主键索引，上图是一种 BTREE 的索引。每一个节点都是主键的 ID

当我们通过 ID 来查询内容的时候，首先去查索引库，在到索引库后能快速的定位索引的具体位置。

### 11.1.1.3. 谈下 B+Tree

要谈 B+TREE 说白了还是 Tree, 但谈这些之前还要从基础开始讲起

#### 11.1.1.3.1. 二分查找

二分查找法 (binary search) 也称为折半查找法, 用来查找一组**有序**的记录数组中的某一记录。

其基本思想是: 将记录按有序化 (递增或递减) 排列, 在查找过程中采用跳跃式方式查找, 即先以有序数列的中点位置作为比较对象, 如果要找的元素值小于该中点元素, 则将待查序列缩小为左半部分, 否则为右半部分。通过一次比较, 将查找区间缩小一半。

# 给出一个例子, 注意该例子已经是升序排序的, 且查找 数字 48

数据: 5, 10, 19, 21, 31, 37, 42, 48, 50, 52

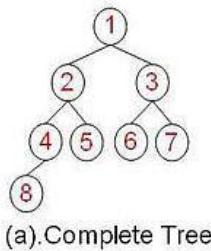
下标: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

- 步骤一: 设 `low` 为下标最小值 0 , `high` 为下标最大值 9;
- 步骤二: 通过 `low` 和 `high` 得到 `mid` , `mid=(low + high) / 2`, 初始时 `mid` 为下标 4 (也可以=5, 看具体算法实现);
- 步骤三 : `mid=4` 对应的数据值是 31,  $31 < 48$  (我们要找的数字) ;
- 步骤四: 通过二分查找的思路, 将 `low` 设置为 31 对应的下标 4 , `high` 保持不变为 9 , 此时 `mid` 为 6 ;
- 步骤五 : `mid=6` 对应的数据值是 42,  $42 < 48$  (我们要找的数字) ;
- 步骤六: 通过二分查找的思路, 将 `low` 设置为 42 对应的下标 6 , `high` 保持不变为 9 , 此时 `mid` 为 7 ;
- 步骤七 : `mid=7` 对应的数据值是 48,  $48 == 48$  (我们要找的数字) , 查找结束;  
通过 3 次 二分查找 就找到了我们所要的数字, 而顺序查找需 8

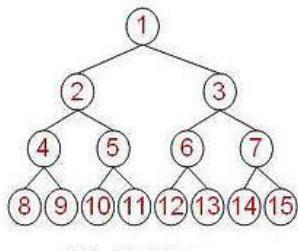
#### 11.1.1.3.2. 二叉树(Binary Tree)

每个节点至多只有二棵子树;

- 二叉树的子树有左右之分, 次序不能颠倒;
- 一棵深度为  $k$ , 且有  $2^k - 1$  个节点, 称为满二叉树(Full Tree);
- 一棵深度为  $k$ , 且 `root` 到  $k-1$  层的节点树都达到最大, 第  $k$  层的所有节点都 连续集中 在最左边, 此时为完全二叉树 (Complete Tree)



(a). Complete Tree

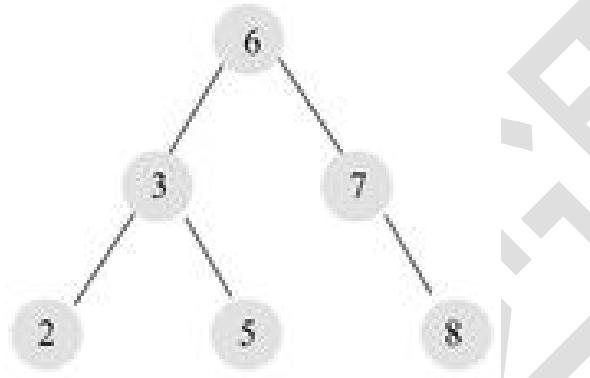


(b). Full Tree

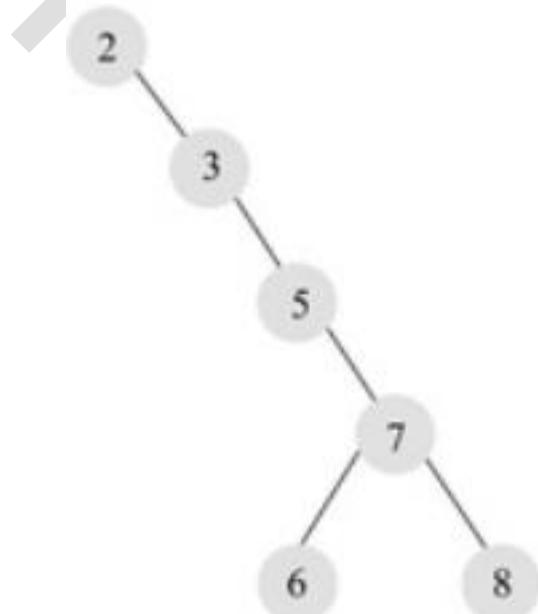
#### 11.1.1.3.3. 平衡二叉树 (AVL-树)

- 左子树和右子树都是平衡二叉树;
- 左子树和右子树的高度差绝对值不超过 1;

• 平衡二叉树



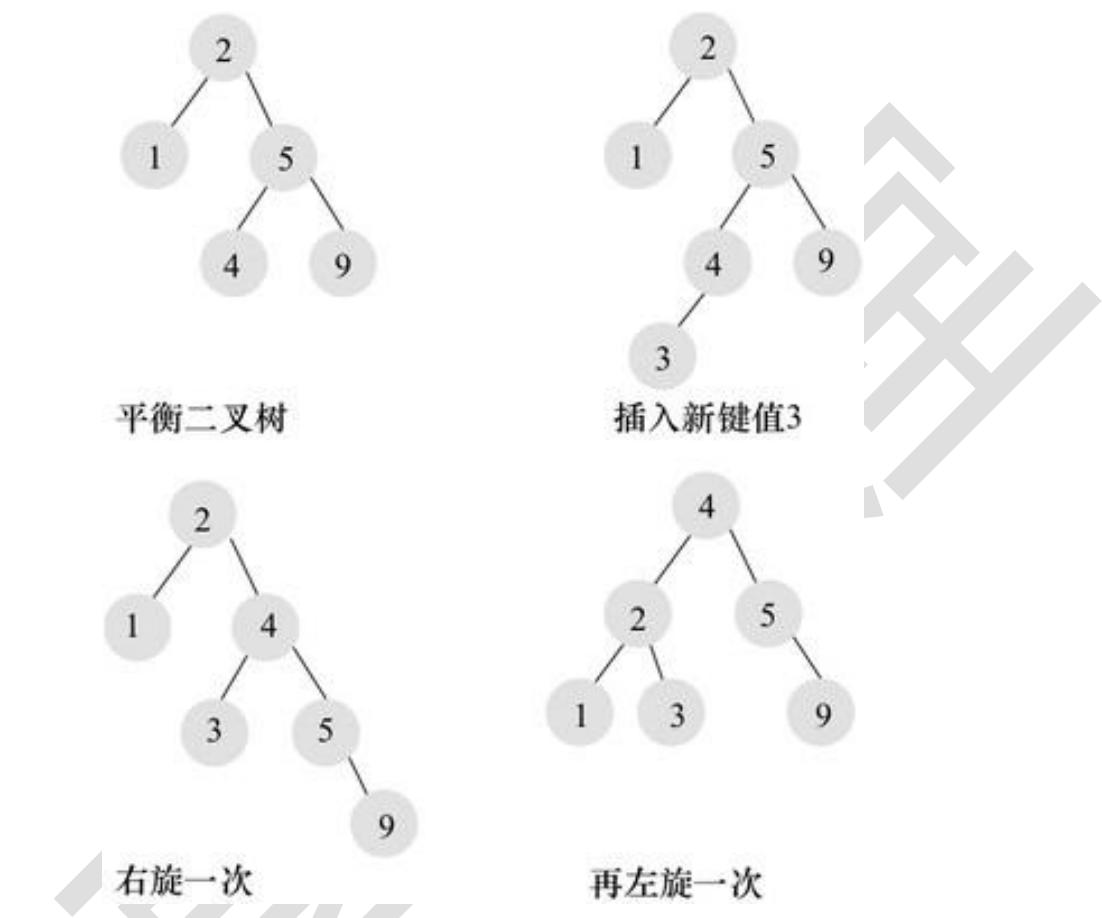
• 非平衡二叉树



### 11.1.1.3.3.1. 平衡二叉树的遍历

- 前序 : 6,3,2,5,7,8 (ROOT 节点在开头, 中 - 左 - 右 顺序)
- 中序 : 2,3,5,6,7,8 (中序遍历即为升序, 左 - 中 - 右 顺序)
- 后序 : 2,5,3,8,7,6 (ROOT 节点在结尾, 左 - 右 - 中 顺序)

### 11.1.1.3.3.2. 平衡二叉树的旋转



需要通过旋转 (左旋, 右旋) 来维护平衡二叉树的平衡, 在添加和删除的时候需要有额外的开销。

### 11.1.1.3.4. B+树

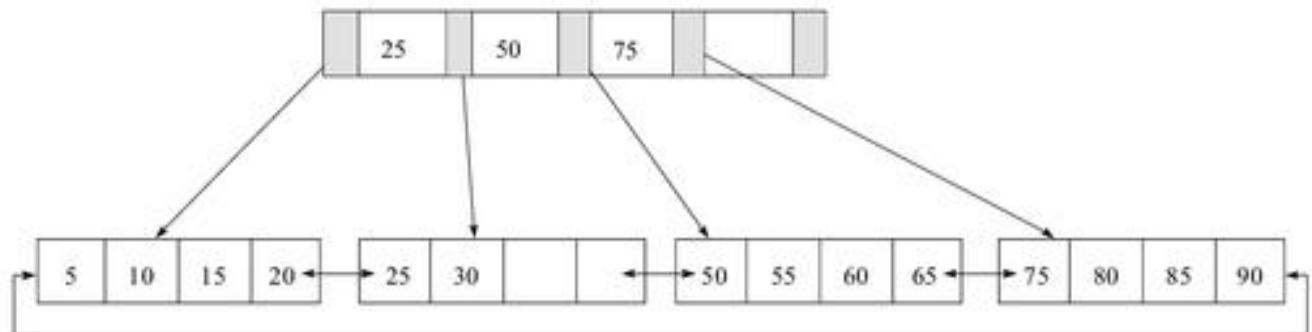
#### 11.1.1.3.4.1. B+树的定义

- 数据只存储在叶子节点上, 非叶子节点只保存索引信息;
  - 非叶子节点 (索引节点) 存储的只是一个 Flag, 不保存实际数据记录;
  - 索引节点指示该节点的左子树比这个 Flag 小, 而右子树大于等于这个 Flag
- 叶子节点本身按照数据的升序排序进行链接(串联起来);
  - 叶子节点中的数据在 **物理存储上是无序的**, 仅仅是在 **逻辑上有序** (通过指针串在一起);

#### 11.1.1.3.4.2. B+树的作用

- 在块设备上，通过 B+树可以有效的存储数据；
- 所有记录都存储在叶子节点上，非叶子(non-leaf)存储索引(keys)信息；
- B+树含有非常高的扇出 (fanout)，通常超过 100，在查找一个记录时，可以有效的减少 IO 操作；

#### 11.1.1.3.4.3. B+树的扇出(fan out)



- 该 B+ 树高度为 2
- 每叶子页 (LeafPage) 4 条记录
- 扇出数为 5
- 叶子节点(LeafPage)由小到大 (有序) 串联在一起

**扇出** 是每个索引节点(Non-LeafPage)指向每个叶子节点(LeafPage)的指针

**扇出数 = 索引节点(Non-LeafPage)可存储的最大关键字个数 + 1**

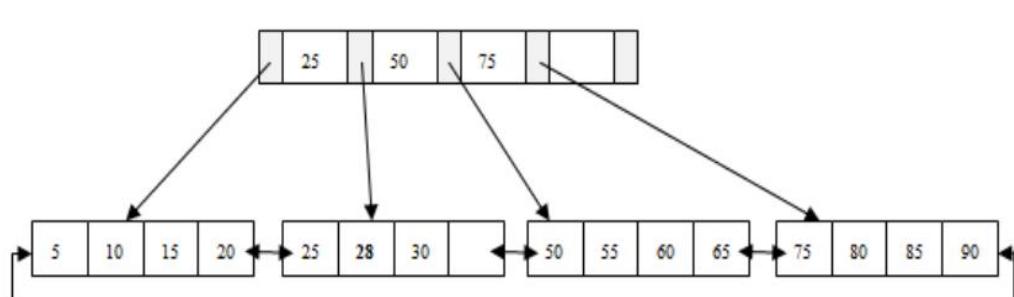
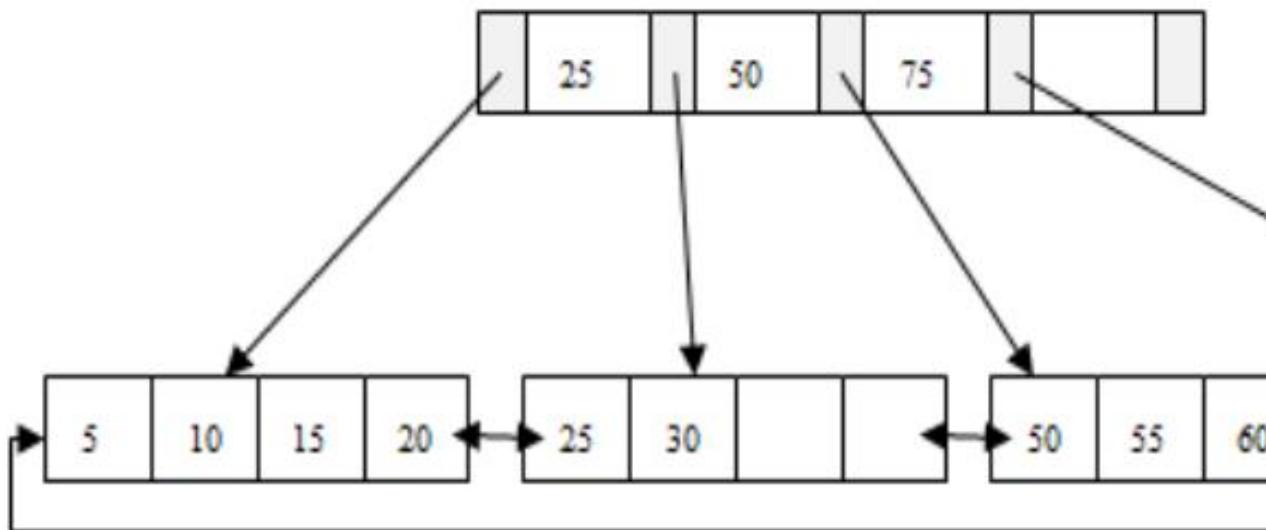
图例中的索引节点(Non-LeafPage)最大可以存放 4 个关键字，但实际使用了 3 个；

#### 11.1.1.3.4.4. B+树的插入操作

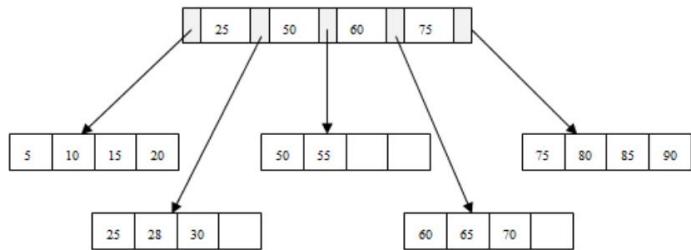
- B+树的插入
- B+树的插入必须保证插入后叶子节点中的记录依然排序。

Leaf Page 满	Index Page 满	操作
No	No	直接将记录插入到叶子节点
Yes	No	1) 拆分 Leaf Page 2) 将中间的节点放入到 Index Page 中 3) 小于中间节点的记录放左边 4) 大于或等于中间节点的记录放右边
Yes	Yes	1) 拆分 Leaf Page 2) 小于中间节点的记录放左边 3) 大于或等于中间节点的记录放右边 4) 拆分 Index Page 5) 小于中间节点的记录放左边 6) 大于中间节点的记录放右边 7) 中间节点放入上一层 Index Page

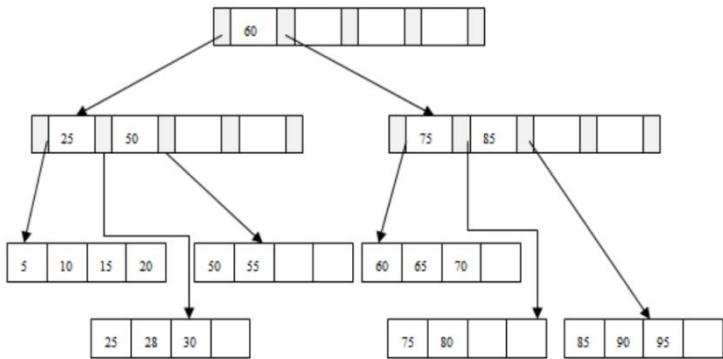
问题：1 插入 28



问题 2：插入 70



问题 3：插入 95



<https://blog.csdn.net/shenchaohao12321/article/details/83243314>

### 11.1.2. 索引的分类

**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

**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

### 11.1.3. 基础语法

**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;
```

## 11.2. 执行计划

### 11.2.1. 什么是执行计划

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。

### 11.2.2. 执行计划的作用

- 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

以上的这些作用会在执行计划详解里面介绍到，在这里不做解释。

### 11.2.3. 执行计划的语法

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

## 11.2.4. 执行计划详解

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   select_type   table   type   possible_keys   key   key_len   ref   rows   Extra									

### 11.2.4.1. 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

分别举例来看

#### 11.2.4.1.1. Id 相同

```
mysql> explain select t2.*  
    >   from t1, t2, t3  
    >   where t1.id = t2.id and t1.id = t3.id  
    >   and t1.other_column = '';
```

id   select_type   table   type   possible_keys   key   key_len   ref   rows   Extra
1   SIMPLE       t1    ref    PRIMARY, idx_t1   idx_t1   92     const   1      Using where
1   SIMPLE       t3    eq_ref   PRIMARY        PRIMARY   4      test.t1.ID   1      Using index
1   SIMPLE       t2    eq_ref   PRIMARY        PRIMARY   4      test.t1.ID   1

3 rows in set (0.00 sec)

如上图所示，ID 列的值全为 1，代表执行的允许从 t1 开始加载，依次为 t3 与 t2

EXPLAIN

```
select t2.* from t1,t2,t3  where t1.id = t2.id and t1.id = t3.id  
and t1.other_column = '';
```

### 11.2.4.1.2. Id 不同

```
mysql> explain SELECT t2.*  
-> FROM t2  
-> WHERE id = (SELECT id  
-> FROM t1  
-> WHERE id = (SELECT t3.id  
-> FROM t3  
-> WHERE t3.other_column = ''));  
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |  
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
| 1 | PRIMARY | t2 | const | PRIMARY | PRIMARY | 4 | const | 1 | |
| 2 | SUBQUERY | t1 | const | PRIMARY | PRIMARY | 4 | const | 1 | Using index |  
| 3 | SUBQUERY | t3 | ALL | NULL | NULL | NULL | NULL | 1 | Using where |  
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
3 rows in set (0.00 sec)
```

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

EXPLAIN

```
select t2.* from t2 where id = (  
select id from t1 where id = (select t3.id from t3 where t3.other_column="")  
);
```

### 11.2.4.1.3. Id 相同又不同

```
mysql> explain select t2.* from (  
-> select t3.id  
-> from t3  
-> where t3.other_column = '') s1, t2  
-> where s1.id = t2.id;  
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |  
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
| 1 | PRIMARY | <derived2> | system | NULL | NULL | NULL | NULL | 1 | |
| 1 | PRIMARY | t2 | const | PRIMARY | PRIMARY | 4 | const | 1 |  
| 2 | DERIVED | t3 | ALL | NULL | NULL | NULL | NULL | 1 | Using where |  
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
3 rows in set (0.00 sec)
```

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

EXPLAIN

```
select t2.* from (  
select t3.id  
from t3 where t3.other_column = "  
) s1 ,t2 where s1.id = t2.id
```

## 11.2.4.2. 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

类型如下

类型	描述
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
SUBQUERY	在SELECT或WHERE列表中包含了子查询
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。
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

### 11.2.4.2.1. SIMPLE

EXPLAIN select \* from t1

简单的 select 查询,查询中不包含子查询或者 UNION

EXPLAIN select * from t1									
信息	结果1	概况	状态						
1	SIMPLE	t1	ALL	(Null)	(Null)	(Null)	(Null)	1	(Null)

### 11.2.4.2.2. PRIMARY 与 SUBQUERY

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

SUBQUERY: 在 SELECT 或 WHERE 列表中包含了子查询

EXPLAIN

select t1.\*,(select t2.id from t2 where t2.id = 1 ) from t1

信息	结果1	概况	状态						
id	select type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶ 1	PRIMARY	t1	ALL	(Null)	(Null)	(Null)	(Null)	1	(Null)
2	SUBQUERY	t2	const	PRIMARY	PRIMAR 4	const	1		Using index

#### **11.2.4.2.3. 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。

```
select t1.* from t1 ,(select t2.* from t2 where t2.id = 1 ) s2  where t1.id = s2.id
```

**1 EXPLAIN** select t1.\* from t1 , (select t2.\* from t2 where t2.id = 1 ) s2 where t1.id = s2.id

信息	结果1	概况	状态							
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra	
1	PRIMARY	<derived2>	system	(Null)	(Null)	(Null)	(Null)	1	(Null)	
1	PRIMARY	t1	const	PRIMARY	PRIMARY	4	const	1	(Null)	
2	DERIVED	t2	const	PRIMARY	PRIMARY	4	const	1	(Null)	

#### **11.2.4.2.4. UNION RESULT 与 UNION**

**UNION:** 若第二个 SELECT 出现在 UNION 之后，则被标记为 UNION;

**UNION RESULT:** 从 UNION 表获取结果的 SELECT

#UNION RESULT ,UNION

## EXPLAIN

```
select * from t2
```

## UNION

```
select * from t2
```

```
2 select * from t1  
3 UNION  
4 select * from t2
```

### 11.2.4.3. table 列

显示这一行的数据是关于哪张表的

```
15  
16 #UNION RESULT ,UNION  
17 EXPLAIN  
18 select * from t1  
19 UNION  
20 select * from t2
```

信息				结果1	概况	状态
id	select_type	table	type	possible_keys	key	key_len
1	PRIMARY	t1	ALL	(Null)	(Null)	(Null)
2	UNION	t2	ALL	(Null)	(Null)	(Null)
(Null)	UNION RESULT <union1,2>		ALL	(Null)	(Null)	(Null)

### 11.2.4.4. Type 列

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。

#### 11.2.4.4.1. 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

```
mysql> explain select * from (select * from t1 where id = 1) d1;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	system	NULL	NULL	NULL	NULL	1	
2	DERIVED	t1	const	PRIMARY	PRIMARY	4	const	1	

2 rows in set (0.00 sec)

```
1 EXPLAIN  
2 SELECT * from (select * from t2 where id = 1) d1;  
3
```

信息 结果1 概况 状态

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	system	(Null)	(Null)	(Null)	(Null)	1	(Null)
2	DERIVED	t2	const	PRIMARY	PRIMARY	4	const	1	(Null)

EXPLAIN

```
SELECT * from (select * from t2 where id = 1) d1;
```

#### 11.2.4.4.2. 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

```
mysql> explain select * from t1, t2 where t1.id = t2.id;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t2	ALL	PRIMARY	NULL	NULL	NULL	639	
1	SIMPLE	t1	eq_ref	PRIMARY	PRIMARY	4	shared.t2.ID	1	

2 rows in set (0.00 sec)

```
2 SELECT * from t1,t2 where t1.id = t2.id  
3
```

信息 结果1 概况 状态

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	ALL	PRIMARY	(Null)	(Null)	(Null)	1	(Null)
1	SIMPLE	t2	eq_ref	PRIMARY	PRIMARY	4	mysqld:1	1	(Null)

EXPLAIN

```
SELECT * from t1,t2 where t1.id = t2.id
```

### 11.2.4.4.3. 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

```
mysql> create index idx_col1_col2 on t1(col1,col2);
Query OK, 1000 rows affected (0.15 sec)
Records: 1000 Duplicates: 0 Warnings: 0

mysql> select count(distinct col1) from t1;
+-----+
| count(distinct col1) |
+-----+
|          7 |
+-----+
1 row in set (0.00 sec)

mysql> explain select * from t1 where col1 = 'ac';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key           | key_len | ref   | rows  | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE      | t1   | ref  | idx_col1_col2 | idx_col1_col2 | 194    | const | 284   |        |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

EXPLAIN

```
select count(DISTINCT col1) from t1 where col1 = 'ac'
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	ref	idx_col1_col2	idx_col1_col2	194	const	284	

或者

EXPLAIN

```
select col1 from t1 where col1 = 'ac'
```

### 11.2.4.4.4. 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。

```
mysql> explain SELECT * FROM t1 WHERE id between 30 and 60;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t1 | range | PRIMARY | PRIMARY | 4 | NULL | 31 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

```
mysql> explain select * from t1 where id in (1, 2, 6);
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t1 | range | PRIMARY | PRIMARY | 4 | NULL | 3 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

EXPLAIN select \* from t1 where id BETWEEN 30 and 60

EXPLAIN select \* from t1 where id in(1,2)

#### 11.2.4.4.5. 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。

```
mysql> explain select id from t1;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t1 | index | NULL | PRIMARY | 4 | NULL | 516 | Using index |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

```
3
4 EXPLAIN
5 select c2 from testdemo
6
```

信息	结果1	概况	状态						
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	testdemo	index	(Null)	idx_c2	5	(Null)	5	Using index

EXPLAIN  
select c2 from testdemo

#### 11.2.4.4.6. All

Full Table Scan, 将遍历全表以找到匹配的行

```

mysql> explain select * from t1 where column_without_index = '';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | t1   | ALL  | NULL        | NULL | NULL    | NULL | 516  |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

```

### 11.2.4.5. possible\_keys 与 Key

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

这里的覆盖索引非常重要，后面会单独的来讲

```

mysql> create index idx_col1_col2 on t2(col1,col2);
Query OK, 1001 rows affected (0.17 sec)
Records: 1001  Duplicates: 0  Warnings: 0

```

```

mysql> explain select col1, col2 from t1;
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | t1   | index | NULL        | idx_col1_col2 | 390 | NULL | 682 |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

```

EXPLAIN select col1,col2 from t1

其中 key 和 possible\_keys 都可以出现 null 的情况（结婚邀请朋友的例子）

### 11.2.4.6. key\_len

```

mysql> desc t1;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra           |
+-----+-----+-----+-----+-----+
| ID    | int(11) | NO   | PRI | NULL    | auto_increment |
| col1  | char(4)  | YES  | MUL | NULL    |                 |
| col2  | char(4)  | YES  |      | NULL    |                 |
+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> explain select * from t1 where col1 = 'ab';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id  | select_type | table | type  | possible_keys | key   | key_len | ref   | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1   | SIMPLE     | t1   | ref   | idx_col1_col2 | idx_col1_col2 | 13    | const | 143  |       |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select * from t1 where col1 = 'ab' and col2 = 'ac';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id  | select_type | table | type  | possible_keys | key   | key_len | ref   | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1   | SIMPLE     | t1   | ref   | idx_col1_col2 | idx_col1_col2 | 26    | const,const | 1 |       |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.01 sec)

```

desc  
select \* from ta where col1 ='ab';

desc  
select \* from ta where col1 ='ab' and col2 = 'ac'

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

### 注意

根据底层使用的不同存储引擎，受影响的行数这个指标可能是一个估计值，也可能是一个精确值。即使受影响的行数是一个估计值(例如当使用 InnoDB 存储引擎管理表存储时)，通常情况下这个估计值也足以使优化器做出一个有充分依据的决定。

- 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）

### 11.2.4.6.1. 字符类型

#### 字符串类型

字符串类型指CHAR、VARCHAR、BINARY、VARBINARY、BLOB、TEXT、ENUM和SET。该节描述了这些类型如何工作以及如何在查询中使用这些类型。

类型	大小	用途
CHAR	0-255字节	定长字符串
VARCHAR	0-65535 字节	变长字符串
TINYBLOB	0-255字节	不超过 255 个字符的二进制字符串
TINYTEXT	0-255字节	短文本字符串
BLOB	0-65 535字节	二进制形式的长文本数据
TEXT	0-65 535字节	长文本数据
MEDIUMBLOB	0-16 777 215字节	二进制形式的中等长度文本数据
MEDIUMTEXT	0-16 777 215字节	中等长度文本数据
LONGBLOB	0-4 294 967 295字节	二进制形式的极大文本数据
LONGTEXT	0-4 294 967 295字节	极大文本数据

CHAR和VARCHAR类型类似，但它们保存和检索的方式不同。它们的最大长度和是否尾部空格被保留等方面也不同。在存储或检索过程中不进行大小写转换。

以上这个表列出了所有字符类型，但真正建所有的类型常用情况只是 CHAR、VARCHAR

#### 11.2.4.6.1.1. 字符类型-索引字段为 char 类型+不可为 Null 时

```

1
2 CREATE TABLE `s1` (
3     `id` int(11) NOT NULL AUTO_INCREMENT,
4     `name` char(10) NOT NULL,
5     `addr` varchar(20) DEFAULT NULL,
6     PRIMARY KEY (`id`),
7     KEY `name` (`name`)
8 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
9
10 explain select * from s1 where name='enjoy';
11

```

信息 结果1 概况 状态									
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	s1	ref	name	name	30	const	1	Using index

```

CREATE TABLE `s1` (
    `id` int(11) NOT NULL AUTO_INCREMENT,
    `name` char(10) NOT NULL,
    `addr` varchar(20) DEFAULT NULL,
    PRIMARY KEY (`id`),
    KEY `name` (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

```
explain select * from s1 where name='enjoy';
```

name 这一列为 char(10), 字符集为 utf-8 占用 3 个字节  
Keylen=10\*3

### 11.2.4.6.1.2. 字符类型-索引字段为 char 类型+允许为 Null 时

```

1 CREATE TABLE `s2` (
2     `id` int(11) NOT NULL AUTO_INCREMENT,
3     `name` char(10) DEFAULT NULL,
4     `addr` varchar(20) DEFAULT NULL,
5     PRIMARY KEY (`id`),
6     KEY `name` (`name`)
7 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
8
9 explain select * from s2 where name='enjoyedu';

```

信息 结果1 概况 状态									
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	s2	ref	name	name	31	const	1	Using index

```

CREATE TABLE `s2` (
    `id` int(11) NOT NULL AUTO_INCREMENT,

```

```

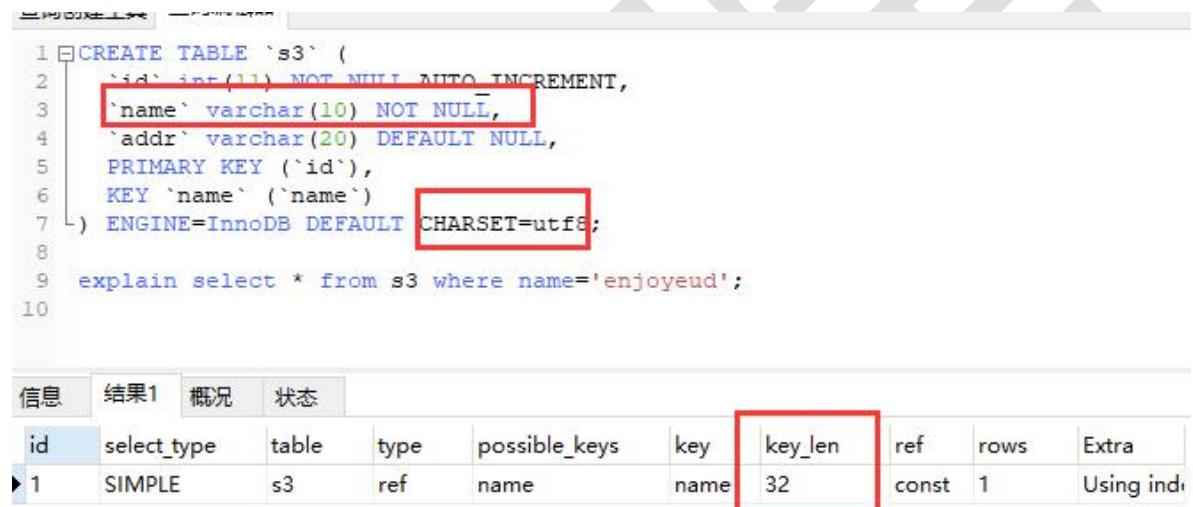
`name` char(10) DEFAULT NULL,
`addr` varchar(20) DEFAULT NULL,
PRIMARY KEY (`id`),
KEY `name` (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

```
explain select * from s2 where name='enjoyedu';
```

name 这一列为 char(10),字符集为 utf-8 占用 3 个字节,外加需要存入一个 null 值  
Keylen=10\*3+1(null) 结果为 31

#### 11.2.4.6.1.3. 索引字段为 varchar 类型+不可为 Null 时



The screenshot shows the MySQL Workbench interface. At the top, there is a code editor window containing the SQL code for creating table s3. The 'name' column is highlighted with a red box. Below the code editor is an 'explain' output table.

信息	结果1	概况	状态						
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	s3	ref	name	name	32	const	1	Using index

```

CREATE TABLE `s3` (
`id` int(11) NOT NULL AUTO_INCREMENT,
`name` varchar(10) NOT NULL,
`addr` varchar(20) DEFAULT NULL,
PRIMARY KEY (`id`),
KEY `name` (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

```
explain select * from s3 where name='enjoyeud';
```

Keylen=varchar(n)变长字段+不允许 Null=n\*(utf8=3,gbk=2,latin1=1)+2

#### 11.2.4.6.1.4. 索引字段为 varchar 类型+允许为 Null 时

```
1 CREATE TABLE `s4` (
2   `id` int(11) NOT NULL AUTO_INCREMENT,
3   `name` varchar(10) DEFAULT NULL,
4   `addr` varchar(20) DEFAULT NULL,
5   PRIMARY KEY (`id`),
6   KEY `name` (`name`)
7 ) ENGINE=InnoDB DEFAULT CHARSET=utf8
8
9 explain select * from s4 where name='enjoyedu';
10
```

信息	结果1	概况	状态							
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra	
1	SIMPLE	s4	ref	name	name	33	const	1	Using index	

```
CREATE TABLE `s3` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(10) NOT NULL,
  `addr` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `name` (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
explain select * from s3 where name='enjoyeud';
```

Keylen=varchar(n)变长字段+允许 Null=n\*(utf8=3,gbk=2,latin1=1)+1(NULL)+2

## 11.2.4.6.2. 数值类型

类型	大小	范围(有符号)	范围(无符号)	用途
TINYINT	1字节	(-128, 127)	(0, 255)	小整数值
SMALLINT	2字节	(-32 768, 32 767)	(0, 65 535)	大整数值
MEDIUMINT	3字节	(-8 388 608, 8 388 607)	(0, 16 777 215)	大整数值
INT或 INTEGER	4字节	(-2 147 483 648, 2 147 483 647)	(0, 4 294 967 295)	大整数值
BIGINT	8字节	(-9 233 372 036 854 775 808, 9 223 372 036 854 775 807)	(0, 18 446 744 073 709 551 615)	极大整数值
FLOAT	4字节	(-3.402 823 466 E+38, -1.175 494 351 E-38), 0, (1.175 494 351 E-38), 3.402 823 466 351 E+38)	0, (1.175 494 351 E-38, 3.402 823 466 351 E+38)	单精度浮点数值
DOUBLE	8字节	(-1.797 693 134 862 315 7 E+308, -2.225 073 858 507 201 4 E-308, 073 858 507 201 4 E-308), 0, (2.225 073 858 507 201 4 E-308, 1.797 693 134 862 315 7 E+308)	0, (2.225 073 858 507 201 4 E-308, 1.797 693 134 862 315 7 E+308)	双精度浮点数值

```

CREATE TABLE `numberKeyLen` (
`c0` int(255) NOT NULL,
`c1` tinyint(255) NULL DEFAULT NULL,
`c2` smallint(255) NULL DEFAULT NULL,
`c3` mediumint(255) NULL DEFAULT NULL,
`c4` int(255) NULL DEFAULT NULL,
`c5` bigint(255) NULL DEFAULT NULL,
`c6` float(255,0) NULL DEFAULT NULL,
`c7` double(255,0) NULL DEFAULT NULL,
PRIMARY KEY (`c0`),
INDEX `index_tinyint`(`c1`) USING BTREE,
INDEX `index_smallint`(`c2`) USING BTREE,
INDEX `index_mediumint`(`c3`) USING BTREE,
INDEX `index_int`(`c4`) USING BTREE,
INDEX `index_bigint`(`c5`) USING BTREE,
INDEX `index_float`(`c6`) USING BTREE,
INDEX `index_double`(`c7`) USING BTREE
)

```

```
)  
ENGINE=InnoDB  
DEFAULT CHARACTER SET=utf8 COLLATE=utf8_general_ci  
ROW_FORMAT=COMPACT  
;
```

```
EXPLAIN  
select * from numberKeyLen where c1=1
```

```
EXPLAIN  
select * from numberKeyLen where c2=1
```

```
EXPLAIN  
select * from numberKeyLen where c3=1
```

```
EXPLAIN  
select * from numberKeyLen where c4=1
```

```
EXPLAIN  
select * from numberKeyLen where c5=1
```

```
EXPLAIN  
select * from numberKeyLen where c6=1
```

```
EXPLAIN  
select * from numberKeyLen where c7=1
```

#### 11.2.4.6.3. 日期和时间

## 日期和时间类型

表示时间值的日期和时间类型为DATETIME、DATE、TIMESTAMP、TIME和YEAR。

每个时间类型有一个有效值范围和一个“零”值，当指定不合法的MySQL不能表示的值时使用“零”值。

TIMESTAMP类型有专有的自动更新特性，将在后面描述。

类型	大小 (字节)	范围	格式	用途
DATE	3	1000-01-01/9999-12-31	YYYY-MM-DD	日期值
TIME	3	'-838:59:59'/'838:59:59'	HH:MM:SS	时间值或持续时间
YEAR	1	1901/2155	YYYY	年份值
DATETIME	8	1000-01-01 00:00:00/9999-12-31 23:59:59	YYYY-MM-DD HH:MM:SS	混合日期和时间值
TIMESTAMP	4	1970-01-01 00:00:00/2037 年某时	YYYYMMDD HHMMSS	混合日期和时间值， 时间戳

datetime 类型在 5.6 中字段长度是 5 个字节

datetime 类型在 5.5 中字段长度是 8 个字节

```
CREATE TABLE `datatimekeylen` (  
`c1` date NULL DEFAULT NULL ,  
`c2` time NULL DEFAULT NULL ,  
`c3` year NULL DEFAULT NULL ,  
`c4` datetime NULL DEFAULT NULL ,  
`c5` timestamp NULL DEFAULT NULL ,  
INDEX `index_date`(`c1`) USING BTREE ,  
INDEX `index_time`(`c2`) USING BTREE ,  
INDEX `index_year`(`c3`) USING BTREE ,  
INDEX `index_datetime`(`c4`) USING BTREE ,  
INDEX `index_timestamp`(`c5`) USING BTREE  
)  
ENGINE=InnoDB  
DEFAULT CHARACTER SET=utf8 COLLATE=utf8_general_ci  
ROW_FORMAT=COMPACT  
;
```

EXPLAIN

```
SELECT * from datatimekeylen where c1 = 1
```

EXPLAIN

```
SELECT * from datatimekeylen where c2 = 1
```

EXPLAIN

```
SELECT * from datatimekeylen where c3 = 1
```

```
EXPLAIN
```

```
SELECT * from datatimekeylen where c4 = 1
```

```
EXPLAIN
```

```
SELECT * from datatimekeylen where c5 = 1
```

## 11.2.4.6.4. 总结

### 11.2.4.6.4.1. 字符类型

变长字段需要额外的 2 个字节（VARCHAR 值保存时只保存需要的字符数，另加一个字节来记录长度(如果列声明的长度超过 255，则使用两个字节)，所以 VARCAHR 索引长度计算时候要加 2），固定长度字段不需要额外的字节。

而 NULL 都需要 1 个字节的额外空间,所以索引字段最好不要为 NULL, 因为 NULL 让统计更加复杂并且需要额外的存储空间。

复合索引有最左前缀的特性，如果复合索引能全部使用上，则是复合索引字段的索引长度之和，这也[可以用来判定复合索引是否部分使用，还是全部使用](#)。

### 11.2.4.6.4.2. 整数/浮点数/时间类型的索引长度

NOT NULL=字段本身的字段长度

NULL=字段本身的字段长度+1(因为需要有是否为空的标记，这个标记需要占用 1 个字节)

datetime 类型在 5.6 中字段长度是 5 个字节， datetime 类型在 5.5 中字段长度是 8 个字节

## 11.2.4.7. 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

```

mysql> explain select * from t1, t2 where t1.col1 = t2.col1 and t1.col2 = 'ac';
+-----+-----+-----+-----+-----+-----+-----+
| id | table | type | possible_keys | key | key_len | ref | rows |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | t2 | ALL | NULL | NULL | NULL | NULL | 640 |
| 1 | t1 | ref | idx_col1_col2 | idx_col1_col2 | 26 | shared.t2.col1,const | 82 |
+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.01 sec)

```

#### EXPLAIN

```
select * from s1 ,s2 where s1.id = s2.id and s1.name = 'enjoy'
```

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】

#### 11.2.4.8. 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

```

mysql> explain select * from t1, t2 where t1.id = t2.id and t2.col1 = 'ac';
+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t2 | ALL | PRIMARY | NULL | NULL | NULL | 640 | Using where |
| 1 | SIMPLE | t1 | eq_ref | PRIMARY | PRIMARY | 4 | shared.t2.ID | 1 |
+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> create index idx_col1_col2 on t2(col1,col2);
Query OK, 1001 rows affected (0.17 sec)
Records: 1001  Duplicates: 0  Warnings: 0

mysql> explain select * from t1, t2 where t1.id = t2.id and t2.col1 = 'ac';
+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t2 | ref | PRIMARY,idx_col1_col2 | idx_col1_col2 | 195 | const | 142 |
| 1 | SIMPLE | t1 | eq_ref | PRIMARY | PRIMARY | 4 | shared.t2.ID | 1 |
+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

```

#### 11.2.4.9. Extra

包含不适合在其他列中显示但十分重要的额外信息。

值	描述
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”
Using temporary	使用了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。
USING index	是否用了覆盖索引
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

#### 11.2.4.9.1. 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”

当发现有 Using filesort 后，实际上就是发现了可以优化的地方

```
mysql> explain select col1 from t1 where col1 = 'ac' order by col3\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: t1
         type: ref
possible_keys: idx_col1_col2_col3
          key: idx_col1_col2_col3
       key_len: 13
         ref: const
        rows: 142
    Extra: Using where; Using index; Using filesort
1 row in set (0.00 sec)
```

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

```
mysql> explain select col1 from t1 where col1 = 'ac' order by col2, col3\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
    table: t1
    type: ref
possible_keys: idx_col1_col2_col3
      key: idx_col1_col2_col3
    key_len: 13
      ref: const
     rows: 142
    Extra: Using where; Using index
1 row in set (0.00 sec)
```

当我排序新增了个 col2，发现 using filesort 就没有了。

```
EXPLAIN select col1 from t1 where col1='ac' order by col3
```

```
EXPLAIN select col1 from t1 where col1='ac' order by col2,col3
```

#### 11.2.4.9.2. Using temporary

用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。

```
mysql> explain select col1 from t1 where col1 in ('ac','ab','aa') group by col2\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
    table: t1
    type: range
possible_keys: idx_col1_col2
      key: idx_col1_col2
    key_len: 13
      ref: NULL
     rows: 569
    Extra: Using where; Using index; Using temporary; Using filesort
1 row in set (0.00 sec)
```

```
mysql> explain select col1 from t1 where col1 in ('ac', 'ab') group by col1, col2\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
    table: t1
    type: range
possible_keys: idx_col1_col2_col3
      key: idx_col1_col2_col3
    key_len: 26
      ref: NULL
     rows: 4
    Extra: Using where; Using index for group-by
1 row in set (0.00 sec)
```

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

```
EXPLAIN select col1 from t1 where col1 in('ac','ab','aa') GROUP BY col2
```

```
EXPLAIN select col1 from t1 where col1 in('ac','ab','aa') GROUP BY col1,col2
```

### 11.2.4.9.3. 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；

```
mysql> explain select col2 from t1 where col1 = 'ab';
+-----+-----+-----+-----+-----+
| id | ... | possible_keys | key           | key_len | ref   | rows | Extra      |
+-----+-----+-----+-----+-----+
| 1  | ... | idx_col1_col2 | idx_col1_col2 | 13     | const | 143  | Using where; Using index |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

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

```
mysql> explain select col1, col2 from t1;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type  | possible_keys | key           | key_len | ref   | rows | Extra      |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE    | t1   | index | NULL    | idx_col1_col2 | 398    | NULL  | 682  | Using index |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

```
EXPLAIN select col2 from t1 where col1 = 'ab'
```

```
EXPLAIN select col2 from t1
```

#### 11.2.4.9.3.1. 覆盖索引：

覆盖索引 (Covering Index) ,一说为索引覆盖。

理解方式一:就是 select 的数据列只用从索引中就能够取得，不必读取数据行，MySQL 可以利用索引返回 select 列表中的字段，而不必根据索引再次读取数据文件,换句话说查询列要被所建的索引覆盖。

理解方式二:索引是高效找到行的一个方法，但是一般数据库也能使用索引找到一个列的数据，因此它不必读取整个行。毕竟索引叶子节点存储了它们索引的数据;当能通过读取索引

就可以得到想要的数据，那就不需要读取行了。一个索引包含了(或覆盖了)满足查询结果的数据就叫做覆盖索引

注意：

如果要使用覆盖索引，一定要注意 select 列表中只取出需要的列，不可 select \*，因为如果将所有字段一起做索引会导致索引文件过大，查询性能下降。

所以，千万不能为了查询而在所有列上都建立索引，会严重影响修改维护的性能。

#### 11.2.4.9.4. 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：

show VARIABLES like '%join\_buffer\_size%'

查询创建工具   语句编辑器										
EXPLAIN 2 select * from t1 INNER JOIN t2 on t1.other_column = t2.other_column										
信息	结果1	概况	状态							
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra	
1	SIMPLE	t1	ALL	(Null)	(Null)	(Null)	(Null)	1	Using where;	
1	SIMPLE	t2	ALL	(Null)	(Null)	(Null)	(Null)	1	Using join buffer	

EXPLAIN

select \* from t1 JOIN t2 on t1.other\_column = t2.other\_column

#### 11.2.4.9.5. impossible where

where 子句的值总是 false，不能用来获取任何元组

EXPLAIN 2 select * from t1 where t1.other_column ='enjoy' and t1.other_column ='edu' 3										
信息	结果1	概况	状态							
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra	
1	SIMPLE	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	Impossible WHERE	

EXPLAIN

select \* from t1 where 1=2

EXPLAIN

```
select * from t1 where t1.other_column = 'enjoy' and t1.other_column = 'edu'
```

## 12. SQL 优化

### 12.1. 优化实战

#### 12.1.1. 策略 1. 尽量全值匹配

```
mysql> EXPLAIN SELECT * FROM staffs WHERE NAME = 'July';
+----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref |
+----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ref | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 74 | const |
+----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> EXPLAIN SELECT * FROM staffs WHERE NAME = 'July' AND age = 25;
+----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref |
+----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ref | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 78 | const,const |
+----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> EXPLAIN SELECT * FROM staffs WHERE NAME = 'July' AND age = 25 AND pos = 'dev';
+----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref |
+----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ref | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 140 | const,const,const |
+----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

```
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 where 条件中使用索引的尽量所用。

### 12.1.2. 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。

```
mysql> EXPLAIN SELECT * FROM staffs WHERE age = 25 AND pos = 'dev';  
+----+----+----+----+----+----+----+----+  
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra  
+----+----+----+----+----+----+----+----+  
| 1 | SIMPLE | staffs | ALL | NULL | NULL | NULL | NULL | 2 | Using where |  
+----+----+----+----+----+----+----+----+  
1 row in set (0.00 sec)  
  
mysql> EXPLAIN SELECT * FROM staffs WHERE pos = 'dev';  
+----+----+----+----+----+----+----+----+  
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra  
+----+----+----+----+----+----+----+----+  
| 1 | SIMPLE | staffs | ALL | NULL | NULL | NULL | NULL | 2 | Using where |  
+----+----+----+----+----+----+----+----+  
1 row in set (0.00 sec)  
  
mysql> EXPLAIN SELECT * FROM staffs WHERE NAME = 'July';  
+----+----+----+----+----+----+----+----+  
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra  
+----+----+----+----+----+----+----+----+  
| 1 | SIMPLE | staffs | ref | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 74 | const | 1 | Using where |  
+----+----+----+----+----+----+----+----+  
1 row in set (0.00 sec)  
  
mysql>
```

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

### 12.1.3. 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

```
mysql> explain select * from staffs where left(name,4) = 'July';  
+----+----+----+----+----+----+----+----+  
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra  
+----+----+----+----+----+----+----+----+  
| 1 | SIMPLE | staffs | ALL | NULL | NULL | NULL | NULL | 5 | Using where |  
+----+----+----+----+----+----+----+----+  
1 row in set (0.00 sec)  
  
mysql> █
```

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

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

## 12.1.4. 策略 4. 范围条件放最后

```
mysql> explain select * from staffs where name='z4';
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ref | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 74 | const | 1 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select * from staffs where name='z4' and age=22;
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ref | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 78 | const,const | 1 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select * from staffs where name='z4' and age=22 and pos='manager';
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ref | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 140 | const,const,const | 1 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select * from staffs where name='z4' and age>11 and pos='manager';
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | range | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 78 | NULL | 1 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

```
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

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

## 12.1.5. 策略 5. 覆盖索引尽量用

```

mysql> select * from staffs where name='z3' and age=22 and pos='manager';
+----+-----+---+---+
| id | NAME | age | pos |
+----+-----+---+---+
| 1  | z3   | 22  | manager |
+----+-----+---+---+
1 row in set (0.00 sec)

mysql> explain select * from staffs where name='z3' and age=22 and pos='manager';
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE      | staffs | ref  | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 140 | const,const,const | 1  | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select name,age,pos from staffs where name='z3' and age=22 and pos='manager';
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE      | staffs | ref  | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 140 | const,const,const | 1  | Using where: Using index |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select * from staffs where name='z3' and age>22 and pos='manager';
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE      | staffs | range | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 78  | NULL | 1  | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select name,age,pos from staffs where name='z3' and age>22 and pos='manager';
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE      | staffs | ref  | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 74  | const | 1  | Using where: Using index |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

```

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'

## 12.1.6. 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

```

mysql> explain select * from staffs where name='July';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ref | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 74 | const | 1 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select * from staffs where name != 'July';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ALL | idx_staffs_nameAgePos | NULL | NULL | NULL | 5 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select * from staffs where name <> 'July';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ALL | idx_staffs_nameAgePos | NULL | NULL | NULL | 5 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

```

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';

## 12.1.7. 策略 7.Null/Not 有影响

注意 null/not null 对索引的可能影响

### 12.1.7.1. 自定义为 NOT NULL

```

mysql> 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';
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO staffs(NAME,age,pos,add_time) VALUES('z3',22,'manager',NOW());
Query OK, 1 row affected (0.01 sec)

mysql> ALTER TABLE staffs ADD INDEX idx_staffs_nameAgePos(NAME, age, pos);
Query OK, 0 rows affected (0.11 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> ALTER TABLE staffs ADD INDEX idx_staffs_name(NAME);
Query OK, 0 rows affected (0.01 sec)
Records: 0 Duplicates: 0 Warnings: 0

```

```

mysql> EXPLAIN SELECT * FROM staffs WHERE NAME IS NULL;
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | NULL | NULL | NULL | NULL | NULL | NULL | Impossible WHERE |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> EXPLAIN SELECT * FROM staffs WHERE NAME IS NOT NULL;
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ALL | idx_staffs_nameAgePos, idx_staffs_name | NULL | NULL | NULL | 1 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

```

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

### 12.1.7.2. 自定义为 NULL 或者不定义

```
mysql> CREATE TABLE staffs2 (
    ->     id INT PRIMARY KEY AUTO_INCREMENT,
    ->     NAME VARCHAR (24),  
          ↓
    ->     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';
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO staffs2(NAME,age,pos,add_time) VALUES('z3',22,'manager',NOW());
Query OK, 1 row affected (0.00 sec)

mysql> ALTER TABLE staffs2 ADD INDEX idx_staffs2_nameAgePos(NAME, age, pos);
Query OK, 0 rows affected (0.02 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> ALTER TABLE staffs2 ADD INDEX idx_staffs2_name(NAME);
Query OK, 0 rows affected (0.02 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql>
```

EXPLAIN select \* from staffs2 where name is null

EXPLAIN select * from staffs2 where name is null									
信息 结果1 概况 状态									
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	staffs2	ref	idx_staffs_nameAg... idx_staffs_nameAgePos	idx_staffs_nameAgePos	75	const	1	Using index

EXPLAIN select \* from staffs2 where name is not null

EXPLAIN select * from staffs2 where name is not null						
信息 结果1 概况 状态						
select_type	table	type	possible_keys	key	key_len	ref
SIMPLE	staffs2	ALL	idx_staffs_nameAg... (Null)	(Null)	(Null)	(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

### 12.1.8. 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

```

mysql> explain select * from staffs where name='July';
+----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref |
+----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ref | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 74 | const | 1 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select * from staffs where name like '%July%';
+----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref |
+----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ALL | NULL | NULL | NULL | NULL | 5 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select * from staffs where name like 'July%';
+----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref |
+----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ALL | NULL | NULL | NULL | NULL | 5 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select * from staffs where name like '%July';
+----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref |
+----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | range | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 74 | NULL | 1 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

```

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%'

## 12.1.9. 策略 9. 字符类型加引号

字符串不加单引号索引失效

```

mysql> explain select * from staffs where name=917;
+----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref |
+----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ALL | idx_staffs_nameAgePos | NULL | NULL | NULL | 5 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

```

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

### 12.1.10. 策略 10.OR 改 UNION 效率高

```
mysql> explain select * from staffs where name='July' or name='z3';
+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ALL | idx_staffs_nameAgePos | NULL | NULL | NULL | 5 | Using where |
+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from staffs where name='July' or name='z3';
+-----+-----+-----+-----+
| id | NAME | age | pos | add_time |
+-----+-----+-----+-----+
| 1 | z3 | 21 | manager | 2016-02-14 23:01:33 |
| 5 | July | 24 | dev | 2016-02-14 23:01:35 |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

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'
```

### 12.1.11. 测试题

假设index(a,b,c)

Where语句	索引是否被使用
where a = 3	Y
where a = 3 and b = 5	Y
where a = 3 and b = 5 and c = 4	Y
where b = 3 或者 where b = 3 and c = 4 或者 where c = 4	N
where a = 3 and c = 5	Y
where a = 3 and b > 4 and c = 5	Y
where a = 3 and b like 'kk%' and c = 4	Y
where a = 3 and b like '%kk' and c = 4	Y
where a = 3 and b like '%kk%' and c = 4	Y
where a = 3 and b like 'k%kk%' and c = 4	Y

答案：

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	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 = 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

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。

## 12.2. 批量导入



The screenshot shows a portion of a Maven project's `pom.xml` file. It includes the following XML code:

```
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
    </dependency>
</dependencies>
```

### 12.2.1. 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

### 12.2.2. LOAD DATA INFILE

LOAD DATA INFILE;

使用 LOAD DATA INFILE ,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
```

```
load data INFILE '/soft/product3.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。

```
secure_file_priv=""
```

```
.
```