MYSQL性能优化概要

MYSQL性能优化概要

数据表结构相关优化

一、建字段类型注意事项

1. int类型的选择
2. varchar、char、text类型
3. date、datetime、timestamp类型

二、表规划

1. 垂直分表
2. 水平分表

查询语句优化

1. 组合索引
2. 避免使用操作符
3. 避免使用null做条件
4. like查询如何优化
5. 在查询子句中避免使用函数操作
6. 尽量避免在where子句中使用or
7. 尽量避免在where子句中使用表达式
8. 有时使用exists代替in
9. 尽量避免在大量重复数据的字段建立索引
10. 索引也是要适可而止
11. 尽量避免使用临时表(create temporary table…select…)
12. 尽量避免使用order by rand()
13. 拆分大的 DELETE、INSERT、update语句
14. 尽量避免客户端返回大数据量
15. 避免使用大事务

数据库配置优化

1. innodb_file_per_table表空间优化
2. innodb_flush_log_at_trx_commit刷新redo日志配置优化
3. innodb_buffer_pool_size缓冲大小优化
4. max_connections数据库最大连接数
5. innodb_log_file_size redo日志大小优化
6. session级内存分配优化
7. 线程池设置
8. CPU相关
9. IO相关的
10. skip_name_resolve使用

数据库的问题如何分析

1. 开启慢查询日志
2. 如何查数据库的处理情况
   - show full processlist
   - show profile
3. explain的使用
   - id
   - select_type
   - table
   - type
   - possible_keys
   - key
   - key_len
   - Extra

4. 如何正确使用索引(待续)
5. percona-toolkit工具的使用(待续)

数据表结构相关优化

一、建字段类型注意事项

1. int类型的选择

字段的容量存储尽可能小，这样子可以减少磁盘IO的开销，以提高数据表读写性能，提高索引的利用率。
在数据类型中，整型类型占用空间比较小是占4个字节，所以在理想状态下用整型来作为表字段类型是表结构最优的。在使用整型的时候也是有区别的，遵循一个原则，int类型也是越短越好。

1.像ip可以用int类型来代替varchar类型，可以用PHP函数ip2long可以把字符串转化成int类型。
2.像金额可以int类型来代替float,decimal,以分为单位来存储金额。
3.像状态值可以用tinyint类型来存储。
4.像订单号可以用bigint来存储

下面是int类型的存储结构图：

类型	含义说明(带符号的)
tinyint	1字节，范围（-128~127）
smallint	2字节，范围（-32768~32767）
mediumint	3字节，范围（-8388608~8388607）
smallint	3字节，范围（-8388608~8388607）
bigint	8字节，范围（+-9.22*10的18次方）

2.varchar、char、text类型

在我们建表的时候，字符串类型使用频率最高的。主要是有varchar,char,text3种类型组成。按照查询速度：char的速度最快，varchar次之，text最慢。由于char类型的容量较小，并且是固定长度所以速度会比较快点。

1. char类型最大长度是255个字符，默认是占用255 * 2个字节，如果是utf-8方式那么最多存储，255 * 3个字节。像手机号码，身份证号，带字母的订单号可以用char类型存储。
2. varchar类型的长度是可变的，可以设置最大长度，大部分字符串类型都用它。注意现在mysql的版本对于utf-8字符也是按字符数来算，不是字节数。
3. text类型，尽量少用，在做表读写操作时，其操作较慢，一般是存储大容量的字段内容，比如文章。

下面是int类型的存储结构图：

类型	含义说明
char(n)	固定长度，最多255个字符

类型	含义说明
varchar(n)	可变长度，最多65535个字符
text	可变长度，最多65535个字符
mediumtext	可变长度，最多2的24次方-1个字符
longtext	可变长度，最多2的32次方-1个字符

3.date、datetime、timestamp类型

日期、时间类型非字符串类型，这个是程序员最容易犯的错误。建议我们在设计时间类型相关的字段时候，建议用日期类型。在网上说时间用int类型的时间戳要优于日期类型，这其实在性能上相差不大，因为时间戳的int长度也较大，占用的字节数和mysql相差不大。日期类型也是支持索引的。

1. 像每日统计的数据在日期上展示的时候可以用date类型
2. 像订单生成时间，支付时间，可以用datetime类型
3. 像日志的生成时间，订单最后更新时间可以用timestamp类型

下面是int类型的存储结构图：

类型	含义说明
date	3字节，日期，格式：2017-09-18
datetime	8字节，日期时间，格式：2014-09-18 08:42:30
timestamp	4字节，自动存储记录修改的时间

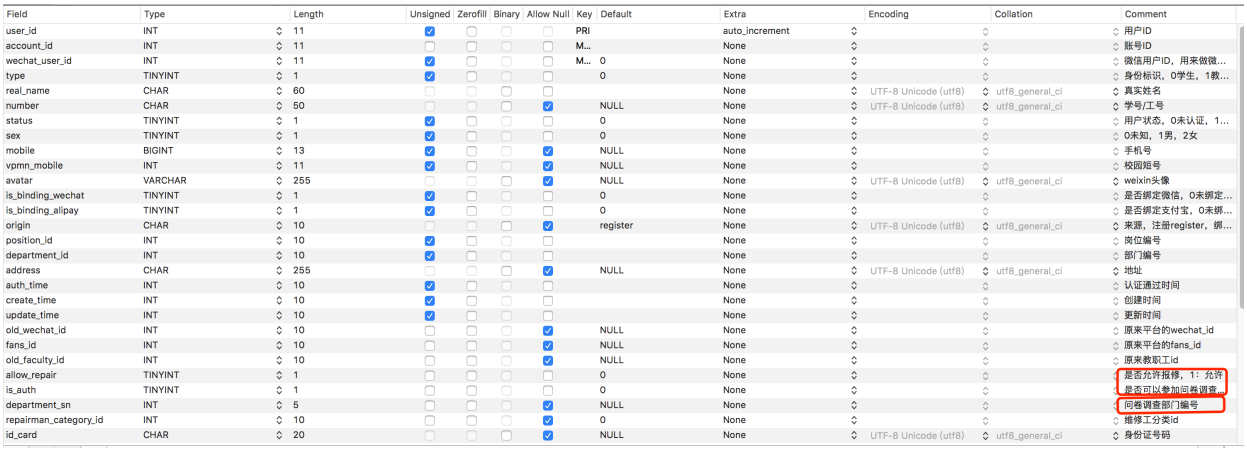
> 补充
>
> 1. 在表中添加字段时候，最好要not null；主要是null会增大表的占用容量，最主要的是带有null值的字段，加索引失效。
> 2. 对于大容量表，不能取操作修改表，像加字段，添加索引等，一定要在业务空档期的时候加。
> 3. 字段容量计算公式，比如int类型占4个字节，1字节=8位，4个字节就是32位，那么长度是从0开始到31，即最大值如果不带符号，2^0~2^31位，如果是带符号的是 -2^31~2^31。

二、表规划

1. 垂直分表

随着需求量的增大，我们表字段也会越来越多，性能也会越来越慢。反面教材就是我们的gc_user表。
这个时候我们就需要拆表，垂直分表的原则是，动静分离，就是将频繁读写的字段单独建成一个表，一些不怎么变动的字段再单独建成一个表。这样子可以提高数据库的性能。比如商品表，库存就应该单独建一个表，里面就存商品id，和对应的库存。在读、写操作的时候可以加快速度。

如下图，下面的某些字段不应该加在gc_user字段表中



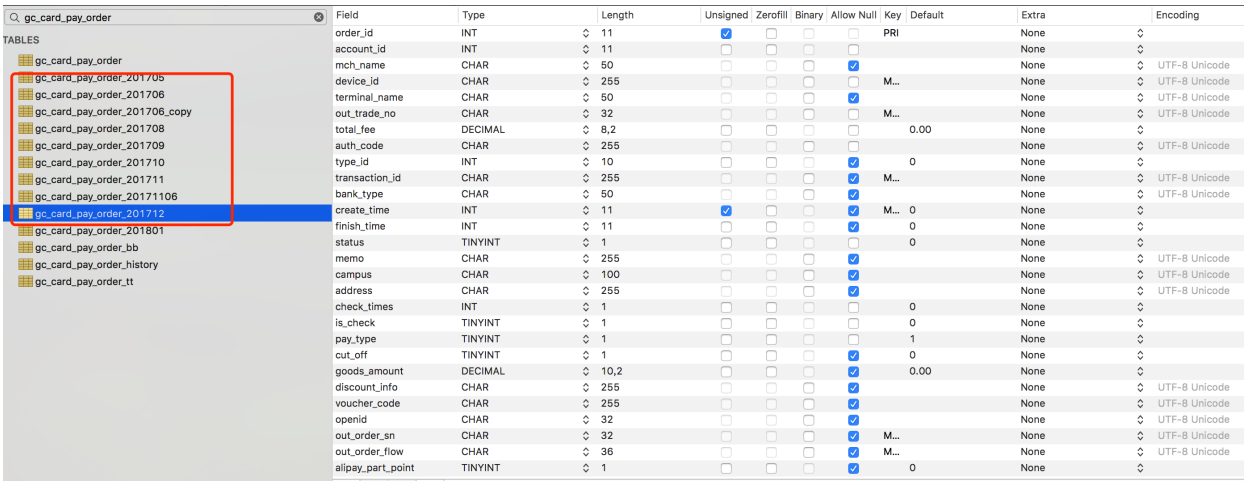
Field	Type	Length	Unsigned	Zerofill	Binary	Allow Null	Key	Default	Extra	Encoding	Collation	Comment
user_id	INT	11	☑	☐	☐	☐	PRI		auto_increment			用户ID
account_id	INT	11	☐	☐	☐	☐	M...		None			账号ID
wechat_user_id	INT	11	☑	☐	☐	☐	M...	0	None			微信用户ID，用来做微...
type	TINYINT	1	☑	☐	☐	☐		0	None			身份标识，0学生，1教...
real_name	CHAR	60	☐	☐	☐	☐			None	UTF-8 Unicode (utf8)	utf8_general_ci	真实姓名
number	CHAR	50	☐	☐	☐	☑		NULL	None	UTF-8 Unicode (utf8)	utf8_general_ci	学号/工号
status	TINYINT	1	☑	☐	☐	☐		0	None			用户状态，0未认证，1...
sex	TINYINT	1	☑	☐	☐	☐		0	None			0未知，1男，2女
mobile	BIGINT	13	☑	☐	☐	☑		NULL	None			手机号
vpmn_mobile	INT	11	☑	☐	☐	☑		NULL	None			校园短号
avatar	VARCHAR	255	☐	☐	☐	☑		NULL	None	UTF-8 Unicode (utf8)	utf8_general_ci	weixin头像
is_binding_wechat	TINYINT	1	☑	☐	☐	☐		0	None			是否绑定微信，0未绑定...
is_binding_alipay	TINYINT	1	☑	☐	☐	☐		0	None			是否绑定支付宝，0未绑...
origin	CHAR	10	☐	☐	☐	☑		register	None	UTF-8 Unicode (utf8)	utf8_general_ci	来源，注册register，绑...
position_id	INT	10	☑	☐	☐	☐			None			岗位编号
department_id	INT	10	☑	☐	☐	☐			None			部门编号
address	CHAR	255	☐	☐	☐	☑		NULL	None	UTF-8 Unicode (utf8)	utf8_general_ci	地址
auth_time	INT	10	☑	☐	☐	☐			None			认证通过时间
create_time	INT	10	☑	☐	☐	☐			None			创建时间
update_time	INT	10	☑	☐	☐	☐			None			更新时间
old_wechat_id	INT	10	☐	☐	☐	☑		NULL	None			原来平台的wechat_id
fans_id	INT	10	☐	☐	☐	☑		NULL	None			原来平台的fans_id
old_faculty_id	INT	10	☐	☐	☐	☑		NULL	None			原来教职工id
allow_repair	TINYINT	1	☐	☐	☐	☐		0	None			是否允许报修，1：允许
is_auth	TINYINT	1	☐	☐	☐	☐		0	None			是否可以参加问卷调查
department_sn	INT	5	☐	☐	☐	☑		NULL	None			问卷调查部门编号
repairman_category_id	INT	10	☐	☐	☐	☑		0	None			维修工分类id
id_card	CHAR	20	☐	☐	☐	☑		NULL	None	UTF-8 Unicode (utf8)	utf8_general_ci	身份证号码

2. 水平分表

随着业务亮越来越多，数据量也随之疯长，单表的磁盘IO开销加大，导致系统访问变慢，这个时候我们就需要来对数据表进行水平分表，也叫数据分片。分表的相关业务操作是我们数据库维护和操作中交为复杂的，对于单节点服务器的分表操作还算简单，但对于读写分离的多服务器在做分表之后，操作是比较复杂的，不管是运维还是程序的实现。

分表的好处就是，减少数据库的磁盘IO，加快数据表的查询性能。主要的案例就是我们的无卡支付订单表，gc_card_pay_order，以前在订餐高峰，系统会变得很卡，经常被客户投诉，优化之后，系统也变得稳定了。

如下图，就是水平分表后的gc_card_pay_order表



gc_card_pay_order

TABLES

- gc_card_pay_order
- gc_card_pay_order_201705
- gc_card_pay_order_201706
- gc_card_pay_order_201706_copy
- gc_card_pay_order_201708
- gc_card_pay_order_201709
- gc_card_pay_order_201710
- gc_card_pay_order_201711
- gc_card_pay_order_20171106
- gc_card_pay_order_201712
- gc_card_pay_order_201801
- gc_card_pay_order_bb
- gc_card_pay_order_history
- gc_card_pay_order_tt

Field	Type	Length	Unsigned	Zerofill	Binary	Allow Null	Key	Default	Extra	Encoding
order_id	INT	11	☑	☐	☐	☐	PRI		None	
account_id	INT	11	☐	☐	☐	☐			None	
mch_name	CHAR	50	☐	☐	☐	☑			None	UTF-8 Unicode
device_id	CHAR	255	☐	☐	☐	☐	M...		None	UTF-8 Unicode
terminal_name	CHAR	50	☐	☐	☐	☑			None	UTF-8 Unicode
out_trade_no	CHAR	32	☐	☐	☐	☐	M...		None	UTF-8 Unicode
total_fee	DECIMAL	8,2	☐	☐	☐	☐		0.00	None	
auth_code	CHAR	255	☐	☐	☐	☐			None	UTF-8 Unicode
type_id	INT	10	☐	☐	☐	☑		0	None	
transaction_id	CHAR	255	☐	☐	☐	☑	M...		None	UTF-8 Unicode
bank_type	CHAR	50	☐	☐	☐	☑			None	UTF-8 Unicode
create_time	INT	11	☑	☐	☐	☑	M...	0	None	
finish_time	INT	11	☐	☐	☐	☑		0	None	
status	TINYINT	1	☐	☐	☐	☐		0	None	
memo	CHAR	255	☐	☐	☐	☑			None	UTF-8 Unicode
campus	CHAR	100	☐	☐	☐	☑			None	UTF-8 Unicode
address	CHAR	255	☐	☐	☐	☑			None	UTF-8 Unicode
check_times	INT	1	☐	☐	☐	☐		0	None	
is_check	TINYINT	1	☐	☐	☐	☐		0	None	
pay_type	TINYINT	1	☐	☐	☐	☐		1	None	
cut_off	TINYINT	1	☐	☐	☐	☑		0	None	
goods_amount	DECIMAL	10,2	☐	☐	☐	☑		0.00	None	
discount_info	CHAR	255	☐	☐	☐	☑			None	UTF-8 Unicode
voucher_code	CHAR	255	☐	☐	☐	☑			None	UTF-8 Unicode
openid	CHAR	32	☐	☐	☐	☑			None	UTF-8 Unicode
out_order_sn	CHAR	32	☐	☐	☐	☑	M...		None	UTF-8 Unicode
out_order_flow	CHAR	36	☐	☐	☐	☑	M...		None	UTF-8 Unicode
alipay_part_point	TINYINT	1	☐	☐	☐	☑		0	None	

> 补充
> 如果对于主键要求唯一的情况下，那么分表处理要求又提高了，每次添加数据的时候，主键一定要唯一。目前主流的有两种方式可以来处理，一种是利用redis计数，另外一种采用算法来计算。如果想看分布式ID生成方法，可以看下这篇文章：
> https://mp.weixin.qq.com/s/0H-GEXlFnM1z-THI8ZGV2Q

查询语句优化

1.组合索引

```
数据表查询，尽量采用单表查询，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引，如果同时需要在条件字段和order_by字段让索引都有效，那么就建立组合索引。
```

2.避免使用操作符

```
应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描，
select order_id,order_number,user_id where user_id <> 23
```

3.避免使用null做条件

```
应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：
select id from t where num is null
可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：
select id from t where num=0
```

4.like查询如何优化

```
对于like查询优化
对于like比较频繁的字段应该建立索引，并且在SQL写法上主要采用左前缀的优化原则
select * from t where order_title like '%abc%';
上面的SQL语句索引是失效的，应该改成这样子
select id from t where name like 'abc%';
```

5.在查询子句中避免使用函数操作

```
应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：
select id from t where datediff(day,createdate,'2005-11-30')=1
应该改成
select id from t where createdate>='2005-11-30' and createdate<'2005-12-1'
```

6.尽量避免在where子句中使用or

应尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如：

```
select id from t where num=10 or num=20
```

可以这样查询：

```
select id from t where num=10
union all
select id from t where num=20
```

7.尽量避免在where子句中使用表达式

应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：

```
select id from t where num/2=100
```

应改为：

```
select id from t where num=100*2
```

8.有时使用exists代替in

8. 很多时候用 exists 代替 in 是一个好的选择：

```
select num from a where num in(select num from b)
```

用下面的语句替换：

```
select num from a where exists(select 1 from b where num=a.num)
```

9.尽量避免在大量重复数据的字段建立索引

并不是所有索引对查询都有效，SQL是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL查询可能不会去利用索引，如一表中有字段sex，status等，在此字段上建了索引也对查询效率起不了作用。

10.索引也是要适可而止

索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有必要。

11.尽量避免使用临时表(create temporary table...select...)

在做数据迁移或临时数据存储操作时，避免使用临时表，(create temporay table [临时表名] select [fields] from [原始表])

应该先建一张物理表，然后(insert into [物理表] select [fields] from [原始表])

12.尽量避免使用order by rand()

不要用随机排序，就是order by rand()
MySQL去执行RAND()函数（很耗CPU时间），而且这是为了每一行记录去记行，然后再对其排序。就算是你用了Limit 1也无济于事（因为要排序）

```
$r = mysql_query("SELECT username FROM user ORDER BY RAND() LIMIT 1");

// 这要会更好:
$res = mysql_query("SELECT count(*) FROM user");
$d = mysql_fetch_row($r);
$rand = mt_rand(0,$d[0] - 1);
$res = mysql_query("SELECT username FROM user LIMIT $rand, 1");
```

13.拆分大的 DELETE、INSERT、update语句

如果你需要在一个在线的网站上去执行一个大的 DELETE 或 INSERT 查询，你需要非常小心，要避免你的操作让你的整个网站停止相应。因为这两个操作是会锁表的，表一锁住了，别的操作都进不来了。
php-fpm会有很多的子进程或线程。所以，其工作起来相当有效率，而我们的服务器也不希望有太多的子进程，线程和数据库链接，这是极大的占服务器资源的事情，尤其是内存。
如果你把你的表锁上一段时间，比如30秒钟，那么对于一个有很高访问量的站点来说，这30秒所积累的访问进程/线程，数据库链接，打开的文件数，可能不仅仅会让你泊WEB服务Crash，还可能会让你的整台服务器马上掛了。
所以，如果你有一个大的处理，你定你一定把其拆分，使用 LIMIT 条件是一个好的方法。下面是一个示例:

```
while(1) {
    //每次只做1000条
    mysql_query("DELETE FROM logs WHERE log_date <= '2009-11-01' LIMIT 1000");
    if(mysql_affected_rows() == 0) {
        // 没得可删了，退出!
        break;
    }
    // 每次都要休息一会儿
    usleep(50000);
}
```

14.尽量避免客户端返回大数据量

尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

15.避免使用大事务

尽量避免大事务操作，如果可以，使用分布式事务，提高系统并发能力。

> 补充
> 用单表操作来代替 联表、子查询。像联表虽然比较方便，但是其内部算法两个foreach循环，比如A表有100条数据，B表有1000条数据，那么A表在连接B表的时候，其实100 * 1000的操作，其实可以采用现将100条数据查出来，作为查询1000条数据的条件。最后通过程序来进行组合

数据库配置优化

1. innodb_file_per_table表空间优化

```
的数据和索引存放在共享表空间里或者单独表空间里。我们的工作场景安装是默认设置了innodb_file_per_table = ON，这样也有助于工作中进行单独表空间的迁移工作。MySQL 5.6中，这个属性默认值是ON。
```

---

2. innodb_flush_log_at_trx_commit刷新redo日志配置优化

```
默认值为1，表示InnoDB完全支持ACID特性。当你的主要关注点是数据安全的时候这个值是最合适的，比如在一个主节点上。但是对于磁盘（读写）速度较慢的系统，它会带来很巨大的开销，因为每次将改变flush到redo日志都需要额外的fsyncs。
如果将它的值设置为2会导致不太可靠（unreliable）。因为提交的事务仅仅每秒才flush一次到redo日志，但对于一些场景是可以接受的，比如对于主节点的备份节点这个值是可以接受的。如果值为0速度就更快了，但在系统崩溃时可能丢失一些数据：只适用于备份节点。说到这个参数就一定会想到另一个sync_binlog。
```

---

3. innodb_buffer_pool_size缓冲大小优化

```
此参数主要用来缓冲行数据，索引缓冲，事务锁，自适应哈希等
这个参数应该是运维中必须关注的了。缓冲池是数据和索引缓存的地方，它属于MySQL的核心参数，默认为128MB，正常的情况下这个参数设置为物理内存的60%~70%。（不过我们的实例基本上都是多实例混部的，所以这个值还要根据业务规模来具体分析。）
```

---

4. max_connections数据库最大连接数

```
MySQL服务器默认连接数比较小，一般也就100来个最好把最大值设大一些。一般设置500~1000即可每一个链接都会占用一定的内存，所以这个参数也不是越大越好。有的人遇到too many connections会去增加这个参数的大小，但其实如果是业务量或者程序逻辑有问题或者sql写的不好，即使增大这个参数也无济于事，再次报错只是时间问题。在应用程序里使用连接池或者在MySQL里使用进程池有助于解决这一问题。
```

---

5. innodb_log_file_size redo日志大小优化

这是redo日志的大小。redo日志被用于确保写操作快速而可靠并且在崩溃时恢复。如果你知道你的应用程序需要频繁地写入数据并且你使用的是MySQL 5.6，那么你可以一开始就把它这是成4G。（具体大小还要根据自身业务进行适当调整）

6. session级内存分配优化

```
根据自己的业务场景进行优化
max_threads-- 当前活跃连接数
read_buffer_size-- 顺序读缓冲，提高顺序读效率
read_rnd_buffer_size-- 随机读缓冲，提高随机读效率
sort_buffer_size-- 排序缓冲，提高排序效率
join_buffer_size-- 表连接缓冲，提高表连接效率
binlog_cache_size-- 二进制日志缓冲，提高二进制日志写入效率ß
tmp_table_size-- 内存临时表，提高临时表存储效率
thread_stack-- 线程堆栈，暂时寄存SQL语句/存储过程
thread_cache_size-- 线程缓存，降低多次反复打开线程开销
net_buffer_length-- 线程持连接缓冲以及读取结果缓冲
bulk_insert_buffer_size-- MyISAM表批量写入数据缓冲
```

7. 线程池设置

针对innodb_write_io_threads 和 innodb_read_io_threads 的调优我们目前没有做，但我相信调整为8或者16，系统 I/O 性能会更好。还有，需要注意以下几点：任何一个调整，都要建立在数据的支撑和严谨的分析基础上，否则都是空谈； 这类调优是非常有意义的，是真正能带来价值的，所以需要多下功夫，并且尽可能地搞明白为什么要这么调整。

8. CPU相关

```
Innodb_thread_concurrency=0
Innodb_sync_spin_loops=288
table_definition_cache=2000
```

9. IO相关的

```
Innodb_flush_method 建议用O_DIRECT
Innodb_io_capacity 设置成磁盘支持最大IOPS
Innodb_wirte_io_threads=8
Innodb_read_io_threads=8
Innodb_purge_threads=1
Innodb的预读方面，如果基于主建或是唯一索引的系统，建议禁用预读
Innodb_random_read_ahead = off
```

10. skip_name_resolve使用

当客户端连接数据库服务器时，服务器会进行主机名解析，并且当DNS很慢时，建立连接也会很慢。因此建议在启动服务器时关闭skip_name_resolve选项而不进行DNS查找。唯一的局限是之后GRANT语句中只能使用IP地址了，因此在添加这项设置到一个已有系统中必须格外小心。

> 补充
> 数据库性能问题最主要的还是在SQL语句的性能上，数据库配置优化也只是辅助

数据库的问题如何分析

1. 开启慢查询日志

查看慢查询日志状态：

```
show variables like '%slow%';
```

结果：

Variable_name	Value	说明
log_slow_queries	ON	开启慢查询
slow_launch_time	2	代表着捕获所有执行时间超过2秒的查询
slow_query_log	ON	代表开启满查询日志写入
slow_query_log_file	/data/mysql-slow/slow.log	日志路径

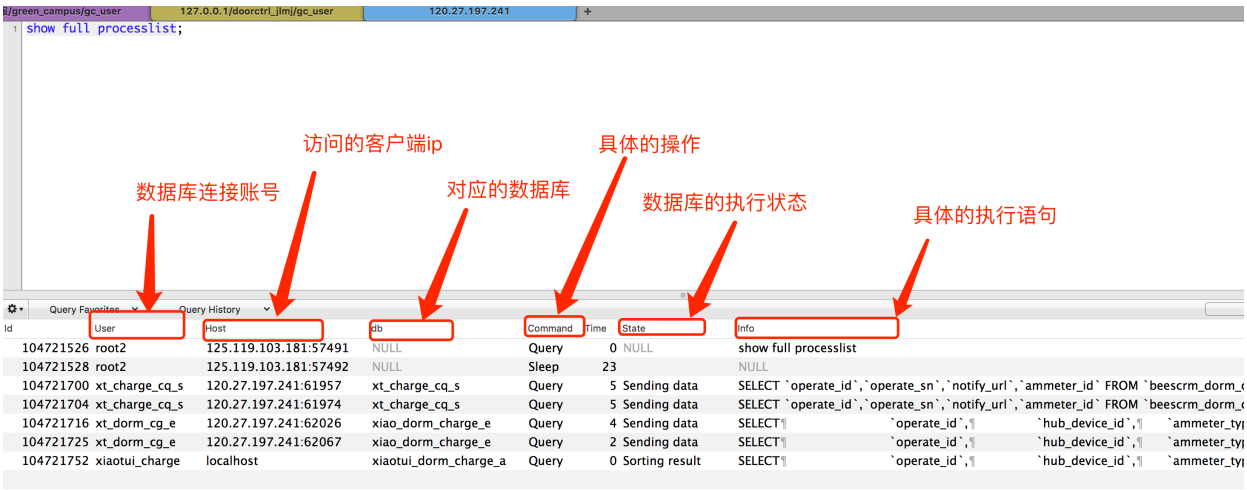
```
在my.cnf配置中将上述4个配置写入进去。

mysql慢查询日志分析工具之pt-query-digest
1.下载
wget http://www.percona.com/get/pt-query-digest

2.分析
./pt-query-digest 【慢查询日志路径】|  more

3.将慢查询日志生成报表
./pt-query-digest 【满查询日志路径】 > slow_report.log
```

---

2. 如何查数据库的处理情况

show full processlist

> 此命令比较好用，可以监控目前正在活动的数据库操作，主要看的是`state`和`info`两个信息，`state`表示这条SQL语句的问题。`info`表示具体的SQL语句



主要来说说state

字段名	说明
Checking table	正在检查数据表（这是自动的
Closing tables	正在将表中修改的数据刷新到磁盘中，同时正在关闭已经用完的表。这是一个很快的操作，如果不是这样的话，就应该确认磁盘空间是否已经满了或者磁盘是否正处于重负中
Sending data	正在处理SELECT查询的记录，同时正在把结果发送给客户端,如果一直处于这个状态就是索引失效或者字段太大的问题
Connect Out	复制从服务器正在连接主服务器
Copying to tmp table on disk	由于临时结果集大于tmp_table_size，正在将临时表从内存存储转为磁盘存储以此节省内存
Creating tmp table	正在创建临时表以存放部分查询结果
Flushing tables	正在执行FLUSH TABLES，等待其他线程关闭数据表
Killed	发送了一个kill请求给某线程，那么这个线程将会检查kill标志位，同时会放弃下一个kill请求。MySQL会在每次的主循环中检查kill标志位，不过有些情况下该线程可能会过一小段才能死掉。如果该线程程被其他线程锁住了，那么kill请求会在锁释放时马上生效
Locked	被其他查询锁住了

字段名	说明
Sorting for group	正在为GROUP BY做排序
Sorting for order	正在为ORDER BY做排序
Sleeping	正在等待客户端发送新请求
System lock	正在等待取得一个外部的系统锁。如果当前没有运行多个mysqld服务器同时请求同一个表，那么可以通过增加–skip-external-locking参数来禁止外部系统锁

show profile

此命令分析当前数据库登陆会话下的执行语句的资源消耗情况(注意只能是当前会话下)

```
1. 输出当前连接下所有执行中sql
show profiles   //会列出所有查询id，执行时间，执行语句
```

Query ID	Duration	Duration
1	0.05912400	select count(1) from user
2	5.82553775	select count(1) from order

```
2. 输出指定SQL语句的信息
show profile for query n
查看指定query_id的sql的执行消耗情况(默认输出Status和Duration columns两列信息)
show profile for query 2
```

Status	Duration
starting	0.000095
checking permissions	0.000011
Opening tables	0.017916
init	0.000028
System lock	0.000018
optimizing	0.000010
statistics	0.000021
preparing	0.000020
executing	0.000005
Sending data	5.794340
end	0.000025
query end	0.000014

Status	Duration
closing tables	0.000026
freeing items	0.000045
logging slow query	0.012937
cleaning up	0.000028

3. explain的使用

```
语法: EXPLAIN [EXTENDED] SELECT select_options
```

下面就是在查询分析器分析的一个例子

```
explain extended select * from gc_user;
```

Query Favorites Query History

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	gc_user	NULL	ALL	NULL	NULL	NULL	NULL	53858	100.00	NULL

如何分析SQL查询

下面所列就是explain返回各列的含义:

id

```
SELECT识别符。这是SELECT的查询序列号
```

select_type

SELECT类型,可以为以下任何一种:

SIMPLE:简单SELECT(不使用UNION或子查询)
PRIMARY: 最外面的SELECT
UNION: UNION中的第二个或后面的SELECT语句
DEPENDENT UNION: UNION中的第二个或后面的SELECT语句,取决于外面的查询
UNION RESULT: UNION 的结果
SUBQUERY: 子查询中的第一个SELECT
DEPENDENT SUBQUERY: 子查询中的第一个SELECT,取决于外面的查询
DERIVED: 导出表的SELECT(FROM子句的子查询)

table

输出的行所引用的表

type

显示连接用了哪种类型, 从最好到最坏的连接类型为:
联接类型。下面给出各种联接类型,按照从最佳类型到最坏类型进行排序:

system: 表仅有一行(=系统表)。这是const联接类型的一个特例。

const: 表最多有一个匹配行,它将在查询开始时被读取。因为仅有一行,在这行的列值可被优化器剩余部分认为是常数。const表很快,因为它们只读取一次!

eq_ref: 对于每个来自于前面的表的行组合,从该表中读取一行。这可能是最好的联接类型,除了const类型。

ref: 对于每个来自于前面的表的行组合,所有有匹配索引值的行将从这张表中读取。

ref_or_null: 该联接类型如同ref,但是添加了MySQL可以专门搜索包含NULL值的行。

index_merge: 该联接类型表示使用了索引合并优化方法。

unique_subquery: 该类型替换了下面形式的IN子查询的ref: value IN (SELECT primary_key FROM single_table WHERE some_expr) unique_subquery是一个索引查找函数,可以完全替换子查询,效率更高。

index_subquery: 该联接类型类似于unique_subquery。可以替换IN子查询,但只适合下列形式的子查询中的非唯一索引: value IN (SELECT key_column FROM single_table WHERE some_expr)

range: 只检索给定范围的行,使用一个索引来选择行。

index: 该联接类型与ALL相同,除了只有索引树被扫描。这通常比ALL快,因为索引文件通常比数据文件小。

ALL: 对于每个来自于先前的表的行组合,进行完整的表扫描。

possible_keys

```
显示可能应用在这张表中的索引，如果为空，则是没有可能的索引
```

key

```
实际使用到的索引。如果为NULL，则没有使用索引
```

key_len

```
使用索引的长度，在不损失精确性的情况下，索引越小越好
mysql以页为单位进行扫描，页越大，索引扫描就越慢
```

Extra

```
Using filesort:
看到这这个的时候，查询就需要优化了，它根据全部行的行指针来排序全部行

Using temporary
看到这个的时候，查询需要优化了，Mysql创建一个临时表来哦存储结果，这通常发生在对不同列级进行GROUP BY，ORDER BY 上。
```

4. 如何正确使用索引(待续)

有时间再补上，主要涉及如何让索引的效率最高、BTree索引背后的算法实现。

5. percona-toolkit工具的使用(待续)

是DBA的专用工具，主要涉及监控、日志分析类还需要学习和实践。